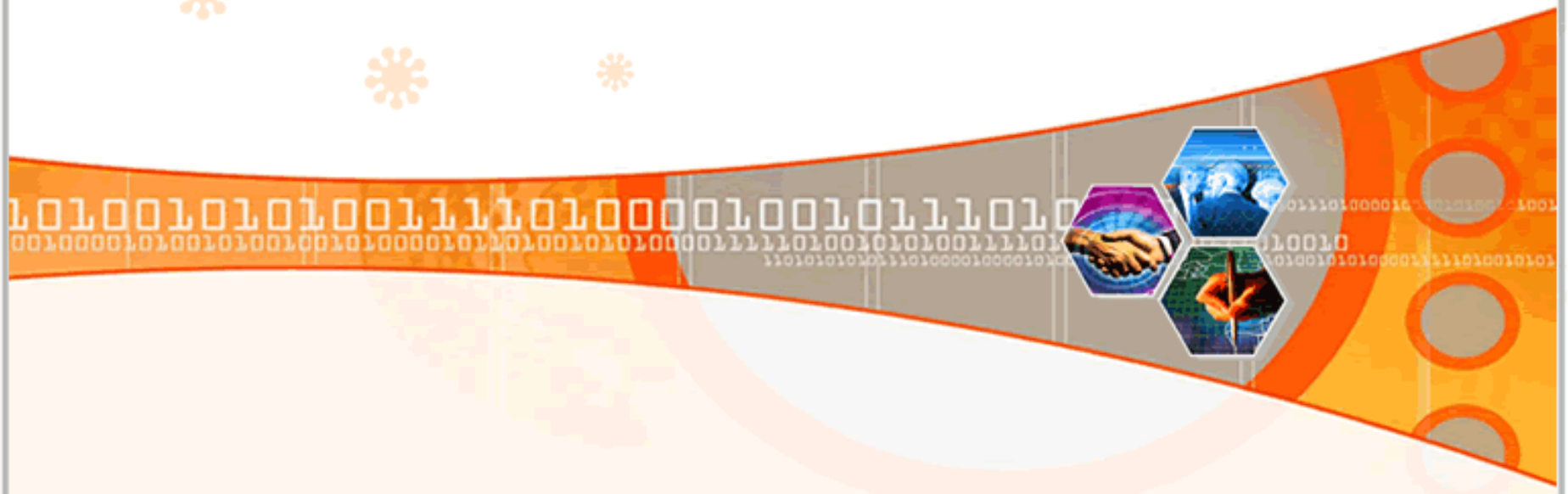
Design Patterns

适配器模式

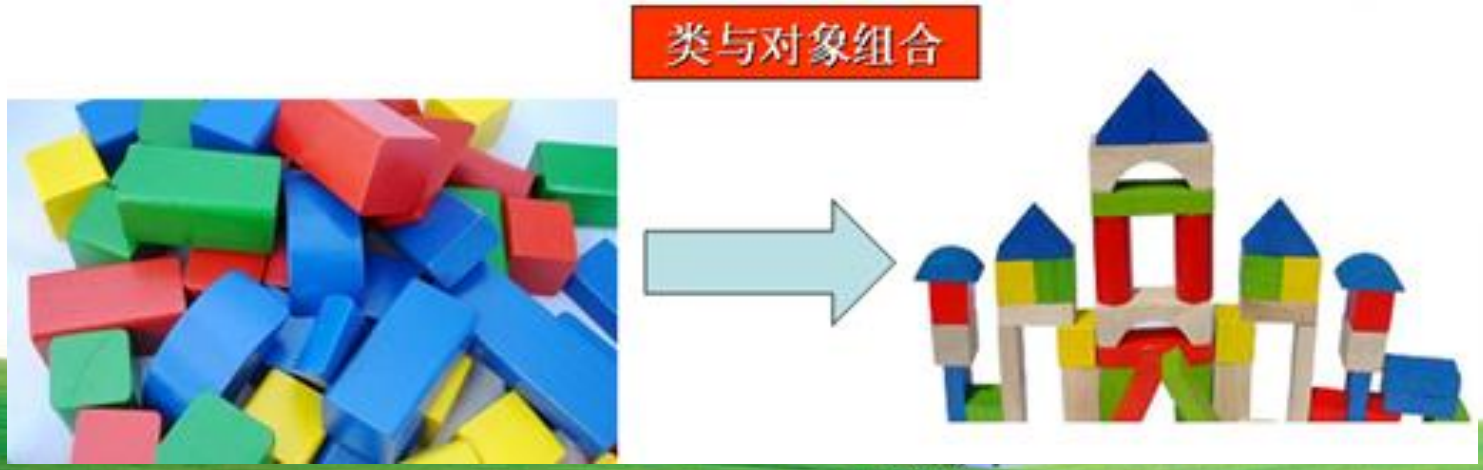


大纲

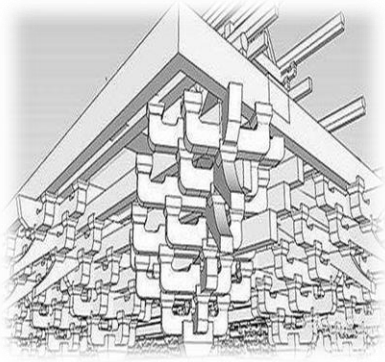
- 结构型模式概述
- 适配器模式概述
- 适配器模式的结构与实现
- 适配器模式的应用实例
- 缺省适配器模式
- 适配器模式的优缺点与适用环境

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

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

- 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.

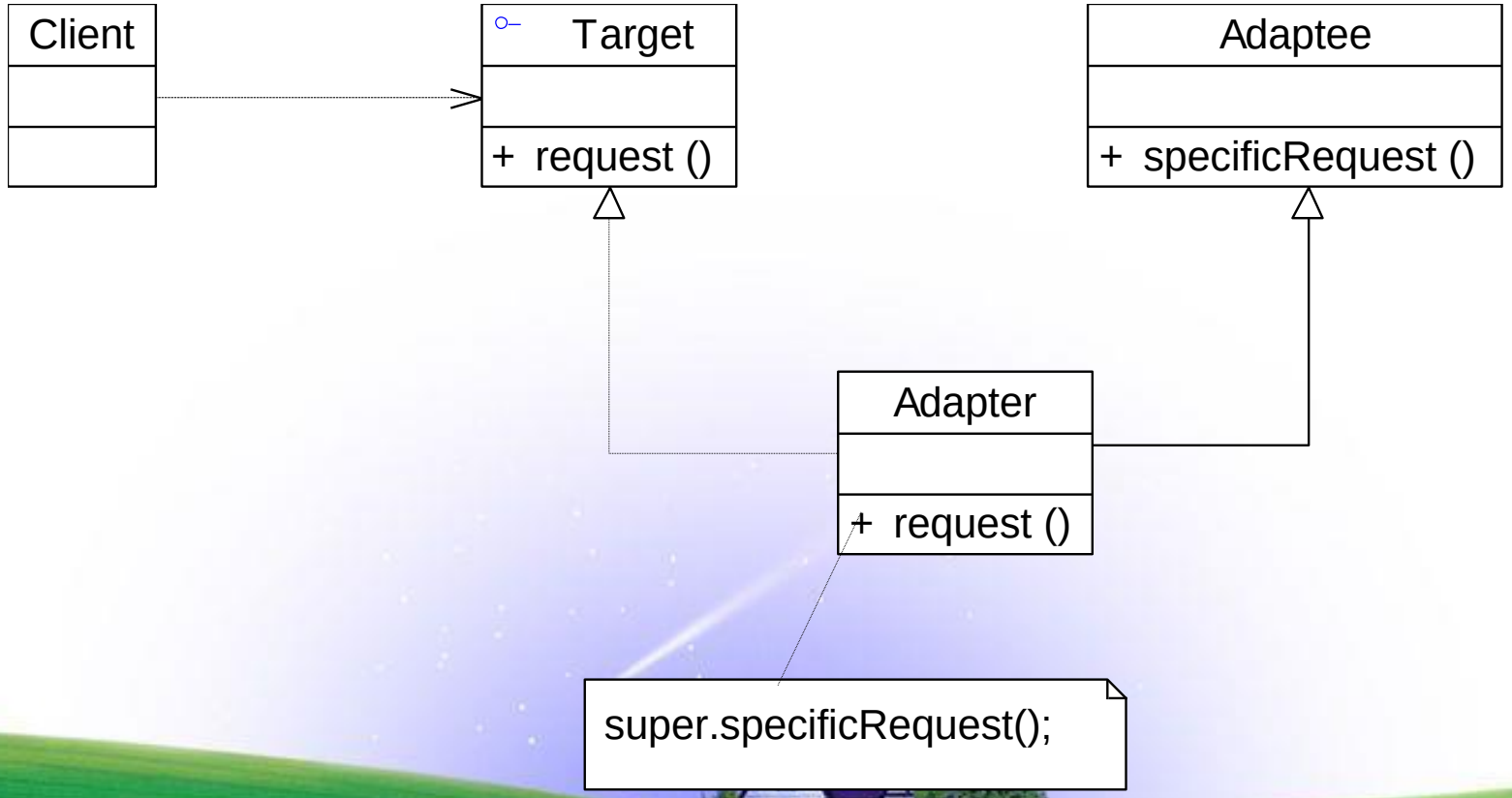
✓对象结构型模式 / 类结构型模式

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

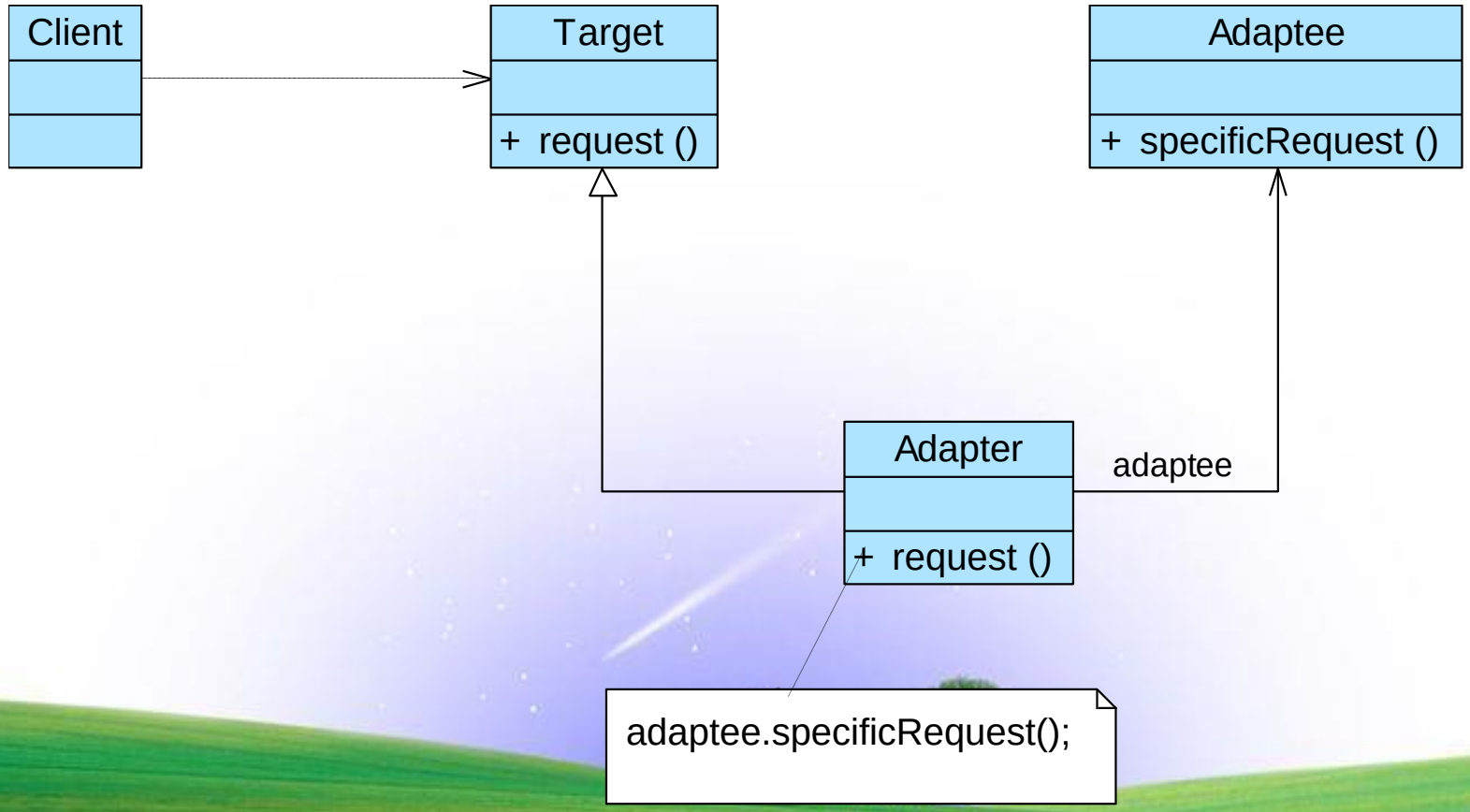
适配器模式的结构与实现

- 适配器模式的结构（类适配器）

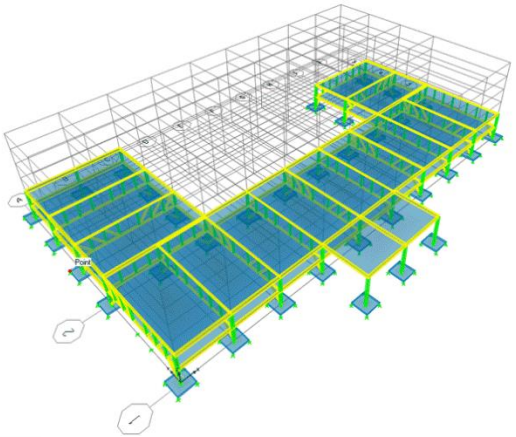


适配器模式的结构与实现

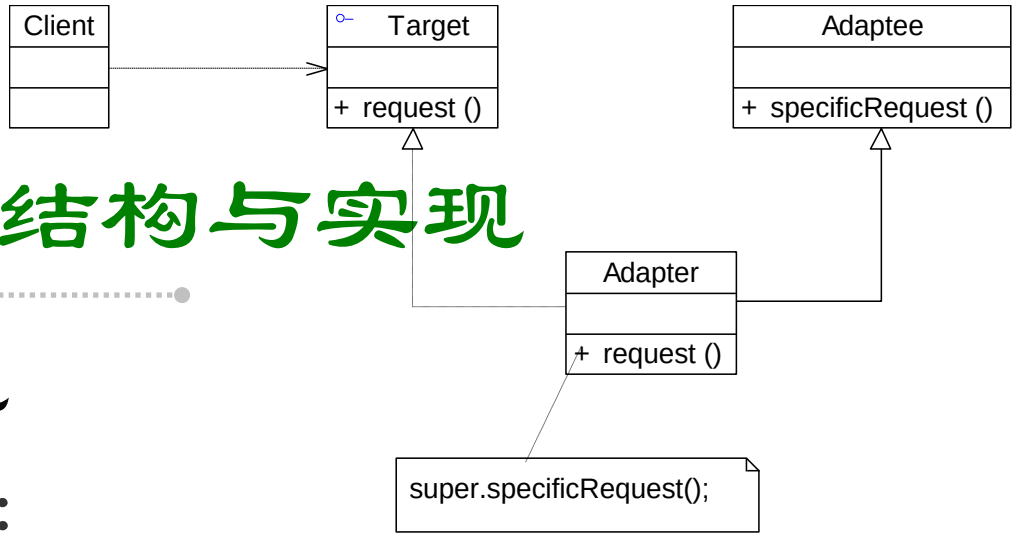
- 适配器模式的结构（对象适配器）



适配器模式的结构与实现



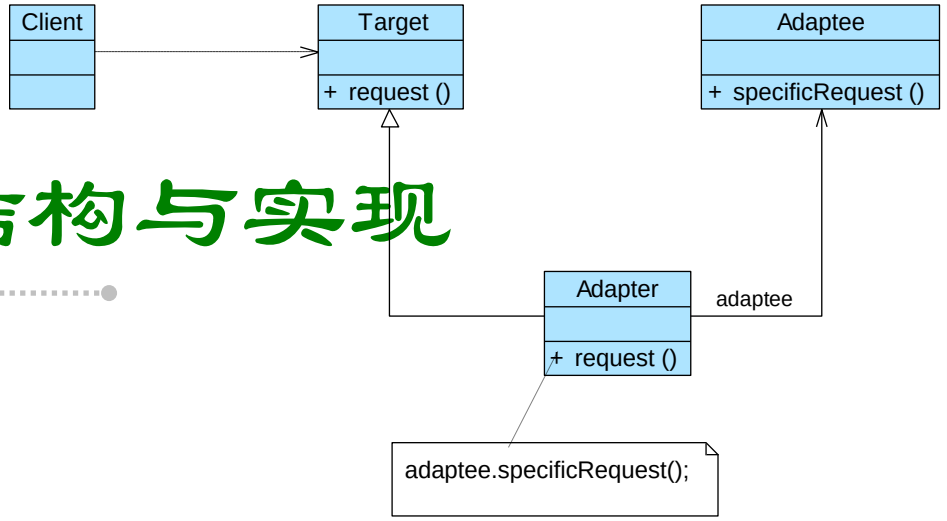
- 适配器模式的结构
  - ✓适配器模式包含以下3个角色:
    - Target（目标抽象类）
    - Adapter（适配器类）
    - Adaptee（适配者类）



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
    }
}
```

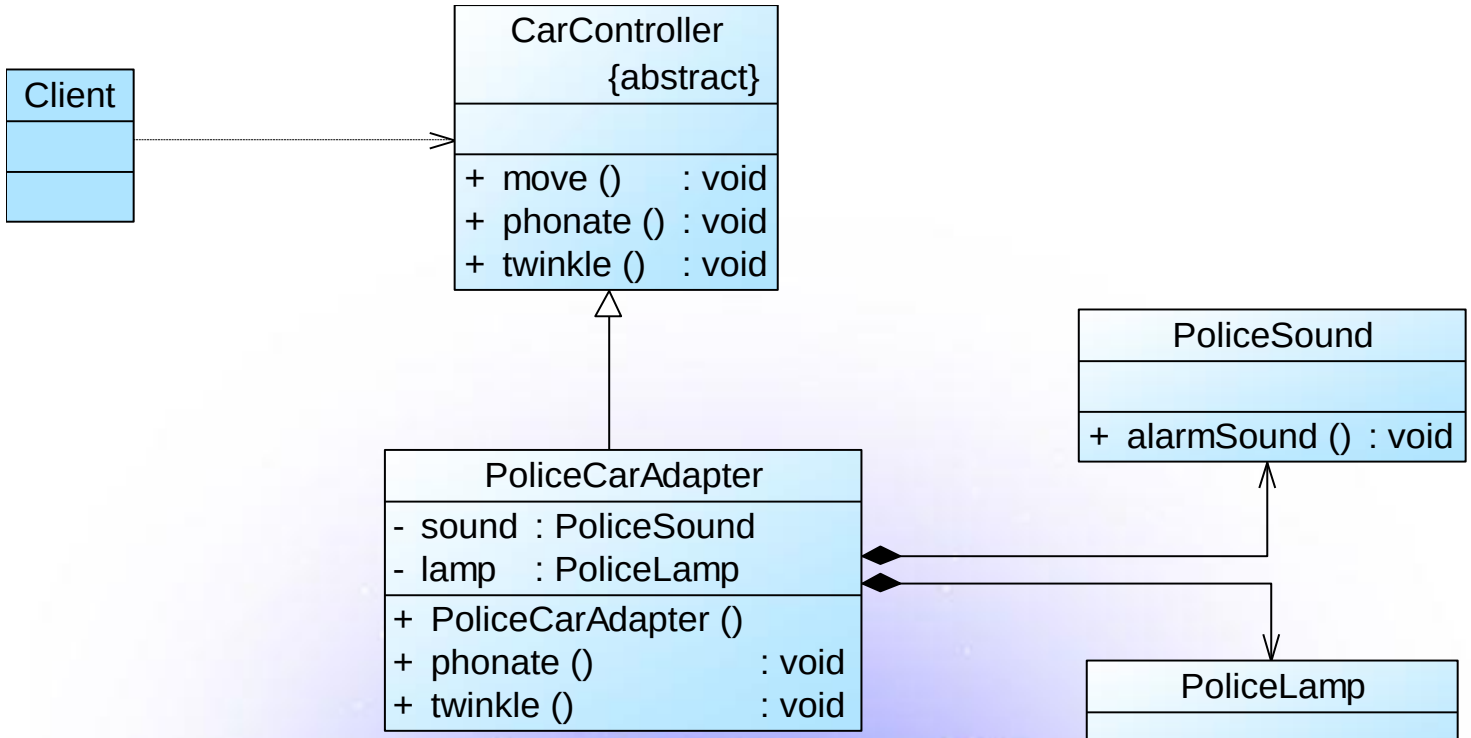
适配器模式的应用实例

实例说明

某公司欲开发一款新的儿童玩具汽车，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“警车”(或“救护车”)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)

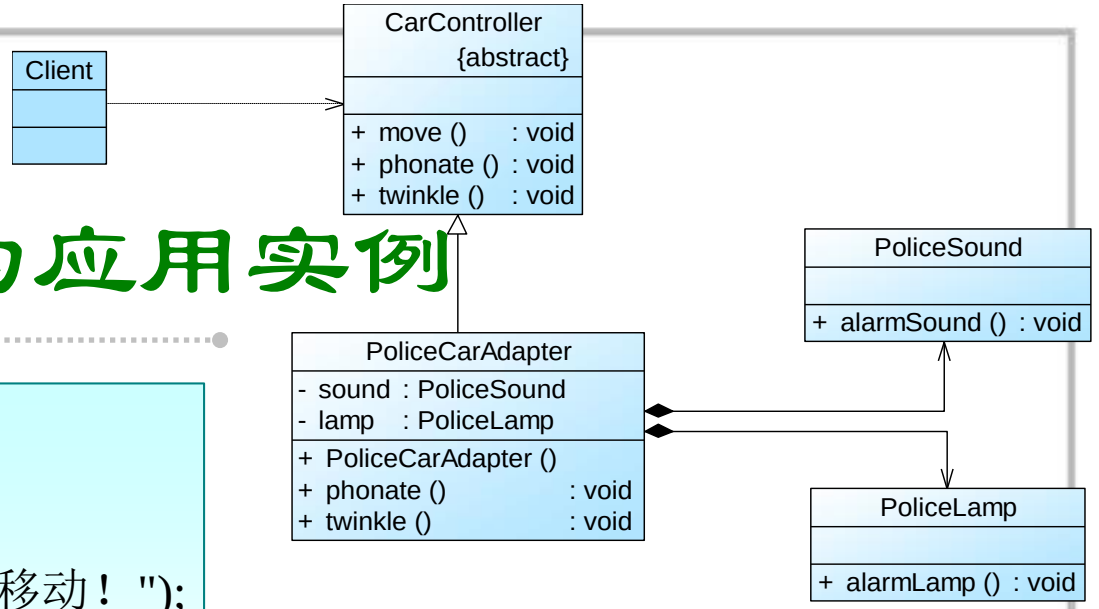
适配器模式的应用实例

- 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

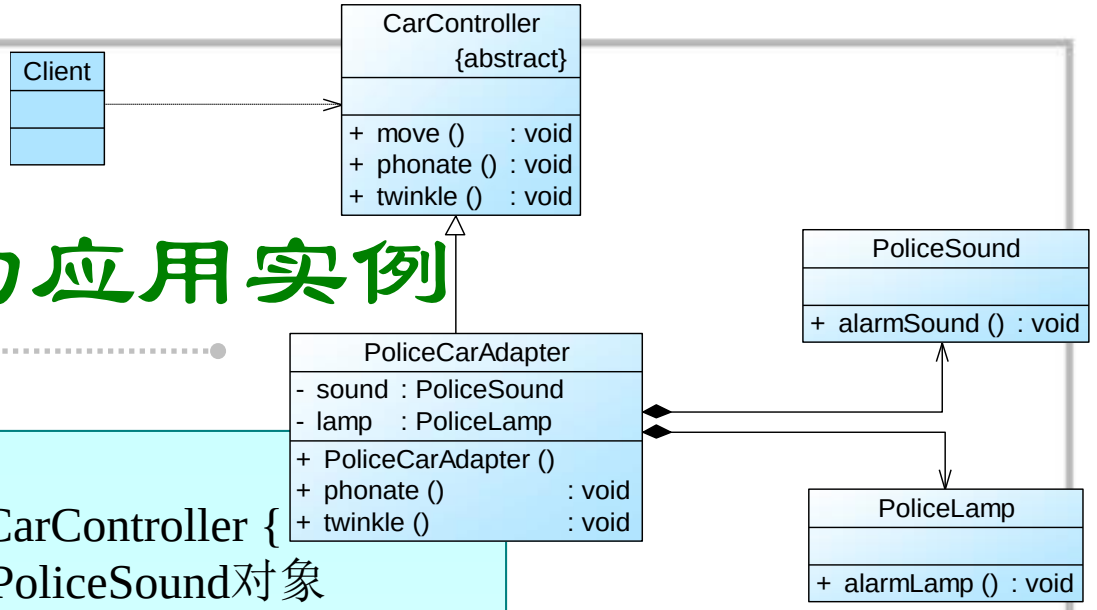



```
//汽车控制类，充当目标抽象类
public abstract class CarController {
    public void move() {
        System.out.println("玩具汽车移动！ ");
    }
    public abstract void phonate(); //发出声音
    public abstract void twinkle(); //灯光闪烁
}
```

```
//警笛类，充当适配者
public class PoliceSound {
    public void alarmSound() {
    System.out.println("发出警笛声音！ ");
   }
}
```

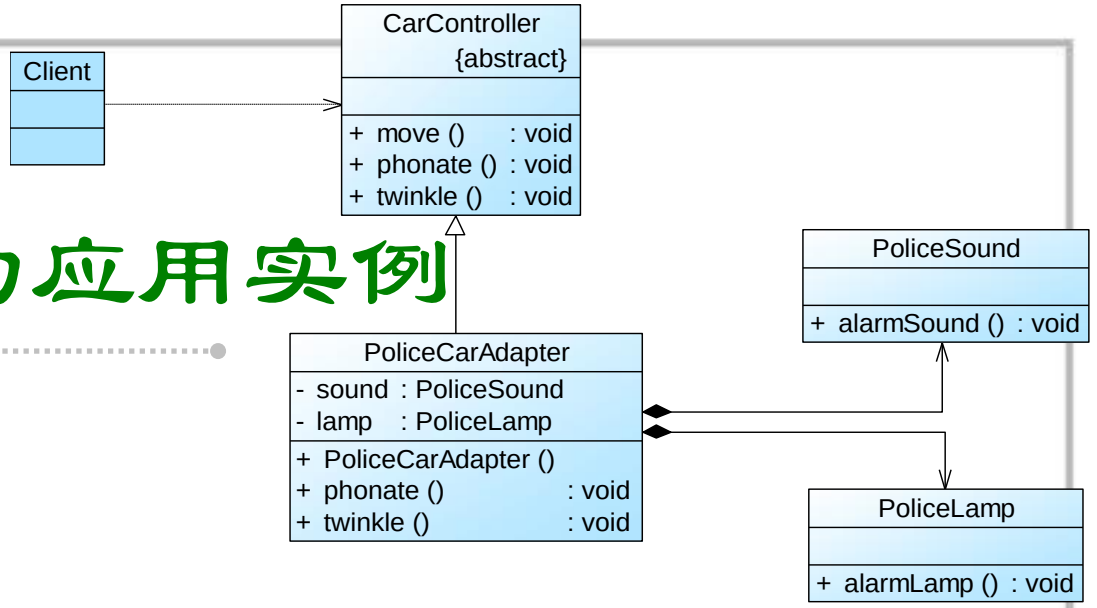
```
//警灯类，充当适配者
public class PoliceLamp {
   public void alarmLamp() {
       System.out.println("呈现警灯闪烁！ ");
   }
}
```

适配器模式的应用实例




```
//警车适配器，充当适配器
public class PoliceCarAdapter extends CarController {
    private PoliceSound sound;//适配者PoliceSound对象
    private PoliceLamp lamp;//适配者PoliceLamp对象
    public PoliceCarAdapter() {
        sound = new PoliceSound();
        lamp = new PoliceLamp();
    }
    //发出警笛声音
    public void phonate() {
        sound.alarmSound();  //调用适配者类PoliceSound的方法
    }
    //呈现警灯闪烁
    public void twinkle() {
        lamp.alarmLamp();   //调用适配者类PoliceLamp的方法
    }
}
```

适配器模式的应用实例




```
public class Client {
    public static void main(String args[]) {
        CarController car ;
        car = new PoliceCarAdapter(); //或new AmbulanceCarAdapter()
        //car = (CarController)XMLUtil.getBean();
        car.move();
        car.phonate();
        car.twinkle();
    }
}
```

```
//救护车适配器，充当适配器
public class AmbulanceCarAdapter extends CarController {
        private AmbulanceSound sound;  //定义适配者AmbulanceSound对象
        private AmbulanceLamp lamp;    //定义适配者AmbulanceLamp对象

        public AmbulanceCarAdapter() {
                sound = new AmbulanceSound();
                lamp = new AmbulanceLamp();
        }

        //发出救护车声音
        public void phonate() {
                sound.alarmSound();  //调用适配者类AmbulanceSound的方法
        }

        //呈现救护车灯闪烁
        public void twinkle() {
                lamp.alarmLamp();   //调用适配者类AmbulanceLamp的方法
        }
}
```

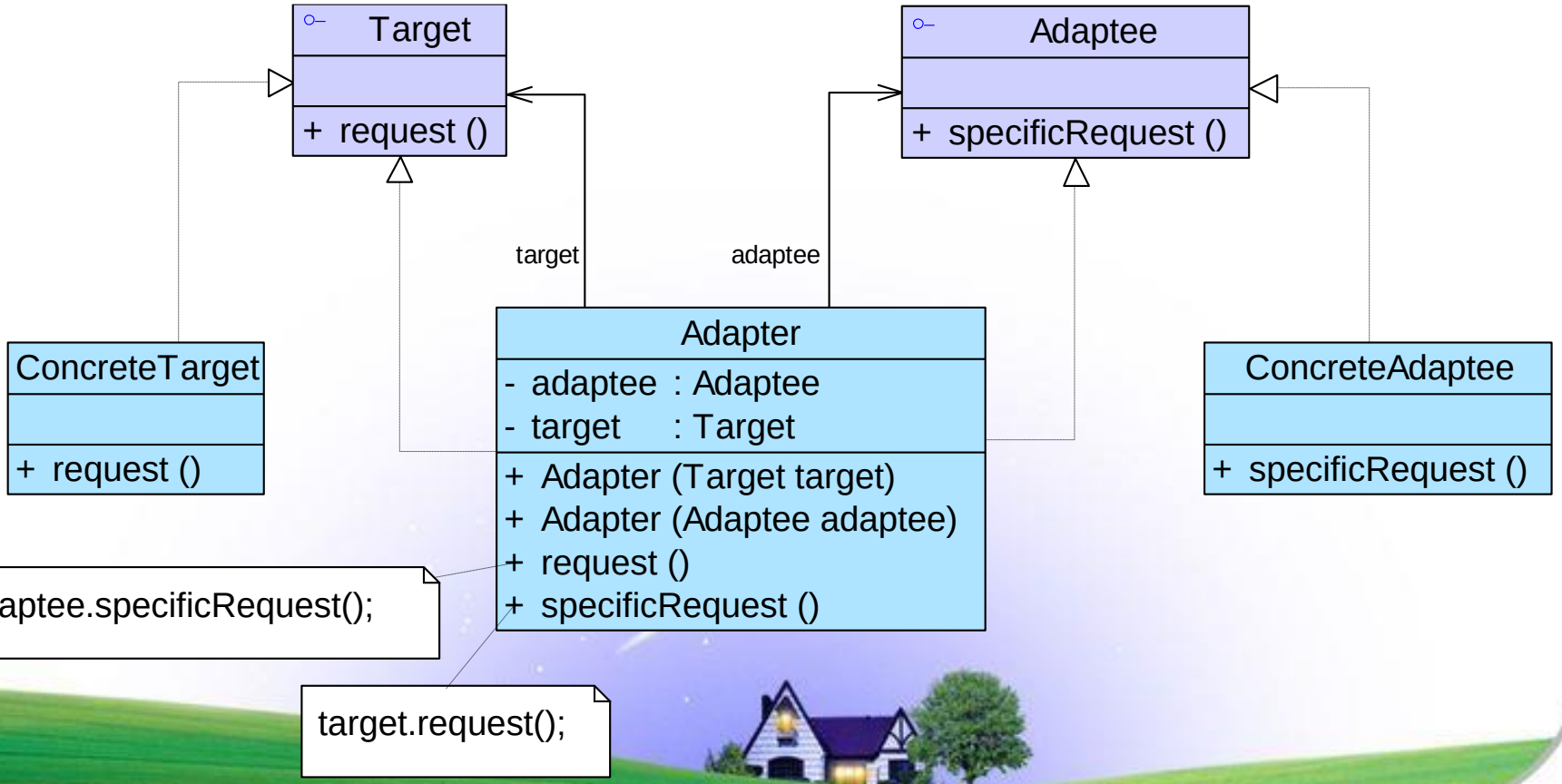
适配器模式的应用实例

```
//救护车灯类，充当适配者
public class AmbulanceLamp {
        public void alarmLamp() {
                System.out.println("呈现救护车灯闪烁！ ");
        }
}

//救护车声音类，充当适配者
public class AmbulanceSound {
        public void alarmSound() {
                System.out.println("发出救护车声音！ ");
        }
}
```

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

```
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，同一个适配者类可在多个不同系统中复用
  - ✓ 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

END