19 Structural: Composite Pattern for a File-System Hierarchy

**Scenario** You are building a command-line tool that must display, add, and remove items in a virtual file system.  
 Requirements:

* Treat **files** and **directories** with one uniform API.
* Allow directories to hold an arbitrary number of children (files or sub-directories).
* Let clients build, traverse, and modify the hierarchy without casting or instanceof checks.

Your task is to apply the **Composite pattern** so that both leaf (File) and composite (Directory) objects share a single interface, letting the client manipulate the structure transparently.

#### **Tasks**

1. **Problem analysis** (analysis/composite\_need.md)  
   * Explain why modelling files and directories with separate, unrelated classes forces the client to branch on type.
   * Show how Composite removes that branching and supports nested trees of unlimited depth.
2. **Code implementation** (src/main/java/vfs/)  
   * FileSystemComponent – the **Component** interface with:  
     + void showDetails(int indent)
     + default helper void showDetails() that calls the indented version with 0.
   * File – the **Leaf** (stores name).
   * Directory – the **Composite** (stores List<FileSystemComponent> plus add, remove, children).
   * Override showDetails(indent) so that directories indent children two spaces per level.
3. **Client demo** (src/main/java/vfs/CompositeDemo.java)

Build the example tree:  
  
 /root

/documents

resume.pdf

/media

holiday.jpg

/music

song.mp3

* + Call root.showDetails()­—output must match the structure with indentation.

1. **Unit tests** (src/test/java/vfs/)  
   * UniformTreatmentTest – ensure a list of mixed File and Directory can be iterated and showDetails() called polymorphically without ClassCastException.
   * AddRemoveChildTest – create a directory, add two files, remove one, assert only the remaining child is printed.
   * RecursiveDepthTest – build a 4-level deep directory and assert four indentation levels appear in the output.
2. **Reflection** (reflection.md)  
   * What benefit does Composite give the client and how does it promote the Open/Closed Principle?
   * Identify a limitation: a directory can contain itself via a careless call; propose a guard to prevent cycles.
   * How would you extend the model to support file sizes and directory size aggregation?

#### **Deliverables**

analysis/composite\_need.md

src/main/java/vfs/FileSystemComponent.java

src/main/java/vfs/File.java

src/main/java/vfs/Directory.java

src/main/java/vfs/CompositeDemo.java

src/test/java/vfs/UniformTreatmentTest.java

src/test/java/vfs/AddRemoveChildTest.java

src/test/java/vfs/RecursiveDepthTest.java

reflection.md

README.md

## **Detailed Solution**

### **1 Component & Leaf**

/\* FileSystemComponent.java \*/

package vfs;

/\*\* Component – common interface for files and directories. \*/

public interface FileSystemComponent {

void showDetails(int indent); // core

default void showDetails() { showDetails(0); } // convenience

}

/\* File.java \*/

package vfs;

public class File implements FileSystemComponent {

private final String name;

public File(String name){ this.name = name; }

@Override public void showDetails(int indent){

System.out.println(" ".repeat(indent) + "File: " + name);

}

}

### **2 Composite**

/\* Directory.java \*/

package vfs;

import java.util.\*;

public class Directory implements FileSystemComponent {

private final String name;

private final List<FileSystemComponent> children = new ArrayList<>();

public Directory(String name){ this.name = name; }

public void add(FileSystemComponent c){ children.add(c); }

public void remove(FileSystemComponent c){ children.remove(c); }

public List<FileSystemComponent> children(){ return List.copyOf(children); }

@Override public void showDetails(int indent){

System.out.println(" ".repeat(indent) + "Directory: " + name);

for(FileSystemComponent c : children){

c.showDetails(indent + 2);

}

}

}

### **3 Demo**

/\* CompositeDemo.java \*/

package vfs;

public class CompositeDemo {

public static void main(String[] args){

Directory root = new Directory("root");

Directory docs = new Directory("documents");

Directory media = new Directory("media");

Directory music = new Directory("music");

docs.add(new File("resume.pdf"));

media.add(new File("holiday.jpg"));

music.add(new File("song.mp3"));

media.add(music);

root.add(docs);

root.add(media);

root.showDetails();

}

}

**Console**

Directory: root

Directory: documents

File: resume.pdf

Directory: media

File: holiday.jpg

Directory: music

File: song.mp3

### **4 Test Highlights (JUnit 5)**

/\* UniformTreatmentTest \*/

@Test void uniformTreatment(){

List<FileSystemComponent> list = List.of(

new File("a.txt"),

new Directory("empty"));

assertDoesNotThrow(() -> list.forEach(FileSystemComponent::showDetails));

}

/\* AddRemoveChildTest \*/

@Test void addRemove(){

Directory dir = new Directory("d");

File a = new File("a.txt"), b = new File("b.txt");

dir.add(a); dir.add(b); dir.remove(a);

String out = capture(dir::showDetails);

assertTrue(out.contains("b.txt"));

assertFalse(out.contains("a.txt"));

}

*(Helper capture(Runnable) redirects System.out to collect output.)*

### **Reflection (extract)**

• Uniform API eliminates client branching → easier recursion, Open/Closed.

• Risk: cycles or invalid combinations; add guard in Directory.add() to forbid

adding itself or ancestors.

• For sizes, make File store long size; Directory.size() sums child sizes.

The Composite pattern neatly models the file-system tree, enabling flexible, nested structures while letting clients operate with one consistent set of operations.