O Princípio L do SOLID

Liskov Substitution Principle - LSP

Introdução ao conceito L do SOLID

Imagine que você está brincando com diferentes tipos de carrinhos de brinquedo. Todos eles têm rodas, então você espera que eles andem quando você os empurra. Mas, se de repente, você empurra um carrinho que não anda ou se comporta de forma estranha, sua brincadeira pode ser arruinada. Isso acontece porque você tinha uma expectativa que não foi atendida.

O conceito L no SOLID é sobre garantir que as expectativas sejam sempre atendidas quando usamos algo. Ele é o Princípio da Substituição de Liskov, que diz:
"Se você tem uma classe base, pode substituir as suas subclasses sem quebrar o funcionamento do código."

Em outras palavras, as "subclasses" (como diferentes tipos de carrinhos) devem sempre funcionar da maneira que você espera, assim como a classe base (o carrinho básico).

Agora, vamos ver isso na prática com exemplos práticos em Java.

Sem aplicar o princípo LSP:

```java
public class Retangulo {
    protected int largura;
    protected int altura;

    public void setLargura(int largura) {
        this.largura = largura;
    }

    public void setAltura(int altura) {
        this.altura = altura;
    }

    public int getArea() {
        return largura * altura;
    }
}

public class Quadrado extends Retangulo {
    @Override
    public void setLargura(int largura) {
        this.largura = largura;
        this.altura = largura; // Para garantir que largura = altura
    }

    @Override
    public void setAltura(int altura) {
        this.altura = altura;
        this.largura = altura; // Para garantir que altura = largura
    }
}
```

Aqui, temos a classe Retangulo e uma subclasse Quadrado. O problema surge quando tentamos usar o quadrado em vez de um retângulo, pois ele modifica a lógica de altura e largura, o que pode gerar resultados inesperados.

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Retangulo retangulo = new Quadrado();
        retangulo.setLargura(largura:5);
        retangulo.setAltura(altura:10);

        System.out.println("Área: " + retangulo.getArea());
        // Deveria ser 50, mas retorna 100!
    }
}
```

Neste exemplo, o comportamento da área do quadrado quebra a expectativa, violando o Princípio da Substituição de Liskov.

Aplicando o princípo LSP:

```java
public abstract class Forma {
    public abstract int getArea();
}

public class Retangulo extends Forma {
    protected int largura;
    protected int altura;

    public Retangulo(int largura, int altura) {
        this.largura = largura;
        this.altura = altura;
    }

    @Override
    public int getArea() {
        return largura * altura;
    }
}

public class Quadrado extends Forma {
    private int lado;

    public Quadrado(int lado) {
        this.lado = lado;
    }

    @Override
    public int getArea() {
        return lado * lado;
    }
}
```

Agora, Quadrado e Retangulo são subclasses de Forma e cada uma implementa sua própria maneira de calcular a área. Quando usamos Quadrado ou Retangulo, eles atendem às expectativas sem confusão:

```java
public class Main {
    public static void main(String[] args) {
        Forma retangulo = new Retangulo(5, 10);
        Forma quadrado = new Quadrado(5);

        System.out.println("Área do Retângulo: " + retangulo.getArea()); // 50
        System.out.println("Área do Quadrado: " + quadrado.getArea());   // 25
    }
}
```

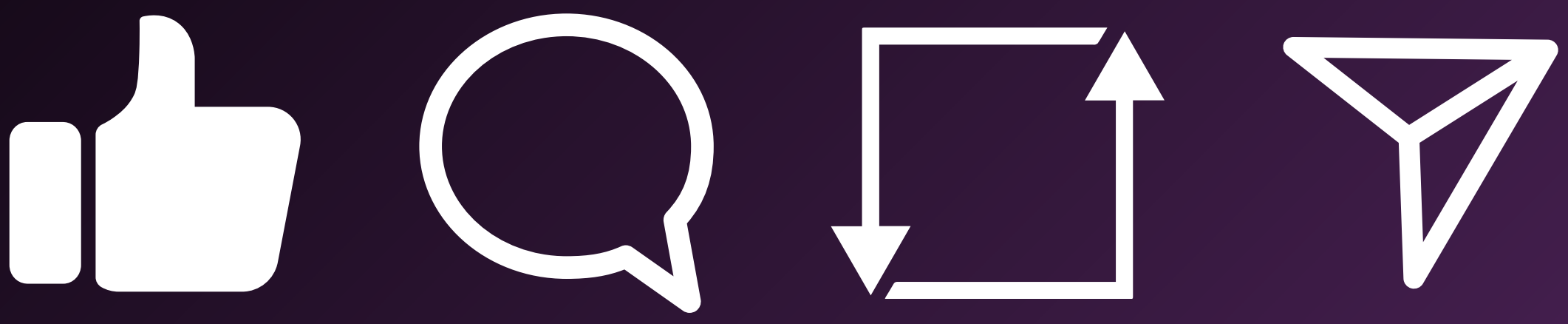
Agora, a substituição funciona corretamente, e as expectativas são atendidas sem surpresas.

CONCLUSÃO

Seguir o Princípio da Substituição de Liskov é como garantir que todos os carrinhos de brinquedo funcionem do jeito que você espera. No código, isso significa que as subclasses devem poder substituir suas classes base sem causar problemas inesperados. Ao aplicar esse conceito, você mantém seu código confiável e fácil de entender!

Ao aplicar esse conceito, você mantém seu código limpo, organizado e pronto para o futuro!

Compartilha com aquele que quer aprender.



curti **comenta** **compartilha** **envia**

Que a força do código esteja com você !