**CS 246 A5 BB7K Final Design Document**

**Hayden Holligan & Jesse Hoek**

Our project followed the same design as what we submitted in due date 1, for the most part. We created a Board class, Player class, and a Tile class. We used inheritance to create children for the tile class. The building children included: AcademicBuilding, Gym, and Residence. We had a separate class Block to identify different Monopoly Blocks.

**Board**

The graphics was created using a 2d array of characters. First an array of strings was created with each line being the board. These strings were then inserted into the 2d array of characters, row by row. Whenever we wanted to output the board, we would simply use a double for statement to output each row. This way we would easily change specific positions in the array (such as player positions or improvements).

The vector structure was used to implement coordinate systems. There were 2 arrays in the Board constructor representing x and y coordinates: one for where a player’s piece should go for each tile, and one for where each improvement should go for a building. Each player has their own designated space on each tile, the x position would be off-set by the number of players (for initialization) or by an iterator (when the players are being moved, after initialization).

The board constructor simply initialized all of the monopoly blocks, the tiles, and the buildings as well as the coordinate arrays and 2d array for the board.

Some functions we used were not thought of in the UML, such as functions to move players on the graphical board.

**Assignment Questions**

**After reading this subsection, would the Observer Pattern be a good pattern to use when implementing a gameboard? Why or why not?**

The Observer Pattern would be a good pattern to use when implementing a gameboard. The gameboard would be the subject that publishes data about the game. The game players would be the observers who receive updates about changes to the board . When a player moves spots or buys a property, all other players would be notified of that change.

**Suppose that we wanted to model SLC and Needles Hall more closely to Chance and Community Chest cards. Is there a suitable design pattern you could use? How would you use it?**

To model SLC and Needles Hall like Chance and Community Chest cards, we would use the Singleton design pattern. In Monopoly, there is only one Chance pile and one Community Chest pile; all players take cards from the same pile. There is only ever one pile at a time.

We would implement SLC and Needles Hall as singletons that have an stack of ‘cards’ that model the probabilities given in the assignment. Players would pop off the card at the top of the stack. If it is a Roll up the Rim Cup, the player keeps the card. Otherwise, the card would then be ‘pushed’ to the bottom of the stack.

We did not implement this DLC.

**What could you do to ensure there are never more than 4 Roll Up the Rim cups?**

Create an array of pointers to player objects, of size 4, which each space originally initialized as NULL. If the chance occurs where a player is given a cup, a void function will run to check if any spaces in the array are available. If there is a spot, that player will be a given the first space available in the array. If there are no spaces available, the function does nothing. When a player is in the DC Tims line up and they want to use their cup, a boolean function will run to check if there is a pointer to them in the array. If there is, the first cup that is theirs will be used up and the function returns true. If there isn’t a pointer, the function returns false and the player must pay, roll doubles, or wait in line.

**Research the Strategy Design Pattern. Consider both the Strategy and Bridge design patterns, would either be useful in implementing computer players with diﬀerent levels of diﬃculty/intelligence?**

The Strategy Design Pattern might be useful when implementing computer players with different levels of difficulty. The Strategy Design Pattern allows an algorithm’s behaviour to be selected at runtime. Each computer player you create will perform the same actions, but each action will vary depending on the level of difficulty selected for the player.

The Bridge Design Pattern separates the abstraction from the implementation. This is useful when a class and its implementation changes often. This design pattern doesn’t seem like it would be very useful in this situation since only the implementation changes, not the class itself.

We did not make it to computer players.

**Is the Decorator Pattern a good pattern to use when implementing Improvements? Why or why not?**

If you treat the improvements as different items, then the decorator pattern would be a good idea. Since they’re different items using the decorator pattern to add them on to the building would work very well.

However if you treat the improvements as identical items, then the decorator pattern is not necessary. The use of a decorator pattern is to be able to add multiple different additions to a Class. Since the improvements are all identical, this is not needed.

We simply treated the improvements as equals. A simple array of improvement costs as well as the number of current improvements.