Sorcery

# Game Design

## Libraries

Player.h, Card.h, Minion.h, Enchantment.h, Ritual.h, Spell.h, Sorcery.h

## Cards

The abstract card class is defined in the most basic way so that the 4 card types can each implement their own functions. This polymorphism is import for allowing card pointer vectors to exist since graveyards, decks, and hands will contain an assortment of the different card types.

Each card has a name, description, type, owner, cost, and trigger variable. These variables are set when the card is constructed. The trigger variable ranges from 0 to 4, 0 indicates no trigger, while the other numbers indicate which trigger type it has (beginning on turn, on minion summon, on spell cast, end of turn). Each card also contains an activate and play function. Although enchantments and spells cannot be activated, the function was necessary for polymorphic design.

When the game is initialized, the cards from each players’ decks will be created as an object on the heap. The location of the card (deck, field, hand, graveyard) are determined by those respective vectors which will contain a pointer to the card. For example, if a card was discarded, the graveyard vector would get an additional element pointing to that card. The hand vector would then lose the element pointing to that card. Because of the operations performed (more details later), the card itself never needs to know where it is located (no location variable). When the game is finished, all card vectors will be deleted.

## Triggers

When a trigger occurs (one of the 4 events), the trigger check function is called with the trigger type given as input. The function will then search all cards on the field in the APNAP order and activate the abilities on the cards via the activate function on the card.

## Minions & Enchantments

Each minion has an attack and defence stat and a vector of enchantments. Minions have the option to attack and use their abilities, both are functions within the class.

Enchantments have two types: enhancers and modifiers. The enhancers are simple stat increases, modifiers alter or add activated abilities. This class contains the type (enhancement or modifier), a calculation function for damage, a calculation function for defence, and an activated ability. In the case of silence, the ability simply does nothing.

## Rituals & Spells

Rituals have additional fields for the number of remaining charges and a charge cost. Rituals can only be activated by the trigger check function and only if there are enough charges.

Spells can only be played directly from the hand and simply call the play function on the card.

## Player

Players have the following fields: a name, magic, life, deck, hand, graveyard, minions, ritual. Name is only for display purposes. The last 5 fields are pointers (or vectors of pointers) to existing cards that were created at initialization. Getter and setter functions were added to the class for each field to accommodate encapsulation.

playCard, useAbility, and minionAttack functions do exactly what you assume they would. By putting these functions within the player class, we can interact with the magic and card vector variables. For example, player 1 calls “play 1” to activate a Raise Dead in his/her hand, the playCard function can assess whether player 1 has enough magic to do so. If so, the graveyard will be checked and the minion added to the field. The magic, graveyard, and minion vector are all contained within player, therefore the function should also be in the player class.

## Main

The main function first checks for the command line arguments: player 1 deck, player 2 deck, and initfile. The decks are then constructed for both players. The program then reads input line by line from initfile until EOF (if it was declared), then takes input from cin. For each command, a series of functions will be called based on the arguments. Most functions involving cards will call the appropriate player functions. Those that involve graphical output will call functions from sorcery.h; these functions have no class. Pointers to the active player and non-active player are used to determine which player functions are called when commands are called (play/use). When a player reaches 0 life, the winner is displayed and the game exits.

# Additional Questions

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

The activate function within the minion class will make calls to other functions based on the minion in question. Therefore, both Apprentice Summoner and Master Summoner will call the same helper function to summon Air Elementals.

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

Enchantments are implemented as a vector of card pointers within the minion class. This is in line with our overall design pattern. The enchantments are added to the back as they are attached to a minion. Therefore, when doing a damage calculation, it will begin at the start of the vector and works its way through the vector. When altering activated abilities, the game will begin searching from the back of the vector for the most recent activated ability enchantment card.

### 4.4.2 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. How might you design your code and the user interface to accommodate this?

Firstly, the function calls for activated abilities would need to be changed. The user would need to add one additional variable in the command: activate minion [target-player] [target-minion] [ability]. Ability would be a number from 1 to [number of activated abilities] and defaults to 1 when there is no value. Triggered abilities would not change much because the trigger check simply cycles through all cards checking for triggers. The only change here would be cycling through all triggers within a minion.

As for the graphical interface, the board itself would simply show the following information for a minion: description (same as before), activated ability count, triggered ability count. The abilities themselves would only be visible upon inspection of the minion. When inspecting, the abilities would be listed out separately. Triggered abilities would be shown in the order by which they would trigger (if there are two for the same trigger).

### 5.2.3 Suppose you choose to implement graphics as a bonus feature. How could you make supporting two (or more) interfaces at once easy while requiring minimal changes to the rest of the code?

Currently the graphical output functions exist separately. For example when board command is input, the function printBoard() is called from within main. Within this function, the choice of interface used would be based on the global variable for interface (bool fancy). If the fancy variable is enabled, then printBoard() will call a different function to generate output. This variable would be set by the command line arguments of Sorcery with “-fancy”.

### 6.5 Describe your additions to the game. How do you expect them to affect your organization of Sorcery’s code compared to if they were not there? Which object-oriented programming concepts were used in your solution?

We have created additional minions, spells, and rituals. This led to the creation of the following new features/mechanics in the game:

1. New trigger – On spell cast. This trigger occurs any time a player uses a spell card
2. Spell cancelling – Stopping a spell before it gets cast
3. Conditional drawing – Draw x cards and discard non-minions

The new trigger means we need to implement an additional trigger check when spells are being cast. Due to the design of the game, that means calling the trigger check function after spells are cast from within the main function.

Spell cancelling (caused by a ritual) is waiting for a spell to be cast and destroying it before the effect occurs. This changes the way spells are activated. Now the trigger check is initiated, and if the Spell Silence ritual is present the spell will be destroyed instead of cast. The operations involved with a normal spell cast are: active ability, remove card from hand, add card to graveyard. Now we simply need to skip the activate portion. The destroy minion on summon works in a similar manner, so this won’t be too far out of the way.

Conditional drawing allows the player to draw x cards and only keep minions. The draw function is already going to be implemented. The discard function will be implemented as well (as a testing function). Therefore, this isn’t much of a burden to add.

Overall, because of the strong design choices, adding new cards and features has not added significant work to the implementation.

# Schedule

## Division of Work

To make sure that the most work can be done, we will complete different sections in tandem, the schedule is as follows:

## Phase 1 (July 14 – 17):

Jason: Create a main harness to use as a template and for testing purposes. Create the libraries for card class, spell class, ritual class, player class.

Steven: Create the libraries for minion class and enchantment class. Implement the graphical libraries so it may be used for testing.

## Phase 2 (July 18 – 20):

Jason: Create additional spell/ritual cards. Implement the spell class, ritual class, and player class (code the .cc files).

Steven: Create additional minion/enchantment cards. Implement the minion and enchantment class (code the .cc files).

## Phase 3 (July 21 – 24):

1. Add error handling for all operations involving user input
2. Beautify the graphical interface
3. Add extra features if time permits