Overview

Our version of Watopoly is focused on classes and leaves room for abstraction as well as improvements. We will go over a couple of the key components within our project, and how they contribute to the overall functionality of the project. Here we will go over a high-level overview of the most essential classes to the project and how they work together to create a cohesive Watopoly game.

- **Abstract Tile Class**
    - All the intended buildings in Watopoly are subclasses of an abstract Tile class.
    - The Tile class utilises the PIMPL idiom, storing a pointer to a structure with all of the required fields for our implementation.
    - Since a Tile will never need to be instantiated, it is abstract with pure virtual methods. All other types of buildings extend from a Tile class.
    - As some methods were specific to certain classes, these were added as virtual methods to the Tile class. They were not pure virtual as this would require adding an unnecessary empty implementation for each different type of Tile.
    - The only pure virtual function implemented was action(), which would act accordingly for the specific tile that was landed on by a player.
- **Derived Tile Subclasses**
    - Derived subclasses all shared one thing in common, the action() function.
    - Each type of building was a new tile subclass
    - Relevant buildings were grouped to the same class with small modifications - for example, all academic classes had only a singular class, as there were only small logical differences required for the costs of the improvements, & tuition with improvement costs.

- **Board Class**
    - The Board class creates and stores a vector of Tiles, while also having all of the user interface required for intaking commands from the user. The board doesn’t own this vector of Tiles, as it is instantiated as a shared_ptr since the class BoardDisplay is in need of them.
    - The Board also instantiates a vector of class Player, which it uses to modify the information depending on the various commands & actions each player may choose. The modifying of player fields is done through getters and setter functions in Player.
    - The Board also stores a class BoardDisplay, which is used to display the current state of the game, including the Board and the Players.
    - The majority of the user interface was implemented in the Board’s play() method, which would then act accordingly based on what command was entered.

- **Player Class**
    - As it would indicate, the Player class stores basic information about the player, such as their money, position, jail status, turns in jail, name, & piece chosen. Most of these are instantiated with default values, except for piece & name.
    - In main.cc, the user is prompted for information regarding the player name, and which piece they pick, which is then passed to the player constructor.
    - As previously mentioned in Board, the Player class has setters and getters for the Board to change these various fields.
    - Similar to Tile, Players owns a pointer to a structure with all of the required implementation fields, making use of the PIMPL idiom.

That was a brief overview of the most important classes in our Watopoly, & how they work together to form the overall structure of the project.

Updated UML

- Our updated UML was not too drastically different from our original. We had a few select added methods for some Tile subclasses, as well a

Design

- **Template Method**
  - We used the template method for our implementation of all the Buildings, with an abstract base Tile class, and then Tile subclasses that descended from it. As such, we could put methods that were applicable to multiple Tiles in the abstract class, while very specific implementations were made into virtual functions with override capabilities.
  - This method allowed for ease of capability in completing nearly all the buildings, as we knew that if one was implemented correctly, all the other ones would have no issue being created. As such, the only challenge for each type of building would be to correctly implement the action that occurred once you landed on the square.
- **BoardDisplay**
  - As posed in the Watopoly pdf, we went over the pros and cons of using the observer method for the display, but ultimately ended up attempting to implement just a txt file with the board, which could be updated any time the board was to be printed. However, there ended up being many difficulties in

    - updating the board, as getting the specific indices correct for modifying proved to be a challenge.
    - Ultimately, the display was hardcoded and was definitely a segment that could have been improved upon to allow for better resilience to change. It however suited the needs for our design of the board, and displayed all improvements as well as player positions correctly.
- **PIMPL Idiom**
    - The PIMPL idiom was used for us to have higher cohesion, and lower coupling. By inserting our private fields into a structure pointer, it allowed for us to create and modify additional fields as we saw fit, that we may have not initially seen from the beginning of the project.
    - This also allowed for cleaner code, and to avoid clutter in private fields by having excessive variables that would distract from the other methods in the Classes.

Resilience to Change

Our Watopoly supports changes through many different avenues. We will go through many specific aspects of our code and explain how it may be modified or added to create alternative versions of Watopoly.

- PIMPL method
    - As mentioned in the design section, as we used the PIMPL design idiom for our private fields in Player as well as Tile, it means that we can easily implement private fields that may be needed, while not drastically increasing compilation. For example, things such as: Horse Rules, could easily be implemented by adding the fields to our impl pointers, and then simply

modifying them through a getImpl() function, which would return a pointer to the implementation structure.

- Abstract classes
    - The use of the abstract tile class easily allows for more types of buildings to be implemented. Creating another subclass that extends from Tile, such as plaza restaurants like Lazeez, could be implemented with ease.
- Private constants
    - The use of private constants fields such as “jailspace”, and “gotoJailSpace”, which stored the respective indices of DcTimsLine, and gotoDCTimsLine, can easily be modified in order to shuffle some of the tiles on the board around.

Answers to Questions

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

Initially, we thought that the Observer pattern would not be a good fit for implementing the game board. This was because it would be unnecessary and redundant, to create a subject, observer, and create notify() methods, all just to update a text display that would involve a 2D vector of characters. We thought that it would be more effective instead to have an update function for the Display that could be called upon printing the function.

After implementing the gameBoard and the display for Watopoly, we still think that the observer pattern would be unnecessary to implement.

We ended up having some struggles with the board display, and implemented it in a different way from what we initially intended. Although we did not end up implementing the visual

display of the board in the way we intended, we still think that the observer pattern would be unnecessary. It still would create a lot of unnecessary code, which would lead to higher coupling as many more classes would depend on each other. As such, this would make it more difficult to identify potential errors and the source.
Still, the ideal method to display the board would be for the BoardDisplay class to have a pointer to the board, which it could then also use to get the list of players. Furthermore, there could then be a text file that contained the basic form of the board. This text file would be initialised onto a 2D char vector. Finally, when called to print, the points on the vector that could potentially contain players or improvements would be updated to represent the current state, and then the vector would be outputted, line by line.

Question. 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?

Implementing the Chance and Community Chest cards, we would use the template method design pattern. This would be created with an abstract deck class, which would have two separate classes which could act as our public interface, and would have two concrete Deck classes in chance and community chance decks. Then, our decks would interact with all of our cards. Our Cards would be also abstract class, with virtual void operations such as :action();

Then, we would additionally have our concrete Chest and Community Cards which would have actions that would be decided at our discretion.

We ended up not modelling SLC and Needles Hall similar to the Chance and Community Chess cards. However, we stand by the initial opinion of how it could have been implemented if we did model it as such. To add on, we could have one deck of 24 cards, 1 deck of 18 cards, and we would have the number of each represent the probability that they get chosen. E.g. 4 cards of Back 2 for SLC, 3 cards for -50 for Needles Hall. SLC and Needles Hall would both own-an instance of their respective decks.

Our approach to the SLC and Needles Hall model was to create a pseudorandom number generator, in an int vector of size 24 or 18, respectively. Then, ranges were assigned to a specific move based on their probability. For example, in SLC, if the vector randomly returns a number from 1-3, the action would be to move back 3 spots. Similarly, a number from 4-7 would be to move back 2 spots. This logic was carried through for all moves of SLC and Needles Hall.

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

Initially, we thought that the Decorator pattern would be unnecessary to implement, similarly to observers, because it would be a lot of unnecessary work to create all the abstract classes and create 5 MORE additional classes for each of the levels of improvements, just to store how much a building's tuition would cost. Furthermore, this would also achieve higher coupling through more classes having dependencies on each other as well.

Instead, in the academic class, we stored a simple vector that represented the improvement tuition for each building, and the current level that the improvement cost was at. The class would then retrieve the cost from that improvement index of the vector to determine how much the tuition should cost per level of improvement.

Extra Credit Features

The most important enhancement that we implemented was completing our project entirely without leaks, using no dynamic memory allocation, and only smart pointers, vectors, and raw pointers that don't express ownership. In the entirety of our program, we did not write a single delete as our memory was entirely explicitly managed. However, what we didn't have in memory leaks we paid for in implementation. As we had only learned about shared pointers and unique pointers a brief time before the start of the project, it was our first real opportunity and chance to use the new tool and get used to it. As such, there were many, many declaration errors as well as confusion with typing, unique vs smart pointers, etc. We solved all of these issues over the course of the project, as we became more comfortable with these smart pointers, the benefits, the disadvantages, and how to use them so as to not require any memory manipulation.

Final Questions

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

Documentation

- Documentation is vital when there are multiple pieces that are reliant on each other.
- Writing clean and clear code would allow anyone to jump into a method to understand how it can be used; this also can save a countless amount of time as others don't have to sift through an entire chunk of code to understand its purpose and how it works.

Collaboration

- Git is a very strong tool and with more familiarity, we were able to work more efficiently.
- Divvying up tasks based on team members' strengths and weaknesses allows for a more efficient allocation of resources

- E.g.
    - Daniel, who had many prior experiences in working on a software team from coops, organised, planned, and set the team on track for the duration of the project.
    - Jason, who has a strong eye & patience for debugging, performed a large amount of the testing for the executable.
    - Anthony enjoyed problem-solving and going through logic-based questions so he implemented some of the more difficult logic areas of the project.

Expectations

- Although we created our UML classes and design document at the end of the page, we ended up having to make modifications to a few classes to better improve the functionality of our project. This is of course not an issue, but it was naive to perhaps expect the UML to carry through for the duration of the project. It leads to stubbornness at times, of not wanting to sway from our original UML, but in the end, most changes lead to improved implementation.
- As such, having the awareness and expectations about what parts of the plan may not have been as concrete, as well as backup plans for how they might work was an important lesson learned in planning as well as expectations going into a project.

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

Planning

- We would have worked on load and save more towards the beginning as they were essential parts of the code for command line arguments, but we ended up leaving them more towards the end. It was originally planned to be completed at the start but the pressure of getting a working product felt overwhelming so this ticket was backlogged and left till the end. It was however completed

- Another issue was the deployment of the board took quite a while. This is another planning issue, potentially developing a more concrete way to display the board from the beginning would be a lot more beneficial. Because of the stress as the project went on and the lack of concrete ideas for a working implementation, the board ended up being hardcoded which offered a low resilience to change and increased coupling between the BoardDisplay and the Board.

Conclusion

To conclude, over the course of 2 weeks, our team got quite familiar with the design process and brainstorming before tackling a large project such as Watopoly. Although we did quite a bit of planning ahead, we learned that plans are often made to be altered & improved. This allowed us to improve upon our initial ideas, which was made easily by our devotion to abstract classes and the PIMPL idiom. Despite its brief shortcomings, our version of Watopoly can easily accept improvements and modification to the buildings, board, or rules of the game. We thoroughly enjoyed working together on this project and had a great overall learning exprience