Overview:

1. Introduction of various classes:
   In out Monopoly project, we have created the following classes.
   (See complete fields and methods in UML)
   - Player: A player object who will record information such as name, piece, number of gyms, number of residence, and number of academic buildings they own for each Monopoly Block.
   - Board: A board is created by controller at the very beginning of the game. It contains all the player objects, cells, and dices. This board object is also responsible for the main game command execution. Therefore, it contains method such as save, bankruptcy, trade, mortgage and so on. (I will discuss the relationship between different classes in detail later in this document)
   - Dice: Dice object is only responsible for generating a random number between one and six and then there is a method for other class to call in order to get this random number.
   - Cell: Cell class is the base class of each single cell/square on the game board. It have the basic ability for saving a list of landers (Player pointer), setting/getting index of this cell.
   - Ownable: Ownable class is a subclass of cell which represents all the cells that can be bought by players. Therefore, it contains filed such as the basic cost of purchasing this property, and the owner of this property (Player pointer).
   - Unownable: Unownable class is also a subclass of cell which represents all the cells that cannot be bought. This class itself does not have any fields but only has a pure virtual method called effect.
   - Gym & Residence: These two classes are subclasses of ownable. They are very similiary so I list them together for convenience. In these classes, they both have the ablity to set rent (the money that lander need to pay to the owner) based their game rules. They also have a charge function which can charge the rent from landers. Both of them has a notify function which will notify the owner of this cell that they are bought so that the information in my player object can be upgraded. The only difference between these two classes is that for gym, it has a stack allocated dice objects because according to the rule, player need roll dice for payment.
   - Academic: This class is also a subclass of ownable. It represents all the academic buildings cells such as ML, DC on the board. This class has the ability for owner to improve, therefore, it has a field to save improvement cost and a filed called totalcost which is the total value of this building (total improvement plus basic cost). Moreover, it has a list of neighbors which are all the academic building objects that shares the same Monopoly Block Number. Thus, if this building has been bought and caused monopoly effect, it will notify all its neighbors to

become Monopoly state (set Monopoly filed as true, it was false by default). Finally, similar to Gym and Residence, academic building also has the ability to set rent (tuition fee) and charge function based on their own logic.

- Tuition, TimLine, SLC, OSAP, NeedlesHall, CoopFee, GotoTim, GooseNest: These classes are all subclassed of Unownable. They all have a Effect method which will have some effect on the lander based on their own rules.
- Controller: Controller is responsible for read user commands such as save, roll, trade, buy/sell improvements, buy buildings, and so on. It is also responsible for initial cells' and players' data from either a saved file or a new game. Finally, it will send state between the game part and the display part so that any change in the game can be shown on the text display.
- View: An abstract of all the display classes
- TextDisplay: It is a subclass of view, therefore, it can receive state change information that is passed by controller.

2. Game logic:

(This section will only introduce our game part but exclude display part)

The main function will send a Boolean value to controller to tell it start from a saved file or name game. My controller has a pointer to the board object so that it can call all the execute functions that were implemented in the board class. As soon as my Controller constructor is called, it will created a board on heap. Inside my controller, it will read in player's name and piece if it starts from a new game and then call the initial player function from my board class. If the game start from a saved mode, my controller will open the file and read line by line and initial players and cells by calling my initial functions from the board. After the game has been fully initialized, the controller will ask the player to type in commands. Based on the commands that the player typed in, the controller will call the corresponding function from the board object. And, every time before read in a new command the controller will also check if the game has a winner or not and it will exit if a winner is found.

In my board class, it has an array of all the pointer to Players that are saved on heap. It also contains an array of 40 pointers to Cell which are allocated on heap. And two heap allocated dice objects and a pointer to the controller. The initialization function in my board class will initial all the cell objects with its origin state. Then, the controller will call both initialPlayer and initialCell function to update the state based on the saved file. There is a deletePlayer function which will delete a player object and set it to NULL, when this player has declared Bankruptcy successfully. Also, there is a currentplayer filed in my board class which is the current player's index of the players array. Therefore, the board will know who is the current active player and my next function will update the currentplayer value very time when next command is executed.

In each turn, my makemove function in the board class will responsible for make movements for a player. The board class and cell are used Observer Pattern here. My board is a subject and every time a movement is happened, it will notify the corresponding cell to set lander field. It will call the rollDice function and get the face value of two dices and then send player to the new position. It will also notify the new position cell that a lander is arrived so that it can has a pointer to this lander, meanwhile, it will delete the lander pointer from the old position cell. However, my makemove function also need to check if the player is in Tim Line. If the player is in Tim Line, it will follow the rule of Tim Line instead of just making movements. After the player go to a new position, my board will check if this land is ownable or not. If it is ownable, it will check if the property has been bought yet. If no one has bought it, it will ask the lander if he/she wants to buy it or not. If the player chose to buy, my board will call the current cell's setter function to update its state such as set an owner pointer, change ownerExist filed to True. It will also call notify functions based on its type (whether it's an academic building or gym or residence). If the player do not want to buy this property, it will execute the auction function.

On the other hand, if the player landed on a property that has already been bought, it will call the current cell's setRent function and then call the charge function to charge the lander with the rent/tuition fee. If the player landed on an unownable cell, it will trigger the current cell's Effect function and passing current player's pointer as an argument.

For my Academic class, we used Observer Pattern. Every time an academic building is bought, it will notify the Owner to upgrade its Monopoly Building list and then it will check if the current Monopoly Block's state has been changed or not. If it changed, then it will notify all the neighbors (which are the academic buildings that shares the same Monopoly Block Number) to change their Monopoly state as well.

For implementing improvements, I just simply added some fields in my academic class which are number of improvements and total value (improvement value plus basic cost) of this property. Therefore, every time the player wants to buy/sell an improvement, it will simply increase/decrease the number of improvements and its corresponding total cost. And for my setRent function in academic class, it will calculate the rent according to a pre-implemented rent array. That means when I created my cells, I will implemented an array of rent and the index of the array represents the number of improvement. Therefore, every time, when I call the setRent function it just need simply find rent in this array based on the number of improvements field. Finally, my charge function in here is just decrease the lander's cash and increase the same amount to the owner's cash field.

For my Gym and Residence class, the logic is similar to academic class except they do not have improvements. And they will notify the player (owner) to update their number of gym and number of residence fields when they are bought.

For my Cell class, Ownable class, and Unownable class are all abstract classes only. For my GooseNest, and Tim Line classes, they do not have any actual effect on the lander but just simply print a message to stdout. For Coop fee and OSAP, their effect just simply change the lander's cash field.

For tuition, I might need to calculate the total wealth of a lander, therefore, there is a function called findWealth in my board class which loop through all ownable cells and check if the cell's owner is the lander then it will add its property's total value and the lander's cash all together which gives the lander's total wealth. This is also the reason why my cells class need a pointer to the board for calling this function.

For SLC and NeedlesHall classes, they will generate random numbers based on a certain probability and then apply the effect on the lander the same as other unownable classes.

3. Display Logic:
   We used MODEL-VIEW-CONTROLLER pattern for our game. We used a subclass of view to implement the function that can draw the game table(which is textdisplay), and set view to be an abstract class. So that if we want to add some more displays we can add it directly without change any code I have been written (reusability of the code). We also decide to add a subclass, which named displaycell to display the single cell of the board. Then, we can use single cell to build up our whole board.

   We implemented three notify functions which will be called by controller so that the textdisplay can be notified after we changes piece or number of improvements. Each of notify function will call the draw function in the textdisplay to draw a new board after the player change the piece or number of improvements. In this way, players can keep track of what they had improved and where they moved to through the board.

Question:

1. Observer is a good pattern for implementing the game board. The board is a subject and each cell on the board is an observer of the game board. Every time when a player rolled a dice, it will notify all the cells that the player is going to pass and then for ownable buildings is also a subject itself. Every time an ownable building is bought and cause a Monopoly Effect it will notify its neighbors to change their Monopoly State.

2. We think use template pattern is suitable for implementing SLC and Needles Hall object. They are both the same type of class and the only difference is their effect so we can use template pattern to handle it.

3. In our solution, we have a field in our player class which is the number of Tim cards they owns. And in our board class we have a method which adds each player's number of Tim cards together and make sure it is less than four. Moreover, in our cell class, we have a pointer to the board class so that every time when we need random a card, we will call this method from board to check if we can create a new Tim card or not.

4. Decorator is not a good way to implementing improvements. We think we can just add a few fields in the Academic Building class to handle this improvement situation. For example, we will have an integer filed to count the improvement level of this building. Therefore, every time when a player buys an improvement the value of this building will increase and meanwhile when someone landed on it, we will get the tuition from a pre-implemented rent array. That means when I created my cells, I will implemented an array of rent and the index of the array represents the number of improvement. when they sell the improvement functions in the Building class will decrement accordingly.

5. After we did this team project, we learned it is very important to make proper comments and follow a standard and consistent pattern. It is extremely hard for other people to read someone's code if he/she did not do these two things. If I worded alone, I learned it is important to design the program properly at the beginning by drawing a UML. It helps to track the state and make the logic clear.

6. If I could start over, I would make a better design and make my code more efficient.