ASSIGNMENT- 1

Name: Harsh Khanna **Branch: BE-CSE**

Semester: 6th **Subject Name: SD**

UID: 23BCS10540 **Section: 23BCS_KRG1A**

Submission date: 4-2-26 **Subject Code: 23CSH-314**

Q1. Violations of Single Responsibility Principle (SRP) and Their Fixes

The Single Responsibility Principle states that a class should have only one responsibility and only one reason to change. If a class handles multiple unrelated tasks, it becomes hard to maintain and test.

SRP Violation

```
class OrderService:
    def create_order(self, order):
        print("Order created")

    def save_to_db(self, order):
        print("Order saved to database")

    def send_email(self, order):
```

```
        print("Confirmation email sent")
```

Why this violates SRP:
This class is responsible for:

- Business logic (creating order)
- Data storage (database)
- Communication (email)

If the database changes, email system changes, or order logic changes, the same class must be modified. This gives the class multiple reasons to change, which violates SRP. It also makes the class harder to test and debug.

Fix for SRP Violation

```
class OrderCreator:
    def create_order(self, order):
        print("Order created")


class OrderRepository:
    def save_to_db(self, order):
        print("Order saved to database")


class EmailService:
    def send_email(self, order):
        print("Confirmation email sent")
```

Why this fixes SRP:
Each class now has only one responsibility. Changes in database logic affect only OrderRepository, and changes in email logic affect only EmailService. This improves maintainability and reduces side effects.

Violations of Open/Closed Principle (OCP) and Their Fixes

What is OCP?

The Open/Closed Principle states that classes should be open for extension but closed for modification. New behavior should be added without changing existing, working code.

OCP Violation

```
class DiscountCalculator:
    def get_discount(self, customer_type, amount):
        if customer_type == "regular":
            return amount * 0.05
        elif customer_type == "premium":
            return amount * 0.10
```

Why this violates OCP:
If a new customer type (for example, "student") is added, this class must be modified:

```
elif customer_type == "student":
    return amount * 0.15
```

This means existing code is edited every time a new rule is added. It increases the risk of bugs and breaks OCP.

Fix for OCP Violation

```
from abc import ABC, abstractmethod


class DiscountStrategy(ABC):
    @abstractmethod
```

```
    def get_discount(self, amount):
        pass


class RegularDiscount(DiscountStrategy):
    def get_discount(self, amount):
        return amount * 0.05


class PremiumDiscount(DiscountStrategy):
    def get_discount(self, amount):
        return amount * 0.10
```

Usage:

```
def calculate_discount(strategy, amount):
    return strategy.get_discount(amount)


discount = calculate_discount(RegularDiscount(), 1000)
```

Why this fixes OCP:
To add a new discount type, a new class (for example, StudentDiscount) can be created without modifying existing classes. The original code remains unchanged, so the system is safer and easier to extend.

Single Responsibility Principle (SRP)

Definition:
The Single Responsibility Principle says that a class should have only one responsibility. This means it should have only one main job and only one reason

to change. If a class is doing multiple unrelated tasks, then changes in one part can affect other parts, which makes the code harder to maintain.

Example (Violation of SRP)

```
class UserService:
    def create_user(self, name):
        print("User created")

    def save_to_db(self, name):
        print("Saved to database")

    def send_email(self, name):
        print("Email sent")
```

This class handles user creation, database operations, and email sending. These are different responsibilities. If the database logic changes or the email system changes, this same class must be modified. This violates SRP because the class has multiple reasons to change.

Improved Example (Follows SRP)

```
class UserCreator:
    def create_user(self, name):
        print("User created")

class UserRepository:
    def save_to_db(self, name):
        print("Saved to database")
```

```
class EmailService:
    def send_email(self, name):
        print("Email sent")
```

Now each class has only one responsibility. Changes in database or email logic affect only their own classes, making the code easier to manage and test.

Open/Closed Principle (OCP)

Definition:
The Open/Closed Principle says that software components should be open for extension but closed for modification. This means new functionality should be added by extending existing code, not by changing working code. This reduces the risk of introducing bugs.

Example (Violation of OCP)

```
class AreaCalculator:
    def area(self, shape):
        if shape["type"] == "circle":
            return 3.14 * shape["r"] * shape["r"]
        elif shape["type"] == "rectangle":
            return shape["w"] * shape["h"]
```

If a new shape is added, this class must be edited again. Each change increases the chance of breaking existing logic, so this design violates OCP.

Improved Example (Follows OCP)

```
from abc import ABC, abstractmethod
```

```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r):
        self.r = r

    def area(self):
        return 3.14 * self.r * self.r

class Rectangle(Shape):
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h
```

To add a new shape, a new class (for example, Triangle) can be created without modifying the existing classes. This follows OCP.

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

SRP Violation

```
class OrderService:
    def create_order(self, order):
        print("Order created")

    def save_to_db(self, order):
        print("Order saved to database")

    def send_email(self, order):
        print("Confirmation email sent")
```

Issue:
This class handles order creation, database operations, and email sending. These are different concerns. If the database logic changes or the email system changes, this same class must be modified. This increases coupling, makes testing harder, and increases the risk of bugs.

Fix:

```
class OrderCreator:
    def create_order(self, order):
        print("Order created")


class OrderRepository:
    def save_to_db(self, order):
```

```
        print("Order saved to database")


class EmailService:
    def send_email(self, order):
        print("Confirmation email sent")
```

Separating responsibilities reduces side effects. Changes in one part do not affect the others, and each class becomes easier to test and maintain.

Violations of OCP and Their Fixes

OCP Violation

```
class DiscountCalculator:
    def get_discount(self, customer_type, amount):
        if customer_type == "regular":
            return amount * 0.05
        elif customer_type == "premium":
            return amount * 0.10
```

Issue:
Every time a new customer type is added, this class must be modified. Repeated changes to the same class increase the chance of breaking existing logic and make the code harder to maintain.

Fix:

```
from abc import ABC, abstractmethod


class DiscountStrategy(ABC):
```

```
    @abstractmethod
    def get_discount(self, amount):
        pass


class RegularDiscount(DiscountStrategy):
    def get_discount(self, amount):
        return amount * 0.05


class PremiumDiscount(DiscountStrategy):
    def get_discount(self, amount):
        return amount * 0.10
```

New discount types can be added by creating new classes without changing existing code. This reduces risk and keeps existing behavior stable.