Plan of Attack – BuildingBuyer Project

Project Breakdown

See UML diagram in uml.pdf

All components of the monopoly project are outlined in the ump.pdf and important classes are described in the following section.

Controller

This class will serve as an interface to the monopoly game for main() functions. The main program will ask the Controller object to start a game, and all further interactions between the user and the program will be handled by methods in the Game class. This controller connects the model part and the view part. This organizes and modularizes the program better, which allows for better parallelization of tasks, and encapsulates implementation details for the Monopoly game.

Mode

Game

Game class act as the central of the program. It will contain:

1. An array contains 40 buildings
2. A pointer to controller
3. Trace the number of available cups
4. Interact between player and buildings

Player

The player class records the all the important information of the specific player. For example, the location field shows which building the player currently on the money presents how much money he owns and the properties field, which is an 40-length array, shows which building he owns. This class also should be able to deal with all possible commands from the player and notify the other classes when it is necessary.

Building

Building is the abstract class for the OwnableBuilding and UnOwnableBuilding. The superclass will contain methods and fields which are common to all buildings while the subclasses will contain specific implementations for each building type. All the buildings have special effect on the players who step at them, most of them requires a payment from the player.

OwnableBuilding

OwnableBuilding is the abstract class for those buildings can be possessed: AL, ML, ECH, PAS, HH, RCH, DWE, CPH, LHI, BMH, OPT, EV1, EV2, EV3, PHYS, B1, B2, EIT, ESC, C2, NC and DC. Also, there are three subclasses of this type: improvable building, gym and residence. In this class we will specify the monopoly collections and interact with players. Once a player locate on this kind of building, the need to pay some money to the owner of this building.

UnownsableBuilding

UnownableBuilding is the abstract class for those buildings can't be possessed: CollectOsap, DCTimsLine, GoToTims, GooseNesting, Tuition, CoopFee, SLC, NeedlesHall and RollUpTheRimCup. They act similar to the OwnableBuildings except that the money will be collected by the bank, not any of the players.

View

View is an abstract class which is used to display the game to the player. It receives information from the model that it uses to generate output to present the game to the user.

TextDisplay

TextDisplay contains a gameboard which consist of Cells, which will be used to present the game every time a move is made and also guide the player by print on the screen. It can receive information the player and building by the controller.

Cell

Each cell corresponds to the building in the Model part and presents it on the screen. There are two types of Cells, one is improvable and the other is not improvable. There is a slightly different when we present them since we need to show the level of the improvable buildings thus we make some space for this purpose.

Software Design Patterns

We use observer pattern most in this program. Once we receive the command from the player, the building, the game, and the view class will all take corresponding actions. In other words, once some change has been made, it will notify its observers. The method connect between different parts of the program is the notification. Also we use singleton pattern here, each monopoly is a collection of buildings and should not be changed.

Schedule Planning

Estimated Date	Goals	Who's Responsible
Fri, July 17	Plan of Attack and UML	Both
Sun, July 19	Headerfiles for all classes Implement Building class	Both
Mon, July 20	Implement the View part of program	Yingluo
	Implement Game and Controller	Jianan
	Implement Player	Both
Friday, July 22	Executable game between Human	Both
Sunday, July 26	Documentations for the game	Both
Tuesday, July 28	If time allows write GraphicDisplay for the program	Both

Answer to Questions

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

   Observer pattern should be a good pattern to use. Use the Building as each cell of the gameboard and the players are the observers of them. Once a player step on a building, the player will notify the game and the building and then have some relevant effect on the observers automatically, thus the Building is the observer of the player. Also, the player is the observer of the Building since the Building will have an effect on the player. Almost all the classes are the observer of each other and they are highly correlated.

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

   In the Game class we make a field names cupAvidable to maintain the current number of cups. Every time when we are going to generate a cup, if there is already 4 cups, we stop this process

3. 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?

   We could use decorator pattern. It does not need to make change of the other objects in the same class. We use this component to generate random numbers and then generate the probabilities. We will implement a method to make a random move for SLC and for the Needles Hall we will implement a method to increase or decrease the amount of money the player owns. These methods will follow specific rules.

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

   Decorator pattern should be a good pattern to use. Improvements is a characteristic of the Building class. We only need to add this behavior to the Building without affecting the behavior of other objects from the same class. Once we buy or sell an improvement for an object, we just need to call relative methods on it.