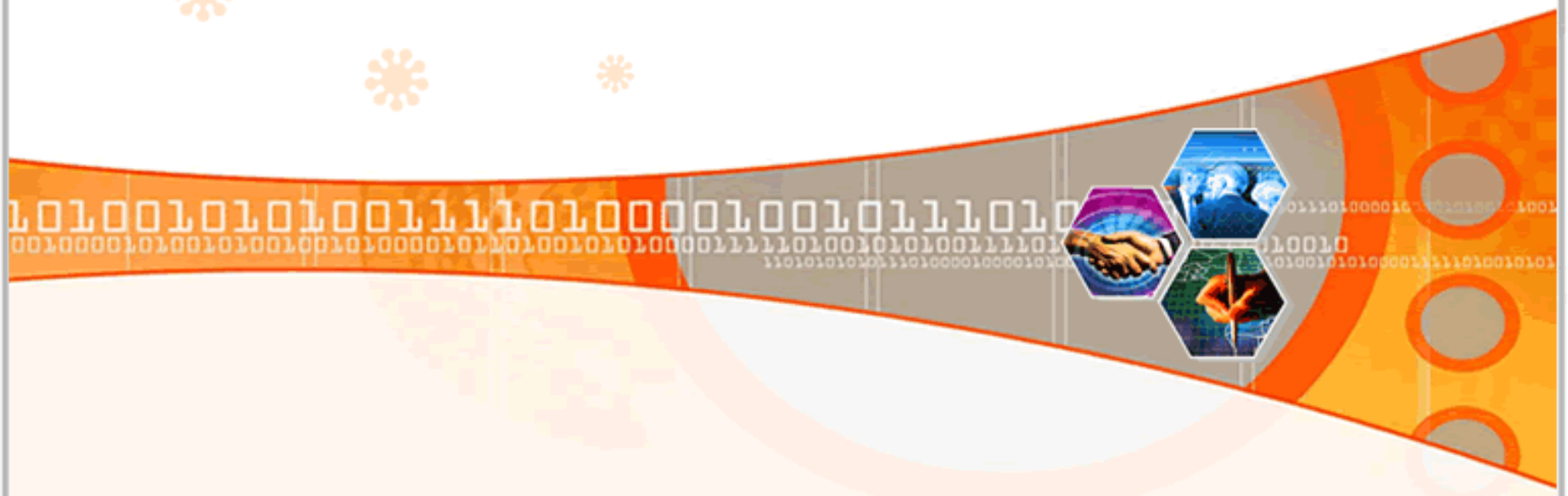
Design Patterns

桥接模式



大纲

- 桥接模式概述
- 桥接模式的结构与实现
- 桥接模式的应用实例
- 桥接模式与适配器模式的联用
- 桥接模式的优缺点与适用环境

桥接模式概述

- 毛笔与蜡笔的“故事”

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

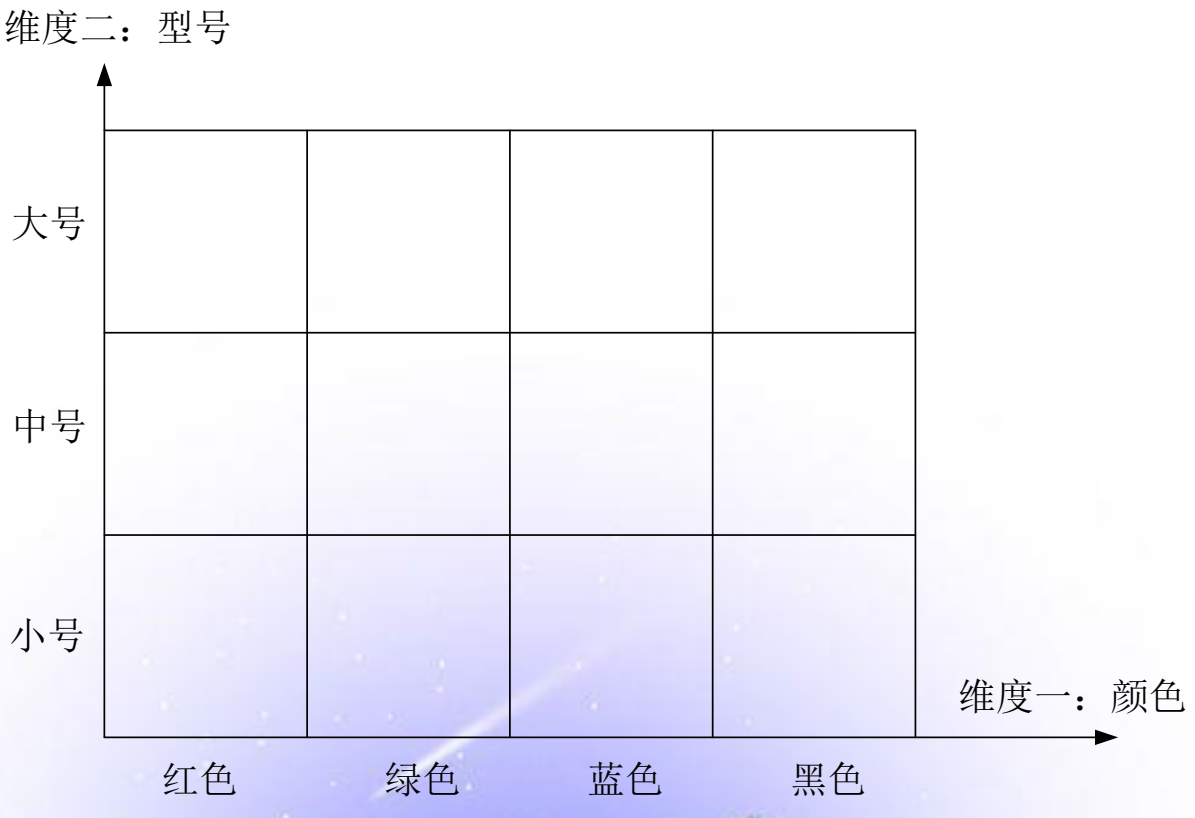
36支蜡笔

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？

桥接模式

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.

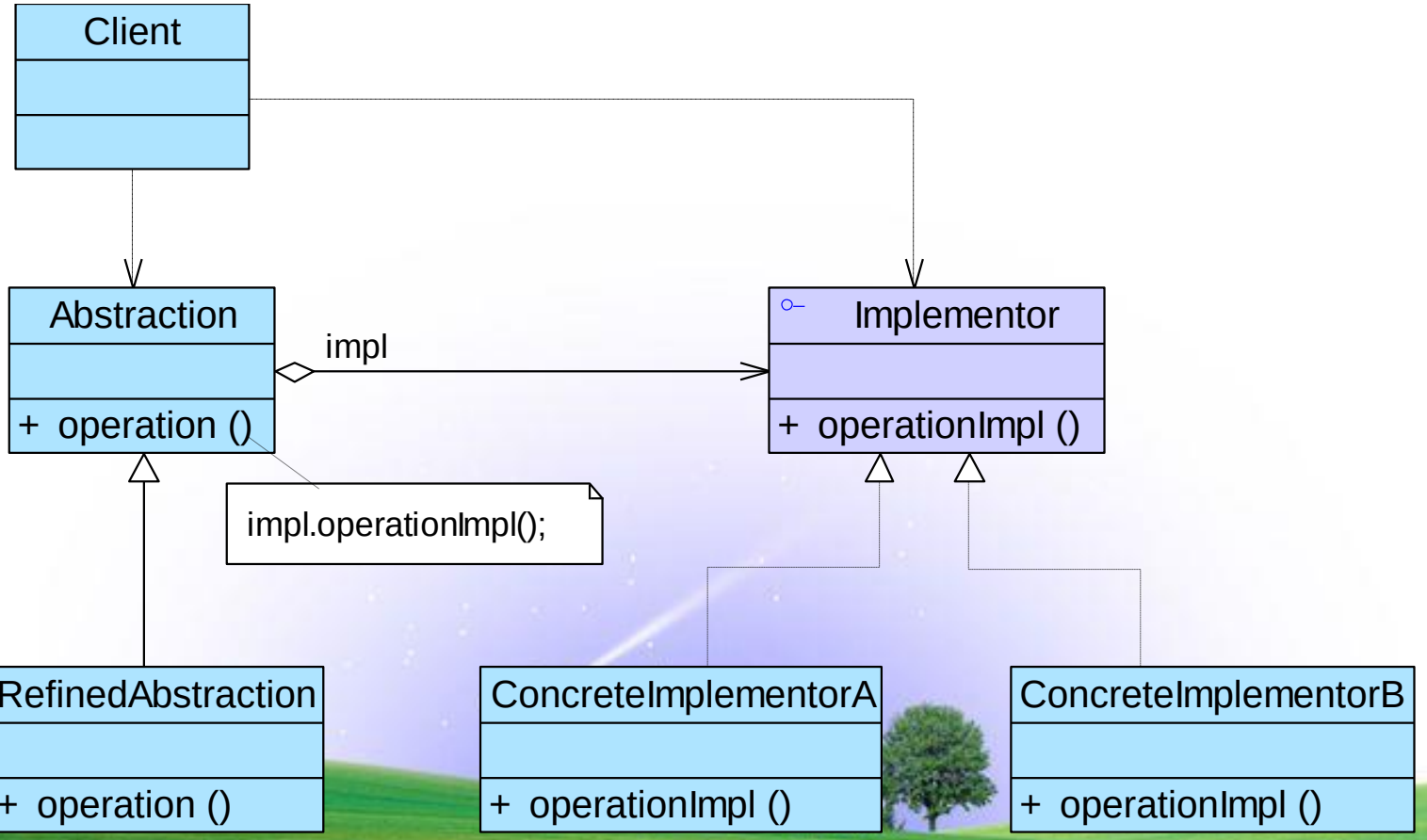
✓对象结构型模式

桥接模式概述

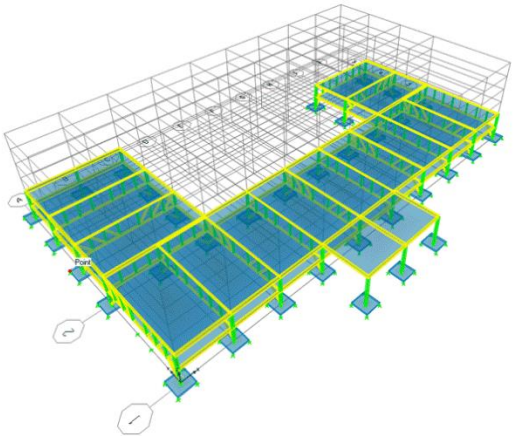
- 桥接模式的定义
  - ✓ 又被称为柄体(Handle and Body)模式或接口(Interface)模式
  - ✓ 用抽象关联取代了传统的多层继承
  - ✓ 将类之间的静态继承关系转换为动态的对象组合关系

桥接模式的结构与实现

- 桥接模式的结构



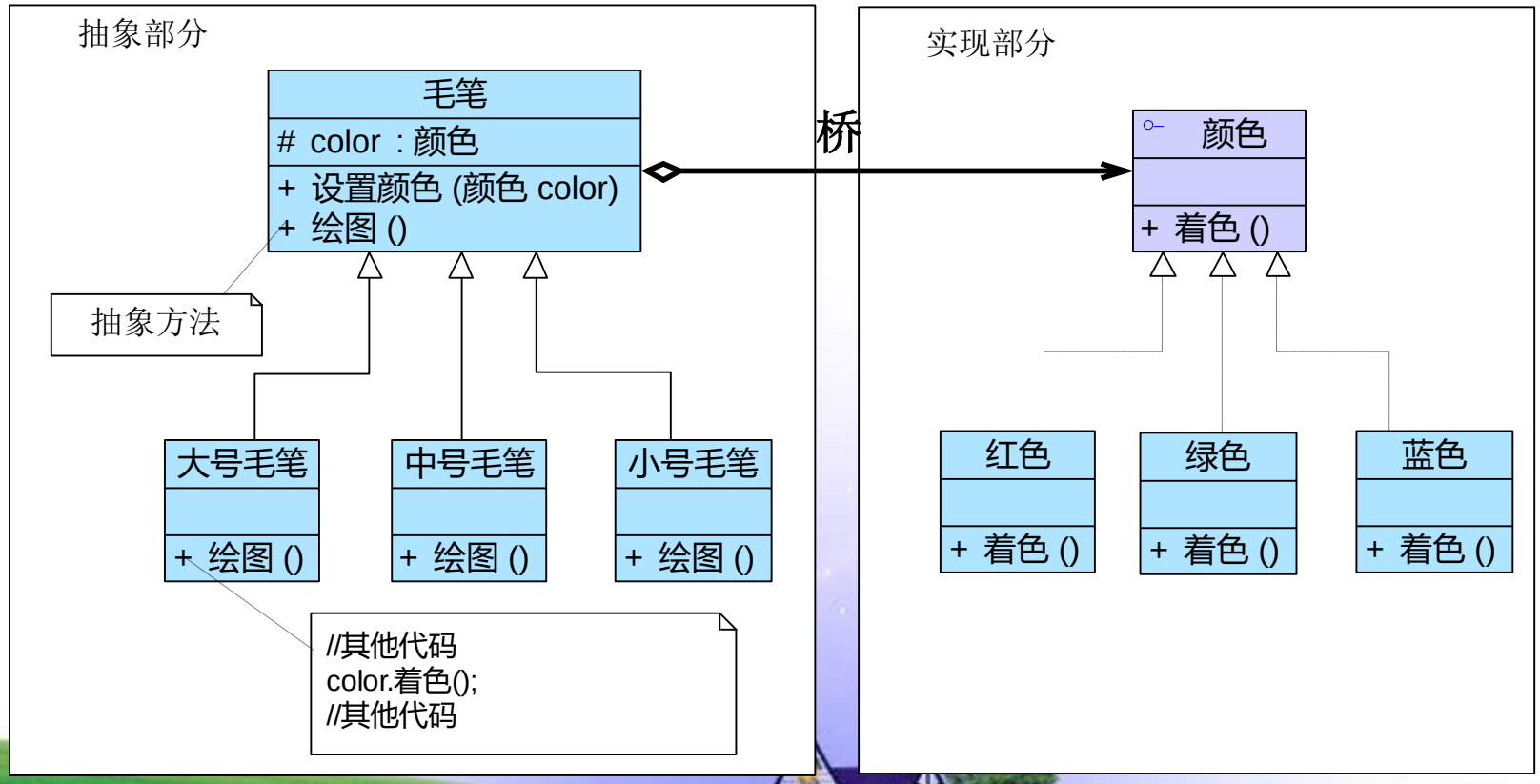
桥接模式的结构与实现



- 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（实现器接口）
    - ConcreteImplementor（具体实现器）

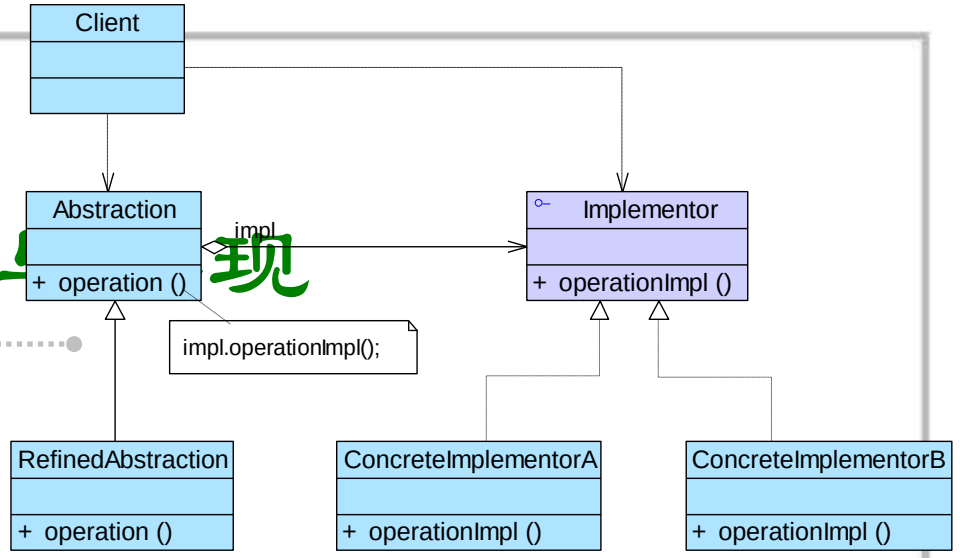
桥接模式的结构与实现

- 桥接模式的实现



毛笔结构示意图

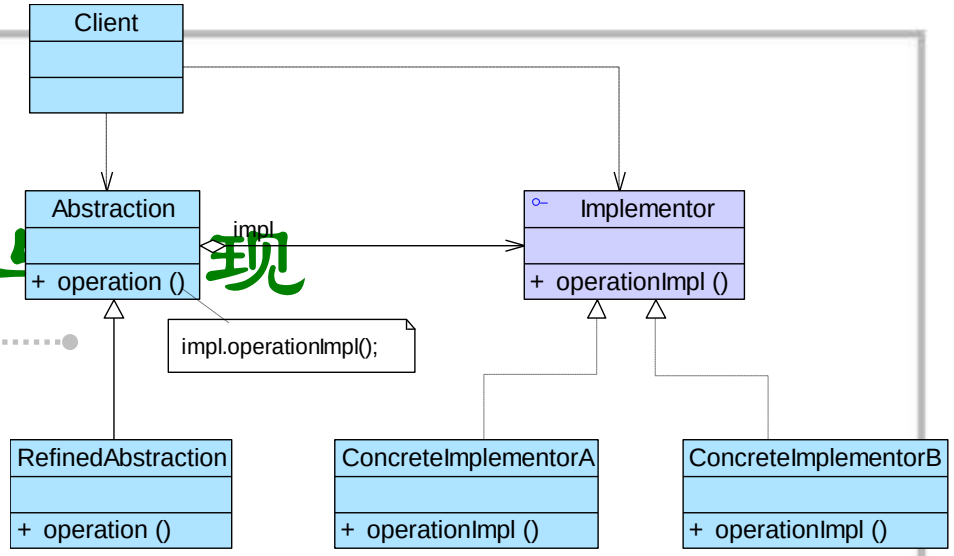
桥接模式的结构与实现



- 桥接模式的实现
  - ✓典型的实现器接口代码:

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

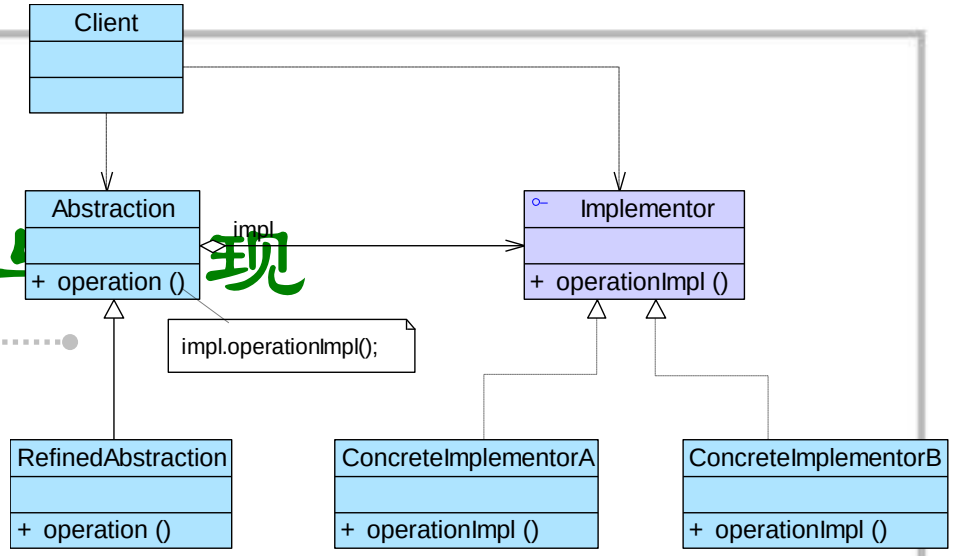
桥接模式的结构与实现



- 桥接模式的实现
  - ✓典型的具体实现器代码:

```
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



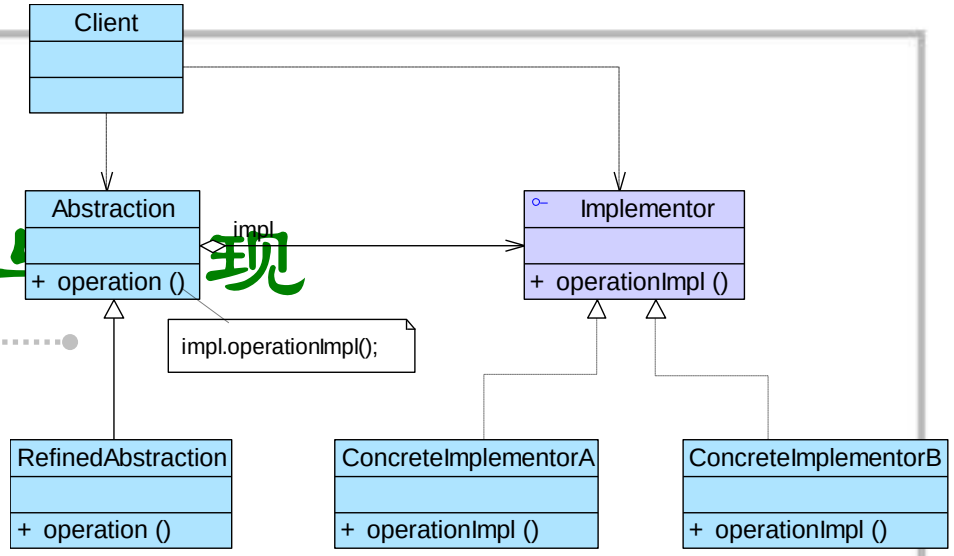
- 桥接模式的实现
  - ✓典型的抽象类代码:

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

    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(); //调用实现器的方法
        //业务代码
    }
}
```

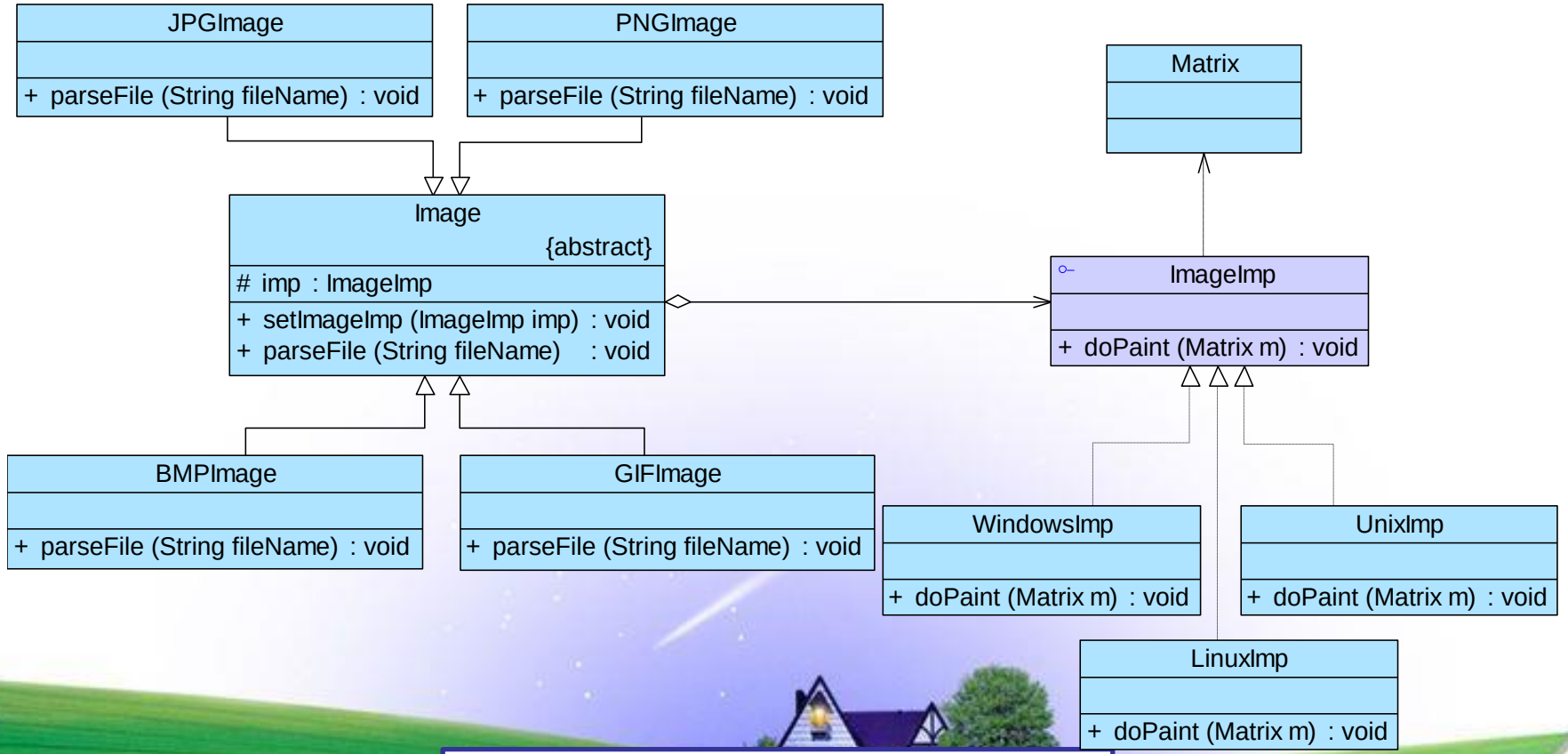
桥接模式的应用实例

- 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

演示.....

- 实例代码

Code (designpatterns.bridge)

- ✓ (1) Matrix：像素矩阵类，辅助类
- ✓ (2) ImageImp：抽象操作系统实现器，充当实现器接口
- ✓ (3) WindowsImp：Windows操作系统实现器，充当具体实现器
- ✓ (4) LinuxImp：Linux操作系统实现器，充当具体实现器
- ✓ (5) UnixImp：UNIX操作系统实现器，充当具体实现器
- ✓ (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

```
//像素矩阵类，辅助类
public class M
   //代码省略
}
```

```
//抽象操作系
public interfa
          pub
}
```

```
//Windows操作系统实现器，充当具体实现器
public class WindowsImp implements ImageImp {
  public void doPaint(Matrix m) {
            //调用Windows系统的绘制函数绘制像素矩阵
            System.out.print("在Windows操作系统中显示图像：");
  }
}

//Linux操作系统实现器，充当具体实现器
public class LinuxImp implements ImageImp {
  public void doPaint(Matrix m) {
            //调用Linux系统的绘制函数绘制像素矩阵
            System.out.print("在Linux操作系统中显示图像：");
  }
}

//Unix操作系统实现器，充当具体实现器
public class UnixImp implements ImageImp {
  public void doPaint(Matrix m) {
            //调用Unix系统的绘制函数绘制像素矩阵
            System.out.print("在Unix操作系统中显示图像：");
  }
}
```

```
//抽象图像类，充当抽象类
public abstract class Image {
    protected ImageImp imp;
    //注入实现类接口对象
    public void setImageImp(ImageImp imp) {
        this.imp = imp;
    }

    public abstract void parseFile(String fileName);
}
```

```
//JPG格式图像类，充当扩充抽象类
public class JPGImage extends Ima
        public void parseFile(Strin
    //模拟解析JPG文件并获得一个像
```

```
//PNG格式图像类，充当扩充抽象类
public class PNGImage extends Image {
        public void parseFile(String fileName) {
    //模拟解析PNG文件并获得一个像素矩阵对象m;
```

```
//GIF格式图像类，充当扩充抽象类
public class GIFImage extends Imag
        public void parseFile(String
    //模拟解析GIF文件并获得一个像
    Matrix m = new Matrix();
    imp.doPaint(m);
    System.out.println(fileName + "，
  }
}
```

```
//BMP格式图像类，充当扩充抽象类
public class BMPImage extends Image {
        public void parseFile(String fileName) {
    //模拟解析BMP文件并获得一个像素矩阵对象m;
    Matrix m = new Matrix();
    imp.doPaint(m);
    System.out.println(fileName + "，格式为BMP。");
  }
}
```

桥接模式的应用实例

```
//客户端测试类
public class Client {
        public static void main(String args[]) {
                Image image;
                ImageImp imp;
                image = new GIFImage();
                //image = (Image)XMLUtil.getBean("image");
                imp = new WindowsImp();
                //imp = (ImageImp)XMLUtil.getBean("os");
                image.setImageImp(imp);
                image.parseFile("小龙女");
        }
}
```

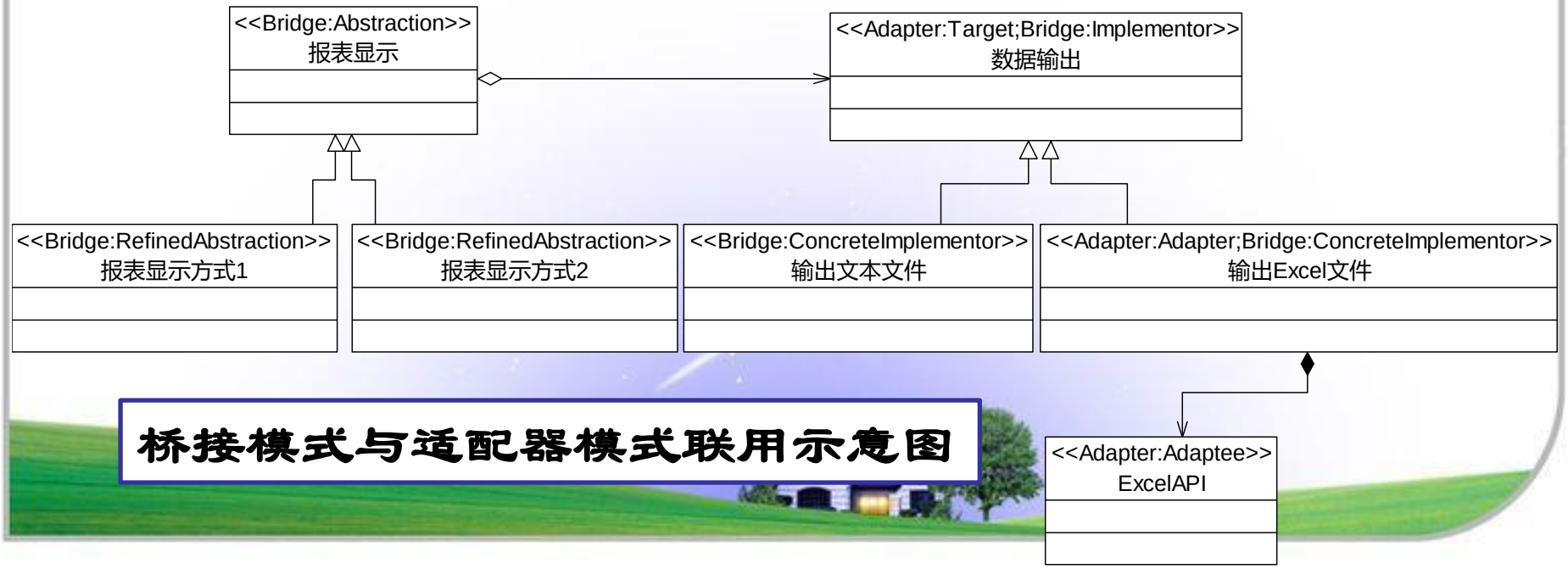
桥接模式的应用实例

- 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

END