Singleton – Design Pattern

Programação Orientada a Objetos

Python

Prof. Diógenes Furlan

Sumário

- Design Pattern
- O que é Singleton?
- Solução com Decorador
- Solução com Método de classe
- Solução com Threads
- Solução com Metaclasse

DESIGN PATTERN

O que é um Padrão?

- *Maneira testada ou documentada de alcançar um objetivo qualquer*
  - *Padrões são comuns em várias áreas da engenharia*
- *Design Patterns, ou Padrões de Projeto*
  - *"**Design Patterns**" de Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, conhecidos como "The Gang of Four", ou GoF, descreve 23 padrões de projeto úteis.*
  - *Padrões para alcançar objetivos na engenharia de software usando classes e métodos em linguagens orientadas a objeto*

Classificação dos Padrões segundo GoF

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Classificação dos Padrões GoF segundo Metsker

Intenção	Padrões
1. Interfaces	Adapter, Facade, Composite, Bridge
2. Responsabilidade	Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight
3. Construção	Builder, Factory Method, Abstract Factory, Prototype, Memento
4. Operações	Template Method, State, Strategy, Command, Interpreter
5. Extensões	Decorator, Iterator, Visitor

SINGLETON

Singleton

- Singleton é um dos padrões de criação que garante que:
  - uma classe tenha apenas uma instância;
  - e fornece um ponto de acesso global a essa instância.

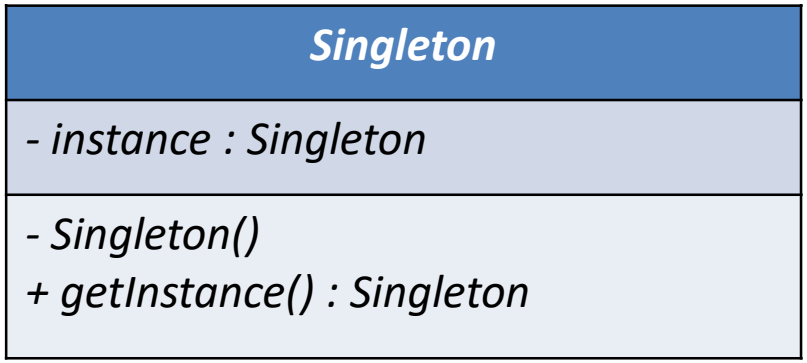
Exemplos Práticos

- Gerenciador de impressão (único)
- *Banco de dados* (único)
- *Acesso ao arquivo de log* (único)
- *Objeto que representa um vídeo* (único)

Como tornar uma classe Singleton?

1. A classe vai conter o **único objeto** existente.
2. Deve-se restringir o acesso ao construtor, tornando-o um **construtor privado**, de forma que novos objetos não possam ser criados.
3. Então utilizar um **método público** que faça o controle da instanciação, de modo que ela só possa ser feita uma vez.

Diagrama de Classe



Em Java

Código Genérico 1 - Criação adiantada

```
public class Singleton {

   private static Singleton instance = new Singleton();

   private Singleton() {}

   public static Singleton getInstance() {
      return instance;
   }
}
```

Em Java

Código Genérico 2 - Criação tardia

```
public class Singleton {

   private static Singleton instance;

   private Singleton() {}

   public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }
}
```

SOLUÇÃO 1

Decoradores

- Decoradores são uma forma poderosa e conveniente de modificar o comportamento de funções ou métodos.
  - permitem "envolver" uma função com outra, adicionando funcionalidade antes e/ou depois da execução da função original.
- O decorador **singleton** é definido como uma função que aceita uma classe **cls** como argumento.

Criando o Singleton

- **INSTANCES**: dicionário que armazenará as instancias únicas das classes
- **GET_INSTANCE**: é uma função interna que aceita argumentos variáveis (*args e **kwargs).
  - Isso permite que ela passe quaisquer argumentos necessários para o construtor da classe.
  - Se não houver uma instância armazenada, ela cria uma nova instância da classe usando **cls(*args, **kwargs)** e armazena essa instância no dicionário.

Singleton – Solução 1

```
def singleton(cls):
    instances = {}

    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

@singleton
class ClasseTeste:
    def __init__(self):
        self.valor = 42
```

Singleton – Solução 1

```
# Testando o Singleton
instancia1 = ClasseTeste()
instancia2 = ClasseTeste()
instancia3 = ClasseTeste()

print(instancia1 is instancia2)  # Deve imprimir: True
print( id(instancia1) )
print( id(instancia2) )
print( id(instancia3) )
```

```
True
2887338199088
2887338199088
2887338199088
```

SOLUÇÃO 2

Método __new__

- O método __new__ é responsável por criar uma nova instância de uma classe.
  - Usando esse método, podemos controlar a criação da instância para garantir que apenas uma instância da classe seja criada.
  - __new__ é chamado antes de __init__ .

Singleton – Solução 2

```
class ClasseTeste:
    _instancia = None

    def __new__(cls, *args, **kwargs):
        if not cls._instancia:
            cls._instancia = super(ClasseTeste, cls).__new__(cls, *args, **kwargs)
        return cls._instancia

    def __init__(self):
        self.valor = 42
```

Singleton – Solução 2

```
# Testando o Singleton
instancia1 = ClasseTeste()
instancia2 = ClasseTeste()
instancia3 = ClasseTeste()

print(instancia1 is instancia2)  # Deve imprimir: True
print( id(instancia1) )
print( id(instancia2) )
print( id(instancia3) )
```

```
True
2887338199088
2887338199088
2887338199088
```

SOLUÇÃO 3

Implementação com Bloqueio (Lock)

- Ambiente multithread.
- Condições de corrida.
- Classe **Lock**: nos ajudará a evitar condições de corrida durante a criação da instância.

- **with cls._lock**: adquire o **lock** antes de executar o bloco de código dentro dele.
  - Isso garante que apenas um thread execute esse bloco de código por vez.

Singleton – Solução 3

```
import threading

class ClasseTeste:
    _instancia = None
    _lock = threading.Lock()

    def __new__(cls, *args, **kwargs):
        with cls._lock:
            if not cls._instancia:
                cls._instancia = super(ClasseTeste, cls).__new__(cls, *args, **kwargs)
        return cls._instancia

    def __init__(self):
        self.valor = 42
```

Singleton – Solução 3

```
# Testando o Singleton
instancia1 = ClasseTeste()
instancia2 = ClasseTeste()
instancia3 = ClasseTeste()

print(instancia1 is instancia2)  # Deve imprimir: True
print( id(instancia1) )
print( id(instancia2) )
print( id(instancia3) )
```

```
True
2887338199088
2887338199088
2887338199088
```

SOLUÇÃO 4

Implementação com Metaclasse

- Uma **metaclasse** é uma "classe de classes".
  - define como uma classe se comporta.
  - use-a para controlar a criação e o comportamento das classes que ela cria.
- O **método __call__** é invocado quando se tenta instanciar uma classe.

Singleton – Solução 4

```
class SingletonMeta(type):
    _instancias = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instancias:
            cls._instancias[cls] = super(SingletonMeta, cls).__call__(*args, **kwargs)
        return cls._instancias[cls]

class ClasseTeste(metaclass=SingletonMeta):
    def __init__(self):
        self.valor = 42
```

Singleton – Solução 4

```
# Testando o Singleton
instancia1 = ClasseTeste()
instancia2 = ClasseTeste()
instancia3 = ClasseTeste()

print(instancia1 is instancia2)  # Deve imprimir: True
print( id(instancia1) )
print( id(instancia2) )
print( id(instancia3) )
```

```
True
2887338199088
2887338199088
2887338199088
```

Entrega em

https://forms.gle/TuDAJ5r9Xp3QGrbb7

EXERCÍCIOS

Exercício Entrega – Fila do Banco

- Vamos simular um sistema gerenciador de senhas na fila de um banco.
- Há 2 filas, uma para cada prioridade. Há 2 tipos de senhas: atendimento normal (AN) e atendimento preferencial (AP).
- Há 4 caixas consumidores e 3 atendentes produtores de senhas.
- O gerador de senhas deve ser um Singleton.

Exercício Entrega – Fila do Banco

- Simule a produção e consumpção de senha aleatoriamente.
  - Os caixas levam de 1 a 10 segundos para fazer o atendimento.
  - O caixa 1 atende somente AP.
  - Os demais caixas atendem somente AN.
  - Quando há mais de 4 pessoas na fila de preferencial, os demais caixas passam a atender 1 AP seguido de 1 AN.
- Simule 100 iterações.