

设计模式

Design Pattern

主讲：辜希武

华中科技大学计算机学院IDC实验室

<http://idc.hust.edu.cn/>

Email: guxiwu@mail.hust.edu.cn

设计模式（Design Pattern）

什么是设计模式

在面向对象程序设计（OOP）过程中，我们经常会遇到很多重复出现的问题，总结解决这些问题的成功经验和最佳实践便形成了设计模式（Design Pattern）。

其核心思想是将可重用的解决方案总结出来，并分门别类。从而指导设计，减少代码重复和优化体系结构。

设计模式（Design Pattern）

采用设计模式的好处

- 重用，避免代码重复冗余
- 优化体系结构
- 提升系统的可维护性和弹性
- 代码更加容易测试，利于测试驱动
- 为性能优化提供便利
- 使软件质量更加有保证
- 增强代码可读性，便于团队交流
- 有助于整体提升团队水平

设计模式（Design Pattern）

设计模式与UML

设计模式是OOP的方法论，其内容就是如何设计对象的结构及其相互间的协作关系。因此需要一种直观的模型将上述内容清晰地表示出来。

统一建模语言（UML）是OOP的建模语言，其核心就是把软件的设计思想通过建模的方法表达出来。故非常适合于表达设计模式。同时UML已经被广泛用于软件设计，这也推动了设计模式的应用。

模式的基本要素

- ✓ **模式名称(Pattern Name)**
- ✓ **问题(Problem)**：描述应该在何时使用模式。解释了设计问题和问题存在的前因后果,可能还描述模式必须满足的先决条件
- ✓ **解决方案(Solution)**：描述了设计的组成成分、相互关系及各自的职责和协作方式。模式就像一个模板，可应用于多种场合，所以解决方案并不描述一个具体的设计或实现，而是提供设计问题的抽象描述和解决问题所采用的元素组合（类和对象）
- ✓ **效果(consequences)**：描述模式的应用效果及使用模式应权衡的问题

如何描述设计模式

◆模式名和分类

◆**意图**：设计模式是做什么的？它的基本原理和意图是什么？它解决的是什么样的特定设计问题？

◆**动机**：说明一个设计问题以及如何用模式中的类、对象来解决该问题的特定情景

◆**适用性**：什么情况下可以使用该设计模式？该模式可用来改进哪些不良设计？如何识别这些情况？

◆**结构**：采用对象建模技术对模式中的类进行图形描述

描述设计模式（续）

- ◆ **参与者**：指设计模式中的类 和/或 对象以及它们各自的职责
- ◆ **协作**：模式的参与者如何协作以实现其职责
- ◆ **效果**：模式如何支持其目标？使用模式的效果和所需做的权衡取舍？系统结构的哪些方面可以独立改变？
- ◆ **实现**：实现模式时需了解的一些提示、技术要点及应避免的缺陷，以及是否存在某些特定于实现语言的问题
- ◆ **代码示例**：用来说明怎样实现该模式的代码片段
- ◆ **相关模式**：与这个模式紧密相关的模式有哪些？其不同之处是什么？这个模式应与哪些其他模式一起使用？

设计模式的原则

- ◆ "开-闭"原则(Open Closed Principal)
- ◆ 单一职责原则
- ◆ 里氏代换原则
- ◆ 依赖倒置原则
- ◆ 接口隔离原则

- ◆ 设计模式就是实现了上述原则，从而达到代码复用、增加可维护性的目的

开闭原则（OCP）

- ◆ **定义**：软件对扩展是开放的，对修改是关闭的。开发一个软件时，应可以对其进行功能扩展(开放)，在进行扩展的时候，不需要对原来的程序进行修改(关闭)
- ◆ **好处**：在软件可用性上非常灵活。可以在软件完成后对软件进行扩展，加入新的功能。这样，软件就可通过不断的增加新模块满足不断变化的新需求
 - ◆ 由于不修改软件原来的模块，不用担心软件的稳定性

开闭原则（OCP）

◆ 实现的主要原则

- ◆ **抽象原则**：把系统的所有可能的行为抽象成一个底层；
由于可从抽象层导出一个或多个具体类来改变系统行为，
因此对于可变部分，系统设计对扩展是开放的
- ◆ **可变性封装原则**：对系统所有可能发生变化的部分进行评估和分类，
每一个可变的因素都单独进行封装

单一职责原则 (SRP)

- ◆ 一个类一个职责
- ◆ 当类具有多职责时,应把多余职责分离出去, 分别创建一些类来完成每一个职责
- ◆ 每一个职责都是一个变化的轴线, 当需求变化时会反映为类的职责的变化

举例

```
interface Modem{  
    public void dial(String pno);  
    public void hangup();  
    public send(char c);  
    public char recv();  
}
```

Modem类有两个职责：连接管理和数据通信，应将它们分离

IsKov替换原则（LSP）

- ◆ **定义**：“继承必须确保超类所拥有的性质在子类中仍然成立”，当一个子类的实例能够替换任何其超类的实例时，它们之间才具有is-A关系
- ◆ 里氏替换原则是继承复用的基石，只有当派生类可以替换掉其基类，而软件功能不受影响时，基类才能真正被复用，派生类也才能够在基类的基础上增加新的行为
- ◆ **LSP本质**：在同一个继承体系中的对象应该有共同的行为特征
- ◆ **例子**：企鹅是鸟吗？
 - 生物学：企鹅属于鸟类
 - LSP原则：企鹅不属于鸟类，因为企鹅不会“飞”
- ◆ **违反LSP的后果**：有可能需要修改客户代码

listKov替换原则（LSP）

- ◆ **定义1**：如果对每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，使得以 T1 定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。
- ◆ **定义2**：所有引用基类的地方必须能透明地使用其子类的对象。
- ◆ **问题由来**：有一功能 P1，由类 A 完成。现需要将功能 P1 进行扩展，扩展后的功能为 P，其中 P 由原有功能 P1 与新功能 P2 组成。新功能 P 由类 A 的子类 B 来完成，则子类 B 在完成新功能 P2 的同时，有可能会破坏原有功能 P1 发生故障。
- ◆ **解决方案**：当使用继承时，遵循里氏替换原则。类 B 继承类 A 时，除添加新的方法完成新增功能 P2 外，尽量不要重写父类 A 的方法，也尽量不要重载父类 A 的方法。

listKov替换原则 (LSP)

```
class A{  
    public int func1(int a, int b){  
        return a-b;  
    }  
}  
  
public class Client{  
    public static void main(String[] args){  
        A a = new A();  
        System.out.println("100-50="+a.func1(100, 50));  
        System.out.println("100-80="+a.func1(100, 80));  
    }  
}
```

listKov 替换原则 (LSP)

后来，我们需要增加一个新的功能：完成两数相加，然后再与100求和，由类B来负责。即类B需要完成两个功能：

两数相减。

两数相加，然后再加100。

```
class B extends A{  
    public int func1(int a, int b){  
        return a+b;  
    }  
    public int func2(int a, int b){  
        return func1(a,b)+100;  
    }  
}
```

listKov替换原则（LSP）

```
public class Client{  
    public static void main(String[] args){  
        A b = new B();  
        System.out.println("100-50="+b.func1(100, 50));  
        System.out.println("100-80="+b.func1(100, 80));  
        System.out.println("100+20+100="+b.func2(100, 20));  
    }  
}
```

我们发现原本运行正常的相减功能发生了错误。原因就是类B在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类B重写后的方法，造成原本运行正常的功能出现了错误。

listKov替换原则（LSP）

- ◆ 在本例中，引用基类A完成的功能，换成子类B之后，发生了异常。
- ◆ 在实际编程中，我们常常会通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大。
- ◆ 如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖、聚合，组合等关系代替。
- ◆ 里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。

依赖倒置原则（DIP）

◆ 定义：**高层模块不应依赖低层模块，二者都应该依赖于抽象**

◆ 高层模块只应该包含重要的业务模型和策略选择，低层模块则是不同业务和策略的实现

◆ 高层抽象不依赖高层和低层模块的具体实现，最多只依赖于低层的抽象

◆ 低层抽象和实现也只依赖于抽象

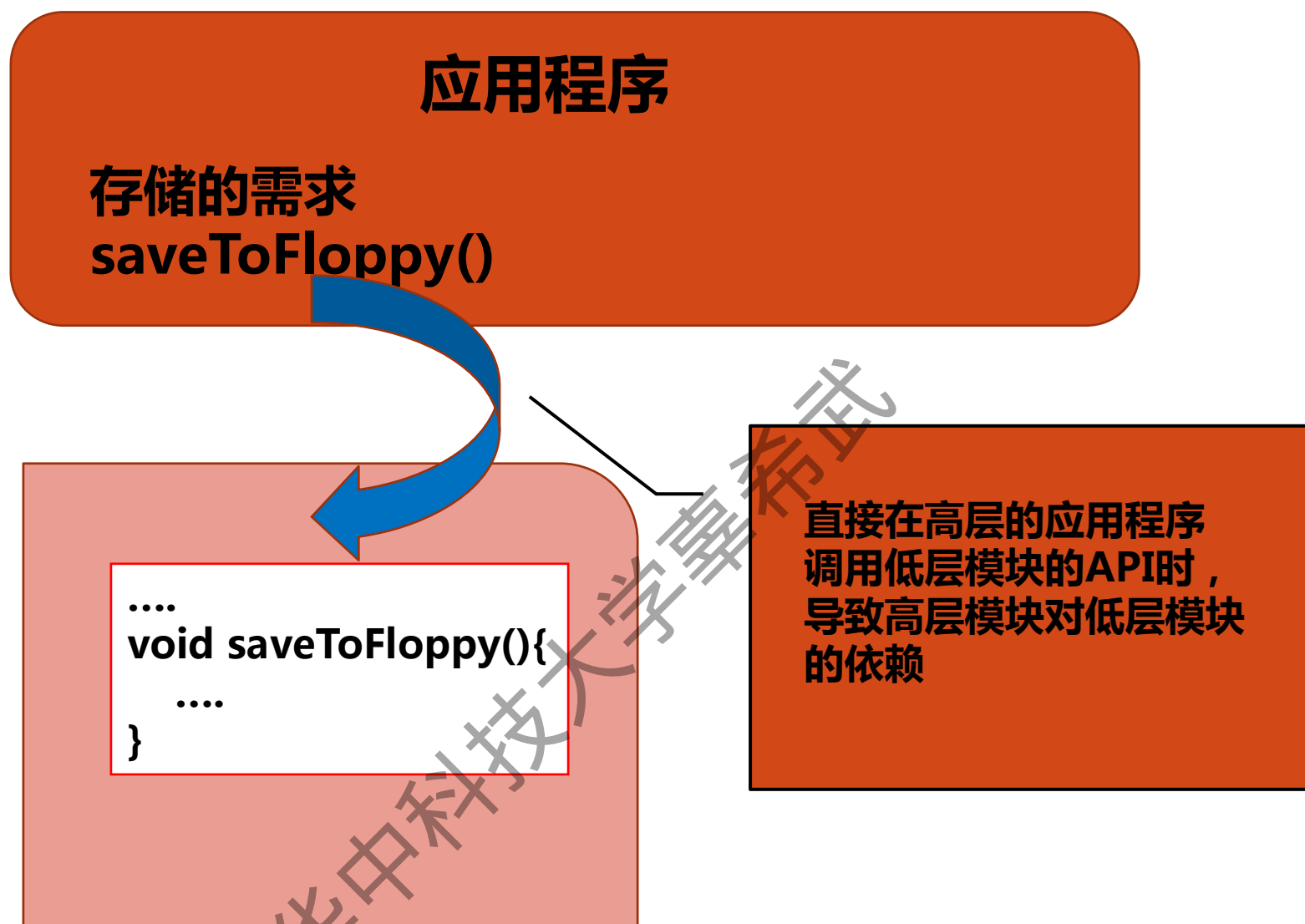
◆ 辅助原则

◆ 任何变量的类型都不应该是具体类

◆ 任何类都不应该从具体类派生

◆ 任何方法都不应覆盖其任何基类中已经实现了的方法

依赖倒置原则（DIP）举例



依赖倒置原则（DIP）举例

```
public class Business{  
    private FloppyWriter writer = new FloppyWriter ();  
    ...  
    public void save(){  
        writer.saveToFloppy();  
    }  
}
```

高层业务类 Business的实现依赖于低层FloppyWriter 类，如果想把存储介质改为USB盘，则必须修改Business类，使得无法重用Business类

违反了依赖倒置原则！

依赖倒置原则（DIP）举例

利用接口抽象，可以改进

```
public interface IDeviceWriter{  
    public void saveToDevice();  
}  
  
public class Business{  
    private IDeviceWriter writer;  
    ...  
    public void setDeviceWriter(IDeviceWriter writer){  
        this.writer = writer;  
    }  
    public void save(){  
        writer.saveToDevice();  
    }  
}
```

高层（Business类）的
实现依赖于抽象（接口）

依赖倒置原则（DIP）举例

在这样的设计下，Business就是可重用的。如果今天有存储至Floppy或USB的需求，只要针对这两种存储需求分别实现IDeviceWriter接口即可。

```
public class FloppyWriter implements IDeviceWriter{  
    public void saveToDevice{  
        ... //存储至软盘的代码  
    }  
}  
  
public class USBWriter implements IDeviceWriter{  
    public void saveToDevice{  
        ...//存储至USB的代码  
    }  
}
```

低层类的实现也
依赖于抽象（接口）

依赖倒置原则（DIP）举例

如果应用程序需要USB存储，则编写如下配置程序：

```
Business business= new Business();  
business.setDeviceWriter(new USBWriter()); //对象注入  
business.save();
```

可以看到，无论低层的存储实现如何变动，对于Business类来说都无需修改。

如果采用对象工厂模式来动态创建实现存储功能的低层对象，连上面三行代码都不用写，只需要编写一个XML的配置文件，在配置文件里指定要注入到Business里面的低层存储对象即可。

这就是著名的Spring框架的一个重要功能：**对象注入，或者叫依赖注入（Dependency Injection）**

接口隔离原则（ISP）

- ◆ 多个和客户相关的接口要好于一个通用接口
- ◆ 如果一个类有几个使用者，与其让这个类载入所有使用者需要使用的所有方法，还不如为每个使用者创建一个特定接口，并让该类分别实现这些接口

华中科技大学


```
public class SeaPlane{  
    //与飞行有关的方法  
    void takeOff() { ... }  
    void fly() { ... }  
    void land() {...}  
  
    //与海上航行有关的方法  
    void dock() { ... }  
    void cruise() {...}  
}
```

```
//客户代码1  
//只使用飞行功能
```

```
void f1(SeaPlane o){  
    o.takeOff();  
    o.fly();  
    o.land();  
}  
f1(new SeaPlane());
```

```
//客户代码2  
//只使用海上航行功能
```

```
void f2(SeaPlane o){  
    o.dock();  
    o.cruise();  
}  
f2(new SeaPlane());
```

类SeaPlane封装了二类不同的功能。有二段客户代码分别只需要使用其中一类功能。但不得不声明SeaPlane类型的对象引用作为方法参数，导致客户代码被绑定到SeaPlane类型。违反了一个原则：**基于接口编程，不要基于类编程**

```
public interface Flyer  
//与飞行有关的方法  
void takeOff();  
void fly() ;  
void land();  
} //声明Flyer接口
```

```
public interface Sailer  
//与海上航行有关的方法  
void dock();  
void cruise();  
} //声明Sailer接口
```

```
public class SeaPlane  
implements Flyer, Sailer{  
//实现与飞行有关的方法  
void takeOff() { ... }  
void fly() { ... }  
void land() {...}  
  
//实现与海上航行有关的方法  
void dock() { ... }  
void cruise() {...}  
}
```

现在首先声明二个分离的功能接口Flyer和Sailer

再用SeaPlane实现这二个接口，这样SeaPlane同时具有了二类不同的行为特征

//客户代码1
//只使用飞行功能

```
void f1(Flyer o){  
    o.takeOff();  
    o.fly();  
    o.land();  
}
```

f1(new SeaPlane());

//客户代码2
//只使用海上航行功能

```
void f2(Sailer o){  
    o.dock();  
    o.cruise();  
}
```

f2(new SeaPlane());

接口类型的引用变量可以直接指向实现该接口的对象而不用强制类型转换！！实参传递给形参时，相当于
Flyer o = new SeaPlane();

当接口类型的引用变量指向不同的实现了接口的对象时，会表现出多态性

现在通过接口，将二类功能分离。同时方法的参数类型是接口，只要接口定义不变，接口的实现类再怎么修改，方法f1和f2都不需要修改。更重要的是，当传递新的接口实现类对象给方法时，方法会自动具有新的行为。这就是接口分离和基于接口编程的好处！

设计模式的类型

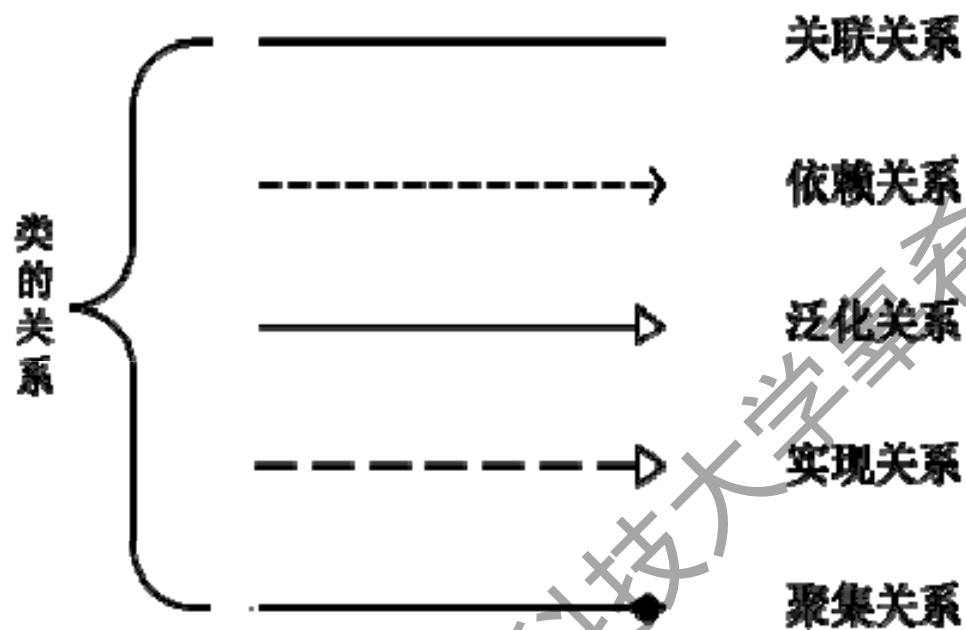
- ◆在设计模式经典著作《GOF95》中，设计模式从应用的角度被分为三个大的类型
 - ◆创建型模式/结构型模式/行为型模式
- ◆根据模式的范围分，模式用于类还是用于对象
 - ◆**类模式**：处理类和子类之间的关系，这些关系通过继承建立，是静态的，在编译时刻便确定下来了
 - ◆**对象模式**：处理对象间的关系，这些关系在运行时刻是可以变化的，更具动态性
- ◆从某种意义上来说，几乎所有模式都使用继承机制，所以“类模式”只指那些集中于处理类间关系的模式，而大部分模式都属于对象模式的范畴

Object Modeling Technique (OMT)

类的关系总结

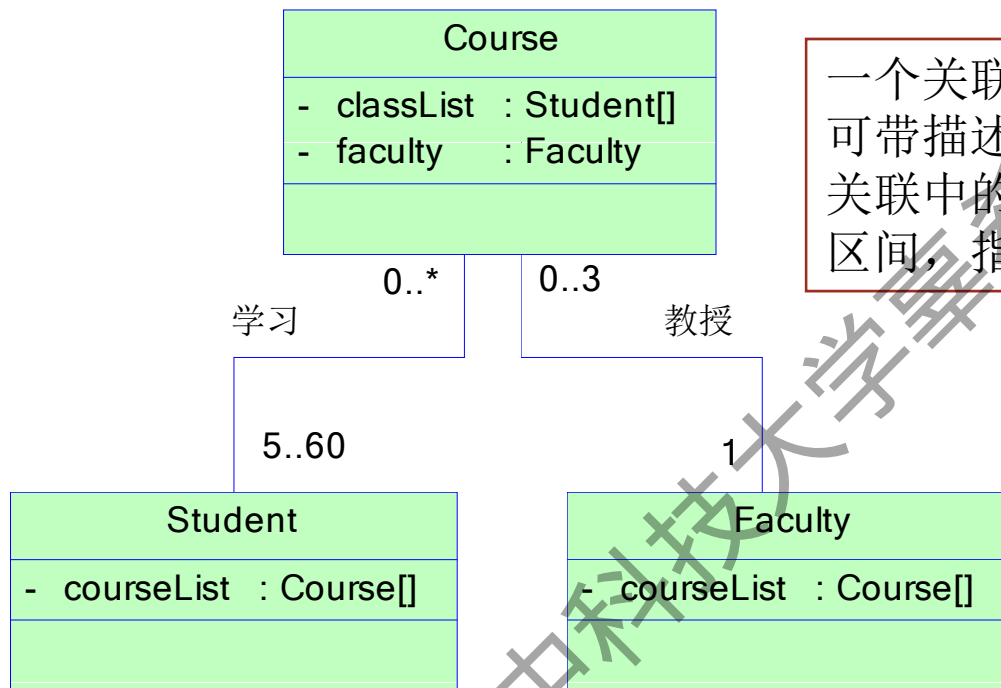
- 类与类之间的关系
 - 关联关系
 - 聚合关系
 - 组合关系
 - 依赖关系
 - 继承关系
- 类与接口之间的关系
 - 实现关系

类的关系的UML表示



关联关系

- 关联关系(association)是一种通用的二元关系，例如学生(Student)选学课程(Course)，教师(Faculty)教授课程(Course)，这些联系可以在UML中表示。



一个关联可以用两个类之间的实线表示。
可带描述关系的标签。
关联中的每个类可以指定一个数目或数字区间，指出这个关系涉及多少个对象

关联关系的实现

- 在Java代码中，关联关系可以用数据域或方法来实现。对于方法，一个类中的方法包含另一个类的参数。

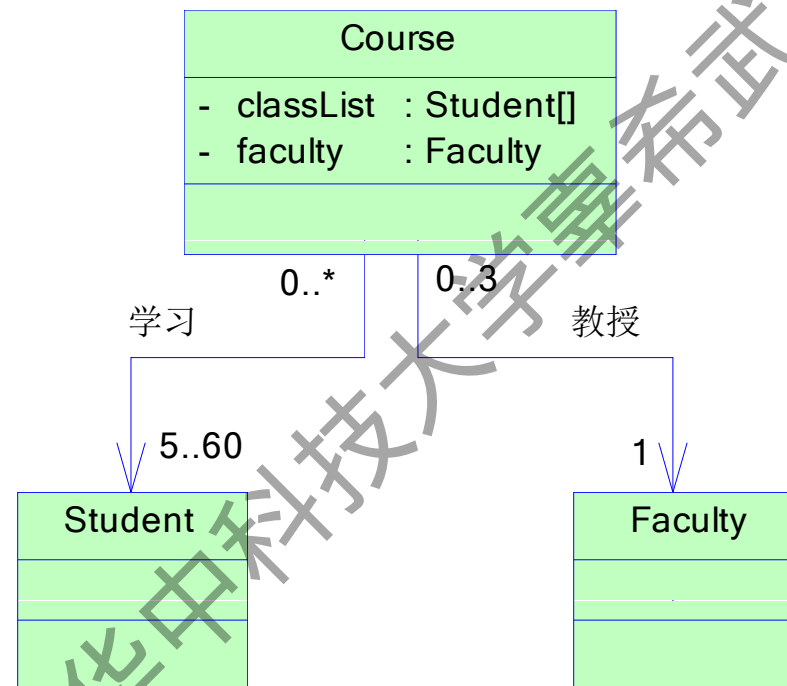
```
public class Student{  
    private Course[] courseList;  
    public void addCourse( Course c){ ..... }  
}
```

```
public class Course{  
    private Student[] classList;  
    private Faculty faculty;  
    public void addStudent( Students s){..... }  
    public void setFaculty( Faculty f) { ..... }  
}
```

```
public class Faculty{  
    private Course[] courseList;  
    public void addCourse( Course c){ ..... }  
}
```

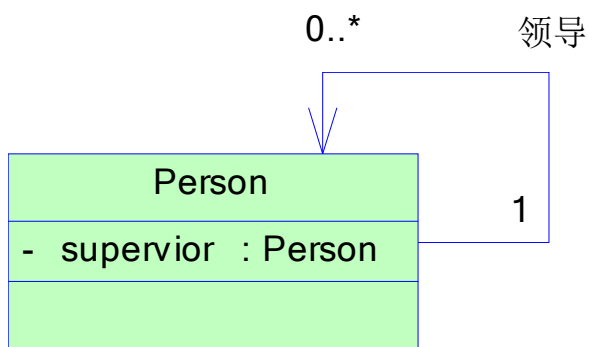

单向关联

- 如果学生或教师不需要知道课程的信息，可以去掉 **courseList** 域，形成单向关联。这时可将 **Student** 类和 **Faculty** 类中的数据域 **courseList** 和 **addCourse** 方法去掉。



自关联

- 同一个类的对象间存在关联，称自关联。例如，一个人有一个领导。



```
public class Person{  
    private Person supervisor;  
    .....  
}
```

聚合和组合

- 聚合关系(aggregation)是一种特殊的关联关系，表示整体与部分之间的关系，即**has-a**的关系。所有者成为聚集者，从属对象称为被聚集者。在聚合关系中，一个从属者可以被多个聚集者拥有(Weak has a)。
- 组合关系(composition)是聚合关系的一种特殊形式，表示从属者强烈依赖于聚集者。一个从属者只能被一个聚集者所拥有，聚集者负责从属者的创建和销毁(Strong has a)。



一个Name对象只能为一个Person所有，但一个Address对象可以被多个Person共享

聚合和组合

- 聚集关系和组合关系在代码中通常表示为聚集类中的数据域，如上图中的关系可以表示为

```
public class
Name{

    ...

}
```

```
public class Person{
    private Name name;
    private Address address;
    ...
}
```

```
public class
Address{

    ...

}
```

- 对于组合关系，聚集者往往负责对从属者的创建和销毁，而聚集关系则不是。

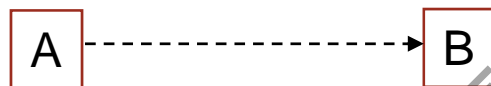
```
public class Person{
    private Name name;
    private Address address;
    public Person(Address a){
        name = new Name();
        address = a;
    }
}
```

Address对象是传递进来的一个引用，而Name对象是在Person对象创建时才创建。

当Person对象被析构时，Name对象也被析构，而Address对象可能还存在

依赖关系

- 依赖关系（dependency）指的是两个类之间一个（称为client）使用另一个（称为supplier）的关系。在UML中，从client画一条带箭头的虚线指向supplier类。
- 例如，A类每个功能的实现要依赖于B类的对象，因此A和B之间的关系可以用依赖描述。

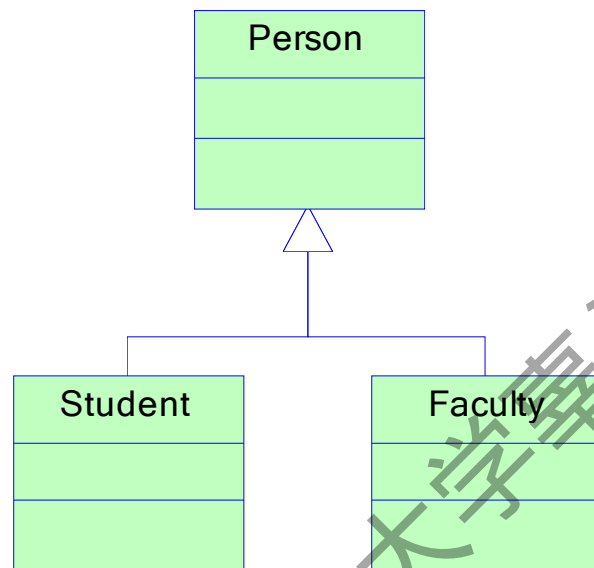


- 依赖关系在代码中通常表示为client类的成员函数以supplier类型的对象为参数

```
public class A{  
    public void fa(B o){  
        ...  
        o.fb();  
    }  
}
```

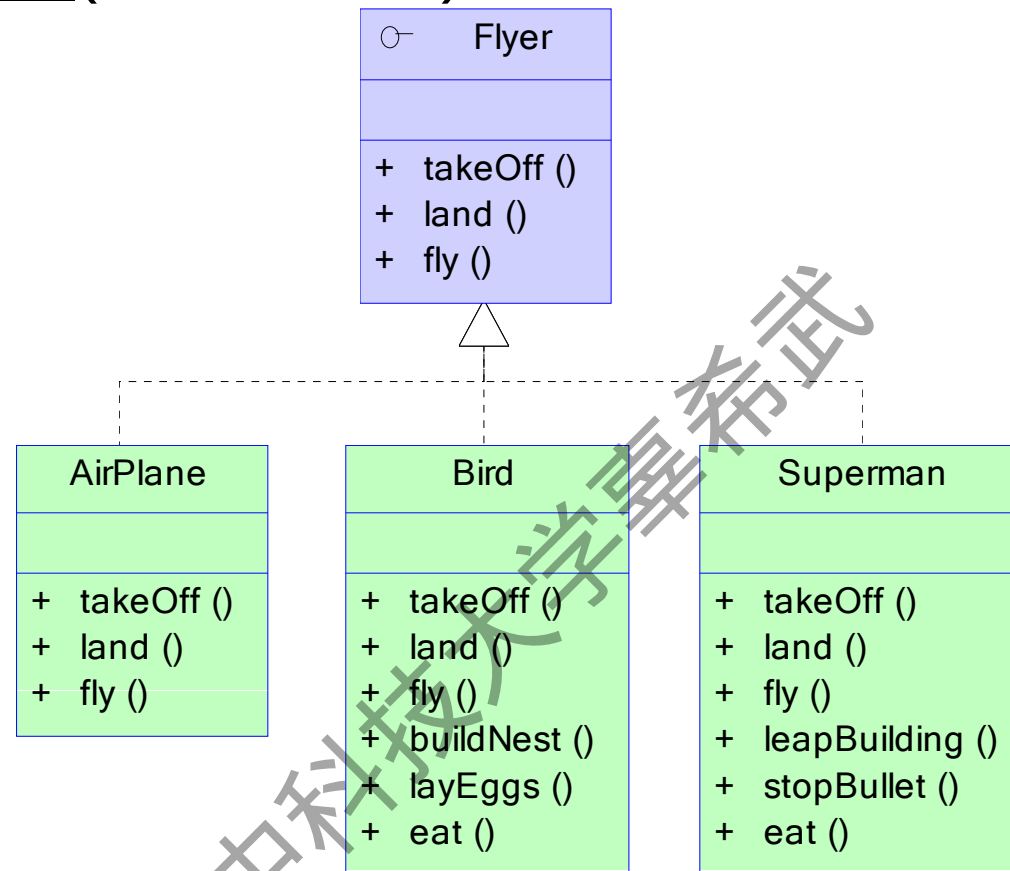
继承关系

- 继承关系(inheritance)表示is-a的关系。



实现关系

- 实现关系(realization)表示类和接口之间的关系。

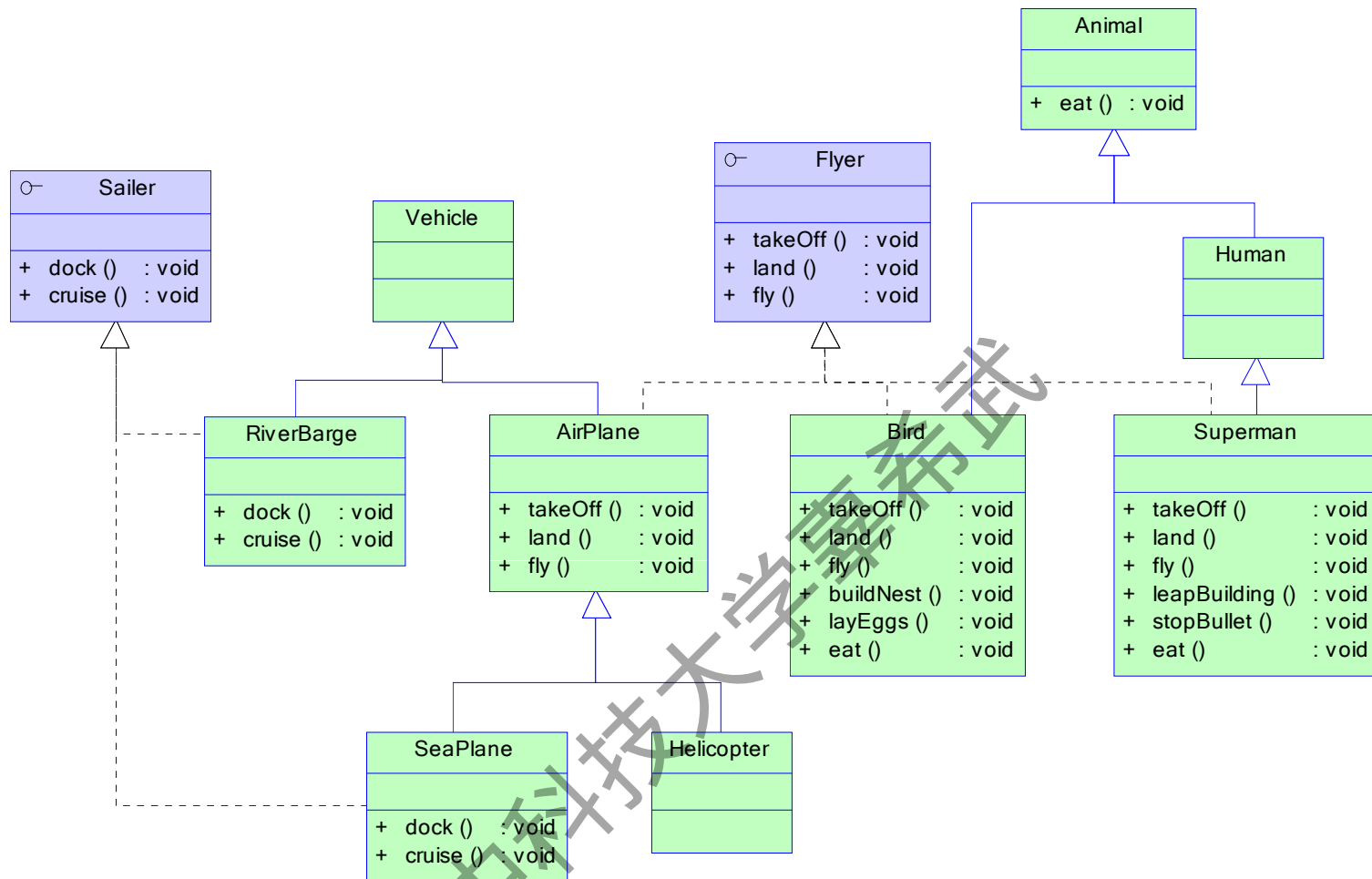


实现关系和继承关系的区别

- 实现关系（接口）：接口通常用于描述一个类的外围能力，而不是核心特征，例如Bird类可以实现Flyer接口，而Flyer可以应用于其他不相关的对象。类与接口之间的实现关系是-able或者can do的关系。
- 继承关系：父类（特别是抽象类）定义了它的后代的核心特征。例如Person类包含了Student类的核心特征。派生类与父类之间的继承关系是is-a的关系。

华中科技大学

继承与实现



创建型设计模式

华中科技大学

创建型设计模式

- ◆工厂模式
- ◆抽象工厂模式
- ◆建造者模式
- ◆单件模式
- ◆原型模式

工厂模式的由来

◆ 在面向对象编程中, 常用的方法是用new操作符构造对象实例, 但在有些情况下, new操作符直接生成对象会带来一些问题

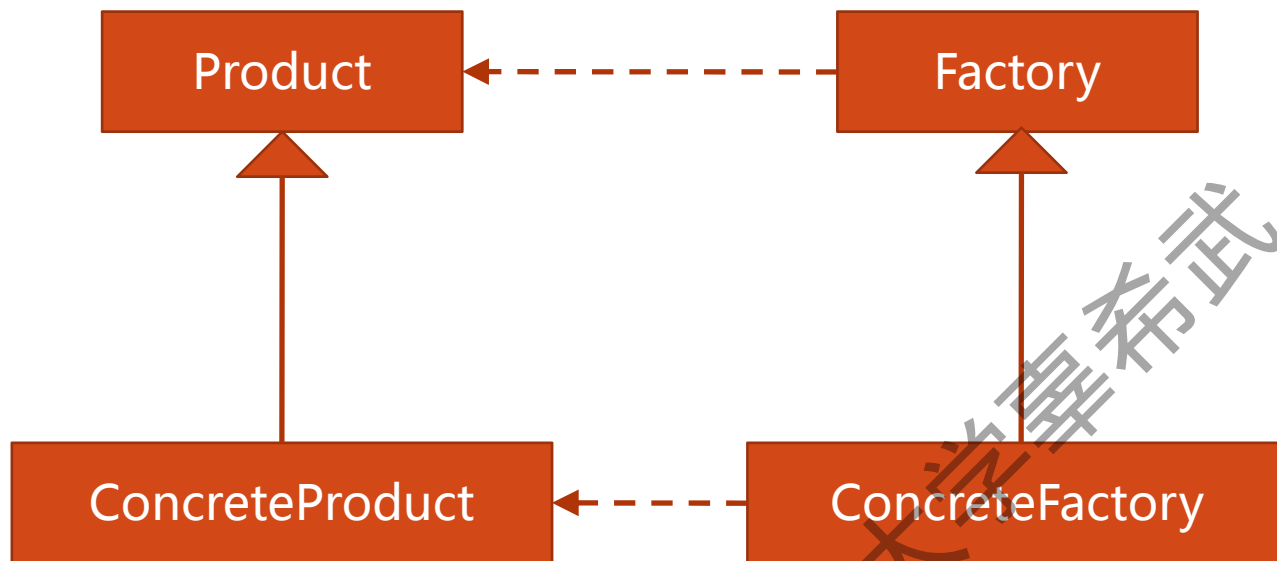
(1) 创建对象之前必须清楚所要创建对象的类信息, 但个别情况下无法达到此要求, 譬如打开一个视频文件需要一个播放器对象, 但是用户可能不知道具体播放器叫什么名字, 需要系统分派给这个视频文件一个合适的播放器, 这种情况下用new运算符并不合适

工厂模式的由来

(2) 许多类型对象的创造需要一系列步骤

- ◆ 需要计算或取得对象的初始设置
- ◆ 需要选择生成哪个子对象实例
- ◆ 在生成需要对象之前必须先生成一些辅助功能对象
- ◆ 在这些情况, 新对象的建立就是一个“过程”, 而不仅仅是一个操作。为了能方便地完成这些复杂的对象创建工作, 可引入工厂模式

工厂模式的结构



工厂模式的参与者

- ◆ Product : 定义工厂方法所创建对象的接口
- ◆ ConcreteProduct : 实现Product接口
- ◆ Factory : 声明工厂方法，返回一个Product类型对象。
Factory也可以定义一个工厂方法的缺省实现，返回一个缺省ConcreteProduct对象
- ◆ Concrete Factory : 重定义工厂方法，返回一个ConcreteProduct实例

工厂模式实例分析

- ✓ 一个日志管理器：设计日志记录类，比如
 - ✓ FileLog
 - ✓ EventLog

```
// AbstractLog类: Product
public abstract class AbstractLog
{
    public void Log();
}
//FileLog: ConcreteProduct
public class FileLog extends AbstractLog{
    public void Log { ...}
}
//EventLog: ConcreteProduct
public class EventLog extends AbstractLog{
    public void Log { ...}
}
```


工厂模式实例分析

```
// LogFactory类: Factory  
public abstract class LogFactory  
{  
    //工厂方法，返回AbstractLog对象  
    public AbstractLog Create();  
}
```

华中科技大学

工厂模式实例分析

```
// EventLogFactory类 : ConcreteFactory
public class EventLogFactory extends LogFactory
{
    public AbstractLog Create() //重新实现工厂方法Create
    {
        return new EventLog(); //创建一个具体的日志对象
    }
}
```

```
// FileLogFactory类:ConcreteFactory
public class FileLogFactory extends LogFactory
{
    public AbstractLog Create() //重新实现工厂方法Create
    {
        return new FileLog(); //创建一个具体的日志对象
    }
}
```

工厂模式客户端程序

```
public class App
{
    public static void main(string[] args)
    {
        LogFactory factory = new FileFactory();
        //FileFactory factory = new FileFactory();

        AbstractLog logger= factory.Create();

        logger.Log();
    }
}
```

✓注意客户代码的特点：

- ✓没有出现一个具体日志类型，除了工厂类型外，全部是抽象数据类型
- ✓没有使用new

客户代码不会随着具体Log类的改变而改变

工厂模式实例分析

- ◆ 客户程序有效避免了具体产品对象和应用程序之间的耦合，客户程序没必要知道它所必须创建的对象的具体类型，只需要知道对象工厂返回一个实现了抽象接口（AbstractLog）的对象就够了
- ◆ 在类内部创建对象通常比直接创建对象更灵活
把创建对象的复杂步骤全部封装到工厂方法Create里
- ◆ 工厂模式通过面向对象的手法，将具体对象的创建工作延迟到子类，提供了一种扩展策略，较好的解决了紧耦合问题

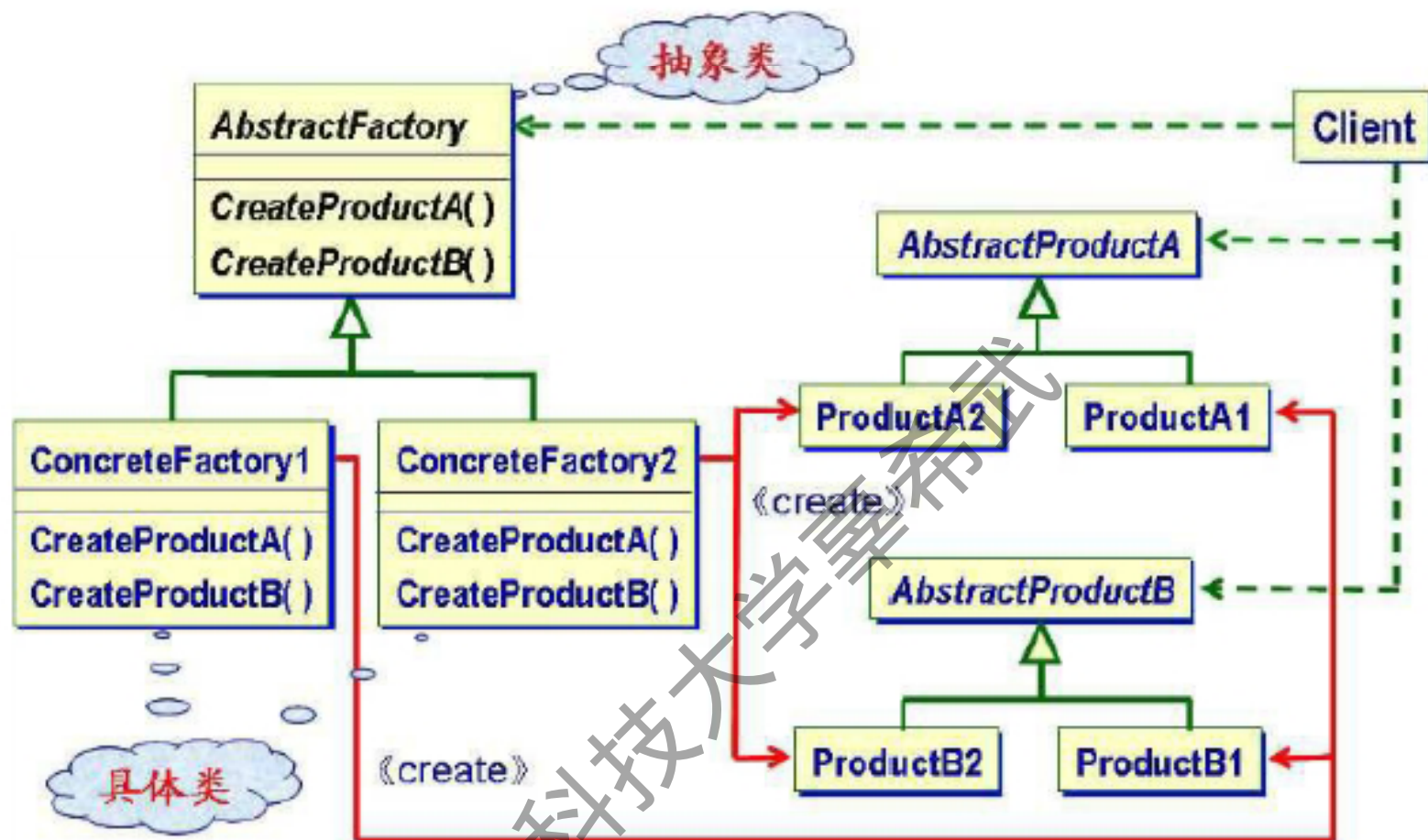
抽象工厂模式的由来

- ◆ 在软件系统中，经常面临“一系列相互依赖对象”的创建工作，由于需求变化，这“一系列相互依赖的对象”也要改变，如何应对这种变化呢？如何像工厂模式一样绕过常规的“new”，提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？
- ◆ 一种说法：可以将这些对象一个个通过工厂模式来创建。但是，既然是一系列相互依赖的对象，它们是有联系的，每个对象都这样解决，如何保证他们的联系呢？

抽象工厂模式的由来

- ◆ 实例：Windows桌面主题，当更换一个桌面主题的时候，系统的开始按钮、任务栏、菜单栏、工具栏等都变了，而且是一起变的，他们的色调都很一致，类似这样的问题如何解决呢？
- ◆ 应用抽象工厂模式，是一种有效的解决途径

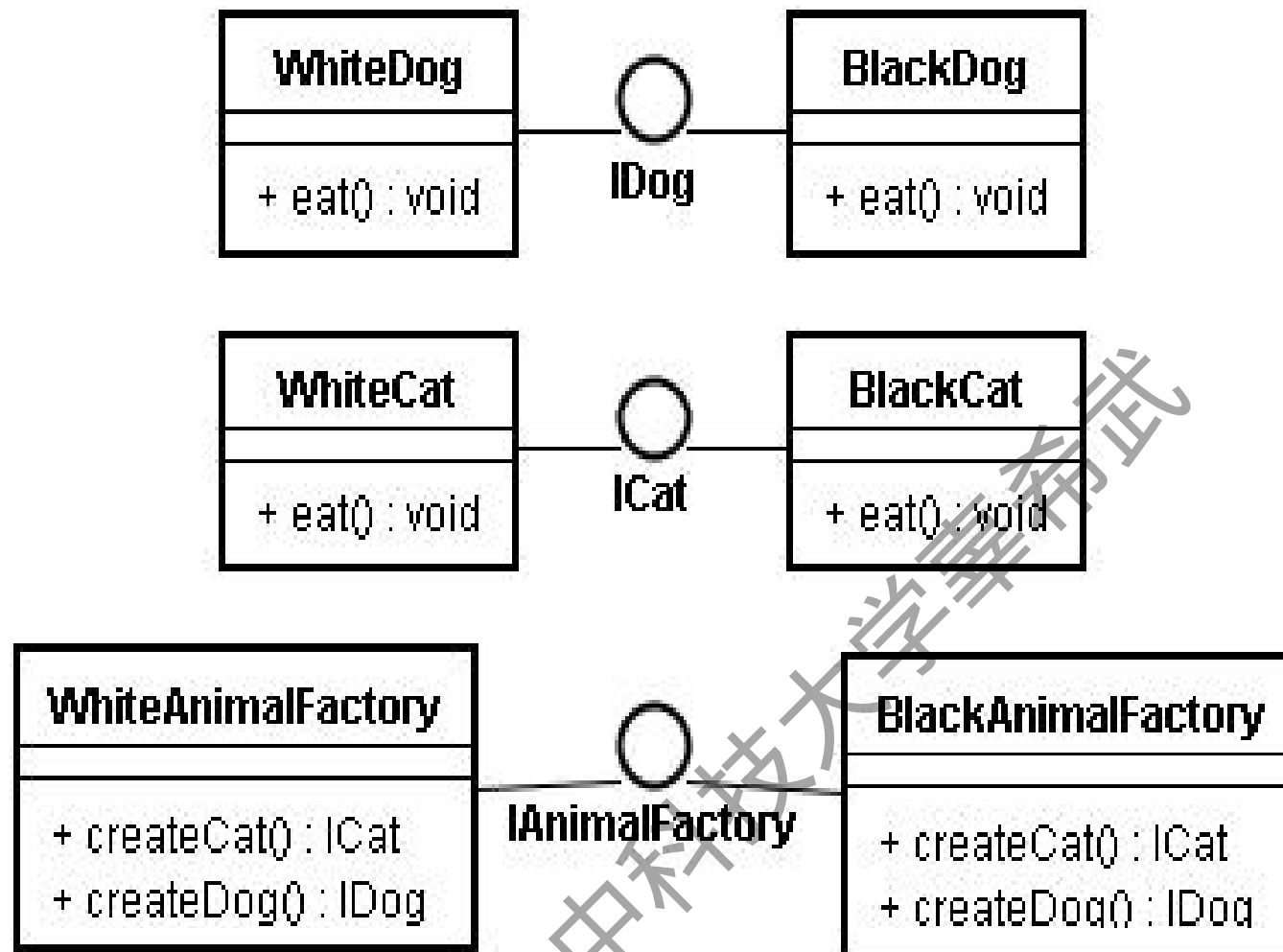
抽象工厂模式的结构



抽象工厂模式的参与者

- ◆ Abstract Factory : 声明创建抽象产品对象的操作接口
- ◆ ConcreteFactory : 实现创建具体对象的操作
- ◆ Abstract Product : 为一类产品对象声明一个接口
- ◆ ConcreteProduct : 定义一个被具体工厂创建的产品对象

抽象工厂模式的示例



AbstractProduct

```
public interface ICat {  
    void eat();  
}
```

```
public interface IDog {  
    void eat();  
}
```

华中科技大学

ConcreteProduct

```
public class Blackcat implements ICat {  
    public void eat() {  
        System.out.println("The black cat is eating!");  
    }  
}  
  
public class WhiteCat implements ICat {  
    public void eat() {  
        System.out.println("The white cat is eating! ");  
    }  
}
```

ConcreteProduct

```
public class BlackDog implements IDog {  
    public void eat() {  
        System.out.println("The black dog is eating");  
    }  
}  
  
public class WhiteDog implements IDog {  
    public void eat() {  
        System.out.println("The white dog is eating!");  
    }  
}
```

AbstractFactory

```
public interface IAnimalFactory {  
    ICat createCat();  
    IDog createDog();  
}
```

华中科技大学 覃希武

ConcreteFactory

```
public class BlackAnimalFactory implements IAnimalFactory {  
    public ICat createCat() {  
        return new BlackCat();    //创建黑色猫  
    }  
    public IDog createDog() {  
        return new BlackDog();    //创建黑色狗  
    }  
}
```

BlackAnimalFactory是具体工厂，创建黑色系列产品

ConcreteFactory

```
public class WhiteAnimalFactory implements IAnimalFactory {  
    public ICat createCat() {  
        return new WhiteCat();           //创建白色猫  
    }  
    public IDog createDog() {  
        return new WhiteDog();           //创建白色狗  
    }  
}
```

WhiteAnimalFactory是具体工厂，创建白色系列产品

Client

```
public static void main(String[] args) {  
    IAnimalFactory blackAnimalFactory = new  
        BlackAnimalFactory();  
    ICat blackCat = blackAnimalFactory.createCat();  
    blackCat.eat();  
    IDog blackDog = blackAnimalFactory.createDog();  
    blackDog.eat();  
    IAnimalFactory whiteAnimalFactory = new  
        WhiteAnimalFactory();  
    ICat whiteCat = whiteAnimalFactory.createCat();  
    whiteCat.eat();  
    IDog whiteDog = whiteAnimalFactory.createDog();  
    whiteDog.eat();  
}
```


工厂模式和抽象工厂模式的区别

工厂方法模式	抽象工厂模式
针对的是一个产品等级结构	针对的是面向多个产品等级结构
一个抽象产品类	多个抽象产品类
可以派生出多个具体产品类	每个抽象产品类可以派生出多个具体产品类
一个抽象工厂类，可以派生出多个具体工厂类	一个抽象工厂类，可以派生出多个具体工厂类
每个具体工厂类只能创建一个具体产品类的实例	每个具体工厂类可以创建多个具体产品类的实例

单件（Singleton）模式的由来

- ◆ 单件模式的实例较为普遍:
 - ◆ 系统中只能有一个窗口管理器
 - ◆ 系统中只能有一个文件系统
 - ◆ 一个数字滤波器只能有一个A/D转换器
 - ◆ 一个会计系统只能专用于一个公司
 - ◆ 一个系统只有一个日志对象
 - ◆ 一个系统只有一个注册表
 - ◆ 一个系统只有一个线程池
 - ◆ . . .

单件模式的由来

- ◆ 如何才能保证一个类只有一个实例并且这个实例易于被访问呢？一个全局变量使得一个对象可以被访问，但它不能防止你实例化多个对象

单件模式的由来

- ◆ 如何做到一个类只有一个实例
- ◆ 如果一个类的构造函数是公有的，你是无法控制该类的使用者多次new出该类的多个实例
- ◆ 如果一个类的构造函数是保护的，无法控制该类的派生类多次new出该类的多个实例

华中科技大学

单件模式的由来

- ◆ 因此该类的构造函数必须是私有的

```
public class MyClass{  
    private MyClass() { }  
}
```

- ◆ 因为只有MyClass类的实例才能调用构造函数，但是又没有其他类能够实例化MyClass，因此我们得不到这样的实例。这似乎是一个“鸡生蛋、蛋生鸡”的问题

单件模式的由来

- ◆ 一个更好的办法是，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可以被创建，并且它可以提供一个访问该实例的方法
- ◆ 如何做到？利用静态变量和静态方法

华中科技大学 辜希武

单件模式的由来

利用私有静态变量来记录
唯一实例

```
public class Singleton{
```

```
    private static Singleton instance = null;
```

私有构造函数

```
    private Singleton() { }
```

```
    public static Singleton getInstance(){
```

公有静态方法实例化对象
并返回实例

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

如果**instance**为空，则实例化
一个对象
并赋值给静态变量**instance**
否则直接返回**instance**
因此保证了构造函数只会被调
用一次

```
    //其它的实例变量、实例方法
```

```
    public void someMethod( ) {}
```

```
}
```

单件模式的由来

```
public class Client{  
    public static void main(String[] args){  
        Singleton instance1 = Singleton.getInstance();  
        Singleton instance2 = Singleton.getInstance();  
  
        System.out.println(instance1 == instance2); //true  
  
        instance1.someMethod();  
    }  
}
```

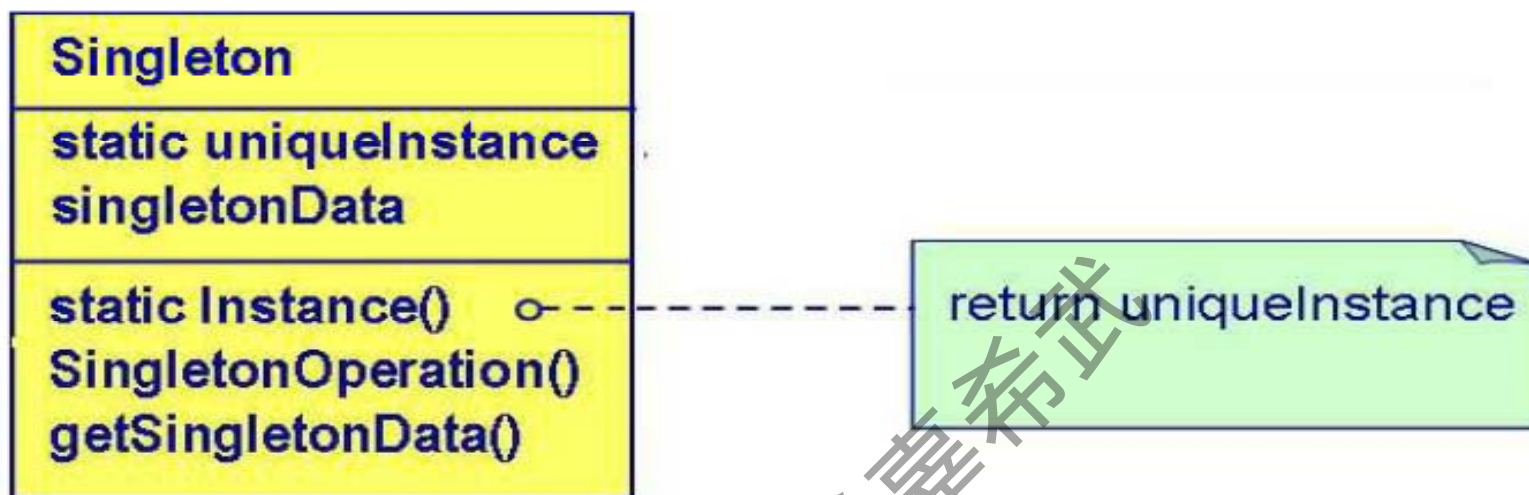

单件模式的意图和适用性

◆**意图**：保证一个类仅有一个实例,并提供一个全局访问点

◆**适用场合**：

◆当类只能有一个实例而且用户可以从一个众所周知的访问点访问它时

单件模式的结构



- ◆ Singleton 被调用的单件对象
 - ◆ 定义一个Instance操作，允许客户访问它的唯一实例。Instance是一个类操作（一个静态成员函数）
 - ◆ 可负责创建它自己的唯一实例

单件模式的进一步分析

- ◆ 有人会认为，利用公有静态变量可以达到同样的效果

```
public Singleton{  
    public static Singleton instance = new Singleton();  
    private Singleton() {}  
}
```

在客户程序里通过Singleton.instance不就可以得到唯一的实例？

- ◆ 但这样的缺点是：在Singleton类被ClassLoader装载进JVM时，Singleton的唯一实例就被创建了。假设这个对象非常耗费资源，而该实例并没有马上被使用（甚至由于程序里某个条件分支导致该对象根本没使用）。这就浪费了资源。这叫急切实例化(eagerly instantiate)
- ◆ 另外一个缺点是：公有静态变量过多容易污染名字空间

单件模式的进一步分析

```
public class Client{  
    public static void main(String[] args){  
        Singleton instance1 = Singleton.getInstance();  
        Singleton instance2 = Singleton.getInstance();  
  
        System.out.println(instance1 == instance2); //true  
  
        instance1.someMethod();  
    }  
}
```

//只有需要实例时，才调用getInstance方法获得实例，这叫
//延迟实例化(lazy instantiate)

单件模式的进一步分析：多线程问题

```
public class ClientThread implements Runnable{  
    public void run(){  
        Singleton instance = Singleton.getInstance();  
    }  
}
```

```
public class Client{  
    public static void main(String[] args){  
        new Thread(new ClientThread()).start();  
        new Thread(new ClientThread()).start();  
    }  
}
```

**//假设有二个并发线程，都调用Singleton.getInstance
//有可能会产生二个实例吗？**

单件模式的进一步分析：多线程问题

线程一

```
public static Singleton  
getInstance(){
```

```
if ( instance == null ) {
```

```
instance = new Singleton();
```

```
return instance;
```

```
}  
}
```

线程二

```
public static Singleton  
getInstance(){
```

```
if ( instance == null ) {
```

```
instance = new Singleton();
```

```
return instance;
```

```
}  
}
```

instance值

null

null

null

null

<object>

<object>

单件模式的进一步分析：多线程

```
public class Singleton{  
  
    private static Singleton instance = null;  
  
    private Singleton() { }  
  
    public static Synchronized Singleton getInstance(){  
  
        if(instance == null){  
            instance = new Singleton();  
        }  
        return instance;  
    }  
    //其它的实例变量、实例方法  
    public void someMethod( ) {}  
}
```

增加**Synchronized**关键字，迫使每个线程在进入这个方法前，必须等候其它线程离开这个方法。即不可能同时二个线程进入这个方法

单件模式的效果分析

- ◆ 对唯一实例的受控访问：因为Singleton类封装唯一实例，所以它可以严格的控制客户怎样以及何时访问它（延迟实例化）
- ◆ 缩小名空间：Singleton模式是对全局变量的一种改进。它避免了那些存储唯一实例的全局变量污染名空间

创建型设计模式总结

- ◆ 工厂模式：定义一个创建对象的接口，由子类决定要实例化的具体类，工厂方法让类把实例化推迟到子类
- ◆ 抽象工厂模式：提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类
- ◆ 单件模式：确保一个类只有一个实例，并提供全局访问点

结构型设计模式

华中科技大学

结构型设计模式

- ◆ 结构型模式讨论的是类和对象的结构，它采用继承机制来组合接口或实现（**类结构型模式**），或者通过组合一些对象来实现新的功能（**对象结构型模式**）

结构型模式的主要内容

- 适配器模式
- 外观模式
- 装饰模式
- 桥接模式
- 享元模式
- 代理模式
- 组合模式

华中科技大学

适配器模式的由来

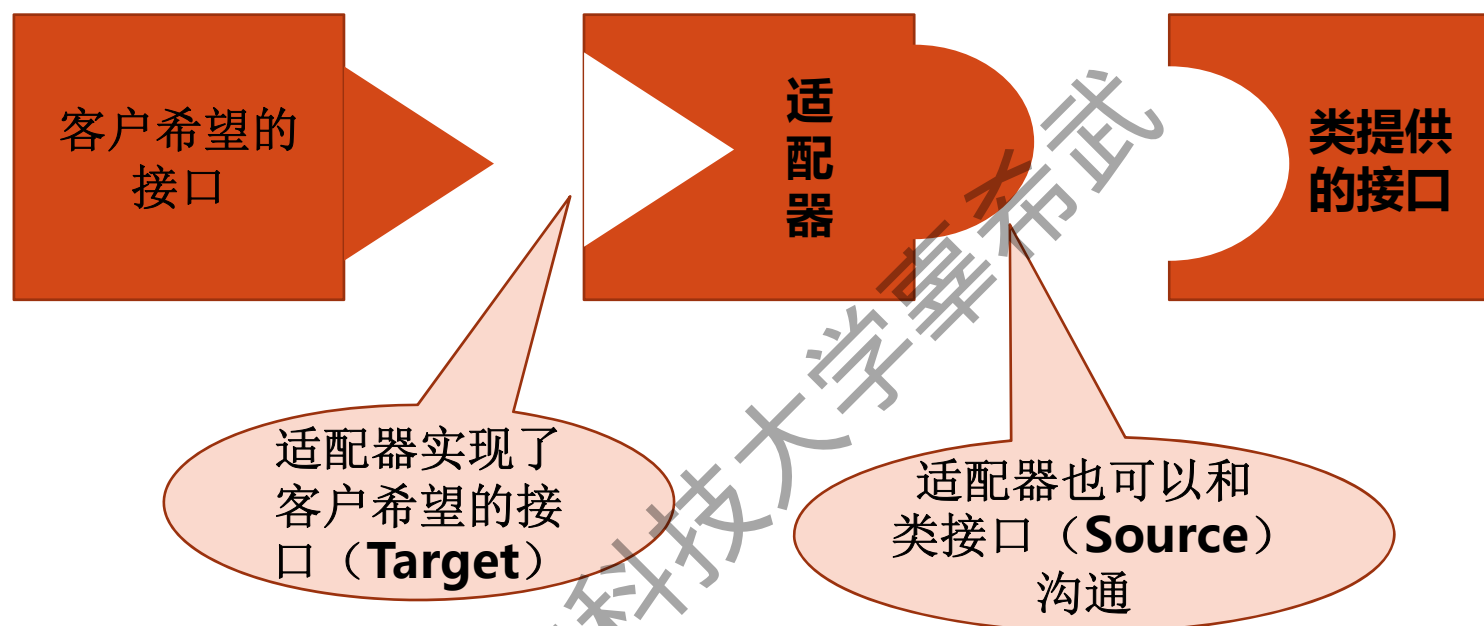
- 将一个类的接口，转换成客户期望的另一个接口，适配器让原本接口不兼容的类可以一起工作



- 改变客户代码或改变类的代码，不是好办法

适配器模式的由来

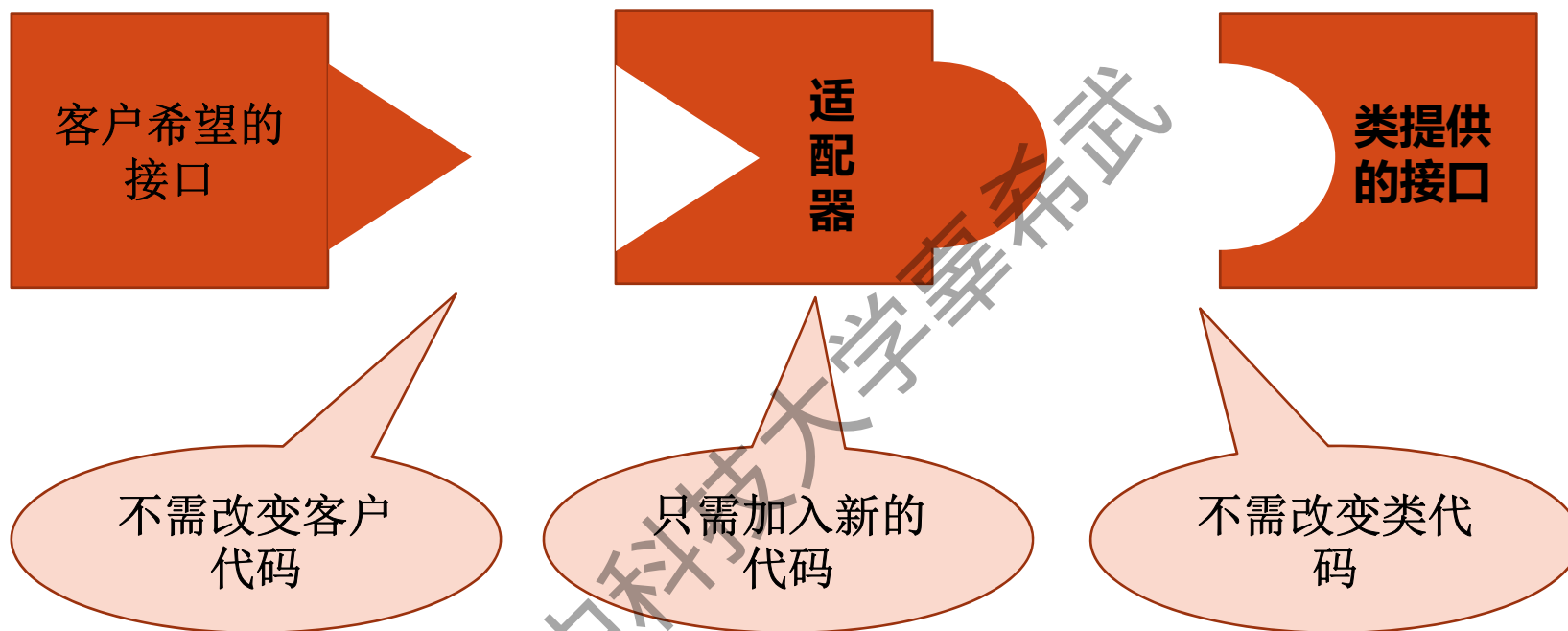
- 写一个类，将类提供的接口变成客户希望的接口
- 类提供的接口叫源接口（Source Interface），客户希望的接口叫目标接口（Target Interface）



适配器模式的由来

□ 适配器完成了一个看似不可能的任务：把一个方块放进圆洞里

✓对扩展是开放的，对修改是关闭的



适配器模式的例子

- 如果它走路像只鸭子，叫起来像只鸭子，那么它必定可能是一只鸭子包装了鸭子适配器的火鸡

```
public interface Duck{  
    public void quack();    //呱呱叫  
    public void fly();      //飞  
}  
  
public interface Turkey{  
    public void gobble();    //咯咯叫  
    public void fly();      //飞  
}
```


适配器模式的例子

```
public class GreenDuck implements Duck{
    public void quack(){ System.out.println("Quack");}
    public void fly(){ System.out.println("I am flying");}
}

public class WildTurkey implements Turkey{
    public void gobble(){ System.out.println("Gobble gobble");}
    public void fly(){ System.out.println("I am flying");}
}
```

现在想用火鸡对象来冒充鸭子对象，由于接口不匹配，需要写个适配器。

注意：目标接口是Duck，源接口是Turkey

适配器模式的例子

适配器实现目标接口

```
public class TurkeyToDuckAdapter implements Duck{
```

```
    Turkey turkey;
```

包含一个被适配接口的引用,利用对象组合。

```
    public TurkeyToDuckAdapter (Turkey turkey){
```

```
        this.turkey = turkey;
```

在构造函数中传入被适配的接口引用

```
    }
```

```
    public void quack() {
```

```
        turkey.gobble();
```

实现**Duck**接口的**quack**方法,调用被适配接口对象的**gobble**方法

```
    }
```

```
    public void fly() {
```

```
        for(int i = 0; i < 5; i++)
```

```
            turkey.fly();
```

实现**Duck**接口的**fly**方法,火鸡飞行距离比鸭子短,因此连续调用**5**次火鸡的飞行

```
    }
```

```
}
```

适配器模式的例子

```
public class Client {  
    static void testDuck(Duck duck){  
        duck.quack();  
        duck.fly();  
    }  
    public static void main(String[] args){  
        GreenDuck duck = new GreenDuck();  
        testDuck(duck);  
        WildTuykey turkey = new WildTurkey();  
        Duck TurkeyToDuckAdapter = new TurkeyToDuckAdapter  
        (turkey);  
        testDuck(tukkeyDuckAdapter);  
    }  
}
```

客户程序需要**Duck**接口

传入一个真正的实现**Duck**接口的鸭子对象

把火鸡包装进适配器，使它看起来像鸭子

客户程序根本不知道，这其实是假装成鸭子的火鸡

适配器模式使用过程

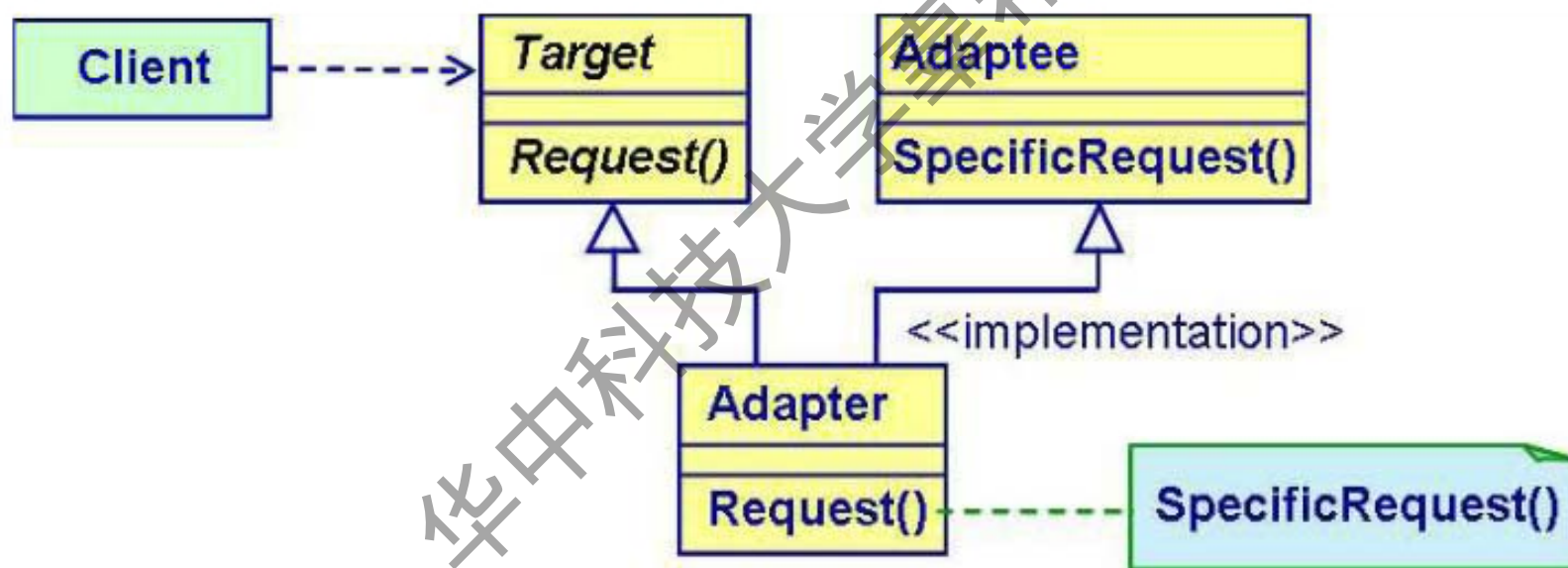
- 客户通过目标接口调用适配器的方法对适配器发出请求
- 适配器使用被适配者接口把请求转换成被适配者的一个或者多个调用接口
- 客户接收到调用的结果，但并未察觉这一切是适配器在起转换作用

适配器模式的意图和适用性

- **意图**：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的类可以一起工作
- **适用场合**：
 - 使用一个已经存在的类，而它的接口不符合要求
 - 创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作
- 适配器模式分为类适配器和对象适配器

类适配器

- 用一个具体的Adapter类对Adaptee和Target进行匹配，Adapter类多重继承Adaptee和Target类
- Adapter可重定义Adaptee的部分行为，因为Adapter是Adaptee的一个子类



从这个例子可以看出利用接口
同样可以达到多重继承的效果

类适配器模式示例

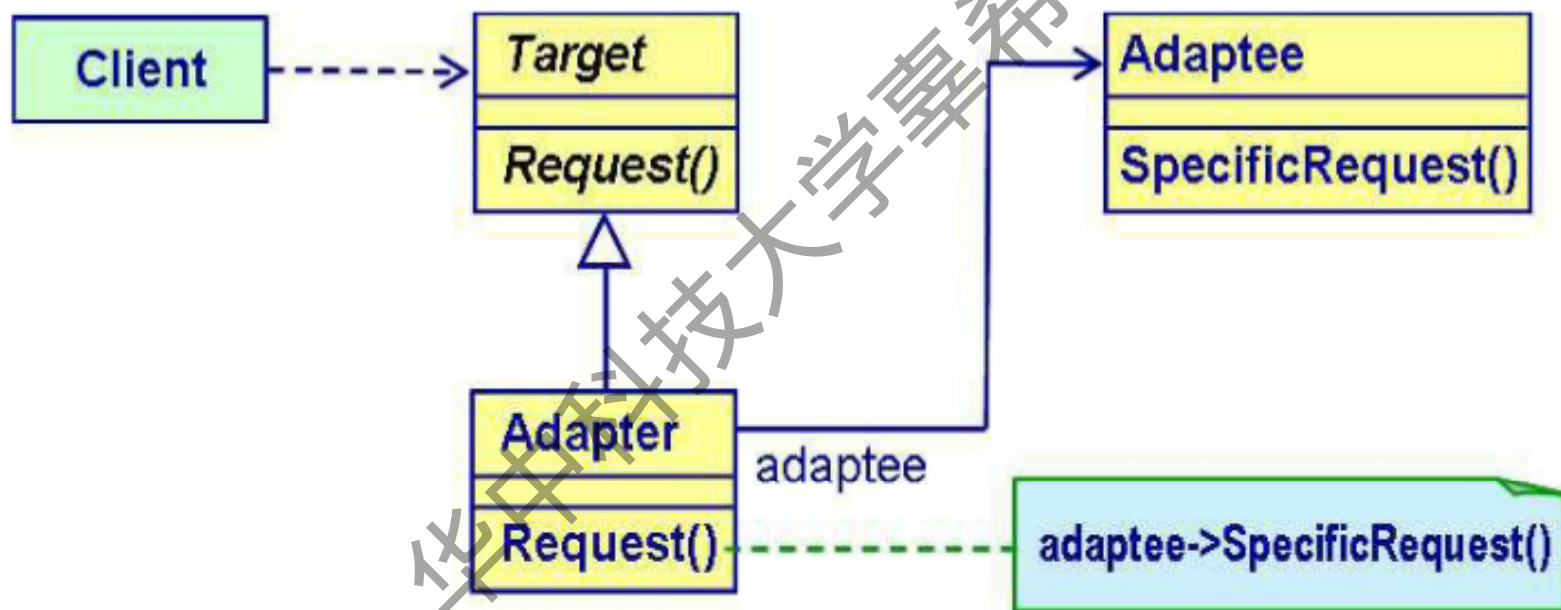
```
//Target : 定义Client使用的与特定领域相关的接口  
public interface Target  
{ void request(); }
```

```
//Adaptee:现在需要适配的已经存在的接口  
public class Adaptee  
{ public void specificRequest(){} }
```

```
//Adapter:对Adaptee 的接口与Target接口进行适配  
public class Adapter extends Adaptee implements Target//通过类的继承  
{  
    public void request() {  
        super. specificRequest();  
    }  
}
```

对象适配器

- 允许一个Adapter与多个Adaptee同时工作，即和Adaptee本身以及它的所有子类（如果有子类的话）同时工作。Adapter可以一次给所有的Adaptee添加功能
- 使用组合，不仅可以适配某个类，也可以适配该类的任何子类



对象适配器模式示例

//Target : 定义Client使用的与特定领域相关的接口
public interface Target
{ void request(); }

//Adaptee:现在需要适配的已经存在的接口
public class Adaptee
{ public void specificRequest(){} }

//Adapter:对Adaptee 的接口与Target接口进行适配
public class Adapter implements Target
{
 public Adapter(Adaptee adaptee)
 { super(); this.adaptee = adaptee; }
 public void request()
 { adaptee.specificRequest(); }
 private Adaptee adaptee; //通过对象组合
}

Adaptee的子类也可以被适配

适配器模式效果分析

□ 优点

- 方便设计者自由定义接口，不用担心匹配问题

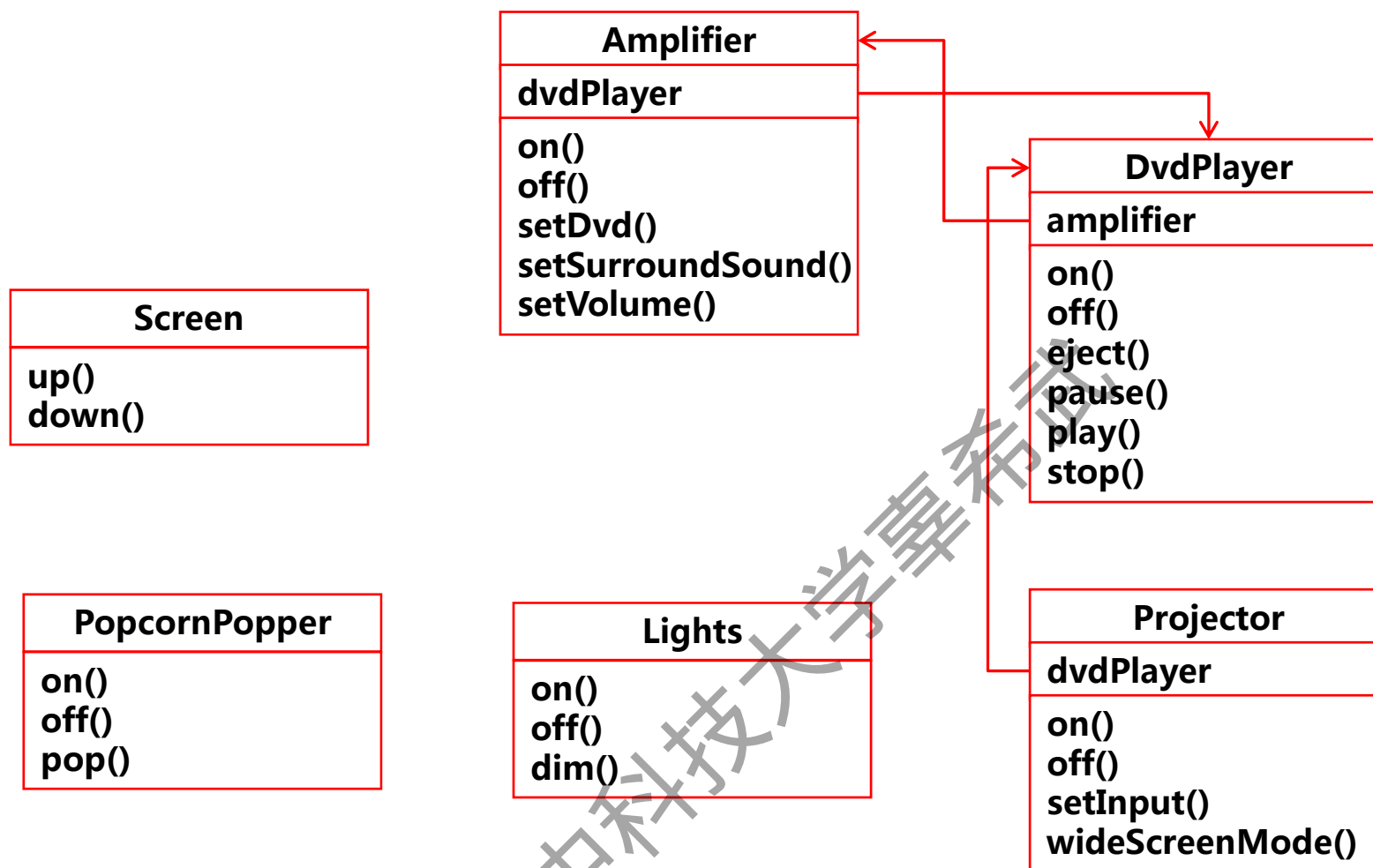
□ 缺点

- 类适配器属于静态结构，由于只能单继承，所以不适用于多种不同的源适配到同一个目标

外观模式（Facade pattern）的由来

- ◆ 对于一个复杂系统提供的一群接口，外观模式提供了一个简单、高层的接口，让子系统更容易使用
- ◆ 考虑一个家庭影院系统，包括如下设备：
 - ◆ DVD播放器(DvdPlayer)
 - ◆ 功放 (Amplifier)
 - ◆ 投影仪(Projector)
 - ◆ 屏幕(Screen)
 - ◆ 灯光(Lights)
 - ◆ 爆米花机(PopCornPopper)

外观模式（Facade pattern）的由来



外观模式（Facade pattern）的由来

◆ 看一部DVD需要以下复杂操作

```
popper.on();  
popper.pop();
```

```
lights.dim(10);
```

```
screen.down();
```

```
projector.on();  
projector.setInput(dvd);  
projector.wideScreenMode();
```

```
amp.on()  
amp.setDvd(dvd)  
amp.setSurroundSound();  
amp.setVolume(5);
```

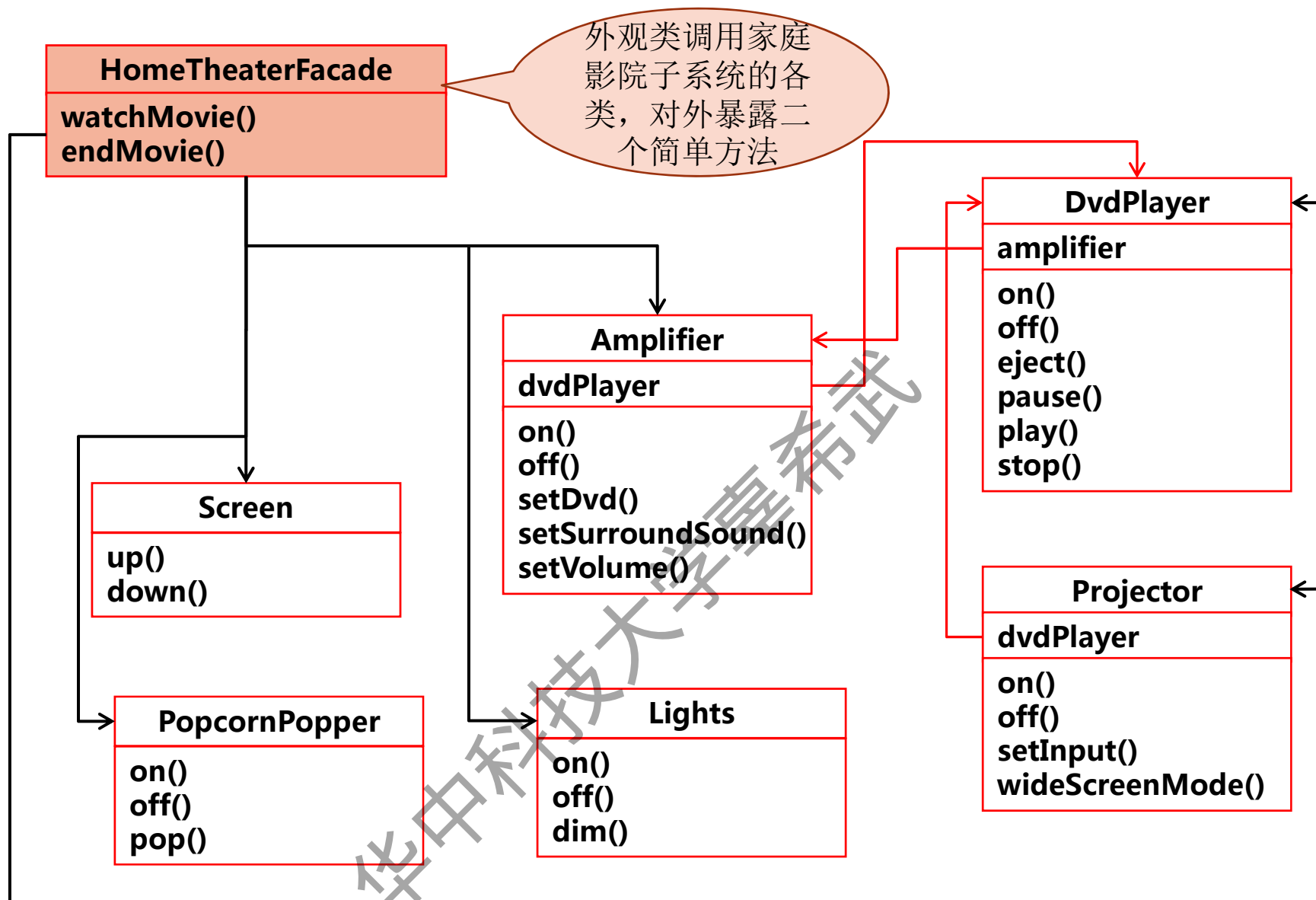
```
dvd.on();  
dvd.play(movie);
```

涉及到**6**个类的
不同方法

看一部电影这么复杂？
要是有个遥控器，按
一个“**WatchMovie**”
就帮我搞定这些事多
好

外观模式来了！，通过实现一个更
合理接口的外观类，可以使得家庭
影院这么一个复杂系统更容易使用

外观模式（Facade pattern）的由来



构造家庭影院的外观

```
public class HomeTheaterFacade{  
    Amplifier amp;  
    DvdPlayer dvd;  
    Projector projector;  
    Lights lights;  
    Screen screen;  
    PopcornPopper popper;  
    public HomeTheaterFacade(Amplifier amp, DvdPlayer dvd  
                             Projector projector, Lights lights  
                             Screen screen , PopcornPopper popper){  
  
        this.amp =amp;  
        this.dvd = dvd;  
        this.projector = projector;  
        this.lights = lights;  
        this.screen = screen;  
        this.popper = popper;  
    }  
}
```

这就是对象组合，会用到的子系统组件全都在这里

子系统中每一个组件的引用全部传入外观构造函数，然后把它们赋值给相应的引用变量

构造家庭影院的外观

```
public class HomeTheaterFacade{  
    public void watchMovie(String movie){  
        popper.on();  
        popper.pop();  
        lights.dim(10);  
        screen.down();  
        projector.on();  
        projector.setInput(dvd);  
        projector.wideScreenMode();  
        amp.on()  
        amp.setDvd(dvd)  
        amp.setSurroundSound();  
        amp.setVolume(5);  
        dvd.on();  
        dvd.play(movie);  
    }  
}
```

watchMovie将之前手动处理的任务依次处理。注意：每项任务都是委托给子系统中相应的组件处理的

华中科技大学

构造家庭影院的外观

```
public class HomeTheaterFacade{  
    public void endMovie(){  
        popper.off();  
  
        lights.on;  
        screen.up();  
        projector.on();  
        projector.off();  
  
        amp.off()  
        dvd.stop();  
        dvd.eject();  
        dvd.off();  
    }  
}
```

endMovie负责关闭一切。
每项任务也都是委托给子系统
中相应的组件处理的

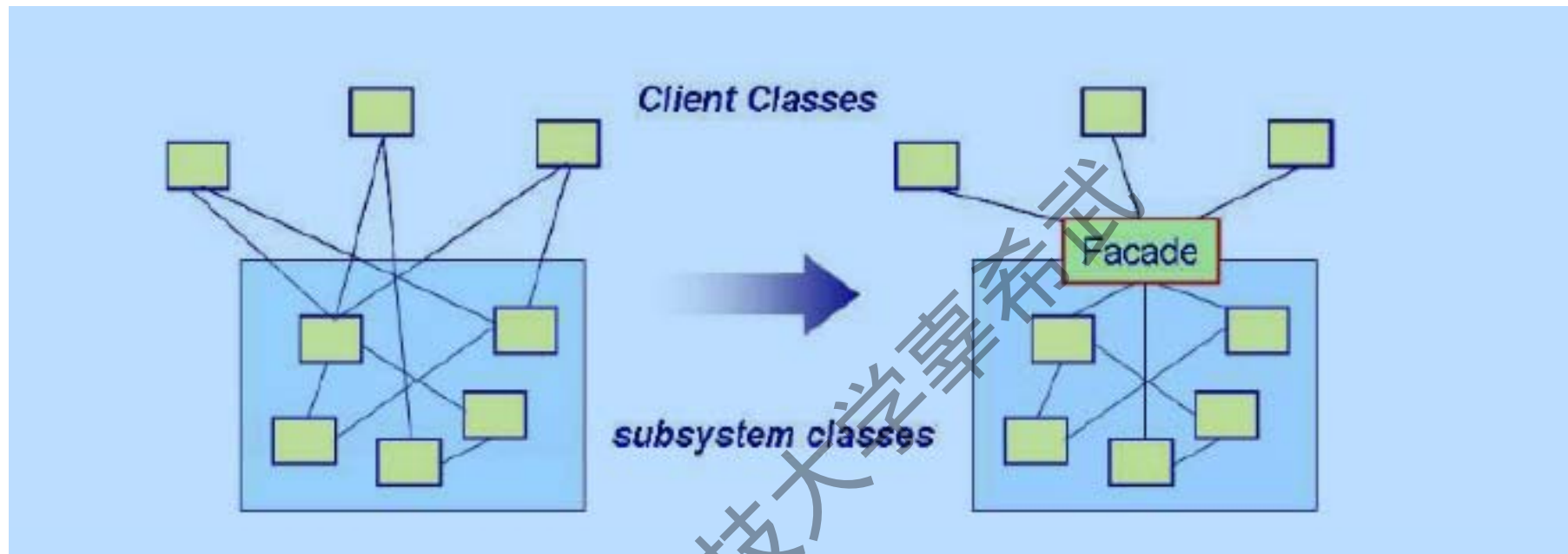
华中科技大学

现在用轻松的方式（通过外观观赏电影）

```
public class Client{  
    public static void main(String[] args){  
  
        //在这里实例化系统组件amp, dvd, ...  
  
        HomeTheaterFacade homeTheater =  
            new HomeTheaterFacade(amp,dvd,projector,  
                lights, screen, popper);  
  
        homeTheater.watchMovie("2012");  
        homeTheater.endMovie();  
    }  
}
```

在上面这个例子里，客户直接建立了外观对象。但实际应用中，某个外观对象会直接指派给客户使用，而不需要由用户自己创建外观对象

外观（Facade）模式的由来



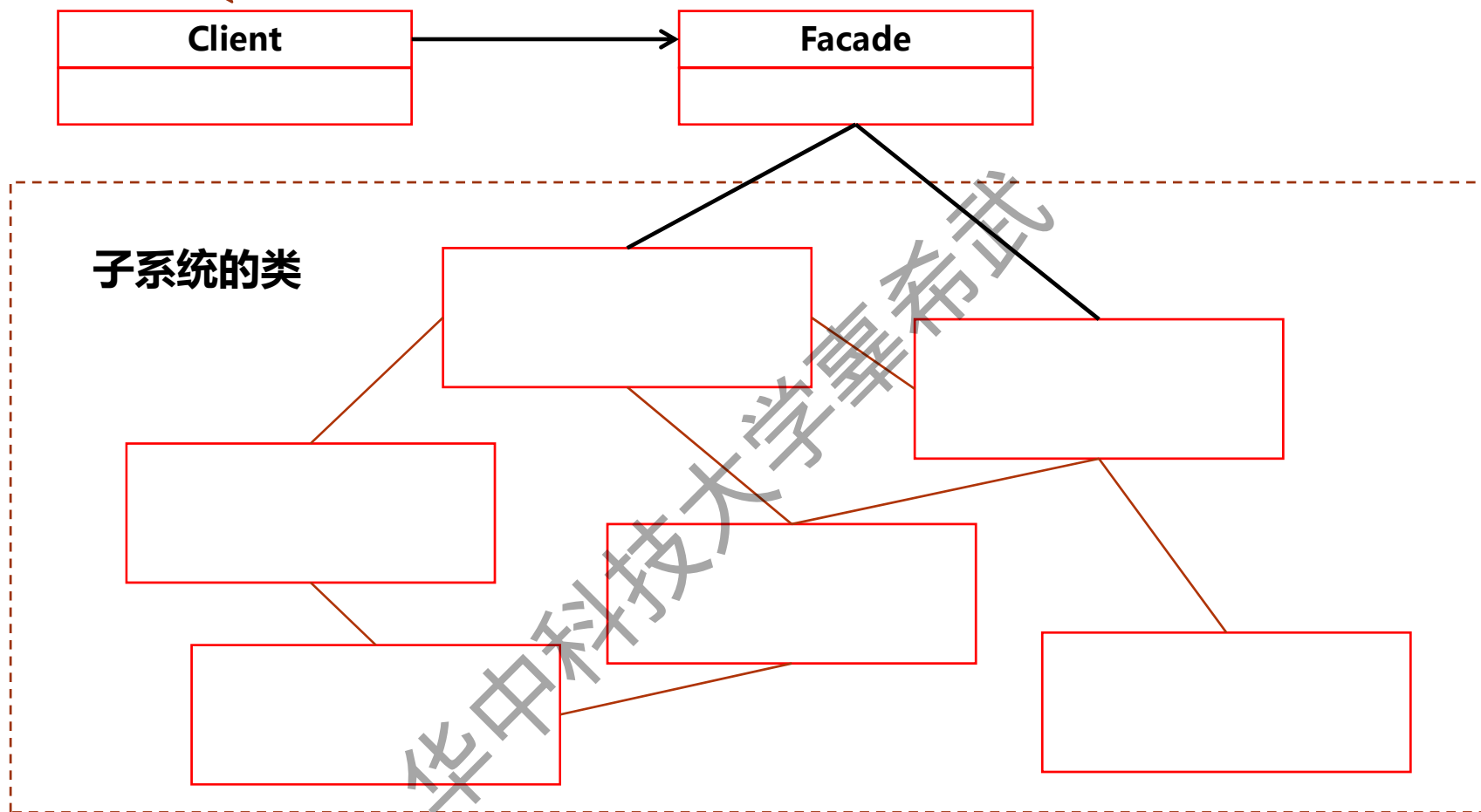
外观（Facade）模式的意图和适用性

- 提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用
- 解除客户程序与子系统具体实现部分的依赖性，避免了客户与子系统的紧耦合，有利于移植和更改
- 当需要构建层次结构的子系统时，使用Facade模式定义每层的入口点。如果子系统间相互依赖，他们只需通过Facade进行通讯
- 外观模式的本质是让接口变得更简单

外观（Facade）模式的结构

快乐的客户只依赖于简单的外观

外观实现了统一、简单的接口，易于使用



外观（**Facade**）模式的参与者

□ Facade

- 知道哪些子系统类负责处理请求
- 将客户的请求代理给适当的子系统对象

□ Subsystem Classes

- 实现子系统的功能
- 处理由Facade 对象指派的任务
- 没有Facade的任何相关信息

外观（Facade）模式的效果分析

- ❑ 简化接口：对客户端屏蔽子系统组件，减少客户端使用对象数目
- ❑ 将客户从子系统中解耦：实现了子系统与客户之间松耦合的关系，使得子系统组件的变化不会影响到客户
- ❑ 不限制客户应用子系统类
 - ❑ 外观只是提供客户更直接的操作，并未将原来的子系统同客户分隔开来，客户除了使用外观以外，还是可以使用原来的子系统类
- ❑ 一个子系统可以实现多个外观

装饰（Decorator）模式的由来

- ❑ 动态给对象添加额外职责。比如：一幅画有没有画框都可以挂在墙上，画是被装饰者。在挂在墙上之前，画可以被蒙上玻璃，装到框子里，玻璃画框就是装饰
- ❑ 不改变接口，但加入责任。Decorator提供了一种给类增加职责的方法，不是通过继承，而是通过对象组合实现的
- ❑ 就增加功能来说，Decorator模式相比生成子类（继承方式）更为灵活。
- ❑ 一个新的原则：多用组合，少用继承。利用组合、委托同样可以达到继承的目的

- Java、C#、SmallTalk等单继承语言在描述多继承的对象时，常常通过对象成员委托（代理）实现多继承。

```
class A{  
    public void f( ) {}  
}
```

```
class B{  
    public void g( ) {}  
}
```

如果需要实现一个类，同时具有类A和类B的行为，但又不能多继承怎么办（如JAVA）？

采用对象组合方式，继承一个类，将另外一个类的对象作为数据成员

```
class C extends A{  
    B b = new B();    //B类行为的代理  
    public void g( ) { //定义一个同名的g函数，但其功能  
        b.g( );       //委托对象b完成(通过调用b.g())，因此  
    }                 //C的g的行为与B的g完全一致  
};
```

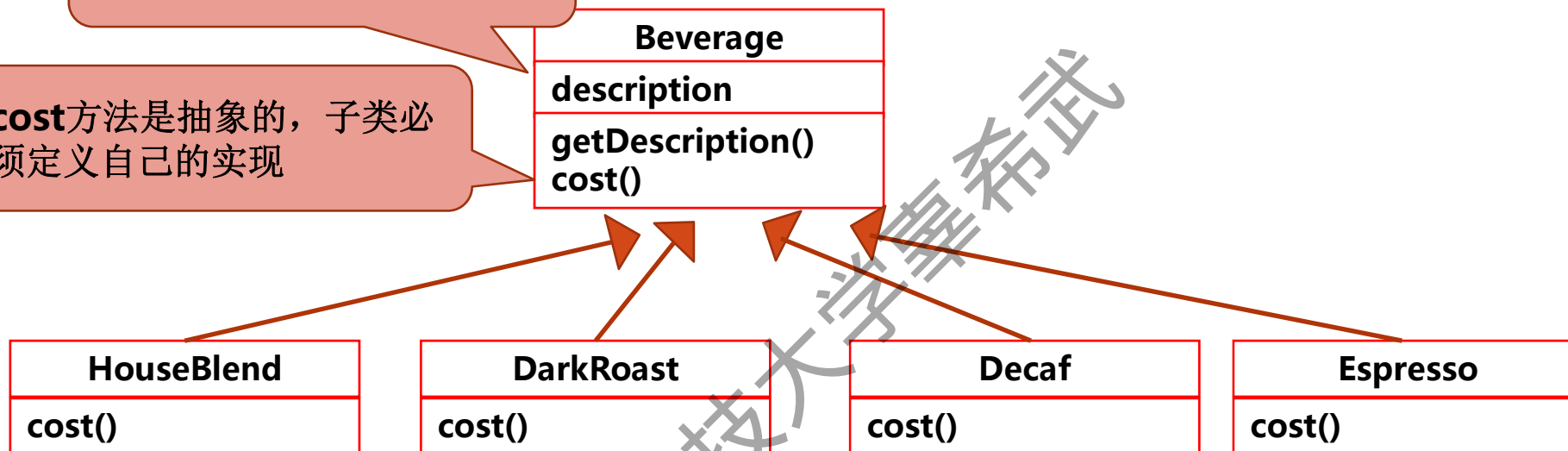
//这样C就具有A的行为f和B的行为g,达到了多重继承的效果

装饰（Decorator）模式的由来

□ 假设要实现一个星巴兹(Starbuzz)咖啡店的饮料类Beverage

Beverage是抽象类，店里所有饮料必须继承它

cost方法是抽象的，子类必须定义自己的实现



每个子类实现**cost**返回饮料的价钱

装饰（Decorator）模式的由来

- ❑ 购买咖啡时，可以在其中加入调料，例如牛奶(Milk)，豆浆(Soy),摩卡(Mocha). 星巴兹会根据加入的调料收取不同的费用
- ❑ 第一个尝试: 每加入一种新的调料，就派生一个新的子类，每个子类根据加入的调料实现cost方法
- ❑ 很明显，星巴兹为自己制造了一个维护的噩梦
 - ❑ 根据加入调料的不同组合，派生子类的数目呈组合爆炸
 - ❑ 如果调料价格变化，必须修改每个派生类的cost方法

装饰（Decorator）模式的由来

□ 能不能从基类下手，利用实例变量+继承，来追踪这些调料？

cost不再是抽象方法，而是计算要加入的各种调料的价钱

Beverage

description
milk
soy
mocha

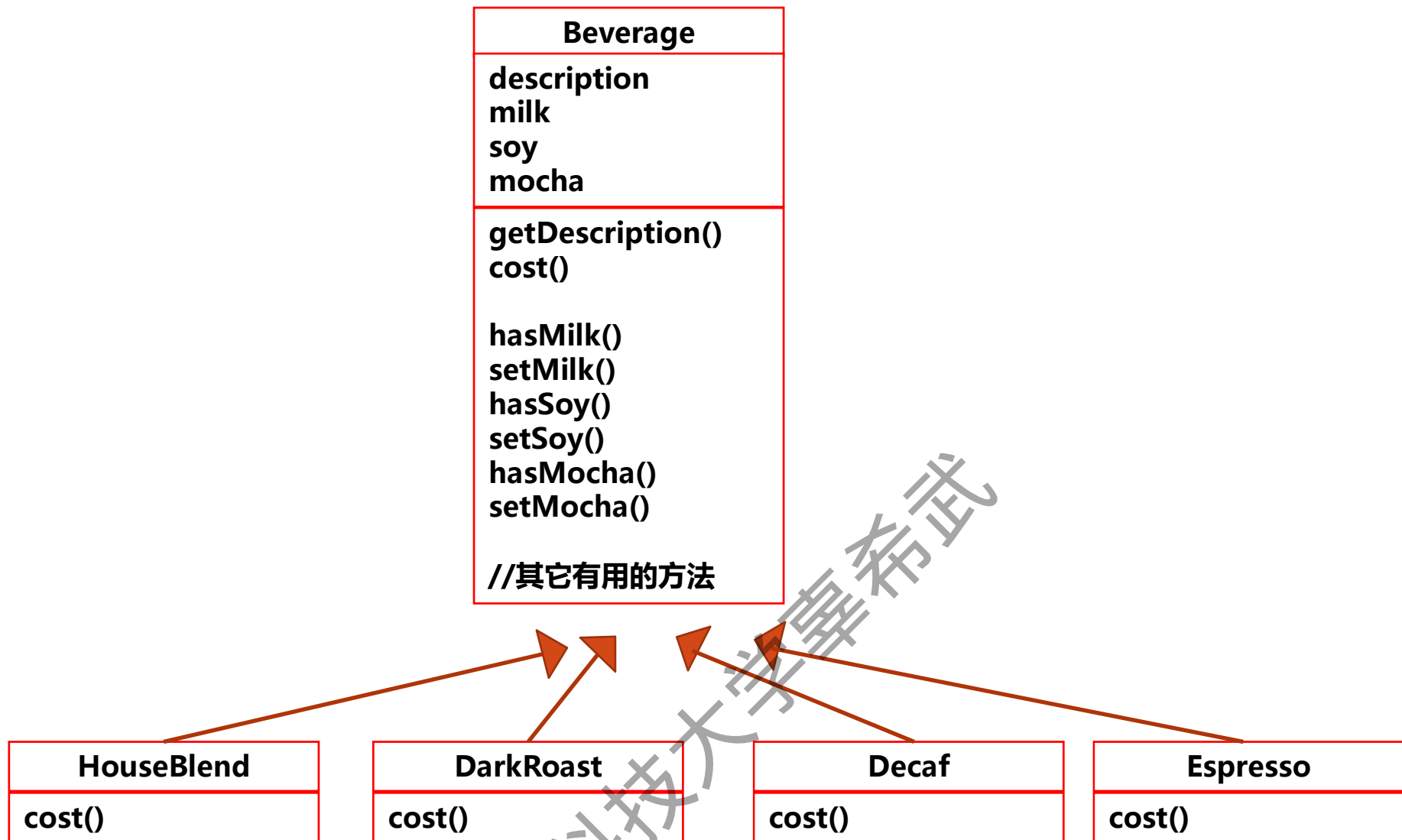
getDescription()
cost()

hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()

//其它有用的方法

各种调料的布尔值

取得和设置调料的布尔值



子类的**cost**方法需要计算该饮料的价钱，然后通过调用超类的**cost**实现，加入调料的价钱

改进的Beverage实现

```
public class Beverage{  
    //为milkCost、soyCost、mochaCost声明实例变量  
    //为milk、soy、mocha声明getter、setter方法  
    public double cost(){  
        float condimentCost = 0.0;  
        if(hasMilk()) condimentCost += milkCost;  
        if(hasSoy()) condimentCost += soyCost;  
        if(hasMocha()) condimentCost += mochaCost;  
        return condimentCost;  
    }  
}  
  
public class DarkRoast extends Beverage{  
    public DarkRoast(){ description = "Most excellent Dark Roast";}   
    public double cost(){  
        return 1.99 + super.cost();  
    }  
}
```

改进的Beverage实现存在的问题

- ❑ 调料价钱改变会使我们更改现有Beverage代码
- ❑ 一旦出现新的调料，我们就需要在超类里加上新的实例变量来记录该调料的布尔值及价格，同时要修改cost实现
- ❑ 万一客户想要双倍mocha怎么办？
- ❑ 以后会出现新的饮料（如茶Tea）。作为Beverage的子类，某些调料可能并不适合（比如milk）。但是现在的设计方式中，Tea类仍将继承那些不适合的方法如hasMilk
 - ❑ 这就是继承带来的问题：利用继承来设计子类的行为，是在编译时静态决定的，而且所有的子类都会继承到相同的行为
- ❑ 如果利用对象组合的做法扩展对象的行为，就可以在运行时动态扩展。
- ❑ 装饰者模式就是利用这个技巧把多个新的职责、甚至是涉及超类时还没有想到的职责加到对象上

认识装饰者模式

- ❑ 现在知道了类继承带来的问题：类数量爆炸、设计死板、基类加入的新功能不适合所有子类
- ❑ 现在采用不一样的做法：以饮料为主体，然后在运行时以调料来“装饰”饮料，具体步骤为：
 - ❑ 拿一个饮料对象,如深焙咖啡(DarkRoast)
 - ❑ 以摩卡(Mocha)对象装饰它
 - ❑ 以牛奶(Milk)对象装饰它
 - ❑ 调用cost方法，并依赖委托（delegate）将调料的价钱加上去

1 以DarkRoast对象开始

cost()
DarkRoast

DarkRoast继承自
Beverage，有cost方法

2 顾客想要Mocha，所以 建立Mocha装饰对象，并用 它将DarkRoast对象包 (wrap)起来

cost()
Mocha

cost()
DarkRoast

Mocha对象是装饰者，它的类型“反映”了被装饰对象（**Beverage**）。所谓反映就是二者类型一致。因此也有cost方法

3 顾客想要Milk，所以 建立Milk装饰对象，并用它将 Mocha对象包 (wrap) 起来

cost()
Milk

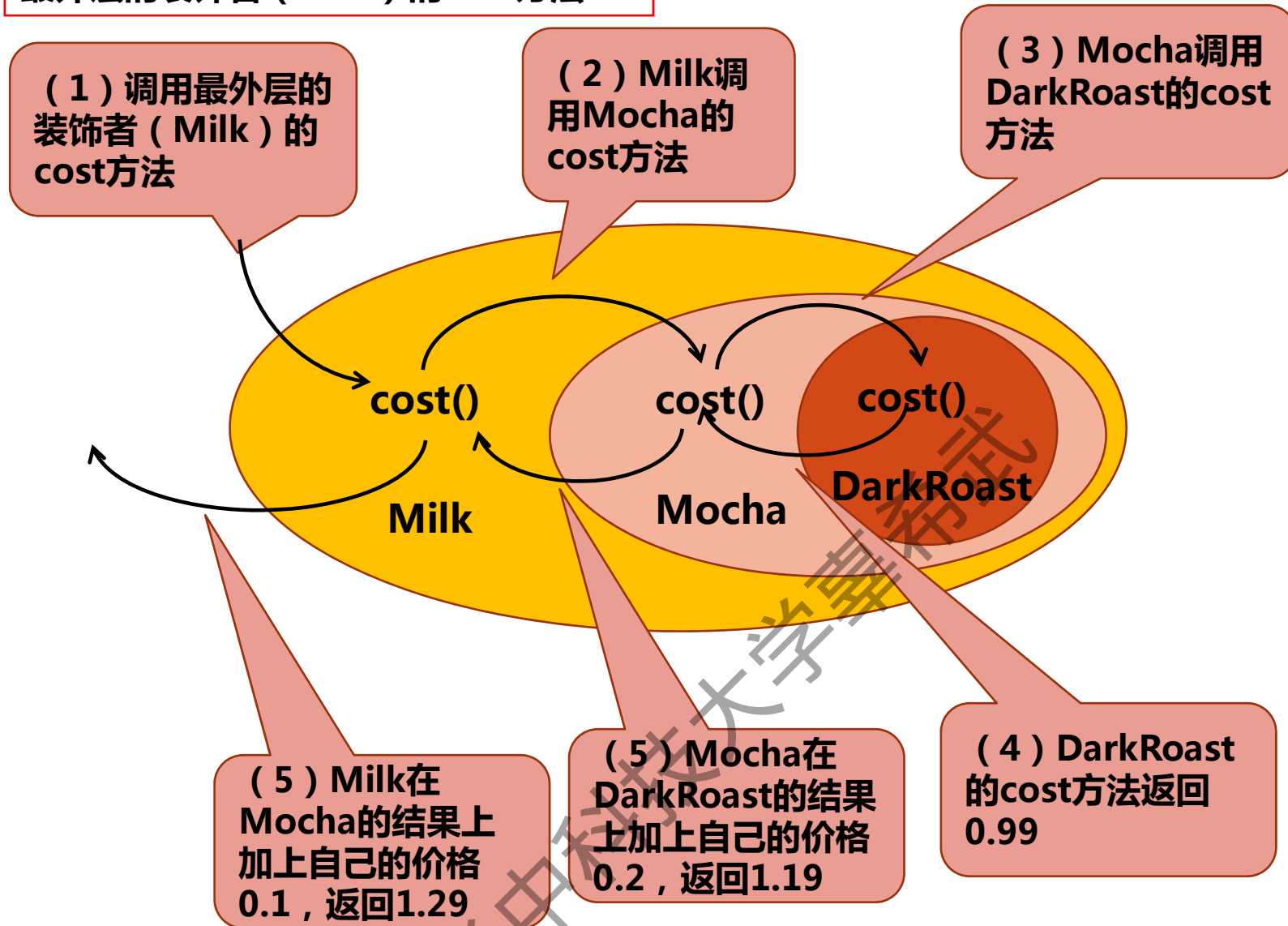
cost()
Mocha

cost()
DarkRoast

Milk对象是装饰者，它的类型“反映”了被装饰对象（**Beverage**），也有cost方法

因此被Mocha和Milk装饰的对象仍然是**Beverage**

4 该是为顾客算钱的时候了。通过调用最外层的装饰者 (Milk) 的cost方法



认识装饰者模式

- 好了，这是目前所知道的一切

- 装饰者和被装饰者有相同的超类型

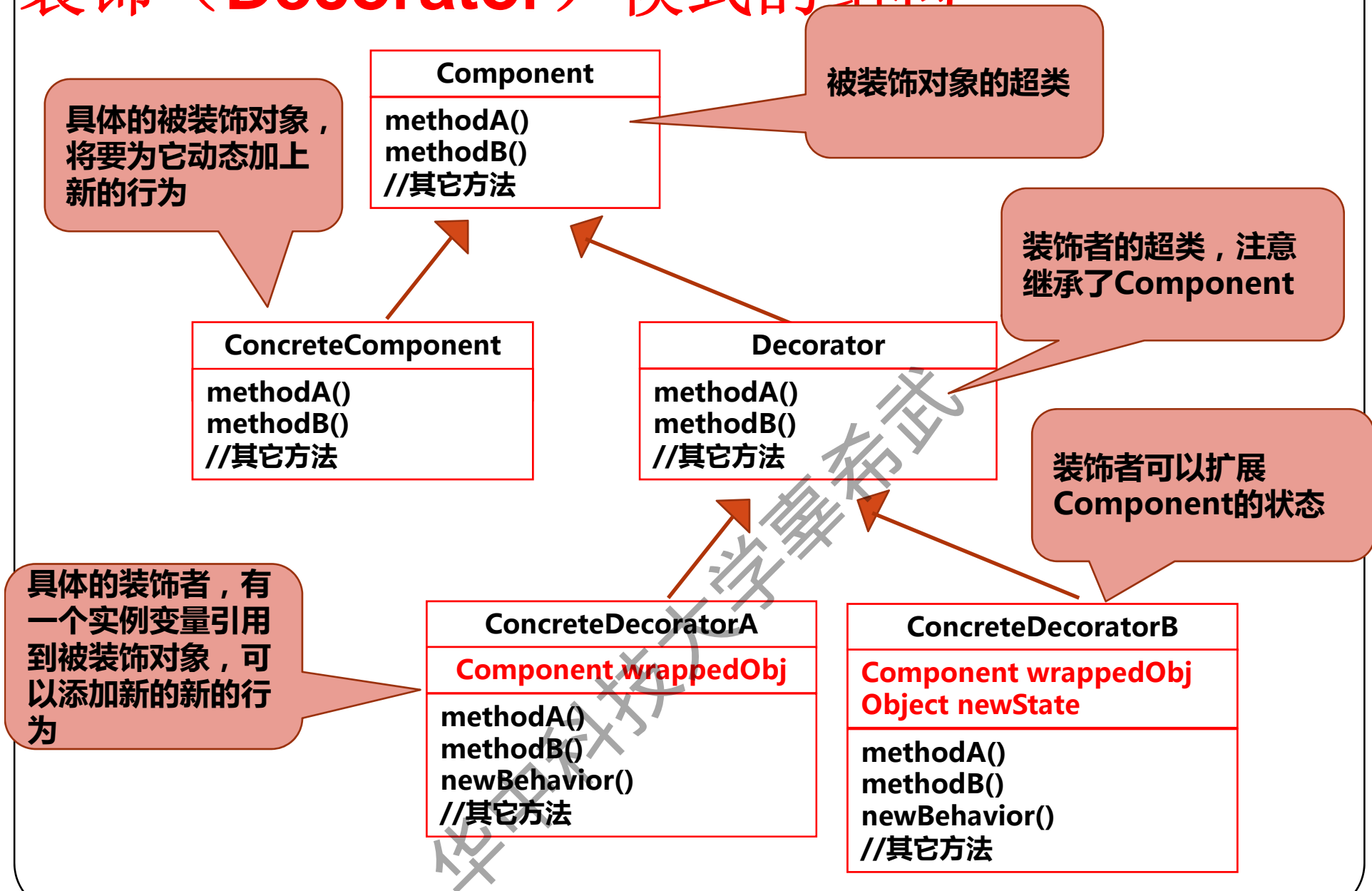
- 可以用一个或多个装饰者装饰一个对象

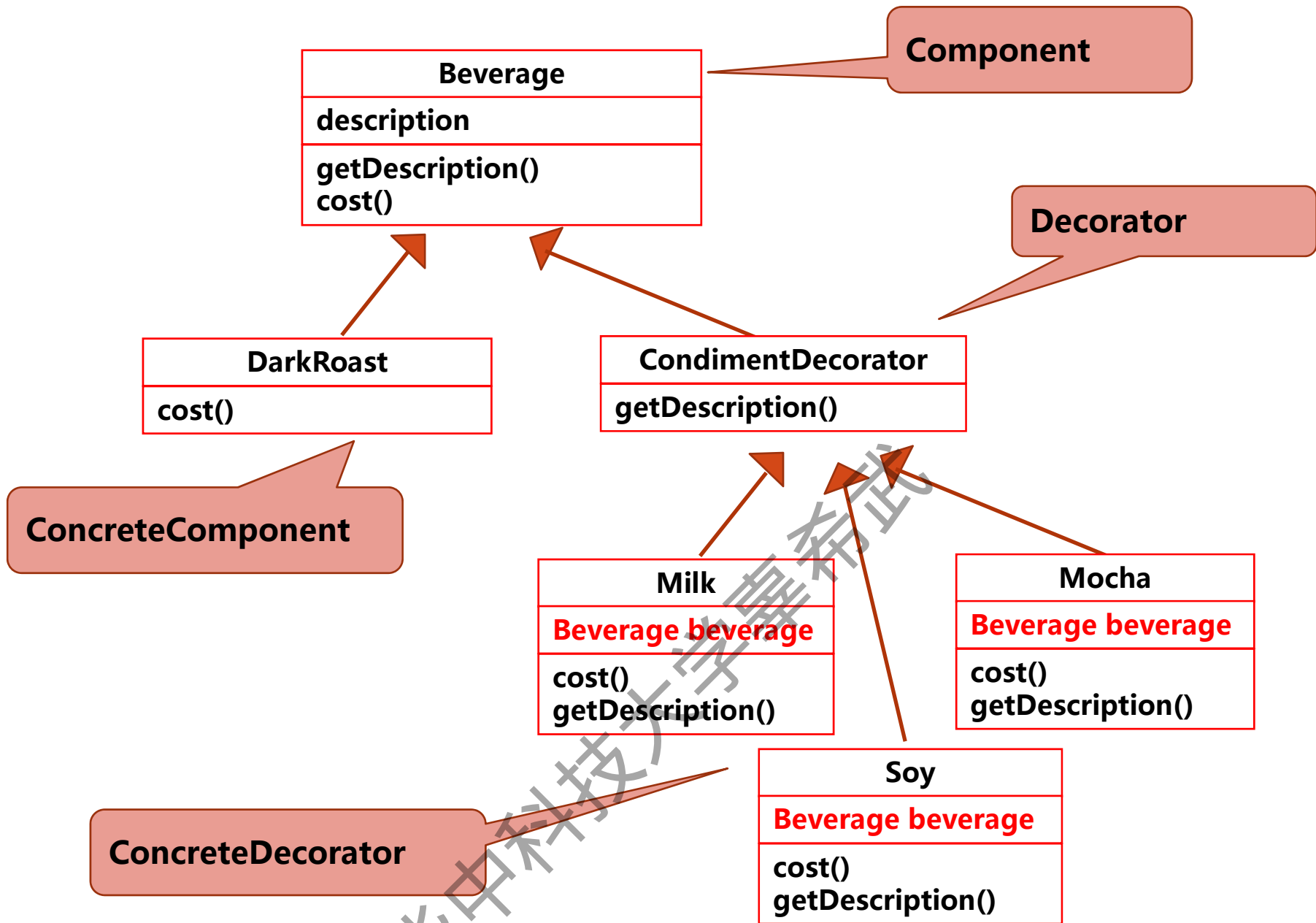
- 既然装饰者和被装饰者有相同的超类型，所以在任何需要原始对象（被装饰的）场合，可以用装饰过的对象代替它

- 装饰者可以在所委托被装饰者的行为之前/之后，加上自己的行为

- 对象可以在任何时候被装饰，所以可以在运行时动态地、不限量地用你喜欢的装饰者来装饰

装饰（Decorator）模式的结构





把设计变成真正的代码的时候到了

```
public abstract class Beverage{  
    String description = "Unknown Beverage" ;
```

```
    public String getDescription(){  
        return description;  
    }
```

给出了getDescription的实现

```
    public abstract double cost();  
}
```

必须在子类实现

```
public abstract class CondimentDecorator extends Beverage{  
    public abstract String getDescription();  
}
```

必须在具体装饰者对象实现子类实现

华中科技大学

把设计变成真正的代码的时候到了

```
public class DarkRoast extends Beverage{  
    public DarkRoast (){  
        description = "Dark Roast Coffee";  
    }  
  
    public double cost(){  
        return 0.99;    //返回饮料的价格  
    }  
}
```

具体的饮料类（被装饰对象）

```
public class HouseBlend extends Beverage{  
    public HouseBlend (){  
        description = "House Blend Coffee";  
    }  
  
    public double cost(){  
        return 0.89;    //返回饮料的价格  
    }  
}
```

具体的饮料类（被装饰对象）

把设计变成真正的代码的时候到了

具体的调料Mocha
(装饰者)

```
public class Mocha extends CondimentDecorator {
```

```
    Beverage beverage ;
```

beverage实例变量
记录被装饰对象

```
    public Mocha(Beverage beverage){  
        this.beverage = beverage;  
    }
```

利用构造函数参数，
传入被装饰对象

```
    public String getDescription(){  
        return beverage.getDescription() + ",Mocha"  
    }
```

希望描述信息不仅描述
饮料(如Dark Roast)，
而且还能包括调料。所以
利用委托，先得到被
包装对象的描述，再加
上装饰者自己的描述

```
    public double cost(){  
        return 0.20 + beverage.cost();  
    }
```

装饰者自己的价格+
被装饰对象的价格

```
}
```


把设计变成真正的代码的时候到了

具体的调料Milk (装饰者)

```
public class Milk extends CondimentDecorator {  
  
    Beverage beverage ;  
  
    public Milk beverage){  
        this.beverage = beverage;  
    }  
  
    public String getDescription(){  
        return beverage.getDescription() + ",Milk"  
    }  
    public double cost(){  
        return 0.1 + beverage.cost(); //返回饮料的价格  
    }  
}
```

把设计变成真正的代码的时候到了

具体的调料Soy (装饰者)

```
public class Soy extends CondimentDecorator {  
  
    Beverage beverage ;  
  
    public Soy beverage){  
        this.beverage = beverage;  
    }  
  
    public String getDescription(){  
        return beverage.getDescription() + ",Soy"  
    }  
    public double cost(){  
        return 0.3 + beverage.cost(); //返回饮料的价格  
    }  
}
```

把设计变成真正的代码的时候到了

```
public class StartBuzzCoffee {  
  
    public static void main(String args[]){  
        Bevegare beverage = new DarkRoast(); //订一杯咖啡，不叫调料  
  
        System.out.println(beverage.getDescription() + “ $” + beverage.cost());  
  
        beverage = new Mocha(beverage);    //加调料Mocha  
  
        beverage = new Milk(beverage);      //加调料Milk  
  
        beverage = new Soy(beverage);       //加调料Soy  
  
        System.out.println(beverage.getDescription() + “ $” + beverage.cost());  
    }  
}  
  
Dark Roast Coffee $0.99  
Dark Roast Coffee, Mocha, Milk, Soy $1.49
```

装饰（Decorator）模式的评价

- ❑ 使用Decorator模式可以很容易地向对象添加职责。
- ❑ 使用Decorator模式可以很容易地重复添加一个特性，而两次继承则极易出错
- ❑ 避免在层次结构高层的类有太多的特征：可以从简单的部件组合出复杂的功能。具有低依赖性和低复杂性
- ❑ 缺点：Decorator与Component不一样；有许多小对象

JAVA API中的装饰者模式

I/O流的分层

- 流能以套接管线的方式互相连接起来，一个用于输出的数据流同时可以是另一个用于输入的流，形成所谓的过滤流。方法是将一个的流对象传递给另一个流的构造函数。

JAVA API中的装饰者模式

I/O流的分层

- 例如，为了从二进制一个文件里读取数字（假设该二进制文件内容全是浮点数），首先要创建FileInputStream，然后将其传递给DataInputStream的构造方法。

```
InputStream fin =  
    new FileInputStream("data.dat");  
InputStream din =  
    new DataInputStream(fin);  
double s = din.readDouble();
```

JAVA API中的装饰者模式

I/O流的分层

- Reader inFromUser =
new BufferedReader(new InputStreamReader(System.in));

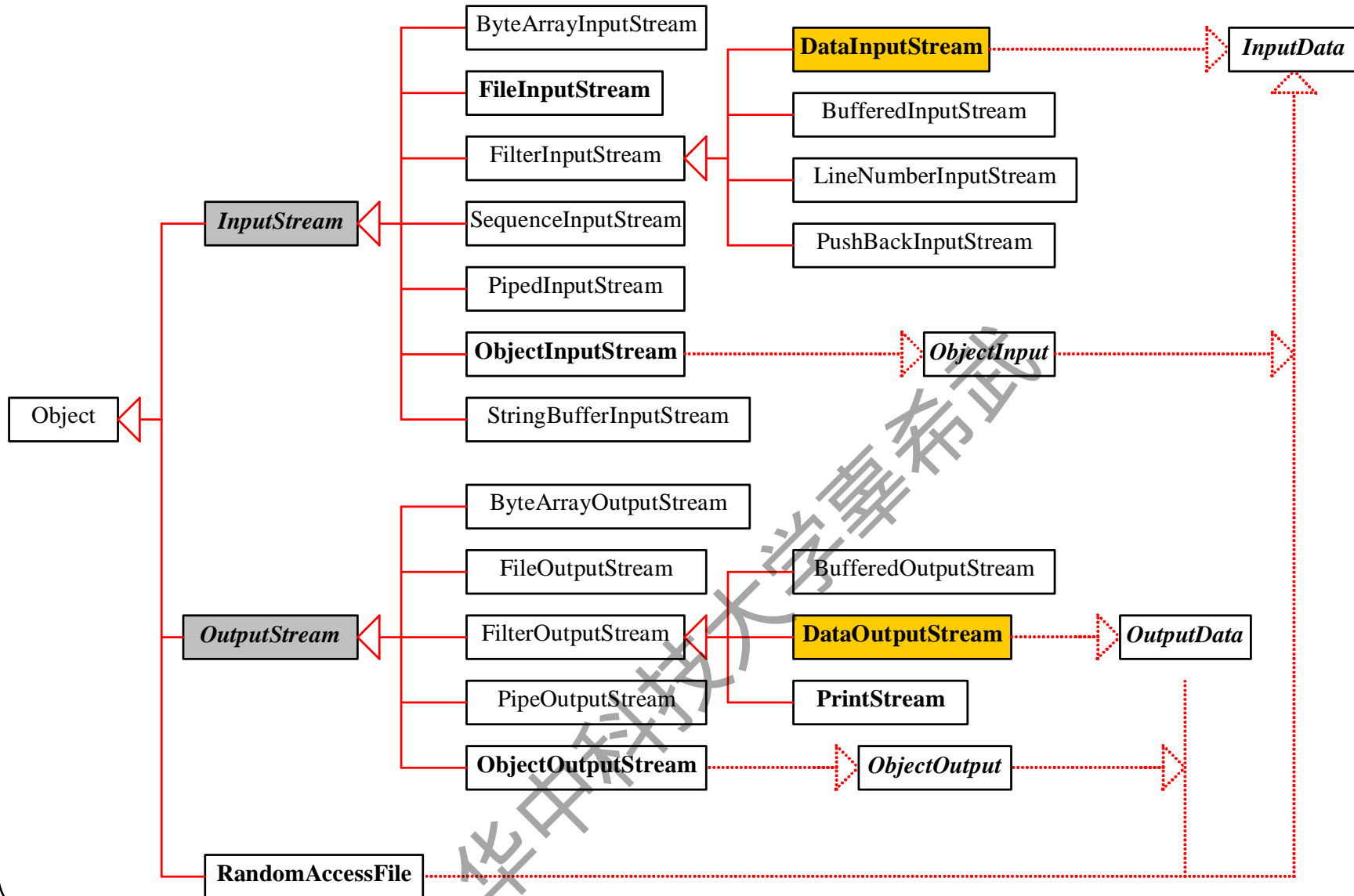
等价于：

```
Reader ins = System.in;  
Reader insReader =  
    new InputStreamReader(ins);  
Reader inFromUser =  
    new BufferedReader (insReader )
```

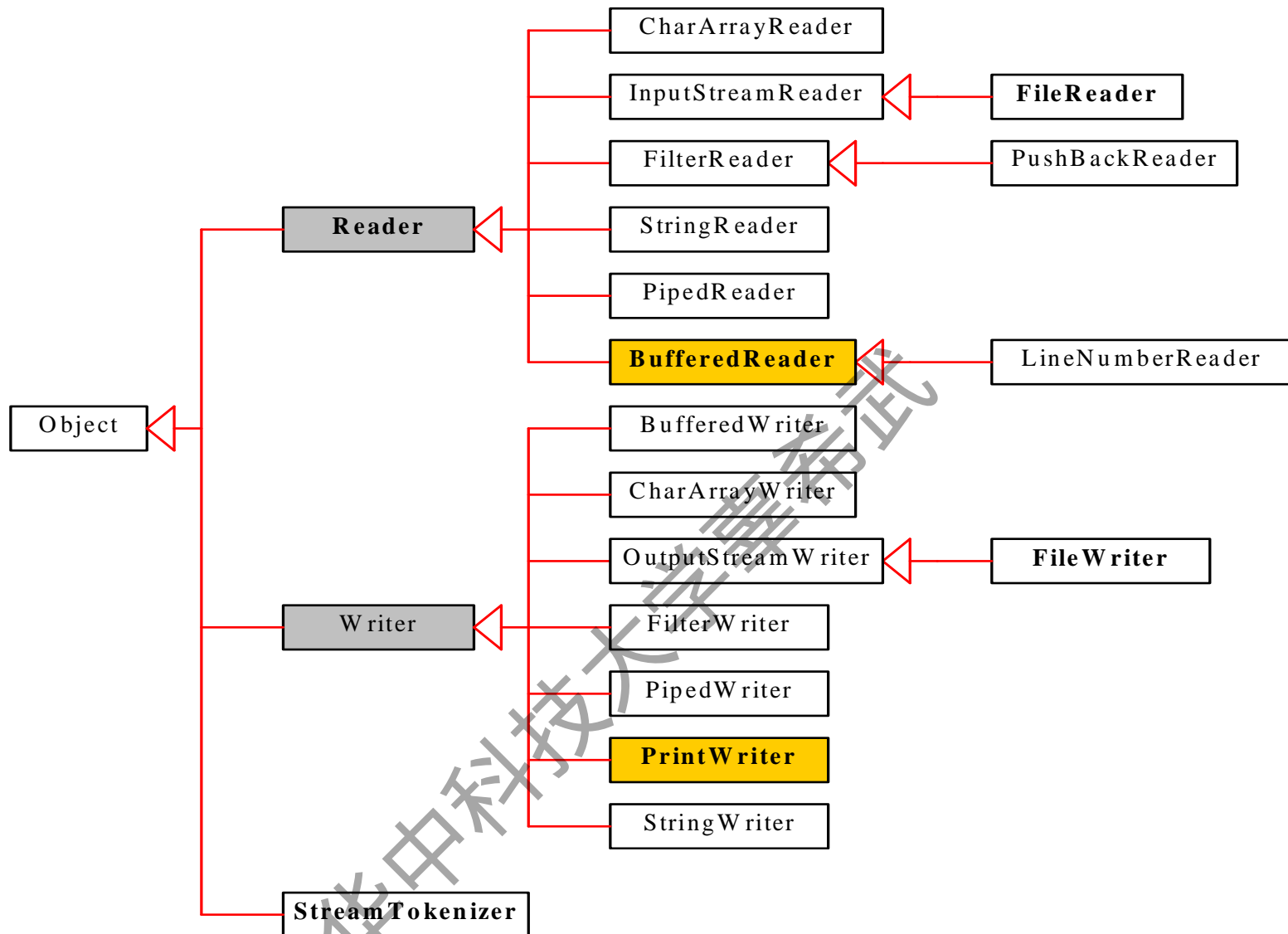


JAVA IO 流的这种机制如何实现：装饰者设计模式（**decorator pattern**）
JAVA API实现大量用到了设计模式：比如**swing**的**GUI**事件处理采用了观察者模式（**Observer Pattern**）

Byte Stream Classes



Character Stream Classes




迭代模式的由来

- 将对象职责分离，最大限度减少彼此之间的耦合程度，从而建立一个松散耦合的对象网络
- 集合对象拥有两个职责：一是存储内部数据；二是遍历内部数据。从依赖性看，前者为对象的根本属性，而后者既是可变化的，又是可分离的。可将遍历行为分离出来，抽象为一个迭代器，专门提供遍历集合内部数据对象行为。这是迭代子模式的本质

迭代模式的由来

□ 考虑煎饼屋 (PancakeHouse) 和餐厅 (Diner) 的菜单

```
public class MenuItem {  
    String name;  
    String description;  
    boolean vegetarian;  
    double price;  
    public MenuItem(String name, String description,  
                     boolean vegetarian, double price) {  
        this.name = name;  
        this.description = description;  
        this.vegetarian = vegetarian;  
        this.price = price;  
    }  
    public String getName() { return name; }  
    public String getDescription() { return description; }  
    public double getPrice() { return price; }  
    public boolean isVegetarian() { return vegetarian; }  
}
```



首先定义菜单项

迭代器模式的由来

煎饼屋菜单

```
public class PancakeHouseMenu {  
    ArrayList menuItems;
```

用ArrayList存储菜单项

```
    public PancakeHouseMenu(){  
        menuItems = new ArrayList();  
        addItem("Pancake1","Pancake1",true,2.99);  
        addItem("Pancake2","Pancake2",false,2.99);  
        addItem("Pancake3","Pancake3",true,3.49);  
        addItem("Pancake4","Pancake4",true,3.59);  
    }  
  
    public void addItem(String name, String description,  
        boolean vegetarian, double price){  
        MenuItem item = new MenuItem(name,description,  
            vegetarian,price);  
        menuItems.add(item);  
    }  
  
    public ArrayList getMenuItems(){ return menuItems; }  
}
```

迭代器模式的由来

餐厅菜单

```
public class DinerMenu {  
    static final int MAX_ITEMS = 6;  
    int numberOfItems = 0;  
    MenuItem[] menuItems;
```

用数组存储菜单项

```
    public DinerMenu(){  
        menuItems = new MenuItem[MAX_ITEMS];  
        addItem("Diner1","Diner1",true,2.99);  
        addItem("Diner2","Diner2",false,2.99);  
    }  
    public void addItem(String name,String description,  
                        boolean vegetarian,double price){  
        MenuItem item = new MenuItem(name,description,vegetarian,price);  
        if(numberOfItems >= MAX_ITEMS){  
            System.out.println("Sorry,menu is full!");  
        }  
        else{  
            menuItems[numberOfItems] = item;  
            numberOfItems ++;  
        }  
    }  
    public MenuItem[] getMenuItems(){ return menuItems; }  
}
```

❑ 现在煎饼屋和餐厅合并了，存在二种菜单。如果女服务员想打印菜单怎么办

```
public class Waitress {  
    public void printMenu(){  
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();  
        ArrayList breakfastItems = pancakeHouseMenu getMenuItems();  
  
        DinerMenu dinerMenu = new DinerMenu();  
        MenuItem[] lunchItems = dinerMenu getMenuItems();  
  
        for(int i = 0; i < breakfastItems.size(); i++){  
            MenuItem menuItem = (MenuItem) breakfastItems.get(i);  
            System.out.print(menuItem.getName() + " ");  
            System.out.println(menuItem.getPrice() + " ");  
            System.out.println(menuItem.getDescription());  
        }  
        for(int i = 0; i < lunchItems.length; i++){  
            MenuItem menuItem = lunchItems[i];  
            System.out.print(menuItem.getName() + " ");  
            System.out.println(menuItem.getPrice() + " ");  
            System.out.println(menuItem.getDescription());  
        }  
    }  
}
```

由于二种菜单存储
在不同的集合里，
因此必须写二个循
环，分别打印。

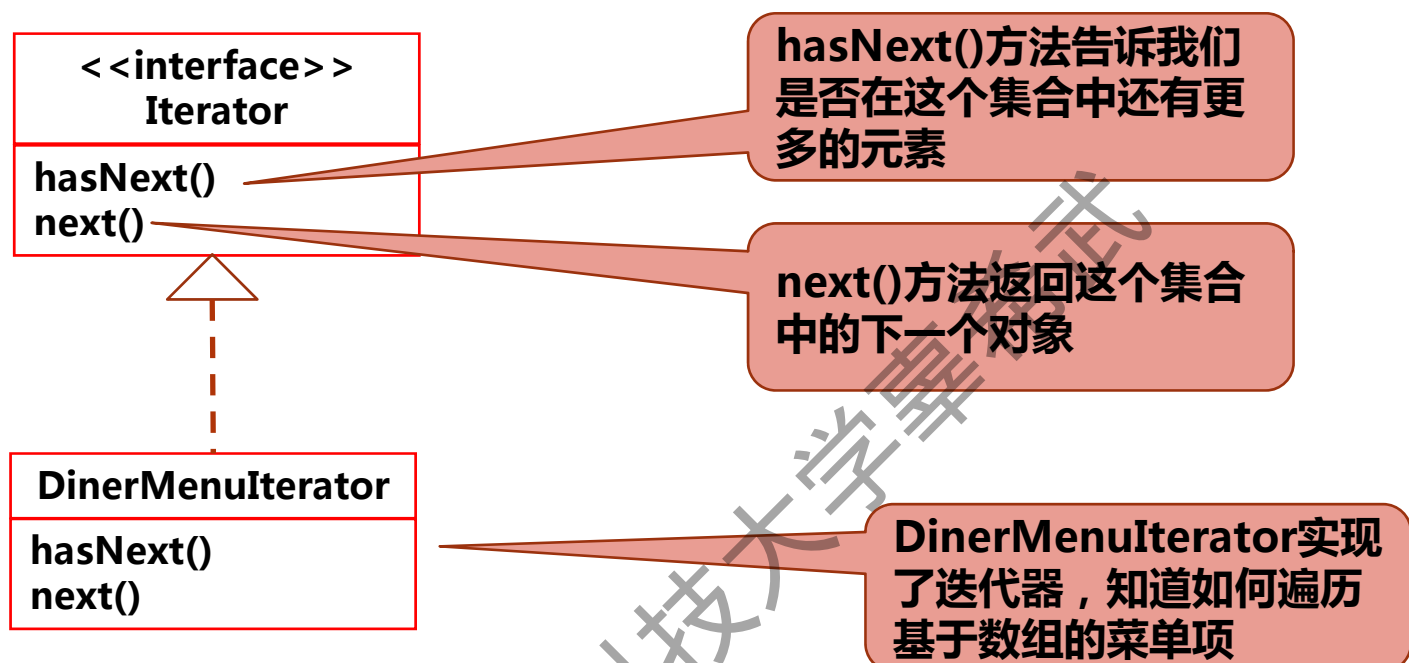
如果增加一个菜单，
放在第三种集合里，
还得增加一个循环

□ 这种实现的问题

- 针对具体类编程，不是针对接口
- 女招待需要知道每种菜单如何表达内部菜单项的集合，违反了封装
- 如果增加新的菜单，而且新菜单用别的集合来保存菜单项，我们又要修改女招待的代码
- 由于这二种集合遍历元素以及取元素的方法不同，导致必须写二个循环。但二个循环代码有重复
- 能不能找到一种统一的方式来遍历集合中的元素，使得只要一个循环就可以打印不同的菜单

初见迭代器模式

- 关于迭代器模式，我们需要知道的第一件事，就是它依赖于一个名为迭代器的接口



- 一旦我们有了迭代器接口，就可以为各种对象集合实现迭代器：数组，链表、Hash表，。。。

迭代器模式的由来

```
public class DinerMenuIterator implements Iterator {
```

```
    MenuItem[] items;
```

```
    int position = 0;
```

position记录当前遍历的位置，
items是集合对象

```
    public DinerMenuIterator(MenuItem[] items){
```

```
        this.items = items;
```

```
    }
```

通过构造函数，传递
集合对象给迭代器

```
    public boolean hasNext() {
```

```
        if(position >= items.length || items[position] == null)
```

```
            return false;
```

```
        else
```

```
            return true;
```

```
    }
```

因为使用数组（长度固定），所以不仅要检查position是否超出数组长度，还必须检查下一项是否为null，如果为null，表示没有下一项了

```
    public Object next() {
```

```
        MenuItem item = items[position];
```

```
        position ++;
```

```
        return item;
```

```
    }
```

```
}
```

返回数组内的下一项，并递增当前遍历的位置

迭代器模式的由来

```
public class PancakeHouseMenuIterator implements Iterator {
```

```
    ArrayList items;
```

```
    int position = 0;
```

position记录当前遍历的位置，
items是集合对象

```
    public PancakeHouseMenuIterator(ArrayList items){
```

```
        this.items = items;
```

```
    }
```

通过构造函数，传递
集合对象给迭代器

```
    public boolean hasNext() {
```

```
        if(position >= items.size())
```

```
            return false;
```

```
        else
```

```
            return true;
```

```
    }
```

```
    public Object next() {
```

```
        MenuItem item = (MenuItem)items.get(position);
```

```
        position ++;
```

```
        return item;
```

```
    }
```

```
}
```

□ 现在改写餐厅菜单

```
public class DinerMenu {  
    static final int MAX_ITEMS = 6;  
    int numberOfItems = 0;  
    MenuItem[] menuItems;  
  
    public DinerMenu(){  
        ...  
    }  
    public void addItem(String name,String description,  
                        boolean vegetarian,double price){  
        ...  
    }  
public MenuItem[] getMenuItems(){ return menuItems; }  
  
    public Iterator createIterator(){  
        return new DinnerMenuIterator(menuItems);  
    }  
}
```

返回迭代器接口。客户不需要知道餐厅菜单如何维护菜单项，也不需要知道迭代器如何实现，只需直接使用迭代器遍历菜单项

□ 现在改写餐厅菜单

```
public class DinerMenu {  
    static final int MAX_ITEMS = 6;  
    int numberOfItems = 0;  
    MenuItem[] menuItems;  
  
    public DinerMenu(){  
        ...  
    }  
    public void addItem(String name,String description,  
                        boolean vegetarian,double price){  
        ...  
    }  
    public MenuItem[] getMenuItems(){ return menuItems; }  
  
    public Iterator createIterator(){  
        return new DinnerMenuIterator(menuItems);  
    }  
}
```

返回迭代器接口。客户不需要知道餐厅菜单如何维护菜单项，也不需要知道迭代器如何实现，只需直接使用迭代器遍历菜单项

□ 现在改写煎饼屋菜单

```
public class PancakeHouseMenu {  
    ArrayList menuItems;  
  
    public PancakeHouseMenu(){  
        ...  
    }  
    public void addItem(String name, String description,  
        boolean vegetarian, double price){  
        ...  
    }  
public ArrayList getMenuItems(){ return menuItems; }  
  
    public Iterator createIterator(){  
        return new PancakeHouseMenuIterator(menuItems);  
    }  
}
```

返回迭代器接口。客户不需要知道餐厅菜单如何维护菜单项，也不需要知道迭代器如何实现，只需直接使用迭代器遍历菜单项

□ 现在改写女招待代码

```
public class Waitress {  
    PancakeHouseMenu pancakeHouseMenu  
    DinerMenu dinerMenu ;  
    public Waitress(PancakeHouseMenu pm, DinerMenu dm){  
        pancakeHouseMenu = pm; dinerMenu = dm;  
    }  
    public void printMenu(){  
        Iterator pancakeIt = pancakeHouseMenu.createIterator();  
        Iterator dinerIt = dinerMenu.createIterator();  
        printMenu(pancakeIt );  
        printMenu(dinerIt);  
    }  
    public void printMenu(Iterator it){  
        while(it.hasNext()){  
            MenuItem menuItem = (MenuItem)it.next();  
            System.out.print(menuItem.getName() + " ");  
            System.out.println(menuItem.getPrice() + " ");  
            System.out.println(menuItem.getDescription());  
        }  
    }  
}
```

分别得到二个菜单的迭代器

现在的printMenu函数利用迭代器来遍历集合中的元素，而不用关心集合是如何组织元素的

❑ 到目前为止，我们做了什么

- ❑ 菜单的实现已经被封装起来了。女招待不知道每种菜单如何存储内部菜单项的
- ❑ 只要实现迭代器，我们只需要一个循环，就可以多态地处理任何项的集合
- ❑ 女招待现在只使用一个接口(迭代器)
- ❑ 但女招待仍然捆绑于二个具体的菜单类，需要修改下。
通过定义Menu接口

```
public interface Menu{  
    public Iterator createIterator();  
}
```

□ 定义Menu接口，对二个菜单类做简单的修改

```
public interface Menu{  
    public Iterator createIterator();  
}
```

```
public class DinerMenu implements Menu{  
    //其他不变  
    public Iterator createIterator(){  
        return new DinnerMenuIterator(menuItems);  
    }  
}
```

```
public class PancakeHouseMenu implements Menu{  
    //其他不变  
    public Iterator createIterator(){  
        return new PancakeHouseMenuIterator (menuItems);  
    }  
}
```


□ 现在女招待不再捆绑到具体类了

```
public class Waitress {  
    Menu pancakeHouseMenu  
    Menu dinerMenu ;  
    public Waitress(Menu pm, Menu dm){  
        pancakeHouseMenu = pm; dinerMenu = dm;  
    }  
    public void printMenu(){  
        Iterator pancakeIt = pancakeHouseMenu.createIterator();  
        Iterator dinerIt = dinerMenu.createIterator();  
        printMenu(pancakeIt );  
        printMenu(dinerIt);  
    }  
    public void printMenu(Iterator it){  
        while(it.hasNext){  
            MenuItem menuItem = (MenuItem)it.next();  
            //打印MenuItem的内容  
        }  
    }  
}
```

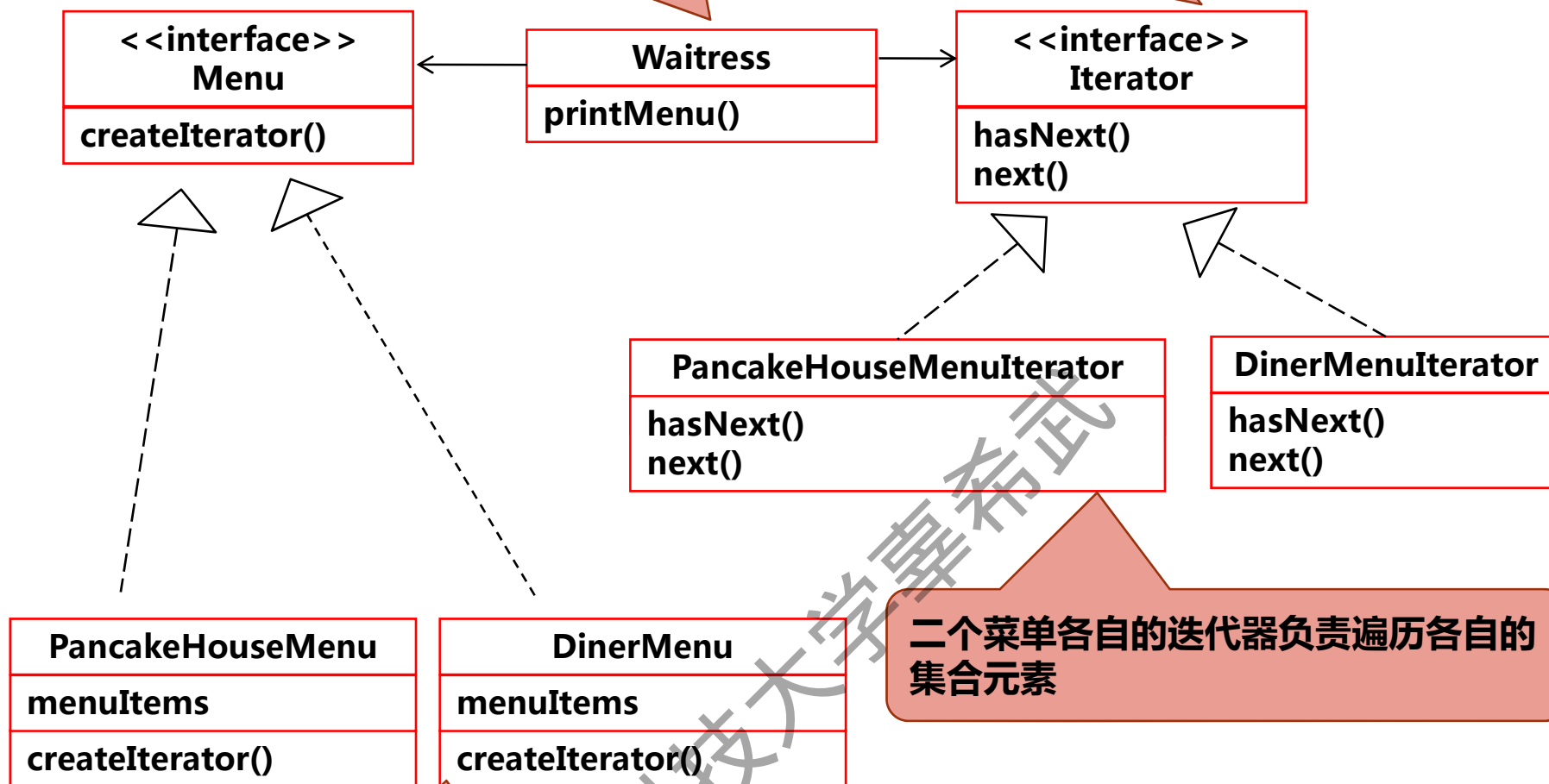
□ 测试我们的代码

```
public class MenuDrive{  
    public void main(String args){  
        Menu pMenu = new PancakeHouseMenu();  
        Menu dMenu = new DinerMenu();  
  
        Waitress waitress = new Waitress(pMenu,dMenu);  
        waitress.printMenu();  
    }  
}
```

华中科技大学

女招待只关心菜单和迭代器接口

女招待从菜单实现解耦，现在利用迭代器遍历菜单项，无须知道菜单具体实现



二个菜单各自的迭代器负责遍历各自的集合元素

二个菜单都实现了createIterator方法，返回各自的迭代器

定义迭代器模式

- ❑ 迭代器模式提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示
- ❑ 把遍历元素的任务从聚合对象剥离出来，放到迭代器上。这样简化了聚合的接口和实现。也让任务各得其所
- ❑ 职责分离！
 - ❑ 一个类一个职责
 - ❑ 一个职责意味着一个变化的可能
 - ❑ 类的职责多了，意味着变化的可能多了，类被修改的可能性就大

迭代器模式的意图和适用性

□ 意图

- 迭代器模式的目的是设计一个迭代器，提供一种可顺序访问聚合对象中各个元素的方法，但不暴露该对象内部表示

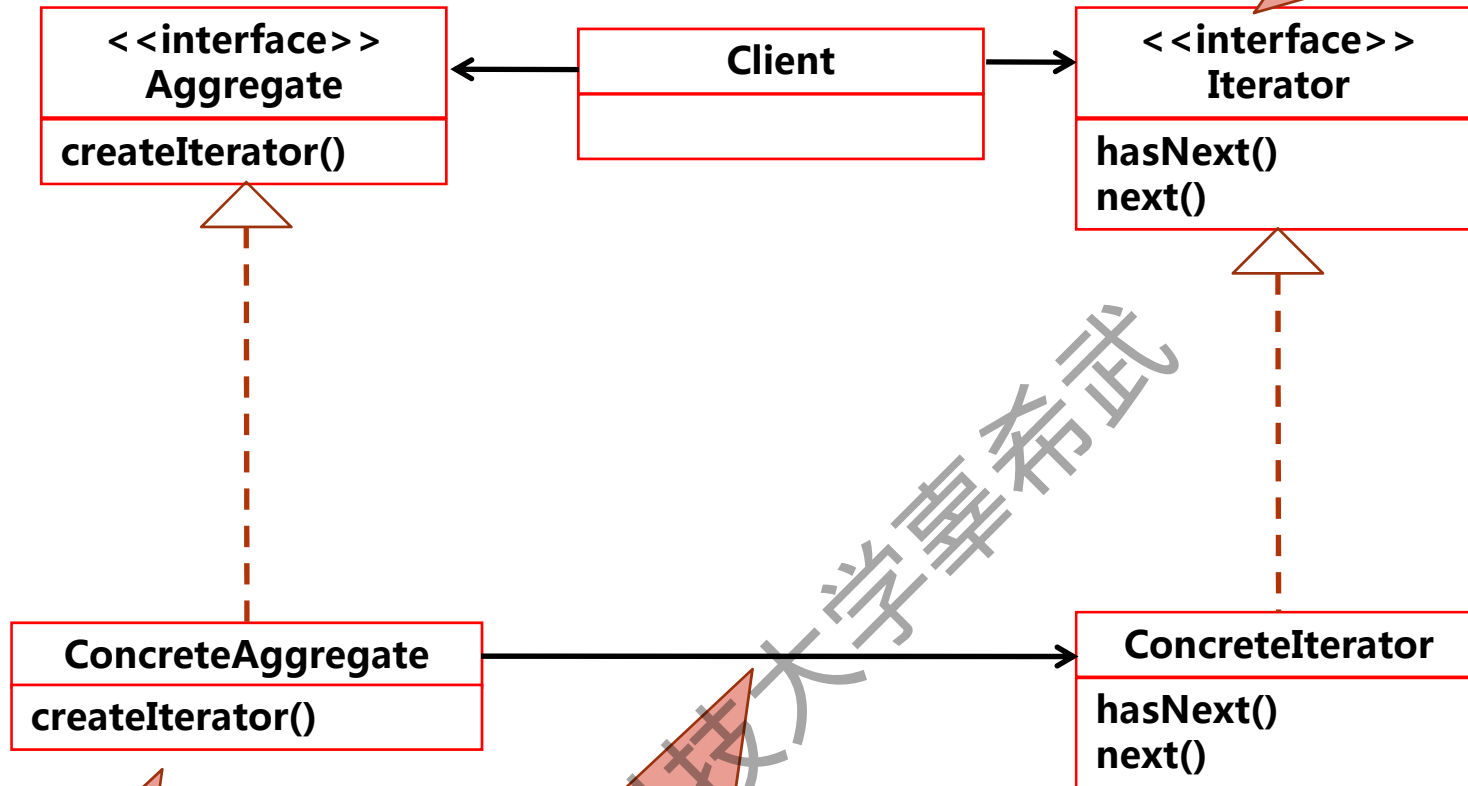
□ 适用场合

- 访问一个聚合对象的内容而无需暴露其内部表示
- 支持对聚合对象的多种遍历
- 为遍历不同的聚合结构提供一个统一接口 (支持多态迭代)

迭代器模式的结构

共同的接口供所有聚合使用

所有迭代器必须实现的接口。接口中的方法可以遍历集合元素。
注意java.util.Iterator接口还定义了remove方法



具体聚合都持有一个对象集合

具体聚合都负责实例化一个具体迭代器

具体迭代器负责遍历具体聚合对象的元素

迭代器模式的参与者

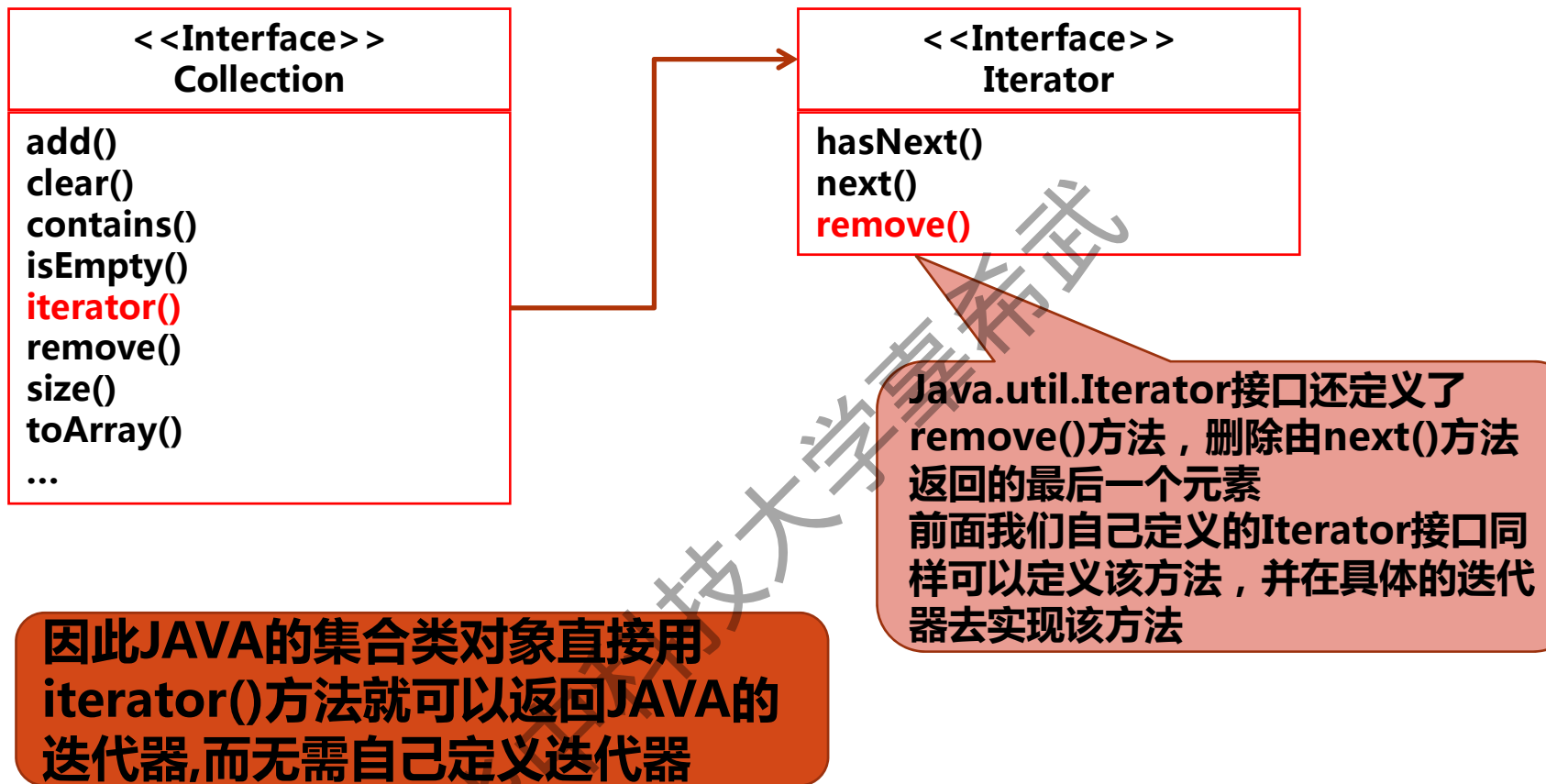
- ❑ Iterator
 - ❑ 定义访问和遍历元素的接口
- ❑ Concrete Iterator
 - ❑ 实现迭代器接口
- ❑ Aggregate
 - ❑ 定义创建迭代器对象的接口
- ❑ Concrete Aggregate
 - ❑ 实现创建迭代器的接口，返回具体迭代器的一个实例

迭代器模式的效果分析

- ❑ 简化了聚集的行为，迭代器具备遍历接口的功能，聚集不必具备遍历接口
- ❑ 每一个聚集对象都可以有一个或者更多的迭代器对象，每一个迭代器的迭代状态可以彼此独立
- ❑ 遍历算法被封装到迭代器对象中，迭代算法可以独立于聚集对象变化。客户端不必知道聚集对象的类型，通过迭代器就可以读取和遍历聚集对象。聚集内部数据发生变化不影响客户端程序

JAVA中的迭代器模式

- JAVA中的集合类都实现了java.util.Collection接口，这个接口中定义了一个iterator()方法，返回java.util.Iterator接口



组合（Composite）模式的由来

□ 既然迭代器工作得这么完美，现在把咖啡厅也合并进来供应晚餐

```
public class CafeMenu implements Menu {  
    Hashtable menuItems = new Hashtable();  
    public CafeMenu() {  
        addItem("Caffe1", "Caffe1", true, 3.99);  
        ...  
    }  
    public void addItem(String name, String description,  
        boolean vegetarian, double price)  
    {  
        MenuItem item = new MenuItem(name, description, vegetarian, price);  
        menuItems.put(menuItem.getName(), item);  
    }  
    public Iterator createIterator() {return  
        menuItems.values().iterator();  
    }  
}
```

Java的集合类（不是数组）都有iterator方法返回java.util.Iterator接口，注意HashTable的每一项都是(key,value)对，因此我们只要获得值的迭代器

□ 让女招待认识咖啡厅菜单

```
public class Waitress {  
    Menu pancakeHouseMenu  
    Menu dinerMenu ;  
    Menu cafeMenu;  
  
    public Waitress(Menu pm, Menu dm, Menu cm){  
        pancakeHouseMenu = pm; dinerMenu = dm; cafeMenu = cm;  
    }  
    public void printMenu(){  
        Iterator pancakeIt = pancakeHouseMenu.createIterator();  
        Iterator dinerIt = dinerMenu.createIterator();  
        Iterator cafeIt = cafeMenu.createIterator();  
        printMenu(pancakeIt );  
        printMenu(dinerIt);  
        printMenu(cafeIt)  
    }  
    public void printMenu(Iterator it){  
        while(it.hasNext){  
            MenuItem menuItem = (MenuItem)it.next();  
            //打印MenuItem的内容  
        }  
    }  
}
```

□ 再来测试我们的代码

```
public class MenuDrive{  
    public void main(String args){  
        PancakeHouseMenu pMenu = new PancakeHouseMenu();  
        DinerMenu dMenu = new DinerMenu();  
        CafeMenu cMenu = new CafeMenu();  
  
        Waitress waitress = new Waitress(pMenu,dMenu,cMenu);  
        waitress.printMenu();  
    }  
}
```

□ Waitress的代码还有什么问题？

```
public class Waitress {
```

```
...
```

```
public void printMenu(){
```

```
    Iterator pancakeIt = pancakeHouseMenu.createIterator();
```

```
    Iterator dinerIt = dinerMenu.createIterator();
```

```
    Iterator cafeIt = cafeMenu.createIterator();
```

```
    printMenu(pancakeIt );
```

```
    printMenu(dinerIt);
```

```
    printMenu(cafeIt)
```

```
}
```

```
public void printMenu(Iterator it){
```

```
...
```

```
}
```

```
}
```

每次增加或删除菜单，必须修改该类的代码
违反了开放-关闭原则

三次调用printMenu，看起来比较丑陋

虽然迭代器模式使得菜单元素的遍历与菜单的实现分离了，但菜单对象还是分离而独立的。我们还需要一种把菜单统一管理起来的方法

□ 利用ArrayList可以把所有菜单管理起来

```
public class Waitress {  
    ArrayList menus ;
```

所有的菜单都放到ArrayList中

```
    public Waitress(ArrayList menus){  
        this.menus = menus;  
    }
```

利用构造函数传入菜单列表

```
    public void printMenu(){  
        Iterator menuiterator = menus.iterator();  
        while(menuiterator.hasNext()){  
            Menu menu = (Menu)menuiterator.next();  
            printMenu(menu.createIterator());  
        }  
    }
```

ArrayList的iterator方法返回
java.util.Iterator迭代器

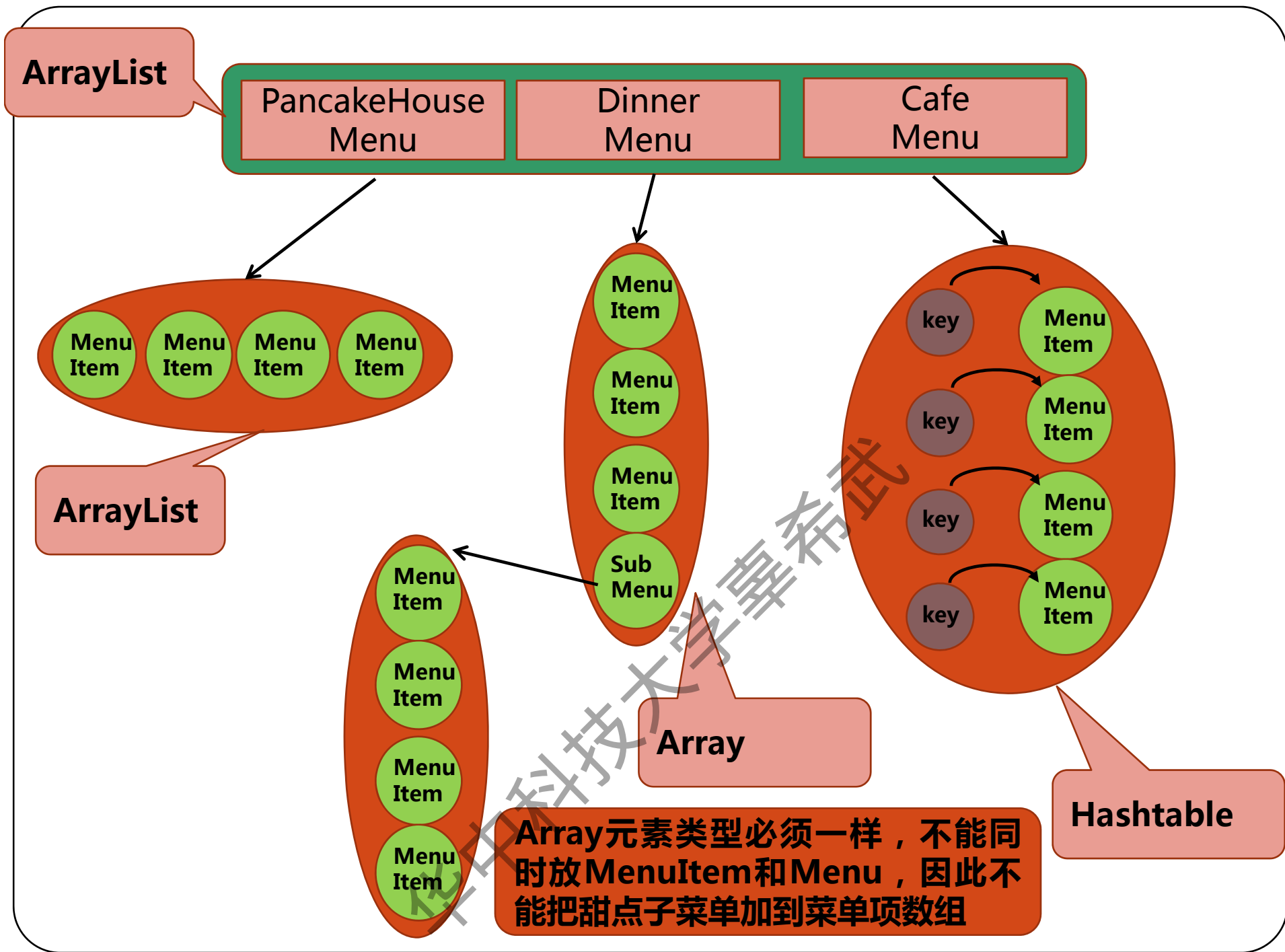
利用java.util.Iterator迭代器遍
历每个菜单

```
    public void printMenu(Iterator it){  
        while(it.hasNext){  
            Menuitem menuitem = (Menuitem)it.next();  
            //打印Menuitem的内容  
        }  
    }  
}
```

打印当前遍历
的菜单

看起来完美无缺了

- 现在我们应该能够加上一份餐后甜点的“子菜单”，即菜单里可以包含子菜单。
- 我们想要的是这样一种结构



□ 现在我们的需求已经到了一个复杂级别，需要重新设计菜单的结构

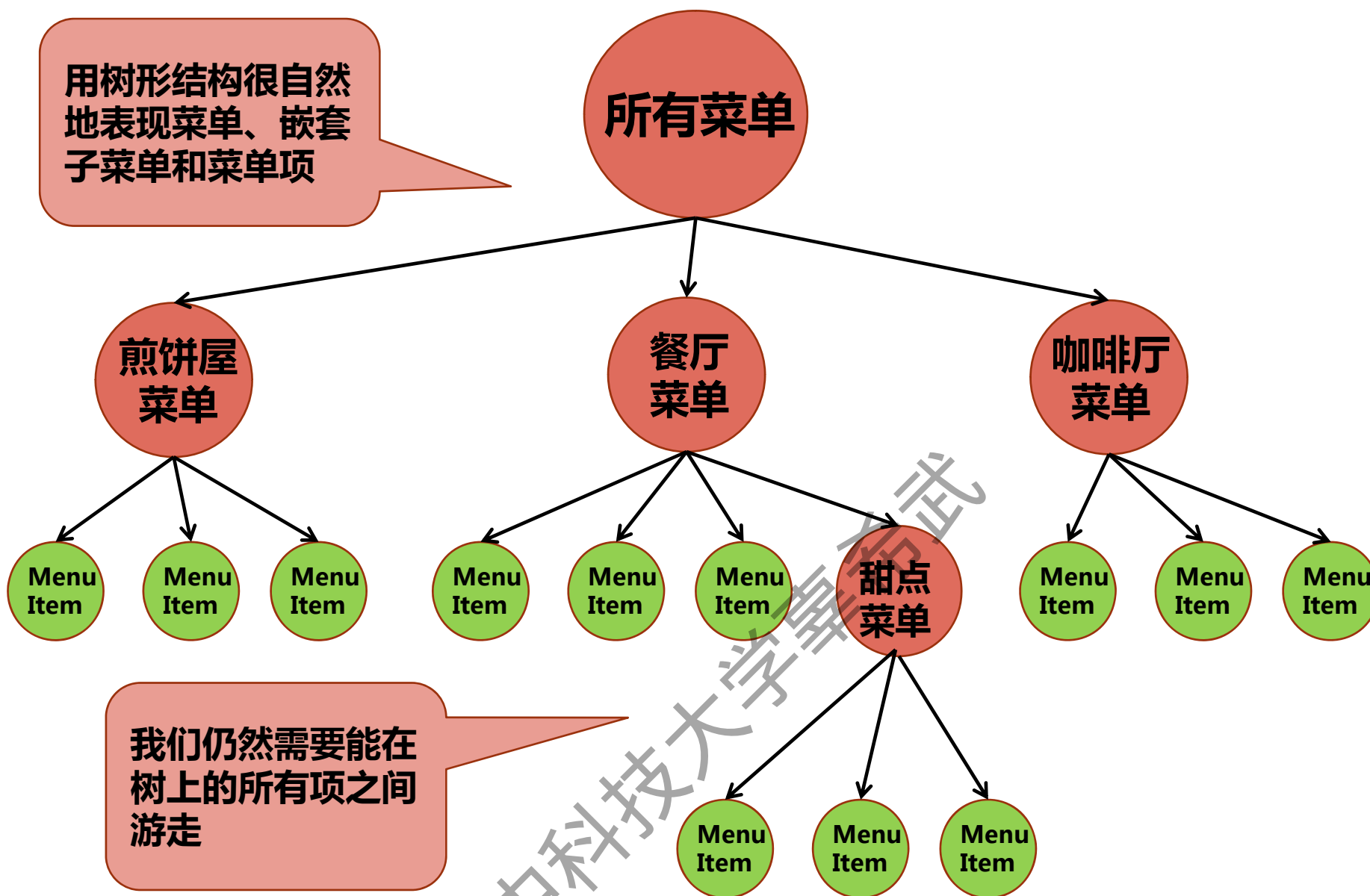
□ 我们真正需要什么？

□ 需要某种树形结构，可以容纳菜单、子菜单和菜单项，而且这样的树形结构能表现整体/部分的层次结构。即能让客户以一致的方式处理菜单和菜单项

□ 能够遍历整个对象树，至少要像迭代器那样方便

□ 我们需要的树形结构看起来应该像这样

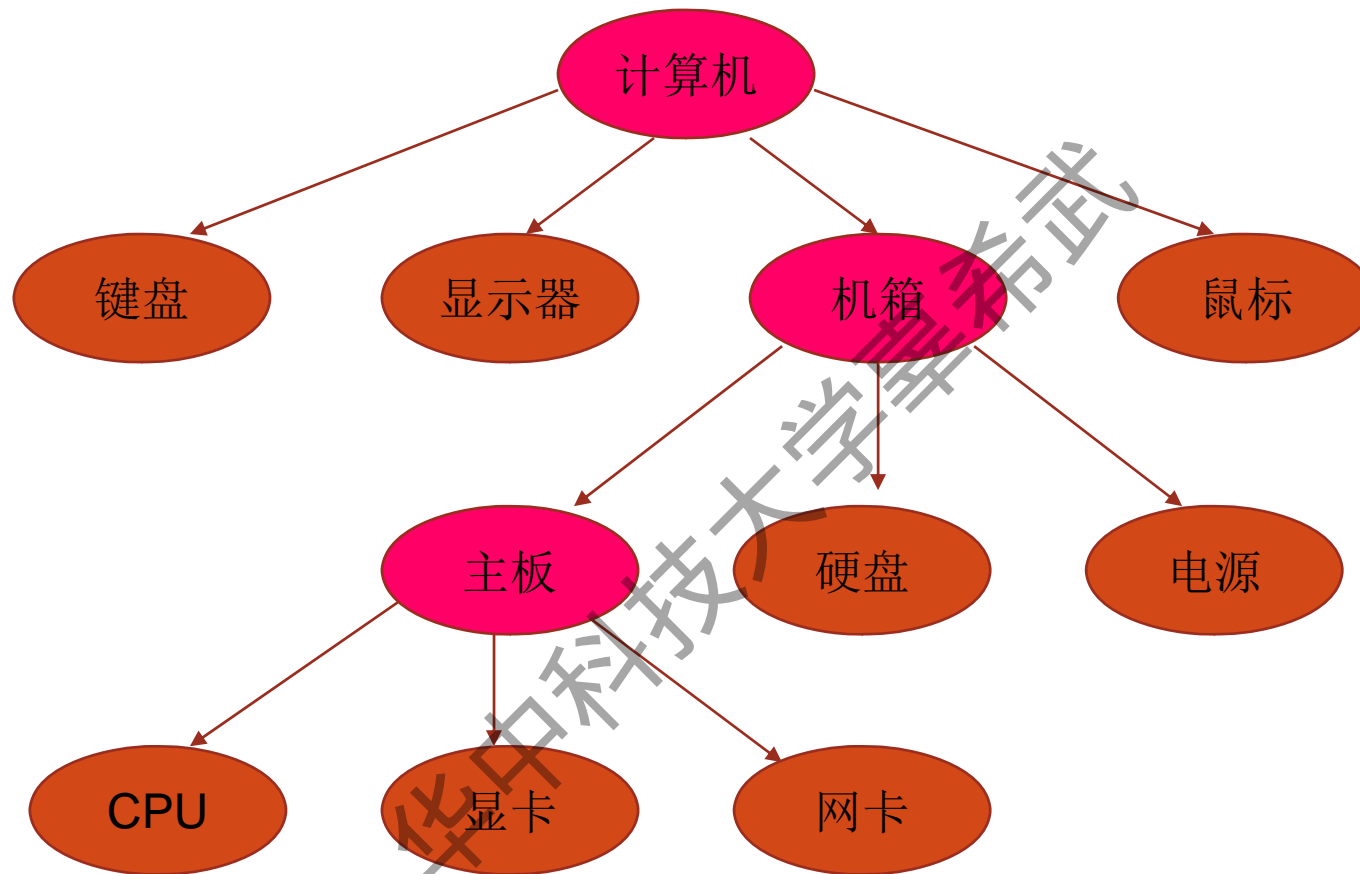
用树形结构很自然地表现菜单、嵌套子菜单和菜单项

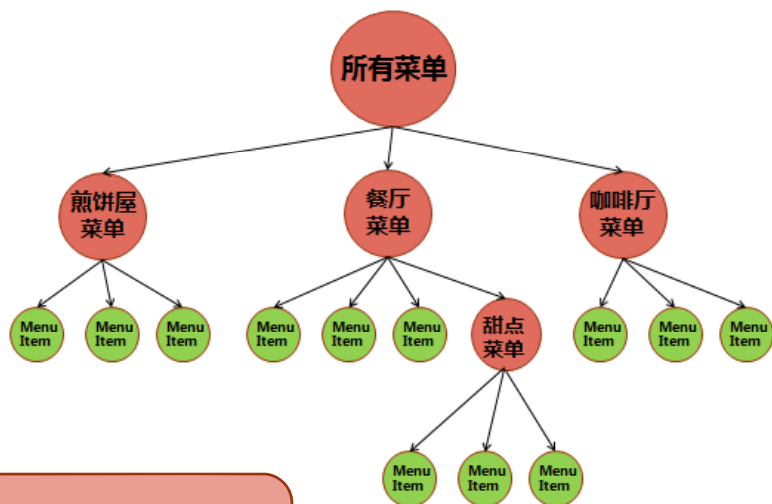


我们仍然需要能在树上的所有项之间游走

定义组合模式（Composite Pattern）

- 组合模式允许你将对象组合成树形结构来表现“整体/部分”层次结构。组合能让客户以一致的方式处理个别对象以及对象组合

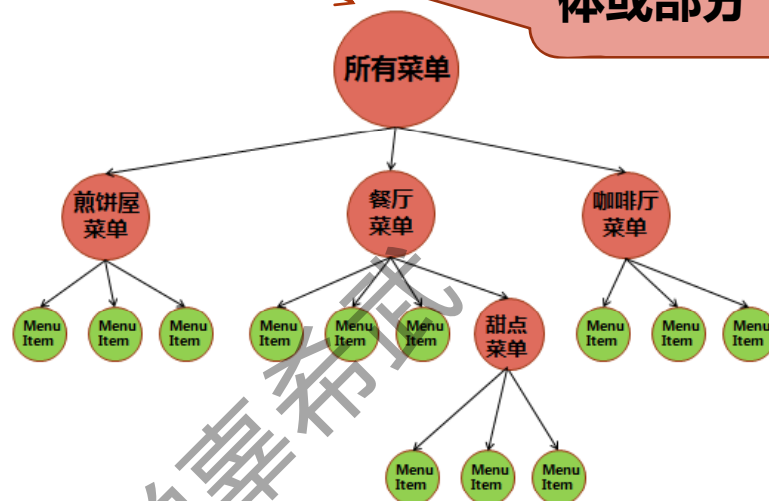




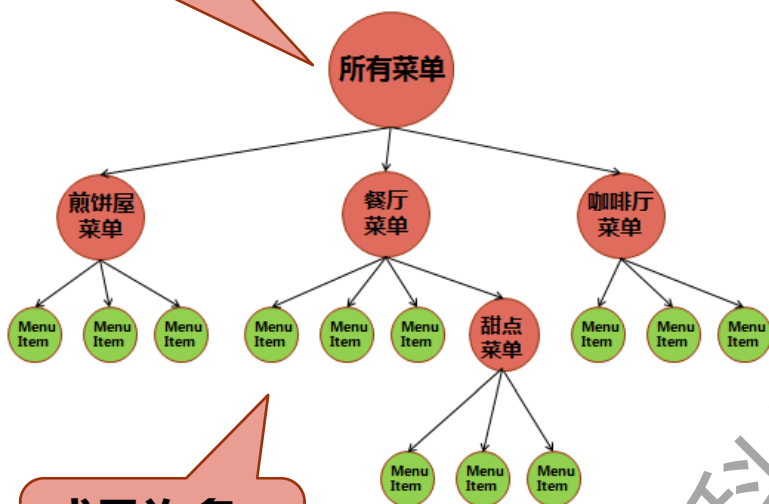
我们可以创建任意复杂的树

print

操作可以应用于整体或部分



然后将它们视为整体



或是许多部分

使用组合结构，能把相同的操作应用在组合或个别对象上。即可以忽略组合对象和个别对象的差别

组合（Composite）模式的意图和适用性

□ 意图：

- 将对象组合成树形结构，表示“部分/整体”层次结构
- 使用户对单一对象和组合对象使用具有一致性接口

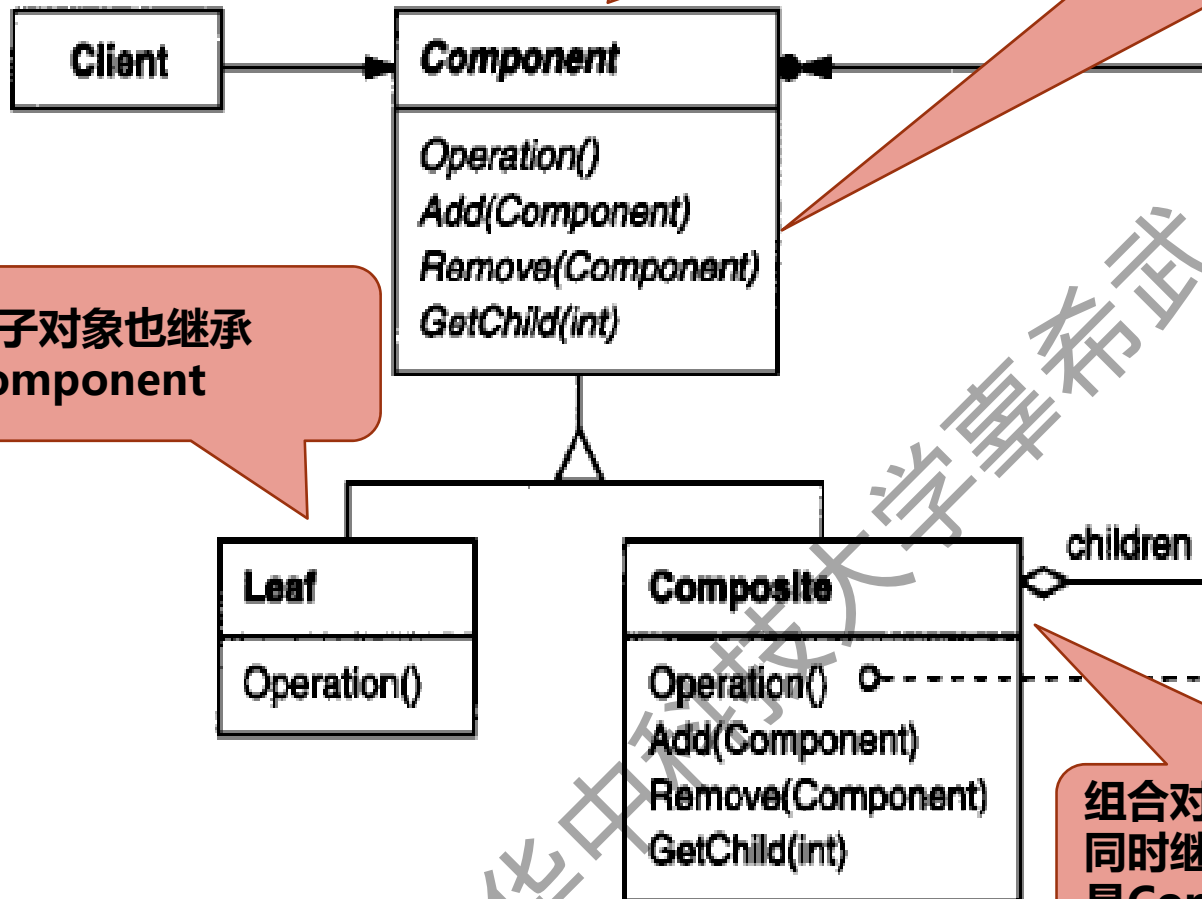
□ 适用性

- 表示对象的部分-整体层次结构
- 忽略总体对象与单一对象差异，统一使用组合结构的所有对象

组合 (Composite) 模式的结构

Component为组合中的所有对象定义一个接口(这里的接口是泛指, 可以是抽象类)

Component可以为Add, Remove, GetChild和操作提供默认行为



叶子对象也继承 Component

树的每个节点 (Composite) 和叶子 (Leaf) 都是Component

组合对象包含多个Component, 同时继承Component, 因此也是Component

组合（Composite）模式的由来

□ 抽象

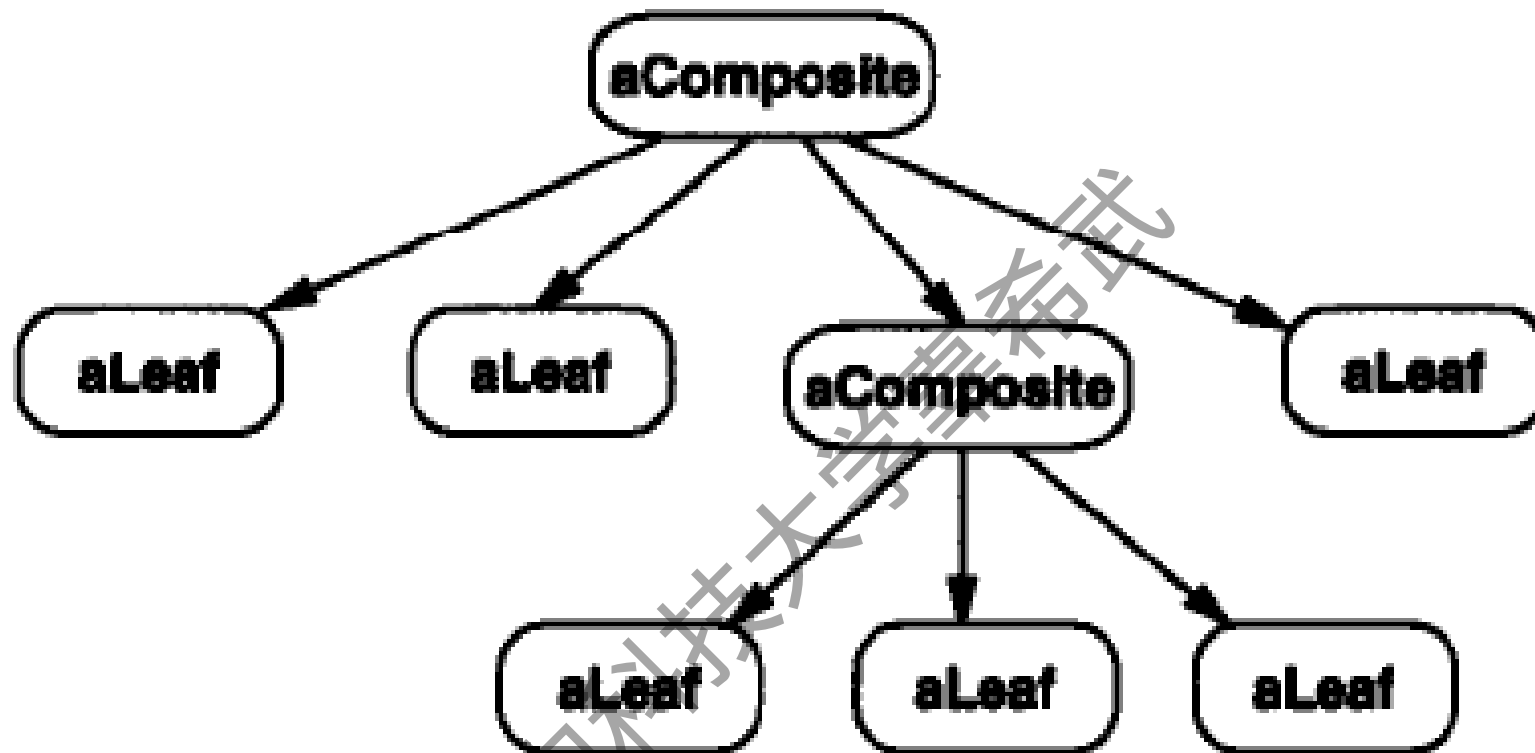
- 单一对象(Leaf)：键盘、鼠标、显示器、硬盘、电源、CPU、显卡、网卡

- 组合对象(Composite)：计算机、机箱、主板

- 部件(Component)：单一对象与组合对象的抽象接口

- 组合对象既可以包括单一对象，也可以包括组合对象

组合（Composite）模式的参与者



组合（Composite）模式的参与者

- Component
 - 为组合对象声明接口
 - 声明一个接口用于访问和管理Composite和Leaf
- Leaf
 - 表示叶节点对象，没有子节点
 - 定义基本对象的行为
- Composite
 - 定义组合对象的行为
 - 存储子部件
- Client
 - 通过Component接口操纵组合部件的对象

利用组合模式设计菜单



MenuComponent

getName()
getDescription()
getPrice()
isVegetarian()
print()
add(Component)
Remove(Component)
getChild(i)

MenuComponent
提供了这些方法的默
认行为

这些方法都是用来操作
MenuComponent，
Menu和MenuItem都是
MenuComponent

Menu对象包含多个
MenuComponent
对象

MenuItem

getName()
getDescription()
getPrice()
isVegetarian()
print()
~~add(Component)~~
~~Remove(Component)~~
getChild(i)

add
Remove
geChild
对菜单项没意义

Menu

getName()
getDescription()
~~getPrice()~~
~~isVegetarian()~~
print()
add(Component)
Remove(Component)
getChild(i)

getPrice
isVegetarian
对Menu没意义

```
public abstract class MenuComponent {  
    public void add(MenuComponent menuComponent) {  
        throw new UnsupportedOperationException();  
    }  
    public void remove(MenuComponent menuComponent) {  
        throw new UnsupportedOperationException();  
    }  
    public MenuComponent getChild(int i) {  
        throw new UnsupportedOperationException();  
    }  
    public String getName() {  
        throw new UnsupportedOperationException();  
    }  
    public String getDescription() {  
        throw new UnsupportedOperationException();  
    }  
    public double getPrice() {  
        throw new UnsupportedOperationException();  
    }  
    public boolean isVegetarian() {  
        throw new UnsupportedOperationException();  
    }  
    public void print() {  
        throw new UnsupportedOperationException();  
    }  
}
```

组合方法：新增、删除、取得
MenuComponent

操作方法，用在菜单项。
getName和getDescription
也可用在菜单上

因为有些方法只对菜单项有意义，有些方法只对菜单有意义，因此默认实现是都抛出异常。这样如果菜单项或菜单不支持某个操作，它们不需要做任何事情，直接继承默认实现就可。

print方法可以被应用到菜单项和菜单

```
public class MenuItem extends MenuComponent {
```

```
    String name;
```

```
    String description;
```

```
    boolean vegetarian;
```

```
    double price;
```

```
    public MenuItem(String name,String description,boolean vegetarian,double price) {
```

```
        this.name = name;
```

```
        this.description = description;
```

```
        this.vegetarian = vegetarian;
```

```
        this.price = price;
```

```
    }
```

```
    public String getName() { return name; }
```

```
    public String getDescription() { return description; }
```

```
    public double getPrice() { return price; }
```

```
    public boolean isVegetarian() { return vegetarian; }
```

```
    public void print() {
```

```
        System.out.print(" " + getName());
```

```
        if (isVegetarian()) {
```

```
            System.out.print("(v)");
```

```
        }
```

```
        System.out.print(", " + getPrice());
```

```
        System.out.print(" -- " + getDescription());
```

```
    }
```

```
}
```

实现菜单项

覆盖了抽象类的print方法。对菜单项，完整地打出所有条目

```

public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;    String description;
    public Menu(String name, String description) {
        this.name = name;    this.description = description;
    }
    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }
    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }
    public MenuComponent getChild(int i) {
        return (MenuComponent)menuComponents.get(i);
    }
    public String getName() { return name; }
    public String getDescription() { return description; }
    public void print() {
        System.out.print("\n" + getName() + " , " + getDescription());
        System.out.println("-----");
        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent = (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}

```

实现组合菜单

菜单可以有任意数目的孩子，
这些孩子都必须是
MenuComponent类型，
我们使用ArrayList来记录

将子菜单或菜单项加入到菜
单里，因为子菜单和菜单项
都是MenuComponent，
因此只要一个add方法就二
者兼顾
同理，也可以删除或取得某
个子菜单或菜单项

菜单只有名字和描述

首先打印菜单的名字
和描述

利用JDKAPI得到ArrayList的迭代器

利用迭代器，遍历每个MenuComponent。可
能是子菜单，可能是菜单项。二者都实现了print。
因此对遍历到的每个MenuComponent，直接
调用print。如果迭代到子菜单，则是递归调用

实现Waitress

```
public class Waitress {  
    MenuComponent allMenus;  
    public Waitress(MenuComponent allMenus) {  
        this.allMenus = allMenus;  
    }  
  
    public void printMenu() {  
        allMenus.print();  
    }  
}
```

只需要交给Waitress最顶层的菜单组件

只需要从顶层的菜单组件开始打印，就可以递归地打印出整个菜单层次

华中科技大学

```
public class MenuTestDrive {  
    public static void main(String args[]) {  
        MenuComponent pancakeHouseMenu =  
            new Menu("PANCAKE MENU", "Breakfast");  
        MenuComponent dinerMenu =  
            new Menu("DINER MENU", "Lunch");  
        MenuComponent cafeMenu =  
            new Menu("CAFE MENU", "Dinner");  
        MenuComponent dessertMenu =  
            new Menu("DESSERT MENU", "Dessert of course!");  
        MenuComponent allMenus =  
            new Menu("ALL MENUS", "All menus combined");
```

创建所有菜单对象

我们需要一个顶层菜单对象

```
allMenus.add(pancakeHouseMenu);  
allMenus.add(dinerMenu);  
allMenus.add(cafeMenu);
```

将子菜单加入到顶层菜单

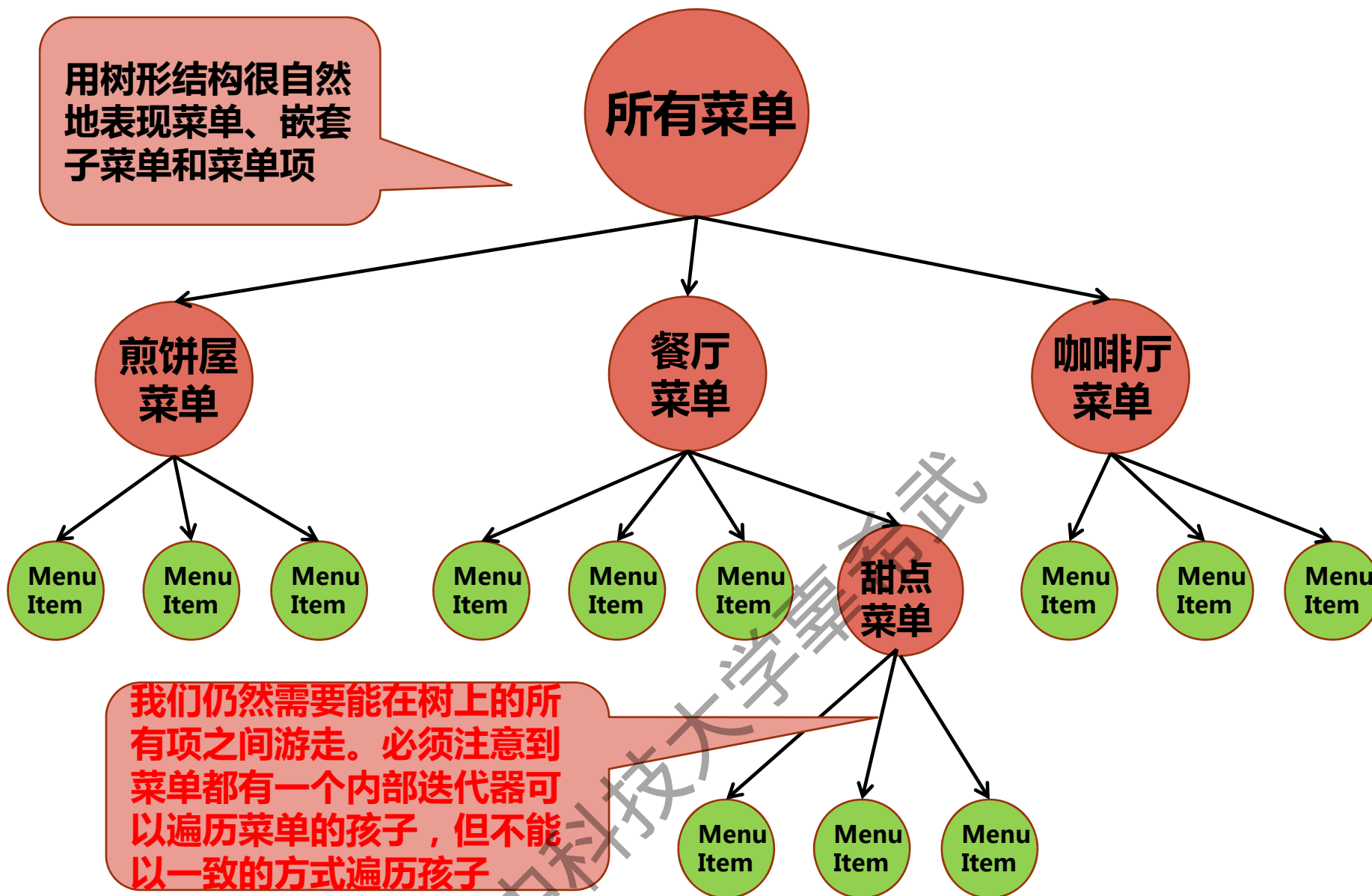
```
pancakeHouseMenu.add(new MenuItem("Pancake Breakfast", "With eggs", true, 2.99);  
dinerMenu.add(new MenuItem("Vegetarian BLT", "With tomato", true, 2.99));  
dessertMenu.add(new MenuItem("Apple Pie", "with a flakey crust", true, 1.59);  
cafeMenu.add(new MenuItem("Burger", "On a whole wheat ", true, 3.99));  
dinerMenu.add(dessertMenu);
```

在子菜单中加入下一级子菜单

在子菜单中加入菜单项

```
Waitress waitress = new Waitress(allMenus);  
waitress.printMenu();  
}  
}
```

用树形结构很自然地表现菜单、嵌套子菜单和菜单项



我们仍然需要能在树上的所有项之间游走。必须注意到菜单都有一个内部迭代器可以遍历菜单的孩子，但不能以一致的方式遍历孩子

组合菜单的内部迭代器

- 以DinnerMenu为例，对于一个DinnerMenu对象dinnerMenu，利用内部迭代器只能这样遍历孩子

利用JDKAPI得到ArrayList的迭代器，
为内部的迭代器

```
Iterator it =  
    dinnerMenu.menuComponents.iterator(); //假设menuComponents是公有的
```

```
while(it.hasNext()){  
    MenuComponent menuComponent = (MenuComponent)it.next();  
    if(menuComponent instanceof MenuItem){  
        //如果孩子是菜单项，做相应处理  
    }  
    if(menuComponent instanceof Menu) //如果孩子是子菜单  
        Iterator subIt =  
            ((Menu)menuComponent).menuComponents.iterator();  
        while(subIt.hasNext()){  
            //继续遍历子菜单  
        }  
    }  
}
```

因此，利用内部迭代器必须利用instanceof判定孩子是为叶子对象还是为下一级复合对象。无法以一致的方式来遍历组合对象

组合（Composite）迭代器的实现

- ❑ 要想实现组合迭代器(或者叫外部迭代器)，需要为每个Component都加上 createIterator方法
- ❑ 以MenuComponent为例

MenuComponent
<code>getName()</code> <code>getDescription()</code> <code>getPrice()</code> <code>isVegetarian()</code> <code>print()</code> <code>add(Component)</code> <code>Remove(Component)</code> <code>getChild(i)</code> <code>createIterator()</code>

在MenuComponent中加入createIterator方法，意味着每个菜单和菜单项都必须实现这个方法。也意味着，对一个组合对象调用该方法，将会应用于该组合对象的所有孩子

华中科技大学

组合（Composite）迭代器的实现

□ 首先修改菜单的实现

```
public class Menu extends MenuComponent {  
    ArrayList menuComponents = new ArrayList();  
    //其余部分不用修改  
    public Iterator createIterator(){  
        return new CompositeIterator(menuComponents.iterator());  
    }  
}
```

使用新的组合迭代器。
构造时，传入目前组合
的内部迭代器

□ 再修改菜单项的实现

```
public class MenuItem extends MenuComponent {  
    //其余部分不用修改  
    public Iterator createIterator(){  
        return new NullIterator();  
    }  
}
```

CompositeIterator, NullIterator都
实现Iterator接口

空迭代器的实现

- ❑ 选择一：菜单项的createIterator直接返回null

```
public class MenuItem extends MenuComponent {  
    //其他部分不用修改  
    public Iterator createIterator(){  
        return null;  
    }  
}
```

但客户代码必须判断createIterator返回值是否为null

- ❑ 选择二：返回一个空迭代器，但这个迭代器的hasNext()永远返回false

```
public class NullIterator extends Iterator{  
    public Object next() {  
        return null;  
    }  
    public boolean hasNext() {  
        return false;  
    }  
}
```

这是比较好的选择，它可以让客户代码以一致的方式来遍历，而无需判断createIterator返回值是否为null

组合（Composite）迭代器的实现

```
public class CompositeIterator implements Iterator {
```

```
    Stack stack = new Stack();
```

遍历过程中所有组合对象的迭代器放到堆栈里

```
    public CompositeIterator(Iterator iterator) { stack.push(iterator);}
```

```
    public Object next() {
```

```
        if (hasNext()) {
```

首先调用组合迭代器的hasNext()来确定是否还有下一个

构造时，传入顶层对象的迭代器并压入堆栈

```
            Iterator iterator = (Iterator) stack.peek();
```

得到栈顶的迭代器（不是弹出）

```
            MenuComponent component = (MenuComponent) iterator.next();
```

```
            if (component instanceof Menu) {
```

```
                stack.push(component.createIterator());
```

如果当前遍历的元素是子菜单，则把子菜单的迭代器压入堆栈

```
            }
```

```
            return component;
```

```
        } else {
```

```
            return null;
```

```
        }
```

```
    }
```

```
}
```

因此堆栈顶的迭代器始终是当前正在遍历的复合对象的迭代器。

组合（Composite）迭代器的实现

```
public class CompositeIterator implements Iterator {
```

```
//...
```

```
public boolean hasNext() {
```

```
    if (stack.empty()) {
```

```
        return false;
```

如果堆栈为空，没有任何迭代器，
则没有下一个元素

```
    } else {
```

```
        Iterator iterator = (Iterator) stack.peek();
```

得到栈顶的迭代器（不是弹出）

```
        if (!iterator.hasNext()) {
```

```
            stack.pop();
```

```
            return hasNext();
```

如果通过栈顶迭代器不能遍历到下一个元素，
则说明这个迭代器能遍历的元素全部遍历完毕。
因此将该迭代器弹出堆栈。同时递归调用hasNext

```
        } else {
```

```
            return true;
```

如果通过栈顶迭代器能遍历到下一个元素，
则返回true

```
        }
```

```
    }
```

```
}
```

```
}
```

因此堆栈顶的迭代器下的所有元素被遍历完毕，该迭代器会弹出堆栈。因此如果最后堆栈为空，则表示所有元素遍历完毕

现在可以利用组合迭代器一致地遍历对象

```
public class Waitress {  
    MenuComponent allMenus;  
    public Waitress(MenuComponent allMenus) {  
        this.allMenus = allMenus;  
    }  
  
    public void visitMenu() {  
        Iterator it = allMenus.createIterator(); //现在返回的是CompositeIterator对象  
        while(it.hasNext()){  
            MenuComponent menuComponent = (MenuComponent )it.next();  
            //...  
        }  
    }  
}
```

组合（Composite）模式的效果

- ❑ 定义了包含基本对象和组合对象的类层次结构
- ❑ 简化客户代码，一致使用组合对象和单个对象（实现了相同接口）
- ❑ 容易增加新的组合对象或基本对象到树中
- ❑ 用户使用Component类接口与组合结构中的对象进行交互。如果接收者是叶节点，直接处理请求；如果接收者是Composite，将请求发送给它的子部件，在转发请求之前和/或之后可能执行一些辅助操作

组合（Composite）模式的实现

- 显式的父对象的引用，在子对象中给出父对象的引用，可以很容易地遍历所有父对象
- 最大化Component接口
- 声明管理子部件的操作
- 组件的存储（什么样的数据结构？）

华中科技大学

结构型设计模式总结

□ Adapter 、 Facade

- Adapter用于两个**已有**的不兼容接口之间的转接

- Facade用于为复杂的子系统定义一个**新的**简单易用的接口

□ Composite、Decorator

- Composite用于构造对象(递归)组合结构

- Decorator用于为对象增加新的职责

行为型设计模式

华中科技大学

行为型模式的主要内容

- ❑ 模板模式(Template method Pattern)
- ❑ 观察者模式(Oberserver Pattern)
- ❑ 迭代器模式 (Iterator Pattern)
- ❑ 责任链模式 (Chain of Responsibility Pattern)
- ❑ 备忘录模式 (Memento Pattern)
- ❑ 策略模式 (Strategy Pattern)
- ❑ 命令模式 (Command Pattern)
- ❑ 状态模式 (State Pattern)
- ❑ 访问者模式 (Visitor Pattern)
- ❑ 中介者模式 (Mediator Pattern)
- ❑ 解释器模式 (Interpreter Pattern)

行为型设计模式

- ◆ 着力解决的是类实体之间的通讯关系，希望以面向对象的方式描述一个控制流程
 - ◆ 观察者（Observer）模式：定义了对象之间一对多的依赖，当这个对象的状态发生改变的时候，多个对象会接收到通知，有机会做出反馈
 - ◆ 迭代器（Iterator）模式：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

行为型设计模式

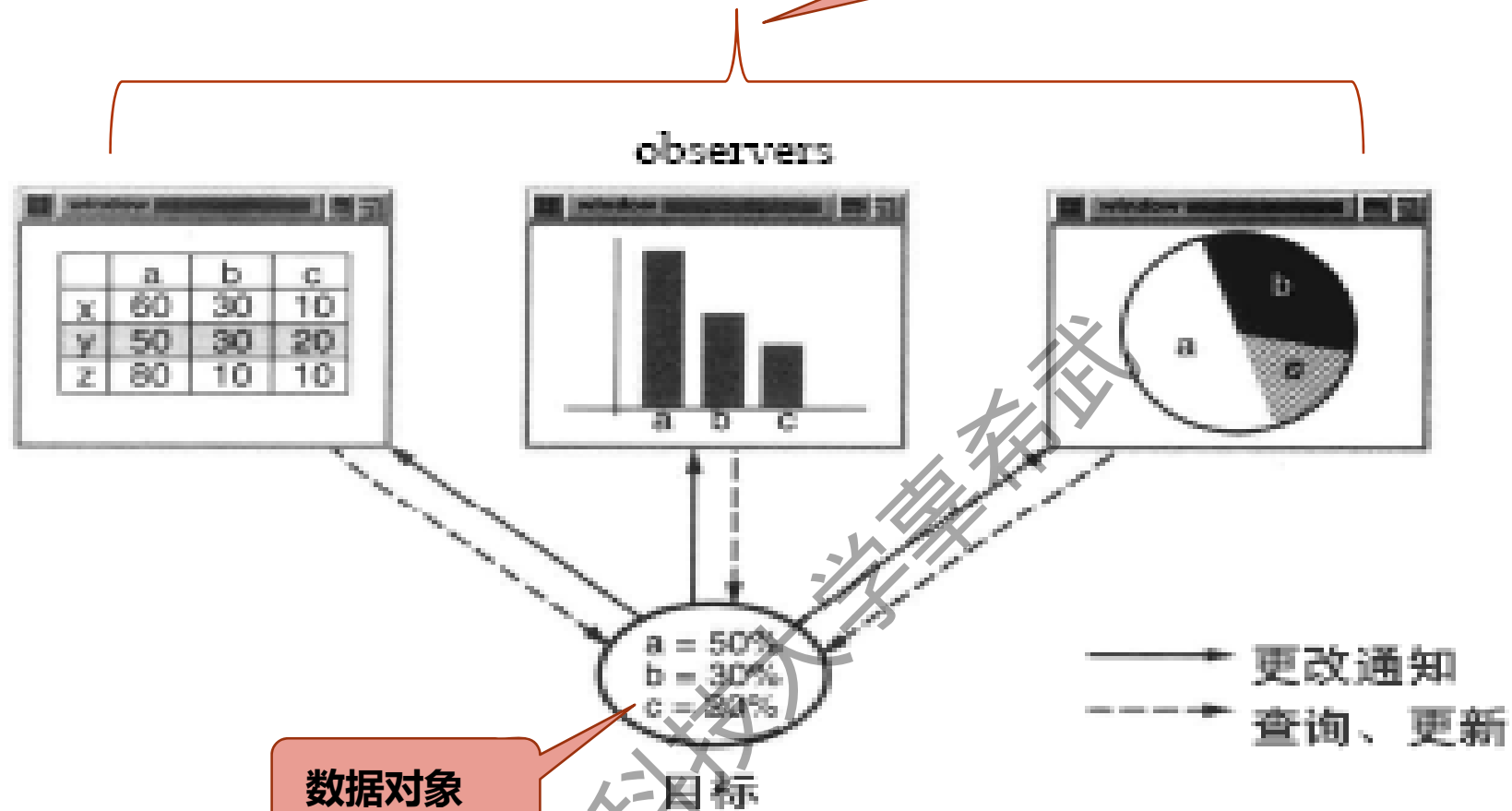
- ◆ 命令（Command）模式：将请求及其参数封装成一个对象，作为命令发起者和接收者的中介，可以对这些请求排队或记录请求日志，以及支持可撤销操作
- ◆ 策略（Strategy）模式：定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。策略模式使这些算法在客户端调用它们的时候能够互不影响地变化

观察者模式的由来

- 将系统分割成一系列相互协作的类有也存在一些不足：
 - 需要维护相关对象间的一致性
 - 为维持一致性而使各类紧密耦合，可能降低其可重用性
- 没有理由限定依赖于某数据对象的对象数目，对相同的数据对象可以有任意数目的不同用户界面

观察者模式的由来

对相同的数据对象，可以有不同的展示方式（不同的观察者）



观察者模式的由来

- ❑ 某气象站通过湿度感应装置、温度感应装置、气压感应装置获取天气情况
- ❑ 实现了一WeatherData类，可以从气象站获得湿度、温度、气压数据
- ❑ 需要根据WeatherData提供的气象数据实时更新三个布告板：目前天气状况、气象统计和天气预报
- ❑ 系统必须是可扩展的，即未来可以任意加入、删除新的布告板

湿度感应器

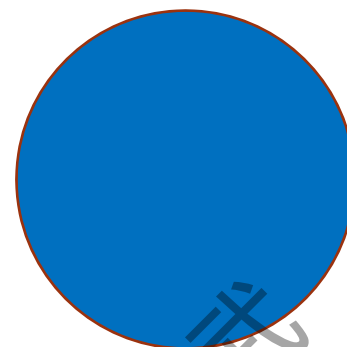
温度感应器

气压感应器



气象站

获取数据



WeatherData
对象

显示

目前状况
公告板

气象统计
公告板

天气预报
公告板

未来新的
公告板

观察者模式的由来

□ WeatherData类应该具有的方法

WeatherData
getTemperature() getHumidity() getPressure() measurementsChanged()

三个getter方法可以取得三个测量值：温度、湿度、气压

一旦测量值被更新，此方法被调用去更新三个布告板

华中科技大学

观察者模式的由来

□ WeatherData类的第一个实现

```
public class WeatherData {
```

```
//实例变量声明
```

```
public void measurementsChanged(){
```

```
    float temp = getTemperature();
```

```
    float humidity = getHumidity();
```

```
    float pressure = getPressure();
```

```
    currentConditionDisplay.update(temp,humidity,pressure);
```

```
    statisticsDisplay.update(temp,humidity,pressure);
```

```
    forecastDisplay.update(temp,humidity,pressure);
```

```
}
```

```
//其他方法
```

```
}
```

获得最新的测量值

更新三个公告板

针对公告板的具体实现编程
以后增加、删除公告板要修改代码，同时无法在运行时增加、删除公告板

update方法看起来像统一接口

认识观察者模式

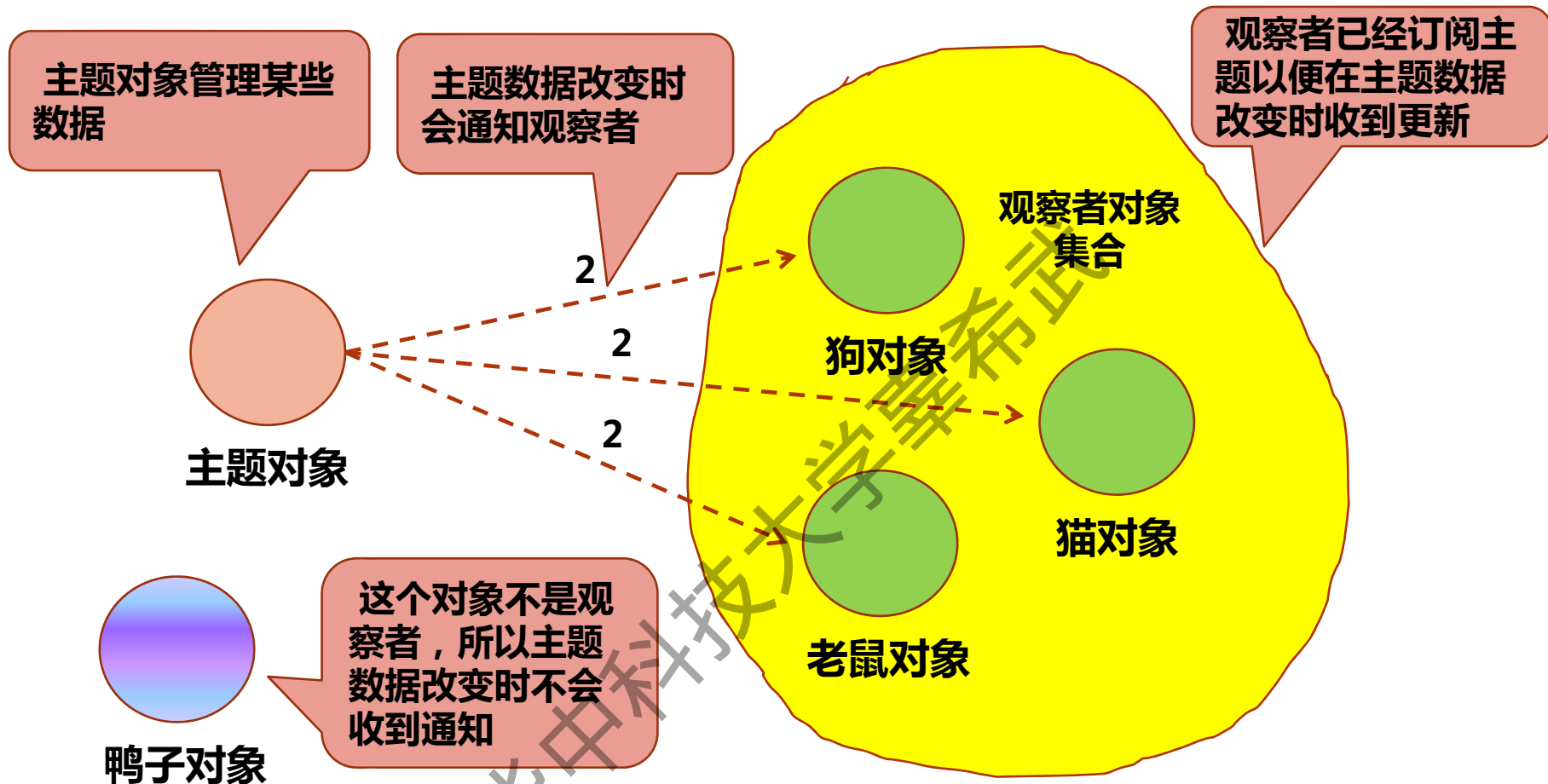
□ 看看报纸和杂志的订阅

- 用户向报社订阅报纸
- 报社只要有新报纸出版，就会给你送来。只要你一直是订户，你就会一直收到报纸
- 你不想看报纸时，可以取消订阅。报社不会再给你送来新报纸
- 只要报社还在运营，就会一直有人向报社订阅或取消订阅

华中科技大学

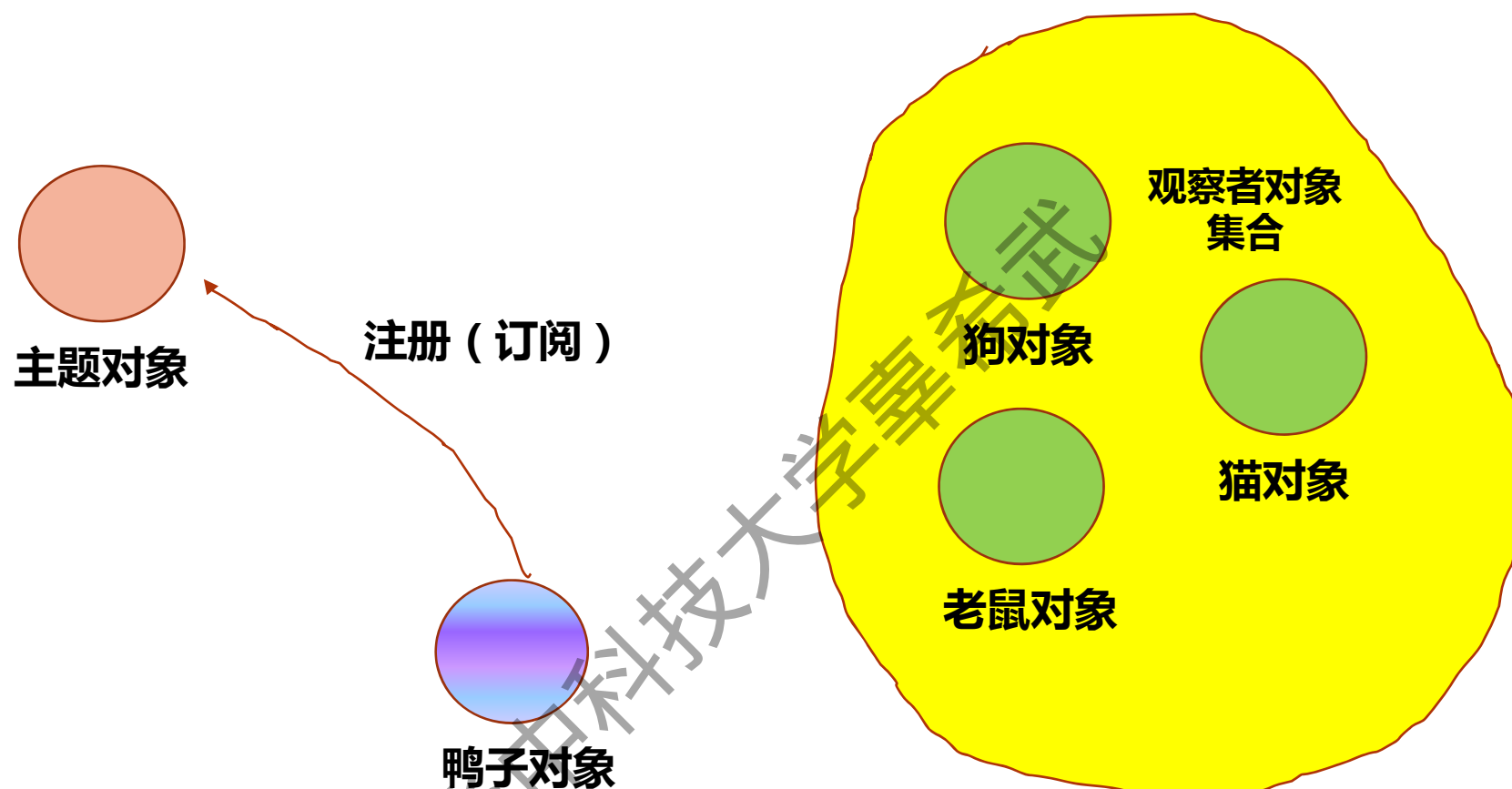
出版者+订阅者=观察者模式

□ 在观察者模式里，出版者叫“主题”（Subject），订阅者叫“观察者”（Observer）



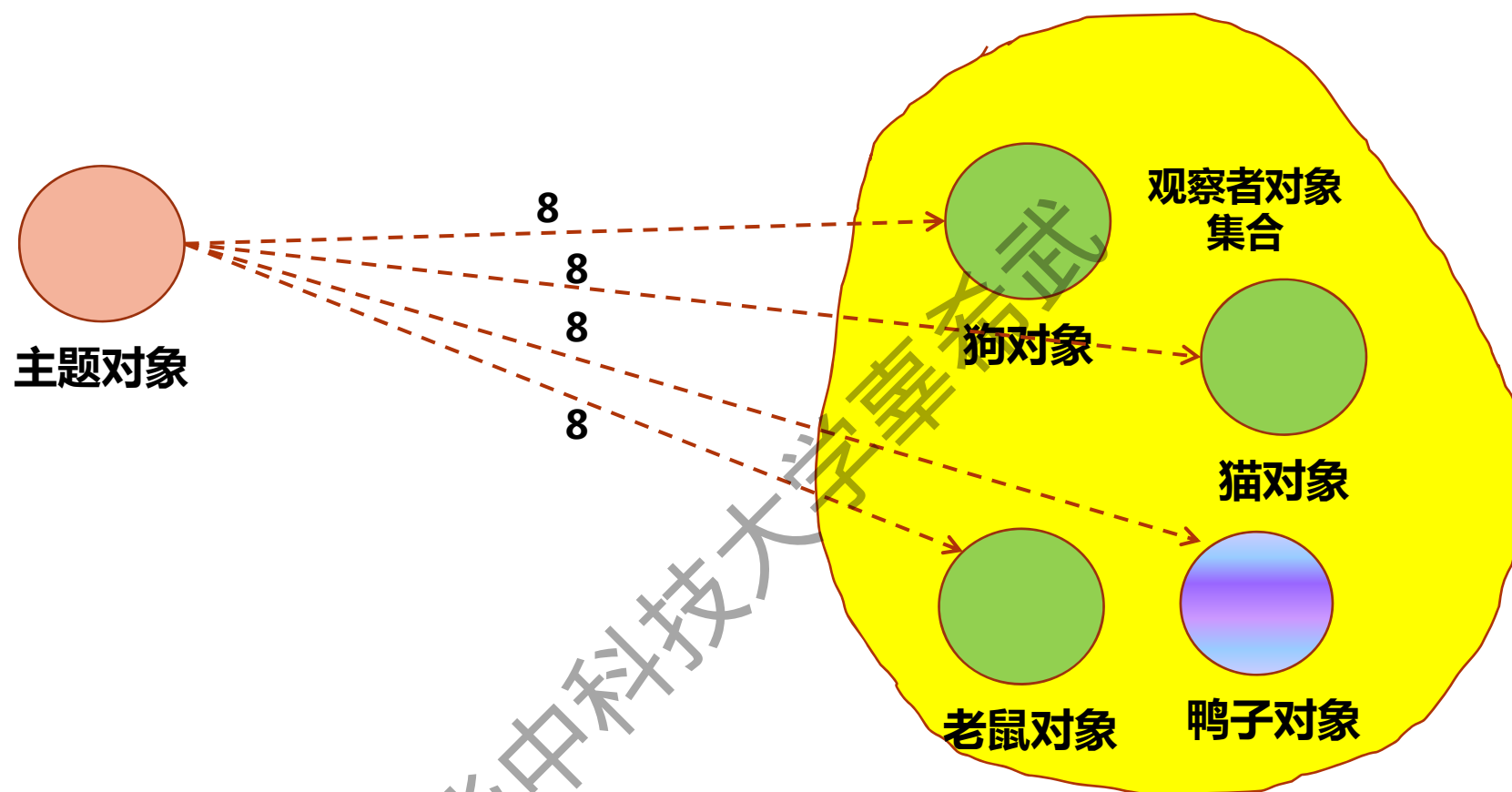
出版者+订阅者=观察者模式

□ 鸭子对象想成为观察者，首先要向主题对象注册，注册后成为观察者



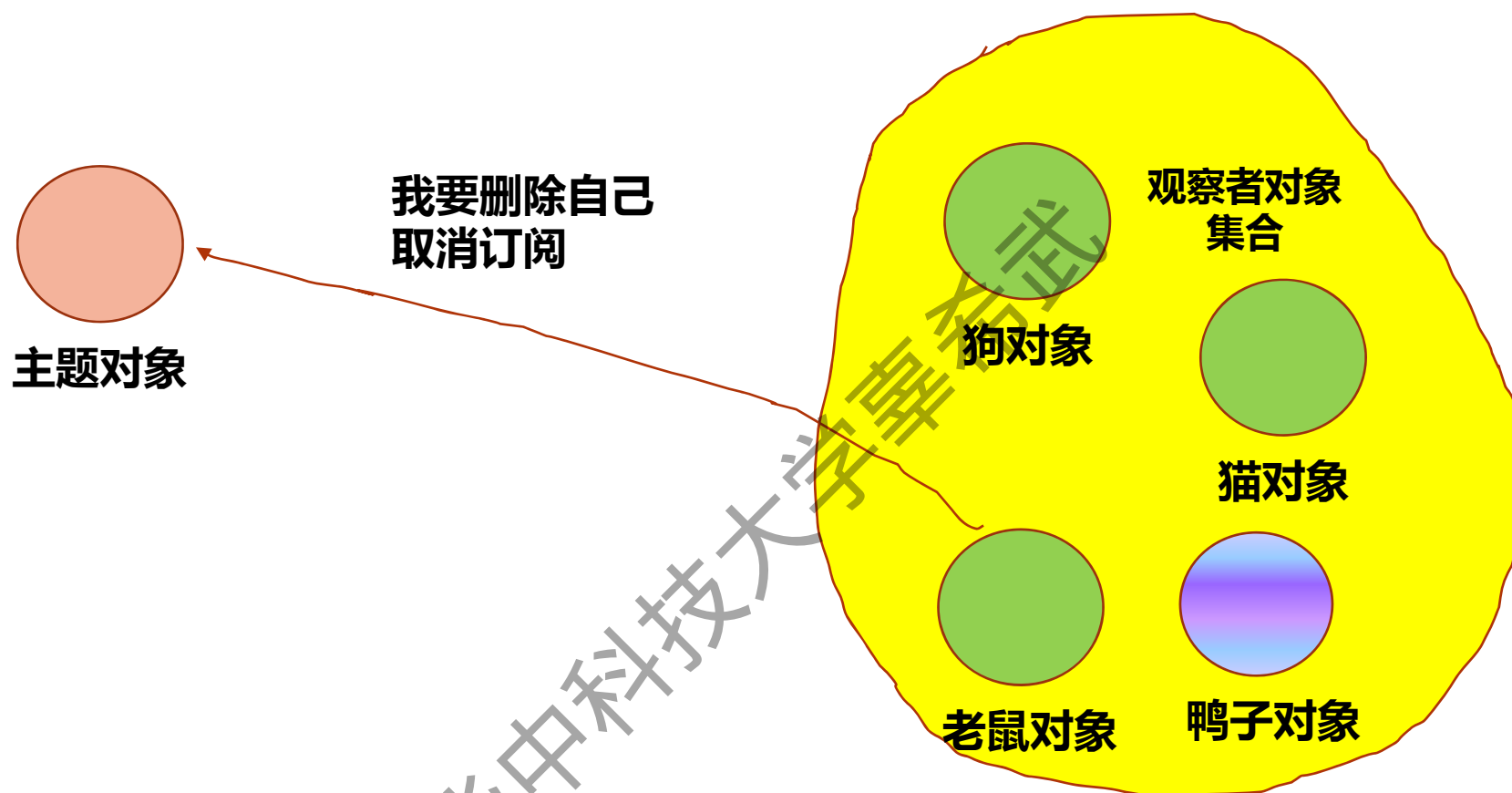
出版者+订阅者=观察者模式

□ 主题有了新数据，会通知所有观察者



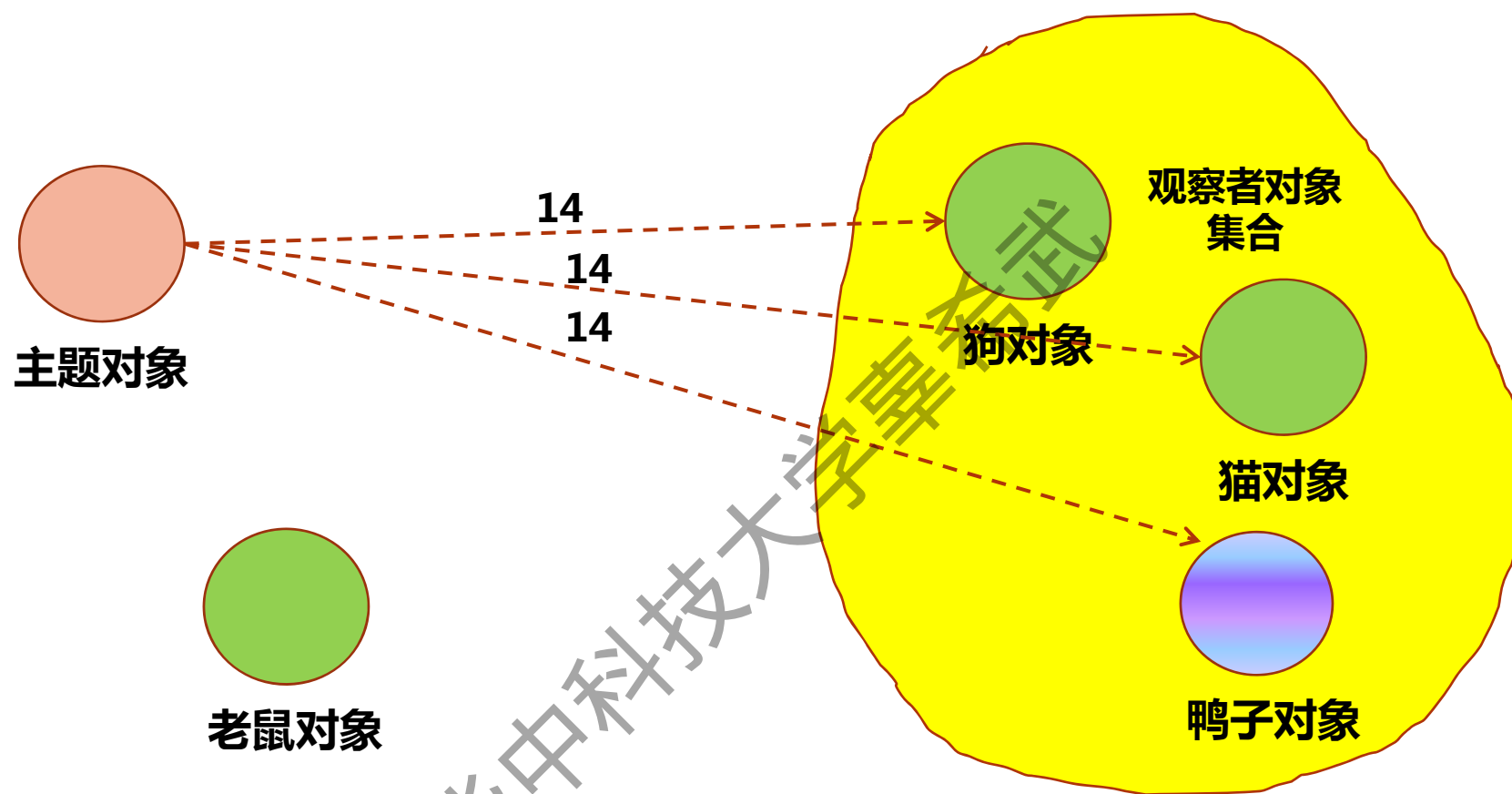
出版者+订阅者=观察者模式

□ 老鼠对象厌倦了，要求从观察者中把自己除名



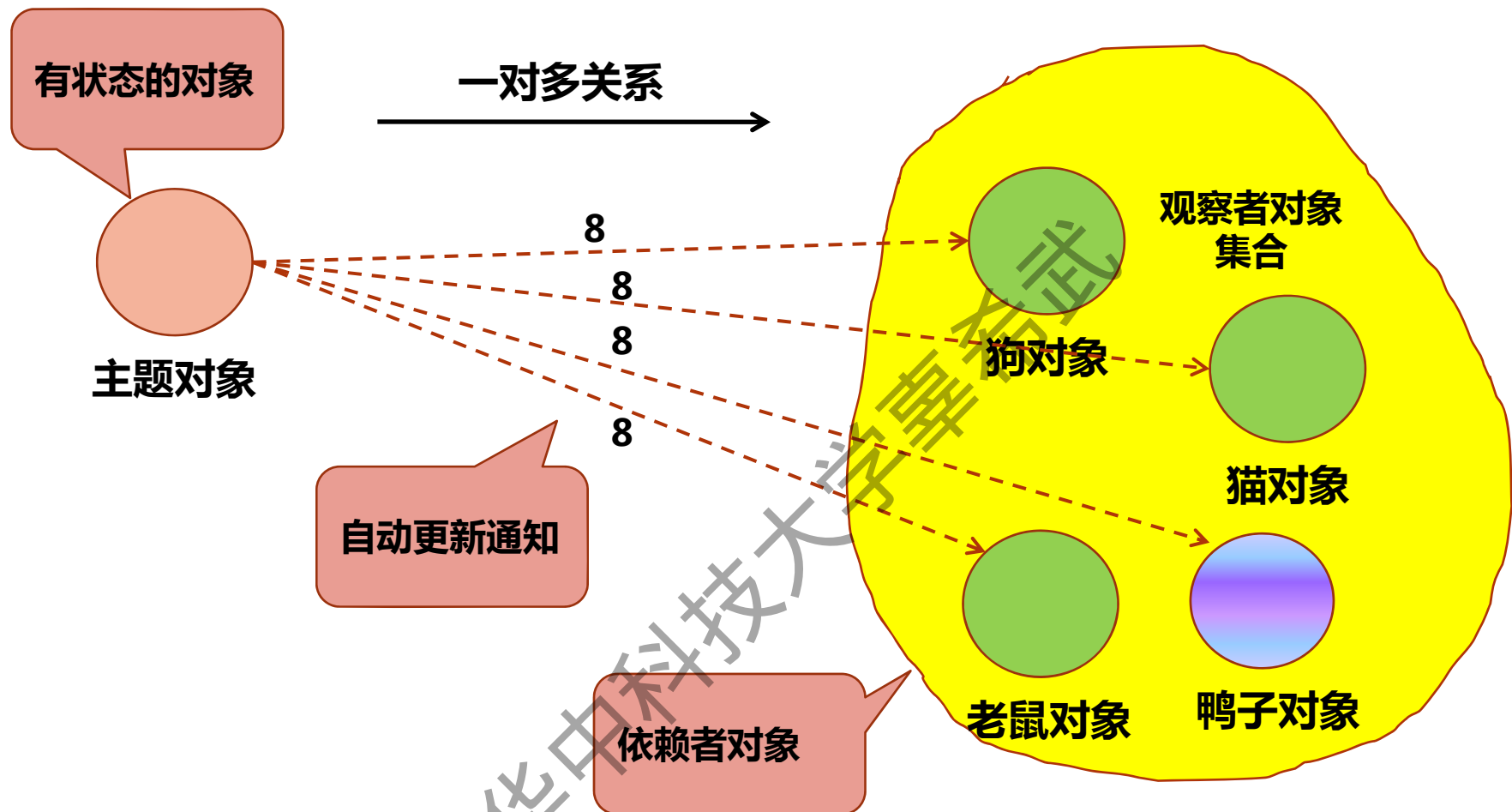
出版者+订阅者=观察者模式

□ 主题又有了新数据，会通知当前剩下的所有观察者



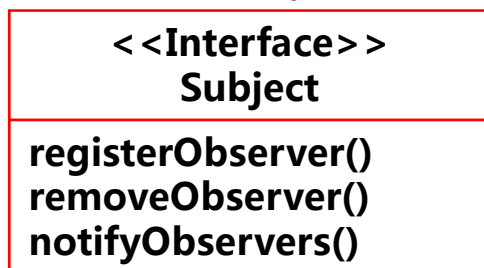
定义观察者模式 (Observer Pattern)

- 观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新



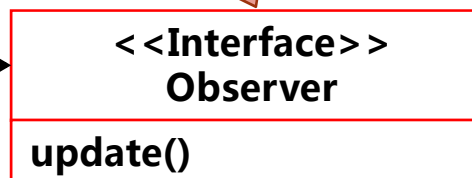
观察者模式的结构

主题接口，对象利用此接口注册为观察者，或者把自己从观察者中删除



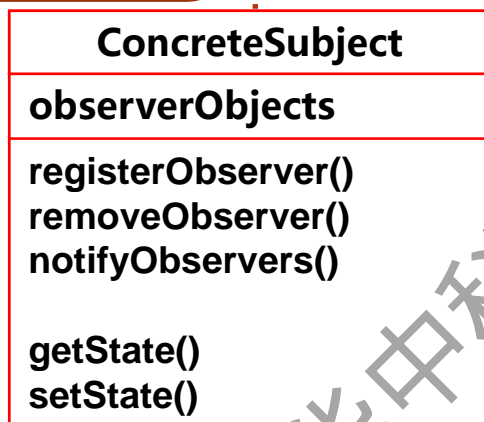
许多观察者

观察者接口，只有update方法，当主题状态改变时被调用

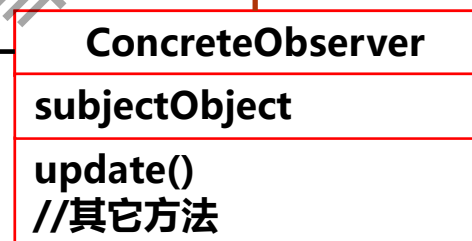


一个主题有许多观察者

具体的主题。有一个观察者列表。还实现了notifyObservers方法，当自己状态改变时通知所有观察者



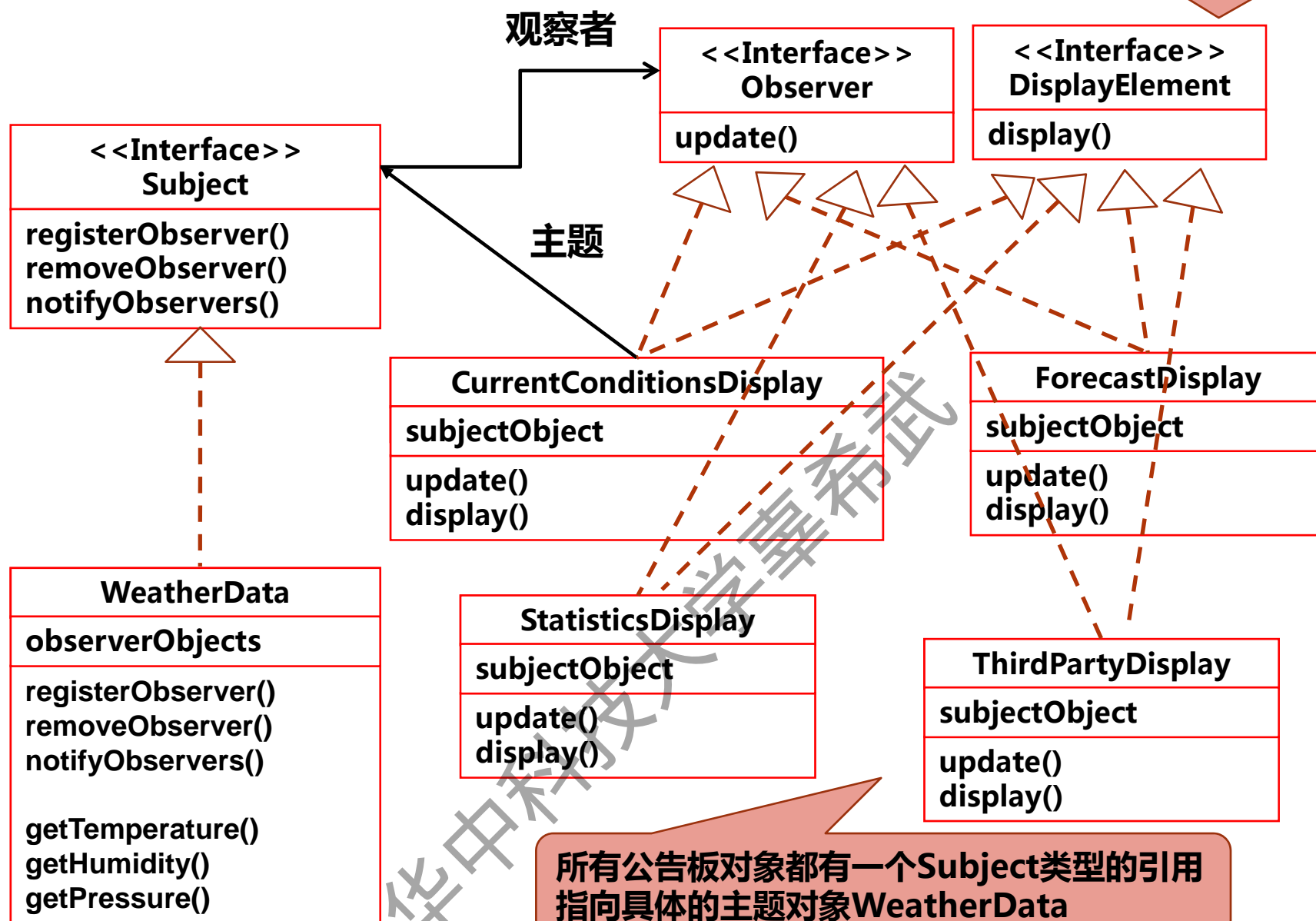
主题



具体的观察者有一个subject对象的引用，通过该引用注册或删除自己，并给出update的具体实现

设计气象站

为公告板设计接口，定义display()方法



实现气象站

```
public interface Subject {  
    public void registerObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
}
```

这二个方法需要Observer接口类型的变量，传入需要注册或删除的具体观察者对象

当主题状态改变时，这个方法被调用以通知所有的观察者

```
public interface Observer {  
    public void update(float temp, float humidity, float pressure);  
}
```

当主题状态改变时，该方法会被调用以把温度、湿度、气压传递给观察者

```
public interface DisplayElement {  
    public void display();  
}
```

当公告板需要显示时，调用该方法

实现气象站

```
public class WeatherData implements Subject {  
    private ArrayList observers;
```

主题对象将所有注册的观察者对象保持在ArrayList中

```
    private float temperature , humidity , pressure;
```

```
    public float getTemperature() { return temperature; }  
    public float getHumidity() { return humidity; }  
    public float getPressure() { return pressure; }
```

```
    public WeatherData() {  
        observers = new ArrayList();  
    }
```

```
    public void registerObserver(Observer o) {  
        observers.add(o);  
    }
```

当注册观察者对象时，将其加入到ArrayList中

```
    public void removeObserver(Observer o) {  
        int i = observers.indexOf(o);  
        if (i >= 0) {  
            observers.remove(i);  
        }  
    }  
}
```

当删除观察者对象时，将其从ArrayList中删除

实现气象站（续）

```
public class WeatherData implements Subject {  
    //...  
    public void notifyObservers() {  
        for (int i = 0; i < observers.size(); i++) {  
            Observer observer = (Observer)observers.get(i);  
            observer.update(temperature, humidity, pressure);  
        }  
    }  
    public void measurementsChanged() {  
        notifyObservers();  
    }  
    public void setMeasurements(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        this.pressure = pressure;  
        measurementsChanged();  
    }  
}
```

遍历ArrayList中的所有观察者对象，调用他们的update方法，将新的状态通知他们。注意每个观察者对象都继承了Observer接口，因此都实现了update方法

当测量值变化时，调用notifyObservers通知所有的观察者

模拟读取传感器数据的动作，用来测试公告板。

华中科技大学

实现公告板

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {  
    private float temperature;  
    private float humidity;  
    private Subject weatherData;  
  
    public CurrentConditionsDisplay(Subject weatherData) {  
        this.weatherData = weatherData;  
        weatherData.registerObserver(this);  
    }  
    public void update(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        display();  
    }  
    public void display() {  
        System.out.println("Current conditions: " + temperature  
            + "F degrees and " + humidity + "% humidity");  
    }  
}
```

观察者对象保存一个Subject类型的引用，指向主题对象

构造函数传入主题对象，并将自己向主题对象注册

当update方法被调用时，保存温度和湿度，并调用display()以更新公告板的显示

为什么要保存Subject引用？构造函数里注册了似乎用不着了。
可能以后需要取消注册，保留该引用比较方便

启动气象站

```
public class WeatherStation {  
    public static void main(String[] args) {  
        WeatherData weatherData = new WeatherData();  
  
        CurrentConditionsDisplay currentDisplay =  
            new CurrentConditionsDisplay(weatherData);  
  
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);  
  
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);  
  
        weatherData.setMeasurements(80, 65, 30.4f);  
        weatherData.setMeasurements(82, 70, 29.2f);  
        weatherData.setMeasurements(78, 90, 29.2f);  
    }  
}
```

观察者获取主题对象数据的二种方式

- ❑ 由主题对象在状态发生改变时，将新的数据 “推送” (Push) 到观察者（目前气象站就采用这种方式，通过调用观察者的 update 方法）。这种做法的缺点：
 - ❑ 有时观察者并不需要所有的数据，而只是需要其中一部分（如 **CurrentConditionsDisplay** ）。但 “推送” 则将所有数据全部推送到观察者。
 - ❑ 如果主题对象状态变化十分频繁（比如气象站温度每变化0.01度就会更新主题对象的状态），导致频繁地将数据发送到观察者。
- ❑ 主题对象提供getter方法让观察者读取状态，同时主题对象在状态改变时只 “通知” 观察者数据改变了，但不将数据推送到观察者，而是由观察者调用主题对象提供的getter方法将数据从主题对象 “拉” (Pull) 到观察者这边。

JAVA内置的观察者模式

- ❑ java.util包内定义了Observer接口和Observable类（可观察者）
- ❑ 支持pull和push二种方式

java.util.Observable
不是接口，而是类。给出了这些方法的实现

java.util.Observer接口

Observable

addObserver()
deleteObserver()
notifyObserver()
setChanged()

<<Interface>>
Observer

update()

JAVA内置的观察者模式

- ❑ 要实现具体的观察者类，只要实现java.util.Observer接口。需要注册时，调用任何Observable对象的addObserver方法。不想再当观察者时，调用deleteObserver方法
- ❑ 要实现可观察者类（即主题），首先要继承java.util.Observable类
- ❑ 可观察者要发送通知，需要二个步骤
 - ❑ 先调用setChanged方法，标记状态已经改变的事实
 - ❑ 调用二个方法中的一个
 - ❑ notifyObservers() //只通知观察者，但没携带新数据
 - ❑ notifyObservers(Object arg) //携带新数据
- ❑ 观察者的update方法签名为 void update(Observable o, Object arg)

第一个参数是主题对象，
好让观察者知道是哪个
主题通知的

第二个参数是携带数据
的对象。可以为null

JAVA内置的观察者模式内幕

❑ setChanged方法如何工作?以下是java.util.Observable内部的代码

```
setChanged(){  
    changed = true;  
}
```

当setChanged被调用，
内部的changed标志
被设为true

```
notifyObservers(Object arg){  
    if(changed){  
        for every observer on the list {  
            call update(this, arg);  
        }  
        changed = false;  
    }  
}
```

notifyObservers()只
会在changed标志位
true时通知观察者
推模式

```
notifyObservers(){  
    notifyObservers(null);  
}
```

通知观察者后把changed标
志设为false
拉模式

JAVA内置的观察者模式内幕

- `setChanged`方法使得在更新观察者时有更多的弹性
 - 你可以更适当地通知观察者
 - 例如没有`setChanged`方法，气象站测量可能非常敏感，以至于温度计读数每十分之一度就会更新。如果我们希望半度以上才更新，就可以在温度差距超过半度时，调用`setChanged()`方法进行有效地更新

基于Observable实现气象站

```
public class WeatherData extends Observable {  
    private float temperature;  
    private float humidity;  
    private float pressure;  
    public WeatherData() { }  
    public void measurementsChanged() {  
        setChanged();  
        notifyObservers();  
    }  
    public void setMeasurements(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        this.pressure = pressure;  
        measurementsChanged();  
    }  
    public float getTemperature() {  
        return temperature;  
    }  
    public float getHumidity() {  
        return humidity;  
    }  
    public float getPressure() {  
        return pressure;  
    }  
}
```

不在需要追踪观察者了，也不需要管理注册、删除观察者，超类全部做好了

注意notifyObservers()方法不带任何参数，意味着是在采用拉模式

这些getter方法是为了支持拉模式：由观察者调用以“拉”数据

重新实现公告板（观察者）

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {  
    Observable observable;  
    private float temperature;  
    private float humidity;  
    public CurrentConditionsDisplay(Observable observable) {  
        this.observable = observable;  
        observable.addObserver(this);  
    }  
    public void update(Observable obs, Object arg) {  
        if (obs instanceof WeatherData) {  
            WeatherData weatherData = (WeatherData)obs;  
            this.temperature = weatherData.getTemperature();  
            this.humidity = weatherData.getHumidity();  
            display();  
        }  
    }  
    public void display() {  
        System.out.println("Current conditions: " + temperature  
            + "F degrees and " + humidity + "% humidity");  
    }  
}
```

实现的是java.util.Observer接口

构造函数传入可观察者对象，并将自己注册到可观察者对象

实现java.util.Observer接口定义的update方法。采用拉模式主动从可观察者对象获取数据忽略arg参数，因此是拉模式

华中科技大学

java.util.Observable 的黑暗面

- ❑ 是一个类，因此你必须继承它。但无法继承其它类
- ❑ 因为没有java.util.Observable接口，因此你无法建立自己的实现和Java内置的Observer接口搭配使用
- ❑ setChanged是保护方法。这意味着什么？意味着你只有继承Observable类，而无法创建Observable实例并组合到自己的对象中来。违反了“多用组合、少用继承”原则

在JDK中，还有哪些地方可以找到观察者模式

□ Java Swing是图形界面编程API，其事件处理就用到了观察者模式

```
public class SwingObserverExample {  
  
    public static void main(String[] args) {  
        JFrame frame = new JFrame();  
        JButton button = new JButton("Should I do it?");  
        button.addActionListener(new AngelListener());  
        button.addActionListener(new DevilListener());  
        frame.getContentPane().add(BorderLayout.CENTER, button);  
        //设置frame其他属性  
    }  
    class AngelListener implements ActionListener {  
        public void actionPerformed(ActionEvent event) {  
            System.out.println("Don't do it, you might regret it!");  
        }  
    }  
    class DevilListener implements ActionListener {  
        public void actionPerformed(ActionEvent event) {  
            System.out.println("Come on, do it!");  
        }  
    }  
}
```

JButton对象就是可观察者对象

产生二个观察者对象，并注册到button上

二个观察者类定义，实现ActionListener接口

观察者模式的意图和适用性

□ **意图**：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改变时，所有依赖于它的对象都得到通知并被自动更新

□ 适用场合

- 当一个抽象模型有两个方面, 一方面依赖于另一方面。将二者封装在独立对象中以使它们可以独立被改变和复用
- 一个对象的改变需要同时改变其它对象, 但不知道具体有多少对象有待改变
- 当一个对象必须通知其它对象, 而它又不能假定其它对象是谁。换言之, 不希望这些对象是紧密耦合的

观察者模式的参与者

- ❑ Subject：抽象的主题，即被观察的对象
- ❑ Observer：抽象的观察者
- ❑ Concrete Subject：具体被观察对象
- ❑ Concrete Observer：具体的观察者
- ❑ 注意：在观察者模式中，Subject通过addObserver和removeObserver方法添加或删除所关联的观察者，并通过notifyObservers进行更新，让每个观察者观察到最新的状态

观察者模式分析

□ 应用场景

- 对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变
- 对象仅需要将自己的更新通知给其他对象而不需要知道其他对象细节

□ 优点

- Subject和Observer之间是松耦合的，可以各自独立改变
- Subject在发送广播通知时，无须指定具体的Observer，Observer可以自己决定是否要订阅Subject的通知
- 高内聚、低耦合

□ 缺陷

- 松耦合导致代码关系不明显，有时可能难以理解
- 如果一个Subject被大量Observer订阅的话，在广播通知的时候可能会有效率问题

命令模式的由来

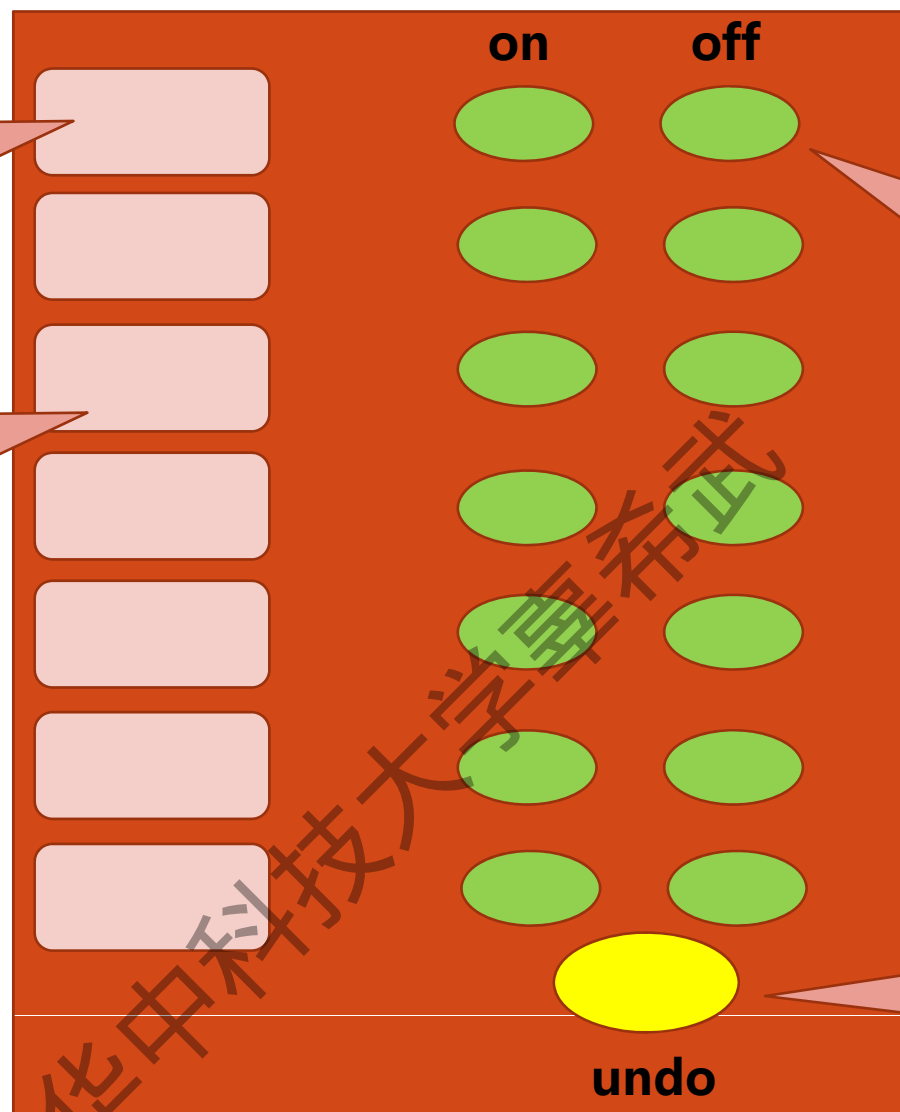
□ 实现一个家电装置的遥控器

遥控器有7个可编程的插槽，每个插槽对应不同的家电装置

可编程的插槽可以随时替换指定插槽对应的家电

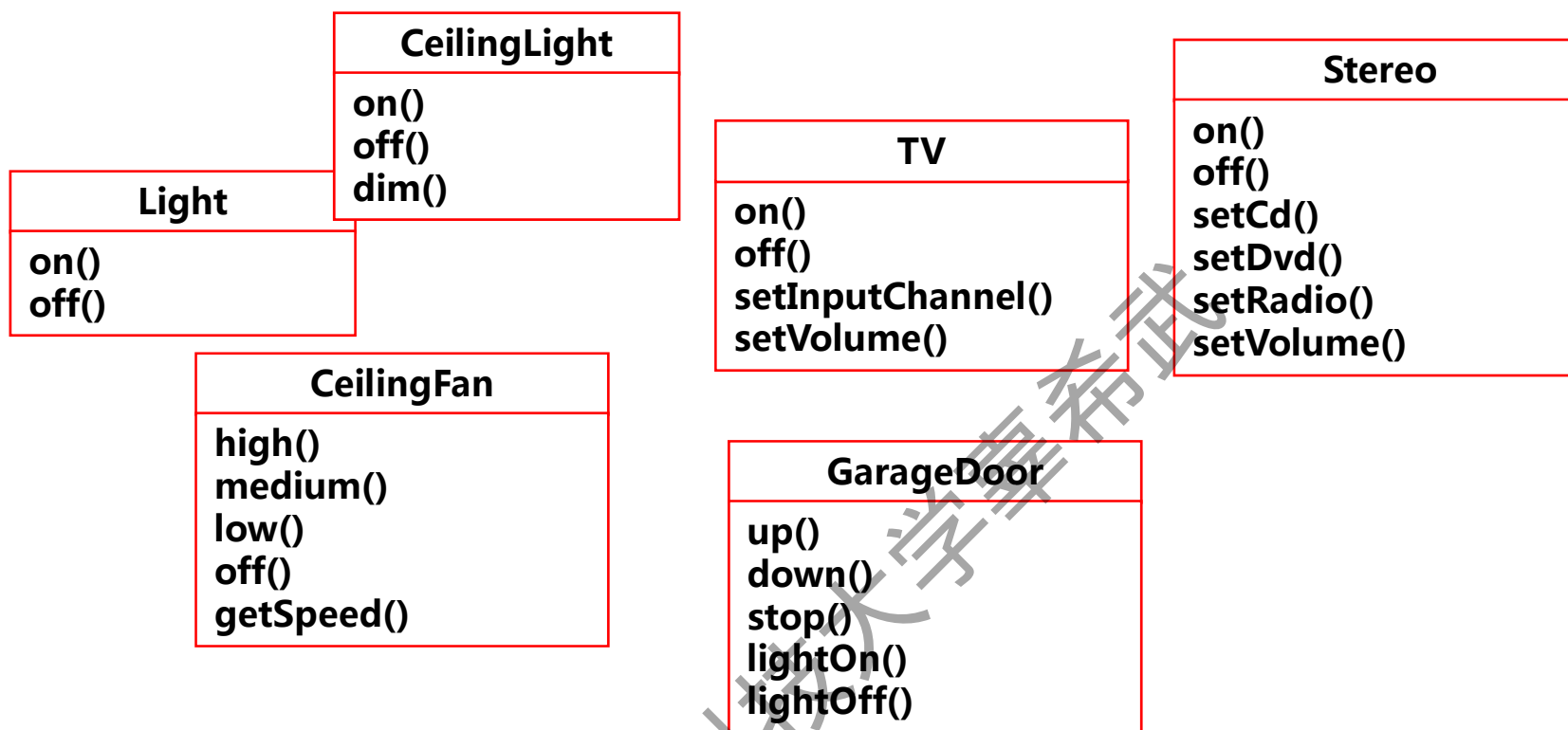
7个插槽具备各自的“开”、“关”按钮

共用的“撤销”按钮，会撤销最后一个按钮的动作



命令模式的由来

□ 有如下一些已经实现的家电类



不同的家电类接口各有差异，以后可能会有新的家电类。因此设计遥控器具有挑战性：遥控器插槽插入新的家电也要能工作

命令模式的由来

□ 遥控器设计要点

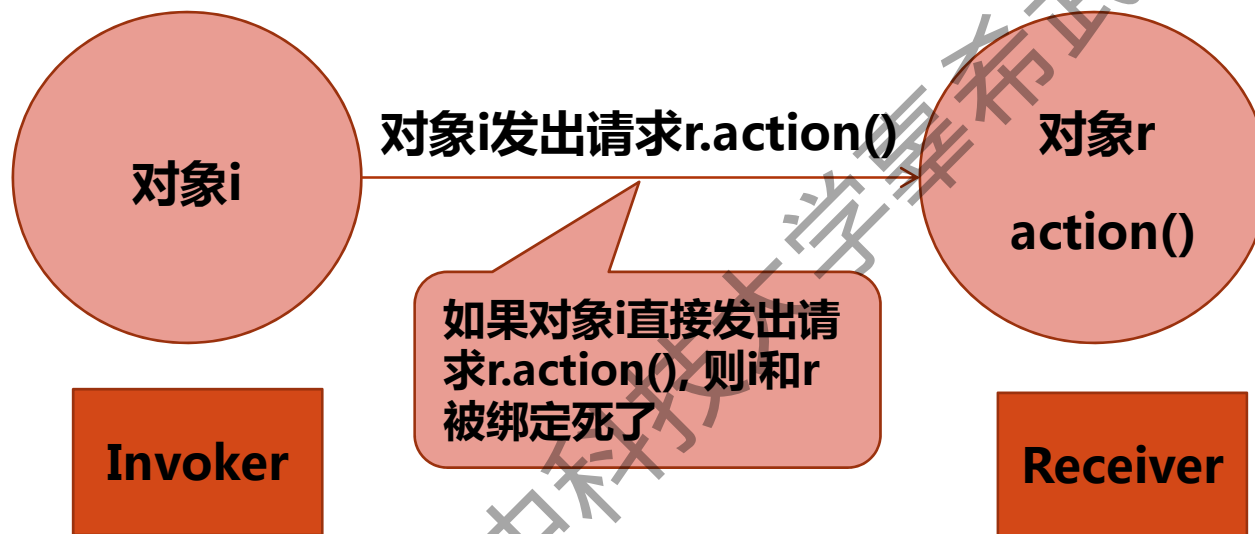
- 遥控器要知道如何解读按钮被按下的动作，然后发出正确的请求。但遥控器不能知道这些家电自动化的细节。否则遥控器就被绑定的具体的家电对象上。
- 因此不能让遥控器包含一大堆if语句，如 “if slot1 == Light , then light.on(), else if slot1 == TV, then tv.on()” 。这样的设计很糟糕。这种设计导致新的家电插进插槽，就得修改遥控器的代码，而且工作没完没了。

命令模式的由来

- 需要将“动作的请求者”（遥控器）和“动作的执行者”（家电对象）解耦。
 - 看起来很困难，毕竟当按下遥控器按钮时，必须打开电灯，即在遥控器代码中调用`light.on()`方法。
 - 采用“命令模式”可以做到：把动作请求（如打开电灯）封装成一个“命令对象”。所有如果每个插槽都存储一个命令对象，那么当对应按钮被按下时，可以请命令对象做相关的工作。遥控器并不需要动作如何执行，只要知道有个命令对象能和正确的家电对象沟通。因此，遥控器和家电对象（如电灯）解耦了。

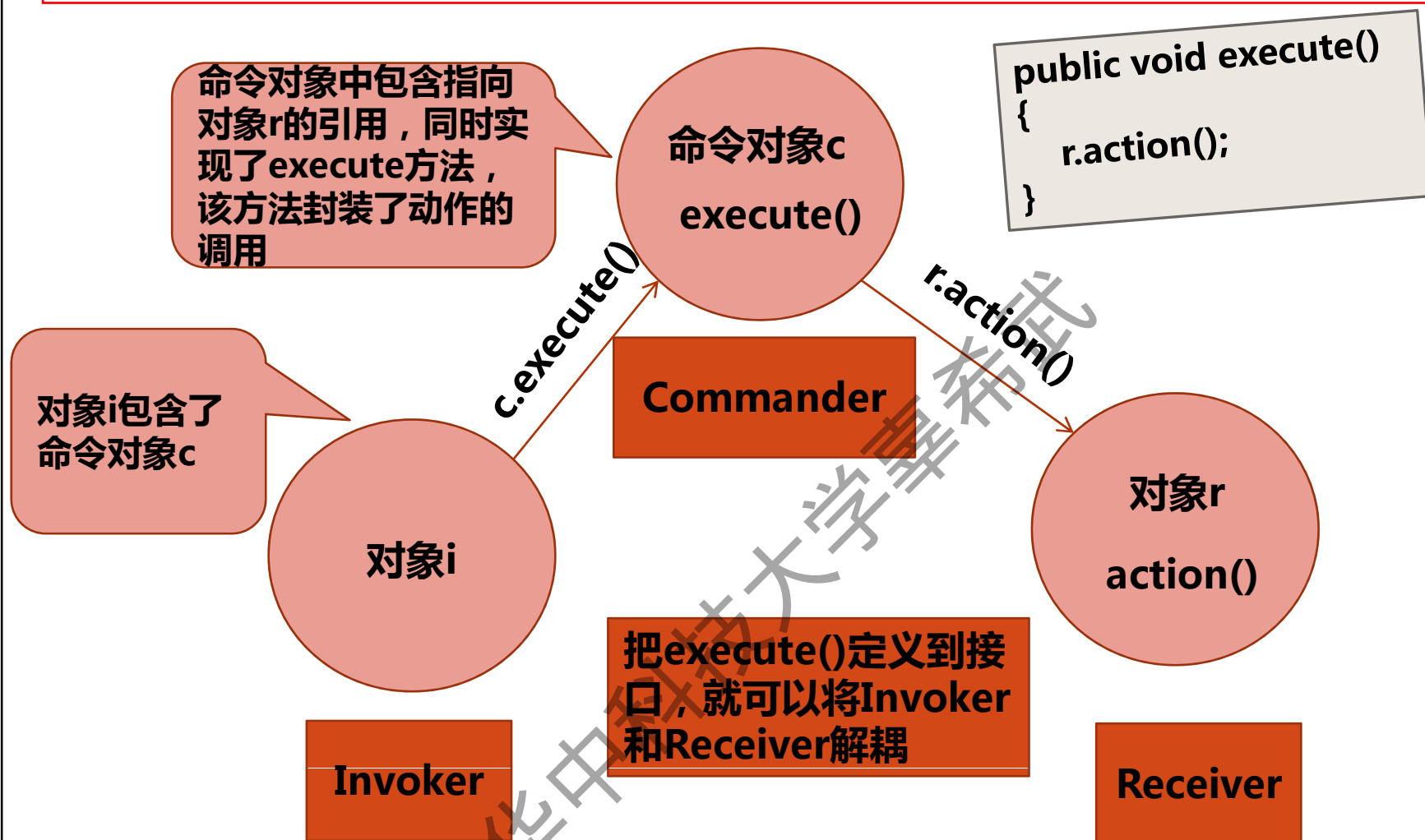
命令模式的由来

- ❑ 二个概念：动作的请求者(Invoker)和动作的接受者(Receiver)
- ❑ 假设某个对象r具有某个方法（或叫动作）action()，如果在另外一个对象i的某个方法中调用r.action(); 则对象i称为动作的请求者，对象r称为动作的接受者。

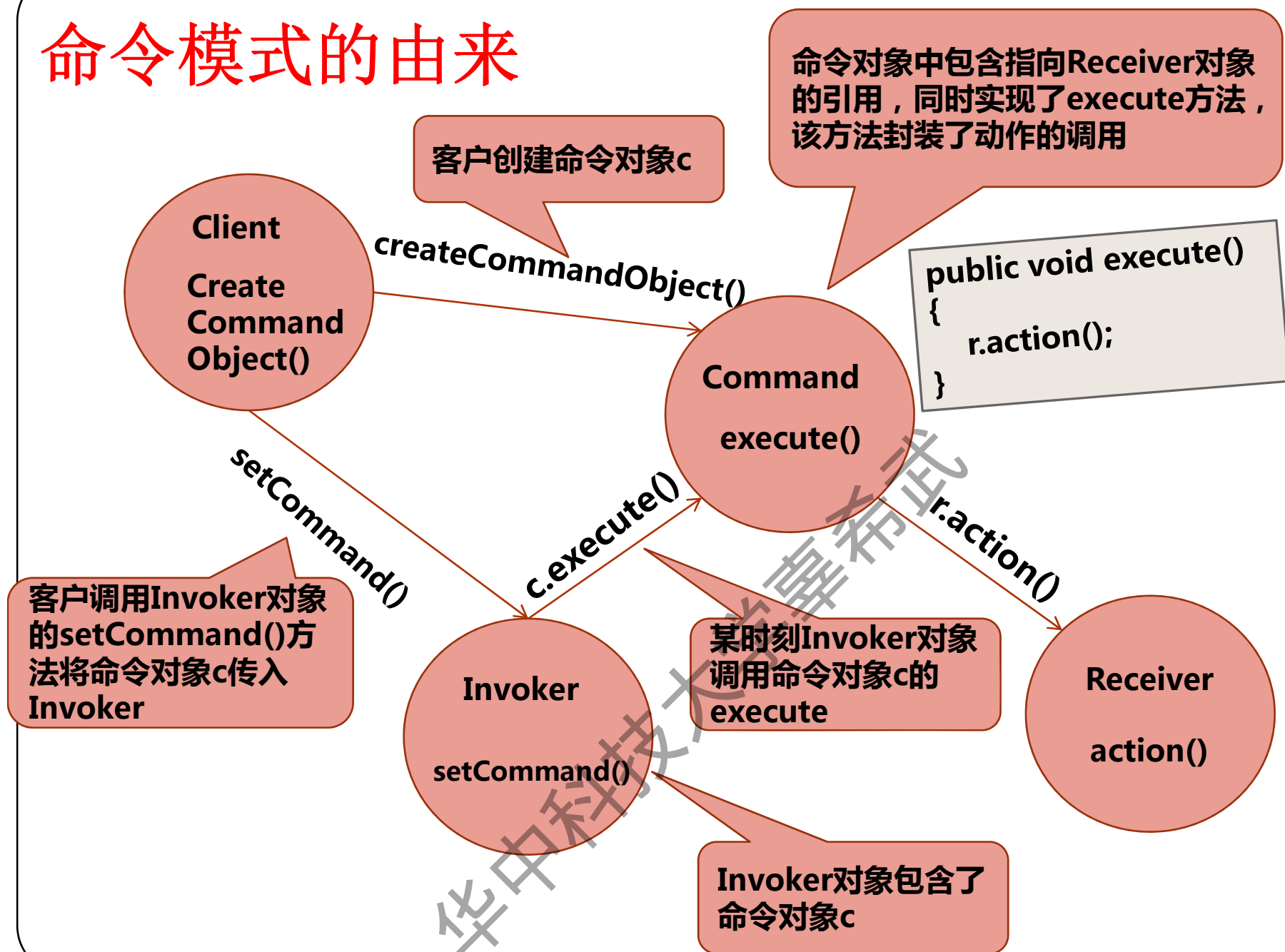


命令模式的由来

- 现在利用“命令对象”将Invoker和Receiver解耦



命令模式的由来



实现一个命令对象

□ 首先定义命令接口

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

□ 实现打开电灯的命令

```
public class LightOnCommand implements Command{
```

```
    Light light ;
```

命令对象包含了Receiver对象（即动作的执行者）的引用

```
    public LightOnCommand(Light light){  
        this.light = light;  
    }
```

构造函数被传入Receiver对象

```
    public void execute(){  
        light.on();  
    }  
}
```

调用Receiver对象（light）的on()方法

使用命令对象

□ 实现Invoker (遥控器)

```
public class SimpleRemoteControl{
```

```
    Command slot ;
```

一个命令插槽持有命令对象，而这个命令对象控制着一个电子设备

```
    public SimpleRemoteControl(){ }
```

```
    public void setCommand(Command command){
```

```
        this.slot = command;
```

```
    }
```

用来设置插槽持有的命令对象。如果需要改变遥控器按钮的行为，可以调用该方法设置新的命令对象

```
    public void buttonPressed(){
```

```
        slot.execute();
```

```
    }
```

```
}
```

当遥控器按钮按下时，调用命令对象的execute()方法

使用命令对象

□ 测试

```
public class SimpleRemoteControlTest{  
    public static void main(String[] args){
```

遥控器对象就是Invoker对象

```
        SimpleRemoteControl remote = new SimpleRemoteControl();
```

```
        Light light = new Light();
```

这是Receiver对象

```
        LightOnCommand lightOn = new LightOnCommand(light);
```

这是Command对象，构造命令对象时需要传入Receiver对象

```
        remote.setCommand(lightOn);  
        remote.buttonPressed();
```

调用Invoker对象的setCommand()方法把命令传给调用者

模拟按下按钮

```
    }  
}
```


定义命令模式

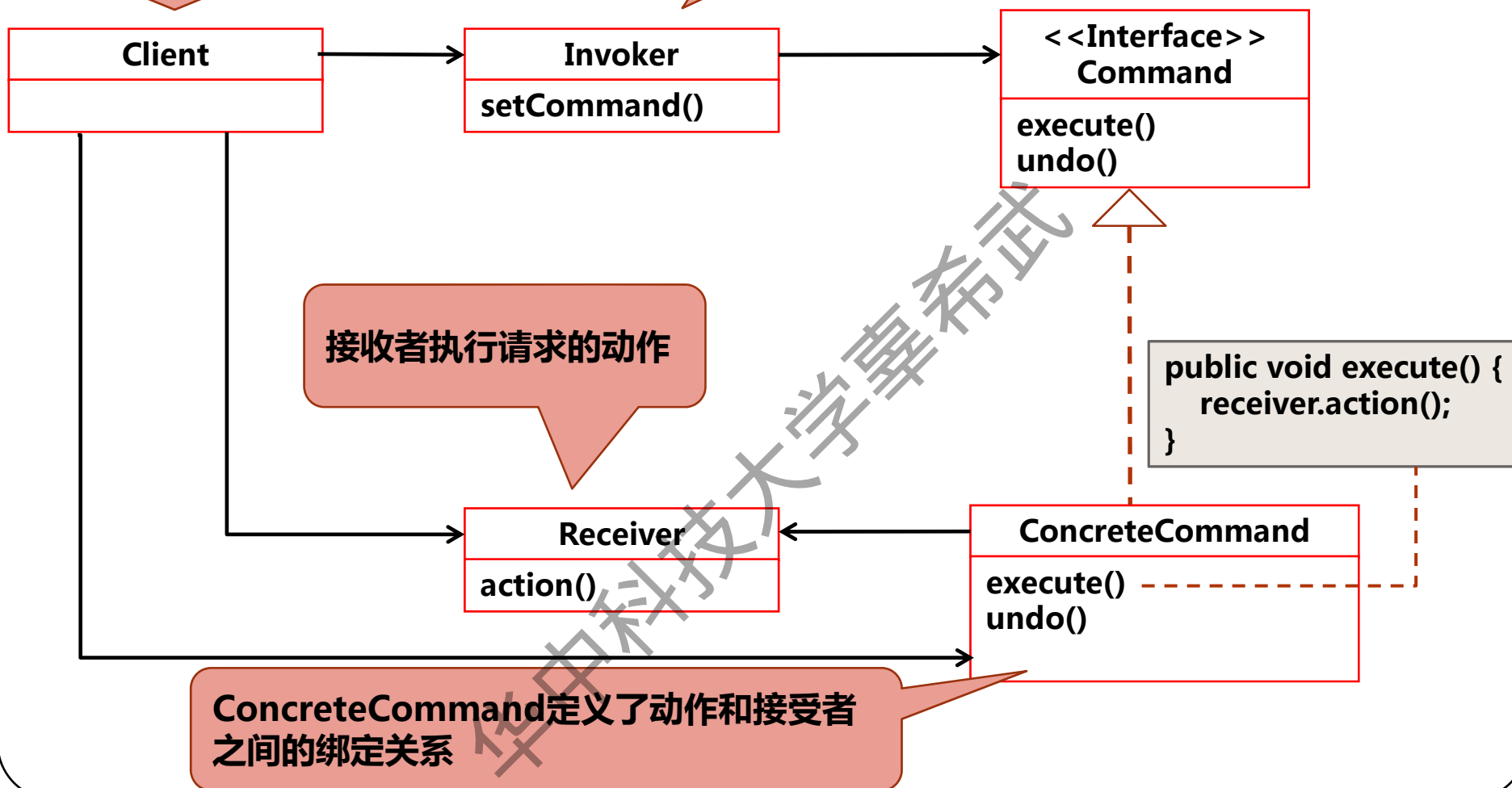
- ❑ 命令模式将“请求”封装成对象，以便参数化其它对象。命令模式也支持可撤销操作。
 - ❑ 命令对象将动作和动作的接受者包装在一起，对外只暴露出execute()方法。客户只知道如果调用execute()方法，请求的目的就可以达到。
 - ❑ 由于“请求”被封装成对象，因此可以用该对象作为参数来设置其它对象。如利用Invoker（遥控器）的setCommand()命令可以动态设置遥控器支持的命令插槽。
 - ❑ 也可以用“命令模式”来实现队列、日志和撤销。

命令模式的结构

客户负责创建ConcreteCommand对象，并对命令对象设置Receiver对象,同时对Invoker设置命令对象

Invoker持有命令对象，并在某时刻调用命令对象的execute()方法

Command为所有命令对象声明了接口，调用execute()方法可以让接受者执行动作，undo()撤销动作



将命令指定到遥控器插槽

```
public class RemoteControl {  
    Command[] onCommands;  
    Command[] offCommands;  
    public RemoteControl() {  
        onCommands = new Command[7];  
        offCommands = new Command[7];  
        Command noCommand = new NoCommand();  
        for (int i = 0; i < 7; i++) {  
            onCommands[i] = noCommand;  
            offCommands[i] = noCommand;  
        }  
    }  
    public void setCommand(int slot, Command onCommand, Command offCommand) {  
        onCommands[slot] = onCommand;  
        offCommands[slot] = offCommand;  
    }  
    public void onButtonWasPushed(int slot) {  
        onCommands[slot].execute();  
    }  
    public void offButtonWasPushed(int slot) {  
        offCommands[slot].execute();  
    }  
}
```

遥控器要处理7个“开”和7个“关”命令。用Command数组来保存

初始化命令数组，命令数组的每个元素保存的是“空命令”

设置指定插槽的On命令和Off命令

当按下开或关的按钮，就会调用相应的方法

遥控器就是Invoker

空命令

□ 空命令的作用是避免每次检查是否某个插槽加载了命令，如

```
public void onButtonWasPressed(int slot){  
    if(onCommand[slot] != null)  
        onCommand[slot].execute();  
}
```

□ 因此为了避免以上的做法，实现一个不做任何事情的命令

```
public class NoCommand implements Command{  
    public void execute() {}  
}
```

空对象在许多设计模式中被使用，
甚至空对象本身也被视为一种设计模式

实现更多的命令对象

□ 实现关闭电灯的命令

```
public class LightOffCommand implements Command{
```

```
    Light light ;
```

命令对象包含了Receiver对象（即动作的执行者）的引用

```
    public LightOnCommand(Light light){
```

```
        this.light = light;
```

```
    }
```

构造函数被传入Receiver对象

```
    public void execute(){
```

```
        light.off();
```

```
    }
```

```
}
```

调用Receiver对象（light）的on()方法

华中科技大学

实现更多的命令对象

□ 实现打开CD音响的命令

```
public class StereoOnWithCDCommand implements Command{
```

```
    Stereo stereo ;
```

命令对象包含了Receiver对象（即动作的执行者）的引用

```
    public StereoOnWithCDCommand (Stereo stereo){
```

```
        this.stereo= stereo;
```

```
    }
```

构造函数被传入Receiver对象

```
    public void execute(){
```

```
        stereo.on();
```

```
        stereo.setCD();
```

```
        stereo.setVolume(11);
```

```
    }
```

```
}
```

调用Receiver对象（stereo）的三个方法

根据其他的家电类，可以实现更多的命令对象。。。

测试遥控器

```
public class RemoteLoader{  
    public static void main(String[] args){  
        RemoteControl remoteControl = new RemoteControl ();  
  
        Light light= new Light();  
        Stereo stereo = new Stereo();  
  
        LightOnCommand lightOn = new LightOnCommand(light);  
        LightOffCommand lightOff = new LightOffCommand(light);  
        StereoOnWithCDCommand stereoOnCD= new StereoOnWithCDCommand (stereo);  
        StereoOffCommand stereoOffCD= new StereoOffCommand (stereo);  
  
        remoteControl.setCommand(0, lightOn,lightOff);  
        remoteControl.setCommand(1, stereoOnCD,stereoOff);  
  
        remoteControl.onButtonWasPushed(0);  
        remoteControl.offButtonWasPushed(0);  
        remoteControl.onButtonWasPushed(1);  
        remoteControl.offButtonWasPushed(1);  
    }  
}
```

创建好家电对象 (Receiver)

创建命令对象

把命令对象加载到命令插槽命令对象

按下遥控器按钮

实现命令撤销

- ❑ 重新定义Command接口，添加undo()方法。
- ❑ undo()执行的动作和execute()相反

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

华中科技大学 辜希武

实现命令撤销

□ LightOnCommand命令的撤销

```
public class LightOnCommand implements Command{  
    Light light ;  
  
    public LightOnCommand(Light light){  
        this.light = light;  
    }  
  
    public void execute(){  
        light.on();  
    }  
  
    public void undo(){  
        light.off();  
    }  
}
```

这里，undo()做的动作和execute()相反

实现命令撤销

□ LightOffCommand命令的撤销

```
public class LightOffCommand implements Command{  
    Light light ;  
  
    public LightOnCommand(Light light){  
        this.light = light;  
    }  
  
    public void execute(){  
        light.off();  
    }  
  
    public void undo(){  
        light.on();  
    }  
}
```

这里，undo()做的动作和execute()相反

```

public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand ;
    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand ;
    }
    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }
    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }
    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }
    public void undoButtonWasPushed() {
        undoCommand .undo();
    }
}

```

前一次被执行的命令保存在这里

刚开始，没有前一个命令，因此undoCommand被初始化为空命令

当按下按钮时，执行相应的命令。同时将这个命令保存在undoCommand中

当按下撤销按钮时，执行undoCommand.undo(),就倒转了前一个被执行的命令

思考：如何撤销多个命令？

实现基于状态的命令撤销

- ❑ 电灯开关操作的撤销太简单，因为只涉及对象的二值状态
- ❑ 有时撤销操作要基于对象更多的内部状态。例如天花板上的吊扇 (CeilingFan)，吊扇允许有多种转动速度，这时的撤销操作会更复杂

```
public class CeilingFan {  
    public static final int HIGH = 3;  
    public static final int MEDIUM = 2;  
    public static final int LOW = 1;  
    public static final int OFF = 0;  
    String location;  
    int speed;  
    public CeilingFan(String location) {  
        this.location = location;  
        speed = OFF;  
    }  
    public void high() { speed = HIGH; }  
    public void medium() { speed = MEDIUM; }  
    public void low() { speed = LOW; }  
    public void off() { speed = OFF; }  
    public int getSpeed() { return speed; }  
}
```

吊扇的内部4种状态

设置吊扇的速度

获取吊扇的速度

支持撤销的吊扇命令对象

□ 要把撤销加入到吊扇的诸多命令对象中，需要追踪吊扇的最后设置速度

```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;
    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }
    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }
    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) { ceilingFan.high(); }
        else if (prevSpeed == CeilingFan.MEDIUM) { ceilingFan.medium(); }
        else if (prevSpeed == CeilingFan.LOW) { ceilingFan.low(); }
        else if (prevSpeed == CeilingFan.OFF) { ceilingFan.off(); }
    }
}
```

记录吊扇以前的速度

在将吊扇速度设置为HIGH前，记录吊扇之前的速度

将吊扇的速度设置回之前的值，达到撤销的目的

支持宏命令

- ❑ 希望按下按钮，遥控器能执行一系列命令：灯光调暗，打开音响和电视，设置好DVD
- ❑ 制造一种新的命令对象，用来执行一堆命令

```
public class MacroCommand implements Command {  
    Command[] commands;  
    public MacroCommand(Command[] commands){  
        this.commands = commands;  
    }  
    public void execute(){  
        for(int i = 0; i < commands.length; i++){  
            commands[i].execute();  
        }  
    }  
    public void undo(){  
        for(int i = 0; i < commands.length; i++){  
            commands[i].undo();  
        }  
    }  
}
```

用命令数组来存储一大堆命令

当这个宏命令被执行时，依次执行数组里的每个命令

当这个宏命令被撤销时，依次撤销数组里的每个命令

使用宏命令

□ 首先创建要进入宏的命令集合

```
Light light = new Light();  
TV tv = new TV();  
Stereo stereo = new Stereo();
```

```
LightOnCommand lightOn = new LightOnCommand(light);  
StereoOnCommand stereoOn = new StereoOnCommand (stereo);  
TVOnCommand tvOn = new TVOnCommand(tv);
```

```
LightOffCommand lightOff = new LightOffCommand(light);  
StereoOffCommand stereoOff = new StereoOffCommand (stereo);  
TVOffCommand tvOff = new TVOffCommand(tv);
```

华中科技大学

使用宏命令

- ❑ 创建两个数组，一个用来记录开启命令，一个用来记录关闭命令

```
Command[] partyOn = { lightOn, stereoOn, tvOn };  
Command[] partyOff = { lightOff, stereoOff, tvOff };
```

- ❑ 创建宏命令

```
MacroCommand partyOnMacro = new MacroCommand(partyOn);  
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

- ❑ 将宏命令指定给希望的按钮

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

华中科技大学

命令模式的更多用途：队列请求和日志请求

- ❑ 命令可以把运算块打包（一个接受者和一组动作）。既然命令被封装成对象，因此可以像一般对象一样传来传去。
 - ❑ 命令对象被创建后，只要对象还在，运算依然可以被调用，甚至可以在不同线程中被调用。
 - ❑ 想象有一个命令队列，在一端添加命令，另一端则是工作线程。线程从队列首部取出一个命令，调用它的execute()方法；等待调用完成再取下一个命令。
-
- ❑ 在命令接口添加二个方法:store()和load()，命令模式可以支持命令对象保存在持久化介质中，或从持久化介质中恢复出命令对象
 - ❑ 这二个方法可以用来记录日志

命令模式的意图和适用性

□ 意图

- 将一个请求封装为一个对象，用不同请求（命令对象）对客户参数化
- 对请求排队或记录请求日志，以及支持可取消的操作

□ 适用性

- 抽象出待执行的动作以参数化某对象
- 在不同时刻指定、排列和执行请求
- 支持取消操作
- 支持修改日志
- 用构建在原语操作上的高层操作构造一个系统

命令模式的参与者

- Command
 - 声明执行操作的接口
- ConcreteCommand
 - 将一个接收者对象绑定于一个动作
 - 调用接收者相应的操作
- Client
 - 创建一个具体命令对象并设定其接收者
- Invoker
 - 要求命令执行请求
- Receiver
 - 知道如何实施与执行一个请求相关的操作

命令模式的效果分析

- ❑ 请求者不直接与接收者交互，即请求者不包接收者的引用，彻底消除了彼此之间的耦合
- ❑ 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
- ❑ 由于请求者的请求被封装到了具体命令中，就可以将具体命令保存到持久化的媒介中，在需要的时候，重新执行这个具体命令。因此，使用命令模式可以记录日志
- ❑ 使用命令模式可以对“请求”进行排队。每个请求都各自对应一个具体命令，因此可以按一定顺序执行这些具体命令

策略模式的由来

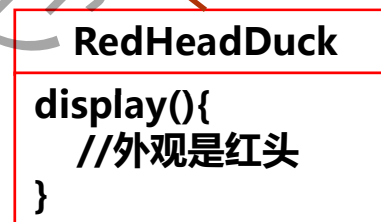
- 如果需要一个模拟鸭子游戏：可以呱呱叫，可以游泳
- 可以首先设计一个超类Duck，并让各种鸭子继承此超类

所有的鸭子会呱呱叫，会游泳，因此超类负责这两个方法的实现



每一种鸭子的外观不同，因此超类里display()方法是抽象的

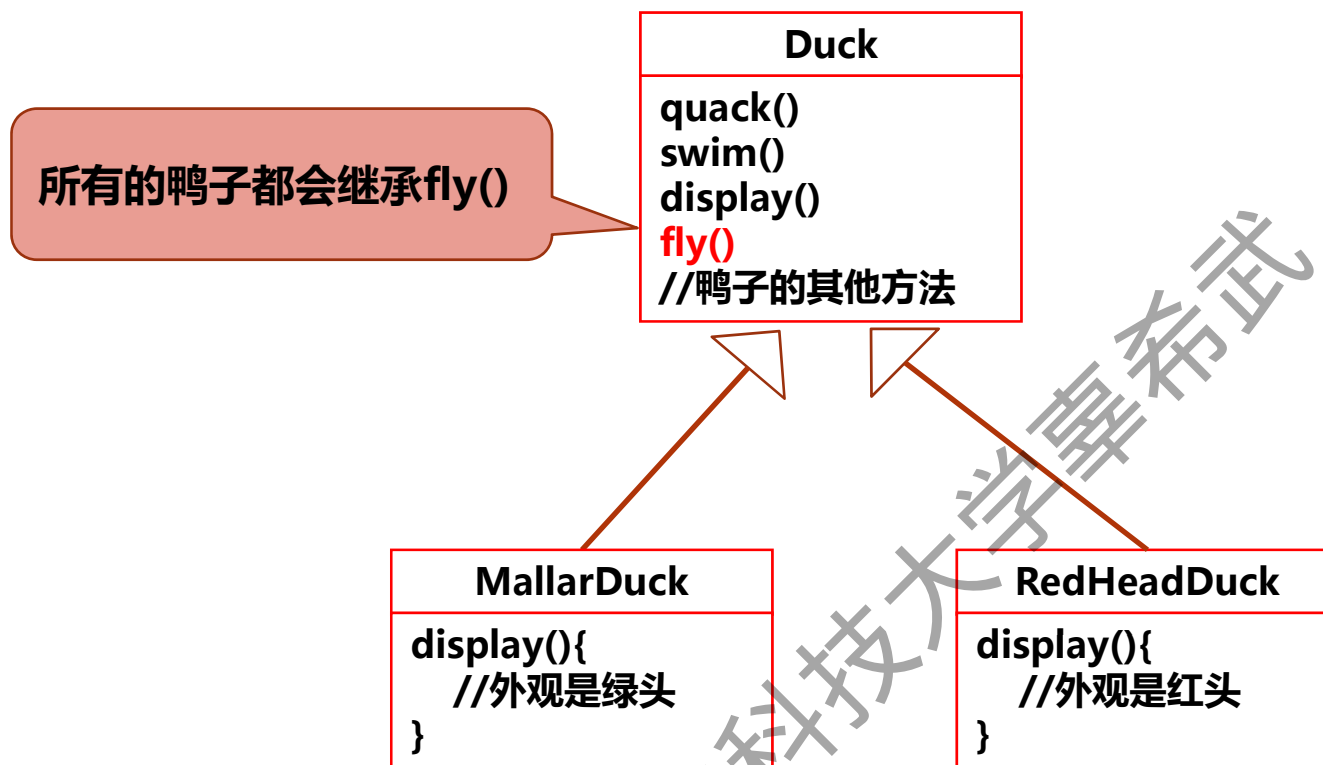
每一种具体的鸭子负责自己display()行为的实现



华中科技大学

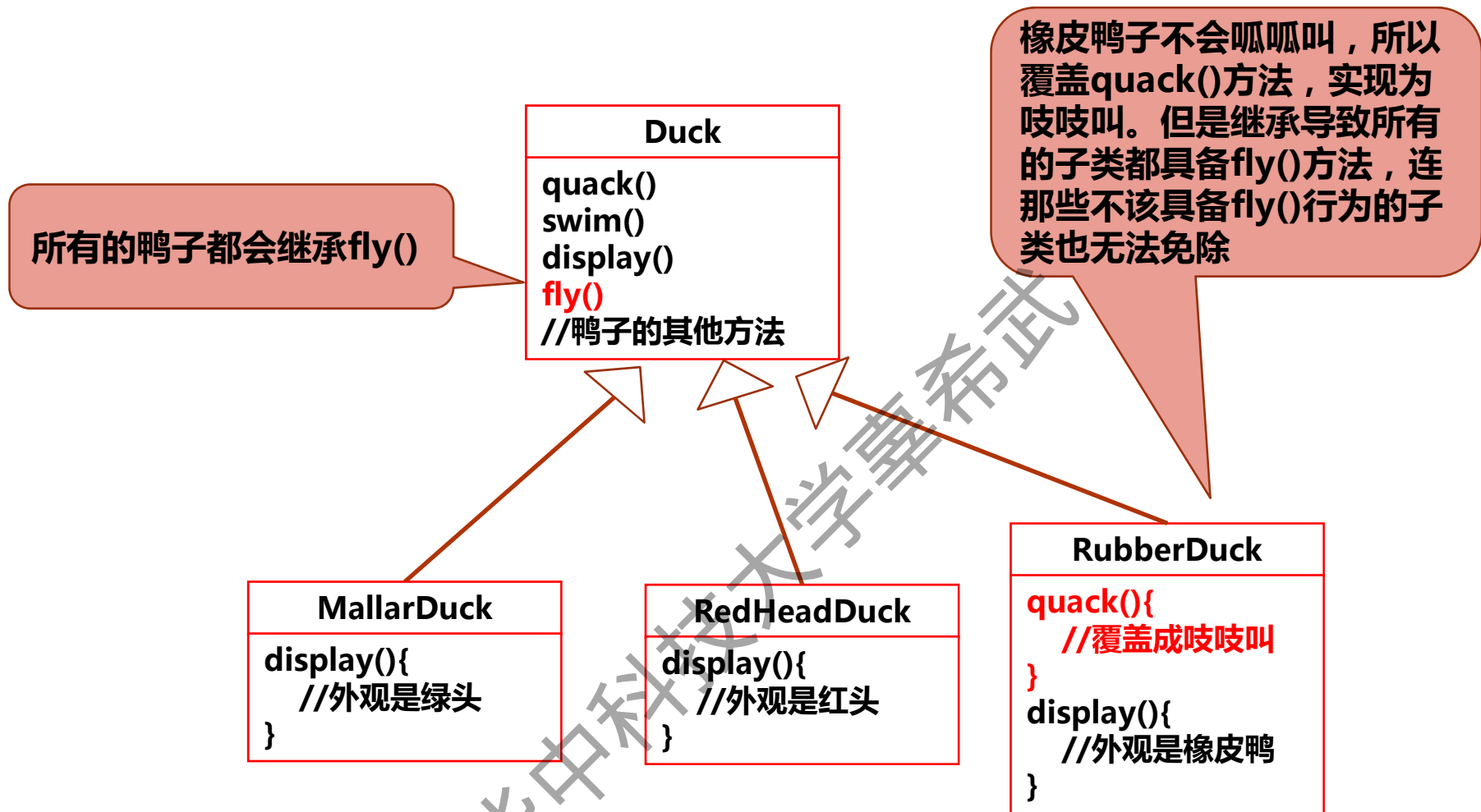
策略模式的由来

- 现在我们想让鸭子能飞
- 在超类里加上fly()方法，利用继承，所有的鸭子能飞了



策略模式的由来

□ 可怕的问题发生了：子类橡皮鸭子也能飞了



策略模式的由来

- 为了解决这个问题，可以把橡皮鸭中的fly()方法覆盖掉

RubberDuck

```
quack(){ //覆盖成吱吱叫 }  
display(){ //外观是橡皮鸭 }  
fly(){ //什么都不做 }
```

- 但是，如果派生出诱饵鸭DecoyDuck，又会如何？诱饵鸭既不会飞，也不会叫

DecoyDuck

```
quack(){ //覆盖成什么都不做 }  
display(){ //外观是诱饵鸭 }  
fly(){ //什么都不做 }
```

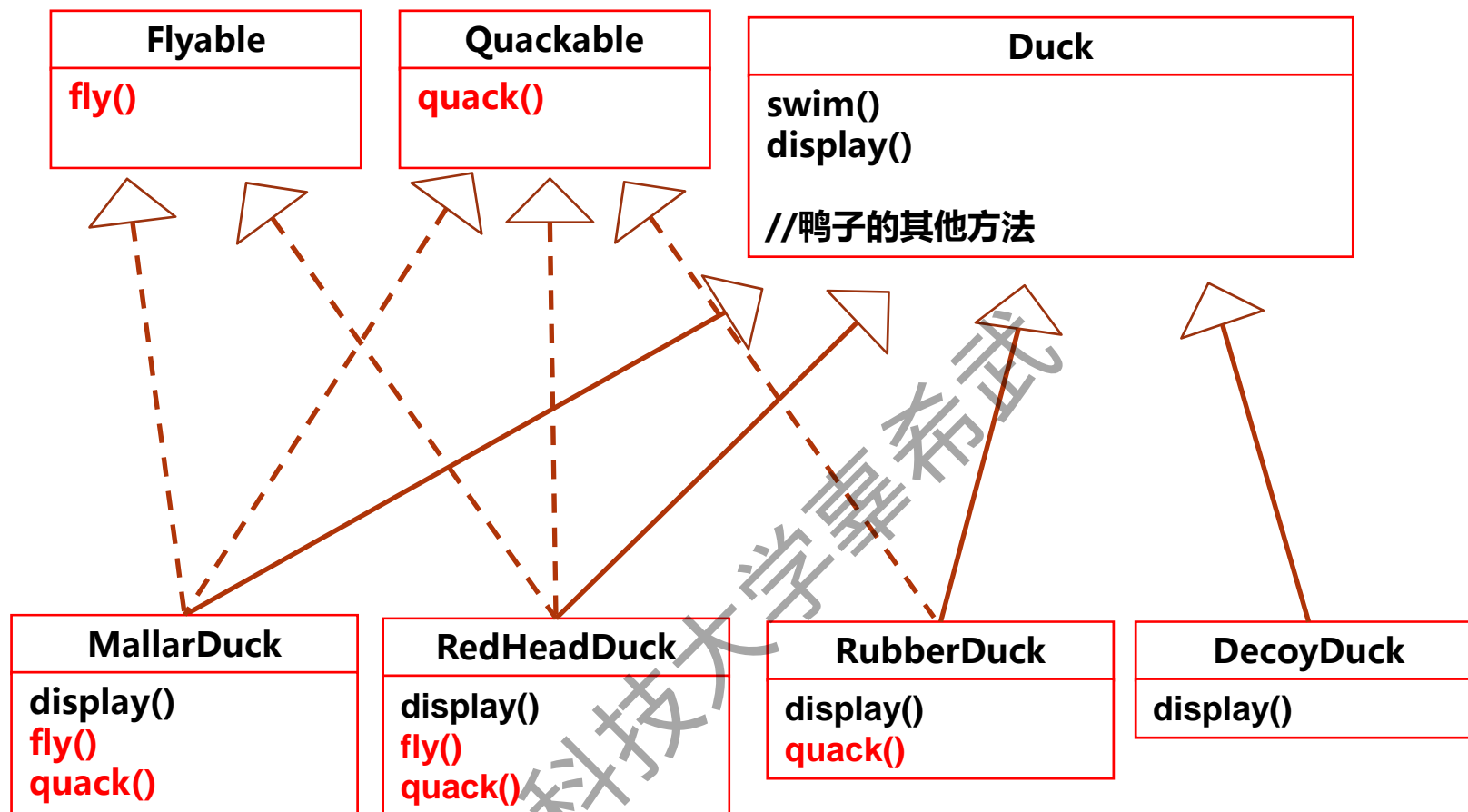
每当有新的鸭子出现，就要被迫检查并可能需要覆盖quack()和fly()，无穷无尽的噩梦

可以看到利用继承来提供鸭子的行为导致如下缺点：

- 1：代码在多个子类中重复
- 2：运行时的行为不容易改变
- 3：很难知道鸭子的全部行为
- 4：改变会牵一发而动全身，造成其它鸭子不想要的改变

策略模式的由来

□ 利用接口如何



利用接口不会再有会飞的橡皮鸭，但代码无法复用，因为每个实现接口的鸭子必须实现`quack()`和`fly()`方法

策略模式的由来

□ 利用继承和接口存在的问题

- 继承使得子类继承了不该继承的方法
- 接口不具有实现代码，因此实现接口无法达到代码的复用

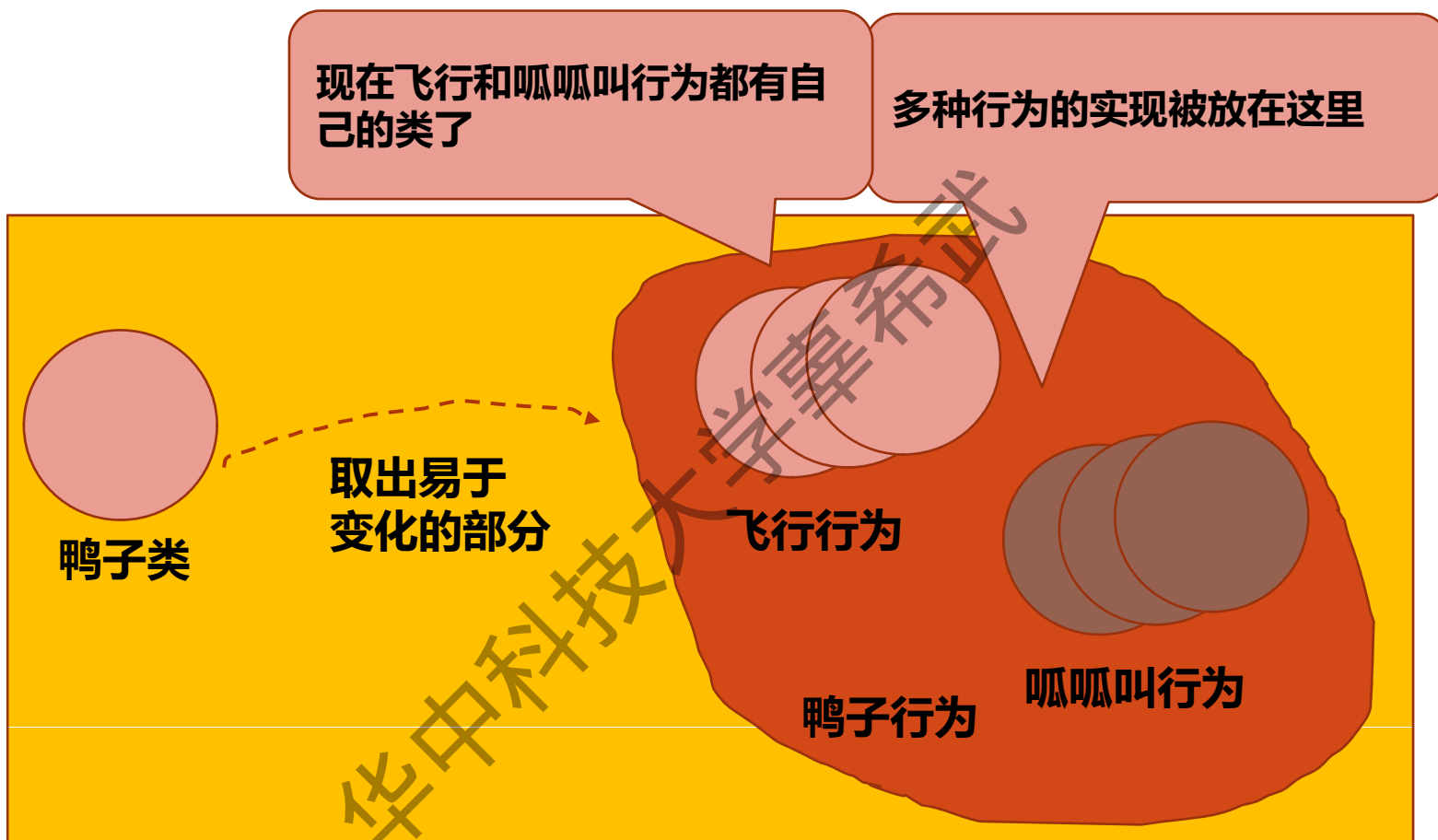
□ 幸好有个设计原则适合此状况

- 找出应用中可能需要变换之处，把它独立出来，不要和那些不需要变换的代码混在一起
- 该把鸭子的行为从Duck类中抽取出来了

华中科技大学

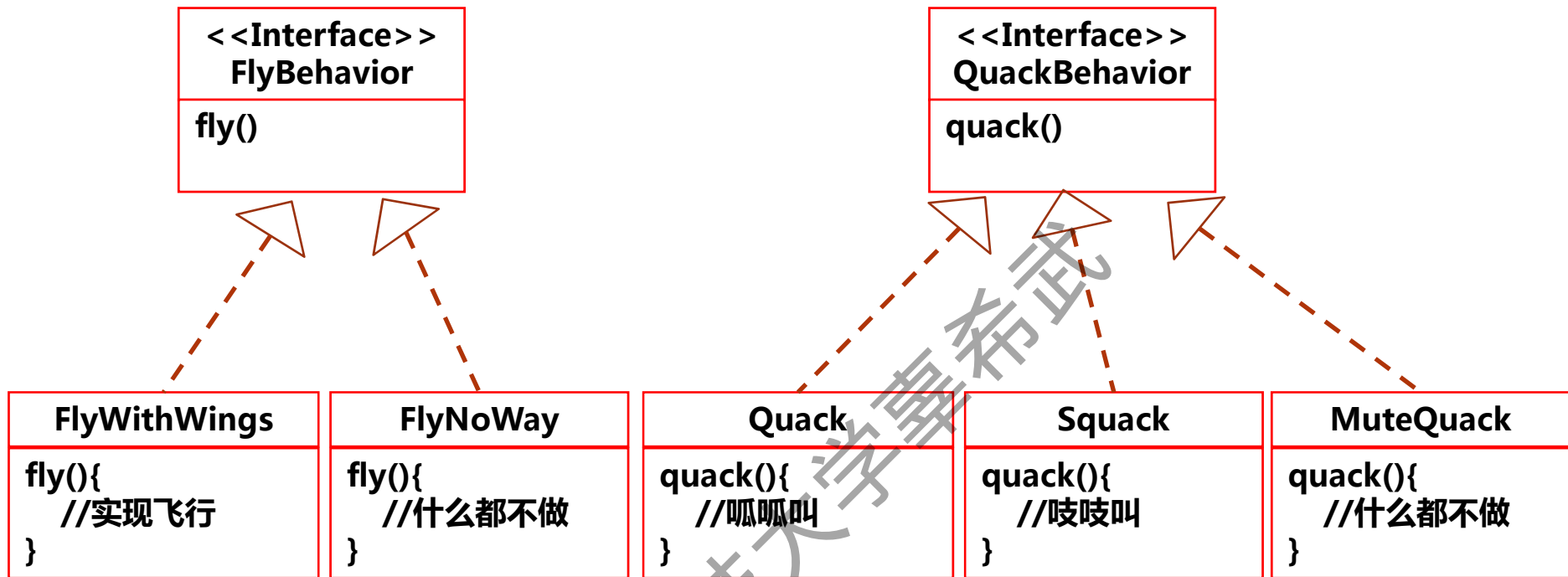
策略模式的由来

- 分开变换和不会变化的部分
 - Duck类的fly()和quack()会随着鸭子的不同而改变
 - 为了把这二个行为从Duck类中分开，我们把它们从Duck类中取出来，建立一组新类来代表每个行为



策略模式的由来

□ 设计鸭子的行为：基于接口编程，不是基于实现编程



这样的设计，可以让飞行和呱呱叫的动作被其它对象复用，因为这些行为已经与鸭子无关了。而我们新增一些行为，不会影响到既有的行为类，也不会影响到“使用”到已有行为的鸭子类

策略模式的由来

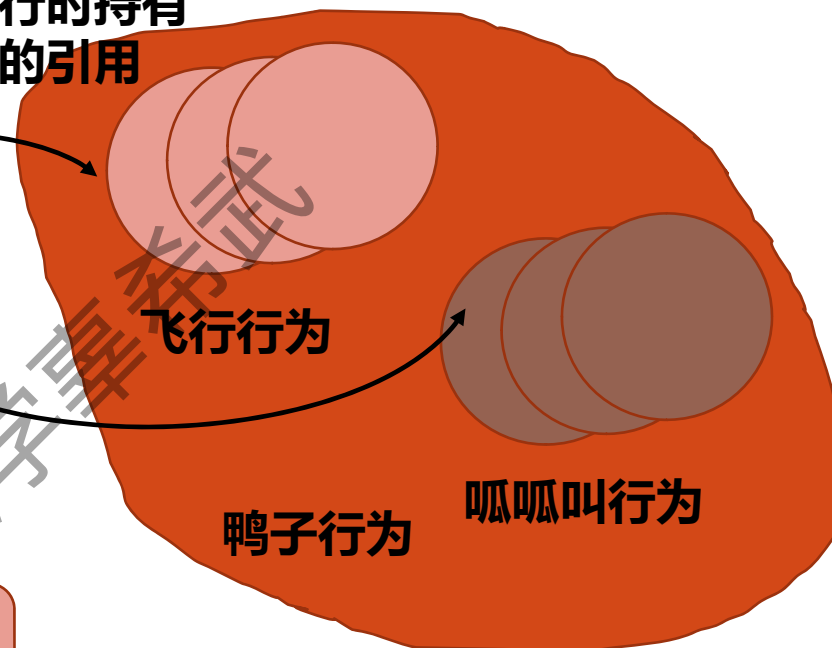
□ 整合鸭子的行为

Duck类中加入二个实例变量，
类型为接口类型



这二个方法取代fly()和quack()

实例变量在运行时持有
指定行为对象的引用



策略模式的由来

□ FlyBehavior接口以及具体的飞行行为类

```
public interface FlyBehavior {  
    public void fly();  
}
```

```
public class FlyWithWings implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying!!");  
    }  
}
```

实现二个具体的飞行行为，你还可以实现更多的飞行行为

```
public class FlyNoWay implements FlyBehavior {  
    public void fly() {  
        System.out.println("I can't fly");  
    }  
}
```

策略模式的由来

□ QuackBehavior接口以及具体的飞行行为类

```
public interface QuackBehavior {  
    public void quack();  
}  
  
public class Quack implements QuackBehavior {  
    public void quack() {  
        System.out.println("Quack");  
    }  
}  
  
public class Squack implements QuackBehavior {  
    public void quack() {  
        System.out.println("Squeak");  
    }  
}  
  
public class MuteQuack implements QuackBehavior {  
    public void quack() {  
        System.out.println("<< Silence >>");  
    }  
}
```

实现三个具体的呱呱叫行为，你还可以实现更多的呱呱叫行为

策略模式的由来

□ Duck类

```
public abstract class Duck {  
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
    public Duck() { }  
    public void setFlyBehavior (FlyBehavior fb) {  
        flyBehavior = fb;  
    }  
    public void setQuackBehavior( QuackBehavior qb)  
        quackBehavior = qb;  
    }  
    abstract void display();  
    public void performFly() {  
        flyBehavior.fly();  
    }  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
    public void swim() {  
        System.out.println("All ducks float, even decoys!");  
    }  
}
```

为行为接口类型声明二个引用变量，指向具体的行为实例。所有的鸭子子类都会继承它们

这二个方法可以动态地设定鸭子的飞行行为和呱呱叫行为

具体的显示行为由子类处理，因此定义为抽象方法

鸭子对象不亲自处理飞行行为，而是委托给flyBehavior引用的对象

鸭子对象不亲自处理呱呱叫行为，而是委托给quackBehavior引用的对象

策略模式的由来

□ 测试

```
public class ModelDuck extends Duck {  
    public ModelDuck() {  
        flyBehavior = new FlyNoWay();  
        quackBehavior = new Quack();  
    }  
    public void display() {  
        System.out.println("I'm a model duck");  
    }  
}
```

模型鸭一开始不会飞，但会叫

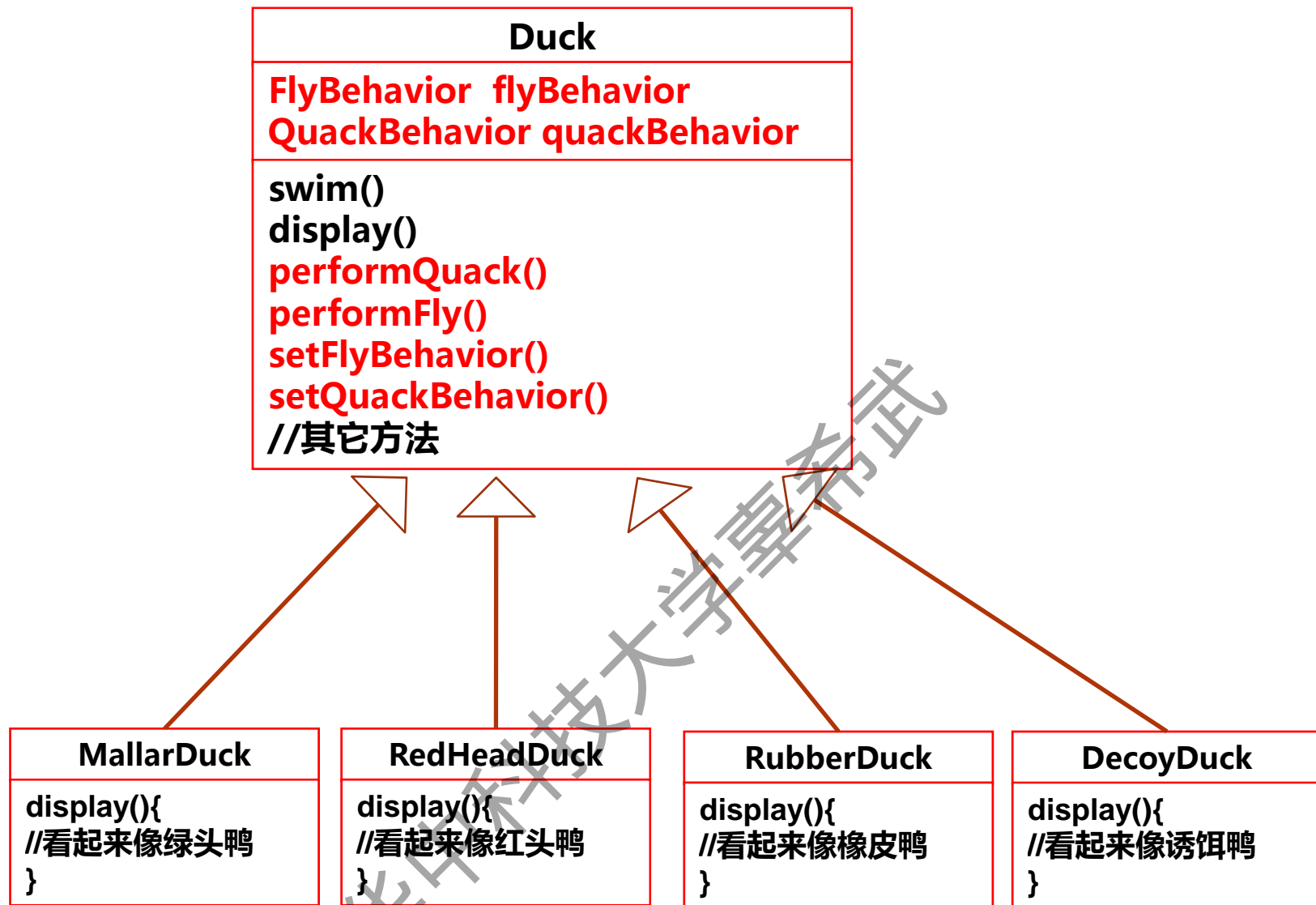
```
public class FlyRocketPowered implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying with a rocket");  
    }  
}
```

实现利用火箭动力的飞行行为

```
public class MiniDuckSimulator1 {  
    public static void main(String[] args) {  
        Duck model = new ModelDuck();  
        model.performFly();  
        model.setFlyBehavior(new FlyRocketPowered());  
        model.performFly();  
    }  
}
```

动态地改变鸭子的飞行行为，同样地，可以改变鸭子的呱呱叫行为

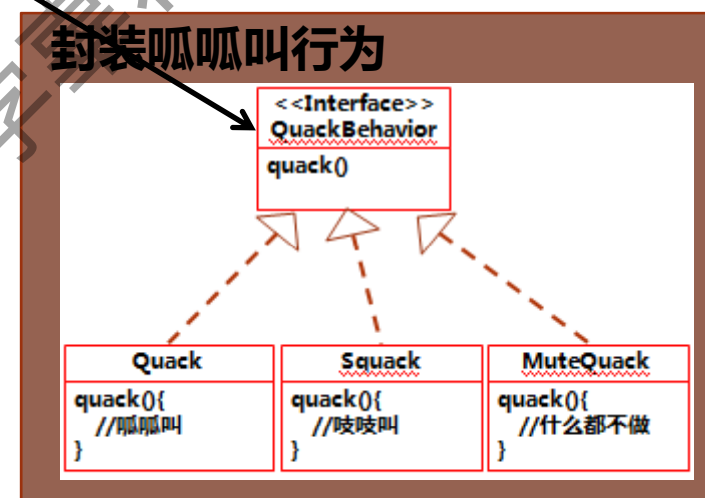
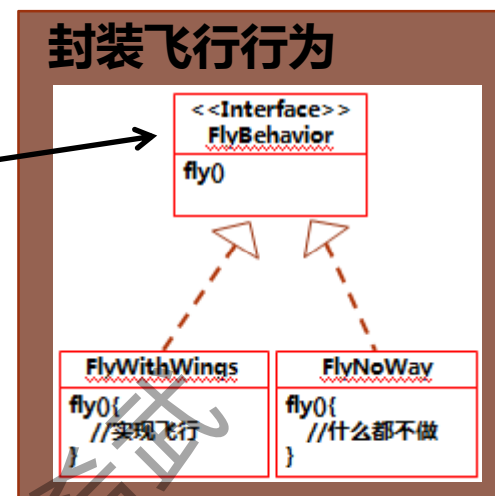
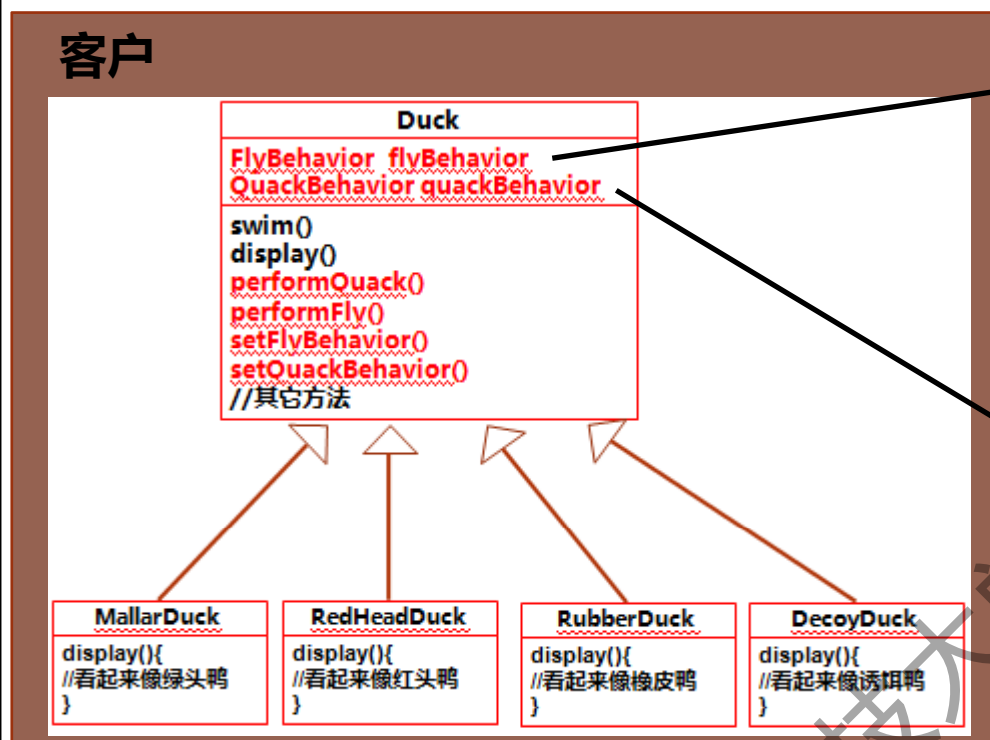
策略模式的由来



策略模式的整体结构

客户使用封装好的“算法”族

现在把每组行为想象成“算法”



客户和算法之间是“HAS-A”关系。”多用组合，少用继承”

这些“算法”是可以互换的

策略模式的定义

□ **策略模式** 定义了算法族，分别封装起来，让它们之间可以互相替换。此模式让算法的变换独立于使用算法的客户

华中科技大学 辜希武

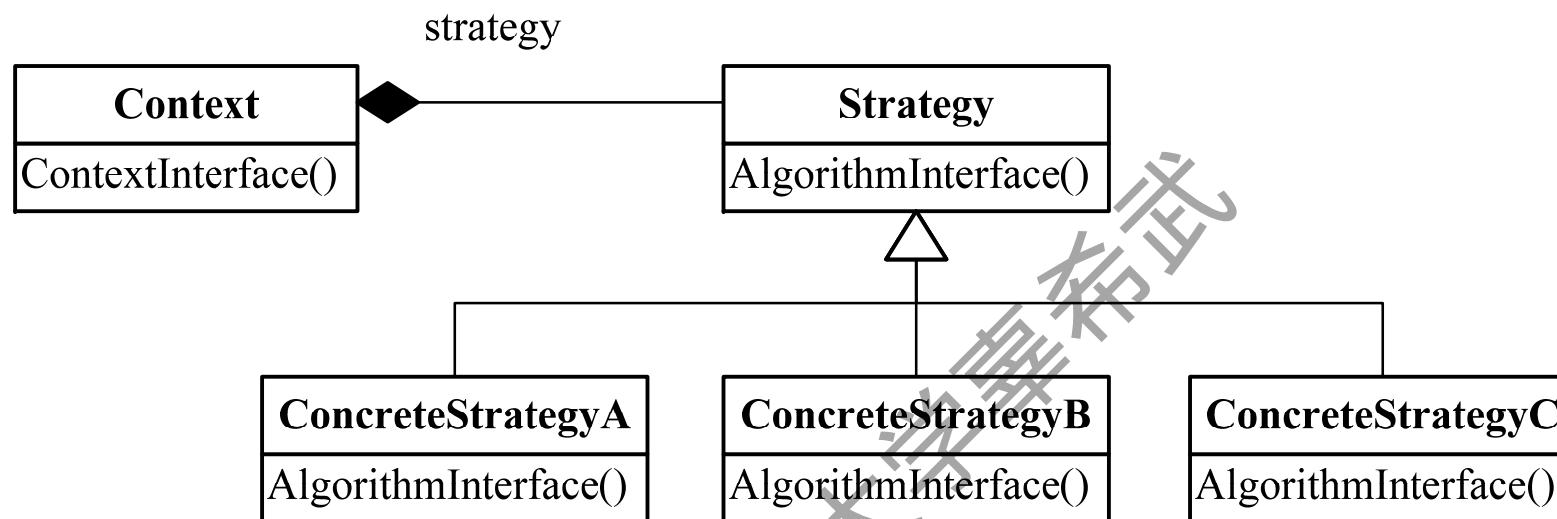
策略模式的由来

- ❑ 多种算法实现同一功能，客户希望在运行时根据上下文选择其中一个算法
- ❑ 传统策略算法的缺点
 - ❑ 使用算法的类复杂而难于维护，尤其当需要支持多种算法且每种算法都很复杂时问题会更加严重
 - ❑ 不同的时候需要不同的算法，支持并不使用的算法可能带来性能的负担
 - ❑ 算法的实现和使用算法的对象紧紧耦合在一起，使新增算法或修改算法变得十分困难，系统应对变化的能力很差
- ❑ 将每种算法的实现都剥离出来构成一个个独立算法对象，再从这些对象中抽象出公共算法接口，最后将算法接口组合到使用算法类中

策略模式的意图和适用性

- 意图：定义一系列算法，一个个进行封装，并使其可相互替换。策略模式使得算法可独立于使用它的客户而变化
- 适用场合
 - 在软件构建过程中，某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码到对象中，将会使得对象变得异常复杂；而且有时候支持不同的算法也是一个性能负担
 - 如何在运行时根据需要透明地更改对象的算法？如何将算法与对象本身解耦，从而避免上述问题？策略模式很好地解决了这两个问题

策略模式的结构



策略模式的参与者

□ Strategy

- 定义所有支持算法的公共接口
- Context使用该接口调用某ConcreteStrategy定义的算法

□ ConcreteStrategy

- 以Strategy接口实现某具体算法

□ Context

- 用一个ConcreteStrategy对象来配置
- 维护一个对Strategy对象的引用
- 可定义一个接口来让Strategy访问它的数据

策略模式的效果分析

- ❑ 算法和使用算法的对象相互分离，客户程序可以在运行时动态选择算法，代码复用性好，便于修改和维护
- ❑ 用组合替代继承，效果更好。若从Context直接生成子类，也可以实现对象的多种算法，但继承使子类 and 父类紧密耦合，使Context类难以理解、难以维护和难以扩展。策略模式采用组合方式，使Context和Strategy之间的依赖很小，更利于代码复用
- ❑ 消除了冗长的条件语句序列，将不同的算法硬编码进一个类中，选择不同的算法必然要使用冗长的条件语句序列，采用策略模式将算法封装在一个个独立的Strategy类中消除了这些条件语句

复合模式

华中科技大学 曹希武

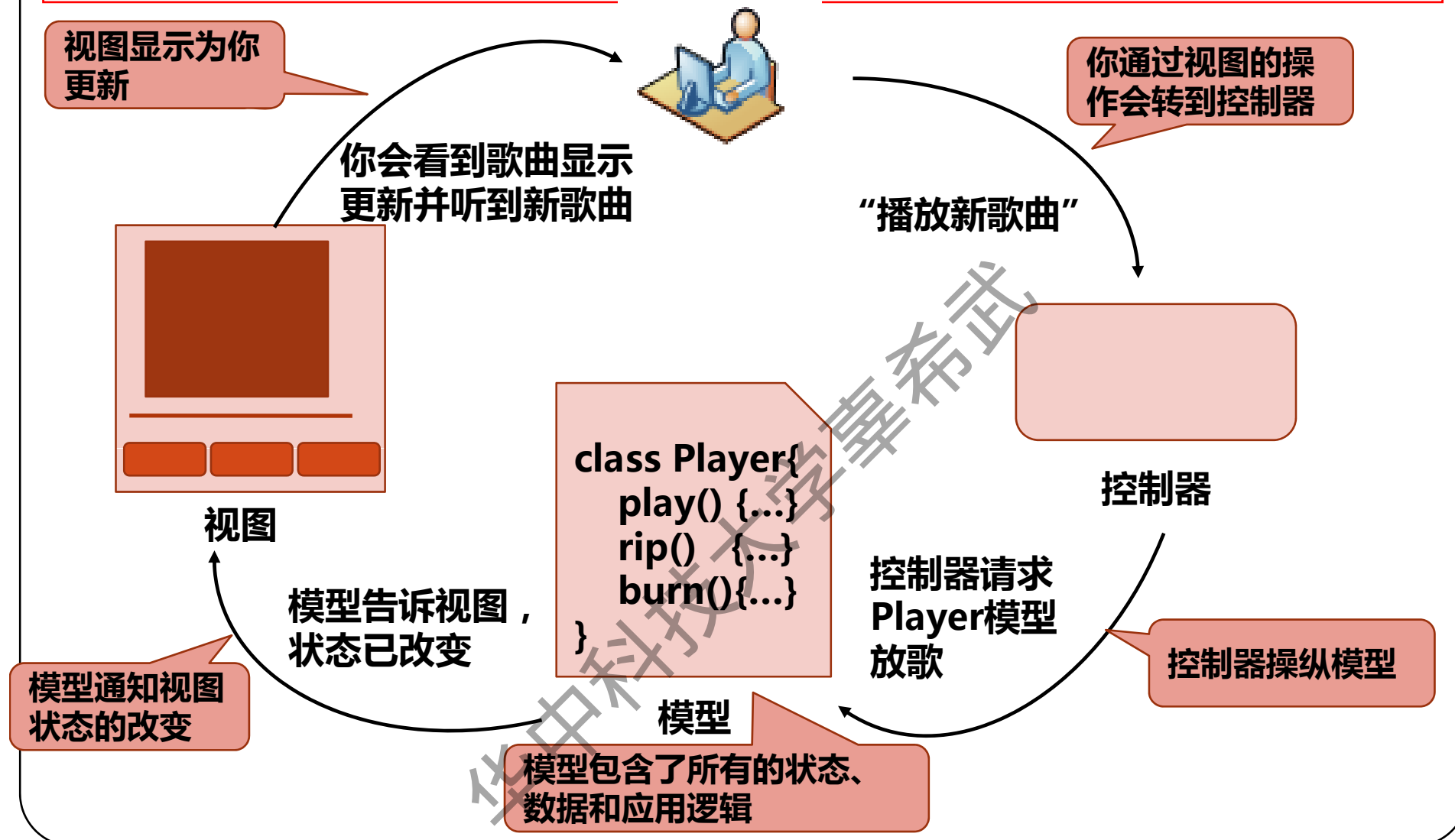
复合模式

- ◆ 模式经常被一起使用，并被组合在同一个设计解决方案中
- ◆ 复合模式在一个解决方案中结合两个或多个模式，以解决一般或重复发生的问题
- ◆ MVC就是经典的复合模式
 - ◆ Model-View-Controller

华中科技大学

认识模型-视图-控制器

□ 一个MP3播放器, 低层就是模型-视图-控制器

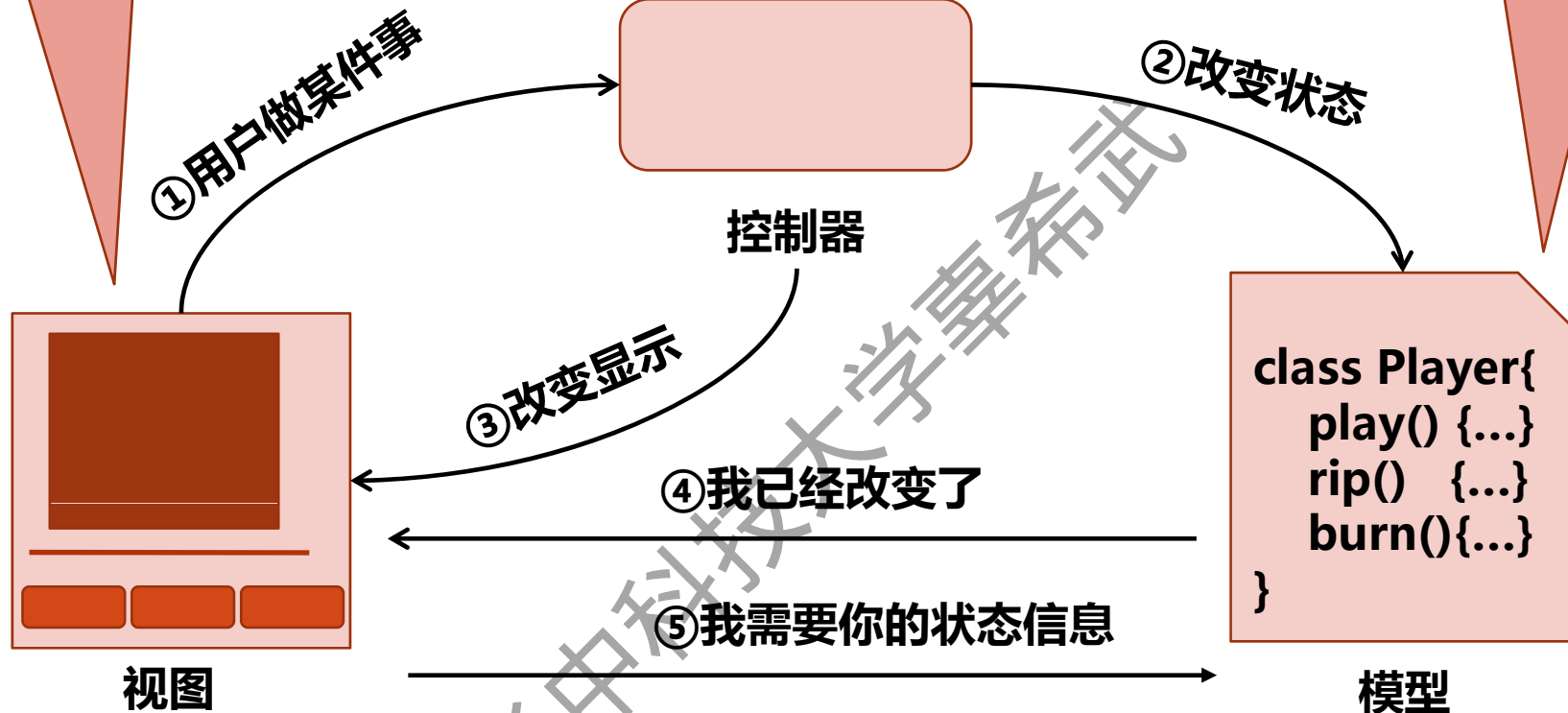


模型-视图-控制器三者关系

用来呈现模型，视图通常直接从模型中取得它需要的显示状态与数据

取得用户的输入并解读其对模型的意思

模型持有所有的数据、状态和程序逻辑。模型没有注意到视图和控制器，虽然它提供了操纵和检索状态的接口，并发送状态改变通知给观察者



模型-视图-控制器三者关系

①你是用户-----你和视图交互

视图是模型的窗口，当你对视图做一些事情（如按下“播放”按钮），视图就告诉控制器你做了什么。控制器会负责处理。

②控制器要求模型改变状态

控制器解读你的动作。如果你按下某个按钮，控制器会理解这个动作的含义，并告知模型如何做出对应的动作。

③控制器也可能要求视图做出改变

当控制器从视图收到某一动作，结果可能是它也需要告诉视图改变其显示结果。比如说把界面某些按钮或菜单项变成有效或无效

④当模型状态改变时，模型会通知视图

只要模型内部的东西改变时，模型都会通知视图它的状态改变了。

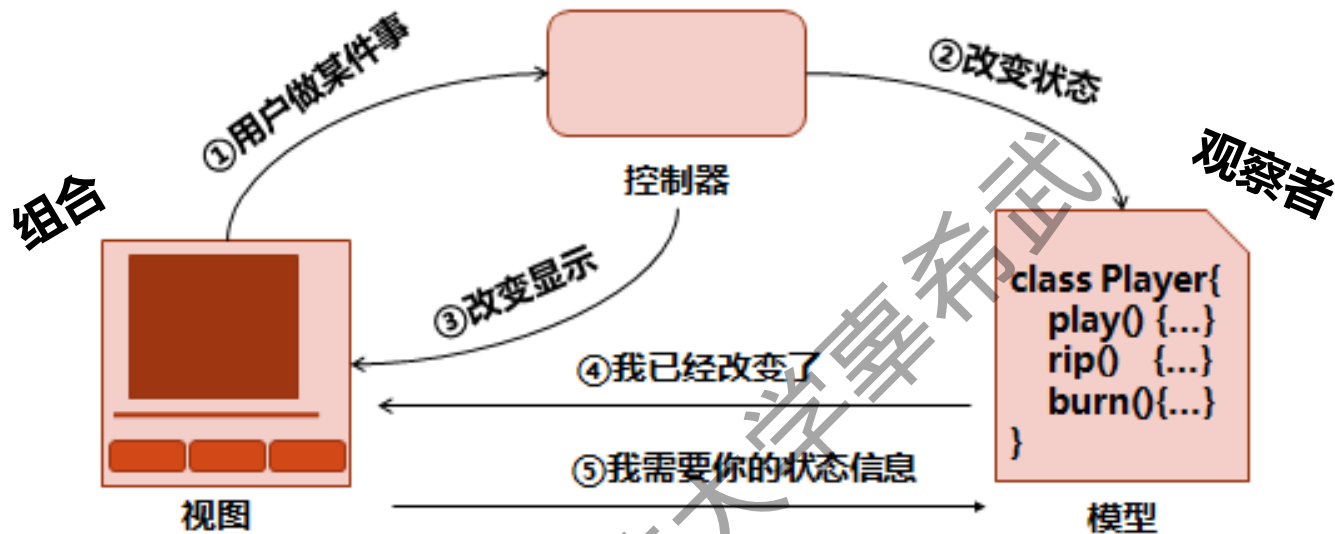
⑤视图向模型询问状态

视图直接从模型取得它显示的状态。比方说，当模型通知新歌开始播放，视图向模型询问歌名并显示出来。当控制器请求视图改变时（③），视图也可能向模型询问某些状态

模型-视图-控制器所用到的模式

策略

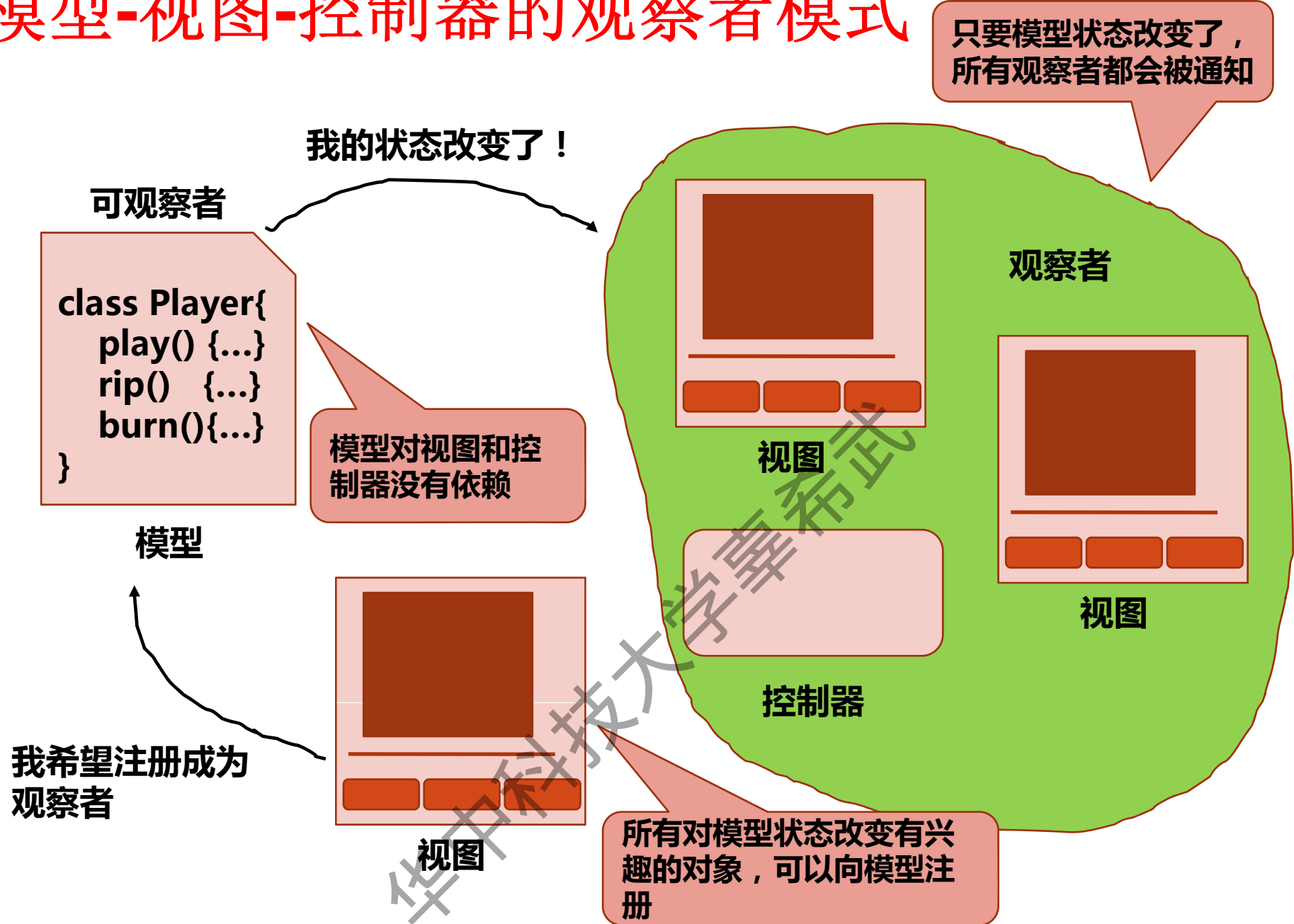
视图和控制器都实现了策略模式：视图是一个对象，可以被调整使用不同的策略，而控制器提供了策略。视图只关心系统中可视化部分，对于任何界面行为，都委托给控制器处理。使用策略模式可以让视图和模型之间的关系解耦，因为控制器负责和模型交互来传递用户的请求。对于工作怎么完成的，视图毫不知情。



视图还实现了组合模式。显示包括了窗口、面板、按钮等。每个显示组件不是组合节点（如窗口），就是叶节点（如按钮）。当控制器告诉视图更新时，只需告诉视图的最顶层组件即可，组合会处理其余的事。

模型实现了观察者模式，当状态改变时，相关对象（如视图）将持续更新。使用观察者模式，可以让模型完全独立于视图和控制器。同一个模型可以使用不同的视图，甚至可以同时使用多个视图

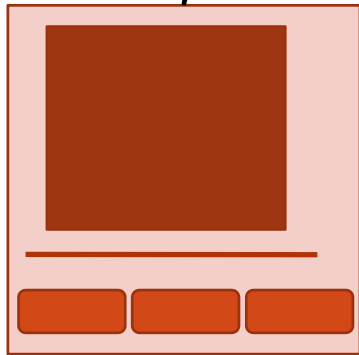
模型-视图-控制器的观察者模式



模型-视图-控制器的策略模式

视图委托控制器来处理
用户动作

用户做了某件事



视图

控制器

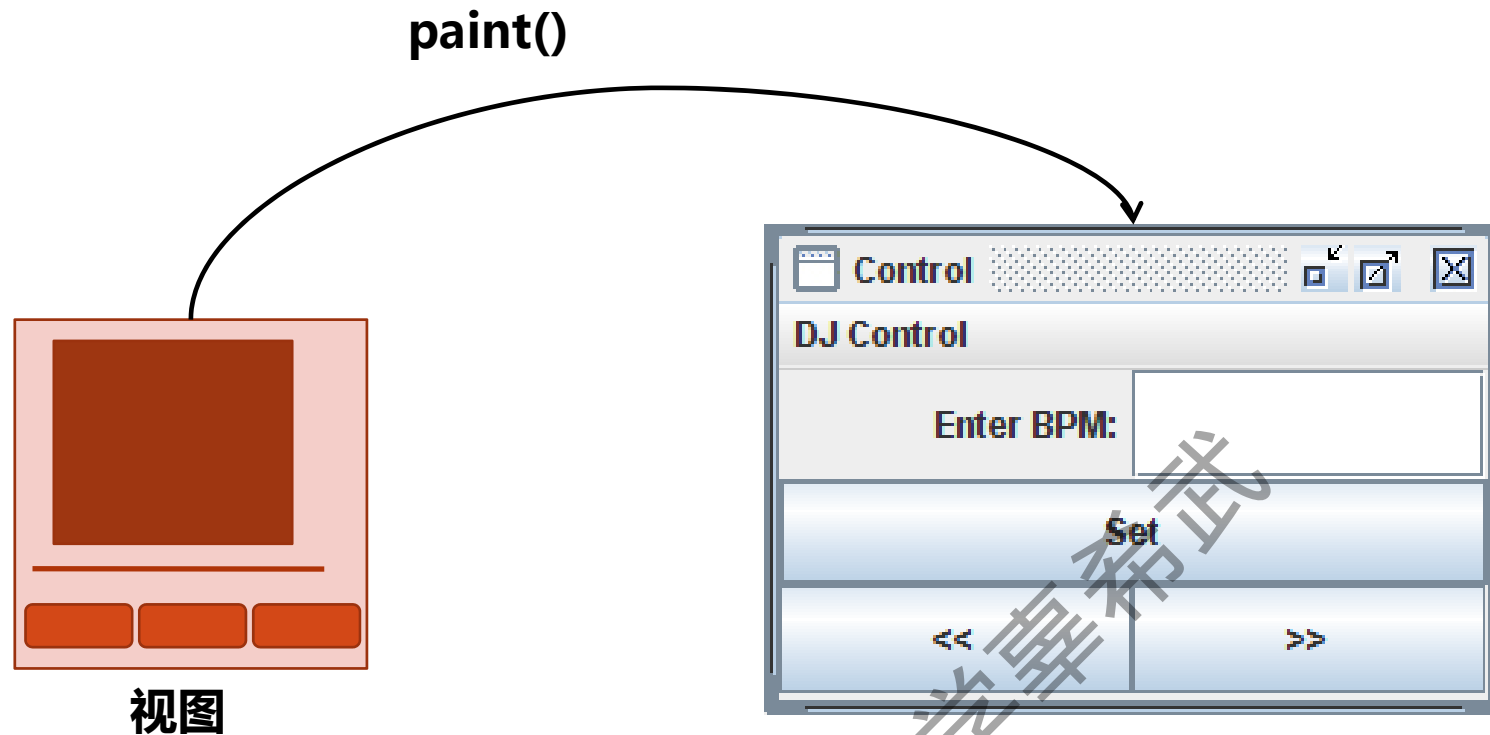
对视图来说，控制器是
策略，也就是知道如何
处理用户动作的对象

控制器

想换另一种行为，换掉
控制器就可以了

视图只关注呈现，控制器关注把用户输入
转为模型上的行为

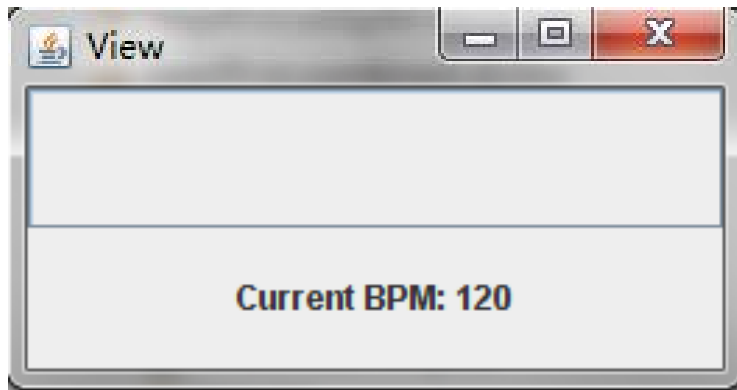
模型-视图-控制器的组合模式



视图是GUI组件（标签，窗口、文本输入框、按钮）的组合。顶层组件包含其他的组件，直到叶子节点

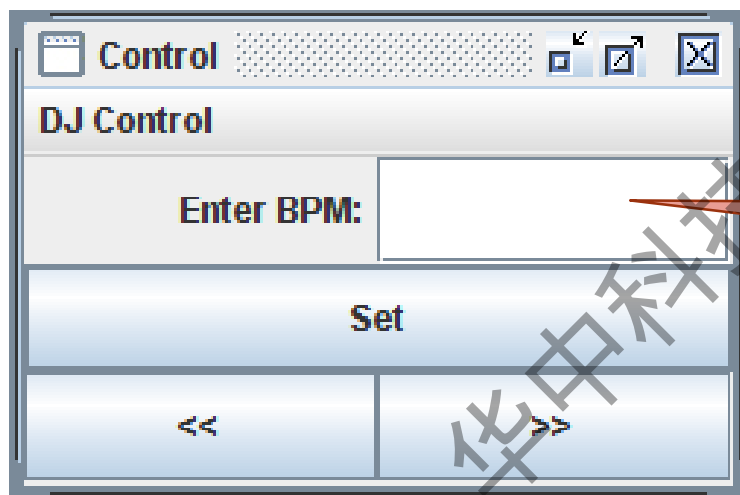
利用MVC控制音乐节拍

- ❑ 音乐节拍：95BPM (beat per minute)
- ❑ 首先设计节拍控制器的视图



脉动柱显示实时节拍

显示当前BPM

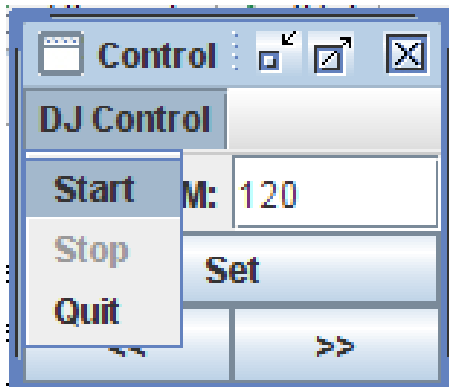


输入BPM值，点击“Set”设置节拍

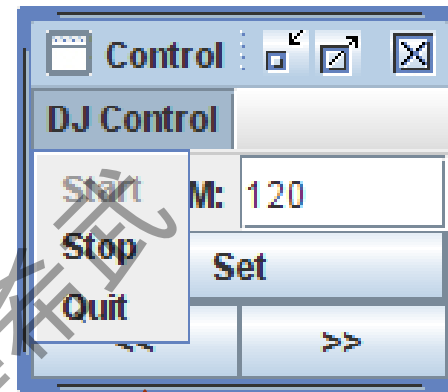
每分钟减少或增加1拍

利用MVC控制音乐节拍

- ❑ 音乐节拍：95BPM (beat per minute)
- ❑ 首先设计节拍控制器的视图



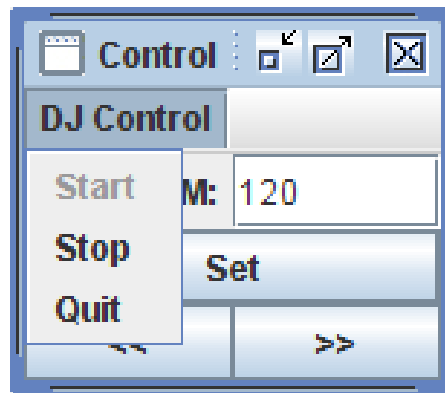
选择“Start”菜单项开始产生节拍
注意直到点击“Start”，“Stop”都是无效的”



选择“Stop”菜单项停止产生节拍
注意节拍产生后，“Start”是无效的

利用MVC控制音乐节拍

□ 控制器位于视图和模型之间



所有用户的动作
被送到控制器

控制器取得输入，了解
怎么回事，然后再对模
型做出请求

控制器

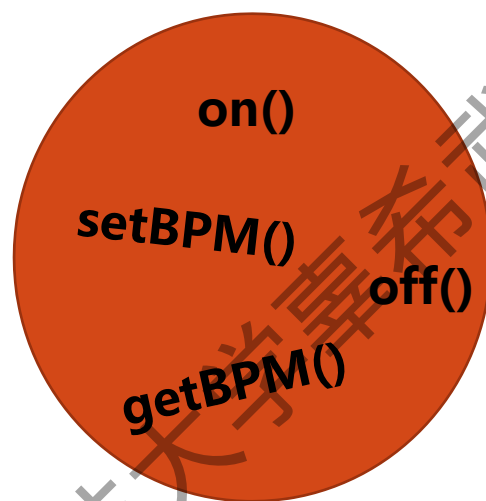
利用MVC控制音乐节拍

□ 别忘了模型

BeatModel是系统的核心，实现了节拍的开始、停止的逻辑，管理BPM

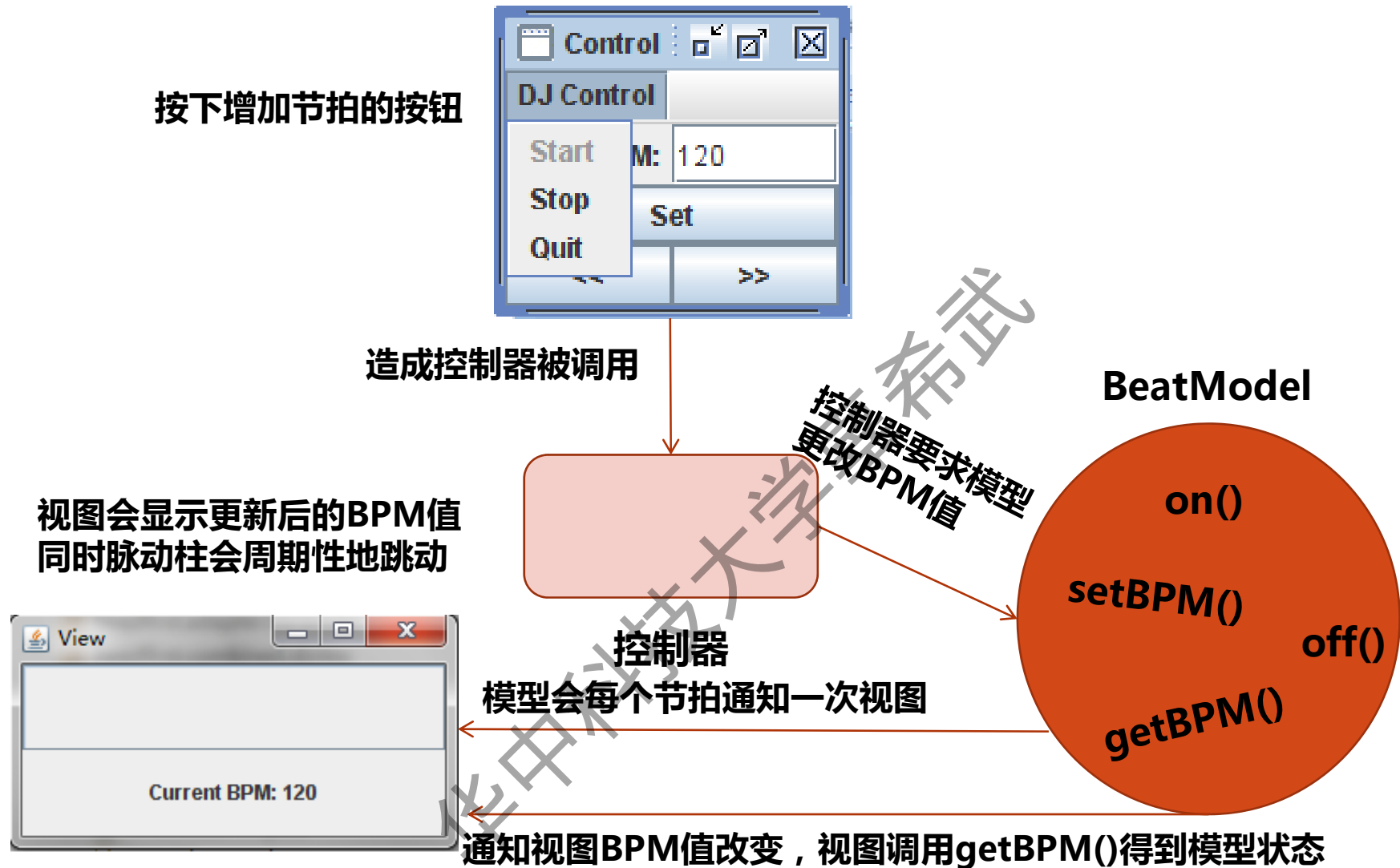
BeatModel也可以让我们通过getBPM()取得它当前的状态

BeatModel



利用MVC控制音乐节拍

□ 拼装在一起



利用MVC控制音乐节拍

□ 实现模型: 暴露一个接口, 让视图和控制器获得模型的状态

这些方法是
由控制器调用

```
public interface BeatModelInterface {  
    void initialize();
```

BeatModel被初始化后会调用此方法

```
    void on();
```

打开或关闭节拍器

```
    void off();
```

```
    void setBPM(int bpm);
```

设置或获取BPM

```
    int getBPM();
```

这些方法
允许视图
取得状态,
并成为观
察者

```
    void registerObserver(BeatObserver o);
```

```
    void removeObserver(BeatObserver o);
```

```
    void registerObserver(BPMObserver o);
```

```
    void removeObserver(BPMObserver o);  
}
```

注册或删除观察者。
分为二种观察者：
一种是周期性收到
节拍通知；一种是
BPM被改变时被通
知

利用MVC控制音乐节拍

□ 定义二种观察者接口

```
public interface BPMObserver {  
    void updateBPM();  
}
```

```
public interface BeatObserver {  
    void updateBeat();  
}
```

华中科技大学 辜希武

利用MVC控制音乐节拍

□ 实现BeatModel

```
public class BeatModel implements BeatModelInterface, MetaEventListener {  
    Sequencer sequencer; //Sequencer对象负责产生真实的节拍  
    ArrayList beatObservers = new ArrayList();  
    ArrayList bpmObservers = new ArrayList();  
    int bpm = 90;  
    //其它实例变量  
    public void initialize() {  
        setUpMidi();  
        buildTrackAndStart();  
    }  
    public void on() {  
        sequencer.start();  
        setBPM(90);  
    }  
    public void off() {  
        setBPM(0);  
        sequencer.stop();  
    }  
    public int getBPM() {  
        return bpm;  
    }  
}
```

MIDI代码需要的接口，在
javax.sound.midi里定义

BeatModel持有二种观察
者列表

设置Midi和音轨

开始定序器，将BPM设置为默认值90

将BPM设置为0，停止定序器

利用MVC控制音乐节拍

□ 实现BeatModel (续)

```
public class BeatModel implements BeatModelInterface, MetaEventListener {  
    public void setBPM(int bpm) {  
        this.bpm = bpm;  
        sequencer.setTempoInBPM(getBPM());  
        notifyBPMObservers();  
    }  
    void beatEvent() {  
        notifyBeatObservers();  
    }  
    public void registerObserver(BeatObserver o) {  
        beatObservers.add(o);  
    }  
  
    public void notifyBeatObservers() {  
        for(int i = 0; i < beatObservers.size(); i++) {  
            BeatObserver observer = (BeatObserver)beatObservers.get(i);  
            observer.updateBeat();  
        }  
    }  
}
```

设置BPM实例变量；要求定序器改变BPM；通知所有BPM观察者

注意此方法不是公有的。当新的节拍开始时，MIDI API会调用此方法，它会通知所有的BeatObserver新的节拍开始了

利用MVC控制音乐节拍

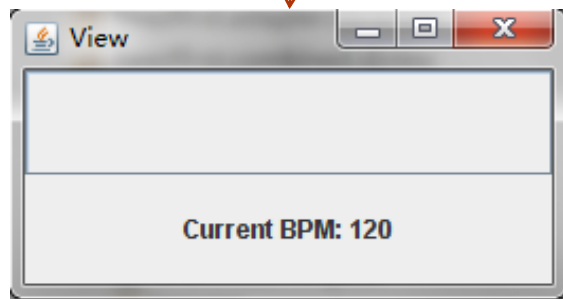
□ 实现BeatModel (续)

```
public class BeatModel implements BeatModelInterface, MetaEventListener {  
    public void registerObserver(BPMObserver o) {  
        bpmObservers.add(o);  
    }  
    public void notifyBPMObservers() {  
        for(int i = 0; i < bpmObservers.size(); i++) {  
            BPMObserver observer = (BPMObserver)bpmObservers.get(i);  
            observer.updateBPM();  
        }  
    }  
    public void removeObserver(BeatObserver o) {  
        int i = bpmObservers.indexOf(o);  
        if (i >= 0) bpmObservers.remove(i);  
    }  
    public void removeObserver(BPMObserver o) {  
        int i = bpmObservers.indexOf(o);  
        if (i >= 0) bpmObservers.remove(i);  
    }  
    // 处理节拍的MIDI代码  
}
```

利用MVC控制音乐节拍

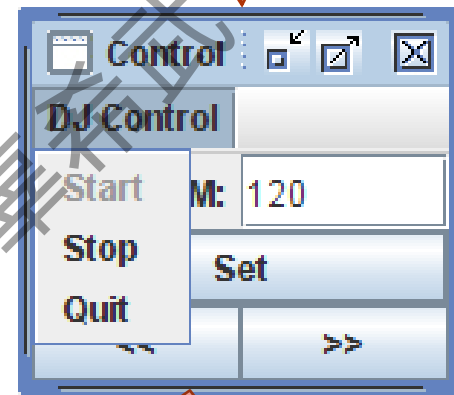
- 挂接视图，使BeatModel可视化
- 视图分为二个部分：一个窗口负责显示当前的BPM值和脉动柱；另一个负责界面控制。虽然是二个分离的窗口，但用同一个类实现

把模型的视图和控制界面的视图分离



模型视图显示模型的二个方面：

- 1：当前的BPM（来自BPMObserver的通知）
- 2：脉动的“节拍柱”（来自BeatObserver的通知）



控制的视图用来改变节拍，会把用户的输入传给控制器

利用MVC控制音乐节拍

□ 实现视图（模型视图部分）

```
public class DJView implements ActionListener, BeatObserver, BPMObserver {
```

```
    BeatModelInterface model;  
    ControllerInterface controller;
```

模型的引用

控制器的引用

```
    JFrame viewFrame;  
    JPanel viewPanel;  
    BeatBar beatBar;  
    JLabel bpmOutputLabel;
```

这几个GUI组件用来显示模型的节拍和当前BPM

```
    public DJView(ControllerInterface controller, BeatModelInterface model) {  
        this.controller = controller;  
        this.model = model;  
        model.registerObserver((BeatObserver)this);  
        model.registerObserver((BPMObserver)this);  
    }
```

构造函数得到控制器和模型的引用

```
    public void createView() {  
        //创建所有的Swing组件  
    }
```

```
}
```

利用MVC控制音乐节拍

□ 实现视图（模型视图部分续）

```
public class DJView implements ActionListener, BeatObserver, BPMObserver {  
    public void updateBPM() {  
        if (model != null) {  
            int bpm = model.getBPM();  
            if (bpm == 0) {  
                if (bpmOutputLabel != null) {  
                    bpmOutputLabel.setText("offline");  
                }  
            } else {  
                if (bpmOutputLabel != null) {  
                    bpmOutputLabel.setText("Current BPM: " + model.getBPM());  
                }  
            }  
        }  
    }  
    public void updateBeat() {  
        if (beatBar != null) {  
            beatBar.setValue(100);  
        }  
    }  
}
```

实现BPMObserver接口的updateBPM()方法。当模型状态（BPM值）发生改变时，该方法被调，用来更新当前BPM的显示

实现BeatObserver接口的updateBeat()方法。当模型开始新的一个节拍时，该方法被调，用来使脉动柱跳动一下：把脉动柱最大值设为100，让beatBar自己处理动画部分

利用MVC控制音乐节拍

□ 实现视图（控制视图部分）

```
public class DJView implements ActionListener, BeatObserver, BPMObserver {
```

```
    BeatModelInterface model;  
    ControllerInterface controller;
```

模型的引用

控制器的引用

```
    JFrame controlFrame;  
    JPanel controlPanel;  
    JLabel bpmLabel;  
    JTextField bpmTextField;  
    JButton setBPMButton;  
    JButton increaseBPMButton;  
    JButton decreaseBPMButton;  
    JMenuBar menuBar;  
    JMenu menu;  
    JMenuItem startMenuItem;  
    JMenuItem stopMenuItem;  
    JMenuItem exitMenuItem;
```

这里是所有和控制有关的GUI组件

```
}
```


利用MVC控制音乐节拍

□ 实现视图（控制视图部分续）

```
public class DJView implements ActionListener, BeatObserver, BPMObserver {  
  
    public void createControls() {  
        //... 创建所有的控制组件  
        startMenuItem.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                controller.start();  
            }  
        });  
  
        stopMenuItem.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                controller.stop();  
            }  
        });  
  
        exitMenuItem.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                System.exit(0);  
            }  
        });  
    }  
}
```

这里创建所有的控制组件, 其中为三个菜单项添加了事件监听器, 在事件处理函数里调用了控制器的方法

利用MVC控制音乐节拍

□ 实现视图（控制视图部分续）

```
public class DJView implements ActionListener, BeatObserver, BPMObserver {  
  
    public void enableStopMenuItem() {  
        stopMenuItem.setEnabled(true);  
    }  
  
    public void disableStopMenuItem() {  
        stopMenuItem.setEnabled(false);  
    }  
  
    public void enableStartMenuItem() {  
        startMenuItem.setEnabled(true);  
    }  
  
    public void disableStartMenuItem() {  
        startMenuItem.setEnabled(false);  
    }  
}
```

这些方法把菜单项Start和Stop变成enable或disable。这些方法被控制器调用以改变用户界面

华中科技大学

利用MVC控制音乐节拍

□ 实现视图（控制视图部分续）

```
public class DJView implements ActionListener, BeatObserver, BPMObserver {  
  
    public void actionPerformed(ActionEvent event) {  
        if (event.getSource() == setBPMButton) {  
            int bpm = Integer.parseInt(bpmTextField.getText());  
            controller.setBPM(bpm);  
        } else if (event.getSource() == increaseBPMButton) {  
            controller.increaseBPM();  
        } else if (event.getSource() == decreaseBPMButton) {  
            controller.decreaseBPM();  
        }  
    }  
}
```

处理Button的点击事件，会调用相应的控制器方法

利用MVC控制音乐节拍

- 实现控制器
- 控制器是策略。由于要实现的是策略模式，因此首先定义可以插进视图的策略接口

```
public interface ControllerInterface {  
    void start();  
    void stop();  
    void increaseBPM();  
    void decreaseBPM();  
    void setBPM(int bpm);  
}
```

视图通过控制器接口调用控制器方法

控制器的接口方法，和模型接口相比，控制器接口方法更丰富，如可以对BPM值“增1”或“减1”

利用MVC控制音乐节拍

□ 控制器的实现

```
public class BeatController implements ControllerInterface {
```

```
    BeatModelInterface model;  
    DJView view;
```

控制器是模型和视图的粘合剂，因此必须同时拥有模型和视图的引用

```
    public BeatController(BeatModelInterface model) {  
        this.model = model;  
        view = new DJView(this, model);  
        view.createView();  
        view.createControls();  
        view.disableStopMenuItem();  
        view.enableStartMenuItem();  
        model.initialize();  
    }
```

构造函数传入模型引用

这里创建视图对象，传入控制器引用和模型引用

```
}
```

华中科技大学

利用MVC控制音乐节拍

□ 控制器的实现（续）

```
public class BeatController implements ControllerInterface {  
    public void start() {  
        model.on();  
        view.disableStartMenuItem();  
        view.enableStopMenuItem();  
    }  
    public void stop() {  
        model.off();  
        view.disableStopMenuItem();  
        view.enableStartMenuItem();  
    }  
    public void increaseBPM() {  
        int bpm = model.getBPM();  
        model.setBPM(bpm + 1);  
    }  
    public void decreaseBPM() {  
        int bpm = model.getBPM();  
        model.setBPM(bpm - 1);  
    }  
    public void setBPM(int bpm) {  
        model.setBPM(bpm);  
    }  
}
```

用户点击 Start菜单项时，控制器调用 model.on(), 然后改变用户界面

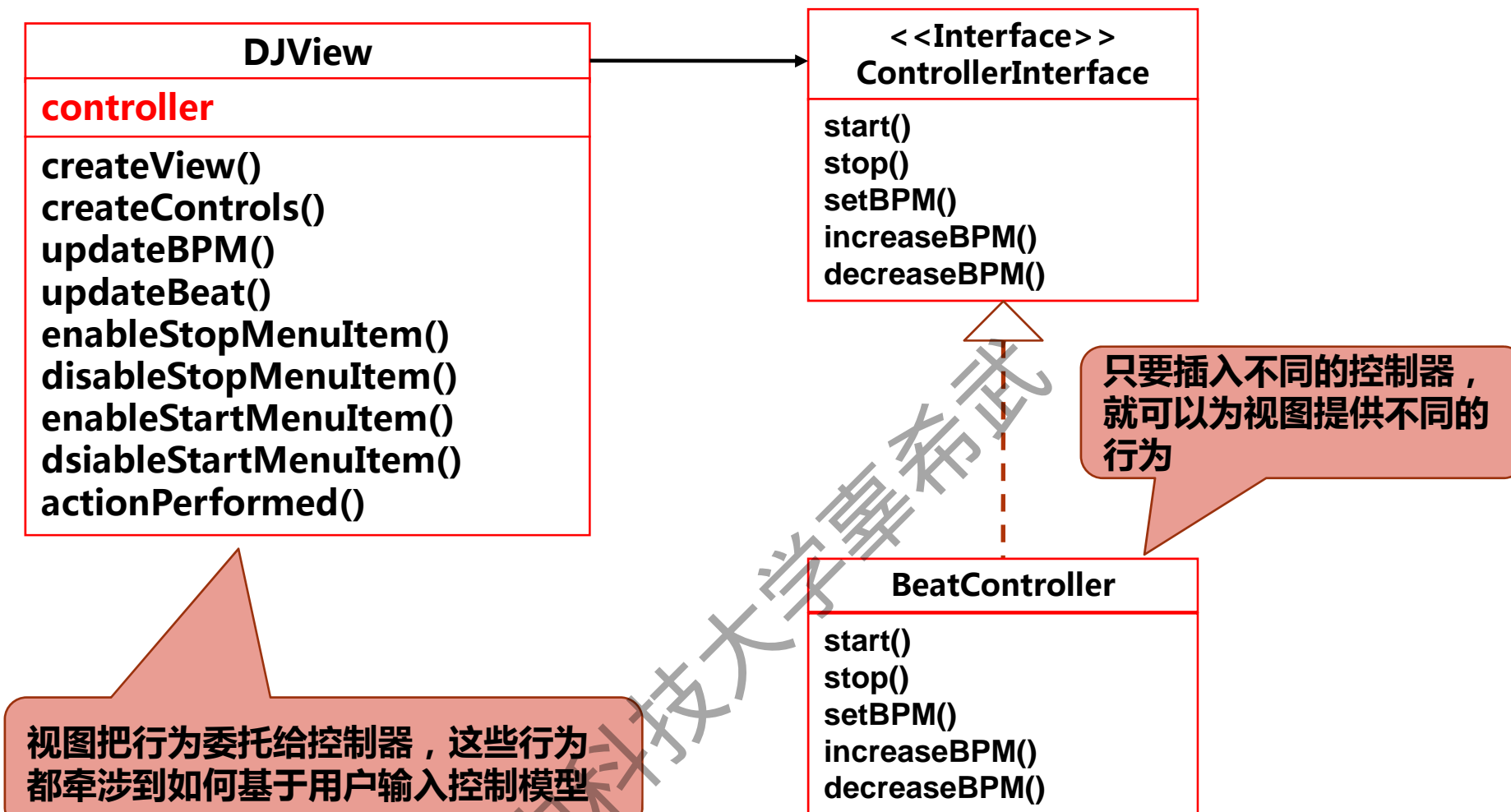
用户点击 Stop菜单项时，控制器调用 model.off(), 然后改变用户界面

用户点击“增1”按钮时，控制器调用 model.getBPM()取得当前BPM值，+1后 再通过model.setBPM()设置BPM

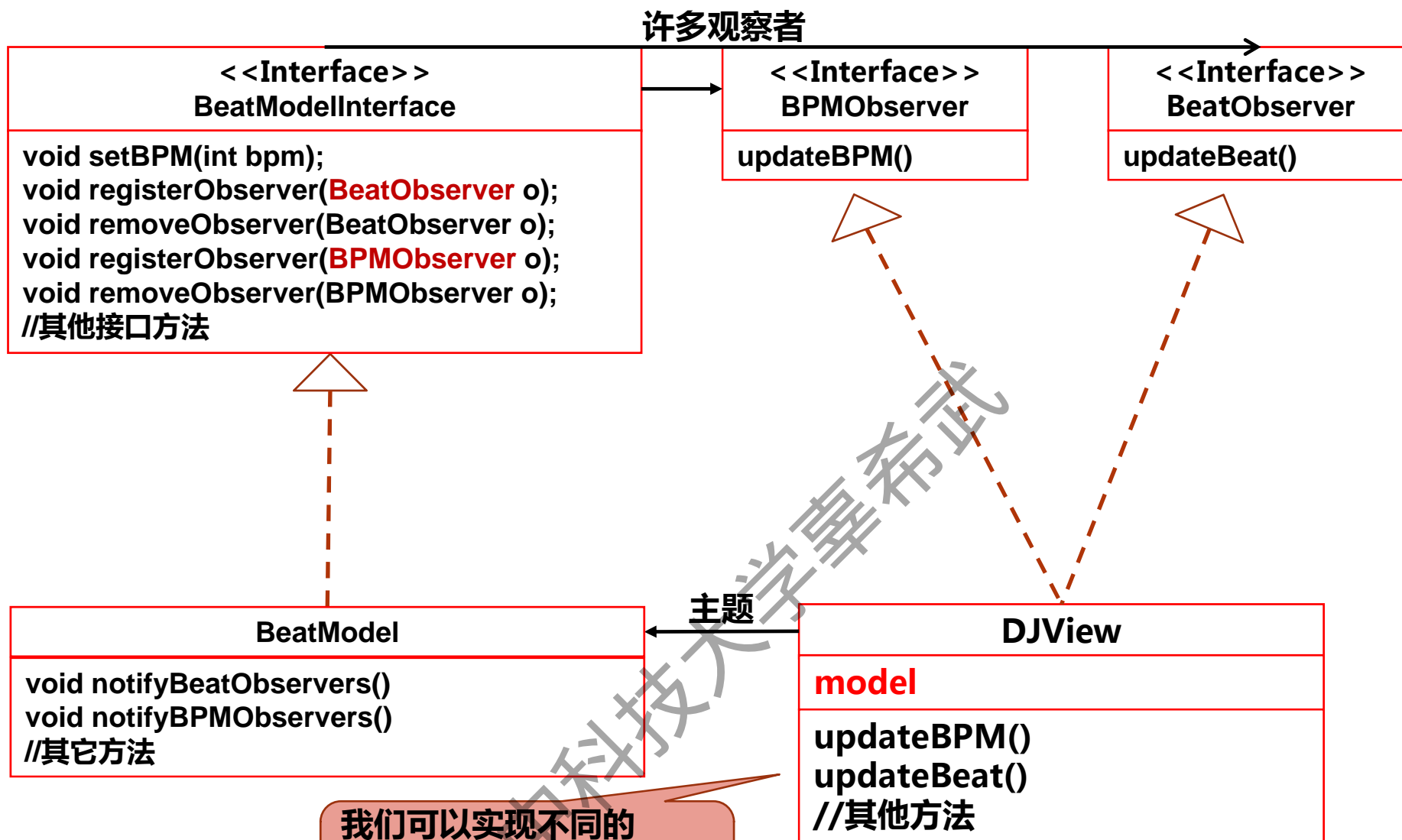
用户点击“减1”按钮时，控制器调用 model.getBPM()取得当前BPM值，-1后 再通过model.setBPM()设置BPM

用户点击“Set”按钮时，控制器调用 model.setBPM() 设置BPM

控制器和视图的策略模式结构

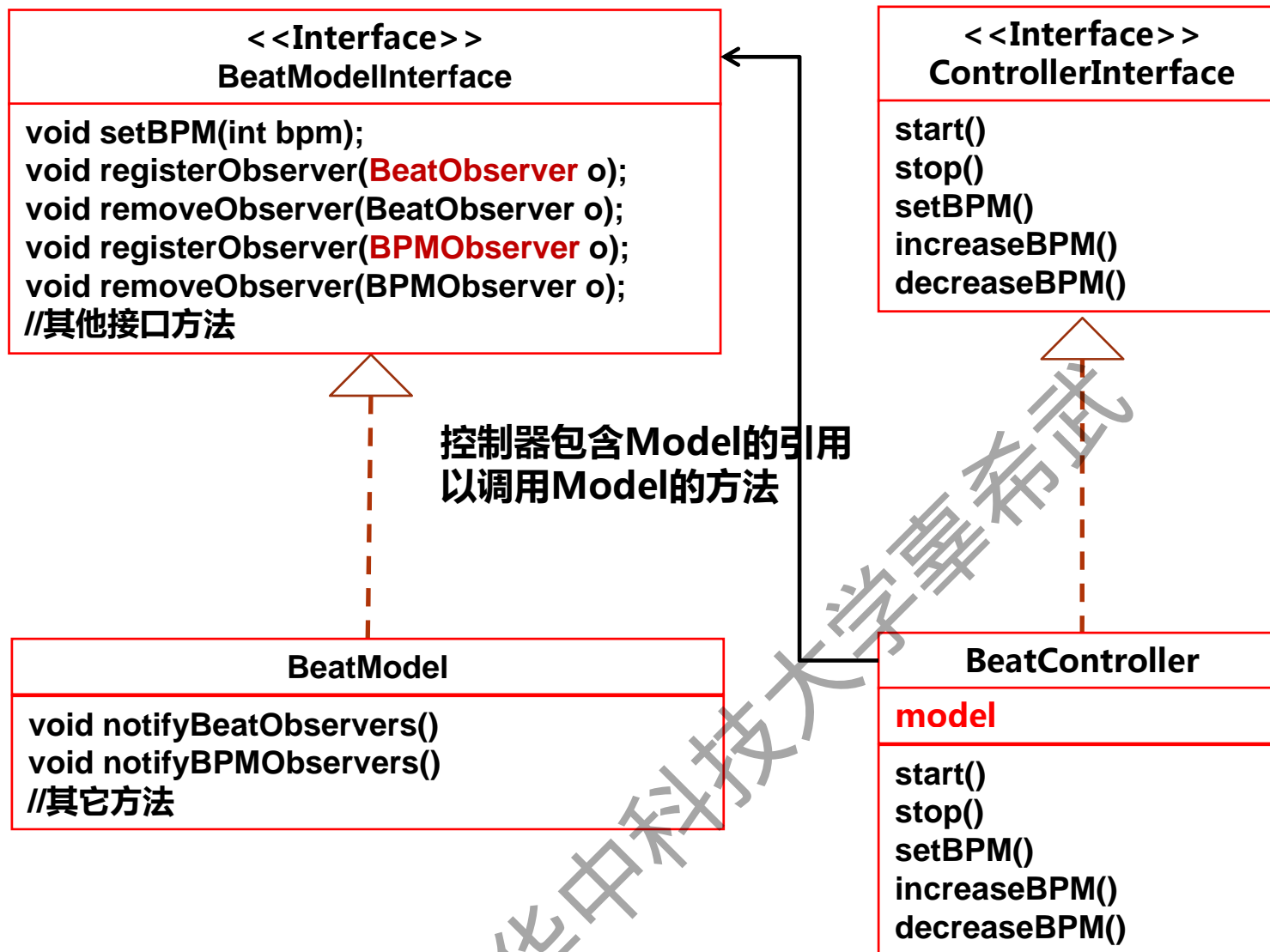


模型和视图的观察者模式结构



我们可以实现不同的View，使得一个模型对应多个View

模型和控制的关系结构



利用MVC控制音乐节拍

□ 测试

```
public class DJTestDrive {  
  
    public static void main (String[] args) {  
        BeatModelInterface model = new BeatModel();  
        ControllerInterface controller = new BeatController(model);  
    }  
}
```

华中科技大学 辜希武

MVC模式总结

- ❑ 模型使用了观察者模式，使得模型完全独立于控制器和视图。我们把应用程序的逻辑放在模型中实现，使得程序的逻辑和视图、控制完全无关
- ❑ 控制是模型和视图的粘合剂，将模型和视图完全解耦
 - ❑ 控制和视图之间实现了策略模式。控制是策略，视图把行为的处理完全委托给策略（控制），使得视图只需要专注于模型的可视化展示
 - ❑ 控制和模型之间是关联关系，控制包含了模型的引用，可以调用模型的方法
- ❑ 视图本身实现了组合模式，Java Swing API实现了组合模式
- ❑ 思考下如何支持多视图

MVC与Web

