Lorenzon 2025-2: OO - S.O.L.I.D. com exemplos em Java

Definição: O SOLID é um conjunto de princípios fundamentais de design orientado a objetos. Ele não é um framework nem uma linguagem, mas um guia de boas práticas para escrever código mais manutenível, flexível e desacoplado.

Origem e História: O termo SOLID foi popularizado no início dos anos 2000 pelo engenheiro de software **Robert C. Martin** (também conhecido como Uncle Bob). A sigla foi organizada por Michael Feathers, que pegou os cinco princípios defendidos por Robert C. Martin e montou o acrônimo SOLID.

Esses princípios surgiram como uma evolução dos conceitos de design orientado a objetos que vinham sendo discutidos desde os anos 80 e 90 (por exemplo, Barbara Liskov já havia formulado seu princípio em 1987).

O SOLID busca:

- Melhorar a **manutenibilidade** do código.
- Facilitar **extensão sem quebrar código existente**.
- Promover **baixo acoplamento** e **alta coesão**.
- Ajudar no desenvolvimento de **sistemas mais robustos e escaláveis**.

SOLID é um acrônimo que representa cinco princípios da programação orientada a objetos:

S : Single Responsibility Principle (Princípio da Responsabilidade Única): SRP;

O : Open/Closed Principle (Princípio do Aberto/Fechado): OCP;

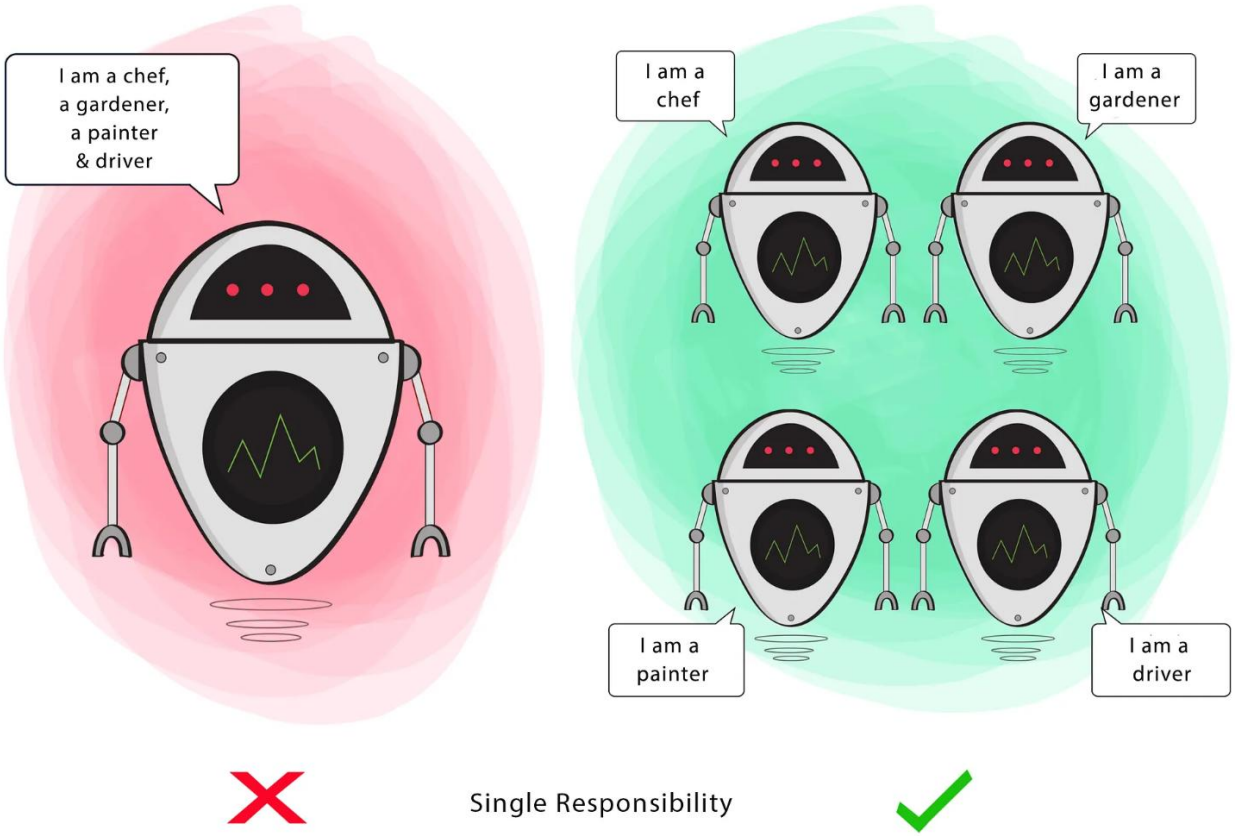
L : Liskov Substitution Principle (Princípio da Substituição de Liskov) : LSP;

I : Interface Segregation Principle (Princípio da Segregação de Interfaces): ISP;

D : Dependency Inversion Principle (Princípio da Inversão de Dependência): DIP.

S – Single Responsibility Principle (Princípio da Responsabilidade Única)

Uma classe deve ter apenas uma razão para mudar, ou seja, ser responsável por apenas uma parte do sistema.



Ou seja, cada classe deve ser **responsável por apenas uma coisa** dentro do sistema. Se ela cuida de mais de um assunto, fica difícil dar manutenção, testar e reaproveitar.

Imagine que você é dono de um restaurante:

- O **cozinheiro** cozinha.
- O **garçom** serve.
- O **caixa** recebe o pagamento.

Se o cozinheiro fizesse tudo, o restaurante não funcionaria direito. No código é igual: cada classe deve ter **sua função clara e única**.

Meta

Este princípio visa separar comportamentos para que, se surgirem bugs como resultado da sua mudança, eles não afetem outros comportamentos não relacionados.

Observe o código a seguir:

```
public class Relatorio {

    public void gerarRelatorio() {

        // lógica para gerar os dados do relatório

        System.out.println("Relatório gerado!");

    }

    public void salvarEmArquivo(String conteudo) {

        // lógica para salvar o relatório em arquivo

        System.out.println("Relatório salvo em arquivo!");

    }

    public void enviarPorEmail(String conteudo) {

        // lógica para enviar o relatório por email

        System.out.println("Relatório enviado por e-mail!");

    }

}
```

O código viola o **Princípio da Responsabilidade Única** porque a classe Relatorio possui **mais de uma responsabilidade**:

- gerar relatório,
- salvar em arquivo,
- enviar por e-mail.

Ou seja, a classe teria **mais de uma razão para mudar**, o que contraria o princípio da responsabilidade única.

Para corrigir o código, então separamos as responsabilidades em classes distintas:

```
// Classe responsável apenas por gerar relatórios

public class Relatorio {

    public String gerarRelatorio() {

        return "Relatório gerado!";

    } }
```

```
// Classe responsável por salvar relatórios

public class RelatorioArquivo {

    public void salvar(String conteudo) {

        System.out.println("Relatório salvo em arquivo: " + conteudo);

    }

}
```

```
// Classe responsável por enviar relatórios

public class RelatorioEmail {

    public void enviar(String conteudo) {

        System.out.println("Relatório enviado por e-mail: " + conteudo);

    }

}
```

Perceba que as nomenclaturas das classes foi desgeneralizada, agregando a si a sua ideia de existir com a sua responsabilidade.

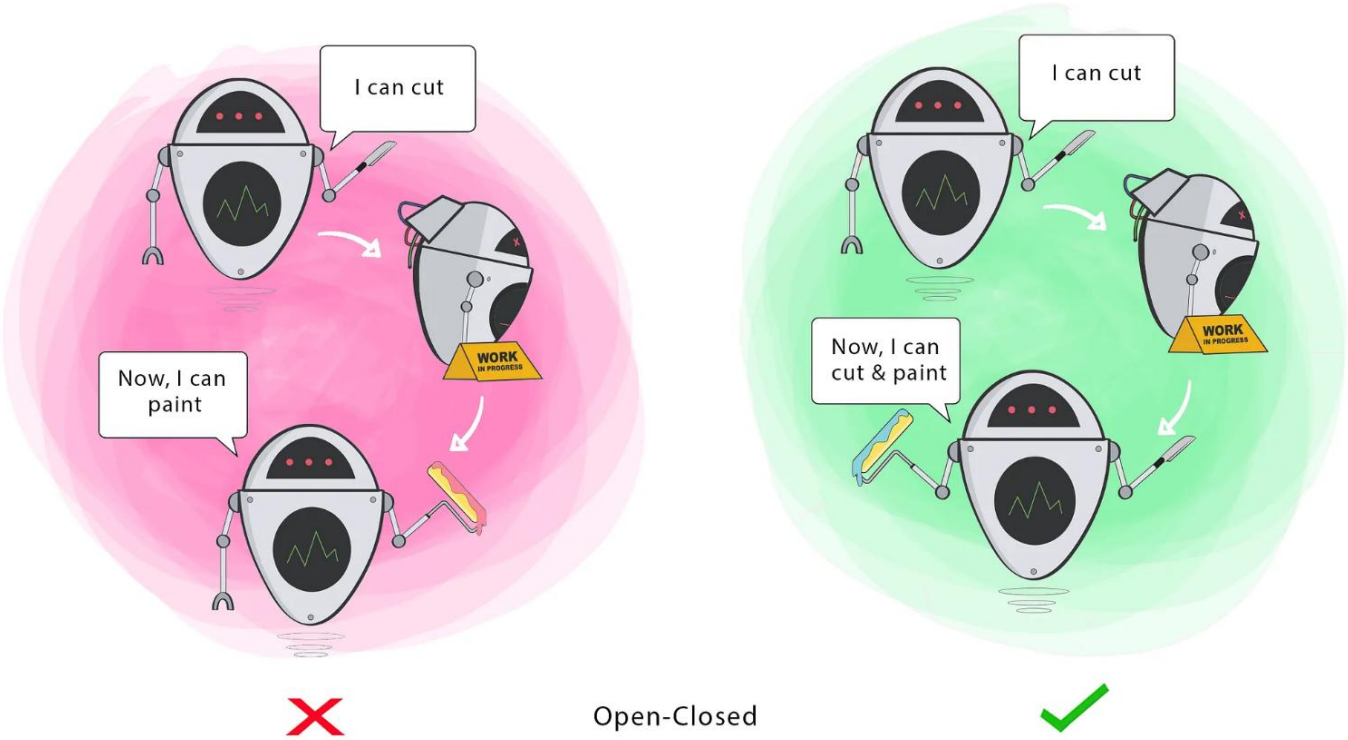
Num outro exemplo, sistema de biblioteca, a aplicação de Single Responsibility Principle (SRP): Cada classe tem **apenas uma responsabilidade**:

- Pessoa : Representar o usuário da biblioteca.
- Livro : Representar um livro.
- Emprestimo : Gerenciar a regra de negócio e os dados de um empréstimo.
- Biblioteca : Gerenciar cadastros e empréstimos.

Violações nítidas: Desenvolver o empréstimo dentro das classes Pessoa ou Livro. Pessoa passaria a representar apenas um usuário, pois teria em si os seus respectivos empréstimos. O mesmo aconteceria para a classe Livros se os empréstimos ficarem agregados a ela. Se não isolar a classe Emprestimos, colocando em Biblioteca, por exemplo, a biblioteca começa a acumular responsabilidades: além dos cadastros, as regras de negócio dos empréstimos. Sendo então o acúmulo de responsabilidades ruim pra manutenção de código, fazendo com que talvez tenha que alterar a classe inteira.

SRP violado : quando uma classe faz mais de uma coisa.

O software deve estar aberto para extensão, mas fechado para modificação. Em vez de alterar código existente, adiciona-se novas funcionalidades através de abstrações.



O **Open/Closed Principle (OCP)** afirma que *"entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação"*.

Isso significa que devemos poder **adicionar novos comportamentos** ao sistema sem precisar **alterar o código existente**, reduzindo riscos de regressões e mantendo a estabilidade do software.

O **Open/Closed Principle (OCP)** afirma que *"entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação"*.

A ideia é garantir **flexibilidade** e **manutenibilidade** no código. Facilitar a adição de novas funcionalidades sem comprometer funcionalidades já implementadas e, minimizar alterações em classes consolidadas e testadas.

A aplicação é realizada através de **Herança** ou **implementação de interfaces** para estender comportamentos ou **inversão de dependências** e **injeção de dependências** para permitir a substituição de componentes.

Os Padrões de projeto como **Strategy, Template Method, Decorator** que permitem adicionar comportamentos sem modificar código existente seguem este príncipio.

Um exemplo de implementação Java

Dado o problema, um exemplo de violação de OCP:

```
class Calculadora {

    public double calcular(String operacao, double a, double b) {

        if (operacao.equals("soma")) {

            return a + b;

        } else if (operacao.equals("subtracao")) {

            return a - b;

        } else if (operacao.equals("multiplicacao")) {

            return a * b;

        } else if (operacao.equals("divisao")) {

            return a / b;

        }

        return 0;

    }

}
```

A solução está em separarmos o comportamento da operação:

```
interface Operacao {

    double executar(double a, double b);

}
```

```
class Soma implements Operacao {

    public double executar(double a, double b) {

        return a + b;

    }

}
```

```
class Subtracao implements Operacao {

    public double executar(double a, double b) {

        return a - b;

    }

}
```

```
class Calculadora {

    public double calcular(Operacao operacao, double a, double b) {

        return operacao.executar(a, b);

    }

}
```

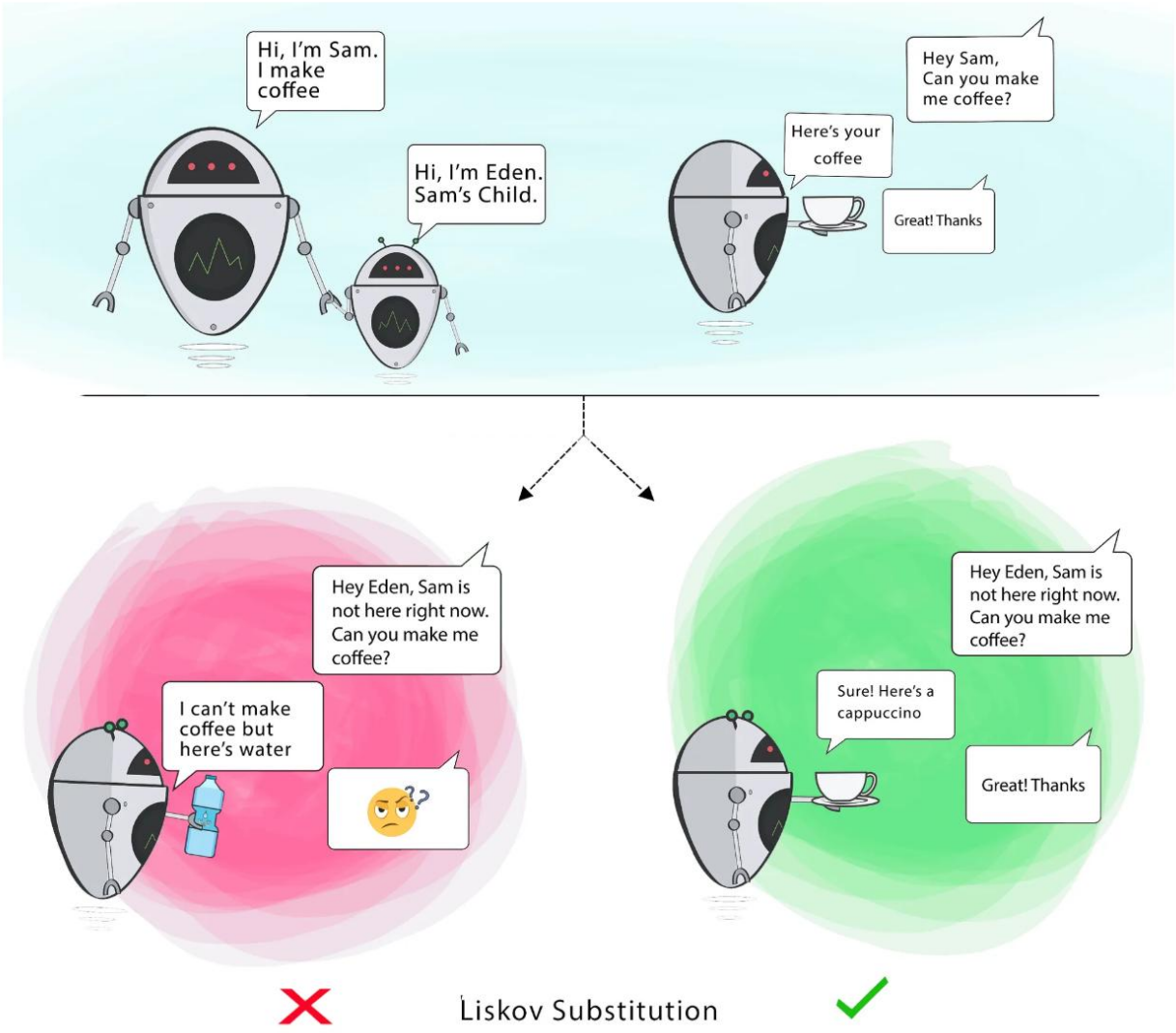
A Calculadora **não foi modificada** em nenhum momento. Apenas **criamos novas classes** (Multiplicacao e Divisao). Isso mostra na prática o **Open/Closed Principle**: o sistema está **aberto para extensão** e **fechado para modificação**.

```
public class App {

    public static void main(String[] args) {

        Calculadora calculadora = new Calculadora();

        // Operações básicas

        Operacao soma = new Soma();

        Operacao subtracao = new Subtracao();


        double resultado1 = calculadora.calcular(soma, 10, 5);

        double resultado2 = calculadora.calcular(subtracao, 10, 5);


        System.out.println("Resultado da Soma: " + resultado1);

        System.out.println("Resultado da Subtração: " + resultado2);

    }

}
```

Formulado por **Barbara Liskov** em 1987, o princípio afirma: "Se **S** é subtipo de **T**, então os objetos do tipo **T** devem poder ser substituídos por objetos do tipo **S** sem alterar as propriedades desejáveis do programa."

Objetos de uma classe derivada devem poder substituir objetos da classe base sem alterar o funcionamento do sistema.

Ou seja: subclasses devem **preservar o comportamento esperado** da superclasse.



O objetivo é garantir que a **herança** mantenha a **coerência semântica**. Evitar que uma subclasse **viole expectativas** do contrato estabelecido pela superclasse e promover **reuso seguro** e confiável.

A seguir um problema, violando o Liskov.

```
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;
    }
}
```

```
class Quadrado extends Retangulo {
    @Override
    public void setLargura(int largura) {
        this.largura = largura;
        this.altura = largura; // força os dois lados iguais
    }
    @Override
    public void setAltura(int altura) {
        this.largura = altura;
        this.altura = altura; // idem
    }
}
```

Um Quadrado é um Retangulo matematicamente, mas ao sobrescrever setters, quebra a expectativa de que largura e altura podem ser ajustadas independentemente. Isso pode gerar comportamentos inesperados em métodos que esperam trabalhar com Retangulo.

Corrigindo o problema:

```
interface Forma {
    int getArea();
}
```

```
class Retangulo implements Forma {
    private int largura;
    private int altura;
    public Retangulo(int largura, int altura) {
        this.largura = largura;
        this.altura = altura;
    }
    @Override
    public int getArea() {
        return largura * altura;
    }
}
```

```
class Quadrado implements Forma {
    private int lado;
    public Quadrado(int lado) {
        this.lado = lado;
    }
    @Override
    public int getArea() {
        return lado * lado;
    }
}
```

Agora Quadrado e Retangulo não têm relação de herança direta. Ambos implementam a mesma abstração (Forma), garantindo que possam ser usados de forma intercambiável.

```
public class App {
    public static void main(String[] args) {
        //Referências na Interface
        Forma f1 = new Retangulo(10, 5);
        Forma f2 = new Quadrado(7);

        System.out.println("Área do Retângulo: " + f1.getArea());
        System.out.println("Área do Quadrado: " + f2.getArea());

        // O uso é consistente: qualquer "Forma" funciona corretamente
        exibirArea(f1);
        exibirArea(f2);
    }
    public static void exibirArea(Forma forma) {
        System.out.println("Área calculada: " + forma.getArea());
    }
}
```

As vantagens do princípio são:

- Evitar violações semânticas em hierarquias de herança.
- Aumentar a confiabilidade do sistema.
- Incentivar uso de interfaces e composição (injeção) ao invés de herança inadequada.

Violando LSP: Quadrado herdando de Retangulo e mudando regras.

Respeitando LSP: ambos compartilham uma abstração comum (Forma). Isso mostra que herança mal utilizada pode quebrar contratos, e que muitas vezes a interface é mais adequada que herança.

Este princípio está associado aos seguintes padrões:

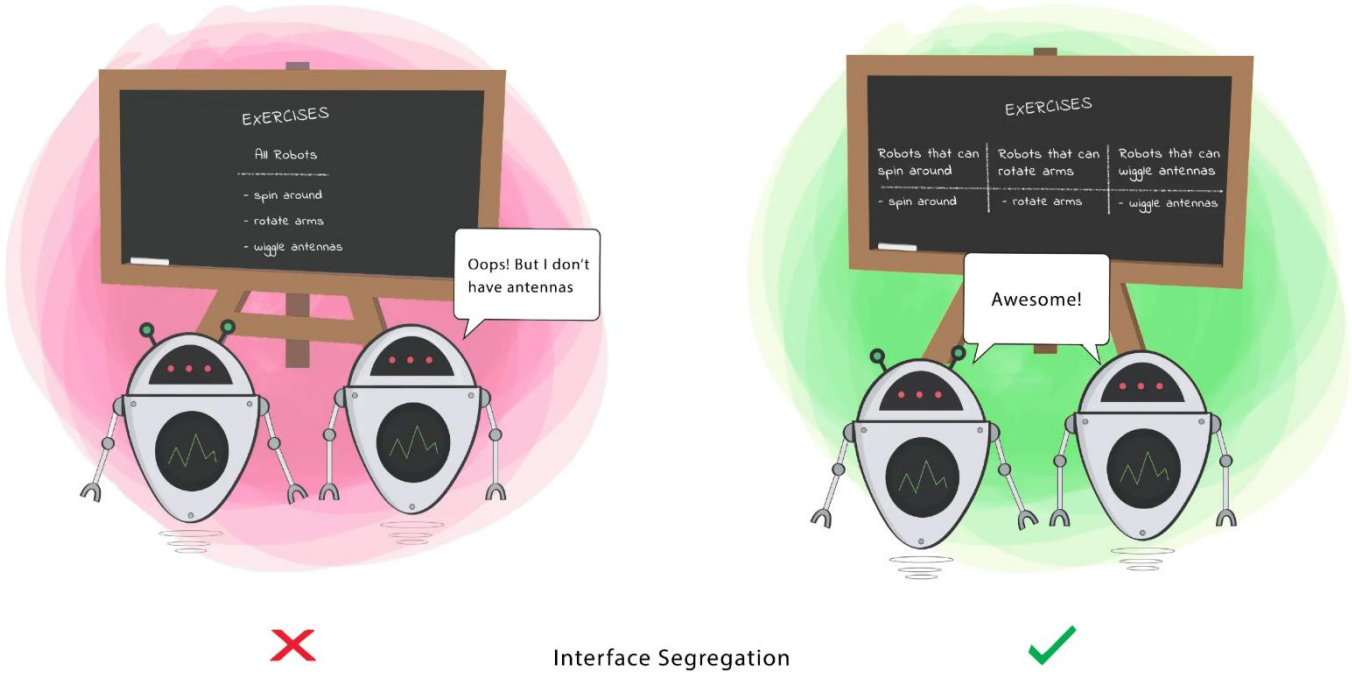
- **Template Method:** subclasses estendem mas seguem o mesmo contrato;
- **Strategy :** diferentes estratégias podem ser substituídas sem quebrar o cliente;
- **State:** cada estado substitui outro de forma consistente.

I – Interface Segregation Principle (Princípio da Segregação de Interfaces)

É melhor criar várias interfaces específicas e coesas, do que uma única interface "inchada" que obriga classes a implementar métodos que não usam.

"Nenhum cliente deve ser forçado a depender de métodos que não utiliza."

Ou seja, é melhor termos **interfaces específicas e coesas**, ao invés de uma única interface grande ("inchada") que obriga classes a implementar métodos desnecessários.



O objetivo está em evitar interfaces gordas (com métodos que não fazem sentido para todas as implementações). Promover coesão e baixo acoplamento, além de garantir que cada classe implemente apenas o que realmente precisa.

Um exemplo de problema, violando o princípio:

```
interface Trabalhador {

    void trabalhar();

    void comer();

}
```

```
class Operario implements Trabalhador {

    @Override

    public void trabalhar() {

        System.out.println("Operário está trabalhando...");

    }


    @Override

    public void comer() {

        System.out.println("Operário está comendo...");

    }

}
```

```
class Robo implements Trabalhador {

    @Override

    public void trabalhar() {

        System.out.println("Robô está trabalhando...");

    }


    @Override

    public void comer() {

        // Problema: um robô não come!

        throw new UnsupportedOperationException

                        ("Robô não precisa comer!");

    }

}
```

Violando ISP: Robo foi forçado a implementar comer(), mesmo não precisando.

A solução é quebrar a interface, já que interfaces podem ser multiplamente implementadas por classes.

```
interface Trabalhador {

    void trabalhar();

}
```

```
interface Alimentavel {

    void comer();

}
```

```
class Operario implements Trabalhador, Alimentavel {

...

}
```

```
class Robo implements Trabalhador {

     ...

}
```

As vantagens de criação de interfaces menores e mais focadas é que classes não são obrigadas a implementar métodos irrelevantes. O Código mais limpo e de fácil manutenção.

Interfaces foram segregadas (Trabalhador e Alimentavel), e cada classe implementa apenas o que faz sentido. O princípio ensina a **não criar interfaces genéricas demais**. Este princípio se relaciona com os padrões:

- **Adapter:** cria interfaces específicas para o cliente;
- **Facade:** fornece interfaces especializadas e reduz complexidade;
- **Proxy:** implementa apenas o que o cliente precisa, sem expor métodos inúteis.

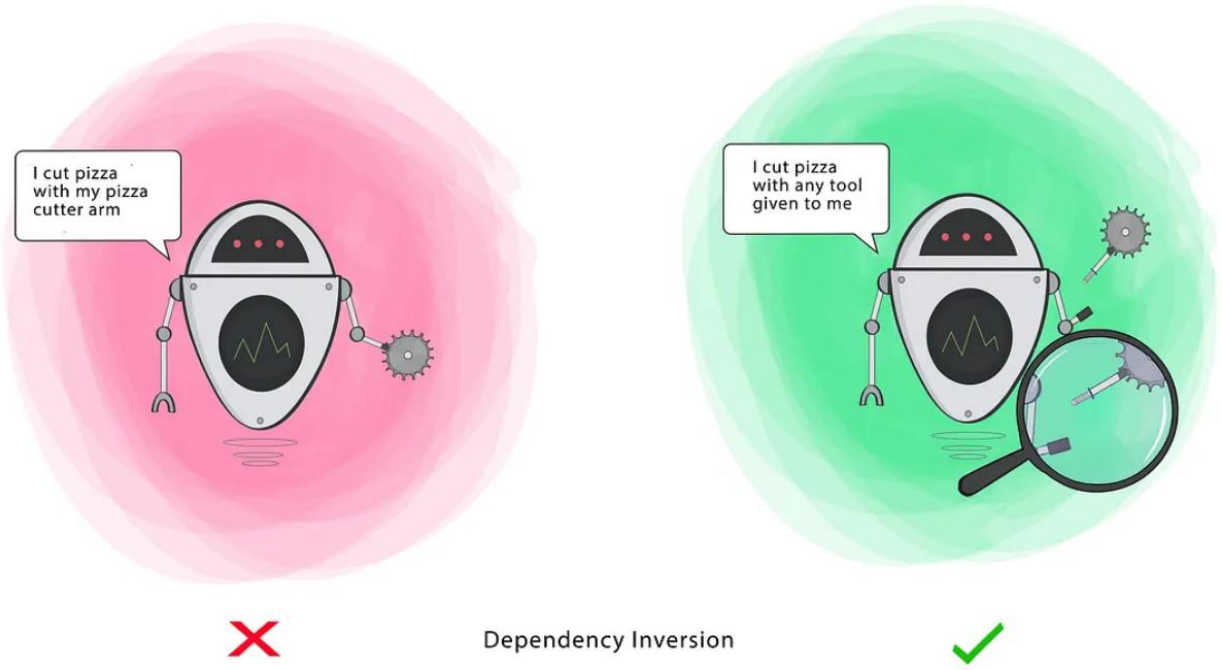
ISP está ligado à ideia de "interfaces voltadas ao cliente": cada padrão foca em entregar apenas o que realmente faz sentido.

D – Dependency Inversion Principle (Princípio da Inversão de Dependência)

Dependa de abstrações e não de implementações concretas. Isso reduz o acoplamento entre módulos.

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.

Ou seja: em vez de classes concretas dependerem diretamente umas das outras, devem depender de interfaces ou classes abstratas.



O objetivo é reduzir acoplamento entre módulos, facilitar testes unitários e substituição de implementações e tornar o sistema mais flexível e manutenível.

Criando o problema, violação de DIP:

No exemplo de Biblioteca, criamos um repositório de empréstimos, onde a persistência de dados será para formato arquivos. Caso haja uma mudança de formato de persistência, será necessário alterar a classe Biblioteca.

```
// Repositório concreto

class RepositorioArquivo {

    public void salvarEmprestimo(Emprestimo emprestimo) {

        System.out.println("Salvando empréstimo em arquivo: "

                           + emprestimo);

    }

}
```

```
class Emprestimo {

    private String aluno;

    private String livro;


    public Emprestimo(String aluno, String livro) {

        this.aluno = aluno;

        this.livro = livro;

    }


    @Override

    public String toString() {

        return "Aluno: " + aluno + ", Livro: " + livro;

    }

}
```

```
// Biblioteca ACOPLADA a RepositorioArquivo

class Biblioteca {

    private RepositorioArquivo repositorio = new RepositorioArquivo();


    public void registrarEmprestimo(String aluno, String livro) {

        Emprestimo e = new Emprestimo(aluno, livro);

        repositorio.salvarEmprestimo(e);

    }

}
```

Violação de DIP: Biblioteca depende de um detalhe (RepositorioArquivo). Se mudar a forma de salvar, preciso alterar o código da Biblioteca.

Ao criamos uma abstração (interface Repositorio), a Biblioteca depende da abstração, e não da implementação concreta.

```
// Abstração

interface Repositorio {

    void salvarEmprestimo(Emprestimo emprestimo);

}
```

```
// Implementação 1 - arquivo

class RepositorioArquivo implements Repositorio {

    public void salvarEmprestimo(Emprestimo emprestimo) {

        System.out.println("Salvando empréstimo em arquivo: "

                           + emprestimo);

    }

}
```

```
// Implementação 2 - banco de dados

class RepositorioBanco implements Repositorio {

    public void salvarEmprestimo(Emprestimo emprestimo) {

        System.out.println("Salvando empréstimo no banco: " + emprestimo);

    }

}
```

```
class Emprestimo {

    private String aluno;

    private String livro;


    public Emprestimo(String aluno, String livro) {

        this.aluno = aluno;

        this.livro = livro;

    }


    @Override

    public String toString() {

        return "Aluno: " + aluno + ", Livro: " + livro;

    }

}
```

```
// Biblioteca depende da abstração

class Biblioteca {

    private Repositorio repositorio;


    //Injeção do repositório

    public Biblioteca(Repositorio repositorio) {

        this.repositorio = repositorio;

    }

    public void registrarEmprestimo(String aluno, String livro) {

        Emprestimo e = new Emprestimo(aluno, livro);

        repositorio.salvarEmprestimo(e);

    }

}
```

```
public class App {

    public static void main(String[] args) {

        Repositorio repoArquivo = new RepositorioArquivo();

        Repositorio repoBanco = new RepositorioBanco();

        Biblioteca biblioteca1 = new Biblioteca(repoArquivo);

        biblioteca1.registrarEmprestimo("Maria", "O Senhor dos Anéis");

        Biblioteca biblioteca2 = new Biblioteca(repoBanco);

        biblioteca2.registrarEmprestimo("João", "Clean Code");

    }

}
```

Biblioteca só conhece a interface Repositorio. O detalhe (salvar em arquivo ou em banco) depende da abstração, não o contrário.

DIP é a base de vários padrões que promovem baixo acoplamento e injeção de abstrações.

- **Strategy:** o cliente depende de uma interface, e não da estratégia concreta;
- **Observer:** os observadores dependem da abstração do "subject";
- **Abstract Factory:** cria objetos a partir de abstrações, desacoplando cliente de implementações;
- **Dependency Injection** (IoC Container): prática moderna que operacionaliza o DIP.

Resumo final

S : foco e simplicidade;

O : extensão sem mexer no que já funciona;

L: herança segura e contratos consistentes;

I : interfaces específicas e coesas;

D : inversão: abstrações primeiro, detalhes depois.

Princípio	Ideia Central	Padrões Relacionados	Explicação Didática
SRP	Uma classe deve ter **apenas uma razão para mudar** (responsabilidade única).	**Facade:** separa subsistemas em interfaces específicas. **Adapter:** converte responsabilidades em classes distintas. **Decorator:** adiciona responsabilidades sem misturar as existentes.	Promove **alta coesão** e **baixa complexidade**: cada classe faz só o que deve.
OCP	Classes devem estar **abertas para extensão, mas fechadas para modificação**.	**Strategy:** permite adicionar novos algoritmos sem alterar o cliente. **Decorator:** estende comportamento dinamicamente. **Template Method:** define esqueleto fixo, mas permite extensões.	Permite **evoluir o sistema** criando novas extensões em vez de alterar código pronto/testado.
LSP	Subclasses devem poder **substituir** a superclasse sem quebrar o comportamento esperado.	**Strategy**: troca de algoritmos transparente. **Template Method:** garante consistência em herança. **State:** estados se substituem sem alterar uso.	Garante **substituição segura**: qualquer implementação deve manter o contrato da abstração.
ISP	Nenhum cliente deve ser forçado a depender de métodos que não usa.	**Adapter:** interfaces sob medida. **Facade:** interface simplificada e específica. **Proxy:** expõe apenas o necessário.	Evita **interfaces gordas**: divide contratos grandes em **interfaces pequenas e coesas**.
DIP	Depender de **abstrações** e não de implementações concretas.	**Strategy:** cliente usa abstração da estratégia. **Observer:** abstrações conectam sujeitos e observadores. **Abstract Factory:** cria objetos via abstração.**IoC / Dependency Injection** – aplica DIP na prática.	Garante **baixo acoplamento**: módulos de alto nível não conhecem detalhes, apenas abstrações.

Referências:

UGONNA, Thelma. *The S.O.L.I.D Principles in Pictures*. Medium, 2019. Disponível em: https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898. Acesso em: 17 set. 2025.

DESCHAMPS, Filipe. *SOLID fica FÁCIL com Essas Ilustrações*. YouTube, 5 fev. 2021. Disponível em: https://www.youtube.com/watch?v=6SfrO3D4dHM. Acesso em: 17 set. 2025.

Apoio e pesquisa, certificação e elaboração: chat GPT 5 em https://chatgpt.com/.

Códigos revisados e elaborados no VS Code em https://code.visualstudio.com/.

Java LTS 17.

Sugestões de continuidade e correlação de estudo

- Design Patterns (GoF);
- Domain-Driven Design;
- Padrões de programação funcional;
- Object Calisthenics;
- Clean Code.

Tempo de estudo e elaboração do material: 20 horas.