

Chapter

26

Composite

26.1 The Problem

Many problems require our software to handle *part-whole* objects, that is, objects that are defined by a hierarchical structure. As an example, consider folders on a file system. Folders contain files but they often also contain other folders that again may contain files and folders. Another example is a graphics editor in which several graphical objects may be grouped into a single object and moved, resized, and manipulated as a single entity. As such a group of figures may in itself be part of a larger group, it has the same hierarchical, part-whole, structure.

By nature, the part objects (files, individual graphical objects) are different from the whole objects (folders, groups), and designers are inclined to let them be defined by different classes. For instance, the design may contain a `Folder` class having methods like `addFile`, `addFolder`, `removeFile`, etc., and another class `File` having methods like `delete`, `size`, etc. However, many operations are actually the same irrespective if it is a folder or a file: both may be moved around in the folder structure, may be deleted, have a size, may be made read-only, etc. This similarity has the unfortunate consequence that the code must contain a lot of conditional statements just to pacify the type system. To see this, consider a user that invokes the `size` operation on some item in the folder hierarchy:

```
private static void displaySize(Object item) {
    if (item instanceof File) {
        File file = (File) item;
        System.out.println( "File size is "+file.size() );
    } else if (item instanceof Folder) {
        Folder folder = (Folder) item;
        System.out.println( "Folder size is "+folder.size() );
    }
}
```

These if-statements and casts will be present for every operation that is shared by folders and files. The stability and changeability problems associated with large amounts of similar looking code leads to the solution proposed by the COMPOSITE pattern.

26.2 A Solution

A solution is to apply the ① principle *program to an interface*, and define a common interface for both the *part* entity as well as the *whole* entity. Both part and whole entities are *components* which are defined by a **Component** interface. The methods of the **Component** interface defines all the methods that both part and whole objects may have, like `addComponent`, `toggleReadOnly` and `size` for a folder structure. As a partial interface for a folder hierarchy (implementing only the responsibility to add folders/files and calculate size to keep the demonstration code small), it may look like this:

Fragment: chapter/composite/CompositeDemo.java

```
/** Define the Component interface
 * (partial for a folder hierarchy) */
interface Component {
    public void addComponent(Component child);
    public int size();
}
```

The **Component** interface is implemented by two classes, the **File** class that defines the *part* objects, and the **Folder** class that defines the *whole* objects. To handle many of the methods in the **Folder** class, the ② principle is used to *recursively compose the behavior*. As an example, consider the `size()` method: for a file it is just the size in bytes of the file on the hard disk, but for a folder, the size is calculated by summing the individual sizes of each entry in the folder:

Fragment: chapter/composite/CompositeDemo.java

```
/** Define a (partial) folder abstraction */
class Folder implements Component {
    private List<Component> components = new ArrayList<Component>();
    public void addComponent(Component child) {
        components.add(child);
    }
    public int size() {
        int size = 0;
        for (Component c: components) {
            size += c.size();
        }
        return size;
    }
}
```

Note how the simple implementation of method `size()` is actually a depth first recursive traversal in the folder structure.

26.3 The Composite Pattern

This is the **COMPOSITE** pattern (design pattern box 26.1, page 53). Its intent is


Copyrighted Material. Do Not Distribute.

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

The central roles are the **Component**, defining the common interface for both part and whole objects, the **Leaf** that defines the primitive, atomic, part component, and finally the **Composite** that aggregates components.

The main advantage of the COMPOSITE pattern is the identical interface, shared between leaf and composite entities in the hierarchical data structure. This makes the use of the individual entities look similar. To some extent it often also allows an abstract **AbstractComponent** class to define some operations that are implemented identically in both the leaf and the composite class.

It does, however, also open for a concern, namely the methods to handle the list of components in the composite object. The **add** and **remove** methods can add and remove components to a component, but these operations are of course meaningless for a leaf object as it has no substructure.

 Restate this reflection in terms of the cohesion property discussed in Chapter 10.

One solution is to remove the **add** and **remove** methods from the **Component** interface, and only define them in **Composite**. The problem, however, is that we are then back in the switch and cast problem that **Composite** sets out to solve. Thus this pattern trades lower cohesion of the leaf abstraction in order to get a uniform interface for all abstractions in the part-whole structure.

26.4 Review Questions

Describe the COMPOSITE pattern. What problem does it solve? What is its structure? What roles and responsibilities are defined? What are the benefits and liabilities?

26.5 Further Exercises

Exercise 26.1. Source code directory:
chapter/composite

Review the **CompositeDemo** code and extend it.

1. Add a method **print** to print the hierarchical structure of any item. The substructure should be shown by indentation and file marked with an a star character, like

```
root
  programs
    emacs.exe *
    vi.exe *
  src
    editor.java *
    utilities
      search.java *
```

2. Add a method `delete` that deletes any item.

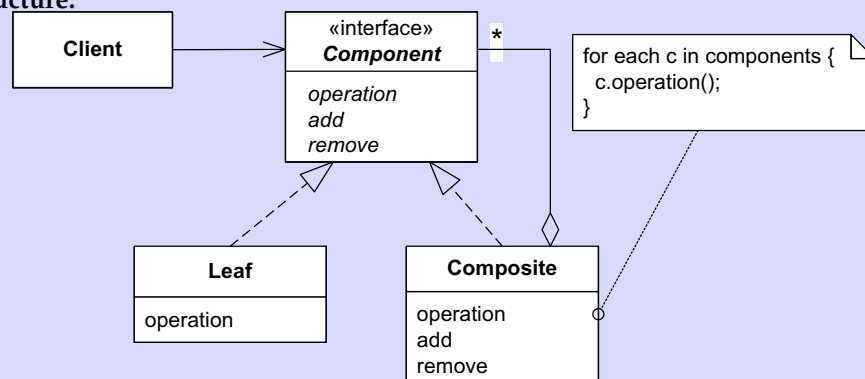
[26.1] Design Pattern: Composite

Intent Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Problem Handling of tree data structures.

Solution Define a common interface for composite and atomic components alike. Define composites in terms of a set of children, each either a composite or atomic component. Define composite behavior in terms of aggregating or composing behavior of each child.

Structure:



Roles **Component** defines a common interface. **Composite** defines a component by means of aggregating other components. **Leaf** defines a primitive, atomic, component i.e. one that has no substructure.

Cost - Benefit It defines a *hierarchy of primitive and composite objects*. It makes the *client interface uniform* as it does not need to know if it is a simple or composite component. It is *easy to add new kinds of components* as they will automatically work with the existing components. A liability is that the *design can become overly general* as it is difficult to constrain the types of leaves a composite may contain. The *interfaces may method bloat* with methods that are irrelevant; for instance an **add** method in a leaf.