Watopoly

Plan of attack:

For the final project, my partner and I decided to go with the Watopoly game design. After thoroughly reading the game instructions, we got started by planning the overall design on paper first. We went through the major components of the game, and came up with several basic classes that are needed, which include Board, Players, Squares and Bank. We also went over the design patterns studied in class and tried to see which one can be implemented. Right away, we realized that we could use the Observer Pattern for Board to notify Players and Squares during the game when the state for the game has been changed. After planning it out on paper, then we went ahead and completed the UML.

UML Explanation:

Board: Board is a class that manages the entire game process and is responsible for notifying Players and Squares to update themselves when the state of the game has been updated. Board has a vector of players that keeps track of all the players in the game and a vector of Squares that keeps track of individual locations of the board. When a new game is started, the board is responsible for initializing a new game board and setting all the players. During the game, the board keeps track of whose turn it is, notifies the player when a building has been purchased, range auctions, and get mortgage on a building upon a player’s request. Another important feature is that board is responsible for getting a saved game from a file and setting the game state (board, players) according to a saved game file. Also, it’s responsible for saving a game at any point and saving it to an output file.

Player: Player is a class that stores information of a single player in the game. It’s most important feature is to update itself according to the state of the game when notified by the board. Inside, the player class, it stores its current location on the board, a vector of all the buildings the player owns, and the total amount of money the player has. After being notified by the board, the player will update its state by either pay money, receive money from others, moving to a new location, buying a building. Also, during every turn, a player will check if it is facing bankruptcy. If this is true, player will notify board for auctions of all its properties after it has mortgaged all its properties.

Squares: The Squares is an abstract class that has two sub-classes that inherit from it, which are Building and nonProperty classes. Buildings can be purchased by players during the game, while nonProperty is similar to Chance and community Chest cards in Monopoly. Buildings is also an abstract class with AcademicBuilding, Gym, Residence classes inherit from it. The reason why we decided to break Buildings into three subclasses is because the fee a player is required to pay when land on the three types of building is calculated differently. The nonProperty class is also an abstract class which classes including, SLC, GoToTims, GooseAttack, OSAP, PayTuition, NeedlesHall, Timesline and COOPFee inherit from it. Those classes all have their own special implementation that will require the current player who has landed on the place to pay certain amount, get certain amount or be directed to certain place on the board.

Bankrupt: Bankrupt is a class which can be called by either the player or the board. The player calls the bank to mortgage all its properties when facing a bankruptcy in attempt to raise enough money to pay the loan. When the attempt to raise money fails, the player will notify the board for bankruptcy, which in turn the board will notify the bank to set auctions on all the properties the bankrupt player owned.

Display: The display class is like a communicator to the human players in the game by displaying messages about possible actions the players could take, or notify the human players where they have landed on the board.

Questions in the assignment:

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

Yes, the Observer pattern would be a good pattern because it enables efficient data transfer between the Board, Player and Squares. Also, the Observer pattern allows the Board to attach and detach a player at anytime during the game without modification to Board or player. Also, when the state of the game has been updated (when purchase made, or trade made), board could notify the players and baord to update it fields.

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

No, the Decorator Pattern is not a good pattern to use when implementing improvements because when an improvement is added, only the tuition price raised for other player landing on it but do not support any other functionality. For a decorator pattern, it adds new functionality to existing class without actually alternating the original structure of the class. However the only thing improvement does would be to increment the tuition, which can be done in the original class. Generating a new class could seem unnecessary. Also, there is no pattern for the amount of tuition incremented after each improvement, and tuition increase various among different buildings, thus a decorator would be hard to general different buildings with different amount of increment each time. Thus a Decorator pattern would not be a good idea; just manipulate the original Buildings class to increment the tuition amount every time would be easier.

Suppose 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?

If we are to model SLC and Needles Hall more closely Chance and Community Chest cards, the visitor pattern can be used. The reason for this is because the visitor pattern can separate an operation from an object structure on which it operates. The Chance and Community Chest cards have different functionalities, Chance cards direct a player to another location while Community Chest cards give or take way certain amount of money from a player. They require different operations on a player object, thus a visitor class can be implemented. When a player picks a Chance or Community Chest card, they would call the corresponding overloaded visitor function in the visitor class to do the right manipulation depending on the card picked.

Schedule and responsibilities:

My partner and I plan to finish the UML by July 13th, and start writing the actual implementation of the game beginning July 14th. We plan to start off by completing all the .h files for every class together. By starting with the .h files first, it helps us to map out all the classes and the relationships more clearly. Our goal is by the 20th, we should have all the classes complied and have the most basic functions of the game working properly. The basic functions we hope to get running by the 20th include initializing a game, players taking turn rolling the dice, moving according to the number rolled, buying properties, and trading properties. From the 20th to the 23rd, we plan to get all the special features of the game running and tested, including DCTimsLine, SLC, NeedlesHall, bankruptcy, auctions ect. By the 24th, the day before the final due day, we should have our entire program functioning properly with all the features working. We have set the 24th as a testing day, where we will try to test all the edge cases for the game and fix small bugs if we find any.

The way we have decided to split up the work is that since we only have two people on the team, we tried to split up the work amount evenly between the two of us. We decided to write all the .h files together, so we could decide together which .h files need to be included in other .h files, while others only need forward declaration. Doing this together could prevent unnecessary inclusions and avoid circular dependency during compilation. After the .h files, we plan to implement the Board class next. The board class is the subject class in our program who will be responsible for notifying and communicating between the Players and Squares. This is a very important piece in our program that sets the overall structure of our game, thus we decided to implement it first. We would implement the Board class together, so both of us get the basic idea of the overall structure of the game. After we finish the implementation of the Board class, we divide the work, with one of us doing the Squares class, and the other person would be completing the implementation of both the Display and the Players classes. We hope to get our own parts compiled by the 20th or the 21st, and we can try to compile everything together without the Bank, and the special squares on 21st after writing the main, to see if we are on the track. If things go well, on the 22rd, my partner and I will write the Bank class together. The reason why we decide to implement this class together is because we found auction is a very hard implementation that involving various classes, thus we try to prevent mistakes by writing it together. Then on 23rd, all the code writing should be finished and all classes should compile individually. Then we will compile the basic features and test it out thoroughly and add the special features in one at a time to make debugging easier.