Daniel Akbarzadeh

Overview

The overall structure of my program is quite simple. Simple Rank and Suit enumerations are used in the creation of a Card class, which is used both in the makeup of the Deck class and particular attributes of the Player class that represent individual cards in real life (reserve and card currently in hand). Jokers are a subclass of Card, with some special functionality of their own. The game itself is implemented through two classes: HydraController (which receives inputs and takes care of displaying the game and the prompts) and HydraModel (which stores the game's state and mechanisms by which rules are checked and performed on this state information).

Design

Outline and Changes in Structure from DD1

This section augments the "Overview" section above.

My design used a heavy variation on MVC. My DD1 UML diagram showed a full MVC implementation of Hydra with HydraView, HydraController, and HydraModel classes with HydraView inheriting from the Observer class while the HydraModel inherited from the Subject class. The changes I decided to make to this design are as follows:

1. I combined the HydraView and HydraController classes. This is because their functions, in this context, are strongly linked: output being provided to the players depends solely on input being available, with, for example, the next showing of players and heads never occurring if a player does not input their move into the game. Thus, the HydraController class takes care of both displaying outputs to and receiving inputs from the player, in the correct succession.
2. I removed the Subject and Observer classes. The use of these was mainly to alert the hypothetical HydraView class to any changes in the HydraModel – however, now that the view is being managed by the Controller, and there are no other classes to alert, I found it more logical to omit the use of the Observer and Subject classes entirely.
   This inherently changed the entire dynamic of the design – HydraController can be a (very advanced) wrapper of sorts to the HydraModel class.
3. The only way cohesion was lowered in the design of the classes was between HydraController and HydraModel, **and this occurred for good reason**. Cohesion was slightly lowered by public methods in the HydraModel receiving the inputs for (a) the player's choice of rank & suit for cards in testing mode and (b) the player's choice of rank for jokers when placing them on a head. Furthermore, the count of the number of cards left for a given player to play during their current turn is determined by the HydraController class as it goes through the necessary aspects of taking inputs and printing outputs for a player's turn. The reasons for these shifts are simple and intuitive:
   a. Although the flow of the game is controlled by the HydraController, the functions behind determining whether a desired move is valid, performing a specific move, cutting off heads and swapping a player's hand with their reserve are all implemented as part of the HydraModel (as these functions should be). It is within these functions that I that I set values for the jokers (as they are being placed on a head) and take values for the cards when in testing mode. Thus, I felt as though it would be smarter to natively implement these reusable functions within the HydraModel rather than adding a pointer back to the Controller, increasing coupling, and then using Controller-based functions to take these inputs.

b. In testing mode, changing cards from Jokers to non-Jokers (and vice-versa) requires completely reallocating space for either a Card or Joker class – so it made more sense to me to perform the action of taking the user's choice of card and then reallocating the Card accordingly within the HydraModel class, which obviously has direct access to each Player object and can mutate them much more easily.
c. As mentioned above, HydraController controls the flow of the game – it determines which player's turn it is and prints output for and takes inputs from that player accordingly. As such, it should be able to control every "round" of a player's turn, i.e., every time the player needs to play a card within a given turn. To be able to do this, it should maintain a count of how many cards a player has yet to play, and decrement/reset this count appropriately at every move. To be clear, however, this count is originally determined at the beginning of every turn by a call to a HydraModel function that returns the number of heads – thus HydraModel is not *completely* absolved of the responsibility of determining the number of cards a player must play in their turn.

Cohesion and Coupling

Cohesion is high and coupling is low throughout. All main actions that HydraController needs of HydraModel are performed via calls of HydraModel's methods, allowing the two classes to be loosely coupled. Furthermore, other than the slight but reasonable breaks in cohesion described above, cohesion is kept relatively high by HydraController staying only focused on I/O and game flow, which seem like two loosely related tasks but are a logical choice to combine when game flow is so tightly bound to if and when the user provides inputs.

Furthermore, all actions with Deck, Card, Joker and Player objects are all performed *only* using public methods of each, again ensuring low coupling. As for cohesion, the Card & Joker classes are only concerned with the maintenance of their suits and ranks, the Deck class is only concerned with typical deck actions in real life such as shuffling, adding and drawing cards, and the Player class is only concerned with the (massive laundry list of) actions a player can do in a game of Hydra, such as drawing and adding cards to the draw and discard piles, swapping with the reserve, playing the card in hand, and specializations of these actions.

Design Decisions

Some important design decisions I made in my implementation:

- You cannot use a joker to "forcibly" cut off a head – if you assign a Joker a rank that is greater than the rank of the top card of the first head, then proceed to try to cut off a head by placing the Joker on the first head, the game will not allow you to do so and will ask for your move again.
- For easier testing, within testing mode itself, entering the number "`1000000`" as your move causes the game to immediately end with the current player winning.
- All my memory needs were handled using unique or shared pointers *only*.

Resilience to Change

My code offers significant resilience to change, as follows.

Firstly, on a superficial level, if any of the *actions* that occur in the game were to change, my classes are prepared for that – my Deck and Player classes come prepared with a massive amount of methods (far, far beyond just getters and setters) allowing all sorts of actions (that could be used inside the HydraModel methods that actually *define* the actions that occur in the game, described in the next paragraph), with attention to detail to the level, where, for example, different methods for drawing cards from the draw or discard piles exist that allow you to choose whether you want the card to be drawn into the Player's hand, or if you want the card to be removed from the Player object entirely and given to the calling function so that it may be placed somewhere else in the game. I will not lie: I am proud of this extensive, possibly excessive functionality.

If the interface or game rules were changed, the impact on the code would be minimal. All the logic that decides whether a given move is valid or not (which makes up the bulk of the rules of the game) can be found in a single, heavily-refined function in HydraModel, *isValidMove*. This is because isValidMove simply returns a code that tells the caller whether the move was invalid, valid, valid but ends the player's turn immediately, etc., which the program then uses in the method *performMove()*. If we wanted to use these codes in a different manner, or change how some moves occur entirely, *performMove()* would be the only method we would need to change. Same goes for the process of cutting off heads, which is all defined in and restricted to the HydraModel method *cutHead*. Evidently, any significant changes to the rules would mainly only change code within these two functions. Any changes to the interface, similarly, would also be easily taken care of: printing the lists of Heads and Players is handled by *printHeads* and *printPlayers* in HydraController, respectively, allowing easy changes to the information outputted to players, as well as its formatting, if we so desire. Similarly, the interactive prompts for players are handled in a single location within the *runGame()* function in HydraController, so those can easily be tweaked as well. Furthermore, in all locations where output is generated, I follow an invariant: I assume there

More obscure rule changes would also be easily handled. All initialization of the game, before moves begin, is handled by the *init* method in HydraModel – any different starting state we desire for the game would require only changing this method. Similarly, in HydraModel, the *testingCardEntry* and *takeJokerValue* methods take care of receiving and processing inputs as to the player's choice of card suit & rank during testing mode, and restrictions on the joker's value immediately before a move, respectively. Any change to how we want these processes to occur, such as which characters we accept, or any changes to the prompt beginning these processes, would also only require changing these methods.

Other rule changes would also be easily accommodated thanks to the level of abstraction afforded by the different classes that are used in HydraModel to make up the game. For example, adding a third pile for players to use within the game would be easy as adding a new Deck instance to the Player class. The player class itself is also a good idea – separating Players into their own class allows easy extension of this class into subclasses, such as ComputerPlayers, which would probably have their own fields that determine their strategy and play style (using the Strategy design pattern). In the same fashion, adding multiple ways to shuffle a Deck, for example, would also be as simple as adding a new method to the Deck class. Even adding more, arbitrary card suits and ranks would simply involve adding new values to the respective enumeration that defines them and then updating the *getName* function in card to support printing a new name for these types as well. Nothing about how they are handled within Deck, Player or the HydraModel classes would need to change.

An important part to mention is that there would also be minimal recompilation if we were to change a method because of any rule change, as all the classes described above are in different files (obviously) – thus, even adding a new kind of Card, for example, would not even require recompilation of the Deck class.

Answers to Questions

Question: 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.

Note: the most important changes I made to my design since DD1 can be found in the Design section → Outline and Changes from DD1. Please also refer the Resilience to Change section how a change to the interface or game rules would have the most minimal impact on the code.

One key improvement that I made to my design related to this question is how I handle the logic behind whether a move is valid – before, I planned to have this occur in a *move* function within the Controller class, which admittedly would have been an assault on cohesion. Now, this logic can be found (and is called by the Controller) in the suitably-named *isValidMove* function within the Model class, which is a much more appropriate place for it to be located.

Furthermore, my class structure for HydraController and HydraModel still counts as a heavily modified version of MVC (which was the design pattern I planned to use in my DD1 plan), with the View being absorbed by the Controller due to it becoming redundant thanks to **easy-access methods in my classes as well as the simple nature of the desired output allowing easy implementation** – it was very easy for me to be able to implement the outputs to the player because I implemented methods in the Deck class that allow easy access to its top card, its size, the head number stored by the Deck, etc. as well as in the Player class that allow me to print the card in hand's name, the size of the discard & draw piles, etc. **It thus seemed excessive and redundant to me to implement a HydraView class that would only have two methods, *printPlayers* and *printHeads,* which themselves would not have any real logic to them at all and would literally only call the appropriate methods of the Player and Deck objects contained within the Model to get the necessary strings to output, and then dumbly print them to `cout`.**

Question: 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?

Joker inherits from Card, so that the main features of a card that are commonly used stay the same (i.e., suit, rank, associated getters). We still need to special-case them when we are adding a card to a Head to ensure we don't simply add a Joker to a Head without asking the player to restrict its value to something first, and the Joker specialization (of a Card) contains a *setSuit()* method to accommodate this. Otherwise, all methods offered by Card are also offered by Joker, allowing seamless interposition of the two types.

Question: 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?

The ComputerPlayer would be a subclass of the Player class (meant for human players), so that it would easily receive all the fields/operations associated with what a Player (i.e., a human player) has and can do.

However, there would also be extra fields/operations associated with a ComputerPlayer, including, but not limited to, picking a strategy algorithm using the Strategy design pattern below, as well as an operation to switch strategies based on certain information available to both human and computer players (such as the cards they already played, whether a reserve is present, etc.).

Question: 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?

Adding to the answer in the question above, within the ComputerPlayer class, I would use the Strategy Design Pattern in the following manner:

We would have a field in the ComputerPlayer class, call it *strat* for example, pointing to a new PlayStrategy class, added for this purpose. PlayStrategy would have its own specializations (i.e., classes that inherit from it, such as Strategy1, Strategy2, etc.) that would each contain implementation details of its particular strategy (such as whether to count cards, how to play based on the player's number of remaining cards and whether a reserve is present, etc.), and of course the ComputerPlayer would be able to dynamically switch between these using an appropriate method.

The best part about this approach is that this would not require changes to any of the other classes in the game nor its structure: all we would technically be doing is adding possible objects that the single *strat* would be pointing to and using to decide the Computer player's moves.

Extra Credit Features

The only extra credit feature I completed was the RAII challenge – the entire project is leak-free, and the memory is managed exclusively using smart pointers. Indeed, the only raw pointers I use in the program do <u>not</u> express ownership and are only used for calling methods and accessing a particular object of a class (such as a particular card in a deck, but even to do that, I chiefly use references). No *delete* statements are present in my program as well.

As to why implementing this was challenging, in fact, I must admit it wasn't – using smart pointers, as well as STL vectors and deques in the implementation of this project was a *blessing*. Being able to spend my time focusing on the internal logic and edge cases of gameplay, rather than intricately managing memory and debugging leaks, felt very liberating. The only experience I had in completing the RAII challenge that was remotely close to a struggle were remembering to use either a raw pointer or a reference to a unique pointer object when trying to mutate and call the methods of my Deck and Player objects that were stored using unique pointers inside of my Hydra Model class.

Final Questions

1. 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?

Lessons I learned:

1. Always test your large programs' new features incrementally, rather than implementing a batch (or worse, all!) features at once and then testing and debugging that entire mess. Thanks to incremental testing of each class as I developed each of its features, I was able to be far more certain of the correctness of each class and its methods, which gave me peace of mind and

allowed me to develop the remaining parts of my program more effectively and quickly, without spending time stressing over details I had already implemented.

   a. On a side note, this allowed my project to work perfectly (barring testing mode) during the first test after I completed it! It was a wonderful feeling.

2. Similar to the first, I learned to always implement the most basic classes of my program first (i.e., Card first, then Joker, then Deck, then Player, etc.), for obvious reasons.
3. I learned that in large programs, it is even more crucial to ensure all your methods work correctly, as silly, absent-minded mistakes are much, much harder to find. Before implementing all algorithms that weren't simple getters or setters, I spent time making notes and planning out the details. This was alongside the copious amounts of time I spent reviewing and checking my code for mindless errors – which fortunately, I found a lot of often before I ran single test. Note that I didn't write pseudocode – rather, I pedantically made lists of all the different details my methods would have to cover and the order they would need to occur in the algorithm to ensure correctness. Proper planning like this is often underrated and I feel as if such planning was crucial to me setting my mind at ease when I was developing my methods and made my program far easier to debug (as well as with incremental testing allowing you to pinpoint far more easily where an error could be).
4. Even when on your own, proper documentation and commenting is especially helpful when developing a large project. Given that I spent almost 2 weeks developing the project, it was apparent I would end up forgetting at least some of the code I had written and what exactly it accomplished. Extensively documenting my code with comments solved this in the obvious way that reading the comments would tell me what a code block was for, but more subtly, simply the act of sitting and writing the comment for each code block allowed me to be more familiar with my code and remember it more easily than if I never took the time to review and document it at all.

2. What would you have done differently if you had the chance to start over?

If I had the chance to start over, I would begin with setting my goals more aggressively so that I could have more time left over to pursue more extra credit features, such as having a few days to try implementing a GUI. I would also implement the mountain of getters/setters that would allow me to fully rid myself of the slight loss of cohesion of my HydraModel and HydraController classes, as detailed in the Design section → Outline and Changes from DD1 → Bullet Point 3. It also would have been very nice to add an option to set a specific seed using a command-line argument, allowing my results to be reproducible.