CS 246 Straights: Final Project (DD1)

Design Document

Rohan Ravindran

Wednesday, December 15th, 2021

Introduction

This design document will cover the design, architecture, implementation, and overall process that went into the creation of this implementation of the Straights card game.

Overview

This project follows an Object-Oriented design structure, using classes and objects to organize information and maintain an extensible codebase for future upgrades. This design follows the Model-View-Controller design method.

The project consists of multiple classes, the main ones being StraightsModel, Player, and View. The StraightsModel class is the main hub of functionality for the game, and handles the information regarding players, the game pile, the deck, and is where the main portion of the game is run. The Controller and View classes both interact with this class and serve as the functionality that allows the StraightsModel class to interact with players. The Controller is an abstract base class, with its children implementing specific methods pertaining to how the game should receive input from the user. The View class is also an abstract base class that interacts with the StrightsModel class and updates information to the implemented method desired. Moreover, the View class is an Observer and the StraightsModel class is the subject in an Observer pattern intended to make it easy to add more views to the application in the future, making it more resilient to future change.

Design

This project uses a myriad of Design patterns that help with solving certain design problems and make the code easier to modify in the future. The first main design pattern that this project implement is the Model-View-Controller (MVC) design pattern. Moreover, to implement the ability to add more views, I have implemented the Observer design pattern to keep track of the different views the application has, and the Model class can update them accordingly. Moreover, the Strategy design pattern was also used in the Controller classes in order to make receiving user input from any device is easy.

The use of multiple classes and Design patterns make this project quite resilient to change and easy to upgrade and extend in the future as the needs of potential players change. A developer can easily add another view to this program by extending from the View class and adding it using the addView method in the Model class.

Resilience to Change

I designed this project from the beginning to be quite extensible and resilient to change. In situations where a new feature would need to be added, or an existing game feature needs to be modified, it can be done with a relatively small amount of work. The following are examples of the specific design choice that allows for this extensibility.

MVC Design Pattern: Using this design pattern allows for great extensibility in terms of the 3 main aspects of the design which are the Model, View, Controller. As mentioned before, the Model and StraightsModel (inherits from Model) handle the logic of the application and serve as the hub for the game logic. The model holds pointers to the corresponding Views and Controllers, and both interact with the Views and Controllers to run the game accordingly. The View and Controller are both abstract base classes that can be extended to implement specific custom Views and methods of receiving user input, making the application very extendable for future developers.

Observer Design Pattern: I also utilized an Observer design pattern as mentioned previously to make adding new and updated views to the application quite easy.

Use of Enum: I used enum classes to make it easier for a future developer who wants to change this program to understand how the code works. I utilized enum classes for the Suits of cards, and types of a user action, making this program quite extendible and easy to work on.

Answers to Questions in Project Specification

- 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 the project, I have implemented multiple design patterns to help with the issue of change resiliency. I have implemented the MVC, Observer, and Strategy design patterns. In the example given above of modifying the user interface from text-based to graphical, my design accommodates for this change as I used an Observer design pattern (similar to Assignment 4), that allows for multiple user interfaces (observers) to attach to the Model class (subject), and be notified of changes accordingly. This way, many user interfaces could be added to the game, with little to no modifications to the main program required. In the case of changing the game rules, I utilized the MVC design pattern which contains a method named “play” within the Model class. This method is responsible for the portion of the game that validates the user input and enforces the game rules. Since this method is virtual, if we wanted to change the rules of the game, we could create a subclass of Model, and re-write the play method with the updated

rules for the game, with little to no modification to the rest of the program. Therefore, the classes I have created fit will even in the situations described above.

- With using the MVC design pattern approach, the coupling has been reduced as different objects/classes mainly communicate through the passing of parameters. Moreover, cohesion is also high with using this design pattern as it clearly separates the Model Class (handles logic and communicate to the view), the View Class (display information to the user), and the Controller Class (Receives input, communicates to the Model). This separation of responsibility makes future modifications to this program easy since it was built to be very modular and extendable. Moreover, this design implementation closely follows the Single Responsibility Principle, which is key in Object-Oriented software design. Therefore, this design choice used to structure the game is the ideal one as changes to the program do not require too much additional work, and it provides ideal qualities of low coupling and high cohesion.
-

- 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?
    - Implementing this feature would not drastically affect my class structures as the design already accommodates this. Each player, including the Human player, a subclass of the abstract Player class, which provides a pure virtual method named getAction. Within the method getAction, subclasses of Player provide an implementation of how cards should be selected, and return action or card they would pick during a given turn. Then, each strategy that would like to be available for a computer would be a subclass of Player, defining its card picking strategy behavior within the getAction method. Then, at runtime, if a computer would like to change the strategy, one can just construct a new class of the new strategy, and copy over the relevant details to this class. Then this new class would have the same details as before, but with the new strategy defined in the getAction method of the class. Therefore, this method of dynamically changing a computer's strategy during a game would require the creation of additional subclasses that describe the strategy, and the creation of robust copy constructor methods, but no other modifications to the overall program structure.
    -
- 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 general design would not need to change, however, modifications to the HumanPlayer class' getAction method would need to be modified in order to

prompt the user to ask which card they would like the Joker card to be when a player decides to play the card. Additionally, the ComputerPlayer class' getAction method would have to be modified to support playing the Joker card. Aside from those two changes, no other modifications would need to be made since we are using dynamically sized arrays (vectors), and if the User or Computer decides what card they would want their Joker to become when it is played, it does not modify the game in other aspects whatsoever. Therefore, only the HumanPlayer and ComputerPlayer classes would need to be modified.

Extra credit features

- No Explicit Memory Management
    - In this project, I exclusively utilized smart and unique pointers when storing objects on heap memory, allowing for no explicit memory management required.
    - Moreover, I occasionally did pass small objects by value in the appropriate cases.

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 biggest lesson I learned while writing this large software program was the importance of planning before coding. In working on previous projects throughout the course, I tend to jump into coding right away before thinking of the overarching architecture of my solution, and if there was a better solution that existed. From this project, I learned how to plan things out and take my time when it comes to designing software architecture, and then implementing it methodically.
    - The second lesson I learned from working on this project is that when writing large software projects, spending time on it and not rushing the process is key in producing a well-crafted solution. Towards the end of this project, I can admit that I was rushing portions of the solution in order to get it submitted in time. If I had more time and prioritized my time earlier, I believe I would have had a cleaner and more extensible solution.
- (b) What would you have done differently if you had the chance to start over?
    - If I had to start over from scratch, I would have spent more time on the DD1 deadline planning out and determining how exactly my classes were going to interact with each other. I would have spent more time creating more detailed UML diagrams depicting the architecture of my solution, and I also would have assessed whether or not my initial proposal was "over-engineered" as that would have saved me a lot of time in the implementation process, where I was wasting

time implementing a more complex solution when a simpler and more elegant solution existed.