UNO Game Engine in Java

1. **Introduction** This report provides an in-depth evaluation of the UNO game engine developed in Java, focusing on its adherence to clean code principles, design patterns, and SOLID principles. The UNO game engine is designed to simulate the popular UNO card game, incorporating various card types and actions, game mechanics, and player interactions. The evaluation aims to assess the code's quality and provide recommendations for improvement to enhance readability, maintainability, and adherence to best practices.
2. **Clean Code Evaluation**

   2.1 **Readability** The codebase demonstrates a clear separation of concerns, with distinct classes and methods handling specific functionalities. For instance:

   - **Class Design**: The hierarchy of classes such as `Card`, `ColoredCard`, `ActionCard`, and the specific card classes like `NumberCard` and `SkipCard` is well-organized. This clear hierarchical structure makes it easy to understand and navigate the relationships between different card types and their attributes.
   - **Method Names**: Methods are named with clarity and precision, such as `getScore()` and `getCardColor()`, which effectively communicate their functionality. This contributes to the overall readability and maintainability of the code.

   2.2 **Naming Conventions**

- **Consistency**: The code adheres to consistent and descriptive naming conventions. For example, classes like `DrawTwoCard`, `WildDrawFourCard`, and methods such as `addCardToHand()` clearly reflect their roles and functionalities. This consistency in naming conventions enhances code readability and understanding.

2.3 **Code Structure**

- **Class Responsibility**: The code structure excels in separating concerns among classes. Each class has a well-defined responsibility, enhancing the modularity and clarity of the codebase. For example:
  - `Card` and its subclasses (`NumberCard`, `SkipCard`) handle card attributes and behaviors, maintaining a focused responsibility.
  - `Deck` efficiently manages the collection of cards and their operations, such as drawing and shuffling, demonstrating a clear single responsibility.
  - `DefaultGame` encapsulates the game logic and player interactions, providing a centralized place for game flow management.

3. **Design Patterns**

3.1 **Patterns**

- **Factory Pattern**: The `CardFactory` class effectively implements the Factory pattern to handle the creation of various card instances. This approach centralizes card creation logic, making it easier to

manage and extend. The `generateCard()` method provides a clear and organized way to instantiate different card types and actions.

- **Strategy Pattern**: The use of the Strategy pattern through the `Action` interface and its implementing classes (e.g., `SkipCard`, `DrawTwoCard`) is well-executed. This pattern encapsulates card actions, allowing for flexible and interchangeable implementations. The separation of action logic into distinct classes promotes clean and manageable code.

3.2 **Appropriateness**

- **Factory Pattern**: The Factory pattern is particularly well-suited for managing the creation of card instances with various attributes and actions. It simplifies the process of adding new card types or actions, contributing to a scalable and maintainable codebase.
- **Strategy Pattern**: The Strategy pattern effectively decouples the card actions from the card itself, enabling the easy addition of new actions or modification of existing ones. This design fosters a flexible and extensible approach to handling card actions.

4. **SOLID Principles**

4.1 **Single Responsibility Principle (SRP)**

- **Class Responsibilities**: The codebase demonstrates strong adherence to SRP by ensuring that most classes focus on a single aspect of the game. For example:

        - `Card` and its subclasses manage specific card properties and behaviors, maintaining a focused responsibility.
        - `Deck` manages card operations such as drawing and shuffling, which aligns with the single responsibility principle.
        - `Player` handles player-specific data, including the hand of cards and score, reflecting a clear and distinct responsibility.

4.2 **Open/Closed Principle (OCP)**

- **Extensibility**: The codebase shows a commendable adherence to OCP by allowing the extension of functionality without modifying existing code. For example, new card types or actions can be introduced by creating new classes or extending existing ones. This design allows for ongoing expansion and enhancement of the game engine.

4.3 **Liskov Substitution Principle (LSP)**

- **Subclass Substitutability**: The code adheres to LSP by ensuring that subclasses of `Card` (e.g., `NumberCard`, `SkipCard`) can be substituted for their parent class without altering the correctness of the program. This guarantees that all card types behave as expected and maintain the integrity of the code.

4.4 **Interface Segregation Principle (ISP)**

- **Interface Design**: The `Action` interface exemplifies ISP by having a single method (`doAction()`) that is relevant and necessary for all implementing classes. This design ensures that classes are not forced to implement methods they do not use, promoting a clean and efficient interface.

4.5 **Dependency Inversion Principle (DIP)**

- **Dependency Management**: The code adheres to DIP by ensuring that high-level modules (e.g., `DefaultGame`) depend on abstractions (e.g., `Card`, `Action`) rather than concrete implementations. This design promotes flexibility and reduces coupling between components, facilitating easier maintenance and testing.

5. Classes Responsibility

Card

- **Responsibility**: Represents a generic card with a type and score. Provides methods to access the card's type and score. Defines `toString()` as an abstract method for subclasses to implement.

ColoredCard

- **Responsibility**: Extends `Card` by adding color information. Represents cards that have an associated color. Provides methods to get the card's color.

WildCard

- **Responsibility**: Extends `Card` and implements `Action`. Represents wild cards that can change the color in the game. Provides a method to get the user-selected color for the wild card.

ActionCard

- **Responsibility**: Extends `ColoredCard` and implements `Action`. Represents cards that have special actions in the game and also include color information. Serves as a base class for action cards.

NumberCard

- **Responsibility**: Extends `ColoredCard`. Represents number cards with a specific color and value. Provides methods to get the card's value and a string representation of the card.

DrawTwoCard

- **Responsibility**: Extends `ActionCard`. Represents a card that forces the next player to draw two cards and skips their turn. Implements the action logic specific to this card.

ReverseCard

- **Responsibility**: Extends `ActionCard`. Represents a card that reverses the direction of play. Implements the action logic specific to this card.

SkipCard

- **Responsibility**: Extends `ActionCard`. Represents a card that skips the next player's turn. Implements the action logic specific to this card.

WildColorChangeCard

- **Responsibility**: Extends `WildCard`. Represents a wild card that allows the player to change the color. Implements the action logic specific to this card.

WildDrawFourCard

- **Responsibility**: Extends `WildCard`. Represents a wild card that forces the next player to draw four cards and change the color. Implements the action logic specific to this card.

Action

- **Responsibility**: Defines the interface for cards that perform actions in the game. Provides the `doAction(Game game)` method that must be implemented by action cards.

CardAction

- **Responsibility**: Enum that defines the possible actions for cards, such as SKIP, REVERSE, DRAW_TWO, etc.

CardColor

- **Responsibility**: Enum that defines the possible colors of the cards.

CardType

- **Responsibility**: Enum that defines the types of cards, such as NUMBERED, ACTION, and WILD.

CardValue

- **Responsibility**: Enum that defines the possible values for numbered cards.

CardFactory

- **Responsibility**: Creates instances of `Card` based on the provided parameters (type, color, value, and action). Centralizes card creation logic to facilitate deck initialization.

Deck

- **Responsibility**: Manages a collection of `Card` objects, including adding, drawing, shuffling, and clearing cards. Initializes the deck with a set of standard cards.

Game

- **Responsibility**: Abstract class representing a game. Manages the game state, including players, the current card color, the current player, and the deck. Defines methods for playing and initializing the game, and drawing hands.

DefaultGame

- **Responsibility**: Extends `Game`. Implements the specific logic for playing a game, including handling player turns, validating card plays, and determining game outcomes. Manages game flow and interactions.

Player

- **Responsibility**: Represents a player in the game. Manages the player's hand of cards, score, and name. Provides methods for manipulating the hand and updating the score.

Main

- **Responsibility**: Entry point for the application. Initializes and starts a `DefaultGame`.