Padrões de Projeto (Design Patterns)

Teoria e Prática: Singleton & Factory Method

Camila Paranhos, Filipe Rubson, Patrícia Silva, Thalyton Vieira, Valdir Neto

5 de dezembro de 2025

Disciplina: Linguagem de Programação I

Introdução aos Padrões

O problema da “Roda Reinventada”

Imagine que você é um chef de cozinha. Toda vez que alguém pede um bolo de chocolate, você tenta inventar a receita do zero.

Resultado:

- ✖ Demora muito.
- ✖ O gosto nunca é o mesmo.
- ✖ Outros cozinheiros não te entendem.

O que são Design Patterns?

Definição Formal

São soluções típicas e reutilizáveis para problemas comuns de design de software orientado a objetos.

Por que estudar isso?

1. **Vocabulário Universal:** Em vez de explicar 20 linhas de código, você diz: *"Usei um Singleton aqui"*.
2. **Qualidade Comprovada:** São soluções testadas por milhares de desenvolvedores ao longo de décadas.
3. **Manutenibilidade:** O código fica mais limpo e desacoplado.

O Catálogo GoF (Gang of Four)

Os padrões são divididos em 3 famílias. Hoje focaremos nos **Criacionais**.

1. Criacionais
Como os objetos são criados?
- → **Singleton**
- → **Factory Method**
- → Builder
- → Prototype

2. Estruturais
Como as classes são montadas?
- → Adapter
- → Facade
- → Decorator

3. Comportamentais
Como eles conversam?
- → Strategy
- → Observer
- → Iterator

Padrão: Singleton

A Analogia do Singleton

O Governo de um País.
Um país pode ter muitos cidadãos, mas apenas **um** governo oficial. Independentemente de quem chame “o governo” , deve-se estar lidando sempre com a mesma entidade central.

No Software: Logs, Configurações, Drivers de Impressora, Pool de Conexão.

Singleton: Conceito Técnico

Objetivo

Garantir que uma classe tenha **apenas uma instância** e fornecer um **ponto global de acesso** a ela.

Como a mágica acontece?

- **Construtor Privado:** Ninguém fora da classe pode dar `new`.
- **Atributo Estático:** A própria classe guarda a única instância dela mesma.
- **Método Estático:** Entrega a instância guardada para quem pedir.

Singleton: Cuidado!

Nem tudo deve ser Singleton. Ele é controverso

👍 Vantagens

- Controle estrito sobre a criação.
- Economia de memória (reutiliza).
- Acesso fácil em qualquer parte do código.

👎 Desvantagens

- Viola o Princípio da Responsabilidade Única.
- **Difícil de Testar:** O estado global persiste entre testes unitários, causando bugs fantasmas.
- Pode mascarar dependências ruins.

Singleton na Prática (Java)

Configuracao.java

```java
public class Configuracao {
    // 1. Atributo estático que guarda a inst ncia única
    private static Configuracao instance;

    private String apiUrl = "http://api.ifmg.edu.br" ;

    // 2. Construtor privado (ninguém faz new Configuracao())
    private Configuracao() {
        System.out.println("Carregando configs...");
    }

    // 3. Método público estático para acesso global
    public static Configuracao getInstance() {
        if (instance == null) {
            instance = new Configuracao(); // Cria só na 1   vez
        }
```

Singleton na Prática (Java)

Continuação...

```
    return instance;
    }

    public String getUrl() {
        return apiUrl;
    }
}
```

Testando o Singleton

Configuracao.java

```java
public class Main {
    public static void main(String[] args) {
        // Configuracao c = new Configuracao(); // ERRO!

        Configuracao c1 = Configuracao.getInstance();
        Configuracao c2 = Configuracao.getInstance();

        // Prova real: c1 e c2 são o MESMO objeto na memória?
        if (c1 == c2) {
            System.out.println("E a mesma instancia! Singleton funcionou." );
        }
    }
}
```

Padrão: Factory Method

O Problema do “Acoplamento”

Imagine um aplicativo de Logística que só fazia entregas de **Caminhão**.

```
Transporte t = new Caminhao();
```

O app cresceu. Agora precisamos entregar de **Navio**, **Bicicleta** e **Drone**.

O pesadelo do IF:

```
if (tipo == “mar” ) { return new Navio(); }
else if (tipo == “ar” ) { return new Drone(); }
```

Se adicionar um novo transporte, você quebra o código existente para editar.

Factory Method: A Solução

Definição

Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar.

A ideia: “Não dê `new` diretamente. Peça para uma ’Fábrica’ criar para você.”

Isso separa o **código que usa** o produto do **código que cria** o produto.

Passo 1: Interface Comum

Todos os transportes devem seguir o mesmo “contrato” .

```java
// Interface comum (ou classe abstrata)
public interface Transporte {
    void entregar();
}

// Implementaç es  concretas
public class Caminhao implements Transporte {
    public void entregar() { System.out.println"Entrega via terra."); }
}

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

Passo 2: A Fábrica (Factory)

A classe que contém a regra de negócio não sabe qual transporte vai usar. Ela apenas pede à fábrica.

```java
// Classe Criadora (Base)
public abstract class Logistica {
    // O M TODO F BRICA
    public abstract Transporte criarTransporte();

    // Regra de negócio
    public void realizarEntrega() {
        // Note: Não existe 'new Caminhao()' aqui!
        Transporte t = criarTransporte();
        t.entregar();
    }
}
```

Passo 3: Fábricas Concretas

```java
// Fábrica especializada em Caminh es
public class LogisticaRodoviaria extends Logistica {
    @Override
    public Transporte criarTransporte() {
        return new Caminhao();
    }
}

// Fábrica especializada em Navios
public class LogisticaMaritima extends Logistica {
    @Override
    public Transporte criarTransporte() {
        return new Navio();
    }
}
```

Resumo Visual do Factory

Sem Factory: `App` $\rightarrow$ (depende de) $\rightarrow$ `Caminhão`, `Navio`, `Drone`...

Com Factory: `App` $\rightarrow$ (depende de) $\rightarrow$ `Transporte (Interface)`

Quem decide o `new` é a subclasse específica da Fábrica, não o código principal. Isso cumpre o **Open/Closed Principle** (Aberto para extensão, fechado para modificação).

Hora da Prática

Desafio em Sala: Sistema de Notificações

Cenário: Você está criando um sistema que envia notificações para usuários. Hoje o sistema envia **Email**, mas amanhã pode precisar enviar **SMS** ou **Push Notification**.

Atividade (10 min)

Implemente usando **Factory Method**:

1. Interface `Notificacao` (método `enviar(String msg)`).
2. Classes `EmailNotificacao` e `SMSNotificacao`.
3. Uma Factory que decide qual criar baseada em um input do usuário.

Gabarito do Desafio

Solução: Contrato e Implementações

Passo 1 e 2: Interface e Classes

```java
// Interface Comum
public interface Notificacao {
    void enviar(String mensagem);
}

// Implementacoes Concretas
public class EmailNotificacao implements Notificacao {
    public void enviar(String msg) {
        System.out.println("Enviando EMAIL: " + msg);
    }
}

public class SMSNotificacao implements Notificacao {
    public void enviar(String msg) {
        System.out.println("Enviando SMS: " + msg);
    }
}
```

Solução: A Factory (Decisora)

NotificacaoFactory.java

```java
public class NotificacaoFactory {

    public Notificacao criarNotificacao(String tipo) {
        if (tipo == null || tipo.isEmpty()) {
            return null;
        }
        // A logica de criacao fica encapsulada aqui
        if ("email".equalsIgnoreCase(tipo)) {
            return new EmailNotificacao();
        } else if ("sms".equalsIgnoreCase(tipo)) {
            return new SMSNotificacao();
        }

        return null;
    }
}
```

Solução: Cliente (Main)

App.java

```java
public class SistemaMensagens {
    public static void main(String[] args) {
        NotificacaoFactory fabrica = new NotificacaoFactory();

        // Usuario escolheu enviar por SMS
        // Note que a variavel 'n1' e do tipo generico
        Notificacao n1 = fabrica.criarNotificacao("sms");
        n1.enviar("Ola, seu codigo e 1234");

        // Usuario mudou para Email
        Notificacao n2 = fabrica.criarNotificacao("email");
        n2.enviar("Bem-vindo ao sistema!");
    }
}
```

Perguntas para Discussão

1. **Singleton:** Se o Singleton é tão “perigoso” para testes, por que o Spring Framework gerencia seus beans como Singleton por padrão?
2. **Factory:** Em um jogo de RPG, como o Factory Method poderia ajudar a criar monstros aleatórios (Orc, Elfo, Dragão) dependendo do nível da fase?
3. **Geral:** Qual a diferença entre ter um método estático que retorna um objeto e um Factory Method real (polimórfico)?

Respostas da Discussão

1. Singleton vs. Spring Framework

Pergunta: *Se o Singleton é ruim para testes, por que o Spring o usa?*

- **Singleton GoF (Clássico):** É “Hardcoded” (acoplado). O código chama explicitamente `Classe.getInstance()`, o que impede a troca por objetos falsos (*Mocks*) nos testes.
- **Singleton do Spring:** É um **Escopo**. O Spring gerencia a instância única “por fora” (Injeção de Dependência).

A Diferença

No Spring, sua classe não sabe que é um Singleton. Isso permite que você injete implementações diferentes nos testes, unindo a eficiência da memória com a testabilidade.

2. Factory Method em Jogos (RPG)

Pergunta: *Como a Factory ajuda a criar monstros aleatórios?*

O padrão encapsula a complexidade da **probabilidade** e do **balanceamento**.

Cenário:

- O jogo apenas chama: `Monstro m = Fabrica.criar(nivelJogador);`
- Dentro da Fábrica (caixa preta):
  - Se Nível 1: Retorna 90% Slime, 10% Goblin.
  - Se Nível 50: Retorna 50% Dragão, 50% Gigante.

→ **Vantagem:** O Game Designer ajusta a dificuldade em um único arquivo sem risco de quebrar o restante do jogo.

3. Método Estático vs. Factory Real

Pergunta: *Diferença entre método estático (Simple Factory) e Factory Method?*

Simple Factory (Estático)

- Resolve a criação, mas não é flexível.
- Não usa herança.
- Para mudar a lógica, você precisa **abrir e editar** a classe existente (Viola Open/Closed).

Factory Method (GoF)

- Baseado em **Polimorfismo**.
- **Subclasses** decidem o que criar.
- Ex: `FaseFogo` cria monstros de fogo; `FaseGelo` cria monstros de gelo. O código principal não muda.

Conclusão

Takeaways (O que levar para casa)

- **Singleton:** Use para recursos que precisam ser únicos e globais (ex: Conexão DB, Logger), mas cuidado com o acoplamento.
- **Factory Method:** Use quando você não sabe de antemão os tipos exatos e dependências dos objetos com os quais seu código deve funcionar.
- **Padrões não são leis:** São ferramentas. Não force um padrão onde um código simples resolve.

Referências Bibliográficas

- **Livro:** GAMMA, Erich et al. *Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos*. Bookman, 2000. (O clássico GoF).
- **Livro:** FREEMAN, Eric. *Use a Cabeça! Padrões de Projetos*. Alta Books.
- **Site:** Refactoring Guru (refactoring.guru/pt-br) - *Excelente para visualizações*.

Obrigado!