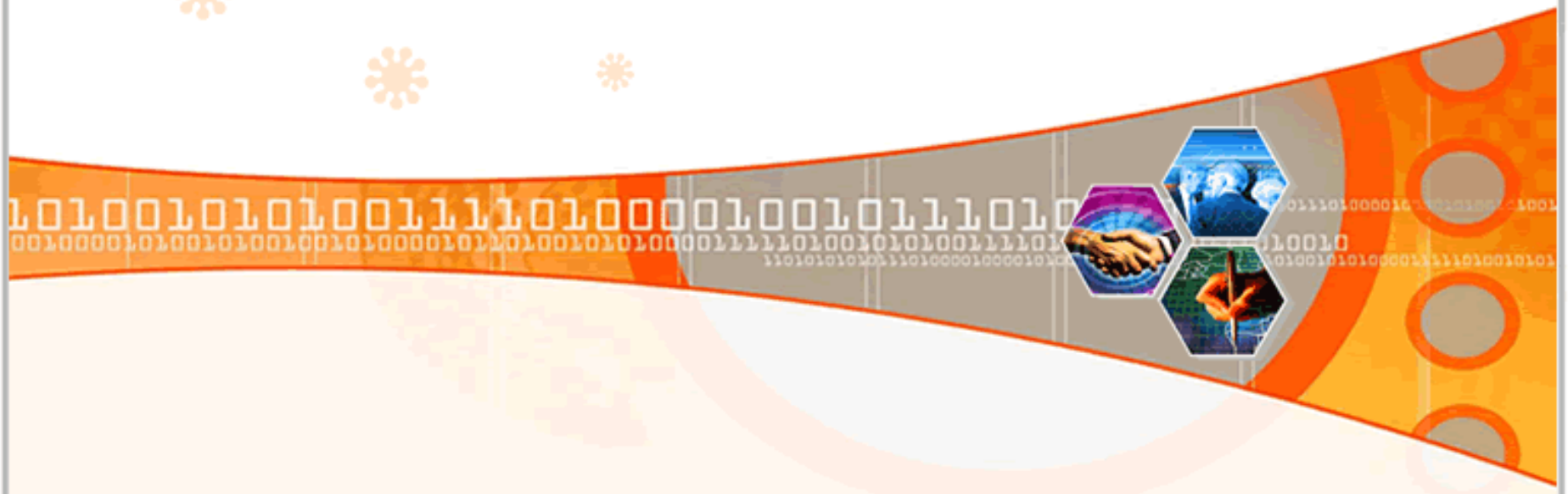
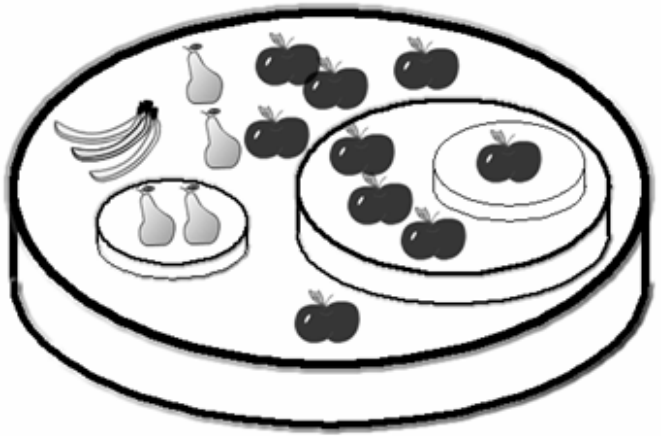
Design Patterns

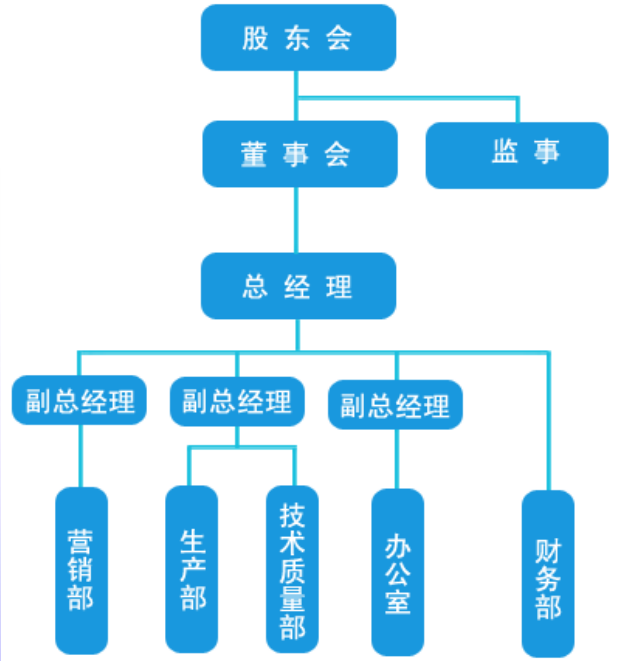
组合模式



大纲

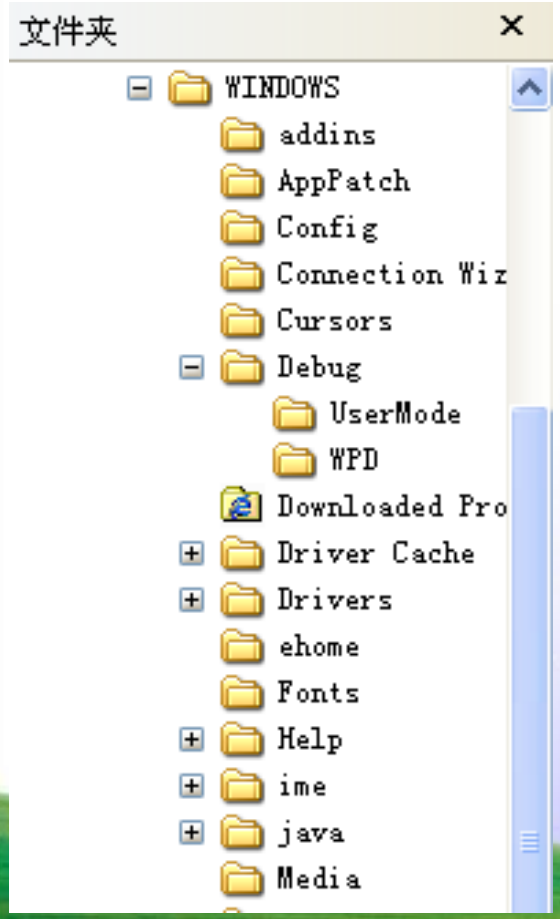


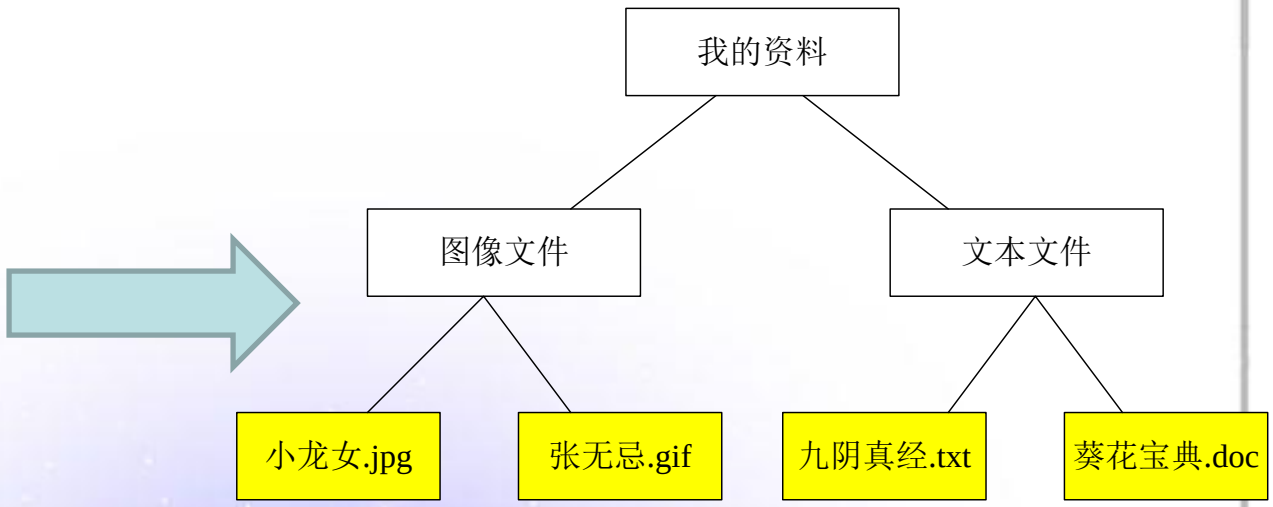
- 组合模式概述
- 组合模式的结构与实现
- 组合模式的应用实例
- 透明组合模式与安全组合模式
- 组合模式的优缺点与适用环境



组合模式概述

- Windows操作系统目录结构





组合模式概述

- 分析
  - ✓树形目录结构中，包含文件和文件夹两类不同的元素
    - 文件夹中可以包含文件，还可以继续包含子文件夹
    - 在文件中不能再包含子文件或者子文件夹
  - ✓文件夹 ←→容器(Container)
  - ✓文件 ← →叶子(Leaf)

组合模式概述

```
if (is 容器对象) {
    //处理容器对象
}
else if (is 叶子对象) {
    //处理叶子对象
}
```

- 分析
  - ✓ 当容器对象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并调用执行，牵一而动百，其中使用了递归调用的机制对整个处理结构。
  - ✓ 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客户端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程序非常复杂。

组合模式概述

- 如何一致地对待容器对象和叶子对象？

组合模式

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，它描述了如何将容器对象和叶子对象进行递归组合，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对象和叶子对象。

组合模式概述

- 组合模式定义

组合模式：组合多个对象形成**树形结构**以表示**具有部分-整体关系的层次结构**。组合模式让客户端可以**统一**对待单个对象和组合对象。

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

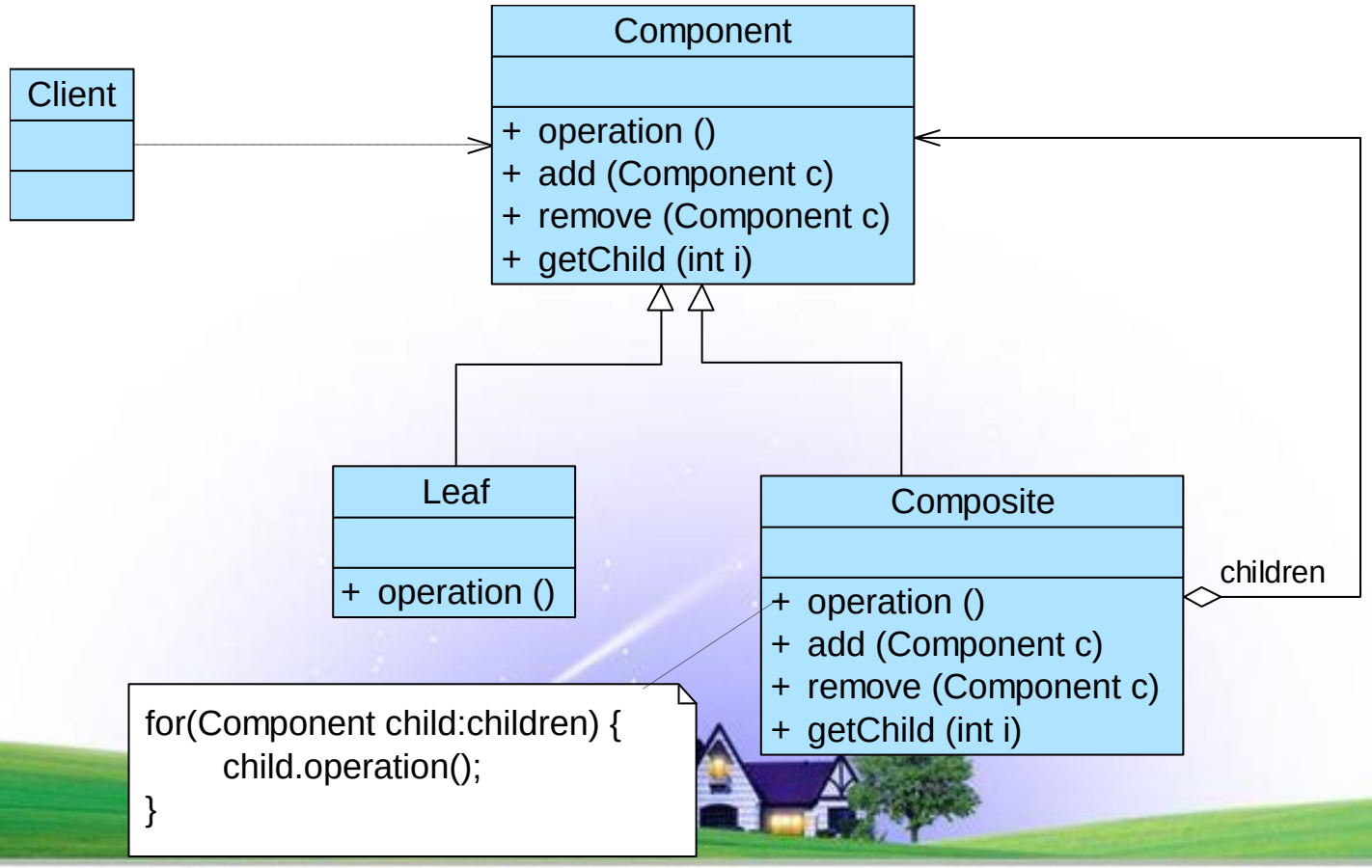
✓对象结构型模式

组合模式概述

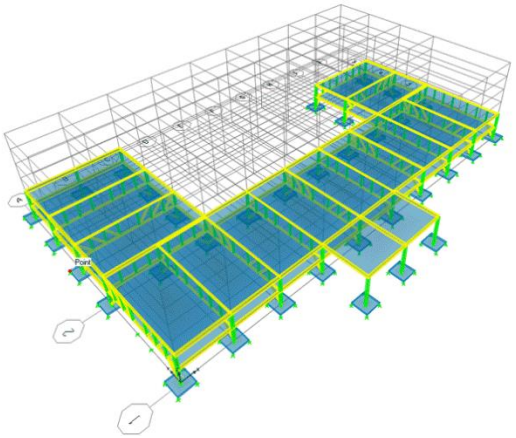
- 组合模式定义
  - ✓ 又称为“部分-整体”(Part-Whole)模式
  - ✓ 将对象组织到树形结构中，可以用来描述整体与部分的关系

组合模式的结构与实现

- 组合模式的结构

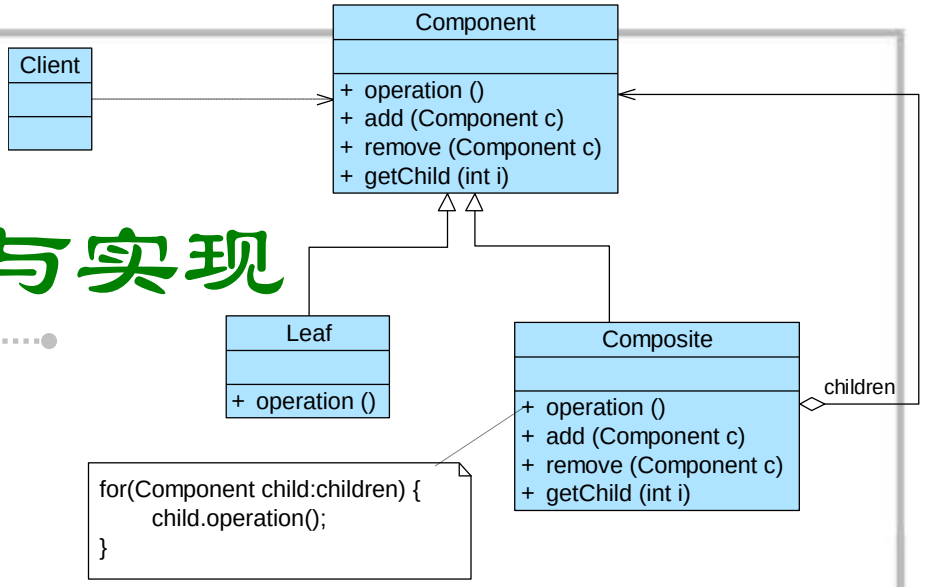


组合模式的结构与实现



- 组合模式的结构
  - ✓组合模式包含以下3个角色:
    - Component（抽象构件）
    - Leaf（叶子构件）
    - Composite（容器构件）

组合模式的结构与实现



- 组合模式的实现
  - ✓抽象构件角色典型代码:

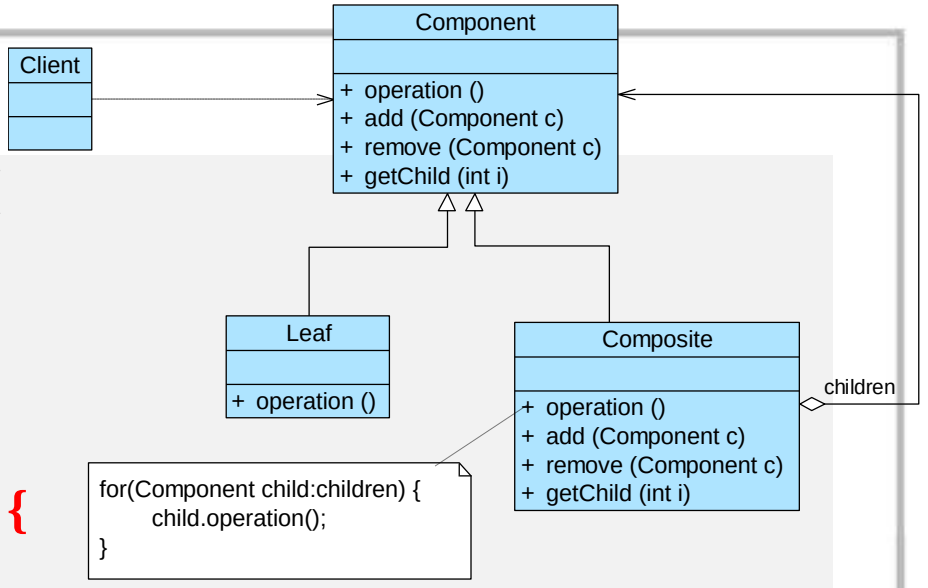
```
public abstract class Component {
    public abstract void add(Component c); //增加成员
    public abstract void remove(Component c); //删除成员
    public abstract Component getChild(int i); //获取成员
    public abstract void operation();  //业务方法
}
```

```
public 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
    }
}
```



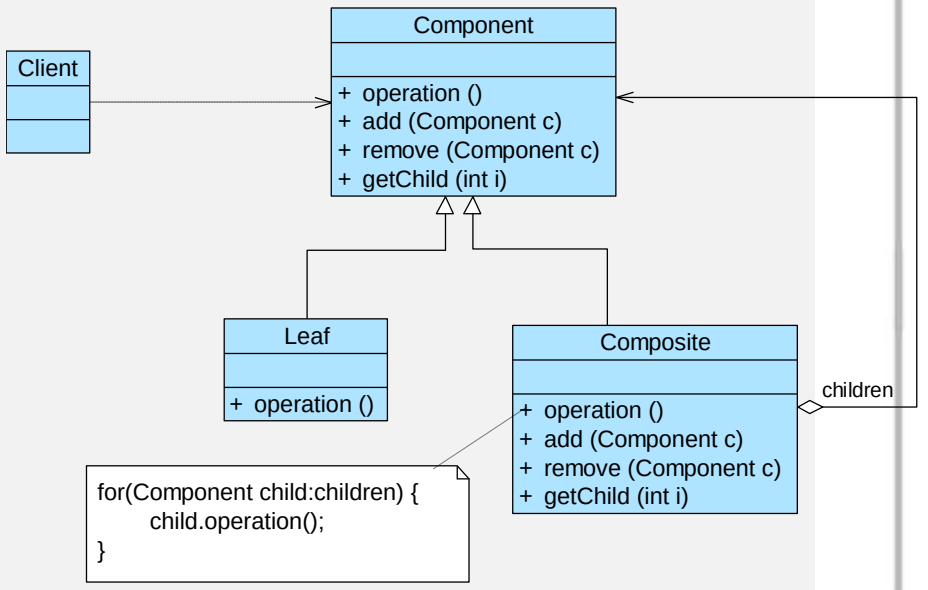
```
public class Composite extends Component {
    private ArrayList<Component> list = new ArrayList<Component>();

    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，将递归调用成员构件的业务方法
        for(Object obj:list) {
            ((Component)obj).operation();
        }
    }
}
```



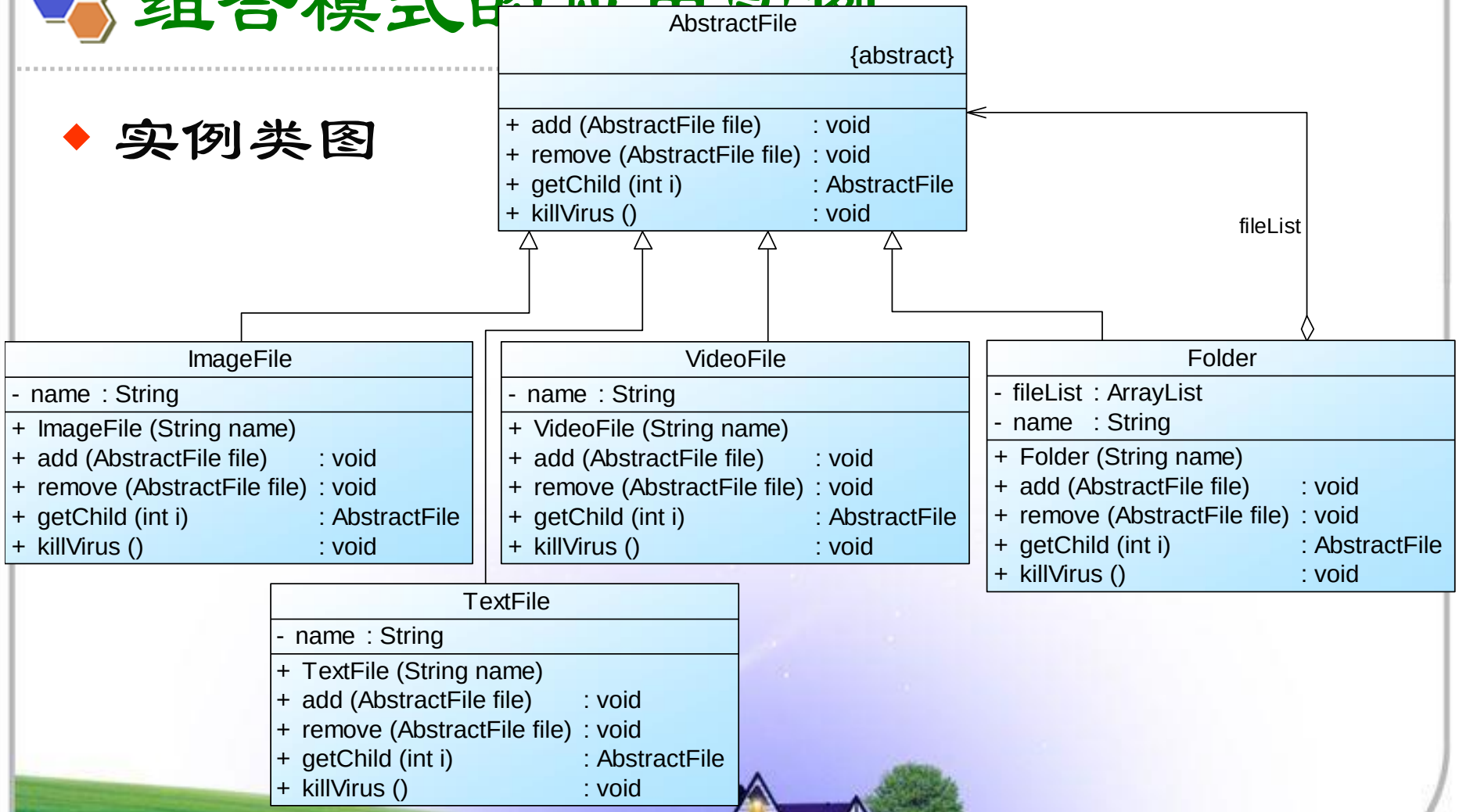
组合模式的应用实例

- 实例说明

某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，也可以对某个指定的文件(File)进行杀毒。该杀毒软件还可以根据各类文件的特点，为不同类型的文件提供不同的杀毒方式，例如图像文件(ImageFile)和文本文件(TextFile)的杀毒方式就有所差异。现使用组合模式来设计该杀毒软件的整体框架。

组合模式的应用实例

- 实例类图



杀毒软件框架设计结构图

组合模式的应用实例

- 实例代码
  - ✓ (1) AbstractFile：抽象文件类，充当抽象构件类
  - ✓ (2) ImageFile：图像文件类，充当叶子构件类
  - ✓ (3) TextFile：文本文件类，充当叶子构件类
  - ✓ (4) VideoFile：视频文件类，充当叶子构件类
  - ✓ (5) Folder：文件夹类，充当容器构件类
  - ✓ (6) Client：客户端测试类

演示.....

Code (designpatterns.composite)

组合模式的应用实例

```
//抽象文件类，充当抽象构件类
public 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，充当叶子构件类
public 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，充当叶子构件类
public 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，充当叶子构件类
public 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");
        }
}
```

```
public class Folder extends AbstractFile {//文件夹类，充当容器构件类
        //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
               for(Object obj : fileList) {//递归调用成员构件的killVirus()方法
                    ((AbstractFile)obj).killVirus();
               }
          }
}
```

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

        folder1.killVirus(); //从“Sunny的资料”结点开始进行杀毒操作
    }
}
```

组合模式的应用实例

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

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

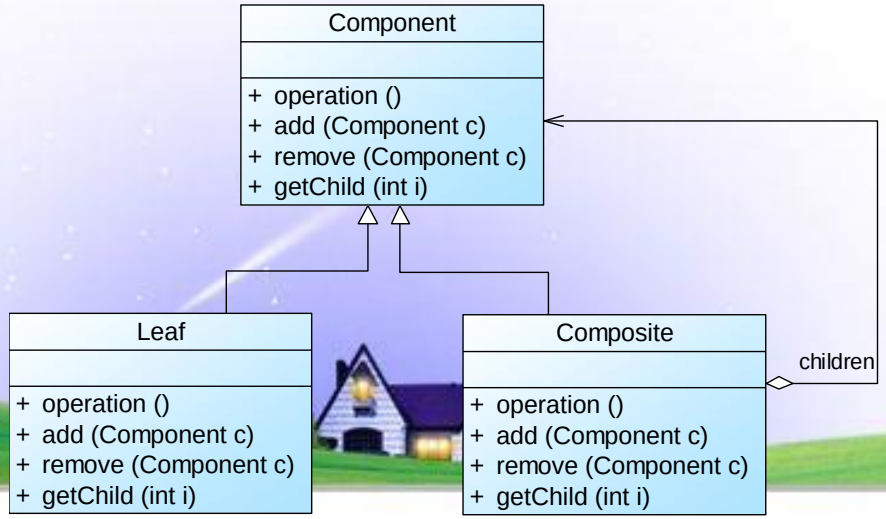
  - ✓ 改为：

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

  - ✓ 在具体实现时，可以创建图形化界面让用户来选择所需操作的根节点，无须修改源代码，符合开闭原则

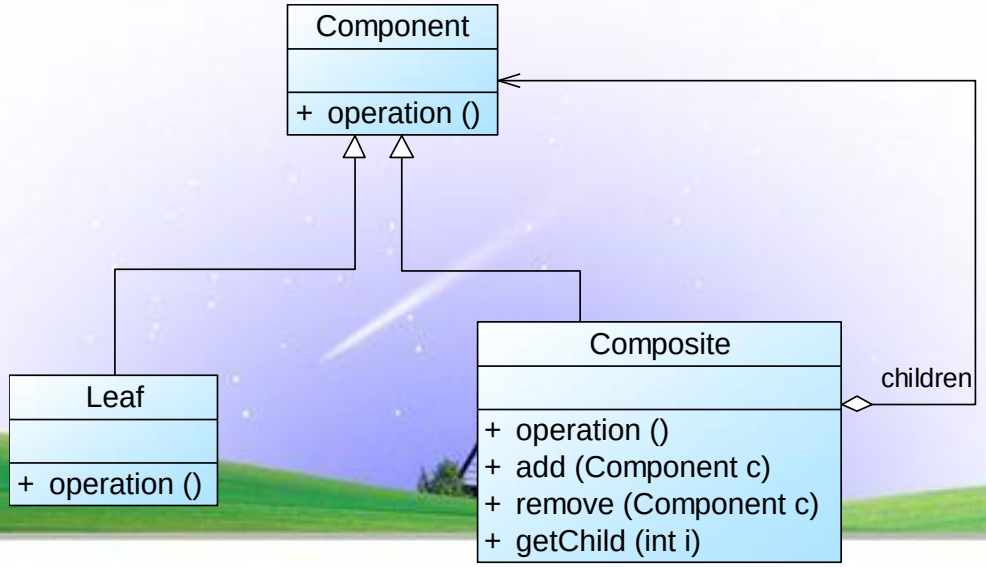
透明组合模式与安全组合模式

- 透明组合模式
  - ✓ 抽象构件Component声明了所有用于管理成员对象的方法，包括add()、remove()，以及getChild()等
  - ✓ 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，客户端可以一致地对待所有的对象
  - ✓ 缺点是不够安全，因为叶子对象和容器对象在本质上是有区别的



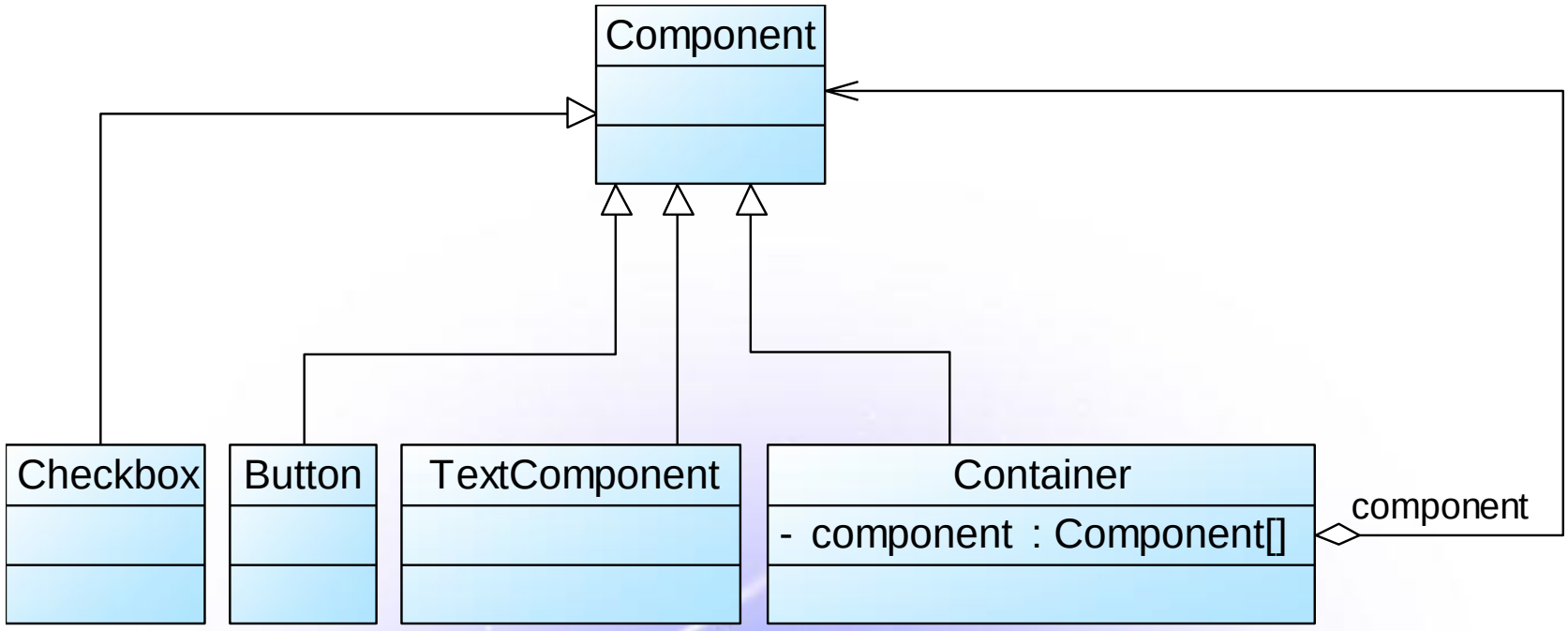
透明组合模式与安全组合模式

- 安全组合模式
  - ✓ 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
  - ✓ 对于叶子对象，客户端不可能调用到这些方法
  - ✓ 缺点是不够透明，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



组合模式实例

- Java AWT中的组件树



组合模式的优缺点与适用环境

- 模式优点
  - ✓ 可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，让客户端忽略了层次的差异，方便对整个层次结构进行控制
  - ✓ 客户端可以一致地使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了客户端代码
  - ✓ 增加新的容器构件和叶子构件都很方便，符合开闭原则
  - ✓ 为树形结构的面向对象实现提供了一种灵活的解决方案

组合模式的优缺点与适用环境

- 模式缺点
  - ✓ 在增加新构件时很难对容器中的构件类型进行限制

组合模式的优缺点与适用环境

- 模式适用环境
  - ✓ 在具有整体和部分的层次结构中，希望通过一种方式忽略整体与部分的差异，客户端可以一致地对待它们
  - ✓ 在一个使用面向对象语言开发的系统中需要处理一个树形结构
  - ✓ 在一个系统中能够分离出叶子对象和容器对象，而且它们的类型不固定，需要增加一些新的类型

END