Padrões de Projeto: Composite

Jonathan Arantes

12 de setembro de 2017

Instituto Federal de Minas Gerais - Campus Formiga

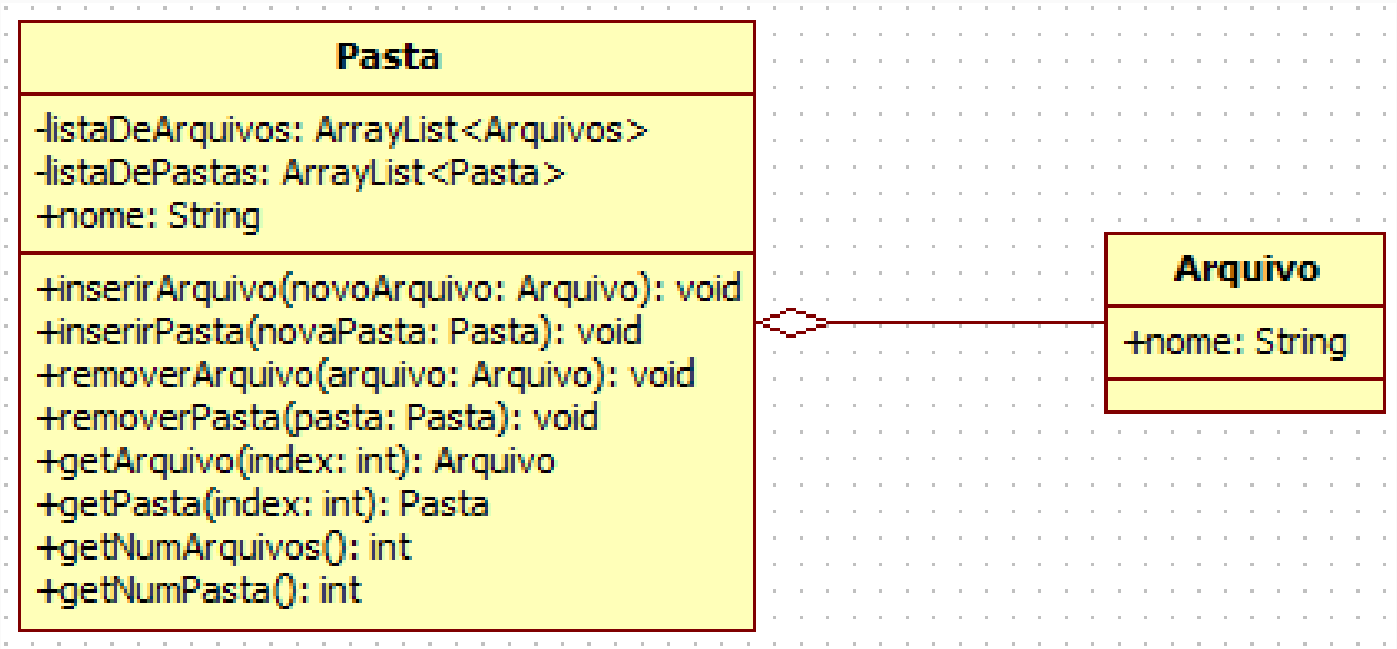
Problema

Problema

Imagine que você está fazendo um sistema de gerenciamento de arquivos. Como você já sabe é possível criar arquivos concretos (vídeos, textos, imagens, etc.) e arquivos pastas, que armazenam outros arquivos. O problema é o mesmo, como fazer um design que atenda estes requerimentos?

Uma Solução

Podemos criar uma classe que representa arquivos que são Pastas, estas pastas teriam uma lista de arquivos concretos e uma lista de arquivos de pastas. Então poderíamos adicionar pastas e arquivos em uma pasta e, a partir dela navegar pelas suas pastas e seus arquivos.



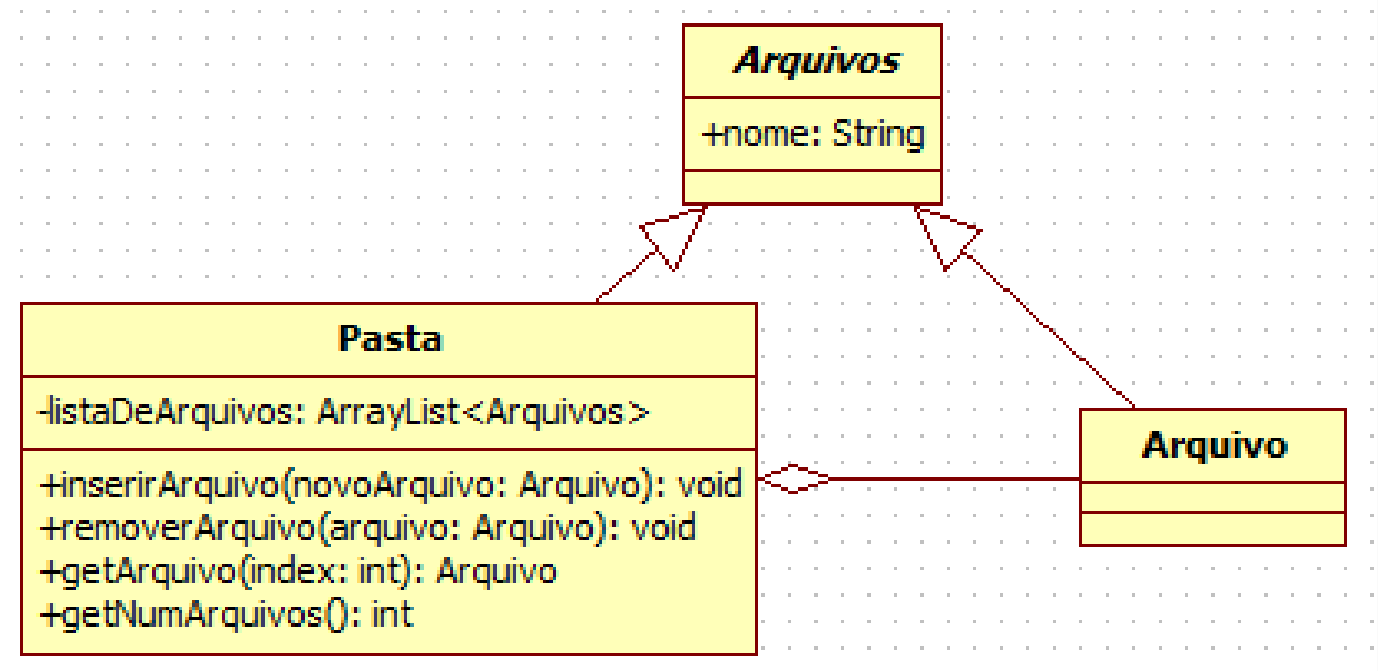
Uma Solução

O problema com este design é a interface que esta classe deverá ter. Como são duas listas diferentes precisaríamos de métodos específicos para tratar cada uma dela, ou seja, um método para inserir arquivos e outro para inserir pastas, um método para excluir pastas e outro para excluir arquivos, e assim vai.

Sempre que quisermos inserir uma nova funcionalidade no gerenciador precisaremos criar a mesma funcionalidade para arquivos e pastas. Além disso, sempre que quisermos percorrer uma pasta será necessário percorrer as duas listas, mesmo que vazias.

Uma Solução

Bom, vamos pensar em outra solução. E se utilizássemos uma classe base Arquivo para todos os arquivos, assim precisaríamos apenas de uma lista e de um conjunto de funções. Vejamos abaixo:



Uma Solução

Pronto, resolvido o problema dos métodos duplicados. E agora, será que está tudo bem? Como faríamos a diferenciação entre um Arquivo e uma Pasta? Poderíamos utilizar o "instace of" e verificar qual o tipo do objeto, o problema é que seria necessário fazer isso SEMPRE, pois não poderíamos confiar que, dado um objeto qualquer, ele é um arquivo ou uma pasta! Sempre teríamos que fazer isso:

```
1       if(arquivo instanceof Arquivo){
2       // Codigo para tratar arquivos de video
3       } else if(arquivo instanceof Pasta){
4       // Codigo para tratar arquivos de audio
5       }
```

Ok, então vamos ver uma boa solução para o problema: o padrão Composite!

Composite

Composite

Vamos ver então qual a intenção do padrão Composite:

"Compor objetos em estruturas de árvore para representar hierarquia partes-todo. Composite permite aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos."

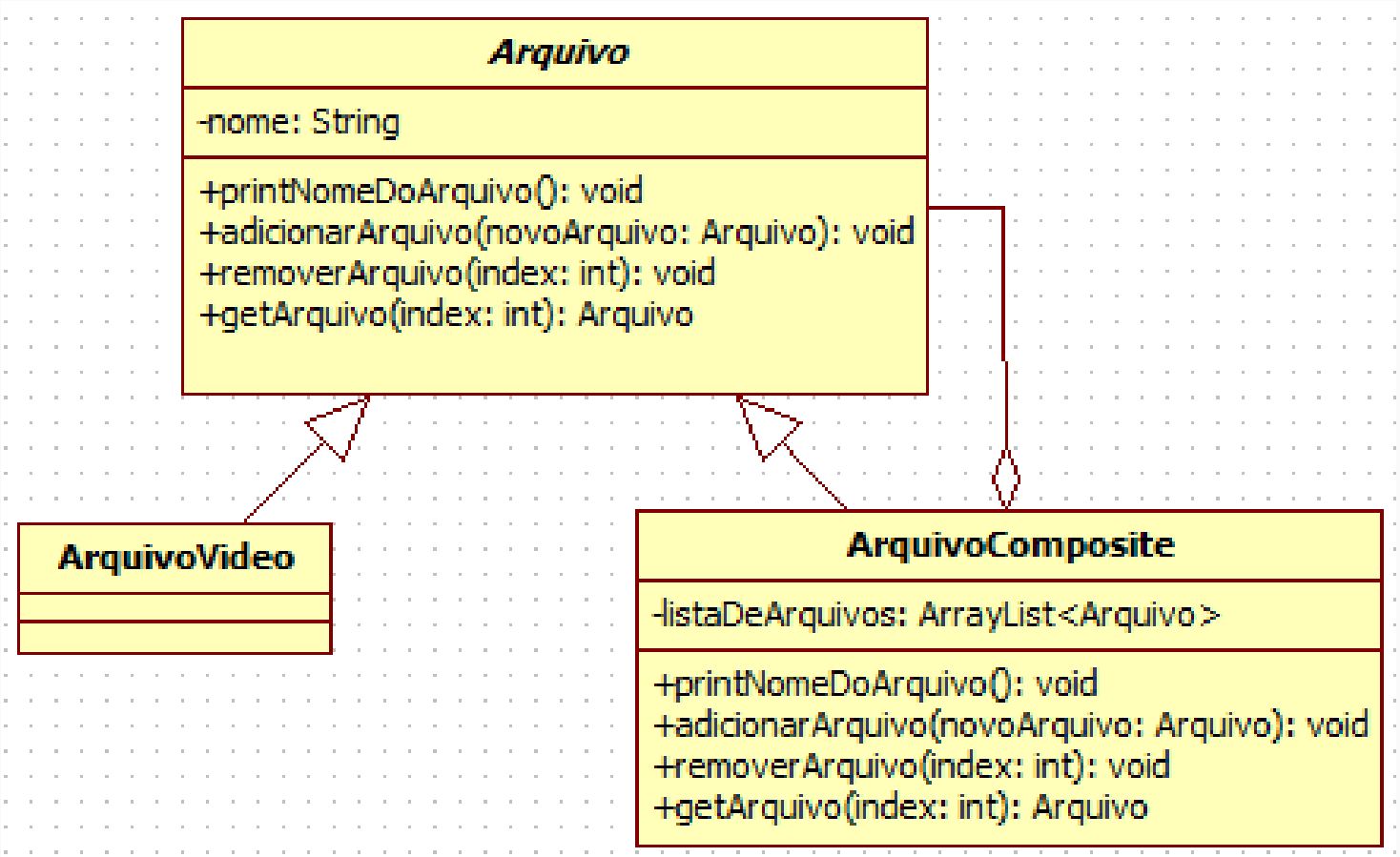
Bom, desta vez a intenção não está tão bem clara. A estrutura de árvore será explicada mais adiante, no momento o que interessa é a segunda parte da inteção: tratar de maneira uniforme objetos individuais.

Composite

Como o nosso problema era uniformizar o acesso aos arquivos e pastas, provavelmente o Composite seja uma boa solução.

A ideia do Composite é criar uma classe base que contém toda a interface necessária para todos os elementos e criar um elemento especial que agrega outros elementos. Vamos trazer para o nosso exemplo inicial para tentar esclarecer:

Composite


Arquivo
-nome: String
+printNomeDoArquivo(): void
+adicionarArquivo(novoArquivo: Arquivo): void
+removerArquivo(index: int): void
+getArquivo(index: int): Arquivo
ArquivoVideo
ArquivoComposite
-listaDeArquivos: ArrayList<Arquivo>
+printNomeDoArquivo(): void
+adicionarArquivo(novoArquivo: Arquivo): void
+removerArquivo(index: int): void
+getArquivo(index: int): Arquivo

Composite

A classe base Arquivo implementa todos os métodos necessários para arquivos e pastas, no entanto considera como implementação padrão a do arquivo, ou seja, caso o usuário tente inserir um arquivo em outro arquivo uma exceção será disparada. Veja o código da classe abaixo:

```
6   public abstract class ArquivoComponent {
7
8       String nomeDoArquivo;
9
10      public void printNomeDoArquivo() {
11          System.out.println(this.nomeDoArquivo);
12      }
13
14      public String getNomeDoArquivo() {
15          return this.nomeDoArquivo;
16      }
17
18      public void adicionar(ArquivoComponent novoArquivo) throws Exception {
19          throw new Exception("Nao pode inserir arquivos em: "
20                  + this.nomeDoArquivo + " - Nao e uma pasta");
21      }
22
23      public void remover(String nomeDoArquivo) throws Exception {
24          throw new Exception("Nao pode remover arquivos em: "
25                  + this.nomeDoArquivo + " -Nao e uma pasta");
26      }
27
28      public ArquivoComponent getArquivo(String nomeDoArquivo) throws Exception {
29          throw new Exception("Nao pode pesquisar arquivos em: "
30                  + this.nomeDoArquivo + " - Nao e uma pasta");
31      }
32  }
```

Uma vez que tudo foi definido nesta classe, para criar um arquivo de vídeo por exemplo, basta implementar o construtor:

```
1
2   public class ArquivoVideo extends ArquivoComponent {
3
4       public ArquivoVideo(String nomeDoArquivo) {
5           this.nomeDoArquivo = nomeDoArquivo;
6       }
7   }
```

Já na classe que representa a Pasta nós sobrescrevemos o comportamento padrão e repassamos a chamada para todos os arquivos, sejam arquivos ou pastas, como podemos ver a seguir:

```
1
2  public class ArquivoComposite extends ArquivoComponent {
3
4      ArrayList<ArquivoComponent> arquivos = new ArrayList<ArquivoComponent>();
5
6      public ArquivoComposite(String nomeDoArquivo) {
7          this.nomeDoArquivo = nomeDoArquivo;
8      }
9
10     @Override
11     public void printNomeDoArquivo() {
12         System.out.println(this.nomeDoArquivo);
13         for (ArquivoComponent arquivoTmp : arquivos) {
14             arquivoTmp.printNomeDoArquivo();
15         }
16     }
17
18     @Override
19     public void adicionar(ArquivoComponent novoArquivo) {
20         this.arquivos.add(novoArquivo);
21     }
22
23     @Override
24     public void remover(String nomeDoArquivo) throws Exception {
25         for (ArquivoComponent arquivoTmp : arquivos) {
26             if (arquivoTmp.getNomeDoArquivo() == nomeDoArquivo) {
27                 this.arquivos.remove(arquivoTmp);
28                 return;
29             }
30         }
31         throw new Exception("Nao existe este arquivo");
32     }
```

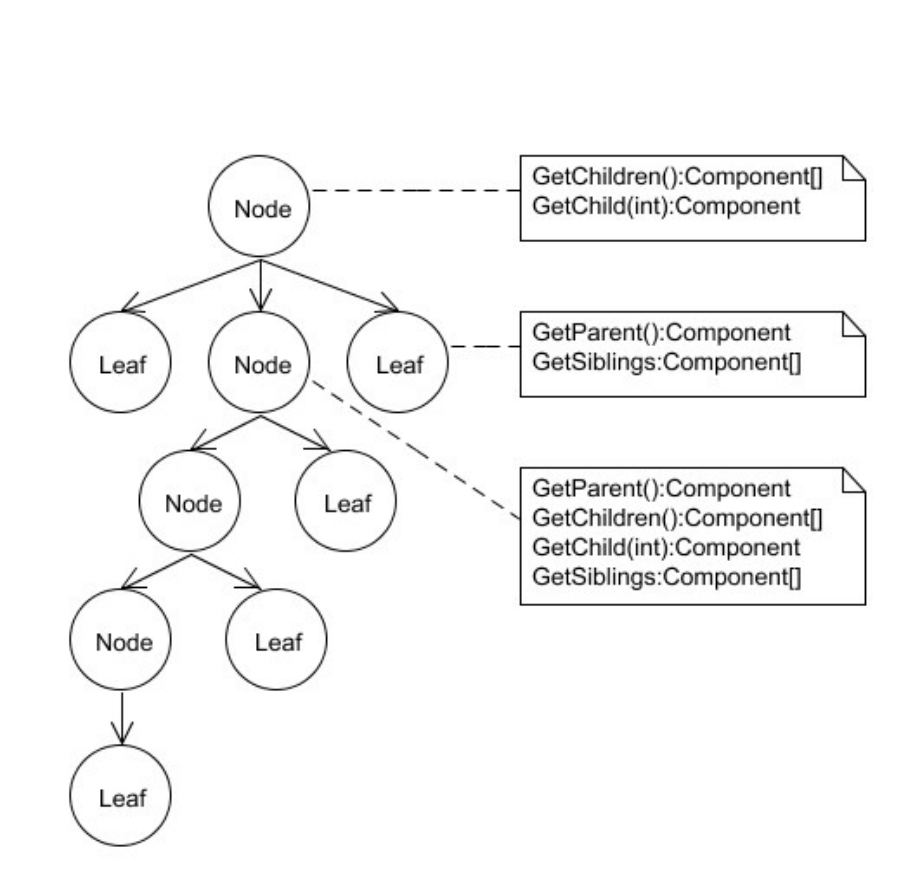
```
1
2       @Override
3       public ArquivoComponent getArquivo(String nomeDoArquivo) throws Exception {
4           for (ArquivoComponent arquivoTmp : arquivos) {
5               if (arquivoTmp.getNomeDoArquivo() == nomeDoArquivo) {
6                   return arquivoTmp;
7               }
8           }
9           throw new Exception("Nao existe este arquivo");
10      }
11
12  }
```

Com isto não é necessário conhecer a implementação dos objetos concretos, muito menos fazer cast. Veja como poderíamos utilizar o código do Composite:

```
1
2   public static void main(String[] args) {
3       ArquivoComponent minhaPasta = new ArquivoComposite("Minha Pasta/");
4       ArquivoComponent meuVideo = new ArquivoVideo("meu video.avi");
5       ArquivoComponent meuOutroVideo = new ArquivoVideo("serieS01E01.mkv");
6
7       try {
8           meuVideo.adicionar(meuOutroVideo);
9       } catch (Exception e) {
10          System.out.println(e.getMessage());
11      }
12
13      try {
14          minhaPasta.adicionar(meuVideo);
15          minhaPasta.adicionar(meuOutroVideo);
16          minhaPasta.printNomeDoArquivo();
17      } catch (Exception e) {
18          System.out.println(e.getMessage());
19      }
20
21      try {
22          System.out.println("\nPesquisando arquivos:");
23          minhaPasta.getArquivo(meuVideo.getNomeDoArquivo())
24                  .printNomeDoArquivo();
25          System.out.println("\nRemover arquivos");
26          minhaPasta.remover(meuVideo.getNomeDoArquivo());
27          minhaPasta.printNomeDoArquivo();
28      } catch (Exception e) {
29          e.printStackTrace();
30      }
31  }
```

Composite

Agora podemos visualizar a tal estrutura de árvore, suponha que temos pastas dentro de pastas com arquivos, a estrutura seria parecida com a de uma árvore, veja a seguir:



Composite

Como uma estrutura de árvore temoas Nós e Folhas. No padrão Composite os arquivos concretos do nosso exemplo são chamados de Folhas, pois não possuem filhos e os arquivos pasta são chamados de Nós, pois possuem filhos e fornecem operações sobre esses filhos.

Conclusão

Conclusão

A primeira vantagem, e talvez a mais forte, seja o fato de os clientes do código Composite serem bem simplificados, pois podem tratar todos os objetos da mesma maneira. No nosso exemplo utilizei exceções para que ficasse mais evidente quando um método de uma Pasta é chamado em um Arquivo, mas suponha que os métodos não fizessem nada, a utilização seria mais simplificada ainda, pois não precisaríamos de blocos try e catch.

Conclusão

No entanto, o mal tratamento destas exceções podem gerar problemas de segurança e ai surge uma outra forma de implementar o padrão, restringindo a interface comum dos objetos. Para isto basta remover os métodos de gerenciamento de arquivos (adicionar, remover, etc) da classe base, assim apenas os arquivos pastas teriam estes métodos.

Conclusão

Em contrapartida o usuário do código precisa ter certeza se um dado objeto é Pasta para realizar um cast e chamar os métodos da pasta.

Referências

[1] GAMMA, Erich et al. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.

Mão na massa: Composite [https://brizeno.wordpress.com/category/padroes-de-projeto/composite/]