河北师范大学软件学院
Software College of Hebei Normal University

第五章 组合模式

任课教师：武永亮
wuyongliang@edu2act.org

上节回顾

- 策略模式适用环境
  - 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。
  - 一个系统需要动态地在几种算法中选择一种。
  - 不希望客户端知道复杂的、与算法相关的数据结构，在具体策略类中封装算法和相关的数据结构，提高算法的保密性与安全性。

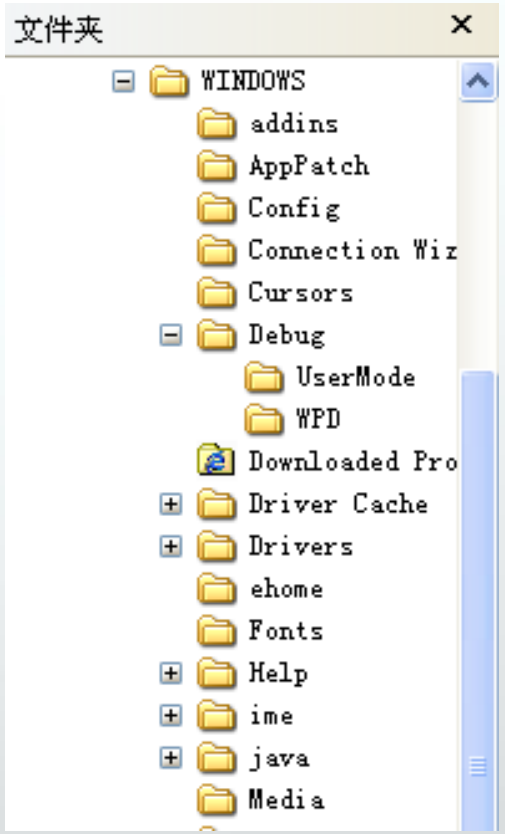
课程内容

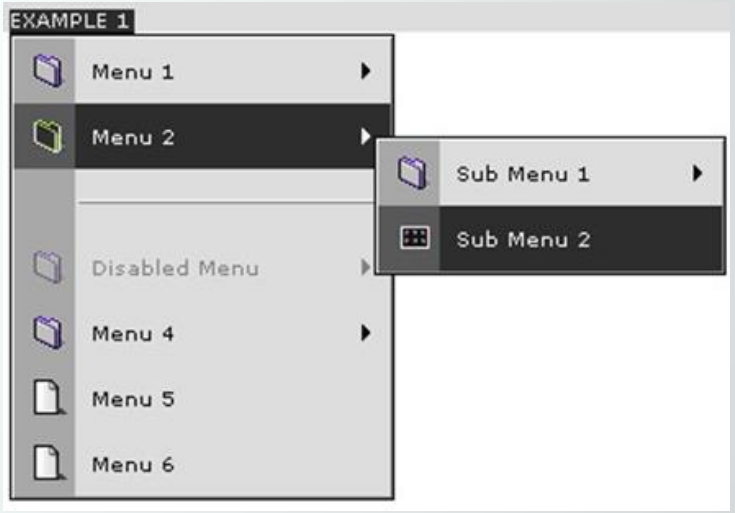
- 环境及问题
- 组合模式详解
- 组合模式实现
- 扩展练习

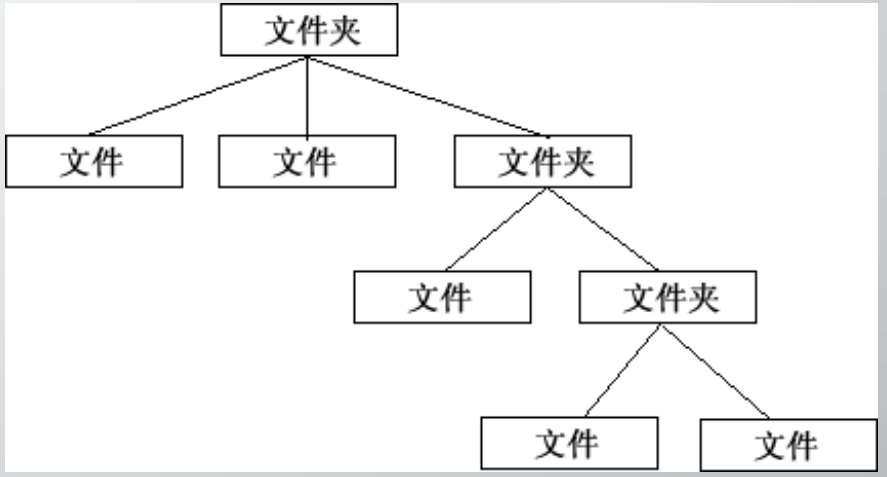
课程内容

- 环境及问题
- 组合模式详解
- 组合模式实现
- 扩展练习

环境


文件夹
WINDOWS
addins
AppPatch
Config
Connection Wiz
Cursors
Debug
UserMode
WPD
Downloaded Pro
Driver Cache
Drivers
ehome
Fonts
Help
ime
java
Media


EXAMPLE 1
Menu 1
Menu 2
Sub Menu 1
Sub Menu 2
Disabled Menu
Menu 4
Menu 5
Menu 6


文件夹
文件
文件
文件夹
文件
文件夹
文件
文件

环境

```
namespace _5
{
    class Doc
    {
        public String name;

        public Doc(String n)
        {
            name = n;
        }

        public void read()
        {
            System.Console.Write("this is doc:\t"+ name+"\n");
        }
    }
}
```

```
namespace _5
{
    class Files
    {
        public String name;
        List<Doc> lc = new List<Doc>();
        List<Files> lf = new List<Files>();


        public Files(String n)
        {
            name = n;
        }

        public void addDoc(Doc item)
        {
            lc.Add(item);
        }

        public void addFiles(Files item)
        {
            lf.Add(item);
        }

        public void read()
        {
            System.Console.Write("This is files:\t" + name+"\n");
            for (int i = 0; i < lc.Count;i++)
            {
                lc[i].read();
            }

            for (int i = 0; i < lf.Count;i++ )
            {
                lf[i].read();
            }
        }
    }
}
```

环境

```
namespace _5
{
    class Program
    {
        static void Main(string[] args)
        {
            Doc c1 = new Doc("c1");
            Doc c2 = new Doc("c2");
            Doc c3 = new Doc("c3");
            Doc c4 = new Doc("c4");
            Doc c5 = new Doc("c5");

            Files f1 = new Files("f1");
            Files f2 = new Files("f2");


            f1.addDoc(c1);
            f1.addDoc(c2);

            f2.addDoc(c3);
            f2.addDoc(c4);
            f2.addDoc(c5);

            f1.addFiles(f2);

            f1.read();

            System.Console.Read();
        }
    }
}
```

问题

- 用户想表示对象的部分-整体层次结构。
- 并希望忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象。

组合模式（Composite ）

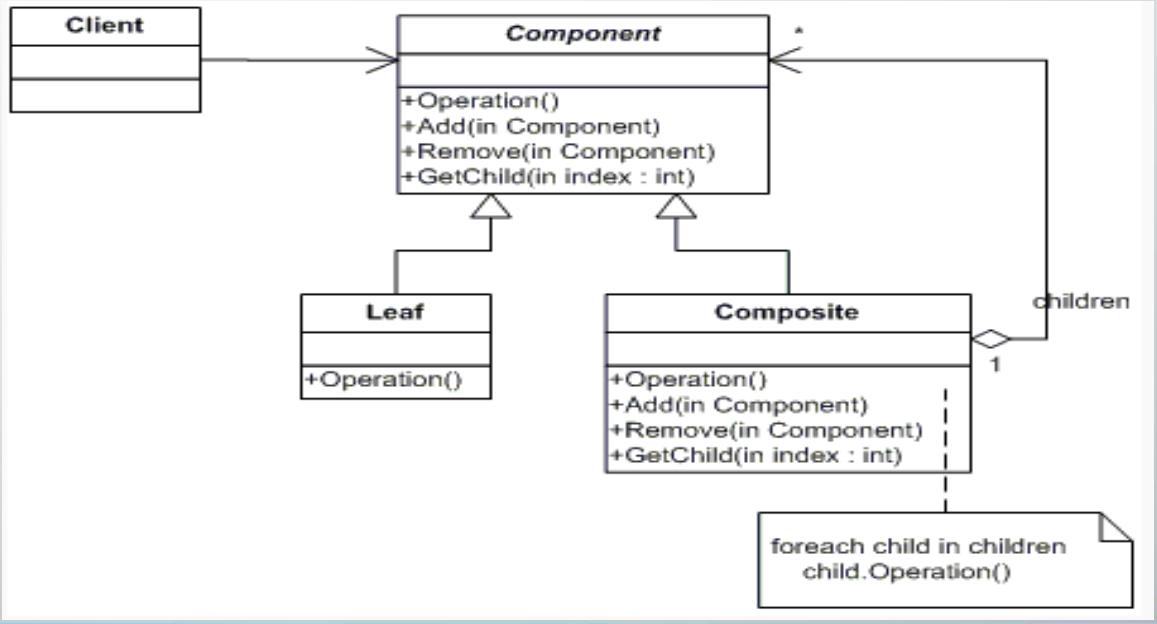
- 描述了如何将容器对象和叶子对象进行递归组合，使得用户在使用时无须对它们进行区分，可以一致地对待容器对象和叶子对象。

课程内容

- 环境及问题
- 组合模式详解
- 组合模式实现
- 扩展练习

组合模式（Composite Pattern ）

- 又叫做“整体-部分模式”。
- 它使树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素,从而使得客户程序与复杂元素的内部结构解耦。



组合模式（Strategy Pattern）

- 组合模式中的有以下的三种角色。
  - 抽象组件类(Component)：组合中的对象声明接口，实现所有类共有接口的行为。声明用于访问和管理Component的子部件的接口。
  - 叶子节点(Leaf)：叶节点对象，叶节点没有子节点。由于叶节点不能增加分支和树叶，所以叶节点的Add和Remove没有实际意义。
  - 组件集合类(Composite)：实现Componet的相关操作，比如Add和Remove操作。其中包含Component的容器，用来存储叶节点集合，有叶节点行为，用来存储叶节点集合。

课程内容

- 环境及问题
- 组合模式详解
- 组合模式实现
- 扩展练习

组合模式实现步骤

- 定义抽象组件接口
- 实现叶子节点类，实现抽象组件类的接口
- 实现组件集合类，实现抽象组件类的接口
- 定义环境类，将叶子节点和组件集合加入根组件集合

组合模式实现步骤一

- 定义抽象组件接口

```
namespace CompositePattern
{
    public abstract class Component
    {
        protected string name;

        public Component(string name)
        {
            this.name = name;
        }

        public abstract void Add(Component c);
        public abstract void Remove(Component c);
        public abstract void Diaplay(int depth);
    }
}
```

组合模式实现步骤二

实现叶子节点

```
namespace CompositePattern
{
    public class Doc : Component
    {
        public Doc(string name): base(name)
        {}
        public override void Add(Component c)
        {
            Console.WriteLine("不能向叶子节点添加子节点");
        }

        public override void Remove(Component c)
        {
            Console.WriteLine("叶子节点没有子节点");
        }

        public override void Diaplay(int depth)
        {
            Console.WriteLine(new string('-', depth) + name);
        }
    }
}
```

组合模式实现步骤三

定义实现组件集合类

```
namespace CompositePattern
{
    public class Files : Component
    {
        List<Component> children = new List<Component>();
        public Files(string name): base(name)
        {
        }
        public override void Add(Component c)
        {
            this.children.Add(c);
        }
        public override void Remove(Component c)
        {
            this.children.Remove(c);
        }
        public override void Diaplay(int depth)
        {
            Console.WriteLine(new String('-', depth) + name);
            foreach (Component component in children)
            {
                component.Diaplay(depth + 2);
            }
        }
    }
}
```

组合模式实现步骤四

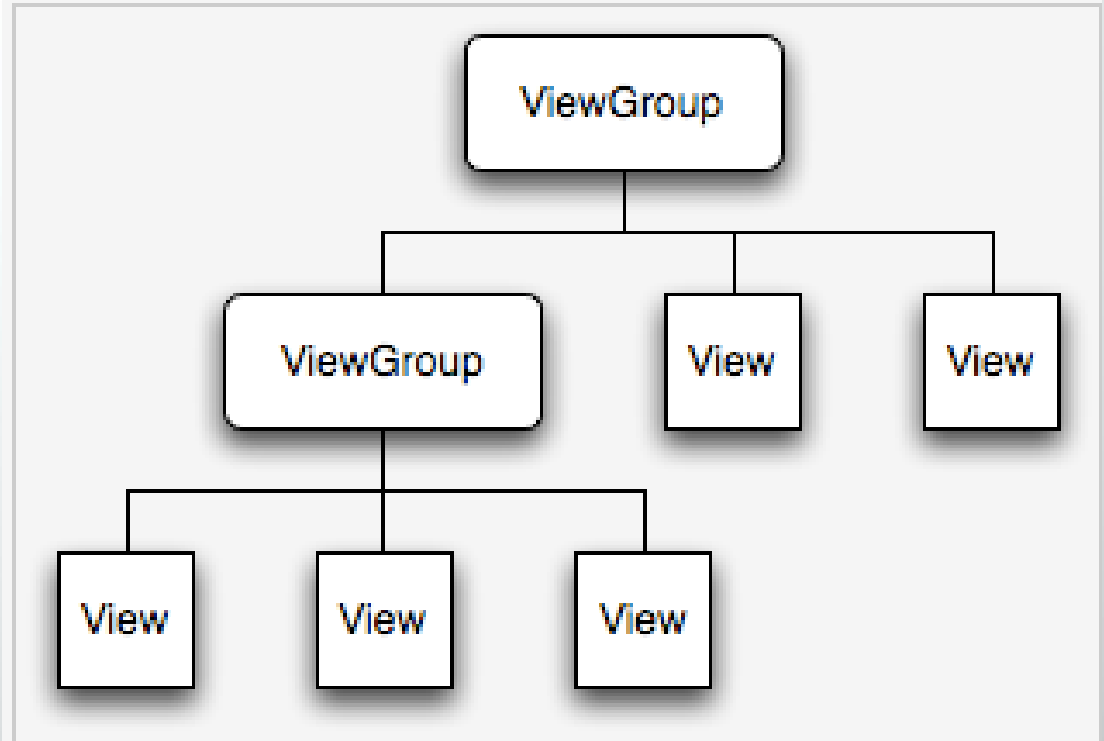
- 模拟用户构建文件树

```
class Program
{
    static void Main(string[] args)
    {
        Files rootFlies = new Files("根目录root");
        rootFlies.Add(new Doc("根目录下的文件A"));
        rootFlies.Add(new Doc("根目录下的文件B"));

        Files comp = new Files("根目录下的文件夹FA");
        comp.Add(new Doc("文件夹FA中的文件AA"));
        comp.Add(new Doc("文件夹FA中的文件AB"));

        rootFlies.Add(comp);

        Files comp2 = new Files("文件夹FA中的文件夹FAX");
        comp2.Add(new Doc("文件夹FAX中的文件AXA"));
        comp2.Add(new Doc("文件夹FAX中的文件AXB"));

        comp.Add(comp2);

        rootFlies.Add(new Doc("根目录下的文件C"));
        Doc leafD = new Doc("根目录下的文件D");
        rootFlies.Add(leafD);
        rootFlies.Remove(leafD);
        rootFlies.Diaplay(1);
```

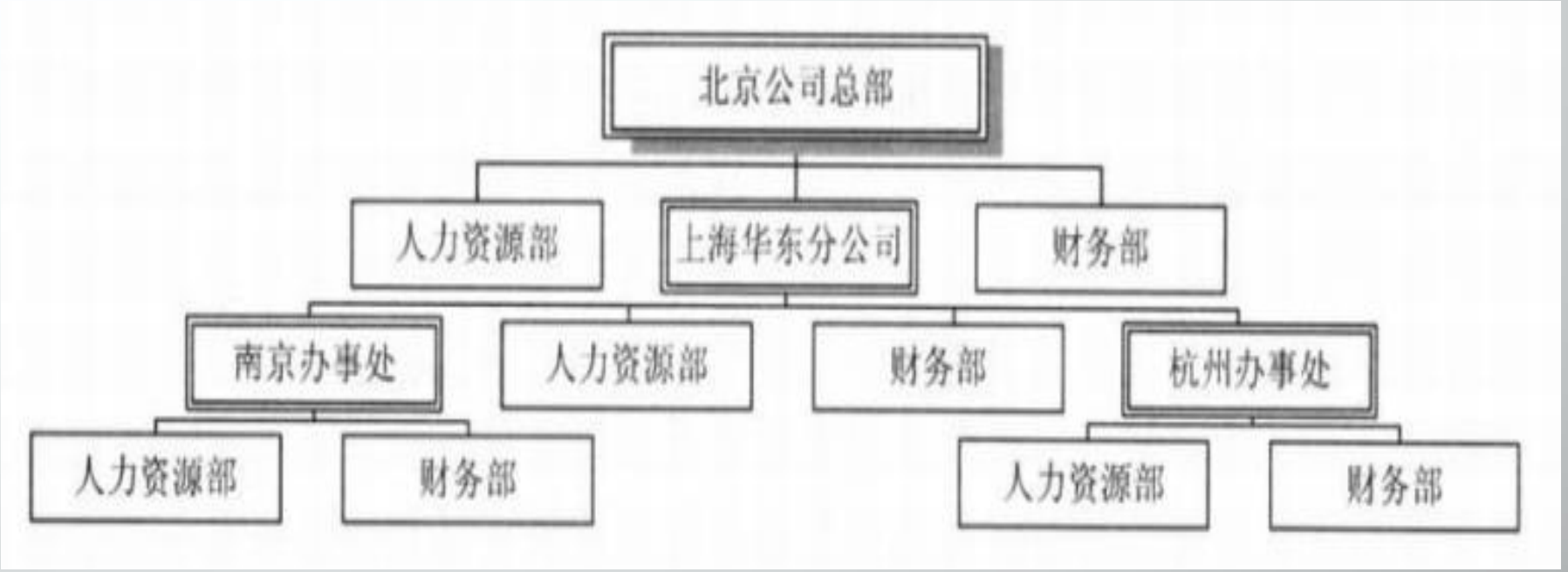
课程内容

- 环境及问题
- 组合模式详解
- 组合模式实现
- 扩展练习

案例练习



案例练习


北京公司总部
人力资源部
上海华东分公司
财务部
南京办事处
人力资源部
财务部
杭州办事处
人力资源部
财务部
人力资源部
财务部

扩展说明

- 组合模式的优点
  - 组合模式以不遵守单一责任原则换取透明性，让Client将组合和叶节点一视同仁。
  - 在实现组合模式时，有很多设计上的折衷。要根据需求平衡透明性和安全性。
  - 有时候系统需要遍历一个树枝构件的子构件很多次，这时候可以把遍历结果缓存起来。

扩展说明

- 组合模式的缺点
  - 使用组合模式后，控制树枝构件的类型不太容易。
  - 用继承的方法来增加新的行为很困难。

小结

- 组合模式适用环境
  - 表示对象的部分-整体层次结构
  - 用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

Thank You，谢谢！