

java设计模式之组合模式

学习难度：★★★☆☆，使用频率：★★★★☆】

树形结构在软件中随处可见，例如操作系统中的目录结构、应用软件中的菜单、办公系统中的公司组织结构等等，如何运用面向对象的方式来处理这种树形结构是组合模式需要解决的问题，组合模式通过一种巧妙的设计方案使得用户可以一致性地处理整个树形结构或者树形结构的一部分，也可以一致性地处理树形结构中的叶子节点（不包含子节点的节点）和容器节点（包含子节点的节点）。下面将学习这种用于处理树形结构的组合模式。

11.1 设计杀毒软件的框架结构

Sunny软件公司欲开发一个杀毒(AntiVirus)软件，该软件既可以对某个文件夹(Folder)杀毒，也可以对

在介绍Sunny公司开发人员提出的初始解决方案之前，我们先来分析一下操作系统中的文件目录结构，例如在Windows操作系统中，存在如图11-1所示目录结构：

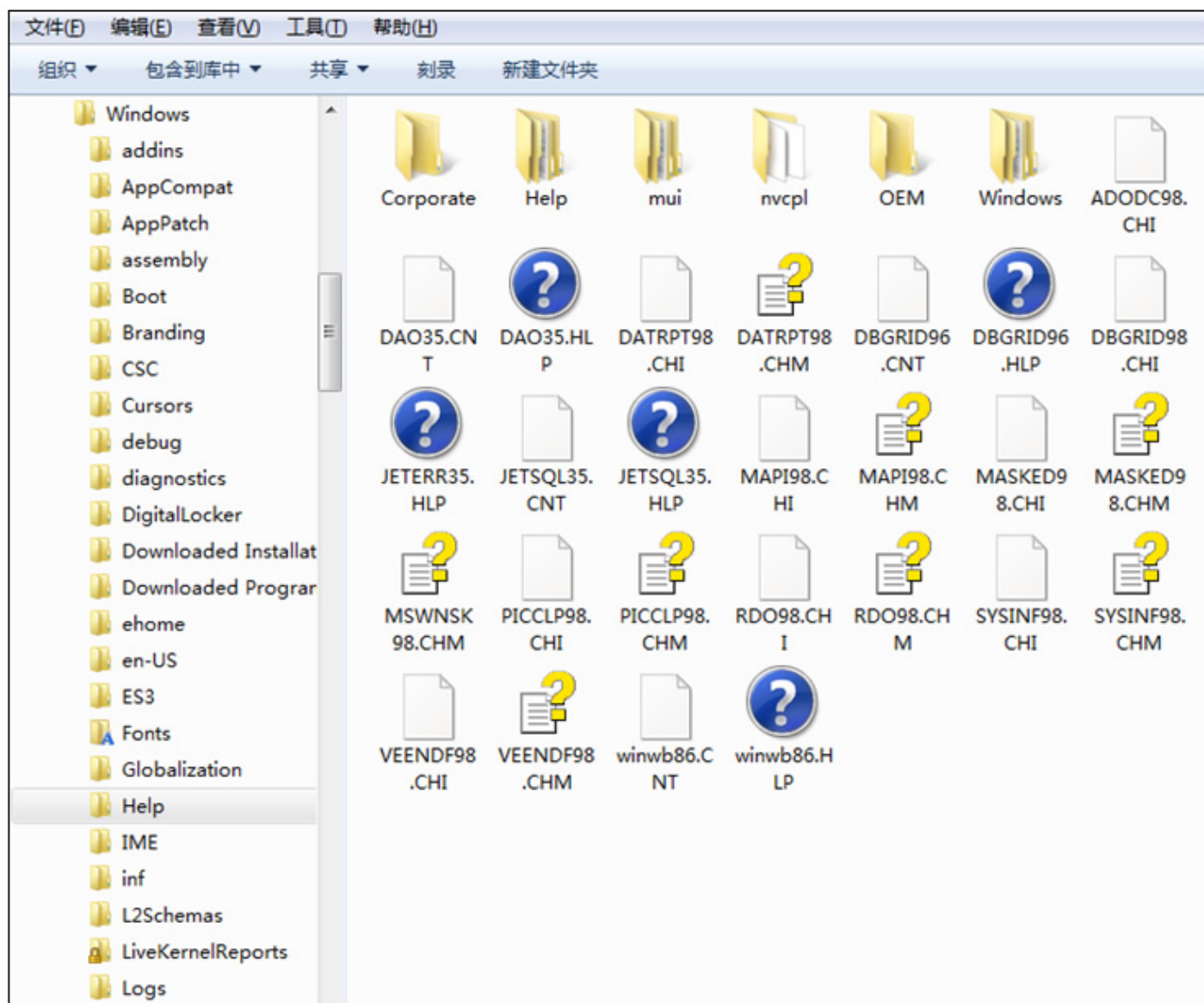


图11-1 Windows目录结构

图11-1可以简化为如图11-2所示树形目录结构：

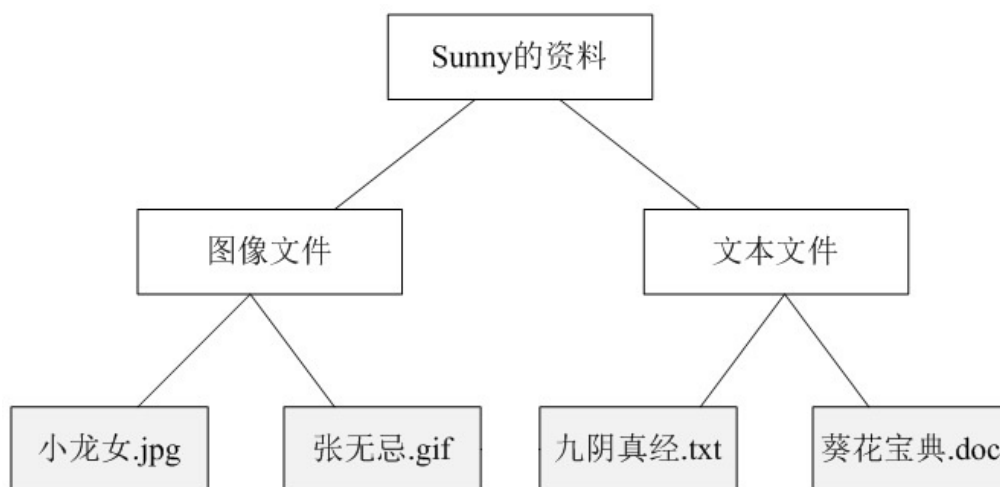


图11-2 树形目录结构示意图

我们可以看出，在图11-2中包含**文件（灰色节点）**和**文件夹（白色节点）**两类不同的元素，其中在文件夹中可以包含文件，还可以继续包含子文件夹，但是在文件中不能再包含子文件或者子文件夹。在此，我们可以称文件夹为容器（**Container**），而不同类型的各种文件是其成员，也称为叶子(Leaf)，一个文件夹也可以作为另一个更大的文件夹的成

员。如果我们现在要对某一个文件夹进行操作，如查找文件，那么需要对指定的文件夹进行遍历，如果存在子文件夹则打开其子文件夹继续遍历，如果是文件则判断之后返回查找结果。

Sunny软件公司的开发人员通过分析，决定使用面向对象的方式来实现对文件和文件夹的操作，定义了如下图像文件类ImageFile、文本文件类TextFile和文件夹类Folder：



//为了突出核心框架代码，我们对杀毒过程的实现进行了大量简化

```
import java.util.*;
```

//图像文件类

```
class ImageFile {
    private String name;

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

    public void killVirus() {
        //简化代码，模拟杀毒
        System.out.println("----对图像文件'" + name + "'进行杀毒");
    }
}
```

//文本文件类

```
class TextFile {
    private String name;

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

    public void killVirus() {
        //简化代码，模拟杀毒
        System.out.println("----对文本文件'" + name + "'进行杀毒");
    }
}
```

//文件夹类

```
class Folder {
    private String name;
    //定义集合folderList，用于存储Folder类型的成员
    private ArrayList<Folder> folderList = new ArrayList<Folder>();
    //定义集合imageList，用于存储ImageFile类型的成员
    private ArrayList<ImageFile> imageList = new ArrayList<ImageFile>();
    //定义集合textList，用于存储TextFile类型的成员
    private ArrayList<TextFile> textList = new ArrayList<TextFile>();

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

    //增加新的Folder类型的成员
    public void addFolder(Folder f) {
        folderList.add(f);
    }
}
```

```

}

//增加新的ImageFile类型的成员
public void addImageFile(ImageFile image) {
    imageList.add(image);
}

//增加新的TextFile类型的成员
public void addTextFile(TextFile text) {
    textList.add(text);
}

//需提供三个不同的方法removeFolder()、removeImageFile()和removeTextFile()来删除成员，代码省略

//需提供三个不同的方法getChildFolder(int i)、getChildImageFile(int i)和getChildTextFile(int i)来获取成员，代码省略

public void killVirus() {
    System.out.println("*****对文件夹'" + name + "'进行杀毒"); //模拟杀毒

    //如果是Folder类型的成员，递归调用Folder的killVirus()方法
    for(Object obj : folderList) {
        ((Folder)obj).killVirus();
    }

    //如果是ImageFile类型的成员，调用ImageFile的killVirus()方法
    for(Object obj : imageList) {
        ((ImageFile)obj).killVirus();
    }

    //如果是TextFile类型的成员，调用TextFile的killVirus()方法
    for(Object obj : textList) {
        ((TextFile)obj).killVirus();
    }
}
}

```



编写如下客户端测试代码进行测试：



```

class Client {
    public static void main(String args[]) {
        Folder folder1, folder2, folder3;
        folder1 = new Folder("Sunny的资料");
        folder2 = new Folder("图像文件");
        folder3 = new Folder("文本文件");

        ImageFile image1, image2;
        image1 = new ImageFile("小龙女.jpg");
        image2 = new ImageFile("张无忌.gif");

        TextFile text1, text2;
        text1 = new TextFile("九阴真经.txt");
        text2 = new TextFile("葵花宝典.doc");
    }
}

```

```
        folder2.addImageFile(image1);
        folder2.addImageFile(image2);
        folder3.addTextFile(text1);
        folder3.addTextFile(text2);
        folder1.addFolder(folder2);
        folder1.addFolder(folder3);

        folder1.killVirus();
    }
}
```



编译并运行程序，输出结果如下：

```
****对文件夹'Sunny的资料'进行杀毒
****对文件夹'图像文件'进行杀毒
----对图像文件'小龙女.jpg'进行杀毒
----对图像文件'张无忌.gif'进行杀毒
****对文件夹'文本文件'进行杀毒
----对文本文件'九阴真经.txt'进行杀毒
----对文本文件'葵花宝典.doc'进行杀毒
```

Sunny公司开发人员“成功”实现了杀毒软件的框架设计，但通过仔细分析，发现该设计方案存在如下问题：

(1) 文件夹类Folder的设计和实现都非常复杂，需要定义多个集合存储不同类型的成员，而且需要针对不同的成员提供增加、删除和获取等管理和访问成员的方法，存在大量的冗余代码，系统维护较为困难；

(2) 由于系统没有提供抽象层，客户端代码必须有区别地对待充当容器的文件夹Folder和充当叶子的ImageFile和TextFile，无法统一对它们进行处理；

(3) 系统的灵活性和可扩展性差，如果需要增加新的类型的叶子和容器都需要对原有代码进行修改，例如如果需要在系统中增加一种新类型的视频文件VideoFile，则必须修改Folder类的源代码，否则无法在文件夹中添加视频文件。

面对以上问题，Sunny软件公司的开发人员该如何来解决？这就需要用到本章将要介绍的组合模式，**组合模式为处理树形结构提供了一种较为完美的解决方案，它描述了如何将容器和叶子进行递归组合，使得用户在使用时无须对它们进行区分，可以一致地对待容器和叶子。**

11.2 组合模式概述

对于树形结构，当容器对象（如文件夹）的某一个方法被调用时，将遍历整个树形结构，寻找也包含这个方法的成员对象（可以是容器对象，也可以是叶子对象）并调用执行，牵一而动百，其中使用了递归调用的机制来对整个结构进行

处理。由于容器对象和叶子对象在功能上的区别，在使用这些对象的代码中必须有区别地对待容器对象和叶子对象，而实际上大多数情况下我们希望一致地处理它们，因为对于这些对象的区别对待将会使得程序非常复杂。组合模式为解决此类问题而诞生，它可以让叶子对象和容器对象的使用具有一致性。

组合模式定义如下：

组合模式(Composite Pattern)：组合多个对象形成树形结构以表示具有“整体—部分”关系的层次结构。

在组合模式中引入了抽象构件类Component，它是所有容器类和叶子类的公共父类，客户端针对Component进行编程。组合模式结构如图11-3所示：

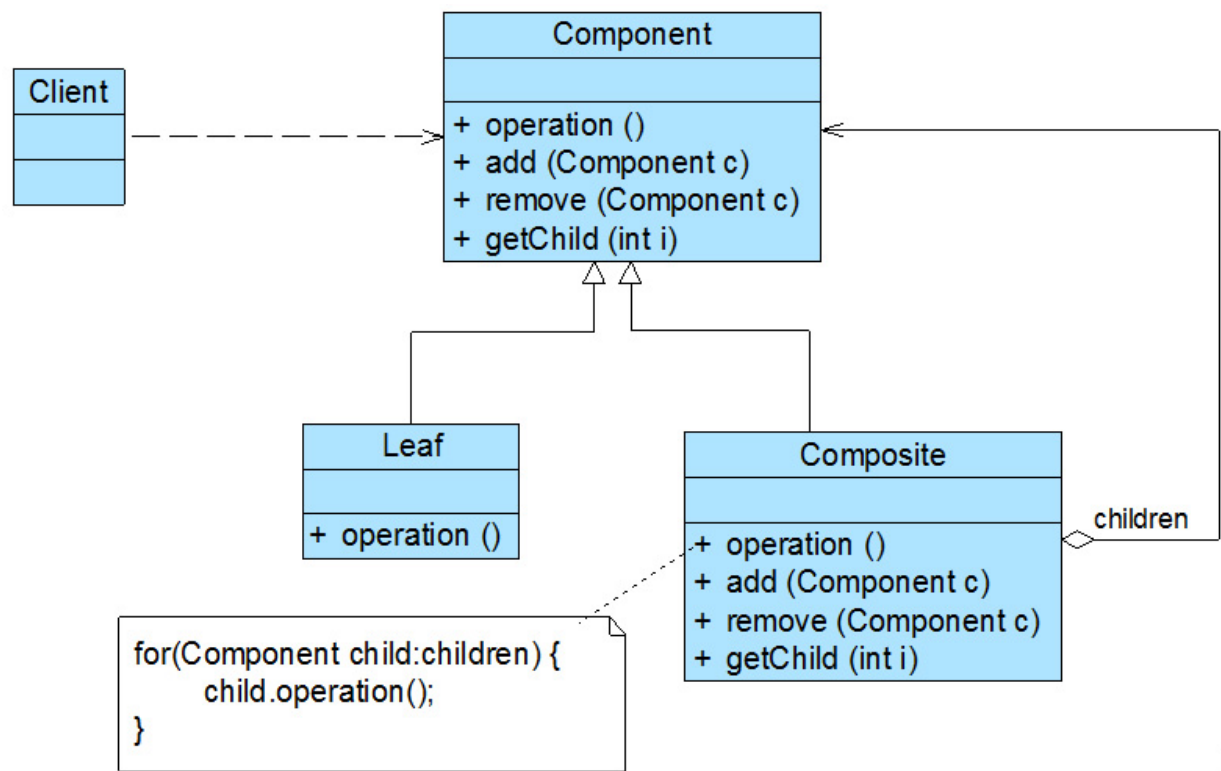


图11-3 组合模式结构图


在组合模式结构图中包含如下几个角色：

- **Component（抽象构件）**：它可以是接口或抽象类，为叶子构件和容器构件对象声明接口，在该角色中可以包含所有子类共有行为的声明和实现。在抽象构件中定义了访问及管理它的子构件的方法，如增加子构件、删除子构件、获取子构件等。
- **Leaf（叶子构件）**：它在组合结构中表示叶子节点对象，叶子节点没有子节点，它实现了在抽象构件中定义的行为。对于那些访问及管理子构件的方法，可以通过异常等方式进行处理。
- **Composite（容器构件）**：它在组合结构中表示容器节点对象，容器节点包含子节点，其子节点可以是叶子节点，也可以是容器节点，它提供一个集合用于存储子节点，实现了在抽象构件中定义的行为，包括那些访问及管理子构件的方法，在其业务方法中可以递归调用其子节点的业务方法。


组合模式的关键是定义了一个抽象构件类，它既可以代表叶子，又可以代表容器，而客户端针对该抽象构件类进行编程，无须知道它到底表示的是叶子还是容器，可以对其进行统一处理。同时容器对象与抽象构件类之间还建立一个聚合关联关系，在容器对象中既可以包含叶子，也可以包含容器，以此实现递归组合，形成一个树形结构。

如果不使用组合模式，客户端代码将过多地依赖于容器对象复杂的内部实现结构，容器对象内部实现结构的变化将引起客户端代码的频繁变化，带来了代码维护复杂、可扩展性差等弊端。组合模式的引入将在一定程度上解决这些问题。

下面通过简单的示例代码来分析组合模式的各个角色的用途和实现。对于组合模式中的抽象构件角色，其典型代码如下所示：



```
abstract class Component {  
    public abstract void add(Component c); //增加成员  
    public abstract void remove(Component c); //删除成员  
    public abstract Component getChild(int i); //获取成员  
    public abstract void operation(); //业务方法  
}
```




一般将抽象构件类设计为接口或抽象类，将所有子类共有方法的声明和实现放在抽象构件类中。对于客户端而言，将针对抽象构件编程，而无须关心其具体子类是容器构件还是叶子构件。

如果继承抽象构件的是叶子构件，则其典型代码如下所示：




```
class Leaf extends Component {  
    public void add(Component c) {  
        //异常处理或错误提示  
    }  
  
    public void remove(Component c) {  
        //异常处理或错误提示  
    }  
  
    public Component getChild(int i) {  
        //异常处理或错误提示  
        return null;  
    }  
  
    public void operation() {  
        //叶子构件具体业务方法的实现  
    }  
}
```



作为抽象构件类的子类，在叶子构件中需要实现在抽象构件类中声明的所有方法，包括业务方法以及管理和访问子构件的方法，但是叶子构件不能再包含子构件，因此在**叶子构件中实现子构件管理和访问方法时需要提供异常处理或错误提示**。当然，这无疑会给叶子构件的实现带来麻烦。

如果继承抽象构件的是容器构件，则其典型代码如下所示：



```
class Composite extends Component {  
    private ArrayList<Component> list = new ArrayList<Component>();  
  
    public void add(Component c) {
```

```

        list.add(c);
    }

    public void remove(Component c) {
        list.remove(c);
    }

    public Component getChild(int i) {
        return (Component)list.get(i);
    }

    public void operation() {
        //容器构件具体业务方法的实现
        //递归调用成员构件的业务方法
        for(Object obj:list) {
            ((Component)obj).operation();
        }
    }
}

```



在容器构件中实现了在抽象构件中声明的所有方法，既包括业务方法，也包括用于访问和管理成员子构件的方法，如 add()、remove()和getChild()等方法。需要注意的是在实现具体业务方法时，由于容器构件充当的是容器角色，包含成员构件，因此它将调用其成员构件的业务方法。**在组合模式结构中，由于容器构件中仍然可以包含容器构件，因此在对容器构件进行处理时需要使用递归算法**，即在容器构件的operation()方法中递归调用其成员构件的operation()方法。

思考

在组合模式结构图中，如果聚合关联关系不是从Composite到Component的，而是从Comp

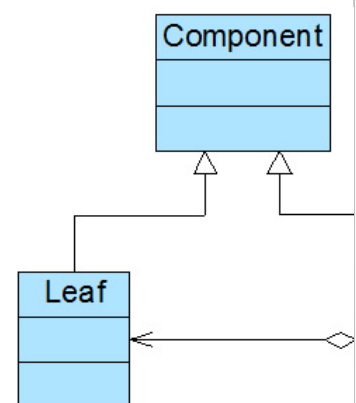


图11-4 组合模式思考

11.3 完整解决方案

为了让系统具有更好的灵活性和可扩展性，客户端可以一致地对待文件和文件夹，Sunny公

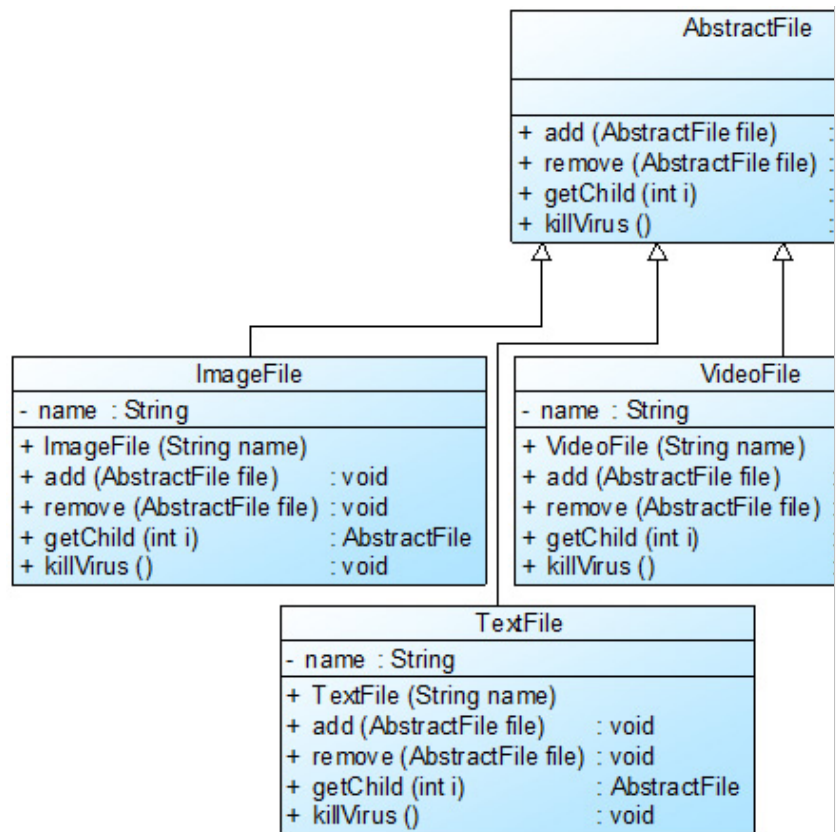


图11-5 杀毒软件框架设计

在图11-5中，AbstractFile充当抽象构件类，Folder充当容器构件类，ImageFile、TextFile和VideoFile充当叶子构件类。



```

import java.util.*;

//抽象文件类：抽象构件
abstract class AbstractFile {
    public abstract void add(AbstractFile file);
    public abstract void remove(AbstractFile file);
    public abstract AbstractFile getChild(int i);
    public abstract void killVirus();
}

//图像文件类：叶子构件
class ImageFile extends AbstractFile {
    private String name;

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

    public void add(AbstractFile file) {
        System.out.println("对不起，不支持该方法！");
    }
}
  
```

```

    }

    public void remove(AbstractFile file) {
        System.out.println("对不起，不支持该方法！");
    }

    public AbstractFile getChild(int i) {
        System.out.println("对不起，不支持该方法！");
        return null;
    }

    public void killVirus() {
        //模拟杀毒
        System.out.println("----对图像文件'" + name + "'进行杀毒");
    }
}

//文本文件类：叶子构件
class TextFile extends AbstractFile {
    private String name;

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

    public void add(AbstractFile file) {
        System.out.println("对不起，不支持该方法！");
    }

    public void remove(AbstractFile file) {
        System.out.println("对不起，不支持该方法！");
    }

    public AbstractFile getChild(int i) {
        System.out.println("对不起，不支持该方法！");
        return null;
    }

    public void killVirus() {
        //模拟杀毒
        System.out.println("----对文本文件'" + name + "'进行杀毒");
    }
}

//视频文件类：叶子构件
class VideoFile extends AbstractFile {
    private String name;

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

    public void add(AbstractFile file) {
        System.out.println("对不起，不支持该方法！");
    }

    public void remove(AbstractFile file) {

```

```

        System.out.println("对不起，不支持该方法！");
    }

    public AbstractFile getChild(int i) {
        System.out.println("对不起，不支持该方法！");
        return null;
    }

    public void killVirus() {
        //模拟杀毒
        System.out.println("----对视频文件'" + name + "'进行杀毒");
    }
}

//文件夹类：容器构件
class Folder extends AbstractFile {
    //定义集合fileList，用于存储AbstractFile类型的成员
    private ArrayList<AbstractFile> fileList=new ArrayList<AbstractFile>();
    private String name;

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

    public void add(AbstractFile file) {
        fileList.add(file);
    }

    public void remove(AbstractFile file) {
        fileList.remove(file);
    }

    public AbstractFile getChild(int i) {
        return (AbstractFile)fileList.get(i);
    }

    public void killVirus() {
        System.out.println("*****对文件夹'" + name + "'进行杀毒"); //模拟杀毒

        //递归调用成员构件的killVirus()方法
        for(Object obj : fileList) {
            ((AbstractFile)obj).killVirus();
        }
    }
}

```



编写如下客户端测试代码：



```

class Client {
    public static void main(String args[]) {
        //针对抽象构件编程
        AbstractFile file1,file2,file3,file4,file5,folder1,folder2,folder3,folder4;
    }
}

```

```

        folder1 = new Folder("Sunny的资料");
        folder2 = new Folder("图像文件");
        folder3 = new Folder("文本文件");
        folder4 = new Folder("视频文件");

        file1 = new ImageFile("小龙女.jpg");
        file2 = new ImageFile("张无忌.gif");
        file3 = new TextFile("九阴真经.txt");
        file4 = new TextFile("葵花宝典.doc");
        file5 = new VideoFile("笑傲江湖.rmvb");

        folder2.add(file1);
        folder2.add(file2);
        folder3.add(file3);
        folder3.add(file4);
        folder4.add(file5);
        folder1.add(folder2);
        folder1.add(folder3);
        folder1.add(folder4);

        //从"Sunny的资料"节点开始进行杀毒操作
        folder1.killVirus();
    }
}

```



编译并运行程序，输出结果如下：

```

****对文件夹'Sunny的资料'进行杀毒
****对文件夹'图像文件'进行杀毒
----对图像文件'小龙女.jpg'进行杀毒
----对图像文件'张无忌.gif'进行杀毒
****对文件夹'文本文件'进行杀毒
----对文本文件'九阴真经.txt'进行杀毒
----对文本文件'葵花宝典.doc'进行杀毒
****对文件夹'视频文件'进行杀毒
----对视频文件'笑傲江湖.rmvb'进行杀毒

```

由于在本实例中使用了组合模式，在抽象构件类中声明了所有方法，包括用于管理和访问子构件的方法，如add()方法和remove()方法等，因此在ImageFile等叶子构件类中实现这些方法时必须进行相应的异常处理或错误提示。在容器构件类Folder的killVirus()方法中将递归调用其成员对象的killVirus()方法，从而实现对整个树形结构的遍历。

如果需要更换操作节点，例如只需对文件夹“文本文件”进行杀毒，客户端代码只需修改一行即可，将代码：

```
folder1.killVirus();
```

改为：

```
folder3.killVirus();
```

输出结果如下：

```
****对文件夹'文本文件'进行杀毒
---对文本文件'九阴真经.txt'进行杀毒
---对文本文件'葵花宝典.doc'进行杀毒
```

在具体实现时，我们可以创建图形化界面让用户选择所需操作的根节点，无须修改源代码，符合“开闭原则”，客户端无须关心节点的层次结构，可以对所选节点进行统一处理，提高系统的灵活性。

11.4 透明组合模式与安全组合模式

通过引入组合模式，Sunny公司设计的杀毒软件具有良好的可扩展性，在增加新的文件类型时，无须修改现有类库代码，只需增加一个新的文件类作为AbstractFile类的子类即可，但是由于在AbstractFile中声明了大量用于管理和访问成员构件的方法，例如add()、remove()等方法，我们不得不在新增的文件类中实现这些方法，提供对应的错误提示和异常处理。为了简化代码，我们有以下两个解决方案：

解决方案一：将叶子构件的add()、remove()等方法的实现代码移至AbstractFile类中，由AbstractFile提供统一的默认实现，代码如下所示：



//提供默认实现的抽象构件类

```
abstract class AbstractFile {  
    public void add(AbstractFile file) {  
        System.out.println("对不起，不支持该方法！");  
    }  
  
    public void remove(AbstractFile file) {  
        System.out.println("对不起，不支持该方法！");  
    }  
  
    public AbstractFile getChild(int i) {  
        System.out.println("对不起，不支持该方法！");  
    }  
}
```

```

        return null;
    }

    public abstract void killVirus();
}

```

如果客户端代码针对抽象类AbstractFile编程，在调用文件对象的这些方法时将出现错误提示。如果不希望出现任何错误提示，我们可以在客户端定义文件对象时不使用抽象层，而直接使用具体叶子构件本身，客户端代码片段如下所示：

```

class Client {
    public static void main(String args[]) {
        //不能透明处理叶子构件
        ImageFile file1,file2;
        TextFile file3,file4;
        VideoFile file5;
        AbstractFile folder1,folder2,folder3,folder4;
        //其他代码省略
    }
}

```

这样就产生了一种不透明的使用方式，即在客户端不能全部针对抽象构件类编程，需要使用具体叶子构件类型来定义叶子对象。

解决方案二：除此之外，还有一种解决方法是在抽象构件AbstractFile中不声明任何用于访问和管理成员构件的方法，代码如下所示：

```

abstract class AbstractFile {
    public abstract void killVirus();
}

```

此时，由于在AbstractFile中没有声明add()、remove()等访问和管理成员的方法，其叶子构件子类无须提供实现；而且无论客户端如何定义叶子构件对象都无法调用到这些方法，不需要做任何错误和异常处理，容器构件再根据需要增加访问和管理成员的方法，但这时候也存在一个问题：客户端不得不使用容器类本身来声明容器构件对象，否则无法访问其中新增的add()、remove()等方法，如果客户端一致性地对待叶子和容器，将会导致容器构件的新增对客户端不可见，客户端代码对于容器构件无法再使用抽象构件来定义，客户端代码片段如下所示：

```

class Client {
    public static void main(String args[]) {

        AbstractFile file1,file2,file3,file4,file5;
        Folder folder1,folder2,folder3,folder4; //不能透明处理容器构件
        //其他代码省略
    }
}

```

在使用组合模式时，根据抽象构件类的定义形式，我们可将组合模式分为透明组合模式和安全组合模式两种形式：

(1) 透明组合模式

透明组合模式中，抽象构件Component中声明了所有用于管理成员对象的方法，包括add()、remove()以及getChild()等方法，这样做的好处是确保所有的构件类都有相同的接口。在客户端看来，叶子对象与容器对象所提供的方法是一致的，客户端可以相同地对待所有的对象。透明组合模式也是组合模式的标准形式，虽然上面的**解决方案一**在客户端可以有透明的实现方法，但是由于在抽象构件中包含add()、remove()等方法，因此它还是透明组合模式，透明组合模式的完整结构如图11-6所示：

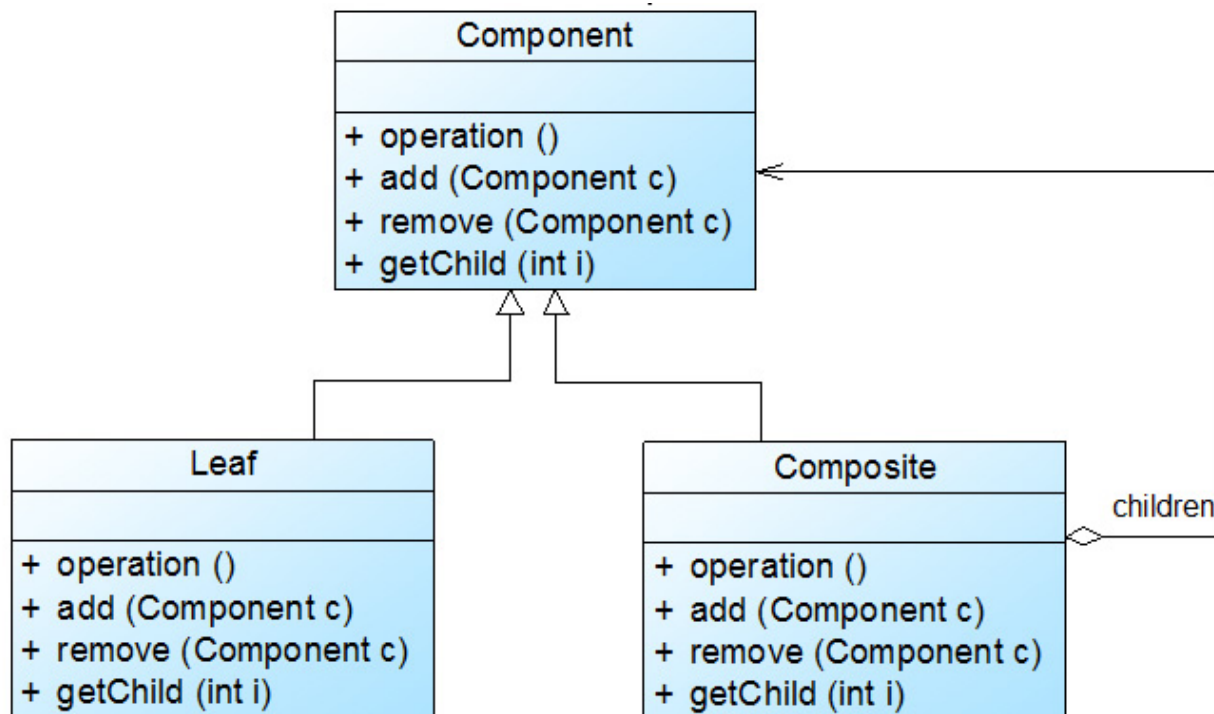


图11-6 透明组合模式结构图

透明组合模式的缺点是不够安全，因为叶子对象和容器对象在本质上是有所区别的。叶子对象不可能有下一个层次的对象，即不可能包含成员对象，因此为其提供add()、remove()以及getChild()等方法是没有意义的，这在编译阶段不会出错，但在运行阶段如果调用这些方法可能会出错（如果没有提供相应的错误处理代码）。

(2) 安全组合模式

安全组合模式中，在抽象构件Component中没有声明任何用于管理成员对象的方法，而是在Composite类中声明并实现这些方法。这种做法是安全的，因为根本不向叶子对象提供这些管理成员对象的方法，对于叶子对象，客户端不可能调用到这些方法，这就是**解决方案二**所采用的实现方式。安全组合模式的结构如图11-7所示：

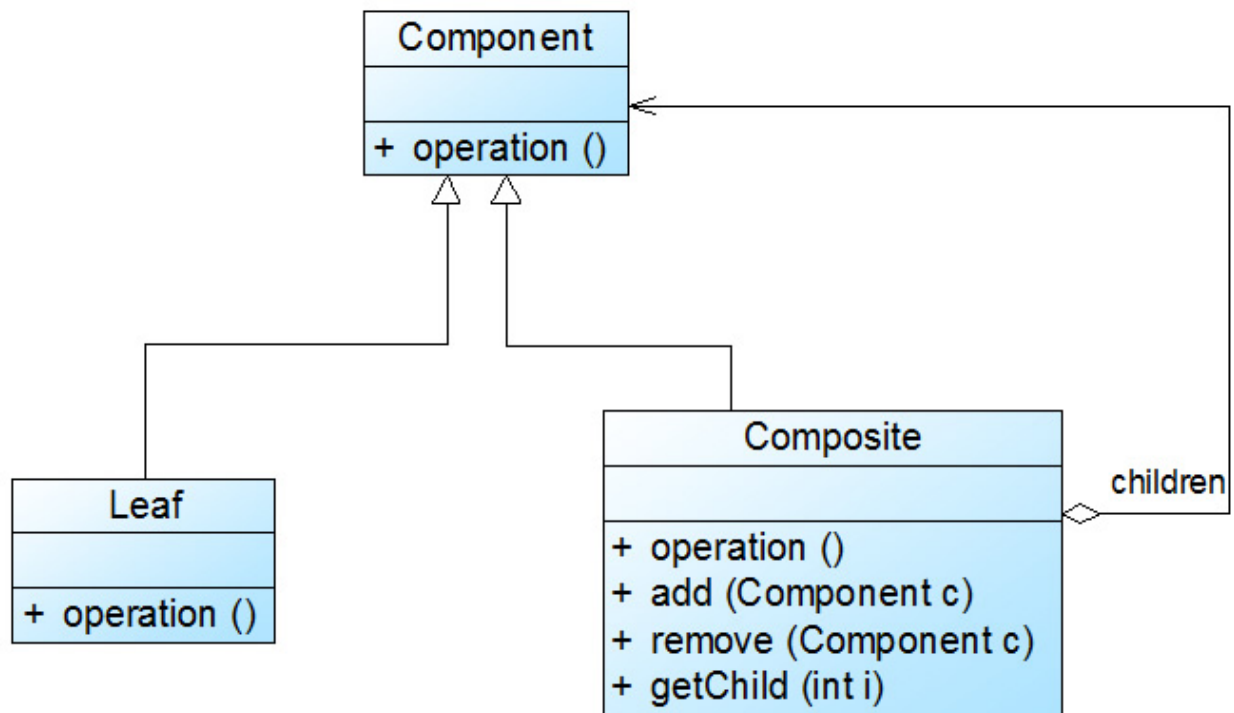


图11-7 安全组合模式结构图

安全组合模式的缺点是不够透明，因为叶子构件和容器构件具有不同的方法，且容器构件中那些用于管理成员对象的方法没有在抽象构件类中定义，因此客户端不能完全针对抽象编程，必须有区别地对待叶子构件和容器构件。在实际应用中，安全组合模式的使用频率也非常高，在Java AWT中使用的组合模式就是安全组合模式。