

Desmistificando os Princípios SOLID

O que é SOLID?

SOLID é um acrônimo para 5 princípios de design que nos ajudam a criar software:

- **C**ompreensível
- **F**lexível
- **F**ácil de manter

Pense neles como os pilares para construir um software de alta qualidade.



1. (S) Princípio da Responsabilidade Única

(Single Responsibility Principle)

“ Uma classe deve ter **um, e apenas um**, motivo para mudar. ”

SRP: Analogia




O Canivete Suíço vs. a Caixa de Ferramentas

- **Canivete Suíço:** Uma ferramenta, muitas funções
 - Se quebra uma parte → Perde tudo! 
- **Caixa de Ferramentas:** Cada ferramenta, uma função
 - Se quebra uma → As outras continuam funcionando! 

No código: Melhor ter classes separadas do que uma classe "faz-tudo".

SRP: O Problema

Diagnóstico: A classe `Funcionario` tem **3 responsabilidades!**

-  Gerenciar dados pessoais
-  Salvar no banco de dados
-  Gerar relatórios

Consequência: Se mudarmos o banco de dados, a classe `Funcionario` quebra!

SRP: Código Problemático


```
// NÃO FAÇA ISSO!  
class Funcionario {  
    private String nome;  
    private double salario;  
  
    // Responsabilidade 1: Gerenciar dados  
    public String getNome() { /* ... */ }  
    public void setNome(String nome) { /* ... */ }
```

SRP: Código Problemático (cont.)

```
// Responsabilidade 2: Persistência  
public void salvarNoBancoDeDados() {  
    // Lógica para salvar no banco...  
    // E se mudarmos de MySQL para PostgreSQL?  
}  
  
// Responsabilidade 3: Relatórios  
public void gerarRelatorioHoras() {  
    // Lógica para gerar relatório...  
    // E se mudarmos o formato do relatório?  
}  
}
```



SRP: A Solução

Estratégia: Uma classe = Uma responsabilidade

```
//  Focada APENAS nos dados
class Funcionario {
    private String nome;
    private double salario;

    public String getNome() { return nome; }
    // ... outros getters/setters
}
```


SRP: Classes Separadas

```
//  Focada APENAS em persistência  
class RepositorioFuncionario {  
    public void salvar(Funcionario funcionario) {  
        // Lógica do banco aqui  
    }  
}  
  
//  Focada APENAS em relatórios  
class ServicoDeRelatorio {  
    public void gerarRelatorio(Funcionario funcionario) {  
        // Lógica de relatório aqui  
    }  
}
```

SRP: Benefícios

Antes: 1 motivo para mudar = 3 classes quebradas ✨

Depois:

- Mudança no banco? → Só `RepositorioFuncionario`
- Novo formato de relatório? → Só `ServicoDeRelatorio`
- Novos dados pessoais? → Só `Funcionario`

Resultado: Código mais seguro e fácil de manter! ✅

2. (O) Princípio do Aberto/Fechado (Open/Closed Principle)

“ As classes devem estar **abertas para extensão**, mas **fechadas para modificação**. ”

OCP: Analogia

O Smartphone e os Apps

- **Sistema Operacional:** FECHADO para modificação
 - Você não mexe no iOS/Android principal
- **Funcionalidades:** ABERTAS para extensão
 - Instala novos apps sem quebrar o sistema

No código: Adicione novas funcionalidades sem alterar código existente!

OCP: O Problema (Antes)




Para cada novo tipo de contrato, precisamos **modificar** a classe `CalculadoraDeBonus`, adicionando mais `if/else`.

```
// NÃO FAÇA ISSO!  
class CalculadoraDeBonus {  
    public double calcular(Funcionario f, String tipoContrato) {  
        if (tipoContrato.equals("CLT")) {  
            return f.getSalario() * 0.1;  
        } else if (tipoContrato.equals("Estagio")) {  
            return f.getSalario() * 0.05;  
        }  
        // Cada novo contrato exige uma alteração aqui!  
        return 0;  
    }  
}
```

OCP: A Estratégia


Solução: Interface para extensão sem modificação! 🔧

Estratégia:

-  Criar interface `Contrato`
-  Cada tipo implementa a interface
-  Calculadora usa a interface (fechada!)


Resultado: Novos contratos = Novas classes, calculadora intacta!

OCP: Criando a Interface

```
//  Interface define o "contrato"  
interface Contrato {  
    double calcularBonus(double salario);  
}
```


Princípio: Defina o que deve ser feito, não como será feito.

OCP: Implementações Específicas

```
//  Cada contrato implementa sua lógica
class ContratoClt implements Contrato {
    public double calcularBonus(double salario) {
        return salario * 0.1; // 10% de bônus
    }
}

class ContratoEstagio implements Contrato {
    public double calcularBonus(double salario) {
        return salario * 0.05; // 5% de bônus
    }
}
```


OCP: Calculadora Fechada

```
//  Calculadora FECHADA para modificação  
class CalculadoraDeBonus {  
    public double calcular(double salario, Contrato contrato) {  
        return contrato.calcularBonus(salario);  
    }  
}
```

Vantagem: Nunca mais precisamos mexer nesta classe!

OCP: Extensão Fácil

Novo contrato PJ? Simples! 🚀

```
// ✅ Nova classe, zero modificações!  
class ContratoPj implements Contrato {  
    public double calcularBonus(double salario) {  
        return salario * 0.15; // 15% de bônus  
    }  
}
```

Resultado: Código extensível sem riscos de quebrar o existente!

3. (L) Princípio da Substituição de Liskov

(Liskov Substitution Principle)

“ Uma classe filha deve ser **substituível por sua classe mãe** sem quebrar o programa. ”

LSP: Analogia

O Pato de Borracha 🦆

- **Pato Real:** Parece pato, voa como pato, é pato! ✅
- **Pato de Borracha:** Parece pato, mas NÃO voa! ❌

Problema: Se seu código espera um `Pato` que voa, substituir por `PatoDeBorracha` vai quebrar!

No código: Classes filhas devem manter o "contrato" da classe mãe.

LSP: O Problema


Situação: `Quadrado` herda de `Retangulo` mas quebra expectativas! ✨

Problema: Herança parece fazer sentido matematicamente...

- Quadrado É UM tipo de retângulo ✓
- Mas no código quebra o comportamento esperado! ✗

Resultado: Substituição gera bugs inesperados!

LSP: Classe Retângulo

```
//  Classe Retângulo "normal"
class Retangulo {
    protected int altura, largura;

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

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

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

LSP: Quadrado Problemático

```
// X Quadrado herda mas "força" comportamento
class Quadrado extends Retangulo {
    @Override
    public void setAltura(int altura) {
        this.altura = altura;
        this.largura = altura; // Força igualdade!
    }

    @Override
    public void setLargura(int largura) {
        this.altura = largura; // Força igualdade!
        this.largura = largura;
    }
}
```


Problema: Muda o comportamento esperado da classe pai!

LSP: O Bug em Ação



```
// 🐛 Teste que quebra!  
Retangulo r = new Quadrado(); // Parece OK...  
  
r.setAltura(10); // altura = 10, largura = 10  
r.setLargura(5); // altura = 5, largura = 5  
  
int area = r.getArea();  
// Esperado: 10 × 5 = 50  
// Real: 5 × 5 = 25  
// COMPORTAMENTO QUEBRADO! 💣
```

Problema: Código que funciona com `Retangulo` falha com `Quadrado` !

LSP: A Estratégia


Solução: Evitar herança problemática! 

Estratégia:

-  Não usar `Quadrado extends Retangulo`
-  Usar abstração comum (interface)


Resultado: Sem substituições que quebram o contrato!

LSP: Criando a Abstração


```
//  Interface comum para todas as formas  
interface Forma {  
    int getArea();  
}
```

Princípio: Defina o que todas as formas devem fazer.

LSP: Implementações Corretas

```
//  Retângulo implementa a interface  
class Retangulo implements Forma {  
    private int altura, largura;  
  
    public Retangulo(int altura, int largura) {  
        this.altura = altura;  
        this.largura = largura;  
    }  
  
    public int getArea() { return altura * largura; }  
}
```

LSP: Quadrado Independente

```
//  Quadrado também implementa a interface  
class Quadrado implements Forma {  
    private int lado;  
  
    public Quadrado(int lado) {  
        this.lado = lado;  
    }  
  
    public int getArea() { return lado * lado; }  
}
```

Vantagem: Cada classe tem sua própria lógica, sem conflitos!

LSP: Benefícios

Antes: `Quadrado extends Retangulo` = Comportamento quebrado 🌟

Depois: Ambos implementam `Forma` = Comportamento consistente
✓

```
// ✓ Ambos funcionam corretamente
Forma f1 = new Retangulo(10, 5); // Área = 50
Forma f2 = new Quadrado(5);      // Área = 25
```

Resultado: Substituição segura, sem surpresas!

4. (I) Princípio da Segregação de Interfaces (Interface Segregation Principle)

“ Clientes não devem ser forçados a depender de **métodos que não usam**. ”

ISP: Analogia

O Restaurante com Menus Separados 🍴

- **Menu Gigante:** Comidas + Bebidas + Sobremesas (confuso!)
- **Menus Separados:**
 - Menu de comidas 🍕
 - Menu de bebidas 🥤
 - Menu de sobremesas 🍰

Vantagem: Cliente pega apenas o que interessa!

No código: Interfaces pequenas e específicas!



ISP: O Problema

Situação: Interface "faz-tudo" força implementações desnecessárias!

```
//  Interface muito "gorda"  
interface Trabalhador {  
    void trabalhar(); //  Faz sentido para todos  
    void comer();    //  Robôs não comem!  
}
```

Resultado: Classes implementam métodos que não usam.

ISP: Código Problemático

```
class Humano implements Trabalhador {  
    @Override  
    public void trabalhar() { /*  OK */ }  
  
    @Override  
    public void comer() { /*  OK */ }  
}
```

Humano: Sem problemas! Pode trabalhar E comer.


ISP: O Dilema do Robô


```
class Robo implements Trabalhador {  
    @Override  
    public void trabalhar() {  
        /* ✅ Perfeito! Robôs trabalham */  
    }  
  
    @Override  
    public void comer() {  
        // 🤔 E agora? Robôs não comem!  
        throw new UnsupportedOperationException();  
    }  
}
```

Problema: Forçado a implementar algo inútil!

ISP: A Solução



Estratégia: Interfaces pequenas e específicas!

```
//  Interface focada em trabalho
interface Trabalhavel {
    void trabalhar();
}

//  Interface focada em alimentação
interface Comivel {
    void comer();
}
```

Princípio: Cada interface = Uma responsabilidade

ISP: Implementações Limpas

```
//  Humano: Trabalha E come  
class Humano implements Trabalhavel, Comivel {  
    public void trabalhar() { /* ... */ }  
    public void comer() { /* ... */ }  
}  
  
//  Robô: Só trabalha (sem métodos inúteis!)  
class Robo implements Trabalhavel {  
    public void trabalhar() { /* ... */ }  
    // Não precisa implementar comer()!  
}
```

ISP: Benefícios

Antes: Robô obrigado a implementar `comer()` 🤖❌

Depois:

- Humano → `Trabalhave1` + `Comivel` 🧑✅
- Robô → Apenas `Trabalhave1` 🤖✅
- Código limpo, sem gambiarras!



Regra de ouro: Interface pequena = Menos dependências = Menos problemas!

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

“ Classes de alto nível não devem depender de classes de baixo nível. **Ambas devem depender de abstrações (interfaces).** ”

DIP: Analogia

O Controle Remoto e a TV

-  **Dependência Direta:** Controle "Samsung" só funciona com TV Samsung
-  **Dependência de Abstração:** Controle universal funciona com qualquer TV




Como? O controle depende da "ideia de TV" (botões padrão), não de uma marca específica.

No código: Dependenda de interfaces, não de classes concretas!

DIP: O Problema

Situação: `Interruptor` está "soldado" na `Lampada`! 

Diagnóstico: Acoplamento forte = Código inflexível

-  Funciona para lâmpadas
-  E se quisermos controlar uma ventoinha?
-  E se quisermos controlar um ar-condicionado?

Resultado: Cada novo dispositivo = Novo interruptor! 

DIP: Código Problemático

```
// X Classes fortemente acopladas  
class Lampada {  
    public void ligar() { /* Liga a lâmpada */ }  
    public void desligar() { /* Desliga a lâmpada */ }  
}
```

Problema: `Lampada` é uma classe concreta específica.

DIP: Interruptor Inflexível



```
// X Interruptor "soldado" na Lâmpada
class Interruptor {
    private Lampada lampada = new Lampada(); // Acoplamento!

    public void acionar() {
        // Só funciona com Lampada!
        if (/* alguma lógica */) {
            lampada.ligar();
        } else {
            lampada.desligar();
        }
    }
}
```

Limitação: Para cada dispositivo novo, precisa de um interruptor


DIP: A Estratégia

Inversão de Dependência: Inverta quem depende de quem!

-  **Antes:** Interruptor → Lampada (classe concreta)
-  **Depois:** Interruptor → Dispositivo (interface)


Resultado: Interruptor não conhece detalhes, só o "contrato"!

DIP: Criando a Abstração


```
//  Interface define o "contrato"  
interface Dispositivo {  
    void ligar();  
    void desligar();  
}
```

Princípio: Defina o que precisa ser feito, não como será feito.

DIP: Implementações Concretas


```
//  Cada dispositivo implementa o contrato  
class Lampada implements Dispositivo {  
    public void ligar() { /* Liga a lâmpada */ }  
    public void desligar() { /* Desliga a lâmpada */ }  
}  
  
class Ventoinha implements Dispositivo {  
    public void ligar() { /* Liga a ventoinha */ }  
    public void desligar() { /* Desliga a ventoinha */ }  
}
```

DIP: Interruptor Flexível

```
//  Interruptor depende da ABSTRAÇÃO  
class Interruptor {  
    private Dispositivo dispositivo;  
  
    // A dependência é injetada de fora!  
    public Interruptor(Dispositivo dispositivo) {  
        this.dispositivo = dispositivo;  
    }  
  
    public void acionar() {  
        // Funciona com QUALQUER dispositivo!  
    }  
}
```

DIP: Benefícios

Flexibilidade Total! 

```
//  Funciona com qualquer dispositivo  
Interruptor int1 = new Interruptor(new Lampada());  
Interruptor int2 = new Interruptor(new Ventoinha());  
Interruptor int3 = new Interruptor(new ArCondicionado());
```

Antes: 1 interruptor = 1 dispositivo fixo

Depois: 1 interruptor = infinitos dispositivos possíveis!

Como os Princípios se Relacionam

- **SRP** e **ISP** promovem **alta coesão**, evitando classes e interfaces "inchadas".
- **OCP** é o objetivo final: código extensível.
- **DIP** e **LSP** são os mecanismos principais para alcançar o OCP.
 - Usamos **Inversão de Dependência (DIP)** para depender de abstrações.
 - Garantimos que as implementações dessas abstrações sejam substituíveis com **Liskov (LSP)**.

Atividade Prática

Objetivo: Fixar os conceitos SOLID através de análise de código!

Tempo: 15-20 minutos

Formato: Individual ou em duplas

Cenário: Sistema de Biblioteca

Você herdou o código de um sistema de biblioteca. Analise cada classe e identifique qual(is) princípio(s) SOLID está(ão) sendo violado(s):

Questão 1: Análise da Responsabilidade Única

```
class Livro {  
    private String titulo, autor;  
    private boolean disponivel;  
  
    // Getters e setters...  
  
    public void salvarNoBanco() { /* SQL aqui */ }  
    public void enviarEmailDisponibilidade() { /* Email */ }  
    public String gerarRelatorioEmprestimo() { /* PDF */ }  
}
```

Pergunta: Quantas responsabilidades esta classe tem? Quais?

Questão 2: Análise do Aberto/Fechado

```
class CalculadoraMulta {  
    public double calcular(String tipoUsuario, int diasAtraso) {  
        if (tipoUsuario.equals("ESTUDANTE")) {  
            return diasAtraso * 0.5;  
        } else if (tipoUsuario.equals("PROFESSOR")) {  
            return diasAtraso * 0.3;  
        } else if (tipoUsuario.equals("FUNCIONARIO")) {  
            return diasAtraso * 0.8;  
        }  
        return diasAtraso * 1.0; // Visitante  
    }  
}
```

Pergunta: Como adicionar um novo tipo "PESQUISADOR" sem modificar esta classe?

Questão 3: Análise da Segregação de Interfaces

```
interface Usuario {  
    void emprestar();  
    void devolver();  
    void renovar();  
    void acessarSalaEstudo();    // Só professores/estudantes  
    void acessarLaboratorio();  // Só pesquisadores  
    void editarCatalogo();      // Só bibliotecários  
}
```

Pergunta: Um "Visitante" deveria implementar todos esses métodos?

Questão 4: Análise da Inversão de Dependência

```
class ServicoEmprestimo {  
    private BancoDeDadosMySQL banco; // Dependência direta!  
  
    public ServicoEmprestimo() {  
        this.banco = new BancoDeDadosMySQL();  
    }  
  
    public void registrarEmprestimo(Emprestimo emp) {  
        banco.salvar(emp);  
    }  
}
```

Pergunta: E se quisermos trocar para PostgreSQL?

Desafio Bonus

Refatore o código da **Questão 1** aplicando os princípios SOLID:

1. **Responsabilidade Única:** Separe as responsabilidades
2. **Aberto/Fechado:** Permita extensão futura
3. **Inversão de Dependência:** Use abstrações

Tempo: 10 minutos extras

Respostas Esperadas

Q1: 4 responsabilidades - dados, persistência, notificação, relatórios

Q2: Criar interface `TipoUsuario` com implementações específicas

Q3: Não! Interfaces menores e específicas

Q4: Usar interface `RepositorioEmprestimo`

Conclusão

Adotar os princípios SOLID não é sobre seguir regras cegamente, mas sobre pensar em como construir um software **mais simples, mais flexível e mais fácil de manter** a longo prazo.

Os princípios trabalham juntos para criar código de qualidade! 🚀

Obrigado!