



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

## Assignment 1

**Student Name:** Ayush Tiwari

**Branch:** BE-CSE

**Semester:** 6th

**Subject Name:** System Design

**UID:** 23BCS11366

**Section/Group:** KRG-3B

**Date of Performance:** 04-02-2026

**Subject Code:** 23CSH-314

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

**Ans:**

**Definition:**

A class should have only one reason to change.

In other words, it should do one job only.

**Why SRP?**

- Easier to understand and maintain
- Fewer bugs when changes happen
- Better reuse and testing

**SRP Violation Example**

```
class Invoice {  
    void calculateTotal() {  
        System.out.println("Calculating total...");  
    }  
    void printInvoice() {  
        System.out.println("Printing invoice...");  
    }  
    void saveToDatabase() {  
        System.out.println("Saving to DB...");  
    }  
}
```

**Problem:**

This class has 3 responsibilities:

1. Business logic (calculate)
2. Presentation (print)
3. Persistence (save)

So it has multiple reasons to change → SRP violated.

### **SRP Fixed Version**

```
class Invoice {
    void calculateTotal() {
        System.out.println("Calculating total...");
    }
}

class InvoicePrinter {
    void printInvoice() {
        System.out.println("Printing invoice...");
    }
}

class InvoiceRepository {
    void saveToDatabase() {
        System.out.println("Saving to DB...");
    }
}
```

Now:

- Each class has one responsibility
- Each has one reason to change

### **2) OCP – *Open/Closed Principle***

Definition:

Software entities should be open for extension but closed for modification.

Means:

- You should add new behavior
- Without changing existing code

### **OCP Violation Example**

```
class DiscountCalculator {
    double calculate(String type, double amount) {
        if (type.equals("STUDENT")) {
```

```

        return amount * 0.9;
    else if (type.equals("SENIOR"))
        return amount * 0.8;
    else
        return amount;
}
}

```

**Problem:**

If a new discount comes, we must modify this class → OCP violated.

### OCP Fixed Version (Using Polymorphism)

```

interface Discount {
    double apply(double amount);
}

```

```

class StudentDiscount implements Discount {
    public double apply(double amount) {
        return amount * 0.9;
    }
}

```

```

class SeniorDiscount implements Discount {
    public double apply(double amount) {
        return amount * 0.8;
    }
}

```

```

class DiscountCalculator {
    double calculate(Discount discount, double amount) {
        return discount.apply(amount);
    }
}

```

Now:

- To add new discount → **create new class**
- No need to modify DiscountCalculator

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

**Ans:**

### **SRP Violation**

<b>Issue</b>	<b>Effect</b>	<b>Fix</b>
One class does many tasks	Hard to maintain	Split into multiple classes
Changes affect unrelated features	Bugs increase	Separate responsibilities

**Fix:**

Break large classes into smaller classes where each does only one job.

---

### **OCP Violation**

<b>Issue</b>	<b>Effect</b>	<b>Fix</b>
Uses many if-else or switch	Needs modification for new features	Use interfaces & inheritance
Tight coupling	Hard to extend	Use polymorphism

**Fix:**

Use interfaces, abstract classes, and polymorphism to extend behavior instead of modifying existing code.

### **Violations of SRP (Single Responsibility Principle)**

#### **SRP Rule**

A class should have only one reason to change.

If a class performs more than one task, it violates SRP.

---

#### **Common SRP Violations**

1. God Class  
One class performs many tasks such as business logic, database access, printing and validation.
  2. Mixed Responsibilities  
One class handles multiple unrelated operations.
  3. Difficult Maintenance  
A small change in one feature can break other features.
  4. Low Reusability  
Parts of the class cannot be reused independently.
- 

#### **Example of SRP Violation**

```
class Invoice {  
    void calculateTotal() {  
        System.out.println("Calculating total");  
    }  
  
    void printInvoice() {  
        System.out.println("Printing invoice");  
    }  
}
```

```

    }

    void saveToDatabase() {
        System.out.println("Saving invoice");
    }
}

```

This class has three responsibilities calculation, printing and saving.  
Therefore it violates SRP.

---

### **Fix for SRP Violation**

Split the class into separate classes, each handling one task.

```

class Invoice {
    void calculateTotal() {
        System.out.println("Calculating total");
    }
}

```

```

class InvoicePrinter {
    void printInvoice() {
        System.out.println("Printing invoice");
    }
}

```

```

class InvoiceRepository {
    void saveToDatabase() {
        System.out.println("Saving invoice");
    }
}

```

Now each class has only one responsibility.

---

### **Violations of OCP (Open Closed Principle)**

#### **OCP Rule**

Software classes should be open for extension but closed for modification.  
This means new features should be added without changing existing code.

---

#### **Common OCP Violations**

1. Too many if else conditions  
Every new feature requires changing old code.
  2. Switch case based logic  
Each new case forces modification of the class.
  3. Tight Coupling  
The class depends directly on concrete classes instead of abstractions.
- 

#### **Example of OCP Violation**

```

class DiscountCalculator {
    double calculate(String type, double amount) {
        if (type.equals("STUDENT")) {

```

```

        return amount * 0.9;
    } else if (type.equals("SENIOR")) {
        return amount * 0.8;
    } else {
        return amount;
    }
}

```

Whenever a new discount type is added, this class must be modified.  
This violates OCP.

---

### Fix for OCP Violation

Use interface and polymorphism.

```
interface Discount {
    double apply(double amount);
}
```

```
class StudentDiscount implements Discount {
    public double apply(double amount) {
        return amount * 0.9;
    }
}
```

```
class SeniorDiscount implements Discount {
    public double apply(double amount) {
        return amount * 0.8;
    }
}
```

```
class DiscountCalculator {
    double calculate(Discount discount, double amount) {
        return discount.apply(amount);
    }
}
```

Now new discounts can be added by creating new classes without modifying the existing calculator.