Watopoly: DD2: Final Report

Joshua Demetrioff & Yingxuan Hu

Introduction

The goal of this project was to create Watopoly, a C++ video game inspired by the classic game Monopoly, that centers around the University of Waterloo campus. In the game, players can navigate the board, purchase and enhance on-campus properties, and pay tuition fees to avoid financial collapse. The project was carried out by a two-person team and included the creation of game mechanics, the implementation of player movements and the game board, the development of a user interface, and the testing and debugging of the code.

This report offers a comprehensive overview of the game's design and architecture, with a particular emphasis on critical features such as the game board, player movements, and user interface. Furthermore, the report delves into the challenges faced during development and outlines the solutions employed to address them. Finally, the report assesses the overall achievement of the project and identifies key insights for future software development projects.

Overview

To break down the structure of the project, the Watopoly game we created included classes of Subject, Observer, Player, Display, Location, Property, Regular Property, Residence, Gym, Game, and Controller.

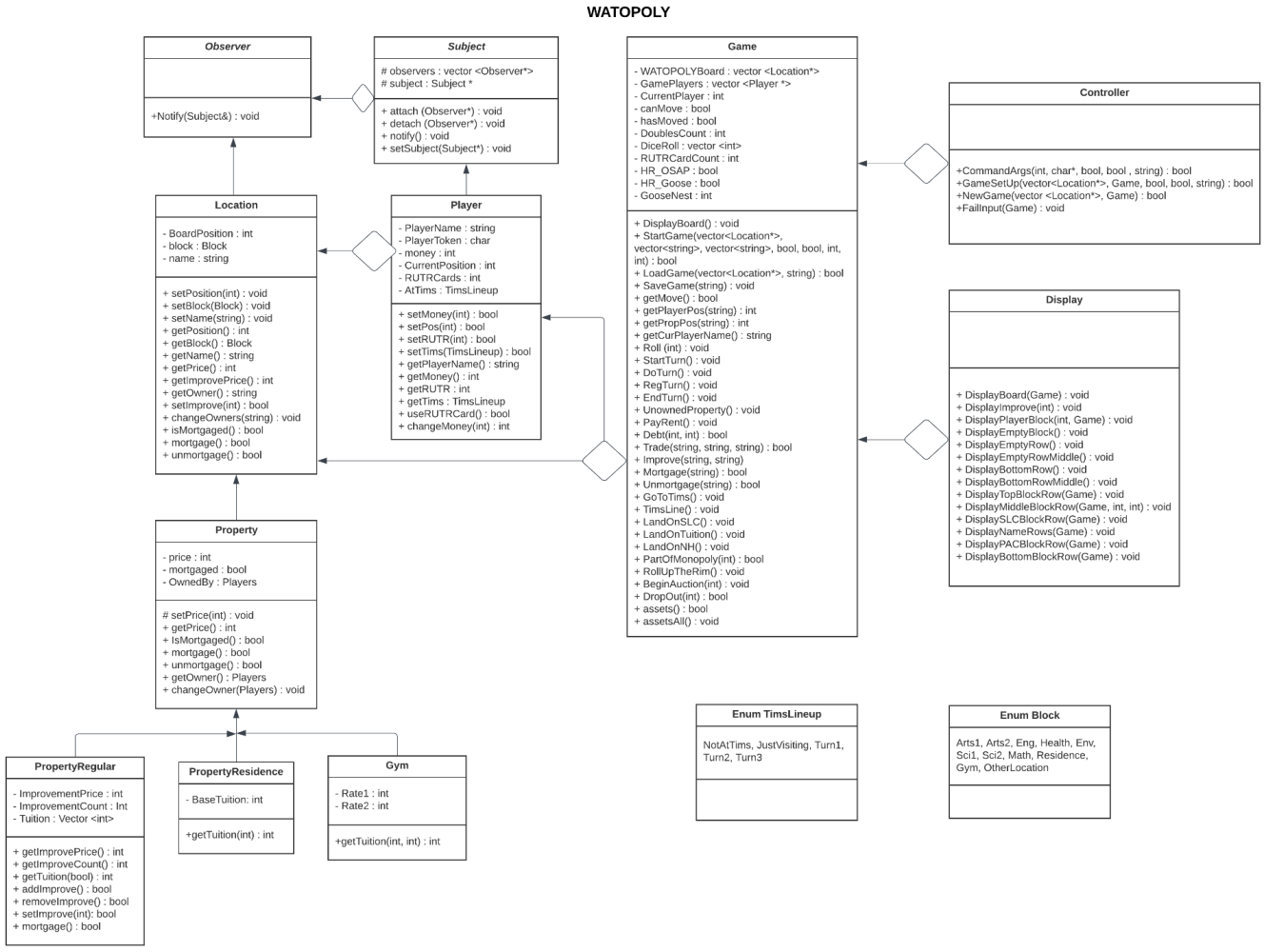
The Observer Pattern is used to allow different parts of the game to receive updates when something changes. The Player class represents the player in the game, and the Display class is used to display the game board and other game-related information.

The Location class represents each location on the game board, and the Property class is used to represent each property that can be bought and sold. There are also different types of properties, such as regular properties, residences, and gyms, each with their own set of rules and behaviors.

The Game class manages the overall state of the game, including keeping track of the players, the properties they own, and the current turn. The Controller class is responsible for handling user input and updating the game state accordingly.

Overall, the project follows a structured design pattern and separates the game logic into different classes to make the code more manageable and maintainable. The different classes work together to create an enjoyable gaming experience for the player.

Updated UML



Design

There were several design challenges in the Watopoly project, and different techniques were used to address them. Here are five specific techniques that we used:

1. Inheritance: Inheritance was used extensively throughout the project to allow different classes to share common properties and behaviors. For example, the Property, RegularProperty, Residence, and Gym classes all inherit from the Location class, which provides common properties such as name and position.
2. Polymorphism: Polymorphism was used to allow different classes to be treated as instances of a common superclass. For example, the Property, RegularProperty, Residence, and Gym classes are all instances of the Location superclass, which allows them to be treated uniformly in certain contexts.
3. Observer Pattern: The Observer pattern was used to allow the Player and Display classes to be notified when certain events occur in the game, such as a player buying a property or landing on a special square. This pattern allows the Player and Display classes to react to changes in the game state without tightly coupling them to the other classes in the game.
4. Interfaces: The Identifiable interface was used to ensure that all classes that implement it have a unique identifier. This interface allows different classes to be treated uniformly in certain contexts, such as when searching for a specific object by its identifier.
5. Abstract classes: Abstract classes were used to define common behaviors that are shared by multiple subclasses. For example, the Property and Player classes both implement the Payable abstract class, which defines getPrice() or getMoney() method. This allows the two classes to share common payment behaviors without duplicating code.

By using these techniques, the project was able to achieve low coupling, high cohesion, and flexibility to accommodate potential changes.

Resilience to Change

In the design of this project, we used a lot of inheritance and polymorphism to ensure the relationship between each class is clear and there is no replication between classes, so that we can compile the minimum code possible. Furthermore, our code achieved low coupling and max cohesion to accommodate potential changes through the use of the Observer Pattern. The Subject class is decoupled from the concrete Observer classes, allowing for new types of observers to be added without affecting the Subject class. Similarly, the Player and Display classes are decoupled from each other through the Observer Pattern, allowing changes to one class without affecting the other.

We tried to have high cohesion within the project by the use of inheritance and polymorphism in the Property hierarchy. Each type of property (RegularProperty, Residence, and Gym) has its own unique behavior, but they are all related to each other and can be treated as Properties. This allows for a more focused and cohesive class structure.

Also, the use of interfaces in the Watopoly design supports the possibility of changes by ensuring that all classes that implement a specific interface have a certain set of behaviors. For example, the Identifiable interface ensures that all classes that implement it have a unique identifier. This allows for easier integration of new classes into the program, as long as they implement the required interface.

The final design is very similar to our original one, except with a few more methods added to support more features that we did not consider.

Answers to Questions

Q1:

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

A:
In the context of Watopoly, the game board can be the subject, and the players can be the observers. Whenever the state of the game board changes, such as when a player lands on a new

location or when a property is bought or sold, the game board can notify all the players who are observing it.

The game can maintain a clear separation of concerns between the game board and the players. The game board can focus on managing the state of the game and notifying the players when changes occur, while the players can focus on making decisions based on the information they receive from the game board.

Moreover, the Observer Pattern provides an extensible way to add new features to the game in the future. For example, if the game were to be expanded to include new types of properties or events, new observers could easily be added to the game board to be notified about these changes

Q2:

SLC/Needles Hall: 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?

A:

Strategy design patterns would still be suitable in this situation. But in the project, we didn't choose to use it. Implementing the Strategy Pattern could add unnecessary complexity to the code. If the behaviors for SLC and Needles Hall can be adequately represented without the use of a separate class hierarchy, it may be more efficient to implement them directly within the Location class.

However, Using the Strategy Pattern would allow us to add new card behaviors easily and to modify the behavior of existing cards without affecting the player object or other parts of the game implementation. It also makes it easier to test each card's behavior in isolation.

Q3:

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

A:

In the Watopoly game, to implement the Decorator Pattern, an Improvement interface can be created, which defines the methods for adding, removing, and updating the improvements to a property. Concrete classes for each type of improvement, such as the properties for rent in Waterloo can then be implemented to provide the specific behavior for each improvement.

When a property is improved, a decorator object can be added to the property object, which adds the behavior of the specific improvement. This decorator object can be added or removed dynamically, allowing the player to add or remove improvements as needed. This approach also maintains the modularity and flexibility of the code, as each improvement can be added or removed dynamically without affecting the rest of the game logic.

Extra Credit Features

- House Rules:
    - Double OSAP Bonus: Enable an option where players who land directly on the "Collect OSAP" square (the Watopoly equivalent of "GO" in Monopoly) receive double the usual amount. This rule adds an element of chance and can provide a significant financial boost to players who land on the square.

    - Tax Refund from Goose Nesting Spot: Introduce an option where players who land on the "Goose Nesting Spot" square (a custom Watopoly square) receive a tax refund from the bank. This rule provides an incentive for landing on the specific square and can help players recover some lost funds. You may allow players to choose the amount refunded at the start of the game.

    - Customizable Goose Nest Fund: Allow players to decide how much money is placed in the Goose Nest at the beginning of the game. This option can be used to adjust the game's difficulty or to create interesting scenarios where the Goose Nest holds a significant amount of money, incentivizing players to land on the "Goose Nesting Spot" square.

- Customization:
    - Change Starting Money: Provide players with the ability to customize their starting money in the game. This feature can be used to create different game scenarios, balance gameplay between experienced and novice players, or adjust the game's difficulty. Players can choose a higher or lower starting amount depending on their desired challenge and gameplay style.

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?*

Answer:

The Watopoly project offered us a wealth of insight into developing software as part of a team. One of the most significant takeaways was the necessity of clear communication and collaboration among team members. This involved establishing unambiguous objectives, assigning tasks based on individual strengths, and checking in regularly. Conflict management was also a critical element of the project, requiring us to address disagreements proactively to ensure a favorable outcome.

By assigning tasks based on each team member's strengths and comfort level, we were able to increase overall efficiency. Additionally, I realized that good program design can significantly reduce coding time by helping us identify the relationship between classes, prioritize essential

methods and features for each class, and minimize the need for unnecessary coding and compiling.

To summarize this, the Watopoly project granted us valuable experience in developing software as part of a team and writing extensive programs. It taught us valuable lessons about communication, collaboration, conflict resolution, and receiving feedback, all of which will serve us well in future software development endeavors.

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

Answer:

The team could have focused more on improving communication and collaboration. This could be achieved by setting up regular check-ins or using collaboration tools to ensure that all team members are on the same page. By establishing a routine for communication, team members can discuss project progress and address any issues or concerns that may arise in a timely manner.

Additionally, we could consider exploring ways to streamline processes, such as automating certain tasks or dividing up responsibilities based on our strengths. By doing so, the team can optimize their workflow and ensure that each member is contributing in a way that maximizes their skills and expertise.

It’d be better to set clearer expectations from the outset of the project. This can help prevent misunderstandings or confusion later on, as we will have a shared understanding of the project's goals, timelines, and deliverables. By setting clear expectations, we can work towards the same objectives and avoid any misalignment or disagreements that may arise otherwise.

Finally, the paragraph recommends soliciting feedback from each other throughout the project. This can be done through regular check-ins or anonymous surveys to gather feedback from team member. The good thing about this is that the team can identify areas for improvement and make changes as needed to ensure a more successful outcome.

Conclusion

Throughout this project, we acquired knowledge about project design, the implementation of design patterns, and the utilization of inheritance and polymorphism in our code. Furthermore, we developed collaborative skills while working alongside our teammate. We believe that we can perform even better in future group coding projects by integrating these newfound lessons.