Technical Design: Server Component

# Classes

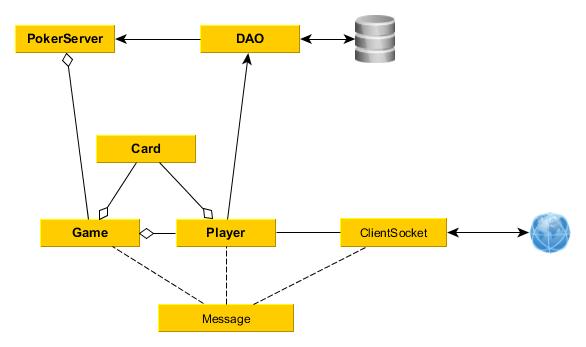


Figure 6: Class Diagram

## PokerServer

Instances of the class “PokerServer” shall represent a single concurrent server running. Each instance of the server shall listen on a specific port. It shall have the responsibility for managing connections and creating game objects as needed. This shall be a Singleton object

**Instance Members:**

* HashMap<Integer, Game> activeGames
* ArrayList<Player> allPlayers
* ArrayList<Player> lobbyPlayers
* DAO Database

**Instance Methods:**

* main
* createGame
* addClient
* addPlayerToGame
* authenticatePlayer

**Class Methods:**

* getInstance

## DAO

Instances of the class “DAO” shall represent individual databases in which data may be persisted. In practice, there should only be one instance per server.

**Instance Members**

**Instance Methods**

* updatePlayer
* checkCredentials
* addPlayer

## Lobby <iObservable>

Instances of the class “Lobby” represent a lobby where players reside when they are not in a Game.

**Instance Members**

* HashMap<Players, boolean> players

## Game <iObservable>

Instances of the class “Game” shall represent a single game currently in progress. It shall handle the turns of the game, including enforcing the business rules. There is the possibility of observing a game without being a player in it, though no client currently implements this behavior.

**Instance Members**

* Integer gameID
* HashMap<Players, boolean> players
  + The Boolean value indicates if they get a turn or not.
* ArrayList<iGameObserver> observers
* ArrayList<Card> tableCards
* ArrayList<Card> deck
* Integer chipsInPot

**Instance Methods:**

* parseMessage
* dealCard
* addPlayer
* removePlayer

## Card

Instances of the class “Card” shall represent individual cards in a deck. A deck consists of 52 Card objects, each different from the last.

**Instance Members**

* Enum suit
* String value

**Instance Methods**

**Class Methods**

* generateDeck

## Player <iStateObserver, iClientObserver>

Instances of the class “Player” shall represent a single player, either in a game or in the lobby.

**Instance Members**

* String username
* URL avatarURL
* Integer chipsRemaining
* ArrayList<Card> currentHand
* Game currentGame
* Lobby currentLobby
* iClient client

**Instance Methods:**

* addCardToHand
* resetHand
* getAvatar
* getClientForPlayer

## ClientSocket <iClient, iObservable>

Instances of the class “ClientSocket” represent external clients connected to the system via TCP sockets. This class will send messages to the client and listen for messages from the client, automatically rejecting anything that is not signed appropriately. This implements the iClient interface.

**Instance Members**

* Socket socket
* String authenticationKey
* ArrayList<iClientObserver> observers

## Message

Instances of the class “Message” represent a message coming from or going to a client.

**Instance Members**

* iClient clientInvolved

## ActionMessage

Instances of the class “ActionMessage” represent a message coming from a client. This can represent a Player Action, Authentication Attempt, or Create User action. ActionMessage inherits from Message and adds the following:

**Instance Members**

* Enum actionType
* HashMap<String,String> parameters

## StateMessage

Instances of the class “StateMessage” represent a message going back to a client. This can represent a Game State or Lobby State message. StateMessage inherits from Message and adds the following:

**Instance Members**

* Enum stateType
* HashMap<String,Object> parameters

## ErrorMessage

Instances of the class “ErrorMessage” represent a message going to a client that encodes an error. This can represent an Error message only. ErrorMessage inherits from Message and adds the following:

**Instance Members**

* Integer errorCode
* String errorText

# Interfaces

## iClient

This represents a client that can send and receive messages

**Instance Methods**

* receiveMessage
* sendMessage

## iClientObserver

This represents something that observes a client. It defines a common callback to be used.

**Instance Methods**

* onMessageReceived

## iStateObserver

This represents something that observes the state of a game or lobby. It defines a common callback to be used which takes as an argument the Message that represents the new state.

**Instance Methods**

* onStateChanged

## iObservable

This represents something that can be observed.

**Instance Methods**

* addObserver
* removeObserver

# Sequence Diagrams

## Login

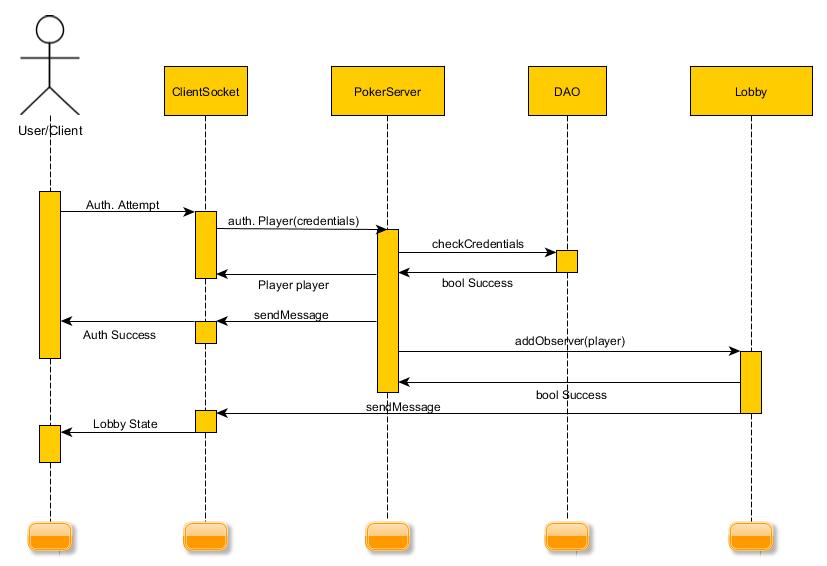


Figure 3: Login Sequence

When an authentication attempt is received by a ClientSocket, it is first validated for formatting. The credentials included in the message are extracted and sent to the PokerServer singleton, which then uses the DAO to verify that the credentials are valid. If they are not, an error message is returned and execution stops.

If the credentials are valid, the Poker Server begins constructing a Player object to represent the client. The client is sent an Authentication Success message, and the Player is assigned to a Lobby. Finally, the Player is returned to the Client so that a reference can be kept. Meanwhile, the Lobby to which the Player was assigned sends a LobbyState message.

## Joining a Game

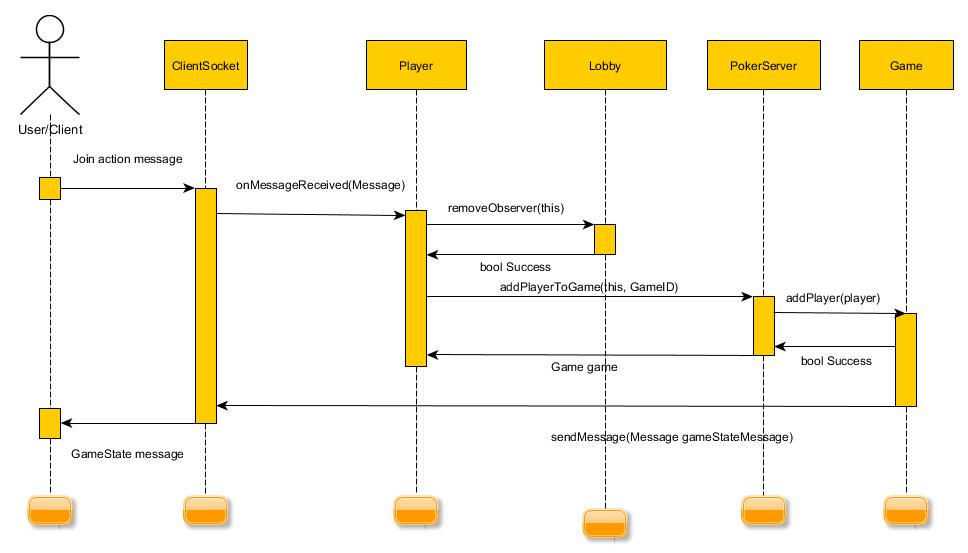
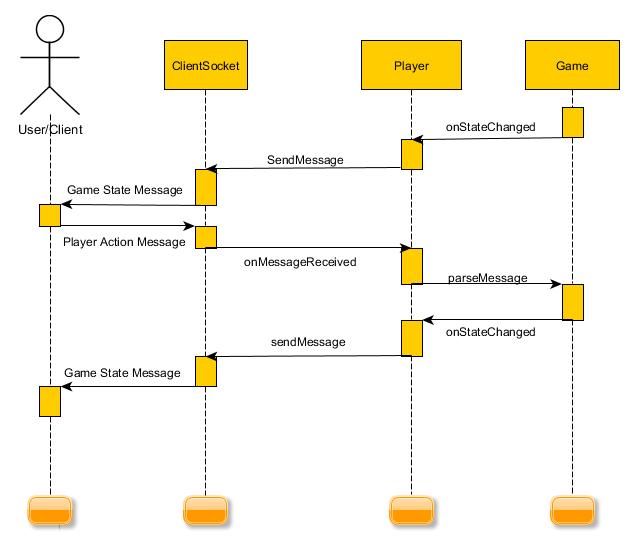


Figure 4: Joining a Game

When a Player Action message is received, it is parsed by the ClientSocket who received it into a Message object. The Message is passed to the Player, who interprets it as a Join action. The Player removes itself from the Lobby to which it was assigned, and asks the PokerServer singleton to add it to the Game that is represented by the GameID in the Join message. Finally, the Game sends a GameState update to all Players in the game, including the one that just joined.

## Player turn



When a Game State action is sent to a client indicating that it is that client’s turn, that client is expected to reply with a Player Action message. This message is passed to the Player, who interprets it as an action. Assuming this action is valid according to Player’s metrics (see System Design document), it passes this to the Game’s parseMessage method. Assuming the Game declares the action valid, it is applied, and the resulting GameState is sent to all Players, including the one initiating the action. This also indicates that it is the next player’s turn.