Assignment: 1

Subject Name: System Design
Subject Code: 23CSH-307

Submitted By: Devjot Singh
UID: 23BCS10864
Section/Group: 23BCSKRG-2A
Branch: BE-CSE

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

1. Single Responsibility Principle (SRP)

Definition

The **Single Responsibility Principle** states that:

A class should have only one reason to change, meaning it should perform only **one well-defined responsibility**.

This principle comes from **SOLID design principles** proposed by **Robert C. Martin (Uncle Bob)**.

Why SRP is important

1. **Improves maintainability** – Changes affect only one small part of the system.
2. **Enhances readability** – Classes become easier to understand.
3. **Simplifies testing** – Smaller responsibilities → easier unit tests.
4. **Reduces coupling** – Independent modules don't interfere with each other.

Bad Example (SRP Violation)

```
class Student {
   String name;
   int marks;

   void calculateGrade() {
      // logic to calculate grade
   }

   void saveToDatabase() {
      // database saving logic
   }
```

```
    void printReport() {
        // printing logic
    }
}
```

Problem

This class has **three responsibilities**:

- Grade calculation → **business logic**
- Saving to DB → **data persistence**
- Printing → **presentation**

So it has **multiple reasons to change**:

- If grading rule changes
- If DB changes
- If printing format changes

➡️ **SRP is violated.**

Correct Example (SRP Applied)

```
class Student {
    String name;
    int marks;
}
class GradeCalculator {
    char calculateGrade(Student s) {
        // grade logic
        return 'A';
    }
}
class StudentRepository {
    void save(Student s) {
        // DB logic
    }
}
class ReportPrinter {
    void print(Student s) {
        // print logic
    }
}
```

Result

Each class now has **one responsibility**:

- GradeCalculator → grading
- StudentRepository → persistence
- ReportPrinter → presentation

➡️ **SRP satisfied.**

---

2. Open/Closed Principle (OCP)

Definition

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

Meaning:

- You **should not modify existing code**
- Instead, **extend behavior using new classes**

Why OCP is important

1. Prevents **breaking working code**
2. Encourages **abstraction & polymorphism**
3. Improves **scalability**
4. Makes systems **future-proof**

Bad Example (OCP Violation)

```
class DiscountCalculator {
   double calculateDiscount(String customerType, double amount) {

      if (customerType.equals("REGULAR"))
         return amount * 0.1;

      else if (customerType.equals("PREMIUM"))
         return amount * 0.2;

      else if (customerType.equals("VIP"))
         return amount * 0.3;

      return 0;
   }
}
```

Problem

Every time a **new customer type** is added:

- You must **modify this class**
- Risk of **introducing bugs**

➡️ **OCP violated.**

Correct Example (OCP Applied)

Step 1 – Create abstraction

```
interface DiscountStrategy {
   double calculate(double amount);
}
```

Step 2 – Implement strategies

```
class RegularDiscount implements DiscountStrategy {
   public double calculate(double amount) {
      return amount * 0.1;
```

```
    }
}
class PremiumDiscount implements DiscountStrategy {
    public double calculate(double amount) {
        return amount * 0.2;
    }
}
class VipDiscount implements DiscountStrategy {
    public double calculate(double amount) {
        return amount * 0.3;
    }
}
```

Step 3 – Use abstraction

```
class DiscountCalculator {
    double calculateDiscount(DiscountStrategy strategy, double amount) {
        return strategy.calculate(amount);
    }
}
```

Result

To add **new discount type**:

```
class FestivalDiscount implements DiscountStrategy {
    public double calculate(double amount) {
        return amount * 0.25;
    }
}
```

No change in old code.

➡️ **OCP satisfied.**

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

1. SRP Violations

Common SRP Violations

(a) God Class

One class doing everything:

```
class OrderManager {
    void createOrder() {}
    void calculatePrice() {}
    void saveToDB() {}
    void sendEmail() {}
}
```

Fix

Split into:

- OrderService
- PriceCalculator
- OrderRepository
- EmailService

(b) Mixed UI + Business Logic

```
class Login {
   void validateUser() {}
   void displayError() {}
}
```

Fix

- AuthService → validation
- LoginUI → display

(c) Multiple Actors Changing Same Class

If **different teams** modify same class → SRP broken.

Fix: Separate by responsibility owner.

---

2. OCP Violations

Common OCP Violations

(a) Long if-else or switch

```
if (shape == "circle") ...
else if (shape == "rectangle") ...
```

Fix → Polymorphism

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

(b) Modifying core class for new feature

Every feature → edit same file.

Fix

- Use **interfaces**
- Use **inheritance**
- Use **Strategy / Factory patterns**

(c) Hard-coded dependencies

```
class Payment {
   void pay() {
      CreditCard card = new CreditCard();
   }
}
```

Fix → Dependency Injection

```
class Payment {
   PaymentMethod method;

   Payment(PaymentMethod method) {
      this.method = method;
   }
}
```

Summary Table

Principle	Key Idea	Violation Sign	Fix
SRP	One class → one responsibility	God class, mixed logic	Split classes
OCP	Extend without modifying	if-else chains	Use abstraction & polymorphism

```
    Order(OrderStatus status) {
        this.status = status;
    }
}
```

Design Explanation

- OrderStatus enum restricts status values.
- Avoids invalid states like "shiped" or "done".
- Improves system reliability and clarity.

1. Using Interface and Enum Together (Real-World Design)

```
enum UserRole {
    ADMIN,
    CUSTOMER
}
interface User {
    void accessLevel();
}
class Admin implements User {
    public void accessLevel() {
        System.out.println("Full access granted");
    }
}
class Customer implements User {
    public void accessLevel() {
        System.out.println("Limited access granted");
    }
}
```

Design Benefit

- Enum controls roles
- Interface controls behavior
- Together they produce clean and scalable architecture

2. Conclusion

Interfaces and enums play a vital role in software design:

- Interfaces support abstraction, flexibility, and clean architecture.
- Enums ensure data consistency, type safety, and readable code.

Using both appropriately leads to robust, maintainable, and extensible software systems, which is a key goal of good software design.

Q2. Discuss how interfaces enable loose coupling with example.

Answer:

1. Meaning of Loose Coupling

Loose coupling refers to a design approach in which **components of a system are minimally dependent on each other**. Changes in one component **do not significantly affect** other components.

Interfaces play a major role in achieving loose coupling by allowing classes to **depend on abstractions rather than concrete implementations**.

2. How Interfaces Enable Loose Coupling

An **interface defines a contract** without specifying implementation details. Classes interact through the interface instead of directly depending on a specific class.

This leads to:

- Reduced dependency between modules
- Improved flexibility and maintainability
- Easier testing and extension of the system

3. Problem Without Interface (Tightly Coupled Design)

```
class CreditCardPayment {
   void pay(double amount) {
      System.out.println("Paid using Credit Card");
   }
}
class ShoppingCart {
   CreditCardPayment payment = new CreditCardPayment();

   void checkout(double amount) {
      payment.pay(amount);
   }
}
```

Issues

- ShoppingCart is directly dependent on CreditCardPayment
- Adding another payment method (UPI, Cash, etc.) requires modifying ShoppingCart
- Violates **Open–Closed Principle**

4. Solution Using Interface (Loosely Coupled Design)

Step 1: Create an Interface

```
interface Payment {
   void pay(double amount);
}
```

Step 2: Implement the Interface

```
class CreditCardPayment implements Payment {
   public void pay(double amount) {
      System.out.println("Paid using Credit Card");
   }
}

class UpiPayment implements Payment {
   public void pay(double amount) {
      System.out.println("Paid using UPI");
   }
}
```

Step 3: Use Interface in Client Class

```
class ShoppingCart {
   Payment payment;

   ShoppingCart(Payment payment) {
      this.payment = payment;
```

```
    }

    void checkout(double amount) {
        payment.pay(amount);
    }
}
```

5. How Loose Coupling Is Achieved

- ShoppingCart depends on the **Payment interface**, not on a specific class
- Payment method can change **without modifying ShoppingCart**
- New payment types can be added easily

Example usage:

```
ShoppingCart cart1 = new ShoppingCart(new CreditCardPayment());
cart1.checkout(1000);
ShoppingCart cart2 = new ShoppingCart(new UpiPayment());
cart2.checkout(500);
```

6. Benefits of Loose Coupling Using Interfaces

1. **Flexibility**
    - Easily switch implementations at runtime
2. **Maintainability**
    - Changes in one class do not affect others
3. **Scalability**
    - New features can be added with minimal changes
4. **Better Testing**
    - Interfaces allow mocking during unit tests
5. **Clean Architecture**
    - Follows SOLID design principles

7. Conclusion

Interfaces enable loose coupling by separating **what a class does** from **how it does it**. By programming to an interface rather than a concrete class, software systems become **flexible, extensible, and easier to maintain**, which is essential for good software design.