Srushti Chaudhari, Ritam Chakraborty, Angela Chung

AP Computer Science A

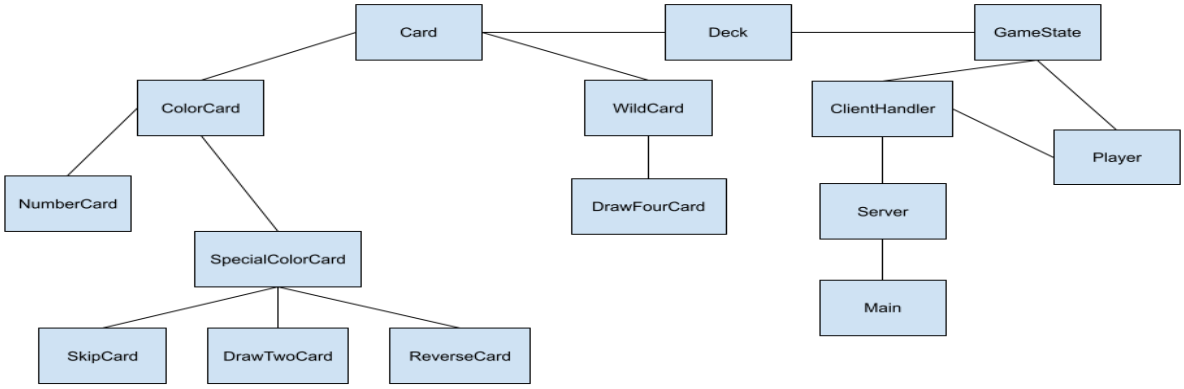
Mr. Fulk

4 May 2022

Uno

Our project is intended to simulate a game of classic Uno which incorporates the multiple player aspect through networking. Java GUI is included to display the game across multiple devices with a user-friendly interface. To implement the game, players can join from different consoles and battle each other based on the cards available to them. The game involves updating real-time information sent by players and verifying that a play is valid, in addition to handling rare anomalies.

Uno Class Diagram



The Card abstract class represents and holds the fields and shared methods of each subsequent Card that extends it. Each Card object maintains a color field and toString() and getType()

methods, which return String representations of the Cards and the type of Card it is. Enumeration is used to create Type classes and each Card stores its own Type, and is accessible from other Classes. Specific Cards have their own properties (ex. NumberCard uses an integer number field, Reverse Card causes the turns of players to change), and Cards with special effects (such as forcing a player to skip a turn) will be implemented in the GameState Class. NumberCard represents a standard card with a number and color that does not generate any additional effects when played . SkipCard forces the opposing player to skip their turn, and ReverseCard reverses the order in which players make their choices. The DrawTwo and DrawFour Cards requires the opposing player to add 2 and 4 cards respectively to their current hand. While DrawTwo contains a color, DrawFour does not. The WildCard is created without color and once played, the current player decides which color it will represent, and the WildCard's color will be updated based on the information. Furthermore, each Card contains a playable method that takes in a Card parameter, and returns true if the Card is playable based on the input, and false if it is not.

Uno uses Java sockets to send data to clients and maintain the game state. The Main class or client class connects to our Server class, which then creates a ClientHandler instance that is able to communicate with that main class. Each ClientHandler instance holds references to the input and output streams and the socket (the connection port) for a specific client, as well as a Player object that holds information about this specific player, attached to the client. In addition, the ClientHandler class holds two static fields, one being the game state, stored in the GameState class, and the other being an ArrayList of other ClientHandler instances, which is used to send message to all other clients at once, when the state of the game has been updated. The ClientHandler implements the Runnable interface, and thus implements its run method, which is executed when a new thread of this object is created. From that method, the input is collected, and will be dealt with in various different ways depending on the nature of that data. When this object is first created, the constructor initializes its streams and sockets, and gets the name of the new player from the client. A player object is then created, and added to the game state. Finally, all other players are notified of a new player joining.

The GameState class' function is to hold and update the state of the game. It itself doesn't carry

the task of notifying clients of updates. It holds a modified Queue of cards through the Deck class, which is used to simulate the shuffling of the deck. It holds a Stack for the discard pile as cards are only put on top of the discard pile, and it would be relatively easy to find out what the top card is, which will be used to determine if a user's card can be played. The game state also holds a LinkedList of players, as players will mainly be added to the end of the list, and removed from the middle, which is a LinkedLists specialty. A ListIterator is used to keep track of the turn of the current player. A reference of the currentPlayer is stored in a Player object, so it can easily be updated, and a boolean field is kept to see whether the actual game has started or not. So far, the GameState is able to add players to the game, and “start” the game. The game however, isn’t playable yet. When the game has not ended and there are no more Cards left in the deck, the update file sets the deck to the discardPile.

The Player class stores properties related to the specific state of the player. It holds a String for the username, and a List of cards for their hand (the Cards in their possession) , which will likely be implemented as an ArrayList, due to the constant need of random access, to determine whether a card from their hand is playable or not. The class has a play method, which takes in the previous Card played and the index of the Card the player wishes to play. It then returns the Card they will play if it is playable based on the previous Card, or null if it isn’t.

The Deck class extends the LinkedList class (holding Card objects), and overrides some of the methods of the Queue interface in order to return cards in a “shuffled” order. A regular queue outputs the first item entered, or a first-in-first-out system. However, our queue instead outputs a random Card every time the remove method is called, so it is automatically shuffled. When instantiated, it creates 108 cards objects, that are the 108 cards in a standard deck of Uno.

The main class uses Java Swing for a GUI by extending the JFrame class and implementing ActionListener for real time changes. Multiple panels are used to display information, where the top and bottom panels are formatted as a FlowLayout. Our main class acts as a client as well, which will constantly read and send information to update the game state, and to update its own GUI components. A new thread is created for listening to messages, and a Java Swing thread is created to update Swing components and listen for user input. We make sure to use a

SwingWorker when updating front end components so the two threads do not intersect. Aside from containing various GUI components, the Main class saves a static reference to itself, which is needed to run the listening thread from the main method. There are the streams and sockets created as well, as described in the ClientHandler class, and we used a LinkedHashMap of Strings and Integers to store outside player information. The only information that we need of the other players is their names, and how many cards they have. This information fits nicely into a HashMap. However, we need the LinkedHashMap to save the players in the order they join the game, which is the order they are added, so when a new player joins, it doesn't completely switch that information up in the GUI. A LinkedHashMap will iterate through itself in a more controlled manner, which matches the way we want the information to be displayed.