Design Document - Sorcery

Mustansir Soni, David Kim & Rebecca Chu

Overview (describe the overall structure of your project)

The overall structure of our project can be classified as Board, Player, and Card class. The Board class uses two pointers of Player class to represent Player 1 and Player 2, and each player can use 0 or more cards of Card class. The Card class is an abstract class whose child classes are AbstractMinion, Spell, and Ritual. Since all child classes are a "card" and share similar properties such as the name and magic cost, we used inheritance to represent such a relationship. By simply inheriting the properties and functionalities of a Card class, we were able to maximize code reusability in the child classes and enhanced the overall readability and structure of our project.

The AbstractMinion, as the name suggests, is an abstract class that can be either a minion itself or a *decorator* to a minion, known as an enchantment. Since enchantment adds an additional functionality to a minion dynamically, and a minion should be able to decouple from the additional functionalities, we introduced a decorator design pattern. We were fully aware of the drawbacks by the incorporation of this design (i.e. complex code, abstract information manipulation), but we realized that there were at least 6 decorators (by default) and subclassing the minion class to to similar the decorators would result in an explosion of subclasses to support every combination of decorators. By introducing this design pattern in our project, we were able to diligently handle any type of enchantment in any combination, including adding, removing, and printing the *extended* abilities.

Design (describe the specific techniques you used to solve the various design challenges in the project)

The overall design of our classes also enables us to use polymorphism for the Card and AbstractMinion classes. We thought this was the best way to implement it as this simplifies our player and board classes, and also improves functionality of the card class.

We recognised that the flow of information for maximum efficiency should be as follows:

- The main controller should call the Board's methods.
- Board class should handle the player's actions.
- Player's should be responsible for activating the Cards' methods

- The Cards' affect the board in some way.

As you can notice, this leads to a sort of circular dependency issue which we solved by using a forward declaration of the Player class. This issue arises because the cards need a pointer to a board to be able to affect the board.

We designed to have high cohesion and low coupling. To minimize our cohesion we minimized the use of public fields. One function doesn't need to know how the other functions work. Also, all the methods and fields are related to their own classes.Instead of displaying everything in one class, you created display function for each class, thus the board only had to use their functions to call, thus achieved high cohesion and less coupling (didn't have to use the getCardType for example)

One of the design challenges was the implementation of the Minion's abilities. We thought it best to design an Ability class as a subclass of Enchantment. This way, if any improvements to the design of the game are to be made regarding the number of abilities a minion can possess, they can be made very easily as a minion can have any number of enchantments.

Another challenge we faced was to handle exceptions and print a meaningful error message that pertains to that specific situation. To solve this, as well as keep our code modular and loosely coupled, we introduced our own exception class in a separate file, inputException.h. That contains the error message and a boolean value that determines whether the program should terminate or not. Having a central exception class that all our classes use allows us to be able to differentiate between an exception from the standard <exception> library and that of our own, which led to creating and displaying a more helpful and specific error message. Additionally, by storing all the parameters of a command in a vector, I was able to calculate the number of parameters in a very efficient manner (simply by checking the size of the vector) and call different methods accordingly, or throw an error message to let the user know that the number of parameters is invalid. Since our project involves lots of inheritance relationships, we also cleverly used the Partial Exception Handling technique, to add more meaningful messages and rethrow the exception.

One other challenge we faced in the controller was to seamlessly switch between a file input stream and a standard input stream. By having an std::istream pointer that can point to either an std::istream object or a std::ifstream object (inheritance and polymorphism), we were able to handle both input streams without creating all possible combinations of istream and ifstream object. We also wanted to go beyond and improve our user interface by asking the "Enter the name of player i" in the display. To accommodate for changes in the number

of players, I have created a constant NUMBER_OF_PLAYERS that acts as a single, central source of information for the number of players and made a for-loop to display the message for each ith player. One unique part of our controller is the flexibility it provides. By smartly using the input stream library (i.e. getLine and eof), we were able to support both the classic ctrl+D and our command quit to safely terminate our program, and simultaneously provide versatility in the commands by truncating all whitespaces. We were able to achieve this by reading inputs from stringstream, which by default, ignores all preceding whitespaces. Thus, in combination with getLine, I was able to read the full name of each player (i.e. you can name a player as "David Kim") and handle all commands regardless of their amount of whitespaces:
"play 1 2"
" play 1 2 "

Another major issue that we faced was the circular dependency of our project. By the nature of a group project, it was very easy for us to get into the trap of circular dependency. Since everyone worked on their own assigned classes, when we merged all classes together we found lots of extraneous included libraries. In order to fix this, we took advantage of our inheritance relationship, and removed all the generic libraries in the subclasses, and included them in the parent class. Thus, the subclass could simply include the parent class and get all the libraries they needed. If there was a specific library that a certain class needed, we included in their own .cc files instead of .h, such that no other file (who could possibly include this file) would not include those libraries (that they may not need). In addition, we used the preprocessor guards for all .h files to avoid class re-declarations, and replaced all libraries with class declarations in files that only needed their incomplete data types (such as a pointer). Through these techniques, we were able to completely eliminate the circular dependency.

Resilience to Change (describe how your design supports the possibility of various changes to the program specification)

Our program design allows for easy modifications and additions if desired. We can add more types of cards. Also add new cards of types that already exist. If you want to add enchantment just add header (.h) and implementation (.cc) files accordingly. If you want to add a minion just add if statements putting conditions on the name to the AbstractMinion class constructor in the AbstractMinion implementation file and it's abilities in the ability .h/.cc files. If you want to add a spell or ritual card, simply add if statements putting conditions on the name to spell or ritual constructor in spell.cc or ritual.cc respectively. You would also need to add these names to the deck.cc file in the "dictionary" for names of cards.

Answers to Questions (the ones in your project specification)

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

This answer changed from due date 1 because we realized it was more efficient and better to implement abilities as enchantments rather than spells. Also, cards won't need to include the ability as a field.Design abilities as enchantments because they are an extension to the minion card's capabilities.

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

We used a decorator pattern as we decided in due date 1. It was the best way to achieve any number & combination of activated and triggered abilities to each minion. It allows us to easily add functionalities (modifications) to an object at run-time. Although we can achieve the exact same effect with subclassing & inheritance, having an abstract decorator pattern that only concerns about 'decorators' like enchantments allows us to achieve separation of concerns and modularization, which greatly reduces the code duplication.

3. 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?

This is the same answer as due date 1 and we implemented abilities exactly this way in our code. Using a decorator pattern, we can add any number & combination of activated and triggered abilities to each minion. Since a decorator is an abstract class, we can simply add concrete instances of a new type, and we can stack up their instances for each minion (like a chain of abilities). For example, we can pass our player and its opponent player, and each ability will do their own unique 'ability' on the opponent player as well as on its player (i.e. restore some resources), and pass the players to its next ability in line.

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

We support the ncurses interface but we didn't implement it as a separate class. We didn't require any changes to the rest of our code either. We implemented the interface using an if statement as ncurses no longer requires the text interface so both will not be used simultaneously. Create an abstract display class that has an association to the Board. Then create concrete classes for each of the interfaces that contains their display methods and call

these methods from Board's display method. Minor editions will need to be made in adding an interface i.e you just add a class.

Extra Credit Features (what you did, why they were challenging, how you solved them—if necessary)

We created an interactive user interface using ncurses as it was specified as a choice in the pdf. The user interface has a moving cursor that can be operated using the arrow keys and ENTER. This makes the game much simpler to play. This was of course very challenging and required some study of using the ncurses library and functionality.

Final Questions (the last two questions in this document)

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?

We learned how to use GitHub overall, especially branches and pull requests. We discovered the benefit of using branches so there's no merge conflicts. We also learned the importance of communication and making sure everyone is updated and on the same page. Each time we made a change to our branch or the main branch we notified the team. We had numerous pull requests to keep track of the changes and make each team member review and approve the changes. It is smart to divide the work up from the start so everyone has a part to work on. This made it more efficient and less likely of merge conflicts.

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

We would have read the assignment outline and instructions more thoroughly from the start to make our implementation process smoother. We spent a lot of time working on the main controller because we were misunderstanding what needed to be done. We should have done small changes at a time and then commit instead of commiting only when something big was changed. Doing so would have made it easier to debug and see where the program got an error.