

# **Solo Project: Othello Game**

**Name:** Yujun Li

**Student Number:** 2889382

**Group:** Red 13

**Date:** July, 15th 2023

# 1. Project Overview

Othello, also called Reversi, is a board game that two people can play. The dimension of the board game is 8x8, meaning that there are 64 fields in total on the board. Before the start of every game, four fields in the center of the board are occupied by two black discs and two white discs. Players make a move by placing a disc on the board. Each player chooses a color of the disc between white and black. The player who uses the black disc makes the first move. Players have to place their discs on valid fields. A valid field refers to the field that once the player sets their disc on it, at least one disc of the opponent will be outflanked. To outflank the opponent's discs means to place a disc on the board so that the current player's row, column, or diagonal of discs is bordered at each end by the disc of the current player. The game ends when there is only one color of the disc on the board, or the board is full.

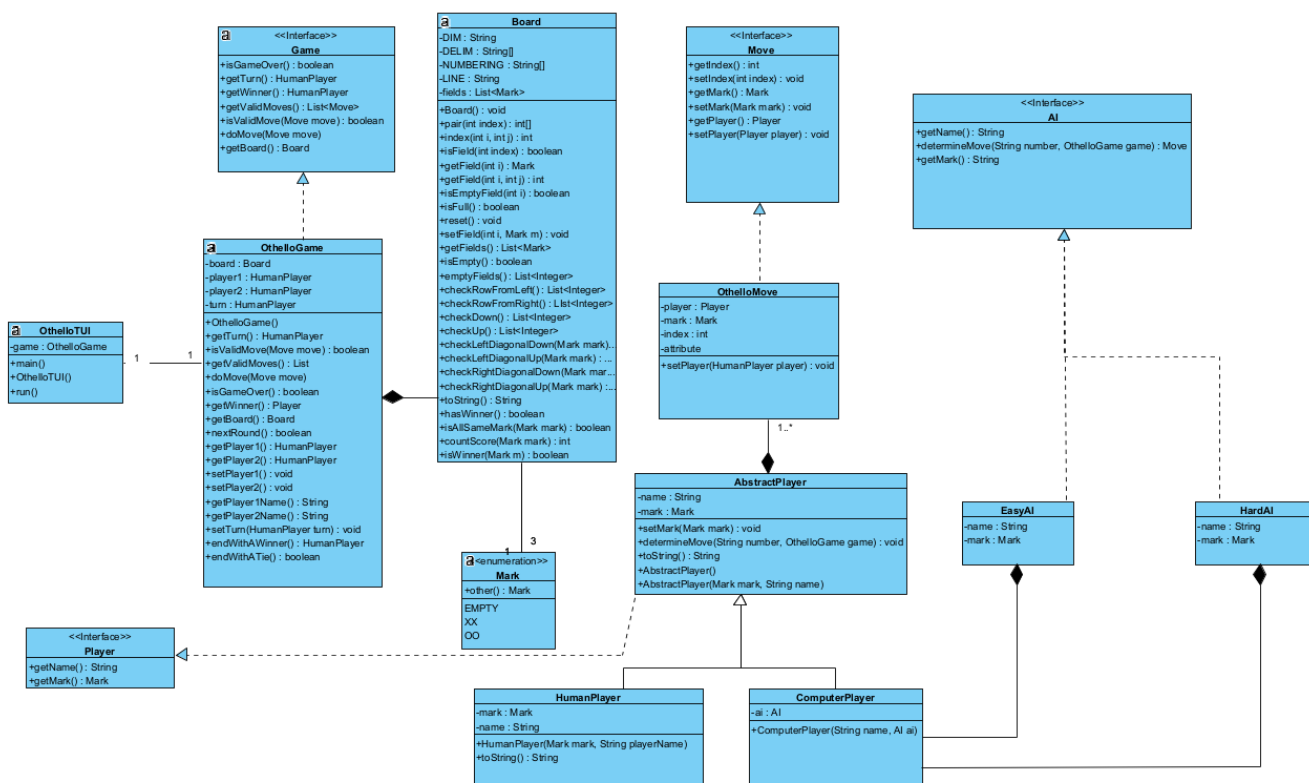
In this project, the logic of the Othello game, a client end, a server end, and corresponding GUIs is designed and implemented using the programming language Java.

## 2. Program Design

### 2.1 Design Overview

There are five packages in this project: client, commonUtil, game, server, and gui. In addition to these, two folders, namely img and lib are include. The img folder stores the images needed for the gui and the file folder record the score, rank, name and password of each client. The game package stores all classes associated with the Othello game logic, such as Board. The second one is the server which stores all classes related to the server functions, such as ClientHandler. The CommonUtil package includes classes that are shared among all the over packages. The gui package stores all the GUI classes used to provide users with a visual view while playing the game. To demonstrate the relationship between classes, a class diagram is made, and classes from both packages are on the class diagram, as shown in Figure 1. Due to the fact that there are many classes, it is infeasible to put all the classes in one diagram and see details of classes. Thus, I made class diagram for each package as shown below.

## 2.2 The List of Responsibilities of Classes in the Game Package



### 2.2.1 Board

- Set up a board with two white discs and two black discs in the center of the board, as shown on the right.
- Places a disc on the board
- Checks if a move is a valid move
- Flips discs
- Checks if there is a winner
- Counts the score for the winner
- Finishes the game by checking if the game ends with a winner or ends with a tie

### 2.2.2 Game<Interface>

- Blue print for making a game class such as Othello Game class

### 2.2.3 OthelloGame

- Allows players to make a move
- Applies rules of OthelloGame

- Checks if the game is over
- Starts the game
- Ensures players play by turns

#### 2.2.4 Player<Interface>

- blueprint for making AbstractPlayer class

#### 2.2.5 AbstractPlayer

- generalization of computer player and human player

#### 2.2.6 ComputerPlayer

- generalization of HardAI and EasyAI class

#### 2.2.7 AI<Interface>

- blueprint for making HardAI and EasyAI class

#### 2.2.8 EasyAI

- EasyAI class that play with human player. Randomly select a move from valid moves to make a move in a game

#### 2.2.9 HardAI

- HardAI class that play with human player. Compute the number of discs that a move can flip from all the possible valid moves and choose the move that flips the most discs

#### 2.2.10 HumanPlayer

- Determines a move and makes a move for users

#### 2.2.11 OthelloMove

- Makes a move

#### 2.2.12 BoardTest

- Tests all operations in the Board class by using JUnit tests

#### 2.2.13 Mark

- Represents fields with three marks: EMPTY, XX, and OO

- X is the black disc, and O is the white disc.

#### 2.2.14 OthelloTUI

- Tests the move made by players visually and manually

•

### 2.3 The List of Responsibilities of Classes in the Client Package



#### 2.3.1 ClientService

- connects with the server
- generates a socket
- makes sure connection between the server and the client is established
- check the credentials typed in by users
- generates a Listener threads when the connection between server is stable and the credentials entered by the user

#### 2.3.2 Listener

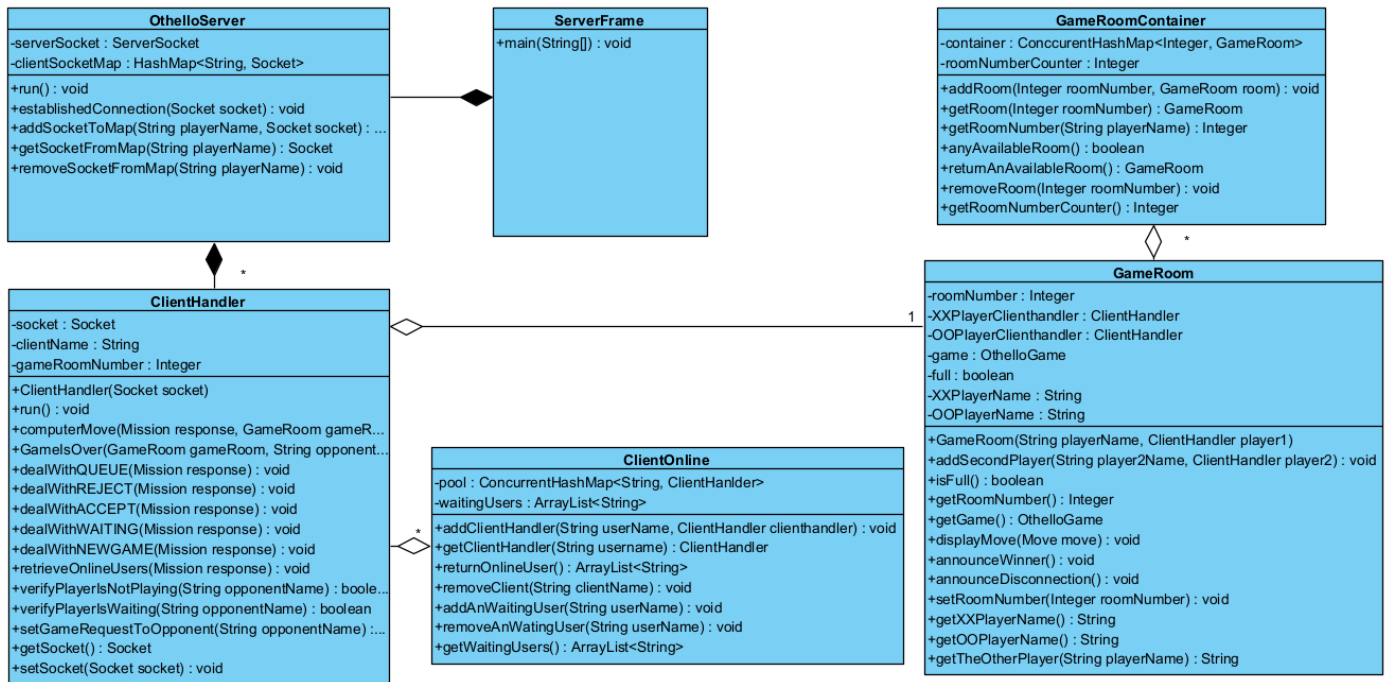
- read the output sent from the server

#### 2.3.3 ListenerContainer

- store all the listener that corresponding to each client

- delete listener threads by usernames
- add listener threads by usernames

## 2.4 The List of Responsibilities of Classes in the Server Package



### 2.4.1 ClientHandler

- Processes and react to the incoming protocols sent from the client to the server
- Adds itself to a client container after the client has successfully logged in
- Adds itself to a game container after the client has successfully queued

### 2.4.2 ClientOnline

- Holds all logged-in clients
- Returns all logged-in clients as online users

### 2.4.3 GameRoom

- Executes moves made by clients
- Allows players to play Othello game in the room
- Adds players into the game room
- Notifies players about their moves

- Notifies players when the game is over

## 2.4.4 GameRoomsContanier

- Holds all rooms that are occupied
- Provides available rooms for players to join

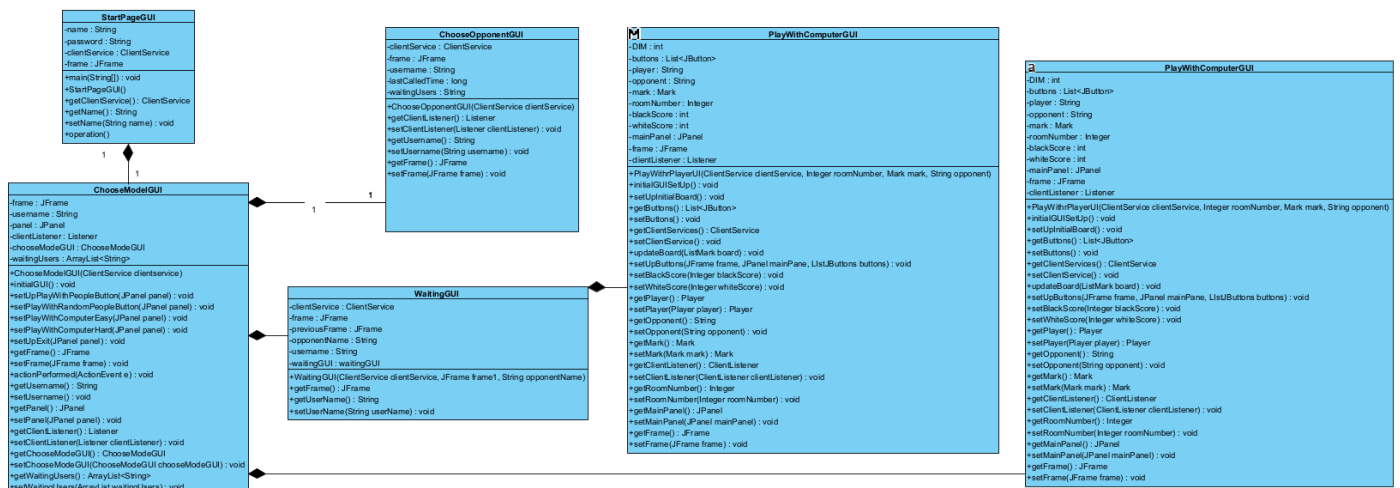
## 2.4.5 OthelloServer

- Starts up the server with a given port, 44444
- Accepts connections from the client

## 2.4.6 ServerFrame

- Starts up the OthelloServer

## 2.5 The List of Responsibilities of Classes in the GUI Package



### 2.5.1 StartPageGUI

- allows users to see if the connection between server and the client is established
- allows user to enter their credentials
- if username is not in the system, the user's username and password are stored in the system and user will be alerted that a new account is created
- if username is already in the system, the user's credential matches with the one stored in the system, the user can login successfully

### 2.5.2 ChooseModeGUI



- allows users to choose different modes of playing the game
- user can choose play with easy computer, hard computer, randomly choose an opponent to play with, choose a specific player to play with or waited to be invited to play a game

#### 2.5.3 WaitingGUI

- allows users to be invited into a game
- users can reject or accept invitation on this page

#### 2.5.4 ChooseOpponentGUI

- users can view which clients are in the waiting list and invite them to play the game
- users can go back to the previous page and come back to refresh the waiting list
- users can choose a waiting player to play the game

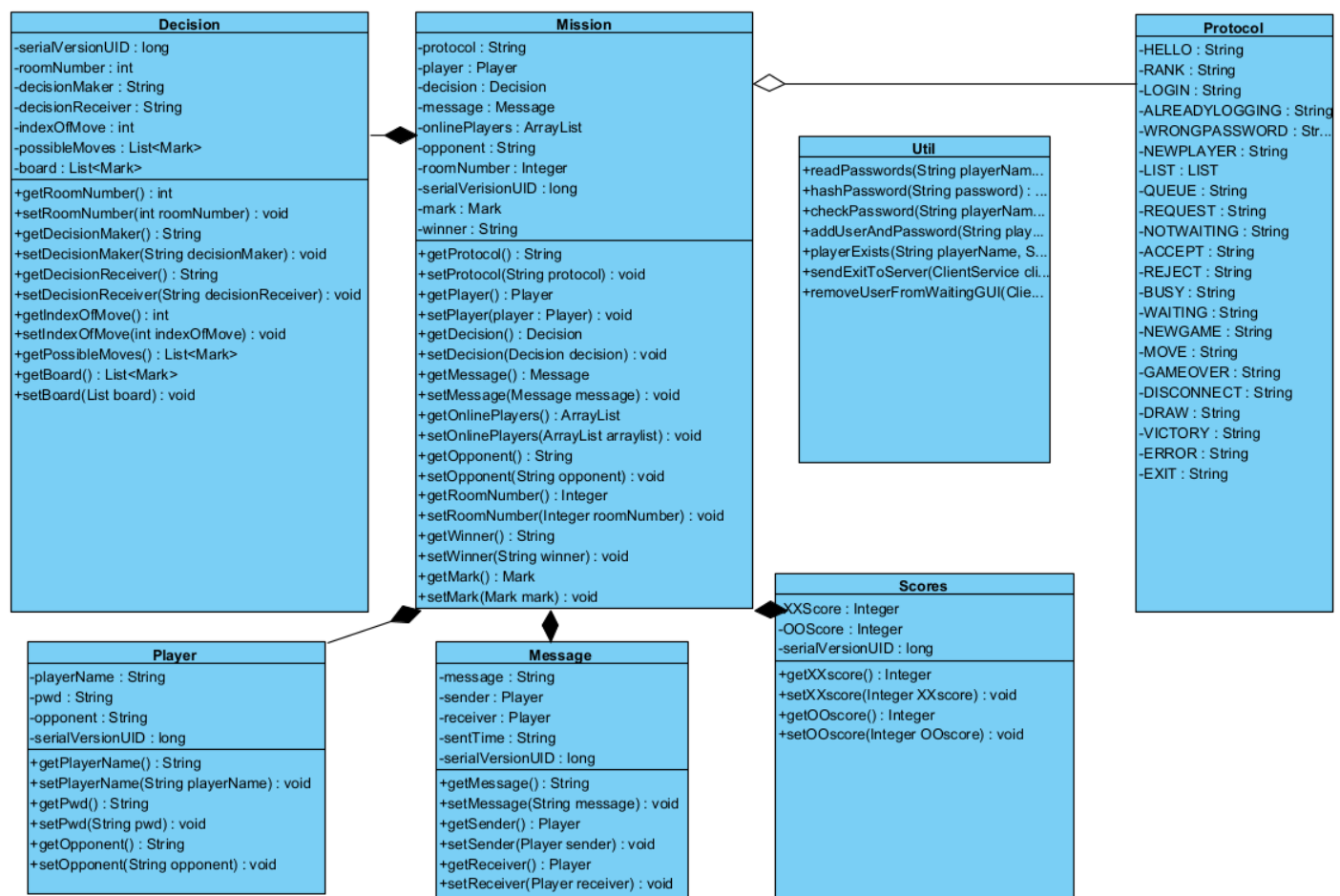
#### 2.5.5 PlayWithComputerGUI

- users can make move by clicking on the buttons of the board to make a move
- users can give up the game by clicking the game button
- users can read the rules of Othello game by clicking the rule button
- users can see the rank of the game

#### 2.5.6 PlayWithPlayerGUI

- users can make move by clicking on the buttons of the board to make a move
- users can give up the game by clicking the game button
- users can read the rules of Othello game by clicking the rule button
- users can see the rank of the game

## 2.6 The List of Responsibilities of Classes in the CommonUtil Package



### 2.6.1 Mission

- it is the means that used by the client and server to communicate
- it contains Message, Player, Scores, and Decision objects and Protocol to indicate the actions made by the client and server

### 2.6.2 Message

- it contains the message sender
- it contains the message receiver
- it contains the message itself

### 2.6.3 Player

- it is used to check the credential of username and password

#### 2.6.4 Scores

- it contains the score of black disc holder
- it contains the name of black disc holder
- it contains the score of white disc holder
- it contains the name of white disc holder
- it contains the room number

#### 2.6.5 Decision

- it contains the index of move that made by the user
- in contains the name of the decision maker
- it contains the name of the decision receiver
- it contains an indication whether the maker made a move or not under the condition that there is no more valid moves

#### 2.6.6 Protocol

- standardized message form used by the client and server to communicate with each other

### 2.7. Each Class's Connection with Other Class:

#### 2.7.1. OthelloServer Class

OthelloServer is the class that receives the request for connection from the Client. In this class, a server socket is created and stored in this class as a private attribute once a client is connected to the server. The OthelloServer implements a Runnable interface and itself is a thread that keeps running and accepting connection requests from the client. The server can connect to multiple clients at the same time. After establishing a successful connection, the server generates a ClientHandler thread. It passes the corresponding socket into this ClientHandler thread, which processes the information sent from the client to the server. All ClientHandler threads that are produced by establishing successful connections are stored in a Vector list. Thus, in the future, when shutting down the server, all ClientHandler threads will be removed and safely closed.

### 2.7.2. ClientHandler Class and Protocol Interface

#### Protocol Interface:

All protocols sent back to and received from the client are stored in this interface as an attribute with the type of String.

#### ClientHandler Class:

This class implements the Runnable interface. Each thread is responsible for handling requests and information sent from the client. The corresponding socket used to communicate with the client listener is stored as an attribute. Once the client handler thread is created, by default, the `threeWayHandshake` attribute in `ClientHandler` is set as false to indicate that a three-way handshake is yet to be established. Before the client can be sent other protocols to the server, a three-way handshake has to happen. The three-way handshake is made by sending correct HELLO and LOGIN protocols between the client and the server. As the client sends the LOGIN protocol, the `ClientHandler` uses the `ClientPool` class to check whether the username is already logged in. If the client with this username has not logged in, the username will be stored in the `clientName` attribute. More details of `ClientPool` will be discussed in the following section. Once a three-way handshake is established, the `three-handshake` attribute in the client handler will be set as true and the client can continue sending other protocols, such as QUEUE, LIST, and MOVE to the

server. Before the client can make a move, the client has to send a QUEUE protocol. Once it is sent, the `ClientHandler` checks if game rooms are available in the `GameRoomsContainer`. If there is at least one open room, this `ClientHandler` will be added to that game room and can start to play, and the name of the room will be stored in the `gameRoomName` attribute. Suppose there is no available room in the `GameRoomsContainer`. In that case, a new `GameRoom` object will be created, and this client will be the first player in the room (this part will be expanded in the `GameRoomsContainer` and `GameRoom` section). The name of the newly created room will be stored in the `gameRoomName` attribute of this `ClientHandler`. Only after being assigned to a game room will the server be able to process the MOVE protocols sent from the client. After a correct MOVE protocol is sent to the client, the `ClientHandler` makes the move on the Othello game stored in the `GameRoom` object.

### 2.7.3. ClientOnline Class

`ClientOnline` is a container that holds all `ClientHandler` threads that establish a three-way handshake in a concurrent `HashMap`. In this `HashMap`, the key is the client's name and the value is the

corresponding ClientHandler. The ClientHandler can use static methods in the ClientOnline pool to pull out the list of logged-in clients and return them in the form of String. Moreover, the ClientHandler can be added to, retrieved, and removed from the ClientPool using static methods in this class.

#### 2.7.4. GameRoom and GameRoomsContainer Class

GameRoom:

it refers to the room where players play OthelloGame. In this room, the name and the ClientHandler of both players are stored. If two players are in the room, the isFull attribute will be set as true. The ClientHandler creates the game and this ClientHandler's name and itself are passed into the constructor of the GameRoom. In the constructor, a new OthelloGame is created and the clientName of this ClientHandler is passed into the OthelloGame by calling the setPlayer1 method. Consequently, the first HumanPlayer is created in this new OthelloGame (more details are explained in the OthelloGame section). After a GameRoom is successfully created, it is put into GameRoomsContainer class by calling the static method addRoom. Three notifyAllPlayers methods notify all players about moves made by both players when the game is not over after a player makes a move, notify players when there is a winner, a tie, or a disconnected player in the game.

GameRoomsContainer:

It represents a container that stores all the GameRoom that are created and are being used. When a client QUEUE, the isAnyAvialable rooms in this class will be called to check if there are any available rooms in the container, if there is at least one available room, return the first available room to the client, and the client can join this room using the addSecondPlayer method. When addSecondPlayer is called, the clientName and the ClientHandler are passed in as parameters and stored in player2Name and player2 attributes. Then, a HumanPlayer is created and stored in the OthelloGame in GameRoom. If there is no available room in the container, as described in the GameRoom section, ClientHandler thread will new a Game room, put the player in as the first player, and this game room is named after the first player. Afterward, this room will be added to the GameRoom container, waiting for the next QUEUE player to join.

#### 2.7.5. Board

this class is the backbone of the game, representing the board players play on. This board has attributes DIM which represents the dimension of the game. The toString method of this class can show the fields and marks on the board. There are three possible statuses of a field on the board:

EMPTY, XX, and OO. The Mark enum represents these three statuses. There are multiple operations in the Board that server the purpose of placing moves made by players on the board and ensuring moves made are valid. Players make their moves through the OthelloGame object.

After a move is determined by the player, the board checks a move from 8 different directions and flip discs when the move is valid as shown in figure 2. After every move is made, the board checks if there is a winner on the board.

#### 2.7.6. OthelloGame and Game Interface:

Game Interface: it acts as a blueprint for OthelloGame.

OthelloGame: it refers to the game that players actually play. In a game, there is one and only board stored as a private attribute. There are two players in one game and each of them is stored in player1 and player2. Also, the turn of the player is stored in the game as well to ensure the fairness of the game. In the game, players can do moves. When a move is made, the move will be checked if it is valid or not. Moreover, when a move is made, the

game checks if the game ends. When a game is over, if the game ends with a winner, the endWithWinner method is called. Else, if the game ends with no winner, the endWithaTie method is called. OthelloGame is an attribute stored in GameRoom as every game room needs only one game.

#### 2.7.7. Move, OthelloMove, OthelloTUI, Player, Abstract Player, HumanPlayer

Move interface: it acts as a blueprint for the OthelloMove class

OthelloMove: this represents a move made by a player. One player can make multiple moves. A move made by a player has to have the mark of this player and the index of the move. To make a move, the player needs to pass in a mark and an index of the move.

Human Player: this refers to the player of the Othello game. A player has a mark and a name. The player can determine the moves that they desire to make and pass in the move they decided to make into the game. The OthelloGame object checks on the move and determines whether it is a valid move or not.

Othello TUI: although it is not required for solo students who are doing the server to develop a TUI for the game and computer players. However, in order to check the correctness of game logic. A simple

TUI is designed. In this TUI, a game is stored.

Player, Abstract Player, Human Player, AI, HardAI and EasyAI: Player is the interface. Abstract Player class acts as the interface for HumanPlayer and AI players. For AI, there are two different kinds of AI, Hard AI and Easy AI which are created when users choose the corresponding mode to play. Player classes are used by the classes in the game package. When they are created they are manipulated by the Othello Game class to make a move.

#### 2.7.8. GUIs and Client Classes

As GUI classes are heavily involved in terms of user experience, the functions and relations of GUI and Client classes will be elaborated in the following section with activity diagrams.

## 3. Diagrams

### 3.7. Sequence Diagrams of Server and Client

#### 3.7.1. Establishing Connection between Two Clients and the Server

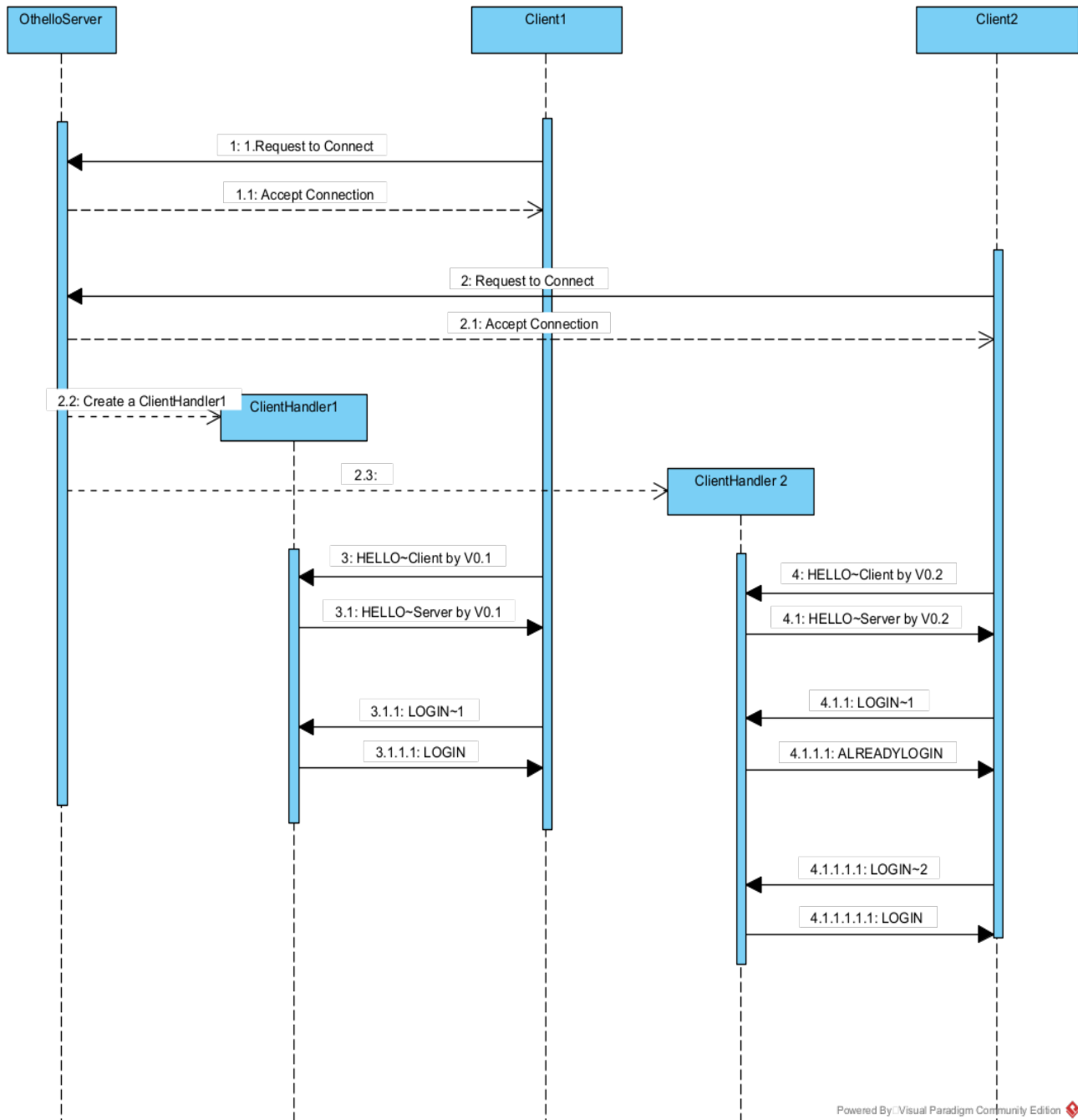
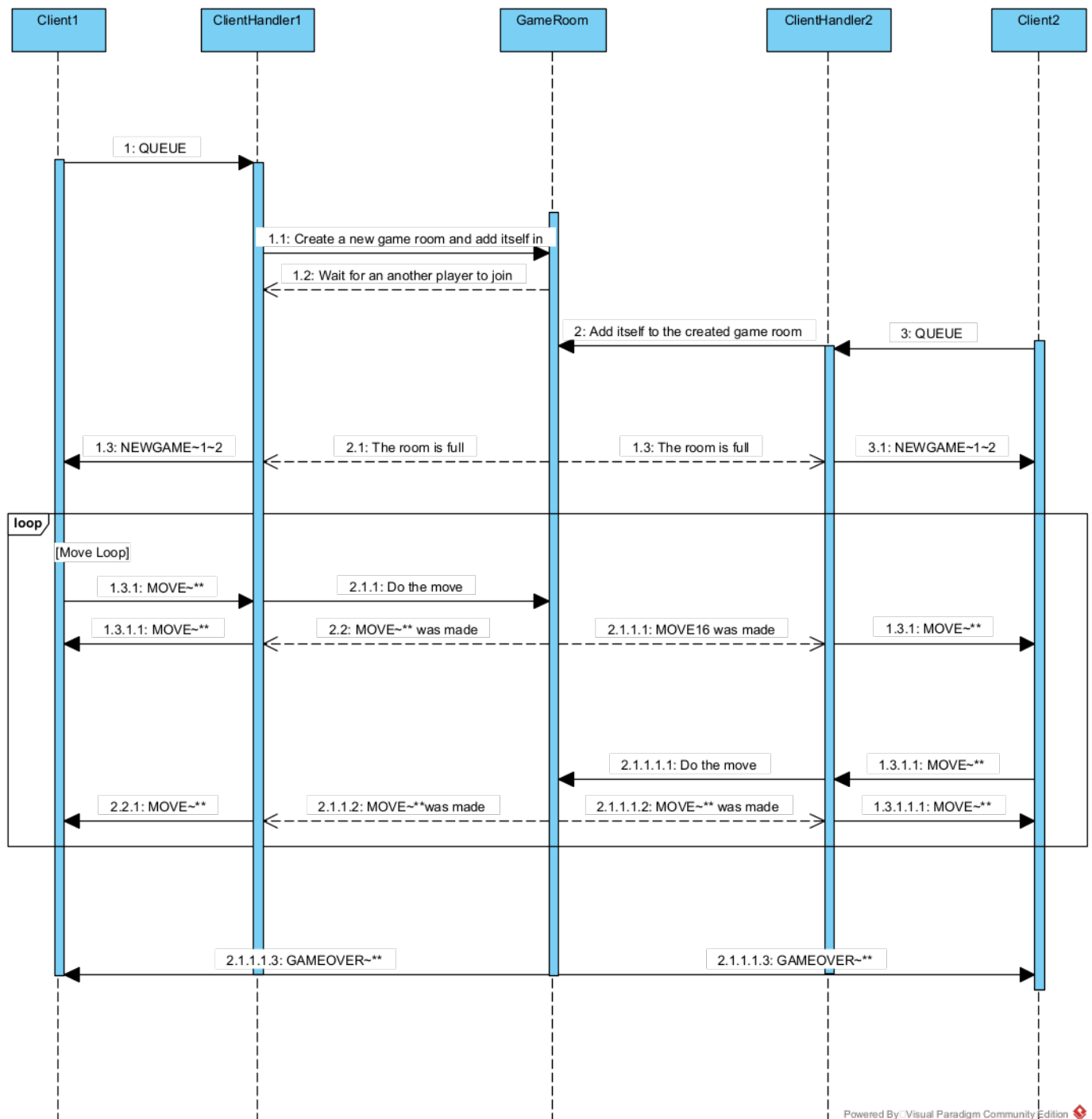


Figure 1: Sequence Diagram



This sequence diagram demonstrates the process of two clients establishing the connection with the OthelloServer. The first client, Client 1, sent a request to connect with the server with a specific address on a specific port, which is 4444. The OthelloServer receives the request from Client 1 and accepts it. Then, a ClientHandler, namely ClientHandler 1, is created to process information sent from Client 1. The second client, Client 2 does theClients Make Moves and Game Over. the server creates a ClientHandler 2 to work with Client 2. Then, to establish the connection, both clients need to send HELLO with the name of the clients. ClientHandlers receive HELLO from clients and sent HELLO back as confirmation. By receiving HELLO, clients then can send LOGIN with users' names to login into the server. Client 1 sends a LOGIN protocol with a username that has not logged into the system. ClientHandler 1 receives this protocol and responds to the client with LOGIN to confirm that this user has successfully logged in. Now, the connection between the server and Client 1 has been successfully established. However, Client 2 first try to log in with a username that has just logged in, which is Client 1. The ClientHandler 2 respond to this with ALREADYLOGGEDIN protocol to indicate that Client 2 needs to log in as a new user. After receiving this information, Client 2 logs in with a new user "2". The server responds to this action by sending back a LOGIN protocol to confirm that this new user is logged in successfully and a three-way handshake has been established between the server and Client 2.

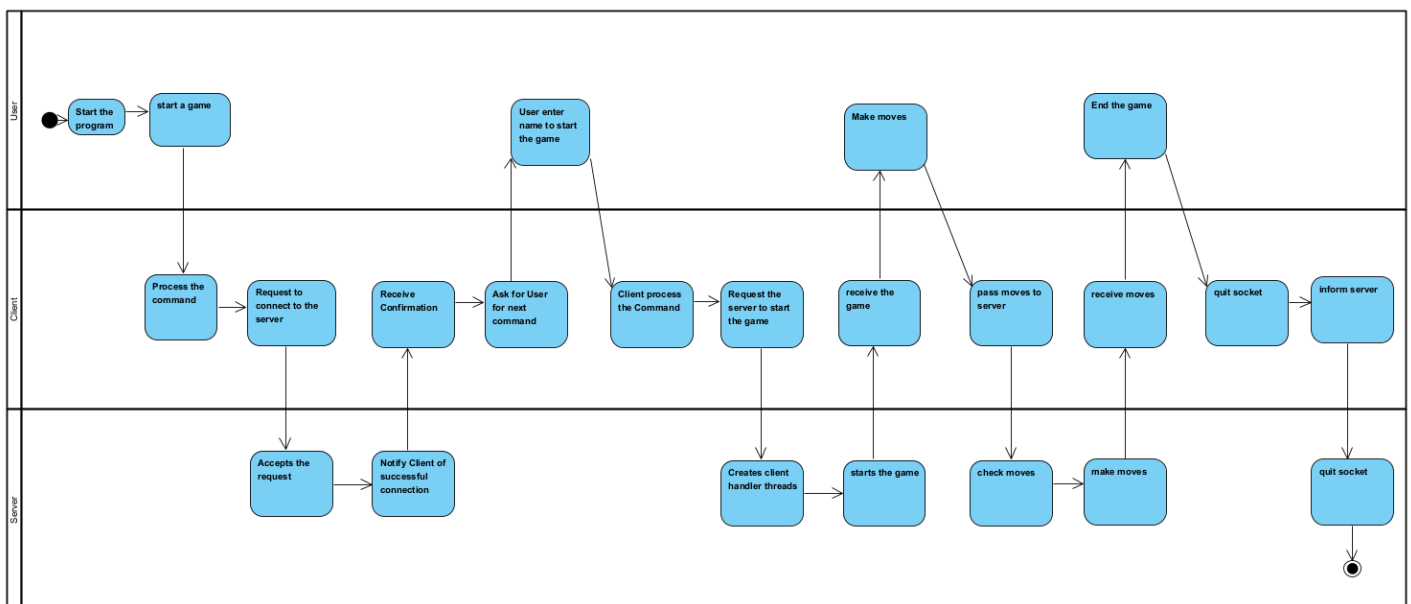
### 3.7.2. Sequence Diagram of Clients Make Moves



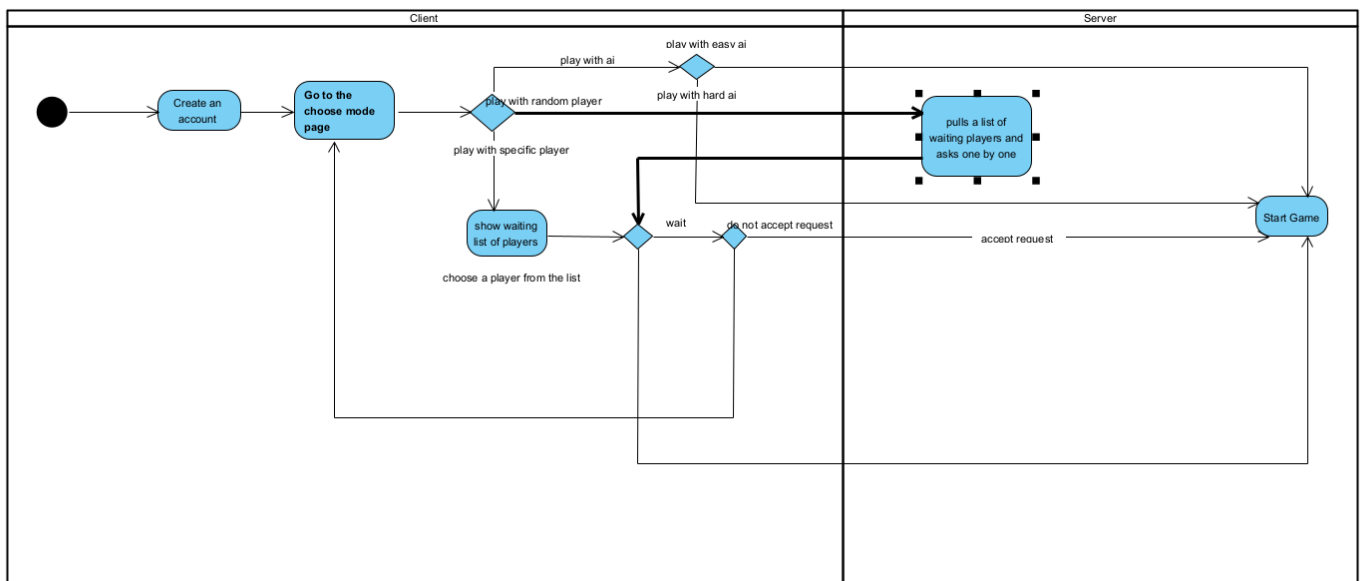
After establishing a three-way handshake as shown in Figure 3, the client can send QUEUE command to wait in line to play an Othello Game. When ClientHandler 1 receives QUEUE protocol from Client 1, it creates a new game room and adds itself to the game room, and wait for another player to join this room. Then, Client 2 sends a QUEUE protocol to ClientHandler 2. At this point, there are two players in the server and they can be in the

same game room and start to play a game. Thus, ClientHandler 2 adds itself to the room created by ClientHandler 2. Once the second player joins the room, the game room notifies both players in the room that the room is full. When ClientHandlers receive this signal from the game room, they send the NEWGAME protocol to both clients, indicating the room is ready and they can start to play the game. Client 1 starts to make a move by sending the MOVE protocol to ClientHandler1. ClientHandler 1 processes this protocol and does a move on to the game in the room. When the move is successfully completed, the game room notifies both ClientHandlers and these ClientHandlers to send the MOVE protocol back to the clients. Afterward, Client 2 makes a move and the same process repeats until the game is over. When the game ends, the game room would notify both clients with a GAMEOVER protocol.

### 3.8. Activity Diagram



### 3.9. State Machine Diagram of the Game



A user starts the game, the user basically starts to run the StartPage GUI. When the GUI starts to run, the GUI uses the ClientService class to send a HELLO message to the server, requesting to connect to the server.

The server receives the connection HELLO request from the server and connection established. The server sends back the HELLO protocol to notify the client the connection is established. As the client receives the connection established HELLO message from the server, the StartPage GUI prompts a message dialog to indicate user that they can enter their credentials now. While both server and client are keep running.

The user enters their credential. GUI passes these information to the client and the client pass these information in terms of a Mission object to the server. The server checks the credential and send back Protocol LOGIN, NEWPLAYER, ALREADYLOGGEDIN, OR WRONGPASSWORD. if the username exists in the system file and the password entered by the user is correct, the LOGIN protocol would be sent back to the client. And the StartPage GUI prompts "Welcome back!" on the frame. Otherwise, the server sends back WRONGPASSWORD protocol and " Wrong Password and Username combination". If the username is already in the ClientOnline hash map, then the server sends back "ALREADYLOGGEDIN" protocol to indicates that user is already logged in and corresponding information will be prompted in the GUI. If the server discovers that the username entered by the player is not in the system, the server would hash and salted the password entered by the user and store in the system file called "nameAndPassword". After the user has successfully logged in, the StartPage GUI frame is disposed, and the

user is redirected to the ChooseMode GUI. In this page, user can either choose to wait to be invited by other players, choose a specific player to play with, play with the computer(easy mode), play with the computer(hard mode) or exit.

If user choose to wait to be invited by other users, the client will send a WAITING protocol to the server. When server receives it, the client would be put into a collection of waiting players by using a method. The ChooseMode GUI is disposed and the WaitingGUI is created.

If user choose to play with random players, the client sends a WAITINGLIST protocol to the server, asking for all the waiting players. If there is no waiting players, the ChooseMode GUI prompts a message to notify user that there is no waiting players. If there is at least one player waiting to be invited, the ChooseMode GUI is disposed and a Waiting GUI is created. The client sends QUEUE protocol along with a game mode "random" to the system along with the username of one of the waiting users. When the server receives the QUEUE message, the server gets the socket of the waiting player chosen by the client and send a REQUEST protocol. The waiting player receives the REQUEST message and a confirm message will pop up on the Waiting GUI to ask user to accept or reject the request. If the chosen waiting player decides to accept the request, its client will send a ACCEPT protocol to its clienthandler and its clienthandler will obtain the socket of the requesting player and send an ACCEPT protocol along with the game room number. In the meanwhile, the clienthandler of the chosen waiting player will create a new game room and put the chosen waiting player as the black disc holder in the game. The newly created game room will be put into a game room container. The requesting user receives the ACCEPT protocol and add itself to the game room to start a new game. At this point, a PlayWithPlayer GUI is created and the Waiting GUI of both parties are disposed. If the waiting player decides to reject the request, its client will send a REJECT protocol to the clienthandler and its clienthandler will obtain the socket of the requesting player and notify them that they are rejected. The client of the requesting player go through all the players in the waiting list until there is one player accept their request or the list is run out. If all players on the list are asked, the Waiting GUI prompts a notification to indicate there is no more available waiting users.

If the user chooses to play with the specific player, the process is similar as above. A ChooseOpponent GUI is created and the GUI display all the waiting players. The list is obtained by client sending and receiving WAITINGLIST protocol to the server. The user can

press rank list to see the rank of each player. The rank list is obtained by the client sending RANK protocol to the server. After the user choose a waiting player, the QUEUE protocol is sent along with the game mode “specific”(which is an attribute of a Mission object) the waiting player can either accept or reject the request. Then, a new game starts. If the user choose to play with a computer, the QUEUE protocol is sent along with the game mode “computer” over to the server. A PlayWithComputer GUI will be created and the player can start to make move on the board by clicking on the button.

## 4. Concurrency Mechanism

### 4.7. General Prevention

First of all, the OthelloServer class implements the Runnable interface and is a thread with a while loop when it is run by ServerFrame. This ensures that the server keeps running and accepting connection requests from the client. When the server accepts a connection request from the client, a ClientHandler thread is created and started for a specific connection. This ensures that the information from each client thread is handled by one and only corresponding ClientHandler thread. Thus, the information will not be miscommunicated between the server and the client.

Same rule applies to the client side, the Client Service class implements the Thread, thus, once the connection with the server is established and the credential is verified, a Listener thread is created for that specific user.

In both the ClientHandler and Listener threads created, the username of that corresponding thread is stored in both threads. In addition to that, every time a ClientHandler and Listener thread is created, they are added into a container.

### 4.8. Prevent Concurrency Issues for Choosing Random Player to Play

To prevent a player to ask multiple players to play a game at the same time and all of them accept the request at the same time, I created an attribute in the Listener thread which is called askedWaitingPlayers. It is an Array List of the type String. It stores the name of the waiting player that has been asked. Every time the client asks a waiting player, the player's name is added to the list. This prevents the client to ask all the waiting players to play at the same time using a for loop.

#### 4.9. Prevent Concurrency Issue for Choosing the Same Waiting Player

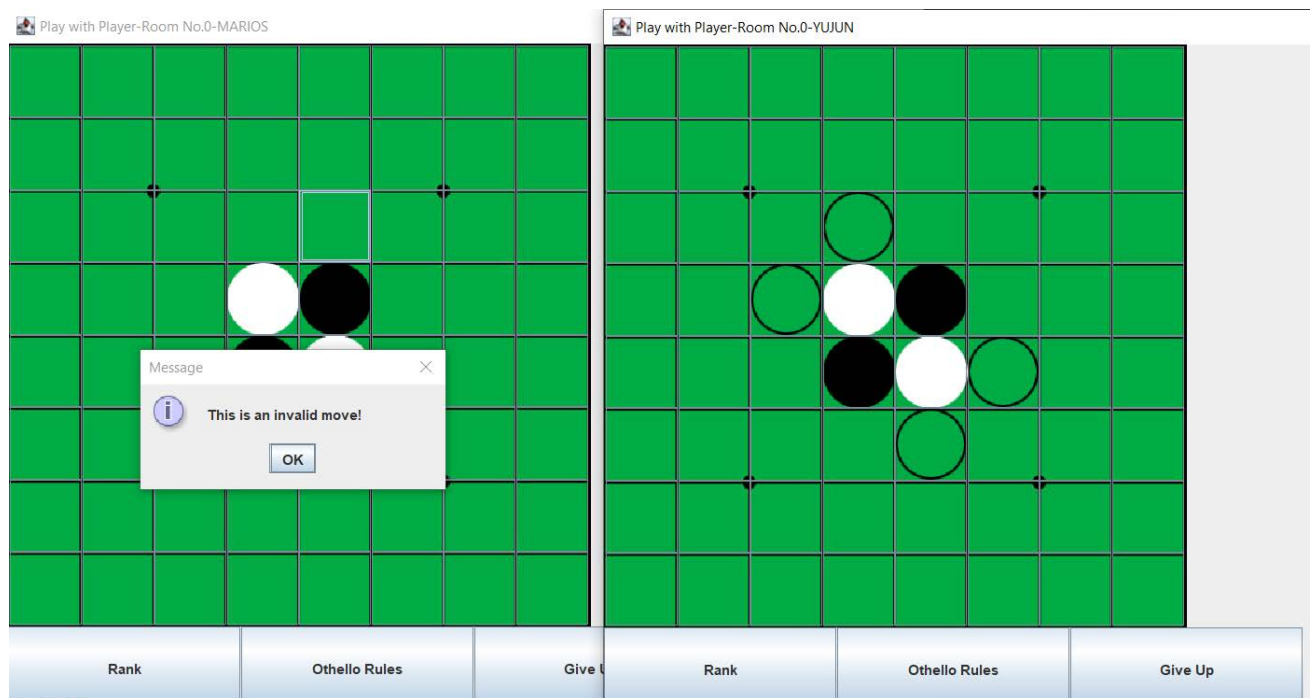
To choose a specific player to player, there might be possibilities that the waiting list of one player is not up to date and the user chooses a player that already accepted other people's invitation to play a game. Also, there might be the possibilities that both player send an invitation to the same waiting player, and the waiting player accept both player's invitation. This problem can be avoided by checking whether a waiting player is in one of the game room created. If the player is in one of the game room, the waiting list player server will not create a new game room but sent a BUSY protocol to the other players that are not in the same room but invited the waiting player.

#### 4.10. Prevent Concurrency Issue for Making Move

The GameRoom class is designed to ensure that only players who belong to a game room can access and make moves on the game in the game room. This prevents other clients that are not in a game room from making changes to the game. Also, as a game room only allows two players to be inside, this prevents more than two players to play the same game. Moreover, the OthelloGame class records the turn of the game. If a client makes a move but the name of the client does not match the name stored in the turn field in the OthelloGame class, this move will be denied and not be executed. Furthermore, concurrent HashMap is used in both GameRoomContainer and the ClientPool container. Concurrent HashMap is thread-safe. This prevents concurrency issues when

multiple threads try to access them at the same time.

In addition to this, the design of GUI prevents a player to that is not supposed to play to make a move to confuse the server. As can be observed from the above picture, only the



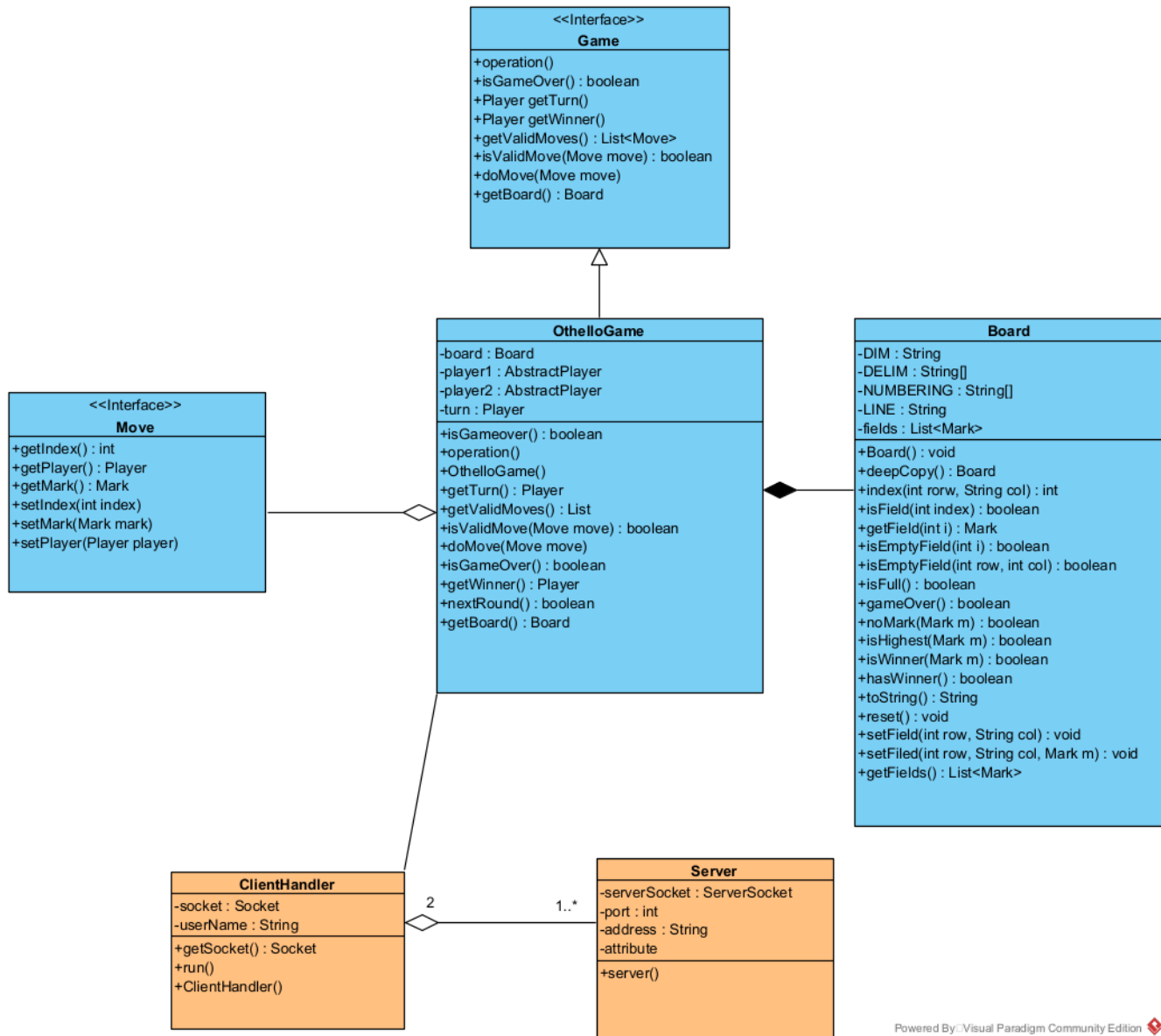
player that has the turn to play has the visual cue of possible moves on their board. Any attempt to click on the button that does not have the hollow circle, the system will prompt it is not the valid move to prevent the user to make a move before they receives a move.





## 5. Reflection on Initial Design and Final Design

### 5.7. Overview of the Initial Design



Class Diagram of Initial Design

More detailed information of Initial Design can be found in the Initial Design report.

## 5.8. Pros and Cons of the Initial Design and Final Design

### 5.8.1. Pros and Cons of the Initial Design:

I am not entirely sure which initial design is asked for the repair project: the initial design of the very beginning or the initial design of the final project I submitted in module 2. Thus, I will elaborate on this part based on the very initial design of mine.

The initial design provided a general idea and an organized structure of classes for future development. The six classes of the initial design are kept and expanded in the final design, so as to have two packages for the program.

However, as the initial design was finished at the beginning of the project, there are plenty of functions of the server and the Othello game was not covered.

### 5.8.2. Pros and Cons of the Final Design:

The structure of the game and the server are more clear and more logical in the final design. The functions that were missing from the initial design are extended and the connection between each class is clear.

Nevertheless, it can be confusing to understand the functions of ClientHandler and the Listener class as there are many methods corresponding to each protocol received by the Listener and the ClientHandler. Also, plenty of if-else statements are used in the ClientHandler class, which can be hard to debug and extend functions.

However, I would say the logic of the game and the network system is much clearer than the initial design. With the help of the GUI, the flow of the game is much clearer and each class serves a clear objective.

## 5.9. Improvements in Final Design Compared to the Initial Design

### 5.9.1. Improvements on the Board Class

In the beginning, my knowledge of the rules of Othello was limited. Consequently, I did not have a clear and logical way in my head of checking if a move is valid or over-flanking discs correctly. In addition to this, I was uncertain about the way of describing the board and storing the content of fields. I initially desired to express the board by using a HashMap and express each field as

coordinates with letters and numbers like A1. However, after trying multiple times, I discovered the most efficient way that matches my design idea of the game is to represent fields from 0 to 63. Also, the game logic of my previous project (final project of module 2) was completely wrong. Thus, I changed the game-over condition in this project.

#### 5.9.2. Improvements on the OthelloGame Class

In the initial design, there were many aspects of the game logic were overlooked. Hence, compared to the initial design, there are more methods added to this class such as `flipDisc()`. These improvements ensure that the functions of the game are complete. Improvements in the HumanPlayer Class

At the beginning of the project, because of insufficient knowledge of project requirements and my uncertainty about how to actualize the game logic, I designed an `AbstractPlayer` class and a `Player` interface based on the experience I had with the Tic Tac Toe game. Then, I discovered that I need another Interface which is AI to generate two different AI modes.

#### 5.9.3. Improvements on the Server Package

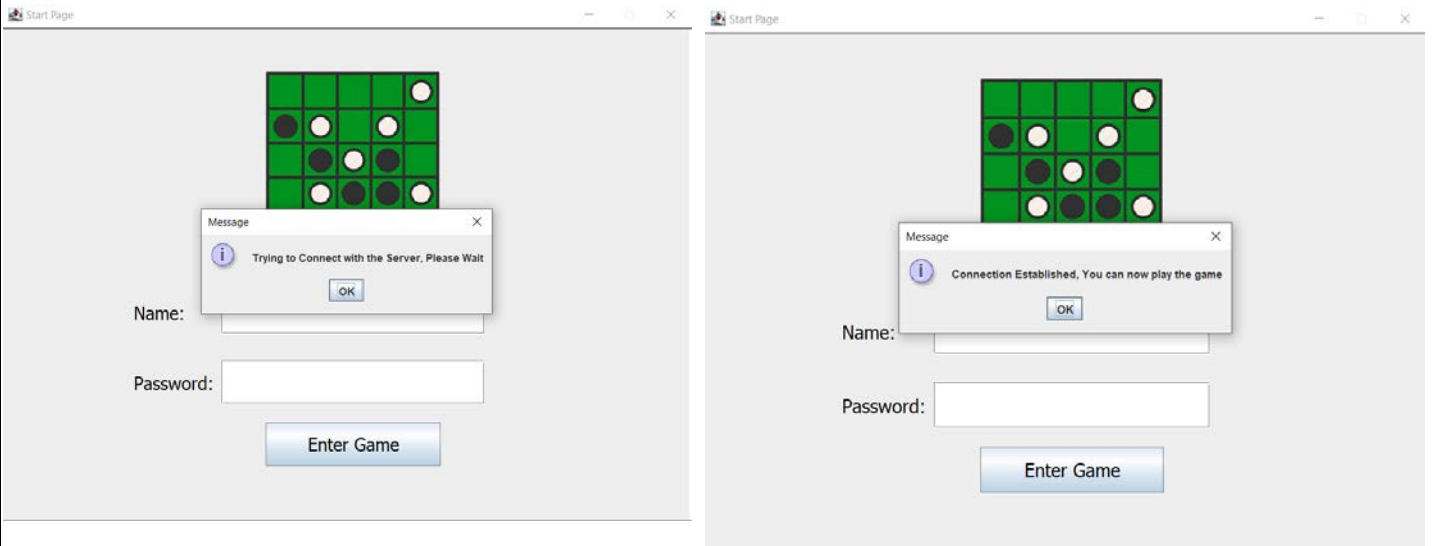
In the initial design, I planned that all messages sent from the client should be processed by the `OthelloServer` class. However, while developing and implementing the server, I discovered that all the actions that need to take to respond to the protocols sent by the client should be dealt with by the `ClientHandlers` thread, not the server. Thus, I moved the majority of the code from the server to the `ClientHandler`.

More features were added to the server package such as `GameRoom`, `ClientPool`, and `GameRoomsContainer`, which were not thought about at the beginning of the project. Adding these classes helped me to deal with the concurrency problems as mentioned in the previous sections.

## 6. Testing

As I have implemented my project by using GUIs and sending objects over the network, it is difficult to test the program using JUnit for server and client interaction. Thus, the testing for server, client and user behavior heavily relies on systematic tests and black box testing.

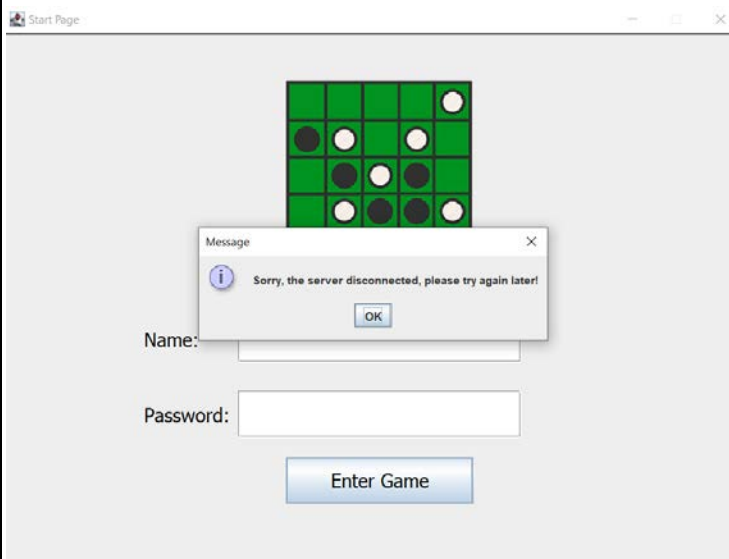
## 6.7. Systematic Testing

Systematic Test	
Test Objective: Establish Connection With the Server and A StartPage GUI shows up	
Testing Steps: <ol style="list-style-type: none"><li>1. run the server</li><li>2. run the client</li></ol>	
Expected Behaviour: <ol style="list-style-type: none"><li>1. connection established</li><li>2. no error is thrown</li><li>3. server and client are running without any error</li><li>4. a successful connection established message pops up on the frame</li></ol>	
Result:	
	
Meet the Expected Behaviour: True	

Systematic Test	
Test Objective: Server disconnect while Client is still running	
Testing Steps: <ol style="list-style-type: none"><li>1. run the server</li><li>2. run the client</li><li>3. disconnect the server</li></ol>	
Expected Behaviour: <ol style="list-style-type: none"><li>1. client stops running and no error is thrown</li><li>2. a warning pops in the frame</li></ol>	

### 3. GUI window closes

Result:



```
Run: ServerFrame x StartPageGUI x
C:\Users\yujun\.jdk\temurin-11.0.17\bin\java.exe "-ja
The server is listening at the Port 44444...
The server is connected to the client
Process finished with exit code 130
```

```
Run: ServerFrame x StartPageGUI x
C:\Users\yujun\.jdk\temurin-11.0.17\bin\j
Connection established
Server is not responding
Process finished with exit code 0
```

Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: Disconnect the Client and no error is thrown

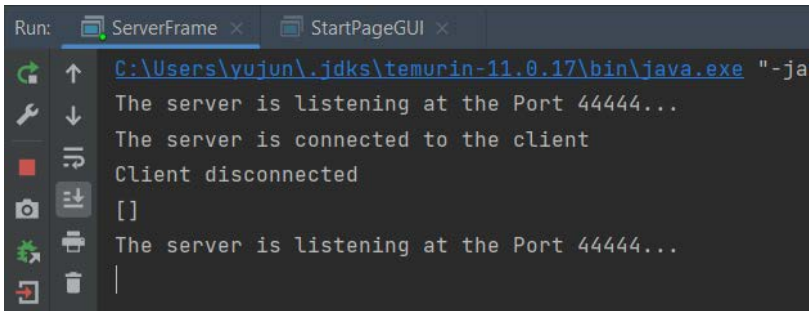
Testing Steps:

1. run the server
2. run the client
3. disconnect the client

Expected Behaviour:

1. the client window is closed
2. there is no error thrown from the server side
3. server is still running
4. there is no client handler in the client handler pool

## Result:



```
Run: ServerFrame x StartPageGUI x
C:\Users\yujun\...jdk\temurin-11.0.17\bin\java.exe "-ja
The server is listening at the Port 44444...
The server is connected to the client
Client disconnected
[ ]
The server is listening at the Port 44444...
|
```

Meet the Expected Behaviour: True

## Systematic Test:

Test Objective: Create a new account

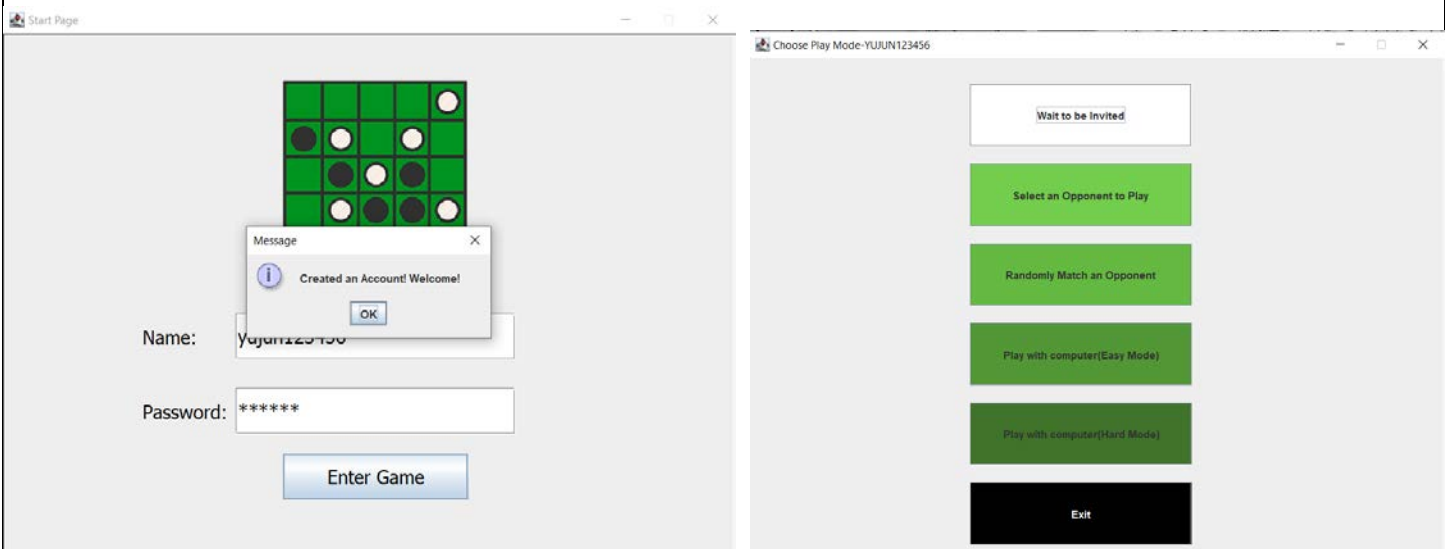
### Testing Steps:

1. run the server
2. run a client
3. enter a username that is not in the system

### Expected Behaviour:

1. the new username and password combination is stored in the file
2. the user and score of 0 is added to the file
3. a message displayed on the GUI
4. a choose mode GUI is created

## Result:



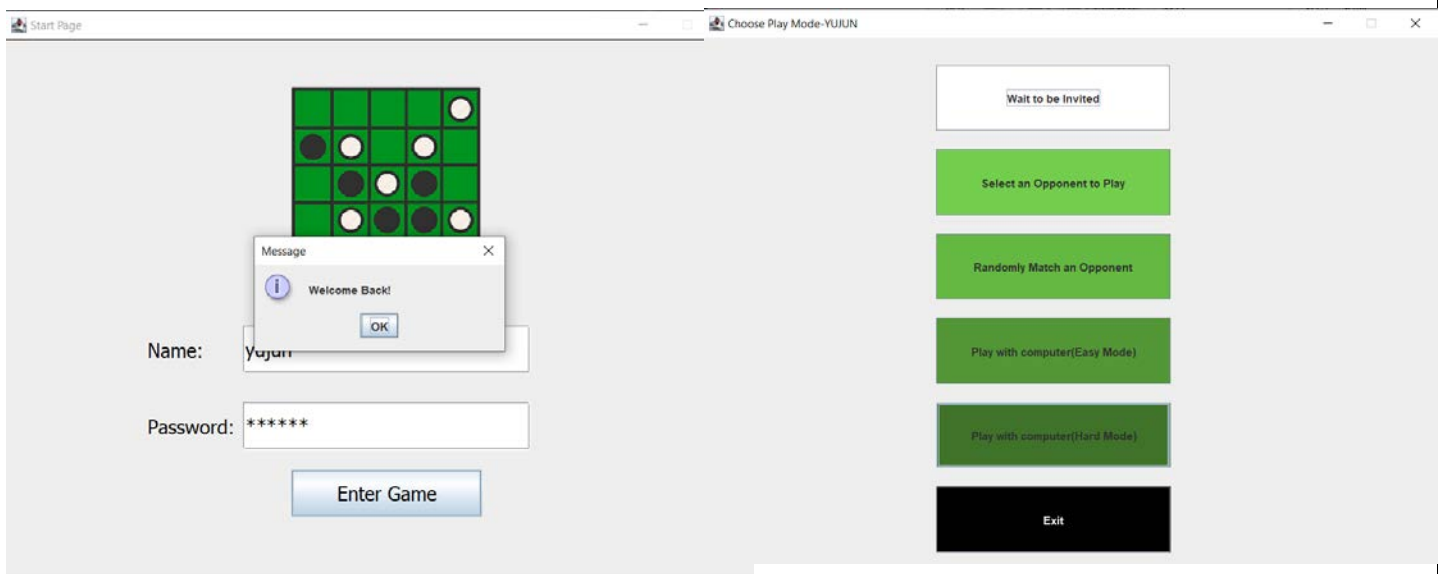
Meet the Expected Behaviour: True

Systematic Test:	
Test Objective:	Login to an account that is already logged in
Testing Steps:	<ol style="list-style-type: none"> <li>1. run the server</li> <li>2. run a client</li> <li>3. login a client</li> <li>4. try to login with the same credential</li> </ol>
Expected Behaviour:	<ol style="list-style-type: none"> <li>1. the behaviour is denied</li> <li>2. an alert message is popped</li> <li>3. the client is not logged in</li> <li>4. restart the start page</li> </ol>
Result:	
Meet the Expected Behaviour:	True

Systematic Test:	
Test Objective:	log into an account that is already in the system
Testing Steps:	<ol style="list-style-type: none"> <li>1. run the server</li> <li>2. enter credential of an user that is already in the system</li> </ol>
Expected Behaviour:	<ol style="list-style-type: none"> <li>1. the client is logged in</li> <li>2. a choose play mode GUI is created</li> <li>3. a welcome back message is prompted</li> </ol>



Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: a player start to wait to be invited

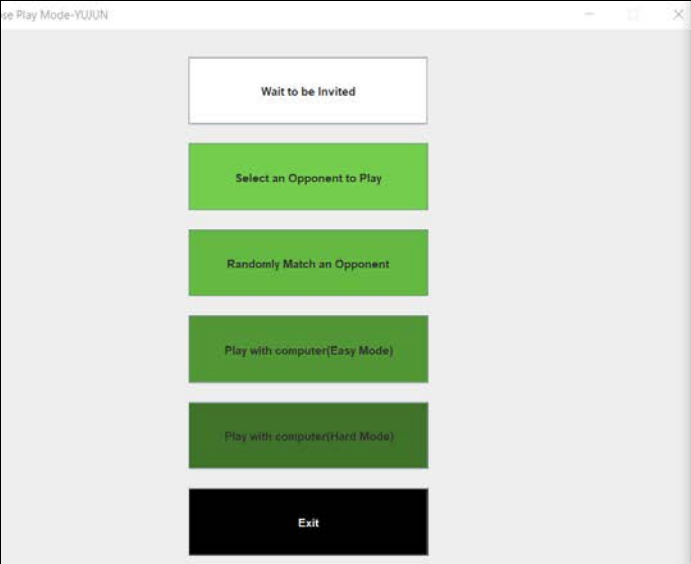
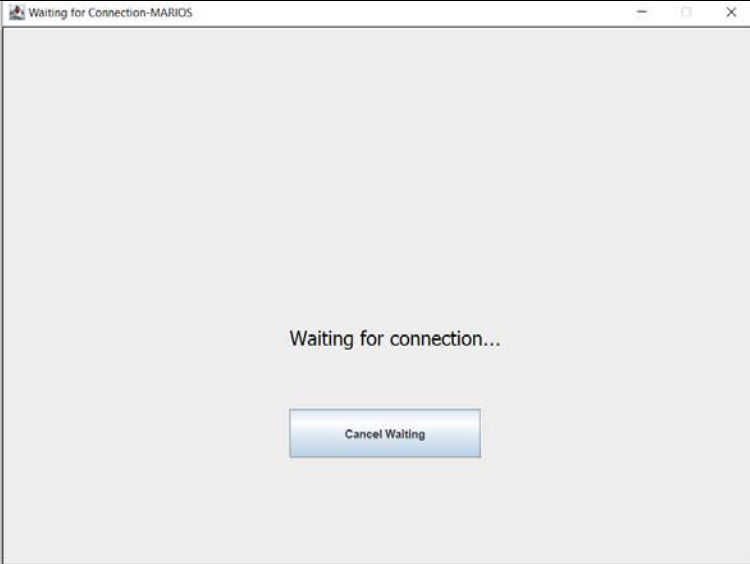
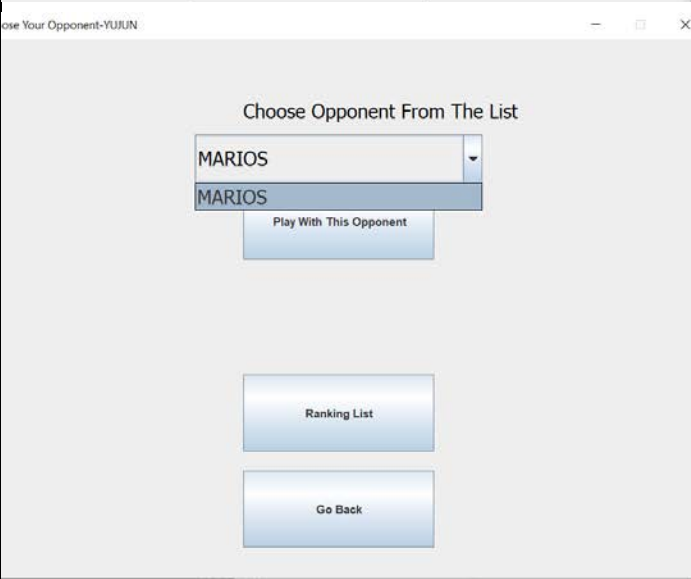
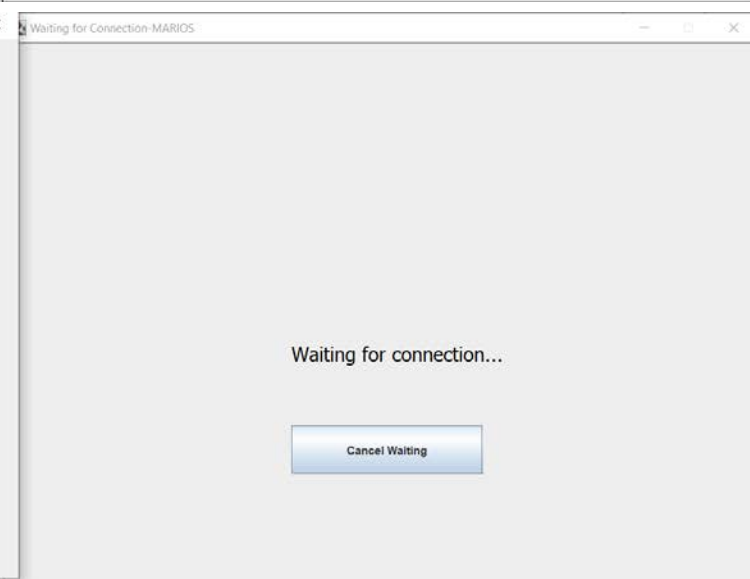
Testing Steps:

1. login two different accounts
2. click wait to be invited button on one account's Choose Mode GUI
3. another account click on Select Opponent to Play

Expected Behaviour:

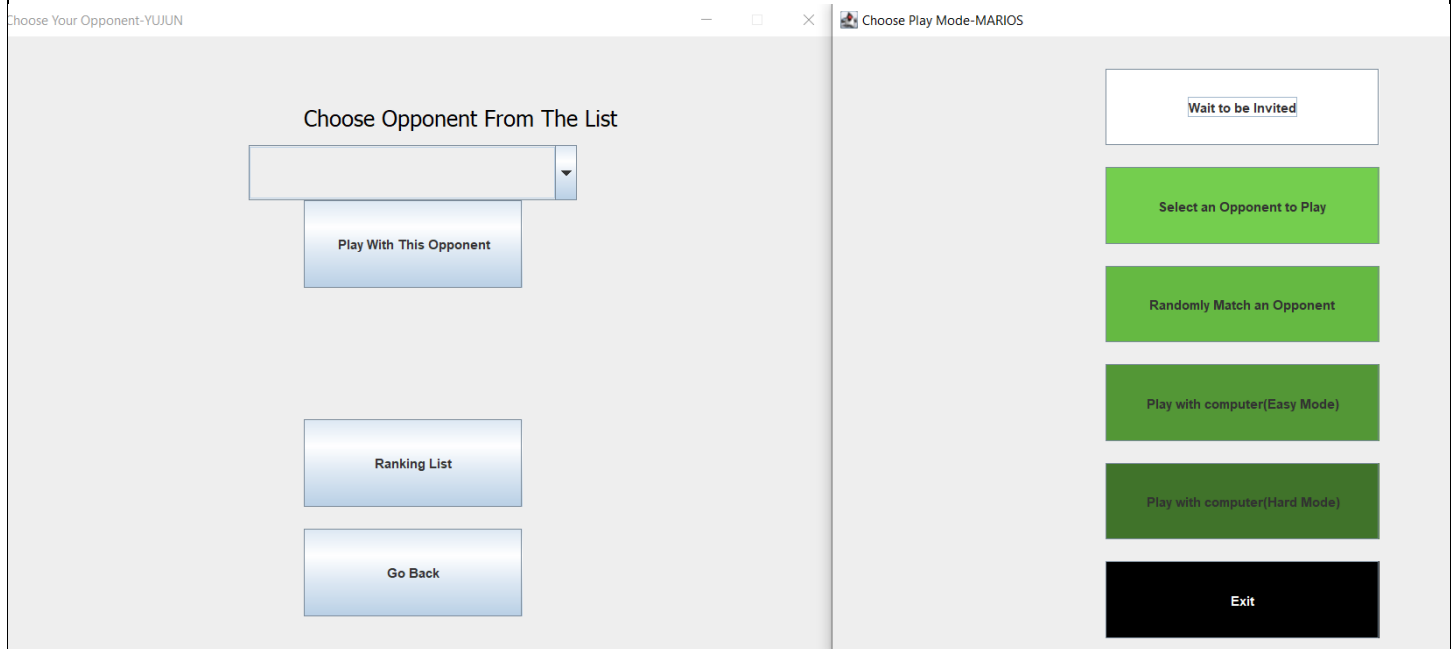
1. on the drop down list, the another account can see the name of the waiting client
2. a waiting GUI should be created

Result:

	
	
<p>Meet the Expected Behaviour: True</p>	

Systematic Test:
Test Objective: a player cancels waiting
<p>Testing Steps:</p> <ol style="list-style-type: none"> <li>1. login two different accounts</li> <li>2. one of the player click on the waiting to be invited button on the Choose Mode GUI</li> <li>3. and then the player click on cancel waiting on the Waiting GUI</li> <li>4. The other user click on Choose Opponent to Play</li> </ol>
<p>Expected Behaviour:</p> <ol style="list-style-type: none"> <li>1. the first player's name should not be on the drop down list</li> <li>2. a Choose Mode GUI is created for the player that canceled on waiting</li> </ol>

## Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: run the client without the server

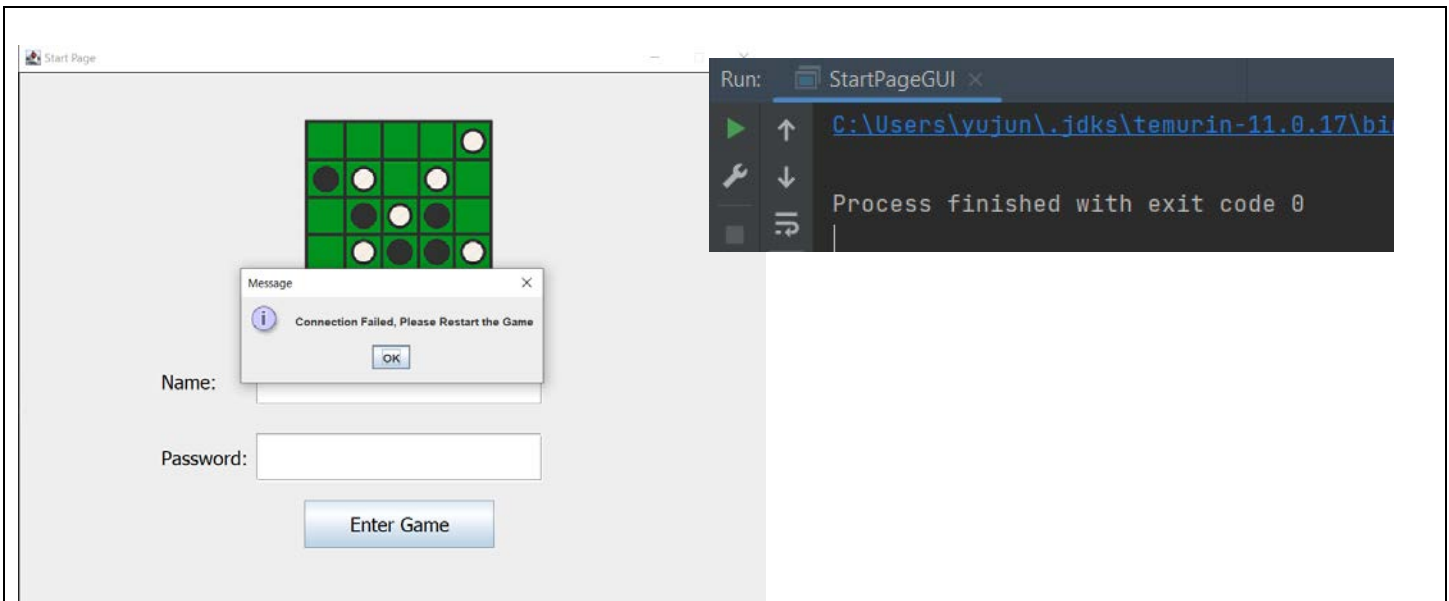
Testing Steps:

1. run the client

Expected Behaviour:

1. a connection is not established
2. a connection failed message appear on the screen
3. the Start Page GUI disposes and the system exits

Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: select randomly play with a player

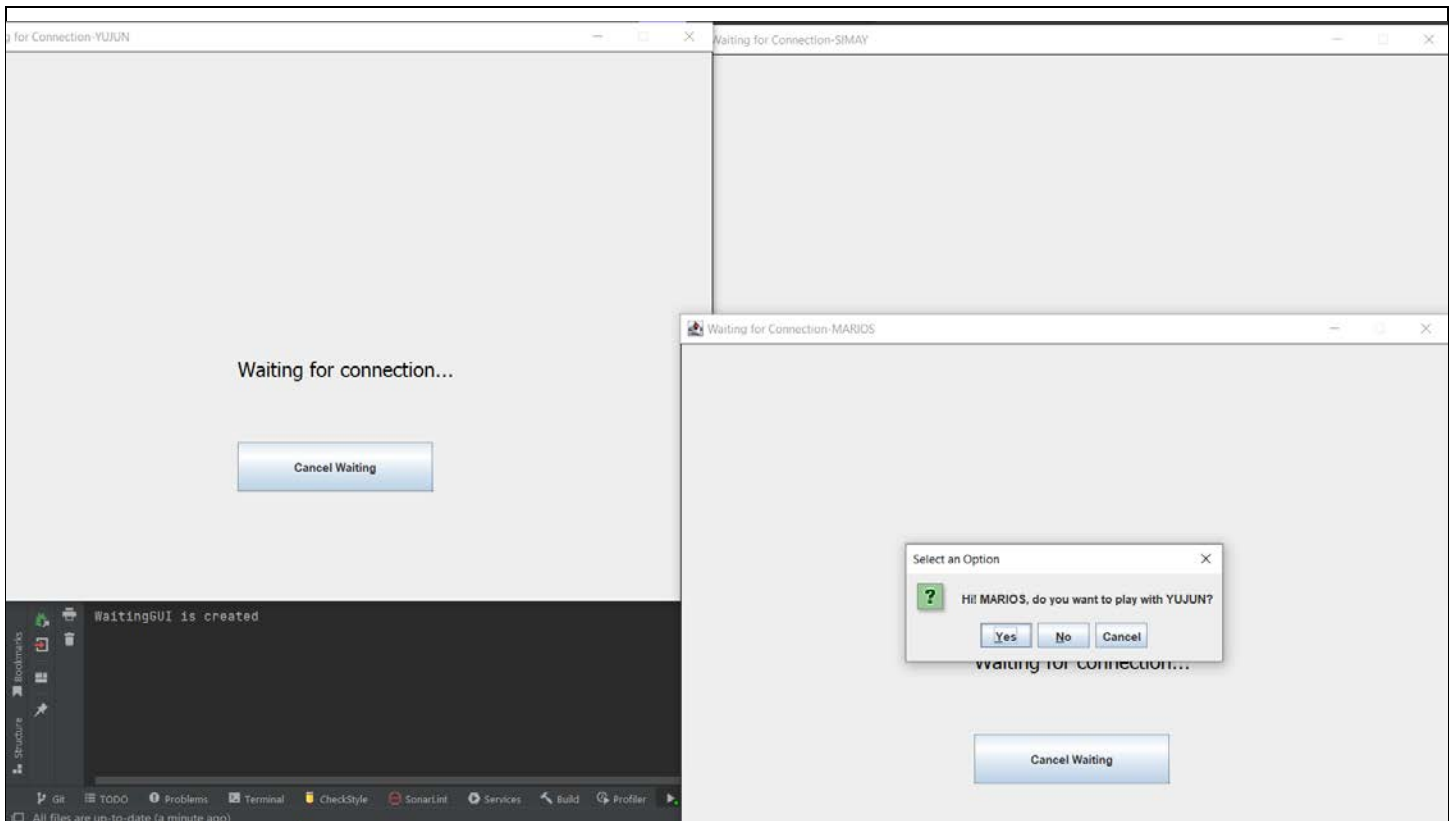
Testing Steps:

1. login to three accounts
2. make two of them waiting to be invited
3. the third one click on the Play With Random player
- 4.

Expected Behaviour:

1. three of them should be in the waiting GUI
2. one invitation should be sent to one of the two waiting players
3. the player should be able to either choose to play or reject

Result:



Meet the Expected Behaviour: True

### Systematic Test:

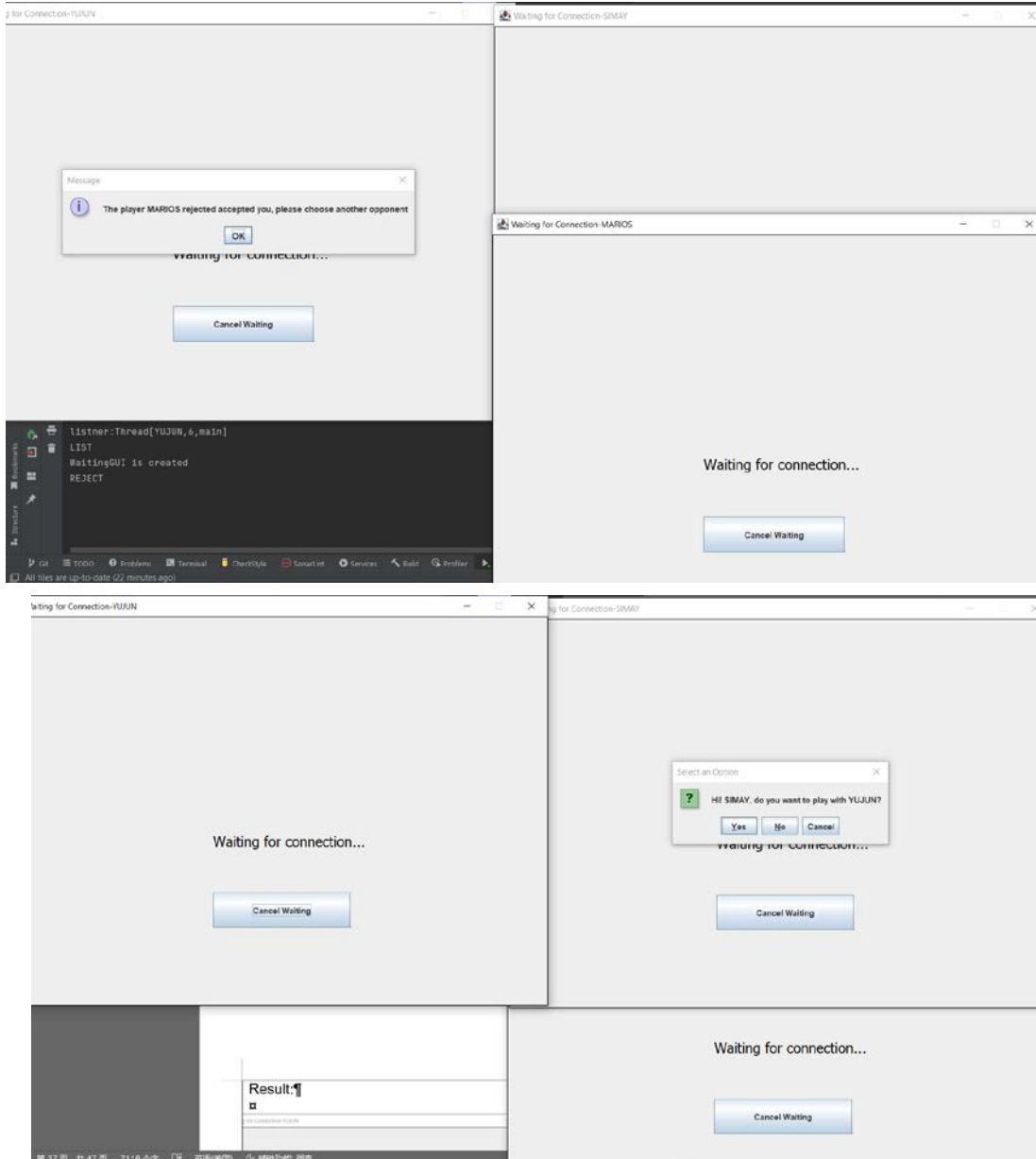
Test Objective: reject an invitation for play with random players

1. Testing Steps: login two different accounts
2. click wait to be invited button on one account's Choose Mode GUI
3. another account click on Select Opponent to Play
4. The first player that receives the invitation rejects the invitation

Expected Behaviour:

1. the player that choose to be play with random player should receives a reject message
2. a new Waiting GUI is created for this player
3. the request message is sent to the other player that is waiting

## Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: all the online users reject the random play request

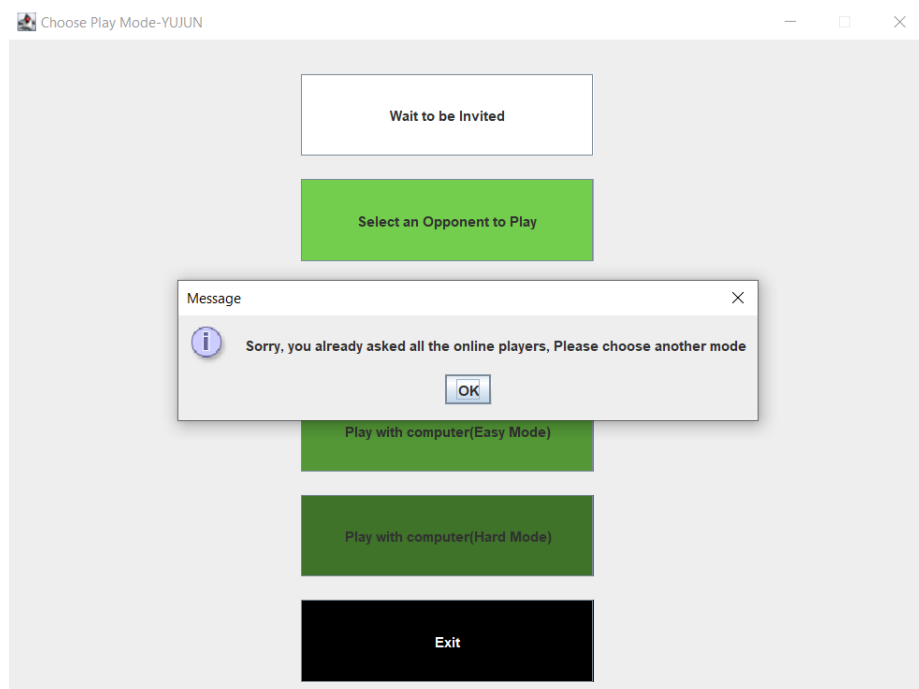
1. Testing Steps: login two different accounts
2. click wait to be invited button on one account's Choose Mode GUI
3. another account click on Select Opponent to Play
4. The first player that receives the invitation rejects the invitation

Expected Behaviour:

1. the player that choose to be play with random player should receives a reject message

2. a new Waiting GUI is created for this player
3. the request message is sent to the other player that is waiting
4. the other player rejects again
5. the requesting player receives a message that indicates that all the online users have rejected you please choose another mode to play

Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: adding a new account after all the waiting players rejects the randomly play invitation

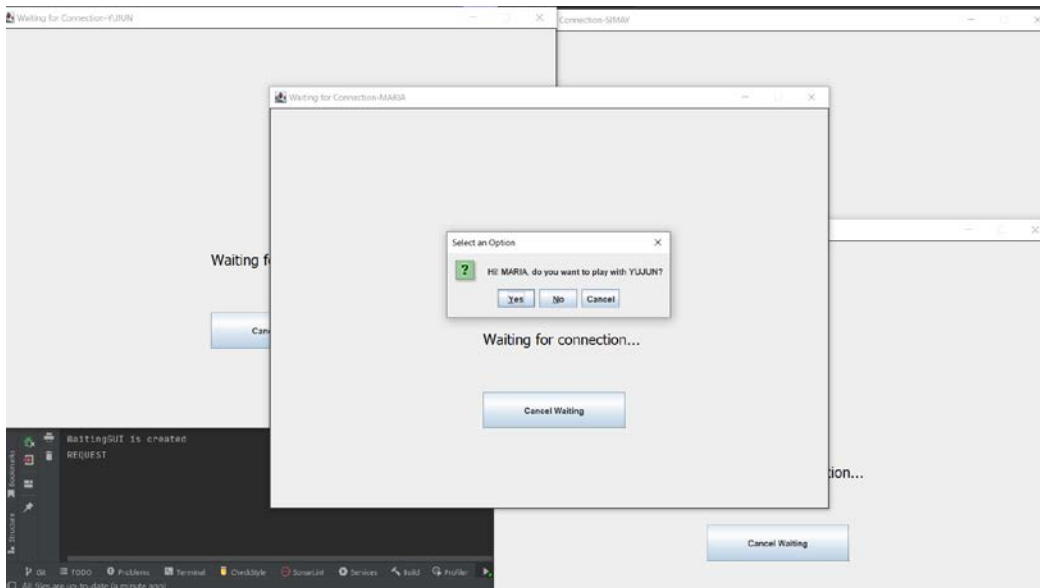
Testing Steps:

1. Testing Steps: login two different accounts
2. click wait to be invited button on one account's Choose Mode GUI
3. another account click on Select Opponent to Play
4. The first player that receives the invitation rejects the invitation
5. the second player that receives the invitation rejects the invitation
6. a new account login
7. the new account start to wait to be invited
8. the requesting player click on the Randomly Play with Players button

Expected Behaviour:

1. No more warning of You have asked all the players
2. a request sent to the newly joined account

Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: accept an play game invitation

Testing Steps:

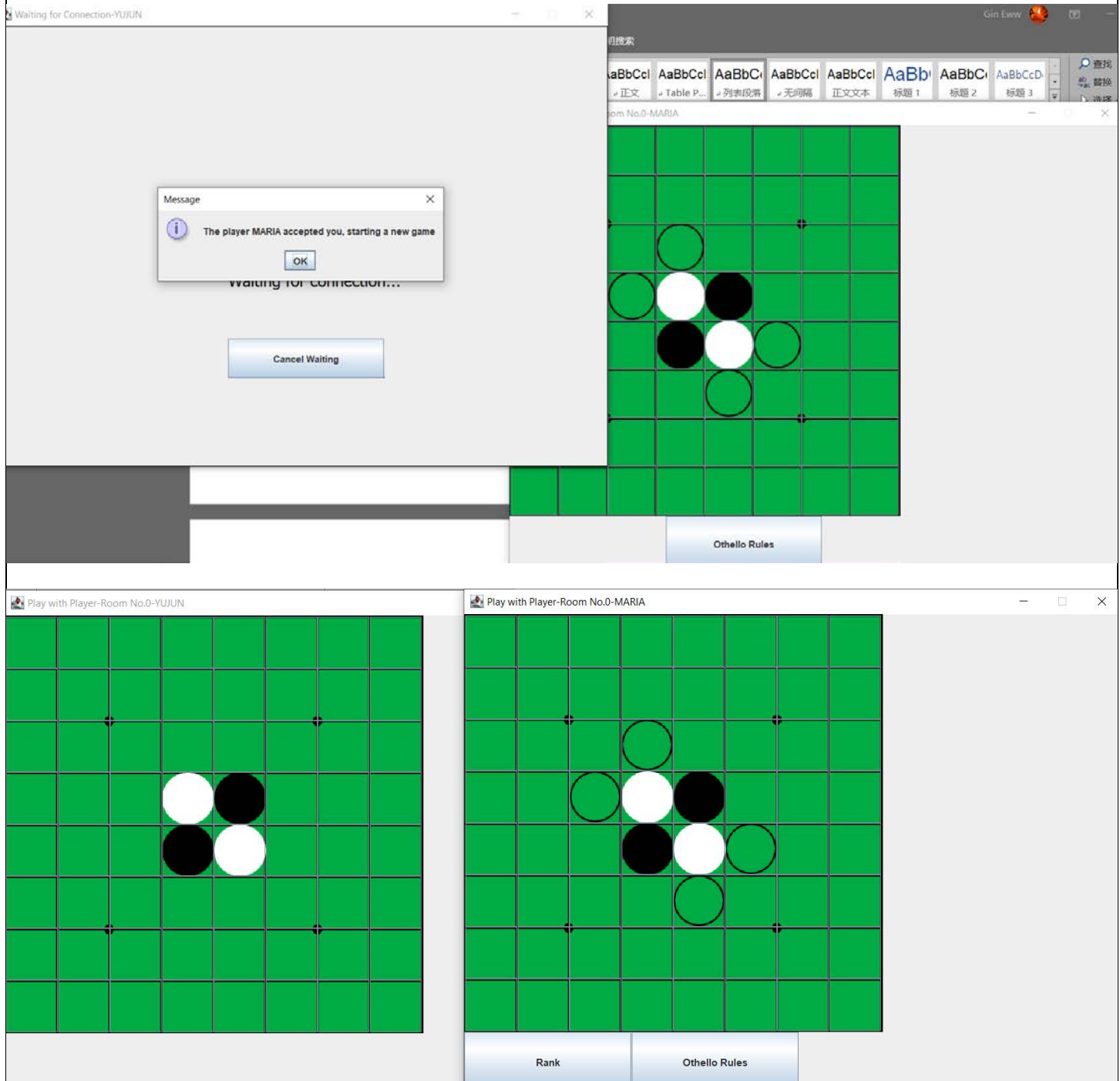
1. Testing Steps: login two different accounts
2. click wait to be invited button on one account's Choose Mode GUI
3. another account click on Select Opponent to Play
4. The first player that receives the invitation accepts the invitation

Expected Behaviour:

1. no invitation is sent to the other waiting player
2. a new game is started for the requesting player and the player who accepted it
3. both players are in the same room
4. a message that the other player has accepted you shows up on the frame



## Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: to check whether the players can make an invalid move

Testing Steps:

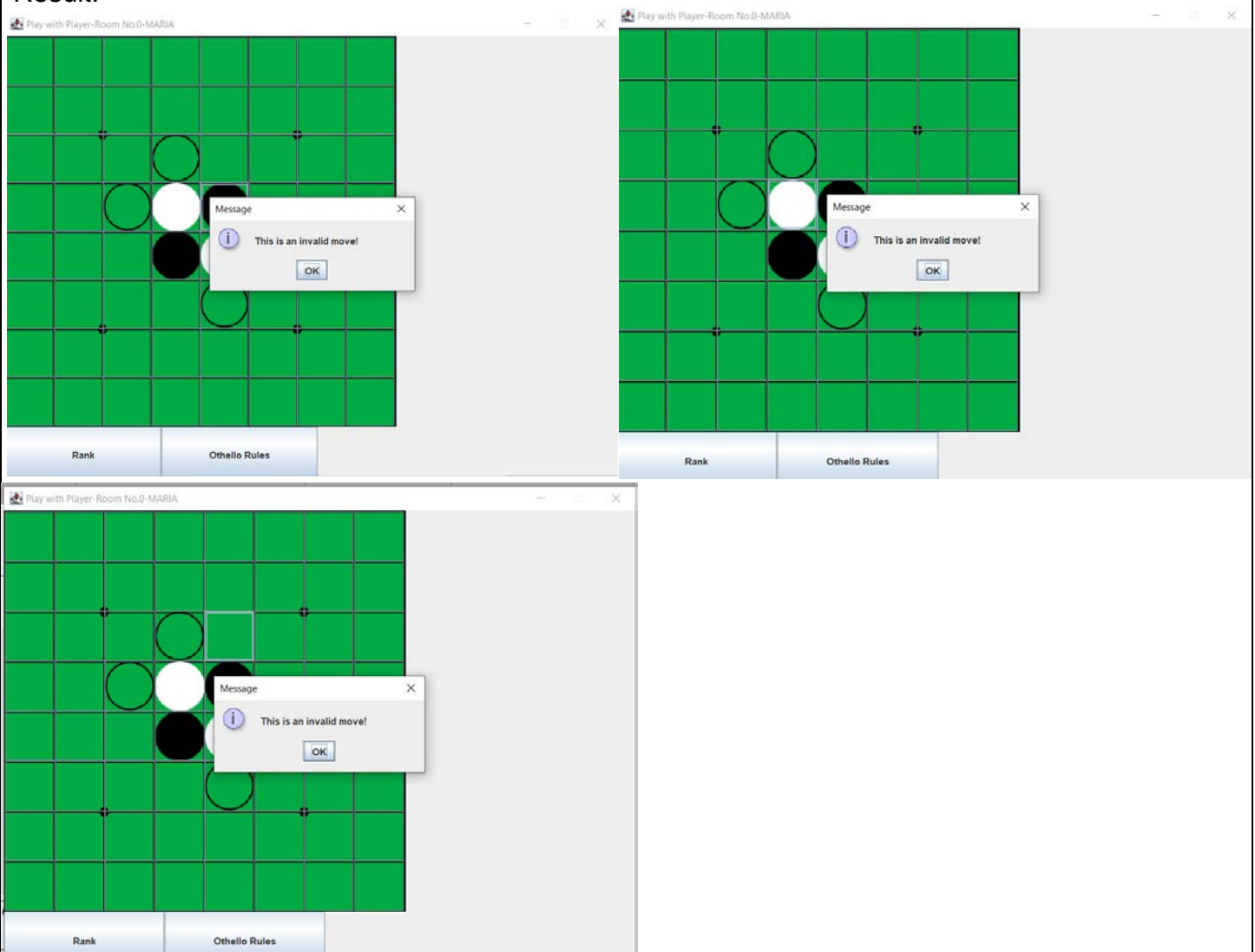
1. Testing Steps: login two different accounts

2. click wait to be invited button on one account's Choose Mode GUI
3. another account click on Select Opponent to Play
4. The first player that receives the invitation accepts the invitation and a game starts between two players
5. The first player tries to make a move that is not valid
6. The game does not send the move and a messages pops up on the screen which says that the move that was made was invalid

Expected behaviour:

1. When an invalid move no move was send to the opponent
2. The turn still remains to the same player
3. The player can still make a move

Result:



Meet the Expected Behaviour: True

**Systematic Test:**

Test Objective: to check whether the moves are send between the players

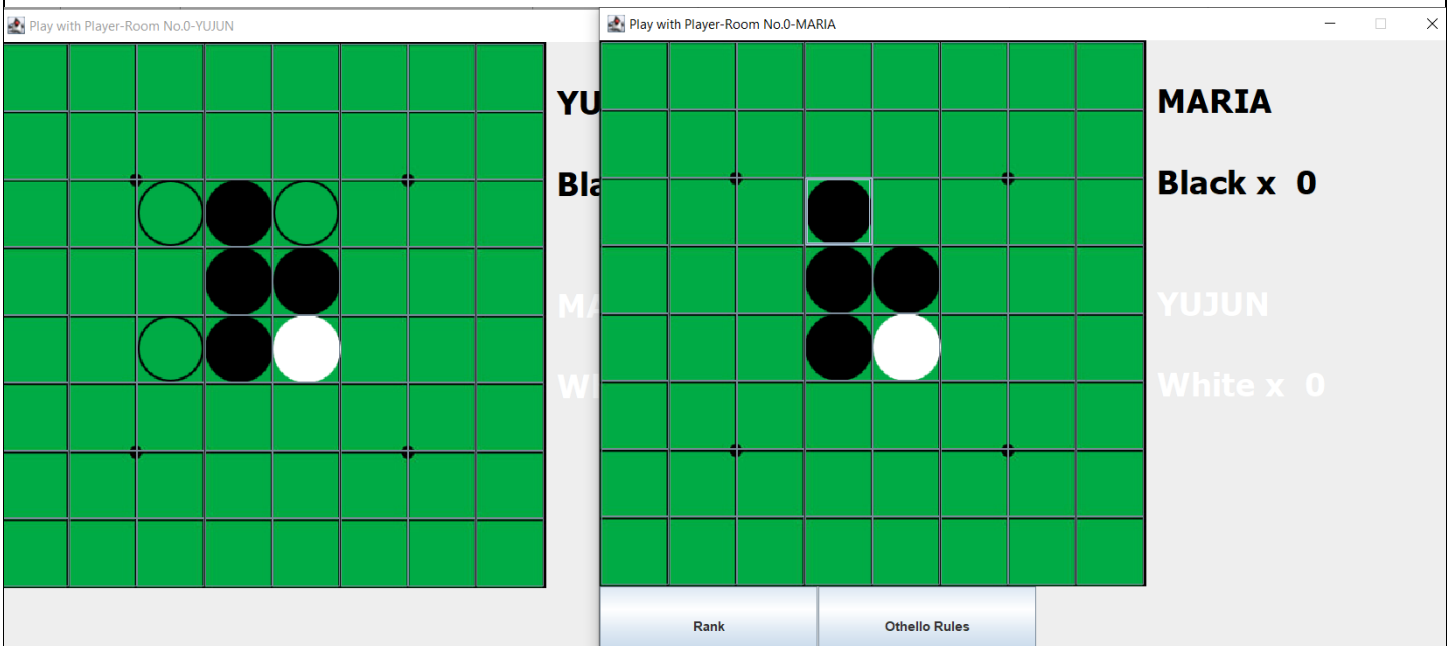
Testing Steps:

1. Testing Steps: login two different accounts
2. click wait to be invited button on one account's Choose Mode GUI
3. another account click on Select Opponent to Play
4. The first player that receives the invitation accepts the invitation
5. A game starts between the two players
6. The first player makes a valid move

Expected Behaviour:

1. The moves are sent between the players
2. The second player receives the move and the first player waits for the second player to make a move
3. Each of them has to wait until to receive a move from the other player to make a move unless they play first

Result:



Meet the Expected Behaviour: True

**Systematic Test:**

Test Objective: to check whether if there are no possible moves for one player, the turn will be given to other player

Testing Steps:

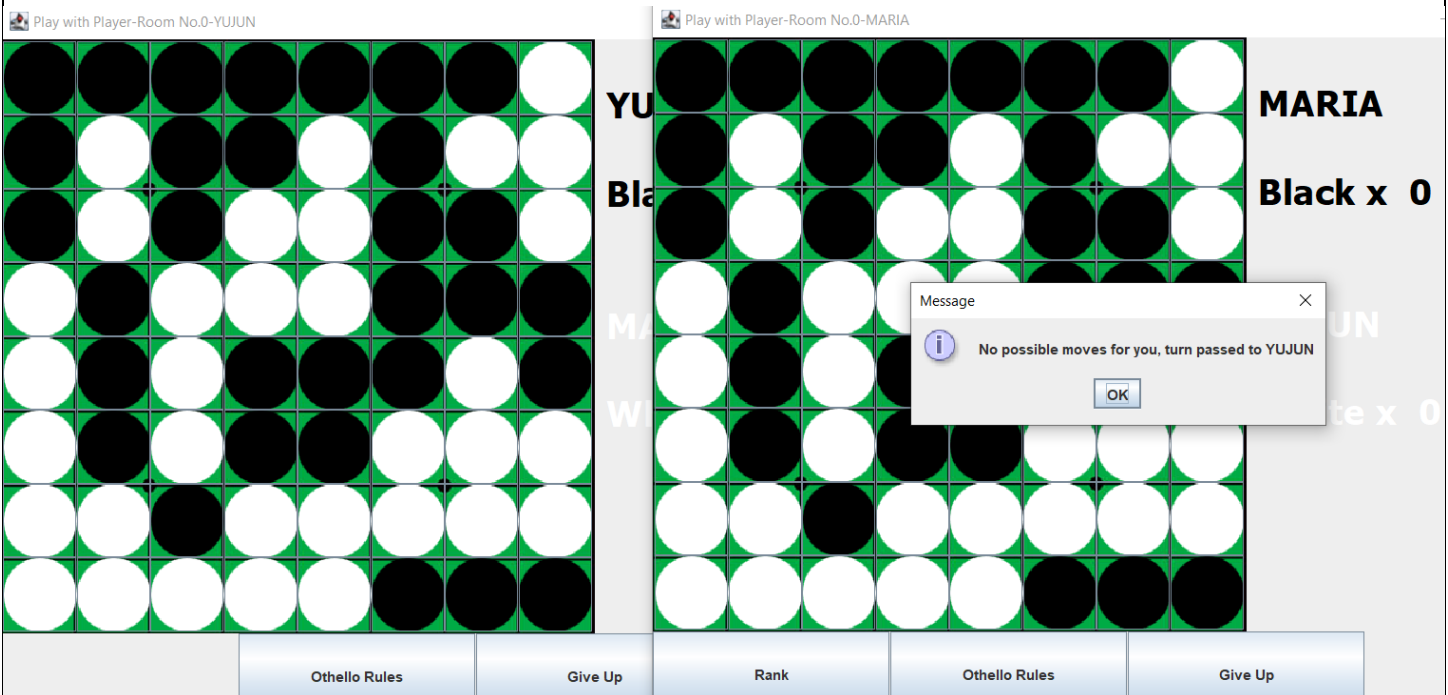
1. Testing Steps: login two different accounts
2. click wait to be invited button on one account's Choose Mode GUI

3. another account click on Select Opponent to Play
4. The first player that receives the invitation accepts the invitation
5. A game starts between the two players
6. The players play until one of them does have any other possible moves
7. The other player will be given the turn to play

#### Expected Behaviour:

1. If there are not possible moves the turn is passed to the other player
2. When a player does not have more possible moves a message will pop saying that the player does not have move to play

#### Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: to verify the game condition functions correctly

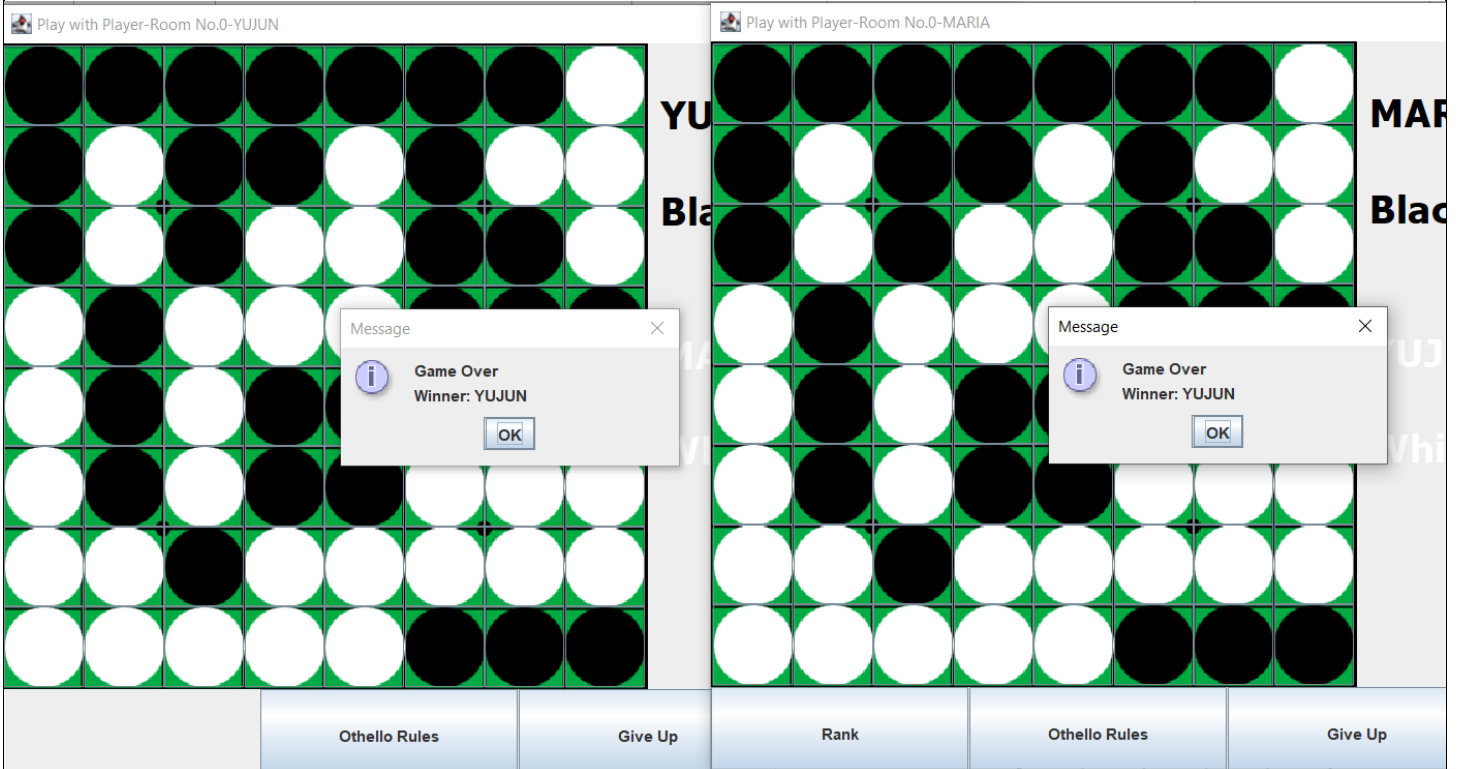
#### Testing Steps:

1. Testing Steps: login two different accounts
2. click wait to be invited button on one account's Choose Mode GUI
3. another account clicks on Select Opponent to Play
4. The first player that receives the invitation accepts the invitation
5. Game starts between players
6. They play until one of them reaches a state in the game where they win

#### Expected Behaviour:

1. If there are not valid moves for both players then game over will pop up
2. The winner will be the one with most marbles on the board
3. Messages pops up to both player that the game is over and specifies the winner of the game

#### Result:



Meet the Expected Behaviour: True

#### Systematic Test:

Test Objective: to check whether the no button on do you want to play another game button message works

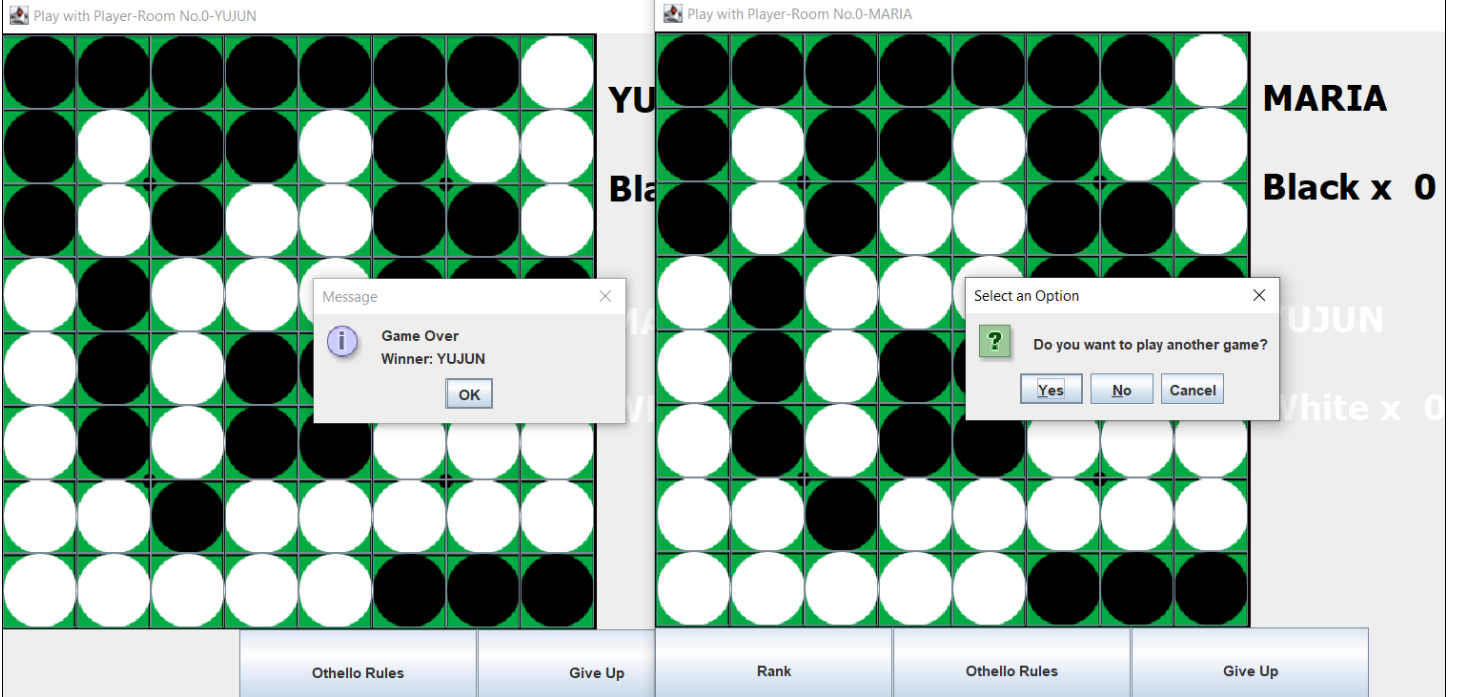
#### Testing Steps:

1. Two players play the game until the game is over
2. A message will pop to both players whether they want to play another round
3. The second player presses no

#### Expected Behaviour:

1. The Game Over message is sent to the server
2. The game is terminated

## Result:



Run: ServerFrame x StartPageGUI x StartPageGUI x StartPageGUI x StartPageGUI x

MOVE  
MOVE  
MOVE  
MOVE  
MOVE  
MOVE  
MOVE  
MOVE  
MOVE  
MOVE  
GAMEOVER

Process finished with exit code 0

Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: to check whether the yes button on do you want to play another game button message works

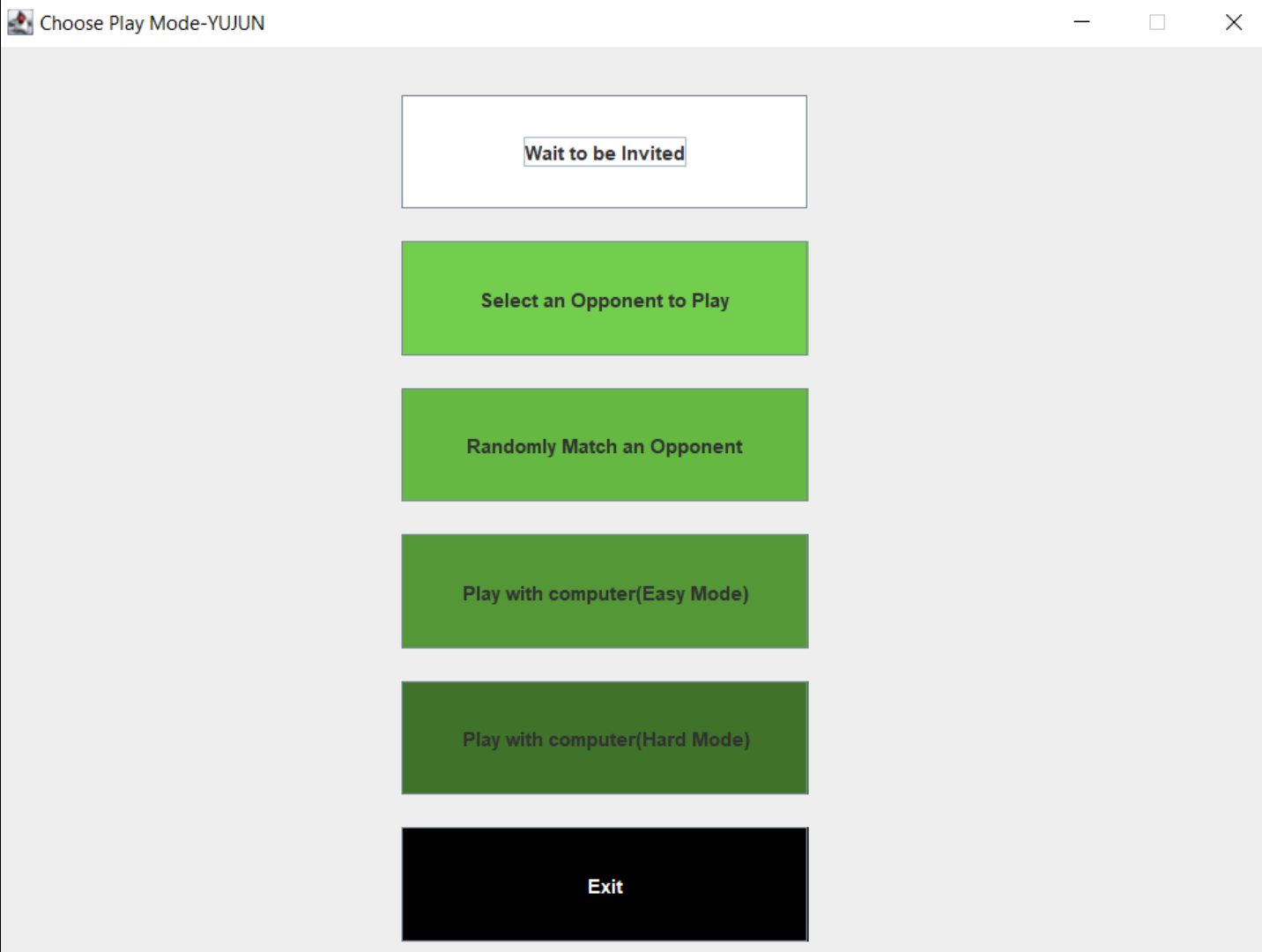
Testing Steps:

1. Two players play the game until the game is over
2. A message will pop to both players whether they want to play another round
3. The second player presses yes

Expected Behaviour:

1. The second player will be directed to Choose Play Mode GUI

Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: to test if a player can invite another player into a game

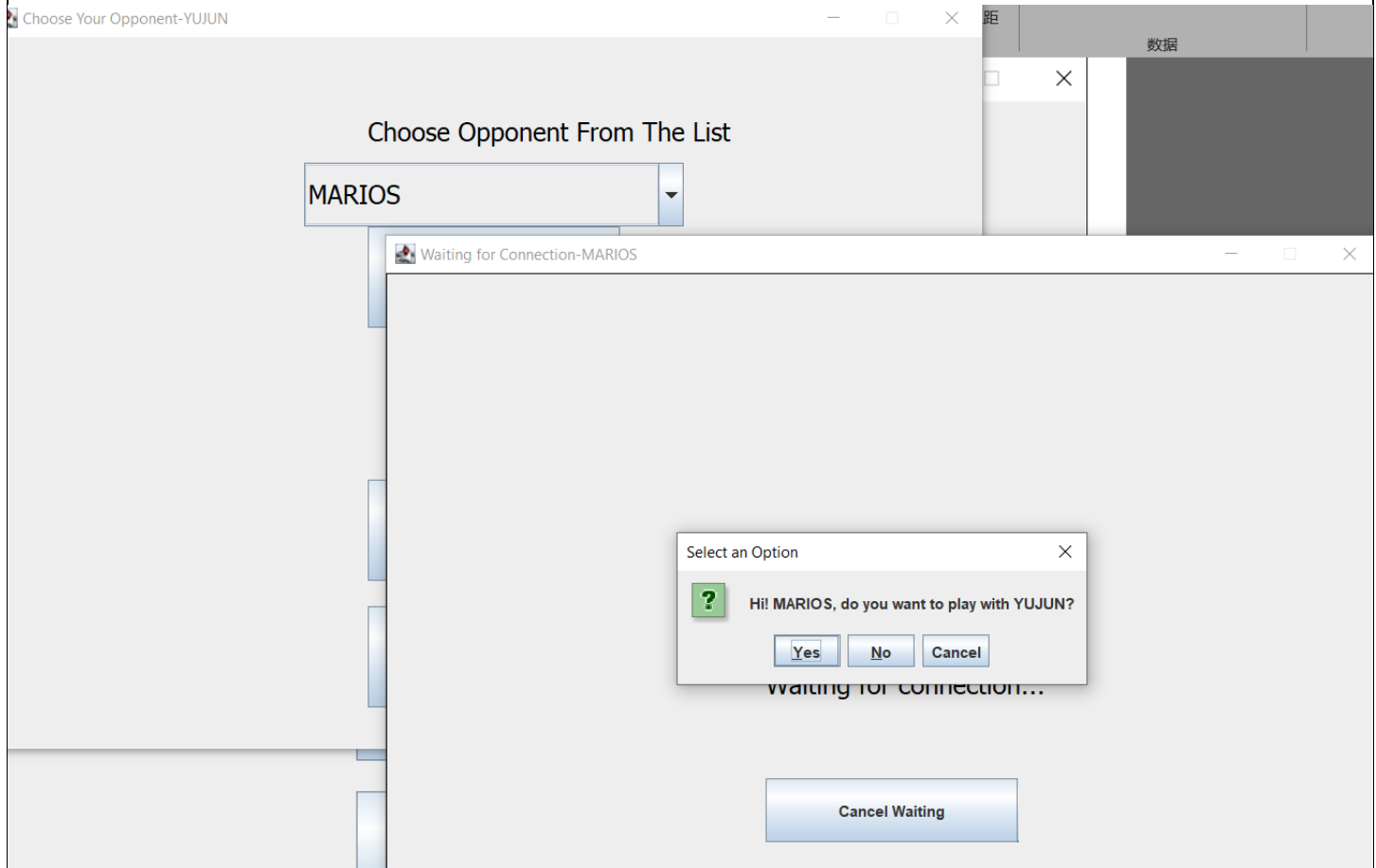
Testing Steps:

1. Testing Steps: login two different accounts
2. one of the players click wait to be invited button Choose Mode GUI
3. the other player presses select an opponent to play button and selects the other player from the list

### Expected Behaviour:

1. the player that was originally waiting will receive an invitation from the other player and has three options to accept reject or cancel the invitation

### Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: to test whether the ai works

#### Testing Steps:

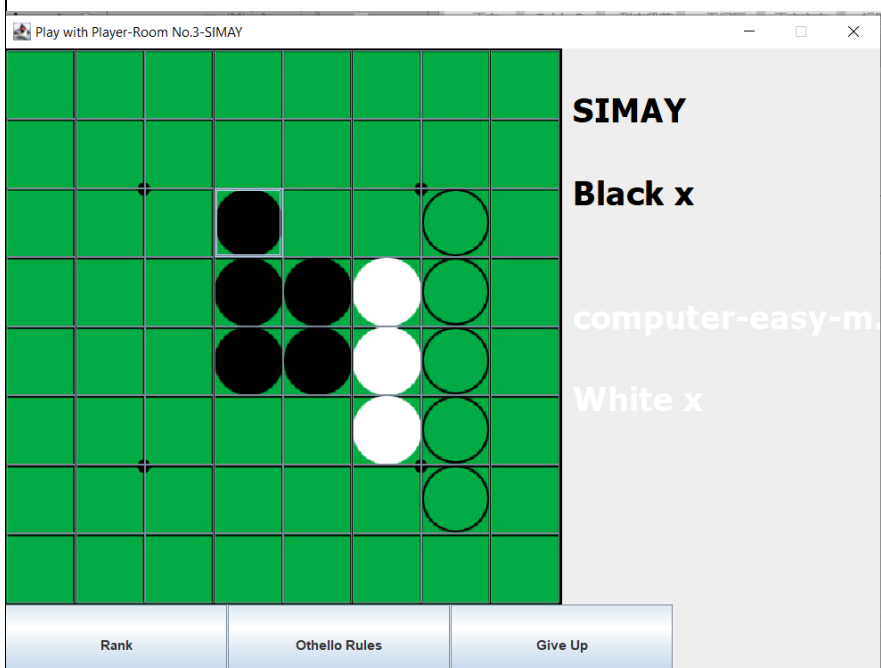
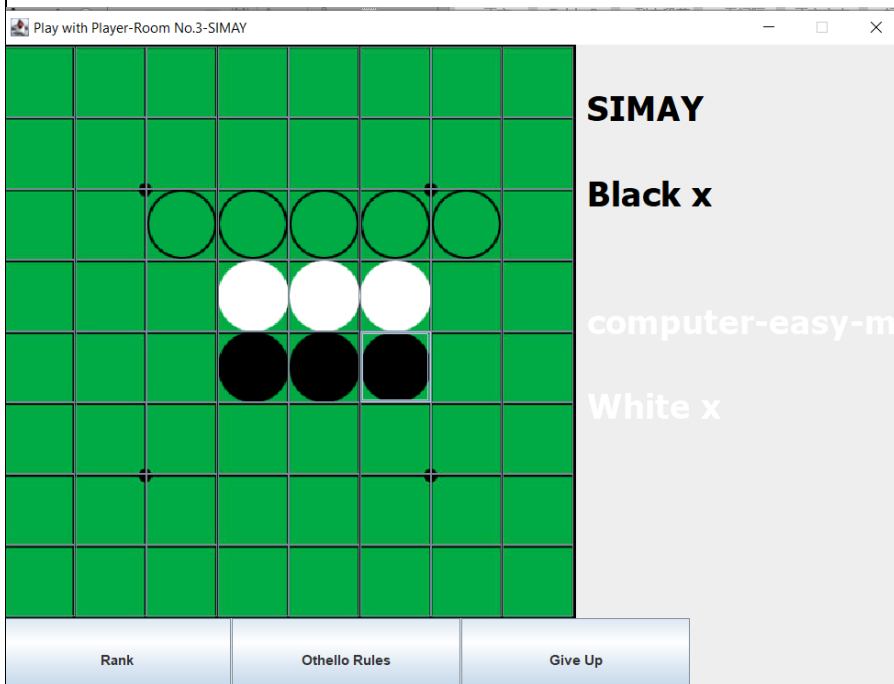
1. Testing Steps: client logs in the game
2. client is directed to Choose Mode Button
3. the clients presses play with computer easy mode
4. a game starts between the client and the computer
5. the clients makes a move



### Expected Behaviour:

1. When the clients makes a move the ai plays the move instantly
2. The turn is back to the client without having to wait

### Result:



Meet the Expected Behaviour: True

### Systematic Test:

Test Objective: to test whether the othello rule button works during the game

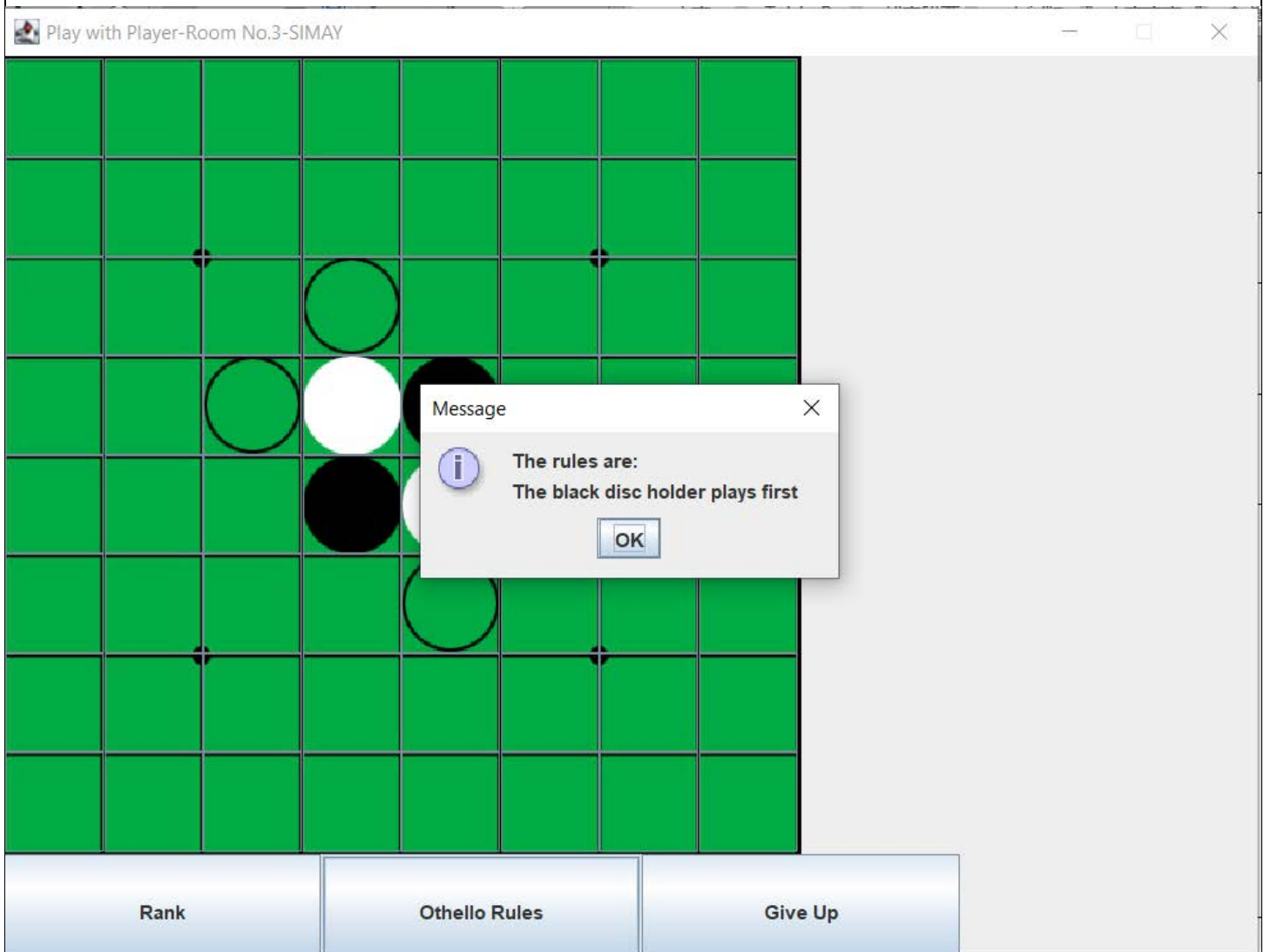
Testing Steps:

1. Clients play between each other
2. During one of the clients presses the Othello Rule button

Expected Behaviour:

1. A message will appear with the rules of the game

Result:



Meet the Expected Behaviour: True

## 6.8. Coverage Metrics

Element ▲	Class, %	Method, %	Line, %
✓ ss.othello.game	57% (4/7)	39% (25/63)	51% (269/519)
C Board	100% (1/1)	68% (20/29)	64% (258/397)
I Game	100% (0/0)	100% (0/0)	100% (0/0)
C HumanPlayer	100% (1/1)	20% (1/5)	33% (3/9)
C Main	0% (0/1)	0% (0/1)	0% (0/11)
E Mark	100% (1/1)	50% (1/2)	28% (2/7)
I Move	100% (0/0)	100% (0/0)	100% (0/0)
C OthelloGame	0% (0/1)	0% (0/17)	0% (0/65)
C OthelloMove	100% (1/1)	50% (3/6)	66% (6/9)
C OthelloTUI	0% (0/1)	0% (0/3)	0% (0/21)

Unit testing was utilized to test the game logic. As shown in the above figure of coverage Metrics of the game logic testing, the unit test covers 68% of the methods in the Board Class. This class is tested extensively because methods in this class are highly related to the game logic and the functionality of the game.

## 6.9. Complexity Metrics and the Riskiest Parts

The riskiest part of the application would be the `run()` method in the `ClientHandler` Class and the `flipDisc()` method in the `Board` class. The `ClientHandler` method includes multiple nested if-else statements. As for the `flipDisc()` method, it includes a nested loop and an if-else statement. For both the `ClientHandler` Class and the `flipDisc()` class the complexity metrics are 5. Thus, for the `flipDisc()` method, a unit test has been established to ensure this method functions perfectly. For the `run()` method in `ClientHandler` Class, extensive systematic tests have been conducted. Nevertheless, since this project is completed by myself and the client provided by the university does not send protocols such as LIST. Thus, there are some functions that cannot be checked by the systematic test.

## 6.10. System tests and Automated tests

Automated tests using Junit ensure that the basic functionality of the game logic works. I also used automatic testing to test the game logic by using a `OthelloTUI` to generate move from the AI players. System tests ensure that the functionality of the server work. System tests allow me to check the functionality of each class step by step. As for the automated tests, it allows me to cover most of the basic functionality of classes.

# 7. Reflection on Development Process

## 7.7. Reflection on the Planning and Possible Improvements for Future Projects

The project took me more time than I expected. I was planning to finish with implementing everything 2 days before the deadline but the bonus required much more work than I expected. This time, I decided to completely change my user interface from a TUI and GUI that helped in a lot of things while implementing the client. It made me to see the moves clearly but unfortunately since it was my first time creating a program with GUI there was a lot of things that did not know how to do, so a lot of days were spent creating the GUI. The mistake that I did early on in the development of my project was not designing the different components and how they are gonna work together. Doing that made me to waste a lot of time. I was improvising on the spot on which classes to create and which methods, where if I designed in the first place I would know what to build and I would know how to

connect the different componets. Unfortunately design was not the only that I did not consider. I finished implementing everything in my project but three days before the deadline I discovered that I did not do any multithreading mechanisms on the client part, which it caused a lot of problems while multiple games were playing at the same. This lead for me to having to change a lot of classes where multithreading needed to be done. In the future, when I decide to build another project, I would start by designing everything, which classes to build and what kinds of methods these classes will have and I would consider multithreading mechanisms if they will be required for my project. I learned from my mistakes and I wouldn't want myself to repeat this cycle again.