

PENTAGO PROGRAMMING PROJECT

Kevin Schurman s2518406

Arsalaan Khan s2558106

Table of Contents

Table of Contents	2
Introduction	3
Justification of Minimal Requirements	4
Explanation of Realized design	7
Class Diagram	7
Sequence Diagram:	8
List of responsibilities	10
List of packages and classes and explaining package structure	10
Concurrency Mechanism	13
List of various threads and where they are created (and their purpose)	13
All variables (objects) that could be accessed by multiple threads and why they are shared	14
How we ensured concurrency	20
Reflection on Design	21
The pros of the initial design	22
The cons of the initial design	23
Possible improvements in the initial design	23
Reflection on final design	23
System Testing	24
Overall Testing Strategy	42
Reflection on Process	44
Arsalaan Khan	44
Kevin Schurman	44

Introduction

For the programming project of module 2 of Technical Computer Science, we have developed a networked board game called *Pentago*. *Pentago* is an abstract strategy game for two players with four 3×3 grids arranged into a larger 6×6 grid. This game reimplements the well known *Connect 4* with a twist: After placing a marble, the player has to twist one of the grids by 90°, thus changing the board after every turn. The first player to get five marbles in a row wins.

The programming language used to develop this software was Java. This language was used because it is relatively easy to understand yet powerful enough to handle big computations. It also allowed for some programming concepts such as encapsulation, interfaces, references/pointers, and many others which we have utilized to allow for our programming project to take off. In this project, we have developed a server-side, client-side, and direct implementations of the game.

The server-side implementation works with anyone that has the correctly configured client-side protocols and application to play and communicate with. The client-side implementation works with any server that has the correctly configured server-side protocol and application to play and communicate with. The direct implementation works with anyone who wants to play by themselves, either with a computer or against another friend.

It is recommended that the environment for all these applications that are being used are used in the Linux environment rather than the Windows environment or any other environment we are not familiar or comfortable with.

In this document, we have described the various methods, decisions, testing, and many others that we have made and done to approach this project and make the best possible product for this assignment and application.

Justification of Minimal Requirements

When the assignment to implement the Pentago game was assigned to us by the university, we were expected to write the program in such a way that it meets some important requirements. We have adhered to this and ensured that our program meets all the significant requirements. Various system tests were also performed to confirm the correct implementation of all of these. **All these minimal requirements have also been tested in the system tests section, which should also be a good justification of these requirements.** The important requirements specified to us and how we have fulfilled them are as follows :

1. A standard game can be played on both client and server in conjunction with the reference server and client, respectively :

To play a full game as the reference client on our server:

- Start the server by running the class "ServerSideMethod.java" and let it run.
- Enter the desired port the server should listen to.
- Start the reference client
- Enter the same port as entered to start the server.
- Enter the username in the reference client
- Start another reference client and do the same.
- The game should now be started, and the server will send the game situation and result to the reference clients correctly.

To play a full game on the reference server as our client:

- Start the client by running the class ClientSideMethod.java
- Enter the IP address and the port of the reference server.
- Input a username
- If the login is successful, it will print "LOGIN SUCCESSFUL!"
- Type "QUEUE" in the console, which will put you in a Queue
- Once it has found another player in the queue, the game will start
- To enter moves on the reference client, you enter them in a particular format. To make it easy for the client, we ask the user to input the row, column and the rotation of their move. All of them must be separated by a space(" "). Example input: 1 0 1. This would place the marble on row 1, column 0 and the rotation number would be 7 which would rotate the top left sub-board anti-clockwise.
- The client will keep printing the game board with the moves made by the client. After the game is over, the client will see a game over message as well. To start the game again, type "QUEUE" again.

2. The client can play as a human player, controlled by the user: By default, the client plays as a human player unless specified otherwise. The client asks the user to input the row, column and rotation of their move whenever it is their turn and does that on the board. After entering your username and successfully logging in, type QUEUE to get in a queue as a human player!

3. The client can play as a computer player, controlled by AI: After successful login, the client can type "-N" to use the Naive strategy AI or "-S" to use the smart strategy bot. If a

client wants to get back to human input, he simply has to type “-H” and human input for future games will be taken again unless specified otherwise. After that has been specified, the client can then queue for the games and either smart, naive or human input will be chosen depending on what the client specified.

4. **When the server is started, it will ask the user to input a port number where it will listen to. If this number is already in use, the server will ask again :**

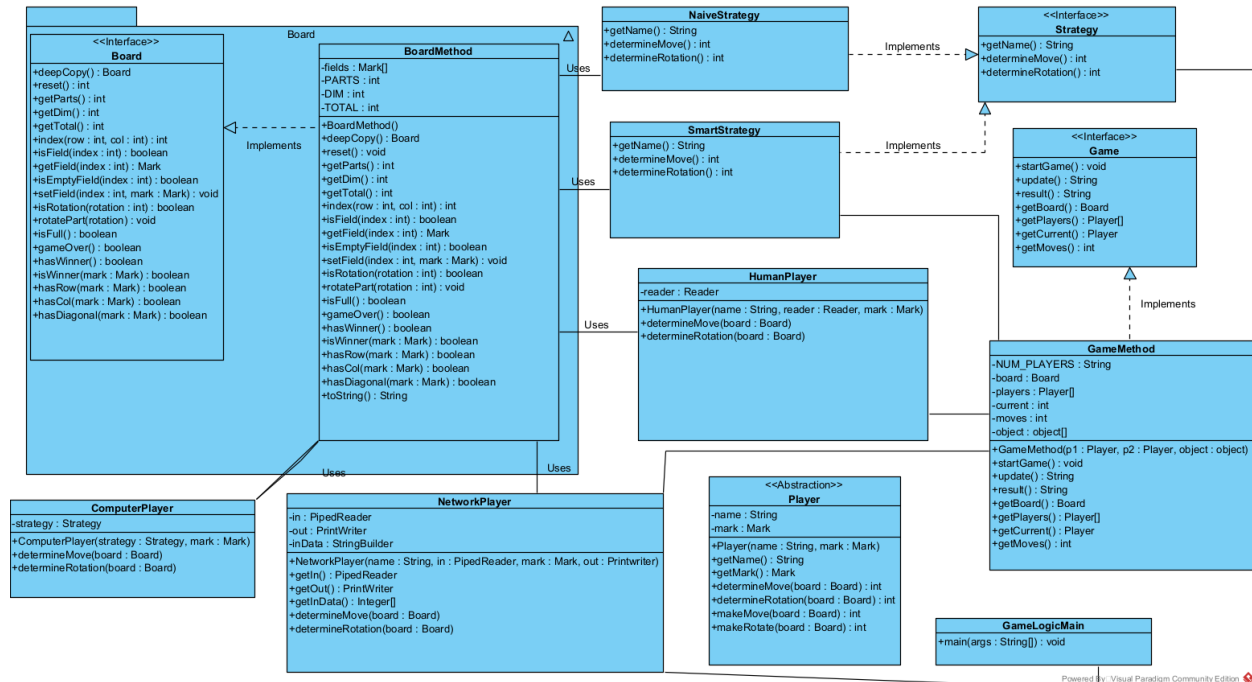
After running the ServerSideMethod.java, it will ask the user to input the port it should listen to, since we use input from System.in and parse it accordingly to start the serverSocket on the port specified. If it is taken, it will ask again, since we catch the IOException which is thrown when the connection fails and handle it accordingly and ask for the input again.

5. **When the client is started, it should ask the user for the IP-address and port number of the server to connect to:** After running the ClientSideMethod.java, it will ask the user for the IP-address and the port it should connect to because the main class starts a thread of the class “ClientMethod” and the start method of the class asks the user for the IP address and the port of the server it should connect to using System.in and then assigns the socket to the IP address and the port.
6. **When the client is controlled by a human player, the user can request a possible legal move as a hint via the TUI:** To request a hint as the client, the user can just enter “HINT” instead of their move and from the TUI, they can see a hint given to them, and then it will ask for input again.
7. **The client can play a full game automatically as the AI without intervention by the user:** After logging in and before queuing up for a game, “-N” or “-S” must be typed to make the player a bot player. After that, once someone queues by entering “QUEUE”, and the game starts(if 2 players are found), the bot makes moves on its own since user input is no longer read for every move. The move is determined by the strategy chosen and passed to the server automatically!
8. **The AI difficulty can be adjusted by the user via the TUI:** After logging in and before queuing up for a game, “-N” or “-S” must be typed to make the player a bot player. The former refers to making the current client a “Naive” bot player, which is not as good since it makes random valid moves on the board. The latter makes the client a “Smart” bot player, which is usually smarter since it checks for winning conditions before placing the move. After selecting one of them, the user can queue up and once the game is ready to start, the user will now play as either the smart bot or naive bot.
9. **Whenever a game has finished (except when the server is disconnected), a new game can be played without needing to establish a new connection in between:** After a game has finished, the client can just enter “QUEUE” again on the console and the client will be put back in the queue and the new game will be started once there are 2 players in the queue!
10. **All communication outside of playing a game, such as handshakes and feature negotiation, works on both client and server in conjunction with the reference server and client, respectively:** The user can just type “HELP” at any time to look at all the available commands to the user.

- 11. Whenever a client loses a connection to a server, the client should gracefully terminate:** Whenever the client disconnects to the server, a nice message is printed "Connection to the server was lost. Exiting..." No stack-trace is seen by the client because the IOException which is caused by the loss of connection is handled by printing this out to the user.
- 12. Whenever a client disconnects during a game, the server should inform the other clients and end the game, allowing the other player to start a new game.**

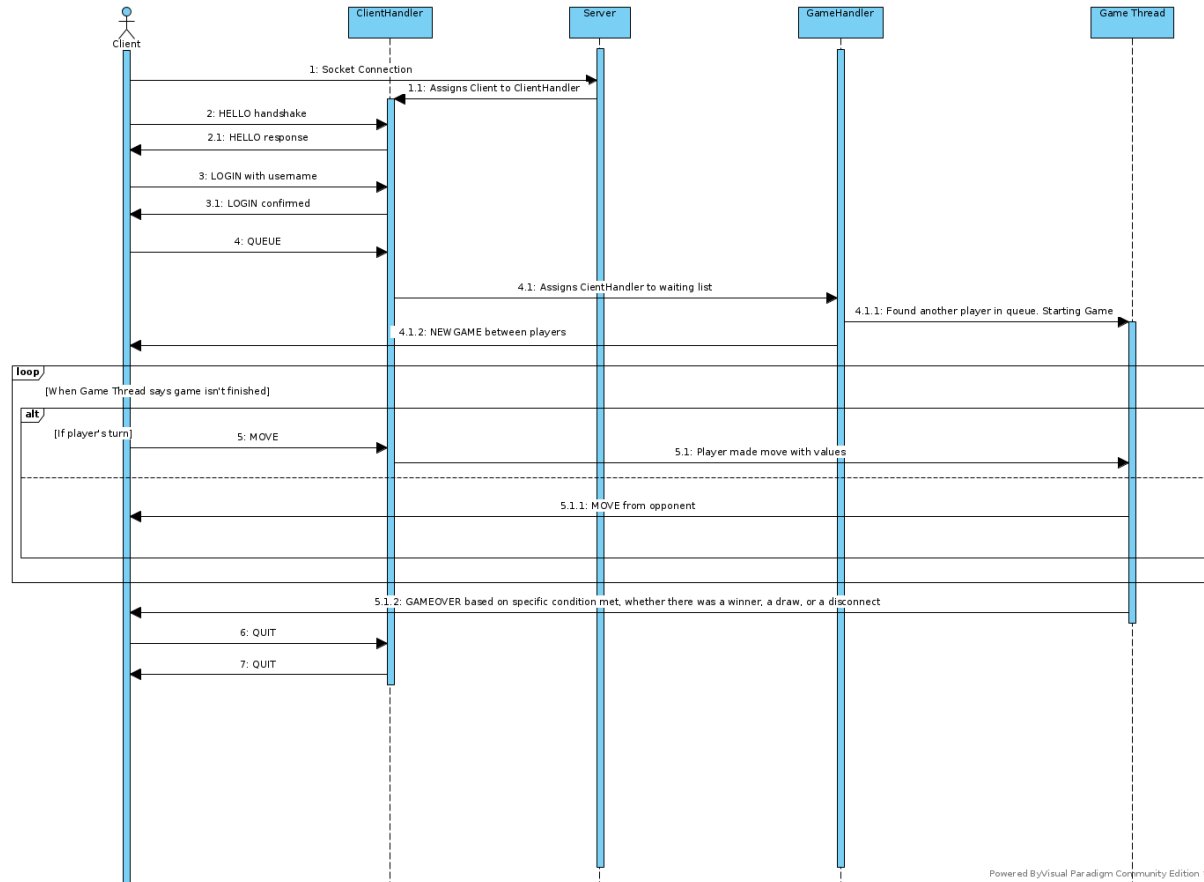
Explanation of Realized design

Class Diagram

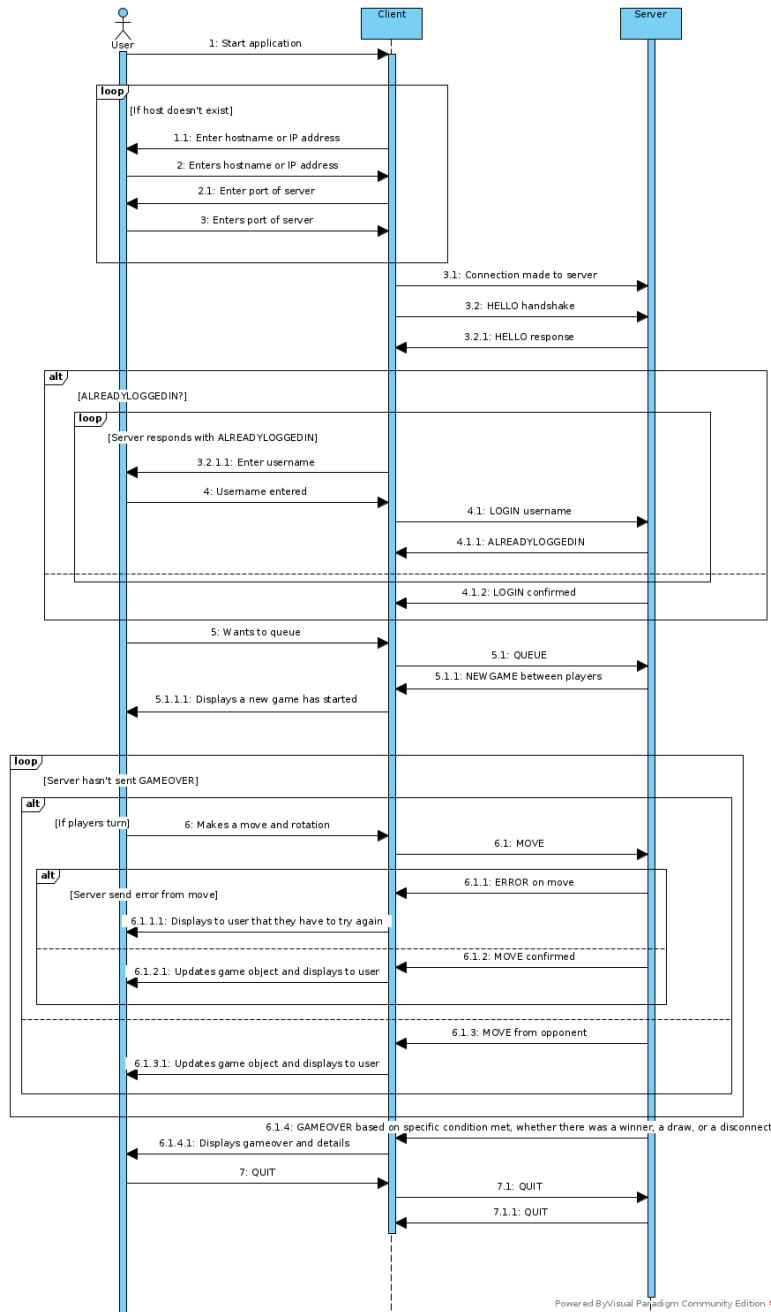


Due to the magnitude of this software, we couldn't include the whole system and hence we decided to go for the largest component of the system which is the gamelogic. We didn't go for the network system, as it would unnecessarily clutter the diagram and make it unreadable if it has too many classes. This class diagram's explanation for each class can be seen in the Javadoc exported in the zip file and also in the package's section below. From the class diagram we can see that there are various types of players, strategies, the board and the game. All of them use each other. We went for this kind of design because it is simple and easy to understand, as no classes have multiple responsibilities and each class has a very specific responsibility to itself. The biggest and most important class would be the BoardMethod class since it has various functions and is associated with every other class and the whole system would fail without it.

Sequence Diagram:



When the client makes a connection to the server, the server makes a dedicated thread for the client and uses a clientHandler. This is where all the communication between the client and the server (specifically clientHandler) happens. A HELLO handshake is required to happen before anything else can happen, as well as the login with the username. Once all said is done, the client will say that they want to queue, in which the clientHandler communicates with the gameHandler and tells it that a new player has been added to the queue. Once enough players are in the queue, the server will then make a new game object and refer the clientHandler socket output directly to the client. The gameHandler will then tell the client that a new game has happened, and now the client and the game object can communicate with each other with the other opponent. If it's the player's turn, the player will make a move and the clientHandler will forward that move to the game thread in which it will process the data. If it's the opponent's turn, they will tell the client the move that the opponent has made. Once the game thread has recognized a game over, it will tell the client that the game over has happened and ended. The client can then quit, in which case they send it to the clientHandler, the client handler sends back QUIT to the client, and the clientHandler ends, despite the server and gameHandler still running.



When the user starts the application, the client will ask for a valid host and port number to connect to the host. Once a connection has been established, the HELLO handshake will commence. Once done, the application will ask for the user to input their username. If the username inputted exists in the server, the client will ask the user to try again until the server gives the LOGIN protocol. When the user wants to queue, the client will send the QUEUE protocol to the server. Once a game has been found from the server, the NEWGAME protocol will be sent back to the client and the client will tell the user that a new game has been established. The game will keep on going until the server sends the GAMEOVER protocol. If the user has made a move, but it was incorrect, the server will send an ERROR protocol message in which the user will have to make a different input, and one that's valid. If the move is valid, the

server will send back the name input from the user confirming the move to be valid, in which the client will update the display to the user with the move. When the opponent makes a move, the server will send a MOVE protocol to the client and the client will update the display to reflect that move. Once a game over has been reached from the server, the client will display that a game over has happened and will display who has won, if it was a draw, or if there was a disconnection. The user now wants to quit, in which the client sends the QUIT protocol, and the server also does the same thing as well, thus ending the application despite the server still running and the user still living.

List of responsibilities

To design a software of this magnitude, we had to come up with a list of responsibilities to give each class a purpose in the whole system. The list of responsibilities would be:

- 1) **Implementation of the game logic:** This includes everything related to the Pentago game rules implementation. Creating the board, checking for win conditions, placing marbles on the board, checking for valid moves, and many other things.
- 2) **Manage the server side:** This includes a working server that would allow other clients to connect to it and play Pentago
- 3) **Manage the client side:** This includes the user interface for the client whenever he or she is playing the Pentago game.
- 4) **Testing:** This includes the testing of the software to ensure a stable product and effective performance in all circumstances.
- 5) **Exceptions:** This includes having custom exceptions to be thrown and handled that are specific to this software
- 6) **Handling the protocol:** This includes classes that will have the protocol to send messages so that the software doesn't have to change entirely if the protocol changes.
- 7) **Creating the game player:** This includes implementing the various types of players who would play the game.

List of packages and classes and explaining package structure

For a more detailed explanation of all these classes, please refer to our Javadoc included in the zip file. That also includes an explanation of all the methods and what they do.

To fulfill all these responsibilities we divided these responsibilities into several packages in the source code with each package and class mainly responsible for fulfilling one of the responsibilities to give our entire program some structure and each package and class a meaning. We have decided to opt for this class structure since each package would now be responsible for a specific part of the system and if a future developer wants to modify a specific part of the system, that could be done easier with this structure, and also it looks extremely organized.

These would be the main packages :

- 1) **clientside**: This package contains all the classes responsible for *managing the client side of the game*, ensuring that the server messages are handled appropriately, and the user has a user-friendly interface to play the game. The classes in this package include:
 - a) **Client**: It is an interface that represents the client of the Pentago game.
 - b) **ClientMethod**: This is a class that implements the Client interface. This class is responsible for handling the start-up of the client and letting it connect to the appropriate server
 - c) **InMethod**: This is the class responsible for handling the received messages from the server once a connection has been established with the server and displaying messages on the client TUI accordingly.
 - d) **OutMethod**: This is the class responsible for sending out messages to the client typed by the user in the client TUI. The messages are sent in a way that it doesn't violate the client server communication protocol.
 - e) **ClientSideMethod**: This is the class that has the main method which is going to run the client by starting a new thread of the class "ClientMethod.java"
- 2) **commands**: This package contains the classes responsible for *handling the client and server protocols*, which contains the correct format of all the messages being exchanged by the client and the server. This format must be followed for the client and server to correctly recognize each other's messages and handle them accordingly. The various classes in this package include :
 - A. ClientCommands.java : This class contains the client commands following the protocol specified to us in the project description. It has methods that return the various protocol messages.
 - B. ServerCommands.java : This class contains the server commands following the protocol specified to us in the project description. It has methods that return the various protocol messages.
- 3) **gamelogic**: This package contains the classes responsible for the *implementation of the game logic of Pentago*. The package contains several other packages inside to further divide the list of responsibilities, since game logic in itself has a lot of various sub-responsibilities. These packages include:
 1. **board** - Contains all the classes that implement the game board and the whole game logic. The classes in this package include:
 - a) Board.java - This is the board interface
 - b) BoardMethod.java - This class implements the gamelogic of the board and also prints out the board.
 2. **computer** - Contains all the classes to determine the strategies for various difficulties of AI. The classes in this package include:
 - a) NaiveStrategy - The dumb AI that makes random moves on the board.
 - b) SmartStrategy - A smarter AI that checks for winning conditions before making the move on the board.
 - c) Strategy - The interface for an AI strategy, can be implemented to have new strategies
 - d) StrategyFactory - Interface to make a new strategy

- e) StrategyFactoryMethod - Implements the strategy factory and is responsible for making new strategies.
- 3. **exception** - Contains all the classes to handle exceptions. All classes in the package are related to having the responsibility of managing exceptions. The classes in this package include:
 - a. IncorrectFormatException : Thrown when the user inputs a move in the incorrect format
 - b. InvalidInputException : Thrown when the user does not space the input accordingly. When prompted, the row, column and rotation should be separated by a space!
 - c. InvalidMoveException : Thrown when the user inputs an illegal move on the board
 - d. InvalidRotationException : Thrown when the user inputs an invalid rotation on the board.
- 4. **game** - Contains all the classes to start the game of Pentago and make it playable. The classes include :
 - a. Game - The game interface!
 - b. GameMethod - Implementation of the game interface, allows the client to play a game and run it. Uses the board method to implement the board logic.
- 5. **player** - Contains all the classes to manage the player playing Pentago. There are various types of players and each class represents a different type of player:
 - a. PlayerFactory : Interface Used to create a new player
 - b. PlayerFactoryMethod : Class used to create a new player
 - c. Mark : Represents the marble on the board
 - d. NetworkPlayer: Represents a player playing online
 - e. Player : The abstract player class, can be extended to create a new player
 - f. ComputerPlayer : The AI player
 - g. HumanPlayer : Represents the human player playing Pentago.
- 6. **test** - Contains all the classes responsible for testing the gamelogic. The classes include :
 - a. BoardMethodTest.java - Extensively tests the board logic
 - b. GameMethodTest.java - Extensively tests the game logic which refers to testing the inputs of the client and having correct output on the board.
- 4) **serverside**: This package contains the various classes responsible for *managing the server side of the game*. The packages include :
 - a) **clientHandler**: Includes the clienthandler interface and the clienthandler.
 - b) **gameHandler**: Includes the gamehandler interface and the gamehandler.
 - c) **server**: Includes the server interface and the server class.

The serverside package also includes a class called "ServerSideMethod.java" which has a main method and is used to start the server.

Concurrency Mechanism

Since multiple clients can connect to the server, there is a huge risk of concurrency issues happening. In multithreaded programs, concurrency is defined as the ability to run several programs in *parallel*. Due to multiple threads being created and running at the same time, there could be times when one thread finishes before the other one and if that thread then goes on doing other tasks, it could lead to problems. For example, while 2 client handler threads are in a game, the threads should be at the same stage of the game at all times, waiting for the slowest thread to complete its computations before proceeding to the next stage of the game. There are multiple ways we can achieve this in Java.

List of various threads and where they are created (and their purpose)

For an application to allow multiple concurrent functions to work, the threads created server the function for the usability of the application:

- Client-side
 - **Main client start method:** The main method is what starts all the threads and processes to allow for the communication of data, the connection with the player by displaying information, the input from the terminal to the server, and many others. This thread/method is vital to start up all the other threads.
 - **Data flow In method:** This method is used to get all flowing input from the server and be processed in the client, either to be displayed to the user, processed in the game, send back information to the server, or others. This thread/method relies on the **Main client start method**, as they will be the thread to tell this thread to start.
 - **Main thread:** The main thread gets all the input from the user and processes that input for the client whether it be for visual, processing, responding, etc.
 - **Side thread:** The side thread is only used when the client is currently in the game. The main thread of **Data flow In method**, indicated when the side thread should be started and should process/display to the client the visual information and such. The game method also occasionally turns the thread back on after doing some processing.
 - **Data flow Out method:** This method is used to get all input from the user or in flowing processed data out back to the server. This method is also used for processing other data, such as the client side information, and many others. This thread/method relies on the **Main client start method**, as they will be the threads to tell this thread to start.
 - **Main thread:** The main thread gets all input from the user or the **Data flow In method** and sends the data/information back to the server or back to the client depending on the command made.

- **Side thread:** The side thread is only used when the client is currently in game and if they told the client to start using the bot over their own input. This thread will only be activated by the **Data flow In method**, as well as the game processed data as well, for the best possible performance.
 - **Awaiting dead server:** This thread is used to wait for the client to tell this thread that the server that they are currently communicating with has either disconnected due to the server going down, or the server losing connection with the user. This thread will await, then exit the application once it has been signalled. This thread/method relies on the **Data flow In method** and the **Data flow Out method**, as they will be the threads to tell this thread to exit.
 - **Game Thread:** This thread is used to allow for the game processing and mainly used for the visuals to be displayed towards the client on the terminal. This thread/method relies on the **Data flow In method** to start, as it houses a condition to be met from the server to indicate if the game has started. This method/thread starts every time that condition is met. If the condition has been met, but the game hasn't ended, the thread will not be created/start again.
- **Server-side**
 - **Main server start method:** The main method is what starts all the threads and processes to allow for the communication of data between many connecting players, as well and processing data to be sent back to the client. It firsts creates a dedicated GameHandler thread and then runs a while loop for all the connecting players in which it creates a ClientHandler specifically dedicated for that user. This thread/method is vital to start up all the other threads.
 - **ClientHandler method:** This thread is used to specifically be dedicated to allow for the client to do tasks, request games, communicate with the game, server, or other related things that the client handler allows for the user to do so. This thread/method relies on the **Main server start method**, as they will be the thread to tell this thread when to start and get working. There can be multiple instances of this thread working simultaneously.
 - **GameHandler method:** This thread is used to manager and create more threads for the game processing when the user wants to queue and play a game against other players. This thread/method relies on the **Main server start method**, as they will be the thread to tell this thread when to start and get working.
 - **Game Thread:** This thread is used to allow for the game processing between the two players that are currently playing against each other to win. This thread/method relies on the **GameHandler method** to start and allow for games to be played.

All variables (objects) that could be accessed by multiple threads and why they are shared

To allow for the communication of data between each thread, it is required that the threads that want to access and communicate with each other must have that instance to connect to, whether it be the class or method. All variables that are shared with each other are private and

are only able to access those methods through the use of encapsulation (using getters and setters). This is used so in order to communicate with multiple threads at any instance:

- Client-side
 - **Main Client method**
 - **getHelloed()** - This function is used to access the condition to check if the client and the server has exchanged the “HELLO” handshake. This is mainly used to communicate with the **Data flow In method**.
 - **isDebug()** - This function is used to get access to more details when it comes to how data is sent, received, processed, and handled with the client. To activate this, the arguments when starting up the application requires the “--debug” flag. This is mainly used to communicate with all the other threads and methods that either send, receive, or handle an exception. This is mainly used to communicate with the **Data flow In method** and **Data flow Out method**.
 - **getLogged()** - This function is used to access the condition to check if the client and server has exchanged login information and if the server has successfully received and accepted the condition. This is mainly used to communicate with the **Data flow In method**.
 - **getGaming()** - This function is used to access the condition to check if client and server are currently in game, so the client can make some inputs. This is mainly used to communicate with the **Main Client method**, **Data flow In method**, and the **Data flow Out method** for checking and communicating both with the client and server.
 - **setLoggedIn()** - This function is used to set the boolean loggedIn depending on if the client has logged in to the server or not. This is mainly used to communicate with the **Data flow In method** to check if the login has occurred.
 - **isInGaming()** - This function is used to access the boolean isInGaming which indicates if a client is in a game of pentago. This is mainly used to communicate with the **Data flow In method** and the **Data flow Out method** to check if the client is in the game and thus should specific game related information to the client.
 - **setInGaming()** - This function is used to set the boolean inGaming if a client is in a game or not. This is mainly used to communicate with the **Data flow In method** to set it depending on the data received by the server.
 - **getDeadServer()** - This function is used to check if the server connected is dead. It communicates with the client that it has died and thus should terminate. This is mainly used to communicate with the **Main Client method**, **Data flow In method**, and **Data flow Out method**.
 - **getOurTurn()** - This function is used to access the condition ourTurn which indicates if it is the client's turn or not. This is mainly used to communicate with the **Data flow In method**.

- **getInCondition()** - This function is used to access the condition inCondition which indicates if it is the client that should read the data from the game object now. This is mainly used to communicate with **Data flow In method** and **Data flow Out method** to allow for to communicate to the user with what they should see.
- **getUsrOut()** - This function is used to access the PrintWriter usrOut which is the print writer used by the client to send data and information to the terminal that the client is currently using. This is mainly used to communicate with the **Data flow In method** and **Data flow Out method**.
- **getOurPlayer()** - This function is used to access the int ourPlayer which is used to represent the current client playing the game. This is mainly used to communicate with the **Data flow In method** and **Data flow Out method**.
- **setMyOwnPlayer()** - This function is used to set the player, whether it be a bot or their own input, that they want to use during a game. They can either use their own input, the naive/random strategy, or a smarter strategy. This is mainly used to communicate with the **Data flow Out method**.
- **getMyOwnPlayer()** - This function is used to access the char myOwnPlayer which is a char representing the current client playing the game. It is used to differentiate between a human player, the Naiver AI player, or the Smarter AI player. This is mainly used to communicate with the **Data flow In method** and **Data flow Out method**.
- **getLock()** - This function is used to access the reentrant lock, used to use different conditions with awaits and/or signals. Locks are used to allow only one thread to access a critical section at one time to avoid concurrency issues. This is mainly used to communicate with the **Data flow In method** and **Data flow Out method**.
- **getOwnStrategy()** - This function is used to return the strategy that the player has inputted, whether they choose to play for themselves or let a bot play for them. This is mainly used to communicate with the **Data flow Out method**.
- **getMOutMethod()** - This function is used to return the out server data flow method in order to use specific functions by others to communicate with the server. This is mainly used to communicate with the **Data flow In method**.
- **setQueuing()** - This function is used to set the queuing boolean to indicate whether the player is currently attempting to queue up for a game to play. This is mainly used to communicate with the **Data flow In method** and **Data flow Out method**.
- **isQueuing()** - This function is used to get the queuing boolean to see if the player is indicating that they want to be able to play a game. This is mainly used to communicate with the **Data flow Out method**.

- **setCurrentPlayer()** - This function is used to set the current player such that the user can be in track of the current player that is currently playing. This is mainly used to communicate with the **Data flow In method**.
- **getCurrentPlayer()** - This function is used to get the current player such that the user can see who is currently playing the game right now. This is mainly used to communicate with the **Data flow In method** and **Data flow Out method**.
- **getShowBoard()** - This function is used to get the condition of showBoard is used to communicate with the game to see whether the client should display the board. This is mainly used to communicate with the **Data flow In method**.
- **newGame()** - This function is used to give a boolean value depending on whether the 2 players passed as the parameters can get in a game or not. This is mainly used to communicate with the **Data flow In method**.
- **getPwGame()** - This function is used to access the PrintWriter pwGame that is used in the game to communicate with the game object. This is mainly used to communicate with the **Data flow In method**.
- **getPwGameNet()** - This function is used to access another PrintWriter pwGameNet that is used from the server/connected player from the server to play the game with the client. This is mainly used to communicate with the **Data flow In method**.
- **getGame()** - This function is used to access the game object that has been assigned, where the client uses it to play the game with the server as well as display the visual information from the communication of the server. This is mainly used to communicate with the **Data flow In method** and **Data flow Out method**.
- **getClientName()** - This function is used to access the client application's name that has been inbuilt. This is mainly used to communicate with the **Data flow Out method**.
- **getUsername()** - This function is used to access the client's inputted username. This is mainly used to communicate with the **Data flow Out method**.
- **getSupportedExtensions()** - This function is used to access the supported extensions that the client can be able to currently handle from the server when asked. This is mainly used to communicate with the **Data flow Out method**.
- **start()** - This function is used to start the client method and get all the functions and data communication working. This is a vital function that is being used from the main function. This is mainly used to communicate with the **Main Client method**.
- **run()** - This is mainly used to indicate that the class will be threaded and run independently than of the **Main Client method**, which serves a different purpose than the current main thread since it's mainly used to check if the server has died and if the client should terminate.

- **Data flow In method**
 - **run()** - This is mainly used to indicate that the class will be threaded and run independently than of the **Main Client method**.
- **Data flow Out method**
 - **sendMessage()** - This is used in order to communicate data from the client to the server. While the function is mainly used by the **Data flow Out method**, if specific conditions are met in order areas of the client-side, they will need access to this function to communicate with the server about those conditions.
 - **run()** - This is mainly used to indicate that the class will be threaded and run independently than of the **Main Client method**.
- **Server-side**
 - **Main Server method**
 - **getSupportedExtensions()** - This function is used to return a String array with all the supported extensions that the server is able to do when communicating with the client. This is mainly used to communicate with the **ClientHandler method**.
 - **start()** - This function is used to start the server on the port it has been instructed to use. This is a vital function as it starts all the other threads and allows functions for the server. This is mainly used to communicate with the **Main Server method**.
 - **stop()** - This function is used to stop all the server threads and functions and close it down. It also removes all the clients as well beforehand. This is mainly used to communicate with the **ClientHandler method**.
 - **run()** - This function is used to create a new thread, where it would constantly make a new dedicated client handler for each new connection made to the server. This is mainly used to communicate with the **Main Server method**.
 - **addClient()** - This function is used to add a clientHandler object to the server and allocate for a new thread to be used. This is mainly used to communicate with the **Main Server method**.
 - **removeClient()** - This function is used to remove a clientHandler server and deallocate for a new thread to be used. This is mainly used to communicate with the **Main Server method** and **ClientHandler method**.
 - **getClientHandlers()** - This function is used to access the list of currently connected clients stored in the server. This is mainly used to communicate with the **ClientHandler method**.
 - **getGameHandler()** - This function is used to return the gameHandler object which handles a game of pentago or allows for other resources to be used. This is mainly used to communicate with the **ClientHandler method**.
 - **GameHandler method**

- **addQueue()** - This function is used to add a clientHandler to the waiting queue of a game. This is mainly used to communicate with the **ClientHandler method**.
- **getLock()** - This function is used to access the ReentrantLock used by this class to ensure thread-safety. This is mainly used to communicate with the **Game method** and **ClientHandler method**.
- **getToQueue()** - This function is used to access the toQueue condition used by this class, which indicates if a client is in a queue. This is mainly used to communicate with the **ClientHandler method**.
- **getToWaitAgain()** - This function is used to access the toWaitAgain condition used by this class, which indicates to the client when they should wait until making a queuing request again. This is mainly used to communicate with the **ClientHandler method**.
- **getToPlay()** - This function is used to access the getToPlay condition used by this class, which indicates if 2 clients are in a queue and are ready to play. This is mainly used to communicate with the **ClientHandler method**.
- **getBoardSignal()** - This function is used to access the boardSignal condition used by different classes, which indicates if the game has made a signal or done a specific task. This is mainly used to communicate with the **ClientHandler method**.
- **getToDone()** - This function is used to access the condition toDone which indicates if the client has just finished a game. This is mainly used to communicate with the **Game method**.
- **removeInGame()** - This function is used to remove the client from a game. This is mainly used to communicate with the **Game method**.
- **getScore()** - This function is used to return the scores of all players who have played on the server. This is mainly used to communicate with the **Game method** and **ClientHandler**.
- **run()** - This function is used to execute every time a thread, specifically a client handler, tells the game method that a new client wants to be able to join and play a game to which the game will be played, and the clients will be sorted if there has enough clients queuing. This is mainly used to communicate with the **Main Server method**.
- **ClientHandler method**
 - **getGameIn()** - This function is used to access the PipedReader used by the clientHandler. This is mainly used to communicate with the **GameHandler method**.
 - **getOut()** - This function is used to access the PrintWriter used by the clientHandler. This is mainly used to communicate with the **GameHandler method**.
 - **getClientName()** - This function is used to access the client's client name. This is mainly used to communicate with the **Main Server method**.

- **getUsername()** - This function is used to access the client's inputted username. This is mainly used to communicate with the **Main Server method**, **GameHandler method**, and **ClientHandler method**.
- **sendToClient()** - This function is used to send a message to the client using the PrintWriter. This is mainly used to communicate with the **GameHandler** and **ClientHandler**.
- **getInGame()** - This function is used to access the boolean inGame which indicates if a clientHandler is in a game or not. This is mainly used to communicate with the **GameHandler method** and **ClientHandler method**.
- **setInGame()** - This function is used to modify the inGame boolean, which indicates if a client is in the game or not. This is mainly used to communicate with the **Game method** and **GameHandler method**.
- **close()** - This function is used to exit and stop the client handler as it closes all the readers and writers as well as the socket used for the communication with the client. This is mainly used to communicate with the **Main Server method**.
- **run()** - This function is used to create a thread and execute information, mainly incoming from the user. This is mainly used to communicate with the **Main Server method**.

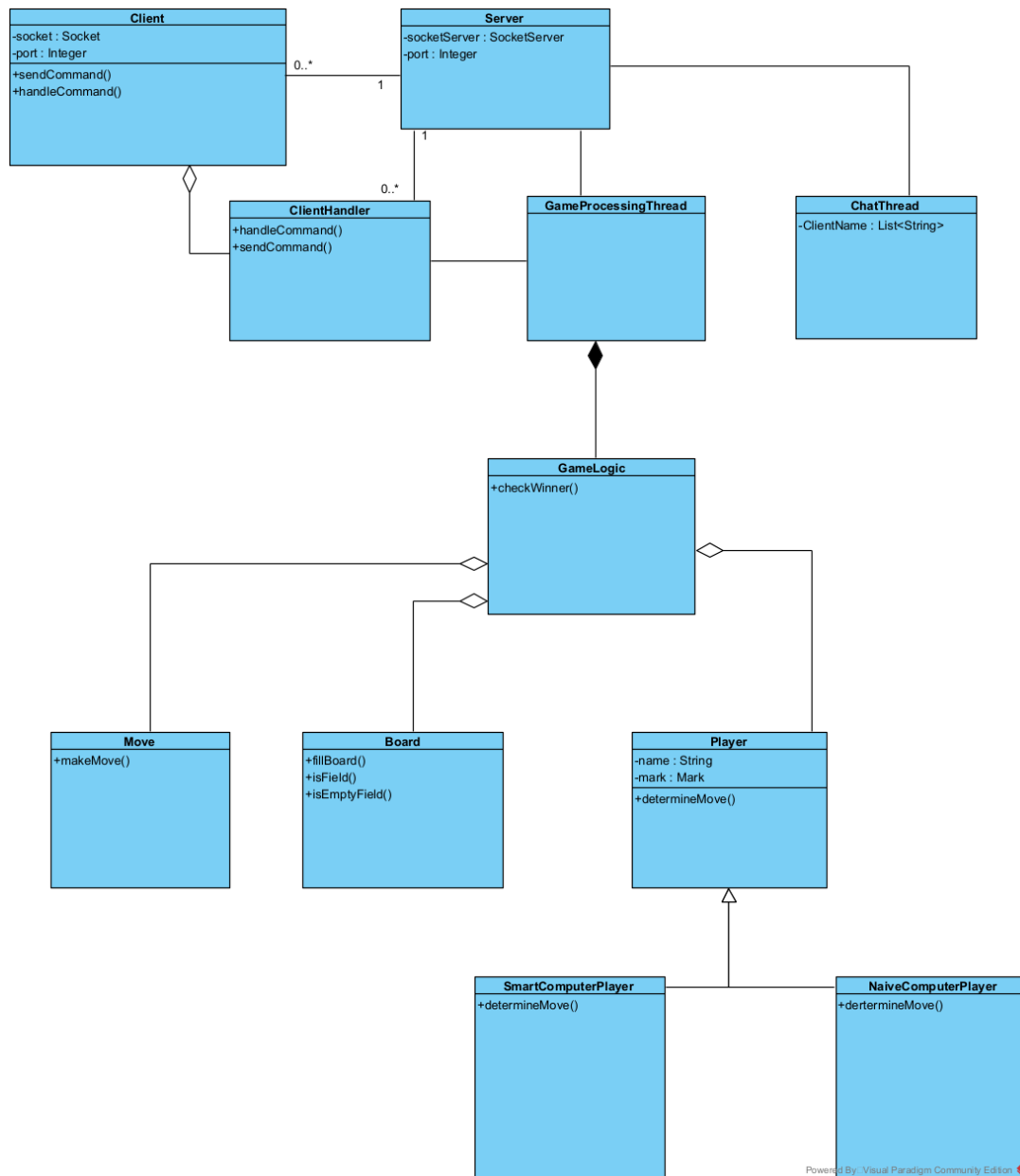
How we ensured concurrency

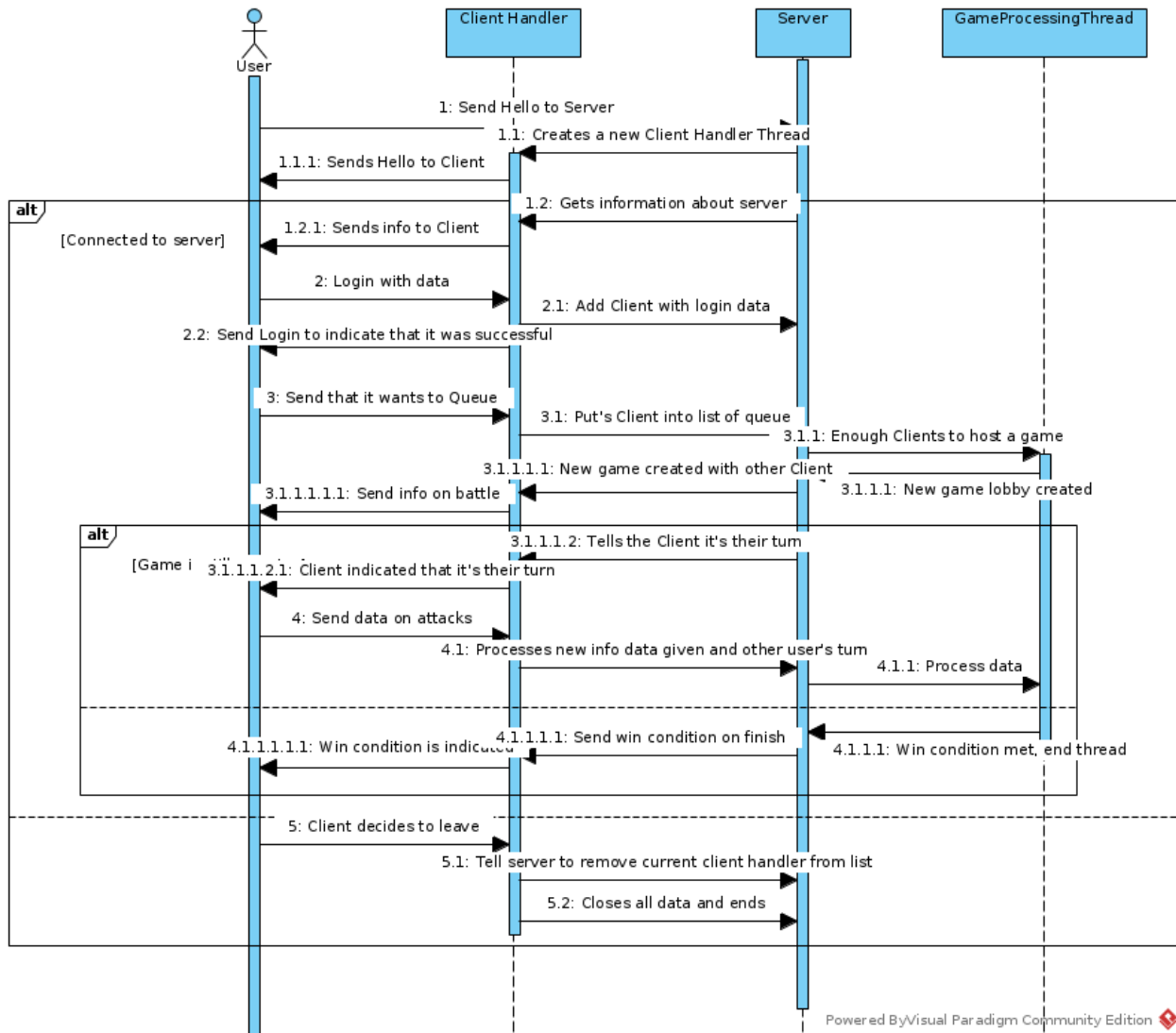
To be able to ensure concurrency between threads and how they interact with each class or function, we have made a couple of decisions when approaching this project:

- **Awaits and Signals:** When we want to be able for two or more threads to communicate with one another, we decided to force ourselves to use as much awaits and signals as possible to make sure that threads communicating with one another are doing it in the order we want them to do. We want data to flow correctly and consistently, thus the approach.
- **Threads Sleeping:** At first, we used threads to sleep when doing processing, either when it's client side or when it's server side. However, when making threads sleep for a specific amount of time, we came to the conclusion that threads sleeping makes the processing of the game or communication with other servers/clients makes things unstable, unpredictable, inefficient, and unprofessional. This is because we can assume that the client or server with their own threads will be communicating data for any specific amount of time, and thus we cannot make an assumption for how long the thread should sleep for when the task is done. The only times when the thread needs to sleep is if we cannot control the data flow on who should send or receive, so we make sure that we wait a little before carrying on with work. This is only used a handful of times.

Reflection on Design

Before we started working on and implementing the project, we came up with a class diagram to give ourselves a direction and have an idea of what we are going to do. However, there were a few things that we realized during the implementation of the game which forced us to stray away from our initial design. In this section, we will talk about the differences between the initial design and the final design of the project. Here is our initial class diagram and sequence diagram design of the game:





Reflection on Initial design: As it can be clearly seen, there are quite a lot of differences between the initial class diagram and what we ended up having after we implemented the project. There are, however, also a lot of similarities between them. The main difference would be that we underestimated the complexity of the software and hence have a lot fewer classes in the initial design. We hadn't really planned that well, and we also didn't really know a lot of things about how to even make a software of this magnitude. While we were implementing our project, we realized we would need a lot more classes since there are a lot of responsibilities that we missed out on when doing the initial design.

The pros of the initial design

- 1) The class diagram is really simple to understand. It is not cluttered at all and has perfect readability.

- 2) The class diagram covers all the basic requirements of the game which is having a client, a server and the game logic that implements the game.
- 3) Class diagram explains the relation between each class.
- 4) The class diagram lists the important methods and functions each class might have.
- 5) Class diagram clearly mentions that 1 server would have multiple clients connected to it.
- 6) The initial design is a lot more detailed than the new one and goes into specifics about the clientHandler and game processing thread, which we would also later use and take inspiration.

The cons of the initial design

- 1) The class diagram missed out on a lot of non-trivial details that the class diagram should have had.
- 2) Doesn't include interfaces.
- 3) There is no class to represent a marble on the board, which should have been an enum.
- 4) No separate classes to handle the protocol.
- 5) Can be a bit messy and sometimes all over the place.

Possible improvements in the initial design

While it's essential to keep a class diagram simple, it doesn't mean that the thinking behind the project should also be simple. We didn't really think that complexly about the game, which resulted in a lot of details being missed out. We should have thought out much further ahead, but then express our complex thoughts in an abstract way to keep things simple.

Reflection on final design

The pros of the final design:

- 1) It is a lot more detailed and covers most of the significant components.
- 2) Explains the relation between each classes in the diagram in a much better way
- 3) Contains all the methods and the fields of the classes in the diagram

The cons of the final design:

- 1) While it is more detailed, it misses out on a lot of important and non-trivial information,
- 2) The diagram has no multiplicities for a lot of relations which could have been useful

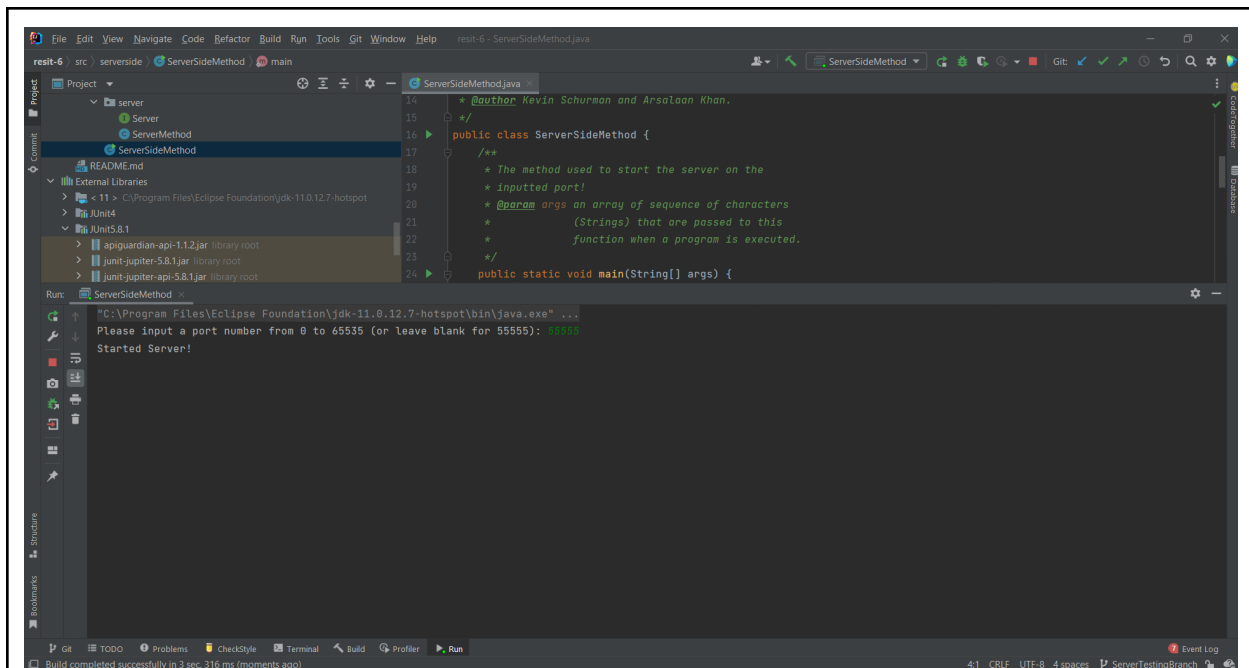
Possible improvements in the final design: Could make it more detailed by adding more classes because a lot of essential classes that are used in the system are not represented in the class diagram. Add some multiplicity in the diagram as well because they could be useful to the reader in understanding the system.

System Testing

Please NOTE : A few of these system tests were done in “debug mode” which is a feature we implemented for future developers interested in managing our software. Which prints out a few extra messages, like what exactly the server sent and what exactly the client received. We thought it might be helpful to see the exact communication happening between the client and the server in some of these tests as well. However, when debug mode is turned off, these exact messages are not printed but printed in a fancy way only that the client can see. By default, debug mode is turned off. To turn on debug mode for the client, specify the program arguments as “--debug” in the run configuration of “ClientSideMethod.java”. To turn on debug mode for the server, do the same in the run configuration of “ServerSideMethod.java”. It would then print our error messages and also what the client and server send and receive, which could be useful for future extension of the software

These are system tests for all the “Important requirements”. To see a list of all the important requirements, please refer to the section “Justification of Minimal Requirements” of this report. The critical requirements in these test reports are referred to as “IR” to keep things simple.

Testing report for IR 1
IR 1 : <i>A standard game can be played on both client and server in conjunction with the reference server and client, respectively</i>
Expected outcome: The user is able to successfully play a full game of Pentago and also receive the result of the game on the reference client when connected to our server. The client we have developed is also able to play a full game when connected to the reference server.
Testing result: Step 1: Starting the server by running the Java class “ServerSideMethod.java” and entering 55555 as the port on the console. Received the started server message indicating the server was started successfully:



Step 2: Starting the reference client and connecting it to the “localhost” address, since the server is running locally on the machine and port as 55555. Entering the username on the reference client when prompted to. Having a successful login and getting in queue for a game:

```

C:\Users\Arsalaan Khan>cd desktop
C:\Users\Arsalaan Khan\Desktop>java -jar client.jar
Please provide a port number (leave blank for 55555):
Please provide an IP address or hostname (leave blank for localhost):
[OUTGOING] 'HELLO~Pentago reference client v1.0'
[INCOMING] 'HELLO~Arsalaan and Kevin's Server'
Please give a username: Arsalaan
[OUTGOING] 'LOGIN~REFERENCE_Arsalaan'
[INCOMING] 'LOGIN'
[OUTGOING] 'QUEUE'

```

As we can see, we receive a hello from the server indicating our names as well, proving that it is indeed our server that the client is connected to. We are also receiving and sending messages correct with the protocol.

Step 3: Repeat step 2 but with a different username and the new game is started with clients exchanging their moves:

```

C:\Users\Arsalaan Khan\Desktop>java -jar client.jar
Please provide a port number (leave blank for 55555):
Please provide an IP address or hostname (leave blank for localhost):
[OUTGOING] 'HELLO-Pentago reference client v1.0'
[INCOMING] 'HELLO-Arsalaan and Kevin's Server'
Please give a username: Kevin
[OUTGOING] 'LOGIN-REFERENCE_Kevin'
[INCOMING] 'LOGIN'
[OUTGOING] 'QUEUE'
[INCOMING] 'NEWGAME-REFERENCE_Arsalaan-REFERENCE_Kevin'
[OUTGOING] 'MOVE-14-7'
[INCOMING] 'MOVE-31-2'
[INCOMING] 'MOVE-31-2'
[INCOMING] 'MOVE-9-0'
[OUTGOING] 'MOVE-21-6'
[INCOMING] 'MOVE-21-6'
[INCOMING] 'MOVE-12-2'
[OUTGOING] 'MOVE-27-5'
[INCOMING] 'MOVE-27-5'
[INCOMING] 'MOVE-5-5'
[OUTGOING] 'MOVE-11-5'
[INCOMING] 'MOVE-11-5'
[INCOMING] 'MOVE-23-4'
[OUTGOING] 'MOVE-22-7'
[INCOMING] 'MOVE-22-7'
[INCOMING] 'MOVE-15-5'
[OUTGOING] 'MOVE-27-1'
[INCOMING] 'MOVE-27-1'
[INCOMING] 'MOVE-18-2'
[OUTGOING] 'MOVE-10-6'
[INCOMING] 'MOVE-10-6'
[INCOMING] 'MOVE-32-2'
[OUTGOING] 'MOVE-13-0'
[INCOMING] 'MOVE-13-0'
[INCOMING] 'MOVE-16-7'
[OUTGOING] 'MOVE-1-1'
[INCOMING] 'MOVE-1-1'
[INCOMING] 'MOVE-11-1'
[OUTGOING] 'MOVE-20-1'
[INCOMING] 'MOVE-20-1'
[INCOMING] 'MOVE-17-2'
[OUTGOING] 'MOVE-2-4'
[INCOMING] 'MOVE-2-4'
[INCOMING] 'MOVE-28-4'
[OUTGOING] 'MOVE-23-3'
[INCOMING] 'MOVE-23-3'
[INCOMING] 'MOVE-34-7'
[OUTGOING] 'MOVE-3-3'

C:\Users\Arsalaan Khan\Desktop>java -jar client.jar
Please provide a port number (leave blank for 55555):
Please provide an IP address or hostname (leave blank for localhost):
[OUTGOING] 'HELLO-Pentago reference client v1.0'
[INCOMING] 'HELLO-Arsalaan and Kevin's Server'
Please give a username: Arsalaan
[OUTGOING] 'LOGIN-REFERENCE_Arsalaan'
[INCOMING] 'LOGIN'
[OUTGOING] 'QUEUE'
[INCOMING] 'NEWGAME-REFERENCE_Arsalaan-REFERENCE_Kevin'
[OUTGOING] 'MOVE-14-7'
[INCOMING] 'MOVE-31-2'
[INCOMING] 'MOVE-31-2'
[OUTGOING] 'MOVE-9-0'
[INCOMING] 'MOVE-9-0'
[INCOMING] 'MOVE-21-6'
[OUTGOING] 'MOVE-12-2'
[INCOMING] 'MOVE-12-2'
[INCOMING] 'MOVE-27-5'
[OUTGOING] 'MOVE-5-5'
[INCOMING] 'MOVE-5-5'
[INCOMING] 'MOVE-23-4'
[OUTGOING] 'MOVE-22-7'
[INCOMING] 'MOVE-22-7'
[INCOMING] 'MOVE-15-5'
[OUTGOING] 'MOVE-27-1'
[INCOMING] 'MOVE-27-1'
[INCOMING] 'MOVE-18-2'
[OUTGOING] 'MOVE-10-6'
[INCOMING] 'MOVE-10-6'
[OUTGOING] 'MOVE-32-2'
[INCOMING] 'MOVE-32-2'
[INCOMING] 'MOVE-13-0'
[OUTGOING] 'MOVE-16-7'
[INCOMING] 'MOVE-16-7'
[INCOMING] 'MOVE-1-1'
[OUTGOING] 'MOVE-11-1'
[INCOMING] 'MOVE-11-1'
[INCOMING] 'MOVE-20-1'
[OUTGOING] 'MOVE-17-2'
[INCOMING] 'MOVE-17-2'
[INCOMING] 'MOVE-2-4'
[OUTGOING] 'MOVE-28-4'
[INCOMING] 'MOVE-28-4'
[INCOMING] 'MOVE-23-3'
[OUTGOING] 'MOVE-34-7'
[INCOMING] 'MOVE-34-7'

C:\Users\Arsalaan Khan\Desktop>java -jar client.jar
[INCOMING] 'MOVE-12-2'
[OUTGOING] 'MOVE-27-5'
[INCOMING] 'MOVE-27-5'
[INCOMING] 'MOVE-5-5'
[OUTGOING] 'MOVE-11-5'
[INCOMING] 'MOVE-11-5'
[INCOMING] 'MOVE-23-4'
[OUTGOING] 'MOVE-22-7'
[INCOMING] 'MOVE-22-7'
[INCOMING] 'MOVE-15-5'
[OUTGOING] 'MOVE-27-1'
[INCOMING] 'MOVE-27-1'
[INCOMING] 'MOVE-18-2'
[OUTGOING] 'MOVE-10-6'
[INCOMING] 'MOVE-10-6'
[INCOMING] 'MOVE-32-2'
[OUTGOING] 'MOVE-13-0'
[INCOMING] 'MOVE-13-0'
[INCOMING] 'MOVE-16-7'
[OUTGOING] 'MOVE-1-1'
[INCOMING] 'MOVE-1-1'
[INCOMING] 'MOVE-11-1'
[OUTGOING] 'MOVE-20-1'
[INCOMING] 'MOVE-20-1'
[INCOMING] 'MOVE-17-2'
[OUTGOING] 'MOVE-2-4'
[INCOMING] 'MOVE-2-4'
[INCOMING] 'MOVE-28-4'
[OUTGOING] 'MOVE-23-3'
[INCOMING] 'MOVE-23-3'
[INCOMING] 'MOVE-34-7'
[OUTGOING] 'MOVE-3-3'
[INCOMING] 'MOVE-8-6'
[OUTGOING] 'MOVE-7-6'
[INCOMING] 'MOVE-7-6'
[INCOMING] 'MOVE-26-7'
[OUTGOING] 'MOVE-13-5'
[INCOMING] 'MOVE-13-5'
[INCOMING] 'MOVE-26-2'
[OUTGOING] 'MOVE-25-6'
[INCOMING] 'GAMEOVER-VICTORY-REFERENCE_Kevin'
[OUTGOING] 'QUEUE'
[INCOMING] 'NEWGAME-REFERENCE_Kevin-REFERENCE_Arsalaan'
[OUTGOING] 'MOVE-20-1'
[INCOMING] 'MOVE-20-1'
[INCOMING] 'MOVE-30-3'
[OUTGOING] 'MOVE-3-0'
[INCOMING] 'MOVE-3-0'

C:\Users\Arsalaan Khan\Desktop>java -jar client.jar
[INCOMING] 'MOVE-21-6'
[OUTGOING] 'MOVE-32-2'
[INCOMING] 'MOVE-32-2'
[INCOMING] 'MOVE-12-2'
[OUTGOING] 'MOVE-27-5'
[INCOMING] 'MOVE-27-5'
[INCOMING] 'MOVE-5-5'
[OUTGOING] 'MOVE-11-5'
[INCOMING] 'MOVE-11-5'
[INCOMING] 'MOVE-23-4'
[OUTGOING] 'MOVE-22-7'
[INCOMING] 'MOVE-22-7'
[INCOMING] 'MOVE-15-5'
[OUTGOING] 'MOVE-27-1'
[INCOMING] 'MOVE-27-1'
[INCOMING] 'MOVE-18-2'
[OUTGOING] 'MOVE-10-6'
[INCOMING] 'MOVE-10-6'
[INCOMING] 'MOVE-32-2'
[OUTGOING] 'MOVE-13-0'
[INCOMING] 'MOVE-13-0'
[INCOMING] 'MOVE-16-7'
[OUTGOING] 'MOVE-1-1'
[INCOMING] 'MOVE-1-1'
[INCOMING] 'MOVE-11-1'
[OUTGOING] 'MOVE-20-1'
[INCOMING] 'MOVE-20-1'
[INCOMING] 'MOVE-17-2'
[OUTGOING] 'MOVE-2-4'
[INCOMING] 'MOVE-2-4'
[INCOMING] 'MOVE-28-4'
[OUTGOING] 'MOVE-23-3'
[INCOMING] 'MOVE-23-3'
[INCOMING] 'MOVE-34-7'
[OUTGOING] 'MOVE-3-3'
[INCOMING] 'MOVE-8-6'
[OUTGOING] 'MOVE-7-6'
[INCOMING] 'MOVE-7-6'
[INCOMING] 'MOVE-26-7'
[OUTGOING] 'MOVE-13-5'
[INCOMING] 'MOVE-13-5'
[INCOMING] 'MOVE-26-2'
[OUTGOING] 'MOVE-25-6'
[INCOMING] 'GAMEOVER-VICTORY-REFERENCE_Kevin'
[OUTGOING] 'QUEUE'
[INCOMING] 'NEWGAME-REFERENCE_Kevin-REFERENCE_Arsalaan'
[OUTGOING] 'MOVE-20-1'
[INCOMING] 'MOVE-20-1'
[OUTGOING] 'MOVE-30-3'
[INCOMING] 'MOVE-30-3'
[OUTGOING] 'MOVE-3-0'
[INCOMING] 'MOVE-3-0'

```

We can also see who won indicating the end of the game and hence, the test was a success!

To test the client:

Step 1: Start an instance of a client by running the class “ClientSideMethod.java”, Enter the IP address and the port of the reference server to connect it to the reference server. Then enter a unique username too. And then type in “QUEUE” to get in the queue to play the game.

```
Run: ClientSideMethod
C:\Program Files\Eclipse Foundation\jdk-11.0.12-hotspot\bin\java.exe ...
Currently using Arsalan and Kevin's Client
Please provide an IP address or hostname(or leave blank for localhost): 192.168.253.44
Please provide a port number (or leave blank for 55555): 55555
[SERVER] : HELLO PentomGo
Please input a username: somethingthat'snottaken
Login successful!
You can now start inputting commands.
Use "HELP" if you need any.
Currently in Queue.
```

Step 2: Repeat step 1 but with a different username if nobody else is in the queue:

```
File Edit View Navigate Code Refactor Build Run Tools Git Window Help resit-6 - ClientSideMethod.java
resit-6 src \ clientside \ ClientSideMethod
Project
resit-6 C:\Users\Arsalaan Khan\IdeaProjects\resit-6
src
clientside
Client
ServerSideMethod.java ClientSideMethod.java ClientMethod.java OutMethod.java InMethod.java
9 /**
10  * The class containing the main method
11  * to run the client! Starts a thread of
12  * clientMethod!
13  * @author Kevin Schurman and Arsalan Khan
14  */
Run: ClientSideMethod
Login successful!
You can now start inputting commands.
Use "HELP" if you need any.
Currently in Queue.
SENDING: QUEUE
RECEIVED: NEWGAME-SomethingThat'sNotTaken-jiaqi
The new game is starting between the users : [SomethingThat'sNotTaken] and [jiaqi]. May the best player win!
COL0 COL1 COL2 COL3 COL4 COL5
ROW0 | | | | |
-----
ROW1 | | | | |
-----
ROW2 | | | | |
=====
ROW3 | | | | |
-----
ROW4 | | | | |
-----
ROW5 | | | | |
(white) Your Turn
ROW COL ROTATION
```

As you can see, the board is printed and moves can be input!

```
Game is now over because we have a winner! Congratulations to testing12345 for winning the game!
It took 31 moves for this game to end.
COL0 COL1 COL2 COL3 COL4 COL5
ROW0 B | B | W | W | W | W
ROW1 W | B | W | | | B | B
ROW2 B | W | W | W | W | W
ROW3 B | | W | | | B
ROW4 W | W | B | B | B | B
ROW5 W | W | B | B | B | B
```

Also, you can see, it also prints the winner of the game!

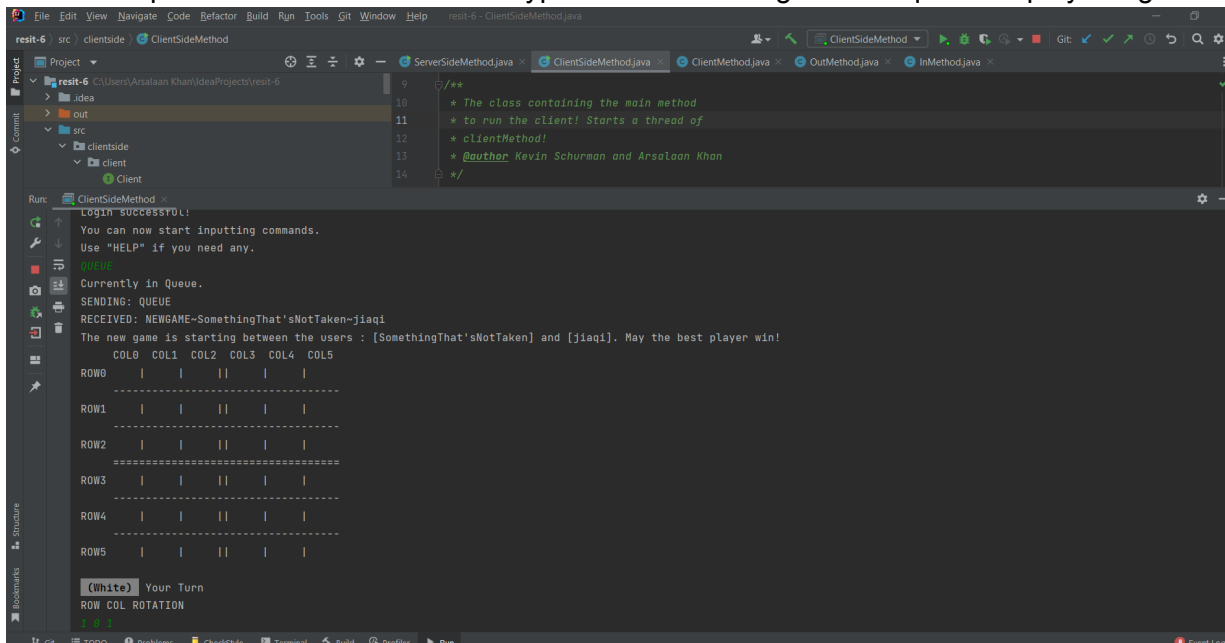
Testing report for IR 2

IR 1 : *The client can play as a human player, controlled by the user*

Expected outcome: The user is able to play as a human player in the client. The client asks the user to input their move and the move is played accordingly on the Pentago board.

Testing result:

Step 1: Start an instance of a client by running the class “ClientSideMethod.java”, Enter the IP address and the port of the reference server to connect it to the reference server. Then enter a unique username too. And then type in “QUEUE” to get in the queue to play the game.



```
ClientSideMethod.java
9  //
10 * The class containing the main method
11 * to run the client! Starts a thread of
12 * clientMethod!
13 * @author Kevin Schurman and Arsalan Khan
14 */

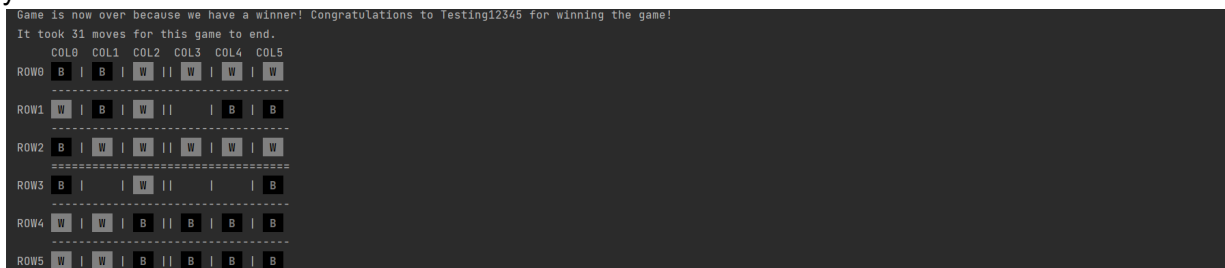
Run: ClientSideMethod
Login successful!
You can now start inputting commands.
Use "HELP" if you need any.

Currently in Queue.
SENDING: QUEUE
RECEIVED: NEWGAME-SomethingThat'sNotTaken-jiaqi
The new game is starting between the users : [SomethingThat'sNotTaken] and [jiaqi]. May the best player win!

COL0 COL1 COL2 COL3 COL4 COL5
ROW0 | | | | |
-----
ROW1 | | | | |
-----
ROW2 | | | | |
=====
ROW3 | | | | |
-----
ROW4 | | | | |
-----
ROW5 | | | | |

(white) Your Turn
ROW COL ROTATION
```

Step 2: Repeat step 1 but with a different username if nobody else is in the queue: By default, human input is taken unless specified otherwise, and you will see you can enter your moves.



```
Game is now over because we have a winner! Congratulations to Testing12345 for winning the game!
It took 31 moves for this game to end.

COL0 COL1 COL2 COL3 COL4 COL5
ROW0 B | B | W | W | W | W
-----
ROW1 W | B | W | | | B | B
-----
ROW2 B | W | W | W | W | W
=====
ROW3 B | | W | | | B
-----
ROW4 W | W | B | B | B | B
-----
ROW5 W | W | B | B | B | B
```

Also prints the winner of the game!

Testing report for IR 3

IR 1 : *The client can play as a computer player, controlled by AI :*

Expected outcome: The user is able to play as a computer player in the client. The client does not ask the user to input their move, and the move is played automatically on the board.

Testing result:

When the user makes an input of either “-N” or “-S”, they will start using the strategy to determine the move.

```
Currently using Arsalaan and Kevin's Client
Please provide an IP address or hostname(or leave blank for localhost):
Please provide a port number (or leave blank for 55555):
[SERVER] : HELLO Arsalaan and Kevin's Server
The supported extensions areRANK
Please input a username: b
Login successful!
You can now start inputting commands.
Use "HELP" if you need any.
-N
Now using naive strategy.
queue
Currently in Queue.
The new game is starting between the users : [REFERENCE_a] and [b]. May the best player win!
```

You can see above that the user has used a Naive strategy in order to play the game.

```
COL0 COL1 COL2 COL3 COL4 COL5
ROW0 W |   |   |   | W | W
-----
ROW1 W |   | B || B | B |
-----
ROW2   | B | B ||   | W | B
=====
ROW3 W |   |   ||   |   | B
-----
ROW4   |   |   ||   |   | B
-----
ROW5   | W | W ||   | W |

Board has no winner and has not been Gamed Over.
Game is now over because we have a winner! Congratulations to REFERENCE_a for winning the game!
It took 17 moves for this game to end.
|
```

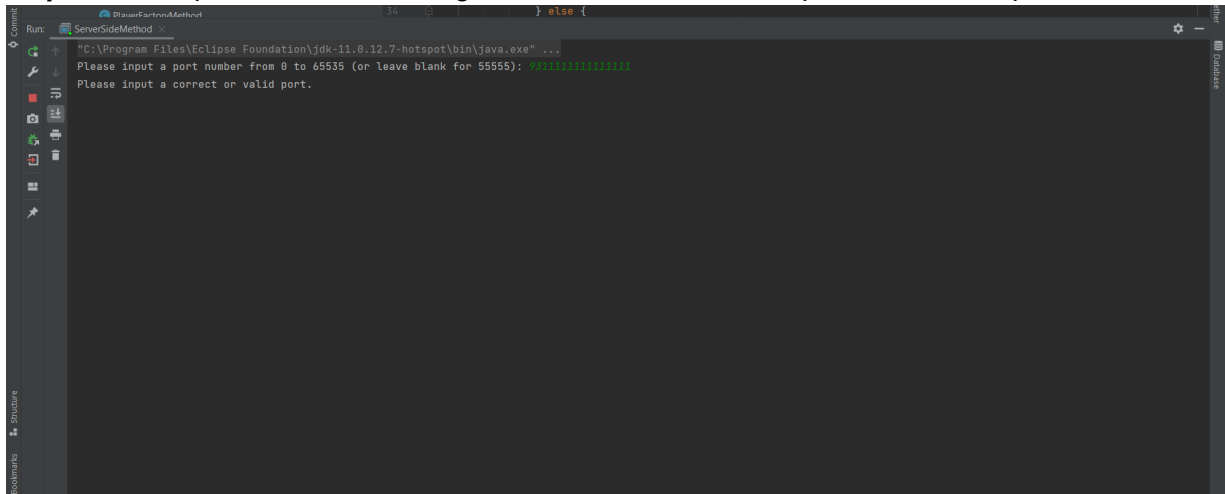
Testing report for IR 4

IR 4 : *When the server is started, it will ask the user to input a port number where it will listen to. If this number is already in use, the server will ask again.*

Expected outcome: The server is started and the server prints a message visible to the user on the console that prompts the user to enter a port it should start at. If the port is already in use, the server will ask for the input again.

Testing result:

Step 1: Start up the server following the instructions in IR1. Input an incorrect port.



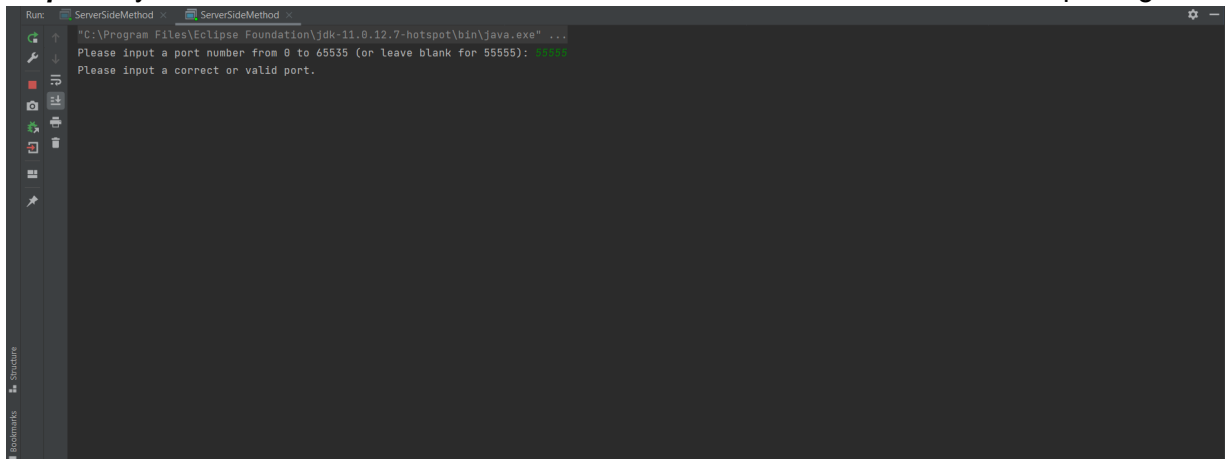
```
Run: ServerSideMethod
"C:\Program Files\Eclipse Foundation\jdk-11.0.12-hotspot\bin\java.exe" ...
Please input a port number from 0 to 65535 (or leave blank for 55555): 8080
Please input a correct or valid port.
```

As you can see, if the port isn't valid, it asks for input again.

Step 2: Start a port on any valid port. The message "Server started!" should be printed.

Step 3: Start another server instance by "allowing multiple run instances" in IntelliJ.

Step 4: Try to start the 2nd server on the same instance, it won't start and ask for port again:



```
Run: ServerSideMethod
"C:\Program Files\Eclipse Foundation\jdk-11.0.12-hotspot\bin\java.exe" ...
Please input a port number from 0 to 65535 (or leave blank for 55555): 8080
Please input a correct or valid port.
```

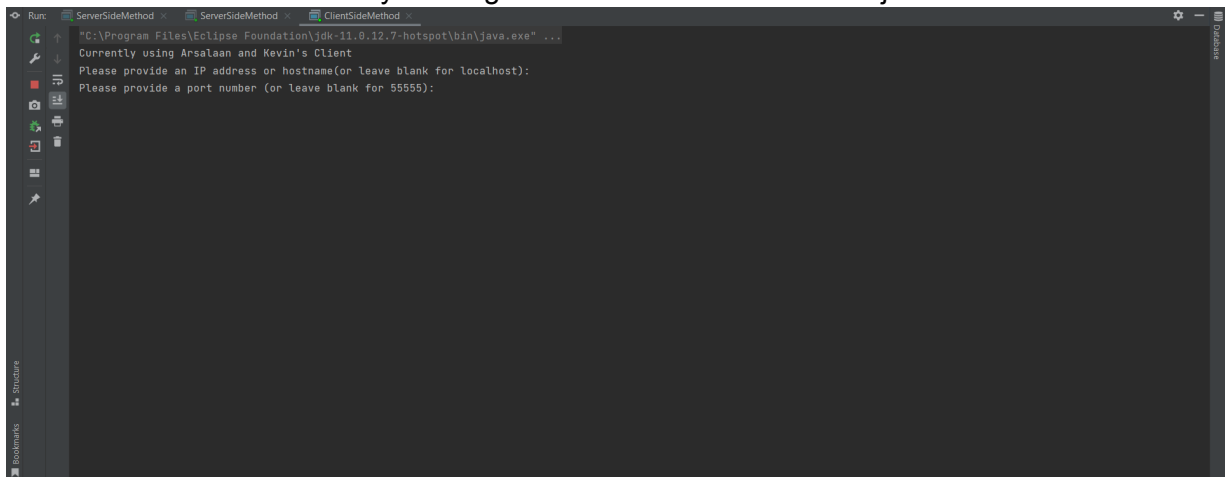
Testing report for IR 5

IR 5 : *When the client is started, it should ask the user for the IP-address and port number of the server to connect to:*

Expected outcome: The client is started and the client prints a message visible to the user that prompts the user to enter the IP address of the server it should connect to. It should then prompt the user to enter the port of the server. After that, it should try to connect to that server and inform the user if the connection failed and prompt the user again.

Testing result:

Start an instance of a client by running the class “ClientSideMethod.java”



```
C:\Program Files\Eclipse Foundation\jdk-11.0.12.7-hotspot\bin\java.exe" ...  
Currently using Arsalan and Kevin's Client  
Please provide an IP address or hostname(or leave blank for localhost):  
Please provide a port number (or leave blank for 55555):
```

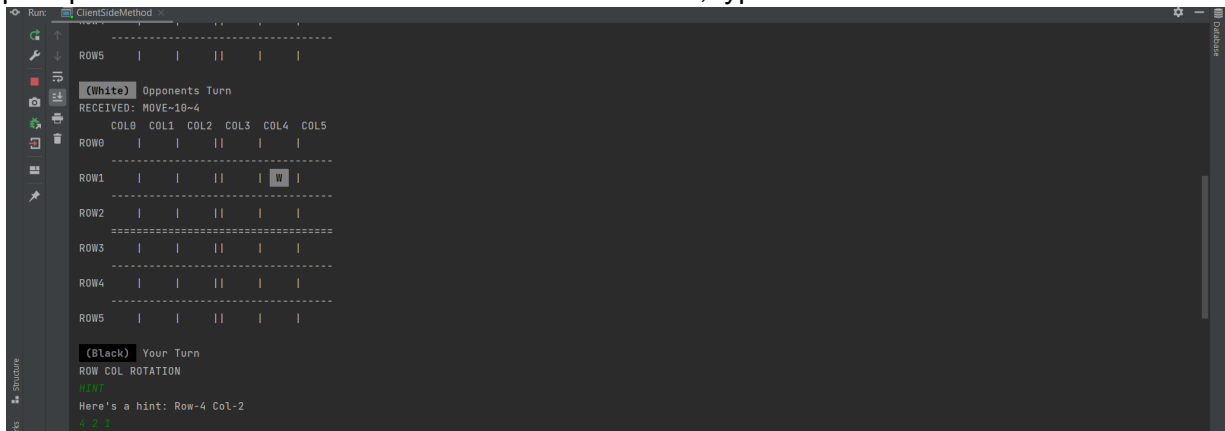
As you can see, it will ask for an IP address and the port. First it will ask for the IP address and then the port.

Testing report for IR 6

IR 6 : *When the client is controlled by a human player, the user can request a possible legal move as a hint via the TUI*

Expected outcome: When the client is in a game and the client prompts the client to enter a move, instead of entering the move, the client can request for a hint which will print out a possible legal move on the console which the client can see and then prompt the user to enter his/her move again.

Testing result: To play as a human player, follow steps 1 and 2 from IR 2. When the client prompts the user to enter the row column and rotation, type "HINT" instead.



The screenshot shows a Java Swing window titled "ClientSideMethod" with a dark background. On the left, there's a sidebar with icons for Run, Debug, and other IDE functions. The main area displays a game board with 6 rows (ROW0 to ROW5) and 6 columns (COL0 to COL5). The board is mostly empty, with a few pieces placed. Below the board, there's a console window showing the following text:

```
(White) Opponents Turn
RECEIVED: MOVE-10-4
COL0 COL1 COL2 COL3 COL4 COL5
ROW0 | | | | | |
ROW1 | | | | | |
ROW2 | | | | | |
ROW3 | | | | | |
ROW4 | | | | | |
ROW5 | | | | | |
(Black) Your Turn
ROW COL ROTATION
Here's a hint: Row-4 Col-2
```

As we can see, when "HINT" was typed, the client was given a possible legal move as the hint.

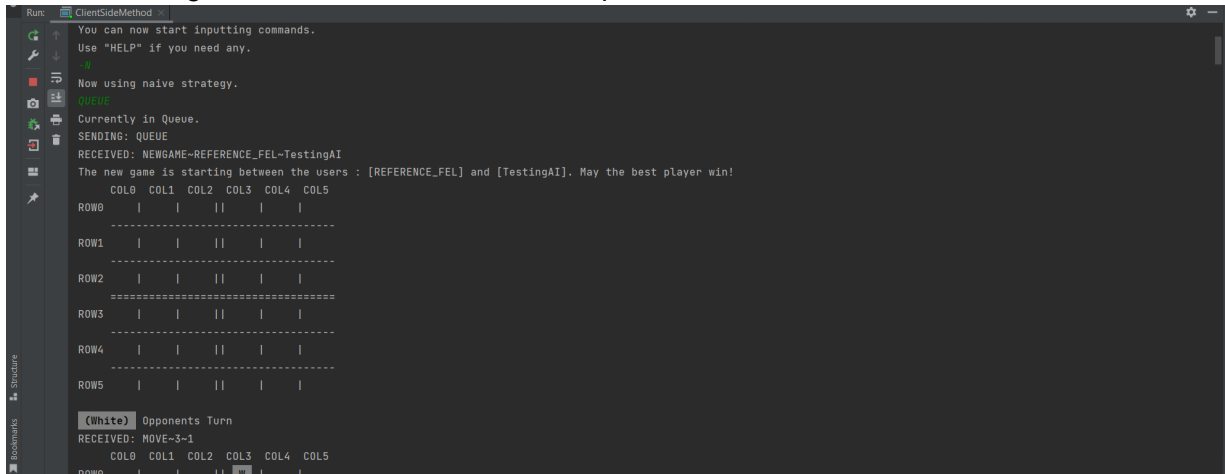
It does not give rotation since all rotation between 1-7 is valid anyway and printing a random number between 1-7 is useless.

Testing report for IR 7

IR 7: *The client can play a full game automatically as the AI without intervention by the user*

Expected outcome: The client can queue for a game as an AI player and when the game starts, the client is not prompted to input his or her move and instead the moves are made automatically by the bot.

Testing result: To queue as an AI, follow step 1 from IR 2 but before typing “QUEUE”, enter “-N” to change your player to a bot. After that is done, the client will send a message to the user confirming it. After that is done, start the queue.



```
Run: ClientSideMethod
You can now start inputting commands.
Use "HELP" if you need any.

Now using naive strategy.

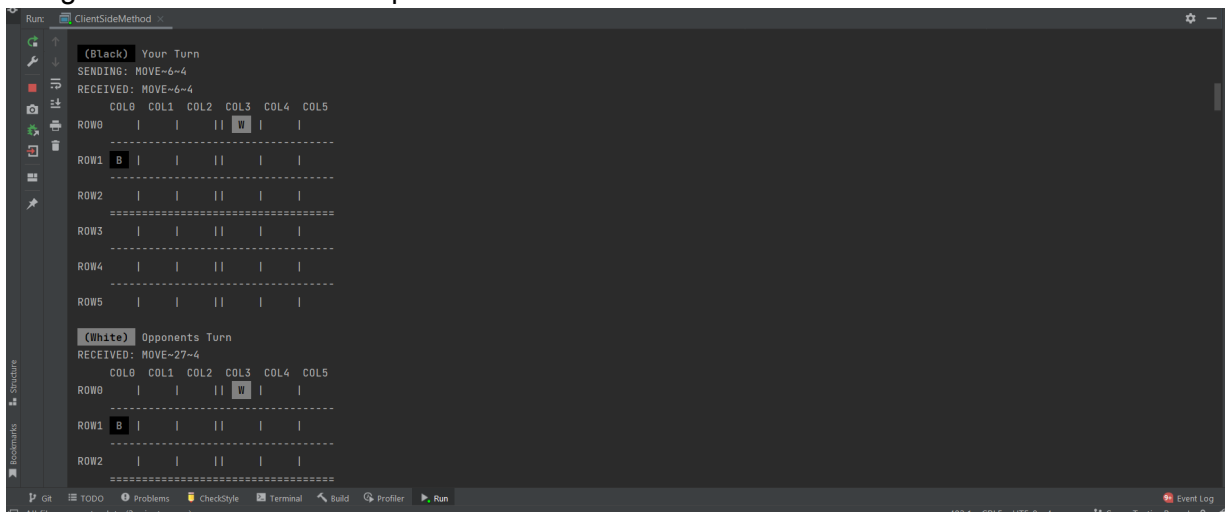
Currently in Queue.
SENDING: QUEUE
RECEIVED: NEWGAME-REFERENCE_FEL-TestingAI
The new game is starting between the users : [REFERENCE_FEL] and [TestingAI]. May the best player win!

COL0 COL1 COL2 COL3 COL4 COL5
ROW0 | | | | | |
-----
ROW1 | | | | | |
-----
ROW2 | | | | | |
=====
ROW3 | | | | | |
-----
ROW4 | | | | | |
-----
ROW5 | | | | | |

(White) Opponents Turn
RECEIVED: MOVE-3-1

COL0 COL1 COL2 COL3 COL4 COL5
ROW0 | | | | | |
-----
ROW1 | | | | | |
-----
ROW2 | | | | | |
=====
ROW3 | | | | | |
-----
ROW4 | | | | | |
-----
ROW5 | | | | | |
```

The game will then not take input and move for me.



```
Run: ClientSideMethod
(Black) Your Turn
SENDING: MOVE-6-4
RECEIVED: MOVE-6-4

COL0 COL1 COL2 COL3 COL4 COL5
ROW0 | | | | | |
-----
ROW1 | B | | | | |
-----
ROW2 | | | | | |
=====
ROW3 | | | | | |
-----
ROW4 | | | | | |
-----
ROW5 | | | | | |

(White) Opponents Turn
RECEIVED: MOVE-27-4

COL0 COL1 COL2 COL3 COL4 COL5
ROW0 | | | | | |
-----
ROW1 | B | | | | |
-----
ROW2 | | | | | |
=====
ROW3 | | | | | |
-----
ROW4 | | | | | |
-----
ROW5 | | | | | |
```

As you can see, no input was taken and my move was sent automatically.

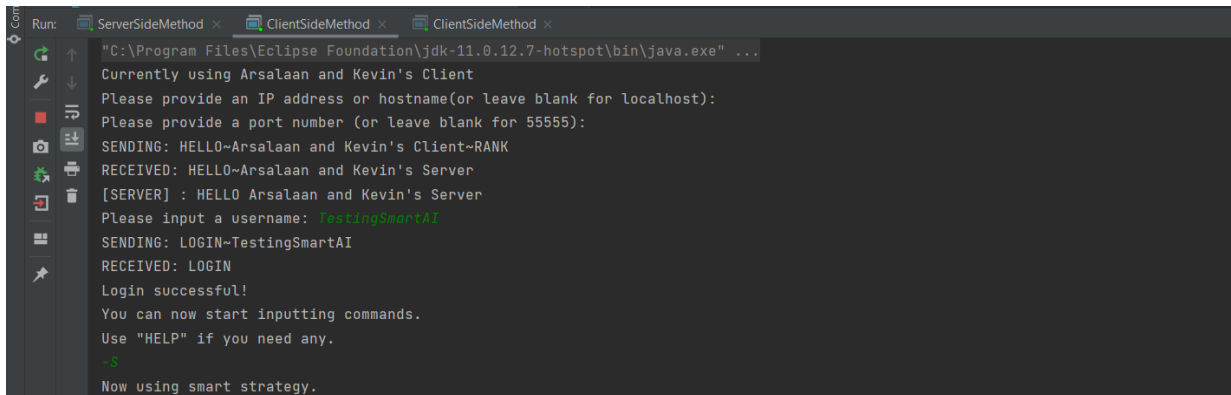
Testing report for IR 8

IR 8 : *The AI difficulty can be adjusted by the user via the TUI*

Expected outcome: The client can specify what difficulty of AI it wants to queue as when playing the game.

Testing result:

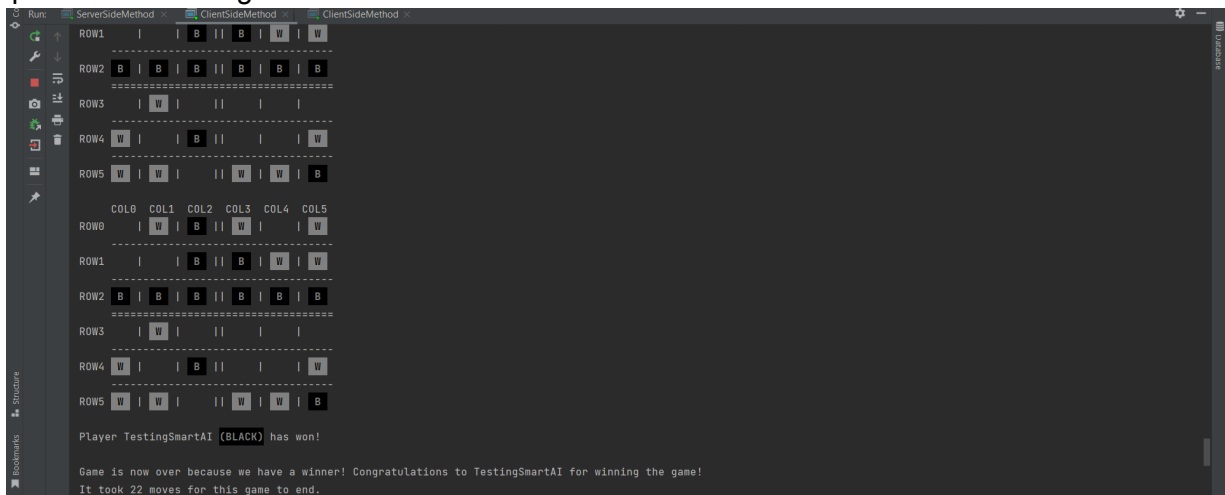
Step 1: Follow the steps in IR 7 but instead of entering “-N”, you can enter “-S” to have a smart bot instead.



```
Run: ServerSideMethod x ClientSideMethod x ClientSideMethod x
"C:\Program Files\Eclipse Foundation\jdk-11.0.12.7-hotspot\bin\java.exe" ...
Currently using Arsalaan and Kevin's Client
Please provide an IP address or hostname(or leave blank for localhost):
Please provide a port number (or leave blank for 55555):
SENDING: HELLO~Arsalaan and Kevin's Client~RANK
RECEIVED: HELLO~Arsalaan and Kevin's Server
[SERVER] : HELLO Arsalaan and Kevin's Server
Please input a username: TestingSmartAI
SENDING: LOGIN~TestingSmartAI
RECEIVED: LOGIN
Login successful!
You can now start inputting commands.
Use "HELP" if you need any.
-S
Now using smart strategy.
```

As you can see, the client confirms to the user that smart strategy has now been chosen. And then join the queue by typing “queue”.

Step 2: Start another instance of a client, but type “-N” before queuing this time. And then queue and let the game start.



```
Run: ServerSideMethod x ClientSideMethod x ClientSideMethod x
ROW1 | | B | | B | | W | |
ROW2 | B | | B | | B | | B | | B |
=====
ROW3 | W | | | | | |
ROW4 | | | B | | | | |
ROWS | W | W | | | W | W | B |
COL0 COL1 COL2 COL3 COL4 COL5
ROW0 | W | B | W | | W |
ROW1 | | B | | B | W |
ROW2 | B | B | B | B | B | B |
=====
ROW3 | W | | | | |
ROW4 | W | | B | | | W |
ROWS | W | W | | W | W | B |
Player TestingSmartAI (BLACK) has won!
Game is now over because we have a winner! Congratulations to TestingSmartAI for winning the game!
It took 22 moves for this game to end.
```

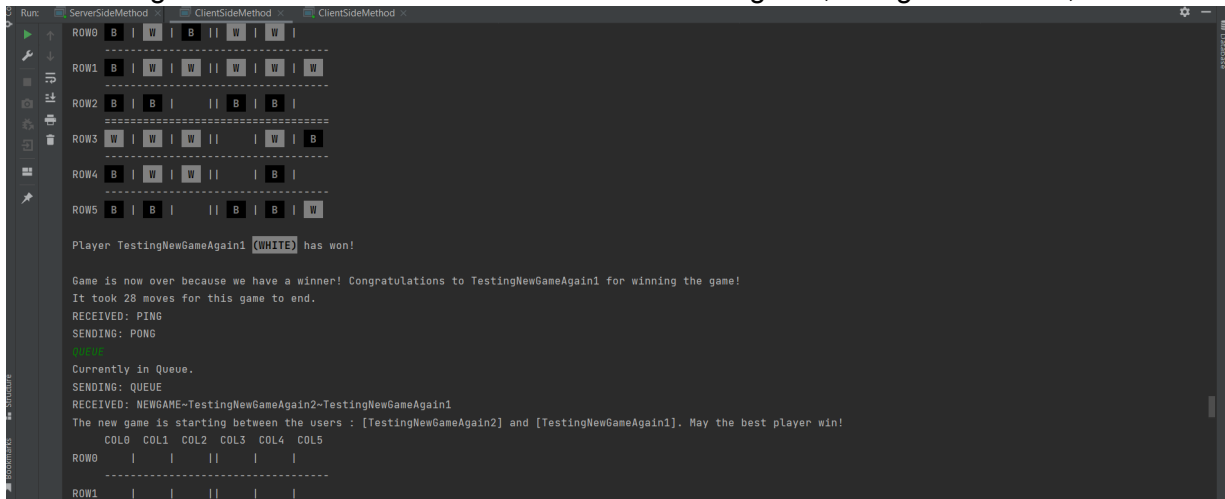
As you can see, the smart AI has won against the Naive AI, proving that the difficulty of the AI has increased. The test was a success!

Testing report for IR 9

IR 9 : *Whenever a game has finished (except when the server is disconnected), a new game can be played without needing to establish a new connection in between*

Expected outcome: The client can join a queue again after it has finished a game, and a new game will be started again when there are at least 2 clients waiting in the queue.

Testing result: Started a game as explained in other tests and after the game is over, type “QUEUE” again and then once there were 2 clients in the game, the game started,



```
Run: ServerSideMethod ClientSideMethod ClientSideMethod
ROW0 B | W | B | W | W |
ROW1 B | W | W | W | W | W
ROW2 B | B | | | B | B |
ROW3 W | W | W | | W | B
ROW4 B | W | W | | B |
ROW5 B | B | | B | B | W
Player TestingNewGameAgain1 (WHITE) has won!
Game is now over because we have a winner! Congratulations to TestingNewGameAgain1 for winning the game!
It took 28 moves for this game to end.
RECEIVED: PING
SENDING: PONG
ping
Currently in Queue.
SENDING: QUEUE
RECEIVED: NEWGAME-TestingNewGameAgain2-TestingNewGameAgain1
The new game is starting between the users : [TestingNewGameAgain2] and [TestingNewGameAgain1]. May the best player win!
COL0 COL1 COL2 COL3 COL4 COL5
ROW0 | | | | |
ROW1 | | | | |
```

As you can see, the game started when we queued again!

Testing report for IR 10

IR 10 : *All communication outside of playing a game, such as handshakes and feature negotiation, works on both client and server in conjunction with the reference server and client, respectively*

Expected outcome: The reference client is able to successfully complete handshakes and see what features the server supports. The reference server is able to successfully complete handshakes and see what features the client supports.

Testing result:

To start the reference client and connect to our server, follow the steps 1-3 in testing the server of IR1.

```
Command Prompt - java -jar client.jar
Microsoft Windows [Version 10.0.22000.434]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Arsalaan Khan>cd desktop

C:\Users\Arsalaan Khan\Desktop>java -jar client.jar
Please provide a port number (leave blank for 55555):
Please provide an IP address or hostname (leave blank for localhost):
[OUTGOING] 'HELLO~Pentago reference client v1.0'
[INCOMING] 'HELLO~Arsalaan and Kevin's Server'
Please give a username: TestingNegotiation
[OUTGOING] 'LOGIN~REFERENCE_TestingNegotiation'
[INCOMING] 'LOGIN'
[OUTGOING] 'QUEUE'
[INCOMING] 'PING'
[OUTGOING] 'PONG'
```

As can be seen, the handshake was a success, which allowed us to queue!

To start our client and connect to the reference server, follow the step 1-3 in testing the client of IR1.

```
Run: ClientSideMethod
C:\Program Files\Eclipse Foundation\jdk-11.0.12.7-hotspot\bin\java.exe ...
Currently using Arsalaan and Kevin's Client
Please provide an IP address or hostname(or leave blank for localhost): 192.168.1.101
Please provide a port number (or leave blank for 55555):
SENDING: HELLO~Arsalaan and Kevin's Client-RANK
RECEIVED: HELLO~PenTomGo-CRYPT-AUTH-RANK-CHAT
[SERVER] : HELLO PenTomGo
Please input a username: TestingHandshakes
SENDING: LOGIN~TestingHandshakes
RECEIVED: LOGIN
Login successful!
You can now start inputting commands.
Use "HELP" if you need any.
```

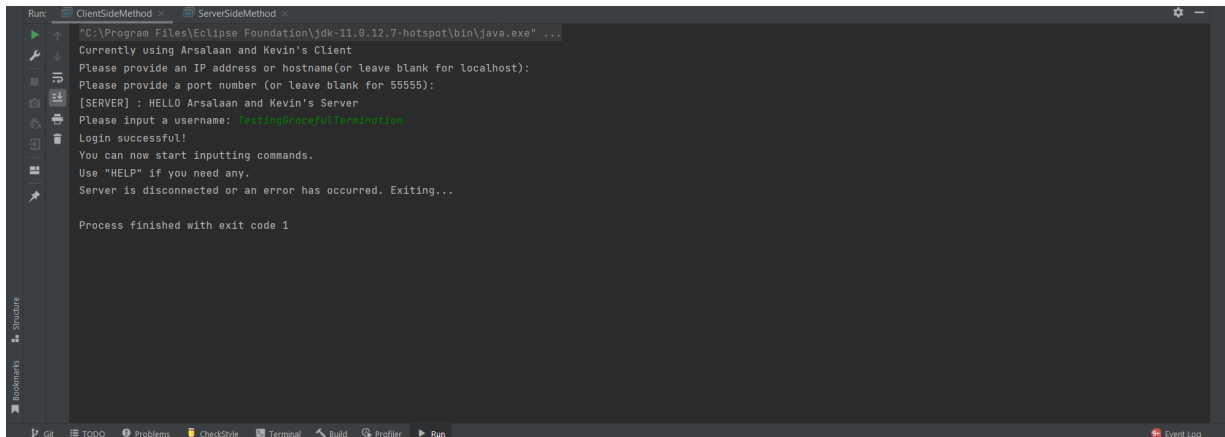
As can be seen, the handshake is successful, and we can start inputting commands!

Testing report for IR 11

IR 11 : *Whenever a client loses a connection to a server, the client should gracefully terminate*

Expected outcome: If the server is stopped for some reason, the client informs the user about the loss of connection and exits cleanly. No error stack trace is seen by the client.

Testing result: Start our server and our client and connect them to each other. If instructions are unclear on how to do that, refer to the earlier testing results in this document. Stop the execution of the server and look at the client's console to check for errors.



```
Run: ClientSideMethod, ServerSideMethod
"C:\Program Files\Eclipse Foundation\jdk-11.0.12-hotspot\bin\java.exe" ...
Currently using Arsalaan and Kevin's Client
Please provide an IP address or hostname(or leave blank for localhost):
Please provide a port number (or leave blank for 55555):
[SERVER] : HELLO Arsalaan and Kevin's Server
Please input a username: ArsalaanKevin
Login successful!
You can now start inputting commands.
Use "HELP" if you need any.
Server is disconnected or an error has occurred. Exiting...

Process finished with exit code 1
```

As you can see, the client prints the message “Server is disconnected or an error has occurred. Exiting....” which indicates graceful termination since it doesn’t crash and no stack trace is seen by the user. The test was a success.

NOTE: Debugger mode was turned off for this. It is off by default unless specified otherwise in the program arguments. Please look at the first paragraph of “System Tests” in this report for more details.

Testing report for IR 12

IR 12 : *Whenever a client disconnects during a game, the server should inform the other clients and end the game, allowing the other player to start a new game.*

Expected outcome: When 2 clients are in a game and one of them disconnects, the other client is informed of the disconnect, gets the win and can still queue up to start a new game.

Testing result: Connect the server and the client using the steps explained in earlier system tests. Have 2 clients connected to the server and queue up for a game

```
Please provide an IP address or hostname(or leave blank for localhost):
Please provide a port number (or leave blank for 55555):
[SERVER] : HELLO Arsalaan and Kevin's Server
The supported extensions areRANK
Please input a username: Test
Login successful!
You can now start inputting commands.
Use "HELP" if you need any.
Game
Currently in Queue.
The new game is starting between the users : [Test] and [TestingDisconnection]. May the best player win!
  COLO COL1 COL2 COL3 COL4 COL5
ROW0  |  |  |  |  |  |
-----
ROW1  |  |  |  |  |  |
-----
ROW2  |  |  |  |  |  |
=====
ROW3  |  |  |  |  |  |
-----
ROW4  |  |  |  |  |  |
-----
ROW5  |  |  |  |  |  |

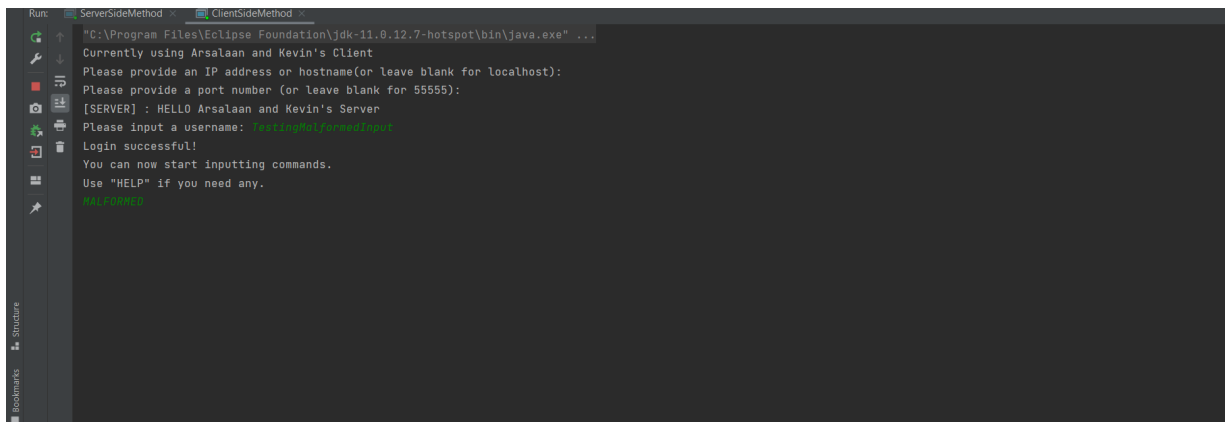
(White) Your Turn
ROW COL ROTATION
```

As you can see below, once the client has exited, the board will be printed again and say that the board has no winners. Since the current client user hasn't disconnected, the user will be told that the game as ended with the disconnection.

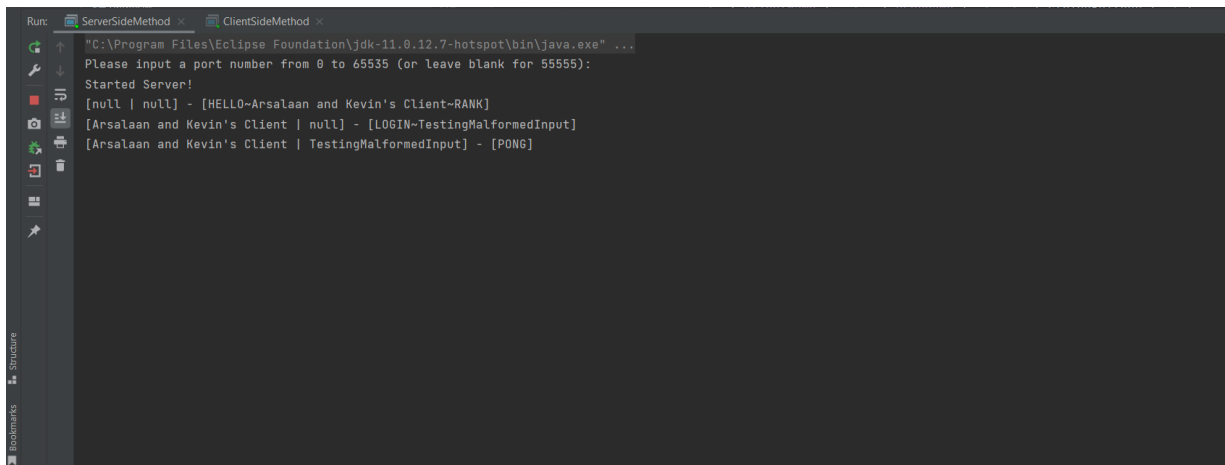
This was the system testing for all the important requirements, however, there is one more test that we thought would be appropriate. It is the client's and server's correct handling of malformed inputs by the client. This is an extra requirement and would be referred to as ER.

Testing report for ER 1
IR 12 : <i>The client or the server does not crash due to the user's malformed input, and the client does not forward malformed input to the server.</i>
Expected outcome: When the user doesn't correctly format his or her input, the client doesn't forward it to the server (so it doesn't affect the game), doesn't crash and informs the user about the incorrect input.

Testing result: Connect our client and server with each other by referring to the other system tests' steps and wait in the lobby, don't queue for a game. Enter something unexpected like "MALFORMED".

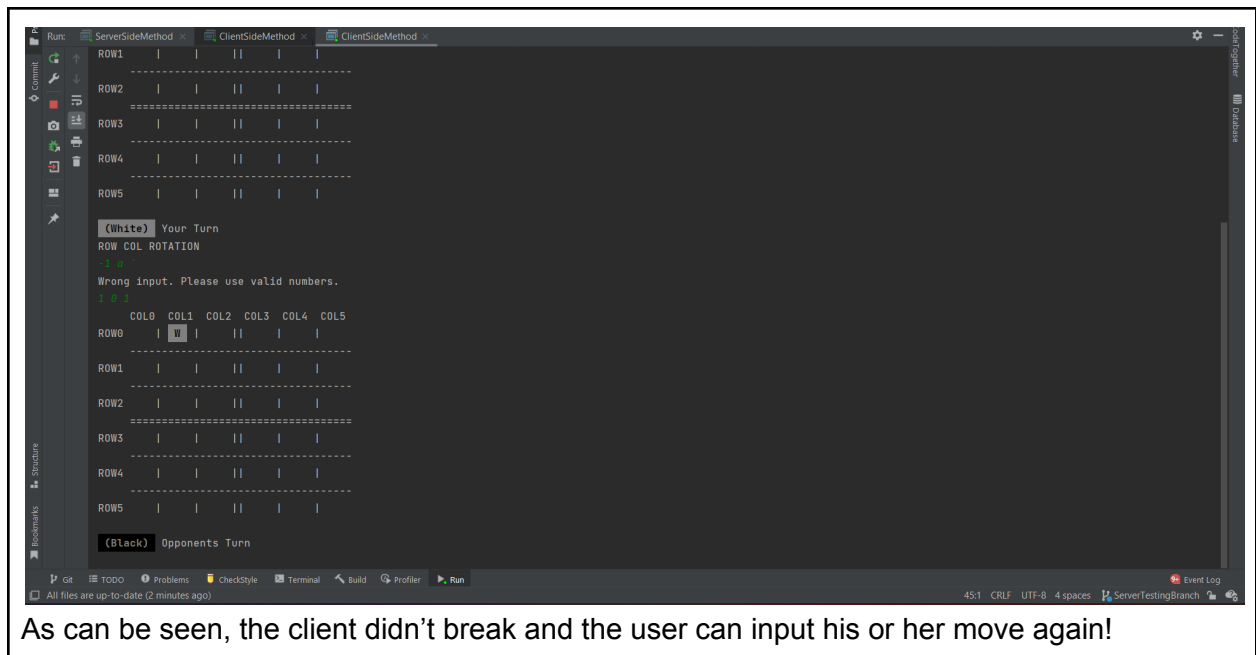


```
Run: ServerSideMethod x ClientSideMethod x
"C:\Program Files\Eclipse Foundation\jdk-11.0.12-hotspot\bin\java.exe" ...
Currently using Arsalaan and Kevin's Client
Please provide an IP address or hostname(or leave blank for localhost):
Please provide a port number (or leave blank for 55555):
[SERVER] : HELLO Arsalaan and Kevin's Server
Please input a username: TestingMalformedInput
Login successful!
You can now start inputting commands.
Use "HELP" if you need any.
MALFORMED
```



```
Run: ServerSideMethod x ClientSideMethod x
"C:\Program Files\Eclipse Foundation\jdk-11.0.12-hotspot\bin\java.exe" ...
Please input a port number from 0 to 65535 (or leave blank for 55555):
Started Server!
[null | null] - [HELLO-Arsalaan and Kevin's Client-RANK]
[Arsalaan and Kevin's Client | null] - [LOGIN-TestingMalformedInput]
[Arsalaan and Kevin's Client | TestingMalformedInput] - [PONG]
```

The client and the server are both stable, as can be seen in the images. Then queue up for a game, and in the game, enter an invalid input when asked for a move like "-1 a `"



As can be seen, the client didn't break and the user can input his or her move again!

Overall Testing Strategy

We have divided our system into three main parts. The client side, the server side and the game logic. In this section, we will discuss how we tested each of these thoroughly. Automated unit testing and system testing was done to thoroughly and extensively test the system.

For the edge cases, we don't have anything that tests the edge cases but not the other fields and vice versa because in almost all test cases, we are testing the entire board itself to ensure thorough testing which ensures that under no circumstances, our game logic fails. However, there are a few instances where the tests do complement each other. For example, the methods `testHasRow`, `testHasCol` and `testHasDiag` test if the player has a row, column or diagonal respectively anywhere on the board but does not check if the board has a winner or the game has ended. But, the `testHasWinnerDiagonal`, `testHasWinnerCol` and `testHasWinnerRow` do make sure that there is a winner when a player has a winner, column or a row.

Our unit tests have extensively tested the board, ensuring that under no circumstances our game logic fails. We have done this by testing almost every possible situation in every test case. For example, the `hasRow` method tests if a marble has 5 in a row in any of the rows on the board or not. For this, we have tested every possible combination of having 5 marbles in a row and tests if the board indeed has 5 in a row or not. We have a similar approach in almost all our test cases. We also have a class called "GameMethodTest" which tests the game logic using the `PrintWriter` output to replicate what the client would be inputting, and we have ensured that correct results are obtained every time. We also have a method in the class called `testRandom()` which makes random legal moves on the board and tests that the game is indeed eventually over, and the test passes all the time.

Since we have tested our game logic so thoroughly using the automated unit tests, we were worried about the effectiveness of our networking part of the game. However, we are now very confident in it too since almost all our system tests focus on the networking part of the game, extensively testing the client-server architecture we have built and also testing malformed inputs and random disconnects. In this way, the unit tests and the system tests perfectly complement each other.

However, since we have no automation testing for the server and the client, I could also say that it is the least tested part of our system. Automation testing goes through a lot of possibilities in a short period of time and tests all of them, which is difficult to do in system testing. But, that doesn't mean that our client and server architecture is unstable because the system testing has thoroughly covered that, but it could be better to write automated tests for the client and the server as well.

For the automated tests, we have 2 test classes that have JUnit test cases. The first one is the `BoardMethodTest` class which tests the class `board.java` which mainly implements the board. The test case has a method coverage of 86% and line coverage of 82%. This is really high

coverage which indicates that the test covers almost everything. The methods and the line coverage would have been higher, but the trivial getters have not been tested in the test class.

The other test class called `GameMethodTest.java` tests if the game is played correctly and is as expected when played by the client by using a `PrintWriter` to replicate a client inputting data for the game to process. That class has a coverage of 96% of methods in the class `BoardMethod` and 93% of the lines in that class. These numbers can clearly justify our automation testing to be more than enough and proves that we have an extremely stable system, at least when it comes to handling the game logic.

To measure the complexity of our various methods and classes, we have used an IntelliJ plugin called "Code Metrics" which measures the Cyclomatic complexity of the various methods and classes of the system. Our board class had a complexity of 39 which was indicated to be extremely high, but our automated tests have thoroughly covered that. Other classes like all the exception classes, the `NaiveStrategy` class have not been tested, but they also had a very low complexity score which might mean that they don't even need much testing because they are not very complex and are just straightforward in what they are supposed to do.

Reflection on Process

Arsalaan Khan

This project was one of the best things that happened to me. I honestly thought I knew how to code, but this project taught me that there is still a lot more to learn. Initially, what we had in mind was something really simple and underestimated the complexity of the project, which was probably our biggest mistake in my opinion. We didn't really plan much and just went along with the process without having much idea of where we were heading. This made our code extremely messy. We spent a lot of time debugging the code we had written instead of writing new code. Because our design choices were poor, we had to go back to everything we had written and write it again and slowly read through it to find any possible bugs, which took a lot of our days. The biggest learning experience here would be to write quality code because it might take longer to write quality code, but the time spent debugging would be saved immensely and honestly, debugging takes up a lot of the time. Next time, I will be working on a project. I will ensure that I think about the whole design of the system before writing a single line of code. Also, I will write quality code without just writing code to fix a temporary problem, but rather think a bit longer to look at the bigger picture. Also, one more thing I could work on would be time management. We thought that splitting the time equally between the game logic and the networking part would be a great idea. But, it wasn't since the networking part turned out to be a lot more complicated than expected, and we became a bit complacent to work once we had the game logic working since we thought we had already done quite a bit of work. This project definitely taught me how to think like a programmer and the next time, I will always look at the bigger picture instead of focusing on just temporary issues to solve.

Kevin Schurman

This project was certainly an interesting case for me, not because I have enjoyed coding and the experience (there have been times when I get into the mood and others when I resent it), but because it was a long and exhausting experience. I have done some coding projects before, but the difference between then and now is when I do it by myself at my own pace and time, and the lack of a deadline. It's fascinating because there are days when I spend hours with my partner and I feel like we both make lots of progress, and thus it feels great, then there are other days when it feels slow, lacklustre, and we have made either no progress or the illusion was there. Even though I am aware that sleeping does lead to ideas for the next day I wouldn't have thought about the day prior, it was definitely a lot more notable during this project. We have had minor incidents where we would spend the whole night working on getting the issue fixed, but they were few and far between. I feel like I spent a lot of the time coding the project and fixing the structure whereas my partner did a lot more of the JML, Javadoc and documentation in the project. Next time, I'll make sure we are both even when it comes to this sort of case, since it would be great if my partner could also learn with me when I am able to figure out some issues.

I also think I could improve with this project later with the structure and approach to the implementation, since we had to rewrite a couple packages or entire classes because we realized the issue that was happening. I also think I can improve by figuring out how synchronization works in Java; not the signals, locks or the thread sleeps or the other such since I fully understand how they work, but more specifically the keyword inbuilt to Java, so I can be more comfortable using it more often.