UNIVERSIDADE DE SÃO PAULO

FACULDADE DE FILOSOFIA, CIÊNCIAS E LETRAS DE RIBEIRÃO PRETO (FFCLRP)

DEPARTAMENTO DE COMPUTAÇÃO E MATEMÁTICA

ANÁLISE E PROJETO DE SOFTWARE

DESIGN PATTERN SINGLETON

Hiago Soares de Araujo

Ribeirão Preto

2025

SUMÁRIO

1 - Introdução

Padrões de projeto (*Design Patterns*) são soluções padronizadas e categorizadas utilizadas para lidar com problemas comuns em desenvolvimento de software. Este trabalho tem como objetivo descrever o design pattern do tipo criacional Singleton, abordando seus fundamentos teóricos, aspectos que exigem atenção em sua utilização e apresentando exemplos práticos de sua aplicação. Também será abordado a utilização desse padrão no projeto sendo desenvolvido para a disciplina de Análise e Projeto de Software.

2 - Intenção e objetivo do Singleton

O objetivo desse design pattern é garantir que uma determinada classe tenha apenas uma única instância, de modo que seja fornecido um ponto de acesso global para o seu uso durante a execução da aplicação.

3 - Motivação

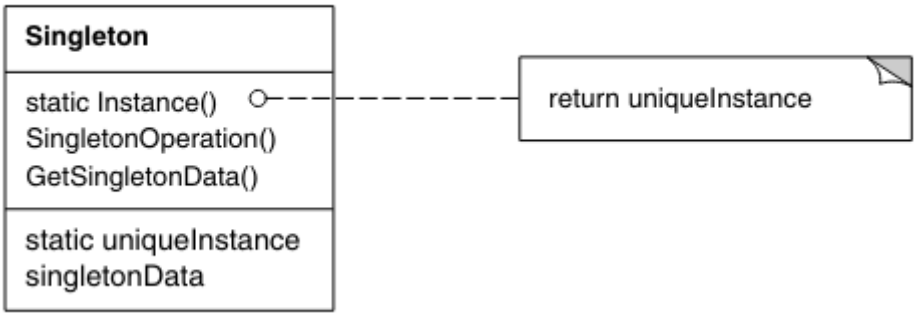
Existem muitas situações em que é altamente recomendável ter apenas uma única instância disponível de uma determinada classe. Pode-se citar, como exemplo, o acesso a recursos compartilhados, como a conexão com banco de dados. Se for criado um objeto de conexão a cada vez que for necessário acessar o banco de dados, múltiplos usuários podem criar múltiplos objetos que, de alguma forma, poderão afetar o desempenho do sistema. Dessa forma, o padrão de projeto Singleton auxilia a enfrentar esse problema ao garantir que múltiplos usuários acessem o mesmo (e único) objeto de conexão.

4 – Aplicabilidade

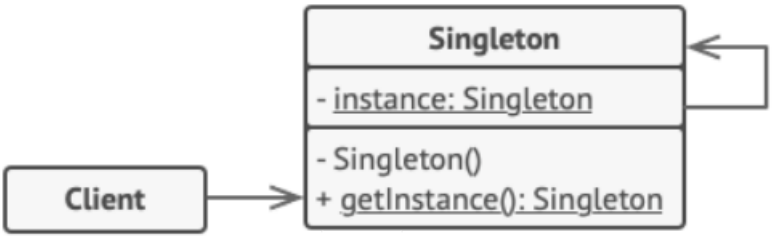
O padrão Singleton pode ser utilizado em diversas ocasiões, como:

- quando for preciso haver apenas uma única instância de uma classe, e essa instância tiver que permitir sua utilização pelos usuários da aplicação por meio de um ponto de acesso bem conhecido;
- a única instância necessitar ser extensível através de subclasses, possibilitando aos clientes usar uma instância estendida sem alterar o seu código.

5 – Estrutura, Participantes e Colaborações



Fonte: Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos



Fonte: Refactoring Guru (https://refactoring.guru/images/patterns/diagrams/singleton/structure-pt-br.png)

A classe **Singleton** é responsável por criar sua única instância, por meio de um atributo de si mesma, e fornecer um ponto de acesso comum para os clientes (usuários da aplicação).

Client são os usuários/clientes que utilizam a instância única.

6 - Consequências

Embora os padrões de projeto ofereçam soluções consolidadas para problemas recorrentes no desenvolvimento de software, sua adoção deve ser feita com discernimento. É fundamental reconhecer que nenhuma solução é completamente eficaz ou isenta de questionamentos. Cada padrão carrega consigo particularidades que podem torná-lo mais ou menos adequado a diferentes cenários. Considerando isso, cabe destacar os pontos positivos e negativos no que se refere ao uso do padrão Singleton.

Pontos positivos:

- Acesso controlado à instância única. Como a classe Singleton encapsula a sua única instância, ela possui controle total sobre como e quando os clientes a acessam.
- Se necessário, permite um número variável de instâncias. O padrão torna fácil mudar de ideia, a fim de permitir mais de uma instância da classe Singleton ou para controlar o número de instâncias que a aplicação utiliza. Somente a operação que permite acesso à instância Singleton necessita ser mudada.

Pontos negativos:

- Viola o Princípio da Responsabilidade Única, segundo o qual uma classe deve realizar apenas as funções para as quais foi projetada. Por exemplo, uma classe ser responsável por estabelecer conexão com banco de dados e gerenciar uma instância de si mesma é uma clara violação desse princípio.
- O padrão requer tratamento especial em um ambiente multithreaded para que múltiplas threads não possam criar um objeto singleton várias vezes.

7 - Implementação

A implementação do padrão Singleton segue, geralmente, os seguintes passos:

- Adicionar um campo privado estático na classe para o armazenamento da instância singleton (uma instância de si mesma).
- Declarar um método público estático para obter a instância singleton.
- Fazer o construtor da classe ser privado, de forma que o método estático da classe ainda seja capaz de chamar esse construtor, mas não os demais objetos.
- Implementar a “inicialização preguiçosa” dentro do método estático, a fim de criar um objeto da classe por meio do construtor privado somente na primeira chamada ao método e armazenar o objeto criado no campo estático. O método deve sempre retornar a instância criada em todas as chamadas subsequentes.

Feito isso, os usuários acessarão a mesma instância ao utilizar o método estático da classe Singleton.

8 – Exemplo de Código

Nesta seção será exemplificada a implementação do design Pattern Singleton utilizando a linguagem Java, incluindo exemplos que abordam sua utilização em um contexto multithreaded.

Abordagem simples:

```java
public class SimpleSingleton {

    private static SimpleSingleton simpleSingleton;
    private int value;

    private SimpleSingleton(int value) {
        this.value = value;
    }

    public static SimpleSingleton getInstance(int value) {
        if(Objects.isNull(simpleSingleton))
            simpleSingleton = new SimpleSingleton(value);
        return simpleSingleton;
    }

    public int getValue() {
        return value;
    }
}
```

Fonte: Autor

```java
public class App {
    Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"Se ocorrer o mesmo valor: SINGLETON FUNCIONOU!");
        System.out.println(x:"Se ocorrerem valores diferentes: PROBLEMA!");

        SimpleSingleton singleton = SimpleSingleton.getInstance(value:1930);
        SimpleSingleton anotherSingleton = SimpleSingleton.getInstance(value:1914);
        SimpleSingleton anotherSingleton2 = SimpleSingleton.getInstance(value:1910);

        System.out.println(singleton.getValue());
        System.out.println(anotherSingleton.getValue());
        System.out.println(anotherSingleton2.getValue());
    }
}
```

Fonte: Autor

```
Se ocorrer o mesmo valor: SINGLETON FUNCIONOU!
Se ocorrerem valores diferentes: PROBLEMA!
1930
1930
1930
```

Fonte: Autor

A abordagem simples, no entanto, enfrenta o problema da condição de corrida se considerarmos uma aplicação multithreaded:

```java
public class MultithreadedApp {
    Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"Se ocorrer o mesmo valor: SINGLETON FUNCIONOU!");
        System.out.println(x:"Se ocorrerem valores diferentes: PROBLEMA!");

        Thread thread1930 = new Thread(new Thread1930());
        Thread thread1910 = new Thread(new Thread1910());
        thread1930.start();
        thread1910.start();
    }

    static class Thread1930 implements Runnable {

        @Override
        public void run() {
            SimpleSingleton singleton = SimpleSingleton.getInstance(value:1930);
            System.out.println(singleton.getValue());
        }
    }

    static class Thread1910 implements Runnable {

        @Override
        public void run() {
            SimpleSingleton singleton = SimpleSingleton.getInstance(value:1910);
            System.out.println(singleton.getValue());
        }
    }
}
```

Fonte: Autor

```
Se ocorrer o mesmo valor: SINGLETON FUNCIONOU!
Se ocorrerem valores diferentes: PROBLEMA!
1930
1910
```

Fonte: Autor

Abordagem multithreaded:

Cabe ressaltar que existe uma implementação mais eficiente do que essa, chamada Double-Checked Locking, que descarta o uso desnecessário do lock após o objeto ter sido instanciado. Mas foge do escopo deste trabalho.

```java
public class MultithreadedSingleton {
    private static MultithreadedSingleton multithreadedSingleton;
    private int value;

    private MultithreadedSingleton(int value) {
        this.value = value;
    }

    public static synchronized MultithreadedSingleton getInstance(int value) {
        if(Objects.isNull(multithreadedSingleton))
            multithreadedSingleton = new MultithreadedSingleton(value);
        return multithreadedSingleton;
    }

    public int getValue() {
        return value;
    }
}
```

Fonte: Autor

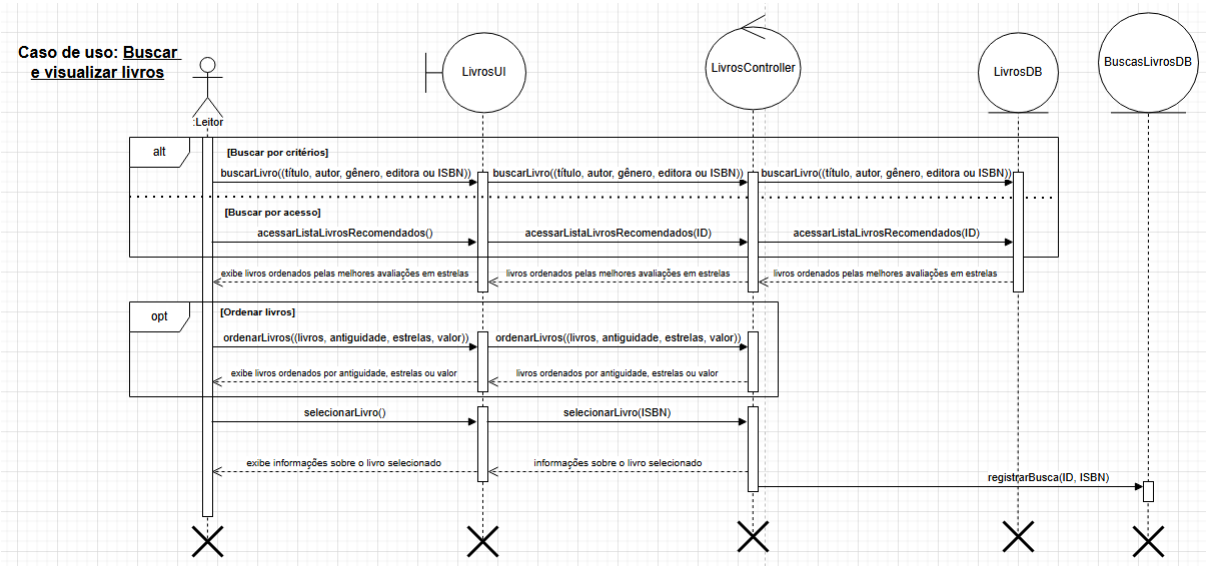
```java
public class MultithreadedApp2 {
    Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"Se ocorrer o mesmo valor: SINGLETON FUNCIONOU!");
        System.out.println(x:"Se ocorrerem valores diferentes: PROBLEMA!");

        Thread thread1930 = new Thread(new Thread1930());
        Thread thread1910 = new Thread(new Thread1910());
        thread1930.start();
        thread1910.start();
    }

    static class Thread1930 implements Runnable {

        @Override
        public void run() {
            MultithreadedSingleton singleton = MultithreadedSingleton.getInstance(value:1930);
            System.out.println(singleton.getValue());
        }
    }

    static class Thread1910 implements Runnable {

        @Override
        public void run() {
            MultithreadedSingleton singleton = MultithreadedSingleton.getInstance(value:1910);
            System.out.println(singleton.getValue());
        }
    }
}
```

Fonte: Autor

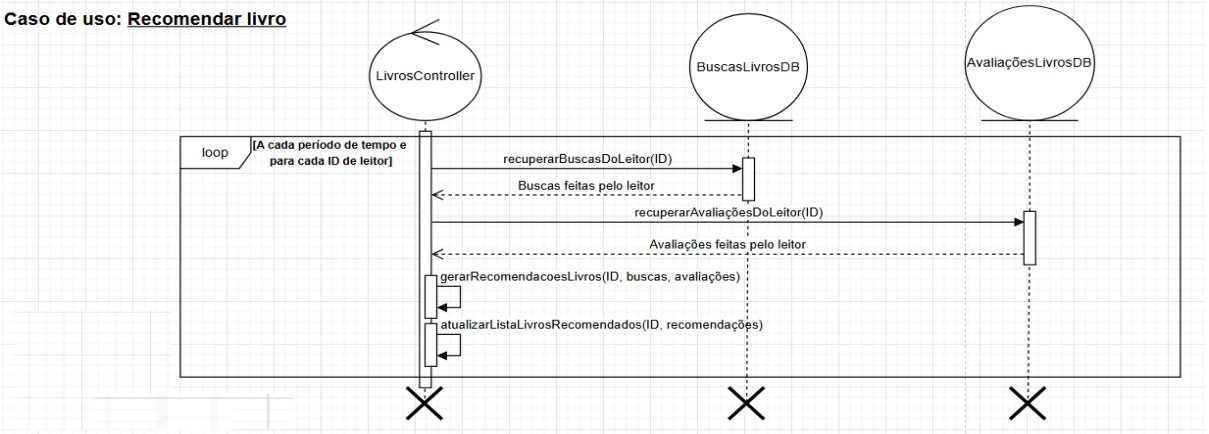
```
Se ocorrer o mesmo valor: SINGLETON FUNCIONOU!
Se ocorrerem valores diferentes: PROBLEMA!
1930
1930
```

Fonte: Autor

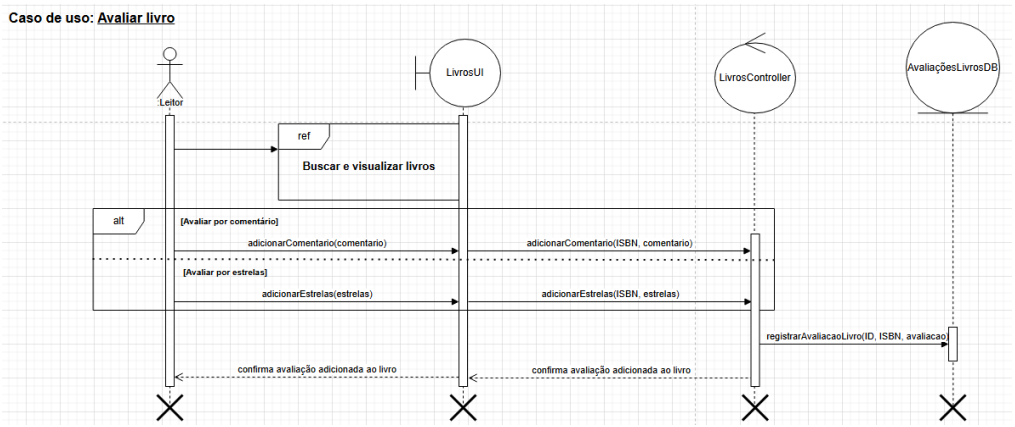
9 - Aplicabilidade ao projeto principal da disciplina



Fonte: Autor



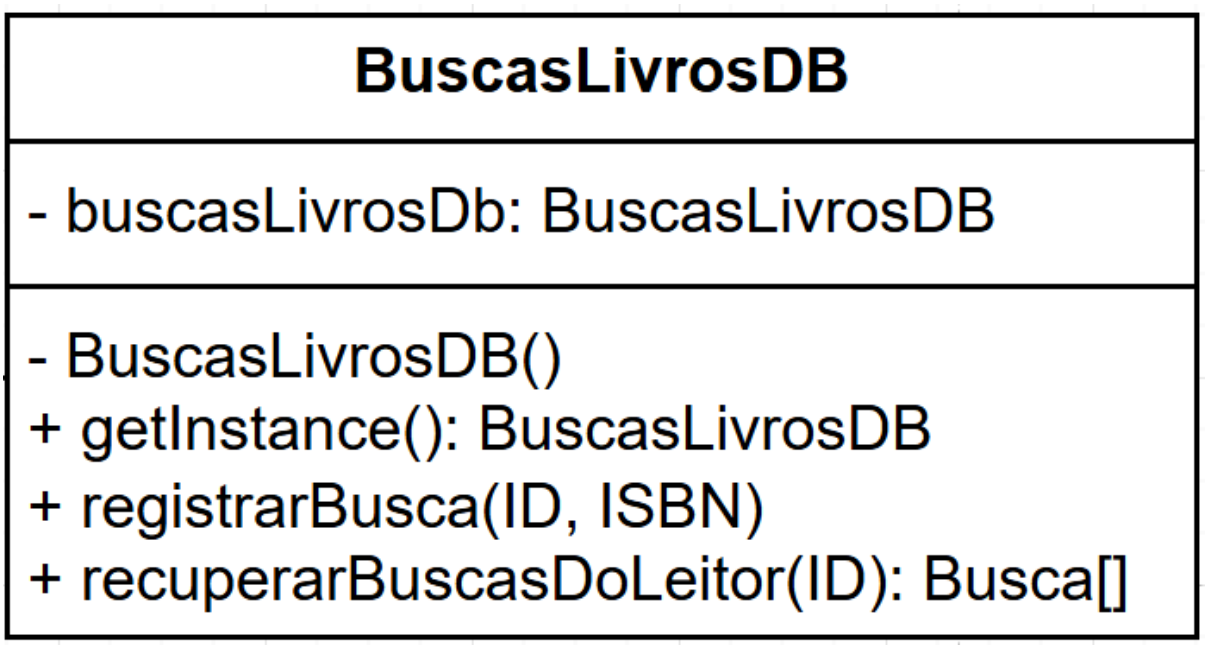
Fonte: Autor



Fonte: Autor

As imagens acima ilustram os diagramas de sequência de alguns casos de uso presentes na aplicação, de forma que é possível observar o acesso a um conjunto variado de bancos de dados durante sua execução. Considerando essa situação, seria extremamente ineficiente criar um objeto de conexão a cada tentativa de acesso aos dados, ainda mais se considerarmos um cenário com diversas threads para tratar requisições de usuários diferentes. Assim, a necessidade de utilização do design pattern Singleton torna-se evidente, ao permitir que uma única instância de um objeto de conexão de um determinado banco de dados seja utilizada para permitir acesso aos dados a esses diversos usuários.

Um exemplo desse padrão aplicado a um banco de dados da aplicação pode ser observado no diagrama de classe a seguir.



Fonte: Autor

10 - Usos conhecidos

Um exemplo de uso desse padrão de projeto em sistemas reais é o método getRuntime, de java.lang.Runtime, da linguagem Java. De acordo com o JavaDoc:

“Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method.”

11 - Padrões relacionados

- Uma classe que utiliza o padrão **Facade** pode frequentemente ser transformada em uma Singleton, já que um único objeto facade é suficiente na maioria dos casos.
- O **Flyweight** pode ser parecido com o Singleton se for possível reduzir todos os estados de objetos compartilhados para apenas um objeto flyweight, considerando algumas mudanças na abordagem.

Referências

Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos

https://refactoring.guru/pt-br/design-patterns/singleton

https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html