

A dark, grayscale background image showing a person's hands interacting with a smartphone. One hand is holding the phone, while the other is hovering over it, possibly gesturing or about to tap. The image is dimly lit, emphasizing the hands and the device.

SOLID Design Principle

Table of Contents

Points to discuss:

- Overview of SOLID
- What does "S" stand for and its meaning
- What does "O" stand for and its meaning
- What does "L" stand for and its meaning
- What does "I" stand for and its meaning
- What does "D" stand for and its meaning

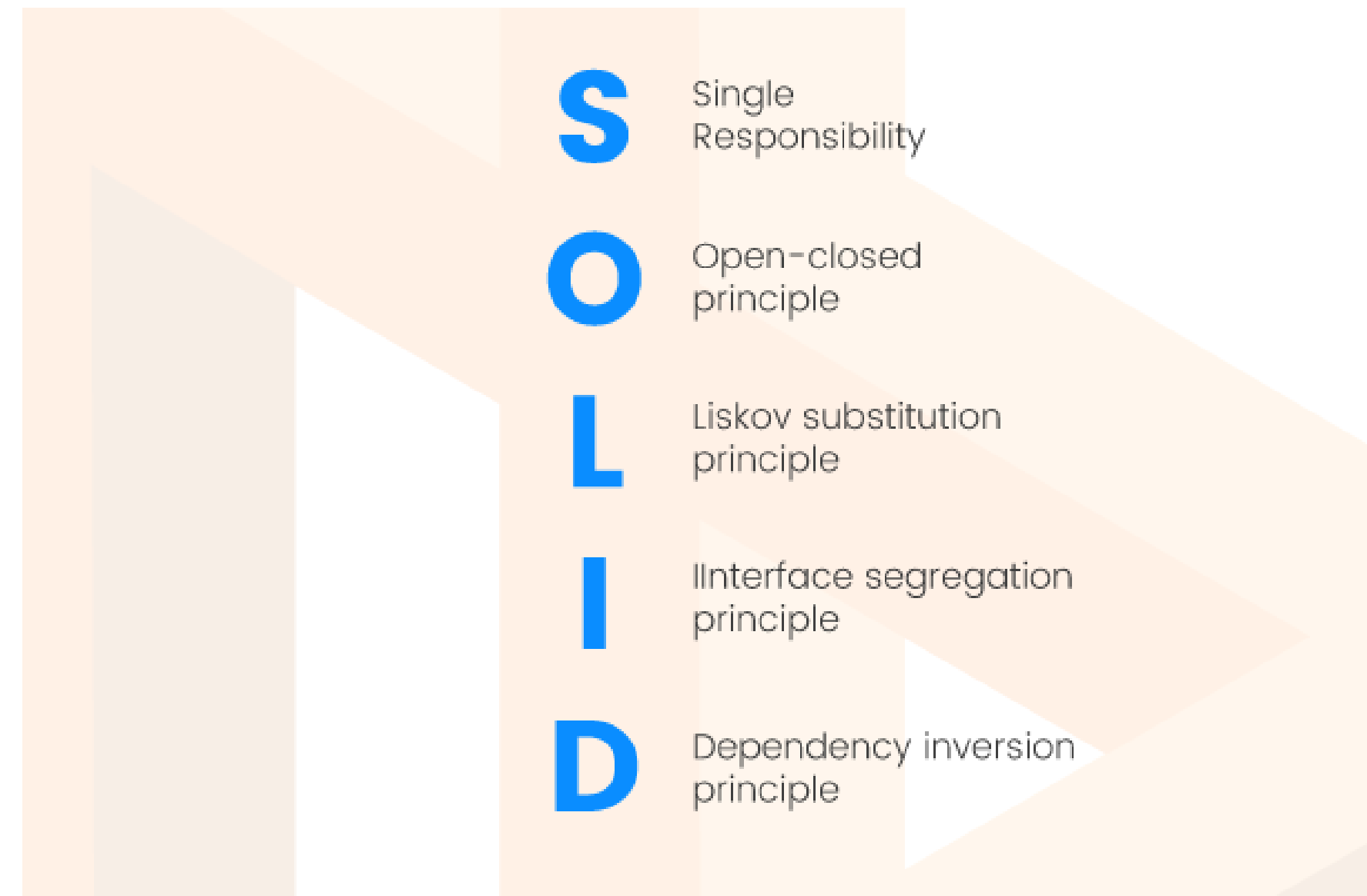
About **SOLID**

First introduced by the famous Computer Scientist Rober J Martin in 2000 .

But the **SOLID** acronym was released later by Michael Feathers.

SOLID refers to 5 design principle of OOP which has been summed up by experienced developers.

It all aims to better understanding, testing and especially maintaining.



1. The Single Responsibility Principle

What is it ?

It states that a class should be responsible just for one job and therefore, it has only a single reason to change.

ValidatePerson.java > ValidatePerson > ValidateAge()

```
1  public class ValidatePerson {
2      public String name;
3      public int age;
4      // Constructor
5      public ValidatePerson(String name, int age) {
6          this.name = name;
7          this.age = age;
8      }
9      // Validate name
10     public boolean ValidateName() {
11         if (name.length() > 3) {
12             return true;
13         } else {
14             return false;
15         }
16     }
17
18     // Validate age
19     public boolean ValidateAge() {
20         if (age > 18) {
21             return true;
22         } else {
23             return false;
24         }
25     }
26
27     // Display information
28     public void display(){
29         if (ValidateAge() && ValidateName()){
30             System.out.println("Name: " + name);
31             System.out.println("Age:" + age);
32         } else {
33             System.out.println("Invalid");
34         }
35     }
36 }
```

This goes against S rule

How we fix?

ValidatePerson.java > ValidatePerson > ValidateName()

```
1 public class ValidatePerson {
2     protected String name;
3     protected int age;
4
5     // Constructor
6     public ValidatePerson(String name, int age) {
7         this.name = name;
8         this.age = age;
9     }
10
11    // Validate name
12    protected boolean ValidateName() {
13        if (name.length() > 3) {
14            return true;
15        } else {
16            return false;
17        }
18    }
19
20    // Validate age
21    protected boolean ValidateAge() {
22        if (age > 18) {
23            return true;
24        } else {
25            return false;
26        }
27    }
28 }
```

and

DisplayPerson.java > DisplayPerson > display()

```
1 public class DisplayPerson extends ValidatePerson {
2
3     // Constructor
4     public DisplayPerson(ValidatePerson p) {
5         super(p.name, p.age);
6     }
7
8     // Display information
9     public void display() {
10        if (ValidateAge() && ValidateName()) {
11            System.out.println("Name: " + name);
12            System.out.println("Age:" + age);
13        }
14    }
15 }
16
```

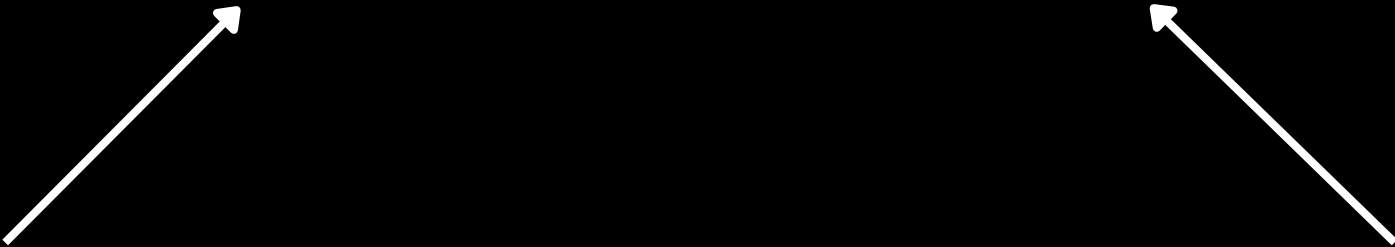
2. Open - Closed Principle

What is it ?

It states that classes can be opened for extension but should be closed to modification.

Example:

```
1  public abstract class Animal {  
2      private String name;  
3      private String type;  
4  
5      public Animal(String name, String type) {  
6          this.name = name;  
7          this.type = type;  
8      }  
9  
10     public void makeSound();  
11 }  
12
```

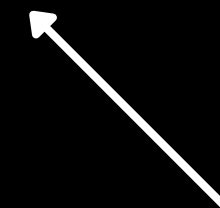
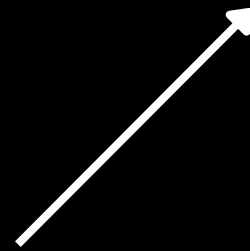


```
12  
13 ✓ public class Dog extends Animal {  
14 ✓     public Dog(String name) {  
15         super(name, "Dog");  
16     }  
17  
18 ✓     public void makeSound() {  
19         System.out.println("Bark");  
20     }  
21 }  
22
```

```
23 public class Cat extends Animal {  
24     public Cat(String name) {  
25         super(name, "Cat");  
26     }  
27  
28     public void makeSound() {  
29         System.out.println("Meow");  
30     }  
31 }
```



```
1  public abstract class Animal {
2      private String name;
3      private String type;
4
5      public Animal(String name, String type) {
6          this.name = name;
7          this.type = type;
8      }
9
10     public void makeSound();
11 }
12
```



```
12
13 ✓ public class Dog extends Animal {
14 ✓     public Dog(String name) {
15         super(name, "Dog");
16     }
17
18 ✓     public void makeSound() {
19         System.out.println("Bark");
20     }
21 }
22
```

```
23 public class Cat extends Animal {
24     public Cat(String name) {
25         super(name, "Cat");
26     }
27
28     public void makeSound() {
29         System.out.println("Meow");
30     }
31 }
```

```
32
33 public class Bird extends Animal {
34     public Bird(String name) {
35         super(name, "Bird");
36     }
37
38     public void makeSound() {
39         System.out.println("Tweet");
40     }
41 }
```

We can still extend without modifying parent class

Another example:

```
public class Employee {  
    private String name;  
    private double salary;  
    private String type;  
    public Employee(String name, double salary, String type) {  
        this.name = name;  
        this.salary = salary;  
        this.type = type;  
    }  
  
    public double getSalary() {  
        if (type.equals("full-time")) {  
            return salary;  
        } else if (type.equals("part-time")) {  
            return salary / 2;  
        } else {  
            return 0;  
        }  
    }  
}
```

What happens if we extend the Employee entity?

Solving method:

```
1  public abstract class Employee {  
2      private String name;  
3      private double salary;  
4  
5      public Employee(String name, double salary, String type) {  
6          this.name = name;  
7          this.salary = salary;  
8          this.type = type;  
9      }  
10  
11     public double getSalary();  
12 }  
13
```

```
18  
19 public class Accountant extends Employee {  
20  
21     // Override the getSalary method  
22 }  
23
```

```
13  
14 public class Manager extends Employee {  
15  
16     // Override the getSalary method  
17 }  
18
```

3. Liskov Substitution Principle

What is it ?

It states that subclasses should be substitutable for their parent classes.

Is it true?

```
1  public class Bird {
2      protected String color;
3
4      // Constructor
5      // Getter and Setter
6
7      public void fly(){
8          System.out.println("I'm flying");
9      }
10 }
11
12 public class Eagle extends Bird {
13     // Code implement here
14 }
15
16 public class Penguin extends Bird {
17     // Code implement here
18 }
19
20 public class Test {
21     public static void main(String[] args){
22         Bird b1 = new Eagle();
23         b1.fly();
24
25         Bird b2 = new Penguin();
26         b2.fly() // Penguin can not fly
27     }
28 }
```

Solving method

```
1 public class Bird {  
2     protected String color;  
3     // Constructor  
4     // Getter and Setter  
5  
6     public void fly();  
7 }
```

```
public class FlyingBird extends Bird {  
    // Code implement here  
    public void fly(){  
        System.out.println("I'm flying");  
    }  
}
```

```
public class NonFlyingBird extends Bird {  
    public void fly(){  
        System.out.println("I can not fly");  
    }  
}
```

```
public class Eagle extends FlyingBird {  
    // Code implement here  
}
```

```
public class Penguin extends NonFlyingBird {  
    // Code implement here  
}
```

Test again:

```
public class Test {  
    public static void main(String[] args){  
        Bird b1 = new Eagle();  
        b1.fly(); // Print I'm flying  
  
        Bird b2 = new Penguin();  
        b2.fly() // Print I can not fly  
    }  
}
```

4. Interface segregation principle

What is it ?

It states that instead of initializing a super big interface class with hundreds of methods, we should separate it into smaller one with specific purpose.

Bad

```
public interface IEmployee {  
    double calculateSalary();  
    void assignProject(Project project);  
    void submitTimesheet(Timesheet timesheet);  
}
```

Good

```
public interface ISalaryCalculator {  
    double calculateSalary();  
}  
  
public interface IProjectAssignee {  
    void assignProject(Project project);  
}  
  
public interface ITimesheetSubmitter {  
    void submitTimesheet(Timesheet timesheet);  
}
```

```
public class SalaryEmployee implements ISalaryCalculator {  
    public double calculateSalary() {  
        // tính lương cho nhân viên  
        return 500.0;  
    }  
}
```


5. Dependency Inversion Principle

What is it ?

It states that higher-level module should not depend on lower-level module, alternatively both of them should rely on abstraction (interface class).

Example:


```
public class Circle {  
    private double radius;  
  
    public Circle(double radius){  
        this.radius = radius;  
    }  
  
    public double getArea(){  
        return Math.PI * radius * radius;  
    }  
}
```

```
25  
26 ✓ public class Test {  
27 ✓     public static void main(String[] args){  
28         Circle c = new Circle(5.5);  
29         ShapeManager shape = new ShapeManager(c);  
30         System.out.println(shape.calculateArea());  
31     }  
32 }
```

```
public class ShapeManager {  
  
    private Circle circle;  
  
    public ShapeManager(Circle circle){  
        this.circle = circle;  
    }  
  
    public double calculateArea(){  
        return circle.getArea;  
    }  
}
```

**What happens if we extend
the ShapeManager to
calculate more geometries?**

```
1 public interface Shape {
2     double getArea();
3 }
4
```



The diagram consists of two white arrows. One arrow originates from the 'Shape' interface box and points to the 'Shape shape;' line in the ShapeManager class box. The other arrow originates from the 'ShapeManager' class box and points to the 'ShapeManager' class box itself, indicating a self-dependency or a dependency on the class.

```
5 public class Circle implements Shape {
6     private double radius;
7
8     //Constructor
9
10    public double getArea(){
11        return Math.PI * radius * radius;
12    }
13 }
14
```

```
public class Rectangle implements Shape {
    private double length, width;

    // Constructor

    public double getArea(){
        return length * width;
    }
}
```

```
25 public class ShapeManager{
26     private Shape shape;
27
28     public ShapeManager(Shape shape){
29         this.shape = shape;
30     }
31
32     public double getArea(){
33         return shape.getArea();
34     }
35 }
```

Here, we see that ShapeManager no longer depend on lower level module, it now working with interface class

In conclusion, we should keep this set of rules in mind when designing, programming so that it can be more readable, extendable and maintainable





Thanks for listenning