DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

# Assignment - 1

**Student Name:** Mohd Shahid
**UID:** 23BCS10258
**Branch:** BE-CSE
**Section/Group:** KRG-2A
**Semester:** 6th
**Date of Submission:** 04/02/26
**Subject Name:** System Design
**Subject Code:** 23CSH-314

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

**Ans:** In object-oriented system design, the **SOLID** principles are widely used to develop software that is easy to understand, maintain, and extend. Among these principles, the **Single Responsibility Principle (SRP)** and **Open/Closed Principle (OCP)** play an important role in writing clean and well-structured code.

1. **Single Responsibility Principle (SRP):**

***"A class should have only one reason to change."***

In simple words:

**One class = One responsibility**

A class should do **only one job**, and do it well.
This means a class should focus on performing a single task. When a class handles multiple responsibilities, it becomes difficult to maintain and may cause unintended issues during modification.

**Why SRP is important**

If a class has multiple responsibilities:

- It becomes hard to understand
- Small changes can break unrelated functionality
- Code becomes difficult to test and maintain

**Example (Violation of SRP):**

```
class Invoice {
    void calculateTotal() {
        // calculate total amount
    }
    void printInvoice() {
        // print invoice
    }
    void saveToDatabase() {
        // save invoice to database
    }
}
```

In the above example, the Invoice class is responsible for calculation, printing, and database operations. Any change in printing format or database logic would require changes in this class, which violates SRP.

**Example (SRP Applied Correctly):**

```
class Invoice {
    void calculateTotal() {
        // calculate total amount
    }
}
class InvoicePrinter {
    void print(Invoice invoice) {
        // print invoice
    }
}
class InvoiceRepository {
    void save(Invoice invoice) {
        // save invoice to database
    }
}
```

Here, each class has a single responsibility, making the design cleaner and easier to maintain.

**2. Open/Closed Principle (OCP):**

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

The Open/Closed Principle states that software entities should be open for extension but closed for modification. This means new functionality should be added without changing existing code, reducing the risk of errors.

**Why OCP is important**

- Reduces bugs in existing working code
- Makes system easier to extend
- Supports scalable system design

**Example (Violation of OCP):**

```
class PaymentProcessor {
    void processPayment(String type) {
        if (type.equals("CreditCard")) {
            // credit card payment logic
        } else if (type.equals("UPI")) {
            // UPI payment logic
        }
    }
}
```

Whenever a new payment method is added, this class must be modified, which violates OCP.

**Example (OCP Applied Correctly):**

```
interface Payment {
   void pay();
}
class CreditCardPayment implements Payment {
   public void pay() {
      // credit card payment logic
   }
}
class UPIPayment implements Payment {
   public void pay() {
      // UPI payment logic
   }
}
class PaymentProcessor {
   void processPayment(Payment payment) {
      payment.pay();
   }
}
```

This design allows new payment methods to be added without modifying existing code.

| Aspect | SRP | OCP |
|---|---|---|
| Focus | Responsibility | Extension |
| Key Idea | One class, one job | Add features without modifying |
| Prevents | God classes | Frequent code changes |
| Achieved Using | Separation of concerns | Abstraction, inheritance, |

**Conclusion**
The Single Responsibility Principle ensures that classes remain focused and maintainable, while the Open/Closed Principle enables systems to grow without affecting existing functionality. Together, these principles help in building scalable and reliable software systems

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

**Ans:** In object-oriented system design, violations of design principles often result in complex and hard-to-maintain software. Among the SOLID principles, the Single Responsibility Principle (SRP) and the Open/Closed Principle (OCP) are commonly violated due to poor design choices. Understanding these violations and their fixes helps in developing scalable and maintainable systems.

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

The Single Responsibility Principle states that a class should have only one responsibility. A violation of SRP occurs when a class performs multiple unrelated tasks.

**Common Causes of SRP Violation**

- Combining business logic, UI logic, and database logic in one class
- Designing "God classes" that control too many operations
- Trying to reduce the number of classes at the cost of clarity

**Example of SRP Violation:**

```
class UserService {
    void registerUser() {
        // user registration logic
    }
    void sendEmail() {
        // send confirmation email
    }
    void saveToDatabase() {
        // store user in database
    }
}
```

In this example, the UserService class is responsible for user registration, sending emails, and database operations. Any change in email logic or database structure would require changes in this class, leading to tight coupling and poor maintainability.

**Problems Caused by SRP Violation**

- Code becomes difficult to understand
- Changes in one feature affect other features
- Testing becomes complicated
- High risk of bugs during modification

**Fix for SRP Violation:**

```
class UserService {
    void registerUser() {
        // user registration logic
    }
}
class EmailService {
    void sendEmail() {
        // send confirmation email
    }
}
```

```
class UserRepository {
    void save() {
        // store user in database
    }
}
```

By separating responsibilities into different classes, each class now has a single purpose. This improves readability, testability, and maintainability.

**Violations of Open/Closed Principle (OCP):**

The Open/Closed Principle states that software entities should be open for extension but closed for modification. A violation occurs when existing code must be modified whenever new functionality is added.

**Common Causes of OCP Violation**

- Heavy use of if-else or switch statements
- Lack of abstraction or interfaces
- Hardcoding logic instead of using polymorphism

**Example of OCP Violation:**

```
class NotificationService {
    void sendNotification(String type) {
        if (type.equals("Email")) {
            // send email
        } else if (type.equals("SMS")) {
            // send SMS
        }
    }
}
```

In this design, adding a new notification type requires modifying the existing class, which violates the Open/Closed Principle and increases the risk of bugs.

**Problems Caused by OCP Violation**

- Existing tested code must be modified repeatedly
- High risk of introducing new bugs
- System becomes rigid and difficult to extend
- Poor scalability

**Fix for OCP Violation:**

```
interface Notification {
    void send();
}

class EmailNotification implements Notification {
    public void send() {
        // send email
    }
}
class SMSNotification implements Notification {
    public void send() {
        // send SMS
    }
}
class NotificationService {
    void send(Notification notification) {
        notification.send();
    }
}
```

Using abstraction and polymorphism allows new notification types to be added without modifying existing code, thus following the Open/Closed Principle.

**Conclusion**
Violations of SRP result in classes with multiple responsibilities, making the system hard to maintain. Violations of OCP lead to rigid designs that require frequent modification of existing code. By applying SRP through separation of concerns and OCP through abstraction, software systems become more flexible, scalable, and reliable.

## Q3. Design an HLD for an Online Examination System applying these principles.

**Ans:** **Functional Requirements:**

- User should be able to register and login
- System should support different roles (Student, Admin)
- Admin should be able to create and schedule exams
- Admin should be able to add/manage questions
- Student should be able to attempt exams online
- System should evaluate answers automatically
- System should generate results
- System should notify users about exam and results

**Non Functional Requirements:**

A. Scalability
- System should handle multiple students simultaneously during exams

B. Security
- Secure authentication and authorization
- Exam data should be protected

C. Performance
- Low response time during exam submission

D. Reliability
- No data loss during exam submission

E. Maintainability
- Easy to add new exam or evaluation types

**Core Entities:**

- User
- Exam
- Question
- Answer
- Result

**API Endpoints:**

Auth & User Service APIs
- POST /login
- POST /register
- GET /users/{id}

Exam Management Service APIs
- POST /exams
- GET /exams

Question Bank Service APIs
- POST /questions
- GET /questions/{examId}

Evaluation Service APIs
- POST /evaluate

Result Service APIs
- GET /results/{userId}

## High Level Diagram: