1. Plan of Attack

1.1 Group Members, Roles, and Responsibilities

Date	Deliverables
18th March	**All:** Select our project and draft a super simple UML diagram. Understand various features and functions that should be implemented.
19th March	**Arav**: Start implementing Board class – setup 40 squares (array/vector), initialize square names. **Austin**: Set up initial Player class – placeholders for name, piece, money, position. **Sachit**: Create base Buildings class and derive PropertyBuilding.
20th March	**Arav**: Implement text-based display method for board (print all squares). **Austin**: Add logic for Tim's Cups, money tracking. **Sachit**: Add Residences, Gyms, and NonPropertyBuildings classes.
21st March	Finalize draft versions of Board, Player, and Building classes. Integrate classes minimally to verify compile. **All**: Sync code, prep UML diagram, and divide initial responsibilities.
22nd March	**All**: Link Board squares to building instances. **Arav & Austin**: Begin command parser – implement minimal commands (e.g., roll).

	All: Finalize UML, submit umuml.pdf and plan.pdf.
23rd March	**Sachit**: Begin coding auction logic (property refusal, bankruptcy triggers). **Austin**: Implement buy/sell improvements, calculate tuition. **Arav**: Logic for Coop Fee, Goose Nesting, Tuition square.
24th March	**Arav**: Implement Go To Tims square, tie into movePlayer(). **Austin**: Finish mortgage/unmortgage with 10% fee, write tests. **All**: Review and polish for DD1. Submit all files.
25th March	**Sachit**: Finalize auction interactions, bid logic. **Austin**: Implement improvement selling, update tuition post-sale. **Arav**: Start SLC random movement function.
26th March	**Sachit & Arav**: Finish SLC and Needles Hall logic, add Roll Up the Rim (1% chance, 4 max). **Austin**: Add Tim's Cup usage logic to Player.
27th March	**Austin**: Implement DC Tims Line escape logic (roll doubles, pay $50, use cup). Handle 3rd turn rules.
28th March	**Austin & Sachit**: Implement full trade command. Validate property status (no improvements). User prompts for accept/reject.
29th March	**Arav**: Implement bankruptcy – asset transfer or auction. Endgame check (1 player left).

	Arav: Implement save and load commands. **Austin & Sachit**: Verify save/load includes all state info.
30th March	**All**: Full playtest (2-player game, save/reload, boundary cases). **Austin**: Final money validation, edge case fixes. **Arav**: Add unknown command handling, edge case logs.
31st March	Final code freeze. Submit project code with Makefile (watopoly). Submit uml-final.pdf and design.pdf. Prepare for demo.

2. Answers to Watopoly Spec Questions

2.1 Using Observer Pattern

Yes, we could use the Observer Pattern (or a close variant) for the game board to maintain a clean separation between the board's underlying data (the "subject") and the view(s) (the "observers"). Each time the board's state changes (for instance, when a player moves, a property changes ownership, or someone builds an improvement), the board could notify the display class which acts as the observer, prompting it to update.

1. **Why Observer Pattern is helpful**

    - It may seem that Observer is overkill, but it could also be beneficial if we want to keep our code flexible and well-organized. For example, we plan on having the display class get notified whenever we have a change to the board. If we wanted to add future modifications or features that rely on the board's active information, it is much easier to do so with the observer pattern as it is scalable.
    - If we were to expand the project to a GUI or multiple concurrent displays, Observer ensures that these displays remain in sync. The observer can register itself with the board (subject), and whenever the state of the board (i.e. of each square) changes, they receive a notification to refresh.
    - We could also follow a more direct route by calling a command that displays the update board for example each time a turn completes. However, that approach is somewhat "manual," and we lose the elegance and modularity that comes from letting multiple observers decide independently how to handle notifications.

2. **Trade-offs**

    - **Complexity**: Adding an Observer may be overly complex. However, it allows for future scalability: If we decide to add new features that rely on the board's information and needs to be updated regularly as it changes, we will simply add it as an observer. Even if new devs were to be added to the team in the future, understanding the project becomes super simple.

    - **Performance**: For a console-based game, performance overhead is minimal, as re-rendering text is not that resource-intensive. In the future, if we want additional features to our project such as For large-scale or real-time UIs, Observer Pattern can actually improve efficiency by letting

each observer decide what minimal updates are needed rather than forcibly redrawing everything.

In summary, **yes**, the Observer Pattern is a valid approach. The observer pattern will help keep responsibilities separated between the group and also us to implement more advanced features, if need be, in the future.

2.2 Treating SLC & Needles Hall like “Chance/Community Chest” Cards

Answer:

No, implementing a specialized design pattern is too complicated for the outcomes we need in SLC and Needles Hall.

1. **Reasons to Consider a Pattern**
    - A “card-like” approach (similar to Monopoly Chance/Community Chest) means that we would have to create a series of distinct “action objects,” each having its own execute() method. That would let us easily add new SLC/Needles Hall outcomes or shuffle them.
    - This approach can be good if we anticipate expansions (e.g., new events, more squares, drastically different outcomes).
2. **Why It May Be Overkill**
    - **Limited, Predictable Outcomes**: The project states that SLC and Needles Hall have only a small set of possible moves or money modifications. They remain largely consistent throughout the game.
    - **Extra Complexity**: Implementing a separate class or structure for each outcome requires additional code and more advanced logic (like a deck, a random draw, etc.). For only a handful of random results, a single function with basic if-statements or a switch-case might be simpler and just as effective.

- **Lack of Variation**: If all we do is choose an integer offset for movement or a money gain/loss from a fixed distribution, we're not layering unique behaviors that would strongly benefit from a pattern.

3. **Recommended Simpler Approach**

    - We can store the probabilities and associated outcomes in a data structure like a vector.

    - We generate a random number, check which range it falls into, and apply the corresponding effect. This direct method is straightforward to code and easy to read.

    - If a 1% chance triggers a Tim's Cup reward, that can be handled with a single condition.

Hence, while we could design an elegant "deck-of-cards" system, we think that we don't need a specialized pattern here because the uniform nature and limited number of outcomes do not justify the complexity. The essential logic can be done in a compact, procedural style without major drawbacks.

2.3 Using the Decorator Pattern for Improvements

We can try the Decorator Pattern to model building improvements, but if that proves unnecessary, we can revert to direct logic (an integer count of improvements) for simplicity and efficiency.

1. **Pros of decorator pattern:**

    - Allows us to treat each improvement (bathroom or cafeteria) as a distinct "decorator," then we can transform the base AcademicBuilding object to reflect the new rent or tuition fees after each improvement is added.

    - This can be better suited if we want to dynamically add or remove features without altering the core class.

2. **Cons:**

- **Overkill**: We already know that we must buy the four bathrooms first and then the cafeteria in order. A decorator pattern is most useful when features can be added in various different orders.

- **Simplicity**: A single integer is enough to know exactly how many upgrades a building has. Checking that integer against a table of costs or rent multipliers is very straightforward to implement.

In conclusion, **yes**, a Decorator Pattern would be a good option if we want a more theoretically flexible structure or to showcase design pattern usage. However, if we believe it to be inefficient or burdensome to manage for these linear improvements, we will go for a simpler design that will likely utilize simply keeping track of improvements using integers.

3. High-level Overview of a round of Watopoly:

1. Board will choose who's turn it is.

2. Board will roll the dice object.

3. Board moves the player to the correct building and removes the player from the old building.

4. Board will notify the displayObserver.

5. Board will call the event() function of the new building it landed on. The event() function should also take in the player that is moving so it knows who to apply the event to.

6. The building's event() will then carry out the rest of the turn, whether it is prompting the buy/tax/etc.

    If the player chooses not to buy (auction), or there is some other special property that requires multiple players, call an edge case method in the board to handle it.

    We will add default calculation methods in the abstract base classes so we don't have to rewrite for similar buildings.

7. After event() ends, the board takes control again and prompts the player to improve properties, end turn, etc.

8. If the player prompts their turn to end, it starts a new turn.