

Uno Game Engine

ATYPON

Author: Ahmad Emad

GitHub: [Uno-Game-Engine](#)

YouTube Video: [Demonstration](#)

Email: ahmademad995.ae@gmail.com

Objective Overview:

The goal of this assignment is to develop an Uno game engine that serves as a flexible foundation for developers to build their own Uno game variations. This engine will not be a playable game itself but rather a backend framework that provides the essential mechanics and rule enforcement for Uno-based games.

Table of Contents

Introduction	3
Object Oriented Design	3
Inheritance	3
Encapsulation	3
Polymorphism	3
Abstraction	4
Aggregation	4
Class Diagram	5
Design Patterns	6
Creational	6
Clean Code	6
Effective Java Items	8
Item 1	8
Item 3	8
Item 4	9
Item 12	9
Item 15	9
Item 28	10
Item 40	10
Item 58	11
Solid Principles	12
Single Responsibility Principle (SRP)	12
Open/Closed Principle (OCP)	12
Liskov Substitution Principle (LSP)	12

Introduction

The Uno Game Engine serves as a robust and adaptable foundation for developers to create their own customized versions of the Uno card game. Designed with best programming practices and a well-structured architecture, the engine ensures both scalability and maintainability. By leveraging object-oriented programming (OOP), design patterns, and principles from Effective Java, it provides a flexible and extensible framework that simplifies game development while maintaining ease of use.

To streamline the development process, the engine includes a predefined set of game rules that developers can select and modify as needed. This modular approach allows for seamless customization, enabling the creation of engaging and unique Uno experiences with minimal coding effort.

Object Oriented Design

Inheritance

The Uno game engine leverages inheritance to enhance code organization, reusability, and maintainability. The Card class acts as a base blueprint for all card types, defining shared behaviors and ensuring a consistent interface for interacting with cards.

Specialized card types, such as ActionCard and WildCard, extend the base Card class, inheriting common functionality while implementing unique behaviors. Similarly, the WildCard class serves as a foundation for all wild cards, maintaining a structured and cohesive design pattern across the codebase. This approach reduces redundancy, promotes flexibility, and simplifies future expansions of the game engine.

Encapsulation

The Uno game engine effectively applies encapsulation by hiding internal implementation details and exposing only well-defined interfaces. This ensures data integrity, modularity, and controlled access to object states.

For instance, the Deck class encapsulates the creation and management of different card types, preventing direct manipulation of the card distribution process. Similarly, the DiscardPile class encapsulates card tracking and management, ensuring that only valid game operations can modify the pile.

By utilizing private members where appropriate and providing controlled access through public getter methods, the engine protects the internal state of objects, reducing unintended modifications and improving maintainability.

Polymorphism

Polymorphism is a key principle in the Uno game engine, demonstrated through the use of the Card interface and its concrete implementations, such as ActionCard, NumberedCard, and WildCards. This allows for the treatment of different card types in a uniform manner while maintaining flexibility and extensibility.

The `PlayAction()` method in the `Card` class exemplifies polymorphism by enabling dynamic behavior based on the specific card type. Each subclass can override this method to implement its own unique action, ensuring that the appropriate behavior is executed for various card types.

This polymorphic approach results in a flexible and scalable design, where cards can be easily managed and manipulated without the need for complex conditional logic, promoting cleaner and more maintainable code.

Abstraction

The Uno game engine effectively utilizes abstraction to simplify the complexity of individual card types. The `CardAction` interface and the `Card` abstract class play a crucial role in this process. The `CardAction` interface defines essential abstract methods that must be implemented by concrete card types, ensuring a consistent behavior across different card types. Similarly, the `Card` abstract class provides a common structure for all card types, encapsulating shared attributes and behaviors, while leaving the implementation details to be defined by subclasses.

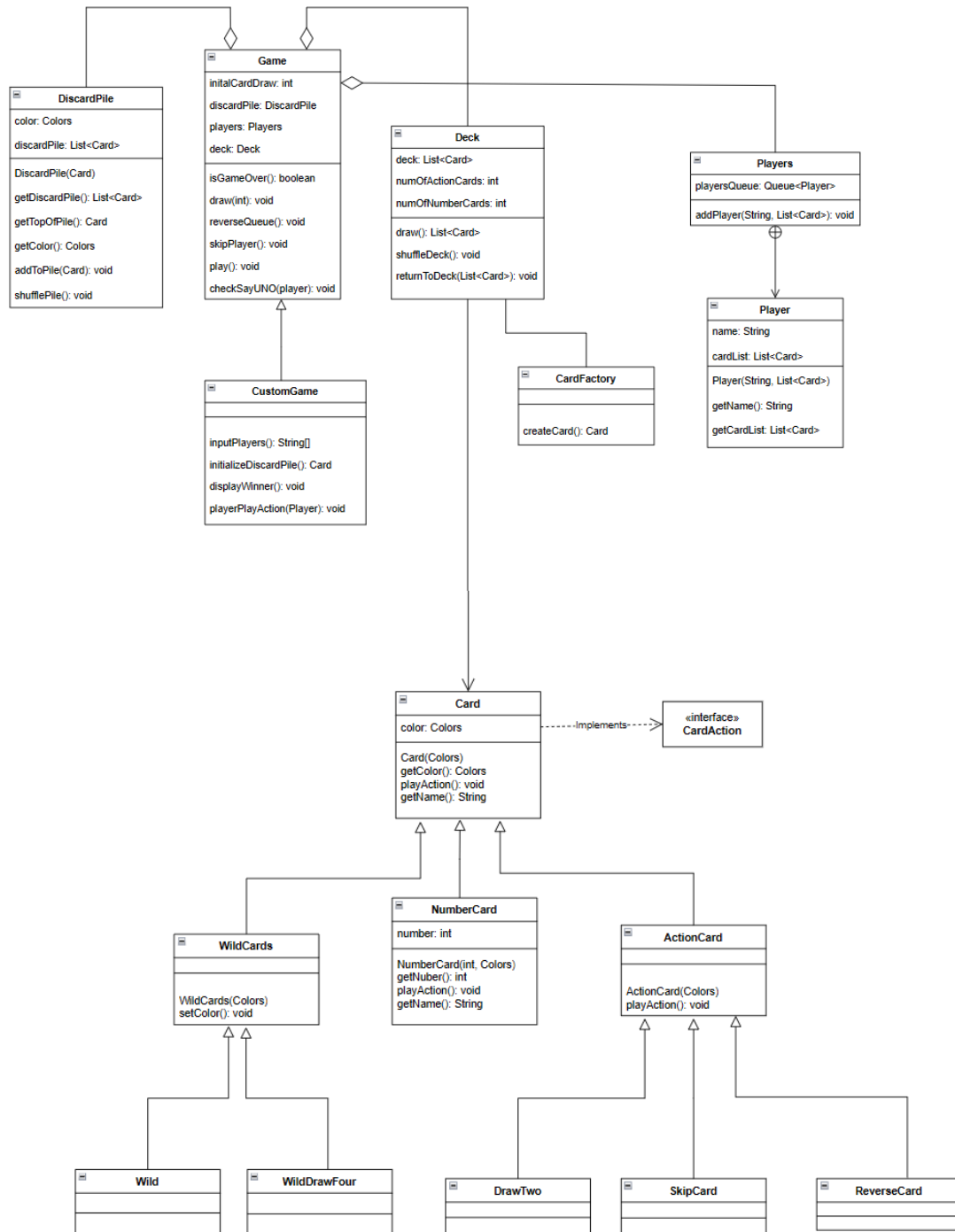
This approach hides the internal complexities of various card types, allowing developers to focus on the interactions at a higher level.

Aggregation

The Uno game engine employs aggregation to establish "has-a" relationships between the `Game` class and several other components, ensuring clear ownership and collaboration between different parts of the system. The `Game` class aggregates the `Deck`, `Player`, and `DiscardPile` classes, signifying that the game contains these objects and relies on them for its functionality.

For example, the `Game` class contains a `Deck` for managing cards, a collection of `Players` who participate in the game, and a `DiscardPile` that tracks cards that have been played. This aggregation allows the game to manage these key components without tightly coupling their implementations, promoting a modular, flexible architecture.

Class Diagram



Design Patterns

Creational

Singleton Pattern

The Uno game engine utilizes the Singleton design pattern for classes like `DiscardPile`, `Deck`, and `Players`, ensuring that only one instance of each exists throughout the game. These classes represent shared resources that all players interact with, making Singleton an ideal choice for managing their state and behavior consistently.

- **Deck:** Ensures that all players draw cards from a single, unified deck.
 - **DiscardPile:** Maintains a central pile where played cards are stored, preventing duplicate or conflicting discard piles.
 - **Players:** Manages the player list and turn order, ensuring a consistent sequence across the game.
- To achieve this, these classes have a special `getInstance()` method. Whenever you need to access the deck for example, you call this method, and it gives you the one and only instance of the deck class. This prevents confusion and keeps the game running smoothly.

Factory Method Pattern

The `CardFactory` class follows the Factory Method design pattern, acting as a centralized card creation mechanism. Instead of scattering card instantiation logic throughout the codebase, the `CardFactory` encapsulates it in a single, well-defined location.

Benefits of Using the Factory Method

- **Encapsulation:** The complexity of creating different card types (e.g., `NumberedCard`, `ActionCard`, `Wildcard`) is hidden within the factory, keeping the main game logic clean and focused.
- **Scalability:** Adding new card types requires minimal changes—developers only need to modify the `CardFactory` without altering other parts of the system.
- **Code Maintainability:** Centralizing card creation improves readability and reduces duplication, making the codebase easier to manage.

By leveraging the Factory Method, the Uno game engine ensures a flexible, extendable, and maintainable approach to card creation, simplifying game expansion and customization.

Clean Code

Meaningful and Consistent Names

The code adheres to clear, descriptive, and meaningful naming conventions across all components. Variables, methods, and exceptions are named in a way that reflects their purpose, ensuring readability and maintainability. The naming follows a consistent convention throughout the project, making it easier for developers to understand and extend the codebase.

Comments

Comments are used only where necessary, focusing on complex logic that may not be immediately clear. Instead of redundant or excessive comments, the code itself is written in a self-explanatory manner, following the principle that "good code is its own best documentation."

Don't Repeat Yourself (DRY)

Duplication is minimized by structuring the code efficiently and reusing logic where possible. An example of DRY implementation is the use of a Draw method with 7 usages in the Deck class, which centralizes drawing from the deck instead of repeating code across different parts of the system. This improves maintainability and reduces redundancy.

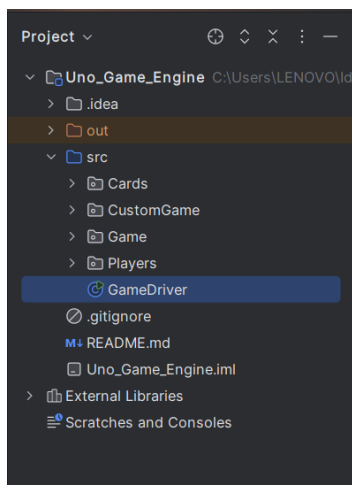
```
public List<Card> draw(int number) { 7 usages  Ahmad Emad
    List<Card> cards = new ArrayList<>();
    for (int i = 0; i < number; i++) {
        cards.add(deck.removeFirst());
    }
    return cards;
}
```

Separation of Concerns (SoC)

The code is modular and well-structured, following the Separation of Concerns (SoC) principle. Classes and packages are organized based on their responsibilities:

- **Players Package:** Manages player queue-related operations.
- **Game Package:** Handles game logic, turn management, and rule enforcement.
- **Cards Package:** Manages card creation, rules, and behavior.

This well-defined architecture enhances scalability, reusability, and maintainability, ensuring that different components can evolve independently without affecting the entire system.



Effective Java Items

Item 1

Consider Static Factory Methods Instead of Constructors

Where applicable, static factory methods are used instead of public constructors to provide more meaningful names, control instantiation, and improve flexibility. This pattern is especially useful for creating instances of cards in the CardFactory class.

```
1 package Cards;
2
3 public class CardFactory { 13 usages new *
4     @ public static Card createCard(int number, Colors color){ 8 usages new *
5         return new NumberCard(number,color);
6     }
7     @ public static Card createCard(String cardType, Colors color) { 3 usages new *
8         return switch (cardType) {
9             case "Skip" -> new SkipCard(color);
10            case "Reverse" -> new ReverseCard(color);
11            case "DrawTwo" -> new DrawTwo(color);
12            default -> throw new IllegalStateException("Unexpected value: " + cardType);
13        };
14    }
15    @ public static Card createCard(String cardType) { 2 usages new *
16        return switch (cardType) {
17            case "Wild" -> new Wild();
18            case "WildDrawFour" -> new WildDrawFour();
19            default -> throw new IllegalStateException("Unexpected value: " + cardType);
20        };
21    }
22 }
23
```

Item 3

Enforce the Singleton Property with a Private Constructor or an Enum Type

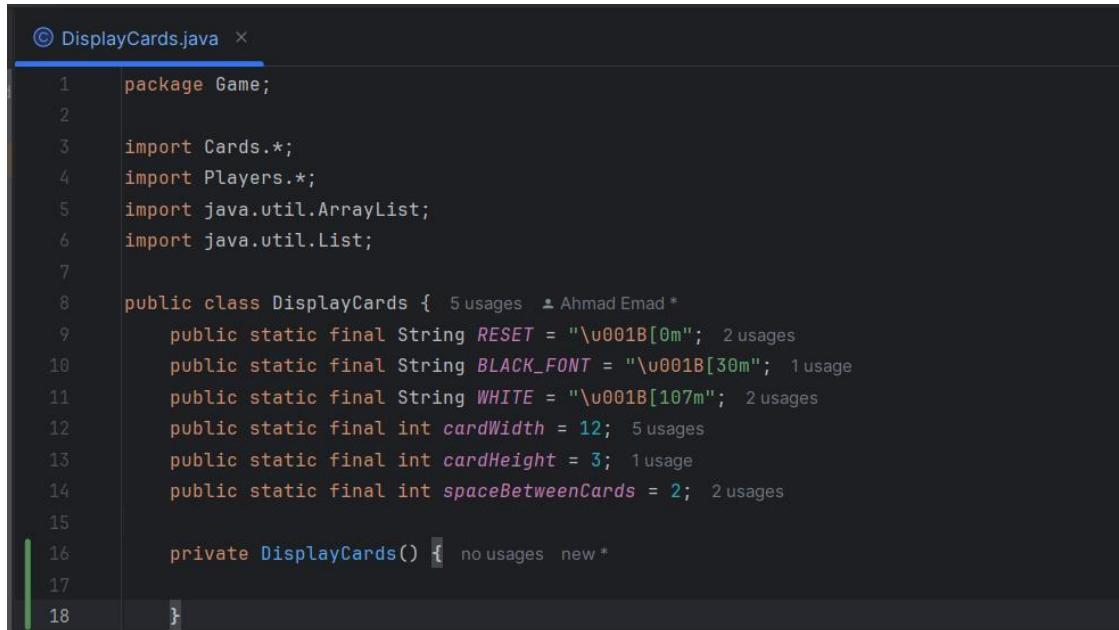
As explained in [creational design patterns](#) (Singleton Pattern), classes such as Deck, DiscardPile, and Players follow the Singleton design pattern, ensuring only one instance exists throughout the game. This is enforced using private constructors and a static instance method to prevent multiple instantiations.

```
(E) Colors.java x
1 package Cards;
2
3 public enum Colors { 1 Ahmad Emad *
4     RED, GREEN, BLUE, YELLOW; 3 usages
5 }
```


Item 4

Enforce Noninstantiability with a Private Constructor

Utility classes that contain only static methods (such as potential helper classes for game logic) include private constructors to prevent instantiation, ensuring they are used solely for their intended purpose.

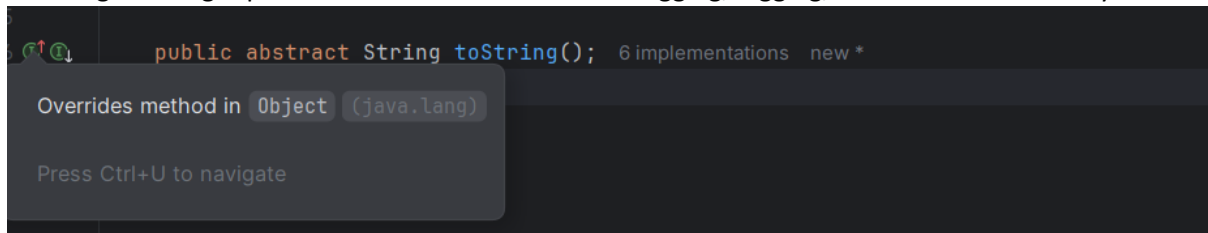


```
1 package Game;
2
3 import Cards.*;
4 import Players.*;
5 import java.util.ArrayList;
6 import java.util.List;
7
8 public class DisplayCards { 5 usages  ⬆ Ahmad Emad *
9     public static final String RESET = "\u001B[0m"; 2 usages
10    public static final String BLACK_FONT = "\u001B[30m"; 1 usage
11    public static final String WHITE = "\u001B[107m"; 2 usages
12    public static final int cardWidth = 12; 5 usages
13    public static final int cardHeight = 3; 1 usage
14    public static final int spaceBetweenCards = 2; 2 usages
15
16    private DisplayCards() { no usages  new *
17
18 }
```

Item 12

Always Override toString

The toString method is overridden in key classes (such as subclasses of Card, and Player class) to provide meaningful string representations. This enhances debugging, logging, and overall code clarity.



```
public abstract String toString(); 6 implementations  new *
```

Overrides method in `Object` (`java.lang`)

Press Ctrl+U to navigate

Item 15

Minimize the Accessibility of Classes and Members

Encapsulation is maintained by declaring class members as private and exposing only necessary functionality through public getters and setters. This prevents unintended modifications to internal state

and adheres to information hiding principles.

```
1 package Cards;
2
3 public class NumberCard extends Card { 7 usages  ⚡ Ahmad Emad *
4     private final int number; 2 usages
5
6     public NumberCard(int number, Colors color) { 2 usages  ⚡ Ahmad Emad
7         super(color);
8         this.number = number;
9     }
10
11     public int getNumber() { 4 usages  ⚡ Ahmad Emad
12         return number;
13     }
14
15     @Override  ⚡ Ahmad Emad
16     public void playAction() {
17     }
18
19     @Override 2 usages  ⚡ Ahmad Emad *
20     public String getName() {
21         return "Number Card";
22     }
23
24     @Override new *
25     public String toString() {
26         return getName() + " " + getNumber();
27     }
28 }
```

Item 28

Prefer Lists to Arrays

In the implementation, Lists (ArrayList, LinkedList, etc.) are preferred over arrays for handling dynamic collections, such as the deck of cards and player hands. Lists offer better flexibility, dynamic sizing, and built-in utility methods, reducing the need for manual array resizing.

```
public class Deck { 5 usages  ⚡ Ahmad Emad
    static final int numOfActionCards = 8; 1 usage
    static final int numOfNumberCards = 9; 1 usage
    private final List<Card> deck; 17 usages
    private static Deck deckInstance; 3 usages

    public Deck(){ 1 usage  ⚡ Ahmad Emad
        deck = new ArrayList<>();
    }
```

Item 40

Consistently Use the @Override Annotation

The @Override annotation is consistently applied when overriding superclass methods, ensuring that methods are correctly overridden and reducing the risk of accidental mistakes due to method signature mismatches.

```

3 public class NumberCard extends Card { 7 usages  ⚡ Ahmad Emad *
4     private final int number; 2 usages
5
6     public NumberCard(int number, Colors color) { 2 usages  ⚡ Ahmad Emad
7         super(color);
8         this.number = number;
9     }
10
11     public int getNumber() { 4 usages  ⚡ Ahmad Emad
12         return number;
13     }
14
15     @Override  ⚡ Ahmad Emad
16     public void playAction() {
17     }
18
19     @Override 2 usages  ⚡ Ahmad Emad *
20     public String getName() {
21         return "Number Card";
22     }
23
24     @Override new *
25     public String toString() {
26         return getName() + " " + getNumber();
27     }
28 }

```

Item 58

Prefer For-Each Loop to Traditional For Loops

Wherever possible, the for-each loop is used instead of traditional for loops, improving readability and reducing the likelihood of off-by-one errors. This is particularly useful when iterating over collections such as card lists, player queues, and game rules.

```

@Override 1 usage  ⚡ Ahmad Emad *
public void play() {
    players = Players.getPlayersInstance();
    deck = Deck.getInstance();

    String[] name = inputPlayers();

    for (String playerName : name) {
        players.addPlayer(playerName, deck.draw(initialCardDraw));
    }

    Card card = initializeDiscardPile();
    discardPile = DiscardPile.getDiscardPileInstance(card);

    while (!isGameOver()) {
        Players.Player player = players.playersQueue.remove();
        players.playersQueue.add(player);
        playerPlayAction(player);
    }
    displayWinner();
}

```

Solid Principles

Single Responsibility Principle (SRP)

The codebase follows the Single Responsibility Principle (SRP) by ensuring that each class has a well-defined and focused responsibility, leading to better modularity, readability, and maintainability.

Here are some examples of how SRP is applied:

- **Player Class** → Manages player-specific actions, such as drawing and playing cards, tracking their hand, and handling turn-based actions.
- **DisplayCards Class** → Handles the display logic for printing player hands and the top card on the discard pile, separating UI concerns from game logic.
- **DiscardPile Class** → Manages the operations related to the discard pile, ensuring that only valid cards are placed and keeping track of the last played card.

Open/Closed Principle (OCP)

The Open/Closed Principle (OCP) states that software entities should be open for extension but closed for modification. The implementation follows this principle effectively, particularly in the card hierarchy and game rule flexibility.

Key Implementations of OCP

- **Card Hierarchy** → The abstract Card class and its concrete subclasses (NumberedCard, ActionCard, and WildCards) enable the addition of new card types without modifying existing logic. New card types can be introduced by extending the Card class without altering its current functionality.
- **Deck Class** → The deck creation logic is designed to accommodate new types of cards dynamically, ensuring flexibility.

This structure ensures that new features and game mechanics can be added with minimal changes to the existing codebase

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) ensures that subtypes can be substituted for their base types without altering program correctness. my card hierarchy effectively follows this principle by allowing different card types to be used interchangeably through polymorphism.

Key Implementations of LSP

- **Card Hierarchy** → The abstract Card class and its subclasses (ActionCard, WildCard, and NumberedCard) ensure that any card type can be handled uniformly in the game logic.
- **Method Overriding** → Methods like playAction() and toString() are properly overridden in subclasses, ensuring that each card type behaves correctly without breaking the expected behavior of the base class.