O Princípio O do SOLID

Open-Closed Principle - OCP

Introdução ao conceito O do SOLID

Imagine que você está construindo uma cidade de blocos de brinquedo. Primeiro, você cria casas e ruas, e tudo está ótimo. Mas, um dia, você quer adicionar mais coisas, como parques ou prédios. Se você tiver que desmontar tudo para adicionar essas novidades, vai dar muito trabalho, né?

Agora, se desde o começo você criar suas casas e ruas de uma forma que seja fácil colocar novos blocos sem mexer no que já está pronto, construir sua cidade vai ser muito mais divertido e organizado.

O conceito O do SOLID é sobre isso! O significa "Open/Closed Principle" ou Princípio Aberto/Fechado. Ele diz que o código deve estar aberto para extensão (podemos adicionar novas funcionalidades) e fechado para modificação (não precisamos mexer no que já está funcionando). Assim, quando novas necessidades surgem, podemos adicionar novas "peças" ao nosso código sem estragar o que já foi feito.

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

Sem aplicar o princípo OCP:

```java
public class PaymentProcessor {

    public void processPayment(String paymentType) {
        if (paymentType.equals(anObject:"creditCard")) {
            System.out.println(x:"Processando pagamento com cartão de crédito...");
        } else if (paymentType.equals(anObject:"paypal")) {
            System.out.println(x:"Processando pagamento com PayPal...");
        } else {
            System.out.println(x:"Método de pagamento não suportado.");
        }
    }
}
```

Aqui, se precisarmos adicionar um novo método de pagamento, como Pix, teremos que modificar a classe **PaymentProcessor**.

Isso vai contra o Princípio Aberto/Fechado, pois estamos sempre alterando o código existente.

Aplicando o princípo OCP:

```java
public interface PaymentMethod {
    void processPayment();
}

public class CreditCardPayment implements PaymentMethod {
    @Override
    public void processPayment() {
        System.out.println(x:"Processando pagamento com cartão de crédito...");
    }
}

public class PayPalPayment implements PaymentMethod {
    @Override
    public void processPayment() {
        System.out.println(x:"Processando pagamento com PayPal...");
    }
}

public class PaymentProcessor {
    public void processPayment(PaymentMethod paymentMethod) {
        paymentMethod.processPayment();
    }
}
```

Agora, se quisermos adicionar o método Pix, basta criar uma nova classe sem alterar o código existente:

Adicionando o método PIX

```java
public interface PaymentMethod {
    void processPayment();
}

public class CreditCardPayment implements PaymentMethod {
    @Override
    public void processPayment() {
        System.out.println(x:"Processando pagamento com cartão de crédito...");
    }
}

public class PayPalPayment implements PaymentMethod {
    @Override
    public void processPayment() {
        System.out.println(x:"Processando pagamento com PayPal...");
    }
}

public class PaymentProcessor {
    public void processPayment(PaymentMethod paymentMethod) {
        paymentMethod.processPayment();
    }
}

public class PixPayment implements PaymentMethod {
    @Override
    public void processPayment() {
        System.out.println(x:"Processando pagamento com Pix...");
    }
}
```

Com essa abordagem:

- O código está aberto para extensão — podemos adicionar novos métodos de pagamento criando novas classes.

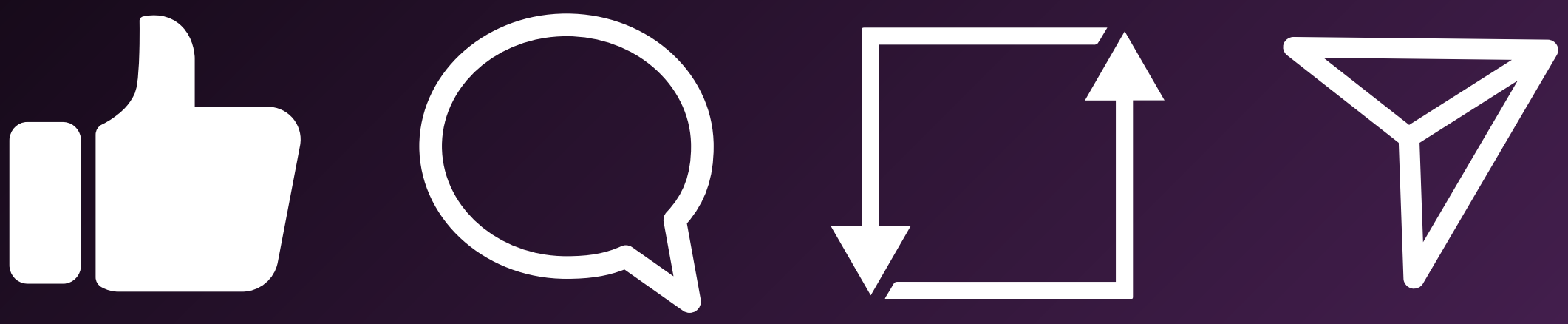
- O código está fechado para modificação — não precisamos mexer no PaymentProcessor sempre que uma nova forma de pagamento surgir.

CONCLUSÃO

Seguir o Princípio Aberto/Fechado é como construir uma cidade de blocos planejada desde o começo para ser expandida. Isso faz com que o código seja mais fácil de manter e adaptar a novas necessidades, sem quebrar o que já está funcionando.

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 !