**Roteiro 10**

Neste roteiro vamos trabalhar com o padrão Proxy.

**Padrão Proxy** – O padrão proxy fornece um substituto ou representante de outro objeto para gerenciar o acesso a ele. Podemos interpretar o padrão proxy para 3 tipos de proxy.

* **Proxy Remoto** – muito comum em ambientes distribuídos, onde precisamos acessar objetos que estão disponíveis em diferentes locais físicos (diferentes computadores ou espaços de endereçamento).
  + O cliente não faz requisições diretas ao Objeto Real. Ele se comunica com o Proxy e “acha” que está se comunicando com o Objeto Real.
  + O Proxy atua como um representante do Objeto Real, sendo que está no mesmo ambiente físico que o Cliente.
  + O Proxy recebe as requisições do Cliente, delega ao Objeto Real. O Objeto Real por sua vez processa a requisição e devolve a resposta ao Proxy, que devolve a resposta ao Cliente.

Diagrama

O conteúdo gerado por IA pode estar incorreto.

* **Proxy Virtual** – neste caso a questão esta relacionada muitas vezes ao tempo de resposta do Objeto Real. Muitas vezes o Objeto Real leva muito tempo para ser instanciado e responder as requisições. Isso pode acontecer estando inclusive em um mesmo ambiente físico.
  + Neste caso, o cliente é totalmente dependente do Objeto Cliente e se o tempo de resposta do Objeto real é alto, poderemos ter o ambiente totalmente travado por algum tempo.
  + O papel do proxy neste caso é retardar ao máximo a criação do Objeto Real, fazendo com que ele seja instanciado quando realmente for necessário, mantendo assim todo o ambiente em equilíbrio.
  + Dessa forma o Cliente não fica travado, pois depende apenas do Proxy, que por sua vez se encarrega de criar o Objeto real no tempo adequado.

Diagrama

O conteúdo gerado por IA pode estar incorreto.

* Proxy de Proteção – neste caso o proxy faz um papel de proteção/segurança. Ele segue algum protocolo de segurança, gerenciando o acesso do Cliente ao Objeto Real, checando permissões e etc.

Abaixo temos o **diagrama formal do Proxy**

Diagrama

O conteúdo gerado por IA pode estar incorreto.

**Cenário :**

Este cenário tem características para implementação de um proxy virtual. Temos uma classe **“Usuario”** que se comunica com a classe **“PessoaFisicaReceitaFederal”**, que por sua vez consome uma API da receita federal. Para efeitos didáticos utilizaremos apenas uma simulação para o consumo desta API, e por algum motivo a execução de alguns métodos consomem um tempo elevado. Suponha que a classe **“PessoaFisicaReceitaFederal”** possui os seguintes métodos :

* Construtor – recebe como parâmetro o cpf e faz a comunicação com a API – **Tempo = 5 segundos**
* getNome() – retorna no nome do cpf consultado – **Tempo = 2 segundos**
* getIdade()– retorna a idade do cpf consultado – **Tempo = 2 segundos**
* getCpf()– retorna o cpf consultado – **Tempo = 2 segundos**
* CPFAtivo()– retorna se o cpf consultado está ativo – **Tempo = 3 segundos**

Na classe **“Usuario”** também temos atributos e métodos semelhantes com os atributos nome, idade e cpf. Assim como um construtor com os respectivos atributos. O problema é que a criação de um simples usuário levará um tempo muito alto, pois é altamente dependente da classe “**PessoaFisicaReceitaFederal”**

**Pacote : roteiro10.parte1**

1 – Dê sequência ao mesmo projeto no NetBeans chamado **PADROESroteiros**

2 – Crie a classe **PessoaFisicaReceitaFederal** conforme o código abaixo. Esta classe simula a conexão com a API da Receita Federal. Por isso, observe que alguns métodos possuem um sleep que representa o tempo de espera descrito no cenário.

package roteiro10.parte1;

import java.util.concurrent.TimeUnit;

public class PessoaFisicaReceitaFederal {

private String nome;

private int idade;

private String cpf;

private boolean cpfAtivo;

public PessoaFisicaReceitaFederal(String cpf){

try {

this.cpf = cpf;

this.nome = "Fulano";

this.idade = 30;

this.cpfAtivo = true;

TimeUnit.SECONDS.sleep(5);

System.out.println("PessoaFisicaReceitaFederal criada com Sucesso ");

} catch (Exception e) {

throw new UnsupportedOperationException("Falha - " + e.getMessage());

}

}

public String getNome() {

try {

TimeUnit.SECONDS.sleep(2);

return nome;

} catch (Exception e) {

throw new UnsupportedOperationException("Falha - " + e.getMessage());

}

}

public int getIdade() {

try {

TimeUnit.SECONDS.sleep(2);

return idade;

} catch (Exception e) {

throw new UnsupportedOperationException("Falha - " + e.getMessage());

}

}

public String getCpf() {

try {

TimeUnit.SECONDS.sleep(2);

return cpf;

} catch (Exception e) {

throw new UnsupportedOperationException("Falha - " + e.getMessage());

}

}

public boolean isCpfAtivo() {

try {

TimeUnit.SECONDS.sleep(3);

return cpfAtivo;

} catch (Exception e) {

throw new UnsupportedOperationException("Falha - " + e.getMessage());

}

}

}

3 – Crie a classe **Usuario** conforme o código abaixo. Esta é a classe que conversa com todo o resto do sistema, mas é também responsável por fazer as solicitações para a classe **PessoaFisicaReceitaFederal** .

Analise alguns pontos no código :

* Dentro do construtor estamos instanciando o objeto pessoaFisica (responsável pela API). A lentidão do objeto pessoaFisica acabará impactando também no objeto usuário.
* Os métodos validarNome() e verificarCPFAtivo() deveriam ser os únicos responsáveis por impactar na lentidão, pois eles precisam do objeto pessoaFisica. Mas, todos os métodos serão impactados. Mesmo os simples getters().

package roteiro10.parte1;

public class Usuario {

private String nome;

private int idade;

private String cpf;

private PessoaFisicaReceitaFederal pessoaFisica;

public Usuario (String nome, int idade, String cpf){

this.nome = nome;

this.idade = idade;

this.cpf = cpf;

this.pessoaFisica = new PessoaFisicaReceitaFederal(cpf);

}

public String getNome() {

return nome;

}

public int getIdade() {

return idade;

}

public String getCpf() {

return cpf;

}

public boolean validarNome(){

return this.getNome().equals(this.pessoaFisica.getNome());

}

public boolean verificaCPFAtivo(){

return this.pessoaFisica.isCpfAtivo();

}

}

4 – Faça os testes na classe **TesteUsuario**

Analise o tempo de execução de execução com o print da variável **duracao**.

Qual foi o tempo apresentado ?

Observe que alguns prints estão comentados. Retire os comentários um a um e analise o tempo de execução. Qual foi o tempo apresentado em cada teste ?

package roteiro10.parte1;

import java.util.concurrent.TimeUnit;

public class TesteUsuario {

public static void main(String[] args) {

long TempoInicio = System.nanoTime();

Usuario pessoaFisica = new Usuario("Fulano", 30, "123456789");

System.out.println("Nome : " + pessoaFisica.getNome());

//System.out.println("Valida Nome : " + pessoaFisica.validarNome());

//System.out.println("CPF Ativo : " + pessoaFisica.verificaCPFAtivo());

long TempoFinal = System.nanoTime();

long duracao = TimeUnit.NANOSECONDS.toSeconds(TempoFinal - TempoInicio);

System.out.println("Duração : " + duracao);

}

}

**Pacote : roteiro10.parte2**

1 – No mesmo projeto crie o pacote **roteiro10.parte2**

2 – Copie todas as classes criadas na parte1 para o novo pacote.

3 – Crie a interface IReceitaFederal conforme o código abaixo

package roteiro10.parte2;

public interface IReceitaFederal {

public String getNome();

public int getIdade();

public String getCpf();

public boolean isCpfAtivo();

}

4 – Faça com que a classe **PessoaFisicaReceitaFederal** implemente a interface **IReceitaFederal**.

Observe que esta refatoração não causará nenhum impacto, pois em teoria já temos todos os métodos da interface implementados na classe.

package roteiro10.parte2;

import java.util.concurrent.TimeUnit;

public class PessoaFisicaReceitaFederal implements IReceitaFederal{

private String nome;

private int idade;

private String cpf;

private boolean cpfAtivo;

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

5 – Crie a classe **PessoaFisicaReceitaFederalProxy** que também implementa a interface **IReceitaFederal**

Esta classe representa o nosso proxy, e agora sim precisamos implementar todos os métodos da interface.

Observe que temos um atributo **pessoaFisicaRF** em que instanciamos o objeto **PessoaFisicaRecetaFederal** apenas quando for necessário. Se precisamos de um dado da Receita Federal chamamos o método **criarPessoaFisicaReceitaFederal**

package roteiro10.parte2;

public class PessoaFisicaReceitaFederalProxy implements IReceitaFederal{

private String cpf;

private IReceitaFederal pessoaFisicaRF = null;

public PessoaFisicaReceitaFederalProxy(String cpf){

this.cpf = cpf;

}

private void criarPessoaFisicaReceitaFederal(){

if (pessoaFisicaRF == null){

this.pessoaFisicaRF = new PessoaFisicaReceitaFederal(cpf);

}

}

@Override

public String getNome() {

this.criarPessoaFisicaReceitaFederal();

return this.pessoaFisicaRF.getNome();

}

**Implemente os outros métodos da interface**

}

6 – Refatorando a classe **Usuario** para que o proxy tenha o resultado esperado.

Acrescente o atributo pessoaFisica e instancie o objeto PessoaFisicaReceitaFederalProxy no construtor.

package roteiro10.parte2;

public class Usuario {

private String nome;

private int idade;

private String cpf;

private IReceitaFederal pessoaFisica;

public Usuario (String nome, int idade, String cpf){

this.nome = nome;

this.idade = idade;

this.cpf = cpf;

this.pessoaFisica = new PessoaFisicaReceitaFederalProxy(cpf);

}

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

}

7 – Refaça os testes utilizando a classe **TesteUsuario** e faça as análises necessárias.