Project Specification - Uno

Names: Ritam Chakraborty, Srushti Chaudhari, Angela Chung
Period 2

Overall Description:

Uno is a simple card game beloved by many. It's very popular among all ages, and thus we decided to build our own computer Uno game using Java.

Our project supports and highly encourages multiple players from different systems at a time. In order to make this function, we have to host a server for our clients to connect to. We use the java.net library to create our communication server, which in this case is a socket. That means only people from the same network as the server can play with each other.

After the server is hosted, clients are able to join as long as the server is running. The server is constantly listening for players that will join, and once one does, it will create a new object that is able to handle the client on a new thread. This object and the client can effectively read and write data to each other.

After this new handler object is created, the client starts a GUI interface for the user. It prompts the user by asking them for a username. Once they enter that information, it gets sent to the handler object which is able to process the request. In this case, it would check whether the username is valid or not, and if the game has already started. If the client is able to join the waiting room, it tells the client that, which then updates its GUI to reflect that. Once multiple clients have joined the waiting room, the leader is able to start the game whenever they want.

Our game is nearly identical to Uno in terms of rules and functionality. We use all the same cards, and they have all the same effects. When users play their cards, that information is sent to the handler object which updates a reference to a game state object which contains information about the game. The only difference is that instead of saying "Uno" when you have one card remaining, an uno button pops up randomly on the interface. Whoever is first to press it gains its effect.

Once a player is able to win, they are sent to a victory screen where they learn what place they got. The other players continue to play until there is all but one left, who is proclaimed the loser. To play again, the players must run their client program again, however the server can continue to run, and play as many games as needed. Ideally the server would be hosted in a way where it wouldn't have to be terminated everytime you close your computer, such as a server room.

Class Overview:

Networking Classes:

Server - A class that is run separately to listen for clients joining the game room. Has information about its port number, IP address, and number commands that are used to send and receive data in the form of strings which indicate what type of data is being sent. The server client setup was inspired by WittCode on YouTube.

ClientHandler - A class which has a thread running on it when the server connects to the client. The instance of the ClientHandler listens to data sent by the client and based on its contents, updates a static reference of the GameState class in order to keep track of what is happening in the game. Also holds a static reference of a list of all the other ClientHandlers in case it wants to send data to everyone.

Main - The main class of the program which represents a client. Connects to the server, and then starts a thread that listens to information from its handler. Starts a new GUI thread as well which creates an instance of GUIHandler.

Card Classes:

Card - An abstract class that defines the basic layout of a card. Has an abstract method pertaining to its ability to be played on top of another card. Subsequent Child Classes will usually override this method to accommodate for different properties. Furthermore, it has a couple static utility methods to convert a string form of a card to a Card object, and another to convert a list of cards into a list of string forms of those cards.

ColorCard - An abstract class that extends Card and represents a Card that has a color property attached to it.

NumberCard - A concrete class that represents a ColorCard that has a number property attached to it. The playable method is overridden to test whether the color or number of the previous Card and the current Card are the same.

SpecialColorCard - An abstract class that extends ColorCard and represents a ColorCard and holds a reference to the type of special ability it has in the form of a string. The playable method is overridden to test whether the color or effect of the previous Card and the current Card are the same. Children of this class aren't able to enact their ability within this class itself, but later when they are referenced in the GameState class.

ReverseCard - A concrete class that is a SpecialColorCard with the ability to reverse the order in which Players play.

SkipCard - A concrete class that is a SpecialColorCard with the ability to skip the next player's turn.

DrawTwoCard - A concrete class that is a SpecialColorCard with the ability to force the next player to draw two cards, and skips their turn.

WildCard - A concrete class that is a Card that has a color property attached to it, but not when initialized. Instead it is set when a user plays that card.

DrawFourCard - A concrete class that is a WildCard that not only sets its color, but also forces the next player to draw four cards, and skips their turn.

Deck - A class that extends LinkedList<Card> and overrides its methods that are implemented from Queue<Card>. Its constructor creates a deck of cards, the standard one in a game of Uno. However, instead of the remove() method of a Queue that returns items in a First-in-First-out fashion, it returns and removes a card at a random index in the list. This way the deck gives off the illusion of being shuffled, but instead of being added randomly, they are removed randomly.

Game Logic Classes:

GameState - A class that contains information about a single game of Uno. Holds a reference to the Deck class which is implemented as a Queue, a Stack representing the discard pile, and a list to keep track of the current players. It keeps track of the current player, and adds cards to the hands of players, or removes them when asked for, and when legal. It also contains private methods such as reverseTurns() and skipTurn() to simulate the effects of SpecialColorCards when played.

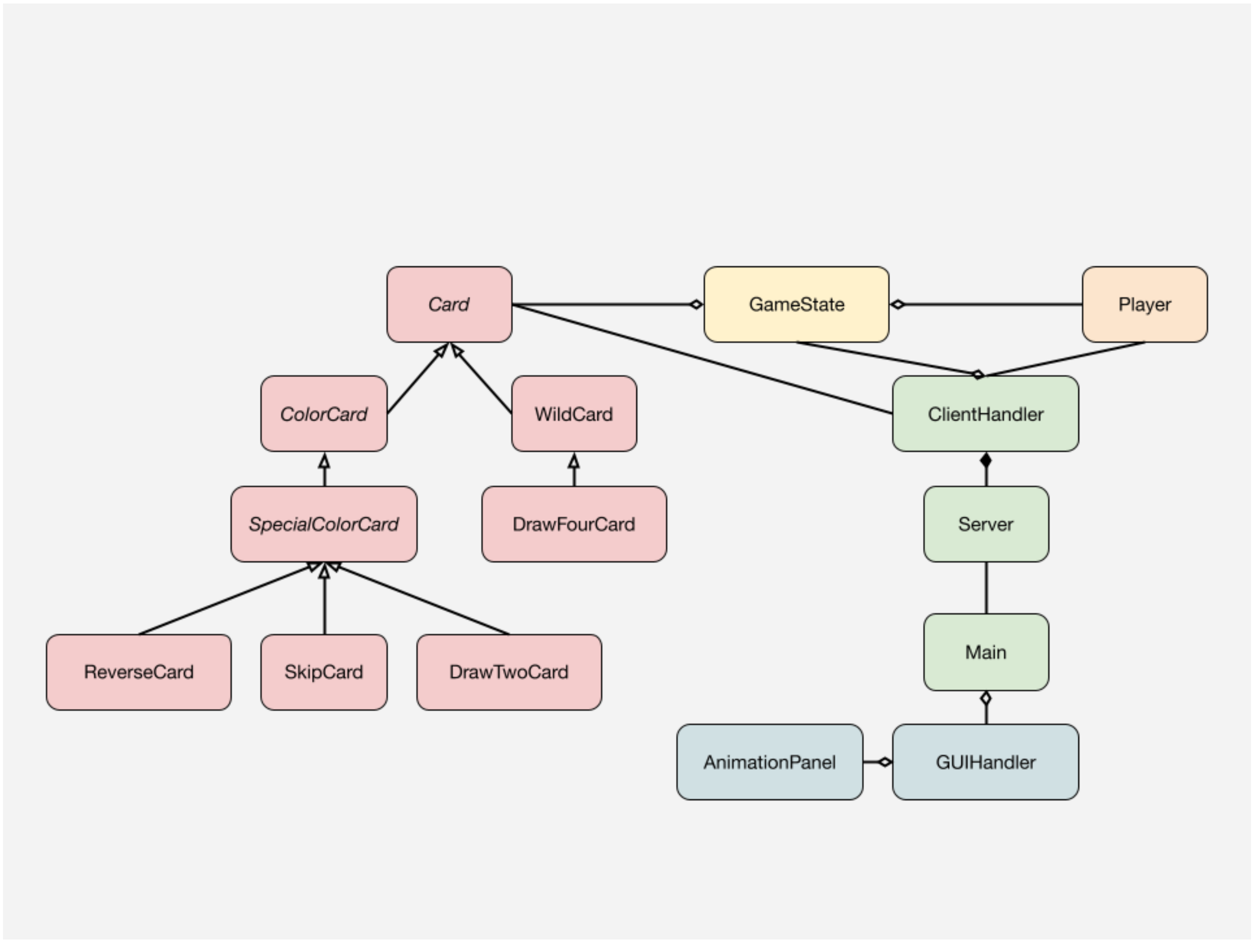
Player - A class that contains information about a singular player. It holds information about their hand, username, and what place they are in the group of players.

GUI Classes:

GUIHandler - A class that represents the GUI of the client program. The GUI is started in the Main class after it has connected to the server. The Main class receives messages from the server, and the data is decoded in the GUI class. Once that data is decoded, the UI is updated. If the UI is interacted with, it will send that information to the ClientHandler associated with its Main class directly from this class.

AnimationPanel - A class that extends JPanel and is able to play animations for certain actions such as when cards are dealt out and drawn, and when cards are played.

Rough Class Diagram:



Structural Design:

Main Data Structures Used:

Description	**Structure**
ClientHandlers for all clients	LinkedList<ClientHandler>
Deck of cards	Queue<Card>
Discard pile of cards	Stack<Card>
List of players in GameState	LinkedList<Player>
Hand of a player in Player	LinkedList<Card>
Hand of a player in GUIHandler	LinkedList<JButton>, LinkedList<String>
“List” of players in GUIHandler	LinkedHashMap<String, Integer>

Data Structure Rationale:

We need to hold a list of all the ClientHandlers because sometimes we want to send messages to all the clients at the same time. Having a static reference to all of them is useful. We need them in a list because sometimes we need to access the first element of the list when deciding the leader of the party. Other than that, we don't use random access and do a lot of removing and adding from the list which is what LinkedList specializes in.

A deck normally is shuffled, and then cards are taken from the top of the deck, and when needed, added again to the bottom. This is an excellent opportunity to use a Queue, however we can take it a step further. Instead of coming up with an algorithm that shuffles the cards in the deck, we decided to make our own modified version of a Queue. Instead of returning in a FIFO fashion, it returns elements randomly which makes it look like the deck is always shuffled. If we need a new card from the deck, we can just call the remove() method, which will return a random Card in the deck, exactly what we need.

In a discard pile, the only card we need to refer to is the top, since it will be compared with cards clients try to play on top of it. Since we only need the top card, a regular stack is perfect since access to the top of the stack is extremely simple. The only other thing we do with this stack is clear when the deck is empty, which takes the same amount of time for any data structure, so it doesn't affect our choice.

We need to use a list for our players solely because of the fact that the ReverseCard class exists. The special ability of this card is to reverse the direction the turn is going. To accomplish this, we have to switch up the elements in our list, and reset our ListIterator. Moving the elements around like that would be woefully awkward in any other data structure, near forcing us to use a list. The reason we use a LinkedList over an ArrayList is because we often remove players from the middle of the list.

We are also forced to use a list for the player's hand. We would love to use a Set to store the cards, however unfortunately in Uno, there are multiple of each type of card. This completely stops us from being able to use Sets to store cards at any point, and we are forced to use lists. Again, a LinkedList is used because we remove cards from the middle of our list all the time.

Storing the player's hand in the GUIHandler class is interesting. We use LinkedLists for the same reason we do in the Player class, but we require two different lists. One holds JButton objects that are initialized with an ImageIcon from our image files. These buttons are displayed on the UI. But we require a second list as well to hold the string versions of those cards. This is because if we want to tell the server that a player wants to play a card, we need to tell them what card they want to play. However, getting that information from the JButton is not possible since we can't get the image path from the ImageIcon after we pass it. We have to store it somewhere else, and at that point, we can just store it as Strings, the data type we want to send instead.

The only information that a player needs to know about other players in our version of Uno is their name, and the amount of cards they currently have. Since there is no worry about having multiple of the same card, since we aren't working with them, there is no reason to not use a Map, since in this case we are working with two pieces of data. A HashMap isn't ideal as the iterator returns the players in a random order, the list of players on the screen would be different for everybody. We could use a TreeMap, which would return the player names in alphabetical order, but even better is a LinkedHashMap. This returns the values in the order they were added, so everyone will be able to see what players entered, so our list of players won't randomly change order every time someone joins.

High Level Major Class Specifications:

The following shows only important information about each of the classes

Server

- Attributes
    - ServerSocket serverSocket
    - Socket socket
    - int PORT_NUM
    - String IP_ADDRESS
    - Around 20 public static final String fields that denote the type of information sent between the server and the client, used for decoding and encoding data

ClientHandler

- Attributes
    - LinkedList<ClientHandler> clientHandlers
    - GameBoard board
- Methods
    - void sendCards(String username, LinkedList<Card> newCards)
    - void newDiscardPile(Card topCard)
    - void run()

Main

- Attributes
    - Socket socket
    - DataInputStream in
    - DataOutputStream out

GUIHandler

- Attributes
    - String myName
    - DataOutputStream out
- Methods
    - void errorScreen()
    - void decode(String allStrData)
    - void actionPerformed(ActionEvent e)

GameState

- Attributes
    - Queue<Card> deck
    - Stack<Card> discardPile
    - List<Player> players;

- Methods
    - boolean addPlayer(String username)
    - getPlayer(String username)
    - void removePlayer(String username)
    - void playerWon(String username)
    - int getPlaceOfPlayer(String username)
    - void advanceTurn(Card card)
    - void startGame()
    - Player getCurrentPlayer()
    - boolean gameHasStarted()
    - String getPlayerList()
    - Card getLastPlayedCard()
    - boolean play(Card card, int index)
    - Card draw()
    - int getTotalPlayers()
    - void saidUno(String username)
    - void endGame()

AnimationPanel

- Attributes
    - BufferedImage bg
    - int yOffset
    - int xOffset
    - int yStep
    - int xStep

    - boolean movingCard
- Methods
    - void moveCard(String cardImage, int fromX, int fromY, int toX, int toY, int stepInterval, int totalStep)
    - void moveCard(String cardImage, ArrayList<Point> fromList, ArrayList<Point> toList, java.util.List<Component> components, int stepInterval, int totalStep)
    - void paintComponent(Graphics g)

AnimationActionListener
- Attributes
    - final ArrayList<Point> fromList
    - final ArrayList<Point> toList
    - java.util.List<Component> components
    - int finishedRound
    - boolean picked
    - int totalStep
    - int total
    - int fromX
    - int fromY
    - int toX
    - int toY
- Methods
    - void actionPerformed(final ActionEvent e)

Deck
- Attributes
    - int nextCardPos
- Methods
    - Card remove()
    - Card poll()
    - Card element()
    - Card peek()

Card
- Attributes
- Methods
    - String toString()
    - abstract boolean playable(Card card)
    - static Card decode()
    - static String listToString(List<Card> cards)

        - abstract Type getType()

ColorCard
- Attributes
    - final String RED
    - final String BLUE
    - final String YELLOW
    - final String GREEN
    - String color
- Methods
    - boolean playable(Card c)
    - String getColor()
    - boolean equals(Object other)
    - String toString()

SpecialColorCard
- Attributes
    - String effect
- Methods
    - boolean playable(Card c)
    - String toString()
    - boolean equals(Object other)

WildCard
- Attributes
    - String color
- Methods
    - boolean playable(Card card)
    - void setColor(String color)
    - String getColor()
    - String toString()
    - Type getType()

DrawFourCard
- Attributes
- Methods
    - String toString()

DrawTwoCard
- Attributes
    - final String EFFECT

- Methods
    - Type getType()

NumberCard
- Attributes
    - int number
- Methods
    - boolean playable(Card card)
    - boolean equals(Object other)
    - String toString()
    - int getNumber()
    - Type getType()

ReverseCard
- Attributes
    - final String EFFECT
- Methods
    - Type getType()

SkipCard
- Attributes
    - final String EFFECT
- Methods
    - Type getType()