Advanced Programming Design Document

Gareth Sears
2493194S

April 1, 2020

1 Introduction

This software design document describes the architecture and system design of a distributed multiplayer version of the popular card game 'twenty-one' written in the Java programming language.

2 System Overview

2.1 Game Objective

As a player, the objective of the game is to increase your tokens by staking them against other players in a round of the card game 'twenty-one'. There is no defined victory condition and players can join and leave the game as they please, however, players will be removed from the game if they run out of tokens.

2.2 Game Rules

The rules adapt those of the original 'twenty-one' game as described on the *Twenty-One (Card Game)* Wikipedia page quoted below [1]. Any adapted rules are marked in **bold** and numbered, with descriptions of the changes or interpretations provided in Table 1.

> The game is played with a French-suited pack of 52 cards. Cards are worth their nominal value except for the Ace which scores 1 or 11 points at the player's discretion and court cards which are worth ten points each. **The first dealer is determined by lot e.g. first to draw an Ace wins (1). The dealer deals two cards to each player and himself, one at a time (2).**
>
> If anyone is dealt an Ace and Ten or Ace and court card as their first two cards this is a 'natural vingt-un' and must be declared immediately. If the dealer has a natural vingt-un, he receives **double stakes (3)** from everyone except any player who also has one, in which case it is a drawn game between them; **no payment is made either way (4)**. If a player has a natural vingt-un, everyone pays him **double stakes (3)** and he becomes the new dealer. If two players have a natural vingt-un, **no payment is made between them (4)** but the player with **positional priority (5)** wins the deal.

> The dealer then asks each player in **clockwise order (5)** whether he wishes to 'stand' or have another card. Players may request cards, one at a time, until they reach or exceed twenty-one, or decide to stand. If a player exceeds twenty-one, he throws in his cards and **pays his stake (3)** to the dealer. If no-one achieves a natural vingt-un, the dealer **pays single stakes (3)** to those whose numbers under twenty-one are higher than his, and receives from those who have lower numbers. No payment is made either way between the dealer and a player who has the same number. If the dealer exceeds twenty-one, he pays everyone who is still in the game.

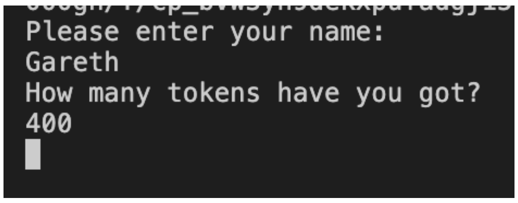
Reference #	**Adaptation / Interpretation**
1	The initial dealer is chosen according to the order players connect to the system. The first player to connect is automatically the dealer. If a dealer leaves during play, the player who connected after them becomes the new dealer in the subsequent round.
2	The system has no predetermined condition for ordering dealt cards, only that the deck should be shuffled prior to dealing. The current implementation deals a player two cards before moving to the next player. There is no concept of ownership of the deck by the dealer.
2	A single stake is hard coded as 20 tokens in this game's implementation.
4	This is interpreted as no payment is made between any players if more than one natural twenty-one is drawn. If a dealer shares a natural twenty-one with another player, they remain the dealer.
5	Positional priority is determined through order of connection, with the player who connected immediately after the dealer having 'positional priority' and going first. This connection list is cyclic, when the last connected player is passed, it returns to the first connected player.

Table 1: Game Rule Adaptations / Interpretations

2.3 Application Flow

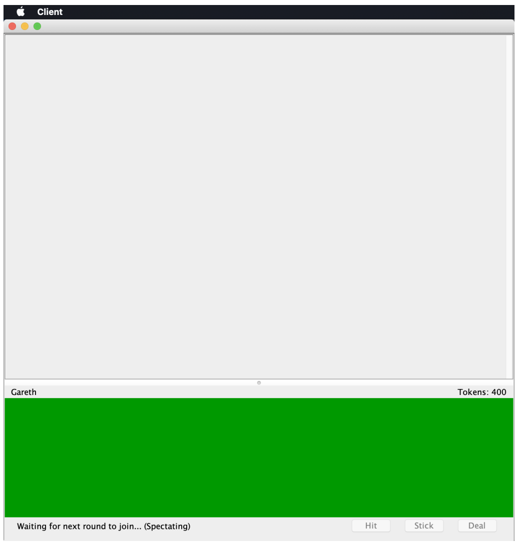
The following section describes the expected flow of control of the system. Firstly, the `Server` application is run. This persists throughout, and if the server is terminated at any point, all clients are closed automatically. Next, a `Client` application is run. The user enters their name and the number of tokens they wish to play with.

Following this, a GUI window opens showing the current round in progress (if any) and the user's own display (see Figure 1). In order to avoid disrupting player order mid-round, players connecting to the system while a round is in progress must wait until the beginning of the subsequent round to participate. However, they are able to spectate the current round, but the other players are not aware of their presence (see Figure 2). If no game is in progress, the system waits until two players have connected and then begins a round.



```
Please enter your name:
Gareth
How many tokens have you got?
400
```

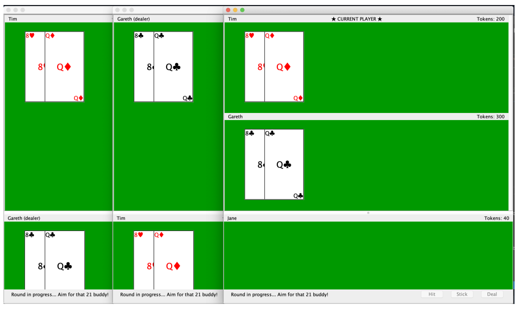
(a) A client initialising via command prompt



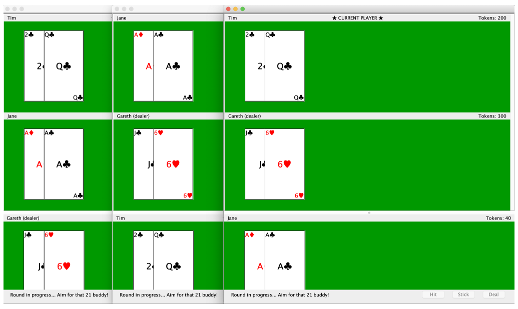
Waiting for next round to join... (Spectating)

(b) A Client GUI with no game in-progress

Figure 1: Starting a Client



(a) Spectating a round in progress

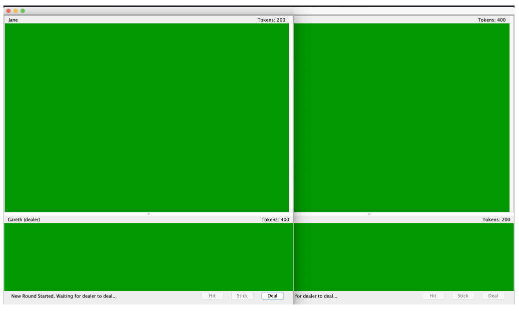


(b) Multiplayer round in progress

Figure 2: Spectating and joining rounds

When a round begins, the dealer's 'deal' button is enabled. After pressing this, the cards are dealt to each player and game play begins as described in Section 2.2. If it is a client's turn to play, their 'hit' and 'stick' buttons are enabled (see Figure 3). If the player reaches a hand value of 21 or surpasses it (goes 'bust'), play is automatically advanced to the next player. Otherwise, this happens only when they choose to settle with their current hand ('stick'). If all players are 'bust', the dealer automatically wins and their turn is skipped. If a natural 21 is scored following a deal, the round terminates with no input from the players needed. Tokens are transferred as described in Section 2.2, with immediate transfer to the dealer whenever a player goes 'bust'. When a game finishes, any players with no remaining tokens are automatically disconnected from the game and a new round begins automatically. Previous rounds' hands remain visible to the users until the next 'deal'.

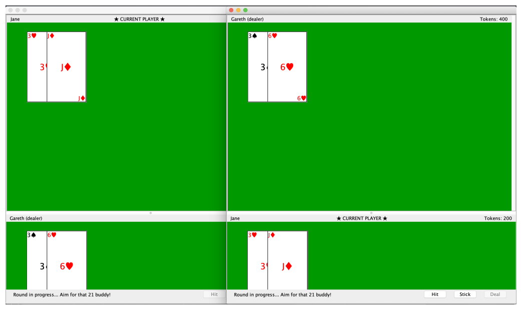
Players may choose to leave the game at any time. If a player disconnects mid-round, their representation is still visible to other players within the round and the computer automatically 'sticks' on their behalf when their turn arises. This is to ensure that tokens are always allocated fairly and to avoid 'rage-quitting'. If a dealer disconnects, the player who connected after them becomes the new dealer in the next round. If a player disconnecting leaves only one player in game, the next round does not continue automatically and the remaining player must wait until another player connects to the system before play is resumed.



New Round Started. Waiting for dealer to deal...

for dealer to deal...

(a) The beginning of a round

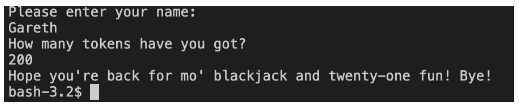


Round in progress... Aim for that 21 buddy!

Round in progress... Aim for that 21 buddy!

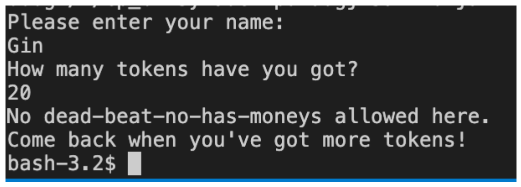
(b) A round in progress

Figure 3: Gameplay



```
Please enter your name:
Gareth
How many tokens have you got?
200
Hope you're back for mo' blackjack and twenty-one fun! Bye!
bash-3.2$
```

(a) Message on player disconnect



```
Please enter your name:
Gin
How many tokens have you got?
20
No dead-beat-no-has-moneys allowed here.
Come back when you've got more tokens!
bash-3.2$
```

(b) Player removed when tokens are zero

Figure 4: Disconnecting from play

Play continues indefinitely as players connect and disconnect. The only condition that terminates play for a particular client is if they run out of tokens or quit (see Figure 4).

3 System Architecture

3.1 Architectural Design

The design implements a server-client architecture where the client connects to the server on initialisation. On the server-side, there are four main sub-systems with different responsibilities. On the client-side, there are three. The responsibilities of these subsystems and how they collaborate, as shown in figure 5, will now be discussed.

On the server-side, the `Server` subsystem is responsible for handling client connections and instantiating a `ClientRunner` when a connection has been made. These `ClientRunners` handle IO streams to and from their respective clients via read and write threads and are also responsible for identifying when their client disconnects. They forward this information to a shared `ServerController`, whose primary role is to translate this information into appropriate action on the `ServerModel`, which handles the game state and business logic. The `ServerModel` has no awareness outside of itself beyond emitting state changes to any listeners. These are picked up by the ServerController and interpreted into appropriate API calls describing the state changes, which are in turn passed to the ClientRunner(s) for broadcast to their respective clients.

On the client-side, the Client subsystem is responsible for the initial connection with the server. Its ClientController translates any received API messages into actions on the ClientView, which visualises the game state in the ServerModel. When the user inputs information, the ClientView emits these events to the ClientController, which converts them into API calls to update the ServerModel via the aforementioned IO chain.

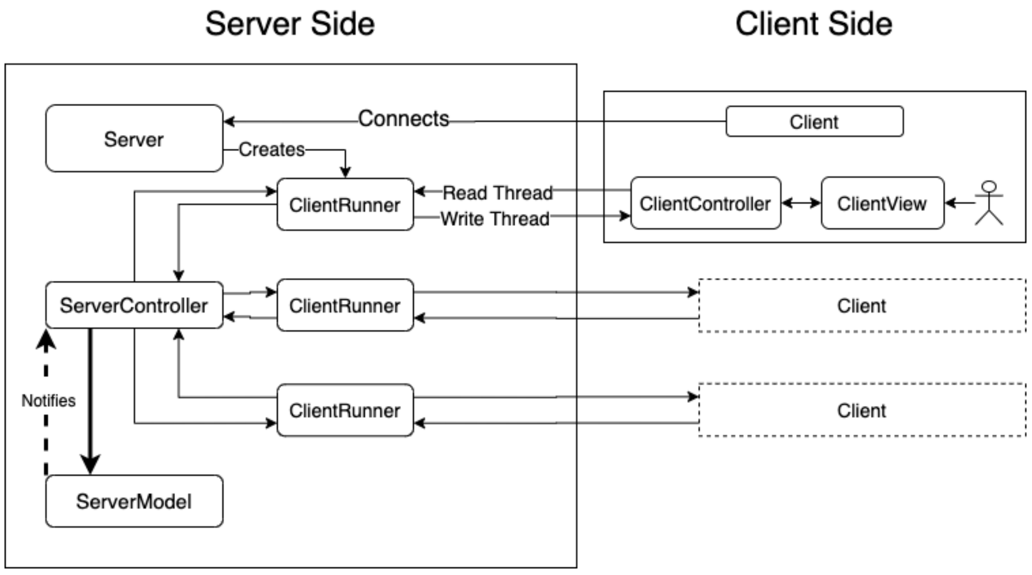


Figure 5: Architectural Design

3.2 Decomposition Description

A UML decomposition of the aforementioned subsystems is given in Figures 6, 7, and 8. Annotated sequence diagrams showing the API protocol and its transmission conditions are given in Figures 9, 10, and 11. API commands are written in *CAPITAL_LETTERS*, with their respective payloads in parenthesis in the format *(description:type)*.

3.3 Architecture Rationale

Several decisions were made during the architecture design that would have an impact on the implementation of the project. Firstly, the client-side has no model component. All game state is handled exclusively on the server and transmitted via API calls. This was decided so as to avoid state inconsistency that may arise by sharing state between client and server. Secondly, the ServerModel and ServerController would be shared between different threads, with the ServerController having knowledge of all ClientRunners. The ServerController was chosen as an intermediary between ClientRunners, as this simplified broadcast to all clients simultaneously and abstracted API call management away from the ClientRunners and `ServerModel`. The single shared model ensured encapsulated game logic, though necessitated thread-safe considerations in the software's implementation (discussed in following section).

4 Implementation

This final section discusses key implementation considerations at a high level. More detailed notes can be found in the project's source code.

4.1 Game Logic Encapsulation and Observables

All game logic and state is encapsulated inside the ServerModel and `Round` classes, with each class responsible for different stages of the `ServerModel` system described above. The `ServerModel` class is responsible for keeping track of `Player` objects which represent each client, as well as the current dealer. The class could be considered a 'lobby' through which players are added and removed, with the dealer changed accordingly. This mechanism ensures that player order is preserved when a round is in-play and avoids difficult edge cases, such as a player may joining while it is the dealer's turn (which should always be last in a round as per the game rules). When the round is initialised, all players in the model are added to the `Round` player list, allowing new players to join in a controlled manner.

The `Round` class is solely responsible for the flow of control as described in Section 2.2. To avoid coupling this control to classes external to this class, it was written as an observable. Thus, the `ServerController` is updated with its internal state via events. In keeping with this encapsulation, only a minimal set of methods which directly correspond to user input were made public on the `Round` class. In order to provide clients with player hand and token updates that take place within this flow, the `Player` was made observable as well.

4.2 API Protocol and Thread-Safety

All API messages are encapsulated within a `SocketMessage` class which contains a *command* as well as a corresponding *payload*. *Command*s dictate how the client and server should treat the accompanying *payload*, and if no *payload* is needed, this can be `NULL`. As both client and server were implemented in Java, the *payload* was implemented as a `Serializable` object. This led to a simpler design, primarily using the `Player` as a *payload*, as this class encapsulates almost all of the required information for display.

As `Player` objects would be shared outside of the model and passed among clients, it was imperative that they (and the classes composed within them) be thread safe. This was achieved through a combination of immutability and appropriate locking. Immutability is demonstrated in classes such as the `Card` class, as well as any lists of `Players` passed outside of the model. These use the `Collections.unmodifiable` family of functions to disable editing operations, ensuring their safety. Appropriate locking is implemented by using `Synchronized` blocks, using the target `Object`'s *intrinsic* or *'monitor'* locks. In some cases, a class may be composed of several of these monitor objects. This avoids locking the whole class when methods only act on a subset of the class's instance variables, preventing unnecessary blocking. An example of this can be found in the `Player` class, where operations on `tokens` do not block operations on the player's `hand`.

Thread safety was also a consideration for when controllers interact with IO classes. When a controller class sends a message to a socket writer class, such as `ClientController` or `ClientRunner`, a provider / consumer pattern is followed. This uses a `ConcurrentBlockingQueue` to ensure thread-safe transfer of messages to each socket writer, as several threads may wish to contact a particular client via a controller simultaneously. The `ServerModel` class was also designed to be thread-safe, with appropriate locks made on objects whose state is being altered at any one time. As the model relies on the observer pattern, any events fired take place *outside* of the synchronised blocks. This avoids potential deadlocks where callback functions attempt to access a block of code from which an event has been fired.

5 Reflections

Overall, the project was highly rewarding and gave an insight into the implementation of threads and sockets as part of a system. Several questions remain based on the author's assumptions. Firstly, is an observer pattern a thread safe mechanism? I believe so, although it is my understanding that there is a possibility for events to be received out-of-order. In the context of this game, as it is turn based, I do not believe it impacts the client's experience of the game, as the displayed state will *eventually* be consistent with the model (which is the best that one can hope for in concurrent design!). Perhaps a better alternative exists with similar cohesion benefits. Secondly, are `Serializable` objects the best way to transfer information over a network? They obviously restrict the client application to being written in Java. This approach also has a drawback in that an `ObjectOutputStream`'s cache needs to be reset when repeatedly passing the same object reference, else its state changes are ignored. This was the cause of much frustration during development until a solution was found! Both of these are avenues for further research by the author.

6 References

References

[1] Wikipedia contributors, "Twenty-one (card game) — Wikipedia, the free encyclopedia," 2020, [Online; accessed 21-March-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Twenty-One$_(card_game)\&oldid = 947222161$

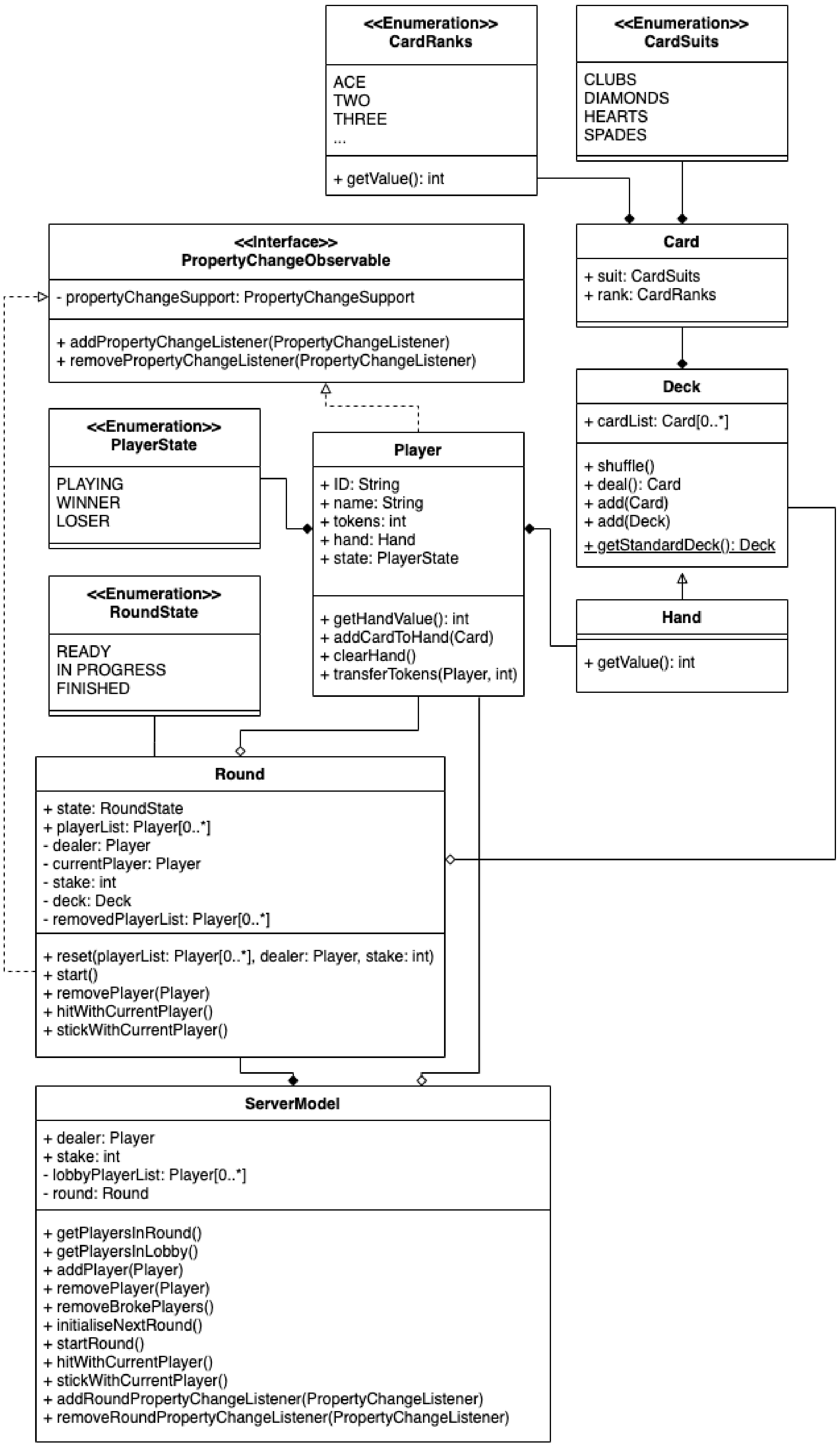


Figure 6: ServerModel Decomposition

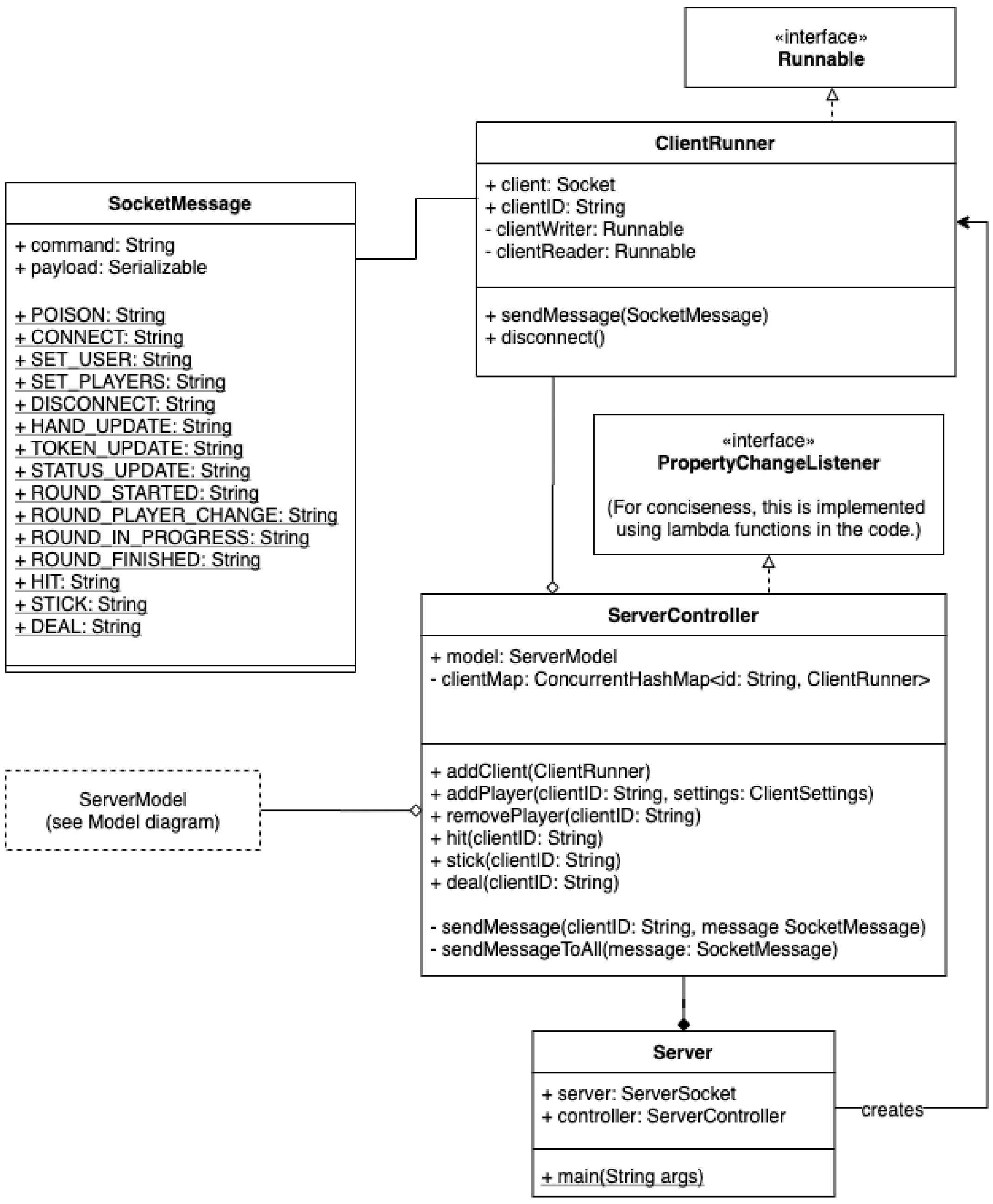


Figure 7: Server, ServerController and ClientRunner Decomposition

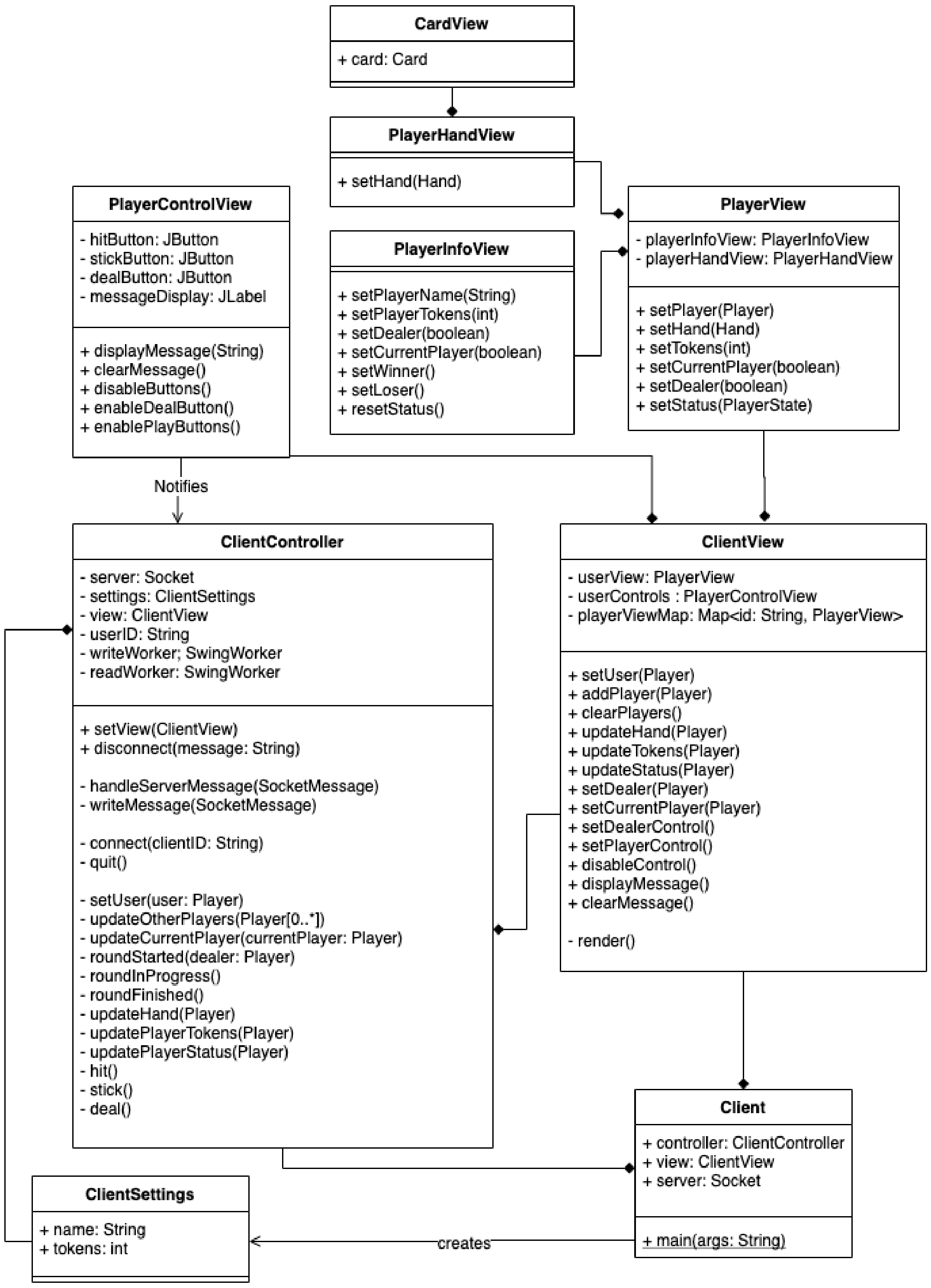


Figure 8: Client, ClientController and ClientView Decomposition

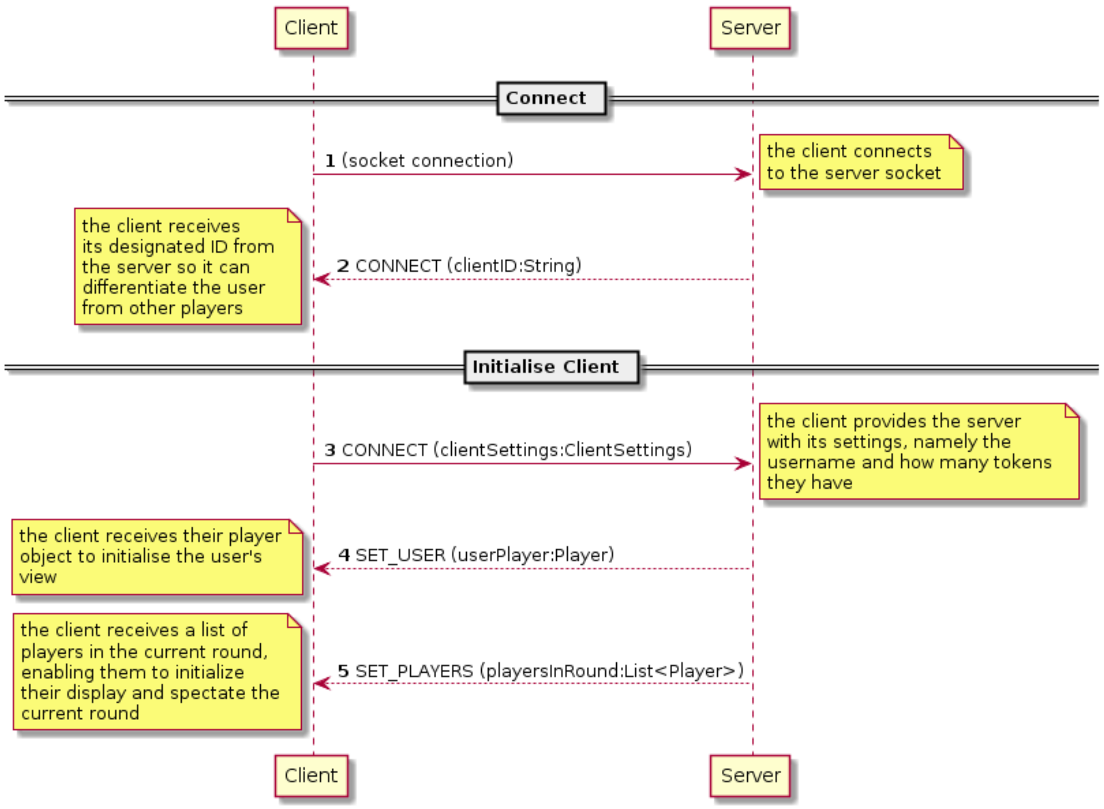

Client
Server
Connect
1 (socket connection)
the client connects to the server socket
2 CONNECT (clientID:String)
the client receives its designated ID from the server so it can differentiate the user from other players
Initialise Client
3 CONNECT (clientSettings:ClientSettings)
the client provides the server with its settings, namely the username and how many tokens they have
4 SET_USER (userPlayer:Player)
the client receives their player object to initialise the user's view
5 SET_PLAYERS (playersInRound:List<Player>)
the client receives a list of players in the current round, enabling them to initialize their display and spectate the current round
Client
Server

Figure 9: API calls made on a client's initial connection

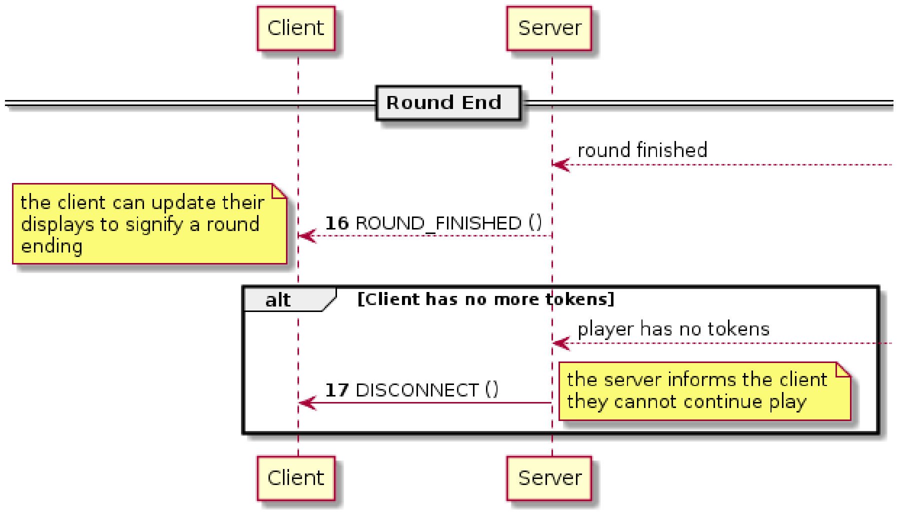

Client
Server
Round End
round finished
16 ROUND_FINISHED ()
the client can update their displays to signify a round ending
alt
[Client has no more tokens]
player has no tokens
17 DISCONNECT ()
the server informs the client they cannot continue play
Client
Server

Figure 10: API calls made at the end of a round

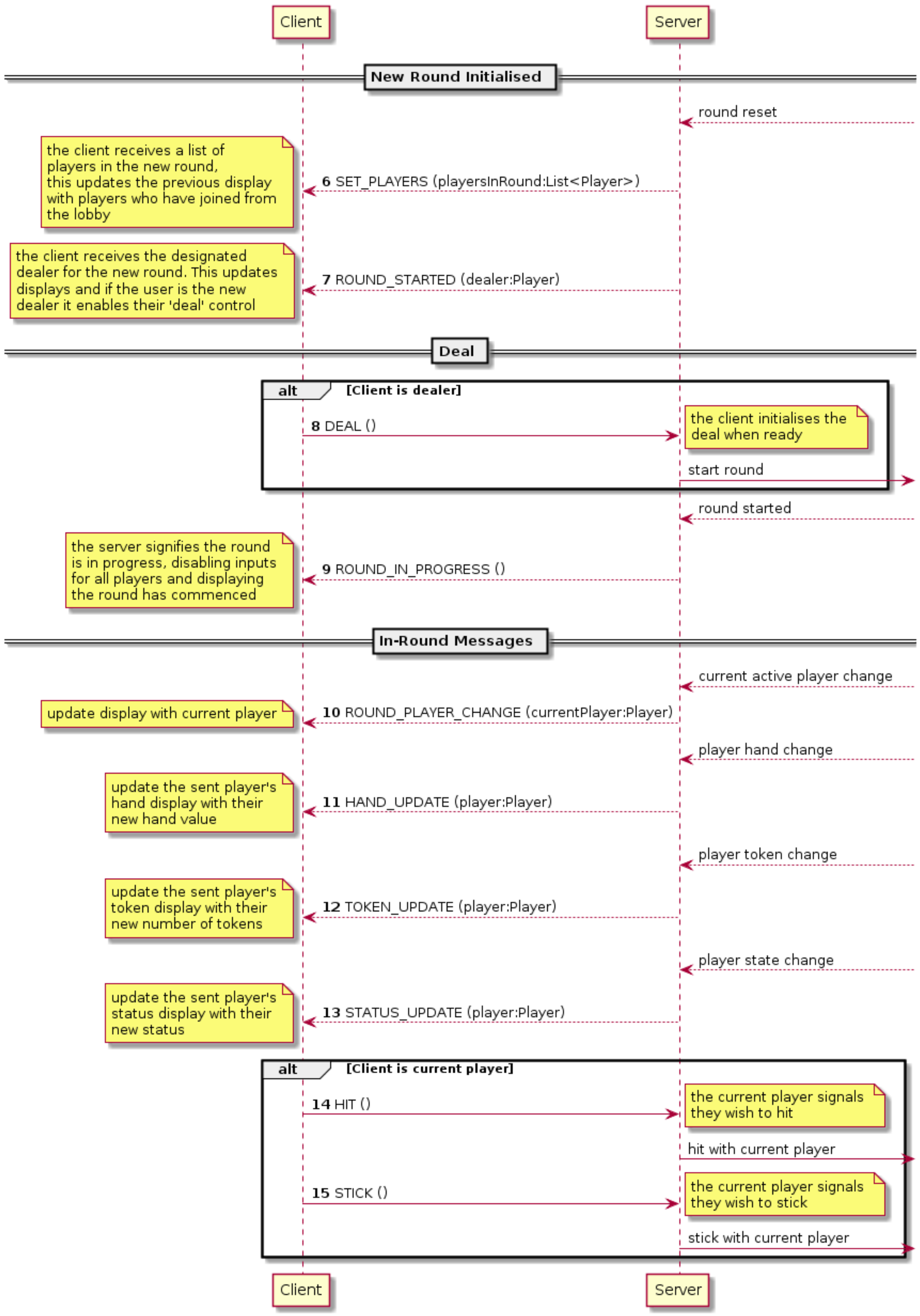

Client
Server
New Round Initialised
round reset
the client receives a list of players in the new round, this updates the previous display with players who have joined from the lobby
6 SET_PLAYERS (playersInRound:List<Player>)
the client receives the designated dealer for the new round. This updates displays and if the user is the new dealer it enables their 'deal' control
7 ROUND_STARTED (dealer:Player)
Deal
alt [Client is dealer]
8 DEAL ()
the client initialises the deal when ready
start round
round started
the server signifies the round is in progress, disabling inputs for all players and displaying the round has commenced
9 ROUND_IN_PROGRESS ()
In-Round Messages
current active player change
update display with current player
10 ROUND_PLAYER_CHANGE (currentPlayer:Player)
player hand change
update the sent player's hand display with their new hand value
11 HAND_UPDATE (player:Player)
player token change
update the sent player's token display with their new number of tokens
12 TOKEN_UPDATE (player:Player)
player state change
update the sent player's status display with their new status
13 STATUS_UPDATE (player:Player)
alt [Client is current player]
14 HIT ()
the current player signals they wish to hit
hit with current player
15 STICK ()
the current player signals they wish to stick
stick with current player
Client
Server

Figure 11: API calls made on a round starting and in-play