

Engenharia de Softwares Escaláveis

Design Patterns e Domain-Driven Design com Java

Agenda

Etapas 1: Design Patterns e Princípios SOLID com Java.

- Introdução aos Princípios SOLID.
- Design Patterns.



Introdução aos Princípios SOLID

Na programação de computadores orientada a objetos, o termo **SOLID** é um acrônimo para cinco postulados de design, destinados a facilitar a compreensão, o desenvolvimento e a manutenção de software.

Os postulados **SOLID** foram apresentados por Robert C. Martin em um artigo publicado no ano 2000 cujo título, em tradução livre, é **Postulados de Projeto e Padrões de Projeto**.

SOLID

● **Single Responsibility / Responsabilidade única**

● **Open-Closed / Aberto-Fechado**

● **Liskov Substitution / Substituição de Liskov**

● **Interface Segregation / Segregação de Interface**

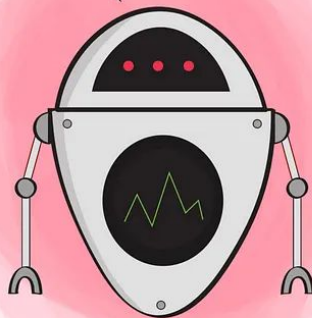
● **Dependency Inversion / Inversão de Dependência**

Single Responsibility

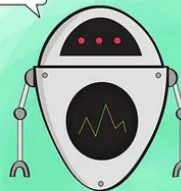
Uma classe deve ter uma única responsabilidade.

Este princípio visa separar comportamentos para que, se surgirem bugs como resultado de sua mudança, isso não afete outros comportamentos não relacionados.

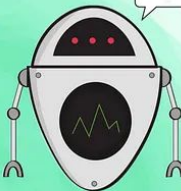
I am a chef,
a gardener,
a painter
& driver



I am a
chef



I am a
gardener



I am a
painter



I am a
driver



Single Responsibility

Problema:

Classe Fatura gera o conteúdo, salva no banco e envia e-mail.

Single
Responsibility

```
Abrir  Srp.java  Salvar  ~/Downloads/Material das Turmas/sol...
1 package katas.srp;
2
3 public class Fatura {
4     public void gerar(String cliente, double valor) {
5         System.out.println("Cliente: " + cliente);
6         System.out.println("Valor: " + valor);
7         salvarNoBanco(cliente, valor);
8         enviarEmail(cliente);
9     }
10
11     private void salvarNoBanco(String cliente, double valor) {
12         System.out.println("Salvando no banco...");
13     }
14
15     private void enviarEmail(String cliente) {
16         System.out.println("Enviando e-mail para " + cliente);
17     }
18 }
```

Java Largura da tabulação: 8 Lin 1, Col 1 INS

Solução:

Separar em classes distintas:

- Fatura → lógica da fatura.
- FaturaRepository → persistência.
- FaturaNotifier → notificação.

Single
Responsibility

```

SrpSolution.java
~/Downloads/Material das Turmas/sol...

1 package katas.srp;
2
3 public class Fatura {
4     private String cliente;
5     private double valor;
6
7     public Fatura(String cliente, double valor) {
8         this.cliente = cliente;
9         this.valor = valor;
10    }
11
12    public String getCliente() {
13        return cliente;
14    }
15
16    public double getValor() {
17        return valor;
18    }
19 }
20
21 class FaturaRepository {
22     public void salvar(Fatura fatura) {
23         System.out.println("Salvando fatura no banco: " +
24             fatura.getCliente());
25     }
26 }
27 class FaturaNotifier {
28     public void enviarEmail(Fatura fatura) {
29         System.out.println("Enviando e-mail para " +
30             fatura.getCliente());
31     }
32 }
```

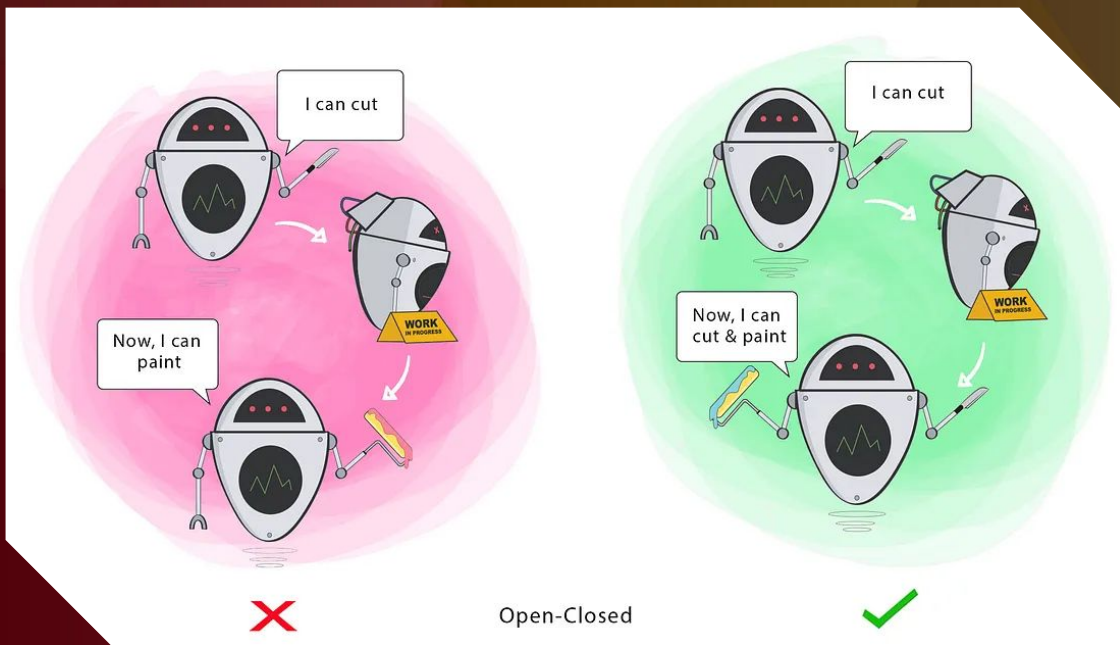
Java ▾ Largura da tabulação: 8 ▾ Lin 3, Col 1 ▾ INS

Open-Closed

As classes devem ser abertas para extensão, mas fechadas para modificação.

Este princípio visa estender o comportamento de uma classe sem alterar o comportamento existente.

Isso evita causar bugs onde quer que a classe esteja sendo usada.



Problema:

CalculadoraFrete usa if-else para decidir entre SEDEX, PAC etc.

Open-Closed

```
Abrir ▾  Ocp.java  Salvar  ≡  _  □  ×
~/Downloads/Material das Turmas/soli...

1 package katas.ocp;
2
3 public class CalculadoraFrete {
4     public double calcular(String tipo, double peso) {
5         if (tipo.equals("SEDEX")) {
6             return peso * 1.5;
7         } else if (tipo.equals("PAC")) {
8             return peso * 1.2;
9         }
10        return peso;
11    }
12 }
```

java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS

Solução:

Criar interface Frete e
implementar FreteSedex,
FretePAC, etc.
Usar polimorfismo para evitar if.

Open-Closed

```
OcpSolution.java
~/Downloads/Material ...

1 package katas.ocp;
2
3 interface Frete {
4     double calcular(double peso);
5 }
6
7 class FreteSedex implements Frete {
8     public double calcular(double peso) {
9         return peso * 1.5;
10    }
11 }
12
13 class FretePAC implements Frete {
14     public double calcular(double peso) {
15         return peso * 1.2;
16    }
17 }
18
19 class CalculadoraFrete {
20     public double calcular(Frete frete, double peso) {
21         return frete.calcular(peso);
22    }
23 }
```

java ▾ Largura da tabulação: 8 ▾ Lin 3, Col 1 ▾ INS

```
Abrir ▾  OcpTest.java  Salvar  ≡  _  □  X
~/Downloads/Material da...

1 package katas.ocp;
2
3 import org.junit.jupiter.api.Test;
4 import static org.junit.jupiter.api.Assertions.*;
5
6 public class CalculadoraFreteTest {
7     @Test
8     void testFreteSedex() {
9         CalculadoraFrete calc = new CalculadoraFrete();
10        assertEquals(15.0, calc.calcular("SEDEX", 10.0));
11    }
12 }
```

Java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS



```
Abrir ▾  OcpTest.java  Salvar  ≡  _  □  X
~/Downloads/Material da...

1 package katas.ocp;
2
3 import org.junit.jupiter.api.Test;
4 import static org.junit.jupiter.api.Assertions.*;
5
6 public class CalculadoraFreteTest {
7
8     @Test
9     void deveCalcularFretePAC() {
10        CalculadoraFrete calc = new CalculadoraFrete();
11        Frete pac = new FretePAC();
12        assertEquals(12.0, calc.calcular(pac, 10.0));
13    }
14
15     @Test
16     void deveCalcularFreteSedex() {
17        CalculadoraFrete calc = new CalculadoraFrete();
18        Frete sedex = new FreteSedex();
19        assertEquals(15.0, calc.calcular(sedex, 10.0));
20    }
21 }
```

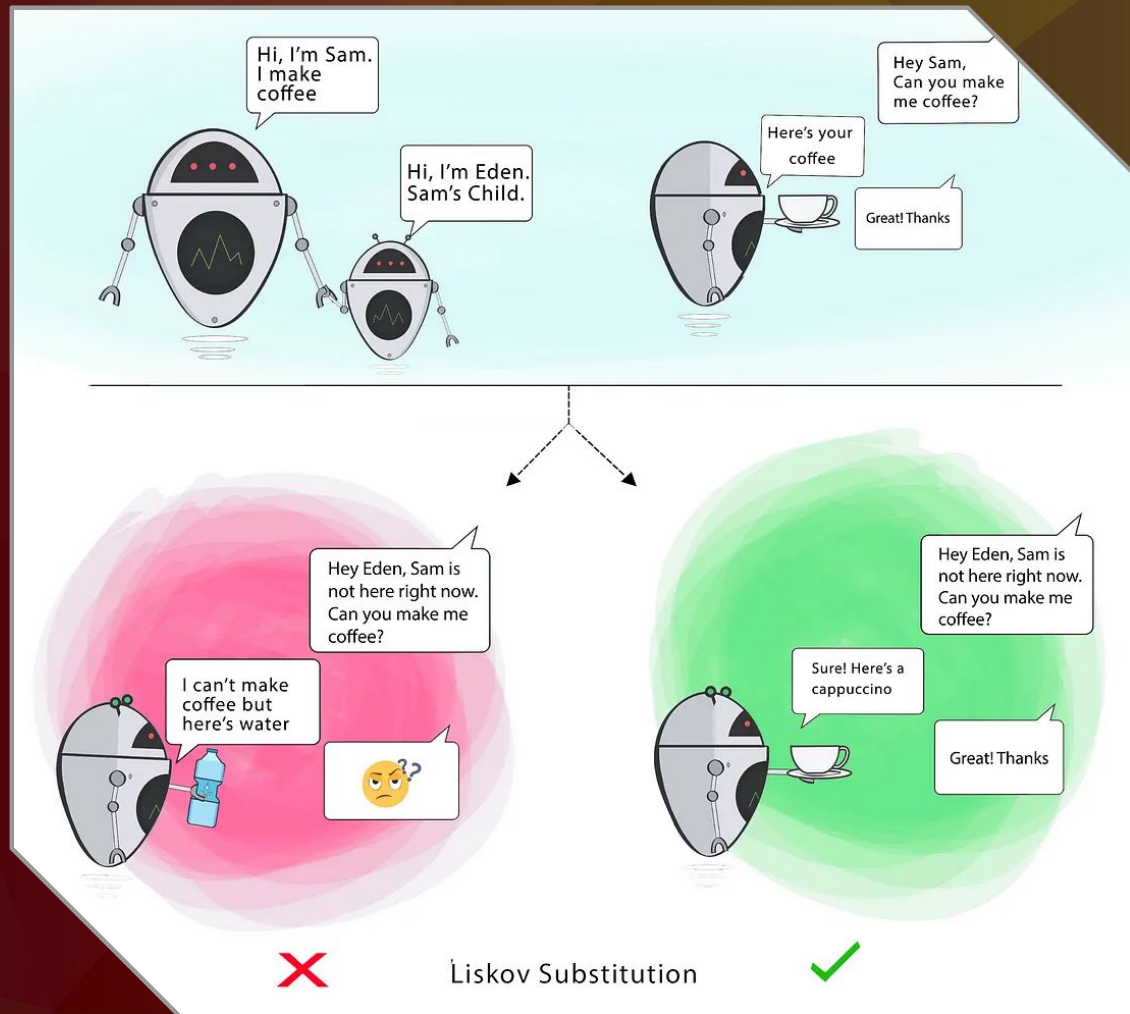
Java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS

Open-Closed

Liskov Substitution

Se S for um subtipo de T, então os objetos do tipo T em um programa podem ser substituídos por objetos do tipo S sem alterar nenhuma das propriedades desejáveis desse programa.

Este princípio visa reforçar a consistência para que a classe pai ou sua classe filha possam ser usadas da mesma maneira, sem erros.



Problema:

Quadrado herda de Retangulo, mas altera o comportamento de setLargura, setAltura, quebrando a área esperada.

Liskov
Substitution

```
Abrir ▾  Lsp.java  Salvar  ≡  _  □  ×
~/Downloads/M...

1 package katas.lsp;
2
3 public class Retangulo {
4     protected int largura;
5     protected int altura;
6
7     public void setLargura(int largura) {
8         this.largura = largura;
9     }
10
11    public void setAltura(int altura) {
12        this.altura = altura;
13    }
14
15    public int getArea() {
16        return largura * altura;
17    }
18 }
19
20 class Quadrado extends Retangulo {
21     @Override
22     public void setLargura(int tamanho) {
23         super.setLargura(tamanho);
24         super.setAltura(tamanho);
25     }
26
27     @Override
28     public void setAltura(int tamanho) {
29         super.setAltura(tamanho);
30         super.setLargura(tamanho);
31     }
32 }
```

Java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS

Solução:

Identificar a violação do LSP e discutir alternativas:


- Interface comum?
- Composição em vez de herança?

Liskov
Substitution

```
Abrir ▾  LspSolution...  Salvar  ≡  _  □  ×
~/Downloads/...

1 package katas.lsp;
2
3 interface Forma {
4     int getArea();
5 }
6
7 class Retangulo implements Forma {
8     protected int largura;
9     protected int altura;
10
11     public void setLargura(int largura) {
12         this.largura = largura;
13     }
14
15     public void setAltura(int altura) {
16         this.altura = altura;
17     }
18
19     public int getArea() {
20         return largura * altura;
21     }
22 }
23
24 class Quadrado implements Forma {
25     private int lado;
26
27     public void setLado(int lado) {
28         this.lado = lado;
29     }
30
31     public int getArea() {
32         return lado * lado;
33     }
34 }
```


java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS

Abrir ▾  LspTest.java
~/Downloads/Material ... Salvar ≡ _ □ ✕

```
1 package katas.lsp;  
2  
3 import org.junit.jupiter.api.Test;  
4 import static org.junit.jupiter.api.Assertions.*;  
5  
6 public class RetanguloTest {  
7     @Test  
8     void testRetanguloArea() {  
9         Retangulo r = new Retangulo();  
10        r.setLargura(4);  
11        r.setAltura(5);  
12        assertEquals(20, r.getArea());  
13    }  
14  
15    @Test  
16    void testQuadradoArea() {  
17        Retangulo q = new Quadrado();  
18        q.setLargura(4);  
19        q.setAltura(5);  
20        // Resultado inesperado, ilustra quebra do LSP  
21        assertNotEquals(20, q.getArea());  
22    }  
23 }
```

Java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS



Abrir ▾  LspTest.java
~/Downloads... Salvar ≡ _ □ ✕

```
1 package katas.lsp;  
2  
3 import org.junit.jupiter.api.Test;  
4 import static org.junit.jupiter.api.Assertions.*;  
5  
6 public class FormaTest {  
7  
8     @Test  
9     void deveCalcularAreaRetangulo() {  
10        Retangulo r = new Retangulo();  
11        r.setLargura(4);  
12        r.setAltura(5);  
13        assertEquals(20, r.getArea());  
14    }  
15  
16    @Test  
17    void deveCalcularAreaQuadrado() {  
18        Quadrado q = new Quadrado();  
19        q.setLado(4);  
20        assertEquals(16, q.getArea());  
21    }  
22 }
```

Java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS

Exercises

All robots: spin around,
rotate arms, wiggle
antennas

Oops! But I don't
have antennas



Interface Segregation

Exercises

Robots that spin around: spin
around
Robots that wiggle antennas:
wiggle antennas

Awesome!



Interface Segregation

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

Este princípio visa dividir um conjunto de ações em conjuntos menores para que uma classe execute APENAS o conjunto de ações que necessita.

Problema:

Interface Funcionario tem comer() e trabalhar(). Robo implementa ambos, mas comer() não faz sentido.

Interface
Segregation

```
Abrir ▾  lsp.java  Salvar  ≡  _  □  ×
~/Downloads/Material das Turmas/solid-kata...

1 package katas.isp;
2
3 interface Funcionario {
4     void trabalhar();
5     void comer();
6 }
7
8 class Robo implements Funcionario {
9     public void trabalhar() {
10         System.out.println("Robô trabalhando.");
11     }
12
13     public void comer() {
14         throw new UnsupportedOperationException("Robôs não comem!");
15     }
16 }
```

Java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS

Solução:

Criar interfaces separadas:
Trabalhador, Comedor, Dorminhoco.
Robôs só implementam o necessário.

Interface
Segregation

```
Abrir ▾ IspSolution.java ~/Downloads/Mat... Salvar ≡ _ □ ×
1 package katas.isp;
2
3 interface Trabalhador {
4     void trabalhar();
5 }
6
7 interface Comedor {
8     void comer();
9 }
10
11 class Humano implements Trabalhador, Comedor {
12     public void trabalhar() {
13         System.out.println("Humano trabalhando.");
14     }
15
16     public void comer() {
17         System.out.println("Humano comendo.");
18     }
19 }
20
21 class Robo implements Trabalhador {
22     public void trabalhar() {
23         System.out.println("Robô trabalhando.");
24     }
25 }
```

Java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS


```
Abrir ▾  IspTest.java  Salvar  ≡  _  □  X
~/Downloads/Material das Turmas/solid-katas-java...

1 package katas.isp;
2
3 import org.junit.jupiter.api.Test;
4
5 import static org.junit.jupiter.api.Assertions.assertThrows;
6
7 public class RoboTest {
8     @Test
9     void testComerDeveFalhar() {
10         Robo robo = new Robo();
11         assertThrows(UnsupportedOperationException.class, robo::comer);
12     }
13 }
```

Java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS



```
Abrir ▾  IspT...  Salvar  ≡  _  □  X
~/Do...

1 package katas.isp;
2
3 import org.junit.jupiter.api.Test;
4
5 public class TrabalhadorTest {
6
7     @Test
8     void humanoTrabalhaECome() {
9         Humano h = new Humano();
10        h.trabalhar();
11        h.comer();
12    }
13
14    @Test
15    void roboTrabalha() {
16        Robo r = new Robo();
17        r.trabalhar();
18    }
19 }
```

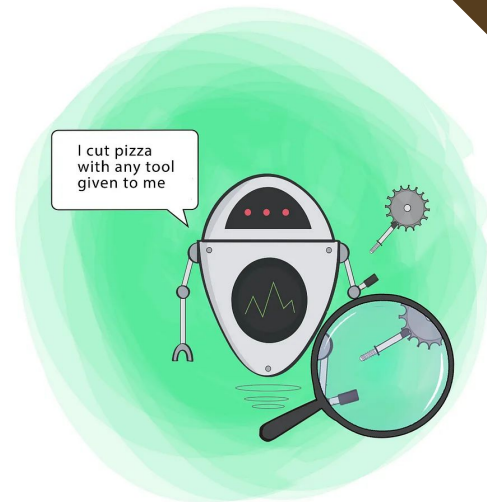
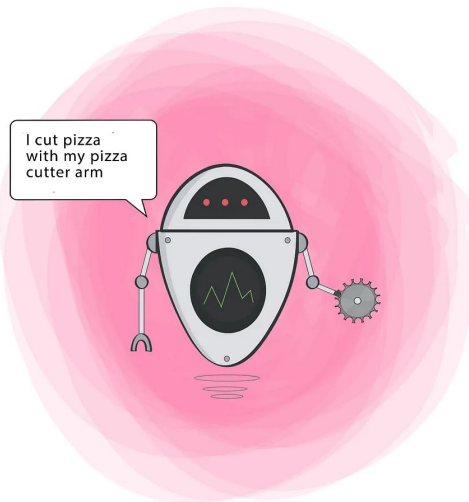
Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS

Dependency Inversion

Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender da abstração.

As abstrações não devem depender de detalhes. Os detalhes devem depender de abstrações.

Este princípio visa reduzir a dependência de uma classe de alto nível em relação à classe de baixo nível, introduzindo uma interface.



Dependency Inversion

Problema:

RelatorioService instancia diretamente RelatorioPDF.

Dependency
Inversion

```
Abrir ▾  Dip.java  Salvar  ≡  _  □  ×
~/Downloads/Material das Turmas/sol...

1 package katas.dip;
2
3 public class RelatorioService {
4     public void gerarRelatorio() {
5         RelatorioPDF pdf = new RelatorioPDF();
6         pdf.formatar("Relatório anual");
7     }
8 }
9
10 class RelatorioPDF {
11     public void formatar(String conteudo) {
12         System.out.println("Formatando como PDF: " + conteudo);
13     }
14 }
```

java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS

Solução:

Criar interface RelatorioFormatter.
RelatorioPDF implementa a interface
e é injetado no construtor de
RelatorioService.

Dependency
Inversion

```
Abrir ▾  DipSolution.java  Salvar  ≡  _  □  ✕
~/Downloads/Material das Turmas/...

1 package katas.dip;
2
3 interface RelatorioFormatter {
4     void formatar(String conteudo);
5 }
6
7 class RelatorioPDF implements RelatorioFormatter {
8     public void formatar(String conteudo) {
9         System.out.println("Formatando como PDF: " + conteudo);
10    }
11 }
12
13 class RelatorioService {
14     private RelatorioFormatter formatter;
15
16     public RelatorioService(RelatorioFormatter formatter) {
17         this.formatter = formatter;
18     }
19
20     public void gerarRelatorio(String conteudo) {
21         formatter.formatar(conteudo);
22     }
23 }
```

java ▾ Largura da tabulação: 8 ▾ Lin 1, Col 1 ▾ INS

Abrir  DipTest.java Salvar    

~/Downloads/Material das Turmas...

```
1 package katas.dip;
2
3 import org.junit.jupiter.api.Test;
4
5 public class RelatorioServiceTest {
6     @Test
7     void testGerarRelatorio() {
8         RelatorioService service = new RelatorioService();
9         service.gerarRelatorio();
10    }
11 }
```

Java  Largura da tabulação: 5  Lin 1, Col 1  INS

Abrir  DipTest.java Salvar    

~/Downloads/Material das Turmas/solid-katas-gab...

```
1 package katas.dip;
2
3 import org.junit.jupiter.api.Test;
4
5 public class RelatorioServiceTest {
6
7     @Test
8     void deveFormatarComoPDF() {
9         RelatorioFormatter formatter = new RelatorioPDF();
10        RelatorioService service = new RelatorioService(formatter);
11        service.gerarRelatorio("Relatório de vendas");
12    }
13 }
```

Java  Largura da tabulação: 8  Lin 1, Col 1  INS

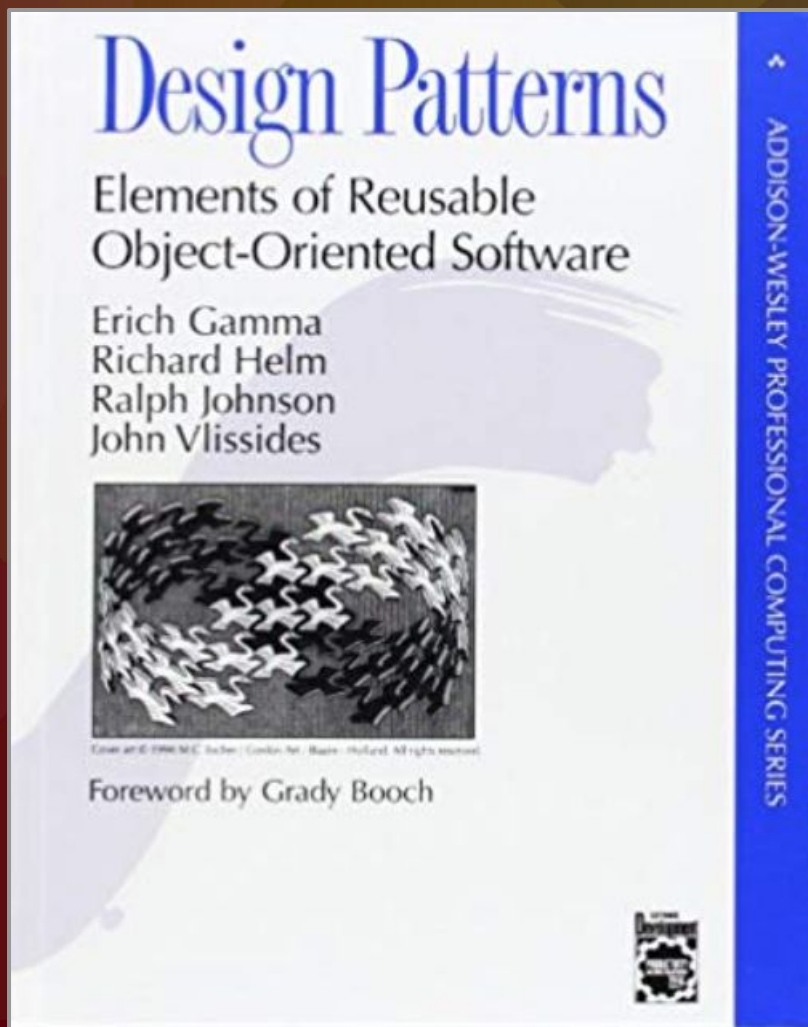


Design Patterns

Design Patterns

A ideia de **Design Patterns** pode ser descrita como um conjunto de abordagens de codificação reutilizáveis que resolvem os problemas mais comuns encontrados durante o desenvolvimento de aplicativos.

Essas abordagens estão alinhadas com os conceitos **APIE** e **SOLID** mencionados anteriormente e têm um impacto incrivelmente positivo em trazer transparência, legibilidade e testabilidade ao caminho de desenvolvimento.



Propósito

Escopo	Classe	Creational	Structural	Behavioral
		Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade (ou Facade) Business Delegate Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor