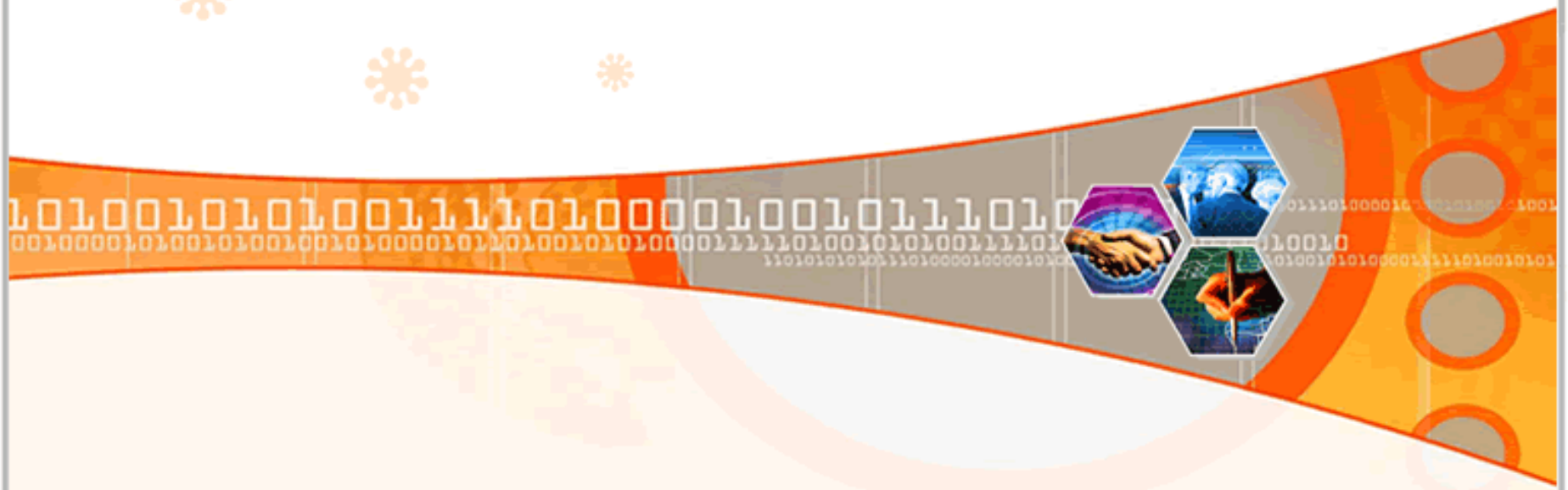
设计模式分类

设计模式分类

- ✓ 创建型模式：对类的实例化过程进行抽象，能够使软件模块与对象的创建和组织无关。包括工厂方法模式、抽象工厂模式、原型模式、单例模式、建造者模式等。
- ✓ 结构型模式：描述如何将类或对象结合在一起形成更大的结构。包括适配器模式、桥接模式、组合模式、装饰模式、代理模式等。
- ✓ 行为型模式：对不同对象之间的划分责任和算法的抽象化。包括职责链模式、命令模式、中介者模式、观察者模式、策略模式、访问者模式。

设计模式

单例模式



本章教学内容

- 单例模式
  - ✓ 模式动机与定义
  - ✓ 模式结构与分析
  - ✓ 模式实例与解析
  - ✓ 模式效果与应用
  - ✓ 模式扩展

单例模式

模式动机

- ✓ 对于系统中的某些类来说，只有一个实例很重要，例如，一个系统中可以存在多个打印任务，但是只能有一个正在工作的任务；一个系统只能有一个窗口管理器或文件系统；一个系统只能有一个计时工具或ID（序号）生成器。

单例模式

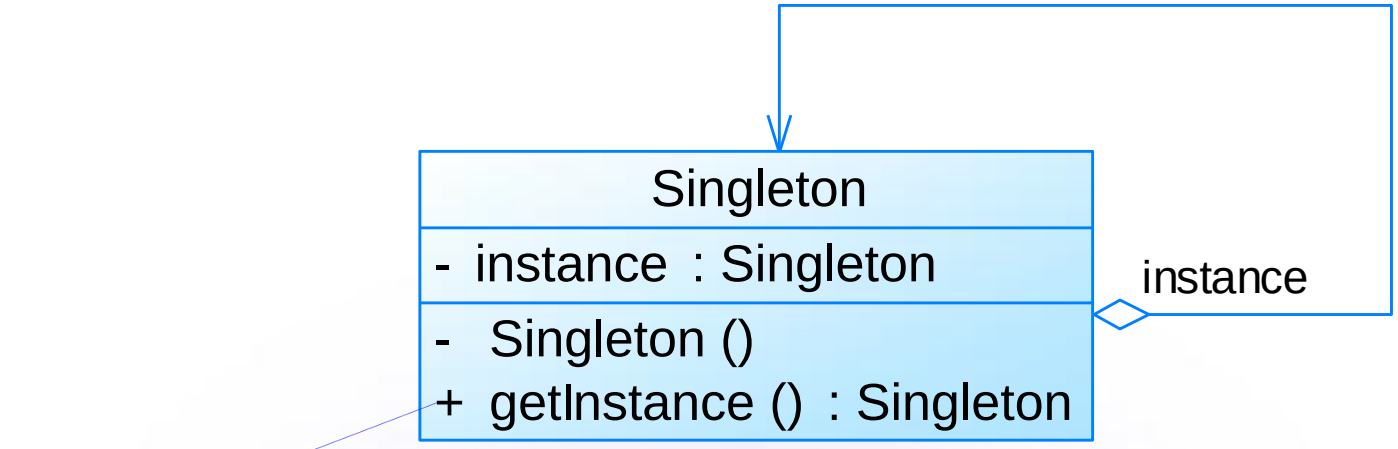
- 模式动机(问题)
  - ✓ 如何保证一个类只有一个实例并且这个实例易于被访问呢？定义一个全局变量可以确保对象随时都可以被访问，但不能防止我们实例化多个对象。
  - ✓ 一个更好的解决办法是让类自身负责保存它的唯一实例。这个类可以保证没有其他实例被创建，并且它可以提供一个访问该实例的方法。这就是单例模式的模式动机。

单例模式

- 模式定义
  - ✓ 单例模式(Singleton Pattern)：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。
  - ✓ 单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式。单例模式又名单件模式或单态模式。

单例模式

- 模式结构




```
if(instance==null)
        instance=new Singleton();
return instance;
```

单例模式

- 模式结构
  - ✓ 单例模式包含如下角色：
    - Singleton：单例

单例模式

- 模式分析
  - ✓ 单例模式的目的是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。单例模式包含的角色只有一个，就是单例类——**Singleton**。单例类拥有一个私有构造函数，确保用户无法通过**new**关键字直接实例化它。除此之外，该模式中包含一个静态私有成员变量与静态公有的工厂方法，该工厂方法负责检验实例的存在性并实例化自己，然后存储在静态成员变量中，以确保只有一个实例被创建。

单例模式

- 模式分析
  - ✓ 单例模式的实现代码如下所示：

```
public class Singleton
{
      private static Singleton instance=null;  //静态私有成员变量
      //私有构造函数
      private Singleton()
      {
      }

     //静态公有工厂方法，返回唯一实例
      public static Singleton getInstance()
      {
                if(instance==null)
                   instance=new Singleton();
                return instance;
      }
}
```

单例模式

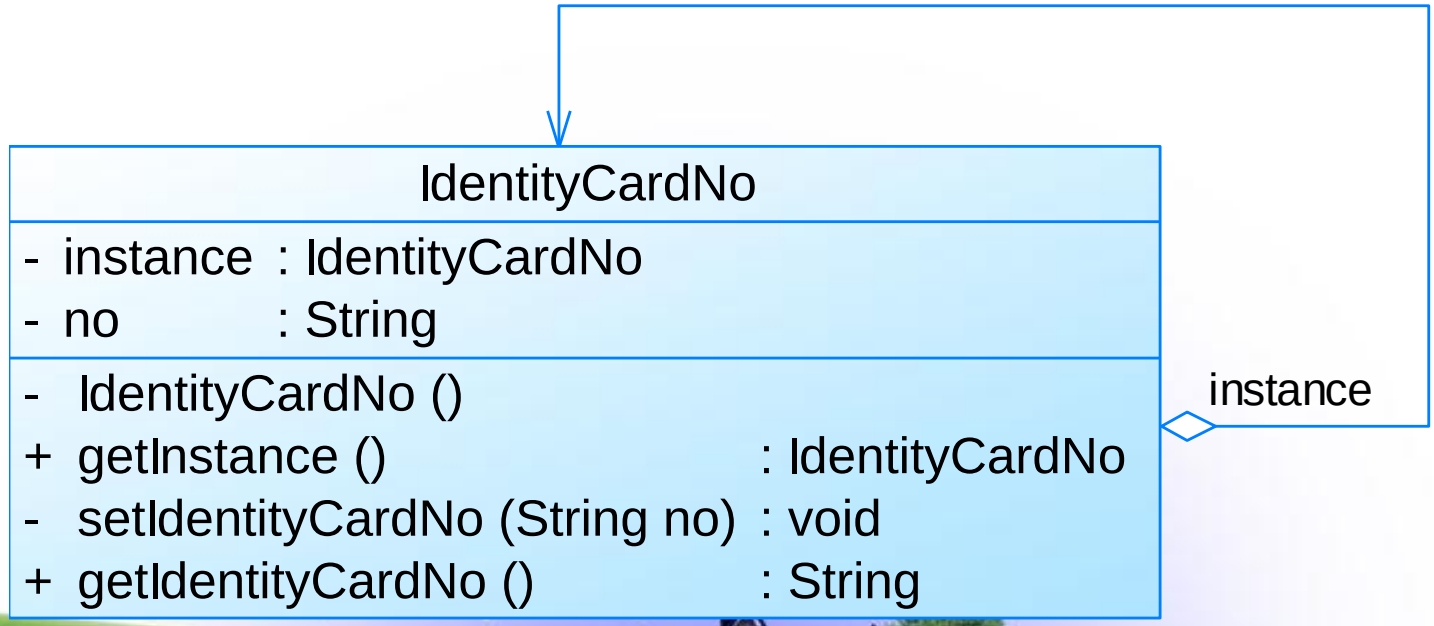
- 模式分析
  - ✓在单例模式的实现过程中，需要注意如下三点：
    - 单例类的构造函数为私有；
    - 提供一个自身的静态私有成员变量；
    - 提供一个公有的静态工厂方法。

单例模式

- 单例模式实例与解析
  - ✓实例一：身份证号码
    - 在现实生活中，居民身份证号码具有唯一性，同一个人不允许有多个身份证号码，第一次申请身份证时将给居民分配一个身份证号码，如果之后因为遗失等原因补办时，还是使用原来的身份证号码，不会产生新的号码。现使用单例模式模拟该场景。

单例模式

- 单例模式实例与解析
  - ✓实例一：身份证号码

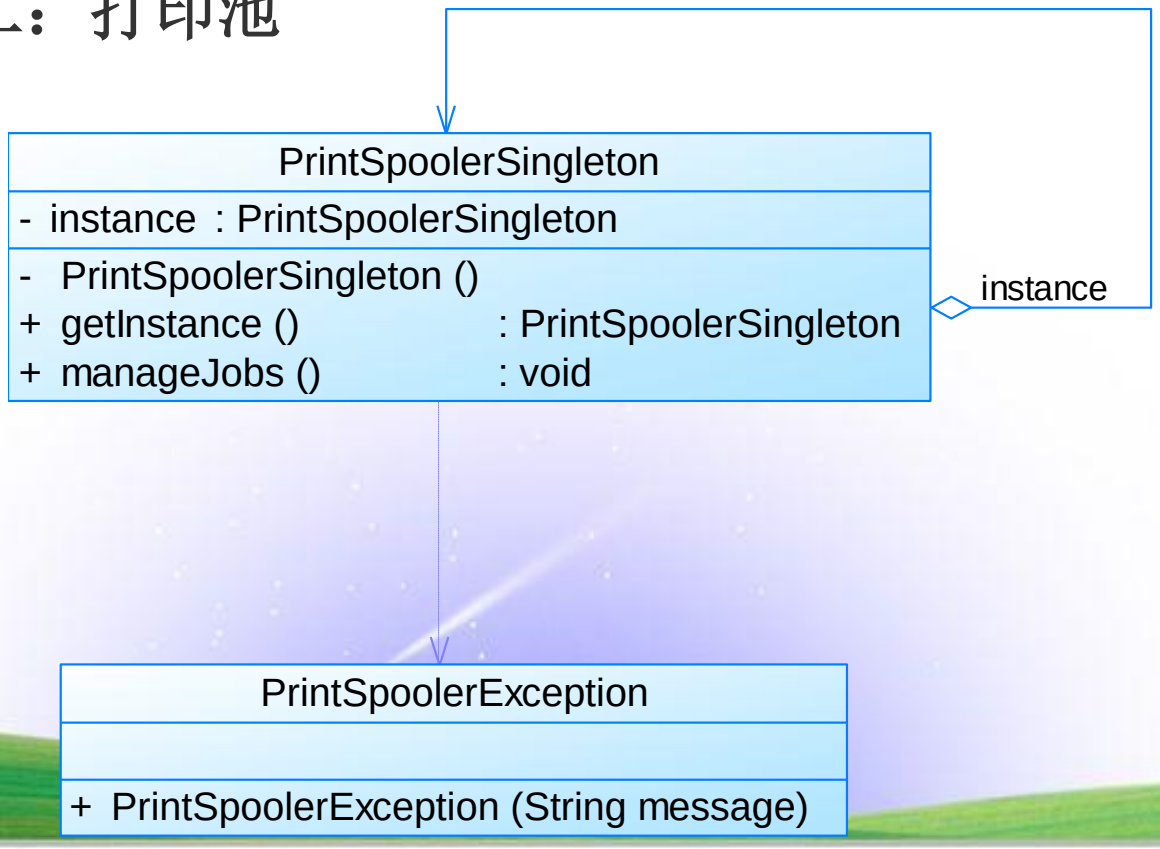


单例模式

- 单例模式实例与解析
  - ✓实例二：打印池
    - 在操作系统中，打印池(Print Spooler)是一个用于管理打印任务的应用程序，通过打印池用户可以删除、中止或者改变打印任务的优先级，在一个系统中只允许运行一个打印池对象，如果重复创建打印池则抛出异常。现使用单例模式来模拟实现打印池的设计。

单例模式

- 单例模式实例与解析
  - ✓实例二：打印池



单例模式

- 模式优缺点
  - ✓ 单例模式的优点
    - 提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它，并为设计及开发团队提供了共享的概念。
    - 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象，单例模式无疑可以提高系统的性能。
    - 允许可变数目的实例。我们可以基于单例模式进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例。

单例模式

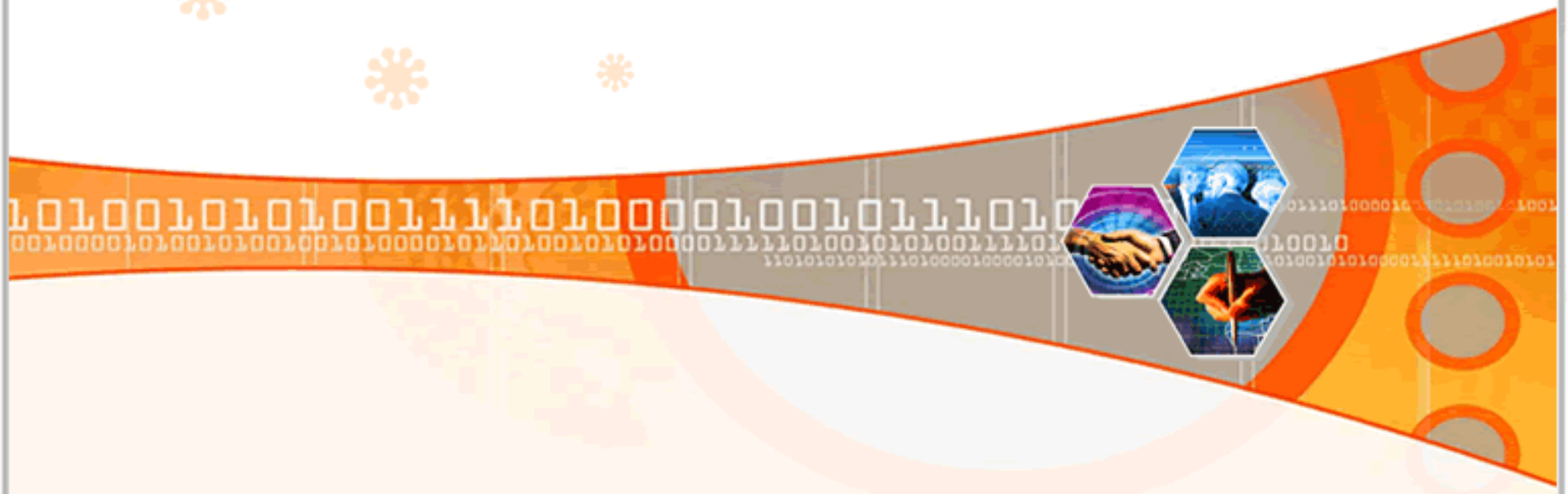
- 模式优缺点
  - ✓ 单例模式的缺点
    - 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。
    - 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品的本身的功能融合到一起。

单例模式

- 模式适用环境
  - ✓在以下情况下可以使用单例模式：
    - 系统只需要一个实例对象，如系统要求提供一个唯一的序列号生成器，或者需要考虑资源消耗太大而只允许创建一个对象。
    - 客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例。
    - 在一个系统中要求一个类只有一个实例时才应当使用单例模式。反过来，如果一个类可以有几个实例共存，就需要对单例模式进行改进，使之成为多例模式。

设计模式

桥接模式



本章教学内容

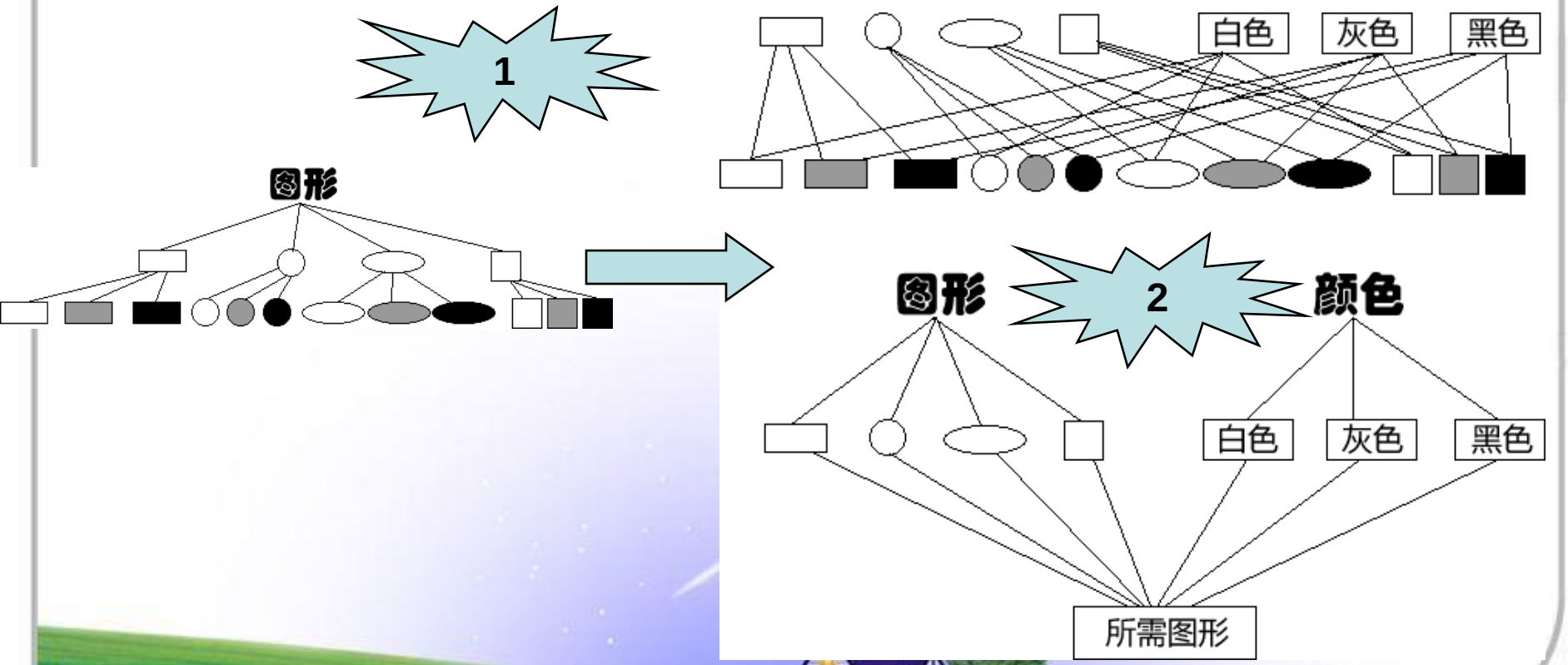
- 桥接模式
  - ✓ 模式动机与定义
  - ✓ 模式结构与分析
  - ✓ 模式实例与解析
  - ✓ 模式效果与应用
  - ✓ 模式扩展

桥接模式

- 模式动机(问题)
  - ✓ 设想如果要绘制矩形、圆形、椭圆、正方形，我们至少需要4个形状类，但是如果绘制的图形需要具有不同的颜色，如红色、绿色、蓝色等，此时至少有如下两种设计方案：
    - 第一种设计方案是为每一种形状都提供一套各种颜色的版本。
    - 第二种设计方案是根据实际需要对形状和颜色进行组合。

桥接模式

- 模式动机

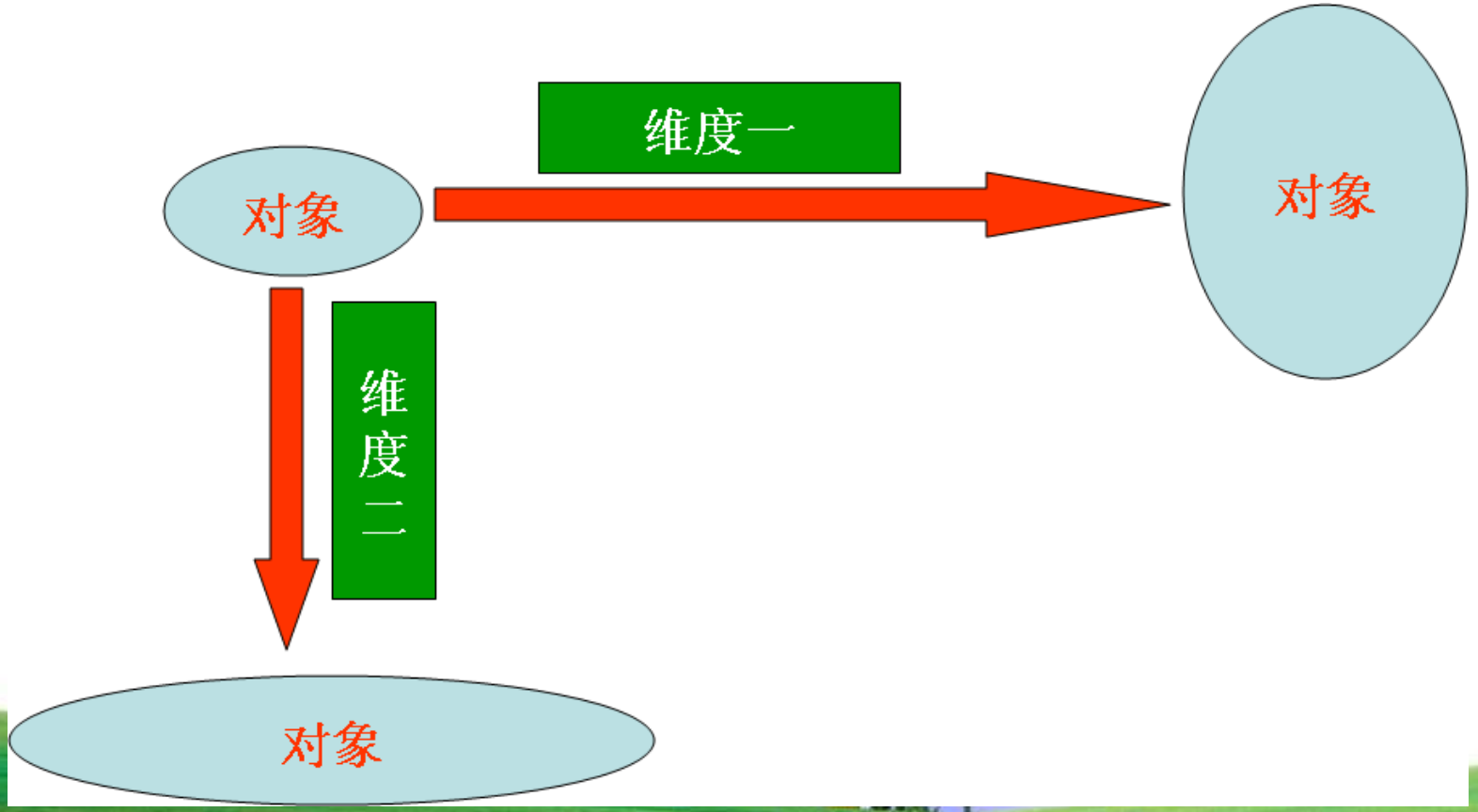


桥接模式

- 模式动机
  - ✓对于有两个变化维度（即两个变化的原因）的系统，采用方案二来进行设计系统中类的个数更少，且系统扩展更为方便。设计方案二即是桥接模式的应用。桥接模式将继承关系转换为关联关系，从而降低了类与类之间的耦合，减少了代码编写量。

桥接模式

- 模式动机

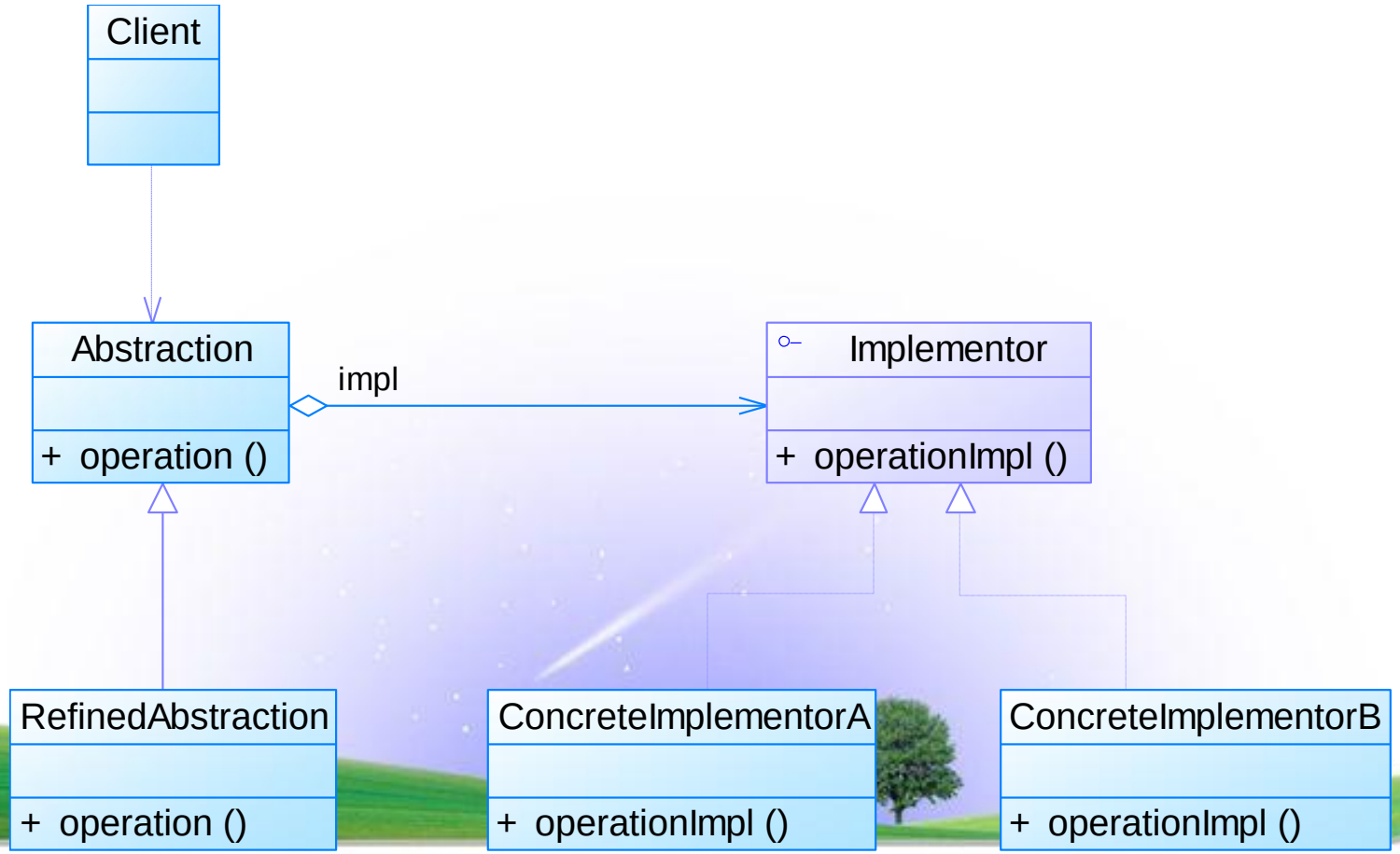


桥接模式

- 模式定义
  - ✓ 桥接模式(Bridge Pattern)：将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interface)模式。

桥接模式

- 模式结构


Client
Abstraction
+ operation ()
impl
Implementor
+ operationImpl ()
RefinedAbstraction
+ operation ()
ConcreteImplementorA
+ operationImpl ()
ConcreteImplementorB
+ operationImpl ()

桥接模式

- 模式结构
  - ✓桥接模式包含如下角色：
    - 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

- 模式分析
  - ✓ 理解桥接模式，重点需要理解如何将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化。
    - 抽象化：抽象化就是忽略一些信息，把不同的实体当作同样的实体对待。在面向对象中，将对象的共同性质抽取出来形成类的过程即为抽象化的过程。
    - 实现化：针对抽象化给出的具体实现，就是实现化，抽象化与实现化是一对互逆的概念，实现化产生的对象比抽象化更具体，是对抽象化事物的进一步具体化的产物。
    - 脱耦：脱耦就是将抽象化和实现化之间的耦合解脱开，或者说是将它们之间的强关联改换成弱关联，将两个角色之间的继承关系改为关联关系。桥接模式中的所谓脱耦，就是指在一个软件系统的抽象化和实现化之间使用关联关系（组合或者聚合关系）而不是继承关系，从而使两者可以相对独立地变化，这就是桥接模式的用意。

桥接模式

- 模式分析
  - ✓ 典型的实现类接口代码：

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

桥接模式

- 模式分析
  - ✓ 典型的抽象类代码：

```
public abstract class Abstraction
{
    protected Implementor impl;

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

    public abstract void operation();
}
```

桥接模式

- 模式分析
  - ✓ 典型的扩充抽象类代码：

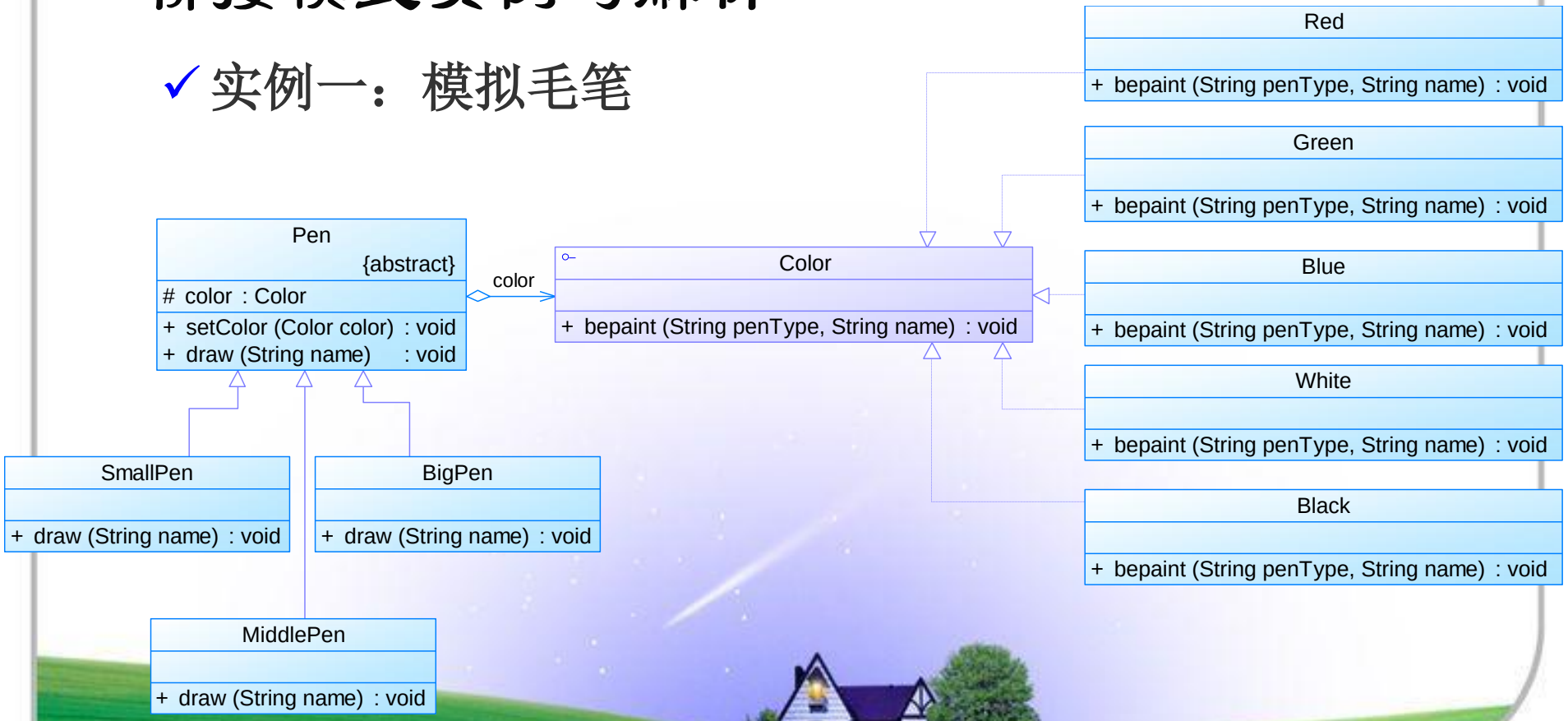
```
public class RefinedAbstraction extends Abstraction
{
    public void operation()
    {
        //代码
        impl.operationImpl();
        //代码
    }
}
```

桥接模式

- 桥接模式实例与解析
  - ✓实例一：模拟毛笔
    - 现需要提供大中小3种型号的画笔，能够绘制5种不同颜色，如果使用蜡笔，我们需要准备3*5=15支蜡笔，也就是说必须准备15个具体的蜡笔类。而如果使用毛笔的话，只需要3种型号的毛笔，外加5个颜料盒，用3+5=8个类就可以实现15支蜡笔的功能。本实例使用桥接模式来模拟毛笔的使用过程。

桥接模式

- 桥接模式实例与解析
  - ✓实例一：模拟毛笔

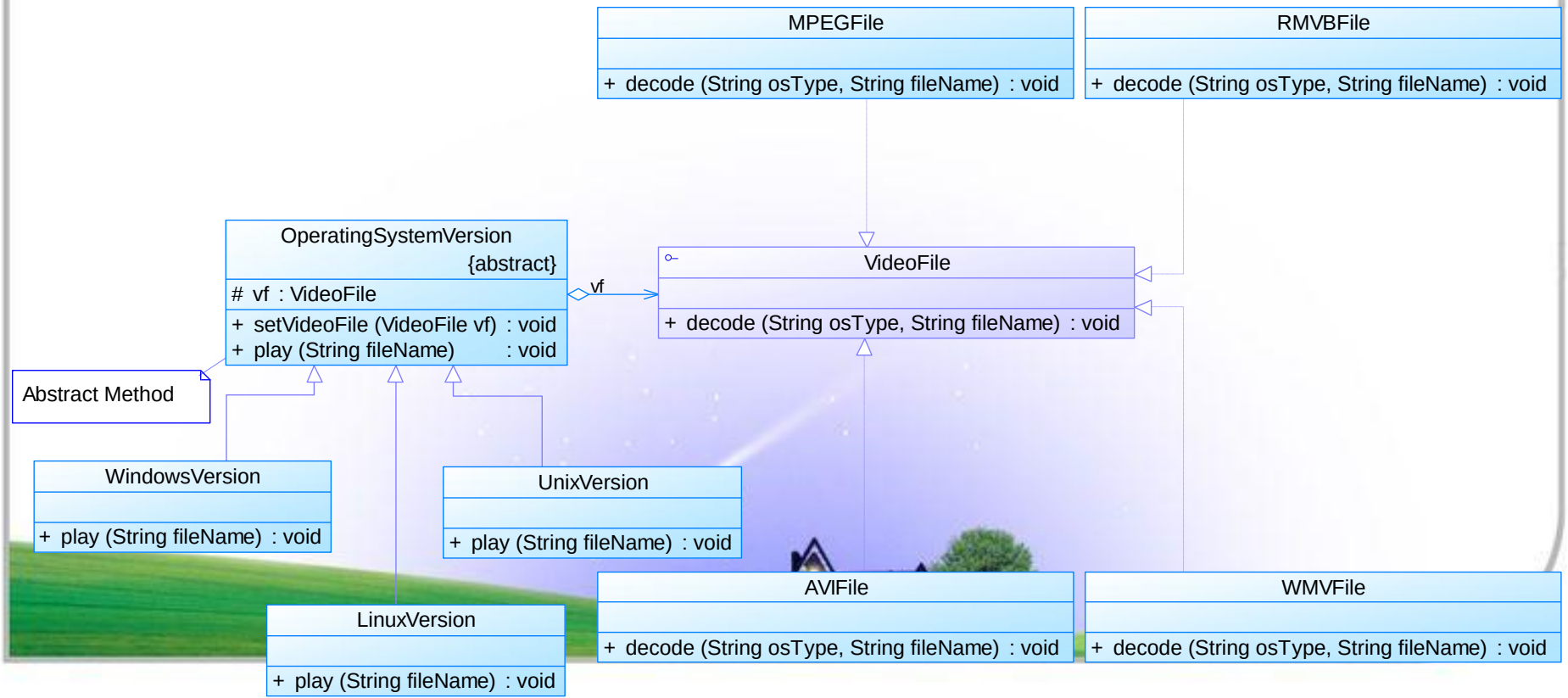


桥接模式

- 桥接模式实例与解析
  - ✓实例二：跨平台视频播放器
    - 如果需要开发一个跨平台视频播放器，可以在不同操作系统平台（如Windows、Linux、Unix等）上播放多种格式的视频文件，常见的视频格式包括MPEG、RMVB、AVI、WMV等。现使用桥接模式设计该播放器。

桥接模式

- 桥接模式实例与解析
  - ✓实例二：跨平台视频播放器



桥接模式

模式优缺点

✓ 桥接模式的优点

- 分离抽象接口及其实现部分。
- 桥接模式有时类似于多继承方案，但是多继承方案违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差，而且多继承结构中类的个数非常庞大，桥接模式是比多继承方案更好的解决方法。
- 桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统。
- 实现细节对客户透明，可以对用户隐藏实现细节。

桥接模式

- 模式优缺点
  - ✓ 桥接模式的缺点
    - 桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。
    - 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性。

桥接模式

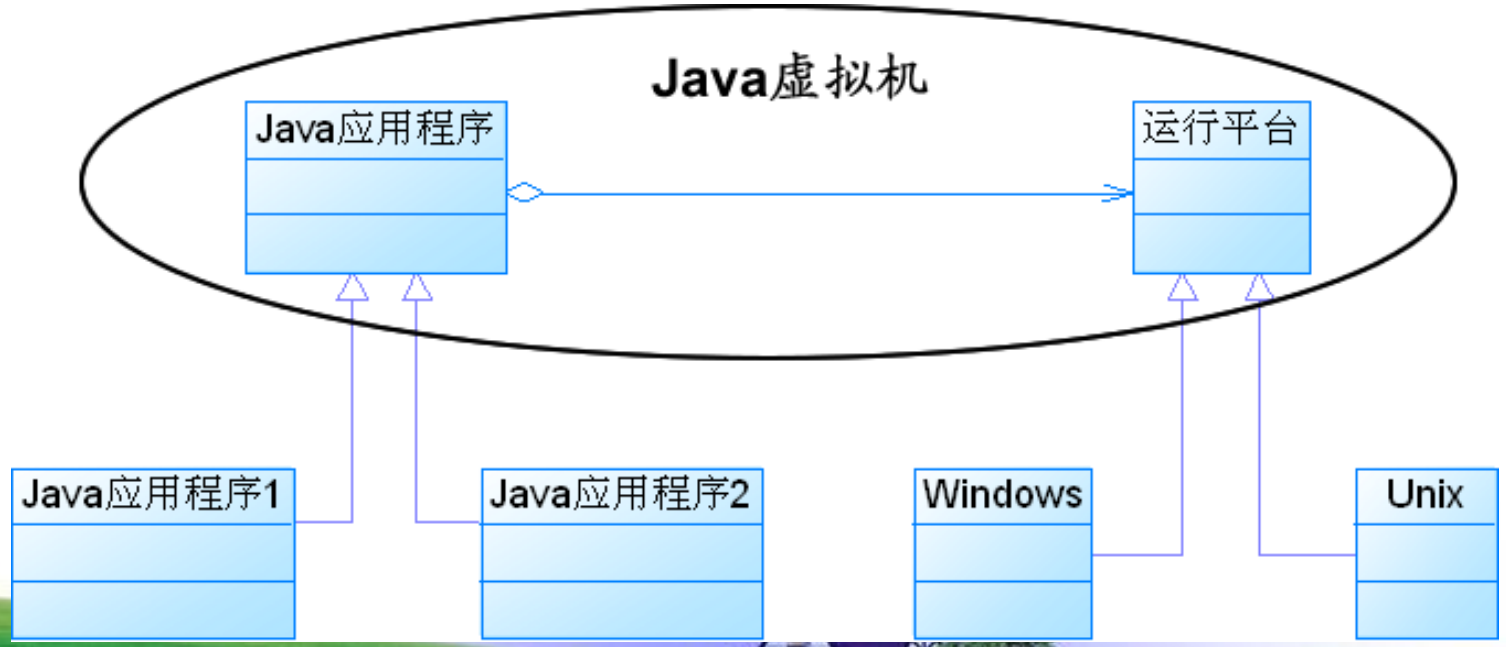
模式适用环境

✓ 在以下情况下可以使用桥接模式：

- 如果一个系统需要在构件的抽象化角色和具体化角色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联系，通过桥接模式可以使它们在抽象层建立一个关联关系。
- 抽象化角色和实现化角色可以以继承的方式独立扩展而互不影响，在程序运行时可以动态将一个抽象化子类的对象和一个实现化子类的对象进行组合，即系统需要对抽象化角色和实现化角色进行动态耦合。
- 一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。
- 虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。
- 对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。

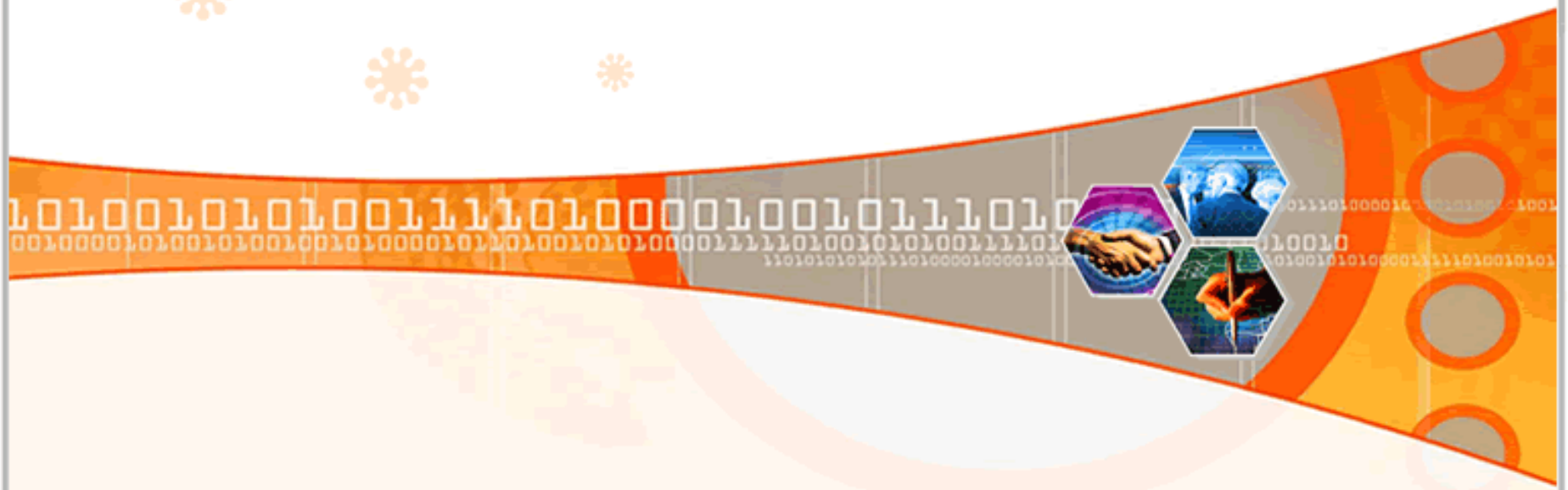
桥接模式

- 模式应用
  - ✓(1) Java语言通过Java虚拟机实现了平台的无关性。



设计模式

适配器模式



适配器模式

- 模式动机

适配器模式

模式动机

- 在软件开发中采用类似于电源适配器的设计和编码技巧被称为适配器模式。
- 通常情况下，客户端可以通过目标类的接口访问它所提供的服务。有时，现有的类可以满足客户类的功能需要，但是它所提供的接口不一定是客户类所期望的，这可能是因为现有类中方法名与目标类中定义的方法名不一致等原因所导致的。
- 在这种情况下，现有的接口需要转化为客户类期望的接口，这样保证了对现有类的重用。如果不进行这样的转化，客户类就不能利用现有类所提供的功能，适配器模式可以完成这样的转化。

适配器模式

模式动机

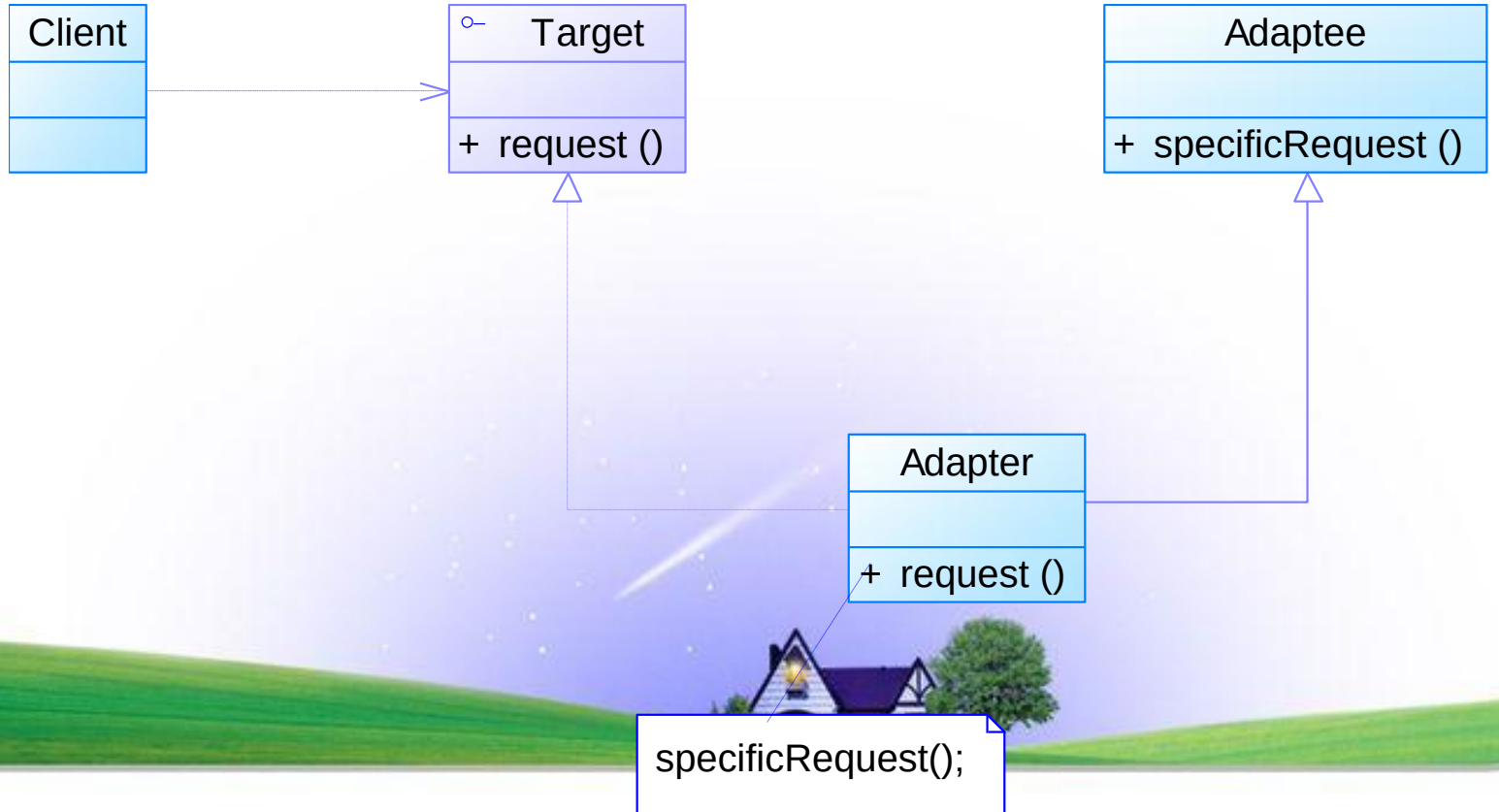
- 在适配器模式中可以定义一个包装类，包装不兼容接口的对象，这个包装类指的就是**适配器(Adapter)**，它所包装的对象就是**适配者(Adaptee)**，即被适配的类。
- 适配器提供客户类需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。也就是说：当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户类是透明的，客户类并不直接访问适配者类。因此，适配器可以使由于接口不兼容而不能交互的类可以一起工作。这就是适配器模式的模式动机。

适配器模式

- 模式定义
  - ✓ 适配器模式(**Adapter Pattern**)：将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(**Wrapper**)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。

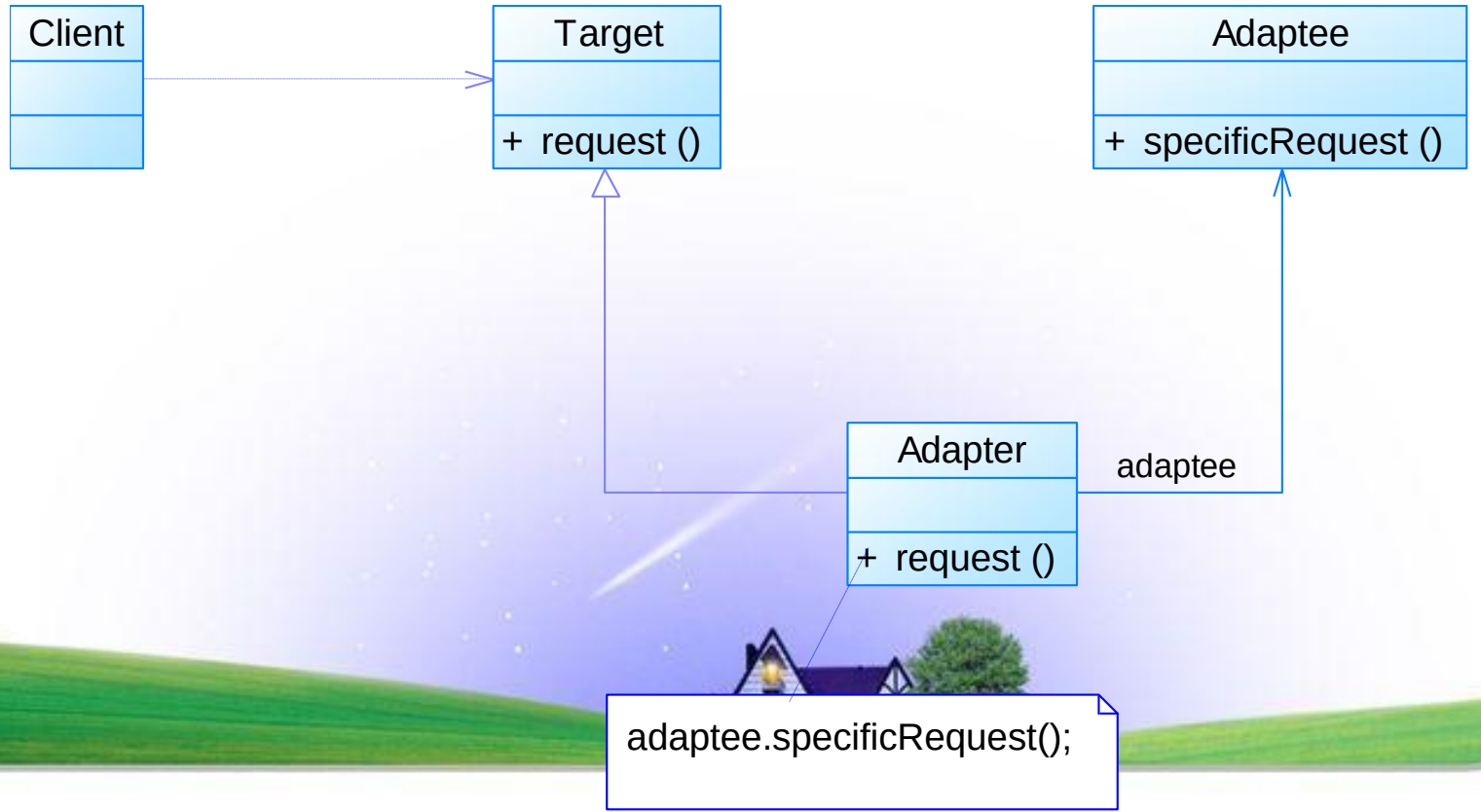
适配器模式

- 模式结构
  - ✓类适配器



适配器模式

- 模式结构
  - ✓对象适配器



适配器模式

- 模式结构
  - ✓适配器模式包含如下角色：
    - Target：目标抽象类
    - Adapter：适配器类
    - Adaptee：适配者类
    - Client：客户类

适配器模式

- 模式分析
  - ✓ 典型的类适配器代码：

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

适配器模式

- 模式分析
  - ✓典型的对象适配器代码：

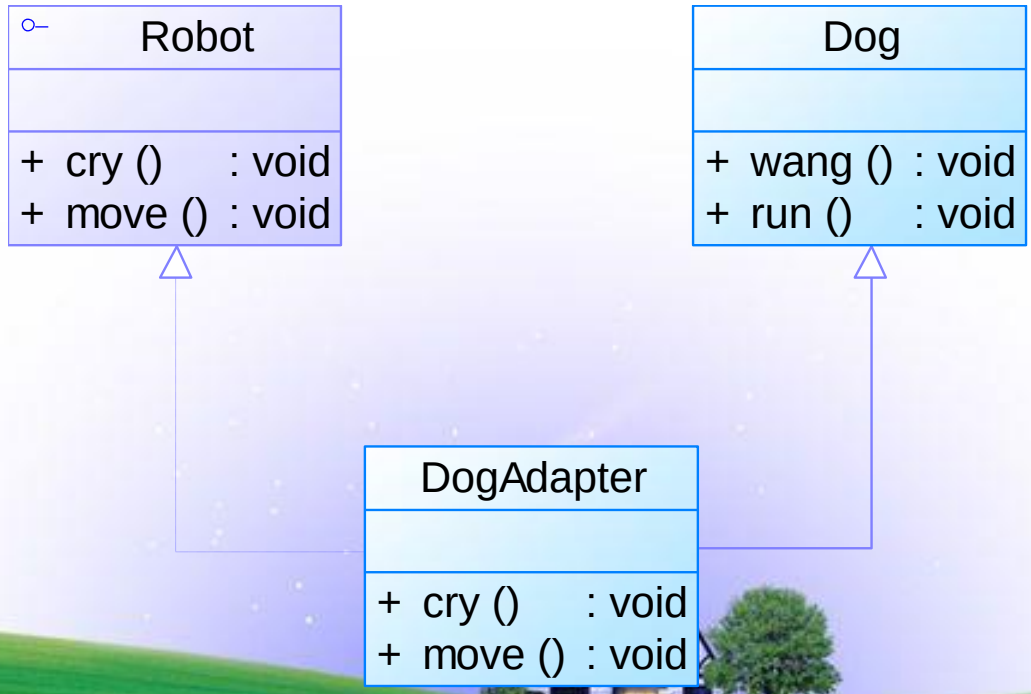
```
public class Adapter extends Target
{
    private Adaptee adaptee;

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

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

适配器模式

- 适配器模式实例与解析
  - ✓实例一：仿生机器人
    - 现需要设计一个可以模拟各种动物行为的机器人，在机器人中定义了一系列方法，如机器人叫喊方法cry()、机器人移动方法move()等。如果希望在不修改已有代码的基础上使得机器人能够像狗一样叫，像狗一样跑，使用适配器模式进行系统设计。

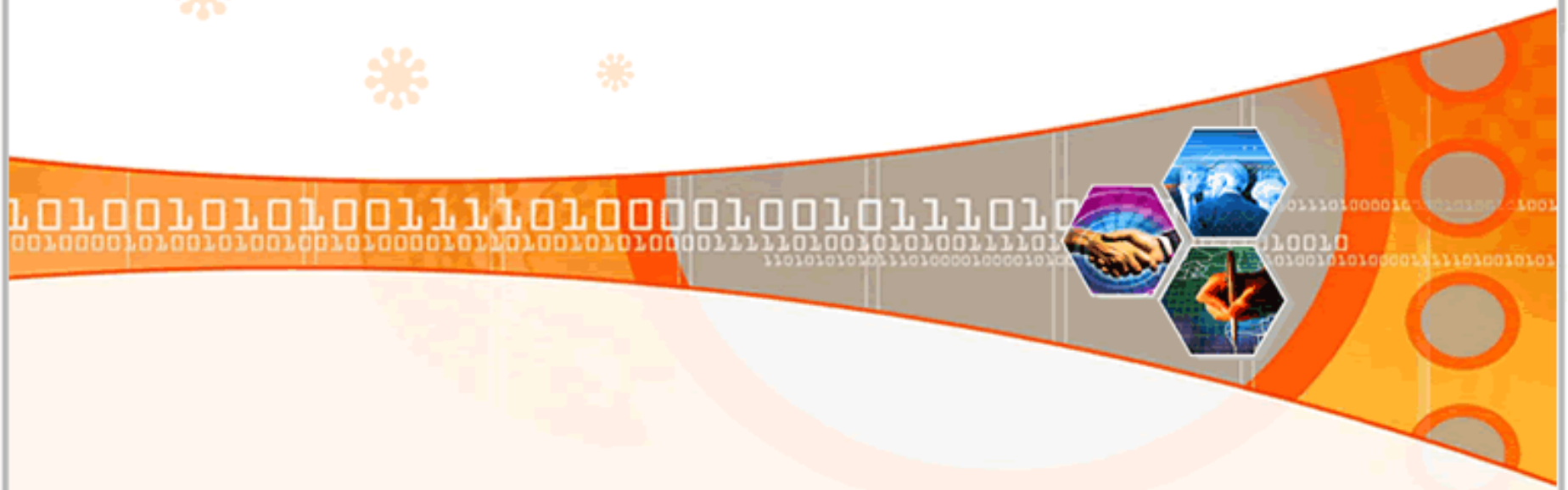
适配器模式

- 适配器模式实例与解析
  - ✓实例一：仿生机器人



设计模式

职责链模式

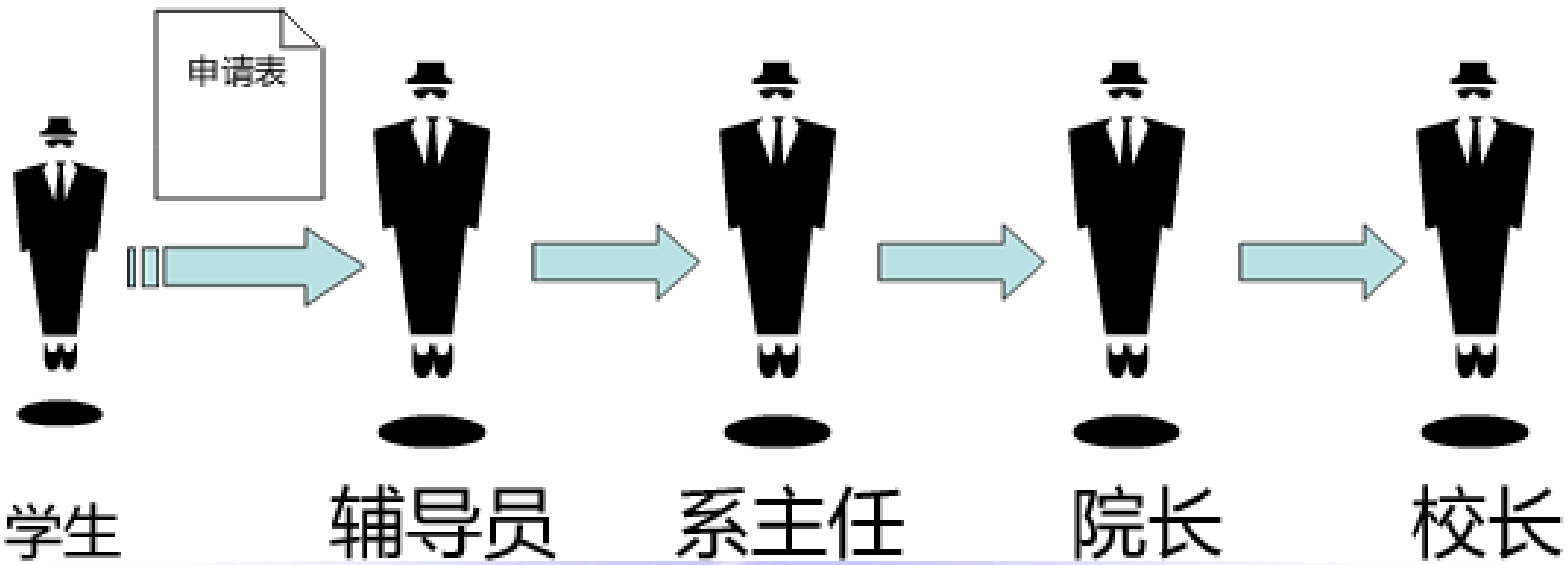


本章教学内容

- 行为型模式
  - ✓ 行为型模式概述
  - ✓ 行为型模式简介
- 职责链模式
  - ✓ 模式动机与定义
  - ✓ 模式结构与分析
  - ✓ 模式实例与解析
  - ✓ 模式效果与应用
  - ✓ 模式扩展

职责链模式

- 模式动机



职责链模式

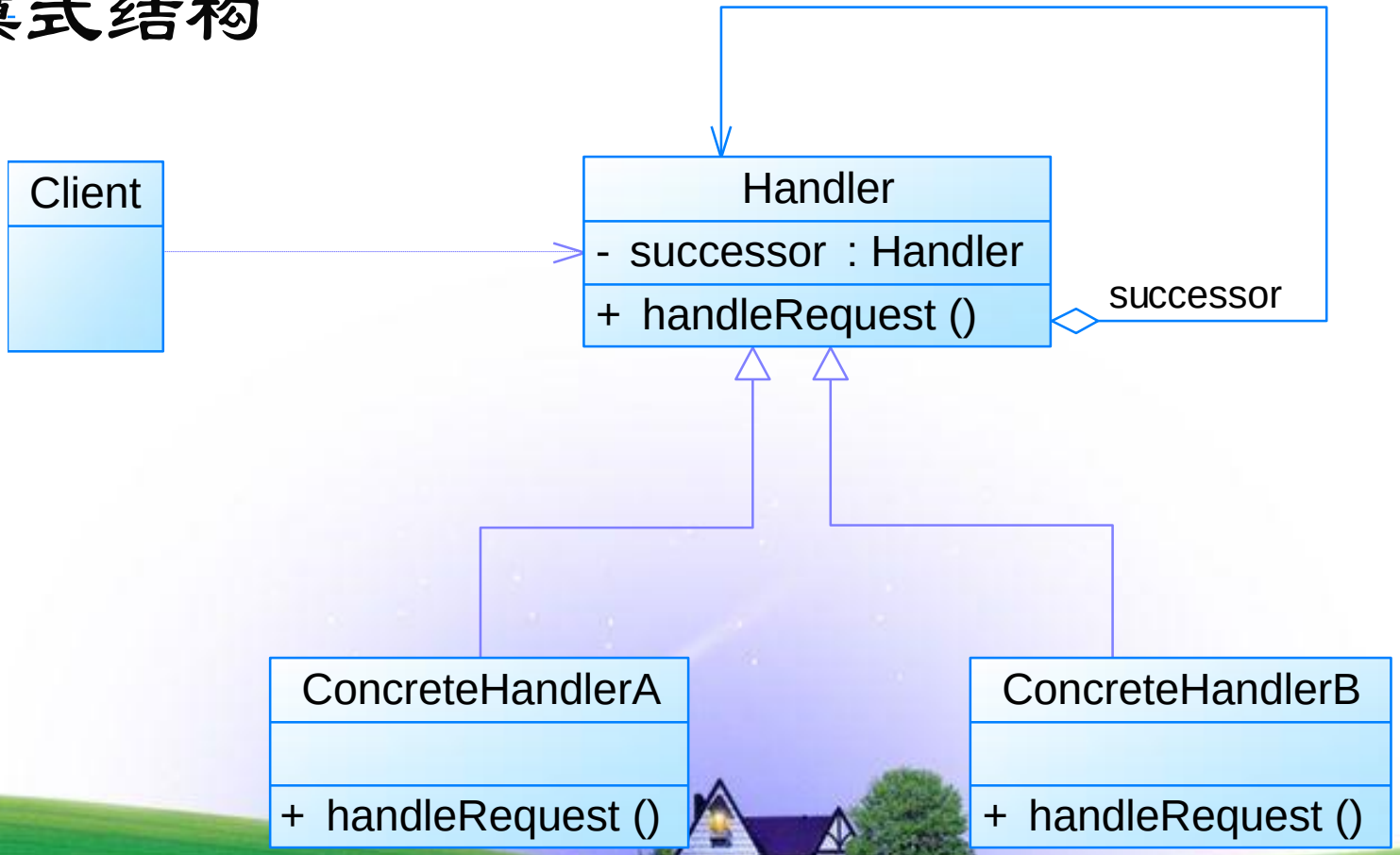
- 模式动机
  - ✓ 职责链可以是一条直线、一个环或者一个树形结构，最常见的职责链是直线型，即沿着一条单向的链来传递请求。
  - ✓ 链上的每一个对象都是请求处理者，职责链模式可以将请求的处理者组织成一条链，并使请求沿着链传递，由链上的处理者对请求进行相应的处理，客户端无须关心请求的处理细节以及请求的传递，只需将请求发送到链上即可，将请求的发送者和请求的处理者解耦。这就是职责链模式的模式动机。

职责链模式

- 模式定义
  - ✓ 职责链模式(Chain of Responsibility Pattern)：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。

职责链模式

- 模式结构


Client
Handler
- successor : Handler
+ handleRequest ()
successor
ConcreteHandlerA
+ handleRequest ()
ConcreteHandlerB
+ handleRequest ()

职责链模式

- 模式结构
  - ✓职责链模式包含如下角色：
    - Handler: 抽象处理者
    - ConcreteHandler: 具体处理者
    - Client: 客户类

职责链模式

- 模式分析
  - ✓ 在职责链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。
  - ✓ 请求在这条链上传递，直到链上的某一个对象处理此请求为止。
  - ✓ 发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。

职责链模式

- 模式分析
  - ✓ 典型的抽象处理者代码：

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

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

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

职责链模式

模式分析

- 典型的具体处理者代码：

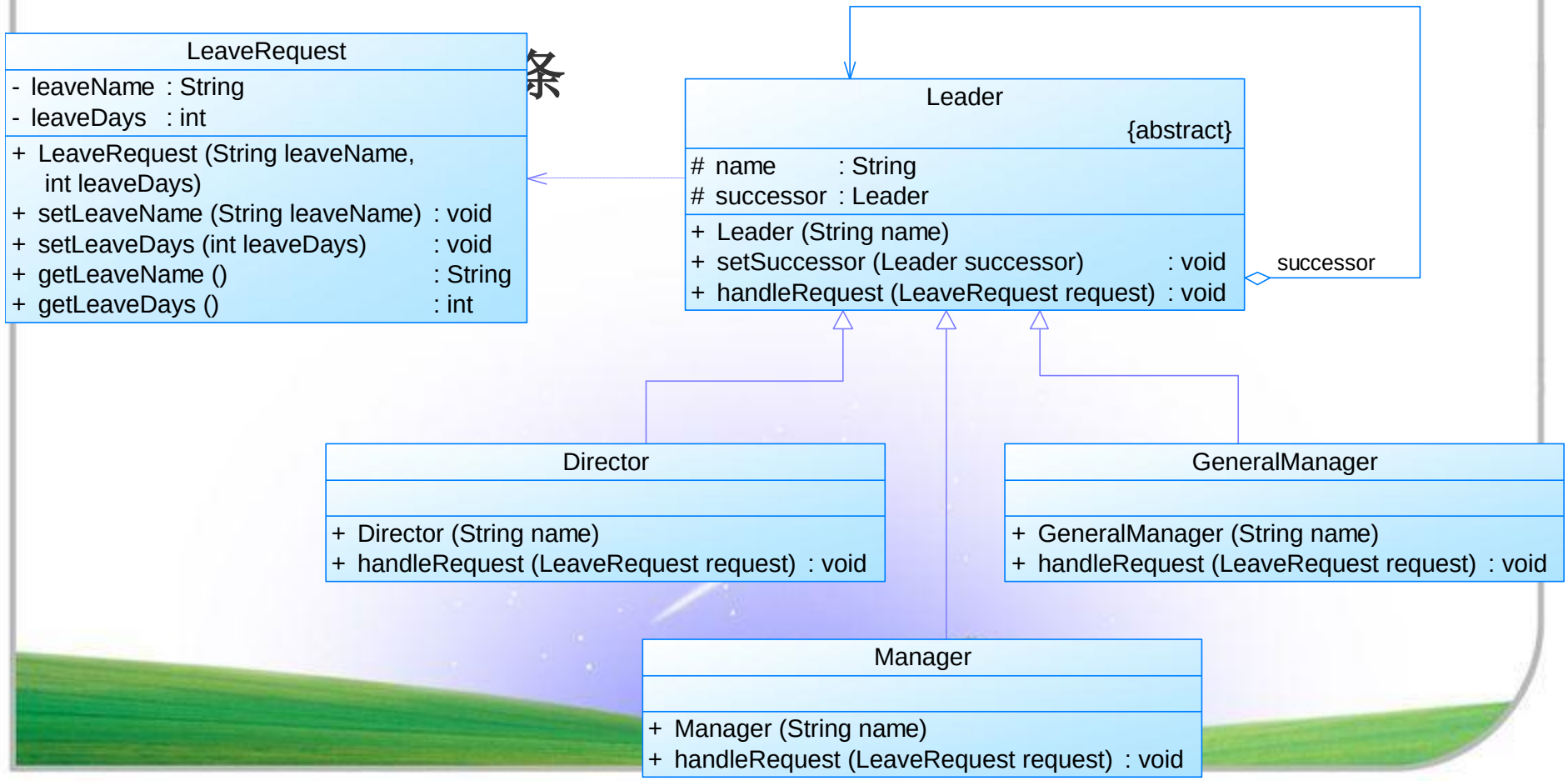
```
public class ConcreteHandler extends Handler
{
    public void handleRequest(String request)
    {
        if(请求request满足条件)
        {
            ..... //处理请求;
        }
        else
        {
            this.successor.handleRequest(request); //转发请求
        }
    }
}
```

职责链模式

- 职责链模式实例与解析
  - ✓实例：审批假条
    - 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，经理可以审批；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。

职责链模式

- 职责链模式实例与解析



职责链模式

- 模式优缺点
  - ✓职责链模式的优点
    - 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

职责链模式

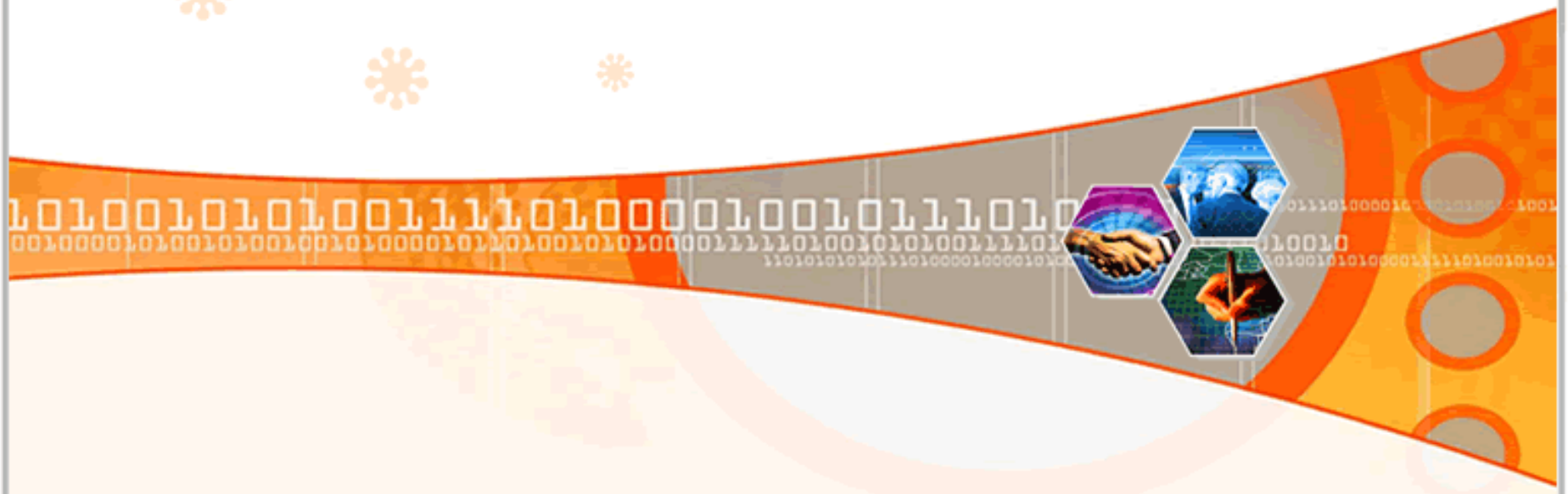
- 模式优缺点
  - ✓职责链模式的缺点
    - 不能保证请求一定被接收。
    - 系统性能将受到一定影响，而且在进行代码调试时不太方便；可能会造成循环调用。

职责链模式

- 模式适用环境
  - ✓在以下情况下可以使用职责链模式：
    - 有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。
    - 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
    - 可动态指定一组对象处理请求。

设计模式

中介者模式

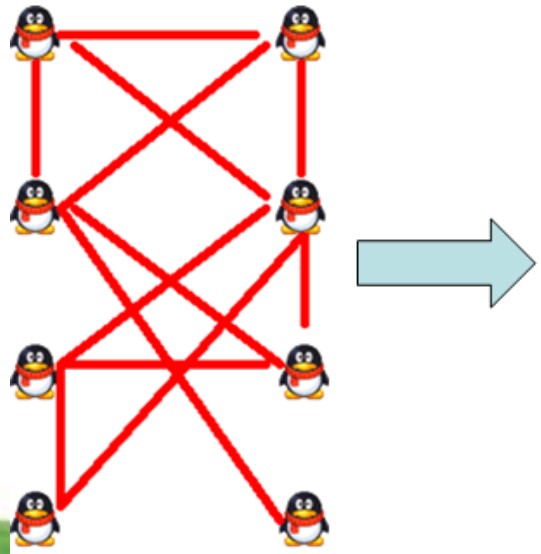


本章教学内容

- 中介者模式
  - ✓ 模式动机与定义
  - ✓ 模式结构与分析
  - ✓ 模式实例与解析
  - ✓ 模式效果与应用
  - ✓ 模式扩展

中介者模式

- 模式动机




友情万岁(12345678)
(普通群) 生活休闲
交谈中请勿轻信汇款、中奖信息、陌生电话，勿使用外挂软件。
群动态
你的群相册最近没有新照片
立刻上传
群成员(1/7)
搜索群成员
赵六
李四
王五
小强
小莉
小花
张三
消息记录
猛点下载，高手实盘数据抢先看
关闭(C)
发送(S)

中介者模式

- **模式动机**
  - ✓ 在用户与用户直接聊天的设计方案中，用户对象之间存在很强的关联性，将导致系统出现如下问题：
    - 系统结构复杂：对象之间存在大量的相互关联和调用，若有一个对象发生变化，则需要跟踪和该对象关联的其他所有对象，并进行适当处理。
    - 对象可重用性差：由于一个对象和其他对象具有很强的关联，若没有其他对象的支持，一个对象很难被另一个系统或模块重用，这些对象表现出来更像一个不可分割的整体，职责较为混乱。
    - 系统扩展性低：增加一个新的对象需要在原有相关对象上增加引用，增加新的引用关系也需要调整原有对象，系统耦合度很高，对象操作很不灵活，扩展性差。

中介者模式

模式动机（问题）

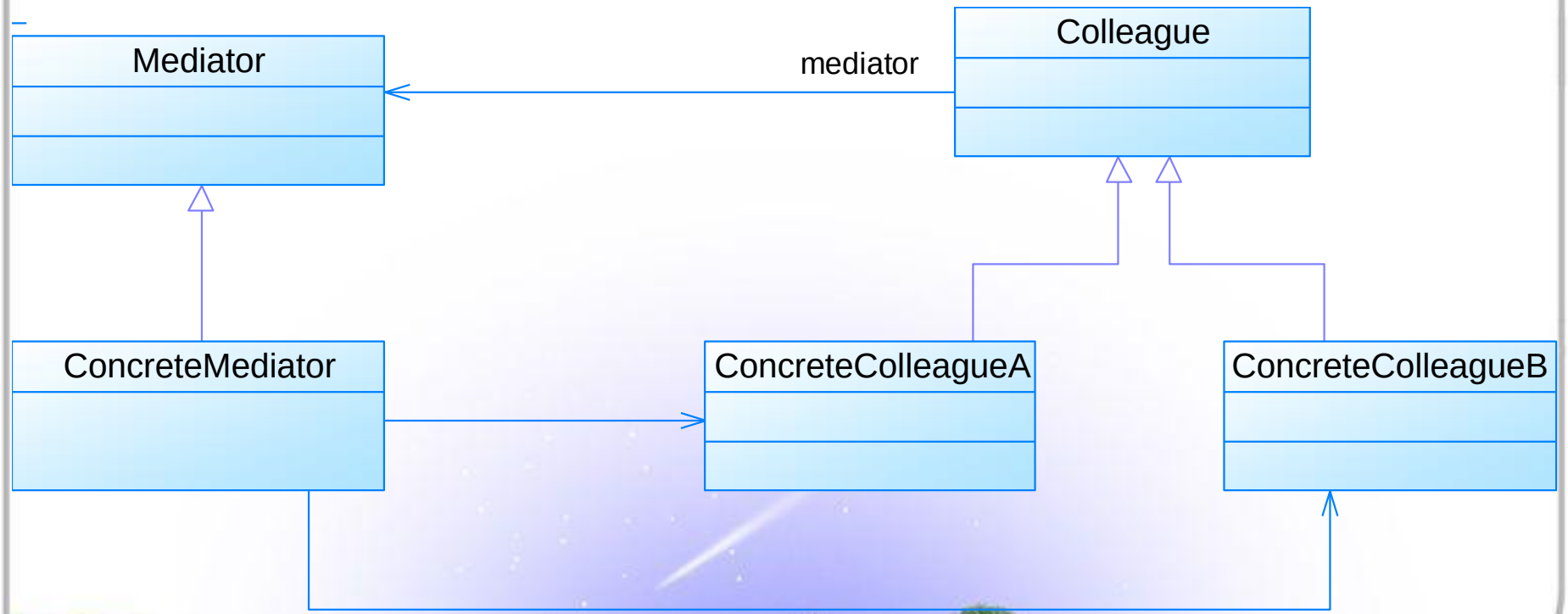
- 在面向对象的软件设计与开发过程中，根据“单一职责原则”，我们应该尽量将对象细化，使其只负责或呈现单一的职责。
- 对于一个模块，可能由很多对象构成，而且这些对象之间可能存在相互的引用，为了减少对象两两之间复杂的引用关系，使之成为一个松耦合的系统，我们需要使用中介者模式，这就是中介者模式的模式动机。

中介者模式

- 模式定义
  - ✓ 中介者模式(**Mediator Pattern**)定义：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式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式。

中介者模式

- 模式结构

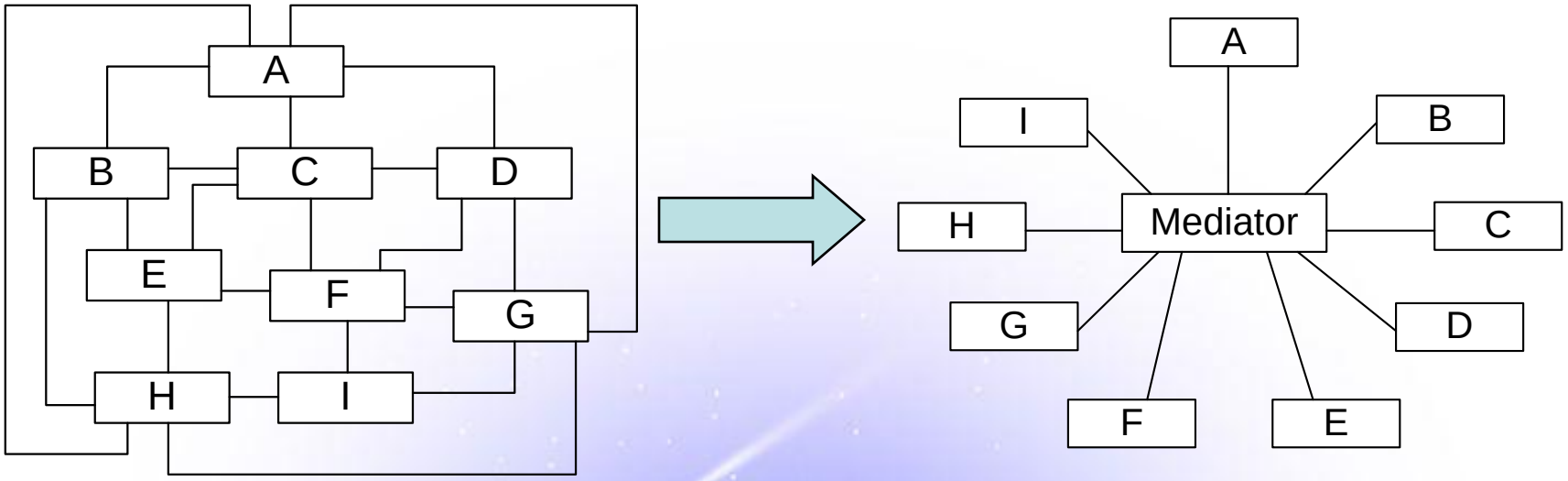

Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleagueA
ConcreteColleagueB

中介者模式

- 模式结构
  - ✓ 中介者模式包含如下角色：
    - Mediator: 抽象中介者
    - ConcreteMediator: 具体中介者
    - Colleague: 抽象同事类
    - ConcreteColleague: 具体同事类

中介者模式

- 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：



中介者模式

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。
  - 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。

中介者模式

- 模式分析
  - ✓ 典型的抽象中介者类代码：

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

        public abstract void operation();
}
```

中介者模式

- 模式分析
  - ✓ 典型的具体中介者类代码：

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

中介者模式

- 模式分析
  - ✓ 典型的抽象同事类代码：

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

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

        public abstract void method1();

        public abstract void method2();
}
```

中介者模式

- 模式分析
  - ✓ 典型的具体同事类代码：

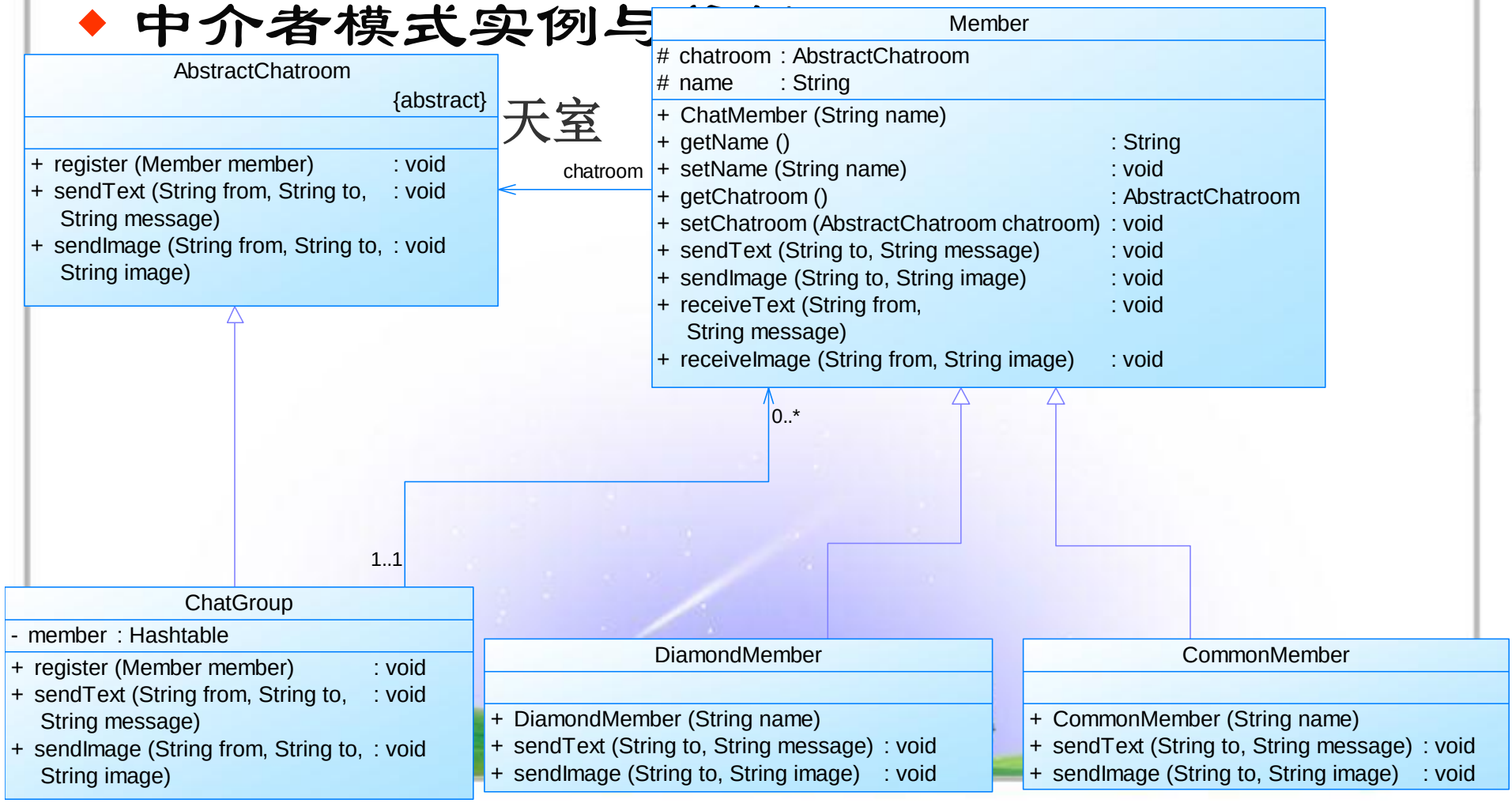
```
public class ConcreteColleague extends Colleague
{
        public ConcreteColleague(Mediator mediator)
        {
                super(mediator);
        }

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

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

中介者模式

- 中介者模式实例与解析
  - ✓实例：虚拟聊天室
    - 某论坛系统欲增加一个虚拟聊天室，允许论坛会员通过该聊天室进行信息交流，普通会员(CommonMember)可以给其他会员发送文本信息，钻石会员(DiamondMember)既可以给其他会员发送文本信息，还可以发送图片信息。该聊天室可以对不雅字符进行过滤，如“打”等字符；还可以对发送的图片大小进行控制。用中介者模式设计该虚拟聊天室。

中介者模式

- 中介者模式实例与



中介者模式

- 模式优缺点
  - ✓中介者模式的优点
    - 简化了对象之间的交互。
    - 将各同事解耦。
    - 减少子类生成。
    - 可以简化各同事类的设计和实现。

中介者模式

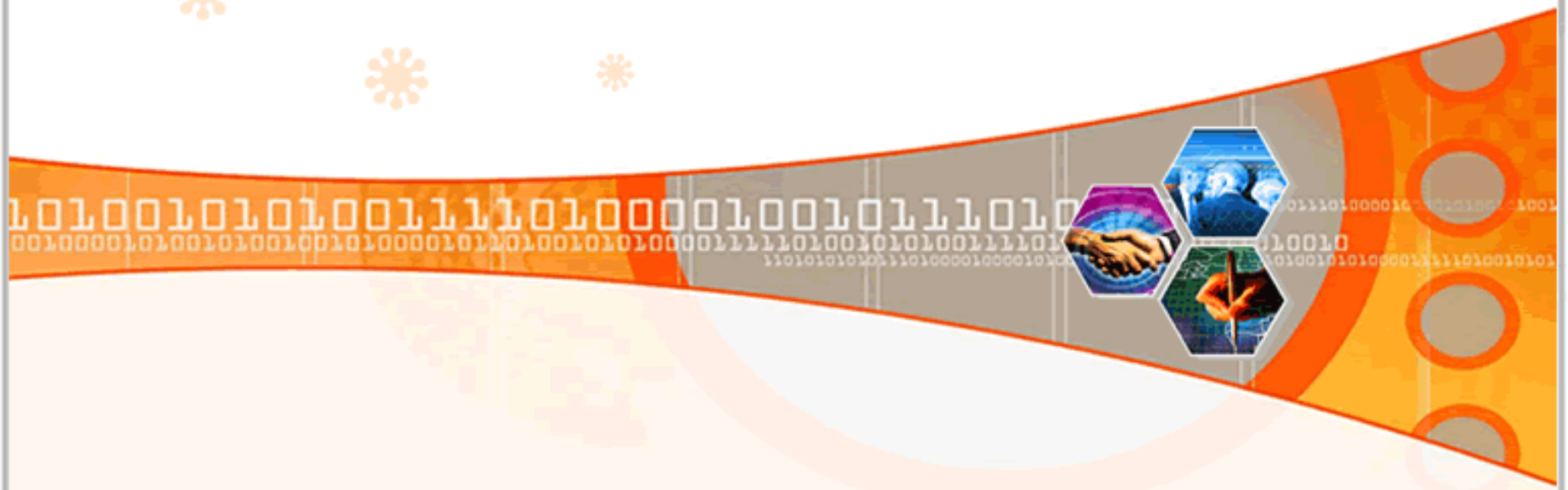
- 模式优缺点
  - ✓中介者模式的缺点
    - 在具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护。

中介者模式

- **模式适用环境**
  - ✓在以下情况下可以使用中介者模式：
    - 系统中对象之间存在复杂的引用关系，产生的相互依赖关系结构混乱且难以理解。
    - 一个对象由于引用了其他很多对象并且直接和这些对象通信，导致难以复用该对象。
    - 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。可以通过引入中介者类来实现，在中介者中定义对象交互的公共行为，如果需要改变行为则可以增加新的中介者类。

设计模式

观察者模式



本章教学内容

- 观察者模式
  - ✓ 模式动机与定义
  - ✓ 模式结构与分析
  - ✓ 模式实例与解析
  - ✓ 模式效果与应用
  - ✓ 模式扩展

观察者模式

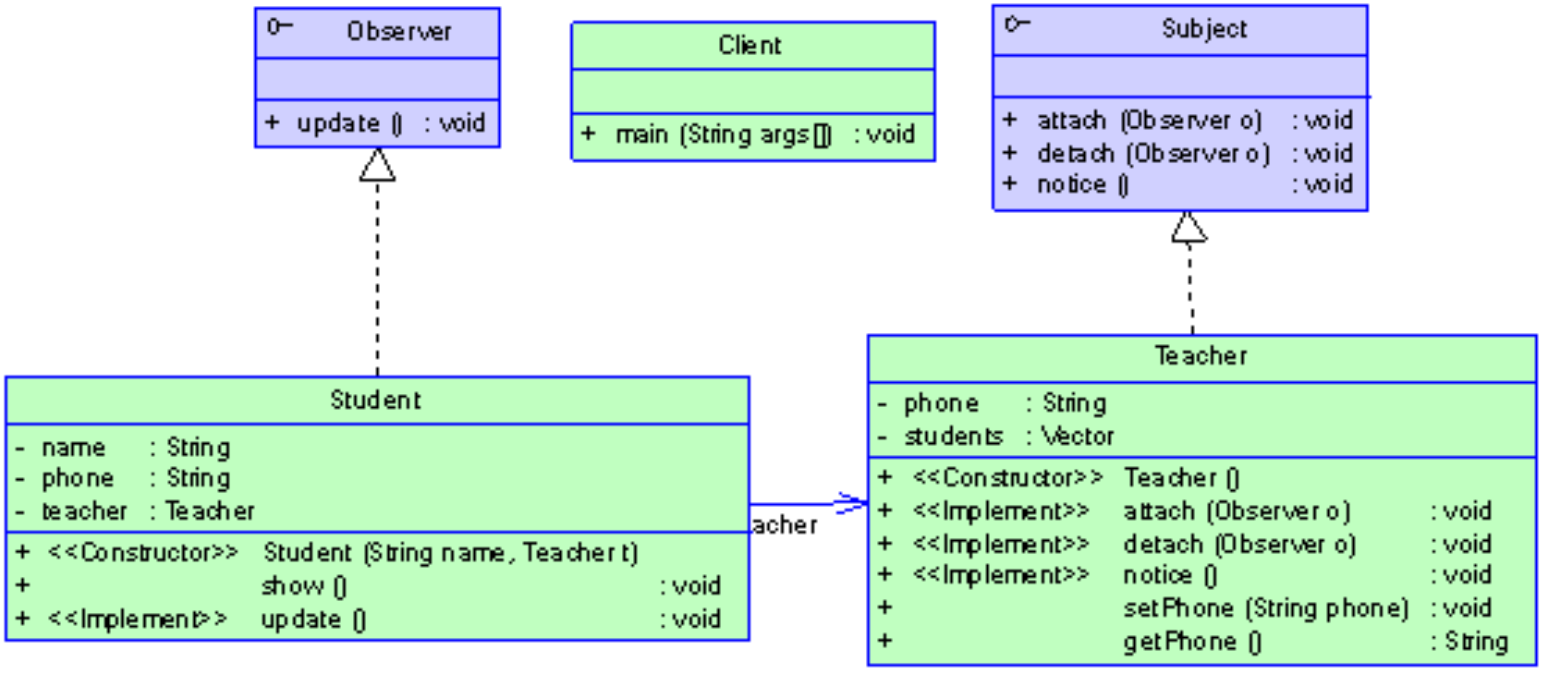
- 模式动机(问题)
  - ✓ 设想学生需要知道老师的电话号码以便于询问课程问题，在这样的组合中，老师就是一个被观察者（**Subject**），学生就是需要知道信息的观察者，当老师的电话号码发生改变时，学生得到通知，并更新相应的电话记录。

观察者模式

- 模式定义
  - ✓ **观察者模式定义：**观察者模式属于行为型模式，其意图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改变时，所有依赖于它的对象都得到通知并被自动更新。这一个模式的关键对象是目标（Subject）和观察者（Observer）。
  - ✓ 一个目标可以有任意数目的依赖它的观察者，一旦目标的状态发生改变，所有的观察者都得到通知，作为对这个通知的响应，每个观察者都将查询目标以使其状态与目标的状态同步。

观察者模式

- 模式结构


Observer
+ update () : void
Client
+ main (String args[]) : void
Subject
+ attach (Observer o) : void
+ detach (Observer o) : void
+ notice () : void
Student
- name : String
- phone : String
- teacher : Teacher
+ <<Constructor>> Student (String name, Teacher t)
+ show () : void
+ <<Implement>> update () : void
acher
Teacher
- phone : String
- students : Vector
+ <<Constructor>> Teacher ()
+ <<Implement>> attach (Observer o) : void
+ <<Implement>> detach (Observer o) : void
+ <<Implement>> notice () : void
+ setPhone (String phone) : void
+ getPhone () : String

观察者模式

- 模式结构
  - ✓观察者模式包含如下角色：
    - Subject (目标)：目标知道它的观察者。可以有任意多个观察者观察同一个目标。
    - Observer (观察者)：为那些在目标发生改变时需要获得通知的对象定义一个更新接口。
    - ConcreteSubject (具体目标)：将有关状态存入各ConcreteObserve对象。
    - ConcreteObserver (具体观察者)：维护一个指向具体目标对象的引用。

观察者模式

- 模式分析
  - ✓ 典型的Subject类代码：

```
package observer;
public interface Subject{
    public void attach(Observer o);
    public void detach(Observer o);
    public void notice();
}
```

观察者模式

- 模式分析
  - ✓ 典型的Observer类代码：

```
package observer;
public interface Observer{
    public void update();
}
```

观察者模式

- 模式分析
  - ✓ 典型的Teacher类代码：

```
public class Teacher implements Subject{
    private String phone;
    private Vector students;
    public Teacher(){
       phone = "";
       students = new Vector();
    }
    public  void attach(Observer o){
       students.add(o);
    }
    public void detach(Observer o){
       students.remove(o);
    }
```

观察者模式

- 模式分析
  - ✓ 典型的Teacher类代码：

```
public class Teacher implements Subject{
    private String phone;
    private Vector students;
    public Teacher(){
       phone = "";
       students = new Vector();
    }
    public  void attach(Observer o){
       students.add(o);
    }
    public void detach(Observer o){
       students.remove(o);
    }
```

观察者模式

- 模式分析
  - ✓ 典型的Teacher类代码（续）：

```
    public void notice(){
        for(int i=0;i<students.size();i++)
            ((Observer)students.get(i)).update();
    }
    public void setPhone(String phone){
        this.phone = phone;
        notice();
    }
    public String getPhone(){
        return phone;
    }
}
```

观察者模式

- 模式分析
  - ✓ 典型的Student类代码：

```
public class Student implements Observer{
    private String name;
    private String phone;
    private Teacher teacher;
    public Student(String name,Teacher t){
       this.name = name;
       teacher = t;
    }
    public void show(){
       System.out.println("Name:"+name+"\nTeacher'sphone:"+phone);
    }
    public void update(){
       phone = teacher.getPhone();
    }
}
```

观察者模式

典型的Client类代码：

```
public class Client{
    public static void main(String[] args){
        Vector students = new Vector();
        Teacher t = new Teacher();
        for(int i= 0 ;i<10;i++){
            Student st = new Student("lili"+i,t);
            students.add(st);
            t.attach(st);
        }
        t.setPhone("88803807");
        for(int i=0;i<10;i++)
                ((Student)students.get(i)).show();
        t.setPhone("88808880");
        for(int i=0;i<10;i++)
                ((Student)students.get(i)).show();
    }
}
```

观察者模式

- 模式适用环境
  - ✓在以下情况下可以使用观察者模式：
    - 当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中可以使他们各自独立地改变和复用。
    - 当对一个对象的改变需要同时改变其它对象，而不知道具体由多少对象有待改变。
    - 当一个对象必须通知其他对象，而它又不能假定其他对象是谁，换言之，你不希望这些对象是紧密耦合的。