Poker Analysis Document

## Class Descriptions

### Card

The Card class contains attributes of suit and rank. Each object of this class represents a specific card in the deck.

It implements the comparable interface that allows us to sort an ArrayList of cards in order of their rank. This becomes useful when the Hand class has to sort the cards and figure out which hand has more value.

### Deck

The Deck class creates a stack of 52 card objects. A stack is the most appropriate data structure for this usage as only the first card needs to be accessed at each instance, random cards need not be accessible to any class for the purposes of the game.

The methods of this class are:

Deal: The deal method is overloaded to allow the deck to deal the community cards as well as hole cards to each player. When the argument provided to the method is an object of the Player class, it adds the topmost card of the stack to the player’s hole cards and removes it from the stack. When the argument is an arraylist, which in this case is communityCards, it adds the topmost card to the arraylist and removes it from the stack.

Shuffle: The shuffle method calls Collections.shuffle() on the stack, which randomly changes the order of the cards in the stack.

Burn: The burn method removes the topmost card from the stack.

### Player

The player class has name and currentChips attributes which are taken as user inputs when the player joins the table. The currentChips attribute is updated through the methods of the class itself. It keeps track of the chips that the person has left. It has an arraylist called holeCards that contains the Card objects dealt to the player by the deck. The playerBet attribute keeps track of the amount that is bet by that player in that round. The isAllIn boolean acts as a flag that tells that the player is no longer left with chips. The hand attribute is used to store the Hand object that includes the player’s hole cards and the community cards.

The methods of this class are:

Call: This method allows the player to bet the Table’s currentBet amount. It deducts the amount from the player’s chips, and increases the pot by the same amount. If the player does not have sufficient chips, the player is marked as AllIn and all their chips are added to the pot.

Check: This method allows the player to not bet any chips, given that no player on the table before them has placed a bet. It sets the playerBet to zero.

Raise: This method allows the player to bet more chips than the currentBet of the table. It takes an additional integer input and places a bet of those many chips. It deducts the amount from the player’s chips, and increases the pot by the same amount. It also sets the currentBet of the table to the increased amount, as all other players should need to match this bet to stay in the game.

Fold: This method removes the player from the current round of the game.

### Hand

The Hand class implements the comparable interface. It contains the code deciding whether a certain set of cards matches any possible hand value, such as flush, straight etc. Each hand value is a boolean that returns true when that combination is present in the hand. The class assigns a worth to the hand depending on the most valuable combination of five cards present. The comparable interface allows us to sort the Hand objects in order of their worth, by providing the compareTo method.

### Table

The table class runs the entire game. It has attributes to keep track of all the variables that affect other classes, such as currentBet and pot. It keeps track of total players on the table as well as roundPlayers and currentPlayer.

The methods of this class are:

printCard(Card c): It prints the rank and suit of the card.

promptPlayer(): This method prints the name of the player that has to select an option and displays all the information they might need to place an informed bet, such as currentChips, currentBet, holeCards etc.

recordBet(): This method takes an integer input from the player and calls the player method that corresponds to the input. It also increments the currentPlayerIndex, so that the current player changes to the next player on the table.

catchChecks(): This function ensures that all players who are not AllIn match the highest bet made in that round of betting. It does that by checking if all players that still have chips have bet the same amount as each other. When it finds a player that does not meet this criteria, it prompts the player to bet more.

revealCard(int n): This method adds n cards to the community cards arraylist, and then prints them to the console.

declareWinner(): This is an overloaded method. When no arguments are provided, the method compares the hands of various players and finds the winner or winners. It distributes the pot amount to the winners, if there are multiple players with the same hand worth. It then prints out the names and amounts won by the winners.

When a player is provided as an argument, the method declares this player as the winner and assigns the pot value to that player’s currentChips. This is called when all players, except one, fold. This ends that round of the game.

## Input/Output Explanation

When the program is run, it prompts the user to enter the number of players in the game. It reads an integer input, then asks the user to input the name and currentChips for each player.

It then prompts the first player to input their bet, by printing the players name, currentChips and holeCards. It also prints out the revealed communityCards, pot, currentBet and the key that the player has to use to give their input.

It then scans the input from the player and runs the corresponding function. It then prompts the next players for their bet. This continues till all the communityCards are revealed.

Then, the program prints out the winner/winners of the round and the amount of chips they have won.

It asks the user if they want to play more rounds of the game, and depending on the user’s response, runs another round of the game of prints a closing message.

## Analysis of OOP Principles

### Encapsulation

Encapsulation uses getter and setter methods to access private attributes of certain classes. This prevents the attributes from being misused or modified by accident.

The code submitted does not include encapsulation, all attributes of the classes are visible to other classes. This puts the programmer at the risk of accidentally reassigning a variable that only had to be accessed.

For example, this principle could have been used for the Card class, since rank and suit only have to be assigned once, and accessed multiple times.

### Composition over Inheritance

Composition over Inheritance means that classes should achieve polymorphic behaviour through their composition instead of inheriting the methods or attributes from a parent class. This makes the classes more independent, and provides more flexibility to the programmer.

The code does not use inheritance. Composition is used in the Deck class, as 52 new Card objects are created in the makeDeck method of the class. The deck accesses the attributes of Card class through composition.

### Loose Coupling Between Objects

Loose coupling between objects refers to the relative independence of the classes from each other. It allows one class to be modified without requiring changes or requiring minimal changes in the other classes.

The code fails this principle, as the Table class calls the methods of other classes. No interfaces or private variables have been used to implement this, the methods are static and therefore can be used without creating objects of the class. Any change in the methods would have to be reflected in the Table class. This makes the code very difficult to debug, as multiple files need to be edited when making changes to the code.

### Abstraction

Abstraction allows classes to share the same basic structure of attributes and methods, and different implementations of the methods. It allows the programmer to hide the unnecessary details from the user, and enhances reusability of code.

The code does not use abstraction, as there is no need for the same methods to be defined differently in multiple classes. The methods are accessed directly from the concrete classes. This makes the classes more interdependent, making the code more difficult to modify and debug.

## 

## Known Limitations and Bugs

### Check

When the first person checks and someone places a bet later, the program does not prompt the first person to match the bet.

### Raise

When someone raises after the first person has already bet, the program prompts the first person to match the bet placed later. However, the program does not account for the amount already bet by the player in this round.

### Best Card

In case the highest hand value is present on the table itself, for example the only combination in the game is a player on the table, the program sets the best card to the same card for both players and declares them as tied winners instead of comparing their holeCards.

### Big Blind Small Blind

The program does not automatically deduct bigBlind and smallBlind from the players.

### GUI

The method of input is such that players can give inputs without being prompted. This can cause major errors in the code as the values of variables get modified.

## 

## Analysis of Design Patterns

### Builder Design Pattern

The builder design pattern separates the building of an object from its representation. This allows the programmer to build complex objects step by step, and reuse the same code to implement different representations.

This design principle is not applicable to the submitted code. This is because the complex classes, such as Table and Deck, have to be created only once. Therefore, the builder design becomes largely redundant. The classes with varying representations, such as Player and Card, require very few arguments and hence do not warrant the use of this design principle.

### Strategy Design Pattern

The strategy pattern uses objects which represent various strategies and a context object whose behavior varies as per its strategy object. This allows the implementation of multiple objects of the same class, that exhibit different behaviors.

This principle is not applicable to the code as multiple objects of the same class are expected to exhibit the same behavior for the purposes of the game. So, using a strategy design pattern is not warranted.

## 