# Final Design Document - Watopoly

## Introduction

*Watopoly* is a special version of the game *Monopoly* implemented by students from the University of Waterloo using C++. In this game, the players buy and trade the properties at Waterloo, and compete with each other to become the richest player. In this design document, we will outline the structure and design techniques *Watopoly* uses, and discuss how specific programming challenges are overcome.

## Structure Overview

The structure of *Watopoly* is categorized into three major classes: Board, Player, and Square, and the MonopolyBlock Class that groups squares in an monopoly together. Aggregation relationships link the three major classes together, where Board contains Player, Square, and Property(a specialization of Square). Square Class is an abstract class and is a generalization of two other abstract classes, Property Class and NonProperty Class. Property Class has aggregation relationships with the Player and the MonopolyBlock. Specializations of the Property Class are Upgradable(also known as the academic buildings), Residence, and Gym. OSAP, GoToTims, TimsLine, GooseNesting, Tuition, CoopFee, SLC, and NH(stands for needle hall) are inherited by the abstract class NonProperty.

## Design

### Design Pattern - Observer

We use an Observer Design Pattern to change the states of the players and the states of the squares. The Board Class is a concrete subject, and the Player, Square, and Property Classes are the observers attached to the board. When 'b.move', 'b.trade', or 'b.startAuction' is called, the Board notifies the players, the squares, and properties to change state accordingly. The code we mentions below can be found in board.cc.

#### 'b.move'

'b.move' moves the player to the position indicated by the dice roll. It changes the position state of the current player. Then, the square in which the current player is stepped on calls the virtual method 'playerEffect' on the current player, and changes the state of the player according to the functionality of the square. For example, If the player steps on an academic building that does not have an owner, players can choose to either purchase the building or not purchase it and let other players participate in an auction. When the player steps on an academic building owned by others, the player pays to others. If the player steps on a nonproperty, rewards or penalties will be given accordingly.

**'b.trade'**

'b.trade' modifies the ownership of squares, and can change the amount of money players have. A player can choose to either a) trade using money, which modifies the amount of money of both trades, or b) trade with property, which exchanges the ownership of the properties between the traders.

**'b.startAuction'**

'b.startAuction' changes the state of the winner of the auction by withdrawing the bid from the winner, and updates the ownership of the auctioned property to the winner.

**Single Dispatch**

**improving and mortgaging**

We use the single dispatch to implement improvement buying, improvement selling, mortgaging, and unmortgaging. The program specification requires that only academic buildings can be improved, and to mortgage an academic building, the player needs to sell all the improvements. This means that we need to determine the specification of the properties we are going to improve or mortgage at run time. Therefore, we overcome this problem by overriding the improvement buying and selling, mortgaging, and unmortgaging functions in the Property Class on players. In the Player Class, we implement these functions by calling the overriding functions on the player. By using a single dispatch, we let the improvement buying, improvement selling, mortgaging, and unmortgaging functions determine the target property at runtime.

**Smart Pointers**

Shared pointers are used for all the pointers we define so that we do not need to call delete to free the memories. The MonopolyBlock Class has a reference to a vector of properties, and the Property Class has a reference to a monopoly block. To avoid potential cyclic references, we define the vector of properties inside the MonopolyBlcok to be regular pointers. This makes sure that all the memories are freed.

**Cast**

We use dynamic casts to try to convert Square objects to either Properties or NonProperties, and Properties to their specifications. This helps the implementation of the display of the board because the display of the square is dependent on the type of the square in terms of whether it is an academic building or a non-property, etc. By using dynamic casting, we can check whether the given square can be cast into the type we assign to, and thus decide which printing function we can apply.

**Vector**

We use std::vector to store the collections of players in the game, the collection of squares on the board, and the collection of academic buildings that belongs to the same monopoly. This is a

helpful technique to use because we can iterate through the vector to locate the element we want, and access the element by index.

#### Monopoly

To keep track of which monopoly the academic building belongs to, we create the MonopolyBlock Class, which contains a vector of pointers to properties that are in the same monopoly. In the Property Class, there is a field named block to keep track of which monopoly the property belongs to.

### Resilience To Change

#### Cohesion

As we mentioned above, the structure of *Watopoly* is categorized into four classes, and each individual class is doing its own task. For example, the Player Class stores the money, net worth, current position and the number of Tim's Cups for the player, the Square Class manages ownership, improvements, mortgage, and effects when player steps on a square and the Board Class combines all the players and the squares to form the state of the game. Therefore, cohesion is high in our implementation of *Watopoly*.

#### Coupling

Coupling is low in our implementation of *Watopoly*. We declare the fields in all classes as private fields and each class has a low dependency with each other, therefore modules do not have access to each other's implementations. We implement the display of the board, and loading and saving outside of the Board Class. Hence the Board Class is reusable by other programs and modifiable to achieve other functionalities without rewriting the whole class.

#### Summary

Since each individual class defined in our implementation serves its own purpose and functionality, and the dependency between each module is low, it is possible to make functional changes to specific modules in order to serve different program specifications.

### Answers to Questions

#### Question 1

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

**Answer**

Observer Pattern would be a good pattern to use for the gameboard. On Due Date 1, we thought that Observer Pattern can be used between Board and Display. In the actually implementation, we use it between Board and Player, and between Board and Square. We let the Board be the subject and let the Players and the Squares be the observers. To change the state of the board, we notify the players and squares to change their states accordingly. By implementing this way, we separate the tasks of the board, improves coupling, and satisfies the single responsibility principle. In the actual implementation, we use the observer design pattern as we mention above.

**Question 2**

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**

We think that the Factory Method is a suitable design pattern to implement such feature, which is the same design pattern as we proposed on Due Date 1. We would build a Chance/Community Chest Class and a Card Class, where the Chance/Community Chest Class is the creator and the Card class is the product. When a player steps on a Chance or a Community Chest block, the class method creates a random Card and applies the effect of the Card to this player.

**Question**

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

**Answer**

Decorator Pattern is a good pattern to implement the improvements because the tuition fee for buildings is different and depends on the current number of improvements. However, in the actual implementation, we do not use the decorator pattern for the improvements. Instead, we define an array of length 6 that stores the tuition fees with a different number of improvements and define an improvement field that stores the number of improvements. The advantage is that we can directly get the tuition fee by accessing the array of tuitions by index, where the index is the number of improvements. Implementing in this way also makes checking whether the number of improvements reaches the maximum easier.

## Extra Credit Features

### Display of the Board

We implement the ASCII display of the board which can display a maximum of 8 players. This is challenging because although the squares are in orders, we need to print the board line by line.

Moreover, the display of squares is different. For example, for the academic buildings, we need to print the number of improvements, while for other buildings we do not. We solve this problem by overloading the helper functions in displayBoard.

#### Display of the Strip

From a user perspective, we would like to implement a display that can clearly show the players' locations, and since the display of the board is large, it is very hard for players to keep track of where their current position is. Moreover, the display of the board might clear important information for the players since it is large. To solve these problems, we implement the function displayStrip, which displays 8 squares around the current player, with the current player being at the center(5th square). This helps clearly indicate the current player's position without occupying a large space of the screen. The implementation is challenging because we define the collection of squares in sequence as an array. If the player is near the origin(OSAP), we need to display squares that are at the end of the array.

#### Help Command

If players forget the command names or command functionalities, the "help" command is there to help. We implement the "help" command to list all the commands as well as their functionalities.

### Final Questions

#### Question 1

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?

#### Answer

This project teaches us how to use Git and Github in software developments with team members. One advantage of working in a team is that we can distribute the tasks. To work independently with each other, we study to use Git and Github, which are the most popular version control software in software development. With the help of Git and Github, we can separate the tasks, work on different functionalities individually, and merge all works together to form the final program. Moreover, this project also teaches us the importance of communication in software development. During the development, there was a time where we did not have enough communication with each other, leading to two team members implementing the same functions, which causes a double definition error during compilation. After this mistake, we tried to make each commit as clear as possible, make frequent group calls, and kept each other updated on the changes by notifying everyone in the group chat. Finally, planning is essential in development. Since before coding, we planned the structures of the implementation well and proposed practical ideas on the design patterns, we saved a lot of time coding.

**Question 2**

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

**Answer**

If we have the chance to start over, we would want to implement the display of the board differently. Since the display of the squares depends on the actual specification of the squares, we made many overloaded functions and used a lot of castings, which is messy. If we can start it over, we would probably want to override these overloaded functions to avoid dynamic castings as much as possible. Moreover, we could add a Dice Class, and control the dice using a concrete class object, instead of using 'rand()' functions and integer representations. Finally, if we have enough time, we would want to implement a graphic display for the board to improve the game quality.