Project Breakdown

Order	Task	Estimated Completion Date	Assignee(s)
1	Design + Setup: a. Review UML together, identify integration points and clarify class responsibilities. b. Set up Git and agree on coding style. c. Create stub files for all classes in the UML to begin building interfaces and skeletons.	March 25	Leon Domagalski, Sheldon Lewis
2	Board + Squares: a. Implement Board class, including: i. Loading squares from a file (e.g., squares.txt) ii. movePlayer() logic to handle movement and square landing b. Implement base Square class and key concrete squares: i. AcademicBuilding ii. Residence iii. SLC, NH (event squares)	March 27	Sheldon Lewis
2	Player + Roll + Tuition Logic: a. Implement Player class: i. Attributes like name, location, money, and owned properties ii. Movement (delegating to Board) b. Implement Roll mechanics, integrating with Dice or Die classes c. Implement tuition logic for AcademicBuilding	March 27	Leon Domagalski
3	Improvements + Buy/Sell: a. Implement Improvement logic in AcademicBuilding b. Add rent calculation depending on number of improvements c. Implement buying, selling, and mortgaging	March 29	Sheldon Lewis

	properties		
3	Game Loop + Observer Integration: a. Build the Controller or main game loop b. Integrate Observer pattern: i. Board and text/graphical displays observing changes in Player and Square	March 29	Leon Domagalski
4	Integration + Testing: a. Merge branches and resolve integration bugs b. Test game flow: turn execution, property transactions, jail logic, bankruptcy c. Implement TextDisplay and test minimal CLI interaction	March 31	Leon Domagalski, Sheldon Lewis

Questions

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

Yes, the Observer Pattern would be a good pattern when implementing a board game. With a board game that has a frontend and a backend, we have to create a way for those two to communicate with each other, as the information displayed on the frontend will depend on the information that is stored in the backend. We want the information that is being displayed to the user to get dynamically updated with the changes that the user creates. Therefore we want the frontend to observe the backend, and the backend to notify the frontend of any changes that are happening. In our case, monopoly, the board would be observing the backend, players and squares, and the players and squares would notify the board of any changes that are happening to them. That would include things such as the player rolling the dice and therefore changing the square. The curretSquare attribute of a Player object would change, but now the Player object has to notify the frontend that it has moved, so that we can display that information to the user.

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?

A suitable design pattern for SLC and Needles Hall, if we wanted to model it more closely to chance and Community Chest cards, would be the Factory Design Pattern. In monopoly we have that Community Chest and Chance cards are randomized and have different effects on the person that drew them. First we'd have to create a base class for a card. Then we could create concrete implementations of different Chance and Community Chest cards. At last we would have to create a factory class that randomly picks one of the cards and returns it to the player, so that the player is affected by the card that has been randomly "drawn" for them. The game doesn't have to know what effects the card had on the player, each event handles itself independently.

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

No, the Decorator Pattern would not be a good fit for implementing Improvements. In Monopoly, Improvements are added in a very structured and countable way — you can have up to 5 on a property, and they directly affect things like rent and mortgage value. If we used the Decorator Pattern, each Improvement would be its own object wrapping around the property, which would make it harder to keep track of how many there are, or to remove them if needed. It also doesn't reflect how the game logic works — for example, you wouldn't want to have to unwrap several decorator layers just to know how many Improvements a property has.

A simpler and more effective approach would be to store the number of Improvements as an integer inside the AcademicBuilding class. That way, the property knows how many Improvements it has, and we can easily calculate rent, check conditions, and update the display. So while the Decorator Pattern allows dynamic behavior, it doesn't suit the clear, rule-based way Improvements work in Monopoly.