Uno Game Engine Report

Prepared by:

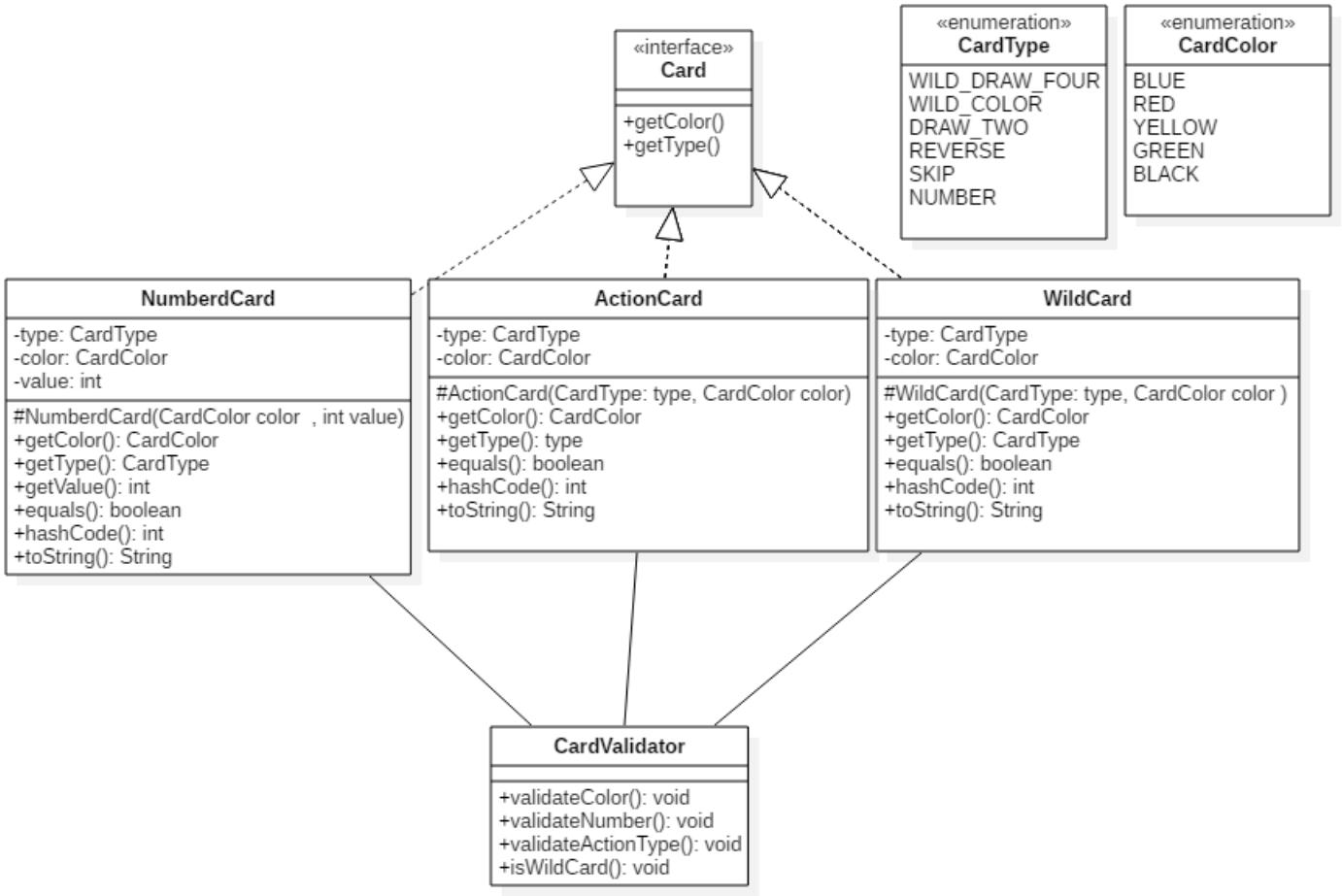
Abdallah Alkayyal

Abstract

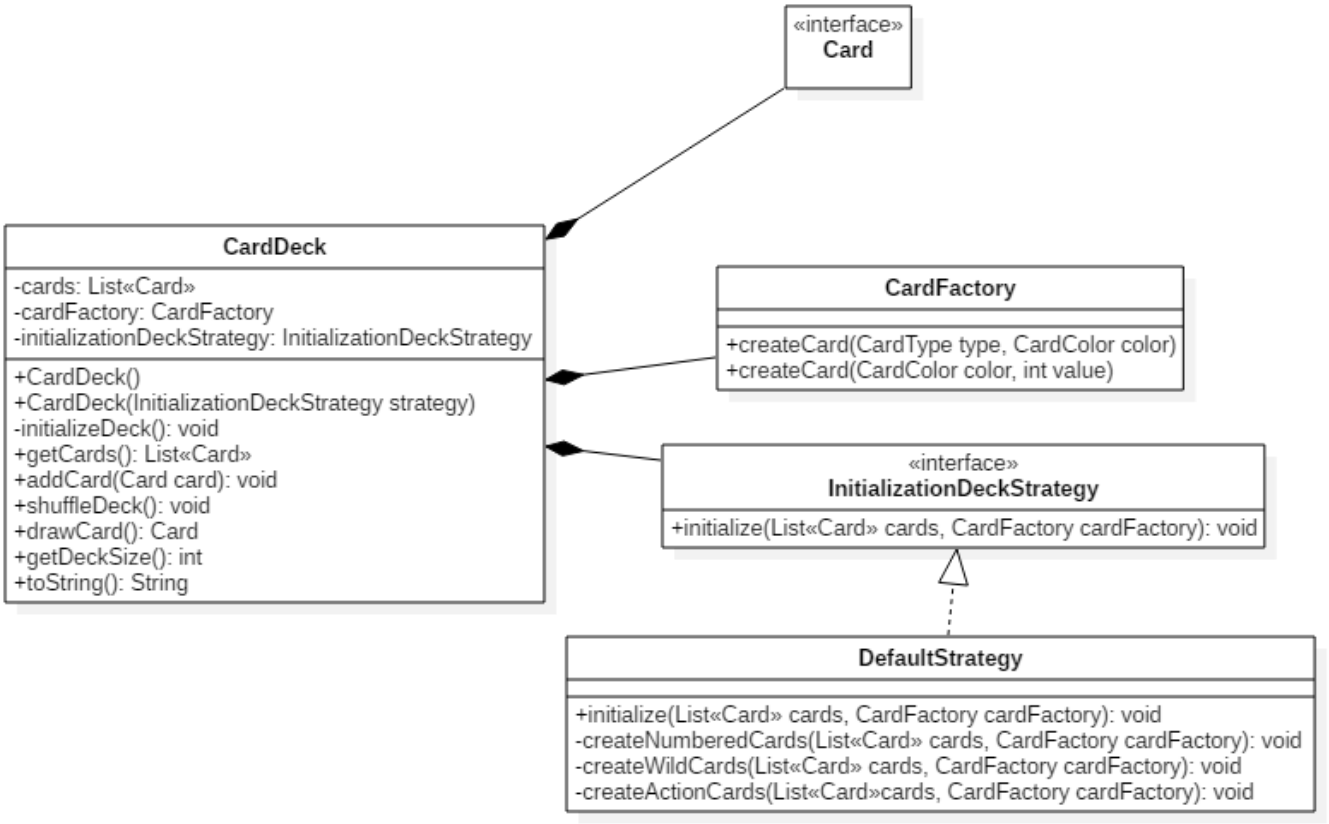
Uno Game Engine is a Java-based framework for developers to create their own variations of the Uno card game. The engine includes an abstract Game class that developers can extend to build their Uno games. Predefined game rules are provided, and developers can easily add new rules or features. The **Game** class has a **play()** method to simulate the game, and the **GameDriver** class handles instantiation and invocation. At least one real Uno game variation is included as an example.

Class Diagrams

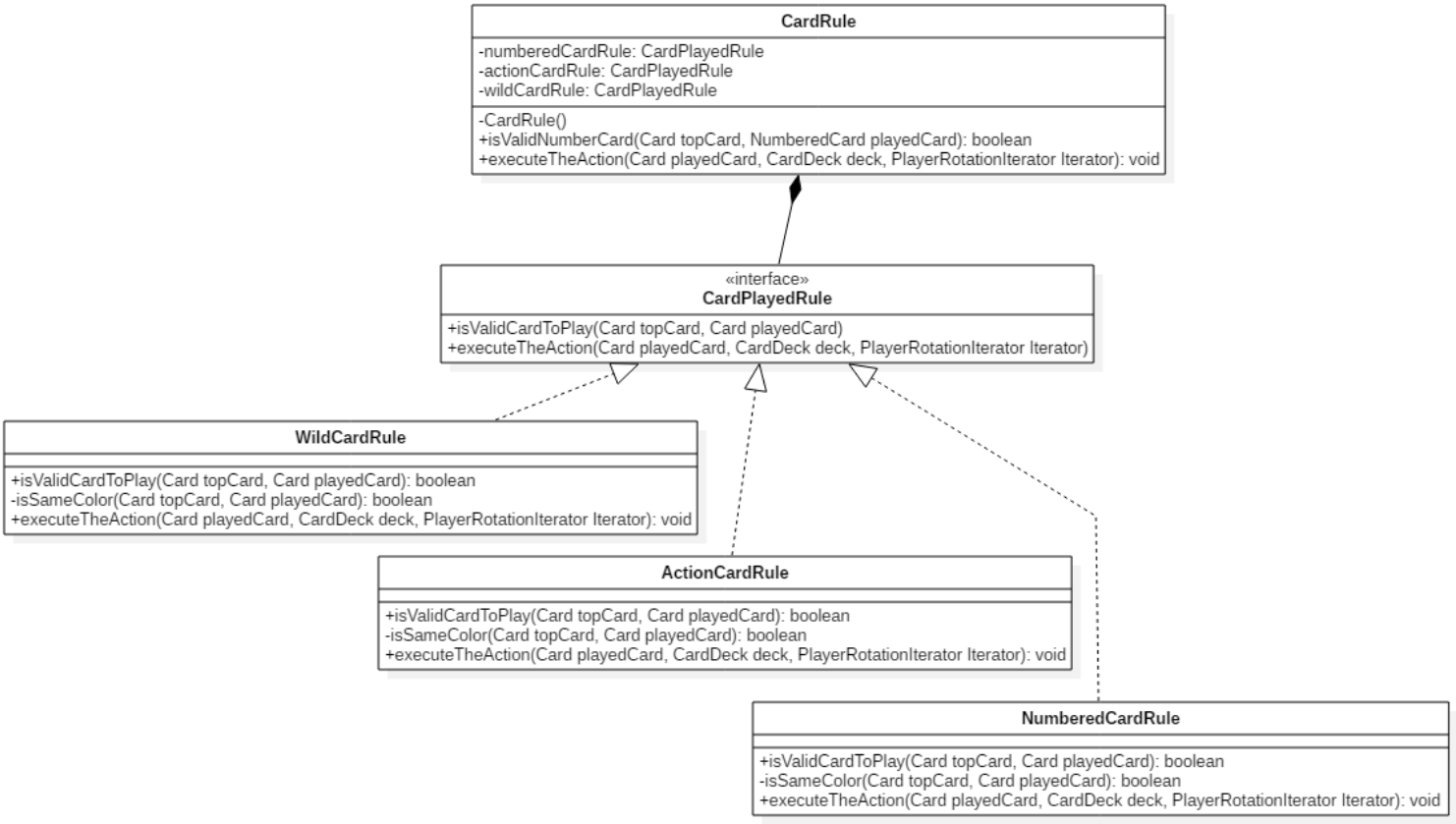
Card Package Diagram



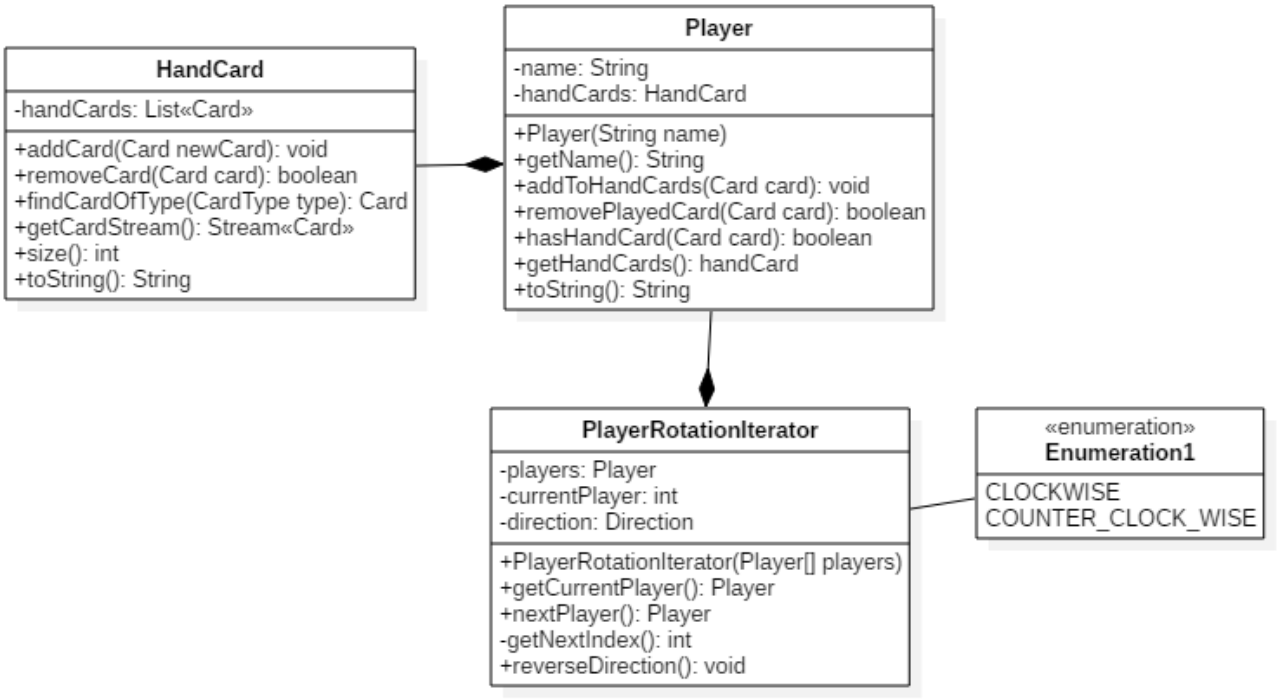
Card Deck Package Diagram



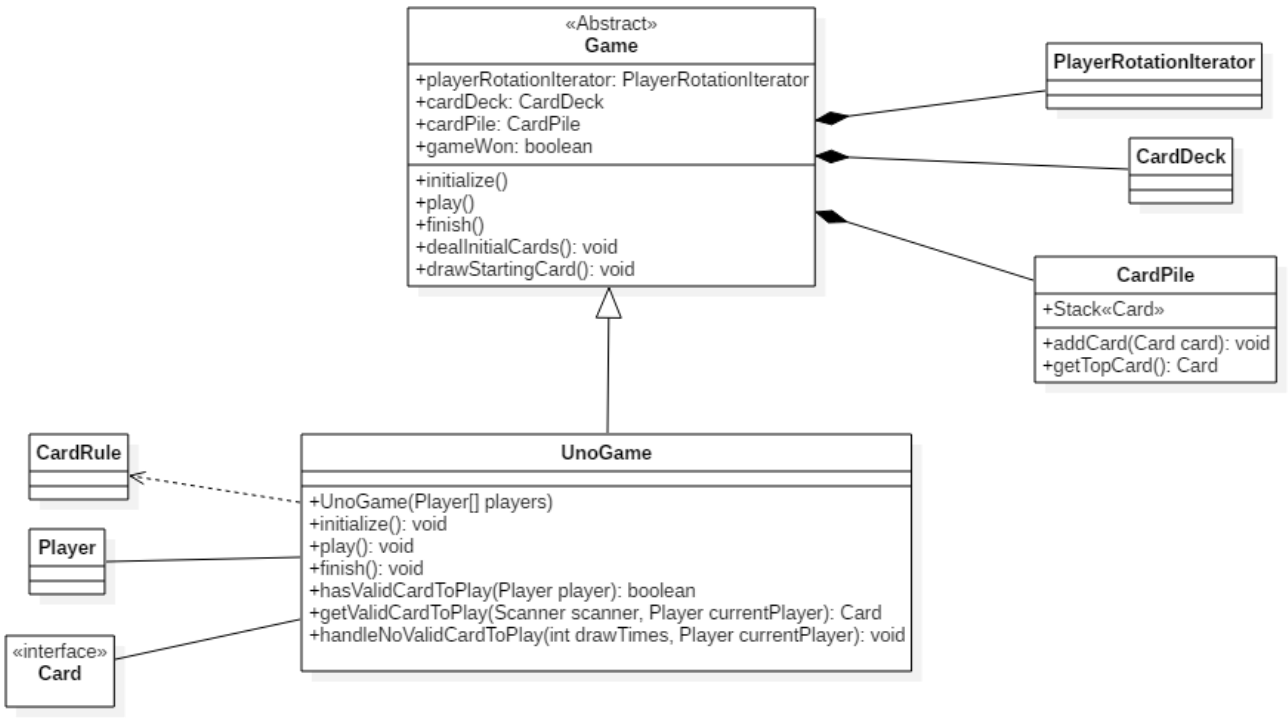
Card Rules Package Diagram



Player Package Diagram



Game Package Diagram



Explanation of The Class Diagrams

Card Package:

- **CardType (enum):** Represents the different types of cards in the game. It includes constants for NUMBER, SKIP, REVERSE, DRAW_TWO, WILD_COLOR, and WILD_DRAW_FOUR. This enum is used to categorize and identify the type of a card.

- **CardValidator (class):** This class provides validation methods for card properties. It has private constructors to prevent instantiation as it only contains static methods. The class includes methods like **validateColor**, which checks if a card color is defined, **validateNumber**, which ensures that a card's number is within the valid range (0-9), and **validateActionType**, which validates the action type of a card. It also includes **isWildCard** method to determine if a card is a wild card.

- **CardColor (enum):** Represents the colors of cards in the game. It defines constants for RED, BLUE, GREEN, YELLOW, and BLACK. This enum is used to specify and identify the color of a card.

- **Card (interface):** This interface defines the common behavior for all types of cards in the game. It declares two methods: **getType**(), which returns the type of the card, and **getColor**(), which returns the color of the card. All card classes, including **NumberedCard**, **WildCard**, and **ActionCard**, implement this interface and provide their own implementations for these methods.

- **NumberedCard (class):** Represents a numbered card in the game. It implements the Card interface and has properties for the card's color and value. The constructor takes a **CardColor** and an integer value to initialize these properties. It provides getter methods for the value and color, and overrides the **equals**(), **hashCode**(), and **toString**() methods for proper comparison and representation of the card.

- **WildCard (class):** Represents a wild card in the game. It implements the Card interface and has properties for the card's type and color. The constructor can take either a **CardType** and a **CardColor** or only a **CardType** to initialize these properties. The class includes getter methods for the type and color, and overrides the **equals**(), **hashCode**(), and **toString**() methods for proper comparison and representation of the card.

- **ActionCard (class):** Represents an action card in the game. It implements the Card interface and has properties for the card's type and color. The constructor uses **CardType** and a **CardColor** to initialize these properties. It includes getter methods for the type and color, and overrides the **equals**(), **hashCode**(), and **toString**() methods for proper comparison and representation of the card.

Card Deck Package:

- **CardFactory (class):** This class is responsible for creating different types of cards based on the provided parameters. It has two factory methods: **createCard(type, color)** for creating action and wild cards, and **createCard(color, value)** for creating numbered cards. It uses the **CardType** and **CardColor** Enums to determine the card type and color.

- **InitializationDeckStrategy (interface):** This is an interface that defines the contract for initializing the card deck. It has a single method, initialize, which takes a **list of cards** and a **CardFactory** as parameters and initializes the card deck according to a specific strategy.

- **DefaultStrategy (class):** This class implements the InitializationDeckStrategy interface. It is responsible for initializing the card deck with the default set of cards, including numbered cards, action cards, and wild cards. The initialize method creates and adds the appropriate cards to the provided list of cards using the CardFactory.

- **CardDeck (class):** This class represents a deck of cards. It contains a **list of cards**, a **CardFactory**, and an **InitializationDeckStrategy**. The constructor takes an **InitializationDeckStrategy** as a parameter to specify the strategy for initializing the card deck. If no strategy is provided, it uses the **DefaultStrategy**. The class provides methods for **adding cards** to the deck, **shuffling** the deck, **drawing** a card from the deck, and **getting the deck size**. The **initializeDeck** method is called during construction to initialize the deck using the specified strategy. The **toString** method returns a string representation of the cards in the deck.

These classes work together to create and manage a deck of cards, providing methods for initialization, shuffling, drawing, and retrieving the cards in the deck. The InitializationDeckStrategy allows for different strategies to be used for initializing the deck, providing flexibility and customization.

Card Rules Package:

- **CardPlayedRule (interface):** This interface defines the contract for card playing rules. It declares a single method **isValidCardToPlay**() that takes the top card and the played card as parameters and returns a boolean value indicating whether the played card is valid to play.

- **WildCardRule (class):** This class implements the **CardPlayedRule** interface and defines the rules for playing a Wild Card. It checks if the played Wild Card has a specified color.

- **NumberedCardRule (class):** This class implements the **CardPlayedRule** interface and defines the rule for playing a Numbered Card. It checks if the played Numbered Card has the same color as the top card or if it has the same value as the top Numbered Card.

- **ActionCardRule (class):** This class implements the CardPlayedRule interface and defines the rule for playing an Action Card. It checks if the played Action Card has the same color as the top card or if it has the same type as the top card.

- **CardRule (class):** This class is a utility class that serves as a central hub for all card playing rules. It provides methods to determine if a card is valid to play based on the top card and the played card. It internally uses instances of **NumberedCardRule**, **ActionCardRule**, and **WildCardRule** to validate the specific card types.

These classes work together to enforce the rules of playing different types of cards in the game. The **CardRule** class acts as a facade, providing a unified interface to validate any type of card based on the specific rules implemented in the other classes.

Player Package:

- ### HandCard (class):

  This class represents a player's hand of cards. It has the following methods:

  - **addCard(Card newCard):** Adds a card to the hand.
  - **removeCard(Card card):** Removes a card from the hand.
  - **findCardOfType(CardType type):** Finds a card of the specified type in the hand.
  - **getCardStream():** Returns a stream of cards in the hand.
  - **hasCard(Card card):** Checks if the hand contains a specific card.
  - **size():** Returns the number of cards in the hand.
  - **toString():** Returns a string representation of the hand.

- ### Direction (enum):

  This enum represents the direction of player rotation in the game. It has two values: **CLOCKWISE** and **COUNTER_CLOCK_WISE**.

- ### Player (class):

  This class represents a player in the game. It has the following methods:

  - **Player(String name):** Constructs a player with the given name.
  - **getName():** Returns the player's name.
  - **addToHandCards(Card card):** Adds a card to the player's hand.
  - **removePlayedCard(Card card):** Removes a card from the player's hand.
  - **hasHandCard(Card card):** Checks if the player's hand contains a specific card.
  - **getHandCards():** Returns the player's hand of cards.
  - **toString():** Returns a string representation of the player, including their name and hand of cards.

- ### PlayerRotationIterator (class):

  This class implements the Iterable interface and provides an iterator over a collection of players. It has the following methods:

  - **PlayerRotationIterator(Player[] players):** Constructs an iterator with the given array of players.
  - **stream():** Returns a stream of players.
  - **getCurrentPlayer():** Returns the current player in the rotation.
  - **reverseDirection():** Reverses the direction of player rotation.
  - **nextPlayer():** Moves to the next player in the rotation and returns it.

Game Package:

- **Game (abstract class):**
  The Game class is an abstract class that serves as the base for specific game implementations, such as UnoGame. It contains common functionality and defines abstract methods that need to be implemented by its subclasses. The key attributes and methods of the Game class are as follows:
  - **playerRotationIterator**: A PlayerRotationIterator object that manages the rotation of players in the game.
  - **cardDeck**: A CardDeck object representing the deck of cards used in the game.
  - **cardPile**: A CardPile object representing the pile of played cards in the game.
  - **gameWon**: A boolean variable indicating whether the game has been won.
  - **initialize**(): An abstract method that initializes the game.
  - **play**(): An abstract method that handles the main game loop and gameplay logic.
  - **finish**(): An abstract method that is called when the game finishes.
  - **dealInitialCards()** method is responsible for distributing a specific number of initial cards to each player in the game. It iterates over the players in the ***playerRotationIterator*** and, for each player, draws a card from the ***cardDeck*** and adds it to the player's hand using the ***addToHandCards*()** method. This process is repeated for the desired number of initial cards.
  - **drawStartingCard()** method draws the starting card for the game. It ensures that the drawn card is a NumberedCard by repeatedly drawing cards from the deck until a NumberedCard is obtained. This ensures that the starting card meets the game's requirement.

- **CardPile (Class):**
  The CardPile class represents a pile of cards in the game. It utilizes a Stack data structure to store the cards. It has two main methods:
  - **addCard(Card card):** Adds a card to the pile.
  - **getTopCard():** Retrieves the top card from the pile without removing it.

- **UnoGame (Class):**
  The UnoGame class is a concrete implementation of the Game abstract class specifically for the Uno game. It extends the Game class and overrides its abstract methods. The key aspects of the UnoGame class are as follows:
  - **UnoGame(Player[] players)**: A constructor that takes an array of Player objects and initializes the player rotation iterator, card deck, card pile, and gameWon status.
  - **initialize()**: Overrides the initialize method from the Game class. It deals initial cards to players and draws the starting card.

- **play()**: Overrides the play method from the Game class. It implements the main game loop, where players take turns, play cards, enforce game rules, and check for game completion conditions. It prompts the current player for their card choice, validates the card, performs the appropriate actions, and checks if the player has won the game.
- **finish():** Overrides the finish method from the Game class. It displays a message indicating the end of the game.
- **Helper methods**: The class includes several helper methods such as **hasValidCardToPlay**(), **getValidCardToPlay**(), and **handleNoValidCardToPlay**() to assist in card validation, player actions, and game progression.

Clean Code

I'll talk about Robert C. Martin's definitions of clean code principles and code smells in this section.

1. **Comments**
   In my code, I used comments sparingly and focused on adding comments where they were necessary to clarify certain aspects of the code. I followed the principle of writing self-explanatory code by using meaningful variable and function names, which reduced the need for extensive commenting.
2. **Duplication**
   To minimize duplication, I followed the DRY (Don't Repeat Yourself) principle in my code. In the **DefaultStrategy** class, I utilized loops and the **CardFactory** to dynamically generate the different types of cards for deck initialization. This approach avoided manual repetition of code and allowed for easier maintenance and scalability. By iterating through the CardColor values, I created numbered cards, action cards, and wild cards, reducing the need for repetitive code blocks. This approach improved code readability and reduced the chances of introducing bugs caused by duplicated logic.

```
public class CardFactory {
    5 usages
    public static Card createCard(CardType type, CardColor color) {
    switch (type) {
        case SKIP:
        case REVERSE:
        case DRAW_TWO:
            return new ActionCard(type, color);
        case WILD_COLOR:
        case WILD_DRAW_FOUR:
            return new WildCard(type);
        default:
            throw new IllegalArgumentException("Invalid card type: " + type);
    }
}
    3 usages
    public static Card createCard(CardColor color , int value) { return new NumberedCard(color, value); }
}
```

3. Code at wrong level of abstraction

This principle is applied in a good manner as I implement abstraction when needed, Like the **CardPlayedRule** interface is the super type for any class that implements it .

```
public interface CardPlayedRule {
    6 usages  3 implementations
    boolean isValidCardToPlay(Card topCard, Card playedCard);
}
```

4. Names

- **Choose descriptive name.**

  I applied this principle for all methods and variables so that they describe their jobs.

```
private List<Card> cards;
2 usages
private CardFactory cardFactory;
2 usages
private InitializationDeckStrategy initializationDeckStrategy ;
```

```
private final String playerName;
6 usages
private HandCard handCards ;
```

- **Choose name at appropriate level of abstraction.**

  i applied this principle for all classes, methods, and variables so that they describe their jobs, with taking care about Java naming conventions level of abstraction

```
public Player getCurrentPlayer() { return players[currentPlayer]; }

1 usage
public void reverseDirection() { direction = Direction.COUNTER_CLOCK_WISE; }

1 usage
public Player nextPlayer() {...}
```

5. Functions

- **Too many arguments**

  Methods in the project follow the principle of minimizing the number of arguments, typically limiting it to two or fewer. This practice improves code readability and maintainability. Although some functions may require additional arguments due to the nature of the operation they perform, the overall goal is to minimize the occurrence of methods with excessive arguments. By doing so, the codebase remains more concise and easier to understand.

- **Flag argument**
  This principle is applied for all methods in the project, as I implement all methods in single responsibility principle, so every method does only one job.

- **Dead function**
  This principle is applied for all methods in the project, as all method implemented are used in the code.

6. Enums and Annotations

Enums like CardType and CardColor improve code clarity and organization by providing a clear and readable way to represent card types and colors. Annotations, although not explicitly used in the provided code, have the potential to enhance code organization and enable automated processes.

```
public enum CardColor {
    no usages
    RED, BLUE, GREEN, YELLOW , BLACK
}
```

```
public enum CardType {
    2 usages
    NUMBER, SKIP, REVERSE, DRAW_TWO, WILD_COLOR, WILD_DRAW_FOUR
}
```

```
public enum Direction {
    2 usages
    CLOCKWISE, COUNTER_CLOCK_WISE
}
```

SOLID design principles

- **Single Responsibility**
  "A class should only have one responsibility. Furthermore, it should only have one reason to change."
  This principle is applied to all classes and methods as they have only one job to do. Classes are built carefully, taking care of each one and its responsibility.

- **Open/close principle**

  I have followed the Open/Closed Principle (OCP) in my codebase. An example of this can be seen in the implementation of the **CardPlayedRule** interface and its various implementations, such as the **WildCardRule**. By using this approach, I have designed the code to be open for extension, allowing for the addition of new card validation rules without modifying the existing code. This adherence to the OCP promotes code reusability, maintainability, and scalability.

```
public interface CardPlayedRule {

    boolean isValidCardToPlay(Card topCard, Card playedCard);
}
```

- **Liskov Substitution**

  I have also applied the Liskov Substitution Principle (LSP) in my code. For instance, all the classes that implement the **Card** interface, including **ActionCard**, **NumberedCard**, and **WildCard**, can be used interchangeably without affecting the correctness of the program. This adherence to the LSP ensures that any subtype can be substituted for its base type, promoting modularity, extensibility, and compatibility within the codebase.

- **Interface segregation**

  "Larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them."
  There is no larger interface in the project, all interface contains only the methods that should be implemented by their subtypes.

```
public interface InitializationDeckStrategy {
    1 usage  1 implementation
    void initialize(List<Card> cards, CardFactory cardFactory);
}
```

- **Dependency injection**

  "A class should not depend on low-level concrete classes; instead, it should depend on abstractions. In other words, client classes should depend on an interface or abstract class, rather than a concrete resource."
  One example of Dependency Injection in my code is in the DefaultStrategy class. The DefaultStrategy relies on a CardFactory instance to create cards during the initialization of the deck. By injecting the CardFactory dependency into the DefaultStrategy class, I decouple the creation of cards from the strategy itself. This allows for greater flexibility, as different implementations of CardFactory can be easily substituted without modifying the DefaultStrategy class. This promotes modularity and simplifies testing, as I can provide a mock or stub CardFactory during unit testing to isolate the behavior of the DefaultStrategy class.

Design Patterns

Design patterns are widely accepted solutions to common problems in software development. They provide best practices and guidelines for structuring code and solving recurring design challenges. There are three main types of design patterns: Creational Patterns, Structural Patterns, and Behavioral Patterns. In this section, I will discuss the design patterns used in the implementation of the Uno game.

- **Factory Method**
  The Factory Method design pattern is a creational pattern that provides an interface for creating objects in a superclass, while allowing subclasses to determine the exact type of objects to be created. In the Uno game, I applied this pattern to create various types of cards, such as numbered cards, action cards, and wild cards, without specifying their concrete classes explicitly. By using the Factory Method, I achieved flexibility in creating different types of cards based on the game's requirements, and it also simplifies the addition of new card types in the future.

- **Strategy Pattern**
  The Strategy design pattern is a behavioral pattern that enables objects to change their behavior dynamically by encapsulating interchangeable algorithms. In the context of the Uno game, each card has a specific behavior or rule associated with it. To handle the different behaviors of cards, I utilized the Strategy pattern. This pattern allowed me to define various strategies or rules for different card types, such as determining the validity of a card play or the actions to be taken based on the played card. By employing the Strategy pattern, the game can easily adapt to new card types or modify existing card behaviors without affecting the core game logic.
  Applying the Strategy pattern in the Uno engine enhances code maintainability, extensibility, and adaptability. It allows for easier addition of new card behaviors and facilitates future changes in the game's requirements without the need to modify existing code extensively.

Thank you for playing Uno!

```
Player 1 played ActionCard :REVERSE, RED
The direction has been reversed, and the next player is Player 2
Player 1 has won the game!
Game finished. Thank you for playing Uno!
```