DESIGN (STRAIGHTS)

Introduction

For the final project of CS246, I decided to implement the card game Straights. I started by reading and understanding the requirements for them and then planning out the possible classes that I will have and also their interactions in the UML. As I implemented the program, there are things that are still implemented according to the original plan. However, there are also things that I realized along the way, that needs to be adjusted and changed.

Overview

For the design, I have a GameController class which is responsible for taking the input from the user and then translating it into appropriate function calls to the Table and Player class. As the user enters a command, GameController manipulates the data in Table and Player class appropriately.

The Player class is an abstract base class. It has two derived classes, Human and Computer. Its job is to keep track of the cards of the players, whether they are discarded or not, and other information about them. For the Table class, it keeps the data of the cards that have been played (not discarded). Lastly, I also have a Card class, which keeps the information of the rank and suit of a card.

- **GameController**

  The GameController class is responsible to take the user's inputs/commands and manipulates the data of Table and Player class. It acts as the "connector" between the Table and Player class and controls the game flow. It decides when a player will play or discard or quit, etc. according to the inputs and decides whether the game still continues or not when a round is over.

  Besides information like the seed for shuffling and number of players, it also keeps track of the cards that are allowed to be played during a current turn in its member, "playable". Every time a player plays a card (play not discard), the GameController will update it.

- **Player**

  The Player class is an abstract base class which has two derived classes, Human and Computer. It defines a single player in the game—a human player or a computer player respectively. It holds the information of the player's ID (1, 2, 3 or 4), the total score that the player has achieved so far and the cards in their hand, legal plays and discard. All of the members of Human and Computer class are the same and all of them are inherited from the Player class. By doing this, it will incorporate the *ragequit* command easily, which will be discussed later.

  The Player class provides three pure virtual methods called "isHuman", "play" and "pDiscard" (stands for play discard) for its subclasses (Human and Computer) to

override. This is done so since the way a human player play and discard are different compared to a computer player's way (i.e. they are implemented differently)

During the game, the GameController class needs to be able to communicate to the Player class to tell them the cards that they are allowed to play. This is accomplished by having a method in the Player class called "makeLegalPlays", where the GameController will "give" its playable to the player and the player will then process it to know what their legal plays are.

- **Table**

  The Table class is responsible for representing a physical table in the game. It contains 4 members: club, diamond, heart and spade, which are all a deque of string. These represent the piles of the appropriate suit according to its name. Deque of string is chosen over deque of card to accommodate the printing format that only involves the ranks of the cards. These four members are responsible to show the cards that have been played by the players to the table (i.e. the cards that are played, but not discarded).

  Table also provides method to "clear its table" (i.e. clearing all of its vector), which will be called by the game controller when a new round has started. Also, the GameController class needs to somehow communicate with the Table class whenever a player plays a card to update it. This is achieved by having a method called "addToTable" in the Table class. Lastly, since whenever it is a human player's turn we need to print the cards that are already on the table, Table class also consist of a method called "printTable", which prints all of the cards on it.

- **Card**

  The Card class defines a single card in the game. It contains information about the rank and suit of the card. It has a function which counts its value according to its rank (e.g. 1 for A, 2 for 2, 10 for T, and 13 for K). It also overloads the operator== for us to able to use the find function of the STL Algorithm. The find function is helpful to let us know if a card inputted by a player is inside the hand or legal plays of the player or not. To help the program to print the cards, the operator<< has been overloaded as well.

Design

One design challenge is to find the best design such that we can solve the ragequit command (i.e. changing a human player to a computer player) easily. Since, this requires transferring all the data from a human to a player computer, it's best for them to have the same attributes. Hence, to not have all of these attributes written repeatedly, an abstract base Player class that derives two classes—Human and Computer class—is made. It includes all of these attributes and also all methods that are common between the Human and Computer class. The Player class then provides virtual methods for these two subclasses to override, like the "play" and "pdiscard" since human and computer player plays and discard their cards differently.

By having this design, switching from a human player to a computer would be easier. This will allow easy transfer without worrying about any attributes being cut off and that the call to "play" and "pdiscard" would directly change from the one implemented in a Human class to the one implemented in a Computer class (polymorphism).

Another design challenge is to have everything separated, but not to add complexity at the same time. This leads to a **change in my DD1 UML and design.**

My previous design involves something that is similar to MVC. It used to involve a GameDisplay class which inherits a TextDisplay class as the view, GameController class as the controller and Player class which inherits Human and Computer class as the model. It also had a Card class as well. However, it did not have a Table class since the piles of cards was placed in the GameDisplay class. The GameDisplay class was also meant to manage all the printings.

This change has been made because when implementing the code, I realized that separating the codes into model, view and controller (i.e. using MVC design pattern) creates more complexity. Since the implementation that I have only involve a text display, I realized that having a function that prints data in the Table, Player and GameController class and call them when needed would make the code more is simple and less complex then having everything located in the GameDisplay class.

Resilience to Change (describe how your design supports the possibility of various changes to the programs specification) (Perhaps a change in rules, or input syntax, or a whole new feature—who knows? Plan all of your project features to accommodate change, with minimal modification to your original program, and minimal recompilation.)

Using this design where the GameController is the one that controls everything, including data of Table and Player, and that each of the Table, Player and Card class don't need to know how the other modules work, we can see that this design shows low coupling. Hence, this allows us to change the implementation easily when various changes to the program specification are made,

Consider a change in the rules of the game, like for example, the starting player could place not only 7S, but could also place other cards that are in their hand. Then, here, we only need to adjust the GameController.

Or consider a change where there could be any number of players (at least 2) in the game which makes it possible for each player to have different number of cards. Then we would again only adjust the GameController such that it can skip a player when the card in that player's hand has finished while the others haven't in that certain round.

A change in the input syntax would also make us to only need to change the GameController since it is the only module with the task of taking the input from the user.

If we want to change the way a computer player plays or discard, we would only change the play or pdiscard method of the computer's. Furthermore, if we want to have a computer class with different levels, for example, ComputerEasy, ComputerMedium and ComputerHard, then we could just easily add more subclasses to the Player class and overload the play and pdiscard functions differently between them.

These examples show that a change in a specification would only change the modules that has that following roles, showing that the modules have low coupling.

The Table, Player and Card class has high cohesion since they are all working toward a "common goal". Table manages everything that is on the table, Player keeps track of everything about a player and Card manages everything that is only about a card.

However, a downside to this design could be seen when a change in the output or display needs to be made. Since most of the classes have a print method (GameController, Table and Player), when this change needs to be made, we need to implement a change in all these three modules. If it's a small change then this would be okay since the game isn't that complicated. However, if there needs to be a big change in the display, like changing from text-based to graphics, then big changes should made which is bad. Hence, separating the display alone in class would actually be a better idea since it will adapt to changes better and therefore minimized the coupling further.

Another downside of this design is that the GameController manages a lot of things from inputs and game flow to shuffling and printing the starting deck. This show low cohesion in the GameController module, making maintenance more difficult. Hence, separating it into more classes, like adding a Deck class that manages the shuffling could be a better idea as it would increase the cohesion between modules.

Answers to Questions (the ones in your project specification)

Question: 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.

I'm planning on using something similar to MVC design pattern where the model, view and controller are separated, but without the observer design pattern, since in this case, changing the state of the cards on the table don't really trigger much change to the other objects. Hence, having an observer design pattern implemented inside will just add more complexity.

Since the model, view and controller has been separated, changing the interface or changing the game rules would have little impact on the code. For example, if we want to change the display to GUI, we can just add another class which inherits from the GameDisplay class. And if we want to change the game rules, we could just adjust the controller.

Question: 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?

We can have a class for Human (player) and Computer (player) inherit from an abstract class of Players. Since the computer player's strategies might change as the game progresses, we can use the Strategy design pattern for this. We can have abstract base classes: PlayStrategy and DiscardStrategy which derives classes of playing strategies and discard strategies respectively. Then the Computer class will have two additional attributes, one for the playing strategy and one for the discard strategy and two additional methods to change these two strategies. Meanwhile, the Human class will only have two methods, play and discard and no extra attribute.

Question: 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?

[Different to previous answer]

I would have both class for Human (player) and class for Computer (player) inherit from the same abstract base class, Player. They share the same attributes that's in the Player class and that the Human class will have no extra attributes. The Computer class will then contain a copy constructor which will construct a Computer player with the data of the player it copies from. Thus, this will allow easy transfer.

Extra Credit Features

Handle all memory management via STL containers and smart pointers.

Final Questions (the last two questions in project_guidelines.pdf)

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?**

   For the last two weeks, having to write a large program individually have taught me a lot. Since, this is the first time in the course where we should write a large program from scratch, I learnt is that planning and designing is a very crucial step before starting. I also learnt that it's okay to not stick completely to our first design and change some of them as we try to implement it.

   Secondly, I learnt that separating and assigning a specific task to each classes is very helpful (single responsibility principle). Like in the design, the Table, Player and Card class. By doing this, it makes unit testing much more easier to be done and makes debugging so much easier, compared to the GameController class which has more tasks in it.

   As a person who always test the code when everything has been done, this projects let me try to do unit testing and I found it to be beneficial since it also make debugging so much easier.

   Lastly, through this project, I also learnt that it's not enough to just write a code that works properly. We should always consider other things too, like how easily will our code adapt to all the possible changes. This is something that I've never really done and thought of before. Hence, I will always keep this in mind when writing codes in the future.

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

   If I would have the chance to start over, I would have the MVC design pattern in my design. Though this may create more complexity, this design will result to having a higher cohesion and lower coupling since the Model, View and Controller are separated into different classes.

   I would also try to increase the cohesion of GameController since it has quite a lot of tasks. I would add another class called "deck" that will consist of a vector of cards of size 52. I will move the shuffle method there so that the GameController wouldn't need to deal with the shuffling.

   Implementing these would increase cohesion and lowers coupling, making the code more resilient to change and easier to debug.

Conclusion

Overall, given that I have never written a large program from scratch before, I'm glad that I was able to accomplish this project. This final project has given me the opportunity to implement the things that I've learnt in the CS246 course to a real large program.