Poker

-Arvind Ram 2020A7PS1210P
-Rishabh Kumar 2020A7PS1211P

Description Of Classes

1) *Card*

   The Card class holds details and describes an individual card in a deck, i.e, the suit and value of each card. A numerical value has been assigned to each card in order to compare them during the game

2) *Deck*

   The deck class is composed of an array of Card objects. Hence, it helps store all the cards in a deck in an array and also shuffle them whenever necessary. The deck class also consists of methods to deal the community cards and the hole cards for each player.

3) *Player*

   This class holds the details of each player. This includes player names, chip count, bet, and position in the table. The class also creates objects of the Card class to store each player's hole cards and their best hand (the best 5 out of the 7 cards available).

4) *StartPage*

   The Start Page class is the first GUI frame, visible to the user. The jframe consists of a start and quit button. This class also declares an array of player names and an integer holding the number of chips that will be used in further classes.

5) *PlayerDetails*

   This is the next jframe visible to the user where he/she can enter the names of 2-4 players. PlayerDetails inherits the StartPage which helps store the player names in the array created earlier.

6) *ChipDetails*

   The ChipDetails class inherits the StartPage.It is a jframe asking the user to input the number of chips every player should start the game with. This value is stored in the integer chips created in the StartPage class.

7) *PlayScreen*

   This class inherits the StartPage class as well. With the details of player names and chip count, it can create an array list of player objects and then start the game. The usual flow of the game, comparing every player's hands, displaying cards, distribution of winnings etc, are taken care of by this class.

Limitations

1) Since the program assigns a small blind and big blind to two players automatically, it requires at least 3 players.
2) Incase of a draw, the code checks the highest high card, instead of checking which hand is better.
3) The complete application does not use GUI. The regular functioning of the game happens on the console. The game required a loop to run to keep the game going for multiple rounds. However, in doing so the main thread was busy evaluating results and taking in input. Since the GUI runs on the same thread, it couldn't run to display the contents of the JFrame. The solution to this is implementing a java SwingWorker. However, further knowledge was necessary to implement this class. Due to the paucity of time, implementing the class wasn't possible in this application.

Future Work Possible

1) The application could make use of the java SwingWorker class in order to make it more interactive.
2) A bot could be implemented in order to support single player.
3) The program could be expanded to allow more players to play at once.
4) The application could be tweaked to allow online multiplayer as well.

OOP Principles

1) *Classes should be open for extension and closed for modification.*

   Most classes in the program are open for extension. For example, the Player class can be easily extended to hold more attributes of the player like age, ranking etc and implement more methods and this can be done without modifying any of the already existing methods or attributes. The same applies for the Card class which can be extended to to hold images of the card. Similarly all the classes in the application are well formed and defined which makes it possible for them to be extended without modification.

2) *Encapsulate what varies*

Most of the application code has been well encapsulated to ensure no redundancy. For example, the presence of methods to find the category of the hand makes it simpler to evaluate each player's best hand and compare the best hands of multiple players. Another example is the option method in the PlayGame class which asks the user to input his choice and since this option should be given to players every turn it has been added to a method. However, there are parts which have not been implemented perfectly. The raise methods have similar functionalities but the code for each of them has been written separately resulting in repetition of code. Hence, even though most of the program is encapsulated there are parts which can be improved to make it even more modular.

3) *Depend on abstractions not concrete classes*

Abstraction enables reusability. In our code we made a concrete class called Card and made objects to access each of the 52 cards. Instead using an abstract class Cards would've helped save space and made the code more efficient.

4) *Favor composition over inheritance*

The given application makes use of composition in a number of places.

a) In order to create a list of player objects, the program creates an ArrayList of the objects in the PlayGame Class rather than inheriting the Player class.
b) The Deck class makes use of composition to create objects of the Card class and form an array of 52 cards to fill the deck. Composition is also used to create the array of hole cards and community cards.

Hence, the given program favors composition over inheritance.

Design Principles

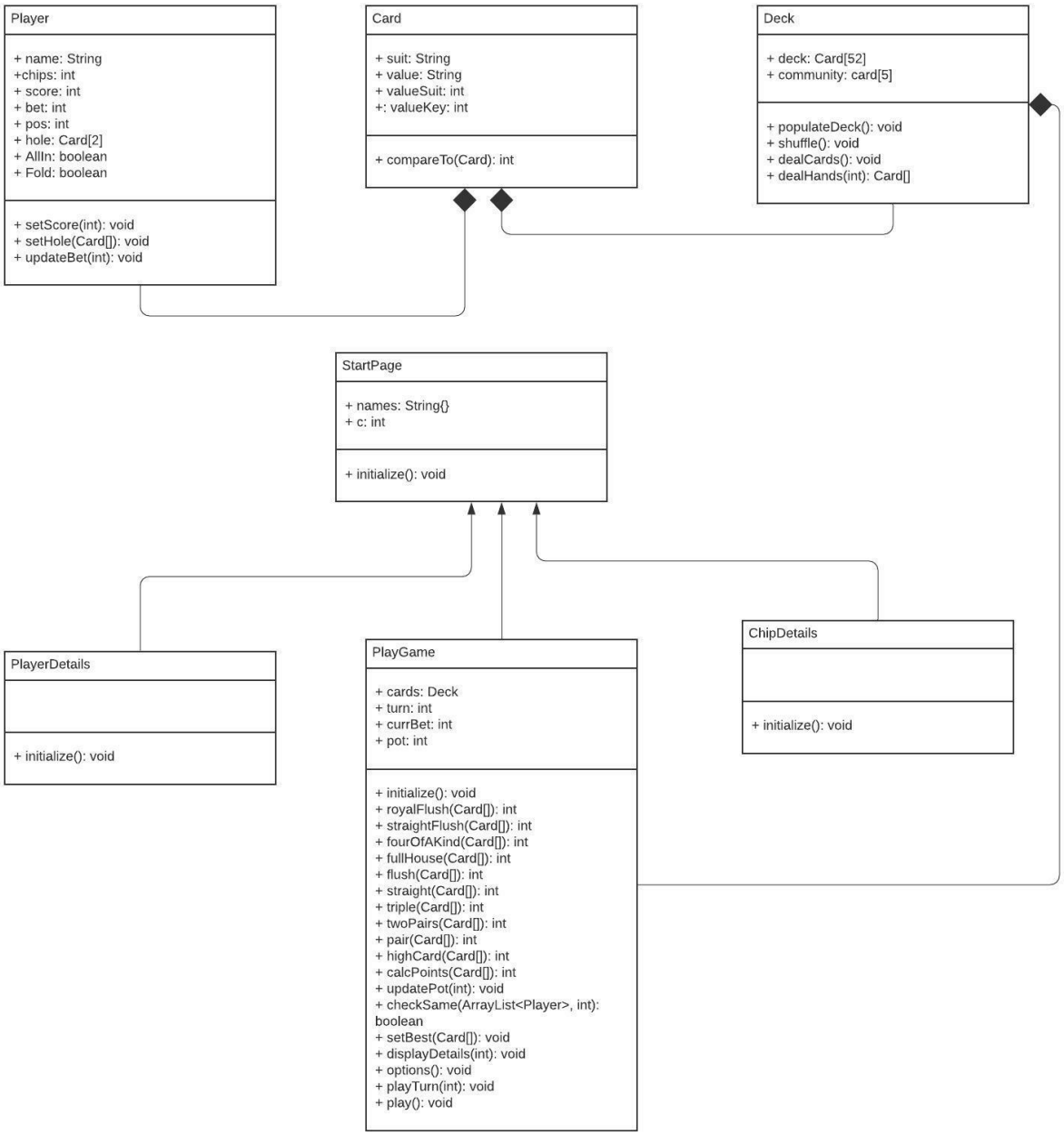
1) *Observer Pattern*

Observer pattern helps in modifying the dependent objects when a certain independent object is modified. On analyzing the code to check for observer patterns we see that there are several parts which do not follow the design. For example, when a player places a bet the player's chip count should decrease and simultaneously this value must be stored in the pot. To implement this we are accessing the player objects everytime and updating the pot every player's turn. In doing so we are prioritising implementation over interface. Hence, to solve this the issue we can add the following interfaces:

a) Observer - This will check when a bid has been made by any player.

b) Subject - This will notify all the dependents, i.e., it will update the player chips and the pot amount. This way both the interfaces will be loosely coupled ensuring that the application would run more efficiently.

UML Class Diagram



Link to GitHub Repository:
https://github.com/k-rishabh/oop-poker