CS246 Final Design Document: Sorcery

Talfryn Yu, Matthew Yang, Jason Wu

Introduction

We have implemented, successfully, all features required in Sorcery.pdf, including some bonus features, including using solely smart pointers with no memory leaks (we have not found any during our testing phase). Immediately after DD1, our UML was discussed and revised. Since then, although some problems have popped up unexpectedly during our implementation, the overarching structure remained roughly the same. We were glad to learn that there were indeed benefits of well-organized planning--as a result of this (and luck), we did not encounter any major problems that we were stuck on for an extended period of time (no problems > 5 hours, collaboratively). Below is our report for this project.

Overview

We will discuss the overall program structure by beginning with a small component of the program (namely the Card class) and building on top of it. We have found this to be the most logical way to explain our program because we came up with our UML design in the exact same fashion. Please bear in mind that when we talk about methods of classes, we skip all constructor, accessor/mutator methods, and the big 5 operations. Furthermore, when we say "a vector of X", we always mean "a vector of shared pointers pointing to X".

So, as we know from Sorcery.pdf, Sorcery is a Hearthstone-like game in which players take turns to play cards and inflict damage on the opponent. When one of the players has life less than zero, the game finishes. Then, a central component of the game is undoubtedly the cards. They are how the players interact with each other and how the game is carried out. Without the cards, the game cannot progress and stays still. When we designed our cards, we gave it an overarching class: the Card class. Each card under the Card class is given four properties: name, owner, cost, type. This is because no matter what the card is, it always has a name that can be displayed, a player it belongs to, a magic cost that is deducted from the player's magic when the card is played, and finally, a type that tells us what *type* of card it is. The first three properties are self-explanatory, and as for the type of the card, recall that in Sorcery.pdf it was specified that a card is one of four different *types*: "Minion", "Enchantment", "Ritual", or "Spell". Each of these four different types of cards has drastically distinct functions, so we chose to assign each a subclass. Note that the type attribute in the Card class is still required so we can easily identify a card's type (and potentially downcast it to its type subclass).

Let us begin with the simplest type of cards: the "Spell" card. Spells are played and after it mutates the game in some type of way, it is immediately removed from the game and destroyed. For this function, we will have an effect method that returns void (overloaded three times, for without target, with target minion, and with target ritual). Each time a spell is played, the effect method is invoked and then the spell is discarded. In addition, each spell has a description attribute that contains an explanation of the effect that the spell causes. This attribute is then displayed when the board is printed.

We now move on to rituals. Each ritual, as described in Sorcery.pdf, has an activationCost, charges, and a triggered ability. We have also added in a method, useTrgAbility, that checks if the player has enough charges to trigger a ritual (if charges - activationCost > 0) and triggers the triggered ability if it indeed does. We are going to explain the triggered ability in the next section on minions and enchantments.

The reason that we are going to discuss the Minion and Enchantment class in the same section is that we have pieced them together using the decorator design pattern: the Minion class is the Component class, and the Enchantment class is the Decorator class. The details of this decorator implementation will be included in the design section below. Each minion has an attribute for attack, defense, activated ability, triggered ability, action points, and action points cap. For activated and triggered abilities, the class contains methods that return whether the concrete minion has activated/triggered abilities. In addition, the minion class has a method for attacking, using its activated ability, restoring its action points, and calling its triggered ability upon death. Activated and triggered abilities have their respective classes, the activated class containing its magic cost, description, and an overloaded method for the effect that it causes while the triggered class contains its trigger conditions, description, and an effect method. Finally, the Enchantment class contains a Minion component (as any decorator would), and strings explaining what the enchantment does for display (its description, and strings for attack/defense change since they have to be displayed separately in the bottom left and right boxes).

All playable cards (e.g. Fire Elemental) are concrete classes for one of the four subclasses described above. Note that none of the four subclasses (and the Card class) are abstract classes. This is so that they can be parameter/return types, a feature inaccessible to abstract classes. The obvious benefit of doing this is that we do not need to write a separate method that would be called on every card (of a certain type) for each individual card (e.g. printing a card, playing a card, etc.).

And that’s it about cards! We now move on to the classes that represent the possible collections where a card may be. They are Deck, Board, Hand, and Graveyard. These classes contain many similar functions with respect to each other, mostly on removing/getting a card by index, getting the index by card, and adding a card to the end of a vector (the end may be right or top, depending on the class) and other manipulations on the vector of cards. The details are described below.

The Graveyard class is probably the simplest of the four. It contains a vector of minions (or rather, shared pointers of minions). We have added methods checking if it is empty, getting the top minion, adding the top minion (when adding minions that have just died to the Graveyard), and removing the top minion (for certain spells).

The Board class is similar to the Graveyard, in that they both contain a vector of minions. However, it also has a shared pointer pointing to a ritual. Methods of the Board class include one that checks if the board is full (i.e. no more minions can be added), a method that adds a minion to the right of all current minions on board (called when the user plays a minion, throws an exception if the board is full), a method that removes minion by index (called when removing dead minions), a method that searches for the minion's index in the board based on the minion pointer (this also comes into handy when removing dead minions), and a method for restoring action points of all minions on the board (of course, this method calls the restoreAction method on each individual minion which we have discussed before).

Next, we will talk about the Hand class. The Hand class, like the Board class, has a method for removing a card by index (when we play a certain card), checking if the hand is full (we call this method before drawing a card), and adding a card to the right of the hand (called when drawing a card, throws an exception if hand full). The difference is that it contains a vector of shared pointers to Card rather than Minion.

Last but not least, we have our Deck class. The Deck class contains an attribute for a vector of cards, a default_random_engine for shuffling the deck, and its owner. You may wonder, "all of these four structures that contain cards have an owner, how come Deck is the only class that has an attribute for one?". The answer to this is that instances of the Card class are constructed by the loadDeck() method, and since the owner is a required attribute for each Card instance, the loadDeck method must have access to the owner. The simplest way to go about this is to just add an "owner" parameter in the Deck class. As for methods, the Deck class has a method for checking if the deck is empty (called before drawing a card), as well as shuffling and loading the deck (both are usually called before the game starts). The shuffling method uses a default_random_engine that is seeded at the beginning of the game loop, and the loading method reads in a deck from a .deck file (e.g. the given default deck).

Now that we have described all collections where a card may belong, the natural question to ask is: where do these collections belong to? As we have briefly mentioned in our description of the Deck class, each of these classes is embedded in a player class. In fact, each player has exactly one deck, board (that contains only the specific player's minions and rituals), hand, and graveyard. So, it makes sense for us to explain the Player class next.

Each player has an attribute for the name, life, magic, number (i.e. player *1* or player *2*), board, deck, graveyard, and hand. In terms of methods, it has a draw method which checks if the hand is full and the deck is empty, then removes a card from the top of the deck and adds it to the right of hand and a discard method that removes a card from the hand by index. These actions are all abstracted away in the hand and deck class, so the implementation of this function is clear and concise. The player also has play and use methods, corresponding to the play and use commands. Both are overridden three times for no target, target minion, and target ritual. The play method receives the card position in hand, checks if the player has enough magic, retrieves the card from hand and downcasts it to its type (type attribute contained in card class so we are able to check), checks if the board is full, play the card (triggering the effect if spell, adding to board otherwise), subtract the magic cost, and removes the card from hand. Note that subtracting the magic cost and removing the card is positioned at the very end to ensure that the after-effects of playing the card are only triggered if the card could be played without any exceptions shown (e.g. if we played a spell that requires a target without the target arguments, the spell would throw an exception during effect, so no effects are actually caused, i.e. we should not subtract magic and remove it from our hand after this). When we are in testing mode, the card is played even if the player does not have enough magic and if so, the player's magic is set to 0. The use method is similar: checks if the user has enough magic, if so, use it and subtract magic. If we are in testing mode, the ability is used anyway and the player's magic is set to 0.

Moving up the ladder, the Game class contains the two players. Other than the two players, the game has an attribute tracking the activePlayer and a mt19937_64 random generator that generates which player to start the game first. The game class includes four important methods: clean, startTurn, endTurn, and checkTriggered. All other methods (with the exception of getWinner, which is later discussed in the game loop section) are created in order to assist these methods, and start, end turn effects, methods removing all dead minions from the board.

We will begin with the method clean. This method loops through all the minions on the board and adds all dead (minion has defense < 0) minions to a separate array. Then, the method checks if each minion in the separate array has a triggered ability with minion death as trigger condition (triggered ability invoked if it does), and the minion is removed from the board and added to the graveyard. The clean method is called after every single player command so that the board is removed of dead minions after each attack, trigger, play, use, etc.

Next, the startTurn and endTurn methods. endTurn and startTurn are called collectively after a player's turn ends. endTurn checks the end turn triggered abilities (invoked if conditions met), as well as the delay enchantment. startTurn changes the active player increases the player's magic restores all minion's action points, draws cards, and checks the start turn triggered abilities (invoked if conditions met).

To check triggered abilities, the method loops through all minions and rituals on board and compares the trigger conditions. If conditions are met, the triggered ability is called. The method that does all of this is called checkTriggered. It is also used after minions are played.

Now that we are done explaining the Game class, we have covered the model of our program, which stores the state of our program. The other parts of our program, i.e. the view, controller, and the game loop, are much easier to understand.

Our program is modeled after a pull-based MVC, so in order for our display to retrieve information from the game, a shared pointer to our main game is stored in the display. The display is split into many components including printing the logo, printing a card, printing a row of cards, etc. For example, printMinion requires printLeftBox (attack value), printRightBox (defence value), and potentially printTopLeftBoxAndDesription (activated ability description and its cost), printDescription (triggered ability's description).

Our controller stores a pointer to the display and game, and has a method for each command given by the player except for "quit". The help, inspect, hand, the board calls the methods of the corresponding name in the display. The play, draw, discard, use methods call the methods of the same name (as mentioned before) in the Player class (for the method play, additional triggered ability checking is called in the controller), and the attack method calls the method of the same name in the Minion class. Finally, the end method calls endTurn then startTurn immediately after.

The game loop constructs the game state, view, and controller after reading in the input and loading and shuffling (if not in testing mode) the decks. All command arguments are checked and the starting player is randomly generated. Finally, the game loop goes over commands given by the init file (if it exists) before reading commands from the standard input. Before each iteration of reading commands, the loop calls the getWinner method from the Game class, which checks if either of the players has life less than 0. If so, the loop breaks, and a winner is declared. Finally, the quit command is handled directly in the game loop with a return statement.

Design

Design 1: We incorporated the decorator design pattern for minion and enchantment. As mentioned in the overview, the Minion class is the Component class and the Enchantment is the Decorator class. In addition to the accessor methods, the Enchantment class has computeAtk, computeDef, computeAct/TrgAbility methods so that when we attach an enchantment to a minion (which may be just another enchantment, since enchantment is the subclass of a minion), these methods are called to calculate and set the properties of the enchantment. These methods are set as virtual methods in the Minion class (by default, they pass on the unchanged value of the minion attached to) and overridden by Enchantments if the Enchantment intends to change any properties of the attached minion (i.e. part of the enchantment "effect"). These compute methods are called only once when the enchantment is attached, and all subsequent attempts to retrieve minion data (attack, defense, etc.) use the accessor (get) method.

This design technique, rather than computing each property from the base up every time and modifying base values, solve the following problem: Suppose that there is a *2 defense enchantment on a minion. If we computed from the base every time and changed the base values, -1 defense on the minion would display as (-1)*2=-2 defense.

Design 2: We used (shared) pointers for storing most classes, so when passing them into functions it was much more efficient.
Design 3: Method overloading on player's play, use method, spell's effect, and many other use cases.
Design 4: Pull-based MVC for our game state, display, and controller, as explained in Overview. Improves code cohesion (e.g. display methods are put in the same class).
Design 5: Heavy use of encapsulation and abstraction on the card collection classes. We created a class for each card collection (hand, board, deck, graveyard) as well as commonly used methods for each class (e.g. remove by index, retrieve by index, addRight/Top, etc.). This is so that whenever we try to modify them (e.g. draw card, minion death, etc.), the code that modifies them is readable and reusable (since we are just using the class functions). The same goes for display methods, as explained in Overview. This improved code reusability and readability and reduced coupling.
Design 6: Inheritance used in Card and its derived classes promotes reusability (e.g. accessor/mutator for the name, cost, owner, type, attack method for Minion, compute methods for Enchantment as mentioned in Design 1, etc.).

Resilience to Change

We will begin our discussion on the resilience to change with our display. If we are merely changing the text display, it is incredibly easy. Everything in the display is coded based on constants given in the definitions file (e.g. CARD_WIDTH, CARD_HEIGHT, BOARD_WIDTH, etc.), so by changing these constants the display would (correctly) change correspondingly. If we wanted to change the display completely, we would create a new class for the display and add methods that correspond to the controller methods. The controller may be changed in a similar fashion. This would be quite simple, as the controller and view are almost completely decoupled from the model.

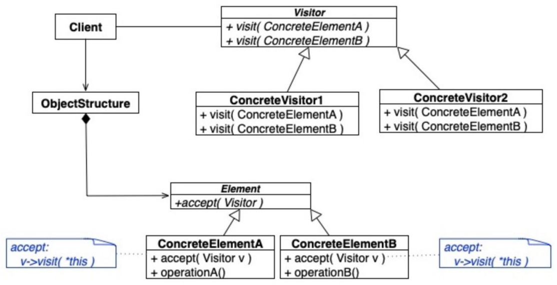
If we were to change actual game components such as HAND_CAP and BOARD_CAP, those constants would need to be updated in Hand and Board files, and also the display file. For starting action points, we just need to change the ACTION_CAP. For changing the life and magic of the player, we just need to change the initial value inputted into the constructor. Adding new cards to the existing card types takes no effort at all: we just need to add a new concrete class of an existing class, which has few parameters. Even adding in new card types would be fairly simple since we would just need to add a subclass under the Card class and concrete instances under that subclass.

Answers to Questions

Question 1: How could you design activated abilities in your code to maximize code reuse?

Answer 1: We created a class for activated abilities that had attributes for cost, description, and a virtual function called effect which would be implemented in every derived instance of an activated ability. In addition, we added a useAbility function in the abstract minion class which would consume magic from the player class (we could implement a consume magic method in the player class), subtract an action point from the minion, and call effects. Then, every time we want to create a new ability, we would only implement the effects that it has on the game (it is difficult to model reusable code for this portion of the code because the effects could be diverse). To use an activated ability, we simply call a minion's useAbility method and it would cover all the requirements of an activated ability.
Question 2: What design pattern would be ideal for implementing enchantments? Why?
Answer 2: We believe decorator to be the ideal design pattern for implementing enchantments. This is because the Decorator design pattern is intended for adding features to an object at run-time rather than to the class. According to the descriptions of enchantments, they are cards that can be played during a turn and lead to modifications to minions. This aligns with the decorator design pattern perfectly: in our case, we can let our component class be the Card class, concrete component object be a specific minion. Our decorator class would be the Enchantment class and our concrete decorators would be specific enchantments.

There are a couple of reasons that this choice makes sense. First, the sorcery document specifies that "If a minion is enchanted by multiple enchantments, they are applied in oldest-to-newest order. " With the decorator design pattern, the oldest enchantment would be the closest to the concrete minion, while the newest enchantment the furthest so it would be technically and intuitively sensible to implement our enchantment in the fashion specified. Second, when we display our minions, we must display the original minion as well as all of the enchantments applied to it. The decorator design pattern would separate each enchantment from one another and the original minion, allowing us to retrieve and display them easily. More information in the design section above.
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 maximizing code reuse?
Answer 3: We think that the visitor design patterns would be helpful for multiple activated and triggered abilities, respectively. We can see activated ability and triggered ability as ConcreteVisitors, and I can see Minion as ConCreteElements.



For each minion, it has vectors of all its activated abilities and triggered abilities . If we grant a minion a new ability, we can add this ability to the vector. When we need to use activated ability or trigger a triggered ability, we search it in the vector and call Minion::accept, which calls the visit of corresponding ability.

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

Answer 4: We used the pull-based MVC design pattern on our interfaces. Specifically, our Game class is the model, our TextDisplay class is the view, and our TextController is a controller class that interprets user input and translates it into model events.

In order to support two or more interfaces at once, we would simply create multiple instances of our view class and controller class, and call different view/controller classes depending on the interface we desire. For example, a new X windows view and a point and click controller.

Extra Credit Features

Feature 1: First and foremost, this entire program only uses shared pointers, and we have not found any memory leaks.

Feature 2: Second, the program incorporates a player elo system that keeps track of each player's elo in the directory /data. The program first loads the player elo (searches for the elo file using the player name, player is given a starting elo of 1000 if the elo is not stored yet), computes the new player elos when the game is over and writes them to a .txt file.

Feature 3: Third, the program adds a new feature to Minion, subtype. So far, we have 2 subtypes: ELEMENTAL and SUMMONER. When there are more than a certain number of same subtype minions on board, there will be an effect on these minions. If we already have more than more than a certain number of same subtype minions, and we play or summon more same subtype minions, all these minions being played and summoned will experience the same effect. (Minions can only experience this effect once) When we don't have enough same-subtype minions on board, the effect will be withdrawn. Notice, if this effect takes place after triggered ability. For ELEMENTAL, if we have more than or equal to 3 ELEMENTALS on board, each of them will have +1/+1. For SUMMONER, if we have more than or equal to 2 SUMMONERS on board, each of them will have +0/*2.

Final Questions

Question 1: What lessons did this project teach you about developing software in teams?

Answer 1: Before this project, all three members of the group have none or little experience in developing group projects. Therefore, the final project provides us an opportunity not only to hone our individual coding skills but also to learn to work as a team.

One thing we learned is that we should discuss our ideas frequently. More communication can bring to less error and more efficiency. For example, when we were creating our header files, we had different ideas on how to implement different classes. Specifically, while one of us promoted the MVC design pattern, another one of us rejected it because he thought it was too complex. To solve this issue, we had an in-depth discussion on how the program would be carried out if we chose to use or not use the MVC pattern. After this discussion, we were able to agree that MVC was the right way to go. On the other hand, we also ran into issues caused by the inadequacy of communication. Because the group members lived in areas in different time-zones, we often finished our parts at different times, and often could not communicate while we were working. This led to a problem where a group member modified another group member's code when another group member was unaware of it. When this occurred, the group member whose code was changed was really unhappy about the change and decided to revert the code back to the original modification. After that, we all recognized the necessity to discuss our ideas frequently. It most definitely enhances efficiency.

Another thing that we learned was that we needed to assign the tasks based on what each member is interested in. Different people are interested in different things, and by focusing each group member on his interest, we were able to not only work better but happier. Examples of this include Matt's interest in writing displays, Jason's interest in designing the different cards, and Talfryn's interest in testing the program.

Last, we really thought that this project was going to be a huge challenge for us. However, when we separated and planned out our work into smaller tasks, we felt that it was not at all that difficult compared to what we anticipated. Hence, we believe that it is important to divide complicated tasks into several simpler sub-tasks to work on.

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

Answer 2: Most importantly, start earlier. Although we did not leave the project to the last minute, we were delayed compared to our plan given in the DD1 plan. We ended up having a lot more bugs (even though our project progressed quite well overall) than we thought we would and it took a long time to verify our program was correct. In addition, some of the group members did not plan out their plans with other finals. Since the last few days before DD2 coincides with many of the other finals of our members, large workloads were dumped onto specific members.

Another thing we should have done was to debug earlier. We should have debugged built functions step-by-step rather than almost writing to the point of completion then debugging. We were lucky to not have bumped into a nasty amount of errors, but even still, lots of errors could have been prevented if we debugged earlier. In fact, debugging later made our entire debugging process extremely complicated and annoying, because it was both time and energy-consuming to examine the wrong code among a large codebase. By doing and compiling step-by-step, it would take less time to debug even offer us insights earlier on to how we should code (so that we would prevent similar mistakes).

Conclusion

This has been a tough but rewarding project. We have all gained much more experience in program architecture and implementation. Most importantly, this project allowed us to experience what the development process of a non-trivial (well, *very* non-trivial) program is like. We used a vast portion of what was taught in 246, from design patterns to reading string streams, from encapsulation to smart pointers, and we were glad to be able to apply theory in practice. Finally, we would like to leave the course with a thoughtful remark (agreed upon, between us, as the most important piece of knowledge learned in the course): debugging with proper tools (gdb and Valgrind) is truly important.