Pontifícia Universidade Católica de Minas Gerais
Sistemas de Informação

Aluno: Arthur Monteiro Parreiras

Brenda Stefany de Oliveira Rocha

Bruna Letícia Silva

Eduarda Farinha Pinheiro

Elenice Florentina de Oliveira Reis

Marco Aurélio de Faria Ramos

Factory Method

O que é e por que usar
O Factory Method é um padrão de criação que define uma interface para criar objetos, permitindo que subclasses decidam qual classe concreta instanciar. Ele substitui o uso direto de new por um método de fábrica, removendo o acoplamento do cliente às classes concretas e favorecendo extensão sem modificar código existente (OCP).

Problema do new e princípio aberto/fechado (OCP)
Quando o cliente usa new (ex.: `new ProdutoConcretoA()`), ele fica acoplado à classe concreta. Trocar a implementação exige alterar todos os pontos que fazem new, violando o OCP. O Factory Method centraliza a decisão de criação em um ponto polimórfico.

Quando usar
- Você não sabe de antemão qual classe concreta instanciar.
- Seu código deve trabalhar contra interfaces, não implementações.
- Quer permitir que subclasses escolham que produto criar.
- Precisa evoluir com novos tipos sem modificar o cliente.

Estrutura
- Product: interface/contrato dos objetos criados.
- ConcreteProduct: implementações do produto.
- Creator (fábrica base): declara o `factoryMethod()`; pode conter lógica de alto nível que usa Product.
- ConcreteCreator: sobrescreve `factoryMethod()` para decidir que ConcreteProduct criar.

Exemplo em Java
// Produto

public interface Transporte { void entregar(); }

// Produtos concretos

public class Caminhao implements Transporte {

@Override public void entregar() { System.out.println("Entrega por caminhão (terrestre)."); }

```
}

public class Navio implements Transporte {

  @Override public void entregar() { System.out.println("Entrega por navio
(marítima)."); }

}

public class Aviao implements Transporte {

  @Override public void entregar() { System.out.println("Entrega por avião
(aérea)."); }

}


// Creator

public abstract class Logistica {

  // Factory Method

  protected abstract Transporte criarTransporte();

  // Lógica de alto nível que usa o produto

  public void planejarEntrega() {

    Transporte t = criarTransporte();

    // ... regras comuns (roteirização, cálculo de custo etc.) ...

    t.entregar();

  }

}


// Concrete Creators

public class LogisticaTerrestre extends Logistica {

  @Override protected Transporte criarTransporte() { return new Caminhao(); }
```

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

// NOVO tipo adicionado sem tocar no código de Logistica/cliente
public class LogisticaAerea extends Logistica {
  @Override protected Transporte criarTransporte() { return new Aviao(); }
}

// Cliente
public class Main {
  public static void main(String[] args) {
    Logistica terrestre = new LogisticaTerrestre();
    Logistica maritima = new LogisticaMaritima();
    Logistica aerea    = new LogisticaAerea(); // extensão natural

    terrestre.planejarEntrega();
    maritima.planejarEntrega();
    aerea.planejarEntrega();
  }
}
```

Vantagens

- Remove acoplamento ao concreto e melhora testabilidade.
- Permite adicionar novos produtos via novas fábricas (extensibilidade).
- Centraliza a decisão de criação, favorecendo coesão.

Desvantagens

- Mais classes (Creators/Products).
- Depuração pode exigir navegar por polimorfismo.

Abstract Factory

O que é por que usar

Cria famílias de produtos relacionados compatíveis sem expor classes concretas. Permite trocar a família inteira (tema/sistema de widgets, drivers, provedores) sem alterar o cliente (OCP).

Quando usar

- Há múltiplas variantes compatíveis (ex.: Windows, Linux, Mac; SQL, NoSQL).
- É necessário manter coerência entre produtos da mesma família.
- Deseja alternar famílias por configuração ou em tempo de execução.

Estrutura (papéis)

- AbstractFactory: operações para criar cada tipo de produto da família.
- ConcreteFactoryX: fabrica a família específica.
- AbstractProductY: contratos dos produtos (ex.: Botao, Janela).
- ConcreteProductY: variantes por família.
- Cliente: usa apenas interfaces abstratas.

Exemplo em Java

```
interface Botao { void desenhar(); }

interface CaixaTexto { void exibir(); }


class BotaoWin implements Botao { public void desenhar(){
System.out.println("Botão Win"); } }

class CaixaTextoWin implements CaixaTexto { public void exibir(){
System.out.println("Caixa Win"); } }

class BotaoLinux implements Botao { public void desenhar(){
System.out.println("Botão Linux"); } }

class CaixaTextoLinux implements CaixaTexto { public void exibir(){
System.out.println("Caixa Linux"); } }
```

```
interface UIFactory {
  Botao criarBotao();
  CaixaTexto criarCaixaTexto();
}
class WinFactory implements UIFactory {
  public Botao criarBotao(){ return new BotaoWin(); }
  public CaixaTexto criarCaixaTexto(){ return new CaixaTextoWin(); }
}
class LinuxFactory implements UIFactory {
  public Botao criarBotao(){ return new BotaoLinux(); }
  public CaixaTexto criarCaixaTexto(){ return new CaixaTextoLinux(); }
}

// Cliente
UIFactory f = /* resolver por config */ new WinFactory();
Botao b = f.criarBotao(); CaixaTexto c = f.criarCaixaTexto();
b.desenhar(); c.exibir();
```

Vantagens

- Consistência entre produtos; baixo acoplamento.
- Troca de famílias sem tocar no cliente.

Desvantagens

- Adicionar novo tipo de produto exige alterar todas as fábricas.
- Pode parecer pesado se você tem poucas variantes.

Builder

O que é e por que usar
Separa a montagem da representação de um objeto complexo. Útil para muitos parâmetros, opcionais e construção fluente com validações concentradas no build().

Quando usar
- Construtores com muitos argumentos (alguns opcionais).
- Múltiplas representações do mesmo produto.
- Deseja imutabilidade após construção.

Exemplo em Java
```
public class Pedido {

  private final String cliente; private final boolean express; private final String endereco;

  private Pedido(String c, boolean e, String end){ this.cliente=c; this.express=e; this.endereco=end; }

  public static class Builder {

    private String cliente; private boolean express; private String endereco;

    public Builder cliente(String v){ this.cliente=v; return this; }

    public Builder express(boolean v){ this.express=v; return this; }

    public Builder endereco(String v){ this.endereco=v; return this; }

    public Pedido build(){

      if(cliente == null || cliente.isBlank()) throw new IllegalArgumentException("cliente obrigatório");

      return new Pedido(cliente, express, endereco);

    }

  }

}

var p = new Pedido.Builder().cliente("Arthur").express(true).endereco("Rua X").build();
```

Vantagens
- Criação legível e validada.
- Facilita imutabilidade.
- Evolução segura com novos campos.

Desvantagens
- Mais classes/boilerplate.
- Cuidado com estados inválidos sem validação.

Prototype

O que é e por que usar
Cria objetos clonando um protótipo existente. Indicado quando a criação é cara/complexa ou quando há muitas combinações de estado derivadas de exemplos.

Exemplo em Java

```
class Relatorio implements Cloneable {

  private String titulo; private java.util.List<String> itens;

  public Relatorio(String t, java.util.List<String> i){ this.titulo=t; this.itens=i; }

  @Override public Relatorio clone(){

    try {

      Relatorio c = (Relatorio) super.clone();

      c.itens = new java.util.ArrayList<>(this.itens); // cópia profunda da coleção

      return c;

    } catch (CloneNotSupportedException e) { throw new AssertionError(e); }

  }

}

Relatorio base = new Relatorio("Base", java.util.List.of("A","B"));

Relatorio copia = base.clone();
```

Vantagens
- Criação rápida de objetos quando a construção “do zero” é cara (I/O, cálculos, hidratação complexa).

- Pode prescrever Configurações inciais padrão em um registry de protótipos

Desvantagens
- Manutenção de um registry de protótipos pode introduzir estado global implícito.
- Semântica de clonagem mal definida gera bugs sutis.

Singleton

O que é e por que usar
Garante uma única instância com acesso global controlado. Útil para recursos únicos (config, log), mas use com parcimônia para não introduzir estado global difícil de testar.

Exemplo em Java

```
// Lazy + thread-safe

public class Config {

  private static volatile Config INSTANCE;

  private Config(){}

  public static Config getInstance(){

    if(INSTANCE == null){

      synchronized(Config.class){

        if(INSTANCE == null) INSTANCE = new Config();

      }

    }

    return INSTANCE;

  }

}

// Via enum (simples e seguro para serialização)
```

public enum LogGlobal { INSTANCE; public void info(String m){
System.out.println(m); } }

Vantagens

- Garante uma única instância com ponto de acesso global controlado.
- Útil para recursos centrais/compartilhados.

Desvantagens

- Requer cuidado com threead-safety e serialização.
- Pode mascarar problemas de design que seriam melhor resolvidos com injeção de dependências.