

Princípios e Padrões de Projeto

Lista 2º Bim.

Prof. Igor Avila Pereira

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Divisão de Computação

`igor.pereira@riogrande.ifrs.edu.br`

1. Decorator

1.1. Teoria

A descrição original do Padrão Decorator é: "O Padrão Decorator anexa responsabilidades adicionais a um objeto dinamicamente. Os decoradores fornecem uma alternativa flexível de subclasse para estender a funcionalidade".

O Decorator é um padrão de projeto estrutural que permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos.

Decorator é um padrão de projeto de software que permite adicionar um comportamento a um objeto já existente em tempo de execução, ou seja, agrega dinamicamente responsabilidades adicionais a um objeto

1.2. Exercícios

1.2.1. Empresa Pública

Em uma empresa pública, um cargo possui um nome e um valor de salário. Os cargos de ingresso são auxiliar, especialista, e gerente. Se alguém com um cargo entrar para um cargo político (Secretário, Prefeito ou Vereador) o salário deve ser incorporado. Um cargo pode ter mais de uma incorporação, os salários base são calculados como se segue:

- Auxiliar = Inicial + 1000
- Especialista = Inicial + 2500
- Gerente = Inicial + 3000

e as incorporações:

- Prefeito = Base + 4000
- Secretário = Base + 2000
- Vereador = Base + 5000

Para o cargo: Se alguém entra como especialista e incorpora vereador e prefeito o cargo fica: especialista incorporado como prefeito incorporado como vereador.

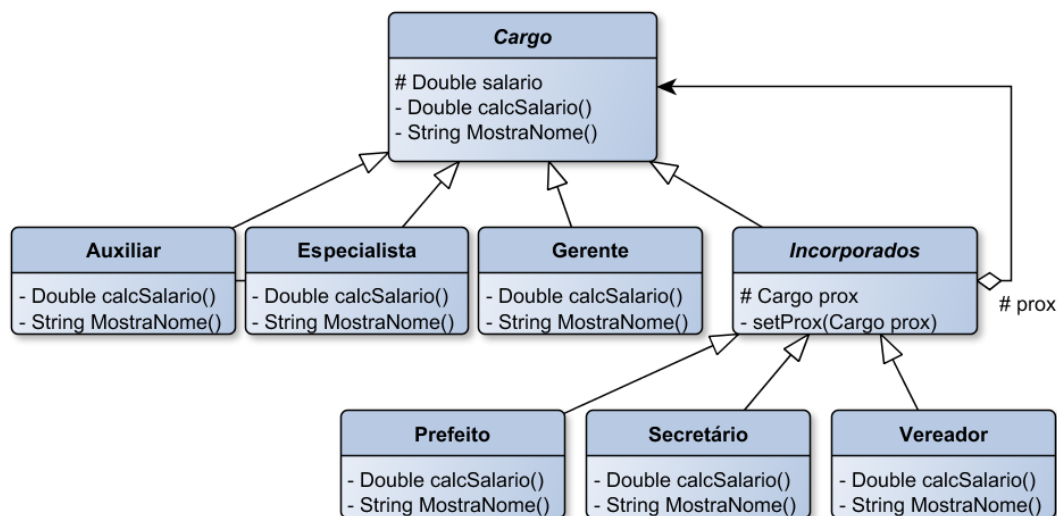


Figure 1. UML: Empresa Pública

1.2.2. Drinks e Coquetéis

Queremos que, dado um objeto **Coquetel**, seja possível adicionar funcionalidades a ele, e somente a ele, em tempo de execução. Vamos ver a arquitetura sugerida pelo padrão:

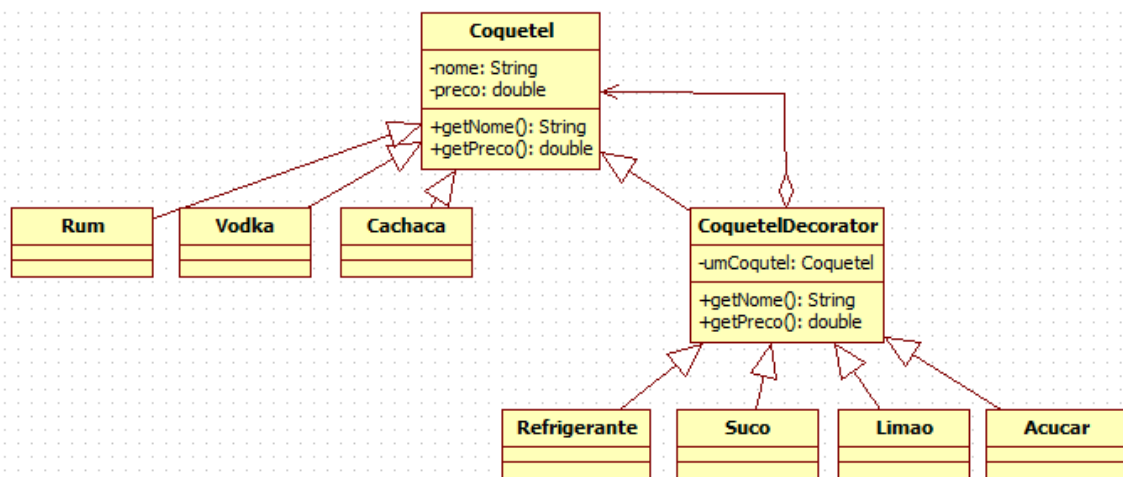


Figure 2. UML: Drinks e Coquetéis

2. Proxy

2.1. Teoria

O Proxy é um padrão de projeto estrutural que permite que você forneça um substituto ou um espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois do pedido chegar ao objeto original.

Tipos de proxy que podem ser utilizados:

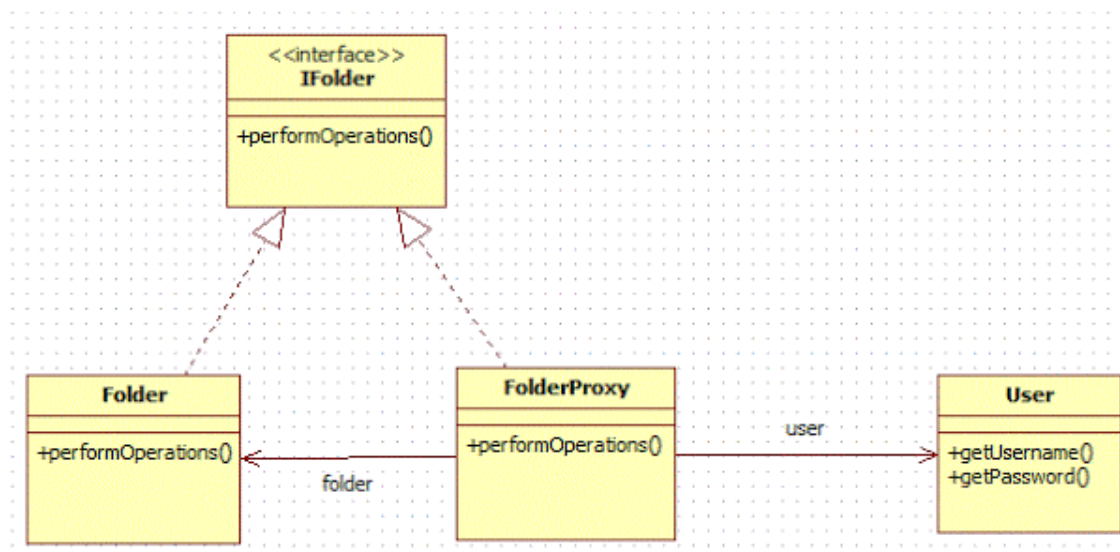
- **Protection Proxy:** Eles controlam o acesso aos objetos, por exemplo, verificando se quem chama possui a devida permissão.
- **Virtual Proxy:** mantem informações sobre o objeto real, adiando o acesso/criação do objeto em si. Como exemplo podemos citar o caso mostrado na aula sobre o padrão, onde é utilizado um Proxy que guarda algumas informações sobre uma imagem, não necessitando criar a imagem em si para acessar parte de suas informações.
- **Remote Proxy:** fornece um representante local para um objeto em outro espaço de endereçamento. Por exemplo, considere que precisamos codificar todas as solicitações enviadas ao banco do exemplo anterior, utilizaríamos um Remote Proxy que codificaria a solicitação e só então faria o envio.
- **Smart Reference:** este proxy é apenas um substituto simples para executar ações adicionais quando o objeto é acessado, por exemplo para implementar mecanismos de sincronização de acesso ao objeto original.

A principal vantagem de utilizar o Proxy é que, ao utilizar um substituto, podemos fazer desde operações otimizadas até proteção do acesso ao objeto. No entanto isto também pode ser visto como um problema, pois, como a responsabilidade de um proxy não é bem definida é necessário conhecer bem seu comportamento para decidir quando utilizá-lo ou não.

2.2. Exercícios

2.2.1. Operações em Pastas

Temos uma pasta na qual você pode executar várias operações, como copiar, colar arquivo ou subpasta. Em termos OOP, temos a interface IFolder e a classe Folder que fornece o método performOperations(), e essas são a classe e a interface existentes que não podemos mudar. Queremos especificar ainda que apenas o usuário com autorização pode acessá-lo e executar operações como recortar ou copiar arquivos e subpastas.



3. Singleton

3.1. Teoria

O Singleton é um padrão de projeto criacional que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.

Singleton é um padrão de projeto de software. Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto. Nota linguística: O termo vem do significado em inglês para um conjunto que contenha apenas um elemento.

O Padrão Singleton tem como definição garantir que uma classe tenha apenas uma instância de si mesma e que forneça um ponto global de acesso a ela. Ou seja, uma classe gerencia a própria instância dela além de evitar que qualquer outra classe crie uma instância dela. Para criar a instância tem-se que passar pela classe obrigatoriamente, nenhuma outra classe pode instanciar ela. O Padrão Singleton também oferece um ponto global de acesso a sua instância. A própria classe sempre vai oferecer a própria instância dela e caso não tenha ainda uma instância, então ela mesma cria e retorna essa nova instância criada.

O padrão Singleton resolve dois problemas de uma só vez, violando o princípio de responsabilidade única:

1. Garantir que uma classe tenha apenas uma única instância. Por que alguém iria querer controlar quantas instâncias uma classe tem? A razão mais comum para isso é para controlar o acesso a algum recurso compartilhado—por exemplo, uma base de dados ou um arquivo.
2. Fornece um ponto de acesso global para aquela instância. Se lembra daquelas variáveis globais que você (tá bom, eu) usou para guardar alguns objetos essenciais? Embora sejam muito úteis, elas também são muito inseguras uma vez que qualquer código pode potencialmente sobrescrever os conteúdos daquelas variáveis e quebrar a aplicação.

Assim como uma variável global, o padrão Singleton permite que você acesse algum objeto de qualquer lugar no programa. Contudo, ele também protege aquela instância de ser sobrescrita por outro código.

Há outro lado para esse problema: você não quer que o código que resolve o problema 1 fique espalhado por todo seu programa. É muito melhor tê-lo dentro de uma classe, especialmente se o resto do seu código já depende dela.

Hoje em dia, o padrão Singleton se tornou tão popular que as pessoas podem chamar algo de singleton mesmo se ele resolve apenas um dos problemas listados.

3.2. Exercícios

3.2.1. Configurações do Usuário

Crie um aplicativo Java com 2 ou mais telas. Na tela principal o usuário poderá definir alguns parâmetros (a definir): nome da aplicação, títulos, tamanho máximo dos campos e etc... O importante é que as definições realizadas na tela principal sejam armazenadas

em uma instância única e que possam ser alteradas e recuperadas pelas demais telas do sistema.

4. Builder

4.1. Teoria

Builder é um padrão de projeto de software criacional que permite a separação da construção de um objeto complexo da sua representação, de forma que o mesmo processo de construção possa criar diferentes representações.

O padrão Builder permite que você construa objetos complexos passo a passo. O Builder não permite que outros objetos acessem o produto enquanto ele está sendo construído. O padrão organiza a construção de objetos em uma série de etapas (construirParedes, construirPorta, etc.).

4.2. Exercícios

4.2.1. Conta Corrente

Suponha que você tenha uma classe **ContaCorrente** que representa contas correntes do fictício IFRSBank! E que ao criar uma conta, podemos fazê-lo apenas com o número da nova conta ou, opcionalmente, informando também o saldo inicial da conta. A nossa classe ficaria mais ou menos assim:

```
class ContaCorrente {
    private int numeroConta;
    private double saldo;

    public ContaCorrente(int numeroConta) {
        this.numeroConta = numeroConta;
    }

    public ContaCorrente(int numeroConta, double saldo) {
        this.numeroConta = numeroConta;
        this.saldo = saldo;
    }
}
```

Pois bem, esse problema foi fácil uma vez que só temos dois construtores. Agora imagine que o sistema cresceu e que opcionalmente podemos receber também o **nome do cliente titular** e que podemos criar uma conta com o **número e o nome do titular** ou **número, nome do titular e saldo**, além das outras duas opções já vistas antes. Isso nos forçaria a criar outros construtores que nos permitissem atender essa nova situação, totalizando assim 4 construtores.

Você já deve ter percebido que quanto mais complexa a situação e quanto mais atributos opcionais tivermos, maior será a quantidade de construtores.

É justamente pra evitar essa dor de cabeça e resolver situações assim que o **Builder** existe! Pegou a ideia?

Você conseguiria fazer uma classe *ContaCorrenteBuilder* implementando esse padrão de projeto e que resolvesse essa nova situação?

5. UML

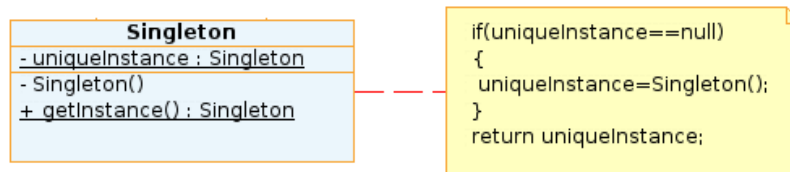


Figure 3. Padrão Singleton

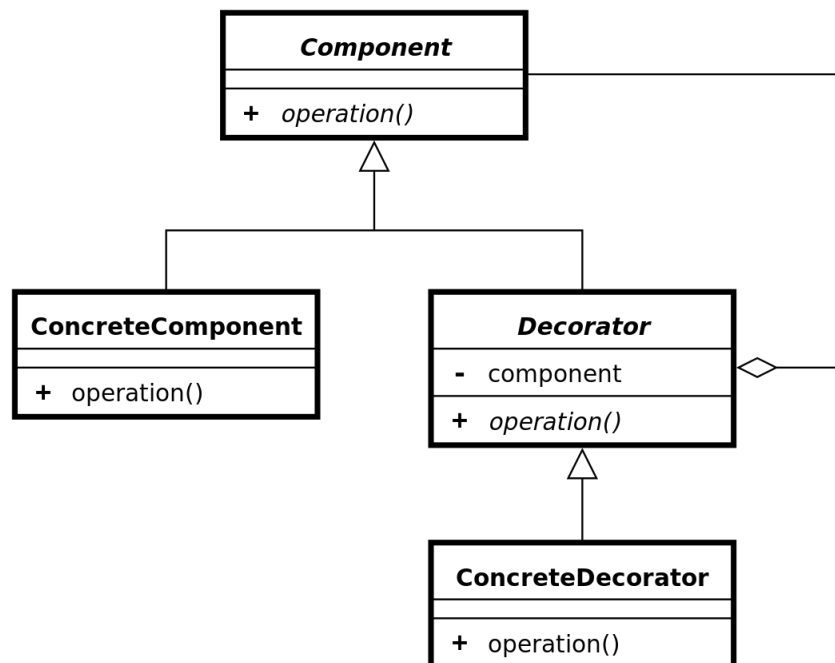


Figure 4. UML - Padrão de Projeto Decorator

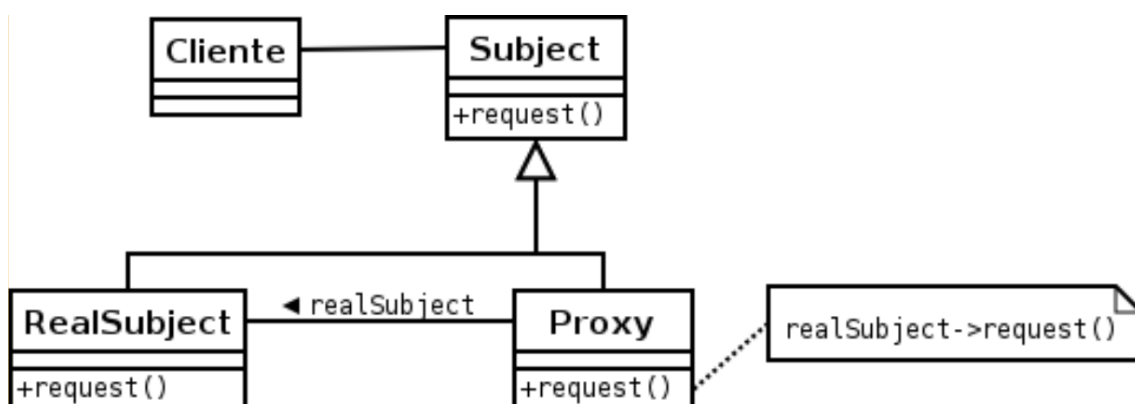


Figure 5. UML - Padrão de Projeto Proxy