Plan of Attack (DD2): Sorcery

Filip Milidrag

Due: 2023-04-10

Brief Overview Of UML Design

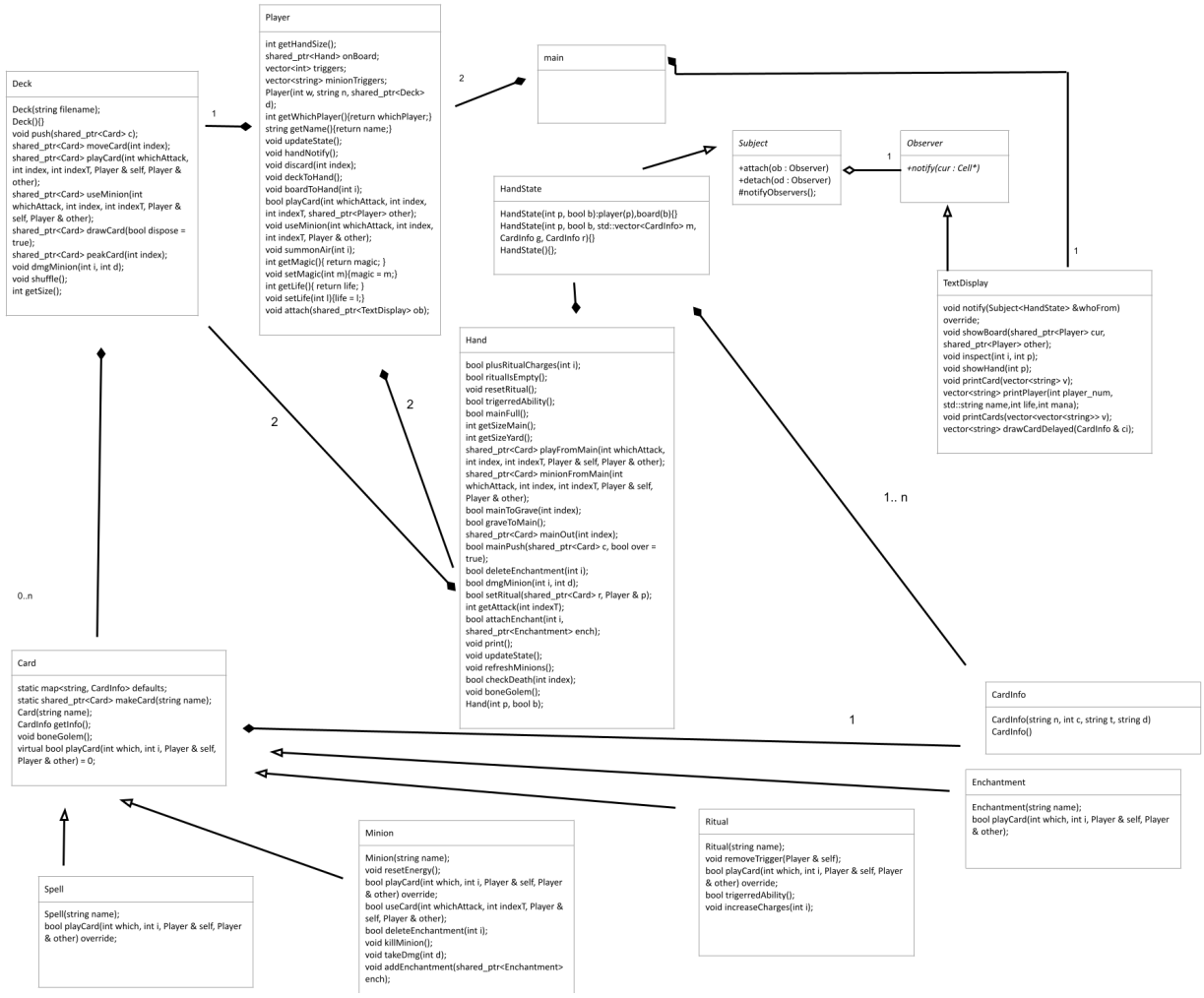
The main function contains the two players which contain all their own decks cards in their hand and their cards on the board. Each hand of cards is a subject in the subject-observer design pattern. The text display is the one and only observer of all the hands. When any hand changes the textdisplay is notified and its fields are changed accordingly. All types of cards are subclasses of Card and Card is contained in decks which are either owned by a player or in a hand which is owned by a player. Each hand has a handstate which is what gets passed to the observer. This hand state is made up of CardInfo which is a summary of some information about a card. So each hand has the cardinfo of its cards in its hand state. Main function loops getting user input and calls the appropriate function through the current player in order to make the appropriate moves.

Design:

To initialize the decks for the players the Card class has a static map that will give the requisite information needed to construct the specific card object in question. Since each Card pointer in Deck is actually a minion,ritual., etc. I used a Factory method that will return the appropriate object pointer for whichever card needs to be made, this uses the aforementioned static map. Note that the factory method is also static as it only requires the use of the static map and is not called as part of some instance of Card. This way we can initialize a deck with this one static function. For the display the subject observer design pattern was used, this separates the logic of the program away from the visuals. The visuals also only need some of the information so this is stored in the CardInfo objects and is used by the textDisplay to display the board at the user's request. In the initial design I planned to use a decorator design pattern for the enchantments. I however, decided not to do this since holding the enchantments in a vector is easier to work with and uses less code. This results in something that is more readable and it does not lose any features of the decorator since when adding a new enchantment it is very easy for it to be added to the vector of enchantment pointers and to implement the overloaded function "enchant" just as the rest of the enchantments. That way we are using polymorphism to run the different enchantments and it works just as well as if we had employed a decorator design pattern. Similarly to this the original design includes subclass of all the types of cards for each specific type of card. I did not do this as the specific cards are in reality very similar and only different in very slight ways that are more easily implemented using a simple if statement in one of the functions rather than making up to 20 extra classes to rewrite functions that do very similar things. I also used smart pointers and vectors in order to avoid any issues with memory so

it will be much easier to add things on as there are no memory problems to worry about. Polymorphism is used throughout in order to avoid repeating code. A good example is using pointers to Card in our decks. All cards have to get played from the players hand so they all have the playCard function and they each overload it to make it slightly different however many of the features are similar so when a player plays a card it will call the player's playCard function which will execute the similar aspects and then it will call the Card playCard function which will run something different based on which derived class is actually present.

UML:



Resilience to change:

The program uses oop inorder to become resilient to change. If we would like to add some new card whether it be a minion, ritual, etc. it suffices to add an extra else if in one or two places since all the minions have similar behavior. For example if a new minion with a new triggered ability were added, no changes need to be made to the minion class. The only change is that at the location where the trigger will happen we need to check through the minionTriggers vector in

the player we were looking for. If we find the name of the minion in it then we can run whatever code is necessary to run for this behavior. Most of the events are isolated which means that changing one thing won't impact the rest. For example the haste enchantment increases attack and defense by 2. If we change this to some other value no part of the code needs to be changed except for 2 values in the static map. The same is true for any other values of cards like a ritual's starting number of charges or how much magic a certain card costs to play. Adding new cards will also affect CardInfo but this won't be a problem since we can easily add a variable to cardInfo and it will not affect any other type of card. Then in the drawing function in TextDisplay we can add one extra if statement to recognize this new field and handle it accordingly. Similarly if we changed what a hand holds handState would not change very much as we would just have to add or remove a CardInfo or maybe a vector of CardInfos to accommodate the change but then the rest of our functions could still run smoothly.

Answers to Project Specific Questions:

How could you design activated abilities in your code to maximize code reuse?

While activated abilities can vary they have certain things in common such as having an activation cost. So all activated abilities will call the same function which will run the similar things and then call other functions depending on which specific ability is being called. To handle especially similar cases such as the summoning of new cards, there is a function that summons new cards and we need only specify how many cards to summon in our activated ability function.

What design pattern would be ideal for implementing enchantments? Why?

I implemented enchantments by having a vector of the enchantments that are attached to the minion. This vector essentially acts as a stack where the last enchantment to be added will be the first removed in case the spell that removes enchantments is played. Then at each stage where an enchantment might be applicable we simply check if it is in the vector and for each time it's in the vector we apply it. This is easy to implement and won't be inefficient since there cannot be very many enchantments on a minion. Additionally there is not much code and extending this with new enchantments is very simple. We also don't have to worry about applying enchantments and unapplying them when we add and remove enchantments since they are never actually applied, just used in the appropriate situations.

Suppose we found a solution to the space limitations of the current user interface and wanted to allow minions to have any number and combination of activated and triggered abilities. What design patterns might help us achieve this while maximizing code reuse?

In this case we would use the decorator design pattern since we could add on any number of abilities and then they would all have to be considered. We could then only call one function in one object and it would go through the linked list of objects to consider all abilities at once and see which should be called. Since there is not an unlimited amount of abilities this design pattern will not be useful and would only make things unnecessarily complicated so it will not be used.

How could you make supporting two (or more) interfaces at once easy while requiring minimal changes to the rest of the code?

This can be done by having one graphics display class that has two separate functions that draw differently. So each time a change is made instead of simply drawing once each draw function is called and draws in its own way.

Extra-credit features

My one extra credit feature was using smart pointers and vectors in order to deal with memory problems. One of the big challenges was needing to read documentation in order to know how to work with these types. However once got over this hump it was largely beneficial as my code became much more memory safe

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?

As someone who worked alone I learned just how much work goes into a large program. A big problem with large programs is organizing everything so that we can work on small parts at a time without affecting other parts of the program. It is not possible to keep everything in our heads after a certain point so we have to be able to focus on something rather than needing to consider the whole program. This requires a lot of thinking in advance and possibly a lot of changing the design as we code since our initial plans might turn out to be flawed in ways that we did not expect. It is better to change things up then to move forward with a flawed design as the longer you work on a project the more of a problem thai design can become and the harder it will become to fix. Especially when trying to find bugs one needs to be able to locate the source which is very difficult if the program can not be seen as isolated parts. Because of all the consideration that goes into large programs they take a lot more time then even doing several small programs since the complexity compounds over time. Especially when working alone, it's important to make sure that an adequate amount of time is assigned to each task in order to get everything done without time pressure. If time becomes an issue the design will suffer and in

turn programming will become even slower and the process falls apart, so staying on top of everything is a very large part of this.

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

If I were to do this project again I would start by getting a team. I think that a team can greatly help as you get different perspectives on how to design the program so you're more likely to find a more optimal design which can make the whole process easier. Additionally, having more people working with you will simply save a lot of time. Another thing is that I would spend more time planning my design as I had to make a lot of changes to my design when coding which was not ideal and I still ended up with a sub-optimal design. The main part of my design that I would change is that Hand contains two Deck. I was initially debating between making Hand a subclass of Deck or making it contain Deck but after making the program I realized that the two are too different to be combined like this. Instead if I were to redo the project I would separate them completely, this would result in Deck needing a lot less functions. A lot of Deck functions were not necessary except for Hand functions to use them and Hand had a lot of functions that simply accessed Deck functions because the actual cards in the hand were privated away from the hand. I would need a lot less functions to implement the same stuff and Deck would have a more straightforward purpose in the whole program.