PROJECT SUBMISSION DD2

By Manit, Rhythm and Vansh

1. Introduction

Watopoly is a Waterloo-themed Monopoly game implemented in C++. The game follows traditional Monopoly mechanics while incorporating University of Waterloo elements such as campus buildings, gyms, DC line for the jail and much more! Watopoly can be played in the regular mode and the Test mode, and also has the feature to load and save a game, to play from wherever we leave it.

Watopoly helps us learn all the Object Oriented Programming core skills, and implement them in a team setting. Writing documentation on our project with a UML makes it very professional and is beneficial in the long run. This document outlines the final design, implementation details, design patterns used, and how the project accommodates change. It also includes an updated UML class diagram reflecting the final structure.

---

2. Project Overview

Watopoly is a theme-board game that mimics Monopoly but set in a University of Waterloo context. Students buy and trade campus buildings (MC, DC, EV3), pay fees, auction, and encounter special cards like "Go to Tim's" (jail) and "Needles Hall" (random money bonuses/punishments). By maintaining the core rules of Monopoly, Watopoly introduces UW-related elements to give a new and exciting experience.

The game has turn-based game play with the players rolling dice, moving, and taking actions like trading, mortgaging, and expanding properties. Property management includes dynamic determination of rent levels based on expansion and ownership. There is an auction mechanism for competitive bidding on the unowned properties, and financial transactions like rent, tuition fees, and trades are handled by the Transactions class. Special tiles such as Go to Tim's, Needles Hall, and Goose Nesting introduce more strategic elements. The Load system makes it possible to save and restore progress in-game.

Watopoly employs an interactive dark mode GUI on X11, dynamic player token colors, property colors matching faculty-themed styles, and real-time tracking of progress and mortgages. The game even employs today's best practices of memory management in modern C++, making use of shared_ptr to hold shared critical objects in the game and unique_ptr to keep exclusive resources. Such advancements in addition to true gameplay mechanisms and UW-inspired designs make Watopoly technically sound yet still fun to play. (This is our bonus implementation)

---

3. Changes from Design on Due Date 1

Since our initial proposal, we have made several significant changes to our design due to unforeseen complexities and the need for better modularity.

Our original UML did not fully account for all possible cases we had initially considered, nor did it optimize for potential bonus marks. As we progressed through the project, we realized that many components were overlapping, leading to an overly long and complex implementation. For example, our Player class initially contained between 1,500 to 2,000 lines of code. To improve maintainability, we restructured our design by delegating responsibilities to other classes that managed transactions between players and auction mechanics.

Initially, we also planned to integrate the load and save game functionalities directly within the Controller class. However, as development progressed, we found that the Controller was becoming the central hub of the project, making it excessively large and difficult to manage. To resolve this, we created separate classes dedicated to handling loading and saving operations.

Furthermore, we initially intended to have the display interact directly with the Controller. However, we soon realized that updating the display every time a change occurred was cumbersome and inefficient. To address this, we introduced a GameBoard class that acts as an observer, responsible for notifying the display when updates are needed. This change not only improved efficiency but also ensured that we could better track players and improvements on the board.

Another major modification involved the handling of non-building tiles on the board. Initially, we did not consider creating a dedicated class for each type of non-building tile. However, as our implementation evolved, we recognized the benefits of doing so and subsequently implemented individual classes for these tiles.

These design changes allowed us to create a more structured and modular implementation, making the project easier to maintain, extend, and debug.

---

4. Class Structure and UML Diagram

4.1 Key Classes & Responsibilities

The UML diagram provides a structured representation of the core components of our game system. Below is an overview of the key classes and their responsibilities:

GameBoard

The GameBoard class is responsible for managing the game state and board layout. It tracks players' positions, updates the game display, and facilitates movement. It provides functions for rendering the game board, managing player movements, and saving or loading game states. These functions ensure that the game progresses smoothly and that player actions are accurately reflected on the board.

Player

The Player class represents an individual player in the game, maintaining information such as their position, balance, owned properties, and in-game status. It allows retrieval of player details and handles financial transactions, including modifying cash balances, making payments, and receiving funds. Additionally, it includes mechanisms for declaring bankruptcy and determining when a player is bankrupt, ensuring proper game flow and rule enforcement.

Building

The Building class serves as the base for all purchasable properties. It stores details about ownership, mortgage status, and improvement levels. Players can buy, sell, and improve buildings, and the class provides functionality to determine the cost of purchase and rent calculations. It also manages the mortgage status of properties, allowing players to strategically mortgage or unmortgage assets when needed.

NonBuilding

The NonBuilding class represents special tiles on the board that do not function as properties, such as Needles Hall or Tuition. These tiles trigger unique in-game events that affect players differently, adding elements of chance and strategy to the gameplay.

Transactions

The Transactions class manages financial interactions between players, the bank, and the game board. It handles rent payments, property purchases and sales, and the mortgage process. Additionally, it facilitates trading between players, allowing for asset exchanges and negotiations. This class plays a crucial role in ensuring smooth economic operations within the game, making transactions straightforward and fair.

Controller

The Controller class is responsible for processing player commands and updating the game state accordingly. Acting as an intermediary between user input and game mechanics, it executes player actions such as trading, improving properties, and managing mortgages. It also oversees bankruptcy proceedings, auctions, and dice rolls, ensuring that all player interactions follow the game's rules and logic.

4.2 UML Diagram

The UML diagram provides a visual representation of the game's class structure and relationships, emphasizing the interactions between the key components described above. It effectively illustrates the flow of data and dependencies, ensuring a clear understanding of how each class contributes to the game's mechanics.

5. Implementation Details

The Controller class manages the game loop, iterating through players in a predetermined order. Players roll dice via CommandRoll, advancing their position on the GameBoard. Trade, mortgage, and improvement commands are executed by Controller methods that interact with Player and Building classes. Rolling doubles provide extra turns, and landing on Tims Line triggers jail logic. Bankruptcy is handled by Player::declareBankruptcy, transferring assets via Transactions.

Properties are constructed with a Factory-like approach, having Academic, Gym, and Residence subtypes. Ownership is controlled with Building::setOwner, and rent calculation is based on subtype. Unowned properties initiate an auction by calling Auction::place, and purchase is handled by Transactions::addPropByAuction. Mortgages are enabled with Building::setMortStatus, and monopoly blocks are handled by Player::updateMonopolyBlock.

The Transactions class aggregates money flow, including rent (payRent), purchase (buyBuilding), fees (CoopFee::pay), and trades (trade1, trade2). Upgrades change through buyImprovement, altering Academic::setImprLevel and GameBoard::squareImprovements. The Load class deserializes game state, saving player data, board state, and property data. During loading, GameBoard reconstructs the board, resetting players and properties, enabling game continuation.

---

6. Design Patterns Used

Our Watopoly project is a top-level C++ realization of a Monopoly-style board game, with an emphasis on modularity, extensibility, and object-oriented principles. The implementation makes use of a variety of design patterns to manage complexity and achieve scalability.

The game itself possesses a Factory Method pattern for dynamic instantiation of different types of properties (e.g., Academic, Gym, Residence) from propertyArray.h data, decoupling instantiation logic from the game board. Strategy pattern is also applied to non-building classes like CoopFee and GetInTim, where some action (paying fees, jailing players) is wrapped as interchangeable strategies. The Controller class is also used as a Facade by providing a single entry point to manage subsystems like auctions, trades, and upgrades to property, thereby simplifying the interaction for the game loop. Meanwhile, the Auction class is an implementation of the Mediator pattern, localizing bidding logic and controlling player interaction to enforce auction rules without direct dependencies.

State management within our game is strongly supported via modern C++ practices, including smart pointers (shared_ptr) for memory security and observer-like objects. The GameBoard and WatopolyDisplay classes employ an Observer pattern, the display being updated dynamically when players move or properties become different states. The Dice class uses a Monostate pattern (Singleton variant) to offer consistent roll states throughout the game. Also, Player class manages rich states (bankruptcy, in jail) that could expand into a State pattern for improved transition. The Transactions class encapsulates economic activities (trades, mortgages) into Command-like static methods to promote separation of concerns and reusability.

In general, our project design is robust on scalability—new game rules or properties can be introduced without interfering with the current operation. Components are kept decoupled through the use of design patterns like Factory, Strategy, and Mediator. The latest features of Modern C++ ensure maximum maintainability. The design is not only sustainable for the current set of features but also provides a good foundation to implement future enhancements, like introducing new game modes or GUI improvement, thus Watopoly a well-engineered and flexible board game implementation.

---

7. Object-Oriented Principles

Our Watopoly project adheres to key object-oriented programming (OOP) principles, ensuring modularity, maintainability, and scalability. The design enforces cohesion, low coupling, encapsulation, and abstraction to create a structured and efficient system.

7.1 Cohesion and Coupling

High cohesion is maintained by ensuring each class has a single, well-defined responsibility. Core components such as GameBoard, Player, Controller, and Display are structured independently, minimizing overlap and improving readability. This approach enhances code clarity and facilitates debugging.

Low coupling is achieved through interfaces and dependency injection, reducing direct dependencies between classes. This allows for easier modifications and future expansions, such as adding new game mechanics or modifying existing rules, without impacting unrelated components.

7.2 Encapsulation

Encapsulation is strictly followed, with all data members set to private and accessed through getter and setter methods. This prevents unauthorized modifications and ensures data integrity. Key game elements, such as player balances, property ownership, and board state, are securely managed within their respective classes.

By maintaining a clear separation of concerns and adhering to OOP best practices, our design ensures Watopoly is both robust and flexible, making future enhancements seamless and maintainable.

---

8. Accommodating Change

Our Watopoly design follows strong object-oriented principles, ensuring flexibility for future modifications while maintaining a clear structure. We prioritized modular design, encapsulation, and abstraction, allowing new features to be added with minimal disruption to existing code.

Adding New Tiles:

The game board is designed as a collection of tile objects, where each tile type (e.g., properties, utilities, special event spaces) inherits from a base Tile class. This makes it easy to introduce new tile types by simply extending this class and implementing any unique behavior. The board setup dynamically reads from a configurable structure, ensuring that new tiles can be integrated without altering the core game logic.

Modifying Rent Rules:

Rent calculations are encapsulated within property objects, making them easy to modify. Instead of hardcoding rent logic, we use dynamic attributes like the number of improvements and ownership status to determine rent. If future changes require progressive tax models, different rent multipliers, or special discounts, these can be incorporated within the rent computation function without affecting unrelated parts of the code.

Introducing New Game Mechanics:

Our event-driven architecture allows new mechanics to be introduced by extending existing game actions. For example, if we wanted to add auctions, special ability cards, or new movement rules, these could be implemented as new Action classes without modifying the core player or board logic. Additionally, our turn-based system is designed to accommodate conditional mechanics, making it straightforward to introduce special events or rule variations without breaking game flow.

Minimizing Dependencies & Maintaining Scalability:

By adhering to low coupling and high cohesion, we ensure that modifying one component does not have unintended side effects elsewhere. Each part of the game – board, tiles, players, and game rules – is modular, meaning new features can be developed and tested in isolation before being integrated. This approach also makes debugging and future expansions significantly easier.

Overall, our design choices allow Watopoly to remain scalable, adaptable, and easy to modify, ensuring that future enhancements can be seamlessly incorporated without overhauling the existing codebase.

---

9. Project reflection DD2 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?**

This project provided an excellent opportunity to work as a team. Managing different branches introduced us to numerous Git conflicts, helping us develop strong debugging skills. We had to split tasks, coordinate effectively, and balance individual coding styles. Some team members preferred extensive comments, while others wrote minimal documentation, which required us to understand and integrate each other's code efficiently.

One major lesson we learned was the impact of delays. Missing one person’s deadline affected the entire team’s progress, reinforcing the importance of planning and accountability. For example, we initially planned to implement a bot but didn’t allocate enough time for debugging. While we knew how to build it, we couldn't complete it due to time constraints. This experience highlighted the need to account for unexpected challenges in project timelines.

Additionally, we had to make collective decisions regarding class structures and feature implementations. We often debated different approaches, which clarified concepts for everyone and reinforced the value of collaboration in coding. This project has been a crucial learning experience, preparing us for future team-based software development roles.

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

If we had the chance to start over, we would have begun the project earlier to allow more time for debugging and refining our design.

One major change we would make is breaking down large classes, such as Player, into smaller, more modular components using virtual functions and a more abstract design. This would have improved code organization and maintainability.

Additionally, we would have taken a different development approach by compiling from the start and following a top-down methodology instead of bottom-up. Our Controller ended up being around 1300 lines, making debugging challenging. If we had structured it more clearly from the beginning, task distribution would have been easier, giving everyone more flexibility. Instead, one person had to handle a disproportionately large workload, while others, particularly those working on the Controller and Display, had to wait for foundational components to be completed before proceeding.

Lastly, we could have structured the project better to allow for easier configuration and integration, reducing delays and improving overall efficiency.

---

10. Final Answers to Project Specification Questions:

After completing the project, we still hold the same views as our initial responses from Due Date 1 regarding the design patterns used in Watopoly.

1) Observer Pattern & Game Board

We still believe the Observer Pattern is unnecessary for the game board. Since board squares rarely change state and do not need to constantly notify other components, a simpler approach is more efficient. Players also do not require automatic notifications for property state changes, as they interact with them only when they land on the respective tiles.

2) Modeling SLC & Needles Hall as Chance/Community Chest Cards

We maintain that the Iterator Pattern is the best fit for handling SLC and Needles Hall, just like Chance and Community Chest cards. It allows for structured traversal, resetting when the last card is reached, and ensures randomness while preserving an orderly cycle.

3) Decorator Pattern for Property Improvements

The Decorator Pattern is unnecessary for property improvements, as changes only affect tuition (rent) in a straightforward manner. Since there is only one way to modify a building (adding improvements), maintaining improvement levels and rent values within the Building object itself remains the simplest and most effective solution.

These conclusions remained consistent throughout development, reaffirming our initial design choices.

---

11. Conclusion

Watopoly successfully implements a Waterloo-themed Monopoly game using object-oriented design principles and best coding practices. We focused on creating a structured, maintainable, and modular codebase, ensuring flexibility for future modifications. Each component, such as GameBoard, Player, Controller, and Display, was designed independently, following principles like encapsulation, inheritance, and polymorphism, making debugging and collaboration easier.

Working as a team presented challenges, particularly with Git conflicts and differing coding styles. Some team members preferred detailed comments, while others wrote minimal documentation, requiring us to improve our ability to read and understand each other's code. Additionally, time management proved critical—delays in debugging prevented us from implementing a bot as initially planned, highlighting the importance of setting realistic deadlines and accounting for unexpected issues.

A key lesson was the impact of design choices on maintainability. Our Controller class grew too large, making debugging harder. A top-down approach would have improved task distribution and code structure. Despite challenges, Watopoly is fully functional, capturing the spirit of Monopoly with a Waterloo twist. This project enhanced our technical and teamwork skills, preparing us for future software development roles.
