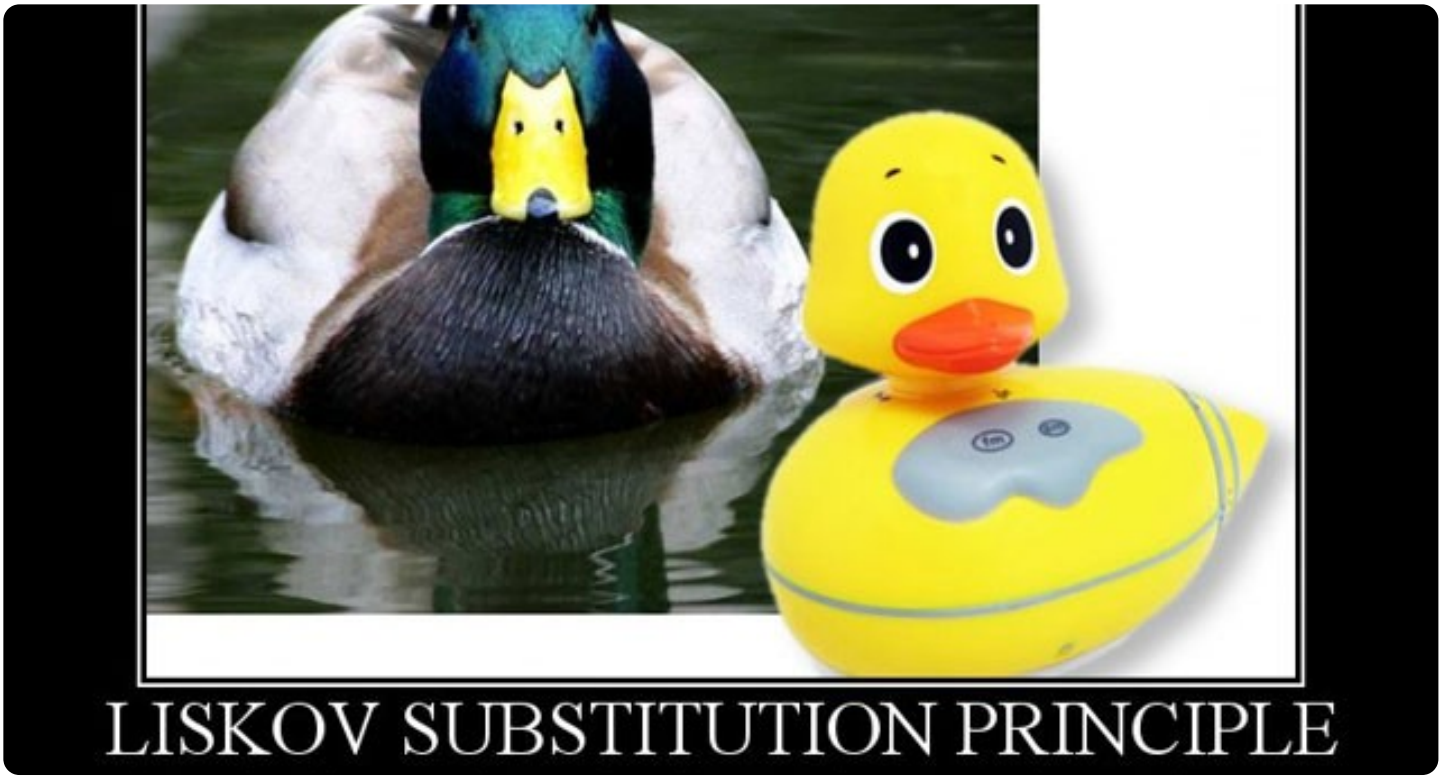

The Liskov Substitution Principle (LSP)

Paul Gichure, CTFL
Solution Architect | Software Engineer | Systems Integration
Data pubblicazione: 10 apr 2020

Segui

So far we have explained the **Single Responsibility Principle(SRP)** and **Open-Closed Principle (OCP)** in our series of articles on the S.O.L.I.D principles. There are key concepts in Object Oriented Programming(OOP) that enables you to write robust, maintainable and reusable software components - classes, modules, functions, e.t.c.

In this article, we will explain the **Liskov Substitution Principle** which focuses on the behavior of a superclass and its subtypes.

The Liskov Substitution Principle (LSP): "*functions that use pointers to base*

Thus, the objects of the subclasses should behave in the same way as the objects of the superclass.

This principle ensures that inheritance (one of the OOP principles) is used correctly. If an override method does nothing or just throws an exception, then you're probably

violating the LSP. The classic example of the inheritance technique causing problems is the **circle-elipse problem** (a.k.a the rectangle-square problem) which is a is a violation of the Liskov substitution principle.

A good example here is that of a bird and a penguin; I will call this *dove-penguin problem*. The below is a Java code snippet showing an example that violates the LSP principle.

```
public class Bird {
  public void fly() {
    System.out.println("I'm flying");
  }
public void walk() {
   System.out.println("I'm walking");
 }
 public class Dove extends Bird{
 }
}
```

Here, the Dove can fly because it is a Bird.

```
public class Penguin extends Bird{
}
```

In this inheritance, much as technically a penguin is a bird, penguins do not fly. Thus, the "fly" method is not applicable to all types of birds; violating the LSP principle.

To correct this, the developer would be forced modify the code to accomodate the instance of a Penguin (this means violating the OCP principle or adhere to LSP principle as below;

```
public class Bird {
```

```
 public void walk() {
    System.out.println("I'm walking");
 }
public class FlyingBird extends Bird{
 public void fly() {
    System.out.println("I'm flying");
 }
}
public class Dove extends FlyingBird{}
public class Penguin extends Bird{ }
}
```

In this snippet, the Dove can both walk(from the Bird class) and fly (from FlyingBird class) and the Penguin can only walk (from the Bird class).

You can get all source files of this example at **https://github.com/gichure/solid-principles.**

*PS. For updates about new posts, sites I find useful and the occasional insights you can **follow me on Twitter**.*