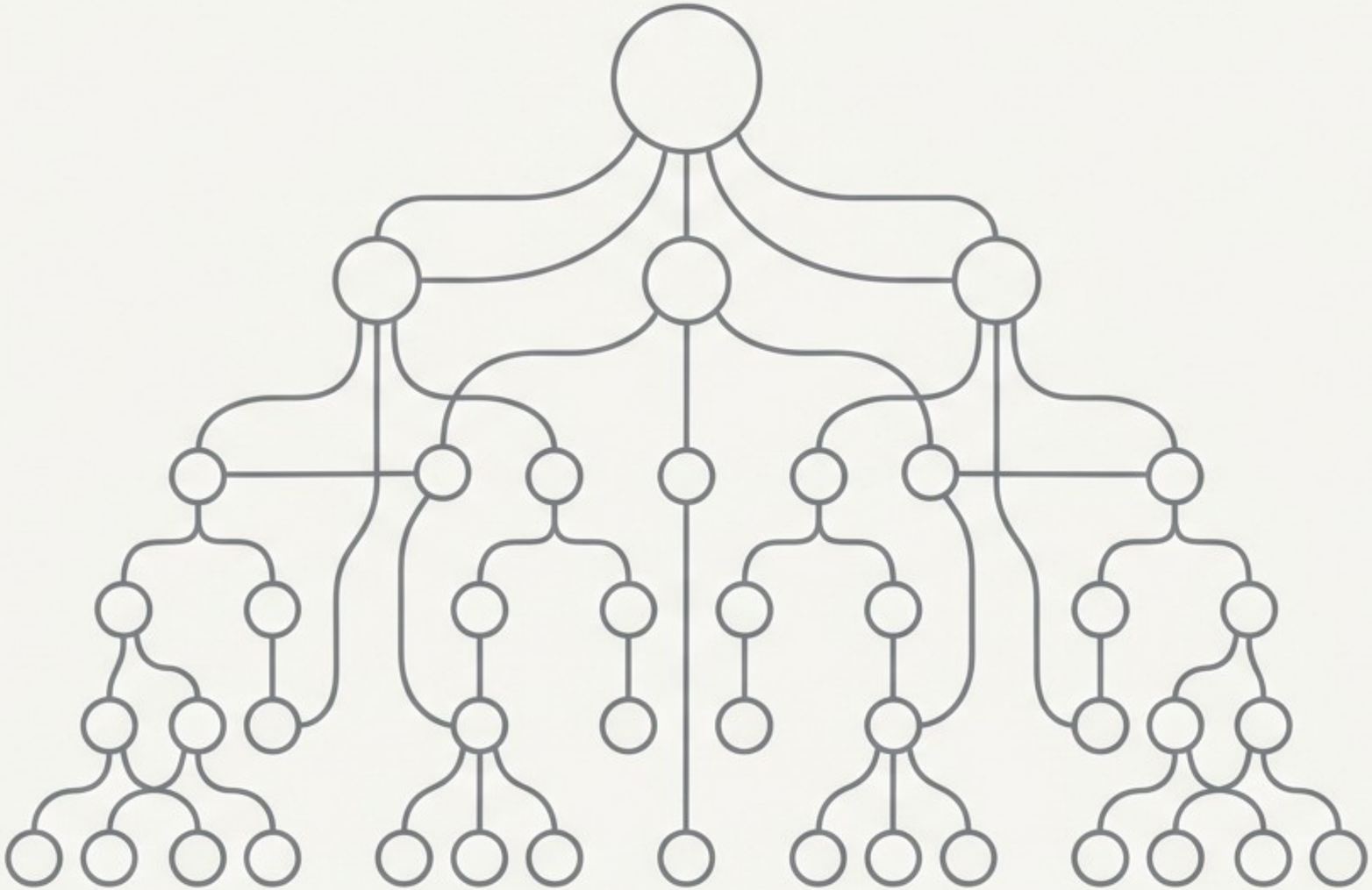
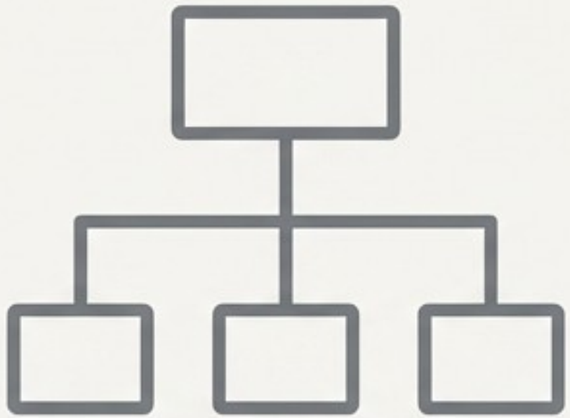
Taming Complexity: The Composite Pattern

How to treat individual objects and groups with uniform elegance.

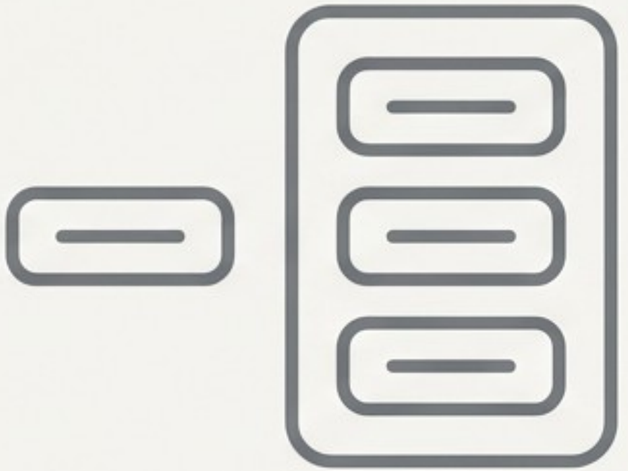


The Common Dilemma: We need to treat individuals and groups in the same way.

Files vs. Folders
in a file system.



Individual Employees vs. Managers
with teams.



Single Buttons vs. Containers
holding multiple elements.

Operating on a single item is different from operating on a collection, leading to tangled logic.

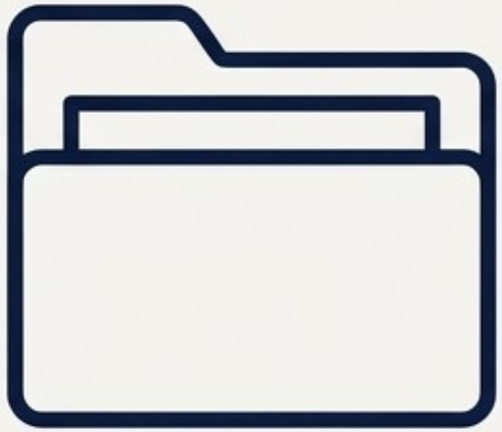
The Brute-Force Approach: A Maze of Conditional Checks.

```
public void Show(Node node)
{
    if (node is File)
        Console.WriteLine("Opening file");
    if (node is Folder)
        foreach (var child in ((Folder)node).Children)
            Show(child);
}
```

- Special case logic everywhere.
- Tightly coupled to concrete types.
- Violates the Open/Closed Principle.
- Hard to maintain and extend.

A Simpler Way to Think: Your Computer's File System

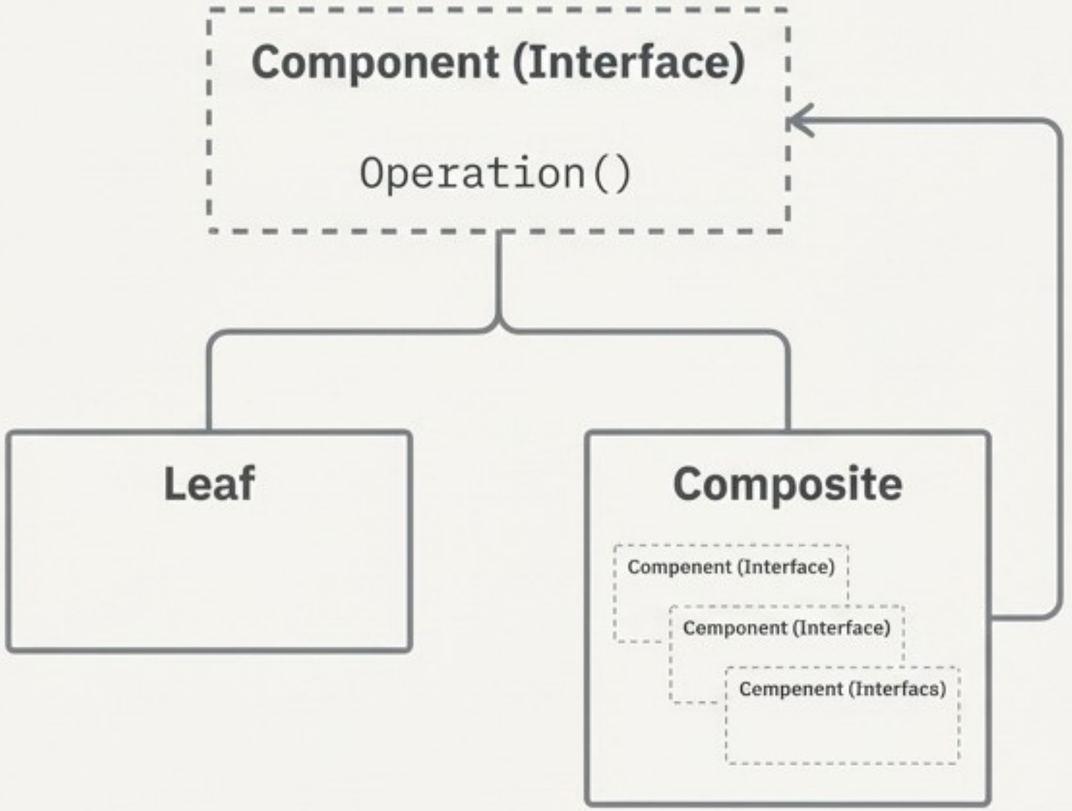
A **File** is a single item.



A **Folder** can contain files AND other folders.

You can `delete`, `move`, or `copy` both using the exact same actions. The system doesn't care if it's a file or a folder—it treats them uniformly. **That is the Composite Pattern**.

The Solution: Unifying the Part and the Whole.



We create a common interface that both single objects (‘Leaves’) and group objects (‘Composites’) implement, allowing them to be treated as one.

The Blueprint: Three Core Roles in the Pattern

The Component

The shared interface for all objects in the composition. It declares the uniform operations (e.g., `Display()`, `GetSize()`).

The Leaf

Represents the individual, end-node objects in the hierarchy. It implements the Component's operations but has no children.

The Composite

Represents the group objects. It stores child components (both Leaves and other Composites) and implements operations by delegating to its children.

Code in Action, Step 1: Define the Common Interface

We start by creating a single interface that all our file system objects will share.

```
public interface IFileSystemItem
{
    void Display();
}
```

This is the uniform operation our client code will use for both files and folders.

Code in Action, Step 2: Implement the ‘Leaf’ Object

The `FileItem` class represents an individual object. It implements the interface directly.

```
public class FileItem : IFileSystemItem
{
    private readonly string _name;

    public FileItem(string name) { _name = name; }

    public void Display()
        => Console.WriteLine($"File: {_name}");
}
```

The Leaf performs the operation on itself. Note that it has no list of children.

Code in Action, Step 3: Implement the 'Composite' Object

The `Folder` class also implements the interface, but it holds a collection of children and delegates the operation to them.

```
public class Folder : IFileSystemItem
{
    private readonly string _name;
    private readonly List<IFileSystemItem> _children = new();

    public Folder(string name) { _name = name; }

    public void Add(IFileSystemItem item) => _children.Add(item);

    public void Display()
    {
        Console.WriteLine($"Folder: {_name}");
        foreach (var item in _children)
            item.Display(); // Delegation
    }
}
```

Manages a collection of other `IFileSystemItem` objects.

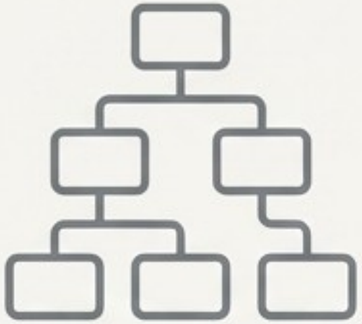
Crucially, it delegates the `Display` call to each child.

The Payoff: Clean, Uniform Client Code

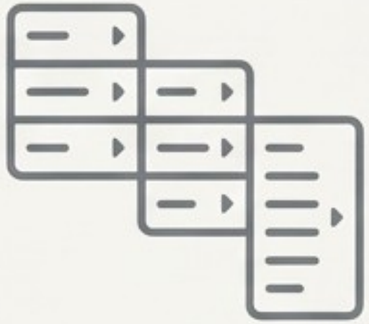
```
var root = new Folder("Root");
root.Add(new FileItem("readme.txt"));

var images = new Folder("Images");
images.Add(new FileItem("logo.png"));
images.Add(new FileItem("banner.jpg"));

root.Add(images);

root.Display(); // One call displays the entire tree
```

- ✓ Uniform operations on every item.
- ✓ No more `if/else` type checking.
- ✓ Clean, hierarchical structure.

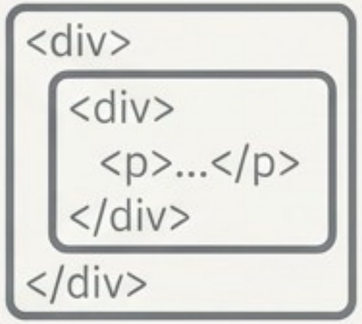
This Pattern is Everywhere in Professional Software.



Organisation Structures

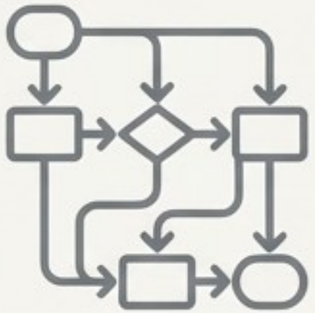


Application Menus

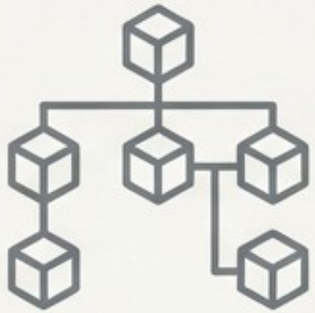


UI DOM Trees

Product Categories



Graphical Workflow Trees



Game Object Hierarchies

When to Reach for the Composite Pattern.

1. You have part-whole hierarchies.

Use this pattern when your model can be represented as a tree structure.

Examples: Files & folders, Categories & subcategories, UI containers & controls.

2. You want to treat objects uniformly.

Use this when you want client code to be ignorant of the difference between a single object and a group of objects.

3. You want to avoid conditional logic.

Use this to eliminate code that constantly checks an object's type before acting on it.

It’s All About Intent: Composite vs. Other Patterns.

Composite

Builds a **tree structure** where leaves and groups share the same interface. The focus is on treating a treating a hierarchy uniformly.

Decorator

Adds **new behaviours** to a **single object** single object dynamically by wrapping it. The focus is on extending functionality.

Iterator

Provides a **standard way** Provides a **standard way** to **traverse** a collection, hiding its internal structure. The focus is on access and traversal.

The intent is different. Composite is about structure, not adding behaviour or simplifying traversal.

Built on a Foundation of Good Design.

Polymorphism

The client works with the `Component` interface, not concrete types. This is what allows for the uniform treatment of different objects.

Open/Closed Principle (OCP)

You can add new kinds of `Leaf` and `Composite` classes to the system without changing the client code that uses them. The system is open to extension but closed for modification.

Single Responsibility Principle (SRP)

Each class has a clear job. `Leaf` handles itself. `Composite` handles its children.

The Composite Pattern in a Nutshell.

Problem	Need to handle single objects and group objects in the same way.
Solution	Build a tree structure where all nodes share a common interface.
Key Idea	A `Leaf` (single item) and a `Composite` (group) share the same operations.
Benefits	Cleaner code, fewer `if/else` checks, easier to extend and maintain.
Common Uses	Menus, org charts, file systems, UI frameworks, product categories.
Principles	Polymorphism, Open/Closed Principle, Single Responsibility.