CS246 Final Design - Hydra

Name: Rachel Li
Student ID: 20828656

Table of Content

Overview:

The Hydra game was implemented using the Observer design pattern. The two subjects are Player and Head, and each of them has an observer attached to them, which is PlayerDisplay and HeadDisplay respectively. A Game object is instantiated at game start with the player number entered. The Game object is the controller of the entire game, because it “owns”, a.k.a., creates all the Players, Heads, a PlayerDisplay and a HeadDisplay. Within each instance of a game there is only one PlayerDisplay and one HeadDisplay, each of which will be associated with each of the Players and Heads respectively. Note that PlayerDisplay is the observer for Player only, while HeadDisplay is the observer for Head only. Each Player has three Piles of Cards; draw, discard and reserve, which are encapsulated. At Game initialization, each Player’s draw pile is loaded with 54 cards from a shuffle Pile of cards generated from n decks of cards(n is the number of players). There will be one Head in the Game at the start of the game, and more Heads will be added and cut as the game proceeds. Head is a Subject as well as a Pile, where each Pile has a pile of Cards in it, and Card has methods that allows comparison between two cards based on their values, as well as methods to get and set their values for display and testing. The main function will be mainly responsible for enabling testing mode by taking a command line argument and getting inputs from stdin to at each Player’s turns, which will be displayed as a simple text UI.

Inside of a turn, the program will do the following: main will take inputs from stdin, and pass them as arguments to the main controller of the Game object, tick(). tick() returns different

integers based on what the results are from that one “tik”. And the number that the Player enters after the “Your move?” prompt will be passed into tick(), where it will make decisions on whether to cut a head, or to draw a card, or to use/switch the reserve, or nothing is that number passed in is not valid. For example, if user input was not valid for the turn, e.g., the input head number is two big or too small, then it will return 0 to let main know that it needs to repeat the prompt, until tick() tells it to stop by returning 1, meaning that everything is good. Tick() will decide whether the current Player has to cut a head or not. If they have no heads to place their top card on, then tick() will cut a head off for the Player regardless of the head number they enter, and generate two new heads using the top cards from their draw pile. If they have a head that is valid to place their top card on, then Tick() will call the Player’s drawCard() method, which will invoke the draw Pile’s PlayCard() function with the Head passed in. The PlayCard() function moves the top card of the Pile to the other Pile. And if Tick() finds that that the number passed in is 0, then it will call the Player’s useReserve() method to switch or use the reserve, which places the top card in the draw pile to the reserve pile, or to switch them.

Pile owns Card. Player and Head both own Pile(s). PlayerDisplay is aggregated to each Player. HeadDisplay is aggregated to each Head. Game owns all Players, all Heads, a PlayerDisplay, and a HeadDisplay. Game is responsible for notifying all PlayerDisplay and HeadDisplay, and it only updates them whenever they need to be output. This is the solution to a problem that was neglected at the initial design. See more details on this in the next section. The main challenges encountered were mostly on how to seamlessly keep the internal attributes of each Player and Head consistent with their Display while applying all the Hydra game rules.

Design:

RAII

The design follows by using smart pointers for all composition associations that involve the use of vectors. And all of the smart pointers used were unique pointers to an object on the stack. They are moved instead of copied every time a change of ownership happens. This greatly reduces memory leak and segmentation faults, although they can still happen since functions that return raw pointers are used.
Some examples of RAII in the design will be Pile and Game. Pile has a vector of unique pointers to Cards, where are allocated on the heap when initiating the unique pointers. In this way, when the Pile goes out of scope, all the unique_ptrs go out of scope, where their destructors will be called, which will free the memories of all the Cards on the heap. Same thing happens with Game. The unique_pts for Players, Heads, PlayerDisplay and HeadDisplay will all go out of scope at the when main exits, where their destructors will be invoked, and their memories will be freed.
I found that using smart pointers is very convenient in that the destructor does not need to do anything to free memories. Stack unwinding does the work since each unique_ptr will delete the instance of Card allocated on the heap.

WorkHorse Function

Hydra involves a lot of changes in the ownership of Cards, so I made a workhorse function in Pile called moveTopCard(). It first moves the top Card of the Pile out of it, and pops off the space in the encapsulated vector for all the cards in the pile, so that the size is reduced by 1, as it should be. Then it calls the other Pile's emplace_back() function by moving the Card that was just moved out as an argument. That way no copying is needed and efficiency is maximized since I am simply moving chunks of memories around.
This function provides the backbone for other functions used during game such as Pile::PlayCard(), Pile::loadCards and Player::useReserve(), which are wrapper functions of moveTopCard() with other features that adhere to different game moves or rules. For example, PlayCard() compares the value of the top cards, while loadCards() is used for loading an arbitrary number of cards from one pile to another, which can be used when distributing cards to each Player's draw pile at game start.

Observer Pattern and Some Challenges

The initial plan was to separate the observers for Player and Head, while still being able to encapsulate PlayerDisplay and HeadDisplay from the Game, so that Player and Head and notify their observers upon changes without having a "third-party" object to do it. However, I didn't think that I actually needed to return different "status" or "info" structs to the observer in the notify() function for Head and Player. For Player, I need to tell the PlayerDisplay about the number of cards on each of its draw, reserve and discard Pile, as well as the remaining card that it needs to draw. For Head, I need to tell HeadDisplay about the head ID, how many Cards are in the head as well as the display value of the top card of the Head. They are very different objects and their display need very different things. The biggest problem I encounter during implementation is that because I am return PlayerStatus for Player::getStatus() and HeadStatus for Head::getStatus, I cannot figure out a way to make a generalized virtual Subject::getStatus() method in the Subject class for each of them to overwrite it. They return different types. If I have both PlayeStatus getStatus() and HeadStatus getStatus() in Subject, then I will have to make them purely virtual since Subject doesn't know whether it is a Head or Player, but then I will have to overwrite both functions in my Player class and Head class, which will not work because I don't need a PlayerStatus getStatus() function in my Head class, but if it is purely virtual in the parent class then I have to overwrite it since Head is a concrete class. Therefore, the solution was to not have inheritance on the getStatus() methods, and let Player and Head have their own getStatus, which is not in Subject(). That leads to another inconvenient change, which is that I have to take out notify() in Observer because it takes a pointer of Subject, and Subject doesn't know about Player::getStatus and Head::getStatus. I can overwrite it in Player and Head with downcasting(downcasting the Subject pointer to a Player/Head pointer), but that will be very bad practice. As a result, I had to take out notifyObserver() in Subject as well because it calls each observer's notify() function.

My solution to this problem was to have Game called notify() for each of the Players and Heads, since it will be responsible for making any changes to the internal structures of Players and Heads as well with Game::tik(). Game owns HeadDisplay and PlayerDisplay as well as all the Players and Heads, so it can call each of their notify() and getStatus(). I have a Game::notifyDisplay() function that updates both PlayerDisplay() and HeadDisplay() by

having each of the Players and Heads notify their observer, which is PlayerDisplay if it is a Player and HeadDisplay if it is a Head. It replaces Subject::notifyObserver() that was taken out. I thought about having one big giant GameDisplay class responsible for all displays, but it was too messy to update so I stick to the separate observers design.

Resilience to Change:

Display

I can easily have multiple displays for the Players or Heads by having a different child to Observer. For example, I can have another observer for Player called PlayerDisplayGUI that would output Player differently from PlayerDisplay. Same applies to Head, where I can add a fancier display for the list of heads with a new display call HeadDisplayFancy. And I can output different displays by taking command line arguments or using inputs from stdin to accommodate specification or preferences.

Game Rules:

For drawing cards, cutting heads, using the reserve and checking if a player has won, I have a specific function for each of them in my Player class. Game::tik() is only responsible for calling them. If I need to change any rules, I can just go to the function for that rule and change a couple lines of codes without changing other functions in the same class or other classes. I encapsulate most of the operation during a player's turn within the Player class.
If I now need different information for those new displays from Head and Player, I can change the fields inside of PlayerStatus and HeadStatus to give the observers the information they need in notify().
I implemented the comparison operator for two Cards. If I want to change the way that two cards are compared, e.g., if suits are taken into consideration, I can simply modify my Card::operator<() and Card::operator==() function to accommodate this change.
Although I didn't implement the part that allows for setting a seed when shuffling cards, I can have a field in my Pile class called seed that would be passed in at initialization to set the seed for any Pile shuffling later on. The seed value can be taken as a command line argument, and Game will initialize the Piles with the seed passed in as an argument. And if the game specification needs a uniform random generator with a specific seed, I can also hard code the seed into the Pile class as initialization.

Structural changes:

Since I am separating the display and the internal data fields, I can also make changes to my data fields if need be without touching my display. I probably only need to change the getStatus() function or the PlayerStatus and the HeadStatus class. For example, if I want to have another Pile in Player called, say, chance pile, then I can add another pile and only add a "int chance" field in the PlayerStatus class to tell PlayerDisplay the number of cards in the chance pile and let it display it if need be.

Answers to Questions:

Question1: 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.

Answer: I will use an observer pattern to structure my classes, because I can separate the views and the actual structure of my classes, and I can display my classes in a way that is more customizable. In Hydra, I need to constantly change my Piles info and I have a text UI as the game interface, therefore an observer pattern will be most appropriate. I only update the displays, a.k.a., notify my observers, when I need to. In my implementation, I have Player and Heads as subjects, and Player has PlayerDisplay as its Observer, and Head has HeadDisplay as its Observer. Player and Head have a getStatus method that allows PlayerDisplay or HeadDisplay to obtain the most updated info on the inner attributes to update the display texts in them, while PlayerDisplay and HeadDisplay have a notify() function that allows Player or Head to tell them to update their display text, and it will call the corresponding getStatus() function from the subject that calls it.

Question 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?

Answer: For my Card class, I have a getValue() function. If I am asked to get the value of a Joker, then I can pass in another parameter telling getValue() whether this is used for the first card of a head. If not, then if the suit is a “J”, then I will prompt the user within the getValue() class to obtain a value for the Joker.
Currently my implementation has a Card::setValue() function for testing mode as well as setting the value of Joker, and I will call it if I getValue() gives me a card with a suit of “J”. My answer above will probably be a better option than my actual implementation. However, I will stick to the fact that I set all the default values of Jokers to 2, that way I don’t need to ask for a value when it is used for Heads.

Question 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, i.e., dynamically during the play of the game. How would that affect your structure?

Answer: I can make a Player class that has children including HumanPlayer class and several ComputerPlayer classes. That way I can store either a HumanPlayer or one of the ComputerPlayer as a Player in Game. Each ComputerPlayer can have its own drawCard() function that uses different strategies.

Question 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 the information associated with the human player to the computer player?

Answer: My answer will be similar to my answer in the initial design document. I can make several functions in Player virtual to allow overwriting in children classes, and I will create multiple children of the Player class that inherits all the fields and public functions from it, whether they are encapsulated or not(private fields and functions). Then if a human player stops, then I can instantiate a new ComputerPlayer using the pointer to the human Player in the ComputerPlayer's constructor by calling the copy constructor of the human Player. That way, the new ComputerPlayer will have all the data and functions that the previous human Player has, and it can also have new features and functions added in the Computer Player class such as strategies in drawing cards etc. Since I am using unique_ptrs to Player in my Game class to store the players, it can also point to a Player object using dynamic dispatch.

Final Questions

Question 1: What lessons did you learn about writing large programs?

Answer: What I learned was that the designing stage is just as important as the implementation stage. Never count on "figuring it out" when you start implementing, because sometimes the more developed the program is, the more complicated and difficult it is to make changes on the design, even when it is beneficial to the overall structure and efficiency of the program. For example, I found out that I have to take out Subject::getStatus, Subject::notifyObserver() and Object::notify(Subject *who) when I started to compile my code, which leaves me very little time to figure out and implement a new way that would be better in terms of encapsulation if I make a major change in design. I would have too many other functions and bugs to fix if I do that. So I had to compromise my design with my functionality since I want to make sure that it works with the time I have. I think my structure is still a solid observer pattern, but I think I can do better in terms of updating the display with more encapsulation, had I thought of the getStatus() problem that I had at the design stage or even the early stage of implementation.

Question 2: What would you have done differently if you have the chance to start over?

Answer: I think I can make a Status class with PlayerStatus and HeadStatus as its children. And when I call notify() it will just return a Status object, which can be either a PlayerStatus and HeadStatus. That way I can call notifyObservers() using the Subject::notify() function instead of having it down in Player and Head, which would not recognize Subject. This is better OOP practice.