P P SAVANI UNIVERSITY

TUTORIAL NO.: 10
ON
SOFTWARE ENGINEERING(SSCS3010)

TITLE: Identify the design principle that is being violated in relation to the given scenario.

BACHELOR OF SCIENCE IN INFORMATION TECHNOLOGY (BSC-IT)

SUBMITTED TO:	**SUBMITTED BY:**
Name: MS. HEMANGINI MEHTA(HGM)	**Name: RAJ MO FAHIM ZAKIR**
Designation: ASSISTANT PROFESSOR	**Enrollment: 23SS02IT161**
P P Savani University	**BSCIT5B-Batch 2023-26**

Faculty Signature: ____________

INSTITUTE OF COMPUTER SCIENCE AND APPLICATIONS
P P SAVANI UNIVERSITY
MANGROL, SURAT- 394125 (GUJARAT)

TUTORIAL-10 **Date:19/09/2025**

Aim: Identify the design principle that is being violated in relation to the given scenario.

Example 1 – Single Responsibility Principle (SRP) Violation

Scenario:
A class is doing many tasks at the same time.

```
class Report {
    public void createReport() { }
    public void printReport() { }
    public void saveReport() { }
}
```

Violation:
This class has **too many responsibilities** (creating, printing, saving).

Correct Way:

- Make separate classes: `ReportCreator`, `ReportPrinter`, `ReportSaver`.
- Each class has only one job → easy to maintain.

Example 2 – Open/Closed Principle (OCP) Violation

Scenario:
A payment class uses **if-else** for every new method of payment.

```
class Payment {
    public void process(String type) {
        if(type.equals("CreditCard")) { }
        else if(type.equals("PayPal")) { }
    }
}
```

Violation:
Whenever a new payment method is added, the class must be changed.

Correct Way:

- Use an **interface** for `PaymentMethod`.
- Add new classes like `CreditCard`, `PayPal`, `UPI` without changing old code.

Example 3 – Liskov Substitution Principle (LSP) Violation

Scenario:
A subclass changes the behavior of the parent class in a way that breaks substitutability.

```
class Bird {
    public void fly() { }
}

class Ostrich extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Ostriches
cannot fly");
    }
}
```

Violation:
The `Ostrich` cannot replace `Bird` without breaking the program, because `Bird` promises `fly()`.

Correct Way:

- Create a better hierarchy:

```
interface Bird { }
interface FlyableBird extends Bird {
    void fly();
}

class Sparrow implements FlyableBird {
    public void fly() { }
}

class Ostrich implements Bird { }
```

- Now, `Ostrich` doesn't break the LSP.

Example 4 – Interface Segregation Principle (ISP) Violation

Scenario:
An interface forces classes to implement methods they don't need.

```
interface Worker {
    void work();
    void eat();
}

class Robot implements Worker {
    public void work() { }
    public void eat() { // Robots don't eat
        throw new UnsupportedOperationException();
    }
}
```

Violation:
`Robot` is forced to implement `eat()`, which doesn't make sense.

Correct Way:
Split into smaller interfaces:

```
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Human implements Workable, Eatable {
    public void work() { }
    public void eat() { }
}
```

```
class Robot implements Workable {
    public void work() { }
}
```

Example 5 – Dependency Inversion Principle (DIP) Violation

Scenario:
A high-level class depends directly on low-level classes instead of abstractions.

```
class LightBulb {
    public void turnOn() { }
    public void turnOff() { }
}

class Switch {
    private LightBulb bulb = new LightBulb();
    public void operate() {
        bulb.turnOn();
    }
}
```

Violation:
`Switch` is tightly coupled with `LightBulb`. If we want a `Fan`, we must modify `Switch`.

Correct Way:
Depend on abstraction (interface):

```
interface Switchable {
    void turnOn();
    void turnOff();
}

class LightBulb implements Switchable { ... }
class Fan implements Switchable { ... }

class Switch {
    private Switchable device;
    public Switch(Switchable device) {
        this.device = device;
    }
    public void operate() {
        device.turnOn();
    }
}
```

Summary in Simple Words:

- **SRP** → One class = One job.
- **OCP** → Don’t modify old code, just extend.
- **LSP** → Subclass must follow parent class rules.
- **ISP** → Don’t force classes to implement unnecessary methods.
- **DIP** → Depend on abstractions, not concrete classes.