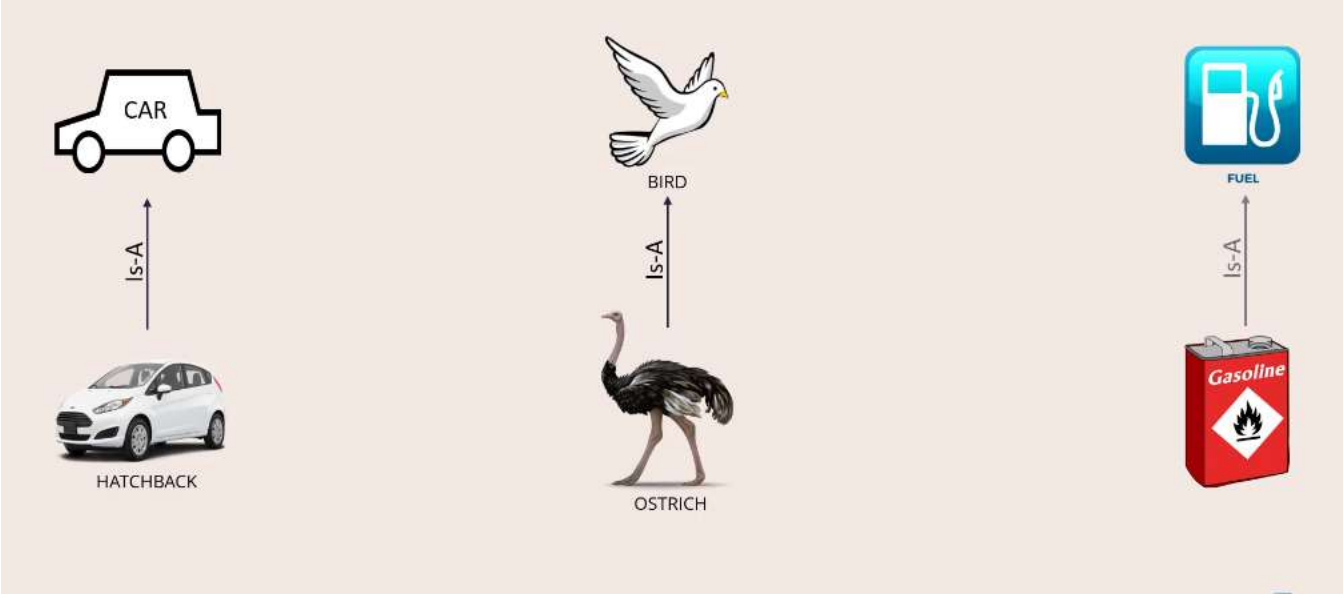
KD Knowledge Diet Follow
Apr 1 · 4 min read · Listen

Liskov Substitution Principle: Top Developer's technique which improves 2.5x the quality of your code

As you write code, doesn't anyone ever question whether your code is logically correct or consistent enough? To solve such problems, it is necessary to study the software principle. Today, I will introduce one of the SOLID principles, the **Liskov Substitution Principle.** The Liskov Principle is named after a computer scientist. It's often called **LSP**.

> ***Objects should be replaceable with their subtypes without affecting the correctness of the program.***

I know. I know. It's very abstract. Let's take an example.



Category and its subcategory

If I define it with code, it would be like this.

```
class Car {
}
class Hatchback extends Car {
}
class Bird {
}
class Ostrich extends Bird {
}
class Fuel {
}
```

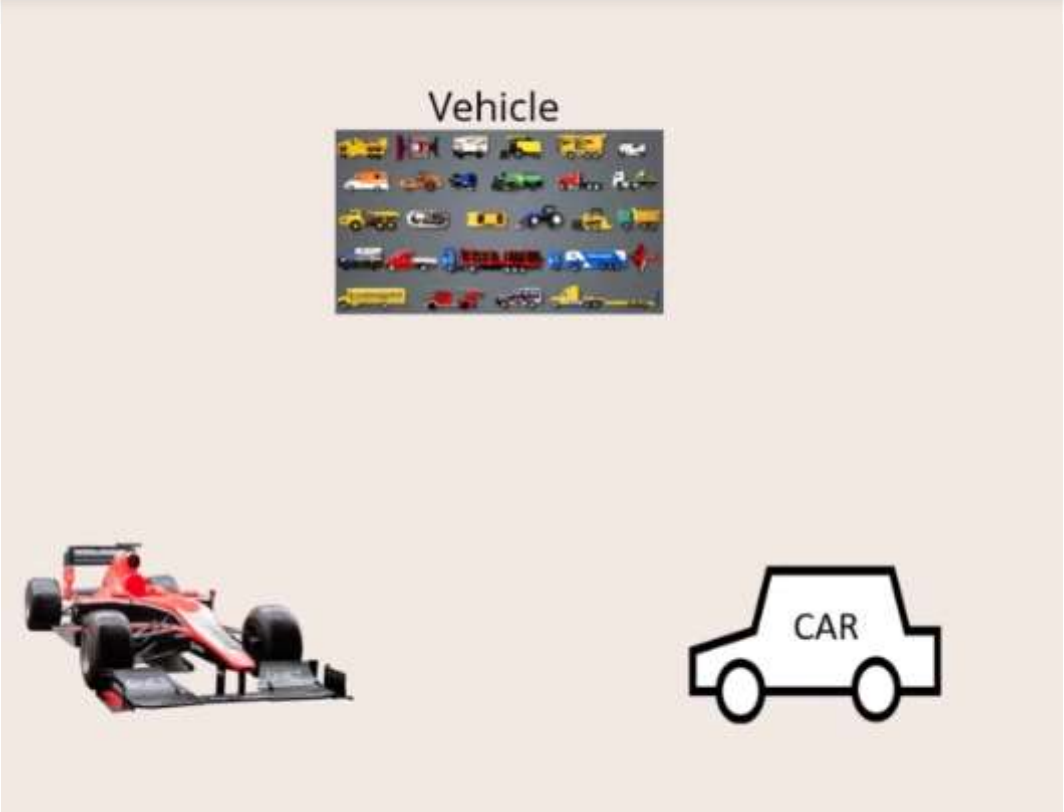
According to LSP, Car can be replaced with Hatchback. As such, bird is replaceable with Ostrich and Fuel with Gasoline. It seems perfect till now.

[1] Violation of LSP Example 1, Fix by Breaking The Hierarchy

> *Break the hierarchy if it fails the substitution test.*

(1) Do all cars have cabin?

```
public class Car {

     public double getCabinWidth() {
          // return cabin width
     }

}

public class RacingCar extends Car {

     @Override
     public double getCabinWidth() {
          //UNIMPLEMENTED!
     }

     public double getCockpitWidth() {
          // return cockpit width
     }

}
```

Racing Car doesn't have cabin!

Racing Car doesn't have a cabin. It can be said that such code is designed incorrectly in software. Then, how do I fix it?

(2) Break the hierarchy

```
public class Vehicle {

     public double getInteriorWidth() {
        // return interior width
     }

}

public class Car extends Vehicle {

     @Override
     public double getInteriorWidth() {
         return this.getCabinWidth();
     }

     public double getCabinWidth() {
        // return cabin width
```

```
        @Override
        public double getCabinWidth() {
             return this.getCockpitWidth();
        }

        public double getCockpitWidth() {
            // return cockpit width
        }

    }
```

Instead of inheriting Car, I created another super class `Vehicle` . With this code, you can say that LSP principle is kept and therefore, better designed.

[2] Violation of LSP Example 2

> *Tell, Don't ask*

(1) Bad Practice. Casting

```
public class Product {

        protected double discount = 20;

        public double getDiscount() {
            return discount;
        }

}

public class InHouseProduct extends Product {

        public void applyExtraDiscount() {
            discount = discount * 1.5; // multiplies 1.5 times
        }

}

public class PricingUtils {
```


```
            Product p3 = new InHouseProduct();

            List<Product> productList = new ArrayList<>();

            productList.add(p1);
            productList.add(p2);
            productList.add(p3);

           for(Product product : productList) {
               if (product instanceof InHouseProduct) {
                   // Type casting
                   // Not Correct Way
                   ((InHouseProdcut) product).applyExtraDiscount();
                }

                System.out.println(product.getDiscount());

            }
        }

    }
```

Can you tell the wrongness of this code?

```
if (product instanceof InHouseProduct) {
   // Type casting
   // Not Correct Way
   ((InHouseProdcut) product).applyExtraDiscount()
}
```

According to LSP, this code breaks the rule. Because "***Objects should be replaceable with their subtypes without affecting the correctness of the program.***"

The code is asking or enquiring about the subtype, from INSIDE the Utils class.

(2) Tell, Don't ask

```
public class Product {

     protected double discount = 20;
```


```
public class InHouseProduct extends Product {

     @Override
     public double getDiscount() {
          this.applyExtraDiscount();
          return discount;
     }

     public void applyExtraDiscount() {
           discount = discount * 1.5; // multiplies 1.5 times
     }

}

public class PricingUtils {

     public static void main(String[] args) {

          Product p1 = new Product();
          Product p2 = new Product();
          Product p3 = new InHouseProduct();

          List<Product> productList = new ArrayList<>();

          productList.add(p1);
          productList.add(p2);
          productList.add(p3);

          for(Product product : productList) {
               System.out.println(product.getDiscount());
          }

     }

}
```

Now, the subclass `InHouseProduct` overrides `getDiscount()` method. And I redefined its functionality.

As a result, It doesn't enquire about subtypes. PricingUtils can just loop through object without casting. This is what it is called "Tell, Don't ask".

You can sooner or later become a competent software engineer capable of writing logically well-defined and concise code.

This needs 'intentional practice'.

Other Articles for Solid Principles

Single Responsibility Principle: What? working as "a software engineer" you don't know it?!?!

Have you ever experienced that code refactoring doesn't make your code better? Adding spaghetti on top of spaghetti...

paigeshin1991.medium.com

Open Closed Principle: Make your code cost-free and flexible

Software is never dormant. It's constantly changing. The annoying fact about the software development is that every...

paigeshin1991.medium.com

Liskov Substitution Principle: Top Developer's technique which improves 2.5x

As you write code, doesn't anyone ever question whether your code is logically correct or consistent enough? To solve...

paigeshin1991.medium.com

Interface Segregation Principle: Your only way to be a super competent developer

During development, the code tends to be long. At some point, the codebase becomes so bloated that you can't handle it...

To become a high-paid developer, you need to learn TDD. Basically, you have to develop software with TDD to get into a...

paigeshin1991.medium.com