Report: In-depth Analysis of Object-Oriented Design and Principles in Custom UNO Game Implementation

Overview

The UNO Engine is designed to provide a flexible and extensible framework for creating different variations of the UNO card game. The engine is implemented using object-oriented principles, ensuring that it can be easily customized for various game rules and card types.

This comprehensive report delves into the **object-oriented design**, **design patterns**, and principles applied in the Custom UNO Game implementation. It aims to defend the code against renowned guidelines such as **Clean Code** principles by Robert C. Martin, "**Effective Java**" items by Joshua Bloch, and the **SOLID** principles. The analysis encompasses a detailed discussion of each class individually, with an emphasis on the applied design techniques and clear visual representations where necessary.

Classes overview

➔ **Class Card:**

- Purpose: Represents a single UNO card with a color and type.

➔ **Class Player**

- Purpose**:** Represents a player in the game with a hand of cards.

Functionality**:**

- Manages the player's hand: draw and play cards.
- Determines the most frequent color in the player's hand.
- Checks if the player has won and manages UNO calls

➔ **Deck (Abstract Class)**

- Purpose**:** Represents a deck of UNO cards, providing methods to draw and discard cards.
- Functionality**:**
    - Manages card piles: draw pile and discard pile.
    - Handles shuffling and reshuffling of cards.
    - Abstract method initializeDeck is implemented by subclasses to define deck contents.

➔ **Game (Abstract Class)**

- Purpose**:** Manages the overall game flow and interactions between players.
- Functionality**:**
    - Handles game setup: choosing a dealer, dealing cards, and initializing the discard pile.
    - Manages the game loop: players take turns, play cards, and check for a winner.
    - Abstract methods are implemented by subclasses to define game-specific rules and behaviors.

➔ **CustomGame**

- Purpose**:** Extends Game with custom rules and card effects.
- Functionality**:**
    - Implements abstract methods from Game to handle specific card types and game rules.
    - Provides custom behavior for special cards like "Steal" and "Donation".

➔ **StandardDeck and CustomDeck**

- Purpose**:** Provide specific implementations of a deck of UNO cards.
- Functionality**:**
    - StandardDeck contains standard UNO cards.
    - CustomDeck adds custom card types and colors.

➔ **GameRules (Interface)**

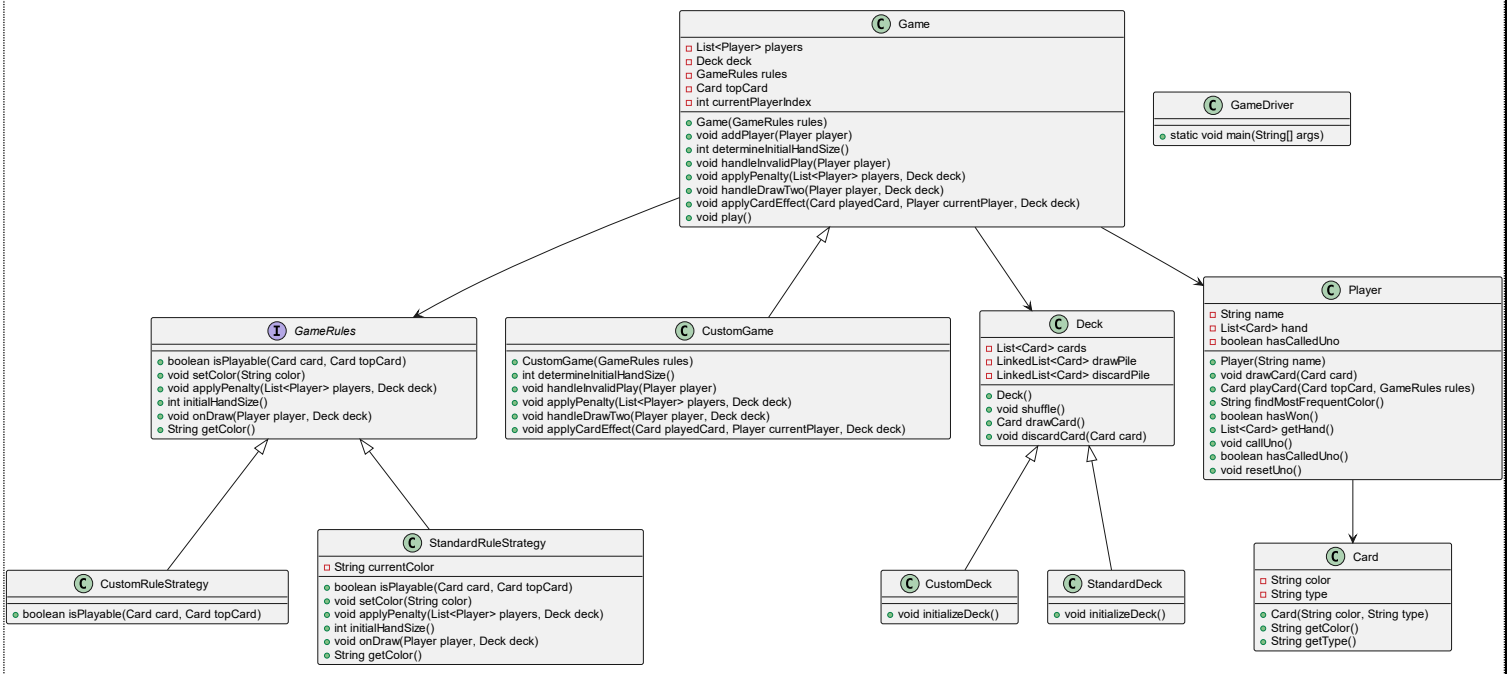
- Purpose**:** Defines the rules for card play and game behavior.
- Functionality**:**
    - Provides methods for determining playable cards, setting colors, applying penalties, and handling special cases.

➔ **StandardRuleStrategy and CustomRuleStrategy**

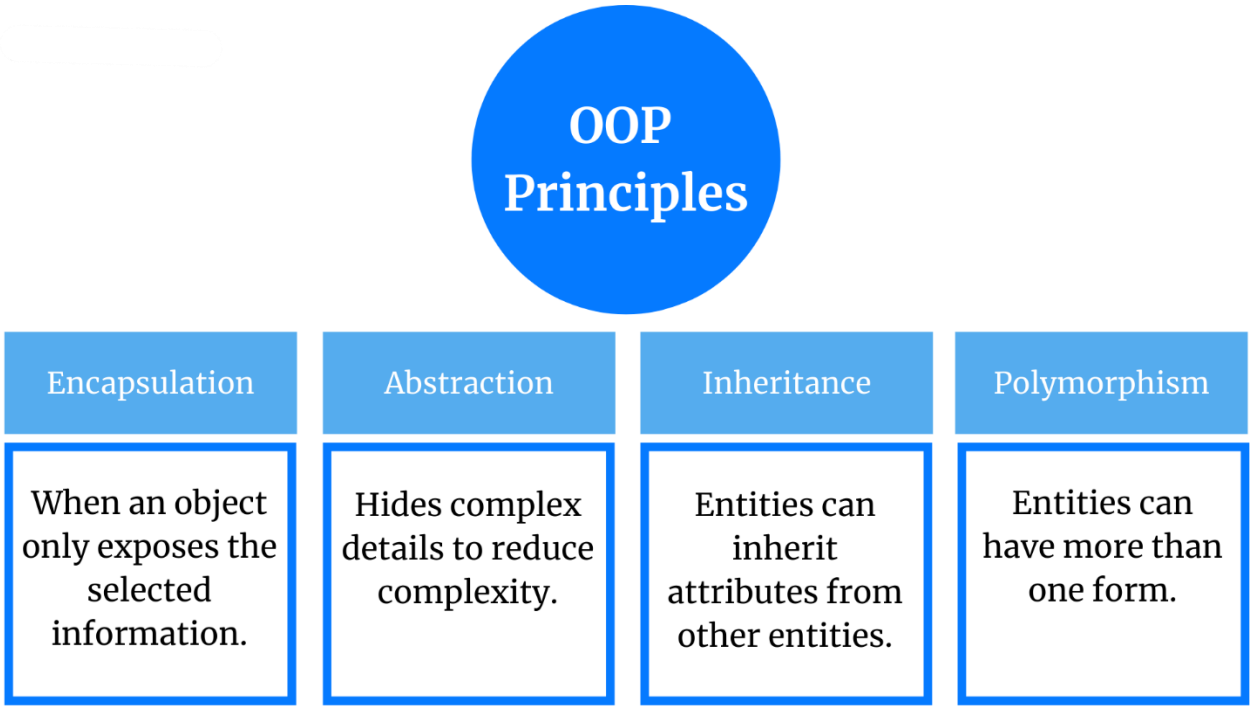
- Purpose**:** Implement game rules for standard and custom game modes.
- Functionality**:**
    - StandardRuleStrategy handles the default UNO rules.
    - CustomRuleStrategy can add or modify rules for custom games.

➔ **GameDriver**

- Purpose**:** Entry point to start and run the game.
- Functionality**:**
    - Sets up the game with players and rules.
    - Starts the game play.



OOP Principles:

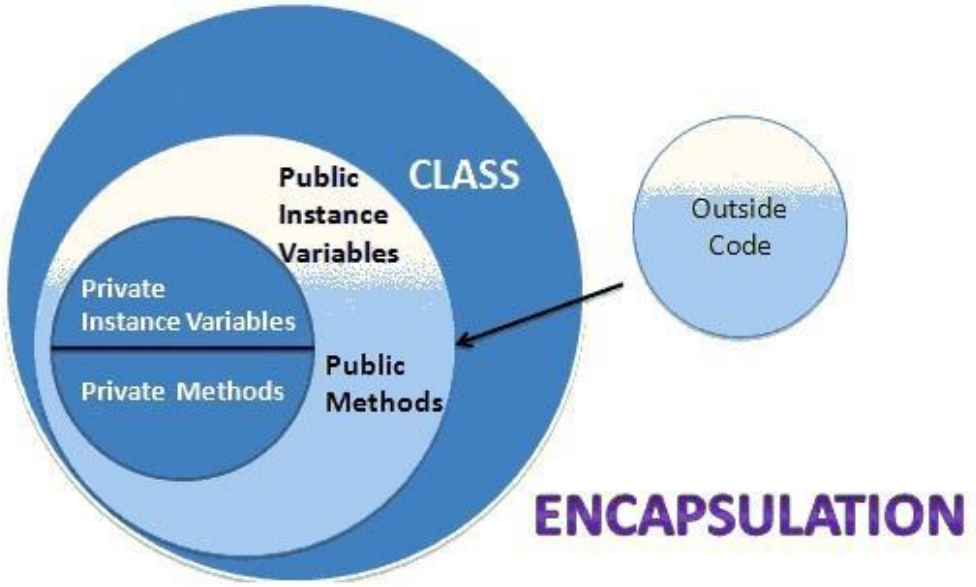


how does my code apply OOP principles?

1. Encapsulation

- Bundling data with methods that operate on that data, and restricting direct access to some of the object's components.
- **Application in the Code:**
    - Classes like Player and Card encapsulate their properties (e.g., name, hand, color, and type) and provide methods to interact with these properties (e.g., drawCard, playCard).

- The Deck class hides the internal details of how cards are managed (e.g., drawPile, discardPile) and exposes methods like drawCard and discardCard.



2. Inheritance

- A mechanism where a new class inherits properties and behaviors from an existing class.
- **Application in the Code:**
  - CustomGame and StandardDeck extend the Game and Deck classes respectively, inheriting their properties and methods. This allows CustomGame and StandardDeck to reuse and extend the functionality of Game and Deck.

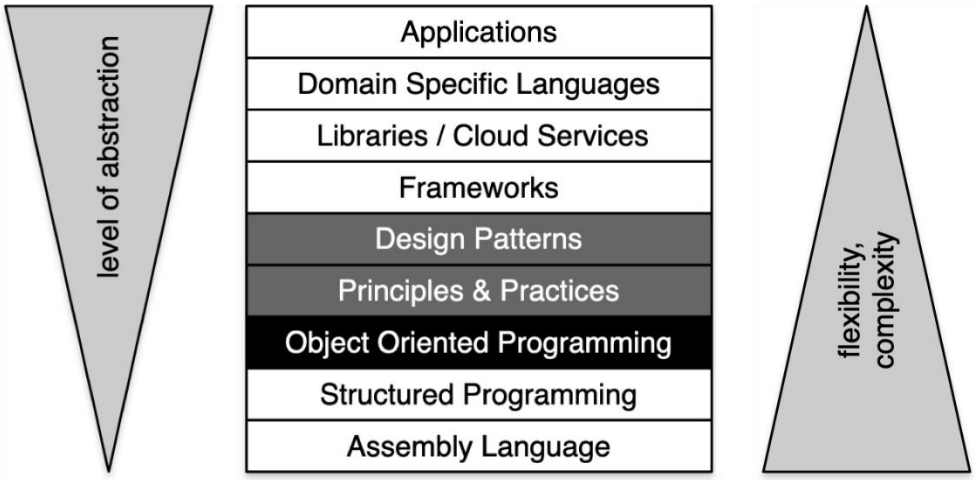
3. Polymorphism

- The ability to present the same interface for different underlying forms (data types).
- **Application in the Code:**
  - The GameRules interface and its implementations (StandardRuleStrategy and CustomRuleStrategy) demonstrate polymorphism. Methods defined in GameRules are implemented differently depending on the strategy used.

- The applyCardEffect method in CustomGame uses polymorphism to handle different card types in a flexible way.

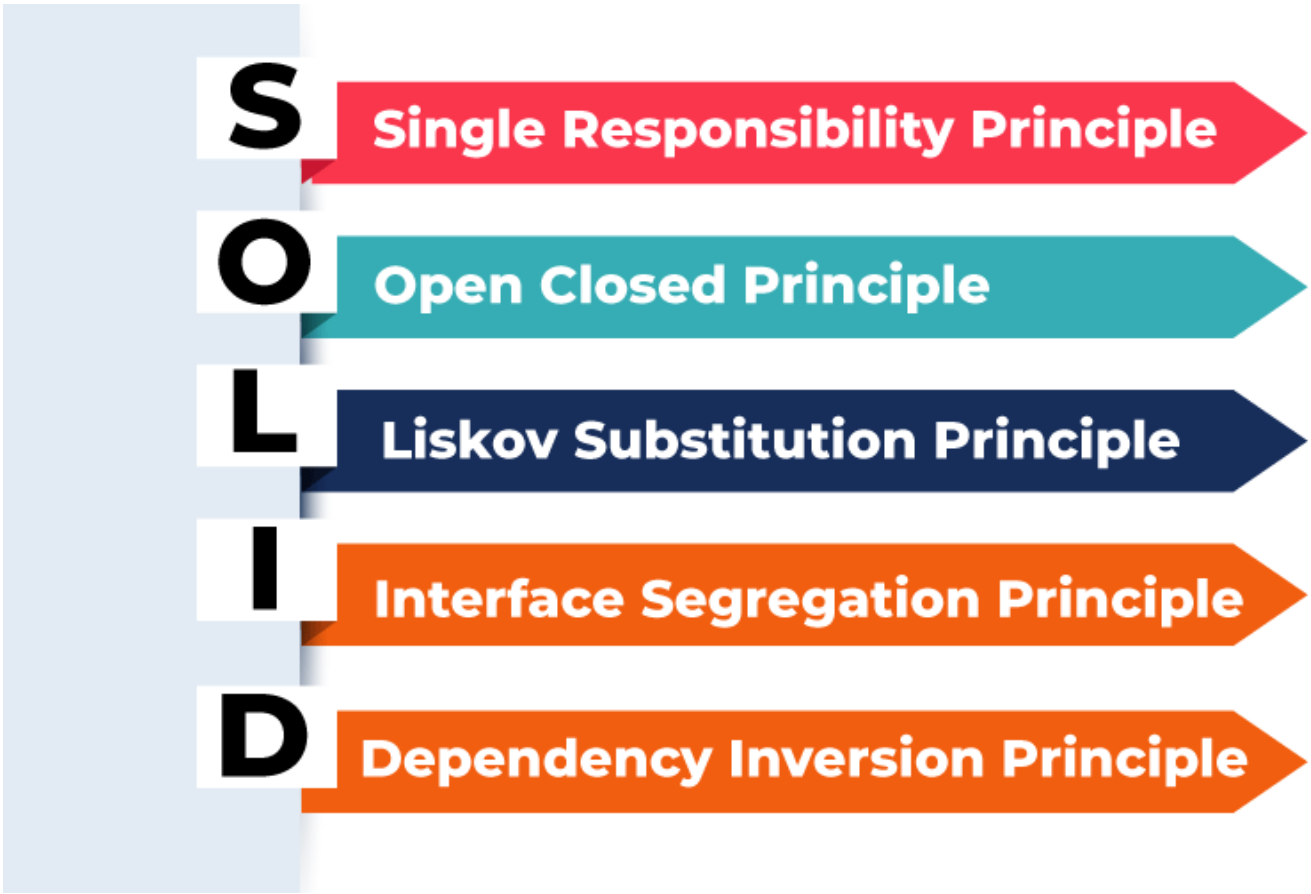
4. Abstraction

- Hiding complex implementation details and showing only the necessary features of an object.
- **Application in the Code:**
  - Abstract classes like Game and Deck provide a simplified interface for managing games and decks while hiding the detailed implementation.
  - The Card class abstracts the concept of a card, providing simple methods to interact with it without exposing its internal details.



In addition there is **Composition** which is Building complex objects from simpler ones, often used in conjunction with encapsulation.

- **Application in the Code:**
  - The Game class uses composition by containing instances of Deck, Player, and GameRules. This allows the game to delegate specific tasks (e.g., card management, rule enforcement) to these composed objects.



1. **ingle Responsibility Principle (SRP)**
    - A class should have only one reason to change, meaning it should have only one responsibility.
    - Application in Code:
        - Card Class: Manages card attributes (color and type). It has a single responsibility—representing a card.
        - Deck Class: Handles deck operations such as drawing and discarding cards. It is responsible for deck management only.
        - Player Class: Manages player actions such as drawing and playing cards. It focuses on player-specific behaviors.
        - Game Class: Manages the overall game flow, ensuring each method handles a specific aspect of game management.
        - CustomGame Class: Extends Game with additional behaviors specific to custom game rules, adhering to the principle by not mixing general and custom rules.

2. **Open/Closed Principle (OCP)**
    - Software entities should be open for extension but closed for modification. You should be able to add new functionality without changing existing code.
    - Application in Code:
        - Deck Class: Can be extended to create different types of decks (e.g., StandardDeck, CustomDeck) without modifying the Deck class itself.
        - Game Class: Can be extended to create different game variants (e.g., CustomGame) by implementing new rules or behaviors, without altering the base Game class.
3. **Liskov Substitution Principle (LSP)**
    - Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
    - Application in Code:
        - Deck and its Subclasses: StandardDeck and CustomDeck can be used interchangeably wherever a Deck is expected, adhering to the principle by maintaining the behavior expected from Deck.
4. **Interface Segregation Principle (ISP)**
    - Definition: Clients should not be forced to depend on interfaces they do not use. Interfaces should be specific to the needs of the client.
    - Application in Code:
        - GameRules Interface: Defines methods relevant to game rules without imposing unnecessary methods. Implementations like StandardRuleStrategy and CustomRuleStrategy only need to implement methods that are relevant to their specific rules.
5. **Dependency Inversion Principle (DIP)**
    - Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces). Abstractions should not depend on details. Details should depend on abstractions.
    - Application in Code:

- Game Class: Depends on the GameRules interface rather than a concrete implementation, allowing different rule strategies (e.g., StandardRuleStrategy, CustomRuleStrategy) to be used interchangeably.
- CustomGame Class: Depends on the Deck abstraction rather than a specific deck implementation, allowing it to work with any deck subclass (e.g., StandardDeck, CustomDeck).

To sum up:

- Keeping classes and interfaces focused on specific responsibilities (SRP).
- Allowing extensions without modifying existing code (OCP).
- Ensuring subclasses can replace their base classes without breaking functionality (LSP).
- Providing interfaces that are specific to client needs (ISP).
- Relying on abstractions rather than concrete implementations (DIP).

Design Pattern

A design pattern is a reusable solution to a common problem in software design. It provides a template for how to solve a particular design issue, promoting best practices and making code more modular, maintainable, and adaptable.

1. **Strategy Pattern**
    - The Strategy pattern defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable. The client can choose which algorithm to use without modifying the context.

- Application in Code:
  - GameRules Interface and Its Implementations:
    - Interface (GameRules): Defines methods for game rules without specifying how they are implemented. It represents the abstraction.
    - Concrete Strategies (StandardRuleStrategy, CustomRuleStrategy): Implement specific rule sets for the game. For example, StandardRuleStrategy handles traditional UNO rules, while CustomRuleStrategy can define custom rules.
  - How It Works in Your Code:
    - The Game class depends on the GameRules interface, allowing it to work with any implementation of game rules. When creating a game, you can pass an instance of StandardRuleStrategy or CustomRuleStrategy to the Game constructor, depending on the rules you want to apply.
    - This allows the game logic to remain unchanged while supporting different sets of rules, demonstrating how the Strategy pattern enables flexibility and adherence to the Open/Closed Principle.

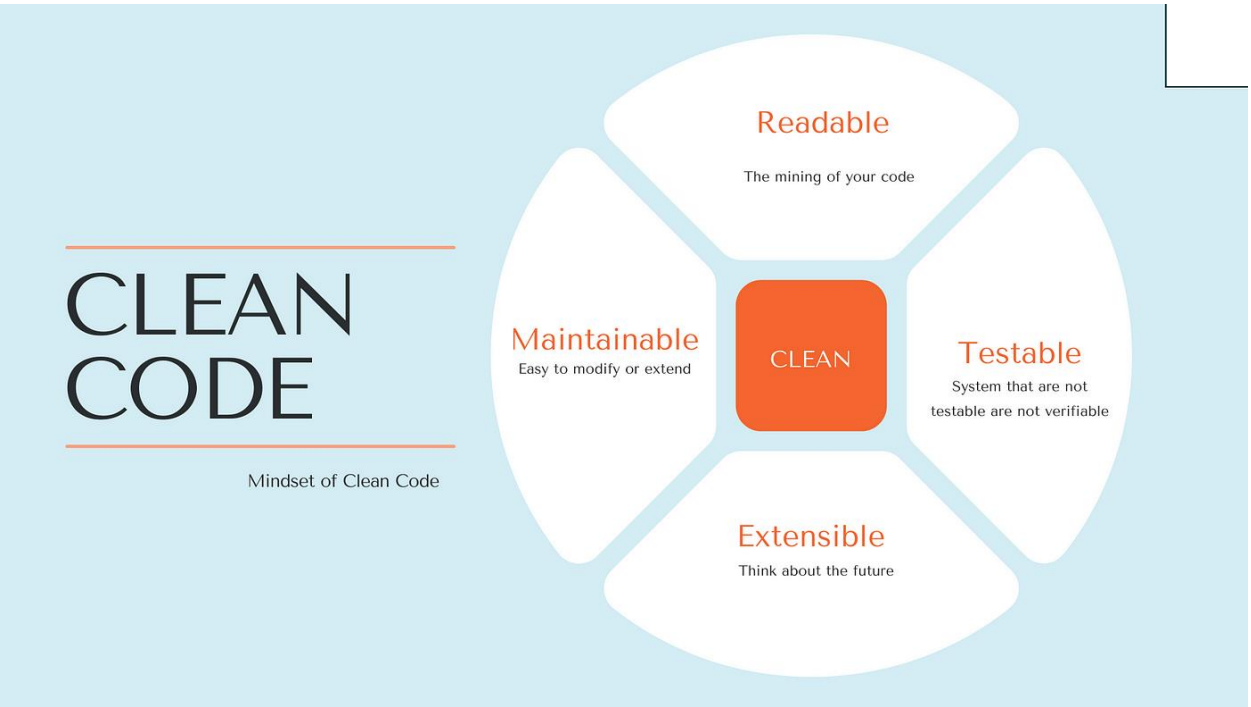
Demonstration**:**

```
GameRules rules = new StandardRuleStrategy();

Game game = new CustomGame(rules);
```

This allows you to run the same game logic with different rule sets by simply changing the strategy, demonstrating the flexibility and extensibility provided by the Strategy pattern.

Clean Code:



Clean Code refers to writing code that is easy to read, understand, and maintain. It emphasizes:

- Readability: Code should be clear and understandable.
- Simplicity: Avoid unnecessary complexity.
- Consistency: Follow consistent naming and structuring conventions.
- Testing: Code should be testable and include appropriate tests.
- Refactoring: Regularly improve code without changing its functionality.

The goal is to make the codebase easier to work with and less prone to errors.

HOW DID I APPLY IT IN MY CODE ?

Readability:

- Meaningful Names: Classes and methods are named clearly, such as CustomGame, CustomDeck, and Player, which convey their roles.
- Comments: Relevant comments explain complex sections of code, such as the purpose of certain methods and the handling of specific card types.

Simplicity:

- Single Responsibility Principle: Each class has a clear, single purpose. For instance, CustomDeck handles deck initialization and shuffling, while Player manages the player's hand.
- Method Length: Methods are kept relatively short and focused on one task, like applyCardEffect in CustomGame, which is divided into smaller methods for different card types.

Consistency:

- Coding Style: Consistent formatting and naming conventions are used throughout the code. For example, method names use camelCase, and classes follow PascalCase.
- Design Patterns: The code applies consistent design patterns, such as Strategy for game rules, which helps in understanding and extending the code.

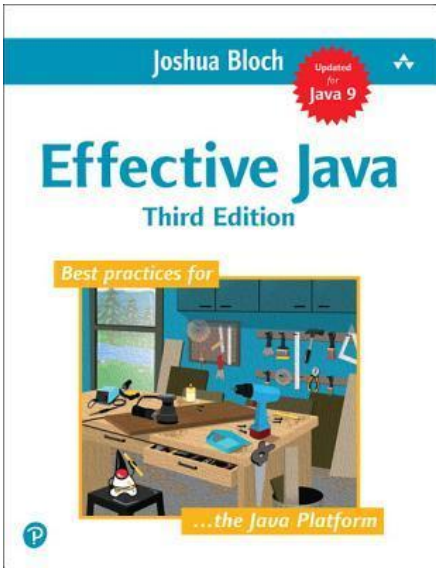
Testing:

- Testable Design: The code is designed in a way that allows for unit testing. For example, GameRules and Deck can be tested independently due to their clear interfaces and responsibilities.

Refactoring:

- Modular Design: The code is modular, allowing for easy updates and maintenance. For instance, new card types or rules can be added by extending existing classes rather than modifying core logic.

"Effective Java" by Joshua Bloch offers practical advice for writing robust, maintainable Java code. It provides guidelines on various aspects of Java programming, from object creation to generics and concurrency.



Key Principles of Effective Java

1. **Use Static Factory Methods Instead of Constructors:**
    - Static factory methods can provide better readability and flexibility compared to constructors. They can return instances of different classes based on input parameters and have descriptive names.

Application in the Code:

- The code uses constructors for class instantiation. Static factory methods could be introduced for creating different types of games or decks to enhance readability and flexibility.

2. **Consider Using Builder Pattern for Complex Objects:**
    - The Builder pattern helps in creating complex objects with many parameters, especially when some parameters are optional.

Application in the Code:

- While the Card and Game classes could potentially use the Builder pattern if they had more parameters or configuration options, the current constructors are sufficient for the existing complexity.

3. **Prefer Composition Over Inheritance:**
    - Favoring composition over inheritance allows for more flexible and reusable code.

Application in the Code:

- The code employs composition in the CustomGame class with instances of Deck and GameRules. This design allows for various implementations of Deck and GameRules, making the game adaptable to different scenarios.

4. **Minimize Mutability:**
    - Immutable objects are inherently thread-safe and easier to understand.

Application in the Code:

- The Card and Player classes could benefit from immutability by using final fields and providing no setters. The current design ensures that once created, the state of these objects is not modified unexpectedly.

5. **Design for Extension, but Code for Use:**
    - Write code that is easy to extend with new features but straightforward to use.

Application in the Code:

- The design allows for easy extension of game rules and card effects. Adding new card types or rule strategies involves creating new classes that implement or extend existing ones, without altering the core Game logic.

6. **Use Dependency Injection**:
    - Dependency injection helps decouple components and simplifies testing.

Application in the Code:

- Dependencies such as Deck and GameRules are injected into the Game class, particularly in the CustomGame constructor. This approach facilitates swapping different implementations for testing or extending the game.

7. **Avoid Overusing Exceptions for Control Flow:**
    - Exceptions should be reserved for exceptional conditions rather than controlling regular execution flow.

Application in the Code:

- Exceptions are used appropriately, such as in the reshuffle method of the Deck class, where a NoSuchElementException is thrown if there are no cards left. This usage of exceptions handles exceptional situations effectively.

8. **Prefer Interfaces to Abstract Classes:**
    - Interfaces provide more flexibility than abstract classes and allow for multiple inheritance.

Application in the Code:

- The code uses interfaces effectively, such as GameRules, which allows different rule strategies to be implemented and used interchangeably.

9. **Use Enum Types Instead of int Constants:**
    - Enums provide type safety and meaningful names for constants.

Application in the Code:

- Strings are used for card types and colors. Enums could enhance type safety and readability by providing meaningful names for card types and colors.

10. **Document Public APIs:**
    - Public methods and classes should be documented to explain their purpose and usage.

Application in the Code:

- The code includes comments and print statements to clarify functionality. Adding JavaDoc comments to public methods and classes would further improve documentation and usability.

References:

UML drawing using drawio

https://drive.google.com/file/d/1r4nXDXmnJsgJqBI3upk4eTIwgiPF_Nyf/view?usp=sharing

Conclusion:

The UNO Engine's design facilitates the creation of different UNO game variations through its modular and extendable architecture. The clear separation of concerns between game logic, rule enforcement, and player interaction makes it easy to modify or extend the engine without affecting existing functionality.