# 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 describes 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 high-level 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 potential 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

Our class-based approach to implementing the Sorcery game involves use of meaningful inheritance relationships between types of cards, and design patterns: the MVC architecture, observer design pattern, command design pattern, and decorator design pattern.

There are two halves to our UML diagram. The first half deals with all objects that are owned by a player (Player, Deck, Hand, Board, Graveyard). The second half deals with the different types of cards. Thus it was natural to use aggregation relationships for the former and inheritance relationships for the latter.

The root of our UML diagram is the **GameController**, which communicates between the **GameMaster** and some number of **SorceryDisplays** (abstract, and has concrete derived classes TextDisplay and GraphicsDisplay. The GameMaster executes methods as called by the GameController, and it mainly manipulates the two Players it owns. A **Player**, in turn, owns one **Deck**, **Hand**, **Board** and **Graveyard** each, which it manipulates in its methods. Each of these 4 objects are mainly composed of Cards. The abstract **Card** class has derived classes up to 3 layers deep. Separate from this relationship but connected via aggregation are the ActivatedAbilities or TriggeredAbilities. The former may be owned by Spells and some Minions, the latter may be owned by Rituals and some Minions.

# 3 | Design

## 3.1 **Design Patterns**

### 3.1.1 | *Aggregation*

A GameMaster permanently owns two **Player**s, which themselves permanently own a **Deck**, **Hand**, **Board**, and **Graveyard**. These are also visually represented on the SorceryDisplays. This makes sense since it should be impossible for a Board to exist without any Players, or for a Player to exist without a GameMaster that they are playing in.

### 3.1.2 | *Polymorphism*

A game of Sorcery involves, at its core, the Players manipulating cards. There are 4 types of cards, which behave differently, interact with other cards and players, and can change the state of the game as a whole. Thus we apply the idea of polymorphism through inheritance relationships. **Spell**, **Minion**, **Enchantment**, and **Ritual** are all derived from **Card**, since they all have member fields: name, cost, type, desc, needTarget, etc.

### 3.1.3 | *Model-View-Controller Architecture Design Pattern*

The root class structure of our project is encapsulated by the Model-View-Controller architecture, which enforces separation of the game logic and the user interface. The Controller is the **GameController** class, the Model is the **GameMaster** class, and the View is the **SorceryDisplay** class. The majority of the program run time is spent in the Controller::go() method , which represents playing one game instance until a player has won or end of input is reached.

Although in our submission code, the main makes only one instance of GameController, there is potential to instantiate any number of games. Similarly, the fact that the GameController owns a vector of SorceryDisplay pointers (gamecontroller.h:12), means that we could have as many views as we want to. However under the project specifications the submitted code creates either 1 or 2, depending on the -graphics flag.

### 3.1.4 | *Command Design Pattern & Observer Design Pattern*

GameMaster contains some of the most important pieces of Sorcery. At the highest level, it is:

- The *invoker* in the Command Pattern to apply ActivatedAbilities
- The *invoker* in the Command Pattern to apply TriggeredAbilities
- The *subject* in the Observer Pattern to make the SorceryDisplays update

ActivatedAbilities: An **ActivatedAbility** is represented as a concrete class that belongs to either a Spell (e.g. an instance of an UnsummonAbility is belongs to an Unsummon Spell) or Minion (e.g. ApprenticeSummonerAbility belongs to an ApprenticeSummoner Minion). Upon the user's request, the applyAbility method (activatedabilities.cc:11) is called. So to send a 'command' back up the hierarchy to the game level, we use the Command Pattern.

TriggeredAbilities: Similarly, Rituals and some Minions own an instance of a concrete **TriggeredAbility** (e.g. DarkRitualAbility). When the game state reaches one of the four appropriate points, it loops through the vector of observers and calls the trigger() method on appropriate TriggeredAbilities. One key

difference is that we encoded information about the observers as a structure called ObserverList with 2 fields: a pointer to a TriggeredAbility, and a pointer to the player who owns the Card which owns the TriggeredAbility. This is crucial when implementing APNAP order.

Updating displays: We use the observer pattern to loop through a vector of SorceryDisplays and notify them to update some display to the user.

#### 3.1.5 | *Decorator Design Pattern*

A fourth design pattern is seen in the abstract **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.

### 3.2 | **Coupling and Cohesion**

#### 3.2.1 | *Low Coupling*

We have designed our program modules with minimal dependencies that operate more individually. This not only makes our program easier to test in isolation (through unit testing), but this is important for adding the behavior of interconnected modules.

We implemented interfaces and abstract classes that support encapsulation by allowing components to only expose the necessary details. As well, this accentuates the benefits of polymorphism, since our use of interfaces and abstract classes allows different implementations to be used interchangeably. This promotes code reuse and flexibility, since components can be replaced with alternative implementations without affecting the code that uses them. This can be seen in the SorceryDisplay, TriggeredAbility and ActivatedAbility classes.

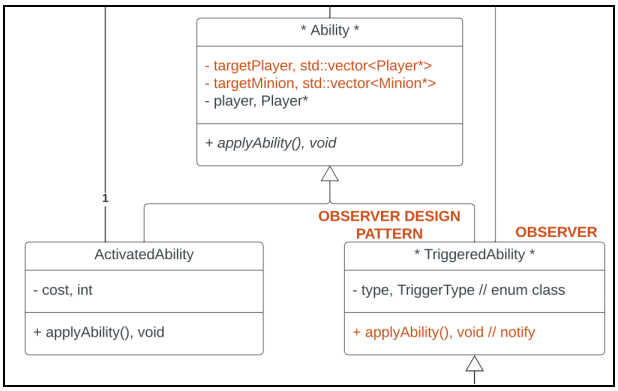
When modules have fewer dependencies on each other, changes to one module are less likely to impact others. This “isolation” allows for more flexibility when updating or modifying specific components without causing a domino effect. This means that when maintaining the program, we can focus on a specific module without considering the complexities of other modules, leading to a more efficient lifecycle of debugging, testing, and modification.

#### 3.2.2 | *High Cohesion*

We have enforced the single responsibility principle (SRP), whereby a class should only have one reason to change, which promotes high cohesion since classes do only what it is supposed to do without dealing with unrelated concerns. This is exemplified through the use of MVC architecture pattern, and separation of the GameMaster, Players, and BoardElements.

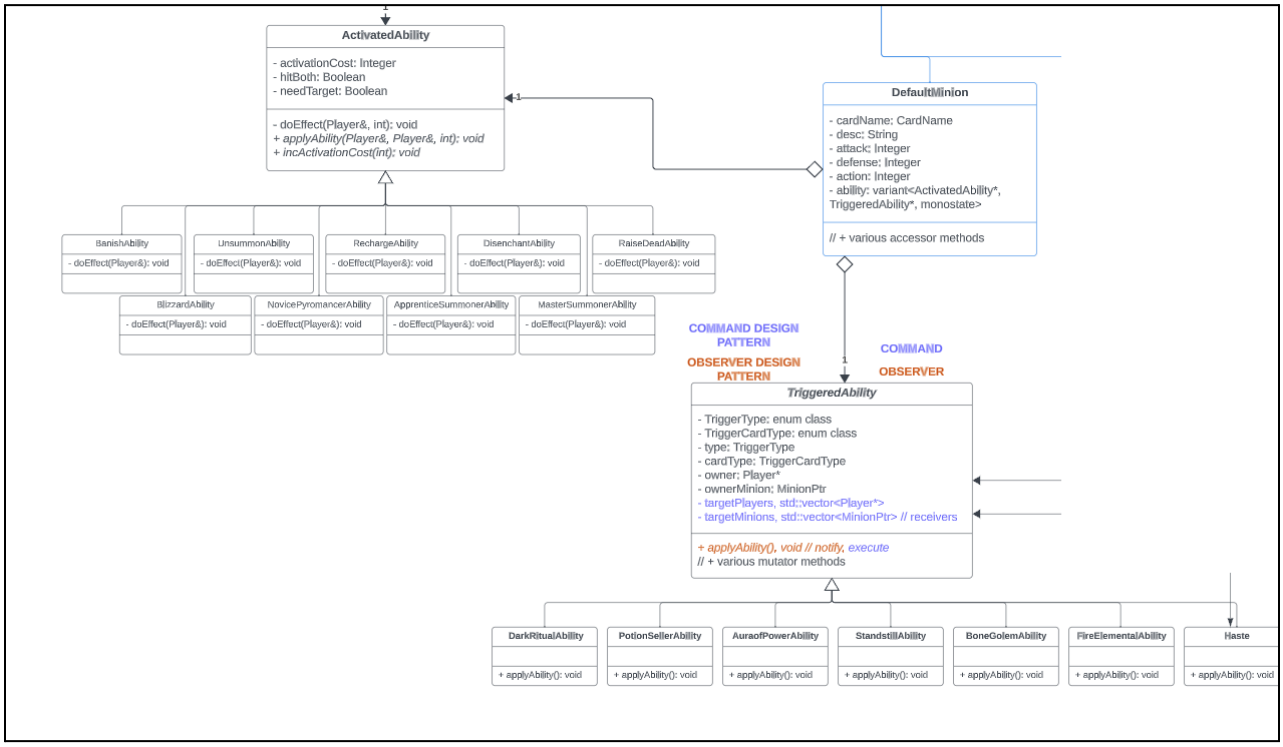
### 3.3 | **Changes to UML**

There was only one major change to our UML diagram between our initial and final submissions, which involves relationships between ActivatedAbilities/TriggeredAbilities and the other Card classes. Previously, a card with any ability owns an Ability object, which is the abstract base class for an underlying ActivatedAbility or TriggeredAbility (see below).




UML diagram for Abilities as of DD1

However as we began actually incorporating the Observer pattern, we realized how different ActivatedAbility and TriggeredAbility were. Intuitively, it made sense to classify both as an Ability, since that's how a user would perceive it. However, from an OOP standpoint, two classes who share little functionality should not share a parent. This led to a different arrangement (see below).




UML diagram for Abilities as of DD2

In our new UML diagram, ActivatedAbility and TriggeredAbility do not have a direct relationship. As illustrated in the image above, there are indeed no common functions. To mediate the fact that a minion can have a TriggeredAbility, an ActivatedAbility, or no ability at all, we used a variant field, since we could take advantage of its behavior as a type safe union type.

## 3.5 | **Interesting Code**

There are two interesting aspects of our design that we would like to highlight: removing the top Enchantment on a Minion, and using the ObserversList structure for TriggeredAbilities.

3.5.1 | *Removing the top Enchantment on a Minion*

Since we used the decorator pattern (EnchantmentDec as the decorator, DefaultMinion as the base, and Minion as the abstract class) to simulate playing an enchantment on a minion, we had a linked list of functionality. However, this requires other modifications to the Minion (i.e. being attacked or being the target of a Spell or Ritual) also be converted to an EnchantmentDec because order within the linked list matters. For example, if Bjarne's Earth Elemental is attacked by a +1/+1 Air Elemental, then Bjarne enchants it with Enrage, his resulting Earth Elemental's is +6/+8. However if it was first enchanted, then attacked, it would be +7/+8. We needed to preserve the order of events. This led us to create "hidden" enchantments to simulate changes to the Minion that are not official enchantments. Therefore, when we remove the top EnchantmentDec, it's not as simple as removing the first element in the linked list. This led to what we thought was an interesting combination of skills developed from CS135, CS136 and CS246 (see below).

```
void Board::stripTopEnchant(int i) {
    MinionPtr m = theBoard[i];
    if (DefaultMinionPtr dm = dynamic_pointer_cast<DefaultMinion>(m)) {
        throw no_enchantments(m);
    } else { // careful that EnchantmentDecs could be "hidden"
        EnchantmentDecPtr curr = dynamic_pointer_cast<EnchantmentDec>(m);
        EnchantmentDecPtr prev = curr;
        EnchantmentDecPtr ednext; // will be set if applicable
        MinionPtr next = curr->getNext();
        while (curr->isHidden()) { // while curr is not a legit Enchantment
            if (ednext = dynamic_pointer_cast<EnchantmentDec>(next)) { // if we have not hit base case
                prev = curr;
                curr = ednext;
                next = ednext->getNext();
            } else throw no_enchantments(m); // hit the base case
        }
        curr->setNext(nullptr);

        // if curr has a trigged ability, remove the observer
        TriggeredAbility* ta = curr->getEnchantmentAbility();
        if (ta) detach(ta);

        prev->setNext(next);
    }
}
```

*Code found at boardelements.cc:300*

3.5.2 | *Using the ObserversList structure for TriggeredAbilities*

```
struct observersList {
    Player* activePlayer;
    vector<TriggeredAbility*> observers;
};
```

*Code found at sorceryutil.h:19*

Due to the unique design of our program which uses two observer patterns, we found an obstacle where both the (Board class) and game (GameMaster class) share and modify the same list of observers, which are represented by a vector of TriggeredAbility pointers. Furthermore, both classes need access to the activePlayer, which only

GameMaster has access to (GameMaster “has-a” Player). To fix this issue, we came up with the solution of creating a structure (with public fields) that holds both the active player and the list of observers. This way, both the GameMaster and Board classes and we send both fields to any necessary methods simultaneously. This would reduce code redundancy, and errors while promoting code reuse.

# 4 | Resilience to Change and Object-Oriented Programming

## 4.1 | *Modularity*

From the initiation of our project, we designed our class structure to be modular - we made sure each module or component was small, and independent as much as possible.

For example, by implementing concrete subclasses for spell, ritual and minion abilities each under the Activated Ability and Triggered Ability superclasses, we are able to easily add more customizable spells, rituals, and minions to the game if needed. This makes our program scalable and extendable, thus more resilient to change. In another area, the subclasses under SorceryDisplay are easily extendable. By placing displays such as the current GraphicsDisplay and TextDisplay under a common “displays” vector, we are able to easily add a new display class to the SorceryController.

Another example of scalability is the use of Variants. In addition to storing Triggered Ability and Activated Ability in one variant type, other potential types of abilities can be added to the game with ease. This shows the multi-level capabilities of our code extensibility. Furthermore, by implementing separate classes for each ability and enchantment, it is possible to use any combination of abilities or enchantments during the game, allowing for numerous possibilities. Additionally, it is also possible to expand the board, or the hand by increasing the number of cards to any specified positive number. This would entail a simple change to vector sizes.

In all, by implementing modular code, we can replace, add or update one module without affecting the entire system. As well, by making our code more modular, it is easier to understand, maintain, and modify specific parts of the code without impacting the entire codebase.

## 4.2 | *Low Coupling*

To emphasize low coupling, one of our focuses was to minimize dependencies between different modules or components. Whenever possible, we utilized interfaces and abstract classes to define inheritance relationships and similarities between components. This ultimately reduces the impact of changes to the implementation of one component on others, making any change more feasible for the program. In our final project, we avoided friend classes and public fields. We also reuse modules in different contexts.

## 4.3 | *Encapsulation*

In Sorcery we encapsulate the internal details of each module and class by using private fields, and protected fields only when necessary. We use accessor and mutator methods to expose only the necessary interfaces. In all, by hiding the implementation details, we reduce the chances of unintended consequences when changes are made.

## 4.4 | *Abstraction*

In a similar effect to encapsulation, we were able to abstract away implementation details by focusing on defining clear, high-level interfaces. Changes to the underlying implementation can be made without affecting the code that relies on the abstractions.

For example, the Minion, and SorceryDisplay classes are two abstract classes in our program design. They combine similar features fields of all the minions and displays and effectively categorize them together. By doing this, we only expose what is necessary for the user or other components to interact with for all minions and displays. This also promotes code reusability since these components can be applied in different contexts without modification.

# 5 | Answers to Questions

## 5.1 | **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.

## 5.2 | **What design pattern would be ideal for implementing enchantments? Why?**

The decorator design pattern was ideal for implementing enchantments. Enchantments are modifications that can be played on Minions, such as modifying attack and defense values or changing 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 essentially treated 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.

## 5.3 | **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?**

The observer design pattern helped us to allow Minions to have any number and combination of abilities. If we replace the existing private variant field (named “ability”) 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 ObserversList structure in GameMaster (noting that ObserversList is a

structure that encapsulates a vector of TriggeredAbility pointers). 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.

### 5.4 | **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 enabled 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 a vector of SorceryDisplays, where SorceryDisplay is an abstract base class from which we inherit TextDisplay and GraphicsDisplay. If we wanted to add a third interface, we simply add another item to this vector, so that it would also be notified when necessary.

## 6 | Extra Credit Features

In our project we aspired and met the challenge of using STL containers and smart pointers to avoid memory leaks and handling memory explicitly. To do this, we used vectors instead of arrays and replaced raw pointers with shared pointers.

## 7 | Final Questions

### 7.1 | **What lessons did this project teach you about developing software in teams?**

As a whole, our group members were impressed – and even surprised – at how smoothly our team progressed. This complex and detailed project taught us how effective communication in a team can go a long way to avoid missteps and setbacks. Also, working as a team made us develop good habits related to documentation, research, Git proficiency, and establishing common goals and timelines.

#### 7.1.1 | *Communication*

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. 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.

Since we just started rigorously studying the OOP paradigm a few months ago, we frequently found ourselves asking ourselves if we were solving problems in an ideal “OOP” way. We often had to reality-check ourselves if there was a better way to reconcile OOP theories with the practicalities of our

project. For example, we found that some classes that are very deep in the inheritance tree need to make changes across the entire game. At first, we thought that an ActivatedAbility should have access to the entire game, however we realized that this would destroy the entire notion of encapsulation.

7.1.1 | *Discussions*
Although this course equipped us with all the tools needed to tackle this problem, we often found ourselves in a state of confusion and uncertainty about what decision to make. These were crucial moments because a misstep could result in a major problem a week down the road. In these moments, we learnt that discussing all the potentials as a team, and weighing the advantages and disadvantages of each option was helpful. When we couldn't agree on a solution, we reached out to professors during office hours to present our situation and potential solutions. This let us proceed with confidence and build a stronger foundation for more design decisions.

7.1.3 | *Source Control and Git*
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.

7.1.4 | *Common Goals and Timelines*
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.

7.1.5 | *Documentation*
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.

## 7.2 | **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 in order 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 player like the other spells. If we could have been more rigorous in our initial discussions, perhaps we could have caught this discrepancy earlier on.

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. Furthermore, it would be interesting to explore more user friendly and impressive graphic displays.

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

In summary, our project lifecycle – from conception to final implementation - has been a fulfilling learning opportunity. We successfully overcame obstacles, refined our strategy, and stayed committed to good design principles and productive teamwork.

Our updated UML class model and design document reflects the project's structure and any changes we have made. Our design places a strong emphasis on being resilient to changes, enabling both major and minor incorporation of modifications to the program specification.