CS246 Final Project Design Document – Straights

By: Irene Xu

Introduction:

I changed my implementation idea several times over the timeline of this project. These changes mainly revolve around what methods to implement for each class which can lead very different applications of the class. Since the overall big structure of the design did not change, I will not explicitly explain how I change the methods. Due to the frequent adjustments I did not get to finish a fully working program, the program assumes all inputs are valid and a seed value is provided when the program is called.

Overview:

The game is overall, constructed with the following classes: Card, Deck, Player, Table, and Game.

1. Card:
   The Card class is a class for holding information of a single card. Key attributes include the rank and suit of the card, methods to access this information, and displaying this information in text.
2. Deck:
   The Deck class is a class for holding the full deck of 52 cards that will be used to play the game. Key attributes include the vector of cards, an integer seed that will be used to shuffle the card, a method for shuffling the cards, and methods for accessing key information and display the deck in text.
3. Player:
   The Player class is an abstract base class for holding information related to a player. There are two derived classes ComputerPlayer and HumanPlayer which each holds information specific to a computer and a human player respectively. Key attributes in the Player class include the hand of cards a player gets for each round, the vector of cards discarded each round, the vector of legal plays available for each turn, the player's cumulative score and score gained each round. The key methods in Player include discarding a card, determining the legal plays, determine if all cards are played, and displaying cards in text.
   ComputerPlayer has a special method for playing a card that is specific to a computer player. Since computer player does not need commands from user, it automatically plays the first legal play in the hand, and discards if no legal play available.
   HumanPlayer has special methods for playing and discarding a card. This is because human player needs command from user to take actions.
4. Table:

The Table class holds information on the game table, namely the piles of cards of each suit played. Key methods of this class include text display of the table, placing a card on the table, and clearing the table.

5. Game:
   The Game class facilitates the game by putting the other classes together. It holds key information about the game including the deck of cards generated for the game, the 4 players, and the game table. The overall game works in the following order: starts the game, begin a new round, prompt players to play when it is their turn, play until a player has scored over 80. The game class receives most of the commands from user except the seed given when calling the program.
   The main class here is only used to create a Game object and call a start on the game with the seed from input.

Design:

To prevent redundant codes I used inheritance relationship to implement the players. As specified already in the overview, I have an abstract base class Player and two derived classes ComputerPlayer and HumanPlayer. Computer players and human players behave very similarly, thus it makes sense to implement a parent class to hold all common attributes and methods of the two derived classes. More importantly this allows the rage quit feature to be easily implemented by simply transferring the common attributes from the human player to the new computer player.

The overall structure of the final design did not differ much from the original design. The biggest difference is that in the original design, I planned to use an observer pattern to implement the displaying classes GraphicDisplay and TextDisplay and the game table Table. Originally, GraphicDisplay and TextDisplay will be the concrete observers, and Table is the concrete subject. However, as I work through the project, I realize that I am not very confident in implementing a graphic display. And I also noticed that the displaying of the information actually depends on the actions of the player and the state of the game, so it does not really make sense to have TextDisplay as an observer of Table.
Moreover, I did not even need a TextDisplay class. We mainly need to display player's hand of cards, legal plays, discards, the deck of all 52 cards, cards on the table, and messages relating to player's actions. Note that these messages come from different classes, thus it is more natural to implement text displaying methods in each class rather than having one TextDisplay class.

I used the observer design pattern for the Player and Table classes instead. The players are the concrete subjects and the game table is the concrete observer. Whenever a player plays a card, the player object will notify the observer, or the game table, who in return will place the played card on the game table. The file info.h defines the structure info which holds the card currently

being played by the player, which is also the key information being passed from the subject to the observer.

In my original design, I planned on taking input commands and calling methods of Game in the main to run the game. However, in the final design, I did most of these in the Game class. When I started implementing the Game class, it appears to be much easier to keep everything rather than jumping between the main and Game. This is most due to the fact that the program needs to frequently receive and display messages from / to the users, separating taking in commands in the main, then pass to Game will be very hard to keep track of and appears to be more complicated to me.

Resilience to Change:

In my original design I've already considered that the best way to accommodate changes, which is to maximize cohesion and minimize coupling.
I try to design my program in a way that only highly related components will have dependency on each other, and try to break down the program as much as possible so that the implementation of one component has as little as an effect on other components.

I broke down the game into the following components: cards, player, and table.
The cards component relates to everything that deals with the cards being played, which should be mostly independent from the other two components. Similarly the player component which deals with everything related to the players of the game should be independent from the other two components. Similarly the game table component deals with everything on the game table and should be mostly independent from the other two components.

The Card and Deck classes implement the cards component, Player, ComputerPlayer, and HumanPlayer implement the player component, and Table implement the game table component.

It would be better if Game is minimally dependent on the other classes. It makes sense to have Game be minimally affected by changes in other classes since all that Game should do is putting the game together using methods and attributes of the other classes, and not caring how those methods are implemented. However, with my codes Game does depends quite heavily on the implementation of the other classes. I have mutator method for almost every private attribute in the classes and I am using these mutators over Game. This kind of makes private declaration meaningless since the whole point of declaring attributes private is to hide these details from other objects. So this increases the dependency of Game on the other classes, which makes it a bit more difficult to accommodate for changes.

Answers to Questions:

Question 1: What sort of class design or design pattern should you use to structure your game classes so that changing the user interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this framework.

Answer 1: As stated previously I used an Observer pattern to implement Player (concrete subject) and Table (concrete observer). This way when Player plays a card Table is simultaneously updated with the card being played. The structure info holds the current card being played.
In terms of structure, as stated in the overview I removed the GraphicDisplay (did not have enough time to implement) and TextDisplay classes and implemented the displaying method of each class within the class. This was the easiest and most intuitive approach that came up to me during the implementation of the program.
There are parts of the code that are slightly redundant or inefficient. I noticed latter that in many places that I needed to find a specific card from a vector of cards, perhaps I should of write a method for doing this instead of looping through the vector everytime.

Question 2: If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses. How would that affect your structure?

Answer 2: My answer to this question hasn't change much. I structured the player classes using inheritance with a base class Player to hold common attributes and derived classes ComputerPlayer and HumanPlayer to perform methods specific to a computer or human player. This also allows me to store them in the same vector. My program does not support different types of computer players. However, similar to my original answer, I think a strategy design pattern can be used such that ComputerPlayer will be the context, and there can be an abstract base strategy class with derived classes for differently playing strategy.
This in theorem should not affect the overall structure of the program by much since the how a player plays is mostly independent from other components.

Question 3: If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

Answer 3: Inheritance results that the key attributes (hand of cards, cumulative score, discards, and current card being played) that need to be transferred are all common attributes for computer and human players. So with the help of accessors and mutators I can easily transfer the information associated with the human player. And it is also much easier if there is copy/move

constructor or copy/move assignment operator. I did not have time to implement them but I latter realized that it would be very useful and more efficient if they were implemented.

Extra Credit Features:

I did not have time to implement any of the extra credit features.

Final Questions:

Question 1: What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Answer 1: I worked alone on this project, and the most important lesson I learned is that it is very important to plan carefully before starting implementation. My planning was relatively brief; I came up with an overall structure of the classes and the main methods of the classes. However, I did not plan detail enough regarding to how these methods would work, how the class attributes involve and the best arguments to use to put the program together. There were many times I realize the method I wrote for one class has arguments that are hard to handle when called outside of the class. Then I had to go back and revise everything to make it work. Most of the solution came to me naturally, they seem to be easiest approach at first, but as I progress through the project I realize that if I were to design the solution more carefully it would be much easier for me to debug and eliminate errors, however when I notice that I do not have any time to make such huge adjustments.
I learned that when writing large programs it is very important that I design the classes carefully, this can avoid many errors and prevent you from going too far in the wrong direction.

Question 2: What would you have done differently if you had the chance to start over?

Answer 2: If I had the chance to start over I will design my classes more carefully. Right now my program causes segmental fault and the program is too large which makes it extremely difficult to find out which part caused the problem. If I took more time to design my classes and methods I probably will be able to eliminate many of these logical errors. I also wish that I work on it more every day and save myself enough time for debugging. I first worked with a C++ shell and did not notice that it wasn't C++14, after I transferred the solution to my repository I got a bunch of compilation errors which wasted a lot of the time. And I changed my class structure many times which resulted that I didn't have time finish a complete working solution. If I had the chance to start over I would really try to work on it more every day to make sure I have enough time to fix errors or make adjustments.