Name: Aayush Aman

UID: 23BCS14046

SOLID Principles: SRP and OCP

System Design Assignment

Q1. Single Responsibility Principle (SRP) and Open/Closed Principle (OCP)

1.1 Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one job or responsibility.

The SRP states that every module, class, or function should be responsible for a single part of the program's functionality. This principle helps in reducing coupling, improving code maintainability, and making the system easier to understand and modify. When a class has multiple responsibilities, changes in one responsibility can affect the implementation of others, leading to fragile code.

Real-Life Example

Consider an Employee Management System in a company. An Employee class should only handle employee data (name, ID, department). It should not also handle database operations, salary calculations, and email notifications. Each of these responsibilities should be delegated to separate classes:

- **Employee class** – stores employee information
- **EmployeeRepository** – handles database operations
- **SalaryCalculator** – calculates salaries and bonuses
- **EmailService** – sends email notifications

Code Example

Violation of SRP:

```
class Employee {
        private String name;
        private double salary;
        // Multiple responsibilities in one class
        public void calculateSalary() { /* calculation logic */ }
```

```
    public void saveToDatabase() { /* database logic */ }
    public void sendEmail() { /* email logic */ }
    }
```

Following SRP:

```
class Employee {
    private String name;
    private double salary;
    // Only stores employee data
}
class SalaryCalculator {
    public double calculate(Employee emp) { /* logic */ }
}
class EmployeeRepository {
    public void save(Employee emp) { /* database logic */
} }
class EmailService {
    public void send(String email) { /* email logic */ }
}
```

1.2 Open/Closed Principle (OCP)

Definition: Software entities (classes, modules, functions) should be open for extension but closed for modification.

The OCP suggests that you should be able to add new functionality to a system without changing existing code. This is achieved through abstraction using interfaces or abstract classes and polymorphism. The principle helps in minimizing the risk of breaking existing, tested code when adding new features, making the system more stable and maintainable.

Real-Life Example

Consider an E-commerce Payment System. Initially, the system supports credit card payments. Later, the business wants to add PayPal, UPI, and cryptocurrency payments. Instead of modifying the existing payment processing code each time a new payment method is added, we use the OCP by creating a payment interface. Each payment method implements this interface, and new methods can be added without touching the existing payment processor code.

Code Example

Violation of OCP:

```
class PaymentProcessor {
    public void processPayment(String type, double amount) {
        if (type.equals("CREDIT_CARD")) {// Credit card logic}
        else if (type.equals("PAYPAL")) {// PayPal logic}
```

```
            // Adding new payment requires modifying this class
        }
}
```

Following OCP:

```
interface PaymentMethod {
        void processPayment(double amount);
}
class CreditCardPayment implements PaymentMethod {
        public void processPayment(double amount) {// Credit card specific logic}
}
class PayPalPayment implements PaymentMethod {
        public void processPayment(double amount) {// PayPal specific logic}
}
class PaymentProcessor {
        public void process(PaymentMethod method, double amount) {      method.processPayment(amount);
        }
}
// Adding UPI doesn't require modifying existing code
class UPIPayment implements PaymentMethod {
        public void processPayment(double amount) {// UPI specific logic}
}
```

Q2. Violations and Fixes in SRP and OCP

2.1 Common SRP Violations and Fixes

Violation 1: God Class (Multiple Responsibilities)

Problem: A single class handles data validation, business logic, database operations, and UI formatting. This creates a tightly coupled system where any change in one area requires understanding and potentially modifying the entire class.

Example Scenario:

```
class UserManager {
        public void createUser(String name, String email) {
        // Validation responsibility
        if (name == null || name.isEmpty()) {
                throw new Exception("Invalid name");
        }
        // Security responsibility
        String hashedPassword = hashPassword("default");
```

```
        // Database responsibility
        database.save(name, email, hashedPassword);
        // Notification responsibility
        emailService.send(email, "Welcome!");
        }
}
```

Fix: Separate Each Responsibility

```
class UserValidator {
        public void validate(User user) {
                if (user.getName() == null) {
                        throw new Exception("Invalid name");
                }
        }
}
class PasswordService {
        public String hashPassword(String password) {
        return BCrypt.hash(password);
        }
}
class UserRepository {
        public void save(User user) {
        database.insert(user);
        }
}
class NotificationService {
        public void sendWelcome(String email) {
        emailService.send(email, "Welcome!");
        }
}
class UserService {
        // Orchestrates the process using specialized classes
        public void createUser(String name, String email) {
                User user = new User(name, email);
                validator.validate(user);       user.setPassword(passwordService.hashPassword("default"));
                repository.save(user);
                notificationService.sendWelcome(email);
        }
}
```

Violation 2: Mixed Data and Business Logic

Problem: Data classes contain business calculations, formatting logic, and persistence methods, making them difficult to reuse and test.

Example Scenario:

```
class Invoice {
        private List<Item> items;
        // Business logic
        public double calculateTotal() { /* ... */ }
        // Persistence
        public void saveToDatabase() { /* ... */ }
        // Formatting
        public String generatePDF() { /* ... */ }
}
```

Fix: Separate Data, Logic, and Services

```
class Invoice {
private List<Item> items;
// Only getters and setters }
class InvoiceCalculator {
public double calculateTotal(Invoice invoice) {// Business logic}
}
class InvoiceRepository {
public void save(Invoice invoice) {// Database operations}
}
class InvoicePDFGenerator {
public String generate(Invoice invoice) {// PDF formatting logic}
}
```

2.2 Common OCP Violations and Fixes

Violation 1: Conditional Logic for Type Checking

Problem: Using if-else or switch statements to handle different types means every new type requires modifying the existing code, violating the closed for modification principle.

Example Scenario:

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

```
// Adding Triangle requires modifying this method
return 0;    }}
```

Fix: Use Polymorphism and Abstraction

```
abstract class Shape {
        public abstract double calculateArea();
}
class Circle extends Shape {
        private double radius;
        public double calculateArea() {
                return Math.PI * radius * radius;
        }
}
class Rectangle extends Shape {
        private double length, width;
        public double calculateArea() {
        return length * width;
        }
}
class ShapeCalculator {
        public double calculateTotalArea(List<Shape> shapes) {
                double total = 0;
                for (Shape shape : shapes) {
                total += shape.calculateArea();
        }
                return total;
        }
}
// Adding Triangle doesn't modify existing code
class Triangle extends Shape {
        private double base, height;
        public double calculateArea() {
        return 0.5 * base * height;
        }
}
```

Violation 2: Hard-coded Behavior

Problem: Business rules are hard-coded into methods, making it impossible to extend behavior without modifying the source code.

Example Scenario:

```
class DiscountCalculator {
        public double getDiscount(String customerType, double amount) {
```

```
        if (customerType.equals("REGULAR")) {
            return amount * 0.05;
        }
        else if (customerType.equals("PREMIUM")) {
            return amount * 0.10;
        }
        // Adding VIP requires code modification
        return 0;
    }
}
```

Fix: Use Strategy Pattern

```
interface DiscountStrategy {
    double calculateDiscount(double amount);
}
class RegularDiscount implements DiscountStrategy {
    public double calculateDiscount(double amount) {
        return amount * 0.05;
    }
}
class PremiumDiscount implements DiscountStrategy {
    public double calculateDiscount(double amount) {
    return amount * 0.10;
    }
}
class DiscountCalculator {
    private DiscountStrategy strategy;
    public DiscountCalculator(DiscountStrategy strategy) {    this.strategy = strategy;
    }
    public double getDiscount(double amount) {
    return strategy.calculateDiscount(amount);
    }
}
// Adding VIP doesn't modify existing code
class VIPDiscount implements DiscountStrategy {
    public double calculateDiscount(double amount) {
        return amount * 0.20;
    }
}
```

2.3 Benefits of Following SRP and OCP

SRP Benefits:

• **Improved Maintainability:** Each class has a focused purpose, making it easier to understand and modify.

• **Better Testability:** Smaller, focused classes are easier to unit test in isolation.

• **Reduced Coupling:** Changes in one responsibility don't affect unrelated functionalities.

• **Easier Collaboration:** Different team members can work on different responsibilities simultaneously.

OCP Benefits:

• **Reduced Risk:** Adding new features doesn't require modifying and potentially breaking existing code.

• **Increased Flexibility:** System can be extended with new behaviors without touching stable code.

• **Better Code Reuse:** Abstraction promotes using common interfaces across different implementations.

• **Scalability:** System can grow with minimal changes to the existing codebase.

Conclusion

Both SRP and OCP are fundamental principles that work together to create maintainable, flexible, and robust software systems. SRP ensures that each component has a single, well-defined purpose, while OCP ensures that the system can evolve and grow without requiring modifications to existing, tested code. Following these principles leads to cleaner architecture, easier debugging, better team collaboration, and ultimately, higher-quality software that can adapt to changing requirements.