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, description, player ID who owns it, the type of card it is, and the type of cards it can be used with. 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 our final version we decided to not implement Enchantment as a decorator because of various reasons we would get into in the design section.

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 unique pointer vectors to act as the board grids for card placement. The Players would each have 2 Card unique pointer vectors, one for the player's hand and the other for the player's deck.

DESIGN

The first design challenge we faced was how were we going to store and move our Card objects. We decided on heap allocating all of our Card objects and storing their pointers in a unique pointer. The decision to heap allocate comes from the benefit of lower computation time compared to a pass by value approach. The reason we chose a unique pointer and not shared pointer is because we want only a single ownership over the card, i.e. we wanted the card to only really “exist” in one spot in the program at any given time. Not only does this follow our initial design philosophy to simulate a real game, but it also alleviates a lot of the dangers that come with heap allocated memory.

With our method of storage and movement decided, we needed a way to pass our objects to functions without transferring ownership. To do this we decided on using references, but this came with another challenge being references are not objects, they cannot be stored in a vector. To accommodate for this we decided to use the library <functional> which includes a wrapper class reference_wrapper that allows for us to treat a reference as an object. This workaround might be a bit of code smell, but it allowed us to pass vectors of references instead of passing them one by one which greatly reduced the amount of code and improved readability.

Following the project guidelines, we separated our cards into 4 types, and using polymorphism each type will have their own abstract class. This not only helps the structure and cohesion of our program but it also allows for us to abstract any card into a Card type and only cast it into a more specific type when needed.

To afford the most flexibility to adding new cards we decided that having each card be its own unique class to be the best way to implement this. While we could have stuck with just a Minion class, Enchantment class, etc and passed in a large number of parameters to specify our cards, we decided that while it might lead to more files in total, having unique classes for cards greatly improves readability and structures the entire program better. It also affords us the ability to be modular and include only cards that we expect to be populated. Another advantage to this method is not having to define all activated abilities during initialization, but rather have it defined in one of the many functions a card can override.

Taking advantage of the existing structure and applying a modified version of the observer pattern(It does not have the typical abstract classes since we only need Card to listen to Board), we also decided to have pure virtual notifier functions to be part of our Card class and for all cards on board to be notified when certain conditions are met. This means all cards can actually listen in on the activities on the board so long as they are played. This makes implementing passive abilities much easier as well as other triggered abilities.

While initially we decided to implement Enchantment as a decorator, the plan has changed. This is because we overlooked the fact that we needed enchantments to exist outside of Minion as well as the fact we want to only use unique pointers. Semantically, it doesn't make sense for Enchantment to have an attack and def if we made it a decorator of Minion, and neither does it make sense to make it a decorator of card when there is no way to enchant a spell. Considering that our main priority was to develop a program that fits the default requirements, the decorator pattern did not have any unique advantages except for the implementation of activated abilities to minions through enchantments which is an enhancement. So we decided to go with a simpler approach of storing each enchantment's vector of unique pointers in the minion, showing ownership, but also allows for us to more easily access the enchantments.

We also designed Player and Card and its subclasses to be completely independent from board and other runtime logic, minimising the coupling between the classes. This means that Player and Card only acts as individuals which carries out modification to themselves and sometimes to others of the same time but nothing else. They do not have access to who's turn it is, the Player's magic and any other states, meaning they are totally independent and it is up to the Board to piece them together into a cohesive system. The only exception is when a card has a board wide effect to which we would have a separate general function that can be called to apply a board wide effect.

Board command functions that were called from main are separated into its own function for clarity (i.e. end, attack, play, etc.). This allows for main to check for most invalid inputs, and simply ask the user for input again. These board command functions provide the base gameplay elements by calling other functions to perform each action and also performs most of the checks required to play the card. Some of the checks are performed in card due to private fields, which is better designed for encapsulation.

We incorporated Model-View-Controller (MVC) architecture into our program by trying to adhere to the single responsibility principle to reduce coupling between classes. We wanted the Printer class to handle most of the communication with the user as the View part of the MVC. The Board class acts as the Model, and handles most of the in-game logic that happens on the board, with some interactions with Player move Cards between different areas and some interactions with specific Card abilities that affect the state of the board. The main function acts as the controller, handling input from the user, calling functions from Printer to communicate with the user, and calling functions from Board to change the state of the game based on the user input. The main function also handles game flow, like whose turn it is to play, or checking if a player has won the game.

To print a Card, we first converted it into a printable object and printing board, player hand, and inspecting minion involved emplacing the printable cards onto the back of a vector of printable cards.

A separate Xwindows class was created to handle the graphical display of sorcery and included in the printer class. This was done to make the code more modular, reusable, and encapsulate graphical printing information that is irrelevant to the printer class. Encapsulating Xwindows class information from the printer class made Xwindow methods simpler to use, protected invariants, as well as preventing unwanted changes. Simple style graphics were used as Xwindow cannot display special characters in fancy style.

The board and current player hand is displayed in Xwindow after every move the player makes. We decided to use this approach instead of an abstract observer class for simplicity purposes, where specific functions were defined for updating them. To prevent the window from displaying when graphics arguments are not inputted, A boolean is passed to the Xwindow class to make its constructor and destructor do nothing if true. This boolean condition is also passed to the printer class to disable graphical printing.

RESILIENCE TO CHANGE

Coming back to our introduction, we designed the program to accept the most changes coming from novel cards with new stats or new actions. This is easily achievable by just creating a new unique card class and including it in the player class to be initialized. But besides just new actions you can also easily implement new triggers or even have multiple triggers on a card because of how the notify function is abstracted to all cards.

To give a better idea of the flexibility we will use minion as an example. Each minion has 2 use action functions that have an empty definition by default, where one receives a reference to Board and the other has a reference to Board and the targeted Card. Depending on whether your new action requires a target or not you would override accordingly. From there you have full access to both the board and the target minion(If you require so). If you need a novel board wide function, you can simply define one in board and call it from the definition of your overridden function in the new card class. The same idea is applied to triggered abilities with our four defined triggers: start turn, end turn, minion enter, and minion leave.

Other than just new actions we can also relatively easily implement completely new functionalities. An example could be spells that act as trap cards. The only modification would be to override the notify function in the new spell class and include used spells as part of the list of card objects that listen in on the 4 notifiers on board. If these trap cards require more complicated requirements, they can simply access the information on board since all notifiers are passed a board reference.

Because of how we have structured our classes and functions there is always an easy workaround to implement any new feature that isn't creating a completely new game. This is because in a turn based game, there are only so many things that can change in a single moment and we have notifiers for almost all of those crucial moments. We have also abstracted Card so that a new type of card could be easily created without much trouble and also made it incredibly easy to add on to existing types since they already contain most of all the actions said type is capable of as well as access to whatever a new card would need access to as a result of being a child to one of our existing card type abstract classes.

However we would also like to mention that an decorator design pattern for our enchantment cards would afford us more flexibility in the range of new functionalities we can implement easily, though because of our time restrictions and since our priority was the default requirements we decided on the easier implementation rather than the more flexible one. An example where a decorator pattern would be advantageous over ours would be an implementation of status on minions, say "onFire" which would damage its attached minion every start of the turn. We could still implement it with our current design, but it would involve storing a minion reference object(reference_wrapper) in all enchantments and using that to point back to the minion that owns it, unlike a decorator pattern where that information comes for free.

If we wanted minions to have the ability to perform immediate action instead of having to wait a turn after they are placed to act, we could simply change that by initializing the actCount to 1 instead of 0 when constructing a minion. This would be a very simple change to make and will not affect other parts of the program (other than the fact that they would be able to act on their first turn).

Some potential handicap changes that could be implemented would be more magic or health for the players, 2 turns in a row for a player. If we wanted Players to start with more magic or health, only a simple change to the Player class would be needed to accept another parameter for magic/health and it would be passed in using a command line argument or asking the users how much magic/health they want before the game flow starts. If we were to implement a handicap for a player to have 2 turns in a row, with our design, the change would be simple, as nothing would have to change in any class other than the main function, because the main function handles the game flow and controls whose turn it is.

If we wanted more than 2 players, this would also be a simple change in main, as we did not cap how many players we could have, but the program only currently supports 2 players. To support more players, we would only need to make slight changes to main and accept a command line argument that indicates the number of players playing. We would also need to make a significant change to the Printer class, specifically to the Graphics window output, as we would need to be able to display more than 2 players at once, and we currently only support 2 players for the Graphics window display. To scale the game, minimal changes to classes should be necessary (other than Printer for the Graphics window), and only a few changes should be needed in main.

To accommodate for changes in the printer class, the methods were broken down into smaller components with helper functions to improve code reusability. This promotes modularity for the printer methods as well as minimizes potential changes. The encapsulation of the printer and window class also allows changes to be made easily and safely as it allows modification of the classes without affecting the code using them.

ANSWERS TO QUESTIONS

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

Based on our new structure and program we believe that the best approach to this would be generalized functions in the Board class. This would be similar to how we implemented summon or heal all. Since all Minions have access to a Board reference when their use action is called as well as a target Card reference if the action requires it, every “new” activated ability could be easily implemented with just a change in the number the class calls the function in Board with.

A prime example of this would be the summon function in the Board class and both the apprentice summoner and Master summoner. You can take a look at their implementation and see that they are calling the same function with different parameters and this saves us from writing the code twice.

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

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 has a link back to the minion which owns it and can also be easily detached, which would be very useful for future implementation of new functionalities. The decorator pattern also would reflect our initial vision of implementing the game as if it existed in real life.

Looking back with hindsight, we should have implemented our enchantment class as a decorator to the Minion class despite the semantics not making the most sense when initialized on its own. The reason we deviated from our original plan to actually implement it as a decorator was due to time limitation and also our overarching vision for the program was not too clear.

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?

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 excluding the decorator. 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.

After our experiences with development on this project, a change we would make to this answer would be to create base classes for activated and triggered ability. We would store the activated and triggered abilities in 2 different vector fields in Minion and then apply the observer pattern to notify triggered abilities based on conditions, and to activate triggered abilities, we would require an additional input for index for which of the activated abilities the user would like to activate.

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

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 essentially allows for us to interact with only Printer to deliver on our interfaces without messing with any of our logical components. The actual implementation ended up combining the 2 interfaces in Printer class with Main calling printer functions to update the window display after every move, and printing the text display in the terminal only when it's called.

EXTRA CREDIT FEATURES

Some ideas of extra credit features were mentioned above. In this project, we only implemented one extra feature (not including the bonus for no deletes and smart pointers), and that is to allow the user to inspect the enemy minion.

FINAL QUESTIONS

1. 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?

One thing that we did was split up the work using the MVC architecture, by having one person focus on the controller, another person focus on the different data structures needed for cards and the model, and the last person focus on the view, the output. This allowed us to work independently, focusing on our own components of the program, without having to rely heavily on the other group members, other than asking for functions to be created that we needed to be able to call within the context of our own part. This allowed us to abstract the architecture in this way for smooth communication, as we didn't need to interact very much with each other's

components of the program, we only needed to know what a certain function did at a high level (i.e. the actions/effects that the function would perform and how it would affect the state of the program where it's being called). In a sense, the MVC architecture provided us the perspective of the classes we have worked on.

Another lesson was that we had a lot of back and forth discussion and clashing ideas in the planning stage, as there was no main architect, so we took a lot longer than desired to plan and design the project. Thus, the solution would be that if we had designated a team lead, or main architect, we would have been able to have one decision maker who designs a potential solution and the other group members would participate in critiquing the idea to improve the proposed solution.

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

One of the main things we would have changed if we had the chance to start over would be to implement our Enchantment cards to be a decorator of Minion. As we mentioned above, the decorator pattern offers more advantages in terms of implementing new features, but due to our time limitation and lacking vision for our program we decided to go with the easier to implement approach than the more flexible approach.

This comes back to us having a main architect. If we could start over we would have chosen someone who is most experienced to build a rough archetype of our program and establish the overarching vision and design philosophy instead of bouncing the ideas between the three of us.

To add to this it would also have been very helpful to have some kind of log of our discussions so we don't end up going in circles during the planning stage. We should have had some kind of structured discussion and once a decision was made on the design it would be final.

We also realized the usefulness of the compiler and syntax correction offered by VS code. Because we had spent more time than we should have planning we didn't realize we had some flaws in our design that would have been easily pointed out if we had started coding sooner and been alerted by the IDE.

In addition, we should have also tested out certain functionalities in a vacuum before implementing it in our programs as there are some errors that can't be detected by the IDE and only occur during runtime. But overall our decision to use smart pointers and more syntactically "restrictive" approaches actually saved us a lot of time debugging as the errors were pointed out before runtime.