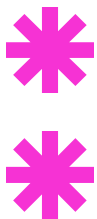




# SOLID

Francine Pereira, Luan Magarão, Lucas Hellmann  
e Matheus Henrique



# Sumário



## 01 História

O que é? Quem criou? Como?

## 02 Princípios

Cada uma das letras de "SOLID" e seus significados

## 03 Vantagens e Desvantagens

Os motivos bons e ruins de usá-lo ou não

## 04 Casos de Uso

Quem usou e quais resultados conseguiu

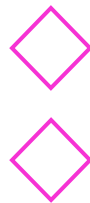
## 05 Aplicação

As maneiras de como usá-lo

## 06 Referências

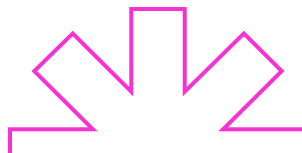
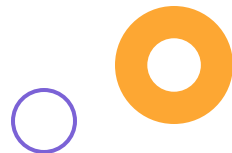
Fontes de Pesquisa





# 01

## História





# O que é?

O acrônimo SOLID representa os cinco princípios que facilitam o processo de desenvolvimento — o que facilita a manutenção e a expansão do software. Estes princípios são fundamentais na programação orientada a objetos e podem ser aplicados em qualquer linguagem que adote este paradigma

# Como?


A história de sua origem começa ao final da década de 1980 enquanto Robert C. Martin (conhecido como Uncle Bob, ou Tio Bob) discutia princípios de design de software com outros usuários da USENET (uma espécie de Facebook da época) com o objetivo de catalogar os mais importantes





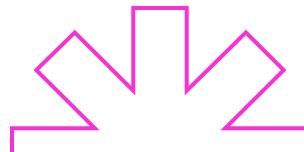
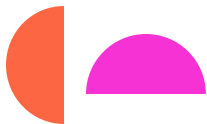
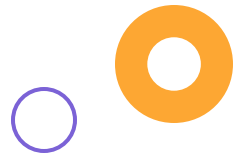
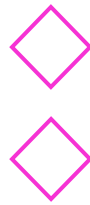
# Quem criou?

O primeiro indício dos princípios SOLID apareceu em 1995, no artigo “The principles of OoD” de Robert C. Martin, o “Uncle Bob”. Nos anos seguintes, Robert se dedicou a escrever mais sobre o tema, consolidando esses princípios de forma categórica. E, em 2002, lançou o livro “Agile Software Development, Principles, Patterns, and Practices” que reúne diversos artigos sobre o tema. No entanto, a sigla SOLID só foi apresentada mais tarde, por Michael Feathers



# 02

## Princípios



# Princípios SOLID

Os 5 princípios são:

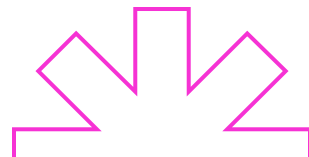
**S** — Single Responsibility Principle (Princípio da responsabilidade única)

**O** — Open-Closed Principle (Princípio Aberto-Fechado)

**L** — Liskov Substitution Principle (Princípio da substituição de Liskov)

**I** — Interface Segregation Principle (Princípio da Segregação da Interface)

**D** — Dependency Inversion Principle (Princípio da inversão da dependência)



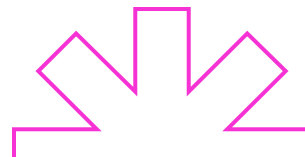
# Single Responsibility Principle (Princípio da responsabilidade única)

- **Definição:** Esse princípio declara que uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade dentro do software, ou seja, a classe deve ter uma única tarefa ou ação para executar.
- **Objetivo:** Evitar 'God Classes' que acumulam múltiplas responsabilidades e tornam o código difícil de manter e atualizar.
- **Benefício:** Facilita as alterações e a manutenção do código, pois cada classe tem um foco claro e definido.



# Open-Closed Principle (Princípio Aberto-Fechado)

- **Definição:** As classes da aplicação devem ser abertas para extensões e fechadas para modificações, ou seja, outras classes podem ter acesso ao que aquela classe possui, porém, não podem alterá-las.
- **Objetivo:** Flexibilidade do código, facilidade de manutenção e reaproveitamento do código evitando novas classes.
- **Benefícios:** Código existente protegido contra alterações, cada classe com um conjunto específico de funcionalidades e cada classe é testada separadamente.



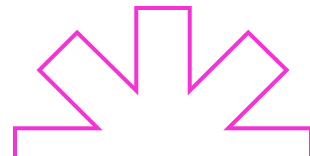
# Liskov Substitution Principle

## (Princípio da substituição de Liskov)

- **Definição:** Criado por Barbara Liskov e Jeannette Wing em 1993, o LSP define uma abordagem específica para subtipos na programação orientada a objetos.
- Subtipos devem ser substituíveis por seus tipos base sem alterar o comportamento correto do programa.
- **Objetivo:** Facilitar a extensibilidade e manutenção do software, permitindo que as subclasses ampliem as superclasses sem introduzir erros.
- **Benefícios:** Promove a reutilização de código e reduz a necessidade de alterações no código quando novas subclasses são adicionadas, mantendo a integridade do sistema.

# Interface Segregation Principle (Princípio da Segregação da Interface)

- **Descrição:** Interfaces extensas devem ser divididas em interfaces menores e mais específicas, de modo que cada cliente dependa apenas dos métodos que precisa utilizar.
- **Objetivo:** Evitar que classes sejam forçadas a implementar métodos que não utilizam.
- **Benefícios:** Código mais flexível e reutilizável, maior facilidade de manutenção, menor chance de erros, código mais fácil de testar e melhor qualidade geral do software.



# Dependency Inversion Principle (Princípio da inversão da dependência)

- **Definição:** O DIP sugere que o código de alto nível, que realiza funções importantes e complexas, não deve depender diretamente do código de baixo nível, que realiza tarefas mais simples e específicas. Em vez disso, ambos devem depender de interfaces abstratas, o que torna o sistema mais modular e fácil de modificar.
- **Objetivo:** Diminuir o acoplamento entre módulos de software, tornando-os mais reutilizáveis e flexíveis.
- **Benefícios:** Facilita a manutenção e a evolução do software, permitindo que mudanças em módulos de baixo nível não afetem módulos de alto nível.



# 03

## Vantagens e Desvantagens



# Vantagens

## Segurança

Mudanças em um local não afetam o sistema todo.

## Manutenção

Código mais modular facilita a compreensão e modificação.

## Reutilização

Evita reescrever código. Herda funcionalidades de classes existentes, promovendo flexibilidade.

## Testabilidade

Classes isoladas facilitam testes unitários, garantindo a qualidade do código.





# Desvantagens

## Curva de Aprendizizado

Dominar os princípios exige tempo e prática, especialmente para iniciantes em POO.

## Abstração Excessiva

Excesso de abstração pode dificultar a leitura e compreensão do código, especialmente para quem não está familiarizado com o projeto.

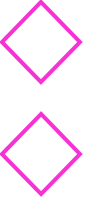
## Aumento do Código

A implementação dos princípios pode levar a um aumento inicial na quantidade de código, principalmente devido à criação de novas classes.

## Rigidez

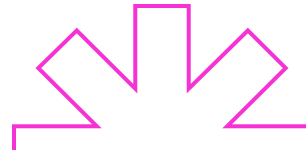
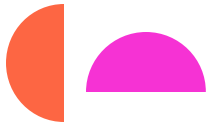
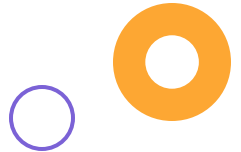
A rigidez dos princípios pode dificultar adaptações a mudanças inesperadas nos requisitos do projeto.






# 04

## Casos de Uso





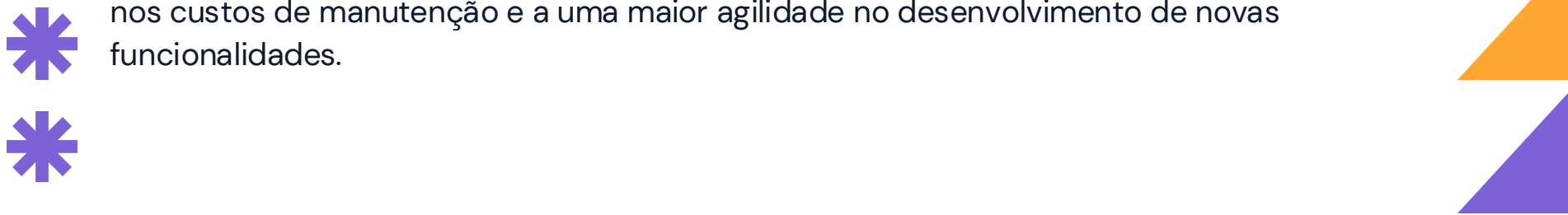


Os princípios SOLID são amplamente utilizados na indústria de software para criar sistemas mais robustos, escaláveis e fáceis de manter.

**Casos de Uso:** Empresas de tecnologia, como Google e Microsoft, aplicam esses princípios para desenvolver seus produtos e serviços. Eles são usados em sistemas bancários, jogos, aplicativos móveis e muitos outros tipos de software.

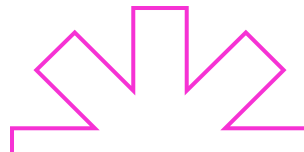
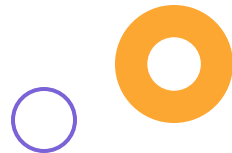
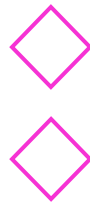
**Quem Usou:** Desenvolvedores e arquitetos de software em todo o mundo usam os princípios SOLID para melhorar a qualidade do código.

**Resultados Conseguídos:** A aplicação dos princípios SOLID geralmente resulta em um código mais limpo, que é mais fácil de testar, entender e modificar. Isso leva a uma redução nos custos de manutenção e a uma maior agilidade no desenvolvimento de novas funcionalidades.



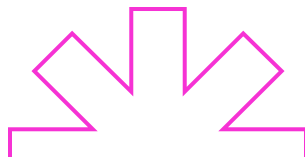
# 05

## Aplicação



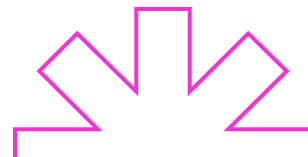
## Single Responsibility Principle (SRP)

- **Caso de Aplicação:** Uma classe Relatorio inicialmente é responsável por gerar o relatório, formatar os dados e enviá-lo por email. Isso viola o SRP porque a classe tem mais de uma razão para mudar.
- **Solução:** Dividir a classe Relatorio em três classes: GeradorRelatorio, FormatadorRelatorio, e EnviadorEmail. Cada uma destas classes terá uma única responsabilidade.



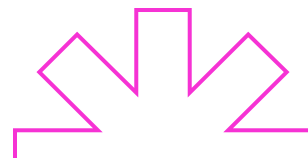
## Open/Closed Principle (OCP)

- **Caso de Aplicação:** Uma função que calcula o preço de produtos com diferentes descontos. Se novos tipos de descontos forem adicionados, o código existente precisa ser modificado.
- **Solução:** Usar herança e polimorfismo. Criar uma classe base Desconto e classes derivadas para cada tipo de desconto (DescontoPorPorcentagem, DescontoPorValorFixo). O cálculo do preço pode então iterar sobre uma coleção de objetos de Desconto.



## Liskov Substitution Principle (LSP)

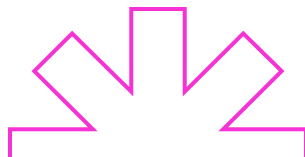
- **Caso de Aplicação:** Uma classe Ave com métodos voar() e fazerNinho(). Uma subclasse Pinguim é adicionada, mas como pinguins não voam, o método voar() não faz sentido para eles.
- **Solução:** Refatorar a hierarquia de classes. Criar uma classe base Ave sem o método voar, e subclasses AveQueVoa e AveQueNaoVoa. Pinguim herda de AveQueNaoVoa.





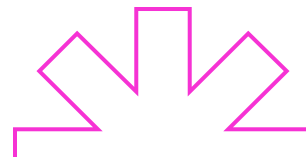
## Interface Segregation Principle (ISP)

- **Caso de Aplicação:** Uma interface `Trabalhador` com métodos `trabalhar()`, `gerenciar()` e `relatar()`. Uma classe `Desenvolvedor` que implementa essa interface precisa implementar métodos que não usa, como `gerenciar()` e `relatar()`.
- **Solução:** Dividir a interface `Trabalhador` em interfaces menores e mais específicas, como `Desenvolvedor`, `Gerente` e `Relator`. `Desenvolvedor` implementa apenas a interface que define `trabalhar()`



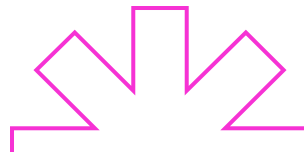
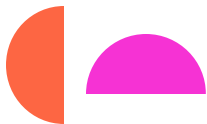
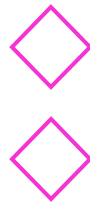
## Dependency Inversion Principle (DIP)

- **Caso de Aplicação:** Uma classe Pedido depende diretamente de uma classe MySQLDatabase para salvar dados. Se a base de dados mudar, a classe Pedido precisa ser alterada.
- **Solução:** Introduzir uma interface IDatabase que MySQLDatabase implementa. Pedido depende de IDatabase em vez de MySQLDatabase, permitindo mudar a implementação do banco de dados sem modificar Pedido.



# 06

## Referências







<https://blog.betrybe.com/linguagem-de-programacao/solid-cinco-principios-poo/>

<https://www.dtidigital.com.br/blog/adote-principios-solid-boas-praticas-de-programacao>

<https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/>

<https://www.alura.com.br/artigos/solid>

<https://medium.com/contexto-delimitado/os-princ%C3%ADpios-solid-34b136f507bb>

