Watopoly Game - Design Document

**Introduction**

This document provides an overview of the Watopoly game project, describing the overall structure, implementation, design patterns, and object-oriented techniques employed. It also highlights how the design accommodates change, discusses cohesion and coupling, and answers the questions posed in the project specification.

More specifically, our design document outlines the key classes and components necessary for implementing the game. The primary objective of this game is to allow players to buy, trade, and develop properties, aiming to “watopolize” the board and force other players into bankruptcy.

**Overview**

The project is designed as an object-oriented C++ program, consisting of several classes representing the game's components, such as players, properties, and the game board. The primary classes include Game, Player, Property, Non property, Square, and their subclasses.

1.Key Components

The game's primary components are the board, players, and properties. The board is a collection of squares, some of which represent properties that can be owned by players. The players interact with the board, moving around, buying and selling properties, and improving them to increase their value.

2. Classes in the project

2.1. Game

The Game class serves as the main control center for the game, managing the board and players, and handling player interactions.

Attributes:

* std::vector<Player\*> players
* std::vector<Square\*> board
* Player\* currPlayer

Methods:

* displayAll(): Display all properties for all players.
* displayAssets(): Display all properties for the current player.

2.2. Player

The Player class represents a player in the game, holding information about their properties and financial situation.

Attributes:

* std::string name
* char avatar
* size\_t balance
* size\_t assets
* Square\* currSquare
* std::vector<Property\*> ownedProperties
* bool isBankrupt
* int turnsStuck
* int timsCups

Methods:

* getInfo(): Return a PlayerInfo object with the player's information.
* owns(): Check if the player owns a specific property.
* declareBankruptcy(): Player declares bankruptcy.
* buyImprovement(): Buy an improvement for a property.
* sellImprovement(): Sell an improvement from a property.
* mortgage(): Mortgage a property.
* unmortgage(): Unmortgage a property.
* makePayment(): Make a payment of a specified amount.
* moveTo(): Move the player to a new square on the board.
* addFunds(): Add funds to the player's balance.
* addProperty(): Add a property to the player's list of owned properties.
* removeProperty(): Remove a property from the player's list of owned properties.
* buyProperty(): Buy a property and add it to the player's list of owned properties.
* makeStuck(): Make the player stuck (e.g., due to being in jail).
* makeUnstuck(): Free the player from being stuck.
* passTurnStuck(): Process the player's turn while they are stuck.
* roll(): Roll the dice and return the result.
* trade(): Trade properties or money with another player.
* addTimsCup(): Add a Tim's cup to the player's inventory.
* useTimsCup(): Use a Tim's cup to get out of a stuck situation.

2.3. Square

The Square class represents a square on the game board.

Attributes:

* std::string name

Methods:

* getInfo(): Return a SquareInfo object with the square's information.

2.4. Property (inherits from Square)

The Property class represents a property on the game board that can be owned by a player.

Attributes:

* Group group
* Player\* owner
* bool isMortgaged
* int numImprove

Methods:

* setMortgaged(): Set the property as mortgaged.
* setUnmortgaged(): Set the property as unmortgaged.
* improve(): Improve the property (abstract).
* unimprove(): Remove an improvement from the property (abstract).
* getVisitFee(): Calculate and return the visit fee for other players (abstract).
* getInfo(): Return a SquareInfo object with the property's information (abstract).
* setOwner(): Set a new owner for the property.

2.5. Group

The Group class represents a group of properties (e.g., color group in Watopoly).

Attributes:

* std::vector<Property\*> properties
* std::string groupName

Methods:

* addProperty(): Add a property to the group.
* removeProperty(): Remove a property from the group.
* allOwnedBy(): Check if all properties in the group are owned by the same player.
* getInfo(): Return a GroupInfo object with the group's information.

2.6. Info Classes

These classes represent the information objects returned by the getInfo() methods of the main classes.

2.6.1. PlayerInfo

Attributes:

* std::string name
* char avatar
* size\_t balance
* size\_t assets
* std::string currSquareName
* std::vectorstd::string ownedPropertyNames
* bool isBankrupt
* int turnsStuck
* int timsCups

2.6.2. SquareInfo

Attributes:

* std::string name

2.6.3. PropertyInfo (inherits from SquareInfo)

Attributes:

* GroupInfo groupInfo
* std::string ownerName
* bool isMortgaged
* int numImprove

2.6.4. GroupInfo

Attributes:

* std::vectorstd::string propertyNames
* std::string groupName

3. User Interface

The user interface will be text-based, with players interacting with the game through a series of prompts and input commands. The user interface will handle displaying the current state of the game, including player positions, properties, and financial information, as well as processing user commands and providing appropriate feedback.

4.Flow of the Game

The game will follow a turn-based structure, with each player taking turns in a clockwise order. On each turn, the player can take a variety of actions, such as buying and selling properties, trading with other players, improving properties, and paying fees for visiting other players' properties. The game continues until all but one player has declared bankruptcy, with the remaining player being declared the winner.

5.Future Enhancements

To improve the game and add more complexity, future enhancements may include the introduction of additional property types, such as utilities and railways, as well as special squares, like Community Chest and Chance, that can introduce random events and opportunities for players. Additionally, the game could be expanded to include a graphical user interface and even networked multiplayer functionality.

**Updated UML**

A detailed UML diagram showcasing the relationships and structure of the classes can be found in the project repository.

**Design**

Below is a discussion of how the design addresses flexibility and accommodates change:

*Encapsulation:* By encapsulating data and behavior within classes, the design ensures that any changes within a class are less likely to impact other parts of the system. For example, the Player and Property classes have private data members and provide public getter and setter methods to access and modify their internal state. This encapsulation allows for easier modification of class internals without affecting external code that uses the classes.

*Inheritance and Polymorphism:* The design leverages inheritance and polymorphism in the Square and Property classes, allowing for easy extension of new types of squares or properties without having to modify the base classes or other parts of the system. This enables the design to easily accommodate changes in the game rules, properties, or squares.

*Modular Design:* The design is structured in a modular fashion, with classes and functions having a single responsibility. This ensures that each module can be modified or replaced independently of the others, thus improving maintainability and accommodating change more easily.

*Observer Pattern:* To handle game state changes and events, the design can make use of the Observer pattern. This pattern allows multiple objects to be notified when a particular object's state changes. In the context of the game, the Game class can act as an observer for various events, such as player moves or property acquisitions, and update the game state accordingly. This decouples the game logic from the classes responsible for the game's state and makes it easier to handle changes in the game mechanics or add new features.

*Factory Method Pattern:* The design could benefit from the Factory Method pattern in creating objects such as Square or Property instances. This would allow for greater flexibility in creating new types of squares or properties without having to modify the main game logic. By encapsulating the object creation process within a factory method, it is easier to accommodate changes in the object hierarchy or introduce new object types.

In summary, the design document takes into consideration the principles of object-oriented design and various design patterns to ensure flexibility and accommodate future changes. By following these principles and patterns, the design becomes more maintainable and extensible, allowing for easier adaptation to new requirements or modifications.

**Resilience to Change**

In this section, we will discuss some of the design aspects that contribute to the resilience to change in the project:

*Encapsulation:* By encapsulating data and behavior within classes, we ensure that modifications to a specific part of the system have minimal impact on other components. This approach allows developers to make changes to the implementation or functionality of individual classes without affecting the rest of the system. Furthermore, using getter and setter methods to access and modify private data members helps maintain the integrity of the data and provides a single point for any necessary changes.

*Inheritance and Polymorphism:* The use of inheritance and polymorphism in classes like Square and Property allows for the easy extension of new types of squares or properties without modifying existing code. For instance, if a new type of property is introduced, we can create a new subclass that inherits from the base Property class, providing the necessary implementation for the new type while maintaining compatibility with the existing game logic.

*Modular Design:* A modular design approach ensures that each class or function has a single responsibility. By keeping modules small and focused, we reduce the complexity of the system, making it easier to maintain and update. This approach facilitates changes, as developers can modify or replace individual modules without affecting other parts of the system.

*Loose Coupling:* The design aims to minimize dependencies between classes and modules, reducing the likelihood that changes in one part of the system will affect others. For example, using interfaces and abstraction in class design can help decouple components, making it easier to swap out specific implementations or add new ones without breaking existing code.

*Design Patterns:* The implementation of various design patterns contributes to the project's resilience to change. For instance, the Observer pattern can be used to handle game state changes and events, decoupling the game logic from the classes responsible for the game's state. The Factory Method pattern allows for greater flexibility in creating new types of Square or Property instances without modifying the main game logic. Incorporating these design patterns ensures that the project can adapt to new requirements or modifications more easily.

*Code Reusability:* By creating reusable and generic components, we make it easier to adapt the project to changes in the requirements. For example, utility functions and classes can be designed in a way that allows them to be used in various parts of the project, reducing the amount of code duplication, and making it easier to modify the system when needed.

In conclusion, the Watopoly game project's design, which follows object-oriented design principles and patterns, ensures a high level of resilience to change. The project can accommodate future changes in the game rules, components, or requirements by building upon its flexible, maintainable, and extensible foundation.

**Answers to Questions**

The answers to the questions posed in the project specification are provided below:

Q1: 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?

*A1: (Please provide the answer)*

Q2: What would you have done differently if you had the chance to start over?

A2: *(Please provide the answer)*

**Extra Credit Features**

(Describe any extra credit features implemented in the project, the challenges faced, and the solutions employed.)

**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?

(*Provide the answer*)

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

(*Provide the answer*)

**Conclusion**

In conclusion, the Watopoly game project demonstrates the power of effective object-oriented design principles, patterns, and techniques in creating a robust and adaptable software system. Through encapsulation, inheritance, polymorphism, modular design, loose coupling, and code reusability, the project achieves a level of flexibility that allows it to accommodate changes with minimal impact on the overall system.

This project showcases how the use of design patterns like the Observer pattern, the Factory Method pattern, and others enhances the ability to adapt to changing requirements, making it easier for developers to extend or modify the system as needed.

The project serves as an example of how the application of solid design principles can lead to a maintainable, extensible, and resilient software system that can withstand the test of time and evolving requirements. As developers, we should always strive to follow these principles to ensure that our software remains adaptable and maintainable in the face of change.

Finally, the Watopoly game project teaches valuable lessons about the challenges of developing software, whether individually or in a team. It highlights the importance of clear communication, meticulous planning, and efficient time management to ensure the successful completion of complex projects. Reflecting on the experiences gained from this project, we can apply these insights to future endeavours, becoming better developers and team members as we continue to learn and grow.