Lab 1 : SOLID Principles

Ex 1: User Service (SRP - Single Responsibility Principle)

The UserService class currently handles multiple responsibilities: user authentication, email notifications, and saving user data to the database. This design violates the Single Responsibility Principle (SRP), as UserService has more than one reason to change.

```
public class UserService {

    public boolean authenticate(String username, String password) {

        // Logic to authenticate user

        return true;

    }

    public void storeUserData(String user) {

        // Logic to store user data in a database

    }

}
```

Instructions:
Refactor the UserService class so that each responsibility (authentication, notification, and data persistence) is handled by a separate class or service.

Ex 2: Payment Processor (OCP - Open/Closed Principle)

The current PaymentProcessor class is designed to handle multiple payment methods (e.g., credit card, bank transfer). However, each time a new payment method is introduced, the PaymentProcessor class needs to be modified. This violates the Open/Closed Principle (OCP) as the class is not closed to modification.

```
public class PaymentProcessor {
    public void processPayment(String type, double amount) {
        if (type.equals("CreditCard")) {
            // Process credit card payment
        } else if (type.equals("PayPal")) {
            // Process PayPal payment
        }
    }
}
```

Instructions:
Refactor the PaymentProcessor class so that it can support new payment methods without modifying the existing code.

Ex 3: Vehicle/Car/Bicycle (LSP - Liskov Substitution Principle)

In the current Vehicle class hierarchy, all vehicles are expected to have an startEngine() method. However, this is problematic for certain types of vehicles, such as bicycles, which do not have engines. This design violates the Liskov Substitution Principle (LSP) because substituting a Bicycle for a Vehicle may cause unexpected behavior in code expecting all Vehicles to have an engine.

```
public class Vehicle {
    public void startEngine() {
        System.out.println("Engine started.");
    }
}

public class Car extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Car engine started.");
    }
}

public class Bicycle extends Vehicle {
    @Override
    public void startEngine() {
        throw new UnsupportedOperationException("Bicycles don't have
engines.");
    }
}
```

Instructions:
Refactor the code to separate the concept of Vehicle from EngineVehicle, so that only vehicles with engines include startEngine() functionality.

Ex 4: **Human Worker and Robot Worker (ISP - Interface Segregation Principle)**

The current Worker interface includes both work() and eat() methods. This design forces all implementations of Worker, including RobotWorker, to implement the eat() method even though it's irrelevant for robots. This violates the Interface Segregation Principle (ISP), as RobotWorker does not need the eat() method.

```
public interface Worker {
    void work();
    void eat();
}

public class HumanWorker implements Worker {
    @Override
    public void work() {
        System.out.println("Human working...");
    }
    @Override
    public void eat() {
        System.out.println("Human eating...");
```

```
    }
}

public class RobotWorker implements Worker {
    @Override
    public void work() {
        System.out.println("Robot working...");
    }

    @Override
    public void eat() {
        // Robot doesn't eat, but it has to implement this method
        throw new UnsupportedOperationException("Robots don't eat");
    }
}
```

Instructions:
Refactor the code to break down the Worker interface into more specific interfaces so that classes only implement the methods they actually need.

Ex 5: Report Generator (DIP - Dependency Inversion Principle)

The current ReportGenerator class is tightly coupled with the PDFReport class, making it difficult to switch to other report formats, like CSV or XML, without changing ReportGenerator itself. This violates the Dependency Inversion Principle (DIP).

```
public class PDFReport {
    public void generateReport(String data) {
        System.out.println("Generating PDF report with data: " +
data);
    }
}

public class ReportGenerator {
    private PDFReport pdfReport;

    public ReportGenerator() {
        this.pdfReport = new PDFReport();
    }

    public void createReport(String data) {
        pdfReport.generateReport(data);
    }
}
```

Instructions:
Refactor the code so that ReportGenerator depends on an abstraction instead of a specific report format. Your solution should allow ReportGenerator to work with any type of report format (e.g., PDF, CSV, XML) that implements the same interface, enabling more flexibility and modularity.