In our Deadwood game, we applied the Single Responsibility Principle to ensure that each class and method serves a distinct purpose. In the Dice class, we kept the responsibility limited to rolling dice and returning the result, avoiding direct printing, which is handled by the view. The Card class is responsible for managing scene card attributes without handling game logic beyond its data. Similarly, the Board class focuses on managing game board and getting rooms without dictating player actions.

We applied the Open/Closed Principle by ensuring that our classes are open for extension but closed for modification. This means that we structured the game so that new features and behaviors can be added without altering the existing code. For example, when introducing new types of scenes or player actions, we designed the Card and Player classes to be easily extendable.

We applied functional cohesion by grouping functions within classes that serve a single, well-defined purpose. A strong example of this is how the Dice class is designed to handle all dice-related operations or our Card class which only has methods related to the cards themselves.

Logical cohesion is present in multiple classes which is where related operations are grouped together despite handling slightly different tasks. In the View class, various methods handle different types of output, such as displaying turn options, invalid Input and winner. These methods are logically related because they all deal with displaying information but serve different purposes. Similarly, the GameController class contains methods for managing different game phases, such as setupGame(), startTurn(), and endTurn(), which are all related to game flow but execute at different points. In the Player class, methods like getFunds(), freshDay(), and setcCurAction() are grouped together because they are needed for player actions, even though each method executes a different in-game behavior. Lastly, the Board class includes

methods like getRoom() and setboard(), all of which deal with managing board and movement logic, even if they perform slightly different tasks. By applying logical cohesion in these classes, we structured the game to group similar operations together while keeping flexibility for different functionalities.

We used message coupling in between multiple classes which is when classes interact only through method calls without directly accessing each other's internal data. A strong example is the interaction between GameController and Player. The GameController calls methods like move(), act(), or rehearse() on Player objects, but it does not access the player’s fields directly. Instead, it relies on method calls to request actions.