Plan of Attack

Introduction:

Our primary goal of this CS246 final project is to implement a fully functioning game of Waterloo-based Monopoly, a.k.a WATopoly! We *(Bhavish and Vyomm)* aim to do this symbiotically, supplementing each other while employing OOPs principles from course notes and lectures wherever applicable to modularize, and simplify our implementation process. One major cornerstone of our plan of attack, motivated by advice in the project guidelines is to work incrementally. We aim to start small, implement, test, and iterate this process until we build up to a showcase-worthy project. While looking at the bigger picture, we plan to emphasize abstraction, encapsulation, low-coupling, and high-cohesion: so that each of our classes, no matter how small they be, handle a specific and integral functionality.

Breakdown:

1. Ideation: *TBD 22nd March*

First, we will collectively finalise how we plan to structure the project! This involves breaking down the project, ie. WATopoly as a whole, into smaller, more actionable, and implementable tasks. This breaking-down process entails us to come up with various classes to represent different aspects of the game, how each of those classes links up with each other, and what functionality each class contains. We plan to achieve this through mutual discussion, sketching out our ideas, and solidifying our plan in the form of a formal UML diagram which we will submit before Due Date 1

2. Implementation of Board and Player Attributes: *TBD 24th March*

Once our UML diagram is finalized and our initial phase of ideation is more or less complete (always prone to refinement) , we focus on implementing the basic building blocks, starting really small. *Vyomm* will work implementing the basic features of what makes up the board. We plan to implement the Board as an array or a container of basic ‘property’ / ‘cell’ objects. The latter constitutes a base class, and *Vyomm* will use inheritance and polymorphism principles to further differentiate the cells into Ownable, Non-Ownable, Academic, Non-Academic, and miscellaneous squares which can be landed on. *Vyomm* will implement each of these, accompanied with necessary attributes

in the forms of fields and member functions until we have a board that is more-or-less complete.
While *Vyomm* is working on the board, *Bhavish* will start work in parallel on the Player, and its various functionalities that contribute to the flow and movement of the game. The Player Class models the player's identity, assets, current state, and how it's linked to other players and various properties on the board. Note, that *Bhavish* will work on the implementation of the Player Class as described above, but will delay the last feature: how the player is linked to the board, until the board is ready so that we can take a step back, test out that our functionalities are working, and proceed. Bhavish's implementation of the Player class will involve high cohesion, as it only manages its own data and decisions. However, there are still numerous player-fields to consider:

- Name and Unique Identifier on Board
- Tradable assets owned (monetary and property)
- Where they are situated on the board
- Number Roll Up the Rim cups they possess.
- Some Boolean flag to determine whether they are stuck in the Tim's line, and another variable to track how long

After these fields have been populated chronologically, *Bhavish* will then work on the player abilities which include but are not limited to the following: 1) moving along the board 2) responding to landing on a buyable/non-buyable square 3) offering or reacting to trades 4) displaying their assets, and 5) declaring bankruptcy :((

The game-flow logic will be addressed later and through a class that acts as a controller and has the board, collection of players, etc. as fields. We are delaying our work on the Controller class until after the Board and Player(s) have been completed so that we can completely test out each functionality we work on with absolute thoroughness, avoiding the notion of working under any assumptions.

3. Main Game Controller and Player/Board Interactions *TBD 26th March*

Now after the basic mechanical building blocks of WATopoly are ready, we will work on how to *move* them to constitute the flow of the game. This will be done through the implementation of the Controller/Watopoly class we spoke about earlier. A key distinction here is that both Vyomm and Bhavish will be working on the same class here (so we will probably meet at a study room physically and bring a lot of food + energy drinks with us) *Bhavish* will work on implementing the turn-management, and representation of player-state. We feel like this is a good idea since he worked very closely on the Player class and should have comparative expertise with its implementation. Concurrently, *Vyomm* will work on formulating and implementing the logic for buying/selling/mortgaging properties, and charging rent based on property-improvement. The implementation of each individual functionality will be accompanied by testing, and continuous integration into the parts of our WATopoly implementation that are ready. Some form of integration-test variation will be conducted at the end of each major stage.

Then, whichever one of us is done implementing our work for this stage earlier, will start work on the command-line processor to interpret player-commands and translate them into functionalities that we've already implemented.
The key focus of this stage is implementing and verifying interactions between the players and board elements, while storing and representing these instructions in our main controller class. Once this stage is complete, we will have a working skeleton of the game, subject to improvements (and handling edge-cases) which we can refine, and further iterate upon to bulletproof our implementation.

4. Miscellaneous Functionalities, Graphical Display, and Load/Save Game: *TBD 28th March*

Lastly, both of us will work collectively to implement Auctioning, Trading, and other miscellaneous events like Roll up the Rim, and lastly being sent to (and staying in) the DC Tim's line. The exact split between us has not been determined as of yet, but *Vyomm* will probably focus on the chance-based roll up the rim and the DC Tim's line functionality while *Bhavish* handles the Auctioning and Trading. These divisions have been chosen due to our comparative advantages in Board vs Player implementation, as seen in the earlier phases. After this is done, we will finally work on the Load/Save Game and Graphical Display using the XWindow/X11 Class to wrap up our project. The last phase of our implementation will be followed by a series of integration testing where we test each of the functionalities we've implemented, and the flow of control through the natural execution of a game.

Question: 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?

Yes, based on our CS246 course notes we could implement a number of Decorator Classes that represent functionalities such as those observed in Chance/Community Chest cards. This collection of 'decorator effect' classes implements inheritance using a virtual-base class, and a collection of the specialised 'effect' classes, one of which can be chosen randomly and applied to the player who lands on said square to mimic the randomness of choosing a card.

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

Well, we tend to use Observer logic when it comes to one class, e.g the Board to inform another class when its state has been changed. Based on our UML, this functionality belongs to the Controller class. The Board simply acts as a representation for properties, which is dynamic and changes as the game progresses. Any of these changes do not need to 'inform' another class, as that logic will be implemented in the Controller class which contains the Board, and the Collection of players as parameters and can thus automatically keep track of everything going on at a higher level.

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

The improvements described in WATopoly's project guidelines do not change or add any additional behaviour to the classes they act upon (Ownable Academic Buildings). They simply increase the rent, which can be implemented through a simple change in said object's rent field. The number of applied improvements can also be tracked using a simple numerical field.

Following the Decorator Pattern here would mean we would have to implement more decorator subclasses, even though the behaviour of the subclass is changed only numerically, with no new functionality! This would greatly add to code-complexity and is not required here, at least in our opinion.