A1-Designing Othello

Stuart Thiel

February 11, 2023

Introduction

This assignment is worth **5%** of your grade. This is an **individual assignment** (so individual, I put *your* student ID on it) and you should not share your assignment with anyone else. The **assignment is due February 24th at 11:59PM, Montreal time**.

All submissions must go through EAS: `https://fis.encs.concordia.ca/eas/`

After you submit, go back to check EAS to confirm that you just uploaded your assignment!

Q1) Designing Othello

Your task is to create design diagrams for a variation of the game Othello. The rules for the game can be found at the following URL:

`https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english`

The purpose of this assignment is to provide information that is relevant for creating a UML Class Diagram to assist in a subsequent implementation assignment. The accuracy and completeness of the diagrams are crucial for evaluation in this assignment. Clear diagrams and concise supplemental information to express design intentions will also be considered. Please note that there is no requirement for submitting a code implementation for this assignment. The subsequent assignment will involve implementing the design, which may differ from what you create in this assignment. The focus should be on the design described here, not the planned implementation.

In this variation of Othello, some positions on the board will be unplayable, and the starting position and the rules for flipping pieces may be different from the standard Othello game.

The program should display the game board using ASCII characters on the screen, with positional labels along the top and side to allow players to input their moves using keyboard commands. The program will provide messages indicating the outcome of moves and instructions for the current player's turn at the start of each turn. The traditional Othello disks are shown in the assignment description, but the actual implementation should display ASCII characters.

At the beginning of the game, a menu will be displayed with the following options:

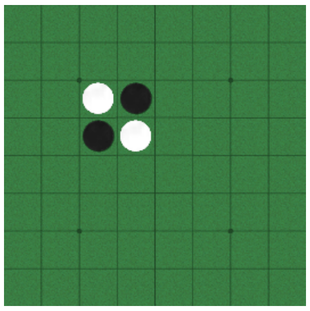
1. Quit
2. Load a Game
3. Start a New Game

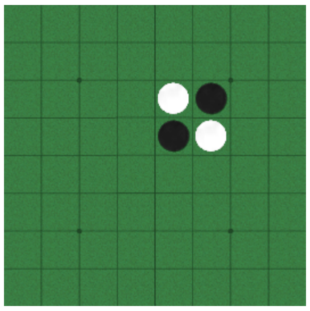
If a player chooses to quit, the program should close. If the player chooses to load a game, the program should prompt for a filename. After loading the game, play will continue from where it left off. The format for saving and loading games will be provided in the subsequent assignment. If the player chooses to start a new game, the program will ask for the names of two players, and the game will begin.

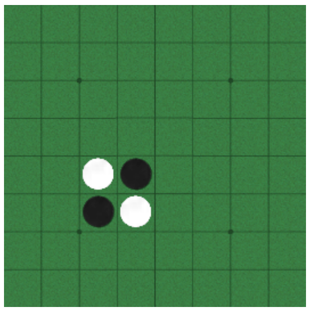
Please note that some squares on the board may be unplayable, and this rule is not part of the standard Othello game. However, the unplayable squares will remain the same in every game in the final implementation.

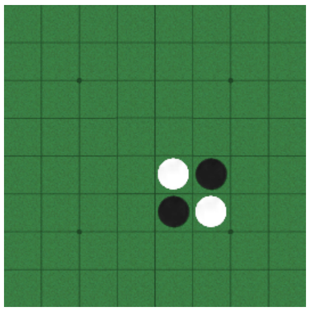
When starting a new game, the player will be given several options for the starting position, and they will select one by entering a number corresponding to the desired choice:

1. An offset starting position, the user will be subsequently asked to choose a number between 1 and 4 to indicate one of the following options


1) A non-standard, offset starting position.


2) A non-standard, offset starting position.


3) A non-standard, offset starting position.


4) A non-standard, offset starting position.

2. Four-by-Four Starting Position

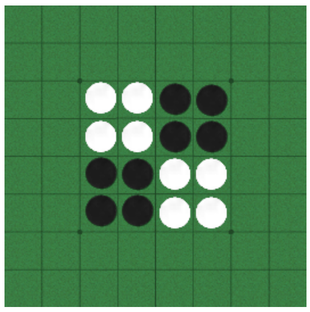


Figure 3: A non-standard, but still centered Othello starting position.

Regardless of whether the game is started from a new game or loaded from an existing game, the current player will have the following options:

- If the current player cannot move, they may choose to save, concede the game or, if they cannot make a play to take an opponent's pieces they may forfeit their turn
- A player may choose to save, concede or make a move

At the end of the game, the Board should record the outcome of the game, including the identity of the losing or whether the game ended in a tie. An appropriate message should be displayed.

Please create a UML class diagram that includes the classes Game, Board, Position, Piece, and Player. A Game object should have a Board object and Players, but it's not specified how they are related to each other. The Position class should be a parent class of UnplayablePosition. The Position class should have a virtual method named "canPlay()" that returns a boolean value indicating whether the position is playable or not. The return value should be ”true” for empty playable positions and ”false” for unplayable positions. The use of polymorphism is required and recommended in this design.

There is some flexibility in the design approach, but the play() method should loop, and the Board class should store its pieces internally. The design should include methods to save the game, a static method to load the game, a makeMove method to make a move, and methods to determine if a move converts any pieces. Additionally, a method to check if there are any valid moves left is also necessary. Consider the game rules and problem when including any additional methods, being mindful of which methods should be public or private.

The classes should have constructors to provide an overview of the life-cycle of the elements of the game or their construction.

UML Notes

Make use of UML Notes! It is acceptable to indicate in a UML Note that accessor/mutator methods are not shown or can be interpreted based on fields. Use UML Notes to clarify important decisions and provide additional information that is not well-communicated through the diagram itself. Generally, if a note would be helpful for someone trying to implement the game based on the design, it is probably a good note. The note should not describe implementation details but should instead explain how the elements in the diagram fit together clearly.

Breaking up Diagrams

In some cases, a UML diagram may become too complex or difficult to understand in its entirety. In these situations, it may be helpful to break the diagram into smaller, more manageable diagrams. This can make the information easier to understand and give a clearer picture of the relationships between the classes and their interactions.

When breaking down a UML diagram, it’s important to keep the relationships between classes consistent in each of the smaller diagrams. For example, if two classes have a relationship in one diagram, they must have the same relationship in another diagram, even if they are only represented as stubs.

Breaking down a UML diagram can also make it easier to focus on specific aspects of the design and make it easier to identify any potential issues. However, it’s important to balance the need for detail and simplicity, as too many smaller diagrams may become confusing and difficult to understand as well.

No Implementation

Please note that the subsequent assignment will be an implementation assignment based on a diagram that meets the requirements of a variation of this assignment. However, it may not correspond to your solution for this assignment as it will likely be the solution for someone else’s assignment.