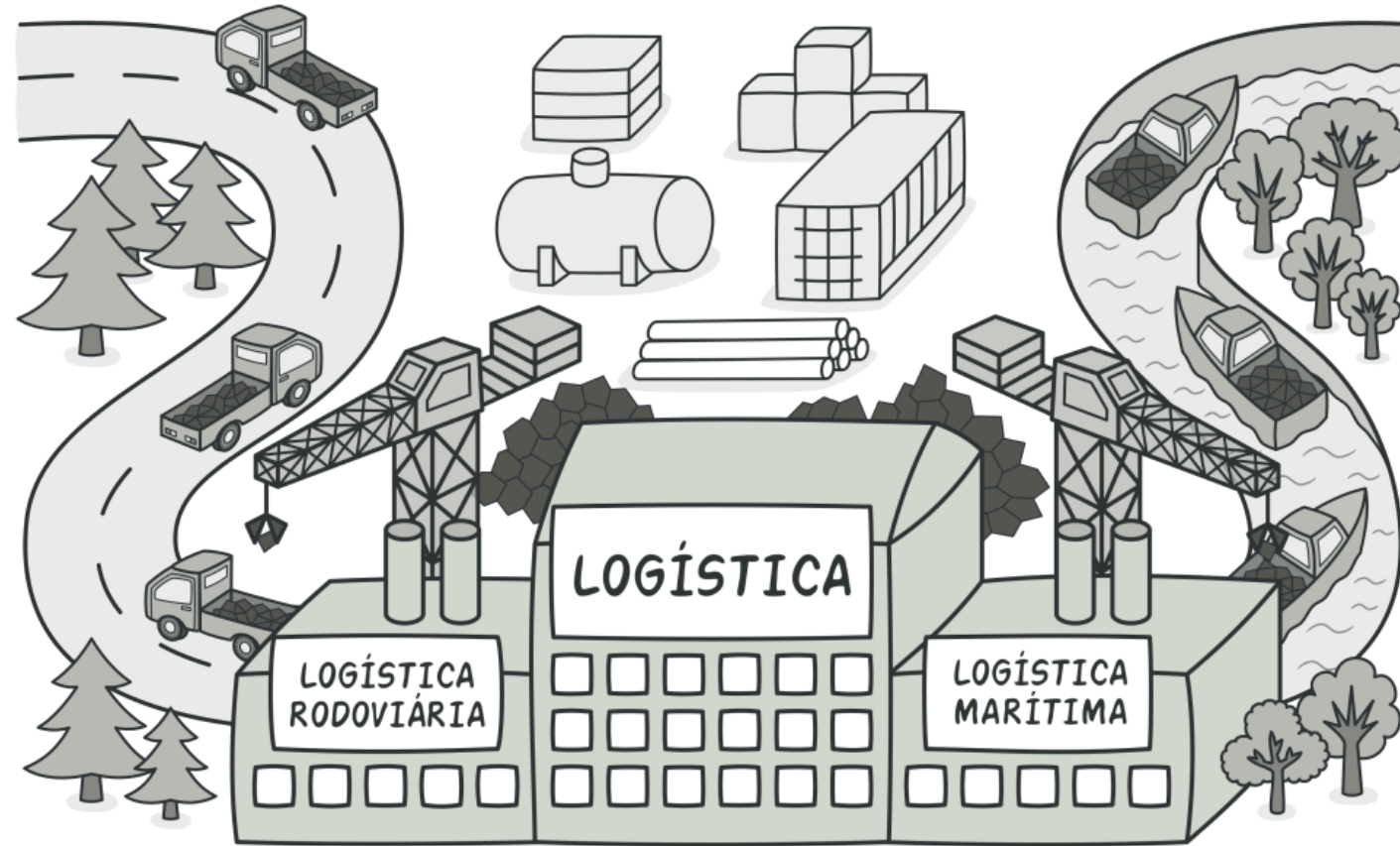


TÓPICO 06: FACTORY METHOD

Design Patterns - Professor Ramon Venson - SATC 2026.2

Definição

Factory Method é um padrão de projeto **criacional** que define uma **interface para criar objetos** em uma superclasse, mas permite que **as subclasses decidam** qual classe concreta instanciar.



Problema

Nem sempre sabemos **de antemão** qual objeto concreto precisamos criar. Quando o código cria objetos diretamente com `new`, ele fica **acoplado à classe concreta** e qualquer novo tipo exige:

- alterar o código que **já funciona**
- espalhar `switch` / `if-else` pelo projeto
- repetir a mesma lógica de escolha em vários lugares



Neste exemplo, a logística escolhe o transporte com um `switch`. Cada novo meio exige mexer nessa classe **testada**:

```
public class Logistica {  
    public void planejarEntrega(String tipo) {  
        Transporte transporte;  
        switch (tipo) {  
            case "rodoviaria": transporte = new Caminhao(); break;  
            case "maritima":    transporte = new Navio();    break;  
            default: throw new IllegalArgumentException(tipo);  
        }  
        transporte.entregar();  
    }  
}
```

Pior: a mesma escolha costuma se **repetir em vários pontos** do sistema, cada um com seu próprio `new`:

```
public class NotaFiscalService {  
    public void emitir() {  
        Transporte transporte = new Caminhao(); // e se mudar?  
        BigDecimal valorFrete = transporte.calcularFrete();  
    }  
}
```

- adicionar um transporte novo exige "**caçar o new**"
- o cliente **conhece a classe concreta**
- o código fica difícil de estender sem introduzir bugs

Solução

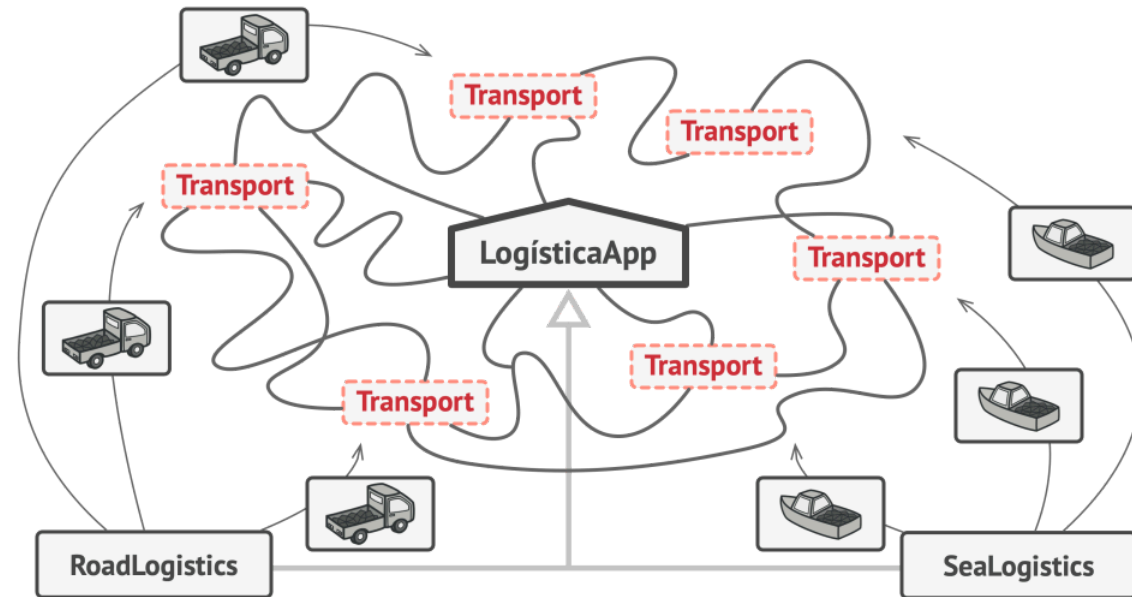
Vamos **encapsular a criação** em um **método fábrica**: o código cliente passa a trabalhar apenas com a **abstração** do produto, e as **subclasses do criador** decidem qual produto concreto retornar.

- **Produto** (interface): o que o cliente usa
- **Criador**: define o processo + o método fábrica
- **Criador concreto**: sobrescreve o método fábrica
- O cliente **não usa** `new` de classes concretas

O processo **não muda** — muda apenas o **que é criado**:

- `planejarEntrega()` segue igual
- cada subclasse do criador decide o **transporte**
- adicionar um transporte novo = **nova subclasse**, sem tocar no processo

A decisão de "qual classe concreta" fica **isolada** em um único método.



Implementação

1. Primeiro, vamos definir a **interface do produto**, que é o contrato que todos os transportes seguem:

```
public interface Transporte {  
    void entregar();  
}
```

2. Em seguida, criamos os **produtos concretos**, cada um com seu comportamento:

```
public class Caminhao implements Transporte {  
    @Override  
    public void entregar() {  
        System.out.println("Entrega por caminhão");  
    }  
}  
  
public class Navio implements Transporte {  
    @Override  
    public void entregar() {  
        System.out.println("Entrega por navio");  
    }  
}
```

3. Agora a classe **criadora**: ela contém a lógica principal (`planejarEntrega`) e declara o **método fábrica** abstrato (`criarTransporte`):

```
public abstract class Logistica {  
  
    public void planejarEntrega() {  
        Transporte transporte = criarTransporte(); // factory method  
        // ...validações, cálculo de rota e custos...  
        transporte.entregar();  
    }  
  
    protected abstract Transporte criarTransporte();  
}
```

4. Por fim, os **criadores concretos** sobrescrevem o método fábrica, devolvendo o produto certo:

```
public class LogisticaRodoviaria extends Logistica {
    @Override
    protected Transporte criarTransporte() {
        return new Caminhao();
    }
}

public class LogisticaMaritima extends Logistica {
    @Override
    protected Transporte criarTransporte() {
        return new Navio();
    }
}
```

Utilização

O cliente escolhe **qual criador** usar e o processo acontece sozinho:

```
Logistica rodoviaria = new LogisticaRodoviaria();  
rodoviaria.planejarEntrega(); // Entrega por caminhão  
  
Logistica maritima = new LogisticaMaritima();  
maritima.planejarEntrega(); // Entrega por navio
```

- `planejarEntrega()` **não muda**
- para adicionar um transporte (ex.: `Aviao`):
 - crie `Aviao implements Transporte`
 - crie `LogisticaAerea extends Logistica`
- sem reescrever o fluxo nem "caçar `new`" no projeto

Simple Factory

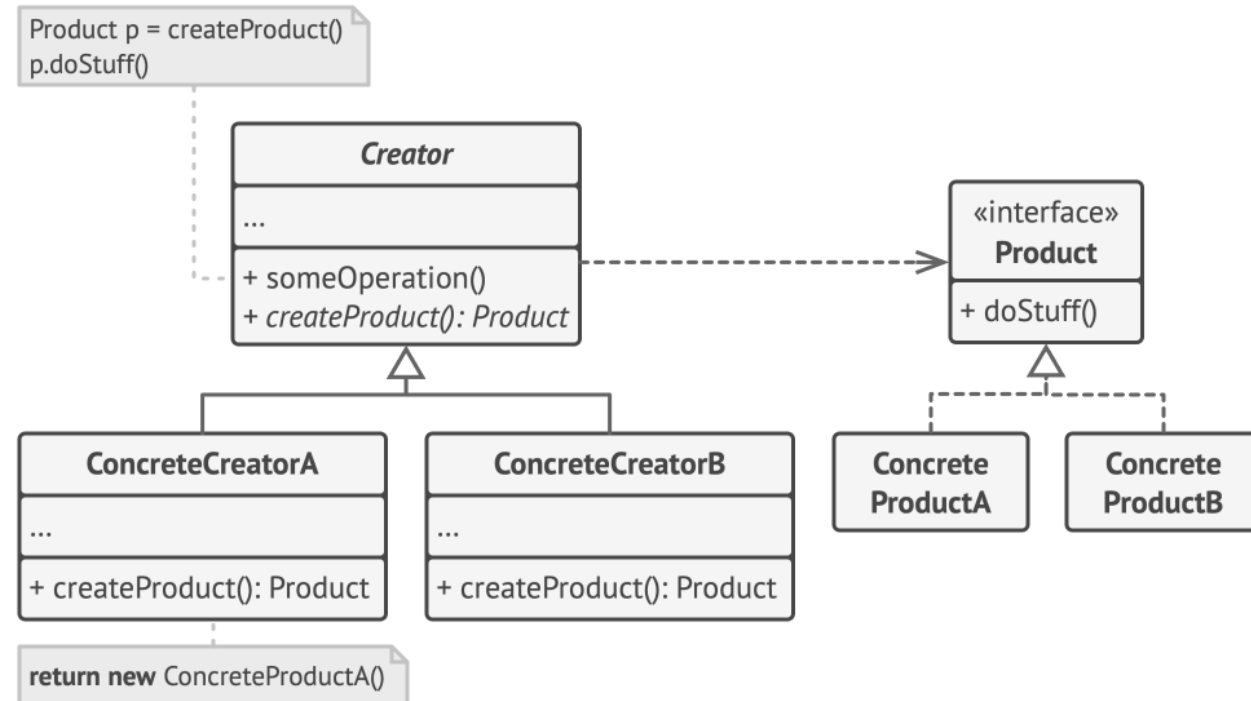
O Simple Factory é uma **variação** do Factory Method, mas **não é um padrão**. Ele centraliza a criação em uma **classe concreta**, que decide qual produto criar:

```
public class SimpleFactory {  
    public static Transporte criarTransporte(String tipo) {  
        switch (tipo) {  
            case "rodoviaria": return new Caminhao();  
            case "maritima":   return new Navio();  
            default: throw new IllegalArgumentException(tipo);  
        }  
    }  
}
```

Diagramas

A estrutura envolve:

- **Product**: interface comum dos objetos criados
- **ConcreteProduct**: implementações do produto
- **Creator**: declara o método fábrica e usa o Product
- **ConcreteCreator**: sobrescreve o método fábrica



Analogias

Rede de restaurantes:

- O processo de atendimento é o mesmo em todas as lojas
- cada loja (hamburgueria, pizzaria) **monta o seu produto**
- quem pede não precisa saber **como** o prato é preparado
- O "criador" reaproveita o **fluxo comum** e cada filial decide o **produto final**

Aplicações

- Uma classe **não deve conhecer** as classes concretas que instancia
- A criação varia conforme:
 - ambiente (dev/prod, mock ou real)
 - tipo de entrada (JSON/XML)
 - regra de negócio (região, cliente, plano)

Exemplos típicos:

- escolher um `Parser` (`JsonParser` , `XmlParser`)
- escolher um `Notificador` (`Email` , `SMS` , `Push`)

Prós

- **Menor acoplamento:** o cliente depende da abstração, não do concreto
- **Facilita estender:** novos produtos entram com novas subclasses
- **Centraliza a criação:** você controla "onde nasce" cada objeto
- Ajuda a aplicar o **OCP** (Open/Closed)

Contras

- **Mais classes:** criadores + produtos para cada variação
- Pode ser **overengineering** para casos simples
- Se exagerar, vira uma "floresta de fábricas" sem necessidade real

Regra: use quando a **variação de criação** for real e frequente.

Resumo

- Factory Method: **deixe subclasses decidirem o que instanciar**
- O método fábrica **isola a criação** do uso do objeto
- Ganho prático: **extensão com menos risco**, fluxo principal estável
- Cuidado para não criar fábricas **sem necessidade** real