Breakdown

Based on the UML we designed, our Sorcery game will be constructed based on two base classes: Card and Ability. Card, which is the core class, derives different types of cards, namely Ritual, Minion, Spell, and Enchantment, each presented by a subclass of Card. In order to maximize code reusability, we designed the Ability class as a base class for triggered ability and activated ability. Ability contains a string that describes the effect of the ability and virtual methods that allow it to be cast on a minion or a player, which needs to be implemented in its child classes Triggered and Activated. Since the methods' parameter can be either a Minion or a player, we use the visitor design pattern to resolve the issue. A minion has up to one triggered ability and one activated ability. Moreover, a Ritual IS A Triggered ability and a Spell IS AN Activated ability.

Player is a class that stores information and attributes for a player, such as Hand, Deck, Graveyard, and Board. Player owns Hand, Deck, Graveyard, and Board, each of which contains Cards or Minions in an array and can modify the objects in its array. For example, the shuffle method in Deck would change the sequence of the objects in its array of Cards. Graveyard, which contains an array of Minions, is a container for all dead minions. Players will play or draw a card through the methods provided in Hand, Deck, Graveyard and Board.

The first step to coding out project is to create the interface (.h files). The interface for Player, Ability and its subclasses, and test harness will be handled by Chang Liu. The rest of the interface will be handled by Yafan Wang. Meanwhile, Amber Chen will take charge of creating test cases.

After that, each of us will take on some classes to implement. Once that's done, we'll compile our code and debug together. After making sure that we meet all the requirements for the basic part of the project, if we have time left, we'll implement some extra features for bonus marks.

Chronological Plan of due dates

1. UML and project design (Together by 2019/07/21)
2. The user interface, test harness and test cases
   (Together 2019/07/24)
3. Implementation of Player, Hand, Board, Graveyard, and Deck
   (Grace by 2019/07/26)
4. Implementation of Card, Minion, Enchantment
   (Amber by 2019/07/26)
5. Implementation of Ability, Triggered Ability, Activated Ability, Ritual, Spell (Chang by 2019/07/26)
6. Implementation and modification for interactions among Classes
   (Together by 2019/07/27)
7. Testing and remodification for functions and classes
   (Together by 2019/07/29)
8. Design for bonus functions or cards (Together by 2019/07/28)
9. Final review and modification (Together by 2019/07/30)

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

Answer: In order to maximize code reusability, we designed the Ability class as a base class for triggered ability and activated ability. Ability contains a string that describes the effect of the ability and virtual methods that allow it to be cast on a minion or a player, which needs to be implemented in its child classes Triggered and Activated. Since the methods' parameter can be either a Minion or a player, we use the visitor design pattern to resolve the issue. A minion has up to one triggered ability and one activated ability. Moreover, a Ritual IS A Triggered ability and a Spell IS AN Activated ability.

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

Answer: Decorator pattern would be ideal. The enchantments can layer on top of each other on a minion, in the oldest-to-newest order, which matches the decorator pattern.

Question: 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: Design a base class Ability as the parent for activated and triggered abilities. Use decorator pattern for Ability.

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