Tommy Trieu
William Diment
CSCI 4448
Boese

Definitely Not Monopoly - Final Report

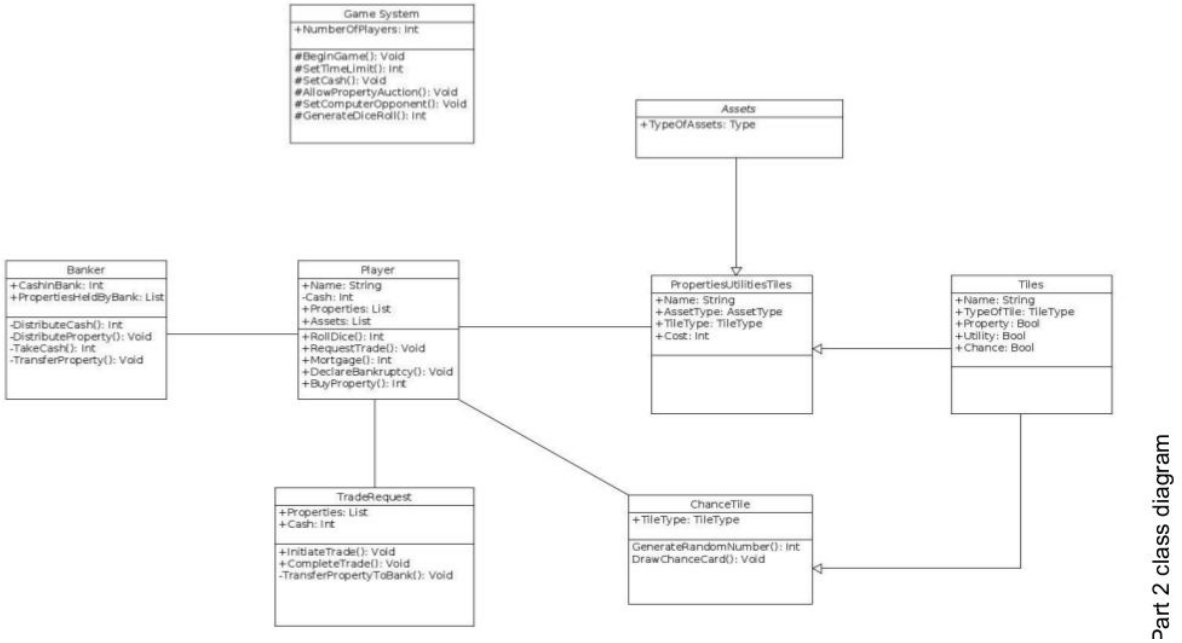
1.

ID	Description
US-001	As a player, I want to be able to play with up to four of my friends
US-002	As a player, I want to be able to see the entirety of the gameboard
US-003	As a player, I want to be able to roll the proper amount of dice so the game is fair
US-004	As a player, I want to be able to move to a new spot on the board according to the dice roll I received
US-005	As a player, I want to be able to see what other properties users own
US-007	As a player, I want to be able to trade my property to other players
US-011	As a player, I want to be able to vote on a time limit with the other players
US-012	As a player, I want to be able to vote on adjusting the starting cash with other players

2.

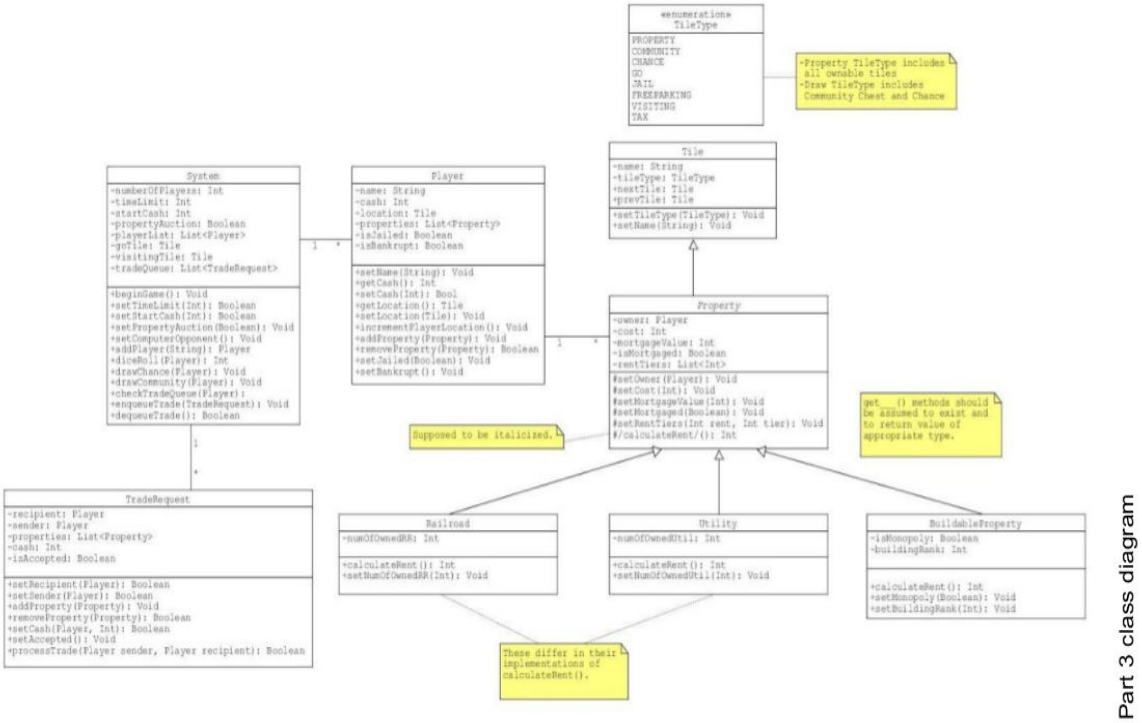
ID	Description
US-006	As a player, if I land on a spot with a property, I want to be able to purchase the property
US-008	As a player, if I can not pay rent or have enough property to sell, I want to be able to declare bankruptcy
US-009	As a player, I want to be able to mortgage my properties to the bank
US-010	As a player, I want to be able to land on the chance/community chest spot and receive a card from their respective piles
US-013	As a player, I want to be able to vote on whether or not properties can be auctioned off after a player declines to buy the property
US-014	As a player, I want to be able to play against a computer opponent

3.
Here is the class diagram from Part 2



Part 2 class diagram

Here is the diagram from Part 3 - both are available in more detail in the part2diagram and part3diagram files on our GitHub repo



Part 3 class diagram

There were a couple of big changes that we made when we originally designed the project to when we refactored, along with a number of changes throughout the implementation process. First was we originally had a class called PropertiesUtilitiesTiles that was going to handle all operations for properties and utilities tiles. We broke this class up into four more classes - a parent abstract class called Property, and then three classes that extended the property class - a BuildableProperty class (for houses, hotels), a utility class (Waterworks) and a railroad class. This was to ensure a clear distinction between each type of property that a player could own, and to ensure that we would be able to instantiate different methods across the three child classes that while they may be named the same, would have different functionality. In addition, we expanded a number of our other classes greatly - we expanded our TradeRequest system to more easily handle trades among players - we wanted to ensure a strict process for handling trades, and so we implemented a series of methods that would carry out this strict process. Compare that to our original TradeRequest class, where we had three methods - InitiateTrade, CompleteTrade, and TransferPropertyToBank. We also got rid of the Banker class - all banking utility would be handled by the system anyways, and it seemed extrenuous. During the implementation of our project, we added a GameBoard class to handle the graphical representation of the GameBoard - we should have anticipated needing this in the original process, and ended up taking (most of) the implementation time, where there are still kinks

needing to be worked out. The way the GameBoard class works is that it draws upon numerous Java libraries, notably the Swing library and the AWT library. The GameBoard class functions as an extension of the JFrame abstract parent class from the Swing library - when the class is first called, we initialize a new JFrame, and set a JPanel onto the frame. The JPanel that is set onto the frame is from a private subclass called MonopolyPanel that extends the JPanel class from Swing. It's originally initialized with default x and y coordinates (representing the starting positions of players) but when need to update the board to graphically represent the new coordinates of the players, we use a different constructor with a permission bit. We then overwrite the paintComponent method to actually draw the panel, and when the panel is complete, we add it onto the JFrame. When we need to update the board, we simply add a new panel to the frame, and then revalidate it. We needed a lot of help with this, there is a ResourcesUsedInProject.txt file that shows our citations for where we received help.

4.

We were not as explicit as using Design Patterns when initially designing the project, however when we refactored the project we kept Design Patterns in mind, and were able to make use of a number of design patterns. When implementing our property system, we identified that we were able to use the Factory Method. This was advantageous to us because a specific property could be any of the three-types of properties, and so we would be able to initialize the objects for these properties by passing in the requirements, and letting the child classes (BuildableProperty, Utility, Railroad) decide whether or not the requirements 'fit' those classes.

Another design pattern that we ended up using in the project was the Decorator design pattern in the GameBoard class. The GameBoard class initializes the monopoly game board to display for the player, but the monopoly game board isn't static - it needs to be updated each turn so that players can have a graphical representation of who is where, but the underlying functionality of the board doesn't change. We dynamically draw the position of each player on the game board using the MonopolyPanel subclass and add it onto the original JFrame that was instantiated. While the implementation is still wonky and not perfect, the Decorator pattern worked well here.

5.

First, the actual design of our project was integral to us trying to implement the system. We found ourselves referencing our Class diagram and our user-requirements constantly, and these were a constant source of help to us throughout the entire process. One of the things that we should have anticipated is the complexity of designing a graphical interface in Java: this should have been worked into the design process, and we should have taken it into account when designing the whole system. We were diligent in our initial design process, and the classes that we designed during that initial process worked well for our mechanics, but we should have taken a much longer look at anticipating the needs for our graphical representation of the game board. Also as always, we should have started implementing the project sooner, as

it proved to be quite complicated in the end - a more ‘finished’ product would have been possible.

One of the things that worked nicely though was the fact that we had a constraint on us in the form of the Monopoly game rules - we always knew what we had to work towards, and this kept us from going off the rails. One of the things that we can take away from this project is a respect for the design process - designing before implementing will always make a project easier, and if you change your design later on it is fine, projects always change and complications arise. This initial design process serves as an excellent springboard for a project, and will probably be referred to quite a bit out of necessity.