Introduction

Group Members: Arteen Mirzaei and Avi Dave
Project Choice: Sorcery

This document will contain a high-level overview and analysis of our CS 246 final project, sorcery. It will go over the purpose of each class and file (Overview), the design choices that serve this purpose and why they were made (Design), the resilience to change of each of these design choices (Resilience to Change), answers to questions posed in the project specifications (Answers to Questions), answers to the questions posed in the project guideline document (Final Questions), and a final summary of the document (Conclusion).

Overview

Main.cc

Main.cc does not contain any class. Its purpose is to parse the command line arguments (such as -testing, -init, -deck1 and -deck2) and to initiate the controller class, which will handle the games input and actions based on that input. Main.cc also creates the players and calls for the decks to be initiated, but all that information is passed to the controller right after. Finally, Main.cc will output the winner if one is received.

Controller

The controller class is the class that handles all user input (either from the cin or a file stream) and carries out the consequences of these commands. It holds the owner (player) classes for both players, the view class and the triggers class. Each of these are essential for turn-to-turn operations.

The controller class works by switching between an “active” and “non_active” player, which point to player 1 and player 2, depending on which player is taking their turn. All operations during the turn are done with respect to the active and non_active player. Since the controller class handles all actions and operations that could be taken during a players turn, including trigger activation, it is the most important class.

View, TextView and GraphicalView

The view class is an abstract observer class to implement the TextView and GraphicalView classes. When notified, it’ll call to the desired view (based on the -graphical CLA) and output to it.

Trigger

Each trigger is assigned a different number, denoting what kind of trigger it is. This is so minions and rituals on the board can be assigned a trigger and be observers of those triggers. Trigger is itself a subject for those observers.

Owner

The Owner class is responsible for managing its own collections, such as the board, graveyard, and deck. It also contains the memory for all the individual cards. The individual cards are stored in unique pointers, which are owned by the Owner class. The Owner class (almost) exclusively interacts with the controller class, which is the key to gameplay.

Collection

The abstract class for all the collection types, such as the hand, deck, board, and graveyard. Contains a vector of pointers to cards, since collections don't inherently own the cards, the owner does.

Deck

A concrete implementation of collection, which starts full of pointers to all the cards owned by Owner and can be drawn to the hand.

Hand

A concrete implementation of collection, which can draw from the deck and has a max capacity of 5 cards. If the hand is full, a card won't be drawn.

Board

A concrete implementation of collection, which holds up to five minions and one ritual. Owned by the Owner class.

Graveyard

A concrete implementation of collection, which can hold any number of minions (and only minions).

Card

An abstract class which holds most of the card data that can be accessed and passed around. Superclass to Spells, Minions, Enchantments and Rituals. Most operations are done with this superclass.

Spells

A concrete implementation of cards which holds basic card data and ability data for using abilities.

Minions

A concrete implementation of cards which holds all relevant minion data (including triggers and abilities).

EnchantmentDec

An abstract decorator which inherits from Card to apply to minions and modify their behaviors.

Enchantment

A concrete implementation of the enchantment decorator to enhance and modify minions.

Ascii_graphics

Allows for use of cards/hand/board display with proper formatting.

Window

Draw the hand and board into a window instead of the terminal to meet the graphics requirements.

Design

Controller

Controller was designed to allow for all input to be handled in one class. Controller is only responsible for the game state after every action, not for anything else. To allow for the game state to be shared with the view class (to allow for display), Controller inherits from Subject and view inherits from observer. The view class is responsible for displaying UI elements, such as the hand of the active player and the board. The controller class notifies the view class when those commands are called, so the view class can display the proper information. This design pattern was chosen to help uphold the single responsibility principle. To continue to uphold this principle, almost all game state changes are done through owners and cards. Controller carries out the actions of cards and applies to effects to the appropriate owner. To determine the appropriate owner, Controller uses an “active” “non_active” system, where the active and non_active variables are pointers to owners. At the end of each turn, the active and non_active switch. This is so everything can be done with respect to the active player, rather than checking for which player is performing which action. This system also allows for easy implementation of the APNAP system.

Owner

Owner was designed to contain all its own information. If it exists within the specific owner object, it must belong to that owner. Cards are contained using a vector of unique pointers within the owner class, to both uphold RAII and to establish ownership. Cards needed to be defined in memory because they should be able to freely move to and in between collections, but still need to be owned by the owner class. Unique pointers were the perfect solution for this implementation. Collections don't need to use memory, since they will never need to move, they only need to be referenced.

Collection

Collections are all similar to each other in design, as they all hold pointers to cards in vectors which can be referenced by the owner to be moved and manipulated. Each collection differs in what they are allowed to hold and how many cards they are allowed to hold. This is represented through the pure virtual "add" function. The add function adds a card to the collection, returning true if it succeeds and false if it fails.

Cards

Cards are designed such that they can all be used similarly. Since each card can be affected by similar commands (draw, play, etc), but sometimes have different effects. To achieve this, cards have virtual and pure virtual methods where appropriate. Cards also employ the observer design pattern, as many cards (notably, rituals and minions) can be impacted by triggers. The Card class observers these triggers so we know when to activate their abilities or effects.

Resilience to Change

The best example of resilience to change in this project can been seen in the owner, collections, and card classes. The owner class is very self-sufficient, as it owns all of the elements it interacts with, such as collections and cards. Any changes in specification regarding the owner can be done easily because of this. An example of this was the implementation of the novice/master summoner abilities. This was one of the last elements of the project to be implemented and was very simple because of the owner classes resilience. The summon was imported and created using existing functions in owner and was one of the easiest abilities to implement.

Collections are also very resilient to change. The biggest (reasonable) change that a collection could incur is the condition under which we are allowed to add to each collection. For example, the current requirement is that only minions are allowed to enter the graveyard, but if this condition were to change, for example, all types of card are allowed to enter the graveyard, the change would be very simple because of how collections are implemented. A change like this would required modification of the "add" function, which is pure virtual in the abstract collection class and implemented within the concrete collections. Simply change the conditions of the graveyards "add" function, and it will accept all types of cards. This applies to all collections and all of their add conditions. It's also worth mentioning

that the “add” function returns a Boolean value. If a card needs to be moved from one collection to another, it has to be added to one and then removed from the other. It would be difficult to check whether the add succeeded without this Boolean value. If the collections weren’t designed with moving cards in mind, then each condition would have to be checked every time, instead of being checked by the collection itself. This makes collections very resilient to change.

The abstract card class is also resilient to change, since most of the properties of each of the concrete cards come from the abstract superclass. Because of this, many things that apply to one card can apply to them all. This is good because if we wanted to, for example, add the ability to enchant rituals as well as minions (or even enchant enchantments themselves), this process wouldn’t involve too many changes.

The examples of resilience to change provided are actual game mechanics in the card game Magic: The Gathering, one of the inspirations for this project. The entire project was designed with implementing MTG (a much larger card game) in mind, so that switching the functionality to fit sorcery would be much easier. Designing with a larger goal for a smaller project has many benefits, and resilience to change is one of the biggest ones.

The Owner, Collection and Card classes are also have high cohesion and low coupling together. They have high cohesion, as all of these classes cooperate to perform one task, store the owners game state. They are intentionally grouped together and very strongly related. These are the properties of high cohesion. They have low coupling together, as the only communicate through function calls and their results. This is best shown, again by the collections “add” method. Since it returns a Boolean value that represents the success or failure of the add, the “move” method in owner can easily verify the add succeeded before removing from the other collection. These classes also don’t share data directly and do not have access to each other’s implementation (like friends). These are the properties of low coupling.

Answers to Questions

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

Although this was not done in our project, the best way to maximize code reuse for a minion’s activated ability is to use the existing spell class. The minion class could own and construct its own spell card, and the “use” command would activate this spell the same way the “play” command would for spells.

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

The Decorator design pattern would be ideal for implementing Enchantments since they can allow modifications placed on Minions during runtime. By implementing Minions as a Concrete Component of the Abstract Component Card, and Enchantments as Concrete Decorators of an Abstract Enchantment Decorator, we can allow Minions to hold enchantments that can modify its behaviour for Activated Abilities, number of actions, amount of Attack and Defence, and Magic cost throughout the game. As well, we can also remove all Enchantments when a Minion dies by removing the private pointer field for the Enchantments in Minion. This is the same as the plan of action, and how we ended up actually implementing enchantments in our project.

Display of Minion Cards: 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?

For Triggered Abilities, we could implement the Observer pattern, so that we would have a Trigger class such that the Controller has 4 different Trigger objects (for the 4 different types of Triggers). Then, we can make the abstract Card an observer of Trigger (which would be the subject). Then all Concrete Minion Cards can choose to observe a Trigger as part of a Triggered Ability. Also, we can specify any combination of Triggers for Minion Cards by adding it to the list of Observers for the Trigger subject. For activated classes, minions could own multiple spells (as described above). This would maximize code re-use by implementing existing solutions into our new problem. The triggers as described here are how we implemented them in the project, and the activated abilities solution was also talked about above as an ideal solution.

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

One simple way of doing so is by implementing the Observer pattern. We can implement an Abstract Observer class called View which is a parent class to the Concrete Observer interfaces and have the Controller class be the Concrete Subject. Then, when necessary, the Controller class will notify our Concrete Observer Interfaces, which can update the Interfaces, thus reducing the impact to the Controller class no matter how many Interfaces there are. This answer is the same as the plan and is how we implemented the interfaces in our final submission.

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?

Software development in teams can be challenging, but it is without question that it lessens the workload on any individual. One lesson we learned about software development in teams is the importance of testing and version control. Testing is always important, but even more so with multiple developers. If poorly tested code is sent down to other individuals, they may not catch it or know how to solve it as easily as the original coder. Version control is also extremely important. Large scale software development in teams can sometimes go awry, and in the worst case, rolling back to previous versions may be necessary. This happened during our project. We lost a whole day of progress due to segmentation faults that we couldn't debug as it seemed to go back multiple versions. We rolled back the code to an old version, but without version control, we may have lost much more time.

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

Start earlier. We underestimated how much time this would take and ended up crunching to the last minute. There are some things we didn't get to implement that we would've liked and some corners we had to cut. This led to worse code overall.