

Paradigmas de Programação

Aula 04 – Como vai ser ???

- Aviso MUUUUIITTTTOOO IMPORTANTE!!!!
- AOP – Programação Orientada a Aspecto
- KISS, YAGNI e DRY
- High Cohesion & Low Coupling
- Princípios SOLID
- Entregáveis



ATENÇÃO!!!!!!

- Seguindo nosso planejamento, na próxima aula, vamos embarcar em uma nova jornada no mundo dos Microservices! 🌐
- E o melhor: vamos colocar a mão na massa! Iniciaremos a programação dos nossos primeiros Microservices utilizando o nosso velho e bom **Spring Boot 3**.
- Tenho certeza de que vocês vão gostar de ver como podemos aplicar esses conceitos na prática, criando soluções robustas e modernas.



ENTÃO!!!!

- É fundamental que vocês tragam seus computadores pessoais com todos os softwares necessários instalados e funcionando corretamente.
- Todos os detalhes sobre os softwares que vocês precisam estão listados na seção **Preparação do Ambiente para as Aulas** no nosso AVA.
- Por favor, confirmem essa seção e sigam as instruções para evitar qualquer contratempo durante a aula.
- Se precisarem de ajuda com a instalação, não hesitem em entrar em contato antes da aula.



Programação Orientada a Aspectos

AOP (Aspect-Oriented Programming)

AOP – Programação orientada a aspectos

- Programação Orientada a Aspectos (AOP, do inglês *Aspect-Oriented Programming*) tem como objetivo separar as preocupações transversais de uma aplicação, ou seja, aquelas funcionalidades que afetam várias partes do sistema, como o registro de logs, segurança, transações, entre outras.

Spring AOP

```
@Aspect
@Component
public class LogAspect {

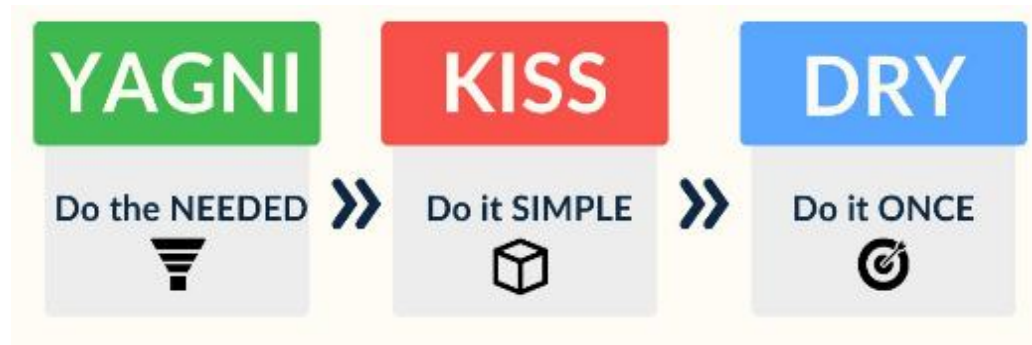
    @Before("execution(* com.exemplo.service.*.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("Método " + joinPoint.getSignature().getName() + " está sendo chamado...");
    }

    @After("execution(* com.exemplo.service.*.*(..))")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("Método " + joinPoint.getSignature().getName() + " foi executado.");
    }
}
```

Entregáveis Aula 03



Princípios KISS, YAGNI e DRY



- 1. KISS (Keep It Simple, Stupid):** Você deve manter suas soluções o mais simples possível. Evite criar soluções complicadas quando uma abordagem mais simples atender ao objetivo. Isso ajuda a reduzir a complexidade do código, facilitando a manutenção e a compreensão.
 - 2. DRY (Don't Repeat Yourself):** O DRY incentiva a reutilização de código. Em vez de repetir você deve modularizar o código e criar funções ou componentes reutilizáveis sempre que possível. Isso torna o código mais limpo, mais eficiente e mais fácil de manter.
 - 3. YAGNI (You Ain't Gonna Need It):** O YAGNI aconselha a não adicionar funcionalidades ou recursos ao seu código até que você realmente precise deles. Evite o desperdício de tempo e esforço em recursos que não são necessários no momento presente. Isso mantém o código mais simples e evita a sobrecarga de funcionalidades não utilizadas.
- Lembrando que esses princípios são diretrizes úteis para escrever código de qualidade e manutenível. Eles promovem a simplicidade, a reutilização e a eficiência no desenvolvimento de software.

O QUE QUEREMOS?



O que é Alta Coesão na Programação?

- É um princípio de design de software que se refere ao grau em que os elementos dentro de um módulo, classe, método ou função estão relacionados e trabalham juntos para cumprir uma única responsabilidade ou propósito.
- Em outras palavras, um módulo, classe, método ou função com alta coesão realiza um conjunto de tarefas intimamente relacionadas e bem definidas.

O que é Alta Coesão na Programação?

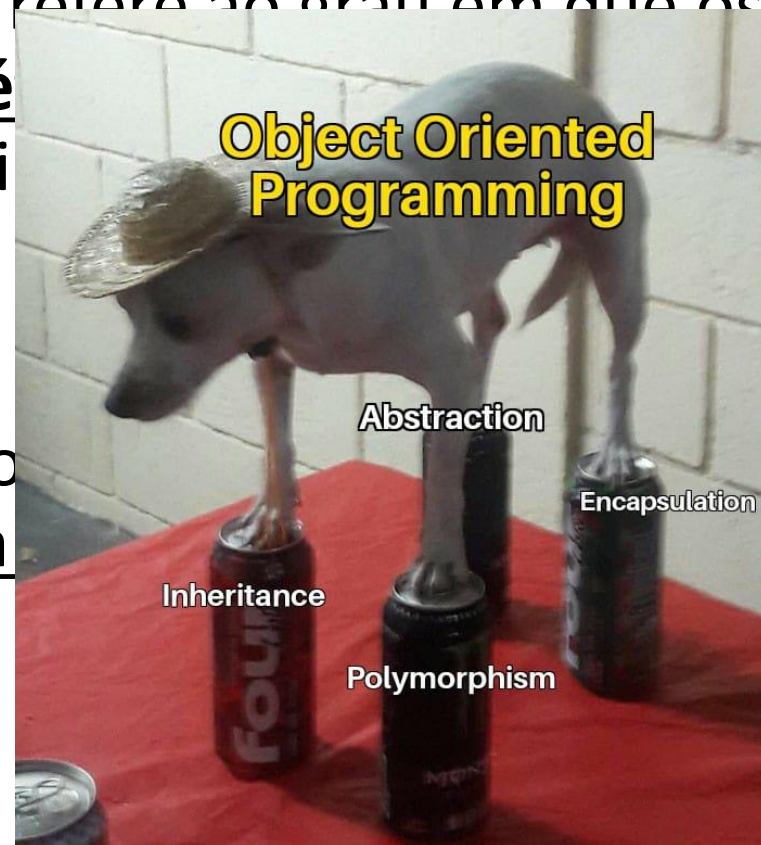
Mas esse conceito só vale
para Programação
Orientada a Objetos ???

- Em outras palavras, um módulo, classe, método ou coesão realiza um conjunto de tarefas intimamente bem definidas.

se refere ao grau em que os

mé

pri



O q

ção?

- É um
elem
relac
resp

o grau em que os
u função estão
única

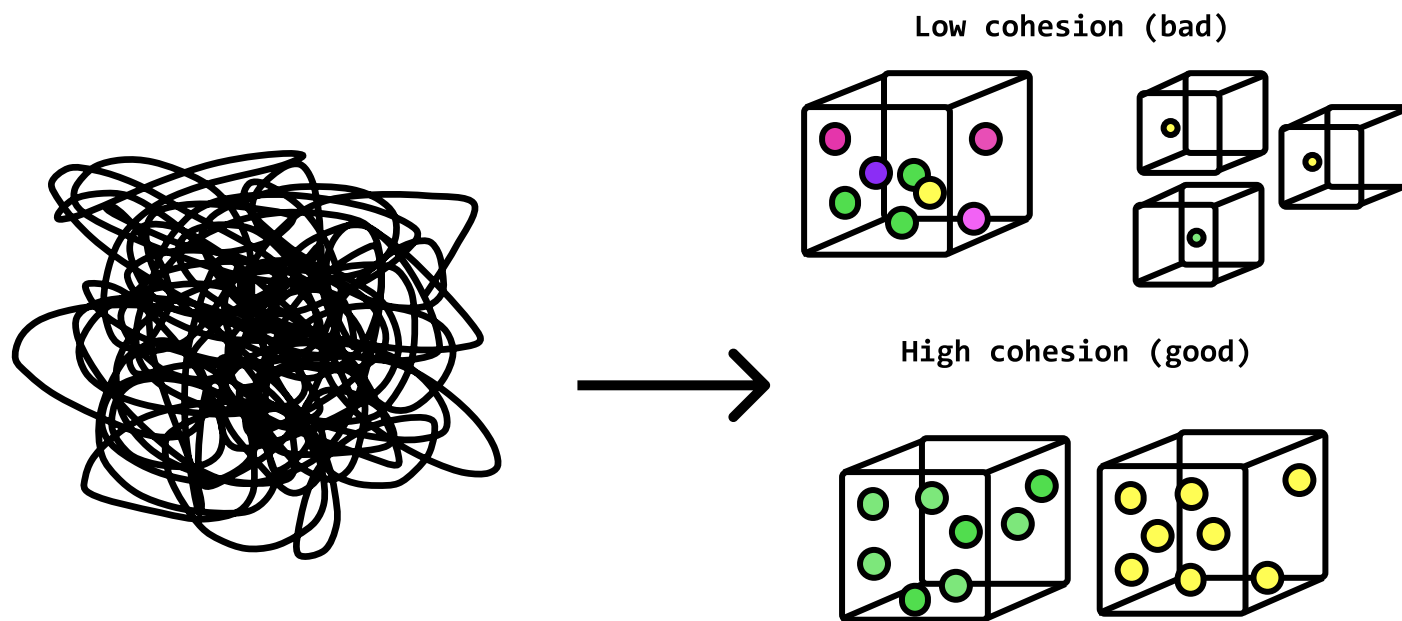
- Em c

coesão re ~~liza um conjunto de tarefas intimamente relacionadas em~~
bem defin

n módulo, classe, método ou função e
m juntos para cumprir uma única



Alta Coesão



Características de Alta Coesão



- **Foco e Clareza:** Cada módulo, classe, método ou função tem funcionalidades específicas e bem delimitadas, facilitando a compreensão do código.
- **Reutilização:** Classes ou módulos com alta coesão são mais fáceis de reutilizar em outros contextos, pois eles encapsulam funcionalidade específica de forma isolada.
- **Facilidade de Refatoração:** As responsabilidades estão bem organizadas e os impactos das mudanças são previsíveis.
- **Testabilidade:** Suas responsabilidades são bem definidas e limitadas, o que permite a criação de testes unitários claros e focados.

E o Baixo Acoplamento ???

- É um princípio de design de software que se refere à ideia de que os diferentes módulos, classes, métodos ou funções, ou seja, os **componentes de um sistema**, devem ter o menor número possível de dependências uns dos outros.
- Em outras palavras, componentes com baixo acoplamento funcionam de maneira mais independente, minimizando o impacto que mudanças em uma parte do sistema podem ter sobre outras partes.

É o Deixar Acoplamento ???

Mas esse conceito só vale
para Programação
Orientada a Objetos ???

- Em outras palavras, componentes com baixo acoplamento podem ser alterados de maneira mais independente, minimizando as mudanças em uma parte do sistema podem



E O

- É um defeito comum de projeto
- Em de r



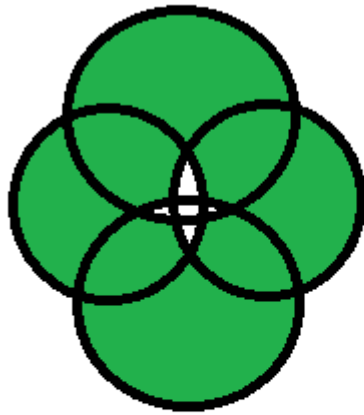
à ideia de que os
ou seja, os
número possível de

amento funcionam
o acto que

mudanças em uma parte do sistema podem ter sobre outras partes

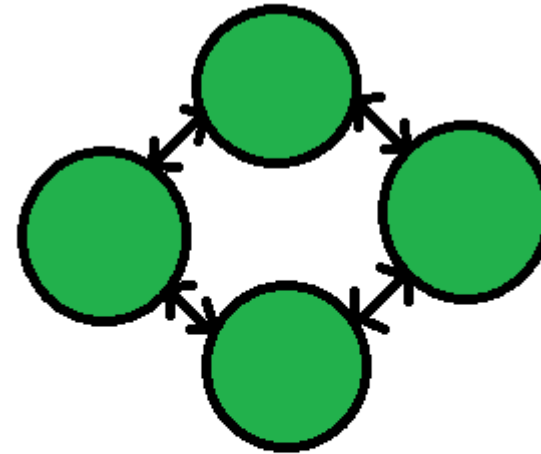
componentes de um sistema, d

Baixo Acoplamento



Tight coupling:

1. More Interdependency
2. More coordination
3. More information flow



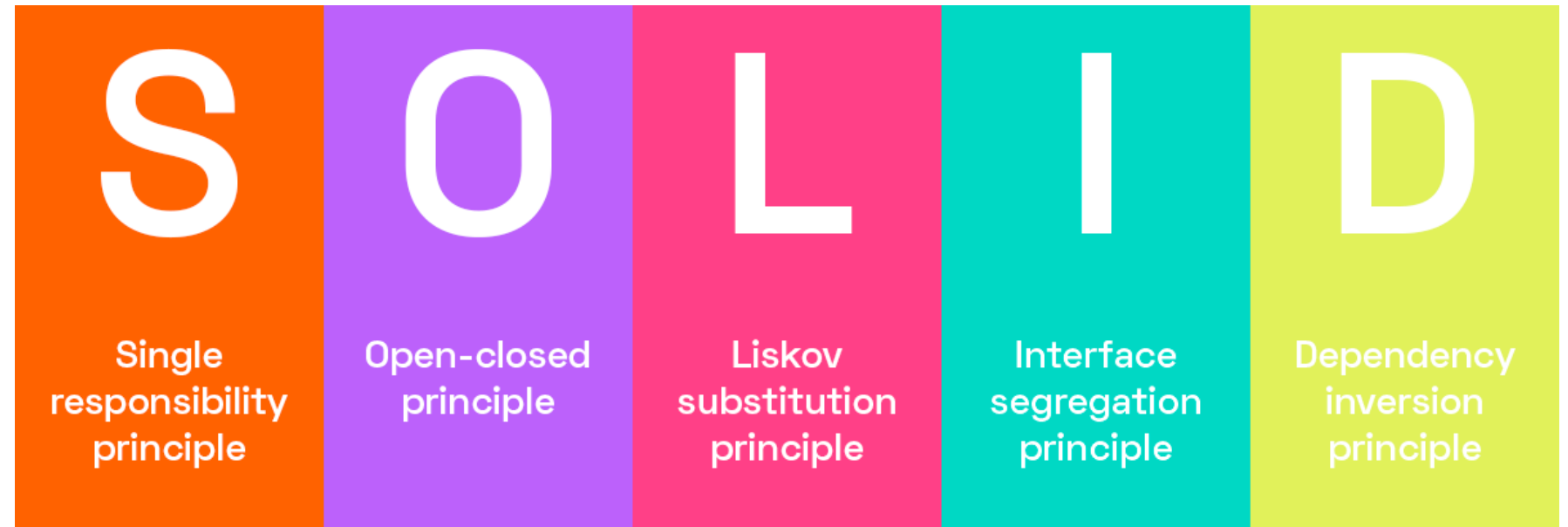
Loose coupling:

1. Less Interdependency
2. Less coordination
3. Less information flow

Características de Baixo Acoplamento



- **Independência:** Componentes com baixo acoplamento são mais independentes, o que significa que podem ser modificados, substituídos ou removidos sem afetar significativamente outras partes do sistema.
- **Facilidade de Manutenção:** A manutenção é facilitada, pois mudanças em um módulo têm menos probabilidade de causar efeitos colaterais em outros módulos.
- **Reutilização:** São mais facilmente reutilizáveis, já que eles não dependem fortemente de outros componentes específicos.
- **Testabilidade:** É mais fácil de testar, uma vez que os módulos podem ser testados de forma isolada, sem a necessidade de configurar muitas dependências.



Princípios SOLID

Princípios SOLID

- O SOLID é um conjunto de princípios de design de software que visa melhorar a qualidade do código, tornando-o mais flexível, manutenível e fácil de entender.
- Esses princípios ajudam a evitar códigos rígidos e difíceis de modificar, promovendo boas práticas de desenvolvimento orientado a objetos.

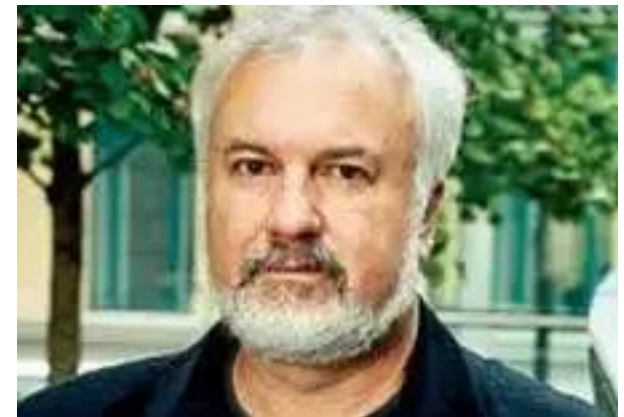
Princípios SOLID

- O primeiro indício dos princípios SOLID apareceu em 1995, no artigo *“The principles of OoD”* de **Robert C Martin**, também conhecido como **Uncle Bob**.
- Robert lançou, em 2002, o livro *“Agile Software Development, Principles, Patterns, and Practices”* que reúne diversos artigos sobre o tema.



Princípios SOLID

- Embora os princípios tenham sido descritos por Uncle Bob, foi **Michael Feathers** quem percebeu que os cinco princípios formavam a sigla "SOLID".
- Os princípios SOLID são amplamente utilizados no desenvolvimento de software, especialmente em aplicações empresariais que exigem escalabilidade, facilidade de manutenção e robustez.
- Eles também são fundamentais no contexto de **arquiteturas de microservices**, desenvolvimento ágil e boas práticas de programação.



Princípios SOLID

S

Single responsibility

Uma classe deve ter apenas uma única responsabilidade.

O

Open/closed

Software deve ser aberto para extensão, mas fechado para modificação.

L

Liskov substitution

Objetos de uma classe derivada devem poder substituir objetos da classe base sem alterar o comportamento desejado do programa.

I

Interface segregation

Os clientes não devem ser forçados a depender de interfaces que não utilizam.

D

Dependency inversion

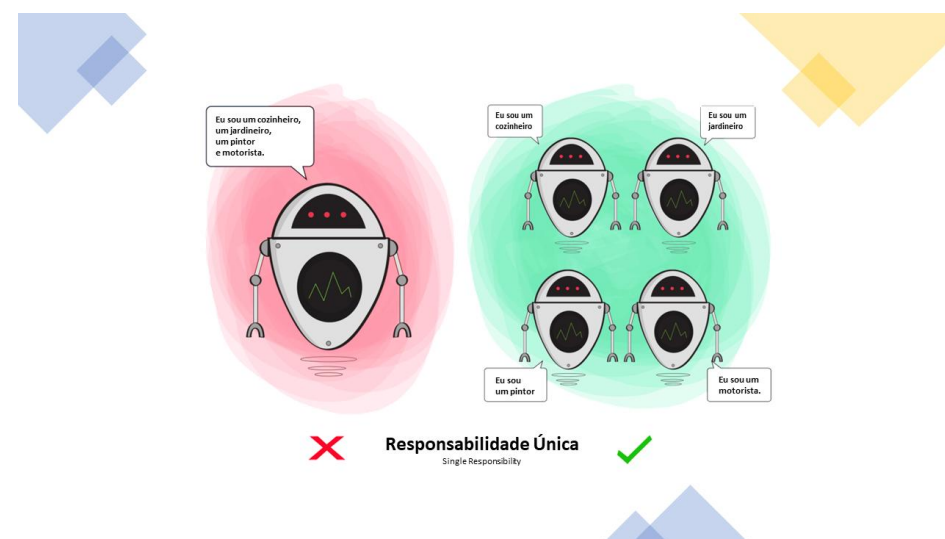
Dependa de abstrações, não de implementações.

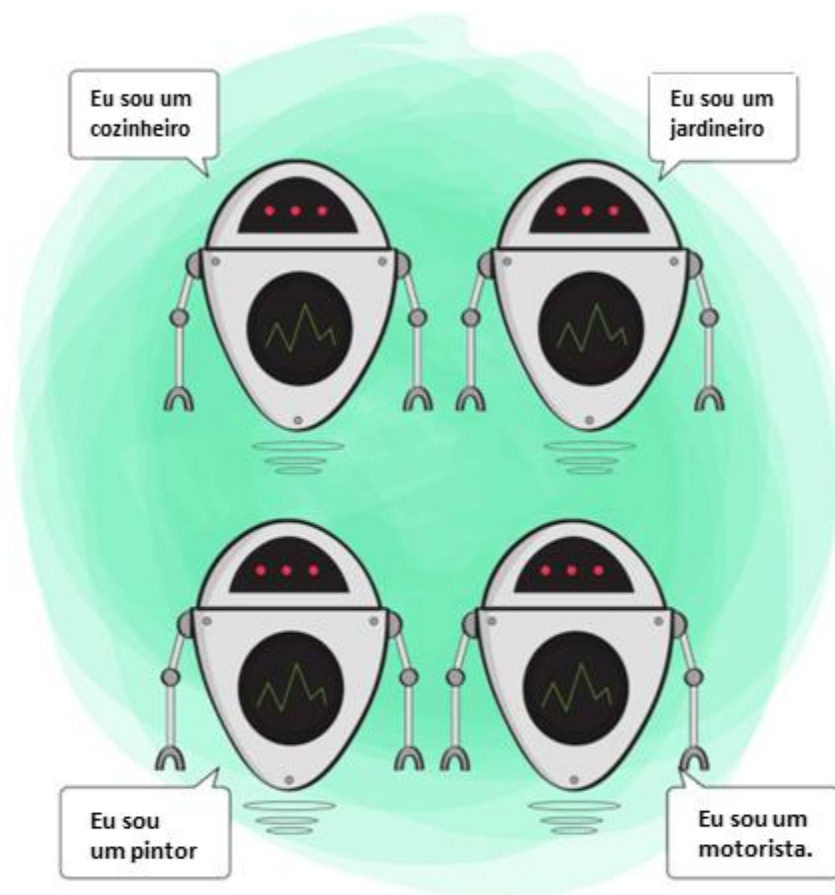
Single Responsibility Principle - SRP

Princípio da Responsabilidade Única

"Uma classe deve ter apenas um motivo para mudar."

- Ou seja, uma classe deve ter **apenas uma única responsabilidade** bem definida, evitando que ela assuma múltiplos papéis dentro do sistema.
- **SRP está diretamente ligado ao conceito de alta coesão.**





Responsabilidade Única

Single Responsibility



Single Responsibility Principle - SRP

Princípio da Responsabilidade Única

- **Se uma classe tem mais de um motivo para mudar, significa que ela está assumindo mais de uma responsabilidade.** Isso pode gerar vários problemas, como:
 - **Dificuldade na manutenção:** Se uma classe tem várias responsabilidades, qualquer alteração pode impactar funcionalidades não relacionadas.
 - **Baixa reutilização:** Classes com múltiplas responsabilidades tendem a ser menos reutilizáveis, pois carregam funcionalidades desnecessárias para certos contextos.
 - **Alto acoplamento:** Quando uma classe faz "de tudo um pouco", outras partes do sistema passam a depender dela de forma desnecessária.

```
public class RelatorioService {  
    public void gerarRelatorio() {  
        // Lógica para gerar o relatório  
        System.out.println("Relatório gerado.");  
        // Lógica para salvar o relatório no banco de dados  
        System.out.println("Relatório salvo no banco.");  
    }  
}
```

```
public class RelatorioService {  
    public void gerarRelatorio() {  
        // Lógica para gerar o relatório  
        System.out.println("Relatório gerado.");  
        // Lógica para salvar o relatório no BD  
        System.out.println("Relatório salvo no BD.");  
    }  
}
```



A Classe RelatorioService possui duas responsabilidades distintas.
E ainda cria um problema:
“Se eu quiser apenas gerar o relatório, mas não salvar no BD”

E agora ???

```
public class RelatorioService {  
    public void gerarRelatorio() {  
        // Lógica para gerar o relatório  
        System.out.println("Relatório gerado.");  
    }  
  
    public void salvarNoBanco() {  
        // Lógica para salvar o relatório no banco de dados  
        System.out.println("Relatório salvo no banco.");  
    }  
}
```



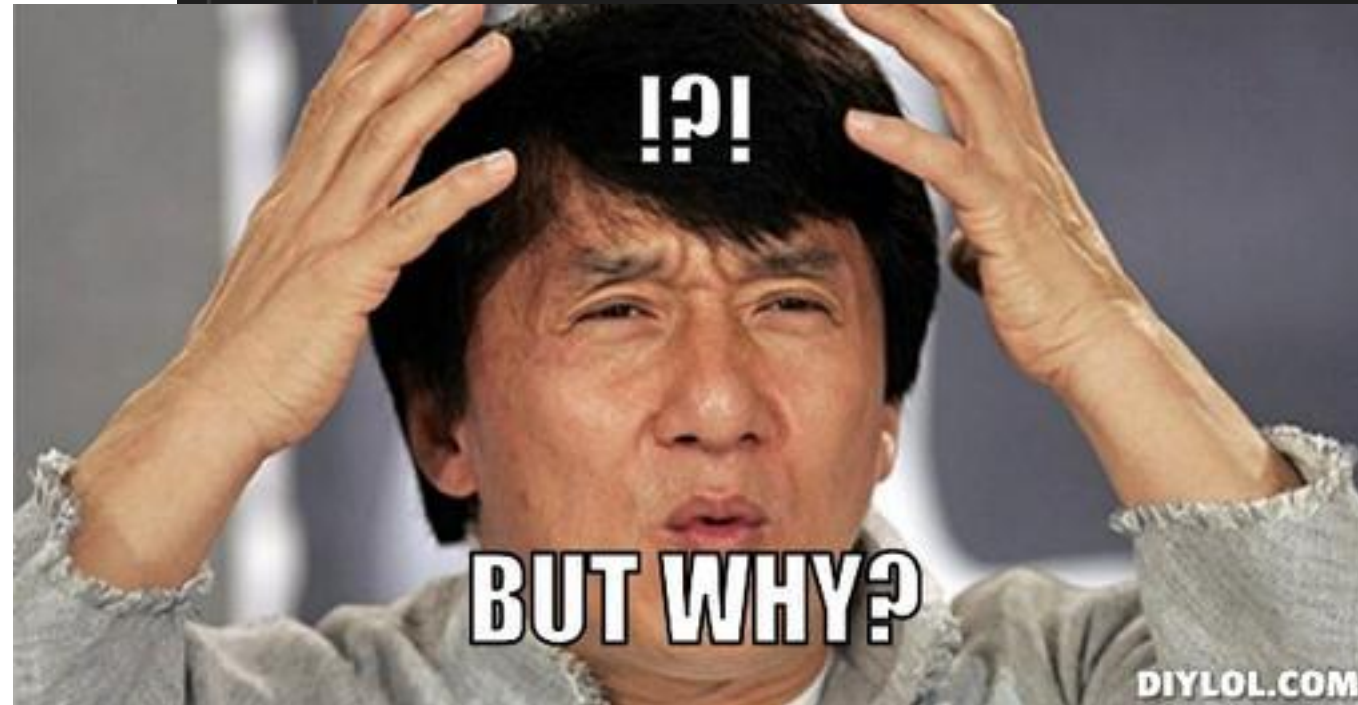
E agora ???

```
public class RelatorioService {  
    public void gerarRelatorio() {  
        // Lógica para gerar o relatório  
        System.out.println("Relatório gerado.");  
    }  
  
    public void salvarNoBanco() {  
        // Lógica para salvar o relatório no banco de dados  
        System.out.println("Relatório salvo no banco.");  
    }  
}
```



E agora ???

```
public class RelatorioService {  
    public void gerarRelatorio() {  
        // Lógica para gerar o relatório
```



```
        .");  
  
        banco de dados  
        no banco.");
```

E agora ???

```
public class RelatorioService {  
    public void gerarRelatorio() {  
        // Lógica para gerar o relatório  
        System.out.println("Relatório gerado.");  
    }  
  
    public void salvarNoBanco() {  
        // Lógica para salvar o relatório no banco de dados  
        System.out.println("Relatório salvo no banco.");  
    }  
}
```

A Classe RelatorioService continua com duas responsabilidades distintas.

DOIS MOTIVOS PARA MUDAR:

- Preciso alterar o formato do relatório (pdf -> xml)
- Preciso alterar o tipo do BD de destino (mysql -> postgres)

E então ???



E então ???

```
public class RelatorioService {  
    public void gerarRelatorio() {  
        System.out.println("Relatório gerado.");  
    }  
}
```



```
public class RelatorioRepository {  
    public void salvarNoBanco() {  
        System.out.println("Relatório salvo no banco.");  
    }  
}
```

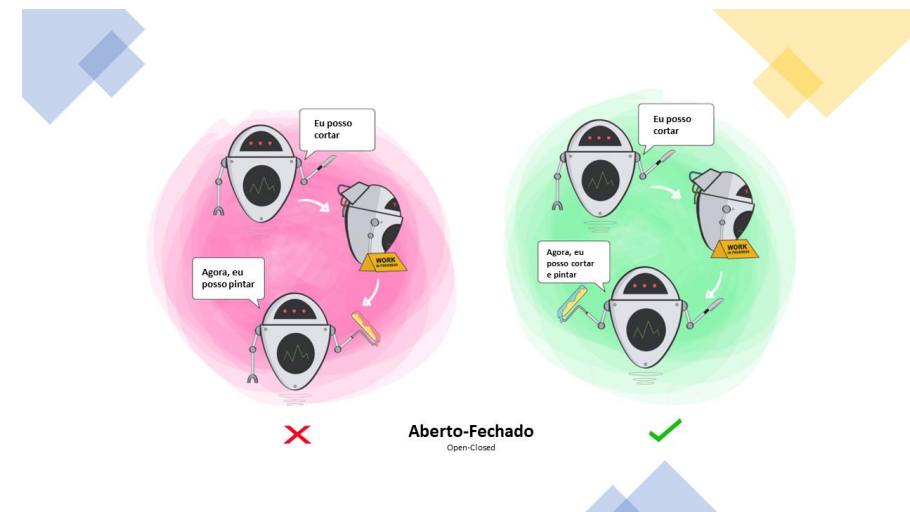
Mais um
exemplo de
violação de
SRP.

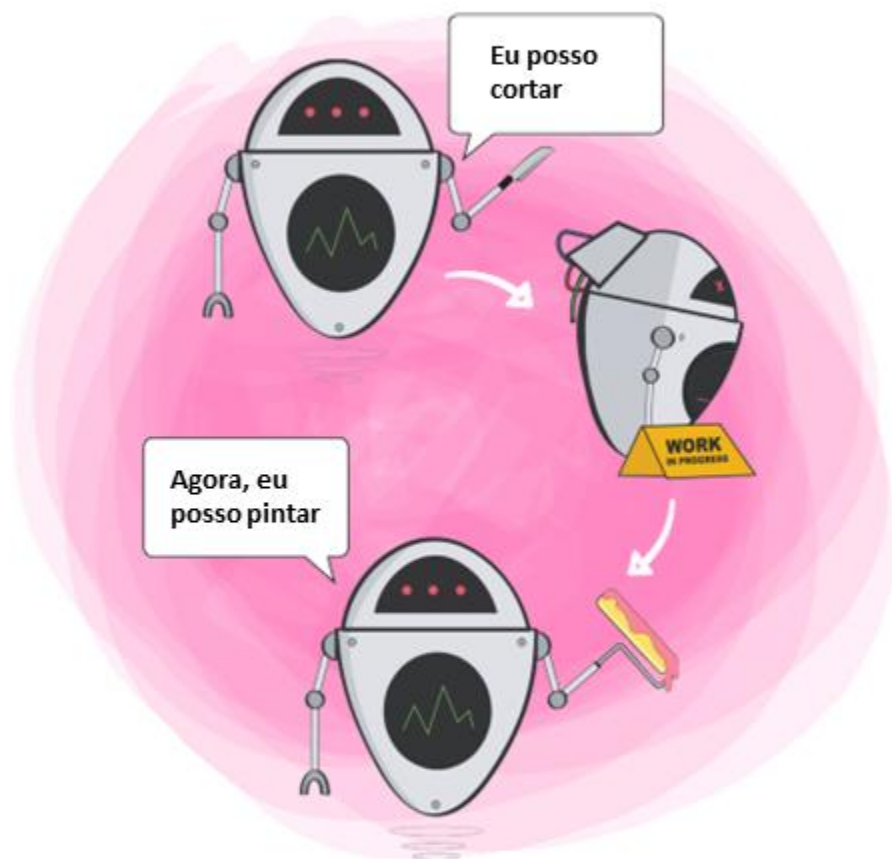
```
public class Pedido {  
    private List<Item> itens;  
    private double total;  
  
    public void adicionarItem(Item item) {  
        itens.add(item);  
    }  
  
    public void calcularTotal() {  
        total = itens.stream().mapToDouble(Item::getPreco).sum();  
    }  
  
    public void gerarNotaFiscal() {  
        System.out.println("Gerando nota fiscal...");  
    }  
  
    public void enviarEmailConfirmacao() {  
        System.out.println("Enviando e-mail de confirmação...");  
    }  
}
```


Open/Closed Principle - OCP

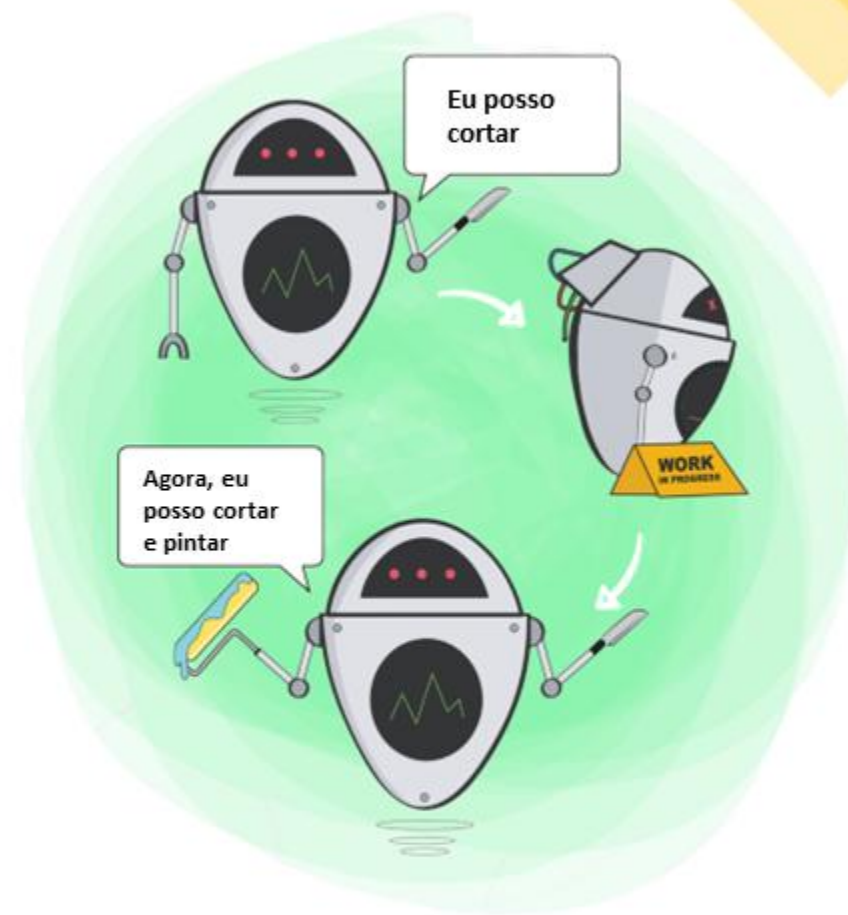
Princípio Aberto-Fechado

- **Entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação.**
- Ou seja, você deve poder **adicionar novos comportamentos ao sistema sem precisar modificar código existente.**
- Esse princípio ajuda a evitar efeitos colaterais em código já testado e mantém o sistema mais **flexível e extensível.**





Aberto-Fechado
Open-Closed




```
public class CalculadoraDesconto {  
    public double calcularDesconto(String tipoCliente, double valorCompra) {  
        if (tipoCliente.equals("Comum")) {  
            return valorCompra * 0.05;  
        } else if (tipoCliente.equals("VIP")) {  
            return valorCompra * 0.10;  
        } else if (tipoCliente.equals("Funcionário")) {  
            return valorCompra * 0.15;  
        } else {  
            return 0;  
        }  
    }  
}
```

```
public class CalculadoraDesconto {  
    public double calcularDesconto(String tipoCliente, double valorCompra) {  
        if (tipoCliente.equals("Comum")) {  
            return valorCompra * 0.05;  
        } else if (tipoCliente.equals("VIP")) {  
            return valorCompra * 0.10;  
        } else if (tipoCliente.equals("Funcionário")) {  
            return valorCompra * 0.15;  
        } else {  
            return 0;  
        }  
    }  
}
```



```
public class CalculadoraDesconto {  
    public double calcularDesconto(String tipoCliente, double valorCompra) {  
        if (tipoCliente.equals("Comum")) {  
            return valorCompra * 0.05;  
        } else if (tipoCliente.equals("VIP")) {  
            return valorCompra * 0.10;  
        } else if (tipoCliente.equals("Funcionário")) {  
            return valorCompra * 0.15;  
        } else {  
            return 0;  
        }  
    }  
}
```

Problemas 🚨

- 1 Toda vez que surgir um novo tipo de cliente, será necessário modificar essa classe. Isso viola o OCP, pois o código não está fechado para modificação.
- 2 A classe fica menos testável e mais sujeita a erros, pois mexer nela pode quebrar funcionalidades existentes.

E então ???



```
public interface Desconto {  
    double calcular(double valorCompra);  
}
```

```
public interface Desconto {  
    double calcular(double valorCompra);  
}  
  
public class DescontoComum implements Desconto {  
    public double calcular(double valorCompra) {  
        return valorCompra * 0.05;  
    }  
}
```

```
public interface Desconto {  
    double calcular(double valorCompra);  
}  
  
public class DescontoComum implements Desconto {  
    public double calcular(double valorCompra) {  
        return valorCompra * 0.05;  
    }  
}  
  
public class DescontoVIP implements Desconto {  
    public double calcular(double valorCompra) {  
        return valorCompra * 0.10;  
    }  
}
```

```
public interface Desconto {  
    double calcular(double valorCompra);  
}  
  
public class DescontoComum implements Desconto {  
    public double calcular(double valorCompra) {  
        return valorCompra * 0.05;  
    }  
}  
  
public class DescontoVIP implements Desconto {  
    public double calcular(double valorCompra) {  
        return valorCompra * 0.10;  
    }  
}
```

```
public class CalculadoraDesconto {  
    public double calcularDesconto(Desconto desconto, double valorCompra) {  
        return desconto.calcular(valorCompra);  
    }  
}
```


E então

Vantagens dessa abordagem ✓

- ✓ **Aberto para extensão** – Podemos adicionar novos tipos de desconto criando novas classes, sem mexer no código original.
- ✓ **Fechado para modificação** – A classe `CalculadoraDesconto` nunca precisará ser alterada.
- ✓ **Código mais organizado** – Segue o princípio da **Responsabilidade Única (SRP)**, pois cada classe tem uma função bem definida.
- ✓ **Facilidade de manutenção** – Novos descontos podem ser adicionados sem riscos de quebrar código existente.

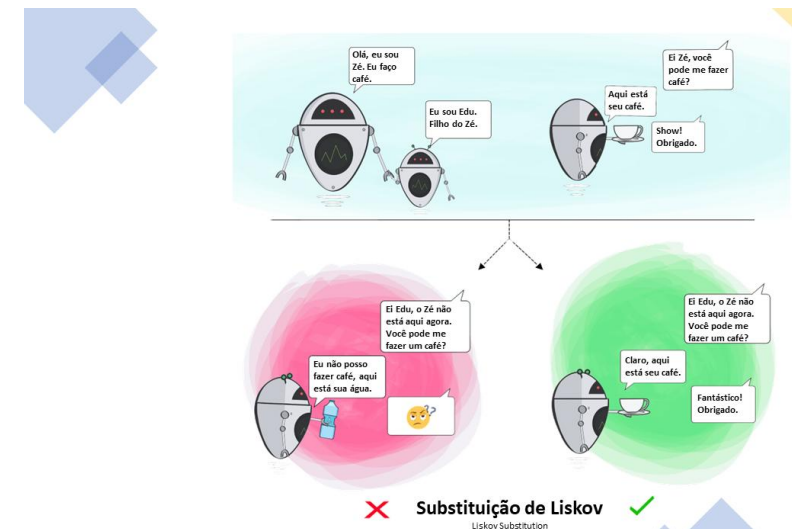


Liskov Substitution Principle - LSP

Princípio da Substituição de Liskov

"Se S é uma subclasse de T , então objetos de tipo T podem ser substituídos por objetos de tipo S sem alterar as propriedades do programa."

- Em outras palavras, uma subclasse deve ser capaz de substituir sua superclasse sem quebrar o comportamento esperado do sistema.





Substituição de Liskov

Liskov Substitution



```
public class Usuario {  
    protected String nome;  
  
    public Usuario(String nome) {  
        this.nome = nome;  
    }  
  
    public void acessarSistema() {  
        System.out.println(nome + " acessou o sistema.");  
    }  
}
```

```
public class Usuario {  
    protected String nome;  
  
    public Usuario(String nome) {  
        this.nome = nome;  
    }  
  
    public void acessarSistema() {  
        System.out.println(nome + " acessou o sistema.");  
    }  
}
```

```
public class UsuarioConvidado extends Usuario {  
    public UsuarioConvidado(String nome) {  
        super(nome);  
    }  
  
    @Override  
    public void acessarSistema() {  
        throw new UnsupportedOperationException  
            ("Convidados não têm acesso ao sistema.");  
    }  
}
```

Até agora tudo bem... Vamos testar

```
public class ControleAcesso {  
    public static void acessarSistema(Usuario usuario) {  
        usuario.acessarSistema();  
    }  
    public static void main(String[] args) {  
        Usuario user = new Usuario("Maria");  
        acessarSistema(user);  
  
        UsuarioConvidado guest = new UsuarioConvidado("João");  
        acessarSistema(guest);  
    }  
}
```

Até agora tudo bem... Vamos testar

```
public class ControleAcesso {  
    public static void acessarSistema(Usua  
        usuario.acessarSistema();  
}
```




Maria acessou o sistema.

Exception in thread "main" java.lang.UnsupportedOperationException:
Convidados não têm acesso ao sistema.

```
at UsuarioConvidado.acessarSistema(UsuarioConvidado.java:8)  
at ControleAcesso.acessarSistema(ControleAcesso.java:3)  
at ControleAcesso.main(ControleAcesso.java:11)
```

```
    }  
}
```

```
public class ControleAcesso {  
    public static void acessarSistema(Usuario usuario) {  
        usuario.acessarSistema();  
    }  
    public static void main(String[] args) {  
        Usuario user = new Usuario("Maria");  
        acessarSistema(user);  
  
        UsuarioConvidado guest = new UsuarioConvidado("João");  
        acessarSistema(guest);  
    }  
}
```



Criamos ***UsuarioConvidado*** como uma subclasse de ***Usuario***, mas ele não pode ser usado no lugar de ***Usuario***, pois lança uma exceção inesperada.

Ou seja, **NÃO** é possível substituir um “***Usuario***” (***superclasse***) por um “***UsuarioConvidado***” (subclasse) sem quebrar o sistema ou ter um comportamento inesperado, ferindo assim o LSP.

E então ???



```
public interface Autenticavel {  
    void acessarSistema();  
}
```

```
public interface Autenticavel {
```

```
}
```

```
public class Usuario implements Autenticavel {
```

```
    protected String nome;
```

```
    public Usuario(String nome) {
```

```
        this.nome = nome;
```

```
    }
```

```
    public void acessarSistema() {
```

```
        System.out.println(nome + " acessou o sistema.");
```

```
    }
```

```
}
```

```
public class UsuarioConvidado {
```

```
    private String nome;
```

```
    public UsuarioConvidado(String nome) {
```

```
        this.nome = nome;
```

```
    }
```

```
    public void visualizarConteudo() {
```

```
        System.out.println
```

```
            (nome + " está visualizando o conteúdo como convidado.");
```

```
    }
```

```
}
```

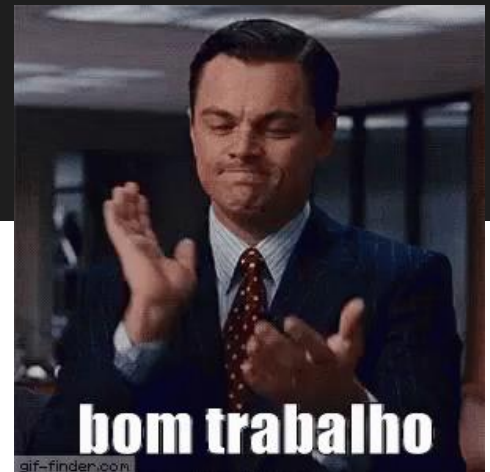
```
public class ControleAcesso {  
    public static void acessarSistema(Autenticavel usuarioAuth) {  
        usuarioAuth.acessarSistema();  
    }  
    public static void main(String[] args) {  
        Usuario user = new Usuario("Maria");  
        acessarSistema(user);  
  
        UsuarioConvidado guest = new UsuarioConvidado("João");  
        acessarSistema(guest);  
    }  
}
```



```
public class ControleAcesso {  
    public static void acessarSistema(Autenticavel usuarioAuth) {  
        usuarioAuth.acessarSistema();  
    }  
}
```

Aqui temos um erro em tempo de compilação, evitando a quebra do programa em tempo de execução. Isso obriga o programador a corrigir o problema antes de rodar o código, garantindo o cumprimento do LSP.

```
UsuarioConvidado guest = new UsuarioConvidado("João");  
acessarSistema(guest);  
}  
}
```



Barbara Liskov

Cientista de computação :

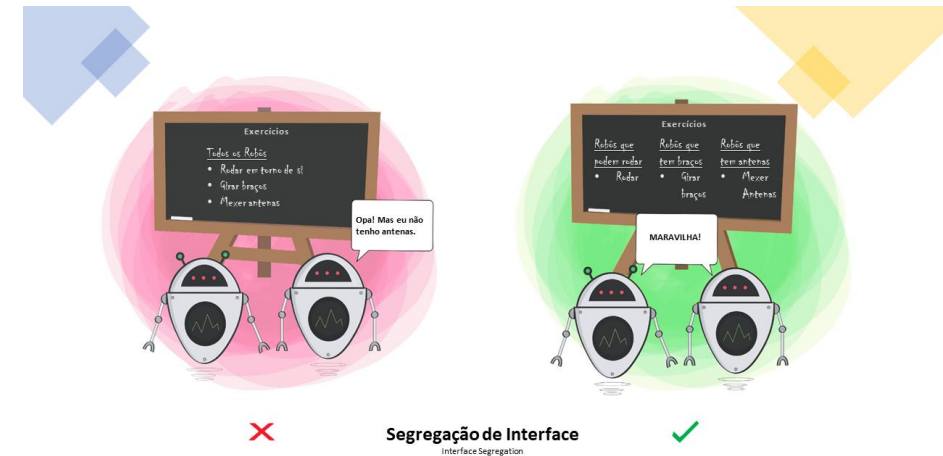


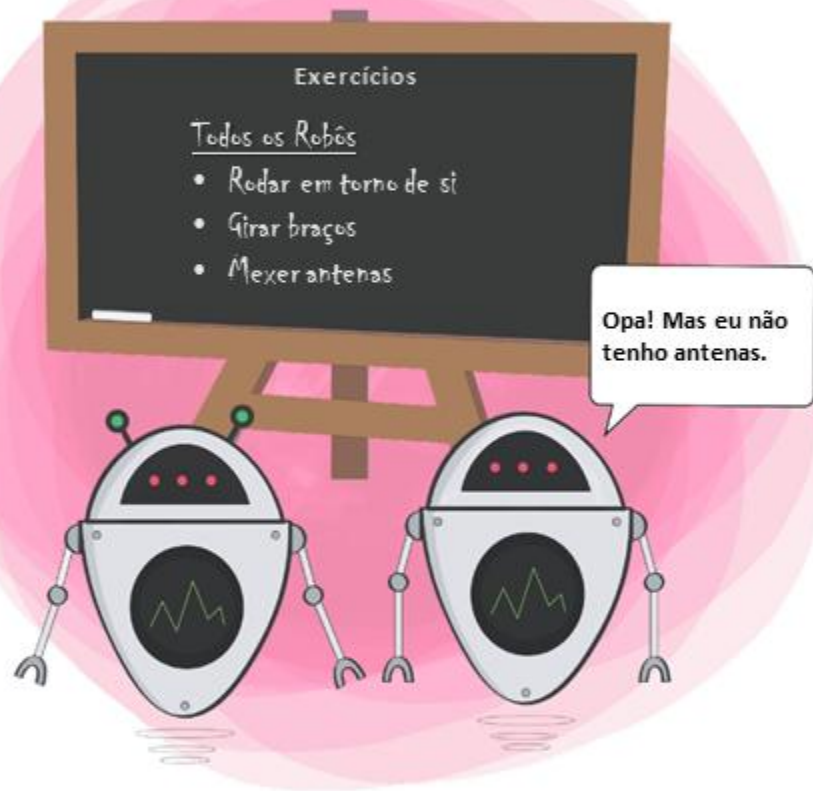
Interface Segregation Principle - ISP

Princípio da Segregação de Interface

"Uma classe não deve ser forçada a depender de métodos que não utiliza."

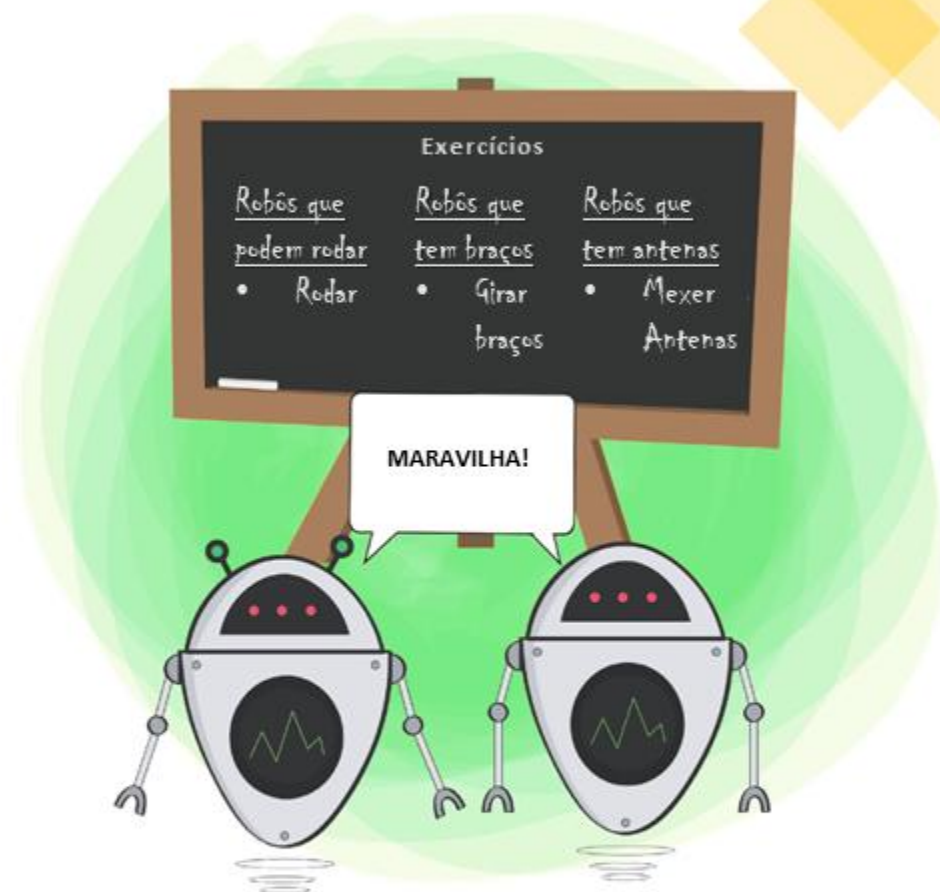
- Ou seja, **interfaces grandes e genéricas devem ser divididas em interfaces menores e mais específicas**, garantindo que as classes implementem apenas o que realmente precisam.





Segregação de Interface

Interface Segregation




```
public interface Trabalhador {  
    void trabalhar();  
    void comer();  
}
```

```
public interface Trabalhador {  
    void trabalhar();  
    void comer();  
}
```

```
public class Programador implements Trabalhador {  
    @Override  
    public void trabalhar() {  
        System.out.println("Programador escrevendo código.");  
    }  
    @Override  
    public void comer() {  
        System.out.println("Programador também se alimenta.");  
    }  
}
```

```
public interface Trabalhador {  
    void trabalhar();  
    void comer();  
}
```

```
public class Programador implements Trabalhador {  
    @Override  
    public void trabalhar() {  
        System.out.println("Programador escrevendo código.");  
    }  
    @Override  
    public void comer() {  
        System.out.println("Programador comendo.");  
    }  
}
```

```
public class Robo implements Trabalhador {  
    @Override  
    public void trabalhar() {  
        System.out.println("Robô executando tarefas automatizadas.");  
    }  
    @Override  
    public void comer() {  
        throw new UnsupportedOperationException("Robôs não comem!");  
    }  
}
```

```
public interface Trabalhador {  
    void trabalhar();  
}
```

```
public class Robo implements Trabalhador {  
    @Override  
    public void trabalhar() {  
        System.out.println("Robô executando tarefa.");  
    }  
    @Override  
    public void comer() {  
        throw new UnsupportedOperationException("Robôs não comem!");  
    }  
}
```



A interface **Trabalhador** obriga todas as classes a implementarem o método **comer()**, mas um robô não precisa desse comportamento. Isso viola o ISP, pois **Robo** precisa sobrescrever um método que não faz sentido para ele, gerando um erro inesperado.

E então ???



```
public interface Trabalhavel {  
    void trabalhar();  
}
```

```
public interface Alimentavel {  
    void comer();  
}
```

```
public interface Trabalhavel {  
    void trabalhar();  
}
```

```
public interface Alimentavel {  
    void comer();  
}
```

```
public class Programador implements Trabalhavel, Alimentavel {  
    @Override  
    public void trabalhar() {  
        System.out.println("Programador escrevendo código.");  
    }  
    @Override  
    public void comer() {  
        System.out.println("Programador também se alimenta.");  
    }  
}
```

```
public interface Trabalhavel {  
    void trabalhar();  
}
```

```
public interface Alimentavel {  
    void comer();  
}
```

```
public class Programador implements Trabalhavel, Alimentavel {  
    @Override  
    public void trabalhar() {  
        System.out.println("Programador escrevendo código.");  
    }  
    @Override
```

```
public class Robo implements Trabalhavel {  
    @Override  
    public void trabalhar() {  
        System.out.println("Robô executando tarefas au  
    }  
}
```

Agora respeitamos o ISP!

- ✓ As classes implementam apenas os métodos que realmente usam.
- ✓ O código ficou mais modular e flexível.
- ✓ Evitamos métodos desnecessários e código propenso a erros.



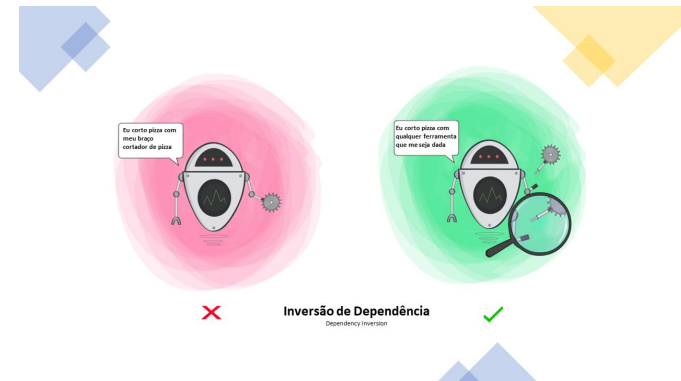
Dependency inversion Principle - DIP

Princípio da Inversão de Dependências

"Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações."

"Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações."

- Em resumo, **evite dependências diretas de implementações concretas**. Em vez disso, use **interfaces ou classes abstratas** para desacoplar os componentes.



Eu corto pizza com
meu braço
cortador de pizza



Eu corto pizza com
qualquer ferramenta
que me seja dada



Inversão de Dependência

Dependency Inversion

```
public class MySQLPedidoRepository {  
    public void salvarPedido() {  
        System.out.println("Pedido salvo no MySQL.");  
    }  
}
```

```
public class MySQLPedidoRepository {  
    public void salvarPedido() {  
        System.out.println("Pedido salvo no MySQL.");  
    }  
}
```

```
public class PedidoService {  
    private MySQLPedidoRepository pedidoRepository;  
    public PedidoService() {  
        this.pedidoRepository = new MySQLPedidoRepository();  
        // DEPENDÊNCIA DIRETA!  
    }  
    public void processarPedido() {  
        System.out.println("Processando pedido...");  
        pedidoRepository.salvarPedido();  
    }  
}
```

```
public class MySQLPedidoRepository {  
    public void salvarPedido() {  
        System.out.println("Pedido salvo no MySQL.");  
    }  
}
```

```
public class PedidoService {  
    private MySQLPedidoRepository pedidoRepository;  
    public PedidoService() {  
        this.pedidoRepository = new MySQLPedidoRepository();  
        // DEPENDÊNCIA DIRETA!  
    }  
    public void processarPedido() {  
        System.out.println("Processando pedido...");  
        pedidoRepository.salvarPedido();  
    }  
}
```

PedidoService depende diretamente de ***MySQLPedidoRepository***.

- Se quisermos trocar o banco de dados (ex.: PostgreSQL ou MongoDB), teremos que modificar o código de **PedidoService**, violando o DIP.
- O código está rígido e difícil de testar.

E então ???



```
public interface PedidoRepository {  
    void salvarPedido();  
}
```

Aqui criamos uma abstração para o Repositório de Pedidos

```
public interface PedidoRepository {  
    void salvarPedido();  
}
```

```
public class MySQLPedidoRepository implements PedidoRepository {  
    @Override  
    public void salvarPedido() {  
        System.out.println("Pedido salvo no MySQL.");  
    }  
}
```

```
public class PostgreSQLPedidoRepository implements PedidoRepository {  
    @Override  
    public void salvarPedido() {  
        System.out.println("Pedido salvo no PostgreSQL");  
    }  
}
```


Apartir da abstração podemos criar “n” classes concretas, cada uma com sua particularidade


```
public interface PedidoRepository {
```

```
public class MySQLPedidoRepository implements PedidoRepository {  
    @Override
```

```
public class PostgreSQLPedidoRepository implements PedidoRepository {  
    @Override  
    public void salvarPedido(  
        System.out.println("P  
    }  
}
```

```
public class PedidoService {  
    private PedidoRepository pedidoRepository;  
  
    // Injeção de dependência via construtor  
    public PedidoService(PedidoRepository pedidoRepository) {  
        this.pedidoRepository = pedidoRepository;  
    }  
  
    public void processarPedido() {  
        System.out.println("Processando pedido...");  
        pedidoRepository.salvarPedido();  
    }  
}
```



Agora nosso **PedidoService** depende de uma **Abstração**. Ou seja, essa classe não depende se o banco vai ser PostgreSQL, MySQL, MariaDB, ou qualquer outro.

```
public interface PedidoRepository {
```

```
    public class MySQLPedidoRepository implements PedidoRepository {  
        @Override
```

```
    public class PostgreSQLPedidoRepository implements PedidoRepository {
```

```
        public class PedidoService {  
            private PedidoRepository pedidoRepository;
```

```
        public static void main(String[] args) {  
            PedidoRepository repository = new MySQLPedidoRepository();  
            PedidoService service = new PedidoService(repository);  
            service.processarPedido();  
  
            PedidoRepository outroRepository = new PostgreSQLPedidoRepository();  
            PedidoService outroService = new PedidoService(outroRepository);  
            outroService.processarPedido();  
        }
```

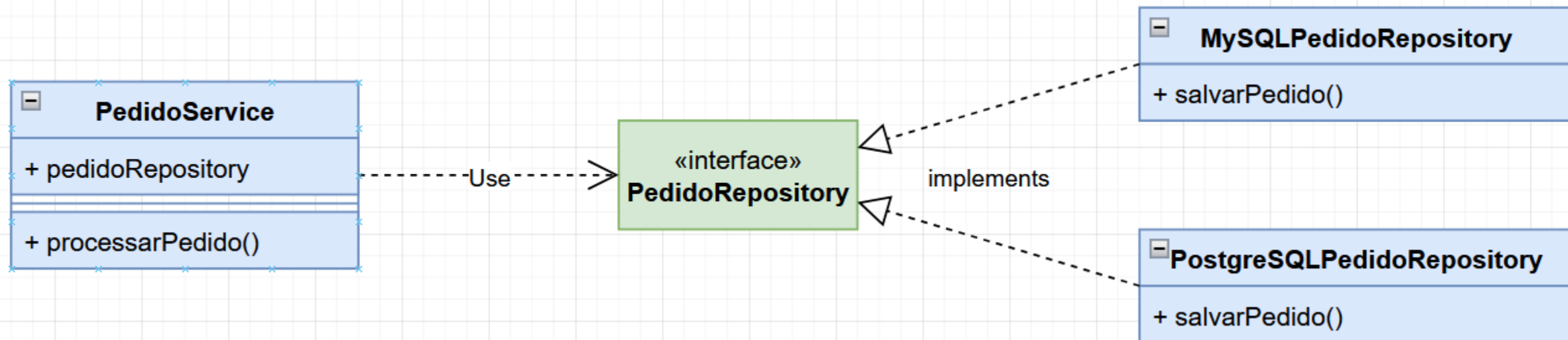


Agora podemos utilizar o **ProdutoService** com qualquer implementação de **ProdutoRepository**.

Por exemplo, podemos mudar o banco sem alterar a implementação de **ProdutoService**

Agora respeitamos o DIP! 🎯

- ✓ `PedidoService` não depende mais diretamente de um banco específico.
- ✓ Podemos trocar a implementação de `PedidoRepository` sem modificar `PedidoService`.
- ✓ O código está mais flexível, testável e modular.

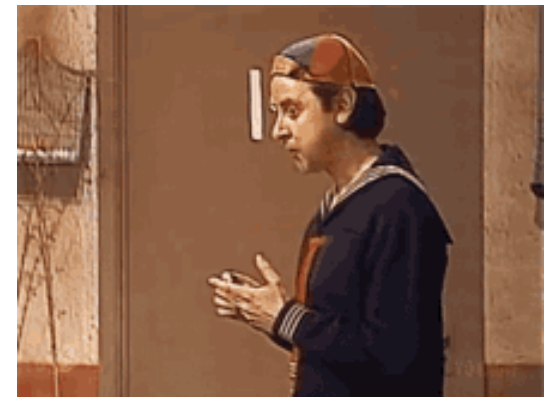


Princípio	Alta Coesão	Baixo Acoplamento	Explicação
SRP - Princípio da Responsabilidade Única	✓ Fortalece	✓ Reduz o acoplamento	Uma classe deve ter apenas uma razão para mudar . Isso mantém os métodos e atributos relacionados, aumentando a coesão , e evita que uma classe dependa de várias responsabilidades diferentes, reduzindo o acoplamento .
OCP - Princípio Aberto-Fechado	✓ Mantém	✓ Maximiza	Permite a extensão do código sem modificar o existente , garantindo que novas funcionalidades não alterem classes já testadas. Isso evita dependências desnecessárias e mantém as classes mais modulares .
LSP - Princípio da Substituição de Liskov	⊘ Neutro	✓ Reduz o acoplamento	Classes derivadas devem poder substituir suas classes base sem quebrar o código. Isso reduz dependências inesperadas e melhora a intercambiabilidade , diminuindo o acoplamento . Não afeta diretamente a coesão , mas impede dependências rígidas.
ISP - Princípio da Segregação de Interface	✓ Fortalece	✓ Maximiza	Evita interfaces gigantes, dividindo-as em partes menores e mais coesas. Isso garante que classes implementem apenas o que precisam , aumentando a coesão e evitando dependências desnecessárias entre partes do sistema.
DIP - Princípio da Inversão de Dependência	⊘ Neutro	✓ Maximiza	Força o código a depender de abstrações, não de implementações concretas , eliminando dependências diretas. Isso reduz drasticamente o acoplamento e torna o sistema mais flexível.

Princípios SOLID - Resumo

- **Alta Coesão** é mais afetada por **SRP e ISP**, pois ambos garantem que cada classe ou módulo tenha uma única responsabilidade bem definida.
- **Baixo Acoplamento** é beneficiado por **todos os princípios SOLID**, mas principalmente **DIP, OCP e ISP**, pois promovem a modularidade e evitam dependências diretas.

🚀 **Conclusão:** Quando aplicamos SOLID corretamente, conseguimos um **código modular, flexível e sustentável**, reduzindo impactos negativos de mudanças no sistema. Isso leva a um software mais **manutenível, testável e escalável**. 🔥



Entregáveis

Vamos lá. Está na hora!!!



