




哈爾濱工業大學  
HARBIN INSTITUTE OF TECHNOLOGY

软件工程  
第六章 OO分析与设计  
6-1 面向对象的基本概念

王忠杰  
rainy@hit.edu.cn

2017年10月31日

# 主要内容

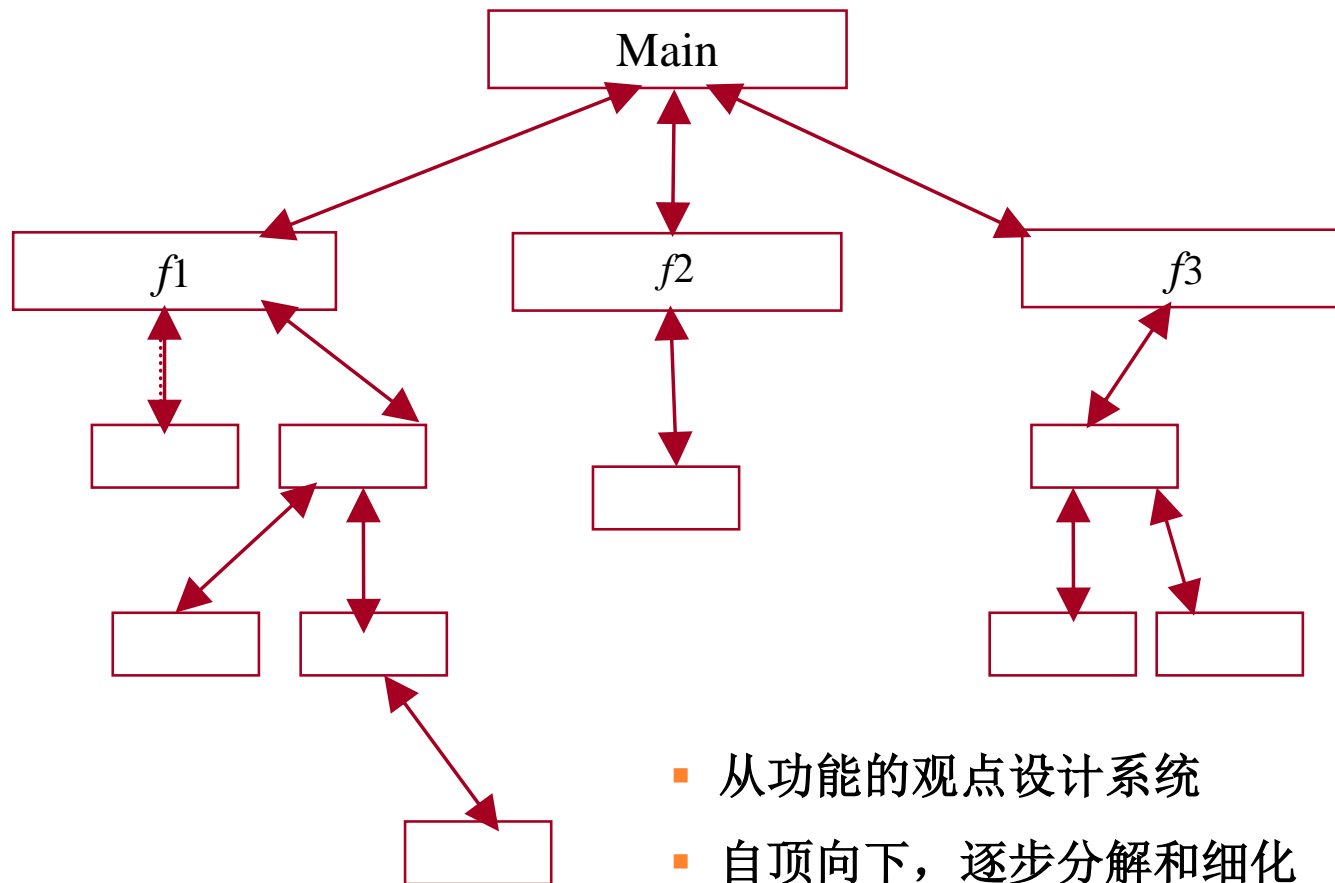
- 
- 1 面向对象技术概述
  - 2 面向对象的基本概念
  - 3 对象之间的五类关系



# 1 面向对象技术概述



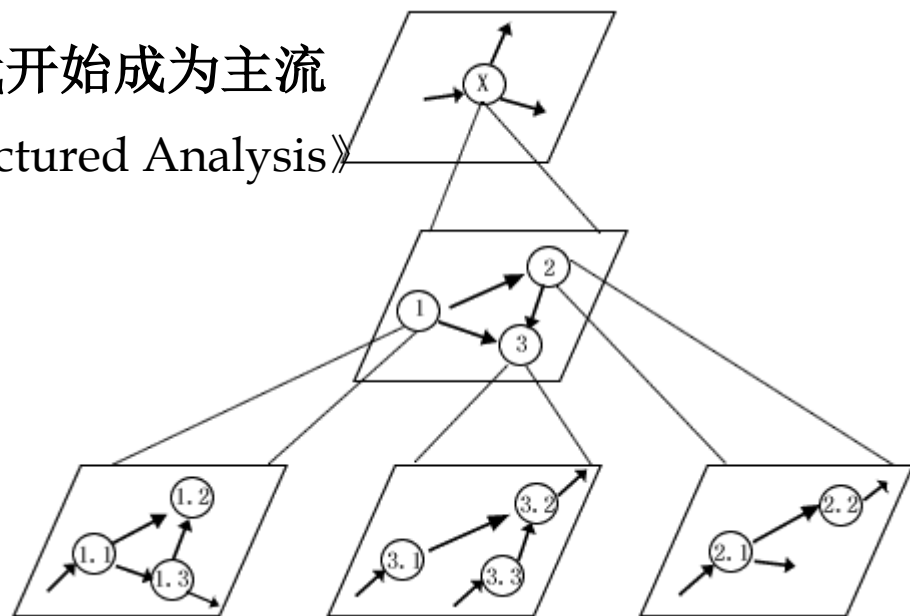
# 先看结构化程序开发...



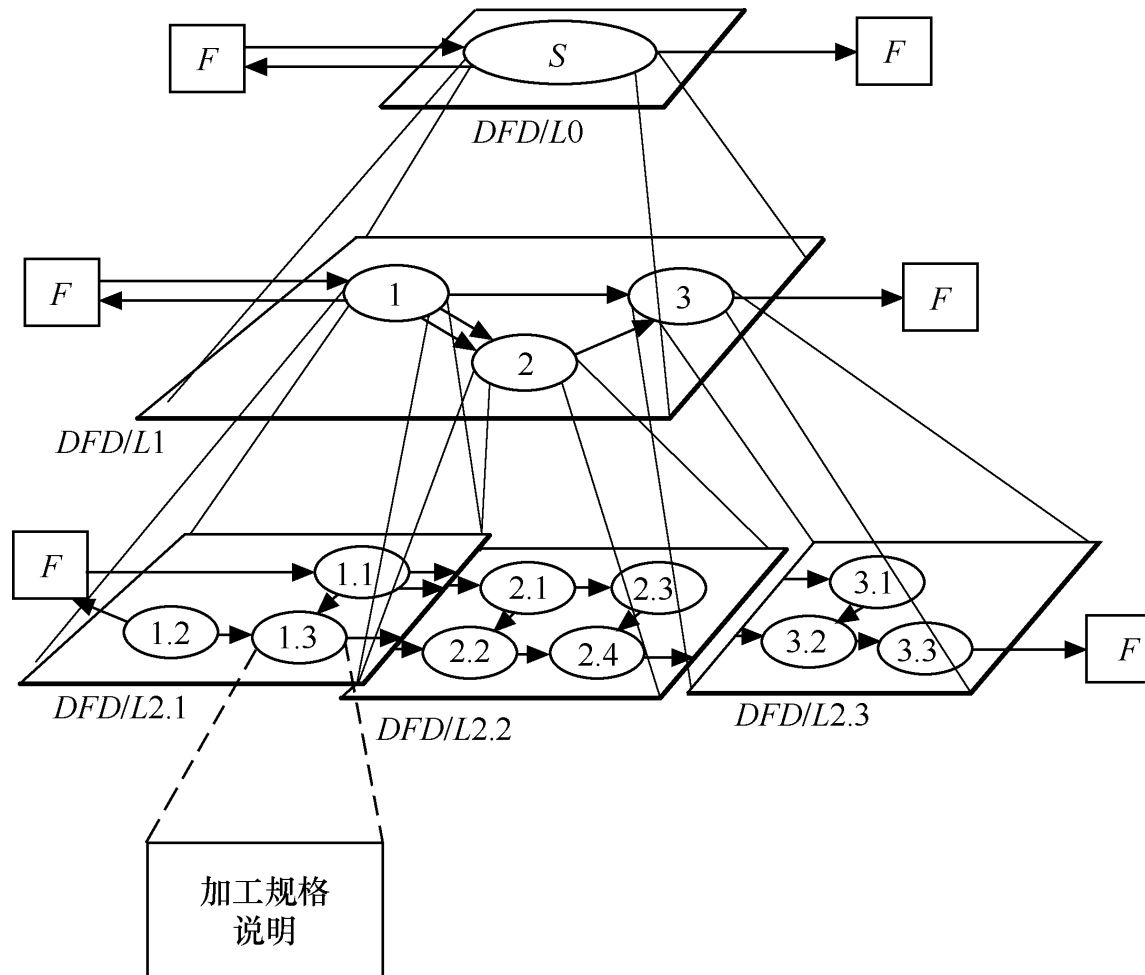
- 从功能的观点设计系统
- 自顶向下，逐步分解和细化
- 将大系统分解为若干模块，主程序调用这些模块实现完整的系统功能

# 结构化分析方法

- **结构化分析方法(SA): 将待解决的问题看作一个系统, 从而用系统科学的思想方法(抽象、分解、模块化)来分析和解决问题。**
  - 起源于结构化程序设计语言(事先设计好每一个具体的功能模块, 然后将这些设计好的模块组装成一个软件系统);
  - 以动词性的“功能”为核心展开分解。
- **最早产生于1970年代中期, 1980年代开始成为主流**
  - Yourdon于1989年出版《Modern Structured Analysis》
- **核心思想:**
  - 自顶向下的分解(top-down)



# 结构化分析方法



# 结构化程序开发的特点

- 把软件视为处理数据的流，并定义成由一系列步骤构成的算法；
- 每一步骤都是带有预定输入和特定输出的一个过程；
- 把这些步骤串联在一起可产生合理的稳定的贯通于整个程序的控制流，最终产生一个简单的具有静态结构的体系结构。
- 数据抽象、数据结构根据算法步骤的要求开发，它贯穿于过程，提供过程所要求操作的信息；
- 系统的状态是一组全局变量，这组全局变量保存状态的值，把它们从一个过程传送到另一个过程。
- 结构化软件=算法+数据结构
- 结构化需求分析 = 结构化语言+数据字典+数据流图(DFD)

# 结构化方法的常见问题

## ■ 需求的错误

- 不完整、不一致、不明确
- 开发人员和用户无法以同样的方式说明需求

## ■ 需求的变化

- 需求在整个项目过程中始终发生变化
- 设计后期发生改变

## ■ 持续的变化

- 系统功能不断变化
- 许多变化出现在项目后期
- 维护过程中发生许多变化

## ■ 系统结构的崩溃

- 系统在不断的变化中最终变得不可用

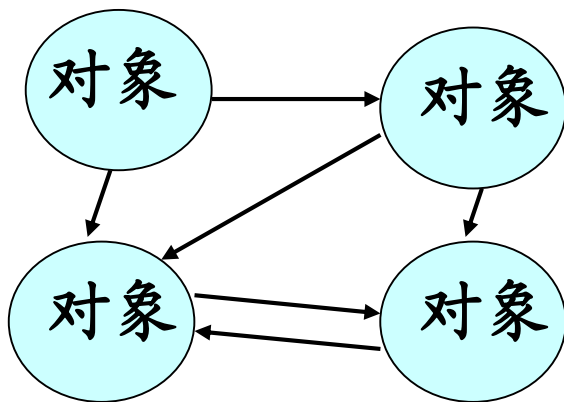


## 造成上述问题的根本原因...

- **结构化方法以功能分解和数据流为核心**，但是...系统功能和数据表示极有可能发生变化；
  - 以ATM银行系统为例：帐户的可选项、利率的不同计算方式、**ATM**的不同界面；
- **而软件设计应尽可能去描述那些极少发生变化的稳定要素：对象**
  - 银行客户、帐户、**ATM**

## 再看面向对象的程序开发...

- 系统被看作对象的集合;
- 每个对象包含一组描述自身特性的数据以及作用在数据上的操作(功能集合)。



# 面向对象的程序开发

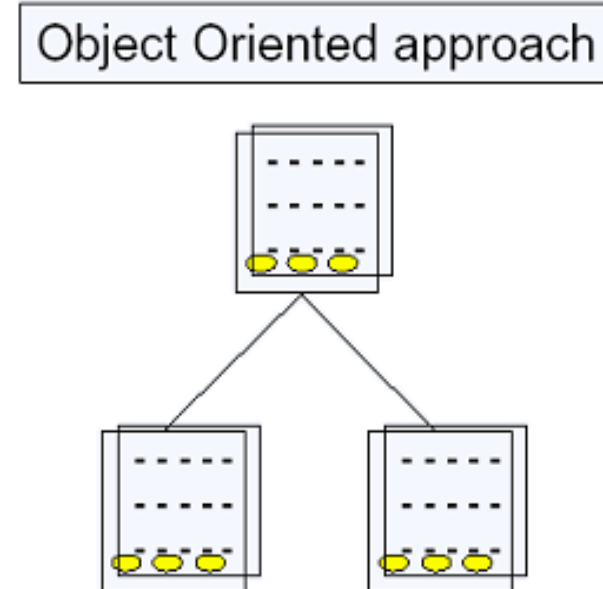
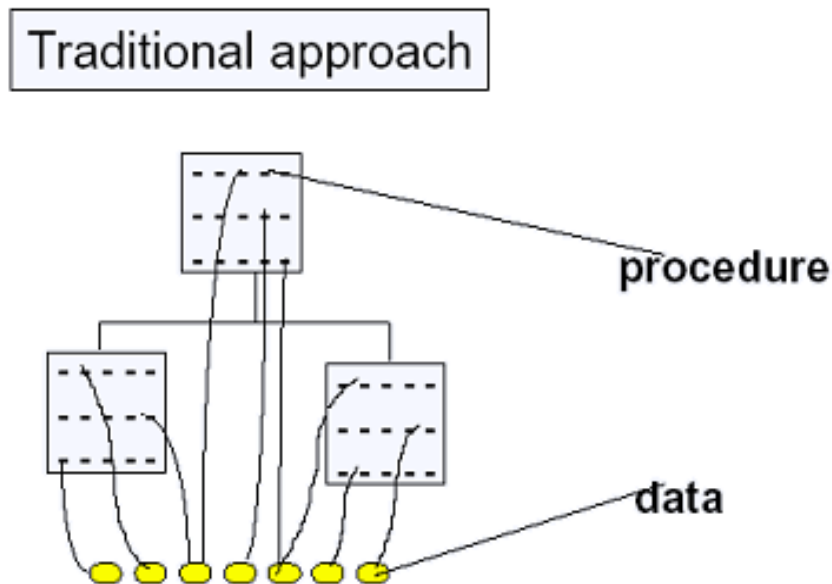
- 在**结构化**程序开发模式中优先考虑的是**过程抽象**，在**面向对象**开发模式中优先考虑的是**实体**(问题论域的对象)；
- 在面向对象开发模式中，把标识和模型化问题论域中的主要实体做为系统开发的起点，主要考虑对象的行为而不是必须执行的一系列动作；
  - 对象是数据抽象与过程抽象的综合；
  - 系统的状态保存在各个数据抽象的所定义的数据存储中；
  - 控制流包含在各个数据抽象中的操作内；
  - 消息从一个对象传送到另一个对象；
  - 算法被分布到各种实体中。

# 面向对象方法的优势

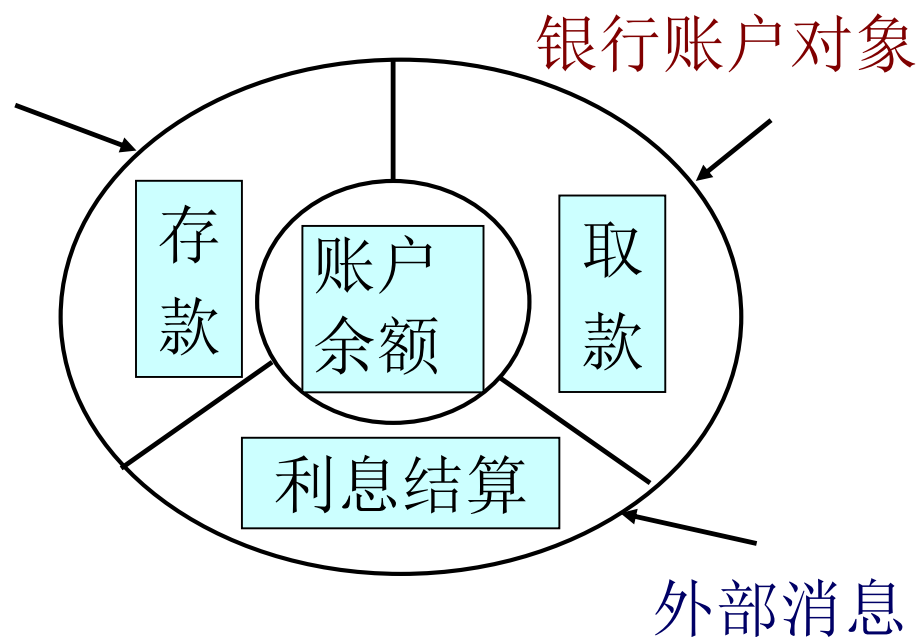
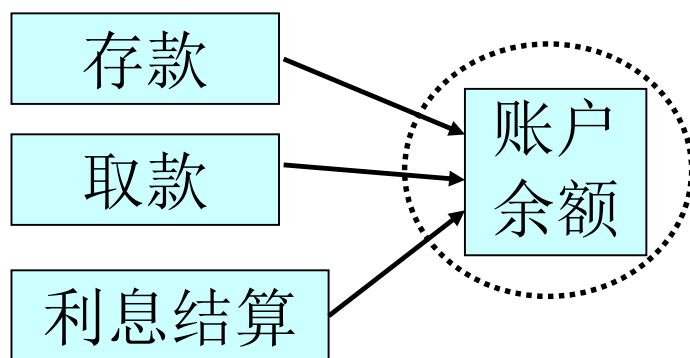
- 面向对象模型更接近于问题域(尽可能模拟人类习惯的思维方式)
  - 以问题域中的对象为基础建模
  - 以对象、属性和操作对问题进行建模
- 反复细化高层模型直到可以实现的程度
  - 努力避免在开发过程中出现大的概念跳变
- 将模型组织成对象的集合
  - 真实世界中的具体事物
    - 驾驶执照、信用卡、飞机等
  - 逻辑概念
    - 操作系统中的分时策略、军事训练中的冲突解决规则等

## 二者的本质区别

- 面向过程的结构化系统 = 功能 + 数据 (软件=算法+数据结构)
- 面向对象的系统 = 对象 + 消息 (对象=数据属性+操作)



## 二者的本质区别



# 面向对象方法的发展历史

## ■ 初始阶段

- 1960's: Simula编程语言
- 1970's: Smalltalk编程语言

## ■ 发展阶段

- 1980's: 理论基础, 许多OO 编程语言(如C++, Objective-C等)

## ■ 成熟阶段

- 1990's: 面向对象分析和设计方法(Booch, OMT等), Java
- 1997: OMG 组织的统一建模语言(UML)
- 逐渐替代了传统的结构化方法

# 面向对象的软件工程

- **面向对象分析(Object Oriented Analysis, OOA)**

- 分析和理解问题域，找出描述问题域和系统责任所需的类及对象，分析它们的内部构成和外部关系，建立OOA 模型。

- **面向对象设计(Object Oriented Design, OOD)**

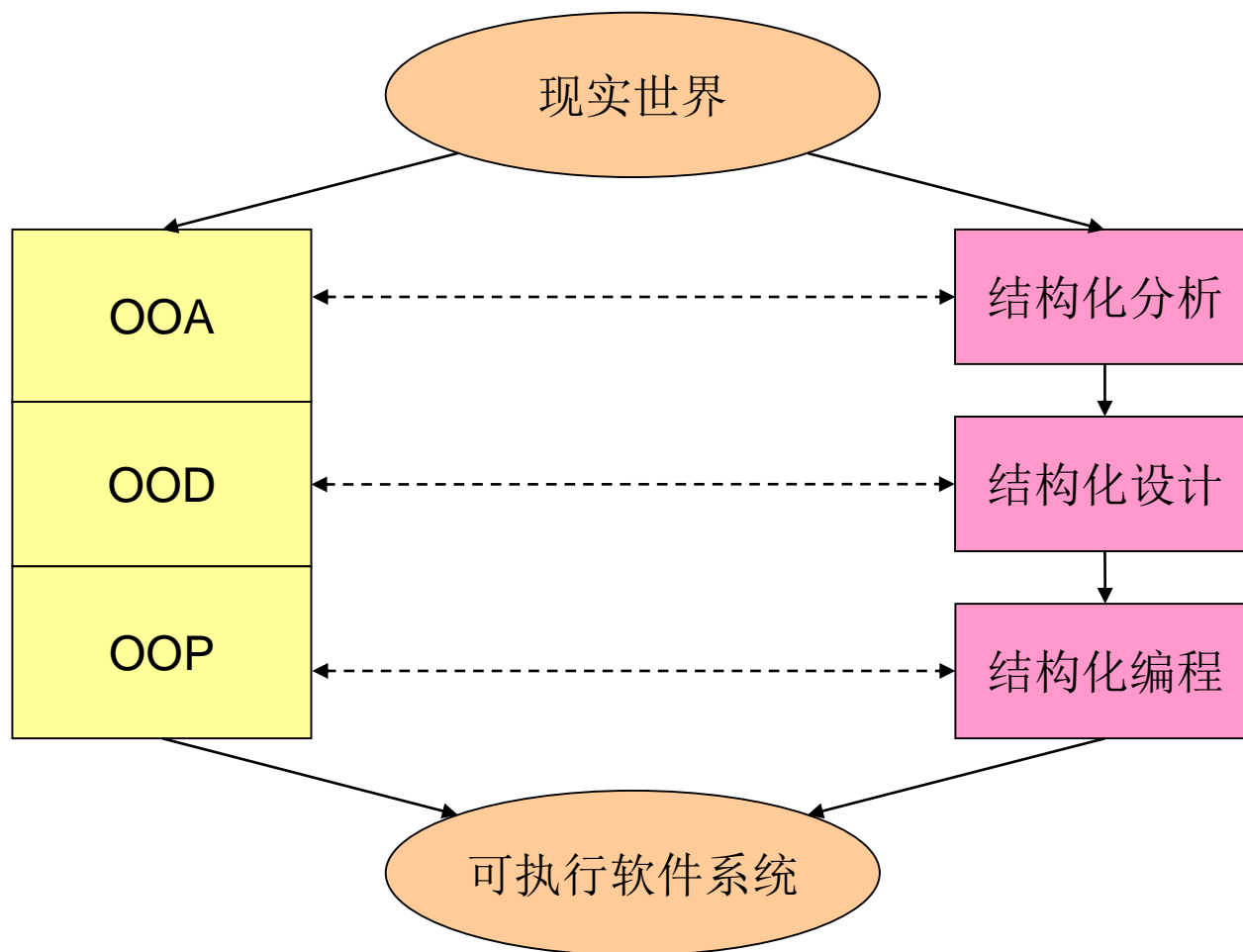
- 将OOA 模型直接变成OOD 模型，并且补充与一些实现有关的部分，如人机界面、数据存储、任务管理等。

- **面向对象编程(Object Oriented Programming, OOP)**

- 用一种面向对象的编程语言将OOD 模型中的各个成分编写成程序，由于从OOA→OOD→OOP实现了无缝连接和平滑过渡，因此提高了开发工作的效率和质量。



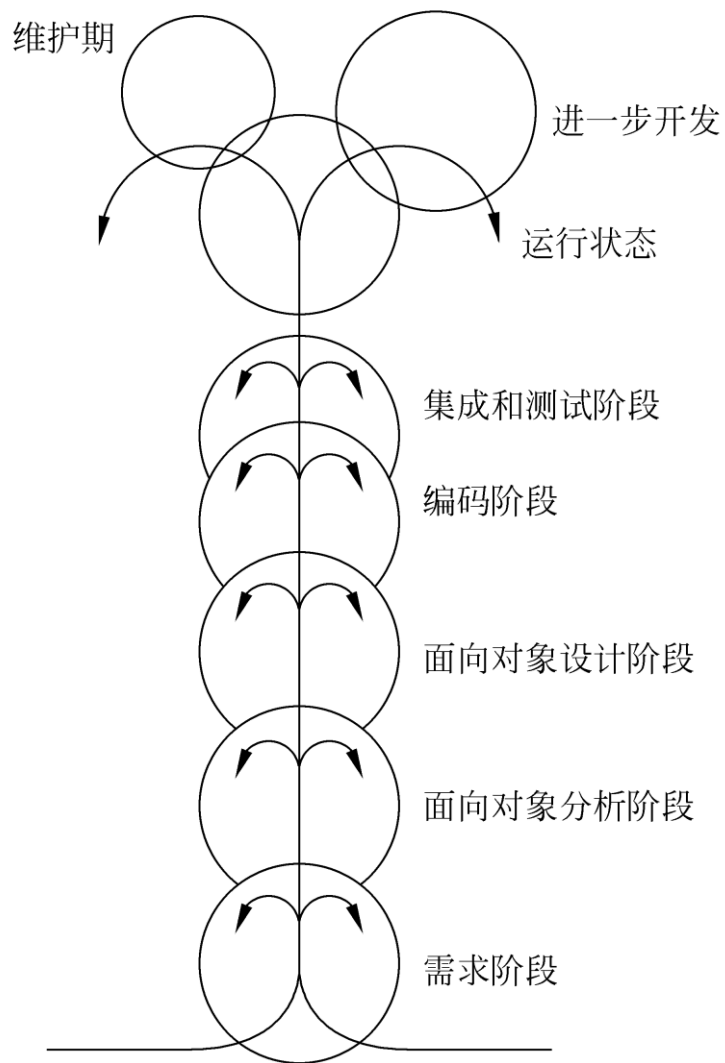
# 面向对象的软件工程



# OO中的喷泉过程模型

## ■ 喷泉模型：

- 在OO开发过程中，各阶段之间形成频繁的迭代；
- OO各阶段均采用统一的“对象”概念，各阶段之间的区分变得不明显，形成“无缝”连接，从而容易实现多次反复迭代。



# 小结：面向对象技术

## ■ 面向对象技术(Object Oriented Technology)

- 客观世界是由**对象**组成的，任何客观事物或实体都是对象；复杂对象可以由简单对象构成；
- 具有相同数据和相同操作的对象可以归并为一个统一的“**类**”，对象是类的实例；
- 类可以派生出子类，子类**继承**父类的全部特性(数据和操作)，同时加入了自己的新特性；子类和父类形成层次结构；
- 对象之间通过**消息**传递相互关联；
- 类具有**封装**性，其数据和操作对外是不可见的，外界只能通过消息请求某些操作。
- 具体的**计算**则是通过新对象的建立和对象之间的通信来执行的。



## 2 面向对象的基本概念

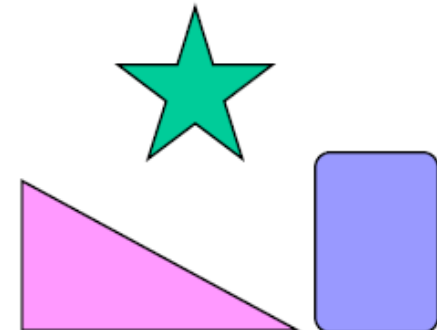
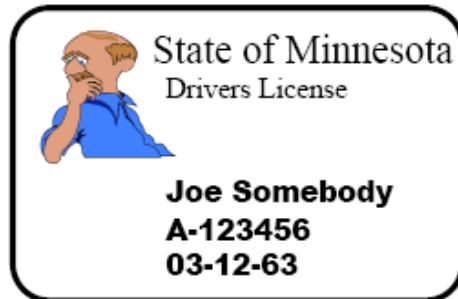


# 面向对象的基本概念

- 对象(Object)/类(Class)
- 封装(Encapsulation)/信息隐藏(Information Hiding)
- 接口(Interface)/实现(Realization)、消息(Message)
- 多态(Polymorphism)、重写(Override)/重载(Overload)
- 类间关系(Relationships)
  - 继承(Inheritance)/泛化(Generalization)
  - 聚合(Aggregation)/组合(Composition)
  - 关联(Association)
  - 依赖(Dependency)
- 抽象类(Abstract Class)

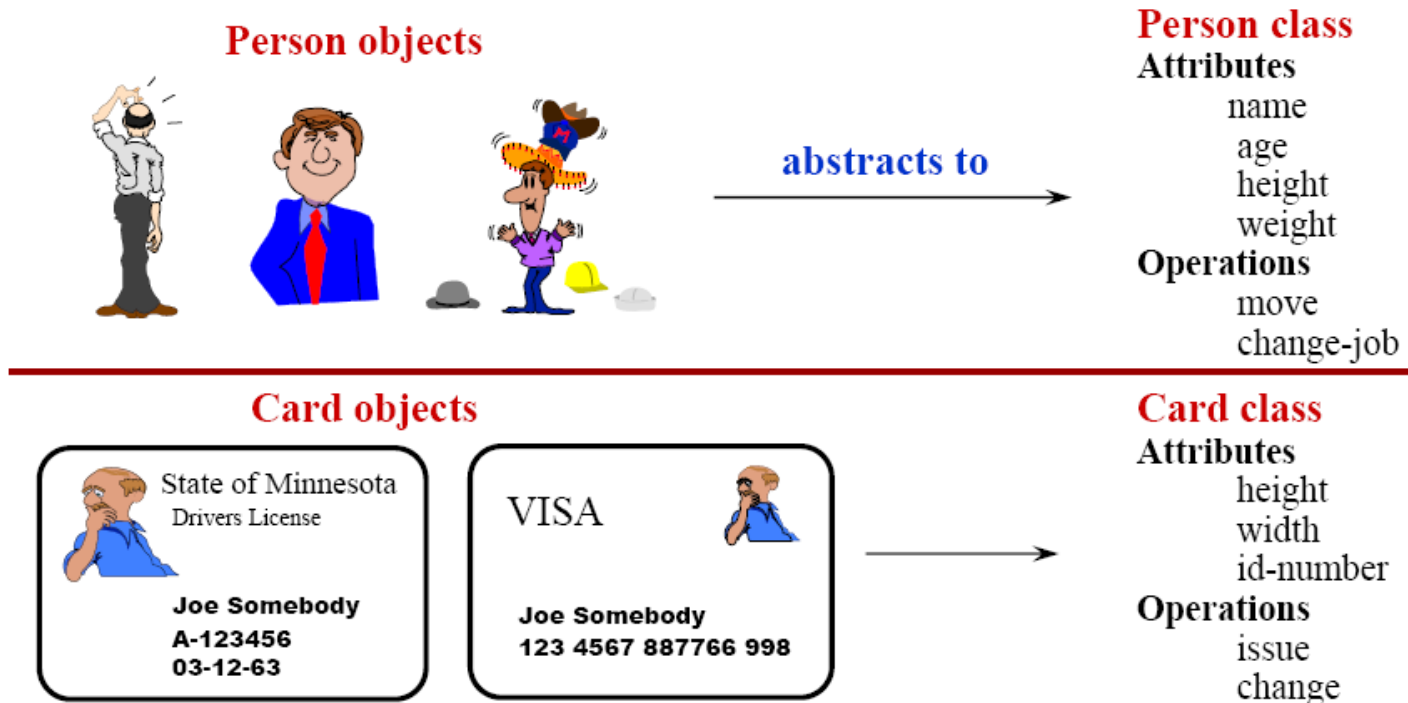
# 对象(Object)

- **对象(Object):** 用来描述客观事物的实体，是构成系统的一个基本单位，由一组属性以及作用在这组属性的操作构成。



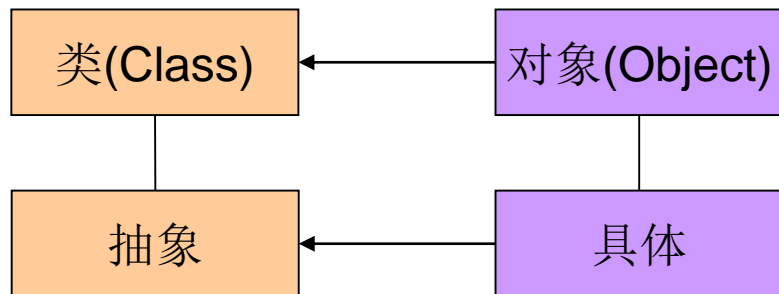
# 属性和操作

- **属性(Attribute):** 描述对象静态特性的数据项;
- **操作(Operation):** 描述对象动态特性的一个动作;



# 类(Class)

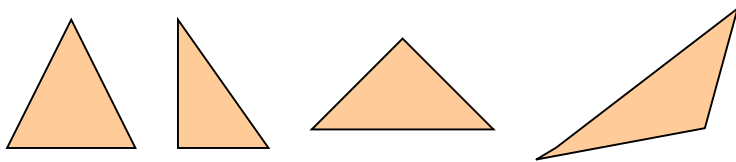
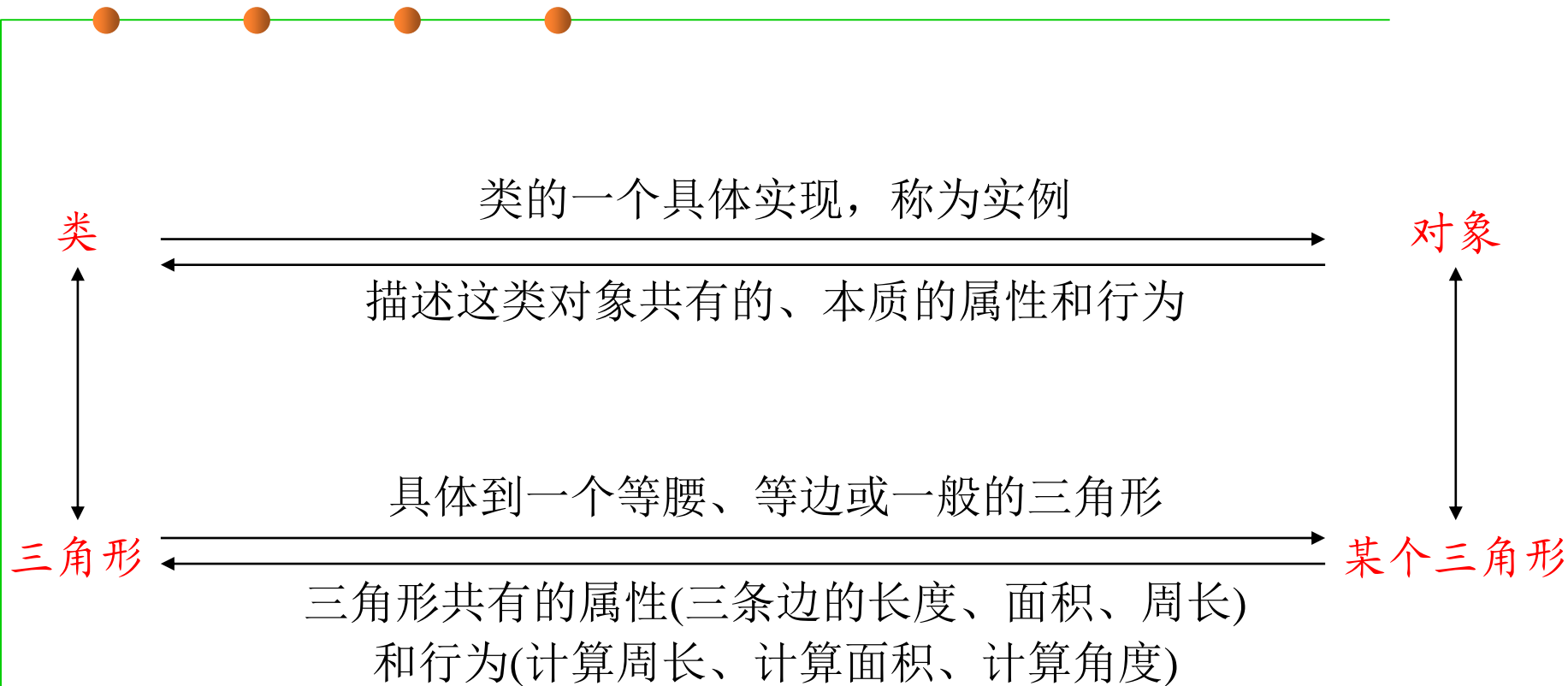
- **类(Class):** 具有相同属性和操作的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述。



- “类”所代表的是一个抽象的概念或事物，现实世界中并不存在，而“对象”是客观存在的。
- [例] “学生”是一个类，计算机学院140310101号学生则是“学生”类的一个实例，是一个具体的“对象”。



# 类与对象的对比



# 类与对象的对比

## ■ 类与对象的比较

- “同类对象具有相同的属性和操作”是指它们的定义形式相同，而不是说每个对象的属性值都相同。
  - 类是静态的，类的存在、语义和关系在程序执行前就已经定义好了。
  - 对象是动态的，对象在程序执行时可以被创建和删除。
- 
- 在面向对象的系统分析和设计中，并不需要逐个对对象进行说明，而是着重描述代表一批对象共性的类。

# 类与对象的对比

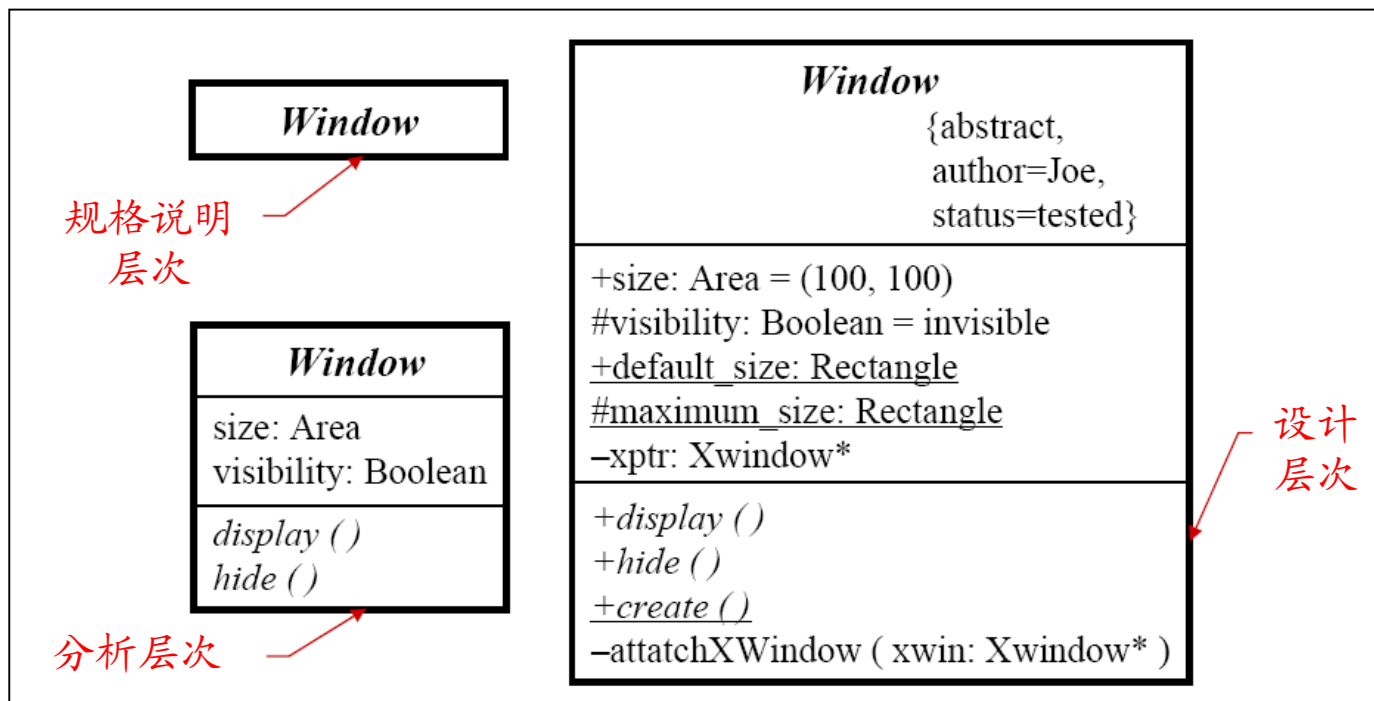
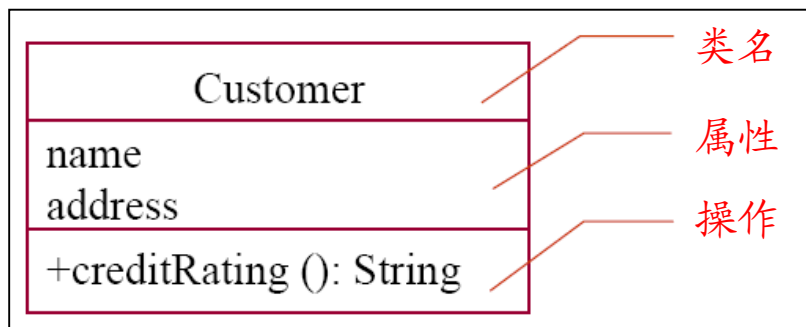
定义一个学生类:

```
Class Student {  
    String sno;  
    String sname;  
    String dept;  
  
    public Student (String sno, string  
sname, string dept) { };  
    public boolean RegisterMyself() { };  
    public boolean SelectCourses() { };  
    private float QueryScore (int courseID)  
    { };  
}
```

使用这个类:

```
Student ABC = new Student  
    ("140310101", "ABC", "CS" );  
Student DEF = new Student  
    ("140310102", "DEF", "CS" );  
Student GHI = new Student  
    ("140310103", "GHI", "CS" );  
  
If (ABC RegisterMyself() == true) {  
    ABC.SelectCourses();  
}  
float score = DEF.QueryScore (32);
```

# 类(Class)的三种抽象层次



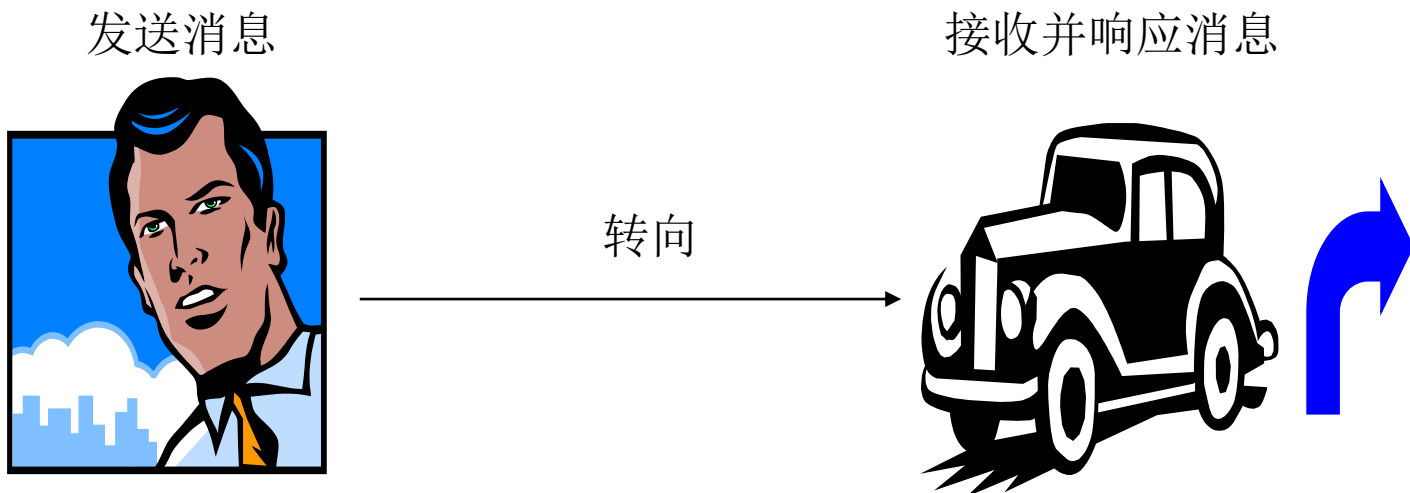


# 类的操作/方法(Operation/Method)

- 类的操作/方法：描述对象“动态” (行为)的特征的一个函数或过程；
- 方法的“可见性” (Visibility)分类：
  - 公有属性(public) +
  - 私有属性(private) -
  - 保护属性(protected) #
- 方法的表达方式：
  - 可见性 方法名(参数列表)：返回值数据类型
  - 例如：+ getNextSentence (int i) : string

# 消息(Message)

- **消息(Message):** 一个对象向其他对象发出的请求，一般包含提供服务的对象标识、服务标识、输入信息和应答信息等信息；
  - 一个对象向另一个对象发出消息请求某项服务；
  - 另一个对象接收该消息，触发某些操作，并将结果返回给发出消息的对象；
  - 对象之间通过消息通信彼此关联在一起，形成完整的系统。



# 封装(Encapsulation)

- **封装(Encapsulation):** 把对象的属性和操作结合成一个独立的单元，并尽可能对外界隐藏对象的内部实现细节；
- 对外界其他对象来说，不需了解对象内部是如何实现的(how to do)，只需要了解对象所呈现出来的外部行为(what to do)即可。
- 例如：对“汽车”对象来说，“司机”对象只能通过方向盘和仪表来操作“汽车”，而“汽车”的内部实现机制则被隐藏起来。



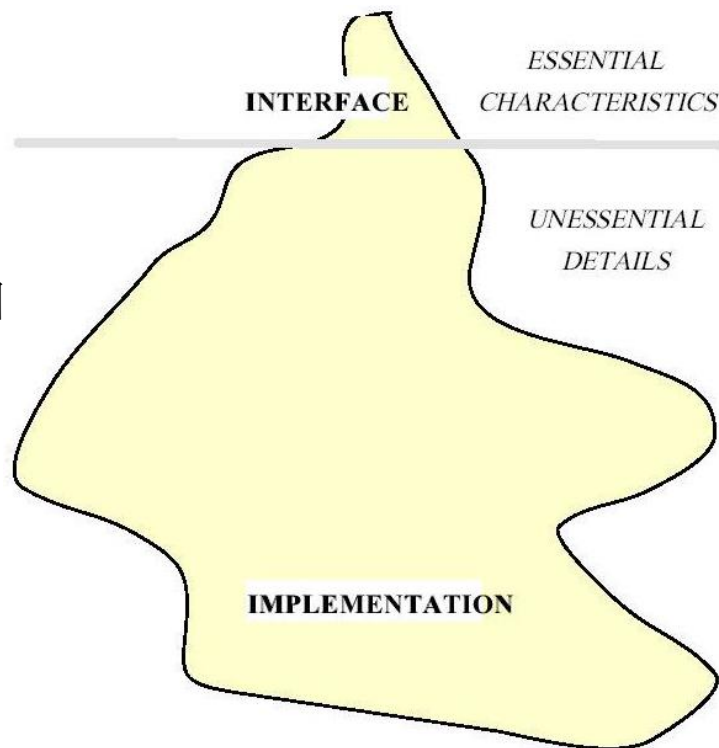


# 封装(Encapsulation)

- 使对象形成两个部分：接口(可见)和实现(不可见)，将对象所声明的功能(行为)与内部实现(细节)分离

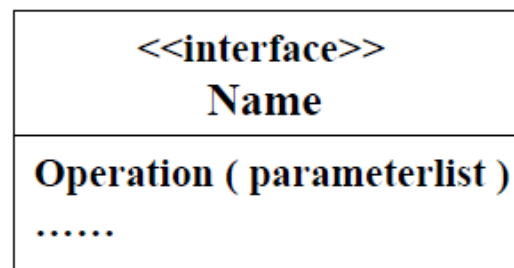
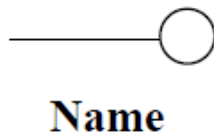
——信息隐藏(Information Hiding)

- “封装”的作用是什么？
  - 保护对象，避免用户误用；
  - 保护客户端，其实现过程的改变不会影响到相应客户端的改变。



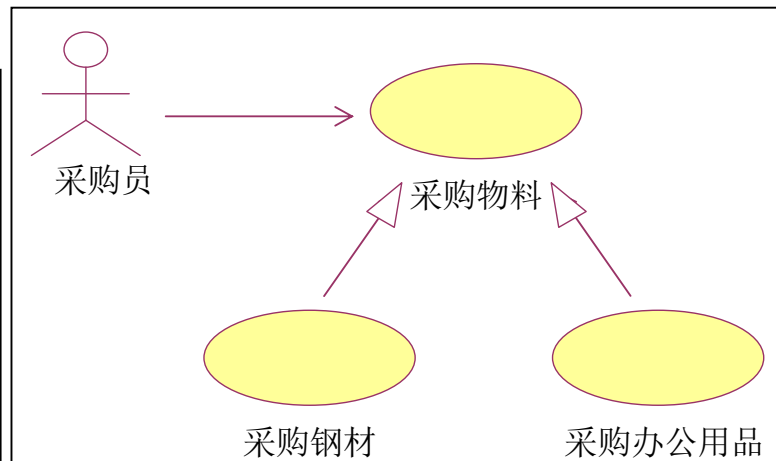
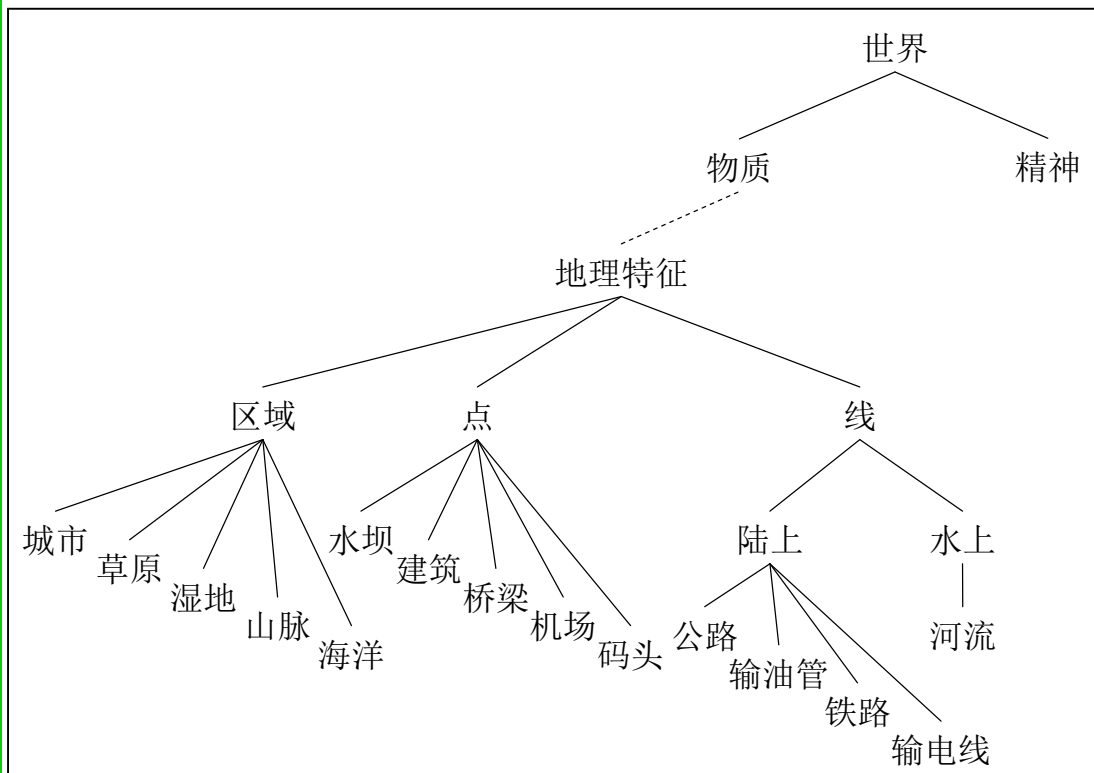
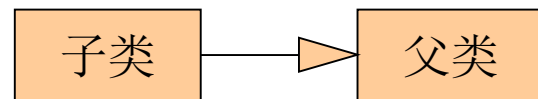
# 接口 (Interface)

- 接口(Interface): 描述了一个类的一组外部可用的属性和服务(操作)集。
- 注意: 接口定义的是一组属性和操作的描述, 而不是操作的实现, 体现了“接口与实现的分离”的思想, 即“信息隐藏”。
- 如果一个类“实现”了接口, 意味着该类需要包含接口中定义的所有属性和操作, 并对其进行具体的实现;
- 表示方式: “棒棒糖”



# 继承(Inheritance)

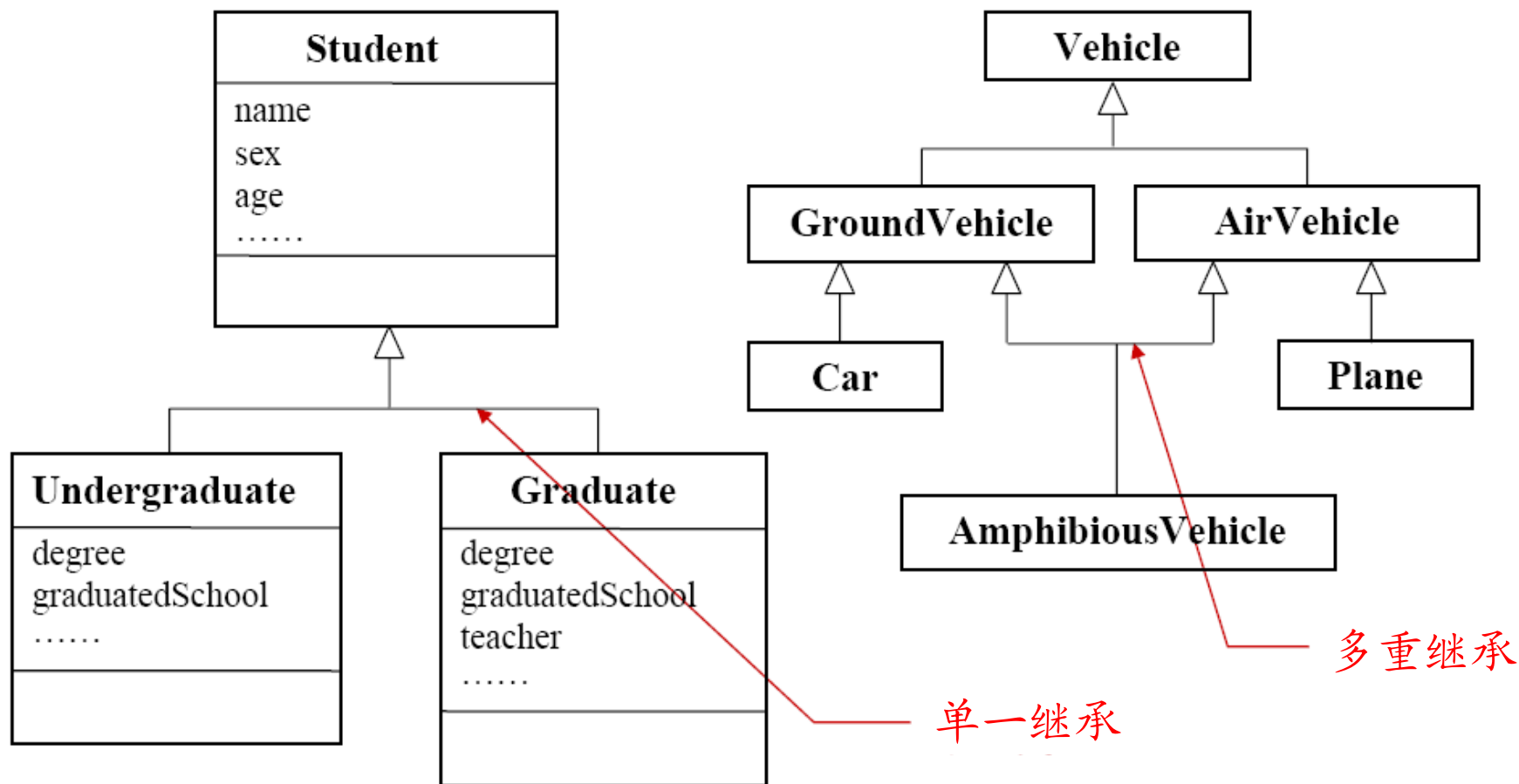
- **继承(Inheritance):** 子类可自动拥有父类的全部属性和操作。



用例之间的继承关系

继承关系使类之间  
形成层次化结构

# 继承(Inheritance)

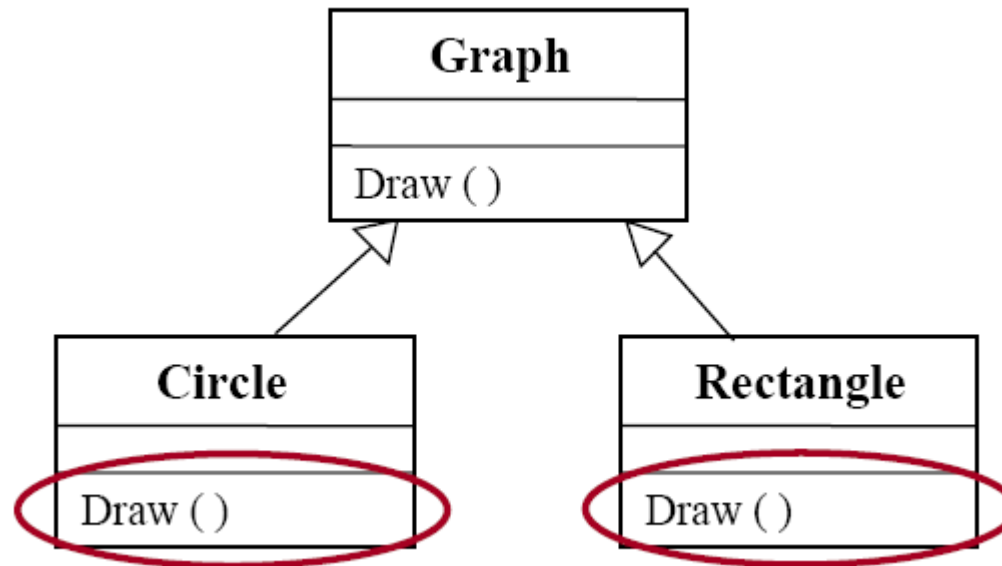


# 继承(Inheritance)

- **单一继承**：一个子类只有唯一的一个父类
- **多重继承**：一个子类有一个以上的父类
- **抽象类**：把一些类组织起来，提供一些公共的行为，但不能使用这个类的实例(即从该类中派生出具体的对象)，而仅仅使用其子类的实例。称不能建立实例的类为抽象类。
  - 抽象类中至少有一个方法被定义为“abstract”类型的。
  - “abstract”类型的方法：只有方法定义，没有方法的具体实现。

# 多态(Polymorphism)

- **多态性(Polymorphism):** 在父类中定义的属性或服务被子类继承后, 可以具有不同的数据类型或表现出不同的行为。



# 多态(Polymorphism)

- **多态性**：同一个操作作用于不同的对象上可以有不同的解释，并产生不同的执行结果。
- 多态性是面向对象程序设计语言的基本机制，是将一个方法名与不同方法关联的能力
- **静态绑定**：传统程序设计语言的过程调用与目标代码的连接(即调用哪个过程)放在程序运行前进行
- **动态绑定**：把这种连接推迟到运行时才进行



### 3 对象之间的五类关系





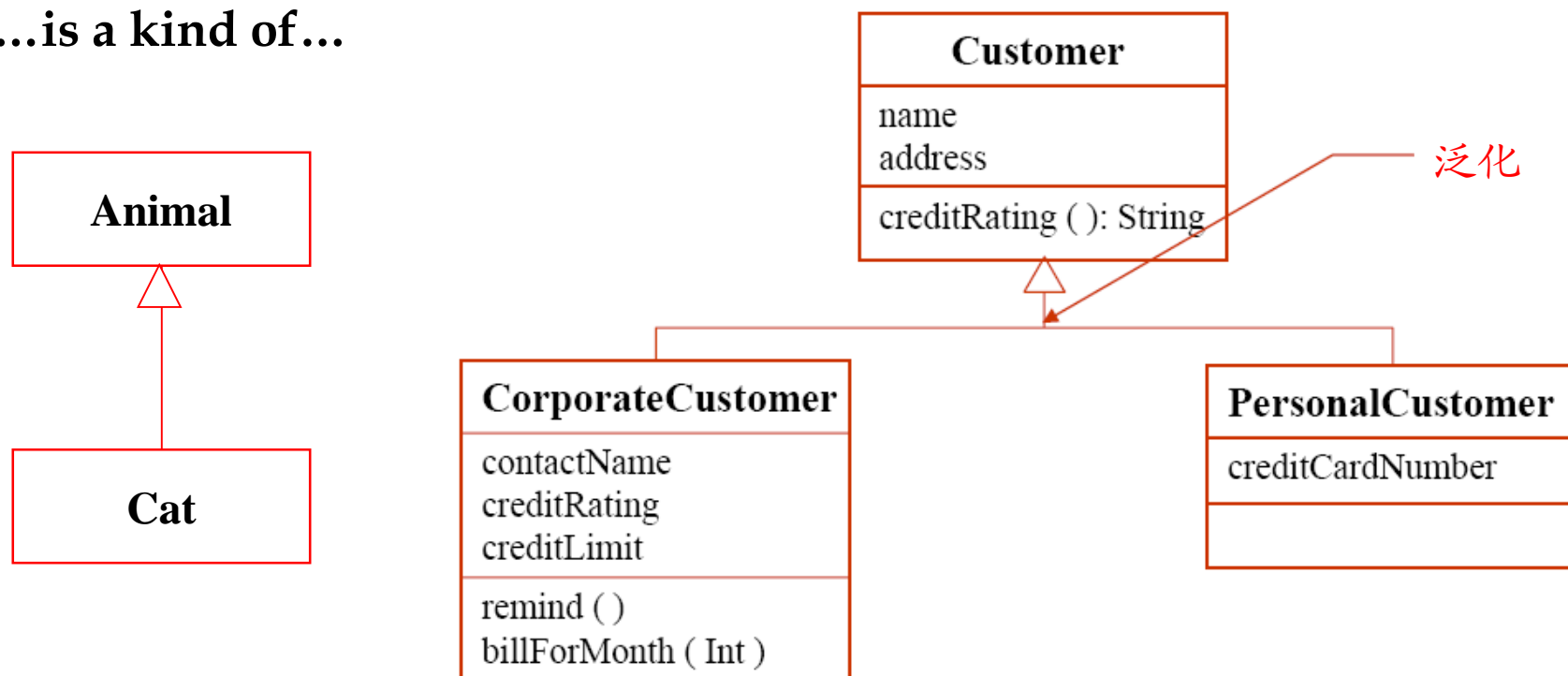
# 对象之间的联系

## ■ 对象之间的联系

- 分类结构：一般与特殊的关系
- 组成结构：部分与整体的关系
- 实例连接：对象之间的静态联系
- 消息连接：对象之间的动态通信联系

# (1) 分类结构：继承/泛化关系

- 分类结构：表示的是事物的“一般-特殊”的关系，也称为泛化 (Generalization) 联系。
- ...is a kind of...

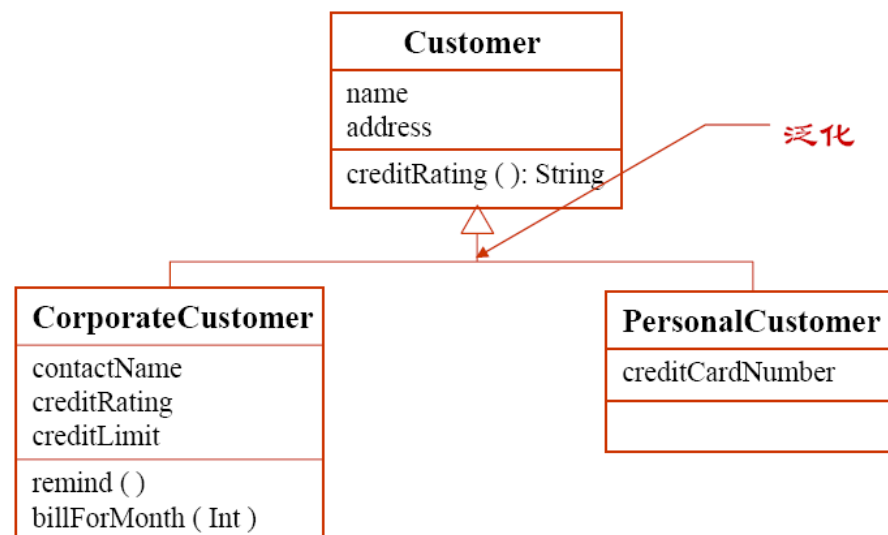


A cat is a kind of animal;  
A corporate customer is a kind of customer;  
A personal customer is a kind of customer, too;

# (1) 分类结构：继承/泛化关系

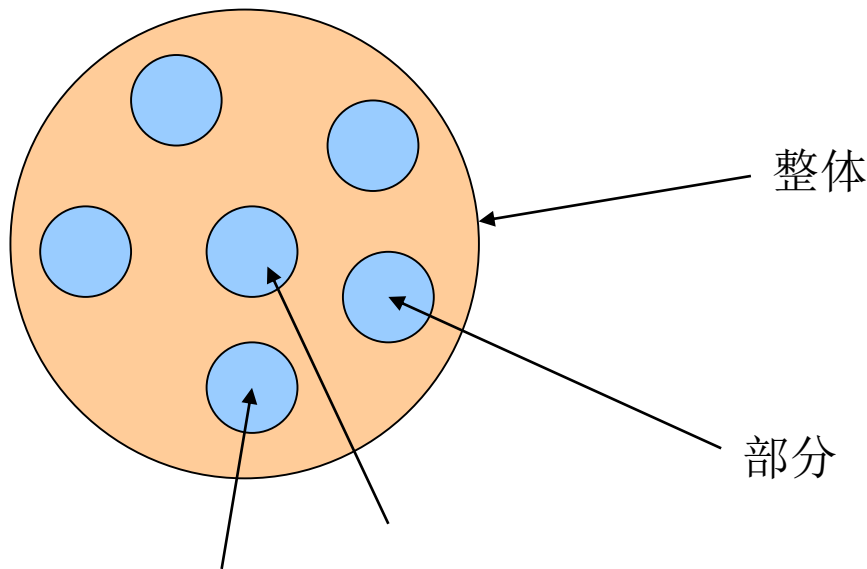
```
class Customer {  
    String name;  
    String address;  
    String creditRating() { };  
}
```

```
class CorporateCustomer : Customer {  
    String contactName;  
    String creditRating;  
    String creditLimit;  
  
    void Remind() { };  
    void billForMonth(int bill) { };  
}
```



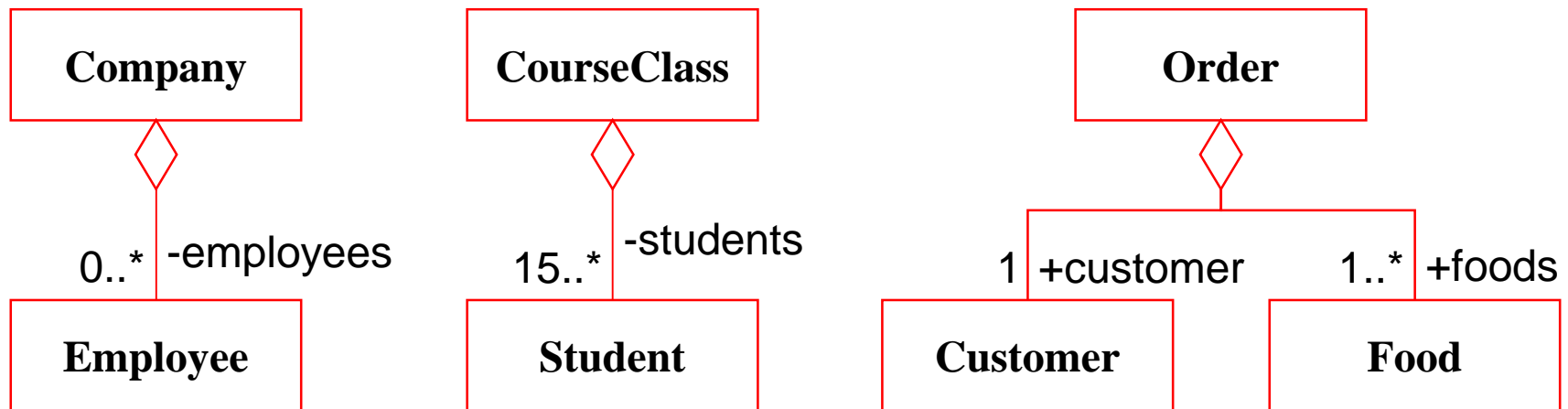
## (2) 组成结构：聚合与组合关系

- 组成结构：表示对象类之间的组成关系，一个对象是另一个对象的一部分，即“部分-整体”关系。
- 分为两个子类：
  - 聚合(Aggregation): 整体与部分在生命周期上是独立的 (...owns a...);
  - 组合(Composition): 整体与部分具有同样的生命周期(...is part of...);



## (2) 组成结构：聚合关系

- 聚合(Aggregation): 整体与部分在生命周期上是独立的



**A company owns zero or multiple employees;  
A course's class owns above 15 students;  
An order owns a customer and a set of foods;**

## (2) 组成结构：聚合关系

定义两个类：

```
class Student {}
```

```
class CourseClass {
```

```
...
```

```
private Student[] students;
```

```
public addStudent (Student s) {
```

```
    students.append(s);
```

```
}
```

```
...
```

```
}
```

使用时的代码：

```
Student a = new Student ();
```

```
Student b = new Student ();
```

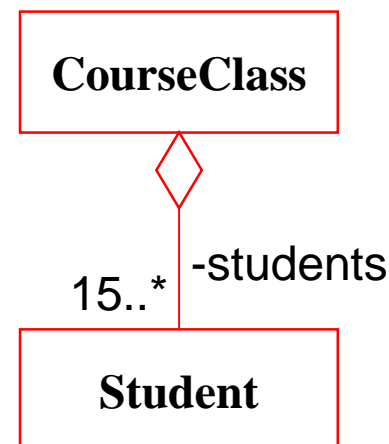
```
Student n = new Student ();
```

```
CourseClass SE = new CourseClass();
```

```
SE.addStudent (a);
```

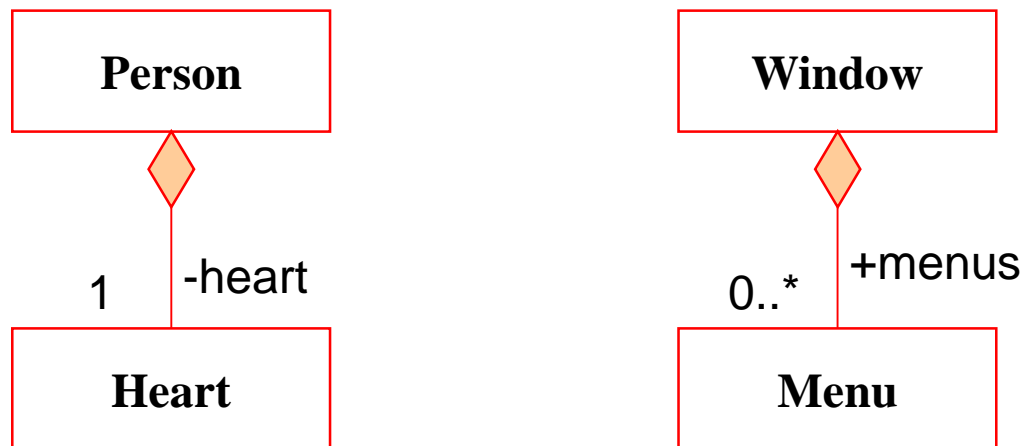
```
SE.addStudent (b);
```

```
SE.addStudent (n);
```



## (2) 组成结构：组合关系

- 组合(Composition): 整体与部分具有同样的生命周期;



**A heart is part of a person;  
A menu is part of a window;**

## (2) 组成结构：组合关系

```
class Heart {}
```

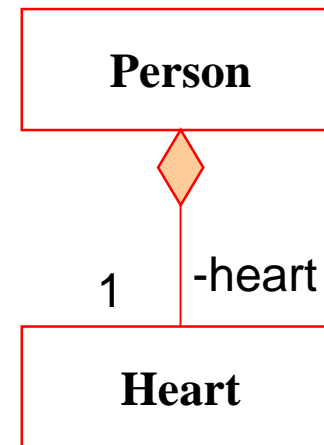
```
class Person {
```

```
...
```

```
private Heart heart = new Heart();
```

```
...
```

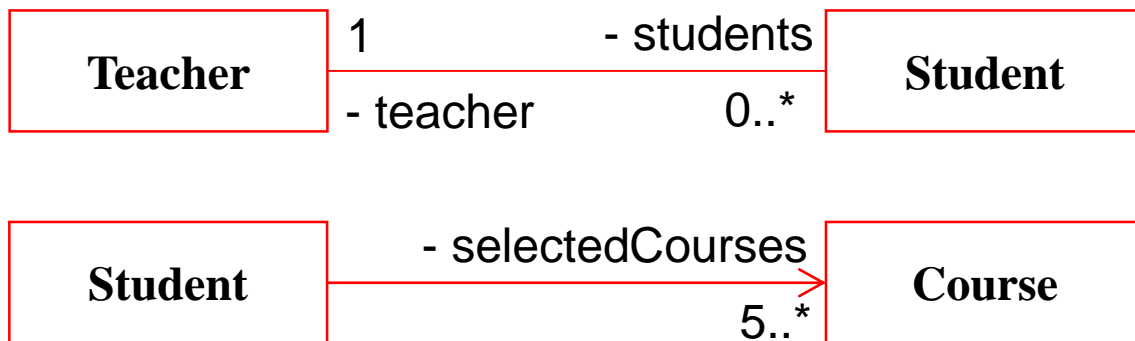
```
}
```





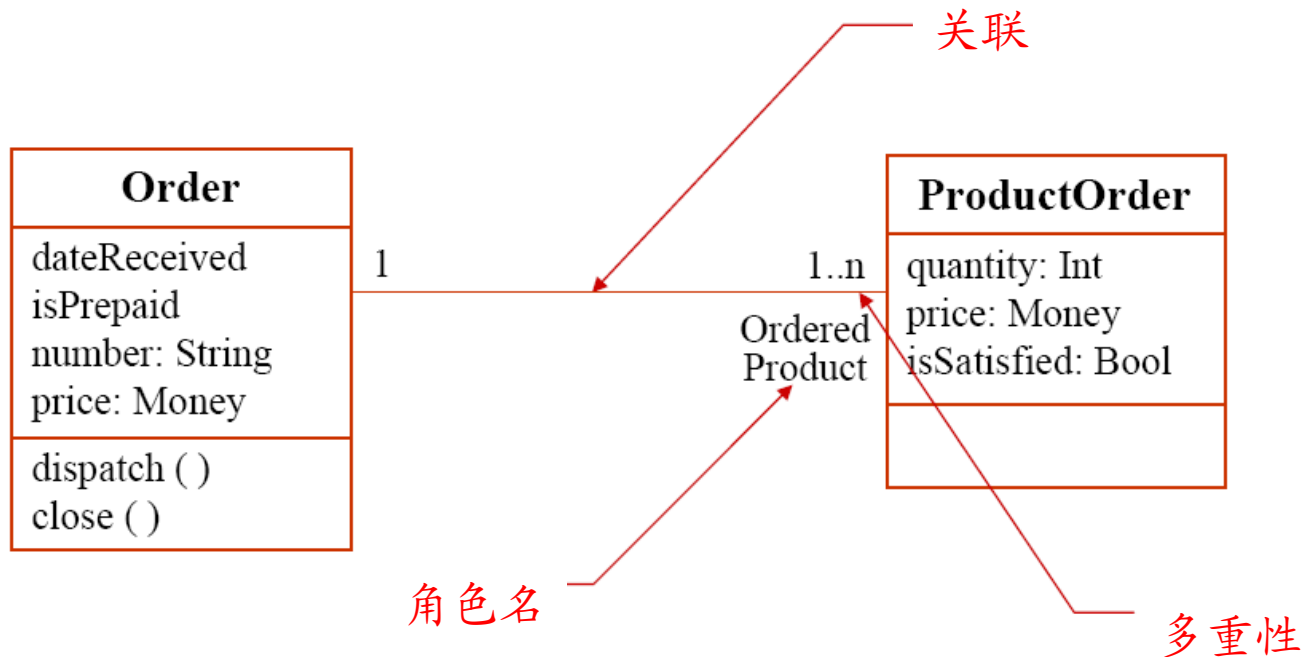
### (3) 实例连接：关联关系

- 实例连接：表示对象之间的静态联系，通过对象的属性之间的联系加以展现。
  - 对象之间的实例连接称为链接(Link)，存在实例连接的对象类之间的联系称为关联(Association)。
  - ... has a ...
- 例如：
  - “教师”与“学生”是两个类，它们之间存在“教-学”关系。
  - “学生”与“课程”是两个类，它们之间存在“学习-被学习”的关系。



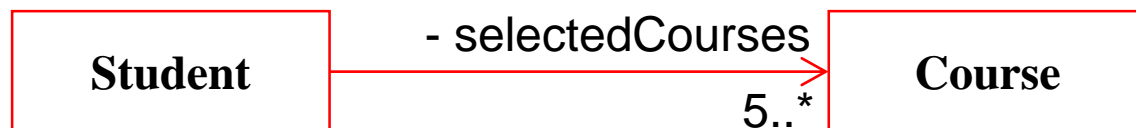
### (3) 实例连接：关联关系

- 关联具有多重性：表示可以有多少个对象参与该关联
- 关联具有方向性：
  - 单向关联：两个类是相关的，但是只有一个类知道这种联系的存在
  - 双向关联：两个类彼此知道它们间的联系



### (3) 实例连接：关联关系

```
class Course {}
class Student {
    private Course [] selectedCourses;
}
```

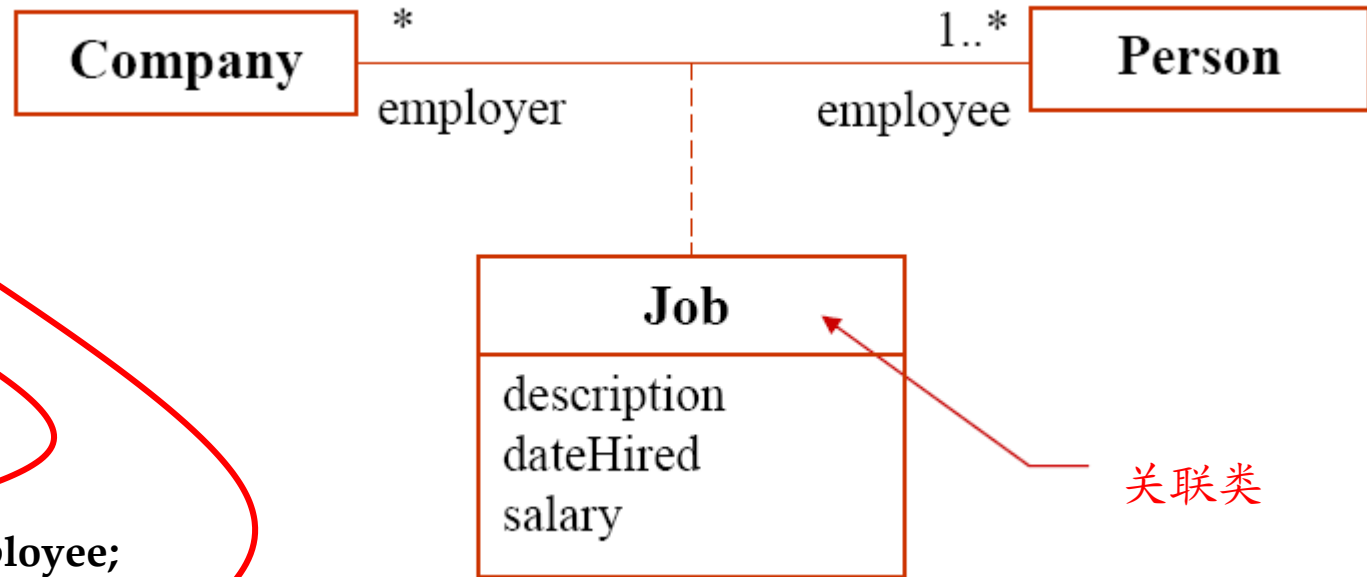


```
class Teacher {
    private Student [] students;
}
class Student {
    private Teacher teacher;
}
```



### (3) 实例连接：关联关系

#### ■ 关联类：



```
class Company {}
```

```
class Person {}
```

```
class Job {
```

```
    public Person employee;
```

```
    public Company employer;
```

```
    String description;
```

```
    Date dateHired;
```

```
    double salary;
```

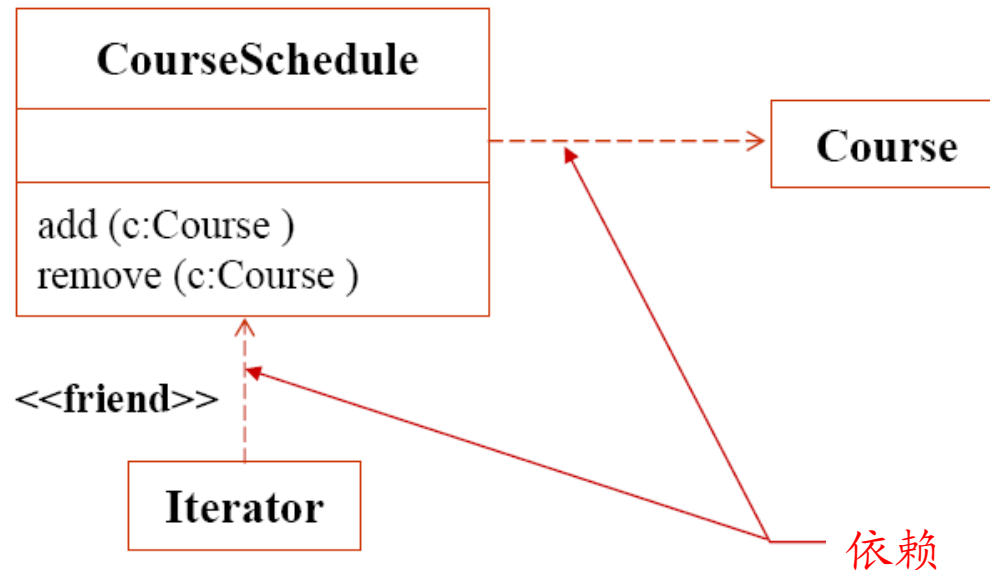
```
}
```

## (4) 消息连接：依赖关系

### ■ 消息连接

- 消息连接是对象之间的通信联系，它表现了对对象行为的动态联系。
- 一个对象需要另一个对象的服务，便向它发出请求服务的消息，接收消息的对象响应消息，触发所要求的服务操作。

### ■ 消息连接也称为“依赖关系”(Dependency)。



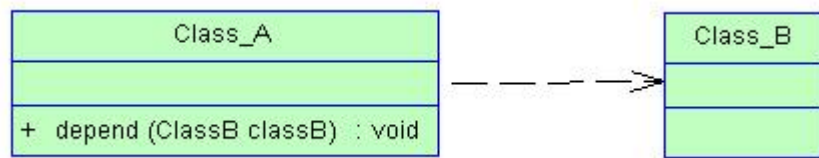
## (4) 消息连接：依赖关系

### ■ 依赖(Dependency): ...use a...

- 依赖是一种使用关系，一个类A使用到了另一个类B，而这种使用关系是偶然性的、临时性的、非常弱的，但是B类的变化会影响到A。

### ■ 类的依赖可能由各种原因引起，例如：

- 一个类是另一个类的某个操作的参数
- 一个类在另一个类的某个操作中被使用



```
class Air {}
```

```
class Human {
    public void breath(Air air) {};
}
```

## “依赖”关系怎么识别？

- 依赖(Dependency)：最弱的一种类间关系，临时的、局部的。
- 除非类A的某个操作op使用了类B，否则不要随意设置依赖关系：
  - op有某个参数param或返回值的类型为B；
  - op的内部业务逻辑中使用了B；
- 判断标准：作用域范围
  - B是否只在A中的某个操作的作用域范围内才被A所使用？若是，则A依赖于B；
  - 若A的某个属性的数据类型是B，那意味着B对A的全部操作都是有意义的，那么A和B之间至少是关联关系。

# 课堂讨论

## ■ 淘宝案例中的类：

- 买家、卖家
- 商品
- 宝贝
- 订单
- 购物车

## ■ 它们之间的关系？

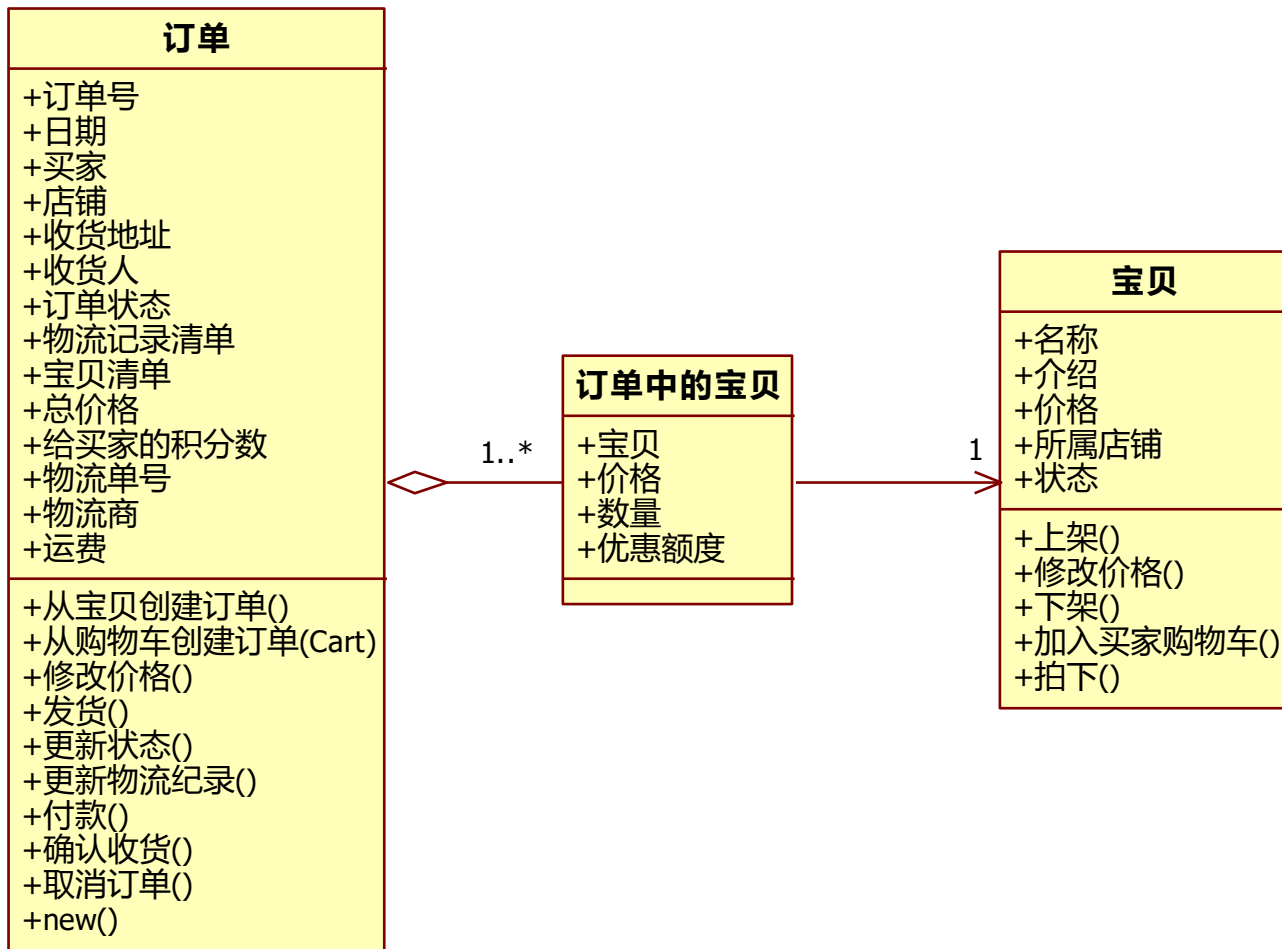
- “宝贝”对象和“商品”对象
- “宝贝”和“买家”对象
- “宝贝”和“卖家”对象
- “购物车”对象和“宝贝”对象
- “订单”和“购物车”对象
- “订单”对象和“宝贝”对象
- “订单”和“买家”对象
- “订单”和“卖家”对象



## 如何识别出“关联类”？

- 关联类：当两个实体类之间产生m:n关联关系，且这种关联关系导致了新的信息产生并需要对其进行管理时，就需要关联类。
- 这里所谓的“新的信息”是指：如果两个实体类不关联在一起，这些信息就不会存在。
- 例如：订单和宝贝这两个实体类，按照常规理解，似乎是聚合关系，一个订单包含多个宝贝。但是在这种聚合发生时，产生了某些新信息(该宝贝在该订单中的价格，可能与卖家给宝贝设定的价格不同；本次购买的数量)。这些新信息，既不属于订单，也不属于宝贝本身，故而拆分出一个关联类“订单项”：
  - 订单：订单号、买家、卖家、总价格、收货地址、物流流转记录(list)、订单项集合(list)、etc；
  - 宝贝：名字、所属卖家、设定价格、etc；
  - 订单项：宝贝、本次购买价格、数量。

# 如何识别出“关联类”？



# 关于“聚合”与“关联”的区别

- 本质上，聚合都是特殊的“关联”关系，只不过关系的强度更大。若两个类之间是聚合关系，其实是可以关联关系来表示的。
- 例如：
  - 对“购物车”和“宝贝”两个类而言，可以说“多个宝贝对象聚合成了购物车对象”，宝贝是购物车的一部分；
  - 也可以说“购物车 has some 宝贝”，二者是多对多的关联关系，一个购物车里有0..\*个宝贝，一个宝贝可以在0..\*个购物车内出现，宝贝对象无需维护自己出现在了哪些购物车对象中，但购物车对象一定要知道自己内部有哪些宝贝对象，故这是一个从购物车类指向宝贝的单向关联。
- 何时用聚合，何时用关联？
  - 一个经验：判断两个类的“地位”是否对等。
  - 若二者对等，用关联关系（例如宝贝和卖家，二者非常独立存在的，彼此地位相同，更适合用关联而不是聚合）；
  - 若二者明显不对等，用聚合关系（例如“买家对宝贝的评价”和“宝贝”，宝贝的地位更高，评价往往看作它的一部分）。

## “依赖”关系怎么识别？

### ■ 例如：

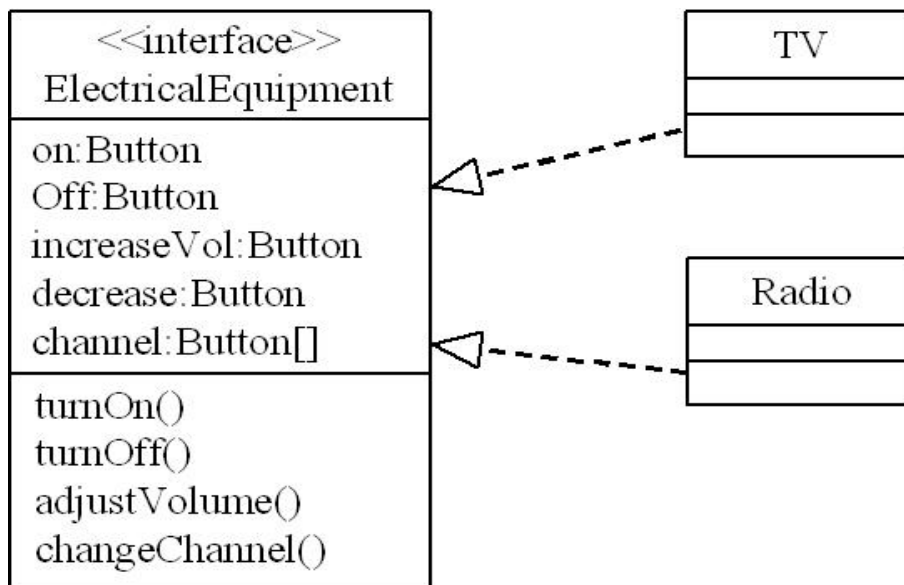
- “订单”类中有一个操作“从购物车生成订单()”，它有一个参数，类型是“购物车”类，该操作在执行时，根据传入的“购物车”对象读取产品信息来生成多个“订单项”对象。这个操作执行完之后，“订单”类和“购物车”类就再无关系，故前者依赖于后者。

### ■ 一个好理解的例子：

- 一个“路人”类和一个“时钟”类，后者的所有属性和操作均与前者无关，而前者在走到时钟面前时抬头看了看时钟以获取时间，故“路人”类有一个操作“抬头看时间”。只有在这个操作范围内，路人需要与时钟发生关系，而其他操作如“走路”、“吃饭”等均与“时钟”无关。故“路人”依赖于“时钟”。

## (5) 接口连接：实现关系

- 实现关系(realization): 是泛化关系和依赖关系的结合，通常用以描述一个接口和实现它们的类之间的关系；
- 是“棒棒糖”的另一种形式。



## 小结：对象之间的联系

- 继承/泛化：一般与特殊的关系——is a kind of
  - 组合：部分与整体的关系，彼此不可分——is part of
  - 聚合：部分与整体的关系，但彼此可分——owns a
  - 关联：对象之间的长期静态联系——has a
  - 依赖：对象之间的动态的、临时的通信联系——use a
- 类间联系的强度：继承 >>> 组合 >> 聚合 >> 关联 >>> 依赖



結束

2017年10月31日