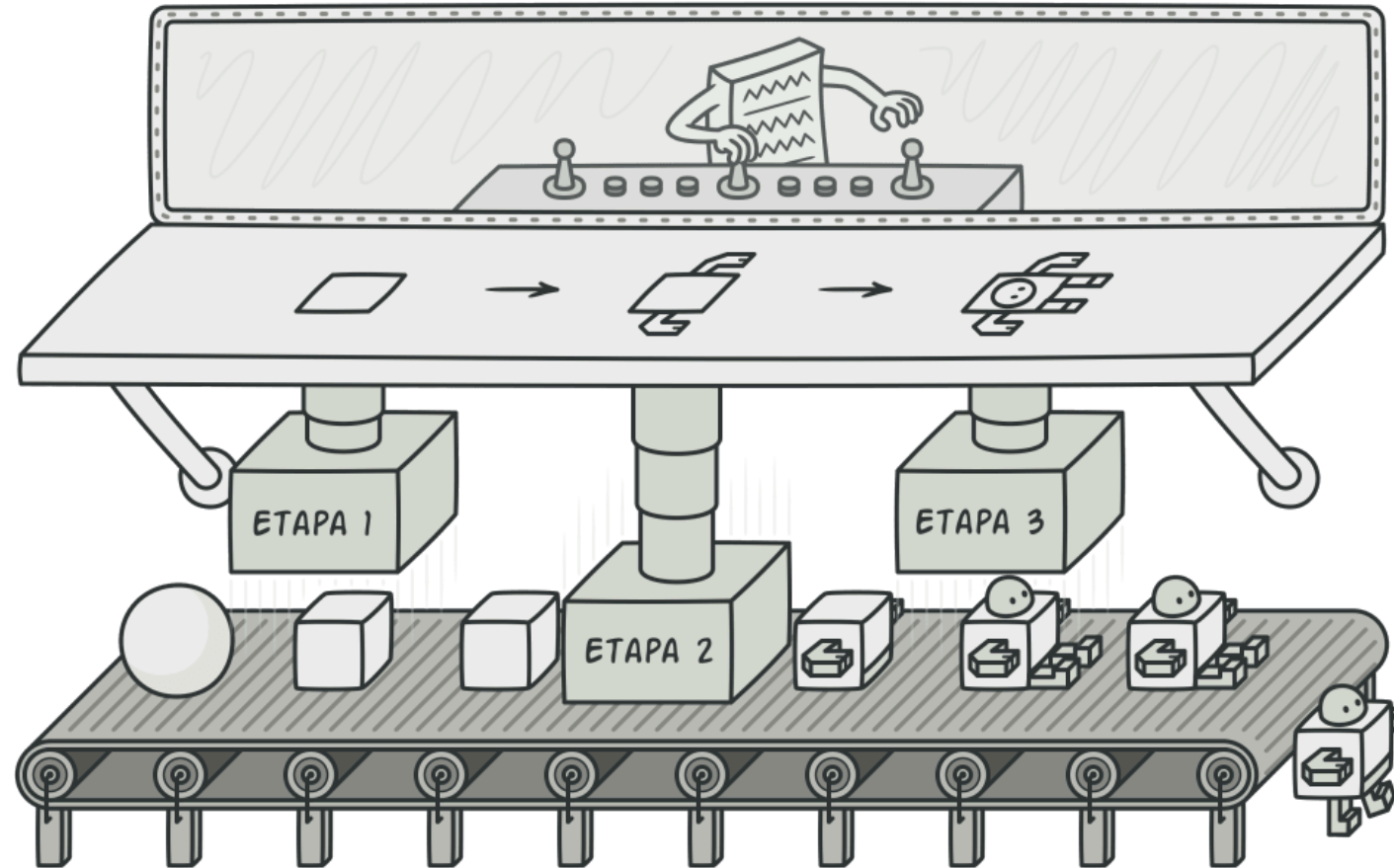


TÓPICO 04: BUILDER

Design Patterns - Professor Ramon Venson - SATC 2026.2

Definição

Builder é um padrão de projeto **criacional** que permite construir **objetos complexos passo a passo**, evitando construtores gigantes e melhorando a **legibilidade** e a **flexibilidade** na criação de objetos.



Problema

Alguns objetos possuem **muitos campos**, e grande parte deles pode ser **opcional**. Quando cada módulo precisa instanciar a classe diretamente, o código de criação tende a ficar:

- difícil de ler
- fácil de errar a ordem dos argumentos
- pior a cada novo campo adicionado

Neste exemplo, temos um exemplo de criação de um objeto `Pedido` com **muitos parâmetros**:

```
public class RelatorioService {  
    public void gerarRelatorio() {  
        Pedido pedido = new Pedido(  
            cliente, enderecoEntrega, itens, desconto, cupom, frete, observacoes  
        );  
        // ...  
    }  
}
```

Uma das soluções sugere criar vários **construtores sobrecarregados**:

```
public class Pedido {  
    public Pedido(String cliente, String endereco) { ... }  
    public Pedido(String cliente, String endereco, List<Item> itens) { ... }  
    public Pedido(String cliente, String endereco, List<Item> itens, double desconto) { ... }  
    public Pedido(String cliente, String endereco, List<Item> itens, double desconto, String cupom) { ... }  
    // ...  
}
```

Porém isso não é o ideal, pois a quantidade de construtores cresce rapidamente a cada novo parâmetro, dificultando a leitura e a manutenção do código.

Solução

Vamos construir o objeto **passo a passo**, separando **o que** é montado (`Product`) de **como** é montado (`Builder`).

- **Construtor privado**: apenas o `Builder` instancia o objeto
- **Builder interno** (`static`): guarda os valores configurados
- **Métodos de configuração** (`setCliente` , `setEndereco`)
- **Método** `build()` : cria o objeto final, validando campos obrigatórios

Implementação

1. Vamos criar a classe `Pedido` deixando o construtor **privado**, garantindo que apenas o `Builder` possa criar novas instâncias:

```
public class Pedido {  
    private final String cliente;  
    private final String endereco;  
    private final double desconto;  
  
    // Construtor privado impede instanciação externa  
    private Pedido(String cliente, String endereco, double desconto) {  
        this.cliente = cliente;  
        this.endereco = endereco;  
        this.desconto = desconto;  
    }  
}
```

2. Em seguida, criamos o **Builder interno** como uma classe `static` dentro de `Pedido`. Ele terá os mesmos campos e métodos de configuração:

```
public static class Builder {  
    private String cliente;  
    private String endereco;  
    private double desconto;  
}
```

Eventualmente, o Builder pode ser uma classe separada, mas a abordagem interna é comum para manter a coesão.

3. Definimos os métodos de configuração que permitem alterar os valores dos campos:

```
public void setCliente(String cliente) {  
    this.cliente = cliente;  
}  
  
public void setEndereco(String endereco) {  
    this.endereco = endereco;  
}  
  
public void setDesconto(double desconto) {  
    this.desconto = desconto;  
}
```

4. Por fim, implementamos o método `build()` que cria o objeto final, validando os campos obrigatórios:

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

Fluent Builder

Uma forma comum de implementar o Builder é usando **métodos encadeados**:

```
public Builder comCliente(String cliente) {  
    this.cliente = cliente;  
    return this;  
}  
  
public Builder comEndereco(String endereco) {  
    this.endereco = endereco;  
    return this;  
}  
  
public Builder comDesconto(double desconto) {  
    this.desconto = desconto;  
    return this;  
}
```

Isso permite criar o objeto de forma **fluyente**, encadeando os métodos de configuração:

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

Além disso, a linguagem usada nos métodos (`comCliente` , `comEndereco`) ajuda a tornar o código próximo do **domínio do problema**, facilitando a compreensão.

Director

O `Director` é uma classe opcional que pode controlar a ordem de construção do objeto, especialmente útil quando há **passos complexos** ou **sequências específicas**.

```
public class PedidoDirector {  
    public Pedido criarPedidoCompleto() {  
        return new Pedido.Builder()  
            .comCliente("Aluno Dev")  
            .comEndereco("Campus Central")  
            .comDesconto(0.1)  
            .build();  
    }  
}
```

Director

O **Director** é uma classe **opcional** que controla a ordem de construção do objeto e seus parâmetros, útil quando há **passos complexos** ou **sequências específicas**.

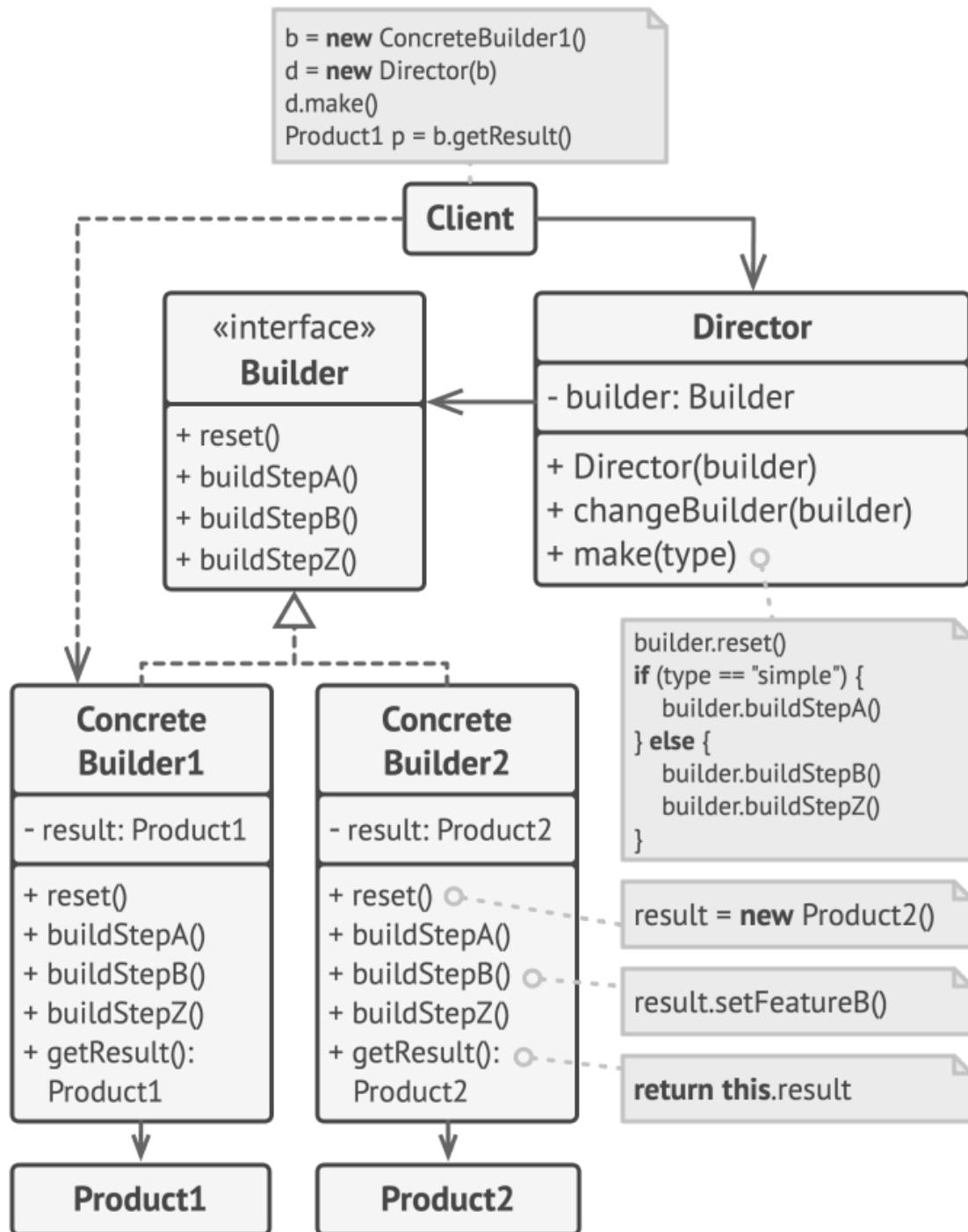
```
public class PedidoDirector {  
    public Pedido criarPedidoPromocional(Builder builder, String cliente, String endereco) {  
        return new Pedido.Builder()  
            .comCliente(cliente)  
            .comEndereco(endereco)  
            .comDesconto(0.25)  
            .build();  
    }  
}
```

Utilização

O método `build()` cria o objeto final usando os valores configurados no Builder:

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

- Leitura natural e encadeada
- Fácil enxergar o que foi configurado
- Valores opcionais podem ser omitidos



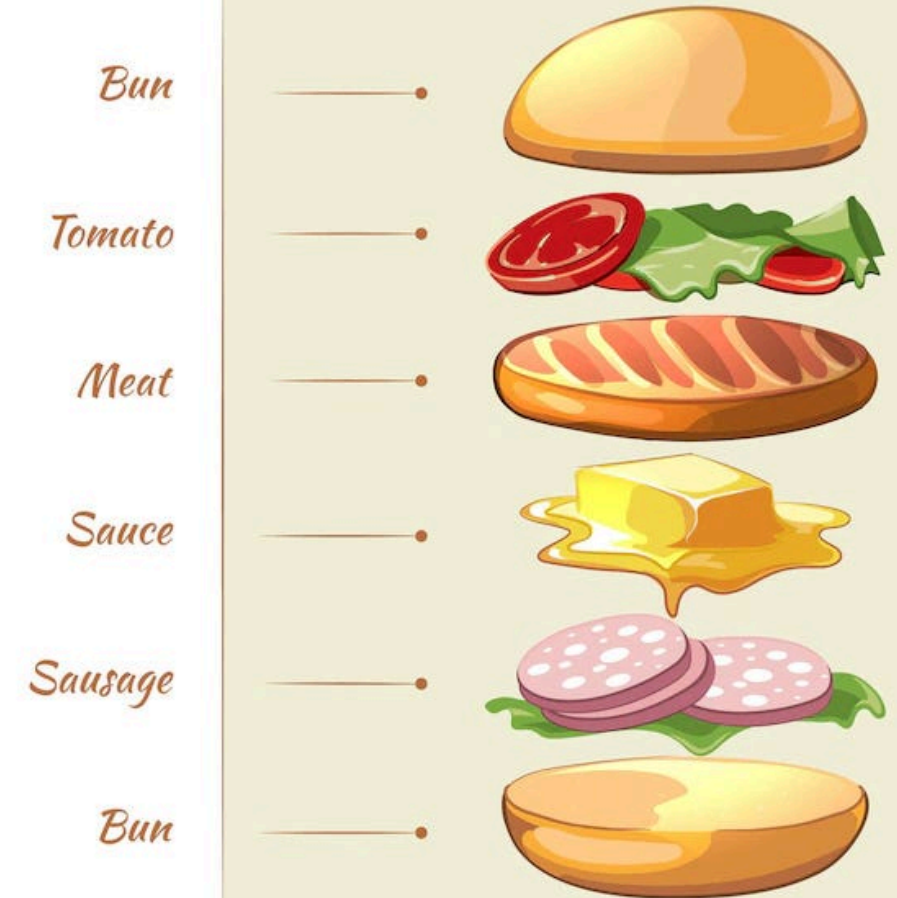
Diagramas

A estrutura envolve:

- **Product**: objeto final a ser construído
- **Builder**: abstração com os passos de construção
- **Concrete Builder**: implementações para variações
- **Director** (opcional): controla a ordem de construção

Analogias

- **Sanduíche:**
 - O sanduíche é o **Produto**, e quem o prepara é o **Builder**
 - Montagem em **passos claros**: pão, proteína, queijo, vegetais, molhos
 - As mesmas etapas podem gerar **variações** diferentes



Aplicações

- Objetos com **muitos parâmetros**, principalmente **opcionais**
- Objetos **imutáveis** com várias combinações de criação
- Processos de construção com **vários passos**
- Quando a montagem precisa de uma **API fluente** (código legível)

Exemplos típicos:

- Requisições HTTP complexas (OkHttp `Request.Builder`)
- Configuração de clientes REST
- Relatórios com muitas opções

Prós

- **Código de criação legível:** encadeamento quase "natural"
- **Imutabilidade:** facilita manter o objeto imutável após a construção
- **Sem construtor telescópico:** evita construtores gigantes e sobrecargas
- **Variações:** fácil criar combinações diferentes sem novos construtores
- **Estados válidos:** valida os obrigatórios antes de criar o objeto

Contras

- **Mais código de estrutura:** exige classes ou métodos extras
- **Overengineering:** pode ser desnecessário para objetos simples
- **Sincronização com o Produto:** exige disciplina para manter o Builder atualizado quando o Produto muda

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

Resumo

- Builder: **objeto complexo construído passo a passo**
- Útil quando o construtor ficar **confuso** ou **gigante**
- Facilita objetos **imutáveis** e cheios de opcionais
- Evite o exagero: para objetos simples, use o construtor tradicional