

TÓPICO 01: INTRODUÇÃO A DESIGN PATTERNS

Design Patterns - Professor Ramon Venson - SATC 2026.2

Design não é aparência, é **como funciona.**



"Design é o ato de decidir como algo deve ser" - Jesse Schell

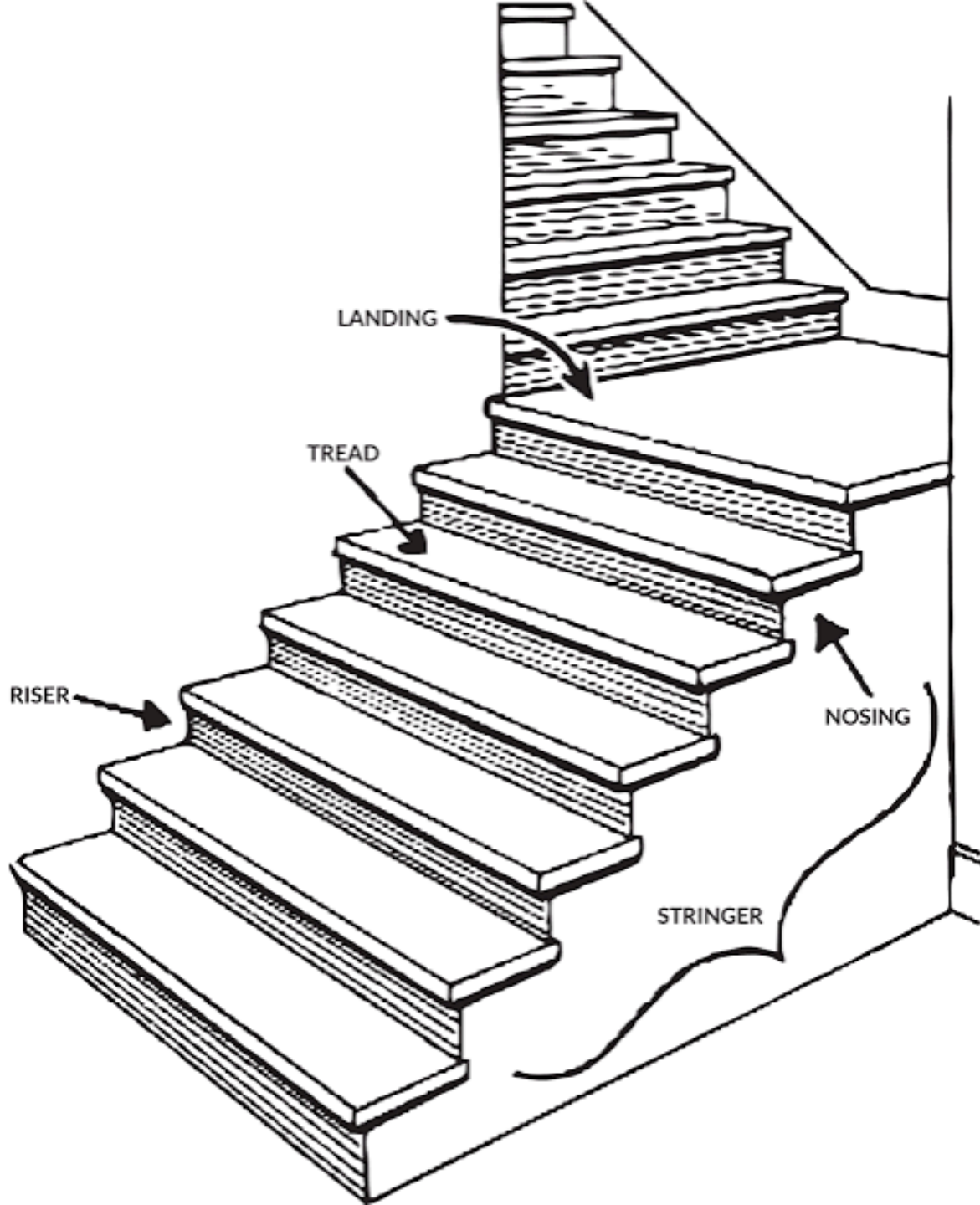
Todo design nasce de um **problema** a ser resolvido e de uma **decisão** tomada pra resolvê-lo.

Nem todo design é bom, mas todo design tem um **propósito**.



Qual a melhor maneira de conectar dois andares?





Conectando os pontos

A escada é um **padrão de design** recorrente. Porém não é a **única** solução possível.

Tópico 02 - Introdução a Design Patterns



Tópico 02 - Introdução a Design Patterns



Design Patterns

Um design pattern é uma **solução reutilizável** para um **problema recorrente**.

Sempre que precisarmos chegar ao segundo andar, podemos usar a **mesma ideia** de design, adaptando-a ao contexto.



A Pattern Language

Towns · Buildings · Construction



Christopher Alexander

Sara Ishikawa · Murray Silverstein

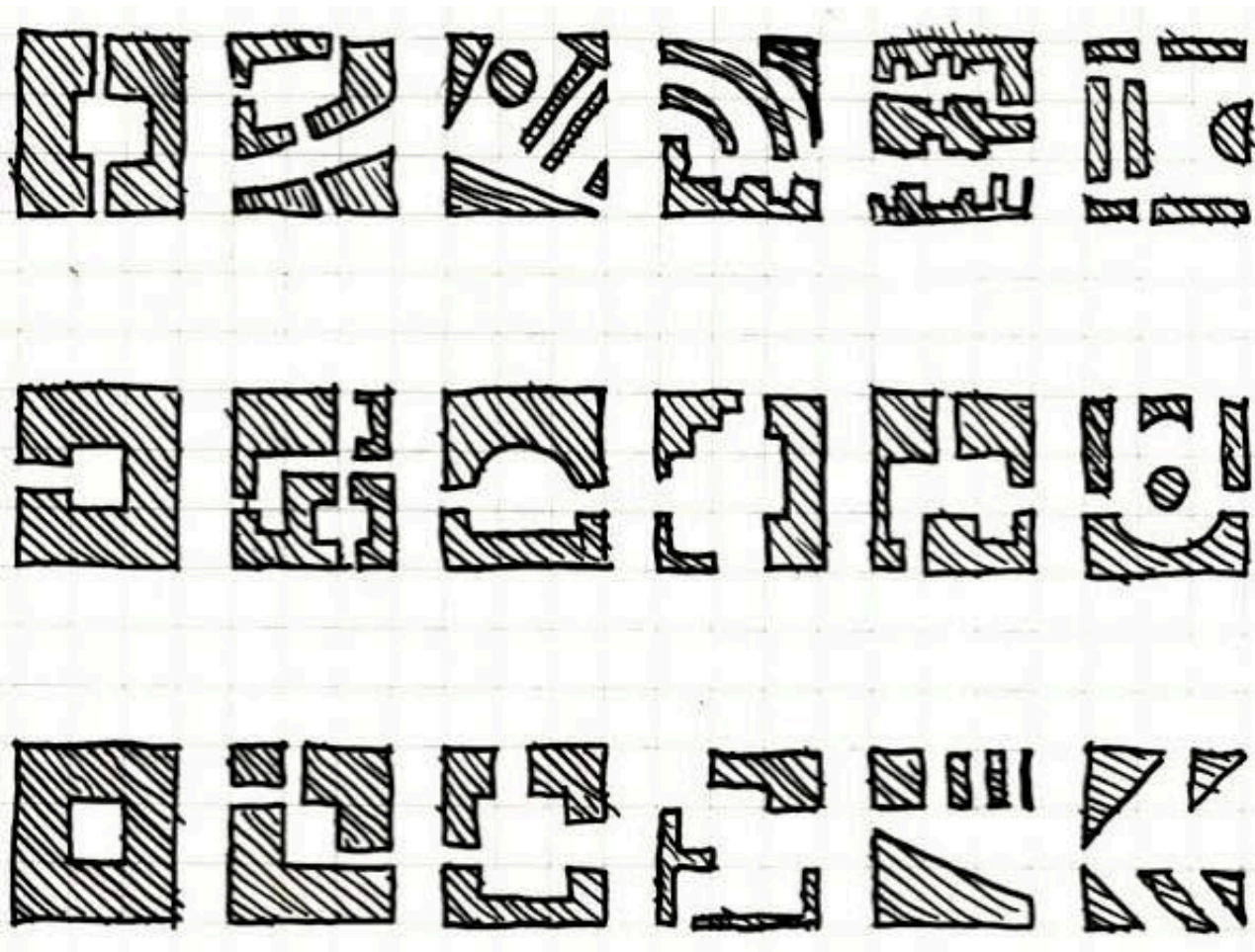
WITH

Max Jacobson · Ingrid Fiksdahl-King

Shlomo Angel

Origem (1/2)

- Conceito de “padrão” veio da **Arquitetura**
- Christopher Alexander: 1977: *A Pattern Language*
 - Padrões para projetar cidades, prédios, ambientes



1 Levels of Scale



2 Strong Centers



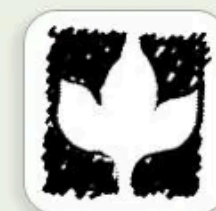
3 Boundaries



4 Alternating Repetition



5 Positive Space



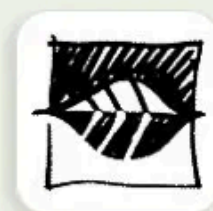
6 Good Shape



7 Local Symmetries



8 Deep Interlock



9 Contrast



10 Graded Variation



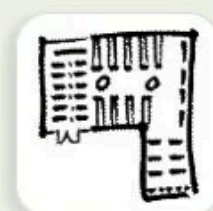
11 Roughness



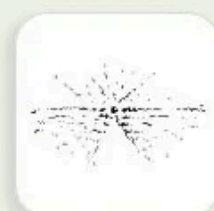
12 Echoes



13 The Void



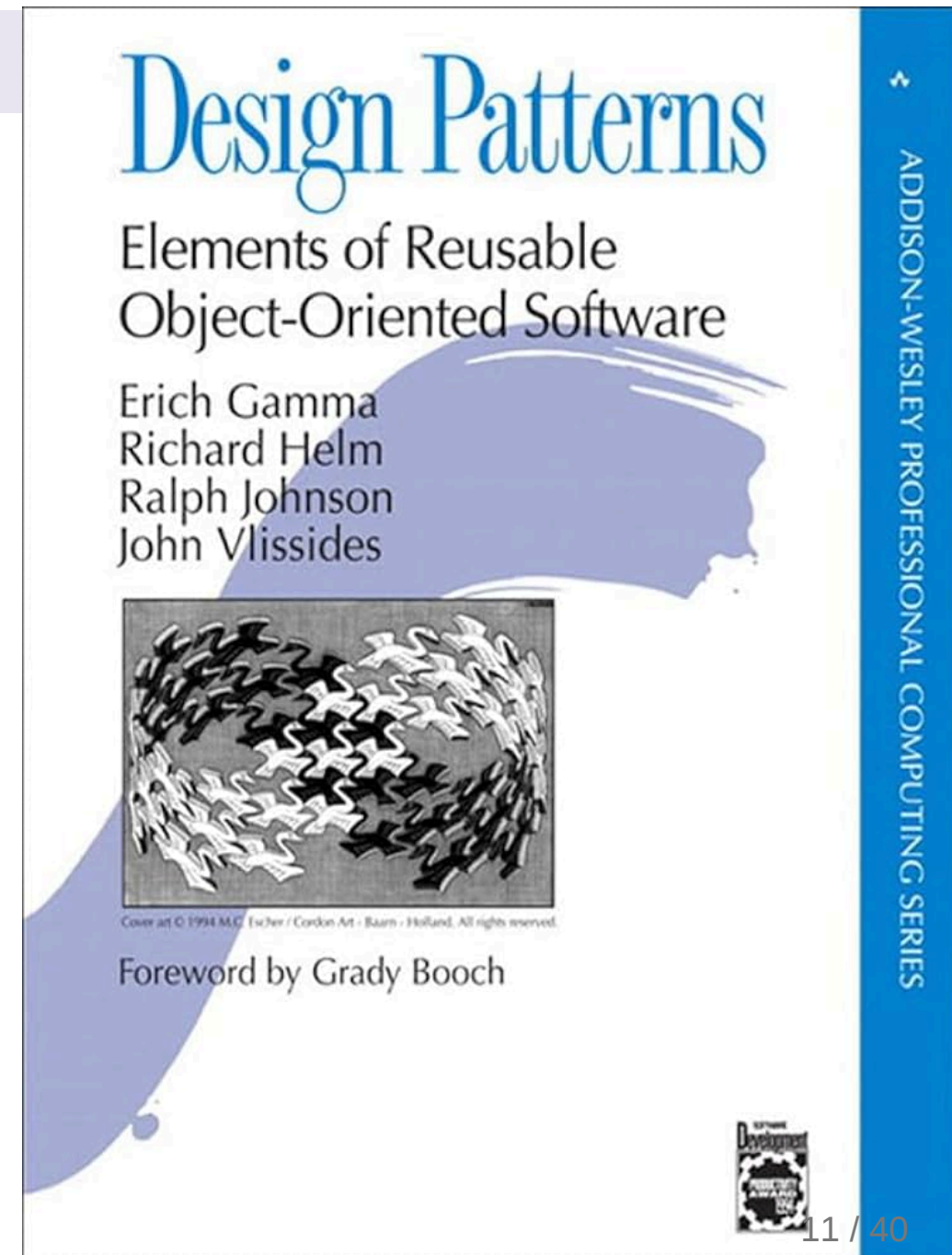
14 Inner Calm

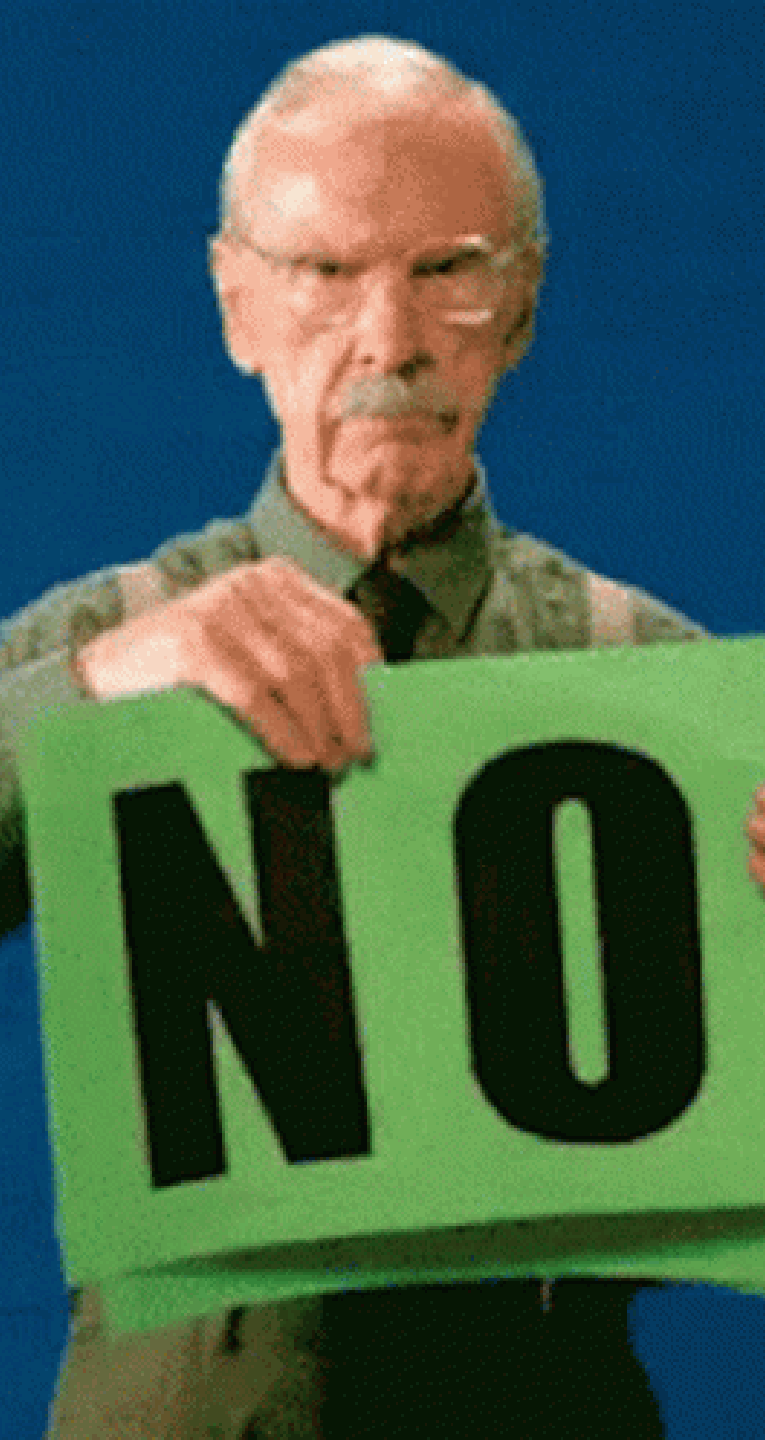


15 Not-Separateness

Origem (2/2)

- Adaptado para software nos anos 90
- “Gang of Four” (GoF):
 - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Livro (1994):
Design Patterns: Elements of Reusable Object-Oriented Software
- Catálogo clássico com **23 padrões**





O que NÃO é Design Pattern

- **Framework:** conjunto de classes e bibliotecas
- **API:** conjunto de funções e métodos
- **Biblioteca:** conjunto de funções, classes e recursos
- **Boilerplate:** código repetitivo e padronizado
- **Código pronto:** código que resolve um problema específico
- **Algoritmo:** sequência de passos para resolver um problema específico

Exemplo de ideia de padrão

O código abaixo cria um objeto `Carro` com **vários parâmetros**:

```
public class Carro {  
    private String cor;  
    private int portas;  
    private float velocidadeMaxima;  
  
    private Carro(String cor, int portas, float velocidadeMaxima) {  
        this.cor = cor;  
        this.portas = portas;  
        this.velocidadeMaxima = velocidadeMaxima;  
    }  
}
```

Para instanciar um carro, precisamos passar **todos os parâmetros**:

```
Carro carro = new Carro("vermelho", 4, 200);
```

Mas o que acontece se quisermos criar um carro com **velocidade máxima padrão** de 200 km/h?

E se nosso carro tiver **muitos parâmetros**? Teremos que criar **muitos construtores**?

Vamos resolver esse problema adaptando uma solução bem conhecida.

```
public static class CarroBuilder {  
    private String cor;  
    private int portas;  
    private float velocidadeMaxima = 200;  
  
    public CarroBuilder setCor(String cor) {  
        this.cor = cor; return this;  
    }  
  
    public CarroBuilder setPortas(int portas) {  
        this.portas = portas; return this;  
    }  
  
    public CarroBuilder setVelocidadeMaxima(float velocidadeMaxima) {  
        this.velocidadeMaxima = velocidadeMaxima; return this;  
    }  
  
    public Carro build() {  
        return new Carro(cor, portas, velocidadeMaxima);  
    }  
}
```

Ao invés de:

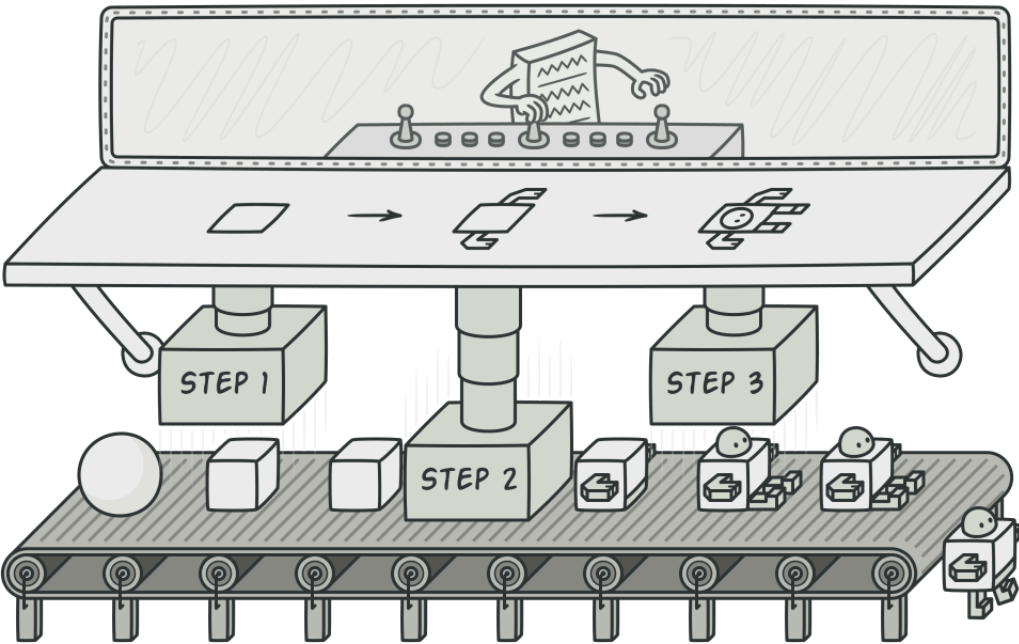
```
Carro carro = new Carro("vermelho", 4, 200);
```

Podemos usar:

```
Carro carro = new CarroBuilder()  
    .setCor("vermelho")  
    .setPortas(4)  
    .build();
```

Será criado um carro vermelho, com 4 portas e velocidade máxima de 200 km/h.

Mas onde foi que definimos que a velocidade máxima padrão é 200 km/h?



Explicação do Padrão

O exemplo anterior utiliza um padrão chamado **Builder**.

Esse padrão é útil quando:

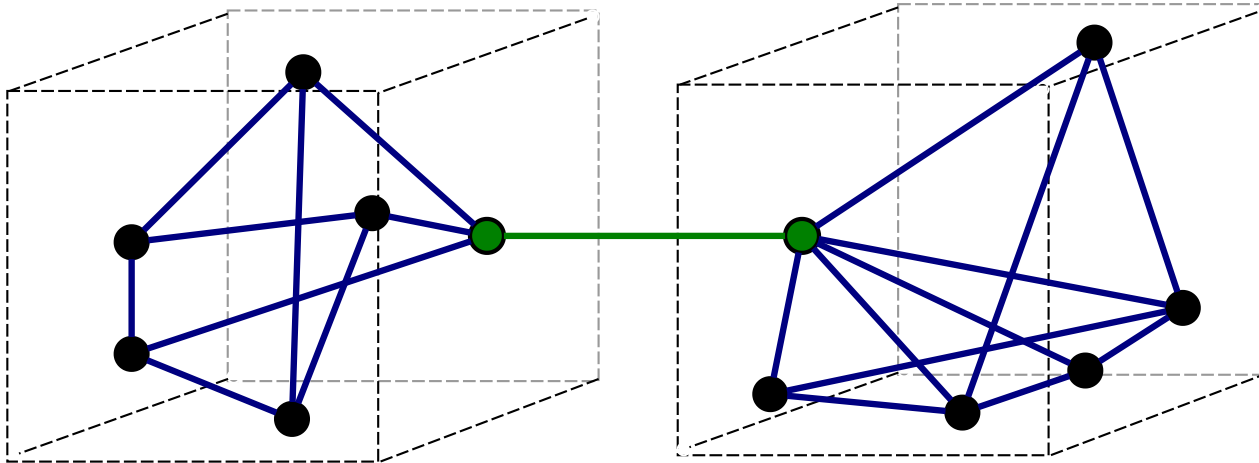
- Um objeto tem **muitos parâmetros**
- Alguns parâmetros são **opcionais**
- Queremos **evitar construtores com muitos argumentos**

Importância

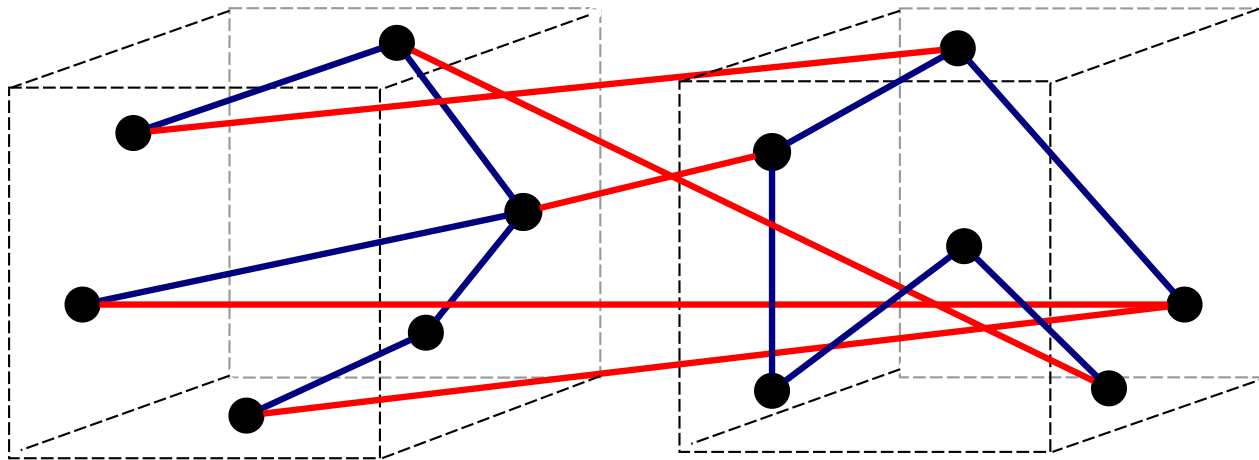
Padrões ajudam a:

- Reutilizar soluções testadas
- Melhorar a **comunicação** entre desenvolvedores
- Organizar melhor o código
- Facilitar manutenção e evolução
- Reduzir riscos de “reinventar a roda”





a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

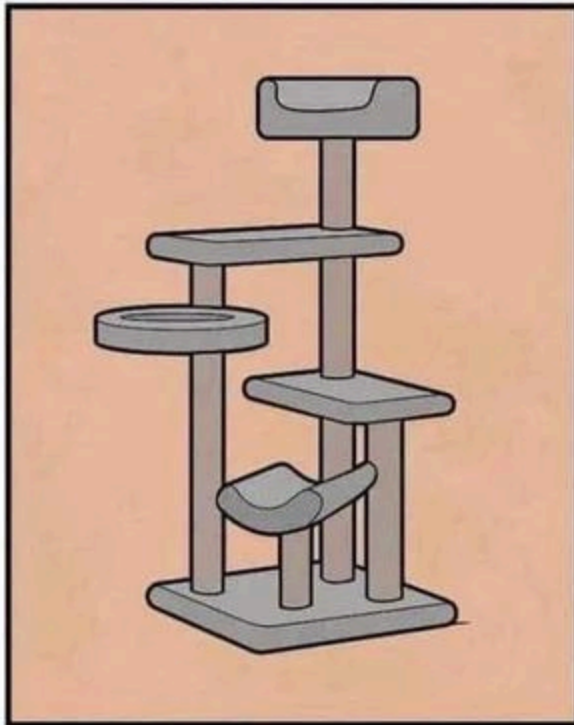
Coesão e Acoplamento

Em engenharia de software, o **acoplamento** é o nível de interdependência entre dois módulos de software.

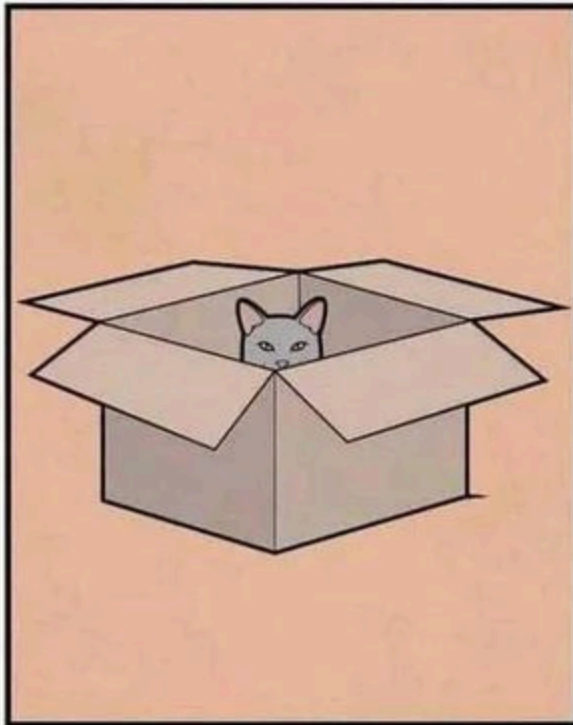
É geralmente um contraste ao conceito de **coesão**.

Coesão é uma medida do quão similares são os elementos de um sistema.

Product features



User needs

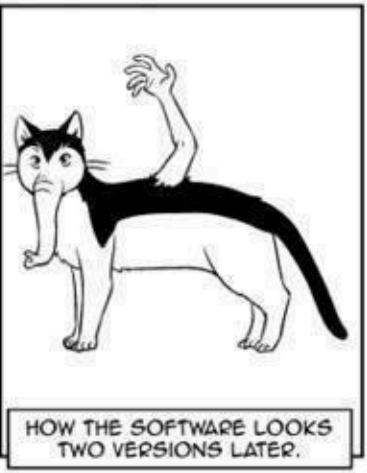
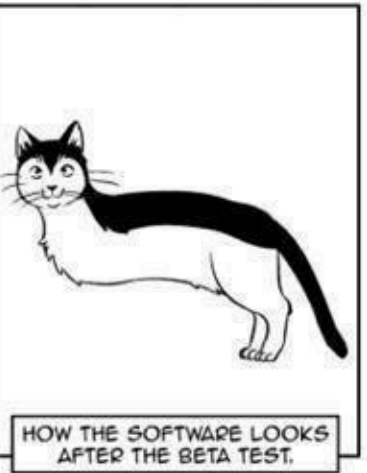
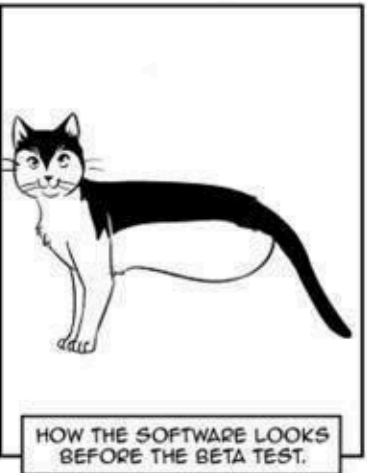
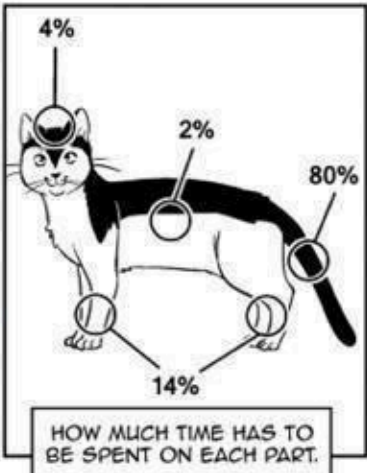
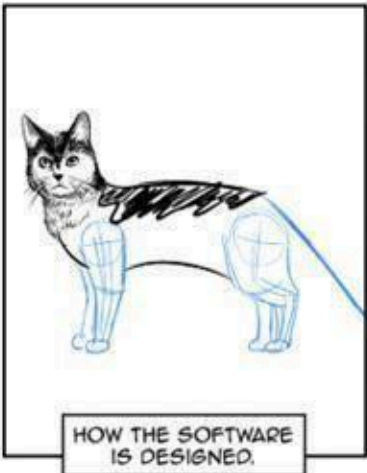


© _yes_but

Ciclo de Evolução do Software

Nenhum software nasce feito sob padrões. Ele **evolui** com o tempo, e padrões podem surgir **quando necessário**.

Richard's guide to software development



Sandra and Woo by Oliver Knörzer (writer) and Powree (artist) – www.sandraandwoo.com

PROGRAMADORES



Comunicação com padrões

- "Vamos usar um **Builder** aqui"
- "Isso parece um **Observer**"
- "Esse código é um **Strategy**"

Cada uma dessas frases comunica:

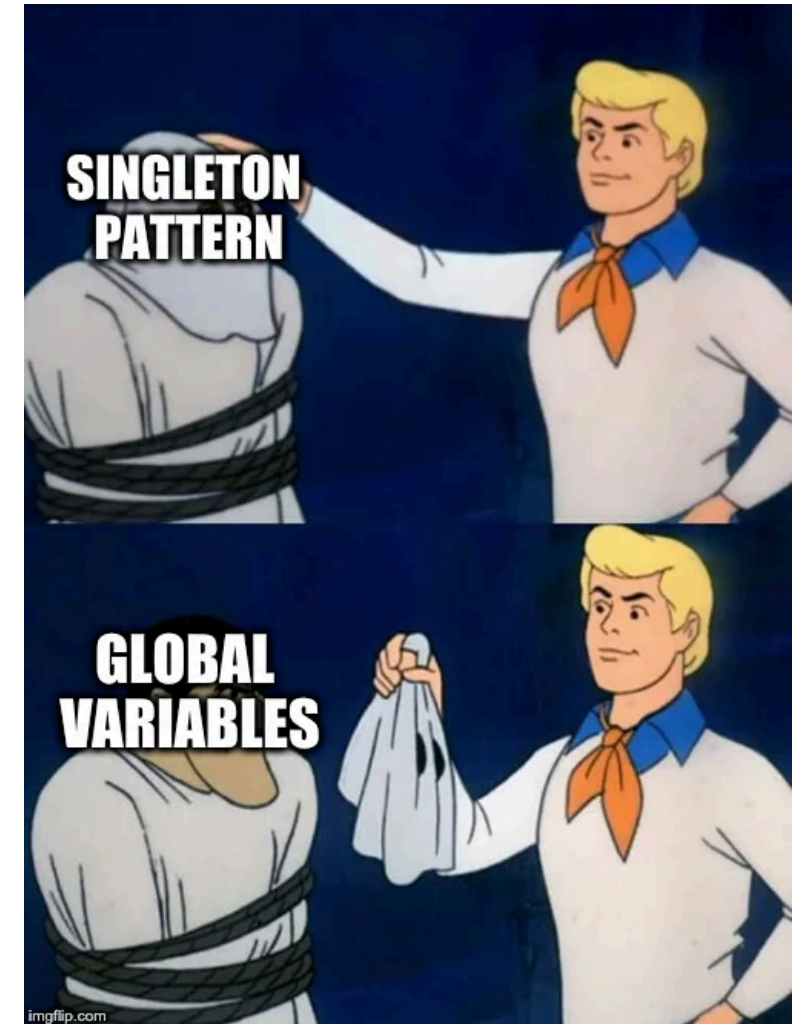
- o **problema**
- a **ideia de solução**
- as **consequências** esperadas

Limitações

Críticas (1/4): Sintoma de problema

- Às vezes o padrão “nasce” de uma **limitação da linguagem**
- Linguagens modernas tornam alguns padrões:
 - triviais
 - desnecessários

Ex.: Strategy pode virar apenas “passar função como parâmetro” em linguagens funcionais.





Críticas (2/4): Overengineering

- Uso exagerado de padrões
- Problemas:
 - muito código para pouco ganho
 - excesso de classes / interfaces
 - mais difícil de entender

Regra prática:

Comece simples. Introduza um padrão quando houver um **motivo claro**.

Críticas (3/4): Não são bala de prata

Padrões **não**:

- consertam modelo de domínio ruim
- resolvem decisões arquiteturais equivocadas
- substituem entendimento do problema

São **ferramentas**, não objetivo final.



Críticas (4/4): Jargão

- Muito nome: Factory, Visitor, Decorator, Proxy...
- Pode:
 - excluir quem é iniciante
 - gerar mal-entendidos se o time não conhece os termos

Dica:

- Use o nome do padrão
- Mas **explique a intenção**: “resolver X fazendo Y”.

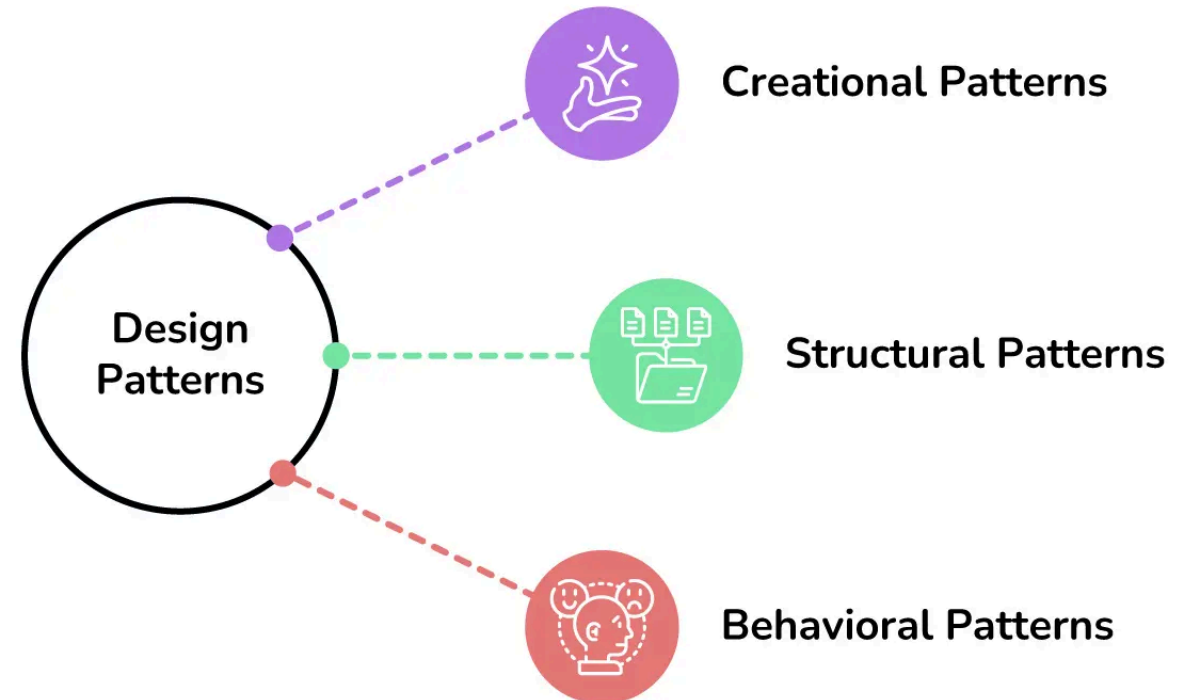


Categorias Clássicas

A GoF agrupou os padrões em **3** categorias:

- **Criacionais**
- **Estruturais**
- **Comportamentais**

Cada categoria agrupa padrões com **propósitos parecidos**.





Padrões Criacionais

- Problema: **como criar objetos de maneira eficiente?**
- Evitar:
 - `new` espalhado
 - acoplamento à implementação
 - permitir **flexibilidade na criação de objetos**

Exemplos de padrões criacionais

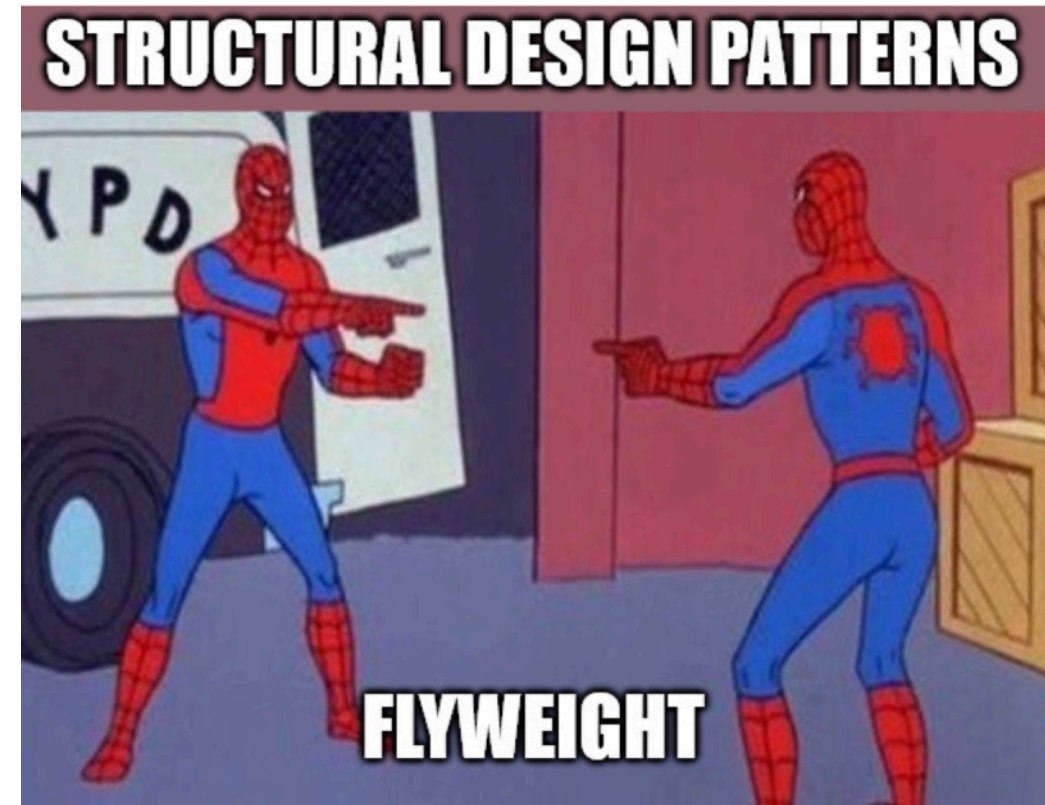
- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Exemplo simples: Factory Method

```
public interface Transporte {  
    void entregar();  
}  
  
public class Caminhao implements Transporte {  
    public void entregar() { System.out.println("Entrega por caminhão"); }  
}  
  
public abstract class Logistica {  
    public void planejarEntrega() {  
        Transporte t = criarTransporte();  
        t.entregar();  
    }  
  
    protected abstract Transporte criarTransporte();  
}
```

Padrões Estruturais

- Problema: **como organizar classes/objetos de maneira flexível?**
- Evitar:
 - acoplamento excessivo
 - dependência de implementação
 - permitir **composição de objetos**



Exemplos de padrões estruturais

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Exemplo simples: Adapter

```
// API antiga
class TomadaAntiga {
    void ligarDoisPinos() { ... }
}

// Nova interface esperada
interface TomadaTresPinos {
    void ligarTresPinos();
}

// Adapter
class Adaptador implements TomadaTresPinos {
    private TomadaAntiga antiga;
    public Adaptador(TomadaAntiga antiga) { this.antiga = antiga; }
    public void ligarTresPinos() { antiga.ligarDoisPinos(); }
}
```

Padrões Comportamentais

- Problema: **como objetos interagem sem alto acoplamento?**
- Evitar:
 - mudanças em uma classe afetarem outras
 - permitir reutilização de algoritmos
 - permitir **flexibilidade na comunicação entre objetos**



The Strategy Pattern

Exemplos de padrões comportamentais

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Observer
- Strategy
- Template Method
- Visitor

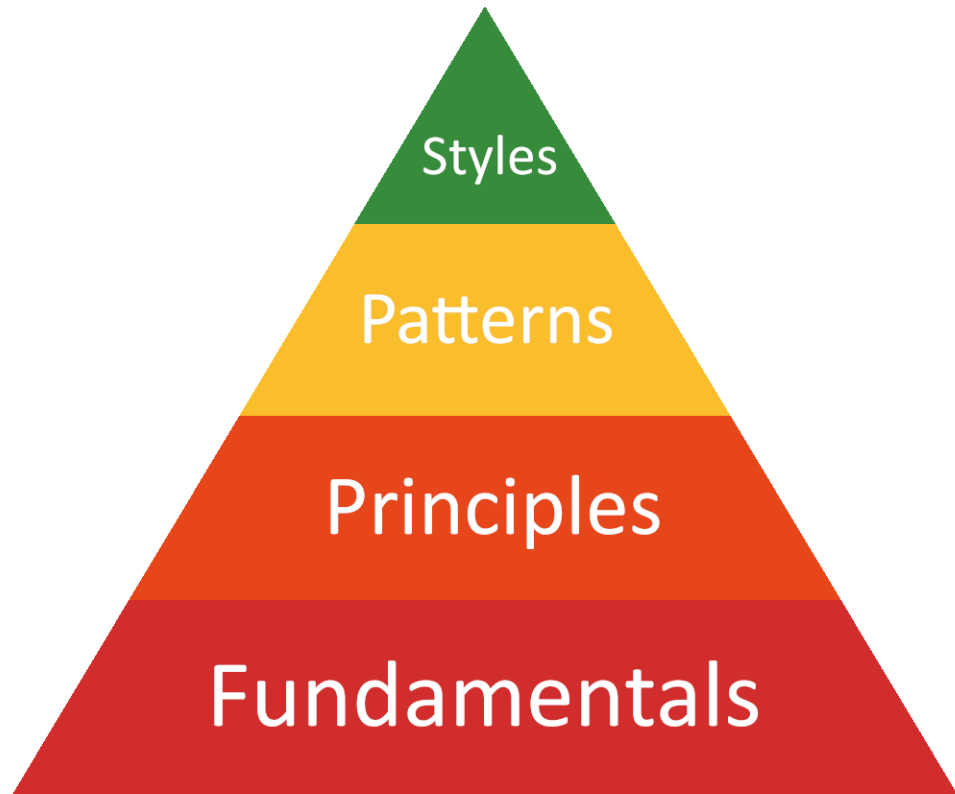
Exemplo simples: Observer

```
interface Observador {  
    void atualizar();  
}  
  
class Sujeito {  
    private List<Observador> obs = new ArrayList<>();  
    void adicionar(Observador o) { obs.add(o); }  
    void notificar() {  
        for (Observador o : obs) o.atualizar();  
    }  
}
```

- Vários objetos reagem a um evento sem o sujeito saber detalhes deles

Estrutura de um Padrão

- **Problema:** resolve um problema recorrente
- **Estrutura UML:** diagrama de classes ou sequência
- **Participantes:** classes e objetos envolvidos
- **Solução:** descreve a ideia de como resolver o problema



Padrões no Desenvolvimento de Software

- **Fundamentos:** encapsulamento, abstração, polimorfismo, herança
- **Princípios:** SOLID, DRY, KISS, YAGNI
- **Padrões de Design:** Criacionais, Estruturais, Comportamentais
- **Estilo:** Code Style, Clean Code, Code Smells, Refactoring

Outros tipos de padrões

Além dos GoF:

- **Arquiteturais:** MVC, MVP, MVVM, Microservices, Event-Driven Architecture...
- **Integração:** Message Bus, Publish-Subscribe, Event Sourcing, Saga Pattern...
- **Teste:** Test Doubles, Mock Objects, Test Data Builders...
- **Distribuídos:** Saga, CQRS, Event Sourcing, Circuit Breaker, Bulkhead...
- **Enterprise:** Repository, Unit of Work, DTO, Active Record, Lazy Load...
- **Resiliência:** Circuit Breaker, Rate Limit, Retry, Timeout, Fallback...
- **Cloud:** Sidecar, Ambassador, Strangler Fig, Service Mesh, Serverless...
- **Segurança:** Role-Based Access Control, Zero Trust, Secrets Management...

Resumo

- Padrões de design = **vocabulário + soluções**
- Use para:
 - comunicar melhor
 - organizar melhor o código
- Evite:
 - aplicar padrão “porque é bonito”
 - complexidade desnecessária

