🐉 Project Hydra 🐉

Bonnie Chin | b22chin | Aug 13th, 2021

Introduction

Say hello to Project Hydra. Project Hydra is Solitaire's big comeback after losing popularity compared to inferior games such as League of Legends and Rocket League 🙄. This is my outline of a proposal for a MVP (minimum viable product) of Solitaire's big comeback. It's time to show these CS kids what gaming really looks like 💪.

Overview

The overall structure of my project is centred around the MVC (Model-View-Controller) design pattern. One change from my original design is that now main.cc, rather than the controller, is responsible for interpreting executable arguments to decide what views will be added as observers to our Game (model) initially, similar to the card example provided.

View(s)

My project is composed of two views, a testing view along with a normal command line view. They're both derived from an abstract observer class. It is solely responsible for collecting information and displaying information, demonstrating high cohesion. Based on the state of game, views decide what needs to be obtained from Game, what should be outputted, along with collecting the necessary raw input.

Controller

The controller is responsible for interpreting the input from the view in a way that is useful for Game's methods to use, and calling methods from Game to carry out the intended actions. It is also responsible for basic input verification, such as ensuring the move number is 0, greater than or equal to the first head or less than or equal to the last head, otherwise an error will be thrown that is to be caught by view — indicating that an appropriate input needs to be collected. It also does so when the move number is deemed invalid by Game. It is solely responsible for interpreting input into actions and verifying its validity, again demonstrating high cohesion.

Game (Model)

Game handles a variety of different tasks through various classes and design patterns that I will outline in the design section. Since we're following the MVC design pattern, it is derived from a subject base class with a series of observers (views). At a very high level, it is responsible for storing the game data, manipulating the data and states related to the game, along with notifying the necessary views of updates to this data when appropriate. It handles all of the logic of the game, applying certain moves, asking for input from view when necessary.

More specifically, it handles actions such as…

- Shuffling and distributing cards to every player
- Initialization, such as creating the first head
- Moving to the next turn and/or player
- Applying the possible action desired by the player, and indicating whenever no moves are applicable
- Determining whether someone won or not

…and stores information such as…

- The players' decks in the game
- The rules applicable to the game
- The heads and the cards that live under them
- The stage of the game, whether it is cutting a head, still initializing or requires a player's card to be set (testing mode)

Game has stores (and thus has relationships) with classes such as Player, Head, and Rules. These classes were created to easily modularize the information that can be passed to the views for them to output. For example, having a class for each player's deck of cards makes it easy for view to ask for the list of players and to output the corresponding information about their decks.

Design

Updating the views (**MVC** Subject & Observer)

Similar to what was used in A4Q3, each view has a subscription type. This is to allow for the simple expansion of the number of and the types of views that may be added to the game. For example, the test view has the subscription type SubscriptionType::Test. Thus, similar to A4Q3, only in circumstances where the test view would need to be prompted would we call notify(), and pass in the test views we want to notify.

Given that many views (such as both testing and the command line view) might all need input (drawing a card) or output a certain piece of information at the same time (displaying the heads), a method called notifyAll was also included.

Applying Rules (**Strategy** Design Pattern)

The game stores all of the concrete rule subclasses of Rules that it will abide by for the game, where rules are possible actions a user can take. Specifically, I leveraged the strategy design pattern. This decision was so that the game could easily call the same methods on each rule to **apply** that rule. This would prevent having to create a long if-else chain.

A notable change that was made to the design was **making the "cut head" action a derived class of Rules**. This is because I noticed that cutting a head shares a lot of similarities, we need to check if it is possible, and if so, apply it. It is only if all of these actions are not applicable that Game can easily throw an error indicating that the desired move is invalid.

We only want to apply a rule if it is both possible and if it is necessary — we see that when we cut a head, we want to check if other rules are possible without actually applying them. In the interest of enabling future rules that wish to have similar conditions, I modified my UML diagram to define **both a check and apply method** as part of the rule base class that derived classes can override. This means changes to a certain rule can simply be made inside of that concrete class without impacting the program in any other locations.

Another change was modifying the parameters of the Rules base class to pass the current player, heads and the headNumber (the head they want to place the card on or 0). Before I had simply anticipated checking, but I realized it would be better to tie the logic for applying a rule to the Rule itself rather than in Game for better cohesion.

Cards (Leveraging polymorphism by overriding virtual functions)

Each card shares similar features. They all have suits, values, and when combined will have names. These similar features make it highly appropriate to create a class to represent all possible cards. As well, it enables us to encode information such as numeric values assigned to each card to define useful methods such as equal and lessThan, which can stand-in as operators that compare two cards. It also forces those who want to make changes to the card to only make changes that uphold class invariants, otherwise an error is thrown.

Joker Card 🤡
There remains the peculiar behaviour of Jokers, in which their value and corresponding numeric value is subject to change. The joker's value and name depends heavily on whether its value has been specified yet or not. Thus, rather than constantly special-casing the joker to decide whether to return "Joker" or the assigned value, overriding the accessor methods to take care of this is much more effective!

Players

Similar to the approach for cards, since each player has the same decks: discard, reserve, and draw. They each have their own corresponding number and can have one card in hand. These similarities, paired with the level of repetition present (multiple players will need to be created) makes it appropriate to create a Player class to store all of this information and modularize the implementations of e.g. the conditions related to drawing a card, getting it from the drawpile or shuffling when using the discard.

Heads (Implementing a stack)

Given the nature of heads, such as how only really the top card is visible, and only one card is placed on it at a time, a "stack" data structure was quite appropriate. Thus, heads are equipped with basic stack functions such as peek, pop and push to manipulate the stack of cards it stores.

Game - elaborated

Beyond what was outlined in the overview, Game generally provides accessors to only what view needs to print out such as the player, heads, and state of the game. The providing accessors to

the state of the game, e.g. needCardValue, setUpComplete, ruleApplied enable the view to know whether to ask for a card value, to display the head that Player1 places initially, or proceed with asking for the next move. Furthermore, because actions such as drawing a card are a very common action that requires prompting the test view (and a few other actions), repeated actions like such have their own methods to emphasize **separation of concerns** and easy debugging.

Error Handling (Custom class exceptions)

Implementing basic, but custom class exceptions prevent having to constantly check for certain rare circumstances. For example, if a player has no cards to draw but their reserve, it would be tedious to have the drawCard method in Player return a boolean, for the private drawCard method in *Game* to also return a boolean, only for the move method to check and interpret that boolean each time. It is both more efficient and more clear to **simply catch that as a OnlyReserveLeft exception and carry out the necessary action** (moving on to the next player) to resolve it.

Resilience to Change

Using the Strategy Design Pattern for Adding and Modifying Rules

Creating an abstract base class for rules makes it easy to add additional functionality for each rule as necessary. But more importantly, doing so enables us to **add and delete as many rules** as we'd like without consequence. If desired, it makes it easy to create various game classes that can adopt different **"packages" of rules.**

Since each concrete rule class lives independently, it is **easy to modify** the behaviour of each rule and the conditions to apply a certain rule without impacting the other rules or how the result of the program operates.

MVC for enabling many different types of views and players

As outlined by the project description, one major potential expansion would be allowing computer players to play or even offering a GUI. These would require having to display and collect information in different ways to more channels. This is handled easily by the MVC by attaching more views to the list of Game's observers.

Using subscription types also ensures that only necessary views are prompted.

Additionally, while the controller class is generally unpopulated as it stands, if we add a GUI that enables gesture commands like many online card games, the controller can easily be expanded to have methods that translate these gestures into meaningful actions for Game.

Card class enables different types of cards with different rules

Obviously, having a card class makes it easier to modify universal methods such as how the cards are evaluated and compared. It means being able to modify the value and therefore the "weight" of different cards.

More importantly, having a separate card class means it is possible to add more special cards similar to the joker with unique functionalities.

Head class enables creation of special types of heads

Since we created a class for head, this can easily become a base class should we want to facilitate the creation of different types of heads that have their own sets of rules (we would likely create another set of HeadRules for this). Again, it would also allow easy universal changes in how cards are processed when placed on the head or popped too.

Class Exceptions

Using custom class exceptions is useful for the reasons outlined earlier, along with the additional functionality and conditions that are possible when creating custom class exceptions.

Player class allows different repositories to be possible

Right now there are mainly 4 “piles”, what’s in hand, the reserve, the discard, and the draw pile. However, having a player class means that we can easily modify the various piles that exist. Furthermore, if we want to create different types of players that will e.g. draw cards with different conditions, we can derive subclasses from this base class.

Answers to Questions

1. **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

   As outlined earlier, rules are easy to modify through the implementation of the strategy design pattern for the rules. The abstract Rules class allows us to derive as many other rules as we’d like. The only other code that would have to be modified is the Game constructor to add certain sets of rules it would like the game to have. My classes fit this framework as I have implemented an abstract Rules class from which each action, e.g. rule1, rule2, cut are derived and Game contains a vector of all these rules.

2. **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?**

   The Card structure should be created as a base class with virtual methods for certain methods such as setValue and getName that the Joker would have to override as a subclass of Card. This is because the name and value that Joker takes on will change throughout the game, and thus requires its own defined method to handle whether to produce Joker or its assigned value. Thus, when we have a deck of cards and would like to

get the name of a card, it isn't necessary to check whether the card is a joker or not to determine what to output.

3. **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.

   To allow computer players, the MVC is extremely handy because of how it is built off of the Subject-Observer pattern. Being able to attach and detach views enables players (human or computer) to enter and leave as needed.

   Furthermore, as explored in the resilience to change section, this allows us to provide a custom view (or in other words, an interface) for the computer player. This is necessary as a computer player would presumably want to be passed information about the state of the game directly as variables rather than read through a command line. Every time the state of the game changes (requires input, turn changes etc), each of the players' views (human and computer) can be updated with the information relevant to them. Along the same lines, this allows us to provide a custom controller for collecting inputs from humans through the command line/GUI versus a computer player.

4. 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 information associated with the human player to the computer player.**

   To enable smooth handoff, all information should be stored in modules or groups of objects to prevent a high number of dependencies or coupling — otherwise, errors could easily take place when information begins to be manipulated by another view. This makes it easy for Game to simply provide access to any appropriate "module" or set of data that a view would like to view.

   Whether it's the heads, what the other player has in hand or in their reserve, since this information is meant to be public, it will all be accessible for the views to interpret and display or pass to the computer player as needed. A modification that could be made to the current design is creating different constructors defined for the ComputerPlayerView depending on whether there is information to be handed off (and "translated") from a human player or not.

Extra Credit Features

Shared pointers

Generally I found shared pointers easier to work with or similarly difficult to regular pointers. It was convenient to not have to worry about allocating and freeing memory and ensuring that no memory leaks would take place.

I did however, have to ensure that there were no pointer cycles to prevent memory leaks. To do so, I ensured no pointer was stored in one place at a time, keeping track of when to set shared pointers to null. I kept a running log of where shared pointers were created and used, ensuring that they would either be deleted by the stack by whoever was using it, or set as a nullptr when being handed off to someone else.

Final Questions

1. **What did you learn about writing large programs?**

   One major lesson that I learnt is that **planning and breaking down the project is extremely important**. It's easy to get lost in the sea of everything that you have to do and jump from part to part. I made a kanban board for myself that broke down the project into dozens of tickets. While you're coding you may think of new ideas, corner cases or ways to optimize your program that are unrelated to the component you're working on. Having a place to jot down all concerns to address later is very helpful to keep track of everything.

   I also learnt that big projects and tight timelines mean sometimes compromises need to be made and **priorities need to be extremely clear** to be as efficient as possible. I would think of new issues or optimizations, but some were less important than others. Adding priorities (P0, P1, P2 etc) helped me gain focus knowing exactly what order in which I should work on different parts of the project.

   Last but not least, it's super important to **build and test little by little.** Performing unit testing (testing certain methods and certain classes) and then integration tests on methods that leverage multiple classes. This makes it much *much* easier to spot bugs as the scope of possibility is narrowed dramatically.

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

   If I had the chance to start over, I'd definitely try to get a rough draft of my UML diagram done earlier. More importantly though, I would spend more time **outlining key functionalities and edge cases** as well — I didn't consider soon enough about how asking for the value of the card would be facilitated and this resulted in having to make Game's drawCard method public for the cut rule to use.

   I would also create **sequence diagrams** to have a better understanding of the degree of coupling and cohesion my UML diagram might create and make modifications as necessary earlier on in the process rather than upon having already declared and implemented certain interfaces. Along the same lines, I would triple check and create a comprehensive UML

diagram (private and public fields and methods) and base classes before creating derived concrete classes.

Finally, I would also **look into more C++ libraries** — there may have been functionalities I could have leveraged but didn't think of or have a chance to explore for this project.

I would also make a few technical changes if I had a chance to start over.

1. **Create more mutator methods in game and controller** for setting card inHand and setting the card of a head when cutting heads rather than modifying in View to increase encapsulation and reduce coupling.
2. Create a **base class for testing view and the command line view** since they are extremely similar.
3. Create a **modified subject class** that takes in a pointer to a list of observers. Each class that Game is responsible for such as Player, Head and Rule could then be derived from this modified subject class such that **they are able to call notify** at more points in the game while keeping various methods and **actions decoupled from Game.**

Conclusion

Thanks so much for reading my proposal! 👋 I really believe this will be the next big thing in gaming. If you invest in my idea, I'll offer **30% equity** in the multibillion dollar business this will form, along with a **0.01% royalty** on all in-game purchases (got to let them create CS246 themed custom decks am I right?) I really believe in this project, and together we can show Jeff Bezos what a real billionaire looks like.