Overview

Note that large changes in concepts/implementation will be noted in the Design *section.*

Most of the basic implementation of our Sorcery is similar to our plan for DD1. As with most programs, the `main` class handles all command-line arguments for the game and receives the inputs, which are immediately passed on to the `Sorcery` class. `Sorcery` has functions for every single command in the game (and variants that target and do not target for `play` and `use`)and is the interface for all the interaction between the two players and their data, which is especially important for attacking, targeting the other player's minions with effects, and similar actions. The game's main loop of turns between the two players is also managed by `Sorcery`, as well as all the actions that must happen when one player's turn ends and the next begins. `Sorcery` also performs error checking for invalid/incorrect inputs for commands fed from `main`.

Like planned, all the data for both players (all cards in all locations and data values) are managed by two `Player` classes. Each `Player` class can have/has cards in four different locations: deck, hand, board, and graveyard. As planned, each `Player` has one each of `Deck`, `Hand`, `Board`, and `Graveyard` (which we will refer to as the card holder classes) and keeps track of their own data for mana and health. All of the commands passed into `Sorcery` that affect only the active player, such as draw, discard, summoning Minions, and activating rituals have their own function in `Player`. For commands that interact with both players, `Player` returns the necessary values for `Sorcery` to deal with. The `Player`s are held in two `unique_ptr`s by Sorcery.

The four card holder classes each hold their own cards, stored as `vector`s of `unique_ptr`s of the either cards or the appropriate card type as opposed to simply vectors of `Card` objects for all of them as we had originally intended. `Board` has a vector of unique pointers to `Minion`s as well as a separate unique pointer pointing to its `Ritual`, while `Graveyard` has only a vector of unique pointers to `Minion`s. Having these pointers explicitly be Minions instead of simply Cards allows us to easily call functions for Minions on them, and ensure that we do not accidentally place incompatible card types on the board, such as `Enchantment` or `Spell`.

The implementation of individual cards is very similar to what we had originally planned. There is an abstract class `Card` which serves as an interface/parent class for the other four card classes, `Minion`, `Enchantment`, `Spell`, and `Ritual`. All cards are created when the player's `Deck` object is first created with a vector of strings for names, matching a name to a card type, and then using that card type's constructor and `map` objects to retrieve the correct data for the card. Each card class holds its own data as well as the methods unique to that card type. For example, enchantments modify a minion's base stats, rituals have charges, etc. Additionally, there are two enumerated classes, `CardType` and `Trigger`. `CardType` identifies the card's type even while it is stored in a container meant to store cards, which is necessary to determine how a card should be played (do we attach it to a minion, put it on the board, etc.) without seeing the subclass type. `Trigger` identifies whether or not a card has a triggered ability, and if it does, what type it is (None, Destroy, Summon, Start, End) to be used for the implementation of abilities.

Originally, we were going to implement enchantments on minions using the decorator design pattern, however, we ended up using a chain of responsibility design pattern by giving each `Minion` a vector of unique pointers to `Enchantment` objects. When the value of a minion (attack, defence, ability cost) needs to be modified, the value is passed through their respective functions for each enchantment, from oldest to newest, to end up with

the modified value. This allows us to retrieve values easily, maintain the base value of a minion, and remove enchantments easily without modifying the base stats of the minion.

Finally, card abilities were implemented directly into the `Sorcery` class. We had originally intended for there to be a separate class designed to handle all cards' abilities, however, we realized that adding such a class would be incredibly difficult, and we could not figure out the proper logistics to implement it. Therefore, `Sorcery` itself handles all the abilities, and calls various functions of the `Player`s to determine which effect to activate, what card to target, whether or not the move is valid, and executing the changes.

`triggerDestruction` triggers destruction effects of a `Minion` if it (and only it, does not support effects that activate when another minion is destroyed) has a destruction trigger, and `destroyMinion`s destroys the `Minion`s that need to be destroyed. Note that "destroy"" in this context means send to the graveyard, and that minions are not destroyed until after the entire effect chain has resolved. The minion whose destruction trigger effect is activated is passed in as a pointer to retrieve values, as well as the minion's owner to determine which player to target with the abilities. `triggerAbilities` checks for all the other abilities and is called for the end and start of the turn, or when a minion is summoned. The entire board is checked in APNAP order and trigger types are compared using their `Trigger`. `activateAbility` takes a card name and the card's owner, and matches the appropriate effect. Note that for summon triggers, the minion summoned is always the last minion on the player's board, so there is no need to pass a pointer to it.

Activated effects are coded directly into the `use` and `play` methods (and their Target variants), matching the card's name to the appropriate effects. Again, `Sorcery` calls the appropriate methods for both players.

Design

For the implementation of the various types of cards, we decided to use unique pointers instead as they provided a much simpler way to manage the program's memory, and to ensure that each card could only physically exist in any one place at any given time. This also ensures that cards are not inadvertently copied when moving from one card holder object to another (e.g. summoning, destroying, activating rituals, drawing, etc.). This was important as `Card` objects, and especially `Minion`s, are constantly being moved between a player's various card holder classes, and we wanted the ownership of the objects to be enforced. This also allowed us to easily compare pointers since each `unique_ptr` can have one pointer to it.

However, this also resulted in one problem: certain card holders hold `Card` objects, while others hold `Minion` and/or `Ritual` objects. A `Card` object cannot be moved into a vector of `Minion` objects, or to a unique pointer for a `Ritual` object. Because unique pointers cannot be copied, there is no inherent way to perform a dynamic cast from a `Card` object to a `Minion` object and vice versa. Thus, we had to create our own method that would attempt to dynamically cast `Card` to subclasses of `Card` while also maintaining/casting the data fields unique to that subclass type. Note that the code does not require dynamic casting between different subclasses of `Card`, as that will result in plenty of errors and is never done anyways. To resolve this issue, we created the dynamic_unique_cast method in the player.cc file, which takes two unique pointers of different types and converts the source unique pointer type into the destination type. The method works well enough to suit the needs for this program, and has not produced any errors.

We also used unique pointers for the `Player` objects, as this ensured that we did not accidentally duplicate player objects when attempting to retrieve one of the pointers, or modifying a Player's data. The pointers to the `Player` objects are frequently needed to access data, so ensuring the pointers always point to the correct instance is very important.

For the implementation of cards, using an interface and inheritance is by far the best option in our opinion, as the four card types are very similar yet very different. Thus, they need to be able to be stored in the same STL containers yet support many different methods. We do not actually ever need to create any normal cards (unless we are casting to a Card, in which case we need no data) so the `Card` class is an abstract interface.

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

For enchantments, we thought using a chain of responsibility design pattern would be a better way of implementing enchantments for a simple program without many cards as opposed to the decorator pattern we had originally planned. This is because it is easier to implement, as we simply just store all of a minion's enchantments in the `Minion` object itself instead of having a base object, a decorator, and many subclasses of the decorator. Passing the attack value to be modified by every enchantment in the vector is just as simple as with decorators.

There are drawbacks to using this design pattern. All the data for the enchantments has to be stored in the minion itself, which can use a lot of memory for that one specific card, making it slower to constantly calculate values. This would be more relevant if the game was larger with many different possible enchantments to apply, or if enchantments were more complex. However, we believe the benefits of using a chain of responsibility for this specific program outweighs its drawbacks.

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?

We would implement an `Ability` class, and likely a framework/design pattern to create multiple different types of abilities and how to execute them, instead of simply hard-coding all effects, their target selection, and all types of triggers. The builder design pattern might be useful to implement various different types of abilities into the same class. However, for a simple game with relatively few abilities, we could implement a series of abilities similar to how we implemented enchantments (with a vector of `Ability` objects) and simply iterate through them all to trigger or activate effects as necessary.

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

Our response does not change for this question, as our code does not maximize code reuse very well. We can design an `Ability` class that is a parent of two subclasses: `Activated` and `Triggered`, which contain the different abilities of each type that can be implemented into cards. Then, we simply attach the abilities to their respective cards and implement a way to trigger them accordingly such that they can target/affect both players' boards (we struggled with this, and our only option in the end was to implement abilities entirely in `Sorcery`).

For such a small game however, we believe that implementing abilities without having separate classes and design patterns for them may be beneficial as it is easier to implement without significant errors. The code is not as easily scaled, and there is more default functionality as abilities in `Sorcery` can call all of `Player`'s methods.

Resilience to Change

Our project supports the possibility of various changes well, unless the changes are complicated. With the exception of the **Sorcery** class (as we implemented abilities in it), the rest of the code has a relatively high level of cohesion and low coupling. Our different card types are separate from each other, as are different card holder classes, and can facilitate any minor adjustments to how each type of card is implemented, or different game rules.

This also allows us to implement new cards much easier, though we did not choose to implement any. Again, with the exception of adding and handling abilities, all it takes to implement another card is to add the card name to the appropriate `vector` in `Deck` and add the card stats to the `map` in the appropriate class. The data for all the card types could even potentially be moved into various text files so users can easily create their own cards without requiring access to the code, assuming they know how to create them. We can do this with csv files and file readers (without complex abilities of course). All in all, easy modifications such as

That said, the ability methods we have now in `Sorcery` can do quite a few things. Any type of trigger can activate any of `Player`'s actions, allowing for a lot of flexibility. This means that if we wanted, we could easily add cards that allow players to draw cards, gain life, gain mana, reset actions or even cast implemented enchantments onto other minions!

Extra Credit Features

We did not add any extra credit features except two slight quality of life changes, highlighted in the demo.

1. Minions now display "Action" if they have an available action. This is removed when the action is used, and is displayed when the Minion gains an action again.
2. Minions now display "X" as their ability cost if their abilities cannot be activated (from the Silence enchantment).

Additionally, our code uses only STL containers and smart pointers!

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

Our answer has changed from DD1, as we did not realize there was already a class available to us, `ascii_graphics`, that helped us implement graphics to our game. We would support two or more interfaces as the ascii graphics methods implement it, by having an entirely separate class(es) return the printed output, and select the interface we would like to display in that class. Then, we simply pass in values, and the printed output for our selected interface is returned.

Final Questions

What lessons did this project teach you about developing software in teams?

Going into this project, our team members had expertise in different areas of software development. One was more experienced and familiar with design patterns, one was more experienced with using smart pointers and one was more familiar with collaborative development with version control tools. Working on a complex project, which required multiple components was not an easy task and resulted in a few roadblocks. As such, we have learned a few valuable lessons.

Firstly, completing a group project mandated that we needed to find a medium to collaboratively code and share ideas in an efficient manner, where all members were familiar with the environment. Initially, we tried to use Github to manage version control and collaborate together, but ultimately decided against it due to the steep learning curve. Instead, we decided to code our assigned tasks individually, locally on our laptop. This was a big mistake as it did not ensure any sort of backup of our work if our laptop died and didn't restart. Unfortunately, this event occurred. Following this, we learned the importance of backing up our work. As such, we discovered an online IDE that supports live-collaborative coding, named repl.it. From this unfortunate event that involved losing many hundred lines of code, we now know to always have some sort of version control that aids in backing up our work.

Secondly, we learned to ~divide-and-conquer~. As cliche as this sounds, this was very important for us to work efficiently as a team. Given that this project required many tasks, and that some group members were more familiar with the Heart Stone game than others, it was important for us to allocate tasks according to our knowledge and effectively communicate with one another. A key lesson for communication was to always comment on our work as we code, to ensure every group member understands what the other has written and can work with it, to enhance it or to use it in another function (though this was mostly done through calls or messages). Moreover, those who were more familiar with the game itself were tasked with implementing card-specific classes, to ensure all the details were implemented.

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

We likely would have started way earlier and backed up our code, as mentioned previously. Having more time to familiarize yourself with your team members and their unique way of coding and explaining will drastically improve the teamwork and rate at which things are coded. Having more time to do things would also let us implement more design patterns and better implementations to make our final product better, more aesthetically pleasing, go beyond expectations, and improve the scalability of the program. All the issues our team encountered were related to time, as we barely had enough time to finish what we did accomplish.