# TÓPICO 05: BUILDER

Design Patterns - Professor Ramon Venson - SATC 2026.1

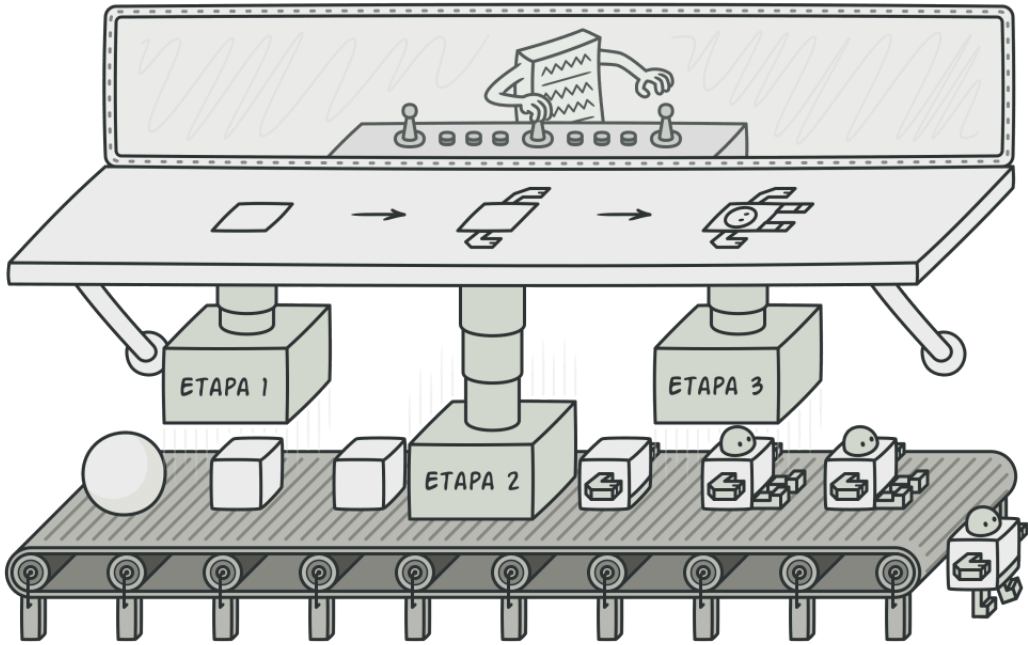
# O que é Builder?

Padrão **criacional** para:

- Construir **objetos complexos passo a passo**
- Evitar construtores gigantes
- Melhorar **legibilidade** e **flexibilidade**

Especialmente útil com:

- muitos parâmetros
- muitos opcionais
- várias combinações possíveis



# Problema: Construtor telescópico

```
Pedido pedido = new Pedido(
  cliente,
  enderecoEntrega,
  itens,
  desconto,
  cupom,
  frete,
  observacoes
  // ...
);
```

- Difícil de ler
- Fácil errar a ordem
- Fica pior quando adicionamos novos campos

# Ideia do Builder

Separar:

- **O que** está sendo montado (Produto)
- **Como** é montado (Builder)

Permite:

- montar em **passos claros**
- reaproveitar o **processo de construção**
- criar **variações** mudando o Builder

# Analogia: Sanduíche

- **Produto**: sanduíche pronto
- **Builder**: pessoa que monta o sanduíche

Passos típicos:

1. escolher o pão
2. escolher a proteína
3. adicionar queijo/opcionais
4. vegetais
5. molhos

# Estrutura (visão geral)

- **Product**: objeto final (ex.: `Sanduiche`, `Pedido`)
- **Builder**: interface/abstração com passos de construção
- **Concrete Builders**: implementações para variações

Na prática, muitas APIs usam **um Builder interno** na própria classe, sem Director explícito ou abstrações maiores.

# Builder fluente

Iniciamos com uma classe `PedidoBuilder`:

```
public static class PedidoBuilder {
  private String cliente;
  private String endereco;
  private double desconto = 0.0; // valor padrão
}
```

> Algumas implementações podem utilizar um **Builder interno** (static nested class) dentro do Produto, para facilitar a construção.

# Métodos de configuração

```java
public PedidoBuilder comCliente(String c) {
  this.cliente = c;
  return this;
}

public PedidoBuilder comEndereco(String e) {
  this.endereco = e;
  return this;
}

public PedidoBuilder comDesconto(double d) {
  this.desconto = d;
  return this;
}
```

O segredo para a fluência é **retornar** `this` em cada método de configuração.

# Finalizando a construção

```
public Pedido build() {
  return new Pedido(cliente, endereco, desconto);
}
```

Por fim, o método `build()` cria o objeto final usando os valores configurados.

# Uso do Builder fluente

```
Pedido pedido = new Pedido.Builder()
  .comCliente("Aluno Dev")
  .comEndereco("Campus Central")
  .comDesconto(0.1)
  .build();
```

- Leitura quase “natural”
- Fácil enxergar o que foi configurado
- Valores opcionais podem ser omitidos

# Lançando exceções

Para parâmetros obrigatórios, o Builder pode lançar exceção se não forem configurados:

```
public Pedido build() {
  if (cliente == null || endereco == null) {
    throw new IllegalStateException("Cliente e endereço são obrigatórios");
  }
  return new Pedido(cliente, endereco, desconto);
}
```

# Quando usar Builder?

- Muitos parâmetros, especialmente **opcionais**
- Objetos **imutáveis** com várias combinações de criação
- Processo de construção tem **vários passos**
- Precisa de **API fluente** (código muito legível na montagem)

Exemplos típicos:

- Requisições HTTP complexas (OkHttp `Request.Builder`)
- Objetos de configuração de cliente REST
- Relatórios com muitas opções

# Prós

- Código de criação **mais legível**
- Ajuda a manter objetos **imutáveis**
- Evita construtor telescópico
- Fácil criar **variações** mudando o Builder
- Reduz risco de objeto ficar em estado inválido

## Contras

- Mais classes / mais código “de estrutura”
- Pode ser overengineering para objetos simples
- Exige disciplina para manter o Builder atualizado com o Produto

Regra: **vale a pena** quando o objeto é realmente **complexo**.

# Relação com outros padrões

- **Factory Method / Abstract Factory**
  - Fábricas criam objetos “de uma vez”
  - Builder cria “passo a passo”
- **Prototype**
  - Prototype copia um objeto pronto
  - Builder monta do zero com variações
- **Fluent Interface**
  - Muito usada junto com Builder para encadear chamadas

# Resumo

- Builder ajuda a construir **objetos complexos** de forma:
  - passo a passo
  - legível
  - flexível
- Use quando o construtor ficar **confuso** ou **gigante**
- Prefira para objetos ricos em configurações e combinações