Composite Design Pattern

Composite Design Pattern is one of the most powerful and elegant patterns in the Structural category. Let's break it down step by step — from real-world analogy to Java code, JDK internal examples, and Spring Boot applications.

🧠 What is Composite Design Pattern?

> **Composite Pattern** is used when we want to treat a group of **objects and individual objects uniformly**.

It forms a **tree structure** where each node is either:

- a **leaf** (cannot contain anything), or
- a **composite** (can contain children including other composites).

🛒 Real-Life Analogy: A File System or Company Org Chart

- You can have **Files (leaves)** and **Folders (composites)**.
- A **folder can contain** both files and other folders.

You want to:

- Call `show()` on a File → shows name
- Call `show()` on a Folder → recursively shows all contents

✅ Both folders and files should share the **same interface**.

✅ Java Implementation – Composite Pattern

Step 1: Common Interface (Component)

```
interface FileSystemComponent {
    void show();
}
```

Step 2: Leaf Node – File

```
class File implements FileSystemComponent {
    private String name;

    public File(String name) {
        this.name = name;
    }
    public void show() {
        System.out.println("File: " + name);
    }
}
```

Step 3: Composite Node – Folder

```
class Folder implements FileSystemComponent {
    private String name;
    private List<FileSystemComponent> children = new ArrayList<>();

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

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

    public void show() {
        System.out.println("Folder: " + name);
        for (FileSystemComponent child : children) {
            child.show();  // recursive call
        }
    }
}
```

Step 4: Client Code

```
public class CompositeDemo {
    public static void main(String[] args) {
        File file1 = new File("resume.pdf");
        File file2 = new File("budget.xlsx");
        Folder docs = new Folder("Documents");

        docs.add(file1);
        docs.add(file2);

        Folder root = new Folder("Root");
        root.add(docs);
        root.add(new File("readme.txt"));

        root.show();
    }
}
```

✅ Output:

```
Folder: Root
Folder: Documents
File: resume.pdf
File: budget.xlsx
File: readme.txt
```

🧬 Key Concepts

Term	Role in Composite Pattern
Component	Common interface (`FileSystemComponent`)
Leaf	Individual object (`File`)
Composite	Group/container (`Folder`)
Client	Code that works with both (`root.show()`)

🧰 Real JDK Examples of Composite Pattern

✅ A. `java.awt.Component` and `Container`

```
Component button = new JButton("Click");
Container panel = new JPanel();
panel.add(button);
```

- `Component` is the **interface**
- `JButton` is a **leaf**
- `JPanel` is a **composite** — it can contain components including other `JPanel`s

✅ B. `org.w3c.dom.Node`

```
Node element = document.getElementById("div1");
NodeList children = element.getChildNodes();
```

- `Node` is a base type
- Each node can be a **leaf** (Text) or **composite** (Element containing children)

🌱 Spring Boot Example: Spring's `ApplicationContext`

✅ Spring Beans as Composite Hierarchy

- Spring's `ApplicationContext` interface **extends** `BeanFactory`
- It can **contain other contexts** (like parent context → child context)
- You can walk through the tree via:

```
ApplicationContext parent = new
AnnotationConfigApplicationContext(ConfigA.class);
ApplicationContext child = new AnnotationConfigApplicationContext();
((GenericApplicationContext) child).setParent(parent);
```

Now:

```
child.getBean("myBean"); // searches in parent if not found
```

✅ Each context is a **composite** — containing beans and even other contexts.

✅ Another Example – Spring Security `AccessDecisionVoter`

```
AccessDecisionManager adm = new AffirmativeBased(Arrays.asList(
    new RoleVoter(), new AuthenticatedVoter()
));
```

- Each `Voter` is a **leaf**
- `AccessDecisionManager` is a **composite** that loops through all voters
- All voters implement the same interface

✅ Interview Answer

> "The Composite Pattern lets us treat **individual objects and groups of objects uniformly**. It's useful when dealing with **tree-like structures** like file systems, DOM trees, or GUI components. In Java, `java.awt.Component`, `org.w3c.dom.Node`, and Spring's `ApplicationContext` use it to let composite objects behave just like leaves, allowing recursive traversal and manipulation."

UML diagram for the **Composite Design Pattern**, based on our FileSystem (Files & Folders) example.

📊 UML Flowchart of Composite Pattern

```
pgsql
CopyEdit
           +----------------------------+
           | FileSystemComponent (interface) |
           |----------------------------|
           | + show()                   |
           +----------------------------+
                   ▲
                   |
      -------------------------------
      |                             |
+---------------------+   +------------------------+
|       File          |   |        Folder          |
|---------------------|   |------------------------|
| - name              |   | - name                 |
|                     |   | - List<FileSystemComponent> children |
| + show()            |   | + add(child)           |
+---------------------+   | + show()               |
                          +------------------------+
```

✅ Explanation of Diagram

- `FileSystemComponent` is the **common interface** – the **Component**.
- `File` is the **Leaf** – cannot contain children.
- `Folder` is the **Composite** – **contains a list of children** (`FileSystemComponent`).

- Client code can call `.show()` on either `File` or `Folder` **without knowing** the type.

🧠 Real-World Flow in Code Terms

```
arduino
CopyEdit
Folder root = new Folder("Root")
  ├── Folder docs
  │     ├── File resume.pdf
  │     └── File budget.xlsx
  └── File readme.txt
```

Calling `root.show()`:

1. Shows `Root`
2. Recursively calls `docs.show()`
3. Recursively calls `resume.pdf.show()`, `budget.xlsx.show()`
4. Then shows `readme.txt`

✅ You are traversing the **entire tree**, treating both `File` and `Folder` **uniformly**.

Would you like the same visual UML version for:

- JDK Composite (`Component`/`Container`)
- Spring Composite (`ApplicationContext`)
- Or move to the next pattern (e.g., **Decorator**)?