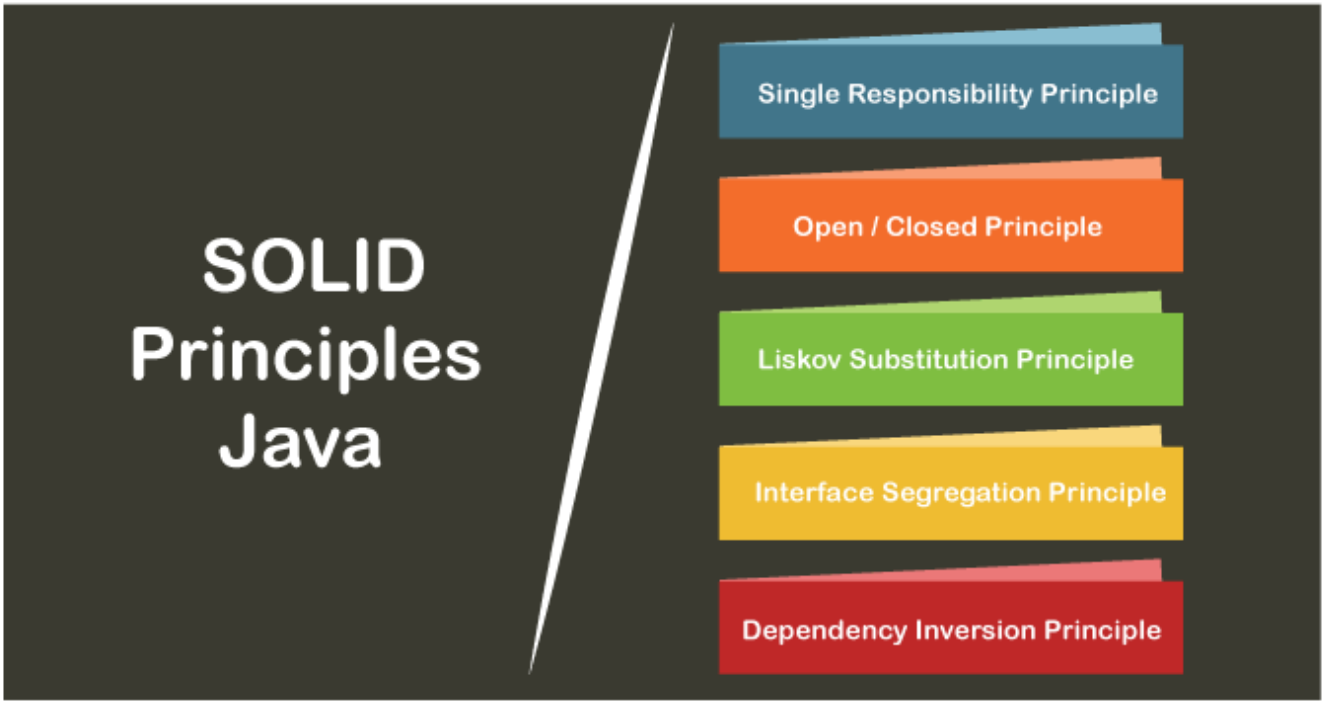
S.O.L.I.D. Principles: Using Functional Programming



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:

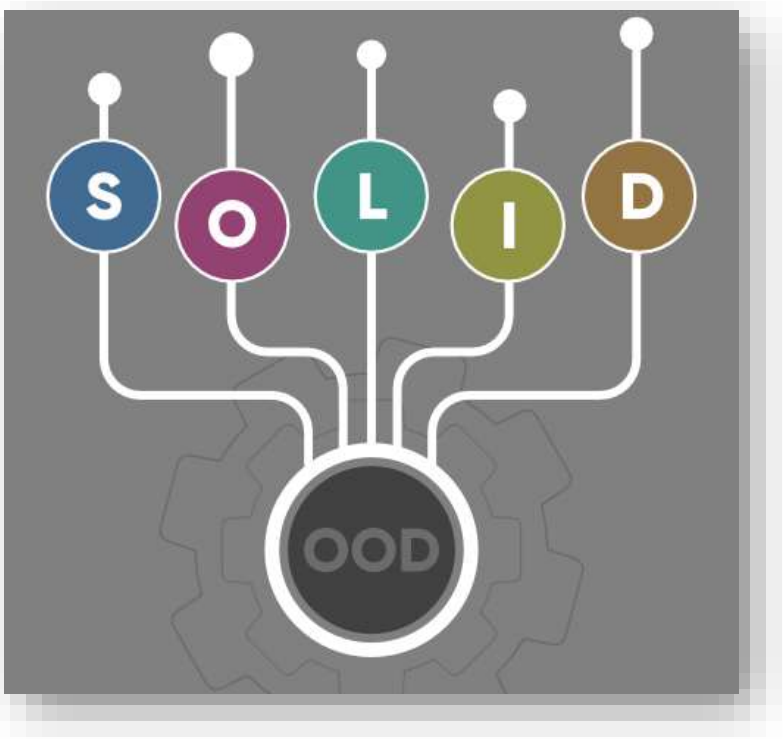
A class should have one, and only one, reason to change.

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)

Principle	Description
Single Responsibility Principle	Each class should be responsible for a single part or functionality of the system.
Open-Closed Principle	Software components should be open for extension, but not for modification.
Liskov Substitution Principle	Objects of a superclass should be replaceable with objects of its subclasses without breaking the system.
Interface Segregation Principle	No client should be forced to depend on methods that it does not use.

Principle	Description
Dependency Inversion Principle	High-level modules should not depend on low-level modules, both should depend on abstractions.



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

- This principle states that “Derived or child classes must be substitutable for their base or parent classes”. It was introduced by Barbara Liskov in 1987 and it also ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behaviour.

- When a child Class cannot perform the same actions as its parent Class, this can cause bugs.

- If you have a Class and create another Class from it, it becomes a parent and the new Class becomes a child. The child Class should be able to do everything the parent Class can do. This process is called Inheritance.

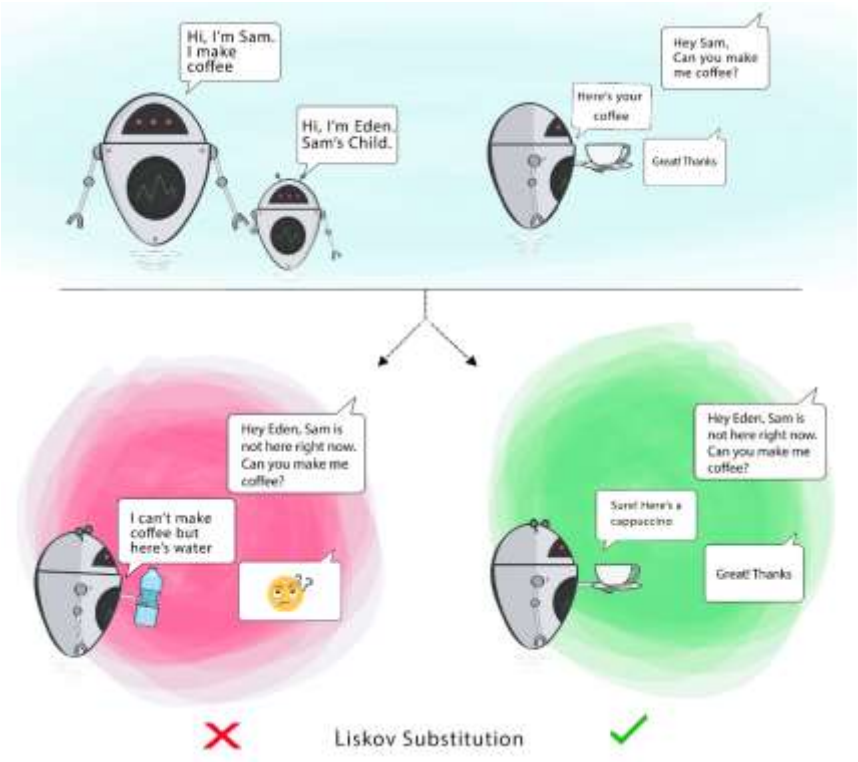
- The child Class should be able to process the same requests and deliver the same result as the parent Class or it could deliver a result that is of the same type.

- The picture shows that the parent Class delivers Coffee(it could be any type of coffee). It is acceptable for the child Class to deliver Cappuccino because it is a specific type of Coffee, but it is NOT acceptable to deliver Water.

- If the child Class doesn’t meet these requirements, it means the child Class is changed completely and violates this principle.

- LSP also applies in case we use generic or parametric programming where we create functions that work on a variety of types; they all hold a common truth that makes them interchangeable.

- This pattern is super common in functional programming, where you create functions that embrace polymorphic types (aka generics) to ensure that one set of inputs can seamlessly be substituted for another without any changes to the underlying code



EXAMPLE:

```
class Rectangle {
  private var length = 0
  private var breadth = 0

  def getLength: Int = length

  def setLength(length: Int): Unit = {
    this.length = length
  }

  def getBreadth: Int = breadth

  def setBreadth(breadth: Int): Unit = {
    this.breadth = breadth
  }

  def getArea: Int = this.length * this.breadth
}

class Square extends Rectangle {
  override def setBreadth(breadth: Int): Unit = {
    super.setBreadth(breadth)
```

```
    super.setLength(breadth)
  }

  override def setLength(length: Int): Unit = {
    super.setLength(length)
    super.setBreadth(length)
  }
}
object Liskov extends App{
  val Shapes = new Square
  Shapes.setLength(22)
  Shapes.setBreadth(24)
  println(Shapes.getArea)

}
```

That satisfies the Liskov substitution principle.

2. **Interface Segregation Principle**

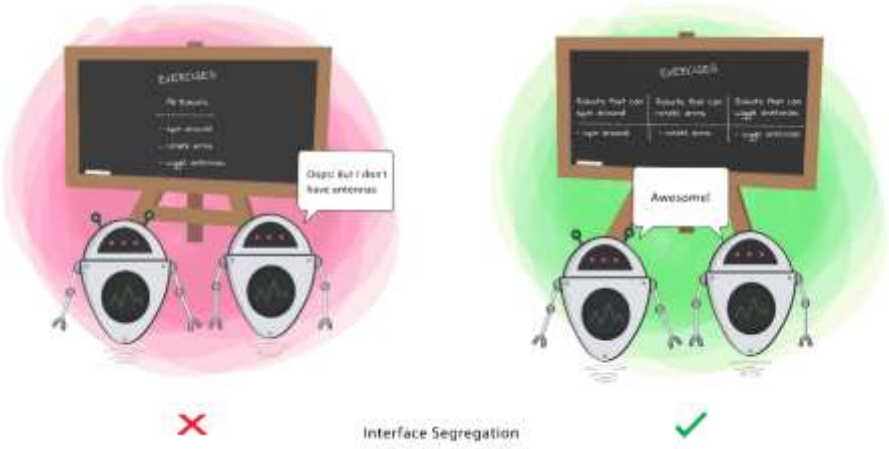
Interface segregation principle states:

> 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.

- In the field of software engineering, the interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.[1] ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called *role interface*s.[2] ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five SOLID principles of object-oriented design, similar to the High Cohesion Principle of GRASP

- The ISP was first used and formulated by Robert C. Martin while consulting for Xerox. Xerox had created a new printer system that could perform a variety of tasks such as stapling and faxing. The software for this system was created from the ground up. As the software grew, making modifications became more and more difficult so that even the smallest change would take a redeployment cycle of an hour, which made development nearly impossible.

- The design problem was that a single Job class was used by almost all of the tasks. Whenever a print job or a stapling job needed to be performed, a call was made to the Job class. This resulted in a 'fat' class with multitudes of methods specific to a variety of different clients. Because of this design, a staple job would know about all the methods of the print job, even though there was no use for them.
- The solution suggested by Martin utilized what is today called the Interface Segregation Principle. Applied to the Xerox software, an interface layer between the Job class and its clients was added using the Dependency Inversion Principle. Instead of having one large Job class, a Staple Job interface or a Print Job interface was created that would be used by the Staple or Print classes, respectively, calling methods of the Job class. Therefore, one interface was created for each job type, which was all implemented by the Job class.



EXAMPLE

```
trait Athlete{

  def highJump(): Unit={}

}

trait Shooting{

  def fiftyMeter():Unit={}

}

class Tom extends Athlete{

  override def highJump():Unit = println("Tom won the gold medal in HighJump")
```

```
}

class Binod extends Shooting{

  override def fiftyMeter(): Unit = println("Binod won the gold medal in Shooting")

}

object Interface extends App{

  val Data1 = new Tom

  Data1.highJump()

  val Data2 = new Binod

  Data2.fiftyMeter()

}
```

That satisfies the interface segregation principle.

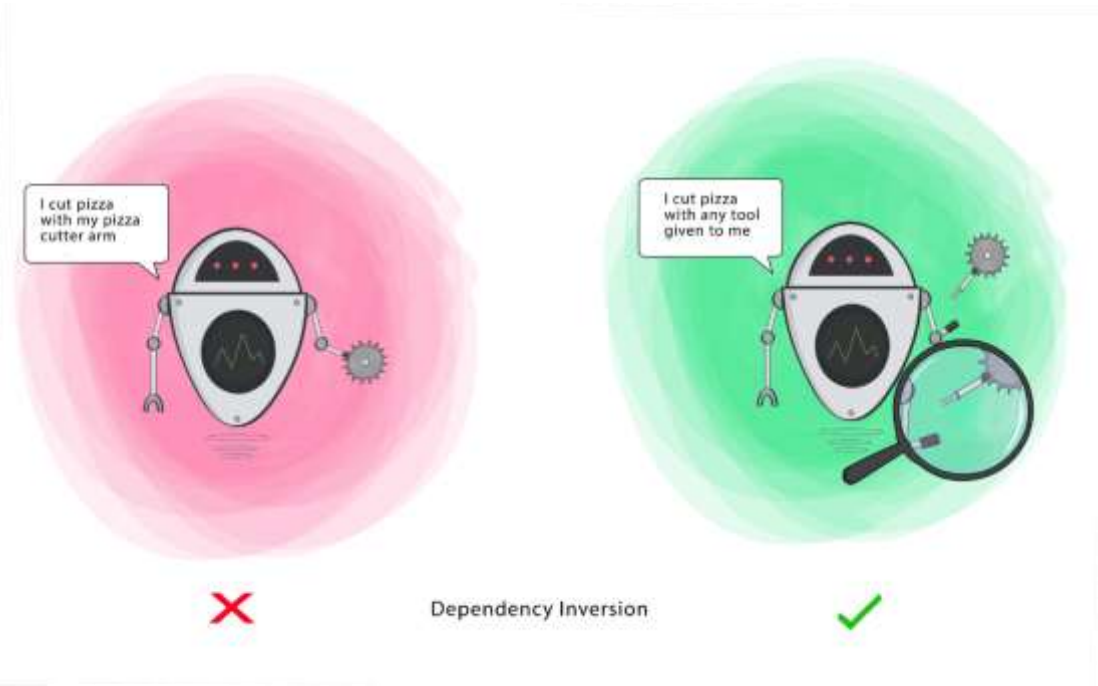
3. **Dependency Inversion Principle**

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.

- Dependency Inversion in OOP means that you code against an interface which is then provided by an implementation in an object.
- Languages that support higher language functions can often solve simple dependency inversion problems by passing behaviour as a function instead of an object which implements an interface in the OO-sense.
- In such languages, the function's signature can become the interface and a function is passed in instead of a traditional object to provide the desired behaviour. The hole in the middle pattern is a good example for this.
- It let's you achieve the same result with less code and more expressiveness, as you don't need to implement a whole class that conforms to an (OOP) interface to provide the desired behaviour for the caller. Instead, you can just pass a simple function

definition. In short: Code is often easier to maintain, more expressive and more flexible when one uses higher order functions.



EXAMPLE

```
object Dependency extends App {
  trait Developer{
    def develop: Any = {}
  }
  class BackEndDeveloper extends Developer{
    override def develop: Any = writeJava()
    def writeJava(): Any = {println("Backend is coded in JAVA")}
  }
  class FrontEndDeveloper extends Developer{
    override def develop: Any = writeJavaScript()
    def writeJavaScript(): Any = {println("Frontend is coded in JS")}
  }
  class Project(var developers: List[Dependency.Developer]) {
    def implement(): Unit = {
      developers.foreach((d: Dependency.Developer) => d.develop)
    }
  }
  val back = new BackEndDeveloper
  val front = new FrontEndDeveloper
  val project = new Project(developers = List(back, front))
  project.implement
```

```
}
```

Summary:

At the end, I have discussed SOLID five Principles using Functional Programming with their uses and definition. This is really helpful for doing effectively coding, and test with all circumstances. I am able to understand the concept of Function programming using SOLID Principal