南京大学

设计模式–结构型模式(一)

Design Pattern-Structural Pattern (1)

结构型模式概述

- **结构型模式(Structural Pattern)**关注如何将现有类或对象组织在一起形成更加强大的结构
- 不同的结构型模式从不同的角度组合类或对象，它们在尽可能满足各种面向对象设计原则的同时为类或对象的组合提供一系列巧妙的解决方案

结构型模式概述

- 类结构型模式
  - 关心类的组合，由多个类组合成一个更大的系统，在类结构型模式中一般只存在继承关系和实现关系
- 对象结构型模式
  - 关心类与对象的组合，通过关联关系，在一个类中定义另一个类的实例对象，然后通过该对象调用相应的方法

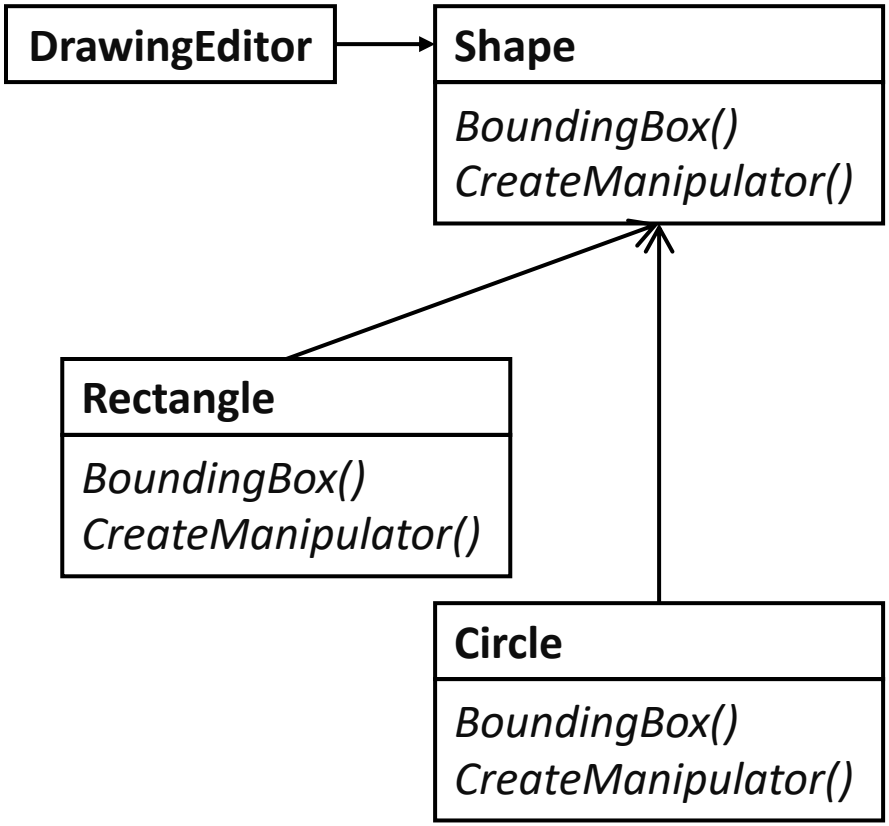
结构型模式概述

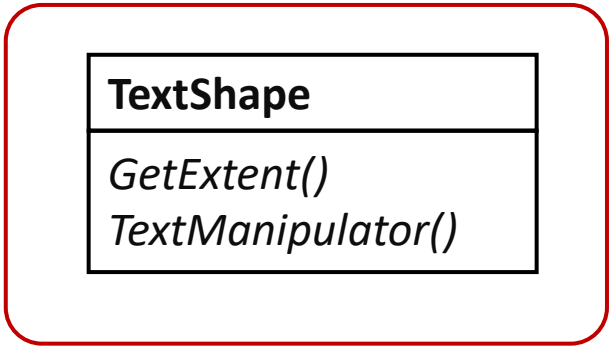
模式名称	定 义	学习难度	使用频率
适配器模式 (Adapter Pattern)	将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作。	★★☆☆☆	★★★★☆
桥接模式 (Bridge Pattern)	将抽象部分与它的实现部分解耦，使得两者都能够独立变化。	★★★☆☆	★★★☆☆
组合模式 (Composite Pattern)	组合多个对象形成树形结构，以表示具有部分-整体关系的层次结构。组合模式让客户端可以统一对待单个对象和组合对象。	★★★☆☆	★★★★☆
装饰模式 (Decorator Pattern)	动态地给一个对象增加一些额外的职责。就扩展功能而言，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。	★★★☆☆	★★★★☆

适配器模式概述

- 电源适配器

适配器模式概述





第三方库如何复用

适配器模式概述

- 分析
  - 现实生活：
    - 不兼容：生活用电**220V** ←→ 笔记电脑**20V**
    - 引入 AC Adapter（交流电适配器）
  - 软件开发：
    - 存在不兼容的结构，例如方法名不一致
    - 引入适配器模式

适配器模式概述

- 适配器模式的定义

适配器模式：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。

Adapter Pattern: Convert the interface of a class into another interface clients expect. Adapter lets classes **work together** that couldn't otherwise because of **incompatible** interfaces.

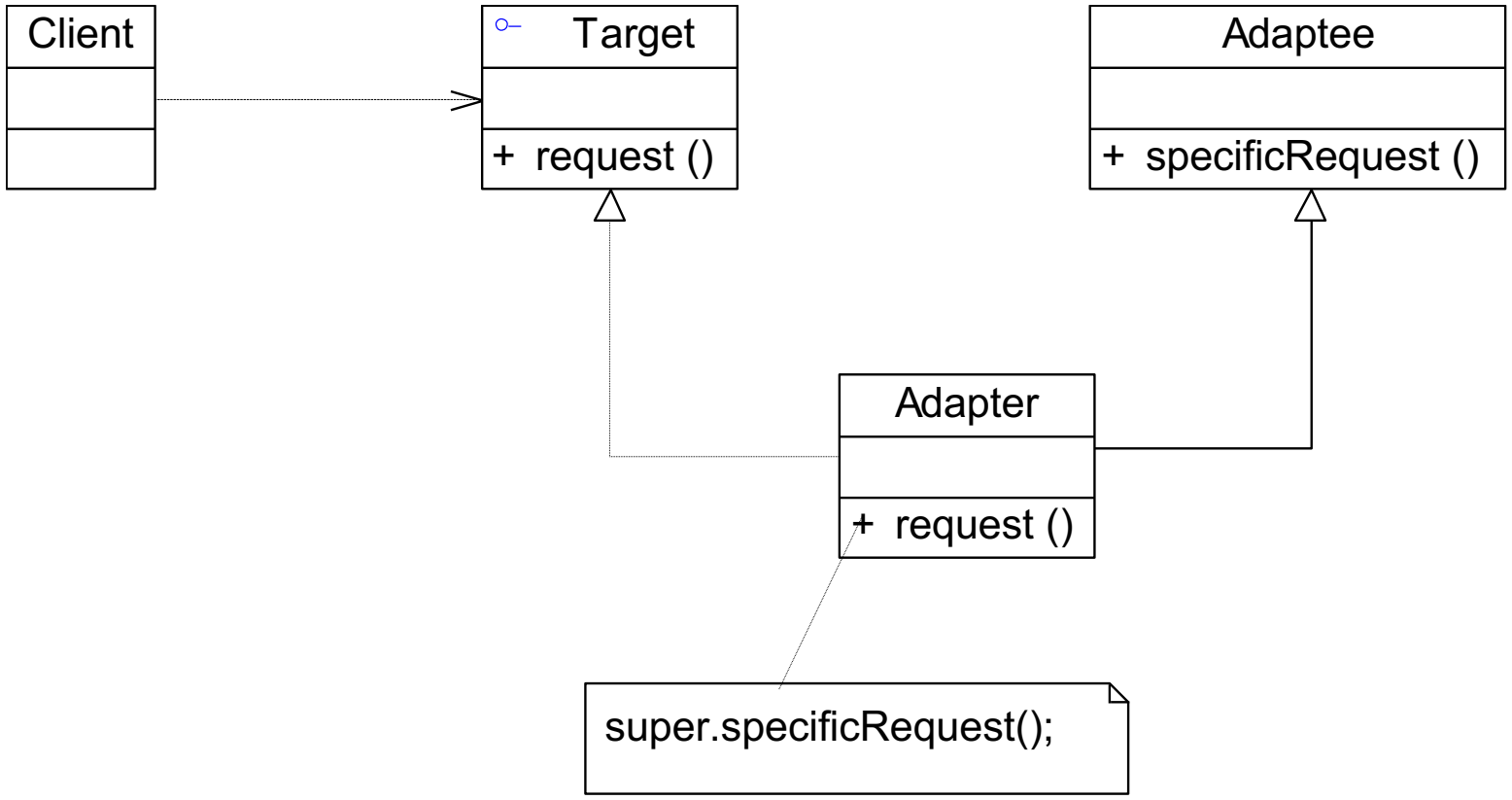
- 对象结构型模式 / 类结构型模式

适配器模式概述

- 适配器模式的定义
  - 别名为包装器(Wrapper)模式
  - 定义中所提及的接口是指广义的接口，它可以表示一个方法或者方法的集合

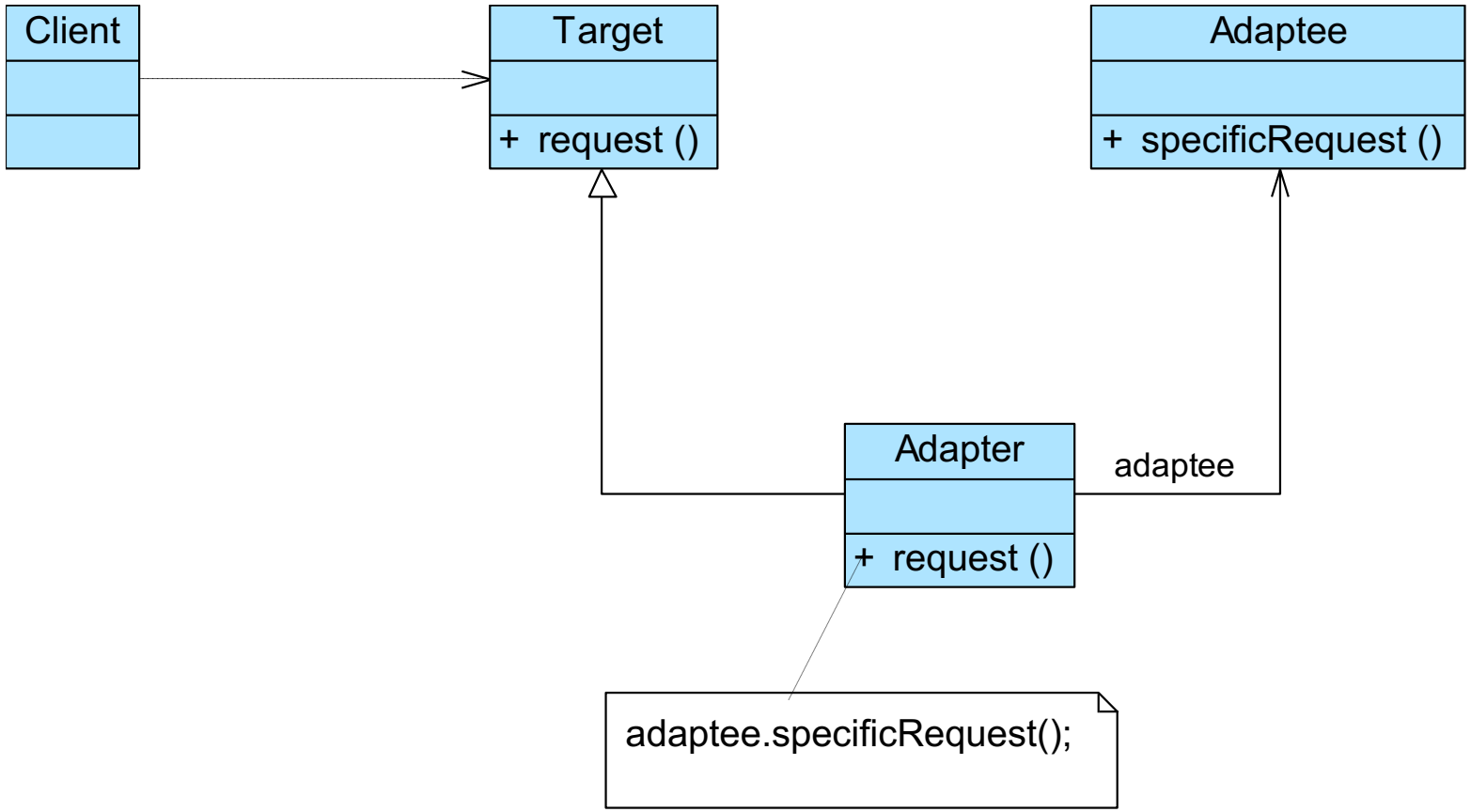
适配器模式的结构与实现

- 适配器模式的结构（类适配器）



适配器模式的结构与实现

- 适配器模式的结构（对象适配器）



适配器模式的结构与实现

- 适配器模式的结构
  - 适配器模式包含以下3个角色：
    - Target（目标抽象类）
    - Adapter（适配器类）
    - Adaptee（适配者类）

适配器模式的结构与实现

- 适配器模式的实现
  - 典型的类适配器代码：

```
public class Adapter extends Adaptee implements Target {
    public void request() {
        super.specificRequest();
    }
}
```

适配器模式的结构与实现

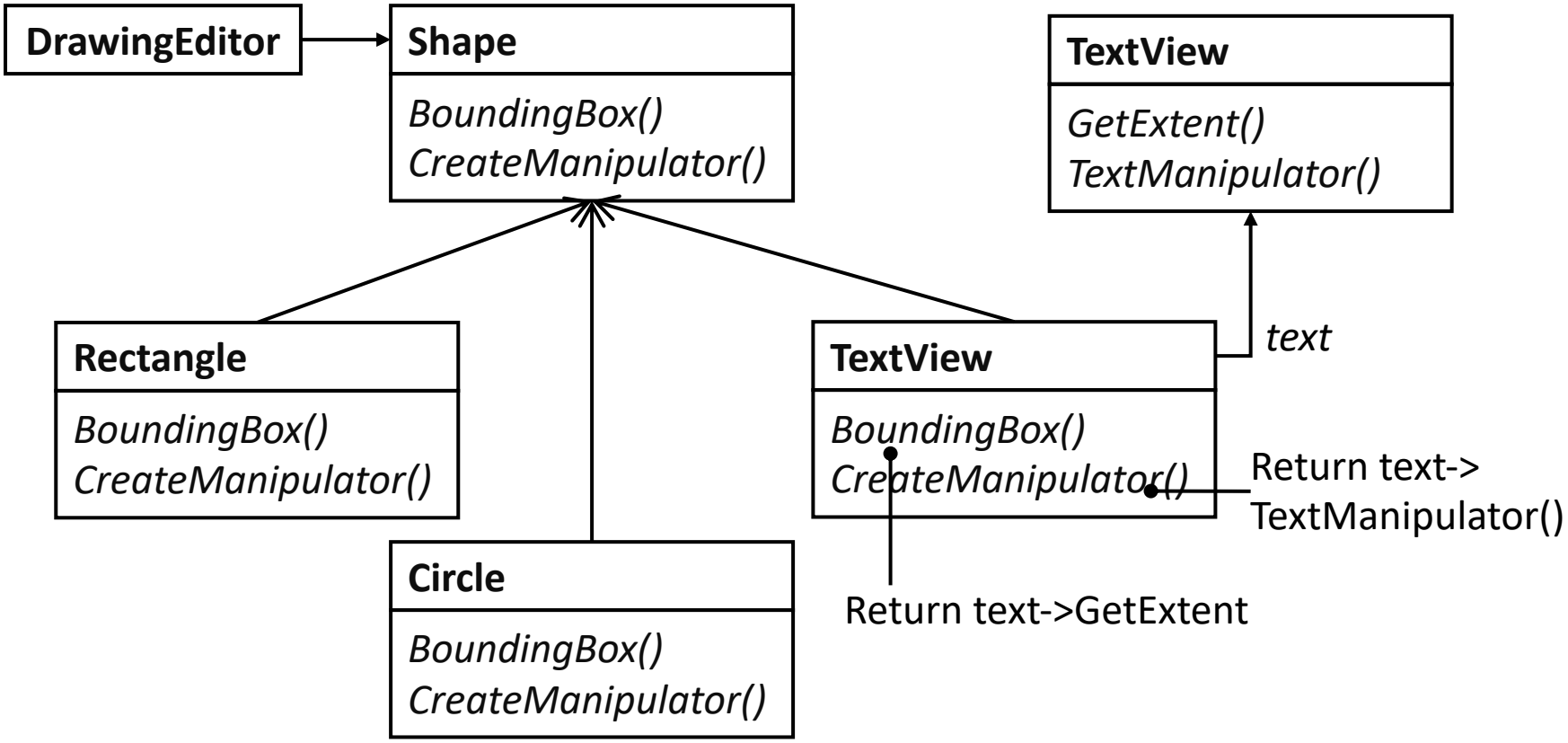
- 适配器模式的实现
  - 典型的对象适配器代码：

```
public class Adapter extends Target {
    private Adaptee adaptee; //维持一个对适配者对象的引用

    public Adapter(Adaptee adaptee) {
        this.adaptee=adaptee;
    }

    public void request() {
        adaptee.specificRequest(); //转发调用
    }
}
```

适配器模式概述


DrawingEditor
Shape
BoundingBox()
CreateManipulator()
TextView
GetExtent()
TextManipulator()
Rectangle
BoundingBox()
CreateManipulator()
TextView
BoundingBox()
CreateManipulator()
text
Return text->
TextManipulator()
Circle
BoundingBox()
CreateManipulator()
Return text->GetExtent

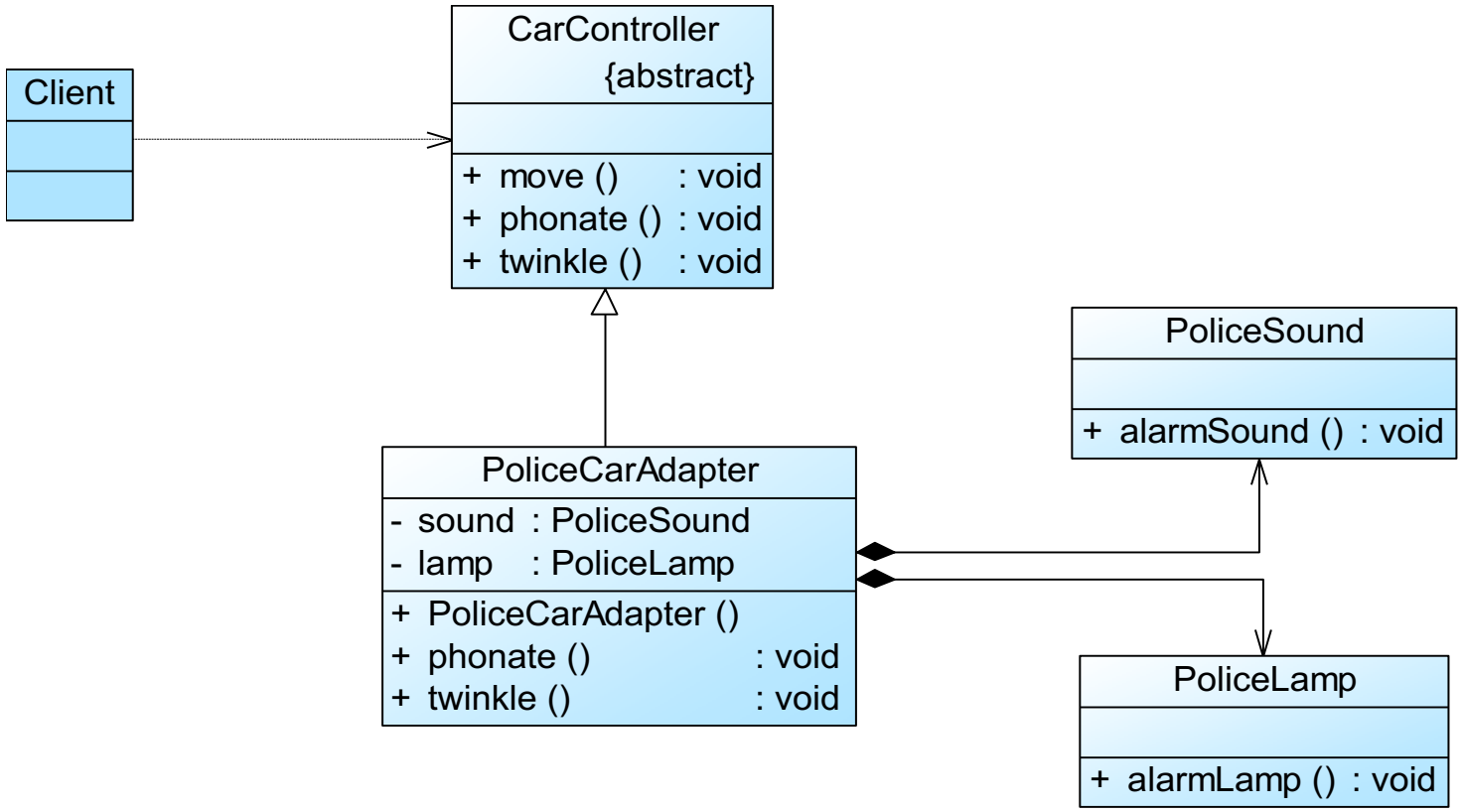
适配器模式的应用实例

- 实例说明

某公司欲开发一款儿童玩具汽车，为了更好地吸引小朋友的注意力，该玩具汽车在移动过程中伴随着灯光闪烁和声音提示。在该公司以往的产品中已经实现了控制灯光闪烁（例如警灯闪烁）和声音提示（例如警笛音效）的程序，为了重用先前的代码并且使得汽车控制软件具有更好的灵活性和扩展性，现使用适配器模式设计该玩具汽车控制软件。

适配器模式的应用实例

- 实例类图



汽车控制软件结构图

适配器模式的应用实例

- 实例代码
  - (1) CarController：汽车控制类，充当目标抽象类
  - (2) PoliceSound：警笛类，充当适配者
  - (3) PoliceLamp：警灯类，充当适配者
  - (4) PoliceCarAdapter：警车适配器，充当适配器
  - (5) Client：客户端测试类
  - (6) XMLUtil：工具类

演示.....

Code (designpatterns.adapter)

适配器模式的应用实例

- 结果及分析
  - 将具体适配器类的类名存储在配置文件中
  - 扩展方便

```
<?xml version="1.0"?>
<config>
<className>designpatterns.adapter.PoliceCarAdapter</className>
</config>
```

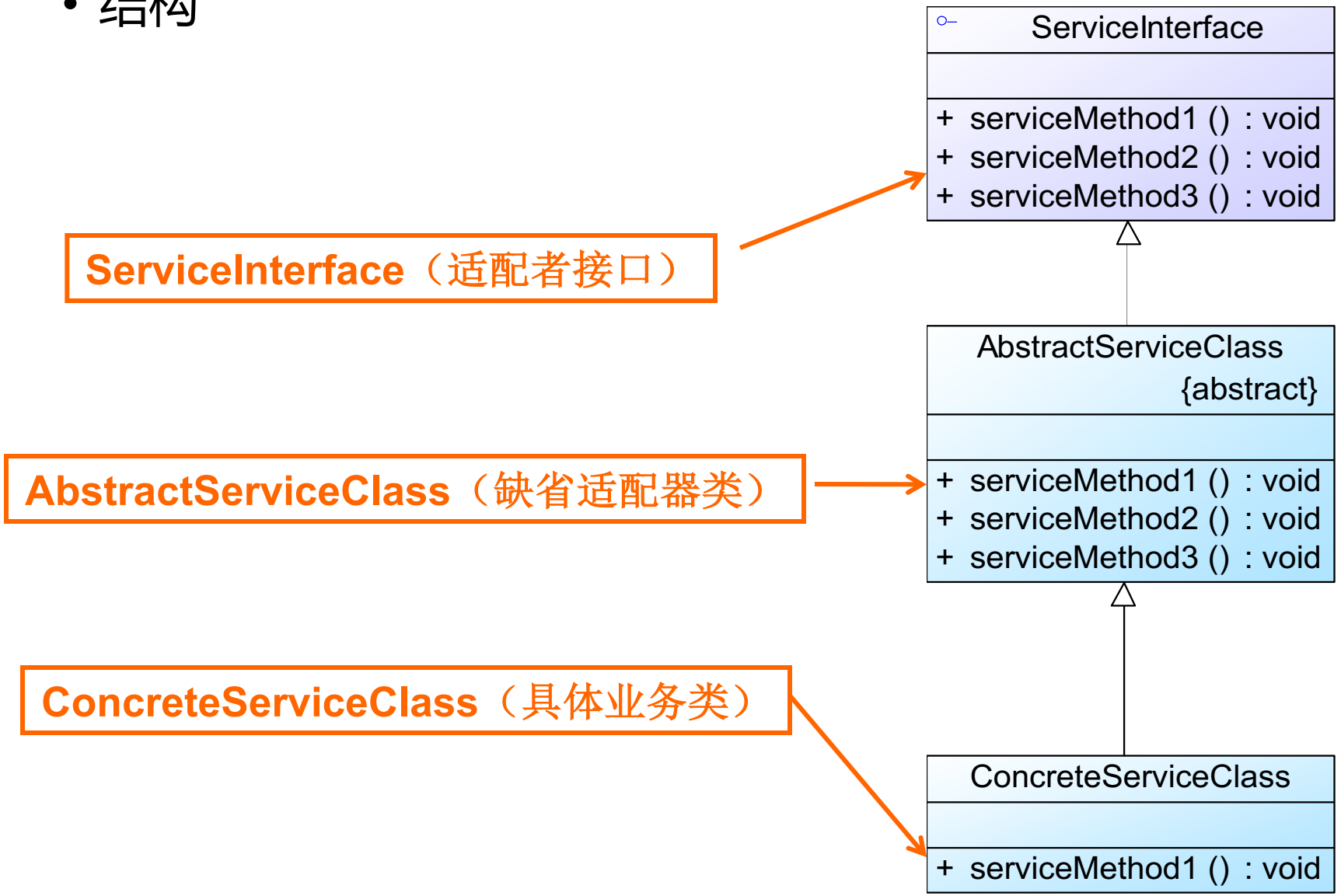
缺省适配器模式

- 定义

缺省适配器模式(Default Adapter Pattern)： 当不需要实现一个接口所提供的所有方法时，可先设计一个抽象类实现该接口，并为接口中每个方法提供一个默认实现（空方法），那么该抽象类的子类可以选择性地覆盖父类的某些方法来实现需求，它适用于不想使用一个接口中的所有方法的情况，又称为**单接口适配器模式**。

缺省适配器模式

- 结构


ServiceInterface
+ serviceMethod1 () : void
+ serviceMethod2 () : void
+ serviceMethod3 () : void
ServiceInterface（适配者接口）
AbstractServiceClass
{abstract}
+ serviceMethod1 () : void
+ serviceMethod2 () : void
+ serviceMethod3 () : void
AbstractServiceClass（缺省适配器类）
ConcreteServiceClass
+ serviceMethod1 () : void
ConcreteServiceClass（具体业务类）

缺省适配器模式

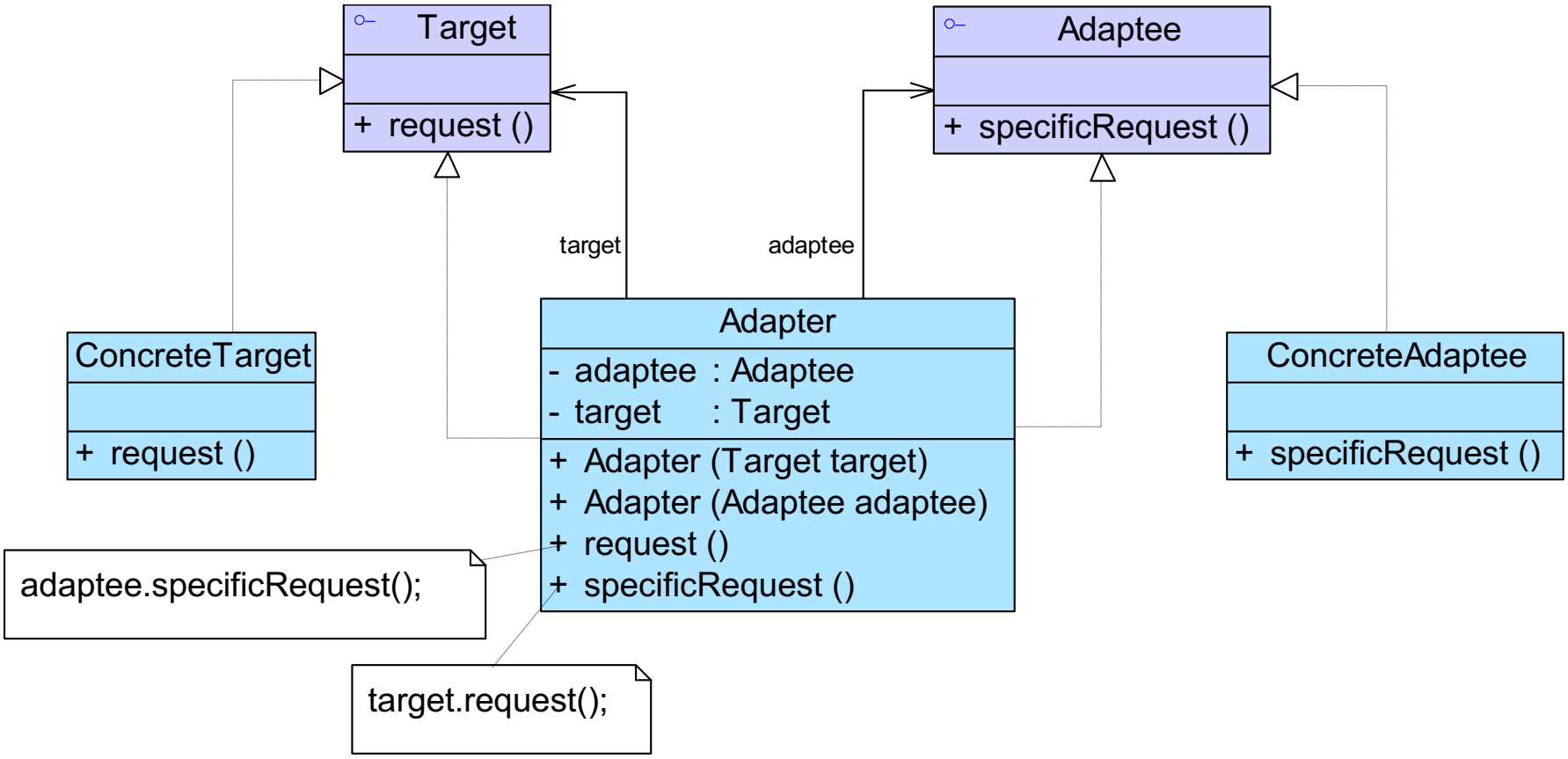
- 实现
  - 缺省适配器类的典型代码片段：

```
public abstract class AbstractServiceClass implements ServiceInterface {
    public void serviceMethod1() { } //空方法
    public void serviceMethod2() { } //空方法
    public void serviceMethod3() { } //空方法
}
```

Java.awt.event中广泛使用了缺省适配器模式，例如WindowAdapter、KeyAdapter、MouseAdapter

双向适配器

- 结构


Target
+ request ()
Adaptee
+ specificRequest ()
target
adaptee
ConcreteTarget
+ request ()
Adapter
- adaptee : Adaptee
- target : Target
+ Adapter (Target target)
+ Adapter (Adaptee adaptee)
+ request ()
+ specificRequest ()
ConcreteAdaptee
+ specificRequest ()
adaptee.specificRequest();
target.request();

双向适配器

- 实现

```
public class Adapter implements Target,Adaptee {
    private Target target;
    private Adaptee adaptee;

    public Adapter(Target target) {
        this.target = target;
    }

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    public void request() {
        adaptee.specificRequest();
    }

    public void specificRequest() {
        target.request();
    }
}
```

适配器模式的优缺点与适用环境

- 模式优点
  - 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，无须修改原有结构
  - 增加了类的透明性和复用性，提高了适配者的复用性，同一个适配者类可以在多个不同的系统中复用
  - 灵活性和扩展性非常好
  - 类适配器模式：置换一些适配者的方法很方便
  - 对象适配器模式：可以把多个不同的适配者适配到同一个目标，还可以适配一个适配者的子类

适配器模式的优缺点与适用环境

- 模式缺点
  - 类适配器模式：
    - (1) 一次最多只能适配一个适配者类，不能同时适配多个适配者；
    - (2) 适配者类不能为最终类；
    - (3) 目标抽象类只能为接口，不能为类
  - 对象适配器模式：
    - 在适配器中置换适配者类的某些方法比较麻烦

适配器模式的优缺点与适用环境

- 模式适用环境
  - 系统需要使用一些现有的类，而这些类的接口不符合系统的需要，甚至没有这些类的源代码
  - 创建一个可以重复使用的类，用于和一些彼此之间没有太大关联的类，包括一些可能在将来引进的类一起工作

思考

- 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？

桥接模式概述

- 毛笔与蜡笔的绘画功能

毛笔与调色板

不同型号的蜡笔

12种颜色，3种型号

3支毛笔+
12种颜色
的调色板

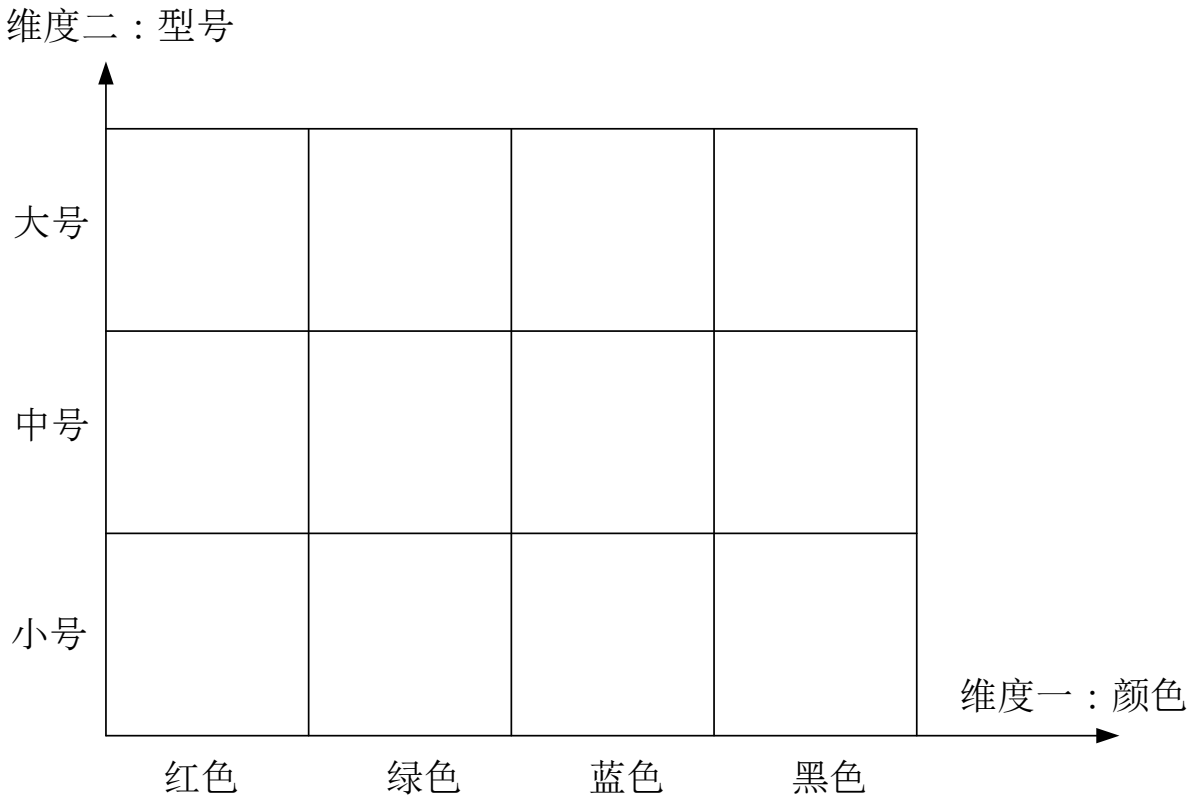
36支蜡笔

桥接模式概述

- 分析
  - 蜡笔：颜色和型号两个不同的变化维度（即两个不同的变化原因）耦合在一起，无论是对颜色进行扩展还是对型号进行扩展都势必会影响另一个维度
  - 毛笔：颜色和型号实现了分离，增加新的颜色或者型号对另一方没有任何影响

桥接模式概述

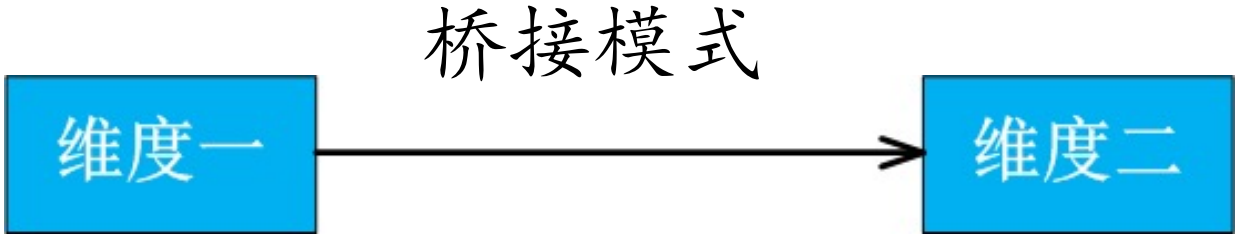
- 分析

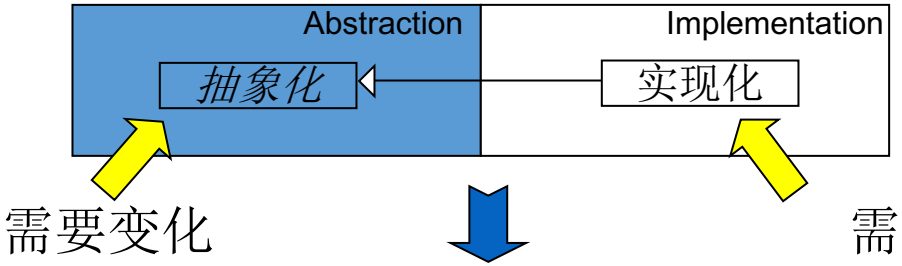


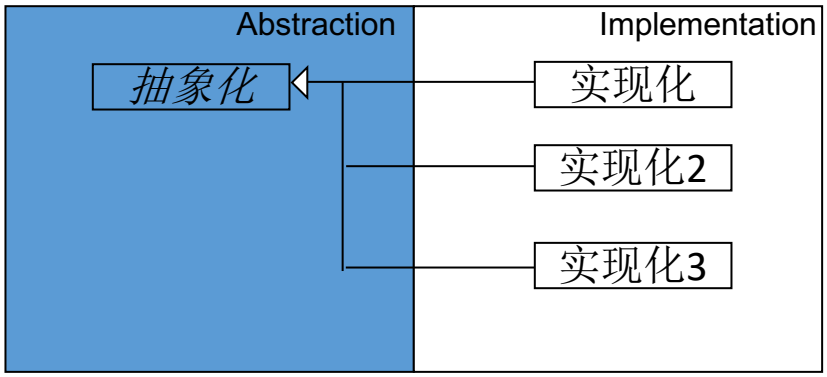
画笔中存在的两个独立变化维度示意图

桥接模式概述

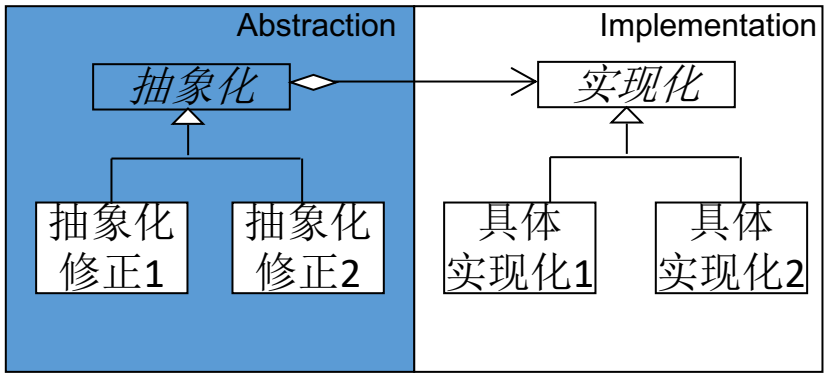
- 在软件开发中如何将多个变化维度分离？







Bridge!



桥接模式概述

- 桥接模式的定义

桥接模式：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。

Bridge Pattern: **Decouple** an **abstraction** from its **implementation** so that the two can vary independently.

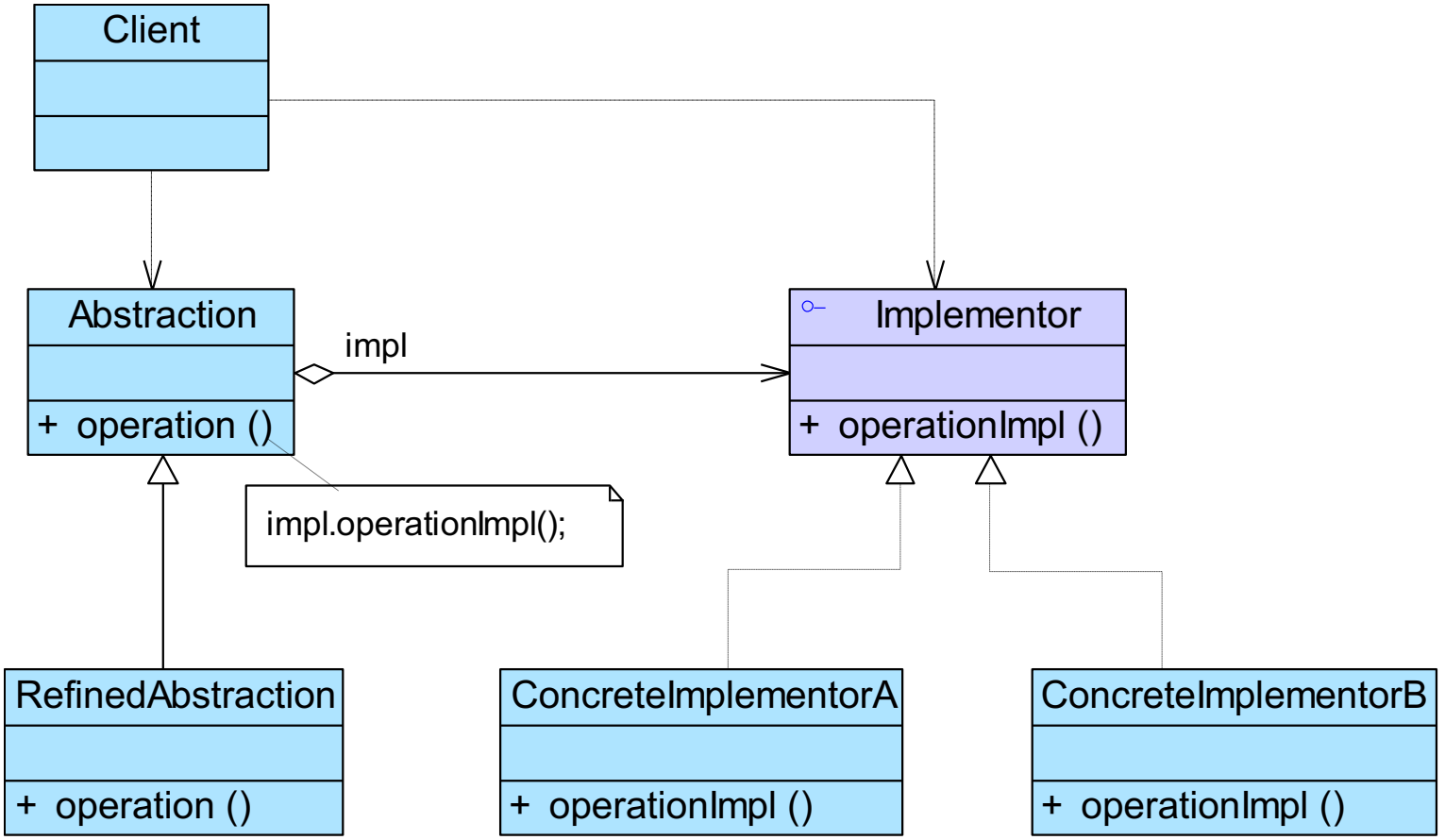
- 对象结构型模式

桥接模式概述

- 桥接模式的定义
  - 又被称为柄体(Handle and Body)模式或接口(Interface)模式
  - 用抽象关联取代了传统的多层继承
  - 将类之间的静态继承关系转换为动态的对象组合关系

桥接模式的结构与实现

- 桥接模式的结构



桥接模式的结构与实现

- 桥接模式的结构
  - 桥接模式包含以下4个角色：
    - Abstraction（抽象类）
    - RefinedAbstraction（扩充抽象类）
    - Implementor（实现类接口）
    - ConcreteImplementor（具体实现类）

桥接模式的结构与实现

- 桥接模式的实现
  - 典型的实现类接口代码：

```
public interface Implementor {
    public void operationImpl();
}
```

桥接模式的结构与实现

- 桥接模式的实现
  - 典型的具体实现类代码：

```
public class ConcreteImplementor implements Implementor {
    public void operationImpl() {
        //具体业务方法的实现
    }
}
```

桥接模式的结构与实现

- 桥接模式的实现
  - 典型的抽象类代码：

```
public abstract class Abstraction {
    protected Implementor impl; //定义实现类接口对象

    public void setImpl(Implementor impl) {
        this.impl=impl;
    }

    public abstract void operation(); //声明抽象业务方法
}
```

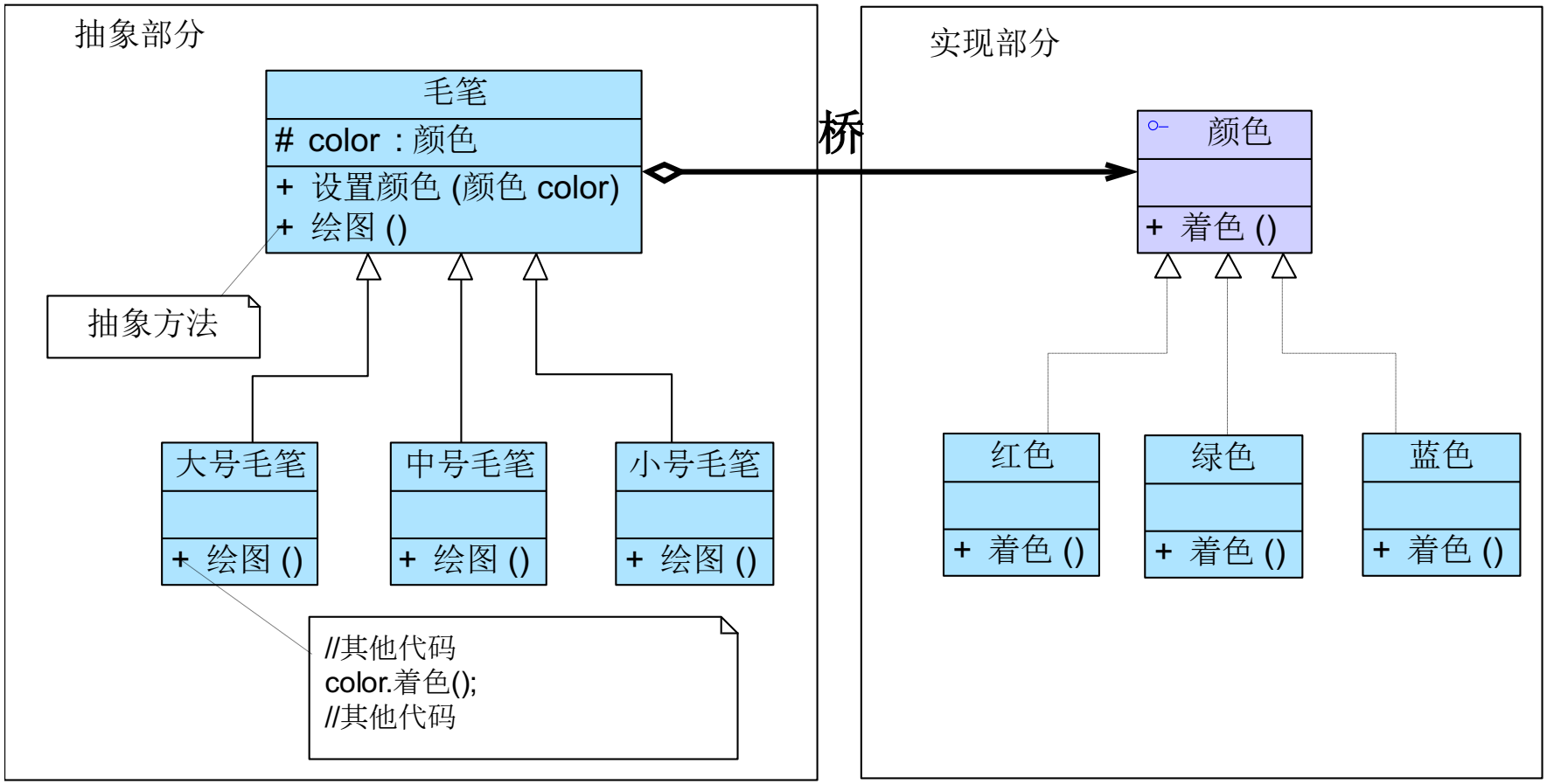
桥接模式的结构与实现

- 桥接模式的实现
  - 典型的扩充抽象类（细化抽象类）代码：

```
public class RefinedAbstraction extends Abstraction {
    public void operation() {
        //业务代码
        impl.operationImpl(); //调用实现类的方法
        //业务代码
    }
}
```

桥接模式的结构与实现

- 桥接模式的实现



毛笔结构示意图

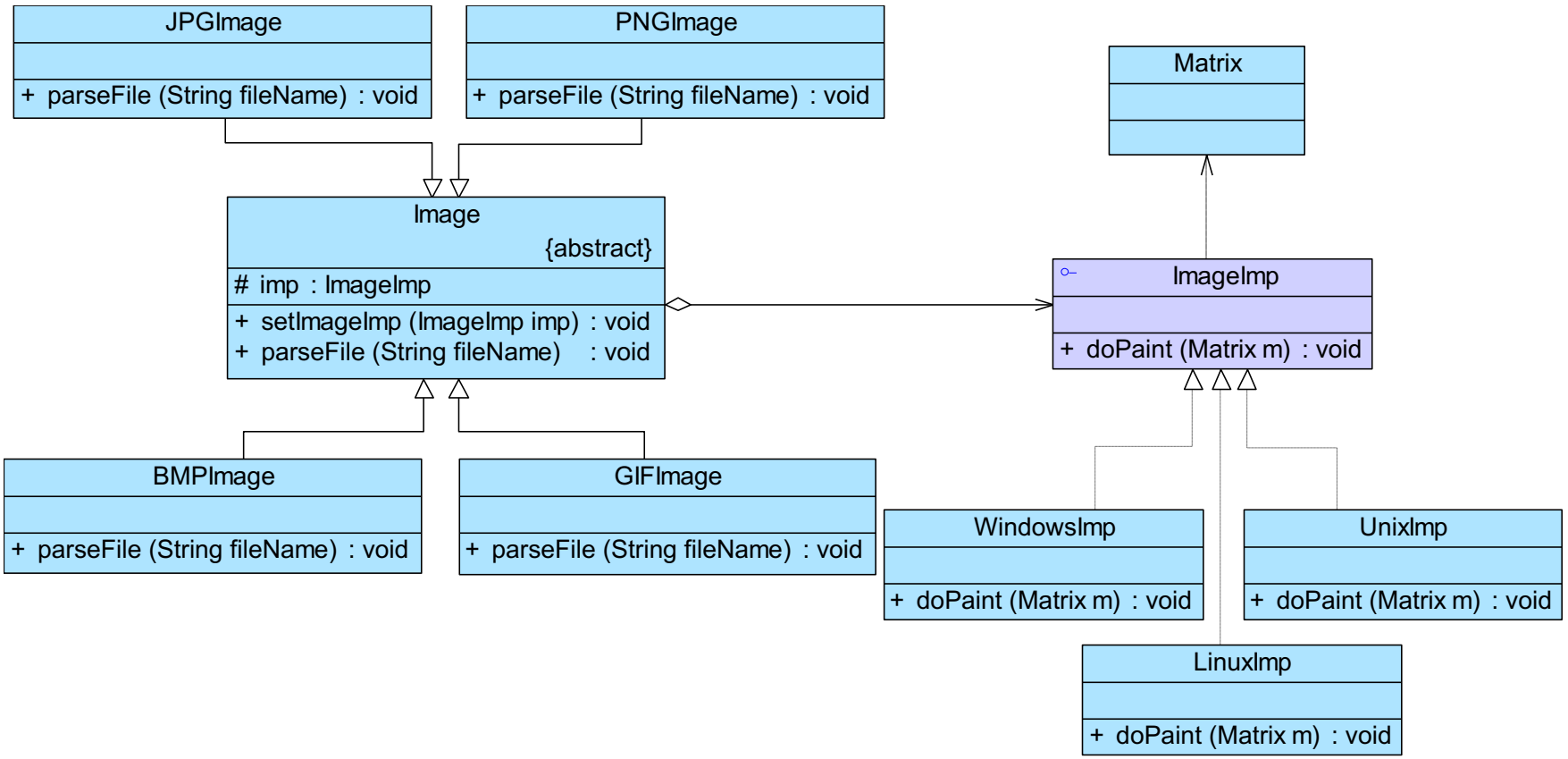
桥接模式的应用实例

- 实例说明

某软件公司要开发一个跨平台图像浏览系统，要求该系统能够显示BMP、JPG、GIF、PNG等多种格式的文件，并且能够在Windows、Linux、UNIX等多个操作系统上运行。系统首先将各种格式的文件解析为像素矩阵(Matrix)，然后将像素矩阵显示在屏幕上，在不同的操作系统中可以调用不同的绘制函数来绘制像素矩阵。另外，系统需具有较好的扩展性，以便在将来支持新的文件格式和操作系统。试使用桥接模式设计该跨平台图像浏览系统。

桥接模式的应用实例

- 实例类图



跨平台图像浏览系统结构图

桥接模式的应用实例

- 实例代码
  - (1) Matrix：像素矩阵类，辅助类
  - (2) ImageImp：抽象操作系统实现类，充当实现类接口
  - (3) WindowsImp：Windows操作系统实现类，充当具体实现类
  - (4) LinuxImp：Linux操作系统实现类，充当具体实现类
  - (5) UnixImp：UNIX操作系统实现类，充当具体实现类
  - (6) Image：抽象图像类，充当抽象类
  - (7) JPGImage：JPG格式图像类，充当扩充抽象类
  - (8) PNGImage：PNG格式图像类，充当扩充抽象类
  - (9) BMPImage：BMP格式图像类，充当扩充抽象类
  - (10) GIFImage：GIF格式图像类，充当扩充抽象类
  - (11) Client：客户端测试类

演示.....

Code (designpatterns.bridge)

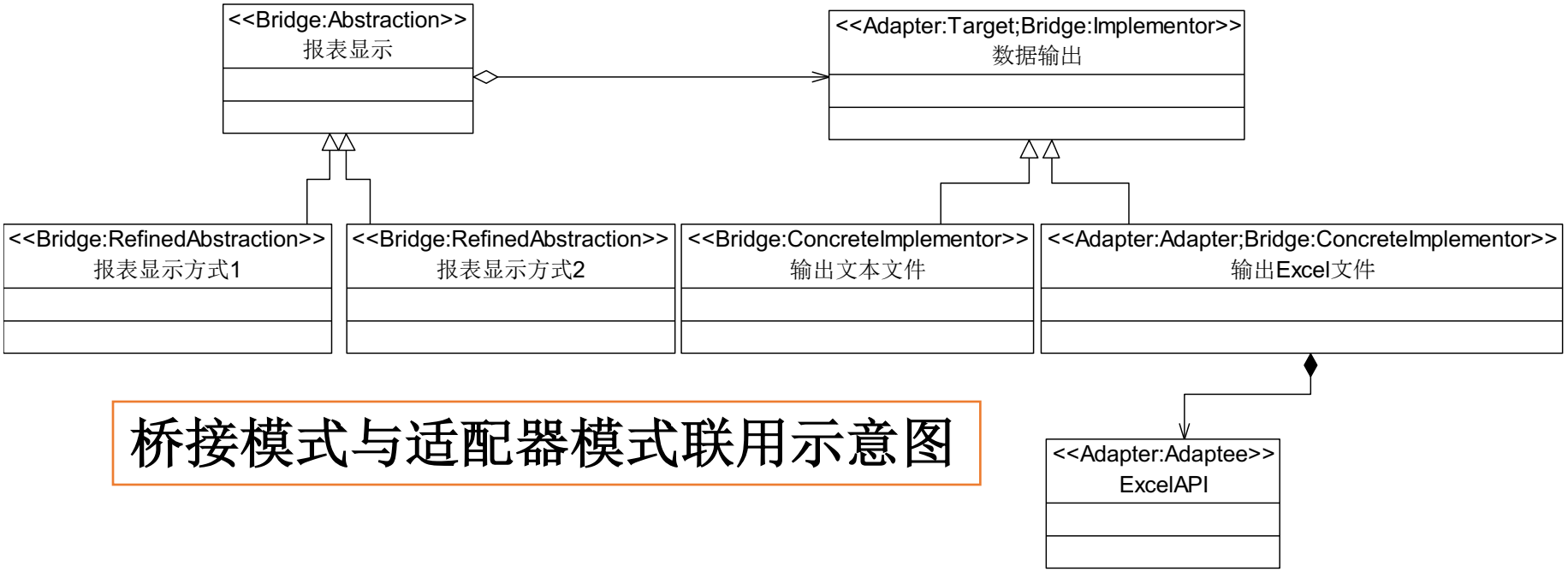
桥接模式的应用实例

- 结果及分析
  - 如果需要更换图像文件格式或者更换操作系统，只需修改配置文件即可

```
<?xml version="1.0"?>
<config>
    <!--RefinedAbstraction-->
    <className>designpatterns.bridge.JPGImage</className>
    <!--ConcreteImplementor-->
    <className>designpatterns.bridge.WindowsImp</className>
</config>
```

桥接模式与适配器模式的联用

- 桥接模式：用于系统的初步设计，对于存在两个独立变化维度的类可以将其分为抽象化和实现化两个角色，使它们可以分别进行变化
- 适配器模式：当发现系统与已有类无法协同工作时



桥接模式与适配器模式联用示意图

桥接模式的优缺点与适用环境

- 模式优点
  - 分离抽象接口及其实现部分
  - 可以取代多层继承方案，极大地减少了子类的个数
  - 提高了系统的可扩展性，在两个变化维度中任意扩展一个维度，不需要修改原有系统，符合开闭原则

桥接模式的优缺点与适用环境

- 模式缺点
  - 会增加系统的理解与设计难度，由于关联关系建立在抽象层，要求开发者一开始就针对抽象层进行设计与编程
  - 正确识别出系统中两个独立变化的维度并不是一件容易的事情

桥接模式的优缺点与适用环境

- 模式适用环境
  - 需要在抽象化和具体化之间增加更多的灵活性，避免在两个层次之间建立静态的继承关系
  - 抽象部分和实现部分可以以继承的方式独立扩展而互不影响
  - 一个类存在两个（或多个）独立变化的维度，且这两个（或多个）维度都需要独立地进行扩展
  - 不希望使用继承或因为多层继承导致系统类的个数急剧增加的系统

思考

- 如果系统中存在两个以上的变化维度，是否可以使用桥接模式进行处理？如果可以，系统该如何设计？