# UNO GAME ENGINE

Writing Effective Code

ABSTRACT

This project implements a customizable Uno game engine in Java, emphasizing flexibility and clean design. The engine supports traditional Uno rules and allows for easy extension with custom rules and cards. Key design patterns like Command and Factory Method ensure maintainable and adaptable code.

NOUREDDIN MOHAMMAD AHMAD LUTFI

Atypon Training

# Outline: UNO GAME ENGINE

1. Introduction
2. Object-Oriented Design
    - Class Structure and Responsibilities
        - Cards Classes & Interfaces
        - Core Classes
        - Utility
    - Encapsulation and Abstraction
    - Inheritance and Polymorphism
3. UML Diagram
    - High level visual representation of class relationships
4. Design Patterns
    - Command Pattern
    - Factory Class
    - Singleton Pattern
    - Builder Pattern
5. Clean Code Principles
    - Meaningful Names
    - Functions
    - Comments
    - Formatting
6. Defence Against SOLID Principles
    - Single Responsibility Principle (SRP)
    - Open/Closed Principle (OCP)
    - Liskov Substitution Principle (LSP)
    - Interface Segregation Principle (ISP)
    - Dependency Inversion Principle (DIP)
7. Adherence to 'Effective Java' Best Practices in Code Design
    - Item 1: Consider Static Factory Methods Instead of Constructors
    - Item 2: Consider a Builder Pattern for Complex Constructors
    - Item 10: Override toString
    - Item 11: Override equals and hashCode Consistently
    - Item 15: Minimize Mutability
    - Item 87: Consider Using a Custom equals Method
8. Scenario: Classic Game Example
9. Conclusion

# Introduction

This report evaluates the object-oriented design, design patterns, and adherence to software principles in the Uno game project. It covers the design choices made, the patterns used, and how the code aligns with principles from "Clean Code" by Uncle Bob, "Effective Java" by Joshua Bloch, and SOLID principles.

# Object-Oriented Design

## ❖ Class Structure and Responsibilities

The Uno game project is built with an object-oriented design to navigate the complexities of the game. At its core, the project is organized into several key classes, each playing a vital role in managing different aspects of gameplay. These primary classes include:

- **Cards Classes & Interfaces**
    - **Card Interface:** Defines methods **getValue()** and **playable(Card card)** for card behavior.
    - **ColoredCard:** Abstract class that extends Card, including a **color attribute**. Provides the **getColor() method**.
    - **ActionCard: Extends ColoredCard and implements Executable**. Adds an **action attribute** and **overrides methods** for **equality** and **string representation**. Includes **playable()** method for card **playability based** on **color** or **action**.
    - **NumberedCard:** Extends **ColoredCard** and represents **numbered cards**. Overrides **getColor()**, **getValue()**, **playable()**, and methods for **equality** and **string representation**.
    - **DrawTwoCard , ReverseCard, SkipCard: Specific action** cards **extending ActionCard**. Each implements the **execute()** method to **perform specific actions** in the game context such as drawing two cards, reversing the game direction, or skipping next player turn.
    - **FillerCard:** Extends **ColoredCard**. It serves as a **placeholder** to add a **FillerCard** for **changing color** after using a WildCard since WildCard doesn't have a color.
    - **BasicWildCard: Implements Card** and **Executable**. Defines a **wildcard card** with abstract **getValue()** and **execute(GameContext gameContext)** methods. It has default **playable()** implementation allowing it to be **always playable**.
    - **WildCard:** Extends **BasicWildCard** and **allows** the player to **choose** a **new color**. It overrides **getValue()** and **execute()** to modify the discard pile and change the color.
    - **WildCard4:** Extends the **WildCard class**, allowing the player to **choose** a **new color** and **forcing** the **next player** to **draw four cards**. This card also **skips** the **affected player's turn**. The WildCard4 overrides the **getValue()** and **execute**(GameContext gameContext) methods.
    - **SwapHandCard:** (Used in **CustomUNO**) Extends BasicWildCard. Swaps the hands between the **current player** and the **next player**.

- **Core Classes**
    - **Game:** Manages the overall **game logic**, **player turns**, and **win conditions**. It acts as the central controller that coordinates **interactions** between different components.
    - **ClassicUNO & CustomUNO**: **Implementations** of **Game**. **ClassicUNO** is a **standard game**, while **CustomUNO** introduces **custom rules** and **additional card interactions**. Also, the CustomUNO game is an example of **how developers** can **extend** the **current design** and is a **proof** of an **easy to extend modular design**.
    - **Player:** Represents a **participant** in the game. Each player has a **hand of cards** and can perform actions based on game rules.
    - **Deck:** Manages the **collection of cards** and **shuffling**. The ClassicDeck and CustomDeck classes demonstrate how the deck can be customized.
    - **ClassicDeck & CustomDeck**: Implementations of Deck. **ClassicDeck** is a **singleton deck** for the classic game, while CustomDeck is a **singleton** that adds **custom cards** and **modifies card drawing behavior**.
    - **DiscardPile**: **Singleton** class **managing** the **discard pile** of the game.
    - **GameContext:** Contains the **state** and **rules** for the current player's turn, including being utilized by special cards like DrawTwoCard, SkipCard, ReverseCard, WildCard, and WildDrawFourCard.
    - **GameDriver:** The **entry point** for **executing** the **game**. It utilizes the **GameFactory** to create an **instance** of the game, either **ClassicUNO** or **CustomUNO**, based on the specified type. It then starts the game by invoking the **play method** on the **game instance**.
- **Utility Classes**
    - **ColorGenerator and Colors:** Utility classes for **handling** card **colors** and **color codes** for **console output**.

## ❖ Encapsulation and Abstraction

**Encapsulation** is used to hide the internal details of the classes, **exposing only necessary interfaces**. For example, the **Card** class encapsulates the details of card properties and behaviors, while the **Game** class manages the overall game flow without exposing the implementation details of individual card actions.

**Abstraction** is achieved using **abstract classes** and **interfaces**, allowing for a flexible and extensible codebase. For instance, the **Executable** interface provides a common contract for executing card actions, which is implemented by **different card types**.

## ❖ Inheritance and Polymorphism

The Project **utilizes inheritance** to **create a hierarchy of card types**. The **Card class** is the base class, with **NumberedCard**, **ActionCard**, and **WildCard inheriting from it**.

# UML Diagram

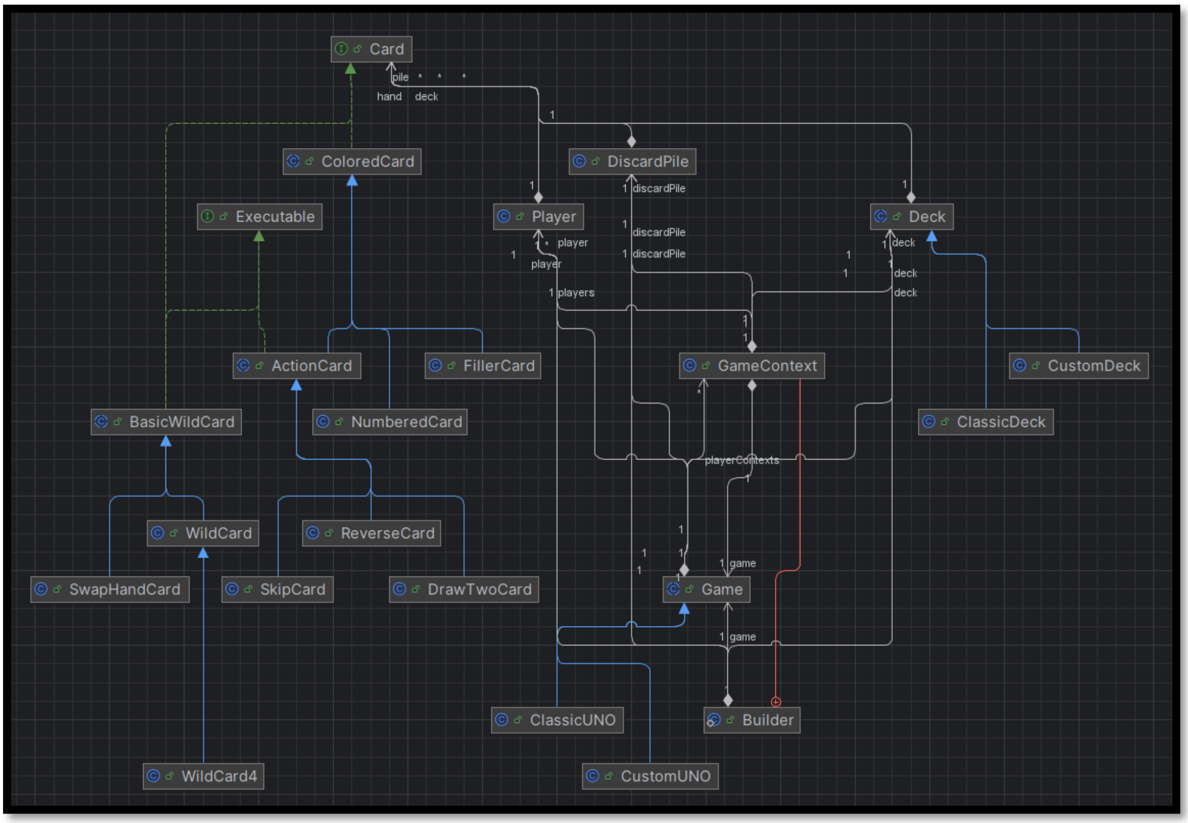


*Figure 1. UML Diagram*

# Design Patterns

The project makes good use of **design patterns** to keep the code **flexible** and **easy to maintain**. These patterns make it simple to add new game features **without messing with the existing code**.

## ❖ Command Pattern

- **Purpose:** The **Command Pattern** helps **encapsulate all the details of an operation in a single object**. For the Uno game, it's used to represent **different card actions**. Each card type (**like DrawTwoCard**, **SkipCard**, etc.) implements the **Executable interface**, which allows them to execute their specific logic when played. This way, card actions can be handled uniformly **without coupling** the game logic to the card behaviors. If a new card is added, it can **simply implement the Executable interface** with its specific behavior, **without affecting** the existing card types.

- **Implementation:** The **Executable interface** plays a **key role** in the **Command Pattern** by defining the **execute**(GameContext context) method, which each card class **implements** to carry out its **specific effect**. This approach ensures that the behavior of each card is **encapsulated** within its **own class**, making it easy to manage and execute actions during gameplay. When adding new card actions, developers **simply need** to **create** a **subclass** that **implements** the **Executable interface**, allowing the new card to **integrate smoothly** with the **existing game logic**.

## ❖ Factory Pattern

- **Purpose:** The **Factory Pattern simplifies object creation by encapsulating the instantiation process**. In the Uno game, the **CardFactory** class manages the creation of various card types such as **NumberedCard**, **ActionCard**, and **WildCard**. This centralization makes the system more **manageable** and **extendable**, while using **static factory methods avoids specifying extra parameters as null** for cards that **don't need them**, keeping the **code clean**. Additionally, the **GameFactory** class utilizes this pattern to handle the creation of different game types, such as **ClassicUNO** and **CustomUNO**, based on a provided type **string**. This approach centralizes game creation, making it straightforward to add new game types **without altering existing logic**.

- **Implementation:**

  - **CardFactory:** Provides **static methods** to create instances of different card types. For instance, **getWildCardInstance** creates **wild cards**, **getActionCardInstance** creates **action cards** with their **color**, and **getNumberedCardInstance creates numbered cards** with **both color** and **number**. This separation avoids **passing unnecessary parameters**, which would be required if a single method were used for all card types.

  - **GameFactory:** Centralizes the creation of game instances by providing the **createGame method**. This method takes a game **type string** and returns an instance of the **appropriate game**, such as **ClassicUNO** or **CustomUNO**. This design simplifies the process of adding new game types and maintains an **extendable** and **clean code**.

## ❖ Singleton Pattern

- **Purpose:** The **Singleton Pattern** ensures that a class has **only one instance** and provides a **global point** of **access to it**. In the Uno game, this pattern is used to manage instances that should be **unique** across the entire application, such as **deck** and **discard pile**.

- **Implementation:** **ClassicDeck**, **CustomDeck**, and **DiscardPile**. For each of these classes, the **Singleton Pattern** is used to ensure that **only one instance** exists throughout the application. This is achieved by employing a **private static instance** and a **public static method** (getInstance()) to manage and provide access to the single instance of each class.

## ❖ Builder Pattern

- **Purpose:** The Builder Pattern **simplifies** the **construction** of **complex objects**, especially when a class has **multiple parameters**. In the Uno game, it's used for creating instances like **GameContext**, where **several configurations** are needed. While the Builder Pattern is **often applied** to handle **multiple constructors** with **different parameters**, in this case, it's used to manage multiple **parameters** in a single constructor, enhancing readability and maintainability.

- **Implementation:** The **GameContext** class **employs** the **Builder Pattern**, allowing the creation of a **GameContext** object with multiple parameters in a flexible and readable manner. The nested Builder class provides methods to **set individual parameters**, and the **build()** method returns the constructed **GameContext instance**. This pattern makes the object **creation** process **intuitive** and **adaptable** to **future changes** or **additional configurations**.

# Clean Code Principles

## ❖ Meaningful Names

The code adheres to the principle of **meaningful names** by ensuring that class, method, and variable names are both **descriptive** and **readable**. I focus on **avoiding ambiguous abbreviations** and using names that clearly convey their purpose. For example, in the **CardFactory** class, **methods** like **getWildCardInstance** and **getActionCardInstance directly indicate** their **function**, **avoiding cryptic names** and making the code self-**explanatory**. Additionally, names are chosen to be **pronounceable** and **easily searchable**, enhancing both code **readability** and **maintainability**.

## ❖ Functions

- **Adherence to Best Practices**

  - **Small Functions:** Functions in the code are designed to be **small** and **focused**, aligning with best practices. For example, **getNextPlayerIndex** and **initializeDeck** each handle a **specific task**, making them easy to understand and maintain.

  - **Functions Doing One Thing:** Each function is crafted to handle a **single responsibility**. For instance, **drawCard** manages the card drawing process, while **checkWinCondition** evaluates if a player has won.

- **Deviations**

  - **Large Functions:** Complex methods like **playTurn** are **necessary** to **handle different game states** and thorough initialization. This approach ensures **comprehensive functionality.**

  - **Functions Arguments:** The **GameContext constructor's multiple parameters** ensure **complete setup** of the game state. While this makes the constructor bulky, it's more **efficient** for **initializing** everything at once rather than **relying** on **numerous setter methods**.

  - **Switch Statements:** The use of **switch statements** in **factory methods** provides a **clear** and **straightforward** way to handle different card types. This method balances simplicity and extensibility, allowing for easy addition of new card types **without overcomplicating** the code.

**Justification**: a need to apply **functionality Over formality**, sometimes practical considerations necessitate larger functions or more complex constructors. These decisions prioritize comprehensive functionality and flexibility, which can be more important than strictly following best practices in certain situations.

## ❖ Comments

The code includes a few **comments**, which are used to **explain parts** that might not be **immediately obvious**. I stick to best practices by adding comments only where they're truly needed and keeping them brief and to the point. This way, they provide helpful context **without overwhelming** the **code**.

### ❖ Formatting

The code adheres to formatting best practices by ensuring **clear vertical** and **horizontal spacing**, **maintaining logical grouping** and **alignment**. I use **consistent indentation** and **avoid excessive density** for **readability**.

# Defence Against SOLID Principles

- **Single Responsibility Principle (SRP):** Each class has a **clear**, **distinct role**. For example, **Game manages game logic**, while **card classes** handle **card-specific behaviors**.

- **Open/Closed Principle (OCP):** **Classes** and **methods** are designed to be open for extension but closed for modification. For instance, you can **add new card types without altering existing code**, using the **Executable interface** and **factory methods**.

- **Liskov Substitution Principle (LSP):** The **ColoredCard** class implements the **Card interface**, and both **NumberedCard** and **ActionCard extend ColoredCard**, ensuring that **all subclasses adhere** to the **Card interface** and can be used **interchangeably without impacting** the system's correctness.

- **Interface Segregation Principle (ISP):** Clients should not be forced to depend on **interfaces** they **do not use**.

  - **Card Interface:** Implemented by **ColoredCard**, which is **extended by NumberedCard** and **ActionCard**, ensuring each class **only handles relevant methods**.

  - **Deck Class:** Includes methods like **drawCard**, **shuffle**, and **initializeDeck**, so clients interact only with **deck-specific functionalities**.

  - **Game Class:** Uses **interfaces** like **Executable** to **separate actions** from card properties, allowing different card types to manage their actions independently and **avoiding unnecessary dependencies**.

- **Dependency Inversion Principle (DIP):** the code adheres to **DIP** by using **abstractions** (**Card**, **Executable**, **Context**, **Game**, **Deck**, etc.) in various places, ensuring that high-level modules (like **Player**) are not tightly coupled with low-level implementations (like **specific card types**), One example is shown in the code below where **Player** depends on **List<Card>** which is an **abstraction** not a **concrete implementation** like **ActionCard** or **NumberedCard**.

```
public class Player {  31 usages
    private final String name;  2 usages
    private List<Card> hand;  5 usages
```

*Figure 2. DIP Example*

# Adherence to 'Effective Java' Best Practices in Code Design

While it's not feasible to address every item from "Effective Java," key principles have been carefully applied throughout the code. This section highlights how specific items have been implemented to enhance the design, efficiency, and maintainability of the codebase.

## ❖ Item 1: Consider Static Factory Methods Instead of Constructors

**Defence:** In the provided code, constructors are used appropriately to initialize classes, particularly where specific attributes such as color and value are required (e.g., NumberedCard, ActionCard). However, **static factory methods** have been effectively employed in other areas of the code, such as in the **CardFactory**, **GameFactory**, and for managing **singleton** instances like **ClassicDeck** and **DiscardPile**. This approach not only enhances readability but also provides better control over the instantiation process, allowing for more flexible and maintainable code.

## ❖ Item 2: Consider a Builder Pattern for Complex Constructors

**Defence:** The code employs the Builder pattern for classes like **GameContext**, which involve **multiple parameters**. This approach streamlines object creation, **enhancing readability** and **maintainability**, while also making the code **adaptable** to **future changes** or **additional configurations**.

## ❖ Item 10: Override toString

**Defence:** I've **overridden** the **toString** method in **all concrete classes** to provide clear and informative string representations, aiding in debugging and enhancing code readability. Abstract classes and interfaces do not override toString, as **they are not directly instantiated**.

## ❖ Item 11: Override equals and hashCode Consistently

**Defence:** The equals and hashCode methods are overridden in classes where logical equality and hash-based collections might be relevant (**ActionCard**, **NumberedCard**, **SwapHandCard**, etc.). These implementations ensure **consistent behavior**, **particularly** when **cards** are **compared** or **stored** in **collections** like **HashMap** or **HashSet**.

## ❖ Item 15: Minimize Mutability

**Defence:** The code **minimizes mutability** by using **final keywords** for attributes that should not change after initialization (e.g., **action in ActionCard**, **number in NumberedCard**).

## ❖ Item 87: Consider Using a Custom equals Method

**Defence:** **Custom equals methods** are **implemented** where **necessary**, considering both **type** and **value** comparisons. This ensures that objects are compared meaningfully within the context of the **game's logic**.

# Scenario: Classic Game Example

The Uno game starts with **Noor** and **Mohammad** as **players**. Noor is the **first** to play and begins with a **Green 1** card. Mohammad responds with a **Green 3** card. Noor then plays a **Green Draw Two** card, causing **Mohammad** to **draw two additional** cards and **skip his turn**.

As the game progresses, **Mohammad uses a Yellow Skip card** and a **Yellow Reverse card**, which alters the turn sequence and **reverses** the **direction** of play (no impact since only 2 players playing). **Noor** plays a **Yellow 5** card, followed by **Mohammad** playing a **Yellow 1** card. Mohammad then plays a **Wild Draw Four card**, changing the color to **Blue** and **forcing Noor** to **draw four cards and skip his turn**.

The game continues with **strategic plays** and **reversals**. **Noor plays** a **Blue Reverse card**, and **Mohammad** then plays a **Blue 8 card**. Noor **draws a card**, and **Mohammad draws a card as well**. Noor continues to play, but Mohammad manages to secure a win by playing a **Green 9** card after using another **Wild Draw Four card** to **change** the **color** to **Green**.

The game ends with **Mohammad victorious**, having managed his cards effectively throughout the rounds.

```
Please Enter Player 1 name:
Noor
Please Enter Player 2 name:
Mohammad
Do you want to add a new player? (Y/N)
n
Top card is NumberedCard{color='Yellow', number='0'}
```

*Figure 3. Output Snippet (Game Start)*

```
Noor's turn. Noor's cards:
--------------------------------------------
NumberedCard{color='Yellow', number='5'}
NumberedCard{color='Yellow', number='4'}
NumberedCard{color='Green', number='1'}
NumberedCard{color='Yellow', number='7'}
ActionCard{color='Green', action='Draw Two'}
ActionCard{color='Red', action='Skip'}
NumberedCard{color='Red', number='4'}
```

*Figure 4. Noor's Starting Cards*

```
Mohammad's turn. Mohammad's cards:
--------------------------------------------
ActionCard{color='Yellow', action='Skip'}
ActionCard{color='Yellow', action='Reverse'}
NumberedCard{color='Green', number='3'}
NumberedCard{color='Blue', number='6'}
NumberedCard{color='Yellow', number='1'}
NumberedCard{color='Green', number='2'}
NumberedCard{color='Red', number='9'}
```

*Figure 5. Mohammad's Starting Cards*

```
Mohammad plays WildCard{action='Wild Draw Four'}
Changing color to Green
Noor draws four cards
Skipping Noor's Turn
Mohammad's turn. Mohammad's cards:
-------------------------------------------------------
NumberedCard{color='Green', number='9'}
-------------------------------------------------------
Mohammad plays NumberedCard{color='Green', number='9'}
Mohammad wins!
```

*Figure 6. Game Ending*

# Conclusion

The Uno game project demonstrates a solid application of object-oriented design principles and design patterns. The code adheres to clean code practices, aligns with "Effective Java" principles, and follows SOLID principles, resulting in a maintainable, flexible, and extensible codebase. The design choices contribute to a robust game engine capable of supporting various Uno game variations.