Introduction

This project involved implementing Watopoly in C++, a spinoff of the popular Monopoly Game, based on the University of Waterloo campus. The game has 40 squares, and players take turns moving around the board, buying/improving buildings, and paying tuition. The major components of the game include the board, buildings (Ownable and Non-ownable ones), and squares(cells). The board should display the 40 squares, any improvements, and a representation of each player's location.

In summary, we followed our original UML quite closely with a few minor changes: for example, we removed the Ownable subclasses, merging them into one subclass of Cell and similarly with NonOwnable properties. **This was done to decrease the amount of coupling and increase cohesion**; as well as remove repeated code. Furthermore, we added methods, where appropriate; and kept with the original **Observer Design Pattern**, and also stayed true to our decision to not utilize the Decorator Pattern to implement Improvements.

Overview

The **Observer Design Pattern** was used to implement the gameboard, where the **subjects, Player and Cell** maintained a list of their dependents **(Observers); specifically: Text Display,** notifying the Display (visualization of Board) automatically of any state changes, by calling the notify method; automatically allowing the observer (TextDsiplay), to stay up to date with any changes made to the subject. Since the Observer pattern promotes loose coupling between objects, this was indeed a flexible and scalable design choice.

Additionally, by using the virtual "event" method in Cell and overriding this method in every subclass of Cell: Ownable and Nonownable, this improved maintainability, extensibility and allows for the separation of concerns, where the events that should occur once a player lands on a square (Cell) are handled in those subclasses, without changing the structure of those Cell objects being visited.

Main.cc is the entry point to the program, and it controls which functions to call for particular command line arguments: "roll, next, trade, improve, mortgage, unmortgage, assets, bankrupt, all, save etc." If no command line arguments a given, a default Watopoly Game is loaded from a file and if such arguments are given, the gameboard is initialized with those. There are 2 modes: Testing and Normal Gameplay.

The following classes are used: Cell (each square on the board), Board (a collection of 40 cells), TextDisplay (aids in printing to the screen and automatically updating Board), Player (for creating Player objects in the game), State (state of player/cell), Ownable (for ownable properties: Academic Buildings, Gyms, Residences) and Non-Ownable (for non-ownable properties : goose nesting, gototims, needleshall, slc, collect osap).

Cell (subject) handles representing every square on the board and has methods/attributes which determine its name, if the cell can be owned (has type Otype), and if so by whom, it's cost, whether it's mortgaged and the number of improvements on it, as well as its location on the board, and a couple Boolean variables to aid in notifications. Each Ownable Cell is of type shared pointer, since **shared pointers for ownership** allow better memory management and movement of property from bank to player or player to player.

Player (subject) creates a player object, and has the following attributes: name, character piece, money, number of roll up rim cups, position on the board, and Booleans for being bankrupt or in Tims (jail)and a **vector of shared pointers** referring to their owned Properties, as well as a map of faculty names to number of such properties owned by a player and whether a player has such a monopoly. Two aspects that stand out are: the use of vectors and **shared pointers** for properties allows for safe memory management, easy changes of ownership, and efficient access to every building; and the **map** was a short clever method to make it easy to check for improvements/determine if a property can be mortgaged (being part of a monopoly is considered in both). Additionally, Player has methods which control the core functionality of the game such as moving, buying/selling properties, trading, mortgaging, and calculating one's assets; as well as a notify method; which is key for the observer pattern.

TextDisplay (observer) contains a **vector of characters** (as the display) and a reference to Board; and updates player positions, property improvements on the board, as well as notifies board of such changes; allowing the display to be created.

Board contains a **shared pointer** to TextDisplay and a vector of shared pointers (Cells), and is used as a visual of the gameplay, i.e., initializing the board and every single cell (square) as well as is responsible for loading/saving games and determining which player wins or when the game is over and also for taking care of auctions.

Ownable and Nonownable are subclasses of Cells. Ownable properties, have a Boolean for determining if a cell is mortgaged and methods for buying/selling improvements, paying Tuition, mortgaging, and setting faculty names. One notable way, we managed Ownable Cells was using a map with strings (cell names) as keys and tuples with faculty names, purchase costs and improvement costs/tuition, which allowed for easy/efficient accessing of such information. Nonownable took care of the slc, goose nesting and tims properties on the board, initializing them. Both such classes have a notify method which comes from Players and cells.

State is quite important for the Observer Pattern and in this case state type is one of Purchase, Mortgage, Unmortgage, AddImprovements, Sell, Unmortgage, Landed, Move, Playing, which are all important for cells and players and essential in notifications. State also has a player position, cell name and new owner attribute.

Updated UML

Design

[Note: this continues from the Overview above where I gave an overview of all the classes as well as some notable Design decisions: e.g., smart pointers, vectors, maps etc]

The following will be discussed: the **utilization of the observer pattern** where the subjects are Player and Cell and the observers also Player and Cell as well as TextDisplay where TextDisplay is a class for aiding in printing the Board after very notification (the Board has a pointer to a TextDisplay Object), as well as the use of Ownable and Nonownable : subclasses of Cell aimed at storing information on all cells on the board (the board is a collection of 40 Ownable and Nonownable cells). Furthermore, we'll dive deep into the virtual event method in Cell and why we overrode this method in every subclass of Cell.

The use of the Observer pattern in the TextDisplay class is a superior design choice as it allows for a separation of concerns and promotes modularity. By having TextDisplay inherit from Observer and observe changes in the Board (i.e., changes in player and cell states: subject states), the TextDisplay class does not need to directly access or modify Board's state. Instead, it only needs to react to the changes it receives through the observer notifications, and thus update the Board since the Board has a pointer to TextDisplay.

The use of structs/Enums (State and Statetype) to hold information relevant to each type of notification is also a superior design choice as it helps to maintain clarity and organization within the code. This allows the TextDisplay class to easily differentiate between the different types of notifications it receives and to extract the relevant information it needs to modify its output. It should also be noted that for every notification, we've done it such that all we require is the Cell name (essentially), and from this Cell Name, we can determine, whether the cell is ownable or not, and if it is: it's owner, its cost, the number of improvements, whether it's mortgaged etc.

It should be noted that by using a generic notification struct that can hold information for all types of notifications, helped reduce code duplication and make the implementation more extensible in the future.

We've separated the view and action aspects of the game in separate classes as is evident above; where text Display and Board handles the view aspect and Player, Cell (and subclasses), takes care of the controls and actions or events in gameplay; making our code modular, high in cohesion and low in coupling.

Additionally, by using the virtual “event” method in Cell and overriding this method in every subclass of Cell: Ownable and Nonownable. This improves maintainability, extensibility and allows for the separation of concerns, where the events that should occur once a player lands on a square (Cell) are handled in those subclasses, without changing the structure of those Cell objects being visited), and encapsulation of functionality, allowing for easy modification or addition of behaviours without affecting other parts of the system. This design pattern promotes extensibility by enabling the addition of new Cell types or player actions without affecting existing code. It also helps with testing, as individual Cell behaviours can be tested in isolation.

Furthermore, the pattern can be easily adapted to implement additional features, such as different game modes or rule sets. This flexibility is key to creating a dynamic and engaging game that can evolve over time to meet the needs and preferences of players.

One notable deduction from the DD1 UML was the removal subclasses of Ownable and Nonownable. Whilst we initially did create all of those with the intended methods, we experienced some difficulty in combining all of the code, because of the multiple inheritance. Ownable and Nonownable inherited from Cell, and each had several subclasses: Residences, Gyms and Academic Buildings, Goose nesting, Collect Osap, Tims etc. Upon merging them all together, we encountered a couple type errors which prompted the use of Dynamic Cast; resulting in unnecessary code complexity and increased all around bugs. Hence, we instead deleted all such subclasses and combined their functionalities into Ownable and Nonownable respectively, making use of the overridden “event” method from Cell, as well as adding a few more member functions in each to handle the logic. Without a doubt, this increased code readability and cohesion and decreased complexity, and coupling with 7 less classes.

Lastly, one design decision, I felt was quite smart was our usage of vectors and smart pointers in the Player and Board Classes; and maps in the Cell and Player Classes; which increased efficiency, ensured safe memory management, and reduced repeated code.

Certainly, it may have been beneficial to include error handling in the TextDisplay class in case it receives notifications with unexpected or invalid data but unfortunately, we fell short of time.

Resilience to Change

Our implementation is fairly resistant to change, as we separated the view and controller/action aspects; and our notification system is fairly generic, such that all we require is the Cell name (essentially), and from this Cell Name, we can determine, whether the cell is ownable or not, and if it is: it's owner, its cost, the number of improvements, whether it's mortgaged etc; and hence make appropriate decisions.

The first potential change is changing the graphics from standard output to a graphics display.

and creating different board themes, which should be fairly simple as we would only need to change the TextDisplay Class since it handles all of the output and can be changed to make a graphical one/apply board themes. We would need to write a separate GraphicsDisplay class and have that observe the Board (Cells and Player) with the same methods as TextDisplay. Accommodating this change only implies writing one extra class and should be fairly simple like A4q4.

The second change is the implementation of house rules. The first is the tuition money collection in the middle (starting at $500 or $0) which only requires adding an attribute to Board to keep track of the money pile in the centre, and modifying the event function in the NonOwnable Class to handle the tuition cases (which should be similar to the Ownable Tuition function) and the Player class already has all of the methods necessary for paying and receiving money. Finally, the last house rule is collecting double the money from OSAP which simply requires modifying the event function in the Nonownable to double the OSAP money to be paid. Hence, our implementation is resistant to implementing these house rules.

Implementing the even build rule, and limiting improvements, is fairly easy since we stored the number of properties owned by every player in a vector of shared pointers and also had a map of faculty and properties owned and whether it was part of a monopoly. Also, we kept track of improvements for every player's property within the player class; and this should be fairly simple to manipulate to perform such rules/functions.

However, while our implementation of Watopoly is resistant to such changes, it would prove a bit difficult to implement SLC and Needles Hall in the same manner as Community Chest and Chance cards. Two solutions are using a vector to store all functions for each "card" for each of the 2 squares, but this requires a lot of extra methods, and another is using the Template Design Pattern which was discussed later on (in the Project Specification Question Section).

Answers to Project Specification Questions (Mostly unchanged from DD1)

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

The Observer Pattern is a behavioural design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This pattern allows objects to be decoupled and promotes loose coupling, making the system more flexible and easier to maintain. In the context of Watopoly, the Observer Pattern was used to notify the gameboard and other components of the game when a player's location or state changes. For example, when a player moves to a new location on the board, the player object notified the board object of its new location, and the gameboard object updated its visual representation accordingly. Similarly, when a player's state changes (e.g., they went bankrupt), the player object notified the board object and other relevant objects (e.g., other players) of the change, and they can update their states accordingly. The Observer Pattern was also used to implement other features in the game, such as notifications for players when they land on certain squares (e.g., the Collect OSAP square) or when they are required to perform certain actions (e.g., roll the dice or pay rent). In these cases, the gameboard squares (cell) acts as the subject, and the player objects can act as the observers. Overall, the Observer Pattern was a useful pattern to use when implementing Watopoly, since it promotes loose coupling between objects and allows for more flexibility and maintainability in the system, as instead of creating a new gameboard every time, there is a change, we can simply notify the observers.

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

Implementing the Decorator pattern would marginally decrease coupling and increase cohesion, as it allows each Property class to be more single purposed to simply store values, while the Improvements decorator class can manage applying Improvements. However, in this game there are multiple different types of cells which can or cannot have improvement methods, which all inherit from an abstract cell class. Implementing the decorator pattern for this would require a lot of extra classes and setup which clutters the design past what is needed. A much simpler approach is to simply have a private Improvement method for only Ownable classes. We held true to this decision; with buy and sell improvement methods in the Ownable subclass which inherits from Cell.

3. **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?**

To model SLC and Needles Hall more closely to Chance and Community Chest cards, the Strategy Pattern or the Template Method Pattern can be used. These patterns can help to encapsulate the behaviour of the SLC and Needles Hall spaces into separate classes, allowing for more flexibility and ease of modification in the future. By using the Strategy Pattern, for example, a set of classes that represent different types of SLC and Needles Hall (cards) would be defined, each with their own behaviour, and then a context class is used to select and execute a specific (card) based on game conditions. This allows for a more modular approach to coding the behaviour of the SLC and Needles Hall spaces, which can simplify maintenance and updates to the game in the future. The Template Method Pattern could be used to define a basic structure for SLC and Needles Hall (cards), with specific methods that can be overridden in subclasses to provide customized behaviour. In this pattern, the basic structure of the SLC and Needles Hall cards would be defined in a parent class, and the specific behaviour for each card would be implemented in its own subclass. This approach allows for greater flexibility and customization of the behaviour of the SLC and Needles Hall spaces, as each card can have its own unique set of behaviours while still adhering to the basic structure defined in the parent class. We abandoned this idea for bonus marks and instead had a NonOwnable subclass which inherits from Cell and dealt with all non-Ownable type squares using the single overridden event method. This decreased code complexity and instead of having multiple inheritance, allowed for an overall smoother implementation; and solved errors associated with incompatible types.

Extra Credit Features:

We utilized shared **pointers for ownership** to allow better memory management and rarely (if any) used new and delete.

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?**

Working on a software team can be a challenging experience that requires effective collaboration and communication. The most significant lesson we learnt from this project is the importance of proper planning. Without a clear understanding of the project's objectives and a well-defined plan, it was easy for us to become disorganized and inefficient. Therefore, at the start of the project, we focused on creating a well-designed UML and a detailed timeline of each team member's responsibilities. This allowed us to work more efficiently and with less confusion. This ensured that we were all working towards the same objectives and that the project remained on track. By having a clear plan in place, each of us could have focused on his tasks without worrying about aspects of the project, outside his scope.

Another lesson we learned is the value of maintaining open communication throughout. By staying in regular contact with one another, we kept each other updated on our progress and ensured that everyone was on the same page, making it easier to spot potential conflicts within one another’s code and resolve them before they caused any issues.

We learned that it's essential to break down the project into smaller, manageable tasks, to avoid becoming overwhelmed. Furthermore, it's vital to test and debug the code regularly to catch any errors early and ensure that the project remains on track.

In conclusion, developing software in teams requires effective planning, communication, and collaboration, as well as breaking down the project into smaller, manageable tasks regular testing and debugging. By creating a clear plan, maintaining open communication, and working together towards a shared objective, we achieved our goals more efficiently and with fewer errors.

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

Reflecting on our experience with the project, there are several areas where we could have done things differently if given a chance to start over. Firstly, we would allocate more time to each part of the project. We underestimated the amount of time it would take to implement the core functionality of the Watopoly immediately started thinking of the bonus features. By doing so, we compromised on the quality of the core game and ended up with less time to work on the bonus features.

Therefore, if we had a chance to start over, we would create a more detailed and realistic timeline that accounts for all aspects of the project. This would allow us to have enough time to work on each component and produce a higher-quality product.

Secondly, we would prioritize regular check-ins with one another to ensure that we are on the same page and that everyone is making progress towards the project's objectives. This would help us identify potential issues early on and allow us to address them before they become significant problems.

Additionally, we would incorporate more testing and debugging phases into our project plan. We found that we spent a lot of time fixing bugs and errors towards the end of the project, which could have been avoided if we had tested the code regularly. By including testing and debugging as part of our project plan, we could avoid these issues and ensure that the final product is of high quality. We wrote most of the code before starting testing; in retrospect we should have gotten the board printed and then started debugging/writing more code from there.

Finally, we would make sure to allocate enough time to implement more of the bonus features. Although we were able to implement one of the bonus features, (the use of smart pointers), we were not able to include all the features we initially planned.

In conclusion, given a chance to start over, we would prioritize a more detailed project plan, regular check-ins with one another testing and debugging, and more time for implementing bonus features. By doing so, we could produce a better product that meets all project specifications and meets our high standards.