Design Document

Sorcery - Derek, Ken, Haokai

Introduction

Our Sorcery implementation aims to create a comprehensive two-player PvP card game inspired by popular collectible card games, and cover all of the features from the original project document, using a maintainable design.

Overview

The program consists of the following components:

Main Components:

- **main.cc** - Entry point of the program, parses command line arguments, creates MVC components.
- **narrator.cc** - Toggleable messaging system for `std::cout`
- **argexception.cc** - Custom exception handling

MVC Architecture:

- **gameModel/** - Manages game state, players, and core game logic.
- **view/** - Handles all display operations for multiple display platforms
- **controller/** - Manages interaction between gameModel/ and view/

Card System:

- **cards/** - Contains Card hierarchy and related classes
- **cards/abilities/** - Contains Ability system and Observer pattern implementation

Class Breakdown

Game Model Directory (gameModel/)

Game (game.h/cc) The `Game` object holds the data necessary for a Sorcery game. It holds the following fields:

- Vector of pointers to `Players` (2)
- `activePlayer` index indicating the current turn
- `winningPlayer` indicating the winner when the game ends

It contains the key methods to `startTurn()` and `endTurn()` as well as global events such as `minionBattle()`.

Player (player.h/cc) Each player instance represents one game participant and contains:

- Data: Name, life, magic
- Pointer to `Hand`, `Deck`, `Graveyard`, `Ritual`, and `Board` objects

It contains all gameplay-related methods like `playCard()`, `activateCard()`, and `minionAttack()` to name a few and manages their `Card` collections listed above.

Board (board.h/cc) Manages the vector of active Minions of a player:

- Contains a vector of `Minion` pointers representing Minions on the field
- Handles minion placement and removal

Hand (hand.h/cc) Manages a player's hand of cards:
- Contains a vector `Card` pointers
- Handles adding and discarding Cards

Deck(deck.h/cc) Manages a player's deck of cards:
- Contains a vector of `Card` pointers
- Handles reading in Cards from a file, shuffling, and popping from `Deck`

Graveyard(graveyard.h/cc) Manages a `Player`'s fallen `Minions`:
- Contains vector of pointers to `Minions`
- Handles card retrieval for `RaiseFromDead` spell

Controller Directory (controller/)

Controller (controller.h/cc) Manages the MVC interaction and contains:
- shared_ptr to `Game` object
- shared_ptr to `Viewer` object
- Game-related Methods: `play()`, `use()`, `printBoard()` etc.

View Directory (view/)

Viewer (viewer.h/cc) An abstract class that manages all visual output and contains:
- shared_ptr to `Game` object for state access
- Vector of `sorcDisplay` pointers for multiple output methods
- Key Method: `display(Command)` - Calls print functions on all active displays

sorcDisplay (sorcDisplay.h) Abstract base class for polymorphic display handling
- Derived classes: `TextDisplay`, `GraphicsDisplay`

TextDisplay (textdisplay.h/cc) Implements ASCII card representation of game state
- Contains `ostream` for text output
- Calls (ascii_graphics.h/cc) methods, containing ASCII card templates

GraphicsDisplay (graphicsdisplay.h/cc) Implements X11 graphical representation
- Owns `Window` object with dimensions

Window (window.h/cc) X11 window management and drawing operations

Cards Directory (cards/)

Card (card.h/cc) Abstract base class for all card types:
- Contains: card name, card type, description, magic cost, player index of owner
- Raw pointer to Game object for state access and to exert boardwide effects
- Derived classes: `Minion`, `Ritual`, `Spell`, `Enchantment` (derived from `Minion`)

Minion (minion.h/cc) Represents cards that can attack and use abilities, and contains:
- Attack and defense statistics, available actions counter
- shared_ptr to `TriggeredAbility`
- shared_ptr to `ActivatedAbility`
- Key Methods:
    - `attack()` - Overloaded for different target types if actions permits
    - `activate()` - Uses activated ability if available magic permits

Enchantment (enchantment.h/cc) Implements Decorator pattern to modify minions:

- Inherits from `Minion` class
- shared_ptr to `Minion` as component being decorated
- Modification values for attack, defense, actions, and `ActivatedAbility` if provided
- Key Methods:
    - `attach()` - Applies enchantment to target minion
    - Overridden `Minion` methods that provide component with modifications

Ritual (ritual.h/cc) Persistent effect cards:

- shared_ptr to `TriggeredAbility`
- Activation cost and charges left counter
- Behavior: Expends charges when triggered ability activates

Spell (spell.h/cc) One-time effect cards:

- activate() - Overloaded for different targeting options
- Behavior: Removed from Hand (destroyed) after use

Abilities Directory (cards/abilities/)

Trigger (trigger.h) Implements subjects in Observer pattern:

- Contains enum of `TriggerType`: TurnStart, MinionExit, etc.
- Accessed by `Game` and `TriggeredAbility`

Observer (observer.h/cc) Abstract interface for ability notification:

- Pure virtual `notify()` method

Ability (ability.h) Abstract base class for all abilities:

- Description field and `Card` pointer
- Pure virtual `activate()` method
- Derived classes: `ActivatedAbility`, `TriggeredAbility`

ActivatedAbility (activatedability.h/cc) Abilities that can be manually triggered:

- Additional cost field beyond base `Ability`
- Implements `activate()` with cost checking and resource validation

TriggeredAbility (triggeredability.h/cc) Abilities that respond to game events:

- Inherits from both `Ability` and `Observer`
- Contains `Trigger::TriggerType`
- Implements both `activate()` and `notify()`
- Integrates with `Trigger` system for automatic activation on event

***Note:** Each of the derived classes of `Card` and `Ability` have their own concrete subclasses that we will not be covering individually.*
In general, they override the virtual methods of the derived class, providing their own effect on the game. We instantiate them when loading `Deck` and provide a raw pointer reference to `Game`, in some cases (`Ability`), and the player index.

Design

Game Progression

The program begins in `main.cpp`, where the command line arguments are processed. The user is first prompted to enter in Player names if no `-init` file is provided, then the `Deck`, `Player`, and `Game` objects are created and initialized respectively. The game loop is then started which continuously calls `Controller` functions when commands are parsed from `std::cin` or the `-init` file.

For example, when a player ends their turn, the `Controller` will call `Game` to handle events such as changing activePlayer, incrementing magic, and notifying the corresponding `Trigger::TriggerType`. When a player either dies or quits, the program will exit.

Game Model

A Game owns a vector of shared_ptr to `Players`, and other fields to keep track of the active player. It has the `startTurn()`, `endTurn()` methods to manage the game loop within the model. We decided to have it include other global methods to facilitate interaction among Players such as `notifyTrigger()`, which iterates through attached `Observers` (abilities) and calls their `notify()` in APNAP (active player: minions→ritual, then inactive player minions→ritual) order.

A `Player` has the fields name, life, magic, and owns a `Hand` object with a vector of `Card` pointers, a `Deck` object with a vector of pointers of `Card` as well, a `Graveyard` with a vector of `Minion` pointers, and a `Board` object with a vector of `Minion` pointers and a `Ritual` pointer representing the minions and ritual in play. It has methods to play, use, command cards, and when testing mode is enabled, draw cards from the deck, and discard from the hand. We designed the ownership structure so it creates a clear separation where each game zone manages and encapsulates its own collection of cards, without `Player` class needing to know any internal details.

A `Card` owns a name, a description, a magic cost, a type to facilitate downcasting, a pointer to its `Game` object and the player number of its owner. Now, we will explain our design and solution for cards.

Card System

We designed `Card` to be an abstract base class allowing a player's `Hand` and `Deck` to contain `Cards` of different types. A `Card` can either be a `Minion`, an `Enchantment`, a `Ritual` or a `Spell`.

`Minion` inherits from `Card`, and owns an attack and defence stat, its number of actions, a `TriggeredAbility` pointer and an `ActivatedAbility` pointer. It has the methods attack() and activate(), to attack a target and activate its `ActivatedAbility` if it has one, both overloaded for different targets. The `attack()` method either directly reduces the opposite player's life, or fights another minion by calling its game's

battleMinion method. The `activate()` method simply calls its `ActivatedAbility`'s `activate()` method.

We designed the `Enchantment` class using the **Decorator pattern** that inherits from `Minion`, and owns a `Minion` pointer as the component. It has `attach()` to put the enchantment on the minion and replace its position on the board, and overriding methods from the Minion class that usually call the corresponding methods of its Minion component, but sometimes with modifications. Its own `atk`, `def`, `actions` fields also serve as the modifications for its `Minion` component, leaving the component as untouched as possible. The decorator pattern allows us to easily add and remove enchantments on minions as well as provide a clear separation between the `Enchantment` and the `Minion` itself.

`Ritual` inherits from `Card`, and also owns a `TriggeredAbility` pointer, an activationCost, and charges. If the `TriggeredAbility` is triggered, charges will be expended, or the ability will prevent itself from activating.

`Spell` inherits from `Card`. It has the `expend()` method, overloaded for different targets or no target at all, which uses its effect. After a `Spell` uses its effect, it is removed from the game.

We decided that unique cards can then be implemented through concrete classes that inherit from the different card type classes, with their own virtual methods. This allows for easier implementation of new cards.

Abilities and Observer Pattern

`Observer` is an abstract class, and has the `notify()` method. This is called by `Game`'s `notifyTrigger()`, with a certain trigger type passed in as the parameter. We decided that the observer pattern allows us to easily attach new observers to game events and is optimal for TriggeredAbilities.

An `Ability` owns a description. We made it an abstract base class and have a virtual method activate(). Abilities can then be easily added and removed from Cards.

- ActivatedAbility inherits from Ability and also has a cost and a Game pointer. Its activate() method activates a specific effect if the cost can be fulfilled.
- TriggeredAbility inherits from both Ability and Observer. It overrides the `notify()` methods. TriggeredAbility can be notified by Game with a specific trigger type which it will then activate itself. If its card is a Ritual, it will only activate if charges can be expended.
- Unique Abilities are then implemented through concrete classes of either ActivatedAbility or TriggeredAbility.

Resilience to Change

Through object-oriented design principles, the program is able to accommodate changes in the design specification easily.

In most classes, fields are encapsulated with either private or protected, and can only be accessed or mutated through public methods. This allows us to easily modify what happens when they are called. For example, when a `Minion`'s `takeDamage()` method is called, reducing its `def` by a specified amount, it can also call `Game`'s `notifyTrigger()` when it dies.

Model

The `Game` class manages global state and turn logic with minimal assumptions about card behavior or player logic. It uses a `notifyTrigger` system through the Observer pattern to handle triggered abilities, easily allowing new trigger events to be added in. `Player` and minion interactions (e.g., combat, death) are centralized, isolating game rules for easier updates. This structure supports future expansion (e.g., new observers/triggeredAbilities, card types, or possibly multiplayer support) with minimal code changes.

Card collections of `Player`, such as `Hand`, `Deck`, `Board`, and `Graveyard` are abstracted via shared_ptr, allowing changes to underlying mechanics (e.g., hand size, draw logic) without modifying the player interface. This allows for each of these classes to work together in `Player` with high cohesion in their specific task. Additionally, method overloading for playCard and activateCard supports multiple ways of interaction (e.g., testingEnabled, targetting).

Cards

The Card class defines the common structure and interface for all card types in the game. By declaring methods like `getName()`, `getDesc()`, and `getCost()` as pure virtual, we enforced consistency across its subclasses (`Minion`, `Spell`, `Ritual`, `Enchantment`).

This design also ensures that any card can be processed generically, making it easier to modify or extend card-handling logic without needing to know the specific type. As a result, we were able to use `sorcDisplay` functions while building the game. Therefore, when a new concrete card is introduced, only a new subclass is needed; no changes are required in code that already uses the Card interface. New card types can also easily be implemented as subclasses of `Card`, with their own concrete subclasses.

`Card` subclasses and concrete subclasses override their superclass's virtual methods, allowing them to provide their own unique implementation while avoiding heavily modifying other code to account for them.

For example, when implementing the decorator pattern in Enchantment to allow for mixing and matching modifications, we did not need to change any existing code of `Card`. `Enchantment`, and its concrete subclasses, simply inherited fields and methods of `Card`. Any new card types we may want to add can be done so with minimal change to our existing structure.

Abilities are separated from their Minions and Rituals, allowing them to be easily added and removed from any of them. Concrete abilities override the same methods, so they can be generically used by any `Card`, allowing easy implementation of new Abilities.

Controller/Game Loop

The controller provides a straightforward interface for adding new commands. Each command parsed from the user is directly mapped to a specific game action, delegating it to either the `Game` or `Player`. This allows for decoupling of the main class, view, and model.

Viewer

The `sorcDisplay` hierarchy was designed with the ability to easily add new display platforms. Viewer holds a vector of `sorcDisplays` and calls each of the generic `sorcDisplay's` functions. Every `sorcDisplay` is able to override the virtual function with their own way of presenting the game, whether it is text or graphics as we have done.

Constants for window dimensions, colours, text standards and card templates facilitate easy changes to how different displays represent the game. The functions in TextDisplay use vectors to build output, allowing for more ASCII art to be added if needed and displayed with minimal changes.

Answers to Questions

Question 1

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

Response

We can design activated abilities by making an abstract Ability base class. Then, we can then create an abstract class ActivatedAbilities that inherits Ability and adds cost management functionality. Specific activatable abilities such as "Deal X damage" or "Draw a card" inherit from the ActivatedAbilities abstract class and override the activate() method with their unique effects.

Each minion maintains an ActivatedAbility unique pointer that points to the desired appropriate concrete ActivatedAbilities class. Our approach uses polymorphism to call activate on any ActivatedAbility type, sharing common cost-checking logic in the base class. This also allows us to easily add new abilities without modifying existing code, as well as maintain type safety through inheritance.

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

Response

The Decorator design pattern is ideal for implementing enchantments because it allows us to dynamically add behaviour (enchantments) to minions without changing the base class structure. An Enchantment class can wrap around a Minion object and override its Minion's methods with its own. Multiple enchantments can be stacked on top of each other by wrapping one decorator with another and vice versa. Thus, the Decorator approach allows for clean separation between the base Minion functionality and Enchantments.

Question 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 maximising code reuse?

Response

To support multiple activated and triggered abilities, we can use the Factory Method design pattern. Specifically, we can create an Ability abstract class which encapsulates the shared properties like `description` and a virtual `activate()`, and two subclasses that inherit it: `ActivatedAbility` and `TriggeredAbility`. Then, derive concrete subclasses like `DealDamage` and `OnStartGainMagic` from them as actual abilities. Each concrete ability can override the activate() method and fields with their own appropriate logic.

A minion can then own a `vector<shared_ptr<Ability>>` to store all of its abilities, both activated and triggered, enabling it to iterate through and handle them polymorphically. This approach allows us to uniformly handle abilities through a shared interface.

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

Response

To support multiple user interfaces, we can introduce an abstract base class `sorcDisplay` that defines a unified interface for rendering `Game` elements. This can be inherited by `TextDisplay`, `GraphicsDisplay` or even other classes. Clients are able to call the same `sorcDisplay` methods for any `sorcDisplay`, although each `sorcDisplay` implementation will be different to account for their own display needs.

Extra Features:

Automatic Memory Management:

We implemented the entire program with RAII principles taken into account, and achieved memory management exclusively through the use of smart pointers. During implementation, we encountered the issue where a chain of smart pointer objects formed a cycle (Player → Board → Minion → Game → Player) and caused memory leaks. To solve this issue, we instead passed arguments of `shared_ptr<Game>` as a raw pointer to `Game`. The decision to make the change on `Game` was due to the fact that `Game` is fundamentally distinct from the objects it manages, and therefore should not be owned by them.

Invalid Input and Exception Handling:

In the event of a user providing invalid input (using the wrong command, index out of bounds, etc), our program is designed to prevent any changes from being made to the game model while still allowing the user to continue on. Errors will throw a custom exception and a descriptive error message is printed to `std::cerr`. This was tricky for us as there were a lot of cases to account for. We also took into account handling exceptions our STL dependencies might throw. For example, exiting the game when memory can not be allocated and invalid stoi calls.

Story Narration (-narrate):

**This feature is fully toggleable using command line arguments.*

When storytelling mode is enabled, a narrator will somewhat-dramatically describe any events happening in the game while being played. This calls static functions in the Narrator class, which holds a boolean, and prints to `std::cout`. Our original approach was to pass a boolean, but we quickly figured that making a class with static methods was much more reusable.

Typewriter Effect (-delay):

**This feature is fully toggleable using command line arguments.*

When the typewriter effect is enabled, game narration will print out individual characters with periodic calls to the STL thread's sleep function. Any calls to print the board and hand will also have a scroll-like effect using the same technique.

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?

Through this project, we were able to see a lot of the design patterns covered in class in action. Although tedious at first, design patterns allowed us to make much more reusable code. We gained experience in how to tackle a large project by breaking it into manageable components. At the start, we collectively designed our UML diagram and delegated specific components to each other, splitting work evenly. We had set goals on what we wanted to accomplish by certain dates.

We also learned how to collaborate effectively and efficiently in a large project through Git version control techniques, as many of the components of the project were self-contained. Regularly reviewing each others' code helped maintain consistency and caught integration issues before they became major problems.

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

If we had a chance to start over, we would have extracted the `Card` collections from the `Player` class in the design stage. These collections resulted in much more related functions than expected and this caused some headache during implementation as we had to refactor a lot of function calls.

We also would like to extract out the main game loop, calls to `Controller`, and some of the input parsing from `main.cc` to another class, possibly named `GameSession`, if we had more time. This would better show the flow of the program and clean up the class.

We also agreed that checking for memory leaks earlier and more frequently in the implementation would have been a good idea. We had run Valgrind and discovered a leak quite late into the project and it was stressful and difficult due to the size and complexity of the program.

Design Changes from DD1 to DD2:

Our final implemented project slightly differed from the original plan as we ended up moving some of the functions in `Trigger` to be in `Game`, in order to properly notify in APNAP order. `Card` collection classes of `Player` were originally planned as vectors, but we decided that abstracting the functions would make for a cleaner interface.

Conclusion

We have provided test examples in `test-files/`

To run:

```
./sorcery $(cat test-files/{filestem}.args)
```

Overall, this has been a difficult but fun project. We have all gained a lot of experience in designing programs and implementation. Being able to apply a lot of what we have learned throughout the course was really useful going forwards. We hope you enjoy our work for this Sorcery implementation! Thank you.