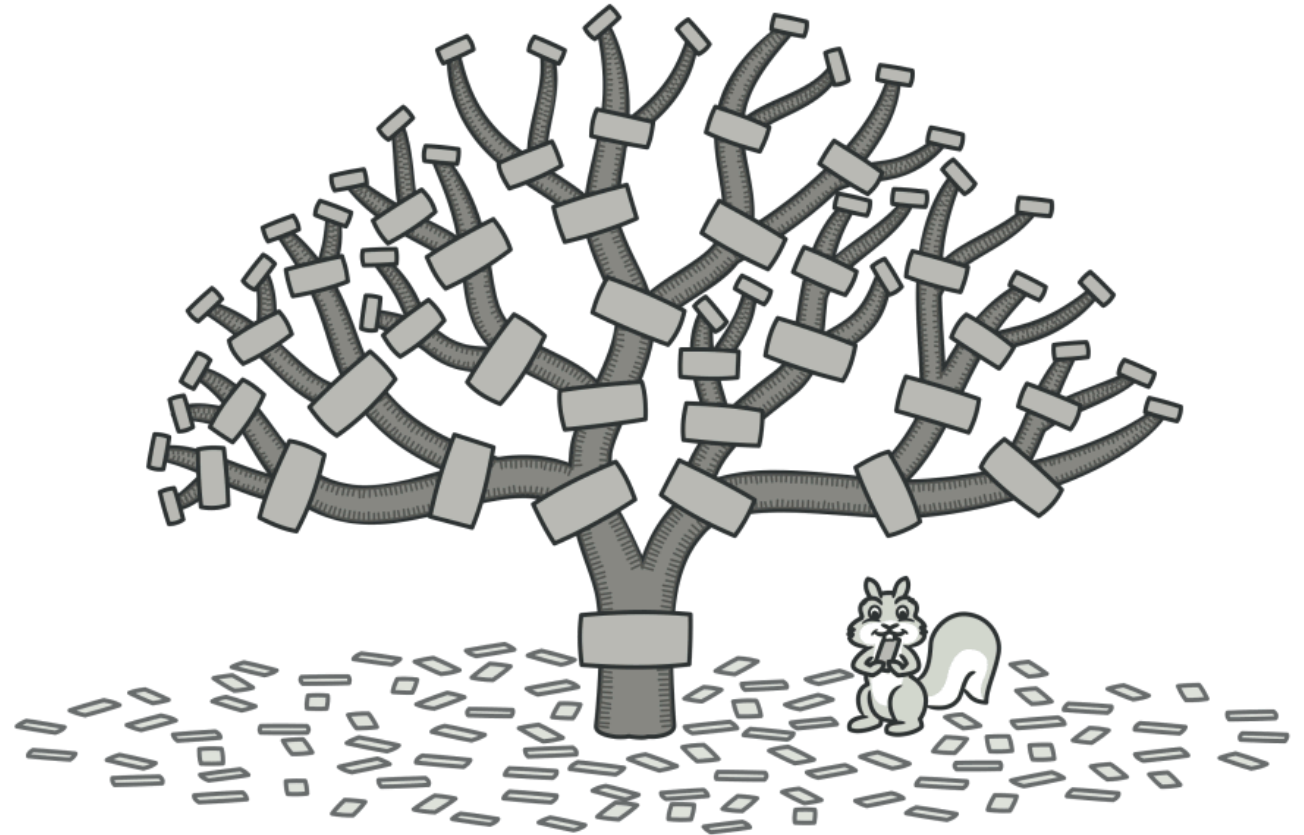


TÓPICO 10: COMPOSITE

Design Patterns - Professor Ramon Venson - SATC 2026.2

Definição

Composite é um padrão de projeto **estrutural** que **compõe objetos em estruturas de árvore** permitindo que o cliente trate **objetos individuais e composições** da mesma forma.



Problema

Imagine uma ferramenta que precisa **percorrer** o conteúdo de um sistema de arquivos. Pastas podem conter **arquivos ou outras pastas**:

```
projeto/  
  README.md  
  src/  
    Main.java  
    Utils.java
```

Tratar **Arquivo** e **Pasta** como coisas **diferentes** complica o código para essa finalidade

Sem uma **interface comum**, o cliente é obrigado a descobrir "quem é quem" antes de agir:

```
public class Arquivo {    // "folha": não tem filhos
    String nome;
    int tamanho;
}

public class Pasta {    // "container": tem filhos
    String nome;
    List<Object> itens = new ArrayList<>();    // Object?!
    void adicionar(Object item) { itens.add(item); }
}
```

- a pasta aceita **qualquer coisa** (`Object`), sem segurança de tipos
- cada tipo guarda os próprios dados de um jeito **diferente**

E o código que lista o conteúdo precisa de `instanceof` e **recursão manual**:

```
public void listar(Object item, String prefixo) {  
    if (item instanceof Pasta p) {  
        System.out.println(prefixo + p.nome + "/" );  
        for (Object filho : p.itens)  
            listar(filho, prefixo + "  ");    // recursão "na mão"  
    } else if (item instanceof Arquivo a) {  
        System.out.println(prefixo + a.nome + " (" + a.tamanho + " KB)");  
    }  
}
```

O resultado é um código frágil que **espalha** a lógica da árvore:

- o cliente **conhece** os tipos concretos (`instanceof`)
- a recursão **duplica** em todo lugar que se usa
- adicionar um tipo novo (ex.: `Atalho`) exige **caçar** cada `instanceof`
- difícil **estender** sem introduzir bugs

Solução

Vamos criar uma **interface comum** para folhas e containers. O container guarda os filhos **usando a própria interface**. Assim a estrutura vira naturalmente uma **árvore recursiva**:

- **Component**: interface comum de folhas e composições
- **Leaf** (Arquivo): nó sem filhos, implementa o comportamento direto
- **Composite** (Pasta): guarda filhos `Component` e **delega** a eles
- O cliente fala **somente com** `Component` — sem `instanceof`

Implementação

1. Primeiro, definimos a **interface comum** `Component`. Esse é o contrato que **leafs** e **composites** compartilham:

```
public interface Component {  
    String getNome();  
    int obterTamanho();  
}
```

2. Criamos a **folha** (`Leaf`): o `Arquivo` , que implementa o comportamento **sem filhos**:

```
public class Arquivo implements Component {  
    private final String nome;  
    private final int tamanho;  
  
    public Arquivo(String nome, int tamanho) {  
        this.nome = nome;  
        this.tamanho = tamanho;  
    }  
  
    public String getNome() { return nome; }  
    public int obterTamanho() { return tamanho; }  
}
```

3. Criamos o **composto** (Composite): a Pasta guarda filhos e **delega** o cálculo:

```
public class Pasta implements Component {  
    private final String nome;  
    private final List<Component> filhos = new ArrayList<>();  
  
    public Pasta(String nome) { this.nome = nome; }  
    public String getNome() { return nome; }  
  
    public void adicionar(Component filho) { filhos.add(filho); }  
    public void remover(Component filho) { filhos.remove(filho); }  
  
    public int obterTamanho() {  
        int total = 0;  
        for (Component filho : filhos)  
            total += filho.obterTamanho();    // recursão!  
        return total;  
    }  
}
```

Utilização

O cliente **monta a árvore** usando sempre a mesma interface:

```
Pasta raiz = new Pasta("projeto");
raiz.adicionar(new Arquivo("README.md", 2));
raiz.adicionar(new Arquivo("pom.xml", 4));

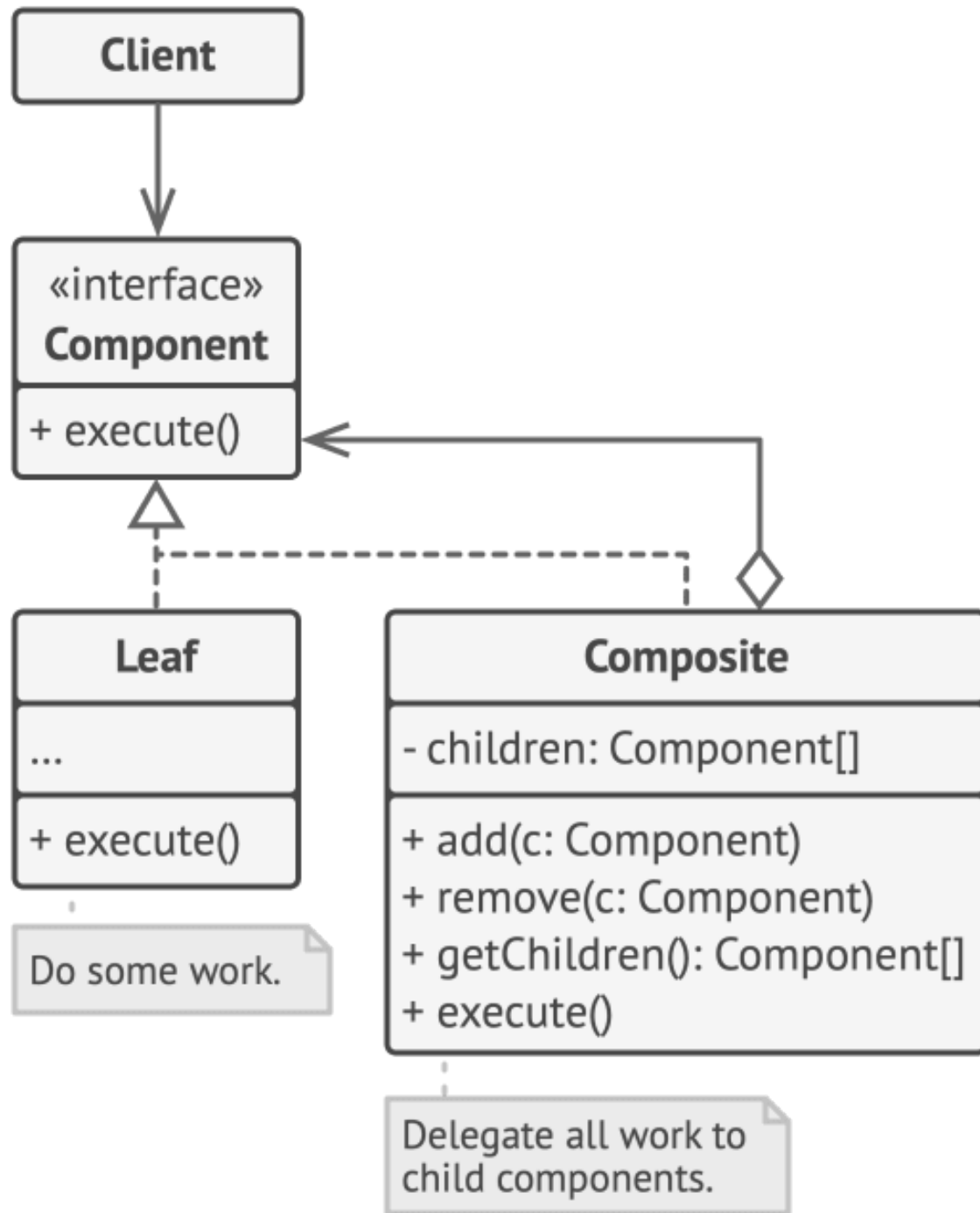
Pasta src = new Pasta("src");
src.adicionar(new Arquivo("Main.java", 8));
raiz.adicionar(src);

System.out.println(raiz.getNome() + " = " +
                   raiz.obterTamanho() + " KB");    // 14 KB
```

Diagramas

A estrutura envolve:

- **Component**: interface comum (operações + acesso aos filhos)
- **Leaf**: nó sem filhos, implementa as operações
- **Composite**: guarda filhos `Component`, delega e combina resultados
- **Client**: manipula a árvore pela interface `Component`



Analogias

- **Organograma de uma empresa:**
 - um departamento contém funcionários **ou** sub-departamentos
 - "quantas pessoas há abaixo de X?"
soma a árvore **recursivamente**
 - chefe e subordinado são, para o cálculo, **todos "funcionários"**
- O cliente pergunta ao **nó raiz** e ele resolve toda a hierarquia



Aplicações

- Representar hierarquias **parte-todo** recursivas
- Quando o cliente deve tratar **item e grupo de itens** de forma uniforme
- Processar a estrutura de forma **recursiva e genérica**

Exemplos típicos:

- **sistema de arquivos** (arquivos e pastas)
- **menus** com submenus e itens
- **DOM** de páginas web (elementos contêm elementos)
- **componentes de interface:** `java.awt` `Component` / `Container`

Prós

- **Uniformidade:** cliente não distingue folha de composição
- **Recursão natural:** a árvore se processa sozinha
- **OCP:** novos tipos de folha entram sem tocar no cliente
- Modela fielmente estruturas "parte-todo"

Contras

- A interface comum pode ficar **genérica demais** (métodos que a folha ignora)
- Dificulta **restringir** o que cada nó aceita (ex.: pasta só com arquivos)
- Nem toda hierarquia de objetos é um bom caso de uso

Regra: use quando a estrutura for **recursiva (parte-todo)** e o cliente precisar tratar **item e grupo uniformemente**.

Resumo

- Composite: **árvore parte-todo** com uma única interface
- **Folhas** implementam o comportamento; **compostos** guardam filhos e **recursam**
- O cliente trata tudo como `Component` , **sem** `instanceof`
- Ideal para estruturas recursivas: arquivos, menus, DOM, UI
- Diferença do **Decorator**: Composite agrega **vários** filhos (árvore); Decorator envolve **um** objeto (cadeia) somando responsabilidades