# Design Document: Watopoly

Group members: Felix Zong(f2zong), Qing Wang(q473wang), Tianhao Chen(t345chen)

## Introduction

This document consists of an overview of all aspects of our chosen project—Watopoly, including how they were implemented as well as the required answers to questions.

## Overview

Our program of Watopoly made extensive use of object-oriented program design techniques, including inheritance, observer design pattern, and abstract classes. We save data in different base classes and implement basic behaviour in base classes, and a complete game logic at gameplay class. We have implemented and tested all core mechanics of a monopoly game, as well as two different themes of the game board.

## Design

### Main

Our main.cc script is a basic argument interpreter. It reads the game arguments and produces a modified gameplay pointer with these arguments. Then it calls an iterative method play() on gameplay. At the end of the game, main will delete the pointer.

Available game arguments:
-theme square.in/snake.in
-load <loadfile>
-testing

### Gameplay

Gameplay class is an overall controller of the game, storing current game status, a vector of all player pointers and a game board pointer. In gameplay, we implement all command logic, and call modifiers of the Building class and Player class for each command. Gameplay has a public method to be called by main.cc, play(), in which it iteratively calls parseAction() that reads commands from standard input and then calls protected methods of the corresponding command.

Available commands:
roll <die1> <die2> // only available when test mode is on;
roll
mortgage <property>
unmortgage <property>
improve <property> sell/buy

assets
all
bankrupt
help
save <filename>

## Player

Player class saves information of a player, and has public methods that modify those fields. Player is a subject to the board. When the position of a player changes, it will notify its observer(board) to update its position on the displayed game board.

## Board

Board stores a vector of pointers of the buildings in a sequence of those displayed at output, a map of players' name and position on board, and a vector of string which is the display text of the board. Board is an observer of both players and buildings. When it's notified by a player, it will get PlayerInfo of the player and update the player's position at text display, and at the map of players' name and position. This map is necessary since we need to keep track of the previous position of the player, and clear it from the board when it has a new position. When the board is notified by a building, it will get BuildingInfo of it, and update the owner and the number of improvements on the text display. It also has an override operator<< that makes output simple.

## Building

Building is an abstract class, containing two fields: building name and building type and it inherits from Subject. Building contains virtual methods that are general to all kinds of buildings: movement() which get the steps players need to move when landed on, and tuition() of the building.

### Property

Property class is another abstract class inherited from building, it contains public methods that are general to all property buildings, for example, getOwner() and so on.

#### AcademicBuilding

AcademicBuilding is an observer and a subject at the same time. Besides basic accessor and modifier, it can identify whether all the academic buildings in its monopoly block are all owned by one player and can tell which player is the monopolist. This feature is implemented by an observer pattern. A building is attached to all other buildings in its monopoly block. When the owner of an academic building changes, it will notify all other buildings, then the other buildings will get BuildingInfo of this building and save it in a vector of BuildingInfo, then other buildings will check if the owner of all the buildings from this monopoly block is the same player. If so, it will update the monopolist field by the pointer of that player. AcademicBuilding has a method that helps implement the "trade" command: when trading a building from a monopoly block that the player currently monopolies, getMonopolyImprovement() will return the sum of improvements from the buildings of this monopoly block. It has to be 0 for the trade to be legal, since if the owner of one building changes, improvements from other buildings will be illegal.

#### GymsBuilding

GymsBuilding is also an observer and a subject at the same time. Its logic is similar to academic building. It needs to know how many gyms the owner owns in order to calculate the correct tuition.

#### ResidenceBuilding

ResidenceBuilding is almost the same as GymsBuilding, it also needs to know how many gyms the owner owns in order to calculate the correct tuition.

### NonpropertyBuilding

NonpropertyBuilding class is a little special. In its movement() and tuition() override methods, we implement different logics and returns for different buildings. For SLC and needles hall, the tuition and movement are random, so we use the randomGen methods we implemented at watutil.h to generate a random number and produce the results based on the generated number.

## Subject

Subject class is a modified version of subject class from assignment 4. Our version only has one template class. We add a notify after each attach, since many fields in the class that inherits observer class will need that information to initiate.

## Observer

Observer class is also modified from assignment 4. Our version only has one template class.

## Info

info.h contains almost all the game constants, enumeration classes, PlayerInfo and BuildingInfo struct, and some useful little functions. The bntostr and strtobn is really helpful when dealing with property input and output, which transform BuildingName enum class member and std::string. And since all these constants are stored in this header file, it's easy to change datas like the cost of a certain building or the cost of improvement or tuition only in this file, and there is no need to change other implementation files.

## WatUtils

watutils.h also contains many useful functions, like randomGen that produces a random number between high and low, isNumeric that tells if a string is a number or not.

## Resilience to Change

Our design is able to accommodate new features and changes to existing features in various ways.

Firstly, we used an info.h file to record all the building information including names of buildings and monopoly blocks, costs of buildings and their improvements. This enables us to add more buildings if we want, or make any changes to the cost of buildings and their improvements simply by changing the information in info.h without changing the implementation of the building class.

Secondly, we implement all commands in separate functions, and if we need to change the logic of one certain command, we can modify its corresponding function only.

## Answers to Questions

**Question:** After reading this subsection, would the Observer Pattern be a good pattern to use when implementing a game board? Why or why not?
**Answer:** Yes, similar to assignment 4 question 3, an observer pattern can notify the board class only when there are changes inside building or player classes that will result in changes in text displayed.

**Question:** 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:** I don't find a proper design pattern that is suitable for model SLC and Needles Hall. In my opinion, these two buildings can be implemented by placing a random number generator inside, by which the number generated each time will determine the behaviour of players on that building.

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

**Answer:** Decorator Pattern might not be a good pattern to use for improvement. Since each improvement level will return different additions on tuition which also differ from building to building, and there is a limit of 5 improvements maximum, if we implement it as a decorator, we need to implement too many decorators or too complicated logic inside decorators. Instead, an integer that saves the number of improvements of a building will be good enough to represent and easy to implement the tuition function.

## Extra Credit Features

We added a feature of the board display theme, we’ve designed two and there could be more. The theme file contains all vertices of the building square, and our board class will automatically generate the board display text based on those vertices.

# 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?
This project has taught us many things about developing software in teams.

**Answer:** Firstly, plan and design of the program before implementation is especially important in team work. As we may have different ideas and strategies, it is important to come up with one design that is recognized by all members and should be followed precisely to avoid change of structure in later stages. As we are usually in charge of different parts, we should also bring up concerns and problems we anticipate with our own implementation to figure out solutions as a group.

Secondly, clear communication is important in improving the efficiency of working in teams. It often happens that team members write or modify the same part of code, which is a waste of time and effort; or there are some unwritten or not tested parts that were not assigned to any of the members. Therefore, it should be communicated clearly that who is in charge of which parts and team members should update the work they have done frequently to avoid repetition of or undone work.

Thirdy, it is a good practice to write more comments in the code. As classes and files have close relationships with one another, we often need to use methods or functions written by another member in our own files. It may be hard and time-consuming to read the whole implementation written by others to understand the precise purposes so that it is correctly used. While with clear comments explaining the purposes and logic of functions, we can easily refer to those functions in a desired way.

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

**Answer:** If we had the chance to start over, we would arrange more meetings to make sure we were all on the right track and to discuss problems encountered more effectively. We would also start implementation earlier so as to reduce our stress level on the last few days before the due date.

We should do proper analysis of the algorithm of each command before implementation. At our implementation stage, we added many accessors/modifiers to base Player and Building class, and some text output is also produced inside base classes rather than gameplay, which is hard to remember whether a condition has been added or not.

The function of the buildings can be better abstracted. Although we abstract certain functionality of building into tuition() and movement() functions, it is still the gameplay's function to determine the type of the building and call the respective functions. I think it's possible to abstract it, as some functions like action() and it can help reduce coupling between buildings and the gameplay.

## Conclusion

In conclusion, we have used many of the object-oriented programming strategies we learnt in CS 246 to create the game of watopoly successfully. This is a fun University of Waterloo version of monopoly which not only strengthens our understanding of the knowledge from the course but also teaches us the importance of teamwork. We will continue to develop our skills in working on large programs and we really appreciate the opportunity offered that gave us such a wonderful experience!