Introduction

Watopoly is a project designed to mimic the popular board game, Monopoly. Our design of Watopoly involves a text display that is capable of game loading or saving from or to a text file, and starting a new instance of a game. Following the rules outlined by the CS 246 Winter 2023 Project — Watopoly document (the spec), anywhere between 2-6 players may play our version of Watopoly.

While our implementation does not encompass the requirements for a fully-functioning version of Watopoly, our intended design would be capable of supporting such an implementation. Aiming for high cohesion and low coupling throughout the design process, the modularity of our program is high, thanks to the polymorphic design philosophy adopted throughout the design and implementation processes.

A few design details needed to be modified as issues were revealed throughout the implementation of our program. These will be discussed in further detail, but did not significantly impact the design overall, primarily working to introduce more getters and setters and reducing a reliance on protected fields, friend classes, and highly-coupled association between classes.

Overview

The Player class of our implementation manages properties and actions inherent to a single player. This includes, but is not limited to, rolling the dice, tracking whether or not the player is in jail, their existing balance, the number of Tim's cups they hold, etc. Most notably, functions involving multiple players are not handled within the Player class. This was intentional, as interactions between players in the traditional game of Monopoly take place over the board, and thus belong in the Board class, discussed later in this section.

Each square on the board is represented by an individual instance of the Square class. The Square class is an abstract class that is divided into 2 further abstract child classes: Non-properties and properties. Each of these child classes then have concrete child classes that specifically implement what function they are outlined to do as per the spec. For example, a concrete implementation of Needles Hall is included as a child of the Non-property abstract class to determine exactly how much the balance of the affected player would be modified by. Likewise, a concrete instance of a property contains information such as the number of improvements the property has, who owns the property, whether it's mortgaged, as well as keeping track of exactly how much is owed to the owner should a non-owning player land on an owned property. Specific design details are discussed further in the Design section.

A Textdisplay class is included, following the example text display provided by the spec. This means that our display only supports up to 6 unique players, contrary to the traditional 8 supported by Monopoly. This meets the spec's requirements, and the text display could easily be made taller or wider to support more players, though this may present a less visually appealing experience for the user.

The Board class is responsible for tying together all the other aforementioned classes. The Board calls on the text display to display the information contained within it through a subject-observer relationship. However, given the existence of only one subject and one observer, we elected not to pursue the Observer design pattern, as abstract classes would ultimately only have 1 concrete class each. The Board tracks and handles information between players, such as the locations of every player, handling trades between players, or executing the sequence of events following a player declaring bankruptcy. Only one instance of the Board exists per execution of the program.

Design

The polymorphic design in dividing each square first into abstract Non-properties and Properties, then forming concrete classes to represent every individual square supports a high level of modularity. Should a specific detail of a single square need to be modified, such as changing the number of spaces SLC moves a player, or how much rent is owed given 3 bathrooms in MC, the rest of the program does not need to be recompiled to support the change. This also means that if need be, a property can easily be changed to a non-property and vice versa, so long as the board is also modified to support and track the change. This design philosophy promotes a high level of cohesion, with each individual concrete property being divided by Monopoly Block (e.g. Arts1, Arts2, Eng, etc.), and each individual non-property having its own concrete class. This way, all fields and functions pertain solely to the responsibility of the specific class. Furthermore, this discourages coupling between squares, as no square's function should depend on the fields or functions of others, and appropriate interactions involving a player and movement are handled within the Board.

Our decision to avoid implementing the Observer design pattern was made early on. It was clear that contrary to another traditional grid-based game such as tic-tac-toe or reversi, the status of each of our squares has little to do with one another. There is no need to retrieve the state of another square within each square, and there is no need to notify other squares of a change of state when or if that occurs. Hence, the only advantage to establishing an Observer pattern would be if there were multiple display types involved, which we did not have time to implement. The board does notify the text display of any state change, but that is the extent of any instance of the Observer design pattern within our program.

The Board class is the most tightly coupled class in the program. Any change to any of the other classes must cause the Board to recompile to accommodate the new changes. However, this is ultimately necessary, as the Board, much like in the actual game of Monopoly, ties everything together. It brings each Player to interact with one another, the pieces move on the Board, and improvements are placed on the Board. This tight coupling allows coupling to be reduced in the rest of the program between the other classes. It also does not sacrifice the cohesion of the Board class. Given that it is the epicenter of all player-to-player interactions, movements, and information relevant to player-to-player transactions, it follows a general theme, and contains fields and functions that work closely with one another to achieve the ultimate purpose of running the game for the remaining players.

Resilience to Change

As mentioned earlier, the Template Method Pattern was implemented to represent property and non-property squares. This allows for squares to efficiently be modified and have the changes reflected in the implementation of the board. Every field of an individual property or non-property can easily be modified without impacting the other squares. Given that the board is also modified accordingly, other changes can also easily be implemented, such as implementing House rules, Chance and Community Chest cards, allowing for more than 4 Bathrooms or 1 Cafeteria, implementing even construction, etc. The size of the board can also be changed without heavily impacting the rest of the program.

Where our resilience to change falls short lies in our display of the board. Given that an observer pattern was not adopted, the board is designed only to be displayed by a text display in our implementation. Additionally, given our existing text display, more than 6 players cannot be supported, as they would not fit on a single square without appearing visually displeasing or unclear.

Answers to 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 do not believe the Observer Pattern is necessary to implement a game-board, but it could be a nice-to-have. For example, if a Player acts as an observer to the Board subject, upon notification, the Player may know that it is their turn, and in turn base their actions on the state of the Board. Additionally, text, graphical and any additional displays are certainly beneficiaries from having a generalized Observer Pattern, as only minor details will need to be modified under the implementation of each display to fit the standard implementation of the Board within the Board class (such as a vector of every Square).

Extending on the above, we have now confirmed that an Observer Pattern was not beneficial beyond implementing additional displays, and have elected not to include it beyond a concrete relationship between the Board and Textdisplay classes as it is ultimately unnecessary to involve more than one or two displays for our version of the game.

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?

The Template Method Pattern feels the most appropriate for the implementation of Chance and Community Chest cards. There are some properties inherent to every card, such as whether or not they can be possessed by a player (such as a get out of jail free/roll up the rim cup) which can be implemented in the abstract Card superclass. Then, we must differentiate between a Chance and CommunityChest card so queue ADTs can be set up to represent the "card stacks" on a real board, creating 2 further abstract subclasses of the Card superclass. Finally, every individual card will serve a different purpose (such as moving the player, sending them straight to jail — so no OSAP bonus, adding or removing money from their balance, etc.), with each card being a concrete subclass of either Chance or CommunityChest abstract classes.

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

No, the Decorator Pattern would not be good to use when implementing Improvements. A number of issues arise. For one, every Bathroom is its own individual decoration, meaning there is no way to determine from the base Property exactly how many Bathrooms are currently constructed. Without this information, a Cafeteria could never be constructed, as we could continue to construct beyond 4 Bathrooms, or we could even construct a Cafeteria before constructing a single Bathroom! In addition, a Cafeteria is a completely different decoration to the Bathroom. Should we somehow circumvent the hurdle of tracking the number of Improvements, simply "overlaying" a Cafeteria atop the existing Bathrooms does not remove/destroy the underlying Bathrooms. Any implications of having previous Improvements, such as the tuition increase they cause, do not disappear. So the tuition charges may overlap (e.g. at 3 Bathrooms, a player is instead charged the equivalent of 3 Bathrooms + 2 Bathrooms + 1 Bathroom + base tuition; 3 Bathroom tuition → 2 Bathroom tuition → 1 Bathroom tuition → base tuition). Hence, we do not believe the decorator pattern is appropriate for implementing Improvements.

Extending on the above, a number of other complications arise when attempting to implement the decorator pattern, which was quickly abandoned. Most significantly, without increasing coupling significantly between improvements and individual concrete property squares, there is no way to prevent an interaction that involves constructing an improvement on a non-property, or constructing a bathroom on top of a cafeteria. Such complications cause the Decorator Pattern to lose its value as an efficient solution to overlaying modifications on a base object.

Extra Credit Features

Unfortunately, given the time constraints, no extra credit features were attempted for this implementation of Watopoly. It can be noted, however, that with the exception of House rules, our polymorphic design does support an easy implementation of any of the suggested extra credit features in the spec given more time.

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?

Developing software in teams seems to be better when working together in person. When you can discuss ideas and chart them on paper prior to actually implementing the code, the actual implementation runs more smoothly. Dividing tasks is a great idea, but runs into a few small issues. Oftentimes, one part of the program depends heavily on another, but if that part of the code has not yet been complete, the entire project may grind to a halt. Furthermore, when a specific part of the program has been completed, it is very helpful to keep your teammates in the loop about what exactly you changed, and why you chose to change it. In fact, it was often better to announce what change an individual thought would be necessary before making the change, so it could be discussed and either accepted or abandoned so that the team remained on the same page.

Overall, working in a team outweighs working alone, especially on large programs, as subtasks can more easily be divided and individual frustrations can be avoided. However, to maximize the benefits of having multiple individuals working towards the same goal, it is imperative that everyone remains communicative and on the same page throughout the entire development process.

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

We would have liked to start earlier. Despite getting off to a strong start, the reality of the vast number of small but significant details settled in a little too late in the process. We quickly strayed from the plan of attack, and struggled to return to schedule. Kinks in the implementation process caused us to focus on prioritizing the core functionality of the game over the additional details that would have completed it. While we are proud of what we were able to achieve, the game remains incomplete due to poor time management as well as other factors.

One such other factor was a poor understanding of the spec and the assignment. Minor details and features of the Watopoly game continued to plague the implementation process, such as random number generation, the purpose of testing mode, and in what priority tasks should have been handled. A text display was the original priority, but instead of shifting toward establishing a Player class, it may have been more beneficial in hindsight to work on the board as soon as possible, so that a relationship between the Board and Textdisplay classes could be hammered out early on to help design and utilize a testing mode. Instead, the first compilation attempt was made relatively late on in the development process, and that eventually led to producing an incomplete project.

Conclusion

In conclusion, we have produced a basically functional version of Watopoly. While lacking a few non-property interactions, we do feature proper property ownership, tuition payment, player trading, auctioning, file loading, saving, text display, and handling of user commands and inputs. This project has the potential to become a completed version of Watopoly given more time, but nevertheless achieves the core goals outlined within the spec.