1.What is the dependency inversion principle? Explain how it contributes to the more testable code.
Answer:
The Dependency Inversion Principle (DIP) is one of the five SOLID principles of object-oriented design, which helps in creating flexible, scalable, and maintainable software systems.

It contributes to the more testable code by:
•By depending on abstractions instead of concrete implementations, DIP makes high-level modules independent of the specific behavior of low-level modules. This decoupling simplifies the process of replacing or modifying components, especially when testing.
•Unit tests focus on individual components of the system. When DIP is applied, the high-level modules can be tested independently because they can be supplied with mock implementations of their dependencies. This makes it easier to isolate the behavior of the module being tested.
•DIP encourages the use of abstractions for components that depend on external systems (like databases or APIs). This allows you to replace real external system calls with mocks or fakes during testing, ensuring tests run in isolation and aren't dependent on external services.

2.Describe the scenario where applying the Open-Closed Principle leads to improved code quality.
Applying the Open-Closed Principle improves code quality by making systems more maintainable, extensible, and less prone to errors. For example, in an e-commerce application, instead of hardcoding payment logic for methods like credit cards, PayPal, or bank transfers into a single PaymentProcessor class, you can define a common PaymentMethod interface and create separate classes for each payment type. This way, the PaymentProcessor relies on abstractions rather than concrete implementations, allowing new payment methods to be added without modifying existing code. This reduces the risk of breaking existing functionality, ensures better organization through separation of concerns, and simplifies testing and future updates, creating a robust and scalable system.

3.Explain the scenario where the Interface Segregation Principle was beneficial.
The Interface Segregation Principle (ISP) is beneficial in scenarios where different clients require only a subset of the functionalities offered by a broader interface. For instance, in a document management system, a single DocumentOperations interface with methods like readDocument(), writeDocument(), and printDocument() could cause problems. A Viewer only needs readDocument(), an Editor needs readDocument() and writeDocument(), while a Printer only requires printDocument(). Forcing all these clients to implement unused methods creates unnecessary dependencies and bloated code. By breaking the interface into smaller, role-specific interfaces such as Readable, Writable, and Printable, each class only depends on the methods it needs. This ensures cleaner, more focused implementations, reduces maintenance overhead, and avoids tightly coupling unrelated functionalities, leading to a more modular and maintainable system.

4.Examine the following code.

The code violates the Single Responsibility Principle (SRP) because the Report class handles multiple responsibilities: generating reports and exporting them to PDF or Excel. These responsibilities are unrelated and could change for different reasons, making the code harder to maintain and extend. To fix this, separate the logic into distinct classes, such as a ReportGenerator for generating reports and individual exporter classes (e.g., PDFExporter, ExcelExporter) for handling exports. This ensures better maintainability, testability, and extensibility.

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.

The SOLID principles can be applied to design an online payment processing system:

Single Responsibility Principle (SRP): Each class should handle one responsibility. For example, the CreditCardPayment, PayPalPayment, EsewaPayment, and KhaltiPayment classes will each handle their respective payment logic.

Open-Closed Principle (OCP): New payment methods can be added without modifying existing code. For instance, adding StripePayment only requires creating a new class that adheres to the existing interface.

Liskov Substitution Principle (LSP): Each payment method (e.g., PayPalPayment) should behave as expected when substituted for a more general type (e.g., PaymentMethod).

Interface Segregation Principle (ISP): Define a PaymentProcessor interface with methods relevant only to payment processing, avoiding unnecessary methods.

Dependency Inversion Principle (DIP): High-level modules like PaymentService should depend on abstractions (e.g., PaymentProcessor), not concrete implementations.

6. Examine the following code.

The current Shape class violates the OCP because adding new shapes requires modifying the existing class. To adhere to the OCP, introduce an abstraction for shapes and implement separate classes for each shape.

```
// Shape Interface
public interface Shape {
    void draw();
}

// Concrete Shape Classes
public class Circle implements Shape {
    @Override
    public void draw() {
```

```
        System.out.println("Drawing Circle");
    }
}

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

// Main
public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape square = new Square();
        circle.draw();
        square.draw();
    }
}
```

7.

The code violates the Liskov Substitution Principle (LSP) because WoodenDuck cannot fully substitute Duck due to the quack method throwing an exception.

```
// SwimBehavior Interface
public interface SwimBehavior {
    void swim();
}

// QuackBehavior Interface
public interface QuackBehavior {
    void quack();
}

// Concrete Duck Classes
public class RealDuck implements SwimBehavior, QuackBehavior {
    @Override
    public void swim() {
        System.out.println("Swimming");
    }

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

public class WoodenDuck implements SwimBehavior {
    @Override
    public void swim() {
```

```
        System.out.println("Floating on water");
    }
}

// Main
public class Main {
    public static void main(String[] args) {
        SwimBehavior woodenDuck = new WoodenDuck();
        SwimBehavior realDuck = new RealDuck();

        woodenDuck.swim();
        ((QuackBehavior) realDuck).quack(); // Cast needed for demonstration.
    }
}
```