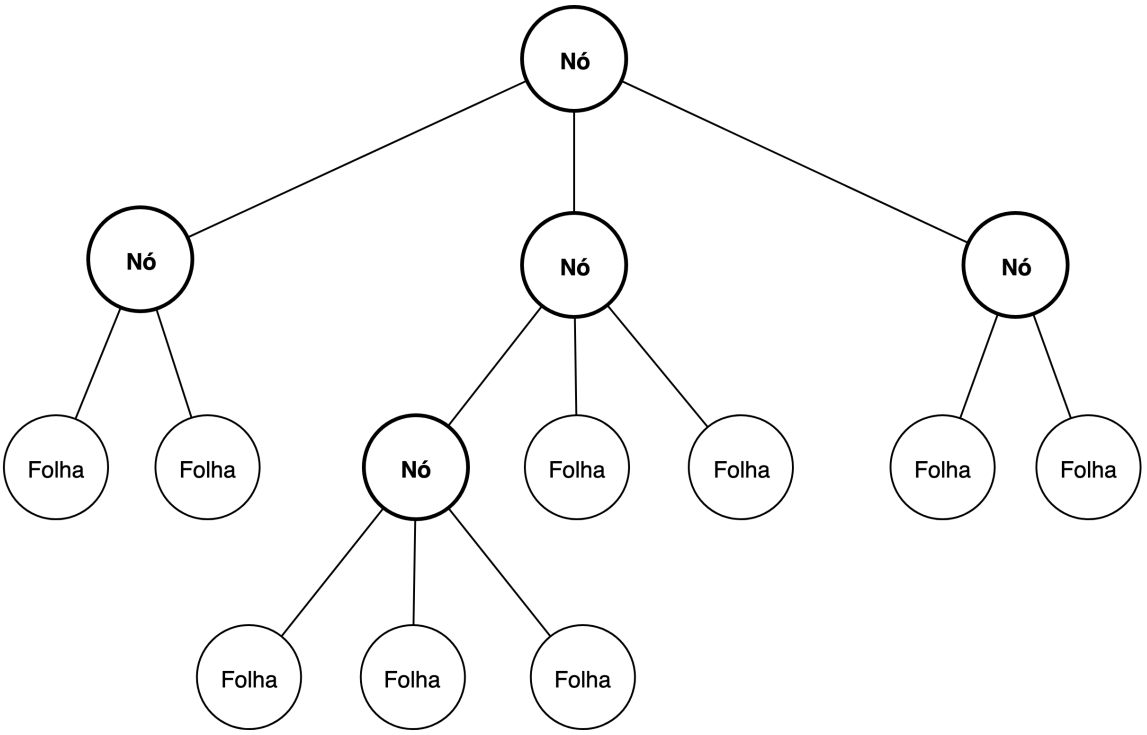
Composite

Padrões Estruturais

O padrão *composite* permite a composição de objetos em estruturas de árvore para representar hierarquias parte-todo. Com esse padrão, os clientes podem tratar objetos individuais ou composições de objetos de maneira transparente e uniforme.

Motivação (Por que utilizar?)

A imagem abaixo ilustra uma estrutura de dados árvore.



Os elementos com filhos são chamados de **Nós** e os elementos sem filhos são chamados de **folhas.**

Há momentos em que um software precisa manipular uma estrutura de dados em árvore, e para evitar complexidade, é necessário tratar os nós e as folhas de maneira uniforme.

Considere, por exemplo, um sistema de arquivos. Pode ser uma estrutura em árvore que contém nós (pastas), e folhas (arquivos). Um objeto de pasta geralmente contém um ou mais objetos de arquivo ou outras pastas (subpastas), portanto, uma pasta é um objeto complexo e um arquivo é um objeto simples.

Pasta 1
- Arquivo 1
- Arquivo 2

Pasta 2
- Arquivo 3
- Arquivo 4

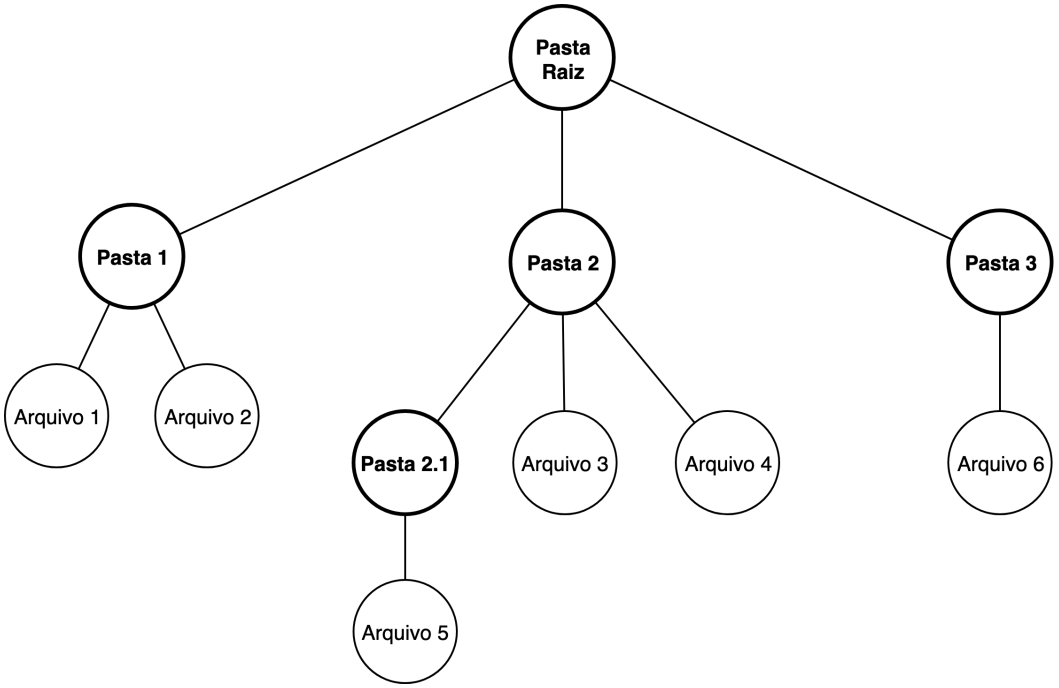
Pasta 2.1
-- Arquivo 5

Pasta 3
- Arquivo 6

Hierarquia de diretórios

Considere a hierarquia do sistema de diretórios acima, nosso objetivo a nível de explicação é organizar essa estrutura e tornar possível a inclusão, remoção e consulta de arquivos, tal consulta pode ser a nível geral, por pastas ou subpastas.

Este mesmo sistema de arquivos organizado como árvore ficaria da seguinte forma:



Hierarquia de diretórios organizada em árvore

O padrão *Composite* propõe que pastas e arquivos sejam tratados pelo cliente de maneira uniforme e transparente, evitando a utilização de operações condicionais (`if`) e teste de tipo de objeto (`instanceof`) feitos pelos clientes.

Devemos criar uma classe que será comum aos arquivos (folhas) e pastas (nós), ou seja, eles terão o mesmo supertipo.

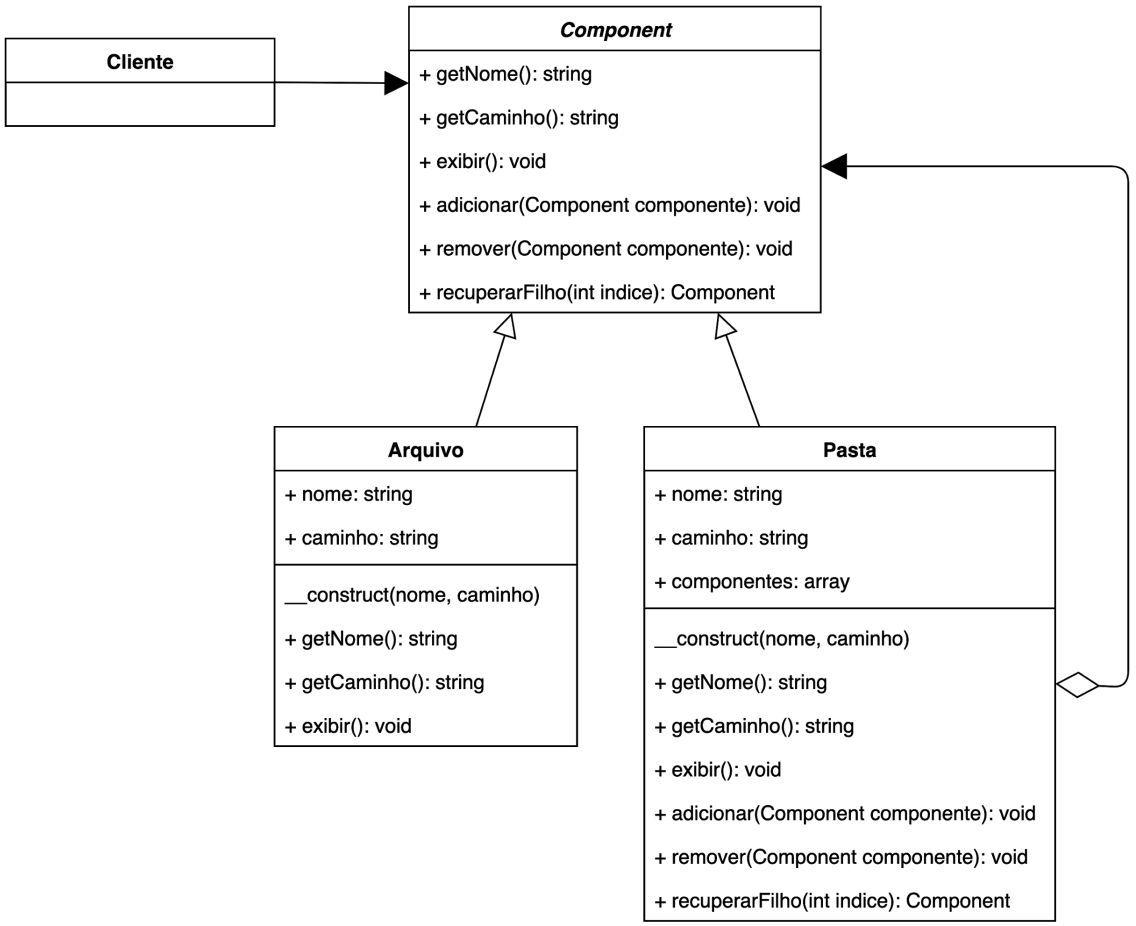


Diagrama de Classes do gerenciador de arquivos

Embora herdem características da mesma classe, os arquivos e pastas são diferentes. Um arquivo não possui filhos, portanto, para arquivos, `adicionar()`, `remover()` ou `recuperarFilho()` são métodos inadequados.

No diagrama de classes dos gerenciador de arquivos não existem métodos de `Component` que são inadequados a classe `Pasta`, pois mesmo sendo diferente de um arquivo, uma pasta ainda possui um nome, caminho e pode ser exibida. Porém, vale ressaltar que dependendo do contexto, podem existir métodos no `Component` que serão inadequados aos nós (`Pasta`) e/ou folhas (`Arquivo`).

Para tratar esses métodos inadequados vamos criar uma exceção que informa ao cliente que a operação solicitada não é suportada.

```
class UnsupportedOperationException extends PhpLogicException
{
}
```

Deste modo, caso o cliente peça para adicionar um arquivo em outro arquivo, ou remover um filho de um arquivo lançaremos essa exceção, pois são operações inadequadas a um arquivo.

Vamos optar por criar uma classe abstrata **Component** que possui todos os métodos que uma pasta ou arquivo podem precisar. A princípio todos os métodos irão lançar a exceção **UnsupportedOperationException**.

```
abstract class Component
{
    public function adicionar(Component $componente): void
    {
        throw new \UnsupportedOperationException();
    }

    public function remover(Component $componente): void
    {
        throw new \UnsupportedOperationException();
    }

    public function recuperarFilho(int $indice): Component
    {
        throw new \UnsupportedOperationException();
    }

    public function exibir(): void
    {
        throw new \UnsupportedOperationException();
    }

    public function getNome(): string
    {
        throw new \UnsupportedOperationException();
    }

    public function getCaminho(): string
    {
        throw new \UnsupportedOperationException();
    }
}
```

Agora cada subclasse irá sobrescrever (*Override*) apenas os métodos que fazem sentido para ela. Assim a subclasse irá utilizar as implementações padrão de **Component** para os métodos que são inadequados a ela, portanto, se o cliente tentar fazer uma operação inadequada uma exceção será lançada.

```php
class Arquivo extends Component
{
    public string $nome;
    public string $caminho;

    //Recebe o nome e o caminho do arquivo como parâmetro e os inicializam.
    public function __construct(string $nome, string $caminho)
    {
        $this->nome = $nome;
        $this->caminho = $caminho;
    }

    //Retorna o nome do arquivo.
    public function getNome(): string
    {
        return $this->nome;
    }

    //Retorna o caminho do arquivo.
    public function getCaminho(): string
    {
        return $this->caminho;
    }

    //Exibe o nome do arquivo formatado.
    public function exibir(): void
    {
        echo "  - " . $this->getNome() . ' - [' . $this->getCaminho() . '] </br>';
    }
}
```

Repare que a classe **Arquivo** estende a classe **Component** e não substitui os métodos **adicionar()**, **remover()** e **recuperarFilho()** , então, caso o cliente tente os utilizar a exceção **UnsupportedOperationException** será lançada.

Por outro lado a **Classe** Pasta deverá substituir todos os métodos de **Component** , pois, uma **Pasta** possui um nome e caminho, além de ter filhos, ou seja, deverá ser possível adicionar, remover e recuperar um objeto **Arquivo** que está contido em uma **Pasta**.

```php
class Pasta extends Component
{
    public string $nome;
    public string $caminho;
    public array $componentes;

    //Recebe o nome e o caminho da pasta.
    public function __construct(string $nome, string $caminho)
    {
        $this->nome = $nome;
        $this->caminho = $caminho;
        $this->componentes = [];
    }

    //Adiciona uma pasta ou arquivo a pasta.
    public function adicionar(Component $componente): void
    {
        $this->componentes[] = $componente;
    }

    //Remove uma pasta ou arquivo da pasta.
    public function remover(Component $componente): void
    {
        foreach ($this->componentes as $indice => $arquivo) {
            if ($arquivo === $component) {
                unset($this->componentes[$indice]);
            }
        }
    }

    //Recupera uma pasta ou arquivo da pasta.
    public function recuperarFilho(int $indice): Component
    {
        return $this->componentes[$indice];
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getCaminho(): string
    {
        return $this->caminho;
    }

    //Exibe o nome e caminho da pasta e também os arquivos contidos nela.
    public function exibir(): void
    {
        echo '<div style="padding-left: 20px"><pre>';
        echo '## ' . $this->getNome() . ' - [' . $this->getCaminho() . '] ##</br>';
        echo '---------------------------------<br>';

        //Para cada arquivo chame exibir();
        foreach ($this->componentes as $componente) {
            $componente->exibir();
        }
        echo '</pre></div>';
    }
}
```

Temos arquivos e pastas implementados, precisamos agora inicializar essas classes para organizar nossa estrutura de diretórios. Vamos criar a classe **GerenciadorDeArquivos** que recebe como parâmetro a **Pasta** raiz de nossa árvore.

```
class GerenciadorDeArquivos
{
    private Component $raiz;

    public function __construct(Component $raiz)
    {
        $this->raiz = $raiz;
    }

    public function exibirTodos(): void
    {
        $this->raiz->exibir();
    }
}
```

O método **exibirTodos()** chama o método exibir da pasta raiz, a partir daí um processamento recursivo é iniciado e todas as pastas e arquivos serão exibidos.

Vale ressaltar que a classe **GerenciadorDeArquivos** não faz parte do padrão *Composite*, aqui ela está desempenhando um papel de cliente.

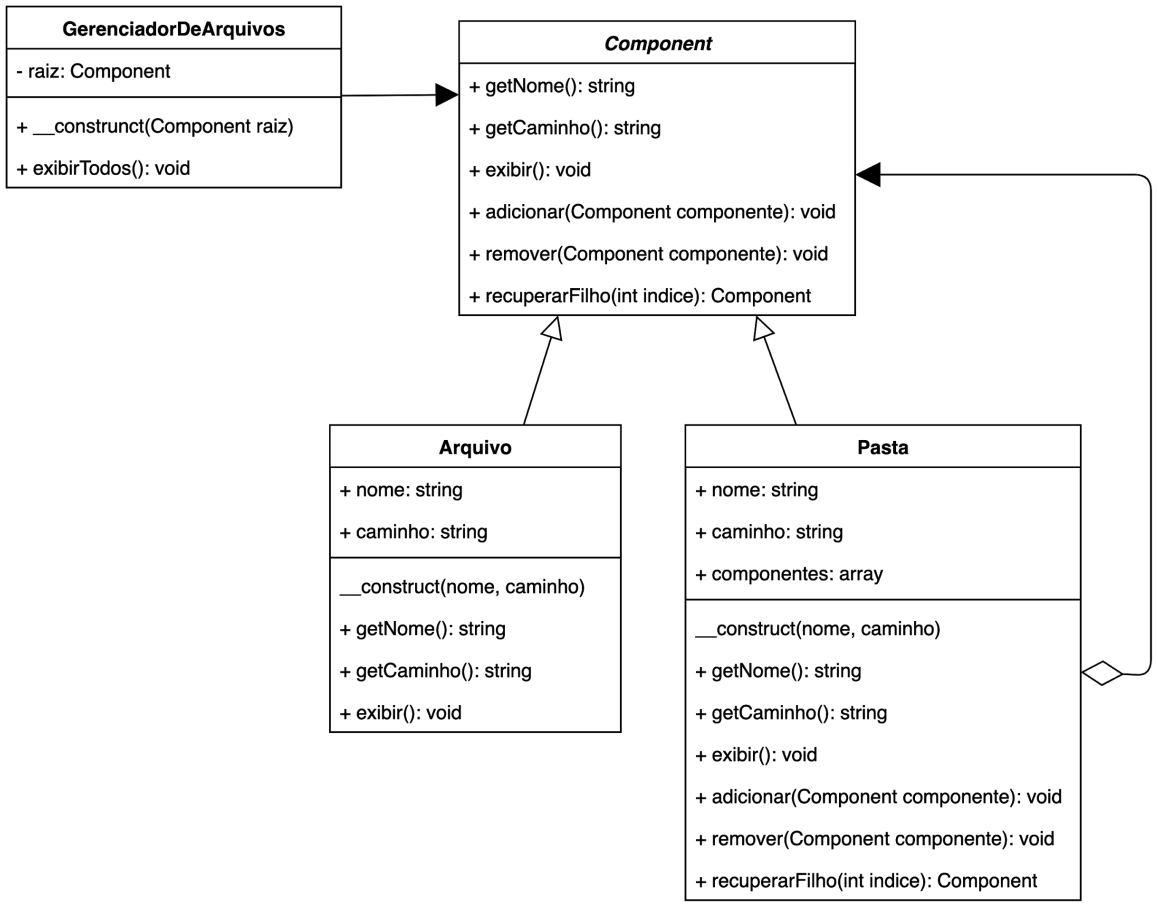


Diagrama de Classes completo do exemplo

Vamos aos testes:

```
//Criação da pasta raiz e demais pastas
$raiz = new Pasta('Raiz', '/');
$pasta1 = new Pasta('Pasta 1', 'pasta1/');
$pasta2 = new Pasta('Pasta 2', 'pasta2/');
$pasta2_1 = new Pasta('Pasta 2.1', 'pasta2-1/');
$pasta3 = new Pasta('Pasta 3', 'pasta3/');

//Adição das pastas 1,2 e 3 a pasta raiz, conforme dita nossa Hierarquia de diretórios.
$raiz->adicionar($pasta1);
$raiz->adicionar($pasta2);
$raiz->adicionar($pasta3);

//Adicição da pasta 2.1 a pasta 2 conforme dita nossa Hierarquia de diretórios.
$pasta2->adicionar($pasta2_1);

//Criação dos arquivos.
$arquivo1 = new Arquivo('Arquivo 1', '/arquivo1');
$arquivo2 = new Arquivo('Arquivo 2', '/arquivo2');
$arquivo3 = new Arquivo('Arquivo 3', '/arquivo3');
$arquivo4 = new Arquivo('Arquivo 4', '/arquivo4');
$arquivo5 = new Arquivo('Arquivo 5', '/arquivo5');
$arquivo6 = new Arquivo('Arquivo 6', '/arquivo6');

//Adição dos arquivos as pastas conforme dita nossa Hierarquia de diretórios.
$pasta1->adicionar($arquivo1);
$pasta1->adicionar($arquivo2);
$pasta2->adicionar($arquivo3);
$pasta2->adicionar($arquivo4);
$pasta2_1->adicionar($arquivo5);
$pasta3->adicionar($arquivo6);
//Criação do gerenciador de arquivos e atribuição da pasta raiz a ele.
$gerenciador = new GerenciadorDeArquivos($raiz);

//Exibição de todas as pastas e arquivos
$gerenciador->exibirTodos();
```

Resultado:

```
## Raiz - [/] ##
---------------------------------
    ## Pasta 1 - [pasta1/] ##
    ---------------------------------
         - Arquivo 1 - [/arquivo1]
         - Arquivo 2 - [/arquivo2]
    ## Pasta 2 - [pasta2/] ##
    ---------------------------------
        ## Pasta 2.1 - [pasta2-1/] ##
        ---------------------------------
             - Arquivo 5 - [/arquivo5]
         - Arquivo 3 - [/arquivo3]
         - Arquivo 4 - [/arquivo4]
    ## Pasta 3 - [pasta3/] ##
    ---------------------------------
         - Arquivo 6 - [/arquivo6]
```

Aplicabilidade (Quando utilizar?)

- Quando é necessário representar uma hierarquia parte-todo de objetos.
- Quando é preciso que os clientes possam ignorar a diferença entre composições de objetos e objetos individuais, ou seja, os clientes devem tratar todos os objetos na estrutura composta de maneira uniforme.

Componentes

- **Cliente:** O cliente utiliza a interface Componente para manipular os objetos da composição.
- **Componente:** Define uma interface para todos os objetos da composição, o que inclui tanto os compostos quanto as folhas. Pode definir uma interface para acessar o pai de um componente na estrutura recursiva e a implementa, se apropriado.
- **Folha:** Define o comportamento dos componentes primitivos, ou seja, que não possuem filhos. Isto é feito implementando as operações que o Componente aceita.
- **Composite:** Também implementa as operações do Componente, o papel do composite é definir o comportamento dos componentes que possuem filhos. O Composite armazena seus filhos

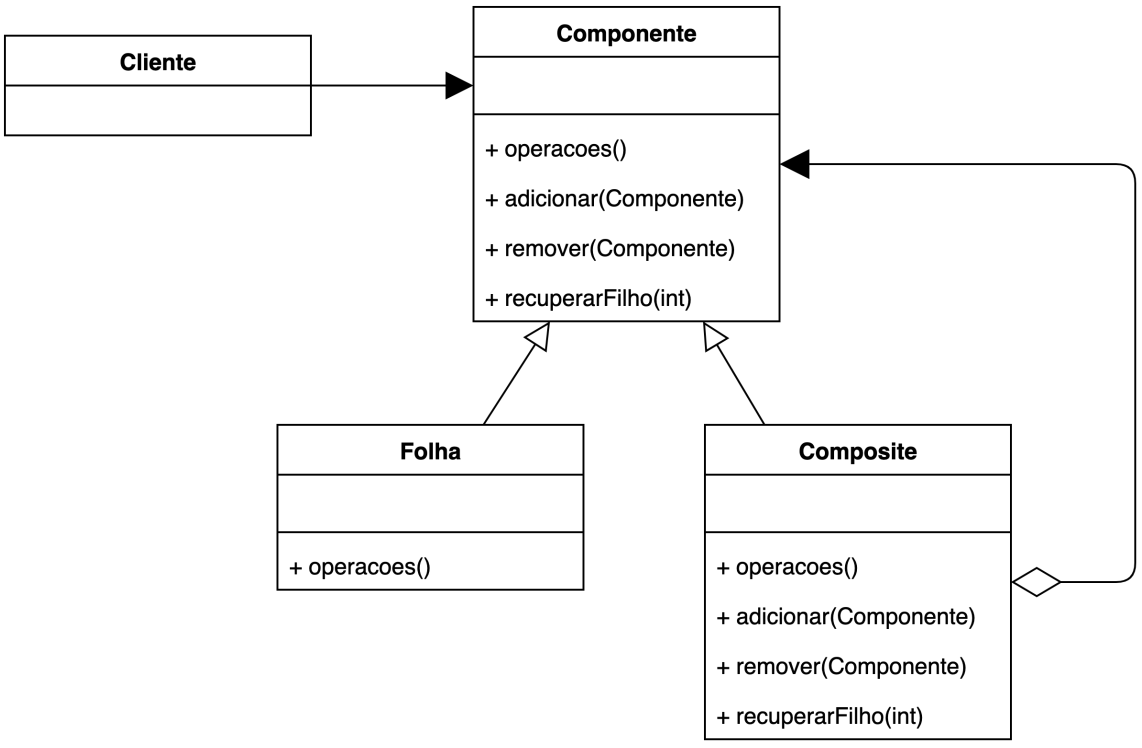


Diagrama de Classes

Consequências

- Definição de hierarquias de classes que consistem em objetos primitivos (folhas) e objetos compostos (nós). Objetos primitivos podem compor objetos mais complexos, que por sua vez podem compor outros objetos e assim por diante recursivamente. Sempre que o cliente espera um objeto primitivo, ele também pode receber um objeto composto.

- Os clientes podem tratar estruturas compostas e objetos individuais de maneira uniforme. Normalmente, os clientes não sabem (e não devem se importar) se estão lidando com uma folha ou um componente composto.

- Facilita a adição de novos tipos de componentes. Os clientes não precisam ser alterados para novas classes de componentes sejam criadas.

- Pode tornar a arquitetura do código excessivamente genérica. A desvantagem de facilitar a adição de novos componentes é a dificuldade de restrição dos tipos de componentes que um composto pode ter. Às vezes, é preciso que um composto tenha apenas determinados componentes. Com o *Composite*, não se pode confiar no sistema de tipos para impor essas restrições. Para isso será necessário realizar verificações em tempo de execução.

  !importante: Para contornar o problema citado acima, é possível separar a classe Componente em duas classes, uma para as folhas e outra para os Compostos (*Composite*) com seus respectivos métodos. Esta alternativa traz mais segurança ao código em relação à verificação dos tipos dos objetos. Por outro lado, a transparência por parte do cliente é perdida, já que assim que receber um objeto, ele terá que verificar se é um composto ou uma folha.

  A decisão de qual abordagem seguir vai de acordo com as necessidades e especificações de seu projeto. Se o mais importante é a transparência e baixa complexidade ou a segurança em relação aos tipos dos objetos.