1. Facade: This is kind of like an interface but allows you to have one class that acts as the interface for an entire subsystem. The best example that I can think of is a web service. All traffic gets routed to one class that then sends the commands to the appropriate locations in the server. It then handles the return calls as well. So as far as the client knows, there is only one class sitting on the server. In our project, this is exactly how we used it. We created a server facade that would handle all traffic to and from our server. I have include just the interface as it is quite large, but you can see that every endpoint is included in our ServerFacade

```
public interface IServerFacade {

    /**
     * Change the level of logging on the server.
     * @param level See LogLevel for options
     * @return result of change
     */
    public String changeLogLevel(String level);

    /**
     * Logs in User
     * @param request Login User Request Parameters
     * @return Success Status
     */
    public LoginResult loginUser(LoginUserRequest request);

    /**
     * Registers a User
     * @param request Register User Request Parameters
     * @return Success Status
     */
    public LoginResult registerUser(RegisterUserRequest request);

    /**
     * Allows a user to join a game
     * @param request Join Game Request Parameters
     * @return Success Status
     */
    public boolean joinGame(JoinGameRequest request, Credentials credientials);

    /**
     * Allows a user to create a game
     * @param request Create Games Request Parameters
     * @return Success Status
     */
    public ModelResult createGame(CreateGamesRequest request);

    /**
     * Get a list of created games
```

```
 *
 * @param request List Games Request Parameters
 * @return List of Games
 */
public List<IGameModel> listGames(ListGamesRequest request);

/**
 * Loads a game that has been previously saved
 * @param request Load Game Reqeust Parameters
 * @return Success Status
 */
public String loadGame(LoadGameRequest request);

/**
 * Saves a game, can restart server and load saved game
 * @param request Save Game Request Parameters
 * @return Success Status
 */
public String saveGame(SaveGameRequest request);

/**
 * Add an AI Player to a game
 * @param request Add AI Request Parameters
 * @return Success Status
 */
public String addAI(AddAIRequest request);

/**
 * Get a list of the available AI Player Types
 * @param request List AI Request Parameters
 * @return List of AI Types
 */
public AIList listAI(ListAIRequest request);

/**
 * Get a list of commands available to send to the server
 * @param request Get Game Commands Request Parameters
 * @return List of Game Commands
 */
public ICommandList getCommands(GetGameCommandsRequest request);

/**
 * Send a list of game commands to the server
 * @param request Post Game Commands Request Parameters
 * @return a Game Model
 */
public IGameModel postCommands(PostGameCommandsRequest request);
```

```
    /**
     * Get the most recent Game Model
     * @param request Game Model Request Parameters
     * @return a Game Model
     */
    public IGameModel getGameModel(GameModelRequest request, Credentials
credientals);

    /**
     * Reset the game model
     * @param request Reset Game Request Parameters
     * @return Success Status
     */
    public String resetGame(ResetGameRequest request);

    /**
     * Send Command to Accept a Trade
     * @param request Accept Trade Request Parameters
     * @return a Game Model
     */
    public IGameModel acceptTrade(AcceptTradeRequest request, Credentials credentials);

    /**
     * Send Command to Build a City
     * @param request Build City Request Parameters
     * @return a Game Model
     */
    public IGameModel buildCity(BuildCityRequest request, Credentials credentials);

    /**
     * Send Command to Build a Road
     * @param request Build Road Request Parameters
     * @return a Game Model
     */
    public IGameModel buildRoad(BuildRoadRequest request, Credentials credentials);

    /**
     * Send Command to Build a Settlement
     *
     * @param request Build Settlement Request Parameters
     * @param credentials
     * @return a Game Model
     */
    public IGameModel buildSettlement(BuildSettlementRequest request, Credentials
credentials);

    /**
     * Send Command to Buy A Development Card
```

```
     * @param request Buy Dev Card Request Parameters
     * @return a Game Model
     */
    public IGameModel buyDevCard(BuyDevCardRequest request, Credentials credentials);

    /**
     * Send Command to Discard Cards
     * @param request Discard Cards Request Parameters
     * @return a Game Model
     */
    public IGameModel discardCards(DiscardCardsRequestServer request, Credentials
credentials);

    /**
     * Send Command to Finish Turn
     * @param request Finish Turn Request Parameters
     * @return a Game Model
     */
    public IGameModel finishTurn(FinishTurnRequest request, Credentials credentials);

    /**
     * Send Command to Make a Maritime Trade
     * @param request Maritime Trade Request Parameters
     * @return a Game Model
     */
    public IGameModel maritimeTrade(MaritimeTradeRequest request, Credentials
credentials);

    /**
     * Send Command to Play Monopoly Dev Card
     * @param request Monopoly Request Parameters
     * @return a Game Model
     */
    public IGameModel monopoly(MonopolyRequest request, Credentials credentials);

    /**
     * Send Command to Play Monument Card
     * @param request Monument Request Parameters
     * @return a Game Model
     */
    public IGameModel monument(MonumentRequest request, Credentials credentials);

    /**
     * Send Command to Make a Domestic Trade
     * @param request Offer Trade Request Parameters
     * @return a Game Model
     */
```

```
    public IGameModel offerTrade(OfferTradeRequestServer request, Credentials
credentials);

    /**
     * Send Command to Play Road Building Card
     * @param request Road Building Request Parameters
     * @return a Game Model
     */
    public IGameModel roadBuilding(RoadBuildingRequest request, Credentials credentials);

    /**
     * Send Command to Rob Player
     * @param request Rob Player Request Parameters
     * @return a Game Model
     */
    public IGameModel robPlayer(RobPlayerRequest request, Credentials credentials);

    /**
     * Send Command to Roll Number
     * @param request Roll Number Request Parameters
     * @return a Game Model
     */
    public IGameModel rollNumber(RollNumberRequest request, Credentials credentials);

    /**
     * Send Command to Send Chat
     * @param request Send Chat Request Parameters
     * @return a Game Model
     */
    public IGameModel sendChat(SendChatRequest request, Credentials credentials);

    /**
     * Send Command to Play Soldier Card
     * @param request Soldier Request Parameters
     * @return a Game Model
     */
    public IGameModel soldier(SoldierRequest request, Credentials credentials);

    /**
     * Send Command to Play Year of Plenty Card
     * @param request Year of Plenty Request Parameters
     * @return a Game Model
     */
    public IGameModel yearOfPlenty(YearOfPlentyRequest request, Credentials
credentials);
}
```

2. Command: This pattern is a way to encapsulate everything that you will need to call a method in the future. This has the benefit of being able to wait to actually call the method until the command as arrived at where it needs to be. It also allows you to undo what you just did because you can store the command, which as we just stated, has all the information need to make a method call in the future, so you know exactly what needs to be reversed. We used this pattern in our Server. As I am writing this report, I realize that we actually implemented the Command Patter in correctly so I am included the corrected version of our code. Here the interface for our commands

```
public interface ICommand {

    /**
     * Execute the Command (Command Design Pattern)
     * @return Object, type depends on Command being made
     */
    public Object execute();
}
```

and here is an example of the Command implmente:

```
public class BuildRoadCommand implements IMoveCommand {
    private BuildRoadRequest request;
    private Credentials credentials;

    public BuildRoadCommand(BuildRoadRequest request, Credentials
credentials) {
        this.request = request;
        this.credentials = credentials;
    }

    @Override
    public IGameModel execute() {
        if(!ServerModel.getSingleton().checkGameCredentials(credentials))
            return null;

        IGameModel gameModel =
ServerModel.getSingleton().getGames().get(this.getGameId());

        EdgeLocation location = request.getRoadLocation();

        gameModel.placeRoad(location,
gameModel.getPlayerIndex(credentials.getPlayerId()), request.isFree());


gameModel.addLog(gameModel.getPlayerIndex(credentials.getPlayerId()),
credentials.getName()+" built a road");
```

```
        return gameModel;
    }

        @Override
        public int getGameId() {
                return credentials.getGameId();
        }
}
```

3. Dependency Injection: This pattern allows us to remove hard-coded dependencies from our program and make it loosely coupled and extendable. We can implement the dependency injection pattern to move the dependency resolution from compile-time to runtime. We did not use this pattern in our project.
4. Team Report

   I spend about 5 hours on this phase

   Curtis: 5
   Mitch: 4
   Kevin:2
   Jacob: 3
   James:1