DESIGN DOCUMENT

OTHELLO PROGRAM

YOLAND S.M. NABABAN
yolandsmnababan@gmail.com

Table of Contents

1. INTRODUCTION

Reversi is an ancient game whose origin is uncertain. The oldest references about similar games go back to late in the 19th century; those games had different names and their boards had different size or shape. In 1870 appeared a similar game using a cross shape board. Subsequently appeared another game played on a 8x8 square shape board.

Two players take part in this game; they need a board with 64 squares distributed in 8 rows and 8 columns, and 64 similar pieces of two colors (normally black and white): the obverse in one color and the reversi in the other color.

The objective for each of the players is to finish the game with more pieces on the board in his own color than the opponent.

Before starting players decide which color will use each of them. Next 4 pieces have to be placed in the central squares of the board, so that each pair of pieces of the same color form a diagonal between them. The player with black pieces moves first; one only move is made every turn.

A move consists in placing from outside one piece on the board. Placed pieces can never be moved to another square later in the game. The incorporation of the pieces must be made according to the following rules:

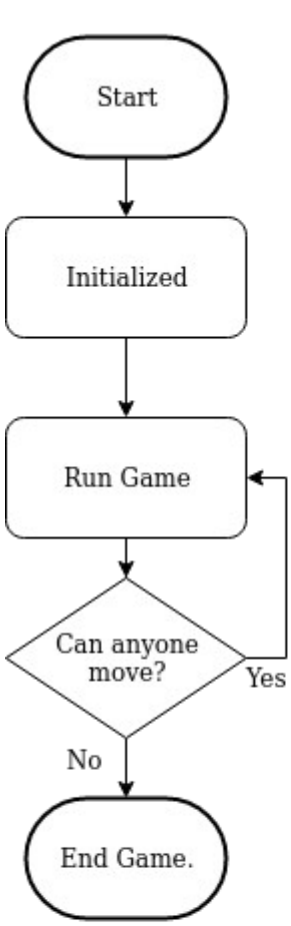
- The incorported piece must outflank one or more of the opponent placed pieces
- To outflank means that a single piece or one straight row (vertical, horizontal or diagonal) of pieces of the opponent is in both sides next to own pieces, with no empty squares between all those pieces
- The player who makes the move turns the outflanked pieces over, becoming all of them in own pieces
- If there is more than one outflanked row, all the involved pieces in those rows have to be flipped
- If it´s not possible to make this kind of move, turn is forfeited and the opponent repeats another move

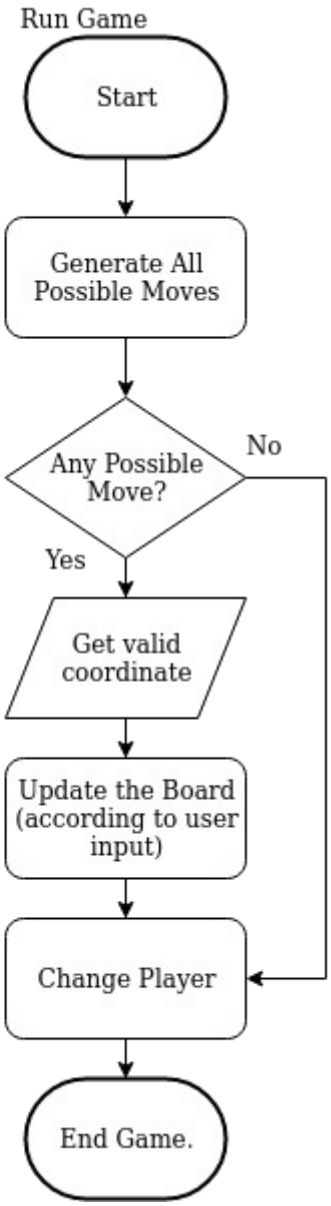
The game is over when all the squares of the board are taken or none of the players can move. In any case the winner is the player who has more pieces on the board. The game ends in a draw when both players have the same number of pieces on the board.

http://www.ludoteka.com/reversi-en.html
http://samsoft.org.uk/reversi/strategy.htm

2. ALGORITHM DESIGN

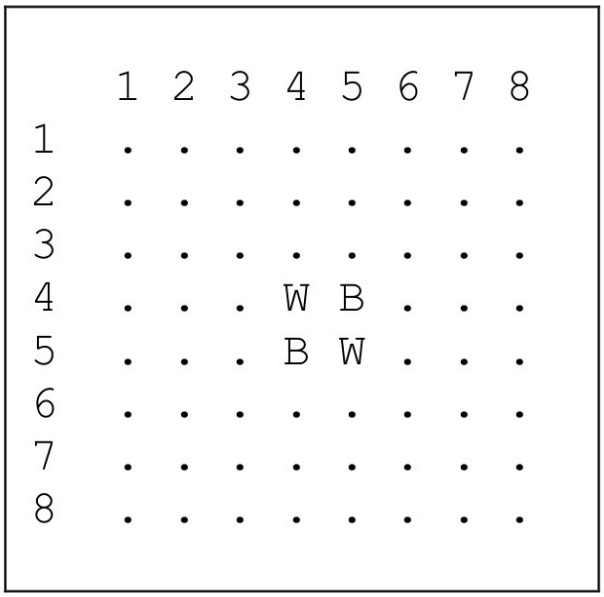
The picture above is a flowchart of an algorithm design to create the program:





Explanation of the algorithm design:

1. Initialize the game using enumeration for each cell. Every cell have 4 possibilities such as, WHITE, BLACK, EMPTY, and POSSIBLE (area the can be selected by a particular player).The display will be formed using 8x8 2-dimensional array.



a) Example of a 2-dimensional array.
BOARD[Y][X]
Y: top to bottom numbering (1-8)
X: left to right numbering (1-8)

b) Coordinates is represented by a number 'yx' for y and x axis, e.g. 45 for y = 4 and x = 5.

c) It will be initialized with WHITE Cells at 44 and 55, BLACK Cells at 45 and 54, while the rest is filled with EMPTY values.

2. The game will be started by a black player.

3. The detail of the game explained below
   a) The game starts by determining who is playing.

   b) Then the board will display an area that can be selected by the player. The procedure is to check each player's color on the board and then produce coordinate that might be chosen from the origin coordinate. Do the procedure for each point around the origin coordinate. That point will be changed from EMPTY to POSSIBLE. The POSSIBLE point must be equipped with the direction from which the point originated. The aim is to determine which direction the POSSIBLE coordinate is formed. If 2 points lead to the same coordinates, then the direction must be saved so that when flipping the cell, the corresponding direction can be replaced.

Method to generate all POSSIBLE coordinates.

```
public void generateAllMove(CellColor color){
    for (int i=offset; i<=ySize; i++){
        for (int j=offset; j<=xSize; j++){
            Point p = new Point(i,j);
            if (getCellColor(p) != color)
                continue;
            //generate possible cell for each direction
            for (Direction dir : Direction.values())
                generateDirMove(dir, color, p);
        }
```

```
        }
        System.out.println(movesCount());
    }
```

Method to generate POSSIBLE coordinate in only one direction.

```
    private void generateDirMove(Direction dir,
                                 CellColor color,
                                 Point p){
        //get valid neighbor
        Point pNext = getNeighbor(dir, p);

        if (!isPosValid(pNext)) return;


        //neighbor cell must be an opposite of current cell
        if (!getCell(pNext).isOppositeOf(color)){
            return;
        }

        // find empty
        boolean found = false;
        do {
            pNext = getNeighbor(dir, pNext);

            if(!isPosValid(pNext)) break;
            if(getCellColor(pNext) == color) break;
            if(getCell(pNext).isEmpty()) found = true;

        }while (!found);

        // if found, add opposite direction to the cell
        // to be turn to the opposite color later
        if (found){
            setCellPossible(pNext);
            addCellDir(pNext, oppositeDir(dir));
        } else {
            return;
        }
    }
```

c) Check the number of POSSIBLE coordinate. If there isn't, then change the player and proceed to point 4. If there is a POSSIBLE coordinate, then take input from the user. The input process will be repeated if the coordinate is less than 2 characters or the coordinate is not valid. The coordinate is valid if the coordinate is the POSSIBLE point. If the selected point is valid, then reverse each cell from the POSSIBLE point to the corresponding color. This must be done in every possible direction (this can be seen from algorithm 3.b above.

d) The procedure ends by changing the player.

Method to update the board (ensure the coordinate is valid first).

```
public boolean updateBoard(Point p, CellColor c){
    if (c == CellColor.EMPTY) return false;

    if (!isPosValid(p)) return false;

    if (getCellColor(p) != CellColor.POSSIBLE) return false;

    flipAll(p, c);
    return true;
}
```

Procedure to flip the Cell

```
public void flipAll(Point p, CellColor c){
    // FLIP all possible direction
    while (getCell(p).getDirSize()>0){
       flipDir(p, c, getCellDir(p));
    }

    //restore to empty cell again
    System.out.println("here");
    for (int i=offset; i<=ySize; i++){
       for (int j=offset; j<=xSize; j++){
            Point tmp = new Point(i,j);
          if (getCellColor(tmp) == CellColor.POSSIBLE){
              restoreCell(tmp);
            }
       }
    }
}
```

4. Check whether there are still players who can move. If there is still a move, then return to point 3. If not, the game is over.

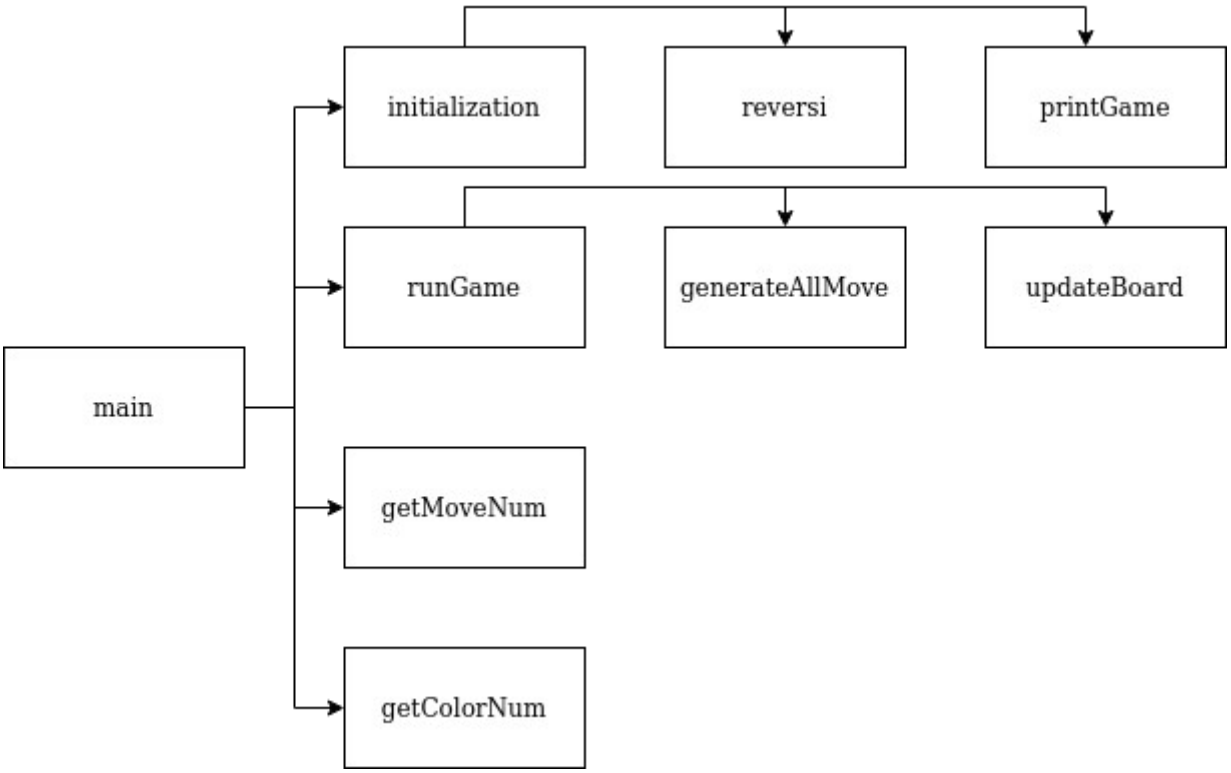
5. The game ends by counting the points of each player. The player who has the most points wins the game.

3. MODULE DESIGN

To run the algorithm, the program will be separated into several Classes. The main class for running the program is Class Reversi. This class will consist of several instances, namely Board objects, 2 Player objects (for Black and White), and the current player status object.

a) BOARD Class is an object that holds each cell. Therefore, this Class has 8x8 cells that carry certain colors. Therefore, this Class requires instances in the form of 2-dimensional arrays with CELL data types. CELL is a class that inherits every point. To access each cell, you can use coordinates. This coordinate is represented by using a Class named POINT Class. The goal is to be able to represent cell coordinates more easily. The CELL class is represented by 2 instances, namely COLOR (represent by enumeration) and POSSIBLEMOVE (represent by stack). COLOR will keep the color value in the corresponding cell. While POSSIBLEMOVE holds the enumeration Direction value. The DIRECTION is the direction of cells that can be flipped.

b) PLAYER class saves the status of the player. This class has several instances namely COLOR, NAME, MOVENUM, COLORNUM. COLOR is the color of the player. NAME is the name of the player. MOVENUM is the number of cells to choose from. while COLORNUM is the number of cells that have the same color as the player's color.



Below is some method that is used (mainly) in the program.

1. main

Purpose	To start the game, to initialize the game, to decide who is the winner.
Prototype	Public static void main (String args []) → Class Reversi
Input	-
Output	-
Calls	• Initialization • runGame • getMoveNum • getColorNum
Algorithm	• Initialize. • Run the game until no one can move. • Counting the number of each color. • Decide the winner.

2. Initialization

Purpose	Displays the game board when firstly started
Prototype	Static void initialization ()→ Class Reversi
Input	-
Output	-
Calls	• Reversi • printGame
Algorithm	• Construct CLASS REVERSI • Print Initial Board Game + info.

2.a. Reversi

Purpose	Instantiate every object and set the BLACK as the first player.
Prototype	Public Reversi ()→ Class Reversi
Input	-
Output	-
Calls	-
Algorithm	• Initialize class for the first and the second player. • Initialize the Board class. • Set whoIsPlaying as a BLACK player.

2.b PrintGame

Purpose	Displays the board on the terminal
Prototype	Static void printGame(String info)→ Class Reversi

Input	Info displayed on the screen (e.g., "Invalid Input!")
Output	-
Calls	• blackCount • whiteCount • printBoard • printStats
Algorithm	• Counting the number of white and black cells. • Display the board. • Show the status • Show the info.

3. runGame

Purpose	Run the game for one particular player. Before the procedure stops, change the player.
Prototype	Static void runGame(CellColor c)→ Class Reversi
Input	Enumeration represents the color of each cell. It depends on the current Player's color.
Output	-
Calls	• GetCurrentPlayer • generateAllMove • movesCount • printGame • updateBoard • changePlayer
Algorithm	• Determine who is playing now • Generate all possible moves from the player's perspective and print the board • If there is no possible move, change player and return. • If there is any possible move, get input from the user. • Flip the Cell according to the user's input. • Change the player

3.a. generateAllMove

Purpose	Update the board so it can display player's possible move(s).
Prototype	Public void generateAllMove(CellColor color)→ Class Board
Input	Enumeration represents the color of each cell. This procedure generates all possible moves.
Output	-
Calls	• GetCellColor • generateDirMove
Algorithm	• Iterate for each coordinate. • If the cell color matches the player color, then call generateDirMove. • GenerateDirMove is a procedure that checks possible moves for each direction (8 directions) from origin cell.

3.b. updateBoard

Purpose	To flip all possible cells according to the coordinate given by the user.
Prototype	Public Boolean updateBoard(Point p, CellColor c)→ Class Board
Input	Point is the coordinates given by the user. This point is not necessarily valid. CellColor c is the player's color.
Output	Return TRUE when the board updated successfully.
Calls	• IsPosValid • getCellColor • FlipAll
Algorithm	• If the Cell is EMPTY, return FALSE • If coordinate is not valid, return FALSE • If the Cell cannot be selected, return FALSE • Call FlipAll method to flip all the possible cells.

4. getMoveNum

Purpose	Get the number of the possible moves.
Prototype	Public int getMoveNum()→ Class Player
Input	-
Output	Numbers of the possible moves.
Calls	-
Algorithm	Return moveNum as the PLAYER`s instance.

5. getColorNum

Purpose	Get the number of Player’s point
Prototype	Public int getColorNum() → Class Player
Input	-
Output	The number of cell s that have the same color as the player’s color.
Calls	-
Algorithm	Return colorNum as the PLAYER`s instance.

Below are supporting methods for the method above

1. BlackCount

Purpose	Count number of Black Cell
Prototype	Public int BlackCount() → Class Board
Input	-
Output	Number of Black Cell
Calls	• getCellColor
Algorithm	• Iterate for all the cells and count the Black Cell

2. WhiteCount

Purpose	Count number of White Cell
Prototype	Public int WhiteCount()→ Class Board
Input	-
Output	Number of White Cell
Calls	• getCellColor
Algorithm	• Iterate for all the cells and count the White Cell

3. PrintBoard

Purpose	Print the 8x8 Cell
Prototype	Public int printBoard() → Class Board
Input	-
Output	-
Calls	• printColor()
Algorithm	• Iterate for all the cells and print the color of each cell.

4. printStats

Purpose	To print the point of every player
Prototype	Static void printStats() → Class Board
Input	-
Output	Number of White Cell
Calls	• getName() • getColorNum() • getColor()
Algorithm	• Print the color and number of the color on the board

5. getCurrentPlayer

Purpose	Get the player who is playing
Prototype	Static Player getCurrentPlayer(CellColor c) → Class Reversi
Input	Color of current player
Output	Return the Objeck Player
Calls	• getCellColor
Algorithm	• Return the player according to the color.

6. movesCount

Purpose	Count number of possible moves
Prototype	Public int movesCount() → Class Board
Input	-

Output	Number of Possible moves
Calls	• getCellColor
Algorithm	• Iterate for all the cell and count the POSSIBLE Cell

7. ChangePlayer

Purpose	Change the player
Prototype	Static void changePlayer() → Class Reversi
Input	-
Output	-
Calls	• -
Algorithm	• If the current player is WHITE, return the BLACK and vice versa.

8. GetCellColor

Purpose	Get the color of a cell in a board
Prototype	Public CellColor getCellColor (Point p) → Class Board
Input	Point p, as the coordinate of a cell.
Output	Return the Color of the current Cell
Calls	• getColor
Algorithm	• Access the cell object and get the color.

9. GenerateDirMove

Purpose	Generate all possible moves in one direction, by searching the empty/possible space in a direction
Prototype	Private void generateDirMove(Direction dir, CellColor c, Point p) → Class Board
Input	The input is a combination of what color is the Cell, coordinate of the cell, and in what direction (direction is an enumeration, consist of 8 direction)
Output	-
Calls	• GetNeighbor • isPosValid • isOppositeOf • getCellColor • isEmpty • setCellPossible • oppositeDir • addCellDir
Algorithm	• Get the coordinate of cell neighbor (according to direction) • If the neighbor has the same color of the current color, end search. • If it doesn’t, search the neighbor again until out of range or opposite color is found • When found, set that cell as POSSIBLE and add the direction to stack (because the direction will be used when the user picks the Cell to be flipped.

10. isPosValid

Purpose	Check whether the coordinate is valid or not
Prototype	Private boolean isPosValid(Point p) → Class Board
Input	The coordinate (POINT) to be checked.
Output	True if the point is valid
Calls	• -
Algorithm	• Check whether the POINT is within the allowed range. If not, return false.

11. FlipAll

Purpose	Flip all the opponent Cells
Prototype	Public void flipAll(Point p, CellColor c) → Class Board
Input	The starting point and the current color.
Output	-
Calls	• GetDirSize • flipDir • getCellDir • getCellColor • restoreCell
Algorithm	• Flip the board starting from the current point to all direction • Reset the board so that every cell is not POSSIBLE again.

12. getColor

Purpose	Return the Cell`s color
Prototype	Public CellColor getColor() → Class Cell
Input	-
Output	Color of the Cell
Calls	• -
Algorithm	• Access the object instance (color).

13. getName

Purpose	Get the Player`s name
Prototype	Public String getName() → Class Player
Input	-
Output	The String of Player`s name
Calls	• -
Algorithm	• Access the object instance (name).

14. printColor

Purpose	Print the character of the cell into the terminal.

Prototype	Public char printColor() → Class Cell
Input	-
Output	Character representation of the Color
Calls	• -
Algorithm	• Return a suitable character according to the color

15. GetNeighbor

Purpose	Get the Coordinate of Neighbor
Prototype	Private Point getNeighbor (Direction dir, Point p) → Class Board
Input	Current coordinate and the direction of the neighbor
Output	Point of the Neighbor
Calls	• -
Algorithm	• Get the coordinate of neighbor according to the direction

16. isOppositeOf

Purpose	To check whether the colors are opposite
Prototype	Public Boolean isOppositeOf (CellColor c) → Class Cell
Input	The color to be compared
Output	Opposite or not (boolean)
Calls	• -
Algorithm	• If the current cell has the same color with the input, return false, else return true.

17. isEmpty

Purpose	Check whether a cell is empty or not
Prototype	Public boolean IsEmpty() → Class Cell
Input	-
Output	Boolean
Calls	• -
Algorithm	• Compare the current color

18. isCellPossible

Purpose	To check whether a cell is possible to fill with a color
Prototype	Private Boolean isCellPossible (Point p) → Class Board
Input	Point of the current cell
Output	Boolean
Calls	• getCellColor
Algorithm	• Compare the current Color

19. oppositeDir

Purpose	To get the opposite of a direction
Prototype	Public Direction oppositeDir (Direction dir) → Class Board
Input	A direction (enumeration)
Output	The opposite direction (e.g. input = UP, output = DOWN)
Calls	• -
Algorithm	• Switch case

20. addCellDir

Purpose	Add current direction to stack
Prototype	Private void addCellDir(Point p, Direction dir) → Class Board
Input	Coordinate of a cell and the current direction
Output	-
Calls	• addDir
Algorithm	• Push the direction to stack

21. GetDirSize

Purpose	Get how many possible directions can be flipped
Prototype	Public Direction getDirSize() → Class Cell
Input	-
Output	Number of the element in the stack (stack of Direction)
Calls	• size()
Algorithm	• Get the stack size

22. flipDir

Purpose	Flip the board in a direction from a coordinate.
Prototype	Private void flipDir (Point p, CellColor c, Direction dir) →Class Board
Input	Point and the current color of the Cell; and the destination
Output	-
Calls	• GetNeighbor • isPosValid • flipCellColor
Algorithm	• Turn the current Cell according to the color • Iterate to all the neighbors and flip the color.

23. getCellDir

Purpose	Get a direction of a POSSIBLE cell
Prototype	Public Direction getCellDir (Point p) → Class Board
Input	Coordinate

Output	Direction
Calls	• getDir
Algorithm	• Pop the stack

24. restoreCell

Purpose	Change POSSIBLE to EMPTY cell
Prototype	Private void restoreCell(Point p) → Class Board
Input	Coordinate
Output	-
Calls	• SetCellEmpty • getDirSize • getCellDir
Algorithm	• Set the cell to become EMPTY • remove all the stack of the cell

25. flipCellColor

Purpose	Turn the cell white or black
Prototype	Private void flipCellColor (Point p) →Class Board
Input	Coordinate
Output	-
Calls	• flipColor
Algorithm	• Turn white into black and vice versa.

26. addDir

Purpose	Add the direction to the stack
Prototype	Public void addDir (Direction d) → Class Cell
Input	Direction enum
Output	-
Calls	• -
Algorithm	• Push the direction to stack

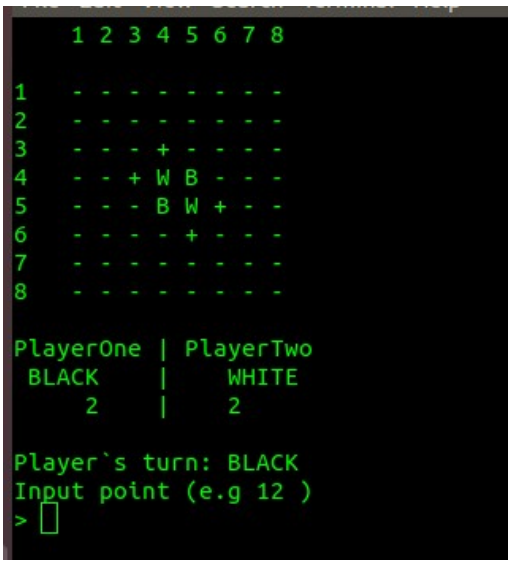
27. getDir

Purpose	Get a direction from the stack
Prototype	Public Direction getDir() → Class Cell
Input	-
Output	Direction enum
Calls	• -
Algorithm	• Pop the direction the stack

4. TEST RESULT

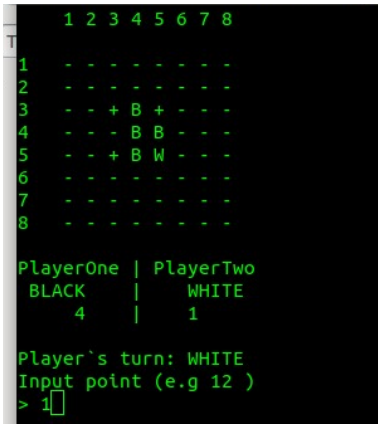
a) Enter

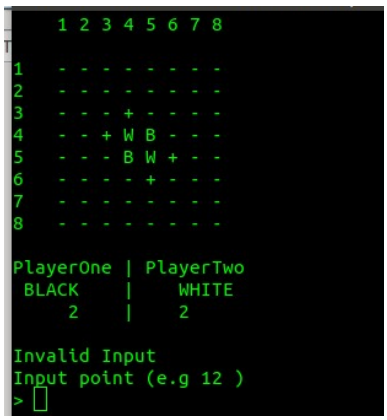
b) Start game (Black Player First)



c) Input out of range

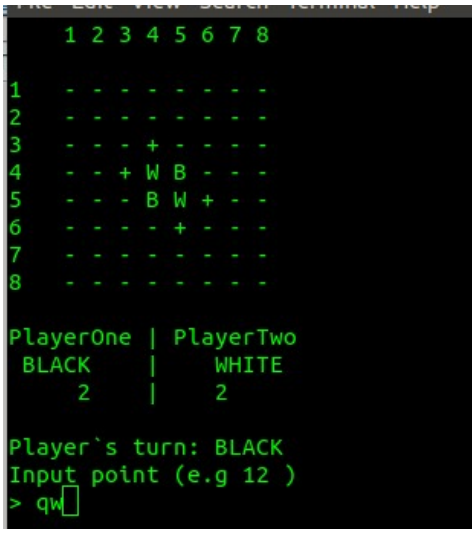
d) Input invalid

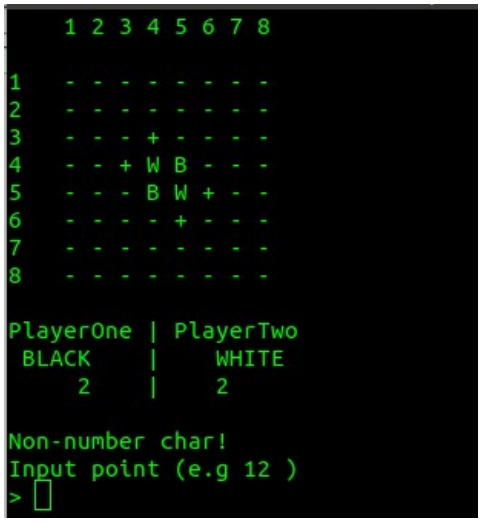




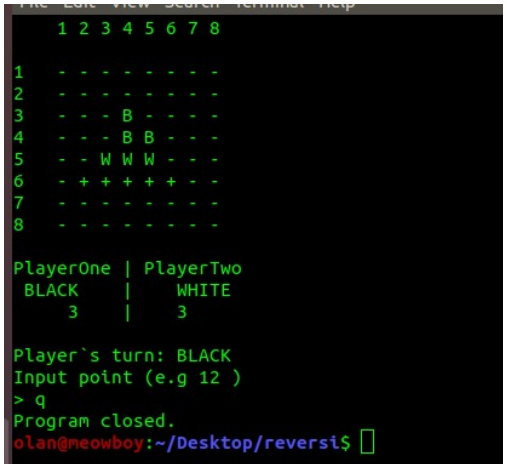
e) Coordinate is not available

f) Input is non-number

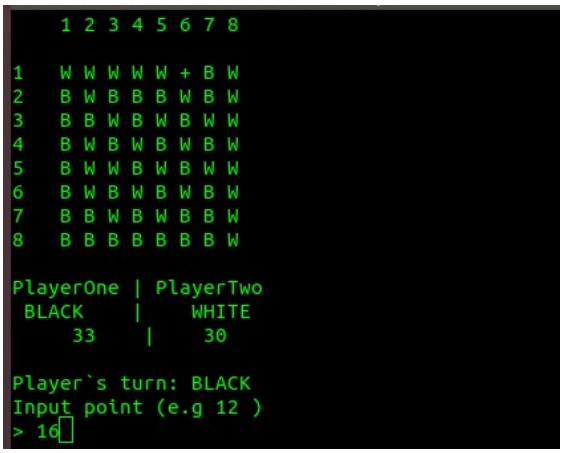




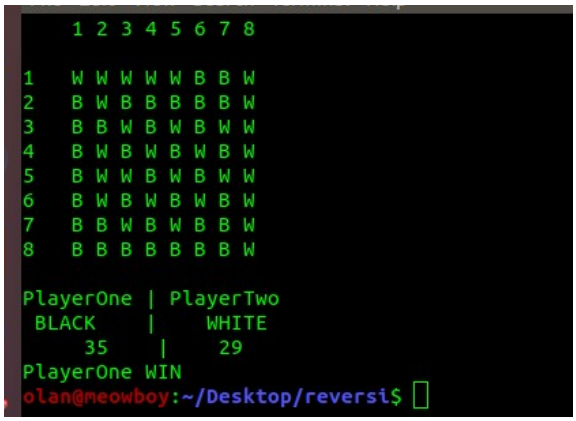
g) Close the program



h) Last step



i) End the game.



5. MANHOUR

Man-hour Estimation

Job Description	**Man-hour**
Implementation Design	6
Module Implementation	
1. main	2
2. initialization	1
3. reversi	1
4. printGame	1
5. runGame	4
6. generateAllMove	3
7. updateBoard	3
8. getMoveNum	1
9. getColorNum	1
10. Accessories module	4
Testing and debugging	3
Troubleshooting	3
Total	33