The rational that we used in designing our version of Monopoly was that we wanted clear separation of class roles, as few tightly coupled classes as possible, and a clear and understandable design that would be easier for us to improve for Iteration 2. For clear separation of class roles we started by creating a class that handles each of the roles that are found in the game, for example the players, the banker, and the game. Due to the complexity of the game we decided that it would be best to further break down the game to make it more manageable. This led to the creation of Turn, GUI, Board, and Square (with its respective subclasses). The GUI itself inherently possesses a lot of repeating code, especially for squares so we pulled out a GUIHelper class that helps manage some of the repetitive code that is pervasive throughout the GUI, we attempted further refactoring but ran out of time to continue this endeavor, we will continue refining the GUI in iteration 2.

In order to achieve our goal of as few tightly coupled classes we determined that the best course of action was to follow the Law of Demeter and make calls out to limit each classes understanding of the others. There was still a very tight coupling between GUI and Turn but for the most part the rest of our classes do not need to know about very many others.

For design patterns we really had three major patterns that we wanted to implement. The first is an observer pattern between the GUI and the game. This manifested itself as the GUI being the observer and updating whenever something significant happens in Turn. This proved efficient since Turn for us represented most of the functionality that occurs during a single players turn. We also wanted to have Turn have most of the state that the GUI needs to display. So for us we found that most of the state that needs to be displayed could be offloaded to the player and the game so that the GUI only had to communicate with the game itself and Turn to get all the information it will need.

The other design that we really wanted to implement was something of a state design with the players. Since the rotation of players is a constant function of the current player we thought that it was simple to represent this association as a state design where the event moving between players was end turn. This simplified many of the interactions since GUI and the Game didn’t need to constantly instantiate a new instance of Turn and keep stack of who the current player was.

Lastly we felt that there was a good ability to generalize Squares as a superclass for deeds and non-deeds since they both share the need the ability to perform some action on a given player such as paying rent, paying taxes, or buying properties. This also gave the benefit of simplicity to the Board since it only needs to know about the square class and keep track of them. We counted on dynamic binding to ensure that the proper PerformAction method was called but it greatly simplified how we handled a player’s interactions with tiles.

As far as design principles go we tried to encapsulate elements whenever possible since there are a lot of moving parts we felt that it was necessary to try and keep as many of the moving parts away from each other. When it came to the GUI there was a lot of repetition so one of our main tasks was to try and pull out as many similarities as we could into helpers and sub classes. Considering the repetition that is present we saw a lot of gains for bug fixing and in maintainability when we followed the Don’t Repeat Yourself principle. Lastly as frequently as possible we attempted to follow the single responsibility principle, such as Turn is simply meant to represent the state and actions of a player, and player has an object assets that keeps track of all the money and deeds that a player can own.

We saw a lot of benefits in using the aforementioned design patterns and principles, namely in how we were able to conceptualize where the project was going, how to implement features, and notably easier bug fixing.