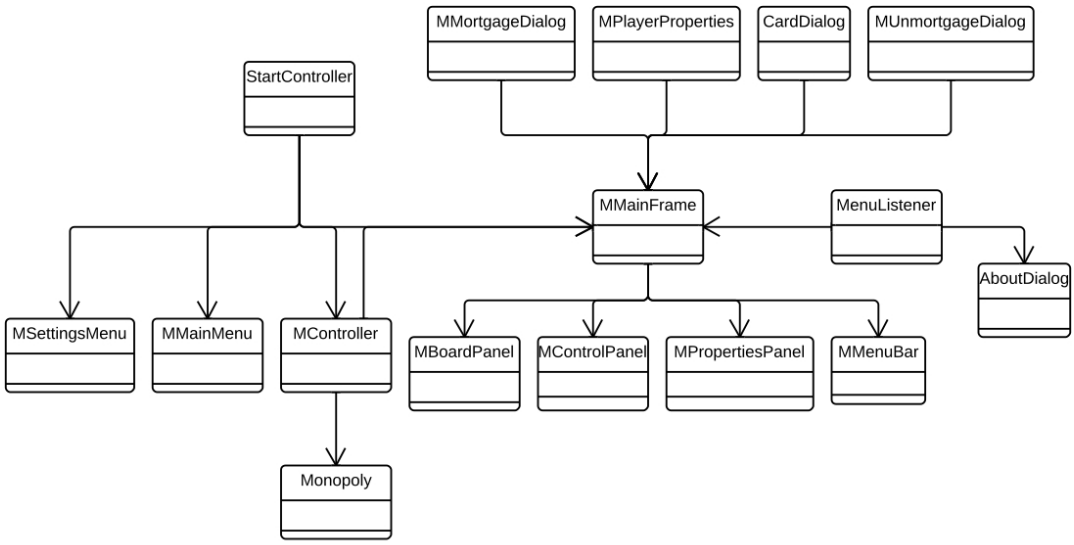
Technical Manual - Monopoly: Reddit Edition

Monopoly is a popular game that everyone is familiar with. It started as a board game in the mid 1900's and has hundreds of different editions and types in a multitude of languages. Also becoming more and more popular are the electronic versions of the game. There are games for windows, apps, and even little flash games that can be played. However, the vast majority of these games are poorly done, cumbersome to use, and often cost more money than they are worth. There are also very few editions online. None of the online or physical board games had the edition we wanted to make, so we decided to make one that is efficient in its implementation, is user friendly, and has a theme that is popular and doesn't exist yet; Reddit.

The Model-View-Controller strategy was implemented for this game. The view is what the user sees and interacts with, the model is the back end with all of the actual functionality and framework for the game, and the controller links the two. The user will interact in some way with the view, and the everything that needs to be handled is done with the controller. The controller will take the input that the player has given it, and manipulate the model. The model will then update the view accordingly.

A somewhat unique and interesting feature that we added to the game is that all of the board tiles, chance, and community chest cards are stored in XML files. These files use simple tags such as <Property>, <Rent>, <Collect>, and much more, that can be parsed in the java program. This was a cool feature because if you go into the XML files, you can change the game to be a completely different themed type of game by changing some of the things inside the tags of the XML files. All the chance and community chest cards can be easily edited so that they say and do different things. And most of the information about the properties: their rent, mortgage value, and name, could be changed very easily by going into their respective XML file.

The View and the Controller



The view uses several different frames that it has the ability to switch between, but it also has several different components that each of the frames use to make a functional GUI. Interwoven with these frames and panels are several action listeners to respond to user input, as well as two controllers that help control the game's flow.

When the game is first launched, an instance of the StartController is launched. This controller helps control the flow of the initial menus that are presented to the user such as the Main Menu, and the settings menu. It also helps launch new instances of the game, whether it be as a completely new game, or as a loaded game. The StartController acts as both a controller and a action listener for the main menu and the settings screen, and this is optimal, as neither screen has a particularly complicated function other than to set variables, and change screens.

When we get past the start menu's, we open up a new instance of the game classes. The main classes that have to be instantiated by the controller - StartController in this case - is the main frame, which itself then opens the necessary panels, as well as Monopoly - the model for the game, and the MController, which is passed a number of settings as well as the View and the Model to complete the Model View Controller design pattern.

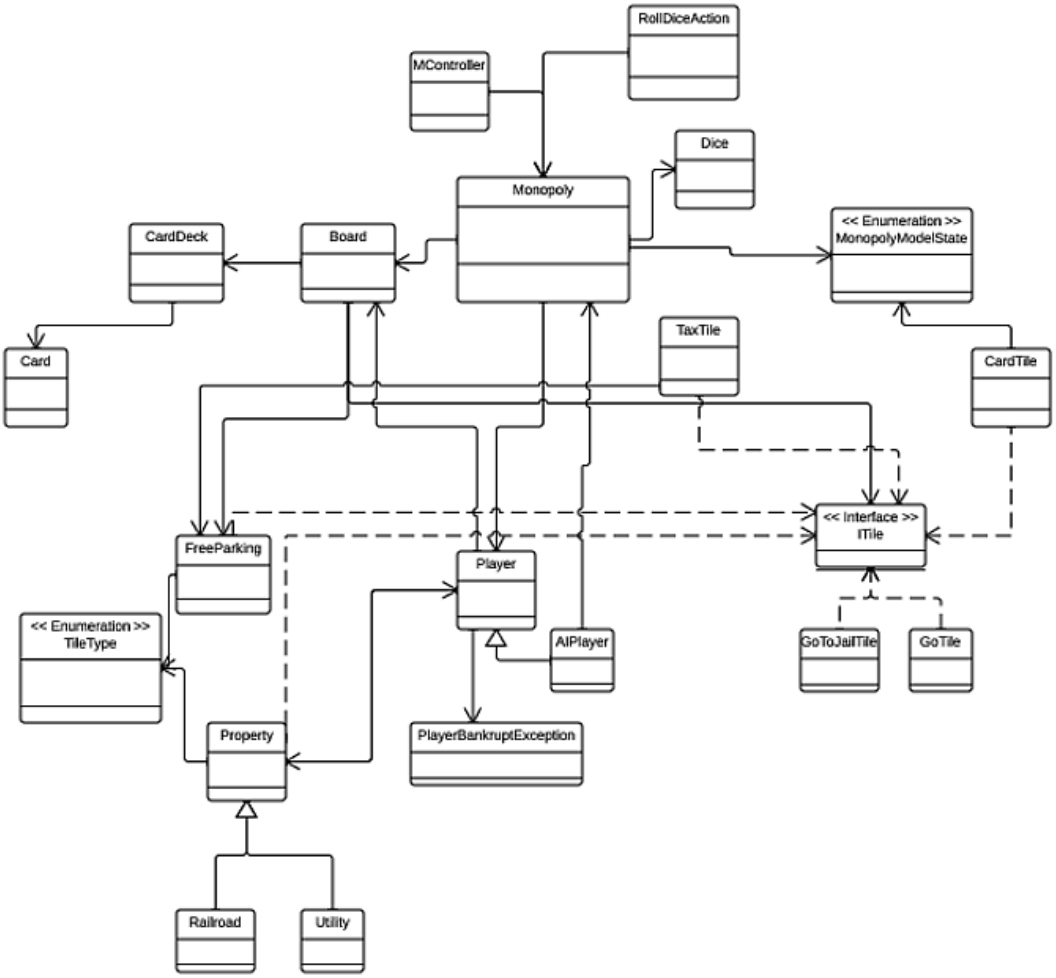
The frame itself encompasses four components, three panels, as well as one menu bar. The main frame uses a border layout to set its three panels, and sets the menu bar using the JMenuBar class. This JMenuBar class allows us to provide more options to the user, but keeping the main GUI as uncluttered as is possible. The three panels we have are the control panel, a panel to keep track of players money as well as containing two buttons - roll dice/end turn, and

the mortgage button. This panel uses a GridLayout to lay out information in a stacked layout. This panel has some setters to set some of the options available, and it uses very basic arrays of components to hold the components that you can set. The middle panel is the board panel. This uses a null layout, as we used boxes to represent each position on the board. This allowed us flexibility, as they can either have a horizontal layout, or a vertical layout, which allowed us to create the board. This has functions to set player positions, and remove players from the board. The final panel is the properties panel, which is set on each turn using the ITile interface. This means that the panel dynamically updates itself depending on what type of tile you land on.

Within the controller, we have several other classes that we use. First, we have the RollDiceAction class, which is the action listener for the main frame. This contains the behaviors needed to move the dice, as well as buying property. We have another listener that is attached to this main panel, which is the mortgage listener. This contains the logic to open one of two panels, both of which are custom JDialog panels. By extending the JDialog class, we are able to create highly customisable dialogs, as the dialog class essentially mimics a JFrame, but as a dialog, offering a lot of the different functionality that applies to a frame. This also allows us to get more functionality that would be possible just using a JOptionPane.

We also have a listener for the menu bar. This has all internal functionality, and opens the appropriate window as necessary. It the getActionCommand method to differentiate between different menu items, and then responds appropriately. This is helpful because we tried to separate up the functionality, so that if the agile method showed us we wouldn't get all features done, then we could remove this.

The Model



Our model is in the form of a finite-state machine. This means that there is one main function in the Monopoly class that is run with each new move that does different things depending on the state of the system. When a player lands on a tile and there requires user input, the model goes into a special state that can't be resolved until the controller runs a handler function. There are several handler functions that manage things from buy requests to the user landing on chance or community chest cards. This state machine allows the model to run independently of the external view, as long as there is a controller that gives the model feedback.

For our tiles, we used an ITile interface with a landOn() function and a getTileType() function. The landOn() function is called when a player lands on a tile and does whatever the tile should do. This also allows all elements in the board to be held in the same array in the Board

class. The way the Board class can distinguish the type of tile is through the getTileType() function. This function returns a different tile type depending on the class. There is one TileType enumeration value for each tile.