# SOLID Principles in Software Development

1. Single Responsibility Principle (SRP)

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

```
# Violating SRP: This class handles both employee data and saving it to the database.
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def save_to_database(self):
        pass  # Code to save employee data to the database

# Following SRP: Separate the employee data from database handling
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class EmployeeDatabase:
    def save(self, employee):
        pass  # Code to save employee to database
```

## 2. Open/Closed Principle (OCP)

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

```
# Violating OCP: Modifying the shape area calculation every time a new shape is added
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

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

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

# Following OCP: Adding a new shape without modifying existing code
class ShapeCalculator:
    def calculate_area(self, shape):
        return shape.area()
```

## 3. Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program.

```
# Violating LSP: A Square class that overrides the behavior of Rectangle in an unexpected way
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)  # This may violate LSP as a square is a specific type of rectangle

# Following LSP: Defining Square and Rectangle separately
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side
```

## 4. Interface Segregation Principle (ISP)

A client should not be forced to implement interfaces it does not use.

```
# Violating ISP: A large interface that forces implementation of unrelated methods
class Machine:
    def print(self):
        pass

    def scan(self):
        pass

    def fax(self):
        pass
```

```
# Following ISP: Splitting interfaces into smaller, more specific interfaces
class Printer:
    def print(self):
        pass

class Scanner:
    def scan(self):
        pass

class Fax:
    def fax(self):
        pass

class MultiFunctionPrinter(Printer, Scanner, Fax):
    pass  # Implements all functionalities as needed
```

5. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules; both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions.

```
# Violating DIP: High-level class depends on low-level class directly
class LightBulb:
    def turn_on(self):
        pass
    def turn_off(self):
        pass


class Switch:
    def __init__(self, bulb: LightBulb):
        self.bulb = bulb


    def operate(self, on: bool):
        if on:
            self.bulb.turn_on()
        else:
            self.bulb.turn_off()


# Following DIP: High-level class depends on an abstraction
class Switchable:
    def turn_on(self):
        pass


    def turn_off(self):
        pass


class LightBulb(Switchable):
    def turn_on(self):
        pass


    def turn_off(self):
        pass

class Switch:
    def __init__(self, device: Switchable):
        self.device = device

    def operate(self, on: bool):
        if on:
            self.device.turn_on()
        else:
            self.device.turn_off()
```