Final Project Design Document

Isobelle Wang and Zihe Zhou

Department of Mathematics, University of Waterloo

CS246: Object-Oriented Software Development

July 26, 2022

Catalogue

Introduction

Our team decided to create the Watopoly game for our cs246 final project. After our carefully planning, designing, coding, modifying, and updating, we have successfully created the Waterloo version monopoly game. Even though we met a variety of obstacles, such as the first version of design is too tedious, the connection between components is not fluent enough, time constraints and so on, our team work together and made a great team. After completing the mandatory features, we upgraded the project again and again and carried out UI design and various design patterns as well to make it more user-friendly and entertaining compared to other monopoly games. Our games can be played by 2-8 players simultaneously. It will be played on a 40square board including a variety of buildings in Uwaterloo. Each player slacks off certain assets when entering Uwaterloo. The number of steps forward is determined by rolling dices. When landing on a ownable building, players can choose to purchase the building (if the building belongs to no one else at that time) to make other players pay a certain amount of fee when passing by. When entering a unownable building, certain activities will take place (possibly increasing or decreasing its assets to a certain extent). When a player owes more money than they have at the moment, they can drop out the game. The winner is the last person who has not dropping out. All extra credit features will be discussed later in this report.

Components & Rules of the Game

Board

A 40-square board represent the state of the current game visually. The board is made up by 2 types of squares. The first one is ownable properties, the other is unownable buildings (will be discussed later in this section). The improvements for the properties as well as the states of players will be shown on the board as well.

Mode of operation

The human players move by manipulating two six-sided dice. The number of moves is the sum of two dices.

Buildings

As we all know, different buildings in Uwaterloo have different purposes.

Part1: Ownable Buildings

a) **Academic Buildings**: When the player lands on an unowned academic building, they can choose to purchase the building depends on its corresponding value. When the building is purchased, other player will need to pay tuition to the owner for attending lectures in the building. In addition, each ownable building is included as a smaller part of a collection of properties, known as monopoly. When monopoly is reached, the owner can choose to upgrade the building by purchase improvements as the tuition will also be increased as well (see the chart below). If the owner chooses not to upgrade the building when the monopoly is reached, the tuition will be costed twice as much as the initial setting.

b) **Residences**: The residences in Uwaterloo (MKV, UWP, V1, and REV) costs $200 each. Instead of tuition, rent will be charged to the visitors. Base on the number of residences the player owns, rent is calculated differently. The rent when one residence is owned is $25, two residences $50, three residences $100, and four residences $200.

c) **Gym:** Two gyms in Uwaterloo (PAC & CIF) cost $150 each. Usage fees will be charged when visitor lands on one of the gyms. Usage fees are calculated by rolling the dice twice. Usage fee is the sum of the dice times four if the player owns one gym. Usage fee is increase to eight times when the player owns two gyms.

Part 2: Unownable Buildings

a) **Collect OSAP**: Each time a player passes over or lands on the Collect OSAP, they gain $200 unless told otherwise.
b) **DC Tims Line**:
   i. If the player lands on DC Tims Line, nothing happens.
   ii. If the player is sent to DC Tims Line, they are not allowed to leave the square unless they roll doubles, pay $50 or use a Roll Up the Rim cup (will be discussed later in this section).
   iii. If the player is in the DC Tims line for the third turn, the player must leave if the plyer still doesn't roll double by paying or using the Roll Up the Rim cup. The player moves the sum of the dice from their last roll under this situation.
c) **Go to Tims**: When the player lands on Go to Tims, instead of collecting $200, they will be sent directly to DC Tims Line.
d) **Goose Nesting**: If the player lands on Goose Nesting square, they are attacked by a flock of nesting geese, nothing happens otherwise.
e) **Tuition**: Ehen the player lands on the tuition square, the player must decide whether to give the school $300 or 10% of their entire value.
f) **Coop Fee**: Pay $150 to the school.
g) **SLC**: If the player lands on SLC, their piece is moved on the board randomly. A message should be printed stating by how much the player moves and to which square they are moved. Play continues as though they had landed on the square they were move to. To movements are listed as below.
h) **Needles Hall**: On Needles Hall square, a player's savings are either increased or decreased depending on where they fall. The distribution below shows how much a player's savings are impacted.
i) Roll Up the Rim Cup
   A 1 percent possibility exists while landing on Needles Hall or SLC that the player will receive a Roll Up the Rim cup instead of the usual result of visiting that building. By using this cup, you can leave the DC Tims line without paying. There should never be more than four active cups at once.

Actions

1. **Improvements**
   When a player has a monopoly, they can construct up to five improvements on academic buildings (four restrooms, then a cafeteria). Chart 1 estimates the cost of each improvement for an academic facility. A player who chooses to sell an improvement will get paid half of its price. Each improvement is denoted by an I in the text display's top row of the corresponding property.
2. **Mortgages**
   A player can always take out a mortgage on a property they own. When a property is mortgaged, the owner receives half of the purchase price, and the occupants are not required to pay rent to the owner. Note that before a building

can be mortgaged, any improvements must be sold (as stated above). A player must pay half the cost of a property plus an additional 10% of the cost of the property to unmortgage it (a total of 60 percent of the cost of the property).

3. **Bankruptcy (Dropping out)**
   Only when a player owes more money than they have at the moment can they leave the game (declare bankruptcy). The gamer who owes money now has two choices: either file for bankruptcy or attempt to raise the money they owe. The player who is owed money inherits all of the bankrupt player's assets if they file for bankruptcy because they owe that person money. Otherwise (if the player owes money to the Bank), all Roll Up the Rim cups are destroyed, and the buildings are put back on the open market as unmortgaged assets (see Auctions below).
4. **Auctions**
   Properties are sold at auction if a player opts not to purchase them or files for bankruptcy with the bank. Each participant in an auction has the choice to either increase their existing bid or leave the event. When just one participant remains in the auction, that person is the winner and is required to make their winning bid.
5. **Game ending**
   When there is only one player remaining who has not quit, the game is over (declared bankruptcy). That player wins the game!

Overview

Part1 (Arguments)

For the command line options, our game is able to process 3 command line arguments.

1. The first command line argument is **-load file,** used for testing purpose for a saved file. The argument follows a fixed pattern as below:
   a) The first line is an integer specifies the number of players.
   b) The followed line specifies each player (per line), their name, the number of Tim Cups (discussed in the rules), the amount of money they have(starting at 0), and the square they are on. Two specific situations when the player is in DC Tims line are discussed below:
   c) If the player's position is DC Tims line but they are not in there, an additional 0 is added at the end of the line.
   d) If the player is actually in DC Tims line, an integer 1 is added following by another integer at the end of the line indicating the number of turns the player has been there (0 -2 inclusive as discussed in the rules).
   e) The following lines should be each building in the order in which they appear on the board. Each line should contain the building's name, its owner (the owner is BANK if it's not owned yet), an integer represents its improvements (-1 or 0-5 inclusive as discussed in the rules).

Example:

2
Goose G 0 1120 25
GRT_Bus B 0 1240 20

ECH GRT_Bus 0
CPH GRT_Bus 0
LHI Goose 0
V1 Goose 0

2. The second command line argument is **-testing**. Roll is the only testing command that is required. This dice roll will be referred to as *roll <die1> <die2>*. The player will shift the sum of the player's two dice, each of which can be any nonnegative number and not needs to be between 1 and 6.
   Example:
   roll <0> <21>
3. The additional argument: The third command line argument is **-seed**. After this argument we should also input a int number for the standard seed. The seed will replace the original random seed, so every time we row the dice will generate a certain result for us to test the program.
   Example:
   If we input 15 as the seed, then my computer will output the following result:
   1 3
   2 1
   1 3
   3 1
   ...

Part2 (Input Command)

1. The user will first be required to enter the total number of players, their names, and the character that will represent each player on the board. Each participant begins with $1500.
2. Commands:
   a) **roll** (used if the player is able to roll): The player rolls two dice, sum them up, and then performs an action on the square where their die landed.
   b) **Next** (used if the player cannot roll): Transfer control to the following player (used if the player is unable to roll).
   c) **Trade <name> <give> <receive>**: The present player offers a trade to name, providing give and requesting receive, where give and receive are either dollar amounts or the names of properties. There are two types of responses: accept and reject.
      i. *trade Brad 500 DC*: Brad is willing to accept $500 from the present player in return for the DC building.
      ii. *trade Rob DC MC*: The current player is willing to trade Rob DC in return for MC.
      Note:
      1. our program rejects all attempt for a player trade money to exchange money.
      2. Only when none of the properties in the monopoly have improvements can a player make a trade offer.
   d) **improve <property> buy/sell**: attempts to buy or sell an improvement for property.
   e) **mortgage <property>**: attempts to mortgage property.

f) **unmortgage <property>**: attempts to unmortgage property.
g) **bankrupt**: player drops out by declaring bankruptcy.
Note:
This command is only available when a player must pay more money than they currently have.
h) **assets**: reveals the player's resources at the moment. Fails if the player is now deciding how to pay for tuition.
i) **all**: displays each player's assets. To confirm the accuracy of your transactions. Fails if a player chooses how to pay for tuition.
j) **save <filename>**: saves the game's current state, as described below, to the specified file.

Part 3 (Classes)

The whole project is divided into three parts, human, shell, and kernel.

Human players generate random instructions during playing and pass it to the controller.

The **Shell** of the program is the Controller, responsible for receiving all information from humans by passing the arguments and input command and interact with the kernel. As for the **Kernel**, it will first generate the initial map and receive instructions and data through the controller by pointers. A series of storage, calculations, and output instructions are carried out inside the kernel.

Two concrete classes are generated from the controller, the **Player** and the **Grid**. Player is responsible for dealing with all the instruction related to the players, while the Grid class is responsible for generating the initial grid. The Grid has a composition relationship with the **Property** class. The Property class is responsible for handling all instructions related to the properties in the game. As our rules explained in the beginning, buildings are divided into two broad categories, Ownable and Unownable, so two derived classes for the Property are designed known as the **Ownable** and **Unownable**. Each class is responsible for handling all the instructions related to its category. All the ownable buildings and unownable buildings we discussed in the rules (chart 1) are designed as the derived class of their category they belong to.

Design

An observer design pattern is implemented since there is a one-to-many relationship between the building and operations such as monopoly, improvement, paying the tuition and so on. To achieve that, we implement a Subject class, along with an Observer. The derived class for both of them is the Building class. The categories of the buildings (whether ownable) and the decision whether or not the player wants to purchase this property should be done in this class. Two derived class of the Building are also added as the Unownable Building and Ownable Building to classify all buildings on campus. Ownable Building sets functionalities including purchasing, improving, setting its owner, getting its type, etc. to achieve all the possible actions that a player can take on a building. The observer for the ownable buildings is the ownable buildings within the same category as well as the player

since we should notify the building to when intended to take actions like monopoly or improvement and notify the player when intended to take actions such as receiving the tuition and so on.

In addition, a decorator design pattern is implemented to achieve those events. As we mentioned above, we want to use the decorator pattern for some special events, like Roll Up the Rim Cup. In this way, the program dynamically chooses the suite event when they are currently in a special location, like SLC and Needles Hall. This is also for the future optimization and better readability to another programmer who read our program.

Resilience to Change

First of all, we carefully modulize the whole program into parts, as discussed above in the classes component. When the change is made such as adding a new method, we only need to add content to the corresponding module, other modules have no need to change, thus minimizing the changing to the original program. Second, we strictly use the inheritance and polymorphism in our program to make every property a concrete class. Last but not the least, we made a lot of method in our program private or protected, we the similar method or functionality need to be added or modified, we can use the similar method that implemented before to reduce the work.

Answers to Questions

Question 1

The observer pattern is an ideal pattern for this game. We first set the Player class, Building class as the child class of the abstract Observer class, and then we set the Building class as the child class of the abstract Subject class. For those ownable properties, they may notify the player(observer) after they owned the buildings, and some players stay on them. Also, we want to notify more than just the owner, but also all ownable building with the same type. For example, we hope that if one ownable building is owned by a player, we hope that this ownable will notify other ownable building with the same type, so they may be able to know whether they can be improved. In addition, as a subject, each ownable building may also know whether they can be sold to other players. For instance, if a player owns a collection of monopoly, and some other ownable building were improved. In this case, this player is not expected to sell this building to another person. In this way, we hope every time, the object of OwnableBuilding will notify its observers when it changes its status, or some events happen. And we also want to make the UnownableBuilding class as the child class of the Observer class and the Subject class. In this way, we hope that every time when player stay at the location like Roll Up the Rim Cup, the player can know the status and number of other active cup. Since we know that the total number of cups owned by all players cannot exceed 4, which means that if 4 cups are active there is a 0% chance of receiving a Roll Up the Rim cup. In this case,

each player can see the status of each other in real time and help make the right choice for each event.

Question 2

We believe the decorator pattern will help SLC and Needles Hall more realistic. Since there will be two models for both places, the actual model, like SLC and the Needles Hall, and the additional models, like Roll Up the Rim Cup. According to the rule of Roll Up the Rim Cup, we can see that when landing on either Needles Hall or SLC, there is a rare chance (1%) that instead of the normal effect of visiting that building the player receives a winning Roll Up the Rim cup. This cup is used to get out of the DC Tims line for free. At any point in time there should be no more than 4 cups active (i.e., the total number of cups owned by all players cannot exceed 4, which means that if 4 cups are active there is a 0% chance of receiving a Roll Up the Rim cup). Since we have already had the real-time information to satisfy each event, we want to model the decorator pattern just as we did in Assignment 4 Question 2. The program can always go back and update the status if it encounters a suitable rule. In addition, if the program meets the incorrect condition of the event, the program will check the next event and return when it finds the suitable event to process. Another reason we want to build decorator pattern for this part is that we also hope that this design can add more events in future optimizations (after the final project). For example, If we want to add a new event to the game, we just need to implement it like the event like Roll Up the Rim Cup. By taking this modular approach, we can save a lot of time and increase the readability of the program.

Question 3

We believe the decorator pattern is not a good pattern to use for the implementation of improvements, since there are too many different situations for the improvement. If we recall the columns of tuition with improvements in Table 1: Academic Purchase/Tuition/Monopoly Block information, we will have about 132 integers related to the tuition of improvement. It is totally possible to build the decorator pattern for this part. However, if we check the data carefully, we will find out that some of the ownable building in the same monopoly does not have the same amount of tuition with the same level of improvements. In this case, the difficulty of building these data with decorator pattern is harder than we simply list each tuition with different level of improvements when we try to implement each of our ownable buildings. All in all, it is not worthy to build a whole design pattern for a simple feature.

Extra Credit Features

First of all, our program implements the fixed seed feature (discussed in Overview, Part1, 3), the load and save feature is implemented as well. In addition, we added a “quit” command in order to allow players to quit in the middle of the game. An additional player can also be added except the original ones. Moreover, our

program is designed with no memory leak completely. Last but not the least, all pointers that store data are shared pointers.

Final Questions

1. We benefited a lot from the group work. To begin with, this project let us understand the importance of cohesion between teammates. Due to the time restriction, the entire project must be assigned to different team members and been developed individually in order to peak efficiency, the problem may occur as the code works well separately but crushed when put things together. To overcome this problem, we spend more time communicate with each other to link up our code perfect things together. Second, this project taught us to be punctual and rigorous as a team member. Since we all need to finish our part within the specified time, we are able to meet with our team members at the set time and proceed smoothly to the next step. One's own unpunctuality will slow down the whole team's efficiency.

   Moreover, a small mistake made by an individual caused by carelessness will greatly reduce the efficiency of the whole team. In a relatively large project, subtle errors are often hidden so deep that they sometimes need a long time to find. This small error can cause the entire project to crush and waste a lot of time looking for. Therefore, it is particularly important to be rigorous and meticulous in your own work and set aside time for inspections when working with a team. To sum up, this project provides us with a precious opportunity to experience programmer's teamwork, and gratefully benefits our study and future career.

2. First of all, we should design more carefully before we start writing code. In the middle of writing the code, we realized that the version we had designed at the beginning had problems like code duplication, over complicated implementation, and link errors, which resulted in the need to redesign the code and greatly increased the amount of useless work. Fortunately, we planned enough time to refine the project. Second, we should communicate thoroughly throughout the whole process. In the beginning, we simply allocated parts and started writing code separately, when putting things together, unexpected problems keep cropping up which took a lot of time to fixed.

Conclusion

First, we are very grateful for the rare opportunity to be a team and complete the entire game code design, we really learned a lot. We gained a lot of skills and experience of working with a team such as planning ahead, allocating tasks wisely, and communicate thoroughly so that we can collaborate in team better in the future, whether in work or study. Secondly, our team made concerted efforts throughout the whole process and helped each other in the face of various difficulties. What seems like a tedious and difficult task at first builds up day by day, finally completed by our effort. We made a great team and are truly proud of each other.