DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

ASSIGNMENT- 1

Name: Jay Shankar Kumar
UID: 23BCS10408
Branch: BE-CSE
Section/Group: 23BCS_KRG-1-A
Semester: 6th
Date of Submission: 04-02-2026
Subject Name: System Design
Subject Code: 23CSH-314

Q1. Explain SRP and OCP in detail with suitable examples Ans: 1. Single Responsibility Principle (SRP)

The Single Responsibility Principle states that a class should have only one responsibility, meaning it should have only one reason to change.
In simpler terms, a class must focus on doing one specific task and do it well.
Each class or module should handle a single concern or business logic, instead of mixing multiple responsibilities together.

Importance of SRP

Following SRP provides several benefits:

1. Improves code readability and clarity
2. Makes unit testing simpler
3. Enhances maintainability
4. Reduces chances of introducing bugs
5. Promotes better reusability of components

SRP Violation Scenario

Consider a situation where one class performs multiple unrelated tasks, such as:
• Managing data
• Generating reports
• Sending emails
This clearly violates SRP because the class has multiple reasons to change.

Example (SRP Violation)

class Student { public:

```
    void addStudent() {
        // logic to store student data in database
    }

    void generateReport() {
        // logic to generate PDF report
    }

    void sendEmail() {
        // logic to send email to student
    }
};
```

Here, the Student class is responsible for **data handling**, **report creation**, and **email communication**, which is poor design.

Correct SRP-Based Design

```
class StudentService { public:
    void addStudent();
};

class ReportService { public:
    void generateReport();
};

class EmailService { public:
    void sendEmail();
};
```

Benefits of Correct SRP Design

1. Each class handles only one responsibility
2. Changes in one feature (e.g., email logic) do not affect others
3. Code becomes easier to test, extend, and maintain

---

2. Open/Closed Principle (OCP)

The Open/Closed Principle states that software components should be open for extension but closed for modification.

This means we should be able to add new functionality without altering already tested and stable code.

Why OCP Matters

- Protects existing logic from accidental breakage
- Helps in building scalable systems
- Encourages flexible and plug-in based designs

Example (OCP Violation)

```
class Payment { public:
   void pay(string type) {
if(type == "UPI") {
          // UPI payment logic
       } else if(type == "CARD") {
          // Card payment logic
       }
   }
};
```

In this design, every new payment method requires modifying the same class, which violates OCP.

OCP-Compliant Design

```
class Payment { public:
   virtual void pay() = 0;
};

class UpiPayment : public Payment { public:
   void pay() override {
// UPI payment logic     }
};

class CardPayment : public Payment { public:
```

```
    void pay() override {
        // Card payment logic
    }
};
```

Now, new payment methods can be added without changing existing code, satisfying OCP.

Q2. Explain SRP and OCP violations along with their solutions Ans:
Software design principles aim to enhance code quality, scalability, maintainability, and reliability.
However, violations of SRP and OCP are common in real-world projects, leading to poor system design.
This section explains the causes, symptoms, consequences, and fixes for both violations.

Part A: Violations of Single Responsibility Principle (SRP) Meaning of SRP Violation
An SRP violation occurs when a single class manages multiple independent responsibilities.
Such a class ends up having more than one reason to change, which contradicts SRP.

Common Reasons for SRP Violations

1. Inadequate design analysis before implementation
2. Using procedural programming mindset in object-oriented systems
3. Rushing development due to tight deadlines
4. Creation of overloaded manager or controller classes

Symptoms of SRP Violation
A class violating SRP often exhibits:
• Large size with many unrelated methods
• Poor readability and understanding
• Difficulty in unit testing
• High coupling and low cohesion
• Changes in one feature affecting others

Such classes are commonly referred to as **God Classes** or **Blob Classes**.

Consequences of SRP Violation

1. Difficult maintenance due to complex logic
2. Limited reusability of individual functionalities
3. Higher probability of bugs
4. Complicated testing process
5. Increased dependency between components

Fix for SRP Violations

The primary solution is **Separation of Concerns**, which involves:

• Identifying distinct responsibilities

• Assigning each responsibility to a separate class

Common Techniques Used

• Layered architecture

• Service-based design

• Repository pattern

• MVC (Model–View–Controller) architecture

Part B: Violations of Open/Closed Principle (OCP) Meaning of OCP Violation

An OCP violation occurs when **existing code must be modified** to introduce new functionality.

This approach increases the risk of breaking already working features.

Common Causes of OCP Violations

1. Overuse of if-else or switch-case statements
2. Absence of abstraction
3. Improper use of polymorphism
4. Hard-coded business logic

Symptoms of OCP Violation

• Long conditional chains

• Frequent modification of core classes

• Ripple effects across the system

• Lack of interfaces or abstract classes

Consequences of OCP Violation

1. High chances of regression bugs
2. Poor scalability
3. Increased testing effort
4. Fragile and unstable codebase
5. Reduced reusability

Fix for OCP Violations

The key solution is **using abstraction and polymorphism**.

Design Techniques

• Interfaces
• Abstract classes
• Strategy pattern
• Factory pattern
• Dependency Inversion Principle
• Plugin-based architectures

Comparative Analysis: SRP vs OCP Violations

Aspect	SRP Violation	OCP Violation
Core Issue	Multiple responsibilities	Lack of extensibility
Primary Cause	Poor separation of concerns	Missing abstraction
Major Impact	Difficult maintenance	Limited scalability
Key Symptom	God classes	Long if-else chains
Main Solution	Splitting responsibilities	Introducing interfaces
Design Level	Structural flaw	Behavioral flaw
