NORTHWEST
MISSOURI STATE UNIVERSITY

CSIS 44-542 Object-Oriented Programming
Spring 2017

Programming Assignment 3

This is an out-of-class programming assignment. It is to be done individually. The assignment is to be submitted as a single zip file via canvas by **5pm Friday, April 14, 2017**.

In this assignment we will develop a program to play the game Othello, also known as Reversi. We will not be using a graphical user interface, but rather a simple text-based solution that will permit us to focus our attention on the use of the appropriate Java constructs, and the algorithm to play the game.

To understand the game itself, we need to know {how} the game is played. This entails understanding the structure of the board, what the legal moves are, how the game is ended, what the rules for deciding the winner are, etc. We will proceed to define the game of Othello as carefully as we can.

Othello basics

The game is played between two players on a board consisting of 64 squares arranged in an 8×8 grid. The rows of the board are labelled "A" to "H", while the columns are labelled "1" to "8". We will refer to any square on the board by giving its row and its column name, for example: "F6". The board is populated by pieces and each square may hold at most one piece. Each piece is a flat disc coloured black on one side and white on the other. One player is called "black" and the other "white".

In our game we will use "O" to represent the white side of the disc (the human player) and "X" to represent the black side (the computer). This is a small simplification, since we can only display characters on our screen and type character has no "black disc" or "white disc" characters (without resorting to the use of graphics). The initial state of the game, using our notation, is shown in Figure 1 with 2 black ("X"s) and 2 white ("O"s) pieces arranged in the centre of the board.

With respect to a particular square, we define the term "*bar of the square*" (or just "*bar*") to refer to any row, column or diagonal (of the board) that contains the square. In addition, we will use the term "player" to refer to either the human user of our program or the program playing for the computer.

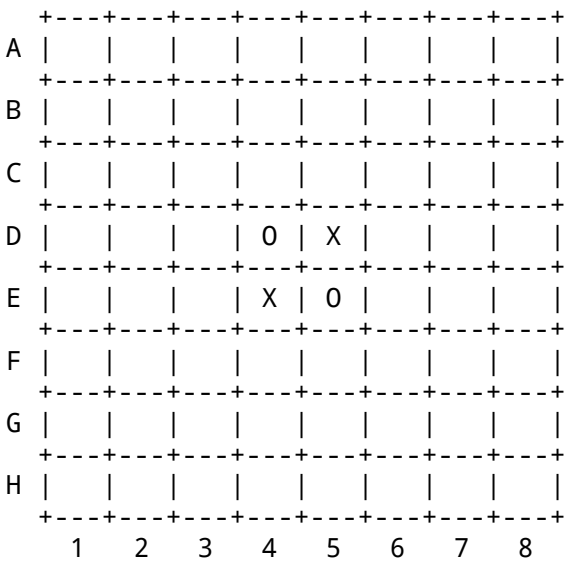

Figure 1. Initial starting setup of the otherllo board.

Move disciplinen and game termination

The players make their moves in turns. Each player makes one move and then waits for the opponent to make a move. A player who is unable to place a piece on the board must pass (forfeit) a turn. If a player can possibly place a piece on the board, then he must do so. The game terminates when neither player can put a piece on the board, either because the board is full, or because they both must pass. The winner is the player who has the largest number of pieces on the board when the game terminates.

Rules for making moves

A legal move is made by placing a single piece (of the player's colour) on a legal square on the board. The set of legal squares for each move is defined by the following two constraints:

1. The candidate square must be empty.
2. There must be at least one bar of the candidate square which contains one of the player's pieces (an *anchor piece*) such that:
   - the set of squares on the bar between the candidate square and the anchor piece is non-empty, and
   - is entirely filled with the opponent's pieces.

If there is no legal move, the player must give up the move (i.e. forfeit or pass a turn).

Effects of a move

The effect of the legal move is that the new piece appears on the board and all the pieces between the new piece and the set of all the {anchor pieces} are replaced by pieces of the player whose turn it is (in the physical game, the discs are flipped over). As an example, consider the board in Figure 2.

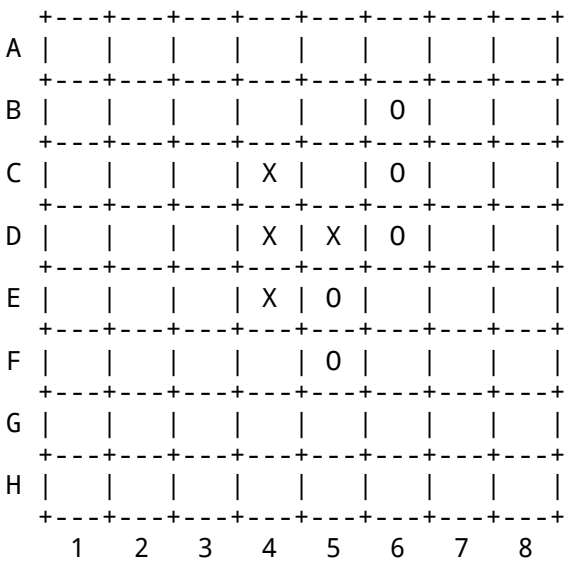

Figure 2. A typicl board position.

Assume that it is the user's turn to move. The user places an "O" at the position D3. The set of anchor squares is then [D6, F5]. The pieces on the row between D3 and D6 are *flipped*, that is to say they are replaced with the player's pieces ("O"). In addition, the piece between D3 and F5 (on the diagonal line) is also replaced with the player's piece. The result is shown in Figure 3.

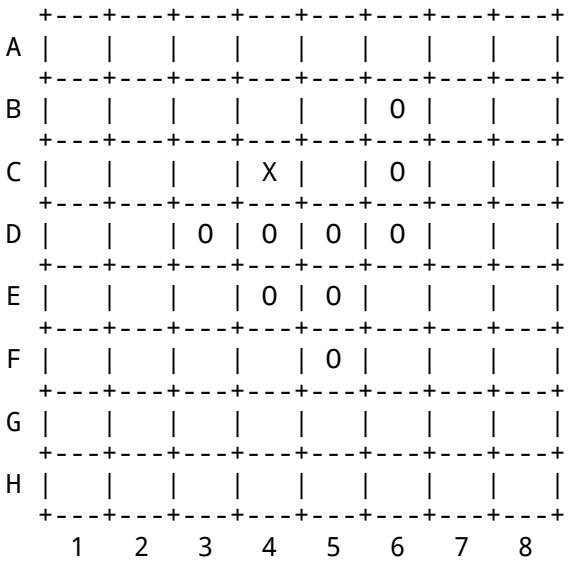

Figure 3. Result of the move to D3.

At this point, it would be useful to go away and play several games of Othello so as to get a "feel" for the game. Since this is difficult to do in a handout, please visit the website http://www.othelloonline.org/ to play an on-line graphical version of the game. As we think about how we might write the code to implement the game, let us just make a cursory survey of some winning strategies.

Strategies
It is clear that we should try to {\em avoid} creating a situation in which we have a long bar filled with {our} pieces and one of the opponent's pieces at the other end. If this situation arises, our opponent can simply put his piece at the end of the bar and flip all of our pieces! This is obviously going to be a key part of the strategy.

How might this situation be avoided? One place where this cannot happen is in the corners. If we have one of *our* pieces in the corner, there is no free square for the opponent to place his marker in! Clearly, control of the corner positions is important. Taking this line of thought one step further, we can see that taking a square {next} to the corner square (while the corner is empty) is highly undesirable! In fact, that is *exactly* what we would want an opponent to do. This is illustrated in Figure 4.

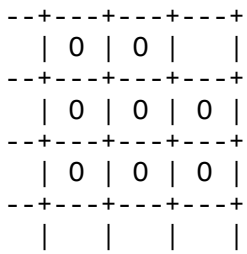

Figure 4 The valee of the corner square.

It is clear that when an “X” is played in the empty corner square, it is safe from “capture”, since there is no empty square next to it. It is equally clear that “X” stands to capture a substantial number of “O”s by making such a move!

From these simple observations, it is clear that there *is* some form of sensible strategy involved in the game, and we will take advantage of it when we implement the algorithm to determine the computer's move.

Designing a solution
In designing our solution to this problem, we will very carefully apply our stepwise refinement methodology. It is clear that our program will have to do many key things:

- display the state of the board after each move,
- input moves from the human player,
- generate moves for itself, and
- determine when the game is over.

These important actions will certainly be fundamental aspects of the solution. It is also clear that the single most important *data structure* in the solution will be the representation of the Othello board. Keeping all this in mind, let us proceed to design the algorithm. Our algorithm will first have to initialize all the necessary data structures (especially the board). t will then have to alternate the play between the human player and the computer displaying the board between each player's turn. A suitable starting point in developing the solution is presented in Figure 5.

1. Set up initial board configuration.
2. Display the board.
3. While the game is unfinished:
    a. Let appropriate player take turn.
    b. Display the board.
4. Determine who won!

Figure 5 Top level algorithm for Othello.

Now, Figure 5 hides one interesting aspect of the problem. If we pause for a moment to consider the various aspects of the problem, we can decompose the system in another way, based not on the hierarchical decomposition of the algorithm used to solve the problem, but based on the {\em nature} of parts of the algorithm. For instance, it is clear that there will be a collection of algorithms which pertain exclusively to the boar}. These will include initialization, display, representation, etc. There will also be a collection of algorithms that pertain to the game itself, such as algorithms to determine the winner, determine whose turn it is, etc. We could also recognize a third class of algorithms which pertain to the strategy we will be implementing so that the computer can make an "intelligent" move. It is very useful to recognize these modules early in the design phase, as it can help us quite substantially in actually implementing the algorithms as we refine them. We will adopt three categories for our algorithms: a module for the board and all the operations applicable to the board, a module for the strategy and a module for all the various control aspects of the game. The control module will form the basis of our program, and is described by the algorithm in Figure 5.

This assignment

The description above has outlines the game rules and behavior. Your assignment is to implement the solution. Some handy hints are provided below:

The board is an 8×8 grid that you may chose to represent as an array. Each square on the board (cell in the array) can take on one of three values: `empty`, `black` or `white`. As you think about the algorithms you need to check for valid moves and implement them, you may want to consider if a 10×10 array might be more helpful with the new value, `border`, being used also. This may help address potential problems of indexing the array outside of its legal bounds, but it will be dependent on choices you make later.

The human play has to be able to enter their move. They should only be allowed to do so if they can make a valide move. If they can not make a valid move then a message should be displayed to state that no valid move exists and the computer moves again. If a valid move exists, the human should be prompted for their move. If no valid move exists, a message telling the human that they mist pass is displayed. A move is entered in the form `CN` where C is a character from "A" to "H" (not case sensitive), and N is a number from 1 to 8. If an invalid move is entered, then an error message is displayed indicating it is invalid, reminding the user what is expected, and they are re-prompted for their move unti a valid move is provided. Note tha the human does not need to explicitly pass on their move as this will happen automatically for them if there is no valid move for them. When the computer moves, the square selected by the computer is displayed in a message to the user before the board is drawn and the human prompted for their move.

The board must be displayed after every move. You may chose to draw the board again if a player (human or copmuyter) must pass, but it is not necessary.

You will need a function to determine if a move is valid for a particular player. A move is only valid if there is one of more pieces that would be flipped if the players piece is placed on the candidate square.

You will need a routine to implement the effect of the more. This includes occupying the candidate square wioth the player's piece and flipping all of the opponenets pieces that are captured.

You will need an algorithm to determine the best place for the computer to move. Thre are many ways that this can be implemented. One approach is to weight each square with its strategic value. The value of a move to a candidate square is then the sum of the strategic values of squares occupied by each of the opponents pieces that would be flipped by moving to the candidate square. Therefore the candidate move that should be selected by the computer is the candidate square with the greatest staregic value. It remains to determine the strategic value of each location on the board. One possibility is whown in Figure 6 but you are free to experiment with the weightins and se if you can get the game to perform better.

```
  +---+---+---+---+---+---+---+---+
A |16 |-4 | 5 | 1 | 1 | 5 |-4 |16 |
  +---+---+---+---+---+---+---+---+
B |-4 |-12|-2 |-2 |-2 |-2 |-12|-4 |
  +---+---+---+---+---+---+---+---+
C | 5 |-2 | 4 | 2 | 2 | 4 |-2 | 5 |
  +---+---+---+---+---+---+---+---+
D | 1 |-2 | 2 | 1 | 1 | 2 |-2 | 1 |
  +---+---+---+---+---+---+---+---+
E | 1 |-2 | 2 | 1 | 1 | 2 |-2 | 1 |
  +---+---+---+---+---+---+---+---+
F | 5 |-2 | 4 | 2 | 2 | 4 |-2 | 5 |
  +---+---+---+---+---+---+---+---+
G |-4 |-12|-2 |-2 |-2 |-2 |-12|-4 |
  +---+---+---+---+---+---+---+---+
H |16 |-4 | 5 | 1 | 1 | 5 |-4 |16 |
  +---+---+---+---+---+---+---+---+
    1   2   3   4   5   6   7   8
```

Figure 6. Possible weightings showing the strategic value of each square.

For those looking for an extra challenge, you might consider dynamically modifying the strategic value of each square based on the squares currently occupied as the game progresses.

In this assignment you are asked to design and write the code to solve the problem (solutions sourced from the internet or other sources are *not* permitted). The code may consist of a number of files/classes – it is important to design the solution so that is is easy to maintain and eay to understand. The code is to be carefully tested and submitted via Canvas in a single zip file. The zip file is to include:

- The source files.
- A text file named "TestResults" which provides a description of the limitations or errors that remain in your solution. If there are still bugs in your submission then identify what they

are and describe what you think the underlying problem is and how you should proceed to fix it.

The source code is to be well written (follow a coding standard), well documented (use appropriate JavaDoc statements and comments), and use meaningful identifier names, All source code must begin with the following comments verifying that the source code is your own work:

```
/**
 * I certify that all code in this file is my own work.
 * This code is submitted as the solution to Assignment 1
 * in CSIS44542 Object-Oriented Programming, 2017, section <Your section number>
 *
 * Due date: 5pm, Friday, April 14, 2017.
 *
 * @author  <Your Name>
*/
```

Michael Oudshoorn
January 2017