Assignment: 1

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

Submitted By: Jaskirat Singh
UID: 23BCS10856
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:

Every class should have only one responsibility and therefore only one reason to change.

In simple words, a class must focus on performing one specific task instead of handling multiple unrelated operations.

This principle is a key part of the SOLID design principles introduced by Robert C. Martin (Uncle Bob).

Importance of SRP

1. Better maintainability – Modifications remain limited to a single module.
2. Improved clarity – Smaller classes are easier to read and understand.
3. Easier testing – Unit testing becomes simple when responsibilities are isolated.
4. Lower coupling – Independent components reduce unwanted dependencies.

Incorrect Example (Violation of SRP)

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

   void calculateGrade() {
      // grading logic
   }

   void saveToDatabase() {
```

```
        // database operation
    }

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

Issue

This class performs three different roles:

- Business logic → grade computation
- Persistence logic → database storage
- Presentation logic → report printing

Because of this, the class will change if:

- Grading rules are modified
- Database structure changes
- Report format updates

➡️ **Hence, SRP is violated.**

Correct Example (Applying SRP)

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

Outcome

Each class now handles one clear responsibility:

- GradeCalculator → grading
- StudentRepository → storage
- ReportPrinter → presentation

➡️ **SRP is successfully followed.**

---

2. Open/Closed Principle (OCP)

Definition

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

This means:

- Existing code should remain unchanged
- New behavior should be introduced through extension mechanisms like inheritance or interfaces.

Significance of OCP

1. Prevents damage to stable code.
2. Promotes abstraction and polymorphism.
3. Supports system scalability.
4. Makes applications adaptable to future requirements.

Incorrect Example (Violation of OCP)

```
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

Whenever a new customer category appears:

- The same class must be edited again
- This increases bug risk and maintenance cost

➡️ **Therefore, OCP is broken.**

Correct Example (Applying OCP)

Step 1 – Define abstraction

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

Step 2 – Provide implementations

```
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 in calculator

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

Result

Adding a new discount type requires only a new class:

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

No modification in previous code.

➡️ **OCP is maintained.**

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

Typical SRP Problems

(a) God Class

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

A single class controls multiple independent tasks.

Solution:
Divide into specialized classes such as:

- OrderService
- PriceCalculator
- OrderRepository
- EmailService

(b) Mixing UI and Business Logic

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

Solution:

- AuthService → validation logic
- LoginUI → user interface display

(c) One Class Modified by Multiple Stakeholders

If several teams frequently edit the same class, SRP is not satisfied.
Solution:
Separate modules based on distinct responsibilities or actors.

---

2. Violations of OCP
Common OCP Issues
(a) Large if-else or switch blocks

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

Fix:
Apply polymorphism using:

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

(b) Editing core logic for each new feature

Repeated modification of a stable class signals poor extensibility.
Fix:

- Introduce interfaces or abstract classes
- Use design patterns like Strategy or Factory.

(c) Hard-coded object creation

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

Fix – Dependency Injection

```
class Payment {
   PaymentMethod method;

   Payment(PaymentMethod method) {
```

```
        this.method = method;
    }
}
```

This enables easy extension without modification.

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.