**Report: Uno Card Game Engine**

Contents

[Overview 2](#_Toc155286505)

[Object-Oriented Design: 2](#_Toc155286506)

[Class Descriptions: 4](#_Toc155286507)

[Clean Code Principles: 6](#_Toc155286508)

[SOLID Principles: 7](#_Toc155286509)

[Note: 7](#_Toc155286510)

Overview:

The Uno Card Game Engine is designed using object-oriented principles, encapsulating the game logic into several classes. The primary classes involved are UnoDeck, UnoCard, and the abstract class Game, along with its concrete implementation MyGame. This design follows the principles of encapsulation, inheritance, and polymorphism, promoting modularity, flexibility, and readability.

# Object-Oriented Design:

1. Class Hierarchy:

The design follows a class hierarchy with a clear separation of concerns. Three main classes - UnoDeck, UnoCard, and Game - form the core structure. MyGame extends the abstract Game class, providing a concrete implementation of the Uno card game.

2. Encapsulation:

Each class encapsulates specific functionalities, promoting information hiding. For example:

UnoDeck encapsulates the deck-related operations and properties.

UnoCard encapsulates the properties and methods related to individual Uno cards.

Game encapsulates the overall game logic, managing players, turns, and the game state.

MyGame extends Game and encapsulates the specific rules and behaviors for the Uno game.

3. Inheritance:

The design utilizes inheritance to create a base class Game that defines common game-related attributes and methods. The MyGame class inherits from Game and provides specific implementations for Uno game rules. This supports code reuse and allows for future extensions.

4. Polymorphism:

Polymorphism is employed through method overriding in the MyGame class. Any methods are overridden to implement Uno-specific rules in MyGame class are consider using runtime Polymorphism.

5. Composition:

For example the Game class is composed of instances of UnoDeck, representing the deck of Uno cards.

6. Abstraction:

The abstract class Game provides a level of abstraction by defining a common behavior for various card games.

# Class Descriptions:

**a) UnoDeck:**

Represents the Uno card deck and its functionalities.

Attributes:

cards: Array of UnoCard objects representing the deck.

cardsInDeck: Integer tracking the number of cards in the deck.

Methods:

reset(): Initializes the deck with Uno cards, respecting colors and values.

replaceDeckWith(ArrayList<UnoCard> cards): Replaces the deck with a custom set of cards(stockPile)

isEmpty(): Checks if the deck is empty.

shuffle(): Shuffles the deck using the Fisher-Yates algorithm.

drawCard(): Draws a single card from the deck.

drawCard(int n): Draws multiple cards from the deck.

**b) UnoCard:**

Represents an Uno card with color and value.

Attributes:

color: Enum representing the color of the card.

value: Enum representing the value of the card (number, action, or wild).

Methods:

getColor(), getValue(): Getter methods for color and value.

toString(): Returns a string representation of the card.

**c) Game:**

Abstract class defining the common structure and behavior of Uno games.

Attributes:

currentPlayer: Integer representing the index of the current player.

playerIds: Array storing player identifiers.

deck: UnoDeck object representing the game deck.

playersHand: ArrayList of ArrayLists representing each player's hand.

stockPile: ArrayList representing the stockpile of played cards.

validColor, validValue: Represents the valid color and value of the last played card.

gameDirection: Boolean indicating the direction of play.

Methods:

play(Game game): Method representing the main game loop.

And various utility methods to handle player turns, drawing cards, checking valid card plays, etc.

**d) MyGame:**

Concrete implementation of the Game class, providing specific rules for the Uno game.

# Clean Code Principles:

Clean code principles prioritize readability and maintainability. This involves using clear names for classes, methods, and variables, maintaining consistent formatting, and favoring small methods. The Single Responsibility Principle ensures each class focuses on one aspect of the game. Comments are used judiciously for complex logic, and error handling improves code robustness. Consistent naming conventions and coding style contribute to overall code simplicity and organization.

# SOLID Principles:

**S - Single Responsibility Principle:** Each class in the Uno Card Game Engine has a clear and single responsibility as I mentioned in class description above.

**O - Open/Closed Principle:** The design is open for extension but closed for modification.

New Uno games can be added by creating a new class that extends the abstract Game class. Existing classes remain unchanged.

**L -** **Liskov Substitution Principle:** Derived classes (e.g., MyGame) can be used interchangeably with the base class (Game).

**I - Interface Segregation Principle:** Class will not be forced to implement unnecessary methods which will not be use (myGame class can select whatever methods it needs from Game class and have the option to override and change the functionality of this methods).

**D - Dependency Inversion Principle:** High-level modules (myGame) is not directly dependent on low-level modules (UnoDeck, UnoCard).

# Note:

1. I've incorporated additional methods in the Uno game engine that might not be directly utilized in the provided game. These methods are strategically included for potential use by developers extending the framework. While they may not be necessary for the current game scenario, they offer a foundation for future customization.
2. I considered moving the play method to MyGame, but I've decided to keep it in Game. My aim was to create a versatile framework that provides a complete Uno game structure. Developers using this framework can freely override behaviors within MyGame while utilizing the standardized core game functions in the Game class.

<https://github.com/hamoodahalabed/Uno-Game/tree/master>