结构型模式（二）

2020年4月20日　14:43

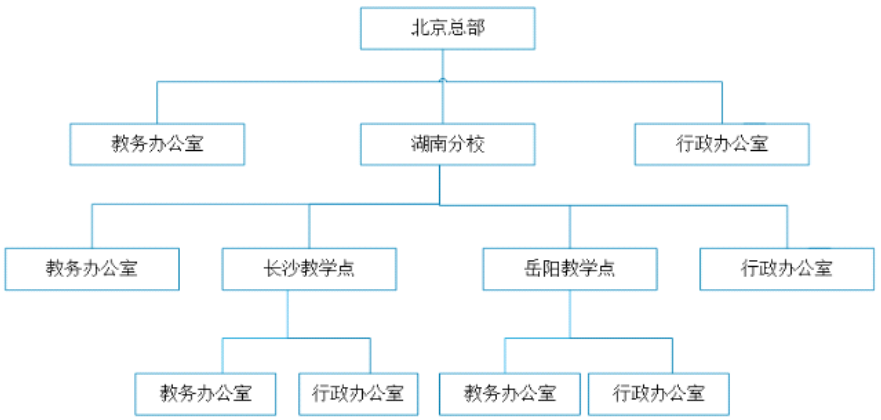
作业要求：

1、试画出组合模式实例的结构图和实现代码，并对模式进行分析。

1、试画出装饰模式实例的结构图和实现代码，并对模式进行分析。

3、下图是某教育机构的组织结构图。

在该教育机构的OA系统中可以给各级办公室下发公文，现采用组合模式设计该机构的结构，绘制相应的类图并编程模拟实现。在客户端 中模拟下发公文。



- 作业与笔记github地址：https://github.com/baobaotql/CCNU_DesignPattern

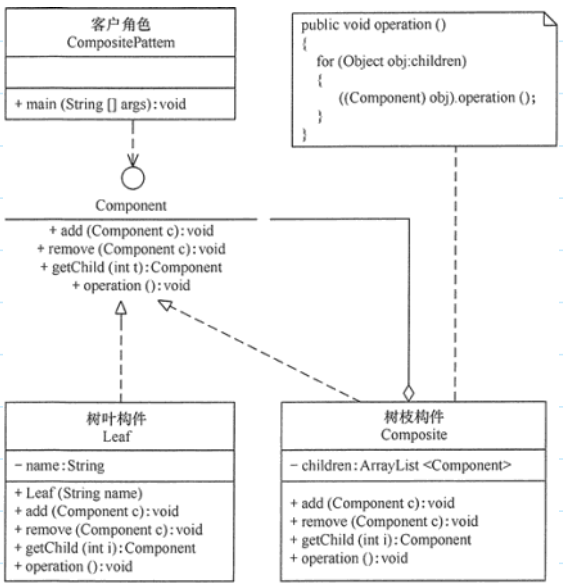
- **Q1:试画出组合模式实例的结构图和实现代码，并对模式进行分析。**

  A1:
  - **组合模式的定义**

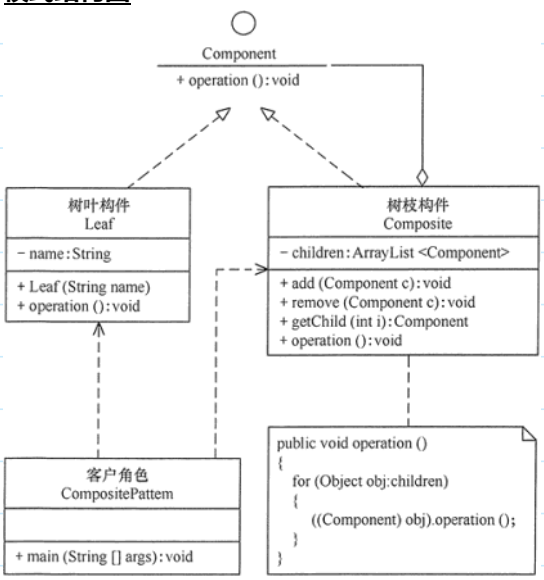
    有时又叫作部分-整体模式，它是一种将对象组合成树状的层次结构的模式，用来表示“部分-整体”的关系，使用户对单个对象和组合对象具有一致的访问性。
  - **优点**
    - 组合模式使得客户端代码可以一致地处理单个对象和组合对象，无须关心自己处理的是单个对象，还是组合对象，这简化了客户端代码
    - 更容易在组合体内加入新的对象，客户端不会因为加入了新的对象而更改源代码，满足“开闭原则”
  - **缺点**
    - 设计较复杂，客户端需要花更多时间理清类之间的层次关系；
    - 不容易限制容器中的构件；
    - 不容易用继承的方法来增加构件的新功能
  - **模式结构**
    - **抽象构件（Component）角色**：它的主要作用是为树叶构件和树枝构件声明公共接口，并实现它们的默认行为。在透明式的组合模式中抽象构件还声明访问和管理子类的接口；在安全式的组合模式中不声明访问和管理子类的接口，管理工作由树枝构件完成。
    - **树叶构件（Leaf）角色**：是组合中的叶节点对象，它没有子节点，用于实现抽象构件角色中 声明的公共接口。
    - **树枝构件（Composite）角色**：是组合中的分支节点对象，它有子节点。它实现了抽象构件角色中声明的接口，它的主要作用是存储和管理子

部件，通常包含 Add()、Remove()、GetChild() 等方法。

- **组合模式分为透明式的组合模式和安全式的组合模式**
  - **透明方式**：在该方式中，由于抽象构件声明了所有子类中的全部方法，所以客户端无须区别树叶对象和树枝对象，对客户端来说是透明的。但其缺点是：树叶构件本来没有 Add()、Remove() 及 GetChild() 方法，却要实现它们（空实现或抛异常），这样会带来一些安全性问题。
    - 模式结构图



  - **安全方式**：在该方式中，将管理子构件的方法移到树枝构件中，抽象构件和树叶构件没有对子对象的管理方法，这样就避免了上一种方式的安全性问题，但由于叶子和分支有不同的接口，客户端在调用时要知道树叶对象和树枝对象的存在，所以失去了透明性。
    - **模式结构图**



  - **模式代码**

```
package composite;

import java.util.ArrayList;

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

```
        {
            Component c0=new Composite();
            Component c1=new Composite();
            Component leaf1=new Leaf("1");
            Component leaf2=new Leaf("2");
            Component leaf3=new Leaf("3");
            c0.add(leaf1);
            c0.add(c1);
            c1.add(leaf2);
            c1.add(leaf3);
            c0.operation();
        }
    }
    //抽象构件
    interface Component
    {
        public void add(Component c);
        public void remove(Component c);
        public Component getChild(int i);
        public void operation();
    }
    //树叶构件
    class Leaf implements Component
    {
        private String name;
        public Leaf(String name)
        {
            this.name=name;
        }
        public void add(Component c){}
        public void remove(Component c){}
        public Component getChild(int i)
        {
            return null;
        }
        public void operation()
        {
            System.out.println("树叶"+name+": 被访问! ");
        }
    }
    //树枝构件
    class Composite implements Component
    {
        private ArrayList<Component> children=new ArrayList<Component>();
        public void add(Component c)
        {
            children.add(c);
        }
        public void remove(Component c)
        {
            children.remove(c);
        }
        public Component getChild(int i)
        {
            return children.get(i);
        }
        public void operation()
```

```
        {
            for(Object obj:children)
            {
                ((Component)obj).operation();
            }
        }
    }
```

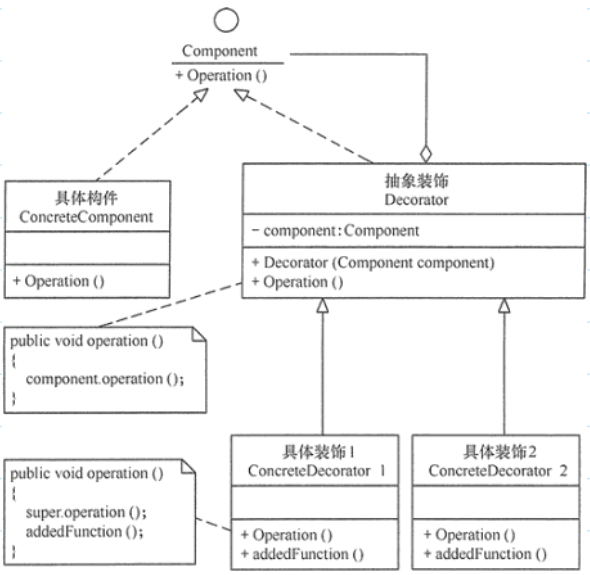
- **Q2：试画出装饰模式实例的结构图和实现代码，并对模式进行分析**

  A2：

  - **装饰模式的定义**

    指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式，它属于对象结构型模式。

  - **优点**
    - 采用装饰模式扩展对象的功能比采用继承方式更加灵活。
    - 可以设计出多个不同的具体装饰类，创造出多个不同行为的组合。
  - **缺点**
    - 装饰模式增加了许多子类，如果过度使用会使程序变得很复杂。
  - **模式结构**
    - **抽象构件（Component）角色**：定义一个抽象接口以规范准备接收附加责任的对象。
    - **具体构件（Concrete　Component）角色**：实现抽象构件，通过装饰角色为其添加一些职责。
    - **抽象装饰（Decorator）角色**：继承抽象构件，并包含具体构件的实例，可以通过其子类扩展具体构件的功能。
    - **具体装饰（ConcreteDecorator）角色**：实现抽象装饰的相关方法，并给具体构件对象添加附加的责任。
    - **模式结构图**



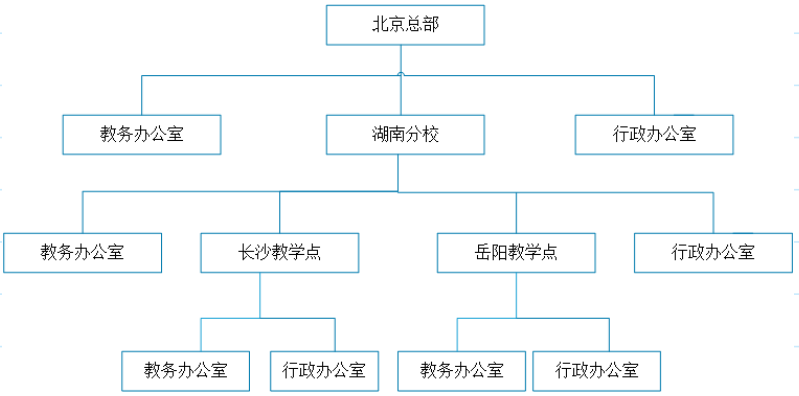
通常情况下，扩展一个类的功能会使用继承方式来实现。但继承具有静态特征，耦合度高，并且随着扩展功能的增多，子类会很膨胀。如果使用组合关系来创建一个包装对象（即装饰对象）来包裹真实对象，并在保持真实对象的类结构不变的前提下，为其提供额外的功能，这就是装饰模式的目标。

模式代码

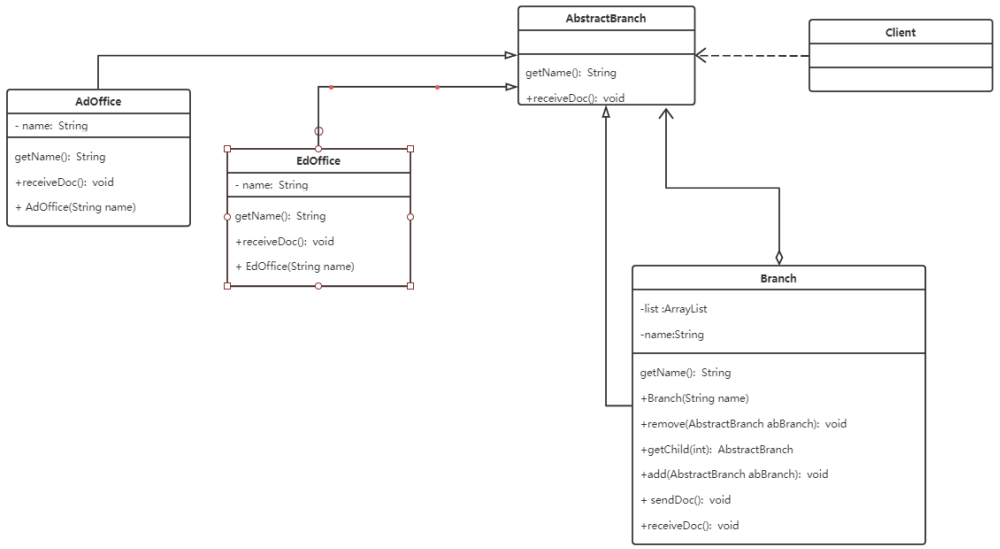
```
package decorator;
```

```
public class DecoratorPattern
{
    public static void main(String[] args)
    {
        Component p=new ConcreteComponent();
        p.operation();
        System.out.println("---------------------------------");
        Component d=new ConcreteDecorator(p);
        d.operation();
    }
}
//抽象构件角色
interface Component
{
    public void operation();
}
//具体构件角色
class ConcreteComponent implements Component
{
    public ConcreteComponent()
    {
        System.out.println("创建具体构件角色");
    }
    public void operation()
    {
        System.out.println("调用具体构件角色的方法
operation()");
    }
}
//抽象装饰角色
class Decorator implements Component
{
    private Component component;
    public Decorator(Component component)
    {
        this.component=component;
    }
    public void operation()
    {
        component.operation();
    }
}
//具体装饰角色
class ConcreteDecorator extends Decorator
{
    public ConcreteDecorator(Component component)
    {
        super(component);
    }
    public void operation()
    {
        super.operation();
        addedFunction();
    }
    public void addedFunction()
    {
        System.out.println("为具体构件角色增加额外的功能
addedFunction()");
    }
}
```

- **Q3:下图是某教育机构的组织结构图。**
  在该教育机构的OA系统中可以给各级办公室下发公文，现采用组合模式设计该机构的结构，绘制相应的类图并编程模拟实现。在客户端 中模拟下发公文。



A3



  - 源代码:

```
/*AbstractBranch.java*/

public abstract class AbstractBranch {
   public abstract void receiveDoc();
   public abstract String getName();
}

AdOffice.java
public class AdOffice extends AbstractBranch {
    private String name;

    public void receiveDoc() {
        System.out.println("行政办公室收到文件");
    }

    public AdOffice(String name){
        this.name = name;
    }
```

```
    public String getName(){
        return name;
    }

}
```

EdOffice.java

```
public class EdOffice extends AbstractBranch {
    private String name;

    public void receiveDoc() {
        System.out.println("教务办公室收到文件");
    }

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

    public String getName(){
        return name;
    }

}
```

Branch.java

```
import java.util.ArrayList;
public class Branch extends AbstractBranch {
   private ArrayList<AbstractBranch> list;
   private String name;

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

   public void receiveDoc() {
       System.out.println(name + "分部收到文件");
   }

   public AbstractBranch getChild(int n) {
      return list.get(n);
   }

   public void add(AbstractBranch abBranch) {
      list.add(abBranch);
   }

   public void remove(AbstractBranch abBranch) {
      list.remove(abBranch);
   }

   public String getName(){
       return name;
```

```
    }

    public void sendDoc(AbstractBranch abBranch) {
        System.out.println(name + "分部发送文件
到" + abBranch.getName());
        abBranch.receiveDoc();
    }

}
```

Client.java

```
public class Client {

    public static void main(String[] args){
        AbstractBranch ab1, ab2;
        Branch b1, b2, b3, b4;
        ab1 = new AdOffice("行政办公室");
        ab2 = new EdOffice("教务办公室");
        b1 = new Branch("北京");
        b2 = new Branch("湖南");
        b3 = new Branch("长沙");
        b4 = new Branch("岳阳");
        b1.sendDoc(b2);
        b2.sendDoc(b3);
        b2.sendDoc(b4);
        b4.sendDoc(ab1);
        b3.sendDoc(ab2);

    }
}
```