CS 246 Final Design Report -

University of Waterloo

Watopoly

Cedric Wang, Akshita Choudhury, Aditya Uppal

Table of Content

Introduction:

This project involves the development of Watopoly, a video game that offers a unique twist on the classic board game, Monopoly. The game features a game board that replicates the University of Waterloo campus.

In Watopoly, there exist two types of buildings - ownable and non-ownable, the game board consists of 40 squares representing those, each triggering a specific action when landed on. The ultimate objective of the game is to be the last player standing without going bankrupt. Gameplay involves taking turns to move around the board, purchasing and upgrading on-campus buildings (properties), and paying tuition (rent).

Hope you enjoy playing the game!

Overview:

The implementation of the Watopoly video game will make use of the Model-View-Controller (MVC) architecture. In this pattern, the model (game class) represents the logic of the game, the controller (controller class) consists of the game commands which the user enters, and the view (view class) is responsible for displaying the game's interface. Class Player handles all the data regarding a player and class BoardSquare is a parent class from which all types of buildings, and other locations on the board will be derived.

As the name suggests, the *controller* acts as an *intermediary* between the model and the view. It handles user input from the view and updates the model accordingly. The controller also communicates with the view to update its display based on the state of the model. The Observer pattern is used to keep the view and the model synchronized, where any changes made in the model are reflected in the view. Here, game class is the subject and view acts like an observer, so is the controller. The game class should notify its observers (view and controller) of any changes made to its state.

We have two types of buildings as mentioned above- ownable and non-ownable, where academic, gym and residence inherit from Ownable, which means where rent (tuition) is applicable. Other boardsqaure like SLC, Needles Hall, coop fee, tuition, collect OSAP, goose nesting, Tim’s line and go to Tims inherit from non-ownable class and all of them have unique functionalities. Similarly, the BoardSquares are also subjects, and when their state change, they also notify their observers (view and controller).

We are also using the *Visitor pattern* to *visit* the buildings (BoardSqaures) when the Player lands on it. Each Player "visits" a board square and each square "accepts" the player. It allows us to keep the object structure and the new functionality separate, making the code more maintainable and easier to extend.

Updated UML

(Refer to UML.pdf)

Design:

In this section we will describe the important design patterns we used and the overall design of the entire software, along with a description of the main classes used.

BoardSquare : Class BoardSqaure (boardsquare.h/.cc) is a class that represents any building or location / occurrence on any square on the board. We use int position (field) to keep track of where on the board the square is, and other fields and get methods for cost, name, type etc.. BoardSquare also has a struct GameState which has fields to keep track of the state of the game while saving and loading the game. We are keeping track of the squares using a vector of shared pointers of the BoardSquare class. The underlying subclass that the shared pointer is pointing to will be used to notify View when a player has arrived at that square or left that square.
There are two classes that inherit from BoardSquare – Ownable and NonOwnable (in the respective .h/.cc files). As the name suggests, Ownable buildings are those which can be owned by a player, and NonOwnable is everything on the board that cannot be owned by a player.

- **Class Ownable:** has methods/fields for mortgaging property and handling owners. From Ownable we have three other concrete children classes – class Academic, Residence and Gym (in the respective .h/.cc). Class Academic has all methods / fields related to academic buildings , for handling improvements, tuition etc. While Class Residence and Gym, have methods / fields used to handle rent, owners etc. They can also notify the Controller and View for actions such as when a player owes tuition, mortgages/remortgages a property, buys/sells improvements, and does not have enough money to pay tuition.

- **Class NonOwnable:** we have multiple children Classes for each type of Non – ownable square : SLC, TimsLine, NeedlesHall, Tuition, Gym, GoToTims, GooseNesting, CoopFee, CollectOSAP (In their respective .h/.cc files). They can also notify the Controller and the View for actions such as gaining/losing money and moving to another square.

Each of these concrete BoardSquare Classes have their own accept method that is called based on the type of BoardSqaure. The accept method takes a player &p as a parameter and performs the required actions on receiving the player (accept method being called).

Player: Class Player is used to keep track of a particular player's information, such as name, turn, money, net worth, properties owned etc. A Player has a map of properties owned, and has several public methods to check if the player has a monopoly, goes bankrupt, add/remove buildings, receive Rim Cups, move (forward and back), receive / pay money etc. The Player also has get-methods defined, to be used by other classes to get its fields.

Game : class Game (game.h/.cc) is the "model" that manages the state of the game. Class Game has a vector of pointers to class BoardSquare, implemented using shared pointers to avoid memory leaks. This is the layout of the board, which can be changed by re-ordering the vector. Class Game also has a vector of Player classes, which are the players actively playing the game and has class Dice. Game manages the entire state of the game, and all functions / methods are called from the class game. Game has several methods to add / remove players, switch between players, get player information, set player positions, change building owners, mortgage, unmortgage, transfer, trade, buy/sell improvements, check for monopoly, Auction, get building states, bankrupt players, etc. (i.e., all methods in regard to gameplay). Most of these methods are implemented with the help of methods from the Player and BoardSquare classes. Game Inherits from abstract class Subject.

View : Class view (view.h/.cc) is used to display the game whenever its required. Class view has a map of player positions and a vector of string vectors, which is the Display. Each players character as seen on the board has been color coded to a different color, to improve visibility on the Board. View has a method to convert the BoardSqaure Number to a struct Coordinate, which allows the BoardSqaure to get printed to the correct position. View also displays owners and improvements made to the square in the iteration of the game state provided. The printBoard method prints the display. View inherits from the Observer class. View has a notify method where it receives the current GameState object, and updates all the fields of View before calling print Board.

Controller : Class Controller (controller.h/.cc) is used to handle all interactions with the players. It receives all commands from standard input. Controller has a class Game and class View.
The main function has a Controller, and it is therefore the entry point to the program and has methods to launch the game, select players, prompt Text (trade, auction, mortgage, improvement, invalid input messages..), save, load the game, keep track of player turns and handle all player commands by calling the appropriate methods in the Game, View classes. Controller inherits from the Observer class and will call certain methods when it receives notifications.

Observer Pattern (MVC) : We used the "Model", "View", "Controller" architecture, implemented using the observer pattern. We have abstract classes Subject and Observer (in the respective .h/.cc files). Class game which is the "model" in MVC inherits from class subject. While class Controller and View are the observers. The class Game manages the state of the game and notifies the observers to perform certain actions like display the game (View), or prompt questions to the players (Controller). All function / method calls, from Controller (player input), are made to the Game class which handles the requests and notifies the observers accordingly. The integration of the Observer pattern with the MVC pattern results in a more flexible and scalable architecture. This design pattern promotes a separation of concerns, as each component has a specific responsibility in the game's functionality. It is efficient and easy to update or make changes as we only need to modify the game class and notify instead of creating a new game each time.

Visitor Pattern:
We are using the visitor design pattern to implement the relationship between class BoardSquare and class Player. Every type of BoardSquare has an accept method, taking a Player & as the parameter. The accept method implements the main functionality of that square. Player visits the

BoardSquare. The accept method chosen depends on the type of BoardSquare pointer. This makes Implementation much easier as we do not need to worry about the types while calling the accept method, as the visitor pattern handles this for us.

Resilience to change:

As we have seen in the UML, All important tasks have been separated into different modules, each of which have a .h and .cc files for: boardsquare, Game, View, Controller, Ownable, NonOwnable, Academic, Residence, etc. Each module has its own classes and children classes with all the relevant methods implemented. For example, we can add a new player by pushing a new player object to our vector of players, and it’s the same for adding new buildings. For other types of Board Squares, as we are using a visitor pattern, we only need to add the new Type of building / square as a module, inheriting from BoardSquare and not worry about calling the right methods elsewhere in the code. Our game is robust and is flexible to accommodate changes.
As previously described in the design section, MVC architecture was used, with Game and View, Controller having a subject - observer relationship. For the board display, our View class reads from a text file, so changing the board simply implies changing the text file, which is a lot more scalable than hard coding each character to our code.
Like all well designed software, we aimed to design our modules with Low coupling and High cohesion. Each module communicates with other modules only using function / method calls, and sometimes passing objects (by address) as parameters. All Classes have private and protected fields, which can only be accessed by other modules using the get methods we implemented. Classes cannot mutate each other’s fields directly.
Therefore, anytime we changed, or added anything to one module - the other modules would be completely unaffected by this change, and we would only need to recompile that particular module. Thus, our modules have low coupling and are reusable.
Our modules have high cohesion. Most of the modules are class Implementations, and each class was designed to handle a specific part of the game, such as Player, Academic buildings, Residence buildings, or SLC, Needle halls etc. . They are all separate concrete classes used to solve their particular tasks. We also have very high cohesion using the MVC architectural pattern. The core functionality of managing, interacting with and viewing the Watopoly game

board was divided into separate modules, each of which was designed to focus on one particular task.

We had the view module with View Class which had different methods all implemented for the sole task of updating and displaying (printing) the gameboard every time there was a change of state. We had the game module with the game class, with the single task of managing the game state. We had the controller module, with the Controller class which is be used to receive commands from the players and based on them call methods of game, or view to “model” the game. Therefore, our modules have very high cohesion, with each module being created to solve a particular task.

As the software is well designed based on coupling and cohesion, it is resilient to change, and we could easily implement more features and functionalities to Watopoly in the future.

For example, while Implementing player commands (trade, mortgage, etc..) we only had to add / alter method certain methods in the relevant classes and recompile those modules.

Extra Credit Feature:

We only used vectors and *smart pointers* (no delete statements), our program handles memory effectively without any leakage. We also added different colors for the players which shows which player owns which BoardSqaures and which properties are mortgaged. It makes the gameboard more visually appealing and easier to navigate as the colors pop.

Questions:

(From Project specifications)

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

Yes, The observer Pattern will be the ideal pattern to use when implementing the game of Watopoly. We implement it with the MVC architecture principles. The Game class acts as the model which is the subject, which owns the BoardSquare (buildings) and the Player class. We also have a Controller class that owns the Game and the View. View and Controller are the observers. Controller acts as a medium between the model and the view by taking in commands from the user.

The Observer pattern enables the Game and the BoardSquares (including all the subclasses) to notify its observers, view class and controller class, ensuring that any changes made in the model are reflected in the view or prompt any action in the controller. It is particularly useful because it allows objects to be notified of changes to the state of another object without tightly coupling them together. Low coupling is preferable in real world scenarios.
Note : *We stuck to our original plan (dd1)*

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?

Initially, we thought of implementing the Template method pattern since SLC and Needles Hall are pretty unique and different from the other buildings. Thus, we can treat them separately. Moreover, they are both impacted by the shuffle method for random number generation which affects the actions taken by the player. Thus, the odds of the random number generator and the outcomes would change depending on if the building is SLC or Needle’s Hall, which can be handled well using the Template Method Pattern. In the real Monopoly, there is the same amount of Chance and Community Chest cards. If we wanted to model SLC and Needle’s Hall in a similar fashion, we can create a superclass from which both SLC and Needle’s Hall inherit. The superclass will have a method, let’s denote it as pickCardAndGetOutcome that will generate a number from 1 to n (through the pickCard method) and will then call a method, let’s call it GetOutcome, that will determine the outcome for the player depending on the number picked. Thus, through the Template Pattern, SLC and Needle’s Hall will only customize some portion of the superclass behaviour, so they will not override the pickCardAndGetOutcome method. Instead, they override the getOutcome method since the pickCard will always have the same behaviour. Thus, for either SLC or Needle’s Hall, the outcomes will be different depending on the number picked. However, when we started coding the project up, we followed the Watopoly guidelines, so it made more sense to use the rand() function. We created subclasses- SLC and NeedlesHall inheriting from Non-Ownable class. We generated the output depending on the probability given to us using rand(). It made everything simpler.

Note : *We completely deviated from our initial plan (dd1), our idea was to use the shuffle method for random number generation which affects the actions taken by the player. Thus, the odds of the random number generator and the outcomes would change depending if the building is SLC or Needle's Hall, which can be handled well using the Template Method Pattern. But what we finally implemented was easier, we didn't want to complicate it.*

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

No, we initially thought it would have been a good approach but later decided not to go with it as the only change was in the amount of tuition with different improvement levels. As we only had to update one thing, it was much simpler and efficient to modify its method rather than adding a decorator. Decorator could have added unnecessary complexity in our case.
Note : *We deviated from our original plan (dd1). We thought it would have been a good approach but were skeptical as the decorator works best with multiple-add ons. Revisiting our plan helped.*

Final questions:

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?

Cedric: I think it was knowing when to use and not use design patterns. Sometimes using design patterns like in the case of SLC/ NeedlesHall was complicating it. In my opinion, it is more important to have an understandable, easy to follow implementation than something which is super efficient but hard to follow, especially when working in groups. The entire project helped me learn Git better for version control and that committing often leads to better efficiency.

Akshita: The most important lesson I learned was the power of UML and how much it can simplify things. Spending time on UML was extremely beneficial as it helped me understand the workflow and how to code. When working in a group, it might get chaotic if we don't stick to our assigned task and cause issues while merging and we need to revisit the code multiple times to debug something trivial.

<u>Aditya:</u> Watopoly not only strengthened my concepts like coupling and cohesion which were once theory based, but also taught the power of Git. It helped me picture how projects are handled in real world scenarios and reducing dependency makes it easier to debug. It enriched my planning skills too.

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

Although we spent 2 days refining the UML, we believe we could have optimized even better and integrated more design patterns. If we enhance our design, we would have reduced coupling and improve cohesion. As of right now, Controller owns Game which owns Player so the controller can only access Player or BoardSquare through Game which is not very efficient as we would need to write a method in Game and a method in Player in order to change a Player object's state. By applying patterns such as the Template Method or by redesigning certain dependencies, we could have made the object classes more cohesive and less coupled. Also, sometimes the Game and BoardSquare subclasses overlap, which can result in redundant code. For example, they both notify the observers. It would have been better if they all had independent tasks. The Game class contains a lot of repetitive methods which could have been reduced by writing more efficient methods. We could improve Watopoly, by simplifying the codebase so that it can be easily modified and extended in the future.