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

testable code.

🡺High-level modules and low-level modules should depend on abstractions, not on each other.  
**Improves Testability:**

* Loosely couples components, allowing dependencies to be swapped for mocks or stubs during testing.
* Simplifies unit testing by isolating parts of the system.

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

quality.

🡺Suppose you're adding a new payment method (e.g., a digital wallet) to an existing payment system. With OCP, you can extend functionality by creating new modules/classes for the digital wallet without altering the existing payment processing logic. This reduces the risk of introducing bugs and promotes code reuse.

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

🡺In a multi-functional printer system, separating interfaces for printing, scanning, and faxing ensures that a basic printer only implements the print functionality, avoiding the need to handle irrelevant methods like scanning or faxing.

4. Examine the following code.

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

}

}

Which principle is violated in the code among Single Responsibility, Open Closed, Interface

Segregation, and Dependency Inversion Principles? Explain in detail.

🡺**Violated Principle:** Single Responsibility Principle (SRP).  
**Reason:** The Report class handles both generating reports and exporting them in different formats. Each responsibility should be managed by a separate class to make the system easier to maintain and extend.

5. Canyou 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.

🡺**Application of SOLID:**

1. **SRP:** Separate classes for each payment method (e.g., CreditCardPayment, PayPalPayment) and their processing logic.
2. **OCP:** Add new payment methods (e.g., Esewa, Khalti) without modifying existing code.
3. **LSP:** Ensure all payment methods can replace one another without breaking the system.
4. **ISP:** Define lightweight interfaces that payment methods must implement.
5. **DIP:** Payment processing depends on abstractions (interfaces), making it flexible and testable.

6. Examine the following code.

public class Shape {

public void drawCircle() {

// drawing circle logic

}

public void drawSquare() {

// drawing square logic

}

}

SOLID PRINCIPLES

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?

🡺**Problem:** Adding new shapes (e.g., triangles) requires modifying the existing class, which violates OCP.  
**Solution:** Define a general abstraction for shapes. New shapes can be introduced as independent implementations, leaving the base class unchanged.

7. Examine the following code.

public class Duck {

public void swim() {

System.out.println("Swimming");

}

public void quack() {

System.out.println("Quacking");

}

}

public class WoodenDuck extends Duck {

@Override

public void quack() {

throw new UnsupportedOperationException("Wooden ducks don't quack");

}

}

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.

🡺**Violated Principle:** Liskov Substitution Principle (LSP).  
**Reason:** WoodenDuck overrides the quack method to throw an exception, breaking the contract of the Duck class.

**Solution:** Use behavior-specific interfaces or segregate functionalities so that WoodenDuck only implements the behaviors it supports.

8. Examine the following code.

public interface PaymentMethod {

void processPayment();

}

public class PaypalPayment implements PaymentMethod {

@Override

public void processPayment() {

System.out.println("Processing PayPal payment");

}

}

public class OrderService {

private PaymentMethod paymentMethod;

public OrderService(PaymentMethod paymentMethod) {

this.paymentMethod = paymentMethod;

SOLID PRINCIPLES

}

public void makePayment() {

paymentMethod.processPayment();

}

}

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

**🡺Followed Principle:** Dependency Inversion Principle (DIP).  
**Reason:** OrderService depends on the abstraction (PaymentMethod interface) rather than concrete implementations like PaypalPayment. This makes the system flexible, extensible, and testable.