Name: Abhinab Sherchan

Class: L5CG24

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

The dependency inversion principle states that "High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions."

The Dependency Inversion principle allows us to test our code by helping us substitute our implemented code with mock objects for testing. Since high level modules depend on abstractions we can mock dependencies and stimulate behaviors. This helps us test our implementations without changing the other parts of our code, resulting in easy maintenance, flexible code and reduced side effects.

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

The Open Closed Principle states that "Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.". It means that we should be able to add new functionality without changing the existing code.

Example:

```
public class ShapeDrawer {
    public void drawShape(String shapeType) {
        if (shapeType.equals("CIRCLE")) {
            System.out.println("Drawing a Circle");
        } else if (shapeType.equals("RECTANGLE")) {
            System.out.println("Drawing a Rectangle");
        }
    }
}
```

Here we calculate area of different shapes. But to add a different shape we need to change the whole drawshape method.

Instead of doing this OCP encourages us to use abstraction and add new interface without modifying the existing ones like this:

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

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

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

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

        Shape circle = new Circle();
        Shape rectangle = new Rectangle();

        shapeDrawer.drawShape(circle);       // Output: Drawing a Circle
        shapeDrawer.drawShape(rectangle);    // Output: Drawing a Rectangle
    }
}
```

Now we can also add more shapes by implementation without changing the existing code but by adding new ones.

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

Interface Segregation Principle states that “Code should not be forced to depend on methods it doesn't use”. It encourages us to split large interfaces into smaller ones and ensures that we only use the methods that we need.

Scenario:

We have a worker interface with methods of work and eat.

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

Now in a factory we have two types of workers : Employee and Robot.

```
public class Employee implements Worker {
    @Override
    public void work() {
        System.out.println("Employee is working.");
    }

    @Override
    public void eat() {
        System.out.println("Employee is eating.");
    }
}

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

    @Override
    public void eat() {
        // Robots don't eat, so this method is irrelevant
        throw new UnsupportedOperationException("Robots don't eat.");
    }
}
```

But a robot eating doesn't make sense, and we are only wasting code using the eat method for robot as it is a method robot doesn't use.

Hence using Interface Segregation Solution we Split the higher interface into two smaller interface: Workable And Eatable.

```java
public interface Workable {
    void work();
}

public interface Eatable {
    void eat();
}
```

Now we need to implement interface in Employee but the robot will only implement Workable.

```java
public class Employee implements Workable, Eatable {
    @Override
    public void work() {
        System.out.println("Employee is working.");
    }

    @Override
    public void eat() {
        System.out.println("Employee is eating.");
    }
}

public class Robot implements Workable {
    @Override
    public void work() {
        System.out.println("Robot is working.");
    }
}
```

In this way, we removed a method for robots that it didn’t use and made the system easier to extend in the future.

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.

The given code violates the Single Responsibility Principle which states that" One class should have only one responsibility and should have only one reason to change is required."

The code has class Report which has multiple responsibilities i.e Generating report, exporting report to PDF and exporting the report to Excel. It makes the code hard to understand and maintain and we cannot test each class independently as well. To make this code follow Single Responsibility Principle we need to modify the code as follows:

```java
package workshop4;

public class workshop6 {
    public class Report {
        public void generateReport() {
            // generate report logic
        }
        public class PDFExporter {
            public void exportToPDF(Report report) {
                // export report to PDF logic
            }
        }

        public class ExcelExporter {
            public void exportToExcel(Report report) {
                // export report to Excel logic
            }
        }

    }

}

```

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.

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

// Implementation for Credit Card Payment
public class CreditCardPayment implements PaymentMethod {
    @Override
    public void transferAmount(double amount) {
        System.out.println("Transferring $" + amount + " via Credit Card.");
    }
}

// Implementation for PayPal Payment
public class PayPalPayment implements PaymentMethod {
    @Override
    public void transferAmount(double amount) {
        System.out.println("Transferring $" + amount + " via PayPal.");
    }
}

// Implementation for Esewa Payment
public class EsewaPayment implements PaymentMethod {
    @Override
    public void transferAmount(double amount) {
        System.out.println("Transferring $" + amount + " via Esewa.");
    }
}

// Implementation for Khalti Payment
public class KhaltiPayment implements PaymentMethod {
    @Override
    public void transferAmount(double amount) {
        System.out.println("Transferring $" + amount + " via Khalti.");
    }
}
```

```java
// High-level module: PaymentProcessor
public class PaymentProcessor {
    private PaymentMethod paymentMethod;

    // Dependency Injection (Dependency Inversion Principle)
    public PaymentProcessor(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

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

How it Follows to SOLID principles:

SRP: Each class has only one responsibility.

DIP: The PaymentProcessor depends on the abstraction PaymentMethod.

OCP: We can simply add new codes for more payments without altering the current code.

LSP: The PaymentMethod implementations can substitute for interface

ISP: we do not have unnecessary code that isn't used anywhere. All the declared methods are used and to the point.

6. 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?

ANS: we can add more shapes without modifying the Shape adhering to the Open-Closed Principle in the following way:

```
public interface Shape {
    void draw();
}
public class Circle implements Shape {
    @Override
    public void draw() {
    }
}
public class Square implements Shape {
    @Override
    public void draw() {
    }
}
public class Triangle implements Shape {
    @Override
    public void draw() {
                }
}
```

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.

ANS: The given code violates the Liskov Substitution Principle(LSP). LSP states that “subclass should extend the behavior of the superclass without changing its expected behavior”.

The given code can be modified and corrected in the given way:

```java
// Define a Quackable interface
public interface Quackable {
    void quack();
}

// Define a general Duck class implementing Quackable
public class Duck implements Quackable {
    public void swim() {
        System.out.println("Swimming");
    }

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

// WoodenDuck does not quack, so it does not implement Quackable
public class WoodenDuck extends Duck {
    @Override
    public void swim() {
        System.out.println("Wooden ducks float, not swim.");
    }
}
```

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;



    }

    public void makePayment() {
      paymentMethod.processPayment();
    }
  }
```

The above code follows the Dependency Inversion Principle which states that “High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.”

The PaymentMethod interface is an abstraction. It defines the method processPayment() without specifying how the payment will be processed. This abstraction ensures that different payment methods (e.g., PayPal, Credit Card, etc.) can be implemented without directly coupling the OrderService class to a specific implementation.