设计模式
Design Pattern

Yang YI

Computer Science Department, SYSU

issyy@mail.sysu.edu.cn

Tel:86-13902295111

目录

- 设计模式的要素
- 设计模式的目的
- 设计模式的分类
- 设计模式的选择
- 设计模式的使用

- 如果仅仅看“设计模式”这个词，估计一些同学会觉得有点抽象，不太容易理解
- 如果看到英文“Design+Pattern”，就可能容易理解多了
- 其，就是 “设计”和“pattern” 的结合，即，在软件生命周期的设计阶段，运用的一些pattern（common solution，见“架构设计”那章课件）

设计模式的要素

一个设计模式有以下四个要素：

- 模式名称（name）：助记名，用来描述模式的问题或者解决问题发方法等
- 问题（a common problem）：描述了应该在何时使用模式，陈述要解决的问题
- 解决方案（solution）：描述了设计的组成部分，它们之间的相互关系及各自的职责和协作方式。
- 效果：分析效果，讨论优点或者特点

目录

- 设计模式的要素
- 设计模式的目的
- 设计模式的分类
- 设计模式的选择
- 设计模式的使用

设计模式的目的

- 重用
  - 设计模式提供了针对软件设计和开发过程中一些常见问题的、比较成熟的解决方案
  - 复用这些解决方法可以提高软件开发效率
  - 提高软件的扩展性，增加软件易变性
  - 降低软件运维成本、以及总的生产成本

设计模式（Design Pattern)的分类

- 针对的common problem 的情形不同，给出的解决方案模式也不同

一 创建型 Design Pattern

- Factory Pattern -工厂模式
- Factory Method Pattern -工厂方法模式
- Builder Pattern -建造模式
- Prototype Pattern -原始模型模式
- Singleton Pattern -单例模式

1.Factory Pattern—工厂模式

- 客户类Client（或模块） 需要某一类服务中的一个产品（在此称之为ConcreteProduct），Factory Pattern提供了一个结构，用一个工厂类 ProductFactory（interface）将 Client与 这些提供具体产品的类分开（ConcreteProductA, ConcreteProductB, 。。。。。。）
- 任何时候，client 需要产品ConcreteProduct时，只需向工厂ProductFactory请求即可
- 消费者无须对具体产品类（ConcreteProductA, ConcreteProductB, 。。。。。。）做任何修改，通过调用工厂类 ProductFactory ，就可以使用产品
- 因此，很容易如果增加/删除一个产品，增加了系统的弹性

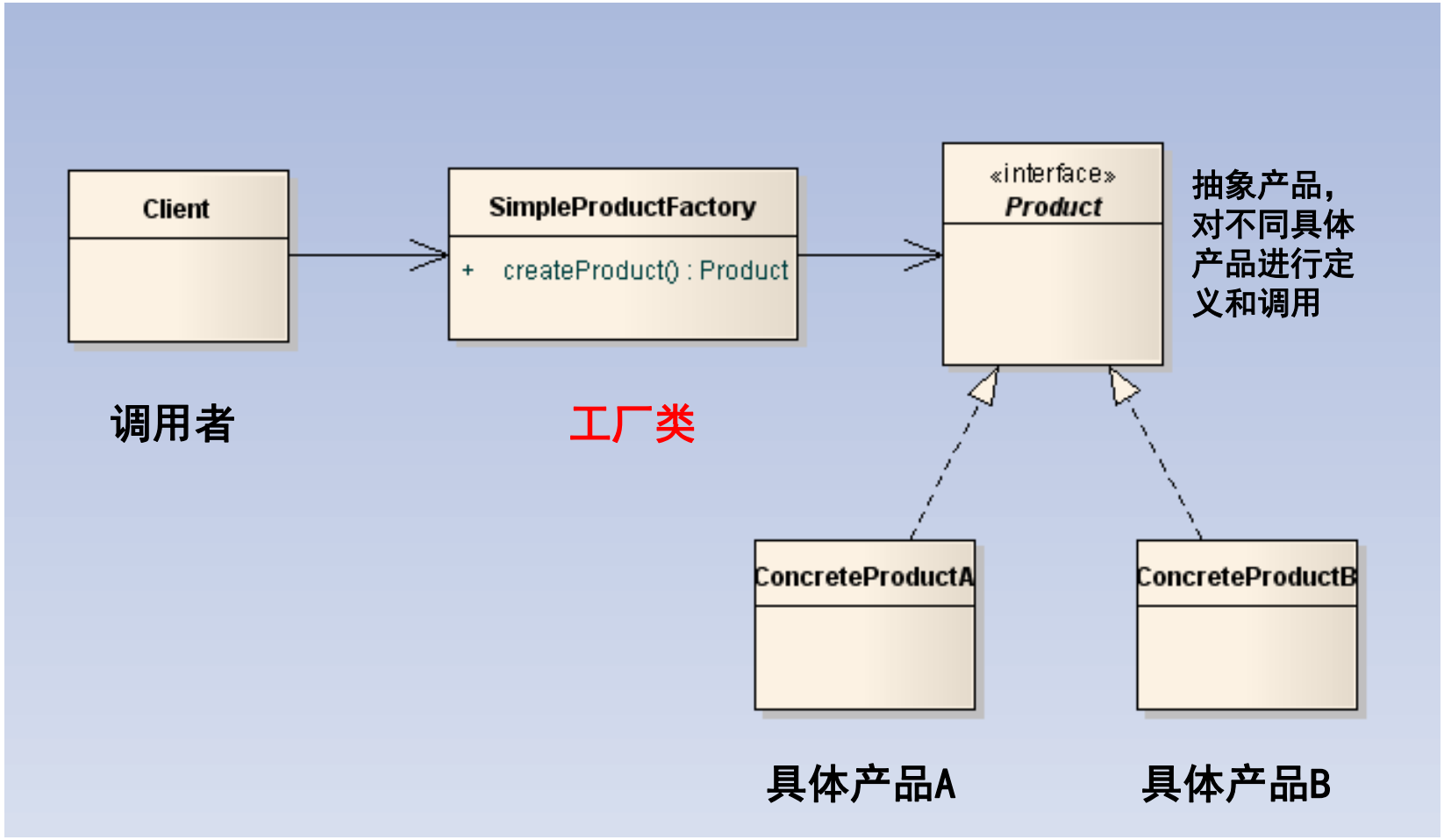
Factory Pattern实例

- 一个应用，需计算各个不同岗位人员的奖金
  - 奖金策略经常改变
  - 岗位包括:项目经理、架构设计师、算法工程师、测试人员、集成人员、销售人员
  - 设计一个类的架构，满足
    - 随时增加一个新岗位，会出现新的策略
    - 随时改变奖金发放策略
    - 架构应支持功能易变，部署运维成本低

Factory Pattern

- 包括四种角色：
  - 工厂（SimpleProductFactory，在后文类图中的名字）：是具体产品类的接口类，该类隔离了Client(客户）与具体的服务（产品）
  - 抽象产品（product）：接口或抽象类，负责定义具体产品及与调用者（客户端）交互
  - 具体产品（conreteProductA, conreteProductA, …）：被工厂类创建的对象，也是客户端实际操作（调用、或者使用）对象。
  - 客户端client：调用工厂类的，服务的请求方

Factory Pattern



Factory Pattern

1. 工厂类

```
public class Factory{
  public static creatProduct(int which){
    if (which==1)
      return new concreteProductA();
    else if (which==2);
      return new concreteProductB();
    }
}
```

Factory Pattern

2. 抽象产品类（用于定义及调用具体产品）

```
public interface Product{
  void function();
 }
```

Factory Pattern

3. 具体产品（也就是每个具体的服务）

```
//产品A的实现
public class concreteProductA implements Product{
    @Override
    public void fuction() {
   System.out.println("Product A");
     }
}
//产品B的实现
public class concreteProductA implements Product{
    @Override
    public void fuction() {
    System.out.println("Product B");
     }
}
```

Factory Pattern

- 4. 客户端（调用者）

```
public class client {
    public static void main(String[] args) {
        factory factoryClass = new factory();
        //获取productA的对象
         product productA = factoryClass.creatProduct(1);
         productA.fuction();
        //获取productB的对象
                  product productB = factoryClass.creatProduct(1);
         productB.fuction();
    }
}
```

Factory Pattern

- 优点
  - 一个调用者想创建一个对象，只要知道其名称就可以了
  - 扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以
  - 屏蔽产品的具体实现，调用者只关心产品的接口
- 缺点
  - 当产品修改时，工厂类也要做相应的修改
  - 当产品有很多、且不同类型时，难以管理，运行效率低

2.Factory Method Pattern-工厂方法模式

- 当产品的类型多了，不是的单一的类型，一个工厂不够，可以通过创建多个工厂，每个工厂满足“工厂模式”来实现
- 核心工厂类不再负责所有产品的创建，而是将具体创建的工作交给子类去做，成为一个抽象工厂角色，仅负责给出具体工厂类必须实现的接口，而不接触产品类

Factory Method Pattern

- 一个控制笔生产的应用系统
  - 有很多类型的圆珠笔
  - 每个类型的笔存在有很多种颜色
  - 新的设计不断出现，很可能不断有新的类型以及新的颜色的笔
- 请你设计一个类的结构，支持对类型和颜色都方便的扩展

Factory Method Pattern

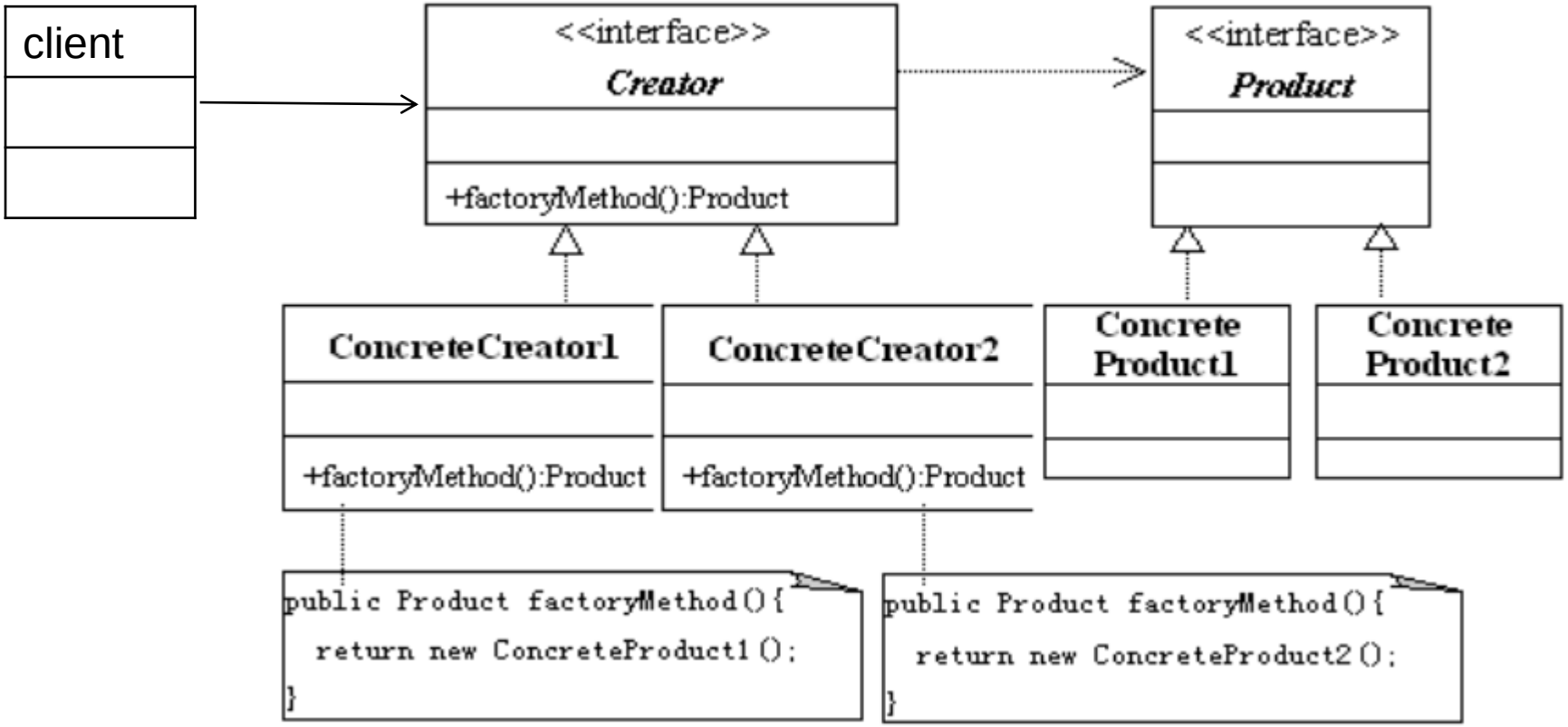
模式的结构中包括四种角色：

- 抽象产品（Product） ：interface
- 具体产品（ConcreteProduct）
- 构造者（Creator） （工厂方法，interface）
- 具体构造者（ConcreteCreator） （工厂）

Factory Method Pattern

工厂方法类，定义和调用具体工厂

抽象产品类，定义和调用具体产品



Factory Method Pattern

1. 抽象产品（Product） ： PenCore.java

```
public abstract class PenCore{
    String color;
    public abstract void writeWord(String s);
}
```

Factory Method Pattern

2. 具体产品（ConcreteProduct）_1 ：

RedPenCore.java

```
public class RedPenCore extends PenCore{
    RedPenCore(){
      color="红色";
    }
    public void writeWord(String s){
       System.out.println("写出"+color+"的字:"+s);
    }
}
```

Factory Method Pattern

2. 具体产品（ConcreteProduct）_2 :

BluePeCore.java

```
public class BluePenCore extends PenCore{
    BluePenCore(){
      color="蓝色";
    }
    public void writeWord(String s){
       System.out.println("写出"+color+"的字:"+s);
    }
}
```

Factory Method Pattern

3. 构造者（Creator）:BallPen.java

```
public abstract class BallPen{
    BallPen(){
        System.out.println("生产了一只装有"+
         getPenCore().color+"笔芯的圆珠笔");
    }
    //工厂方法
    public abstract PenCore getPenCore();
}
```

Factory Method Pattern

4. 具体构造者（ConcreteCreator）：

```
//RedBallPen.java
public class RedBallPen extends BallPen{
    public PenCore getPenCore(){
        return new RedPenCore();
    }
}

//BlueBallPen.java
public class BlueBallPen extends BallPen{
    public PenCore getPenCore(){
        return new BluePenCore();
    }
}
```

Factory Method Pattern

5. 应用 Application.java

```
public class Application{
    public static void main(String args[]){
        PenCore penCore;
        BallPen ballPen=new BlueBallPen();
        penCore=ballPen.getPenCore();
        penCore.writeWord("你好,很高兴认识你");
        ballPen=new RedBallPen();
        penCore=ballPen.getPenCore();
        penCore.writeWord("How are you");
    }
}
```

Factory Method Pattern

- 可让用户的代码和某个特定类的子类的代码解耦，从而提高了“类型”方面的扩展性
- 用户不必知道它所使用对象是怎样被创建的，只需知道该对象有哪些方法即可
- 支持的具体产品可以很多类型，支持大规模系统

3.Builder Pattern-建造模式

- 将一个复杂对象的构建与它的表示(demonstration，web page)分离，使得同样的构建过程可以创建不同的表示
- 使得产品内部实现的内容可以独立的变化，客户不必知道产品内部组成的细节
- 提高了封装和可扩展性
- 可强制实行一种分步骤进行的建造过程

Builder Pattern 概述

- 当系统准备为用户提供一个内部结构复杂的对象时，例如构造一个应用的复杂界面，就可以使用
- 可以逐步地构造对象（满足界面实现的易变性），使得对象的创建更具弹性
- 关键步骤是：将一个包含有多个组件对象的创建，分成若干个步骤，并将这些步骤封装在一个称作生成器的接口中

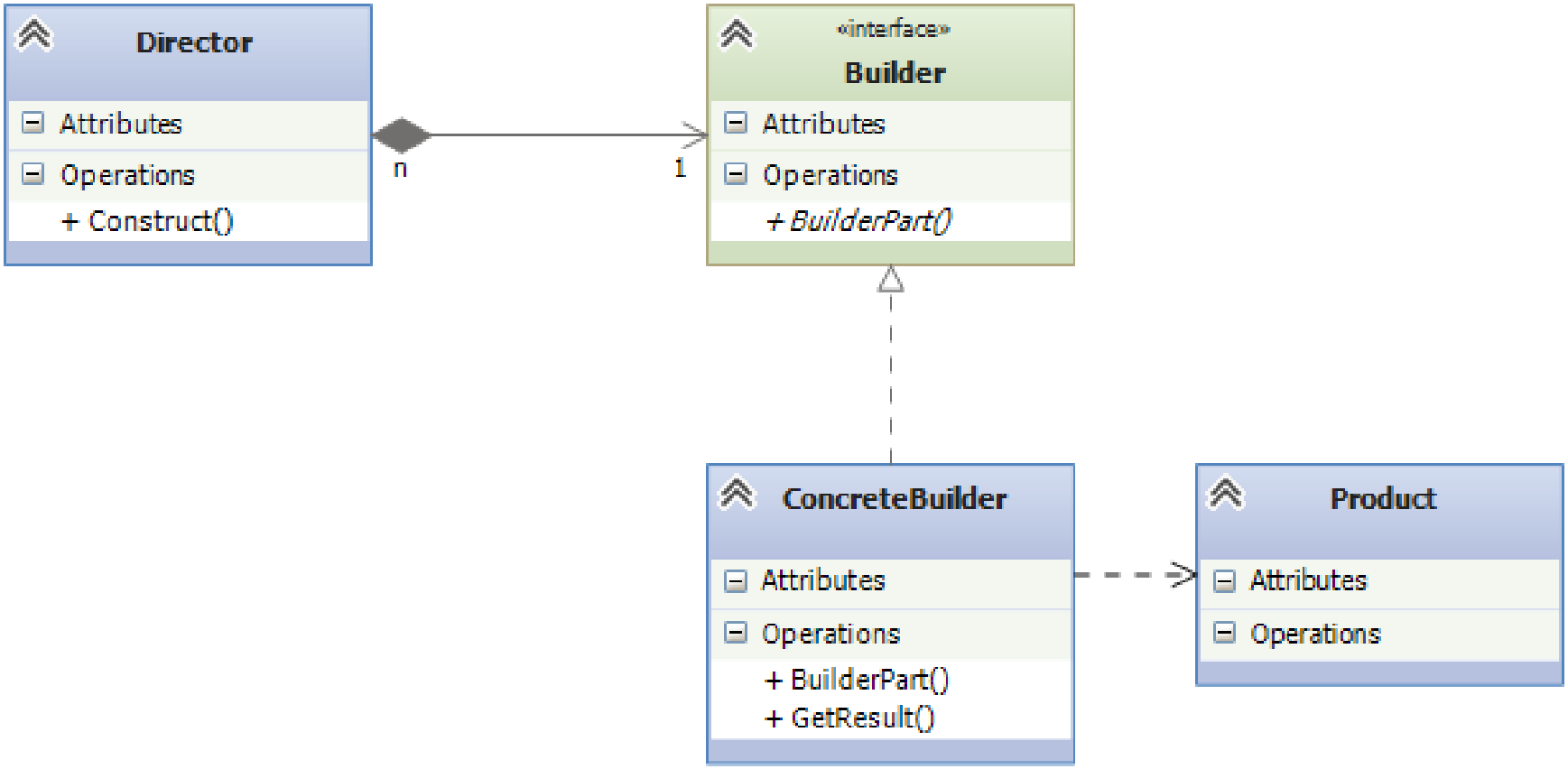
Builder Pattern实例

- 创建含有“按钮、标签和文本框组件的容器”
- 不同用户对容器有不同的要求，如某些用户希望容器中只含有按钮和标签，某些用户希望容器只含有按钮和文本框等
- 另外，用户对组件在容器中的顺序位置也有不同的要求，如某些用户要求组件在容器中从左至右的排列顺序是按钮、标签、文本框

Builder Pattern 结构

- 本模式的结构中包括四种角色：
  - 产品（Product）
  - 抽象生成器（Builder）
  - 具体生成器（ConcreteBuilder）
  - 指挥者（Director）

Builder Pattern



请各位同学思考:如计划再新增一个界面，新的类图需要增加什么？并请画出UML类图

Builder Pattern结构描述和使用

```
public interface Builder {
  //创建部件A,比如创建汽车车轮
  void buildPartA();
  //创建部件B 比如创建创建汽车方向盘
  void buildPartB();
  //创建部件C 比如创建汽车发动机
  void buildPartC();
  //返回最后组装成品结果 (返回最后装配好的汽车 )
  //成品的组装过程不在这里进行,而是转移到下面的Director类别中进行
  Product getResult();
 }
public class Director {
  private Builder builder;
  public Director( Builder builder ) {
    this.builder = builder;
  }
  // 将部件partA partB partC最后组成物体
  public void construct() {
    builder.buildPartA();
    builderbuilder.buildPartB();
    builderbuilderbuilder.buildPartC();
  }
}
```

Builder Pattern结构描述和使用

1. 产品（Product）：PanelProduct.java

```
import javax.swing.*;
public class PanelProduct extends JPanel{
    JButton button;
    JLabel label;
}
```

Builder Pattern结构描述和使用

2. 抽象生成器（Builder）：Builer.java

```
import.javax.swing.*;
public interface Builder{
    public abstract void buildButton();
    public abstract void buildLabel();
    public abstract JPanel getPanel();
}
```

Builder Pattern结构描述和使用

3. 具体生成器（ConcreteBuilder）_1:

ConcreteBuilderOne.java

```
import javax.swing.*;
public class ConcreteBuilderOne implements Builder{
        private PanelProduct panel;
        ConcreteBuilderOne(){ panel=new PanelProduct();}
        public void buildButton(){  panel.button=new JButton("按钮");}
        public void buildLabel(){ panel.label=new JLabel("标签");}
        public JPanel  getPanel(){
          panel.add(panel.button);
          panel.add(panel.label);
          return panel;
    }
}
```

Builder Pattern结构描述和使用

3．具体生成器（ConcreteBuilder）_2:

ConcreteBuilderTwo.java

```
import javax.swing.*;
public class ConcreteBuilderTwo implements Builder{
  private PanelProduct panel;
  ConcreteBuilderTwo(){ panel=new PanelProduct();}
  public void buildButton(){  panel.button=new JButton("button");}
  public void buildLabel(){}
  public void buildTextField(){}
  public JPanel  getPanel(){
    panel.add(panel.button);
    return panel;
  }
}
```

Builder Pattern结构描述和使用

4. 指挥者（Director）:Director.java

```
import javax.swing.*;
public class Director{
    private Builder builder;
    Director(Builder builder){
        this.builder=builder;}
    public JPanel constructProduct(){
        builder.buildButton();
        builder.buildLabel();
        JPanel product=builder.getPanel();
        return product;
  }
 }
```

Builder Pattern结构描述和使用

5. 应用 Application.java

```
import javax.swing.*;
public class Application{
    public static void main(String args[]){
        Builder builder=new ConcreteBuilderOne();
        Director director=new Director(builder);
        JPanel panel=director.constructProduct();
        JFrame frameOne=new JFrame();
        frameOne.add(panel);
        frameOne.setBounds(12,12,200,120);
        frameOne.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE)
        frameOne.setVisible(true);
        builder=new ConcreteBuilderTwo();
        director=new Director(builder);
        panel=director.constructProduct();
        JFrame frameTwo=new JFrame();
        frameTwo.add(panel);
        frameTwo.setBounds(212,12,200,120);
        frameTwo.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE)
        frameTwo.setVisible(true);
    }
}
```

Builder Pattern

- 优点
  - 可以很容易呈现丰富的结果
    - 将对象的构造过程封装在具体生成器中，用户使用不同的具体生成器可得到该对象的不同结果
  - 可更加精细有效地控制对象的构造过程
    - 生成器将对象的构造过程分解成若干步骤，这就使得程序可以更加精细，有效地控制整个对象的构造
- 缺点
  - 产品必须有共同点，范围有限制
  - 如内部变化复杂，会有很多的建造类

4. Prototype Pattern–原型模式

- 给出一个原型对象，指明所要创建的对象
- 然后，用复制这个原型对象的方法创建出更多同样的对象
- 目标:大规模快速复制和重用

Prototype Pattern概述

- 实现了一个原型（对象）的接口，该接口可创建当前对象的克隆
- 当直接创建对象的代价较大时，可采用这种模式
  - 例如，一个对象需要在一个高代价的数据库操作之后被创建，我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用

Prototype Pattern实例

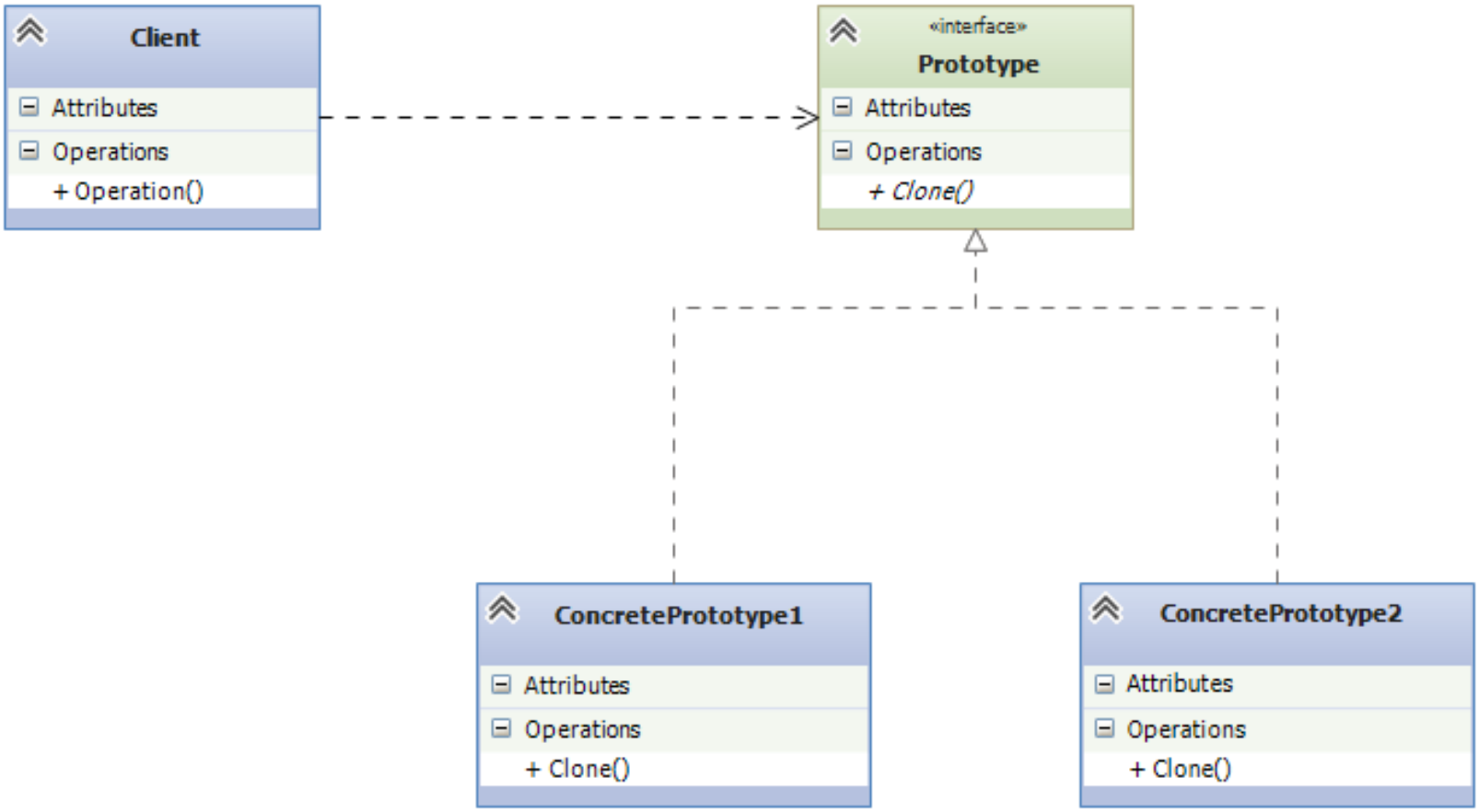
- 细胞分裂操作
- JAVA 中的 Object clone() 方法
  - 实验室电脑中OS 以及基础应用软件的安装

结构描述与使用

原型模式的结构中包括两种角色：

- 抽象原型（Prototype）
- 具体原型（Concrete Prototype）

Prototype Pattern



请思考:如计划再新增一个对象，新的类图需要增加什么？

并请画出UML类图

Prototype Pattern结构描述与使用

1．抽象原型（Prototype）：Prototype.java

```
public interface  Prototype {
    Public Object cloneMe() throws
    CloneNotSupportedException;
}
```

Prototype Pattern结构描述与使用

2. 具体原型（Concrete Prototype）_1:

Cubic.java

```
public class Cubic implements Prototype, Cloneable{
        double  length,width,height;
        Cubic(double a,double b,double c){
                length=a;
                width=b;
                height=c;
        }
        public Object cloneMe() throws CloneNotSupportedException{
                Cubic object=(Cubic)clone();
                return object;
        }
}
```

Prototype Pattern结构描述与使用

2. 具体原型（Concrete Prototype）_2:

Goat.java

```
import java.io.*;
public class Goat implements Prototype,Serializable{
  StringBuffer color;
  public void setColor(StringBuffer c){ color=c;}
  public  StringBuffer getColor(){  return color;}
  public Object cloneMe() throws CloneNotSupportedException{
    Object object=null;
    try{
      ByteArrayOutputStream outOne=new ByteArrayOutputStream();
      ObjectOutputStream outTwo=new ObjectOutputStream(outOne);
      outTwo.writeObject(this);
      ByteArrayInputStream  inOne=
      new ByteArrayInputStream(outOne.toByteArray());
      ObjectInputStream inTwo=new ObjectInputStream(inOne);
      object=inTwo.readObject();
      }
    catch(Exception event){System.out.println(event);}
    return object;
  }
}
```

Prototype Pattern结构描述与使用

3. 应用 Application.java

```
public class Application{
    public static void main(String args[]){
        Cubic  cubic=new Cubic(12,20,66);
        System.out.println(cubic.length+","+cubic.width+","+cubic.height);
        try{
            Cubic  cubicCopy=(Cubic)cubic.cloneMe();
            System.out.println(cubicCopy.length+","+cubicCopy.width+","
            +cubicCopy.height); }
        catch(CloneNotSupportedException exp){}
        Goat  goat=new Goat();
        goat.setColor(new StringBuffer("白颜色的山羊"));
        System.out.println("goat是"+goat.getColor());
         try{
            Goat  goatCopy=(Goat)goat.cloneMe();
            System.out.println("goatCopy是"+goatCopy.getColor());
            System.out.println("goatCopy将自己的颜色改变成黑色");
            goatCopy.setColor(new StringBuffer("黑颜色的山羊"));
            System.out.println("goat仍然是"+goat.getColor());
            System.out.println("goatCopy是"+goatCopy.getColor()); }
         catch(CloneNotSupportedException exp){}
    }
}
```

Prototype Pattern

- 优点
  - 当创建类的新实例的代价更大时，使用原型模式复制一个已有的实例可以提高创建新实例的效率
  - 可动态地保存当前对象的状态
    - 在运行时刻，可随时使用对象流保存当前对象的一个复制品
- 缺点
  - 每一个类都必须配备一个克隆方法

5. Singleton Pattern -单例模式

- 单例模式确保某个类只有一个实例
- 而且, 自行实例化并向整个系统提供这个实例
- 单例模式只应在有真正的“单一实例”的需求时才可使用

Singleton Pattern概述

- 怎样设计一个类，并使得该类只有一个实例的成熟模式?
  - 将类的构造方法设置为private权限，并提供一个返回的唯一实例的类方法
  - 它可保证一个类仅有一个实例，并提供一个访问它的全局访问点

Singleton Pattern实例

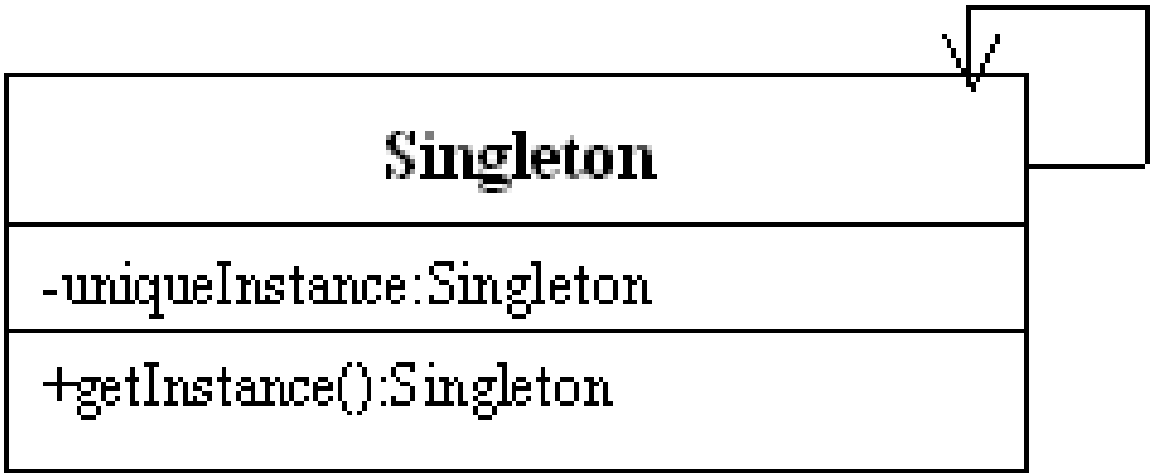
- Windows 在操作一个文件的时候，会出现多个进程或线程同时操作一个文件的现象，所以，所有文件的处理必须通过唯一的实例来进行
- 一些设备管理器常常设计为单例模式，如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件

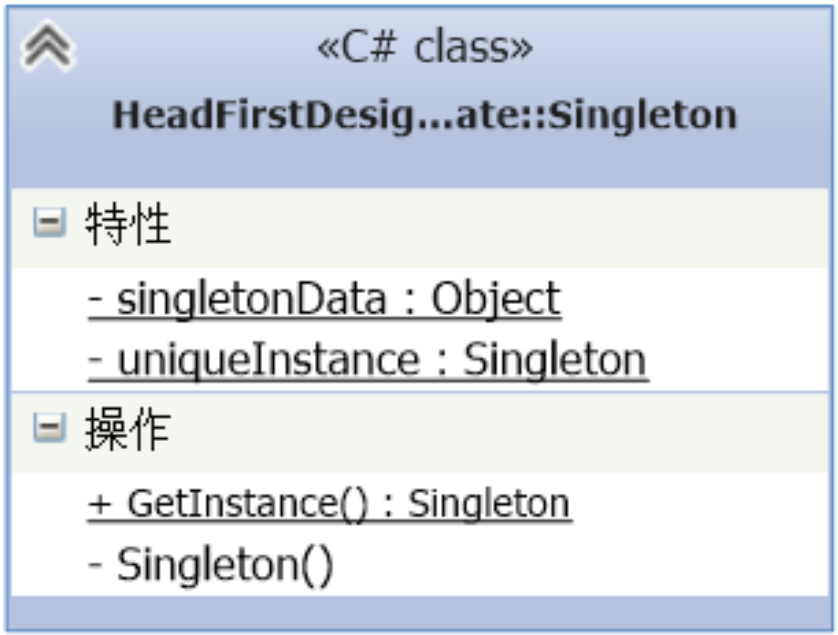
Singleton Pattern结构描述与使用

模式的结构中只包括一个角色：

- 单件类（Singleton）

Singleton Pattern





Singleton Pattern结构描述与使用

- 单件类（Singleton）

```
public class Singleton {
  private static Singleton instance = null;
  public static synchronized Singleton
  getInstance() {
  //这个方法比上面有所改进，不用每次都进行生成
  //对象，只是第一次。使用时生成实例，提高了效率！
  if (instance==null) instance=new Singleton();
    return instance;
  }
}
```

Singleton Pattern

- 优点
  - 类的唯一实例由单件类本身来控制，可很好地控制用户何时访问它，不受干扰，避免“脏写”
  - 在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例

Singleton Pattern

- 缺点
  - 单例类的扩展很困难
  - 单例类的职责过重，在一定程度上违背了“单一职责原则”

二 结构型模式

- **Adapter Pattern -适配器**
- **Bridge Pattern -桥梁模式**
- **Composite Pattern -合成模式**
- **Decorator Pattern -装饰模式**
- **Façade Pattern -门面模式**
- **Flyweight Pattern -享元模式**
- **Proxy Pattern -代理模式**

6. Adapter Pattern -适配器模式

- 某一个类提供了某项需要的服务，可是，该类的接口不可调用（接口不匹配），如何处理？
- 把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法一起工作的两个类能够一起工作
- 适配类可根据参数返还一个合适的实例给客户端

Adapter Pattern实例

- 在实际生活中有一个常见的类似应用
  - 美国电源插口与中国不同，中国人出国如何接电
  - 在 Linux上运行 Windows程序

Adapter Pattern概述

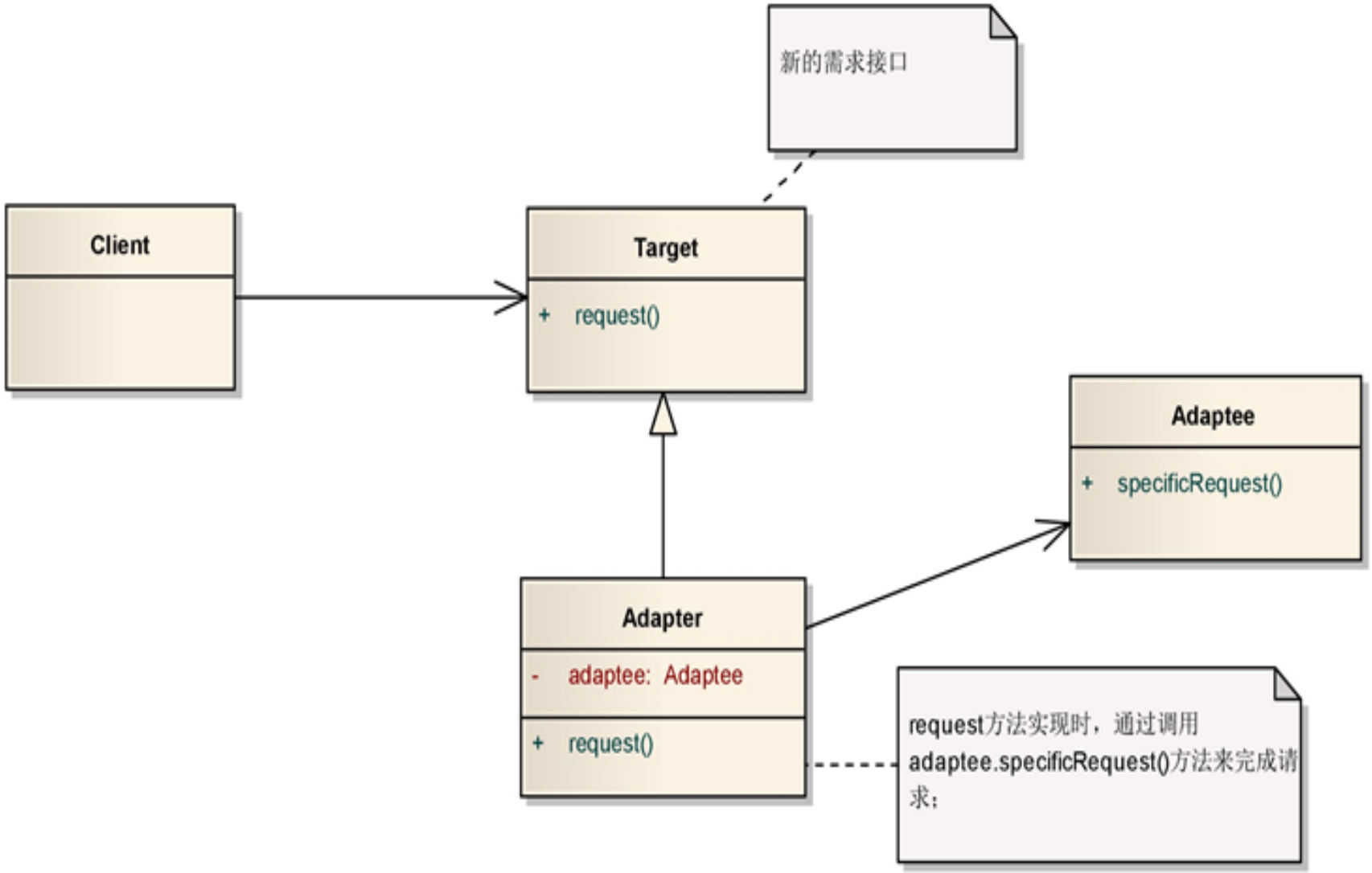
- 将一个类的接口（被适配者）转换成客户希望的另外一个接口（目标）
  - 该模式中涉及有目标、被适配者和适配器
- 关键是建立一个适配器，这个适配器实现了目标接口并包含有被适配者的引用

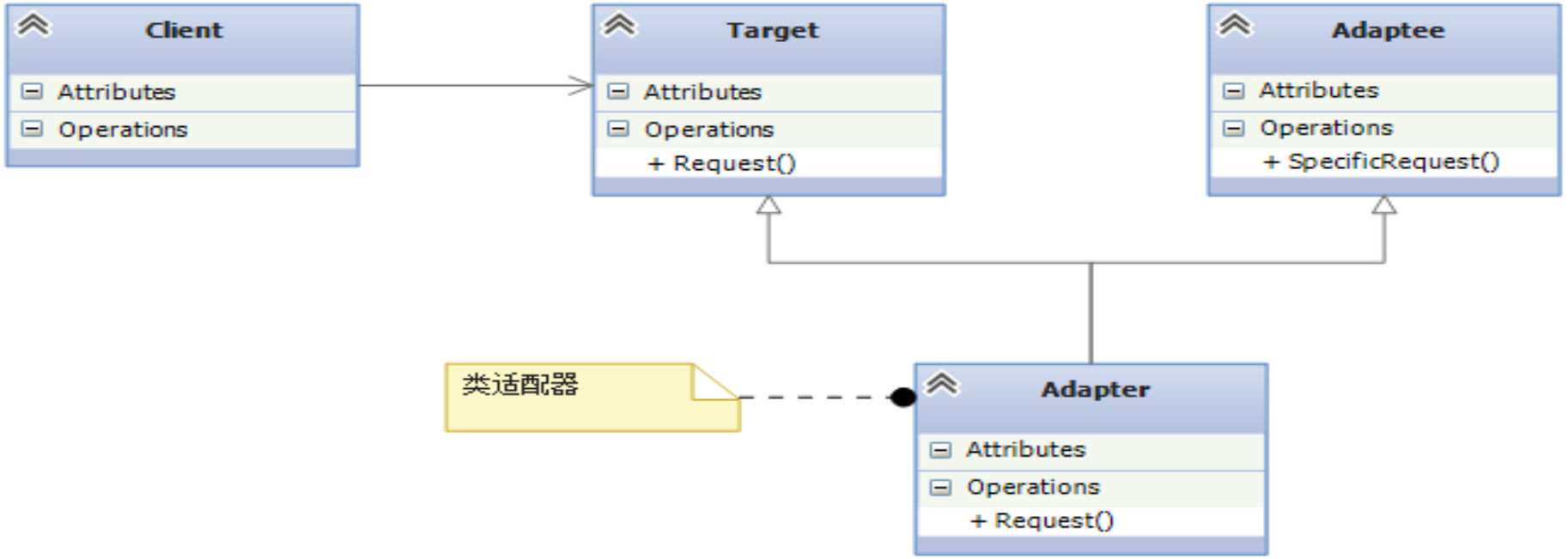
Adapter Pattern结构与使用

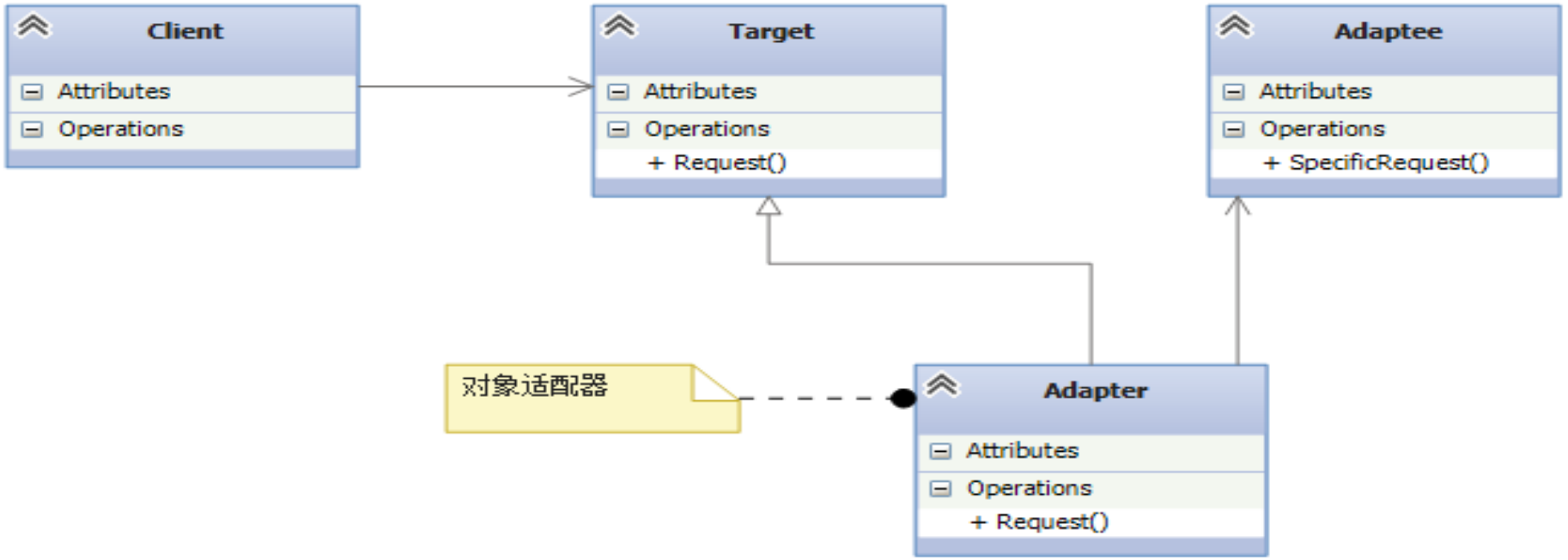
模式的结构中包括三种角色：

- 目标（Target）
- 被适配者（Adaptee）
- 适配器（Adapter）

Adapter Pattern




Client
Attributes
Operations
Target
Attributes
Operations
+ Request()
Adaptee
Attributes
Operations
+ SpecificRequest()
类适配器
Adapter
Attributes
Operations
+ Request()


Client
Attributes
Operations
Target
Attributes
Operations
+ Request()
Adaptee
Attributes
Operations
+ SpecificRequest()
对象适配器
Adapter
Attributes
Operations
+ Request()

Adapter Pattern结构的描述与使用

1．目标（Target）/被适配者（Adaptee）：

```
//IRoundPeg.java
public interface IRoundPeg{
  public void insertIntoHole(String msg);
}

//ISquarePeg.java
public interface ISquarePeg{
  public void insert(String str);
}
```

两个接口相互适配，所以他们既是目标又是被适配者

Adapter Pattern结构的描述与使用

2. 适配器（Adapter）：PegAdapter.java

```
public class PegAdapter implements IRoundPeg,ISquarePeg{
  private RoundPeg roundPeg;
  private SquarePeg squarePeg;
  public PegAdapter(RoundPeg peg){this.roundPeg=peg;}
  public PegAdapter(SquarePeg peg){this.squarePeg=peg;}
  public void insert(String str){
    roundPeg.insertIntoHole(str);
  }
  public void insertIntoHole(String str){
    SquarePeg.insert(str);
  }
}
```

Adapter Pattern

- 优点
  - 目标（Target）和被适配者（Adaptee）是完全解耦的关系
  - 满足“开-闭原则”当
    - 添加一个实现Adaptee接口的新类时，不必修改Adapter，Adapter就能对这个新类的实例进行适配
- 缺点
  - 过多地使用适配器，会让系统非常零乱，不易整体进行把握

7. Bridge Pattern -桥梁模式

- 将抽象化与实现化脱耦，使得二者可以独立的变化也
- 即，将他们之间的强关联变成弱关联
- 即，指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系，从而使两者可以独立的变化

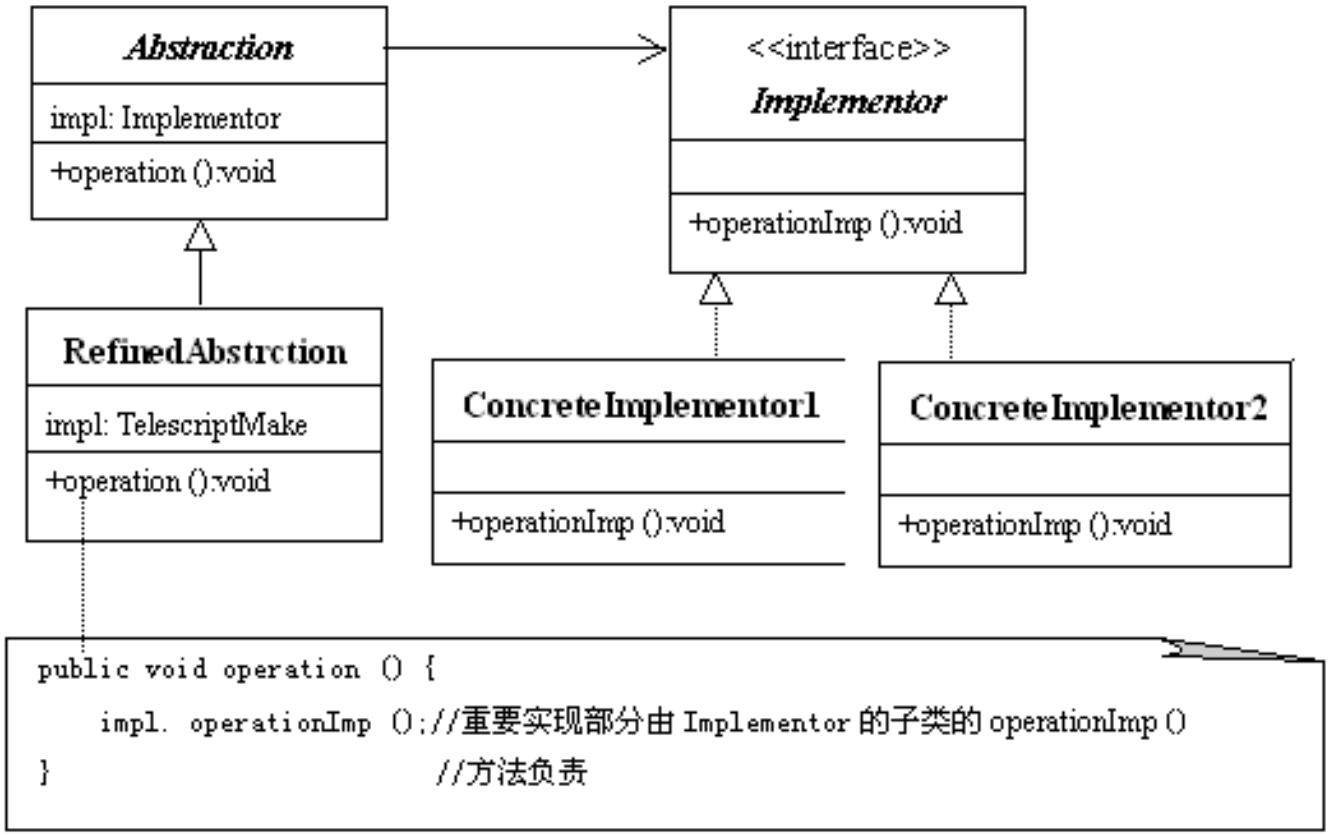
Bridge Pattern实例

- 比如电灯和风扇都由开关控制，任何时候，可在不更换开关的情况下换掉灯
- 也可在不接触灯泡或风扇的情况下更换开关
- 甚至可在不接触开关的情况下，把灯泡和风扇的开关互换

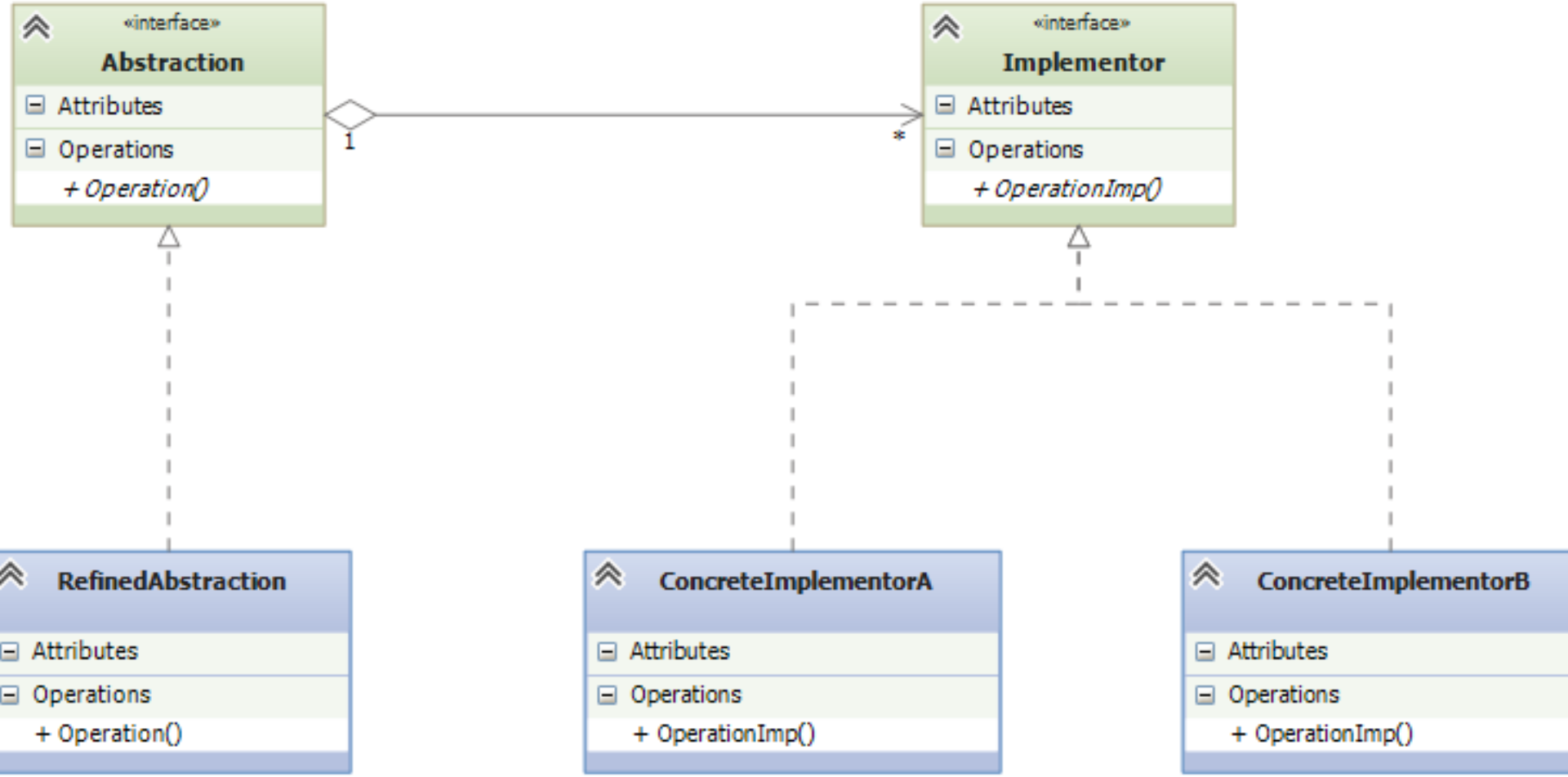
Bridge Pattern结构和使用

- 包括四种角色：
  - 抽象（Abstraction）
  - 实现者（Implementor）
  - 细化抽象（Refined Abstraction）
  - 具体实现者（Concrete Implementor）

Bridge Pattern



Bridge Pattern



Bridge Pattern结构描述和使用

1. 抽象（Abstraction）：

ArchitectureCose.java

```
public abstract class  ArchitectureCost{
        BuildingDesign  design;
        double unitPrice;
        public  abstract double giveCost() ;
}
```

Bridge Pattern结构的描述和使用

2. 实现者（Implementor）：

BuildingDesign.java

```
public interface BuildingDesign{
        public double computerArea();
 }
```

Bridge Pattern结构的描述和使用

3. 细化抽象（Refined Abstraction）：

BuildingCose.java

```
public  class  BuildingCost extends ArchitectureCost{
        BuildingCost(BuildingDesign design,double unitPrice){
                this.design=design;
                this.unitPrice=unitPrice;
        }
        public  double giveCost() {
                double area=design.computerArea();
                return area*unitPrice;
        }
}
```

Bridge Pattern结构的描述和使用

4. 具体实现者（Concrete Implementor）：

HouseDesign.java

```
public class HouseDesign implements BuildingDesign{
    double width,length;
    int floorNumber;
    HouseDesign(double width,double length,int floorNumber){
        this.width=width;
        this.length=length;
        this.floorNumber=floorNumber;
    }
    public double computerArea(){
        return width*length*floorNumber;
    }
}
```

Bridge Pattern结构描述和使用

- 如果具体实现者HouseDesign类决定将面积的计算加上一个额外的值，即修改了computerArea()方法，那么并不影响到细化抽象者的代码
- 如果抽象者决定增加一个参与计算的参数：adjust，即细化抽象者修改代码，在计算成本时通过设置该参数的值来计算成本，那么并不影响实现着者的代码

Bridge Pattern

- 分离了实现与抽象
  - 使得抽象和实现可以独立的扩展
  - 当修改实现的代码时，不影响抽象的代码，反之也一样
- 满足开闭-原则
  - 抽象和实现者处于同层次，使得系统可独立地扩展这两个层次

8. Composite Pattern -合成模式

- 将对象组合成树形结构以表示“部分-整体”的层次结构
- 使得用户对单个对象和组合对象的使用具有一致性
  - 通过一些基本图像元素（直线、园等）以及一些复合图像元素（由基本图像元素组合而成）构建出复杂的图形
  - 奶奶讲的故事：从前有个山，山里有个庙，庙里有个老和尚在给小和尚讲故事，讲的什么故事呢？从前有个山，山里有个庙……

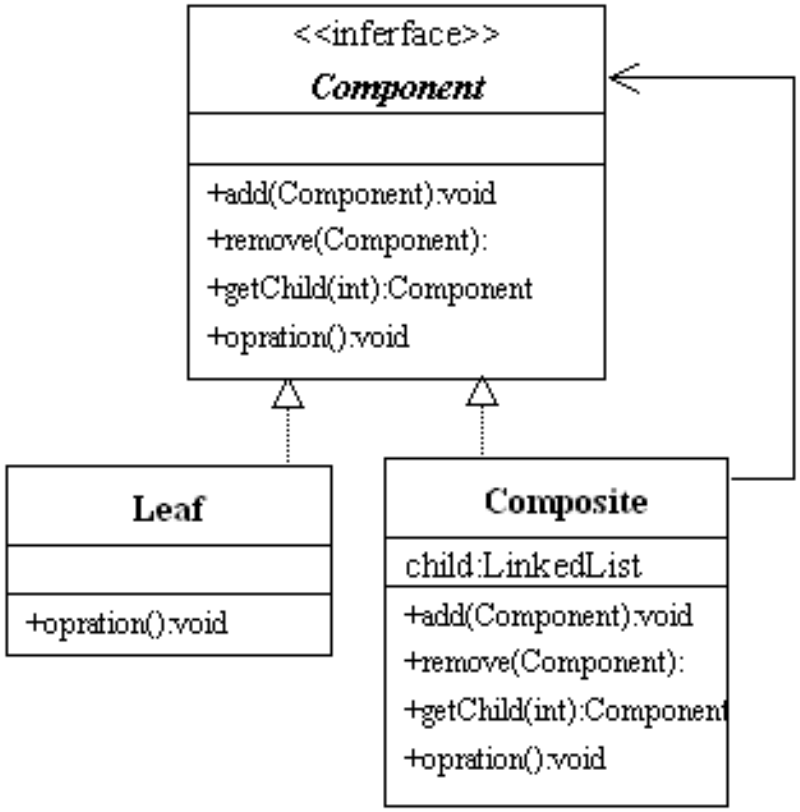
Composite Pattern实例

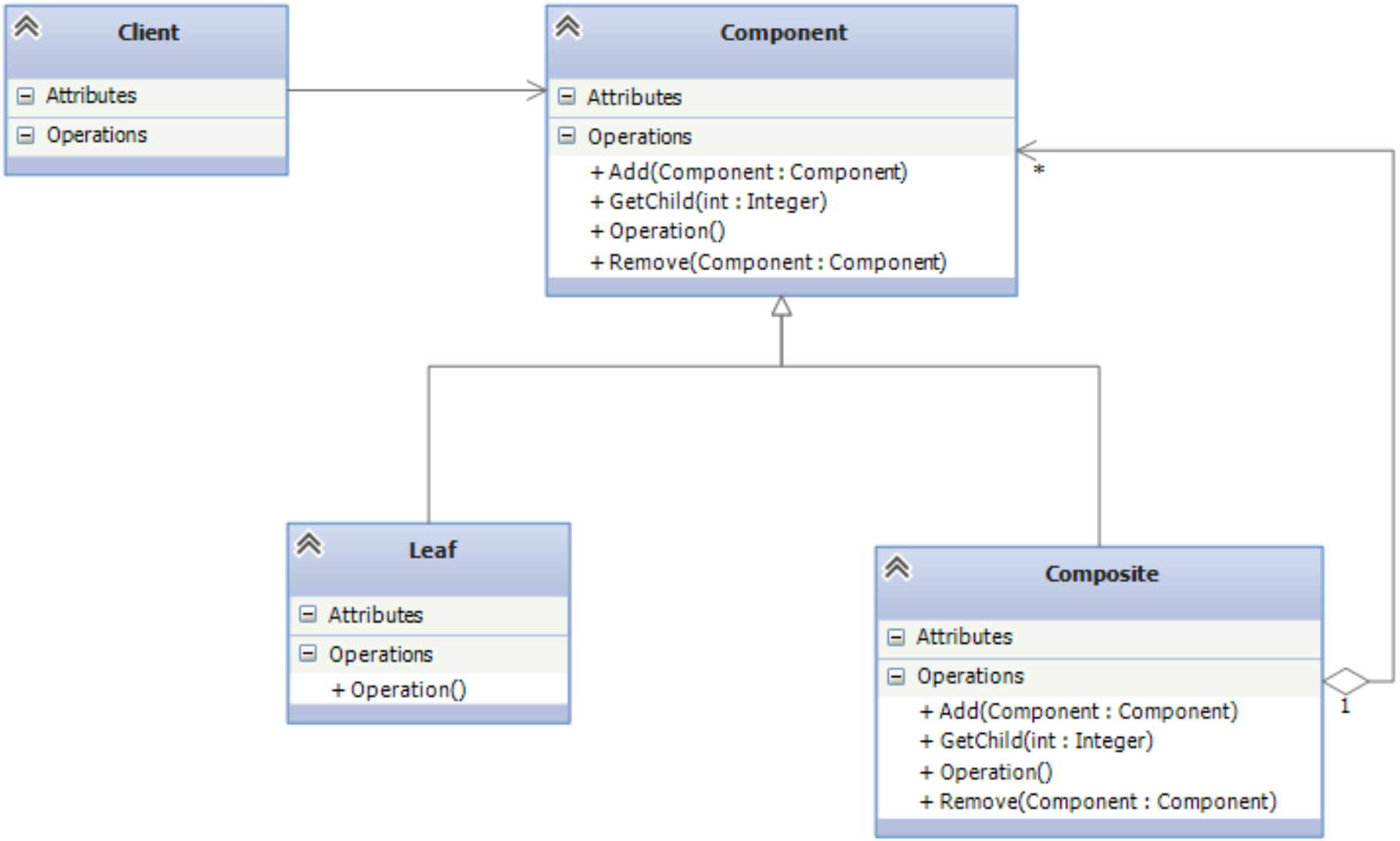
- 算术表达式包括操作数、操作符和另一个操作数，其中，另一个操作符也可以是操作树、操作符和另一个操作数
- 在JAVA AWT 和 Swing 中，对于 Button 和 Checkbox 是树叶，Container 是树枝

Composite Pattern结构和使用

- 模式的结构中包括三种角色：
  - 抽象组件（Component）
  - Composite节点（Composite Node）
  - Leaf节点（Leaf Node）

Composite Pattern




Client
Attributes
Operations
Component
Attributes
Operations
+ Add(Component : Component)
+ GetChild(int : Integer)
+ Operation()
+ Remove(Component : Component)
*
Leaf
Attributes
Operations
+ Operation()
Composite
Attributes
Operations
+ Add(Component : Component)
+ GetChild(int : Integer)
+ Operation()
+ Remove(Component : Component)
1

Composite Pattern结构描述和使用

1．抽象组件（Component）：

```
//MilitaryPerson.java
import java.util.*;
public interface MilitaryPerson{
  public void add(MilitaryPerson person);
  public void remove(MilitaryPerson person);
  public MilitaryPerson getChild(int index);
  public Iterator<MilitaryPerson>  getAllChildren();
  public boolean isLeaf();
  public double getSalary();
  public void setSalary(double salary);
}
```

Composite Pattern结构描述和使用

2. Composite节点（Composite Node）：

```
//MilitaryOfficer.java
import java.util.*;
public class MilitaryOfficer implements MilitaryPerson{
  LinkedList<MilitaryPerson> list;
  String name;
  double salary;
  MilitaryOfficer(String name,double salary){
    this.name=name;
    this.salary=salary;
    list=new LinkedList<MilitaryPerson>();
  }
  public void add(MilitaryPerson person) {  list.add(person);}
  public void remove(MilitaryPerson person){  list.remove(person);}
  public MilitaryPerson getChild(int index) { return list.get(index);}
  public Iterator<MilitaryPerson>  getAllChildren() {
    return list.iterator();
  }
  public boolean isLeaf(){  return false;}
  public double getSalary(){  return salary;}
  public void setSalary(double salary){ this.salary=salary;}
}
```

Composite Pattern结构描述和使用

3. Leaf节点（Leaf Node）：

```
//MilitarySoldier.java
import java.util.*;
public class MilitarySoldier implements MilitaryPerson{
  double salary;
  String name;
  MilitarySoldier(String name,double salary){
    this.name=name;
    this.salary=salary;
  }
  public void add(MilitaryPerson person)  {}
  public void remove (MilitaryPerson person){}
  public MilitaryPerson getChild(int index) {return null;}
  public Iterator<MilitaryPerson>  getAllChildren() {return null;}
  public boolean isLeaf(){return true;}
  public double getSalary(){return salary;}
  public void setSalary(double salary){this.salary=salary;}
}
```

Composite Pattern的优点

- 包含有个体对象和组合对象，并形成树形结构，使用户可方便地处理个体对象和组合对象
- 合成对象和个体对象实现了相同的接口，用户一般不需区分个体对象和组合对象
- 当增加新的Composite节点和Leaf节点时，用户的重要代码不需要作出修改

9. Decorator Pattern - 装饰模式

- 动态地给对象添加一些额外的职责
  - 就功能来说装饰模式相比生成子类更为灵活

Decorator Pattern概述

- 以对客户端透明的方式扩展对象的功能
  - 是继承关系的一个替代方案
  - 提供比继承更多的灵活性
- 动态给一个对象增加功能
  - 这些功能可以再动态的撤消
  - 增加由一些基本功能的排列组合而产生的非常大量的功能

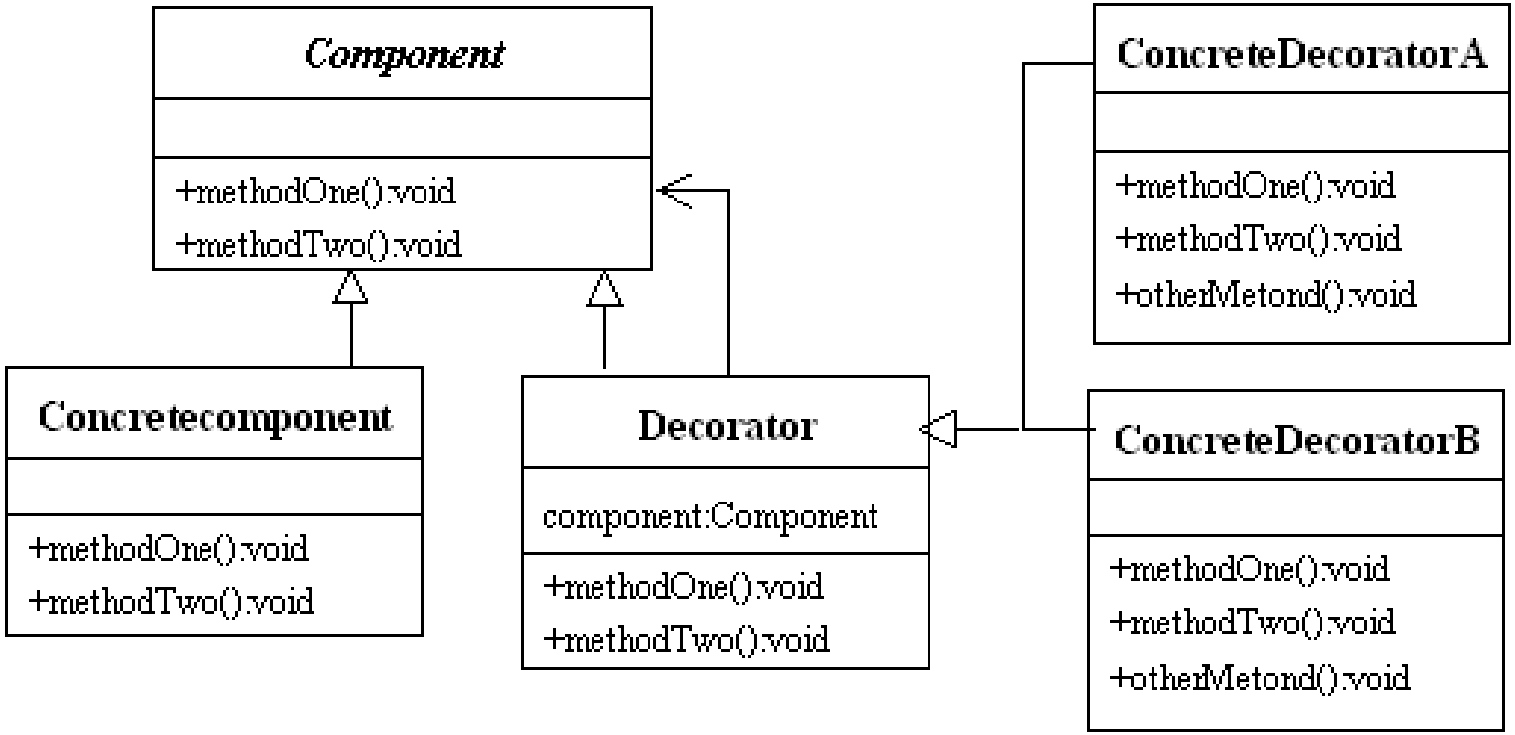
Decorator Pattern实例

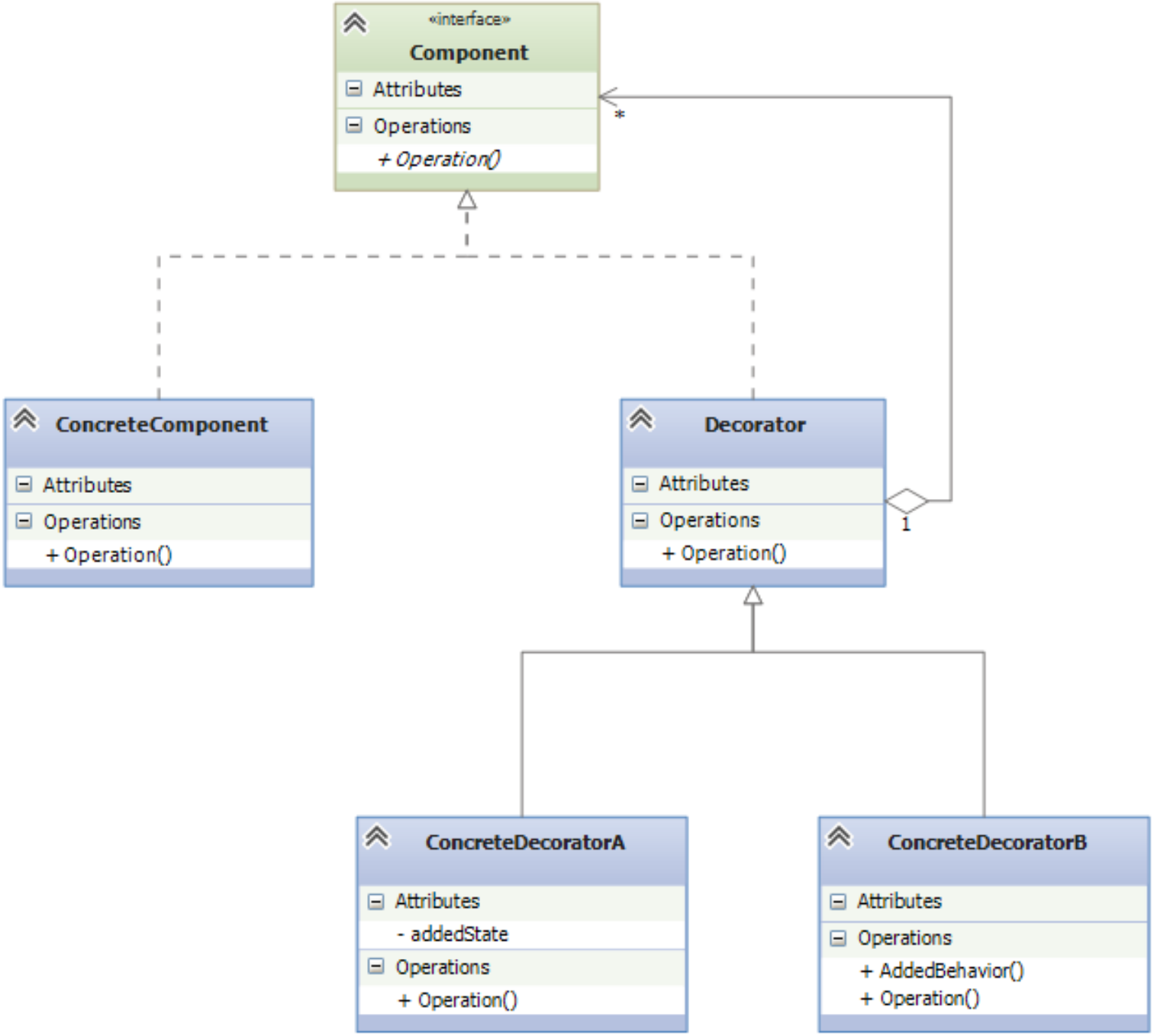
- 假设系统中有一个Bird抽象类以及Bird类的一个子类：Sparrow
- Sparrow类实现了Bird类的fly方法，使得Sparrow类创建的对象调用fly方法能连续飞行100米
- 现在用户需要两种鸟，必须分别能连续飞行150米和200米

Decorator Pattern结构和使用

- 装饰模式的结构中包括四种角色：
  - 抽象组件（Component）
  - 具体组件（ConcreteComponent）
  - 装饰（Decorator）
  - 具体装饰（ConcreteDecotator）

Decorator Pattern




«interface»
Component
Attributes
Operations
+ Operation()
*
ConcreteComponent
Attributes
Operations
+ Operation()
Decorator
Attributes
Operations
+ Operation()
1
ConcreteDecoratorA
Attributes
- addedState
Operations
+ Operation()
ConcreteDecoratorB
Attributes
Operations
+ AddedBehavior()
+ Operation()

Decorator Pattern结构描述和使用

1．抽象组件 ：Bird.java

```
public abstract class Bird{
    public abstract int fly();
}
```

Decorator Pattern结构描述和使用

2. 具体组件 ： Sparrow.java

```
public class Sparrow extends Bird{
        public final int DISTANCE=100;
        public int fly(){
                return DISTANCE;
        }
}
```

Decorator Pattern结构描述和使用

3. 装饰 （Decorator）：Decorator.java

```
public abstract class Decorator extends Bird{
    protected Bird bird;
    public Decorator(){
    }
    public Decorator(Bird bird){
        this.bird=bird;
    }
}
```

Decorator Pattern结构描述和使用

4. 具体装饰（ConcreteDecotator）：SparrowDecorator.java

```
public class  SparrowDecorator extends Decorator{
    public final int DISTANCE=50;//eleFly方法能飞50米
    SparrowDecorator(Bird bird){
        super(bird);
    }
    public int fly(){
        int distance=0;
        distance=bird.fly()+eleFly();
        return distance;
    }
    private int eleFly(){//装饰者新添加的方法
        return DISTANCE;
    }
}
```

Decorator Pattern优点

- 被装饰者和装饰者是松耦合关系
  - 由于装饰（Decorator）仅仅依赖于抽象组件（Component），因此具体装饰只知道它要装饰的对象是抽象组件的某一个子类的实例，但不需要知道是哪一个具体子类
- 装饰模式满足“开-闭原则”
  - 不必修改具体组件，就可以增加新的针对该具体组件的具体装饰
- 可以使用多个具体装饰来装饰具体组件的实例

10. Façade Pattern –门面模式

- 外部与一个子系统的通信必须通过一个统一的门面对象进行
- 提供一个高层次的接口，使得子系统更易于使用
- 每一个子系统只有一个门面类，而且此门面类只有一个实例，也就是说它是一个单例模式，但整个系统可以有多个门面类

Façade Pattern概述

- 是简化用户和子系统进行交互的模式
- 为子系统提供一个称作外观的类
  - 该门面类的实例负责和子系统中类的实例打交道
  - 当用户想要和子系统中的若干个类的实例打交道时，可代替地和子系统的门面类的实例打交道

Façade Pattern实例

- 邮政系统负责邮寄包裹的子系统包含Check、Weight和Transport类
  - Check类的实例负责对包裹进行安全检查
  - Weight类的实例负责根据包裹的重量计算邮资
  - Transport类的实例负责为包裹选择运输工具
- 一个要邮寄的包裹的用户如果直接和负责邮寄包裹的子系统的类打交道就会非常不方便
  - 解决方法：建立门面类，按需与相关的子系统交互
  - 提高交互效率

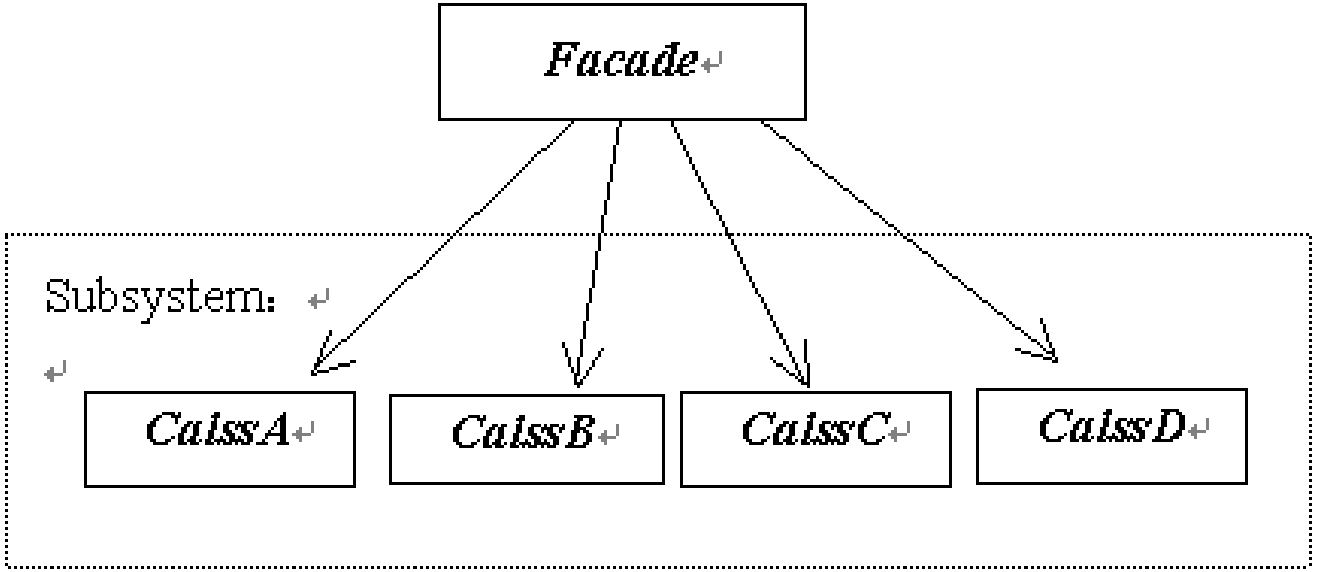
Façade Pattern实例

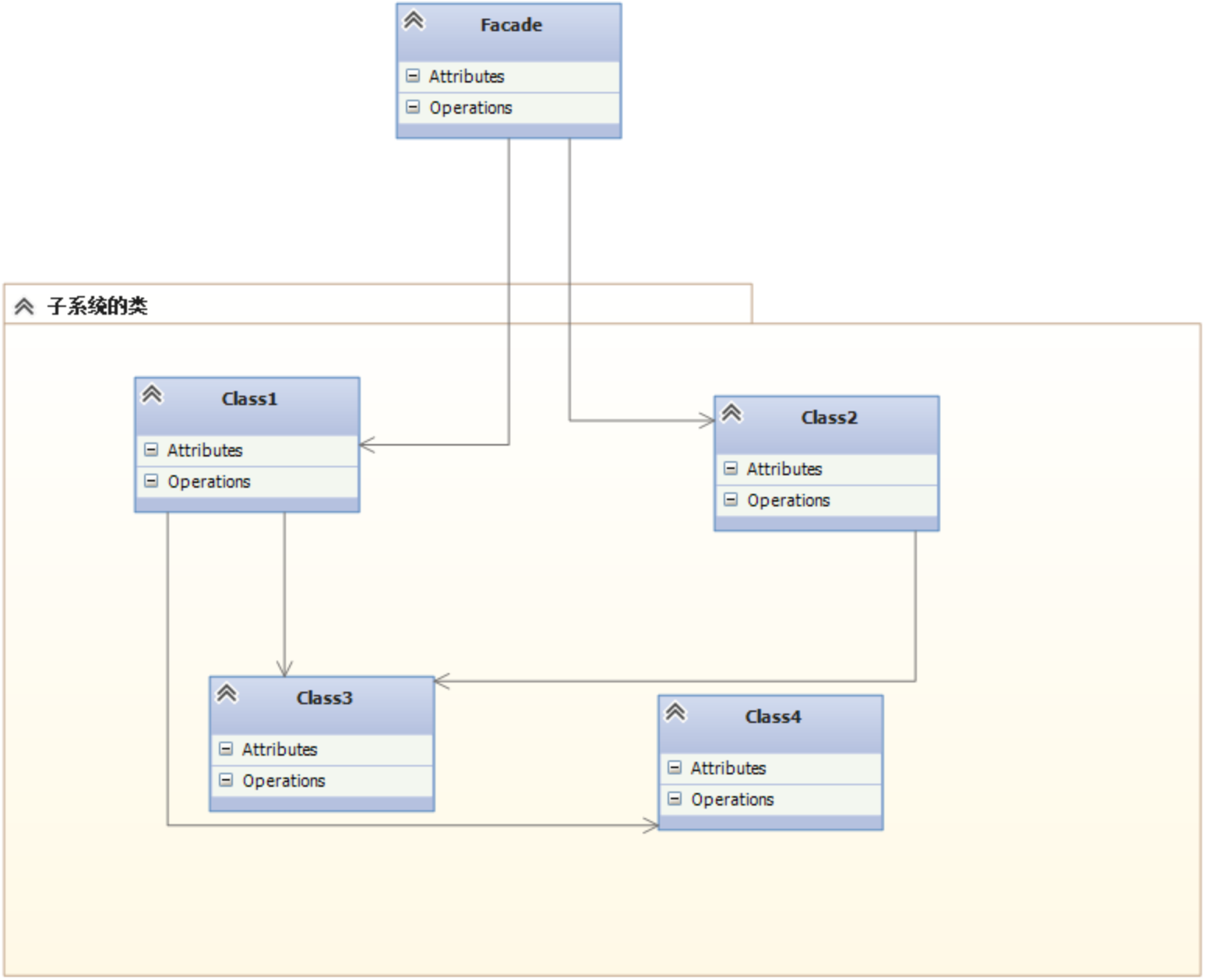
- 报社的广告系统有三个类CheckWord、Charge和TypeSetting类，各个类的职责如下：
- CheckWord类负责检查广告内容含有的字符数量；Charge类的实例负责计算费用；TypeSetting的实例负责对广告进行排版。
- 使用门面模式简化用户和上述子系统所进行的交互。

Façade Pattern结构与使用

- 模式的结构中包括两种角色：
  - 子系统（Subsystem）
  - 外观（Facade）

Façade Pattern




Facade
Attributes
Operations
子系统的类
Class1
Attributes
Operations
Class2
Attributes
Operations
Class3
Attributes
Operations
Class4
Attributes
Operations

Façade Pattern结构描述与使用

- 访问数据库：

```
public class DBCompare {
  String sql = "SELECT * FROM <table> WHERE <column name> = ?";
  try {
     Mysql msql=new mysql(sql);
     prep.setString( 1, "<column value>" );
     rset = prep.executeQuery();
     if( rset.next() ) {
        System.out.println( rset.getString( "<column name" ) );
    }
  }
  catch( SException e ) {e.printStackTrace();}
  finally {
    mysql.close(); mysql=null;
  }
}
```

Façade Pattern优点

- 使客户和子系统中的类无耦合
- 外观只是提供了一个更加简洁的界面，并不影响用户直接使用子系统中的类
- 子系统中任何类对其方法的内容进行修改，不影响外观的代码

11. Flyweight Pattern - 享元模式

- Flyweight在拳击比赛中是指最轻量级
- 享元模式以共享的方式高效的支持大量的细粒度对象

Flyweight Pattern概述

- 享元模式能做到共享的关键是区分内蕴状态和外蕴状态。
  - 内蕴状态，存储在享元内部，不会随环境的改变而有所不同
  - 外蕴状态是随环境的改变而改变的。外蕴状态不能影响内蕴状态，它们是相互独立的
- 客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象
- 可以大幅度的降低内存中对象的数量

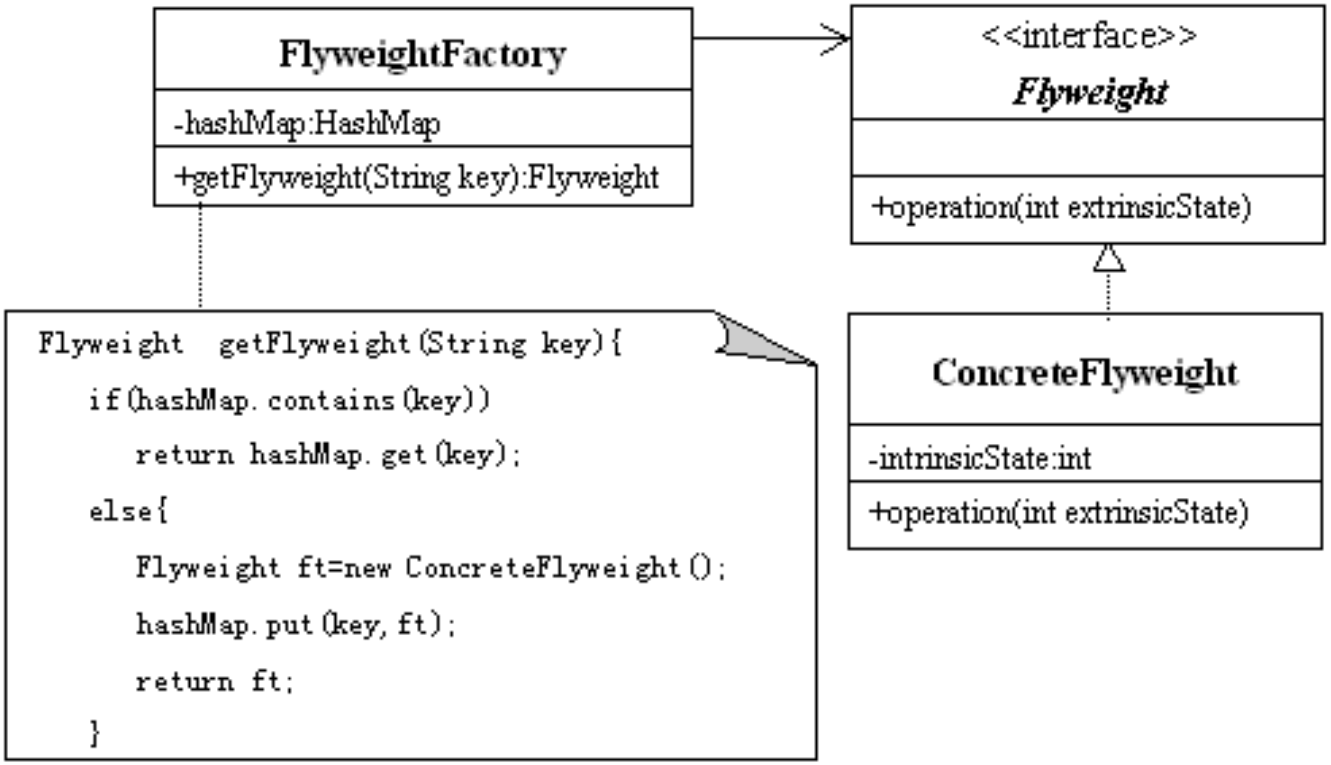
Flyweight Pattern实例

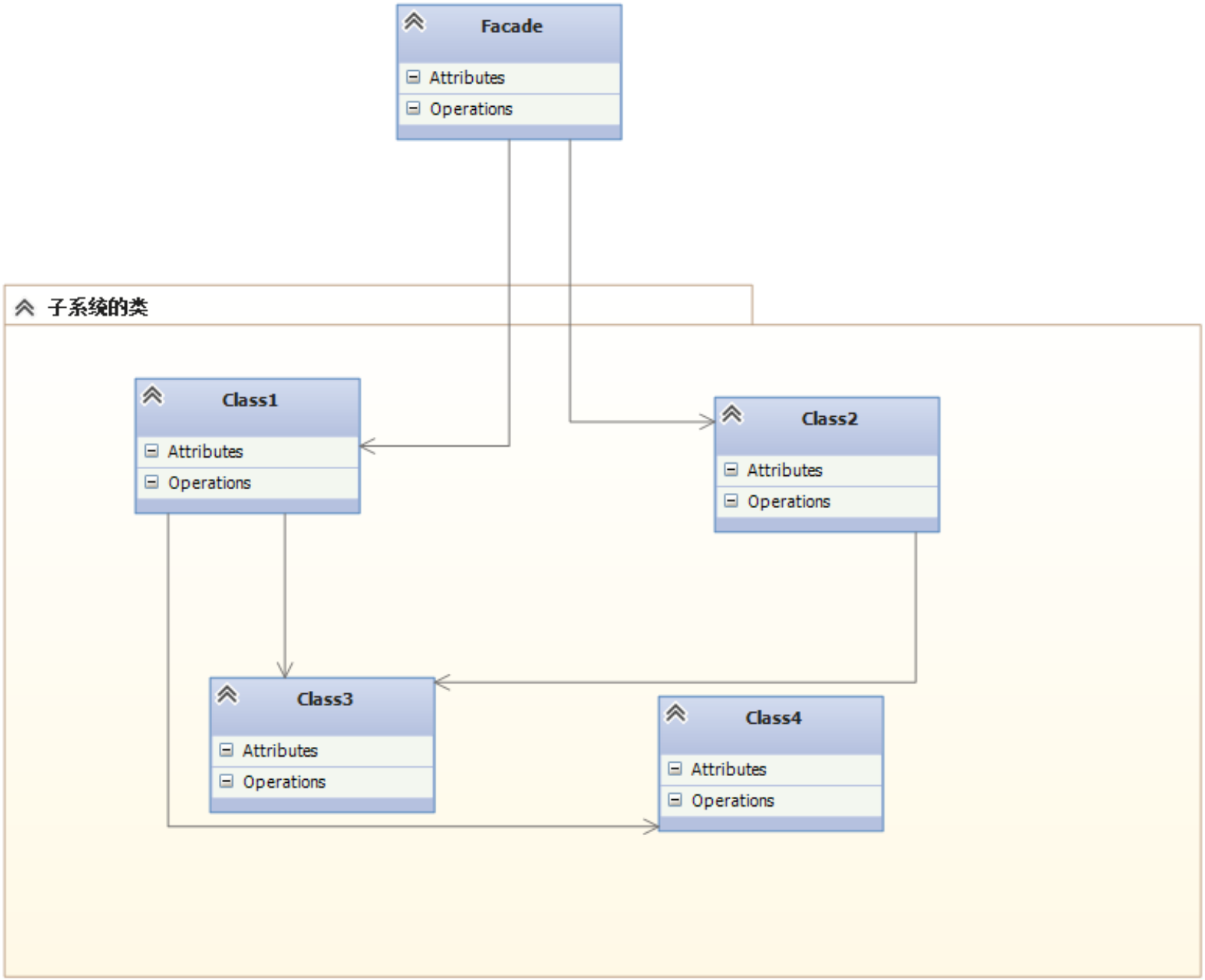
- JAVA 中的 String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面
- 数据库的数据池

Flyweight Pattern结构与使用

- 模式的结构中包括三种角色：
  - 享元接口（Flyweight）
  - 具体享元（Concrete Flyweight）
  - 享元工厂（Flyweight Factory）

Flyweight Pattern




Facade
Attributes
Operations
子系统的类
Class1
Attributes
Operations
Class2
Attributes
Operations
Class3
Attributes
Operations
Class4
Attributes
Operations

Flyweight Pattern结构描述与使用

1．享元接口（Flyweight）：Flyweight.java

```
public interface Flyweight{
        public double getHeight();
        public double getWidth();
        public double getLength();
        public void printMess(String mess);
}
```

Flyweight Pattern结构描述与使用

2．享元工厂与具体享元_1：

```
//FlyweightFactory.java
import java.util.HashMap;
public class FlyweightFactory{
  private   HashMap<String,Flyweight>  hashMap;
  static  FlyweightFactory factory=new FlyweightFactory();
  private FlyweightFactory(){
    hashMap=new HashMap<String,Flyweight>();
  }
  public static FlyweightFactory getFactory(){return factory;}
  public synchronized Flyweight getFlyweight(String key){
    if(hashMap.containsKey(key)) return hashMap.get(key);
    else{
      double width=0,height=0,length=0;
      String [] str=key.split("#");
      width=Double.parseDouble(str[0]);
      height=Double.parseDouble(str[1]);
      length=Double.parseDouble(str[2]);
      Flyweight ft=new ConcreteFlyweight(width,height,length);
      hashMap.put(key,ft);
      return ft;
      }
    }
}
```

Flyweight Pattern结构描述与使用

2. 享元工厂与具体享元_2:

```
class ConcreteFlyweight implements Flyweight{
  private double width;
  private double height;
  private double length;
  private ConcreteFlyweight(double width,double height,double length){
    this.width=width;
    this.height=height;
    this.length=length;
  }
  public double getHeight(){return height;}
  public double getWidth(){return width;}
  public double getLength(){return length;}
  public void printMess(String mess){
    System.out.print(mess);
    System.out.print(" 宽度: "+width);
    System.out.print(" 高度: "+height);
    System.out.println("长度: "+length);
    }
  }
}
```

Flyweight Pattern结构描述与使用

3. 应用_1： Car.java

```
public class Car{
      Flyweight  flyweight;
      String name,color;
      int power;
      Car(Flyweight flyweight,String name,String color,int power){
            this.flyweight=flyweight;
            this.name=name;
            this.color=color;
            this.power=power;
      }
     public void print(){
            System.out.print(" 名称: "+name);
            System.out.print(" 颜色: "+color);
            System.out.print(" 功率: "+power);
            System.out.print(" 宽度: "+flyweight.getWidth());
            System.out.print(" 高度: "+flyweight.getHeight());
            System.out.println("长度: "+flyweight.getLength());
      }
}
```

Flyweight Pattern结构描述与使用

3. 应用_2: Application.java

```
public class Application{
    public static void main(String args[]) {
        FlyweightFactory  factory=FlyweightFactory.getFactory();
        double width=1.82,height=1.47,length=5.12;
        String key=""+width+"#"+height+"#"+length;
        Flyweight flyweight=factory.getFlyweight(key);
        Car audiA6One=new Car(flyweight,"奥迪A6","黑色",128);
        Car audiA6Two=new Car(flyweight,"奥迪A6","灰色",160);
        audiA6One.print();
        audiA6Two.print();
        width=1.77;
        height=1.45;
        length=4.63;
        key=""+width+"#"+height+"#"+length;
        flyweight=factory.getFlyweight(key);
        Car audiA4One=new Car(flyweight,"奥迪A4","蓝色",126);
        Car audiA4Two=new Car(flyweight,"奥迪A4","红色",138);
        flyweight.printMess(" 名称：奥迪A4 颜色：蓝色 功率：126");
        flyweight.printMess(" 名称：奥迪A4 颜色：红色 功率：138");
    }
}
```

Flyweight Pattern优点

- 使用享元可以节省内存的开销，特别适合处理大量细粒度对象，这些对象的许多属性值是相同的，而且一旦创建则不容许修改
- 享元可以使用方法的参数接受外部状态中的数据，但外部状态数据不会干扰到享元中的内部数据，这就使得享元可以在不同的环境中被共享

12. Proxy Pattern -代理模式

- 给某一个对象提供一个代理对象，并由代理对象控制对源对象的引用
- 代理就是一个人或一个机构代表另一个人或者一个机构采取行动

Proxy Pattern概述

- 某些情况下，客户不想或者不能够直接引用一个对象，代理对象在客户和目标对象之间起到中介的作用
- 客户端分辨不出代理主题对象与真实主题对象
  - 代理模式可以并不知道真正的被代理对象，而仅仅持有一个被代理对象的接口
  - 代理对象不能够创建被代理对象，被代理对象必须有系统的其他角色代为创建并传入

Proxy Pattern实例

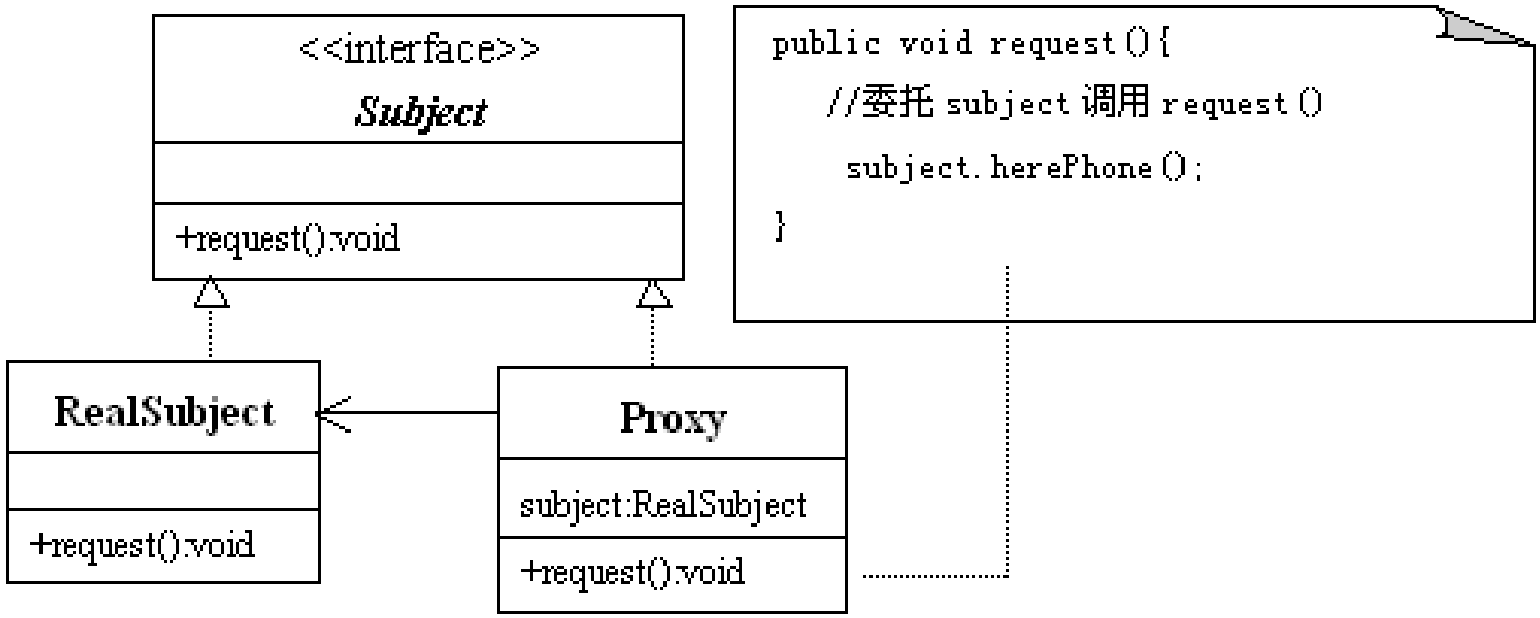
- Windows 里面的快捷方式
- 用户输入三个代表三角形三边长度的数值，代理对象验证用户输入的三个数值是否能构成三角形，如果能构成三角形，就创建一个三角形对象，并让三角形对象计算自身的面积
- 一些平台系统中针对不同用户的界面，通过代理链接View层和 Controller层的实体

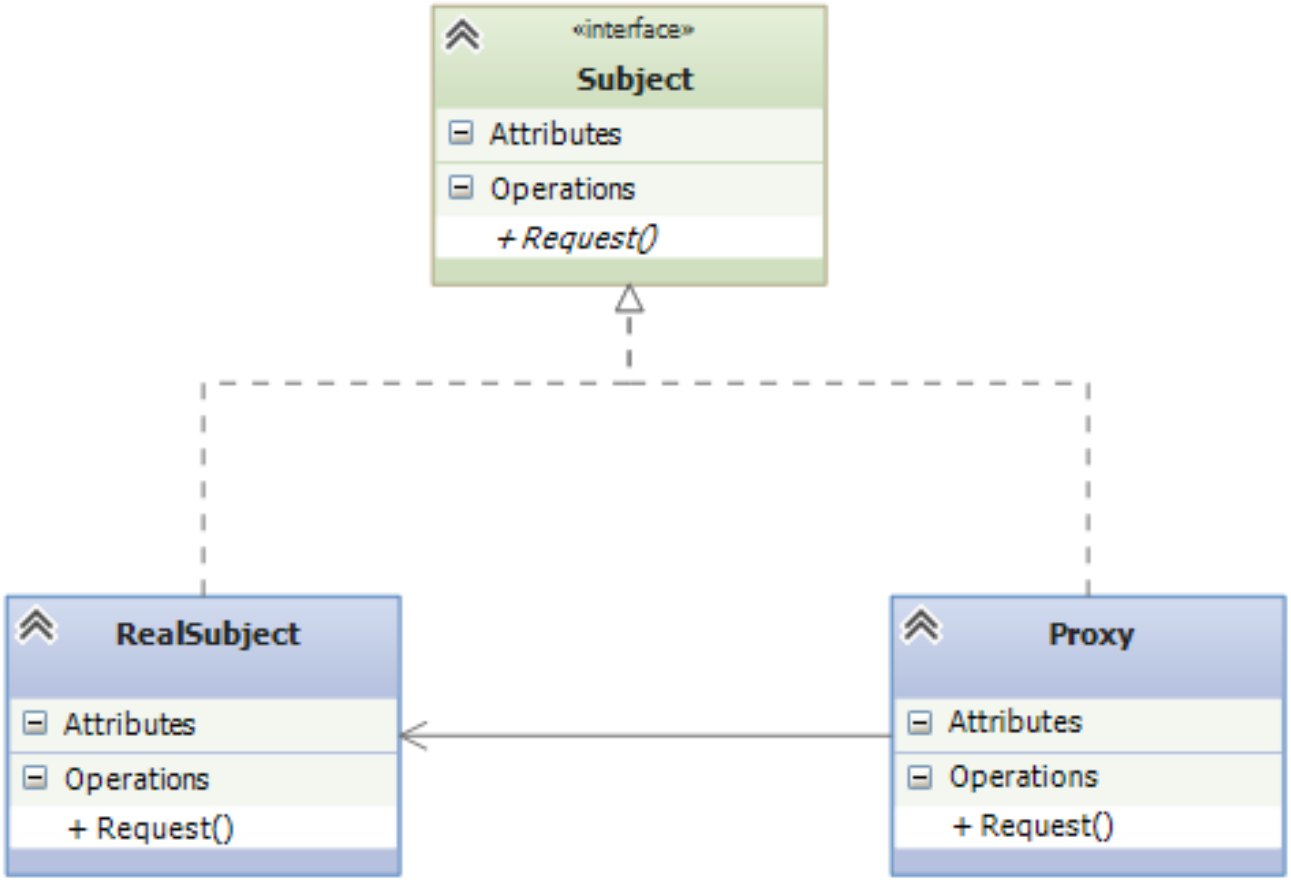
Proxy Pattern结构与使用

- 模式的结构中包括三种角色：

- 抽象主题（Subject）
- 实际主题（RealSubject）
- 代理（Proxy）

Proxy Pattern





为其他对象提供一种代理以控制对这个对象的访问

Proxy Pattern结构描述与使用

1. 抽象主题（Subject）：Geometry.java

```
public  interface Geometry{
        public double getArea();
 }
```

Proxy Pattern结构描述与使用

2. 具体模板（Concrete Template）：

Trangle.java

```
public class Triangle implements Geometry{
        double sideA,sideB,sideC,area;
        public  Triangle(double a,double b,double c) {
                sideA=a;
                sideB=b;
                sideC=c;
        }
        public double  getArea(){
                double p=(sideA+sideB+sideC)/2.0;
                area=Math.sqrt(p*(p-sideA)*(p-sideB)*(p-sideC)) ;
                return area;
        }
}
```

Proxy Pattern结构描述与使用

3. 代理（Proxy）：TriangleProxy.java

```
public class TriangleProxy implements Geometry{
    double sideA,sideB,sideC;
     Triangle triangle;
    public  void setABC(double a,double b,double c) {
            sideA=a;
            sideB=b;
            sideC=c;
     }
     public double  getArea(){
            if(sideA+sideB>sideC&&sideA+sideC>sideB&&sideB+sideC>sideA){
                triangle=new Triangle(sideA,sideB,sideC);
                double area=triangle.getArea();
                return area;
            }
            else
                return -1;
    }
}
```

Proxy Pattern结构描述与使用

4. 应用 Application.java

```
import java.util.Scanner;
public class Application{
	public static void main(String args[]) {
		Scanner reader=new Scanner(System.in);
		System.out.println("请输入三个数，每输入一个数回车确认");
		double a=-1,b=-1,c=-1;
		a=reader.nextDouble();
		b=reader.nextDouble();
		c=reader.nextDouble();
		TriangleProxy proxy=new TriangleProxy();
		proxy.setABC(a,b,c);
		double area=proxy.getArea();
		System.out.println("面积是: "+area);
	}
}
```

Proxy Pattern优点

- 可以屏蔽用户真正请求的对象，使用户程序和真正的对象之间解耦
- 使用代理来担当那些创建耗时的对象的替身

三 行为模式

- Chain of Responsibility Pattern－责任链
- Command Pattern － 命令模式
- Interpreter Pattern － 解释器模式
- Iterator Pattern － 迭代子模式
- Mediator Pattern － 调停者模式
- Memento Pattern － 备忘录模式
- Observer Pattern － 观察者模式
- State Pattern － 状态模式
- Strategy Pattern － 策略模式
- Template Method Pattern － 模板方法模式
- Visitor Pattern － 访问者模式

13. Chain of Responsibility Pattern 责任链模式

- 责任链模式：避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止
- 由于英文翻译的不同，职责链模式又称为责任链模式，它是一种对象行为型模式

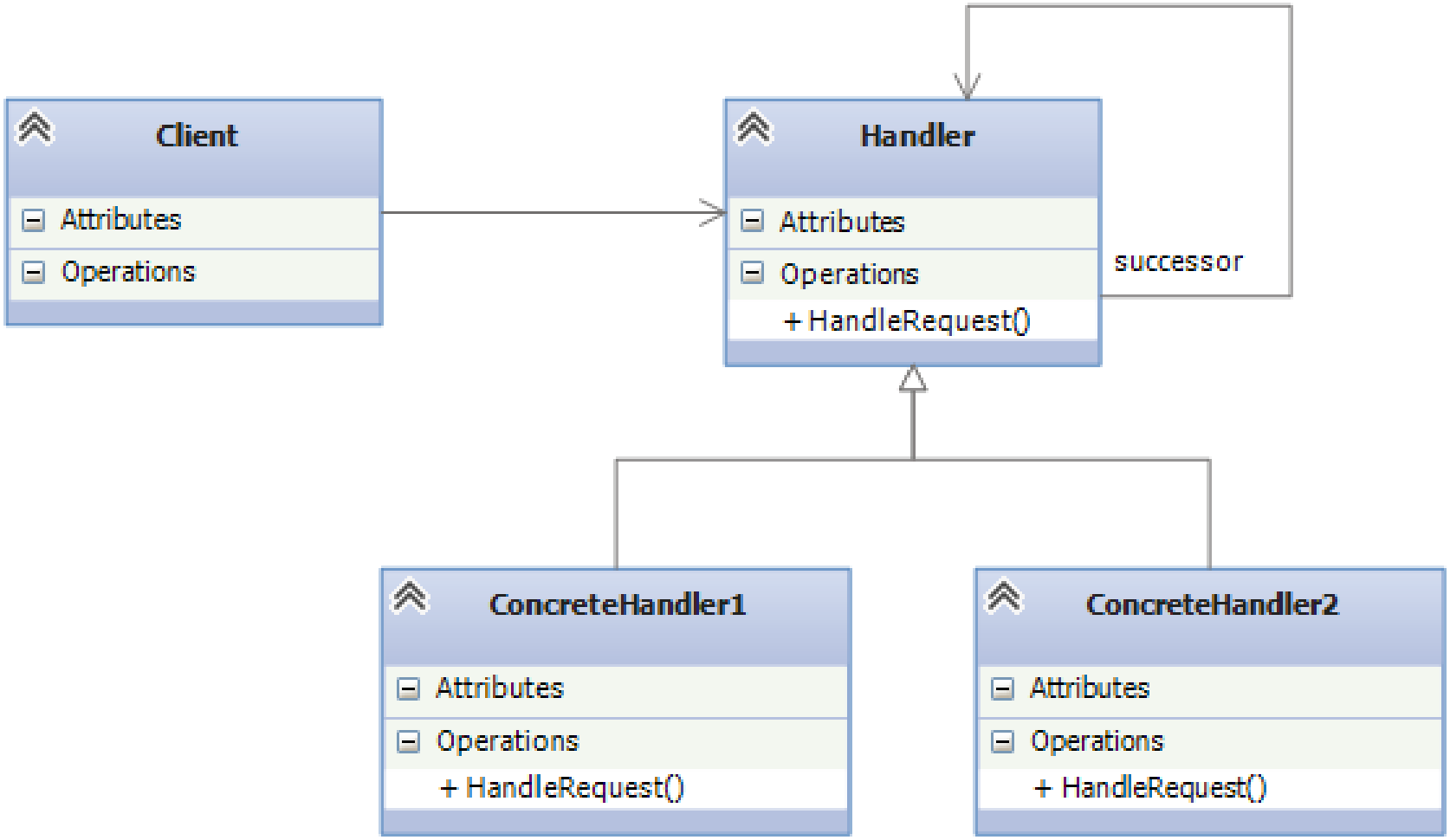
Chain of Responsibility Pattern

- 红楼梦中的"击鼓传花"
- JAVA WEB 中 Apache Tomcat 对 Encoding 的处理，Struts2 的拦截器， JSP srvlet 的 Filter

Chain of Responsibility Pattern

- 模式的结构中包括三种角色：
- Handler：抽象处理者
- ConcreteHandler：具体处理者
- Client：客户类

Chain of Responsibility Pattern



Chain of Responsibility Pattern

1．抽象处理者:

```
public abstract class Handler
{
  protected Handler successor;

  public void setSuccessor(Handler successor)
  {
    this.successor=successor;
  }

  public abstract void handleRequest(String request);
}
```

Chain of Responsibility Pattern

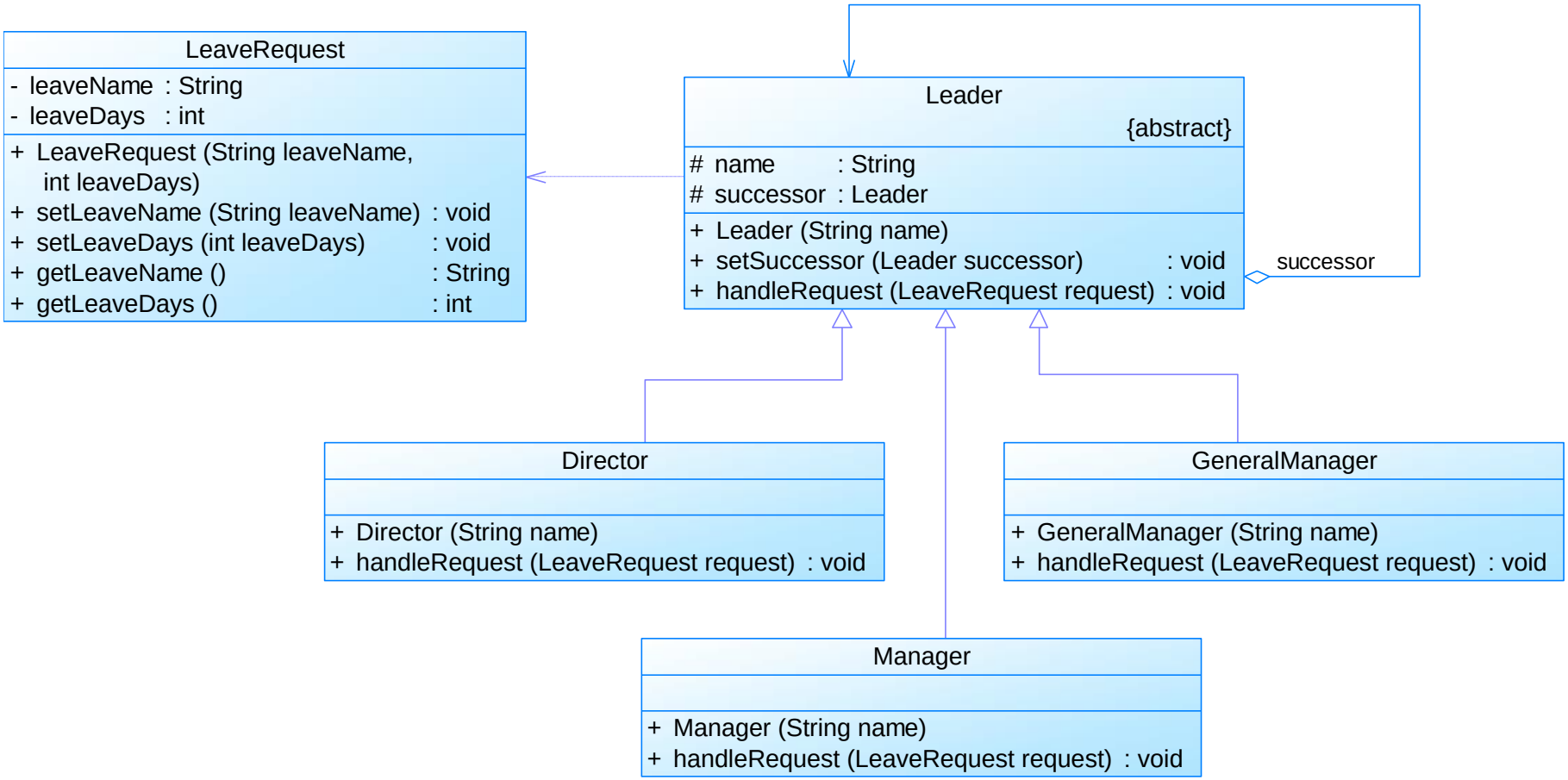
2. 典型的具体处理者代码

```
public class ConcreteHandler extends Handler
{
  public void handleRequest(String request)
  {
    if(请求request满足条件)
    {
      ......  //处理请求;
    }
    else
    {
      this.successor.handleRequest(request); //转发请求
    }
  }
}
```

Chain of Responsibility Pattern

- 某OA系统需要提供一个假条审批的模块
  - 如果员工请假天数小于3天，主任可以审批该假条
  - 如果员工请假天数大于等于3天，小于10天，经理可以审
  - 如果员工请假天数大于等于10天，小于30天，总经理可以审批
  - 如果超过30天，总经理也不能审批，提示相应的拒绝信息

Chain of Responsibility Pattern



Chain of Responsibility Pattern

- 降低耦合度
- 可简化对象的相互连接
- 增强给对象指派职责的灵活性
- 增加新的请求处理类很方便

14. Command Pattern 命令模式

- 命令模式（Command Pattern）是一种数据驱动的设计模式，它属于行为型模式
- 请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令

Command Pattern实例

- struts1中的action核心控制器 Action-Servlet只有一个，相当于Invoker，而模型层的类会随着不同的应用有不同的模型类，相当于具体的Command

Command Pattern结构和使用

- 模式的结构中包括四种角色：
  - 接收者（Receiver）
  - 命令(Command)接口
  - 具体命令（ConcreteCommand）
  - 请求者（Invoker）

Command Pattern

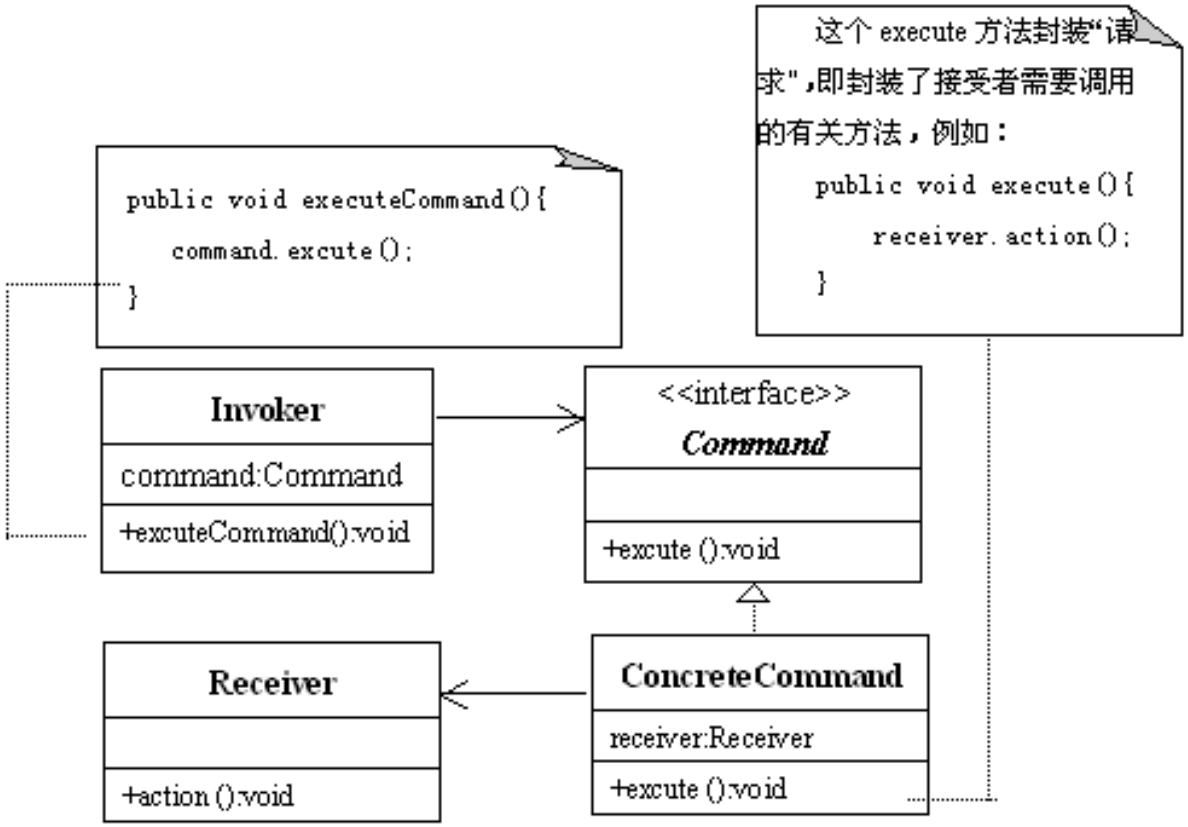
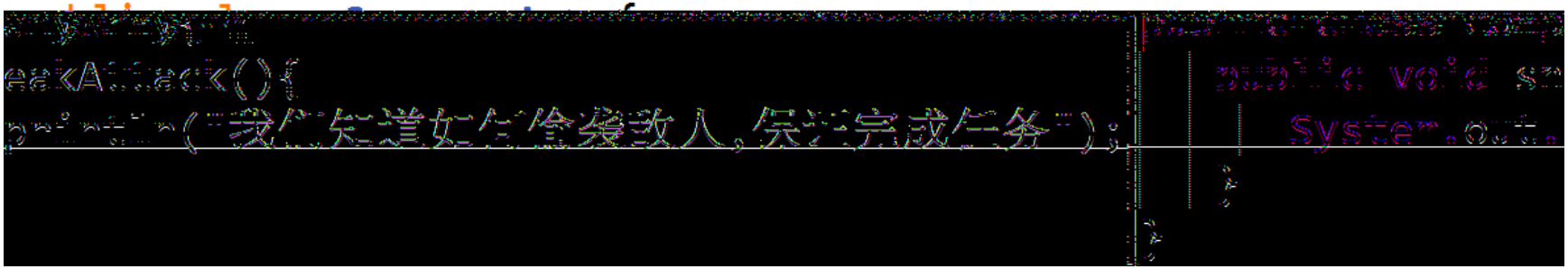


图 4.3 命令模式的类图

Command Pattern结构描述和使用

1．接收者：CompanyArmy.java



Command Pattern结构描述和使用

2. 命令接口 ： Command.java

```
public interface Command {
    public abstract void execute();
}
```

Command Pattern结构描述和使用

3．具体命令：ConcreteCommand.java

```
public class ConcreteCommand implements Command{
    CompanyArmy army;                    //含有接收者的引用
    ConcreteCommand(CompanyArmy army){
        this.army=army;
    }
    public  void execute(){              //封装着指挥官的请求
        army.sneakAttack();              //偷袭敌人
    }
}
```

Command Pattern

Command Pattern结构描述和使用

4. 请求者：ArmySuperior.java

```
public class ArmySuperior{
    Command command;                    //用来存放具体命令的引用
    public void setCommand(Command command){
        this.command=command;
    }
    public void startExecuteCommand(){
                                        //让具体命令执行execute()方法
        command.execute();
    }
}
```

Command Pattern结构描述和使用

5. 应用 Application.java

```
public class Application{
    public static void main(String args[]){
        CompanyArmy 三连=new CompanyArmy();
        Command command=new ConcreteCommand(三连);
        ArmySuperior 指挥官=new ArmySuperior();
        指挥官.setCommand(command);
        指挥官.startExecuteCommand();
    }
}
```

Command Pattern优点

- 在命令模式中，请求者（Invoker）不直接与接收者（Receiver）交互，因此彻底消除了彼此之间的耦合
- 命令模式满足“开-闭原则”
  - 如果增加新的具体命令和该命令的接受者，不必修改调用者的代码，调用者就可以使用新的命令对象
  - 反之，如果增加新的调用者，不必修改现有的具体命令和接受者，新增加的调用者就可以使用已有的具体命令

15. Interpreter Pattern -解释器模式

- 提供了评估语言的语法或表达式的方式，它属于行为型模式
- 该种模式实现了一个表达式接口，该接口解释一个特定的上下文
- 这种模式被用在 SQL 解析、符号处理引擎等

Interpreter Pattern实例

- 编译器、运算表达式计算

Interpreter Pattern结构和使用

- 模式的结构中包括四种角色：

- 抽象表达式（AbstractExpression）
- 终结符表达式（TerminalExpression）
- 非终结符表达式（NonterminalExpression）
- 上下文（Context）

Interpreter Pattern

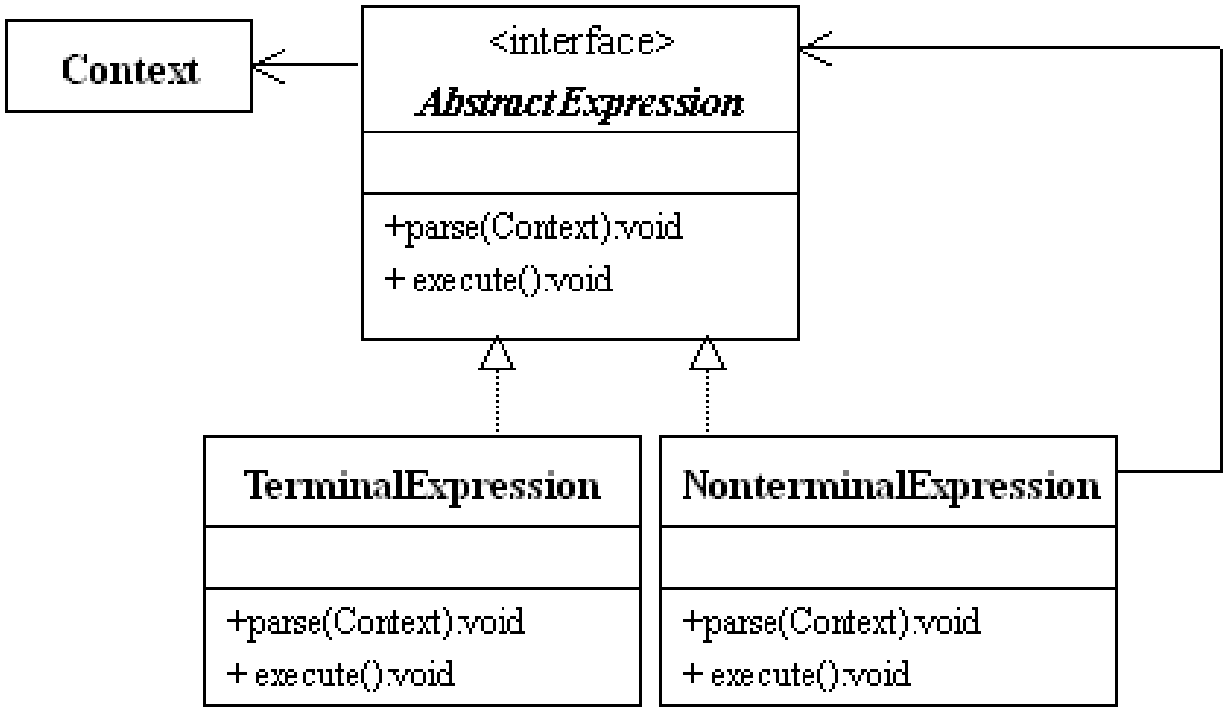
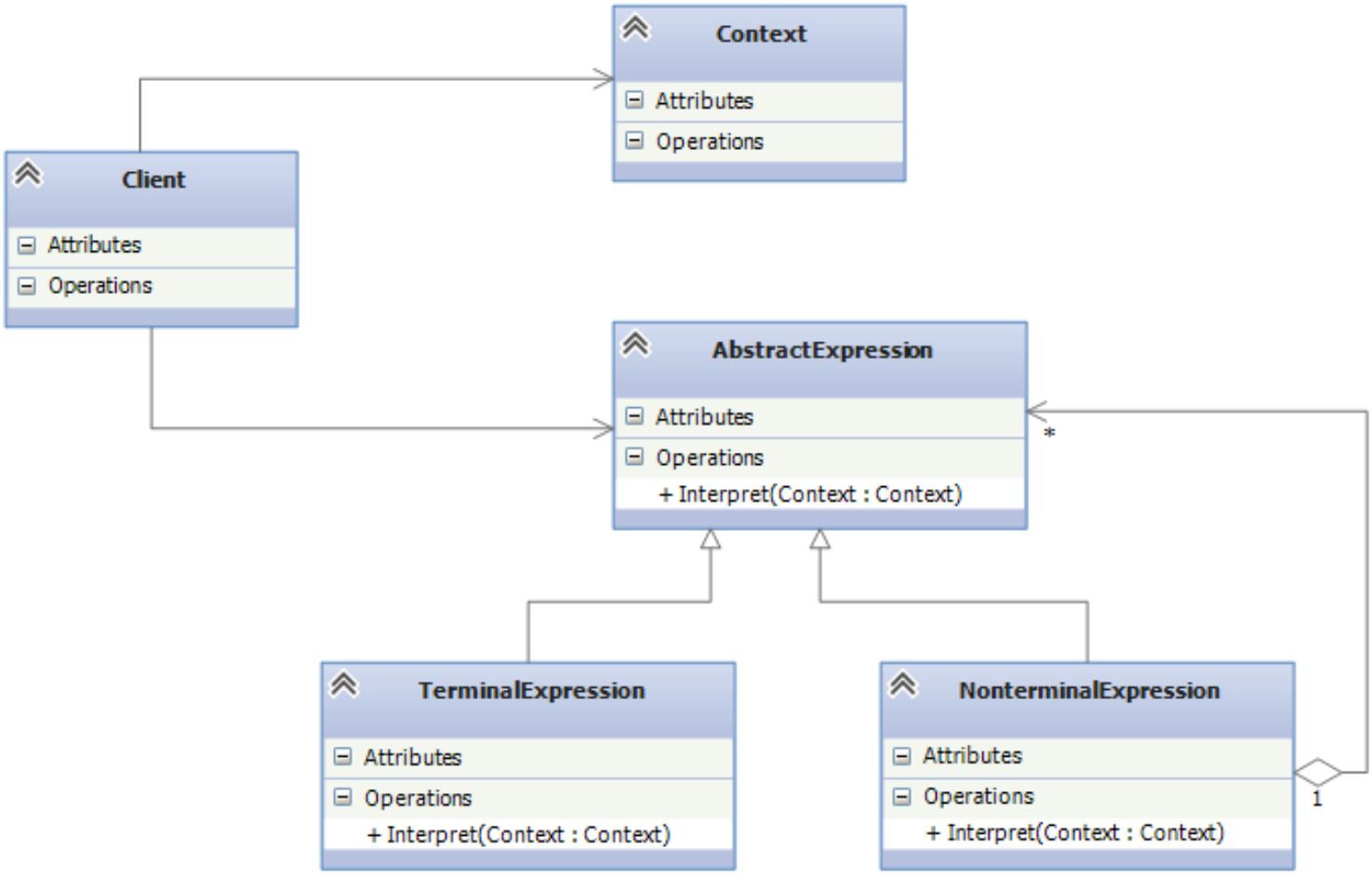


图 26.2　解释器模式的类图



给定一个语言，定义它的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

Interpreter Pattern结构描述和使用

1．抽象表达式：

```
public interface Node{
   public void parse(Context text);
   public void execute();
}
```

Interpreter Pattern结构描述和使用

2. 终结符表达式：

```
public class SubjectPronounOrNounNode implements Node{
  String [] word={"You","He","Teacher","Student"};
  String token;boolean boo;
  public void parse(Context context){
    token=context.nextToken();
    int i=0;
    for(i=0;i<word.length;i++){
      if(token.equalsIgnoreCase(word[i])){ boo=true;break;}}
    if(i==word.length) boo=false;}
  public void execute(){
    if(boo){
      if(token.equalsIgnoreCase(word[0]))
        System.out.pSystem.out.print("你");
      if(token.equalsIgnoreCase(word[1]))
        System.out.print("他");
      if(token.equalsIgnoreCase(word[2]))
        System.out.print("老师");
      if(token.equalsIgnoreCase(word[3]))
        System.out.print("学生");
      }
      else{System.out.print(token+"(不是该语言中的语句)");}
     }
}
```

Interpreter Pattern结构描述和使用

3. 非终结符表达式

```
public class SentenceNode implements Node{
        Node subjectNode,predicateNode;
        public void parse(Context context){
                subjectNode =new SubjectNode();
                predicateNode=new PredicateNode();
                subjectNode.parse(context);
                predicateNode.parse(context);
        }
        public void execute(){
                subjectNode.execute();
                predicateNode.execute();
        }
}
```

Interpreter Pattern描述和使用

4. 上下文：

```
import java.util.StringTokenizer;
public class Context{
  StringTokenizer tokenizer;
  String token;
  public Context(String text){
    setContext(text);
  }
  public void setContext(String text){
    tokenizer=new StringTokenizer(text);
  }
  String nextToken(){
    if(tokenizer.hasMoreTokens()){
      token=tokenizer.nextToken();
  }
  else
    token="";
  return token;
  }
}
```

Interpreter Pattern结构描述和使用

5. 应用 Application.java

```
public class Application{
  public static void main(String args[]){
    String text="Teacher beat tiger";
    Context context=new Context(text);
    Node node=new SentenceNode();
    node.parse(context);
    node.execute();
    text="You eat  apple";
    context.setContext(text);
    System.out.println();
    node.parse(context);
    node.execute();
    text="you look  him";
    context.setContext(text);
    System.out.println();
    node.parse(context);
    node.execute();
    }
}
```

Interpreter Pattern优点

- 将每一个语法规则表示成一个类，方便于实现简单的语言
- 由于使用类表示语法规则，可以较容易改变或扩展语言的行为
- 通过在类结构中加入新的方法，可以在解释的同时增加新的行为

Interpreter Pattern缺点

- 对于可利用场景比较少
- 复杂的文法比较难维护
- 解释器模式会引起类膨胀
- 解释器模式采用递归调用方法

16. Iterator Pattern -迭代器模式

- 迭代器模式（Iterator Pattern）是 在Java和 .Net 编程环境中非常常用的设计模式
- 这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示

Iterator Pattern实例

- 一栋楼中住着张三、李四、刘武三个人，分别被安排在不同的房间中，张三知道李四的房间，李四知道刘武的房间。假设有一个警察，他并不知道这些人是以什么方式在此居住，只想找到他们，那么警察可以使用一个名字为next()的方法找人，找到一个人的同时立刻让这个人说出他所知道下一个人所在的房间。

Iterator Pattern结构和使用

- 模式的结构中包括四种角色：

- 集合（Aggregate）
- 具体集合（ConcreteAggregate）
- 迭代器（Iterator）
- 具体迭代器（ConcreteIterator）

Iterator Pattern

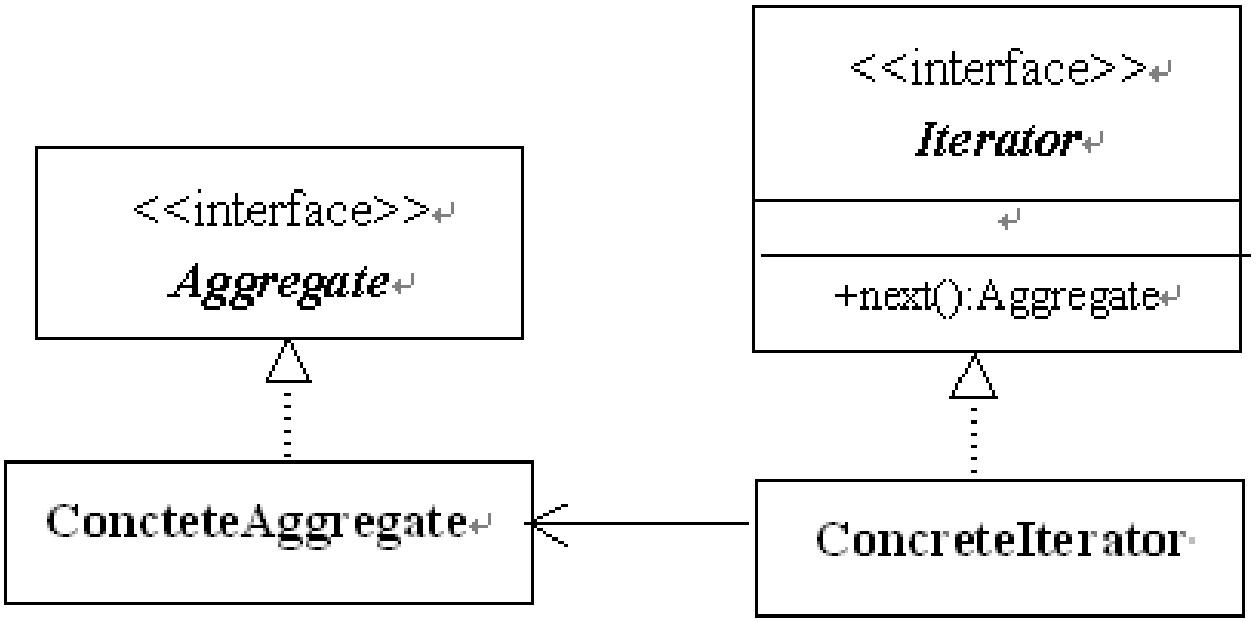
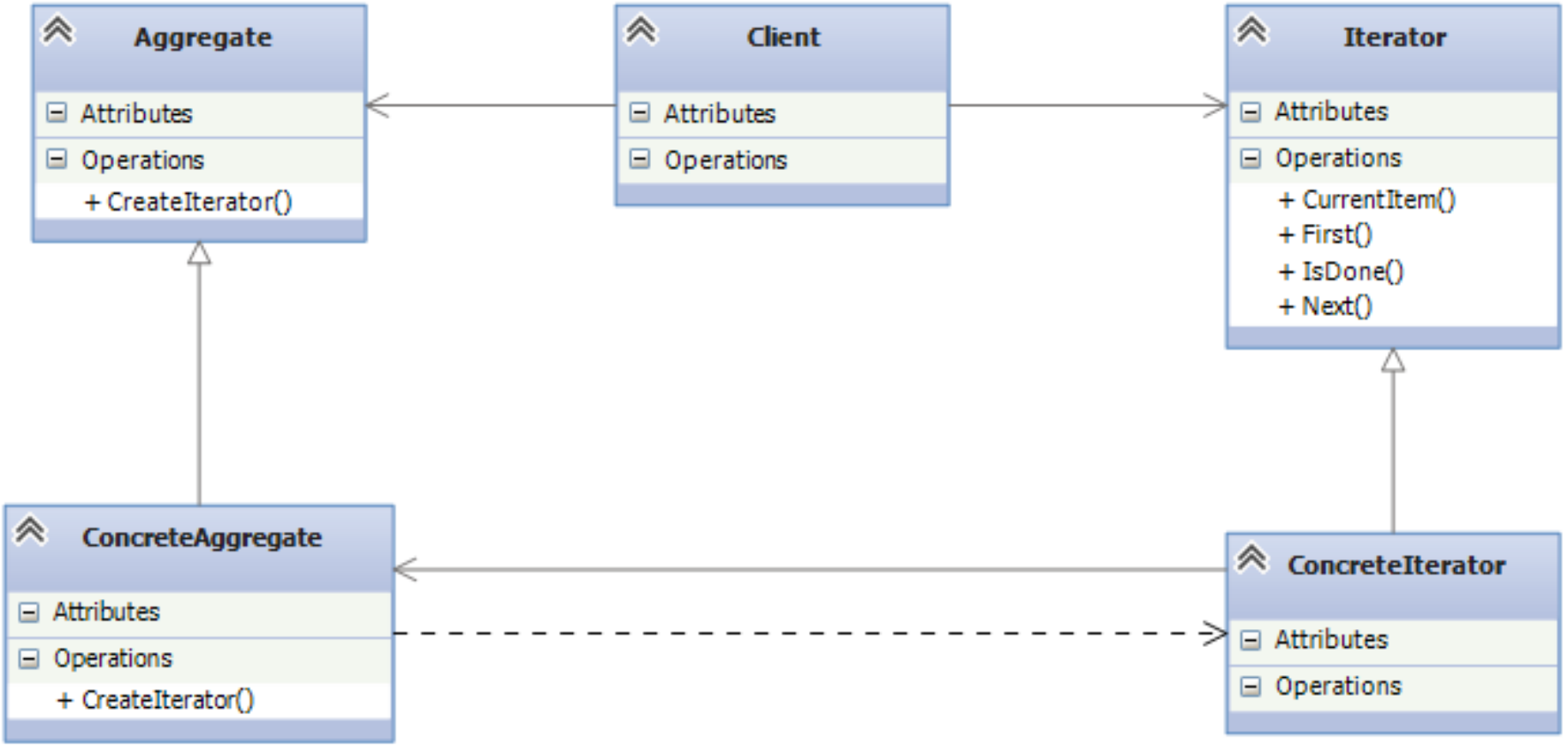


图 11.2 迭代器模式的类图



提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示

Iterator Pattern描述和使用

1．集合（Aggregate）：

在这里我们使用java.util包中的Collection接口作为模式中的集合角色。Java所有的集合都实现了该接口。

Iterator Pattern描述和使用

2．具体集合（ConcreteAggregate）：

在这里我们使用java.util包中的HashSet类的实例作为模式中的具体集合角色

Iterator Pattern描述和使用

3．迭代器（Iterator）：

在本问题中，我们使用的迭代器是java.util包中的Itertator接口

Iterator Pattern描述和使用

4. 具体迭代器（ConcreteIterator）

HashSet创建的集合可以使用iterator()方法返回一个实现Iterator接口类的实例，即一个具体迭代器

Iterator Pattern描述和使用

5. 应用 Application.java_1

```
import java.util.*;
public class Application{
    public static void main(String args[]){
        int n=20;
        int sum=0;
        Collection<RenMinMony> set=new HashSet<RenMinMony>();
        for(int i=1;i<=n;i++){
            if(i==n/2||i==n/5||i==n/6)
              set.add(new RenMinMony(100,false));
            else
              set.add(new RenMinMony(100,true));
        }
        Iterator<RenMinMony> iterator=set.iterator();
        int jia=1,zhen=1;
        System.out.println("保险箱共有"+set.size()+"张人民币");
        int k=0;
        RenMinMony money=iterator.next();
        k++;
        if(money.getIsTrue()==false){
            System.out.println("第"+k+"张是假币,被销毁");
            iterator.remove();
            k++;
        }
    }
    System.out.println("保险箱现有真人民币"+set.size()+"张,总价值是:")
    iterator=set.iterator();
    while(iterator.hasNext()){
        RenMinMony money=iterator.next();
        sum=sum+money.getValue();
    }
    System.out.println(sum+"元");
}
```

Iterator Pattern优点

- 用户使用迭代器访问集合中的对象，而不需要知道这些对象在集合中是如何表示及存储的
- 用户可以同时使用多个迭代器遍历一个集合

17. Mediator Pattern -中介者模式

- 中介者模式（Mediator Pattern）是用来降低多个对象和类之间的通信复杂性
- 这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护

Mediator Pattern实例

- MVC 框架，其中C（控制器）就是 M（模型）和 V（视图）的中介者

Mediator Pattern结构和使用

- 模式的结构中包括四种角色：

- 中介者（Mediator）
- 具体中介者（ConcreteMediator）
- 同事（Colleague）
- 具体同事（ConcreteColleague）

Mediator Pattern

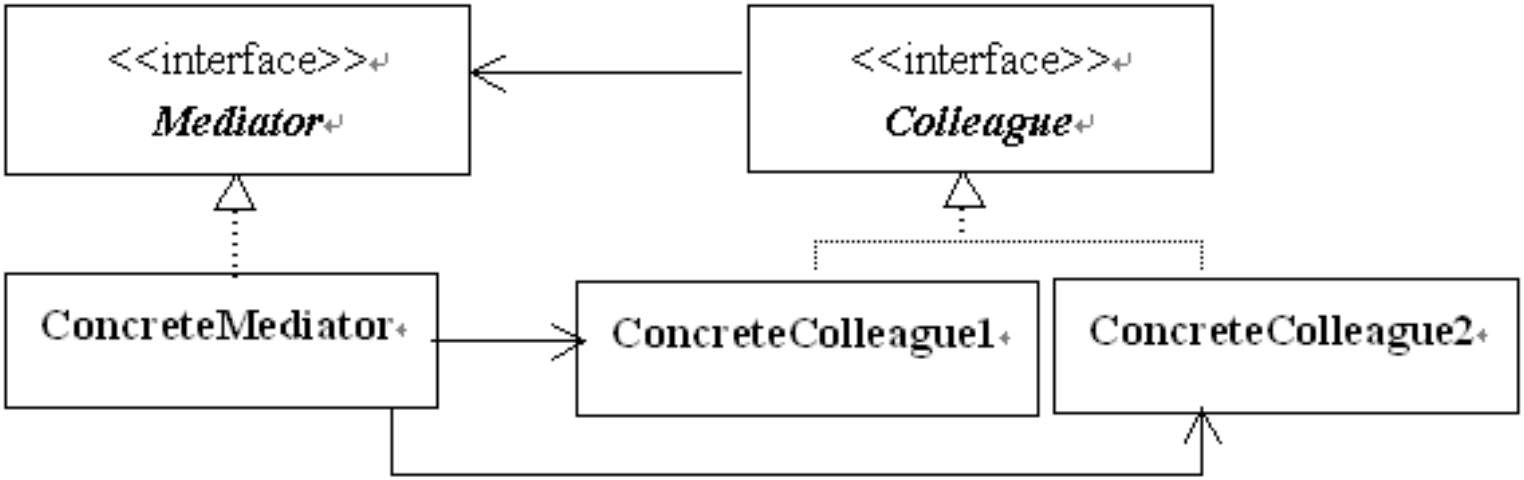
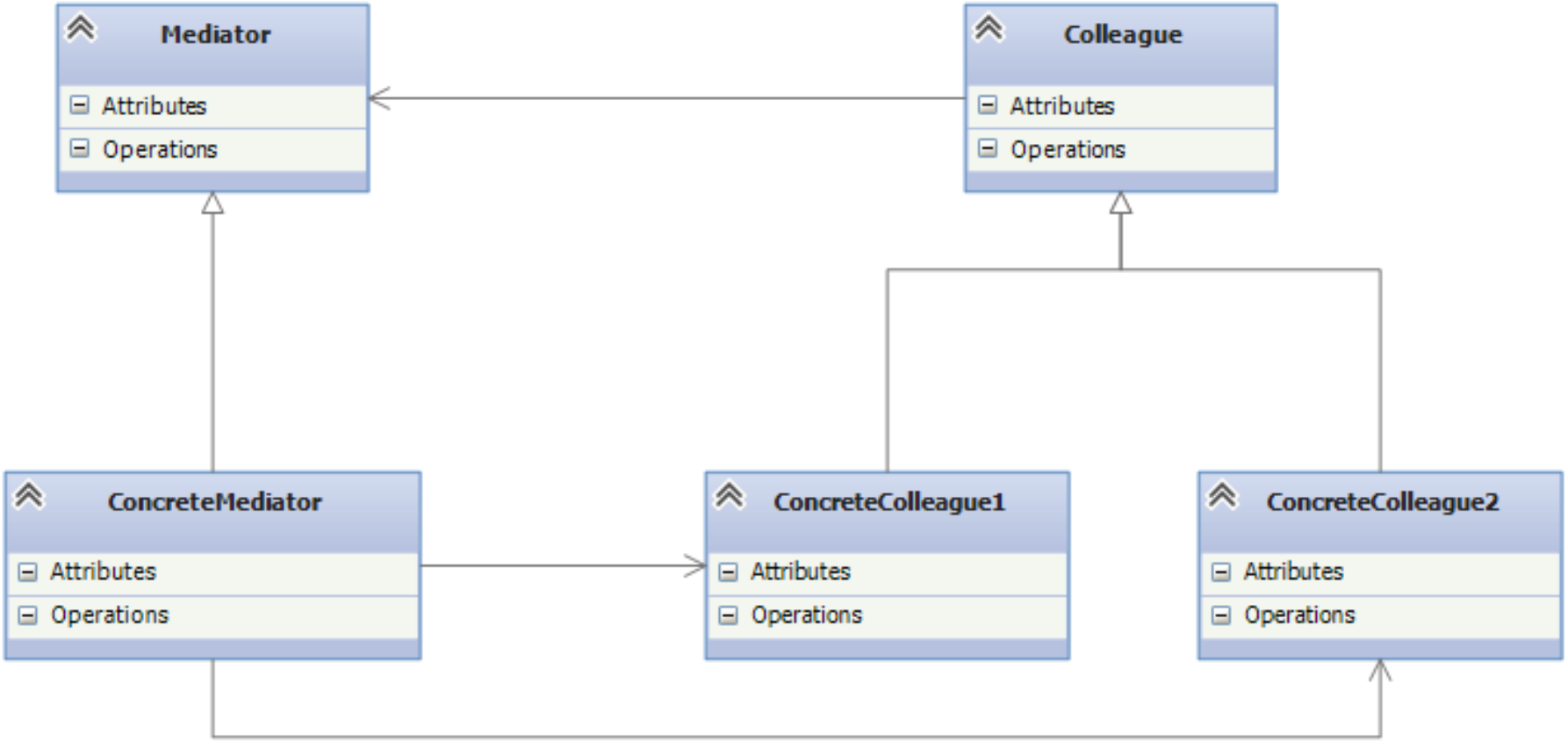


图 12.2 中介者模式的类图



用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显示地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互

Mediator Pattern描述和使用

1．中介者类代码：

```
public abstract class Mediator
 {
   protected ArrayList colleagues;
   public void register(Colleague colleague)
   {
     colleagues.add(colleague);
   }
   public abstract void operation();
 }
```

Mediator Pattern描述和使用

2. 具体中介者（Mediator）

```
public class ConcreteMediator extends Mediator
{
   public void operation()
   {
      ......
      ((Colleague)(colleagues.get(0))).method1();
      ......
   }
}
```

Mediator Pattern描述和使用

3. 同事（ConcreteColleague）：

```
public abstract class Colleague{
    protected Mediator mediator;

    public Colleague(Mediator mediator)
    {
      this.mediator=mediator;
    }

    public abstract void method1();

    public abstract void method2();
}
```

Mediator Pattern描述和使用

4. 具体同事（ConcreteColleague）：

```
public class ConcreteColleague extends Colleague{
    public ConcreteColleague(Mediator mediator)
    {
      super(mediator);
    }

    public void method1()
    {
      ......
    }

    public void method2()
    {
      mediator.operation1();
    }
```

Mediator Pattern描述和使用

5. 应用 Application.java

```
public class Application{
  public static void main(String args[]){
    ConcreteMediator mediator=new ConcreteMediator();
    ColleagueA colleagueA=new ColleagueA(mediator);
    ColleagueB colleagueB=new ColleagueB(mediator);
    ColleagueC colleagueC=new ColleagueC(mediator);
    colleagueA.setName("A国");
    colleagueB.setName("B国");
    colleagueC.setName("C国");
    String [] messA={"要求归还曾抢夺的100斤土豆","要求归还曾抢夺的20头牛"};
    colleagueA.giveMess(messA);
    String [] messB={"要求归还曾抢夺的10只公鸡","要求归还曾抢夺的15匹马"};
    colleagueB.giveMess(messB);
    String [] messC={"要求归还曾抢夺的300斤小麦","要求归还曾抢夺的50头驴"};
    colleagueC.giveMess(messC);
    }
}
```

Mediator Pattern优点

- 避免许多的对象为了之间的通信而相互显示引用
  - 不仅系统难于维护，而且也使其他系统难以复用这些对象
- 具体中介者使得各个具体同事完全解耦
  - 修改任何一个具体同事代码不会影响到其他同事
- 当一些对象想互相通信，但又无法相互包含对方的引用，那么使用中介者模式就可以使得这些对象互相通信

18. Memento Pattern -备忘录模式

- 备忘录模式（ Memento Pattern ）是保存一个对象的某个状态，以便在适当的时候恢复对象
- 备忘录模式属于行为型模式

Memento Pattern实例

- 打游戏时的存档
- Windows 里的 ctrl + z
- IE 中的后退
- 数据库的事务管理

Memento Pattern结构和使用

- **模式的结构中包括三种角色：**

- 原发者（Originator）
  - 创建并在 Memento 对象中存储状态
- 备忘录（Memento）
  - 包含了要被恢复的对象的状态
- 负责人（Caretaker）：
  - 负责从 Memento 中恢复对象的状态

Memento Pattern

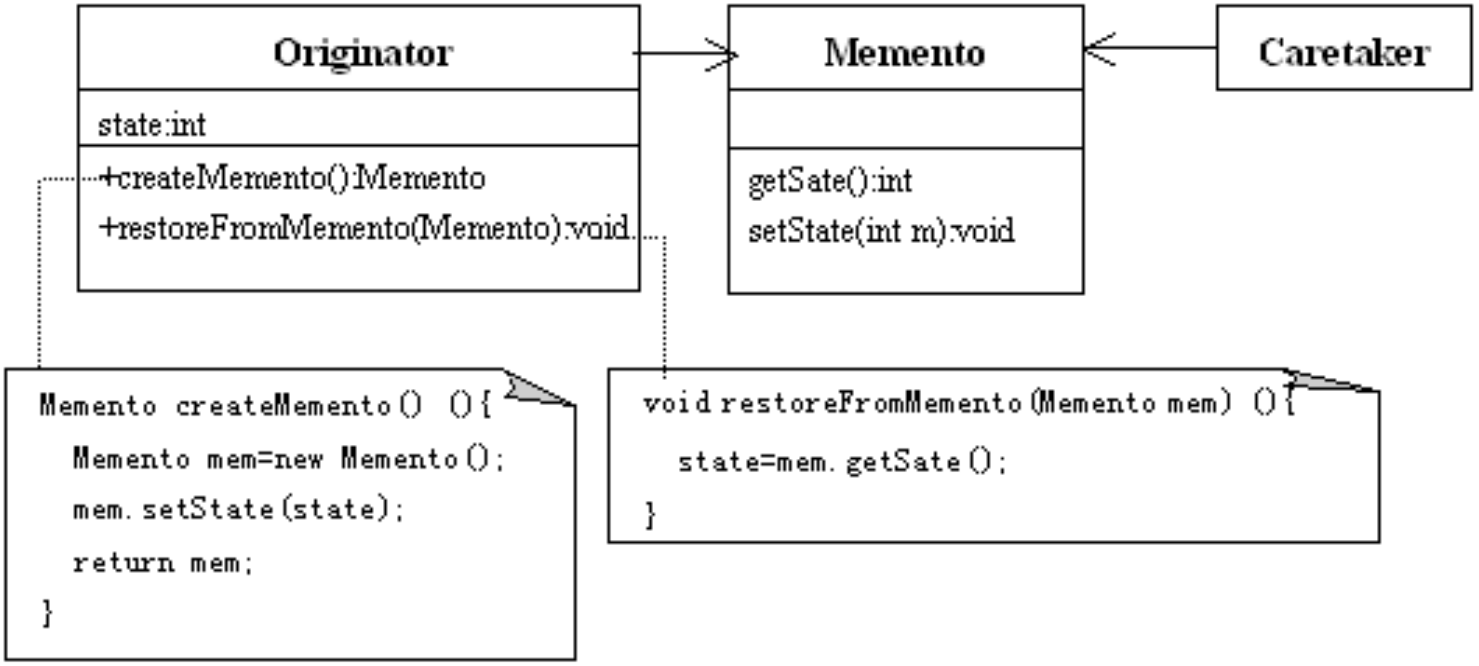
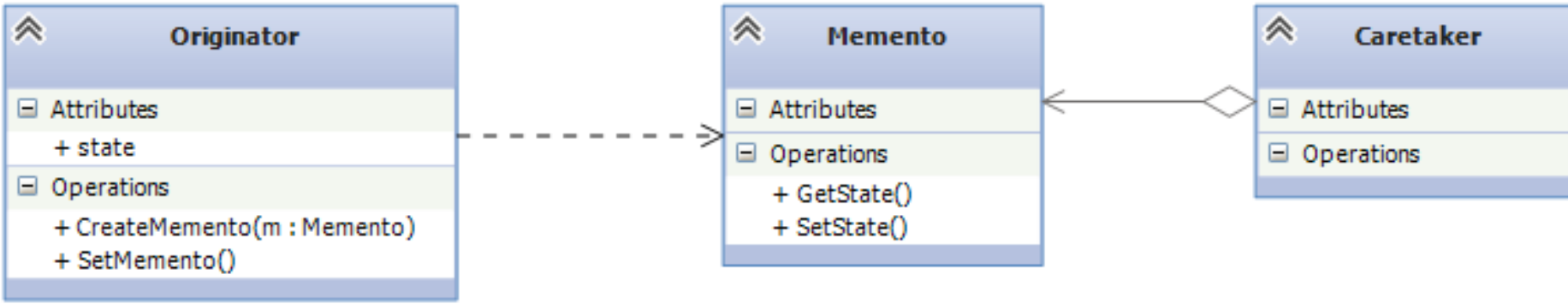


图 25.2　备忘录模式的类图



在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态

Memento Pattern描述和使用

1.Memento类代码：

```
public class Memento {
    private String state;
    public Memento(String state){
    this.state = state;
     }
    public String getState(){
     return state;
    }
}
```

Memento Pattern描述和使用

2. Originator类代码：

```
public class Originator {
    private String state;
    public void setState(String state){
      this.state = state;
    }
    public String getState(){
      return state;
    }
    public Memento saveStateToMemento(){
      return new Memento(state);
    }
     public void getStateFromMemento(Memento Memento){
      state = Memento.getState();
    }
}
```

Memento Pattern描述和使用

3.CareTaker类代码：

```
import java.util.ArrayList;
import java.util.List;
public class CareTaker {
    private List<Memento> mementoList =new ArrayList<Memento>();
    public void add(Memento state){
     mementoList.add(state);
   }
    public Memento get(int index){
      return mementoList.get(index);
    }
}
```

Memento Pattern描述和使用

4. 应用：Application.java _1

```
import tom.jiafei.*;
import java.util.Scanner;
import java.io.*;
public class Application{
  public static void main(String args[]) {
    Scanner reader=new Scanner(System.in);
    ReadPhrase  readPhrase=new ReadPhrase(new File("phrase.txt"));
    File favorPhrase=new File("favorPhrase.txt");
    RandomAccessFile out=null;
    try{    out=new RandomAccessFile(favorPhrase,"rw");}
    catch(IOException exp){}
    System.out.println("是否从上次读取的位置继续读取成语（输入y或n）");
    String answer=reader.nextLine();
    if(answer.startsWith("y")||answer.startsWith("Y")){
      Caretaker  caretaker=new Caretaker();        //创建负责人
      Memento memento=caretaker.getMemento();      //得到备忘录
    if(memento!=null) readPhrase.restoreFromMemento(memento);}
    String phrase=null;
    while((phrase=readPhrase.readLine())!=null){
    System.out.println(phrase);
    System.out.println("是否将该成语保存到"+favorPhrase.getName());
    answer=reader.nextLine();
```

Memento Pattern描述和使用

4. 应用：Application.java _1(接上面)

```
if(answer.startsWith("y")||answer.startsWith("Y")){
  try{     out.seek(favorPhrase.length());
  byte [] b=phrase.getBytes();
  out.write(b);
  out.writeChar('\n');}
  catch(IOException exp){}
}
System.out.println("是否继续读取成语？(输入y或n)");
answer=reader.nextLine();
if(answer.startsWith("y")||answer.startsWith("Y")) continue;
else{
  readPhrase.closeRead();
  Caretaker  caretaker=new Caretaker();  //创建负责人
  caretaker.saveMemento(readPhrase.createMemento());//保存备忘录
  try{  out.close();}
  catch(IOException exp){}
  System.exit(0);
}
 }
System.out.println("读完全部成语");
 }
}
```

Memento Pattern优点

- 使用备忘录可以把原发者的内部状态保存起来，使得只有很“亲密的”的对象可以访问备忘录中的数据
- 备忘录模式强调了类设计单一责任原则，即将状态的刻画和保存分开

19. Observer Pattern –观察者模式

- 观察者模式（Observer Pattern）是定义对象间的一种一对多的依赖关系
- 当一个对象的状态发生变化时，所有依赖于它的对象都得到通知并被自动更新

Observer Pattern实例

- 拍卖的时候，拍卖师观察最高标价，然后通知给其他竞价者竞价
- 西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟，这个乌龟就是观察者，他观察菩萨洒水这个动作

Observer Pattern结构和使用

- 模式的结构中包括三种角色：
  - 主题（Subject）
  - 观察者（Observer）
  - 具体主题（ConcreteSubject）
  - 具体观察者（ConcreteObserver）

Observer Pattern

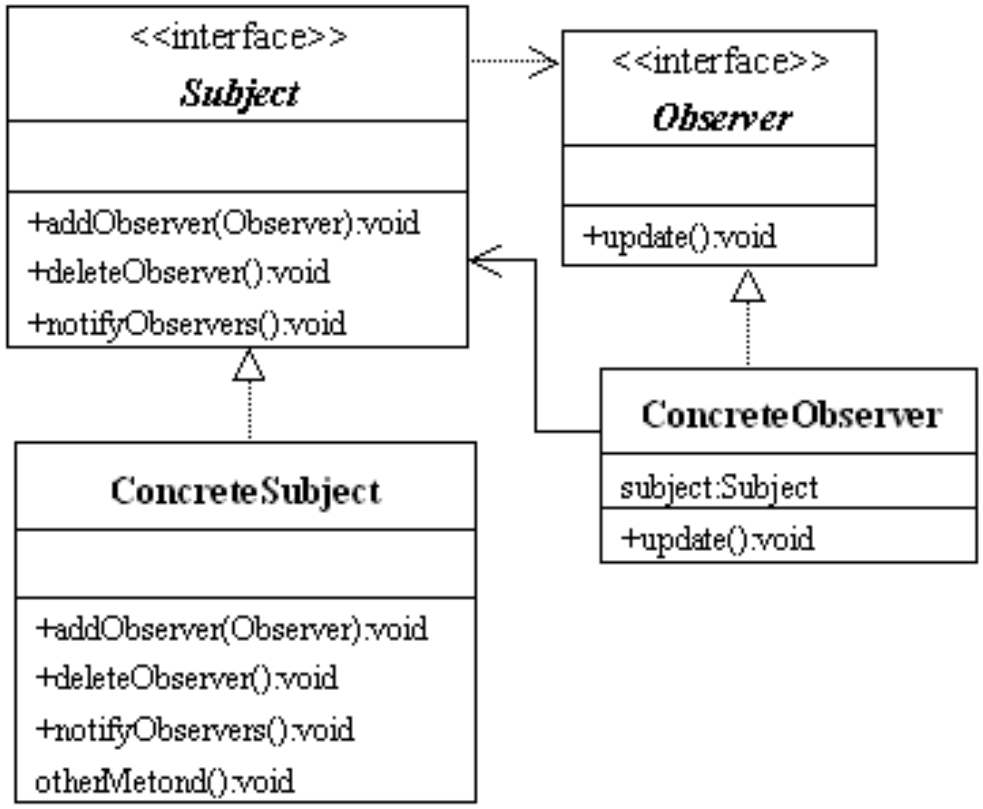
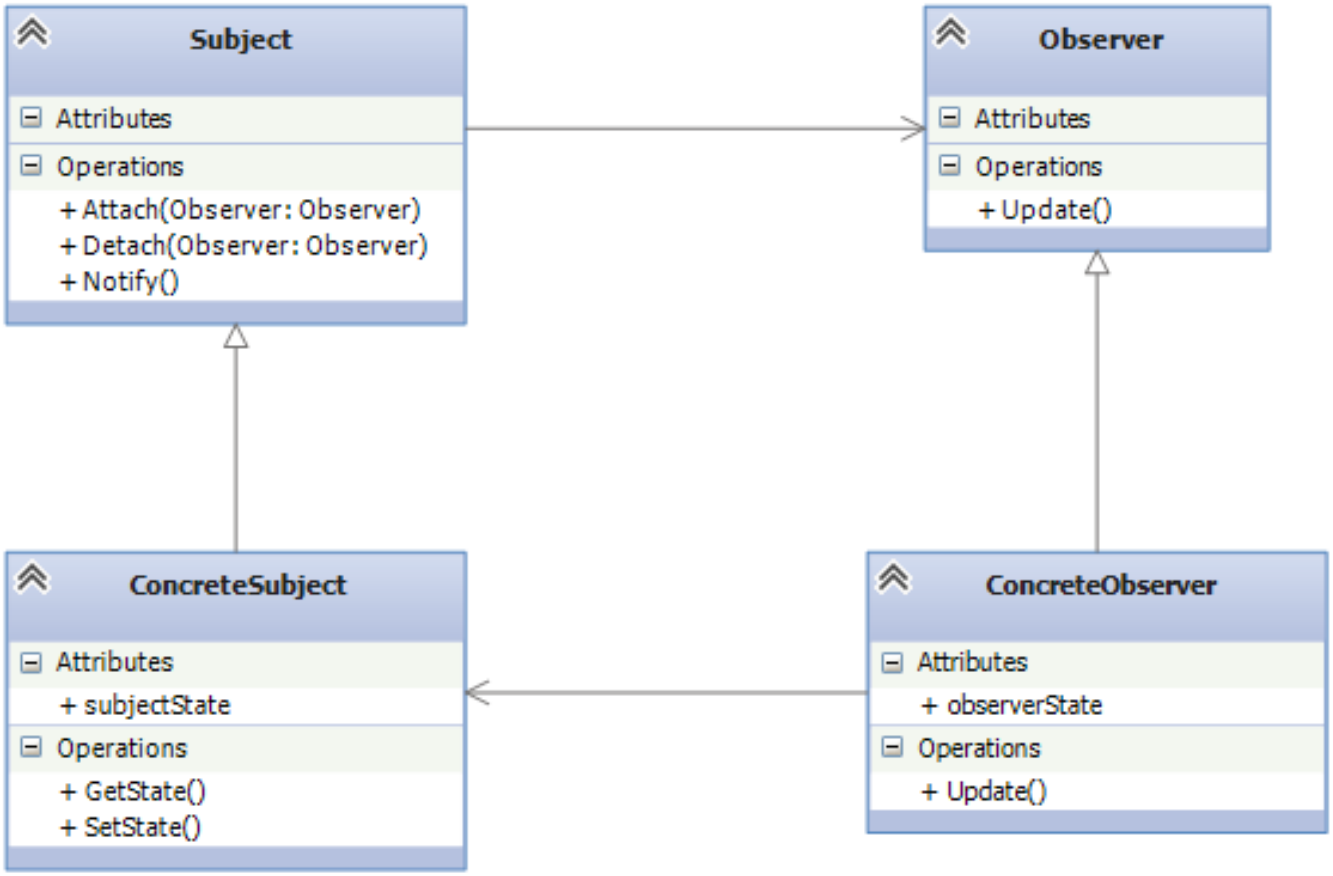


图 5.2　观察者模式的类图



定义对象间的一种一对多的依赖关系，当一个对象的状态发生变化时，所有依赖于它的对象都得到通知并被自动更新

Observer Pattern描述和使用

1．主题 ： Subject.java

```
public interface Subject{
   public void addObserver(Observer o);
   public void deleteObserver(Observer o);
   public void notifyObservers();
}
```

Observer Pattern描述和使用

2．观察者 ： Obsever.java

```
public interface Observer{
   public void hearTelephone(String heardMess);
}
```

Observer Pattern描述和使用

3. 具体主题 SeekJobCenter.java_2

```
public void notifyObservers(){
    if(changed){for(int i=0;i<personList.size();i++){
                Observer observer=personList.get(i);
                observer.hearTelephone(mess); }
                changed=false;
        }
    }
    public void giveNewMess(String str){
        if(str.equals(mess))
                changed=false;
        else{
                mess=str;
                changed=true;
        }
    }
}
```

Observer Pattern描述和使用

4. 具体观察者_1UniversityStudent.java

```
import java.io.*;
  public class UniverStudent implements Observer{
      Subject subject;File myFile;
      UniverStudent(Subject subject,String fileName){
      this.subject=subject;
      subject.addObserver(this);//使当前实例成为subject所引用具体主题观察者
      myFile=new File(fileName);
     }
       public void hearTelephone(String heardMess){
        try{ RandomAccessFile out=new RandomAccessFile(myFile,"rw");
         out.seek(out.length());
         byte [] b=heardMess.getBytes();
         out.write(b); //更新文件中的内容
         System.out.print("我是一个大学生,");
         System.out.println("我向文件"+myFile.getName()+"写入如下内容:");
           System.out.println(heardMess);
         }
        catch(IOException exp){System.out.println(exp.toString());}
    }
  }
```

Observer Pattern描述和使用

4. 具体观察者_2 HaiGui.java

```
import java.io.*;
import java.util.regex.*;
public class HaiGui implements Observer{
   Subject subject;File myFile;
   HaiGui(Subject subject,String fileName){
      this.subject=subject;
      subject.addObserver(this);//使当前实例成为subject所引用的具体主题的观察者
      myFile=new File(fileName);}
   public void hearTelephone(String heardMess){
      try{ boolean boo=heardMess.contains("java程序员")||heardMess.contains("软件");
           if(boo){
             RandomAccessFile out=new RandomAccessFile(myFile,"rw");
             out.seek(out.length());
             byte [] b=heardMess.getBytes();
             out.write(b);
             System.out.print("我是一个海归,");
             System.out.println("我向文件"+myFile.getName()+"写入如下内容:");
             System.out.println(heardMess);}
           else{System.out.println("我是海归,这次的信息中没有我需要的信息");} }
      catch(IOException exp){System.out.println(exp.toString());}
   }
}
```

Observer Pattern描述和使用

5. 应用 Application.java

```
public class Application{
    public static void main(String args[]){
        SeekJobCenter center=new SeekJobCenter();
        UniverStudent zhangLin=new UniverStudent(center,"A.txt");
        HaiGui wangHao=new HaiGui(center,"B.txt");
        center.giveNewMess("腾辉公司需要10个java程序员。");
        center.notifyObservers();
        center.giveNewMess("海景公司需要8个动画设计师。");
        center.notifyObservers();
        center.giveNewMess("仁海公司需要9个电工。");
        center.notifyObservers();
        center.giveNewMess("仁海公司需要9个电工。");
        center.notifyObservers();
}
```

Observer Pattern优点

- 具体主题和具体观察者是松耦合关系
  - 由于主题（Subject）接口仅仅依赖于观察者（Observer）接口，因此具体主题只是知道它的观察者是实现观察者（Observer）接口的某个类的实例，但不需要知道具体是哪个类
- 观察模式满足“开-闭原则”

20. State Pattern – 状态模式

- 一个对象的状态依赖于它的成员变量的取值情况。对象在不同的运行环境或运行时刻，可能具有不同的状态
- 在许多情况下，对象调用方法所产生的行为效果依赖于它当时的状态。

State Pattern实例

- 打篮球的时候运动员可以有正常状态、不正常状态和超常状态
- 曾侯乙编钟中，'钟是抽象接口','钟A'等是具体状态，'曾侯乙编钟'是具体环境（Context）

State Pattern结构和使用

- 模式的结构中包括三种角色：

- 环境（Context）
- 抽象状态（State）
- 具体状态（Concrete State）

State Pattern

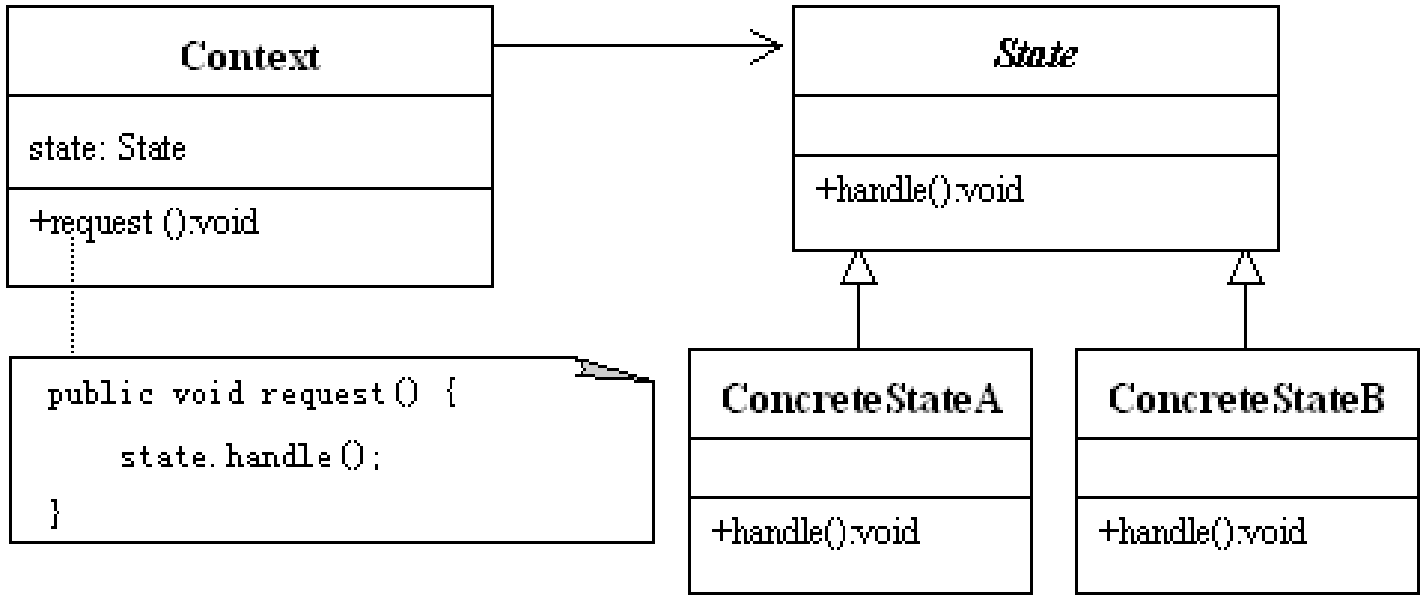
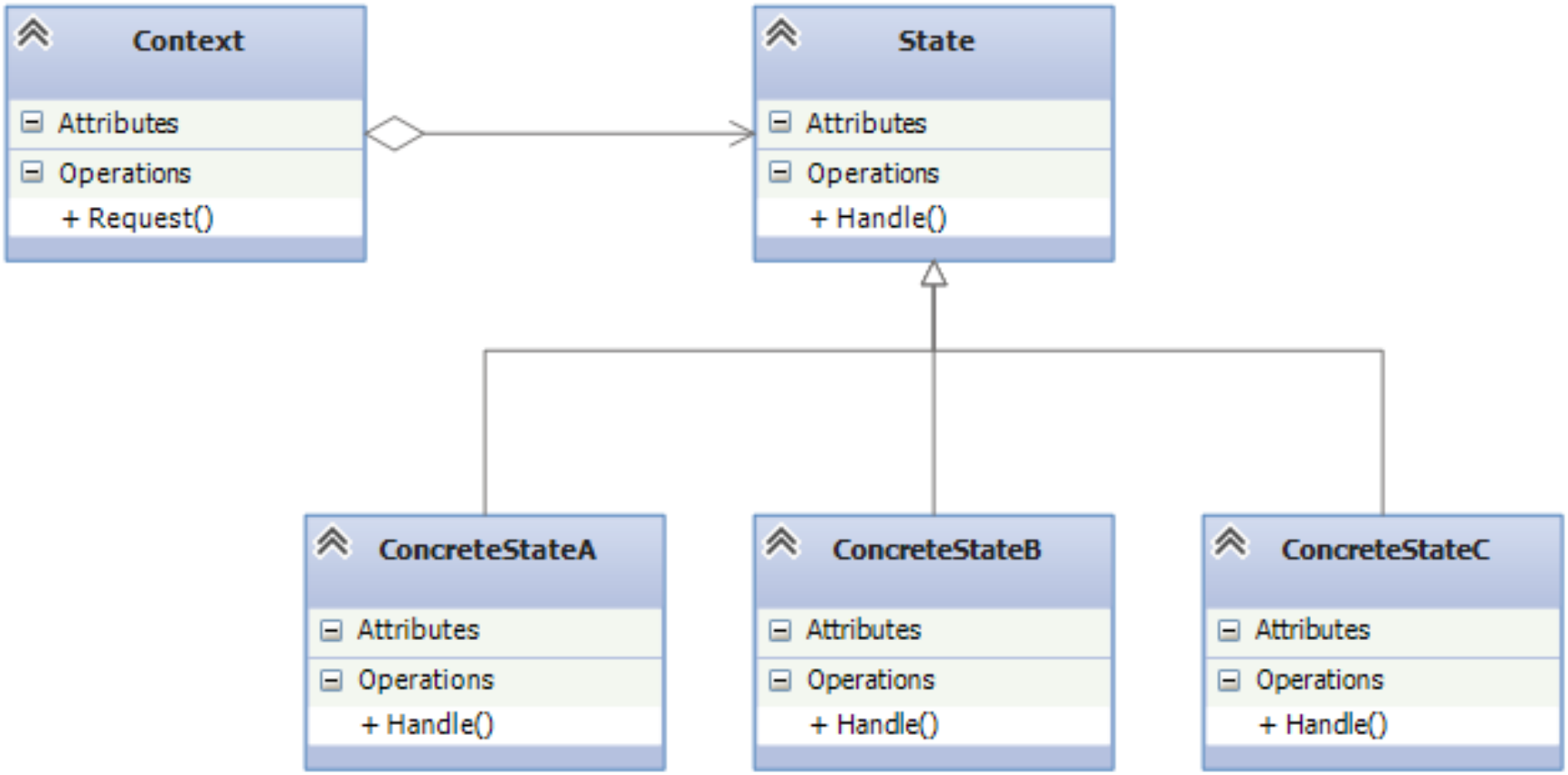


图 20.4 状态模式的类图


Context
Attributes
Operations
+ Request()
State
Attributes
Operations
+ Handle()
ConcreteStateA
Attributes
Operations
+ Handle()
ConcreteStateB
Attributes
Operations
+ Handle()
ConcreteStateC
Attributes
Operations
+ Handle()

State Pattern描述和使用

1. 环境：Thermometer.java

```
public  class  Thermometer{
  TemperatureState  state;
  public void showMessage(){
    System.out.println("***********");
    state.showTemperature();
    System.out.println("***********");
  }
  public void setState(TemperatureState  state){
    this.state=state;
  }
}
```

State Pattern描述和使用

2. 抽象状态：TemperatureState.java

```
public interface TemperatureState{
  public void showTemperature();
}
```

State Pattern描述和使用

3. 具体状态：LowState.java

```
public class LowState implements TemperatureState{
   double n=0;
   LowState(double n){
    if(n<=0) this.n=n;
   }
   public void showTemperature(){
     System.out.println("现在温度是"+n+"属于低温度");
   }
}
```

State Pattern描述和使用

3．具体状态_2：MiddleState.java

```
public class MiddleState implements TemperatureState{
  double n=15;
  MiddleState(int n){
    if(n>0&&n<26)
    this.n=n;
  }
  public void showTemperature(){
    System.out.println("现在温度是"+n+"属于正常温度");
  }
}
```

State Pattern描述和使用

3. 具体状态_3：HeightState.java

```
public class HeightState implements TemperatureState{
        double n=39;
        HeightState(int n){
             if(n>=39)
                this.n=n;
        }
        public void showTemperature(){
                System.out.println("现在温度是"+n+"属于高温度");
        }
}
```

State Pattern描述和使用

4. 应用 Application.java

```
public class Application{
    public static void main(String args[]) {
        TemperatureState state=new LowState(-12);
        Thermometer  thermometer=new Thermometer();
        thermometer.setState(state);
        thermometer.showMessage();
        state=new MiddleState(20);
        thermometer.setState(state);
        thermometer.showMessage();
        state=new HeightState(39);
        thermometer.setState(state);
        thermometer.showMessage();
    }
}
```

State Pattern优点

- 使用一个类封装对象的一种状态，很容易增加新的状态
- 在状态模式中，环境（context）中不必出现大量的条件判断语句
  - 环境（context）实例所呈现的状态变得更加清晰、容易理解
- 使用状态模式可以让用户程序很方便的切换环境（context）实例的状态

21. Strategy Pattern – 策略模式

- 在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改
- 在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象
- 策略对象改变 context 对象的执行算法

Strategy Pattern概述

- 策略模式是处理算法的不同变体的一种成熟模式
- 策略模式通过接口或抽象类封装算法的标识，即在接口中定义一个抽象方法，实现该接口的类将实现接口中的抽象方法
- 在策略模式中，封装算法标识的接口称作策略，实现该接口的类称作具体策略

Strategy Pattern结构和使用

- 模式的结构中包括三种角色：

- 策略（Strategy）
- 具体策略（ConcreteStrategy）
- 上下文（Context）

Strategy Pattern

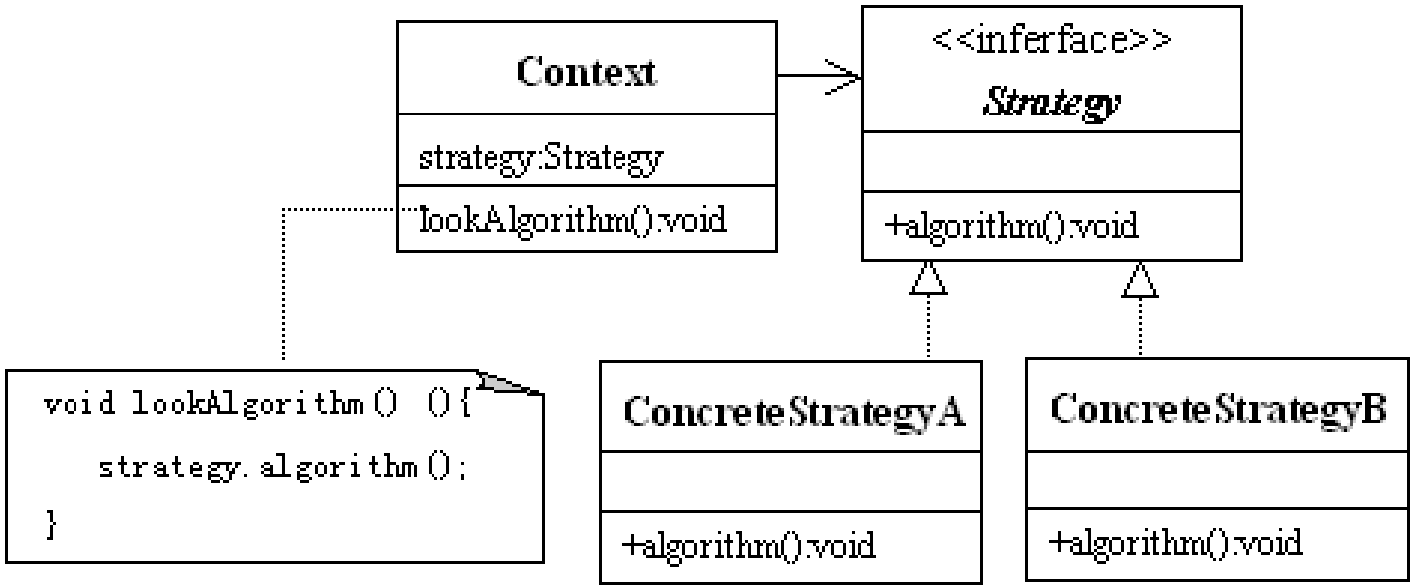
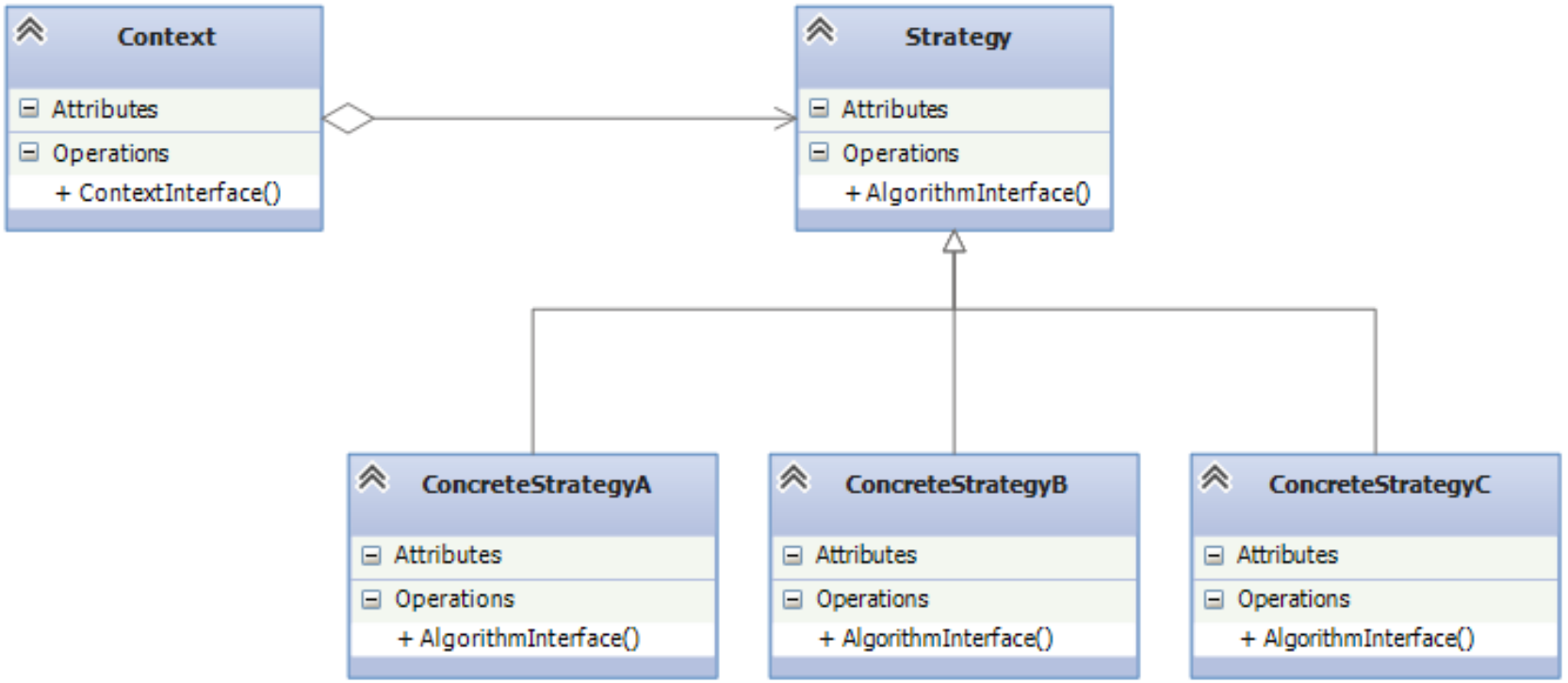


图 7.6　策略模式的类图


Context
Attributes
Operations
+ ContextInterface()
Strategy
Attributes
Operations
+ AlgorithmInterface()
ConcreteStrategyA
Attributes
Operations
+ AlgorithmInterface()
ConcreteStrategyB
Attributes
Operations
+ AlgorithmInterface()
ConcreteStrategyC
Attributes
Operations
+ AlgorithmInterface()

Strategy Pattern描述和使用

1. 策略: Computable.java

```
public interface ComputableStrategy{
   public abstract double computeScore(double [] a);
}
```

Strategy Pattern描述和使用

2. 具体策略: StrategyOne.java

```
public class StrategyOne implements ComputableStrategy{
    public double computeScore(double [] a){
        double score=0,sum=0;
        for(int i=0;i<a.length;i++){
            sum=sum+a[i];
        }
        score=sum/a.length;
        return score;
    }
}
```

Strategy Pattern描述和使用

2. 具体策略: StrategyTwo.java

```
public class StrategyTwo implements ComputableStrategy{
    public double computeScore(double [] a){
        double score=0,multi=1;
        int n=a.length;
        for(int i=0;i<a.length;i++){
            multi=multi*a[i];
        }
        score=Math.pow(multi,1.0/n);
        return score;
    }
}
```

Strategy Pattern描述和使用

2. 具体策略:StrategyThree.java

```
import java.util.Arrays;
public class StrategyThree implements ComputableStrategy{
    public double computeScore(double [] a){
        if(a.length<=2)
            return 0;
        double score=0,sum=0;
        Arrays.sort(a);
        for(int i=1;i<a.length-1;i++){
            sum=sum+a[i];
        }
        score=sum/(a.length-2);
        return score;
    }
}
```

Strategy Pattern描述和使用

3. 上下文： GymnasticsGame.java

```
public class  GymnasticsGame{
 ComputableStrategy strategy;
 public void setStrategy(ComputableStrategy strategy){
          this.strategy=strategy;
     }
 public double getPersonScore(double [] a){
          if(strategy!=null)
            return strategy.computeScore(a);
          else
            return 0;
     }
}
```

Strategy Pattern描述和使用

4. 应用：Application.java_1

```
public class Application{
    public static void main(String args[]){
    GymnasticsGame game=new GymnasticsGame();
    game.setStrategy(new StrategyOne());
    Person zhang=new Person();
    zhang.setName("张三");
    double [] a={9.12,9.25,8.87,9.99,6.99,7.88};
    Person li=new Person();
    li.setName("李四");
    double [] b={9.15,9.26,8.97,9.89,6.97,7.89};
    zhang.setScore(game.getPersonScore(a));
    li.setScore(game.getPersonScore(b));
    System.out.println("使用算术平均值方案:");
    System.out.printf("%s最后得分:%5.3f%n",zhang.getName(),zhang.getScore());
    System.out.printf("%s最后得分:%5.3f%n",li.getName(),li.getScore());
    game.setStrategy(new StrategyTwo());
    zhang.setScore(game.getPersonScore(a));
    li.setScore(game.getPersonScore(b));
    System.out.println("使用几何平均值方案:");
    System.out.printf("%s最后得分:%5.3f%n",zhang.getName(),zhang.getScore());
    System.out.printf("%s最后得分:%5.3f%n",li.getName(),li.getScore());
    }
}
```

Strategy Pattern优点

- 上下文和具体策略是松耦合关系
  - 因此上下文只知道它要使用某一个实现Strategy接口类的实例，但不需要知道具体是哪一个类
- 策略模式满足“开-闭原则”
  - 当增加新的具体策略时，不需要修改上下文类的代码，上下文就可以引用新的具体策略的实例

22. Template Pattern – 模板模式

- 在模板模式（Template Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板
- 它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行

Template Pattern实例

- 在造房子的时候，地基、走线、水管都一样，只有在建筑的后期才有加壁橱加栅栏等差异
- Spring 中对 Hibernate 的支持，将一些已经定好的方法封装起来，比如开启事务、获取Session、关闭 Session 等，程序员不重复写那些已经规范好的代码，直接丢一个实体就可以保存

Template Pattern结构和使用

- 模式的结构中包括两种角色：

- 抽象模板（Abstract Template）
- 具体模板（Concrete Template）

Template Pattern

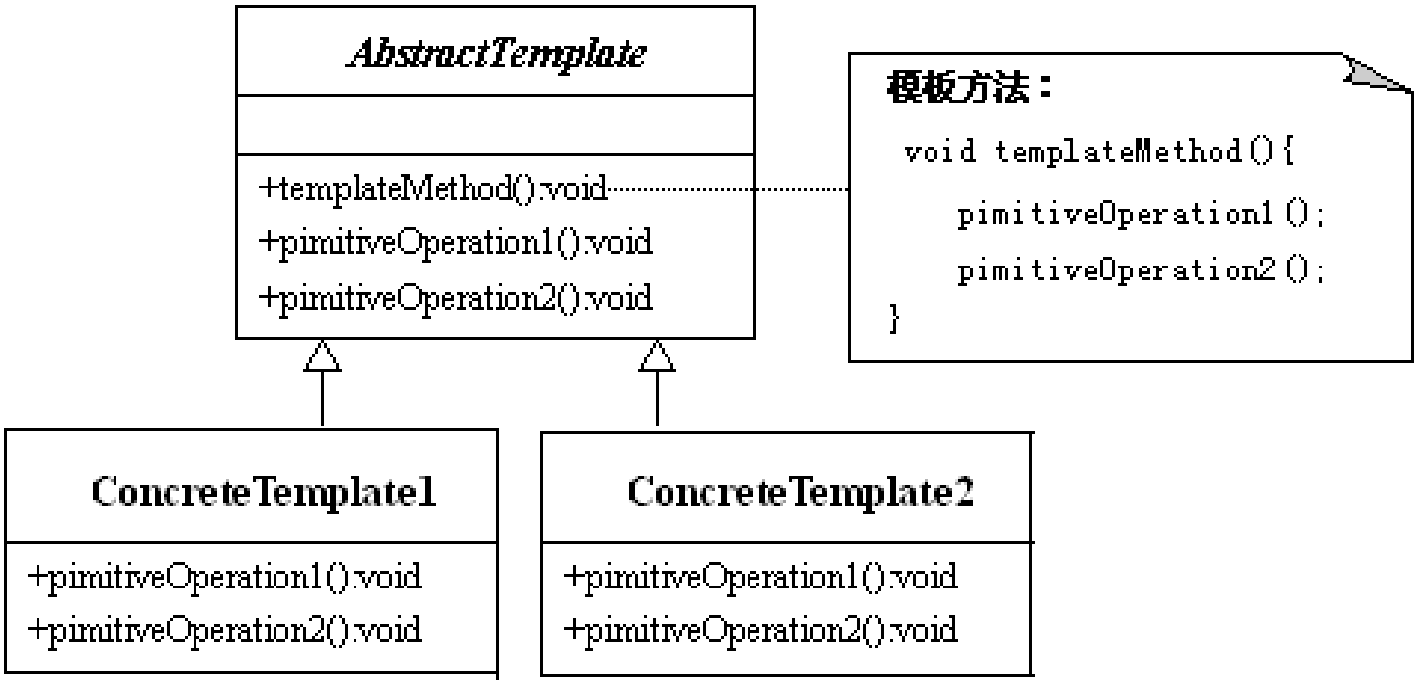
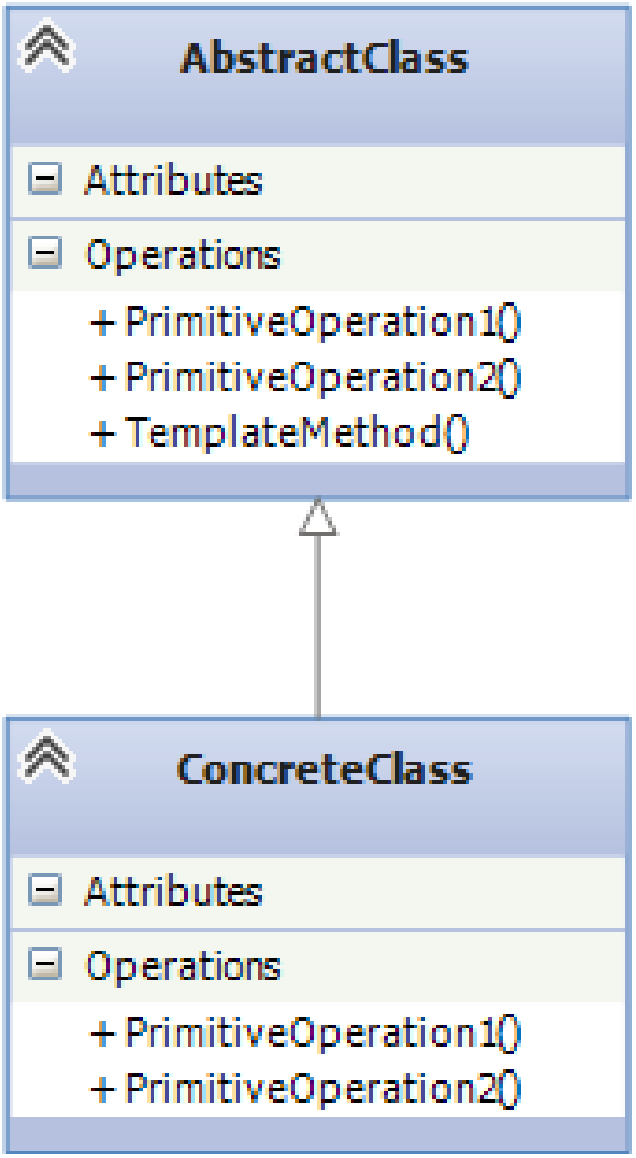


图 21.2　模板方法模式的类图



*定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。**TemplateMethod**使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤*

Template Pattern描述和使用

1. 抽象模板: AbstractTemplate.java

```
import java.io.*;
public abstract  class  AbstractTemplate{
        File [] allFiles;
        File dir;
        AbstractTemplate(File dir){
            this.dir=dir;
        }
         public final void  showFileName(){
             allFiles=dir.listFiles();
             sort();
             printFiles();
        }
        public abstract void sort();
        public abstract void printFiles();
}
```

Template Pattern描述和使用

2. 具体模板:ConcreteTemplate1.java

```
import java.io.*;
import java.awt.*;
import java.util.Date;
import java.text.SimpleDateFormat;
public  class ConcreteTemplate1 extends  AbstractTemplate{
  ConcreteTemplate1(File dir){super(dir);}
  public void sort(){
    for(int i=0;i<allFiles.length;i++)
      for(int j=i+1;j<allFiles.length;j++)
        if(allFiles[j].lastModified()<allFiles[i].lastModified()){
          File file=allFiles[j];
          allFiles[j]=allFiles[i];
          allFiles[i]=file;
        }
      }
```

Template Pattern描述和使用

2. 具体模板:ConcreteTemplate1.java（接上）

```
public void printFiles(){
  for(int i=0;i<allFiles.length;i++){
    long time=allFiles[i].lastModified();
    Date date=new Date(time);
    SimpleDateFormat matter= new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    String str=matter.format(date);
    String name=allFiles[i].getName();
    int k=i+1;
    System.out.println(k+" "+name+"("+str+")");
    }
  }
}
```

钩子方法

- 钩子方法是抽象模板中定义的具体方法，但给出了空实现或默认的实现，并允许子类重写这个具体方法，否则应用final修饰
- 某些钩子方法的作用是对模板方法中的某些步骤进行“挂钩”，即允许具体模板对算法的不同点进行“挂钩”，以确定在什么条件下执行模板方法中的哪些算法步骤，因此这类钩子方法的类型一般是boolean类型

Template Pattern优点

- 封装不变部分，扩展可变部分
- 提取公共代码，便于维护
- 行为由父类控制，子类实现

23. Visitor Pattern – 访问者模式

- 表示一个作用于某对象结构中的各个元素的操作
- 它使你可以在不改变各个元素的类的前提下定义作用于这些元素的新操作

Visitor Pattern实例

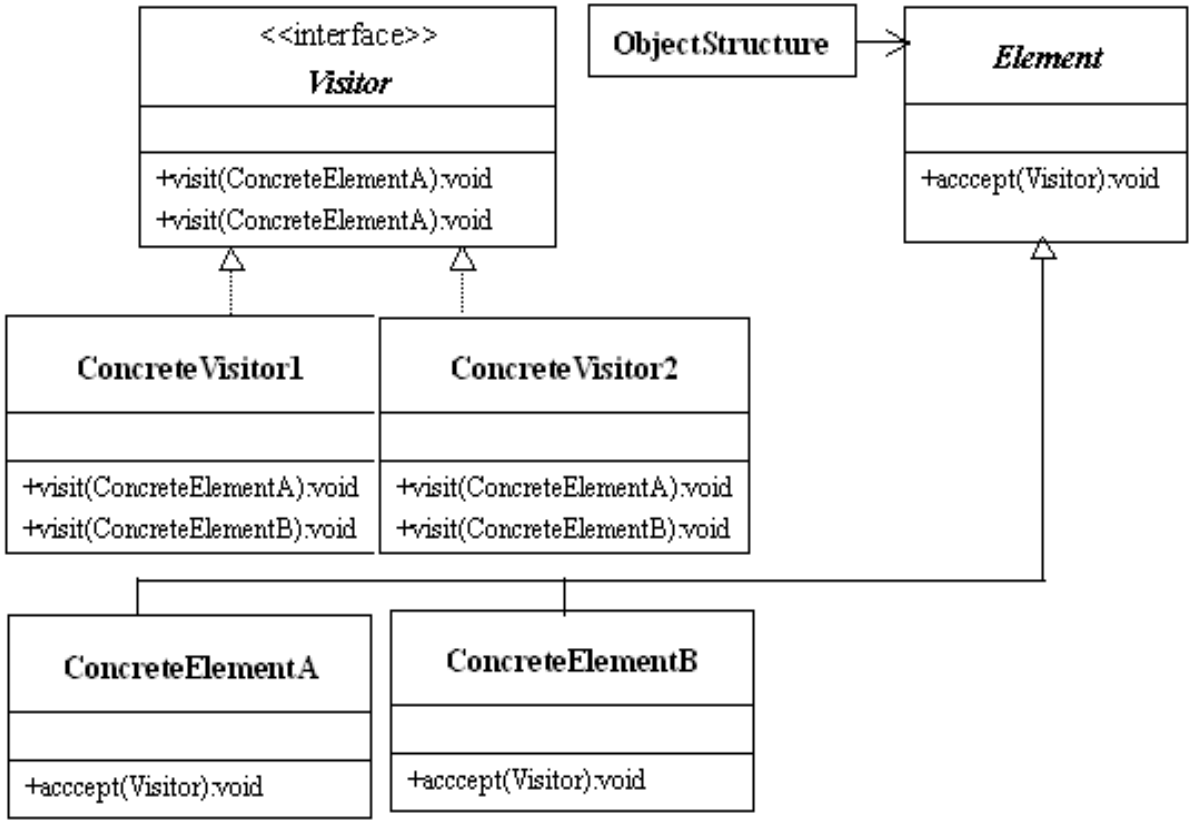
- 你在朋友家做客，您是访问者，朋友接受您的访问，您通过朋友的描述，然后对朋友的描述做出一个判断，这就是访问者模式

Visitor Pattern结构和使用

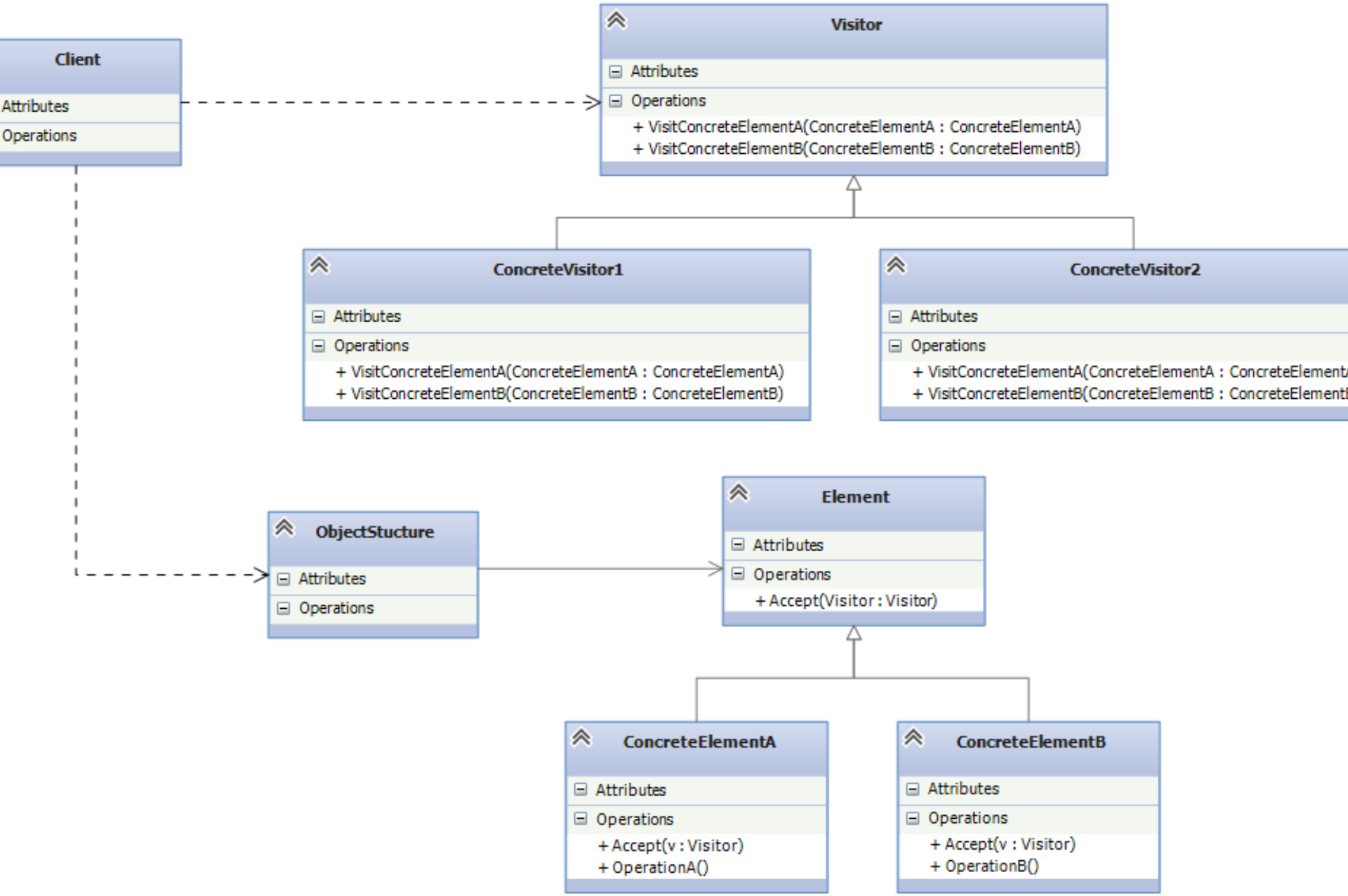
- 模式的结构中包括两种角色：

- 抽象元素（Element）
- 具体元素（Concrete Element）
- 对象结构（Object Structure）
- 抽象访问者（Visitor）
- 具体访问者（Concrete Visitor）

Visitor Pattern



表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的前提下定义作用于这些元素的新操作


Client
Attributes
Operations
Visitor
Attributes
Operations
+ VisitConcreteElementA(ConcreteElementA : ConcreteElementA)
+ VisitConcreteElementB(ConcreteElementB : ConcreteElementB)
ConcreteVisitor1
Attributes
Operations
+ VisitConcreteElementA(ConcreteElementA : ConcreteElementA)
+ VisitConcreteElementB(ConcreteElementB : ConcreteElementB)
ConcreteVisitor2
Attributes
Operations
+ VisitConcreteElementA(ConcreteElementA : ConcreteElement
+ VisitConcreteElementB(ConcreteElementB : ConcreteElement
ObjectStucture
Attributes
Operations
Element
Attributes
Operations
+ Accept(Visitor : Visitor)
ConcreteElementA
Attributes
Operations
+ Accept(v : Visitor)
+ OperationA()
ConcreteElementB
Attributes
Operations
+ Accept(v : Visitor)
+ OperationB()

Visitor Pattern描述和使用

1．抽象元素（Element）：Student.java

```
public abstract class Student{
    public abstract void accept(Visitor v);
}
```

Visitor Pattern描述和使用

2. 具体元素:Undergraduate.java

```
public  class Undergraduate extends Student{
        double math,english;    //成绩
        String name;
        Undergraduate(String name,double math,double english){
                this.name=name;
                this.math=math;
                this.english=english; }
        public double getMath(){
                return math;}
        public double getEnglish(){
                return english;
        }
        public String getName(){
                return name;
        }
        public void accept(Visitor v){
             v.visit(this);
        }
}
```

Visitor Pattern描述和使用

3．对象结构（Object Structure）

本问题中，我们让该角色是java.util包中的ArrayList集合

Visitor Pattern描述和使用

4. 抽象访问者（Visitor）：Visitor.java

```
public interface Visitor{
    public void visit(Undergraduate stu);
    public void visit(GraduateStudent stu);
}
```

Visitor Pattern描述和使用

5．具体访问者：Company.java

```
public class  Company implements Visitor{
    public void visit(Undergraduate stu){
            double math=stu.getMath();
            double english=stu.getEnglish();
            if(math>80&&english>90)
                System.out.println(stu.getName()+"被录用");
    }
    public void visit(GraduateStudent stu){
          double math=stu.getMath();
          double english=stu.getEnglish();
          double physics=stu.getPhysics();
          if(math>80&&english>90&&physics>70)
                System.out.println(stu.getName()+"被录用");
    }
}
}
```

Visitor Pattern描述和使用

6. 应用：Application.java

```
import java.util.*;
public class Application{
    public static void main(String args[]) {
        Visitor visitor=new Company();
        ArrayList<Student>  studentList=new  ArrayList<Student>();
        Student student=null;
        studentList.add(student=new Undergraduate("张三",67,88));
        studentList.add(student=new Undergraduate("李四",90,98));
        studentList.add(student=new Undergraduate("将粼粼",85,92));
        studentList.add(student=new GraduateStudent("刘名",88,70,87));
        studentList.add(student=new GraduateStudent("郝人",90,95,82));
        Iterator<Student> iter=studentList.iterator();
        while(iter.hasNext()){
            Student stu=iter.next();
            stu.accept(visitor);
        }
    }
}
```

Visitor Pattern优点

- 可以在不改变一个集合中的元素的类的情况下，增加新的施加于该元素上的新操作
- 可以将集合中各个元素的某些操作集中到访问者中，不仅便于集合的维护，也有利于集合中元素的复用。

设计模式

- 设计模式的要素
- 设计模式的目的
- 设计模式的分类
- 设计模式的选择
- 设计模式的使用

设计模式的选择

- 使用设计模式能给设计人员带来很多好处。选择模式的方法很多，人们也开始寻找自动获取模式的方法，但还不成熟。
- 在目前的实际工作当中，人们仍然采用传统的模式选择方法，设计模式功能的理解和自身的设计经验。这要求设计人员主要凭借对对所有设计模式都有较深的理解和掌握。

设计模式的选择

- 考虑设计模式是怎样解决问题的
- 浏览模式的意图部分
- 研究模式怎样相互关联
- 研究目的相似的模式
- 检查重新设计的原因
- 考虑你的设计中哪些是可变的

设计模式

- 设计模式的要素
- 设计模式的目的
- 设计模式的分类
- 设计模式的选择
- 设计模式的使用

设计模式的使用

- 大致浏览一遍模式
- 回头研究结构部分、参与者部分和协作部分
- 看代码示例部分，看看此模式代码的具体例子
- 选择模式参与者的名字，使它们在应用中有意义
- 定义类
- 定义模式中专用于应用的操作名称
- 实现执行模式中责任和协作的操作

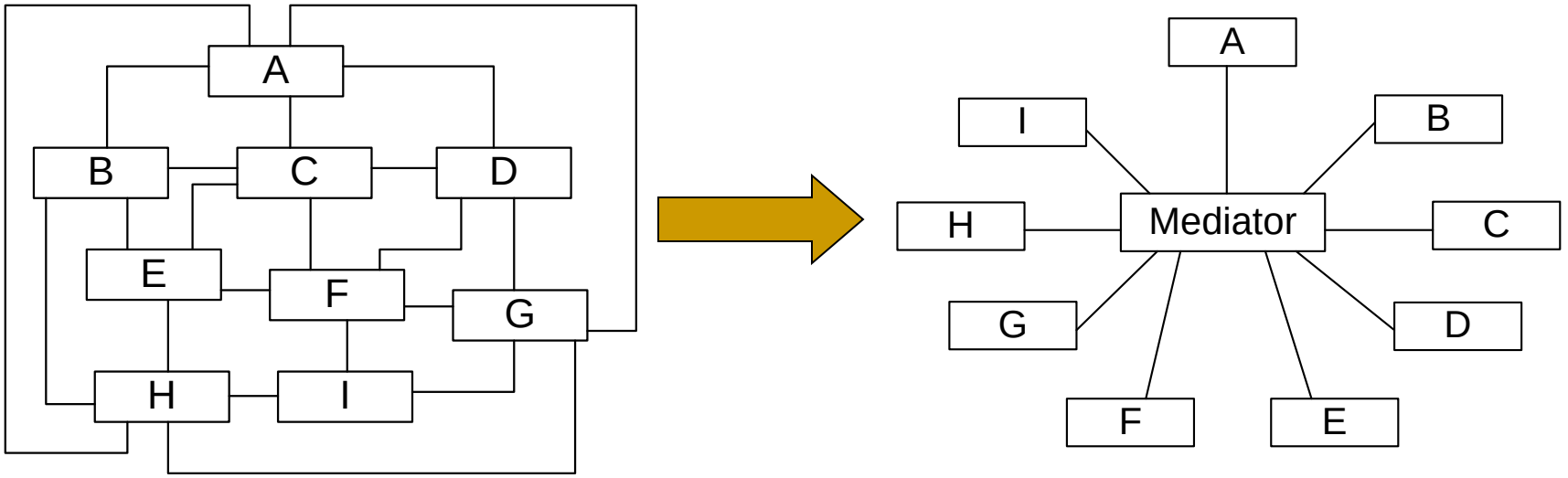
举例：中介者模式的使用

模式的结构中包括四种角色：

- 中介者（Mediator）
- 具体中介者（ConcreteMediator）
- 同事（Colleague）
- 具体同事（ConcreteColleague）

举例：中介者模式的使用

- 模式分析
  - 中介者模式可以使对象之间的关系数量急剧减少：



举例：中介者模式的使用

- 模式分析
  - 中介者承担两方面的职责
    - 中转作用（结构性）：通过中介者提供的中转作用，各个同事对象就不再需要显式引用其他同事，当需要和其他同事进行通信时，通过中介者即可。该中转作用属于中介者在结构上的支持

举例：中介者模式的使用

- 模式分析
  - 中介者承担两方面的职责
    - 协调作用（行为性）：中介者可以更进一步的对同事之间的关系进行封装，同事可以一致地和中介者进行交互，而不需要指明中介者需要具体怎么做，中介者根据封装在自身内部的协调逻辑，对同事的请求进行进一步处理，将同事成员之间的关系行为进行分离和封装。该协调作用属于中介者在行为上的支持

举例：中介者模式的使用

- 例子：
- 古代相互交战的A,B,C三方，想通过一个中介者调停之间的战火
  - A方委托调停者转达的信息是：“要求B方归还曾抢夺的100斤土豆，要求C方归还曾抢夺的20头牛”
  - B方委托调停者转达的信息是：“要求A方归还曾抢夺的10只鸡，要求C方归还曾抢夺的15匹马”
  - C方委托调停者转达的信息是：“要求A方归还曾抢夺的300斤小麦，要求B方归还曾抢夺的50头驴”

举例：中介者模式的使用

1．同事（Colleague）：Colleague.java

```
public interface Colleague{
    public void giveMess(String [] mess);
    public void receiverMess(String mess);
    public void setName(String name);
    public String getName();
}
```

注：本问题中，只需要一个具体中介者，我们并不需要一个中介者（Mediator）接口 。

举例：中介者模式的使用

2. 具体中介者（Mediator）：ConcreteMediator.java

```
public class ConcreteMediator{
    ColleagueA colleagueA;
    ColleagueB colleagueB;
    ColleagueC colleagueC;
    public void registerColleagueA(ColleagueA colleagueA){
       this.colleagueA=colleagueA;
    }
    public void registerColleagueB(ColleagueB colleagueB){
       this.colleagueB=colleagueB;
    }
    public void registerColleagueC(ColleagueC colleagueC){
       this.colleagueC=colleagueC;
    }
    public void deliverMess(Colleague colleague,String [] mess){
       if(colleague==colleagueA){
          if(mess.length>=2){
            colleagueB.receiverMess(colleague.getName()+mess[0]);
            colleagueC.receiverMess(colleague.getName()+mess[1]);
          }
       }
       else if(colleague==colleagueB){
          if(mess.length>=2){
            colleagueA.receiverMess(colleague.getName()+mess[0]);
            colleagueC.receiverMess(colleague.getName()+mess[1]);
          }
       }
       else if(colleague==colleagueC){
          if(mess.length>=2){
            colleagueA.receiverMess(colleague.getName()+mess[0]);
            colleagueB.receiverMess(colleague.getName()+mess[1]);
          }
       }
    }
}
```

举例：中介者模式的使用

3．具体同事（ConcreteColleague）_1：ColleagueA.java

```
public class ColleagueA implements Colleague{
    ConcreteMediator mediator; String name;
    ColleagueA(ConcreteMediator mediator){
       this.mediator=mediator;
       mediator.registerColleagueA(this);
    }
    public void setName(String name){
       this.name=name;
    }
    public String getName(){
        return name;
    }
    public void giveMess(String [] mess){
        mediator.deliverMess(this,mess);
    }
    public void receiverMess(String mess){
       System.out.println(name+"收到的信息:");
       System.out.println("\t"+mess);
    }
}
```

举例：中介者模式的使用

4. 应用 Application.java

```
public class Application{
    public static void main(String args[]){
        ConcreteMediator mediator=new ConcreteMediator();
        ColleagueA colleagueA=new ColleagueA(mediator);
        ColleagueB colleagueB=new ColleagueB(mediator);
        ColleagueC colleagueC=new ColleagueC(mediator);
        colleagueA.setName("A国");
        colleagueB.setName("B国");
        colleagueC.setName("C国");
        String [] messA={"要求归还曾抢夺的100斤土豆","要求归还曾抢夺的20头牛"};
        colleagueA.giveMess(messA);
        String [] messB={"要求归还曾抢夺的10只公鸡","要求归还曾抢夺的15匹马"};
        colleagueB.giveMess(messB);
        String [] messC={"要求归还曾抢夺的300斤小麦","要求归还曾抢夺的50头驴"};
        colleagueC.giveMess(messC);
    }
}
```

Thanks!