Aditya Kotak, Rahul Sarathy, William Wang

Mr. Peck

APCS, Period 6

26 May 2015

Final Specification for Monopoly Final Project

**Case Study**

In this program, we create a Monopoly Game using Java and its object orientated programming capabilities. When creating this Monopoly Game, we must consider the vast array of rules that dictate how Monopoly is played.

* *When can a player buy a property space?*
* *When can a player buy houses?*
* *When can a player buy hotels?*
* *When is a player in jail?*
* *When is a player bankrupt?*

All these questions (and more) affect the code going into the Monopoly game. In order to create a comprehensive Monopoly game.

The actual game play should be fairly intuitive with a player moving based on a roll of dice and do an action based on where the player lands. This can depend based on where the player lands. This can be a property, railroad, utility, jail, go to jail, tax, chance, community chest, or go. So this means that this program should create numerous objects for each one of those spaces but also an object for the player, bank and the cards.

In the end, the most basic summary for how this program works is let a player play until that player doesn’t have any money left. The player has properties and houses only if that player owns an entire group of properties. If the player is at negative money but has assets such as houses, hotels, properties, etc, this program will sell them off to keep the player in the game.

An important note: from the beginning of our project, we decided to not implement trading and auctions as per the traditional Monopoly because that was not how we played it in years passed. The point of our program was not to simulate the professional version of Monopoly but to make a game that we grew up with and would play without any extra hoops to go through.

**I: Structural Design**

*GameBoard Class*

|  |  |
| --- | --- |
| **Data** |  |
| GameBoard | Circular LinkedList (ListNode2) => ListNode2<Space> |
| Chance cards | Queue => LinkedList<Chance> |
| Community Chest cards | Queue => LinkedList<CommunityChest> |
| List of players | Queue => LinkedList<Player> |

*Player Class*

|  |  |
| --- | --- |
| **Data** |  |
| List of properties | List => ArrayList<PropertySpace> |

*Space Class (Abstract)*

|  |  |
| --- | --- |
| **Data** |  |
| List of players | Queue => LinkedList<Players> |

**II: Object-Oriented Design**

MonopolyGame is the main class that will run the entire program. It contains Card objects, Player objects, a Bank object, a GameBoard object, two Die objects, and Building objects.

A Card object represents a card in the game, of which there are two types, Chance and CommunityChest. It is able to be dealt, and has a general function to allow it to execute actions based upon what each card is supposed to do.

A Player object represents one player of the game. It has variables to store the player’s name, total cash, total number of houses, total number of hotels, the total worth (cash + property + buildings), list of properties owned, as well as whether or not the player is in jail, whether or not the player has a “get-out-of-jail-free card”, whether or not the player has money to make a purchase, whether or not the player has any cash left, and where the player is on the board. A player can roll dice, move, get information about the space it is on, draw either a Chance or a CommunityChest, buy and sell buildings and properties, pay and receive money from the bank and other players, and mortgage its properties if it has them and needs cash.

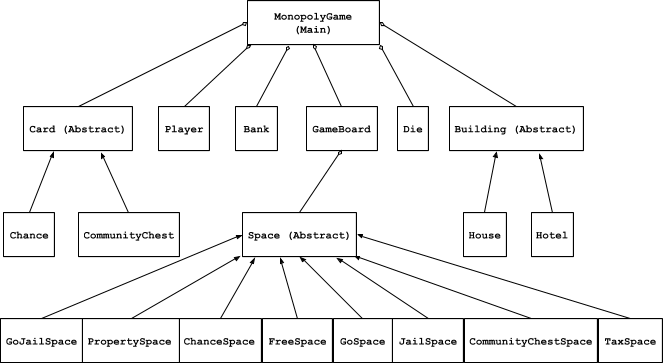
A Bank object manages the cash transactions of the game. Before the game, it gives each player a preset amount of cash to start with. During the game, the bank is able to pay players and get paid by players.

A Die object models a 6-sided die in the game. It generates a number from 1 to 6 inclusive, which helps determine events like player movement, how many times a player can roll, whether or not the player goes to jail, and whether or not the player is freed from jail.

A Building object represents a generic building in the game, of which there are two types, House and Hotel. Both types can be bought or sold, but you must have bought 4 houses before you are allowed to buy 1 hotel.

A GameBoard object represents the gameboard on which the players will be moving around and interacting with the Space objects that make it up. It is represented by a circular linked list of Space objects , allowing continuous circulation around the board.

A Space object represents one generic space of a GameBoard. There are 8 different types of Space object, each of which has a slightly different function, but all generic spaces have a name, a location on the board, whether or not it can be bought, whether or not it can have buildings on it, and a list of players occupying it.



**III: Detailed Design**

Documentation for all our classes below can be found in our submission under MonopolyGameDocs.

**Gameboard**

**Player**

**PropertySpace**

**AI**

**JailSpace**

**House**

These 6 classes are the most important for the building of our game so only these 6 are included in our Functional Specification. All other classes are some type of variation of these classes and have similar requirements and functional specifications.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Gameboard**

Requirements

The Gameboard class is required to run the main engine of the Monopoly game. This class will have one method that calls various other methods to do the actions depending on where the player lands. This method is also required to actually create the gameboard on which players will play the Monopoly game. Finally, this class is required to make sure appropriate actions can be taken to advance the game forward.

Functional Specifications

To manage the requirements, the Gameboard class will have various private methods that creates an object that needs to be represented in the board.

The Gameboard class will also make several reference calls to abstract methods in other classes in order to avoid having intricate methods all in one giant class.

The class will also have multiple arrays of Strings that contain the information for space and cards in order to create space and cards depending on the strings of that array. These strings are from the traditional monopoly game.

This class will also increment from a base property and house price depending on the side of the gameboard and location of the gameboard on a circular linked list.

**Player**

Requirements

The player class is required to create a player object that will play the monopoly game. This class must keep track of which properties this player buys and must also be able to buy houses and hotels.

Functional Specifications

In order to complete this task, the Player class uses a variety of helper methods from the gameboard class as well as other sub methods in its own class. It uses the global arraylist of properties to loop through to make sure that the owner for an entire group of properties is the same and it also buys a house depending on the prompt given by the end user.

This class will also have the definition for how mortgaging works and sets the order as follows:

Hotels, Houses, Utilities, Railroads, Properties.

If there is still no money after selling all assets, the player is declared bankrupt.

Overall, this class helps the development team understand how the game is being played and offers debugging tools since the output for actions taken in this class are outputted to the console.

**PropertySpace**

Requirements

This class is in charge of each individual property space and representing a general template for each propertyspace on the gameboard. It should be buyable and keep track of houses on this property. This class is also required to be able to be sold if necessary.

Functional Specifications

This class has the method act() which allows it to be bought by a player depending on the circumstances of the player. It will collaborate with the house class and player class to add houses to the propertyspace and have an individualized list for the houses. Property space also can remove houses if needed when a player is too poor to sustain that house. It will be sold based on recency and uses the player’s mortgaging methods to execute this task. Finally, PropertySpace will determine the pricing of the rent of the propertyspace should a player land on this space during the game. It will use a formula based on the number of properties owned int the group and the number of houses on that group (methods this class has because it can access the properties in its group using the group ID number). This allows for fair rent distribution across all players.

**AI**

Requirements

The AI is required to act as a player when a human is not available. It should be able to make a decision on its own whenever the normal player is supposed to make a decision.

Functional Specifications

* The constructor will be a type of player and has all the same properties that a player would have.
* The logic behind the AI is to keep on buying until there is no more money left. Once the AI has an entire property block, the AI will buy houses on the properties until on 25% of the money is left. This way, the AI is able to accumulate more and more wealth in the game.
* The AI will try to buy hotels first if possible in order to maximize the number of hotels in the game and to maximize the opportunity to make more money.
* If needed, the mortgaging will happen automatically as per the player class because the AI is a player.

**JailSpace**

Requirements

This class is required to make sure the player if in jail does not move out of jail without paying a fine, rolling doubles, or having a get out of jail free card. This space is represented on the gameboard and will have all aspects of space since it is a type of space.

Functional Specifications

This class will be called every time the player needs to go to jail.

* Go to Jail space
* Chance card
* Community Chest card
* Triple doubles.

While in jail, there will be three options presented to the user through the act method called in gameboard.

* Roll the dice to try to get doubles
* Pay the $50 fine
* Use a get out of jail free card

This method will make sure that the player attempting to use the card actually has a card and will make sure that only 3 tries are allowed to roll out of jail.

**House**

Requirements

The house class is required to represent a building to be placed on properties.

Functional Specifications

This class requires the least amount of code however is very important because it allows to advance the game forward. This class will also function as the hotel class because a hotel is not required when it can be represented as 5 houses.

The house class will be linked to property space and will directly influence how the rent is paid for that property.

All our classes in Monopoly

|  |
| --- |
| AI  Bank  Card  Chance  ChanceSpace  CommunityChest  CommunityChestSpace  Die  FreeSpace  GameBoard  GoJailSpace  GoSpace  House  JailSpace  ListNode2  MonopolyGame  Player  PropertySpace  RailroadSpace  Space  TaxSpace  UtilitySpace |
| Detailed information for each can be found in our documentation under the file MonopolyGameDocs within our submission. |

**IV: Testing**

A JUnitTest will have to test for several different scenarios in which we anticipate the most problems. We have provided strategies to test for some features which we believe to be most likely to contain bugs.

* Does a player receive money after passing GO?
  + Check a player’s balance before and after go and confirm that 200 has been added after passing GO
* Transferring assets after a player goes bankrupt
  + Confirm that if player goes bankrupt, assets are sold to (and owned by) bank until player has at least 200
* Converting from a regular property to a property with houses to one with a hotel
  + Confirm that while a hotel exists on a property, there are no houses on that property
  + Confirm that the proper rent corresponding to the number of houses is being charged
* Staying in jail for the correct amount of time
  + Allowing the user to leave if they pay bail, or if they roll doubles
  + Confirm that the player stays in jail for a maximum of 3 turns
* Correctly implementing a double roll (if doubles allowing for an extra turn)
  + Confirm that if person A rolls two of the same number, person A is granted an extra turn
  + Confirm that if person A rolls doubles 3 times in a row, they will be sent to jail

After extensive testing and bug fixing, we only had to make one modification in our game: the mortgaging system. While traditionally the player can choose whenever to mortgage a property, our system only allows mortgaging to happen when needed. Moreover, the engine will automatically instead of keeping it on hold, mortgaging will sell the property back to the bank. This is to incentivize the player to be more fiscal with his or her money.