SOLID

Single Responsibility Principle (SRP)

A class should have only one reason to change, meaning it should **only have one job or responsibility**. If a class is handling multiple tasks, changing one part of its functionality could affect other parts.

```
# Bad Example - Violates SRP
class Report:
    def __init__(self, title):
        self.title = title

    def generate_report(self):
        # Generate report logic here
        pass

    def print_report(self):
        # Print report logic here
        pass
```

```python
# Good Example - Violates SRP

class Report:
    def __init__(self, title):
        self.title = title

    def generate_report(self):
        # Generate report logic here
        pass

class ReportPrinter:
    def print_report(self, report):
        # Print report logic here
        pass
```

Why Better: In the "Good Example," Report handles report data, while ReportPrinter handles printing. This separation makes the code easier to maintain and modify since changes in printing logic won’t affect report generation and vice versa.

Open/Closed Principle (OCP)

Software entities should be **open for extension** but **closed for modification**.
You should be able to add new functionality without changing existing code.

Scenerio: Suppose you’re designing a payment processing system that can handle different types of payments. Initially, you may only need to handle credit card payments, but later, you might want to add support for other payment methods like Bkash or Nagad.

```python
# Bad Example - Violates OCP

class PaymentProcessor:
    def process_payment(self, payment_type, amount):
        if payment_type == 'credit_card':
            self.process_credit_card(amount)
        elif payment_type == 'bkash':
            self.process_bkash(amount)
        else:
            raise ValueError('Unsupported payment type')

    def process_credit_card(self, amount):
        # Process credit card payment
        print(f"Processing credit card payment of ${amount}")

    def process_bkash(self, amount):
        # Process Bkash payment
        print(f"Processing Bkash payment of ${amount}")

# Usage
processor = PaymentProcessor()
processor.process_payment('credit_card', 100)
processor.process_payment('bkash', 200)
```

Why It’s Bad: Every time you add a new payment method, you need to **modify** the PaymentProcessor class.The code becomes **harder to maintain** and extend.

```python
from abc import ABC, abstractmethod

# Abstract class defining the payment method interface
class PaymentMethod(ABC):
    @abstractmethod
    def process(self, amount):
        pass

# Concrete implementation for CreditCard
class CreditCard(PaymentMethod):
    def process(self, amount):
        # Process credit card payment
        print(f"Processing credit card payment of ${amount}")

# Concrete implementation for Bkash
class Bkash(PaymentMethod):
    def process(self, amount):
        # Process Bkash payment
        print(f"Processing Bkash payment of ${amount}")

# PaymentProcessor depends on the abstraction, not the concrete classes
class PaymentProcessor:
    def __init__(self, payment_method: PaymentMethod):
        self.payment_method = payment_method

    def process_payment(self, amount):
        self.payment_method.process(amount)
```

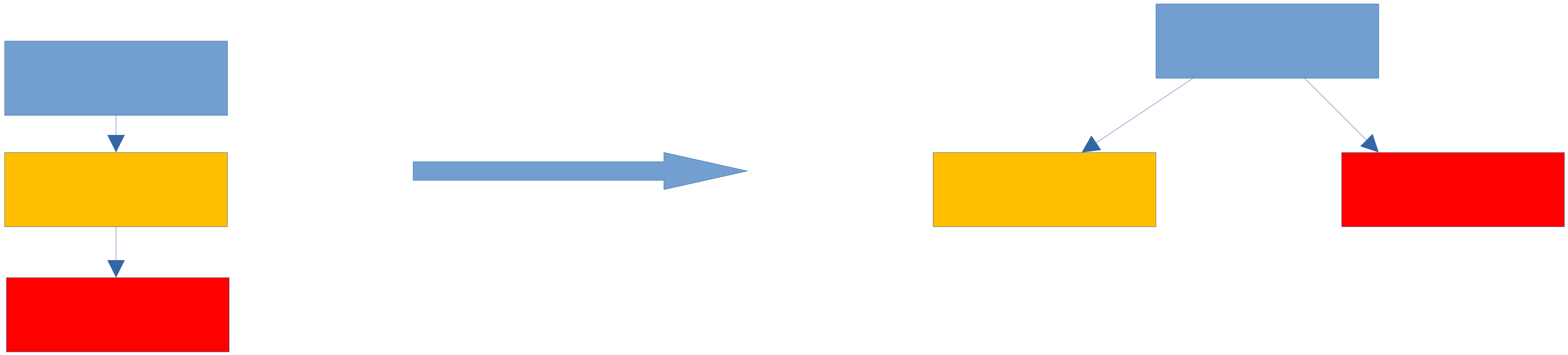
```python
# Usage
credit_card = CreditCard()
bkash = Bkash()

processor = PaymentProcessor(credit_card)
processor.process_payment(100)

processor = PaymentProcessor(bkash)
processor.process_payment(200)
```

Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. Essentially, subclasses should extend the functionality of the base class without changing its expected behavior.



Simple examplanation: Try to avoid multi-level inheritance. If you have **same method in each subclasses.**

```python
# Without LSP
class Shape:
    def area(self):
        raise NotImplementedError("Subclasses should implement this!")

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Square(Rectangle):
    def __init__(self, side_length):
        # This violates LSP because a square is a special type of rectangle
        # where width and height should be the same
        super().__init__(side_length, side_length)



rectangle = Rectangle(5, 10)
print(rectangle.area())
square = Square(5)
print(square.area())
```

```python
# With LSP
class Shape:
    def area(self):
        raise NotImplementedError("Subclasses should implement this!")

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length ** 2



rectangle = Rectangle(5, 10)
print(rectangle.area())
square = Square(5)
print(square.area())
```

Interface Segregation Principle

Interface Segregation Principle states that a class should not be forced to implement interfaces it does not use. In other words, an interface should have methods that are relevant to the classes implementing it. This principle helps in reducing the impact of changes and avoids having classes that are burdened with methods they don’t need.

```
class MultiFunctionDevice:
    def print(self, document):
        pass

    def scan(self, document):
        pass

class SimplePrinter(MultiFunctionDevice):
    def print(self, document):
        print(f'Printing the {document}')

    def scan(self, document):
        raise NotImplementedError("I can not scan")


printer = SimplePrinter()
printer.print(document='file.txt')
printer.scan(document='file.txt') # error
```

Unnecessary Implementation: The printer class has to provide a dummy or empty implementation for the scan method.
Maintenance Issues: Changes to the scan method could inadvertently affect the printer, even though it doesn't need scanning functionality.

To apply the Interface Segregation Principle, you should create more specific interfaces. For our example, you would separate the print and scan functionalities into different interfaces:

```
from abc import ABC, abstractmethod

# Define separate interfaces
class Printer(ABC):
    @abstractmethod
    def print(self, document):
        pass

class Scanner(ABC):
    @abstractmethod
    def scan(self, document):
        pass
```

```python
class SimplePrinter(Printer):
    def print(self, document):
        print(f"Printing document: {document}")

class AllInOneDevice(Printer, Scanner):
    def print(self, document):
        print(f"Printing document: {document}")

    def scan(self, document):
        print(f"Scanning document: {document}")
```

In this design:

- SimplePrinter only implements the Printer interface, which is appropriate for a device that can only print.
- AllInOneDevice implements both Printer and Scanner interfaces, suitable for a device that can both print and scan.

Dependency Inversion Principle

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2 . Abstractions should not depend on details. Details should depend on abstractions.

In simpler terms, the principle suggests that you should depend on abstractions (like interfaces or abstract classes) rather than concrete implementations.

Scenario: Let's say we have a simple application that involves a Report class that depends on a `PDFGenerator` class to generate reports in PDF format.

```python
# without DIP
class PDFGenerator:
    def generate(self, content):
        return f"PDF content: {content}"


class Report:
    def __init__(self):
        self.generator = PDFGenerator()

    def create_report(self, content):
        pdf_content = self.generator.generate(content)
        return pdf_content


# Usage
report = Report()
print(report.create_report("Report Data"))
```

In this example:
- The Report class is tightly coupled to the PDFGenerator class.
- If we need to change the format (e.g., to HTML or DOCX), we would need to modify the Report class to accommodate different generators.

```python
from abc import ABC, abstractmethod

# Define an abstraction (interface)
class ReportGenerator(ABC):
      @abstractmethod
      def generate(self, content):
            pass

# Concrete implementation of the abstraction
class PDFGenerator(ReportGenerator):
      def generate(self, content):
            return f"PDF content: {content}"

class HTMLGenerator(ReportGenerator):
      def generate(self, content):
            return f"HTML content: {content}"

# The Report class depends on the abstraction
class Report:
      def __init__(self, generator: ReportGenerator):
            self.generator = generator

      def create_report(self, content):
            generated_content = self.generator.generate(content)
            return generated_content

# Usage
pdf_generator = PDFGenerator()
html_generator = HTMLGenerator()

report_pdf = Report(pdf_generator)
print(report_pdf.create_report("Report Data"))

report_html = Report(html_generator)
print(report_html.create_report("Report Data"))
```

In this example:
- `ReportGenerator` is an abstract base class (interface) that defines a contract for report generation.
- `PDFGenerator` and `HTMLGenerator` are concrete implementations of `ReportGenerator`.
- The Report class now depends on the `ReportGenerator` abstraction rather than a specific generator implementation.