Programming Project Module 2 TCS

Minor-2
Marinus Bos (s2827603)
Ylona van de Kerk (s2613468)

January 2023

Contents

1 Introduction

In this report the design and implementation are explained of a Dots and Boxes game. The structure of the program is explained by giving a description of the classes and their relation. Furthermore, the concurrency mechanism of the program is explained. The improvement from the initial design of the game is discussed and it is explained how the program is tested to ensure that it meets all the requirements. Lastly, we reflect on the working methods.

2 Realised design

2.1 Classes and Packages

The different classes with a summary of its main responsibility:

1. Game: implements the rules of the Dots and Boxes game, used by both client and server.
2. Board: represents the current state of the board used in the Dots and Boxes game.
3. Player (abstract): used as generic input for moves.
4. HumanPlayer: ask for a move via the console.
5. AIPlayer: give moves via a computer Strategy.
6. Strategy (interface): interface for computer Strategies.
7. NaiveStrategy: a very simple Strategy.
8. SmartStrategy: a more competent Strategy.
9. Protocol: contains constants for communicating between client and server.
10. SocketConnection (abstract): contains the basis for communicating via a socket.
11. ClientConnection: uses the Protocol to communicate with a server using the framework of SocketConnection.
12. Client: manages communication between a server and local interfaces. Furthermore, it keeps track of the state of the game.
13. ClientListener (interface): interface for local classes to receive updates about the state of the game.
14. ClientTUI: text-based user interface for the client. Also responsible for instantiating the client. Uses Player to get input for moves. Get updates from the Client via the ClientListener interface. Instantiates itsself in Human mode
15. AIClientTUI: used to instantiate the ClientTUI in AI mode.
16. ClientMoveInput (interface): interfaces that can handle a move input request.
17. NonBlockingScanner: reads lines from an input stream without blocking.
18. UIState (enum): A list of states that can be used by a UI. The clientTUI uses this enum class as a representation of the state of the client.
19. SocketServer (abstract): contains the basis for creating a server for clients to connect to via a socket.
20. Server: handles incoming connections from clients and contains a queue for matching clients and starting a game. Creates a ServerGameManager when two clients are in the queue.
21. ServerGameManager: manages a game between two clients.
22. ServerClientHandler: manages the communication between a client and the server. Communicates with the ServerGameManager if a game is currently running.
23. ServerConnection: uses the Protocol to communicate with a client using the framework of SocketConnection.

The classes are connected as described in the class diagrams in figure 1 and figure 2.

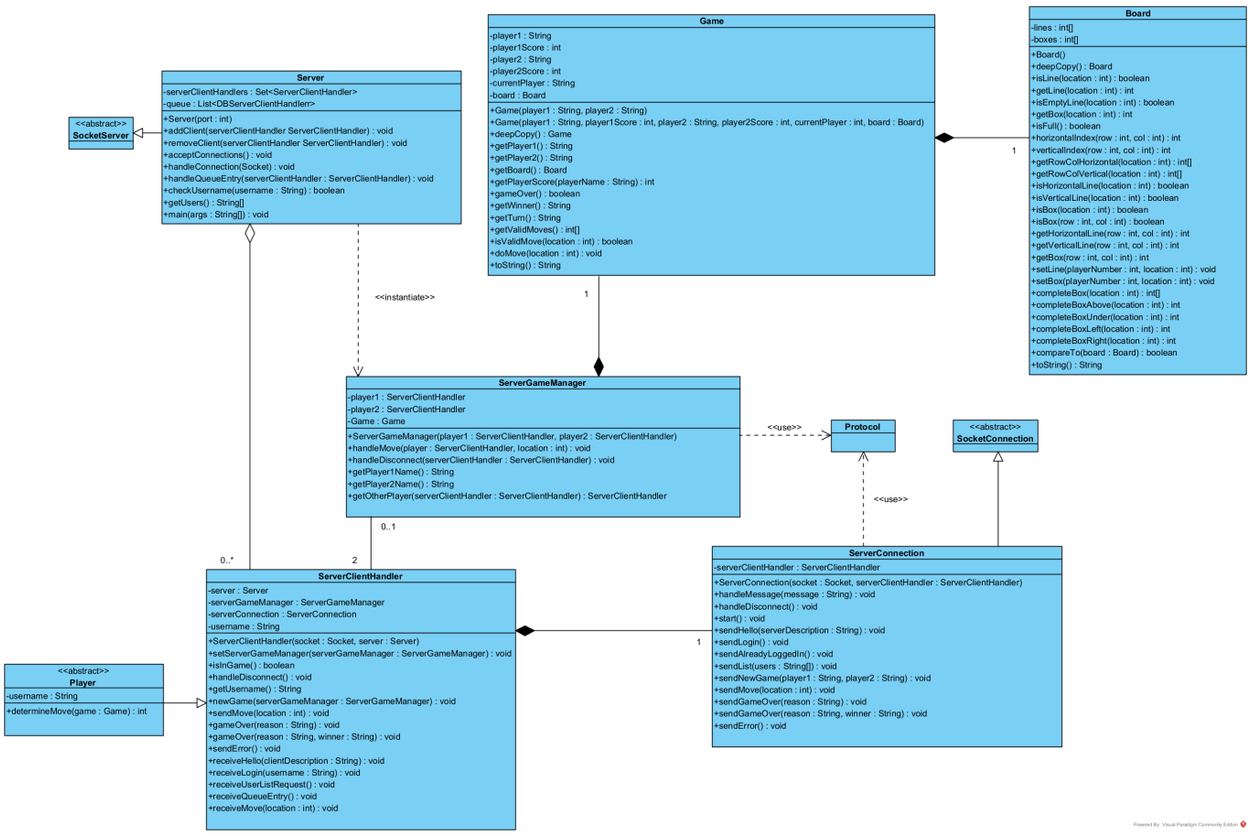


Figure 1: Class diagram of the server

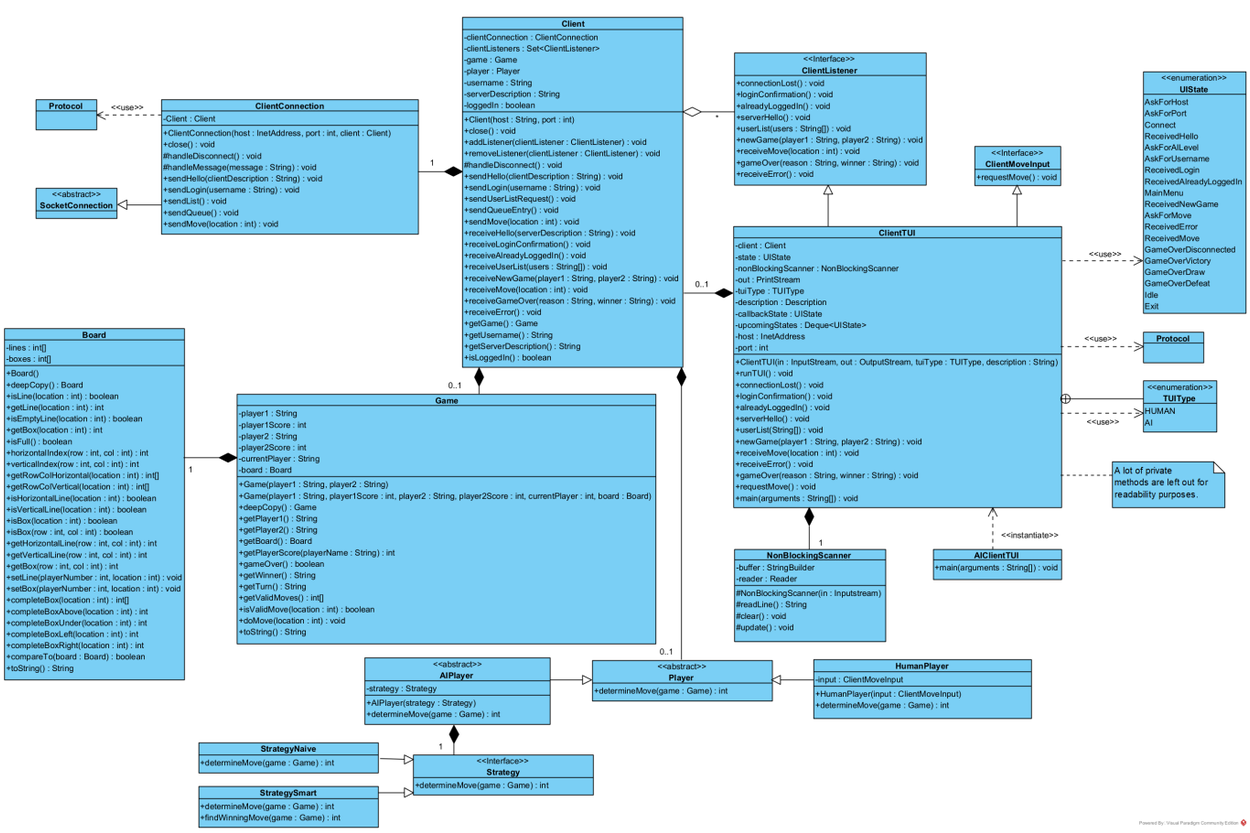


Figure 2: class diagram of the client

The classes are structured in packages as follows:
Game:

- Game
- Board

Players:

- Player (abstract)
- HumanPlayer
- AIPlayer

- Strategy (interface)
- NaiveStrategy
- SmartStrategy

Server:

- SocketServer (abstract)
- Server
- ServerGameManager
- ServerClientHandler
- ServerConnection

Client:

- ClientConnection
- Client
- ClientListener (interface)
- ClientTUI
- ClientMoveInput
- NonBlockingScanner
- UIState
- AIClientTUI

Networking:

- Protocol
- SocketConnection (abstract)

This structure is chosen to separate the different parts of the program. All the game logic is put in the game package, all the classes that are used for constructing the players (AI and Human) are put in the players package, all the classes that are used in the implementation of the server are put into the server package, all the classes that are used in the implementation of the client are put into the client package and lastly the classes that are used as communication between the server and the client are put into the networking package. Structuring the program like this ensures separation of concerns, which means that the implementations of the different parts of the Dots and Boxes game do not influence classes outside their package. Breaking down the code like this also makes it easier to implement and increases the readability of it.

Relationships between classes:

- Game package: The board class is used by the game class. Every game object creates a new board upon construction.
- Players package: Every player object is able to determine a move, since they all implement the method determineMove from the abstract class Player. Two types of players are implemented, that is a human and an AI player. The AI player has an attribute strategy, which is used to determine a move. This strategy is a class that implements the interface Strategy.
- Server package: The server extends the class SocketServer and creates a server client handler object to represent every client that wants to connect to the server. Furthermore, it creates a server game manager for every game that is started between two clients, and it keeps track of all the clients that are logged into the server and what clients are waiting to start a game. Every server client handler object has a connection which is used to send messages to the client using the Protocol. This connection class extends the class SocketConnection. Once the server decides to start a game, the players of this game get a server game manager assigned, which creates a new game.
- Client package: Every client has a connection which is used to send messages to the server using the Protocol. This connection class extends the class SocketConnection. Every client represents a player. The type of player is represented by the Player class. The client creates a (new) game object, once it receives the message from the server that a new game has started with the client as one of the players. The client also keeps track of so called listeners, that want to be notified of any state changes of the client and game. One of the listener is the client TUI which is a textual interface for the user. If the client represents a human player, a move is determined via the client TUI that asks the user to input a move.
- Networking package: The classes inside this package are used to communicate between a server and a client. As mentioned earlier the client and server both have a connection class that extend the SocketConnection class and uses the protocol to send messages.

The following design pattern are used:

- MVC
- Strategy
- Listener
- the creational pattern: Factory

The MVC pattern is used in the design of the client. The class ClientTUI represents the view, the Client class represents the controller of the model, that is represented by the Game and Board class.

We used the Strategy pattern in the design of the computer player. The computer player determines moves by using a specific strategy. For now, we have defined two strategies: NaiveStrategy and SmartStrategy. Both these classes implement the interface strategy. To make the computer player smarter, we can create another strategy that implements the interface strategy. This way you can easily make the computer player use another strategy, without having to chance the code of the AIPlayer class.

We used the Listener pattern in the design of the client. The client keeps track of listeners that are notified of any state changes. These listeners are of a type created by a class that implements the ClientListener interface. For now, this interface is implemented by the ClientTUI class. Every object created by this class is a listener and is appended to the list of listeners in the class Client. We used the Listener pattern to make it easier to expand the program by adding another type of listener that wants to be notified of any state changes. This can be done by creating a class that implements the ClientListener interface and let the ClientTUI create a object of this type.

We used the Factory pattern in the design of players. We made an abstract class Player that has an abstract method determineMove(). Every class that extends Player implements this method. In our case, we made a HumanPlayer and a AIPlayer class that extend Player. They both override the method determineMove(). This pattern is used to make it easier to create objects of type Player in the rest of the program, without having to specify what exact type of player (Human or AI) it is.

2.2 Sequence diagrams

To show how the classes work together for certain executing flows, sequence diagrams are made.

In figure 3 the executing flow for a client sending and receiving a move is displayed. First the player inputs a move which is send to the server, via the ClientConnection. After the server has handled the move, it send a message back to the clients involved in the game. The move is received by the client and the client calls the method doMove() with this move on the game. Afterwards the ClientTUI displays the move by displaying the updated board. This is once again for the received move of the opponent. This process is repeated until the game is over or the connection is lost. In this case the server sends a message to the client with GAMEOVER together with a reason and a possible winner. The client removes the Game object and the ClientTUI displayed to the player that the game is over.

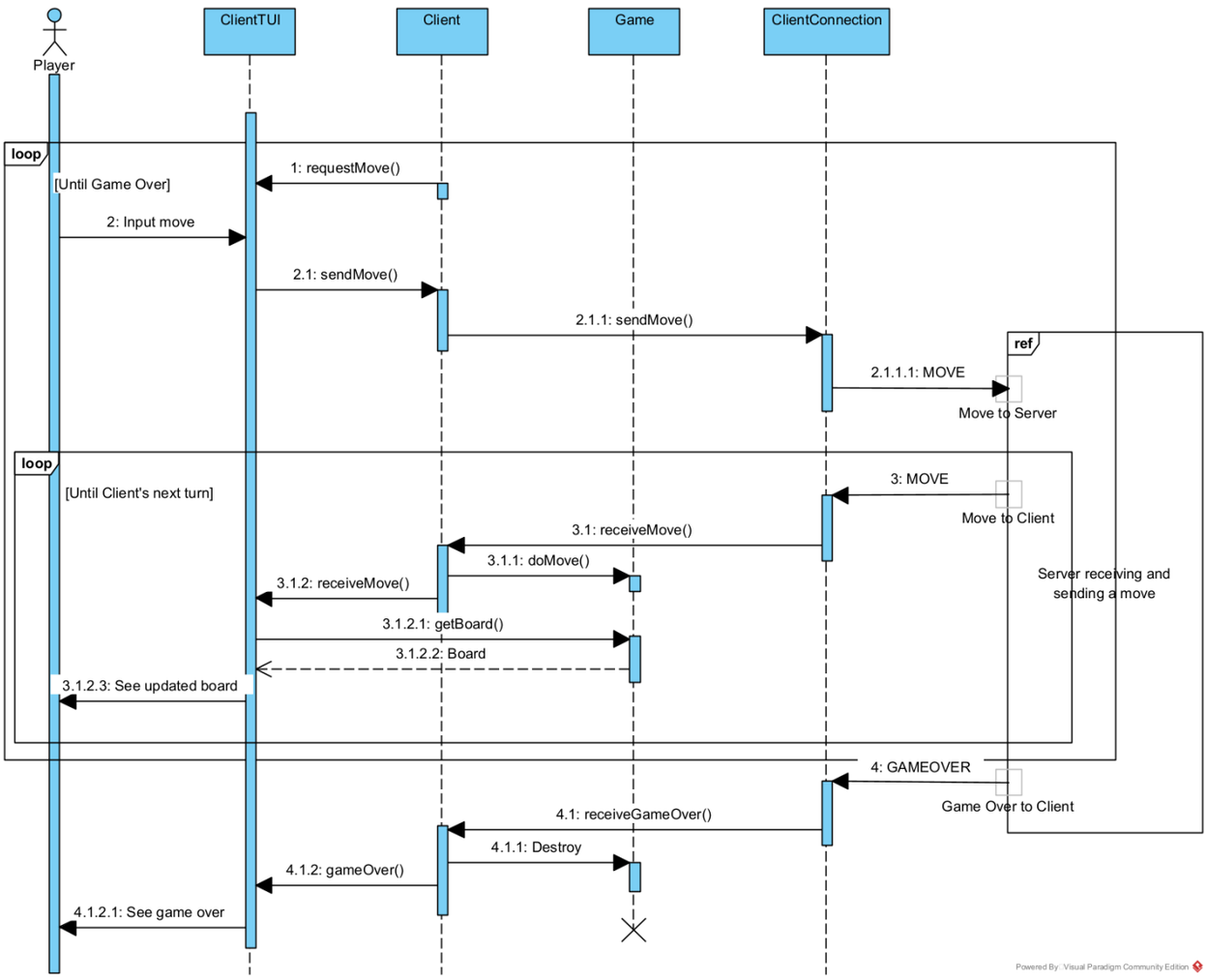


Figure 3: Sequence diagram of the client handling a move

In figure 4 the executing flow for a server sending and receiving a move is displayed. If the server receives a move from a client via the ServerConnection, it is send to the ServerClientHandler, which sends it to the ServerGameManager. The ServerGameManager performs the move on the game and sends a message to the clients indicating the move that was performed. This process is repeated until the ServerGameManager assessed the game as game-over. In

this case a game-over message is send to both clients that are connected to the game and the ServerGameManager's association with the game is deleted.

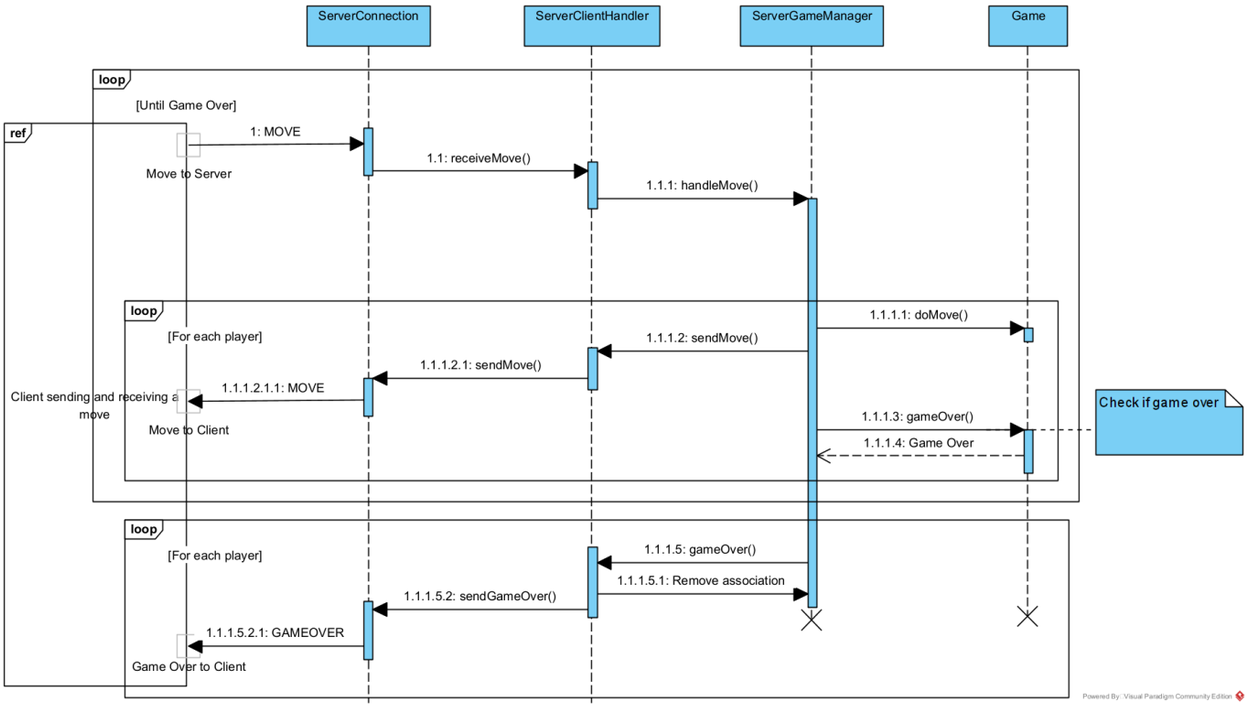


Figure 4: Sequence diagram of the server handling a move

In figure 5 the executing flow for a server establishing a connection is displayed. The server first gets a connection, for which it then creates a ServerClientHandler which creates a ServerConnection. It then exchanges HELLO messages with the client. Then the client sends a username, which the server checks for uniqueness and then responds with a login confirmation of error. After the client is logged in, it asks the server to be put in a queue. The server puts the client in a queue, and if there are now 2 clients in the queue it creates a ServerGameManager and sends a NEWGAME message to the two clients, thus beginning the game.

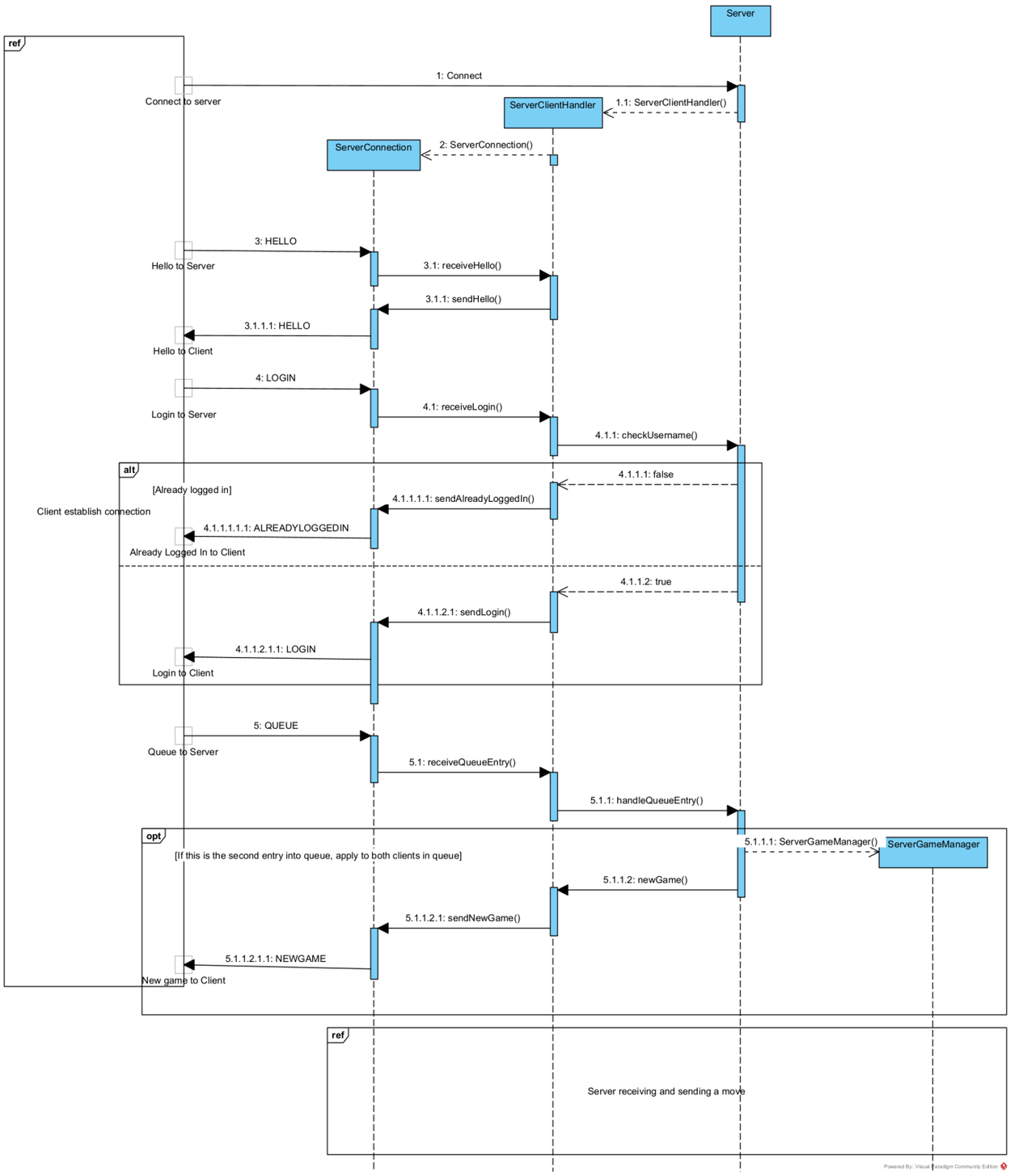


Figure 5: Sequence diagram of the server establishing a connection

In figure 6 the executing flow for a client establishing a connection is displayed. The ClientTUI asks the user for a host and port number to connect to. It then creates a Client object with this host and port number, which creates a ClientConnection object. If this is done correctly a hello message is send to the server with a client description. The server responds to this by sending a hello message back with a server description. Afterwards the ClientTUI asks the user for a username, which is send to the server. If this name is already taken, the client receives a message that this name is already known to the server. If the client is correctly logged in, it receives a confirmation from the server and it enters the queue. This request for entering the queue is send to the server. Once the client receives the message NEWGAME, it knows that it is put into a newly started game with another client.

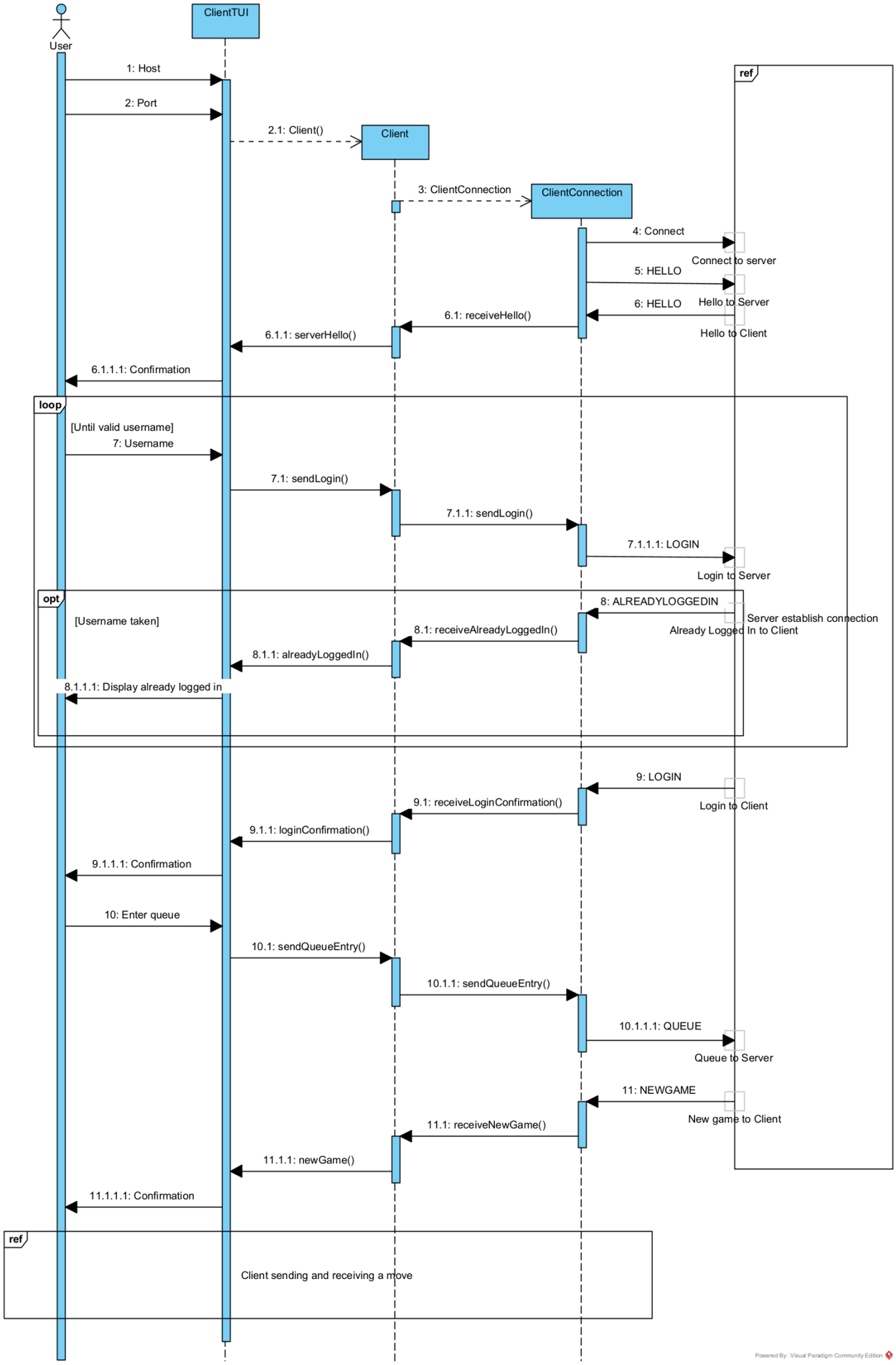


Figure 6: Sequence diagram of the client establishing a connection

2.3 State machine diagram

Figure 7 shows a state diagram of the TUIClient. It indicates the flow of the user interface. Each state in this diagram will be part of a enumerator, with a looping switch case calling the right method for the current state. These methods can then change the state to the next applicable state before returning.

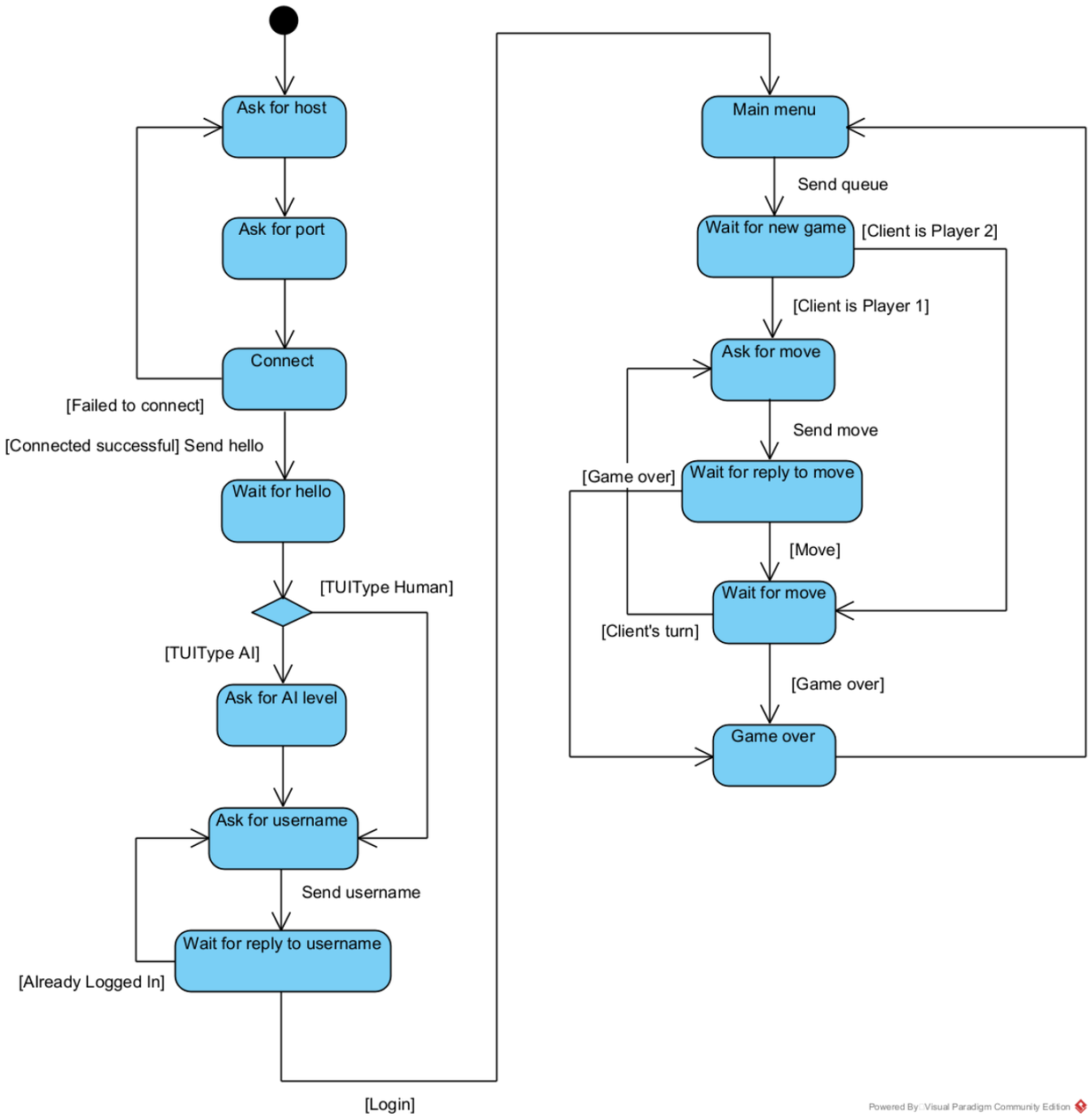


Figure 7: State diagram of the TUIClient

3 Concurrency mechanism

A couple of threads are used in the implementation of the network to play the Dots and Boxes game.

Server:
The initial thread from the server waits for and accepts new ServerConnections. When the server gets a connection, it calls the method handleConnection() and it creates a ServerClientHandler for this connection. The ServerClientHandler updates the list of clients, where concurrency could occur. Furthermore it creates and starts a ServerConnection. Each ServerConnection has its own thread, that waits for messages from its assigned client. These threads end when the connection with the client is closed. The initial server thread only ends when the server closes.

The list of clientHandlers can be updated by the main server thread when a client connects and by server connection threads when their client disconnects. Furthermore, it can be iterated over by the server connection threads. To prevent conflicts, these operations should be synchronized over this property.

Furthermore, receiveLogin() is called by server connection threads and iterates over the usernames of all clients and then updates its own username. To prevent the same username from being changed and read at the same time, this property should be synchronized for each of the ServerClientHandler instances.

The method receiveUserListRequest() is called by server connection threads and iterates over the usernames of all clients. To prevent one of the usernames being read and changed by another server connection thread at the same time, this property should by synchronized for each of the ServerClientHandler instances.

The queue of clientHandlers can be updated by server connection threads when their client sends a queue entry request or when their client disconnects. To prevent conflicts, these operations should be synchronized for each of the ServerClientHandler instances.

There is a potential for race conditions in the ServerGameManager, as this receives calls from two server connection threads on its handleMove() and handleDisconnect() methods. handleMove() only allows calls from the thread of

the player who currently has a turn and the turn is only handed over to the other player after the move has been processed. Howerver, we discovered a race condition where a client could send a new move before the Game Over state had been processed for the previous move. If this new move caused the game to be over, the thread processing the previous move and the thread processing the current move would both handle the game over logic, thus entering an invalid state. For this reason, both handleMove() and handleDisconnect() are synchronized. The handleDisconnect() causes a game over message to be send to both clients and then removes the reference to the ServerGameManager from both ServerClientHandlers. The thread of the other client synchronizes any threads that access its reference to the ServerGameManager and checks if its null to prevent a race condition where the reference is set to null because of a disconnect while the other thread is processing a move.

Client:
The client has two threads: the interface and the connection. The thread for the connection is created when connection with the server is established and ends when the connection closes. The ClientTUI uses a state system that ensures only the main TUI thread is able to make meaningful actions preventing a lot of concurrency issues. Messages coming in from the server are added to a queue of states and executed one at a time by the main TUI thread. When the main TUI thread has finished executing states, it parses user input and handles it before returning to look at the queue of states. Any changes to the queue of states is executed via the specific methods that are all synchronized to prevent the corruption of this queue by the main TUI thread and the server connection thread accessing it at the same time. The idle() method is synchronized to allow for the use of wait() so it can be woken up upon a message from the server.

The Client class does not have any special concurrency mitigation, as we could identify no potential issues that could occur without the client entering an illegal state regardless of concurrency.

4 Reflection on design

The initial design was already quite complete, compared to our realised design: The Server communicates with a client via a ServerClientHandler and a ServerConnection and for every game that is started a ServerGameManager is created which controls the game. The Client communicates with the server via a ClientConnection and communicates with the user via a ClientTUI, which is a ClientListener and wants to be updated of any state changes of the client. A client is a Player, which can Human or AI. The AI player determines moves using a Strategy, which can be naive or smart. Every Client that is represents a Human player, asks for moves via the ClientMoveInput interface, which is part of the ClientTUI.

One of the strengths of this initial design is that the model, i.e. the game itself, is independent of the other components of the system. The same goes for the user interface, which is independent of the controller, i.e. the client, and the model. These independencies are useful in the future when someone wants to change one component, without changing the whole system. Another strength is that strategies can easily be added, by implementing the Strategy interface. The same goes for ClientListeners. Letting the server create a game manager for every game that is started is also a strength of our design, since this separates the control of the game from the control of a connection with a client.

The weakness of the initial design is that the ClientTUI does not keep track of the order of states the Client can be in. This results in performing actions in possible different order than is described. For example, first the client should ask the user for a host name and port number, but the ClientTUI does not know this order and could possibly ask for a username first.

The whole initial design can be found in the appendix, section 8.

Compared to the initial design the main difference for the realised game logic classes are the helper methods and the methods concerning boxes that are added. When making the initial design we did not take into account that we need to print who captured what box. For this we needed to add a couple of methods in the Board class. Furthermore, we added a couple methods in the Board class that are used to determine the score of the players. A lot of helper methods are added to prevent duplication of code and to split long methods into smaller ones, to increase the readability.

The realised design of the server does not differ much from the initial design. The only difference in the structure in the class diagram is that we made the mistake to connect the Protocol class to the Server class instead of the ServerConnection class that uses the Protocol. The classes themselves changed a bit compared to the initial design, a couple of methods and helper methods were added. In the implementation we defined a couple of helper methods to prevent duplication of code and to make the methods more readable. We also caught and fixed a concurrency issue that was not expected in the initial evaluation of the concurrency.

We did not change much about the server because we could identify no major issues with the design and could not see a significantly better better way of implementing it. The server worked to our satisfaction with the initial design and with only a small amount of changes.

Compared to the initial design a couple of changes were made for the realised design of the client. The AI player got its own version of the TUI, a NonBlockingScanner class was created to be able to read lines from the input without blocking and an enumaration class is made for the states the TUIClient can be in. Furthermore an enumeration class is made for the types of players, which is nested in the TUI and also used here.

The enumeration class for the states of the TUIClient was described by the state machine diagram in the initial design and is used in the implementation of the ClientTUI. This system was not yet fully conceptualized during the design phase, and has undergone iteration during the implementation, including the addition of a queue system and a callback system. Some private attributes and a lot of private methods were added, these have been left out of the class diagram for readability.

If we were to do the project again, it would be better to split the ClientTUI into multiple classes, including a state manager, a server connection listener and the actual user interface. The current ClientTUI is too large, making it hard to maintain and read.

In the initial design and during most of the implementation, we had decided to include the human player and the AI in the same TUI. This was, contrary to the belief of the coordinator, not because we had not read the manual and also not a decision made hastily. Instead, this was because they would share a large amount of code, and it would leave open the possibility of changing player type without restarting the client. We have also had help multiple times from TAs with the design of this part of the class diagram, and none of them commented on the fact we were not seperating them into separate TUIs. We only changed to a separate main class for the AI after the announcement from the coordinator. We still stand by our initial decision.

5 System tests

To ensure that the entire program as a whole operates as expected, not only its components themselves, we construct system tests. This type of testing ensures that the software meets the technical and functional requirements.

Title Testing report for playing a game against an AI player.
Scope Test the ability of the server, client, and the AI to handle a game of a human player against an AI player.
Expected Behaviour It is expected that the the server correctly handles constructing a game with an AI player and human player and that their moves are correctly handled by the server and both clients.
Test steps

1. Start a server with a valid port number, a human client and a AI client with corresponding host name and port number.
2. On the human client, enter a valid username and queue for a game. Observe the behaviour of the server and this client.
3. On the AI client, choose as difficulty level smart. Enter the same username as the first client. Observe the response from this client and the server.
4. On the AI client, ask for the list of users. Observe the response from the this client and the server.
5. On the AI client, enter a valid username and ask for the list of users. Observe the behaviour of the server and this client.
6. On the AI client, queue for a game. Observe the behaviour of the server, this client and the other client.
7. On the human client, ask for a hint. Observe the response from the server and this client.
8. On the human client, ask for the list of users. Observe the response from the server and this client.
9. On the human client, enter a valid move. Observe the behaviour from this client, the other client and the server.
10. On the human client, keep entering valid moves until the game is over. Observer the behaviour from this client, the other client and the server.

Results

1. The server is started with port number 4567 and the two clients are connected. Both clients have send hello messages with their client description, and the server responded to both with a hello message with the server description.

```
Input port number
4567
waiting for clients to connect
client is connected
SocketConnection.sendMessage: HELLO~server Ylona
client is connected
SocketConnection.sendMessage: HELLO~server Ylona
```

Figure 8: server response step 1

```
Dots and Boxes game
At any point, type help to see this message and exit to exit the game.
Once you're logged in, at any point type list to see a list of players.
The aim of the game is to get as many points. Complete the 4th side of a box to get a point.
When asked for a move, type a number to put a line at that location, or type hint for a hint.
If this line completes a box, you get a point and another turn! Otherwise, it's the opponents turn.
The game ends when the board is full. The player with the most points wins.

Host? (Reference for 130.89.253.64, empty for localhost):
Checking host...
Port? (empty for 4567):
Connecting...
SocketConnection.sendMessage: HELLO~Minor-2's Client
ClientConnection.handleMessage: HELLO~server Ylona
Server description: server Ylona
Username?
```

Figure 9: human client response step 1

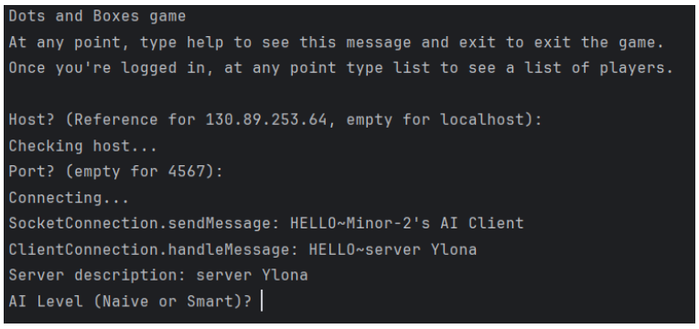


Figure 10: AI client response step 1

2. The human client has send a login request to the server with the entered username. The server responded with login confirmation and a welcome message is showed to the user. The human client send a queue entry request to the server. The server put the human client into the queue. The human client is now waiting for a game to start.

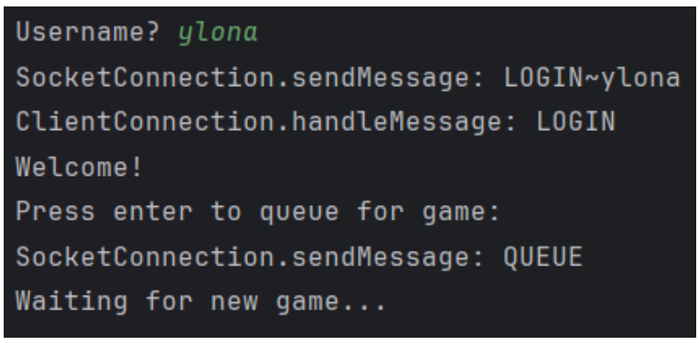


Figure 11: Human client response step 2

3. The AI client has send the login request with the entered username to the server. The server responded with an already logged in message. The AI client shows the user that the entered username is already taken and asks the user to enter an username.

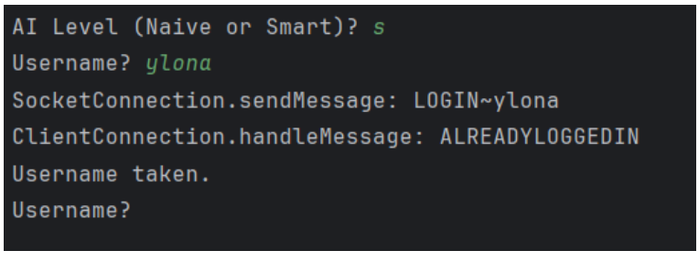


Figure 12: AI client response step 3

4. The AI client does not send the list request to the server, since the client is not allowed to ask for the list of users when it is not logged in. Instead the AI client shows to the user the message that the it is not logged in.

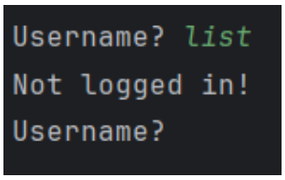


Figure 13: AI client response step 4

5. The AI client has send a login request to the server with the entered username. The server responded with a login confirmation and a welcome message is showed to the user. The user is asked to queue for a game. The AI client send the list request to the server, which responded with a list message which consist of all the user that are connected to the server. The AI client shows these users to the user.

```
Username? smart ai
SocketConnection.sendMessage: LOGIN~smart ai
ClientConnection.handleMessage: LOGIN
Welcome!
Press enter to queue for game: list
Waiting for user list...
SocketConnection.sendMessage: LIST
ClientConnection.handleMessage: LIST~ylona~smart ai
Connected: (2)
    ylona
    smart ai

Press enter to queue for game:
```

Figure 14: AI client response step 5

6. The AI client has send a queue entry request to the server, which put this client into the queue. Furthermore, the server has started a new game between the two connected clients and send a new game message to both of them. Both clients show the board together with the current scores and current player to their user. The first client is asked to enter a move.

```
smart ai is put in queue
SocketConnection.sendMessage: NEWGAME~ylona~smart ai
SocketConnection.sendMessage: NEWGAME~ylona~smart ai
A new game has started between ylona and smart ai
```

Figure 15: Server response step 6

```
ylona vs smart ai
.    .    .    .    .    .       . 0   . 1   . 2   . 3   . 4   .
                                   5     6     7     8     9     10
.    .    .    .    .    .       . 11  . 12  . 13  . 14  . 15  .
                                   16    17    18    19    20    21
.    .    .    .    .    .       . 22  . 23  . 24  . 25  . 26  .
                                   27    28    29    30    31    32
.    .    .    .    .    .       . 33  . 34  . 35  . 36  . 37  .
                                   38    39    40    41    42    43
.    .    .    .    .    .       . 44  . 45  . 46  . 47  . 48  .
                                   49    50    51    52    53    54
.    .    .    .    .    .       . 55  . 56  . 57  . 58  . 59  .

Player, playing with number 1, ylona score = 0
Player, playing with number 2, smart ai score = 0
Current player is ylona
Line? (hint for hint):
```

Figure 16: Human client response step 6

7. The human client responded to the hint request by showing a valid move to the user. The human client asks the user to enter a move.

```
Line? (hint for hint): hint
Hint: line 27
Line? (hint for hint):
```

Figure 17: First client response step 7

8. The human client has send a list request to the server, which responded with a list message with all the users connected to the server. The human client shows these users to the user and asks the user to enter a move.

```
Line? (hint for hint): list
Waiting for user list...
SocketConnection.sendMessage: LIST
ClientConnection.handleMessage: LIST~ylona~smart ai
Connected: (2)
    ylona
    smart ai

Line? (hint for hint):
```

Figure 18: First client response step 8

9. The human client has send a the move to the server, which has send a move confirmation to both clients. The AI client is asked to send a move back to the server. This client determines a move and sends a move to the server. The server receives this move and send a move confirmation to both clients. The human client is asked to determine a move. The changed board together with the current scores and the current player is showed to both users. The human client asks the user to enter a move.
10. The server keeps handling the moves the clients send, until the game is over. In this case, the server sends a game over message to both clients with the reason of game over and the winner of the game. The clients show to the user if the user won, lost or if the game ended in a draw and asks the user to enter the queue.

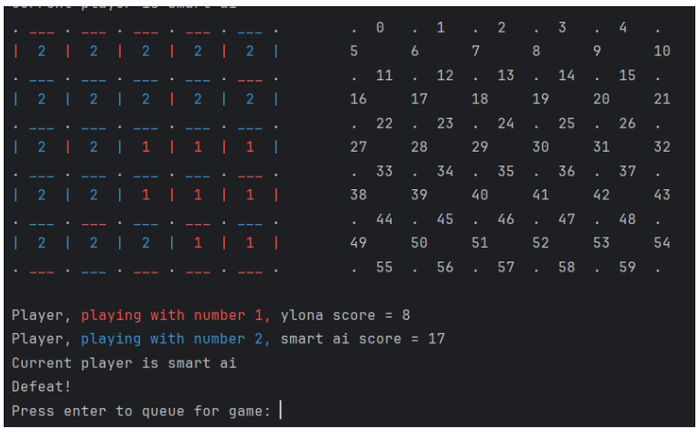


Figure 19: Client for human player response step 10

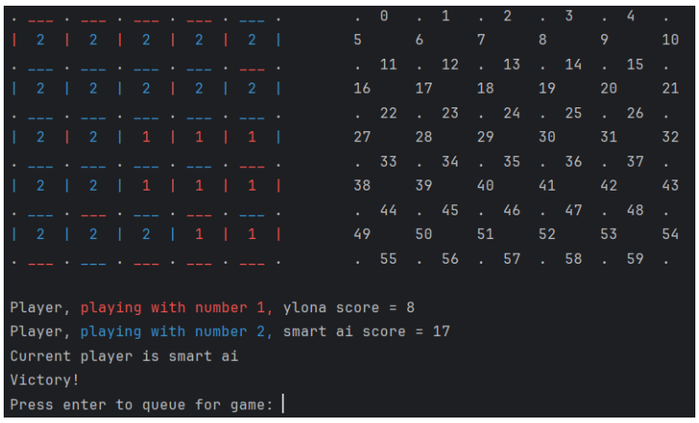


Figure 20: Client for AI player response step 10

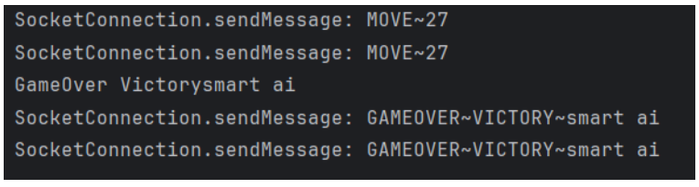


Figure 21: Server response step 10

Title Testing report for the server and client handling a disconnection
Scope Test the ability of the server and client to handle sudden disconnection from a client during a game and a disconnection of the server. Furthermore test the server to handle malformed or unexpected input from the user.
Expected Behaviour It is expected that the server gracefully terminates the game when a client disconnect and that it informs the other client according to the protocol. It is expected that the client gracefully terminates when connection to the server is lost. It is expected that the server does not crash on invalid input from the user, but asks the user for the input again.
Test steps

1. Run the server and input a invalid port number. Observe the response from the server.
2. On the server, enter a valid port number. Observe the response from the server.
3. Run a human client and let it connect to the server by entering localhost and the corresponding port number of the server. Enter a valid username and queue for a game.
4. Run another human client and let it connect to the server by entering localhost and the corresponding port number of the server. Enter a valid username and queue for a game.
5. Input a valid move for the first client.
6. Input a valid move for the second client.

7. Terminate client1 by entering exit
8. Observe the response from the server to the disconnection of client1.
9. Observe the response from client2 to the disconnection of client1.
10. Terminate the server. Observe the response from client2 to the disconnection of the server.

Results

1. The server marks the entered port number as invalid and asks the user again to enter a port number.

```
Input port number: hello
Invalid port.
Input port number: -1
Invalid port.
Input port number:
```

Figure 22: server response step 1

2. The server is constructed with port number 44000 and waits for clients to connect.

```
Input port number
44000
waiting for clients to connect
```

Figure 23: server response step 2

3. The server connects client1, sends a hello message with it's server description, sends a login message to client1 and puts client1 in the queue.

```
client is connected
SocketConnection.sendMessage: HELLO~server Ylona
SocketConnection.sendMessage: LOGIN
client1 is put in queue
```

Figure 24: server response step 3

4. The server connects client2, sends a hello message with it's server description, sends a login message to client2 and puts client2 in the queue. Furthermore it start a game between client1 and client2 and sends the new game message to these clients.

```
client is connected
SocketConnection.sendMessage: HELLO~server Ylona
SocketConnection.sendMessage: LOGIN
client2 is put in queue
SocketConnection.sendMessage: NEWGAME~client1~client2
SocketConnection.sendMessage: NEWGAME~client1~client2
A new game has started between client1 and client2
```

Figure 25: server response step 4

5. The server performs the move received by client1 and send the move message to both playing clients.

```
SocketConnection.sendMessage: MOVE~40
SocketConnection.sendMessage: MOVE~40
```

Figure 26: server response step 5

6. The server performs the move received by client2 and send the move message to both playing clients.

```
SocketConnection.sendMessage: MOVE~58
SocketConnection.sendMessage: MOVE~58
```

Figure 27: server response step 6

7. The server sends the game over message with reason disconnect and winner client2 to both clients, removes client1 from the list of users and removes the connection of client2 with the game.

```
client1 is disconnected
SocketConnection.sendMessage: GAMEOVER~DISCONNECT~client2
SocketConnection.sendMessage: GAMEOVER~DISCONNECT~client2
```

Figure 28: server response step 7

8. Client2 receives the message game over from the server and gets asked to queue for a (new) game.

```
Opponent Disconnected.
Press enter to queue for game:
```

Figure 29: Client2 response step 8

9. Client2 receive the message that the connection to the server is lost and it terminates.

```
Opponent Disconnected.
Press enter to queue for game: Connection lost.

Process finished with exit code 0
```

Figure 30: Client2 response step 9

Title Testing report for the client handling invalid user input
Scope Test the ability of the client to handle malformed or unexpected input from the user.
Expected Behaviour It is expected that the client does not crash on invalid input, but asks the user for input again.
Test steps

1. Run the server and input a valid port number.
2. Run a AI client. Enter an invalid host name and observe the behaviour of this client.
3. Enter a to the server corresponding host name and an invalid port number. Observe the behaviour of this client.
4. Enter a to the server corresponding port number and an invalid difficulty level. Observe the behaviour of this client.
5. Enter a valid difficulty level and enter an invalid username. Observe the behaviour of this client.
6. Enter a valid username and queue for a game.
7. Run a human client and connect it with the server by giving it a corresponding host name and port number. Enter an invalid username and observe the behaviour of this client.
8. On the human client enter a valid username and queue for a game. Wait for the other player to play a move.
9. On the human client enter an invalid move. Observe the behaviour of this client.

Results

1. The server is constructed with port number 44000 and it waits for clients to connect.

```
Input port number
44000
waiting for clients to connect
```

Figure 31: server response step 1

2. The AI client shows the start menu to the user and asks for a host name to connect to. It checks the entered host name, lets the user know that the entered host name is invalid and it asks the user again to enter a host name.

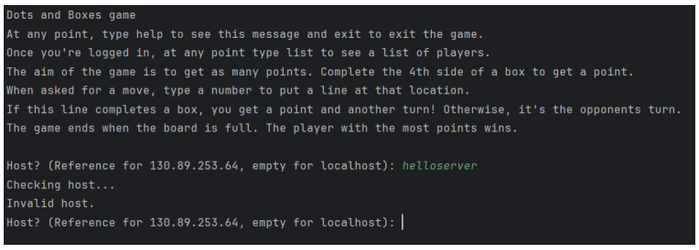


Figure 32: AI client response step 2

3. The AI client has marked the host name as valid and asks for a port number. It checks the entered port number, lets the user know that the entered port number is invalid and again asks the user to enter a port number.

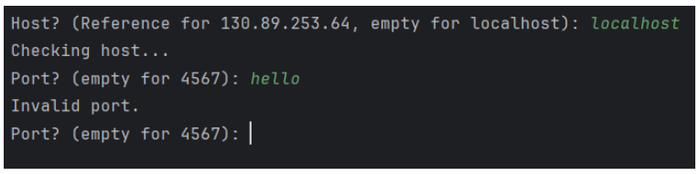


Figure 33: AI client response step 3

4. The AI client is connected to the server and send a hello message with the client description. It received a hello message from the server and and asks the user for the difficulty level of the AI player. It checks the entered difficulty level, lets the user know that the entered level is invalid and again asks the user to choose a difficulty level.

Figure 34: AI client response step 4

5. The AI client has marked the entered difficulty level as valid and asks the user for an username. It checks the entered username, lets the user know that it is invalid and again asks the user to enter an username.

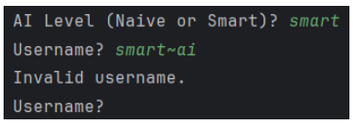


Figure 35: AI client response step 5

6. The AI client has marked the entered username as valid and asks the user to queue for a game. It sends the queue request to the server.

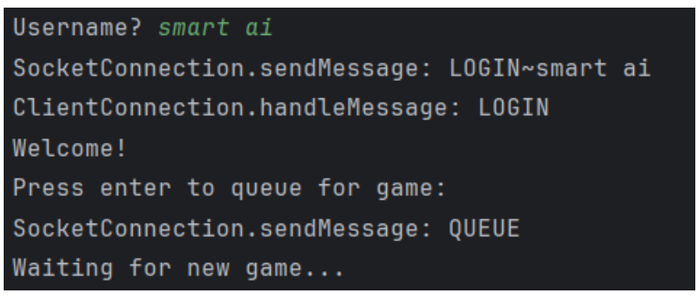


Figure 36: AI client response step 6

7. The human client is connected to the server, has send a hello message to the server with the client description, has received a hello message from the server, asks the user to enter an username and has marked the entered username as invalid. It lets the user know that it is invalid and again asks the user to enter an username.

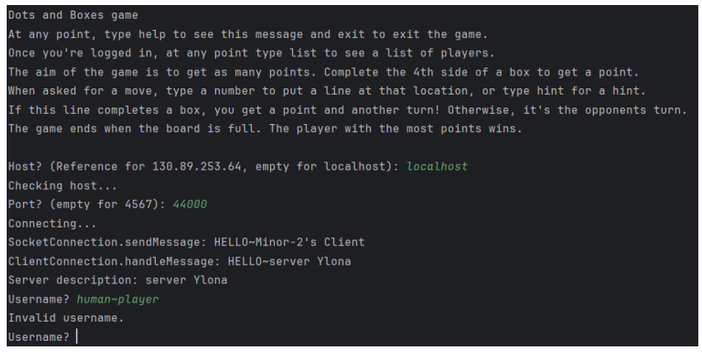
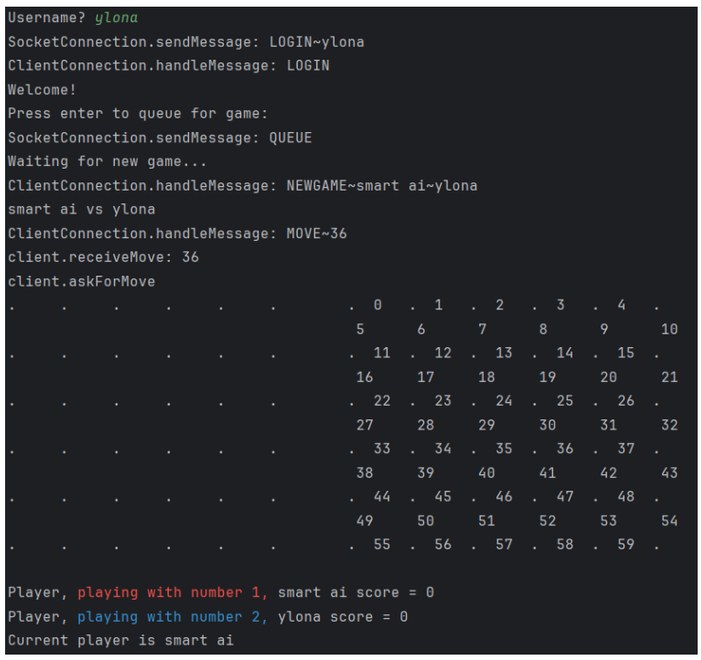


Figure 37: Human client response step 7

8. The human client has marked the entered username as valid and has received a login confirmation from the server. The human client asks the user to queue for a game. The server has created a new game with the two clients and lets the AI client know to determine a move. The AI player sends a move to the server and the server sends a confirmation of the move to both clients. The human client asks the user to enter a move.



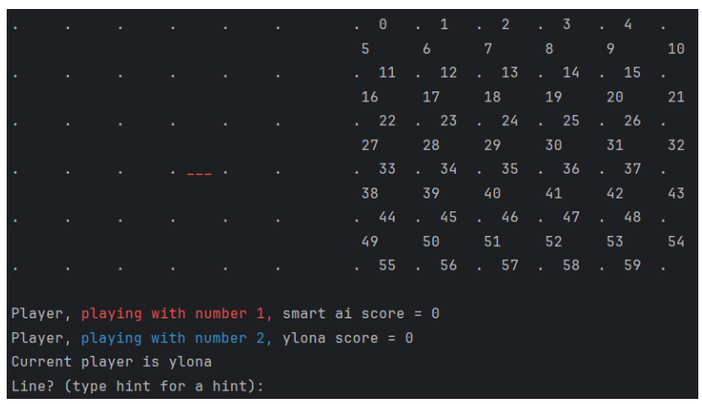


Figure 38: Human client response step 9

9. The human client has marked the move as invalid and lets the user know that the entered move is not valid. It again asks the user to enter a move.

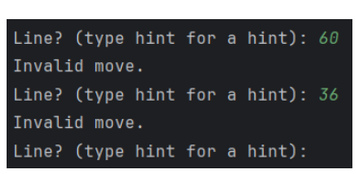


Figure 39: Human client response step 10

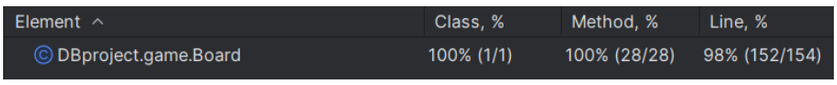
6 Testing Strategy

The aim of the overall test strategy is to ensure that our program follows the game rules and the program correctly handles any input.

For each class used in the implementation of the game logic, unit tests are made, to test every method of these classes. These test should have a line coverage of at least 75%, but preferably at least 90%. Furthermore the tests should cover 100% of the class and the methods of this class. The unit testing strategy is chosen to test the game logic, to ensure that every method of these classes functions as it is intended. This is needed to ensure correct usage of these classes in the further implementation of the Dots and Boxes game. This implementation is tested with integration tests.

The aim of testing is to ensure that a game is implemented according to the game rules. The game logic is tested against the example server, which will give an error in case of any mistake.

Board Test



Element	Class, %	Method, %	Line, %
DBproject.game.Board	100% (1/1)	100% (28/28)	98% (152/154)

Figure 40: Coverage of Board test

The main tests of the board are as follows:

- to test that a board is correctly constructed
- to test that lines are placed correctly
- to test that completed boxes are detected and marked correctly
- to test that a board is full

Next to these main tests helper functions are tested. The line coverage of the constructed board test is 98% and the function coverage is 100%, see figure 40.

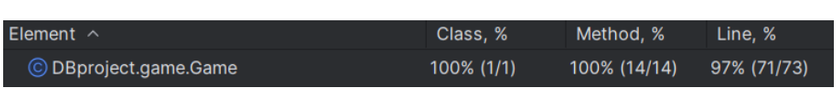
Two main tests of the board will be explained in detail.

First we look at the test for placing a line on the board. The corresponding methods in Board that are tested are called setLine and getLine. The method getLine(int location) returns the content of the line at location if this location is a representation of a line on the board. Functional coverage is satisfied for this method by calling this method in the board test. Statement coverage is satisfied, since this method was called with a location for which isLine(location) equals true, for example getLine(0); Branch coverage is satisfied, since it was called with a location for which isLine(location) equals false, for example getLine(-1); Condition coverage is also satisfied by the executions for statement coverage and branch coverage. The test detects possible bugs by assuring that an Illegal Argument exception is thrown when the input is not of the correct type and otherwise assuring that the returned line matches the requested line.

The method setLine(int location, int playerNumber) sets a line at location, if this location is a representation of a line on the board, to playerNumber. Functional coverage is satisfied for this method by calling this method in the board test. Statement coverage is satisfied, since this method was called with a location for which isLine(location) equals true, for example setLine(0,1) Branch coverage is satisfied, since it was called with a location for which isLine(location) equals false, for example setLine(-1,1) Condition coverage is also satisfied by the executions for statement coverage and branch coverage. The test detects possible bugs by assuring that an Illegal Argument exception is thrown when the input is not of the correct type and otherwise assuring that the line at location matches the playerNumber after calling the method.

Next we look at the test for detecting a completed box. The corresponding method in Board that is tested is called completeBox and takes as input a line that possibly completes a box. It returns an array of the indices of the boxes the line completes, or -1 if the line does not complete box. Functional coverage is satisfied for this method by calling this method in the board test. Statement coverage is satisfied, since this method was called for lines which are horizontal and complete a box above, horizontal and complete a box under, horizontal and both complete a box under and above, horizontal and do not complete a box, vertical and complete a box left, vertical and complete a box right, vertical and both complete a box left and right and lastly for lines which are vertical and do not complete a box. Branch coverage is satisfied by also calling this method with a location for which isLine(location) equals false. Condition coverage is also satisfied by the executions for statement coverage and branch coverage. The test detects possible bugs by assuring that an Illegal Argument exception is thrown when the input does not represent a line on the board and otherwise assuring that the method returns the correct boxes the line possibly completes.

Game Test



Element	Class, %	Method, %	Line, %
DBproject.game.Game	100% (1/1)	100% (14/14)	97% (71/73)

Figure 41: Coverage of Game test

The main tests of the game are as follows:

- to test that a game is correctly constructed

- to test that a game is correctly marked as game-over
- to test that the winner is determined correctly
- to test that a move is performed correctly

Next to these main tests helper functions are tested. The line coverage of the constructed board test is 97% and 100% of the class and methods are tested, see figure 41.

For this test we also look in detail to a specific test.

To test that a move is performed correctly, the method doMove in the game class is tested. The method doMove(int location) performs the move, if the input represents a valid move. Functional coverage is satisfied for this method by calling this method in the game test. Statement coverage is satisfied, since this method was called for lines which represent a valid move, for lines which complete a box and for lines which complete two boxes. Furthermore the method was called for each of these lines, when the current player is player1 and when the current player is player 2. Branch coverage is satisfied, since by calling the method with a line which does not represent a valid move, for example doMove(0) and after that again calling doMove(0). Condition coverage is satisfied by the executions for statement coverage ad branch coverage. The test detects possible bugs by assuring that nothing is done when the input does not represent a valid move and otherwise assuring that the line at the input of the method doMove is set on the board and scores and players are updated correctly.

As mentioned in the previous section, system tests are constructed to ensure that the entire program as a whole operates as expected. The unit tests complement these system test, since they ensure that every class itself functions as expected, so that the system test can ensure that these classes collaborate correctly. Three system tests were made to test the main requirements of the system. These are: playing a game with different types of clients, disconnecting and ensuring that the program does not crash handling unexpected input. As mentioned in the previous section all these test are passed by our program. The limitations of system tests are that we have created them ourselves, which means that we could have overlooked some edge cases that should have been tested.

7 Reflection on progress

The tasks during the project were divided as follows:

Together:

- making initial design
- updating the class diagram for realised design
- looking at the prevention of a couple of race conditions
- writing the realised design section
- writing the concurrency section
- writing the reflection on design section
- writing the reflection on progress section

Marinus:

- implementing and documenting (javadoc + comments) networking classes
- implementing and documenting (javadoc + comments) the client classes
- updating the sequence diagrams for realised design
- exporting the program

Ylona:

- implementing and documenting (javadoc + comments + JML) the game logic classes
- implementing and documenting (javadoc + comments) the player classes
- implementing and documenting (javadoc + comments) the server classes
- structuring the report
- writing the system tests
- testing the game logic classes
- writing overall test strategy section

Personal reflections:
Marinus:
Working on this project was nice, although external factors ended up making it quite stressful for me at times. We started out making the design together, which helped us both understand the program and knew what we were going to be making. The design was quite extensive, which made it easy to sit down and start implementing.

My main challenge was the TUI, which I tried to implement using a state system. This ended up only working out partly, and the class ended up very large. We also assumed that the AI TUI could be combined with the Human TUI, which we ended up having to change last minute.

Once the design was finished we mostly split the work and worked separately, but we still tried to meet up everyday. Unfortunately I had to be away from 3 days due to personal circumstances, leaving Ylona to do a lot of work on the report by herself with me doing a small amount of work whenever possible. In the end this meant that we

couldn't implement any of the extensions and we decided to rush or exclude a few things that we knew would cost us some points.

Overall I'm pretty happy with the result, although I wish we could have implemented some of the extensions.

Ylona:
We designed the program together. The reason for this is that we wanted to ensure that both of us understood what it is we wanted to implement. Discussing the design together made it easier to come up with new ideas and to argument why we made certain choices. A consequence of these collaborative design sessions is that we both knew exactly what the implementation of each class was going to look like, which made it easier to familiarize yourself with the code the other person wrote, this is at least how I experienced it.

During the project we hit a couple of roadblocks time wise. We were not able to work together for three days, which made it a bit more difficult to keep up to date with each other's work and to complete the project in time. For the rest of the days we met on campus to work together, which made it easier to asks questions to each other and to discuss some ideas. We divided the tasks, but we informed each other daily about our progress and once we finished a specific task. We kept to our initial planning, which was manageable, and we were able to finish the project in time despite the hiccups. However, due to the hiccups we did not manage to implement any extensions and we had to rush a few things.

Another roadblock we hit is that up until the last day, we did not know that the AI should be a separate client with its own TUI. This Tuesday we read that this should be the case and Marinus made the last minute solution to give the AI its own TUI, but this caused me a lot of stress to finish the project in time.

One shortcoming of our planning would be that there was not a lot of time to do the last checks of the code and to package it correctly. Next time I would improve this by reserving more time in the planning to hand everything in, instead of looking at this the last day.

We wrote this report together using Overleaf to share documentation. We shared our code together using GIT. This made it easier to work together and to share our work with each other.

8 Appendix

8.1 Initial design

8.1.1 Classes and Packages

The different classes with a summary of its main responsibility:

1. Game: implements the rules of the Dots and Boxes game, used by both client and server.
2. Board: represents the current state of the board used in the Dots and Boxes game.
3. Player (abstract): used as generic input for moves.
4. HumanPlayer: ask for a move via the console.
5. AIPlayer: give moves via a computer Strategy.
6. Strategy (interface): interface for computer Strategies.
7. NaiveStrategy: a very simple Strategy.
8. SmartStrategy: a more competent Strategy.
9. Protocol: contains constants for communicating between client and server.
10. SocketConnection (abstract): contains the basis for communicating via a socket.
11. ClientConnection: uses the Protocol to communicate with a server using the framework of SocketConnection.
12. Client: manages communication between a server and local interfaces. Furthermore, it keeps track of the state of the game.
13. ClientListener (interface): interface for local classes to receive updates about the state of the game.
14. ClientTUI: text-based user interface for the client. Also responsible for instantiating the client. Uses Player to get input for moves. Get updates from the Client via the ClientListener interface.
15. ClientMoveInput (interface): interfaces that can respond to a move input request.
16. SocketServer (abstract): contains the basis for creating a server for clients to connect to via a socket.
17. Server: handles incoming connections from clients and contains a queue for matching clients and starting a game. Creates a ServerGameManager when two clients are in the queue.
18. ServerGameManager: manages a game between two clients.
19. ServerClientHandler: manages the communication between a client and the server. Communicates with the ServerGameManager if a game is currently running.
20. ServerConnection: uses the Protocol to communicate with a client using the framework of SocketConnection.

The classes are connected as described in the class diagrams in figure 42, figure 43 and figure 44.

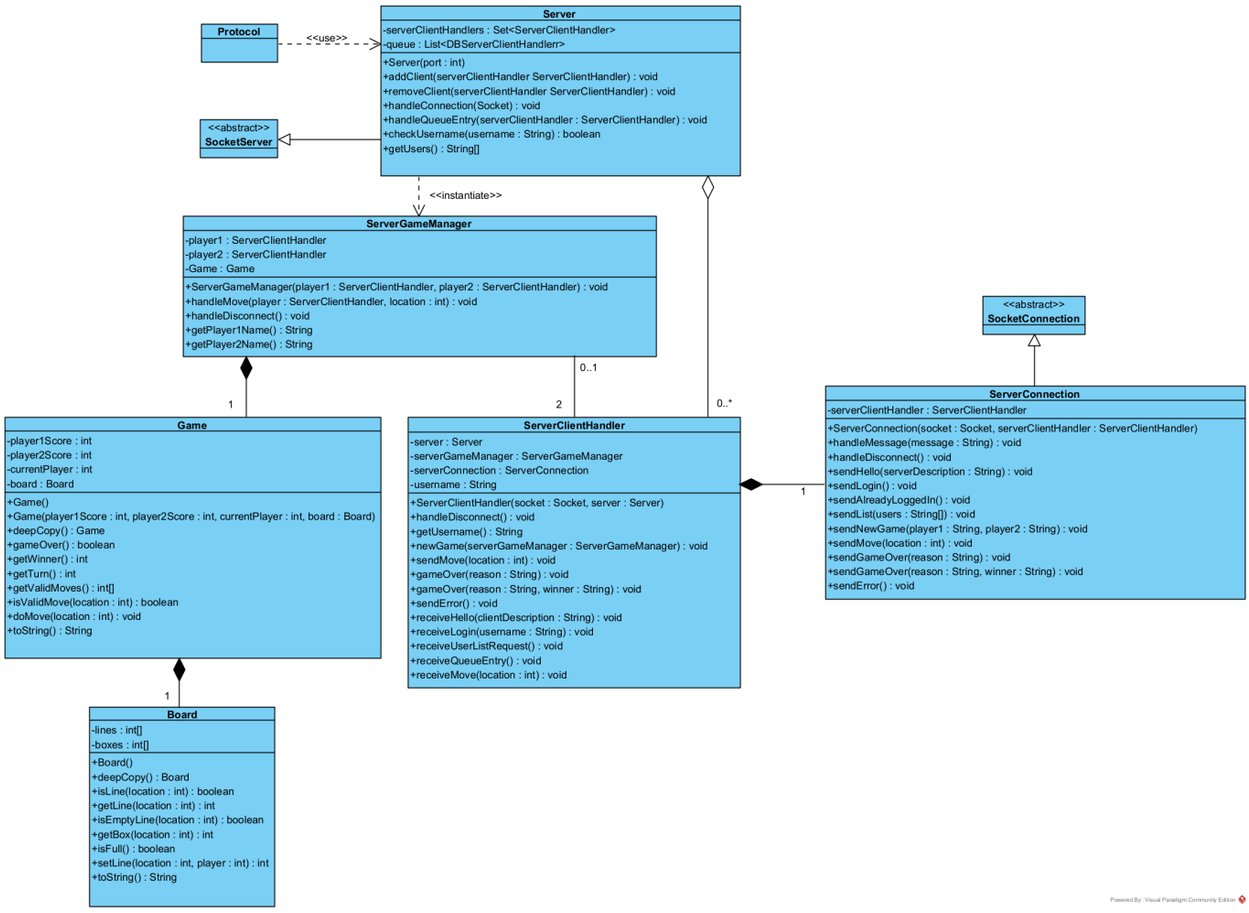


Figure 42: Class diagram of the server

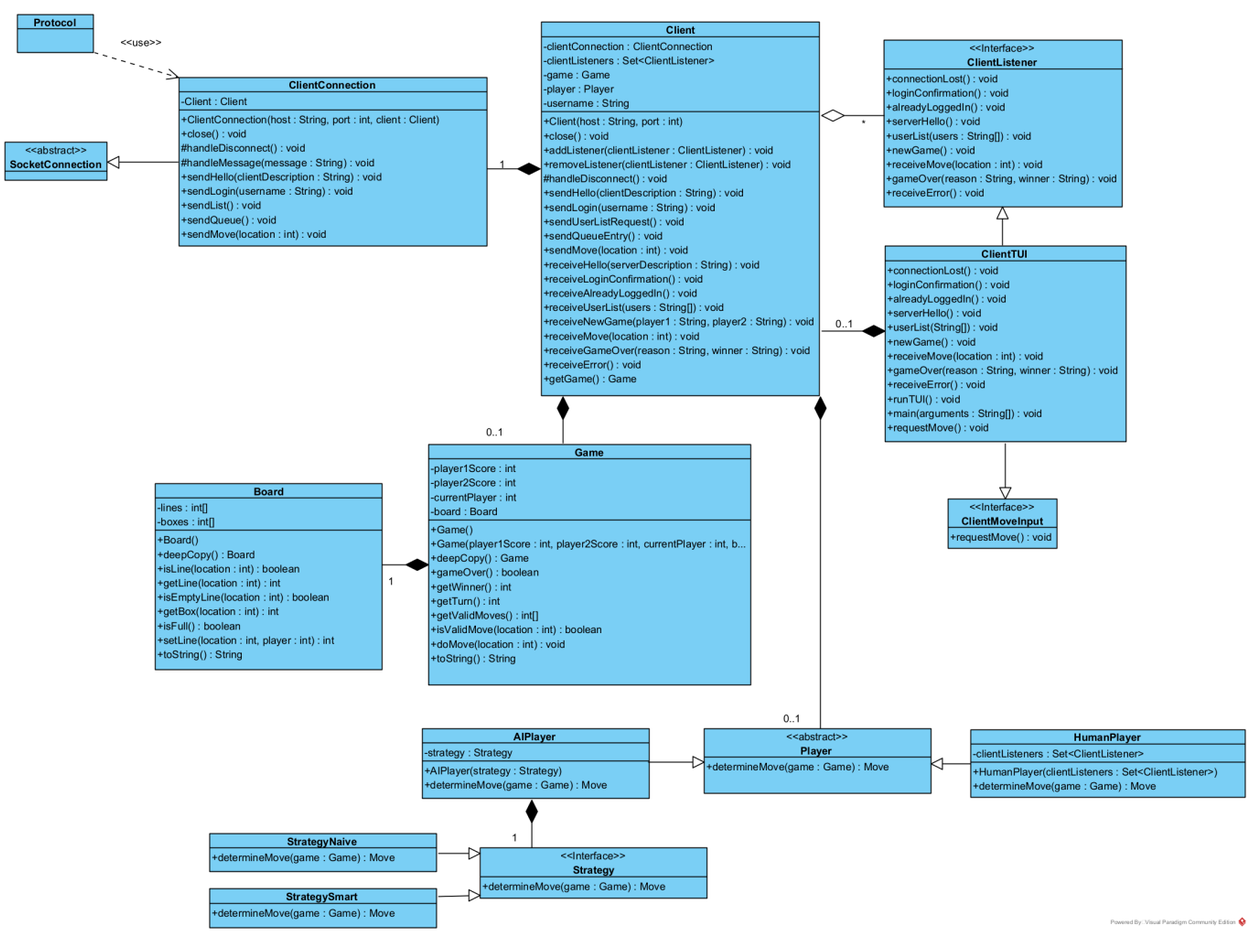


Figure 43: class diagram of the client

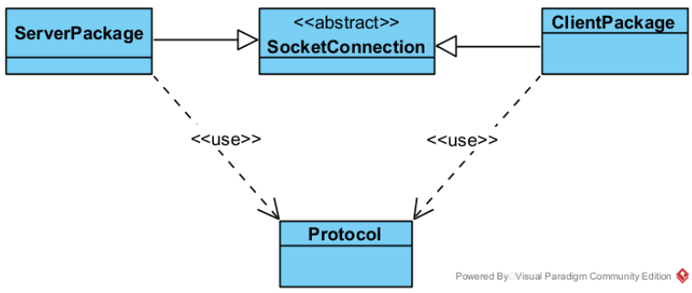


Figure 44: class diagram of the connection between server and client

The classes are structured in packages as follows:
Game:

- Game
- Board

Players:

- Player (abstract)
- HumanPlayer
- AIPlayer
- Strategy (interface)
- NaiveStrategy
- SmartStrategy

Server:

- SocketServer (abstract)
- Server
- ServerGameManager
- ServerClientHandler
- ServerConnection

Client:

- ClientConnection
- Client
- ClientListener (interface)
- ClientTUI
- ClientMoveInput

Networking:

- Protocol
- SocketConnection (abstract)

The following design pattern are used:

- MVC
- Strategy
- Listener
- the creational pattern: Factory
- State

The MVC pattern is used in the design of the client. The class ClientTUI represents the view, the Client class represents the controller of the model, that is represented by the Game and Board class.

We used the Strategy pattern in the design of the computer player. The computer player determines moves by using a specific strategy. For now, we have defined two strategies: NaiveStrategy and SmartStrategy. Both these classes implement the interface strategy. To make the computer player smarter, we can create another strategy that implements the interface strategy. This way you can easily make the computer player use another strategy, without having to chance the code of the AIPlayer class.

We used the Listener pattern in the design of the client. The client keeps track of listeners that are notified of any state changes. These listeners are of a type created by a class that implements the ClientListener interface. For now, this interface is implemented by the ClientTUI class. Every object created by this class is a listener and is appended to the list of listeners in the class Client. We used the Listener pattern to make it easier to expand the program by adding another type of listener that wants to be notified of any state changes. This can be done by creating a class that implements the ClientListener interface and let the ClientTUI create a object of this type.

We used the Factory pattern in the design of players. We made an abstract class Player that has an abstract method determineMove(). Every class that extends Player implements this method. In our case, we made a HumanPlayer and a AIPlayer class that extend Player. They both override the method determineMove(). This pattern is used to make it easier to create objects of type Player in the rest of the program, without having to specify what exact type of player (Human or AI) it is.

The State pattern is applied in the ClientTUI class. Depending on the internal state of the client methods are called. If the client, for example, is in the state to play a game, the method newGame() is called that sends a request to the server to start a game. The server puts the client in the queue. In the same way methods are defined for each state the client can be in.

8.1.2 Sequence Diagrams

To show how the classes work together for certain executing flows, sequence diagrams are made.

In figure 45 the executing flow for a client sending and receiving a move is displayed. First the player inputs a move which is send to the server, via the ClientConnection. After the server has handled the move, it send a message back to the client. If the move was marked valid by the server, the message MOVE ¡N¿ is received by the client and the client calls the method doMove() with this move on the game. Afterwards the ClientTUI displays the move by displaying the updated board. This is once again for the received move of the opponent. If the move was marked invalid by the server, the server returns an error message that is displayed to the player via the ClientTUI and the client ask for another input of the player. This process is repeated until the game is over or the connection is lost. In this case the server sends a message to the client with GAMEOVER together with a reason and a possible winner. The client removes the Game object and the ClientTUI displayed to the player that the game is over.

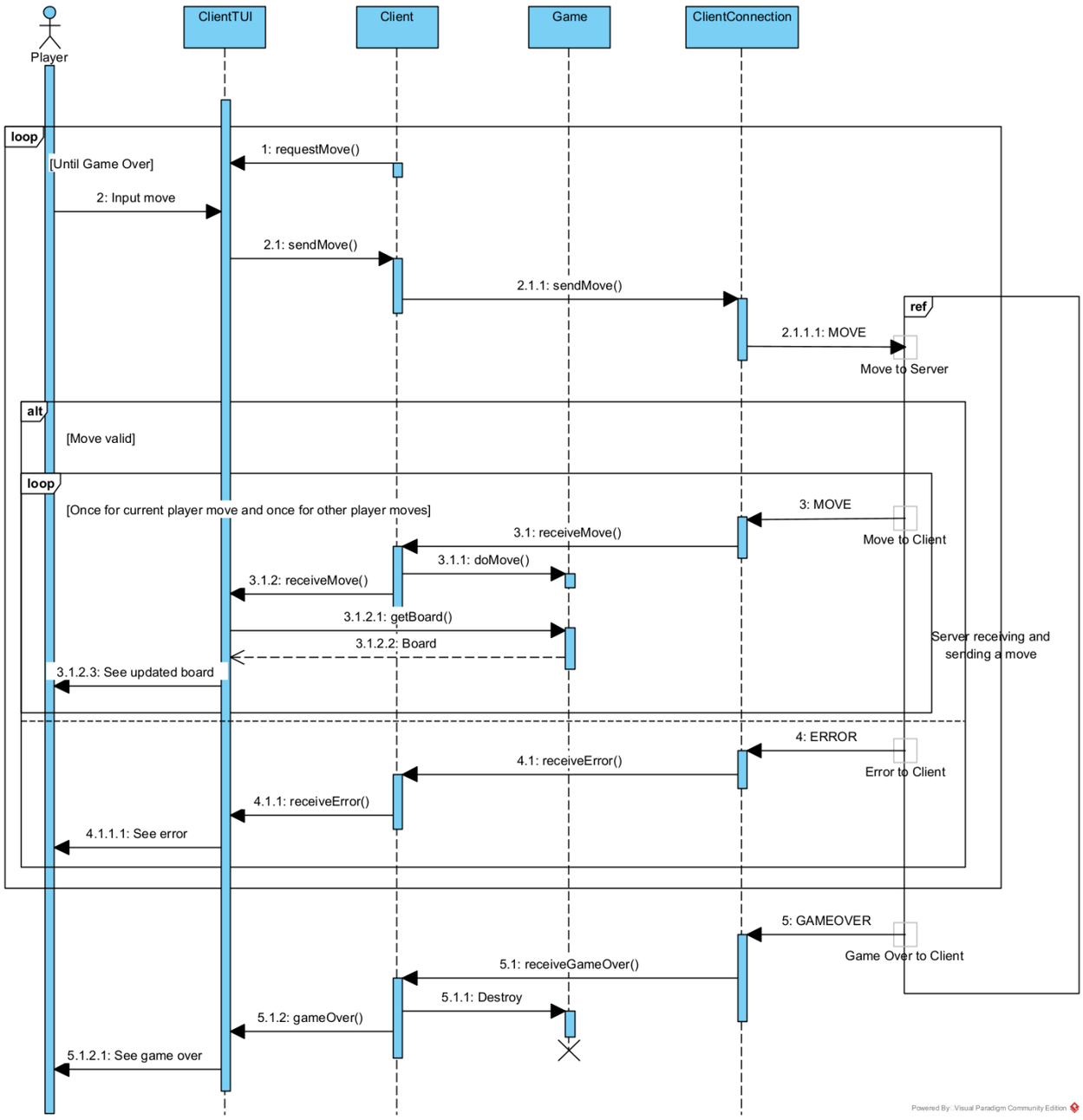


Figure 45: Sequence diagram of the client handling a move

In figure 46 the executing flow for a server sending and receiving a move is displayed. If the server receives a move from a client via the ServerConnection, it is send to the ServerClientHandler, which sends it to the ServerGameManager. This ServerGameManager handles the move by checking if it is valid or not. If it is a valid

move the move is performed on the game and a message is send to the client that the move is performed. If the move is invalid an error message is send to the client. This process is repeated until the ServerGameManager assessed the game as game-over. In this case a game-over message is send to both clients that are connected to the game and the ServerGameManager's association with the game is deleted.

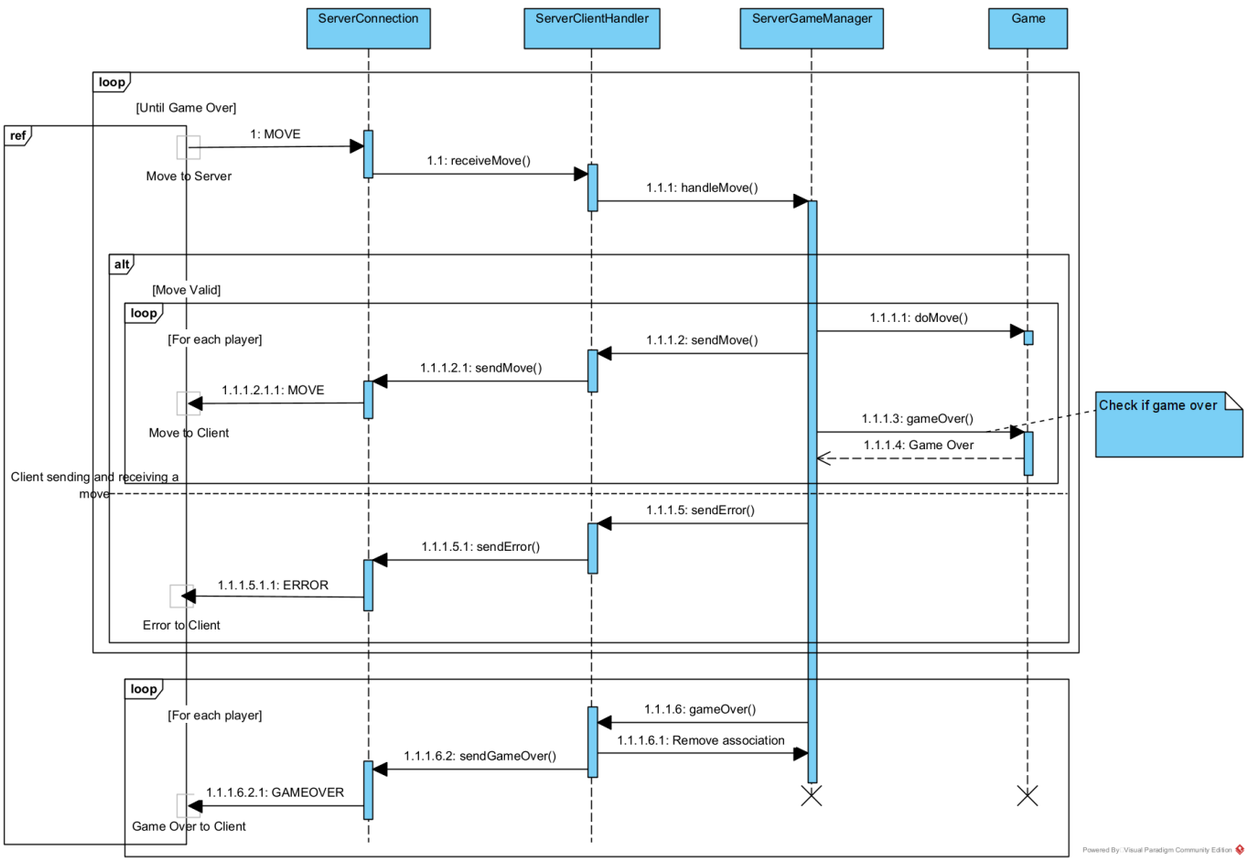


Figure 46: Sequence diagram of the server handling a move

In figure 47 the executing flow for a server establishing a connection is displayed. The server first gets a connection, for which it then creates a ServerClientHandler which creates a ServerConnection. It then exchanges HELLO messages with the client. Then the client sends a username, which the server checks for uniqueness and then responds with a login confirmation of error. After the client is logged in, it asks the server to be put in a queue. The server puts the client in a queue, and if there are now 2 clients in the queue it creates a ServerGameManager and sends a NEWGAME message to the two clients, thus beginning the game.

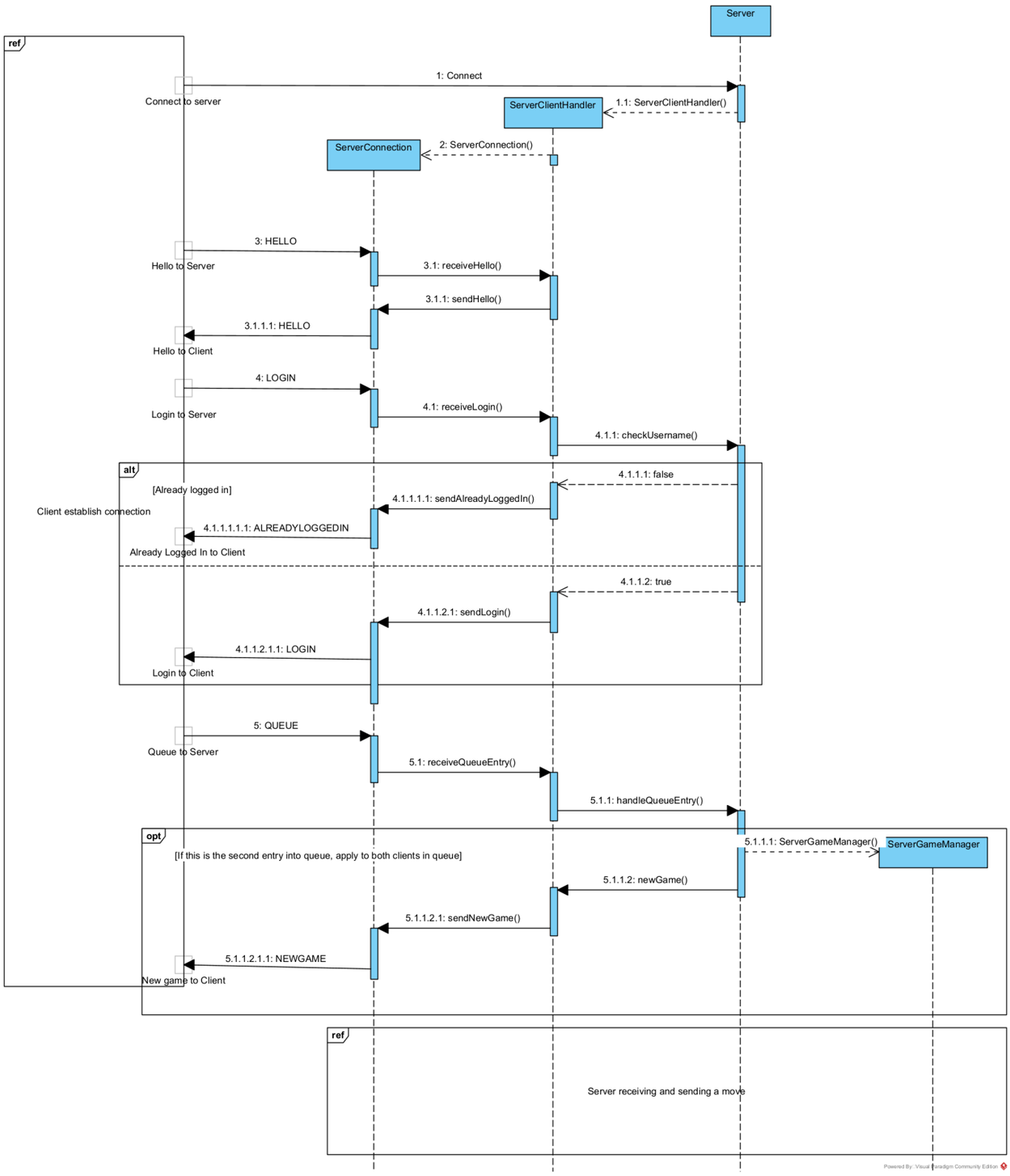


Figure 47: Sequence diagram of the server establishing a connection

In figure 48 the executing flow for a client establishing a connection is displayed. The ClientTUI asks the user for a host and port number to connect to. It then creates a Client object with this host and port number, which creates a ClientConnection object. If this is done correctly a hello message is send to the server with a client description. The server responds to this by sending a hello message back with a server description. Afterwards the ClientTUI asks the user for a username, which is send to the server. If this name is already taken, the client receives a message that this name is already known to the server. If the client is correctly logged in, it receives a confirmation from the server and it enters the queue. This request for entering the queue is send to the server. Once the client receives the message NEWGAME, it knows that it is put into a newly started game with another client.

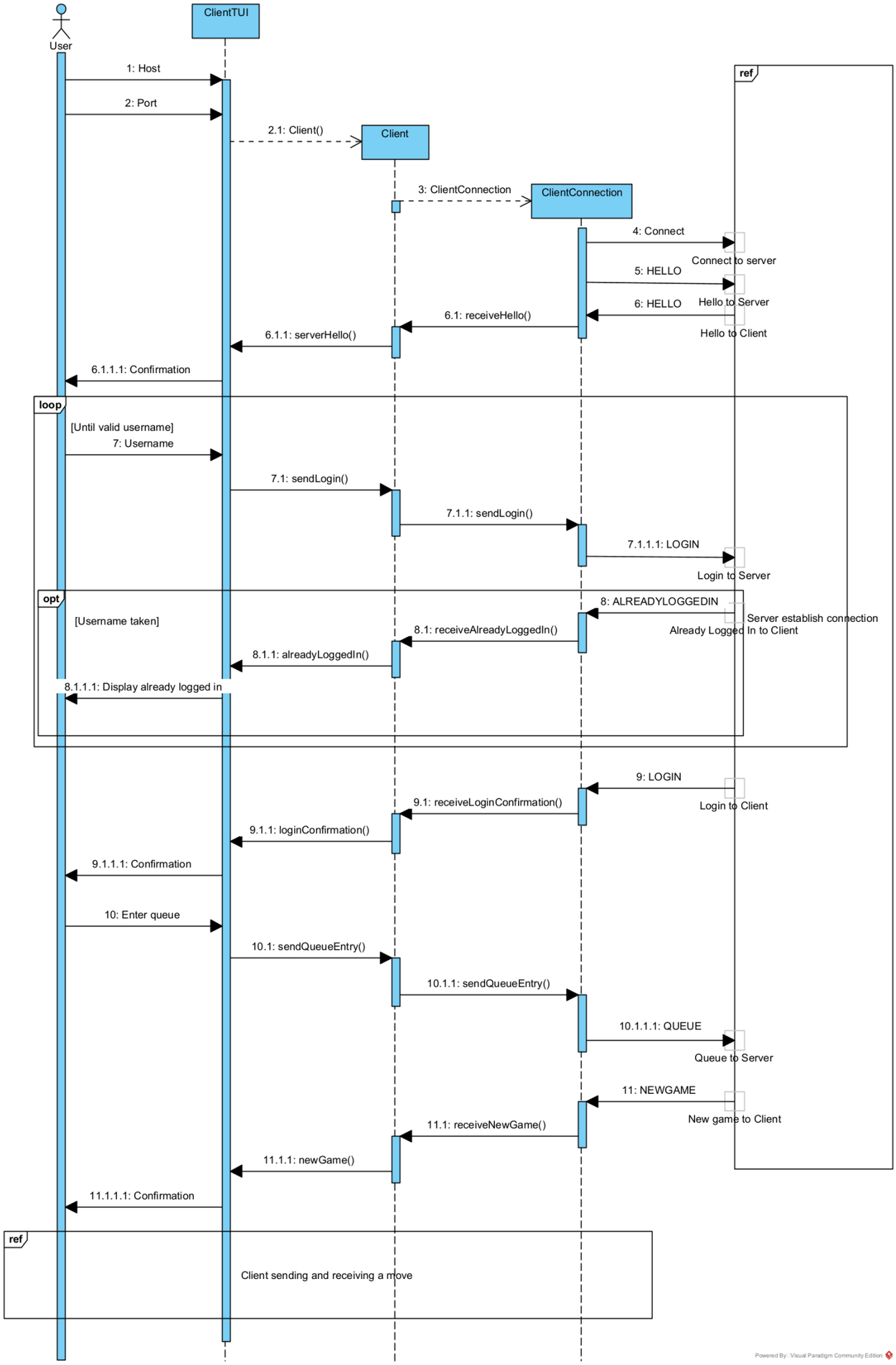


Figure 48: Sequence diagram of the client establishing a connection

Figure 49 shows a state diagram of the TUIClient. It indicates the flow of the user interface. Each state in this diagram will be part of a enumerator, with a looping switch case calling the right method for the current state. These methods can then change the state to the next applicable state before returning.

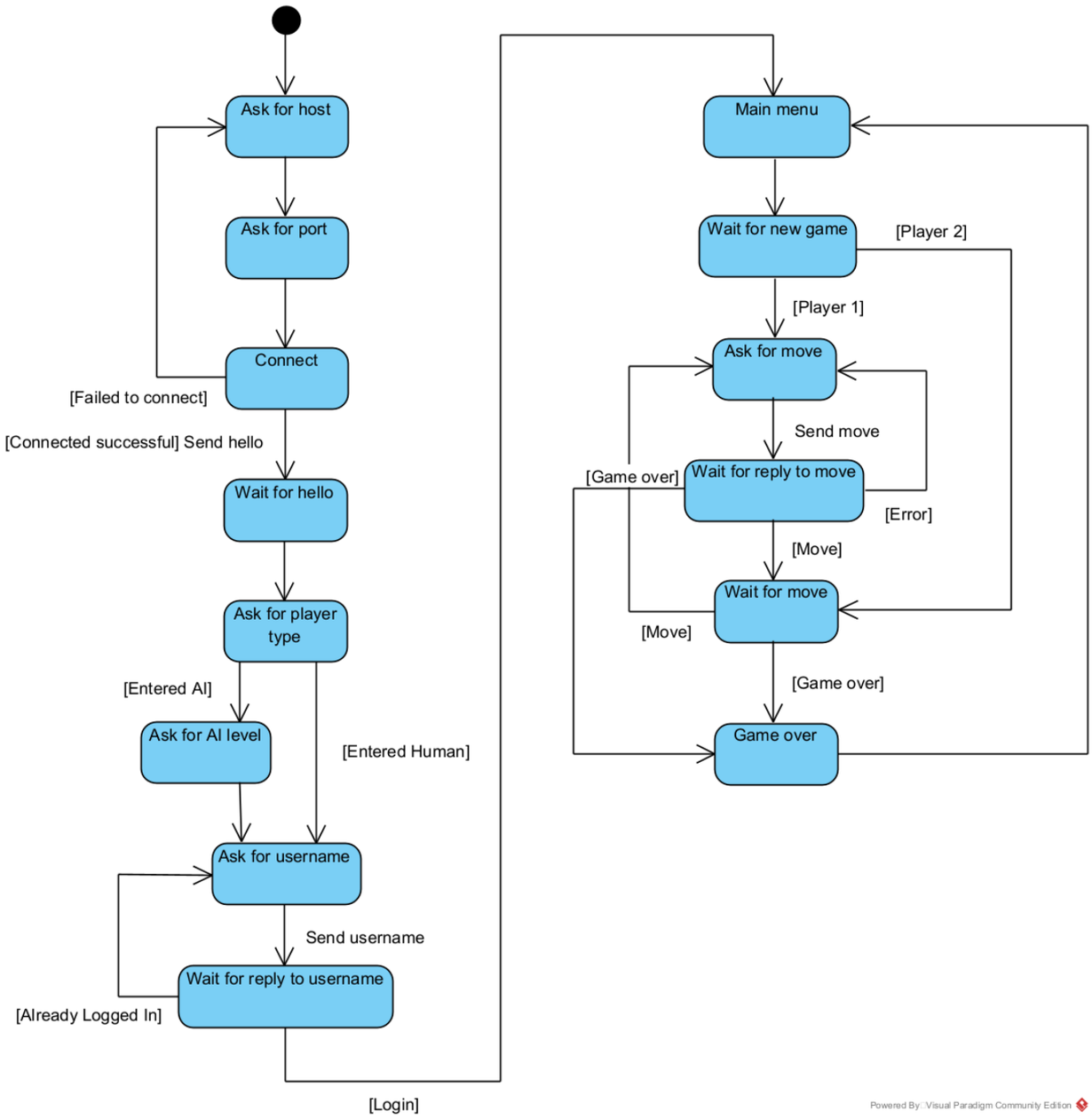


Figure 49: State diagram of the TUIClient

8.1.3 Threads

We need a couple of threads in our program.

Server:
The initial thread from the server waits for and accepts new ServerConnections. When the server gets a connection, it calls the method handleConnection() and it creates a ServerClientHandler for this connection. The ServerClientHandler updates the list of clients, where concurrency could occur. Furthermore it creates and starts a ServerConnection. Each ServerConnection has its own thread, that waits for messages from its assigned client. These threads end when the connection with the client is closed. The initial server thread only ends when the server closes.

The list of clientHandlers can be updated by the main server thread when a client connects and by client connection threads when their client disconnects. Furthermore, it can be iterated over by the client connection threads. To prevent conflicts, these operations should be synchronized over this property.

Furthermore, receiveLogin() is called by client connection threads and iterates over the usernames of all clients and then updates its own username. To prevent the same username from being changed and read at the same time, this property should be synchronized for each of the ServerClientHandler instances.

There is a potential for concurrency in the ServerGameManager, as this receives calls from two client connection threads on its handleMove() and handleDisconnect() methods. However, handleMove() only allows calls from the thread of the player who currently has a turn, and the turn is only handed over to the other player after everything has finished updating. The handleDisconnect() causes a game over message to be send to both clients and then removes the reference to the ServerGameManager from both ServerClientHandlers. The thread of the other client locks its reference to the ServerGameManager and checks if its null to prevent a race condition where the reference is set to null because of a disconnect while the other thread is processing a move.

Client:
The client has two threads: the interface and the connection. The thread for the connection is created when connection with the server is established and ends when the connection closes. To prevent conflicts, the interface

will be designed to be able to handle incoming messages at any time. This will be expanded upon further when the design of the interface is decided upon.

8.1.4 Responsibilities

We divided the work as follows. Marinus:

- Networking
- UI
- Client

Ylona:

- game logic
- AI
- Server