CS246 FINAL PROJECT

SORCERY

Param Shah

Aadya Garg

Eric Zhu

Introduction

For our final project, we have developed "Sorcery," an intriguing card game inspired by renowned titles like "Hearthstone: Heroes of Warcraft" and "Magic: the Gathering." Set in a mystical realm, "Sorcery" immerses players in a strategic environment where they utilise a variety of cards—minions, spells, enchantments, and rituals—to duel opponents and ultimately reduce their life points to zero.

At its core, "Sorcery" is a game where players manage hands of cards and a board representing the battlefield. Gameplay involves drawing cards from a personal deck, playing cards onto the board, and engaging in battles with opponents' minions or directly attacking their life points. Each card has unique abilities and requirements, adding layers of strategy and decision-making to the game. Players must carefully manage their resources, represented by 'magic,' to deploy cards effectively and activate abilities.

The game is designed to be played in a terminal environment, with commands issued through standard input, making it accessible and straightforward. An optional graphical interface can enhance the visual experience for players who prefer a more immersive interaction.

"Sorcery" challenges students to implement complex game mechanics and handle various game states, providing a comprehensive learning experience in object-oriented programming and event-driven architecture. The goal is to craft a fully functional game by the end of the project period, demonstrating proficiency in software design and the practical application of programming concepts learned throughout the course. The last player standing, having successfully depleted the life points of all opponents, is declared the victor of "Sorcery."

Overview

The project comprises multiple abstract and concrete classes with different relationships such as composition, aggregation, inheritance, and dependencies.

1. Card: This is the most basic object identified by the name and cost. There are 4 types of cards (inherent from Card): Ritual, Spell, Minion, and Enchantment. Each card may also have a special ability that is used based on the card type. Additionally, each type has its own concrete classes (inheritance). For example, AuraOfPower, DarkRitual, and Standstill are Ritual Cards. Minions can be modified by Enchantments.
2. Player: Player: There are two players in the game. Each owns a hand and deck of cards and has access to the Board. They can perform various functionalities such as drawing a card, playing a card, using a minion ability, using mana, and gaining mana. Other methods include shuffling the deck, adding to the deck, restoring health, etc.
3. Board: The board is a central component. This is where players play their cards. It has 2 players, 2 graves, 2 rituals, and minions in play by both the players. It also manages the turn-based events such as triggering (or ending) abilities of cards at the start (or end) of the turn for the specified player and its opponent, and also handling other abilities as minions enter and leave. The board is displayed by Display which renders the board, hands, and minions.
4. Special Abilities: This is another class that implements all special abilities in the game (via static methods) such as those of bone golem, master summoner, apprentice summoner, and 13 more. These abilities modify the board (cards on the board).

Design

For our CS246 Spring 2024 Sorcery project, we embraced an object-oriented programming (OOP) approach to effectively manage the complexity of a card-based strategy game. This method allowed us to encapsulate data securely, inherit common functionalities, and utilise polymorphism to handle dynamic game interactions efficiently.

Encapsulation: We encapsulated player, card, and game state data within classes like `Player`, `Card`, and `Board`. Each class uses private or protected fields to safeguard data integrity and expose only necessary operations through public methods. For instance, the `Player` class encapsulates attributes such as `health`, `magic`, and collections like `hand` and `deck`, offering methods to manipulate these fields while preserving data consistency.

Inheritance: Our design heavily relies on inheritance to streamline code reuse and establish a clear class hierarchy, particularly evident in the `Card` subclasses. The abstract base class `Card` includes common fields like `name` and `cost` and a method `use_ability` for card actions. Specific card types like `Minion`, `Spell`, `Enchantment`, and `Ritual` inherit from `Card` and extend their functionalities. For example, `Minion` includes additional attributes for attack and defence and manages its abilities and interactions with other cards.

Polymorphism: Polymorphism is central to handling the diverse behaviours of cards during game play dynamically. By overriding the `use_ability` method in subclasses, each card type can implement specific strategies and interactions, such as spells affecting the game state or minions attacking other cards. This use of virtual methods and dynamic casting allows our game engine to interact with different card types uniformly, reducing the need for conditional checks and simplifying code maintenance.

Design Pattern: Initially, we considered several design patterns, such as Observer for game state notifications, Decorator for enhancing card abilities, and Factory for creating card instances with varied effects. However, the core implementation focused on simpler, more direct structures due to time constraints and complexity considerations. The design evolved to primarily employ a form of the Composite pattern in the `Board` class, where the board acts as a composite of various cards and players, managing the overall game flow and interactions. This approach allowed us to manage game states effectively without overly complex patterns that could obscure the project's educational objectives. We have also implemented core concepts of the Observer Pattern in parts of the game for example trigger events(game state change).

Input Validation: Robust input validation mechanisms are implemented to ensure the game operates correctly under various user interactions and game scenarios. This includes checks in the command processing logic to handle user inputs safely and maintain game integrity, ensuring actions like card plays, attacks, and ability uses are valid given the current game state.

In summary, our design leverages OOP principles to create a flexible, robust card game framework. This framework supports extending the game with new card types and rules, aligning with both educational goals and gameplay depth. By adhering to these design principles, we've constructed a project that not only meets the course requirements but also provides a solid foundation for further exploration and enhancement.

Resilience to Change

Our implementation is robust and resilient to changes due to the incorporation of several key software engineering principles and practices.

Our project is designed to be resilient to change through several key architectural decisions that enhance its adaptability and maintainability. Here's how the design of our Sorcery project accommodates and manages changes effectively:

1. Modular Design:

- **Component-based Structure:** Our codebase is structured in a way that segregates different functionalities into distinct classes and modules (e.g., `Player`, `Card`, `Minion`, `Spell`, etc.). This separation of concerns ensures that modifications in one part of the system (like adding a new type of card or modifying a gameplay rule) generally do not affect other unrelated parts, making the system easier to update and maintain.
- **Encapsulation:** By encapsulating the data and behaviours within these modules, we limit the impact of changes to the internals of these components. Other parts of the system interact with these components through well-defined interfaces, reducing the likelihood of ripple effects from changes.

2. Use of Inheritance and Polymorphism:

- **Extensible Class Hierarchy:** The inheritance structure used in defining cards allows for easy extension. For instance, introducing a new card type involves creating a new subclass that inherits from the `Card` class. This subclass can then override or extend the base class methods to provide new functionalities.
- **Polymorphism:** This allows the game engine to treat all types of cards generically where appropriate. For example, a general method to display card details can work across all card types, whereas specific actions can be customized per card type through overridden methods. Adding new behaviours to existing classes without altering the usage of these objects in the application ensures backward compatibility and reduces testing overhead.

3. Design Patterns:

- **Composite Pattern in `Board` Management:** This pattern, as used in managing the layout of the game (like the board containing players, cards, etc.), simplifies the addition of new elements into the game. Each component can be independently modified or extended without affecting the overall structure.
- **Factory Method for Card Creation:** Even though we simplified the initial design, the underlying idea to use a factory method for card creation could be easily integrated to manage the instantiation of different types of cards based on game progression or scenarios, enhancing the flexibility in game dynamics.

4. Robust Input Validation:

- Ensuring that all user inputs and file-based inputs are rigorously validated not only secures the game from erroneous or malicious inputs but also makes sure that changes in input formats or types can be incorporated and validated without impacting the core gameplay mechanics.

5. Configuration and Initialization Flexibility:

- Our approach to loading game configurations from files (like `default.deck`, `deck1.deck`, etc.) and initializing the game state allows for easy adjustments to the game setup without needing code changes. This externalization of configuration makes it easier to update the game or experiment with different game settings without altering the codebase.

By designing our project with these principles, we ensure that it is not only resilient to changes but can be easily extended and maintained. This resilience is crucial in

educational projects, where requirements can evolve based on educational goals or new insights into better teaching methodologies.

Answer to Questions

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

In our Sorcery project, we approached the design of activated abilities with a focus on maximizing code reuse and enhancing maintainability through several strategic implementations:

We abstracted common behaviours into the base `Card` class and relevant subclasses like `Minion` and `Spell`, where we defined methods for activating abilities as virtual. This design allows subclasses to override these methods to deliver specific behaviours while maintaining a uniform method signature, employing polymorphism to dynamically handle various card behaviours at runtime based on the actual object classes.

We encapsulated the logic for abilities within each card type capable of activated abilities, making each card class self-contained in managing its actions. For more complex abilities or those shared across multiple card types, we created separate classes or functions, allowing for significant code reuse and reducing redundancy.

In terms of modular design, we built ability components to be composable, where complex abilities that involve multiple effects are broken down into modular functions. This composability ensures that common functionalities like applying damage or altering stats are abstracted into reusable methods that can be invoked by any card's abilities, streamlining the codebase and facilitating easier updates and maintenance.

Additionally, we employed design patterns like the Strategy Pattern to manage diverse ability execution strategies across different cards and contexts, enhancing flexibility. The Command Pattern was also considered to encapsulate each action as a command object, which aids in executing, undoing, and logging actions—key aspects that can improve gameplay and debugging.

By externalizing ability definitions and parameters to configuration files, we adopted a data-driven design that allows us to introduce new abilities or adjust

existing ones without altering the core codebase. This not only makes our game engine more adaptable to changes but also simplifies testing and debugging by isolating each component for independent validation before full integration.

Through these strategies, we ensured that our design not only supports the current game requirements but is also robust enough to accommodate future expansions or modifications with minimal disruption to the existing code structure.

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

In our Sorcery project, we chose the **Decorator Pattern** to implement enchantments, a decision driven by the need for dynamic composability and the flexibility to modify card behaviours on the fly. Enchantments in a card game typically alter the properties or abilities of the cards, such as increasing attack power or adding special abilities. The Decorator Pattern allows these modifications to be layered onto cards dynamically, without changing the base structure of the objects themselves. This approach is particularly beneficial because it preserves the integrity of the original card objects, allowing for easy removal or alteration of enchantments without affecting the core characteristics of the cards.

This pattern also prevents class explosion, a common problem in object-oriented design where the number of classes can grow excessively due to the need for numerous combinations of behaviours. By using decorators, we can avoid this complexity and instead apply combinations of behaviours dynamically, enhancing maintainability and scalability. Each new enchantment introduced into the game does not require a new subclass of cards but instead can be treated as a new decorator, which can be wrapped around any card. This flexibility is crucial for a game's evolution, allowing us to introduce new gameplay features and enhancements without significant redesigns of the underlying system.

Furthermore, the Decorator Pattern supports the stacking of multiple enchantments on a single card by wrapping it with multiple decorators. This capability allows for the creation of complex and varied card behaviours, enriching the game's strategic depth. Overall, the choice of the Decorator Pattern has provided a robust framework for managing card enchantments, ensuring that our game is both adaptable to future expansions and straightforward to manage and maintain.

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?

In addressing the expansion of minion abilities in our Sorcery project to support any number and combination of activated and triggered abilities, the **Decorator Pattern** proves especially useful. This pattern enables us to dynamically attach additional responsibilities to objects without altering their underlying structures, making it ideal for our needs.

By applying the Decorator Pattern, each new ability is designed as a decorator that wraps around a minion object, enhancing or modifying its behaviour. For instance, if a minion needs a new attack ability or a triggered effect, these can be added as layers of decorators around the basic minion. This layering allows for an elegant stacking of multiple abilities on a single minion, where each layer adds its specific functionality while delegating other responsibilities to the base object or the next wrapper.

This design choice provides tremendous flexibility, as it allows us to mix and match abilities freely and modify them at runtime, which is crucial for a game where strategies and conditions can change rapidly. Furthermore, it simplifies maintenance and scalability because adding or removing an ability doesn't require changes to the minion’s core class or the creation of countless subclasses for each possible combination of abilities. Instead, we can develop a new decorator class for each unique ability and apply it as needed without disrupting existing code.

The Decorator Pattern also facilitates clean separation of concerns. Each decorator class focuses solely on the specific enhancement it provides, adhering to the Single Responsibility Principle. This approach not only makes our codebase more manageable and understandable but also enhances its testability and reduces potential bugs associated with complex inheritance hierarchies.

Ultimately, by using the Decorator Pattern, we maintain a robust and adaptable design that accommodates the evolving nature of our game, enabling us to introduce new gameplay features and enhancements seamlessly and without significant rework. This design strategy aligns perfectly with our goal to build a dynamic, expandable card game environment

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?**

Working as a team of three on the Sorcery project over a two-week period provided substantial insights into the dynamics of software development in a team setting, particularly highlighting the importance of communication, planning, and effective use of tools.

One of the foremost lessons was the critical role of clear and continuous communication. Regular meetings and updates were essential to keep the team aligned on project goals, progress, and to promptly address any roadblocks. We realized the importance of having clear role definitions from the start, which helped in distributing tasks according to individual strengths and in minimizing overlaps in responsibilities. This organization allowed for independent progress on different components of the project while maintaining coherence in the overall development effort.

The value of thorough planning and a modular design approach also became evident. We dedicated the initial phase of the project to detailed planning and designing the game's architecture. This preparation proved invaluable during the coding phase, making implementation smoother and more efficient. Utilizing design patterns such as the Decorator and Strategy patterns not only facilitated code reuse and scalability but also provided a structured way to discuss and resolve complex design challenges within the team.

We also learned crucial lessons about the importance of using version control systems effectively. Employing Git allowed us to work on different aspects of the project concurrently without interfering with each other’s work, enabling efficient integration of various components. This practice was complemented by maintaining good documentation, which served as a reference point for understanding each other’s code and for onboarding new features seamlessly.

Overall, the project was a practical reinforcement of the idea that while technical skills are essential, the ability to work well in a team, to communicate effectively,

and to plan and organize work systematically are equally important for the success of software development projects.

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

If we had the chance to start over on the Sorcery project, we would prioritize setting up a more robust issue-tracking and project management system from the beginning to better monitor tasks and deadlines. Additionally, we would allocate more time for initial architectural design discussions to further enhance modularity and ensure scalability. Incorporating automated testing early in the development process would also be a focus to catch issues earlier and streamline the integration of various components. Lastly, we would encourage more peer programming sessions to foster knowledge sharing and reduce the learning curve on complex parts of the project.