Sorcery: Final Program Design

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

Tuesday, December 6th, 2023
CS246

1 | Introduction

The completion of Due Date 2 marks the completion in the development of our project. This report encapsulates our project's journey, containing critical reflections on the evolution of our project from its conception, describing changes made during the development process.,

In this report, we first present a detailed overview of our program: its structure, and the implementation strategies we used. Our design adheres to object-oriented design principles discussed in class, and we incorporate design patterns where applicable. The document also includes an updated UML model. This serves as an overview of our project structure, while highlighting any deviations from the original design submitted on Due Date 1.

Furthermore, we dive into the aspect of how our chosen design accommodates change, where we emphasize choices we made to accommodate any modifications in the program specification. Next, we discuss topics on cohesion and coupling our program modules, providing insight on the connection between different components and their adaptability to alterations.

The document is organized into distinct sections, each contributing to a comprehensive understanding of our project.

2 | Overview

To begin, the class structure of our project is encapsulated by the Model-View-Controller architecture. From this, we can identify the TextDisplay and GraphicsDisplay classes as the “views”. They are abstracted by the SorceryDisplay abstract class. Meanwhile, the GameController class is the “controller”. This class bridges the communication between the “view” and the “model”. The “model”, represented by GameMaster, captures the backend and all control flow of the Sorcery game. In other words, the class that contains all the state information of the game is GameMaster.

GameMaster contains some of the most important pieces of Sorcery. At the highest level, it is the invoker in the Command Pattern, and two subjects in two separate observer patterns. The first being the subject in the observer pattern that updates the display objects, with SorceryDisplay. Most notably, it “has(-a)” players and their respective decks. Following this, players “has-a” Deck, Hand, Board, and Graveyard. GameMaster keeps track of TriggeredAbilities as observers in the second observer pattern. Thus, GameMaster also “Has-a” TriggeredAbility.

TriggeredAbility is part of two design patterns. The first being the command in the Command Design Pattern, and the second being the Observer in the Observer design pattern with GameMaster as described above. The TriggeredAbility class contains all triggered abilities as concrete subclass implementations.

A fourth design pattern is seen in the Minion class, where the Decorator Design Pattern is used to add enchantments to minions at runtime. The EnchantmentDec class is the decorator class in this setup, while the individual enchantments are the concrete implementations of the class. This pattern is used to modify DefaultMinion.

3 | Design

specific techniques used to solve design challenges
- high level explanation of how aspects of your project work together
- coupling and cohesion
- UML, differences from DD1
- OOP design
- highlight certain interesting code

4 | Resilience to Change

5 | Answers to Questions

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.~~

6 | Extra Credit Features

7 | Final Questions

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

This project showed us the importance of consistent communication and documentation, Git proficiency, and establishing common goals and timelines.

To begin, our group held work sessions and meetings on a daily basis, where we would first establish any changes made from the last session, and what we planned to work on. We would confer that we were moving in the correct direction. During these sessions, it was easy for us to vocalize any questions or uncertainties we had, where we could discuss and agree on a solution within a short timeframe. However, if we could not agree on an answer, we would jot the inquiry down and ask a professor during office hours. This working cadence worked well for us, and kept our program compiling at every stage. We were able to work on the project at a uniform rate, and with this strategy, we were certain that all group members were engaged and always on the same page. This taught us the importance of having frequent discussions when developing software in teams, and how crucial it was that everyone worked in parallel.

As we were all new users to Git just a year ago, our group was able to successfully use branches, pull requests, and other Git tools in order to work synchronously. Git was at the center of all of our version control, and we were able to handle merge conflicts effectively. Without Git, we would have had to spend exponentially more time attempting to code simultaneously, which would have severely impacted our timeline. This showed us the importance of version control, and how to effectively use commands to avoid conflicts.

Additionally, our project taught us the significance of establishing common goals and timelines from the beginning of the planning stage. At our initial team meeting, we collectively defined the overarching objectives and milestones. We took extra time to ensure that everyone had a shared understanding of the project's scope and the individual responsibilities each team member would be responsible for. This initial investment in goal-setting led us to establish a more cohesive workflow throughout the development process.

In hindsight, this project also highlighted the critical role of documentation. Since we were all working on different sections of the project, documentation helped us ensure knowledge transfer and project continuity. This documentation not only helped the team members organize code, but also served as a valuable resource for addressing issues that arose during the project's maintenance phase.

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

If we had the chance to start over, we would have focused on more detailed planning while drawing our UML diagram, spent more time exploring the technology stack, and better optimized workload allocation.

While coding, we found that there were discrepancies between our implementation plan in our UML diagram versus what we needed to implement to achieve what we wanted to accomplish. There were some details that we didn't foresee or didn't anticipate. For example, a few days after completing our UML diagram, we spent lots of time discussing how we could implement the "Blizzard" spell, which changes the state of the entire board, instead of just one minion like the other spells. If we could have been more rigorous in our initial discussions, perhaps we could have caught this discrepancy earlier on, and implemented a more thorough solution more quickly.

Related to this, one challenge we faced was learning new topics in class that would have simplified our implementation, but requiring significant changes. For example, we learned about variants in class when we had already implemented Triggered Ability and Activated Ability functionalities. This led us to change our implementation in the DefaultMinion class from two TriggeredAbility and ActivatedAbility pointers to a variant that contains both.

A more creative idea that we could have done differently, was spending more time revisiting the technology stack and development tools early on could have improved efficiency. If there were any new tools, or libraries (such as the algorithms library) that could have aided our development during the project, incorporating them at the forefront might have made our development more streamlined and less error-prone.

Another area for improvement is related to the allocation of resources and workload distribution. In a team of three, it's important to ensure that each member's skills and strengths are effectively utilized. A more detailed analysis could have helped us identify instances where tasks might have been better assigned based on individual expertise. This would potentially have improved the overall productivity and quality of our code.

8 | Conclusion