Final Design document for Watopoly

By Anzhou Hou, Yifan Hu, and Enze Xing

1. Introduction

Watopoly is a game based on the rules of Monopoly, but with its background based on the University of Waterloo. The game requires 2 to 8 players, with each player assigned to a name and a symbol. When the game begins, each player has $1500 and rolls 2 dice and moves corresponding steps. When arriving at a building, a player can buy an unowned property, but if the property has an owner, the player must pay a certain fee to the owner. There are also non-properties which have unique features. When a player cannot afford to pay money, he must declare bankruptcy, and the game ends when only one player stays. Our group has successfully implemented most of the features in the game with object-oriented design principles, therefore our code has good maintainability and allows for extension of features.

2. Overview

The main design pattern we decided to use is the MVC pattern. Since this game requires users to type in command, and each command has different effects on the game data, we designed a CommandInput class which serves as the controller and deals with inputs from users. It has a field which is a pointer to the Game class, which is the class that manipulates all the game data, so that it can send different requests and calls different methods of the Game class corresponding to different commands. Next, we designed the Game class. It needs to store all the data about all the buildings and players in Watopoly. To achieve this, we designed a base class building to represent all the buildings, and the base class has two derived classes, property and non-property. The property class represents all academic buildings, gyms and residences since these buildings can be bought by players, and the non-property class represents all buildings that cannot be bought. Each player is also represented by a player object, which records all data for a player, including money, properties, Roll up the Tim Cups, and two dice for rolling. The game class has a vector of buildings and a vector of players, so that the game class stores all the data in

a game, and by calling the accesser of these two classes, the Game class can access all data in the game. Also, when we save a current game, we just copy all the information in the Game class into a file, using the given format; when loading a game, we just construct a new Game class with the data in the file and attach it to a CommandInput.

Finally, the View component of the MVC pattern is the Board class, which mainly serves as printing the current board of the game and all the messages that the players need to know. The Board class serves as an observer of the Game class, and the Game class can notify the Board when needing to print messages or the map. Also, the Game class notifies the Board when a player moves or buys/sells an improvement, so the board can update these information on the map. The Board also contains a pointer to the CommandInput in order to inform players when the game went into special mode, such as the auction mode. Such mode often requires interactive response between the players and the game, so both the CommandInput and Board class need to work together. Overall, our design of Watopoly shows a mature adaptation of MVC, which allows us to manipulate game logic, read input and display information all separately. This facilitates our design procedure, and it ensures encapsulation so that players won't mutate game data when typing commands.

3. Design

During the design and implementation of Watopoly, we have encountered multiple challenges, and we have used different design patterns that were taught in the class. The first challenge is how to implement the features for different buildings, since they all invoke different situations when a player arrives. The solution to this problem is using the visitor design pattern. Each player object serves as a visitor to a building, with a visit method that takes in a building as a parameter. This visit method will in turn call the accept method of the building, which is a virtual method that is overridden by each child class. For example, for the property class, the accept method will ask the player to buy the building if it doesn't have an owner yet. If it has an owner, the accept method will calculate the amount of fee that is charged to the player, and call the player's giveMoney method to give money. The visitor pattern

makes our code more efficient to implement and allows us to extend such patterns to any new buildings.

Another design pattern we used is the strategy method. When implementing the SLC and Needles Hall, their accept method should vary based on different random generations. Therefore, we have implemented different suits of accept methods to accommodate different situations. For example, when arriving at Needles Hall, a player is sent to different places based on probability. By instruction, there is a ⅛ chance that a player is sent 3 steps backwards. Therefore, we would randomly generate a number between 1 and 24. If this number is between 1 to 3, we would call the strategy to move the player back 3 steps, etc. The strategy design pattern enables us to incorporate different algorithms in one function, and it can also be extended to add new features.

4. Resilience to change

One of the advantages of our design and implementation of Watopoly is that it can be easily extended to add new features. First of all, our MVC design pattern separates the input, the game data, and the output sections. This ensures the encapsulation of our project. For example, if we need to design a new board theme, we just need to change the function of drawBoard in Board.cc, but we don't need to change anything in the Game class and the commandInput class, because the game simply calls the drawBoard method when needing to display the map. Also, if we need to implement a new command option, we just need to change the commandInput class, but do not need to change the Game and Board. Our use of design patterns also ensures the facility of implementing new changes. For example, if we need to change the SLC's probability of moving, we just need to change the algorithm we wrote in the accept method of SLC, but do not need to modify the moving procedure in the Game. Also, if we need to add a new building, we just need to add a new class to either property or non-property and design its accept method. Overall, the resilience of changes in our project shows that we have understood the principles of object-oriented design, and shows our ability to break down the project into different sections, with each section working on a specific task, and ensures their

independence of each other by encapsulation. This makes our project have low coupling and high cohesion.

5. Answers to questions
    a. Would the Observer Pattern be a good pattern to use when implementing a game board? Why or why not?

        Answer: The Observer Pattern is feasible to implement as in our design, the Game class is observing and controlling the procedure of the Watopoly game. However, personally during our design, it becomes complicated as it conflicts more or less with our MVC Pattern design. Because under our design, the Game class serves as the manipulator of the data of this game, thus it is tedious to add another observer or make Game the observer. Therefore, we did not add the actual abstract Observer base class. In our due date 1 submission, we believed an Observer Pattern is a good pattern which indeed is. However, we did not consider if it's proper for our implementation specifically. During the actual implementation, we found out that it is relatively tedious and unnecessary to implement in that way.

    b. 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?

        Answer: A Template Method Pattern is a proper choice. Since the Chance and Community Chest cards contain a lot of similar cards who vary a little from each other in detail. So we can create general abstract classes for the similarities and each subclass of it can add their own features. Such a way of implementation satisfies the Template Method Pattern. Prior to our due date 1 submission, we regarded Bridge Pattern as a proper method however after consideration, we believe a Template Method Pattern should be more appropriate.

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

Answer: Not at all. Because the only place where Decorator Pattern seems appropriate is the improvements of the properties. However, it becomes tedious to implement using Decorator Pattern as the improvement for each property is totally different (hard to identify how much money to add) and when involving operations such as selling improvements, it is hard to trace how much money to give back.

6. Extra credit features

During the implementation of the game Watopoly, we added in additional commands to make the game clearer and execute more logically with more options provided to the players. The additional commands we added include: “yes”/ “no”, “sell”, “A”/ “B”. The “yes”/ “no” commands allow people to make the corresponding choice in different situations such as when deciding whether to go bankrupt and whether to purchase a building. The “sell” command helps players to make an easier choice at the time they don’t have enough cash to pay for something. If they simply type “sell”, they can sell the corresponding improvement without tediously typing the “improve” keyword. The “A”/ “B” option allows players to choose the method to pay the tuition as there are two different options to select.

7. Final questions

a. What lessons did this project teach you about developing software in teams?

The key of developing software in teams is to firstly decide approximately who will be responsible for which part and keep in touch with the team members in case the modules written by other people lack some features and cannot satisfy your own needs. Secondly, keeping the codes clear and well documented can help others understand your part better otherwise your code cannot be used by team members as their APIs or helper tools.

Thirdly, you need to trust your team members in terms of believing in their abilities of understanding and implementing the code. Therefore, working in teams is totally different than working alone as communication and trust becomes more important than individual coding skills.

b. What would you have done differently if you had the chance to start over?

What we would have done differently is starting working earlier than I did. We started the project more than a week prior to the deadline as we thought there was enough time. But the working period got stretched over several days which reduced our efficiency. Thus, we should have kept our attention just on the project without being distracted until we finished. In addition, we might try some different design patterns other than the ones we are using because it would be interesting and useful for us to have different developing experience.

8. Conclusion

To conclude, our actual implementation is quite similar to which we designed initially. However, there are several cases where we made changes to our plan. Firstly, we did not use the Observer Pattern (not implementing the base class) as expected because it conflicts our MVC a little so we omitted that part to make our project cleaner. Secondly, we added some additional methods for the classes to meet our expectations as during the implementation we encountered problems of passing data from one class to another so we have to write the methods to make the communication more fluent and consistent. Similarly, we added a few private fields to some of the classes in order to keep track of some of the needed properties such as the number of times each player has been consecutively in the Tims Line, etc.