Overview:

I used the MVC and observer pattern to implement straights. The player has access to a front-end display of cards and the commands that affect the game state (included in the View Class). The commands included in View delegate the changes to game state to the Round Class. The game state is updated, the Observers are notified, and the next player sees an updated front-end display.

The Game class is composed of 4 Players and 52 Cards

The Player class is responsible for player data such as the player's hand, their discards, their points, and whether they are a CPU.

The Card class is responsible for the representation of a card, the comparison of cards, and the increment/decrement of cards (e.g. 7S -> 8S or 7S->6S).

The Round is a child of the Game class. It is responsible for holding the state of the current round's card piles and the changing the game logic for each round (e.g. player actions like play, discard, etc.)

The Observer Class is an abstract class used to facilitate the Observer Pattern.

The View Class is responsible for the game interface (View->update()) and their input controls (View->controller()).

Design:

Dealing with Cards using Card Classes and Encapsulation

In order for the game to determine legal moves, we must compare the cards in the player's hand with the cards currently on the table. In straights, we can play a card onto a pile (assuming the same suite) if it is either exactly one greater than the biggest card in the "pile" or one lesser than the smallest card in the "pile". For numbered cards, this process is simple arithmetic but for face cards, this poses a problem. For example, I needed a way to determine that the next card after the J was the Q.

In addition, we needed to be able to determine the value of cards when tallying the points accumulated for a player's discards (at the end of each round). Again, the value

associated with a number card is it's number but the logic for determining the value associated with a face card needs to be programmed and is not as trivial.

To solve both of these problems, I created the Card class that holds the information for a card's representation (i.e. AD), its value (i.e. 1), its suit (i.e. D) along with the relevant methods such as a comparator function between the object and another card, and overloaded increment and decrement operators (i.e. ++QS gives you KS). This way, the code for comparing cards does not need to be repeated in different modules and the complexity is hidden from the client who is using it.

Encapsulating all the card logic in information in a dedicated Card class solves the problem of getting "one larger" or "one smaller" of a card for determining legal moves and solves the problem of tallying discard values.

Memory Management and Smart Pointers

In my program, there are many objects. Each of these objects require heap memory to persist but managing the memory of many objects can be tedious.

A solution to this problem was the use of smart pointers. These leverage the concept of RAII to acquire resources on initialization and release them in the destructor (smart pointers return resources when they are no longer in use (e.g. out of scope)).

I used shared pointers for resources that needed multiple references (e.g. Players, Cards, etc.) and unique pointers for the Game, the View, and the Round (only need one reference to them).

Code-reuse with Polymorphism

A game consists of multiple rounds where each round acts as a miniature game until enough points are accumulated to end the game. In this way, the game and round share a lot of similar data such as the players, the deck, and the functions that act on them. Instead of completely rewriting this code for the Round class, I simply made it inherit these fields from the Game Class. Here, using polymorphism improves code re-use. Another use of polymorphism is with implementing the Observer pattern. This is hugely beneficial for code resilience and I will talk about this in the next part.

Improving Cohesion with the Single Responsibility Principle

To increase cohesion, I tried to stick to the single-responsibility principle; I tried to separate classes by functionality (e.g. Card class is responsible for card data and Player class is responsible for player data) so that only related information and functions are kept together. This makes it easier to add code and makes it easier to find bugs (facilitate) since we know exactly where to find the code that is responsible for something.

Using <list> and <algorithm> library

I decided to use the std::list data structure to represent a player's hand. I did this because I often needed to remove an element from the middle of a player's hand. In an array, the time complexity (worst case used) of removing an element is O(n) since we must shift all the elements that are to the right of that element one space left. With a linked list, deleting an element is O(1) time. To traverse through the list I used the iterator and std::find which made my code shorter and more readable.

Resilience to Change:

The general idea for resilience to change was to decrease coupling and increase cohesion as much as possible. If someone wanted to make a change, I wanted it to be so that they would not have to make a large number of updates to different parts of the code. In the examples below, significant modifications to specification only require changes to one or two (at most) classes.

Change to Game Display:

The display that is shown to the player is in View->update() . If we want to change what is shown to the player during each turn, we only need to change this function. In addition, if we wanted to add a visual display (i.e. not text-based), we could implement that as a child of the observer class and simply make an object of it in main.cc (since notifyObservers is called at each turn to update the display). The Observer pattern makes this very easy.

Adding New Cards:

The Card class gives support to adding cards with special effects. For example, say we wanted a card that clears a pile that the player selects. We would simply have to instantiate such a card using the "special" constructor when the deck is being created (Game->fillDeck()) in the Game class and then update the Round->setState() function as required. This is possible because of the cohesion of modules. The Card class

contains all the necessary information about the cards and the Round class contains all the round data (such as the player turn, card piles, etc.) and Round->setState() has access to them.

Change Command Options:

Say we wanted to add to the options available to players (e.g. play, discard, etc.) . To give a player new commands (or remove commands), changes would only need to be made to the main.cc (where the input is processed) and in the View class (where the command logic is implemented).

Changes to Bot Strategy:

The current strategy for the bot is to play the first card in legal moves or discard the first card in hand. What if we wanted a greedy strategy that maximizes points by playing the lowest ranked cards possible (in order to discard cards of the highest rank)? We would simply have to write a new function in the Round class and make the required changes to Round->botMove().

Straights Q/A:

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

I used the MVC and observer pattern to separate the game state from the view/controller. This way, changes made to the visuals from text-based to graphical in the View class does not affect the game data stored in the Round class. As noted above in resilience to change, changes made to game rules are highly localized and only need to be made to the class responsible for that part of the game.

Question: Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your class structures?

In this case, the problem could be solved by using the Strategy pattern to interchange between algorithms for the computer player. However, since the Round class already has a dedicated function for the move of the computer player (i.e. Round->botMove()) , it is totally possible to write helper functions in the View.cc and call them in the

Round->botMove() method when certain conditions are met (the method can check). This would mean no changes to class structures.

Question: How would your design change, if at all, if the two Jokers in a deck were added to the game as wildcards i.e. the player in possession of a Joker could choose it to take the place of any card in the game except the 7S?

My design would not change and fully enables the addition of two Jokers. In the Game Class there is a method called fillDeck (fills the deck vector with shared pointers to cards) which is called in the constructor when a new game object is made. To add this feature, I would make a change to the fillDeck function to add the two jokers using the "special" constructor (for cards with special effects) from the Card class. Once the card is added to the deck, I would make a change to the Round->setState() function to add the necessary functionality when the function is called with the Joker card.

The difference between the current answer and the answer in DD1 is that I did not have a dedicated Card class in my UML for DD1. This meant that I had to hard code the cards in the deck as strings instead of using a loop and instantiating using the Card constructor. This also meant that I had to manually make changes to the legalMoves method for each new special card. The "special" field in the Card class generalizes this and makes adding special cards much easier.

Final Questions

Question: 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?

I worked alone since Straights was a solo project. I learned about the importance of good version control. There were many times when I wanted to undo changes and it was much more efficient to go to a previous version than to reverse the changes I made. This is especially true in a large program where changes are made to classes separated in different files.

In addition, I realized the value of writing a UML prior to coding for a large program. In a large program, having a general overview of the interactions between classes, relevant functions, and their ownership relationships (e.g. composition) was an incredibly helpful reference that made implementation go much faster.

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

I would have spent more time planning prior to programming in order to save a lot of time spent editing code. In particular, I would have spent my time gaining a better understanding of the MVC pattern and I would have developed a Card class before implementing my UML in code. Initially, I included the functions that would change gameplay logic directly in the View class instead of delegating it to the Round class (acts as the Model in MVC). I realized that this increased coupling and decreased overall cohesion in the view class. As a result, I had to spend a good amount of time editing the class structures. Additionally, I did not include a dedicated Card class in my first UML and implemented straights without it. This led to repeated and confusing code when processing the logic for comparing and incrementing cards. Since I didn't like the code that I wrote, I spent a good deal of time editing it to include a Card class. In the end this was worth it, but I could have saved a lot of time if I thought of it at first.