1. What is the dependency inversion principle? Explain how it contributes to the more testable code.

The dependency inversion principle states that your code should not depend on specific details but rather general rules that are made from interfaces / abstractions. This helps in making more testable code because these dependencies can be easily replicated by fake tests.

2. Describe the scenario where applying the Open-Closed Principle leads to improved code quality.

Open-Closed principle states that the code should be open to extention but not open to modification.

Scenario: For a bank, by desigining with OCP, it will be easier to add new payment methods that follow a common abstraction, leaving existing payment methods as it is.

3. Explain the scenario where the Interface Segregation Principle was beneficial.

The Interface Segregation Principle states that a class shouldn't implement interfaces it doesn't use.

Scenario: Say if we have a Vehicle interface that has a `drive()`, `engine()` and `petrol()` methods, a car that implements this interface is fine but a bicycle will be forced to implement `engine()` and `petrol()` which, it doesnt have.

Benefit: By splitting these interfaces into smaller ones, each vehicle only implements interfaces it needs.

```
~

public class Report {
    public void generateReport() {
    // generate report logic
    }
    public void exportToPDF() {
    // export report to PDF logic
    }
    public void exportToExcel() {
    // export report to Excel logic
    }
}
```

4. Which principle is violated in the code among Single Responsibility, Open Closed, Interface Segregation, and Dependency Inversion Principles? Explain in detail.

-> The code violates the Single Responsibility Principle. The SRP states that a class should have only one Responsibility. The `Report` class,

- Generates the report `generateReport()`
- Exports the report `exportPDF()`, `exportToExcel()`

We can solve this by seperating the exporting logic in a seperate class.

5. Can you provide an example of how to design an online payment processing system while adhering to the SOLID principles? Please explain how each principle can be applied in the context of this system and illustrate with code or a conceptual overview. Let’s assume we have payment types like CreditCardPayment, PayPalPayment, Esewa, and Khalti. Each of these payments should have a method of transferring the amount.

Single Responsibility Principle:

- A `PaymentProcessor` class handles all processing payments.
- Individual classes for `CreditCardPayment` `PayPalPayment` `Esewa` `Khalti` .

Open-Closed Principle:

- Make a new `PaymentMethod` abstract class for all payment types so that adding a new Payment method requires a new class.c:w

Liskov Substitution Principle :

- The `PaymentMethod` parent type, should be able to be substituted with any subtype like `PayPalPayment`.

Interface Segregation Principle :

- prevent classes from implementing methods it doesnt need to.

Dependency Inversion Principle:

- classes like `PaymentProcessor` should depend on abstractions instead of concrete classes.

```
~
public interface PaymentMethod {
    void transferAmount(double amount);
}

public class CreditCardPayment implements PaymentMethod {
    public void transferAmount(double amount) {
        System.out.println("Processing payment", amount);
    }
}

... Same for other payment methods

public class PaymentProcessor {
    private final PaymentMethod paymentMethod;

    public PaymentProcessor(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    public void processPayment(double amount) {
        paymentMethod.transferAmount(amount);
    }
}
```

Examine the following code.

```
~

public class Shape {
    public void drawCircle() {
        // drawing circle logic
    }
    public void drawSquare() {
        // drawing square logic
    }
}
```

You want to add more shapes (e.g., triangles, rectangles) without modifying the existing Shape class. Which design change would adhere to the Open-Closed Principle?
To follow OCP, the `Shape` class shouldnt be modified, but extended.

```
~

public interface Shape {
    void draw();
}

// Separate classes
public class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

public class Square implements Shape {
    public void draw() {
        System.out.println("Drawing a Square");
    }
}

...

public class ShapeDrawer {
    public void drawShape(Shape shape) {
        shape.draw();
    }
}

public class Main {
    public static void main(String[] args) {
        ShapeDrawer drawer = new ShapeDrawer();

        Shape circle = new Circle();

        drawer.drawShape(circle);
    }
}
```

Examine the following code.

```
~

public class Duck {
    public void swim() {
        System.out.println(&amp;quot;Swimming&amp;quot;);
    }
    public void quack() {
        System.out.println(&amp;quot;Quacking&amp;quot;);
    }
}
public class WoodenDuck extends Duck {
    public void quack() {
        throw new UnsupportedOperationException(&amp;quot;Wooden ducks don&amp;#39;t quack&amp;quot;);
    }
}
```

Which principle is violated in the above code among Open Closed, Single Responsibility, Liskov, and Interface Segregation Principle? Explain in detail. Also, update the above code base to solve the issue.

LSP is violated as we cannot substitute the `Duck` class with the `WoodenDuck` properly.

```
~

public interface Swimmable {
    void swim();
}

public class Duck implements Swimmable {
    public void swim() {
        System.out.println("Swimming");
    }

    public void quack() {
        System.out.println("Quacking");
    }
}

public class WoodenDuck implements Swimmable {
    public void swim() {
        System.out.println("Wooden duck swims");
    }
}

public class Main {
    public static void main(String[] args) {
        Swimmable duck = new Duck();
        duck.swim();

        Swimmable woodenDuck = new WoodenDuck();
        woodenDuck.swim();
    }
}
```

Examine the following code.

```
~
public interface PaymentMethod {
    void processPayment();
}
public class PaypalPayment implements PaymentMethod {
    public void processPayment() {
        System.out.println(&amp;quot;Processing PayPal payment&amp;quot;);
    }
}
public class OrderService {
    private PaymentMethod paymentMethod;
    public OrderService(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }
    public void makePayment() {
        paymentMethod.processPayment();
    }
}
```

Which solid principle is being followed above? Explain in detail.

-> The code is following the Dependency Inversion Principle because `OrderService` is dependent on abstractions like `PaymentMethod` and not a specific thing like `PayPalPaymentMethod`, which means that `OrderService` will work with any method as long as its following the `processPaayment()`.