南京大学

设计模式–结构型模式(二)

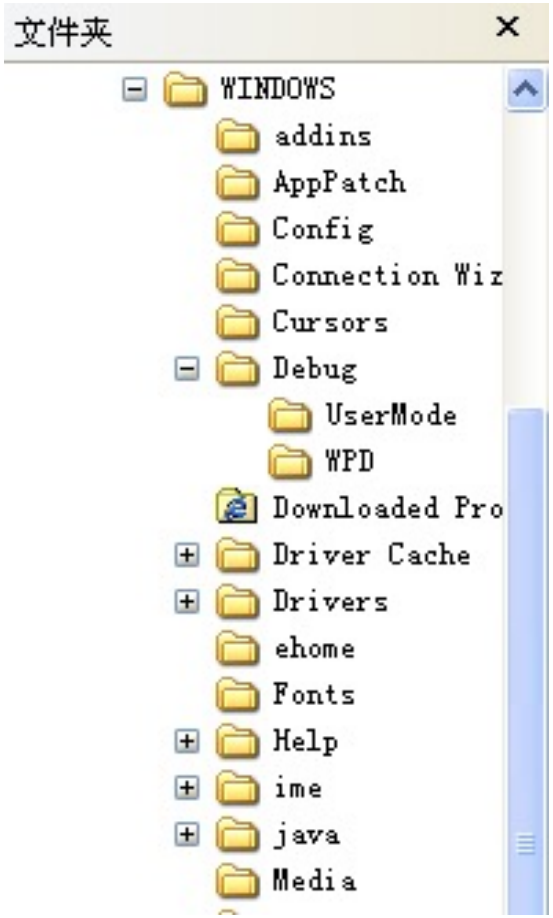
Design Pattern-Structural Pattern (2)

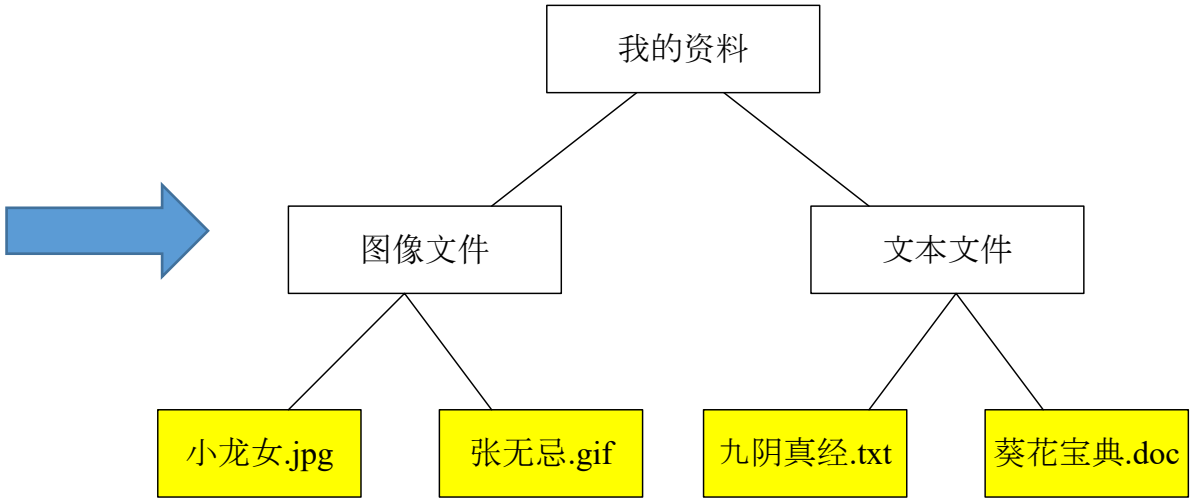
结构型模式概述

模式名称	定 义	学习难度	使用频率
适配器模式 (Adapter Pattern)	将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作。	★★☆☆☆	★★★★☆
桥接模式 (Bridge Pattern)	将抽象部分与它的实现部分解耦，使得两者都能够独立变化。	★★★☆☆	★★★☆☆
组合模式 (Composite Pattern)	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。	★★★☆☆	★★★★☆
装饰模式 (Decorator Pattern)	动态地给一个对象增加一些额外的职责。就扩展功能而言，装饰模式提供了一种比使用子类更加灵活的替代方案。	★★★☆☆	★★★☆☆
外观模式 (Facade Pattern)	为子系统中的一组接口提供一个统一的入口。外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。	★☆☆☆☆	★★★★★
享元模式 (Flyweight Pattern)	运用共享技术有效地支持大量细粒度对象的复用。	★★★★☆	★☆☆☆☆
代理模式 (Proxy Pattern)	给某一个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。	★★★☆☆	★★★★☆

组合模式概述

- Windows操作系统目录结构





组合模式概述

- 分析
  - 在树形目录结构中，包含文件和文件夹两类不同的元素
    - 在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

- 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，
  - 使用这
  - 处理它
  - 非常复杂

```
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
}
```

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**.

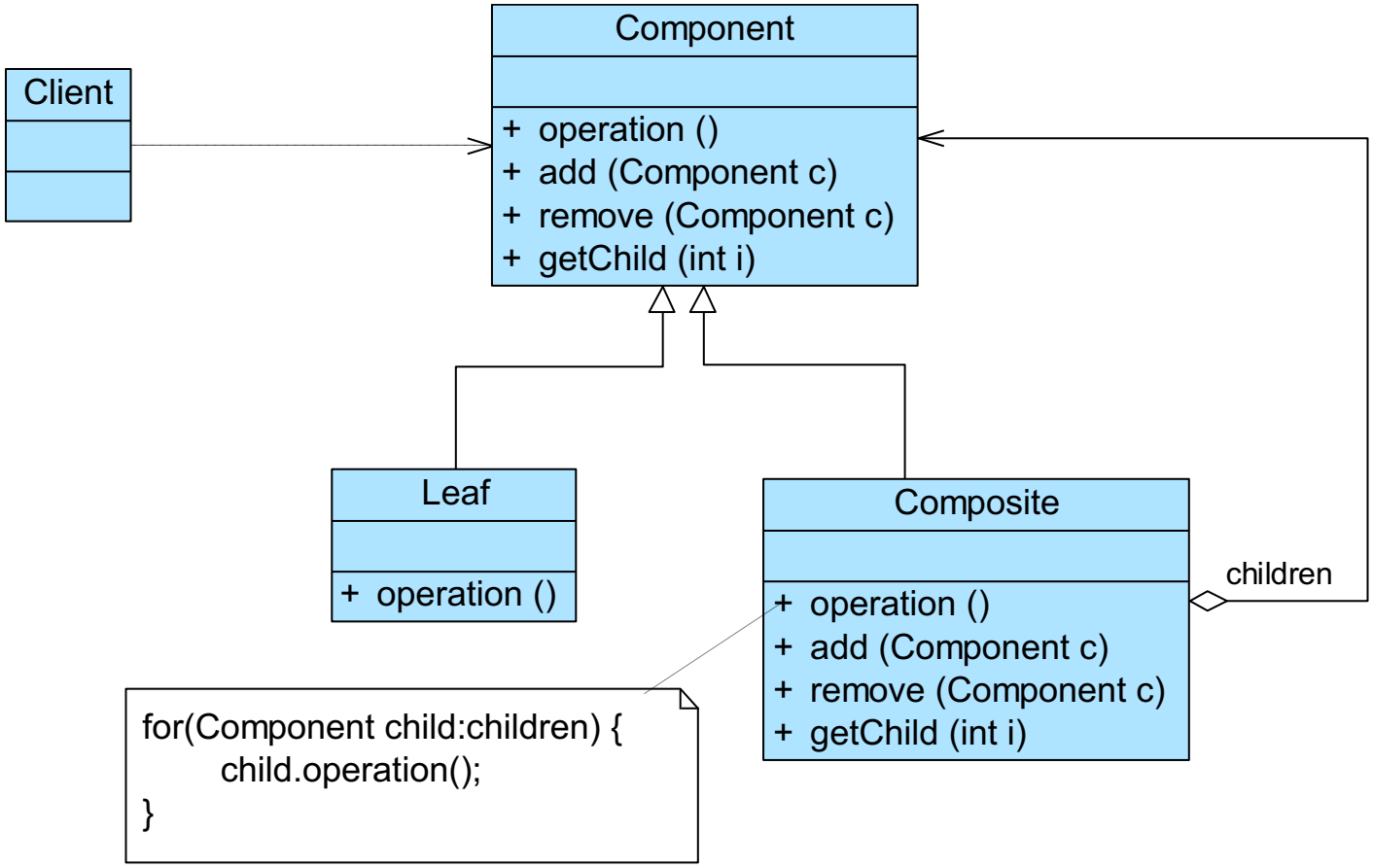
- 对象结构型模式

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
}
```

组合模式的结构与实现

```
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
    }
}
```

组合模式的结构与实现

```
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();
        }
    }
}
```

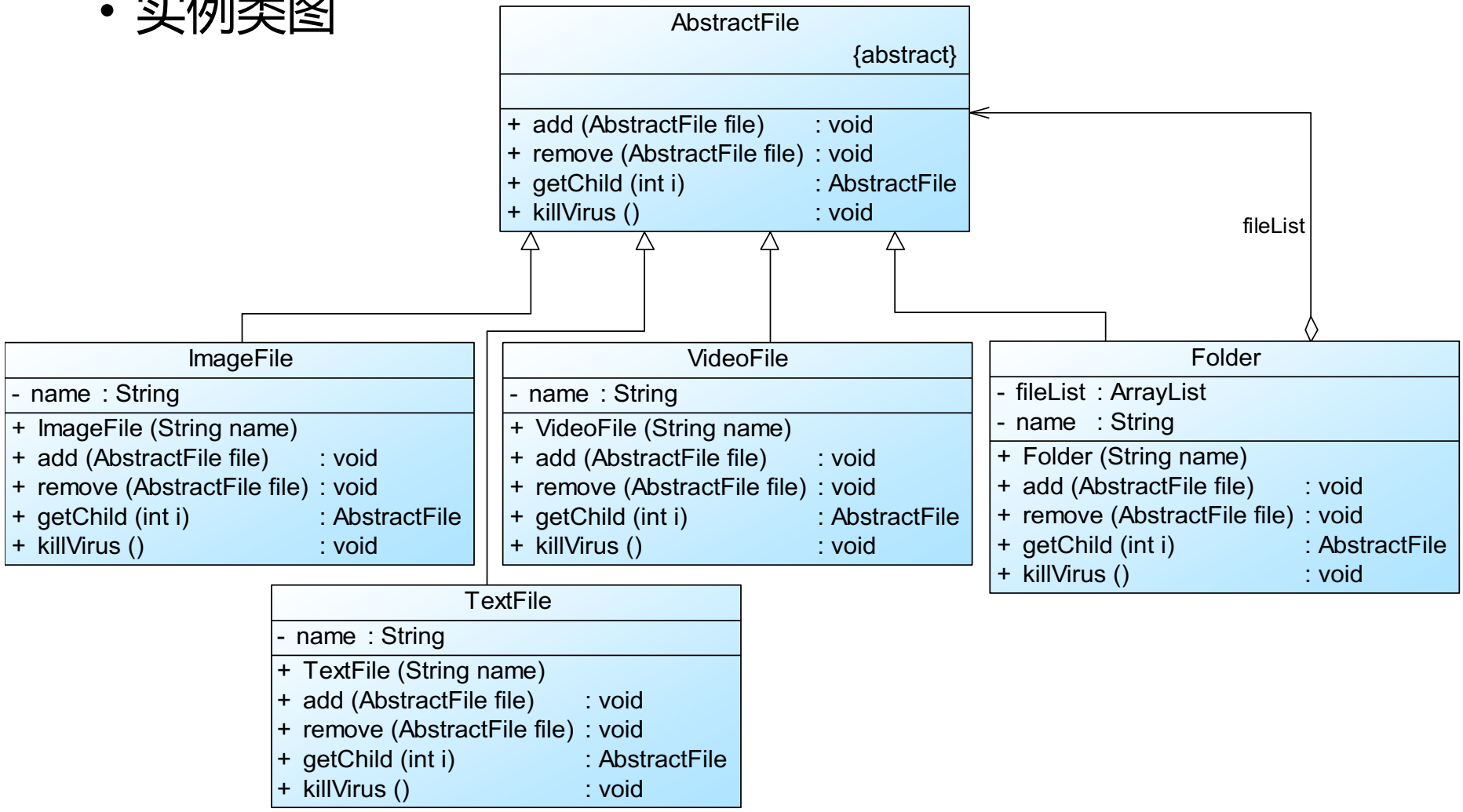
组合模式的应用实例

- 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

- 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();
```

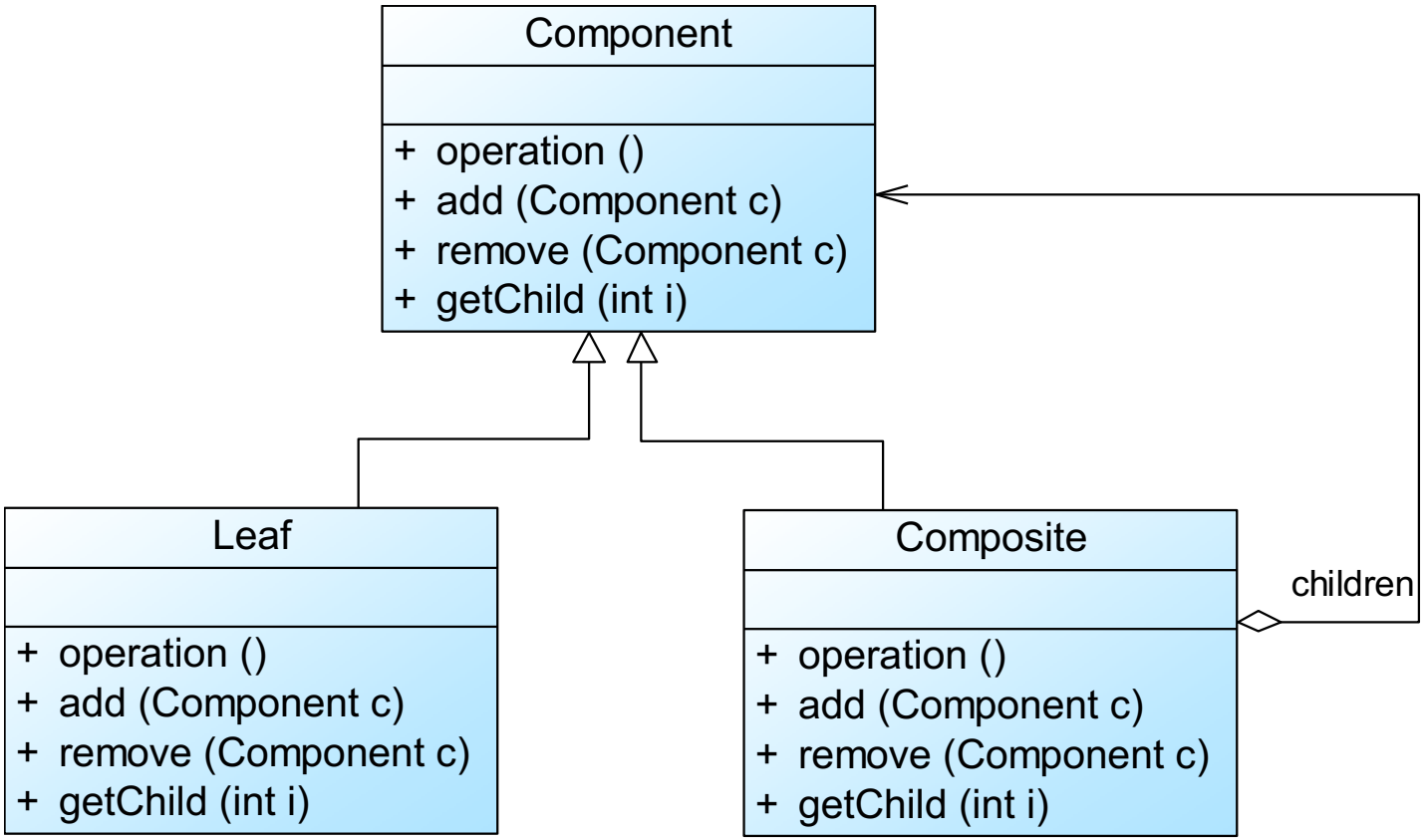
  - 在具体实现时，可以创建图形化界面让用户来选择所需操作的根节点，无须修改源代码，符合开闭原则

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
  - 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

- 透明组合模式

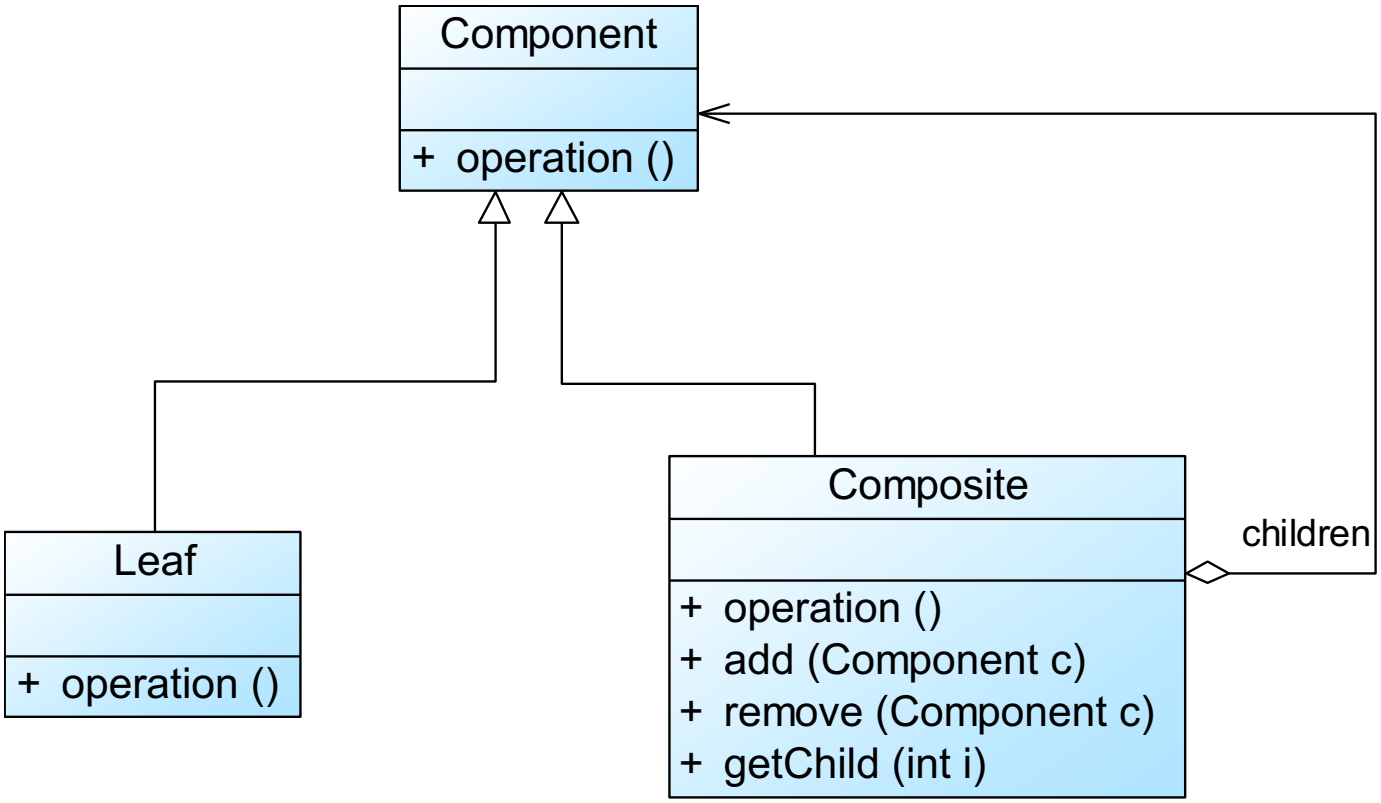


透明组合模式与安全组合模式

- 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

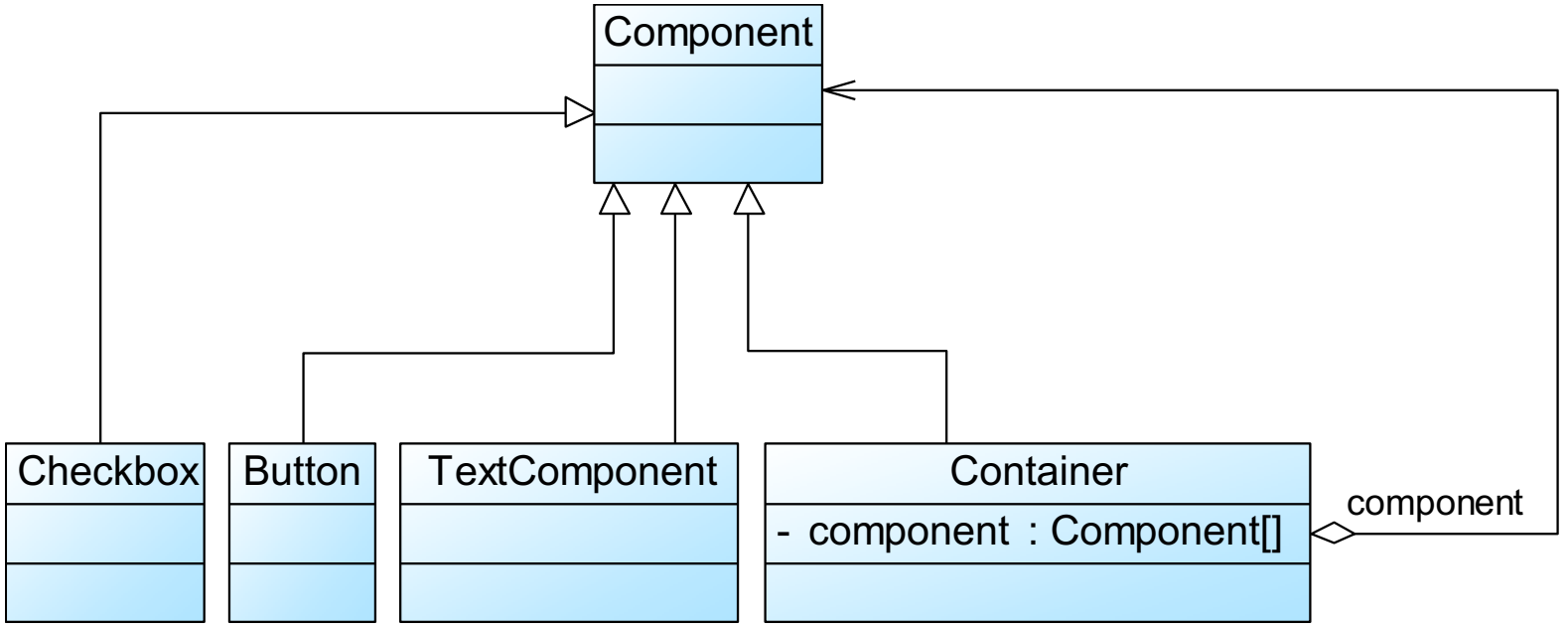
透明组合模式与安全组合模式

- 安全组合模式



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

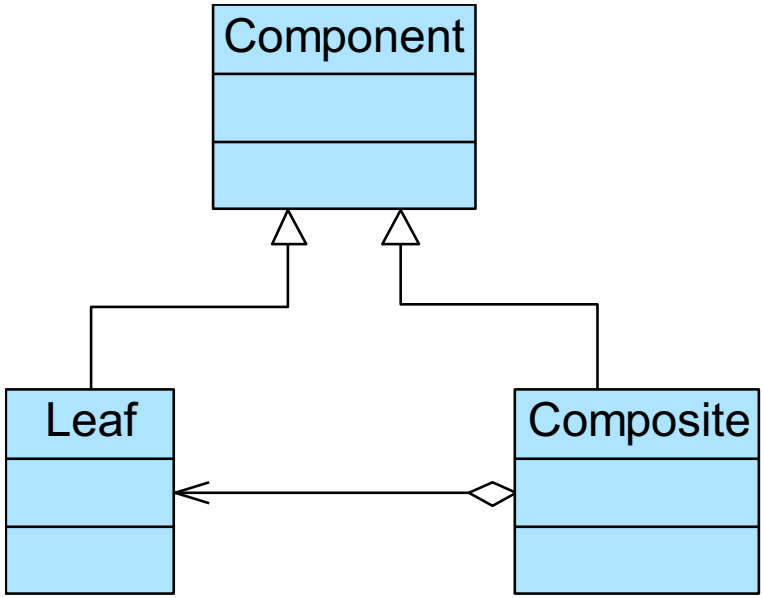
- 模式缺点
  - 在增加新构件时很难对容器中的构件类型进行限制

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

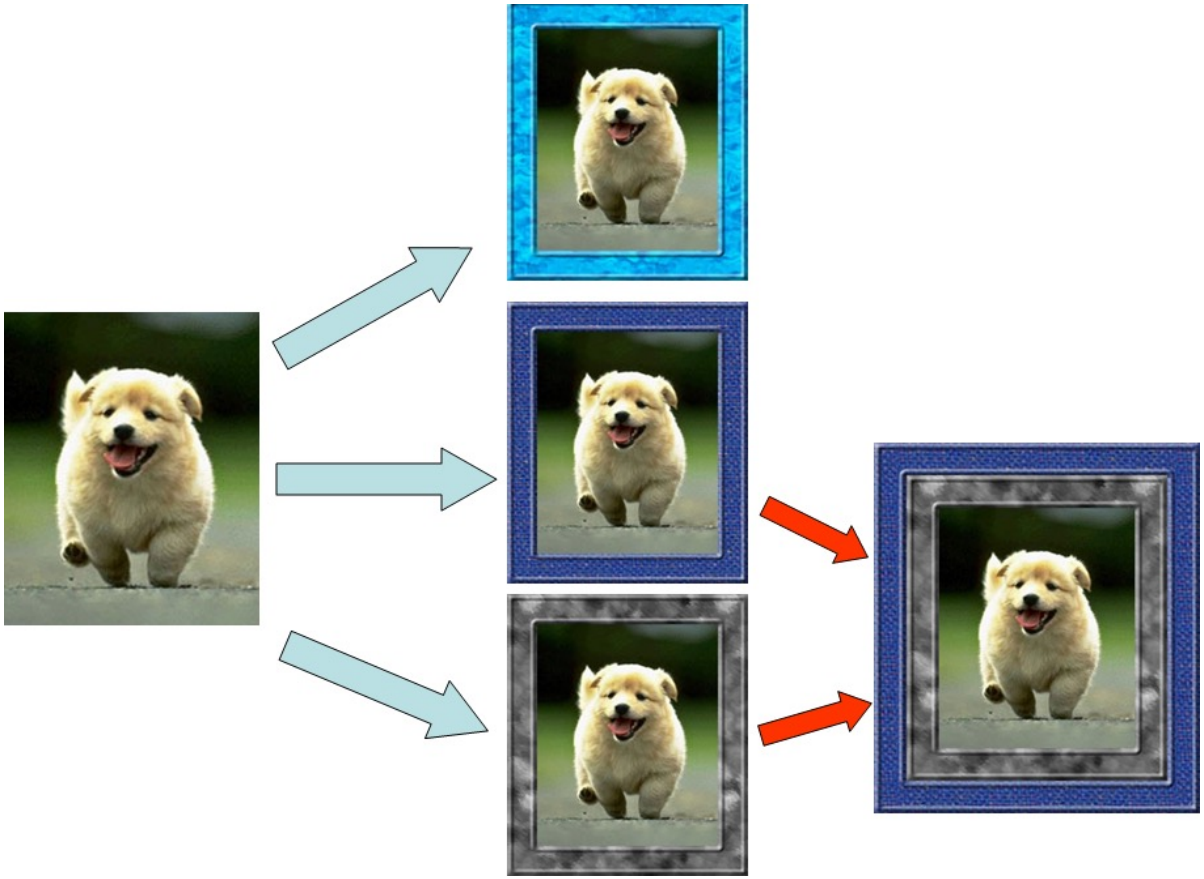
思考

- 在组合模式的结构图中，如果聚合关联关系不是从Composite到Component的，而是从Composite到Leaf，如下图所示，会产生怎样的结果？



装饰模式概述

- 现实生活中的“装饰”实例



装饰模式概述

- 装饰模式分析
  - 可以在不改变一个对象本身功能的基础上给对象增加额外的新行为
  - 是一种用于替代继承的技术，它通过一种无须定义子类的方式给对象动态增加职责，使用对象之间的关联关系取代类之间的继承关系
  - 引入了装饰类，在装饰类中既可以调用待装饰的原有类的方法，还可以增加新的方法，以扩展原有类的功能

装饰模式概述

- 装饰模式的定义

装饰模式：动态地给一个对象增加一些额外的职责。就扩展功能而言，装饰模式提供了一种比使用子类更加灵活的替代方案。

Decorator Pattern: Attach additional responsibilities to an object **dynamically**. Decorators provide **a flexible alternative to subclassing** for extending functionality.

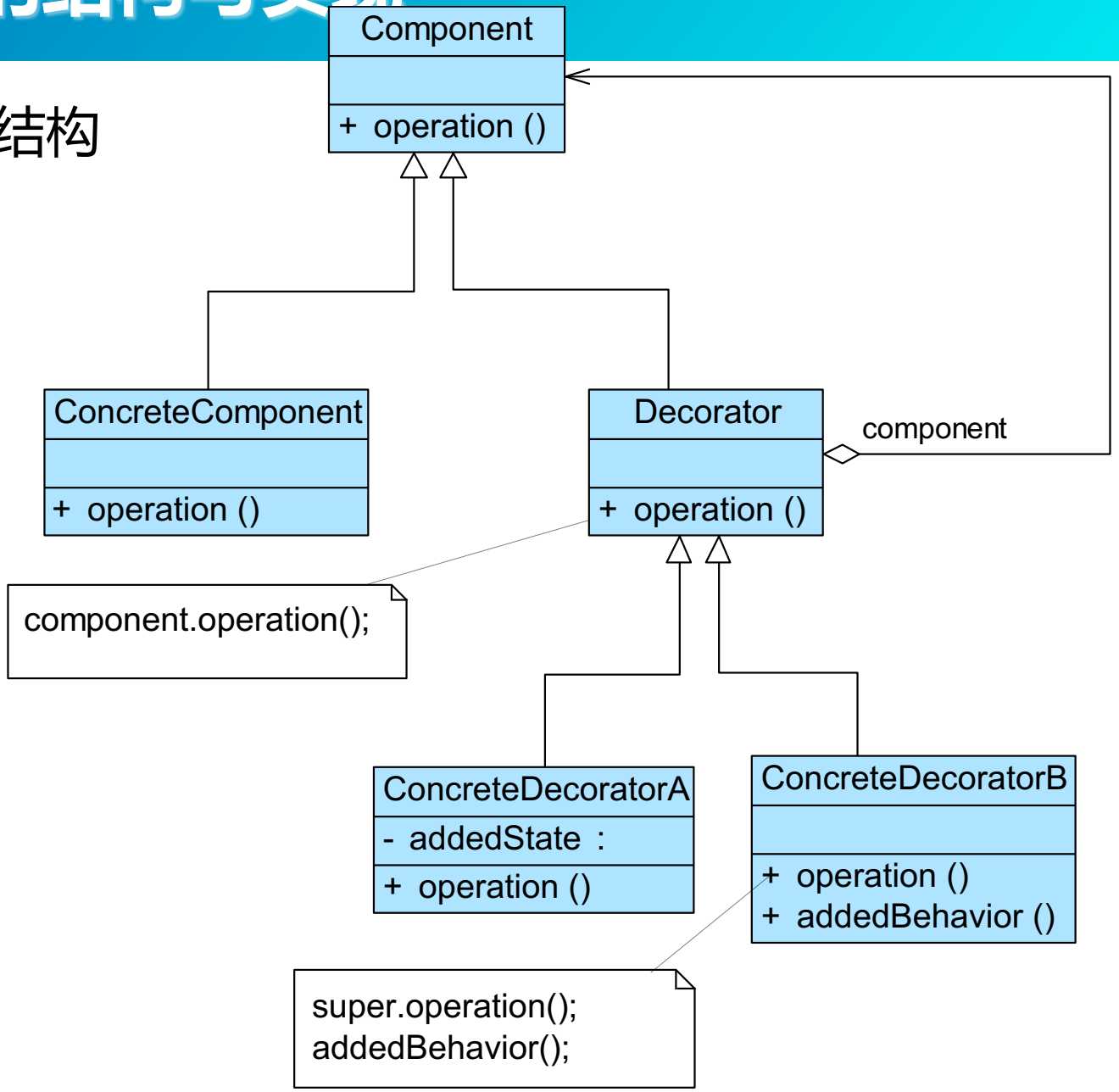
- 对象结构型模式

装饰模式概述

- 装饰模式的定义
  - 以对客户透明的方式动态地给一个对象附加上更多的责任
  - 可以在不需要创建更多子类的情况下，让对象的功能得以扩展

装饰模式的结构与实现

- 装饰模式的结构



装饰模式的结构与实现

- 装饰模式的结构
  - 装饰模式包含以下4个角色：
    - Component（抽象构件）
    - ConcreteComponent（具体构件）
    - Decorator（抽象装饰类）
    - ConcreteDecorator（具体装饰类）

装饰模式的结构与实现

- 装饰模式的实现
  - 抽象构件类典型代码：

```
public abstract class Component {
    public abstract void operation();
}
```

装饰模式的结构与实现

- 装饰模式的实现
  - 具体构件类典型代码：

```
public class ConcreteComponent  extends Component {
    public void operation() {
        //实现基本功能
    }
}
```

装饰模式的结构与实现

- 装饰模式的实现
  - 抽象装饰类典型代码：

```
public class Decorator extends Component {
    private Component component; //维持一个对抽象构件对象的引用

    //注入一个抽象构件类型的对象
    public Decorator(Component component) {
        this.component=component;
    }

    public void operation() {
        component.operation();  //调用原有业务方法
    }
}
```

装饰模式的结构与实现

- 装饰模式的实现
  - 具体装饰类典型代码：

```
public class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component component) {
        super(component);
    }

    public void operation() {
        super.operation(); //调用原有业务方法
        addedBehavior(); //调用新增业务方法
    }

    //新增业务方法
    public void addedBehavior() {
        ……
    }
}
```

装饰模式的应用实例

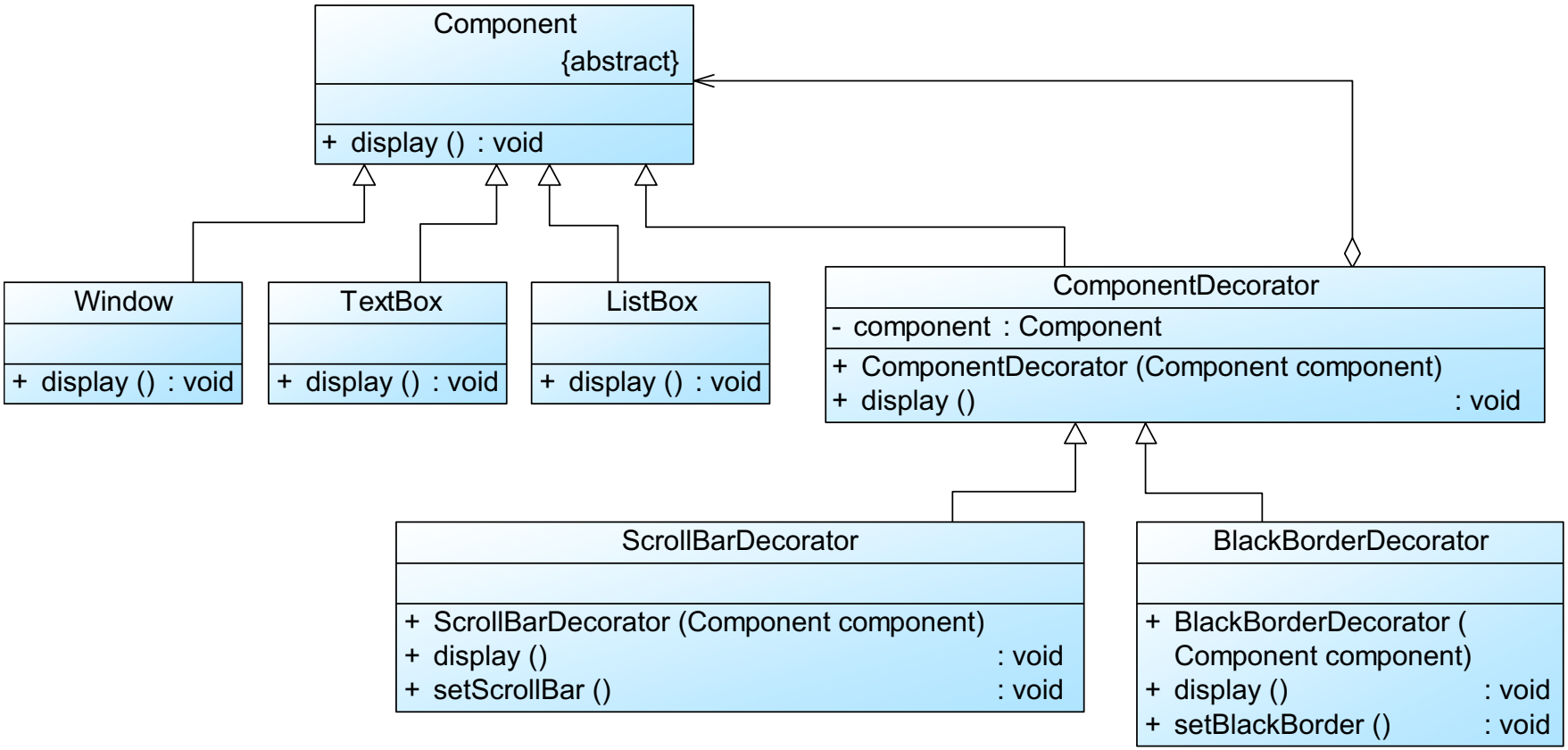
- 实例说明

某软件公司基于面向对象技术开发了一套图形界面构件库——VisualComponent，该构件库提供了大量基本构件，如窗体、文本框、列表框等，由于在使用该构件库时，用户经常要求定制一些特殊的显示效果，如**带滚动条的窗体、带黑色边框的文本框、既带滚动条又带黑色边框的列表框**等等，因此经常需要对该构件库进行扩展以增强其功能。

现使用装饰模式来设计该图形界面构件库。

装饰模式的应用实例

- 实例类图



图形界面构件库结构图

装饰模式的应用实例

- 实例代码
  - (1) Component：抽象界面构件类，充当抽象构件类
  - (2) Window：窗体类，充当具体构件类
  - (3) TextBox：文本框类，充当具体构件类
  - (4) ListBox：列表框类，充当具体构件类
  - (5) ComponentDecorator：构件装饰类，充当抽象装饰类
  - (6) ScrollBarDecorator：滚动条装饰类，充当具体装饰类
  - (7) BlackBorderDecorator：黑色边框装饰类，充当具体装饰类
  - (8) Client：客户端测试类

演示……

Code (designpatterns.decorator)

装饰模式的应用实例

- 结果及分析
  - 实现多次装饰

```
package designpatterns.decorator;

public class Client {
    public static void main(String args[]) {
        为构件增加黑色边框！
        为构件增加滚动条！
        显示窗体！
        componentBB = new BlackBorderDecorator(componentSB);
        componentBB.display();
    }
}
```

透明装饰模式与半透明装饰模式

- 透明装饰模式
  - 透明(Transparent)装饰模式：要求客户端完全针对抽象编程，装饰模式的透明性要求客户端程序不应该将对象声明为具体构件类型或具体装饰类型，而应该全部声明为抽象构件类型
  - 对于客户端而言，具体构件对象和具体装饰对象没有任何区别

透明装饰模式与半透明装饰模式

- 透明装饰模式
  - 可以让客户端透明地使用装饰之前的对象和装饰之后的对象，无须关心它们的区别
  - 可以对一个已装饰过的对象进行多次装饰，得到更为

```
……
Component component_o,component_d1,component_d2; //全部使用抽象构件定义
component_o = new ConcreteComponent();
component_d1 = new ConcreteDecorator1(component_o);
component_d2 = new ConcreteDecorator2(component_d1);
component_d2.operation();
//无法单独调用component_d2的addedBehavior()方法
……
```

透明装饰模式与半透明装饰模式

- 半透明装饰模式
  - 半透明(Semi-transparent)装饰模式：用具体装饰类型来定义装饰之后的对象，而具体构件使用抽象构件类型来定义
  - 对于客户端而言，具体构件类型无须关心，是透明的；但是具体装饰类型必须指定，这是不透明的

透明装饰模式与半透明装饰模式

- 半透明装饰模式
  - 可以给系统带来更多的灵活性，设计相对简单，使用起来也非常方便
  - 客户端使用具体装饰类型来定义装饰后的对象，因此可以单独调用addedBehavior()方法

```
……
Component component_o; //使用抽象构件类型定义
component_o = new ConcreteComponent();
component_o.operation();
ConcreteDecorator component_d; //使用具体装饰类型定义
component_d = new ConcreteDecorator(component_o);
component_d.operation();
component_d.addedBehavior(); //单独调用新增业务方法
……
```

装饰模式的优缺点与适用环境

- 模式优点
  - 对于扩展一个对象的功能，装饰模式比继承更加灵活，不会导致类的个数急剧增加
  - 可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的具体装饰类，从而实现不同的行为
  - 可以对一个对象进行多次装饰
  - 具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，且原有类库代码无须改变，符合开闭原则

装饰模式的优缺点与适用环境

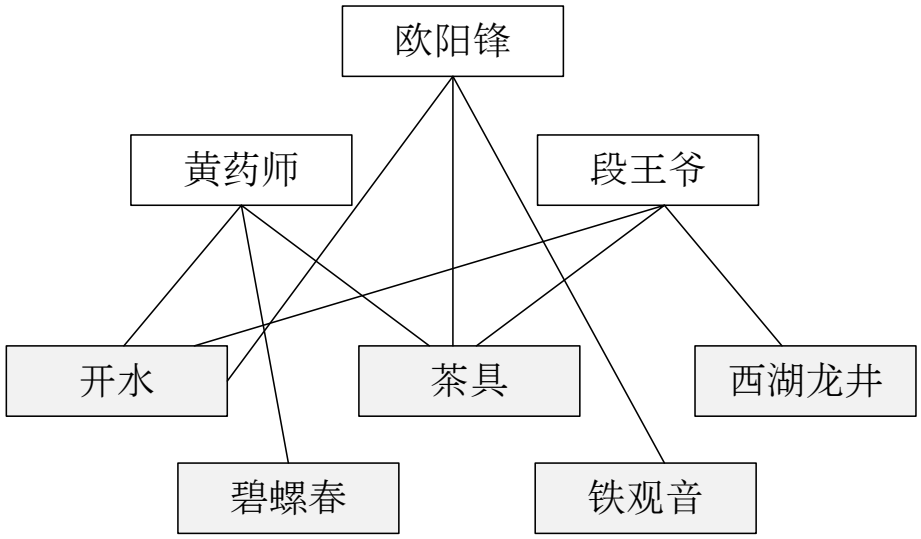
- 模式缺点
  - 使用装饰模式进行系统设计时将产生很多小对象，大量小对象的产生势必会占用更多的系统资源，在一定程度上影响程序的性能
  - 比继承更加易于出错，排错也更困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐

装饰模式的优缺点与适用环境

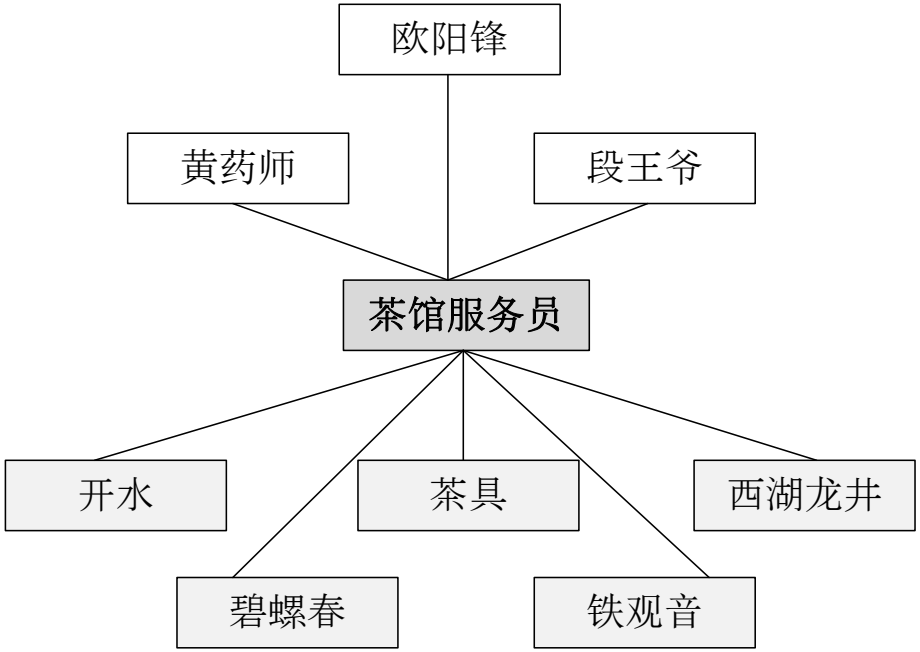
- 模式适用环境
  - 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责
  - 当不能采用继承的方式对系统进行扩展或者采用继承不利于系统扩展和维护时可以使用装饰模式

外观模式概述

- 两种喝茶方式示意图



(A) 自己泡茶



(B) 去茶馆喝茶

外观模式概述

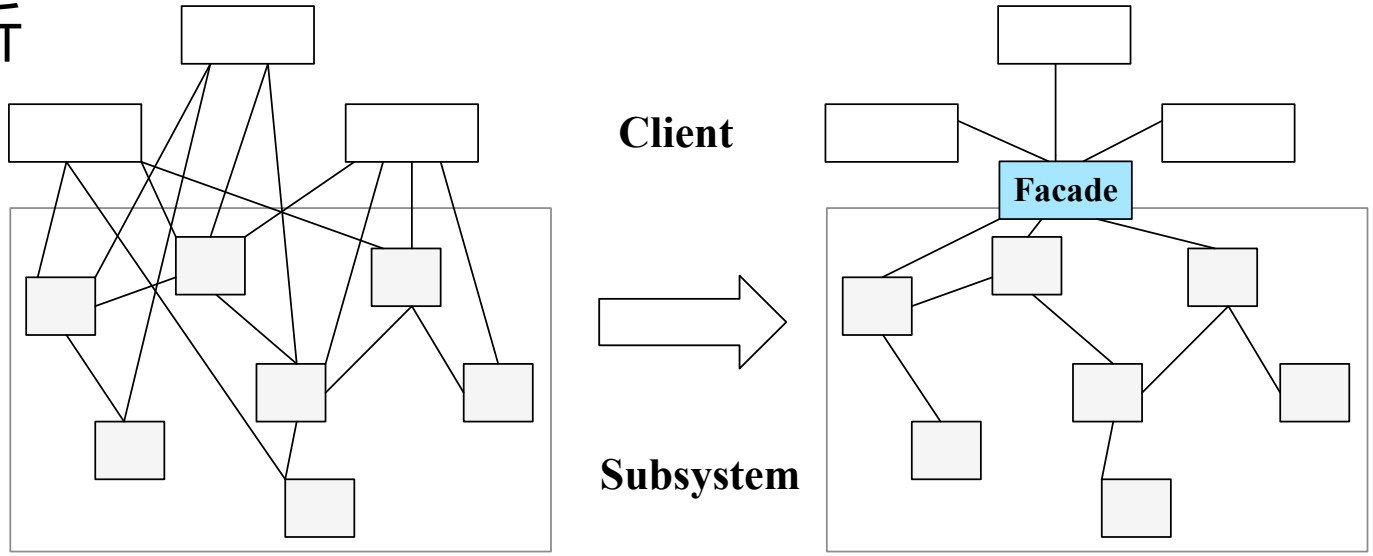
- 分析
  - 一个客户类需要和多个业务类交互，而这些需要交互的业务类经常会作为一个整体出现
  - 引入一个新的外观类(Facade)来负责和多个业务类【子系统(Subsystem)】进行交互，而客户类只需与外观类交互
  - 为多个业务类的调用提供了一个统一的入口，简化了类与类之间的交互

外观模式概述

- 分析
  - 没有外观类：每个客户类需要和多个子系统之间进行复杂的交互，系统的耦合度将很大
  - 引入外观类：客户类只需要直接与外观类交互，客户类与子系统之间原有的复杂引用关系由外观类来实现，从而降低了系统的耦合度

外观模式概述

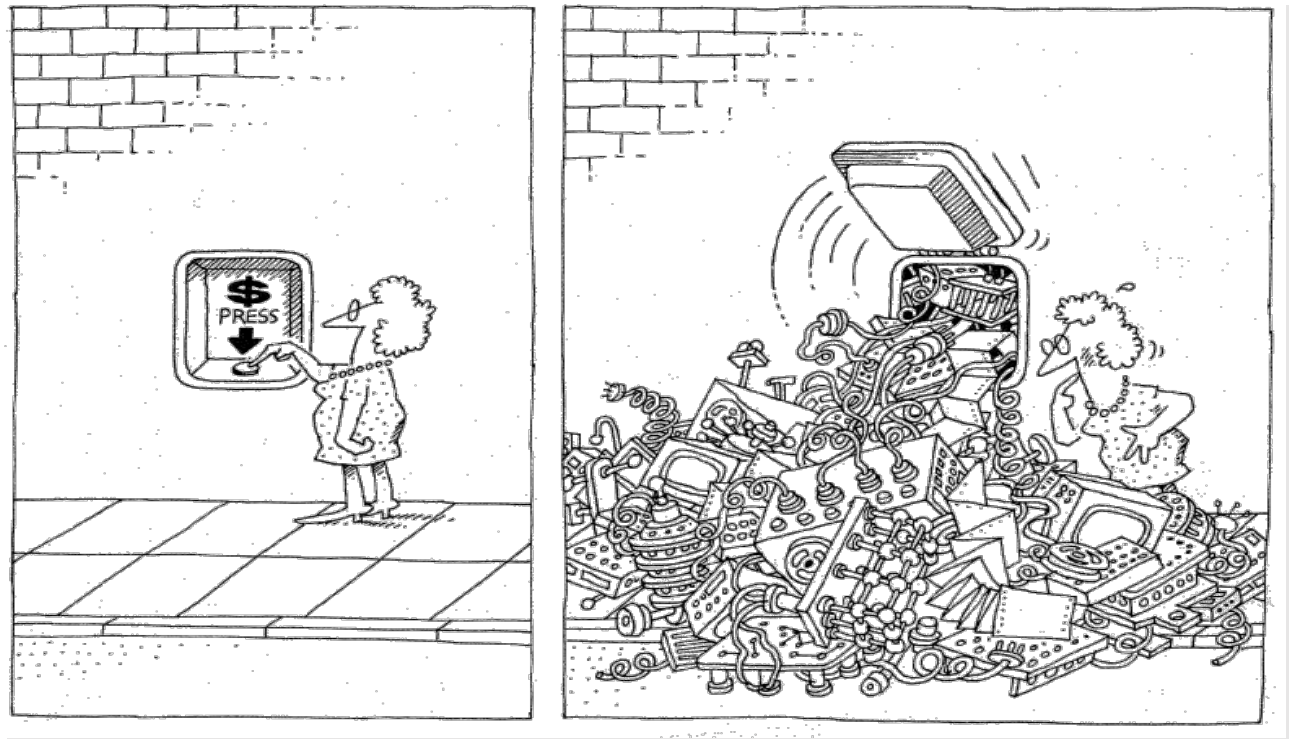
- 分析



  - 一个子系统的外部与其内部的通信通过一个统一的外观类进行，外观类将客户类与子系统的内部复杂性分隔开，使得客户类只需要与外观角色打交道，而不需要与子系统内部的很多对象打交道

外观模式概述

- 分析
  - 为复杂子系统提供一个简单的访问入口



外观模式概述

- 外观模式的定义

外观模式：为子系统中的一组接口提供一个统一的入口。外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

Facade Pattern: Provide **a unified interface** to a set of interfaces in a subsystem. Facade defines **a higher-level interface** that makes the subsystem easier to use.

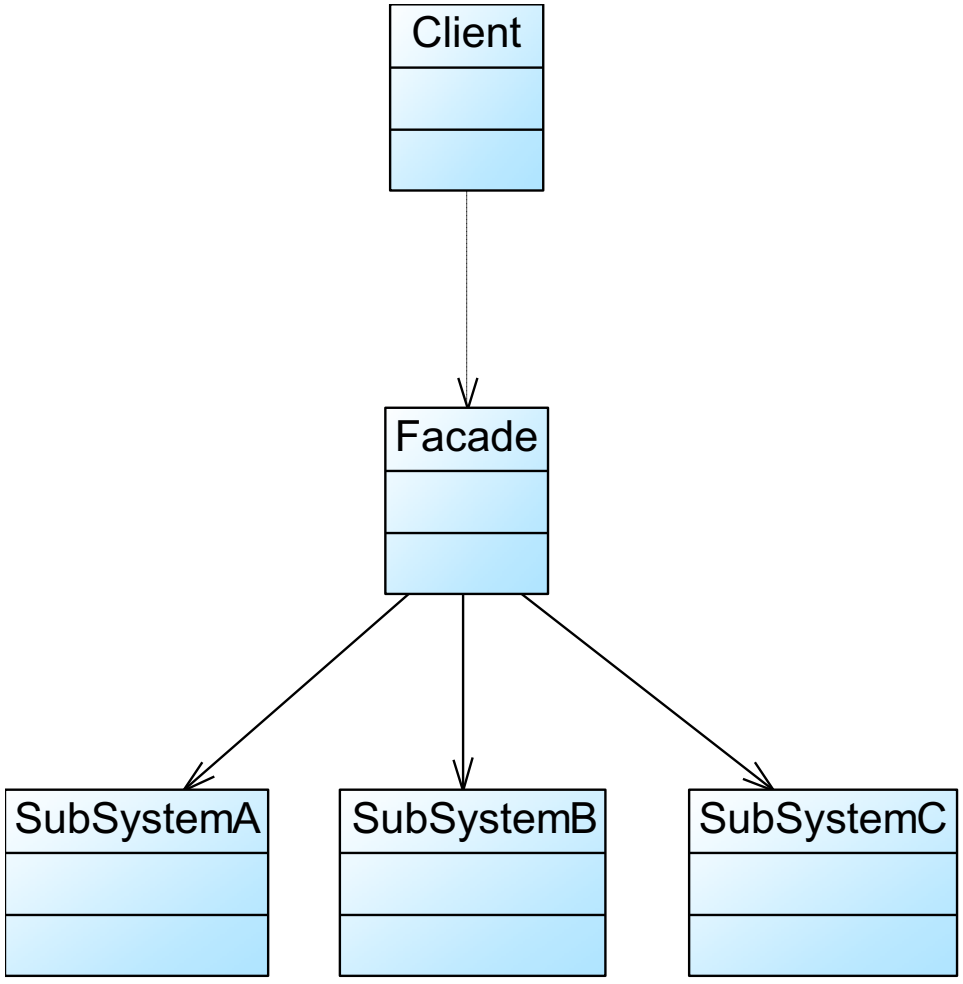
- 对象结构型模式

外观模式概述

- 外观模式的定义
  - 又称为门面模式
  - 是迪米特法则的一种具体实现
  - 通过引入一个新的外观角色来降低原有系统的复杂度，同时降低客户类与子系统的耦合度
  - 所指的子系统是一个广义的概念，它可以是一个类、一个功能模块、系统的一个组成部分或者一个完整的系统

外观模式的结构与实现

- 外观模式的结构



外观模式的结构与实现

- 外观模式的结构
  - 外观模式包含以下2个角色：
    - Facade（外观角色）
    - SubSystem（子系统角色）

外观模式的结构与实现

- 外观模式的实现
  - 子系统类典型代码：

```
public class SubSystemA {
    public void methodA() {
        //业务实现代码
    }
}
public class SubSystemB {
    public void methodB() {
        //业务实现代码
     }
}
public class SubSystemC {
    public void methodC() {
        //业务实现代码
    }
}
```

外观模式的结构与实现

- 外观模式的实现
  - 外观类典型代码：

```
public class Facade {
    private SubSystemA obj1 = new SubSystemA();
    private SubSystemB obj2 = new SubSystemB();
    private SubSystemC obj3 = new SubSystemC();

    public void method() {
        obj1.method();
        obj2.method();
        obj3.method();
    }
}
```

外观模式的结构与实现

- 外观模式的实现
  - 客户类典型代码：

```
public class Client {
    public static void main(String args[]) {
        Facade facade = new Facade();
        facade.method();
    }
}
```

外观模式的应用实例

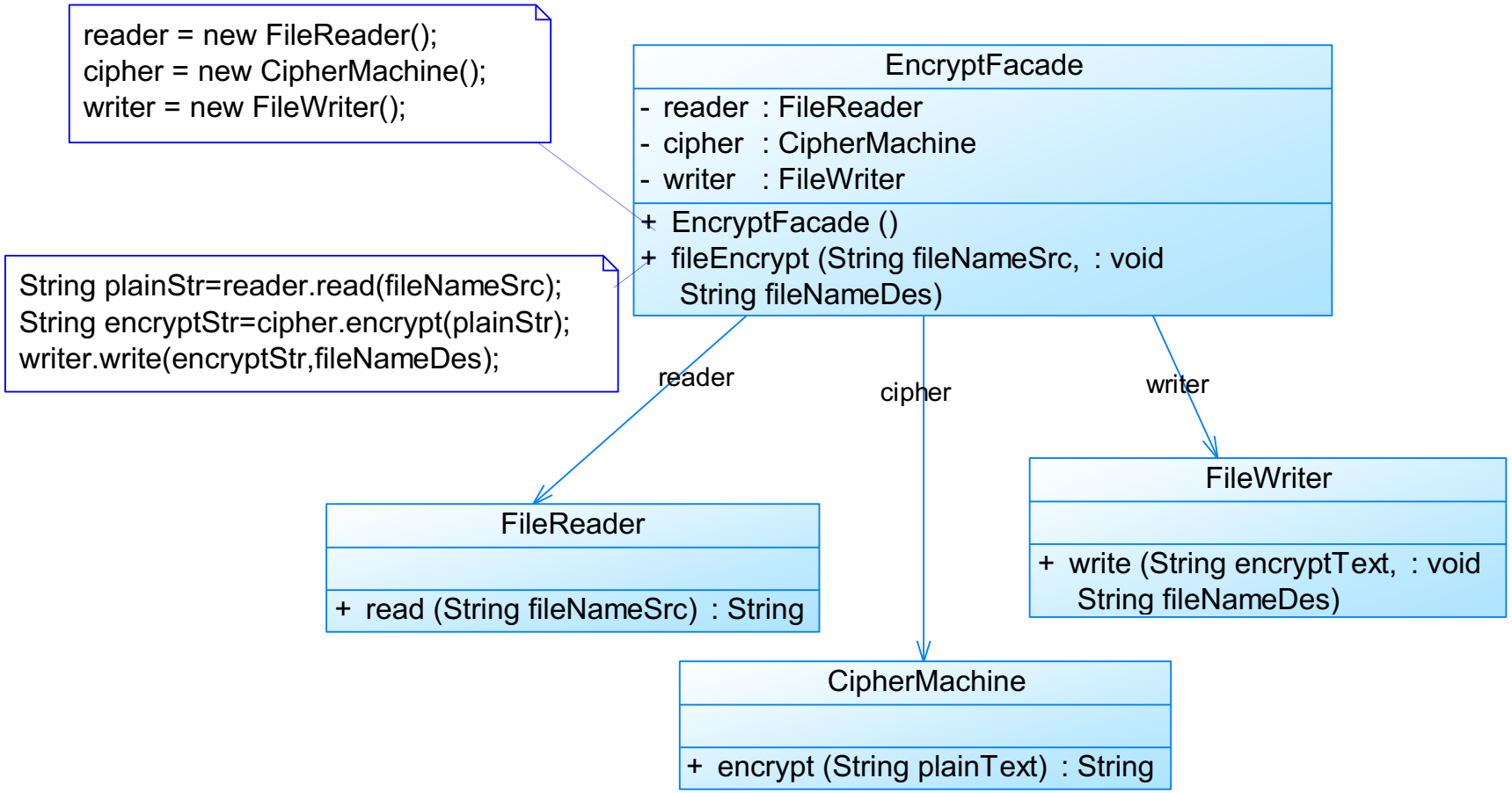
- 实例说明

某软件公司要开发一个可应用于多个软件的文件加密模块，该模块可以对文件中的数据进行加密并将加密之后的数据存储在一个新文件中，具体的流程包括3个部分，分别是读取源文件、加密、保存加密之后的文件，其中，读取文件和保存文件使用流来实现，加密操作通过求模运算实现。这3个操作相对独立，为了实现代码的独立重用，让设计更符合单一职责原则，这3个操作的业务代码封装在3个不同的类中。

现使用外观模式设计该文件加密模块。

外观模式的应用实例

- 实例类图



文件加密模块结构图

外观模式的应用实例

- 实例代码
  - (1) FileReader：文件读取类，充当子系统类
  - (2) CipherMachine：数据加密类，充当子系统类
  - (3) FileWriter：文件保存类，充当子系统类
  - (4) EncryptFacade：加密外观类，充当外观类
  - (5) Client：客户端测试类

演示……

Code (designpatterns.facade)

外观模式的应用实例

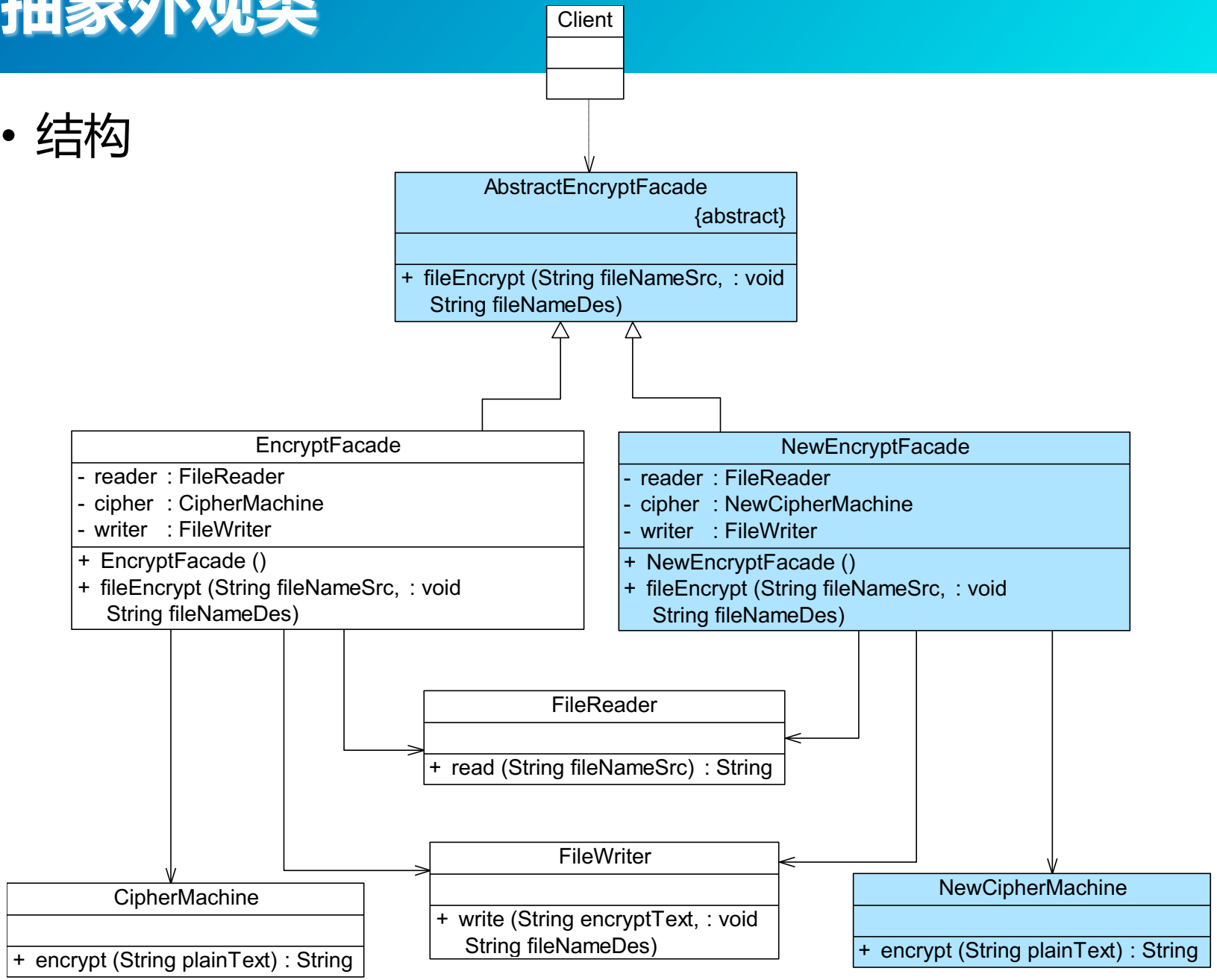
- 结果及分析
  - Hello world! → 233364062325

抽象外观类

- 动机
  - 在标准的外观模式的结构图中，如果需要增加、删除或更换与外观类交互的子系统类，必须修改外观类或客户端的源代码，这将违背开闭原则，因此可以通过引入抽象外观类对系统进行改进，在一定程度上解决该问题

抽象外观类

- 结构


Client
AbstractEncryptFacade
{abstract}
+ fileEncrypt (String fileNameSrc, : void
String fileNameDes)
EncryptFacade
- reader : FileReader
- cipher : CipherMachine
- writer : FileWriter
+ EncryptFacade ()
+ fileEncrypt (String fileNameSrc, : void
String fileNameDes)
NewEncryptFacade
- reader : FileReader
- cipher : NewCipherMachine
- writer : FileWriter
+ NewEncryptFacade ()
+ fileEncrypt (String fileNameSrc, : void
String fileNameDes)
FileReader
+ read (String fileNameSrc) : String
FileWriter
+ write (String encryptText, : void
String fileNameDes)
CipherMachine
+ encrypt (String plainText) : String
NewCipherMachine
+ encrypt (String plainText) : String

抽象外观类

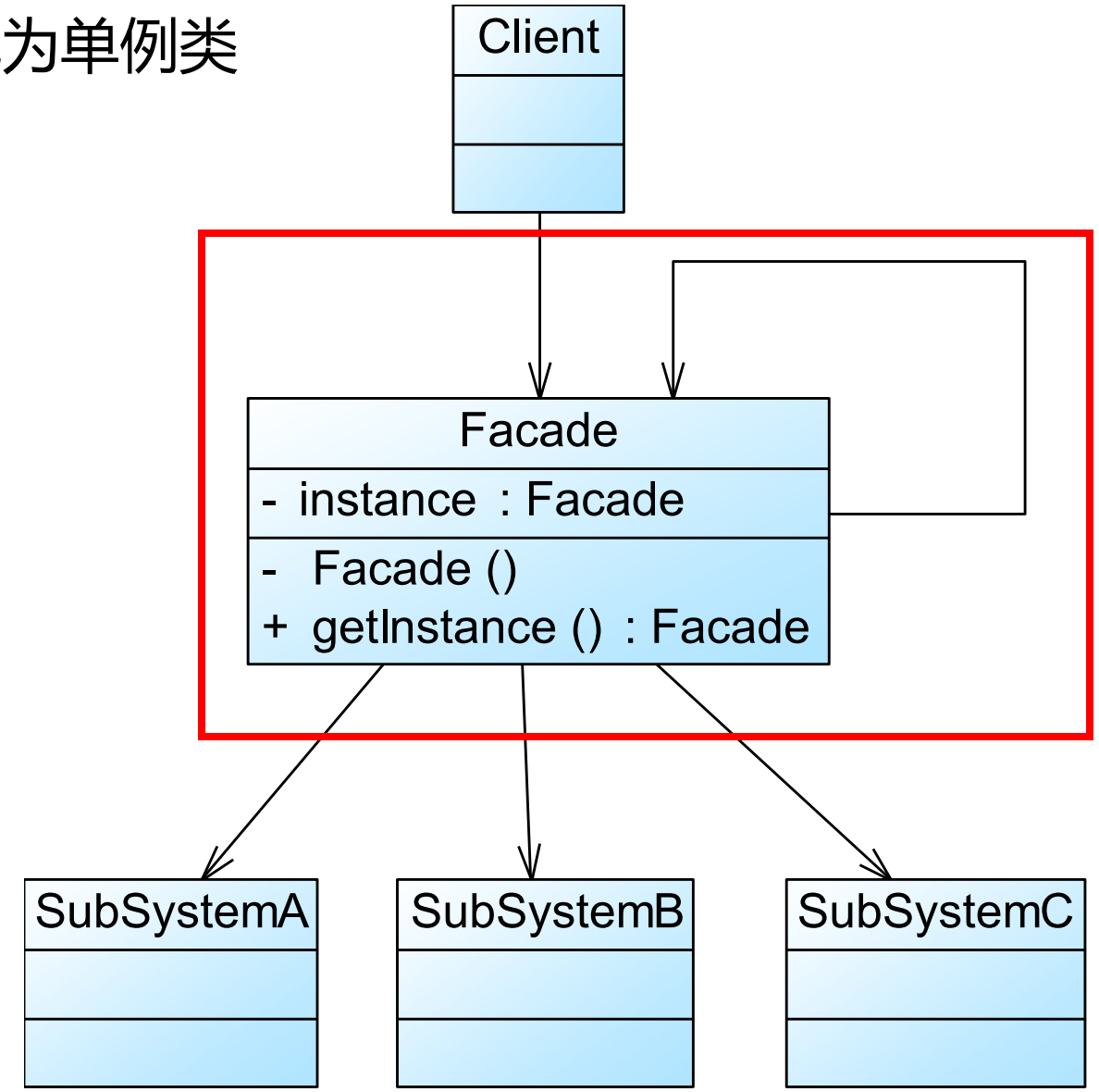
- 实现

```
public class NewEncryptFacade extends AbstractEncryptFacade {
    private FileReader reader;
    private NewCipherMachine cipher;
    private FileWriter writer;

    public NewEncryptFacade() {
        reader = new FileReader();
public class Client {
    public static void main(String args[]) {
        AbstractEncryptFacade ef;
        ef = (AbstractEncryptFacade)XMLUtil.getBean();
        ef.fileEncrypt("src//designpatterns//facade//src.txt","src//designpatterns
//facade//des.txt");    }
}
        writer.write(encryptStr,fileNameDes);
    }
}
```

外观模式与单例模式联用

- 将外观类改造成为单例类



外观模式的优缺点与适用环境

- 模式优点
  - 它对客户端屏蔽了子系统组件，减少了客户端所需处理的对象数目，并使得子系统使用起来更加容易
  - 它实现了子系统与客户端之间的松耦合关系，这使得子系统的变化不会影响到调用它的客户端，只需要调整外观类即可
  - 一个子系统的修改对其他子系统没有任何影响，而且子系统的内部变化也不会影响到外观对象

外观模式的优缺点与适用环境

- 模式缺点
  - 不能很好地限制客户端直接使用子系统类，如果对客户端访问子系统类做太多的限制则减少了可变性和灵活性
  - 如果设计不当，增加新的子系统可能需要修改外观类的源代码，违背了开闭原则

外观模式的优缺点与适用环境

- 模式适用环境
  - 要为访问一系列复杂的子系统提供一个简单入口
  - 客户端程序与多个子系统之间存在很大的依赖性
  - 在层次化结构中，可以使用外观模式的定义系统中每一层的入口，层与层之间不直接产生联系，而是通过外观类建立联系，降低层之间的耦合度

作业

- 在对象适配器中，一个适配器能否适配多个适配者？如果能，应该如何实现？如果不能，请说明原因？如果是类适配器呢？
- JDBC/ODBC桥梁是Bridge模式吗？
- 如何在Composite模式中避免环状引用？

提交作业到教学立方（4月21号24点截止）