Kok Yen Zein 20977899
Thomas Zhou 20958711

CS246 Watopoly Final Design Document

Project Specified: Watopoly

Introduction

What is Watopoly? Watopoly is a game similar to Monopoly based on the University of Waterloo campus. A game of Watopoly consists of a board that has 40 squares, where each square activates an action. Players take turns moving around the board buying, improving properties, paying rent, and various other actions until only one player is left. The ultimate goal is to make all other players drop out of university (declare bankruptcy) while remaining the only player on the board!

Overview

As indicated in the UML design, we opted for 8 classes, which are all interdependent to operate the game. Here is a description of each class and its purpose.

Observer:
The *Observer* class is a superclass containing two public methods, which purpose is to receive notifications from the *subject* when its state changes; in this case, when a *BoardPiece* object state changes. For instance, a BoardPiece is an ownable property called UWP, which is a residence. When the UWP property is bought, thus owned, the *observer* class is called to take note of the changes.

TextDisplay:
The *TextDisplay* class is a subclass inheriting from the *Observer* class. The purpose of this class is to maintain, update and output the Watopoly game board display containing all *BoardPieces*, ownership, mortgages, improvements, and player location. For this reason, the *TextDisplay* class is inherited from the *Observer* class, it is able to receive notifications when a subject's state changes, thus making those changes in the display. Continuing from the example in the *Observer* description, when the UWP property is bought by, for instance, player Thomas with a player character G (Goose), the *TextDisplay* class will change the display of the UWP *BoardPiece* to indicate that the property is owned and who contains the ownership, and output the updated game board.

Subject:
The *Subject* class is a superclass containing three public methods, which purpose is to manage the list of observers and notify the *observers* of any state changes. The *subject* object provides an interface for attaching *observers*, thus allowing *observers* to receive the notifications.

BoardPiece:
The *BoardPiece* class is both a superclass and a subclass inheriting from the *Subject* class. The purpose of this class is to maintain and store information about each property (ownable or unownable properties) such as the property's name, its position on the game board, its purchase cost, and many more. Furthermore, the reason the *BoardPiece* class is inherited from the *Subject* class, each *BoardPiece* can attach to an observer to notify the *TextDisplay* class of any changes in each *BoardPiece* object. For instance, the UWP property is sold.

OwnableProperty:
The *OwnableProperty* class is a subclass inheriting from the *BoardPiece* class. The purpose of this class is to maintain and store information about a *BoardPiece* object that is ownable; this includes Academic Buildings, Residences, and Gyms. For that reason, there are different actions/methods that are included in the *OwnableProperty* class; for instance, mortgaging a property which is not allowed from an unownable property.

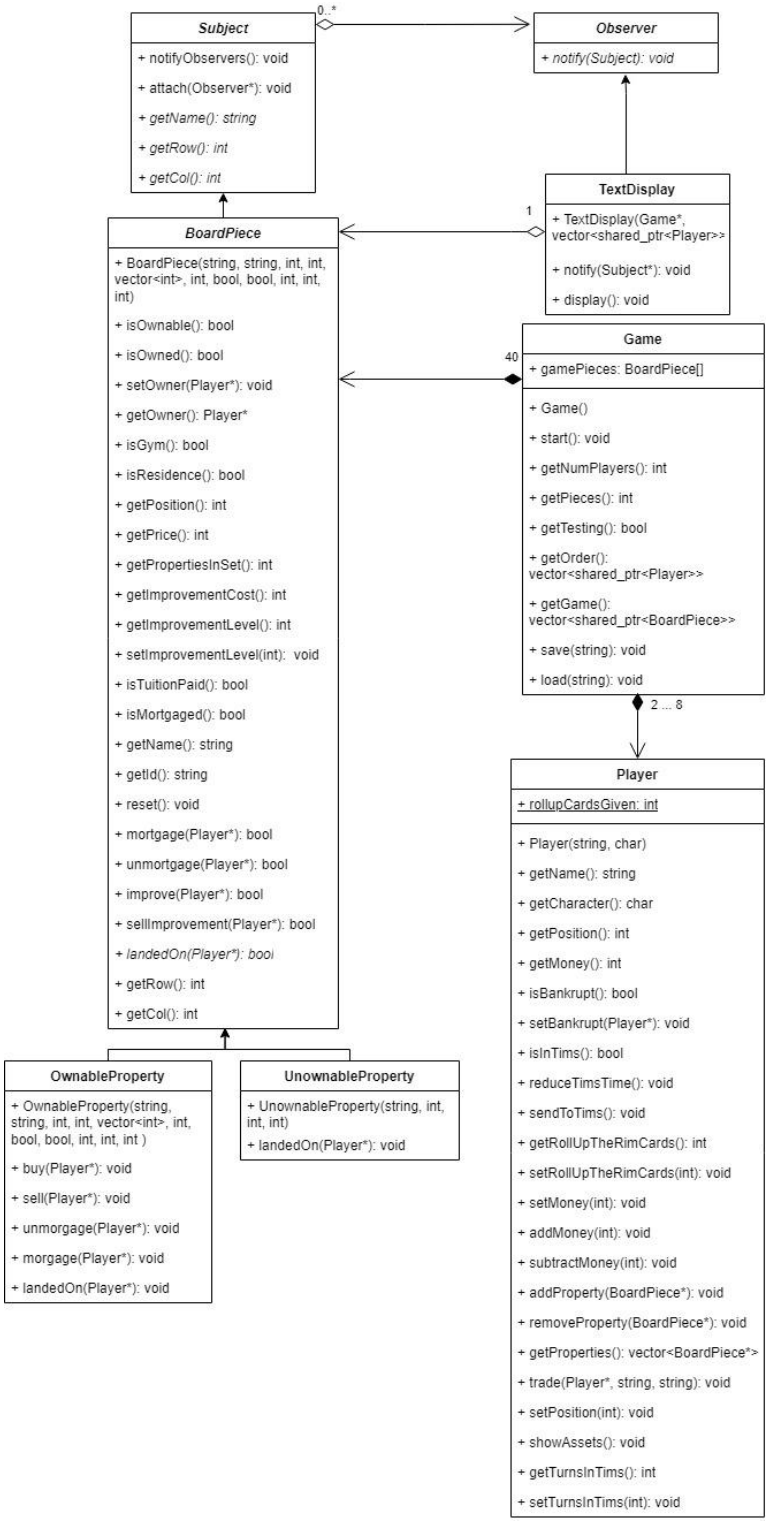
UnownableProperty:
The UnownableProperty class is a subclass inheriting from the BoardPiece class. The purpose of this class is to maintain and store information about a BoardPiece object that is unownable; this includes Collect OSAP, DC Tims Line, Go to Tims, Goose Nesting, Tuition, Coop Fee, SLC, and Needles Hall. For that reason, there are different actions when a player lands on an unownable property; for instance, when a player lands on Collect OSAP, they receive $200.

Players:
The *Player* class maintains and stores information about a player. The *Game* class interacts with the *Player* class, containing several methods which indicate the player's character, position on the board, amount of money, if the player is bankrupt, the properties the player owns, and several more.

Game:
The *Game* class is the interface of the game, which contains all the *BoardPieces* objects and *Players* objects. The purpose of the game class is to run and maintain the Watopoly game until there is only one player left. It is the class mainly used to interact with the user, allowing the user to call methods test the program, load previously saved games, and perform several actions on a BoardPiece such as rolling, trading, buying an improvement, selling an improvement, mortgaging, unmortgaging, listing out *Player* assets, listing out all assets, and saving.

Kok Yen Zein 20977899
Thomas Zhou 20958711

Updated UML



CS 246 Kok Yen Zein 20977899
Watopoly Thomas Zhou 20958711

Design

For our implementation, we used the observer design pattern, to establish a one-to-many dependency between an OwnableProperty or UnownableProperty object and the TextDisplay object, such that when the information of any of the OwnableProperty or UnownableProperty object changes, the TextDisplay is notified, resulting in the output display being changed and printed.

The observer design pattern can be seen implemented in the specific Classes: Subject, BoardPiece, OwnableProperty, UnownableProperty, Observer, and TextDisplay. However, most of the use of the Observer Design Pattern to call attach(Observer), notifyObservers() and notify(Subject) is used in game.cc. For instance, when starting or a loading a Game, all BoardPieces are attached with the TextDisplay Observer, and are then notified with any changes. This allows the TextDisplay to format the displayable game board such that it conforms to the data for each BoardPiece object.

Furthermore, other instances where notifyObservers() is called are:

- Trading, where the ownership of a BoardPiece changes
    - trade <name> <property> <property>
    - trade <name> <property> <money>
    - trade <name> <money> <property>
- Improving, where the Improvement level of a BoardPiece changes
    - improve <property> buy
    - improve <property> sell
- Mortaging and Unmortgaging, where the mortgaged status of a BoardPiece changes
    - mortgage
    - unmortgage
- Auctioning, where the ownership and owned status of a BoardPiece changes
    - Where an auction can occur when a Player does not want to buy the property at its purchase cost or when a Player declares bankruptcy to the BANK, thus allowing their properties to be auctioned.
- Bankruptcy, where if the Player declares bankruptcy to another Player, ownership of each BoardPiece of the bankrupt player is then given to the owed Player.
- Movement, when a player rolls resulting in movement, or lands on SLC or Needles Hall resulting in movement, the position of a Player changes.

Resilience to change

In terms of the implementation of the classes, with the way we have separated the responsibilities of each class, changes to one class would not drastically affect other classes. For example, the Player class includes all the information about a player, a BoardPiece contains all information relating to a specific BoardPiece, and the Game contains all the logic to

manipulate the Players and BoardPieces. Each class's information is contained in private variables which encapsulate and prevent access by other classes. This ensures that the only way to interact with the objects is through the specific public methods that are defined. Additionally, since the BoardPiece class is abstract, we can include other types of subclasses other than OwnableProperty and UnownableProperty that inherit from BoardPiece and implement them in the program. Throughout the program, we employ the goal of low coupling and high cohesion. Each class can only interact with other classes through specific publicly defined methods, and the classes work together to form the completed program.

Besides that, as indicated above, our implementation is based on the observer pattern, which promotes loose coupling between objects, where the subject and observers are independent and do not need to know anything about each other's implementation details. This allows for flexible and extensible designs where new observers can be added without affecting the behaviour of the subject or other observers.

Answers to Questions

Question: After reading this subsection (Buildings), would the Observer Pattern be a good pattern to use when implementing a board? Why or why not?

After implementing Watopoly, we stick by our answer from Due Date 1 that the Observer Pattern is the most suitable when implementing the Watopoly game.

The difference to Due Date 1 would be that, we can think of it as a Publish-Subscribe Model where the each BoardPiece (instead of the general Board) acts as a subject, and the TextDisplay (instead of players' location, tuition/rent cost, and more) acts as observers.

As explained in Due Date 1, whenever the state of the board changes, for example, two die are rolled and the player moves to a new location, the changed BoardPiece object (subject) will notify its observer, a TextDisplay object to be updated, ensuring that all relevant parts of the board are always up-to-date and in sync with each other.

Question: 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?

We did not pursue the extra credit feature of SLC and Needles Hall similar to Community Chest and Chance cards; however, we believe that the most suitable design pattern is still the Template Method Pattern - agreeing with Due Date 1.

This is because the sequence of events that occur when a chance or community chest card is drawn is similar, but their effects are different.

In practice, an abstract class called “randomCard” can be defined with a method called “execute.” Inside “execute,” there could be functions such as “pay,” “recieveMoney,” and “goToJail” as examples. Then, each subclass can modify these methods to their intended effects to represent the different chance/community chest cards. This ensures that all necessary functions of a random card are called in a fixed order, but its functionality can be altered. In the game, each change/community chest card is initialized and stored in a stack of class “randomCard.” Whenever a player lands on a change/community chest square, the “execute” method of that card is called and its effect is used

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

After implementing Watopoly, we stick by our answer from Due Date 1 that the Decorator Pattern is the not suitable when implementing Improvements.

The decorator pattern would not be a good pattern to use when implementing improvements. This is because the primary benefit to using the decorator pattern is to add additional functionality to a program at runtime. Since the only effect of adding or selling improvements is changing the price of tuition and not the functionality of the building, it is not necessary to use the pattern. Instead, adding or selling improvements should only change the amount of tuition owed when a player lands on the building.

Extra Credit Features

We used shared_ptr for all instances where we would have needed to use new, and vectors instead of arrays. We have no delete statements in the code, and there is no need to manage memory.

Final Questions

Question: What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Thomas: This project has taught me the importance of organizing and planning in a large collective project like this. Having a clear vision of the program structure and its implementation was vital to the success of the project. We created design documents outlining the functionality of classes and the methods they included which made it easy to implement. We also had the code shared over Github which made collaboration and code syncing easy. One thing I’ve realized when working as a team is that comments in code are very important and that frequent commenting can clear confusion and make code much more readable, which is something I can improve on.

YZ: Throughout this project, I have gained significant skills not only in Object Oriented Programming in C++, but also soft skills. For instance, the importance of communication is a key factor to the success of the project, and I believe Thomas and I had maintained a high level of transparency and communication. Furthermore, we regularly notified each other of any changes in the code, to avoid redundancies and misunderstandings. Furthermore, I have gained experience in sharing code through GitHub Desktop, Msys, and WSL, which allowed for simple collaboration and maintenance.

Question: What would you have done differently if you had the chance to start over?

Thomas: I would have planned out each method and the control flow of each piece of the game logic in order to avoid confusion during implementation. I would have also tried to implement more flexibility in the code to implement more enhancements.

YZ: After Implementing the code for Watopoly, I felt like I could have allocated more time to the project so I could have tried extra credit features such as Implementing Computer Players, or Different board themes. Furthermore, if I could have allocated more time, I would try implementing the code with smart pointers, so I could gain additional experience in them.