Plan of Attack for Sorcery Game Program

Adeline Su	a26su	20996771
Julia Zhu	j44zhu	21019909
Sherry Feng	s5feng	21006664

Friday November 24th, 2023
CS246

1| Project Breakdown

In this project, we plan to employ the Object Oriented Paradigm in C++ to implement a game of “Sorcery” for two battling players using magical cards of various types (Minions, Enchantments, Spells, Rituals) to effectively “kill” the other player. Users will interact with the game on the command line and receive visual output on the command line and a graphical display. The program will be runnable with various command line arguments. As a group of three members, we plan to tackle this project within a two-week timeline by separating the coding into four phases, ensuring that a complete runnable program (with different enabled features) is produced at the end of each stage. We will also consider the use of the Decorator, Observer, and MVC architecture design patterns as probable solutions to anticipated problems including: activated abilities, enchantments, and extensibility of activities and interfaces.

2| Order of Project Tasks

We will tackle this project by separating coding tasks into four phases, three of which make up the specified features in Sorcery, and one stretch phase as a chance to add bonus features. The separation and order of tasks will largely follow those outlined in the project specifications, with some adjustments from our end.

Phase 1: Planning & Game Setup

This phase involves understanding the project requirements, frequent meetings within the team to plan the architecture of the project, and implementing features that happen before gameplay begins.

i	Understanding specifications	Reading and discussing specifications, playing Hearthstone to get an intuitive understanding.
ii	UML Diagram	Required classes, high-level relationships, and design patterns to be used.
iii	Writing Header Files	Discussion of coupling and cohesion. Flushing out the ideas started in the UML diagram, including parameters and how information will be passed between files.
iv	Compiler Flags	Receiving compiler flags in main.cc
v	Initializing Players, Decks	Skeleton functionality to load decks from file, creating Players with names.
vi	Deck Shuffling and Card distribution	Preparing deck and Player's hands for the beginning of the game.
vii	Game Loop	Working on GameController.cc and GameMaster.cc

Phase 2: Basic Battle Features

After this phase is complete, the game should run from beginning to end with limited features. In particular, the functional cards would include minions and spells, where minions can only attack other minions and players, spells can only interact with minions.

i	Text display	Working on Display.h and TextDisplay.cc so that we can test future functionality against visual output.
ii	Minions that can only attack	DefaultMinions without activated, triggered abilities, or enchantments, allowing them to attack players and minions.
iii	Spells that can only interact with Minions	Spells have an ActivatedAbility, thus implement ActivatedAbility and apply to Spell
iv	Life and Magic Points	Deduct life and magic points as required, ending the game or providing reprompt messages as part of game loop.
v	Compiler Flags (con't)	Functionality related to the -testing compiler flag will be implemented alongside the other features.

Phase 3: Advanced Battle Features

This phase completes the project features as specified. This includes magic, enchantments, rituals, abilities.

i	Activated abilities	Integrate ActivatedAbility into Minion
ii	Enchantments	Implement Enchantment cards and functionality under Decorator design pattern
iii	Rituals, triggered abilities	Rituals' functionality comes from aggregation relationship with TriggeredAbility. Thus implement TriggeredAbility under Observer design pattern
iv	Remaining complicated cards	Outstanding cards in any subclass
v	Compiler Flags (con't)	Functionality related to the -testing compiler flag will be implemented alongside the other features.

Phase 4: Bonus Features (Stretch)

This phase opens a possibility for our team to implement additional features that are contingent on time availability.

i	Graphical display	Use XQuartz to implement GraphicsDisplay subclass under the MCV architecture design pattern.
ii	Other interesting features	(decorator design pattern)
iii	Compiler Flags (con't)	Functionality related to the -testing compiler flag will be implemented alongside the other features.

3| Estimated Deadlines and Division of Work

Sun	**Mon**	**Tues**	**Wed**	**Thur**	**Fri**	**Sat**
19	20	21	22	23	24	25
Draw UML diagram			Finish Phase 1		**DD1** Finish report	Finish Phase 2
26	27	28	29	30	1	2
			Finish Phase 3			Finish Phase 4
3	4	5				
Testing	Testing	**DD2**				

The team will hold daily meetings to hold discussions about the code and direction of the project. At the beginning of each phase, we will divide labor for the few upcoming tasks. So far, it has been decided that the team will collectively create and update the UML diagram during meetings; Adeline will handle compiler flags, initializing players and decks, and shuffling the deck; Julia will work on default Minion attacking behavior, and Sherry will work on enchantments on Minions with the Decorator design pattern.

4| Solutions to Anticipated Problems

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

We create an ActivatedAbility class and create a composition relationship between Spell and DefaultMinion. Thus Spells owns-a ActivatedAbility and DefaultMinion owns-a ActivatedAbility, since both cards are able to change the game state when they are explicitly played by the player. This maximizes code reuse since we only need to create one concrete subclass per ActivatedAbility, whether it will eventually be used for a Spell, a DefaultMinion, or an Enchantment. This way, there will be no duplication of code in order to implement ActivatedAbility.

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

The decorator design pattern would be ideal for implementing enchantments. Enchantments are modifications that can be played on Minions, such as modifying attack and defense values or granting new abilities. The decorator pattern is used to add new behavior dynamically at run time to objects, which is what enchantments are to objects of the Minion class. This works in this case because in Sorcery, enchantments on Minions are applied by users while the program is running. We will essentially treat each Minion object as a "linked list of functionality," which terminates at the base Minion, which is just the Minion with its original functionality. Each enchantment, then would be a decorator to be layered on the base minion.

This is effective because without the Decorator design pattern, there could be exponentially many variations of Minions that users could require, which is inefficient in the short run. In the long run, an implementation without the Decorator pattern could fail in extensibility if users wish to specify custom cards during game setup.

Question: 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?

The observer design pattern can help us allow Minions to have any number and combination of abilities. If we replace the private ability field in DefaultMinion with a vector of ability pointers, then we can construct each ability to be either an Activated or Triggered Ability. Upon construction of a TriggeredAbility, the Observer design pattern enables us to automatically attach that ability as an element of the observerTrig vector in GameMaster. That is, the TriggeredAbility will take place once it senses the appropriate change in the game state. This requires minimal code changes since we would not need to make any changes to GameMaster; at each stage of the game loop, it would still perform the same checks through the observers vectors to apply Abilities that match the TriggerType.

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

Employing the MVC (Model-View-Controller) architecture design pattern would enable us to support multiple interfaces with minimal changes to the rest of the code. The MVC enforces the Single Responsibility Principle (SRP), whereby each class should only have one reason to change. This makes it easy to add more interfaces since the Viewers have separate responsibilities from the Model. In our UML diagram, we have the GameController, which owns a GameMaster (the model), and two Displays, where a Display is an abstract base class from which we inherit TextDisplay and GraphicsDisplay. If we wanted to add a third interface, we simply write a third inherited class from Display, and add it to the list of observers to notify in GameManager.