Introduction

Overview (overall structure of project)

The Straights game that I implemented was an object-oriented game.

Main

The main function takes the command-line argument which is the seed if any. The straight class object is created with the seed as a parameter. The controller is created, which takes the memory address of the straight object as a parameter, which will result in an aggregate relationship between the Straight class and Controller class. Finally, the view class takes the straight class object and the controller class object, which is an aggregate relationship between view and controller and view and straight. The game recursively proceeds. The view class calls the controller based on the current state of the game, then the controller calls the appropriate method in the Straight(model) class. Finally the straight class performs the method, and notifies the view class. This recursive style continues until the game ends or the human quits. And then all the call stacks are returned.

Straight Class

The main game logic lives in a class named Straight. This is the class that keeps the game together and was considered the model for the model view controller design pattern. All the game specifications live in this class. Thus, majority of the public methods are inside of this class. Since in the Straights game there are 4 players, its followed that the Straight class has a composite relation with the Player class. Once the game starts 4 players are created by the Straight class. Furthermore, since the Straights game is a card game, and the game cannot exist without a deck of cards, I created a cards class that has a composite relationship with the Straight class as well.

Cards

The cards class is where all the card objects live. The cards class create 52 unique cards (regular deck), and stores all the cards, thus, it has a composite relationship with the card class (has rank and suit). Since after each round the cards must be shuffled there is a shuffle function. And the cards class holds all the cards that have been played by the players, thus the straight game can add cards that the players played.

Player

The player class is an abstract base class. Since every player must play a card or discard, these methods exist in the player class. Also, it holds 0-13 cards that they have in their hand, thus making this an aggregate relationship. Since there are 2 types of players, a computer and human, these types of players inherit from the player class.

Computer/Human

Majority of the logic of humans and computers are similar thus, they inherit from the player class. However, both types of players have a different method of playing a card, thus both these classes override the play/discard function in the Player class. In the human class the first card that is playable is played whereas the Human class chooses what legal card to play.

Design/Changes from DD1

Design Problem 1 Displaying:

Since I wanted to make its easy to go from terminal text interface to GUI interface, I needed to design my code so that all the output and user input must exist in one class. As a result, I decided to use the Model View Controller design pattern as I mentioned in DD1. By implementing this design pattern all my output and input was handled by the view class. Since the view class had a pointer to the straight object, it was able to see the changes made to the state of the game, and produce the output based on the changes to the game. Now if I want to go from my Text interface to a GUI interface, I will only have to change the view class instead of having to change the classes that contain all the logic. Separating the output, and controller and logic was a very smart technique to use for this project, because it kept the code organized. If there was an error in output, I would only need to look at the view class. However, if the wrong task is being performed then I would need to look at the controller. Finally, if there was a logic error (score not adding up, or card not being removed from hand) then I would need to look at the classes related to the Straight game. Overall using the MVC design pattern increased the cohesion as proven by the examples above. In order to implement the Model View Controller design it required me to use the Observer design pattern. The Straight class also considered the model inherited from the subject class, which notified its observers (view) when a modification was made. The view class inherited from the observer class. Thus, it had a virtual override method called update which was called by the subject when there were any modifications made to the game.

Making Cards Class (Design Change)

During due date one I decided to have all the functionality of the deck within the Straights class. But once I started coding, I realized that a deck of cards is a functionality that does not have much to do with the logic of the game but it is something that the Straight game requires and any game with a deck of cards would require. Thus, to increase the cohesion of Straights classes I decided to remove everything related to the deck from the Straights class to a separate class called Cards. The Cards class was responsible for creating a deck of cards and being able to shuffle the cards when needed and distribute the cards to the players by giving an iterator of the deck at a certain position, to the players. Now the relationship between the Cards and Straight Class was a composite relationship. So whenever the Straight game needed to do something with the cards it would call on its instance of the Cards class and call the appropriate function.

Making Each Card an Object (Design Change)

Furthermore, initially at due date one I designed the program so that each card would be represented as a string. However, I realized that this was not an effective strategy because it did not accurately describe the relationship between a card and a player. By storing it as a string, the Human or Player can easily change the vector of strings or the cards they have. However, by creating 52 unique cards, and giving a pointer of the cards to the players, resembles how the game truly is. In a card there are 2 private members consisting of rank and suit. And two public accessor methods that tell the rank and suit of the card. This way, even though a player points to a card they will not be able to change the value of the card. Furthermore, the card class is cohesive because it contains all the information of a card and gives the ability to see the card without being able to modify it.

Small Modification to Player Method

Lastly, during DD1 I decided to make a method named make_move instead of a discard and a play method, because I knew the computer makes a move automatically. But I realized that the make_move method does not describe a player accurately. A player must play or discard, making a move is more related to the computer class. Now, the make _move method logic is instead inside the computer class instead of the player class.

<u>Resilience to Change</u>

There are a bunch of techniques that I implemented to make my program resilient to change.

1. First I used the MVC design pattern in order to make swapping views more convenient. As I mentioned before using MVC design pattern allows all the user interface related task to be grouped in one class. As a result, when changing from text-interface to GUI I would simply have to swap my view class with a view class that allows GUI related features. To continue, even if a different card game was to be played, many of the classes can be reused. For example, the Cards, Card, Players, Human and Computer class can all be reused even if a different card game was to be played. By making cohesive classes, many similar aspects like the ones I mentioned can be used for a different card game.

2. Furthermore, my program is designed so changing the number of players in game would not have an effect on the logic of the game. The only aspect that would change is the process of the dividing the cards evenly when the game begins. Since the logic of the game does not depend on the number of players, this would be an easy change. This shows that my players class is loosely coupled.

3. Adding more our less cards can happen with ease because of the way my program is designed. The only modification that would need to take place, is during the construction of each cards, a new card set would need to be specified. And then, the game will continue the same way its continued with 52 cards. Once everyone has no cards the round ends and all the cards would be distributed again. This shows that the cards class is loosely coupled.

4. Lastly, if there is any characteristic that needs to be added to Humans and Computers only the player class would need to be changed instead of changing every type of player. The benefit comes from having inheritance. Since both a human and computer inherit from player, then whenever, a new feature to a player is added (ie now players call see other players cards) then

only the player needs to be modified and then both humans and computers will get that feature.

Answers to Questions

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

To minimize the affect of changing the interface, the model view controller design pattern should be implemented. This design pattern separates the game logic and the interface, therefore, if any changes need to be made to the interface only the view class will need to change. For example, if I decide to first do the output on the stdout, then I would make a view class that is able to do so with the following model and controller class. But if later I decide to add a GUI, then I would only need to swap the view class with a view class that supports GUI. And if the game rules change then only the model class will need to change since, no changes will need to be made for the view and controller class. Using the MVC design pattern fits with the framework because the Straight game class will be model class in MVC. The view class will display the cards of the human player, and display they cards the computer plays and the stats that need to be displayed at the end of the round. And the view will also take input from the user and transfer it to the controller. The controller will decide what function needs to be called in the model class. After completing the program I still agree with this, and I implemented it in the game.

5.5a) 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 i.e. dynamically during the play of the game. How would that affect your structure?

To allow both computer players and human players, I would make an abstract base class of players. The functionality that pertain to both the human and the computer would be included in the base class of players. For example, the players would have all discard cards and remaining cards. Then I would have two subclasses, human and computer which would inherit from the players class. This would be an inheritance relationship. If you want the computer player to apply different strategies, then the Strategy Design pattern should be implemented. I would create an abstract class named Strategies. This class will provide an interface that allows a user to implement a specific strategy. Then I would create various Concrete Strategy Classes that inherit from the Strategy class and provide their specific algorithm. Finally, the computer subclass will have an aggregate relation with the Strategy class so it can access the current algorithm, but also dynamically change the algorithm it is using. After completing the program, I still agree with this, and I implemented the parent-child relationship between the players and the computer or human. I did not implement the strategy design pattern because I didn't have the time to implement multiple strategies.

5.5b) 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?

The Straight class will have a composite relationship will the players. So, if a human player wants to be replaced with a computer, then the Straight class will call the accessor functions for the human and retrieve all important data. For example, the current cards, the discarded cards, and the scores. Then I will have a constructor for computer that is able to take all of that information as parameters and initialize the computer based on the given parameters. Finally, the human player will be destructed, by erasing the player in the vector and since it is a unique_ptr the human player is destroyed immediately when it is deleted from the vector. This is the same as DD1 except, I decide to use unique_ptr to store the players, in order to make memory management easier.

<u>Extra Credits Features</u>

I did not implement any extra feature to my program. The only extra credit enhancement I included is adhering to RAII and using smart pointers instead of using the new and delete statement.

<u>Final Questions</u>

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

There are many things that I took away from completing this project.

The most important thing I took away from this project is the importance of planning. When I was doing the plan during DD1 I was anxious to get started coding, thus, I didn't have 100% clear idea about how my program was going to work. For example, when I started, I knew I was going to implement the MVC design to make the transition from Text to GUI easy, however, I did not have a clear idea about how to implement MVC. I should have learned from the example given in the cs246 folder, and then made the UML which correctly modelled MVC. Furthermore, I did not consider all the accessor functions I would need in order to display all the required content. Thus, when it came time to implementing the view, I constantly had to add methods to the model, in order to display all the required information. Overall this project made me appreciate the process of planning A LOT more.

Another lesson from writing a large program is testing the program as you go a long. I had the false perception that if I am careful I could complete the entire model, and then do the test harness (ie the view) and check my program at the end when everything is complete. However, debugging became very painful at the end because there were so many parts working together, thus, it was hard to pinpoint what caused the problem. This resulted in many more hours for testing and debugging. Also, every once in a while I would encounter an unexpected bug. Thus, I spent many hours carefully going through every line of my code to make sure that the program ran correctly.

Finally, another lesson I learned is the significance of documentation. Since the program was long, after taking a break from working on a certain class in the program I forgot why I was doing the things I was doing in the classes I worked on previously. As a result, mid way into writing the program I revised all the documentation on my h files and made them a lot more thorough. This helped me significantly, when I was debugging because when I wanted to know exactly what at method was doing I would just read the h file. I should have also had inline comments to understand specific lines.

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

All the lesson I mentioned above, including, being thorough with planning, testing as I went a long and adding more meaningful documentation, I would implement if I had the chance to start over. I would have implemented the strategy design pattern to allow the computer to use multiple strategies instead of being locked up with one strategy. I would also, come up with different strategies the computer should use at different instances. Lastly, if I could start over I would provide the definition of a single card within the cards class. This way it prevents any of the other classes to create cards. And in reality if this game was to be played no one should be able to make cards, it should be decided by the given deck.

Conclusion

Overall, I believe I was successful in developing the Straights game and applying object oriented programming, and smart pointers.