Poker game

(Sprint1 Retrospective)

https://github.com/srhee91/PokerGame

Team 3

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

Sprint1 Retrospective

1. Description of tasks

Implemented and working:

1. Create the *PokerGameState* class and implement all the classes and variables needed for the *PokerGamestate*: *PlayerInfo*, *TableInfo*, *Pot*, *Bet*, etc.

   We have successfully implemented the *GameState* class and its included classes according to our design document and class diagram. The hours spent figuring out the necessary components of the *GameState* class while making the design document helped us implement it later without much struggle. We weren't able to test the whole *GameState* class since we haven't implemented our *Host* and *GameSystem* classes yet. However, we were able to make test cases for each included class (*PlayerInfo.java*, *TableInfo.java* and *Pot.java*) separately.

2. Design the UI elements and their placements on the screens of the four game modes. No interactivity will be implemented, but the look of the four UIs will be completely finished.

   The layout of the UI elements was decided for each mode. One of the goals was to make the the GUI professional-looking. The background images and the poker-chip image were found on the web. Button colors were chosen from a UI-design color pallette found online, and the button images were generated by a button builder found online. The card images were extracted from Windows 7 Solitaire. The font used was the Windows UI font. Another goal was to make the GUI simple and intuitive. Each mode has few, highly-visible buttons. Pop-up messages are large and very basic. There are no complicated sub-menus or options panels.

3. Create the *ClientMessageHandler* class that implements Java Socket to connect to a specific port listened to by *HostMessageHandler*.

   In the constructor for *ClientMessageHandler* class, we take in an IP address of type *InetAddress* and a port number as arguments. We then create and initialize a *Socket* object socket that connects to the IP address and the port number by using methods of *Socket*.

4. Make the *ClientMessageHandler* class be able to continuously receive the game state from *HostMessageHandler*.

   An *ObjectInputStream* object will be created in the constructor of *ClientMessageHandler* and assigned the socket's input stream by using *getOutputStream()*, allowing it to read objects from socket. To receive a *GameState* object, a *ReceivingThread* is created and has the *ObjectInputStream* instance read the it with *readObject()* in a loop. The object is printed it out for debugging after the method returns.

5. Make the *ClientMessageHandler* class be able to send user actions through *Stream* to *HostMessageHandler*.

   After receiving an instance of *UserAction* to send, an *ObjectOutputStream* object is created in the *ClientMessageHandler* class. It will be assigned the socket's output stream by using *getInputStream()* so that it can write an object to the socket. To send the *UserAction* instance, we call the *send()* method. It uses the *writeObject()* method that accepts an instance of *UserAction* as an argument and writes it to the socket.

6. Create the *HostMessageHandler* class that implements *SocketServer* to listen on a specific port and be able to send the game state.

   When a host is constructed, a new thread is created for accepting incoming connect-requests from new clients. Whenever a new connection is established, the host a thread for the purpose of sending *GameState* objects to the client through *ObjectOutputStream.*

7. Make the *HostMessageHandler* class be able to receive actions from *ClientMessageHandler.*

   Whenever a new connection from client to host is established, the host creates another thread (in addition to the one mentioned in task 6) for the purpose of receiving *PlayerAction* objects from the client through ObjectInputStream.

8. Create test cases to check if the *HostMessageHandler* can communicate with up to 20 *ClientMessageHandlers* without a noticeable increase in latency.

   We created a *Test* class for this purpose. It first creates an instance of *HostMessageHandler* with a hard-coded port number as an argument to establish the server. Then, it creates 20 instances of *ClientMessageHandler* that takes in server's IP address and the port number as arguments for the *Client* to connect to host server. We verify the connection using print statements when a message is exchanged between the client and host.

Implemented but did not work well

N/A

Not Implemented

N/A

Extra tasks done:

1. Create the *Card* class to be used as a library.

   In the *Card* class, the constructor takes two int arguments representing the suit and value of the card.

2. Create *Deck* class that contains instances of the *Card* class. It has methods to shuffle cards, to burns cards, and to deal cards.

   In the *Deck* class, a constructor instantiates 52 *Card* objects. Then, these cards are shuffled. Following the rules of poker, it can deal 2 cards to each player, burn a card, or have cards removed to becoming the flop, turn, and river.

3. Create *CalculateRank* class that can determine the player with the best hand given each player's pair of cards and the five cards on the table.

   The main purpose of the *CalculateRank* class is to identify the five cards out of seven (player's two cards combined with the five on the table) that will result in the best hand. Once each player's best hand is determined, they should be ranked from the best hand to the worst. The code for ranking the players has not yet been implemented.

2. How to improve

1. Try every IP address in the subnet to find and list all the game lobbies in the local area network instead of asking users to enter host IP address manually.

2. Handle more exceptions. Make host/client message handler be able to retry or show error message if the network doesn't respond for a long time.

3. We should add game stats to OverMode or remove it altogether and use a pop-up message in OngoingMode when the game ends instead. Right now, OverMode simply informs the user that they have won or lost and has two buttons for the user to choose from. This could easily fit into a pop-up message in OngoingMode and would make much more sense. The only reason to leave it as a separate mode is if a lot of game stats will be displayed.