Assignment System Design

Q1. Explain SRP and OCP in detail with proper examples.

Q2. Discuss in detail about the violations in SRP and OCP along with their fixes.

S – Single Responsibility Principle (SRP)

Definition

The **Single Responsibility Principle** states that **a class should perform only one specific responsibility**.

A class must have **only one reason to change**.

Combining multiple responsibilities in a single class leads to:

- Tight coupling
- Difficult maintenance
- Frequent code changes

Real-Time Example: E-Commerce Order Processing System

SRP Violation

```
classOrderService {

publicvoidplaceOrder(Order order) {
// order placement logic
    }

publicvoidcalculateTotal(Order order) {
// price calculation logic
```

```
    }

publicvoidgenerateInvoice(Order order) {
// invoice generation logic
    }

publicvoidprocessPayment(String paymentMode) {
if (paymentMode.equals("CARD")) {
// card payment gateway logic
        }
if (paymentMode.equals("UPI")) {
// UPI payment logic
        }
    }

publicvoidsendOrderNotification(String channel) {
if (channel.equals("EMAIL")) {
// send email
        }
if (channel.equals("SMS")) {
// send SMS
        }
    }
}
```

Why this violates SRP?

This class has **multiple reasons to change**, such as:

- Change in **invoice format**
- Addition of **new payment method** (Wallet, NetBanking)
- New **notification channel** (WhatsApp, Push Notification)
- Modification in **pricing logic**

All these changes force updates in the **same class**, which violates SRP.

Solution – Applying SRP

Each responsibility is moved to a **separate service**, as done in real-world microservice or layered architectures.

Order Management Service

```
classOrderService {
publicvoidplaceOrder(Order order) {
// order placement logic
    }
}
```

Pricing Service

```
classPricingService {
publicvoidcalculateTotal(Order order) {
// price calculation logic
    }
}
```

Invoice Service

```
classInvoiceService {
publicvoidgenerateInvoice(Order order) {
// invoice generation logic
    }
}
```

Payment Service

```
classPaymentService {
publicvoidprocessPayment(String paymentMode) {
if (paymentMode.equals("CARD")) {
// card gateway
      }
if (paymentMode.equals("UPI")) {
// UPI gateway
      }
   }
}
```

Notification Service

```
classNotificationService {
publicvoidsendOrderNotification(String channel) {
if (channel.equals("EMAIL")) {
// email service
      }
if (channel.equals("SMS")) {
// SMS service
      }
   }
}
```

Advantages of SRP in Real Applications

- Easier **feature upgrades**
- Independent **testing and debugging**
- Better **scalability**

- Aligns with **real-world enterprise architecture**

OCP – Open Closed Principle

Definition

The **Open Closed Principle (OCP)** states that:

> **According to new requirements, a module should be open for extension but closed for modification.**

This means:

- We should be able to **extend the behavior of a class**
- **Without modifying the existing source code**

Why OCP is required

- Existing code is already **tested and stable**
- Modifying it again may **introduce new bugs**
- OCP helps in building **scalable and maintainable applications**

OCP Violation Example (Area Calculation)

Consider a class that calculates the area of different shapes.

```
classAreaCalculator {

publicdoublecalculateArea(String shape, double a,double b) {

if (shape.equals("RECTANGLE")) {
return a * b;
	}

if (shape.equals("CIRCLE")) {
```

```
return3.14 * a * a;
      }

return0;
   }
}
```

Why this design violates OCP

If a new requirement comes:

> "Calculate the area of a Triangle"

Then we must:

- Add another `if` condition
- Modify the **existing AreaCalculator class**

```
if (shape.equals("TRIANGLE")) {
return0.5 * a * b;
}
```

This means:

- Existing code is **modified**
- Risk of breaking old logic
- Multiple changes in the same class

❌ **OCP is violated**

Applying Open Closed Principle

To follow OCP, we use **abstraction** (interfaces) and **polymorphism**.

Step 1: Create Shape Interface

```
publicinterfaceShape {
doublearea();
}
```

This interface defines a **common contract** for all shapes.

Step 2: Rectangle Class

```
publicclassRectangleimplementsShape {

double length;
double breadth;

publicRectangle(double length,double breadth) {
this.length = length;
this.breadth = breadth;
   }

@Override
publicdoublearea() {
return length * breadth;
   }
}
```

Step 3: Circle Class

```
publicclassCircleimplementsShape {

double radius;

publicCircle(double radius) {
this.radius = radius;
```

```
    }

@Override
publicdoublearea() {
return3.14 * radius * radius;
    }
}
```

Step 4: Triangle Class (New Requirement)

```
publicclassTriangleimplementsShape {

double base;
double height;

publicTriangle(double base,double height) {
this.base = base;
this.height = height;
    }

@Override
publicdoublearea() {
return0.5 * base * height;
    }
}
```

✔️ New shape added

✔️ No existing class modified

Step 5: Area Calculator (Closed for Modification)

```
classAreaCalculator {

publicdoublecalculateArea(Shape shape) {
return shape.area();
   }
}
```

This class:

- Depends on the **Shape interface**
- Does not change when new shapes are added

How this follows OCP

- New shapes are added by **creating new classes**
- Existing logic remains **unchanged**
- System is **open for extension**
- System is **closed for modification**

Advantages of OCP

- Easy to add new features
- Improves code stability
- Encourages loose coupling
- Widely used in real-world applications (graphics engines, CAD tools, games)