In our Java implementation of Deadwood, we primarily used the Model-View-Controller (MVC) design pattern to separate concerns and create a structured, readable and scalable code. The Model represents the game logic, including player states, room configurations, and game mechanics. This was mostly completed in a previous assignment and the logic itself of the game didn't change but the way our other components interacted with our model did change. The View is responsible for displaying the current state of the game, using graphical elements such as player icons, role indicators, and shot counters. Finally, the Controller manages user interactions, translating mouse clicks and other gestures into method calls that update the model. By using MVC, we ensured that changes to one part of the application, such as updating the graphical representation or modifying the game logic, could be made with minimal impact on other components.

Additionally, we incorporated the Observer design pattern to manage real-time updates between the Model and the View. Since the game's state changes dynamically like how a player moves to a different room, earns credits, or completes a scene. We needed a way to notify the View to update accordingly. The Model acts as the Subject, while the View components serve as Observers that listen for changes and refresh the display when necessary. This allowed us to decouple the View from the Model while maintaining synchronization between them. The Observer pattern efficiently pushed changes when needed, improving performance and responsiveness.