For Iteration 2 we shifted our focus from setting up good design to maintaining good design. For us we decided that meant adjusting the design of a couple of classes, refactoring repetitive code into better designed units, and enforcing the single responsibility principle. The main rationale for these focuses was to try and avoid as much headache as possible while we were adding more functionality. Much of our refactoring and adjustments was fixing the mistakes that we made during the first iteration, or in some cases, simply needing specific functionality from other classes that we didn’t have previously.

For adjusting the design of classes our main change was to make GUI a singleton class. It became apparent that many classes would need to be able to affect the GUI and thus would need to be able to have a reference to a single version of the GUI. This proved very beneficial when we were adding text to the dialog box that we use to give information, and so that the GuiStateTracker can affect the GuiHelpers.

This iteration really was a lesson in how not to repeat ourselves. When adding the ability to select properties, there were a large number of changes that had to be made to the tiles. This in turn made it all the more apparent that GuiHelper needed to have control of everything relating to a tile. After determining position of houses and hotels, we added those boxes into the GuiHelpers since it was unmaintainable to handle boxes independently of tiles. In a different class, Cards, we learned a lot about repetitive constructors from Board.java, so in an effort to avoid this we abstracted all of the data of the individual cards into a config file so as to ensure that we wouldn’t need to write a constructor for each individual card that we wanted to add.

The MVP of this iteration for design principles was the single responsibility principle. As our classes expanded it became infeasible to work on classes that did too many things. For us the main response to this was to strictly enforce single responsibility whenever possible. For some of the bigger classes like GUI, Monopoly, and Turn we couldn’t enforce this due to their scope; however for each of the smaller classes this principle became a life saver. This had the side benefit of isolating many of the changes so that a single class only changed when it was supposed to. For a class like Turn, the facade of many different classes, there is a lot of distinct states that need to be accounted for and thus caused a lot of headaches whenever we made the smallest change since many of the states would react poorly to any change. However as previously discussed the Don’t Repeat Yourself principle solved a lot of these problems as we were able to isolate many of these state changes into single methods that were responsible for all relevant changes. While this wasn’t exactly Single Responsibility, in the larger classes it was able to function in a similar fashion as Single Responsibility.

This iteration in particular was when the design decisions in the past came to fruition or ruin. For some of our decisions like having a couple larger classes, we had many issues that arose mainly due to our design choices. However when we were working on classes that had been really well designed ahead of time changes could be minimal and added with almost no difficulty whatsoever.