Persistência – Parte 1

Leitura e Escrita de Fluxos de Dados em Arquivos
Estudo de Caso: Design Patterns e Persistência

Prof. Dirson S. de Campos

dirson_campos@ufg.br

Material elaborado em parceria com os professores **Nádia F. F. da Silva, Juliana P. Félix, Guilherme S. Marques** e **Reinaldo de S. Júnior**.

11/12/2023

INF
INSTITUTO DE INFORMÁTICA

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

Sumário

1. **Arquivos**
   - Leitura e Escrita de Dados
2. **Estudo de Caso com padrões**
   - *Design Patterns Composite e Persistência de Arquivos*

INF
INSTITUTO DE INFORMÁTICA

Arquivos

Arquivos

Muitas vezes as aplicações necessitam buscar informações de uma **fonte externa** ou enviar dados para um **destino externo**, que não fique armazenado apenas em memória RAM (como é o caso das variáveis que criamos, que vivem no ciclo de execução do problema).

Assim como todo o resto das bibliotecas em Java, a parte de controle de entrada (*input*) e saída (*output*) de dados, conhecida como `I/O`, é orientada a objetos e usa os principais conceitos mostrados até agora, como o polimorfismo.

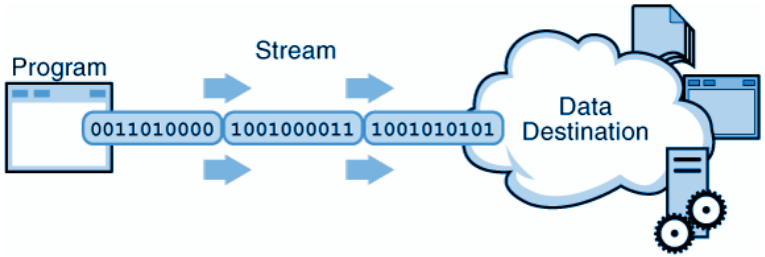
As classes que veremos a seguir tentam padronizar o os meios para se ler e escrever dados, mesmo que oriundos de diferentes origens e formatação.

Arquivos

Assim como em outras linguagens, para o Java, todo arquivo é um *fluxo sequencial de bytes*. A interação de um programa com um dispositivo através de arquivos passa por três etapas:

- Criação e/ou "abertura" de um arquivo;
- Transferência de dados (entrada e/ou saída);
- "Fechamento" do arquivo;

O Java trabalha com várias classes em conjunto para facilitar a manipulação de arquivos. Uma maneira de se usar todas facilmente é importando `java.io.*`.



Arquivos

A classe `File` representa sistemas de arquivos com um alto nível de abstração.

Quando declaramos um objeto seu, não quer dizer obrigatoriamente que exista um arquivo, mas que se houver, ele pode ser "representado" por este objeto na memória durante a execução do seu programa.

```
File arquivo = new File("arquivo.txt");
```

Além disso, o conceito de `File` aqui é mais amplo... como em muitos *SO*s, pode valer tanto para arquivos como para diretórios (pastas).

```
File pastaDownloads = new File("c:\\Users\fulano\Downloads")
```

```
File pastaDownloadsNoLinux = new File("/home/fulano/Downloads");
```

Observação**: A maioria dos tratamentos de arquivos requer um bloco de exceção (`try`/`catch`). **Nos exemplos deste slide omitiremos eles por brevidade, mas demonstraremos seu uso no código.

Arquivos

Esta classe contém métodos para testar a existência de arquivos, para definir permissões, para apagar arquivos, criar diretórios, listar o conteúdo de diretórios, *etc*.

```
public class File {
  public String getParent();                    // Retorna o diretório (objeto File) pai
  public String[] list();                       // Retorna lista de arquivos no
diretório
  public boolean exists();   // Retorna se o arquivo passado para o construtor existe
  public boolean isFile();                      // Retorna se é um arquivo
  public boolean isDirectory();                 // Retorna se é um diretório
  public boolean delete();                      // Tenta apagar o diretório ou
arquivo
  public long length();                         // Retorna o tamanho do arquivo em
bytes
  public boolean mkdir();                       // Cria um diretório com o nome do
arquivo
  public boolean createNewFile();     // Cria um arquivo com o nome do arquivo
  public String getAbsolutePath();    // Retorna o caminho absoluto (path)
  public String getPath();                      // Retorna o caminho relativo
    ...
```

Arquivos

A seguir está um exemplo de acesso de uma pasta e criação de algumas pastas e arquivos.

```
import java.io.*;
public class Arquivos {
          public static void exemplo1() {
          File pasta = new File("nova_pasta");

          if ( !pasta.exists() || !pasta.isDirectory() ) {
               pasta.mkdir(); // Aqui cria-se de fato uma pasta.
               System.out.println("Não encontrei esta pasta então criei uma em "
               + pasta.getAbsolutePath() );
          }
          File outraPasta = new File(pasta, "outra_pasta");
          outraPasta.mkdir(); // Aqui cria-se de fato outra pasta.
          File umArquivo = new File(pasta, "meu_arquivo.txt");
          umArquivo.createNewFile(); // Para criar um arquivo, precisaremos tratar um erro...

          String[] conteúdoDaPasta = pasta.list();
          for (String coisas: conteúdoDaPasta)
                    System.out.println(coisas); // Listará o meu_arquivo.txt e a
     outra_pasta
     }
}
```

Como não indicamos um caminho absoluto, ele procurou o caminho relativo à pasta onde está o código fonte. Veja o caminho exibido em `getAbsolutePath()` para entender onde o programa esperava encontrar esta pasta.

Note que o construtor de `File` pode aceitar um primeiro parâmetro: o "pai" do arquivo na hierarquia do sistema de arquivos, neste caso a pasta acima.

Leitura e Escrita de Dados

Leitura e Escrita de Dados

A leitura e escrita de dados pode ser classificada por diferentes dimensões: **pelo acesso** (*sequencial* × *aleatório*), **pelo tipo de dado** (*bytes* × *caracteres),* **pela quantidade** de informação lida (*linha por linha* × *palavra por palavra*)...

As APIs Java para `I/O` oferecem objetos que abstraem fontes e destinos (nós), fluxos de bytes e caracteres e permitem que a leitura e gravação sejam feitas de diversas formas.

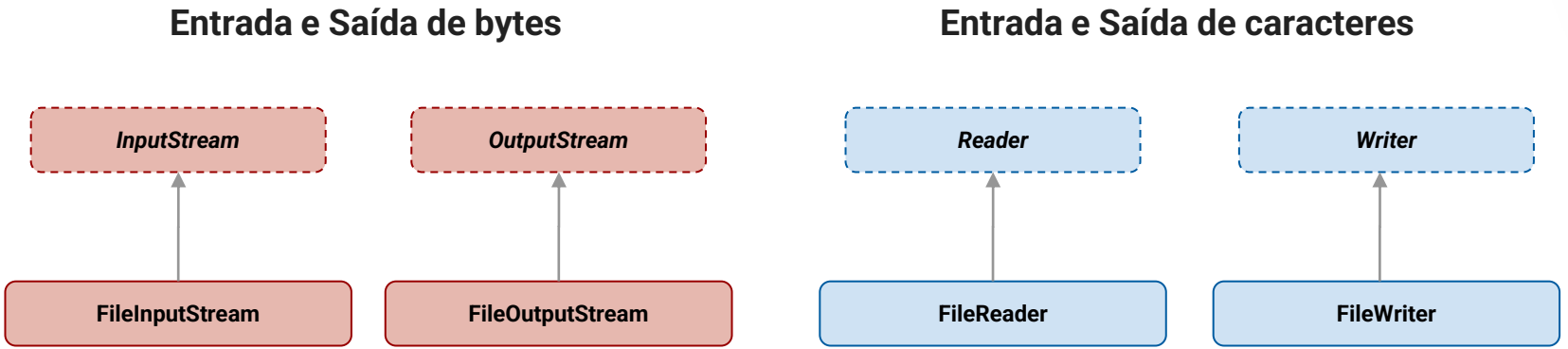
São ao total mais de 40 classes, divididas em:

- Fluxos de entrada (*input streams*): fazem a leitura direta dos bytes;
- Fluxos de saída (*output streams*): fazem a escrita direta dos bytes;
- Leitores (*readers*): oferecem uma interface mais apropriada para leitura;
- Escritores (*writers*): oferecem uma interface mais apropriada para escrita;
- Arquivo de acesso aleatório (*random access file*).

Leitura e Escrita de Dados

As classes podem indicar a mídia de `I/O` ou a forma de manipulação dos dados e podem (devem) ser combinadas para atingirmos o resultado desejado.

Os arquivos são abertos criando-se objetos das chamadas *classes de fluxo* que herdam de `InputStream`, `OutputStream`, `Reader` e `Writer`.



Leitura e Escrita de Dados

Leitura de Bytes: Classe `InputStream`

Métodos disponíveis nas classes Abstratas:

`available()`, `close()`, `read()`, `reset()`, `skip(long l)`, *etc...*

Esperam como parâmetro em seu construtor uma `String` que será usada internamente para criar um objeto `File`.

Classes concretas que herdam de InputStream:

- `FileInputStream`
- `ObjectInputStream`
- `AudioInputStream`
- *dentre outras....*

Leitura e Escrita de Dados

Leitura de Caracteres: Classe `Reader`

Métodos disponíveis nas classes Abstratas:

`available()`, `close()`, `read()`, `reset()`, `skip(long l)`, *etc...*

Recebe um `InputStream` como argumento para seu construtor. Manipula caracteres e possui várias subclasses como a `InputStreamReader` - responsável pela tradução dos *bytes* com o *enconding* dado para código "unicode".

```
public class Arquivos {
        public static void exemplo2() {
        InputStream fluxo = new FileInputStream("arquivo.txt");
        InputStreamReader leitor = new InputStreamReader(fluxo);
        int c = leitor.read();
    }
}
```

O quê isto vai retornar?

Poderia ser um objeto do tipo `File`. Além disso, esta linha demanda um `try/catch`!

Leitura e Escrita de Dados

Leitura de Strings: Classe `BufferedReader`

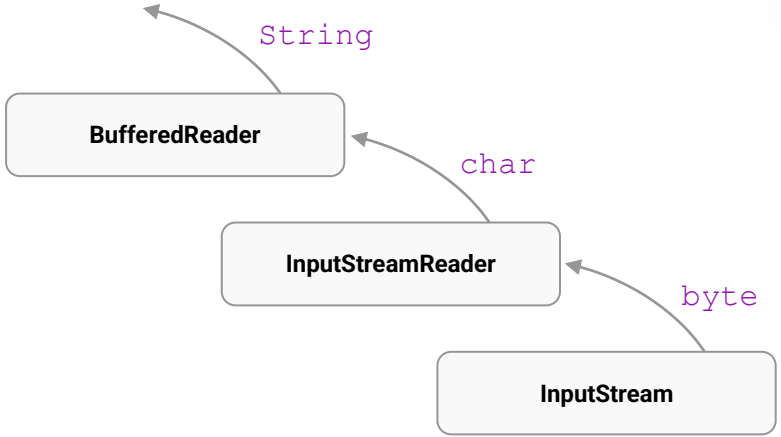
Este é um `Reader` especial que recebe outro `Reader` pelo construtor e concatena os diversos `char`s para formar uma `String` através do método `readLine()`.

```
public class Arquivos {
          public static void exemplo3() {
          InputStream fluxo = new FileInputStream("arquivo.txt");
          InputStreamReader leitor = new InputStreamReader(fluxo);
          BufferedReader leitorComBuffer = new BufferedReader(leitor);
          String linha = leitorComBuffer.readLine();
          while (linha != null) {
                    System.out.println(linha);
                    linha = leitorComBuffer.readLine();
          }
          leitorComBuffer.close();
     }
}
```

Leitura e Escrita de Dados

Leitura de Strings: Classe `BufferedReader`

Esta *composição de classes*, onde um `Reader` lê outro `Reader` por pedaços via *Buffer* evita muitas chamadas no Sistema Operacional. Na verdade, trata-se de um *Padrão de Projetos Estrutural* conhecido como **Decorador** (*Decorator Pattern*).




```
new FileInputStream("arquivo.txt")
```

Leitura e Escrita de Dados

Leitura de Strings: Classe `BufferedReader`

O construtor do `InputStreamReader` também pode ler dados da "entrada padrão" do Java (aquela que passamos para a `Scanner`), ou seja... do teclado:

```
public class Arquivos {
          public static void exemplo4(String args[]) {
          InputStreamReader leitor = new InputStreamReader(System.in);
          BufferedReader leitorComBuffer = new BufferedReader(leitor);
          String linha = leitorComBuffer.readLine();
          while (linha != null) {
                    System.out.println(linha);
                    linha = leitorComBuffer.readLine();
          }
          leitorComBuffer.close();
     }
}
```

Aqui, ao invés de abrir um fluxo de um arquivo, estou acessando o teclado, portanto escutando entradas do usuário.

Leitura e Escrita de Dados

Escrita de Bytes: Classe `OutputStream`

Métodos disponíveis nas classes Abstratas:

`close()`, `flush()`, `write(byte b)`, `write(byte[] b)`, *etc...*

Esperam como parâmetro em seu construtor uma `String` que será usada internamente para criar um objeto `File`.

Classes concretas que herdam de OutputStream:

- `FileOutputStream`
- `ObjectOutputStream`
- *dentre outras....*

Leitura e Escrita de Dados

Escrita de Caracteres: Classe `Writer`

Métodos disponíveis nas classes Abstratas:

`append(char c),close(),flush(),write(char[] c), write(String s),` *etc...*

Classes concretas que herdam de Writer:

- `FileWriter`
- `BufferedWriter`
- `PrintWriter`
- `OutputStreamWriter`
- *dentre outras....*

Leitura e Escrita de Dados

Escrita de Strings: Classe `BufferedWriter`

De maneira análoga ao BufferedReader, aqui usamos uma composição de classes para fazer uma escrita de dados avançada:


```
public class Arquivos {
          public static void exemplo5() {
          OutputStream fluxo = new FileOutputStream("arquivo.txt");
          OutputStreamWriter escritor = new OutputStreamWriter(fluxo);
          BufferedWriter escritorComBuffer = new BufferedWriter(escritor);
          escritorComBuffer.write("Olá, tudo bem?");
          escritorComBuffer.flush();
          escritorComBuffer.close();
     }
}
```

Também poderia ser um objeto do tipo `File`. Além disso, esta linha demanda um `try/catch`!

Já esta linha demandará outro `try/catch`...

Esta linha "descarrega" as mudanças no arquivo, já que pelo uso do *buffer* a escrita não acontece imediatamente ao se chamar o `write`.

Observações Finais

- `FileInputStream` e `FileOutputStream` são adequadas para escrever dados binários (bytes).
- Para manipular mais facilmente texto (Strings) podemos utilizar as classes `FileReader`, `BufferedReader`, `FileWriter` e `PrintWriter`.
- Ao usarmos `BufferedWriter` devemos chamar o método `.flush()` para concretizar as mudanças passadas para o método `.write()`.
- Muitos dos Streams obrigam você a tratar de uma `FileNotFoundException` ao serem construídos e de uma `IOException` ao chamar seus métodos de leitura ou escrita.
- Note que é importante executar o método `close()` ao concluir a manipulação de arquivos. Se for necessário usar `try/catch` para abrir o arquivo, o `close()` é um bom caso para se chamar no bloco `finally`.

Estudos de Casos: Persistência e Padrões

Estudo de Caso 1
Design Patterns
Composite e Persistência de Arquivos

Padrão Estrutural Composite e a Classe Java File

- A classe Java File é um exemplo do **padrão Composite da GoF** para estruturação de Objetos.

- *Composite Pattern* (Objetos Composição) que é um padrão de projeto de software utilizado para representar um objeto formado pela composição de objetos similares. Este conjunto de objetos pressupõe uma mesma hierarquia de classes a que ele pertence.

Padrão Estrutural Composite e a Classe Java File

- O padrão Composite é, normalmente, utilizado para representar listas recorrentes ou recursivas de elementos como uma estrutura hierárquica e uma estrutura em árvore tal como o estrutura de Arquivo do Sistemas Operacional que são manipuladas em Java pela classe File.

Padrão Estrutural Composite e a Classe Java File

- O padrão Composite é, normalmente, utilizado para representar listas recorrentes ou recursivas de elementos como uma estrutura hierárquica e uma estrutura em árvore tal como o estrutura de Arquivo do Sistemas Operacional que são manipuladas em Java pela classe File.

Padrão: Composite Object

❑ Objetivo Principal

- Permite que uma hierarquia de objetos seja tratada como um objeto só.

Padrão: Composite Object

❑ Visão Geral do Padrão

- **Composite Object (Objeto Composto)** é um padrão de projeto que permite que um objeto seja constituído de outros objetos semelhantes a ele, formando uma hierarquia. Semelhantes, significa aqui, objetos que implementam um contrato comum seja porque implementam uma interface comum ou derivam de uma mesma classe.
- Seguindo este padrão podemos construir um objeto que seja construído de outros objetos tal que, um ou mais objetos desses podem ser do mesmo tipo do objeto construído.
- O objeto é constituído por uma coleção de outros objetos semelhantes e assim sucessivamente

Padrão: Composite Object

❑ Visão Geral do Padrão

- Este padrão pode ser usado sempre que desejarmos construir objetos pela composição recursiva de objetos do mesmo tipo e/ou queiramos construir uma hierarquia de objetos do mesmo tipo.
  - Este é o padrão por detrás da estrutura de árvore, muito utilizada em computação.

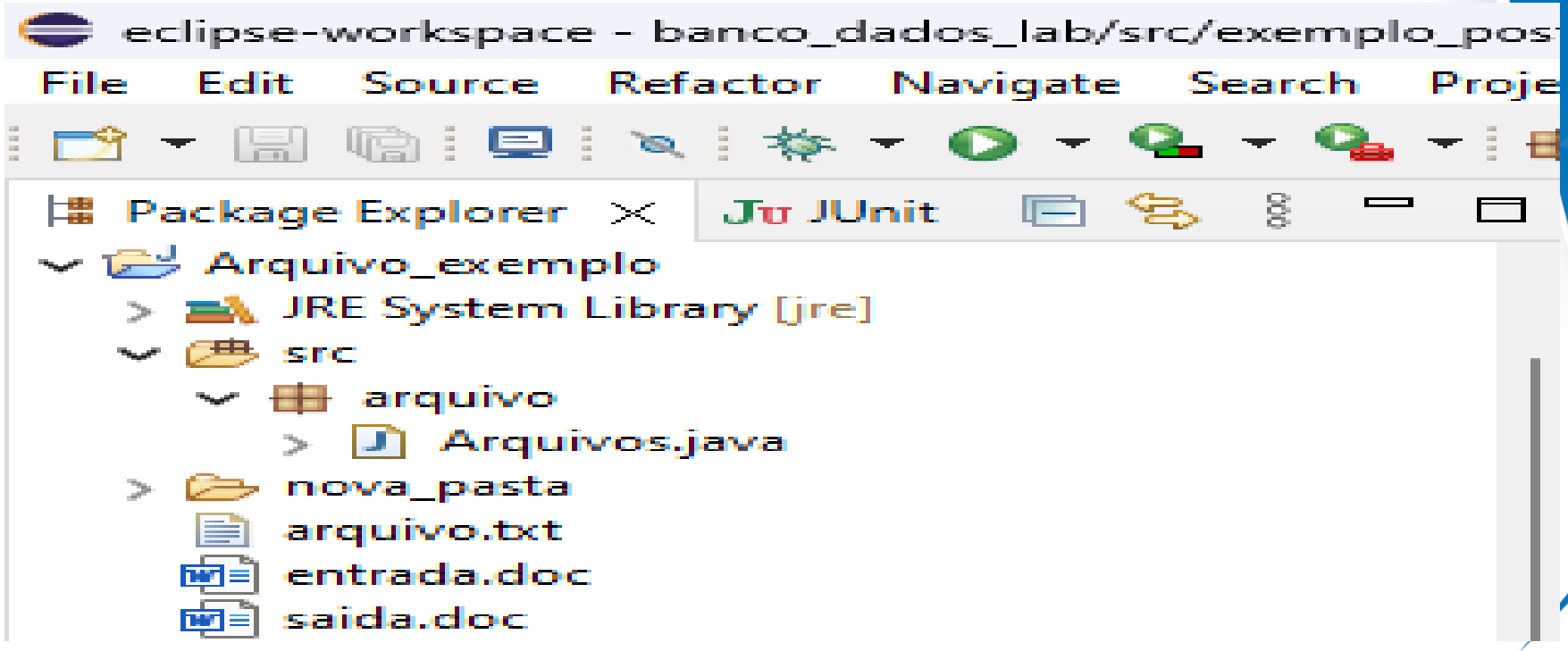
Padrão: Composite Object

❑ Visão Geral do Padrão

- A Estrutura de Árvore tem como primitivas as folhas.
- Os ramos são os objetos compostos.
- A árvore é uma composição de ramos, por sua vez compostos de ramos menores, por sua vez compostos de ramos menores e folhas.
- Sempre que estiver perante uma estrutura deste tipo, ou semelhante, poderá usar o padrão **Composite Object**

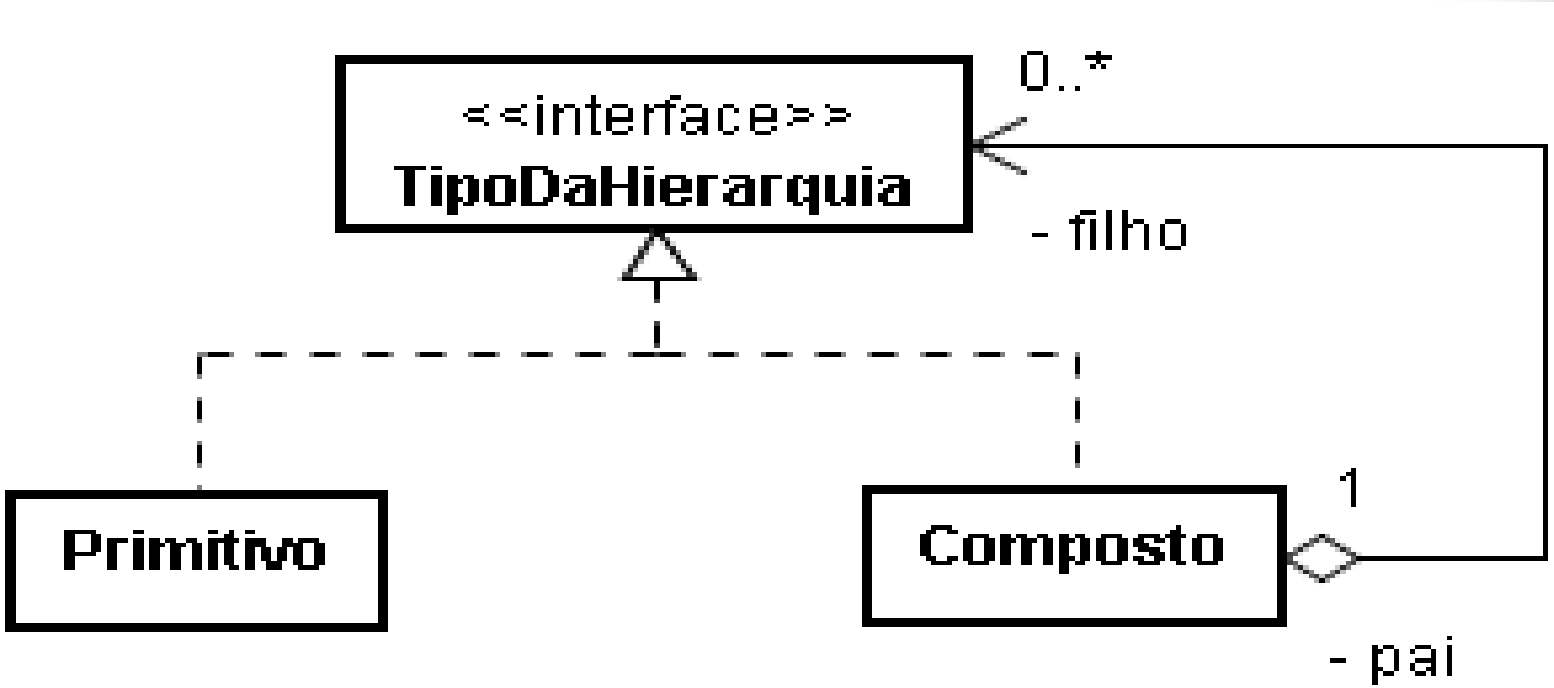
Padrão: Composite Object

- Exemplo: IDE – Eclipse – Package Explorer



Padrão: Composite Object

❑ Diagrama de Classe



Padrão: Composite Object

❑ Diagrama de Classes

- O padrão propõe que se construa uma interface, ou classe abstrata, que representa o tipo de objeto na hierarquia.
- Para que o padrão possa ser aplicado tem que existir esta interface ou classe no domínio do problema, ou seja, esta interface ou classe tem que representar um conceito presente no modelo.
- Objetos que implementam esta interface podem ser de dois tipos:
  - Primitivos ou
  - Compostos.

Padrão: Composite Object

❑ Self-Composite (auto-composto)

- Em algumas situações especiais não existe um objeto primitivo na hierarquia ou a sua interface não se distingue da interface do objeto composto.
- Neste caso o objeto que representa o tipo de hierarquia é simultaneamente o objeto composto e o objeto primitivo.
- O objeto composto é constituído de objetos iguais. O objeto é **self-composite (auto-composto).**

Padrão: Composite Object

❑ Self-Composite (auto-composto)

- Na API padrão do Java encontramos o objeto **java.io.File**.
- O objeto File implementa o padrão self-composite. O objeto representa simultaneamente um arquivo e uma pasta de arquivos num sistema de arquivos (que tem uma estrutura em árvore).
- Neste caso é utilizada a mesma interface para tratar ambos os tipos de objeto minimizando o número de classes necessárias para trabalhar com o sistema de arquivos.

Padrão: Composite Object

❑ Self-Composite (auto-composto)

- É interessante notar que ao fundir o elemento primitivo com o composto numa só interface, mas maiorias das vezes, somos obrigados a implementar métodos que nos informam se o elemento é de um , ou de outro tipo.
  - Por exemplo, no caso de File os métodos **isFile()** e **isDirectory()** informam se o objeto corresponde a um arquivo ou a uma pasta.

Padrão: Composite Object

- **Exemplo de API (Application Programming Interface, ou Interface de Programação de Aplicações) em Java**
  - Na API padrão, Composite Object, é utilizado nas classes **java.io.File**, **javax.swing.JComponent e javax.swing.tree.TreeNode**, por exemplo.
  - Na classe **java.io.File** acontece a composição de objetos da mesma classe sendo um exemplo do padrão self-composite para modelar a estrutura de arvores do sistema de arquivos.
  - Na classe **javax.swing.JComponent** acontece a composição de objetos de classes derivadas. Esta composição é depois traduzida visualmente o que torna esta classe um exemplo do padrão Composite View.
  - Na classe **javax.swing.tree.TreeNode** acontece a composição de objetos da mesma classe ou classes derivadas, sendo a representação direta de uma estrutura de dados em arvore utilizada pelo componente **javax.swing.JTree**.