TEAM 1 - NETWORKING AND DESIGN

Networking

The networking solution we have chosen was the from the Chat Example given to the class by Howard, which was used for Assignment 1. Each ClientThread connects to a corresponding ServerThread through the IP address and port number of the Server machine. Both the server and the client have a *processInput* function that takes in a string and returns a string. The strings passed into the function are messages sent from the server to the client and visa versa. We believe that this networking strategy works best with our game design because it allowed us to simply append together strings and pass that over the network, and generate new strings in response. Therefore, the client or server can parse the received messages, take out the information needed for their end of the game functionality, and send a corresponding response back.

Design/Architecture

We decided to implement 3 different kind of patterns:

1) *Observer Pattern*

   This pattern was used to integrate the AI to the server. The server is an Observer that receives the messages from the AI, the Subject. We chose to implement the Observer pattern in this case because the Server could create an instance of AI. However, if the AI class were to create an instance of the server, we would have 2 separate concurrent Game Engines running at the same time, which would defeat the purpose of having an AI player. Every time the Server receives something from the Game Engine, it sends it directly through the instance of the AI class. However, in order for the AI to communicate back with the Server, it has to notify its Observers.

   This pattern was also implemented for the GUI to the GUI Controller, with the GUI being the Subject for the GUI Controller to Observe. We chose to do this so that the MainWindow Class, which contains main GUI display, could notify the MainWindowController Class when a button click is made. This will return a signal dictating whether the user wants to see or play a card, see a player's display, end their or quit the game. Using the information gained for receiving a message from the server, the GUI Controller can do one of two things; it can edit what the user sees displayed, or the controller can translate that click into a message that reflects what has been played, or if the player has terminated their turn or game. To deliver the latter option, we utilize our final instance of the observer pattern where the GUI Controller is now the Subject and the Client is the observer. By doing this the Client, which created the GUI Controller, can now send the results gained from the user input that have been translated by the GUI Controller into an action. This action can then be sent to the server to be interpreted as an action taken by the player in the game. We set up the GUI flow in this way so that at each part in the creation of the objects, there is only one instance of a Client, one of a GUI Controller and one GUI, for each player. This also serves to limit the information that each class contains, while allowing the classes to communicate with changes from the client to the GUI, and subsequently from the GUI to either the GUI Controller or the Client as required.

2) *Facade Pattern*

The client and server side game implementation was refactored into facade patterns, which attempted to mirror each other's functionality for smooth integration and communication, without allowing the classes responsible for communication between the client and server handle the game's logic and functionality directly.

On the server side, the server class only knows about the ServerProcessor class, which analyzes the input messages from the client side, and sends it to the necessary function in the GameProcessor class. The game processor class then handles all of the the output processing necessary to communicate with the client side, and updates the functionality of the game engine, before sending the corresponding output back to the ServerProcessor. None of the processing implementation is visible to the server. There was however the need for a getGame() function in the GameProcessor, in order to test all of the functionality properly.

The client side refactoring was a little more difficult, as there are a lot of moving parts. The main MainWindowController, which is responsible for updating the GUI, communicates with the Client class directly. However the implementation for processing information from the server is handled in the ClientProcessor class. This is called only once, from the Client class's handle function, when a message comes in from the server. The ClientProcessor then analyzes the input, mirroring the functionality of the ServerProcessor class, and calls the necessary functions in the GUIProcessor to generate the corresponding output update the GUI window. There remains some functionality in the client class for playing a card, which controls the GUI but does not handle any of the client to server communication.

3) *Strategy Pattern*

This pattern was used to implement the AI. Each AI inherits from the Player class. Once a new instance of the Player class is created, it initializes a new strategy which extends the Strategy Interface. There are 2 different kind of strategies used in our program:

1. Withdraw Strategy: the AI will choose the tournament colour based on their first card and then withdraw after their first turn.
2. Play All Strategy: the AI will choose the tournament based on the number of each coloured card that the AI has. It will play all of its possible cards, including its supporters and stop playing until it has no more cards to play. If the AI does not have enough of each coloured cards it will choose a tournament colour based on the token that it needs, as long as it has supporters. If the AI wins a purple tournament, it will only choose a purple token rather than choosing the token it needs. The default choice of tournament is green but once the AI has received its cards it will change the choice of tournament colour to the colour that it has the most of.