Design

Design Rationale

For the second iteration, we kept the structure of our code mostly the same. We were still keeping our user interface code separate from our underlying code and using a class to share code between the two. We tried to keep our code adaptable since we needed to add functionality for buying houses or hotels and chance and community chest cards. When starting the second iteration, we wanted to be able to add classes for chance and community chest without breaking any other code and we wanted to add methods for buying a house or hotel without messing with the other code. We were unable to implement the Observer pattern or other patterns like State/Strategy, Decorator, or Adapter because we did not have the time. What we did instead, was refactor out the user interface, so the code is much more readable and the main GUI class isn't hundreds of lines of code.

Design Principles and Patterns

The design principles we used in this iteration were all the ones we used from the first iteration, but mostly the single responsibility principle and the open/closed principles for this iteration. We also kept the same design pattern which was Model-View-Controller.

Where the Principles and Patterns were used

The single responsibility pattern can be seen with TopPanel, BottomPanel, ListPanel and BoardPanel. These classes are the result of refactoring Window. These classes are JPanels that make up the different parts of the game window. TopPanel deals with the new game button, and shows a player's name, money, how many spaces they moved and how much time is left. BottomPanel has all the buttons to perform roll, buying a house/hotel, end turn, and mortgage/unmortgage. ListPanel shows a player's owned properties and includes a description box so users understand what is happening in the game. Finally, BoardPanel draws the board, implements moving pieces on the board and drawing houses/hotels on properties. Each of these classes make up the game window but only have one functionality. These classes do not try to do the job of another. They only have one job.

We used the MVC pattern, but did not structure the files into different packages. We did not use getters and setters all the time which caused problems when trying to make packages for Model, View, and Controller. We still tried to use the design pattern, but not formally. Classes in the Model like MonopolyBoard and Property give information or data to Controller so that Controller can pass it to classes in the View such as Window so the information can be displayed.

The open-closed principle can be seen be with Deck, CommunityChestCards and ChanceCards. CommunityChestCards and ChanceCards inherit from Deck, which is an abstract class. This

Deck represents a deck of cards on the monopoly board. CommunityChestCards and ChanceCards share similar features such as shuffling the deck, drawing a card from the deck and adding the card back to the deck. The difference between these two decks is the cards in them. CommunityChestCards and ChanceCards inherit the cardDrawn method from Deck but implement them on their own. By using the open-closed principle here, we can share features between the two decks, but also add functionality for the card that is drawn and create different cards for each deck.