SOLID PRINCIPLES

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

   - Decoupling the software modules is called Dependency Inversion. In this, the modules depend upon abstraction rather than high level modules.

   - Dependency inversion contributes to more testable code by the following points:
     - Improved Flexibility: using an in-memory database for testing becomes straightforward instead of production database, by swapping implementations.
     - Isolation code: individual components can be tested without requiring the actual implementations of dependencies.
     - Easy Mocking: Using the abstractions, it allows for easier creation of mock or fake implementations for unit testing.

2. Describe the scenario where applying the Open-Closed Principle leads to improved code quality.
   - Open-closed Principle should be opened for extension but closed for modification, which prevents the modification of existing code and potential few bugs.
   - If we want to calculate the area of various shapes, we will need to add new shape, so we will need to modify and create a new shape method nut it will violate as it is closed for modification. So, to add a new shape, method of drawShape should be modified.

3. Explain the scenario where the Interface Segregation Principle was beneficial.
   - In interface Segregation Principle, larger interface should be split into smaller ones and ensure that implementing classes only need to be concerned about the methods that have interest to them
   - In a scenario where an interface ‘Working’ is made, and defines some actions ‘eat’ and ‘work’ and working beings are human and robot. Now, in this case the the eat method is liable only for the human but not for the robot as robot cannot eat anything but is forced to have an implement of eat method.
   - In this case, now the for both Human and Robot, we will be needing to create two interface of working and eating instead of creating the method.
   - In this way, the robot will not be forced to have a eat method.

SOLID PRINCIPLES

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 **Single Responsibilty Principle (SRP)** because SRP states that a class should have only one reason to change. The 'report' class is handling two responsibility, which is harder to maintain and test the report. We can fix this issue by separating the responsibilities into different classes:
  - One for generating reports
  - Other to export the report to different formats.

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.

SOLID PRINCIPLES

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?

- To append the Open-Closed Principle (OCP) we need to refactor thr design to allow adding new shapes without modifying the existing ‘Shape’ Class because OCP states that class should be open for extension but closed for modification.
- The best approach is to use inheritance and polymorphism by defining an abstract class or an interface.
- For Example;

  Defining an abstract class

```
// Abstraction
interface Shape {
    void draw();
}
```

  Creation of Shape class

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

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

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

Polymorphism

```
import java.util.List;

public class ShapeDrawer {
    public void drawShapes(List<Shape> shapes) {
        for (Shape shape : shapes) {
            shape.draw();  // Polymorphic call
        }
    }
}
```

New shape

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

Main Method

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        List<Shape> shapes = Arrays.asList(
            new Circle(),
            new Square(),
            new Triangle(),
            new Rectangle()
        );

        ShapeDrawer drawer = new ShapeDrawer();
        drawer.drawShapes(shapes);
    }
}
```

SOLID PRINCIPLES

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 resolve the issue.

```
- // Behavior interfaces
- public interface SwimBehavior {
-   void swim();
- }
-
- public interface QuackBehavior {
-   void quack();
- }
-
- // Implementations of behaviors
- public class CanSwim implements SwimBehavior {
-   @Override
-   public void swim() {
-     System.out.println("Swimming");
-   }
- }
-
- public class CannotSwim implements SwimBehavior {
-   @Override
-   public void swim() {
-     System.out.println("This duck cannot swim");
-   }
```

```
-   }
-
-   public class CanQuack implements QuackBehavior {
-     @Override
-     public void quack() {
-       System.out.println("Quacking");
-     }
-   }
-
-   public class CannotQuack implements QuackBehavior {
-     @Override
-     public void quack() {
-       System.out.println("This duck cannot quack");
-     }
-   }
-
-   // Duck class using composition
-   public class Duck {
-     private SwimBehavior swimBehavior;
-     private QuackBehavior quackBehavior;
-
-     public Duck(SwimBehavior swimBehavior, QuackBehavior quackBehavior) {
-       this.swimBehavior = swimBehavior;
-       this.quackBehavior = quackBehavior;
-     }
-
-     public void swim() {
-       swimBehavior.swim();
-     }
-
-     public void quack() {
-       quackBehavior.quack();
-     }
-   }
-
-   // WoodenDuck class
-   public class WoodenDuck extends Duck {
-     public WoodenDuck() {
-       super(new CannotSwim(), new CannotQuack());
-     }
-   }
-
-   // Test the code
-   public class Main {
```

```
-       public static void main(String[] args) {
-         Duck mallardDuck = new Duck(new CanSwim(), new CanQuack());
-         Duck woodenDuck = new WoodenDuck();
-
-         System.out.println("Mallard Duck:");
-         mallardDuck.swim();
-         mallardDuck.quack();
-
-         System.out.println("\nWooden Duck:");
-         woodenDuck.swim();
-         woodenDuck.quack();
-       }
-   }
```

- The above code violates the **Liskov Substitution Principle (LSP)**. The code violates as ‘WoodenDuck’ extends Duck and a quack method is thrown in it which is an UnsupportedOperationException, which violates the exceptions of the Duck class. To resolve this violation and adhere to the Liskov Substitution Principle (LSP), we should refactor the code to avoid using inheritance in a way that allows a subclass to deviate from the expected behavior of its superclass. The solution involves using composition over inheritance, where behaviors like swimming and quacking are abstracted into separate interfaces or classes. This approach ensures that each duck type can define its behavior without violating the principles of object-oriented design.

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();
    }
}
```

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

- The code provided demonstrates the Dependency Inversion Principle (DIP) form SOLID principles because DIP states that high-level modules should noit depend on low level modules and both should depend on abstractions. Whereas, abstractions should not depend on details but details should be depend upon abstraction.