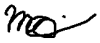
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

EPE 321

SOFTWARE ENGINEERING

GROUP DESIGN DOCUMENT

GROUP M

Name and Surname	Student Number	Signature	% Contribution
H. van der Westhuizen	18141235		25.0
C.H. Conroy	18072918		25.0
M. Winnan	18037497		25.0
C.F. Wagner	04503237		25.0

By submitting this assignment I confirm that I have read and am aware of the University of Pretoria's policy on academic dishonesty and plagiarism and I declare that the work submitted in this assignment is my own as delimited by the mentioned policies. I explicitly declare that no parts of this assignment have been copied from current or previous students' work or any other sources (including the internet), whether copyrighted or not. I understand that I will be subjected to disciplinary actions should it be found that the work I submit here does not comply with the said policies.

September 2, 2021

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The rubber contract bridge application is divided into four major parts. Each member is responsible for the design, implementation and integration of their respective classes and features.

2 REFERENCES

[1] C. H. Conroy, *Assignment 3: Individual Design Document*, Pretoria, South Africa, Sep. 2020.

[2] N. Nguyen. (Apr. 2018). 30+ pretty iphone wallpapers that don't cost a thing, [Online]. Available: `https://www.popsugar.com/tech/photo-gallery/31811686/image/31812265/Geometric-Squares`.

[3] P. Kok, *EPE 321 - Communication Protocol*, Pretoria, South Africa, Sep. 2020.

[4] C. F. Wagner, *Individual Design Document*, Pretoria, South Africa, Sep. 2020.

[5] M. L. Ginsberg, "Gib: Steps toward an expert-level bridge-playing program," in *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999, pp. 584–589.

[6] M. Winnan, "Individual design document," University of Pretoria, Tech. Rep., Sep. 2020, p. 21.

3 GAME SERVER

The Game Server portion of the software entails several classes which provide functionality related to the core server and game functionality. The Server class is responsible for coordinating the server's networking, GUI and network classes. Of those classes only the GameServer falls within the Game Server scope. The GameServer is responsible for controlling the game flow between players and maintaining the game state through the GameState object and its subclasses. The Player class aims to provide a common interface to both the human and AI players for the GameServer. The Bid, Card and CardSet classes represent bids and cards that are used by several of the other classes within the Game Server section. The classes found in [1] was refined for this design.

3.1 UML CLASS DIAGRAMS

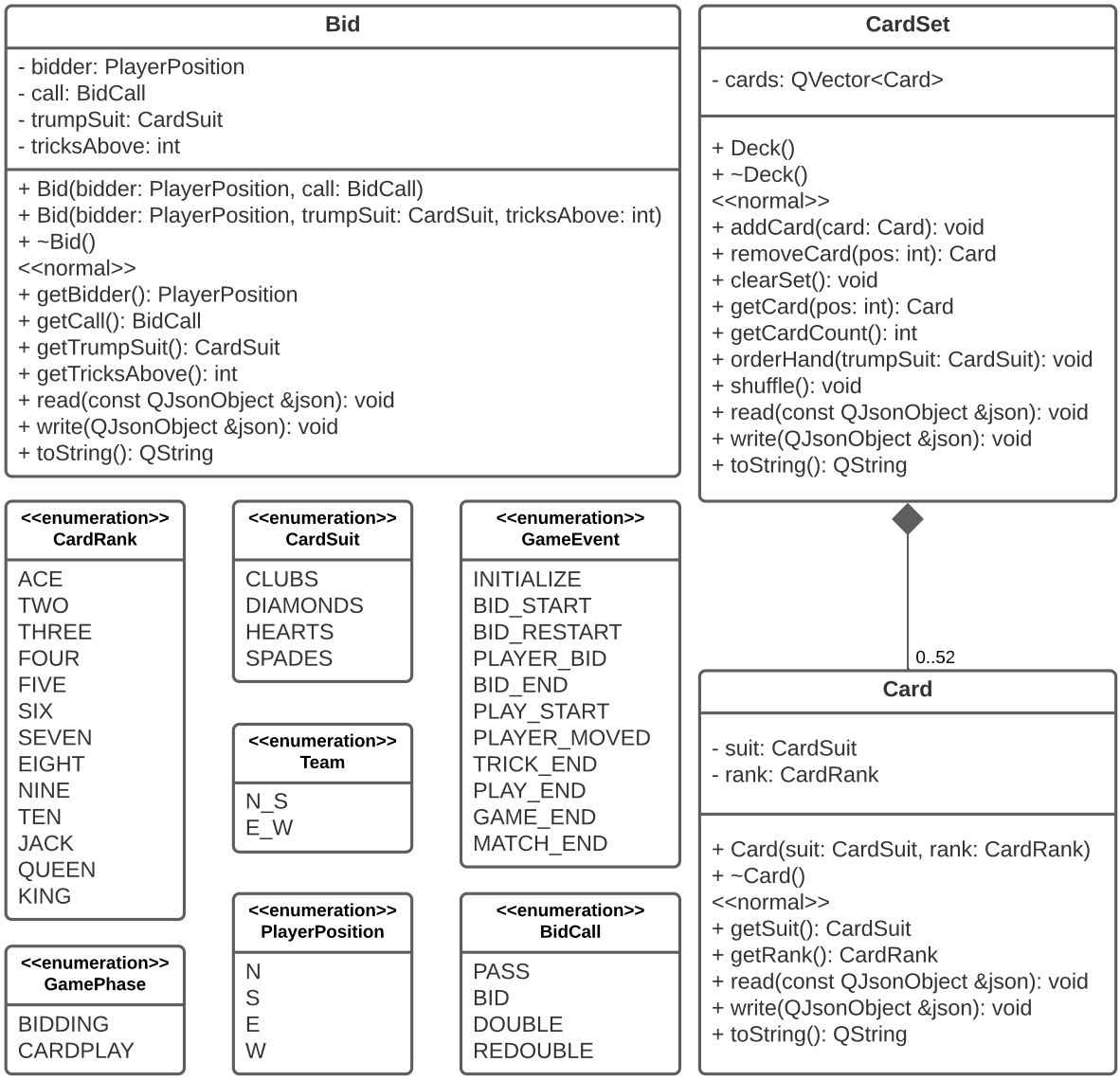


Figure 1: GameServer supporting classes and enumerations UML class diagram.

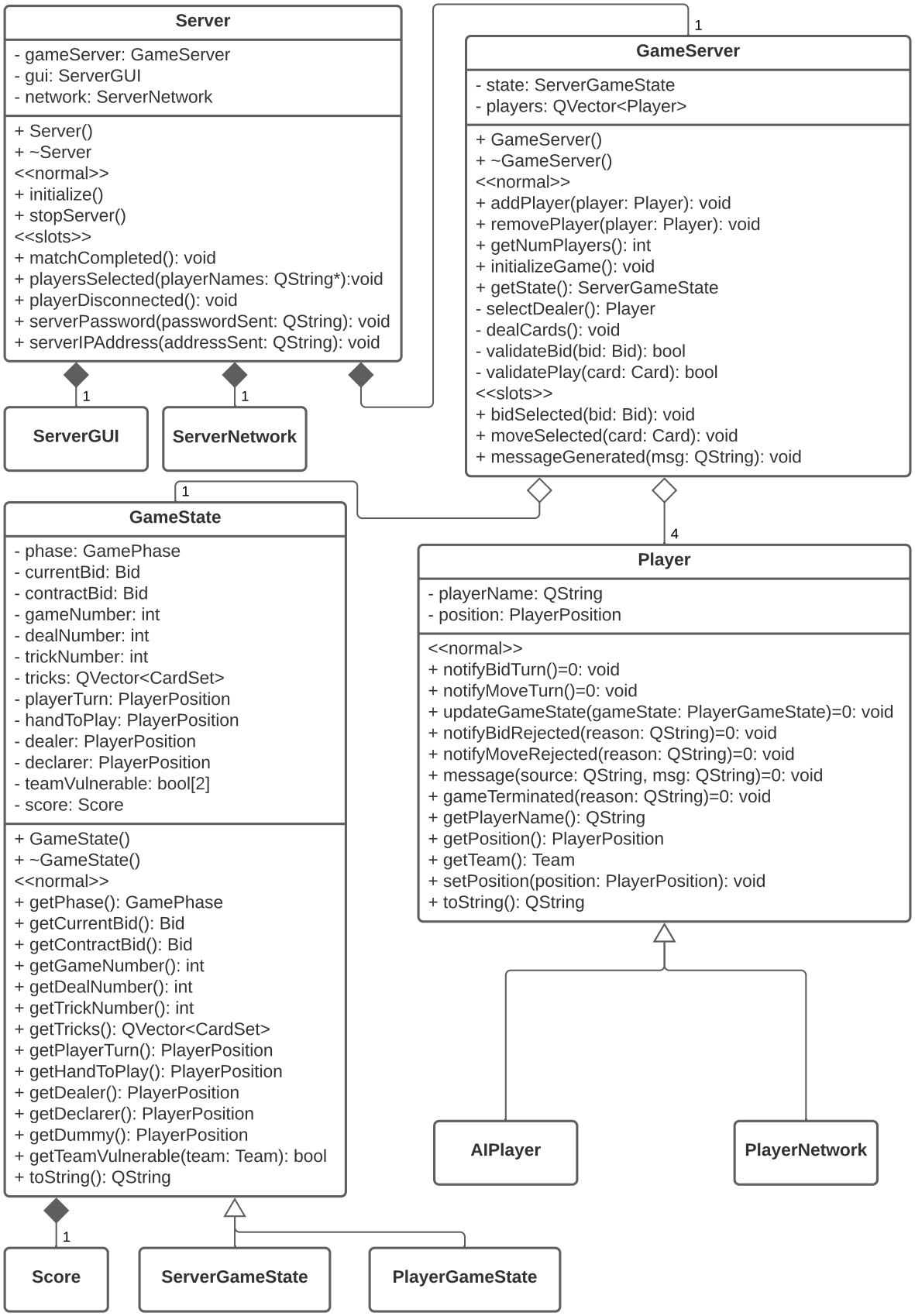


Figure 2: GameServer UML class diagram.

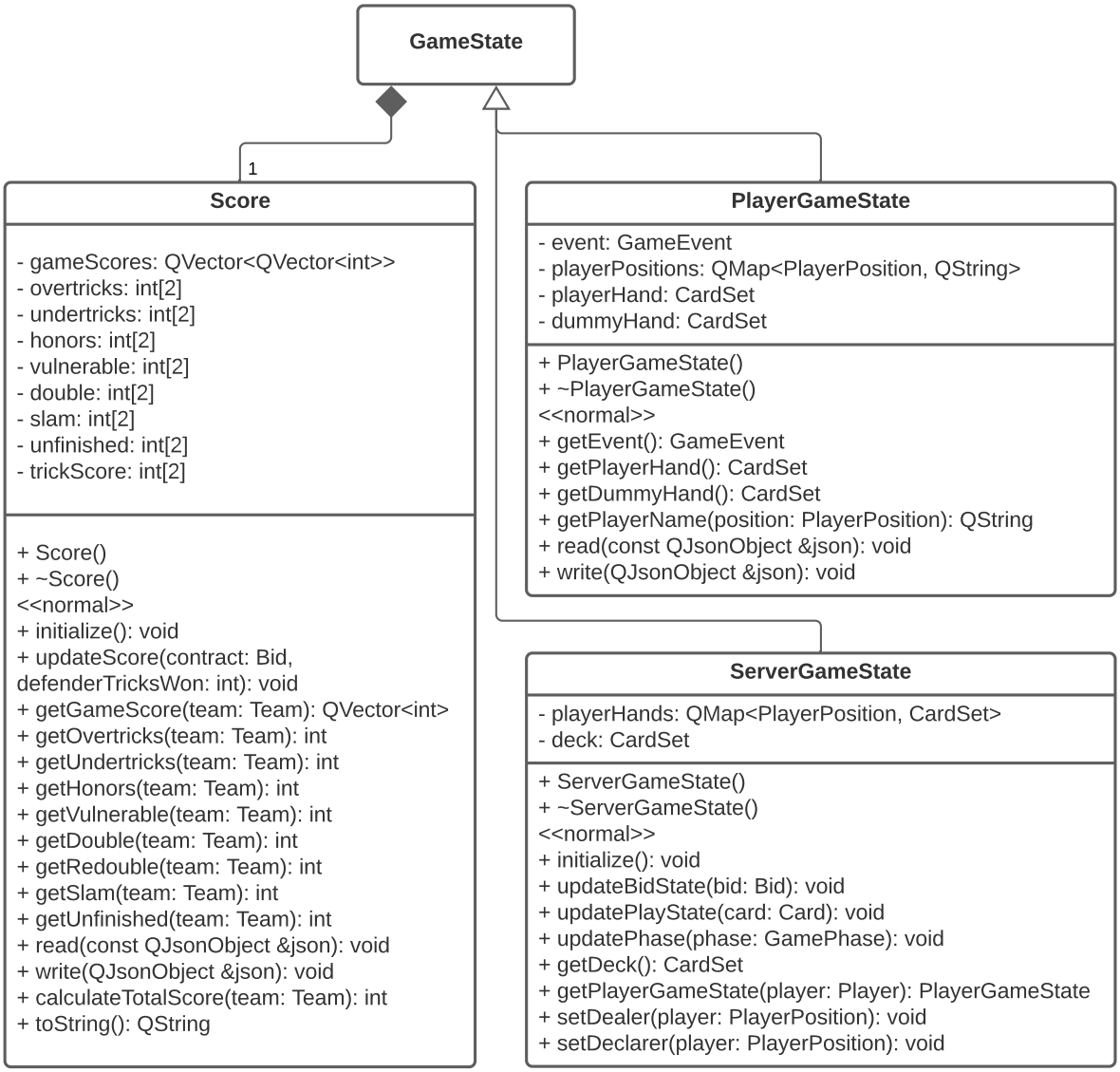


Figure 3: GameState supporting classes UML class diagram.

3.2 SIGNAL AND SLOTS

Signals	Slots [Class]	Purpose
matchStarted()	matchStarted [ServerGUI]	Indicate to the server GUI that the match process has begun to update the server display
matchCompleted()	matchCompleted [Server]	Indicate to the server instance that the match has been completed
gameStateUpdated(gameState: ServerGameState)	gameStateReceived [ServerGUI]	Indicate the to the server GUI that the game state has been updated to update the server display

Table 1: GameServer's signals.

Signals	Slots [Class]	Purpose
bidSelected (bid: Bid)	bidSelected [GameServer]	Send the bid made by the player to the Game Server.
moveSelected (card: Card)	moveSelected [GameServer]	Send the move made by the player to the Game Server.
messageGenerated (msg: QString)	messageGenerated [GameServer]	Send message received from the player to the Server, so that the server can relay it to the other players.

Table 2: Player's signals.

3.3 UNIT TESTS

Number	Tested component	Input	Expected output
1	Bid object	Either Bid constructor is called with a set of valid or invalid arguments	Exception thrown if arguments are invalid or string output indicating correct member variables when toString is called
2	Card object	Card constructor is called with a set of valid or invalid arguments	Exception thrown if arguments are invalid or string output indicating correct member variables when toString is called
3	CardSet object	CardSet constructor is called with a set of valid or invalid arguments and several addCard and removeCard calls	Exception thrown if arguments are invalid or string output indicating correct member variables when toString is called
4	Ordering of card objects in the card set	Call orderHand on CardSet object containing 13 Card objects	Card objects ordered by suit then rank in cards vector
5	Score object initialization	Score constructor is called followed by initialize function	String output indicating correct member variables when toString is called
6	Score update after trick	updateScore is called with valid bid object	String output indicating correctly updated member variables when toString is called
7	Total score calculation	calculateTotalScore is called for each Team value	Correct numerical value compared to sum of member variables shown by toString

Number	Tested component	Input	Expected output
8	Update ServerGameState based new bid	updateBidState slot is called with valid Bid object	String output indicating correctly updated member variables when toString is called
9	Update ServerGameState with new move	updatePlayState slot is called with valid Card object	String output indicating correctly updated member variables when toString is called
10	Validate bid	Valid or invalid Bid object based on previous bid	True if the bid input is valid or false otherwise
11	Validate move	Valid or invalid Card object based on current game state	True if the card input is valid or false otherwise
12	Game initialization	GameServer constructor followed by 4 addPlayer calls and an initialize call	String output indicating correct member variables when toString is called
13	Creation of QJsonObject from Bid, Card, CardSet, Score or PlayerGameState object	Call to the write function of each of the stated classes for an existing instance	String output indicating correct member variables when toString is called for instance of each class
14	Creation of Bid, Card, CardSet, Score or PlayerGameState object from QJsonObject	Call to the read function of each of the stated classes with a QJsonObject instance	QJsonObject with correct values for member variables and classes

Table 3: Unit tests for the Game Server classes.

4 GUI AND CLIENT

The Graphical User Interface of the application will be implemented through the use of two classes namely the ServerGUI and ClientGUI. These classes are responsible for handling the interactions between the game host and the server as well as the interactions between the players and the game.

4.1 UML CLASS DIAGRAMS

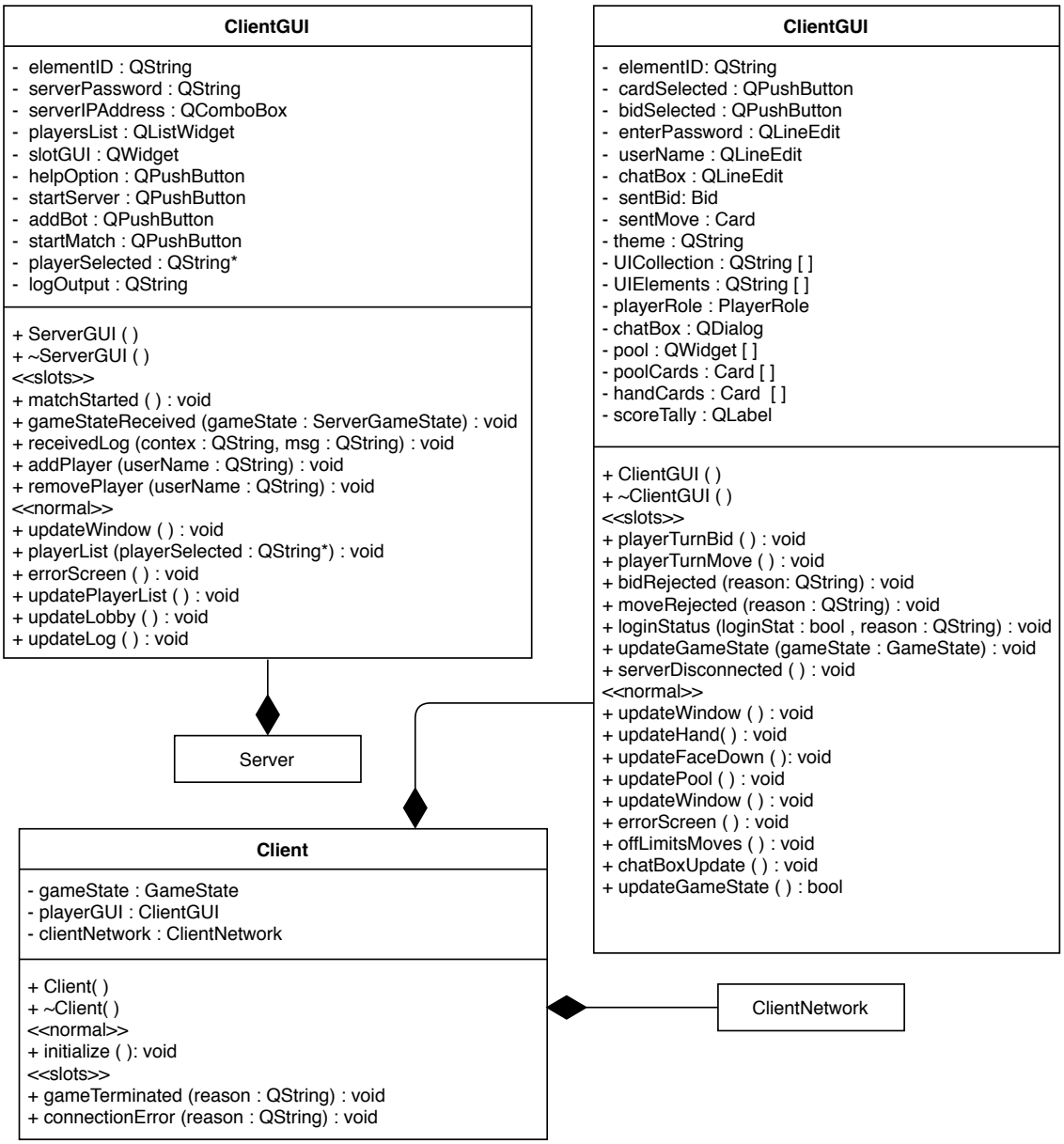


Figure 4: GUI and Client UML class diagram.

Since a large portion of the game-play is facilitated by the ClientGUI the Client class just acts as the instantiating class for the ClientGUI and ClientNetwork.

4.2 SIGNAL AND SLOTS

Signals	Slots [Class]	Purpose
bidAction(bid : Bid)	txBidSelected [ClientNetwork]	The player's GUI sends the desired bid as an object to the ClientNetwork.
moveAction(card : Card)	txMoveSelected [ClientNetwork]	The player's GUI sends the desired move as an object to the ClientNetwork.
dialogMessage(message : QString)	txMessage [ClientNetwork]	A message sent through the dialog box is sent to the other players via the ClientNetwork.
connectToServer (serverIP: QHostAddress, playerName : QString, password : QString)	txRequestLogin [ClientNetwork]	The GUI sends their IP address, username and password to the ClientNetwork to connect the client to the server.

Table 4: ClientGUI's signals.

Signals	Slots [Class]	Purpose
playerList (playerSelected: QString [])	playerSelected [Server]	An array of usernames are sent to the server to indicate which slots are occupied by the selected players and by AI controlled bots.
serverPassword(passwordSent: QString)	serverPassword [Server]	The server receives the password that needs to be enter by the clients to start the game.
serverIPAddress (addressSent: QString)	serverIPAddress [Server]	The server receives the IP address that will be used to host the game.

Table 5: SeverGUI's signals.

4.3 UNIT TESTS

Number	Tested component	Input	Expected output
1	Player login details are received and checked for compliance.	A username and password is typed into the given fields. Login button is pressed.	The username and password is sent to the ClientNetwork and the player establishes a connection with the server.
2	Player selects a valid bid option and the option is validated by the server.	The player clicks on a of the valid bid option in the bidding phase.	The bid is placed and the action is sent to the server through the ClientNetwork. The GUI updates and indicates the bid has been made successfully.
3	Player selects an invalid bid option and is rejected by the server.	The player clicks on an invalid (greyed out) bid option.	The player is prompted with a QMessageBox indicating that an invalid bid has been made.
4	Player selects a valid move and the move is validated to the Server.	The player clicks on a card to play during their turn.	The move is sent in the form of a Card object to the server through the ClientNetwork.
5	Player selects a invalid move is rejected by the Server.	The player clicks on a opponents card or out of their turn.	The player will not be able to play a opponents card or out of turn. No Card object is sent.
6	The client receives the updated GameState which causes the GUI to change validated accordingly.	The player's Client Class receives the updated GameState from the Server class.	The ClientGUI updates the game board by either adding a card from the player's hand into the pool or reducing the possible bids available.
7	The ServerGUI sends the password and server IP address to the server.	The host enters the password and IP address of the host server.	The server password and IP address will allow the clients to connect to the server.
8	The user enters the Server or Client side of the application.	The user clicks on the desired option.	The application sets up all the classes the server will need if the user selected the Host option otherwise all the client side classes are setup.

Number	Tested component	Input	Expected output
9	A log of all the moves and bids by the players are displayed on the ServerGUI.	The ServerGUI receives the bid or move information from the Logger class.	The ServerGUI displays the moves and bids in chronological order.
10	Players and AI bots are added to the correct slot in the lobby.	The host selects the players as well as any AI bots for the match.	The game should start with the players in the correct positions.

Table 6: Unit tests for the GUI and Client classes.

4.4 GUI DESIGN

The GUI interface is designed and implemented with easy of use as the highest priority. The player should feel as if they are guided through the various interaction windows such as the login screen, server select screen, bidding screen and gameplay screen.

The preliminary design specifications of the various interaction windows and card graphics are shown below:

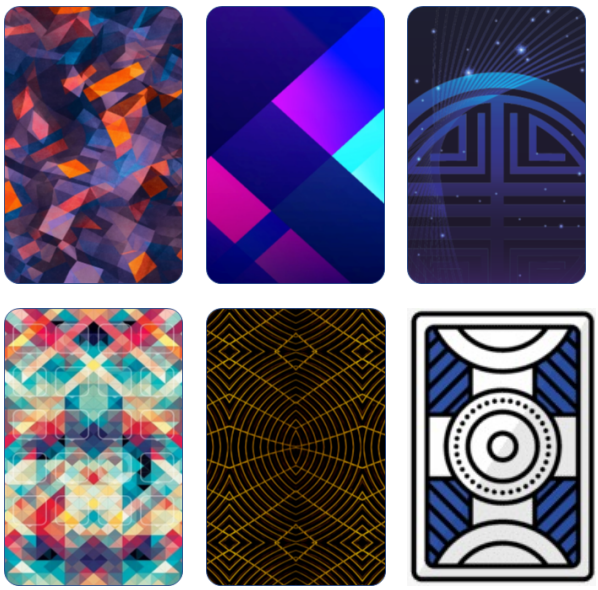


Figure 5: Deck design ideas.

Figure 5 show a few deck design ideas that may be used in the final implementation of the game. All the designs are copy right free and fall under the fair use act as they appear on [2]. Players may also have the ability to choose their desired design while playing.

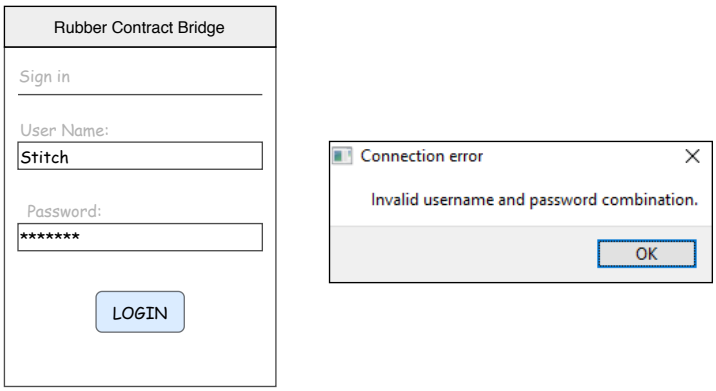


Figure 6: Player login window and error alert.

The login screen for the players is very standard and consists of 2 text boxes for the username and password. If the username is invalid or has already been taken, the user is notified with a QMessageBox popup. On the server side the game host will create the game by selecting the players from a pool of players connected to the server. The slots can either be filled with human or AI controlled bots.

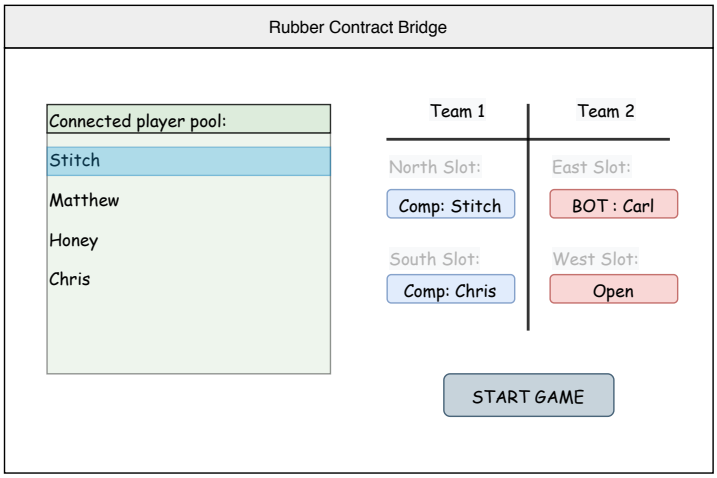


Figure 7: Lobby window.

Once the game has started the players will be loaded into the main game window where the bidding phase begins.

In the bidding phase the players are presented with all the available bidding options. Every time a player places a bid, the card goes into their play area as seen by the 1 of Diamonds. If a bid has been made the remaining options should be updated, this means that any invalid bids should be removed such as the 1 of clubs. The bidding phase ends once no other bids can be placed or 3 Passes have been made.

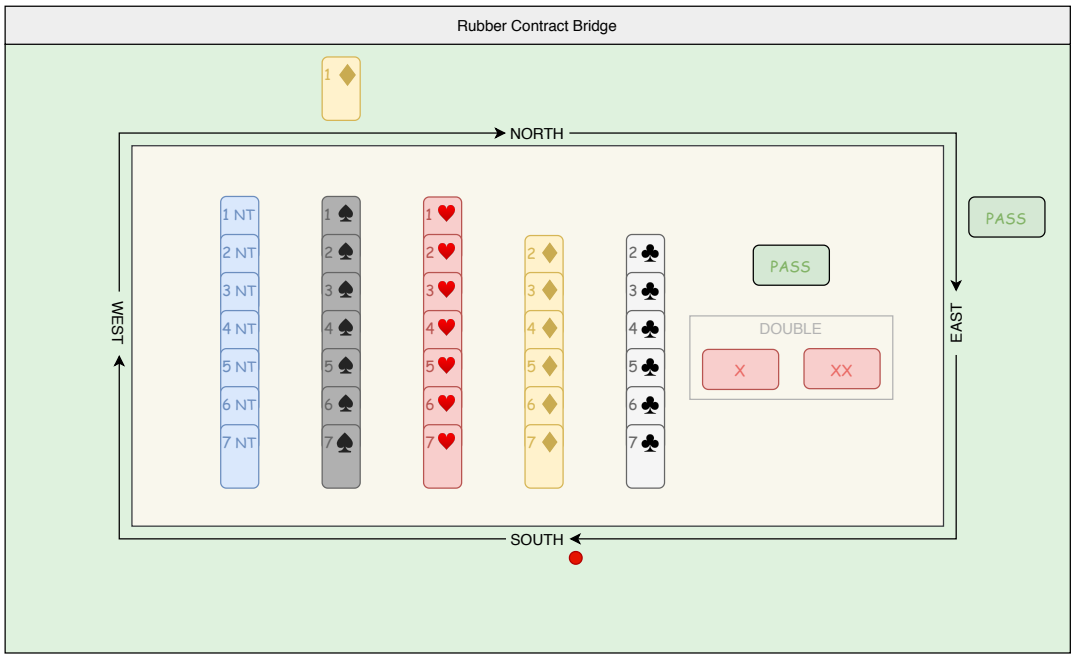


Figure 8: Bidding window.

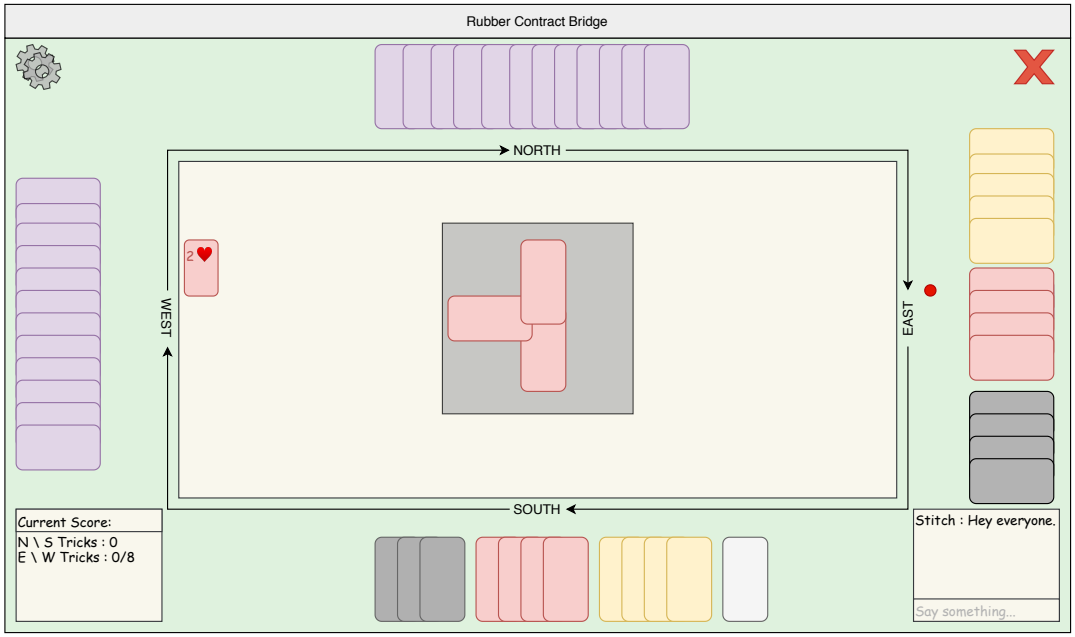


Figure 9: Game window.

Once the bidding phase has been completed the game starts. The host deals the cards out to the players, each player can only see their own hand as indicated by the different coloured cards at the bottom. Once the dummy hand gets his turn their cards are revealed as seen on the EAST side of the board. The red dot indicates it is the EAST player's turn to play a card. Once a card is picked it is added to the pool where the highest value card wins the pool for the team. In the bottom right the chat box can be interacted with to send messages to all the other players.

5 NETWORK AND LOGGER

The server and client communicates with each other over a network. The Network classes handle the connection and login process. The password and username validation functionality is contained within the NetworkServer class.

The Network perform the conversion to and from the QJsonObject format, that is used to send data over the network. The ”Typ” names outlined in [3] will be followed as far as it is relevant.

While a game is in progress, the client and server exchanges keep alive messages. This is to quickly notify the server and client when a client expectantly disconnects. A detailed description can be found in [4].

The Logger class is signaled every time something should be logged. It will then print the log on the command line and send the log to the server's GUI via a signal.

The function descriptions in the Network classes can be found in [4].

5.1 UML CLASS DIAGRAMS

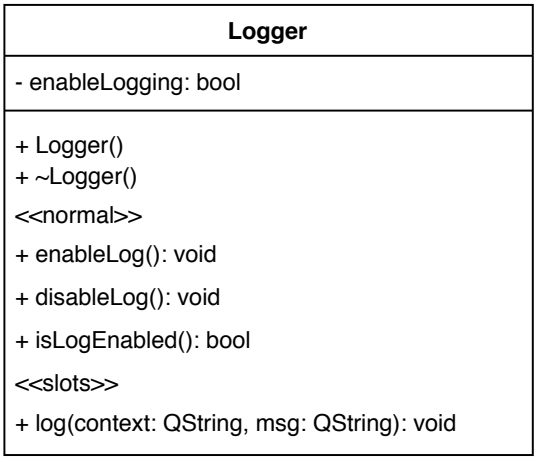


Figure 10: Logger UML class diagram.

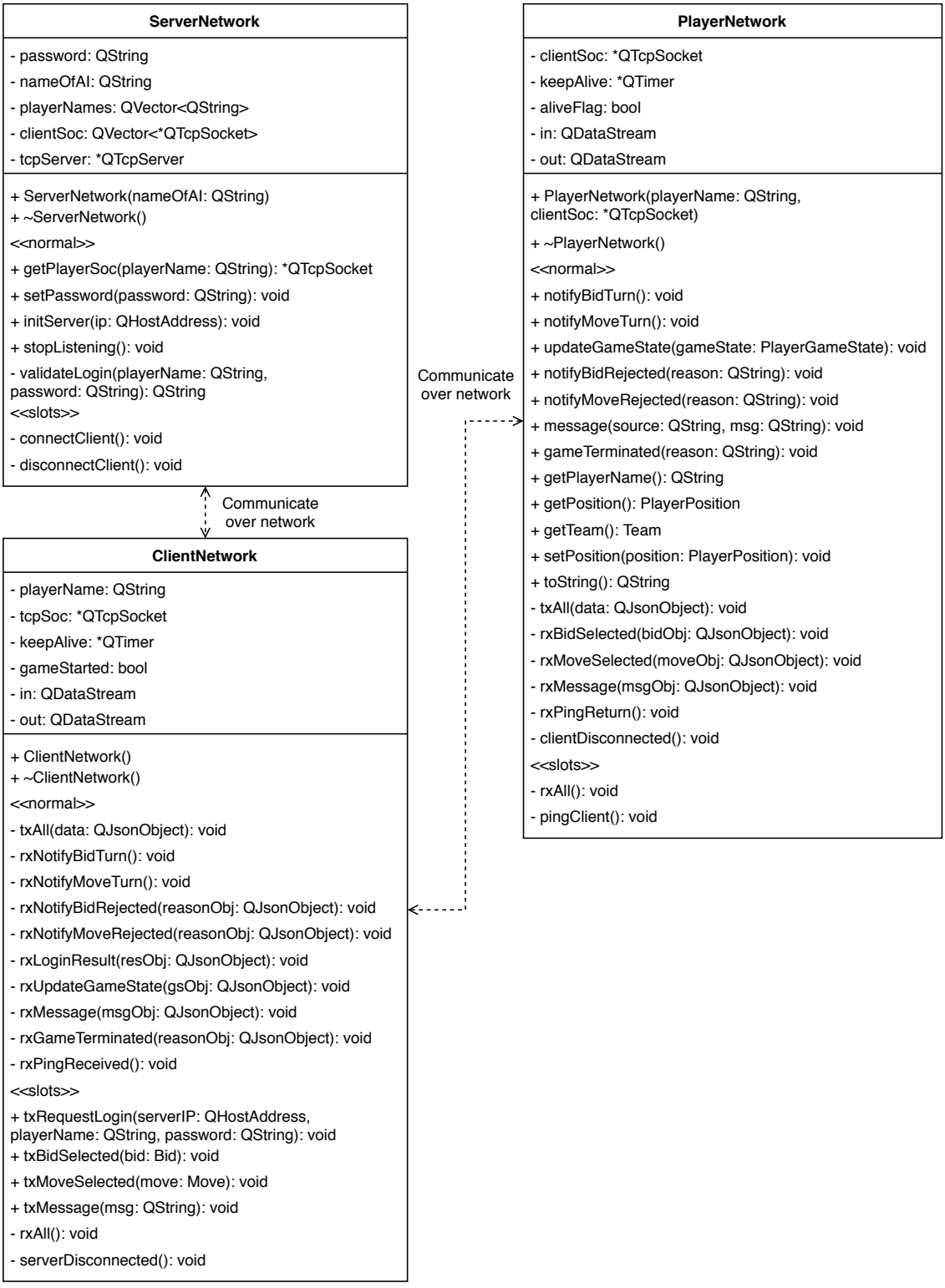


Figure 11: Network UML class diagram.

5.2 SIGNAL AND SLOTS

Signals	Slots [Class]	Purpose
sendLog (context: QString, msg: QString)	receivedLog [ServerGUI]	Display the log item on the server's GUI.

Table 7: Loggers' signals.

Signals	Slots [Class]	Purpose
playerJoined (playerName: QString)	addPlayer [ServerGUI]	To inform the server's GUI that a new player has successfully logged into the server and should be displayed on the GUI.
playerDisconnected (playerName: QString)	removePlayer [ServerGUI]	To inform the server's GUI that a player has left the server and should be removed from the GUI.

Table 8: ServerNetwork's signals.

Signals	Slots [Class]	Purpose
bidSelected (bid: Bid)	bidSelected [GameServer]	Send the bid made by the player to the Game Server.
moveSelected (card: Card)	moveSelected [GameServer]	Send the move made by the player to the Game Server.
messageGenerated (msg: QString)	messageGenerated [GameServer]	Send message received from the player to the Server, so that the server can relay it to the other players.
timeout()	pingClient [PlayerNetwork]	Notify the PlayerNetwork class that the keepAlive timer has timed out. Thus the aliveFlag should be checked and the next ping should be sent to the client.
clientDisconnected()	playerDisconnected [Server]	Infrom the server that the client has disconnected and that the game should be terminated.

Table 9: PlayerNetwork's signals.

Signals	Slots [Class]	Purpose
serverNotFound (reason: QString)	connectionError [Client]	To infrom the client that the server could not be found.
notifyBidTurn()	playerTurnBid [ClientGUI]	Notify the client that it is their turn to make a bid.
notifyMoveTurn()	playerTurnMove [ClientGUI]	Notify the client that it is their turn to make a move.
notifyBidRejected (reason: QString)	bidRejected [ClientGUI]	Notify the client that their previous bid was invalid.
notifyMoveRejected (reason: QString)	moveRejected [ClientGUI]	Notify the client that their previous move was invalid.
loginResult (loginSuccessful: bool, reason: QString)	loginStatus [ClientGUI]	Inform client whether or not the login was successful.
updateGameState (gameState: GameState)	updateGameState [ClientGUI]	Send the newest game state to the client.
messageReceived (source: QString, msg: QString)	chatMsgReceived [ClientGUI]	Send message received to the client's GUI.
timeout()	serverDisconnected [ClientNetwork]	Notify the ClientNetwork class that the keepAlive timer has timed out. Thus the server has been disconnected.
serverDisconnected()	serverDisconnected [ClientGUI]	Infrom the client that the server has disconnected and that the game has been terminated.
gameTerminated (reason: QString)	gameTerminated [Client]	Infrom the client that the server has terminated the game and provide a message that should be displayed to the user.

Table 10: ClientNetwork's signals.

5.3 UNIT TESTS

Number	Tested component	Input	Expected output
1	Logging data received.	A log signal with context and msg QString parameters.	A logSend signal with the exact same context and msg QStrings are signaled on the server.
2	Converting the data to a QJsonObject.	Call all the functions that should create a QJsonObject with their required parameters.	The resulting QJsonObject contains the "Typ" and "ID" fields, as well as all the expected fields and data for that function.

Number	Tested component	Input	Expected output
3	Converting a valid QJsonObject to the original data format.	QJsonObjects with different fields and data. Valid and expected fields and data for that JSON message. Some with and without extra fields.	The ”Typ” and ”ID” fields are present, ”Typ” contains expected data and all the expected fields and data are present. Extra fields are ignored.
4	Converting an invalid QJsonObject to the original data format.	QJsonObjects with different fields. Some without the ”Typ” and/or ”ID” fields, as well as unexpected data in the Typ fields.	A log signal with ”An invalid QJsonObject was received and ignored.” is generated on the server.
5	Valid username and password.	Call validatePassword with a unique, non-empty username with a max length of 10 chars and the correct password.	validateLogin returns an empty QString.
6	Invalid password.	Call validatePassword with a unique, non-empty username with a max length of 10 chars and the incorrect password.	validateLogin returns a QString containing ”The password is invalid. Please try again.”.
7	Invalid username.	Call validatePassword with an already used, empty, longer than 10 chars or the AI’s username. The correct password.	validateLogin returns a QString containing ”The username has already been used. Please try another username.”, ”The username should not be empty. Please try again.”, ”The username may not be longer than 10 characters. Please try again.” or ”The username may not be the same as an AI’s username. Please use a different username.” depending on the mistake in the username.

Number	Tested component	Input	Expected output
8	Client connects to the server successfully.	Call requestLogin with a valid username, password and host IP address on the client. (Server should be listening.)	A loginResult signal with True and an empty QString is generated on the client.
9	Client connects to the server with an incorrect host IP address.	Call requestLogin with a valid username, password and incorrect host IP address on the client.	A serverNotFound signal with a non-empty QString is generated on the client.
10	Client connecting to the server with invalid username or password.	Call requestLogin with an invalid username and/or password and valid host IP address on the client. (Server should be listening.)	A loginResult signal with False and a non-empty QString is generated on the client.
11	Sending a QJsonObject between the client and server. (Ping and Pong.)	Send a QJsonObject containing the ”Typ”: ”PING” and ”ID” fields. (There should already be a connection between the client and server.)	A QJsonObject with the ”Typ”: ”PONG” and an ”ID” field are received on the server within 5 seconds.
12	Client disconnected from the server.	On a server that is not connected to a client, set aliveFlag to True and signal pingClient().	clientDisconnected is signaled on the server.
13	Server disconnected from the client.	On a client that is not connected to a server, start the keepAlive timer.	serverDisconnected is signaled on the client.

Table 11: Unit tests for the Network and Logger classes.

6 AI

The AI just like the Player class must be able to receive a signal which contains the GameState object. This game state will be in received in a QJsonObject format. The AI must use this to analyse the game state and look for potential moves.

When it's the AI's turn it must use the information from the game state to suggest the best possible move and send a signal that contains this suggested move back to the game server class.

There are two phases in bridge and every phase will have a different approach that the AI will use to calculate the potential move. For bidding the AI will try to make sure that the trump suit will be a suit that it has at least majority high cards in and try to keep the overall tricks it needs to win around the the amount of cards it has in the trump suit. It will make use of its ally's bids to try and predict what its ally might have.

For the trick round the AI will make use of a fairly small game tree to find some basic legal game play sequences. The game play sequences that will be generated will be limited to a min-max approach. This approach is detailed in [5].

All definitions and preliminary design specifications were taken from [6].

6.1 UML CLASS DIAGRAMS

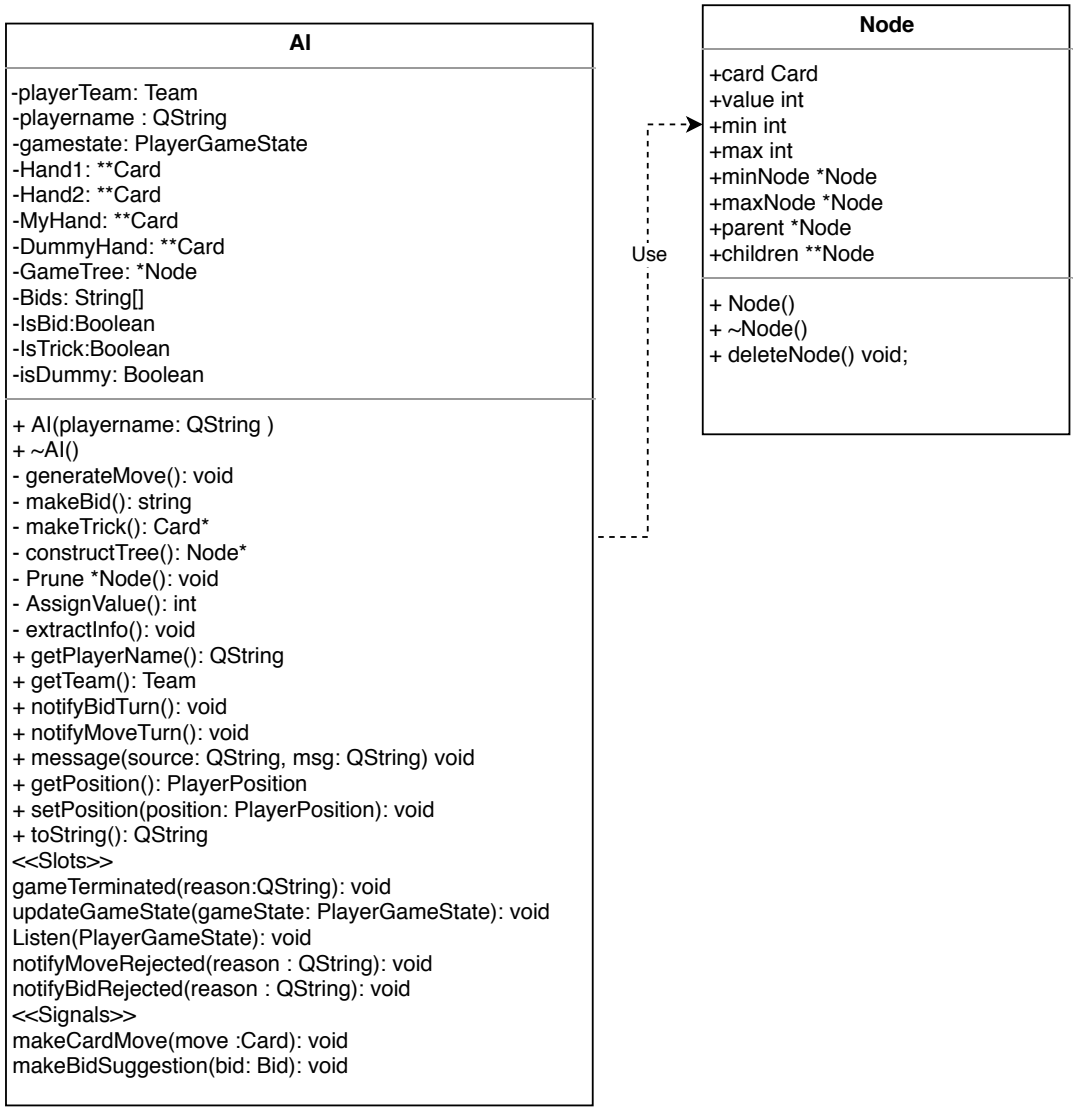


Figure 12: AI class diagram.

6.2 SIGNALS AND SLOTS

Signals	Slots [Class]	Purpose
makeCardMove (card: Card)	moveSelected [GameServer]	Sending a request to make the suggested move to the Game Server.
makeBidSuggestion (bid: Bid)	bidSelected [GameServer]	Sending a request to make the suggested bid to the Game Server.

Table 12: AI's signals

6.3 UNIT TESTS

Number	Tested component	Input	Expected output
1	Receiving game state update.	Slot for receiving game state updates.	QDebug showing the information extracted from the received QJsonObject.
2	Making a logical suggested move.	Different arrays for the dummy hand and AI hand, as well as different trick states.	A suggested move that is legal dummy hand and AI hand and makes sense. It shouldn't repeat the same move for every state.
3	Sending the suggested move.	Random QJsonObject containing the legal move.	The signal to make a move must be sent and the corresponding slot in the game server must receive it.
4	Generating the game tree.	GameState object received from the server.	QDebug prints out the BFS tree.

Table 13: Unit tests for the AI class.

7 SPRINT BACKLOG AND BURNDOWN CHART

The duration of a sprint was decided to be 1 week long.

The tasks for each sprint and the estimated time that it will take to complete is shown in the Sprint Backlog table. The Burndown chart in Figure 13 show the remaining effort at the end of each week/sprint.

Task name	**Responsible**	**Description**	**Duration (Hours)**
Sprint 1 (21 - 25 Sep)			*41*
Logger and skeleton Network	C.F. Wagner	Complete the Logger class. Create the skeleton of the Network classes.	7
Asset creation	H. vd Westhuizen	All the assists for the cards, board and windows should be finished.	12
Fundamental GameServer classes and enumerations	C.H. Conroy	Create all enumerations. Complete Bid, CardSet and Card classes. Create skeleton for GameState, PlayerGameState and Player classes.	12
Gameplay functionality	M. Winnan	Implement the node class and generate a usable game tree. Further reasonable bids must be made.	10
Sprint 2 (28 Sep - 2 Oct)			*37*
Connection, QJasonObject transmission and login.	C.F. Wagner	Add functionality for server and client to connect, exchange data in a QJsonObject format and a client to log into the server.	10
Animation and relay.	H. vd Westhuizen	The GUI should be able to receive the game state object and update the GUI as needed.	8
Core GameServer classes	C.H. Conroy	Complete GameState, ServerGameState, PlayerGameState and Player classes. Create skeleton for Server and GameServer class.	10
Connection, QJasonObject transmission	M. Winnan	Add functionality for server and AI communication.	9
Sprint 3 (5 - 9 Oct)		*Progress demo on Wednesday, 7 Oct 2020.*	*33*
QJsonObject conversion.	C.F. Wagner	Add functionality to convert to and from the QJsonObject.	10
Sending bid and card moves.	H. vd Westhuizen	Any moves and bids should be sent to server, if the actions are valid then the GUI should update.	6
GameServer functionality	C.H. Conroy	Complete Server and GameServer classes. Create skeleton for Score class.	9
Receiving game state update.	M. Winnan	Add functionality to receive a game state and update current game state.	8

Task name	Responsible	Description	Duration (Hours)
Sprint 4 (12 - 16 Oct)		*Test week 2.*	*31*
Ping and pong.	C.F. Wagner	Add functionality to quickly detect when connection has been lost.	6
Server GUI information relay.	H. vd Westhuizen	The server password and username should be sent to the server and the lobby should be functional.	10
Scoring and refactoring	C.H. Conroy	Complete Score class. Review and refactor all GameServer related classes.	6
QJsonObject conversion	M. Winnan	Add functionality to convert to and from a QJsonObject.	9
Finish individual parts.		*Individual demo on Wednesday, 21 Oct.*	19 - 20 Oct

Table 14: Sprint Backlog.

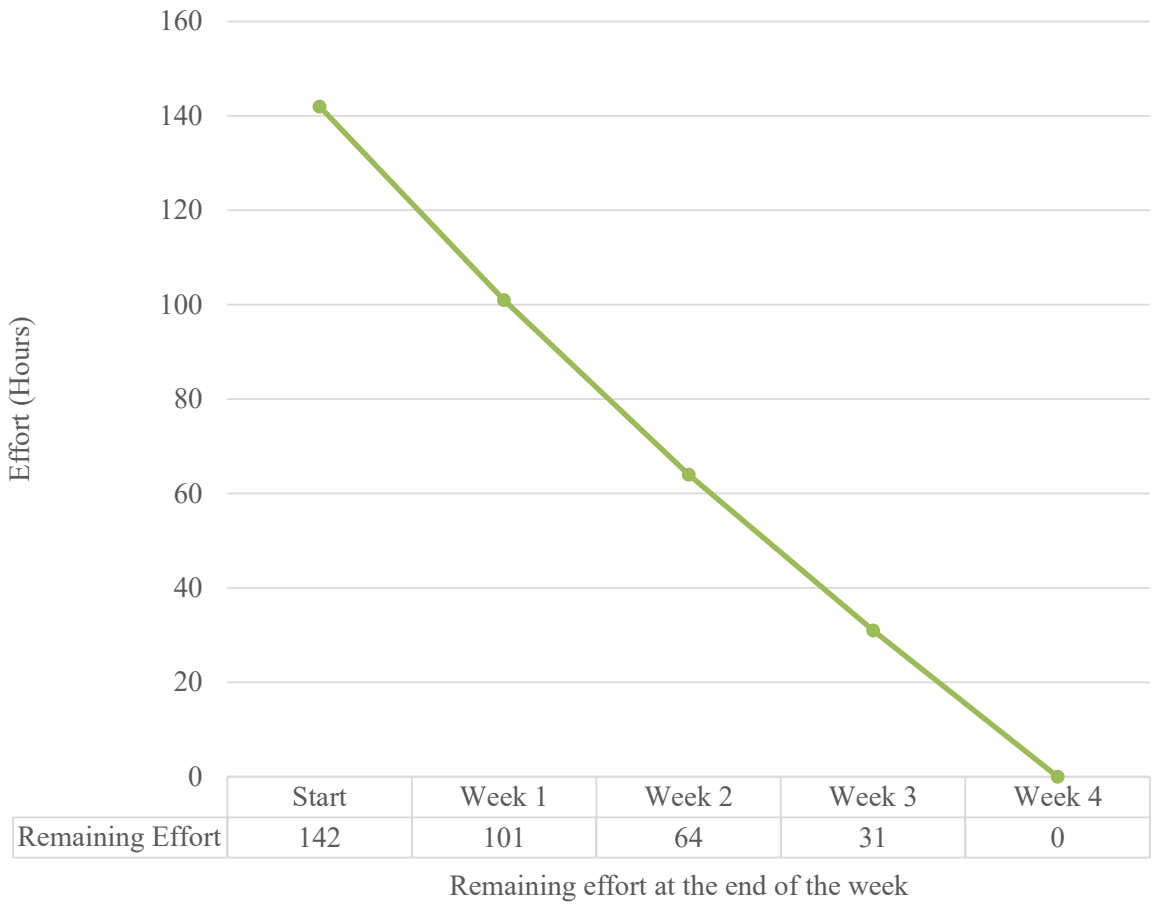


Figure 13: Burndown chart showing the total estimated effort remaining after each week/sprint.