Ada Vu, 20820832, a6vu

Design Document

introduction

The final structure and design of my project follows an MVC design pattern. Initially, I was planning to implement the project with observers however, I felt that I was not very familiar with observers and opted for an MVC design pattern instead. In my initial phase I was in between MVC and observers so my final structure and design is not far off of from my design from Due Date 1.

Overview

My project contains different aspects that are responsible for different things in the program. Following an MVC design, I will explain the model first.

An aspect of the model is the card class. The card class is responsible for holding the integer card value as well as the string suit (along with a string symbol for printing purposes). The functions within a card class are simply to retrieve the card. For example, retrieving its integer value or its suit type. As well, as setting the card to a desired integer value or suit type. Anything dealing with cards will use the card class. Furthermore, I did not implement a deck class because a deck is simply a vector of cards. Thus, wherever a deck is required, the deck will be a vector of cards.

Another aspect of the model is the head class. The head class has a generalization association with the card class. In other words, a head is also a card. While the head has a composite association with a player. In other words, when a player cuts a head that specific head deck belongs to that player and is not shared with other players. Since a head will become a deck, the head is a vector of cards. Any actions that deal with the head can be retrieved using the head class. In particular, the head class is responsible for holding the head decks, and there could be multiple. Some function actions could be, retrieving the top card in a particular head deck. Or the size of a head deck. Or even cutting a particular head.

Lastly, we have the player class which is a more complex class dealing with player actions. The player class has an aggregate association with the card class since each player has cards. It would not be a composite association as the cards are shared amongst them. A player has many actions within this game so this model for the player class should suffice for all the actions a player is allowed to do. Implementing the player class means understanding what belongs to each unique player. This means that the player class is responsible for holding a unique discard deck (vector of cards), draw deck and reserve pile. The functions implemented would include a player adding to their discard deck or drawing the next card from their draw deck or cutting a particular head. The head class has a cut head function since it deals with heads. But a player also has a cut head function because someone needs to cut the head somehow and that is the player. In other words, any player action is implemented within the player class.

Moving onto the view aspect of my design structure of my program. The view is generally what the user (in our case, the players) would see on screen, it could be the prompts or it could just be the words that appear on screen to welcome the players to my fantastic game! In my case, the main.cc is my view. Most prints and prompts were located in my main.cc since my program will start and end in there. While the main.cc it is running, it will fetch stuff from the controller which will then fetch from my model. For example, when starting the game, the game will prompt for the number of players. Or if the player decides to go into testing phase the game will have a welcome text. Another example would be when the game ends, there will be a text to indicate which player has won. All of these are in the (view) main.cc.

And finally the bulk of the program lies within the controller. The controller is the class that deals with every action on the board. To implement this is to understand how the game works as a whole. Since the main is the view, it will prompt for a player's move. After the player moves by inputting some value, the value is then pushed into the controller to do a certain action. This means a function in the controller will be called. In other words, a controller is like a game board where all the actions of each player are observed. Sometimes the controller will push information back into the view to print it out. An obvious controller action (function) would be to create players or create a deck and distributing it.

Design

In the project, I used smart pointers everywhere that required a pointer. Some pointer examples included a pointer pointing to a specific player or a pointer pointing to a specific head deck. I used shared pointers as opposed to unique pointers because I want multiple pointers pointing to the same resource. This greatly helped solve one major design challenge which are memory leaks. After testing my code through a memory checker (multiple times), it turns out I had no memory leaks and all heaps were freed. If I had not used smart pointers in my program, it might be difficult to trace down where I had memory leaks (if I had any). Using smart pointers, I didn't think much about deleting my pointers and leaking memory because it is almost certain I won't if I use smart pointers.

Another design technique is using separate compilation. In doing this, this creates more organization within the program. Otherwise, without separate compilation the whole program may very well written in one whole function (which we all know is very very bad design). In addition, separating my code into .h and .cc files helped me code quicker (assuming I named most of my functions in a smart manner which I think I did!). If I needed a function from a different class, I didn't have to search for it through all the code of the .cc file. The .h file shows me all the functions in a more short and concise way so I didn't have to go fishing for which function I needed.

Lastly, I was going to add in exception safety to my program but I did not have enough time. However, I do know that exception safety is an important design technique in any program and is one of the aspects I would add in if I had more time.

Ada Vu, 20820832, a6vu

Resilience to change

I believe my program and design is resilient to change to some extent. Since every class is separated to its own .h and .cc files, if there are any changes made, I can easily access the specific file that the change is happening in.

For example, if for some reason the game specifications change and we no longer want to have a reserve pile anymore. In my program, that is easy to change because adding to a reserve is a player action and a controller action. Thus, I will go into my player class to do remove reserve action functions and do the same in my controller class. This also affects my view because whenever a player enters 0, it means they want to access the reserve so now we will add an error message instead. In other words, my program is able to handle change by accessing the classes and files that contain aspects of that change.

Despite it being resilient, I believe there is still room for improvement. In my specific example, I had to access three different classes to change one aspect of the game, a very simple one of that matter. It would be great if it only required one step. To do this, I could perhaps break my classes (specifically my controller class) further down into smaller subclasses. The more classes the better for minimizing coupling.

However, it depends on what the change is. An example to prove that my design is resilient is changing the specification of when to cut a head. In my program, I could easily change that in only one function of the whole program.

As I said, my program is not perfectly resilient however, it is resilient to some extent. Depending on the change, I would say the resiliency of my program changes. Thus, this would be another aspect I could work on to better improve my program to fully maximize cohesion.

I need to talk about cohesion and coupling of my classes. I believe that my program's modules provide high cohesion since each module is very focused on the intentions of the class. For example, the card class is only doing card actions and methods relating to the intention of that class (that being action dealing with a card). The card class is not broad and is very focused. The same goes with the head class and the player class. The only class I would say that has a lower cohesion than the rest might be my controller class. It seems my controller class is doing all the actions pertaining to the board. As I have been mentioning, I should probably break down my controller class further to fully implement high cohesion of all my classes.

As for coupling, my classes are somewhat quite related and dependent on each other. This results in high coupling. However, I do think this is the nature of the game. Each player has cards, so the cards and players are related. The board needs information on all of the other classes so those are related because it is just the nature of the game. This is most likely the main reason why my program still has room for improvement on resiliency. Perhaps there is another way to lower coupling that I just need to think about.

Ada Vu, 20820832, a6vu

Answers to Questions

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.

From experience, I think my structure/design pattern (MVC) is suitable for a game like this however, I do not think I implemented it correctly. If you were to look at my controller class, it is somewhat large. Probably not the best. It would be better if I could cut it down to subclasses and place it in the model aspect of the design. The more modular a program is, the less the impact on the code if rules were to change. So I would continue to say (similar to how I answered my DD1) an MVC is suitable for this program. This is because there is a board and prompts (which can be seen as the view) and then there is the controller which controls the board. The model (which includes the card, head, player class) is what makes up everything else in the back to support the program.

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?

Initially, I did not understand this question (I still don't really understand it) but after completing the program I think I have a better understanding of the question. I will answer this question to how I implemented the jokers. When I am creating the deck, I added two jokers as a card (as required in the game specifications). Now my constructor for the card class takes an integer value and a string suit. For the integer, I set at to 2. For the string suit a set it to "J". Now, every time a player draws a card, I first inspect it to see if the suit is equal to "J". If it is, that means the card is a Joker and I will do the required joker actions. I think in doing this I do not have to special case jokers everywhere except in one place, right when the game starts. Initially, (in my DD1 submission) I was planning to create a Boolean for joker card. I did not go through with this plan because I think that is a harder way to implement the joker card.

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 would have another class that would probably be called computer.h and computer.cc. In this class, I would have functions that pertain to the actions of what a computer would do. Since it is a computer, it would most likely play the most suitable move so I would have to specifically code those types of functions in the computer class. In addition, I would also have a prompt in my main to ask how many computer players there would be. Now in my player class, I will have a Boolean determining whether the player is human or a computer. If the player is human, actions will go through the human controller class otherwise it would go through the computer controller class. Clearly, some things will definitely change in my structure. Similar to my DD1 response.

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?

My answer now will probably be starkly different from my DD1 because I didn't create the game yet so I didn't think about it much at the time.

I don't think my answer is much different from the previous question's response. I will still have a Boolean in my player class to determine if a player is human. I will also have the computer controller (computer.cc) so that whenever a player is a computer it will go through that computer class. Now, the only difference is how to allow a player to stop playing. This can be done by inputting a unique message. For example, when the program prompts "Your move?" and a user inputs "stop" then the program recognizes that and stores all the information of that player and allows a computer to take over. This storing and takeover procedure can be done in the computer class. It will have a function that will let a computer play in place of a human and only run when a human player inputs "stop".

Extra credit features

Sadly and unfortunately, I did not have enough time to do any of the features. I think the first feature (the grammar feature where the program says "an 8" instead of "a 8") would have been fun to implement.

Ada Vu, 20820832, a6vu

Final question

1. One big thing I learned with this project is that what I initially plan is not what will actually come out. Just take a look at my UML, it is starkly different. Some functions I didn't know I would need was implemented. Perhaps, it may be because I'm not used to writing such large programs that I did not plan thoroughly. Or it also may be that I did not think it through enough. But to be honest, it is very hard to think of what functions you'll need until you program and implement it. That is one of the biggest things I realized while doing this project. Another point, it is very important to plan first. Although I did not follow my plan very closely, at least I had an idea of what to do. I truly believe that if I did not plan and jumped right into coding, I would be confused and stuck for hours trying to figure out. It was a very good idea to sit down and have at least a big picture of what you'll be building. I mentioned the stark difference between my initial UML and my final UML however it is not different in terms of how many classes there are, it is different in terms of the functions within those classes.

2. I don't think I would have done anything differently if I started over. I believe I planned and carried through with the project quite smoothly. Perhaps I would have started earlier and make use of the full two weeks instead of a week. This way, I would have more time to implement some extra features as well as add exception safety which are currently lacking in the program. As well, fix the one bug I found (as mentioned in the demo). Also, I would have a different mentality going through with the project. At first when I completed the most basic part of the program and getting it to run and compile, I thought I was finished most of the project. However, I greatly underestimated how long it would take to fix bugs. It took me a full day to catch them all (gotta catch em all! POKEMON!). I think that is all.

Conclusion

All in all, I really enjoyed making this project. It was surprisingly a lot of fun. I felt really accomplished implementing feature after feature. But it was fun until I realized I was on a time crunch and I can't code at my own pace since I have 10 million other assignments due within the week. (I'm just kidding). But to be very honest, I really liked this course. So thank you to all the instructors and TA's who continuously supported us!