# CS246 Final Project Watopoly - Design

**By Martin Kong, Zhaocheng Che, Xiaoyu Zhu**

## Introduction

This is a University of Waterloo version of the Monopoly board game. The submitted version of Watopoly supports all the standard features and some enhancements: changeable House rules and Graphical Display. There are no known unfixed bugs; for our decisions on unspecified program behaviours, please refer to demo. This design document explains how we apply OOP principles and Design Patterns to structure the program. By increasing cohesion, decreasing coupling, and following RAII thoroughly, our solution provides excellent resilience and compilation dependence, thus makes further maintenance and extension jobs easier.

## Overview

First of all, our design mainly revolves around the Model-View-Controller (MVC) design pattern. We have a seperate Model, View and Controller class for each part. The Controller class is responsible for handling user input and translating it to the appropriate "actions" which will be carried out by the Model class.

The Model class is responsible for managing the game state, as well as modifying the game state according to the appropriate game logic. The Model class has a set of public methods which can be called by the Controller. The Model class owns a vector of Player class representing the set of all players currently in the game. The Model class also owns an instance of the Board class, which represents the game board of 40 squares. The Board class in turn owns a vector of Square class representing the set of 40 squares on the game board. The Board class also provides a bunch of utility methods that can be called to retrieve information from the set of squares.

The Square class has its own inheritance hierarchy. The Square class itself is the abstract base class, and it holds information that is common in all types of squares. The Building class is a subclass of the Square class, and it represents all the squares that can be purchased by the player. The Building class has three subclasses, AcademicBuilding, Residence and Gym. The entire inheritance hierarchy of the Square class is solely used to store information about each square, and they are not responsible for doing anything else.

Finally, the last piece of the MVC design pattern is the View class. It is responsible for displaying the current game state to the user in an appropriate form. In order to implement the View class, we have made use of the Observer design pattern. The View class is the observer, and the Player class and Building class are the subjects. When a player changes location or when a building changes owner or improvement level, the View class will be notified so that it can update the display. The View class has two subclasses, TextView and GraphicsView, whose names are self-explanatory.

A few things left that I need to mention. First, we have implemented the Model class using the Pimpl Idiom and the Bridge design pattern. Furthermore, we have also made use of the Strategy design pattern to implement all the possible consequences when a player lands on a square. For example, when a player lands on SLC square, the corresponding SLCStrategy will be used to handle the situation. Given that this is an overview, we will not go into too much detail. We will talk more about these design patterns in subsequent sections.

## Design

At the time of writing actual code, various challenges appear from the DDL1 design.

**Cyclic Reference:**

According to the logic of the game, there is an association from player to building, specifically, one player owns many buildings, with other specific relationships (whether in the same monopoly) between buildings. However, if we implement the relationship by association, then cyclic reference is inevitable. For example, by letting buildings observe their neighbours, there must be a pair of buildings holding a pointer to each other.

To solve the problem, we transfer the responsibility of representing this relationship entirely to the class Board. By calling public methods such as getOwner, Model and Strategies can access the information through the Board class. Currently, we maintain a vector of Buildings, a vector of corresponding owners, and a vector of vector of Buildings for monopolies. In this way, we are able to avoid raw pointer and stick to RAII principle thoroughly. In fact, we use only shared pointers for all code.

Another benefit of such a solution is the decrease of coupling and the increase of cohesion. We reduce the coupling between player and building: they do not even need to know anything about each other now. The responsibility of maintaining the relationship among player and building is concentrated in a single class rather than spread among Players and Buildings, which practically avoids both cyclic reference and complex structure. In conclusion, the degree of extensibility and the difficulty of maintenance are extensively decreased.

For example, if there is a need to change the way relationship is stored, (by map or vector or whatever), we only need to change the specific implementation of Board, without tampering the code in Player or Model.

**Model's Role and the associated design patterns:**

According to MVC, the Model's role is to store information about the game state and manipulate the data when requested by the Controller. However, in this game, many problems arise. For example, some of the commands such as trade have a direct effect on the ownership of the building and the player's money, while others such as roll would require keeping track of the number of doubles that have been rolled. What's more, some of the game logic such as prison would require additional interaction with the user. If we embody all of these functionalities in a single Model class, it would be a huge class which is hard to navigate in, not to mention maintaining this gigantic class. To solve this problem, we have applied the Pimpl Idiom, the Bridge design pattern, and the Strategy design pattern.

<u>Pimpl:</u>

To start off, a challenge that we have faced is that given the size of the Model class, its implementation details need to be updated very frequently whenever we decide to add another cool new feature. Even if we are merely changing private fields and methods, the header file of Model will need to be changed, which means that any other files which include Model.h will also need to be recompiled. This is obviously not desirable.

In order to solve this problem, we have used the Pimpl Idiom so that Model's public interface is completely separated from its implementation details. The Model class itself is just a public interface, which contains a set of public methods such as trade, improve, mortgage, etc. The Controller class can call these public methods as needed. But the Model class itself is not responsible for actually implementing these public methods. Instead, we have created a ModelImplementation class which encapsulates the implementation details of the Model.

To be more explicit, the only private member of the Model class is a pointer to an instance of the ModelImplementation class. The Model class does not have any other private member. When one of the public methods of Model is called, the Model will ask the ModelImplementation class to carry out the task by calling the appropriate method of the ModelImplementation class. This is the Pimpl (Pointer to Implementation) Idiom.

The benefit of using the Pimpl Idiom is that if we decide to change the implementation details of the Model class later on (which is very likely to happen given that all the game logic is contained in the Model class), we only need to change the header file of the ModelImplementation class so that its private members can be changed.

But the header file of the Model class does not need to be changed, because it only has one private member which is a pointer to its implementation! This means that we only need to recompile the ModelImplementation class and the Model class, but we do not need to recompile any other files which include Model.h. Hence separate compilation is achieved.

Bridge:

The next problem that we faced is that, as we have mentioned previously, the Model class is just too big. Even if we put all of the implementation details of Model into the ModelImplementation class, that would just make the ModelImplementation class very big. We still have a nasty gigantic class that is very hard to maintain. In order to solve this problem, we have extended the Pimpl Idiom into the Bridge design pattern. The Model class will still hold a pointer to the ModelImplementation class and ask the ModelImplementation class to do all the work. But the ModelImplementation class will have its own hierarchy!

The idea is that different subclasses of the ModelImplementation class will be responsible for implementing different aspects of the game logic. For example, subclasses of ModelImplementation can be responsible for implementing a player's movement, while subclasses of subclasses of ModelImplementation will be responsible for implementing the enhancement of properties, and so on.

One obvious benefit of such a design is that we have achieved high cohesion! Each one of the subclasses of ModelImplementation will only be responsible for exactly one aspect of the game logic, so each class performs exactly one task, which means we have successfully followed the Single Responsibility Principle. Another direct benefit of such a design is that each one of these classes is much shorter and cleaner. We no longer have a gigantic Model class handling all of the game logic. Instead, the responsibility of Model class is broken down into separate pieces, which will be handled by different subclasses of ModelImplementation. As a result, our code base is much cleaner and much easier to maintain.

Finally, the most important benefit of such a design is that it is very resilient to changes. The main advantage of the Bridge design pattern is that the public interface is separated from the implementation details, allowing the two to vary independently. That means when we want to change the game logic (for example, we might want to change the game logic to implement the house rules or the even build rule), we can simply create another subclass of ModelImplementation to handle that change in game logic! We will talk more about this in the "Resilience to Change" section.

Strategy:

Naturally, the bunch of functions to handle the event of a player passing or visiting a Square should form a specific group, as they are triggered by the same event. However, the results of them vary significantly: paying rent would only require the information about the player and the building's owner and its rent, but waiting in DC Tims Line would require rolling dice and interacting with the user for choosing from a category of options. If the functionalities are simpler, we would want to put them in the Squares themselves as that would decently fit in a dispatched call, which is in fact the decision we made in DDL1. However, because these functions require almost all of the data, we later think of a more rational solution to store them in the model, to decouple the Squares and Buildings from the rest of the Model, so that maintenance and extension would be easier.

Currently, the model stores a map of Strategies to handle the event of a player passing or visiting a Square. The strategies would take in a Player, a Board, and an interaction interface with the user. The degree of exposure of game data is quite high, providing more power to the strategies. One benefit from such a decision is the easy accommodation for more strategies: new strategy is provided with any arbitraries functionality as long as it depends on only one player (this is already enough since functionalities involving more players are to be handled by other functions in the model) (and refer to the next section for more details). In conclusion, Strategy pattern

enables us to switch strategy on a certain square position dynamically at runtime, and easily maintain the group of functionalities handling the event of a player passing or visiting a Square.

Controller:

Controller provides a macro level of decoupling. Controller is decoupled from Model: it has no knowledge of how the data is stored and manipulated. Its sole responsibility is to provide an interface for the users and filter the commands for the model (Command Interpreter).

Observer and View:

We implemented the View class so that, whenever a View is notified by a subject (a player or a building), the View will immediately update its data, and modify the game board (the TextView will redraw a new game board in the standard output, and the GraphicView will redraw the changed squares on the display window). Due to the advantage of the Observer design pattern, the View class does not have to know anything about the implementation of the player class and the building class. When notified by a subject, View will simply call that subject's overloaded getSubjectInfo() method which returns an Info struct, and since the subject is passed by reference to View's notify() method, the Info struct correctly contains all the required information to draw the subject on the view display. So any changes to the Player class or the Building class will not cause the View class to recompile.

One challenge of implementing the TextView classes is, drawing the game board in the standard output requires the TextView to output the text representation of the board line by line from the top, not block by block from Collect OSAP. We introduced a private nested Block class in TextView to resolve the problem. Each Block contains a vector of strings where each string represents a line in the Block's text drawing. And the TextView holds a 2-d vector of the Blocks. Whenever a change to a square is applied, for example, a player moved into or out of the square, the TextView will translate the square's 1-d coordinate two its 2-d position on the game board, then find the corresponding Block in the 2-d vector and changes the strings stored in that Block. When the TextView draws the game board, it will loop through each row of the 2-d Block vector, output the current row's Blocks' first line, then the current row's Blocks' second line, etc. This solution has the benefit that the layout of the text display only needs to be thoroughly calculated once at the start of the game, and we can easily change the board's theme by modifying the Block class's implementation.

## Resilience to Change

initializing file:

We created an initializing file that stores all the required information of a Watopoly game board: the number of squares, each square's type, monopoly block, name, cost, etc. And when the game first runs, it will read in the data stored in the initializing file, and initializes the Controller, Model, Board, Views, according to the data. By storing the game's initialization data in a separate file, we can simply modify that file if we want to change the squares' information based on the current rules, for example, increasing the total number of squares, or regrouping the monopoly blocks. So the program does not even need to be recompiled with such modifications.

Changing squares:

For simple changes on squares, such as adding 4 more SLC squares, changing the name of ML to MML, or let B1 become a gym, we can simply change the initializing file as stated above, and no recompilation will be required at all.

When changing a square's function, the Strategy subclass and the Square subclass corresponding to this certain square need to be modified. And because of the implementation of Strategy pattern and object orientation pattern, in most parts of the program, the code only needs to know an object is a Square/Strategy, and does not need knowing what specific kind of Square/Strategy the object is, so most relevant classes just have to include the Square class and the Strategy. And by changing a square's function, merely the square's Square subclass, Strategy subclass, the main function are recompiled, which enhances separate compilation.

When adding squares with brand new properties and functions. A new strategy class that inherits the VisitorStrategy class needs to be implemented for each of the new squares. And if some of the squares have a high resemblance and need to store information that are different for each instance, we will create a new subclass of Square in order to increase the code's reusability. Also, if the new squares require special display on the game board, the View subclasses will also be modified to accommodate the change. Then we only need to modify the main function which reads in the squares' information, and initializes the square classes and square strategies. And similar to changing squares' functions, only the new Square subclass, the new Strategy subclass, the main function, and the View subclasses may have to be compiled again.

House rules

We have elaborated on how we used the Bridge design pattern to create an inheritance hierarchy of the ModelImplementation class (see the "Design" section). Due to this design pattern, we can add new custom house rules very easily if we want. Whenever we need to change some of the game logic to account for the custom house rules, we can simply extend the inheritance hierarchy of ModelImplementation.

Let's work through a concrete example. Suppose we want to implement the "even build" rule. In order to do so, we can create a new class called ModelImplHouseRules which will extend ModelImplDefault (ModelImplDefault represents the default rules of the game). Inside ModelImplHouseRules, we can override the superclass method isImprovable, which is responsible for checking whether or not a square can be improved. In the overridden version we can add an additional check for the "even build" rule, and we can call the superclass's version of isImprovable for the other checks. As shown, the amount of code that needs to be added/changed is kept to a bare minimum.

In terms of separate compilation, the only class which knows about ModelImplementation is the Model class. Hence when we add a new class to the inheritance hierarchy of ModelImplementation, we only need to recompile Model.cc. Since Model.h only specifies a pointer to the base class ModelImplementation without any mention of its subclasses, Model.h does not need to be changed, so any other files which include Model.h does not need to be recompiled.

Chance cards

By the idea of using Strategy Pattern to handle the event of a player passing or visiting a Square (see section in Design), we are able to add any new functionalities as a Strategy as long as it satisfy the conditions: dependence on only one player (can only access other players indirectly as building owners), and the event resolved immediately without lasting effects (except go to Jail).

Under these condition, we can easily add a new strategy for the functionality and let the Chance Cards / Community Chest strategies call them randomly according to their corresponding probability. Furthermore, because the strategies are constructed in main, it is even feasible to let the player select the possible card sets before the game starts.

Adding a new strategy only requires recompilation of main and any other strategies that might call it, e.g. Chance Cards. In other words, it is quite viable in terms of recompilation resource. Also, very little code need to be changed (only the constructor in main), so possibility of introducing error is quite low.

For more information about how Strategy Pattern helps us with Chance Cards and Community Chest, see the answer to the second question in the next section.

## Answers to Questions

1. Would the Observer Pattern be a good pattern to use when implementing a game board? Why or why not?
Yes, we will implement the Observer Pattern such that Player and Building classes are the Subjects, and TextView and GraphicsView classes are the Observers. Whenever the position of a player or the number of improvements of a building changes, the View will be notified and the game board can be redrawn appropriately. Observer Pattern is a good choice due to the following reasons:

1. We need a one-to-many dependency between the set of players and buildings and the Text/Graphics Viewer. On the other hand, the Text/Graphics Viewer needs to keep track of the state of each player and building. So the Observer Pattern will be very useful in this circumstance.
2. Observer Pattern allows for cleaner control flow. With the Observer Pattern, the Text/Graphics View will be "automatically" notified when the state of the player/building changes. Without the Observer Pattern, we need to explicitly change the View ***whenever*** we want to change the state of the player/building, which is clearly going to be painful.
3. Observer Pattern supports the principle of loose coupling between objects that interact with each other. Subjects and Observers interact and only need to know that the classes follow the specified interface, i.e. they do not need to know the specific details of the concrete classes other than what is specified by the Subject and Observer abstract base class. This is good because in general, loosely coupled objects are flexible with changing requirements.

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?
We could use the Strategy Pattern. We will implement each Chance and Community Chest card having an attribute of Strategy. This Strategy, by maintaining a list of possible concrete strategies, we can determine at runtime, by a random seed, which concrete strategy to apply on. This implementation has three benefits:

1. Encapsulation: classes outside these two classes can treat them the same as the other non-property Squares. The behaviours are encapsulated in a single acceptVisitor method, as a unified interface to outside callers.
2. Maintenance: it is highly possible to add new possibilities of these chance cards in the future. Using strategy pattern enables us to easily subclass the Strategy interface with these new features and include them in the Cards' strategy lists. This means we only have to modify the minimum amount of code.
3. Dependency: Since the implementation of the Cards does not involve deciding which strategy to use, it does not depend on the strategies, only depending on a unified strategy interface.

Compared to overriding the function by creating many subclasses of the Chance Card, Strategy Pattern can change the behaviour of these classes at runtime without having to expose the complicated structure in class Board. This improves flexibility and extensibility.

3. Is the Decorator Design Pattern a good pattern to use when implementing Improvements? Why or why not?
No. The Decorator Design Pattern is clearly a bad choice when it comes to implementing the Improvements. The reasons are as follows:

1. Decorator Pattern cannot possibly put a restriction on the number of decorator being used. That is, if we use the Decorator Pattern to implement the Improvements, we cannot put a restriction on the number of

Improvements made to a particular building. But according to the project specification, each building can only have a maximum of 5 Improvements, so the Decorator Pattern is clearly a bad choice.

2. The whole point of using Decorator Pattern is that we can dynamically add *any combination* of functionalities to an existing object. That is, we can implement each different functionality as a subclass of Decorator, then add *any combination* of them to an existing object. But in this case, there is only one possible "functionality" (i.e. the Improvement) that can be added to the building. Hence we no longer have the largest benefit associated with Decorator Pattern.
3. The tuition fee of a building varies according to the number of times the building has already been improved. Furthermore, different buildings have totally different sets of tuition fees. Hence when we are trying to implement the concrete Decorator class, we have a problem: What exactly should the Decorator do? How much money should it charge in tuition?

Due to these pitfalls of using Decorator Pattern, we choose to simply add a private field in the building class called improvementLevel, which represents the number of times a building has been improved. In addition, it will have an array of 6 integers, whose ith element represents the amount of tuition fee when improvementLevel = i. Then, the accessor method can simply return improvementFee[i].

## Extra Credit Features

1. Smart Pointer. We have completed the entire project without leaks, and without explicitly managing our own memory. In other words, we have handled all memory management via STL containers and smart (no weak) pointers. That is, there are no "new" or "delete" statements at all in our program.

2. Graphic Display. We implemented a GraphicView class for the purpose of displaying the game board and the players graphically as the game goes on. This feature can be turned on by the command line option "-vGraph". In the Graphic display window:

- Each Square's position and name is correctly drawn.
- Non-building squares, such as SLC, are drawn with a light yellow background to distinguish with other ownable buildings.
- Each building's improvement level will be correctly shown on the window.
- Each player's symbol is placed on the correct square in an appropriate location. Each player will also be assigned with a distinguishable colour, and the buildings owned by the player will be drawn in the same colour.

One difficulty we meet when implementing the GraphicView is, redrawing the entire board whenever a player moves to a new position or a building changes its improvement or ownership takes up a lot of time, and will provide the players with a poor gaming experience. We resolved this challenge by calling the drawboard() method(which draws the entire board on the window) only once at the start of the game, and only redraw the squares that have been changed when notified by a player or an academic building.

This raises the second problem, since we need to create the GraphicView object first, and attach it to the players and buildings, a blank window will pop up before the players enter their symbols and names. Also when loading a saved game, the improvement level and ownership of a building may change, and will cause the GraphicView to draw the improved building before the game starts. We solved this challenge by using the advantage that the drawboard() function will only be called once at the start of the game, so we made the GraphicView not to draw anything and not to map its window, until the drawboard() function is called once.

3. House Rules and "Even Build" Rule. We have implemented the three house rules specified in the watopoly.pdf for extra credit. The three house rules are:

- “Money collection from Tuition, Coop Fee, and DC Tims Line goes to the center of the board. Landing on Goose Nesting gives a Player all the money in the center.”
- “The above except that the center starts with $500.”
- “Landing on “Collect OSAP" doubles the amount received.”

We have also implemented the “even build” rule in the classic Monopoly game.
If you want to activate the first house rule, you can use the command line argument -rule1. Similarly use -rule2 or -rule3 if you want the second or third house rule. And use -rule4 if you want to activate the “even build” rule. Any combination of these rules can be used at the same time. We have talked about how we implemented these house rules in the “Resilience to change” section, so we will not repeat these things again.

## Final Questions

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?

First, full discussion on the logic of the program can drastically reduce potential bugs and possible bottlenecks for features. One person might overlook some specific corner cases and propose a design that leads to a difficult situation on implementing these. However, a team, together, can unveil most of the hidden problems and come up with a more full considered design.

Secondly, when implementing the program, group members should try to avoid editing the same file at the same time, otherwise merging different branches of the code and deciding whose code to use will waste a lot of time.

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

First, we would read the specification more carefully. We thought that what determines a good OOP design is how the program handles the general case, but it turns out that the real indication is how delicate the corner cases are handled. A better strategy to solve the design problem would be to list all the possible features and choose the design supporting the maximum number of features.

Secondly, we would start the project earlier. Even though we did not actually run out of time for this project, having more time would obviously be very helpful. We could spend more time to fine tune the project and/or work on more bonus features.

## Conclusion

In a nutshell, compared to the first attempt, our new design demonstrates better OOP principles and applies more design techniques we learned in this course. Using Observer, Bridge, and Strategy Pattern, our solution decreases coupling and increases cohesion; organizing game data by Board class, our solution is able to follow the RAII Principle thoroughly. As a result, the project demonstrates excellent extensibility and memory management ability.