Aluno: Kayke Oliveira dos Santos RA: 24102484

Os princípios SOLID foram criados para resolver problemas
ocorridos em projetos orientados a objetos, como
- código rígido (difícil de modificar),
- código frágil (quebra facilmente),
- código não reutilizável.

Os 5 Princípios SOLID

SRP (Princípio da responsabilidade única)
- Uma classe deve ter apenas um motivo para mudar.

- OCP (Princípio Aberto/Fechado)
- Classes devem ser extensíveis sem modificação direta.

- LSP (Princípio da Substituição de Liskov)
Subclasses devem poder substituir suas classes base sem causar erros

- ISP (Princípio da Segregação de Interface)
- Interfaces devem ser específicas, evitando métodos desnecessários

- DIP (Princípio da Inversão de Dependência)
- Dependa de abstrações (interfaces) classes abstratas).

- São os princípios de Uncle em Programação Orientada a

Solid em ação IV

A classe CalculoArea viola o SRP porque tem duas responsabilidades?

- Calcula a soma das Áreas (SumAreas()).
- Gera saída dos resultados (output()).
- Consequências:

- Se precisar mudar a saída, terei obrigado a alterar a classe, mesmo que o cálculo esteja correto.

- Solução Proposta (SRP Aplicado)

- Dividir as responsabilidades em duas classes distintas

- CalculateAreas (cálculo)

```
Class CalculateAreas {
Shape[] Shapes
double pu_Total = 0;

Public CalculateAreas(Shape[] sh) {
This.shapes = sh;
}

Public double SumAreas() {
sum Total = 0;
for (int i=0; i<shapes.length; i++) {
sum Total += Shapes[i].CalcArea();
}
return sum Total;
}
}
```

- Class OutputArea

```
double area;

public OutputAreas(double a) {
  This.areas = a;
}

Public void consoles() {
System.out.println("...");
}
Public void HTML() {
System.out.println("...");
}
```

. Princípio Chave (SRP)

- Uma classe deve ter apenas um motivo para mudar.

Para o código mais organizado, menos risco de bugs.

• OCP (Open/Closed Principle) - Resumo ultra-curto

- "Classes devem ser abertas para extensão (nova funcionalidade) mas fechadas para modificação (não mude o que já funciona!

• Ao herdar pode adicionar novas comportamentos sem modificar o código original.

• LSP (Princípio de Substituição de Liskov)

"Se algo funciona com o pai, deve funcionar com o filho

```
// • Regra Herança
// Funciona quando
// classe filha:
// • Faz tudo que o pai faz
// • Não faz nada que o pai faz
```

```
Abstract class Shape {} // pai
Class Rectangle extends Shape {} // filho 1
Class Square extends Shape {} // filho 2
```

Analogia:

Se um pássaro de borracha herda de "Pato", não pode:

• Não fazer "quake"

• Quebrar quando colocar na água

• Isso evita bugs

- ISP | Princípio de Segregação de Interfaces |

- Divida interfaces em pequenas partes específicas, não deve implementar o que não for usar.

```
interface Animal {
void comer();
void voar();
void nadar();
}
```

```
interface Comedor {
    void comer();
}
interface Voador {
    void voar();
}
```

Class Pinguim implementa Comedor, Voador.

- Fica mais organizado

- DIP - Princípio de Inversão de Dependência

" Seu código vive depende de contratos (interfaces), não de implementações específicas - assim, outros sistemas podem plugar sua própria implementação sem modificar seu código.

DIP = Depende de interfaces + injeta implementação = código que aceita plugins!

- Seu sistema vira um lego, pronto encaixa peça no código.

```
Interface ConexaoAPI {
    String enviaDados(String dados);

Class ProcessadorDados {
Private ConexaoAPI api;
    Public ProcessadorDados(ConexaoAPI api) {
    this.api = api;
```