TÓPICO 06: FACTORY METHOD

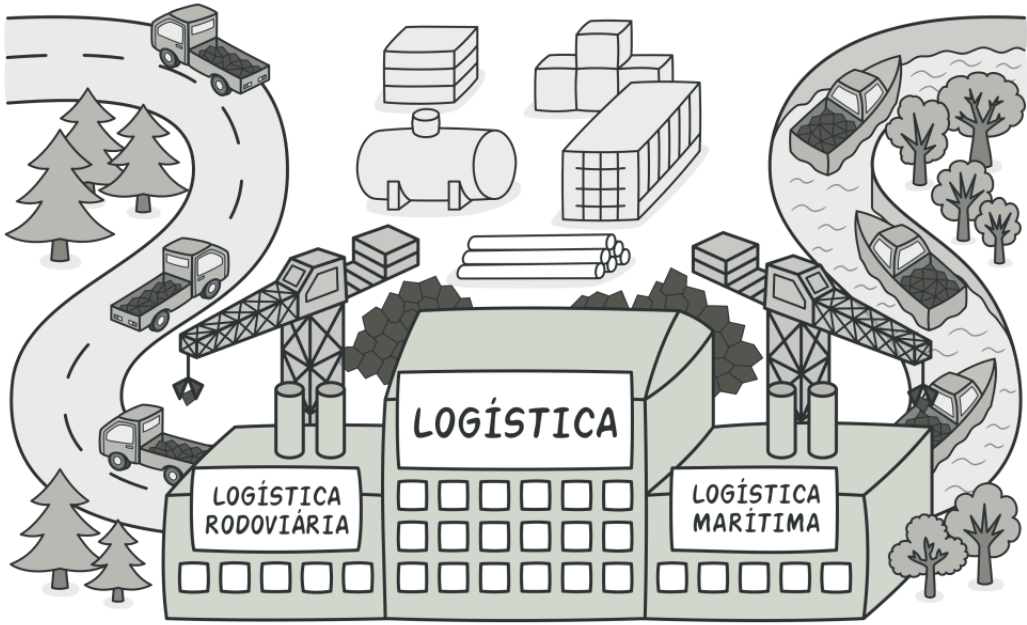
Design Patterns - Professor Ramon Venson - SATC 2026.1

O que é Factory Method?

Padrão **criacional** que:

- Define um **método de criação** (factory method)
- Deixa **subclasses decidirem** qual classe concreta instanciar
- Faz o código cliente depender de **abstrações**, não de `new Concreto()`

Objetivo: **desacoplar criação** do uso.



Problema: acoplamento ao concreto

```
public void planejarEntrega() {
  Caminhao c = new Caminhao();
  c.entregar();
}
```

Quando surge `Navio`, `Aviao`, etc.:

- aparece `if/else` / `switch`
- alterações frequentes em código já testado
- difícil testar (sem mocks)
- extensão vira “caçar `new` no projeto”

Ideia principal

Separar:

- **fluxo principal** (o que fazer)
- **decisão de criação** (qual objeto usar)

O Creator chama:

- `criarTransporte()` (factory method)
- e usa o resultado via interface `Transporte`

Analogia: Logística

- Mesmo processo:
    i. planejar entrega
    ii. escolher transporte
    iii. entregar
- Variações:
    - Rodoviária → `Caminhao`
    - Marítima → `Navio`

Mesma lógica, **produto criado muda**.

Estrutura (visão geral)

- **Product**: `Transporte`
- **ConcreteProduct**: `Caminhao`, `Navio`
- **Creator**: `Logistica` (tem `planejarEntrega()`)
- **ConcreteCreator**: `LogisticaRodoviaria`, `LogisticaMaritima`

Exemplo (1/2) — Product + Concretos

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

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

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

Exemplo (2/2) — Creator + Subclasses

```
public abstract class Logistica {
  public void planejarEntrega() {
    Transporte t = criarTransporte();
    t.entregar();
  }
  protected abstract Transporte criarTransporte();
}

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

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

Uso

```
Logistica l1 = new LogisticaRodoviaria();
l1.planejarEntrega();

Logistica l2 = new LogisticaMaritima();
l2.planejarEntrega();
```

- `planejarEntrega()` não muda
- o tipo concreto é decidido pela subclasse

Quando usar?

- seu código precisa criar objetos, mas **não deve conhecer** classes concretas
- há chance real de **novos tipos** surgirem
- você quer aplicar **OCP** (estender sem modificar)
- criação tem regras/variações por ambiente, por cliente, por contexto

Uso com o Builder

```
LogisticaFactory factory = new LogisticaRodoviariaFactory();
factory.caminhao()
       .setCapacidade(1000)
       .setDestino("Cidade A")
       .build()
       .entregar();
```

- Factory Method pode criar Builders
- Ex.: `Logistica` pode ter `criarBuilder()` que retorna um `TransporteBuilder`
- O Builder pode ter variações (ex.: `CaminhaoBuilder`, `NavioBuilder`)
- Combinação poderosa para objetos complexos com variações

Prós

- Menos acoplamento a classes concretas
- Facilita extensão (novos produtos)
- Centraliza decisão de criação
- Melhora testabilidade (depender de abstrações)

Contras

- Mais classes (Creator/ConcreteCreator)
- Pode ser overengineering para casos simples
- Requer disciplina para manter a estrutura clara

Relação com outros padrões

- **Abstract Factory**: famílias de objetos (nível acima)
- **Template Method**: o Creator parece um “template”, e o factory method é ponto de variação
- **Builder**: constrói passo a passo; Factory Method cria “de uma vez”

Resumo

- Factory Method = **subclasses escolhem o que instanciar**
- Melhora:
  - extensibilidade
  - desacoplamento
  - manutenção
- Use quando houver variação real de “produtos” e risco de mudanças futuras