15

COMPOSITE

This pattern was previously described in GoF95.

DESCRIPTION

Every component or object can be classified into one of the two categories — Individual Components or Composite Components — which are composed of individual components or other composite components. The Composite pattern is useful in designing a common interface for both individual and composite components so that client programs can view both the individual components and groups of components uniformly. In other words, the Composite design pattern allows a client object to treat both single components and collections of components in an identical manner.

This can also be explained in terms of a tree structure. The Composite pattern allows uniform reference to both Nonterminal nodes (which represent collections of components or composites) and terminal nodes (which represent individual components).

EXAMPLE

Let us create an application to simulate the Windows/UNIX file system. The file system consists mainly of two types of components — directories and files. Directories can be made up of other directories or files, whereas files cannot contain any other file system component. In this aspect, directories act as nonterminal nodes and files act as terminal nodes of a tree structure.

DESIGN APPROACH I

Let us define a common interface for both directories and files in the form of a Java interface `FileSystemComponent` (Figure 15.1). The `FileSystemComponent` interface declares methods that are common for both file components and directory components.

Let us further define two classes — `FileComponent` and `DirComponent` — as implementers of the common `FileSystemComponent` interface. Figure 15.2 shows the resulting class hierarchy.

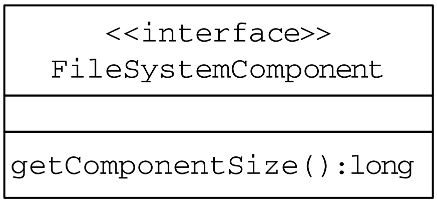
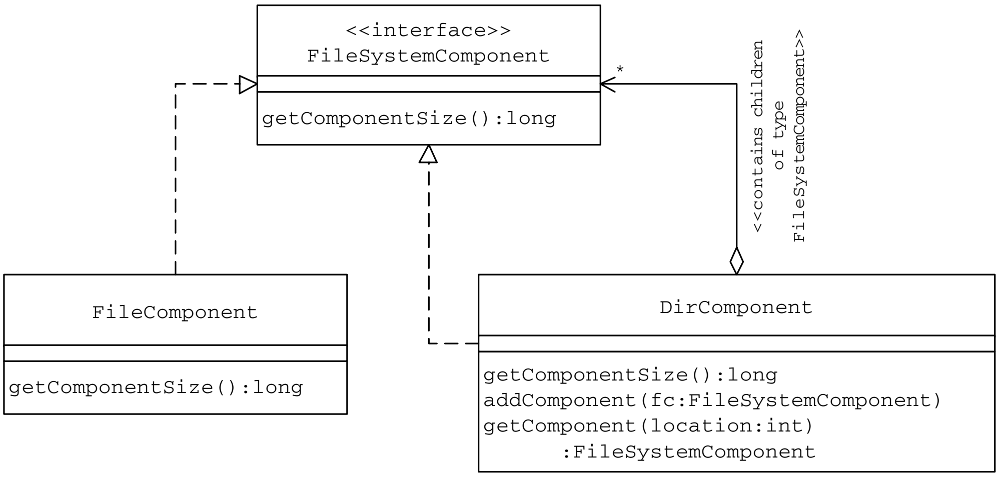


Figure 15.1 The Common `FileSystemComponent` Interface



<<interface>>
FileSystemComponent
getComponentSize():long
*
<<contains children of type FileSystemComponent>>
FileComponent
getComponentSize():long
DirComponent
getComponentSize():long
addComponent(fc:FileSystemComponent)
getComponent(location:int) :FileSystemComponent

Figure 15.2 The `FileSystemComponent` Class Hierarchy

FileComponent

The `FileComponent` class represents a file in the file system and offers implementation for the following methods.

getComponentSize()

This method returns the size (in kilobytes) of the file represented by the `File-Component` object.

DirComponent

This class represents a directory in the file system. Since directories are composite entities, the `DirComponent` provides methods to deal with the components it contains. These methods are in addition to the common `getComponentSize` method declared in the `FileSystemComponent` interface.

addComponent(FileSystemComponent)

This method is used by client applications to add different `DirComponent` and `FileComponent` objects to a `DirComponent` object.

getComponent(int)

The `DirComponent` stores the other `FileSystemComponent` objects inside a vector. This method is used to retrieve one such object stored at the specified location.

getComponentSize()

This method returns the size (in kilobytes) of the directory represented by the `DirComponent` object. As part of the implementation, the `DirComponent` object iterates over the collection of `FileSystemComponent` objects it contains, in a recursive manner, and sums up the sizes of all individual `FileComponents.` The final sum is returned as the size of the directory it represents.

A typical client would first create a set of `FileSystemComponent` objects (both `DirComponent` and `FileComponent` instances). It can use the `addComponent` method of the `DirComponent` to add different `FileSystemComponents` to a `DirComponent`, creating a hierarchy of file system `(FileSystemComponent)` objects.

When the client wants to query any of these objects for its size, it can simply invoke the `getComponentSize` method. The client does not have to be aware of the calculations involved or the manner in which the calculations are carried out in determining the component size. In this aspect, the client treats both the `FileComponent` and the `DirComponent` object in the same manner. No separate code is required to query `FileComponent` objects and `DirComponent` objects for their size.

Though the client treats both the `FileComponent` and `DirComponent` objects in a uniform manner in the case of the common `getComponentSize` method, it does need to distinguish when calling composite specific methods such as `addComponent` and `getComponent` defined exclusively in the `DirComponent.` Because these methods are not available with `FileComponent` objects, the client needs to check to make sure that the `FileSystemComponent` object it is working with is in fact a `DirComponent` object.

The following Design Approach II eliminates this requirement from the client.

DESIGN APPROACH II

The objective of this approach is to:

- Provide the same advantage of allowing the client application to treat both the composite `DirComponent` and the individual `FileComponent` objects in a uniform manner while invoking the `getComponentSize` method

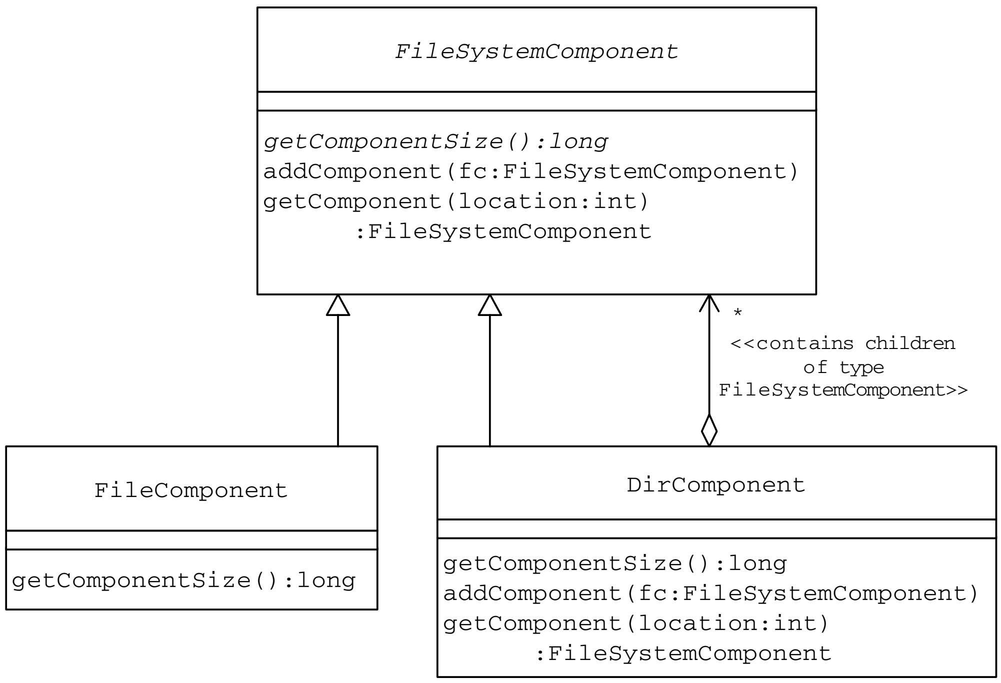


Figure 15.3 Class Association

- Free the client application from having to check to make sure that the FileSystemComponent it is dealing with is an instance of the DirComponent class while invoking any of the composite-specific methods such as addComponent or getComponent

In the new design (Figure 15.3), the composite-specific addComponent and getComponent methods are moved to the common interface FileSystemComponent. The FileSystemComponent provides the default implementation for these methods and is designed as an abstract class (Listing 15.1).

The default implementation of these methods consists of what is applicable to FileComponent objects. FileComponent objects are individual objects and do not contain other FileSystemComponent objects within. Hence, the default implementation does nothing and simply throws a custom CompositeException exception. The derived composite DirComponent class overrides these methods to provide custom implementation (Listing 15.2).

Because there is no change in the way the common getComponentSize method is designed, the client will still be able to treat both the composite DirComponent and FileComponent objects identically.

Because the common parent FileSystemComponent class now contains default implementations for the addComponent and the getComponent methods, the client application does not need to make any check before making a call to these composite-specific methods.

Listing 15.1 `FileSystemComponent` Abstract Class

```
public abstract class FileSystemComponent {
  String name;
  public FileSystemComponent(String cName) {
    name = cName;
  }
  public void addComponent(FileSystemComponent component)
    throws CompositeException {
    throw new CompositeException(
      "Invalid Operation. Not Supported");
  }
  public FileSystemComponent getComponent(int componentNum)
    throws CompositeException {
    throw new CompositeException(
      "Invalid Operation. Not Supported");
  }
  public abstract long getComponentSize();
}//End of class FileSystemComponent
```

Whenever a new composite-specific method such as `removeComponent` is to be added to the composite `DirComponent`, it also needs to be added to the parent `FileSystemComponent` class. The `FileSystemComponent` class must provide a `FileComponent`-specific default implementation for the new method.

The test client application `CompositeDemo` (Listing 15.3) creates a set of `DirComponent, FileComponent` objects. Using the composite method addComponent, it builds a file system component hierarchy as in Figure 15.4.

When the client needs to find the size of a file system component, it simply invokes the `getComponentSize` method on the `DirComponent` or `FileComponent` object that represents the file system component. The client does not need to treat the component differently depending on if the component is an individual component `(FileComponent)` or a composite component `(DirComponent)`. When the client `CompositeDemo` is run, the following output is displayed:

```
Main Folder Size= 10000kb
Sub Folder1 Size= 3000kb
File1 in Folder1 Size= 1000kb
```

Listing 15.2 `FileSystemComponent` Concrete Subclasses: `FileComponent` and `DirComponent`

```
public class FileComponent extends FileSystemComponent {
  private long size;
  public FileComponent(String cName, long sz) {
    super(cName);
    size = sz;
  }
  public long getComponentSize() {
    return size;
  }
}//End of class
public class DirComponent extends FileSystemComponent {
  Vector dirContents = new Vector();
  //individual files/sub folders collection
  public DirComponent(String cName) {
    super(cName);
  }
  public void addComponent(FileSystemComponent fc)
    throws CompositeException {
    dirContents.add(fc);
  }
  public FileSystemComponent getComponent(int location)
  throws CompositeException {
    return (FileSystemComponent) dirContents.elementAt(
              location);
  }
  public long getComponentSize() {
    long sizeOfAllFiles = 0;
    Enumeration e = dirContents.elements();
    while (e.hasMoreElements()) {
      FileSystemComponent component =
        (FileSystemComponent) e.nextElement();
      sizeOfAllFiles = sizeOfAllFiles +
                       (component.getComponentSize());
    }
    return sizeOfAllFiles;
  }
}//End of class
```

Listing 15.3 Client `CompositeDemo` Class

```
public class CompositeDemo {
  public static final String SEPARATOR = ”, ";
  public static void main(String[] args) {
    FileSystemComponent mainFolder =
      new DirComponent("Year2000");
    FileSystemComponent subFolder1 = new DirComponent("Jan");
    FileSystemComponent subFolder2 = new DirComponent("Feb");
    FileSystemComponent folder1File1 =
      new FileComponent("Jan1DataFile.txt,”1000);
    FileSystemComponent folder1File2 =
      new FileComponent("Jan2DataFile.txt”,2000);
    FileSystemComponent folder2File1 =
      new FileComponent("Feb1DataFile.txt”,3000);
    FileSystemComponent folder2File2 =
      new FileComponent("Feb2DataFile.txt”,4000);
    try {
      mainFolder.addComponent(subFolder1);
      mainFolder.addComponent(subFolder2);
      subFolder1.addComponent(folder1File1);
      subFolder1.addComponent(folder1File2);
      subFolder2.addComponent(folder2File1);
      subFolder2.addComponent(folder2File2);
    } catch (CompositeException ex) {
      //
    }
    //Client refers to both composite &
    //individual components in a uniform manner
    System.out.println(" Main Folder Size= " +
                       mainFolder.getComponentSize() + "kb");
    System.out.println(" Sub Folder1 Size= " +
                       subFolder1.getComponentSize() + "kb");
    System.out.println(" File1 in Folder1 size= " +
                       folder1File1.getComponentSize() + "kb");
  }
}
```

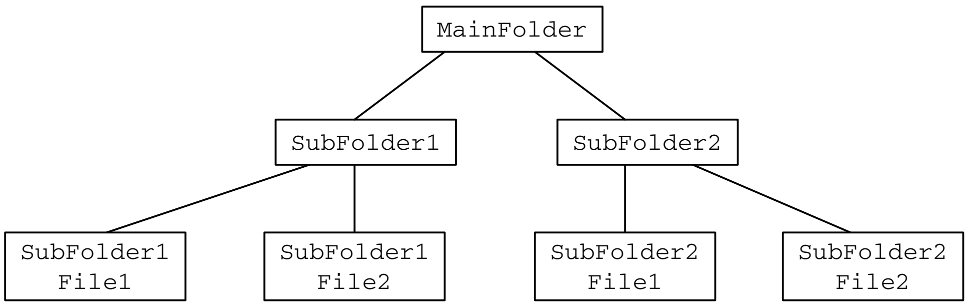


Figure 15.4 Example Application: `FileSystemComponent` Hierarchy

PRACTICE QUESTIONS

1. The following is an example of the FQDN (fully qualified domain name) on the Internet:

```
nwest.sales.DomainName.com
```

   It consists of different subdomains. Each such subdomain can be mapped onto a specific directory on the file system of the computer where the `DomainName.com` is hosted. Each such subdomain can have different HTML files, which can be accessed through a URL. Thus a subdomain and the set of HTML files can be viewed as two main components of a Web site.
   a. Define a subdomain hierarchy for an example domain.
   b. Create an application that uses the Composite pattern to:
      i. Display the directory a given subdomain is mapped onto
      ii. Display the URLs of Web site components (either subdomains or single HTML files) in a uniform manner
2. Let us consider the HTML `<frameset>` tag. The `<frameset>` tag is used to divide a Web page into different sections for the purpose of displaying multiple Web pages. Each such section can display a separate Web page. The actual Web page to be displayed inside a section can be specified using the `<frame>` tag. Further, a `<frameset>` tag can be nested, allowing further division of these sections.

```
E.g.,
<html>
<head></head>
<frameset rows="20%,80%">
   <frameset rows="100,200">
        <frame src="frame1.html">
          <frame src="frame2.html">
     </frameset>
     <frame src="frame3.html">
```

```
</frameset>
</html>
```

From the description of the `<frameset>` and `<frame>` tags above, it can be seen that these can be arranged in a tree-like structure with each `<frameset>` tag as a nonterminal node and each `<frame>` tag as a terminal node.

a. Design two classes — `FrameSet` and `Frame` — to represent the `<frameset>` and the `<frame>` tags, respectively.
b. Define an operation `getSourceFiles()` on these classes that returns the HTML file(s) specified to be displayed by a specific `FrameSet` or `Frame` object.
c. Design and implement this operation applying the Composite pattern so that a client can refer to these classes in an identical manner.

3. A typical product database consists of two types of product components — product categories and product items. A product category is generally composite in nature. It can contain product items and also other product categories as its subcategories.

   Example Product Categories:

   a. Computers
   b. Desktops
   c. Laptops
   d. Peripherals
   e. Printers
   f. Cables

   The Computers product category contains both the Desktops and the Laptops product categories as its subcategories. The Desktop category can contain a product item such as Compaq Presario 5050.

   Product items are usually individual, in the sense that they do not contain any product component within.

   Design and implement an application to list the dollar value of a product component. Use the Composite pattern to allow the client application to refer to both the product categories and the product items in a uniform manner.