DES-SIS-II - A6 Texto de apoio

Site: EAD Mackenzie
Tema: DESENVOLVIMENTO DE SISTEMAS II {TURMA 03B} 2023/1
Livro: DES-SIS-II - A6 Texto de apoio

Impresso por: FELIPE BALDIM GUERRA .
Data: sexta, 28 abr 2023, 18:53

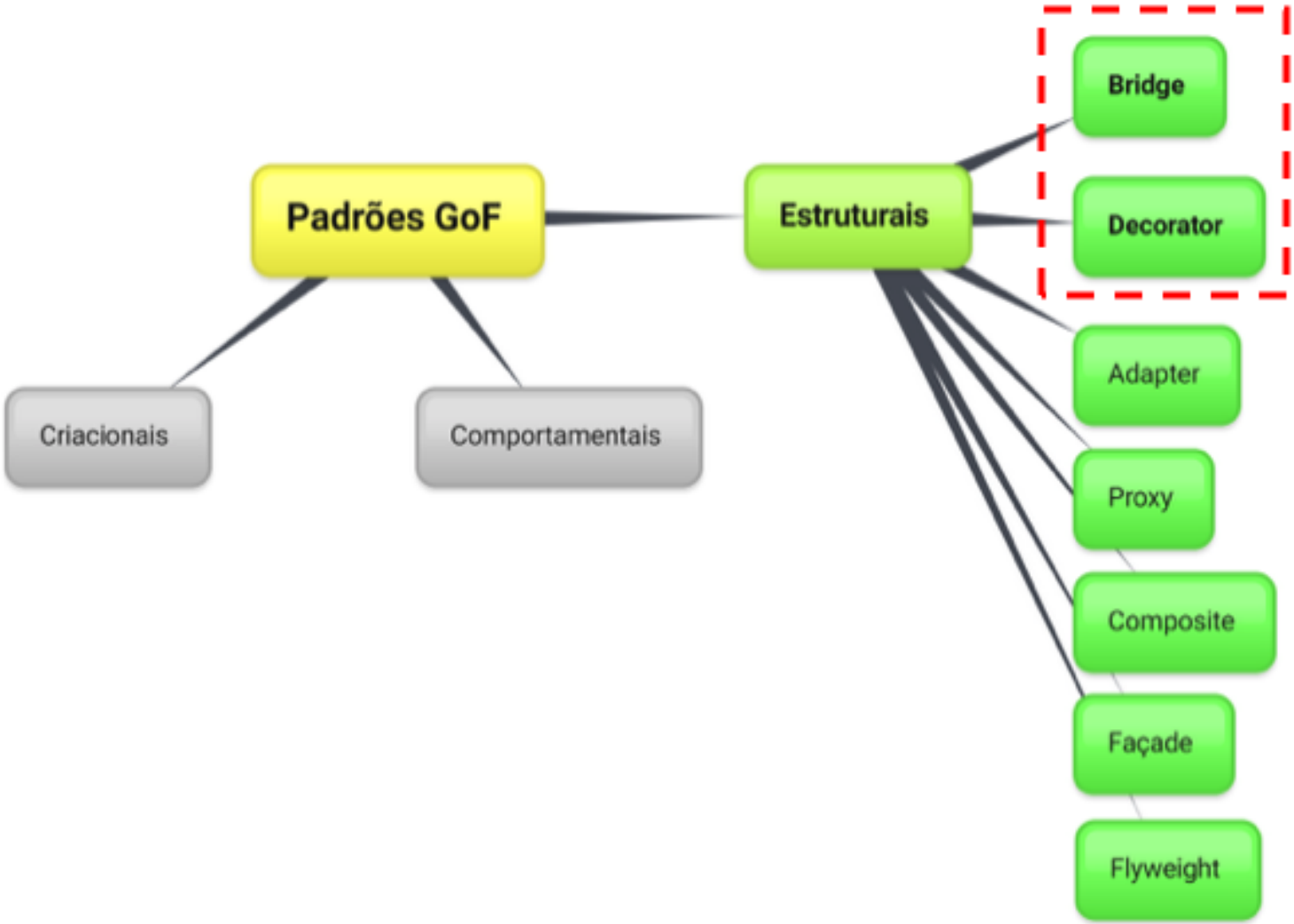
Descrição

Índice

1. PADRÕES GOF ESTRUTURAIS

Na aula anterior, aprendemos a respeito dos padrões GoF criacionais. Vimos que são cinco padrões que respondem questões relacionadas à instanciação de objetos e dentre eles, estudamos três mais a fundo: Singleton, Factory Method e Abstract Factory.

Nesta aula, trabalharemos outra família de padrões GoF, os estruturais. Trata-se de sete padrões de projeto que visam resolver problemas relacionados aos relacionamentos entre classes e objetos. São eles:



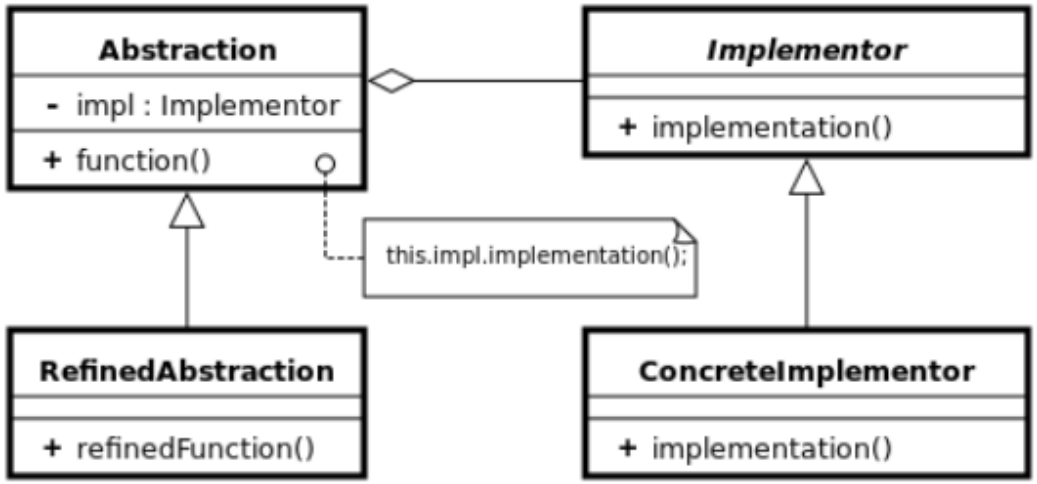
Fonte: Elaborada pelo autor.

Nesta aula, focaremos em dois desses padrões estruturais:Bridge e Decorator. Vamos a eles!

2. BRIDGE

Problema: Como separar uma abstração de sua implementação, permitindo que variem independentemente?

Solução: Separe a abstração e a implementação em hierarquias diferentes, fazendo uma ligação entre elas.



Fonte: Wikipedia .

```
class Abstraction{

    private Implementor impl;

    //métodos

}


class RefinedAbstraction extends
Abstraction{

    //métodos

}


interface Implementor{

    public void implementation();

}


class ConcreteImplementor implements
Implementor{

    public void implementation(){

        //implementação do comportamento

    }

}
```

Fonte: Elaborada pelo autor .

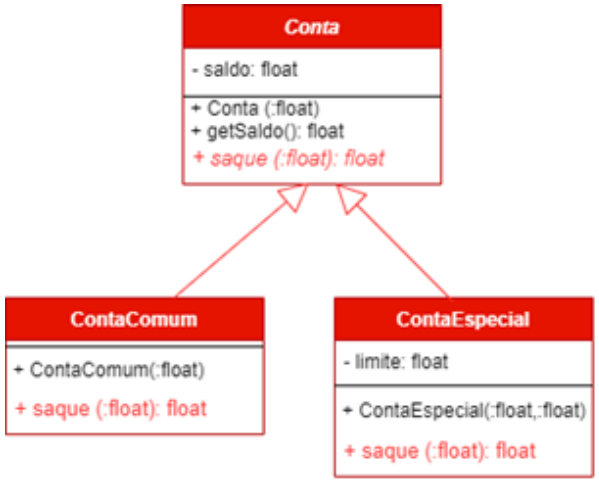
Pontos a considerar:

- O padrão *Bridge* torna mais reusáveis e flexíveis projetos que seriam baseados apenas em uma hierarquia, dividindo-a em duas , uma que representa a ideia abstrata e a outra que representa seus detalhes de implementação.

Exemplo do Mundo Real

Em um banco, é comum que as contas dos clientes sejam divididas em conta comum e conta especial, com diferentes regras para cada uma delas. Por exemplo, suponhamos que uma conta comum não tenha limite de crédito e uma conta especial tenha – o que permite, por exemplo, saques além do saldo em conta.

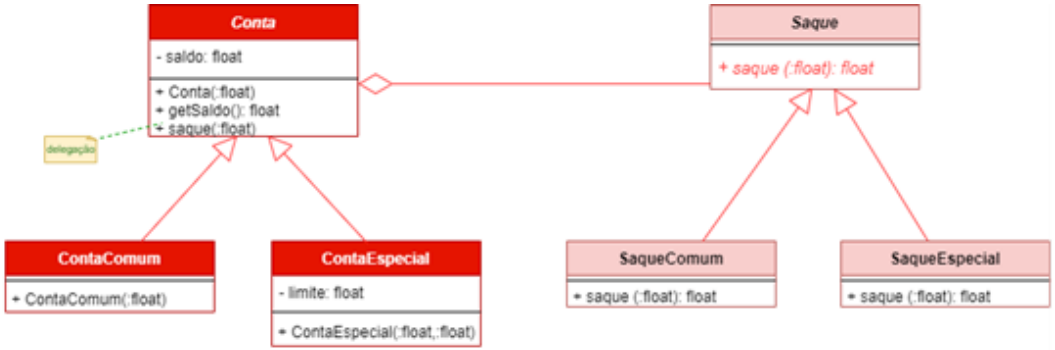
Uma primeira modelagem desta situação pode ser vista a seguir:



Fonte: Elaborada pelo autor.

Apesar de representar uma modelagem conceitualmente correta, é um projeto que torna difícil a implantação de mudanças (por exemplo, mudanças nas regras de saques, por exemplo, permitindo que os clientes comuns possam sacar um limite fixo extra, ou que se implante um limite máximo para saque – ou qualquer outra mudança na regra de negócio de saque): todas as classes teriam que ser alteradas!

Com a aplicação do padrão *Bridge*, pode-se separar as contas e as diferentes variações de saque em hierarquias diferentes:



Fonte: Elaborada pelo autor.

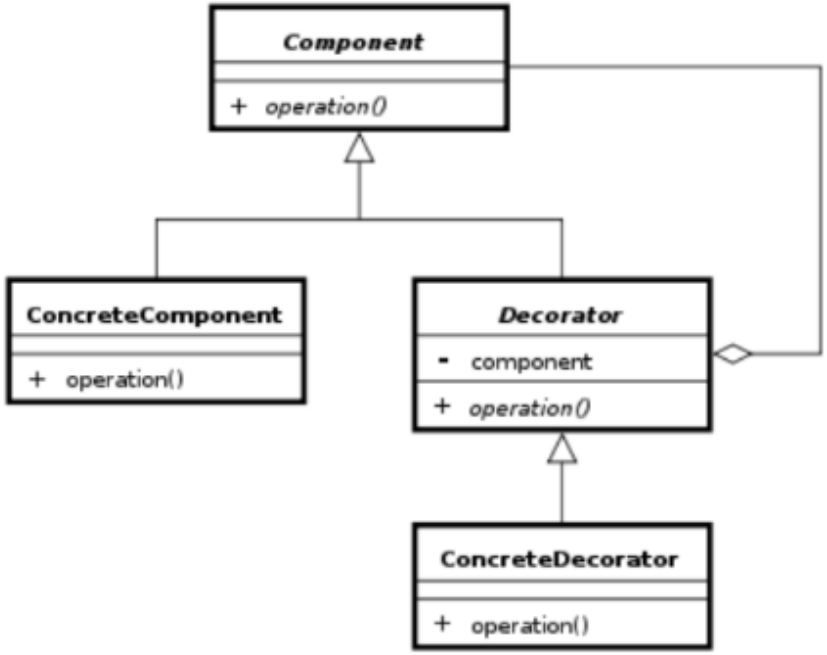
Veja mais em < https://refactoring.guru/design-patterns/bridge>.

Aplique o que aprendeu agora na primeira atividade Praticando desta aula.

3. DECORATOR

Problema: Como adicionar funcionalidades dinamicamente a um objeto?

Solução : Encapsule cada nova funcionalidade em uma classe de “decoração”. Use , então , a inversão de responsabilidade: em vez do elemento principal conter cada uma das “decorações”, as decorações são aplicadas umas sobre as outras até chegar ao elemento a ser decorado.



Fonte: Wikipedia .

```
abstract class Component{
    public abstract void operation();
}

class ConcreteComponent extends Component{
    public void operation(){
        //implementação
    }
}

abstract class Decorator extends Component{
    protected Component component;
}

class ConcreteDecorator extends Decorator{
    public void operation(){
        //implementação
    }
}
//Exemplo de uso
Component c = new ConcreteDecorator(new ConcreteComponent());
```

Fonte: Elaborada pelo autor.

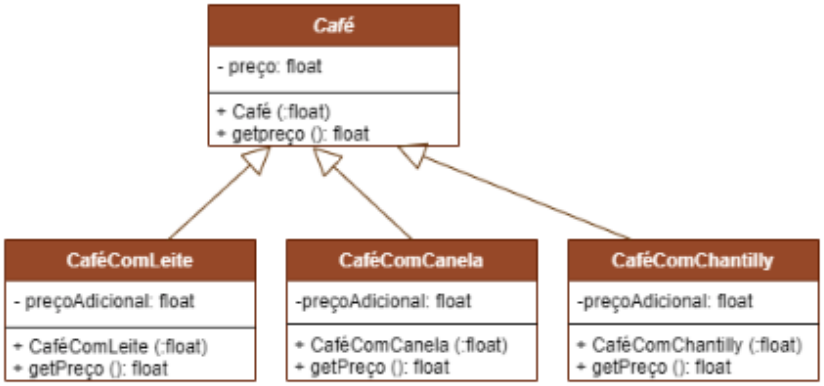
Pontos a considerar:

- Um projetista iniciante poderia tentar resolver o problema de adicionar funcionalidades dinamicam ente apenas por meio de subclasses. Porém, as subclasses são definidas em tempo de compilação, não permitindo que funcionalidades sejam adicionadas de forma dinâmica, em tempo de execução.
- O uso d o *Decorator* baseia-se no princípio de inversão de responsabilidades: em vez da classe que representa o componente principal estar agregada a diferentes classes que encapsulam as distintas funcionalidades de decoração, são as funcionalidades que podem estar associadas a qualquer elemento, que pode ser um componente ou uma decoração.

Exemplo do Mundo Real

Considere uma cafeteria que vende (obviamente) café, que pode vir com alguns acompanhamentos, como leite, canela e chantilly, por exemplo. Cada um desses elementos (o café ou os adicionais) têm preços individuais, que vão se somando. Como modelar es sa situação?

Uma primeira abordagem poderia ser:



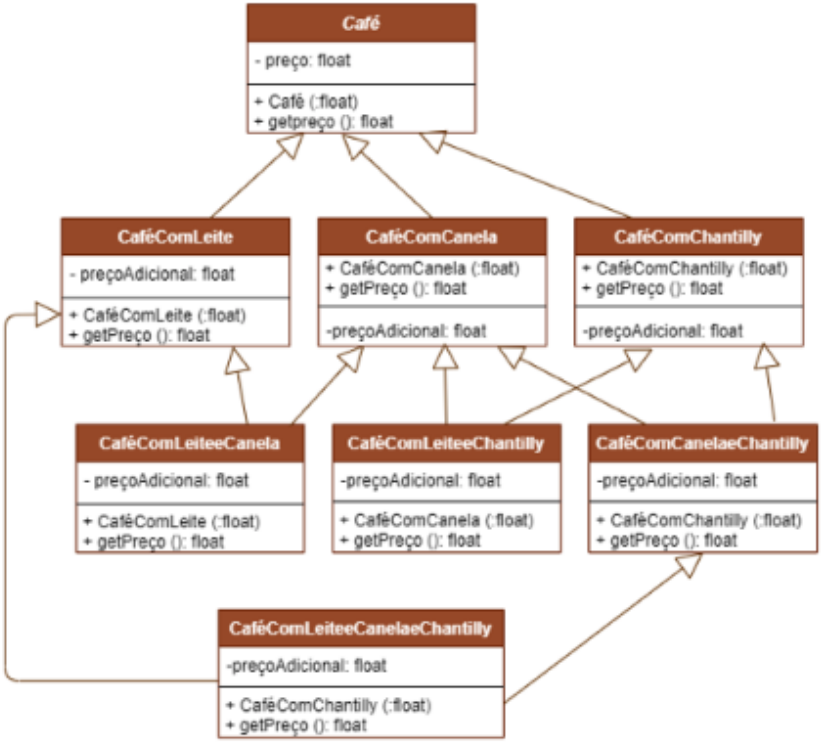
Fonte: Elaborada pelo autor.

Parece uma modelagem razoável, não? Porém, ela tem alguns problemas:

Para ser possível tomar um café puro, a classe Café não deveria ser abstrata.

Esta modelagem não permite acrescentar dois adicionais ao mesmo tempo, por exemplo, café com leite e canela.

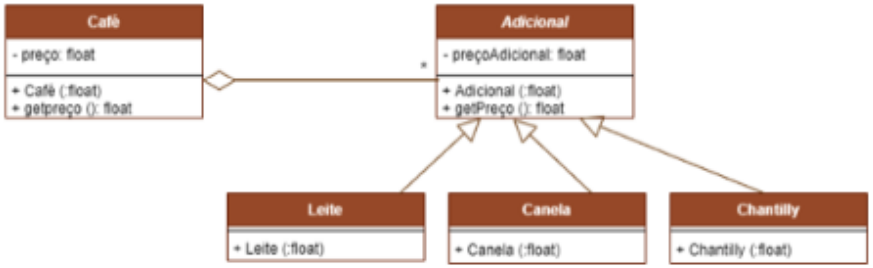
Vamos ver uma outra possível modelagem para resolver es se último problema:



Fonte: Elaborada pelo autor.

Note que essa solução está longe de ser a melhor: tivemos uma explosão de classes! E , se inseríssemos um novo adicional, por exemplo, chocolate, teríamos que criar várias classes prevendo todas as possíveis combinações!

Bom, você poderia imaginar uma outra solução:



Fonte: Elaborada pelo autor.

> Bom, embora pareça, este não é exatamente o caso do padrão Bridge : note que há uma cardinalidade múltipla na agregação, o que não é previsto no Bridge .

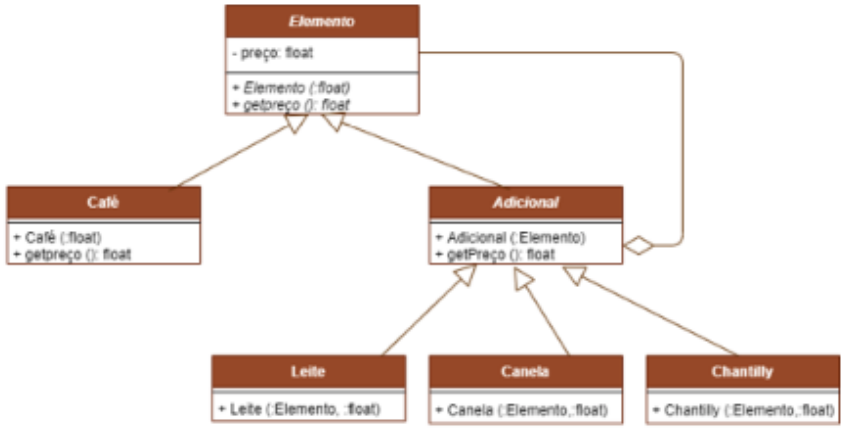
Não é de todo uma má solução. Porém, o componente principal (café) está acoplado a elementos secundários (adicionais). E, para se pedir um café puro e calcular seu preço, uma condicional teria que ser usada:

```
class Café{
    protected float preço;
    public List<Adicional> adicionais = new ArrayList();
    public float getPreço(){
        float temp=0;
        if (adicionais!=null)
            foreach(Adicional a: adicionais)
                temp += a.getPreço();
        temp += preço;
        return temp;
}
```

Fonte: Elaborada pelo autor.

> Para evitar a condicional, poderia se pensar no uso do conceito de Null Object , estudado na parte de refatoração.

O uso do padrão *Decorator* faz com que o projeto fique muito mais flexível:



Fonte: Elaborada pelo autor.

Veja um exemplo de uso para um pedido de café com leite e chantilly:

```
Elemento e = new Chantilly(new Leite(new Café(4.0),1.5),4.55);
System.out.println(e.getPreço());//resultado: 10.05
```

Veja mais em: <https://refactoring.guru/design-patterns/decorator>.

Aplique o que aprendeu agora na segunda atividade Praticando dest a aula.