
# OTHELLO GAME

FINAL REPORT PROJECT MODULE 2

Bui, D.C. (Duc Cuong, Student B-TCS)
s-29966174

# Contents

**I.Introduction** ..... 2
**II.System requirements** ..... 3
1.Logic game requirements: ..... 3
+/ Functional requirements: ..... 3
+/ Non – functional requirements: ..... 3
2.Client requirements ..... 3
+/ Functional requirements: ..... 3
+/ Non- functional requirements: ..... 3
**III.Package** ..... 4
**IV.Initial Design** ..... 5
**V.Overall testing Strategy:** ..... 8
**1.For game logic:** ..... 8
➢Coverage metrics ..... 8
➢Complexity metrics: ..... 9
**2.For whole system:** ..... 10
➢Coverage metrics: ..... 10
➢Complexity metrics: ..... 12
**VI.Test Plan:** ..... 14
**VII.Concurrency mechanism.** ..... 16
**VIII.Reflection on design** ..... 16
**Individual reflection:** ..... 24
**Reference:** ..... 25

# Othello - Programming Project

## I. Introduction

This report outlines the design and development of the Othello game software. The Othello program was written in Java and developed by student: Bui Duc Cuong – s29966174.

Othello is a two-player board game where the two players take turn placing discs of their color. Each move must capture at least one disc of the opponent. If this is not possible, the player must pass, unless the other player also cannot make such a move. The game ends when neither player can capture from their opponent. The winner is the player with the most discs. The game ends in a draw when both players have the same number of discs.
Othello is a variant of the game Reversi. [3]

Type: two-player strategy board game
Mode: with human or with computer
Victory: win if the board is full or the players do not have valid move, then person has more
pieces on the board who is a winner.
Move: mark on an empty field and is a valid move.

## II. System requirements

1. Logic game requirements:

+/ Functional requirements:

- Display of the board: The game should show an 8x8 board with 64 playable squares. There are two players in a game.
- Logic game methods do not crash when missing exceptions from incorrect usage.
- Placement of pieces: Players should be able to click on a square to put their pieces on the board.
- When a player does a piece, the game should flip any opponent pieces that are between the newly placed piece and another player's piece to the player's color.
- Win requirement: The player with the most pieces shall be declared the winner after counting the number of pieces of each color.
- Pass turn: If a player has no action to do, they should be able to skip their turn.

+/ Non – functional requirements:

- User interface: The game should have a simple, intuitive user interface with clear images and instructions.
- Performance: There should be less latency and the game should be quick and responsive.
- Compatibility: The game ought to work on a variety of platforms and gadgets.
- Accessibility: Players with disabilities should be able to access the game.

2. Client requirements

+/ Functional requirements:

- The client has a TUI for the user can follow and does actions.
- Users be able to connect using a port number and IP address.
- Users be able to login with the specific username.
- Client can play as a human player and controlled by user.
- Users can choose AI player to play instance of human player.
- Data retrieval: The user should be shown data that the client has obtained from a server.
- Submission of data: The client should enable user submission of data to the server.
- User interaction: The client must offer a user interface that enables users to interact with data and carry out operations like adding, removing, and updating information.
- Notification system: The user should get notifications from the client when specific events, take place.

+/ Non- functional requirements:

- Performance: the client should be fast and responsive with minimal lag.
- Scalability: the client should be able to handle increasing amounts of data and user traffic.

- Usability: the client should have a user-friendly interface and be easy to use.
- Compatibility: the client should be compatible with different platforms and devices.
- Accessibility: the client should be accessible to users with disabilities and provide options for font size and color contrast.

## III. Package

* We separated my programme into 2 big packages: src and test.

- In the src file, I have 2 files including: Client and Othello.

+ Client: Client (Class), ClientStatus (Interface) – contains the different status in client, ClientTUI – UI for user, Command (Interface) – contains commands to communicate between server and client, GameOver (Interface) – contains commands for gameover status and using to communicate with server.

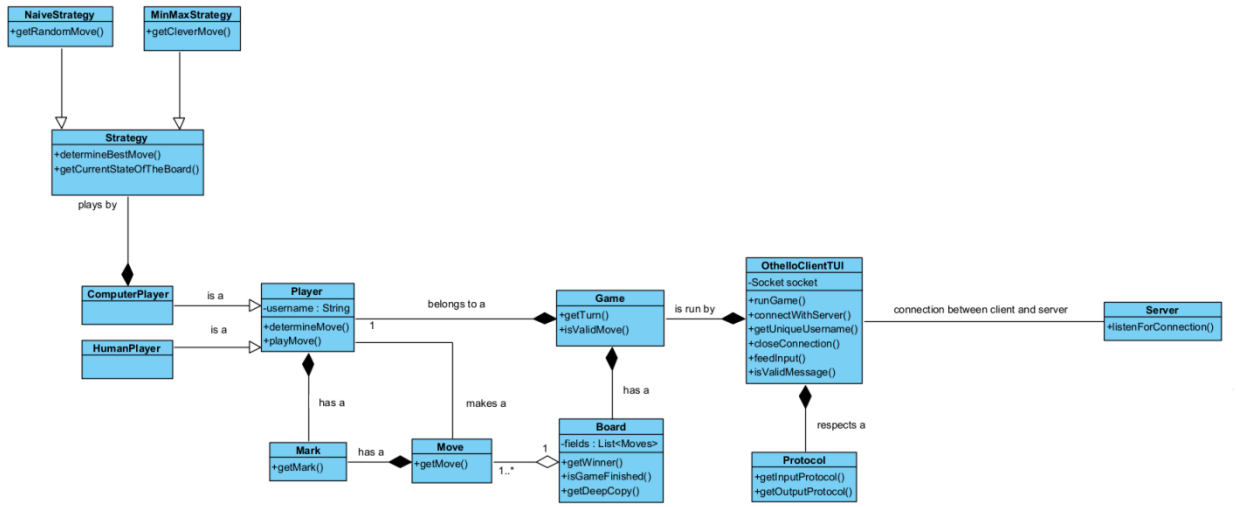
+ Othello: there are 3 pakages:

1. ai: ComputerPlayer, NaiveStrategy, SmartStrategy, Strategy (Interface).
2. model: AbstractPlayer, Board, Mark, Move (Interface), Game (Interface), OthelloGame, OthelloMove, Player (Interface).
3. ui: HumanPlayer, OthelloTUI.

- In the test file, I have a BoardTest.java for testing methods for logic game.

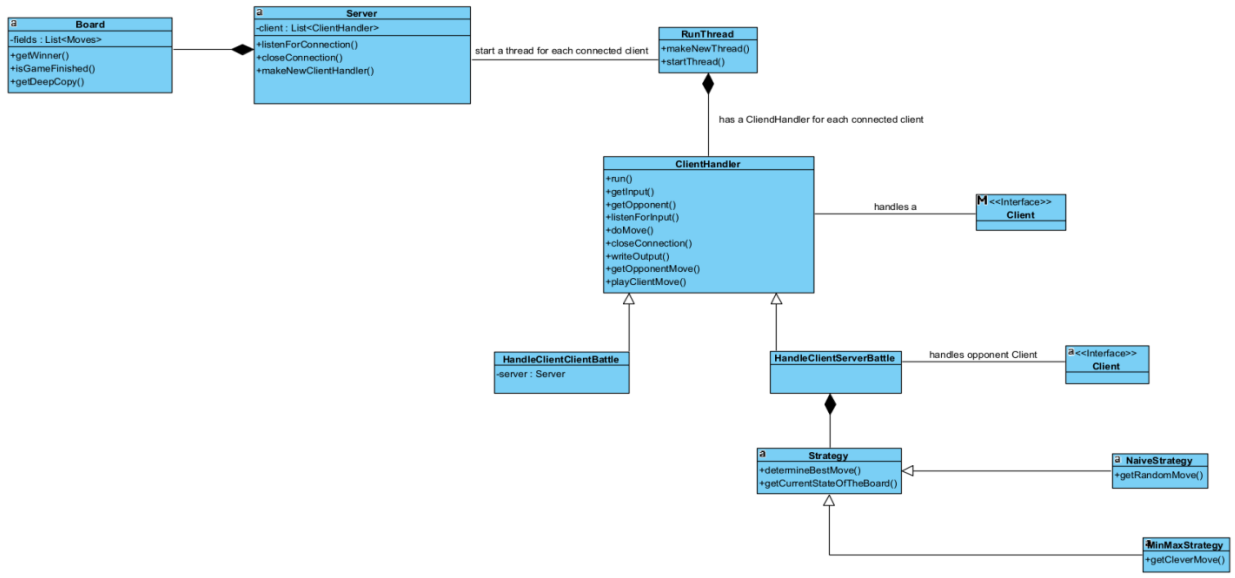
## IV. Initial Design

+/ Client class diagram.



Client class diagram describes the connection between Client class and logic game. We will have an OthelloClientTUI class to help user write the message and input the IP address and a port number to connect with server.

+/ Server class diagram.

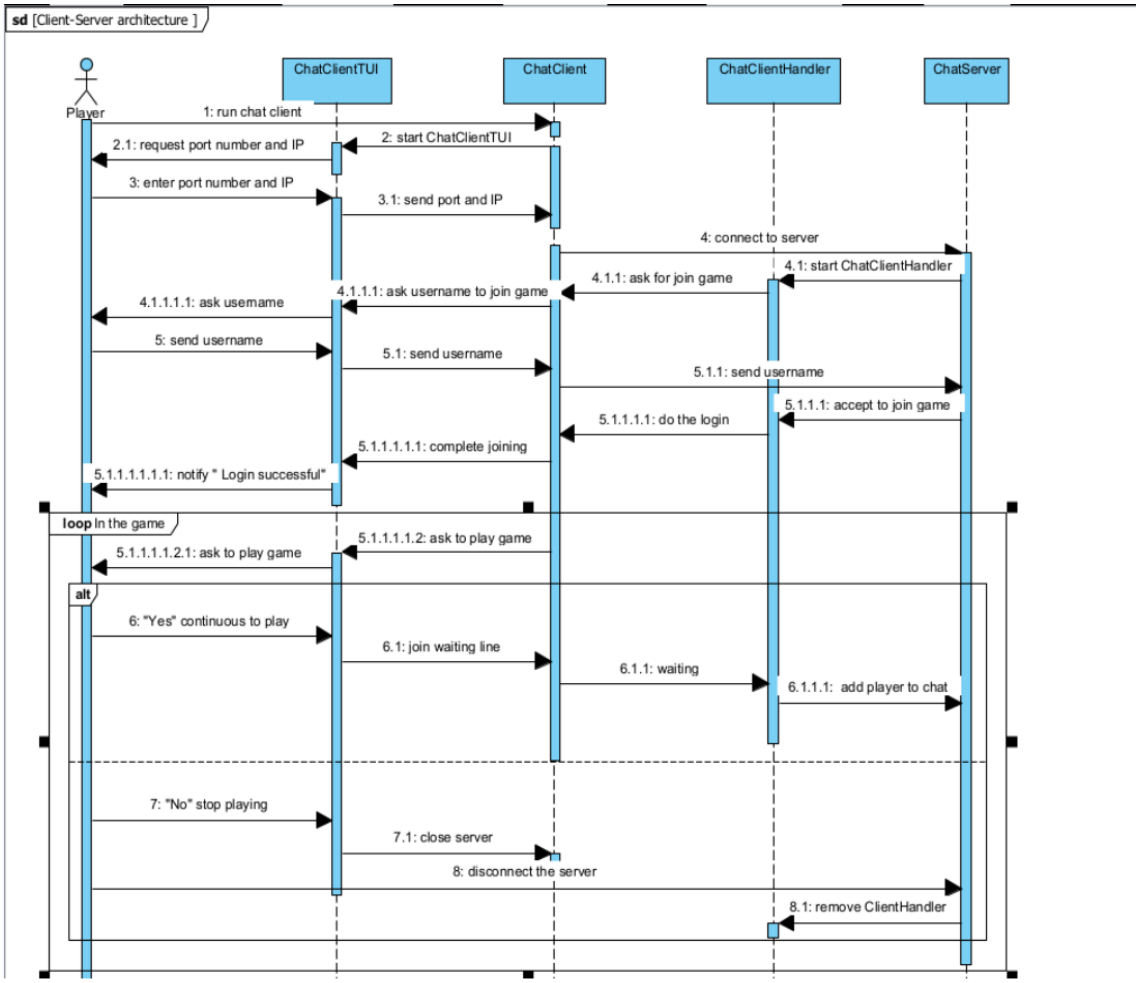



The server class diagram shows the classes and methods that we will need to implement and make a server that connect with client and handle message from client class and then we also can play Othello game on this server.

+/ Sequence diagram Client – Server architecture.

*Description:

- ChatClient and ChatServer will run on a board, and these could be connected together.
- When the ChatServer starts, there are 1-2 threads running and making a connection with ChatClient. And preventing concurrency problems, we use synchronized so it will prevent errors when two clients try to chat at the same time.




+/ Othello Client TUI Board Mapping: the first user interface for the Othello board game

```
   |   |   |   |   |   |   |   |          1  | 2  | 3  | 4  | 5  | 6  | 7  | 8
---+---+---+---+---+---+---+---+---      ---+----+----+----+----+----+----+----
   |   |   |   |   |   |   |   |          9  | 10 | 11 | 12 | 13 | 14 | 15 | 16
---+---+---+---+---+---+---+---+---      ---+----+----+----+----+----+----+----
   |   |   |   |   |   |   |   |          17 | 18 | 19 | 20 | 21 | 22 | 23 | 24
---+---+---+---+---+---+---+---+---      ---+----+----+----+----+----+----+----
   |   |   |   |   |   |   |   |          25 | 26 | 27 | 28 | 29 | 30 | 31 | 32
---+---+---+---+---+---+---+---+---      ---+----+----+----+----+----+----+----
   |   |   |   |   |   |   |   |          33 | 34 | 35 | 36 | 37 | 38 | 39 | 40
---+---+---+---+---+---+---+---+---      ---+----+----+----+----+----+----+----
   |   |   |   |   |   |   |   |          41 | 42 | 43 | 44 | 45 | 46 | 47 | 48
---+---+---+---+---+---+---+---+---      ---+----+----+----+----+----+----+----
   |   |   |   |   |   |   |   |          49 | 50 | 51 | 52 | 53 | 54 | 55 | 56
---+---+---+---+---+---+---+---+---      ---+----+----+----+----+----+----+----
   |   |   |   |   |   |   |   |          57 | 58 | 59 | 60 | 61 | 62 | 63 | 64
```

Schedule for project:

- **Design (18/1-20/1)**
- Making initial design (submit on canvas - 18/1)
- Draw class diagram

- Draw sequence diagram
- Check design and fix
- **Programming (21/1 - 28/1)**
- Create board - 21/1
- A way to move - 21/1
- A way to collect moves - 21/1
- hasWin - 22/1
- isGameOver- 22/1
- Create human player -22/1
- Make AI 23/1-25/1
- Make the client 24/1-25/1
- Make the server 24/1-26/1
- Make TUI
- Fix bugs and improve functions 16/1 - 28/1 ‘
- **Testing and Extensions (21/1 - 28/1)-**
- Unit test functions
- Test TUI
- Add Extensions to the program
- Make System Tests

**+ Writing report, documentation and reflection (29/1 - 1/2)**

- Write introduction 29/1
- Write requirements 29/1-30/1
- Write reflection on design 30/1-31/1
- Explain system testing 30/1-31/1
- Write conclusion 1/2
- Finish documentation

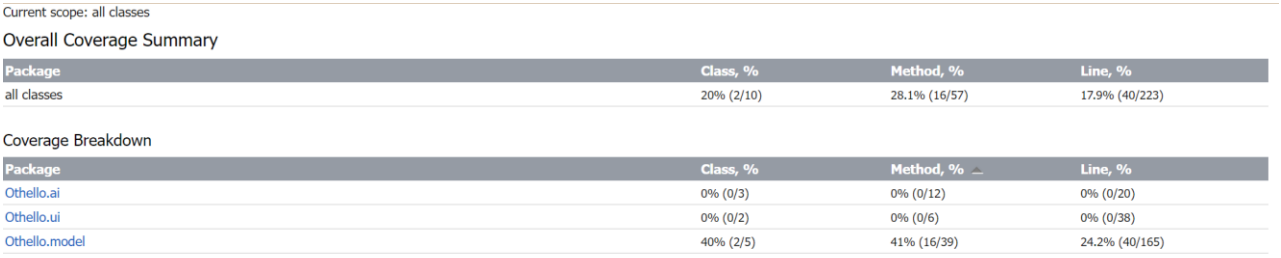
*Group Violet 20 was split on 20/01/2023.

## V. Overall testing Strategy:

### 1. For game logic:

- Coverage metrics
  - Code coverage is a metric that can help you understand how much of your source is tested. It's a very useful metric that can help you assess the quality of your test suite, and we will see here how you can get started with your projects.[1]

  - Test coverage metrics:
    - Run BoardTest with coverage metrics in IntelliJ:



Current scope: all classes

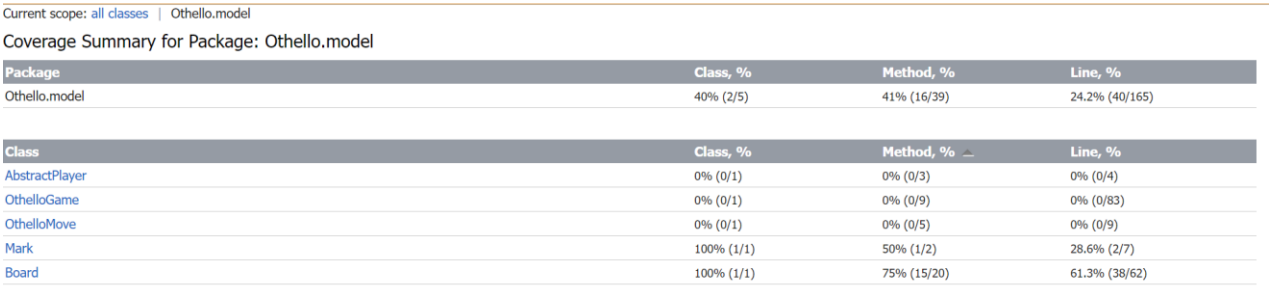
Overall Coverage Summary

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| all classes | 20% (2/10) | 28.1% (16/57) | 17.9% (40/223) |

Coverage Breakdown

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| Othello.ai | 0% (0/3) | 0% (0/12) | 0% (0/20) |
| Othello.ui | 0% (0/2) | 0% (0/6) | 0% (0/38) |
| Othello.model | 40% (2/5) | 41% (16/39) | 24.2% (40/165) |

Image 1: Test coverage for package Othello game.



Current scope: all classes | Othello.model

Coverage Summary for Package: Othello.model

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| Othello.model | 40% (2/5) | 41% (16/39) | 24.2% (40/165) |

| Class | Class, % | Method, % | Line, % |
|---|---|---|---|
| AbstractPlayer | 0% (0/1) | 0% (0/3) | 0% (0/4) |
| OthelloGame | 0% (0/1) | 0% (0/9) | 0% (0/83) |
| OthelloMove | 0% (0/1) | 0% (0/5) | 0% (0/9) |
| Mark | 100% (1/1) | 50% (1/2) | 28.6% (2/7) |
| Board | 100% (1/1) | 75% (15/20) | 61.3% (38/62) |

Image 2: Test coverage for package Othello.model (game logic).

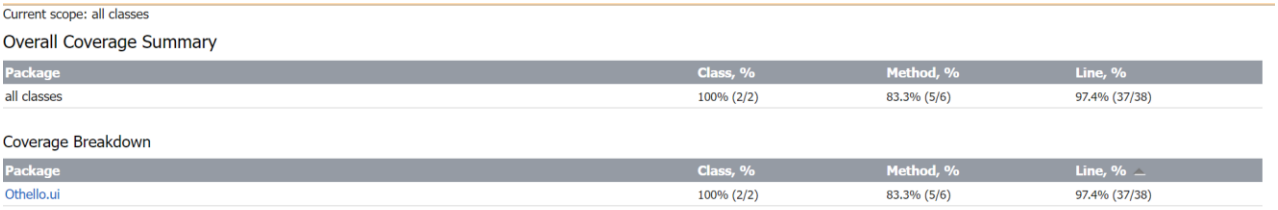
- The overall coverage summary indicates that 20% of the classes, 28.1% of the methods, and 17.9% of the lines have been covered in testing.

- When looking at the breakdown by package, it appears that the Othello.ai and Othello.ui packages have not been tested at all, as indicated by the 0% coverage. The Othello.model package has some coverage with 40% of classes, 41% of methods, and 24.2% of lines covered.

=> There are some classes that have a higher coverage because the classes which cover most of the functionality of the system are tested more than others which are not that critical for the system. And classes that are more connected or dependent with other classes are more likely to be covered during testing than those that are isolated.

- Code coverage metrics:

- Run code coverage metrics (OthelloTUI) in IntelliJ:


Current scope: all classes

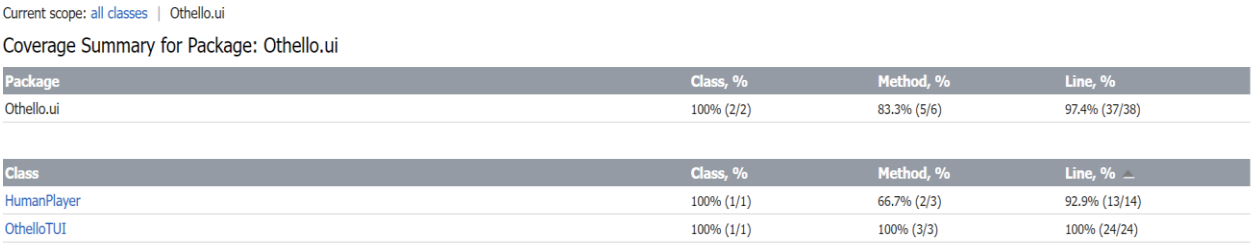
Overall Coverage Summary

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| all classes | 100% (2/2) | 83.3% (5/6) | 97.4% (37/38) |

Coverage Breakdown

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| Othello.ui | 100% (2/2) | 83.3% (5/6) | 97.4% (37/38) |

Image 3: Code coverage for package Othello.ui


Current scope: all classes | Othello.ui

Coverage Summary for Package: Othello.ui

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| Othello.ui | 100% (2/2) | 83.3% (5/6) | 97.4% (37/38) |

| Class | Class, % | Method, % | Line, % |
|---|---|---|---|
| HumanPlayer | 100% (1/1) | 66.7% (2/3) | 92.9% (13/14) |
| OthelloTUI | 100% (1/1) | 100% (3/3) | 100% (24/24) |

Image 4: Code coverage for OthelloTUI

- The coverage summary for the package Othello.ui indicates that all classes (100%), 83.3% of the methods, and 97.4% of the lines have been covered in testing. When looking at the breakdown by class, it appears that both HumanPlayer class and OthelloTUI class have been fully tested, with 100% coverage for both classes.

- The HumanPlayer class has 66.7% coverage for methods and 92.9% coverage for lines. The OthelloTUI class has 100% coverage for methods and lines. It seems that the package Othello.ui has been thoroughly tested, with high coverage for both classes, methods, and lines. The only area that may need further testing is the methods of the HumanPlayer class.

  - Complexity metrics:

    - Cyclomatic complexity is used to gauge the overall intricacy of an application or specific functionality within it. The software metric quantitatively measures a program's logical strength based on existing decision paths in the source code. It is computed by using the control flow graph, where each node on the graph represents indivisible groups or commands within the program.[2]

    - Cyclomatic Complexity: It's a measure of the number of linearly independent paths through a program's source code. It provides an estimate of the number

of test cases needed to achieve a certain level of coverage. A high value of cyclomatic complexity indicates that the method is complex and may have multiple conditions or branches, making it harder to test and understand.

- The table shows the complexity metrics for different methods in the Othello project

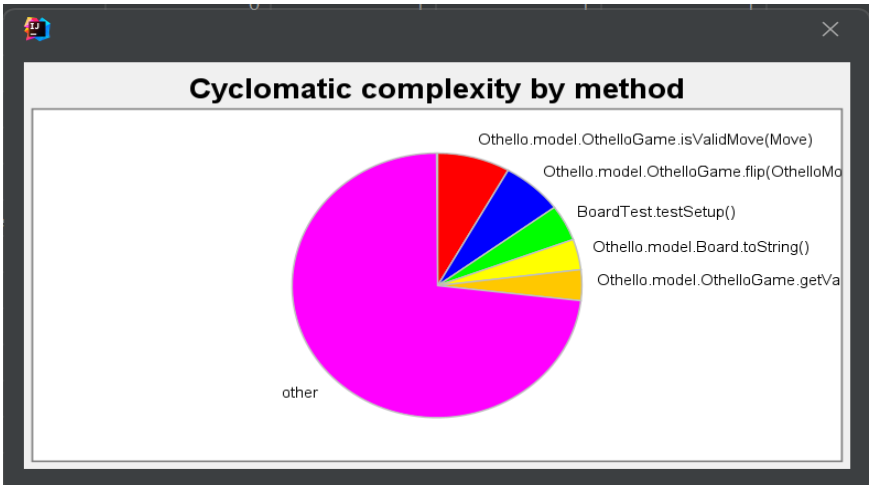



Image 4: Cyclomatic complexity metric

- The table shows that the exception of the BoardTest.testSetup() method which has higher values.
- It seems that the project has methods with low complexity which makes them easy to test and understand. But the method BoardTest.testSetup() may need some refactoring to reduce its complexity.

**2. For whole system:**

➢ Coverage metrics:

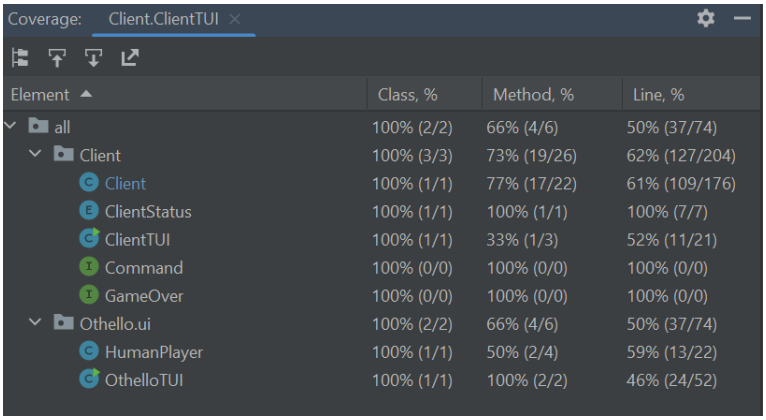



Image 5: Code coverage metric for all classes.

Current scope: all classes

Overall Coverage Summary

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| all classes | 100% (5/5) | 72.7% (24/33) | 59% (164/278) |

Coverage Breakdown

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| Client | 100% (3/3) | 73.1% (19/26) | 62.3% (127/204) |
| Othello.ui | 100% (2/2) | 71.4% (5/7) | 50% (37/74) |

Image 6: Code coverage metric for Client and Othello.ui

Current scope: all classes | Client

Coverage Summary for Class: ClientTUI (Client)

| Class | Class, % | Method, % | Line, % |
|---|---|---|---|
| ClientTUI | 100% (1/1) | 33.3% (1/3) | 52.4% (11/21) |

```
1  package Client;
2
3  import Othello.model.Board;
4
5  import java.io.*;
6  import java.net.InetAddress;
7  import java.net.UnknownHostException;
8
9  public class ClientTUI implements Runnable {
10     private final Board board;
11     public static void main(String[] args) throws IOException {
12         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
13         Client client = new Client();
14         boolean isConnected = false;
15         while (!isConnected) {
16             try {
17                 System.out.println("\n Enter your port number (from 0 to 65536): ");
18                 int portServer = Integer.parseInt(in.readLine());
19                 System.out.println("\n Enter your server address: ");
20                 String serverAddress = in.readLine();
21                 if (!client.connect(InetAddress.getByName(serverAddress), portServer)) {
22                     System.out.println("Error: failed to connect");
23                     System.out.println("Let's try again.");
24                 }
25                 isConnected = true;
26             } catch (UnknownHostException e) {
27                 System.out.println("Error: server is invalid");
28             } catch (NumberFormatException e) {
29                 System.out.println("Error: port number is invalid");
30             }
31         }
32     }
33
34     public ClientTUI() {
35         this.board = new Board();
36         new Thread(this).start();
37     }
38     @Override
39     public void run() {}
40 }
```

Image 7: Code coverage metric check the ClientTUI class.

- The Othello game project has a 59% code coverage overall. This indicates that throughout the testing phase, 59% of the code has been run and tested. Three categories—Class, Method, and Line—are used in the coverage report to categorize the coverage.

- All 5 classes in the project have been tested, as shown by the 100% class coverage. This is a good indicator since it shows that all of the project's classes were covered in the tests.

- A 72.7% method coverage rate means that 24 of the project's 33 methods have been examined. Although this is a respectable coverage percentage, it indicates that certain strategies remain to be tried.

- The Line coverage is 59%, which indicates that 164 out of the 278 lines in the project have been executed and tested. This is a relatively low coverage percentage and suggests that there are still many lines of code that have not been tested.

- When the coverage is broken down by package, the Client package has the highest coverage, with 100% Class coverage, 73.1% Method coverage, and 62.3% Line coverage. Class coverage for the Othello.ui package is 100%, Method coverage is 71.4%, however Line coverage is just 50%.

    ⇨ In conclusion, while the Class coverage is 100%, the overall Method and Line coverage are relatively low, suggesting that there is still room for improvement in the testing phase. It is recommended to increase the coverage by writing additional tests for the untested methods and lines of code.

➢ Complexity metrics:

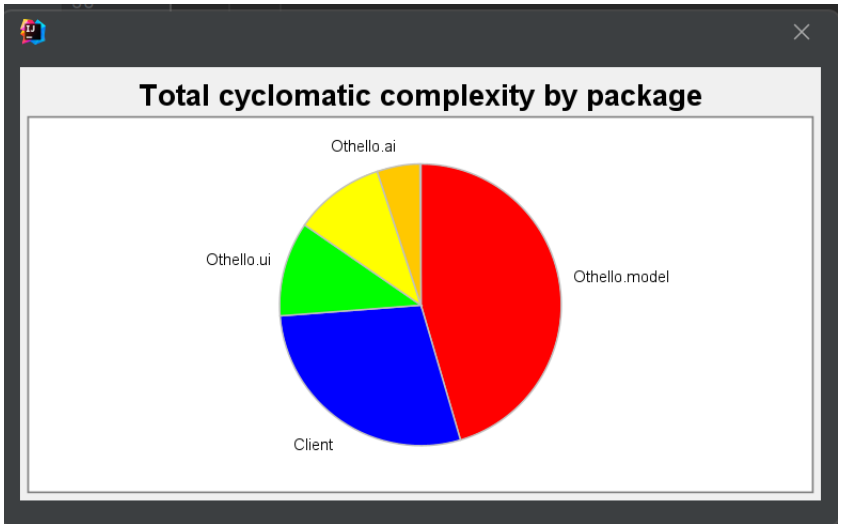



Image 8: Total cyclomatic complexity by package

| package | v(G)avg | v(G)tot |
|---|---|---|
| | 1.79 | 25 |
| Client | 2.72 | 68 |
| Othello.ai | 1.71 | 12 |
| Othello.model | 2.66 | 109 |
| Othello.ui | 4.33 | 26 |
| **Total** | | **240** |
| Average | 2.58 | 48.00 |

Image 9: Table about the total cyclomatic complexity

- The table shows the average and total cyclomatic complexity of four packages: Client, Othello.ai, Othello.model, and Othello.ui.

    - The average cyclomatic complexity (v(G)avg) across all packages is 2.58.
    - The total cyclomatic complexity (v(G)tot) for all packages is 240.0.
- The highest average cyclomatic complexity is found in the Othello.ui package with a value of 4.33, while the highest total cyclomatic complexity is found in the Othello.model package with a value of 109.0

## VI. Test Plan:

| What to test | Description | Steps | Expected results | Verdicts |
|---|---|---|---|---|
| **Automatically** | | | | |
| BoardTest | The BoardTest class is a JUnit test class that tests the functionality of the Board class. It tests the correctness of the methods and the properties of the Board class. | 1.Setup new board<br>2.Create some methods what want to test.<br>3.Run test and fix bugs until no errors. | - The correct calculation of the index of a field.<br>- The correct counting of a certain mark.<br>- The correct flipping of the pieces on the board.<br>- The correct determination of the end of the game. | - All the test cases run correctly.<br>- The Board class is working correctly, and the implementation is free of errors. |
| **Manually** | | | | |
| isGameOver | It checks if the method is correctly determining the end of the game. | 1.Run OthelloTUI<br>2.Invite 2 player<br>3.Play together<br>4.View and check the move | - The gameOver method is correctly determining the end of the game, and that the game can end correctly when the board is full or no more moves are available. | -The test run correctly, it can end the game and return the winner. |
| doMove and flip pieces | Checks if the method is correctly placing the piece of the current player on the board and flipping the opponent's pieces. | 1.Run OthelloTUI<br>2.Invite 2 player<br>3.Play together<br>4.Do move<br>5.Check if the opponent's piece is flipped after that move | - The doMove method is correctly placing the current player's piece on the board and flipping the opponent's pieces. | -The game is moving forward correctly and that the players are able to make valid moves. |
| isValidMove | Checks if the method is correctly determining if a move is valid or not. | 1.Run OthelloTUI<br>2.Invite 2 player<br>3.Play together<br>4.Do move | - The isValidMove method with different indexes and the current player's mark and asserts | -The players can only make valid moves and that the game is played correctly. |

| | | | | |
|---|---|---|---|---|
| | | 5. If the move is valid, do move, otherwise require enter the other move. | that the method correctly determines if the move is valid or not. | |

University of Twente
Group Violet 22 - Bui Duc Cuong - s2966174

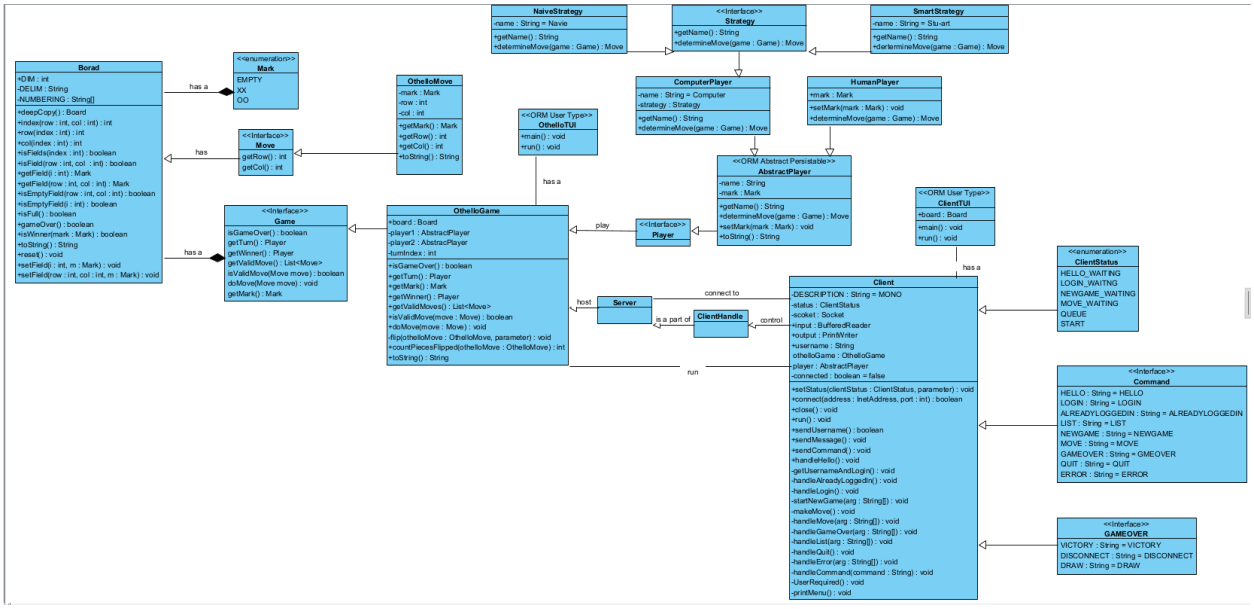
## VII. Concurrency mechanism.

+ [4]Definition: **Concurrency** is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel. The running process threads always communicate with each other through shared memory or message passing. Concurrency results in sharing of resources result in problems like deadlocks and resources starvation.
It helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput.

+ Client: in the programme, I choose using concurrency mechanism when starting thread and starting up the client to play a game on server. Using at sendCommand, sendMessage to avoid many messages and commands being sent at the same time and prevent concurrency errors. Moreover, I also used to synchronize at getUsernameAndLogin, handleAlreadyLoggedIn, handleLogin methods avoiding any race conditions that could occur when multiple threads try to access the same data simultaneously. This can help ensure that the data remains consistent and that there are no unexpected results due to the concurrent access of multiple threads. Synchronizes was used at startNewGame, makeMove, handleMove, handleGameOver methods. By synchronizing these methods, it is implementing a simple form of mutual exclusion, which ensures that only one thread can execute the method at a time, therefore the moves will be send, receive and handle step by step. The handleError method was synchronized because it will avoid there are many errors sent at the same time and the client cannot handle these.

## VIII. Reflection on design

### +/ Class diagram:



***Description:**

+/ In the class diagram for whole system:

- The methods in Client class allow you to connect to the server using its IP address and port number, get game status updates, transmit moves to the server, and get messages from other players.

- The class defines a protocol for message sending and receiving between the client and server and creates a connection to the server using socket programming. This protocol specifies the format for player messages, game status changes, and moves sent and received.

- The player can begin playing the game by making moves and getting information from the server on the current game state after the connection is established and the player has been verified. Additionally, the class offers ways to communicate with other players and receive their messages.

- The ClientTUI class which displays the reply from server and current game's situation and displays the commands from server.

- The Game logic contains Board, Player, Game, Move and Mark:

➔ The Board class in an Othello game is responsible for representing the state of the game board and implementing game logic. This class contains the following methods:

+ deepCopy(): creates a copy of the current board state. This is useful for making temporary changes to the board to evaluate possible moves.

+ index(): returns the index of a specific field in the board, based on its row and column indices.

+ row() and col(): return the row and column indices of a specific field, based on its index.

+ isFields() and isField(): check whether a specific field is part of the board and occupied by either a black or white piece, respectively.

+ getField(): returns the mark (black, white, or empty) of a specific field.

+ isEmptyField(): checks whether a specific field is empty.

+ isFull(): returns whether the board is full, i.e., all fields are occupied by either black or white pieces.

+ gameOver(): returns whether the game is over, i.e., the board is full or no legal moves are available.

+ isWinner(): returns whether the current player has won the game.

+ toString(): returns a string representation of the board, useful for debugging and displaying the board to the user.

+ reset(): resets the board to its initial state.

+ setField(): sets the mark of a specific field on the board, updating the board state based on the rules of the game.

➔ The OthelloGame class in an Othello game is responsible for implementing the game rules and keeping track of the game state. This class contains the following methods:

+ isGameOver(): returns whether the game is over, i.e., the board is full or no legal moves are available.

+ getTurn(): returns the current player's turn.

+ getMark(): returns the mark (black or white) of a specific player.

+ getWinner(): returns the player who has won the game, if there is one.

+ getValidMoves(): returns a list of all valid moves for the current player.

+ isValidMove(): checks whether a specific move is valid for the current player.

+ doMove(): executes a specific move for the current player, updating the board state and switching to the next player's turn.

+ flip(): flips the pieces on the board that would be captured by a specific move. This is a private helper method.

+ countPiecesFlipped(): returns the number of pieces that would be captured by a specific move.

+ toString(): returns a string representation of the current game state, useful for debugging and displaying the game state to the user.

➔ The Player has HumnamPlayer and CompurterPlayer (NaiveStrategy and SmartStrategy).

- There are some changes between the initial design and the new version:

    a. Somethings are still working well in the part:

        i. Some useful methods are still had in classes and the relations between other classes still suitable therefore these still exists but still have some changes in these.

        ii. The relation between Board – Game, Game – Player, ComputerPlayer - Strategy – NaiveStrategy – SmartStrategy.

        iii. The algorithm for logic game inside Board are working as well as in the initial design.

    b. Parts was changed:

    + Client class:

        i. There are many methods implemented to suit the needs and purposes of the client class

        ii. Instance of protocol class to send and receive messages from the server and then handle these. Later, I created two interfaces include command and gameover to support handling the messages.

        iii. I also decided to separate Client into two class: Client contains the methods sending and handing the messages and connect to server, while ClientTUI is created for users to write input and choose the AI players or Human player.

    + Board class:

        i. I implemented more important methods to support converting the move to correctly between row, col and index and direction for the moves. I also created isValisMoves and getValidMoves methods to check and collect the valid moves that player can only set valid moves into board.

        ii. Besides, I added doMove and flip and countFlipPieces methods to set valid moves and flip the opponent's pieces following the rule and the countFlipPieces to count the number of pieces flipped.

        iii. In the initial design, we chose to create two ai strategy and now I made a NaiveStrategy and SmartStrategy.

    + Game

        i. In the initial design, we have some mistakes about the Othello's rules but now I had changed to get better and correct.

ii. At the first version, the Game class was quite simple, and it only can run offline, I later changed to add more setter and getter methods and update it to play online with server and client.

+ Player

i. In the initial design, we have a computerPlayer but it did not have methods that can receive input, so later I decided to improve it and can connect to server to receive data and moves.

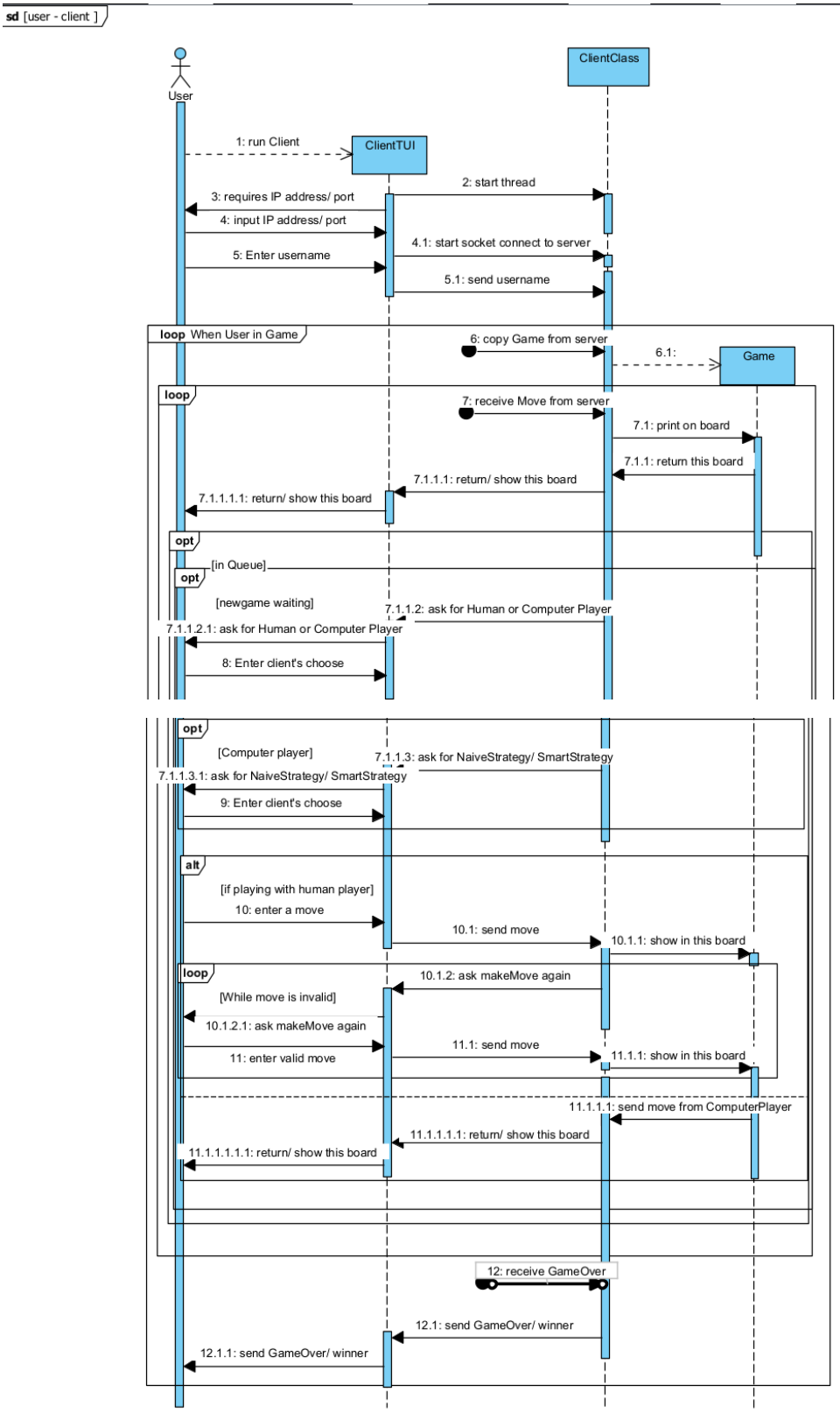
ii. The current AbstractPlayer class ha more getter and setter methods than initial design to suitable create the corresponding data from server for the opponent when playing online.

c. In the future:

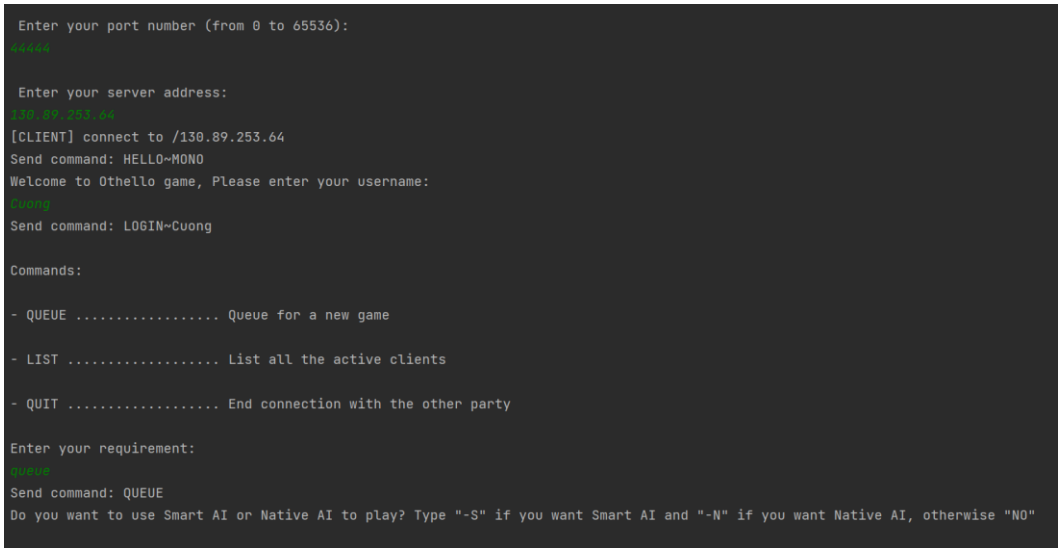
i. I would like to improve TUIs that are user-friendly to players, have the guide for new players and players can get hint when they do not know what a legal move is to doMove.

ii. I would like to improve SmartStrategy to have a better AI by using a stronger algorithm such as Minimax or Monte Carlo Tree Search. These algorithms can explore more possible moves and select the best move based on a score the evaluates the state of game.

**+/ Sequence diagram:**


sd [user - client ]
User
ClientClass
1: run Client
ClientTUI
2: start thread
3: requires IP address/ port
4: input IP address/ port
4.1: start socket connect to server
5: Enter username
5.1: send username
loop When User in Game
6: copy Game from server
6.1:
Game
loop
7: receive Move from server
7.1: print on board
7.1.1: return this board
7.1.1.1: return/ show this board
7.1.1.1.1: return/ show this board
opt
[in Queue]
opt
[newgame waiting]
7.1.1.2: ask for Human or Computer Player
7.1.1.2.1: ask for Human or Computer Player
8: Enter client's choose
opt
[Computer player]
7.1.1.3: ask for NaiveStrategy/ SmartStrategy
7.1.1.3.1: ask for NaiveStrategy/ SmartStrategy
9: Enter client's choose
alt
[if playing with human player]
10: enter a move
10.1: send move
10.1.1: show in this board
loop
10.1.2: ask makeMove again
[While move is invalid]
10.1.2.1: ask makeMove again
11: enter valid move
11.1: send move
11.1.1: show in this board
11.1.1.1: send move from ComputerPlayer
11.1.1.1.1: return/ show this board
11.1.1.1.1.1: return/ show this board
12: receive GameOver
12.1: send GameOver/ winner
12.1.1: send GameOver/ winner

***Description:**




➔ This is user interface when user login into server with client to play the Othello game. The diagram shows the interaction between client and user while into the game. And the TUI and Handler only works as the view to show information and an input reader for the user.

i. Start game:

When it gets a NEWGAME instruction from the server along with the board, the game will begin. It then makes a replica of the game on its side with data for two online players since the game won't be updated unless it receives a MOVE from the server, necessitating the requirement for two online players.

ii. User's first turn:

The ClientTUI will ask "Do you want to use Smart AI or Naïve AI to play? Type "-S" if you want Smart AI and "-N" if you want Naive AI, otherwise "No".

iii. Player's turn:

The Client class will wait for the doMove received from server. And then when the server sent move command, it will be updated on the board, then it also updates the current turn and display it for the player.

If the current turn is the opponent's turn, then the ui will display "Waiting the opponent move?" and it is player's turn, it will display "Enter a valid move:".

iv. User's turn:

Player can only make a move when it comes to the user's turn.

+ Playing as HumanPlayer:

- The user only doMove if it is a valid move, then user need to enter another valid move to doMove.

- A valid move is checked by server. Moreover, the size of the list valid moves is zero, server will send Move~64 and user will change mark to the opponent player. And if two-players also do not have a valid move, then server will send Game-Over to end game.

+ Playing as ComputerPlayer:

- If the user decides AI to make a move for them, then the game will calculate the move and send it to Client and Server.

- For the NaiveStrategy, it was created by using a random method in the valid move list.

- For the SmartStrategy, it was created to do a valid move that can flip the most opponent piece.

+/ User – interface: Board game.

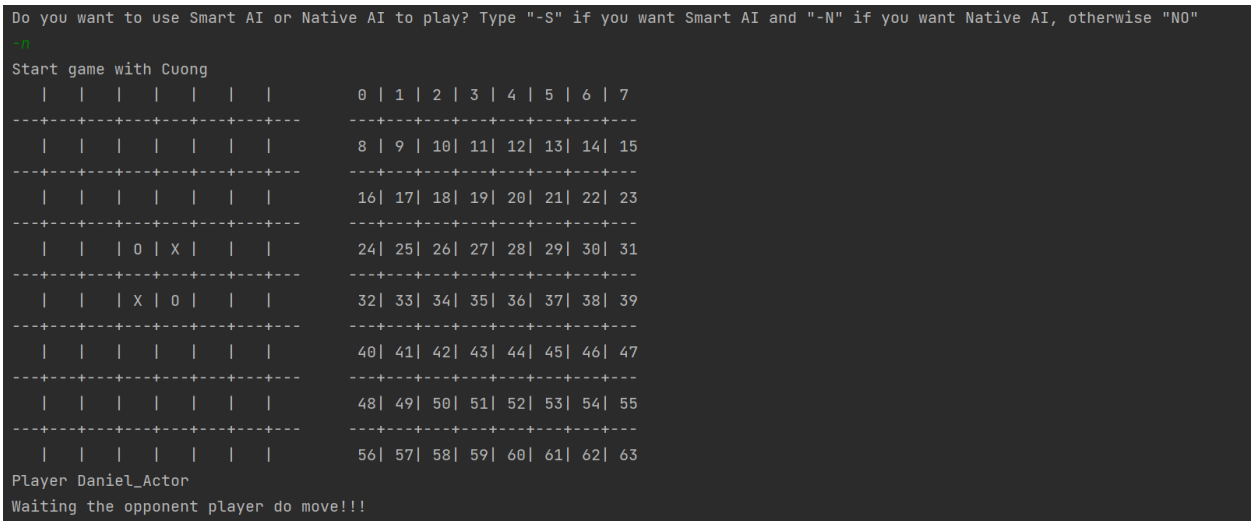



Image 10: Interface when using Computer's Strategy to play.

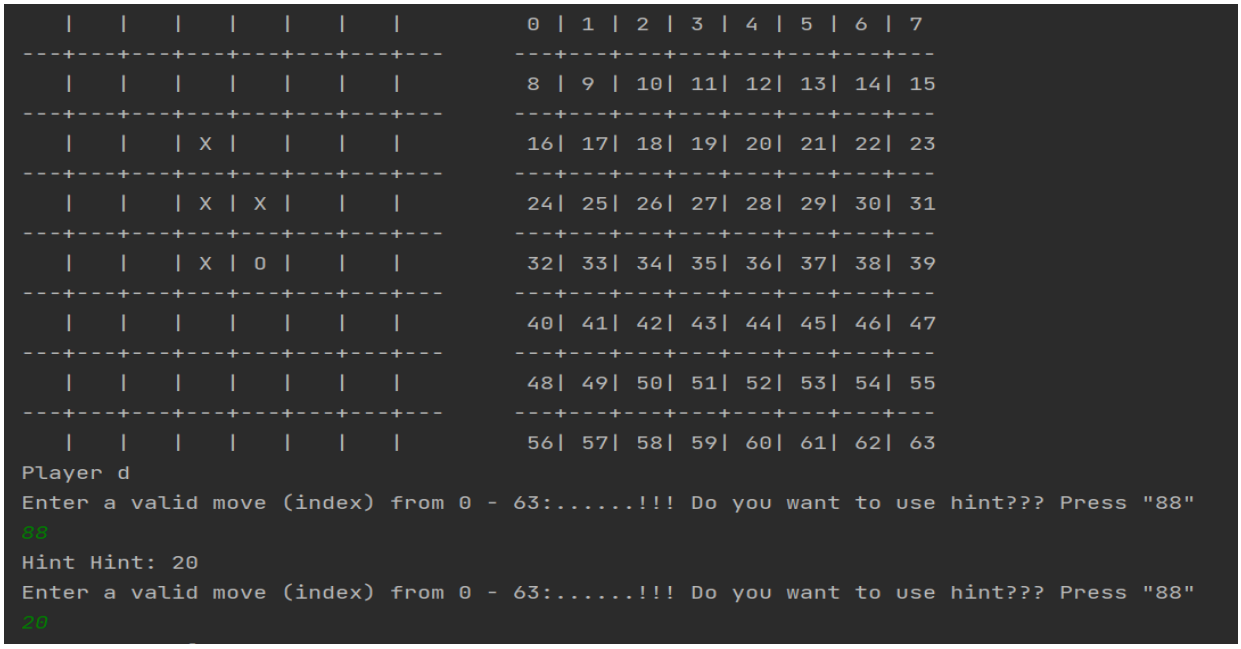



Image 11: Interface when two-players are playing

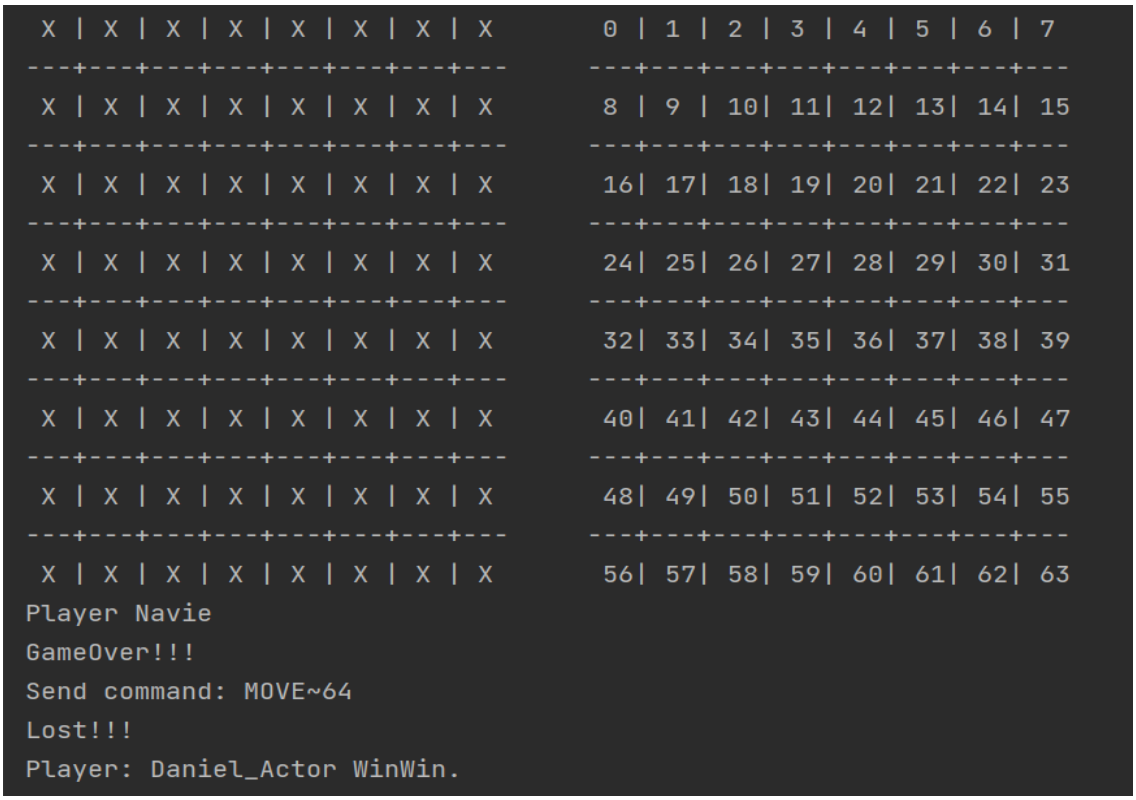



Image 12: The board display the winner when game over.

**Individual reflection:**

It is the first time, I had a chance to do project by myself. Well, my old partner did not talk to me any things about splitting pair group before. At that time, I was very surprised about that when group TA noticed me. However, I think it is a good opportunity for me to understand about myself. When starting the project, I felt some struggle with the logic game but after that I think I learned a lot during the project. It is not only about knowledge; it is but also about the time management. During the project, I also felt little stress because I did not have partner to discuss ideas and workings, I implemented the ideas by myself and sometimes these had a lot of bugs. But with the support of TAs in practical, I handled them. Beside that, in the module and the project, I knew it is important to be consistent with the commits into GitLab. All in all, I saw that this module overall was crazy and I guess we could make it way better if we had more time for different stuff. Thank you for everything!

**Reference:**

[1] https://www.atlassian.com/continuous-delivery/software-testing/code-coverage

[2] https://www.castsoftware.com/glossary/cyclomatic-complexity

[3] Othello game rules

[4] https://www.geeksforgeeks.org/concurrency-in-operating-system