CS246 Final Project: Design (Hydra)

Ethan Santos

2021-08-13

1 Overview

From the start, this project was built with MVC architecture in mind. As such, I will begin by explaining the structure of the classes that the program "model" consists of.

When designing the structure of this project, I found that the best approach to gain an intuitive understanding of the program flow was to structure the program similarly to how the game would be structured if it were being played in real life. The program model is based on this:

- A Hydra object represents the game itself, and contains pointers to each player in the game. Pointers to each head are also contained in this object, along with methods that implement the game logic, and work as an interface between the controller and the model.
- Each player is represented with a Player object, which has the draw/discard piles associated with the player, in addition to other player information. Methods involving the individual player's state are implemented here.
- All piles of cards are Pile objects, which are functionally a stack of Card objects, along with other methods involving a pile (such as shuffling the pile, randomly or with a seed).
- Finally, Cards objects represent a card, with value/suit fields and the required get/set methods and implementation of the Joker's special behaviour.

Now that I have a functional model to represent the state of the Hydra game, I implemented the view and controller parts of the MVC architecture. Both were implemented as part of the View class, and they control I/O to input and output streams passed in through the constructor, as well as program flow and game logic. So, the View class has methods to play the game, and the necessary methods to recieve/output messages. View interacts with the model (Hydra/Player classes).

2 Design

To begin, from a high level, I wanted to use MVC architecture to help me to organize the different aspects of the program. In order to keep the game model seperate from I/O and game logic, I used my View class to handle the latter two responsibilities, and the Hydra/Player classes to handle the former. This allowed me to work through design challenges in my project in only one of these

contexts, rather than trying to find solutions that keep all of these responsibilities in mind. Thanks to this, the challenges I will discuss in this section tend to be isolated to one of these areas (game model, I/O, or game logic).

One problem that I ran into was in my I/O code. I wanted to be able to handle a variety of invalid inputs without crashing. This required me to utilize exception handling to throw and catch exceptions when necessary, in order to handle invalid inputs.

Another problem that I handled had to do with how my game model communicated with my View class. I needed the View class to be able to access a variety of information about the game state, such as the cards in a player's draw pile or the currently active heads. However, giving external classes access to this data would break encapsulation. So, I made View a friend class of the game model classes for my program, Hydra and Player.

Finally, I used smart pointers (specifically, unique pointers) to handle memory management. Since each card only needs to be referenced from a single context at the same time (i.e. it can only be part of a single pile at a time), we only need one pointer to it at a time. So, I found that unique pointers would be a good solution to handling memory management without having to explicitly delete dynamically allocated memory.

3 Resilience to Change

Starting with the game model components of my program, this project shows an ability to change and accomodate new features through high cohesion, low coupling, and a structure designed with the single responsibility principle in mind.

The game model shows high cohesion through the hierarchical nature of the classes. Each piece of the model is a smaller part of a greater whole. A Hydra object consists of Player objects and Pile objects, with Pile objects consisting of Card objects, and so on. Each class represents a single "part" of the Hydra game, and its responsibilities extend no further than the responsibilities of that part. For example, the Pile class has methods to build and maintain a pile of cards. It does not have methods to work with individual cards, or to interact with the Player object that might utilize the Pile. This level of cohesion means that we can make significant changes to the program by only touching a single class, and without affecting other functionalities of the program, since each class has a single job that it is meant to fulfill.

Next, we can show that this project has achieved low coupling by showing that, outside of the high coupling between the View class and the game model (necessary to pass information to output), classes only communicate with each other through function calls and parameters. For example, to add a Card to a Pile, we can use a Pile object's push() method to push a card onto the stack of the internal vector "cards". We do not give Cards direct access to the "cards" vector-instead, we force it to utilize methods that we have defined for it to use. This low coupling also allows for resilience to change by allowing us to make changes to the inner structure of a class, and remain confident that no external classes are still using the "old" private fields of the class. That is, we can focus our changes on individual classes rather than on the program as a whole.

This brings us to the single responsibility principle, which states that a class should have only one reason to change (as it is responsible for exactly one task). Through the implementation of the MVC architecture, we have separated the game model from the rest of the program, giving it only one responsibility: maintaining fields related to the state of the game. We further break down this responsibility in the classes that this model consists of. The Card class needs to change if and only

if we want to change how cards store information, the Player class needs to change if and only if we want to change what information is related to the player or how it is stored/transferred, and so on. This is very important in allowing us to easily make changes to the program by ensuring that any change we want to make corresponds to some class in our program.

We have shown that the game model module of the project is capable of handling changes to existing code as well as addition of new features. We can also consider the various ways in which the View class, which handles I/O and game logic, allows for changes.

We can consider the individual methods of View. Most of the important input and output actions that are performed by the program occur in dedicated I/O methods, such as outputHeads() or chooseNext(), which are an example of a method used for output and input respectively. Meanwhile, the game logic occurs largely in the method playGame(). This means that changing the I/O of the program requires changing the methods corresponding to I/O in View, which changing game logic requires making changes to playGame(). To take it a step further, if one wanted to support various different types of I/O or game logic in the same program, we could create various different children to the View class that would override View's functions in different ways. This is an example of the Strategy design pattern, and is an example of how changes can be implemented using the View class.

Finally, I will consider a few areas in the program that were made arbitrary in order to support changes made in the future, if necessary. These decisions made during the design process allows for certain changes to be made to the program, if desired.

One such area is in the implementation of random shuffling in the Pile class (shufflePile()). While we typically want to shuffle the pile as randomly as possible, we also might want to have the option of utilizing a seed, so shufflePile() was designed with this in mind, allowing for new opportunities for change, if needed. The method will by default use the seed std::time(0), but it also allows for a seed argument.

Another area that allows for more flexibility is the constructor for View. When creating a View object, we must pass it an in and out stream. While we typically want in to be standard input and out to be standard output, allowing these arguments to remain arbitrary allows other options to be available, making it easier to make changes regarding where input is coming from or where output is going.

4 Answer to Questions

4.1

Q: 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: Utilizing the MVC architecture allows the changes to the interface (the view) or the changes to the game rules (the controller) to have little impact on how other aspects of the program work. Through the single responsibility principle, if we are successfully able to achieve high cohesion and low coupling on our view and controller classes, then we can easily make changes to one of these classes without impacting other parts of the game, as we only need to make changes to a single class. This is implemented in my program structure though my View class. It is seperate from much of the gamestate information in the Hydra and Player classes, so I can make changes to the

game rules and interface in the View class without impacting too much of my code.

4.2

Q: Jokers have a different behaviour from any other card. How should you structure your card type to support this without special-casing jokers everywhere they are used?

A: Since jokers have a "suit" of J, we can easily identify jokers whenever they occur. I can implement jokers by adding a method to the Card class that allows me to modify the value of a card. So, I can treat each joker as a 2 (so that it is a 2 when drawn as a head) until it is drawn to a players head. Then, I simply need to change the joker's value to the desired value when it is placed on a head, and change it back to a 2 when it goes back to the discard pile. With these components in place, no special-casing is required.

4.3

Q: 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?

A: If one wants to allow both human and computer players, you would first want to have a seperate abstract Controller class as per MVC architecture that handles the game logic. Then, we could make concrete Controller classes for human and computer players, which override methods from the parent class in different ways, so that how they "respond" is different. That is, we implement the Strategy design pattern, by having a different way to solve the same problem depending on whether or not the player is human (ask for player input) or the player is a computer (run some function to recieve computer input). We can implement different computer strategies by having different concrete "computer" children of the Controller class, that all override its methods in different ways.

For example, say we have come to a point in the program where we need to ask player 1 for their move in the game. This requires us to call some function PlayerMove(), which asks the player for a move, and returns the number corresponding to the move:

- The HumanController class, derived from Controller, has PlayerMove() ask the player to enter an input into standard input.
- The ComputerAController has PlayerMove() execute some algorithm A to return its move.
- The ComputerBController has PlayerMove() execute some algorithm B to return its move.

Ultimately, this way to structure my classes would require me to fully seperate my View class into an I/O class and a game logic class, so that I have a "pure" Controller class to work with.

4.4

Q: 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 for the information associated with the human player to the computer player?

A: As mentioned above, the MVC architecture allows for the class structure to be able to easily support this type of feature. If I implement the described Strategy design pattern as described in the question above, then the only difference between a human and a computer player is the Controller object being used for that player. Since all of the game data is stored in the model (Hydra/Player classes), we can simply change the Controller being used for a player from Human-Controller to a ComputerController (i.e. choose a different strategy). That is, we don't actually need to transfer the information from the human player to the computer player. We instead can transform the human player into a computer player by using a different strategy.

5 Extra Credit Features

There are two extra credit features I would like to discuss:

5.1 Memory Management

The first extra credit feature that I implemented in my project is the challenge presented in the project guidelines document. I managed memory in my project using only smart pointers and STL containers, rather than through the use of the new/delete keywords.

The challenging part of implementing this feature was learning how to adapt to using unique pointers. In Hydra, many of the concrete objects in the game are unique by nature; only one of each card actually "exist" when playing the game in real life using physical cards, so it is intuitive to refer to cards in the Hydra program using unique pointers. However, utilizing unique pointers required a solid understanding of move semantics and rvalues/lvalues in order to keep track of "where" each pointer actually is at a given point in the program (that is, which variable is actually holding the unique pointer).

This feature was implemented by keeping it in mind from the very beginning; unline many of the other potential enhancements, even with good object oriented design, it is not easy to implement such a fundamental feature (changing the structure of pointers) once the structure has already been laid down for the program. As such, while planning the structure of the project, I considered how I could perform certain tasks using unique pointers (ex: moving a card from one pile to another) and built functions accordingly such that they could accomodate smart pointers.

5.2 Cheats: A "Sandbox" Mode

The next extra credit feature that I implemented was a "sandbox" mode that allows the user to change the gamestate in a variety of different ways. It is helpful for both testing the program as well as showcasing its functionality. As such, it is an extension of the testing mode, and adds certain extra features to it, as all cards can be chosed before being drawn in this mode. At the start:

- The number of cards in each player's deck can be chosen
- The number of active heads can be chose
- The number of cards in each head can be chosen
- The topmost card in each head can be chosen

The most challenging part of implementing this feature was trying to reduce coupling as much as possible between the View class and the Hydra and Player classes, as well as trying to keep encapsulation in Hydra and Player. To directly manipulate parts of the game such as the active heads or each player's draw pile required my View class to have a high level of access to both the Hydra and Player classes, which is not something that I want classes to be able to have easy access to. I found that allowing View to make these changes to the state of Hydra/Player would break encapsulation, and instead of trying to prevent this, I found a different solution. I allowed View to have direct access to the private fields of Hydra/Player by making it a friend class of both of these classes, thereby breaking encapsulation, but only for View; not for any other classes that might be implemented later, which allows me to keep encapsulation so a certain extent.

6 Final Questions

6.1

Q: What lessons did you learn about writing large programs?

A: As I did my project by myself, I had to build a large program from the ground up, and I had to design my program's structure by myself. One thing that I found, especially when working by myself, is that since there is nobody that I have to check in with or communicate with, it becomes very easy to just start writing code without any planning or foresight. The project didn't necessarily have to be split up into seperate pieces like a group project would, so I found it difficult to refrain from writing long functions to solve problems rather than break up a problem using object oriented principles. As such, one important lesson I have learned about working on large programs is the imporance of planning and utilizing design principles to seperate a project into smaller pieces. The benifits of this include:

- easier to conceptualize problems if they are broken down
- much easier to implement object oriented design
- better readability and easier to maintain code

I found that a lot of the problems I ran into would have been prevented if I had simply planned out my program beforehand, instead of writing long functions that are not flexible.

6.2

Q: What would you have done differently if you had the chance to start over?

A: As mentioned in the eariler question, I would have found a lot of value in spending more time at the beginning of the project on planning out my program structure. Doing this would allow me to both increase cohesion while decreasing coupling:

- I could increase cohesion by choosing well thought out classes that perform a single purpose, rather than choosing classes that work functionally, but have too many unnecessary parts.
- I could decrease coupling by finding better solutions to how my classes interact with one another. In a few cases, I ended up sacrificing the encapsulation of my classes in order to make my program work, due to bad planning.

Next, I would choose to separate my View class into a "View" class for I/O and a "Controller" class for game logic and program flow. As mentioned in section 4 of this document, having a seperate Controller class would help a lot in allowing for added functionality that is very difficult to implement given the current structure of my program. Doing this would more accurately follow MVC architure, and would allow me to implement various bonus features much more easily, including the "Cheats" feature that I added.