Padrões de Projeto de Software Orientados a Objetos

Tecnologia em Análise e Desenvolvimento de Sistemas

Paulo Mauricio Gonçalves Júnior

Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco

2 de abril de 2018

Parte I

Singleton

Introdução I

- Padrão mais simples em termos de diagrama de classe, composto de uma única classe.
- Usado para manter uma única instância de uma classe: pool de threads, caches, caixas de diálogo, logs, drivers de dispositivos, etc.

```
public class Singleton {
  private static Singleton uniqueInstance;
  // other useful instance variables here
  private Singleton() {}
    public static Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
  // other useful methods here
}
```

Introdução II

Princípio de Projeto

O padrão Singleton garante que uma classe possui uma única instância, e provê um ponto de acesso global a ele.

- Uma fábrica de chocolate possui uma caldeira que mistura chocolate e leite, fervendo-as, e enviando para a próxima fase de criação de barras de chocolate.

```
public class ChocolateBoiler {
  private boolean empty;
  private boolean boiled;

  // This code is only started when the boiler is empty!
  public ChocolateBoiler() {
    empty = true;
    boiled = false;
  }
```

Introdução III

```
// To fill the boiler it must be empty, and, once it's full, we
     set the empty and boiled flags.
public void fill() {
  if (isEmpty()) {
    empty = false;
    boiled = false;
  // fill the boiler with a milk/chocolate mixture
  }
}

// To drain the boiler, it must be full (non empty) and also
     boiled. Once it is drained we set empty back to true.
public void drain() {
  if (!isEmpty() && isBoiled()) {
    // drain the boiled milk and chocolate
    empty = true;
  }
}

// To boil the mixture, the boiler has to be full and not already
     boiled. Once it's boiled we set the boiled flag to true.
```

Introdução IV

```
  public void boil() {
    if (!isEmpty() && !isBoiled()) {
      // bring the contents to a boil
      boiled = true;
    }
  }

  public boolean isEmpty() {
    return empty;
  }

  public boolean isBoiled() {
    return boiled;
  }
}
```

- De alguma forma, o método `fill()` foi chamado para encher a caldeira mesmo ela estando com chocolate e leite fervendo!

Lidando com Multithreading I

```
public class Singleton {
  private static Singleton uniqueInstance;
  // other useful instance variables here
  private Singleton() {}
  public static synchronized Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
  // other useful methods here
}
```

- O único momento em que a sincronização é relevante é na primeira vez que o método é chamado.
- Pode reduzir a performance em até 100 vezes.

Lidando com Multithreading II

```
public class Singleton {
  private static Singleton uniqueInstance = new Singleton();
  private Singleton() {}
  public static Singleton getInstance() {
    return uniqueInstance;
  }
}
```

- Se a aplicação sempre cria e usa uma instância de Singleton ou a carga de criação e execução não é onerosa.
- A máquina virtual garante que a instância será criada antes que qualquer thread acesse a variável estática.

Lidando com Multithreading III

```
public class Singleton {
  private volatile static Singleton uniqueInstance;
  private Singleton() {}

  public static Singleton getInstance() {
    if (uniqueInstance == null) {
      synchronized (Singleton.class) {
        if (uniqueInstance == null) {
          uniqueInstance = new Singleton();
        }
      }
    }
    return uniqueInstance;
  }
}
```

- Usando "double-checked locking" para reduzir o uso de sincronização em `getInstance()`.