



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程

软件工程

第六章 OO分析与设计

6-1 面向对象的基本概念

徐汉川

xhc@hit.edu.cn

2017年10月25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
4. UML建模语言简介



1. 面向对象技术概述



软件工程方法

■ 结构化

- 复杂世界→复杂处理过程（事情的发生发展）
- 设计一系列功能（或算法）以解决某一问题
- 寻找适当的方法存储数据

■ 面向对象

- 任何系统都是由能够完成一组相关任务的对象构成
- 如果对象依赖于一个不属于它负责的任务，那么就需要访问负责此任务的另一个对象（调用其他对象的方法）
- 一个对象不能直接操作另一个对象内部的数据，它也不能使其它对象直接访问自己的数据
- 所有的交流都必须通过方法调用

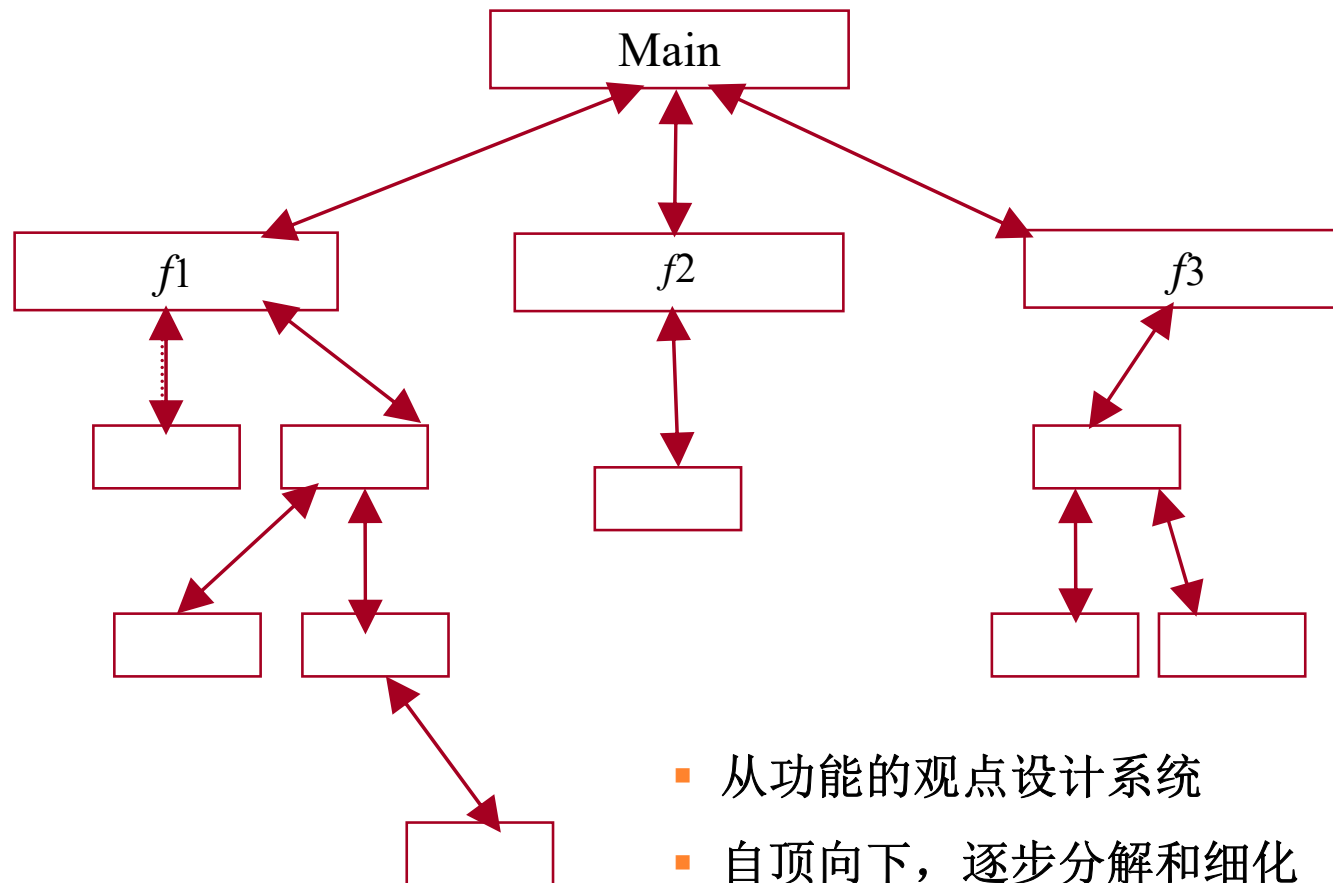
举例：五子棋游戏

- 面向过程（事件）的设计思路就是首先分析问题的步骤：
 1. 开始游戏，初始化画面
 2. 黑子走，绘制画面，
 3. 判断输赢，如分出输赢，跳至步骤6
 4. 白子走，绘制画面，
 5. 判断输赢，如未分出输赢，返回步骤2，
 6. 输出最后结果。
- 面向对象的设计思路是分析与问题有关的实体：
 1. 玩家：黑白双方，这两方的行为是一模一样的，
 2. 棋盘：负责绘制画面
 3. 规则：负责判定诸如犯规、输赢等。

先看结构化方法

- 使用结构化编程、结构化分析和结构化设计技术的系统开发方法
 - 结构化编程
 - 结构化设计
 - 结构化分析
- 程序=数据结构+算法
- 以过程（事件）为中心的设计方法

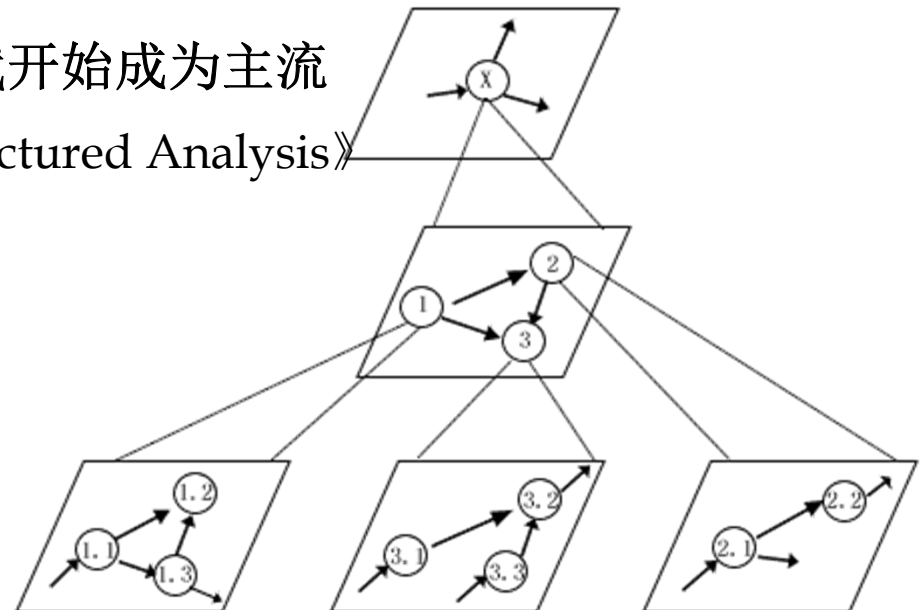
先看结构化程序开发...



- 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

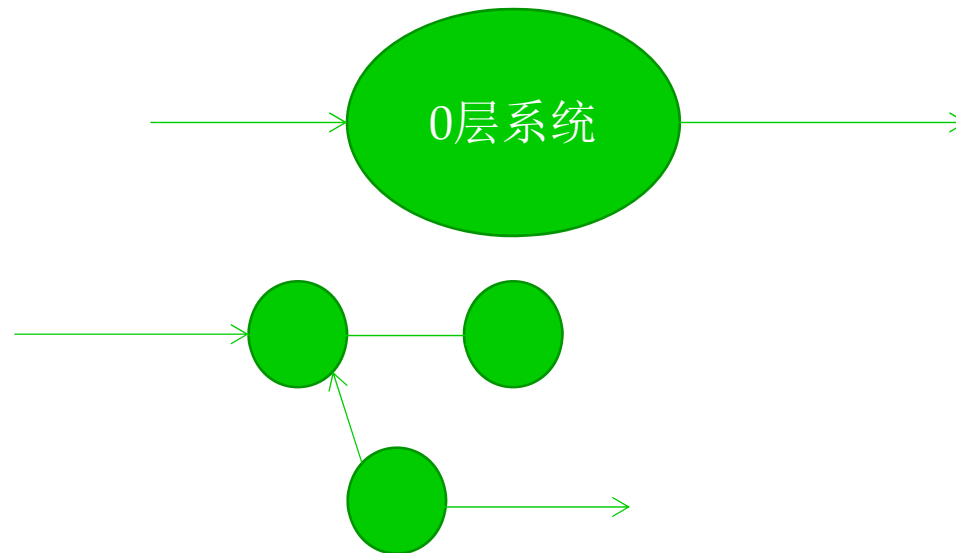
结构化分析方法

- **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)

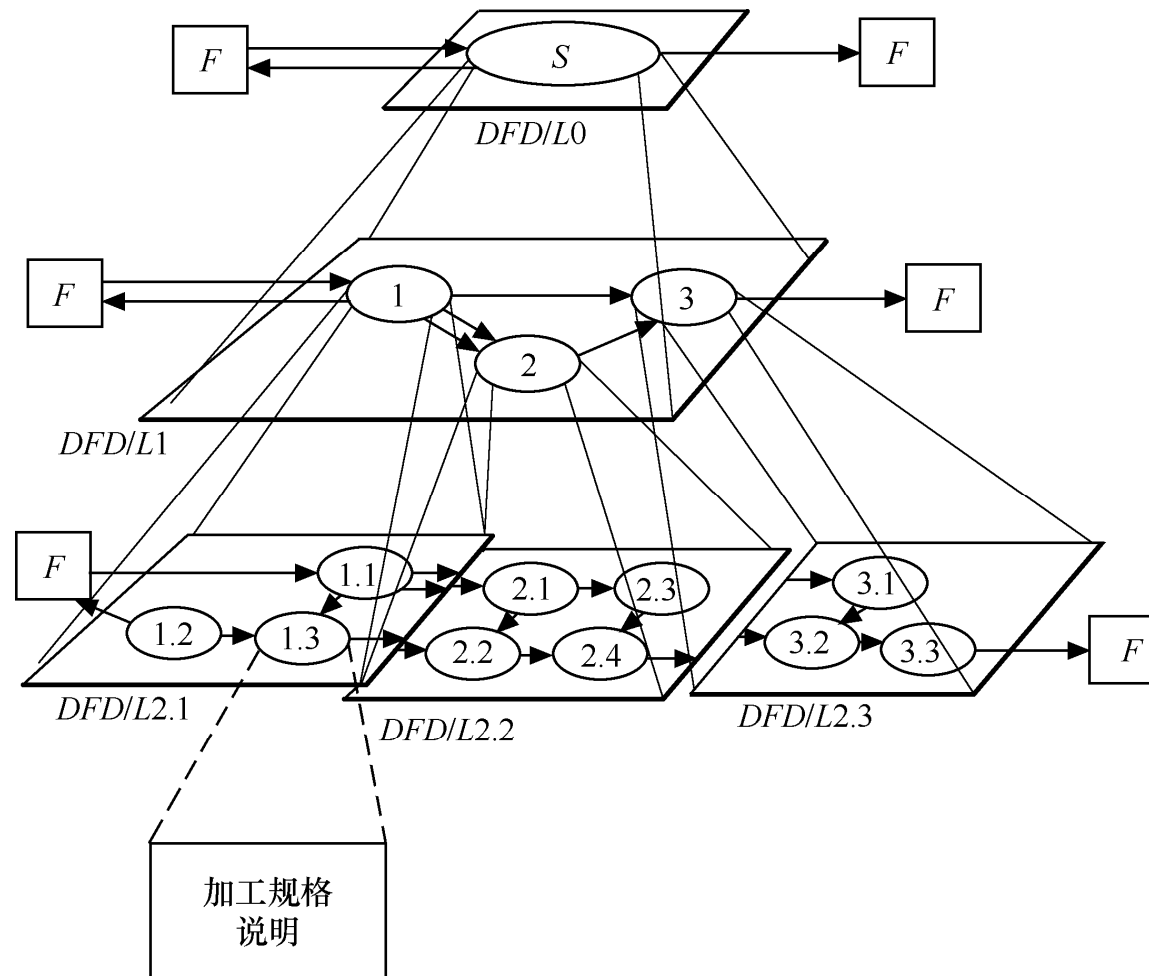


结构化分析

- 帮助开发人员定义系统需要做什么（处理需求），系统需要存储和使用哪些数据（数据需求），系统需要什么样的输入和输出以及如何把这些功能结合在一起完成任务。
 - 数据流图(DFD图)
 - 实体-联系图(ER,IDEF1X)



结构化分析方法



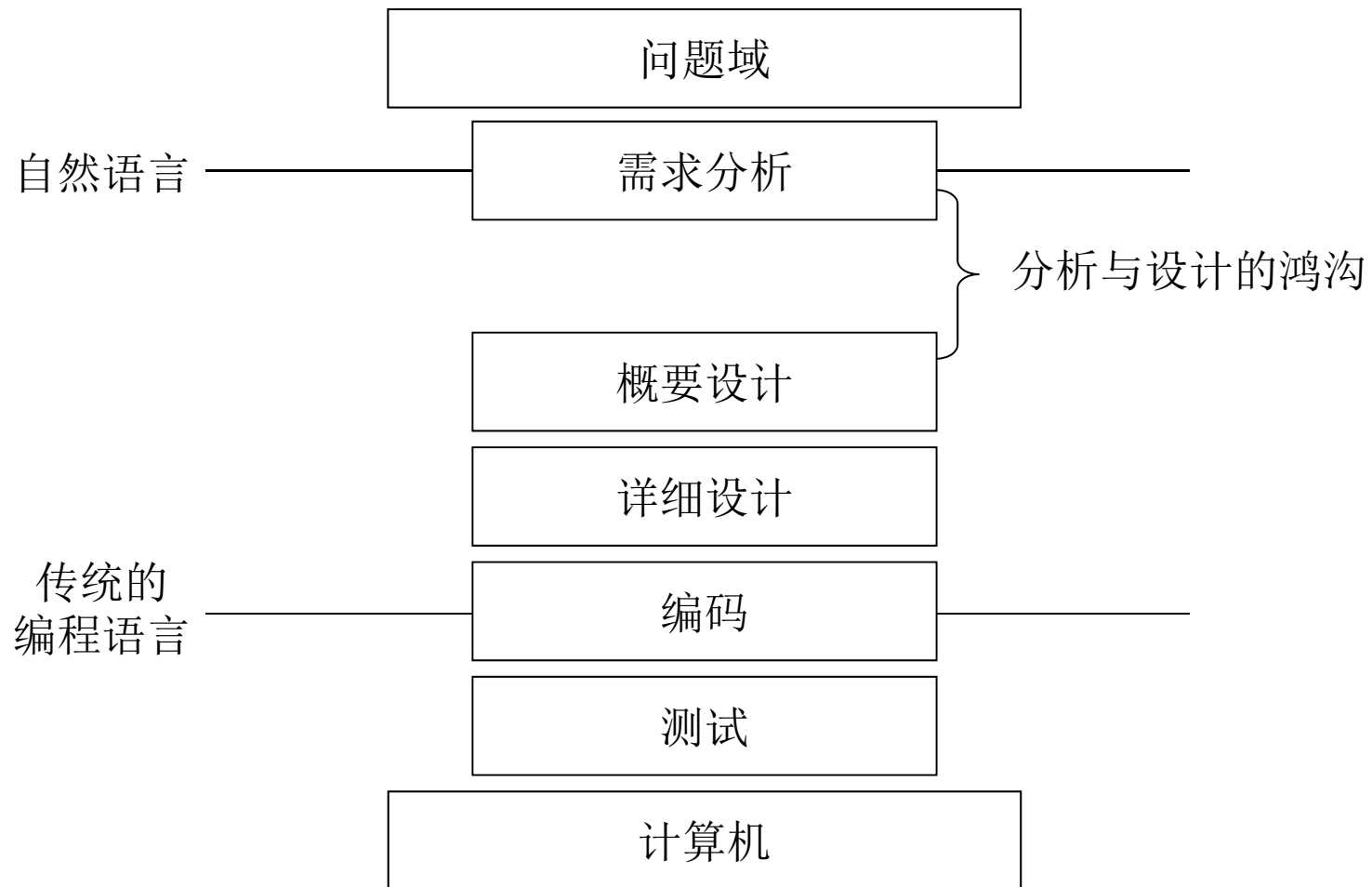
结构化程序开发的特点

- 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(DD)+数据流图(DFD)

结构化方法的常见问题

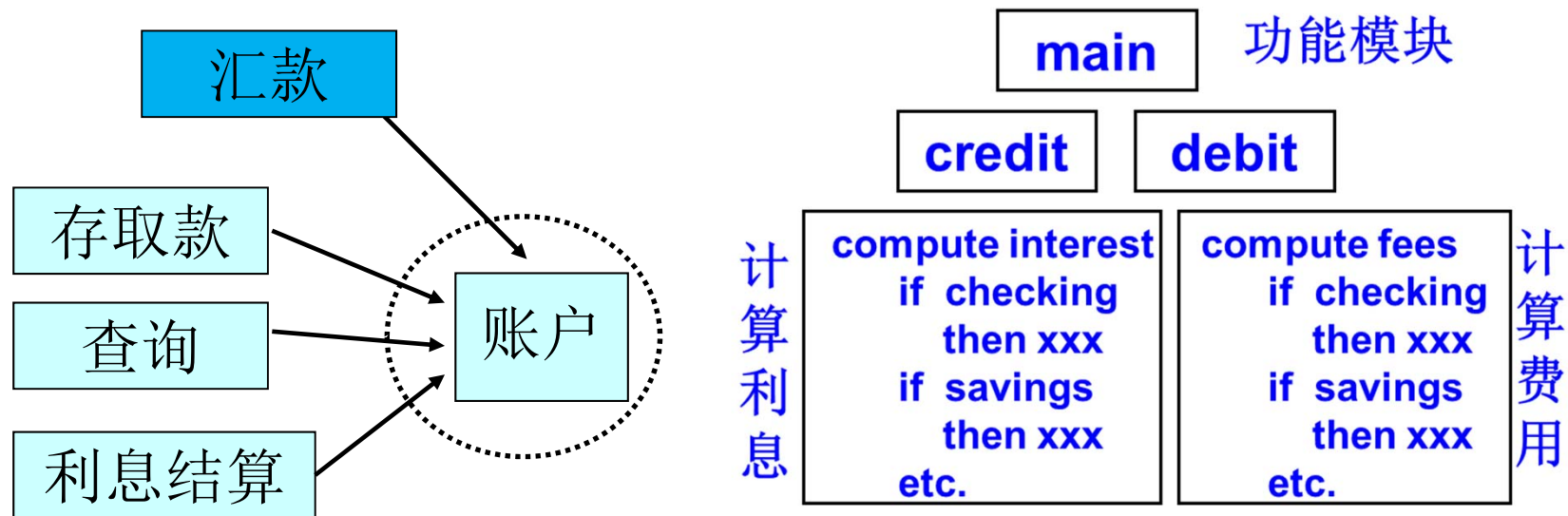
- 需求的错误
 - 不完整、不一致、不明确
 - 开发人员和用户无法以同样的方式说明需求
- 需求的变化
 - 需求在整个项目过程中始终发生变化
 - 设计后期发生改变
- 持续的变化
 - 系统功能不断变化
 - 许多变化出现在项目后期
 - 维护过程中发生许多变化
- 系统结构的崩溃
 - 系统在不断的变化中最终变得不可用

传统软件工程方法：结构化方法



造成上述问题的根本原因…

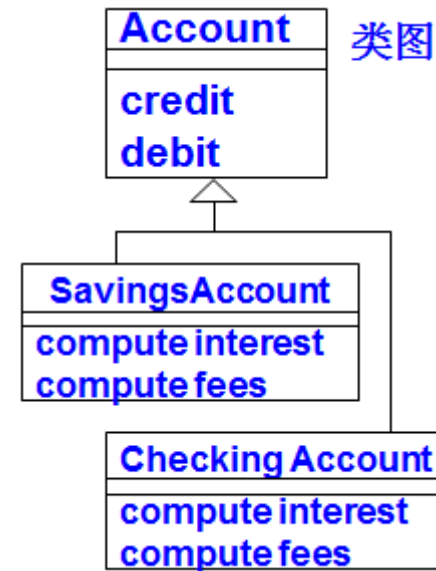
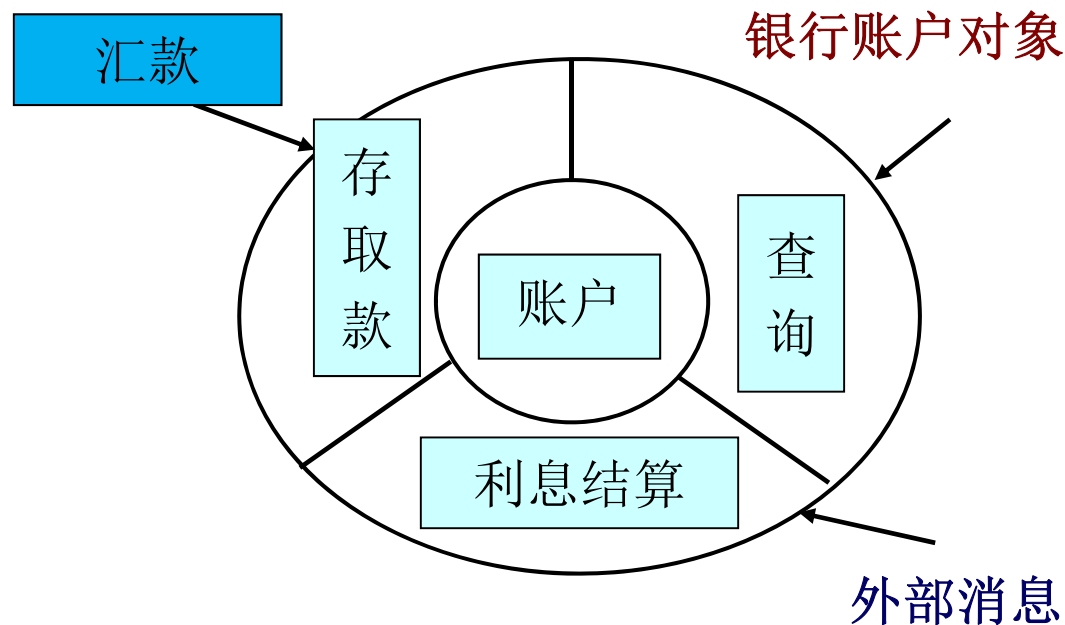
- **结构化方法以功能分解和数据流为核心**，但是…系统功能和数据表示极有可能发生变化；
 - 以ATM银行系统为例：帐户的可选项、利率的不同计算方式、**ATM**的不同界面；



数据和施加在其上的操作分离。
代码按过程组织的，每个过程操作不同类型的数据。

面向对象的程序开发...

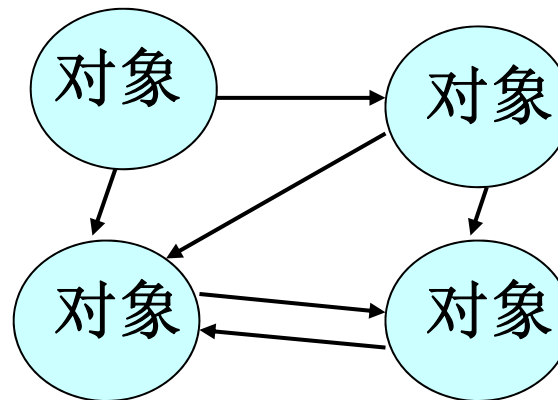
- 软件设计应尽可能去描述那些极少发生变化的稳定要素：对象
 - 银行客户、帐户



在面向对象中，是在数据抽象内来组织过程，即代码是按类组织的，每个类包含对该类实例进行操作的过程。

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)；
- 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；
 - 算法被分布到各种实体中；
 - 消息从一个对象传送到另一个对象；
 - 控制流包含在各个数据抽象中的操作内
 - 系统的状态保存在各个数据抽象所定义的数据存储中；

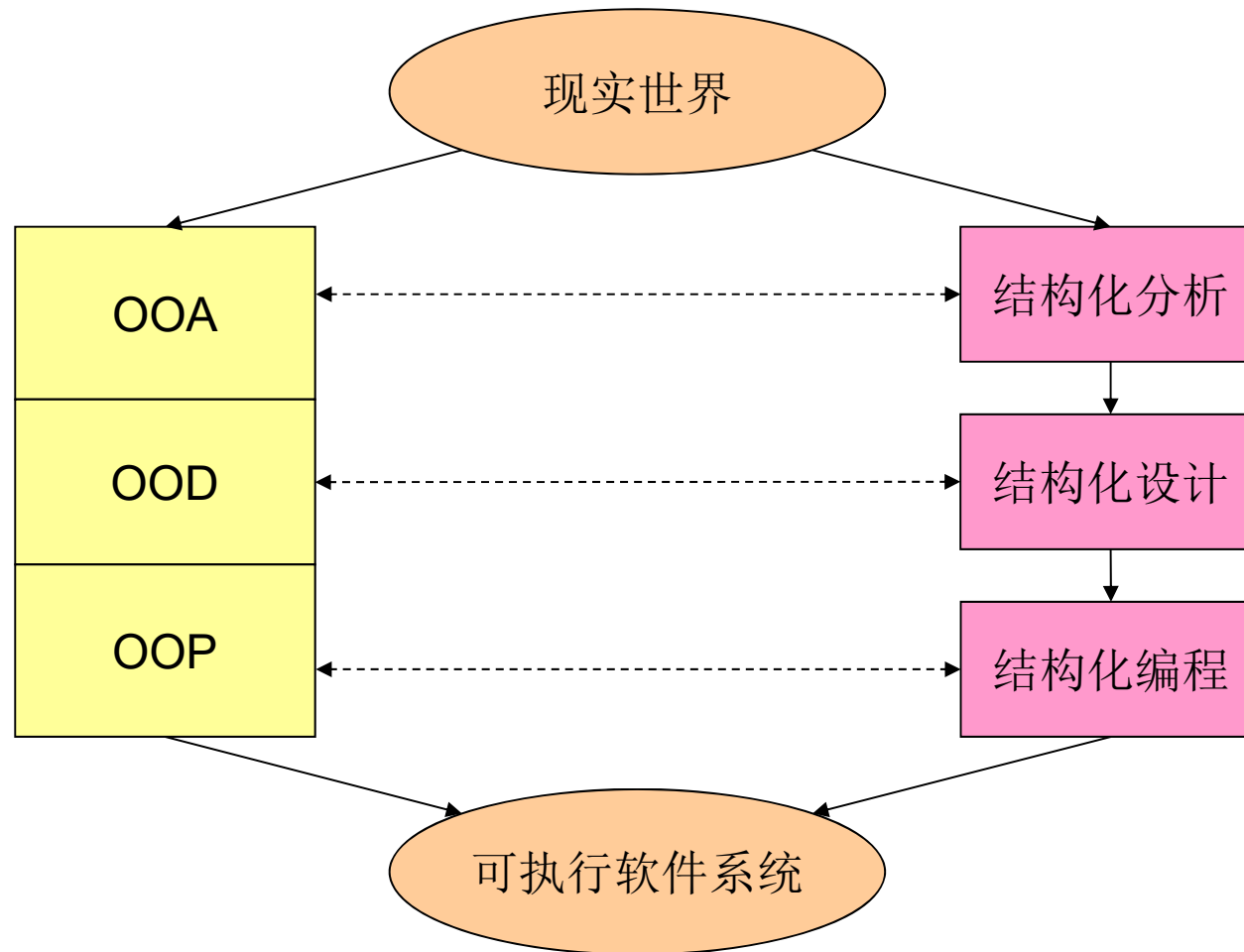
面向对象方法的优势

- 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
 - 售货员、商品、仓库、顾客、飞机、机场等
 - 逻辑概念
 - 商品目录、生产计划、销售
 - 操作系统中的分时策略、军事训练中的冲突解决规则等

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。

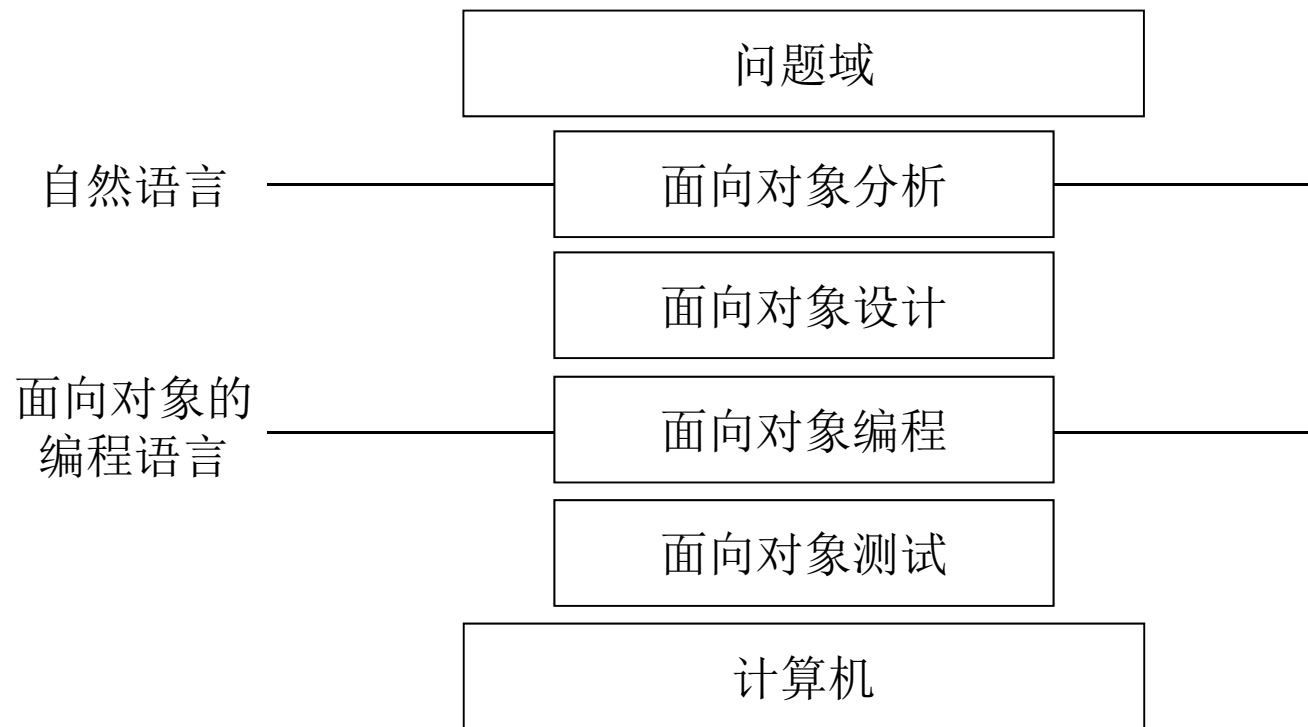
面向对象的软件工程



OOA/D

- 分析：强调的是对问题和需求的**调查研究**，而不是解决方案
 - 面向对象分析过程中，**强调的是在问题领域内发现和描述对象**
- 设计：强调的是满足需求的概念上的**解决方案**（在软件方面和硬件方面），而不是其实现。
 - 面向对象设计过程中，**强调的是定义软件对象以及它们如何协作以实现需求。**
- 有价值的分析和设计可以概括为：**做正确的事**（分析）和**正确地做事**（设计）

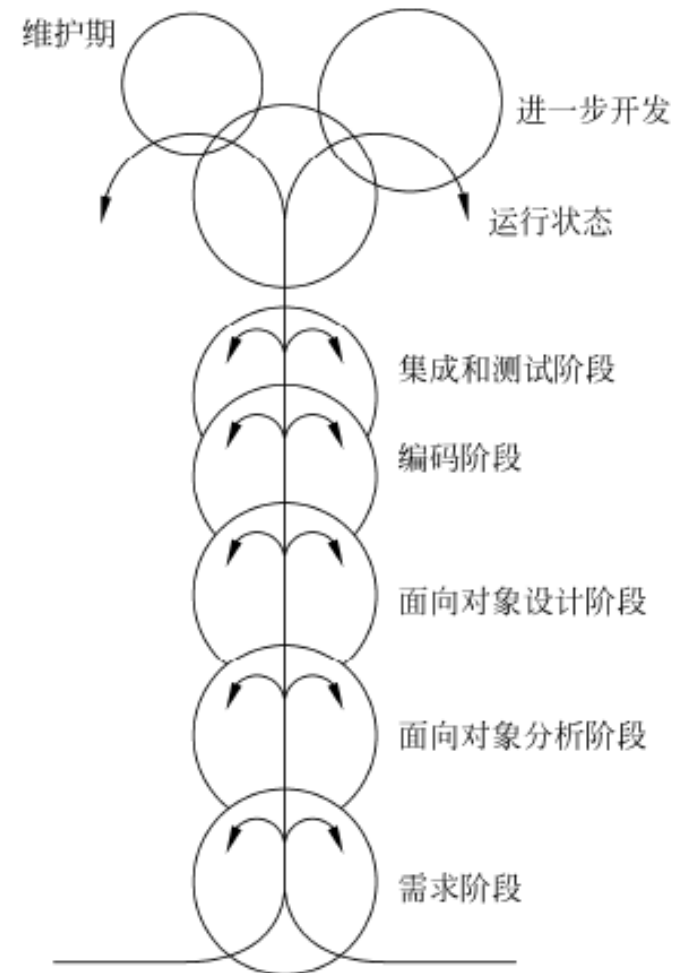
面向对象方法



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。





2. 面向对象的基本概念



面向对象中的基本概念

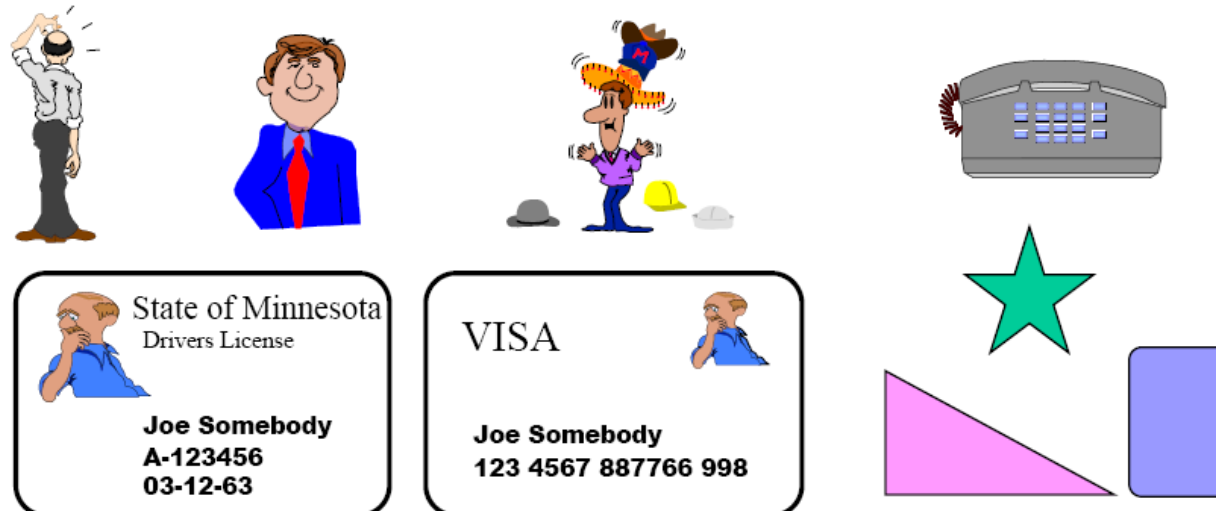
- Coad和Yourdon给出了面向对象的定义：

面向对象 = 对象 + 类 + 继承 + 消息

- 对象(Object)
- 类(Class)
- 消息(Message)
- 封装(Encapsulation)
- 泛化(Generalization)与继承(Inheritance)
- 多态(Polymorphism)
- 抽象类(abstract class)与接口(Interface)

对象(Object)

- **对象(Object): 具有责任的实体。**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。
- **Object::=(OID,DS,OS,MI)** (标识, 数据结构, 操作集, 消息集)
- **构成三要素: 标识符(区别其他对象)、属性(状态)和操作(行为)。**
 - **属性(Attribute): 与对象关联的数据, 描述对象静态特性;**
 - **操作(Operation): 与对象关联的程序, 描述对象动态特性;**



面向对象中的基本概念

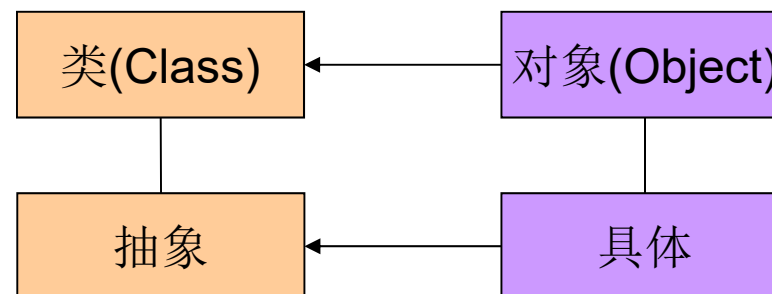
- Coad和Yourdon给出了面向对象的定义：

面向对象 = 对象 + 类 + 继承 + 消息

- 对象(Object)
- 类(Class)
- 消息(Message)
- 封装(Encapsulation)
- 泛化(Generalization)与继承(Inheritance)
- 多态(Polymorphism)
- 抽象类(abstract class)与接口(Interface)

类(Class)

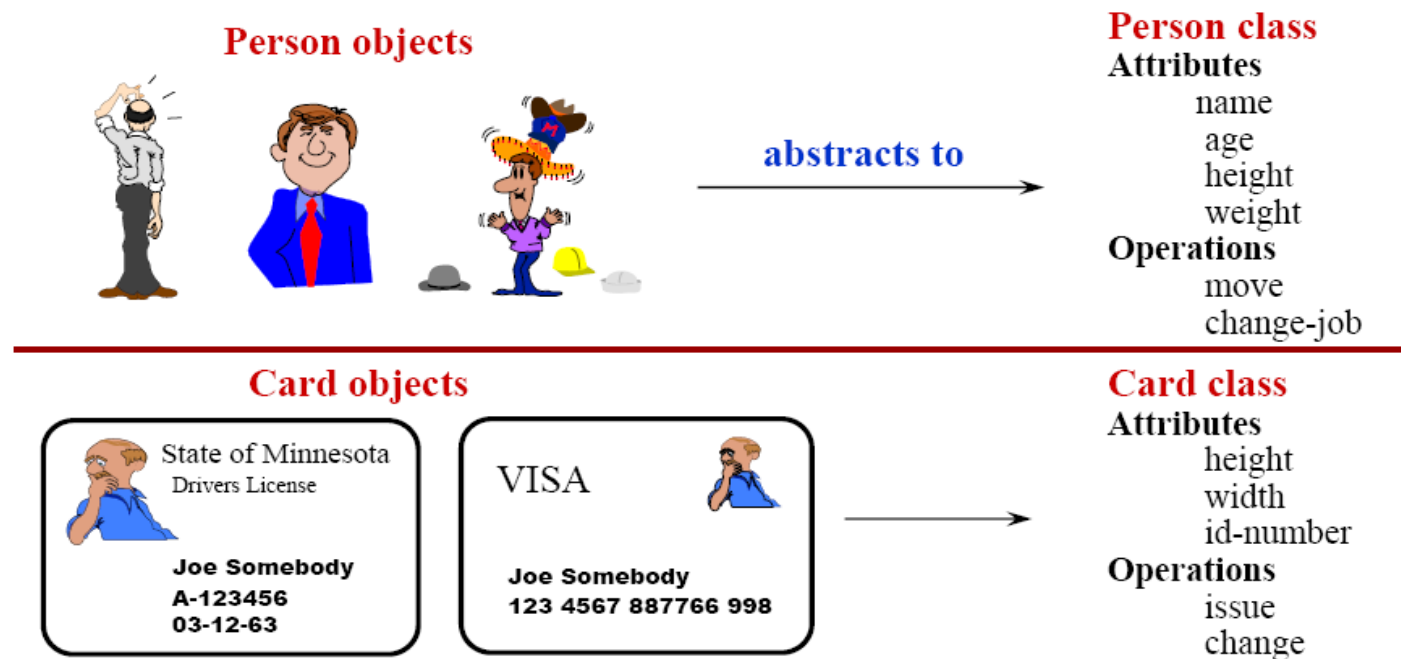
- **类(Class):** 具有相同属性和操作的一组对象的抽象，它为属于该类的全部对象提供了统一的抽象描述。
 - 类是概念定义，抽象了同类对象共同的属性和操作
 - 对象是类的一个实例
- **Class::=<ID,DS,OP,ITF,INH>**
(标识, 数据结构描述, 操作集合, 外部接口, 继承性描述)



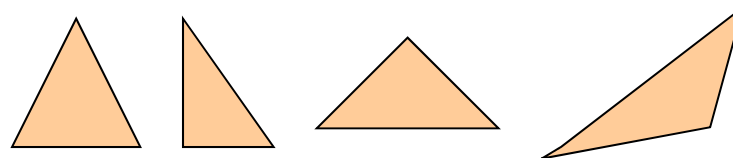
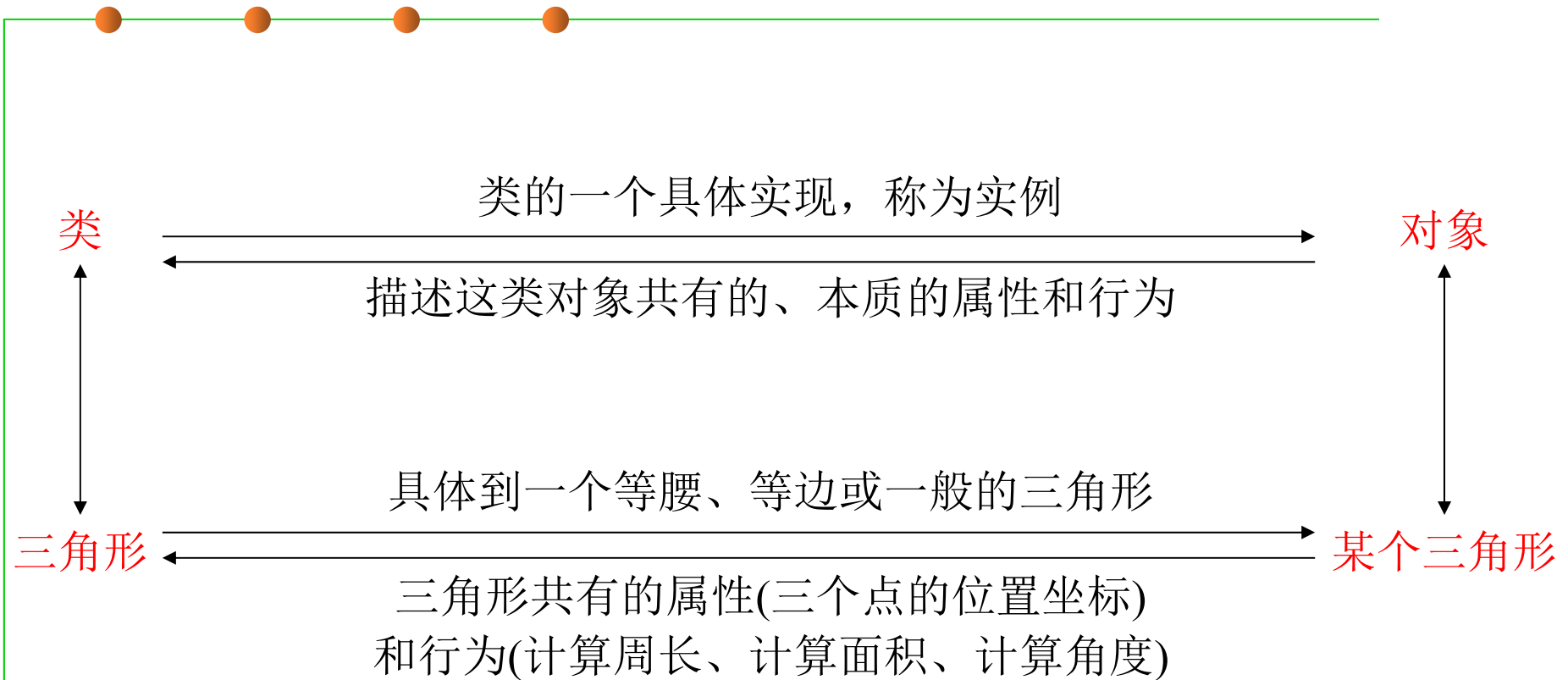
- “类”所代表的是一个抽象的概念或事物，现实世界中并不存在，而“对象”是客观存在的。

类 (Class)

- [例] “学生”是一个类，计算机学院1090310101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

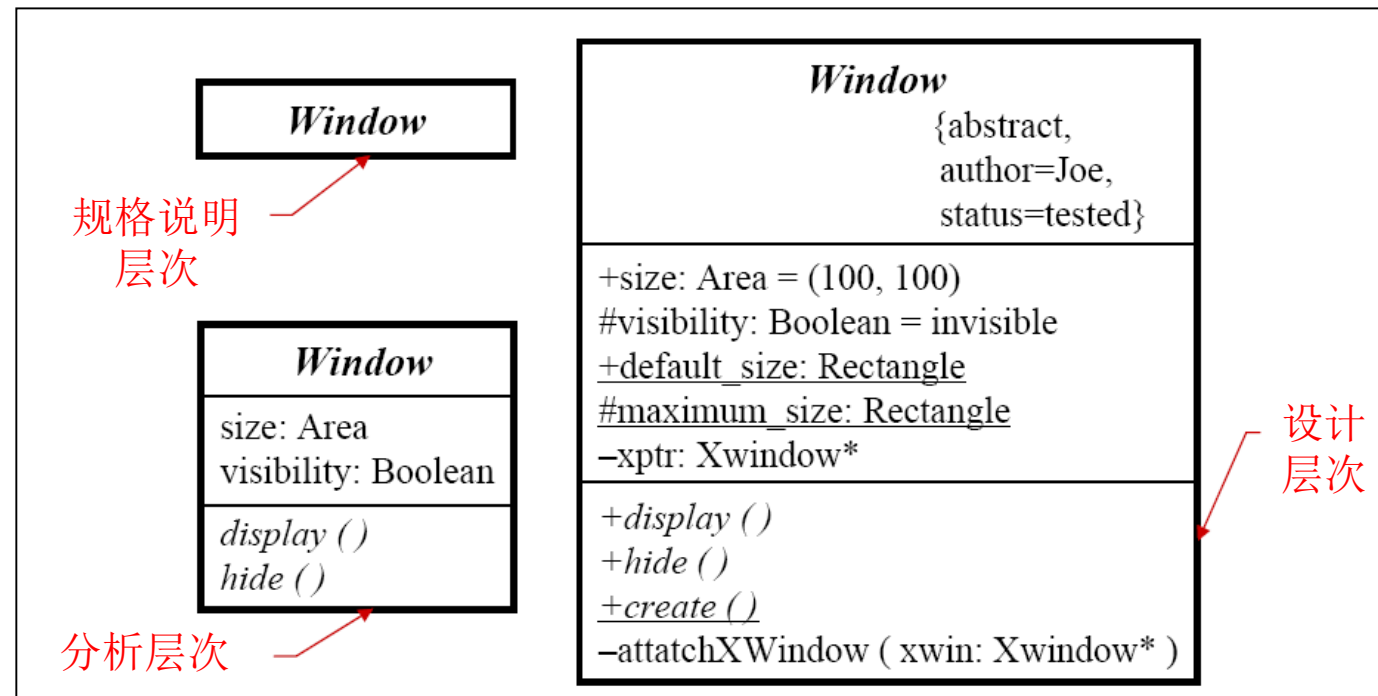
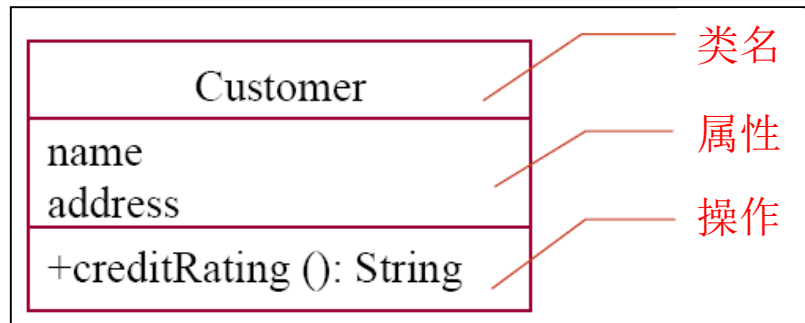
定义一个学生类:

```
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 ZSY= new Student ( "1080310501" , "张三" , "CS" );  
Student GY = new Student ( "1080310502" , "李四" , "CS" );  
Student WY = new Student ( "1080310503" , "王五" , "CS" );  
  
If ( ZSY.RegisterMyself() == true) {  
    ZSY.SelectCourses();  
}  
float score = ZSY.QueryScore (32);
```


类(Class)的三种抽象层次

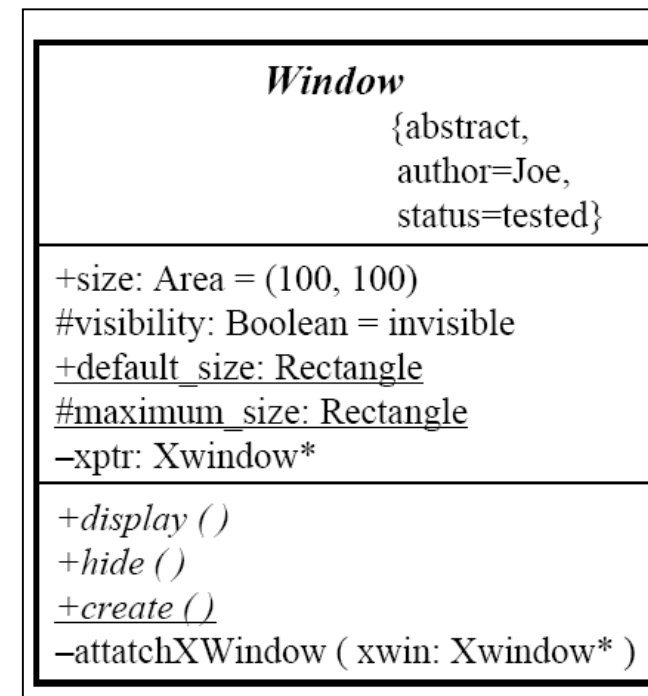


类的命名

- 尽量采用代表真实世界的**事物**去命名。
- 如类名：**Person, Employee(雇员), Hospital(医院), Doctor(医生).**
- 不好的例子：类名**PersonData, PartTimeEmployee(部分定期职员)**
- 采用单数形式命名，可以清楚地知道类中的实例是单独的项目，不是集合。

类的属性 (Attribute)

- 类的属性：描述对象“静态” (结构)特征的一个数据项；
- 属性的“可见性” (Visibility)分类：
 - 公有属性(public) +
 - 私有属性(private) -
 - 保护属性(protected) #
- 属性的表达方式：
 - 可见性 属性名：数据类型=初始值

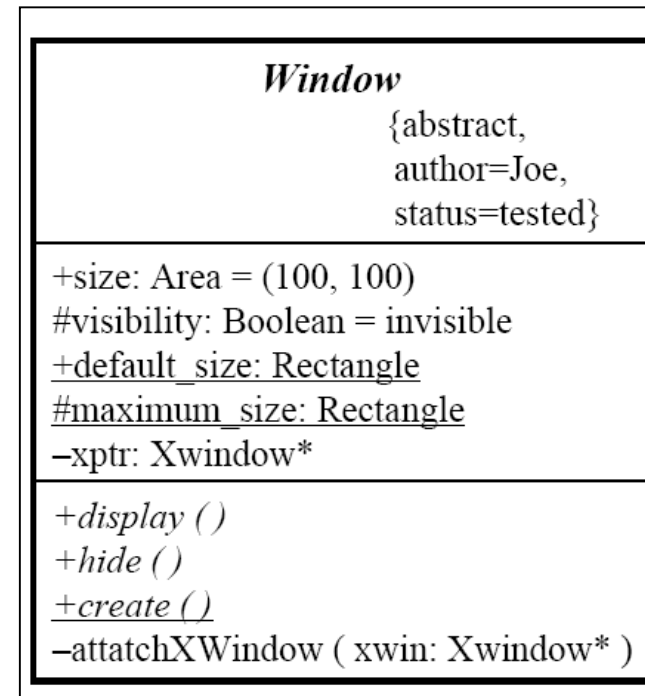


- 类的操作：描述对象“动态” (行为) 的特征的一个函数或过程；
 - 方法的“可见性” (Visibility) 分类：
 - 公有属性(public) +
 - 私有属性(private) -
 - 保护属性(protected) #
 - 方法的表达方式：
 - 可见性 方法名(参数列表)：返回值数据类型
 - 例如：+ getNextSentence (int i) : string
- ```

classDiagram
 class Window {
 +ab
 aut
 sta
 +size: Area = (100, 100)
 #visibility: Boolean = inv
 +default_size: Rectangle
 #maximum_size: Rectangle
 }

```

The diagram shows a class named **Window**. It has four attributes: `+ab`, `aut`, `sta`, and `+size: Area = (100, 100)`. It has three methods: `#visibility: Boolean = inv`, `+default_size: Rectangle`, and `#maximum_size: Rectangle`. The attributes `+ab`, `aut`, and `sta` are listed in the top section, while the others are in the bottom section. The methods are listed in the bottom section.



## 类的作用

- 类是一个支持继承的抽象数据类型；
- 类是创建(实例化)对象的模板，类是对对象的抽象；
- 类是一个命名空间，为类的泛化声明建立作用域；
- 类类似一张表，表内描述了数据和操作的封装体。

## 类使用的重点

- 类的确定与划分是面向对象方法成功应用的关键
- 类与对象分析实现的好坏严重影响软件的质量
- 类划分的好，有利于系统的理解与实现，程序的修改扩充，可重用性好，软件生命力更强。
- 后续章节讲述的重点内容

## 面向对象中的基本概念

- Coad和Yourdon给出了面向对象的定义：

面向对象 = 对象 + 类 + 继承 + 消息

- 对象(Object)
- 类(Class)
- 消息(Message)
- 封装(Encapsulation)
- 泛化(Generalization)与继承(Inheritance)
- 多态(Polymorphism)
- 抽象类(abstract class)与接口(Interface)

## 消息(Message)

- **消息(Message):** 一个对象向其他对象发出的请求，一般包含**消息接收对象、接收对象所采用的方法、方法需要的参数、返回信息**等；
  - 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。





## 面向对象中的基本概念

- Coad和Yourdon给出了面向对象的定义：

面向对象 = 对象 + 类 + 继承 + 消息

- 对象(Object)
- 类(Class)
- 消息(Message)
- 封装(Encapsulation)
- 泛化(Generalization)与继承(Inheritance)
- 多态(Polymorphism)
- 抽象类(abstract class)与接口(Interface)

# 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。
- 一个对象不能直接操作另一个对象内部数据，它也不能使其他对象直接访问自己的数据，所有的交流必须通过方法调用。
  - getXXX()
  - setXXX()
- 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。

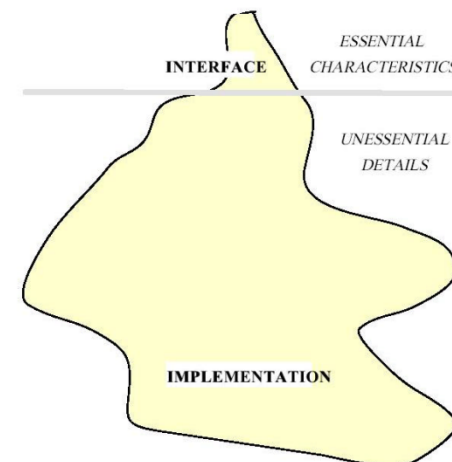


## “封装”的作用

- 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？
  - 数据的安全性：保护对象，避免用户误用。
  - 模块的独立性：保护客户端(调用程序)，其实现过程的改变不会影响到相应客户端的改变。
  - 易开发、易维护性：隐藏复杂性，降低了软件系统开发难度；各模块独立组件修改方便，重用性好。



# 面向对象的基本概念

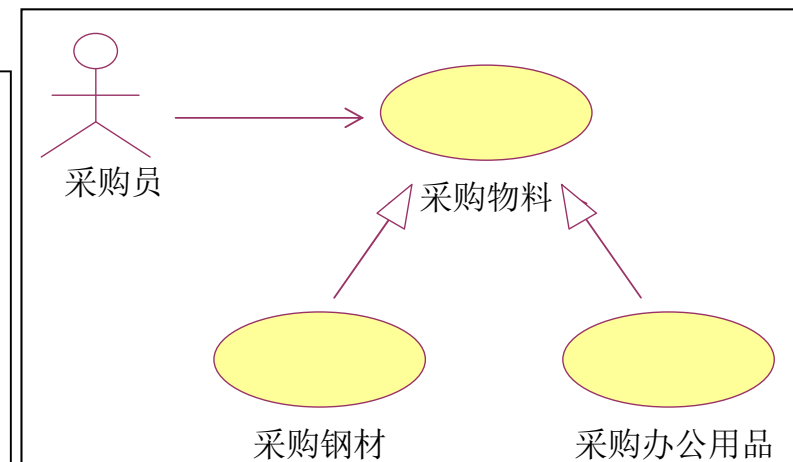
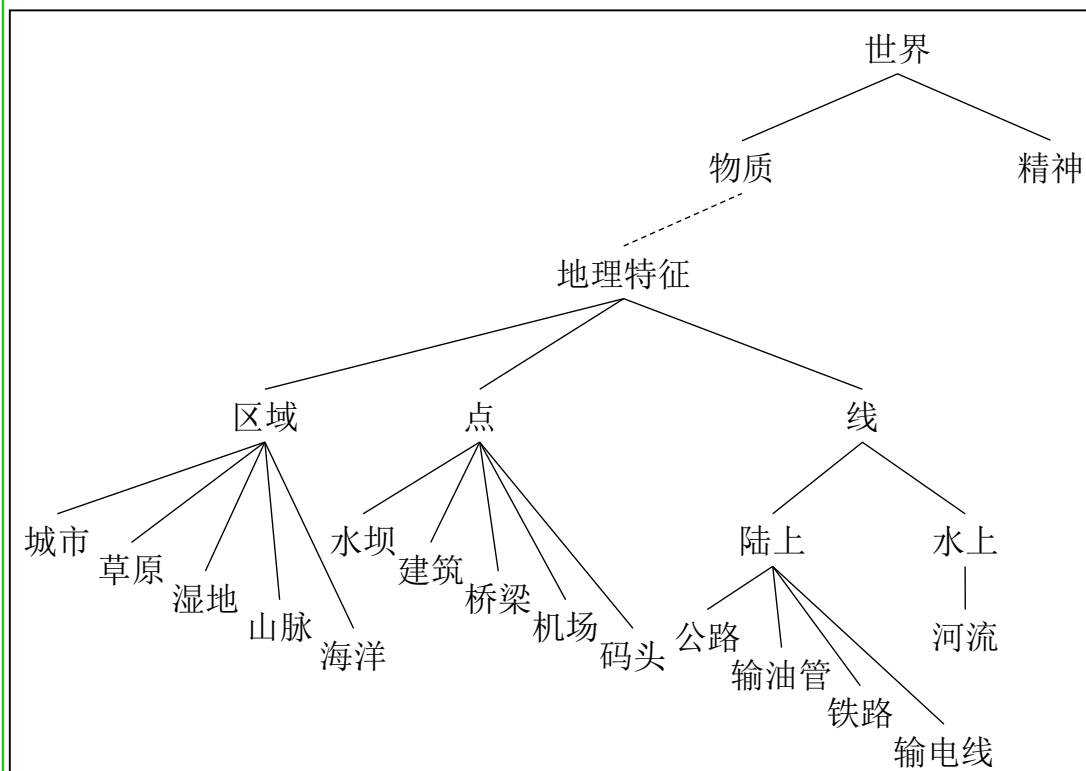
- Coad和Yourdon给出了面向对象的定义：

面向对象 = 对象 + 类 + 继承 + 消息

- 对象(Object)
- 类(Class)
- 消息(Message)
- 封装(Encapsulation)
- 泛化(**generalization**)与继承(**Inheritance**)
- 多态(Polymorphism)
- 抽象类(**abstract class**)与接口(**Interface**)

# 泛化(generalization)与继承(Inheritance)

- **泛化(generalization)/继承(Inheritance)** 关系是类元的一般描述和具体描述之间的关系，具体描述建立在一般描述的基础之上，并对其进行了扩展。

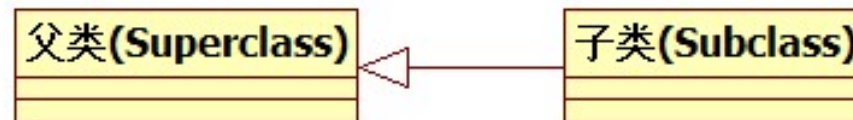


用例之间的泛化关系

继承关系使类之间  
形成层次化结构

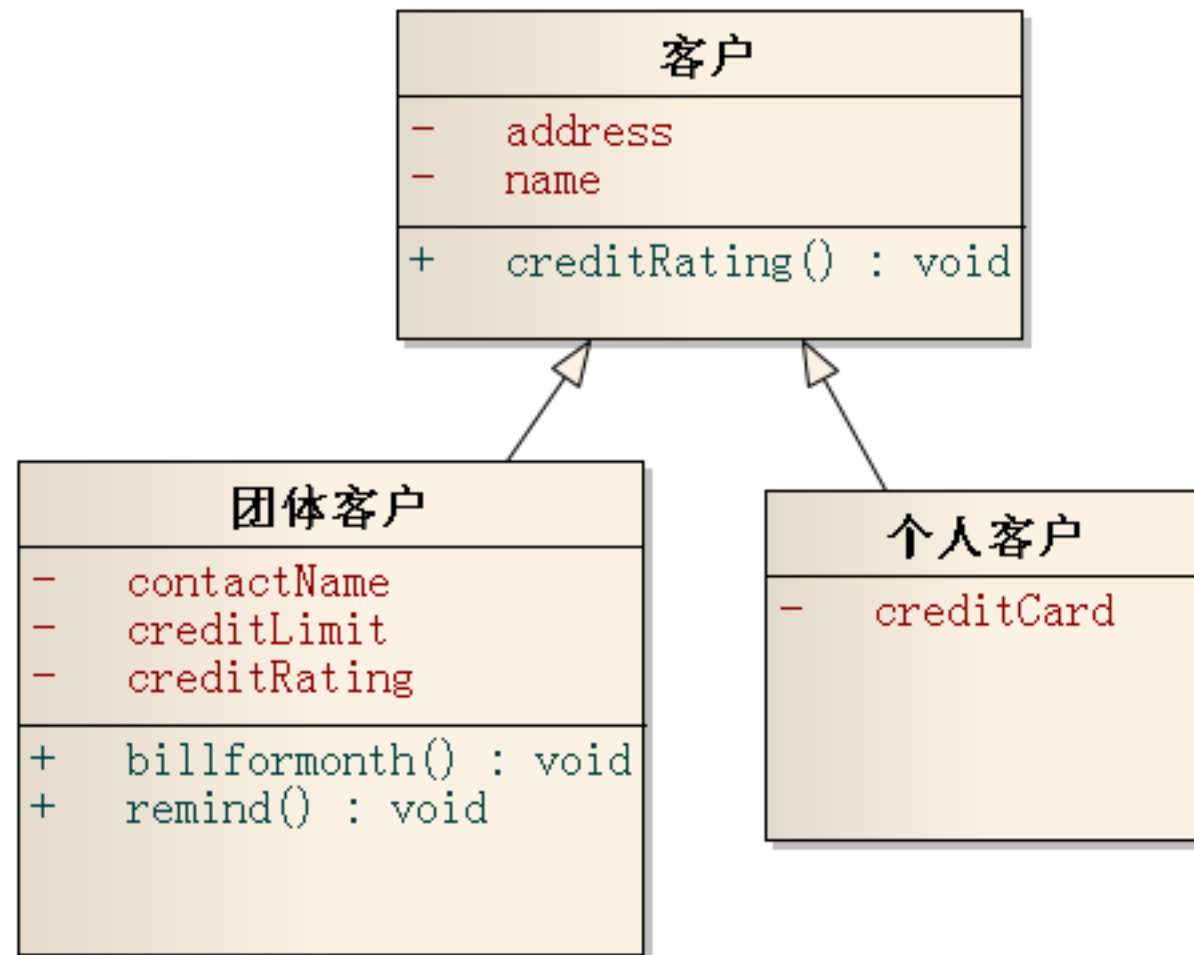
## 继承(Inheritance)

- **继承(Inheritance):** 子类 (Subclass) 可自动拥有父类/超类 (SuperClass) 的全部属性和操作。
- 表示两个类之间是 “is a”、“is like” 或 “is a kind of” 关系。
- 继承使两个以上的类共享相同的属性和/或相同方法。
- 继承的各个类间形成层次结构。
- **继承提高了复用性，使多态成为了可能。**
- “继承” 与 “封装” 间存在一定的违背，利用继承暴露了被继承类的一些秘密，要理解某个类的含义，有时需要研究它的所有超类，有时还要研究超类的内部结构。



## 继承 (Inheritance)

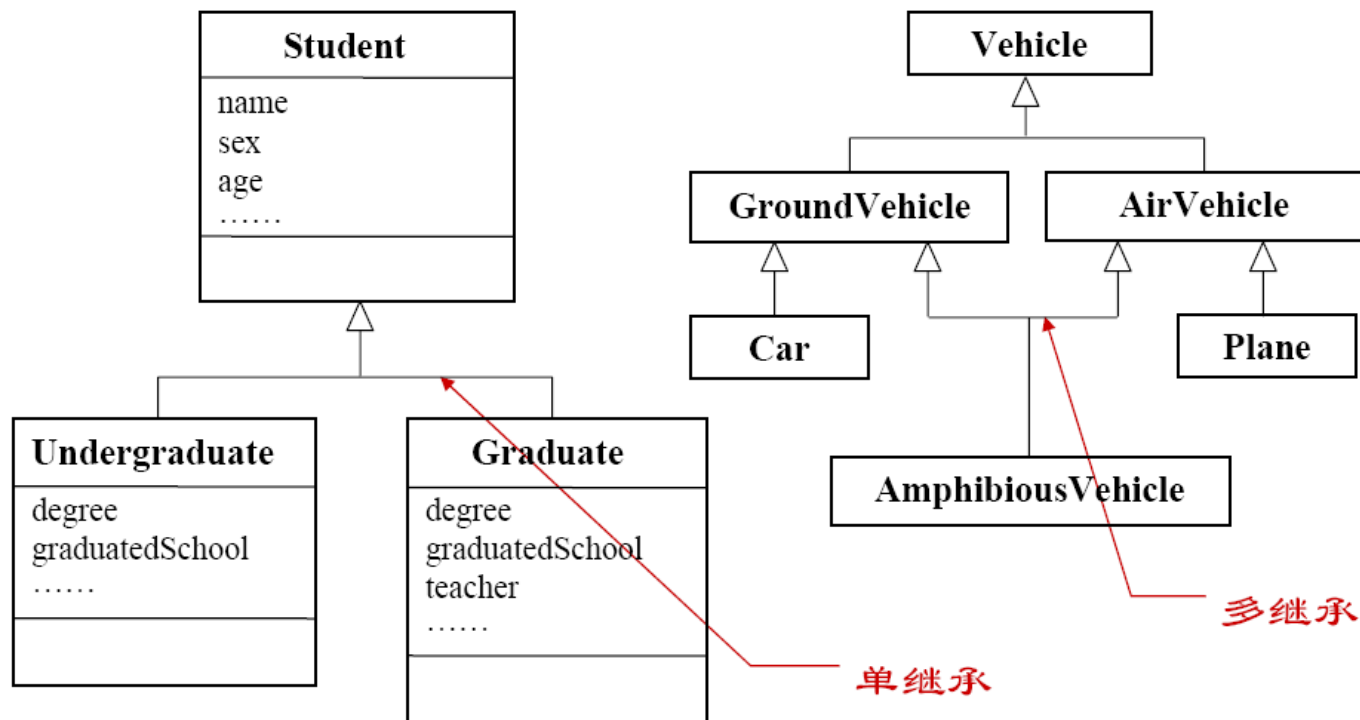
- 银行客户的继承



# 继承(Inheritance)

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

```
class Graduate extends Student{
 //methods and fields
}
```





## 继承 (Inheritance)

- **多重继承 (multiple inheritance)**：允许子类拥有一个以上的父类，并继承父类的特征。优点是在识别类和增加复用上会更具效力，缺点是失去了概念上和实现上的简洁性。
- **多重继承注意事项：**
  - 如何处理来自多个超类的名字冲突，以及重复的继承？
  - 方法1：编程语言认为冲突非法；
  - 方法2：编程语言认为引入的相同名字是相同的属性；
  - 方法3：编程语言允许冲突，但是冲突变量要被严格限定出处位置；
  - 建议：在建模阶段避免冲突，通过重命名的方式解决。

# 面向对象的基本概念

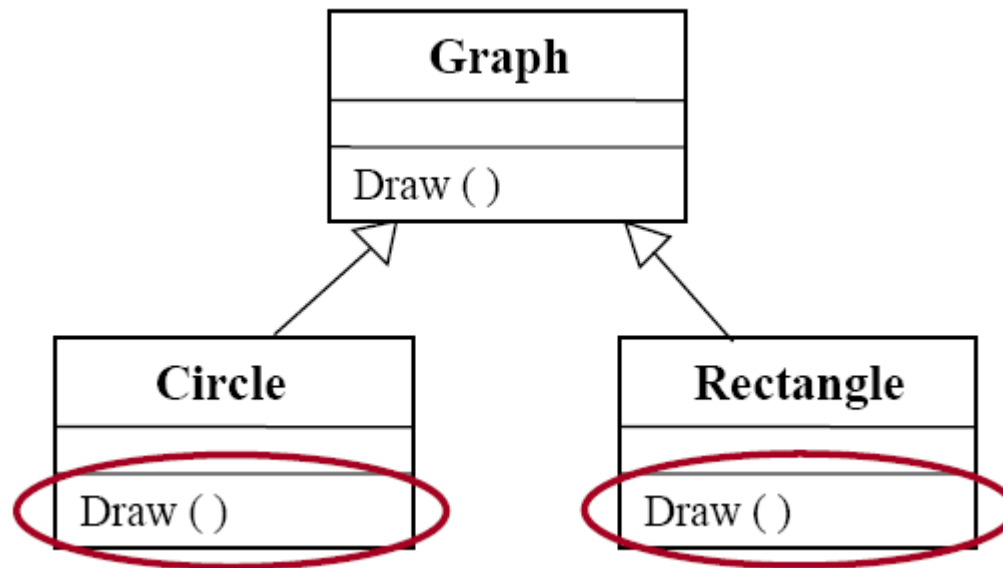
- Coad和Yourdon给出了面向对象的定义：

面向对象 = 对象 + 类 + 继承 + 消息

- 对象(Object)
- 类(Class)
- 消息(Message)
- 封装(Encapsulation)
- 泛化(generalization)与继承(Inheritance)
- 多态(Polymorphism)
- 抽象类(abstract class)与接口(Interface)

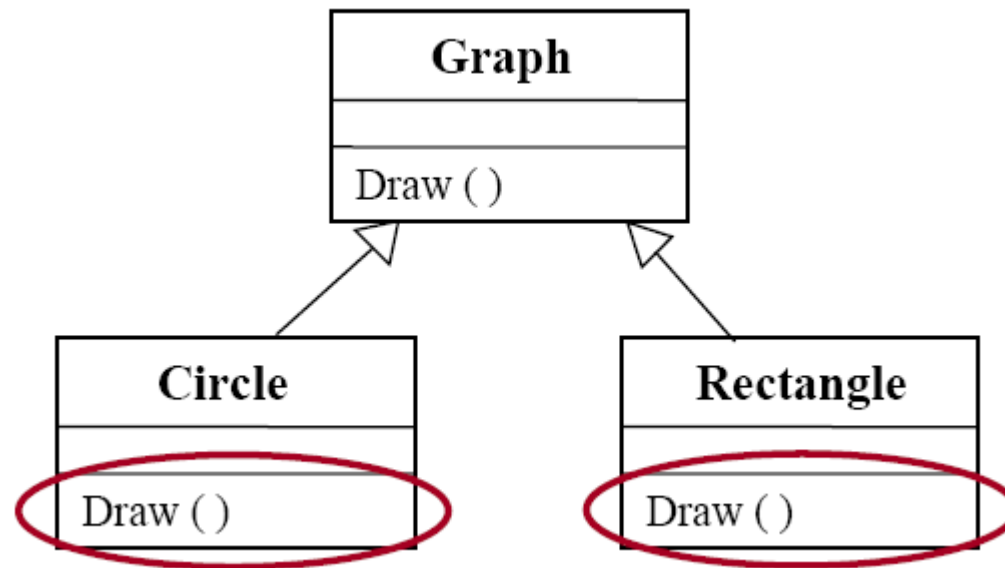
## 多态 (Polymorphism)

- **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。
- **继承**使得**多态**操作成为可能，即操作的实现是由它们所使用的对象的类，而不是由调用者确定的。



## 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。
- **继承**使得**多态**操作成为可能，即操作的实现是由它们所使用的对象的类，而不是由调用者确定的。



## 定义的父类型可用子类型替换

```
Vehicle[] vehicles=new Vehicle[3];
vehicles[0]=new Car();
vehicles[1]=new Plane();
vehicles[2]=new Train();

for(int i=0;i<vehicles.length;i++){
 vehicles[i].start();
 vehicles[i].ring();
}
```

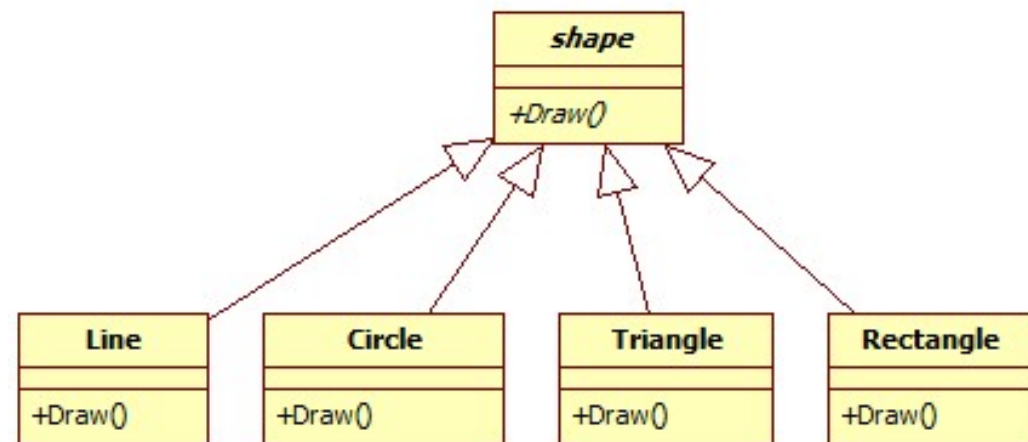
## 多态示例

- 各种图形具有共同的属性和操作：颜色、线型、旋转、移动。但在不同图形中对**Draw**的描绘虽然不同，外界都可以采用以下方式调用：

- **Shape aShape;**  
**aShape = new Line();**  
**aShape.draw();** //调用的是类**Line**的方法。

.....其他操作//如果对不同图形的调用逻辑和顺序是一致的，则此处的代码对所有图形均有效，利于程序的扩展

- 为外界调用提供统一的接口，使对图形扩充成为可能。



## 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，是将一个操作与不同方法关联的能力；
- 多态性增加了OO的灵活性，减少冗余信息，提高可重用性，可扩展性，易于维护。
- **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

## 面向对象的基本概念

- Coad和Yourdon给出了面向对象的定义：

面向对象 = 对象 + 类 + 继承 + 消息

- 对象(Object)
- 类(Class)
- 消息(Message)
- 封装(Encapsulation)
- 泛化(generalization)与继承(Inheritance)
- 多态(Polymorphism)
- 抽象类(abstract class)与接口(Interface)



## 抽象类

- **抽象类(abstract class)**: 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), 而仅仅能使用其子类的实例。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。
  - 抽象类的类名在模型中一般为“斜体”, 或在类名上方添加<<abstract>>标注; 抽象类型方法的名字也用“斜体”表示

```
abstract class Demo {

 abstract void method1();

 abstract void method2();

 ...

}
```



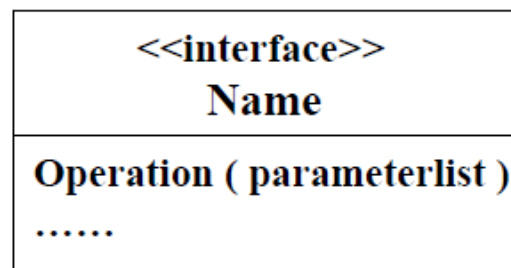
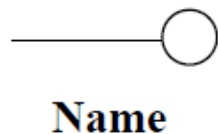
# 抽象类

## ■ 抽象类的出发点:

- 在OO中，所有的对象都是通过类来描绘的，但是反过来并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。
- 抽象类往往用来表征在对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。比如：图形编辑软件的开发，问题域存在着圆、三角形等不同的具体概念，但都属于形状这样一个概念，形状这个概念在问题域是不存在的，它就是一个抽象概念。正是因为抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能够实例化的。

## 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。
- 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”



## 接口 (Interface)

- 接口与类类似，但是只为其成员提供规约而不提供实现。
- 接口与只含有抽象方法的抽象类很相似。
- 以接口为中心的设计方法

```
public interface Comparable{

 public abstract int compareTo(Object other);
}

class Student implements Comparable{
 public int compareTo(Object otherobject){
 Student other=(Student)otherobject;
 if (age<>other.age) return -1;
 if (age==other.age) return 1;
 return 0;
 }
}
```

## 接口与抽象类的区别

- **interface与abstract class**对于“抽象的类”定义的支持方面具有很大的相似性，甚至可以相互替换，但**两者之间还是有很大区别的**，对于它们的选择甚至反映出对于问题领域本质的理解、对于设计意图的理解是否正确、合理。
- **从语法定义层面看**
  - abstract class可以有自己的数据成员，也可以有非abstract的成员方法；
  - interface方式的实现中，接口只能够有静态的不能被修改的数据成员（也就是必须是static final的，不过在interface中一般不定义数据成员），所有的成员方法都是abstract的。
  - 从某种意义上说，interface是一种特殊形式的abstract class。

```
abstract class Demo {

 abstract void method1();
 void method2(XXXXX);
 ...
}
```

```
public interface Demo {

 abstract void method1();
 abstract void method2();
 ...
}
```

## 接口与抽象类的区别

### ■ 从编程层面看

- `abstract class`在Java语言中表示的是一种**继承关系**，一个类只能使用一次继承关系(单继承)。但是，一个类却可以实现多个**interface**。这是Java对于多重继承的支持方面的一种折中考虑。
- 在**abstract class**的定义中，可以赋予方法默认的行为。但是在**interface**的定义中，方法却不能拥有默认行为。
- 不能定义默认行为可能会造成维护上的麻烦，如：类的界面发生变化时，如果是通过接口实现的，需要修改大量派生类的方法进行处理，而通过**abstract class**实现的只需要修改**abstract class**中的默认行为即可。
- 不能定义默认行为会导致同样的方法实现出现在该抽象类的每一个派生类中，违反了“one rule, one place”原则，造成代码重复，同样不利于以后的维护。
- 因此，在**abstract class**和**interface**间进行选择时要慎重。

## 接口与抽象类的区别

### ■ 从设计理念层面看

- `abstract class`体现了一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在“is a”关系，即父类和派生类在概念本质上应该是相同的。
- 对于`interface`来说则不然，并不要求`interface`的实现者和`interface`定义在概念本质上是一致的，仅仅是实现了`interface`定义的契约而已。

## 接口与抽象类的区别

- 要求**Door**具有报警的功能应该如何处理？

使用**abstract class**方式定义**Door**:

```
public abstract class Door {

 abstract void open();

 abstract void close();

}
```

使用**interface**方式定义**Door**:

```
public interface Door {

 abstract void open();

 abstract void close();

}
```



## 接口与抽象类的区别

- 要求Door具有报警的功能应该如何处理？
- 解决方案一：简单的在Door的定义中增加一个alarm方法

使用**abstract class**方式定义Door:

```
public abstract class Door {
 abstract void open();
 abstract void close();
 abstract void alarm();
}
```

class AlarmDoor extends Door{...}

使用**interface**方式定义Door:

```
public interface Door {
 abstract void open();
 abstract void close();
 abstract void alarm();
}
```

class AlarmDoor implements Door{...}

- 这种方法违反了接口隔离原则ISP，在Door的定义中把Door概念本身固有的行为方法和另外一个概念“报警器”的行为方法混在了一起。这样引起的一个问题是那些仅仅依赖于Door这个概念的模块会因为“报警器”这个概念的改变（比如：修改alarm方法的参数）而改变，反之亦然。

## 接口与抽象类的区别

- 要求**Door**具有报警的功能应该如何处理？
- 解决方案二
  - 方法open()和close()同方法alarm()分属于不同的概念，应分别定义在不同的抽象类或接口中。
  - 使用两个abstract class? Java不支持多继承，不可行。
  - 两个interface? 一个abstract class，一个interface? 方式可行，但反映出对于问题领域中的概念本质的理解、对于设计意图的反映不同。
- 如果两个概念都使用**interface**方式来定义？
  - 1、AlarmDoor在概念本质上到底是Door还是报警器？
  - 2、如果AlarmDoor在概念本质上和Door是一致的，那么“均使用interface方式”在实现时就没有能够正确的揭示设计意图，因为在这两个概念的定义上反映不出上述含义。

## 接口与抽象类的区别

### ■ 一个abstract class，一个interface？

- 如果对于问题领域的理解是：AlarmDoor在概念本质上是Door，同时它有具有报警的功能。
- 对于Door这个概念，应该使用abstract class方式来定义。
- 另外，AlarmDoor又具有报警功能，说明它又能够完成报警概念中定义的行为，所以报警概念可以通过interface方式定义。

```
Public abstract class Door {
```

```
 abstract void open();
 abstract void close();
}
```

```
Public interface Alarm {
```

```
 abstract void alarm();
}
```

```
Public class AlarmDoor extends Door
implements Alarm {
```

```
 void open() { ... }
 void close() { ... }
 void alarm() { ... }
}
```



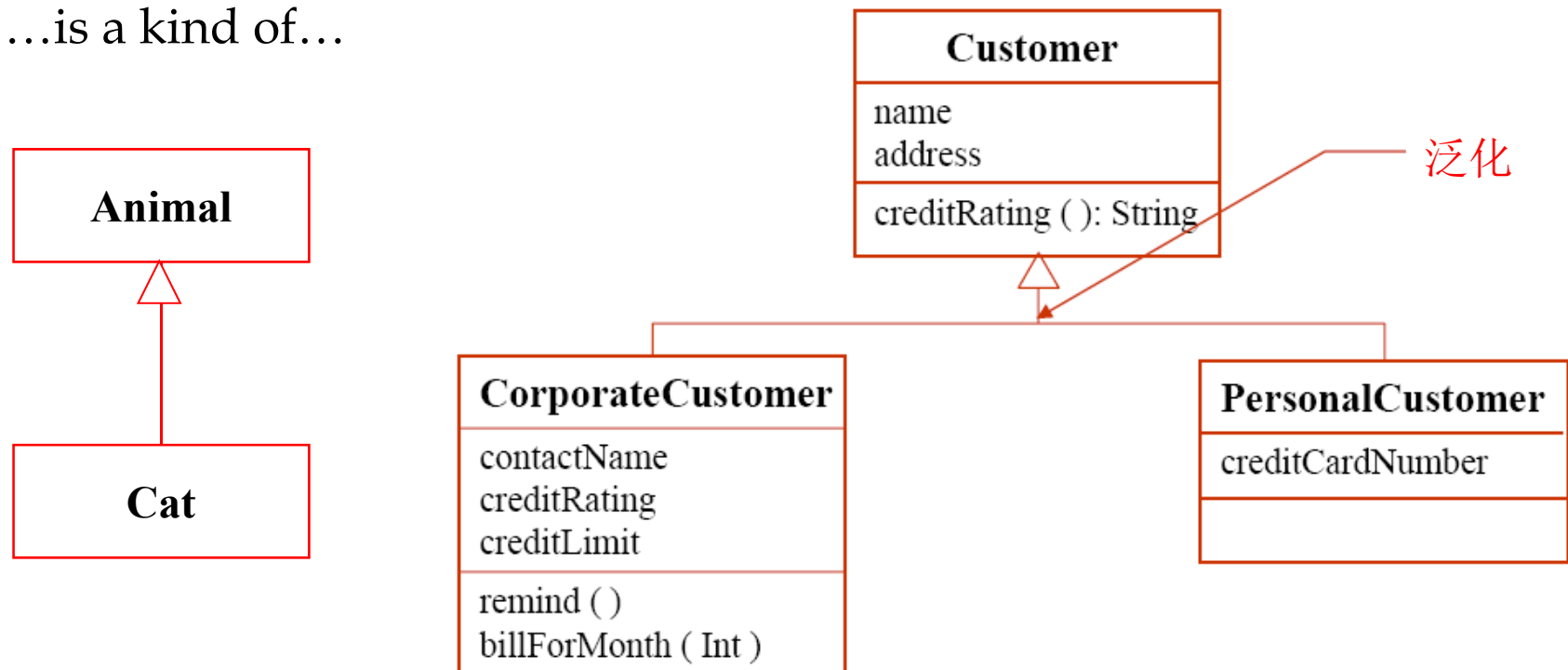
### 3. 对象之间的五类关系

## 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...



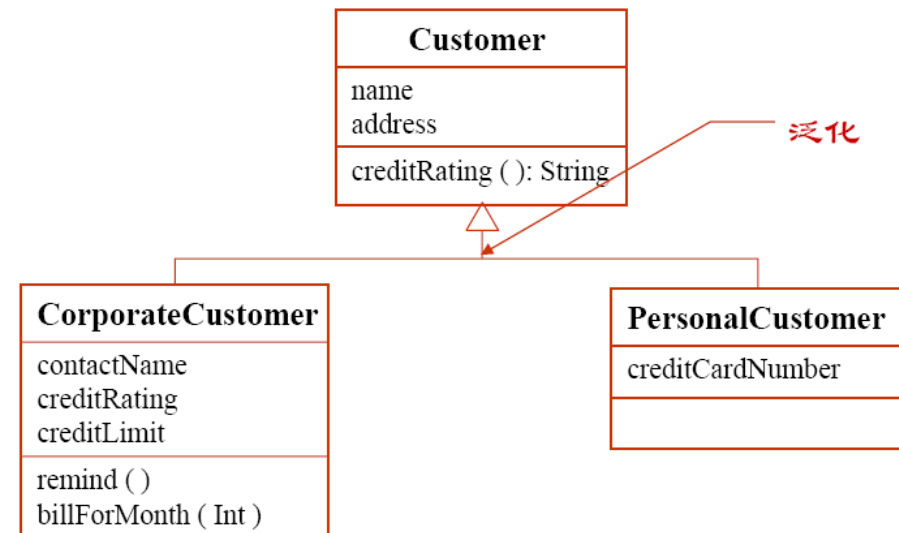
**A cat is a kind of animal;**  
**A corporate customer is a kind of customer;**  
**A personal customer is a kind of customer, too;**

## (1) 分类结构：继承/泛化关系

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

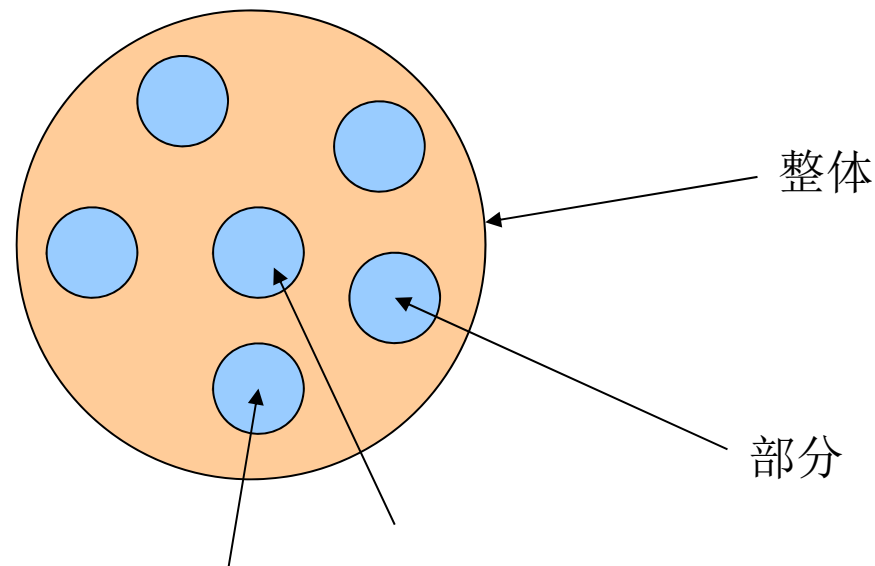
```
class CorporateCustomer : Customer {
 String contactName;
 String creditRating;
 String creditLimit;

 void Remind() { };
 void billForMonth(int bill) { };
}
```



## (2) 组成结构：聚合与组合关系

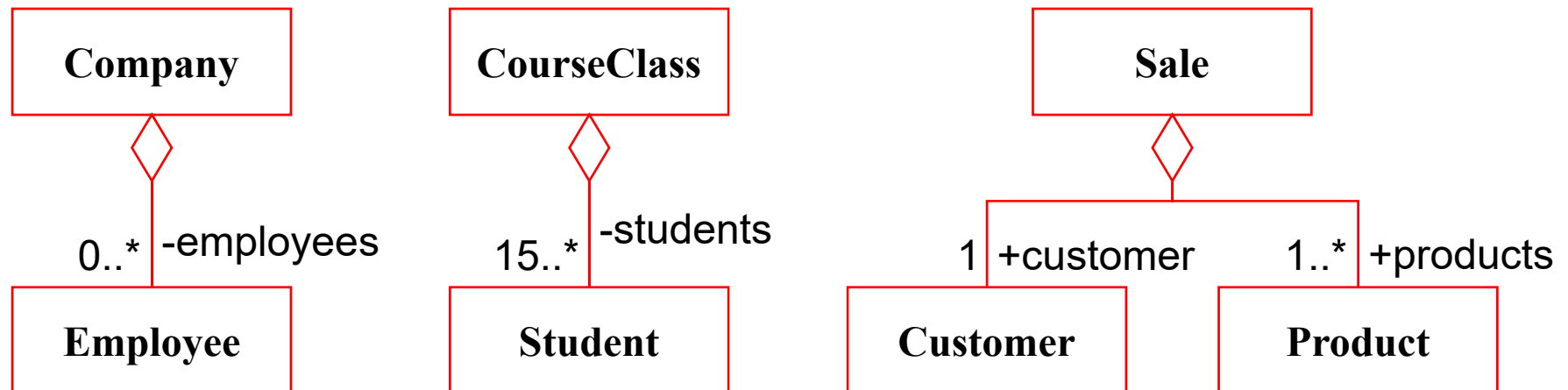
- 组成结构：表示对象类之间的组成关系，一个对象是另一个对象的一部分，即“部分-整体”关系。
- 分为两个子类：
  - 聚合(Aggregation): 整体与部分在生命周期上是独立的 (...owns a...);
  - 组合(Composition): 整体与部分具有同样的生命周期(...is part of...);





## (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 Sale owns a customer and a set of products;**

## (2) 组成结构：聚合关系

定义两个类：

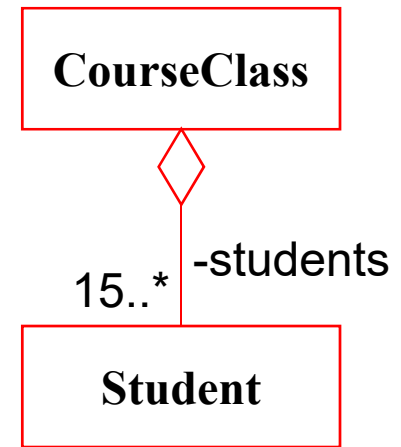
```
class Student {}
```

```
class CourseClass {
 ...
 private Student[] students;
 public addStudent (Student s) {
 students.append(s);
 }
 ...
}
```

使用时的代码：

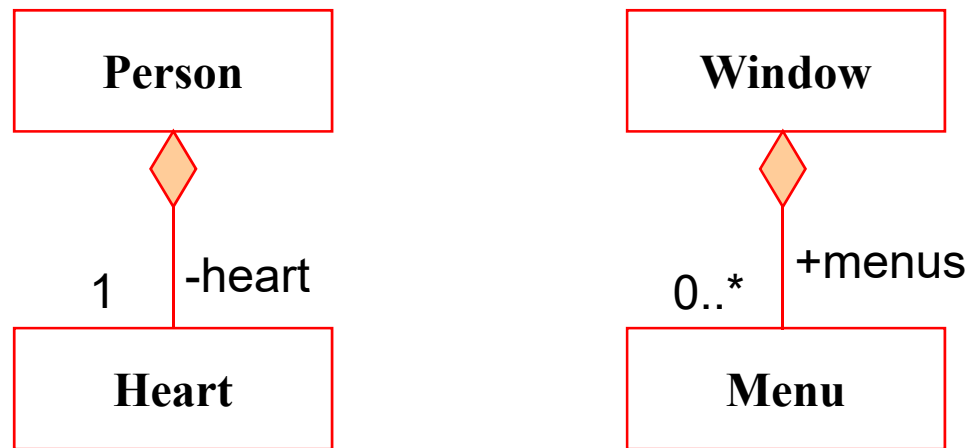
```
Student a = new Student ();
Student b = new Student ();
Student n = new Student ();
```

```
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): 强调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 {}
```

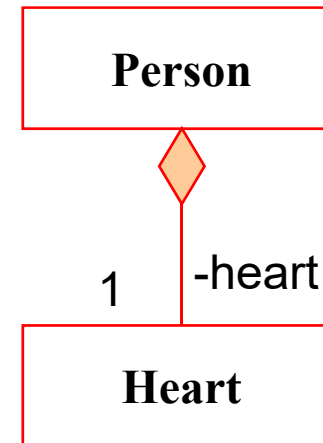
```
class Person {
```

```
...
```

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

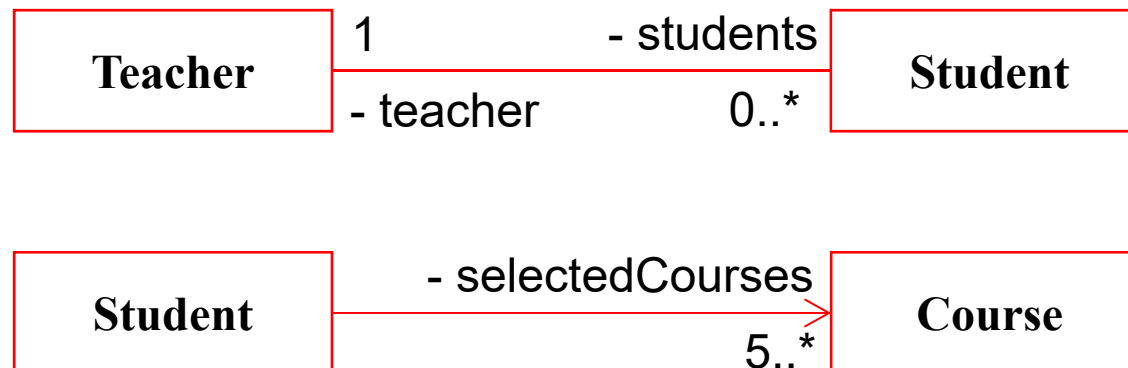
```
...
```

```
}
```



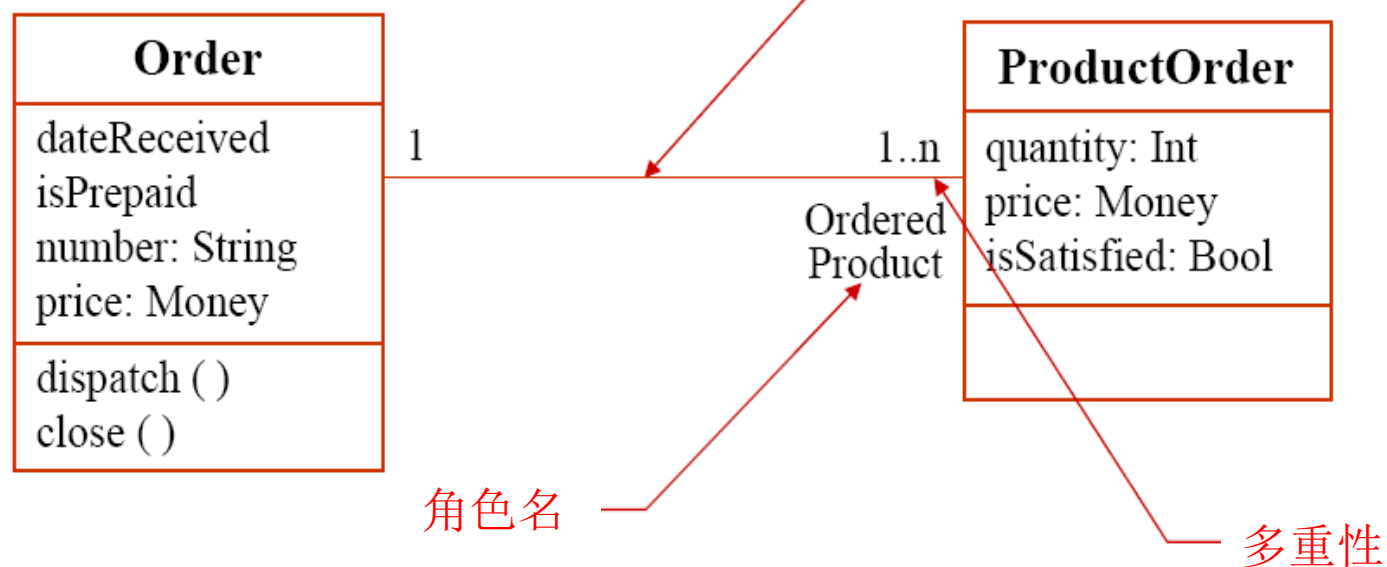
### (3) 实例连接：关联关系

- 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。



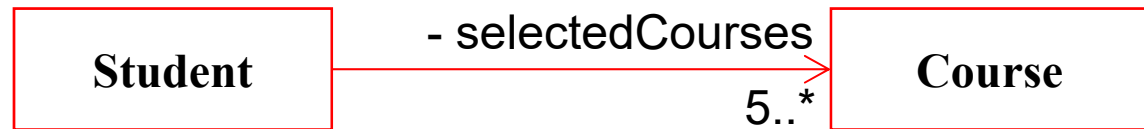
### (3) 实例连接：关联关系

- 关联具有多重性(重数): 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

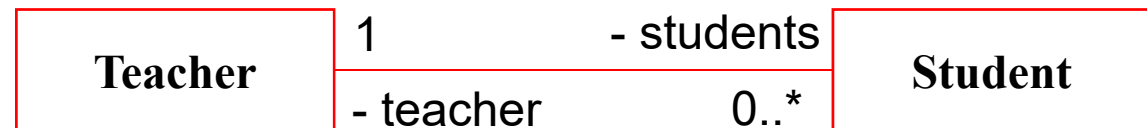


### (3) 实例连接：关联关系

```
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 {}
```

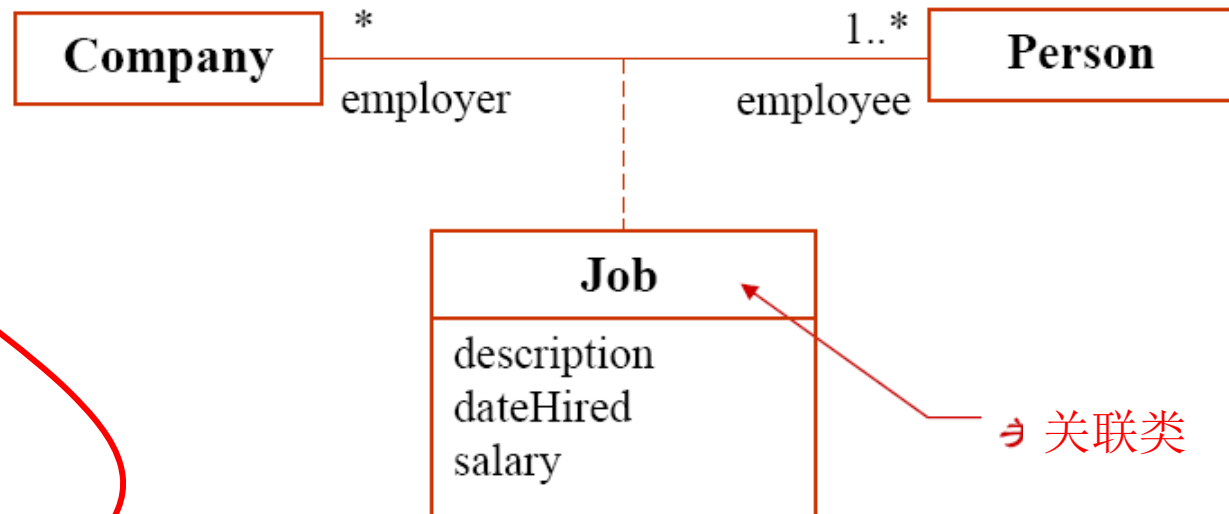
```
class Job {
 public Person employee;
 public Company employer;
}
```

```
String description;
```

```
Date dateHired;
```

```
double salary;
```

```
}
```



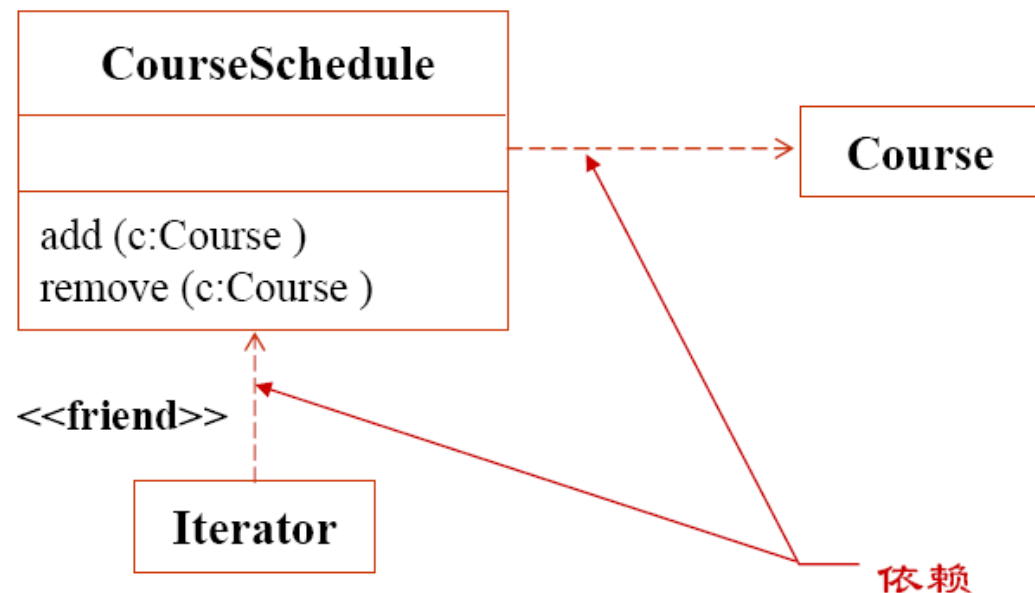


## (4) 消息连接：依赖关系

### ■ 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)。



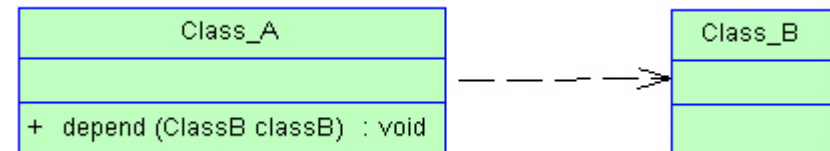
## (4) 消息连接：依赖关系

### ■ 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) {}
}
```

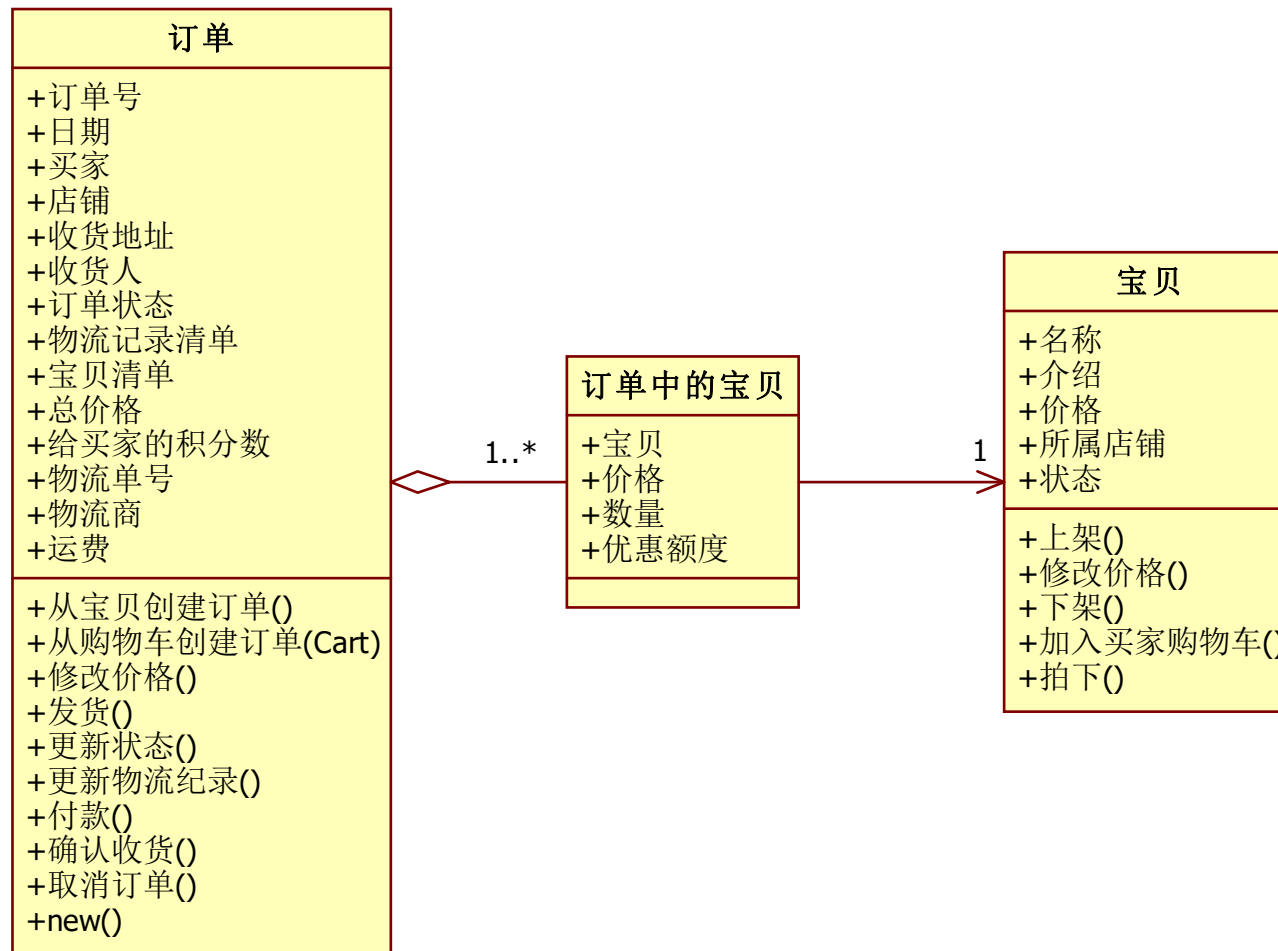
## “依赖”关系怎么识别？

- 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。

## 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：订单和商品这两个实体类，按照常规理解，似乎是聚合关系，一个订单包含多种商品。但是在这种聚合发生时，产生了某些新信息(该商品在该订单中的价格，可能与卖家给商品设定的价格不同；本次购买的数量)。这些新信息，既不属于订单，也不属于商品本身，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；
  - 商品：名字、所属卖家、设定价格、etc；
  - 订单项：商品、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，其实是可以用来表示的。
- 例如：
  - 对“购物车”和“商品”两个类而言，可以说“多个商品对象聚合成了购物车对象”，商品是购物车的一部分；
  - 也可以说“购物车 has some 商品”，二者是多对多的关联关系，一个购物车里有0..\*个商品，一个商品可以在0..\*个购物车内出现，商品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商品对象，故这是一个从购物车类指向商品的单项关联。
- 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（例如商品和卖家，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“买家对商品的评价”和“商品”，商品的地位更高，评价往往看作它的一部分）。

## “依赖”关系怎么识别？

### ■ 例如：

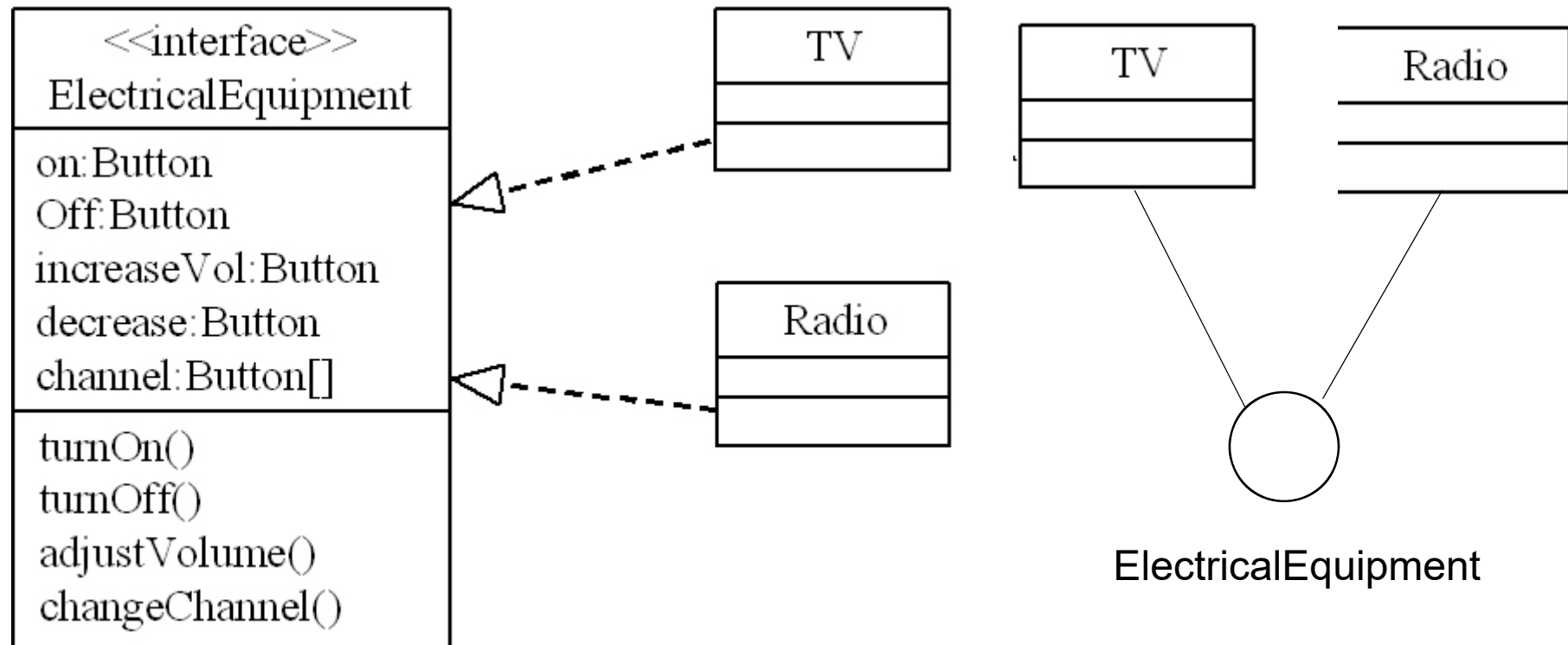
- “订单”类中有一个操作“从购物车生成订单()”，它有一个参数，类型是“购物车”类，该操作在执行时，根据传入的“购物车”对象读取产品信息来生成多个“订单项”对象。这个操作执行完之后，“订单”类和“购物车”类就再无关系，故前者依赖于后者。

### ■ 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;
- “棒棒糖” 另一种形式。

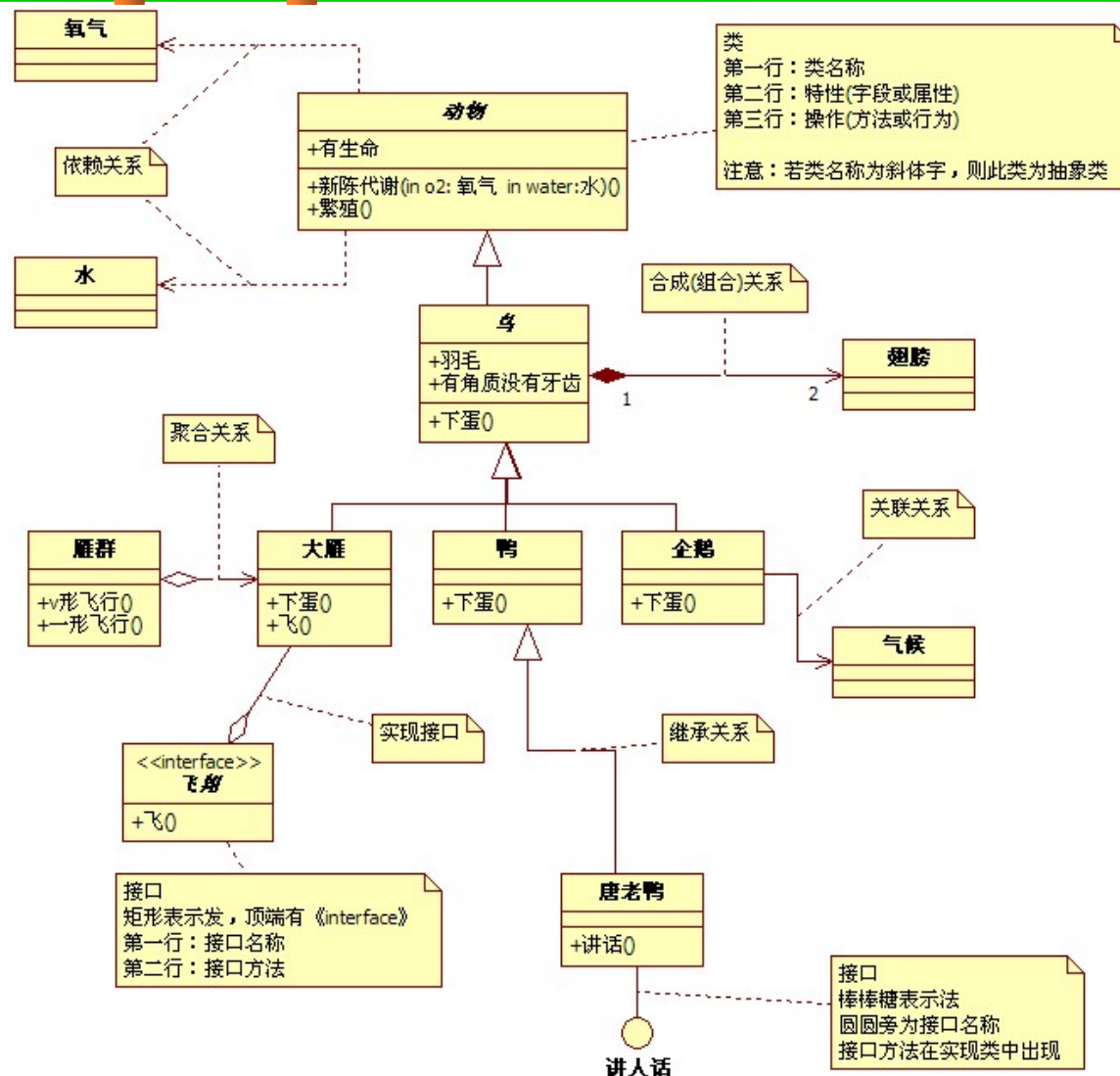




## 小结：对象之间的联系

- 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

# 面向对象概念的一个综合例子



摘自：[http://www.nowamagic.net/architecture/architecture\\_PicsToIntroduceOOP.php](http://www.nowamagic.net/architecture/architecture_PicsToIntroduceOOP.php)



## 4. UML建模语言简述



## 建模是一种设计技术

- 模型是某个事物的抽象，其目的是在构建这个事物之前先来理解它
  - 在构建物理实体之前先验证
  - 通过抽象降低复杂度
  - 可视化，便于与客户和其他小组成员交流
  - 为维护 and 升级提供文档
- 建模的主要目的是为理解，而非文档
- 软件开发中建模的过程
  - 分析建模
  - 设计建模
  - 实现建模
  - 部署建模

## 面向对象建模的三个不同视角

- 类模型：表示系统静态的、结构化的“数据”层面
  - 描述系统中对象的结构：它们的标识、属性和操作
  - 描述与其他对象的关系
- 状态模型：表示对象时序的、行为的“控制”层面
  - 标记变化的事件
  - 界定事件上下文的状态
  - 一个类有一个状态图
- 交互模型：表示独立对象的协作“交互”层面
  - 系统行为如何完成
  - 对象间如何协作

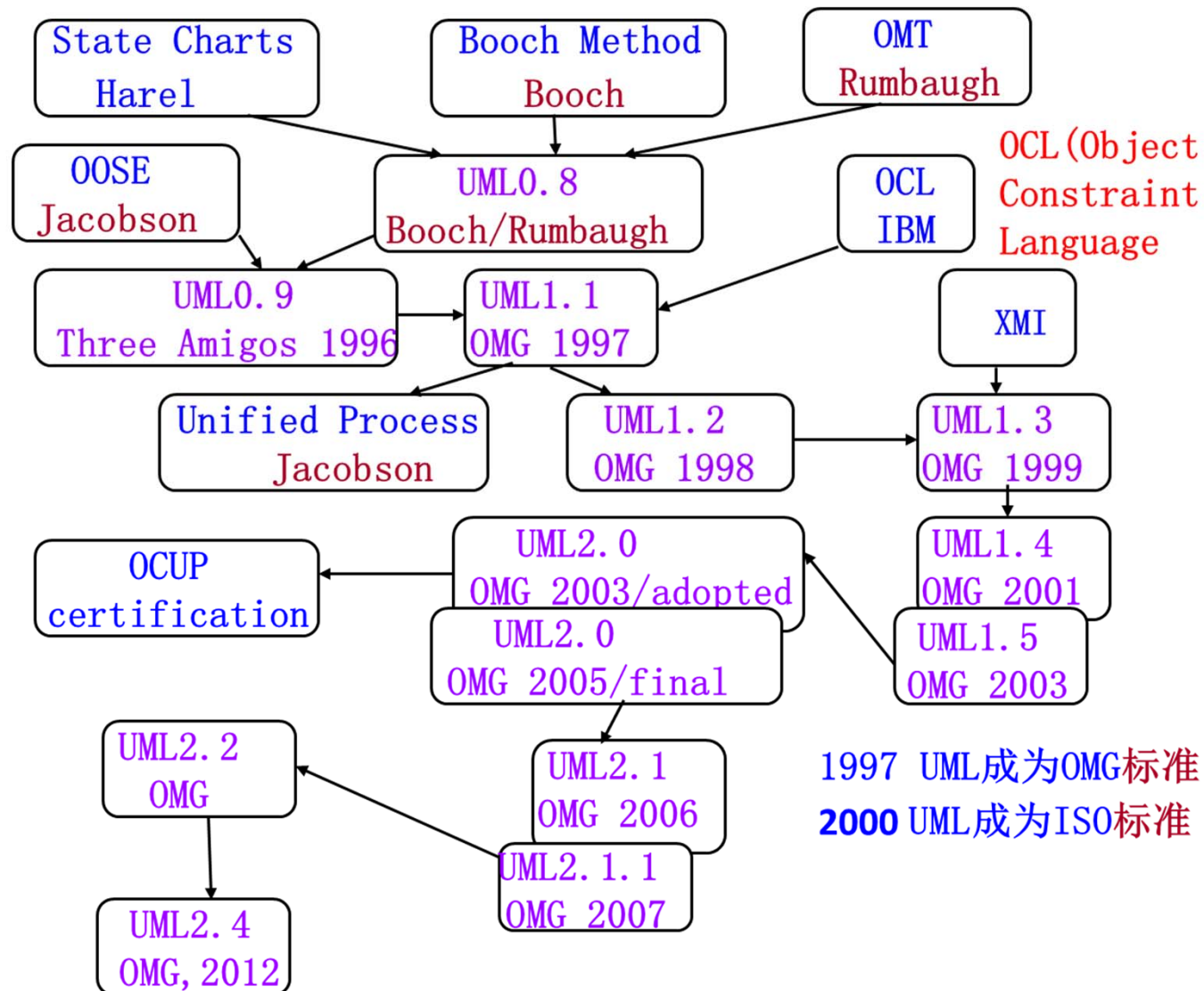
# 什么是UML?

- 统一建模语言（ Unified Modeling Language, UML）是描述、构造和文档化系统制品的可视化语言（OMG03a）。
  - 由Booch、Rumbaugh和Jacobson合作创建
  - 统一了主流的面向对象的分析设计的表示方法
- UML的定义包括UML语义和UML表示法两个部分。
  - UML语义：UML对语义的描述使开发者能在语义上取得一致认识，消除了因人而异的表达方法所造成的影响。
  - UML表示法：UML表示法定义UML符号的表示法，为开发者或开发工具使用这些图形符号和文本语法为系统建模提供了标准。
- 面向对象建模的图形化表示法的标准
  - UML不是可视化程序设计语言，而是一个可视化的建模语言
  - UML不是OOA/D，也不是方法，它只是图形表示工具

## 讨论题

- UML是由三个流派合并形成，分析三者的异同点？ 1组

# UML的发展历史

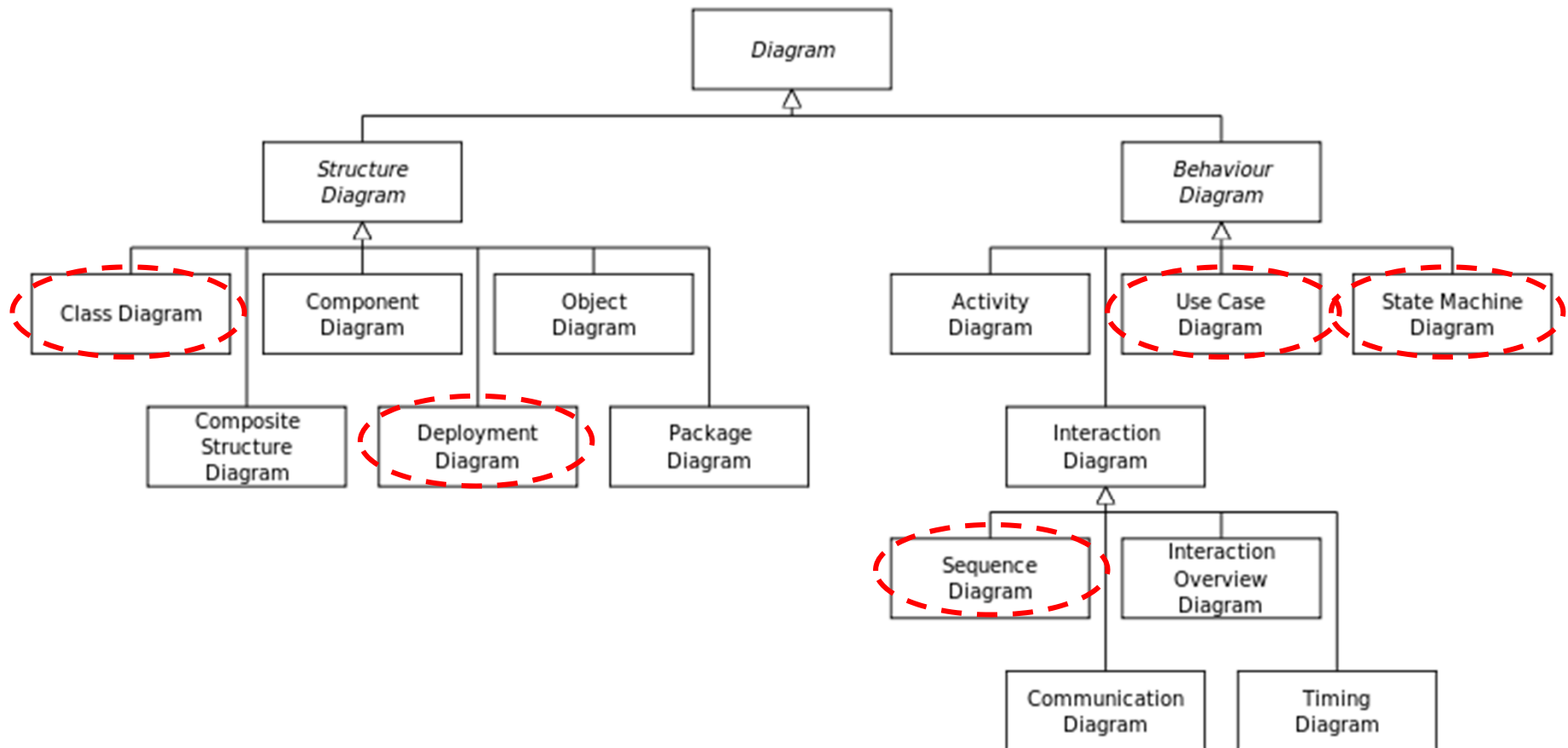




## 应用UML的三种方式

- **UML作为草图：**非正式、不完整的，用于探讨问题和交流
- **UML作为蓝图：**相对详细的设计图，用于代码生成和逆向工程
- **UML作为编程语言：**用UML完成系统可执行规格说明，模型驱动体系结构（MDA）的应用方式，尚在发展阶段

# UML模型



# UML视图和图

|      | 视图    | 图         | 主要概念                    |
|------|-------|-----------|-------------------------|
| 结构分类 | 静态视图  | 类图/对象图/包图 | 类、对象、包、关联、泛化、依赖关系、实现、接口 |
|      | 物理视图  | 构件图       | 构件、接口、依赖关系、实现           |
|      |       | 部署图       | 节点、构件、依赖关系、位置           |
| 行为分类 | 用例视图  | 用例图       | 用例、参与者、关联、扩展、包括、用例泛化    |
|      | 状态机视图 | 状态图       | 状态、事件、转换、动作、            |
|      | 活动视图  | 活动图       | 状态、活动、完成转换、分叉、结合        |
|      | 交互视图  | 顺序图       | 交互、对象、消息、激活             |
|      |       | 协作图       | 协作、交互、协作角色、消息           |

# UML视图

- 结构分类：描述了系统中的结构成员及其相互关系
  - 静态视图对应用领域中的概念以及与系统实现有关的内部概念建模
  - 物理视图对应用自身的实现结构建模。
    - 物理视图有两种：实现视图和部署视图
    - 实现视图为系统的构件模型及构件之间的依赖关系。
    - 部署视图描述位于节点实例上的运行构件实例的安排。
- 行为分类：描述了系统的功能和行为
  - 用例视图是外部用户所能观察到的系统功能的模型图
  - 状态机视图是一个类对象所可能经历的所有历程的模型图。
  - 活动视图是状态机的一个变体，用来描述执行算法或工作流程中涉及的活动
  - 交互视图描述了执行系统功能的各个角色之间相互传递消息的顺序关系。

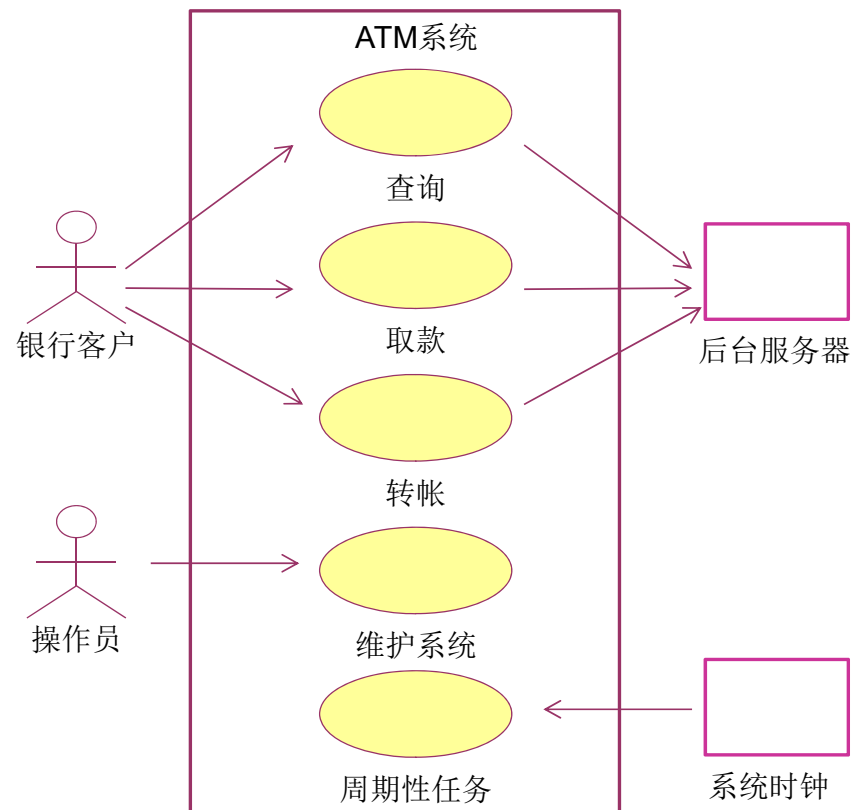
## UML 图

- **类图/对象图**：描述系统的各个对象类型以及存在的各种静态关系
- **用例图**：描述用户与系统如何交互
- **构件图**：构件间的组织结构及链接关系
- **部署图**：制品在节点上的部署
- **状态图**：事件在对象的周期内如何改变状态
- **活动图**：过程及并行行为
- **顺序图**：对象间的交互；强调顺序
- **协作图**：对象间交互，重点在对象间链接关系

## 各UML图及特征

### ■ 1. 用例图（ Use Case Diagram ）

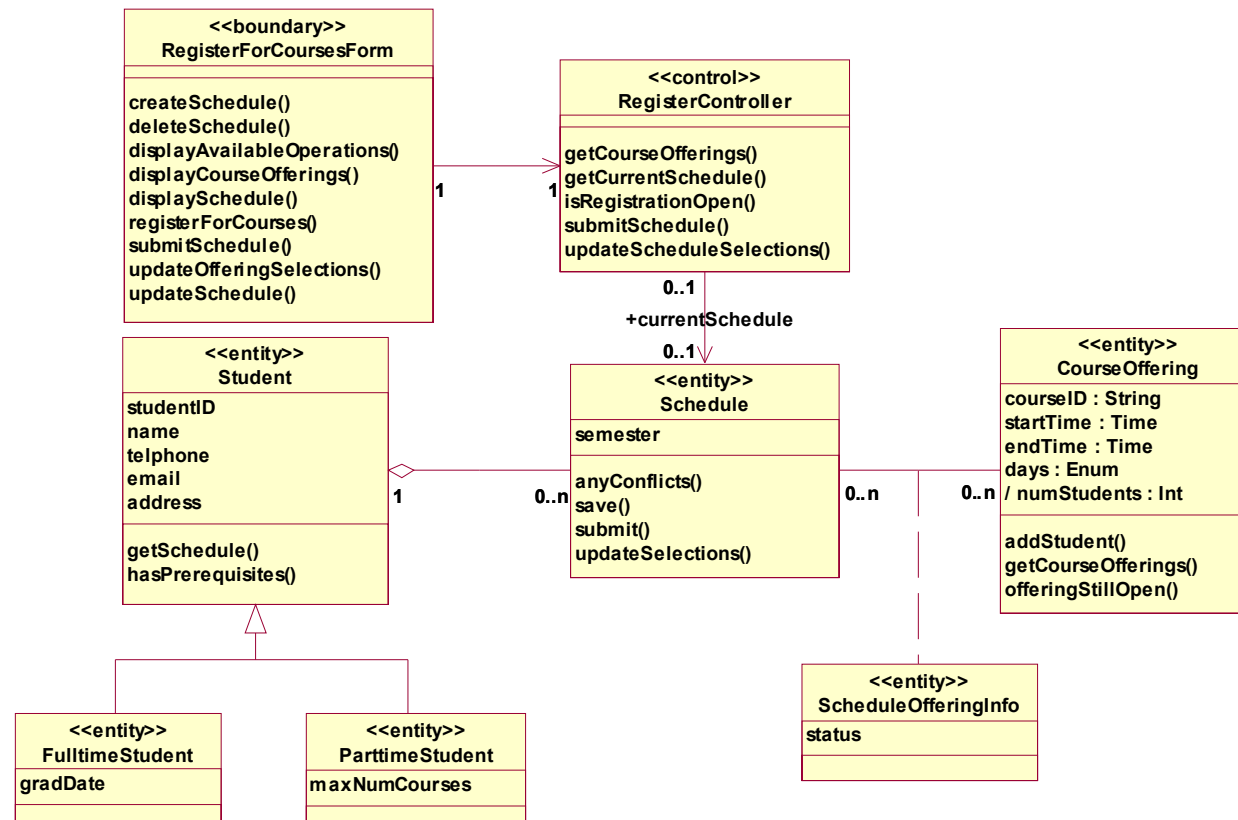
- 描述用户与系统如何交互。从用户角度描述系统功能， 是用户所能观察到的系统功能的模型图，用例是系统中的一个功能单元。



## 各UML图及特征

### ■ 2.类图(Class Diagram)/对象图(Object Diagram)

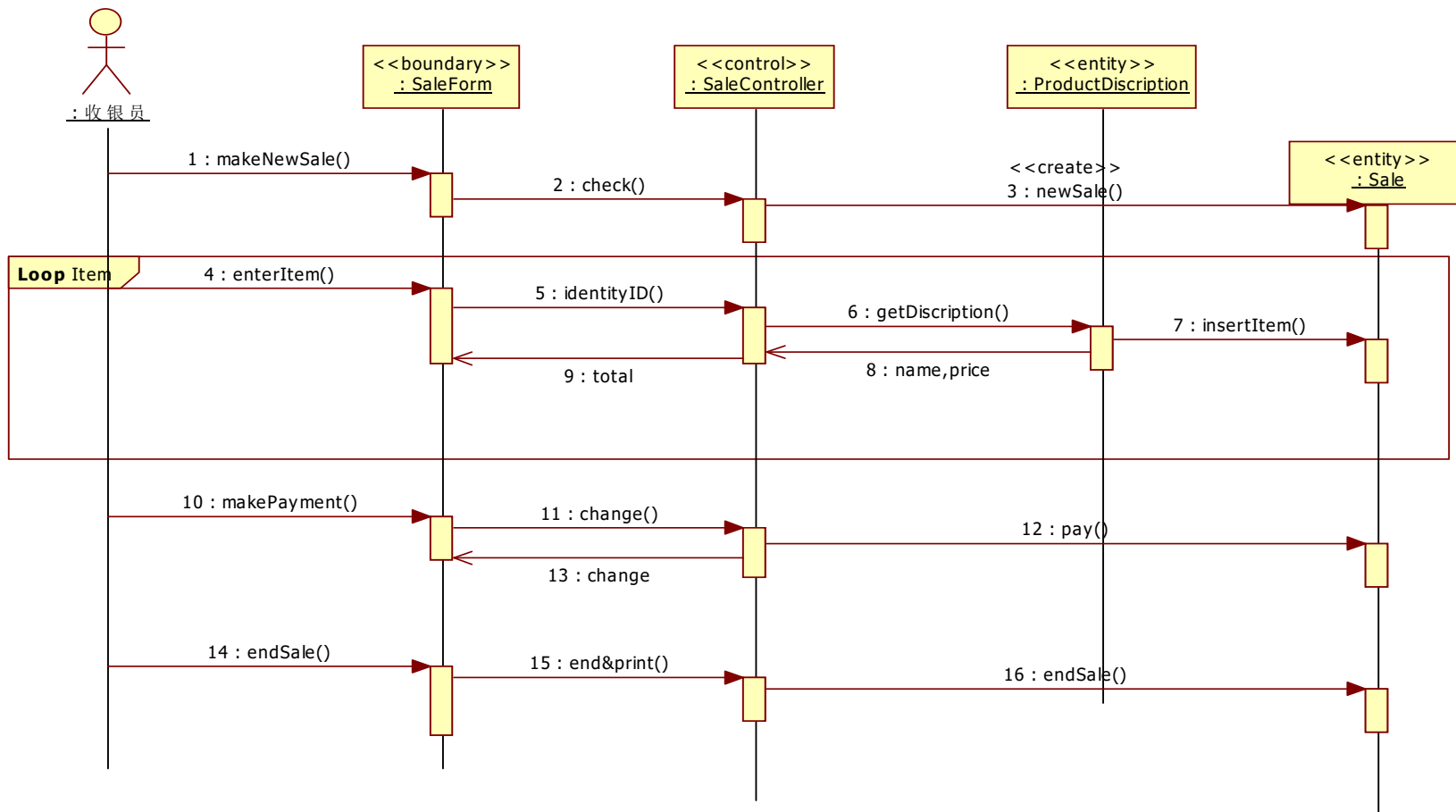
- 描述系统的各个对象类型以及存在的各种静态关系。对象图是类图的实例，几乎使用与类图完全相同的标识。它们的不同点在于对象图显示类的多个对象实例，而不是实际的类。



## 各UML图及特征

### ■ 3.顺序图(Sequence Diagram)

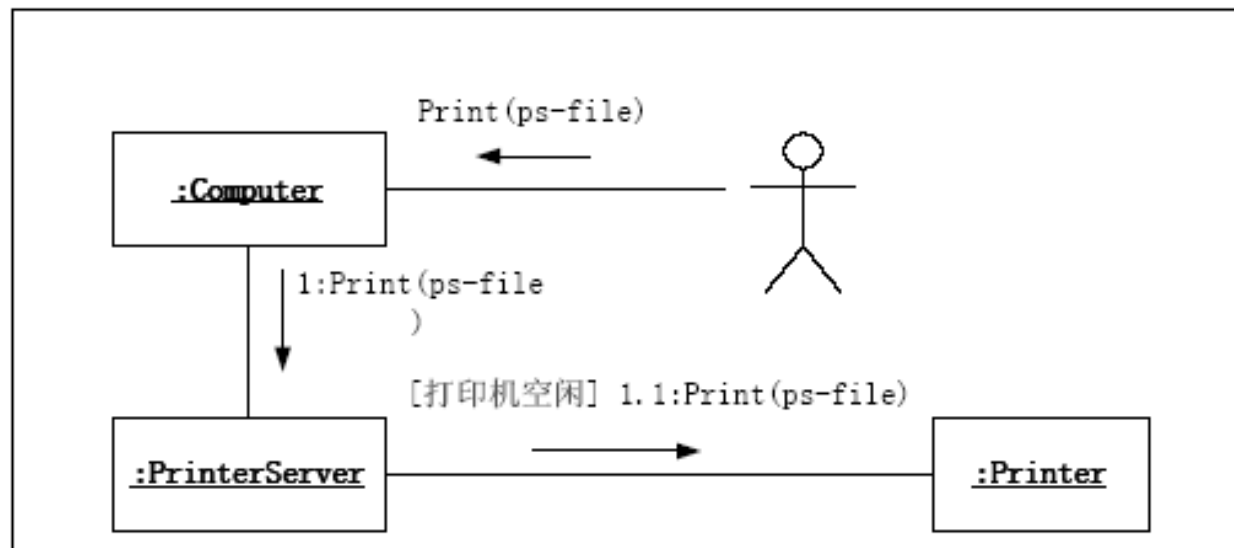
- 顺序图显示对象之间的动态合作关系，它强调对象之间消息发送的顺序，同时显示对象之间的交互。





## 各UML图及特征

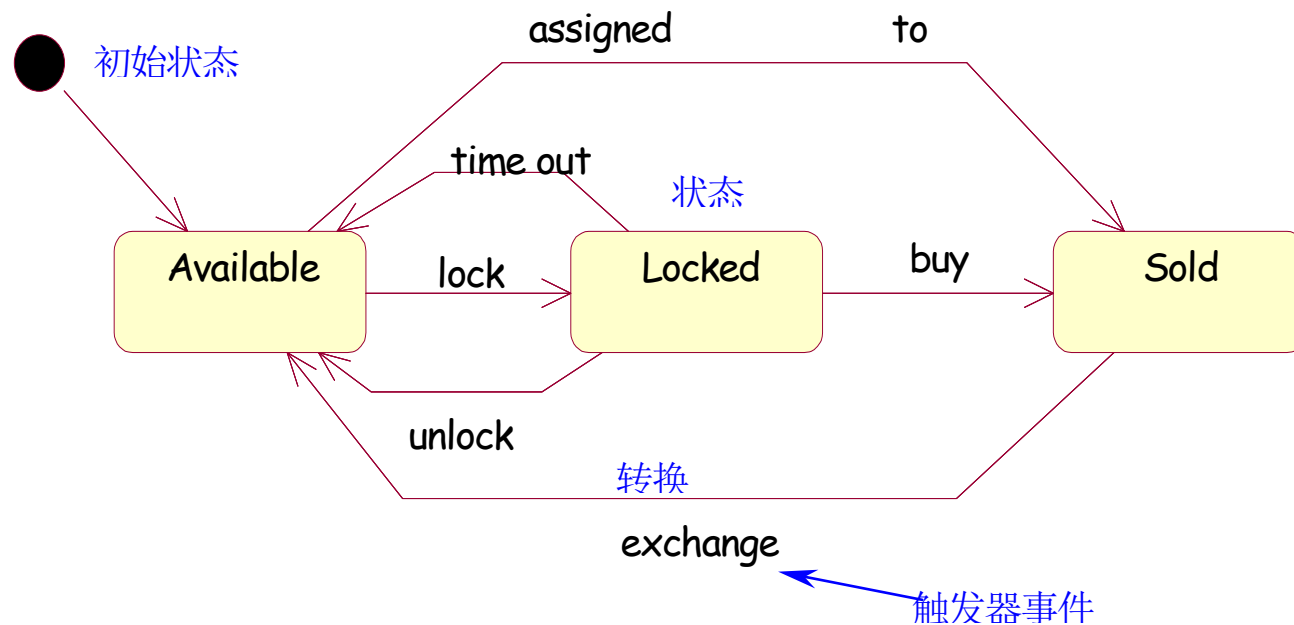
- 4.协作图(Collaboration Diagram)
  - 对象间交互，重点在对象间链接关系；
  - 协作图描述对象间的协作关系，协作图跟顺序图 相似，显示对象间的动态合作关系。除显示信息交换外，协作图还显示对象以及它们之间的关系。
  - 协作图的一个用途是表示一个类操作的实现



## 各UML图及特征

### ■ 5.状态图(State Chart Diagram)

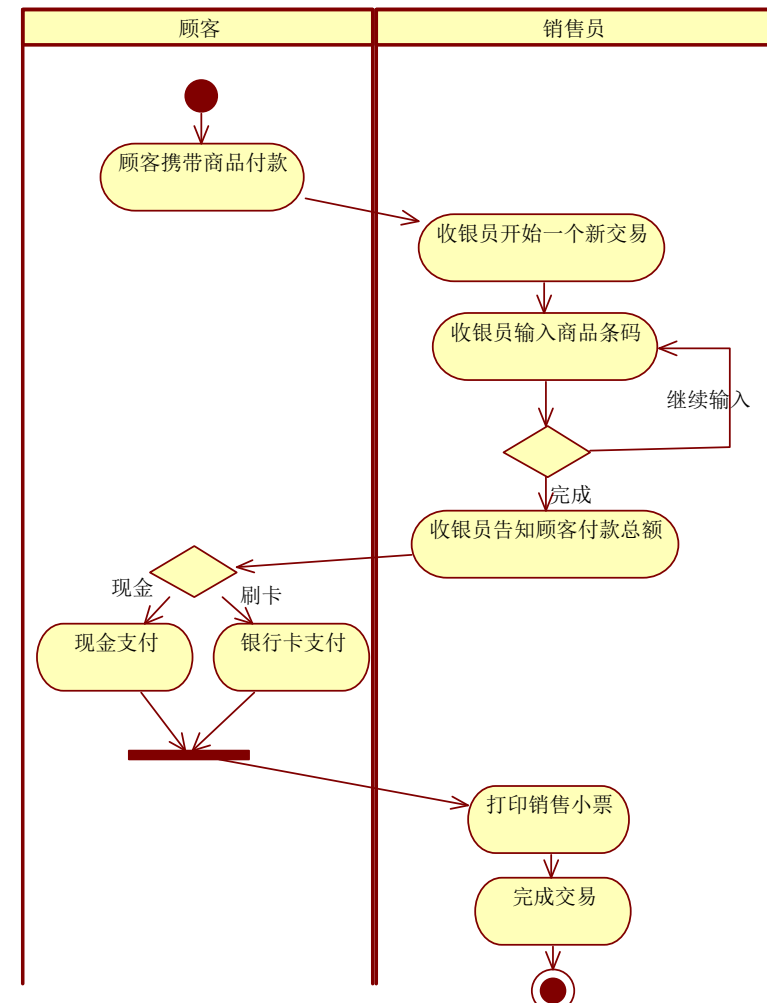
- 事件在对象的周期内如何改变状态;
- 状态图是一个类对象所可能经历的所有历程的模型图。状态图由对象的各个状态和连接这些状态的转换组成



## 各UML图及特征

### ■ 6.活动图(Activity Diagram)

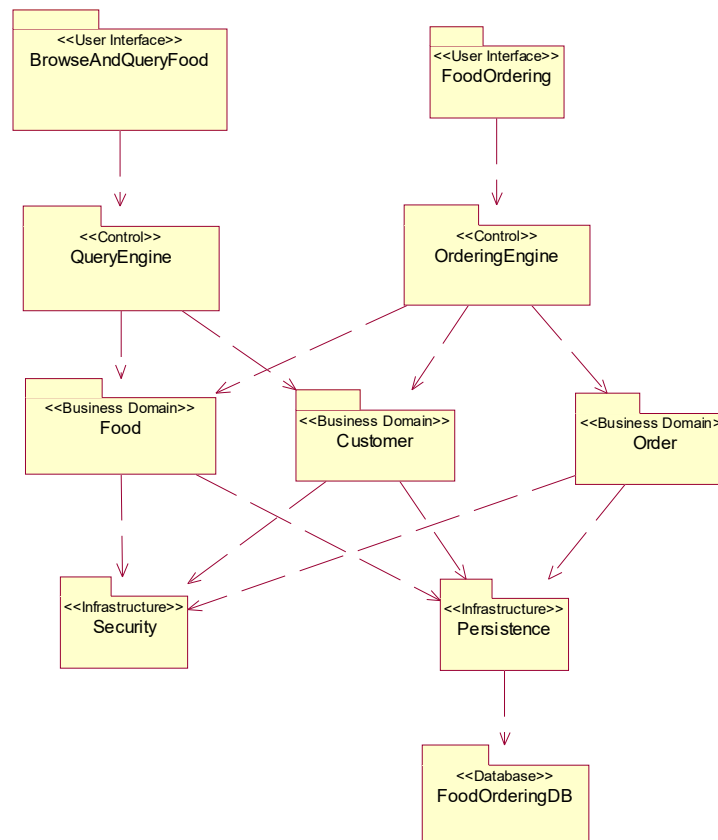
- 活动图是状态图的一个变体，用来描述执行算法的工作流程中涉及的活动
- 活动图描述了一组顺序的或并发的活动
- 过程及并行行为



## 各UML图及特征

### ■ 7.包图(Package Diagram)

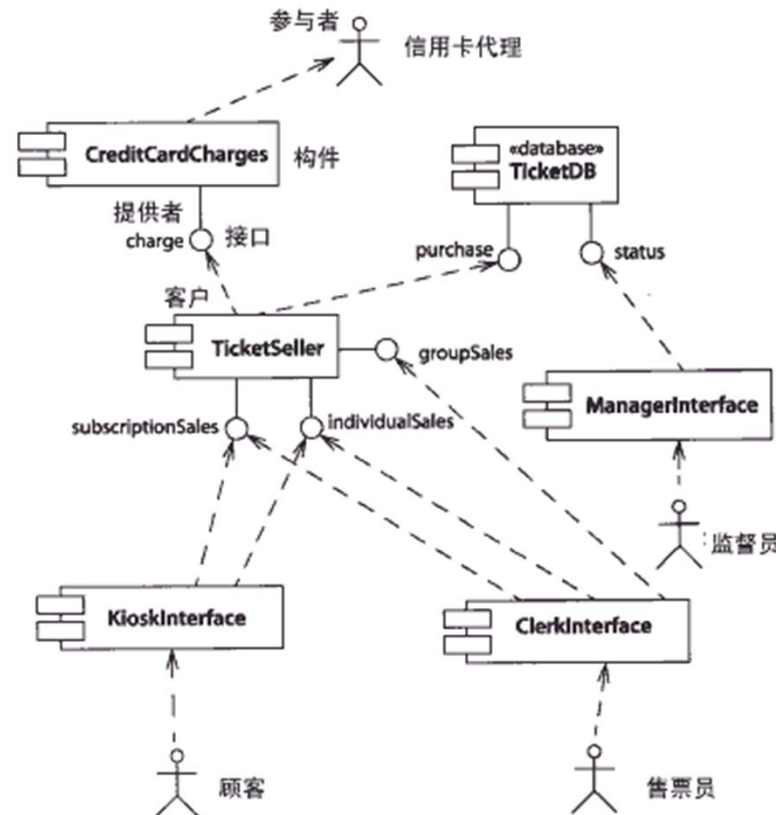
- 包图是在 UML 中用类似于文件夹的符号表示的模型元素的组合。
- 把在语义上接近且倾向于一起变化的类组织在一起形成“包”，既可控制模型的复杂度，有助于理解，而且也有助于按组来控制类的可见性；



## 各UML图及特征

### ■ 8. 构件图(Component Diagram)

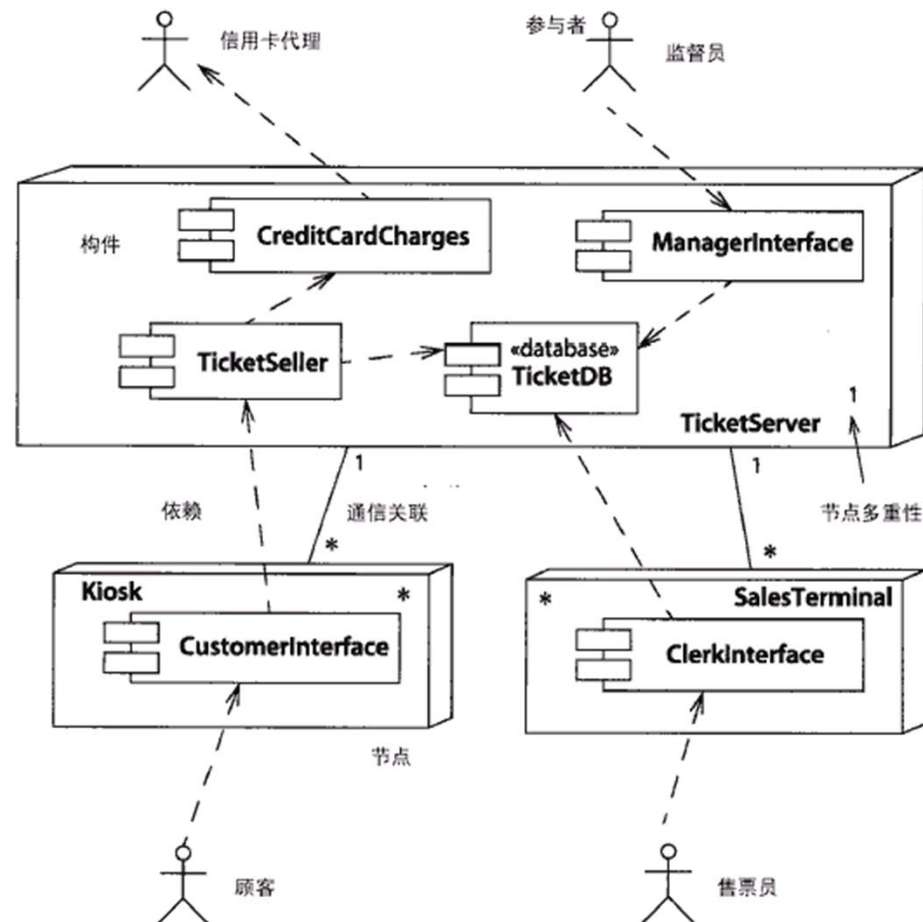
- 构件间的组织结构及链接关系
- 构件图为系统的构件建模—构件即构造应用的软件单元—还包括各构件之间的依赖关系，以便通过这些依赖关系来估计对系统构件的修改给系统可能带来的影响



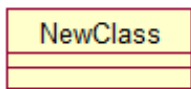




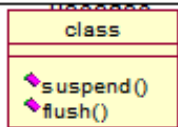
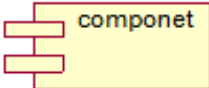

## 各UML图及特征

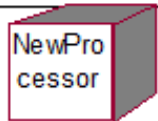

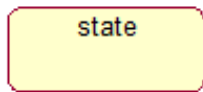
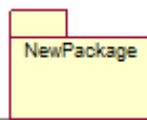





### ■ 9.部署图(Deployment Diagram)

- 部署视图描述位于节点实例上的运行构件实例的安排。节点是一组运行资源，如计算机、设备或存储器。这个视图允许评估分配结果和资源分配。



# UML 语法描述

|     |                                           |                                                                                     |
|-----|-------------------------------------------|-------------------------------------------------------------------------------------|
| 类   | 是对一组具有相同属性、相同操作、相同关系和相同语义的对象的描述           |    |
| 对象  |                                           |    |
| 接口  | 是描述了一个类或构件的一个服务的操作集                       |    |
| 协作  | 定义了一个交互，它是由一组共同工作以提供某种协作行为的角色和其他元素构成的一个群体 |    |
| 用例  | 是对一组动作序列的描述                               |    |
| 主动类 | 对象至少拥有一个进程或线程的类                           |  |
| 构件  | 是系统中物理的、可替代的部件                            |  |
| 参与者 | 在系统外部与系统直接交互的人或事物                         |  |

|      |                                |                                                                                       |
|------|--------------------------------|---------------------------------------------------------------------------------------|
| 节点   | 是在运行时存在的物理元素                   |    |
| 交互   | 它由在特定语境中共同完成一定任务的一组对象间交换的消息组成  |    |
| 状态机  | 它描述了一个对象或一个交互在生命期内响应事件所经历的状态序列 |    |
| 包    | 把元素组织成组的机制                     |    |
| 注释事物 | 是UML模型的解释部分                    |   |
| 依赖   | 一条可能有方向的虚线                     |  |
| 关联   | 一条实线，可能有方向                     |  |
| 泛化   | 一条带有空心箭头的实线                    |  |
| 实现   | 一条带有空心箭头的虚线                    |  |

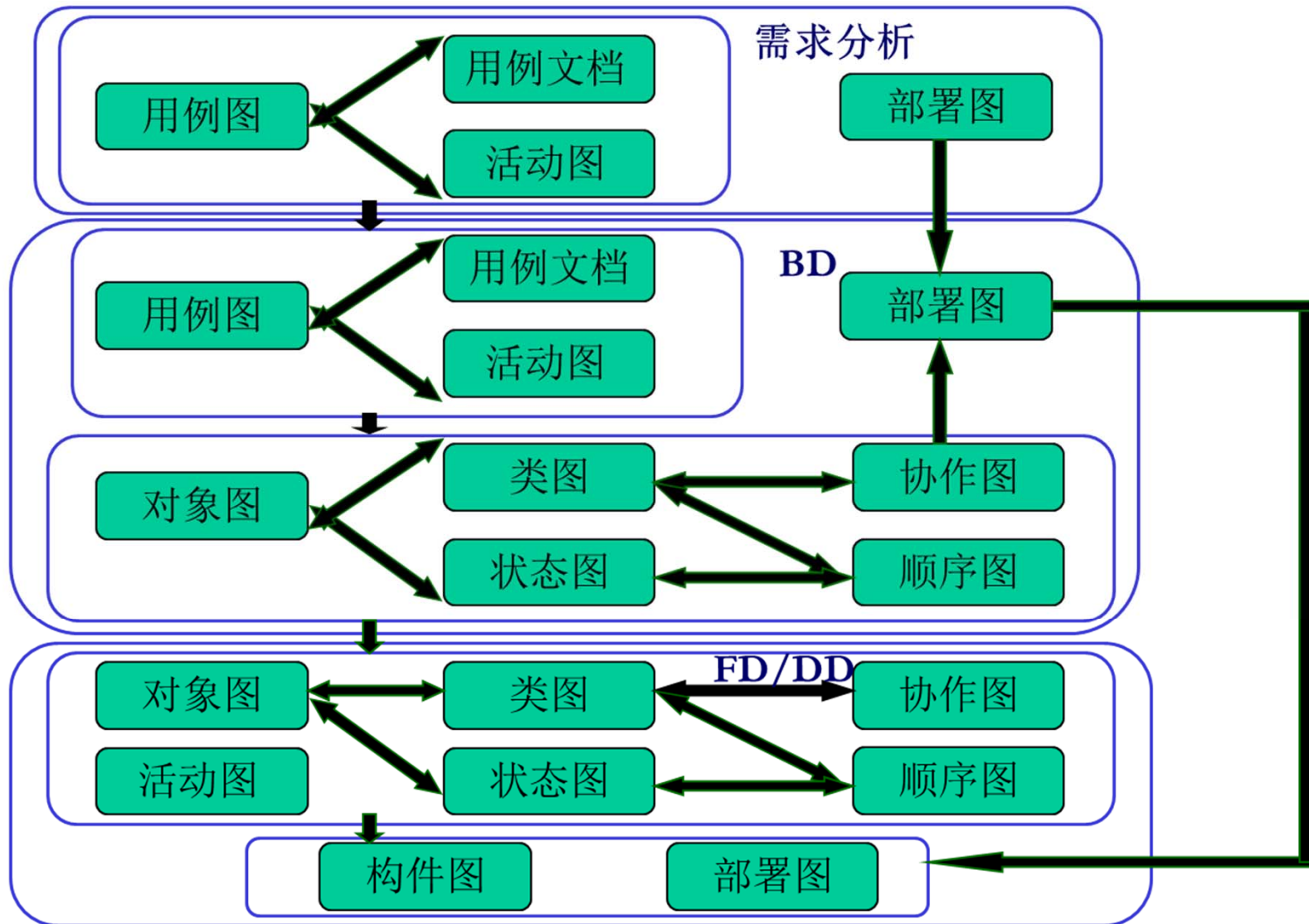
## UML图的适用场合

- 需求获取
  - 用例图：建立应用场景，如何使用系统
  - 活动图：明确组织机构的工作流程，软件如何与人交互
- 分析与设计
  - 从概念视角绘制的概念类图：用于领域分析
  - 软件视角的设计类图：设计软件中的类及相互联系
  - 常用用例的顺序图：重要用例的对象交互顺序
  - 构件图：构件设计及构件间关系
  - 包图：设计软件的组织结构
  - 具有复杂生命周期的类的状态图
- 实施
  - 部署图：描述软件系统在硬件平台和运行环境中的物理分布



# UML图的适用场合

全部图之间的关系



## 主流的UML建模工具

- 主流的**UML**建模工具
  - **IBM Rational Rose / IBM rational software architect(RSA)**
  - **Enterprise Architect**
  - **Visual Paradigm**
  - **Microsoft Visio**
  - **Sybase Powerdesigner**
  - **Together**
  - **StarUML**
  
  - **Eclipse**
  - **Microsoft Visual Studio**
  - **JBulider**



哈爾濱工業大學  
HARBIN INSTITUTE OF TECHNOLOGY

软件工程

結束

2017年10月25日