Watopoly: Design Document

May Ly & Michael Qian

University of Waterloo

CS246

July 26, 2022

Structure Overview:

Our structure is quite different from our original plan, but we are content with all our changes. Our game is implemented using the MVC design pattern. Our view class is called “boardview” which handles all the printing that the user will see such as printing the board, printing the assets of a character or printing the trade proposals. Our controller class is called “controller” and that is what the player interracts with in which reads command input and redirects it to the model to handel. Our model is our game class which handles most of the functions in the game and all other classes are a component of the game.

With MVC, we used an observer design pattern which is why we have the abstract classes subject and observer. This was a part of the MVC modelallowed us to continuously update the board with any changes to be printed. We had three subjects who were able to send a notification to our observer (the board) - Game, Player and Square. If a notification were to be sent from either the player or square, it would be dynamic casted to identify who sent the notification, and either a the player will be moved or an improvement will be placed on the board.

Our structure uses lots of different classes which create objects that are all components of the game class. In game, we have instances of the player class to store all the players. In game we store a unique dice object, and a unique game board which come from the classes Board and Dice. The board class is made up of 40 squares objects which make up the game board. The squares are shared pointers and are stored in a vector of size 40. Below the square object, we have the use of inheritance where we have many child classes that represent the major categories of square types: non-ownable and ownable - academic, gyms and residences. In our square class, we also have a virtual action method which will be overridden by everysingle square beneath it to take on the action that pertains to their square specifically. To add, in the original UMLfrom due date 1, we actually had a non-ownable class which all the properties like SLC, Needles Hall, Goose Nesting etc. would inherit from. Shortly after setting up the project however, we realized there would be no methods or attributes to store in the “non-ownable” class so we scraped it. Instead, we had the non-ownable properties as a direct child of square. On the other hand, we had two levels of inheritance for ownable buildings - the ownable class and each ownable type such as academic, gym or residence. This is something we had since the beginning because the actions and attributes that pertain to each building is very different, but making them a child still allows them to inherit the key attributes such as cost, name, owner.

Some of the major changes we made to structure include removing of the non-ownable class and also removing the tims cup object. We realized that it would be much easier to have the cups be a part of the game logic and can just be stored as attributes in the player. Other than that, we kept the structure pretty similar and continued to encapsulate as much as we could as we followed object oriented and MVC principles.

Design:

Our Watopoly implementation takes on many object oriented principles, design patterns and uses of smart pointers.

To start, when planning the original UML diagram before due date 1, we knew we wanted to use the model view controller design pattern. This way, the player only interacts with the controller, and the controller has no information about how the game is implemented. For the view component of MVC, it allowed us to separate our printing functions such as printing assets, the trade proposals or the board. This model view controller design pattern is something we continued to implement throughout the entire project - ensuring we encasuate every piece of data in game.

Another design pattern we used and kept up from due date 1 was the Observer design pattern. Having the observer design pattern allowed us to effortlessly make changes to our board, and allows for loosely coupled code. If a player was moved in the game or an improvement was bought, they would send the notification over to the boardview (the observer) to be changed. In terms of what we would change in our code, if we had more time we would have removed Game as a subject. This was something we thought we needed in our original plan, but did not end up using.

One thing we would like to touch on is our use of smart pointers and RAII principles. When coming up with a solution to how we handle our gameboard, we initially decided to have one class that would manage our gameboard and all the squares included within our gameboard. However, we soon realized that with this type of implementation it became increasingly hard to ensure our program would not leak memory as deleting our gameboard might not mean all our square pointers were deleted as well. Our solution to this problem was to implement RAII, and have a surrounding Board type while binding the squares(which we separated into another class) to the Board itself. This way, we unsure that every square we initialize will always be destroyed by the destructors of our classes since they have a lifetime that is bounded by a temporary object and we enforce encapsulation by hiding our squares in a container class.

Our program structure uses lots of object oriented principles. Starting from the logic of the game, we decided to separate the game and the board. The game was essentially our “Model” in MVC which handled all the game logic, while board (a unique pointer) was simply a holder for all the individual squares. We also had a player class and a dice class which were the components of the game. The players are stored in the game in a vector, and the dice object is stored as a unique pointer. Having a separate dice class from the game allowed us to implement the testing mode easily by initializing the pips for each dice we wanted. This is exactly what we wanted to implement in our original plan before due date 1 and we were able to follow through with it.

Continuing with our object-oriented structure, our game class has a board and our board class has 40 square objects. Under the squares we used lots of inheritance in which was mentioned earlier when discussing the structure. This allowed us to minimize repeated code, as

we wouldn’t have to write key attributes such as name or index for every square, but rather inherit them and continue to add on to the classes with other attributes.

In terms of our actual code implementation, we tried our best to encapsulate and hide as much information as possible by using getters and setters for everything. This was quite tedious, and in the future we probably would have used friend classes more effectively. We also used many instances error handling when dealing with command input from the user or invaid plays such as paying when you don’t have enough money. Having try and catch error handling allowed us to write reliable code that wouldn’t shut down automatically. In addition, we used lots of dynamic casting to determine certain plays we should execute depending if it casted to an ownable building, player, gym etc. This was great because we didn’t need properties such as “bool isProperty”.

Resilience to Change

Our program was written with the intention of it being able to reused in other projects or easily changed if needed. This was predominantly done by using object oriented design principles, low coupling and high cohesion. By using many different classes that are very specific to their tasks, actions and attributes, we made sure that we were creating individual objects that can be swapped out or changed without affecting the others. For example, making a separate dice class allows us to make changes to the dice without affecting the logic of the game. Making a separate board class allows us to change the make up of the board so we can possible increase the size, or change the different squares that belong in the board. Having these separate, lowly coupled classes allow us to make changes with ease.

Questions

1. *After reading this subsection, would the Observer Pattern be a good pattern to use when implementing a game board? Why or why not?*

We believe adding the observer pattern through MVC would be a great pattern to use when implementing the game board. In MVC, the Model and View are a classic implementation of the Observer design pattern, where the Model is the subject and the View is the observer. In our case for Watopoly our BoardView class acts as our view which is responsible for updating and printing the board. The BoardView class receives information from the model or the concrete subjects which are our Game, Square and Player classes. Certain notifications that these concrete subjects will notify the observer include if a player has moved to a new spot, if an improvement has been added onto a square, or if a player declares bankruptcy and needs to be deleted from the game. This allows us to increase cohesion as the classes only have one responsibility (only the BoardView is responsible for printing), and decreases coupling as each class communicates with each other solely through the abstract observer and subjects, allowing us to reuse code if needed.

Update:
We ended up using the observer design pattern through MVC and it worked perfectly with our program. It allowed us to hide any logic or information about the game and it let us create changes to the board game display easily.

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?*

In the original Monopoly, Chance cards (which mimic SLC) are not only limited to movement actions and Community Chest cards (which mimic Needles Hall) are not only limited to money transactions. Instead, in each deck there are cards that will prompt a player to both move or collect/pay money. In this case, it would be best to use the template design pattern as we could define the common behavior of movement or paying/collecting money in the parent abstract class in which the child SLC and Needle Hall classes could override specifically to their deck. That way we define the steps of a movement or transaction, but let the subclasses redefine certain steps that are specific to their cards to avoid duplicate code. In our project, since we do not need to worry about having both decks be a mix or movement or transaction cards, we will just be implementing the actions in the SLC or Needles Hall card itself. This will also make it easier to work with the different probabilities.

Update:
We did not end up using the template design pattern in mimicking the SLC and Needles squares to be like chance and community chest cards. We just used the shuffle function and randomizer to select actions.

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

We do not think the decorator pattern is a good pattern to use when implementing improvements because it would require us to make a lot more classes than we need, and we would have a lot of hard code to define every improvement. This is because each improvement differs for every building - there is not one uniform improvement rule that we can apply to all properties. For example, going from 2 to 3 improvements for Arts1 is a difference of $60 in tuition ($30 -> $90), but going from 2 to 3 improvements for Arts2 is a difference of $200 in tuition ($100 -> $300). Every improvement associated with an ownable building will require us to hardcode our increases which would be stored more efficiently in the ownable building class itself. In addition, decorators are often used to be able to stack multiple features or layers of code on top of eachother recursively, but for improvements we know there will be at most 5 improvements in chronological order, so we do not need decorators to anticipate adding more than 5 improvements or applying improvements in a different order.

Update:
We did not end up using the decorator pattern for the same reasons we specified earlier. Instead, if we wanted to add an improvement, we simply incremented the improvement count on the academic building.

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?

We both learned alot about developing software in teams. One of the most importance lessons we learned was merging and editing code when there are multiple people working on it at the same time. We had to learn how to work on different parts of our game and to keep our code as loosely coupled as possible to avoid causing any merge errors. We also learned how big of an impact a well thought out UML diagram can have and will be defeinitely incorporating in the future when we both take on programming projects.

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

If we started over again, we probably would spend last time figuring out the fine details of our implementation such as our board layout and more on the high-level logic of our program. This way, this method could have given us a more solid foundation to work on rather than scrambling at the end and having our code be messy and unorganized.

Extra Credit Features:

We wish we could have had more time to implement more features, but the major ones we did was add more command input to simplify the game and using the chrono and thread libraries to make the program sleep for 1 second between actions. Using the sleep_for function we were able to make the game more playable and readable. It mimicked real life actions more.