S.O.L.I.D. Principles: Simplified Explanation & Example

What S.O.L.I.D. Principles is?

S.O.L.I.D. Principles is a Software Development Principle for programming with OOP Paradigm. These Principles were introduced by Robert C. Martin (Uncle Bob) in his paper Design Principles and Design Patterns (2000)
Yes, you're not reading it wrongly nor I have a typo. It was written in 2000, 20 years ago since this article was published. But these Principles are still quite popular and are still being used widely in the world of OOP Paradigm.

Why S.O.L.I.D. Principles?

Because it makes your software :

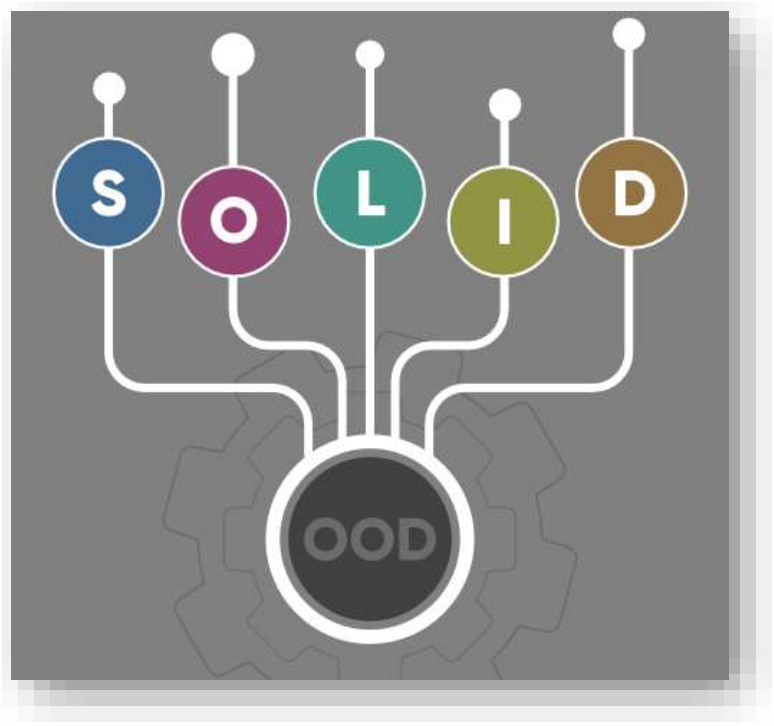
- More Understandable: When you come back to your code six months later, you still understand what you wrote back then.
- More Flexible: It's quite easy when you need to add features in your code because all codes are loosely coupled.
- More Maintainable: With the help of points 1 & 2, you will be easier in maintaining your code.

Robert C. Martin describes it as:

> *A class should have one, and only one, reason to change.*

This principle is an acronym of the five principles which is given below…

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle
3. Liskov's Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)



1. Liskov's Substitution Principle:

Liskov Substitution Principle states:

> Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of

- The principle was introduced by Barbara Liskov in 1987 and according to this principle "*Derived or child classes must be substitutable for their base or parent classes*". This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.
- You can understand it in a way that a farmer's son should inherit farming skills from his father and should be able to replace his father if needed. If the son wants to become a farmer then he can replace his father but if he wants to become a cricketer then definitely the son can't replace his father even though they both belong to the same family hierarchy.
- One of the classic examples of this principle is a rectangle having four sides. A rectangle's height can be any value and width can be any value. A square is a rectangle with equal width and height. So we can say that we can extend the properties of the rectangle class into square class. In order to do that you need to swap the child (square) class with parent (rectangle) class to fit the definition of a square having four equal sides but a derived class does not affect the behavior of the parent class so if you will do that it will violate the Liskov Substitution Principle.

EXAMPLE:

```
package com.codemate.solid;

import java.util.ArrayList;
import java.util.List;

public class Vehicle {

    void startEngine() {

    }

}

class Car extends Vehicle {

    @Override
    public void startEngine() {

    }
}


class Bycle extends Vehicle {
```

```
    @Override
    public void startEngine() {
        try {
            throw new EngineNotFoundExceptin("Engine Missing.");
        } catch (EngineNotFoundExceptin e) {
            e.printStackTrace();
        }
    }

    public class EngineNotFoundExceptin extends Throwable {
        EngineNotFoundExceptin(final String s) {
        }
    }
}

class VehicleMonitor {

    public void startVehiclees() {

        List<Vehicle> vehicles = new ArrayList<Vehicle>();
        Vehicle car = new Car();
        Vehicle cycle = new Bycle();

        vehicles.add(car);
        vehicles.add(cycle);

        vehicles.forEach(vehicle -> vehicle.startEngine());
    }
}
```

That satisfies the Liskov substitution principle.

2. **Interface Segregation Principle**

Interface segregation principle states:

> A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.

- This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that "*do not force any client to implement an interface which is irrelevant to them*". Here your

main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

- Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets. In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don't eat in your food. Here the menu should be different for different types of customers. The common or general menu card for everyone can be divided into multiple cards instead of just one. Using this principle helps in reducing the side effects and frequency of required changes.

EXAMPLE

```
package com.codemate.solid;

interface TwoDimensionalShape {
```

```
    double calculateArea();
}
interface ThreeDimensionalShape {
    double calculateVolume();
}
class Cuboid implements TwoDimensionalShape, ThreeDimensionalShape {
    @Override
    public double calculateArea() {
        // TODO Auto-generated method Stub
        return 0;
    }
    @Override
    public double calculateVolume() {
        // TODO Auto-generated method Stub
        return 0;
    }
}
class Square implements TwoDimensionalShape {
    @Override
    public double calculateArea() {
        // TODO Auto-generated method Stub
        return 0;
    }
```

```
}
```

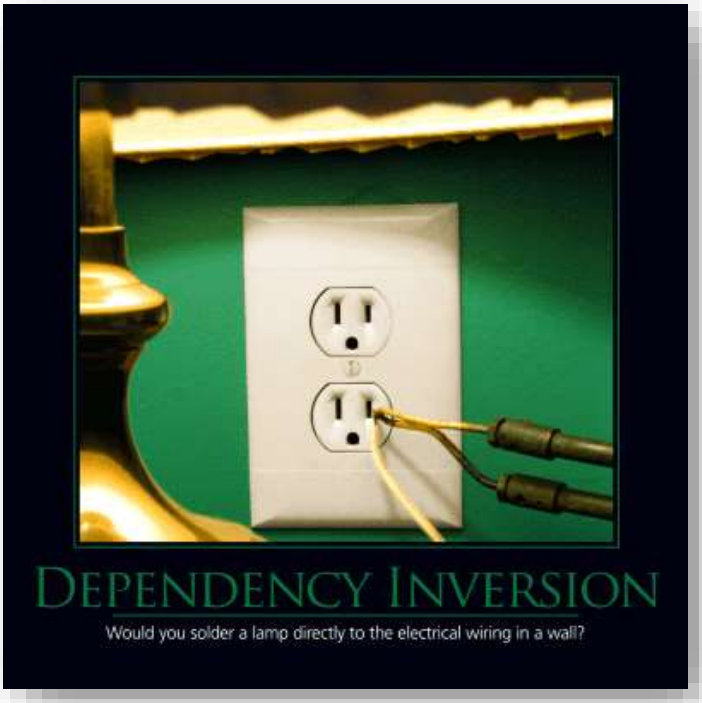
That satisfies the interface segregation principle.

3. **Dependency Inversion Principle**

Dependency inversion principle states:

> Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

- The Dependency Inversion Principle (DIP) states that high level modules should not depend on low level modules; both should depend on abstractions.
- Abstractions should not depend on details. Details should depend upon abstractions. It's extremely common when writing software to implement it such that each module or method specifically refers to its collaborators, which does the same.
- This type of programming typically lacks sufficient layers of abstraction, and results in a very tightly coupled system, since every module is directly referencing lower level modules.
- The above lines simply state that if a high module or class will be dependent more on low-level modules or class then your code would have tight coupling and if you will try to make a change in one class it can break another class which is risky at the production level. So always try to make classes loosely coupled as much as you can and you can achieve this through *abstraction*. The main motive of this principle is decoupling the dependencies so if class A changes the class B doesn’t need to care or know about the changes.
- You can consider the real-life example of a TV remote battery. Your remote needs a battery but it’s not dependent on the battery brand. You can use any XYZ brand that you want and it will work. So we can say that the TV remote is loosely coupled with the brand name. Dependency Inversion makes your code more reusable.



EXAMPLE

```
package com.codemate.solid;

public class Desktop {

    private Monitor monitor;
    private Keyboard keyboard;

    public Desktop(Keyboard keyboard, Monitor monitor) {
        this.monitor = monitor;
        this.keyboard = keyboard;
    }
}

interface Keyboard {
}
class Monitor {
}
class QwertyKeyBoard implements Keyboard {
}
```