1. The Observer Pattern: This pattern has two major parts. The Observer and the Observable or as the book calls it, The Subject. The basic premise of this pattern is that the Observer registers as an observer of the Subject. When the Subject changes it notifies it's list of observers. The Observer then does whatever it needs to. The Observer pattern can be used when you don't know before hand how many objects will need to change when the subject is modified, or if you don't want objects too tightly coupled. The Observer Pattern seems to be very applicable when using the MVC or MVP design pattern and you want the model to be able to notify either the view or the presenter. This is why we used it in our design for phase 2. We used this pattern as a communication system between the model and the controllers that were given to us. We created a Class called EventObservable that allows us to register a class as an observer for specific events. For example, in the MapController we registered it as an observer for several events, one of them was addCity. The code for that is as follows:

```
EventObservable.getSingleton().subscribeToEvent(Event.UpdateMapAddCity, new
IObserver<ICommunity>() {
        @Override
        public void update(final ICommunity metadata) {
                EventQueue.invokeLater(new Runnable() {
                        @Override
                        public void run() {
                                addCity(metadata);
                        }
                });
        }
});
```

Here you can see that we are subscribing or registering the MapController for the event

UpdateMapAddCity. When this event happens, the EventObservable Class sends out a notification to the MapController. This causes the update function to run which puts the event on the View's thread to run when it can. When it does run it calls the addCity function on MapController. This allows MapController to decide how to handle the notification.

2. The State Pattern: The State Pattern is a design pattern that is used when an object's behavior depends on the state of the object. How this works is as follows. Class A contains object B. B implements the State pattern. B could be an abstract class or an interface for example. Whenever somebody calls method C on object B, They don't really care or even know what implementation of B they are calling. Because B is either extended or implemented by it's subclasses, every method in B will be in the class that was actually instantiated. B then changes to point to the correct subclass

the corresponds with the current state. The reason why would want to use this is if you have large, conditonal statements like a switch case or a bunch of if else's and these condtions depend on the state. For example, we used the State pattern in the MapController. The MapController should respond very differently depending on the current state of the game. You can buy or play Dev cards during round's one or two or you can place a road while you are rolling. One way to handle this would have been to have a lot of conditional logic in a single function. We decided to use the State Pattern and have a separate class for each state. All the MapController has to do is call MapState.getState().placeCity(vertLoc); for example. The MapController does not care what state the game is currently in. The State determines how to handle the call by trading out the class that gets called based on the state of the game. For example during the Rolling state the placeCity function is

```
@Override
public void placeCity(VertexLocation vertLoc) {

}
```

but in the Playing State the same function is

```
@Override
public void placeCity(VertexLocation vertLoc) {
    if(canPlaceCity(vertLoc)){
        ClientModel success = null;
        int playerIndex =
        ClientModel.getUpdatableModel().getLocalPlayer().getPlayerIndex();
        try {
                success =
                ClientCommunicator.getClientCommunicator().buildCity(playerIndex,
                vertLoc);
                if(success == null){
                        throw new Exception();
                }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
  }
```

The great part about this is if in the future we added another state, we would only have to change the code that would allow us to reach that state. We would not have to change the logic of the addCity function anywhere, just add it to the new state.
Team Report:

- Curtis: 5
- Mitchell: 3
- Kevin: 3
- Jacob: 2
- James: 1

I don't think James actually did any programming for this phase. He was better at letting us know that he would be around but still not great.

Jacob created the EventObservable class which helped out a ton but that was it as far as I know. He did not help implement any of the controllers and was not available for towards the end of the phase I felt.