Design Manual: Housing Lottery

Introduction

Our team's Bucknell themed Monopoly very closely mirrors the traditional Monopoly game. The code can be broken down into a few major parts: the game itself, the board containing all the various spaces and their functionality, and the players. In order for the game to come together, all these elements must be connected in some way so that they can interact. The Game class within our code keeps track of all the instances of the players and the board with the spaces. The game class also instantiates a dice class, variables to keep track of the current player, the player statistics, etc. The Player class is actually a fairly minimal class, only keeping track of the player statistics and the movement of the player. All the spaces on the board (Properties, Academic Spaces, Tax, Chance, Community Chest, Go To Jail, Jail, Free Space, Go) are subclasses of the Space class. It is necessary for these all to be separate subclasses of the Space class because each subclass type has different that goes with it. In order to decrease coupling, our team has split up much of the functionality into the respective classes they belong to. This allows for the functionality to be independent of one another, and makes the JUnit testing come along much easier.

User Stories

Listed below are all of our user stories:

"As a customer, I would like to have Bucknell properties, so I can purchase them"

"As a customer, I would like to have a bank account so I can manage it"

"As a customer, I would like to have the board itself set up like a traditional monopoly board"

"As a customer, I would like to keep track of the properties I own so I can have monopolies"

"As a customer, I would like to have the non-property taxes be Bucknell related"

"As a customer, I would like to have the game handle taking turns, starting, and terminating the game, for a real game experience."

"As a customer, I would like the player to move around the board and interact with the spots on the board"

"As a customer, I would like to be able to roll the dice, in order to see how many spots I will go"

"As a customer, I would like to be able to view the board as I play"

"As a customer, I would like to have an interactive user interface for selecting options for purchasing property"

"As a customer, I would like to have my turn be interactive, roll dice, move space, show consequence"

"As a customer, I would like to be able to see my playing piece move around the board"

"As a customer, I would like to visually see what properties are available for purchase"

"As a customer, I would like to be able to visually see my player stats (what I own, my bank account etc.)"

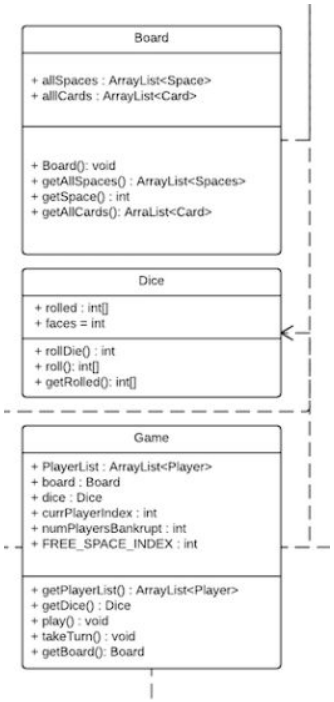
"As a customer, I would like to have fun visuals showing the dice rolls"

The program splits the user stories into two sections: the functionality and the GUI. In terms of accomplishing the general game functionality, the algorithm was used as follows:

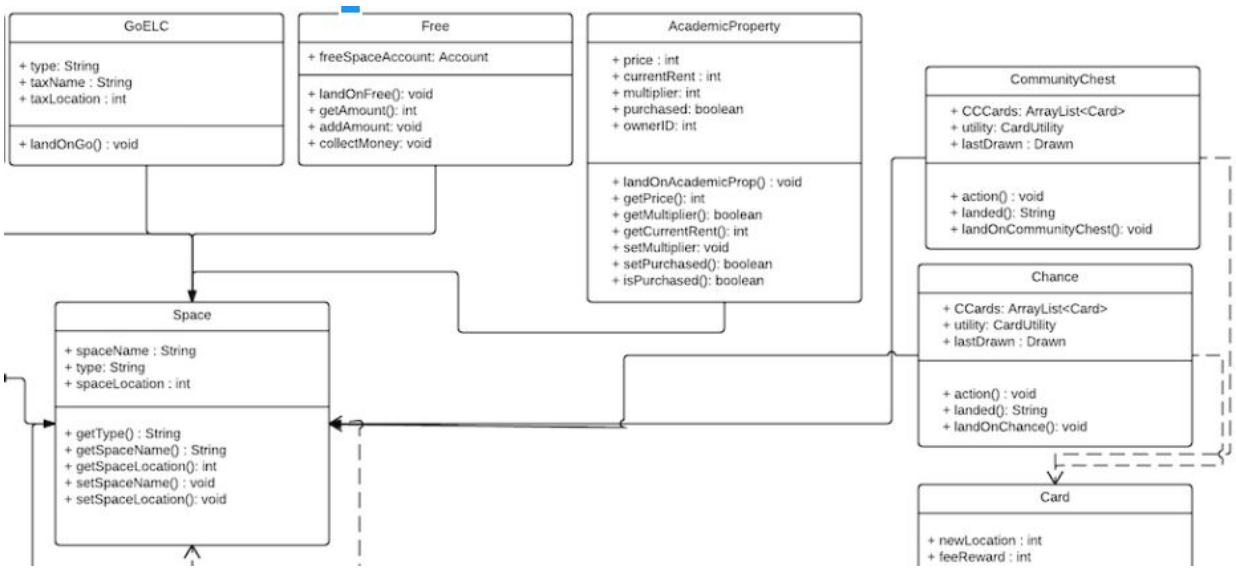
- Player rolls the dice within the game class, moves the correct number of spaces, and updates its current space instance variable
- The Game class references the corresponding Space on the board with the current space of the player, then checks for the type of subclass that the Space is, and performs the functionality of the given Space
    - In order to update the player statistics based on the space they land on (for example, deduct a certain amount for landing on a tax space), the Player ArrayList used in the Game class is passed as an argument along with an index referencing the current player.
    - In doing this, the code passes the actual player object, rather than a reference to the player in order to properly update and preserve the player statistics
    - Once the turn is over, switch to the next player's turn

In using this algorithm, the code accomplishes the aims of the user stories geared towards the functionality of the Monopoly game. The user stories that revolve around the visuals of the game were accomplished within Sprint 2 of our project cycle. We focused on having our user interface look exactly like a regular monopoly game. We utilized MVC design in order to do this. All player statistics were updated after each turn by updating the view from the model. Player dots were paint component circles that moved about the board according the the proper coordinates that corresponded to each space.

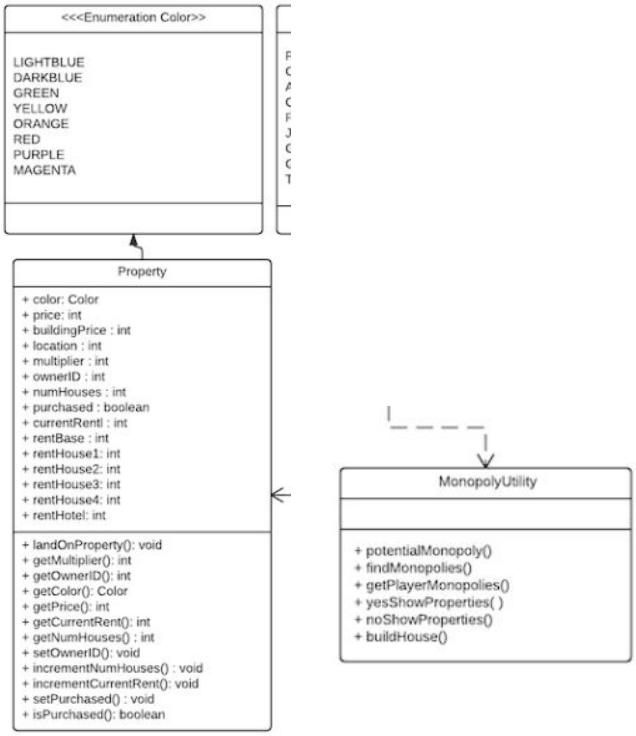
UML Diagram



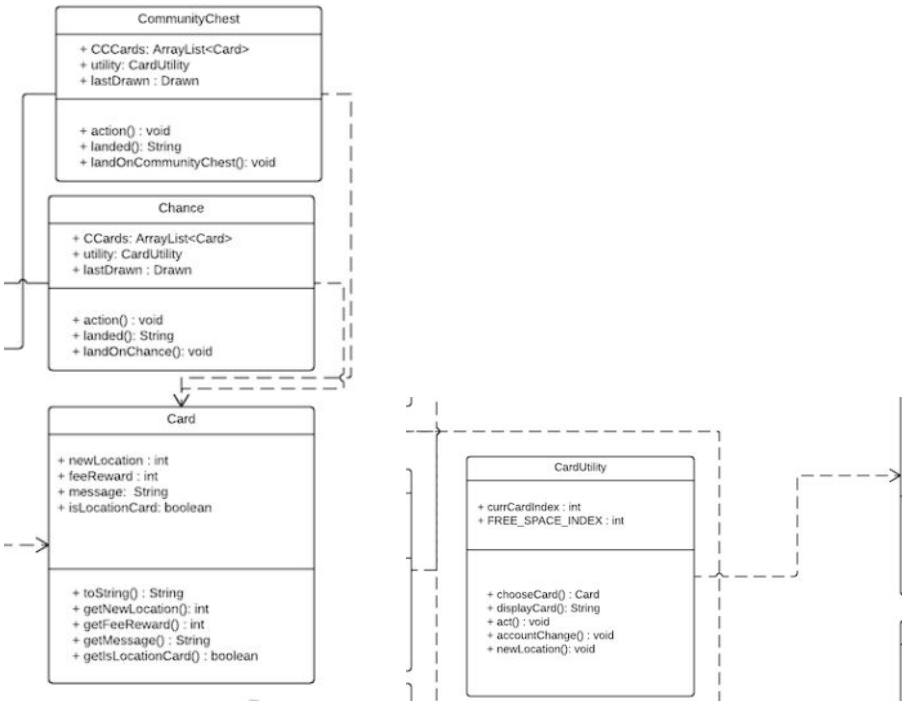
The heart of the our version of Monopoly is the Game class. The game class instantiates all the controlling instances of the Player class, the Board class, the Dice class, and keeps track of the current player turn, the current dice roll. The player objects are added to an ArrayList within the Game class and the turn is kept track of by incrementing the currPlayerIndex. The Board class contains an ArrayList of Spaces. The details of each Space (for example, the name, the price etc.) are hard coded within the Board class.



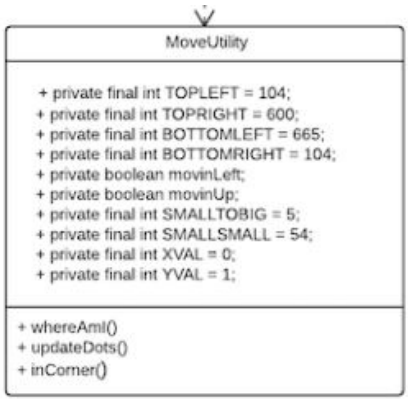
Space is a superclass, and all the subclasses of Space are the different types of Spaces shown (Properties, Academic Spaces, Tax, Chance, Community Chest, Go To Jail, Jail, Free Space, Go). Our group made the decision to make all the different spaces subclass because some of the functionality and instance variables would be different. But all subclasses of Space inherit instance variables necessary to keep track of all the spaces and that is the Space name, type, and location (0-39). Each of the subclasses of Space have a method that is landOn(), which takes in the PlayerList and currPlayerIndex as an argument, in order reference the player object itself rather than just a reference.



One of our goals of coding this game, was to allow the players to check to see if they have monopolies, and if they have any monopoly, have the option to purchase houses and hotels on the properties in order to increment the rent. The color group that a particular property belongs to was kept track of by the Color enum. The Monopoly utility class was a static class used in the Game class in order to check for the properties owned by a player by Color. The Color enum kept track of the total possible spaces within a given Color and how many of those spaces were purchased by one player.



Both the Community Chest spaces and the Chance Spaces contained an ArrayList instance variable of the Card class. In order to create a realistic "choosing" of a card from a deck of Chance cards or Community Chest cards, the static CardUtility class was used in order to "select" a card from the deck and consequently perform the action the card said, using the Player that was passed in as an argument to the method.



Finally, the MoveUtility class was used to keep track of the Paint Components of the player's pieces (or dots) moving around the board based on the hard coded coordinates that allowed the players to move in reference to these known positions on the board.