Overview

This document discusses the design and functions of the Straights card game. I implemented the straights game using MVC (**m**odel-**v**iew-**c**ontroller) with text-based inputs and outputs. View is implemented with the observer pattern.

Model is the Round class, *view* is the View class as well as other parts of the observer pattern (the Observer class and the Subject class), and *controller* is the Controller class.

1) Model (Round): manages and modifies the game data

It holds data about the cards that have been played, legal plays and game players. The data about players are stored in the Player object and cards are stored in the Card object.

2) View: manages the interface of the game data

The View class is a text-based observer that outputs any changes or necessary commands made during the game. The Observer class is an abstract class with only pure virtual functions. The View class is a subclass of the Observer class. Whenever an update is needed, an Observer is notified through Subject, and View has methods to print the appropriate updates.

3) Controller: manages the user inputs

The Controller class is responsible for getting the user input and starting the operations in the model. It gets the user inputs about the player types, Human or Computer, and also handles inputs for Human players.

Design

This outlines some of the important methods and attributes in each class.

1) Player

The Player class has two subclasses: Human and Computer. The attributes `hand` and `discards` are a vector of Cards, `score` is a vector of length 2 where the first element is the old score, and the second element is the current score in the round. The `id` (0-3) specifies the order that the Player plays.

a) `removePlayed(char, int)`: The `suit` and `rank` of the Card are passed in as the first and second argument. It removes the Card with the `suit` and `rank` from `hand` of the player.

b) `checkValid(string, vector<Card>)`: The Card in a string format (e.g. 8S, TD) is passed in as the first argument. It checks if the Card is present in the vector of Cards that is passed in as the second argument. It is used to check if the players are playing cards that are a part of the legal plays.

c) `resetPlayer()`: It clears `discards` and adds the old score (`score[0]`) and current score (`score[1]`) and make this new score as the first element of the `score` vector.

d) `intRank(char)` and `stringRank(i)`: They return the integer representation and string representation of a `rank`. (e.g. `stringRank(10) = "T"`, `intRank('T') = 10`)

Human

a) `Play(vector<Card>)`: It takes another user input, which is the Card that the Player is playing. It first calls `checkValid` with the new Card and the legal plays, which is passed in as the argument. If the Card is valid, it calls `removePlayed` on the card played. It returns the new Card that has been played. The returned card is used to update the table.

b) `Discard(string)`: The Card in a string format (e.g. 8S, TD) is passed in as the first argument. It calls `removePlayed` and adds the Card to the `discards` vector of the player.

Computer

a) `Play(vector<Card>)`: `legalCards` is passed in as the argument. It takes the first Card of `legalCards` and calls `removePlayed` on the card played. It returns the new Card that has been played. The returned card is used to update the table.

b) `Discard(string)`: It takes the first Card of `hand`. It calls `removePlayed` and adds the Card to the `discards` vector of the player.

2) Card

The Card class has two attributes: `rank` and `suit`. `getRank()` and `getSuit()` each returns the `rank` and `suit` of the given Card.

3) Deck

The Deck class has a deck attribute, which stores the deck for each round. `createDeck()` makes a new ordered `deck` of 52 Cards.

4) Round

The Round class has a `playerList`, a vector of Cards of each suit, `legalCards`, `turn` and `current`. `playerList` is a vector of Players and `legalCards` is a vector of Cards that can be played in each turn. The vector of Cards of each suit is a vector of length 2, where the first element is the Card with the lowest rank of the suit that has been played and the second element is the Card with the highest rank of the suit that has been played. `turn` is an integer that represents the index of the first player in the list (i.e. the player who has 7 of Spades) and `current` is an integer that represents the index of the current player.

a) `updateLegals(vector<Cards>)`: It adds adjacent cards of the suit to `legalCards`. The vector of Cards of each suit is passed in as the argument.

b) `legalPlays()`: It calls `updateLegals(vector<Cards>)` with each suit and returns the updated `legalCards`.

c) `updateSuit(vector<Cards>, Card)`: The first argument is the vector of Cards of a suit and the second argument is the Card that is being added to the suit. It replaces either the first or the second element of the vector depending on the rank of the new Card.

d) `updateTable(Card)`: The new Card that has been played is passed in as the argument. It calls `updateSuit(vector<Cards>, Card)` depending on the suit of the Card.

e) `getWinner()`: It adds the old score and the new score of each Player and saves the `id` of the Players with the highest score. It returns a vector of integers, which are the `id`s of the winners.

f) `resetTable()`: It clears the table i.e. it clears the vector of Cards of each suit for the next round.

g) `startPlayer(int)`: The int argument is for the index of the current player in the `playerList`. From `legalPlays()`, it returns `possible_cards`, a vector of Cards that can be played with the cards that are present on the table. Then, it calls `getLegalPlays(possible_cards)` for the current player, which then compares the `hand` of the current player and `possible_cards` and returns the legal plays for the current player. Then, it notifies the observer with `notifyTable()` for the output of the cards on the table, hand, legal plays, and other appropriate commands.

e) `endRound(default_random_engine, Deck)`: It calls `getWinner()`and gets `winners`, a vector of winners' `id`s. Then, it notifies the observer with `notifyScore(winners)`, which prints all players `discards` and scores, as well as the winners. Then, it calls `resetTable()`. The seed that is optionally given in the command line argument is passed in as the first argument. The deck, that is passed in as the second argument, is reshuffled using this seed. While going through the `playerList`, it sets new hand with the newly shuffled deck and sets the new `turn` depending on who has 7 of Spades. It also calls `resetPlayer()` on each player.

h) `rageQuit(int)`: The index of the player that is ragequitting in the `playerList` is passed in as the second argument. It creates a new Computer player while keeping the same `discards`, `hand` and `score`.

5) Controller

The Controller class has two attributes: `Round` and `quit`. `quit` is a boolean that determines if the Player has inputted "quit". If `quit` is true, the program breaks from all the loops and quits the game immediately.

a) `startRound(default_random_engine, Deck)`: The seed that is optionally given in the command line argument is passed in as the first argument. The deck, that is passed in as the second argument, is shuffled using this seed. In a loop that iterates 4 times, it then asks for the player types (either 'h' or 'c'). Depending on the player type, it creates a new Human or Computer Player object and assigns 13 cards to its `hand`. Each player has an empty vector as its `discards` and a vector of 0s for its `score`. While assigning 13 cards to its `hand`, if there is 7 of Spades among the cards, the `turn` in the Round class is set. After each Player is created, it is added to the list of players. After all 4 players are created, the `playerList` in the Round class is set.

b) `getInput(int, Deck)`: The int argument is for the index of the current player in the `playerList`. If the current player is a Human player, the program asks for user inputs. If the command is "play", it

accepts another input, the new card, and then calls the `play` function for the current player and calls `updateTable` from the Round class. If the command is “discard”, it accepts another input, the new card, and then calls the `discard` function for the current player. If the command is “deck”, it calls `printDeck()`of the deck. If the command is “quit”, it sets the variable `quit` to true. If the command is “ragequit”, it calls the `rageQuit` function in the Round class.

c) `checkDone()`: It checks if the game is over by checking if any player’s score is greater than or equal to 80.

6) View

The View class is a subclass of the Observer class. The View class has `round` as the attribute.

a) `updateTable()`: It calls `printTable()` which prints the Cards on the table of each suit. It then prints `hand` of the current player and the `legalCards`.

b) `updateScores(vector<int>)`: It prints each Player’s score in the following format: old score (`score[0]`) + current score (`score[1]`) = new score. The vector of winners’ `id`s is passed in as the first argument. From this vector, the winner(s) of the round is printed.

Differences from the Original Design

The overall structure of the code is similar to the original design that I have created.

Some differences are listed here.

- The Deck and Card class have a composition relationship – Deck is composed of 52 Cards. Originally, the Deck and Card class had an aggregation relationship
- Originally, the Card class had an aggregation relationship with the Player class. Now, the Card class does not have any relationship with the Player class.
- The Deck and Round class have an aggregation relationship. Originally, the Deck and Round class had an association relationship
- Originally, the View class and the Controller class had an aggregation relationship. Now the View class does not have any relationship with the Controller class.
- More methods and attributes were added
- `discard` function in the Player class became pure virtual. This is because I needed different `discard` functions for each subclass.
- `legalPlays` are stored in the Round class rather than the Player class. I set the `legalPlays` in Round depending on the cards on the table. Then, I simply use `legalPlays` to compare with the user `hand` and get the legal moves. In my original design, I wanted to verify if each cards in `hand` of the Player can be played, and then store the result in Player.

Resilience to Change

The program is constructed in a way to minimize coupling and maximize cohesion. Each class focuses on a different component of the game with low dependency with other modules/classes. Each class serves a single purpose and everything in the class is extremely closely related. The game data is stored separately in the Player, Card, Deck, and Round class and they are responsible for different roles.

Since the program has low coupling and high cohesion, changes to the game can be easily implemented without affecting other components. For example, I can add multiple subclasses that inherit from the Player class without causing a change in the Card, Deck, or the Round class. Similarly, I can add different shuffling methodologies of the Deck class without affecting other classes.

Furthermore, I have implemented a design pattern that makes my program resilient to changes. If I wanted to add a graphical display for the game, it could be easily done by adding another subclass of the Observer class. This subclass can have methods that override the pure virtual functions in the Observer class to have a graphical display instead of the text view (from the View class). My code is extremely resilient to the changes regarding the display of data as no other changes need to be made to the code.

Answers to Questions

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?

My code is programmed such that computer players automatically discard the first card in hand and play the first legal play. If I want to have different types of computer players with differing strategies, I can implement the strategy design pattern where I define a family of algorithms that have different playing strategies. In order to do this, I can add subclasses of the Player class. Each new subclass of the Player class will have differing play and discard functions. To allow the computer players to change their strategy dynamically, I will implement a function so that each computer player can change to another subclass of the Player class. For example, I can call the different computer player subclasses as “FirstCardPlayer”, “HighestValuePlayer”, and “RandomPlayer” and define play and discard functions that define different strategies. If I want to changed a strategy as the game progresses, I can change to another computer player subclass, such as from “FirstCardPlayer” to “HighestValuePlayer”, without changing any other information about the player (`hand`, `discards`, `score`, etc.). This implementation will be similar to the `ragequit` function in my code right now, as in the `ragequit` function, Human players are changed to Computer players while keeping all necessary information about the player. To determine when to change the strategy, I can implement another function in the Round class. For example, the FirstCardPlayer can decide to use strategies of the HighestValuePlayer when there are more than 5 cards on the table. Then, I can create a function to determine the number of cards on the table and switch the player class accordingly. Implementing this feature will have a minimal impact on my design structure; there will be more subclasses of the virtual class Player.

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?

There will be no design changes needed to implement the two Jokers; I would only need to change some of the functions. I can add the Joker to the deck as a Card with rank = 0 and suit = 'J'. I can change the getLegalCards function in the Round class so that the Jokers appear in any legal plays. If a Human player plays the Joker, there should be another text prompt that asks for the user input that indicated which card the Joker is taking the place of. This card must be a part of the legal plays, must not be 7S, and should not be from the player's hand. For a Computer player, there should be an algorithm to indicate which card the Joker is taking the place of. The player would not be able to use the Joker if the only possible legal play is 7S. When the Joker card is being added to the vector of Cards on the table, it should be added as the card the Joker is replacing.

Originally in the answer I gave in Due Date 1, I thought of adding another Boolean attribute isJoker to the Card class. However, I believe that this is unnecessary, and I can simply add Jokers as Card with differing suit or rank than the rest of the Cards.

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.

In order to have as little impact on the code as possible when implementing these changes, I should use the MVC model. Using the MVC model, I can change the model, view, or the controller part to make changes without having big impacts on the other parts of the code.

For changing the user interface from text-based to graphical, I can implement an observer pattern as part of the View in the MVC model. With the observer pattern, I can add multiple subclasses of the Observer class that implements different ways of user interface. Different ways of receiving user input can be implemented through the modifications of the Controller class. If I were to change from text-based command line inputs to graphical inputs, such as finger taps or mouse clicks, I can change the `getInput` function in the Controller class to accept different input types. If I wish to change the game rules, I can modify functions in the Model, which is the Round class in my implementation. If the rules affect the way players play their cards, I can modify the functions in the Player class.

Extra Credit Features

a) Preventing invalid inputs

I implemented features that prevent invalid inputs. The program checks for invalid formatting of the card (e.g. 7s, D2, 10H, 10Hearts).

1) Invalid play: If a player tries to play a card that is not in its `hand` or when there are no `legalCards`, the program gives an error message.

2) Invalid discard: If a player tries to discard when the player has a legal play or tries to discard a card that is not in its `hand`, the program gives an error message.

3) Invalid player type: The program gives an error message if the player's input for the player type is not 'h' or 'c' or 'H' or 'C'.

For each invalid input, the program asks for a new command until a correct input is provided.

b) Case insensitive features

For Card suits, I implemented a case insensitive feature. For example, user can input "play 4s" instead of "play 4S".

c) Memory management

I handled all memory management via STL containers and smart pointers.

Final Questions

(a) 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?

The most important lesson I learned about writing large programs is the significance of the UML and designing the structure before I start writing the code. As I had created an UML for the project beforehand, I was able to follow it while creating my classes. It was also easier to see the relationships between the classes. As I was writing a program with numerous functions, classes, and files, it would have been extremely hard to follow if I did not have an UML or a design in mind. Even with the initial UML, I constantly had to go back and change my original design. As I was writing many different functions in different files, I often forgot what functions I wrote and had to go back to remind myself of what the functions do. Also, when I was adding a functionality, I did not consider other parts of the code and I simply added it wherever I needed the functionality. Because of this, I ended up writing some duplicate codes that could have been done with one function. At the end, as I went through my code, I had to change multiple parts of my code as there were some duplicates, and functions that essentially could have been done in one function. This helped me realize the importance of documentation not just at the beginning of the project, but during the process of writing the code. It would have been easier to keep track of the functions I write if I continued updating the UML or kept a document with brief explanations of the functions. Furthermore, I realized that it is important to think about the functionality and its relations with other classes in the code before I implement it. This way, I will be able to save a lot of time and make my code clearer and well structured while keeping high cohesion and low coupling.

(b) What would you have done differently if you had the chance to start over?

If I had the chance to start over, I would add another private attribute to the Round class for the pointer to the current player. Right now, I have a playerList, which is a vector to the players, and I pass in the current index in order to find the current player. I think a more effective and comprehensible way will be

to add a pointer attribute for the current player. I would also want to try working in a group. When I get stuck while debugging or writing the code, I think it will be effective if I have a partner to discuss my problem with. I also want to have the experience of working with other people to complete one project. Furthermore, I believe that I would be able to enhance the code more if I had worked with more people. I wish to enhance the code with graphical display, or with different strategies of the Computer players.

Conclusion

In conclusion, I implemented the MVC model with the observer pattern and used object-oriented programming that allowed my code to have high cohesion and low coupling. Looking back at the project, there are some parts that I wish I had done differently. However, this was my first time writing a large program with separate compilations and object-oriented programming, and there were a lot of lessons learned from completing this project. It also helped me understand the importance of design patterns better and sparked my interest in writing large programs.