Sorcery

Chow Sheng Liang (slchow), Anthony Cho (a25cho), and Hong Xiang He (hx2he)

INTRODUCTION

Our general approach to Sorcery was to imagine it as an actual card game in real life and break it down into its components and structure our objects around those elementary components. The goal was to identify properties that were unique to said components, as well as their relationships to other components and translate that into our program classes.

We believe the most probable changes to occur to our program would be the addition of new cards and/or possible status effects on minions. For this reason we decided to structure and design our program to be most suited to implement new cards with unique actions, and for our minions to be as flexible as possible.

OVERVIEW

In our program, the main data type would be Card. Besides that, we also have the Player and Board class which both act to give structure to our program. We also have a Printer class to support all of our interfaces. We structured our program around the MVC pattern, with the model being Board and Player, the view would be Printer and the controller would be main.

Our Card class is broken down into 3 different types, Spell, Ritual, and Minion. Every Card at least has a function to get its magic cost, name and the type of card it is. Every Card is also an observer with at least a function listening in on the start of turn, end of turn, addition of card to board, and removal of card from Board. These observer functions are only overridden if they are needed/part of a card's property. In addition to that, we also have a special form of Card called Enchantment that can act as a decorator for Minion. Every Enchantment inherits from Card so it has a function to return its type as well. Every Enchantment is also initialized with a null pointer in place of its component; a Minion would be attached to it if it is played.

Because of how we structured our program, every Card would be its own unique concrete class. This allows for each card to be uniquely defined but still have the existing framework to easily implement new actions.

Main would own a Board and a Player as part of the MVC pattern. The Board would have 2 Players, and Card pointer vectors to act as the board grids for card placement. The Players would each have 2 Card pointer vectors, one for the player's hand and the other for the player's deck.

DESIGN

The first thing we needed to agree on was how we were going to store Cards. To maximize efficiency in our program we decided to store all Cards in heap through a unique pointer and whenever we needed a copy to pass through a function we would use a regular pointer.

We needed a way to store multiple types of Cards in an array as well as be able to call common functions without having to identify their types, so we decided to use polymorphism to tackle this issue and came up with our sub-class structure of Card.

Another challenge we faced was also how we would consistently trigger passive actions that relied on certain conditions. Taking into account future changes and also because both rituals and minions have passive actions, we decided to make the entire Card class an observer of the Board with unique functions for each condition. This makes it a lot easier to implement new cards on existing triggers, we simply have to override necessary functions, it also allows us to watch for multiple triggers if necessary. Because Spell is included in Card it also sets up the framework for trap cards that activate under certain conditions.

The other hurdle was how we wanted to implement the Enchantment class. We decided it would be best to implement it as a decorator to Minion for the advantages discussed below. However unlike the usual decorator we needed Enchantment to be able to exist as a standalone, so our initialization diverged from the usual pattern as discussed above.

A dilemma we had was also if it was a worthwhile endeavor to create a separate subject class to alert our Card class. We decided against this and instead we would leverage on the fact that every Card class can be an observer. Our solution would be just to parse through both arrays Board containing each player's active cards in APNAP order regardless of if they need to be triggered or not. While we may trigger some dead-end functions, the wasted computation is trivial especially when we consider the size of the board, the most wasted function calls would only be 12 per event.

The last big design decision we had to make was how were we going to apply board wide effects. To us it seemed unreasonable and involved too much coupling if we were to uniquely identify Cards that required to be passed an array of Card pointers so they can affect the whole board of cards rather than the usual case of just one other Card. After thorough discussion we decided on passing a Board pointer to the Card class so it can call an appropriate function in Board to accomplish a Board wide effect. The reason we decided to tackle it this way is because not only does it allow us to leverage on reusing generalized functions in board for other cards that have board wide effects but the alternatives just involved too much complications and goes against our philosophy to make it as simple and easy to implement new cards.

RESILIENCE TO CHANGE

As discussed in the introduction we expect most changes to come in the form of new cards with new actions. Examples of such changes could be a trap spell card, enchantments for rituals, or a minion with an action once it is placed on the board. Because of how we structured our program, the previous changes can be added as such:

Trap spell card - Override trap condition as all cards are observers, add another Card pointer vector to hold all active trap cards, add new vector to the list of vectors notified in Board.

Enchantment for rituals - Create another Enchantment equivalent decorator card inheriting from Ritual, proceed the same as Enchantment for Minions.

Minion with action once placed - Override Card added notifier to catch the addition of itself and trigger action.

Other changes we anticipated could be a special events where players go twice, players get extra magic, or cards have extra actions, but those could be implemented by simply changing the control flow in Board or changing a Player/Card initialization respectively

More complicated changes could include players being able to directly attack each other like Hearthstone or having status effects Minions on deck like “on fire” which damages every round. While more intricate, these could be implemented by adding an attack field to Player and adding an attack function in Board or creating a concrete enchantment that is only attachable through initialization and is never created as a stand alone Card, and have it observe the start of turn which would be as easy as overriding a function as all Card class children are observers.

ANSWERS TO QUESTIONS

<u>How could you design activated abilities in your code to maximize code reuse?</u>

In the context of our program we expect most actions to be unique like the minimum deck we need to build with a few exceptions like summoning minions. However, if it was the case that many different cards would share similar activated abilities, like the ability to damage without taking damage or to summon more minions.

The best design for this situation would be decorators for Minions that contain general action functions that are decided upon on initialization. For example a decorator could have the deal damage function, but the amount of damage dealt would be set during its initialization. This way activated abilities could be easily stacked and combined on top of each other. We do not think this is necessary for our program, especially since every action in the default deck is unique, it would be hard to create a generalized function and not worthwhile.

<u>What design pattern would be ideal for implementing enchantments? Why?</u>

The ideal design pattern for an enchantment would be the decorator design pattern. This is because enchantments can only be played as an attachment to a minion, and enchantments can either add to or completely override a minion's actions passive or active. Besides just having these core characteristics, a decorator pattern implemented in our way where it can be initialized to be a standalone card and be attached to a minion later which is similar to real life which would not only make it easier to code with but aligns with our vision for the game. The decorator pattern also allows us to detach when needed which is an action of a default spell card.

<u>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?</u>

A combination of the observer and decorator design pattern would help us maximize code reuse, similar to how we have it implemented in our program. A main Card class would be an observer with notifying functions listening to the Board, and each minion can be initialized with decorators that contain activated or triggered abilities. Because Card is the observer, all decorators are observers as well as they inherit from Card which makes triggered abilities much easier to implement. Then a minion would be initialized with its name, attack, defense and a combination of decorators to describe its activated and triggered abilities.

Depending on how we handle the abilities, when it is activated it could only run the latest ability or all the abilities attached by either returning directly from the decorator or calling the trigger of the previous card before returning. The main complication comes when we want to allow the option to activate a specific ability. A solution to this would be to map every decorator added, meaning Minion would have key "base_abil", and every decorator with an activated ability would have a unique key like "damage_abil" or "sumon_abil". Then to select the ability to use, the user would need to input the specified key, this makes the user experience easier than relying on a numbered index.

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

We would use a similar implementation to our program, where we would have a general Printer class that holds the updated pointer to Cards on hand and the Board. Since we are storing them as pointers, all updates to a Card's status will be automatically updated as well. This means we only need Printer to be an observer of Player and Board whenever a card is added or removed so that we have an accurate presentation of the board and hand at all times. Then each interface should have its own class that contains functions that can extract and format information from a Card pointer to display accordingly. Depending on whether it is a real-time interface or not will determine whether we update it every time our notifier in Printer goes off or every time a Printer function is called.

This implementation essentially allows for us to interact with only Printer to deliver on our interfaces without messing with any of our logical components.

EXTRA CREDIT FEATURES

Some ideas of extra credit features were mentioned above. They include but are not limited to : trap Spell cards, enchantments for Rituals, Minions with an action once placed, Player abilities, and status effects on Minions.

These come as a second priority to the main program.

TIMELINE

- [ ] November 24 : Set up skeleton code, class fields, and empty functions from UML and comment as necessary
- [ ] November 25 : Set up input interpretation, basic error handling, and Printer class
- [ ] November 26 : Be able to initialize Cards and pass them between vectors in Player and Board
- [ ] November 27 : Be able to attack, use Spells, add/remove Minions/Rituals from Board and check that notifiers go off properly
- [ ] November 28 : Implement Rituals and triggered abilities
- [ ] November 29 : Implement Enchantments and activated abilities
- [ ] November 30 : Implement remaining details, player magic and Minion action points
- [ ] December 1 : Implement GUI
- [ ] December 2 : Program testing
- [ ] December 3 : Documentation/Enhancement (Flexible)
- [ ] December 4 : Documentation/Enhancement (Flexible)
- [ ] December 5 : Documentation/Enhancement (Flexible)