Introduction: *The assignment involved programming an engine for a Uno game, where the next developer will be able to easily implement his logic or use my logic, and I have to implement one variation for the game.*

Problem Statement: The task required creating a solid structure or a skeleton for the project. The complexity lay in handling the idea of giving the next developers the freedom of implementing there code without breaking the solid principles.

Object-Oriented Design:

The Uno Game Engine is designed using object-oriented principles to represent the components and there interactions within the Uno game.

The key classes include:

1. **Game Class (Abstract):**
    - It's the basic plan or outline of how the Uno game is supposed to run, it does not go into any specific details of the game.
    - Manages the Uno deck, players, game direction, and card handling during gameplay.
    - Abstract methods that defines the gameplay flow, allowing for customization in subclasses.
    - Concrete methods that are needed in the engine.
    - Has the Play template method which starts the game, we will talk about it in the Design Patterns part.
2. **MyUnoGame Class (Concrete Subclass):**
    - Extends the abstract Game class to provide specific implementation details for my game variation.
    - Implements methods for player turns, card actions, and determining round winners.
    - Encapsulates the game entities (UnoDeck, UnoPlayer, and UnoHand) and uses them in the implementation.
3. **UnoDeck Class and interface:**
    - Represents the Uno card deck, including methods for initializing, drawing cards, and setting the active card and others.
    - Uses the UnoCard class to represent individual Uno cards.
4. **UnoCard:**
    - Represents the Uno card, including methods for checking if the card is an active card.
    - Uses 2 enum (color, type) to give each card a color and a type.

5. **UnoPlayer and UnoHand Classes:**
    - Model individual players and their hands in the Uno game.
    - Implement methods for adding cards, playing cards, and calculating the player score.

SOLID Principles Defense:

1. **Single Responsibility Principle (S):**
    - Each class has a single responsibility. For example, UnoDeck manages the deck, UnoPlayer handles the player functions, and Game controls the game flow.
2. **Open/Closed Principle (O):**
    - The design is open for extension and closed for modification. New game variations can be easily added by creating subclasses without modifying existing code, but the developer has to overwrite some methods.
3. **Liskov Substitution Principle (L):**
    - Subclasses like MyUnoGame can be substituted for their base class (Game) without affecting the correctness of the program.
    - This **Principle** is shown in the relationship between the UnoPlayer interface and its class MyUnoPlayer.
4. **Interface Segregation Principle (I):**
    - Interfaces like UnoPlayer are specific to the needs of implementing classes, every method in it is used in the subclass, avoiding unnecessary methods.
5. **Dependency Inversion Principle (D):**
    - Mostly all the inheritors in the project implement the dependency inversion principle where the class depend on abstract classes or interfaces.

Design Patterns Explanation:

1. **Factory Method Pattern:**
    - Implemented in the **createGameInstance** method in the **MyUnoGame** class.
    - Defines an interface for creating instances of the **Game** class, allowing subclasses to determine the subclass that they want to instantiate.
    - Provides flexibility for extending the game engine with new game variations.
2. **Singleton Pattern:**
    - Implemented in the **Game** class with the **getInstance** method.
    - Ensures that only one instance of the Game class is created during the application's lifecycle.

3. **Template method Pattern:**
    - Utilized in the **Game** class in the Play method.
    - The play method defines the high-level structure of the Uno game but leaves certain steps to be implemented by subclasses (e.g., startRotation, playerTurn, determineRoundWinner).
    - It maintains an overall structure while allowing customization of specific game behaviors in the subclasses

Challenges Faced:

1. **Building the project structure:**
    - Creating a structure for the project was not easy since a lot of Bugs were made and a lot of improvements have been made.
2. **Circular References in Player Linking:**
    - The circular linking of UnoPlayer instances (leftPlayer and rightPlayer) introduced complexities in maintaining references during reversing the game play direction and other actions. Solving these circular references while avoiding null values required a lot of time and effort.

3. **Implementation of Strategy Pattern for Uno Cards:**
    - There was an attempt to apply the **Strategy** pattern to create different types of Uno cards (NumberedCard, ActionCard, WildCard, etc.) and apply there action on their class. However, due to the nature of Uno cards and the limited variation in their creation logic, the Factory Method pattern was found to be less applicable and overly complex for this specific scenario, so I went with the most straightforward approach.

Conclusion: Successfully developed a Java-based solution for the assignment, addressing the challenges of taking the right decision while creating the structure and utilizing the design pattern to implement the SOLID principles, and tried to avoid all the Code Smells.

Future Work: next time I will put more work into designing the structure / skeleton of the project over the implementation the logic in the methods.