DEPARTMENTOF

COMPUTERSCIENCE&ENGINEERING

Discover. Learn. Empower.

## Assignment

**Student Name :** Reyansh Goswami
**UID:** 23BCS11136
**Branch:** BE-CSE
**Section/Group:** KRG_2B
**Semester:** 6th
**Date of Submission:** 03/02/26
**Subject Name:** System Design
**Subject Code:** 23CSH-314

**Aim:**

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

**Objectives:**

1. To understand SOLID principle inorder to explain what is SRP and OCP
2. To make an .io file representing the Online Examination System.

## Single Responsibility Principle (SRP) - The S in SOLID principle:

- The Single Responsibility Principle operates on the idea that a class should have one, and only one, reason to change. Fundamentally, this means a class should encapsulate a single job or function within the software system.
- When a class assumes multiple responsibilities (e.g., managing user data *and* sending emails), it becomes tightly coupled. Modifying the logic for one responsibility (like changing the email provider) might inadvertently break the code related to the other responsibility (user data management). Separation ensures that changes are isolated and safe.

Example of Violation:

```
 1. class UserManager {
 2.     void registerUser(String username) {
 4.     }
 5.
 6.     void logError(String error) {
 8.         System.out.println("Logging error: " + error);
 9.     }
10.
11.     void sendWelcomeEmail(String email) {
13.     }
14. }
15.
```

*Critique:* The UserManager class is handling authentication, logging, and email notifications. A change in the logging format or email protocol would force a change in this core user class.

Example of Correct:

```
class UserRegistry {
    void registerUser(String username) {
    }
}

class ErrorLogger {
    void logError(String error) {
    }
}

class EmailService {
    void sendWelcomeEmail(String email) {
    }
}

```

Benefit: This structure significantly lowers coupling. You can now swap the ErrorLogger from a file-based system to a database-based system without ever touching the UserRegistry code.

## Open Closed Principle (OCP) – O of SOLID:

- The Open-Closed Principle asserts that software entities (classes, modules, functions) should be **open for extension** but **closed for modification**.
- This principle encourages developers to design systems where new features can be added by creating new code (classes/modules) rather than rewriting existing, tested code. This is usually achieved through polymorphism and interfaces.

Example of Violation:

```
class AreaCalculator {
    public double calculateArea(Object shape) {
        if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.length * r.width;
        } else if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return Math.PI * c.radius * c.radius;
        }
        return 0;
    }
}

```

*Critique:* Every time a new shape (e.g., Triangle) is introduced, the AreaCalculator class must be modified and recompiled, violating OCP.

Example of Correct:

```
interface Shape {
    double calculateArea();
}

class Rectangle implements Shape {
    double length;
    double width;
    public double calculateArea() { return length * width; }
}

class Circle implements Shape {
    double radius;
    public double calculateArea() { return Math.PI * radius * radius; }
}

class AreaCalculator {
    public double calculateArea(Shape shape) {
        return shape.calculateArea();
    }
}

```

Benefit**:** The system is extensible. Adding a Triangle class requires no changes to the AreaCalculator or existing shapes, preventing regression bugs.

## #Violation of SRP and OCP along with their fixes:

1. **Violation in SRP-**
    - SRP violations occur when a class becomes a "God Object"—knowing too much or doing too much. This is common when developers mix **business logic** (rules about data) with **infrastructure logic** (saving to DB, printing, UI rendering).

    Example Violation:

```
class Book {
    String title;
    String author;

    // Responsibility 1: Data Management
    String getTitle() { return title; }

    // Responsibility 2: Presentation/Formatting
    void printToConsole() {
        System.out.println("Title: " + title + " by " + author);
    }

    // Responsibility 3: Persistence
    void saveToDatabase() {
        // SQL code to save book
    }
}

```

*Issue:* The Book class changes if the library rules change, if the printing format changes, or if the database schema changes.

**Fix:** Apply **Decomposition**. Delegate each distinct responsibility to a specialized class.

Example of Fix:

```
class Book {
    String title;
    String author;
}

class BookPrinter {
    void printToConsole(Book book) {
        System.out.println("Title: " + book.title);
    }
}


class BookRepository {
    void save(Book book) {

    }
}

```

Result: The Book class is now stable. Changing the database from MySQL to MongoDB only requires changes in BookRepository, leaving the rest of the application untouched.

## 2. Violation in OCP-

- OCP is most commonly violated by the use of rigid conditional logic (like switch statements or chained if-else blocks) that checks for specific types to determine behavior. This indicates that the calling class is too aware of the implementation details of its dependencies.

Example of Violation:

```
class NotificationSender {
    public void send(String type, String message) {
        if (type.equals("SMS")) {
            System.out.println("Sending SMS: " + message);
        } else if (type.equals("Email")) {
            System.out.println("Sending Email: " + message);
        }
    }
}

```

*Issue:* To add "Push Notification," you must modify the send method, which risks breaking existing SMS or Email functionality.

**Fix:** Use **Abstraction** (Interfaces) and **Polymorphism**. The main class should depend on an abstract interface, not concrete strings or types.

Example of Fix:

```
interface Notification {
    void send(String message);
}
class SMSNotification implements Notification {
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

class EmailNotification implements Notification {
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}
class PushNotification implements Notification {
    public void send(String message) {
        System.out.println("Sending Push: " + message);
    }
}

```

**Result:** The system is now compliant. You can add infinite new notification types by simply creating new classes implementing the Notification interface, without ever opening or changing the original logic.

## High level diagram of Online exam system