O Princípio D do SOLID

Dependency Inversion Principle - DIP

Introdução ao conceito I do SOLID

Imagine que você tem um brinquedo que funciona com qualquer tipo de bateria: pequenas, grandes ou recarregáveis. Dessa forma, você não depende de uma bateria específica e pode sempre trocar por outra quando precisar. É muito prático, certo?

Esse é o espírito do conceito D do SOLID, conhecido como Princípio da Inversão de Dependência. Ele diz que o código não deve depender de detalhes específicos (como um tipo exato de bateria), mas sim de abstrações, ou seja, de "formas gerais" que podem ser substituídas conforme necessário. Assim, o código se torna mais flexível, fácil de testar e de manter.

Sem aplicar o princípo DIP:

Aqui temos um sistema onde uma classe PedidoService depende diretamente de EmailService para enviar notificações:

```java
public class EmailService {
    public void enviarEmail(String mensagem) {
        System.out.println("Enviando e-mail: " + mensagem);
    }
}

public class PedidoService {
    private EmailService emailService = new EmailService();

    public void processarPedido() {
        // Processa o pedido
        emailService.enviarEmail(
            mensagem:"Pedido processado com sucesso!");
    }
}
```

Neste exemplo, PedidoService depende diretamente de EmailService. Isso dificulta a troca de EmailService por outra forma de notificação (como SMS) e torna os testes mais difíceis, pois não há uma abstração.

Aplicando o princípo DIP:

```java
public interface NotificacaoService {
    void enviarNotificacao(String mensagem);
}

public class EmailService implements NotificacaoService {
    @Override
    public void enviarNotificacao(String mensagem) {
        System.out.println("Enviando e-mail: " + mensagem);
    }
}

public class SmsService implements NotificacaoService {
    @Override
    public void enviarNotificacao(String mensagem) {
        System.out.println("Enviando SMS: " + mensagem);
    }
}
```

Agora, PedidoService depende de NotificacaoService, e podemos injetar a dependência:

```
public class PedidoService {
    private NotificacaoService notificacaoService;

    public PedidoService(NotificacaoService notificacaoService) {
        this.notificacaoService = notificacaoService;
    }

    public void processarPedido() {
        // Processa o pedido
        notificacaoService.enviarNotificacao(
            "Pedido processado com sucesso!");
    }
}
```

Assim, ao criar um PedidoService, podemos escolher qual serviço de notificação usar:

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        NotificacaoService emailService = new EmailService();
        PedidoService pedidoService = new PedidoService(emailService);

        pedidoService.processarPedido();  // Usa o serviço de e-mail

        // Ou usando SMS
        NotificacaoService smsService = new SmsService();
        PedidoService pedidoServiceSms = new PedidoService(smsService);

        pedidoServiceSms.processarPedido();  // Usa o serviço de SMS
    }
}
```

Com isso:

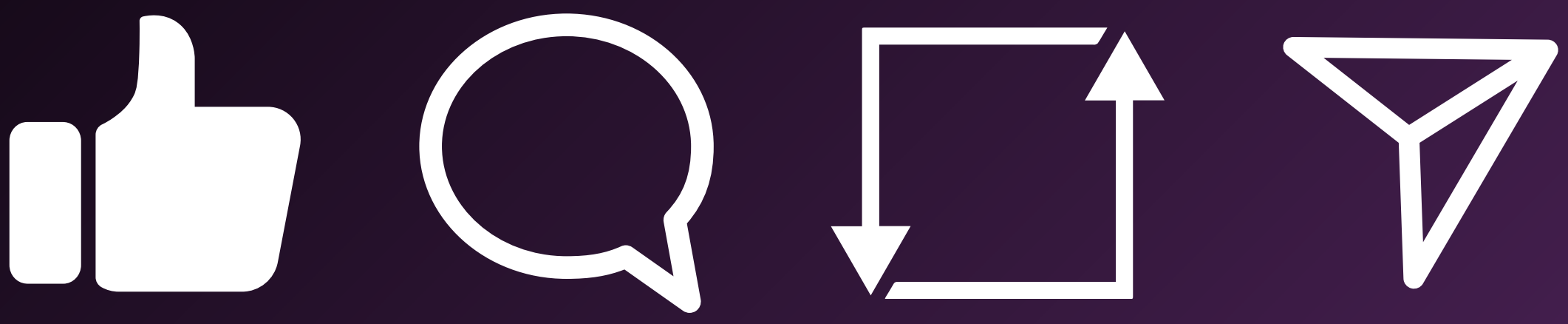
- PedidoService está desacoplado de EmailService e pode usar qualquer tipo de notificação.
- O código segue o Princípio da Inversão de Dependência, pois depende de uma abstração (NotificacaoService), não de um detalhe específico

CONCLUSÃO

Seguir o Princípio da Inversão de Dependência é como ter um brinquedo que funciona com qualquer bateria. No código, isso significa que as classes devem depender de abstrações, o que as torna mais flexíveis e fáceis de modificar.

Com esse conceito, o código fica mais organizado e preparado para o futuro!

Compartilha com aquele que quer aprender.



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

Que a força do código esteja com você !