Poker game

(Design Document)

https://github.com/srhee91/PokerGame

Team 3

So Mi Choi, Bo Heon Jeong, Hanchen Li,
Sang Rhee, Yixin Wang, Yuying Wang

Design Document

Purpose:

We are designing a Texas Hold'em game for desktops and laptops designed to be played over a local network with friends. Each game can support up to 8 players and additional spectators with individual GUIs for each person. The GUIs will have simple animations and sounds to display each player's actions.

Non-Functional requirements:

- Response time:
  As a user, I would like a low latency connection between my client and the game server so I can see other players' actions in as close to real-time as possible.

- Throughput:
  - As a user, I would like the game server to be able to communicate with up to 20 clients (players or spectators) with no discernible increase in latency. This way, the server will be able to easily communicate with all players in an ongoing game as well as a significant number of spectators.
  - As a user, I would like the GUI to run at a minimum of 40 frames per second at all times so the game animations are smooth and responsive.

- Resource usage:
  - As a user, I would like this game to be light on resource usage in terms of CPU and network communications. Since the game will not a very computationally-intensive program and there can be a lot of waiting from one turn to the next, I should be able to do other tasks outside of the game while I'm waiting.

- Reliability:
  - As a user, I would like to be able to play multiple sessions of Poker in one sitting without the program crashing or my local game state going out of sync with that of the server.

- Availability:
  - As a user, I would want this program to be able to run on a variety of operating systems including Windows, Mac, and distributions of Linux. This way, I will be able to enjoy the game on most desktops and laptops.

- Recovery from failure:
  - As a user, I would like the game to be able to reconnect players to the game server if their connection is temporarily lost because I don't want to lose all the progress I've made in a game due to a spotty connection.

- Allowances for maintainability and enhancement:
  - As a developer, I would like the game code to be well-organized and compliant with standard Java coding standards with comments where necessary. This way, I will be able to modify the code easily to add new features in the future.

- Allowances for reusability:
  - As a developer, I would like certain part of the game to be reusable, such as the network code and the game's graphical resources. This way, developing more networked card games in the future will not require as much effot.

Design Outline:

a. Design Decisions:

We'll be following the client-server model. The major components will be the server (game host), clients (players and spectators), and GUI (one per client).

Server (game host):

One server instance will be created per game. The purpose of the server is to keep track of the poker game state (e.g. chip amounts and cards for each player, the cards on the table, the pot and side pots, player actions). It will poll player clients in order and receive their responses and update the player state according to the user action it received.

Clients (players and spectators):

The client will run on each player's or spectator's systems and will send and receive messages from the server to update its local copy of the game state, which will include just enough information for the GUI of that player or spectator (spectators will have all players' cards face up, while players will only have their cards face up, except in the case of showdown).

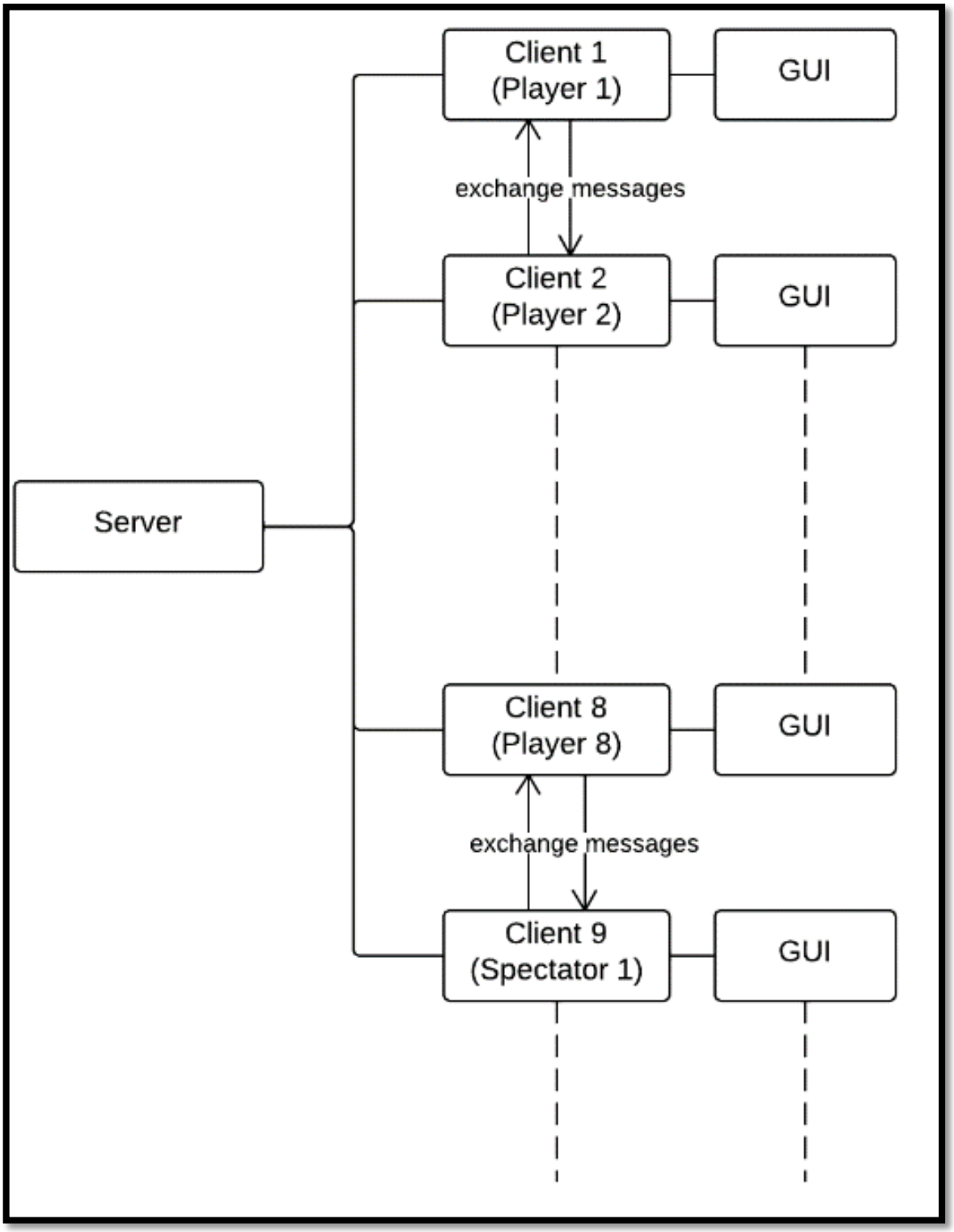
GUI (one per client):

The GUI will run on each player's or spectator's system in its own thread. The GUI will be responsible for displaying the onscreen buttons and the actions of players through simple animations and accompanying sounds. The player's button presses will be detected and the corresponding action will be carried out.

b. Component Interactions:

Before creating/joining/spectating a game, the user's system will only have the client code and GUI code running. If they choose to create a game, a new host process will be spawned on their system and will be connected to by other clients through the network. If they choose to join or spectate a game, their client will simply connect to the host of the desired game, which is specified by a socket number. During an ongoing game, the host will send messages to each client to indicate it's their turn, basically polling that system for a player action as well as sending a subset of the game state to that player so their GUI can be updated. That player's action will originate from their GUI, which will have methods to relay these actions to the client running on the same system. The client will send a message containing the player's action to the server (which may or may not be running on the same system, depending if this player is the game host or not). The server will then calculate the next poker game state. Then the host will move on to the next player in the order of play and poll it. When the game ends (when only one player remains), the host process will no longer run on the host's system. In summary, the client code and the GUI code will run on every player and spectator's systems, while the host code will only run on one player's system per game.

Figure1. Client-Server diagram



Design Issues:

Issue 1. On which system will the server reside?

Option1: One of the players will run the host code in addition to the client code.
Option2: Use an existing server-hosting or server-rental provider.
Option3: Use an additional system as a dedicated server.

Choice: **option1** - Our game is aimed at people who want to get together and play. For the best usability, we decided not to have the server on a separate system to avoid the need to bring an extra system. Also, by not using a remote third-party server, it improves reliability, reduces latency, and allows players to play with just a LAN connection.

Issue 2. What if a player loses the game?

Option1: He becomes a spectator.
Option2: He may choose to either leave the game or become a spectator.

Choice: **option2** - For best usability, we should allow the player to choose between leaving the game and becoming a spectator when they lose. If the player simply wants to join a different game after the loss, they shouldn't have to become a spectator first and then leave, which requires an extra action for the user.

Issue 3. How should the game's tasks be distributed among processes and threads for the best performance?

Option1: Have the GUI class handle all the actions from the player and the server in one process.
Option2: Have the GUI and client in one process, and have the host in another process.
Option3: Have the GUI and client in one process but two threads, and have the host in another process.

Choice: **option3** - In order for the GUI to have a high and consistent frame rate, the GUI should run in its own thread. As for the client and host code in the case of a host player, they should run in separate processes since they can be viewed as two independent programs with separate tasks. When they're both running on the host player's system, they should remain separate so the server-client interaction is similar to those of the other players.

Issue 4. What should happen to the server process if the host loses or quits out of the game?

Option1: The host will be forced to spectate the game until it ends, allowing the server process to continue to run until the end of the game.
Option2: The server process will continue to run on that host's system, even if he joins or starts hosting a different game, or even if he exits the program completely.
Option3: The server process will transfer the entire game state to a different player and then terminate. The player who received the game state will spawn a server process and become the new host (host migration).

Choice: **option3** – Even though host migration is the most difficult option to implement, it makes the most sense in terms of game design. Once the host loses/quits the game, their system should not be forced to still take part in that game, either directly (option1) or indirectly (option2). Most likely, the host will want to move on to a different game or a different program.

Issue 5. Who should keep track of the time remaining for a player's turn, the client or the server?
Option1: The client's timer will be used.
Option2: The server's timer will be used.

Choice: **Option2** – The server's timer will be used. Even though the client will also run its timer during a player's turn so its GUI can display the time remaining, it's ultimately the server's timer that will force that player to fold when it runs out. This may lead to a slight discrepancy between the time remaining that a player's GUI shows and what the server timer says due to network latency, but this can be addressed by adding some extra time into the server's timer. Option 2's main advantage is that it's more resistant to cheating: the client won't be able to cheat by messing with the system clock ticks.

Design Details:

a. Class Descriptions and Interactions:

- *Poker* is the main class. It is responsible for initializing the client code and the host code. It will instantiate a *Client* object every time by calling *startClientProc()*, and it will instantiate a *Host* object with *startHostProc(port)* only if the player wants to create a new game.

- *Host* is the class that's responsible for running the server of the game. It holds the overall game state (*PokerGameState* class*)*, handles all the calculations (*GameSystem* class*)*, and communicates to all the clients (*HostMessageHandler* class*)* with the *sendGameState()* and *receivePlayerAction()* methods*.*

- *GameSystem* is the class that has static methods for computing the next game state such as distributing the pot, shuffling, and calculating the best hand. The *Host* class will call the *updateGameState(action)* method of this class, which will in turn call its other static methods to calculate the new game state.

- *HostMessageHandler* and *ClientMessageHandler* are classes contained in a *Host* instance and a *Client* instance respectively. They're responsible for the socket communication between a server and client, which consists of messages containing the game state or a player's action. The *HostMessageHandler* uses the methods *sendGameState()* and *receivePlayerAction()* to communicate with *ClientMessageHandler*, and *ClientMessageHandler* uses the complementary methods *sendPlayerAction(action)* and *receiveGameState()* to communicate with *HostMessageHandler.*

- *Client* is the class that's responsible for the client side of the game. It holds a copy of the local copy of the game state (*PokerGameState* class), handles communications from the host (*ClientMessageHandler)*, and contains code for the GUI *(GUI* class).

- The *GUI* class holds and controls the classes representing the four game modes. It switches between the four game mode classes and keeps track of the current mode in the variable *currentMode*.

- The *BasicGameState* class is a superclass for the four game mode classes. It provides functions to be overridden by the game mode classes, including the *init()* method to initialize GUI elements, the *update()* method to updating the screen state, and *render()* to draw all the GUI elements to the screen.

- The *StartMode* class provides the interface when the user opens the application at the beginning. It will allow the user to input a name and to either create, join, or spectate a game (the game is specified by entering a port number into a textbox).

- The *LobbyMode* class provides the interface when the user joins an existing game. It shows the names of all the players who have joined and new players as they join. The host can start the game through a "Start" button when there are enough players.

- The *OngoingMode* class provides the interface for the actual game as it plays. It shows each player's cards (face up or face down), the flop, turn, river, the pots and side pots, the chip amounts, etc. It also provides buttons for user actions like raise or fold.

- The *OverMode* class provides the interface for when the player loses or the game ends. It will show game stats and options to either become a spectator (if the game is still going) or return to the start screen.

- *PokerGameState* is a structure class for saving game state variables, which will be used in the *Host* and *Client* classes. It contains one instance of the *TableInfo* class and eight instances of the *PlayerInfo* class.

- The *TableInfo* class is a structure class for saving variables used for table information, such as player ID, dealer, big blind, small blind, card.

- The PlayerInfo class is a structure class for saving integers or boolean variables used for players including the cards in their hand, their chip amount, and if the cards are visible.

Figure2. UML Class Diagram

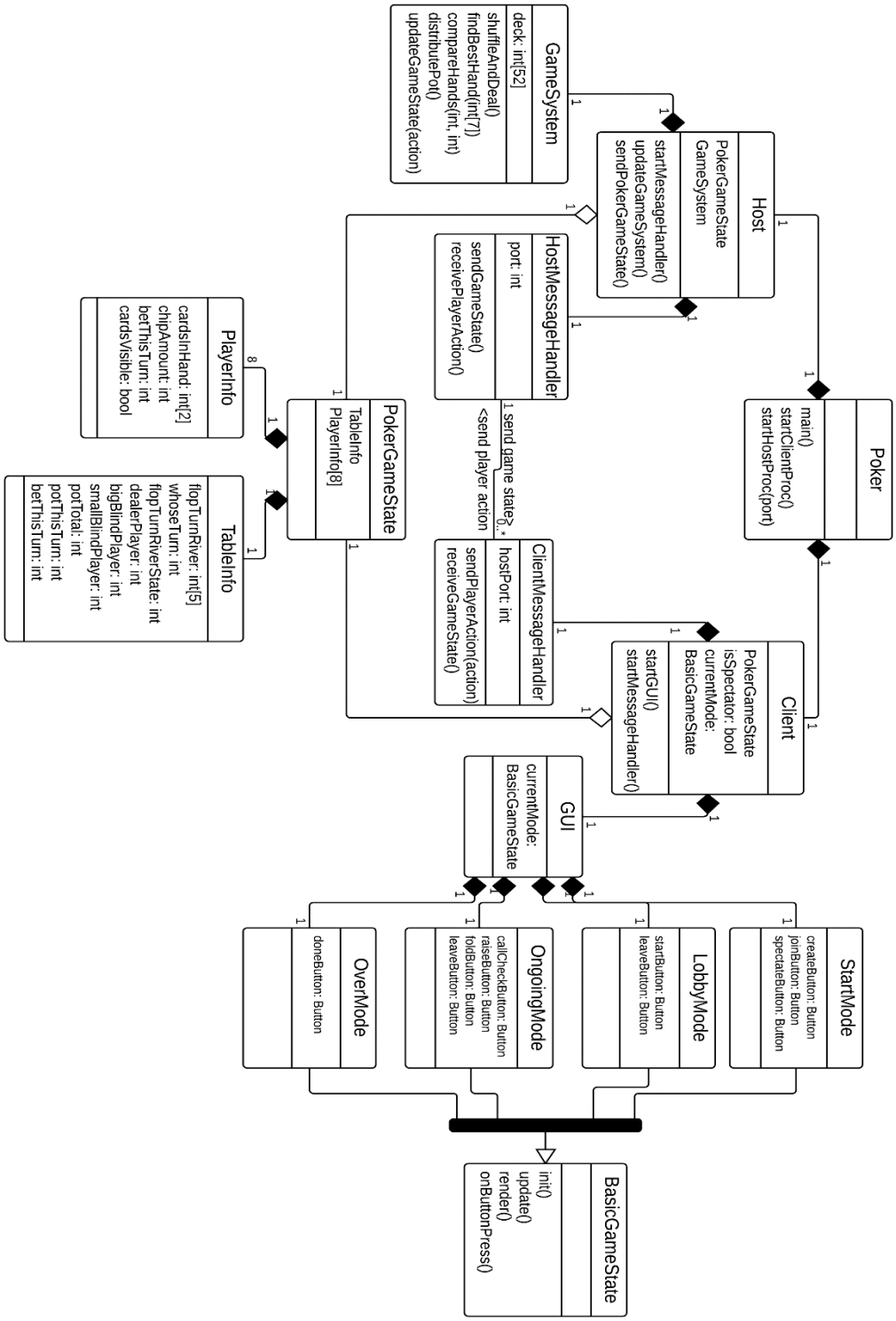


Figure3. Host Activity Diagram

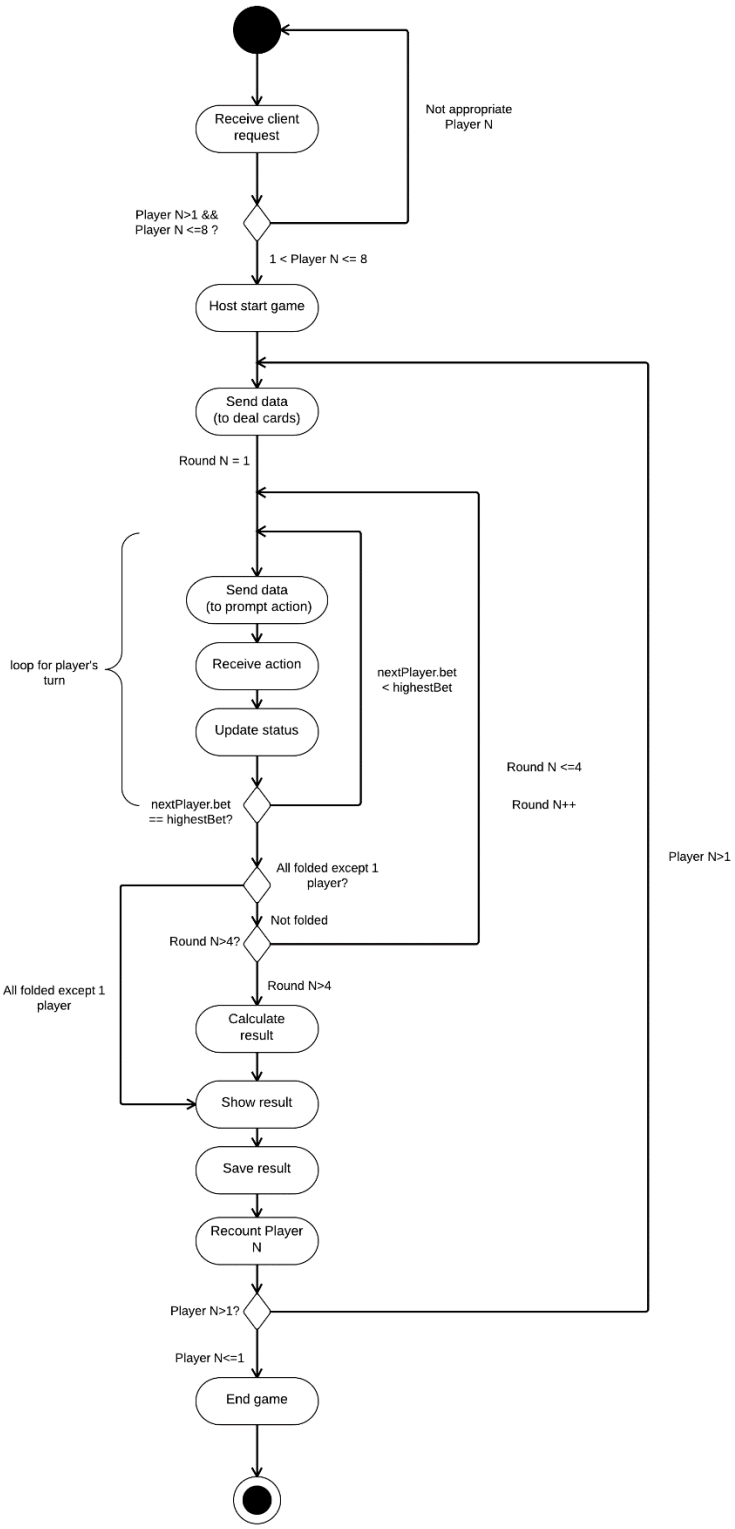


Figure4. Client's state when the program starts

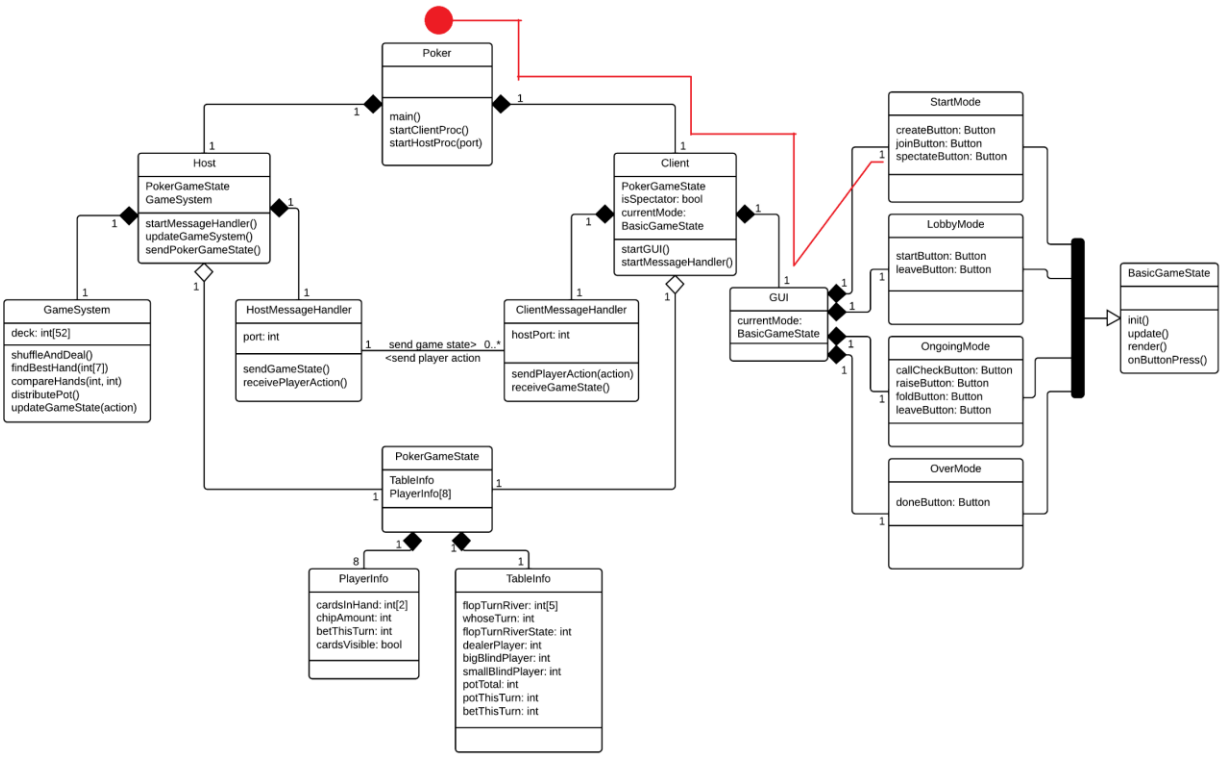


Figure5. When a client joins game as Player/Spectator

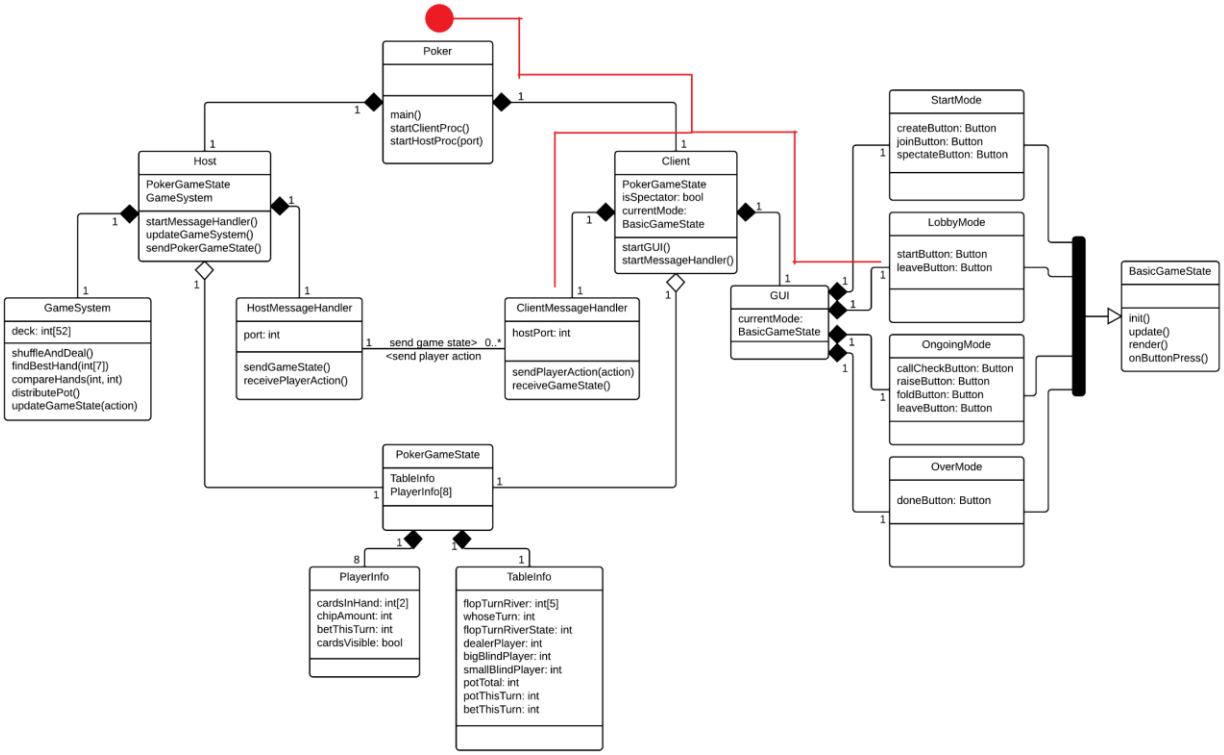


Figure6. When a client joins game as Host

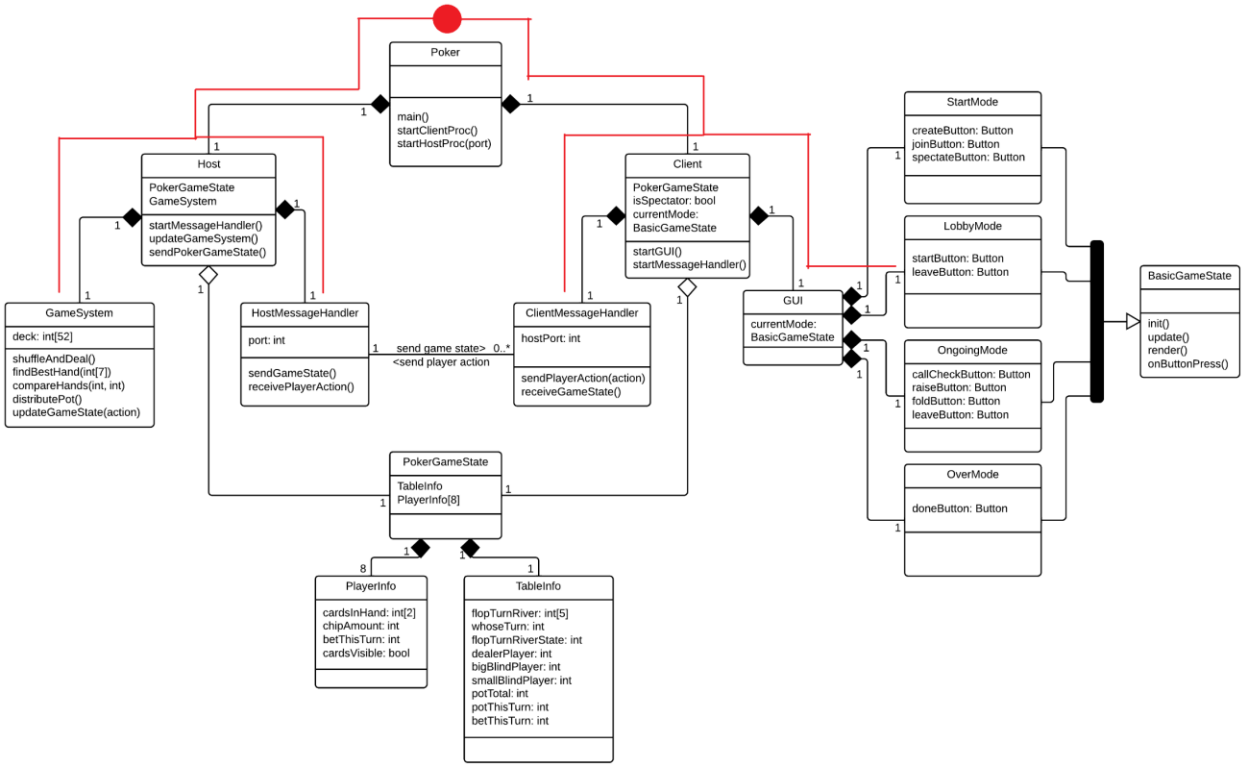


Figure7. Sample GUI when the game is in OngoingMode

Figure8. Host to Client interaction

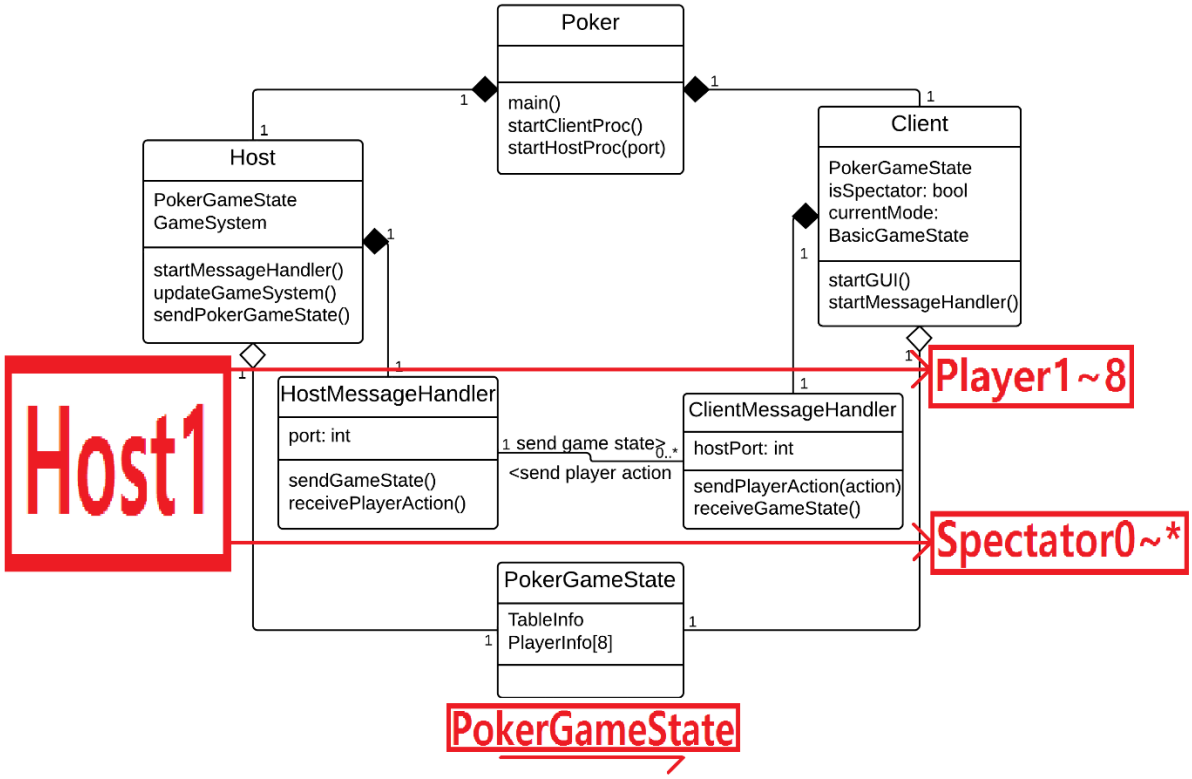


Figure9. Player to Host interaction