Prof. Me. Alexandre Henrick

Análise e Desenvolvimento de Sistemas - 6º período Sistemas de Informação

Composite

Descrição

O padrão de projeto Composite é um padrão estrutural que permite que objetos individuais e composições de objetos sejam tratados de maneira uniforme. Ele é usado para criar estruturas hierárquicas de objetos, onde os clientes podem tratar objetos individuais e composições de objetos de maneira semelhante.

Conseguimos criar essa hierarquia através do uso de uma classe que implementa uma interface em comum com os outros objetos (chamados de objetos simples). Essa classe será nosso "objeto composto", e irá delegar a execução das tarefas para os objetos simples. No composite temos 3 elementos importantes:

Interface:

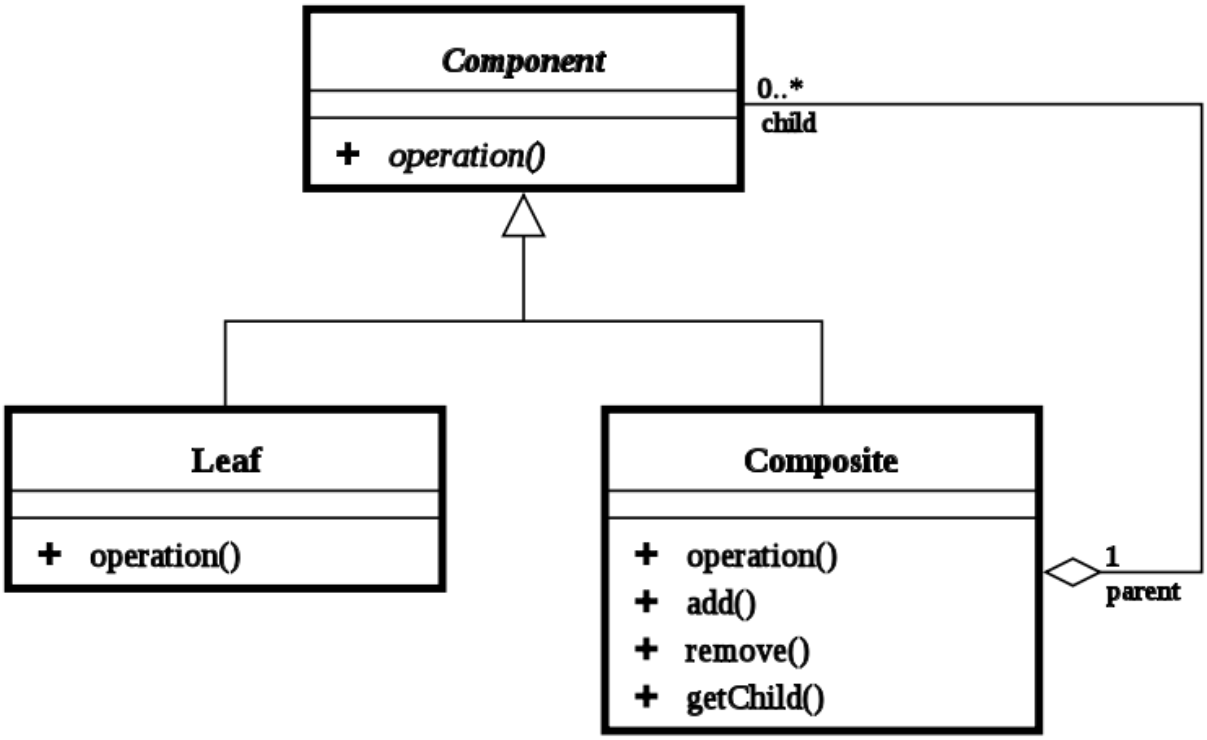
- O componente que será interface para os objetos simples e compostos;
- Ambos herdam dessa classe e a implementa

Folhas:

- São as classes concretas "folhas". Aqui criamos nosso **objetos simples**;
- Não possuem classes filhas;
- Geralmente são elas que executam as taregas, delegadas pelo objeto composto

Composite (objeto composto):

- Esse objeto precisa de um "conteiner" que armazena seus outros objetos;
- Esse conteiner geralmente é uma lista ou outra estrutura de dados parecida;
- Podem ser objetos simples ou compostos;
- O composite precisa de uma função para adicionar novos objetos ao seu conteiner e outro para remover



Motivações

1 - Tratamento uniforme: O Composite permite que os clientes tratem objetos simples e composições de objetos de maneira uniforme. Isso simplifica o código do cliente, pois ele não precisa distinguir entre objetos individuais e composições.

2 - Hierarquias de objetos: É útil quando você tem uma hierarquia de objetos que pode ser representada como uma árvore. Isso é comum em estruturas de GUI, estruturas organizacionais, elementos de documento e muito mais.

3 - Recursividade: O padrão Composite facilita a implementação de operações recursivas em estruturas de árvore. Por exemplo, percorrer todos os elementos de uma árvore e executar uma operação específica em cada um deles.

Pontos positivos

- Tratamento uniforme. O cliente pode interagir com objetos simples e compostos de maneira uniforme;
- Hierarquia de objetos. Conseguimos representar hierarquia de objetos;
- Composição dinâmica. Conseguimos adicionar novos objetos (compostos ou simples) e removê-los;
- Recursão simples. Podemos aplicar recursão em estruturas de hierarquia e maneira simplificada

Pontos negativos

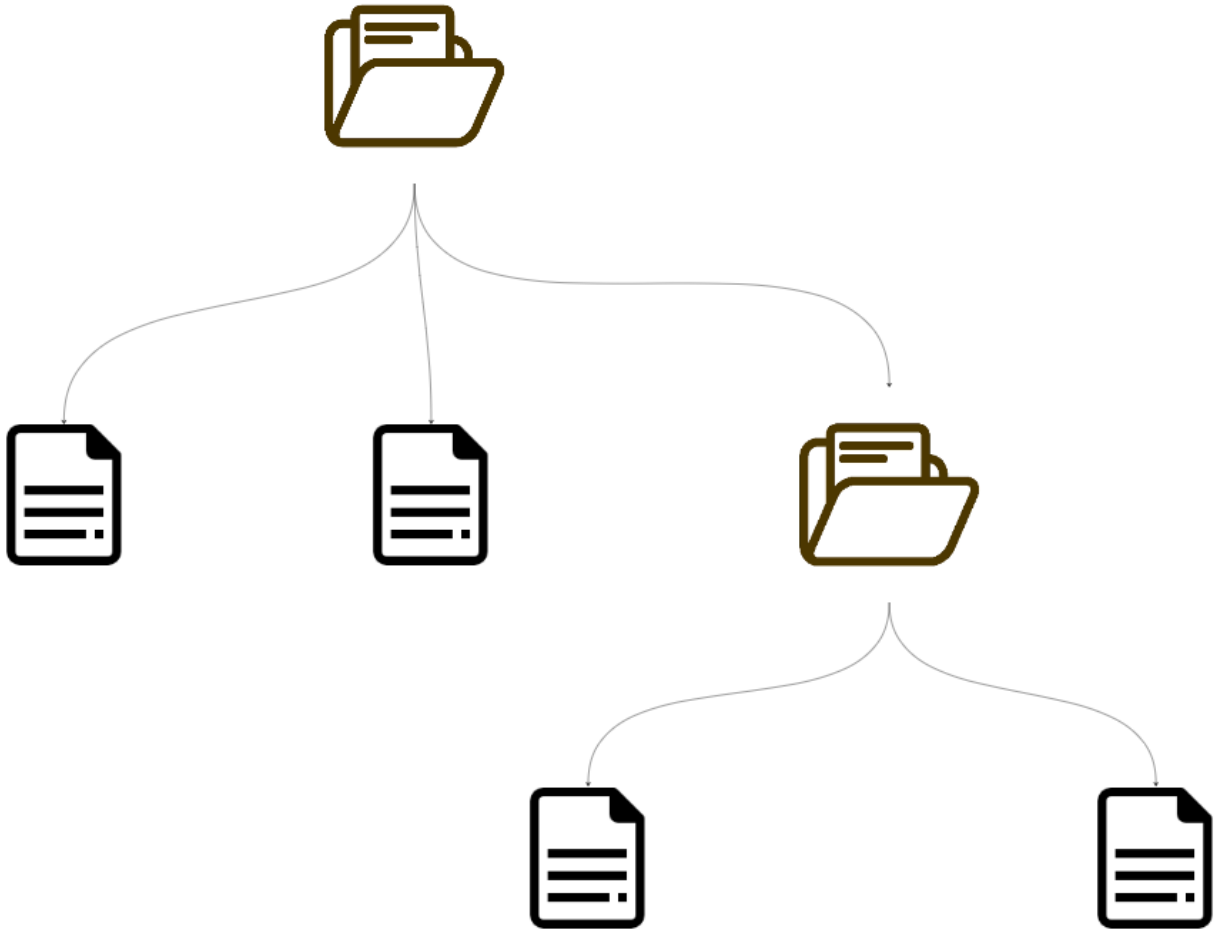
- Complexidade Potencial: Em algumas situações, o padrão Composite pode introduzir complexidade, especialmente se a hierarquia de objetos for muito grande ou se houver muitas operações a serem definidas no componente base.

- Restrições de Tipo: Em linguagens de programação que não suportam herança múltipla ou que têm restrições de tipo rígidas, a implementação do Composite pode ser mais complicada.
- Desempenho: Em algumas implementações, pode haver um pequeno impacto no desempenho devido à necessidade de navegar na hierarquia de objetos para realizar operações.

Cenário

Imagine um sistema de gerenciamento de arquivos, ou o próprio File System de seu sistema operacional. Nele podemos ter pastas e arquivos simples. O File System funciona como uma hierarquia. Dentro de uma pasta podemos ter arquivos simples ou outras pastas. No contexto do Composite, a pasta seria um objeto composto e os arquivos simples seriam os objetos simples.

Para criar um script que consiga interagir em uma estrutura como essa, podemos aplicar o padrão **Composite**.



Exemplo de código

```
from abc import ABC, abstractmethod

# Componente base
class FileSystemComponent(ABC):
    @abstractmethod
    def display(self, indent):
        pass
```

```python
# Classe Folha: representa um arquivo
class File(FileSystemComponent):
    def __init__(self, name):
        self.name = name

    def display(self, indent):
        print(f"{indent}Arquivo: {self.name}")

# Classe Composta: representa uma pasta
class Folder(FileSystemComponent):
    def __init__(self, name):
        self.name = name
        self.children = [] # Aqui vamos popular com objetos tipo
FileSystemComponent.

    def add(self, component):
        self.children.append(component)

    def remove(self, component):
        self.children.remove(component)

    def display(self, indent):
        print(f"{indent}Pasta: {self.name}")
        for child in self.children:
            child.display(indent + "  ")

# Criar uma estrutura de arquivos
root = Folder("Raiz")
folder1 = Folder("Documentos")
folder2 = Folder("Fotos")
file1 = File("documento.txt")
file2 = File("foto1.jpg")
file3 = File("foto2.jpg")

root.add(folder1)
root.add(folder2)
folder1.add(file1)
folder2.add(file2)
folder2.add(file3)

# Exibir a estrutura de arquivos
# Veja que ao chamar o display da pasta root, tantos os objetos compostos
quanto os simples
# são tratados da mesma maneira
root.display(indent=" ")
#file1.display(indent=" ")
```