Introduction

Our group chose to write the implementation for Sorcery. We chose Sorcery because we each played card games similar to it, and found that it would best to work on a project that we had some familiarity with.

Overview

Our project will have the structure of the model, view, control design pattern. We use the controller class to manipulate and perform various tasks with the models we write. The models will be the classes for all the different types of cards and objects that will be used throughout the game. We have the player class, which will act as the controller. Finally, the main class, which handles the input from the command line and translates it for the controller to perform the appropriate task. We have a view class which holds all the graphics that will be printed to the screen.

Design

The Card

The biggest challenge was on how to implement the card class and the many children to follow. We knew that a deck will hold the generic card objects, as it is known that a vector can only hold one type of data. Hence, we knew that we need to make use of polymorphism. We decided it would be best for Card to be an abstract class, that way we can make Card object pointers but no actual Card object could exist. This would force us to use polymorphism and only make child objects. The abstract also worked best, as opposed to an interface, as there were some fields and methods that are shared amongst all card types. Thus, an abstract class was able to provide us with the functionality to hold those fields and methods (Such as the card name field, and the appropriate accessor function).

The Deck

The fundamentals of deck were very easy to implement and we chose that the deck should follow the abstract data type stack. As a stack holds the functionality of a deck, where when you add a card, it should go to the top of the deck, and when you draw a card, it takes from the top and removes it from the deck like pop. The biggest challenge was create all the card objects to be added into the deck. As a .deck file only contains the card name to be added. We knew that we needed some sort of database that contained all the information of each card. We opted to use .csv files for each card type, the minion.csv file will hold all the minion cards, and their attack, defence, ability values respectively. The deck class then loads in each csv file into a vector where we hold all the information. Thus we have the add function that takes a card name as parameter, checks which card type vector the card name is in, then pulls all the information of that card and creates the appropriate card object.

Hand, Graveyard and Field

The graveyard followed the deck with using the stack implementation pattern. Where the hand and field, we use a more direct approach with the implementation. The field and hand, needed to be accessed directly, so that we can remove or access a card from a given index. But, when removing a card, the other cards needed to shift over, so that we don't have any empty space between cards to cause any errors. Hence, why we chose the vector to hold cards, as it is easier to use than a dynamic array, as the vector object does a lot of the tedious work itself, such as reallocate size and remove empty space between elements.

The Board

We decided that the board class should be the hub where all the card objects are centralized. A board object contains the field to play cards on, a hand, a graveyard and a

deck. This was so moving cards from one place to another can be done all in one place. We also made the decision that there will be two boards in the game, one for each player. This was done to reduce redundant code, such as having two decks in one board, etc. The board's methods are only accessor and mutator functions. Meaning, from board you can only get pointers to its objects or move a card from one place to another. The board contains no game logic at all.

The Player

We deemed the Player class to be the controller. The player class contains all the actions that a player playing the game can perform. A majority of the game's logic lies within this class. The Player has access to the board, which in turn, allows it to have access to all the subsequent members of board, thus player has access to everything. Hence, it makes it easy when implementing the logic of the game, as we can implement certain rules that apply during gameplay. Like drawing a card, player can call on board to get hand, and player can then check if the hand is full before drawing a card from deck.

Resilience to Change

Abstract Class: Card

We made Card an abstract class, so it can handle having children classes that represent a different card type. Since, all objects that hold cards, have vectors with type Card*, it makes implementing a new Card type very easy. When writing a new card, it inherits from Card.h and just needs to be added as a condition in deck.cc, and it will be implemented.

Player Class

The way the player class is currently set up, the game can hold more than one player if we were to implement this in the main class. When designing the models, we tried to keep everything as general and dynamic as possible by reducing the amount of hard coding there is. The player class when attacking the other player, has a pointer the opposing player and then deals the necessary modifications. Changing the opponent pointer to any other player object will result in the same effects, and hence why our implementation could handle the change of increasing the number of players.

Use of Constants

We also made use of constant variables for max sizes, for example for the hand, the max cards in hand is 5. We have a constant variable that holds that value, in case we wish to change that value, we change the definition of that variable and it will change throughout the entire game without us changing any other line of code.

Answers to Questions

Q1: How could you design activated abilities in your code to maximize code reuse?

Answer: We noticed that some abilities have the same premise but use different values. For example: there are cards that increase a minion's defence by x amount. Hence, we can implement generic abilities and then have them take parameters that will take a specific value for the card's ability.

Q2: What design pattern would be ideal for implementing enchantments? Why?

Answer: Our answer does not change from due date 1. The decorator design pattern is best suited for implementing enchantments because enchantments act as a feature that is added to

a minion during the game. In other worlds, we are adding extra features to minions at run-time, and there is a possibility that they can be withdrawn. If we apply the decorator class to our project, the minion class will take the role of component and the enchantment will be the decorator concrete component.

Q3: 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: Some design patterns that would help us achieve this would be through the use of the Factory method and observer. We can make each minion observer one another, thus when a minion uses an ability, each minion will react accordingly to it. We can have functions that react to a certain type of ability to keep things generic and have a parameter that would hold a value, to handle the severity of the impact of the ability. To implement a minion having multiple abilities, we can use the factory method. We can have each minion card be responsible for creating its own ability object based on its card information. A minion card can hold a method such as createAbility that is pure virtual and returns an ability pointer to whatever ability the minion should have.

Final Questions

1. **What lessons did this project teach you about developing software in teams?**

We learned a lot about the importance of communication. Communication is vital in a team project, and is a must-have at all times. It was extremely important to know who was working on which file at a time in order to coordinate what other people should be working on to avoid any code conflicts when we merge everything together.

We also learned about coding styles and that although we all learned the same material, we all code differently. This proved challenging when trying to make everyone's code work, as it meant we had to adapt to each other's styles to make our program as cohesive as possible.

Another thing we learned is why design recipes and commenting code is vital to a team environment. When debugging someone else's code, it is sometimes impossible to figure out what they are trying to do when there are no comments explaining what a block of code is doing.

Finally, the major thing we learned was how to set up and use git. We made use of GitLab and set up our computers to create local repositories, that we can pull and push from. This enabled us to code simultaneously, then use a GitLab to merge our code together.

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

One thing we would have done differently is spend more time on the planning stage of the project. Throughout the coding, we found a lot of our initial planning was flawed and didn't work as we initially thought it would. We needed to do more research for this project, on different types of classes and relationships, and choose the best one to fit our needs. Also, get a better understanding of everyone's time availability in terms of amount of time they are willing to allocate to this project whilst not ignoring their other school work. There were many days, where only some group members were working due to this, and it caused a bit of imbalance in terms of who did more work than others. A better thought out schedule would prevent such an issue.

Conclusion

We have spent a lot of time and effort on this project. As a group we believe we achieved the goal of learning by working on this project. We learned valuable lessons when it came to teamwork and coding in a team environment. We were able to challenge ourselves by applying a multitude of concepts learned in class as well as concepts learned through research. There are a few things that we would do differently if we got the chance to start over, but we do not regret going through this experience as we gained so much from it.