CS246 Plan of Attack: Sorcery

Breakdown of the project

Project Task	Task Description	Due Date	Assignees
Initial UML	- Decide on a class hierarchy - Identify object-oriented pattern	November 21, 2017	Jafer, Berges, Catalin
Complete interface	- All header files (.h files) - All functions should be properly commented -Files affected: **all header files**	November 22, 2017	Catalin
Complete load function in main logic	- Loading of decks, cards, abilities from text file - Involves creating test cards, minions, etc. - Files affected: **main.cc, Player.cc, Card.cc, NonPlayer.cc**	November 22, 2017	Berges
Complete constructors for each card type	- Taking in the string containing information about a card and initializing using that - Files affected: **Minion.cc, Enchantment.cc, Ritual.cc, Ability.cc, TriggeredAbility.cc, ActivatedAbility.cc, Spell.cc**	November 23, 2017	Berges
Implement all card type logic	- Cast function (applies the card functionality) - Files affected: **Minion.cc, Enchantment.cc, Ritual.cc, Ability.cc, TriggeredAbility.cc, ActivatedAbility.cc, Spell.cc**	November 25, 2017	Catalin, Berges
Implement text-based graphical interfaces using observer pattern and MVC	- Implement textdisplay - Implement observer pattern, setting the Board as the subject and textDisplay as the observer - Handle all overloaded operators for deck - Files affected: **TextDisplay.cc, Observer.cc, Subject.cc**	November 25, 2017	Jafer
Implement board, player logic and main game loop	- Implement the interactions between the Board and Players - Implement all methods in Board - Implement command loop in main.cc - handle parameters specified in requirements - Files affected: **Player.cc, Board.cc**	November 27, 2017	Catalin, Berges
Implement graphics based	- Implement graphics display of the game -Files affected: **GraphicsDisplay.cc**	November 27, 2017	Jafer

Questions

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

Activated abilities in our project were designed as an inherited class from the abstract Ability class; this allowed us to inherit common functionality between abilities and all other cards in general such as casting and updating the card. Since activated abilities are a separate subclass of the Ability class and each minion has a list of abilities, we were able to store both triggered and activated abilities within one list in each minion. As a result, we did not have to write separate handlers for the different types of abilities which effectively maximize code reuse. In addition, if new activated abilities need to be created, we simply need to implement a new concrete Activated Ability class.

Question 2: What design pattern would be ideal for implementing enchantments? Why?

In our design, we're using a modified command pattern. In our case, each enchantment is a concrete command and each minion has a list of concrete commands (or enchantments). Thus, when the Caller/Invoker (which is the BoardController) wants to add an enchantment to the minion (which is the receiver), it is as simple as adding an enchantment to the minion's list. Thus, all modifications that an enchantment applies are handled. Furthermore, this design is effective because it decreases coupling. That is, adding a new enchantment means we only need to create another concrete enchantment class. Similar to our activated abilities, no other changes are required to implement a new enchantment card.

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?

Using our initial implementation where the minion has its abilities as a property, to accommodate for this change in space, our design implements a vector of abilities within each minion. Since both triggered and activated abilities inherit from the Ability superclass, this vector can contain any combination and number of both types of abilities making this implementation very dynamic. Our current design pattern maximizes code reuse because if there is a new type of ability apart from triggered and activated abilities, we simply need to create a new class that inherits from Ability and no further changes are needed. Implementing a new concrete ability of any type only requires the creation of a simple function.

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

We can use the observer subject pattern and add each type of display required as an observer and the borardController as the subject. Thus, at every stage of a turn; pre-stage, execute-stage and, post-stage, each observer is notified and then displays the new state of the board. In this way if a new interface is created we can simply add it to the list of observers, this way no other changes are needed.