**Final Week Hours:**

Ben:

Fernando:

Sami:

Ziming:

**Repo Link:**

**Video Link:**

**Teamwork:**

We’ve elected to submit individual emails with regards to our contributions.

**Setup Instructions:**

The project is to be started up through the CardQuestApplication.java file in the model folder. Once the application, players can open their localhost on port 4200 to join the game.

Once enough players have joined,

**Design Patterns:**

* Strategy:
  + The strategy pattern is used when we need to designate specific behaviour for a context object at runtime. By creating an interface that declares a method for concrete strategy objects to use, we can have these concrete strategy objects implement different behaviours for the method. The context object can be composed of a concrete strategy, and when we need to call upon the behaviour we have “strategized”, we call the appropriate method in the context’s strategy object. In our program, we use this to designate behaviour to classes of cards (quest, event, foe, etc.). We have created a Playable interface that declares an OnPlay() method. A concrete class implementing the Playable interface then defines behaviour for OnPlay() based on what type of “Playable” object the class is (PlayableQuest, PlayableEvent, PlayableFoe, etc.). Cards are composed of composed of these objects. When a card is played, the card calls on its Playable objects OnPlay() method. This way, cards can have dynamic behaviours and minimize coupling, since each card is only coupled to an abstraction of “Playable”.
* Observer:
  + The observer pattern is used to send updates about a single object that many other objects are observing. Observers are registered to a subject and can then either be pushed updates from the subject or can pull updates from it. With the information acquired, the observers can act accordingly. We use the observer pattern in our game to update players about the game state. Our game class has an arraylist of Players that receive updates from the Game. Every so often, players will automatically request updates from the game to update their state. Despite the mediator pattern allowing for looser coupling, we chose this pattern over the mediator pattern because it is best used when the complexity of object communication becomes a hinderance to object reusability. We felt that this was not the case and that having to add an extra layer of abstraction would be more tedious than helpful in this situation. So, we did not use the mediator pattern.
* Decorator:
  + The decorator pattern is used to attach responsibilities to an object dynamically without having to subclass. This pattern is normally useful because it allows specific instances of a class to have unique behaviour. We did not choose to use this pattern. The obvious place to use it, however, would have been for dynamically creating cards. We felt that this was taken care of satisfactorily by the strategy pattern. Many individual cards do not have behaviour that is unique to only them, but instead have generic behaviour that can be attributed to their whole type of card. As such, we opted to use the strategy pattern to create behaviour for card types, instead of creating many components build a card out of to achieve uniqueness. Although this pattern allows for loose coupling, we felt that it was overkill for this project, and that the strategy pattern would allow us to achieve this sufficiently.
* Factory (concrete):
  + The factory pattern is used to standardize the instantiation objects whose class are the child classes of a specific parent class without having to expose the instantiation logic for the child classes to the client. The client calls a method of the factory object to create a specific type of child object of the parent that the factory is assigned to. The client only needs to know what the parent class is and so the client is loosely coupled from the parent’s children. We use this design patterns to instantiate the story and adventure decks. We have a class for Deck objects that have arrays of Cards in them. Depending on what type of deck we are instantiating we need to fill this array out accordingly. Although there are no subclasses for Decks (as different decks do not have different behaviours, only different types of cards) we still felt this pattern was appropriate for this purpose since there is a substantial amount of logic required to create both types of decks, and we do not want to expose this to the client. We also opted not to put this logic in the constructor of the Deck class so that we could separate the logic of filling out the decks from the behaviour of the decks once they are filled out. We do not use the abstract factory because we only have the one DeckFactory that we need to create.