

DESIGN PATTERNS

**COMPOSITE
&
PURE FABRICATION**

Engenharia de
Software II

Camile Pedrolo e
Guilherme Thomas

SOBRE

PURE FABRICATION

- *Design pattern* do conjunto **GRASP**;
- Visa evitar que **classes de domínio** comportem responsabilidades **extras** (temporárias ou específicas);
- Criação de **novas classes** para delegar as **responsabilidades**.

QUANDO APLICAR?

PURE FABRICATION

Códigos com funcionalidades que não pertencem diretamente a uma classe do domínio.

Importante avaliar a necessidade do uso do padrão.

```
public class Produto { 3 usages
    private String nome; 2 usages
    private double preco; 2 usages

    public Produto(String nome, double preco) { 1 usage
        this.nome = nome;
        this.preco = preco;
    }

    public String getNome() { 1 usage
        return nome;
    }

    public double getPreco() { 1 usage
        return preco;
    }
}
```

```
public class CalculadoraImposto {  
    private static final double TAXA_IMPOSTO = 0.1;  
  
    public double calcular(Produto produto) {  
        return produto.getPreco() * TAXA_IMPOSTO;  
    }  
}  
no usages
```

```
public class Main {  
    public static void main(String[] args) {  
        Produto produto = new Produto( nome: "Notebook",  preco: 3000.00);  
  
        CalculadoraImposto calculadoraImposto = new CalculadoraImposto();  
  
        double imposto = calculadoraImposto.calcular(produto);  
        System.out.println("Imposto sobre " + produto.getNome() + ": R$" + imposto);  
    }  
}
```

BENEFÍCIOS

PURE FABRICATION

- Fácil manutenção;
- Reutilização de código;
- Alta coesão.

SOBRE

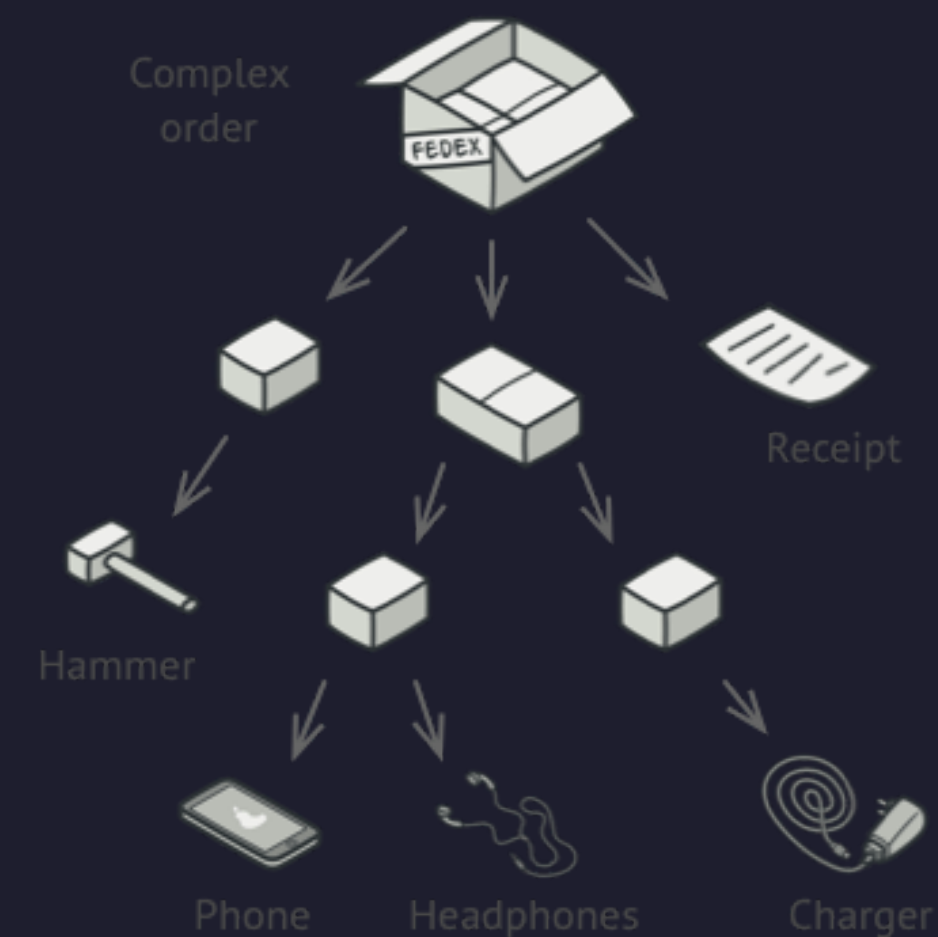
COMPOSITE

- *Design pattern* do conjunto **GOF**;
- Composite é um padrão de design estrutural que permite compor objetos em estruturas de árvore e, então, trabalhar com essas estruturas como se fossem objetos individuais.

QUANDO APLICAR?

COMPOSITE

Usar o padrão Composite só faz sentido quando o modelo principal do seu aplicativo pode ser representado como uma árvore.



QUANDO APLICAR?

COMPOSITE

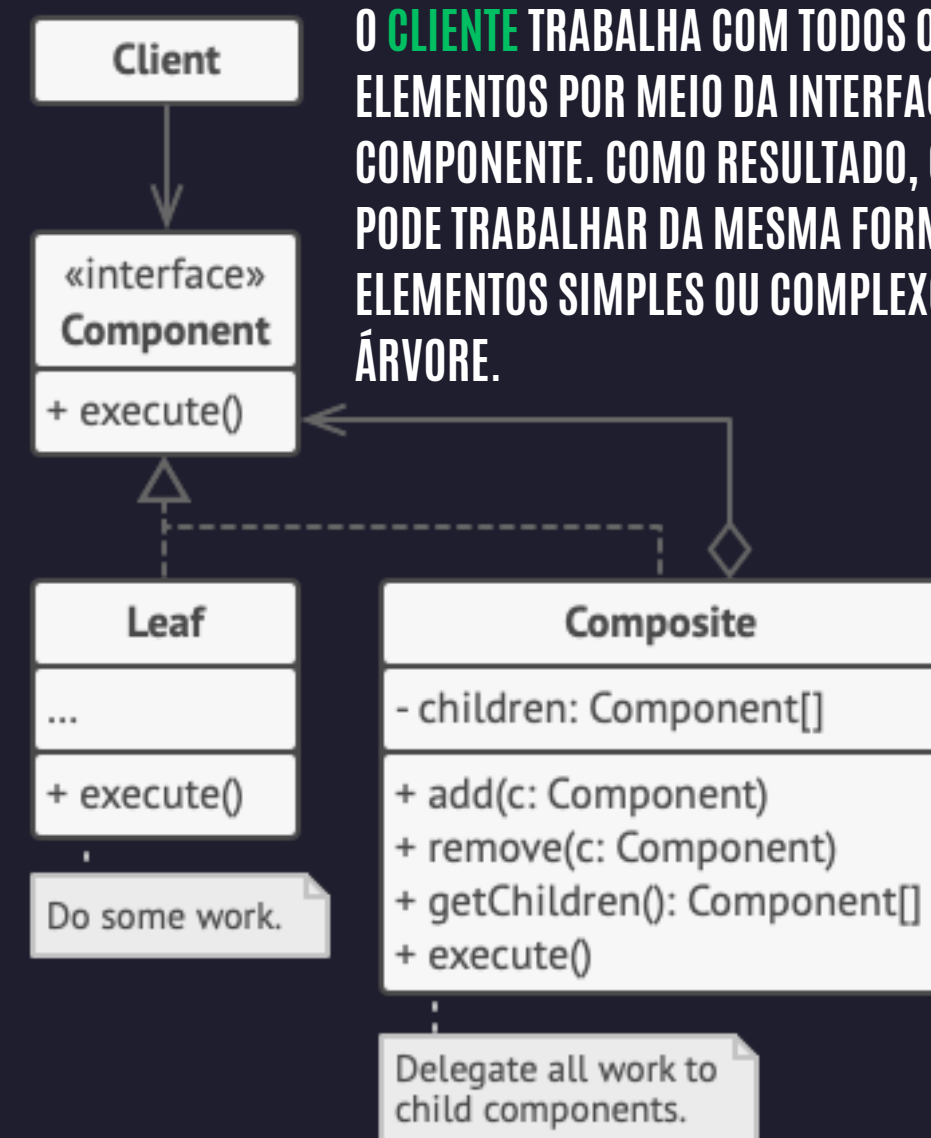
O maior benefício dessa abordagem é que você não precisa se importar com as classes concretas de objetos que compõem a árvore. Você não precisa saber se um objeto é um produto simples ou uma caixa sofisticada. Você pode tratá-los todos da mesma forma por meio da interface comum. Quando você chama um método, os próprios objetos passam a solicitação para baixo na árvore.

COMPOSITE

ESTRUTURA

A INTERFACE DO **COMPONENTE** DESCREVE OPERAÇÕES QUE SÃO COMUNS A ELEMENTOS SIMPLES E COMPLEXOS DA ÁRVORE.

A **FOLHA** É UM ELEMENTO BÁSICO DE UMA ÁRVORE QUE NÃO POSSUI SUBELEMENTOS.



O **CLIENTE** TRABALHA COM TODOS OS ELEMENTOS POR MEIO DA INTERFACE DO COMPONENTE. COMO RESULTADO, O CLIENT PODE TRABALHAR DA MESMA FORMA COM ELEMENTOS SIMPLES OU COMPLEXOS DA ÁRVORE.


O **CONTAINER** (TAMBÉM CONHECIDO COMO COMPOSITE) É UM ELEMENTO QUE TEM SUBELEMENTOS: FOLHAS OU OUTROS CONTAINERS.



```
public interface Produto{  
    public Double getValor();  
}
```



```
public class Refrigerante implements Produto{  
    private String nome;  
    private Double valor;  
  
    public Refrigerante(String nome, Double valor){  
        this.nome = nome;  
        this.valor = valor;  
    }  
    public String getNome(){  
        return nome;  
    }  
  
    @Override  
    public Double getValor(){  
        return valor;  
    }  
}
```



```
public class FardoDeRefrigerante implements Produto{
    private List<Produto> produtos = new ArrayList<>();

    public void add(Produto ...produtos){
        this.produtos.addAll(Arrays.asList(produtos));
    }

    public void add(Produto produto, int quantidade){
        for (int i = 0; i< quantidade; i++){
            this.produtos.add(produto);
        }
    }

    public void remove(Produto produto){
        produtos.remove(produto);
    }

    @Override
    public Double getValor(){
        Double soma = 0d;

        for(Produto produto: produtos){
            soma += produto.getValor();
        }

        return soma;
    }
}
```



```
Produto soda = new Refrigerante("Soda", 3.5);  
FardoDeRefrigerante fardoDeRefrigerante = new FardoDeRefrigerante();  
fardoDeRefrigerante.add(soda,6);
```



```
Produto soda1 = new Refrigerante("Soda", 2.75);  
Produto soda2 = new Refrigerante("Soda", 3.99);  
  
FardoDeRefrigerante fardoDeRefrigerante = new FardoDeRefrigerante();  
fardoDeRefrigerante.add(soda1,soda2,...);
```

- COMO A CLASSE FARDODEREFRIGERANTE POSSUI UMA LISTA DENTRO DE SI DO TIPO PRODUTO, ELA É CAPAZ DE SUPORTAR ATÉ MESMO OUTROS FARDOS DENTRO DE SI MESMA.

BENEFÍCIOS

COMPOSITE

- Poder introduzir novos elementos sem quebrar o código;
- Não se preocupar com classes concretas;

REFERÊNCIAS

- <https://medium.com/@safonovb2c/pure-fabrication-in-software-design-bd4c7ce34bf7>
- [https://translateme.github.io/docs/desenho de software/padroes/grasp/#:~:text=O%20padr%C3%A3o%20Fabrica%C3%A7%C3%A3o%20Pura%20%C3%A9,similar%20%C3%A0%20presta%C3%A7%C3%A3o%20de%20servi%C3%A7os.](https://translateme.github.io/docs/desenho%20de%20software/padroes/grasp/#:~:text=O%20padr%C3%A3o%20Fabrica%C3%A7%C3%A3o%20Pura%20%C3%A9,similar%20%C3%A0%20presta%C3%A7%C3%A3o%20de%20servi%C3%A7os.)
- <https://refactoring.guru/pt-br/design-patterns/composite>