Design Document: Watopoly

Hans Wang and Thomas Liu

April 5th, 2024

Introduction

The game Watopoly is a variant of Monopoly, with the board based on the University of Waterloo campus. Game rules are based upon the project specification while additional edge cases take inspiration from traditional monopoly rules. Specific areas of our game that differ from traditional Monopoly are the upgrade mechanics and the fees associated with owning a full set of cells in a given block group.

Overview

The overall structure of the project is as follows:

- main.cc: takes in all the users' commands, checks whether it's a valid command. If it is a valid command, calls the methods that are defined in board.cc
- board.cc: where the game board is stored. The game board is made by an array of cells, which is defined by cell.cc. Major functions include move (move player), buy (buy a property, etc. as defined by the assignment guideline). These functions control the ownership of the building, balance of each player, position of the players, etc. Whenever the player position gets updated, it will notify both the textDisplay and graphicsDisplay
- player.cc: where the player's information is stored. The information includes the player's name, balance, the number of times he gets sent to DC Tims, etc.
- cell.cc: used to represent each block of the game board. It also stores which players are currently on the cell. Most of the functions are virtual as it is the base class for the different types of buildings, gyms, residences, and events the players will interact with.
- property.cc, feeBuilding.cc, event.cc: all subclasses of cell. They each represent a different block on the game board. Overwrites the functions in cell.cc to perform different features provided by different classes. Key functionalities include resolving tuition fees (given upgrades), and displaying any messages associated with a cell.
- textDisplay.cc and graphicsDisplay.cc: These are the output functions where text display overloads the output operator to print a board's state, cells, and players while the graphics display (bonus component) works with window.h and window.cc to draw cells to the X11 window. Both of these displays utilize the observer design pattern to notify on board events.

Design

The final **design of our project** has a certain amount of changes compared to our initial UML design due on due date 1. The two major changes are not using a decorator for fee buildings' improvements, and the relationship between each individual class. We realized that the decorator design pattern is not necessary to the improvements, as there are 6 constant levels of improvements. It would be much easier if we use an index to track the level of improvements, and an array of length 6 to store the fee for each improvement. For classes like FeeBuilding, Properties, etc. they are all subclasses of cell, which we can have a lot of virtual functions defined. Also, we had to increase the number of getters and setters for the private and protected values of all our classes which was expected as we filled in additional data being tracked through each class. In the process of designing our final implementation, we started with the base classes and classes that hold ownership of other classes, and expanded outwards. This allowed us to complete features independent of each other. One key change in our design is the usage of **unique pointers**. This gave us the opportunity to both implement a bonus mark improvement and eliminate the need for memory management.

The key design pattern we implemented was the **observer** design pattern which is used to update the position of players, charging tuition, and interact with events. By implementing an observer design pattern, we could naturally define the piece interactions between event, tuition, and the special events (such as entering/leaving the DC Tims line). The observer design pattern was especially useful given the consistency of operations where the board consisting of unique pointers to cells and the text display that holds a vector of dictionaries both function the same way. Thus, pure virtual functions are used which share a large number of similarities between updating cells in a board, and cells in the display. By implementing an observer design pattern, we save a large amount of time and have a direct positive influence on our design process.

For our design, we **reduced coupling** between each class through associating the necessary methods for the commands into each class. For example, when we move players, the move functionality is called by the board, but the change in player position is managed by the player class itself. Similarly, the trading, buying, and improving functionalities are all called and validated in the board class but only depend on player or property methods for giving, receiving and transfer of ownership. The general structure of our code is that the board has a large number of operations associated with it but the player, buildings and events all manage their own methods within the respective class. Each class only has operations that are logical for itself and associated with the data fields of the class.

In addition, we utilized the theory of **high cohesion** of classes primarily through our UML diagram. This involves assigning class attributes first, then having the cohesive methods which are directly related to the data stored in the class. By following this design principle, it was very clear where functionalities like setting owner, receiving/giving money, and moving pieces are going to be implemented. By thinking about cohesion early on, it helped us through speeding up implementation and avoiding re-implementation of existing methods.

Another C++ concept we benefited from are **virtual/pure virtual functions**. By associating the various functionality of cells, we were able to design our program in a way where all cells are derived classes of a cell base class. This gave us the chance for consolidating the functionalities as pure virtual methods so each cell the player lands on interacts across the same functionalities but can be unique given the cell type. By having the derived classes, we were able to take inspiration between the derived classes and speed up our development process. By having these pure virtual methods in the base class, we were able to once again design our program to be highly cohesive.

Resilience to Change

For our design, we have very low coupling, where each class serves their own purpose. Therefore, changes in specifications means we can swap out class definitions that include additional methods. Each class handles a specific set of tasks associated with it and can operate on its own. In addition, our design operates with base classes for all the Watopoly cells, by designing on top of a base class, future changes can be made to define a different variation of the cell easily. It would just be another child class of cells and as long as it has the implementation for pure virtual methods of the base, the rent, fees, and events would be operational in gameplay with minimal changes.

One area that is not as resilient to change is the board's print functionality, where each cell would be set in place and if the board were to be resized, there would have to be changes in the output overload. It would be challenging working with a board that resizes itself given the number of cells in a Watopoly game.

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?

The observer design pattern is the optimal solution when implementing a game board because the different components would normally require large amounts of code to be grouped together into a notify function whenever a piece were to interact with cells of the game board. By having the cells as observers to the player piece subjects, it is quite natural to write the necessary notifying method. This would make it very clear which cells are updated and is particularly useful for our text display which can track the cell content changing.

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?

SLC and NH can be modeled more closely to Chance and Community Chest cards as they do very similar stuff to the players that landed at the cell. They both provide functionalities including give/charge money from players, send them to another cell, send to jail, jail pass, etc. We could use a **decorator** design pattern. With the use of the decorator design pattern, we could dynamically modify the number of cards in the pool, but also didn't make any change to their functionalities. This will allow us to have more flexibility modifying the events happening to the user.

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

Decorator would be the go-to design pattern for implementing the improvements. However, we chose not to utilize this design pattern because it was faster for the building classes to initialize with a stored improvement array which is very direct. By tracking the number of improvements and an array of associated fees (there is a constant 6 level of improvements), we were able to access the necessary fee with limited work. This shortened the amount of time spent coding drastically and allowed us to explore other components (specifically enhancements and smart pointers).

Extra Credit Features

For our final implementation, we had the chance to eliminate tedious memory management which includes implementing members as either vectors or unique pointers. Thus, we did not have any "delete" statements as they can automatically allocate the memory. A second improvement we made was to increase the detail in the text display, where the owner of each building is shown along with the upgrades. The final extra credit feature was a X11 display that prints out the cell, images for upgrades, and player pieces as figures. This enhances the gameplay and allows for a clearer representation of the board. An addition on top of basic X11 is the design and usage of game pieces which are pixel arts that represent each playable piece (instead of using something like drawString). This involved designing and coding the necessary functions to draw these figures consistently over a particular pixel space of 10x10 pixels. Another addition that increases the player experience is color coding of squares, such that each faculty block type has a separate color code that increases the readability of the board. A third enhancement on top of X11 is that every enhancement is drawn up as either a toilet or a cup of coffee(depending on the number of improvements). The fourth enhancement is adding extra command line functionalities to test purely command line outputs. Specifically, we added the ability for games to be run if a user chooses not to have the X11 display operational ,where adding the --no-display flag will only run the command line component. This allows for efficient testing and gives players the opportunity to choose if they would like a display.

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?
By collaborating on the Watopoly project, we gained knowledge in a variety of C++ concepts across design patterns and data structures. Specifically in a team setting, we had to collaborate on GitHub which was briefly introduced in CS136L. During our project, we got the chance to

practice GitHub fundamentals including creating branches, caches, rebases, and pull requests. This is also the source of various challenges. Notably, at the start of our project, the source code was overridden and completely wiped out from a pull request around 7 days before the deadline which was frustrating. From these experiences, we became more proficient in managing versions of our code, resolving merge conflicts in our IDE, and communicating between ourselves to review PRs together. In addition, we started off the project tackling specific components by ourselves which meant we each had a feature list, a series of deadline goals, and worked on each component individually. This was very efficient in terms of outputting code, but lacks testability when we implement a new feature. Eventually we shifted to working in complementary areas where one member would start a section that the other team member needs and utilized the main function as a checklist of command line and input options to implement. This change allowed us to speed up testing and mash it into the process of writing new code and is one of the primary reasons we were able to finish our project ahead of time.

In the process of working as a team, something we found helpful was communicating face to face in a group study area. This keeps everyone up to date and avoids miscommunication especially during the game flow portion of the project. In addition, we started pushing code more frequently and pinging the other group members even when they do not need to review code. This prevented version control issues such as merge conflicts and served as a notice for members to pull the master branch's updates. When resolving challenges, we would consult with each other and understand the existing implementation, then follow up with a proposed solution and dividing tasks. By building on top of what we had implemented, the process of adding features felt much more natural and increased our productivity.

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

If we had the chance to start over with the project, the most important change would be to start our project earlier. In particular, we only spent five full days developing the project but would benefit from starting earlier and taking on more risks. Specifically, if we started earlier, we could explore more design patterns, and shortened our code. Another key area would be setting up a GitHub environment ahead of time. This would avoid any confusion or loss of code when we try to pull/push extensive changes together. For example, half of our code was wiped out five days before the deadline because we merged branches which deleted certain untracked changes. For each pull request we set up, it was beneficial to reduce the amount of changes. Even when we broke down the problems into small pieces, because we did not have a lot of time, we often squashed entire functionalities into a pull request. Some examples include notifying, buying, trading, selling, improving/unimproving. However, we did not manage separate branches for the different methods these core commands rely on. This directly led to bugs that took longer than needed to resolve not because we cannot find the source of the issue but rather which set of files will be affected. If we start over and increase the frequency of pull requests with targeted goals, we would improve our efficiency because we can quickly identify the changes made which caused the unexpected behavior.

Conclusion

From this project, we revisited various design patterns and object oriented program concepts we learned throughout the course. These include observer and decorator design patterns, UML, smart pointer, virtual function, polymorphism, etc. We deepened our understanding and proficiency in object-oriented design and programming. This is also a great opportunity to re-practice the skill taught in CS136L, how to use git. It prepares us not only for the final assessment but also for real-world software development scenarios where these principles and techniques are commonly applied.