Watopoly Project Plan of Attack

Team Members:

- *Yashila Barnwal*
- *Dhruv Kumar*
- *Lakshit Luhadia*

Initial Plan Date: March 19, 2025

1. Project Breakdown

The Watopoly project will be broken down into the following modules, based on the provided UML class diagram and project description. The order reflects a logical progression, starting with core components and moving towards more complex features.

- **Phase 1: Core Components (March 19 - March 24)**
  - To build the game, we first need to figure out its main parts: the Board, the Player, and the Property. The Board is where everything happens. It's the playing surface that holds all the squares, like properties, utilities, and special spots. The Player is just that someone playing the game. They have money, a position on the board, and own different properties. Then there's the Property, which are things players can buy, mortgage, or upgrade. They're a big part of the game since they decide how money moves around.
  - We also have to set up how they interact. Players can own more than one property, which affects their money and strategy. The Board decides where players move and handles what happens when they land on different squares. And running the whole thing is the Game class, making sure everything works together properly. These connections are super important because they make the game actually playable things like moving around, buying properties, and paying rent wouldn't work without them.

- **Phase 2: Property Management and Actions (March 24 - March 27)**
  - Board and Squares: Implementation of the Board class and the various Square subclasses (Academic, Residence, Gym, Non-Property squares). This includes the basic board structure and the performAction methods for each square type.
  - Player: Implementation of the Player class, including attributes like money, position, properties, and methods for movement, managing money, and handling properties.
  - Dice: Implementation of the Dice class for rolling dice and determining the outcome.
  - Game: Implementation of the Game class, which manages the overall game flow, players, and board. This includes the roll() and next() methods.

  - Command Interpreter: Initial implementation of the command interpreter to handle basic commands like roll, next, and player setup.
  - Property: Implementation of the Property class and its derived classes (Academic, Residence, Gym), including ownership, mortgage status, and methods for calculating rent/fees.
  - Improvements: Implementation of the improvement system for Academic buildings, including buying and selling improvements.
  - Mortgage/Unmortgage: Implementation of the mortgage and unmortgage commands and the associated logic in the Property class.
  - Bankruptcy: Implementation of the bankrupt command and the logic for handling player bankruptcy, including asset transfer or property return to the bank.
  - Auctions: Implementation of the auction system for properties, as described in the specification.

- **Phase 3: Advanced Features and Game Logic (March 27 - March 29)**
  - Trading: Implementation of the trade command and the associated logic for handling property and money trades between players.
  - Special Squares: Implementation of the logic for SLC, Needles Hall, Go to Tims, and DC Tims Line squares, including the random movement and special rules.
  - Roll Up the Rim Cups: Implementation of the Roll Up the Rim cup mechanism, including awarding cups and using them to get out of DC Tims Line.
  - Saving/Loading: Implementation of the save command and the -load command-line option to save and load game states.

- **Phase 4: Testing, Refinement, and Documentation (March 29 - March 31)**
  - Testing: Thorough testing of all game features, including edge cases and error handling.
  - Refinement: Code cleanup, optimization, and addressing any bugs or issues found during testing.
  - Documentation: Finalizing all documentation, including code comments and a user manual.

2. Task Assignment

To ensure efficient development, tasks will be divided among the team members as follows:

- **Yashila Barnwal**

  Dice, Player, Subject, Observer, NonProperty, Game, Text Display, Graphical Display

- **Dhruv Kumar**

  Property, Square, Residence, Academic, Gym, Game, Text Display, Graphical Display

- **Lakshit Luhadia**

  Graphical Display, TextDisplay, Game, Board, GoToTimsSquare, ChanceSquare, FeesSquare, OSAPSquare, TimsLineSquare

3. Estimated Completion Dates

The following is a realistic schedule for completing each phase:

- Phase 1: March 19 - March 22
- Phase 2: March 24 - March 27
- Phase 3: March 27 - March 29
- Phase 4: March 29 - March 31

4. Game Structure

The game will start with main.cc which will accept the initial command line arguments expected when starting the game, and the subsequent commands being entered by the players.

Game
Main.cc will call the Game class which is the main driver of the Watopoly game, responsible for controlling gameplay, player turns, and the board. It uses the Observer pattern to keep the UI updated (it is a concrete subject class). It owns the board and dice and provides functions to handle everything from rolling dice to trading and improving properties.

TextDisplay
The TextDisplay class is a concrete observer and contains a unique pointer to Game class. Every time a player takes a turn, the Game will notify TextDisplay, which then will print the updated board. It follows the Observer Pattern to cleanly separate the game logic from the display logic.

Board
The Game Class owns Board, which manages the squares and players in the game. It keeps track of whose turn it is and provides functions to add or remove players, access squares, move to the next player, and print the current state of the board.

Square
The square class describes each square of the Watopoly board (all 40 squares), and contains its name and position, and a boolean for if it is a property type.

Dice
This class is owned by Game and is called to roll dice for each player during their turn and randomly generates numbers for 2 dice from 1 to 6. It also has an option for testing mode, during which numbers for the dice can be inputted.

Player
The player class is owned by Board and describes the possessions of the player in the game, like their properties, money, number of RimCups and stores their position on the board and if they are in the TimsLine.

Property
This is a Square class and this stores the cost and mortgage value of the ownable property and what player owns it. The Gym, Residence and Academic class is a Property class.
They each contain a monopoly checker, to check if each player owns all the properties of the same block. The Player class contains the checker for Residence to see how many of those a player has.

NonProperty
This is a Square class for all the non-ownable squares and the classes GoToTimsSquare, ChanceSquare, FeesSquare, OSAPSquare and TimsLineSquare all inherit from it with each class containing their specific consequence that happens to a player which happens from the performAction function they each contain since that overwrites the pure virtual function performAction in Square.

GraphicalDisplay
The GraphicalDisplay follows the same principles as the TextDisplay and is executed via the Xwindow. Additionally, the game will by default create a TextDisplay unless '-graphics' is passed as a command line argument by the user, which is when a GraphicalDisplay is created.

5. Answers to Questions in the Project Specification

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

The Observer Pattern is a great way to set up the game board. The Game class is like the main controller. It is the one in charge. The displays, whether graphical or text-based, are the observers. Anytime something changes in the game, like a player moving or buying a property, the Game class lets all the observers know. This keeps everything up to date, so the board always shows the right information. This method has some big benefits. First, it keeps the display stuff separate from the actual game logic, which makes the code way easier to manage. Second, it makes the game more flexible to new observers.

Question 2: 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 represent SLC and Needles Hall, similar to Chance or Community Chest in Monopoly, we can use the Factory Method pattern. First, we define a general class with a method like performAction(Player &player), which will handle what happens when a player draws. Then, we create specific types like SLC and NeedlesHall, each with its own effects like moving a player to a different spot on the board or adjusting their money. To manage, we use something that generates based on predefined probabilities. This setup has a few advantages. It keeps the creation logic separate from the rest of the game.

Question 3: Is the Decorator Pattern a good pattern to use when implementing Improvements? Why or why not?

The Decorator Pattern isn’t a good fit for handling Improvements in Watopoly because it would add unnecessary complexity. Improvements don’t follow a clear formula for tuition increases, and each level directly sets a new fee instead of building on the previous one. Using Decorator would require wrapping properties in multiple layers, making the code harder to manage. Plus, game rules like selling improvements in reverse order would make decorators even more complicated.