Composite

Fernando Lucas da Silva

Necessidade

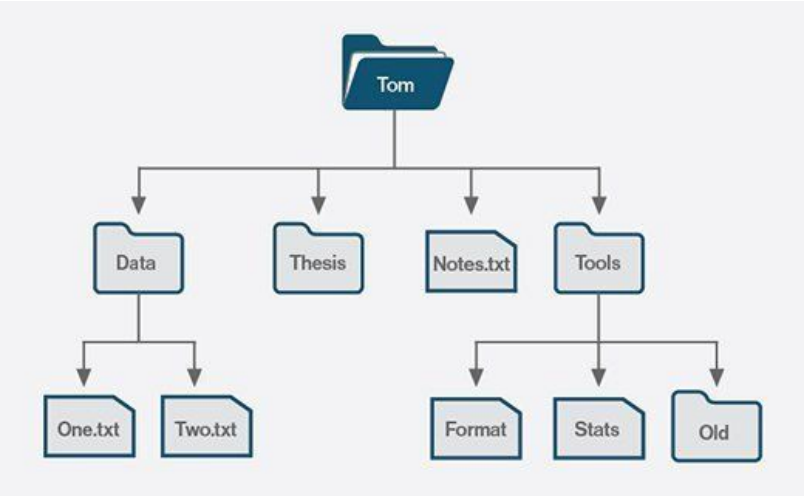
Em determinados cenários das nossas aplicações, precisamos trabalhar com **agrupamentos de objetos** de forma recursiva.

Cada objeto tem que conter outros objetos **internamente**, e seus “sub-objetos” também têm que se comportar **da mesma forma**.

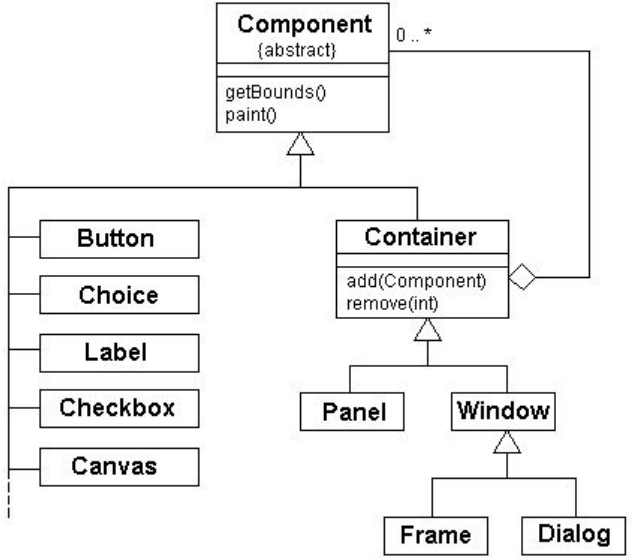
E assim, sucessivamente.

Por exemplo

Sistemas de arquivos

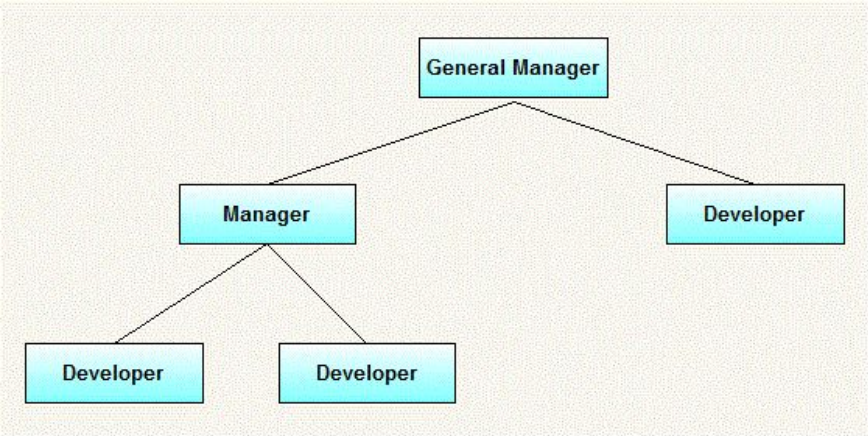


Interfaces Gráficas de Usuário (GUIs)

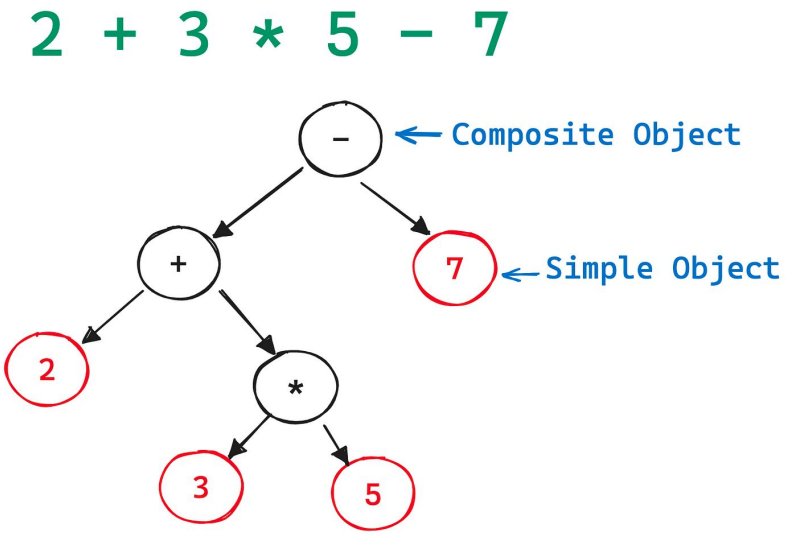


Por exemplo

Estruturas organizacionais



Expressões matemáticas



Tá, mas usar **só a composição** não resolve?

Não :)

Isso porque, nessa estrutura de objetos **hierárquica** (como uma árvore), temos os conceitos de:

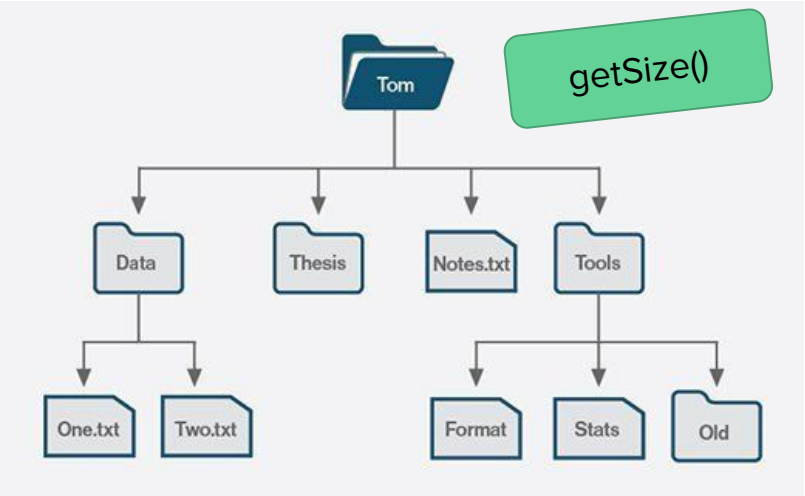
- Objetos **individuais**
- **Grupos** de objetos

E precisamos que ambos **compartilhem** os mesmos **comportamentos**. A simples composição **não garante isso**, só garante que o objeto *B* é atributo do objeto *A*.

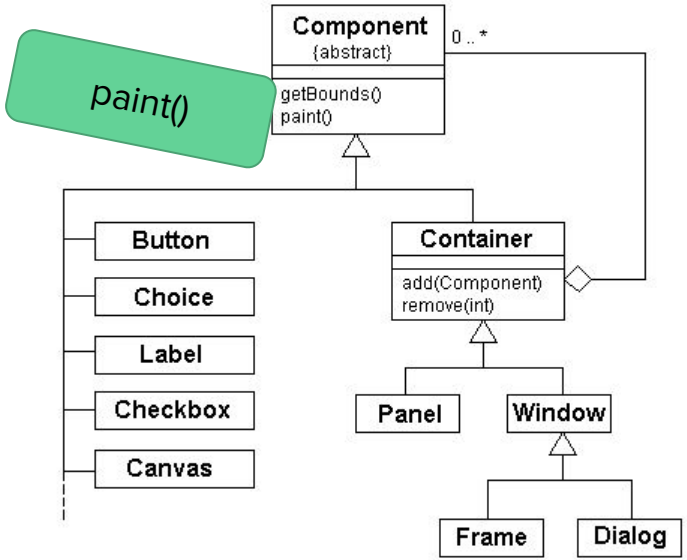
O objetivo é que o código cliente **não precise saber** se está lidando com um objeto simples ou um container complexo.

Voltando aos exemplos...

Sistemas de arquivos

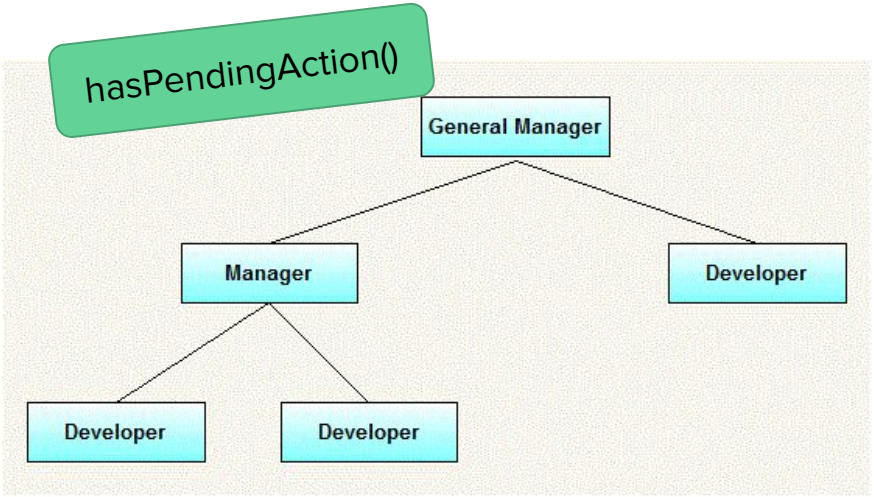


Interfaces Gráficas de Usuário (GUIs)

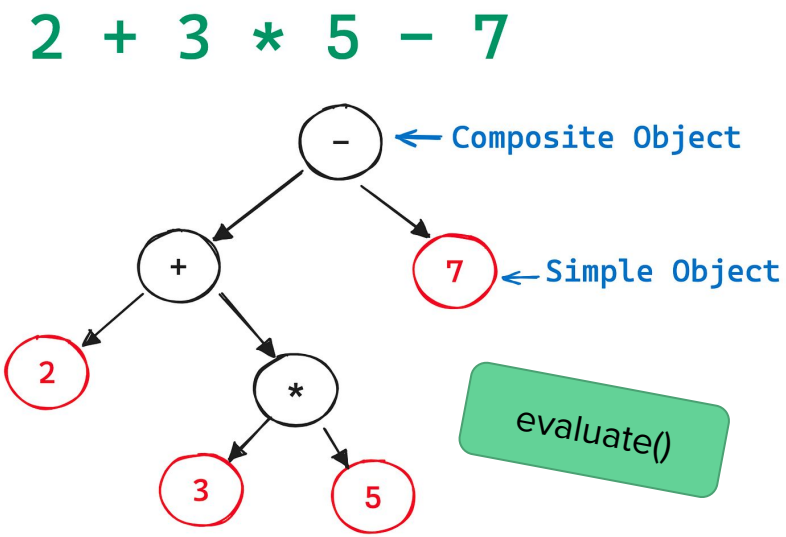


Voltando aos exemplos...

Estruturas organizacionais



Expressões matemáticas



Problema (sem composite ainda)

Como lidar com a **diferença** entre *folhas* e *containeres*?

Se tivéssemos que tratar cada tipo de objeto **separadamente**, o código ficaria repleto de **verificações**:

- Se é uma **folha**, aplicamos a operação **diretamente**
- Se é um **container**, precisamos percorrer recursivamente todos os seus **filhos** e aplicar a operação em **cada um**

Essa lógica de diferenciação **suja o código** cliente com o gerenciamento da estrutura de árvore, quebrando o princípio da **responsabilidade única**.

Então é só usar uma interface comum em todos, né?

Mais ou menos :)

Com esse conceito, chegamos beeeem mais perto do **Composite**.

Recapitulando, precisamos que tanto containers quanto folhas tenham **métodos em comum**, só temos que nos atentar para que sua chamada seja **recursiva**.

Isso quer dizer que o método no container deve **chamar** e **usar** esse mesmo método nos itens internos.

Por exemplo, chamar o **getSize()** de uma pasta implica em chamar o getSize() de todas as suas subpastas e arquivos e **somar seus resultados**.

Senhoras e senhores, temos o Composite.

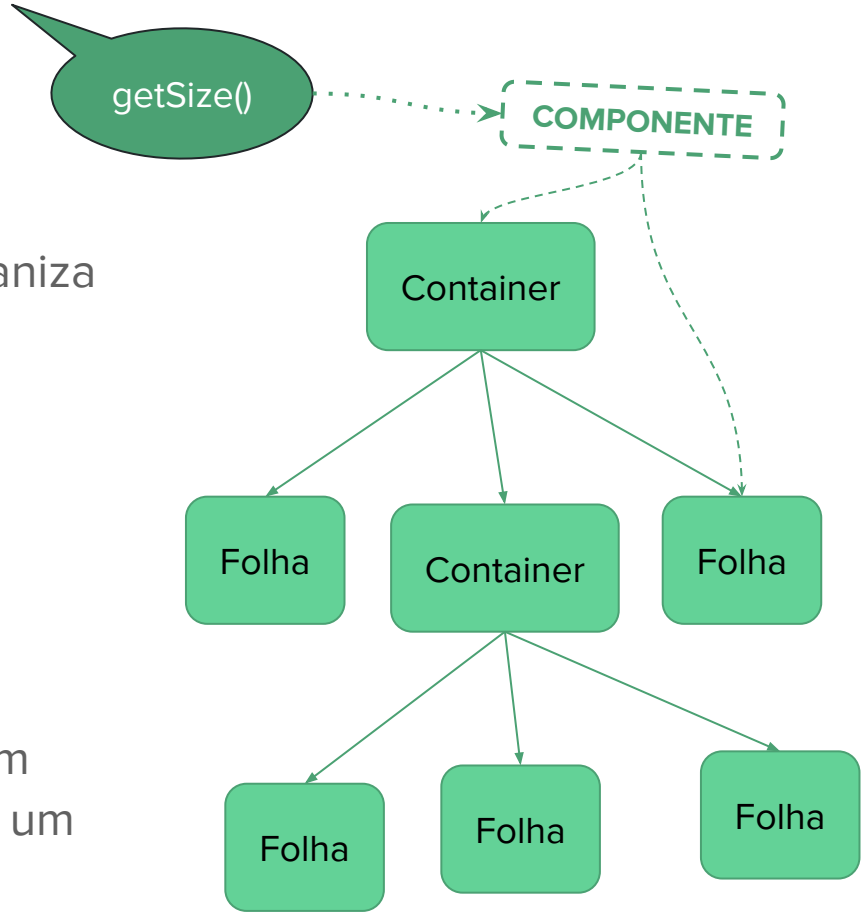
Exemplo do Refactoring Guru

Explorando o **Composite**

É um padrão de projeto **estrutural** que organiza objetos em estruturas de **árvore** para representar **hierarquias** parte-todo.

Ele permite que os clientes tratem objetos individuais e composições de objetos de maneira **uniforme**.

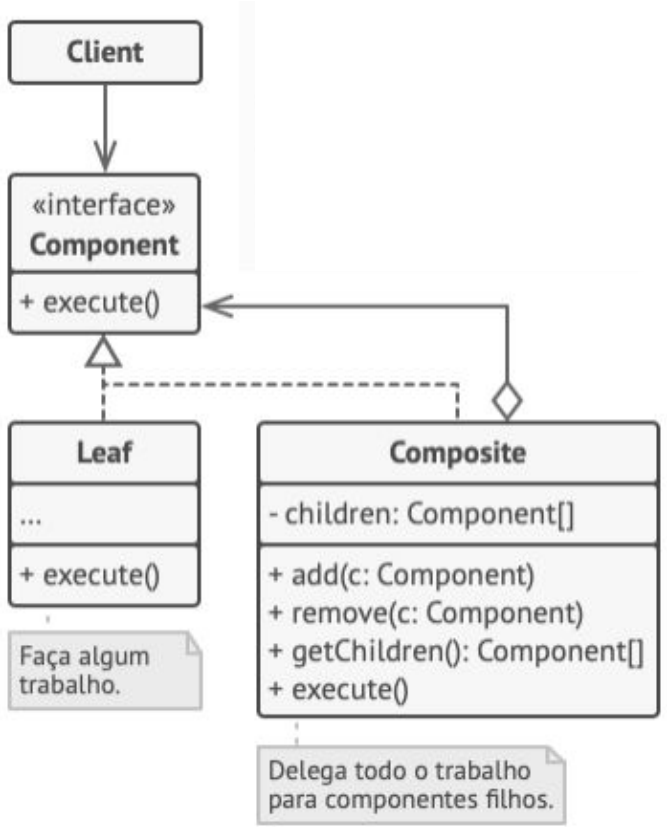
O cliente interage apenas com a interface **Componente**. Se ele chama getSize() em um objeto, ele **não precisa saber** se o objeto é um único arquivo ou um diretório inteiro.



Estrutura (implementação)

Nesse exemplo do Refactoring Guru, observamos:

- Interface **Component**: define os métodos em comum
- **Composite**: objetos que podem ter componentes filhos
- **Leaf**: objetos "folha" da composição, que não podem ter filhos
- **Client**: interage com os objetos através da interface Component



Vamos à prática!