UNIVERSIDADE PITÁGORAS UNOPAR

Engenharia de Software
Sistemas Operacionais

AUTOR: ALLAS MAYCON DO VALLE
RA: 3649457105

**Programação Orientada a Objetos II**

# Padrões de Projeto, Ferramentas e Métodos Ágeis

Cidade de Serrana, SP
2025

AUTOR: ALLAS MAYCON DO VALLE
RA: 3649457105

## Programação Orientada a Objetos II
# Padrões de Projeto, Ferramentas e Métodos Ágeis

Neste projeto, o foco é um conceito de programação chamado Padrão Singleton. A ideia principal do Singleton é muito simples: garantir que uma classe inteira, como a que gerencia configurações do sistema, só para ter uma única cópia (instância) em todo o nosso programa Java.
Usei alguns “truque” de código. O primeiro tornar o construtor privado, o que significa que ninguém de fora pode user o comando new para criar. O próximo passo é criar um método especial e estático, geralmente chamado getInstance(). Esse métodos será a única forma de acessar a classe. Ele checa se a cópia ´nica já existe. Se não existir, ele a cria uma única vez. Se já existir, ele simplesmente devolve a cópia que já está pronta.
Orientador:
Tutor à Distância: Frederico Aparecido Faedo Pinto.
Prof. Renan Cleverson Laureano Flor da Rosa.

Cidade de Serrana, SP
2025

**SUMÁRIO**

1 INTRODUÇÃO ..... 3
2 DESENVOLVIMENTO ..... 4
2.1 METODOLOGIA DE IMPLEMENTAÇÃO ..... 4
2.2 CÓDIGO-FONTE COMPLETO ..... 4
2.3 DETALHAMENTO DA CONFORMIDADE COM OS REQUISITOS ..... 5
3 CONCLUSÃO ..... 6
4 REFERÊNCIAS ..... 7
5 APÊNDICE: REPOSITÓRIO DO CÓDIGO FONTE ..... 7

# 1 INTRODUÇÃO

Este terceiro projeto, chamado **U3: Padrões de Projeto**, nos levou para um lado mais avançado da programação: o estudo de **Padrões de Projeto (Design Patterns)**. O padrão que escolhemos para implementar e entender foi o **Singleton**.

O que é o Singleton? É um dos padrões mais famosos e serve para resolver um problema simples, mas muito importante: **garantir que uma classe tenha sempre e apenas uma única cópia (instância) durante todo o tempo que o programa estiver rodando.** Pense, por exemplo, em uma classe que gerencia as configurações do sistema ou a conexão com um banco de dados: você não quer que existam dez cópias dessas classes, cada uma com informações diferentes; você precisa de uma cópia central e única que todos acessem.

Para alcançar este objetivo, o Singleton exige alguns truques no código que são a chave do projeto:

**- Bloquear a criação normal:** O construtor da classe precisa ser **privado**. Isso impede que outras partes do código usem o comando new para criar novas cópias da classe.

**- Fornecer a cópia única:** Criamos um **método estático** especial chamado, geralmente, getInstance(). É a única porta de entrada. Ele checa se a cópia única já existe. Se não existir, ele a cria; se já existir, ele só devolve a cópia que já está pronta.

Neste projeto, criamos uma classe chamada ConfiguracaoSistema para simular um cenário prático. O procedimento foi seguir à risca o checklist: criar o construtor privado, o campo estático e o método estático.

O resultado esperado é comprovar, no programa principal, que ao pedir a instância duas vezes, recebemos exatamente o mesmo objeto, e a mensagem de criação ("Instância Singleton criada!") só aparece a primeira vez. Isso valida que o padrão Singleton foi implementado corretamente.

## 2 DESENVOLVIMENTO

Criamos duas classes:

**- ConfiguracaoSistema**: A classe que implementa o padrão Singleton.

**- TesteSingleton**: A classe principal (main) que testa e demonstra o uso do padrão.

### 2.1 Classe 1: ConfiguracaoSistema (O Singleton)

Esta classe simula um gerenciador de configurações que deve ser único. Ela segue os três passos essenciais do padrão: construtor privado, campo estático e método estático público.

Java

```
public class ConfiguracaoSistema {

    // REQUISITO 4: Um campo estático para armazenar a única instância (Singleton)
    // O 'instance' é privado e estático, garantindo que só existe um e ninguém muda ele.
    private static ConfiguracaoSistema instance;

    // Campo simples para demonstrar que a instância única carrega dados
    private String logStatus = "Sistema de Log Desativado";

    // REQUISITO 3: Um construtor privado para impedir instâncias externas
    // Isso bloqueia o uso de 'new ConfiguracaoSistema()' fora desta classe.
    private ConfiguracaoSistema() {
        System.out.println("✅ Instância Singleton criada! (Só deve aparecer uma vez)");
        // Inicializa o status no momento da criação
        this.logStatus = "Sistema de Log Ativo e Operacional.";
    }

    // REQUISITO 5: Um método estático público para acessar a instância
    public static ConfiguracaoSistema getInstance() {
        // Verifica se a instância é nula (se ainda não foi criada)
        if (instance == null) {
            // Se for nula, cria a única instância permitida.
            instance = new ConfiguracaoSistema();
        }
        // Retorna a instância única existente.
        return instance;
    }
```

```
    // Método de demonstração para provar que a instância funciona
    public String getStatus() {
        return "Status da Configuração: " + logStatus;
    }
}
```

## 2.2 Classe 2: TesteSingleton (O Programa Principal)

Esta classe testa a implementação do Singleton, garantindo que as variáveis recebam o mesmo objeto e que a mensagem de criação apareça apenas uma vez.

Java

```
public class TesteSingleton {

    public static void main(String[] args) {

        System.out.println("--- Teste de Unicidade do Singleton ---");

        System.out.println("\n[1] Chamando getInstance() pela primeira vez...");
        ConfiguracaoSistema singleton1 = ConfiguracaoSistema.getInstance();

        System.out.println("[2] Chamando getInstance() pela segunda vez...");
        ConfiguracaoSistema singleton2 = ConfiguracaoSistema.getInstance();

        System.out.println("\n--- Verificação de Referências ---");

        if (singleton1 == singleton2) {
            System.out.println("SUCESSO! As referências singleton1 e singleton2 SÃO IDÊNTICAS.");
            System.out.println("Isso prova que o Padrão Singleton GARANTIU uma única instância.");
        } else {
            System.out.println("FALHA! O Singleton não funcionou corretamente.");
        }
        System.out.println("\n--- Demonstração Prática ---");
        System.out.println("Status T1: " + singleton1.getStatus());
        System.out.println("Status T2: " + singleton2.getStatus());
    }
}
```

# 3 CONCLUSÃO

Este projeto permitiu mergulhar no mundo dos **Padrões de Projeto**, implementando com sucesso o padrão **Singleton** em Java. Este projeto prático foi essencial para entender como garantir o controle total sobre a criação de objetos em uma aplicação.

O objetivo principal, que era **garantir que apenas uma instância** da classe ConfiguracaoSistema fosse criada, foi plenamente atingido. O segredo para o sucesso foi a combinação de três elementos-chave:

**- O construtor privado:** Ele bloqueou qualquer tentativa externa de criar a classe usando o comando new.

**- O campo estático:** Ele armazenou a única cópia da classe.

**- O método getInstance():** Ele agiu como um "porteiro", verificando se a cópia já existia antes de permitir sua criação.

A execução do código no programa principal (TesteSingleton) comprovou o funcionamento do padrão de forma irrefutável. A mensagem de criação da instância apareceu **apenas uma vez**, mesmo chamando o método getInstance() duas vezes. Mais importante ainda, a comparação usando o operador == entre as duas referências (singleton1 e singleton2) retornou **verdadeiro**. Isso prova que as duas variáveis estavam apontando para o **mesmo objeto na memória**, cumprindo o requisito de unicidade.

O Padrão Singleton é extremamente útil para gerenciar recursos globais, como *loggers*, *pools* de conexão de banco de dados ou, como no nosso exemplo, um sistema de configurações. Ele assegura a consistência dos dados, pois todas as partes do sistema acessam a mesma fonte de informação.

Este projeto consolidou nossa compreensão sobre Padrões Criacionais e as regras de visibilidade (private e static) em Java. O próximo passo lógico para aprofundar este conhecimento seria aprender sobre o **Padrão Factory**, que lida com a criação de *famílias* de objetos, ou estudar como tornar este Singleton totalmente **seguro para Threads (Thread-Safe)**, usando conceitos de sincronização aprendidos no Projeto 2, o que é crucial em sistemas mais complexos.

O Projeto 3 foi um marco no uso de design patterns para criar código mais limpo, controlado e com responsabilidades bem definidas.

## REFERÊNCIAS

**GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John.** Livro - *Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos*. Porto Alegre: Bookman, 2000. (Esta é a obra clássica, fundamental para o estudo do Padrão Singleton e dos outros 22 padrões de projeto. É a fonte principal que define o padrão implementado.)

**FREEMAN, Eric et al.** Livro - *Use a Cabeça! Padrões de Projeto*. Rio de Janeiro: Alta Books, 2014. (Este livro é conhecido por sua abordagem didática e visualmente engajadora, tornando os Padrões de Projeto, incluindo o Singleton, mais acessíveis e fáceis de aplicar em linguagens como Java.)

**DEITEL, Harvey M.; DEITEL, Paul J.** Livro - *Java: Como Programar*. 10. ed. São Paulo: Pearson Education do Brasil, 2016. (Esta referência é essencial para os fundamentos de Java, servindo como base para entender os conceitos de classes, construtores privados, e membros estáticos, que são os pilares da implementação do padrão Singleton.)

## APÊNDICE: Repositório do Código Fonte

As atividades possuem material exclusivo do portfólio que se encontra no GitHub para download.

https://github.com/allas-amk/portfolio_POO_projeto_III