University of Waterloo

CS247, Spring 2025

Project, Final Report

Vincent Chung, Ryan Nguyen, Jeff Mak

Overview

This project is structured around a couple major class hierarchies:

Card:
An abstract class that represents any card type in the game. It has 4 subclasses: Minion, Spell, Enchantment, and Ritual.

Minion:
An abstract class that represents the minion card type. These cards have an attack and defense member field, and some have either activated abilities or triggered abilities. It has 3 intermediate, abstract subclasses: BaseMinion, ActivatedMinion, and TriggeredMinion, representing normal minions, minions with activated abilities, and minions with triggered abilities, respectively, and each minion type is represented under one of these three subclasses.

Spell:
An abstract class that represents the spell card type. Each of these cards have an action function that applies a one time effect and then gets destroyed. Its subclasses are the specific spells that each player can play throughout the game.

Enchantment:
An abstract class that represents the enchantment card type. Each of these cards have an apply function that takes in a Minion parameter, which the enchantment will either modify the stats or ability of. The enchantment class is implemented using the decorator design pattern because minions can be enchanted by multiple enchantments, applied to in oldest to newest order, via the EnchantmentDecorator class. This decorator class gives each enchantment an Enchantment* next field which is updated every time a new enchantment is applied to a minion, so that the enchantments can be applied in the correct order.

Ritual:
An abstract class that represents the ritual card type. These cards have an activation cost, a number of charges, and a trigger function for its triggered ability. The ritual can only take effect if there are enough charges left in the ritual. Every time this ability activates, the trigger function is called to do an effect and the number of charges decreases by the activation cost. Its subclasses are the specific rituals that each player can play throughout the game.

Player:
This class stores the information on the player, like their name, life, and magic. It also stores information on the player’s board (the active minion cards that they’ve played), hand (the cards that they can play), graveyard (the minions that have died), and the current active ritual. Additionally,

this class contains functions for gameplay actions like playing cards, drawing cards, and modifying its life member variable when damaged.

Game:
This class manages the turn based structure of the game and coordinates interactions between the players. It also implements the trigger system using the observer design pattern with the TriggerType enum (EndTurnPlayer1, EndTurnPlayer2, StartTurnPlayer1, StartTurnPlayer2, MinionLeave, MinionEnter) and the TriggerTopic class, to notify cards that have trigger based effects. Furthermore, this class gives the user functionality to enter commands to complete different actions and also controls when a game is won.

Notification:
This class is used by the trigger system to send event data through the trigger system. Its subclasses, TurnChangeNotification and MinionNotification, specify whether a player has started their turn or whether a minion has entered/left play, respectively.

Design

Key Techniques and Patterns

Observer Pattern for Triggers: The game publishes events (TriggerType), and rituals/minions with triggered abilities subscribe to corresponding TriggerTopics. We pass typed Notifications (TurnChangeNotification, MinionNotification) so observers can react without querying global state. This decouples event producers (Game/Player) from consumers (cards) and simplifies adding new triggers.

Decorator Pattern for Enchantments: Each minion optionally owns a head enchantment. Enchantments form a singly-linked chain; effective stats/abilities are computed by folding from base → newest. This avoids subclass explosion (no "Minion+X+Y+Z" classes) and supports dynamic add/remove.

Command/Strategy for Abilities (internal structure): Activated abilities delegate to small, reusable Effect helpers (deal damage, grant stats, summon). The "what to do" is an effect; the "who/when" is governed by a Targeting and Cost policy. This separation enables code reuse and makes abilities easy to test.

Factory/Registry for Card Creation: Deck loading uses a name→constructor registry to create unique_ptr<Card> (prefer std::make_unique<T>(...) over raw new). This eliminates long if/else chains and localizes the cost of adding new cards to a single registry entry.

RAII & Ownership Semantics: std::unique_ptr<Card> models single ownership of cards by areas (deck/hand/board/graveyard). Movement between areas transfers ownership explicitly, preventing leaks and double-frees. Triggers detach on destruction to avoid dangling subscribers.

Explicit Status Codes: Operations that can fail (e.g., useSkill) return UseSkillStatus. This makes error handling explicit at call sites (UI messaging, magic refund policies, action counters) and keeps control flow clear.

Invariants & Validation: We enforce constraints like max hand size, board capacity, and non-negative life/magic as preconditions/postconditions on mutating functions, reducing defensive checks across the codebase. However, there are still many defensive checks that throw runtime errors throughout the system to ensure explicit error handling and that bugs do not go unnoticed.

Constants & Configuration: Tunable values (hand size, board size, default life/magic, ritual charge costs) are centralized constants (helper.h), supporting changing the stats of cards without invasive changes. There are very few “magic numbers” throughout the system. All constants have a name that makes it clear about its purpose.

Model View Controller Pattern: Model-View-Controller (MVC) guided how we separated responsibilities among Game (Model), CliDisplay (View), and Controller (Controller). The Model owns all domain state—players, cards, board, triggers—and exposes intent-level operations (play card, attack, end turn) that are UI-agnostic. The View renders immutable snapshots of this state to the terminal (enforced by accessing the Model’s information via const references from const functions). The Controller translates user commands into validated Model calls: it parses input, resolves targets, checks preconditions, and maps failures to clear messages without bleeding rule logic into the UI. This separation makes the CLI easily swappable for a GUI/web front-end. It also enables features like command logging and potential undo/redo, because the Controller issues discrete actions that the Model executes atomically and reports via events for reliable auditing.

Comparison to Original Design

In comparison to our original design, our final design has a few changes implemented. For the triggers, we added a passing around of two types of notifications, TurnChangeNotifications and MinionNotifications. The former would be issued when the player turns change, while the latter would be issued when minions enter or leave the playing board. Due to this, there will be sufficient information for each ritual or minion with triggered abilities to know whether a minion has entered or left the board or which player’s turn has just finished.

Next, for minions, we added 3 intermediate subclasses so that there is a better categorization for minions. The first intermediate subclass is the BaseMinion, which is for minions without any abilities like the Air Elemental and the Earth Elemental. The second is the ActivatedMinion, which is for minions with activated abilities like the Novice Pyromancer and the Master Summoner. This class adds an additional activation cost member variable and a different signature to the useSkill function, returning a UseSkillStatus enum (OK, NoAction, Silenced, NotEnoughMagic) instead of not returning a value like the other minion classes. Finally, the last intermediate subclass is the TriggeredMinion, which is for minions with a triggered ability like the Bone Golem and Fire Elemental. This class allows minions to own an additional trigger member variable.

Resilience to Change

Our design emphasizes the Open/Closed Principle. We can add new content with minimal changes to existing code.

New Cards: Add a subclass (or reuse existing ability/effect objects), then register it in the factory. No engine changes required.

New Abilities: Compose new Effects/Targeting policies; most cards can reuse these building blocks without new hierarchies.

New Triggers: Add a TriggerType, a Notification payload struct if needed, and a TriggerTopic entry. Consumers subscribe as needed; producers emit one new event.

Balance/Rules Tweaks: Hand/board limits, life/magic defaults, action counts, and ritual economics live in constants.

UI Evolution: Since command parsing and game logic are separated, a richer UI can reuse the same Game/Player/Card APIs.

Testing: Effects and targeting logic are unit-testable in isolation. Trigger delivery can be tested via injected notifications.

Answers to Questions

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

We maximized code reuse by creating helper functions that the abilities are able to dependency inject into to specify their individual effects. The effect implementations are thus shared amongst the abilities. This was possible because abilities can be grouped into one of several effect categories: dealing damage to other minions, gaining attack/defence (i.e. stats) and summoning new minions. Therefore, we have the functions attackMinion, gainStats and summonMinion in order to reuse the ability implementations.

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

The decorator design pattern is ideal for implementing enchantments. The primary reason is the requirement that enchantments need to be applied to minions dynamically at runtime with the property that enchantments are applied from oldest to newest. Due to this requirement, the decorator design pattern is ideal.

Specifically, each minion can optionally have an enchantment, with each enchantment having an Enchantment* next field. Every time a minion receives a new enchantment, the linked list will get updated so that the minion receives the new effects as well as the old effects in the correct order. This enables us to maintain the order of enchantments and dynamically add new ones.

4.4.2 Minions: 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?

This can be solved using the command pattern. We wish to store an arbitrary number of abilities, in which we should allow the minion to cast any ability, so long the trigger condition has been met, or there is enough magic to cast the activated abilities. Hence, an invoker can be used that is attached to each minion in order to allow the minions to cast any abilities they wish to.

Extra Credit Features

One thing that we have done, per the project guidelines, is to complete the entire project without any memory leaks. This is done by having unique pointers to initialize objects in our program, via std::make_unique. We also use unique pointers to indicate ownership, so if A owns-a B, then we would have A owning a unique pointer to B, and B would usually hold a raw pointer to A since B does not hold ownership to A. Hence, we did not have to do any "new" memory allocations or manual "delete"'s in our entire program. The containers that we have for holding multiple objects are mostly using std::vector, and sometimes using std::array, instead of C-style array. This is for better error reporting and prevents access to unmapped memory addresses.

Another extra credit feature that we have is a new type of minion, called the cloner. It has default attack 0, defence 2, card cost 3, and an activated ability which costs 3 magic. It allows the cloner to clone at most 2 minions of the same type as any minion on the board at that moment, whether it is on the cloner's side or the opponent's side, onto the cloner's side. Note that it will keep the default attack and defence stats of the minion cloned. Since this minion has very low defence, it is fragile, but it provides a high-risk high-reward playstyle as it allows cloning of any type of minion on the board, but at the expensive card and activation cost of 3 magic each.

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?

Interfaces First: Freezing card, ability, and trigger interfaces early reduced merge conflicts and let us parallelize work safely.

Small Patterns, Big Payoff: Observer + Decorator + Command covered most variability points; resisting ad-hoc flags kept code malleable.

Ownership Matters: Standardizing on unique_ptr and RAII made object lifetimes predictable and eliminated a class of bugs.

Status over Exceptions for Flow: Returning UseSkillStatus from abilities made the UI and rule enforcement clearer than exception-driven control flow.

Testing in Layers: We unit-tested Effect helpers and targeting separately, then exercised full scenarios via scripted command sequences.

Communication & Reviews: Short, frequent PRs with code reviews caught subtle invariants (e.g., when to decrement actions vs. magic) before they spread.

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

Adopt the Factory/Registry on Day 1: Would have avoided early if/else chains in deck loading and simplified adding cards.

Data‑Driven Cards: Move more card parameters to JSON (costs, stats, effect wiring). The engine already supports it; we'd push further to reduce C++ churn.

Earlier Notification Design: We'd standardize notification payloads sooner (we converged on TurnChangeNotification/MinionNotification mid‑project).

Property‑Based Tests: Add fuzz/property tests for target selection and trigger delivery to catch edge cases (empty boards, full boards, lethal damage races).

Consistent Terminology: Enforce one spelling ("defence") and consistent names for areas and events to reduce confusion.

Metrics & Logging: Add structured logs (card id, event id, timestamps) from the start to accelerate debugging complex trigger chains.