A5: Sorcery

Anne Sun, Laura Tan, Grace Wang

July 26, 2024

1 Overview

Our project consists of three main base classes: Board, Player, and Card. Card consists of the abstract subclasses Minion, Enchantment, Ritual, and Spell, each of which contain concrete subclasses which denote the specific types of Cards, such as Air Elemental, Dark Ritual, etc.

1.1 Board

Board is the class that manages the players in the game. It also provides print functions to print the cards that are placed down on the board.

1.2 Subject

The observer pattern is used to implement the calling of Minion abilities and Ritual effects of those currently on the board. Each subject has four vectors of observers. These vectors are categorized to be called during four different parts of the game: at the start of a player's turn, at the end of a player's turn, when a minion is placed on the board and when a minion leaves the board. Thus there are also four different attach and detach methods to allow different observers to attach to different vectors of observers.

1.3 Player

The player class creates the players that are currently playing the game. Each player in the game is represented by a player object. Each player object is then stored in a vector of player pointers in main.cc. Each player owns the cards that they will be playing in the game. In addition, we use the **Observer Pattern** to define the relationship between each player and the Ritual and Minion Abilities. Our Player class is a subclass of the Subject Class.

1.4 Card

Card is the base abstract class for our implementation of Minion, Enchantment, Ritual and Spell cards.

1.4.1 Minion, Decorator and ChangeStat

Minion is an abstract subclass of Card that has additional fields and getter functions that extend to accommodate for each Minion's relationship with its abilities and the Spells, Enchantments and Attacks cast on it. Additionally, we have implemented the casting of Enchantments and Attacks using the **Decorator Pattern**.

The subclasses of Minion are the eight minion cards: Air Elemental, Earth Elemental, Bone Golem, Fire Elemental, Potion Seller, Novice Pyromancer, Apprentice Summoner, and Master Summoner. There is also a Decorator subclass of Minion from which our concrete decorator ChangeStat inherits. ChangeStat is a decorator that represents the damage and enchantment buffs/debuffs dealt on the minion.

Example: *Bjarne's Earth Elemental 4/4 is attacked by Linus's Air Elemental 1/1, thus Bjarne's Earth Elemental's current stats are 4/3. A ChangeStat object is wrapped around*

Bjarne's Earth Elemental so that we get the value 3 when we call the Earth Elemental's getter function for attack.

To accommodate for the possibility of concrete minions having multiple abilities, we represent the "triggered abilities" of minions as rituals and the "activated abilities" of minions as spells. Therefore, we define a vector of Rituals and Spells in each minion to hold these abilities.

1.4.2 Enchantment

Enchantment is an abstract class which inherits Card. There are also five concrete Enchantment subclasses which represent the different enchantments that can be played in the game. These include Giant Strength, Enrage, Haste, Magic Fatigue, and Silence. Since enchantments only affect minions, using any enchantment card will create a ChangeStat object that will wrap around the target minion.

Furthermore, each Enchantment has an activate(Minion *) function which behaves according to the purpose of the specific enchantment.

Example: *Bjarne casts Giant Strength on his 1/1 Air Elemental which he had previously summoned. This increases the minion's stats by +2/+2. When this happens, we create a new ChangeStat object that applies the +2/+2 buff to the concrete Air Elemental.*

Another example: *Linus uses the Silence enchantment on Bjarne's Master Summoner, which has an activated ability. When this happens, decorate Bjarne's Master Summoner with a ChangeStat object that does not change its attack and defence. However, every ChangeStat has a boolean blockAbilities field which is now set to true.*

On the other hand, if Linus were to caste Silence on Bjarne's Bone Golem, which has a triggered ability, Silence's `activate()` *function would remove the Bone Golem's ability as a ChangeStat is wrapped around it. When Silence is disenchanted from the Bone Golem, the ability is added back in.*

1.4.3 Ritual

Ritual is an abstract class inheriting Card and is also our abstract observer (except that we named it Ritual). Ritual has concrete subclasses consisting of the three Rituals that can be used by the players, as well as the triggered abilities of the minions summoned onto the board. This includes Dark Ritual, Aura of Power, Standstill, BoneGolemAbility, FireElementalAbility, and PotionSellerAbility. Furthermore, in our implementation of the Observer pattern, each concrete ritual is an observer of the player subject that called it.

Example: *Bjarne summons the Dark Ritual card onto the board. This is the ritual that he currently has in play. When this ritual is summoned to the board, it is automatically attached to the player that summoned it so that it can be notified to call its ability at the start of the player's turn.*

Another example: *A Golem's ability is represented as a concrete ritual. When Bjarne summons his 1/3 Bone Golem to the board, a new BoneGolemAbility object is created and added to Bjarne's Bone Golem's vector of triggered abilities. At the same time, the newly created BoneGolemAbility object is attached to* ***both*** *players' list of observers which are notified when a minion leaves play.*

1.4.4 Spell

Spell is an abstract class of Card whose subclasses are either the playable Spells or Spell objects that represent a Minions' activated abilities. The concrete Spells include Banish, Unsummon, Recharge, Disenchant, Raise Dead, Blizzard, ApprenticeSummonerAbility, MasterSummonerAbility, and NovicePyromancerAbility. In the Spell abstract class, a virtual function called `activate()` is defined. We call `activate()` when we use the spell. `activate()` takes in the same parameters for each Concrete Spell to ensure that we can use each spell without needing to know which exact concrete spell it is.

Example: *Bjarne uses his Banish spell card. If the target that Bjarne is banishing is valid, then Banish will use its ability and return true, indicating that this spell was used and can be destroyed. Otherwise, it will return false which indicates that there was an error while attempting to use this spell and the spell cannot be destroyed (as it technically hasn't been used yet).*

Another example: *A Novice Pyromancer's ability is represented as a Spell. This spell is stored in a vector of Spells created using the getSpells() getter function. Whenever we call this function, we create a stack-allocated empty vector and we add the ability NovicePyromancerAbility() to it.*

2 Design

Each individual card type is its own separate concrete class since

1. Each card has different effects. The effects of cards differ widely. It would be impractical to create a generalized function when it comes to using the ability of cards. By creating abstract subclasses of Card, we could categorize each Card into the four main categories of Minion, Ritual, Enchantment and Spells and categorize them further from there. We created a different general virtual `activate(...)` method which takes a different set of parameters for each of the four Card type abstract subclasses so that it could be overridden in the respective concrete subclasses. Each overridden `activate(...)` method does what the card ability is supposed to do.

2. We can instantiate multiple objects of the same class. For example, in certain decks, there may be multiple Air Elemental Minions, which can be manually loaded into the player's deck through a deck loading function. However, more Air Elementals can be created outside of the deck loading function. Let's say that we implemented Minion cards using a concrete Minion class instead of an abstract Minion class. Therefore to initialize an Air Elemental minion, we would need to initialize the concrete Minion object with the name, cost, attack, defense, and ability description (which in this case is an empty string since Air Elementals have no ability). By making Air Elemental a class, we do not need to manually pass through all of this information. Instead, this information is created with the Air Elemental object, meaning that the class itself holds the information that we need.

2.1 Cohesion

Our program has **high cohesion** as each of our classes only have one reason to exist: to be played on the board by the player. Our Spell, Enchantment and Ritual classes have a minimal number of functions. Each one of them only has getter functions, mutator functions, a print function and the function that activates their ability. Therefore, the only changes that can be made to these three abstract subclasses of Card is to use them or change the cost of using these cards. On the other hand, the Minion class also has multiple virtual getters to access the "fields" of each Concrete Minion using the decorator pattern. Note that we call the cost, attack, defence, name, etc. of each concrete Minion a "field"; however, we are just returning that value in the getter function (ie. return "Air Elemental").

In short, we override each getter function when we wrap a decorator around a Minion. These wrapped decorators are essentially the "mutators" for our "fields". The only three methods of Minion that affect the state of the game are the attack functions and the getSpell() method for minions that have activated abilities. The attack functions either attack a player or another minion. The getSpell() method will allow the minion to use its activated ability if it has one. Therefore the minion doesn't take on any extra responsibilities other than those in which a minion makes a move. Lastly, our Board and Player classes also exhibit high cohesion as our Board class just serves to print the cards on the board.

As a result, it has access to both players and the cards that each player holds. The player on the other hand manages the cards that it owns. This includes the deck, hand, summoned (aka the summoned minions) and graveyard fields. In addition, it is in charge of handling the moving of these cards from one of these fields into the other.

Furthermore, since we need to consider the effects of different types of Cards, Player provides us with methods that link the gameplay. For instance, in *main.cc* we call the removeSummonedMinion() method of Player which unsummons a minion, notifies and calls any triggered abilities or rituals that come into effect when a minion leaves a field (using the observer pattern). In addition, any triggered abilities of the minion will also be detached from the player (subject) and removed from the Minion. Therefore, since the player owns the cards, its purpose is to manage those cards which is done through the many card managing methods that we have created within the player class.

2.2 Coupling

We believe our program has **moderate coupling** as each of our classes (Board, Player, Card, subclasses of Card, etc.) have relationships with one another. However, these relationships and dependencies are not strong to the extent that changes in one class will require significant changes in other classes.

In the instances where our classes depend on each, this dependency is necessary by the nature of the game. For instance, Player depends on some of the direct subclasses of Card, while some of the four direct subclasses of Card depend on Player. This is because certain spells make use of the abilities and fields of Player. For example, in concrete spell cards, spells without a target will affect the players. Therefore, we need access to the player's fields.

On the other hand, certain spells with targets involve accessing the minions that are summoned on a player's board. This means that we need to mutate that minion and the only way to mutate that minion all while keeping it on the board is by accessing that minion through the player's accessor and mutator functions. Thus see some amount of dependency between the classes. Since Minions own concrete Rituals and concrete Spells that represent their abilities, there is also a degree of dependency between these abstract classes. Minions will need access to the Spells class and Rituals class to store these fields and call on the abilities of the concrete types.

With the implementation of the decorator and observer pattern, we can see that multiple of these classes depend on one another. However, note that most of these dependencies exist because the usage of each card has a wide variety, affecting minions, players, rituals and enchantments. To allow the existence of virtual ability-activating functions, it must take in generalized parameters to account for a variety of abilities from the concrete subclasses. For example, the banish spell destroys Minions OR Rituals while Blizzard deals damage to ALL Minions, therefore we can pass in Player* parameters which will give us access to the Minions and Rituals under each player.

3 Resilience to Change

The following features of our program make it resilient to changes.

1. **We do not need to hard code and check each concrete Card before calling its `activate()` function:**

   Our implementation allows for a lot of flexibility when it comes to the addition of new cards. Our card class is generalized into 4 abstract subclasses: Ritual, Enchantment, Spell and Minion. Thus we have generalized abstract functions that belong to each subclass, specifically

   - `Enchantment::activate(Minion*)`,
   - `Spell::activate(Player*, Player*, int)`, and
   - `Ritual::notify()`

   These three functions are in charge of calling the spells, enchantments, rituals and abilities of minions. As long as our test harness can differentiate between the four types of cards that are being used, any Concrete card can be called and used, so we do not need to explicitly check for whether or not the conditions meet for a card's ability to be used.

   In addition, our implementation for our abstract subject class allows for all subjects to take in a pointer to each player and the index (`1, 2, 3, 4, 5` for minions, `r` for rituals, and `-1` for no target) of the target. Thus we have no difficulty with implementing more spell cards as they take in all the important parameters needed to affect either a player or a specified target. For instance, we can call any concrete spell using the `spell.activate(Player*, Player*, int)` method.

2. **We have automated the calling of triggered abilities and rituals:**

   Using our observer pattern, our Subjects are our Player objects and our Observers are our Concrete Rituals. Our Subject contains four different vectors of Ritual pointers

(a) preTurn,

(b) afterTurn,

(c) minionEnter, and

(d) minionLeave,

Each of which correspond to a different state in the gameplay.

In addition, since there are Minion abilities and Rituals where the ability may be triggered at different states for BOTH players, we allow certain Ritual objects to attach themselves to more than one player.

Calling the Standstill Ritual: When a Standstill object is created, it automatically attaches itself to both players so that it can be triggered whenever a minion under either player is summoned.

3. **We have accommodated for the possibility of additional abilities in Minions:**
   One of the questions that have been asked since the start of this project has been about how to maximize code reuse in the case that minions have multiple abilities. Our solution to that is to store our minion's abilities in a vector of Spell* and a vector of Ritual*. A triggered ability represents a Ritual and an activated ability represents a Spell. In our current game, since each minion only has one ability, that single ability is stored in either the vector of Spell or the vector of Ritual. Therefore, if we want to add more abilities to the minion, we simply need to initialize and add a new ability in the vector that stores the corresponding type. If the ability is triggered then we will also attach it to the corresponding subject to be notified when the ability is triggered.

4 Answers to Questions

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

Since spells behave the same way as activated abilities, we decided to represent activated abilities by concrete spell classes. That is, concrete minions with activated abilities own concrete spells which has the same effect as these activated abilities. The only difference is that these concrete spells which represent activated abilities do not get deleted after each use. Instead, the minion's actions and the Player's magic are deducted, and the activated ability/spell is no longer accessible during that turn (since minions can only have a maximum of one action point under normal circumstances).

4.2 What design pattern would be ideal for implementing enchantments? Why?

Implementing a decorator pattern would be ideal for implementing enchantments. In our case, our Minion abstract class is the Component. The abstract class Decorator inherits Minion and we have a concrete Decorator called ChangeStat. Whenever a minion is enchanted and their stats are modified, we 'wrap' a ChangeStat object around that minion. The ChangeStat class overrides the getters of Minion to reflect the changes in a minion's stats accordingly. When an enchantment is removed from a minion, we simply remove or 'unwrap' the corresponding ChangeStat from the minion.

The decorator pattern is ideal for this situation because we can keep track of the effects of enchantments on minions. In particular, if a minion dies and moves from the board to the graveyard, we need to remove the enchantment while keeping all other effects (e.g. damages) applied to the minion. The decorator pattern enables easy removal of the enchantment while keeping other effects on the minion in place. An analogy of this would be a linked list since we are detaching a particular ChangeStat from a concrete minion while keeping other ChangeStats connected to the concrete minion.

4.3 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?

We used vectors of Spells and Rituals and the Observer Design Pattern to implement the calling of Minion abilities. Notice that triggered abilities are similar to rituals and activated abilities are similar to spells. Thus triggered abilities are placed in a vector of rituals and activated abilities are placed in a vector of spells within each concrete minion. Since triggered abilities of minions are only triggered at certain points in time, using our Observer Pattern, each triggered ability is attached to either player so that they can be notified and thus called. On the other hand, activated abilities are manually activated thus we just need to call the function that uses the ith activated ability of the minion. That ability will be in the $(i-1)$th index of the minion's vector of Spells.

5 Final Questions

5.1 What lessons did this project teach you about developing software in teams?

This project has allowed us to experience the game development life cycle, although it is shorter than the usual cycle. We had to brainstorm, design implement, test and go back to brainstorming, designing, etc. This is because sometimes we would have an idea, and try to implement it but then realize that it fails to account for all possibilities. This was especially the case when we would split up tasks into different sections and have each person implement that task individually. We would later find that when we pieced our individual implementations together, we never considered how the other person's implementation could create more edge cases for us or how their piece of implementation may not fully fit together with ours. As a result, a lot of re-implementation had to be done. That is why I've learned that it's really important to brainstorm with the group on how to implement parts of the program so that everyone can be on the same track and everyone's ideas can be uniform.

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

Something we would do differently would be testing as we coded, ensuring that our implementations were working bit by bit. A major pitfall we fell into was that we all made a bunch of files and wrote a plethora of classes, but we could not figure out what went wrong when we tried to compile.

In addition, as was mentioned in the above question, we learned that it's a good idea to discuss our ideas as we implement each bit of our code so that everyone in the team is on the same track.