Jack Zhang
12/16/20

Design Plan

Overview

The central class in my project is the Game class. This is where I implemented core functions that initiate, navigate, and terminate the game. For instance, initGame() is used to start the game, asking the user to invite players, while playGame() manages the actual course of the game, allowing for the autoplay of 7S at the start of each round and checking, at each turn, if the round is over and, after each round, if the game is over. Those are two of the pivotal functions in this class, and they are the only functions called in main.cc after the user starts up the game.

In the Game class, there are numerous other helper functions to aid in carrying out these two important game mechanics. For instance, the function initDealing() distributes cards into each player's hand when called by initGame(), while others like isRoundEnd() and gameResults(), which prints out the results when a winner is found, are called at varying points in playGame() to offer their functionality. These helper functions, in turn, rely on functions found in other classes like the Deck and Player classes.

The deck class contains functions that interact with the deck of cards. Some of the notable functions are shuffle(), allDeckStr(), getDeal(), isCardIncluded() and addCardToTable(). allDeckStr() is used to print out the contents of the ordered deck. getDeal() is used to distribute 13 cards to each player, and it is called by initDealing() in the game.cc. Meanwhile, isCardIncluded() checks if a card numbers among a given vector of cards. As an example, it is used by isValidPlay() in the Player class to check if a card is in the given player's hand and if this card is among the legal plays this player can make. Another universally helpful set of functions in this class are the ones that convert between elements of string class to cardInfo form and vice versa, allowing for other functions to output information in the proper form. As such, hopefully this shows how the functions in Deck are very useful in many other portions of the project, which rely on them to carry out game elements.

Although some of these elements are executed at a broad level in the Game class, many are also carried out a smaller level by individual players, at individual points in time. This is the idea behind the Player class is that each player has their own set of information (total score, score for this round, id, player type, etc.), as well as actions they can perform at a personal level. To illustrate, some of the functions in this Class include play(CardInfo cInfo), discard(CardInfo cInfo), getId(), and setCurScores(int s). As can be seen, these functions' abilities range from allowing for essential game commands each human player can make, like play, quit, ragequit, discard, etc., to providing information about the current player, as pertains to their id, player type, to changing fields like their score.

However, the most important functions here are autoPlay(), prePlay(), and isValidPlay(PlayInfo pInfo). autoPlay() is essential as it implements the behavior of the computer players, instructing them to play and discard the first cards in their hand. prePlay() is needed to obtain, store, and eventually return some of the information about the current game state that greets a player when it is their turn, including the cards in their hand and their legal plays. Both of these functions rely on the last, isValidPlay, which checks a play is valid given the state of the cards currently on the table. Across these functions, many rely on the Deck class, as previously noted. Additionally, many of these functions find usage in the Game class, although the opposite is not true, highlighting, once more, how Game is the central class that utilizes others like Player and Deck for added functionality.

As for how this project handles user input and commands, I have constructed a Command class, which houses various virtual classes that correspond to each of the possible user commands, like PlayCard, RageQuit, and DeckPrint. This class and the class Inputhandler are fundamental aspects of the command design pattern, which I built my project using. Each of these commands also interacts with another field called next. Certain commands like play and discard, after being executed, transitions the game to the next player. On the other hand, other commands like DeckPrint to not do this, so in these cases, Next is set to false. These commands are, in turn, evaluated by inputhandler. Going back to the playGame() function from Game, whenever the human player enters input, such as the play or discard command, this, command is sent to inputHandler. The inputHandler is used to check if the command is valid, returning an error message if not, before returning the command back to playGame() so that it can be easily executed.

One difference between my final project and the one I had planned initially is that I did not incorporate the Singleton design pattern while developing my Game class. I found that it was not necessary and that introducing it would likely just overcomplicate the class. Furthermore, I chose not to create a ComandHistory class in the end. Originally, I had considered creating it because, while doing research, I thought it was an important part of the command design pattern. However, because there was no retracing or redoing of player's executions and commands in this game, I realized it was not all that necessary.

Design

Because this project was a lot larger than any of the assignment questions I have tackled before, I decided to approach the code structures I would implement with the mentality of simplifying things as much as possible. This meant that for each class, I built a multitude of functions, rather than a couple bigger ones. This made it a lot easier to clearly understand how my functions worked together and revise them whenever bugs arose—without possibly altering other aspects of the implementation. This also allowed for me to recycle certain functions a surprising number of times because many of the helper functions I developed were very simple but were needed across different classes.

This leads to my choice of using the command design pattern, which paired very well with the requirements of this undertaking, in the sense that there was a set selection of possible commands that needed to be passed around. In a similar vein, having a concerted design plan and a relevant design pattern helped me decide how to structure my classes in the most effective way possible. This prevented me from having any one disproportionately large class and, instead, helped me to implement classes that were all reasonably sized with appropriate and beneficial interactions between one another. This, in turn, increased cohesion between my classes, as evidenced by how many of my classes borrow functions from others to run more smoothly.

As for specific coding techniques, I frequently made use of vectors because they excelled at modelling the idea of cards being part of a larger arrangement, such as the cards in a player's hand and the cards in various suits currently arrayed on the table. I also knew that it would be a lot simpler to iterate through a vector to search for a specific card or general element than to do so using something like arrays. Using vectors also made it far simpler to push new cards into vectors of cards Furthermore, because many of the functions across classes dealt with numerous conditions, such as my inputHandler, I used a lot more switch statements and try while loops than I would normally during assignments. This made it more efficient for my functions to work through these operations than if I had used regular if conditionals, with which I was more comfortable. In addition, I frequently made use of shared pointers, such as in my inputHandler class as well as in some functions, like Player* getNextPlayer() in my Game class. This helped with my referencing so that I could better avoid memory leaks. In relation to error handling, it helped that I had a central location (my inputHandler class), where I was able to check the validity of user commands. Additionally, because of the high cohesion between my classes, I was able to borrow validity-checking functions like isValidPlay(playInfo) from other classes to streamline my error checking.

Resilience to Change

Because I structured my project around a command design pattern, I have a lot of control over the individual commands of the game specifications. Thus, if the rules were changed to introduce new commands, it would be a simple matter of adding a new command in my Command class and inputHandler class (so that playGame() can check if it is appropriate to execute this new command at any given point) and implementing it. If it is like the other commands, which are executed by human players during the game, I would add it as another function in my Player class as I have with the other commands.

Similarly, because the structure of my project allows for good cohesion, seeing as many of my classes interact strongly with each other to execute the game, it would be easy to incorporate new elements, which, in turn, can make use of the cohesion between existing classes, allowing for easier implementation of these changes. This is helped along by the low amount of coupling, as seen in how the various classes freely call functions from other classes without any worry of the other classes' different implementation. For instance, adding in new variants of computer players is simplified by the fact that I can just implement a new play function for this new type, making use of existing class relationships to integrate the new player type in without needing to worry about issues with overlap.

Another way in which my project may be resilient to change lies in the fact that it would be pretty good at updating the UI or even adding a GUI. This is attributed to low coupling in my project and the fact that the existing inputHandler class is a standalone, so I can just create another class that deals with input dealing with the new GUI.

Answers to Questions

<u>Question 1</u>: 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.

A good pattern to use is the command design pattern. For one, it helps to minimize coupling as it separates the classes that implements an operation from those that call upon it. Because of this separation, it makes it easier to adopt a new set of rules. If we wish to code, for instance, a new functionality, we can implement this operation in a new class and simply allow other classes to invoke it, at will. Likewise, if the new rules call for us to invoke existing methods in new ways, we can do so without worrying about altering the code itself in any way. Furthermore, because this pattern allows us to defer certain command invocations, if we wish to, instead, alter the order in which we call on certain functions, we can do so easily, too, by creating a rollback system.

In the case of my classes, it helps that my five commands (the concrete command classes) are decoupled from the command interface. This enables me to introduce new commands in the future if I choose without affecting any existing code. Additionally, I have that my class Players maps to the Receiver class in the command design model, while my Game class contains the functionality of the Invoker class. Lastly, the InputHandler class plays the role of the client in the command design pattern.

<u>Question 2</u>: 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?

I think the class structures I have right now are satisfactory, seeing as I was able to implement two new player types. Furthermore, the setup I have presently allows me to access information about the game state and players' scores at nearly any point in the game, making it easier to assess different game checkpoints to change strategies. To address the issue with dynamically changing strategies, I might adopt a procedure like the one below:

In my info.h, I would declare an enum class like the one below:

enum class PlayerType { Computer, Human };

I would add a private property in the Player class called <u>playerType</u>. If the computer player's strategy is liable to change, I may consider adding another enum class, one that is called strategy, like the one below:

enum class Strategy { level1, level2, level3, level4 };

Afterwards, I would also need to add a private property called <u>level</u> in the Player class. Then, as the game reaches certain checkpoints, I can change the value of strategy to a different level. Also, in the Player class, I would add a level strategy for the Play and Discard methods.

To assess these checkpoints, I might look at data like the current scores of the various players, information which is readily accessible in my current code structures.

<u>Question 3</u>: 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 play

Currently, I have a global field called playerType, which corresponds to the type of every player. My classes are set up such that this field can be changed at any point for a player. Thus, if a player chooses to rageQuit, I have a function called rageQuit() in my Player class. This changes the player's type to Computer and allows for the program to continue running. This is because this change occurs inside of playGame(), meaning that the game continues without issues.

Extra Credit Features

I implemented two new player types, called the Senior and Principal player types. Essentially, the Senior player type is a slightly smarter version of the computer player, who always plays the largest legal card when possible. The Principal is slightly smarter than the Senior, always playing the largest legal card and discarding the smallest card when possible. The seniorPlay() and principalPlay() functions are, thus, the Senior and Principal equivalents of the autoPlay() function, which works for computer players.

Because of my project structure, it was not very challenging to introduce a new playerType into the game. Since these were player types that were already somewhat similar to computer players, I was able to create a new function for these playstyles in my Player class based on the function I had already made to automate gameplay for computer players. I was then able to add these types to my initPlayers function in Game.cc, allowing for the program to program to accept representative characters for these player types as input when inviting players at the game's start.

Final Questions

1. This project taught me a lot about how to properly plan how to write a large program. Previously, in assignments, a lot of the class structures, relationships, and behaviors are established for you. However, this project gave a lot more freedom, forcing me to view these large programs from new angles. For one, it reinforced, to me, the virtues of writing many simpler functions rather than large, inflexible ones. This aided me a lot as it allowed me to reuse the same functions again and again, making the coding process a lot more efficient. Furthermore, it taught me about how to devise solid class relationships beforehand. To elaborate, I had to consider which classes were the central ones, and which served to offer heightened functionality to the more important ones by allowing for the use of its functions.
2. If I had had the chance to redo this assignment, I'm unsure what I would have changed about my process except start it earlier. This project took a lot of time to plan out and implement alone, along with the time it took to familiarize myself with various applicable design patterns like command pattern, which I ended up using.