Introduction

Our group chose to implement Watopoly, a University of Waterloo twist on monopoly. We designed it so that it is playable with up to eight players, with a strong focus on trying to bring the board game experience to a digital application. By using the principles of Object-Oriented Programming (OOP) taught throughout the course, we used key concepts like encapsulation, inheritance, polymorphism and design patterns to simplify the more challenging features. Throughout the development process, our group collaborated closely, and communicated effectively to create a working implementation of Watopoly.

Overview

The overall structure of our Watopoly program is organized around the necessary components needed to facilitate proper gameplay with a suitable interface. Each of our classes corresponds to a key component of the game, with additional classes designed to support the functionalities of these key classes.

The Player class represents the participants in the game. This class encapsulates the essential player information, such as: name, cash, player piece, and position on the board. With an implementation designed for an aggregation relationship with Bank and Board, the Player class is able to seamlessly perform actions the expected actions, like: buying properties (and modifying them if the option is available), paying fees, moving around the board, rolling the dice, and even interacting with other players through trading. The Player class was implemented as a subject (using the Observer Design Pattern) to better facilitate interactions between the Player and the Board. We added additional commands that the player can enter, including buy and pay to make the experience more closely related to the board game.

The Board class acts as the backbone of the game, and provides a structural representation of the physical board. Implemented by encapsulating all the tiles, the players, the bank, and a text display, the Board is able to control the flow of the game and is responsible for interpreting and executing the actions the player wants to do. The board keeps track of the game state making sure that each component functions as expected. The Board is implemented as an observer (using the Observer Design Pattern) so it can properly observe the Player and be notified where relevant.

The TextDisplay class is a component of the Board that is responsible for maintaining the graphical representation of the Board. By reading in a text file, that is the Watopoly game board's structure, the TextDisplay is able to properly display and update this display to ensure that the players have a clear visual throughout the game. The TextDisplay is implemented as an observer (using the Observer Design Pattern) to the Player and Tile classes so that it can accurately update

the display whenever a player moves around the board, or when an improvement is added to an academic building.

The Bank class functions as the central repository for ownable property information, financial transactions, and property management (transferring properties from bank to player or player to player). The Bank serves as a component of the Board class, as it solely exists in a 1-1 ratio with the Board. The Bank's key responsibility is to manage property ownership, property transactions, and handle monetary exchanges between players and the bank, or players with other players. Storing the property information within the Bank simplifies the property management process, which in turn simplifies the Player class.

The PropertyConfig class is a component of the Bank which holds the relevant information for each ownable property, to aid in streamlining property management and transactions. The PropertyConfig is initialized by reading in a CSV file that stores the essential values for each ownable property in the order: BuildingType, BuildingName, MonopolyBlock, PurchaseCost, ImprovementCost, DefaultFee, Imp1, Imp2, Imp3, Imp4, Imp5. For residences and gyms the monopoly block is set to their type, and the improvement values represent the new fee based on how many residences or gyms the player owns.

The Tile class represents each of the game squares that make up the board. This class is further divided into subclasses to differentiate between the ownable and non-ownable properties. This was further designed to use polymorphism, where each of the subclasses of OwnableProperty and NonOwnableProperty get treated as if they were the Tile class, allowing us to effectively store all Tiles in one vector while still being able to call the desired method for each one. Tile was further implemented to be a subject to the TextDisplay, so that whenever an academic building added an improvement, the TextDisplay would be notified and the display would update accordingly.

A further breakdown of the NonOwnableProperty and OwnableProperty class consist of the subclass SLC, NH, DC Tims Line, Goose Nesting, Tuition, CoopFee, GoToTims and CollectOsap to represent the unique non-ownable tiles on the board, and the Residence, AcademicBuilding, and Gym classes to represent the distinct types of ownable properties. Through the use of inheritance and polymorphism, the implementation and use of these child classes allowed us to solve many of the more challenging design aspects when implementing Watopoly.

Throughout the planning and implementation process, careful consideration towards the class relationships of classes and the different hierarchies was maintained to ensure our program functions accordingly.

Design

For our design, we used different object-oriented techniques to address the various design challenges when developing Watopoly. Our solutions revolved around the use of the Inheritance and Polymorphism principles, as well as the utilization of the Observer Design Pattern.

Inheritance was useful for breaking down each of the components and categorizing them through different subclasses. For the Tile class, we had two subclasses: NonOwnableProperty and OwnableProperty. NonOwnableProperty represents the properties that cannot be owned, such as SLC, NH, DC Tims Line, Goose Nesting, Tuition, Coop Fee, Go To Tims and Collect Osap. NonOwnableProperty is an abstract class used to define the non-ownables, with each of the above-mentioned tiles being its own subclass. OwnableProperty represents all the properties that the player can purchase from the bank and collect fees from other players who land on it. Furthermore, OwnableProperty is broken down further into three more subclasses: Residences, Gyms, and AcademicBuildings, each representing the different types of ownable properties, allowing for the encapsulation of the unique functionalities while maintaining a cohesive class hierarchy.

Polymorphism enabled flexible and extensible interactions between the interaction of objects of different classes. By using virtual methods, like performAction(...) in the Tile class, we ensured that each child class would have an implementation of its own unique behavior. Using this, we were able to store the diverse tiles in a single vector, and then leveraged off the compiler to invoke the appropriate polymorphic method.

The Observer Design Pattern was used to link the Board and TextDisplay (observers) to the Player and Tile (subjects). The Board observed the player's movement, so that whenever a player moved the Board would get notified, causing it to activate the action corresponding to the Tile that the player had just moved to. The TextDisplay observed both the Player and the Tile. It observed Player so it could update the outputted display board alongside the player so the player's position would be accurately displayed on the board. It observed Tile so that it could update the AcademicBuilding tiles display to accurately show the number of improvements it has whenever an improvement was added or sold. This facilitated loose coupling and allowed for effective and efficient communication between the different components within our system.

We chose to use these OOP techniques to simplify the design challenges while also giving our code more maintainability and flexibility, which are essential, especially if we wanted to add enhancements to our game.

Resilience to Change

Our program was designed to be very adaptable and flexible, to accommodate any enhancements and/or changes that we may have wanted to implement. Our intention when designing our code was to make it so that it has low coupling and high cohesion.

To achieve low coupling we minimized the dependencies between modules and components. For example, the Bank module communicates with the Player module through a well-defined interface for the different transactions, creating seamless interactions while keeping the dependencies to a minimum. This design choice to minimize dependencies allows our code to maintain flexibility and allows for easy incorporation of new features or modifications.

To achieve high cohesion, we ensured that each module encapsulates a single, well-defined responsibility. For example, the TextDisplay is only responsible for the graphical display, the Bank is only responsible for property or monetary transactions, the Board is only responsible for maintaining the game state, and the Player is only responsible for maintaining relevant fields, and notifying its observers when necessary. This design helps promote clarity and simplifies the codebase. This also allows the code to be more readily and easily modifiable.

This combination of low coupling and high cohesion provides a solid foundation, making our code adaptable, flexible, and resilient. We are able to boast seamless integration of new functionalities without causing issues to arise in different parts of our program.

Answers to Questions

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

As stated in the plan.pdf, the Observer pattern is a good fit when implementing a game-board because it is able to manage the complexity of notification, providing a scalable design that is flexible to alterations depending on the changes to the requirements.

We used the Observer design pattern in our implementation, with the Player and Tile classes being subjects and the Board and TextDisplay being observers. This setup allows the classes responsible for maintaining the game state and the visual display to be notified when the player is moved or when an improvement is added to an academic building. This seamless combination simplified the complexity and reduced the reliance of one component over the other.

In our original UML and plan we mentioned how the Bank would also be an observer and the Tile would not be a subject. This was changed as the Bank was designed with the sole responsibility of property and monetary transactions, and so it was redundant to make it an observer, as there was nothing for the player to notify the Bank for. We changed Tile to be an observer, as we realized that the visual display also showed the improvements for academic buildings. To solve this, we made the Tile a subject and attached a TextDisplay to observe it and update accordingly. We didn't make the AcademicBuilding a subject instead, since we wanted our code to be adaptable incase we wanted other tiles (not just AcademicBuildings) to have the potential to be subjects if we wanted relevant enhancements.

For example, if we implemented the free parking rule (where the fees that are paid to the Bank get pooled together and whatever player lands on it gets that money) we may then wanted to attach the TextDisplay to NH, CoopFee, Tuition, and GooseNesting so that we could display the amount of money in the current pool (would get updated whenever a player pays a fee to the bank or wins the pool, which occurs on these tiles), so the Player has a sense of how much the could win if they land on it.

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?

As stated in our plan.pdf, a suitable design pattern for modeling the Chance and Community Chest cards is the Strategy Design Pattern. This Design Pattern could simulate a respective deck of cards for the Chance and Community chest piles, while keeping the code relatively simple.

While we did not implement this Design Pattern in our code (as we could achieve the current requirements for Chance (NH) and Community Chest (SLC) in a much simpler way) it was tempting to add it as an enhancement to truly simulate the feel that the player was taking a card from the pile. It would also give us the opportunity to add more unique cards with descriptions that would further match the two card types in Monopoly. Unfortunately, our group did not have time to add enhancements such as this, but if we had more time this would have been near the top of the list for enhancements to be added.

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

As stated in the plan.pdf, although the decorator pattern is not necessary for basic improvements, like adding bathrooms or cafeterias, as outlined in the project specifications, the Decorator Pattern offers significant advantages when considering resilience to change and potential for future enhancements.

While we originally had it set up in our initial UML and plan.pdf that we were going to use the Decorator Pattern to implement the improvements, we ultimately decided against it. This was because implementing it offered little for what we wanted to do with our game. We could not think of any relevant enhancements that we could implement through this pattern that would actually enhance the experience of our game.

Furthermore, the process of implementing it was not as simple as we expected. It also, in a way, conflicted with the already existing PropertyConfig, which was a much easier way to implement the improvements. While the adaptability using the Decorator Pattern for the improvements was a desirable feature, we ultimately decided that it was not relevant to how we approached the implementation, and was thus not needed.

Extra Credit Features

For extra credit, we used only smart pointers and no raw pointers in our implementation. Initially, transitioning from the standard raw pointers to smart pointers introduced some difficulties (like when trying to dereference a weak_ptr, only to find out you can't and you need to use lock() on it). While they were different, it didn't take long for us to appropriately integrate them throughout our code. Though they were trickier to use than raw pointers, they did simplify many things in that we did not have to track where memory was allocated and make sure that it was properly freed.

For enhancements to our Watopoly game, we sadly didn't have much time and were not able to implement most of what we had planned. The few things we wanted to add, but did not get the chance to, was to add the 'Free Parking' rule where money paid to the bank through fees is pooled in the middle and whoever lands on Free Parking (Goose Nesting in Watopoly) claims the pot. This was a very simple implementation, as we designed our code to be adaptable. We already have all the necessary elements needed to add this without really affecting the rest of the code. Another thing would be to get a working GUI interface. We briefly looked into using the <gtkmm> library to get a more visually appealing display, but again we ran out of time.

Final Questions

1. What lessons did this project teach you about developing software in teams?

Working in a team to develop a software was a first for many of us and provided us with a unique and valuable experience. This allowed us to gain more experience with version control software like the university's GitLab as well as giving each group member the opportunity to develop their communication skills and share a sense of responsibility.

Being able to become more comfortable using version control softwares like GitLab is an essential skill for all developers. As it is useful for both independent work and for collaboration, so developing these relevant skills will definitely be passed to to future academic terms, or in the workplace.

When working by yourself, you feel that you can procrastinate a lot more since you are only responsible for your own work, but when working in a group you are also responsible for your peers' work as well. This forced us to take more initiative in getting things done earlier so as to not be the dead weight and be responsible for achieving a bad grade.

Working in a group also provides ample opportunity to communicate with your group members to tackle problems, bounce around ideas, or even just socialize. Being able to communicate with others' makes solving problems easier and gives you the opportunity to express and put forth your own ideas. It also serves as an error checker, since nobody's perfect you'll eventually make a mistake, but that's what group members are for, to help sort out the issues or catch errors before they become one.

All in all, working in a group was a great experience and provided us with many opportunities to grow skills which will most definitely be useful in feature terms and in the workplace.

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

We were ambitious and had a decently well drawn-up plan so we thought we would be able to swiftly implement Watopoly and have plenty of time to add enhancements, but that was far off. Our approach ended with us compiling only once all our classes had been fully implemented which caused much pain and suffering when trying to fix all the compiler errors and catch other bugs within our code.

If we started over, the first I would implement is the Board::playGame() method. This was the backbone of our program, the loop that would take in commands from the player and output what was expected (in our approach we ended up doing this method near the end). Once we had a rough implementation (just an if-else chain that would accept commands, not necessarily do anything) we would gain insight into how we should approach each command,

and what should be expected. This would also allow us to go through the process of implementing one command at a time, making sure our program compiles, and testing the command. If all these check-out we can move onto the next command. This allows for a systematic approach that would've helped us avoid bugs and compiler errors, as well as thoroughly test our code before moving on. It would also aid in avoiding junk code that may have seemed useful at one point only to be useless when we implemented this method (which did happen to us). While we did not do this, we were still able to create a functioning game of Watopoly, but we did have to struggle through many bugs and the pain of compiling only after everything was already implemented.

Conclusion

To conclude, this project was a rewarding and insightful experience, fitting for the final project of CS246. It allowed us to apply the various OOP principles we were taught throughout the course to design and implement Watopoly. Through a collaborative effort, we overcame challenges and developed valuable skills for software development practices.