Training Program
Assignment #5
Uno Game Assignment Report

Instructor: Prof. Motasem Al-Diab

Done by: Ahmad Al-Sou'b

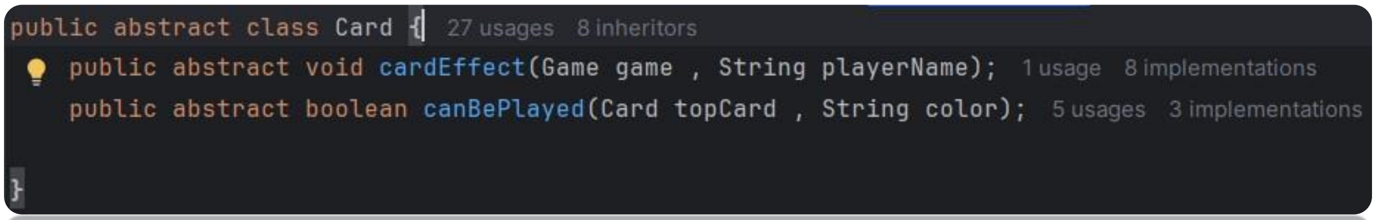
TABLE OF CONTENTS

ABSTRACTION

The engine abstracts Uno's complexity through strategic use of inheritance, polymorphism, and encapsulation:

1. Class Hierarchy for Cards

- **Card:** In Figure **1**, The abstract base class defines shared behavior (e.g., cardEffect(), canBePlayed()).
    - **ActionCard:** Handles cards like Reverse or Skip, overriding cardEffect() to implement turn-altering logic.
    - **WildCard:** Manages color-changing cards (e.g., Wild or Wild Draw Four), with effects decoupled from specific game rules.



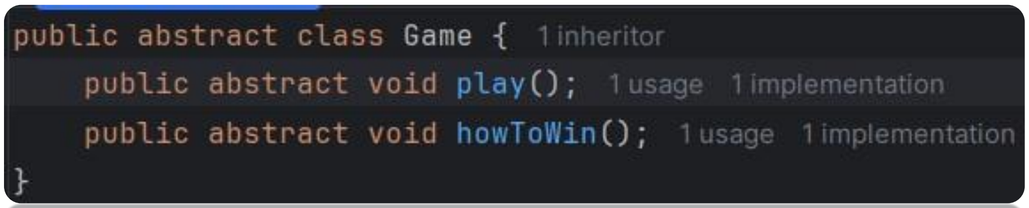
```
public abstract class Card {  27 usages  8 inheritors
    public abstract void cardEffect(Game game , String playerName);  1 usage  8 implementations
    public abstract boolean canBePlayed(Card topCard , String color);  5 usages  3 implementations

}
```

Figure 1

2. Game Logic via the Game Template

- In Figure **2**, The abstract Game class serves as a template for all Uno variations. It defines:
    - Core mechanics (e.g., play(), skipPlayer(), reverseDirection()).
    - Abstract methods (e.g., howToWin()) for subclasses to implement custom win conditions.



```
public abstract class Game {  1 inheritor
    public abstract void play();  1 usage  1 implementation
    public abstract void howToWin();  1 usage  1 implementation
}
```

Figure 2

3. Configurable Components

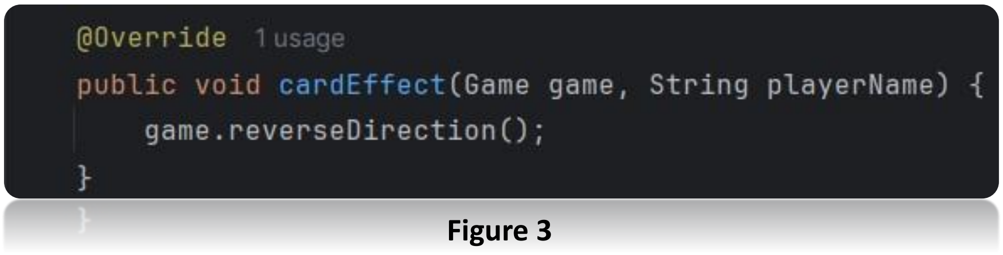
- DeckInitializer Interface: Decouples deck creation from game logic, allowing developers to define custom card sets (e.g., holiday-themed decks).
- Player Class: Encapsulates player-specific actions (e.g., choosing cards, declaring UNO), ensuring state management remains isolated from game flow.

4. Encapsulation of State

- Private Fields: Critical data like playerCards (in Player) and tableCard (in ClassicUno) are encapsulated, with access restricted to controlled methods (e.g., getPlayerCards()).
- Immutable Objects: Cards (e.g., NumberCard) are immutable after creation, preventing unintended state changes.

5. Polymorphic Behavior

- In Figure **3**, Dynamic Dispatch: The cardEffect() method behaves differently across card types (e.g., ReverseCard reverses turn order, while WildAddFour forces the next player to draw cards).



```
@Override  1 usage
public void cardEffect(Game game, String playerName) {
    game.reverseDirection();
}
```

Figure 3

INTRODUCTION

The Uno game engine is designed to provide a flexible, extensible framework for developers to create custom variations of the classic Uno card game. Built using Java and object-oriented programming (OOP) principles, this engine emphasizes modularity, scalability, and adherence to design patterns like Strategy, Template Method, and Factory. By abstracting core game mechanics (e.g., card effects, turn management, and win conditions) into reusable components, the engine allows developers to focus on implementing unique rules or features without rewriting foundational logic.

- **Key objectives include:**
  1. **Extensibility:** Developers can create new card types, game modes, or rules by extending abstract classes (e.g., Card, Game).
  2. **Decoupling:** Separation of concerns (e.g., DeckInitializer for card generation, Player for turn logic) ensures components remain independent and testable.
  3. **Adherence to Best Practices:** The codebase follows Clean Code principles (e.g., meaningful naming, small functions) and SOLID design principles (e.g., dependency inversion via interfaces)
- This report evaluates the engine’s architecture through the lens of design patterns, Effective Java guidelines, and Clean Code principles, demonstrating how it balances flexibility with robustness.

UML DIAGRAM

A UML Class Diagram is one of the main diagrams in Unified Modeling Language (UML) and is used to design software systems based on Object-Oriented Programming (OOP) principles. It visually represents the structure of a system by showing classes, attributes, methods, and relationships between different objects.

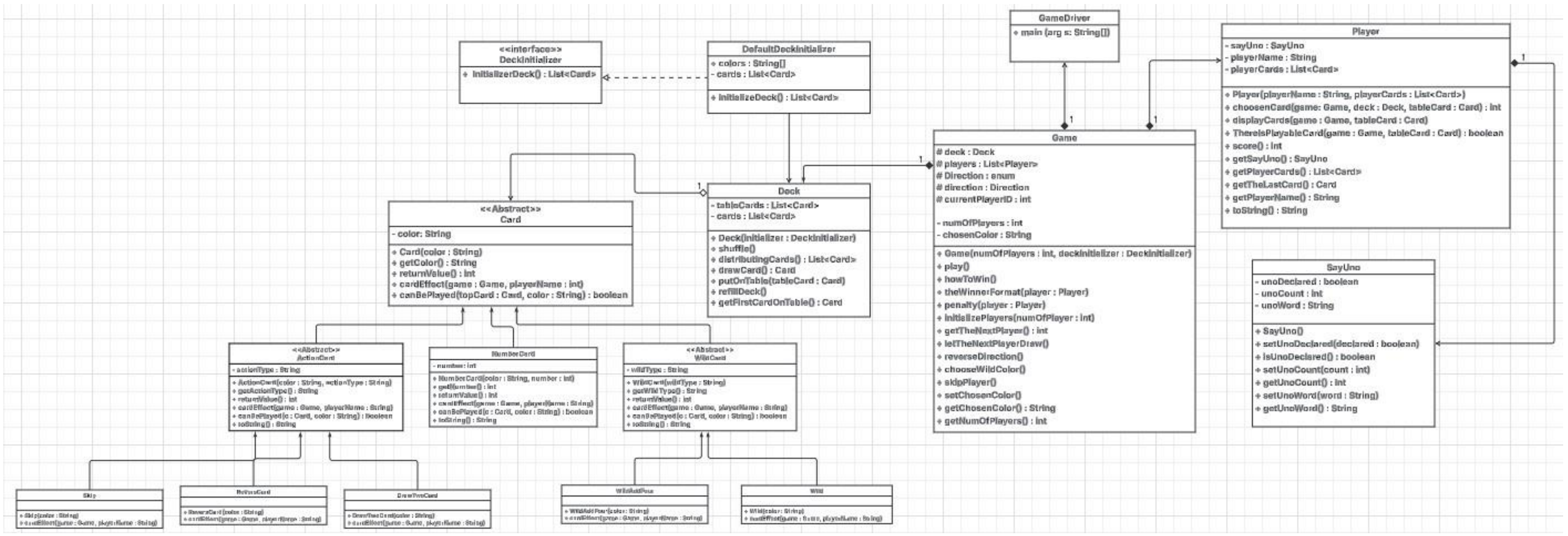
❖ Key Components of a UML Class Diagram:

1. **Class:** Represented as a rectangle divided into three sections:
   - Class Name (at the top).
   - Attributes (Fields/Variables) (in the middle).
   - Methods (Functions/Behaviors) (at the bottom).
2. **Access Modifiers:**
   - + (Public): Can be accessed from anywhere.
   - - (Private): Can only be accessed within the class.
   - # (Protected): Can be accessed within the class and its subclasses.
3. **Relationships Between Classes:**
   - Association (→): A basic relationship where one class uses another.
     Aggregation (◇—): A "has-a" relationship, where a class contains another but can exist independently.
   - Composition (◆—): A stronger "owns" relationship, where if the container class is destroyed, the contained class is too.
   - Inheritance (⋆—▷ or ▲—▷): Represents the "is-a" relationship, where one class extends another (e.g., Dog inherits from Animal).
   - Dependency (—> dashed line): A class depends on another temporarily (e.g., passing an object as a method parameter).

❖ List of UML Diagram Types:

1. **Structure Diagrams:**
   - Class Diagram.
   - Component Diagram.
   - Deployment Diagram.
   - **Object Diagram.**
   - Package Diagram.
   - Profile Diagram.
   - Composite Structure Diagram.
2. **Behavioral Diagrams:**
   - Use Case Diagram.
   - Activity Diagram.
   - State Machine Diagram.
   - Sequence Diagram.
   - Communication Diagram.
   - Interaction Overview Diagram.
   - Timing Diagram.

❖ UML Diagram OOP Design



<<interface>> DeckInitializer
+ InitializerDeck() : List<Card>
DefaultDeckInitializer
+ colors : String[]
- cards : List<Card>
+ initializeDeck() : List<Card>
GameDriver
+ main (arg s: String[])
Player
- sayUno : SayUno
- playerName : String
- playerCards : List<Card>
+ Player(playerName : String, playerCards : List<Card>)
+ choosenCard(game: Game, deck : Deck, tableCard : Card) : int
+ displayCards(game : Game, tableCard : Card)
+ ThereIsPlayableCard(game : Game, tableCard : Card) : boolean
+ score() : int
+ getSayUno() : SayUno
+ getPlayerCards() : List<Card>
+ getTheLastCard() : Card
+ getPlayerName() : String
+ toString() : String
Game
# deck : Deck
# players : List<Player>
# Direction : enum
# direction : Direction
# currentPlayerID : int
- numOfPlayers : int
- chosenColor : String
+ Game(numOfPlayers : int, deckInitializer : DeckInitializer)
+ play()
+ howToWin()
+ theWinnerFormat(player : Player)
+ penalty(player : Player)
+ initializePlayers(numOfPlayer : int)
+ getTheNextPlayer() : int
+ letTheNextPlayerDraw()
+ reverseDirection()
+ chooseWildColor()
+ skipPlayer()
+ setChosenColor()
+ getChosenColor() : String
+ getNumOfPlayers() : int
SayUno
- unoDeclared : boolean
- unoCount : int
- unoWord : String
+ SayUno()
+ setUnoDeclared(declared : boolean)
+ isUnoDeclared() : boolean
+ setUnoCount(count : int)
+ getUnoCount() : int
+ setUnoWord(word : String)
+ getUnoWord() : String
Deck
- tableCards : List<Card>
- cards : List<Card>
+ Deck(initializer : DeckInitializer)
+ shuffle()
+ distributingCards() : List<Card>
+ drawCard() : Card
+ putOnTable(tableCard : Card)
+ refillDeck()
+ getFirstCardOnTable() : Card
<<Abstract>> Card
- color: String
+ Card(color : String)
+ getColor() : String
+ returnValue() : int
+ cardEffect(game : Game, playerName : int)
+ canBePlayed(topCard : Card, color : String) : boolean
<<Abstract>> ActionCard
- actionType : String
+ ActionCard(color : String, actionType : String)
+ getActionType() : String
+ returnValue() : int
+ cardEffect(game : Game, playerName : String)
+ canBePlayed(c : Card, color : String) : boolean
+ toString() : String
NumberCard
- number : int
+ NumberCard(color : String, number : int)
+ getNumber() : int
+ returnValue() : int
+ cardEffect(game : Game, playerName : String)
+ canBePlayed(c : Card, color : String) : boolean
+ toString() : String
<<Abstract>> WildCard
- wildType : String
+ WildCard(wildType : String)
+ getWildType() : String
+ returnValue() : int
+ cardEffect(game : Game, playerName : String)
+ canBePlayed(c : Card, color : String) : boolean
+ toString() : String
Skip
+ Skip(color : String)
+ cardEffect(game : Game, playerName : String)
ReverseCard
+ ReverseCard(color : String)
+ cardEffect(game : Game, playerName : String)
DrawTwoCard
+ DrawTwoCard(color : String)
+ cardEffect(game : Game, playerName : String)
WildAddFour
+ WildAddFour(color : String)
+ cardEffect(game : Game, playerName : String)
Wild
+ Wild(color : String)
+ cardEffect(game : Game, playerName : String)
1
1
1
1
1
1

Figure 4

OBJECT ORIENTED DESIGN

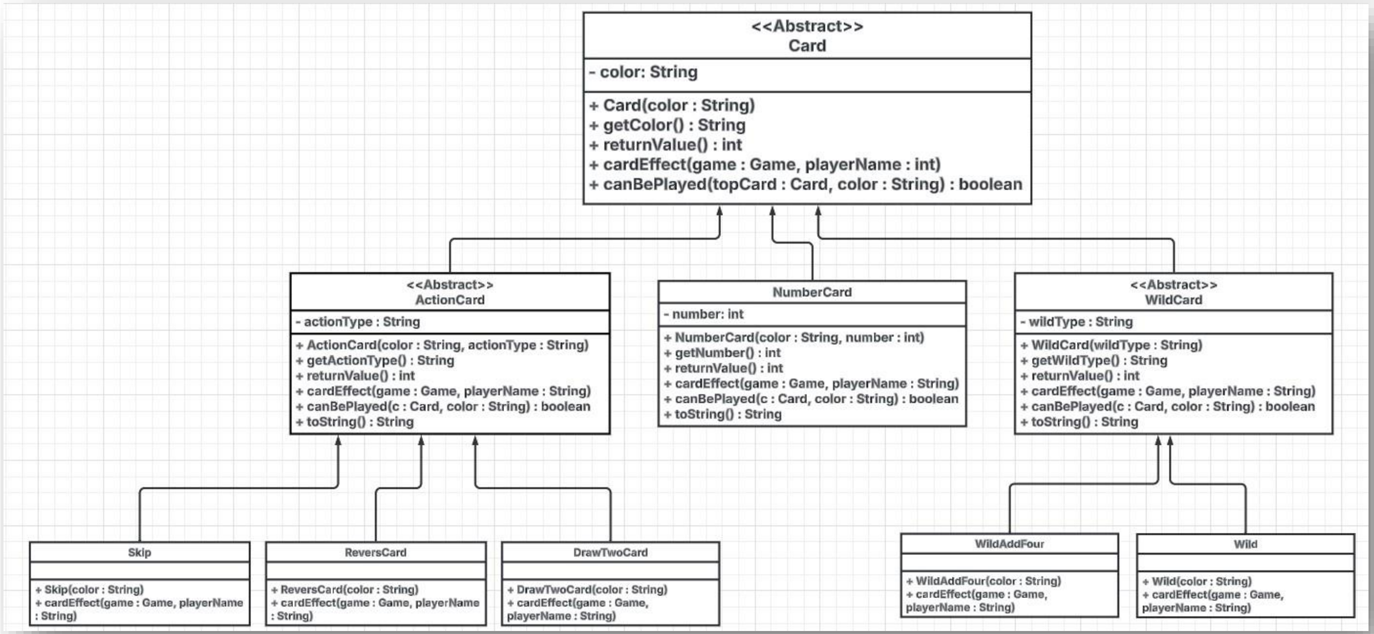
➢ **Classes Structure and Responsibility:**

A.Card:

This class is an abstract card class that serves as the superclass for all card types. It has three subclasses: NumberCard, and two additional abstract classes named ActionCard and Wildcard. This structure allows developers to create their own Wildcard and ActionCard classes with custom effects by extending these base classes. In my implementation, I included only the most common cards: two types of Wildcards (Wild and Wild Draw Four) and three ActionCards (Skip, Reverse, Draw Two), each represented by its own class, utilizing the Factory Method Design Pattern. These classes provide a standardized way to manage these cards, ensuring consistency across various Uno versions and contains the following main method:

- **cardEffect():** This method is invoked when the player chose a valid card , so it is used to activate the card's effect.
- **canBePlayed() :** This method check if the player play a playable card.

And the is the UML diagram for the Card Class at figure 5:



<<Abstract>>
Card
- color: String
+ Card(color : String)
+ getColor() : String
+ returnValue() : int
+ cardEffect(game : Game, playerName : int)
+ canBePlayed(topCard : Card, color : String) : boolean
<<Abstract>>
ActionCard
- actionType : String
+ ActionCard(color : String, actionType : String)
+ getActionType() : String
+ returnValue() : int
+ cardEffect(game : Game, playerName : String)
+ canBePlayed(c : Card, color : String) : boolean
+ toString() : String
NumberCard
- number: int
+ NumberCard(color : String, number : int)
+ getNumber() : int
+ returnValue() : int
+ cardEffect(game : Game, playerName : String)
+ canBePlayed(c : Card, color : String) : boolean
+ toString() : String
<<Abstract>>
WildCard
- wildType : String
+ WildCard(wildType : String)
+ getWildType() : String
+ returnValue() : int
+ cardEffect(game : Game, playerName : String)
+ canBePlayed(c : Card, color : String) : boolean
+ toString() : String
Skip
+ Skip(color : String)
+ cardEffect(game : Game, playerName : String)
ReversCard
+ ReversCard(color : String)
+ cardEffect(game : Game, playerName : String)
DrawTwoCard
+ DrawTwoCard(color : String)
+ cardEffect(game : Game, playerName : String)
WildAddFour
+ WildAddFour(color : String)
+ cardEffect(game : Game, playerName : String)
Wild
+ Wild(color : String)
+ cardEffect(game : Game, playerName : String)

Figure 5

B. Deck:

This class is designed to compile all the necessary Uno game cards into a deck, stored in an ArrayList of cards. It includes the following main methods:

- **distributingCards() :** this method is invoked when the game start , so each player will get his initial cards.
- **Shuffle() :** this method use to shuffle the cards.
- **drawCard() :** this method invoked each time the player want to draw a card.
- **putOnTable() :** this method will take the chosen card and put it in discarded Cards.
- **getFirstCardOnTable() :** this method is invoked when the game starts to choose the initial card will be drawn and it should be a numbered card.

This class initializes the deck with either a user-provided card or a default card. I have created an interface called DeckInitializer to allow each developer to customize the deck based on their specific card requirements.

C.Player:

The Player class manages the state of a player in a card game. It enables the player to select a card to play based on their hand and the current table card. The class handles card drawing when no playable cards are available and categorizes the player's cards based on their playability. Additionally, it tracks the player's score and provides information about the player and contains the following main methods:

- **chooseCard() :** this method use to let the player choose the card that he wants to play (only if he can play a card) by entering the index of the card and check that he enter a right index.
- **displayCards() :** this method use to display the player’s cards.
- **ThereIsPlayanleCard() :** this method use to check if the user can play any card from his cards.
- **Score():** this method for calculating the player score at the end of the game.
- **getSayUno() :** the method use to return to a SayUno Class.

D.SayUno:

This class is responsible for verifying whether a player has called "Uno" as required. It is used to determine the winner and to assess any penalties if the player fails to announce “Uno” and contains the following main methods:

- **setUnoDeclared():** this method use to declared if the player said Uno or not to check who is the winner in the game .
- **setUnoCount() :**this counter help me to know when the player said UNO (When he have one card or two cards).
- **setUnoWord() :** this method use to let the next player draw a card.
- **getUnoWord() :** this method will take the input from the player and check it is a “UNO” or Enter.

E. Game:

This class is abstract and designed for developers to extend to create their own game-specific classes. It includes the following main methods:

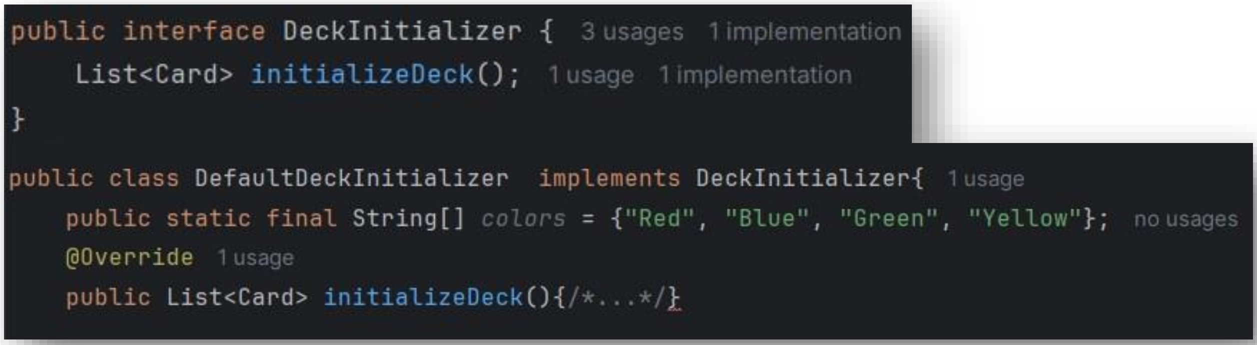
- **play() :** this method is abstract to let the developer choose how he wants to play.
- **howToWin() :** another abstract method , to determine the winning rules of the game.
- **theWinnerFormat() :** to print the format of the winner and the scores for the other playes.
- **Penalty(player : Player) :** this use to let the player who did not say “UNO” when he have one card to draw 2 cards.
- **initializePlayers() :** it takes the number of players as input , takes the names of the players and initializes them.
- **getTheNextPlayer():** this method is used to know who is the next player.
- **letTheNextPlayerDraw() :** this method use to let the next player draw a card.
- **reverseDirection () :** this method use to change the direction of the game.
- **chooseColorIfWild() :** this method makes the player choose the color of the next player if he played a one card.
- **skipPlayer() :** this method skip the current player to next one.

I Have created my own game in the ClassicUno class, then I created some object from it in the DefaultDeckInitializer class that implements the DeckInitializer interface.

DESIGN PATTERNS

I utilized multiple design patterns to make my code more flexible, readable, and easy to modify in the future by other developers. The following are the design patterns I implemented in my code:

- **Mediator Pattern:** The Game class acts as a centralized hub, coordinating interactions between all other classes. This design ensures that the classes are fully encapsulated, with communication managed through the mediator, reducing dependencies between them.
- **Strategy Pattern:** The DeckInitializer interface represents the Strategy Pattern by allowing different strategies for deck initialization. Developers can implement various concrete classes to initialize the deck based on specific game requirements, providing flexibility in how the deck is constructed, in figure 6.



```
public interface DeckInitializer {  3 usages  1 implementation
    List<Card> initializeDeck();  1 usage  1 implementation
}
public class DefaultDeckInitializer  implements DeckInitializer{  1 usage
    public static final String[] colors = {"Red", "Blue", "Green", "Yellow"};  no usages
    @Override  1 usage
    public List<Card> initializeDeck(){/*...*/}
```

Figure 6

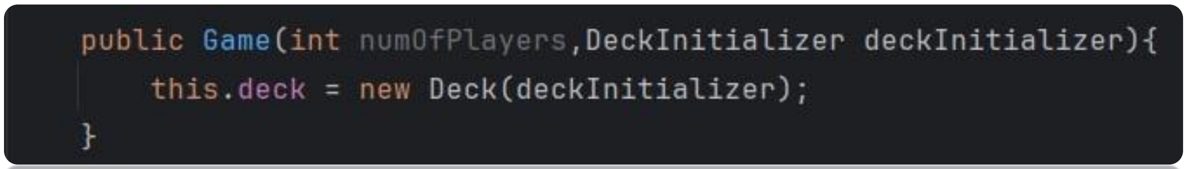
- **Factory Method Pattern:** The abstract Card class, along with its subclasses (NormalCard, ActionCard, SkipCard, ReverseCard, DrawTwoCard Wildcard, Wild, WildDrawFour), demonstrates the use of the Abstract Factory Pattern. Each type of card is treated as a product created by a corresponding concrete factory, enabling easy extension and customization of card types within the game.

- **Template Method Pattern:** The abstract Game class defines the skeleton of the game loop in play(), while allowing subclasses like ClassicUno to override specific steps (e.g., howToWin()), in figure 7 .

DEFEND CODE AGAINST:

Defend Code Against SOLID Principles

- **Single Responsibility Principle (SRP):** Each of the previously discussed classes adheres to the Single Responsibility Principle, with each class representing a distinct component within the Uno game.
- **Open/Closed Principle (OCP):** I followed the Clean Code approach by using comments to make the code easier to understand. I focused on why certain decisions were made and how tricky bits work. I kept comments straightforward and to the point, avoiding any unnecessary details. This way, anyone who looks at the code can quickly get a sense of its purpose and logic without having to dig through everything.
- **Liskov Substitution Principle (LSP):** The implementation of methods in these classes aligns with the Liskov Substitution Principle. The functionality of each method is well-defined and ensures compatibility with its subclasses.
- **Interface Segregation Principle (ISP)**: Small interfaces like DeckInitializer focus on a single responsibility (deck initialization).
- **Dependency Inversion Principle (DIP):** Game depends on the DeckInitializer interface, not concrete implementations in figure 7.



```
public Game(int numOfPlayers,DeckInitializer deckInitializer){
    this.deck = new Deck(deckInitializer);
}
```

Figure 7

Defend Code Against Effective Java Items

A. Item 1: Consider Static Factory Methods

- **Opportunity for Improvement**:
  - ✓ While not explicitly used, the DeckInitializer acts as a factory interface. A static factory method in Deck could further simplify deck creation.

B. Item 15: Minimize Accessibility of Classes and Members

- Fields like color in Card are private final, and access is provided via getColor().
- The SayUno class encapsulates UNO declaration logic, hiding internal state (unoDeclared, unoCount).

C. Item 17: Minimize Mutability

- Card subclasses (NumberCard, WildCard) are immutable after creation (e.g., color is final).
- The Deck class uses immutable lists for cards and tableCards.

D. Item 18: Favor Composition Over Inheritance

- The Game class uses composition (e.g., Deck, Player) instead of inheriting from a concrete class. Subclasses like ClassicUno extend Game but do not violate encapsulation.

Defend Code Against Clean Code Principles

A. Meaningful Names:

In my code, I prioritized using meaningful names for classes, methods, and variables to enhance readability and maintainability. By choosing descriptive and clear names, I made it easier for other developers to understand the purpose and functionality of different components without needing extensive comments or documentation like The Classes (Game, SayUno,Card…) and Methods (canBePlayed, cardEffect….).

B. Functions:

I tried to make all the functions in my code clear and focused on a single task to make the code easier to maintain. Additionally, I ensured that each function handles its responsibilities efficiently, avoiding complex logic that could be broken down further.

C. Comment:

I followed the Clean Code approach by using comments to make the code easier to understand. I focused on why certain decisions were made and how tricky bits work. I kept comments straightforward and to the point, avoiding any unnecessary details. This way, anyone who looks at the code can quickly get a sense of its purpose and logic without having to dig through everything.

D. Formatting:

I kept the code tidy by using consistent indentation and spacing. This makes it easy to follow the structure and quickly spot any issues.

E. Design and structure:

I focused on creating a clear and logical design for the code to make it easy to understand and maintain. The structure is organized so that each component has a well-defined role, and related pieces of functionality are grouped together (high cohesion). I used design patterns to manage complexity and ensure the code can be easily extended or modified in the future and I tried to reduce the dependencies as much as I can (less coupling).

CONCLUSION

The Uno game engine successfully balances flexibility, maintainability, and adherence to software design principles. By leveraging composition over inheritance, encapsulation, and polymorphism, the codebase enables seamless extension for new card types, rules, and game variations while minimizing technical debt. Key achievements include:

1. **Design Pattern Alignment:** Use of Strategy (via DeckInitializer), Template Method (abstract Game class), and Factory patterns ensures modularity and decoupling.
2. **Clean Code Compliance:** Meaningful naming, small-method philosophy, and DRY (Don't Repeat Yourself) principles are largely upheld, though opportunities exist to further decouple I/O logic and reduce function size.
3. **Effective Java Practices:** Immutability (Card subclasses), accessibility control, and abstraction align with Joshua Bloch's guidelines, fostering robustness.

Areas for improvement include refactoring monolithic methods (e.g., play()), enhancing testability via dependency injection, and stricter separation of UI/domain logic. By addressing these, the engine would fully embody Uncle Bob's Clean Code principles and SOLID design, ensuring long-term scalability and ease of innovation. Overall, the implementation demonstrates a strong foundation for building customizable, rule-driven card games while adhering to industry best practices.