# Uno Game Engine

**UNO Game Customization Engine for Developers**


## Outlines:

- **Introduction**
- **OOP design**
- **Design pattern**
- **Code defending against:**
  - **SOLID principle**
  - **Effective Java Items**
  - **Clean Code principle**

# Introduction

The Uno game engine is a **software framework** designed using **native Java** programming language **to facilitate** the development of Uno based applications. **Unlike** a traditional game implementation **aimed at players**, this engine provides an **abstraction** layer that allow **developers to integrate**, **extend**, and **manipulate** game mechanism.

The primary goal is to offer **modular**, **extensible**, and **maintainable** codebase that enables developer **to build custom Uno games without modifying core logic**.

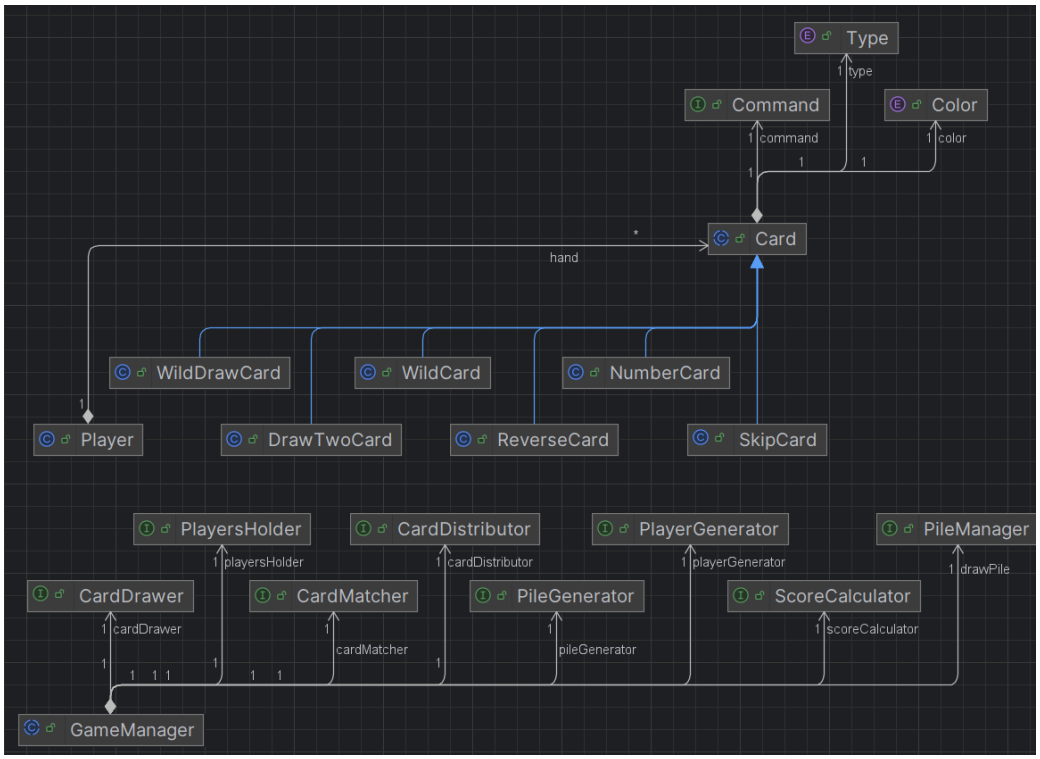
This project was developed with **key software engineering principles** ensuring code **clarity** and **flexibility**. Object oriented programming (**OOP**) serves as the foundation, allowing for **well-structured** and **scalable** design. Additionally, with the help of using **design patterns** to enhance code **reusability** and **maintainability**.

To ensure **industry standards**, the implementation adhere to **Clean Code** principles, **SOLID** design principles, and best practice mentioned in **Effective Java** by Joshua Bloch. **This report** will provide **analysis** of project's OOP design, the design pattern utilized, and justifications for design decisions based on these established software engineering guidelines.

# OOP design

The Uno game engine is structured using OOP principles to ensure modularity, scalability, and maintainability. The design follows encapsulation, inheritance, and polymorphism, allowing developers to extend or modify game rules easily.

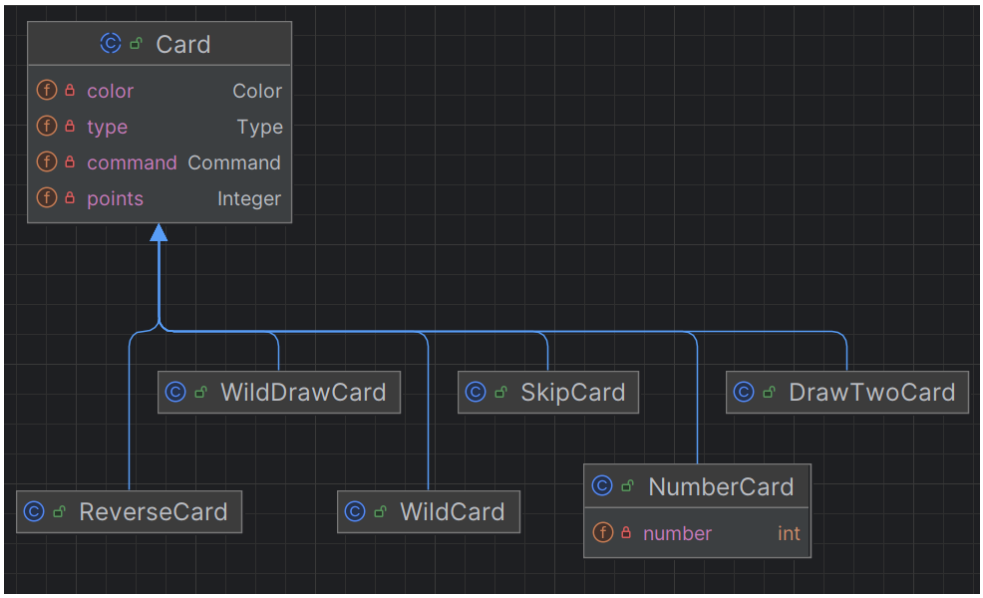
Look at the figure to see the design of the key classes and their relationships, then we will talk about each component independently.



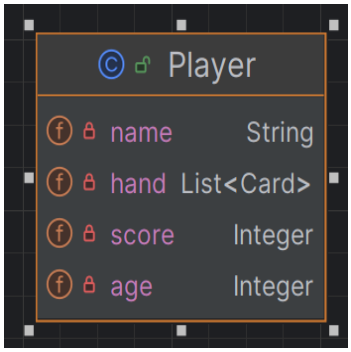

Now let's start to talk about each component dependently

- **Card**
  The abstract class **Card** is the parent class of all cards, and any card will inherit this Card class, but as we know, in UNO game, there are two types of cards, **Action** Cards and **Numbered** cards, and the action cards have specific features that distinguish them from the other cards, so instead of let them implement another interface to override the action method we will use **command pattern** (we will talk about it later) and initialize it with **NoOpCommand** (which do nothing).

With this design we can add **any** new **type** of **card** whether the card has an action or not, and using the **command** can add any type of **action** on the game that I will show later. Each card will have **Color** and **Type** (Enum Objects) and the **points** (Integer) of this card.

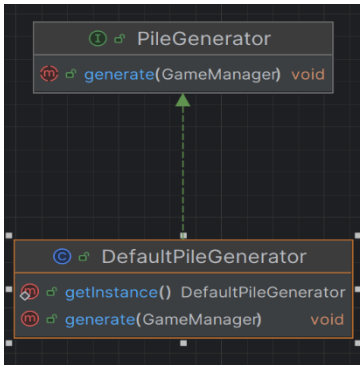



- **Player**
  Each player will have an **age** (Integer), **name** (String), **score** (Integer), and **hand** (List of objects of type card) which represent the current **player's card** in the current round.



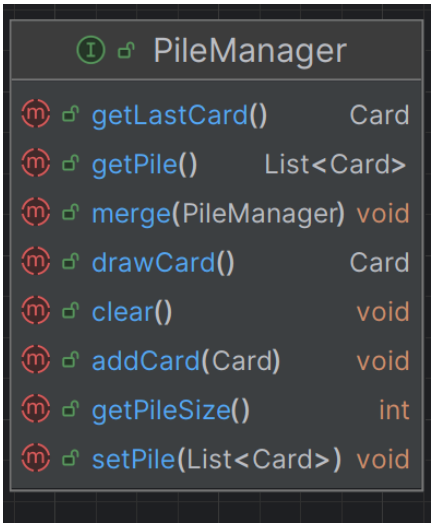

- **Services**
  The services of the game including:
  - Pile Generator:

  The interface **PileGenerator** used to encapsulate the card initialization then put them in the draw pile (which used by the player to draw cards) and then create the discard pile.



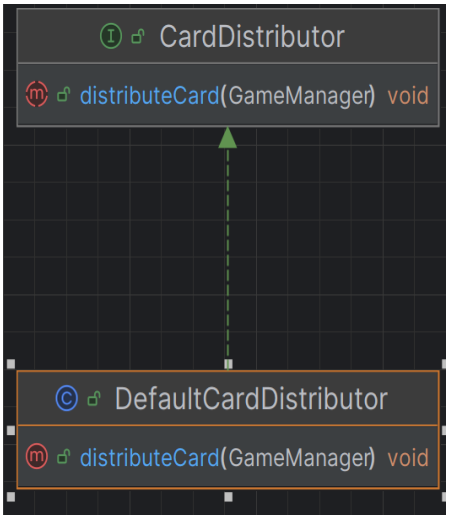
- Pile Manager:

The **PileManager** interface is designed to **manage** the **cards** in both the **draw** and **discard piles**. It **encapsulates** the **complexity** of **handling** these piles by providing a set of functions that must be **overridden** by any class implementing this interface. You can see these functions in the figure.




- Card Distributor:

**CardDistributor** interface that is responsible for distributing the cards over the players throw the method distributeCard, which takes the **gameManger** which hold the card in the **PileManager** object,then distribute these cards across the players which also holds by the gameManager in the **PlayersHolder** object.

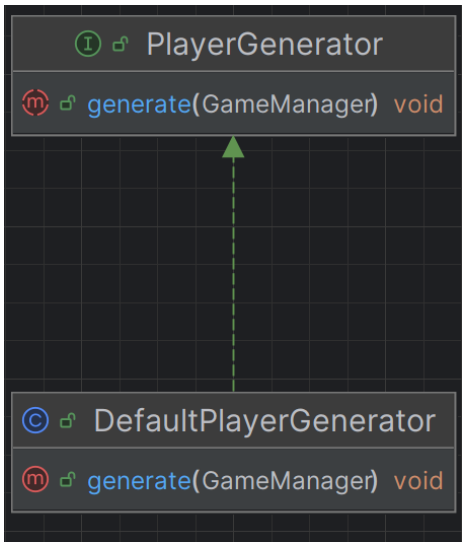



- Players Holder:

The **PlayersHolder** interface is responsible for managing the players in the game, including their order and turn direction. It encapsulate the complexity of handling player interactions and turn management through set of essential function must override when implement it.

- Player Generator:

The **PlayerGenerator** interface it is responsible and encapsulate the complexity for generating players in the game. It defines a method generate that takes a **GameManager** object. This method is used to create players and add them to **PlayresHolder** object within the GameManger, ensuring that the players are initialized and integrated to the game.

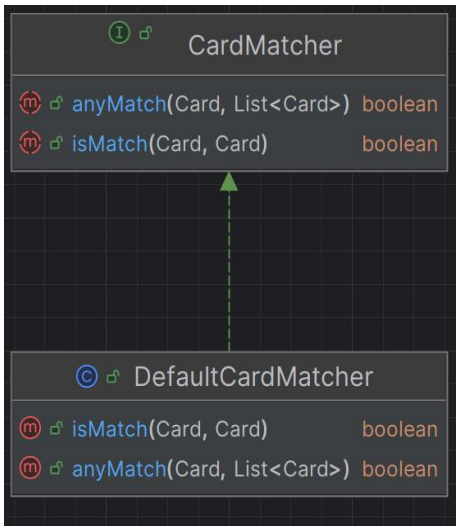



- Card Matcher:

The **CardMathcer** interface is responsible for encapsulating the logic required to match cards within the game. It provides two key methods to facilitate matching:

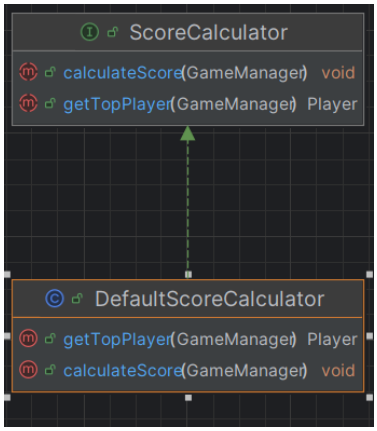
1 – **isMatch**: it takes two cards as a parameter then check if they match or not.

2 - **anyMatch**: it takes a card and list of cards to check if there is any card in the list match with the given card. It could be also use isMatch method.




- Score Calculator:

The ScoreCalculator interface is responsible for encapsulating the logic required to calculate and manage player scores at the end of each round. It provide two key method, both use the gameManager object to get  player information and cards.



- **Game Manager**
  Here is the **GameManager abstract** class, where all the game variations will be implemented. This class will have all the rules mentioned previously as attributes and will be responsible for preparing and starting the game. It will have all the abstract methods that will be implemented by the developer when inheriting this class, which is about defining the rules of variation.


We will have a class called **MyGame** which is inheriting the class **GmaeManage**r.
As we can see, the class **MyGame** is implementing the abstract methods in the super class **GameManager** to make a specific variation and specific rules and will also inherit the attributes from the Game class.

We will see later how it is *flexible* to make a new version.

# Design pattern

Now, I will outline the key design patterns used and explain the purpose of using them:

- **Singleton:**
  We will notice the singleton creational design pattern when we want to create an object from a class **only one time** and **provide global point of access** to it. In this project, I am used this design in **PileGenerator** service, which his work to generate cards, then generate the draw and discard pile using these generated cards. Since the card generation is **time consuming**, instead of generate the same cards each time, why not to generate them just one time, then when there is need to generate new piles, just take a **copy** of the generated card at first time. See the figure for more understanding.

```
public class DefaultPileGenerator implements PileGenerator {  5 usages
    private static DefaultPileGenerator defaultPileGenerator = new DefaultPileGenerator();  1 usage
    private List<Card> cards;  10 usages

    private DefaultPileGenerator() {  1 usage
        cards = new ArrayList<>();
    }

    public static DefaultPileGenerator getInstance() { return defaultPileGenerator; }

    @Override  1 usage
    public void generate(GameManager gameManager) throws Exception {
        if (!cards.isEmpty()) {
            List<Card> newCards = new ArrayList<>(cards);
            Collections.shuffle(newCards);
            gameManager.getDrawPile().setPile(newCards);
            gameManager.getDiscardPile().clear();
            gameManager.getDiscardPile().addCard(gameManager.getDrawPile().drawCard());
        }
```

- **Factory method:**
  The Factory method design pattern is used to encapsulate the complexity of object creation, allowing subclasses to alter the type of object that will be created. In this project this pattern is useful when a player plays a **Wild** or **Wild Draw** card. Both of them allow the player to **change** the color, where you need to take the color form the player as an **input**.

```
public static Color getColor(String color) {
    switch (color) {
        case "red":
            return Color.Red;
        case "blue":
            return Color.Blue;
        case "green":
            return Color.Green;
        case "yellow":
            return Color.Yellow;
        default:
            return Color.Wield;
    }
}
```

  By using this pattern, the game can **encapsulate** the **logic** for **initiating** the selected color. And, ensuring this process handled in a **flexible** and **maintainable** way. See the figure.

- **Template:**
  When initializing the game, I've provided a template pattern to prepare the objects that implements the interfaces as the following:

```
public final void initGame() {
    initDrawPile();
    initCardDrawer();
    initCardMatcher();
    initDiscardPile();
    initPlayersHolder();
    initPileGenerator();
    initCardDistributor();
    initScoreCalculator();
    initPlayerGenerator();
}
```

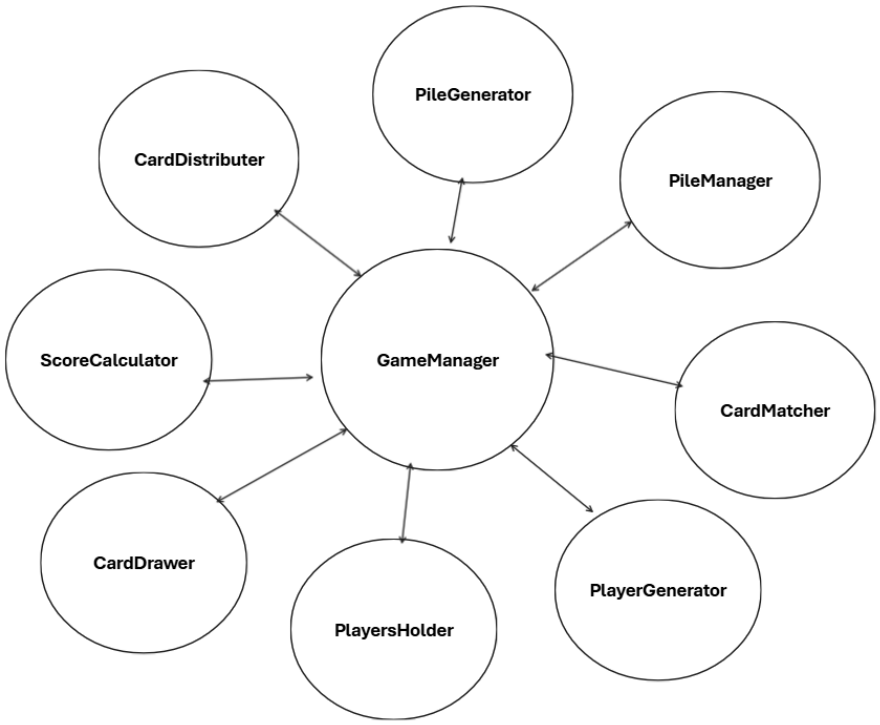
```
public abstract void initDrawPile();  1 implementation  1 usage
public abstract void initCardDrawer();  1 implementation  1 usage
public abstract void initCardMatcher();  1 implementation  1 usage
public abstract void initDiscardPile();  1 implementation  1 usage
public abstract void initPlayersHolder();  1 implementation  1 usage
public abstract void initPileGenerator();  1 implementation  1 usage
public abstract void initCardDistributor();  1 implementation  1 usage
public abstract void initScoreCalculator();  1 implementation  1 usage
public abstract void initPlayerGenerator();  1 implementation  1 usage
```

  As we can see, the **abstract** methods will be called inside **initGame()**, and each Class inherits the **GameManager** class, will **implement** these methods (That will totally create a game variation).

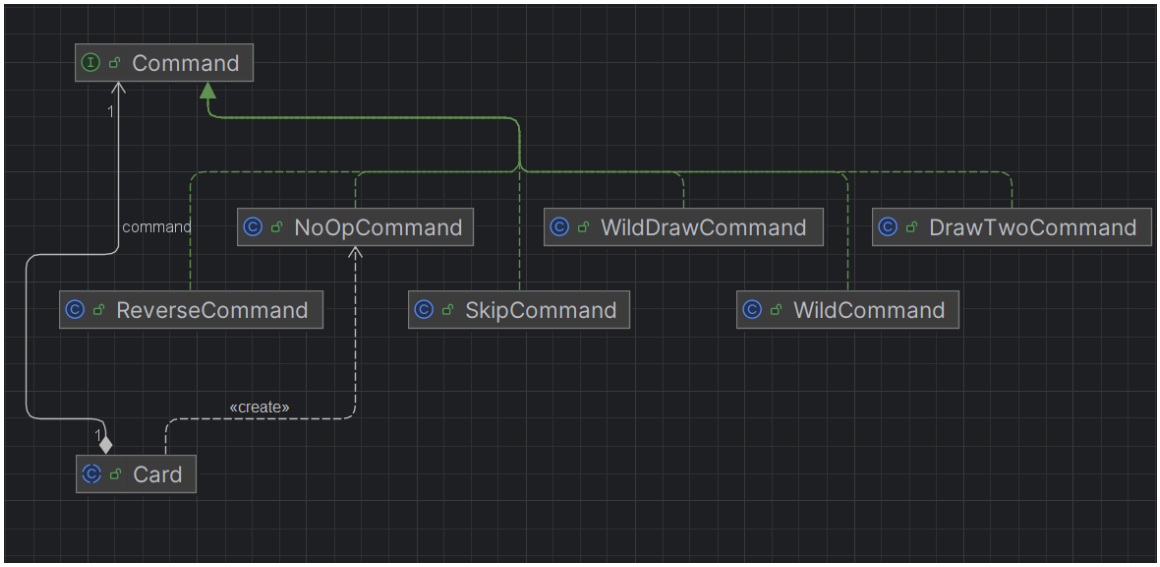
But why the method is **final**?

This method is public final so that we achieve the templet pattern by preventing changing the algorithm (inheriting then changing), but let the user change some steps by implementing the abstract methods.

- **Mediator:**
  The **Mediator** design pattern is employed to reduce the **complexity** of direct **communication** between multiple services by introducing a **central** mediator. In this project, the **GameManager** class acts as the Mediator, **facilitating** communication between various services. Without a mediator, direct interactions between services would lead to a **tightly coupled** and **inflexible** design. For instance, the **ScoreCalculator** service needs to calculate scores for each player at the end of a round. Instead of directly interacting with the PlayersHolder service, the ScoreCalculator communicates with the GameManager. The GameManager then retrieves the necessary player information and updates their scores through the PlayersHolder service.



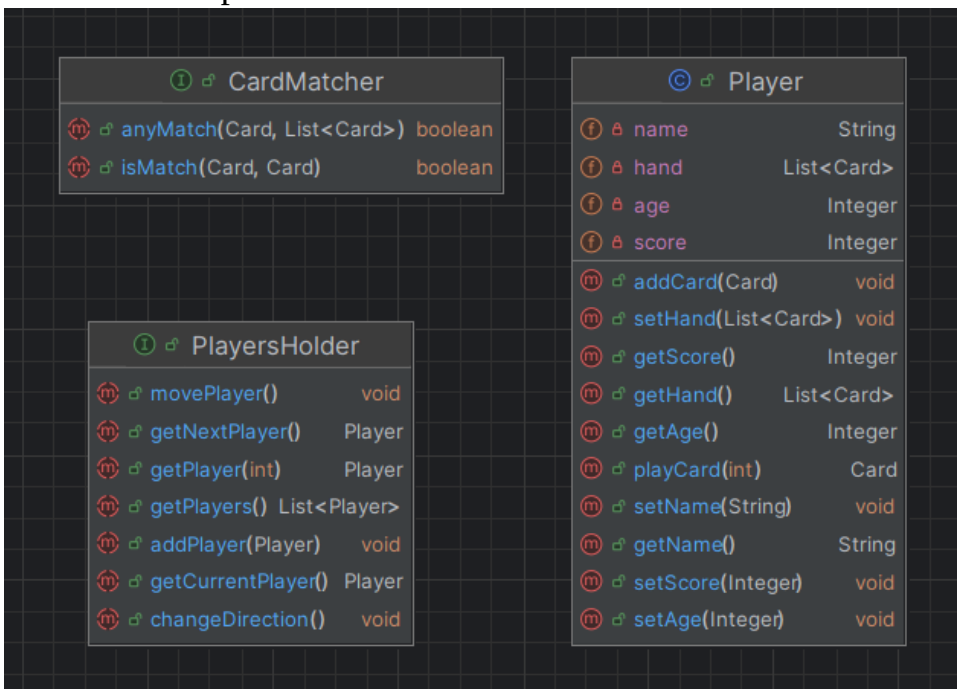
- **Command:**
  The Command design pattern is utilized to encapsulate a request as an object, thereby allowing for parameterization of clients with different requests, queuing of requests, and logging of the operations. In this project, the Command pattern is applied to handle various **card actions**.
  The implementation begins with an abstract **Card** class that includes a **NoOpCommand** (No Operation Command) as a **default** behavior. For specific card types, the command executed depends on the type of card, which is determined through the **constructor**. This approach allows each card type to **define** its own action while **maintaining** a **consistent** interface for executing those actions.
  By using the Command pattern, the project gains **flexibility** in managing card actions, making it easier to **extend** or **modify** the behavior of different card types without altering the core game logic.



# SOLID Principles

Next, we will talk about the SOLID Principles (*developed by uncle bob*) and how they improve the code.

- **SRP (Single responsibility principle):**
  *Class should have only one reason to change (it should have only one responsibility).*
  And that's what I've implemented in my code, each class has one responsibility, all the methods and attributes are related to the class (*high cohesion*), which will make the classes more maintainable.
  For example, The Player class will have the attributes: *name*, *age*, *score*, *cards* (*List<Card>*) with setters and getters for each one.
  Here are examples of some classes:

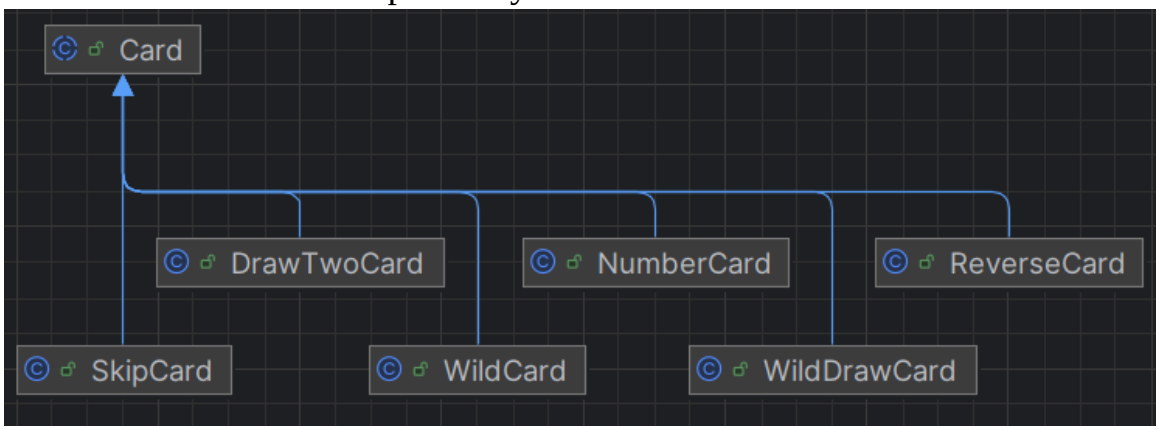



- **OCP (Open / Closed Principle):**
  *Software components should be closed for modification, open for extension.*
  Here I will show an example in my code:

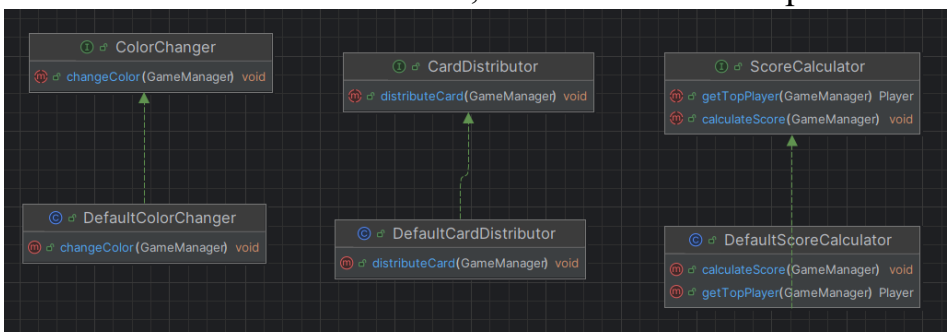
```
@Override  1 usage
public boolean anyMatch(Card lastDrawnCard, List<Card> cards) {
    return cards.stream()
            .anyMatch( Card card -> isMatch(lastDrawnCard , card));
}
```

Here, we are passing the general (*super*) class **Card** rather than sending a specific card like (***Reverse***, ***Skip***, …), and with this, **anyM*atch*()** method will become open for extension (if we add a new card, it will completely word) and close for modification.
I've been also careful about this principle everywhere in my code.

- **LSP (Liskov Substitution Principle):**
  *Objects should be replaceable with their subtypes without affecting the correctness of the program.*
  Here I will show an example in my code on LSP:




- **ISP (Interface Segregation Principle):**
  *No client should be forced to depend on methods it doesn't use.*
  All interfaces in my code will not force any other class to inherit methods that aren't related to it, here are some examples:




All these interfaces will not force any classes to depend on methods they don't use.

- **DIP (Dependency Inversion Principle):**
  *High level modules shouldn't depend on low level modules, both should depend on abstractions.*
  Example in my code:

```
public WildDrawCard(Integer points, Color color, Type type, Command command) {
    super(points, color, type);
    setCommand(command);
}
```

# Effective Java (Items)

Here I will discuss some of the Effective Java Items in my code with some examples for each one:

- **Item 1:**
  *Consider static factory methods instead of constructors*.
  Which I've already considered with applying the singleton pattern like the following:

```
private static DefaultPileGenerator defaultPileGenerator = new DefaultPileGenerator();

private DefaultPileGenerator() {  1 usage
    cards = new ArrayList<>();
}

public static DefaultPileGenerator getInstance() {  1 usage
    return defaultPileGenerator;
}
```

  Which provides more control when creating an object (Singleton in this case).
- **Item 2:**
  *Consider a builder when faced with constructors that have many parameters.*

```
Player player = new Player.PlayerBuilder(name)
        .age(age)
        .score(0)
        .cards(new ArrayList<>())
        .build();
```

- **Item 3:**
  *Enforce the singleton property with a private constructor or an enum type.*
  I've talked about this in the design patterns section where I used the private constructor with a private static inner class to ensure that the singleton is work correctly (Only one object will be created from a class)
- **Item 4:**
  *Eliminate obsolete object references.*
  Because the garbage collector will not collect these objects, but in my code, I've no such references that will not be collected.
- **Item 5:**
  *Prefer try-with-resources to try-finally.*
  All the resources that I've used is the **Scanner()**, and I tried to use try with resources, but the problem was with closing the Scanner(), which will not be allowed to use the scanner any more in the code, so all I've done is to close the resource Scanner after the game ends (the other scanners will remain with try-catch).
- **Item 6:**
  *Obey the general contract when overriding equals.*
  I've included the equals method in both Card and Player classes, here is the Card class with overriding the equals method:

```java
@Override
public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass()) return false;
    Card card = (Card) o;
    return type == card.type && color == card.color && Objects.equals(points, card.points) && Objects.equals(command, card.command);
}

@Override
public int hashCode() {
    return Objects.hash(type, color, points, command);
}
```

- **Item 7:**
  *Always Override toString() method.*
  In my code, any class that has at least one attribute, will have the method toString().
  Here is the toString() method in the Card class:

```java
@Override  1 override
public String toString() {
    return "Card{" +
            "type=" + type +
            ", color=" + color +
            '}';
}
```

# Clean Code Principles (Uncle Bob)

Last thing is the **Robert C. Martin** clean code principles (not the SOLID principles), which are:

- **Naming:**
  In my code, I haven't named a class with a general (not specific) name, every class name has a specific name related to the functionality of this class.

- **Null:**
  I haven't returned **NULL** in a method because of the "**NullPointerException**" also will increase the "CYC"

- **Parameters:**
  In my code, there are no methods that take more than 4 parameters, the methods often take 2-3 parameters or at most 4.

- **Flag Arguments:**
  Also, no flag arguments (boolean value) will be passed to any method.

**Thanks**

**Nabeel Ismaeel**