Watopoly Design Document

Bhavish Goyal and Vyomm Khanna

Introduction

Over the past two weeks, the both of us have worked collectively to write Watopoly, A fun UWaterloo-based simulation of the popular board game *Monopoly* (which Vyomm is unhealthily obsessed with). Approximately 4600 lines of final code later, we have a working, bulletproof implementation that simulates the game of Monopoly using strong Object-Oriented principles taught in CS246 as guidepoints.

While breaking down the core aspects of Monopoly *(Buildings, Properties, Squares, Players, Trades, Auctions, Bank)*, we realised that we would require a centralised *Game-Controller* to coordinate the highly complex relationship between the machineries of the game described above. We modeled our classes based on real facets of the game, giving them access and control to only those features that concern them: a heavy amount of encapsulation. This made us realise we would need to separate our implementation into a Model that stores data and information about core aspects necessary for Watopoly *(squares, buildings, gyms, residences, action-squares, board)*, a View that uses the observer idiom to provide users with an insight into how the flow of the game dynamically influences the model (text-display), and a Controller to centralise core logic (landing on squares, tracking player-property relations, and numerous rule enforcements).

We began working from the ground up, adding small features, considering our design motives, and iterating until we created the highly modular and flexible implementation that allowed us to make changes more easily as our implementation became more comprehensive. The focus of our design is *adaptability*, I'm proud to say that I really feel it's resilient to change. Clean module boundaries, highly-abstracted functionalities, and *polymorphic hierarchies* of related subclasses allowed us to mimic the structure of the real game, while using centralised logic to drive real play forward.

This document outlines how we slowly fleshed out our implementation of Watopoly, right from the initial plan and time we first read the project description! It reflects the changes we made to initial plans, how we adapted to problems, and how adapting to these problems encouraged us to make our code resilient to change. Most importantly, this document highlights the insights we've had the privilege to gain as we engineered through multiple challenges, through the power of design principles and the chance to work together.

Overview

We have structured Watopoly to be highly modular, specifically employing *C++20* for this purpose. We structured our components to be tightly-scoped specifically to their specific domain, and concerned with nothing else whatsoever. We split out implementation into Player, Square (along with its numerous subclasses Building, with further subclasses Residence, Gym, Academic Building; and ActionSquares with further subclasses for SLC, NeedlesHall, Tuition, Co-op-Fee, CollectOSAP etc.) . We handle communication across these specialised components using a mediator, which we call the Game-Contoller that functions as the engine of the game.

The heart of Watopoly: each square facilitating a 'unique' action when landed upon is implemented using an *overridden polymorphic method* which we call *onLand(Player *)*. Every distinct type of square overrides the onLand method to produce a unique action. However, our modular and encapsulated design does not allow each Square, by virtue of its onLand method to directly modify game-state. Instead, each onLand returns a *LandAction enum* describing the type of event that occurred (e.g., PromptPurchase, PayRent, None, SendToTims). This abstraction defers control logic to the GameController, achieving inversion of control and preserving clean domain separation.

We also design a cleverly structured *Square Hierarchy*, where each square encapsulates type-specific data (and nothing else) : whether it is an Academic Building, Residence, Gym, or Action-Square (e.g GoToTims). An instance of this is seen in our implementation of AcademicBuilding, which stores field data like its price, rent-progression, monopoly-block and improvement-count but does **not** attempt to track other buildings or ownership logic — that is delegated to the *GameController*.

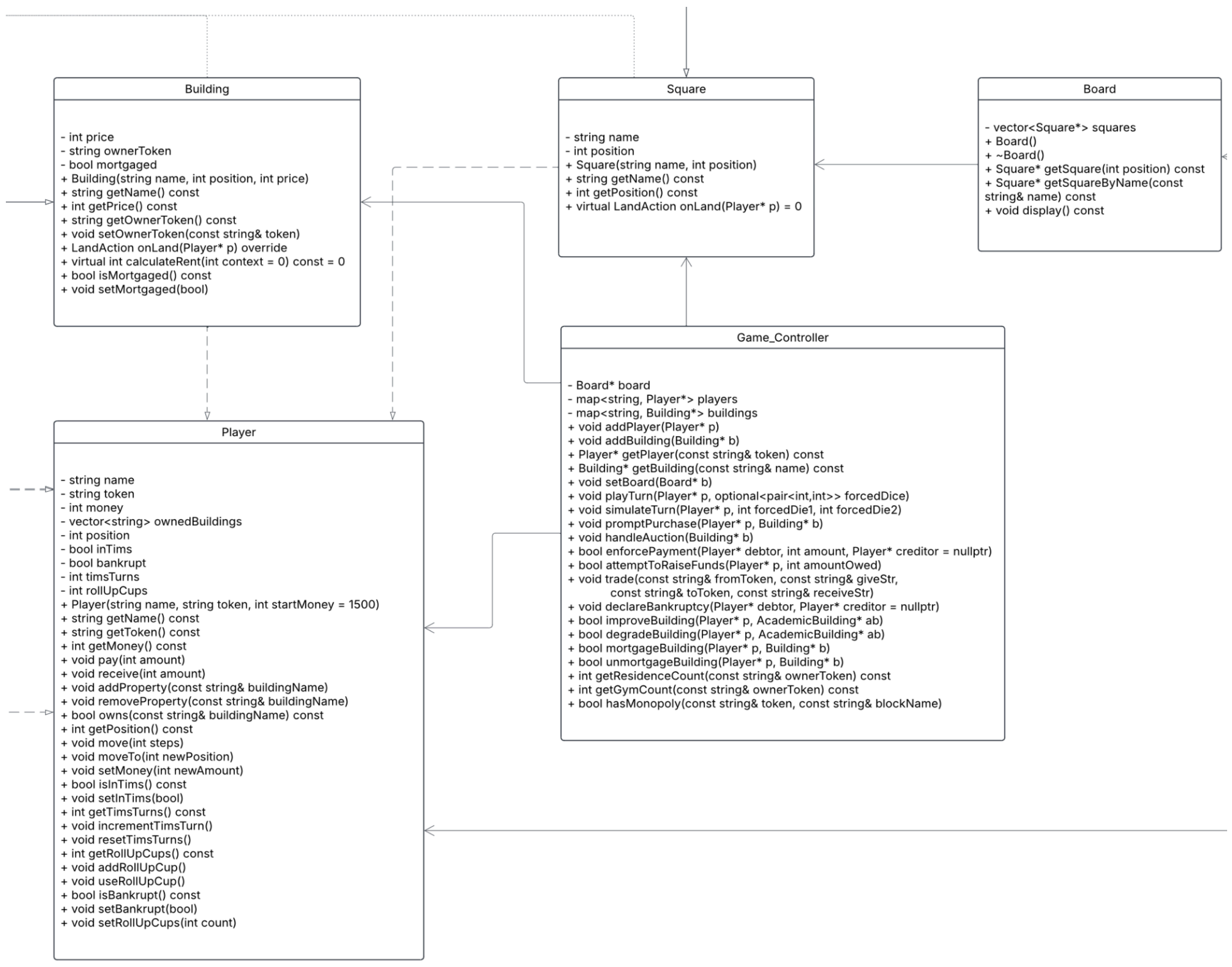
Our Game-Controller acts as a Controller and a *Mediator* for Players, and our tightly-linked square-subclasses which make up the Board. The GameController orchestrates all high-level gameplay, including dice rolls, turn sequencing, player movement, property purchases, auctions, rent calculation, jail behavior, and bankruptcy handling. It acts as the central decision engine of the game. When a player lands on a square, the *GameController interprets the LandAction* and carries out the corresponding logic, such as deducting rent or triggering an auction.

This controller-centric model was chosen to eliminate *entanglement* between Player and Square subclasses. For example, no Square object directly accesses or mutates Player data. Instead, the controller queries player state, makes decisions, and invokes changes via controlled interfaces.

To represent Participants in the game, we use a simple Player class, containing basic fields like name, token, position, ownership, jail-status. Especially important to the working of our Player class is the fact that while the Player has methods to move Position, and change the status of either of its fields by paying or receiving funds, it does not make autonomous decisions. The context behind these decisions is not handled in the Player Class, but rather the Game-Controller which mediates the m*utation of Player-State*. This separation ensures that game logic is never buried inside Player methods, and every gameplay action is transparent at the controller level. For example, even jail escapes — via dice, cups, or payment — are evaluated and processed through GameController.

The board is the pinnacle of the Model in our MVC inspired object-oriented structure. The Board owns and initialises each of the 40 distinct Squares in correct order. Each square is created and inserted into a *std::vector<std::unique_ptr<Square>>*, and indexed by position. The Board simply creates, and holds the model of our game. It functions as a store of data, and distinction among squares. The Board contains methods to retrieve Squares by index or name but does not participate in any sort of gameplay-logic. Its responsibility is limited to layout and initialization. Notably, buildings do not hold references to the Board, which avoids recursive dependencies.

(A snippet of our final UML for an overview of our structure and division-of-responsibilities)



```
Building

- int price
- string ownerToken
- bool mortgaged
+ Building(string name, int position, int price)
+ string getName() const
+ int getPrice() const
+ string getOwnerToken() const
+ void setOwnerToken(const string& token)
+ LandAction onLand(Player* p) override
+ virtual int calculateRent(int context = 0) const = 0
+ bool isMortgaged() const
+ void setMortgaged(bool)
```

```
Square

- string name
- int position
+ Square(string name, int position)
+ string getName() const
+ int getPosition() const
+ virtual LandAction onLand(Player* p) = 0
```

```
Board

- vector<Square*> squares
+ Board()
+ ~Board()
+ Square* getSquare(int position) const
+ Square* getSquareByName(const
string& name) const
+ void display() const
```

```
Game_Controller

- Board* board
- map<string, Player*> players
- map<string, Building*> buildings
+ void addPlayer(Player* p)
+ void addBuilding(Building* b)
+ Player* getPlayer(const string& token) const
+ Building* getBuilding(const string& name) const
+ void setBoard(Board* b)
+ void playTurn(Player* p, optional<pair<int,int>> forcedDice)
+ void simulateTurn(Player* p, int forcedDie1, int forcedDie2)
+ void promptPurchase(Player* p, Building* b)
+ void handleAuction(Building* b)
+ bool enforcePayment(Player* debtor, int amount, Player* creditor = nullptr)
+ bool attemptToRaiseFunds(Player* p, int amountOwed)
+ void trade(const string& fromToken, const string& giveStr,
          const string& toToken, const string& receiveStr)
+ void declareBankruptcy(Player* debtor, Player* creditor = nullptr)
+ bool improveBuilding(Player* p, AcademicBuilding* ab)
+ bool degradeBuilding(Player* p, AcademicBuilding* ab)
+ bool mortgageBuilding(Player* p, Building* b)
+ bool unmortgageBuilding(Player* p, Building* b)
+ int getResidenceCount(const string& ownerToken) const
+ int getGymCount(const string& ownerToken) const
+ bool hasMonopoly(const string& token, const string& blockName)
```

```
Player

- string name
- string token
- int money
- vector<string> ownedBuildings
- int position
- bool inTims
- bool bankrupt
- int timsTurns
- int rollUpCups
+ Player(string name, string token, int startMoney = 1500)
+ string getName() const
+ string getToken() const
+ int getMoney() const
+ void pay(int amount)
+ void receive(int amount)
+ void addProperty(const string& buildingName)
+ void removeProperty(const string& buildingName)
+ bool owns(const string& buildingName) const
+ int getPosition() const
+ void move(int steps)
+ void moveTo(int newPosition)
+ void setMoney(int newAmount)
+ bool isInTims() const
+ void setInTims(bool)
+ int getTimsTurns() const
+ void incrementTimsTurn()
+ void resetTimsTurns()
+ int getRollUpCups() const
+ void addRollUpCup()
+ void useRollUpCup()
+ bool isBankrupt() const
+ void setBankrupt(bool)
+ void setRollUpCups(int count)
```

Design

The Watopoly system was engineered with the explicit goal of achieving *modularity, reusability, and low coupling between components*. The architecture relies heavily on **polymorphism**, **inversion of control**, and **delegated decision-making** to centralize logic in a clean, extensible way. The final UML (included below) reflects a highly decoupled design, with each component isolated by clear interface boundaries and behavioral responsibilities.

1) Decoupling Through LandAction and onLand()

The most foundational design decision was the use of a *LandAction enum* as the return type of the *Square::onLand(Player*)* function. Rather than letting each square mutate game state directly, this method simply describes what should happen. The GameController interprets this result and decides *how* to handle it.:

- It enforces inversion of control, keeping all game flow in one place.
- It allows Square subclasses to focus purely on data and behavior modeling.
  It enables future gameplay rule changes (e.g., adding taxes, alternate rent behavior) without modifying individual square classes.

2) Controller-Centric Gameplay Logic

The GameController implements all core gameplay operations, including:

- Turn simulation (playTurn, simulateTurn)
- Jail logic (tracking turns, cups, rolling doubles, forced pay)
- Property purchases and auctions
- Bankruptcy processing and asset transfers
- Rent deduction using contextual inputs

By centralizing these features, we avoided scattered logic across Player, Square, or Board. This improves maintainability and enables cleaner unit testing of gameplay scenarios.For example, rent is never calculated inside a square. Instead, GameController calls *calculateRent(int context)* on a Building once ownership and dice context are resolved.

3) Polymorphic Rent Calculation

calculateRent(int context) is a pure virtual function defined in Building, and overridden in AcademicBuilding, Gym, and Residence.

- AcademicBuilding uses a static rent table indexed by improvement count.
- Residence rent is based upon the number of residences owned.

- Gym rent is 10 × diceSum if one is owned, 20 × diceSum if both are owned.

This model makes rent rules *data-driven and flexible*, rather than hardcoded. The context parameter allows external computation (like number of owned gyms) to remain controller-side.

Resilience to Change

We purposely designed Watopoly's architecture to allow for the *absorption of rapid new changes,* with minimal effect on other parts of the implementation. Our implementation achieves this through specialised subclasses to model the architecture of the game, but centralized control.

The use of *LandAction* as an intermediate abstraction layer ensures that new square types or modified square behaviors (e.g., new penalty tiles or bonus events) can be added by subclassing Square and returning the appropriate action, without even touching GameController logic! Say we want to extend and make the board larger, for a variant like **Mega Monopoly** which also contains new types of squares. We would define new subclasses of Square, AcademicBuilding, and/or Action-Square to handle the new logic, and simply add it to our Board's indexed and owned collection of squares..

Changes to rent logic are trivially handled by modifying the *calculateRent(int context)* implementations inside individual Building subclasses. For example, if Gym rent rules were altered to depend on total property value or ownership tiers, only Gym-impl.cc would require updates.

The GameController's orchestration model also makes it easy to evolve *turn flow* (e.g., adding new dice effects, turn modifiers, or alternate jail mechanics) by changing controller logic alone, without needing to rework Player or Square.

Trade, auction, and bankruptcy rules, often the most volatile areas, are *encapsulated in pure controller logic*, allowing for complex game rule modifications (e.g., trade taxes, bankruptcy exceptions) with no downstream impact on data classes.

Overall,this architecture greatly enables us to make changes to the program specification with surgical precision, allowing us to iterate and improve as per demand!

Answers to Questions

[Q] 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

[A] Yes, the command design pattern proves suitable for this purpose. We can model each distinct *Chance/Community-Chest* card to be a command object with an overridden function that takes in a Player* as a parameter and executes the unique functionality 'written' on the card. The Chance/Community-Chest holds a list of these objects and randomly selects one based on chance. We can thus use commands to encapsulate operations/functionalities as objects!

[Q] After reading this subsection, would the Observer Pattern be a good pattern to use when implementing a game-board? Why or why not?

For implementing a game-board, not exactly. The observer would be a good idea when one particular object's change of state must dynamically notify other observers, without tight-coupling. Technically we could implement such logic, take for instance a Player buying a Property. This changes Player state and we could potentially use the Observer pattern to update board/game state. However, the *Game-Controller mediator* approach used in our implementation is an astronomically better way to handle model-logic interruptions, especially in a structure liek Watopoly where each class serves only its specific purpose. Applying Observer to the core board logic would complicate control flow and violate the CS 246 design principle of keeping logic centralized and explicit.

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

My answer to this question remains the same from DD1. The improvements described in WATopoly's project guidelines *do not introduce new functionality or behaviors to the classes they modify* (namely, Ownable Academic Buildings); instead, they simply affect a numerical attribute—rent. This change can be efficiently handled by modifying the rent calculation based on a numeric field that tracks the number of improvements applied to a property. Using the Decorator Pattern in this context would *unnecessarily complicate* the design. It would require the creation of multiple decorator subclasses to account for different improvement levels, even though the only change is an incremental numerical adjustment rather than any addition of new behavior. This added complexity offers no real design benefit and would hinder readability and maintainability, making the Decorator Pattern an unsuitable choice for modeling improvements in this case.

Extra Credit Features

To simplify memory management and avoid manual deallocation, we used *std::unique_ptr<Square>* to store all 40 squares on the Board. This ensured that each square had a single clear owner and would be automatically cleaned up when the board was destroyed. This greatly reduced the risk of dangling pointers and memory leaks. However, our separation of our game model (board, squares, buildings) from centralized engine logic helped us assure that our model-logic interactions were valid and stable, reducing the chance for issues with heap-allocated memory

Although we couldn't eliminate all new keywords (e.g., in main.cc due to runtime setup constraints), we verified the entire program with Valgrind and confirmed there were no memory leaks or invalid accesses throughout gameplay. This approach kept the code clean, modern, and compliant with *RAII principles* covered in CS 246.

Final Questions

[Q] What lessons did this project teach you about developing software in teams?

[A] Working on Watopoly as a team taught us so much about building relatively larger-scale software solutions, especially when more than one person is in charge of the success of the project. The most important thing it taught us was the significance of coming up with an architectural model that best fits our goal, and in this case, best fits the unique case of Watopoly. We needed to come up with *(and eventually implement)* a solution that allows us to concretely define our model of the game, the relations between different aspects of this model and the players, and lastly the core-engine logic that drives the game forward. Mutual conversation and brainstorming, along with critically thinking about different approaches taught in the CS246 curriculum showed us the *advantages/disadvantages of each potential approach*.

This way of thinking about the problem, and our approach of working on incrementing a simplest possible solution until we covered all solutions taught us how to *iterate* and work in sprints to develop a finished product. We learned to communicate frequently about dependencies, delay feature development, and use mock behavior to unblock each other during integration. It reinforced how thoughtful modular design allows multiple people to work independently without causing conflicts.

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

If starting over, we would have formalized the use of *interfaces and enums earlier*, especially around gameplay flow and square behavior. Initially, some responsibilities (like purchasing or rent logic) were scattered, and consolidating them later into the *GameController* added extra work. Establishing the *LandAction pattern* and controller-centric design earlier would have streamlined development and testing. We also would have adopted *unique_ptr* and *enum* class more consistently from the start to reduce technical debt. We should probably have thought more about the dependencies between each class in more depth before we started out, as it required us to deviate from our original UML due to (now rather obvious) circular dependencies. However, I quite liked the idea of how we approached an implementation rather naively, and then realised we would need to employ focused object-oriented design principles to achieve our goal. I felt it helped clarify numerous approaches taught in this course, and truly motivated the reasons behind them as compared to just greedily hacking our way towards a working solution.