Software Workshop Group Project

Blackjack Game with Chat Client

UNIVERSITY OF BIRMINGHAM

Group Bordeaux

Tom Brereton, Lloyd Carswell,
Alex Davenport, Lois Holman and
Yifan Wu

Introduction

This project explores the development of a concurrent client-server Blackjack card game with a chat client. The aim was to create a system whereby users are able to create an account and sign into their own homepage. Once signed in, the user can browse a series of current games from a lobby screen and choose to either create a new game or join an existing one. Up to four clients are able to join one single game from which they can play Blackjack and communicate their experiences to other players in the same room using a chat client on the side of the screen. Notable features of the project include having multiple private game rooms, a Blackjack card game engine and a custom HUD GUI that adapts to frame resizing.

Statement of Contribution

All team members have agreed that the following percentage of contribution towards the project has been made:

Tom: 24%
Alex: 24%
Lois: 10%
Yifan: 24%
Lloyd: 18%

Team members have also all agreed to Lois' absence for the presentation.

Signed: Tom, Alex, Yifan, Lloyd, Lois

Overview

System Description

The system has been designed using a three tier architecture of client, server, database (Figure 1). This architecture permits playing and chatting remotely across a network within multiple private game rooms. In addition to the client server architecture, the system incorporates a Blackjack card game engine and a Graphical User Interface (GUI) that utilises a Model View Controller format (MVC). The GUI has been designed with using a custom resizing method that repositions components whilst maintaining the desired look and feel of the heads up display (HUD) employed in the game room. The database is persistent and is split into three tables to store the user information, the user's game information and the outcome of each game.

The server handles all client connections through a multithreaded system, it begins by waiting for a client socket connection. Upon connection, the server creates a new thread for the connected client, this new thread will then handle all the requests from the client and send appropriate responses. Additional clients that try to connect will be given a new thread to handle their requests. Such requests include, but are not limited to, requests to login, create account or join a game. The server consists of methods that send and receive information from multiple clients who are accessing the same game room, such information includes messages and game moves. In addition, the server contains a game engine which decides what game data to send dependent on the requests from the client such as using "hit" to receive another card or "stand" to stick with their hand. Implementing the game logic server side allows for the game to take place in one single location and avoids client to client dependency.

On the client side, a user initialises their connection by running the main GUI class, BlackjackOnline, which begins the client and connects to the server via a socket. Once a connection is established, the user is then able to use the buttons through the GUI to invoke action listeners that in-turn send requests from their client to the server. An appropriate response is then sent back from the server. Upon logging in a thread is spawned to continually poll (send requests) for the list of game names shown in the lobby. Moreover, this thread is terminated and two other threads are spawned when joining or creating a game. One of these threads poll for messages sent in the game screen, the other polls for game data. When leaving, these two threads terminate and the game name threads starts again. To avoid conflicts with these threads, including the main thread, the request methods are designed to always receive a response in an atomic fashion (synchronized). This is to avoid thread interference and ensure memory consistency.

The database is controlled through a FunctionsDB class that contains all SQL queries, these are used to update and reference the database. Upon starting the game a user is required to create an account before logging in. Every account is required to have a unique username which is handled by referencing the user information database. After sign up the user can then login to their personal homepage from which data is pulled from the database showing the user's current statistics. When a user leaves a game of Blackjack the database is updated with their new statistics.

From the homepage a user can choose to go the the lobby. The lobby displays all the available games that the user can join, this is stored as a list server side with updates being sent to client threads continuously (polling). This same method is used to receive messages within the game room. The user can also choose to join or create a new game, his unique username is used as the name on the list of games. In the game room the user can play Blackjack with the ability for other users to join in and play the game.

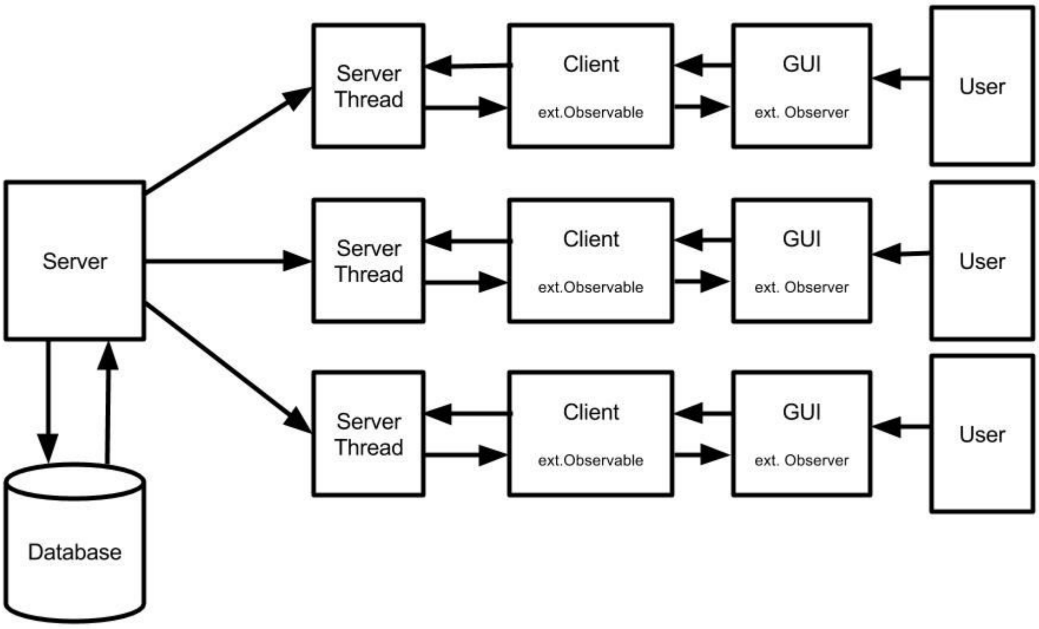


Figure 1: System Architecture

Requirements

A series of functional and nonfunctional requirements were established for the project and are broken down into what the system must include, should include and could include leaving room for future expansion.

Functional Requirements

The system must

- The system must contain a database containing the first names, last names, usernames, passwords and other relevant fields required for user authentication.
- The system must be built around a client-server architecture where users can interact with the system from a remote client application.
- The client must communicate with the server via requests and responses.
- The system must enable users to create accounts and log in from the client.
- The system must enable users to play blackjack with other users.
- The system must enable users to chat with other players in a dedicated chat-room.
- The system must contain a game lobby to enable users to create and join blackjack games on the server..
- The system must allot a new user with a fixed amount of funds for playing blackjack.
- Message boxes must appear if a client request fails with the reason it failed.

The system should

- The system should store a database of user statistics pertaining to the user's blackjack performance.
- The system should allot more funds to a user if their respective funds drop below a set threshold.

The system could

- The system could enable users to reset passwords.
- The system could enable users to recover their passwords via an email address stored in the database.
- The system could enable users to select a different avatar.
- The system could enable users to update their personal details.
- The lobby could show the number of players currently in the room.

Non-Functional Requirements

The system must

- The client must be developed in Java using Swing components.
- The server must be developed in Java.
- The system development must be managed via version control.
- The system must be capable of running at least one or more blackjack games at a time.
- The server must store all passwords as a shar2 hash of the password.
- The user's passwords must be hashed before being sent to the server.
- All communication between the server and client must adhere to a strict protocol.
- The client must request updates to game status, message logs and currently active games at a fixed frequency.

The system should

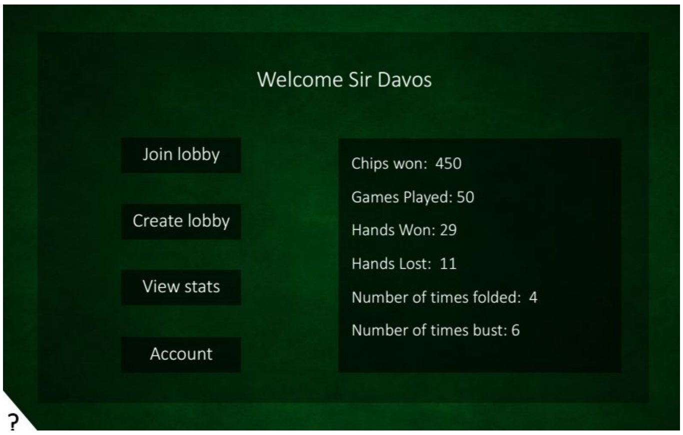
- All source code should be well documented with comments and Java documentation.
- All classes, variables and packages should follow standard Java naming convention.
- All code blocks should be indented with white space.
- All game data updates should occur at a time interval less than 5 seconds.
- The client application should handle all potential error types appropriately.
- The graphical user interface must be aesthetically pleasing and easy to navigate without prior instruction.

The system could

- All graphical interfaces could enable greater access to users with additional needs through adhering to Java's accessibility standards.
- Moderation features could be implemented to manage inappropriate user behaviour.
- Help pages could be contained in the client application including documentation on how to use the system, the rules of blackjack and troubleshooting common errors.

Prototyping

Upon deciding the concept of a concurrent Blackjack card game, initial prototypes were drawn up to help visualise the final product. Building on the requirements, the prototype concepts provide a goal to aim for the project. The game would contain a homescreen (Figure 2) personal to the user with statistics being displayed that were drawn from the database using sql queries.

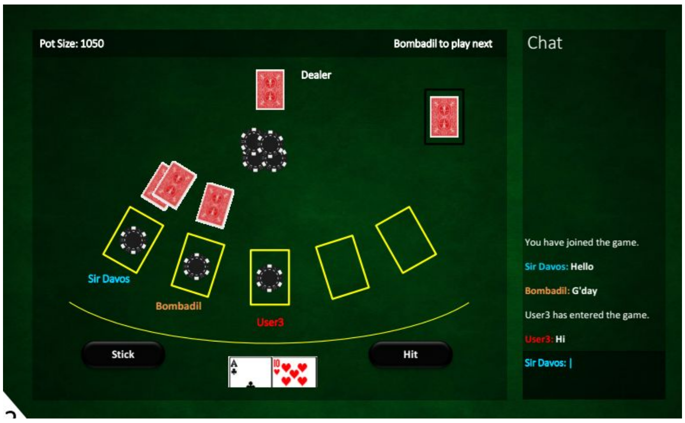


Welcome Sir Davos

Chips won: 450
Games Played: 50
Hands Won: 29
Hands Lost: 11
Number of times folded: 4
Number of times bust: 6

Figure 2: Prototype Home Screen

After joining a game from the lobby, multiple users can play blackjack within the same game room (Figure 3). Users can chat using the chat box on the side of the screen whilst playing blackjack together against the dealer. The username of each player would appear next to their respective position. This prototype would be referred to most when creating the GUI layout for the project in addition to the design of the game engine and how the two interact.



Pot Size: 1050
Bombadil to play next

You have joined the game.
Sir Davos: Hello
Bombadil: G'day
User3 has entered the game.
User3: Hi
Sir Davos: |

Figure 3: Prototype Game Screen

System Design

Whole System

The system has been designed as a three tiered client server based architecture, with a database, server, and client.

Initially, the GUI, client, server and database were implemented and tested separately until the system was ready for a user to register and sign in. At this point, the system was connected to ensure the preliminary design of the system and communication worked. After this, the GUI, client, server, and database were implemented and tested independently, using a combination of functional tests for the client side and unit tests for the server side.

To use the system, the user interacts with the GUI and upon certain button events, the client sends a request to the server. The server ‘handles’ this request and sends an appropriate response based upon the server actions. For example, a bet request will be sent to the server, the server will place a bet for the player in the game and change their budget, a response is then sent back to the client.

For certain actions, the database is access via the server. For example, logging in tries to retrieve this user sent in the request and returns success if it matches, false if not.

A thin client design has been implemented, therefore the server processes the request and handles all the logic. The client simply stores and updates data as per the responses which the GUI uses to decide what element to display (Observer pattern) e.g. cards, players, games.

The communication protocol is designed so that a java object is serialised into a json string. This json string is sent through a socket and deserialized at the other end as a super class which includes the object type. Once the object type is known, the json string is deserialized again as the correct subclass which contains relevant information for the object type.

Use Case Diagram

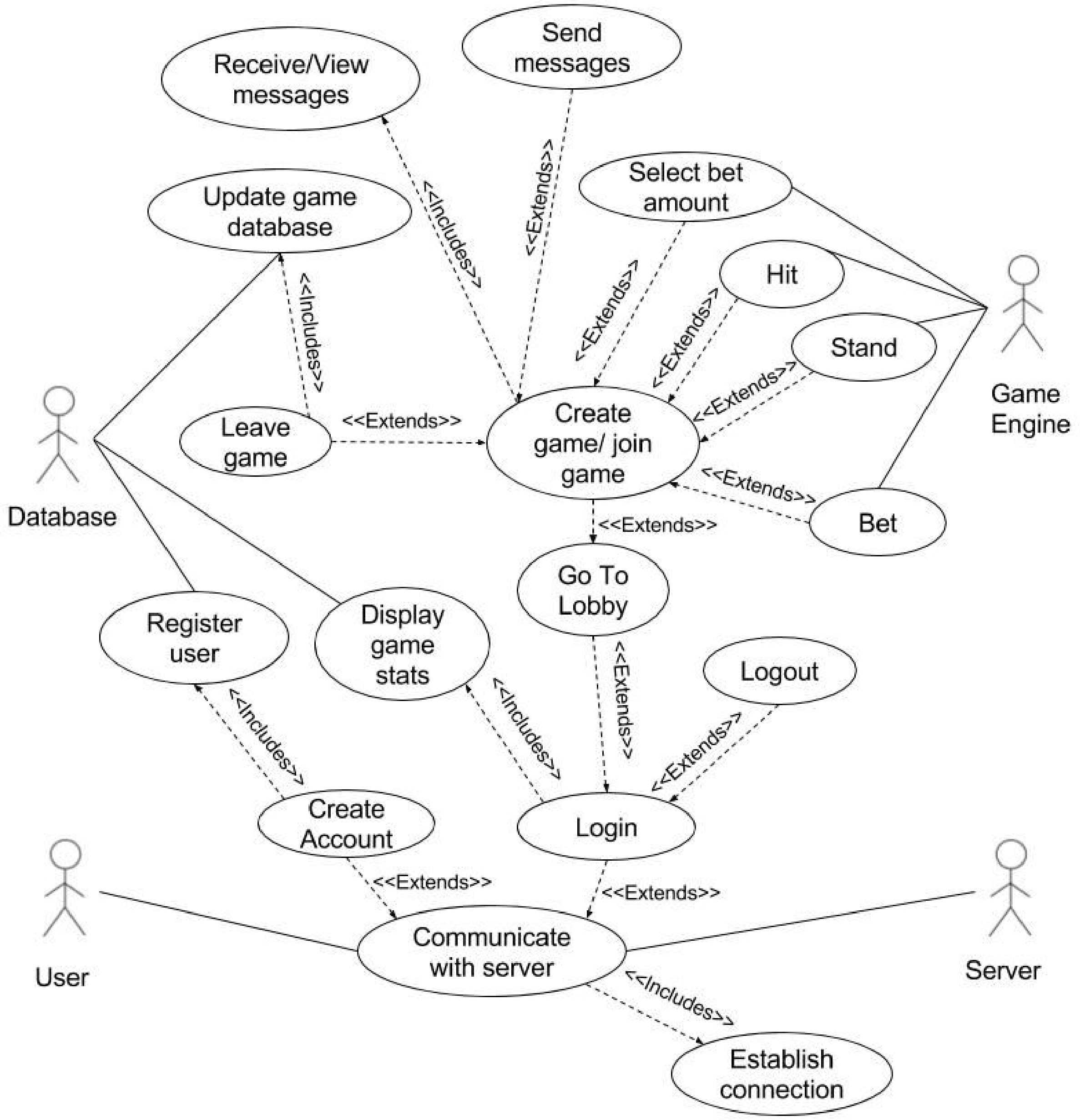


Figure 4 Use Case Diagram

This use case diagram (Figure 4) visually describes the overview of the whole system. There are four actors: User, Server, Database, and Game Engine. Initially a user starts the Blackjack game which then connects to the server. After establishing a connection the user client and server continually communicate with each other throughout all use cases that extend from that communication. When a user logs in, registers, or leaves a game the database is required to be accessed to either be updated or to reference information.

Database

The database is split into three sections: users, users_games and game_outcomes. There a one to many relationship for user to user games. There is also a one to many relationship for game outcomes to user games. As seen in Figure 5, the database is built from 3 separate tables, linked by userID and outcomeID. User data for signup and login are stored in the users table, with gameplay data logged into user_games and game_outcomes.

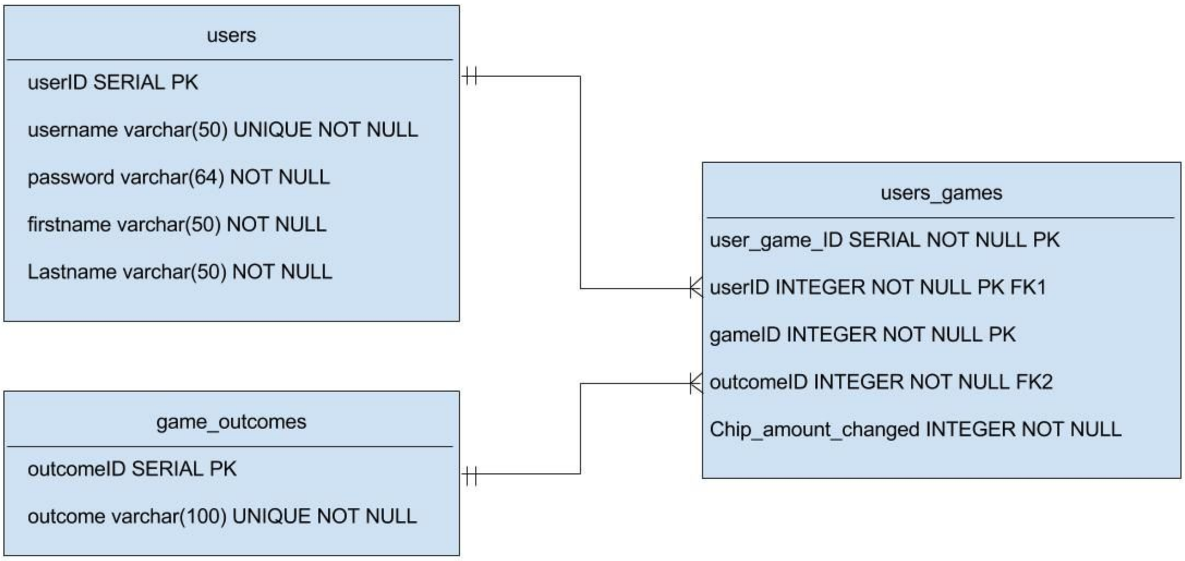

users
userID SERIAL PK
username varchar(50) UNIQUE NOT NULL
password varchar(64) NOT NULL
firstname varchar(50) NOT NULL
Lastname varchar(50) NOT NULL
game_outcomes
outcomeID SERIAL PK
outcome varchar(100) UNIQUE NOT NULL
users_games
user_game_ID SERIAL NOT NULL PK
userID INTEGER NOT NULL PK FK1
gameID INTEGER NOT NULL PK
outcomeID INTEGER NOT NULL FK2
Chip_amount_changed INTEGER NOT NULL

Figure 5: Entity Relationship Diagram

For security we hash the the password in the client using Sha256 before sending the request. This hashed password is stored in the database and retrieved for login requests. If the hashed password from the database and the request match, the user is logged into the client and server thread. The hashing is one directional, therefore visibility of the hashed password does not compromise security.

Server

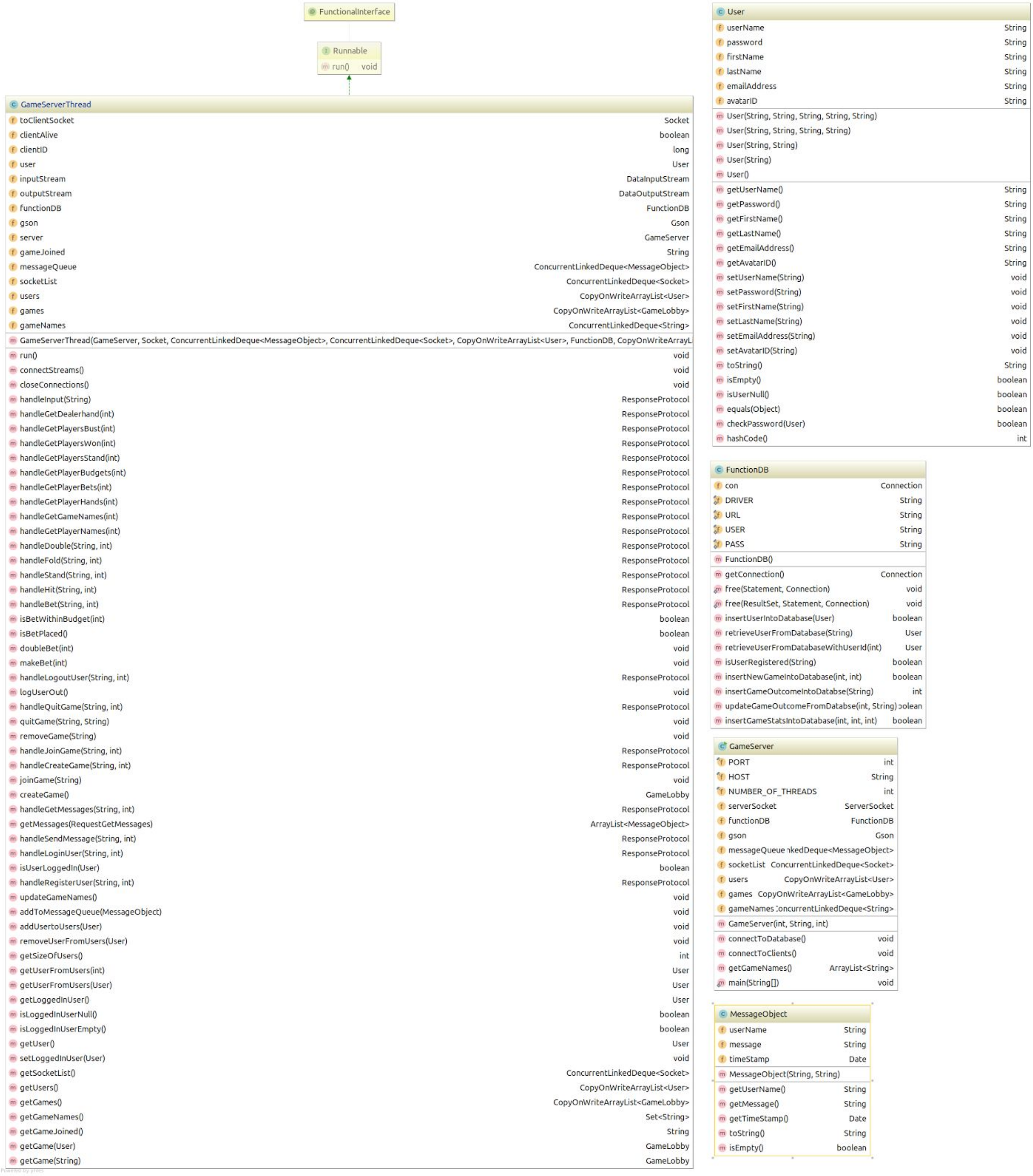
The server is split into four main sections; (1) an overall server which wait for client connection, (2) a server thread which handles client communication, (3) the game lobby and game logic, (4) and database interaction.

The server creates a thread pool of arbitrary size then waits for a client to connect. Upon connection, the socket is passed into a thread, which now handles all the communication and logic for that client. All data shared between clients are instantiated on the server and passed to each new thread as clients connect, this allows any changes made by a client to be communicated to all other clients.

This multithreaded server also handles all the gamelogic, game lobby's, user connected, database interaction, and general logic (e.g. user registration). The server contains a list of game lobby's, which enables enables users to have private game. A game lobby (read game) is added to the list if a create game request is sent. Up to three players can join a game and if no players are in a game, it is removed.

Each game lobby also contains its own message queue, which allows the players to have a private chat respective to the game they are in.

Server Class Diagram

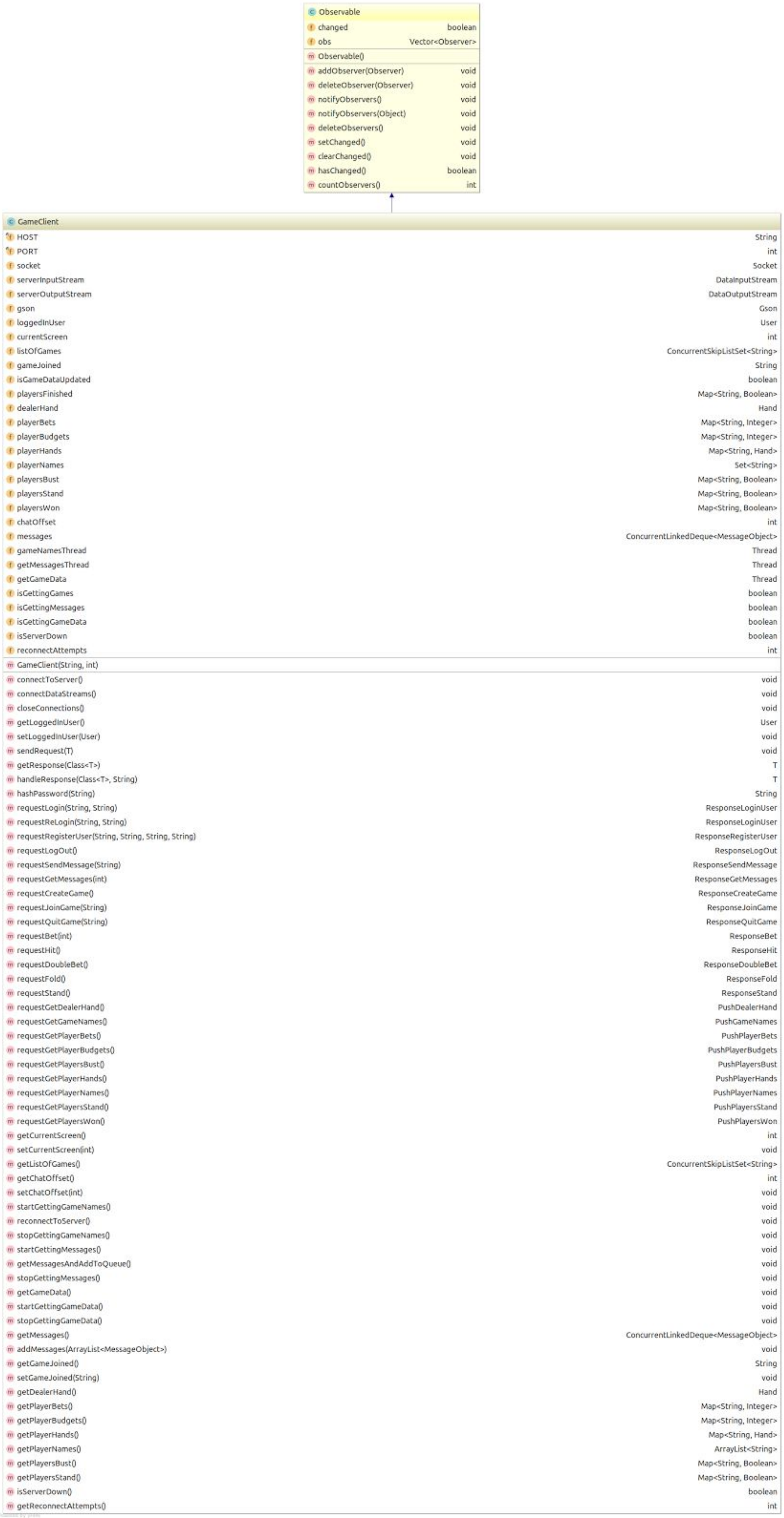

FunctionalInterface
Runnable
run() void
GameServerThread
toClientSocket Socket
clientAlive boolean
clientID long
user User
inputStream DataInputStream
outputStream DataOutputStream
functionDB FunctionDB
gson Gson
server GameServer
gameJoined String
messageQueue ConcurrentLinkedDeque<MessageObject>
socketList ConcurrentLinkedDeque<Socket>
users CopyOnWriteArrayList<User>
games CopyOnWriteArrayList<GameLobby>
gameNames ConcurrentLinkedDeque<String>
GameServerThread(GameServer, Socket, ConcurrentLinkedDeque<MessageObject>, ConcurrentLinkedDeque<Socket>, CopyOnWriteArrayList<User>, FunctionDB, CopyOnWriteArrayL
run() void
connectStreams() void
closeConnections() void
handleInput(String) ResponseProtocol
handleGetDealerhand(int) ResponseProtocol
handleGetPlayersBust(int) ResponseProtocol
handleGetPlayersWon(int) ResponseProtocol
handleGetPlayersStand(int) ResponseProtocol
handleGetPlayerBudgets(int) ResponseProtocol
handleGetPlayerBets(int) ResponseProtocol
handleGetPlayerHands(int) ResponseProtocol
handleGetGameNames(int) ResponseProtocol
handleGetPlayerNames(int) ResponseProtocol
handleDouble(String, int) ResponseProtocol
handleFold(String, int) ResponseProtocol
handleStand(String, int) ResponseProtocol
handleHit(String, int) ResponseProtocol
handleBet(String, int) ResponseProtocol
isBetWithinBudget(int) boolean
isBetPlaced() boolean
doubleBet(int) void
makeBet(int) void
handleLogoutUser(String, int) ResponseProtocol
logUserOut() void
handleQuitGame(String, int) ResponseProtocol
quitGame(String, String) void
removeGame(String) void
handleJoinGame(String, int) ResponseProtocol
handleCreateGame(String, int) ResponseProtocol
joinGame(String) void
createGame() GameLobby
handleGetMessages(String, int) ResponseProtocol
getMessages(RequestGetMessages) ArrayList<MessageObject>
handleSendMessage(String, int) ResponseProtocol
handleLoginUser(String, int) ResponseProtocol
isUserLoggedIn(User) boolean
handleRegisterUser(String, int) ResponseProtocol
updateGameNames() void
addToMessageQueue(MessageObject) void
addUsertoUsers(User) void
removeUserFromUsers(User) void
getSizeOfUsers() int
getUserFromUsers(int) User
getUserFromUsers(User) User
getLoggedInUser() User
isLoggedInUserNull() boolean
isLoggedInUserEmpty() boolean
getUser() User
setLoggedInUser(User) void
getSocketList() ConcurrentLinkedDeque<Socket>
getUsers() CopyOnWriteArrayList<User>
getGames() CopyOnWriteArrayList<GameLobby>
getGameNames() Set<String>
getGameJoined() String
getGame(User) GameLobby
getGame(String) GameLobby
Powered by yFiles
User
userName String
password String
firstName String
lastName String
emailAddress String
avatarID String
User(String, String, String, String, String)
User(String, String, String, String)
User(String, String)
User(String)
User()
getUserName() String
getPassword() String
getFirstName() String
getLastName() String
getEmailAddress() String
getAvatarID() String
setUserName(String) void
setPassword(String) void
setFirstName(String) void
setLastName(String) void
setEmailAddress(String) void
setAvatarID(String) void
toString() String
isEmpty() boolean
isUserNull() boolean
equals(Object) boolean
checkPassword(User) boolean
hashCode() int
FunctionDB
con Connection
DRIVER String
URL String
USER String
PASS String
FunctionDB()
getConnection() Connection
free(Statement, Connection) void
free(ResultSet, Statement, Connection) void
insertUserIntoDatabase(User) boolean
retrieveUserFromDatabase(String) User
retrieveUserFromDatabaseWithUserId(int) User
isUserRegistered(String) boolean
insertNewGameIntoDatabase(int, int) boolean
insertGameOutcomeIntoDatabse(String) int
updateGameOutcomeFromDatabse(int, String) ɔolean
insertGameStatsIntoDatabase(int, int, int) boolean
GameServer
PORT int
HOST String
NUMBER_OF_THREADS int
serverSocket ServerSocket
functionDB FunctionDB
gson Gson
messageQueue ıkedDeque<MessageObject>
socketList ConcurrentLinkedDeque<Socket>
users CopyOnWriteArrayList<User>
games CopyOnWriteArrayList<GameLobby>
gameNames :oncurrentLinkedDeque<String>
GameServer(int, String, int)
connectToDatabase() void
connectToClients() void
getGameNames() ArrayList<String>
main(String[]) void
MessageObject
userName String
message String
timeStamp Date
MessageObject(String, String)
getUserName() String
getMessage() String
getTimeStamp() Date
toString() String
isEmpty() boolean

Client

The GameClient class facilitates the client's communication with the server through a socket, stores all relevant information for the client's current state and enables the GUI to send requests and observe its current field values through notifications sent by extending the observable class.

The client is based on a model-view-controller (MVC) architecture where the gui components are separated from the data models and controllers. The GameClient class acts as the client model in this paradigm by extending the observable class and enabling notifications of updates to its observers, i.e the GUI. The workflow of a request is as follows: the class contains methods that send requests over a socket to the server, waits for responses, updates the classes fields and notifies its observers. The observers can access the field variables through getter methods. Requests can be sent by actions being triggered in the GUI or by one of the internally defined threads that request for certain information at a given frequency. When logged in, the class will spawn a thread to ask for updates to the current list of available games. Whilst in a game, the previous thread will stop and two threads will spawn to update the chat log and the game data respectively.

Client Class Diagram


Observable
changed boolean
obs Vector<Observer>
Observable()
addObserver(Observer) void
deleteObserver(Observer) void
notifyObservers() void
notifyObservers(Object) void
deleteObservers() void
setChanged() void
clearChanged() void
hasChanged() boolean
countObservers() int
GameClient
HOST String
PORT int
socket Socket
serverInputStream DataInputStream
serverOutputStream DataOutputStream
gson Gson
loggedInUser User
currentScreen int
listOfGames ConcurrentSkipListSet<String>
gameJoined String
isGameDataUpdated boolean
playersFinished Map<String, Boolean>
dealerHand Hand
playerBets Map<String, Integer>
playerBudgets Map<String, Integer>
playerHands Map<String, Hand>
playerNames Set<String>
playersBust Map<String, Boolean>
playersStand Map<String, Boolean>
playersWon Map<String, Boolean>
chatOffset int
messages ConcurrentLinkedDeque<MessageObject>
gameNamesThread Thread
getMessagesThread Thread
getGameData Thread
isGettingGames boolean
isGettingMessages boolean
isGettingGameData boolean
isServerDown boolean
reconnectAttempts int
GameClient(String, int)
connectToServer() void
connectDataStreams() void
closeConnections() void
getLoggedInUser() User
setLoggedInUser(User) void
sendRequest(T) void
getResponse(Class<T>) T
handleResponse(Class<T>, String) T
hashPassword(String) String
requestLogin(String, String) ResponseLoginUser
requestReLogin(String, String) ResponseLoginUser
requestRegisterUser(String, String, String, String) ResponseRegisterUser
requestLogOut() ResponseLogOut
requestSendMessage(String) ResponseSendMessage
requestGetMessages(int) ResponseGetMessages
requestCreateGame() ResponseCreateGame
requestJoinGame(String) ResponseJoinGame
requestQuitGame(String) ResponseQuitGame
requestBet(int) ResponseBet
requestHit() ResponseHit
requestDoubleBet() ResponseDoubleBet
requestFold() ResponseFold
requestStand() ResponseStand
requestGetDealerHand() PushDealerHand
requestGetGameNames() PushGameNames
requestGetPlayerBets() PushPlayerBets
requestGetPlayerBudgets() PushPlayerBudgets
requestGetPlayersBust() PushPlayersBust
requestGetPlayerHands() PushPlayerHands
requestGetPlayerNames() PushPlayerNames
requestGetPlayersStand() PushPlayersStand
requestGetPlayersWon() PushPlayersWon
getCurrentScreen() int
setCurrentScreen(int) void
getListOfGames() ConcurrentSkipListSet<String>
getChatOffset() int
setChatOffset(int) void
startGettingGameNames() void
reconnectToServer() void
stopGettingGameNames() void
startGettingMessages() void
getMessagesAndAddToQueue() void
stopGettingMessages() void
getGameData() void
startGettingGameData() void
stopGettingGameData() void
getMessages() ConcurrentLinkedDeque<MessageObject>
addMessages(ArrayList<MessageObject>) void
getGameJoined() String
setGameJoined(String) void
getDealerHand() Hand
getPlayerBets() Map<String, Integer>
getPlayerBudgets() Map<String, Integer>
getPlayerHands() Map<String, Hand>
getPlayerNames() ArrayList<String>
getPlayersBust() Map<String, Boolean>
getPlayersStand() Map<String, Boolean>
isServerDown() boolean
getReconnectAttempts() int

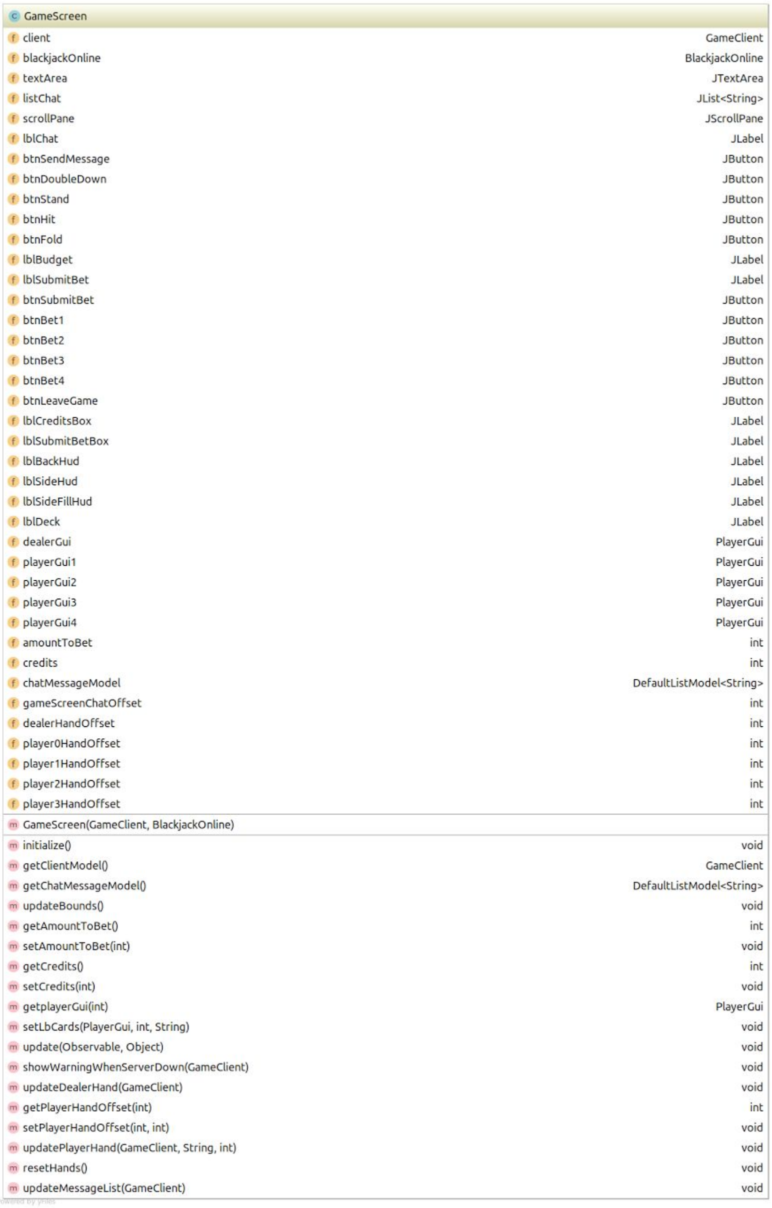
Graphical User Interface

The GUI is the contact point between the server system and the client. It presents a visual interface with which the client can interact and send requests through to the server using JButtons and their corresponding actionListener methods. When the actionListener event is called a request from the client model is then passed to the server which depending on the scenario will return an appropriate response. Our system uses an observer pattern where the client is observable and screens are observers. When the client updates the data the observers are notified and use the observer's pull method to retrieve relevant data to the observable.

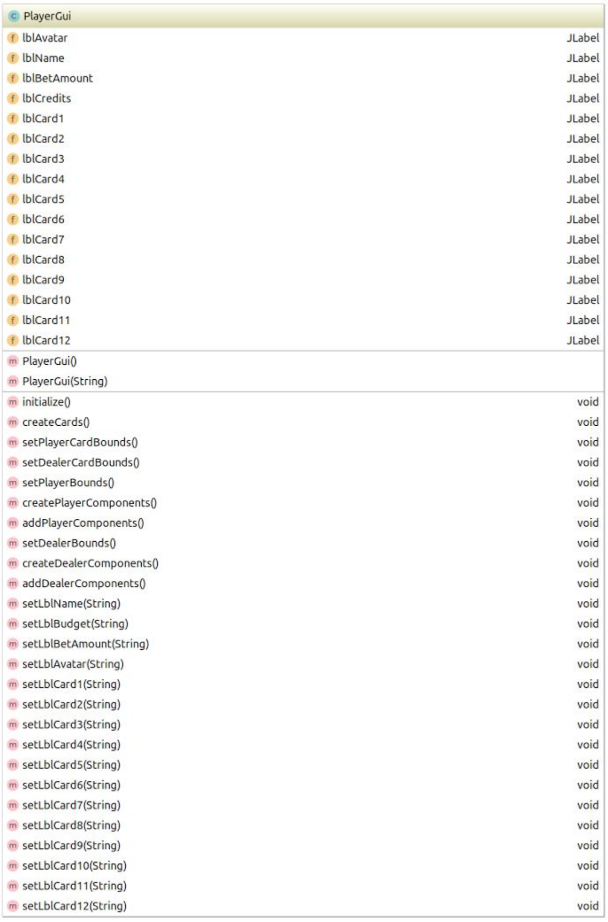
The GUI consists of a main client class BlackjackOnline which contains a main method that attempts to connect to a server at runtime. This class extends a Jframe and implements both observable and component listeners. The class takes a panel from one of the screen classes that extend JPanel such as LoginScreen and CreateAccountScreen. A componentListener is used to check for any changes in the screen size and makes use of an update bounds method that is called everytime a screen resize listener is instigated. The bounds of the components are updated according to where they should be positioned on the screen. Components can be fixed to the screen edges, center or any position in between. Distance between or the size of components can expand and contract depending on which variable is being referenced. This method is used most notably in the GameScreen where the chat window and HUD panel expands whilst keeping the chat window to a reasonable size. The buttons are also kept at the same scale, their position is however changed depending on how the screen is modified.

All graphical assets are stored in a folder called resources, with sub folders for cards, avatars, and game hud. Card images are given a 3 digit string with which they can be accessed using a get card image method. This method is based off the integer values used to determine the suit and number in the card object. The first digit identifies the suit "1-4", the second and third are used to determine the value from "01"-"13" giving values of Ace through to King (for example "105" would call the 5 of Spades). The String "000" displays a facedown card. Avatars are done in a similar manner. Users are assigned a random avatarID upon signing up "2"-"40". Within the game room, empty placeholder player positions are created with a "0" avatarID which appears as an empty chair. The dealer is set to an avatarID of "1".

GUI Class Diagram 1

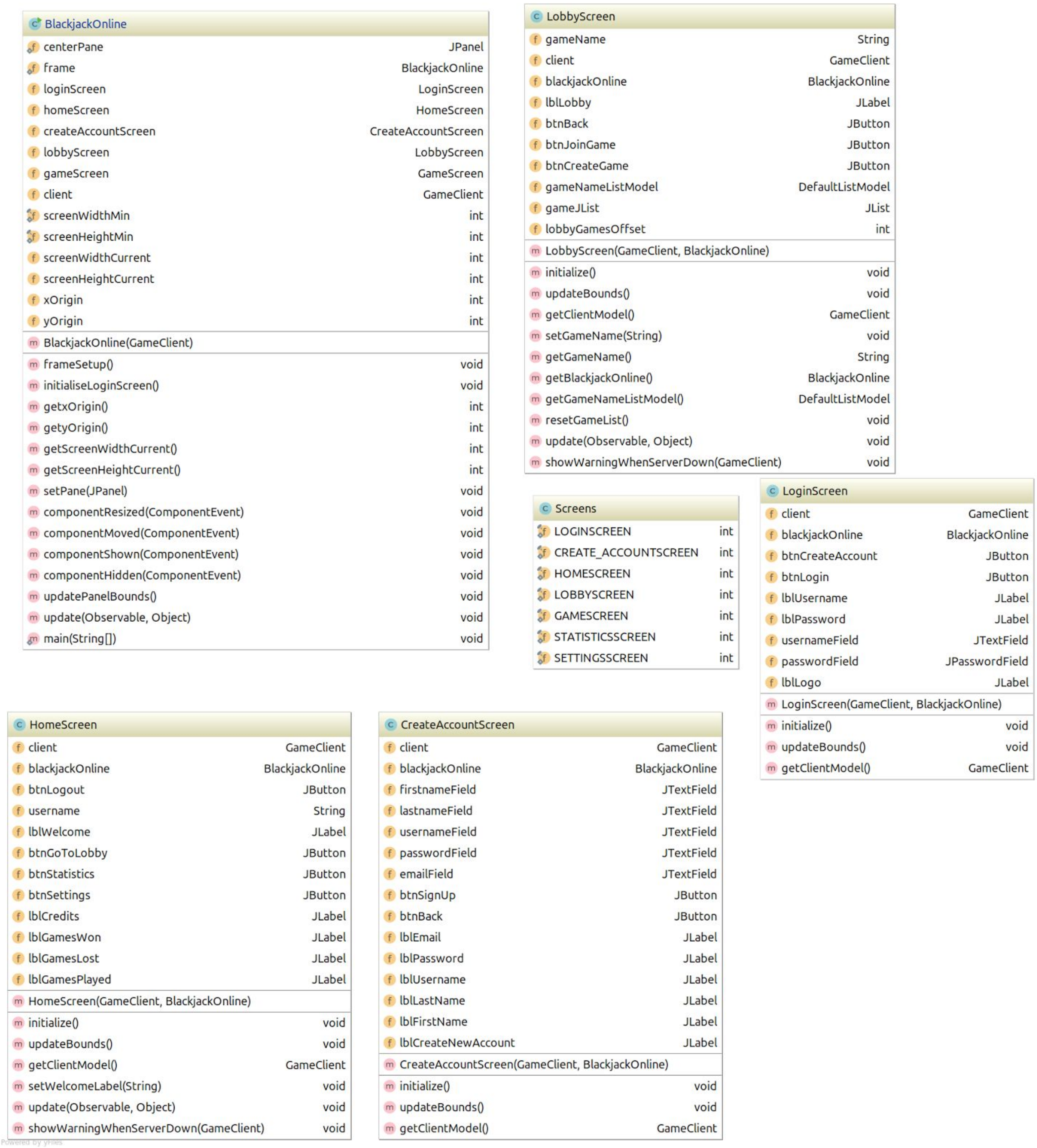


	GameScreen	
f	client	GameClient
f	blackjackOnline	BlackjackOnline
f	textArea	JTextArea
f	listChat	JList<String>
f	scrollPane	JScrollPane
f	lblChat	JLabel
f	btnSendMessage	JButton
f	btnDoubleDown	JButton
f	btnStand	JButton
f	btnHit	JButton
f	btnFold	JButton
f	lblBudget	JLabel
f	lblSubmitBet	JLabel
f	btnSubmitBet	JButton
f	btnBet1	JButton
f	btnBet2	JButton
f	btnBet3	JButton
f	btnBet4	JButton
f	btnLeaveGame	JButton
f	lblCreditsBox	JLabel
f	lblSubmitBetBox	JLabel
f	lblBackHud	JLabel
f	lblSideHud	JLabel
f	lblSideFillHud	JLabel
f	lblDeck	JLabel
f	dealerGui	PlayerGui
f	playerGui1	PlayerGui
f	playerGui2	PlayerGui
f	playerGui3	PlayerGui
f	playerGui4	PlayerGui
f	amountToBet	int
f	credits	int
f	chatMessageModel	DefaultListModel<String>
f	gameScreenChatOffset	int
f	dealerHandOffset	int
f	player0HandOffset	int
f	player1HandOffset	int
f	player2HandOffset	int
f	player3HandOffset	int
m	GameScreen(GameClient, BlackjackOnline)	
m	initialize()	void
m	getClientModel()	GameClient
m	getChatMessageModel()	DefaultListModel<String>
m	updateBounds()	void
m	getAmountToBet()	int
m	setAmountToBet(int)	void
m	getCredits()	int
m	setCredits(int)	void
m	getplayerGui(int)	PlayerGui
m	setLbCards(PlayerGui, int, String)	void
m	update(Observable, Object)	void
m	showWarningWhenServerDown(GameClient)	void
m	updateDealerHand(GameClient)	void
m	getPlayerHandOffset(int)	int
m	setPlayerHandOffset(int, int)	void
m	updatePlayerHand(GameClient, String, int)	void
m	resetHands()	void
m	updateMessageList(GameClient)	void



	PlayerGui	
f	lblAvatar	JLabel
f	lblName	JLabel
f	lblBetAmount	JLabel
f	lblCredits	JLabel
f	lblCard1	JLabel
f	lblCard2	JLabel
f	lblCard3	JLabel
f	lblCard4	JLabel
f	lblCard5	JLabel
f	lblCard6	JLabel
f	lblCard7	JLabel
f	lblCard8	JLabel
f	lblCard9	JLabel
f	lblCard10	JLabel
f	lblCard11	JLabel
f	lblCard12	JLabel
m	PlayerGui()	
m	PlayerGui(String)	
m	initialize()	void
m	createCards()	void
m	setPlayerCardBounds()	void
m	setDealerCardBounds()	void
m	setPlayerBounds()	void
m	createPlayerComponents()	void
m	addPlayerComponents()	void
m	setDealerBounds()	void
m	createDealerComponents()	void
m	addDealerComponents()	void
m	setLblName(String)	void
m	setLblBudget(String)	void
m	setLblBetAmount(String)	void
m	setLblAvatar(String)	void
m	setLblCard1(String)	void
m	setLblCard2(String)	void
m	setLblCard3(String)	void
m	setLblCard4(String)	void
m	setLblCard5(String)	void
m	setLblCard6(String)	void
m	setLblCard7(String)	void
m	setLblCard8(String)	void
m	setLblCard9(String)	void
m	setLblCard10(String)	void
m	setLblCard11(String)	void
m	setLblCard12(String)	void

GUI Class Diagram 2



c	BlackjackOnline	
f	centerPane	JPanel
f	frame	BlackjackOnline
f	loginScreen	LoginScreen
f	homeScreen	HomeScreen
f	createAccountScreen	CreateAccountScreen
f	lobbyScreen	LobbyScreen
f	gameScreen	GameScreen
f	client	GameClient
f	screenWidthMin	int
f	screenHeightMin	int
f	screenWidthCurrent	int
f	screenHeightCurrent	int
f	xOrigin	int
f	yOrigin	int
m	BlackjackOnline(GameClient)	
m	frameSetup()	void
m	initialiseLoginScreen()	void
m	getxOrigin()	int
m	getyOrigin()	int
m	getScreenWidthCurrent()	int
m	getScreenHeightCurrent()	int
m	setPane(JPanel)	void
m	componentResized(ComponentEvent)	void
m	componentMoved(ComponentEvent)	void
m	componentShown(ComponentEvent)	void
m	componentHidden(ComponentEvent)	void
m	updatePanelBounds()	void
m	update(Observable, Object)	void
m	main(String[])	void

c	LobbyScreen	
f	gameName	String
f	client	GameClient
f	blackjackOnline	BlackjackOnline
f	lblLobby	JLabel
f	btnBack	JButton
f	btnJoinGame	JButton
f	btnCreateGame	JButton
f	gameNameListModel	DefaultListModel
f	gameJList	JList
f	lobbyGamesOffset	int
m	LobbyScreen(GameClient, BlackjackOnline)	
m	initialize()	void
m	updateBounds()	void
m	getClientModel()	GameClient
m	setGameName(String)	void
m	getGameName()	String
m	getBlackjackOnline()	BlackjackOnline
m	getGameNameListModel()	DefaultListModel
m	resetGameList()	void
m	update(Observable, Object)	void
m	showWarningWhenServerDown(GameClient)	void

c	Screens	
f	LOGINSCREEN	int
f	CREATE_ACCOUNTSCREEN	int
f	HOMESCREEN	int
f	LOBBYSCREEN	int
f	GAMESCREEN	int
f	STATISTICSSCREEN	int
f	SETTINGSSCREEN	int

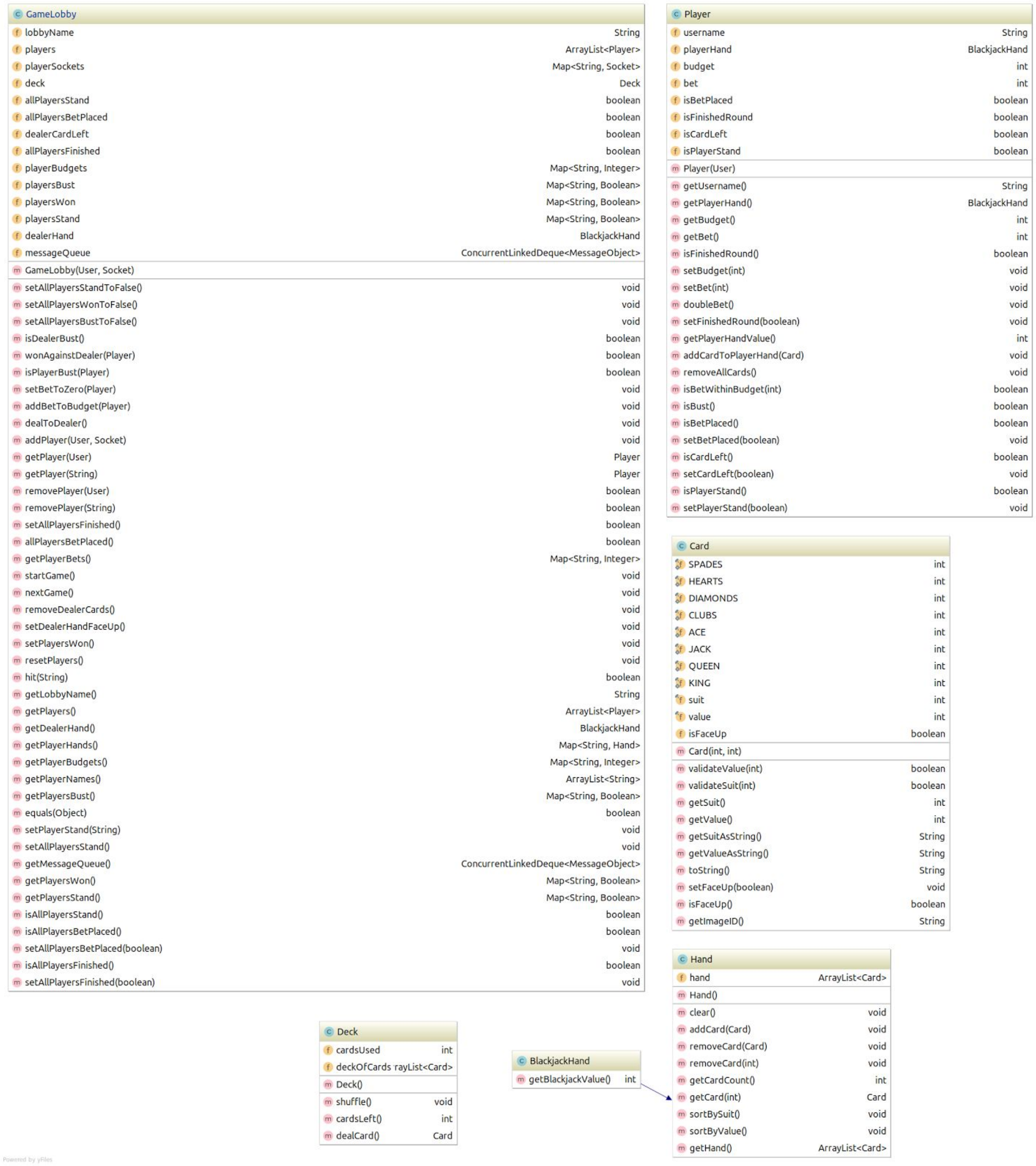
c	LoginScreen	
f	client	GameClient
f	blackjackOnline	BlackjackOnline
f	btnCreateAccount	JButton
f	btnLogin	JButton
f	lblUsername	JLabel
f	lblPassword	JLabel
f	usernameField	JTextField
f	passwordField	JPasswordField
f	lblLogo	JLabel
m	LoginScreen(GameClient, BlackjackOnline)	
m	initialize()	void
m	updateBounds()	void
m	getClientModel()	GameClient

c	HomeScreen	
f	client	GameClient
f	blackjackOnline	BlackjackOnline
f	btnLogout	JButton
f	username	String
f	lblWelcome	JLabel
f	btnGoToLobby	JButton
f	btnStatistics	JButton
f	btnSettings	JButton
f	lblCredits	JLabel
f	lblGamesWon	JLabel
f	lblGamesLost	JLabel
f	lblGamesPlayed	JLabel
m	HomeScreen(GameClient, BlackjackOnline)	
m	initialize()	void
m	updateBounds()	void
m	getClientModel()	GameClient
m	setWelcomeLabel(String)	void
m	update(Observable, Object)	void
m	showWarningWhenServerDown(GameClient)	void

c	CreateAccountScreen	
f	client	GameClient
f	blackjackOnline	BlackjackOnline
f	firstnameField	JTextField
f	lastnameField	JTextField
f	usernameField	JTextField
f	passwordField	JTextField
f	emailField	JTextField
f	btnSignUp	JButton
f	btnBack	JButton
f	lblEmail	JLabel
f	lblPassword	JLabel
f	lblUsername	JLabel
f	lblLastName	JLabel
f	lblFirstName	JLabel
f	lblCreateNewAccount	JLabel
m	CreateAccountScreen(GameClient, BlackjackOnline)	
m	initialize()	void
m	updateBounds()	void
m	getClientModel()	GameClient

Game Engine

Game Engine Class Diagram



GameLobby	
lobbyName	String
players	ArrayList<Player>
playerSockets	Map<String, Socket>
deck	Deck
allPlayersStand	boolean
allPlayersBetPlaced	boolean
dealerCardLeft	boolean
allPlayersFinished	boolean
playerBudgets	Map<String, Integer>
playersBust	Map<String, Boolean>
playersWon	Map<String, Boolean>
playersStand	Map<String, Boolean>
dealerHand	BlackjackHand
messageQueue	ConcurrentLinkedDeque<MessageObject>
GameLobby(User, Socket)	
setAllPlayersStandToFalse()	void
setAllPlayersWonToFalse()	void
setAllPlayersBustToFalse()	void
isDealerBust()	boolean
wonAgainstDealer(Player)	boolean
isPlayerBust(Player)	boolean
setBetToZero(Player)	void
addBetToBudget(Player)	void
dealToDealer()	void
addPlayer(User, Socket)	void
getPlayer(User)	Player
getPlayer(String)	Player
removePlayer(User)	boolean
removePlayer(String)	boolean
setAllPlayersFinished()	boolean
allPlayersBetPlaced()	boolean
getPlayerBets()	Map<String, Integer>
startGame()	void
nextGame()	void
removeDealerCards()	void
setDealerHandFaceUp()	void
setPlayersWon()	void
resetPlayers()	void
hit(String)	boolean
getLobbyName()	String
getPlayers()	ArrayList<Player>
getDealerHand()	BlackjackHand
getPlayerHands()	Map<String, Hand>
getPlayerBudgets()	Map<String, Integer>
getPlayerNames()	ArrayList<String>
getPlayersBust()	Map<String, Boolean>
equals(Object)	boolean
setPlayerStand(String)	void
setAllPlayersStand()	void
getMessageQueue()	ConcurrentLinkedDeque<MessageObject>
getPlayersWon()	Map<String, Boolean>
getPlayersStand()	Map<String, Boolean>
isAllPlayersStand()	boolean
isAllPlayersBetPlaced()	boolean
setAllPlayersBetPlaced(boolean)	void
isAllPlayersFinished()	boolean
setAllPlayersFinished(boolean)	void

Player	
username	String
playerHand	BlackjackHand
budget	int
bet	int
isBetPlaced	boolean
isFinishedRound	boolean
isCardLeft	boolean
isPlayerStand	boolean
Player(User)	
getUsername()	String
getPlayerHand()	BlackjackHand
getBudget()	int
getBet()	int
isFinishedRound()	boolean
setBudget(int)	void
setBet(int)	void
doubleBet()	void
setFinishedRound(boolean)	void
getPlayerHandValue()	int
addCardToPlayerHand(Card)	void
removeAllCards()	void
isBetWithinBudget(int)	boolean
isBust()	boolean
isBetPlaced()	boolean
setBetPlaced(boolean)	void
isCardLeft()	boolean
setCardLeft(boolean)	void
isPlayerStand()	boolean
setPlayerStand(boolean)	void

Card	
SPADES	int
HEARTS	int
DIAMONDS	int
CLUBS	int
ACE	int
JACK	int
QUEEN	int
KING	int
suit	int
value	int
isFaceUp	boolean
Card(int, int)	
validateValue(int)	boolean
validateSuit(int)	boolean
getSuit()	int
getValue()	int
getSuitAsString()	String
getValueAsString()	String
toString()	String
setFaceUp(boolean)	void
isFaceUp()	boolean
getImageID()	String

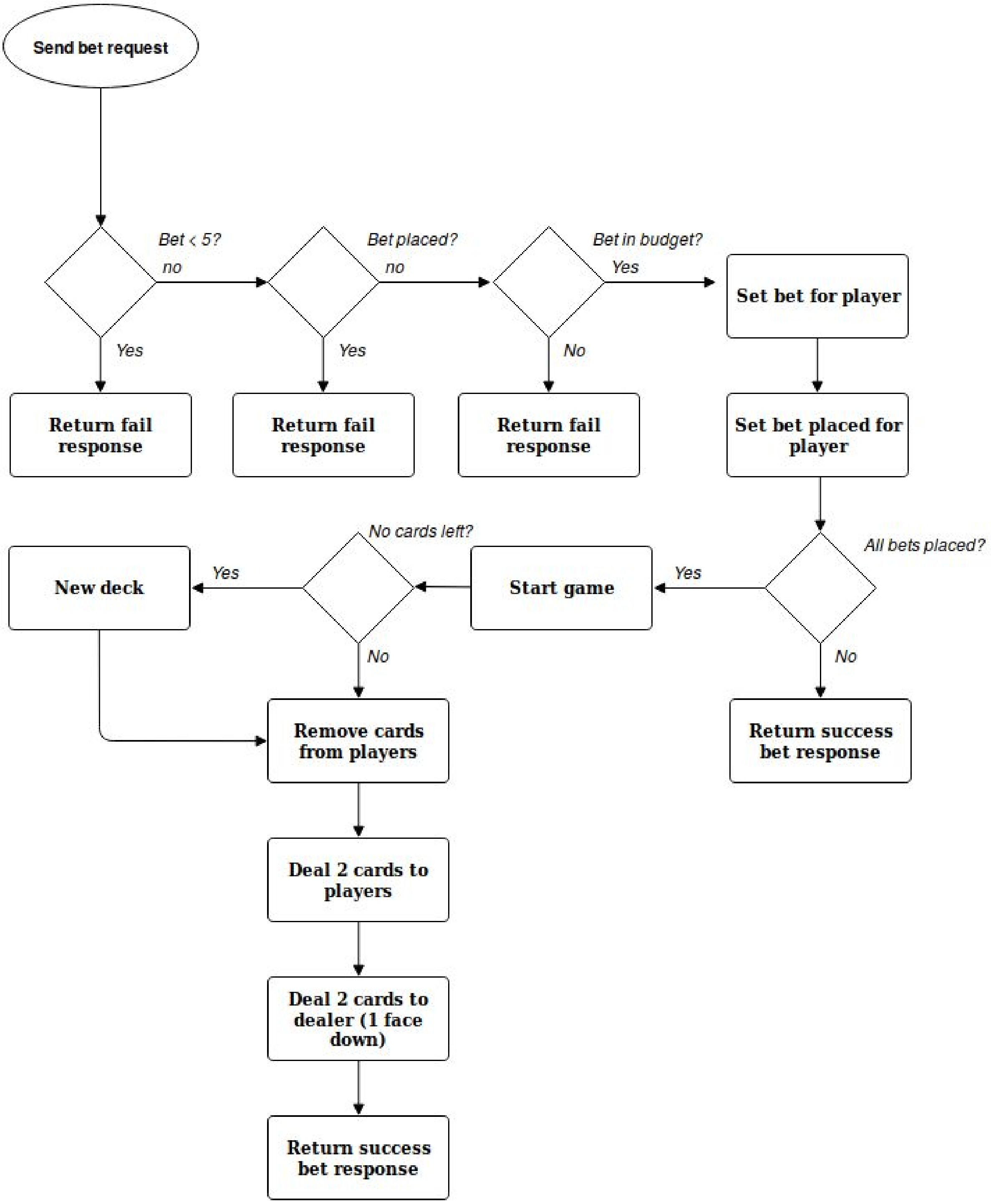
Deck	
cardsUsed	int
deckOfCards	rayList<Card>
Deck()	
shuffle()	void
cardsLeft()	int
dealCard()	Card

BlackjackHand	
getBlackjackValue()	int

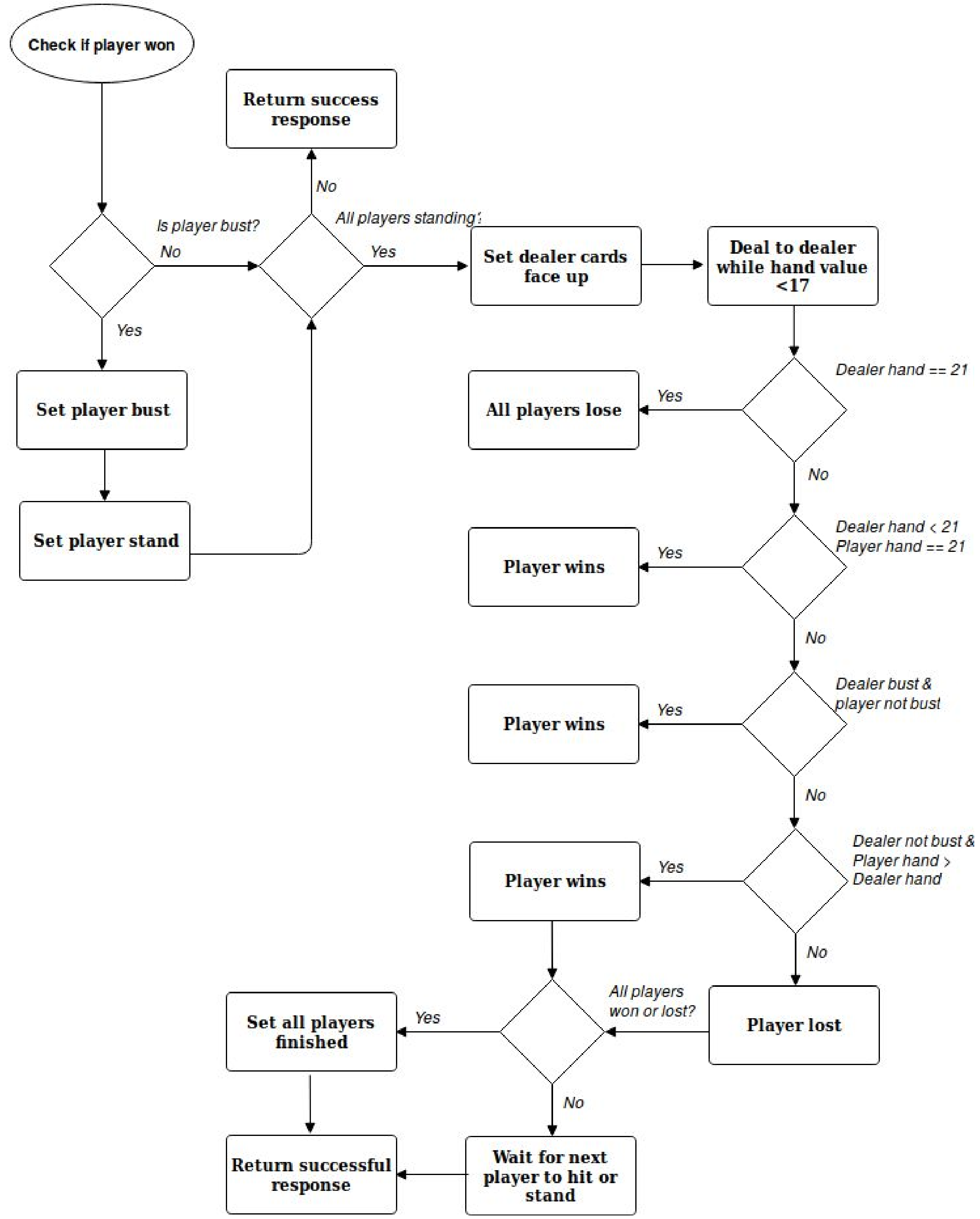
Hand	
hand	ArrayList<Card>
Hand()	
clear()	void
addCard(Card)	void
removeCard(Card)	void
removeCard(int)	void
getCardCount()	int
getCard(int)	Card
sortBySuit()	void
sortByValue()	void
getHand()	ArrayList<Card>

Game Engine Activity Diagrams

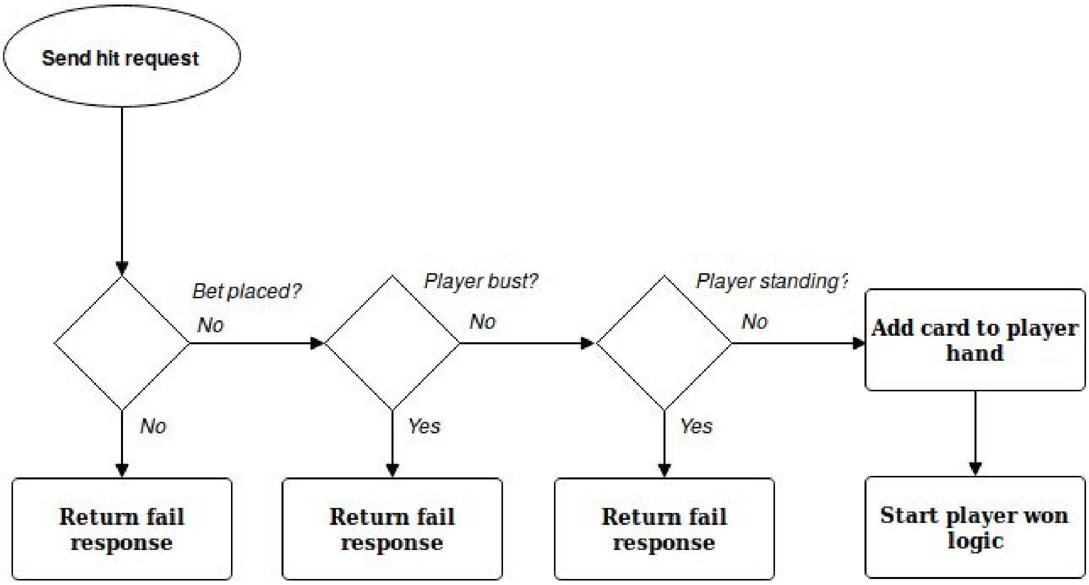
Blackjack Online - Placing a bet



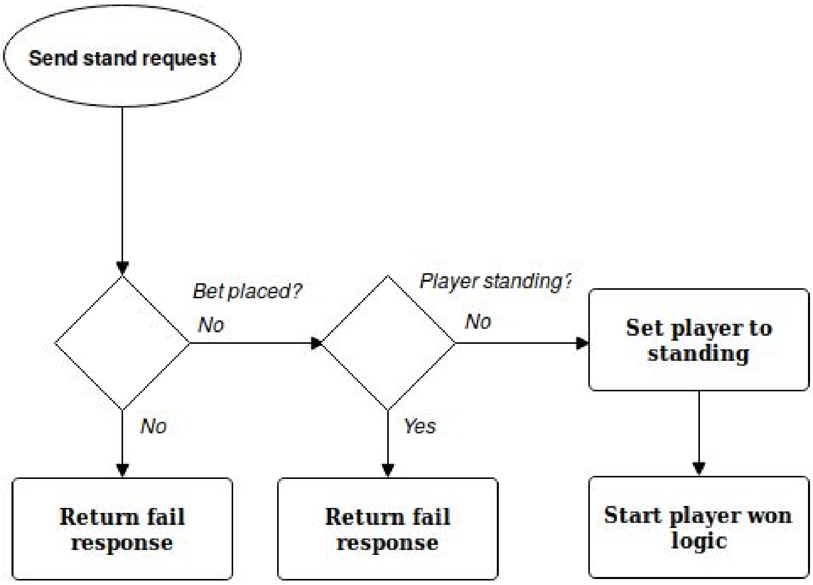
Blackjack Online - Player won logic



Blackjack Online - Hit request



Blackjack Online - Stand request



Testing

Test Plan

The overall test plan was based upon the user stories, which are shown in the following User Stores section. The user stories describe desired functionality for a user when interacting with the BlackjackOnline game. For each user functionality described in a user story, we also list the required functions that need to be written in Java. This helped us plan what code needed to be written. The last section of the user story describes how this functionality will be tested, which can be either functional or unit tests. This section helped with designing our test plan as explained in the following sections.

Test Plan for Server Side

Unit testing were employed to test the server, database, and game engine architecture. To test the server, unit tests were written that send a fake request to the server thread. The server thread handles these requests and should send an appropriate response. To determine if these pass, the expected response is asserted with the actual response. An example of such a test is a sending a login request with a correct username but incorrect password. In this test, a failed response is expected and the actual response is asserted to match this failed response. The response should also contain an error message of password mismatch which is also tested for. This allows us to test the communication of the client with the server, and also the server logic for handling requests.

The database and database interaction was tested via certain requests sent. The aforementioned example also tests the methods which interact with the database, as to check a username and password, the correct user details must be retrieved from the database. Several other unit test will be written for successful login and sign up requests, and unsuccessful login and signup requests. This enabled us to test the storing and retrieving of users in the database.

To test just the server architecture (including the game engine), fake request were sent as described previously. But only requests that solely interact with the server, not the database, are sent. An example of testing just the server architecture is a bet request. When sending a bet request with zero credits, a failed response is expected with the error message of ‘bet must be greater than or equal to 5.' This allowed us to further test the server for handling the requests which do not interact with the database and just the program memory.

The game engine was tested independently to the server and database. This involved testing the game lobby function as singular units with expected returns. To assist with these tests, a fake game will also be setup to test hit and stand functions (blackjack moves). Blackjack involves random generation so full game cannot be tested properly but enough unit testing was employed to ensure correctness. However, before distribution of the game, the random seed will be set to 1 so the cards are always dealt in the same order after shuffling. This will allow the game engine to be fully tested. Once tested, this seed will be set to the current time, to ensure correct randomness of shuffling. This allowed us to test the game logic and ensured correctness before connecting the game engine to the server.

Test Plan for Client Side

Client side tests focussed on functional testing in which a server was started and multiple clients were run. Guided by both the user stories and the functional test listed, the expected outcomes could be compared to what actually occurred. Any discrepancies that were uncovered could be found and debugged until the user story could be correctly performed in game. The use of this strategy enabled the testing of the GUI; its corresponding buttons and action performed events, frame resizing, chat messenger and game logic. This was initially tested with one single client before leading to tests with multiple clients.

An example runthrough of a user story (User Story 1) to test if a user can chat to others in a room and send messages would be as follows. Two clients would join the same game room, each user would compose and send a message to see if they both receive each other's message on their screen. This would, in practice, test all the functions laid out within the user story such as a function to compose and a button to send the message. If part of the story produced an unexpected outcome such as the button not sending a message, the error could be pinpointed and resolved.

Functional Tests

Test	Action	Sent requests and expected Response
1	Select Login with correct username and password	Request sent to server to check the database; the input matches and a response is sent to login into the client's account and send them to their HomeScreen.
2	Select Login with incorrect username or password	Request sent to server to check the database; the input does not match and a response is sent to display a warning message explaining the reason for a failed login is displayed.
3	Press Create Account button	Updates current screen to CreateAccountScreen.
4	Select Sign Up with fields inputted and unique username	Request sent to server to sign up user, a check is made on the database, the username is unique, the data is added to the database and the user is taken back to the LoginScreen
5	Select Sign Up with no fields inputted	Request sent to server to sign up user, a check is made on the database, no fields are filled, a warning message explaining the reason for a failed sign up is displayed.
6	Select Sign Up with existing username	Request sent to server to sign up user, a check is made on the database, username is not unique, a warning message explaining the reason for a failed sign up is displayed.
7	Select Back from Create Account page	Updates the current screen to LoginScreen
8	Select Go To Lobby	Updates the current screen to LobbyScreen
9	Select Logout	Request sent to log user out, response logs user out and updates the current screen to LoginScreen
10	Select Create Game	Request sent to create a new game, response adds a new game to the list of games stored locally in the server using the username as the game title. Client is then taken to their GameScreen.
11	Select Create Game if game with username already exists	Request sent to create a new game, game with your username already exists, a warning message explaining the reason for a failed game creation is displayed.
12	Select Join Game	Request sent to join a game that has been selected from the list of games in the Lobby panel. Response checks if game is not full.
13	Select Back from Lobby	Updates the current screen to HomeScreen
14	Select Bet	Request to submit bet amount that has been selected,

		response takes bet, deducts credits amount from local server
15	Select Bet1 (small bet of 5 credits)	Adds 5 credits to the local bet amount variable
16	Select Bet2 (medium bet of 10 credits)	Adds 10 credits to the local bet amount variable
17	Select Bet3 (large bet of 20 credits)	Adds 20 credits to the local bet amount variable
18	Select Bet4 (very large bet of 50 credits)	Adds 50 credits to the local bet amount variable
19	Select Hit	Request sent to Hit, game updates giving the player another card
20	Select Stand	Request sent to stand, game updates by allowing the player to keep their cards the game then progresses
21	Select Send Message	Request sent to add a new message to the private chatbox. Response adds message to the list so long as the message is non empty..
22	Polling Message	Continual requests from the client to update the message box corresponding to what is contained within the server.
23	Select Leave	Request to leave the GameScreen, response updates the current screen to LobbyScreen
24	Player creates a game, sends messages, another player joins same game	Second player enters and receives previously sent messages
25	2 players in the lobby screen. 1 players creates a game, then after several seconds, leaves the game	The player in the lobby screen sees the game created in the lobby screen, when the player leaves the game, the game is removed from the lobby screen.

User Stories

A series of user stories were created in order perform more functional tests on the created system.

User Story 1
Card: As a user, I want to talk in a chat room so that I can speak to other gamers.
Conversation: requirements to talk to uses

- *A function to compose messages*
- *A function to send messages*
- *A button to send/enter key*
- *A function to receive messages*
- *A way to store the chat history*

Confirmation: to test this we will send several messages between 2 or more users and if they are sent, received, and stored in the chat history then the goal is met.

User Story 2
Card: As a user, I want to be dealt cards so I can make a decision on what to do next
Conversation: Requirements necessary to deal cards
– A Deck of 52 cards
– A shuffle function for the deck
– A deal function that gives two cards two the dealer (one face up) and two to the player
Confirmation: To test this we will initialize several games of Black Jack. If the player is dealt two face up cards and the dealer has two cards (one face up, one face down) that are different each time, then the goal is met.

User Story 3
Card: As a user, I want to choose whether to "hit" (take a card), "stand" (end their turn).
Conversation: Requirements for the options to be implemented
– Functions to determine if the option selected is possible
– A button for each option
– Function to determine if two cards are of equal rank
– A hit function to add a card to the user's hand)
– A stay function to switch to the dealer's turn until the round is over
Confirmation: In order to test this we will play several games of Black Jack and attempt to use each option, if the attempts are successful then the goal is accomplished.

User Story 4
Card: As a user, I want the game to tell me if I go bust (over 21).
Conversation:

- *A function to count the users cards and tell them if they go over 21*

Confirmation: To test this, the user will keep hitting until the user has a total of over 21 and if the game says that the user is bust and makes them lose the round the goal is met.

User Story 5

Card: As a user, I want to have virtual chips/money to bet with so that there is an incentive to win.
Conversation: Requirements for this to happen
– A function to keep track of the user's money
– A function to determine if the user has enough dough (can't bet what you don't have)
– A function to add or subtract money from the user
– A button that allows the user to choose how much money to bet
Confirmation: To determine if this works we will place several different bets and examine if they work as expected. Such as if I bet too much money the game should prompt the user and request another bet.

User Story 6
Card: As a user, I want the dealer to deal according to the blackjack rules.
Conversation: Rules for the dealer
– Dealer hits until his hand is >= 17
If the dealer has an Ace, he will count it as low and hit again
Requirements for these rules
– A function to determine the total of the dealer's hand
– A function to stop the round when the dealer's total is >= 17
Confirmation: To test this we will play against the dealer for a few rounds to make sure that he keeps hitting until his hand is worth 17 or more.

User Story 7
Card: As a user, I want to have my own username in the chatroom so that people online know who I am.
Conversation:

- *A function to edit my username*
- *A function to display my username in chat*

Confirmation: to test this we will join a chat with other users and if both the other users and myself can see the user name then the goal is met.

Team organisation

The project has been split into five main sections: client, server, database, and GUI. Each team member was assigned to each of the following areas however there was overlap within the project and as sections were combined towards the end.

Tom: Server/Game engine/Client/Gui/Database
Lois: Database
Alex: Gui
Lloyd: Client
Yifan: Server/Game engine/Client/Gui/Database

Weekly meetings and group work sessions were undertaken throughout the project, meeting logs can be found in the appendix.

Evaluation

Blackjack Online was an ambitious project that required full commitment from all team members to achieve the full initial concept. The final product comes close to achieving this however certain elements were simplified such as the gameplay which only allowed for players to “Hit” or “Stand”. There were some initial difficulties using a version control code sharing platform Github and connecting work between different users since most of the team was new to this concept and hence resulted in a slow start to the project.

This slow start ultimately resulted in an increased workload towards the end of the project which, in-conjunction with the absence of a team member, increased the pressure for the rest of the team. Time management could have been handled better in order to make more progress earlier in the time frame when all team members were present.

Aside from these issue there are well implemented systems including the ability to start multiple private chat rooms and play the Blackjack card game with room for expansion in the future. Given more time, the team would have explored implementation of access to users with additional needs through adhering to Java's accessibility standards.

Appendix

Project Diary

Meeting 1

Date and Time	Friday 17th February 14:00
Attendance	All
Key Discussions	Potential project ideas and breakdown of work 1st part chat client 2nd part card game(s) BlackJack Potentially other games too. setup: Github for code sharing Google docs for documentation Maven as a build tool
Decisions	Breakdown of work: Tom: Server/Game engine/Client/Gui Lois: Database Alex: Gui Lloyd: Client Yifan: Game engine

Meeting 2

Date and Time	Wednesday 22nd February 11:00
Attendance	All
Key Discussions	● System architecture ● Database design ● GUI ● Client server Classes and class diagrams: *Key for the classes below* **## ClassName:** [optional] description

	--- Variables --- Methods **## User:** Descriptions --- Username, password, email, firstName, lastName, dateRegistered final --- Getters and setters for all. Stats from database is added on construction. Any changes in these variables updates in the database. **## FunctionsDB**: Description --- Variables --- isUserRegistered(username, password) return true / false; createAccount (username, password, firstName, lastName, email) return User; **## UserStats**: Contains the individual user stats which are stored in the database --- chips, gamesPlayed, gamesWon, gamesLost, totalFolds, totalBusts --- getters and setters. Contains all of the stats for the user **## CurrentGameStats:** Contains the stats for the game --- potSize, players[] --- Methods **## CardGameServer:** A multithreaded server which uses client threads and links the CardGameClient to the database. --- ServerSocket ClientThread --- Methods

	## ClientThread implements Runnable: A class which implements runnable and receives and send messages to the client. --- Socket --- Methods **## CardGameClient extends Observable:** The client which receives and sends messages to the server. It also extends observable and notifies the observers (screens) when there is a change. --- ScreenFactory --- SetChanged(), NotifyObservers() **## ScreenFactory:** This class controls encapsulates and controls which screen should be displayed depending on the user actions. --- LoginScreen, LobbyScreen, GameRoomScreen --- Methods **## LoginScreen extends Application:** This class contains all the buttons, text fields, and forms needed for the login screen. See the tutorial for building it with the ‘Applications’ class: http://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.htm. It looks way simpler than Swing, thoughts? --- GridPane, Scene, Button, Text, Textfield, Label --- EventHandlers etc. **## LobbyScreen extends Application:** This screen is what you see after logging in. It allows you to see available games and join them. --- GridPane, Scene, Button, Text, Textfield, Label --- EventHandlers etc. **## GameRoomScreen extends Application** This screen is where you can play blackjack with up to 4 other users. You can also chat to the other players.

	--- GridPane, Scene, Button, Text, Textfield, Label --- EventHandlers etc. **## MessageObject:** A data object to encapsulate messages. --- username, words , timestamp --- Getters, Setters **## ChatArea:** The area where the messageObjects are displayed. In other words, where the users talk to each other during the game. Is this necessary if we have the GameRoomScreen? --- ArrayList<MessageObject> --- Methods *GameObservable, ChatObservable* ScreenFactory: Extend JFrame/JPanel, controls which screen to display/room to open
Decisions	Work to be done following the key discussions, each team member will begin to implement their part of the project. Tom: Server Lois: Database design Alex: Gui Lloyd: Client model Yifan: Database functions, server

Meeting 3

Date and Time	Thursday 23rd February 14:00
Attendance	Alex, Tom, Lois, Lloyd
Key Discussions	● Creating observable and screen classes for the GUI: LoginObservable, CreateAccountObservable, LoginScreen, CreateAccountScreen ● Create a client class ● Expand server classes to: ClientThread, Server, User, and FunctionsDB. ● Design the user database
Decisions	Work to be done following the key discussions, each team member will continue to implement their part of the project. Tom: Server Lois: Database design Alex: Gui Lloyd: Client model Yifan: Database functions, server

Meeting 4

Date and Time	Tuesday 28th February
Attendance	All
Key Discussions	● Discussed ideas with David, ● Investigate using JavaFX for GUI ● Running client server tests ● Expand database design to cover the game database ● Look at creating the game engine
Decisions	Work to be done following the key discussions, each team member will continue to implement their part of the project. Aim to get Gui with database and client model together 3 screens of LoginScreen, CreateAccountScreen, and HomeScreen Setup both the user database and game database Tom: Server Lois: Database for game and corresponding functions Alex: Update GUI to work with client Lloyd: Client model Yifan: Game Engine

Meeting 5

Date and Time	Tuesday 7th March
Attendance	All
Key Discussions	Work done: Gui, Login, account creation, sign up, client server Required: ● Database for the game, Game database and functionsDB ● Game Engine finishing ● GUI instantiate panels at runtime ● Breakdown Gui into separate classes ● Add games to lobby screen ● Relate images to Card Class ● Screen Factory methods ● ArrayList for storing the number of games that can be joined ● Client model, methods to handle remaining protocols ● Begin report write up and corresponding diagrams
Decisions	Work to be done following the key discussions, each team member will continue to implement their part of the project. Tom: Server Lois: Database for game and corresponding functions Alex: Update GUI to work with client Lloyd: Client model Yifan: Game Engine

Meeting 6

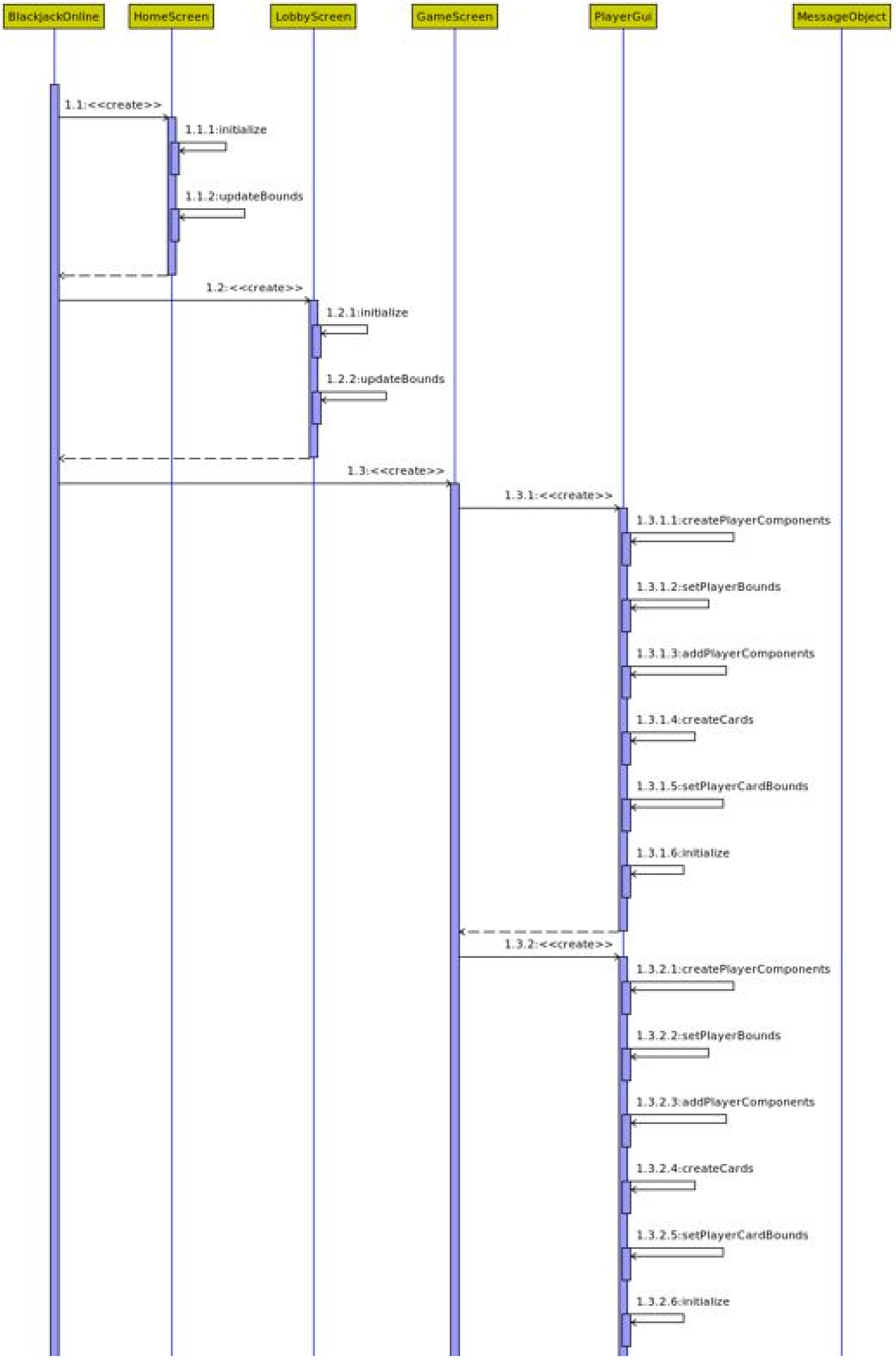
Date and Time	Tuesday 14th March
Attendance	All
Key Discussions	Work done: Gui now in separate classes frames are able to be resized, Login, account creation, sign up, client server, Lobby screen. Required: ● Game creation and joining ● Concurrency in the game lobby, update dynamically ● Writing of tests ● Expansion of GUI to be resizable in the game room ● Creation of GUI assets: avatars, cards, buttons, chips ● Send messages within the game room ● Continue Report write up
Decisions	Work to be done:

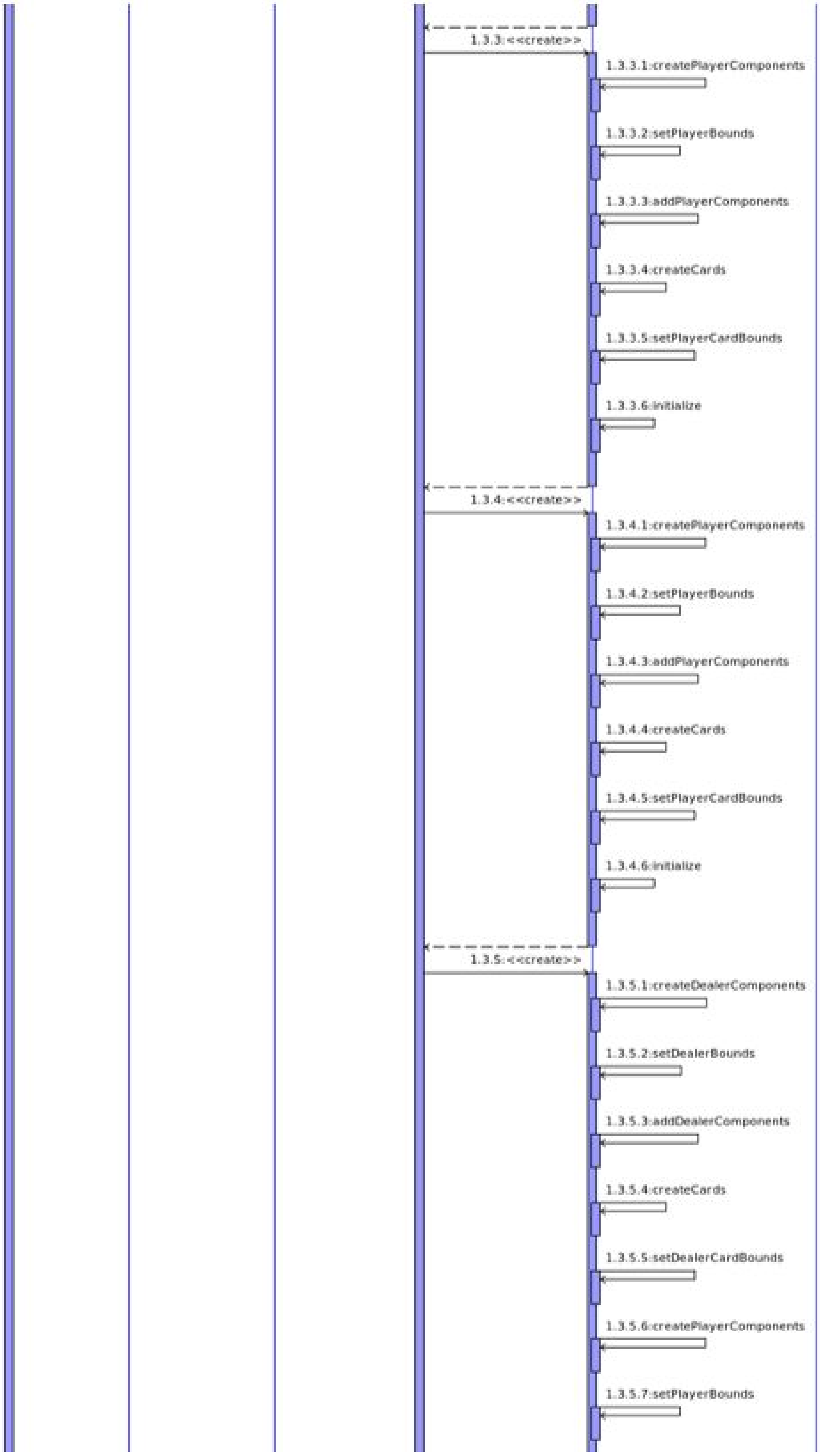
	Tom: Server and client model Lois: Database for game and corresponding functions Alex: Update GUI to work with client Lloyd: Client model Yifan: Game Engine

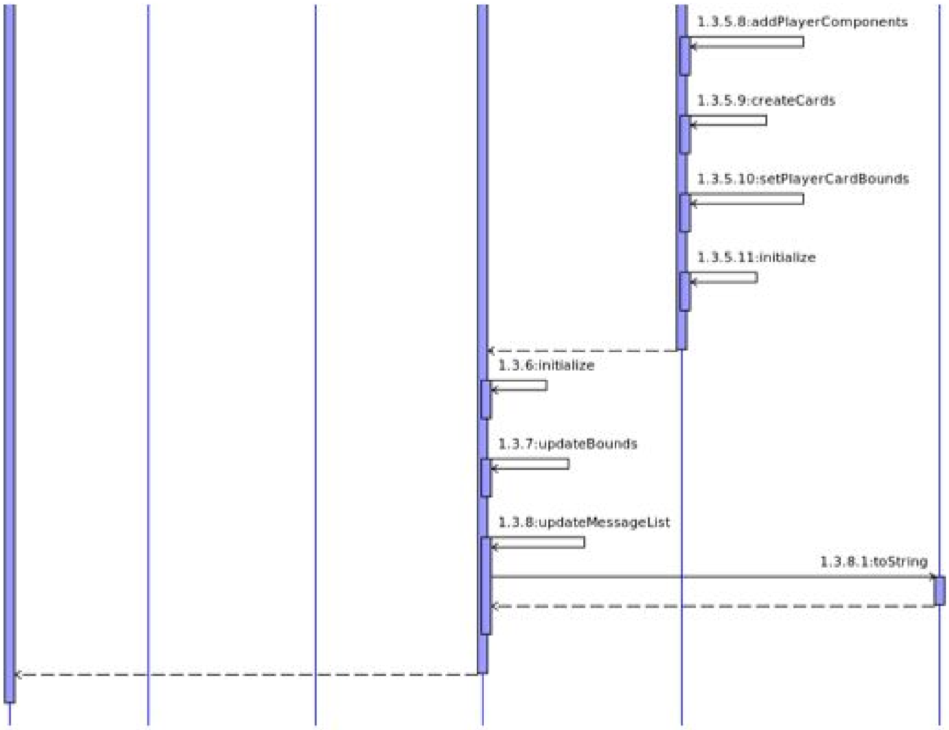
Meeting 7

Date and Time	Friday 17th March
Attendance	Alex, Tom, Lois, Yifan
Key Discussions	Work done: Gui now in separate classes frames are able to be resized, login, account creation, sign up, client server, lobby screen, game creation and joining. Working prototype shown, users are able to sign up, login, logout, create games, join games Required: ● Implement private chat room functionality ● Link game engine with GUI and client requests ● Deletion of games once last person has left the game room ● Updating of game database for users and their stats shown on the homepage ● Functional tests ● Testing functionality on multiple computers ● Finish report write up
Decisions	Team members will continue to implement their component parts: Tom: Server and client model Alex: GUI Yifan: Game engine and Server Lloyd: Client model

GUI Sequence Diagram




1.3.3:<<create>>
1.3.3.1:createPlayerComponents
1.3.3.2:setPlayerBounds
1.3.3.3:addPlayerComponents
1.3.3.4:createCards
1.3.3.5:setPlayerCardBounds
1.3.3.6:initialize
1.3.4:<<create>>
1.3.4.1:createPlayerComponents
1.3.4.2:setPlayerBounds
1.3.4.3:addPlayerComponents
1.3.4.4:createCards
1.3.4.5:setPlayerCardBounds
1.3.4.6:initialize
1.3.5:<<create>>
1.3.5.1:createDealerComponents
1.3.5.2:setDealerBounds
1.3.5.3:addDealerComponents
1.3.5.4:createCards
1.3.5.5:setDealerCardBounds
1.3.5.6:createPlayerComponents
1.3.5.7:setPlayerBounds


1.3.5.8:addPlayerComponents
1.3.5.9:createCards
1.3.5.10:setPlayerCardBounds
1.3.5.11:initialize
1.3.6:initialize
1.3.7:updateBounds
1.3.8:updateMessageList
1.3.8.1:toString