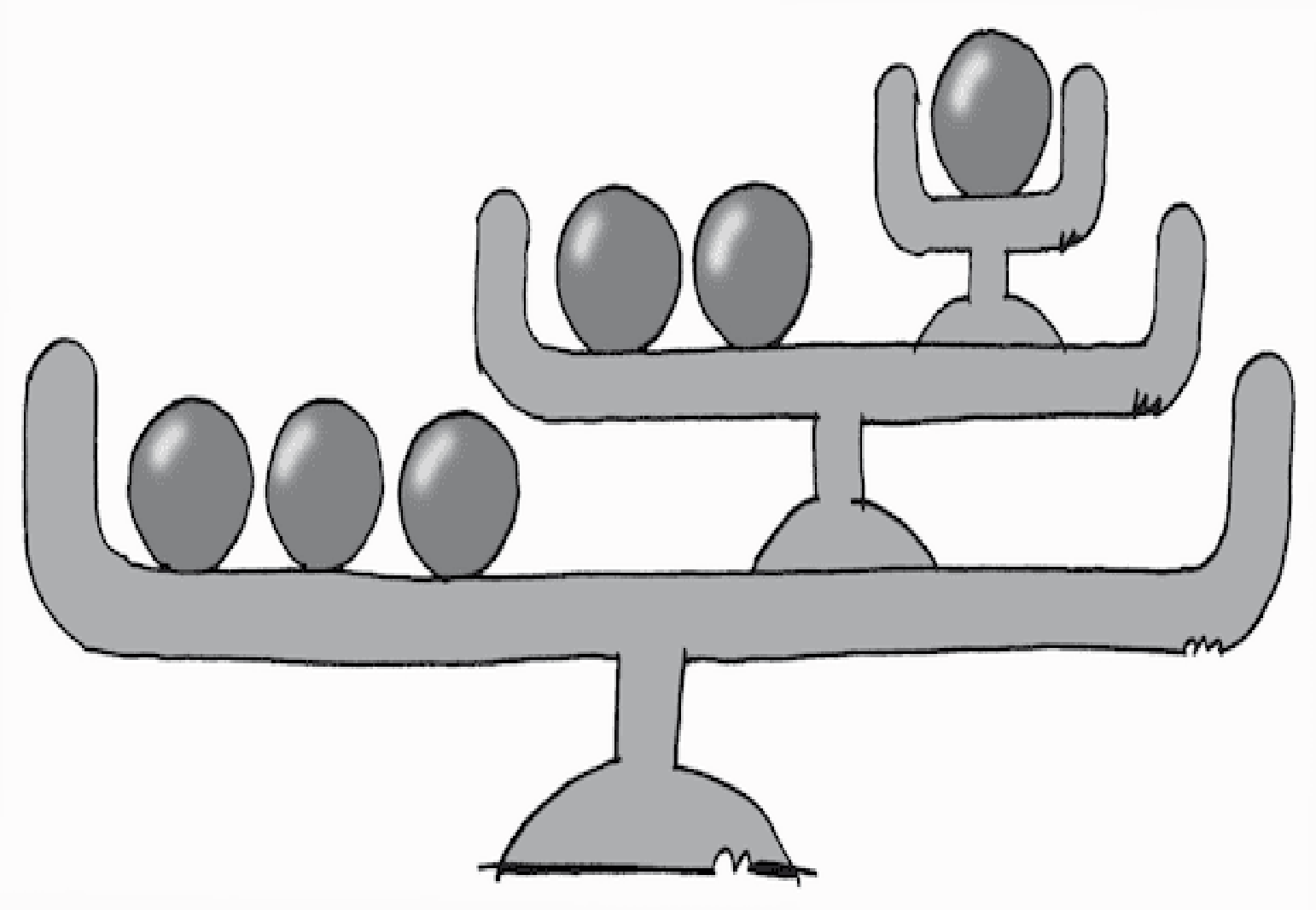
Composite Pattern

Treat Individual Objects and Compositions Uniformly



Composite Pattern

We find some recursive structure everywhere:

- **File system**: Files and folders (folders can contain files/other folders)
- **GUI Components**: Buttons and panels (panels contain buttons/other panels)
- **Arithmetic expressions**: Numbers and operations (operations contain sub-expressions)

The Problem

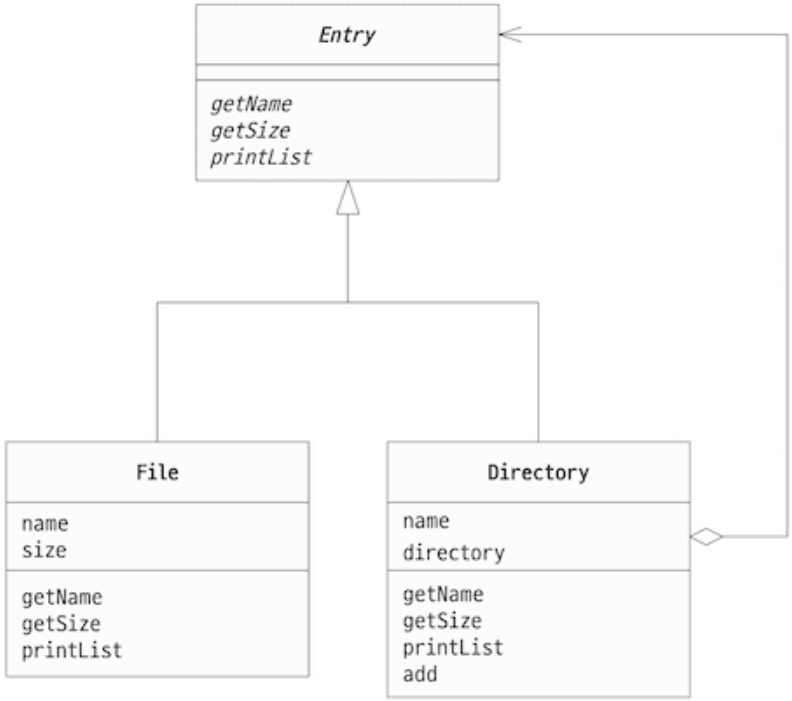
- We have a **file system** with **files** and **directories**.
- **Directories** can contain both files and other **directories** (tree structure).

The challenge: how to treat files and directories uniformly while handling their distinct behaviors?

The *Composite* as the Solution

- We have an abstraction *Component* that defines common operations for both *leaf* and *composite* objects.
- We do not need to `distinguish` between individual and composite objects; we only need to `use` the common *interface*.

The Solution (Design)



- Entry:Component
- File:**Item**:Leaf
- Directory:**Box**:Composite

Step 1: Understand the Players

In this design, we have players:

- **Box** = like a directory (can contain items *or* other boxes)
- **Item** = like a single file (cannot contain others)

The user who works with items and boxes

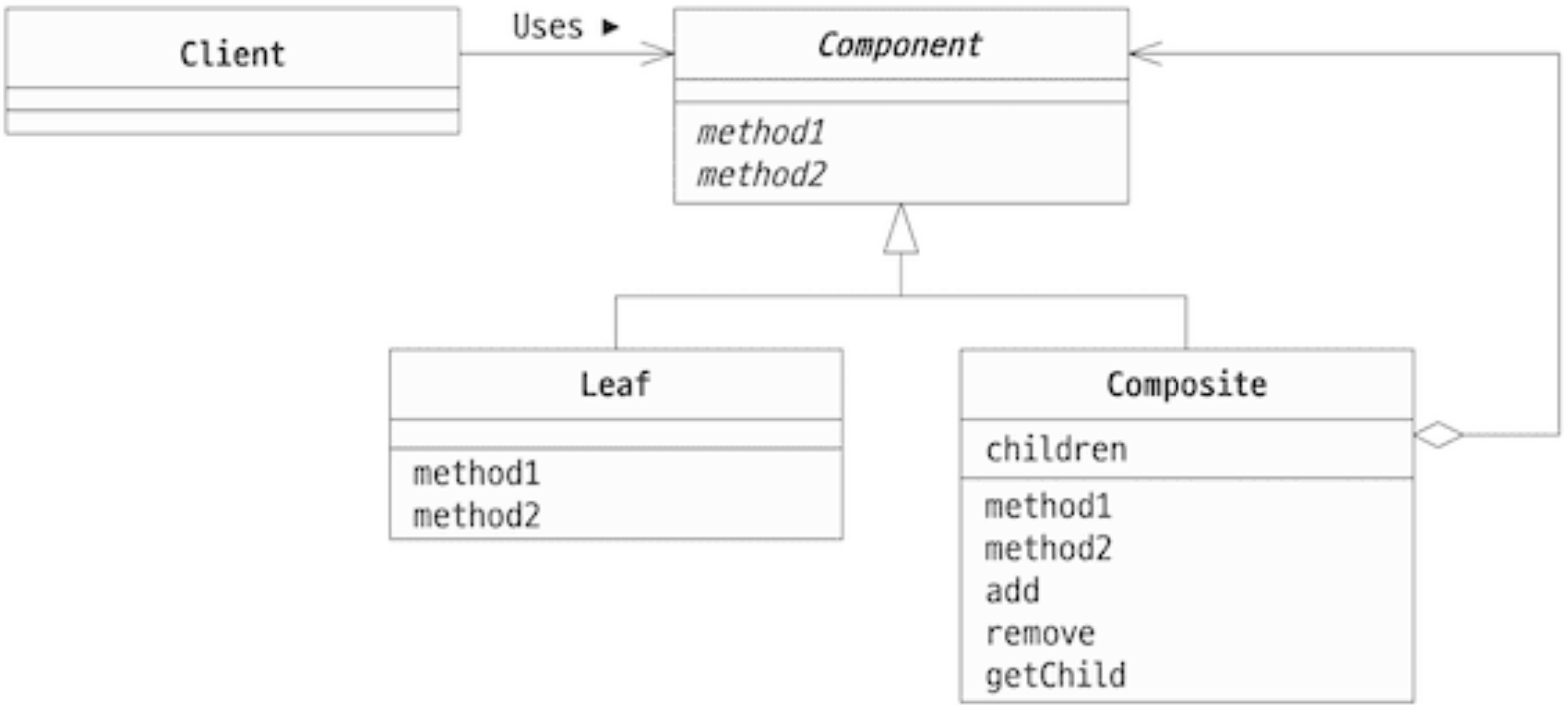
- **Client**

Step 2: One Common Interface

- Both **Item** and **Box** share the same set of actions (interfaces):
  - `get_name()`
  - `get_size()`
  - `print_list()`
- So the client doesn't care if it's dealing with an **Item** or a **Box**.

Step 3: The Trick

- A **Box** can hold:
  - Items (files)
  - Other Boxes (directories)
- This is **recursion**:
  - Boxes can contain boxes... which can contain boxes...



- Component → Thing (general word for both)
- Leaf → Item
- Composite → Box

Code

- Main Method
- Component Classes
- Tree Operations

Main Method

```
from entry import Entry
from file import File
from directory import Directory

def main():
    print("=== Composite Pattern Example ===\n")

    # Create composite structure
    root_dir = Directory("root")
    bin_dir = Directory("bin")
    root_dir.add(bin_dir)
    bin_dir.add(File("vi", 10000))

    # Uniform treatment
    print(f"Root size: {root_dir.get_size()}")
    root_dir.print_list()
```

Step 1: Create the composite structure

```
root_dir = Directory("root")
bin_dir = Directory("bin")
root_dir.add(bin_dir)
bin_dir.add(File("vi", 10000))
```

- **Directory** can contain both **files** and other **directories**.
- This creates a tree structure with uniform treatment.

Step 2: Use uniform interface

```
print(f"Root size: {root_dir.get_size()}")    # Composite operation
print(f"File size: {file.get_size()}")        # Leaf operation
```

- Both **File** and **Directory** respond to `get_size()` differently:
  - **File**: returns its own size
  - **Directory**: returns the sum of all children's sizes

Step 3: Recursive operations

```
root_dir.print_list()  # Prints entire tree structure
```

- **Composite** operations are naturally recursive.
- **Directory** prints itself and delegates to children.
- **File** prints only itself.

Uniform Treatment Example

This function works with both files and directories:

```
def process_entry(entry):
    print(f"Processing: {entry.get_name()}")
    print(f"Size: {entry.get_size()} bytes")

    # This works for both File and Directory!
    entry.print_list()

# Can be called with either type
process_entry(File("document.txt", 1000))
process_entry(Directory("my_folder"))
```

Discussion

Uniform Interface

Both individual objects (Leaf/Item) and compositions (Composite/Box) implement the same interface, allowing recursive composition and uniform treatment.

When to Use Composite

- When you have part-whole hierarchies
- When you want to ignore differences between individual and composite objects
- When you have tree structures
- When operations should work uniformly across the structure

Benefits of Composite

- **Uniform treatment** — same interface for leaves and composites
- **Recursive composition** — can build complex trees
- **Simplified client code** — no need to distinguish object types
- **Easy to add new components** — both leaves and composites
- **Natural tree traversal** — operations work recursively

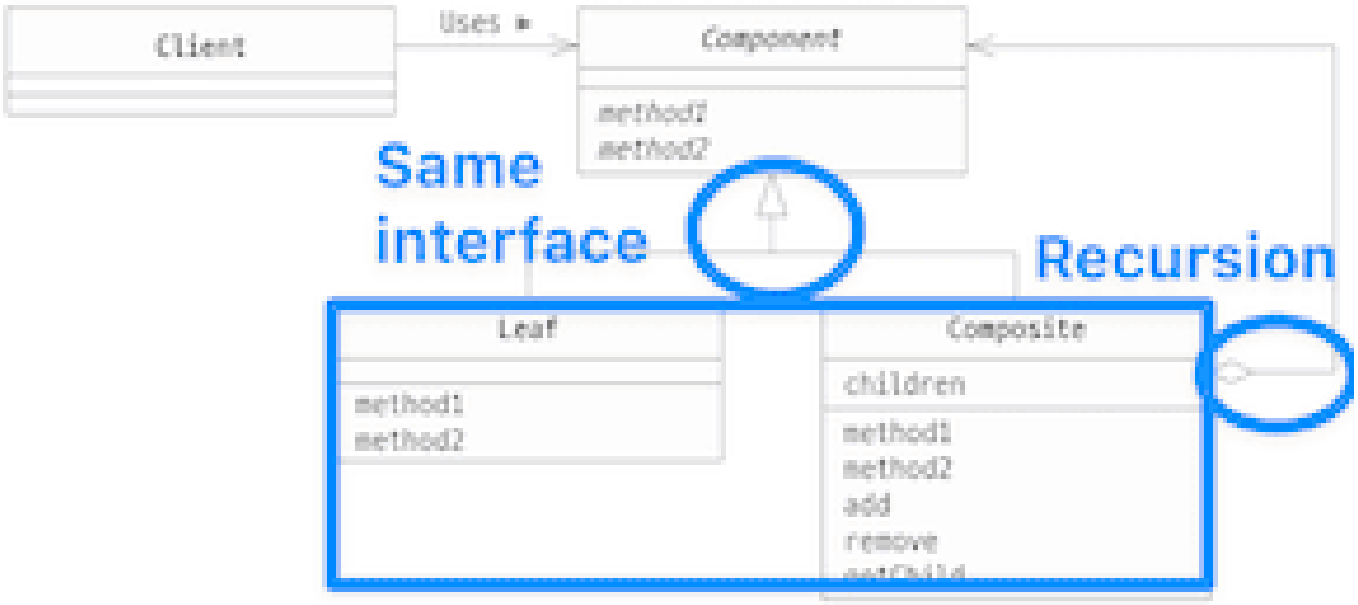
Potential Drawback

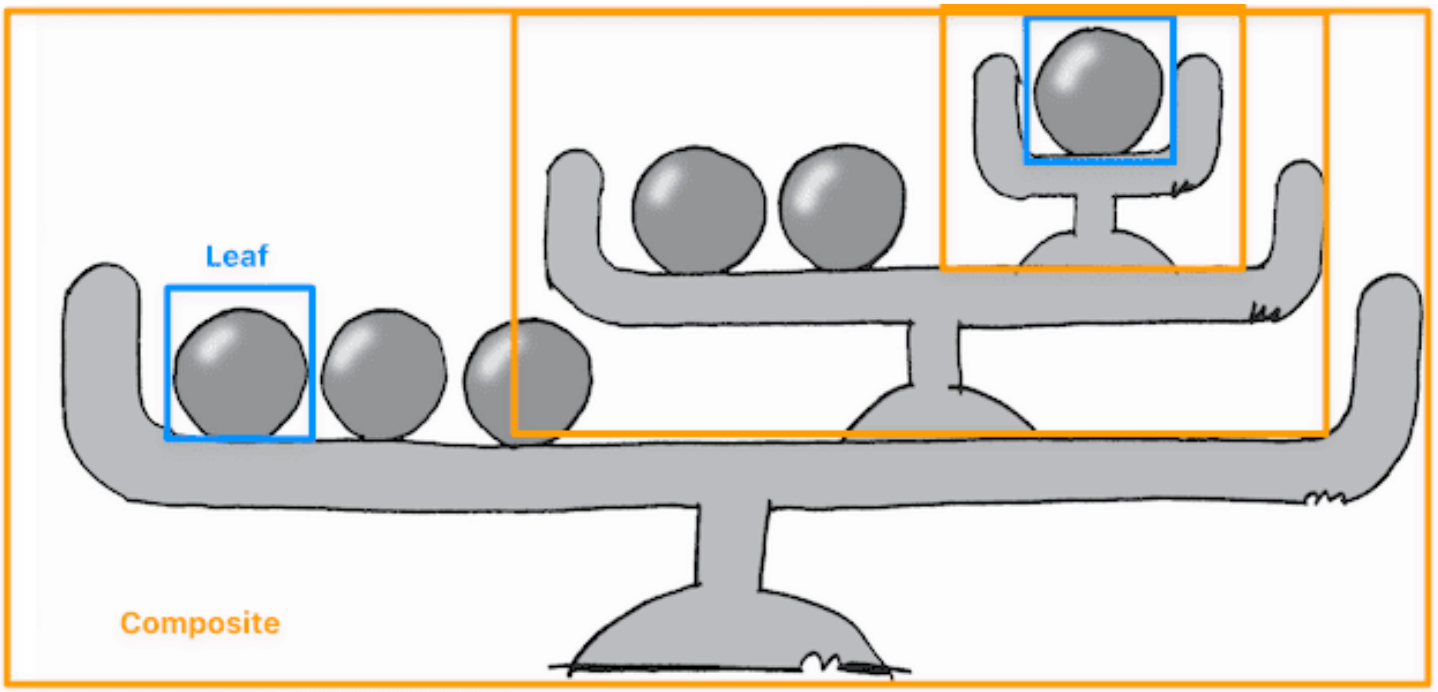
Design can become overly general** - the common interface might force leaf objects to implement operations that don't make sense for them (e.g., add/remove operations for leaf nodes).

Related Patterns

- **Iterator**: Often used to traverse composite structures

UML


Client
Uses
Component
method1
method2
Same interface
Recursion
Leaf
method1
method2
Composite
children
method1
method2
add
remove


Leaf
Composite