a-object-oriented design

Using Encapsulation Classes encapsulate their data and behavior, with methods providing controlled access to internal attributes.
Inheritance: MyGame class inherits from the Game class, utilizing its structure and adding specific game logic.
Polymorphism is achieved through method overriding. For example, different games can have different implementations of play( ), reversePlayDirection( )

Game Class (Abstract):

- an abstract class representing the generic structure of the card game.
- has the attributes players, draw pile, discard pile, & the current direction of play.
- The constructor initializes these attributes, initializes and shuffles the deck, and deals cards to players.
- abstract methods like play( ), reversePlayDirection( ), skipNextPlayer( ), drawTwoNextPlayer( ), handleWildCard( ), and handleWildDrawFourCard( ), which are specific to each game and need to be implemented by concrete game classes.

MyGame Class

- class that extends the abstract Game class, representing a specific card game (a variation of Uno) and any developer can modifiy it.
- The constructor initializes the game with a list of players, and call the superclass constructor to set up the initial game state.
- implement the play( ) method, managing the game loop, player turns, and handling special card effects.
- override methods like reversePlayDirection( ), skipNextPlayer( ), drawTwoNextPlayer( ), handleWildCard( ), and handleWildDrawFourCard( ) to provide game-specific implementations.

Player Class:

- a class representing a player in the card game.
- Has attributes like a name and a hand of cards.
- The methods include getters for the name and hand, as well as methods to add cards, play cards, and check for playable cards.

Card Class:

- representing a playing card.
- has attributes for color and value.
- The methods like isPlayable( ) to check if can be played on the top discard card, setColor( ) to change my color, getValue( ) to get my value, and toString( ) to print.

b-design patterns used in your code

Strategy Pattern:

- The handleSpecialCards method in the MyGame class uses a switch statement to determine the type of special card played (Skip, Reverse, Draw Two, Wild, Wild Draw Four) and then delegates the handling to different methods. Each handling method represents a strategy for dealing with a specific type of special card.

Observer Pattern:

- The displayGameState method in the Game class provides a simple form of observation. In a more elaborate implementation, observers could be notified of changes in the game state.

c-clean code principles (Uncle Bob)

I try as hard as possible to minimize dependencies and avoid duplicating of code to prevent redundancy, Use as few elements as possible, keep it simple, and ensure it is readable and easy to test by make it maintainable.

d-code against “Effective Java” Items (Jushua Bloch)

In public classes, I used accessor methods, not public fields.

Favor composition over inheritance :**inheritance** is a powerful way to achieve code reuse.

Handling null in playCard method:

- The playCard method in the Player class returns null if no playable card is found. Consider throwing an exception or returning a special card indicating no playable card.

```
public Card playCard(Card topDiscard) {
    for (Card card : hand) {
        if (card.isPlayable(topDiscard)) {
            hand.remove(card);
            return card;
        }
    }
    throw new IllegalStateException("No playable card found.");
}
```

Handling equality:

- Instead of using (==) to compare strings in the isPlayable method of the Card class , I used the equals method. This ensures correct behavior when comparing string contents.

```
9
    2 usages
10  @   public boolean isPlayable(Card topDiscard) {
11          return color.equals(topDiscard.color) || value.equals(topDiscard.value) || color.equals("Wild");
12      }
13
    2 usages
```

e-SOLID principles

Single Responsibility Principle :

- The Game class is responsible for managing the game state, player turns, and card handling. It has methods like play(), initializeDeck(), shuffleDeck(), etc. It have a single responsibility related to the game logic.
- The Player class is responsible for managing a players hand and actions. It has methods like playCard(), hasPlayableCard(), and addCard(). This class also have a single responsibility related to player actions.
- The Card class represents a card and has methods for checking if its playable and setting its color. It adheres to SRP by handling card-specific functionalities.

Open/Closed Principle Principle :

The code doesn't explicitly demonstrate adherence to OCP. However, the structure of the classes and their methods allows for extension (adding new special cards or game variations) without modifying existing code.

Liskov Substitution Principle (LSP):

- The code adhere to LSP, as subclasses (MyGame) can be used interchangeably with their base class (Game) without affecting the correctness of the program.

Interface Segregation Principle (ISP):

- There are no interfaces in my code, but the classes (Game, Player, Card) have well-defined and specific methods. Each class implements methods relevant to its responsibilities, and there's no indication of unnecessary methods

Dependency Inversion Principle (DIP):

- The code adheres to DIP to some extent. High-level modules (MyGame, Game) depend on abstractions (List<Player>, List<Card>), and low-level modules (Player, Card) implement these abstractions. This allows for flexibility in changing implementations without affecting the higher-level logic.