

研究生课程

软件工程方法论

第二部分：分析篇

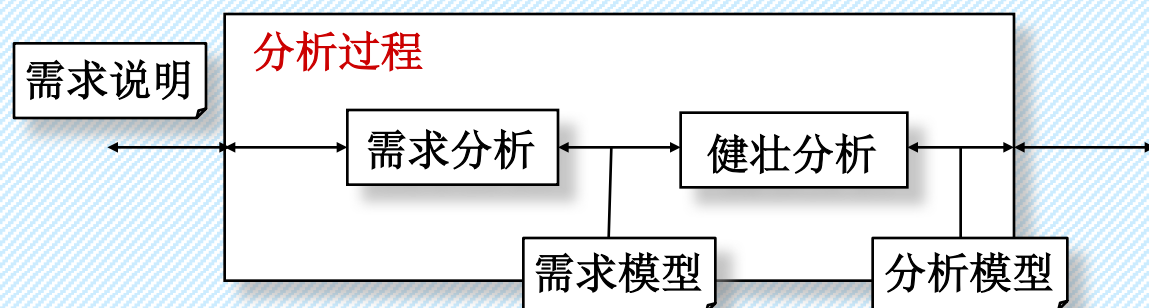
5.1 需求分析和系统分析

需求分析的确切含义是对用户需求进行分析，旨在产生一份明确、规范的需求定义。

OOA的主要内容是研究问题域中与需求有关的事物，把它们抽象为系统中的对象，建立类图。确切地讲，这些工作应该叫做**系统分析**，而不是严格意义上的需求分析。

早期的**OOA**缺乏一个良好的基础——对需求的规范描述。

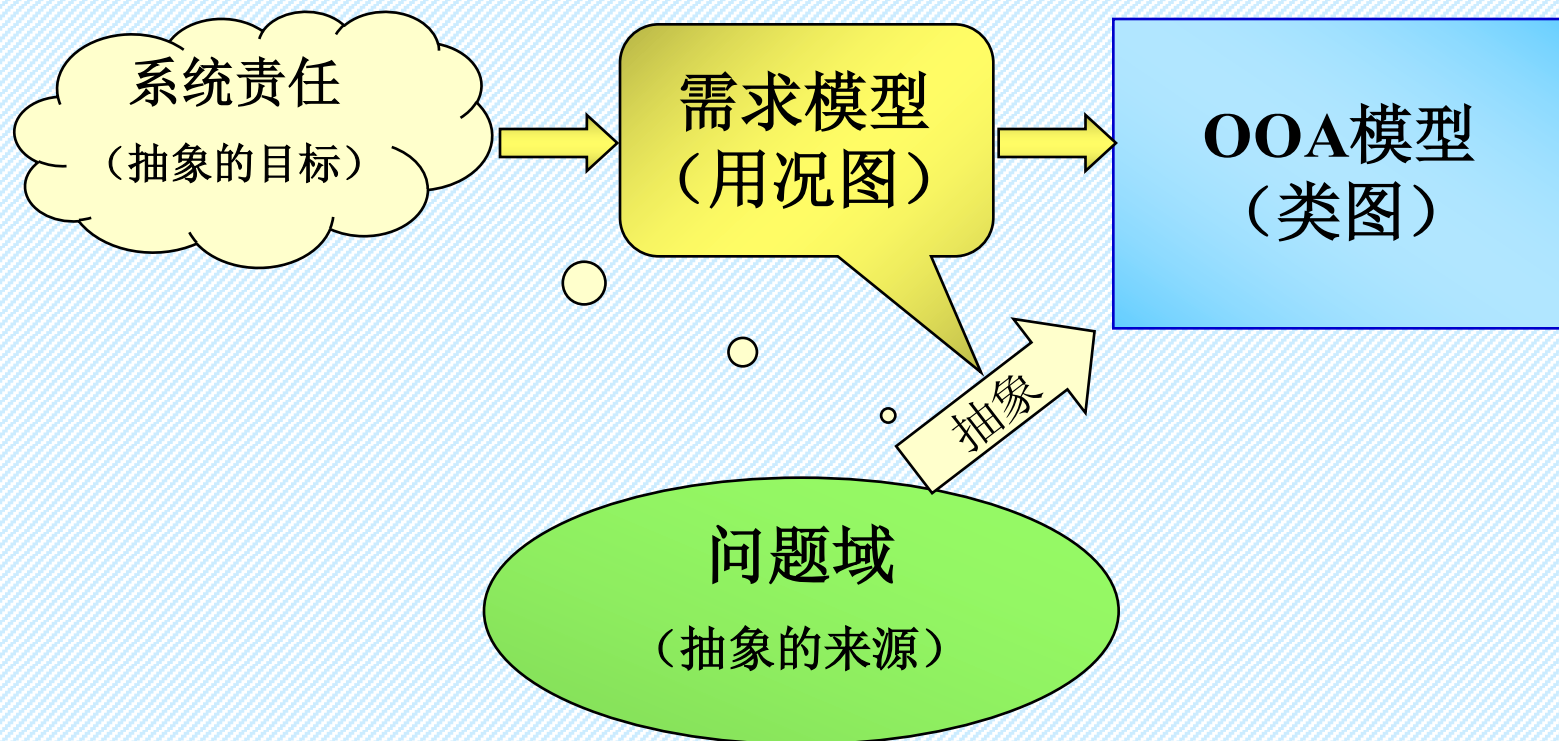
Jacobson方法（**OOSE**）提出用况（**use case**）概念，解决了对需求的描述问题，其分析过程如下：



OOA是将问题域中的事物抽象为系统中的对象

抽象的目标是系统责任——需求

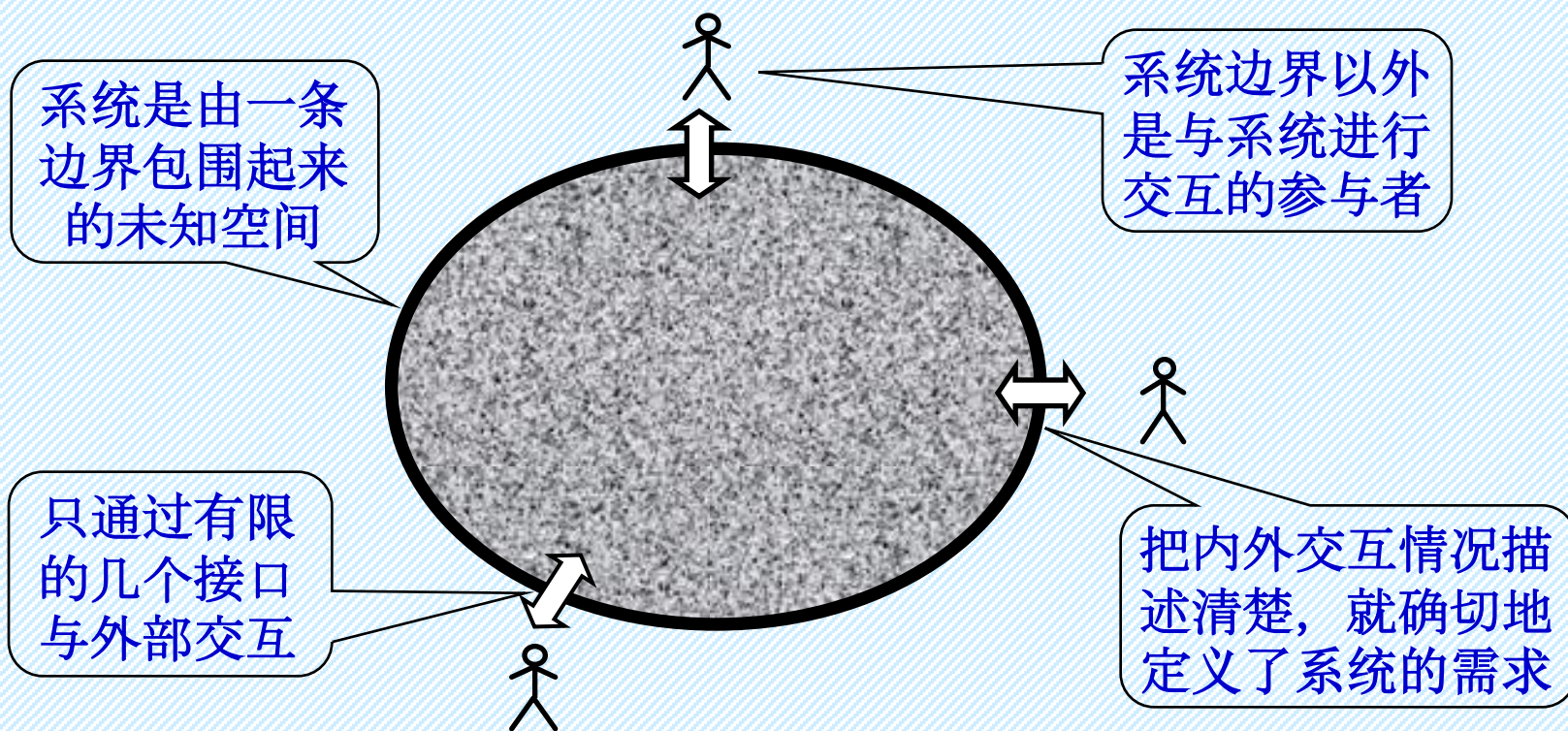
用况的概念解决了对需求的描述问题



5.2 基本思路

问题的提出：在系统尚未存在时，如何描绘用户需要一个什么样的系统？如何规范地定义用户需求？

考虑问题的思路：把系统看作一个黑箱，看它对外部的客观世界发挥什么作用，描述其**外部可见的行为**。



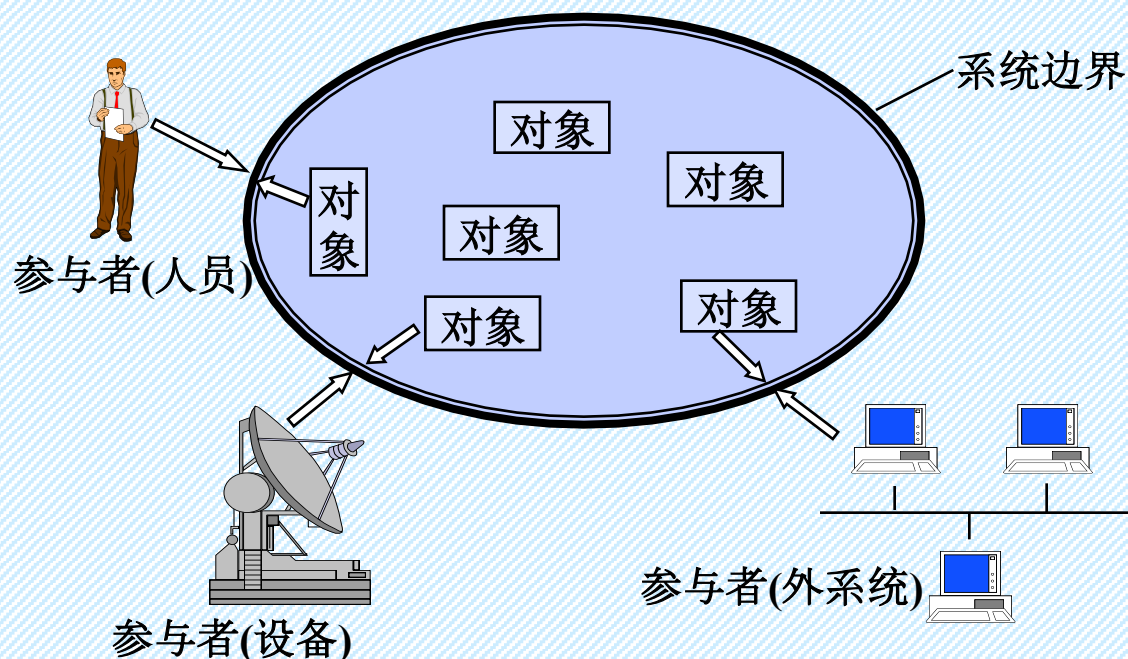
5.3 系统边界与参与者

系统边界：一个系统所包含的所有系统成分与系统以外各种事物的分界线。

系统：被开发的计算机软硬件系统，不是指现实系统。

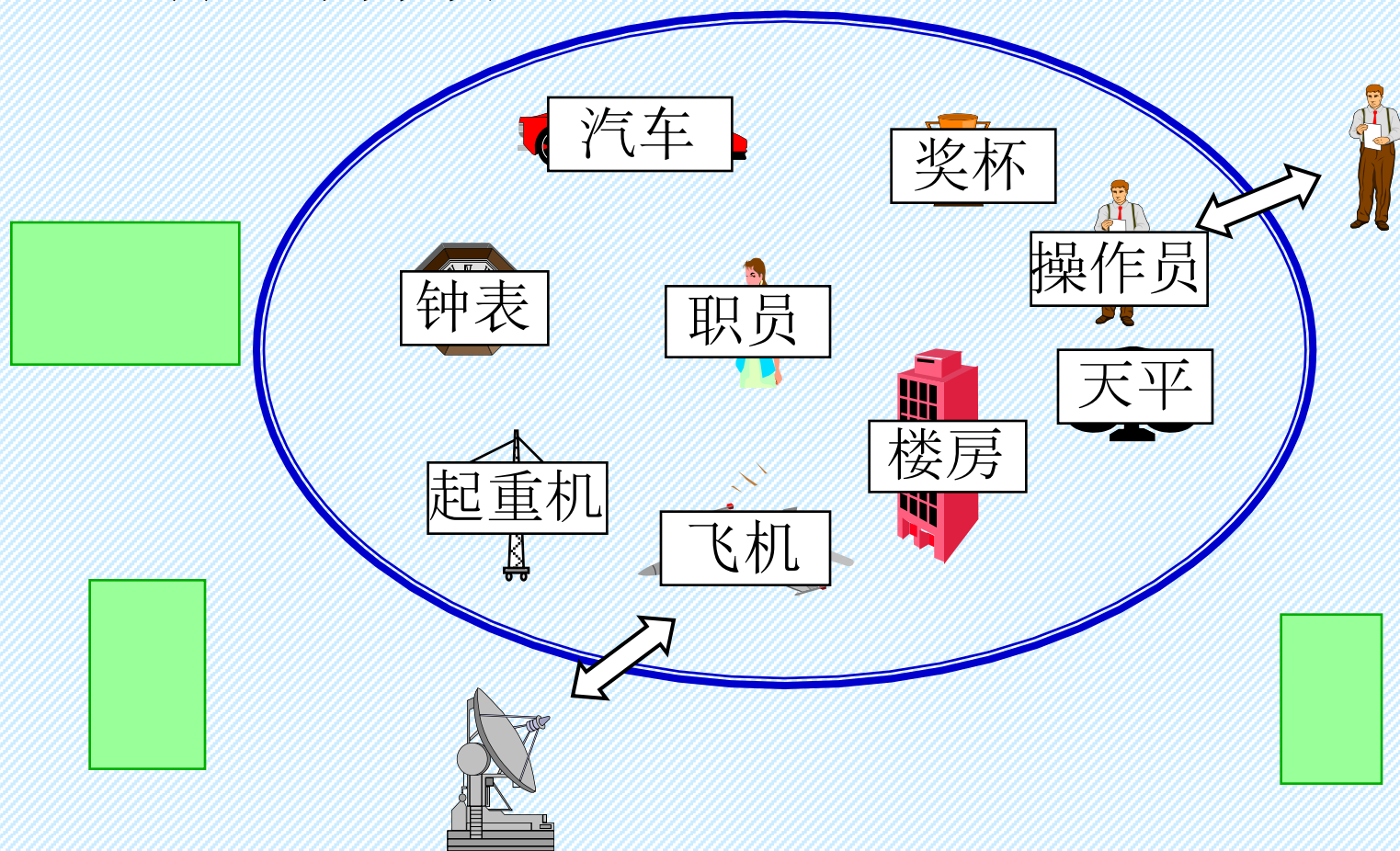
系统成分：在OOA和OOD中定义并且在编程时加以实现的系统元素——对象

参与者：在系统边界以外，与系统进行交互的事物——人员、设备、外系统



现实世界中的事物与系统之间的关系——分四种情况

- (1) 被抽象为系统中的对象
- (2) 只作为系统外部的参与者与系统交互
- (3) 既是系统中的对象，本身又作为参与者与系统交互
- (4) 与系统无关



如何发现参与者

——考虑人员、设备、外系统

人员——

系统的直接使用者

直接为系统服务的人员

设备——

与系统直接相联的设备

为系统提供信息

在系统控制下运行

不与系统相连的设备 ✕

计算机设备 ✕

外系统——

上级系统

子系统

其它系统

5.4 用况 (use case)

什么是用况

I. Jacobson:

用况是通过使用系统功能的某些部分而使用系统的一种具体方式。每个用况包括一个由参与者发动的完整的事件过程。它详细说明了参与者和系统之间发生的交互。因此，一个用况是一个由参与者和系统在一次对话中执行的特定的相关事务序列。全部用况的集合则说明了所有可能存在的系统使用方式。

《对象技术词典》:

1. 对一个系统或者一个应用的一种单一的使用方式所进行的描述。
2. 关于单个参与者在与系统的对话中所执行的处理的行为陈述序列。

UML:

对系统在与它的参与者交互时所能执行的一组动作序列（包括其变体）的描述。

本书的定义:

用况是对参与者使用系统的一项功能时所进行的交互过程的描述，其中包含由双方交替执行的一系列动作。

术语“use case”的准确含义——**使用情况**

是对一项系统功能使用情况的一般描述，它对于每一次使用都普遍适应，既不是应用实例，也不是举例说明。

——因此译为**“用况”**，而不是**“用例”**。

几点说明：

- (1) 一个用况只描述参与者对**单独一项**系统功能的使用情况；
- (2) 通常是平铺直叙的**文字**描述，UML也允许其他描述方式；
- (3) 陈述参与者和系统在交互过程中**双方**所做的事；
- (4) 所描述的交互既可能由**参与者发起**也可能由**系统发起**；
- (5) 描述彼此为对方**直接地**做什么事，不描述怎么做；
- (6) 描述应力求准确，允许概括，但**不要把双方的行为混在一起**；
- (7) 一个用况可以由**多种参与者**分别参与或共同参与。

内容与书写格式：

名称

行为陈述（分左右栏）

调用语句

控制语句

括号或标号

收款

输入开始本次收款的命令；

作好收款准备，应收款总数置为0，输出提示信息；

for 顾客选购的每种商品 do

输入商品编号；

if 此种商品多于一件 then

输入商品数量

end if;

检索商品名称及单价；

货架商品数减去售出数；

if 货架商品数低于下限 then

call 通知上货

end if;

计算本种商品总价并打印编号、

名称、数量、单价、总价；

总价累加到应收款总数；

end for;

打印应收款总数；

输入顾客付款数；

计算应找回款数，

打印付款数及找回款，

应收款数计入账册。

如何定义用况

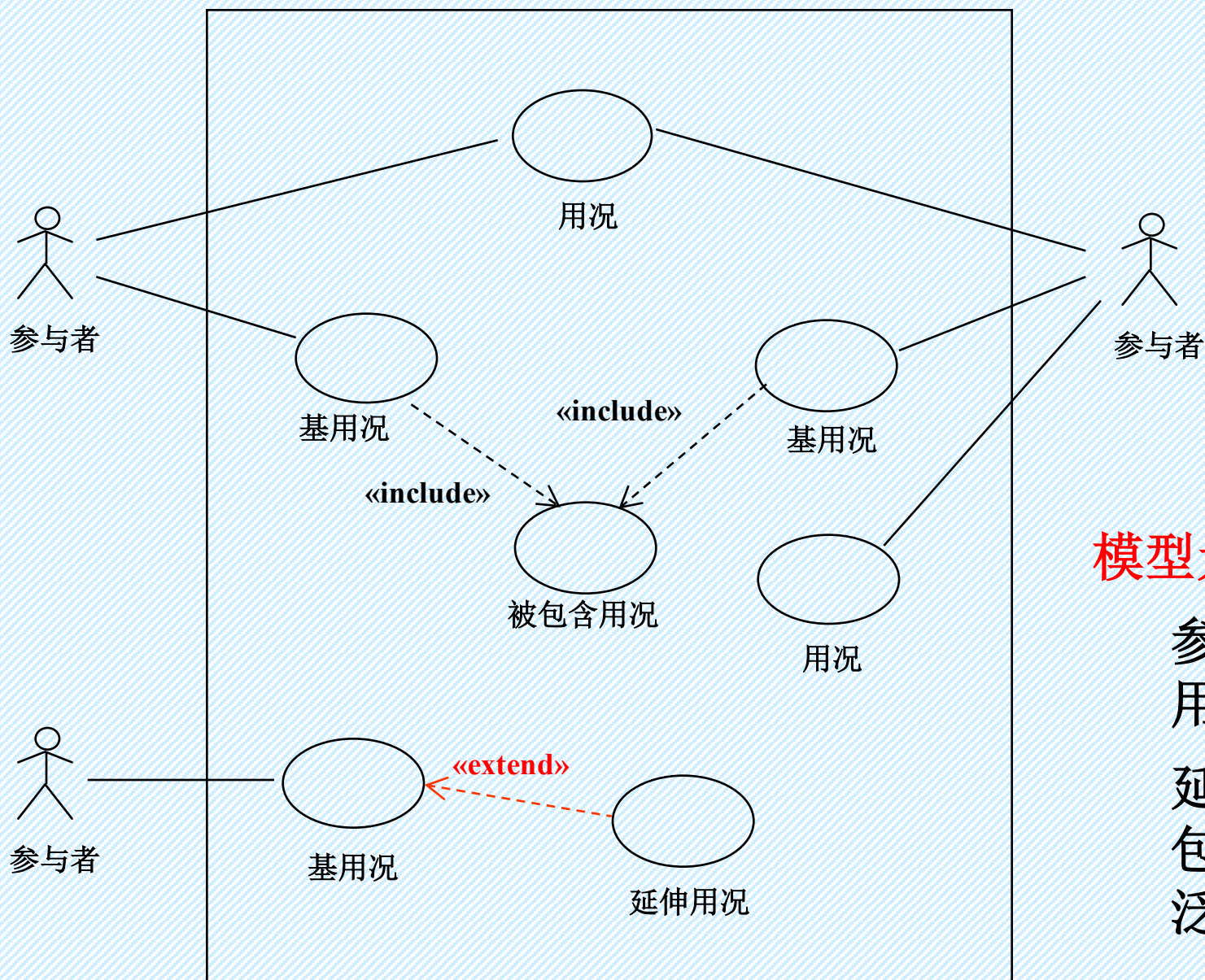
针对单个用况的描述策略：

把自己当作参与者，与设想中的系统进行交互。考虑：交互的目的是什么？需要向系统输入什么信息？希望由系统进行什么处理并从它得到何种结果？把上述交互过程描述出来。

定义系统中所有的用况：

- (1) 全面地了解和收集用户所要求的各项系统功能，找出所有的参与者，了解与各项功能相关的业务流程；
- (2) 把用户提出的功能组织成适当的单位，每一项功能完成一项完整而相对独立的工作；
- (3) 穷举每一类参与者所使用的每一项系统功能，定义相应的用况；
- (4) 检查用户对系统的各项功能需求是否都通过相应的用况做了描述。

5.5 用况图

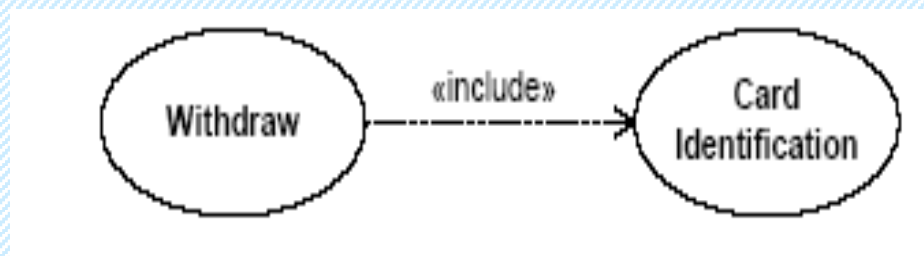


模型元素:

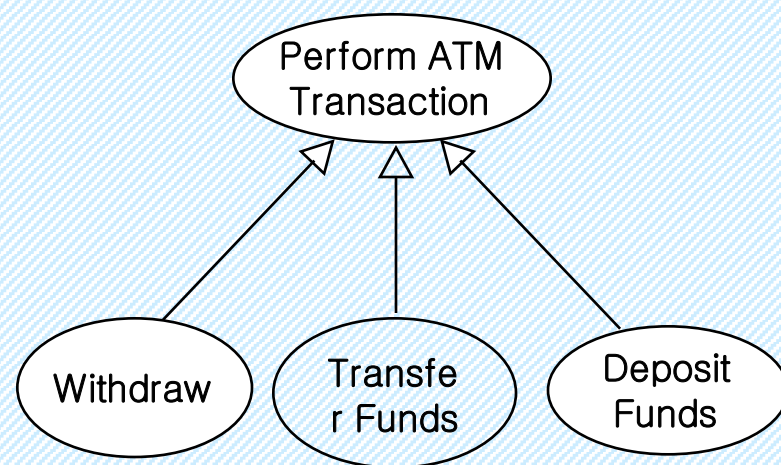
参与者
用况
延伸
包含
泛化

用例之间的关系

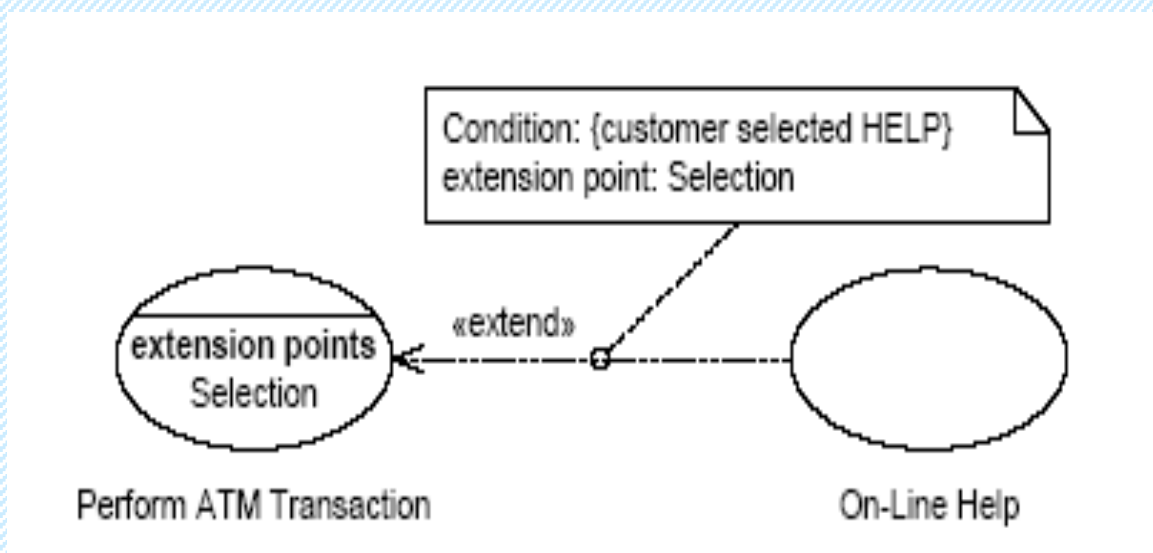
——包含、延伸、泛化



包含



泛化



延伸

问题:

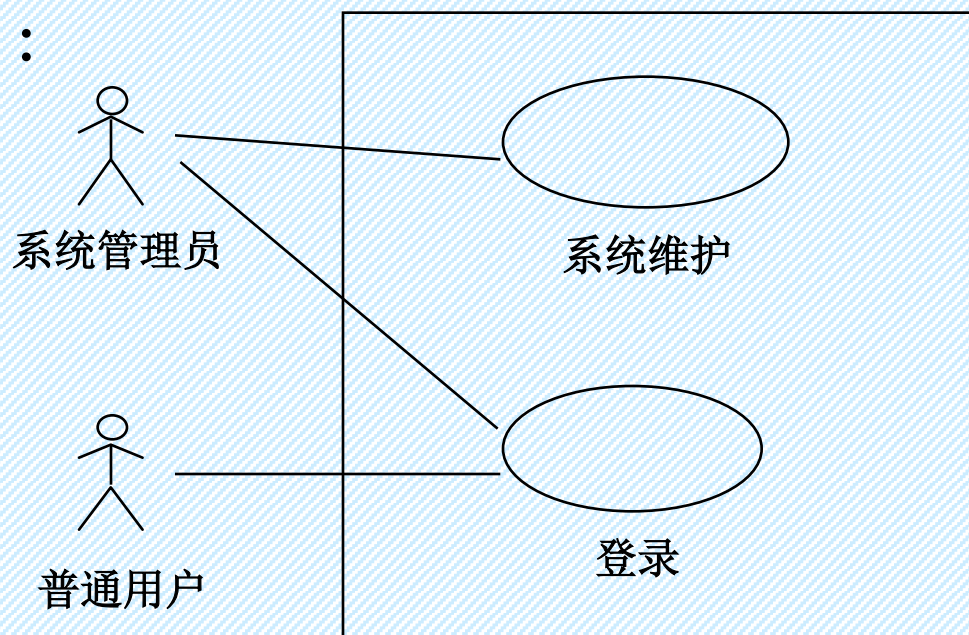
延伸与包含的相似性
延伸的方向问题
“条件”和“延伸点”问题
“泛化”问题
系统边界问题

用况的两种复杂情况

1、两个（或多个）参与者共享一个用况

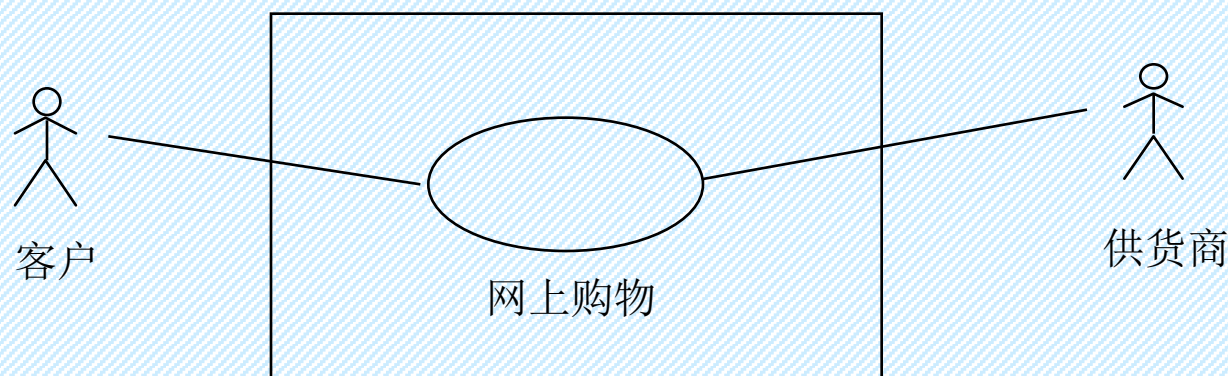
不同种类的参与者可能都要使用某一项系统功能，因此它们可能共享同一个用况

例：



2、一个用况的执行，可能需要两个（甚至多个）参与者同时与系统交互。

例：网上购物



5.6 开发过程与建议

用况图的开发过程

- 确定系统边界
- 发现参与者
- 定义用况
- 建立用况之间的关系
- 确定参与者和用况之间的关系
- 绘制用况图

使用用况图的几条建议

- 最重要的工作是对用况的描述
- 不要过分深入地描述系统内部的行为细节
- 运用最主要概念，加强用况内容的描述
 - 不要陷入延伸与包含、延伸点、泛化等问题的争论和辨别
- 了解用况的局限性——主要作用是描述功能需求

6.1 对象和类的概念及其运用

概念：

对象（object）是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位，由一组属性和施加于这组属性的一组操作构成。

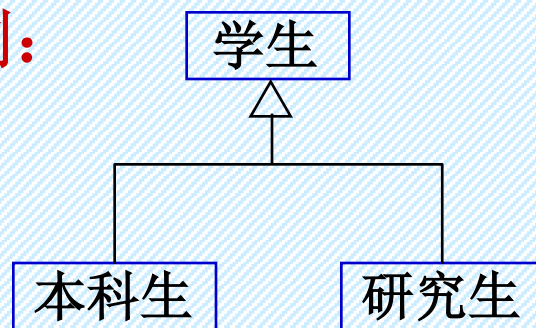
类（class）是具有相同属性和操作的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，它由一个类名、一组属性和一组操作构成。

类和对象的关系——集合与成员，对象是类的实例

在一般-特殊结构中，特殊类的对象实例在逻辑上也都是其一般类的对象实例。

不直接创建对象实例的类称为**抽象类（abstract class）**

例：



主动对象（active object）——至少有一个操作不需要接收消息就能主动执行的对象
用于描述具有主动行为的事物

主动对象的类叫做**主动类（active class）**

被动对象（passive object）——每个操作都必须在消息的驱动下才能执行的对象

类的语义

OO方法中的类在不同的语境下有两种不同的语义：

1. 一个类代表由它的全部对象实例所构成的群体
日常语言表达中的例子：

“公司里有管理人员、技术人员和市场人员”

“马路上汽车很多”

在OO模型中：

每个类都是由它的全部对象实例所构成的集合
类代表了它的全部对象实例。

2. 一个类代表属于该类的任意一个对象实例
从大量的个体中抽象出一个概念，再运用这个概念时就可以代表其中的任何一个个体，例如：

“学生有一个学号，属于一个班级，要上课”

在OO系统模型中定义了一个类，它就可以代表它的任何一个对象实例，例如：

汽车与发动机之间的聚合关系，表示任何一辆汽车都有一台发动机，任何一台发动机都可以装在0—1辆汽车上

在类的抽象层次建模

理由：

- (1) 充分性：模型中一个类描述了它的全部对象实例
- (2) 必要性：个别对象实例不能代表其他对象实例
- (3) 符合人类的思维方式：在概念层次上表达描述事物规律
- (4) 与**OOPL**保持良好的对应
- (5) 避免建模概念复杂化
- (6) 消除抽象层次的混乱

如何运用类和对象的概念

从对象出发认识问题域
将问题域中的事物抽象为对象；

将具有共同特征的对象抽象为类
用类以及它们之间的关系构成整个系统模型；

归纳

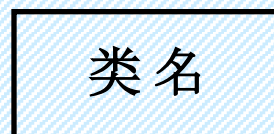
在模型中用类表示属于该类的任何对象
在类的规约中说明这个类将创建那些对象实例

在程序中用类定义它的全部对象
编程时静态声明类的对象
运行时动态创建类的对象

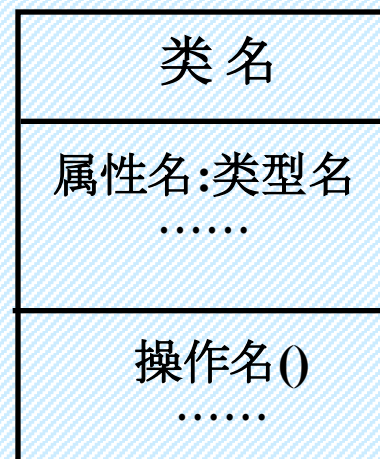
演绎

6.2 表示法

在模型中用类符号来表示一个类
它代表了属于该类的全部对象实例

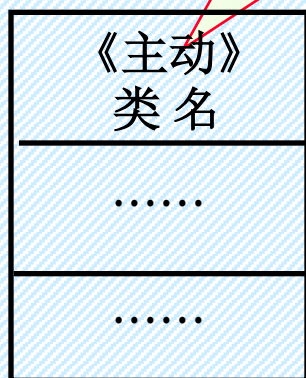


压缩方式

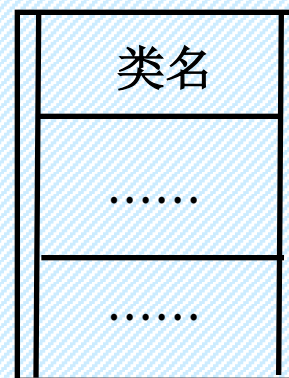


展开方式

衍型 (stereotype)
用关键字或者
用图标表示



主动类



UML2 主动类表示法

UML的对象表示法:

匿名对象

对象名:类名

压缩方式

对象名:类名

属性名=值
.....

细节方式

:类名

压缩方式

:类名

属性名=值
.....

细节方式

用所谓“匿名对象”代表类的任何一个对象实例，然而这恰恰是类的作用。

6.3 发现对象

研究问题域

亲临现场深入调查研究

直接观察并向用户及相关的业务人员进行调查和交流，考察问题域中各种各样的事物、它们的特征及相互关系

听取问题域专家的见解

领域专家——包括技术人员、管理者、老职员和富有经验的工人等

阅读相关材料

阅读各种与问题域有关的材料，学习相关行业和领域的基本知识

借鉴以往的系统

查阅以往在该问题域中开发过的同类系统的分析文档，吸取经验，发现可以复用的类

正确地运用抽象原则

对什么进行抽象——问题域
当前目标——系统责任

忽略与系统责任无关的事物

只注意与之有关的事物，抽象为系统中的对象

例如：学校的教师、学生、教务员 和 警卫

忽略与系统责任无关的事物特征

只注意与之有关的特征，抽象为对象的属性或操作

例如：教师的专业、职称 和 身高、体重

正确地提炼对象

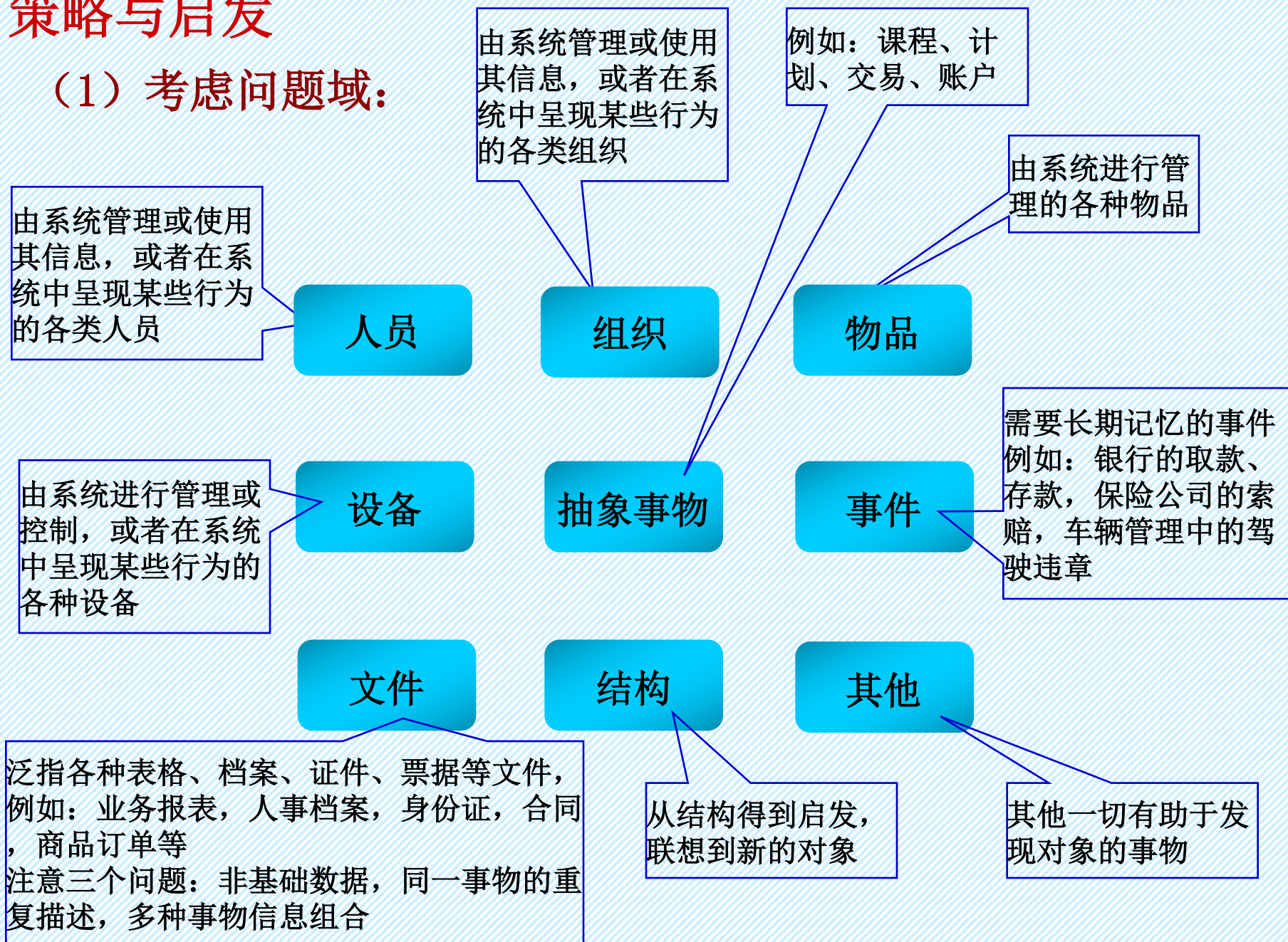
例如：对书的不同抽象

在图书馆管理系统中以一本书作为一个对象实例

在书店管理系统中以一种书作为一个对象实例

策略与启发

(1) 考虑问题域:



(2) 考虑系统边界:

考察在系统边界以外与系统交互的各类参与者
考虑通过那些对象处理这些参与者的交互

人员

设备

外系统

(3) 考虑系统责任:

检查每一项功能需求是否已有相应的对象提供,
发现遗漏的对象

审查与筛选

(1) 舍弃无用的对象

通过属性判断：

是否通过属性记录了某些有用的信息？

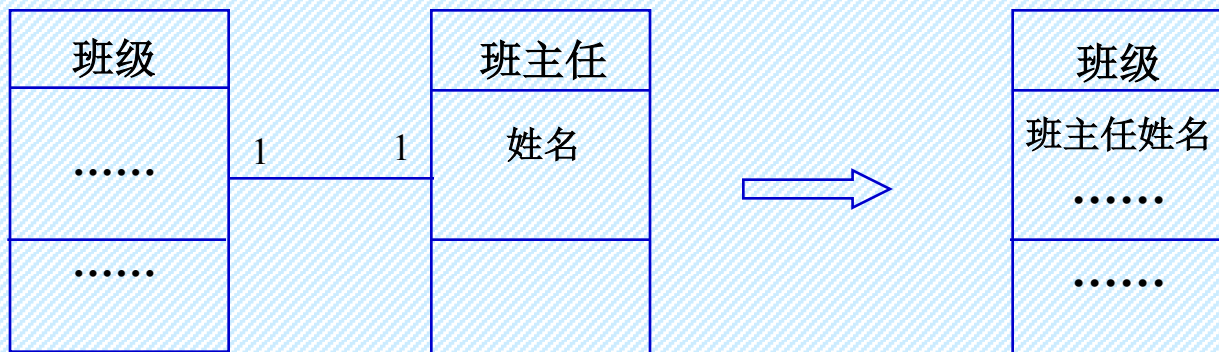
通过操作判断：

是否通过操作提供了某些有用的功能？

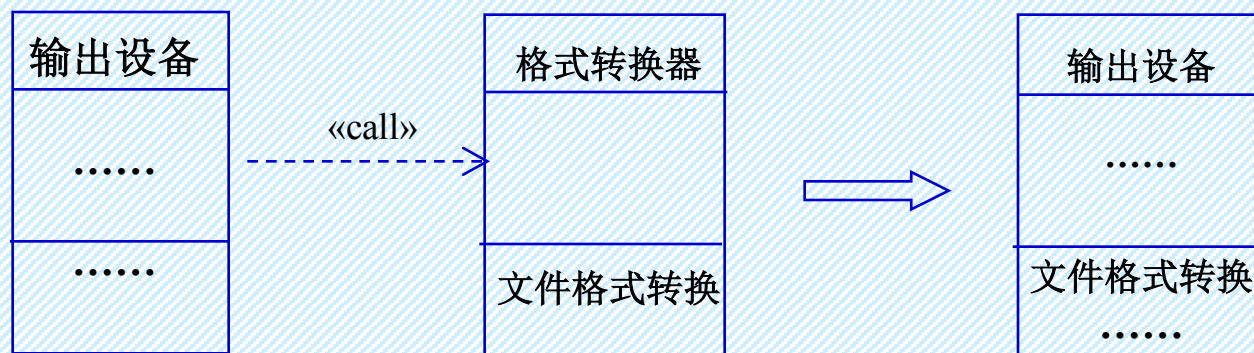
二者都不是——无用

(2) 对象的精简

只有一个属性的对象



只有一个操作的对象



(3) 与实现条件有关的对象

例如：与

图形用户界面（**GUI**）

数据管理系统

硬件 及

操作系统 有关的对象

——推迟到**OOD**考虑

6.4 对象分类

(1) 将对象抽象为类，用类表示它的全部对象

(2) 审查和调整

类的属性或操作不适合该类的全部对象实例

例：“汽车”类的“乘客限量”属性

——进一步划分特殊类

属性及操作相同的类

经过抽象，差别很大的事物可能只保留相同的特征

——考虑能否合并为一个类

属性及操作相似的类

——考虑能否提升出一个一般类

同一事物的重复描述

例：“职员”和“工作证”

——取消其中一个

(3) 类的命名

类的名字应适合该类（及其特殊类）的全部对象实例

反映个体而不是群体

使用名词 或 带定语的名词

避免市井俚语和无意义的符号

使用问题域通用的词汇

使用便于交流的语言文字

可以用本地文字和英文双重命名

7.1 属性和操作

属性 (attribute) 是用来描述对象静态特征的一个数据项。

实例属性 (instance attribute) 和**类属性** (class attribute) 的区别

例如：仪表类

输入电压、功率及各种规定的质量指标——类属性

编号、出厂日期、精度等实际性能参数——实例属性

操作（operation）是用来描述对象动态特征（行为）的一个动作序列。

近义词：方法（method），服务（service）

被动操作（passive operation）：

只有接收到消息才能执行的操作

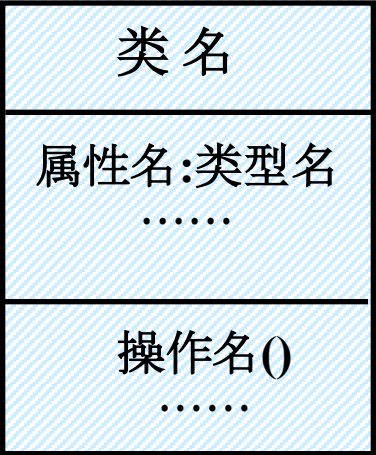
编程语言中的函数、过程等被动成分

主动操作（active operation）：

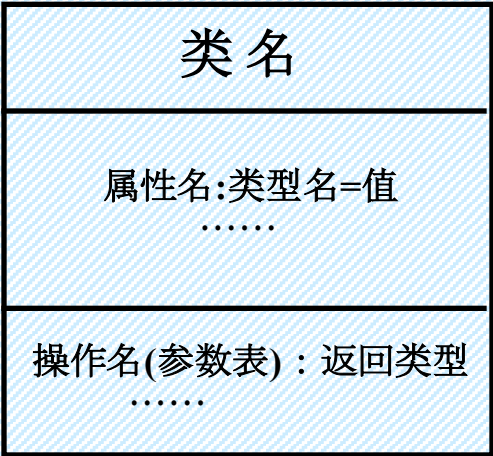
不需要接收消息就能主动执行的操作

编程语言中的进程、线程等主动成分

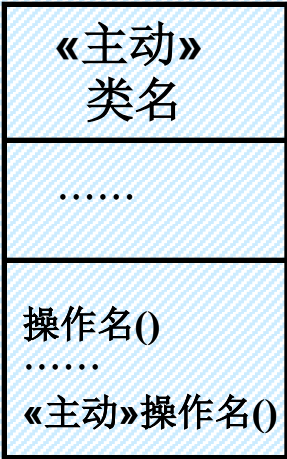
7.2 属性和操作的表示法



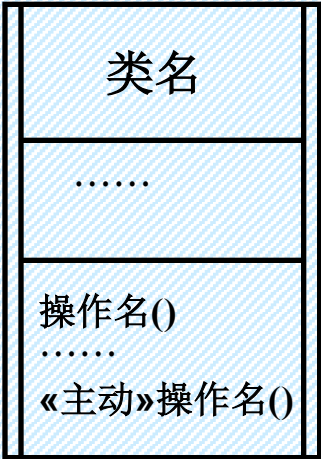
分析级细节方式



实现级细节方式



用衍型表示主动操作



7.3 定义属性

(1) 策略与启发

按常识这个对象应该有哪些属性？

人→姓名、地址、出生年月

在当前的问题域中，对象应该有哪些属性？

商品→条形码

根据系统责任，这个对象应具有哪些属性？

乘客→手机号码

建立这个对象是为了保存和管理哪些信息？

物资→型号、规格、库存量

为实现操作的功能，需要增设哪些属性？

传感器（信号采集功能）→时间间隔

是否需要增加描述对象状态的属性？

设备→状态

用什么属性表示关联和聚合？

课程→任课教师，汽车→发动机

(2) 审查与筛选

是否体现了以系统责任为目标的抽象

例：书→重量？

是否描述对象本身的特征

例：课程→电话号码？

是否可通过继承得到？

是否可从其他属性直接导出？

(3) 推迟到OOD考虑的问题

规范化问题

对象标识

性能问题

(4) 属性的命名与定位

命名：原则与类的命名相同

定位：针对所描述的对象

适合类（及其子类）的全部对象实例

7.4 定义操作

(1) 对象行为分类

系统行为

例：创建、删除、复制、转存

对象自身的行为——算法简单的操作

例：读、写属性值

对象自身的行为——算法复杂的操作

计算或监控

(3) 策略与启发

考虑系统责任

有哪些功能要求在本对象提供？

考虑问题域

对象在问题域对应的事物有哪些行为？

分析对象状态

对象状态的转换是由哪些操作引起的？

追踪操作的执行路线

模拟操作的执行，并在整个系统中跟踪

(3) 审查与调整

审查对象的每个操作是否**真正有用**

是否直接提供系统责任所要求的某项功能？

或者 响应其它操作的请求间接地完成这种功能的某些局部操作？

调整——取消无用的操作

审查操作是不是**高内聚**的

一个操作应该只完成一项单一的、完整的功能

调整——拆分 或 合并

(4) 认识对象的主动行为

考虑问题域

对象行为是被引发的，
还是主动呈现的？

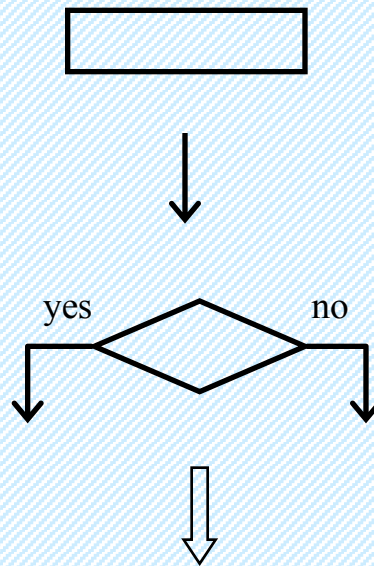
与参与者直接交互的对象操作

操作执行路线逆向追踪

(5) 操作过程描述

——可采用流程图或活动图

流程图：



动作陈述框，在框内填写要执行的动作。

转接，用于连接各个框，表示它们之间的转接关系。

条件判断框，给出一个判断条件。

入口/出口标记，指出操作的开始或结束。

活动图：在流程图基础上进行了一些扩展，有更强的描述能力

问题：分析阶段为什么要给出操作流程？
关于OOA/OOD分工的两种不同观点

(6) 操作的命名和定位

命名：动词或动宾结构

定位：

与实际事物一致

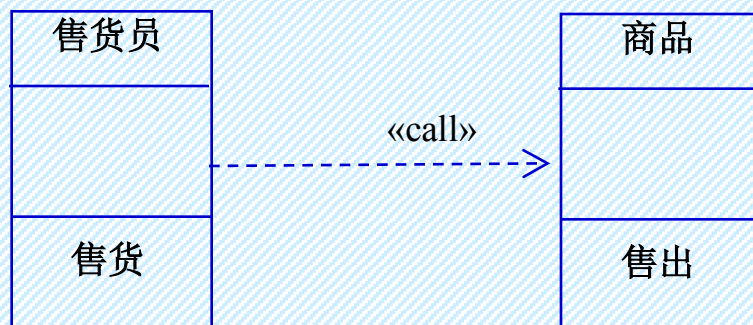
例：售货员——售货，商品——售出

在一般-特殊结构中的位置

——适合类的全部对象实例

从主语-谓语-宾语结构看对象操作的设置

“售货员销售商品”——操作应该放在哪里？



7.5 接口的概念及用途

早期的面向对象方法并没有把接口作为正式的OO概念和系统成分，只是用来解释OO概念

“操作是对象（类）对外提供的访问接口”

20世纪90年代中后期，接口才作为一种系统成分出现在OOPL中，并且被UML作为一种模型元素

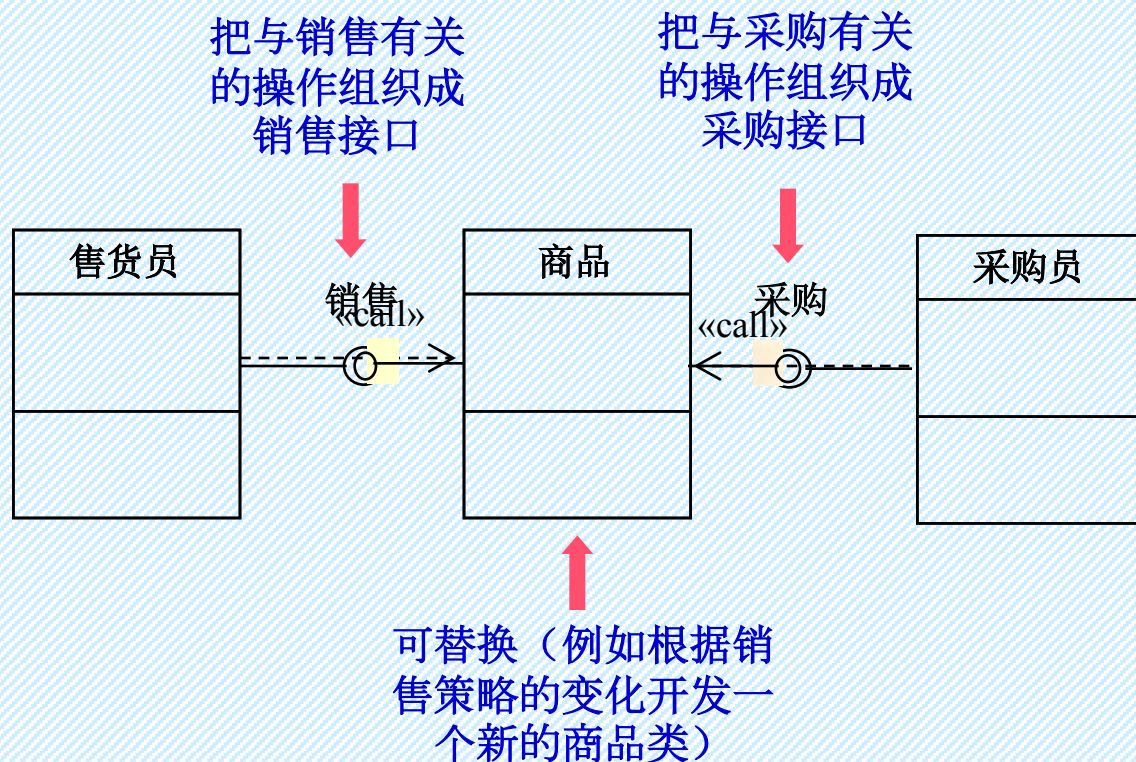
UML对接口的定义及解释：

“接口（**interface**）是一种类目（**classifier**），它表示对一组紧凑的公共特征和职责的声明。一个接口说明了一个合约；实现接口的任何类目的实例必须履行这个合约。”

“一个给定的类目可以实现多个接口，而一个接口可以由多个不同的类目来实现。”

为什么引入接口的概念

针对不同的应用场合组织对象的操作



接口提供了更灵活的衔接机制

接口 (interface)

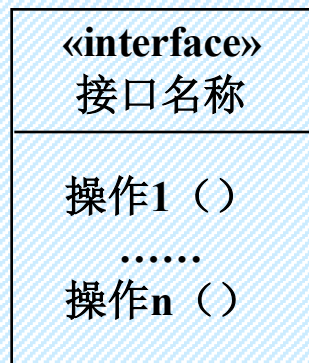
是由一组操作所形成的一个集合，它由一个名字和代表其中每个操作的特征标记构成。

特征标记 (signature)

代表了一个操作，但并不具体地定义操作的实现

特征标记 ::= <操作名> ([<参数>:<类型>] {,<参数>:<类型>}) [:<返回类型>]

表示法 (详细方式) :

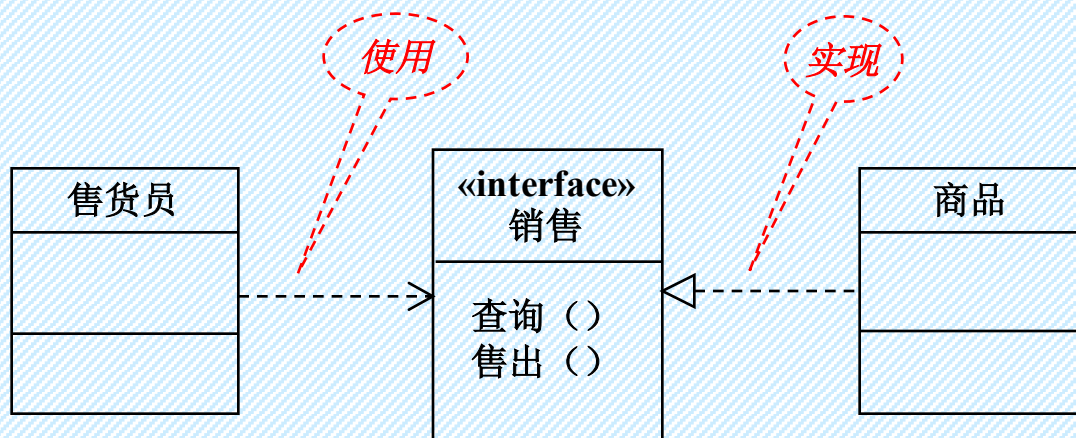


接口与类的关系

接口由某些类实现（提供），被另外某些类使用（需要）

前者与接口的关系称为**实现**（**realization**）

后者与接口的关系称为**使用**（**use**）



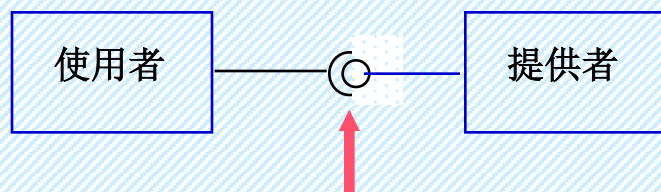
同一个接口

对实现者而言是**供接口**（**provided interface**）

对使用者而言是**需接口**（**required interface**）

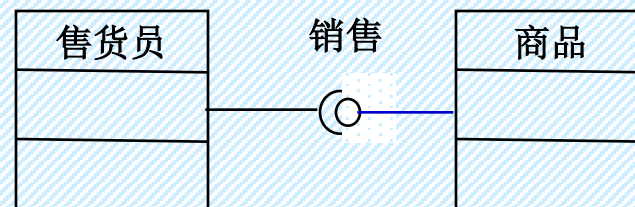
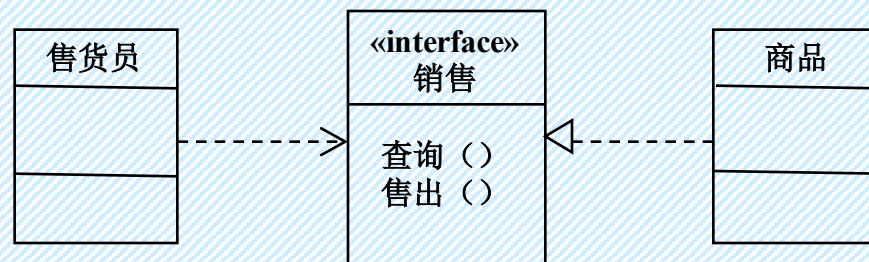
表示法（简略方式）：

——托球-托座

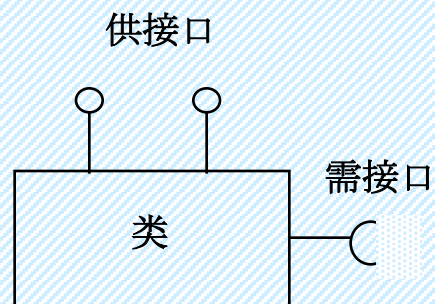


提供者的供接口（托球）
使用者的需接口（托座）

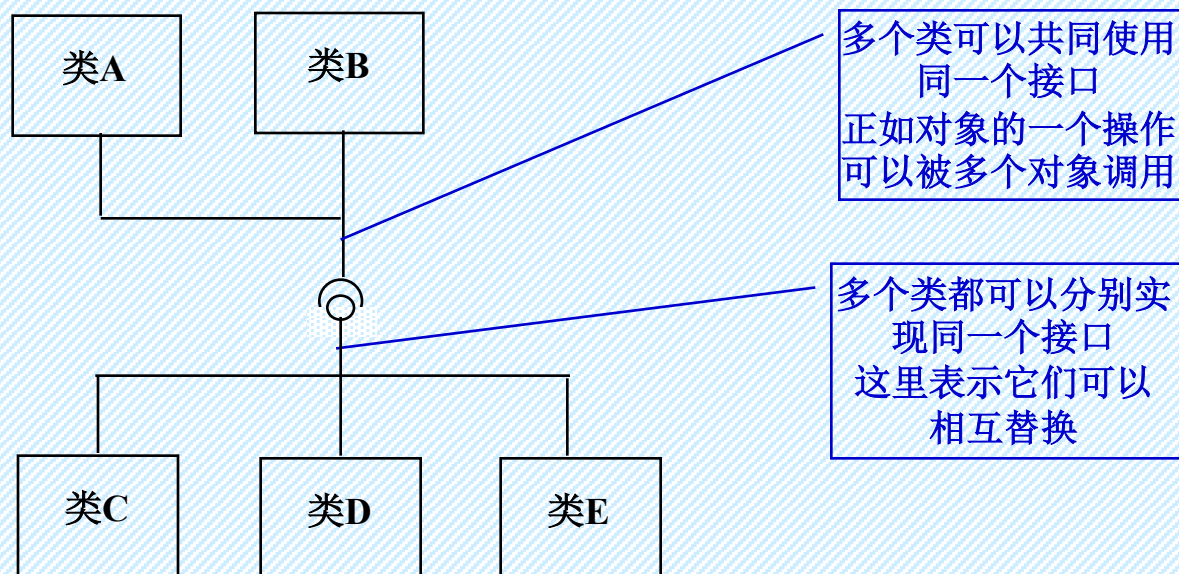
例：



在一个类上可以画出它所有的供接口和需接口



一个接口可以由多个类使用，它也可以由多个类实现



接口与类的区别

类既有属性又有操作；

接口只是声明了一组操作，没有属性。

在一个类中定义了一个操作，就要在这个类中真正地实现它；

接口中的操作只是一个声明，不需要在接口中加以实现。

类可以创建对象实例；

接口则没有任何实例。

引入接口概念的好处

在接口的使用者和提供者之间建立了一种灵活的衔接机制，有利于对类、构件等软件成分进行灵活的组装和复用。

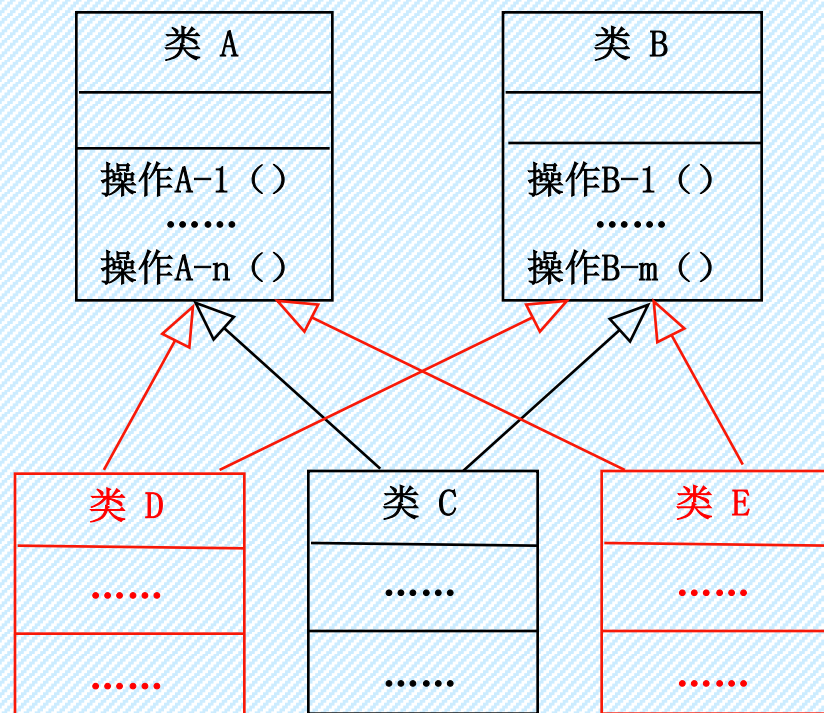
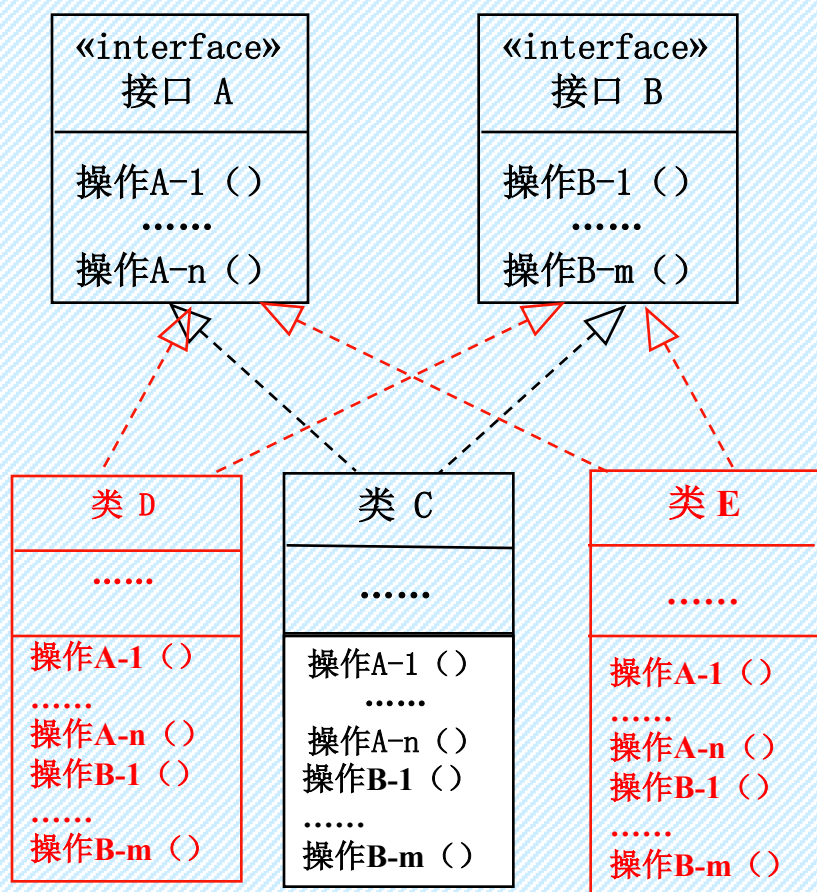
将操作的声明与实现相分离，隔离了接口的使用者和提供者的相互影响。使用者只需关注接口的声明，不必关心它的实现；提供者不必关心哪些类将使用这个接口，只是根据接口的声明中所承诺的功能来实现它，并且可以有多种不同的实现。

接口概念对描述构件之间的关系具有更重要的意义

——教材将做进一步介绍

接口与多继承的比较

接口果真能部分地解决多继承问题吗？



对象之间的四种关系

1. **一般-特殊关系** —— 又称**继承**关系，反映事物的分类。由这种关系可以形成**一般-特殊结构**。
2. **整体-部分关系** —— 即**聚合**关系。反映事物的构成。由这种关系可以形成**整体-部分结构**。
3. **关联关系** —— 对象实例集合（类）上的一个关系，其中的元素提供了被开发系统的应用领域中一组有意义的信息。
4. **消息关系** —— 对象之间的动态联系，即一个对象在执行其操作时，请求其他对象为它执行某个操作，或者向其他对象传送某些信息。反映了事物之间的行为依赖关系。

这些关系形成了类图的关系层

8.1 一般-特殊结构

概念——同义词和近义词

继承 (inheritance) 是描述一般类和特殊类之间关系的最传统、最经典的术语。有时作为动词或形容词出现。

一般-特殊 (generalization-specialization) 含义最准确，而且不容易产生误解，恰切地反映了一般类（概念）和特殊类（概念）之间的相对（二元）关系；也用于描述结构，即一般-特殊结构。缺点是书写和阅读比较累赘。

泛化 (generalization) 取“一般-特殊”的一半，是UML的做法。比较简练，但是只反映了问题的一方面。作为关系的名称尚可，说结构是一个“泛化”则很勉强。

分类 (classification) 接近人类日常的语言习惯，体现了类的层次划分，也作为结构的名称。在许多的场合被作为一种原则。

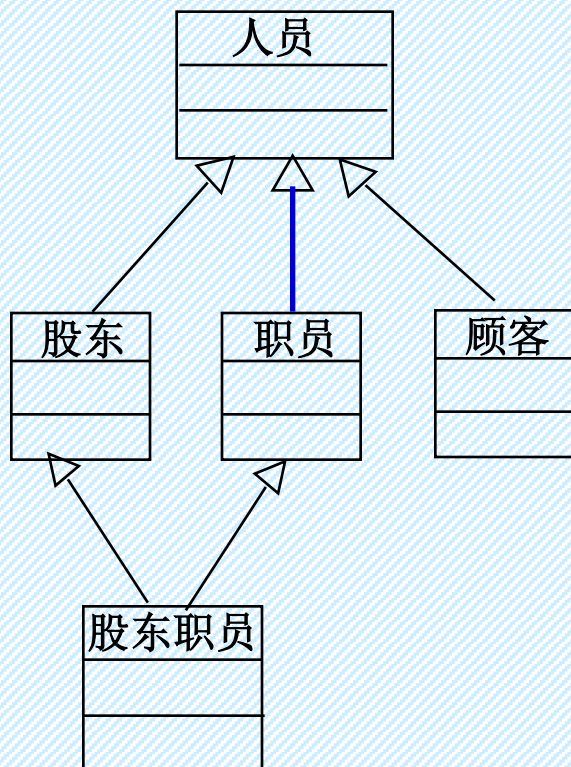
本书主要采用“一般-特殊”这个术语

相关概念：一般类、特殊类、继承、多继承、多态

语义：“is a kind of”

一般-特殊关系（继承关系）是类之间的一种二元关系
——是一种**基本的模型元素**；
由这种关系所形成的结构是一般-特殊结构
——是一种**复合的模型成分**。

例：



这是**1**个一般-特殊结构
包含**5**个一般特殊关系

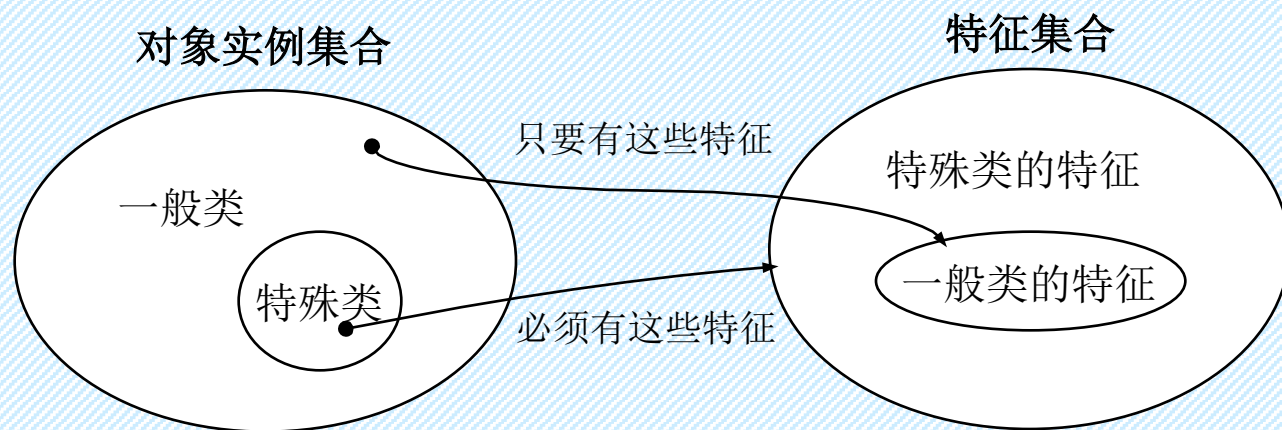
一般类和特殊类的两个定义

定义1: 如果类A具有类B的全部属性和全部操作，而且具有自己特有的某些属性或操作，则A叫做B的特殊类，B叫做A的一般类。一般类与特殊类又称父类与子类。

定义2: 如果类A的全部对象都是类B的对象，而且类B中存在不属于类A的对象，则A是B的特殊类，B是A的一般类。

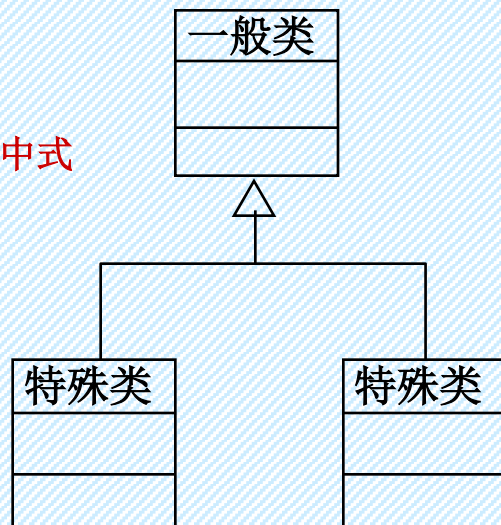
——可证明，以上两种定义是等价的

理解一般类与特殊类之间的关系

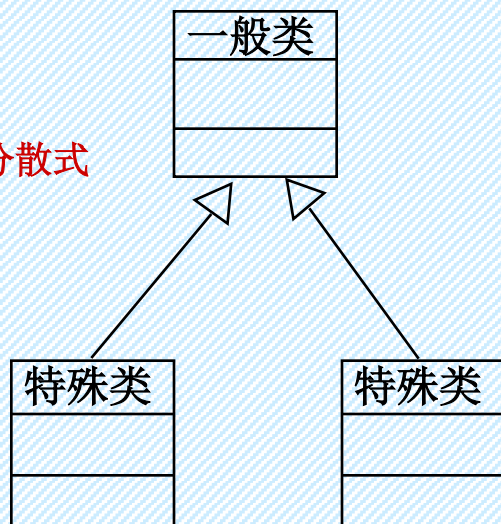


表示法

集中式



分散式

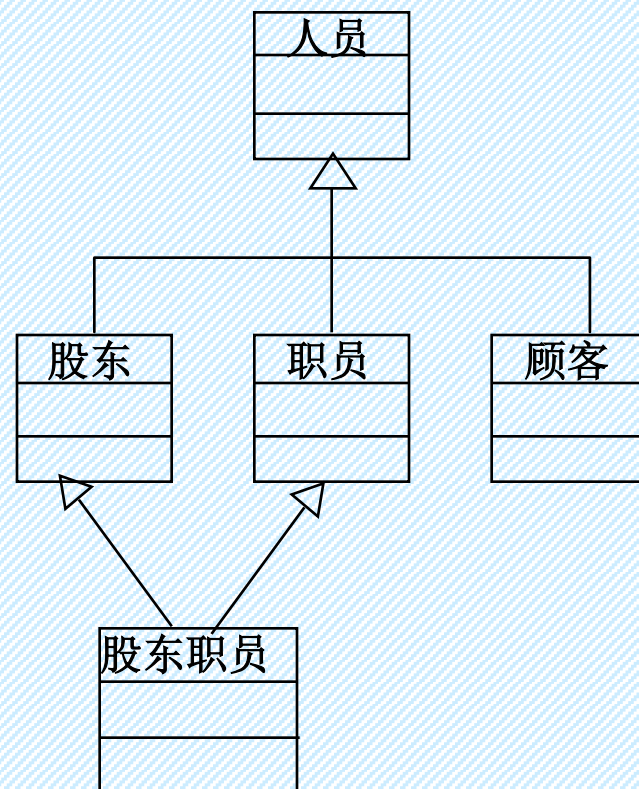


* 对继承的属性或操作重新定义

× 拒绝继承

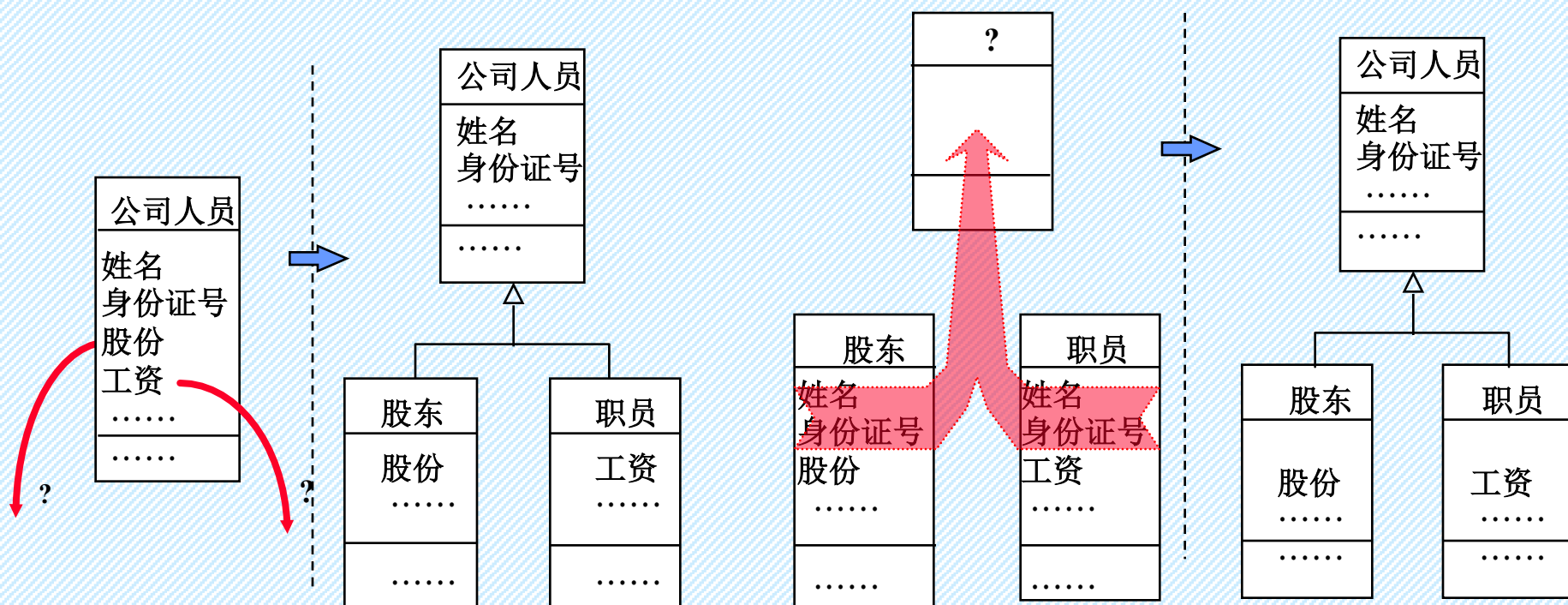
多态性的表示符号

例：

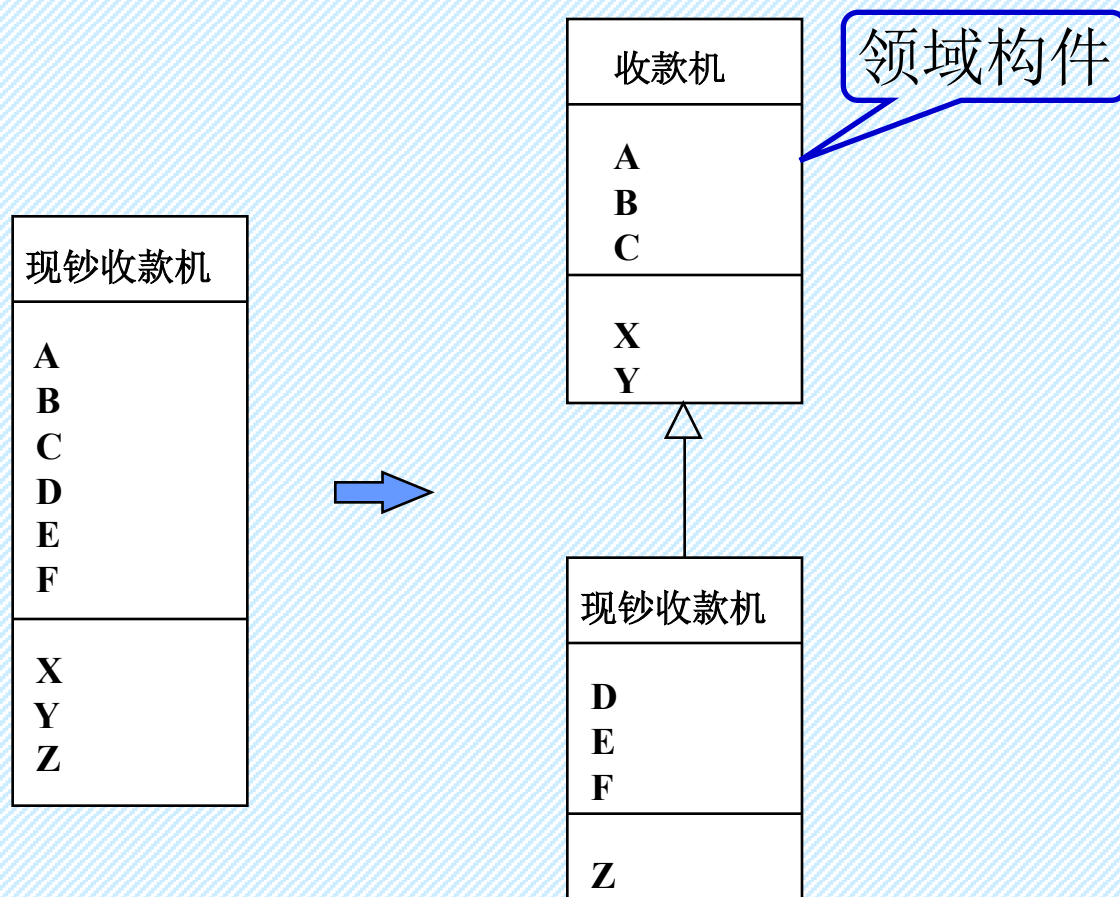


如何发现一般-特殊结构

- (1) 学习当前领域的分类学知识
- (2) 按常识考虑事物的分类
- (3) 根据一般类和特殊类的两种定义
- (4) 考察属性与操作的适应范围

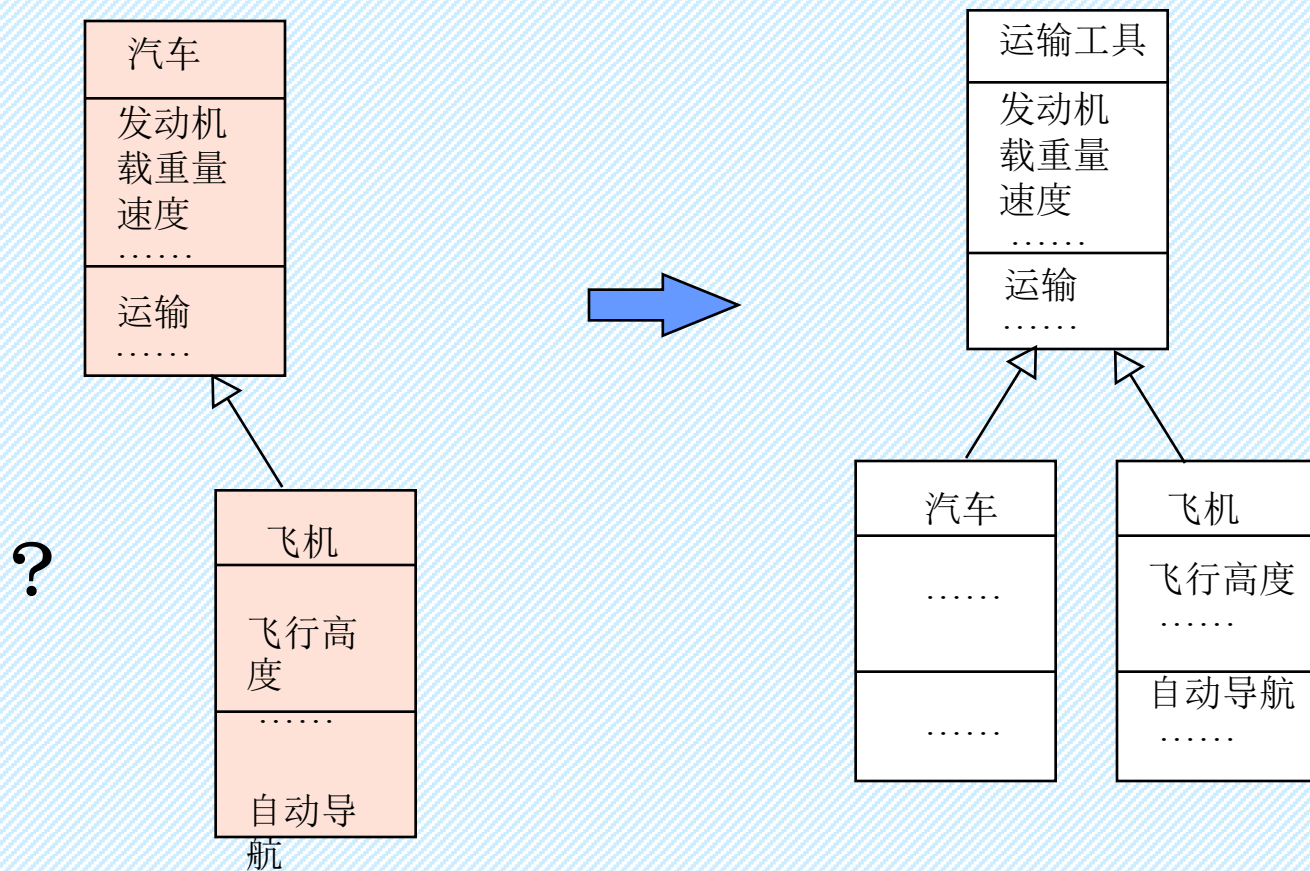


(5) 考虑领域范围内的复用



审查与调整

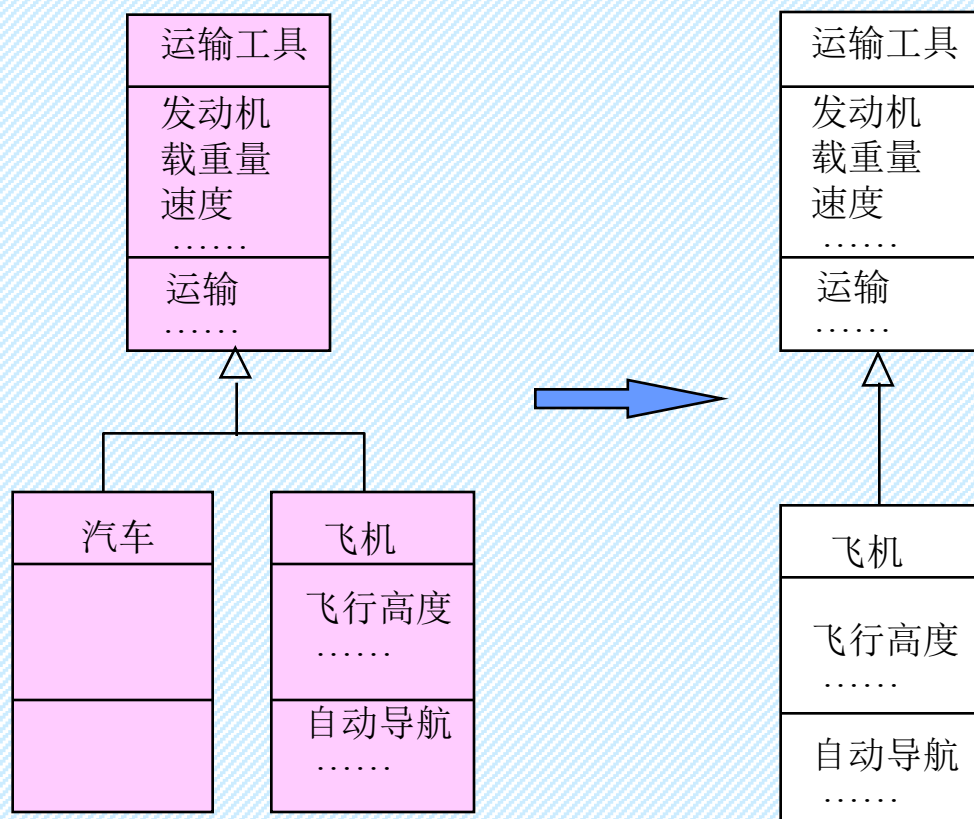
- (1) 问题域是否需要这样的分类？（例：书—线装书）
- (2) 系统责任是否需要这样的分类？（例：职员—本市职员）
- (3) 是否符合分类学的常识？（用“is a kind of”来衡量）



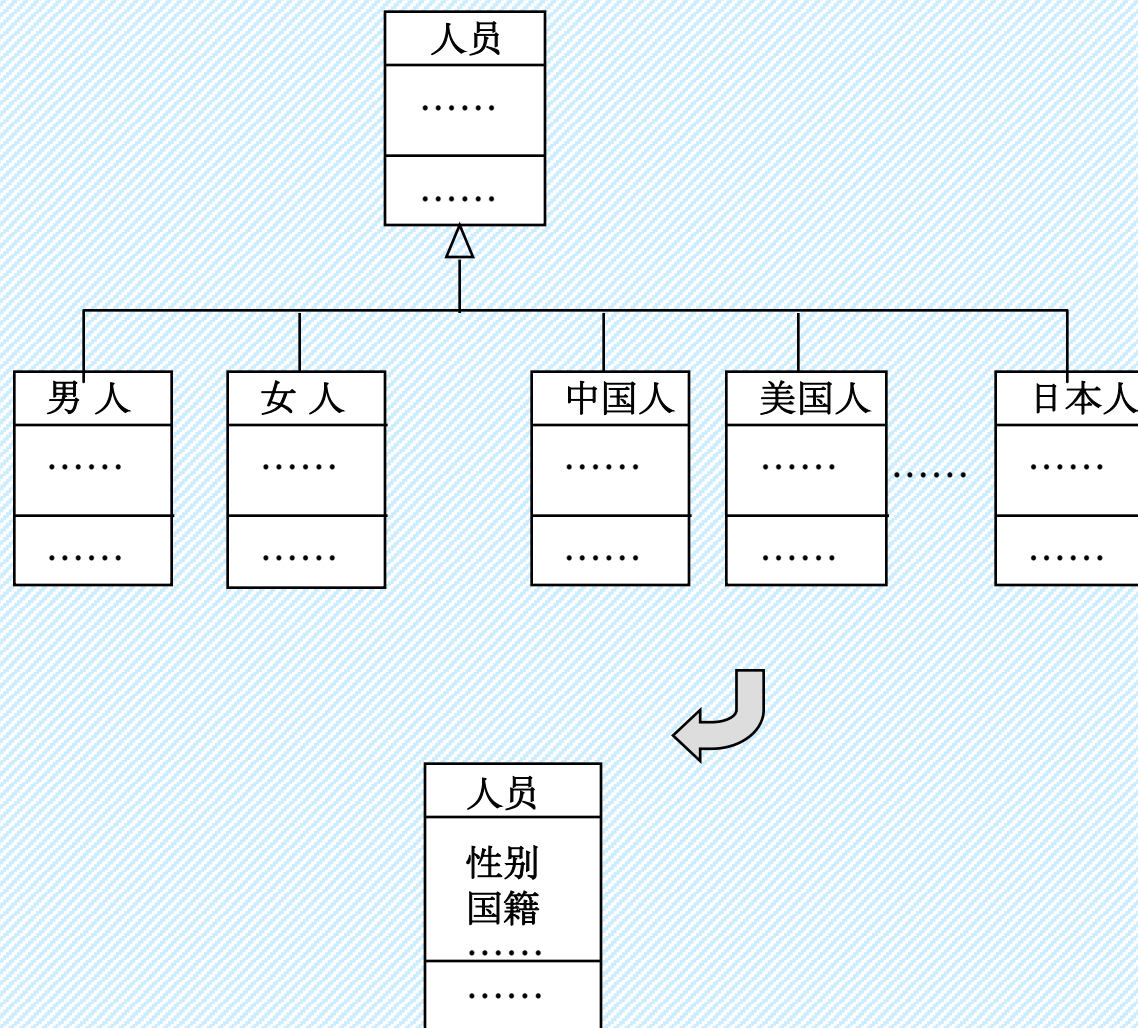
- (4) 是否真正的继承了一些属性或操作？

一般-特殊结构的简化

(1) 取消没有特殊性的特殊类

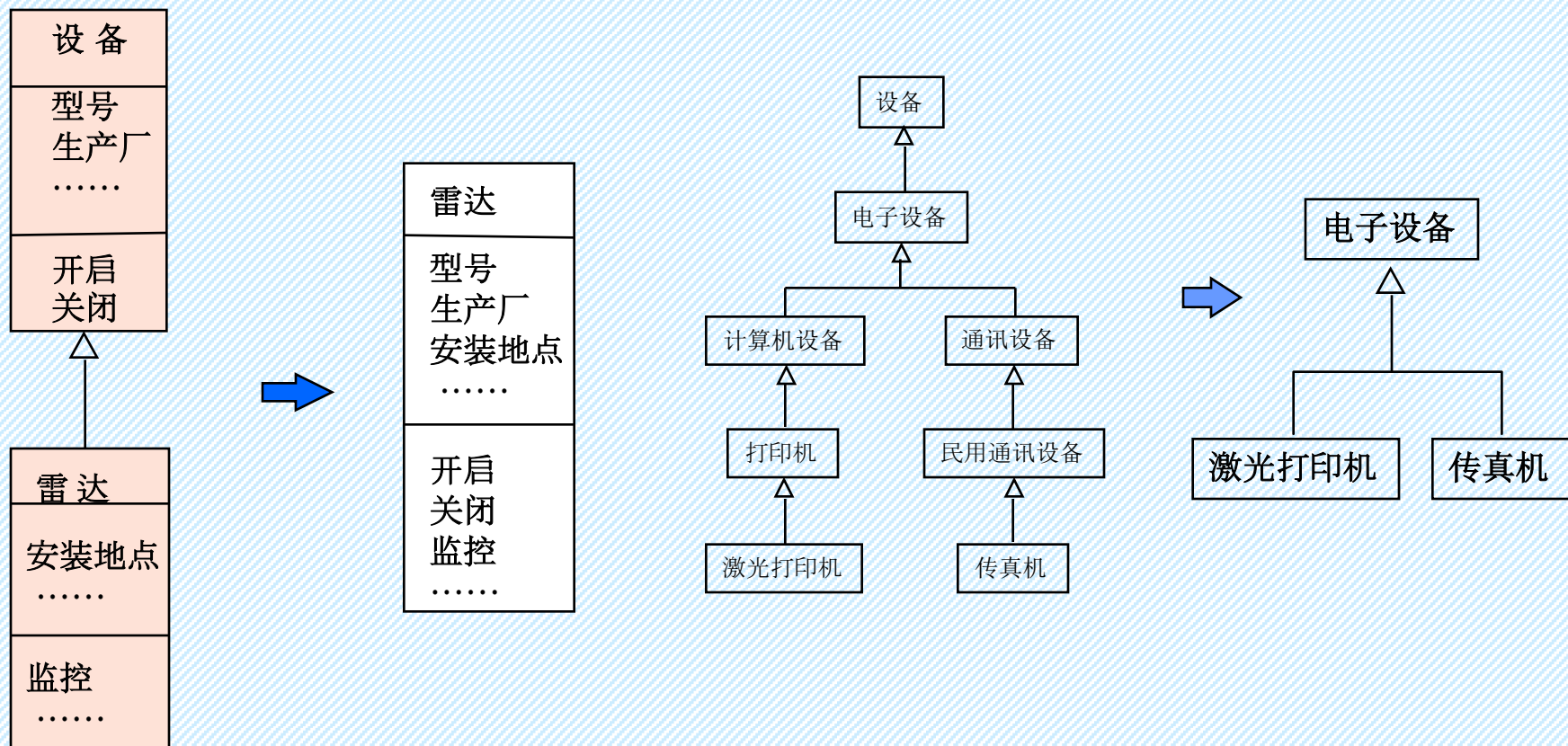


(2) 增加属性简化一般—特殊结构



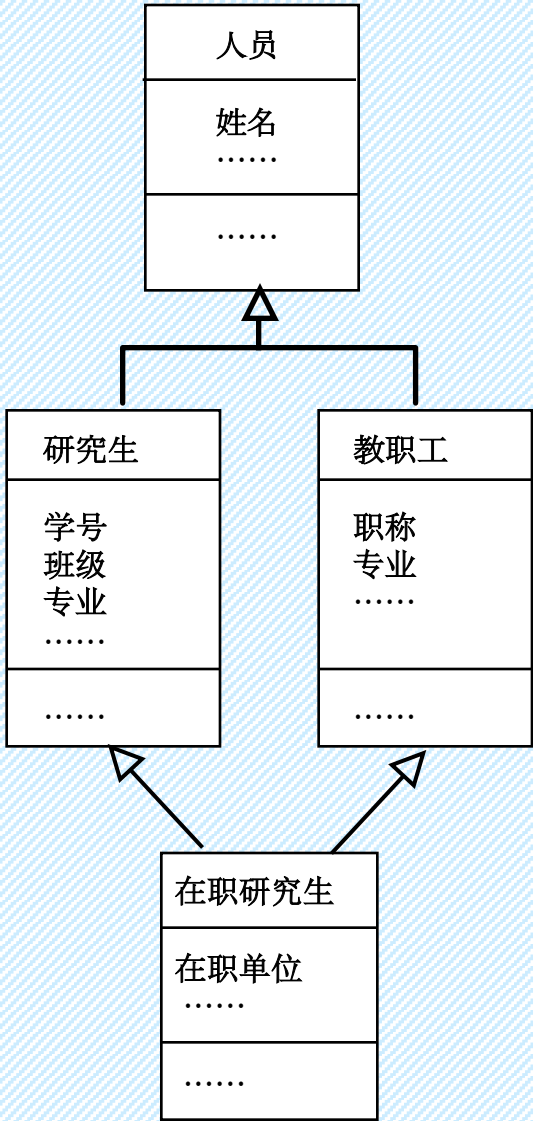
(3) 取消用途单一的一般类，减少继承层次 一般类存在的理由：

- * 有两个或两个上以上的特殊类
- * 需要用它创建对象实例
- * 有助于软件复用

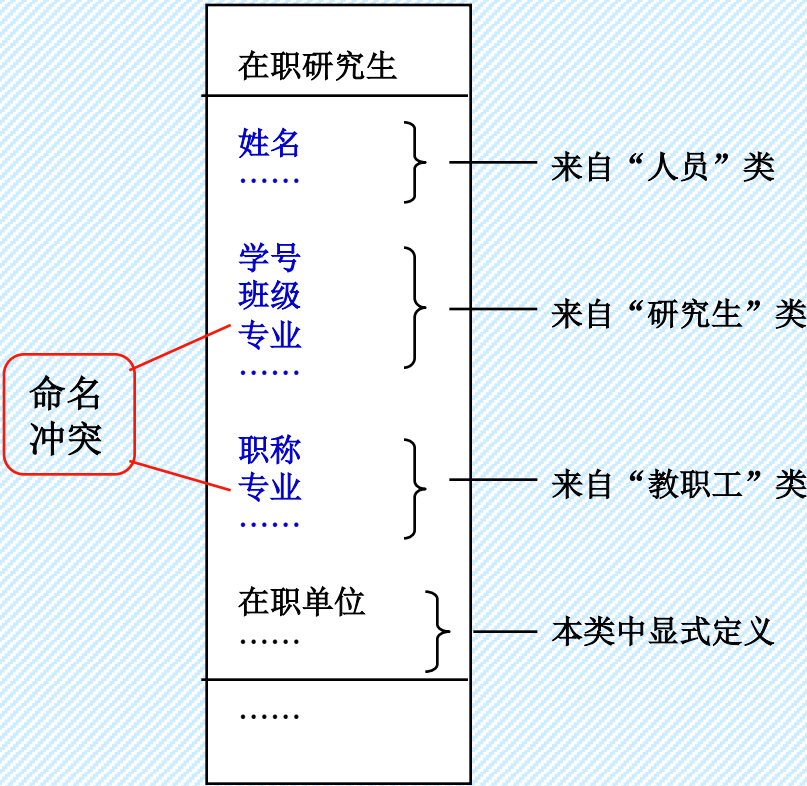


多继承：允许一个特殊类具有一个以上一般类的继承模式

例：

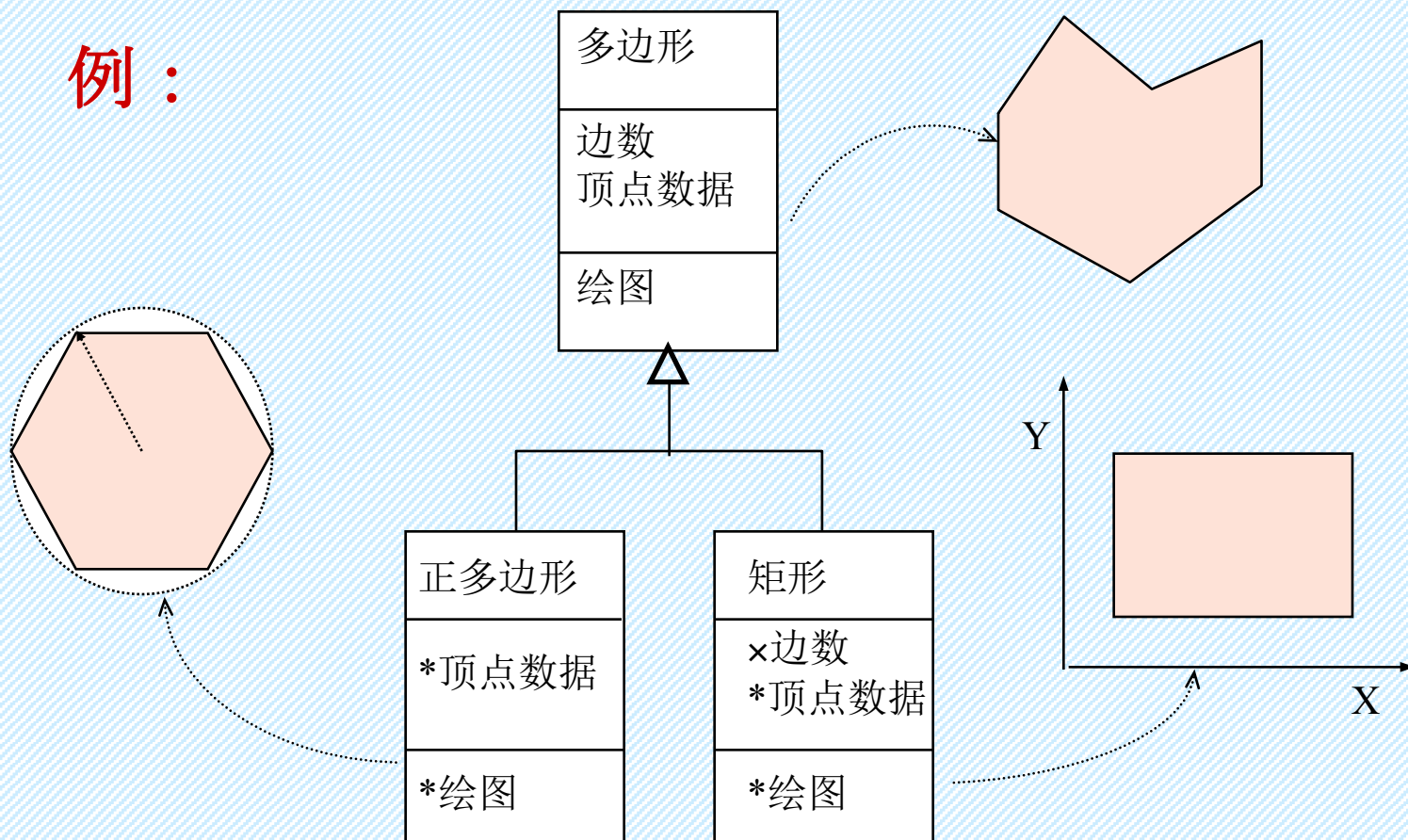


多继承特殊类的内部情况



多态： 多态是指同一个命名可具有不同的语义。**OO**方法中，常指在一般类中定义的属性或操作被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为。

例：



8.2 整体-部分结构

概念:

聚合 (aggregation), 组合 (composition)

整体-部分 (whole-part)

整体对象, 部分对象

语义: “a part of”或 “has a”

聚合关系描述了对象实例之间的构成情况, 然而它的定义却是在类的抽象层次给出的。

——从集合论的观点看聚合关系

整体-部分关系 (聚合关系) 是两个类之间的二元关系, 其中一个类的某些对象是另一个类的某些对象的组成部分。

整体-部分结构是把若干具有聚合关系的类组织在一起所形成的结构。它是一个以类为结点, 以聚合关系为边的连通有向图。

一种基本的
模型元素

由若干聚合关系形成的复合
模型成分

若类 **A** 的对象 **a** 是类 **B** 对象 **b** 的一个组成部分
——判断以下几种说法正确与否：

“对象 **b** 和对象 **a** 之间具有聚合关系” ——可以

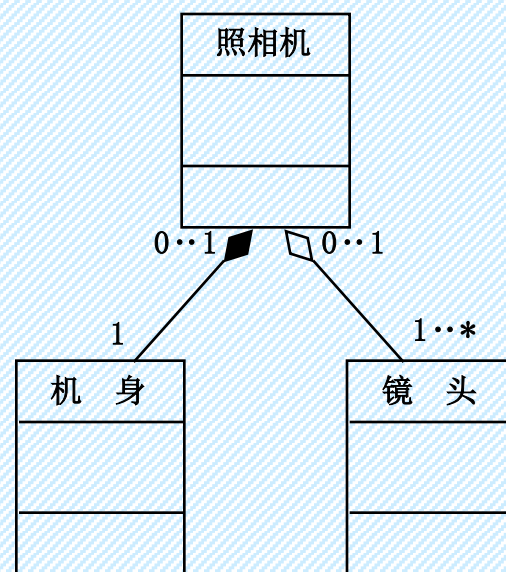
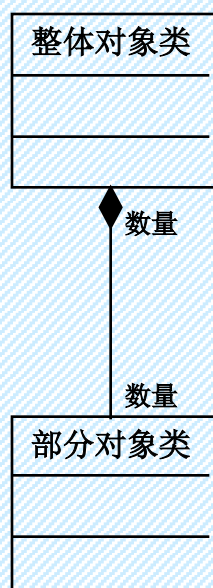
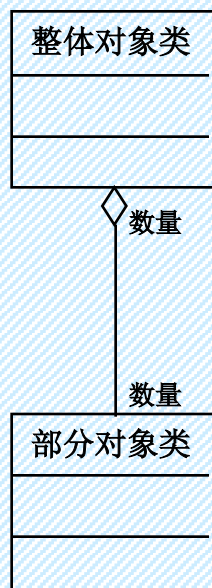
“类 **B** 和类 **A** 之间具有聚合关系” ——正确

“类 **A** 是类 **B** 的一个组成部分” ——有问题

组合 (composition) 是聚合关系的一种特殊情况，它表明整体对于部分的强拥有关系，即整体与部分之间具有紧密、固定的组成关系。

UML 把聚合定义为关联的一种特殊情况
而组合关系是聚合关系的特殊情况

表示法



在连接符两端通过数字或者符号给出关系双方对象实例的数量约束，称为多重性（multiplicity）

确定的整数 —— 给出确定的数量

—— 例如：1, 2

下界..上界 —— 给出一个范围

—— 例如：0..1, 1..4

* —— 表示多个，数量不确定

下界..*

—— 表示多个，下界确定

—— 例如 0..*, 1..*

多重性有以下3种情况：

一对一，一对多，多对多

如何发现整体-部分结构

基本策略——

考察问题域中各种具有构成关系的事物

(1) 物理上的整体事物和它的组成部分

例：机器、设备和它的零部件

(2) 组织机构和它的下级组织及部门

例：公司与子公司、部门

(3) 团体（组织）与成员

例：公司与职员

(4) 一种事物在空间上包容其它事物

例：生产车间与机器

(5) 抽象事物的整体与部分

例：学科与分支学科、法律与法律条款

(6) 具体事物和它的某个抽象方面

例：人员与身份、履历

审查与筛选

(1) 是否属于问题域？

例：公司职员与家庭

(2) 是不是系统责任的需要？

例：员工与工会

(3) 部分对象是否有一个以上的属性？

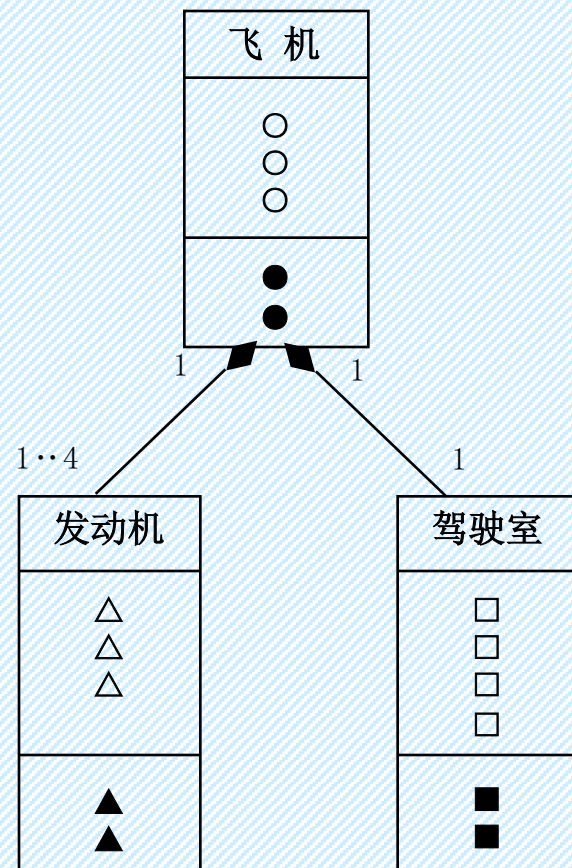
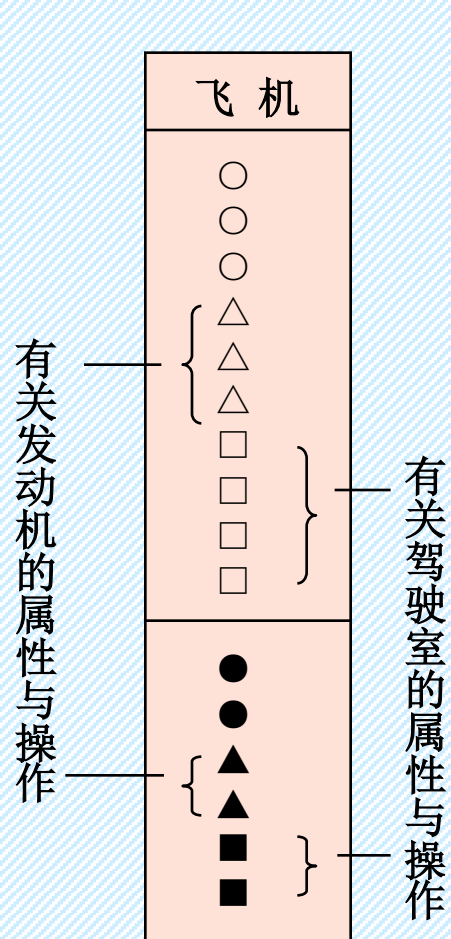
例：汽车与车轮（规格）

(4) 是否有明显的整体-部分关系？

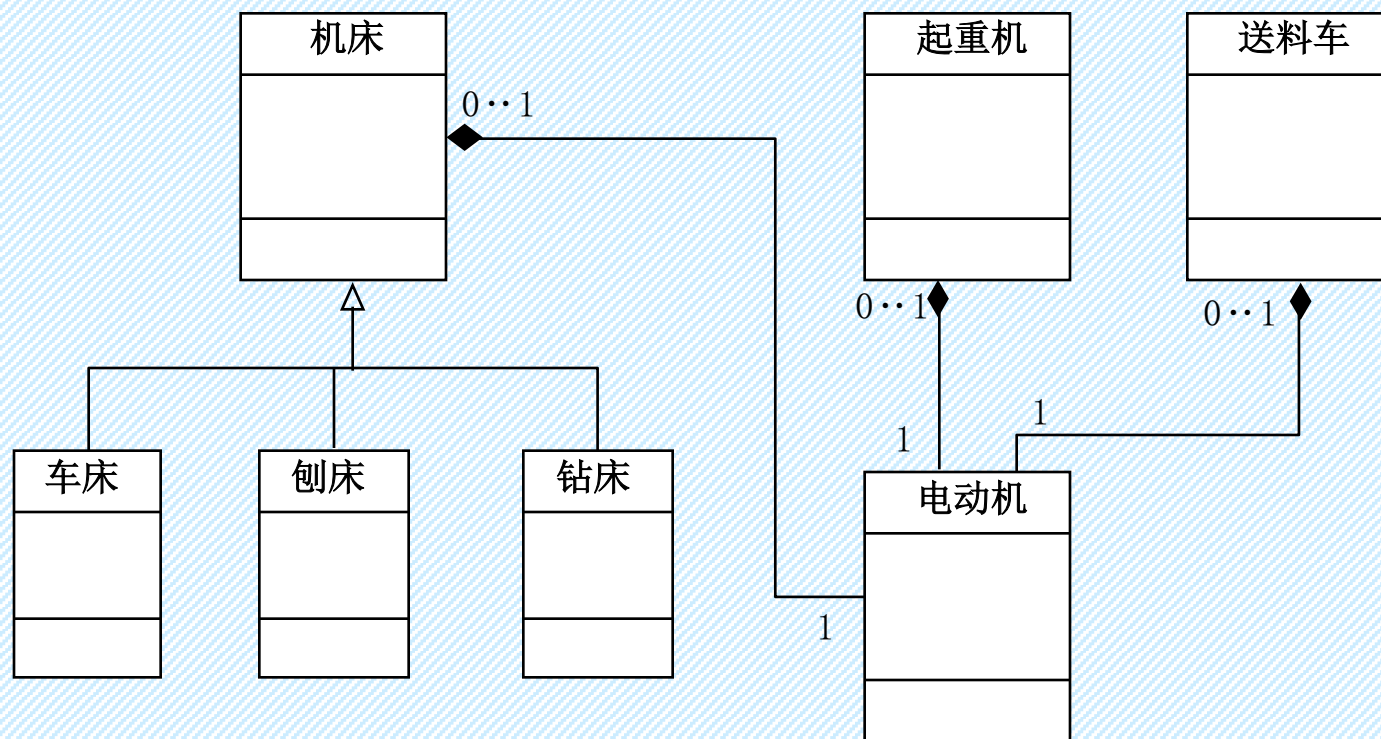
例：学生与课程

整体-部分结构的高级应用技巧

(1) 简化对象的定义



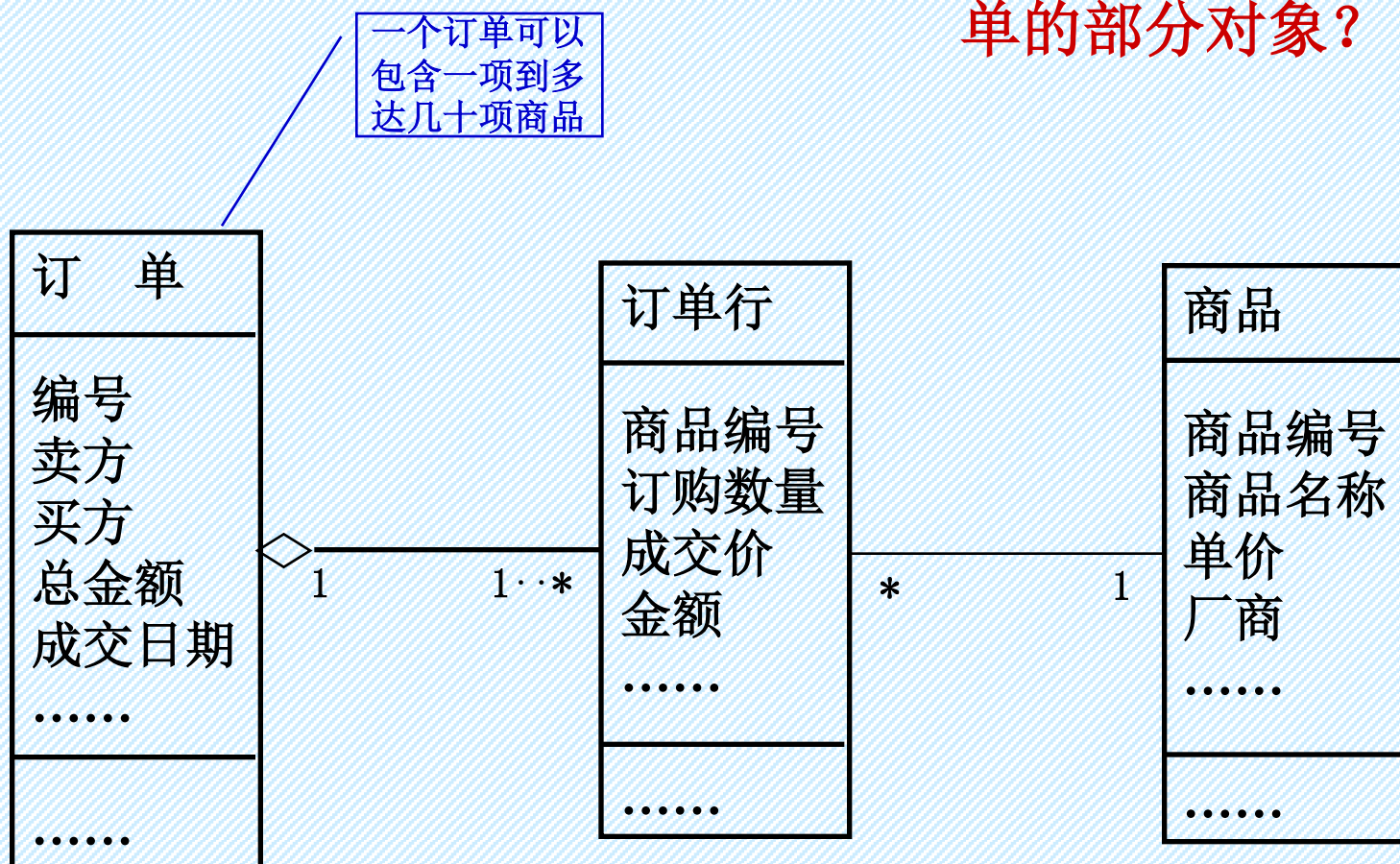
(2) 支持软件复用



(3) 表示数量不定的组成部分

提问：

能否不要订单行，
直接用商品作为订
单的部分对象？



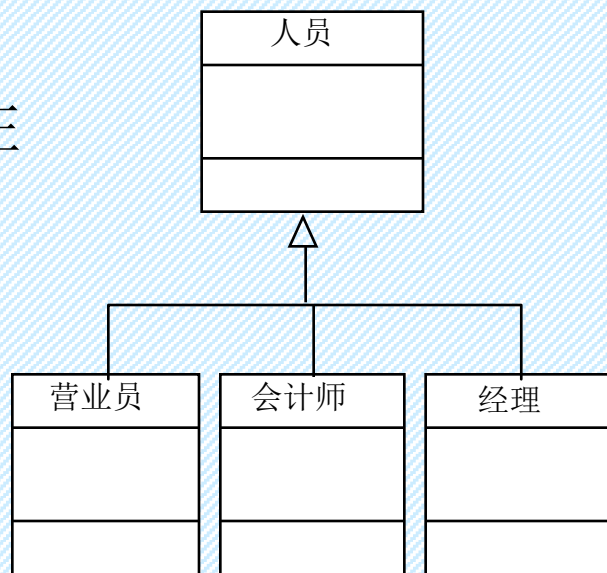
(4) 表示动态变化的对象特征

问题：对象的属性与操作定义在系统运行中动态变化，例如：

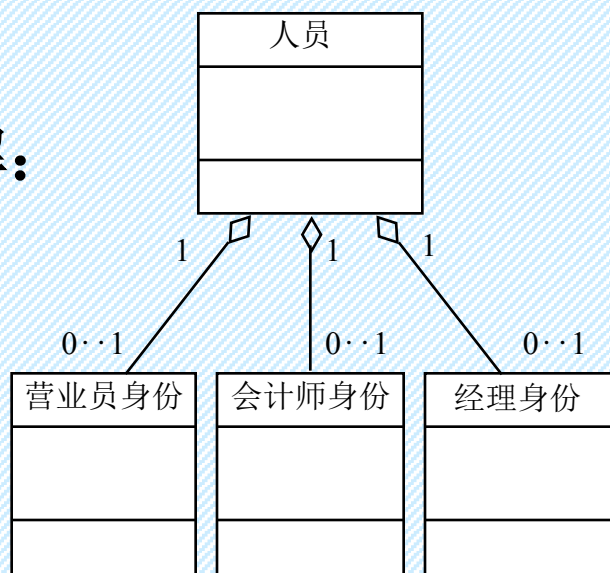
不理想的解决办法：

删除、重建

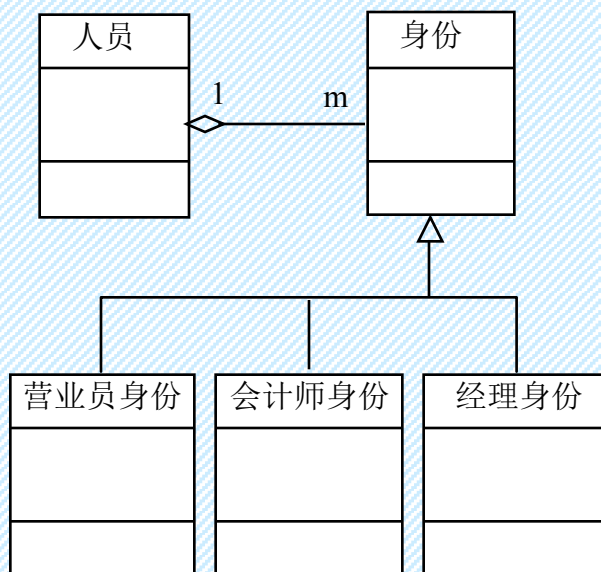
Shlaer/ Mellor的子类型迁移
“动态对象”



解：



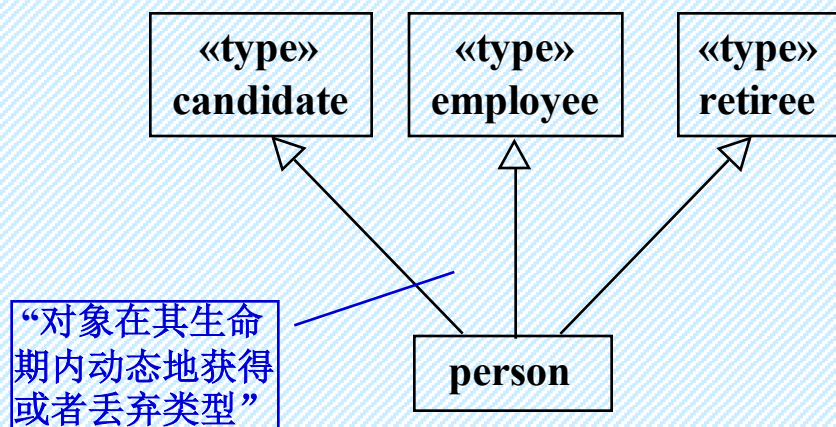
或



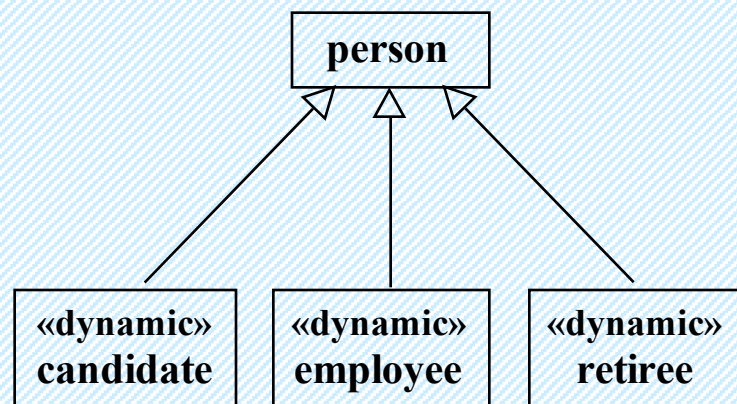
“三友”对问题的描述及解决方法

“大多数面向对象的编程语言是静态类型化的，这意味着在创建对象时就限定了对象的类型。但是随着时间的推移对象还可能扮演不同的角色。”

例子：候选者，雇员，退休者



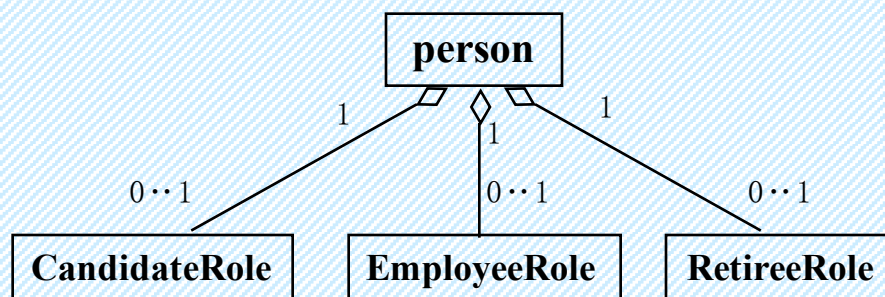
1999年第一版



2005年第二版

总之都是围绕着继承想主意，没有运用聚合。

用聚合概念解决：



从上述例子得到的启示：

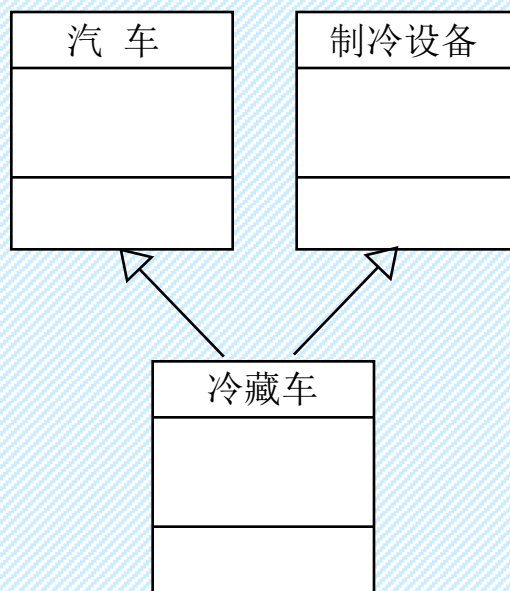
整体-部分结构有很强的表达能力

运用OO方法的基本概念可以自然而有效地解决许多在其他方法中用扩充概念解决的问题

加强对基本概念的运用，不要轻易创造新的扩充概念

两种结构的变通

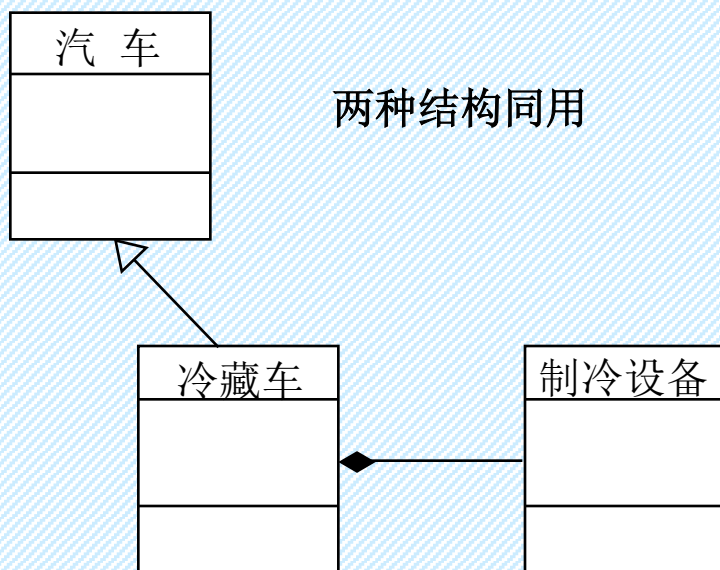
用一般—特殊结构



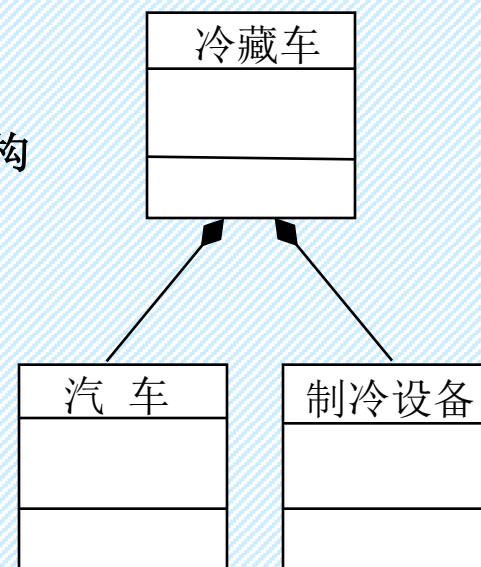
解释:

继承和聚合都是使一类对象获得另一类对象的特征，只是观察问题的角度不同。

两种结构同用



用整体—部分结构



8.3 关联

概念:

关联（**association**）是两个或者多个类上的一个关系（即这些类的对象实例集合的笛卡儿积的一个子集合），其中的元素提供了被开发系统的应用领域中一组有意义的信息。

二元关联（**binary association**）

n元关联（**n-ary association**）

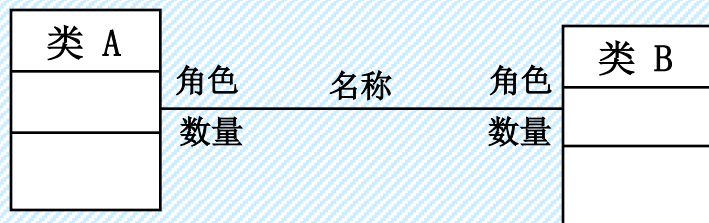
关联的实例——**有序对** 或 **n元组**，又称**链**（**link**）

关联是这些**有序对** 或 **n元组**的集合

关联位于类的抽象层次，链位于对象的抽象层次

提问：一个**n元关联**中所涉及的类的数量是否可以小于**n**？

二元关联的表示法



数量约束

固定数值：例如 1

数值范围：例如 0..1

符号： * 表示多个

0..* = * 1..* 表示 1到多个

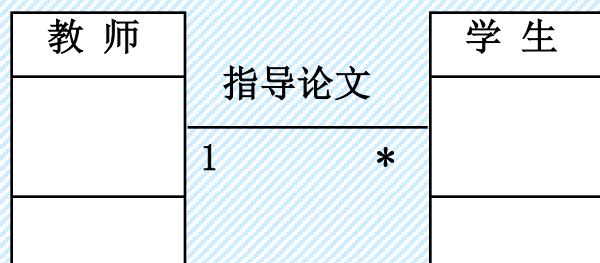
多重性的表示

一对一： $\frac{1}{1}$

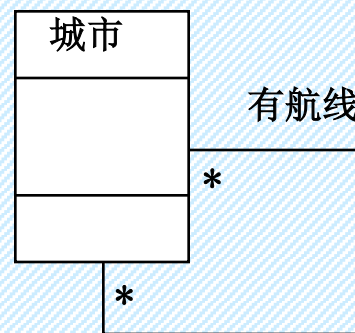
一对多： $\frac{1}{*}$

多对多： $\frac{*}{*}$

例子



教师为学生指导论文

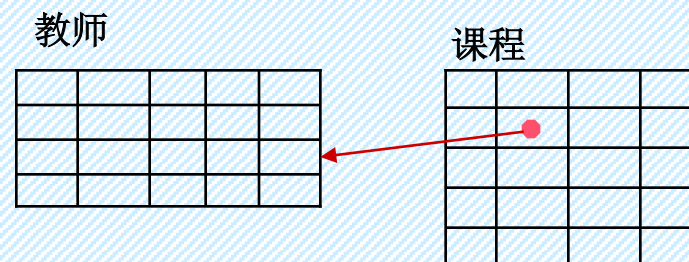
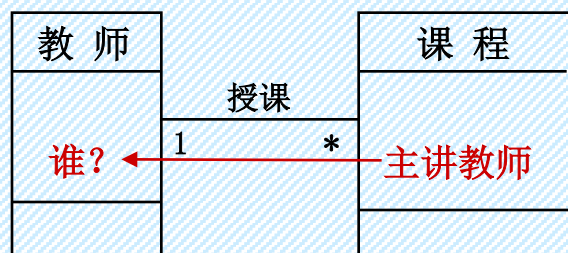


(d) 城市之间有航线

二元关联的实现（一对一和一对多）

编程语言：在程序中用两个类分别实现关联两端的类；以数量约束为“1”的类的对象实例为目标，在关联另一端的类中设置一个指向该目标的指针或者对象标识（源类的属性）。

。



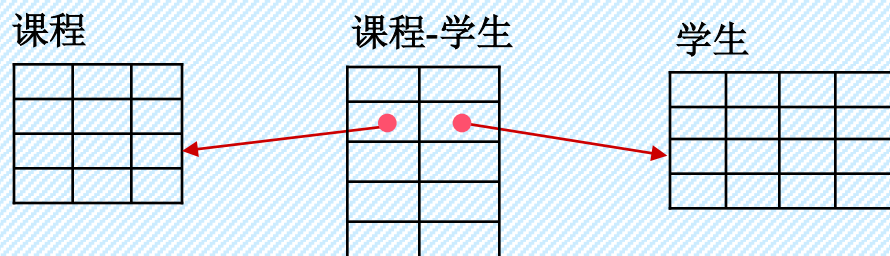
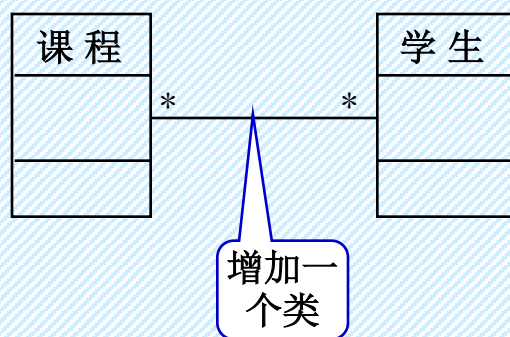
关系数据库：用两个数据库表分别实现关联两端的类；以数量约束为“1”的类对应的表的元组为目标，在关联另一端的类对应的表中设置一个指向该目标的外键（目标的主键）。

二元关联的实现（多对多）

问题：任何一端的一个对象实例的要和另一端多个对象实例发生关联，而且数量不确定。实现时不知道该设立多少个指针（或者对象标识、外键）才能够用。

编程语言：用两个类分别实现关联两端的类，同时用另外一个类来实现它们之间的关联。实现关联的类含有两个属性，分别是指向两端的类的对象实例的指针或者对象标识。

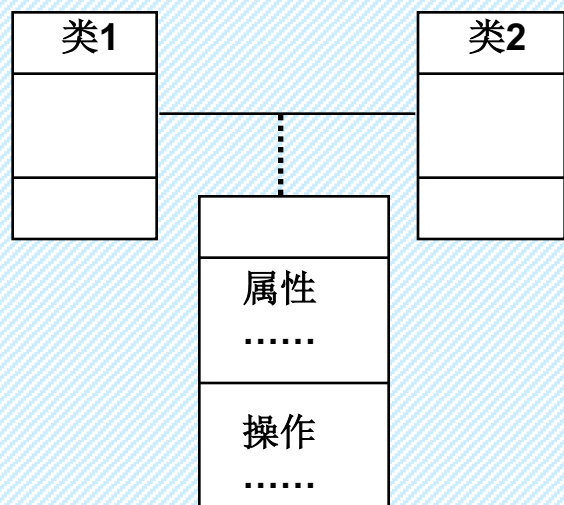
关系数据库：用两个数据库表分别实现关联两端的类，同时用另外一个数据库表来实现它们之间的关联。实现关联的数据库表含有两个属性，它们分别是指向两端的表的元组的外键。



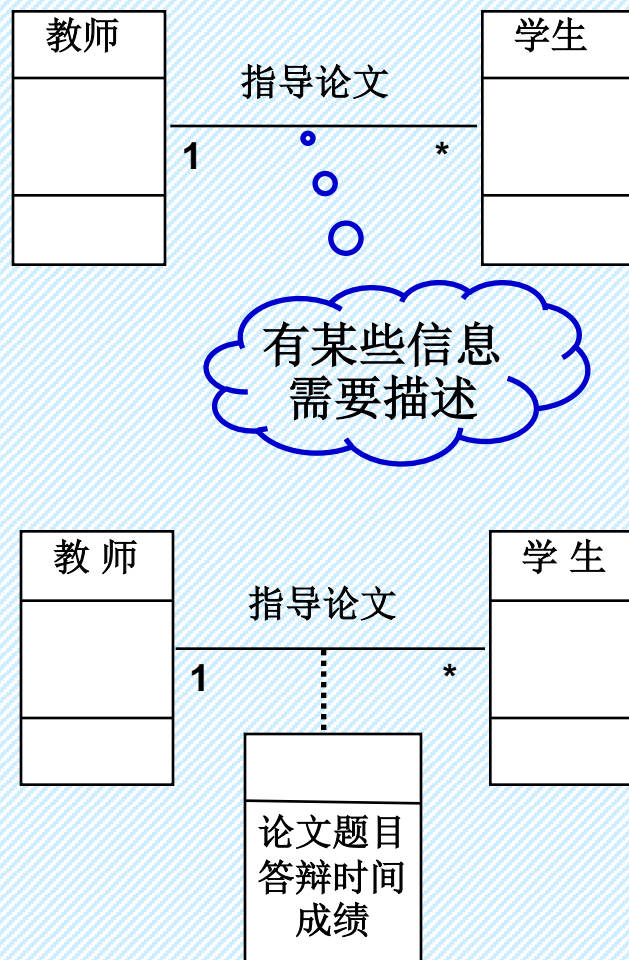
运用简单的概念及表示法解决各种复杂的关联问题

(1) 带有属性和操作的关联

OMT（及UML）的概念扩充
关联类（**association class**）



例子



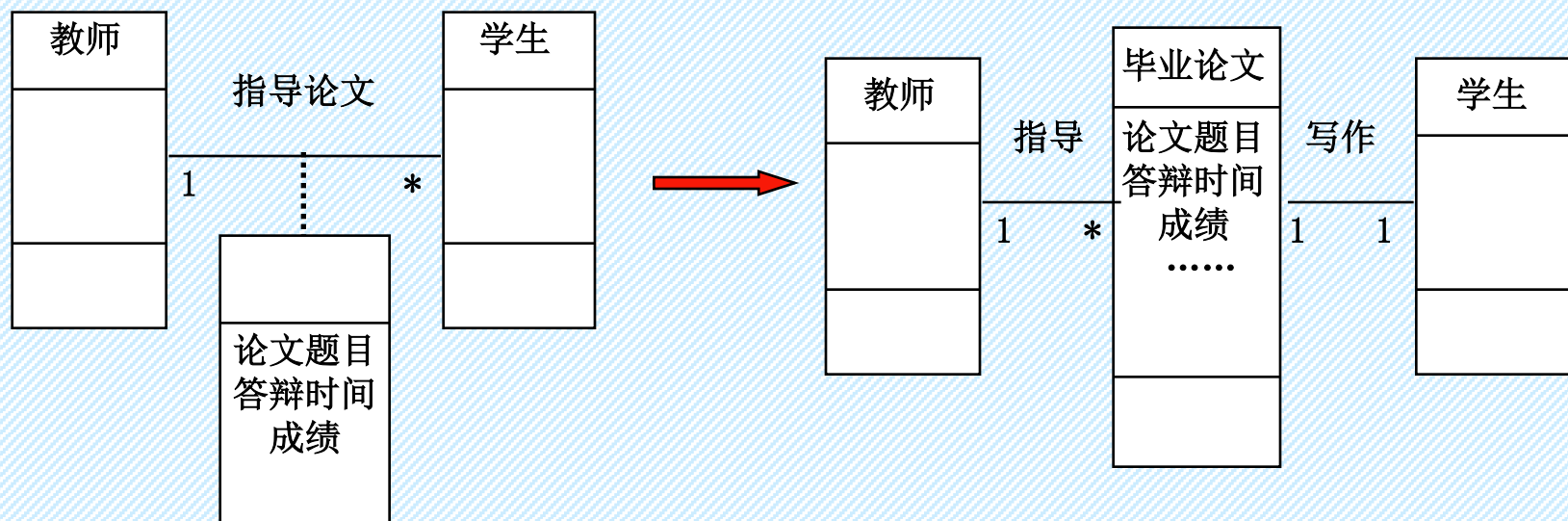
问题：增加了概念的复杂性，缺乏编程语言支持

换一种思路考虑问题：

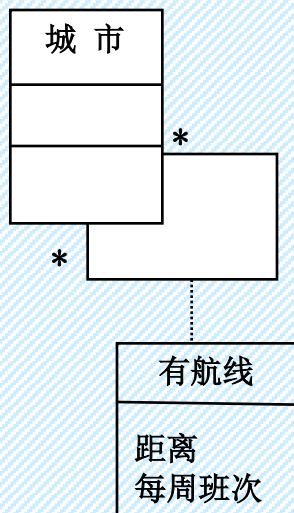
两类对象之间的关联带有某些复杂的信息，说明它们之间存在着某种事物（尽管可能是抽象事物）。

用普通的对象概念来表示这种事物，简化关联，减少概念，并加强与OOPL的对应。

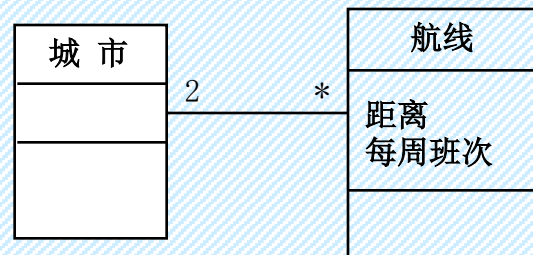
例1



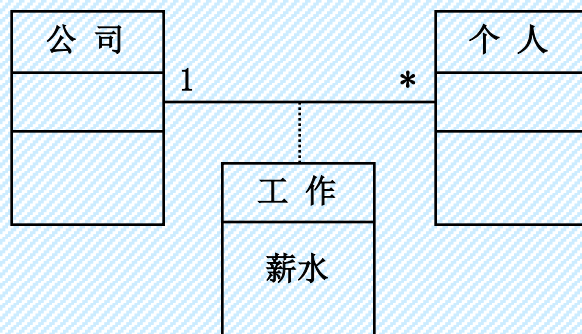
其他例子



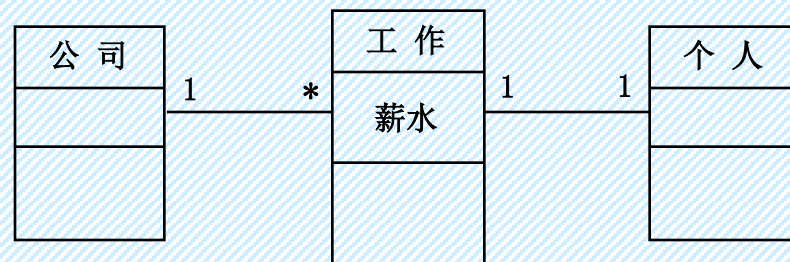
城市之间有航线



城市之间存在航线对象

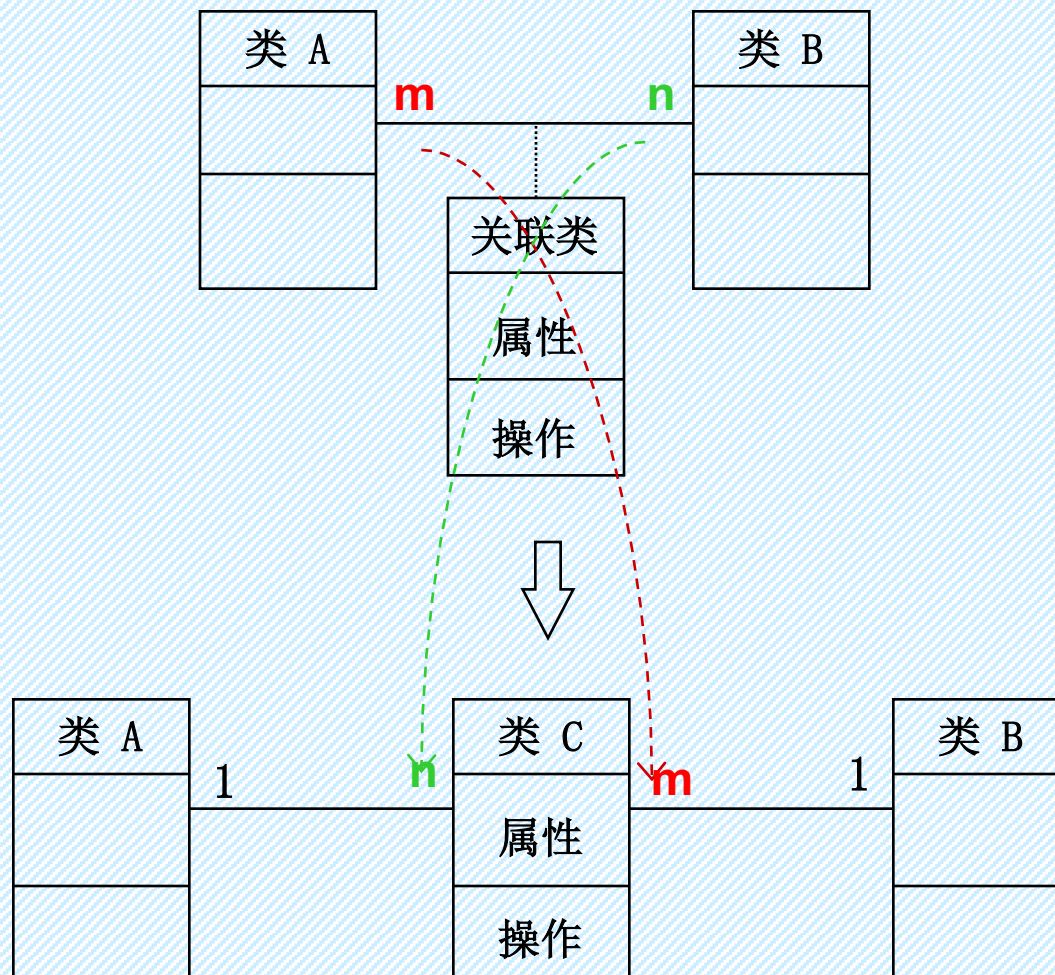


公司与个人



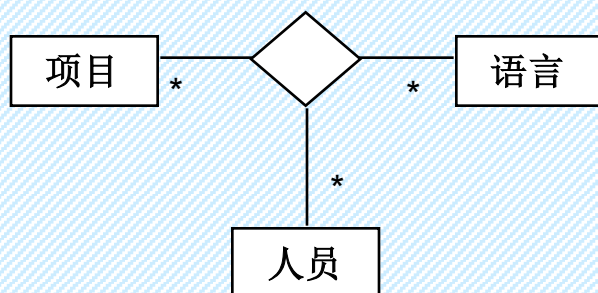
公司与个人之间存在工作对象

复杂关联表示法的转换

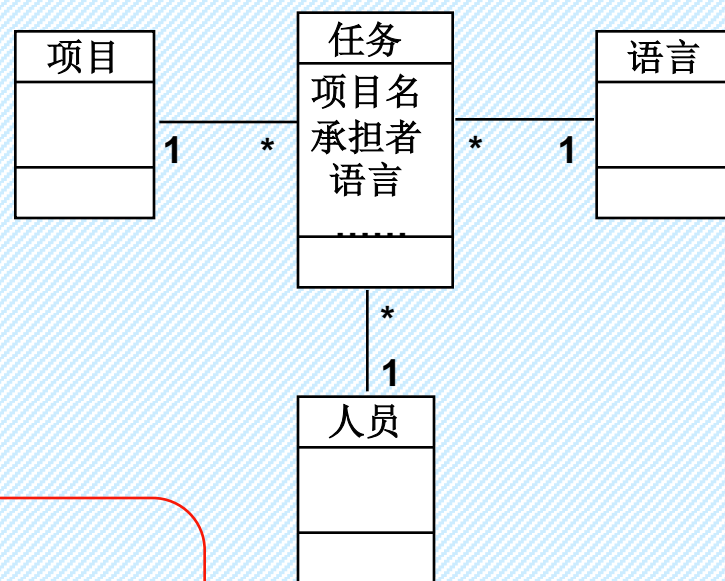


(2) n元关联

OMT的三元关联及其表示法



增设对象类表示多元关联



问题:

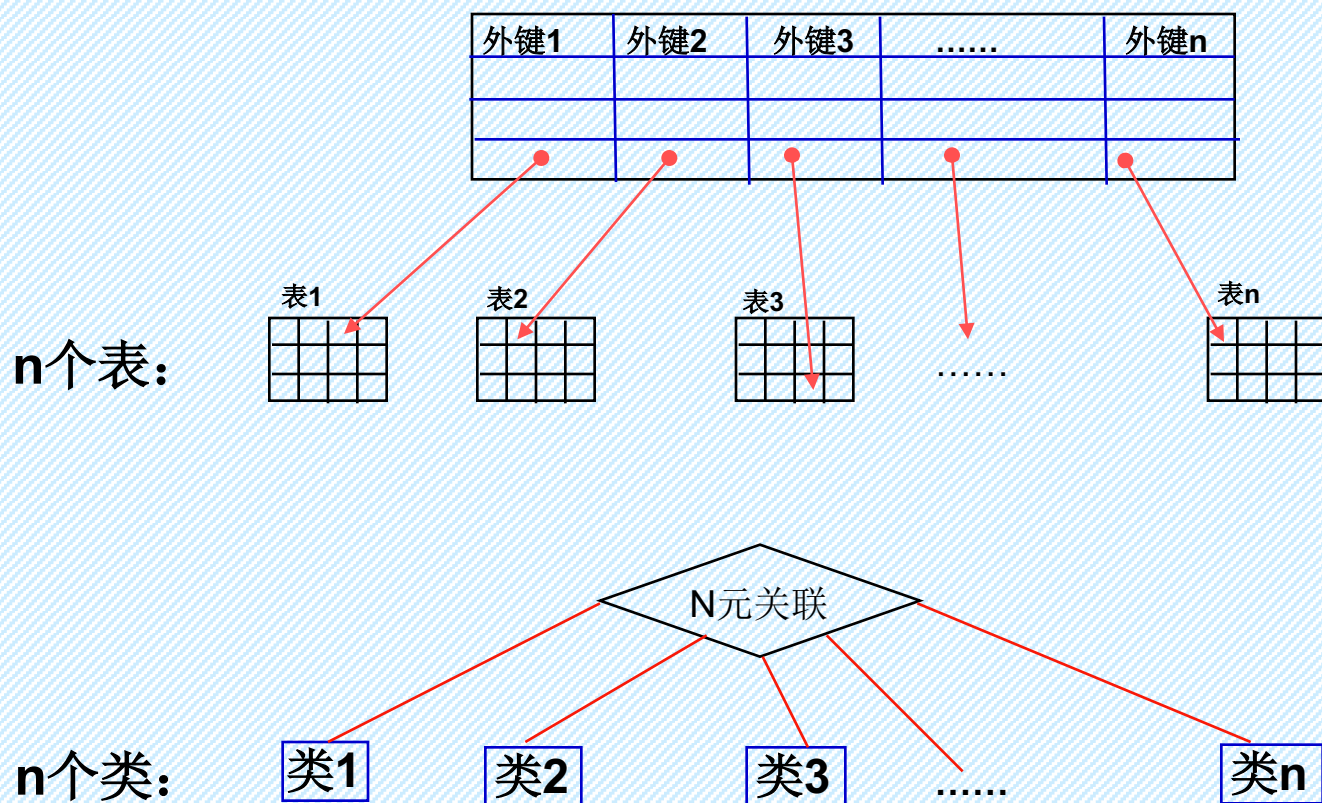
编程语言不能直接支持

可推广到n元关联, 是否要创造更多的符号?

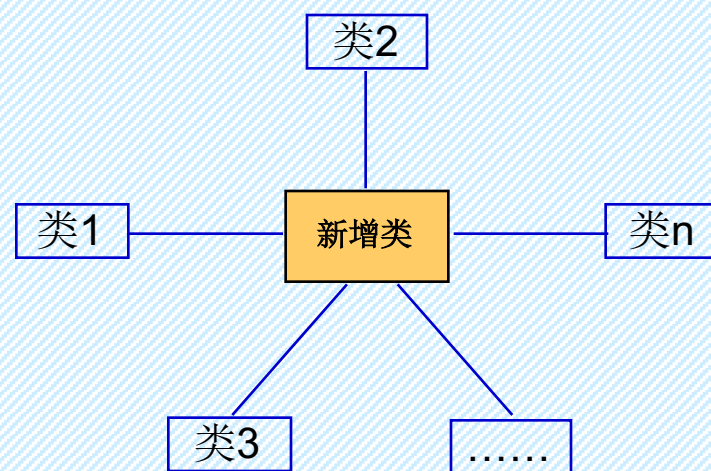
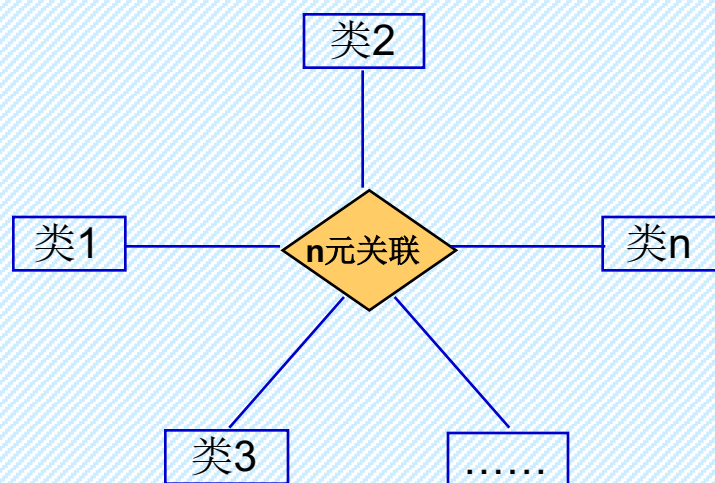
多重性表示的困难 (详后)

在理论上，**n元关联**是由若干**n元组**形成的集合，本质上也是一个类——是由每个**n元组**作为对象实例的类

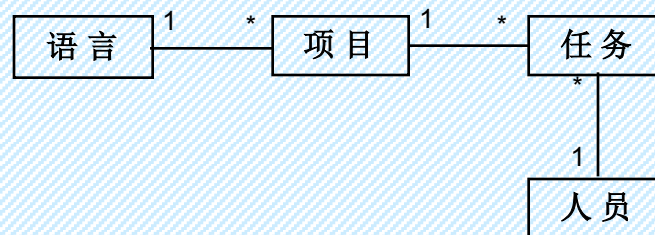
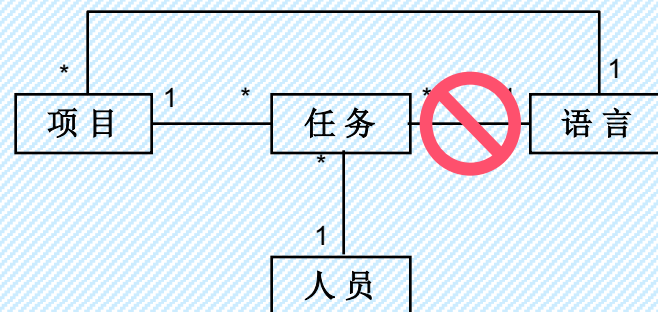
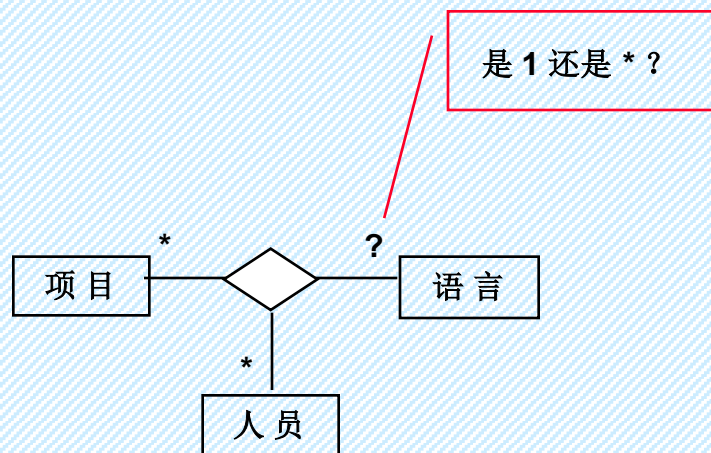
从实现的角度看，用类实现**n元关联**是最自然的选择
例如：用一个数据库表存放**n元关联**的全部**n元组**



在模型中，把n元关联定义为一个类
并定义它与原有的各个类之间的关系——都是二元关联



n元关联多重性表示的困难和解决办法



(3) 一个类在一个关联中多次出现

例：课程实习中每两名学生在一台设备上合作完成一个题目

1) 若系统要求记录和查阅哪两名学生是合作者

建立学生类到它自身的关联（如同城市之间有航线）
是一个二元关联，其中学生类在关联中出现了两次

2) 如果还要记录每组学生的实习题目和使用的设备

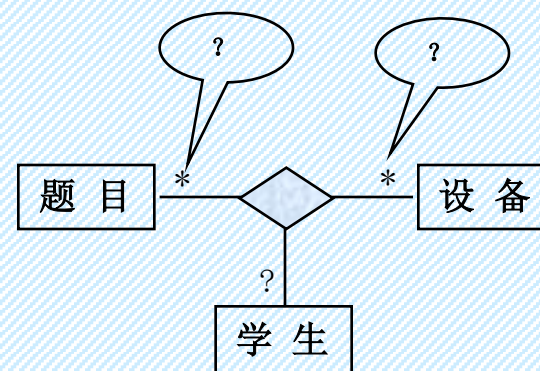
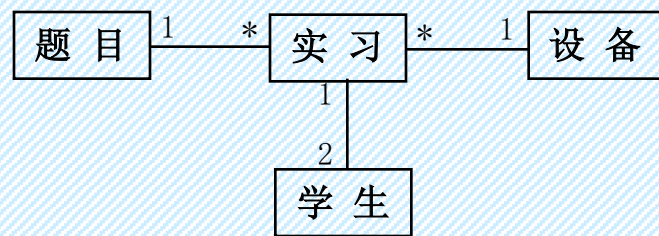
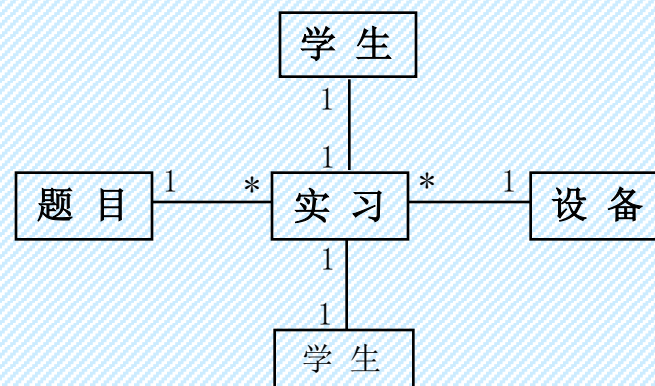
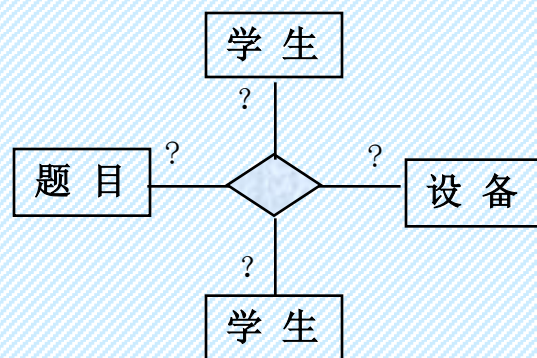
建立学生、题目、设备三个类之间的4元关联
学生类在这个关联中出现了两次

假如该系统的多重性要求是：

每两名学生在一台设备上合作完成一个题目；

一个题目可以供多组学生实习，可以在不同的设备上完成；

一台设备可以供多组学生使用，可以做不同的题目。



关联端点的复杂情况

关联端点：关联的连接线与类符号相衔接的点

修饰：在端点附近标注符号或者文字，或者画成不同的形状

多重性 (**multiplicity**)

√

有序 (**ordered**)

{ordered}

限定符 (**qualifier**)

详后

导航性 (**navigability**)

聚合标志 (**aggregation indicator**)

√

角色名 (**rolename**)

√

接口说明 (**interface specifier**)

角色名:接口说明

可变性 (**changeability**)

{frozen} ,{addOnly}

可见性 (**visibility**) 。

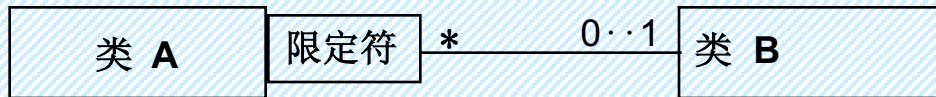
“+”、“#”或者“-”

UML对限定符的解释

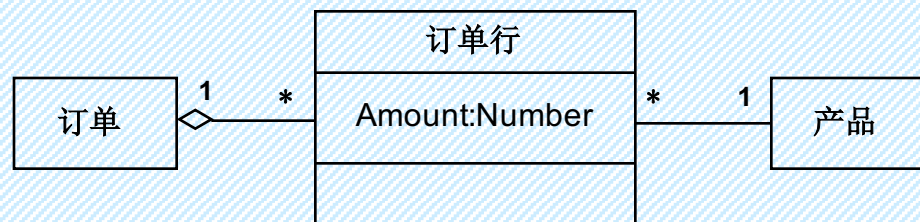
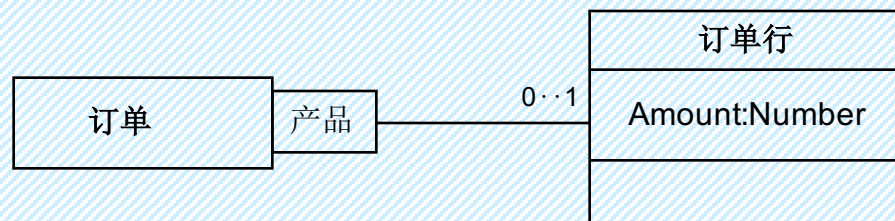
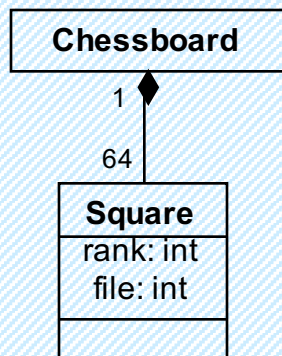
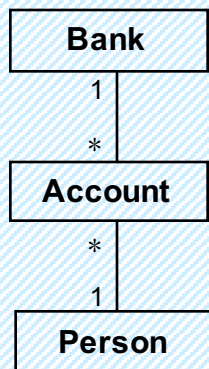
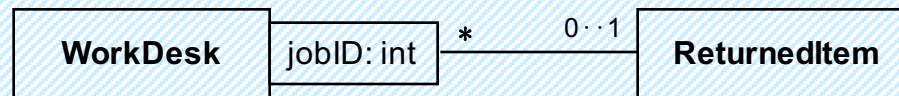
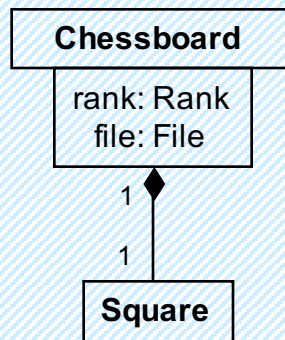
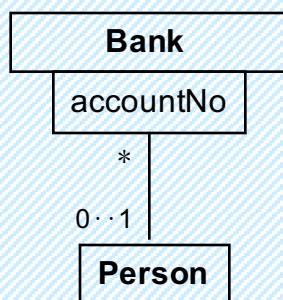
“限定符是关联的一种属性，它的值划定了跨过一个关联与一个对象相关的对象集合。”

用限定符修饰的关联称为受限关联（**qualified association**）

表示法



限定符的例子及其简单解决方案



关联端点的修饰在UML2的变化

(1) 取消了接口说明和可变性两种修饰

(2) 新增两种图形方式的修饰

X: 表示不可导航

●: 表示拥有权

(3) 增加了花括号内的特性串 (**property string**)

{subsets } 子集合

{redefines } 重定义

{union} 合并

{ordered} 有序

{nonunique} 不唯一

{sequence}, {seq} 序列

如何建立关联

1. 根据问题域和系统责任发现所需要的关联

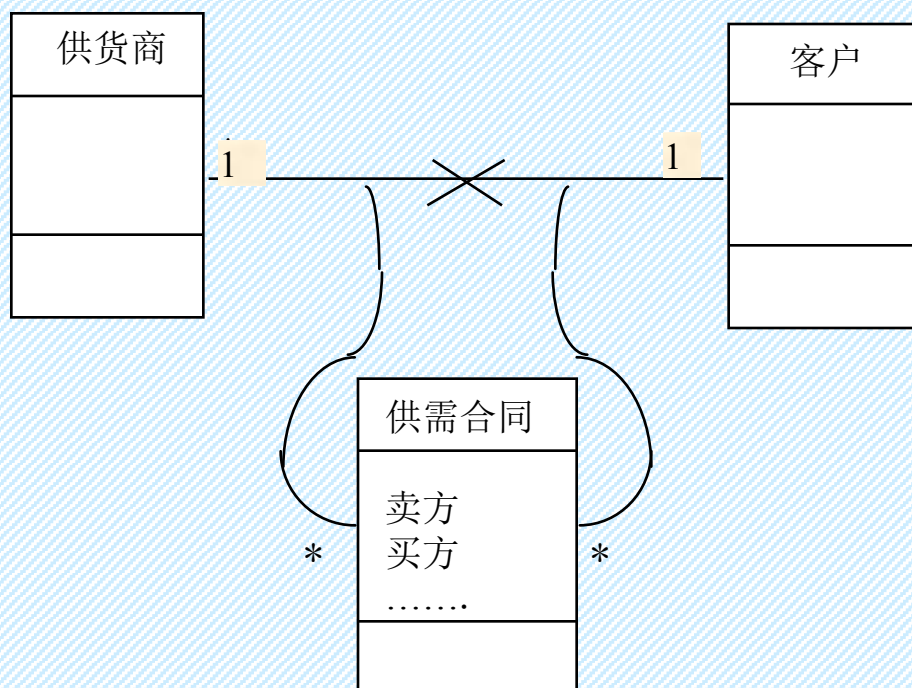
哪些类的对象实例之间存在着对用户业务有意义的关系？

- 问题域中实际事物之间有哪些值得注意的关系？
- 这种信息是否需要通过有序对（或者n元组）来体现？
- 这些信息是否需要在系统中进行保存、管理或维护？
- 系统是否需要查阅和使用由这种关系所体现的信息？

2. 关联的复杂情况处理

- 对关联属性和操作的处理
- 对n元关联的处理
- 避免一个类在关联中多次出现
- 多对多关联的处理

多对多关联
的处理



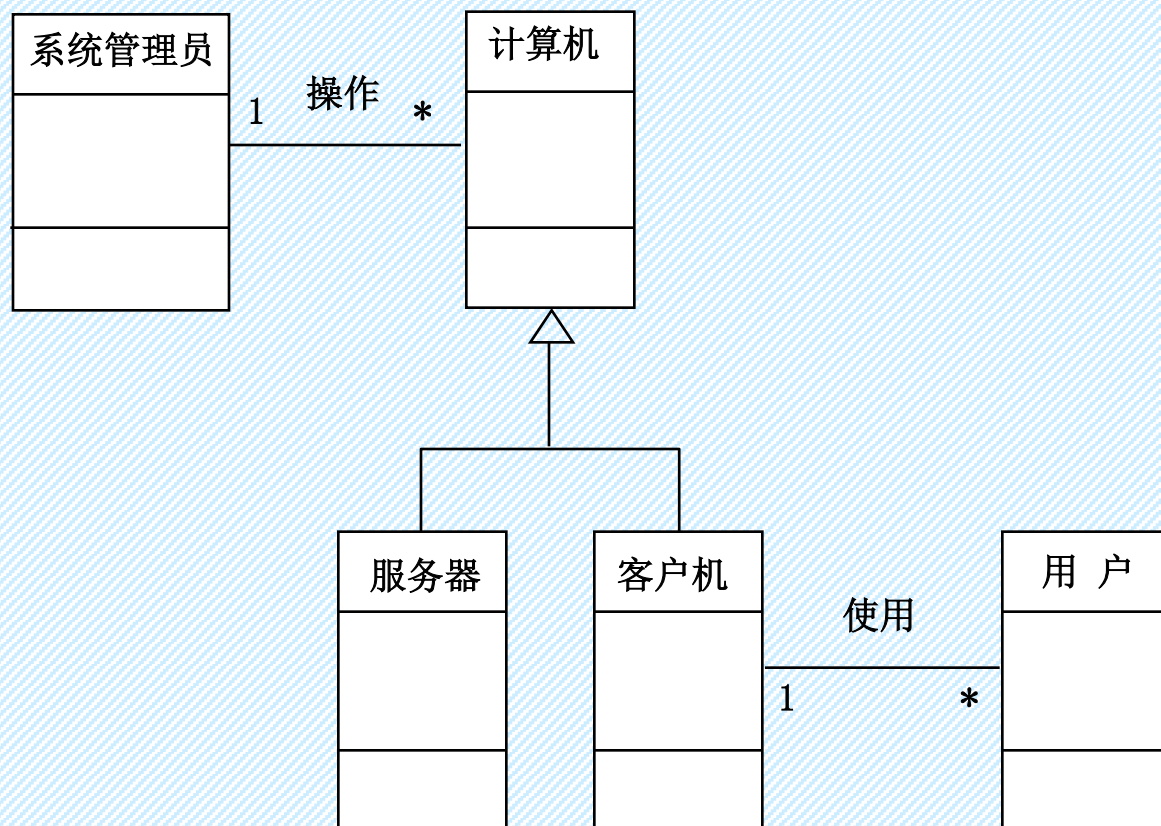
3. 为关联端点添加修饰

- 分析关联的多重性
- 给出关联名或角色名
- 识别聚合种类
- 其他修饰

导航性、特性串等——根据实际情况决定是否采用
限定符——用简单的类和关联的概念解决

4. 在类中设立实现关联的属性

5. 关联定位



1、什么是消息（message）

现实生活中——人或事物之间传递的信息，例如：
人与人之间的对话、通信、发通知、留言
交通信号灯对车辆和行人发出的信号
人发给设备的遥控信号等.....

软件系统中——进程或软件成分之间传送的信息
控制信息 例如一次函数调用，或唤醒一个进程
数据信息 例如传送一个数据文件

面向对象的系统中——（按严格封装的要求）消息是对象之间在行为上的唯一联系方式

消息是向对象发出的服务请求（狭义）

消息是对象之间在一次交互中所传送的信息（广义）

消息有发送者和接收者，遵守共同约定的语法和语义

顺序系统中的消息

- 每个消息都是向对象发出的服务请求
最常见的是函数调用
- 消息都是同步的。
- 接收者执行消息所请求的服务。
- 发送者等待消息处理完毕再继续执行。
- 每个消息只有唯一的接收者。

并发系统中的消息

控制流内部的消息——与顺序系统相同

控制流之间的消息——情况复杂得多

- 消息有多种用途

服务请求，传送数据，发送通知，传递控制信号……

- 消息有同步与异步之分

同步消息 (**synchronous message**)

异步消息 (**asynchronous message**)

- 接收者对消息有不同响应方式

创建控制流，立即响应，延迟响应，不响应

- 发送者对消息处理结果有不同期待方式

等待回应，事后查看结果，不等待不查看

- 消息的接收者可能不唯一

定向消息 (**directed message**)

广播消息 (**broadcast message**)

消息对面向对象建模的意义

消息体现了对象之间的行为依赖关系，是实现对象之间的动态联系，使系统成为一个能运行的整体，并使各个部分能够协调工作的关键因素。

在顺序系统中 消息体现了过程抽象的原则

一个对象的操作通过消息调用其他对象的操作

在OO模型中通过消息把对象操作贯穿在一起

系统实现后这些操作将在一个控制流中顺序地执行

在并发系统中

控制流内部的消息

使系统中的每个控制流呈现出清晰的脉络

控制流之间的消息

体现了控制流之间的通信关系

OO模型需要表示消息的哪些信息？（按重要性排序）

- （1）对象之间是否存在着某种消息？
- （2）这种消息是控制流内部的还是控制流之间的？
- （3）每一种消息是从发送者的哪个操作发出的？是由接收者的哪个操作响应和处理的？
- （4）消息是同步的还是异步的？
- （5）发送者是否等待消息的处理结果？

要不要在类图中表示消息

以往不同的OOA&D方法有不同的处理方式 例如：

Coad/ Yourdon方法 ——在类图中表示消息

Booch方法——只在实例级的模型图（对象图和交互图）中表示消息

UML的处理方式：

不在类图中表示消息，只在协作图和顺序图中表示
理由：把类图定义为静态结构图，不表示动态信息

问题：

抽象级别问题




局部与全局问题

实际上类图中仍然包含动态信息

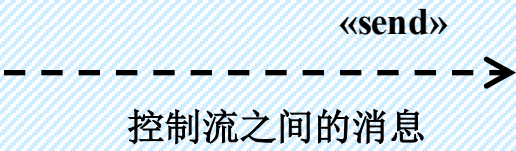
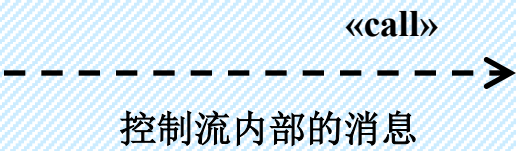
操作，调用（call）依赖

用什么符号表示消息

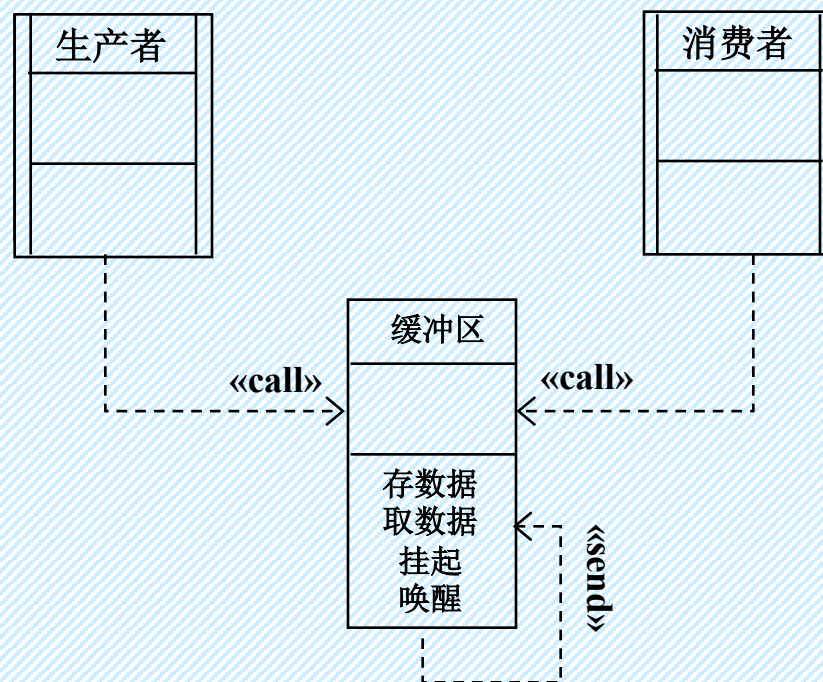
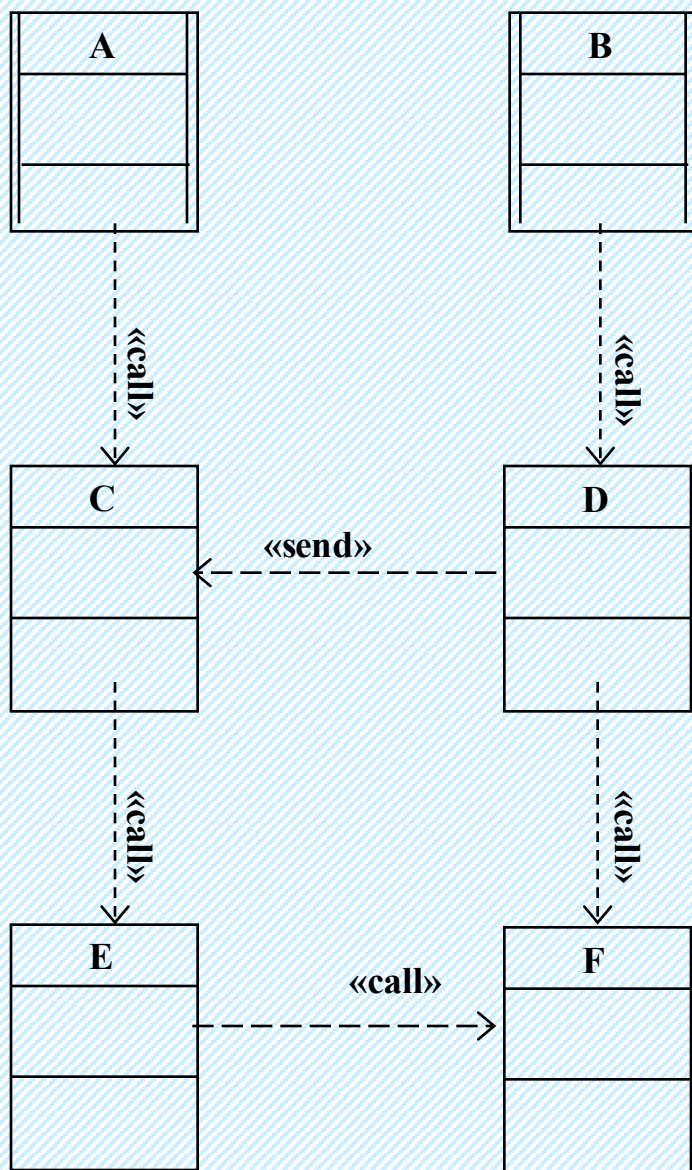
UML对各种箭头的用法

箭头种类	图形符号	用 途
实线开放箭头		关联的导航性（类图） 异步消息（顺序图）
虚线开放箭头		依赖（类图、包图、用况图、构件图） 从消息接收者的操作返回（顺序图）
实线封闭箭头		同步消息（顺序图、协作图）

借用依赖关系表示类图中的消息



例子:



如何建立的消息（控制流内部）

策略——“操作模拟”和“执行路线追踪”

(1) 人为地模拟当前对象操作的执行

考虑：需要其它对象（或本对象）提供什么服务

(2) 判断该消息是否属于同一个控制流：

- 二者应该顺序地执行还是并发地执行？
- 是否引起控制流的切换？
- 接收者是否只有通过当前消息的触发才能执行？

(3) 向接收者画出消息连接线，填写模型规约

上述工作进行到当前的操作模拟执行完毕

(4) 沿着控制流内部的每一种消息追踪到接收该消息的对象操作，重复进行以上的工作，直到已发现的全部消息都经历一遍。

针对每个主动类的每个主动操作进行上述模拟与追踪
检查系统中每个操作是否都被经历过

发现遗漏的消息或多余的操作

建立控制流之间消息

对每个控制流考虑以下问题：

(1) 它在执行时，是否需要请求其他控制流中的对象为它提供某种服务？

(2) 它在执行时是否要向其他控制流中的对象提供或索取某些数据？

(3) 它在执行时是否将产生某些可影响其他控制流执行的事件？

(4) 各个控制流的并发执行，是否需要相互传递一些同步控制信号？

(5) 一个控制流将在何种条件下中止执行？在它中止之后将在何种条件下被唤醒？由哪个控制流唤醒？

从上述各个角度发现控制流之间的消息
在相应的类之间画出消息连接线

什么是依赖（dependency）

在以往的OO方法中，只有**Firesmith**方法用到这个概念，其大意是：“**客户/服务者（client/server）**关系，表示**客户对服务者的依赖**。”列举的情况包括：

消息传送——其中客户发送消息给服务者；

聚合——其中聚合体（客户）的定义依赖它的构成部分（服务者）；

继承——其中派生类（客户）依赖它的基类（服务者）以继承其特征。

结论：在**Firesmith**方法中，依赖并不是对象之间的一种基本关系，而是为了指出在消息、聚合、继承等基本关系中哪个模型成分是客户（依赖者），哪个模型成分或服务者（被依赖者）所采用的一个概括性的术语。

UML1.4对依赖的定义和解释

“依赖：两个建模元素之间的一种关系，其中一个建模元素（独立元素）的一个改变将影响到另一个建模元素（依赖元素）。”

“依赖是除了关联、泛化、流以及元关系之外的关系的方便术语。”

“依赖表明一个或者一组元素的实现或者功能需要另外一个或者一组元素出现。”

“依赖指出了两个模型元素（或者两组模型元素）之间的语义关系。它指的是这些模型元素本身，而不需要一组实例来说明其含义。它指出这样一种情况：目标元素的一个变化可能需要依赖中的源元素发生变化。”

Booch等《UML用户指南》的解释

“两个事物之间的语义关系，其中一个事物（独立事物）的改变将影响到另一个事物（依赖事物）。”

UML2对依赖关系的新闻述

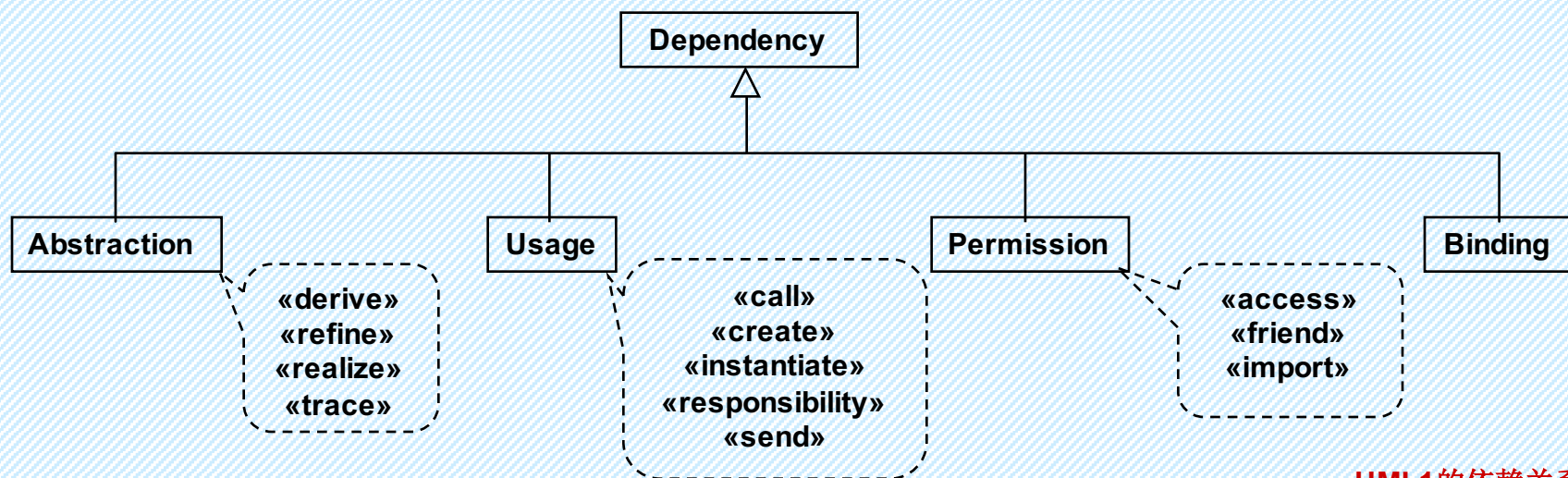
“依赖表明模型元素之间的供方/客方（**supplier /client**）关系，其中供方的修改可能影响到客方元素。依赖意味着，如果没有供方，客方的语义就是不完整的。依赖在一个模型中出现并不含有任何运行时的语义，它完全是以参与这种关系的模型元素的名义而不是以其实例的名义给出的。”

“依赖是这样一个关系，它表明一个或者一组模型元素的详细说明或者实现需要另外一些模型元素。这意味着，依赖元素的完整语义在语义和结构上都依赖这些供方元素的定义。”

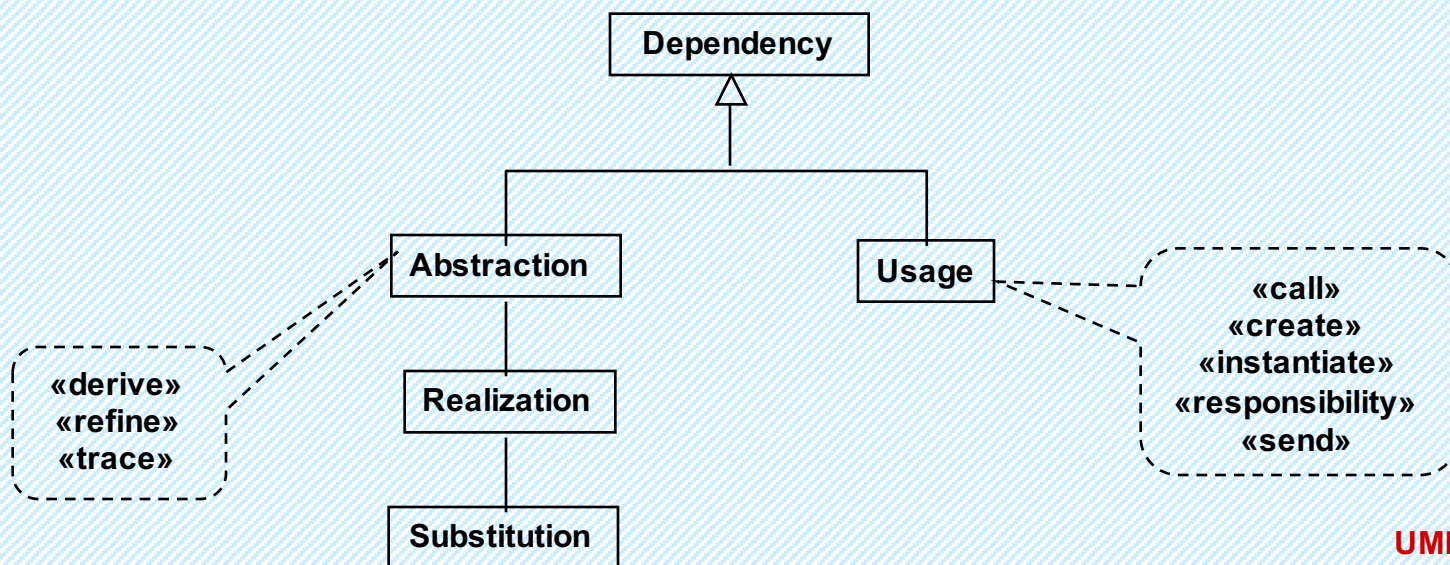
认识：

1. 依赖着眼于表达纸面上的模型元素之间的关系，而不是这些元素所描述的客观事物在问题域中的固有关系。
2. 依赖基本上不能视为一种面向对象的概念。

依赖所指的并不是单独的一种关系，而是包含了许多情况



UML1的依赖关系



UML2的依赖关系

依赖的表示法以及其他外观类似的表示

①

作为依赖关系标准
衍型的关键词

«instantiate»
«create»
«call»
«derive»
«refine»
«responsibility»
«send»
«trace»

②

用于依赖但未
明确称为标准
衍型的关键词

«abstraction»
«realize»
«substitute»
«permit»
«use»

③

附加于虚线开放
箭头但不称为依
赖关系的关键词

«access»
«apply»
«bind»
«extend»
«import»
«include»
«manifest»
«occurrence»
«represents»

各种标准衍
型和关键词
的含义，查
阅教材

«关键词»

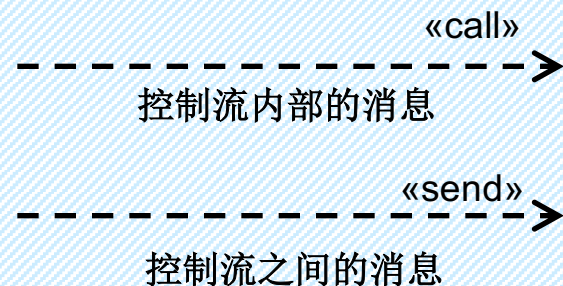
----->

依赖关系对面向对象建模的作用

继承、聚合、关联、消息这四种关系中都已经蕴涵了依赖的含义，不需要再用依赖关系再做重复的表示

除了上述关系之外，在OO建模中没有多少重要的信息必须用依赖关系表达

由于UML没有在类图中提供消息的表示法，可以借用依赖关系来表示类图中的消息



告戒：避免建立语义含糊不清的依赖关系，更要避免用这些含糊不清的依赖关系代替含义明确的OO关系。
建立一种依赖关系，就要具体地指出它是一种什么依赖。

9.1 类图与其他模型图之间的关系

历史上OO方法采用其他模型图的三种不同情况
掺杂了其他方法

OMT 对象模型+动态模型+功能模型
解决不同阶段的问题

OOSE 需求模型+健壮模型+设计模型
以面向对象方法为核心，以其他模型图作为补充

Booch方法 基本模型+补充模型

Coad/Yourdon 类图+流程图

UML的状况和发展趋势

收集了大量的模型图，从9种发展到13种

从不同的视角对复杂系统建模

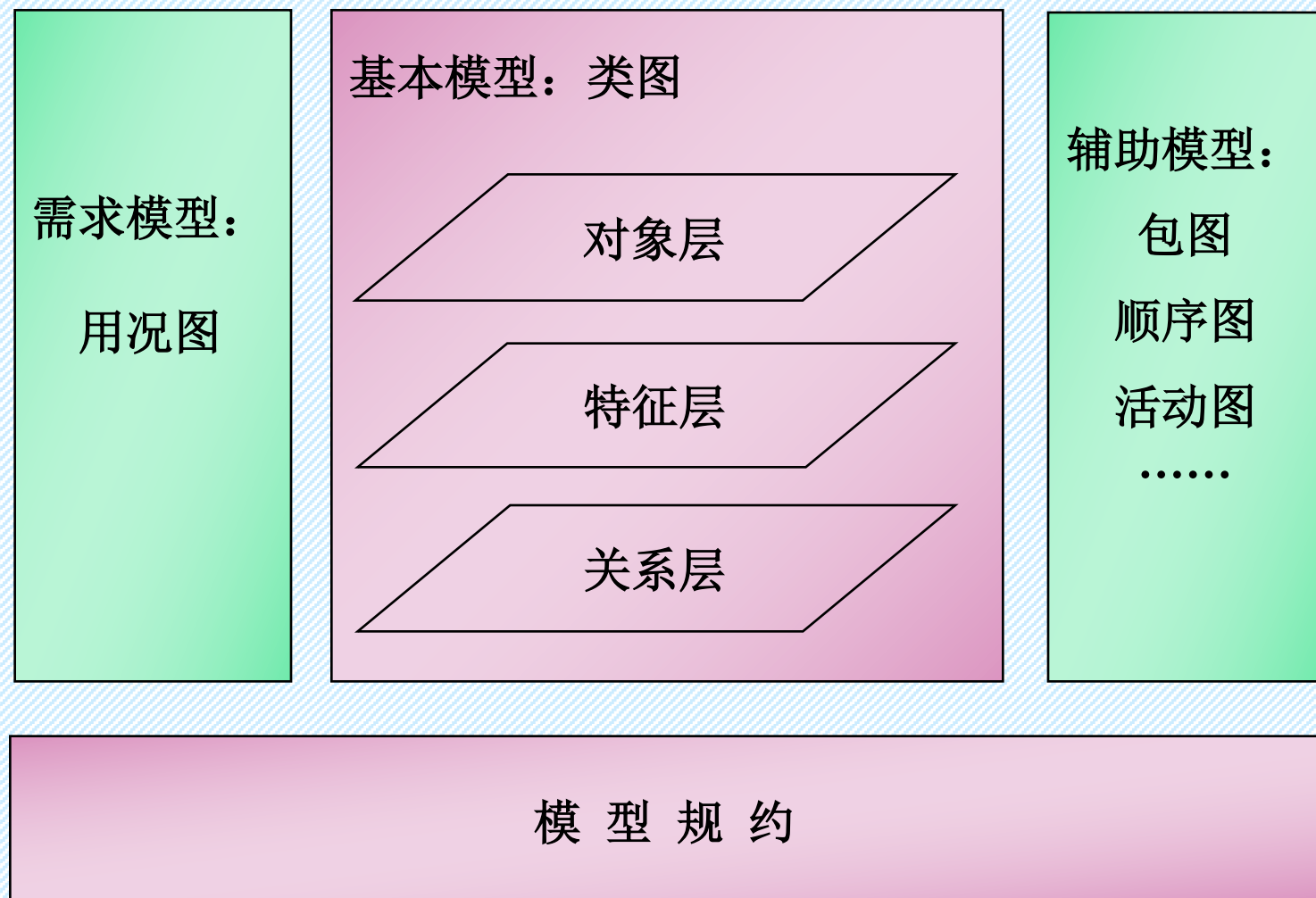
各种图向着健全和复杂的方向发展

面向对象建模方法中

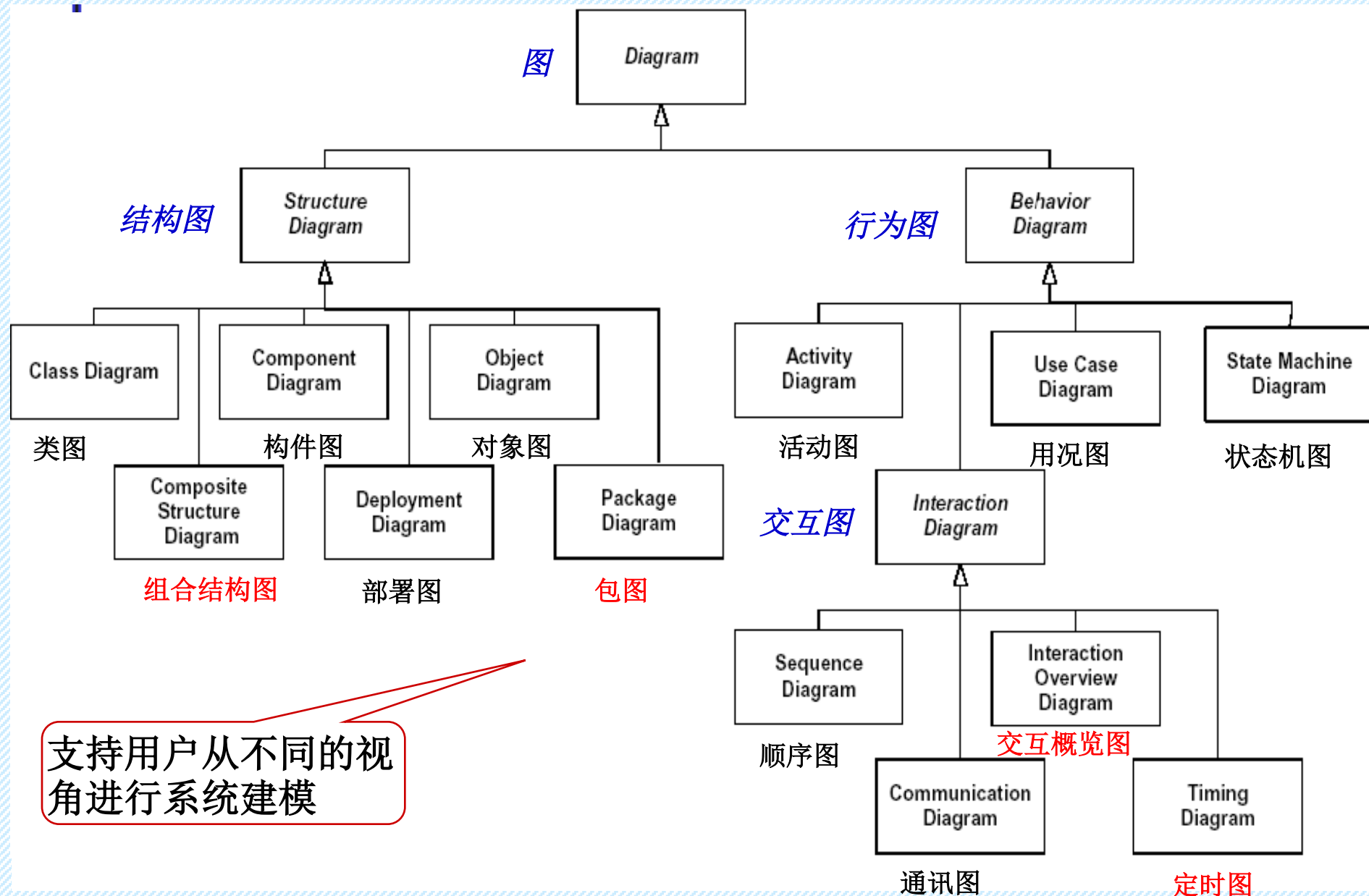
以类图作为主要模型——基本模型

用况图作为需求模型

其他模型图作为辅助模型



UML2中的各种图



各种UML模型图的作用

类图：基本模型，是面向对象的建模最重要的模型，必不可少。

用况图：需求模型，是开展面向对象建模的基础，提倡尽可能使用。

包图：辅助模型，各种模型图的组织机制，系统规模较大时使用。

顺序图：辅助模型，清晰地表示一组对象之间的交互，对类图起到补充作用。在交互情况较复杂时使用。

活动图：辅助模型，可描述对象的操作流程，也可描述高层的行为。

状态机图：辅助模型，对于状态与行为复杂的对象，可描述对象状态及转移，以便更准确地定义对象的操作。

构件图，部署图：辅助模型，在转入实现阶段之前，可以用它们表示如何组织构件以及如何把软件制品部署的各个结点（计算机）上。

组合结构图、交互概览图、定时图：都可以作为辅助模型，无强烈建议。

对象图、通信图：建议不使用。

9.2 包图 (package diagram)

包 (package) 是一种将其他模型元素组织起来，形成较大粒度的系统单位的通用机制。

基本思想：

从不同的粒度描述和观察系统——从宏观到微观

G. Miller 的 “ 7 ± 2 原则”

注意：

包是一种组织机制而不是一种基本模型元素

包可以嵌套

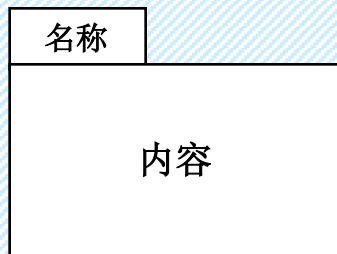
包中的模型元素应具有某种意义的内在联系

包的划分有一定的灵活性或随意性

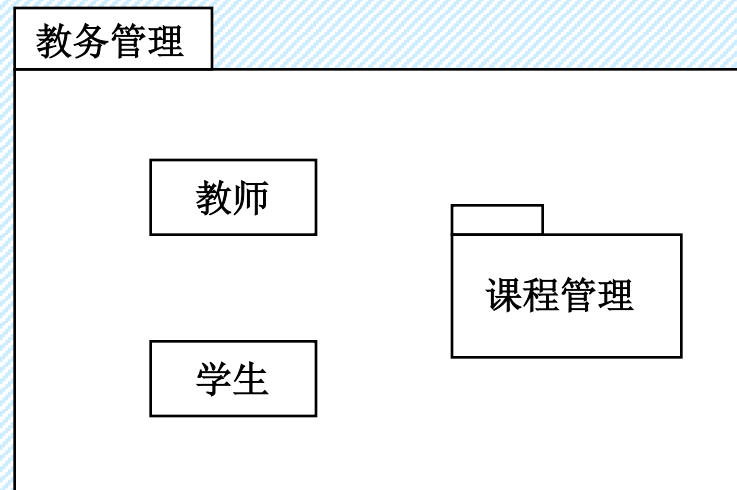
包的表示法



包的压缩方式



包的展开方式



一个包的例子

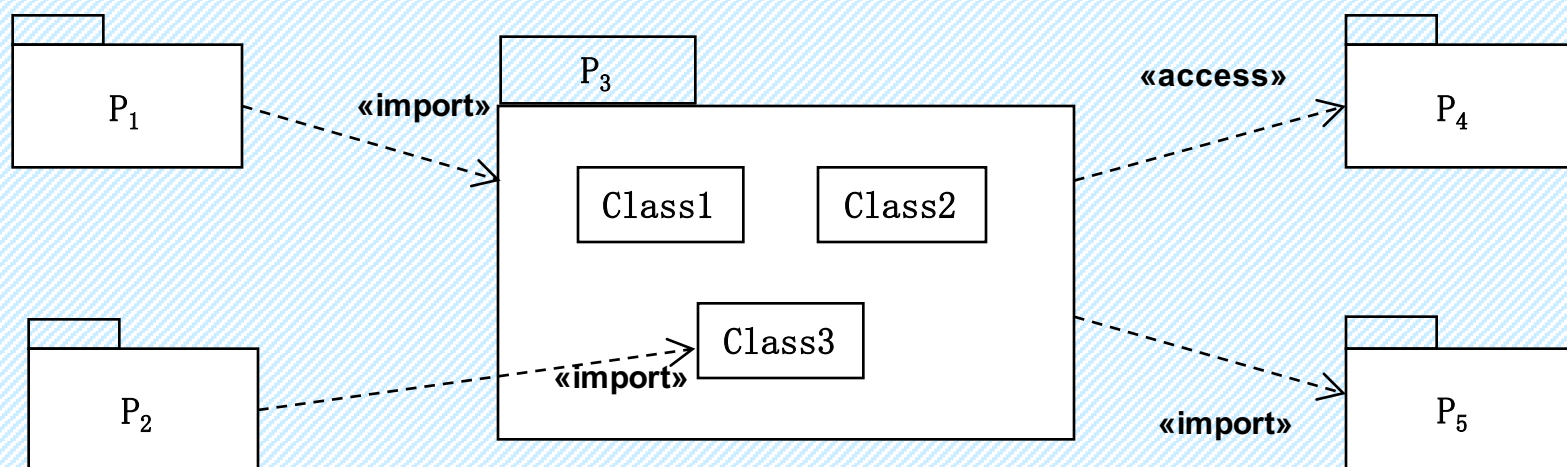
包之间的关系及表示法

引入（**import**）是包之间的一种依赖关系，表明源包中的模型元素能够**直接**引用目标包中的模型元素。

原由——名字空间（为了避免命名冲突）

以包为单位划分名字空间，引用时给出路径名

引入关系表明可以直接引用

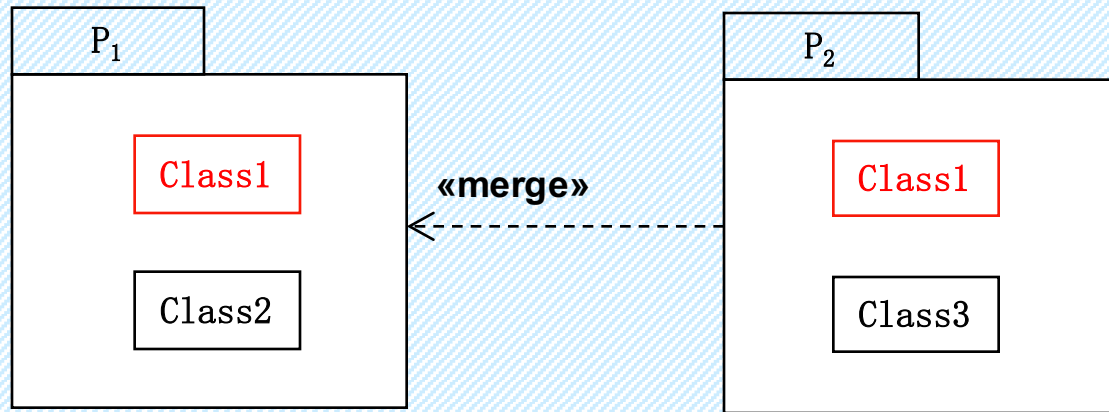


访问（**access**）关系与引入关系类似，二者之间的差别：

UML1: 目标包的元素是否可以被引入源包的其他包引入

UML2: 目标包（或目标元素）的可见性不同采用不同的关系

合并（merge）是包之间的一个有向关系，表明目标包的概念定义被合并到源包中。



在UML自身的定义中经常被使用
对大部分应用系统而言，这种层层追加的定义方式并不常见，也不值得提倡。

包之间的其他关系

泛化——没有正式的定义，只在包图的某些例子中出现
依赖——不加任何关键词的依赖关系，没有确切的定义
见讲义9.2.2节（其中第4条）的介绍

如何建立包图

1、将模型元素打包（类图为例）

根据类图中的各种关系：

一般-特殊结构

整体-部分结构

关联

消息

其他关系

根据问题域和用况：

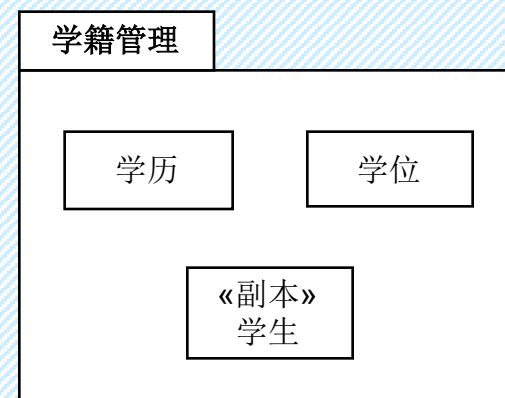
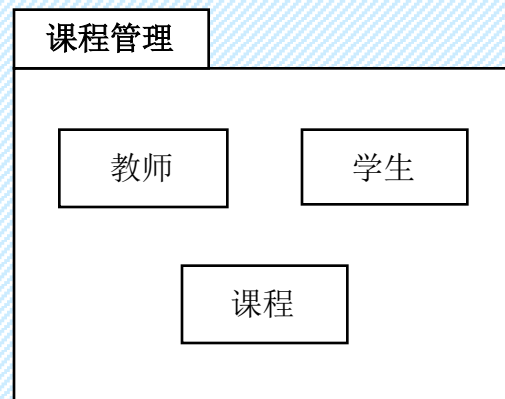
对象来源

功能类别

通信频繁程度

并发和分布情况

包的内容交叉问题



2、包的命名

一般-特殊结构中的根类

整体-部分结构中的整体对象类

其他：事物、功能、来源

3、组织嵌套的包

7±2 原则

将若干低层包合并为高层包

将低层的包与零散模型元素组织到高层的包中

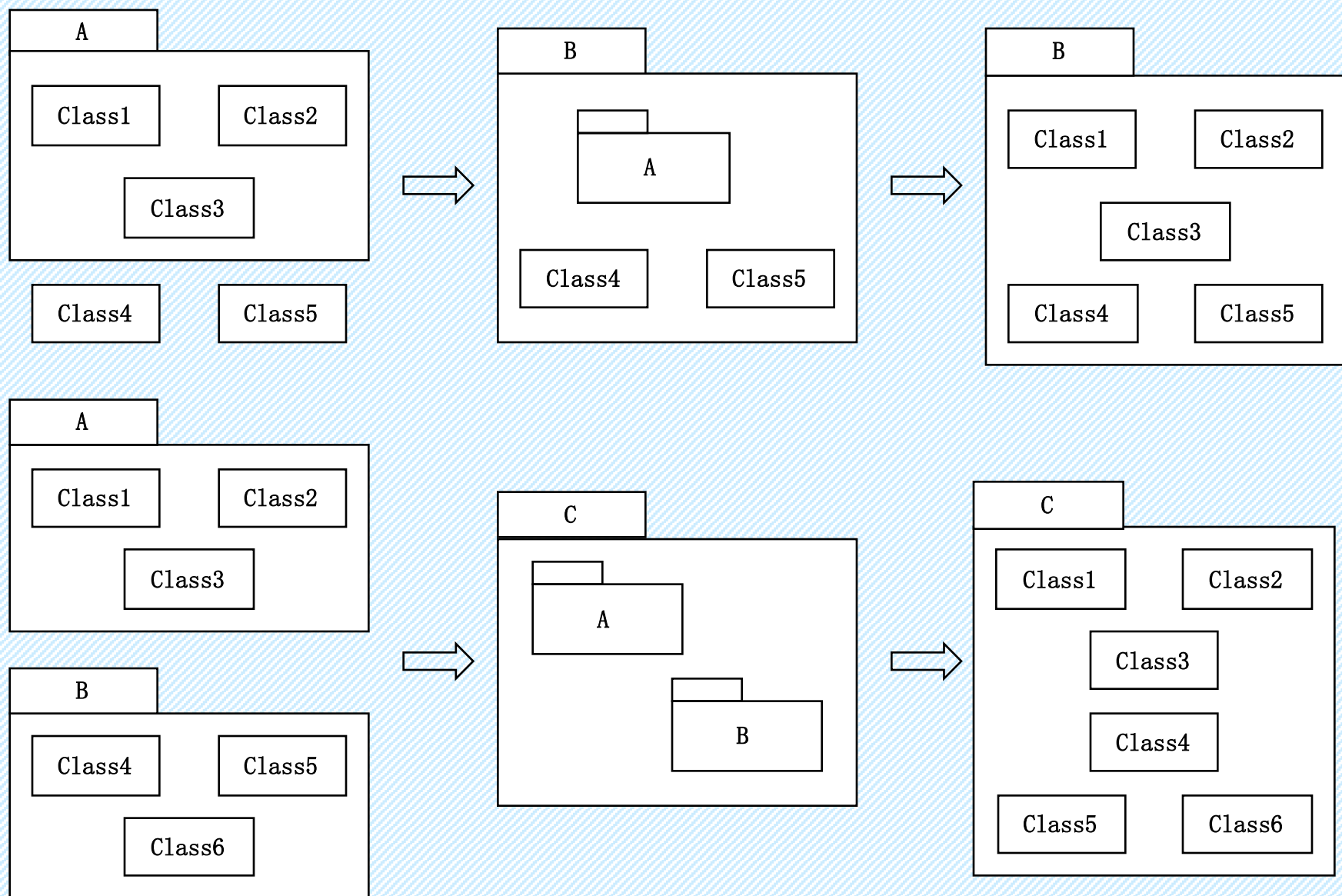
合并的依据

参照将模型元素打包的考虑因素

包之间内容是否有交叉

包之间的关系是否紧密

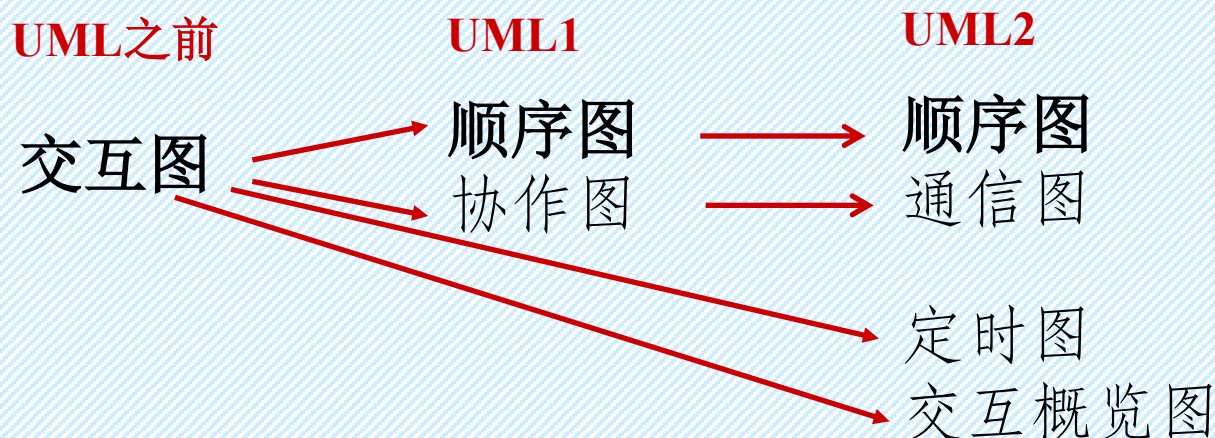
4、减少包的嵌套层次



9.3 顺序图 (sequence diagram)

顺序图是一种详细地表示对象之间行为关系的图。它按时间顺序展现了一组相互协作的对象在完成一项功能时所执行的操作，以及它们之间所传送的消息，从而清晰地表示对象之间的行为关系以及操作和消息的时序关系。

名称的演变：



适应范围：通常只适合表示一组相互协作的对象执行一项功能时的交互情况，包括外部可见的功能和内部功能。难以表示整个系统的交互情况。

主要概念及表示法

对象
生命线
实体

:类名

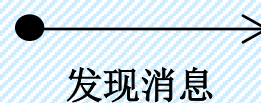
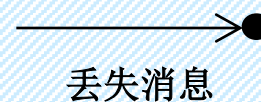
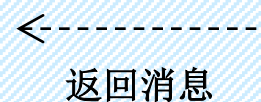
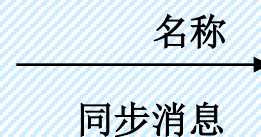


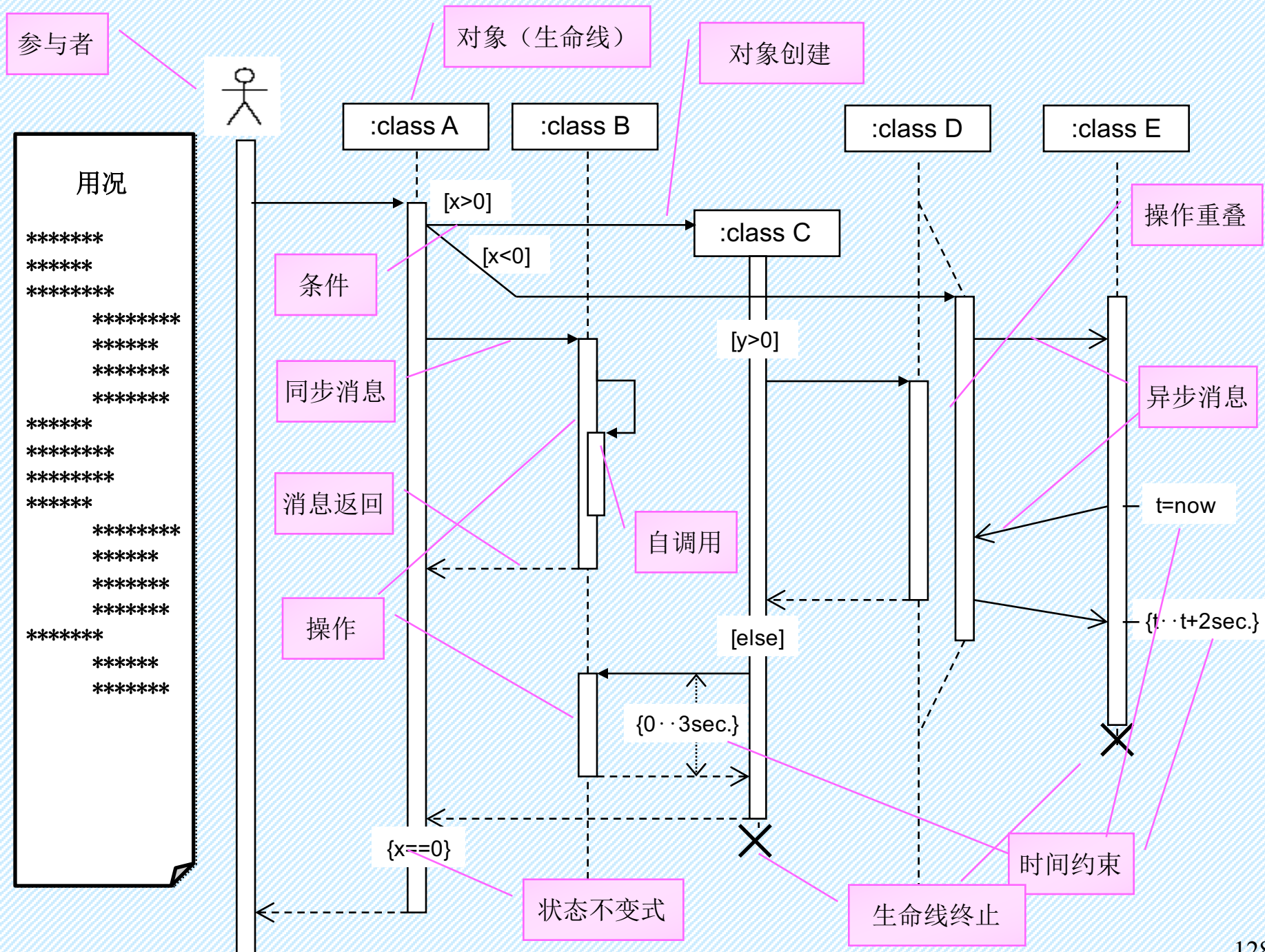
操作
(控制焦点)
执行规约

:类名



消息
(激发)





组织机制与复用

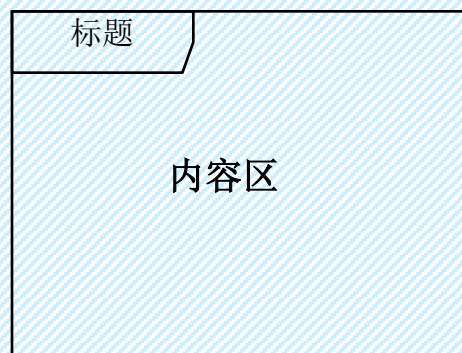
帧（frame）：即一幅图，不仅用于顺序图，也可以在其他多种图中使用，特别是各种交互图。

交互片段（interaction fragment）：交互中的一个片段

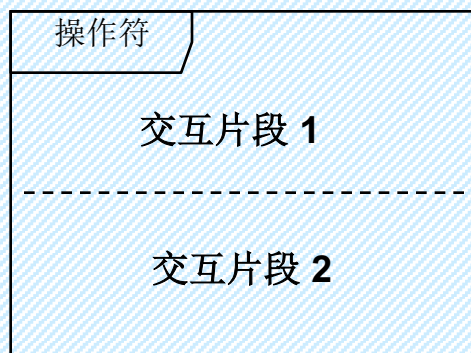
组合片段（combined fragment）：若干交互片段的组合

引用（reference）

交互使用（interaction use）



帧

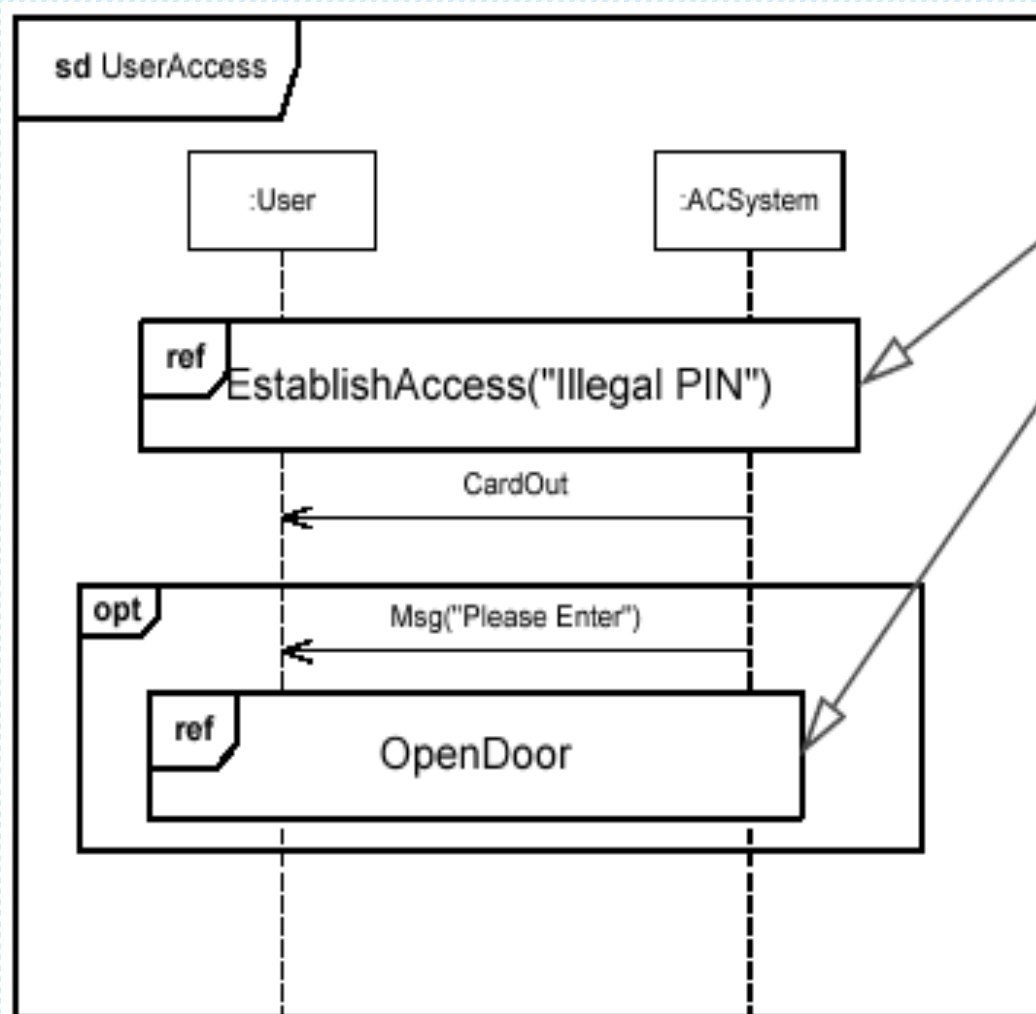


组合片段



引用（交互使用）

例子



交互使用

如何建立顺序图

1. 决定为系统建立哪些顺序图
基本以用况为单位，但是不绝对
简单的用况不必用顺序图描述
系统内部的功能也可以用顺序图描述
2. 确定参加交互的对象和参与者
确定参加交互的参与者
找出与参与者直接交互的对象
以消息为线索，找出与交互有关的全部对象
3. 顺序图的绘制（看书）

9.4 活动图（activity diagram）

活动图是一种描述系统行为的图，它把一项行为表示成一个可以由计算机、人或者其他执行者执行的活动，通过给出活动中的各个动作以及动作之间的转移关系来描述系统的行为。

UML1曾经把活动图称为状态图的变种，**UML2**放弃了这种说法。

活动图起源于流程图（**flow chart**），同时借鉴了工作流、**Petri**网等领域的若干概念，使其表达能力比流程图更强，应用范围也更宽。

活动图可以描述各种不同的行为，例如：

顺序执行的过程 — 并发执行的过程

一个对象的操作 — 多个对象协同完成的一项功能

全由计算机完成任务 — 有人员参与的业务流程

活动图的表达能力要比流程图强得多。

主要概念及表示法

在有些图中也称为顶点（vertex）和弧（arc）

活动图由结点（**node**）和边（**edge**）两种基本元素构成

活动结点——动作、判断、合并、分岔、汇合、起点、结束

活动边——控制流和对象流

动作与活动

动作（action）是活动的基本构成单位，被看作一种原子的构造成分。

活动（activity）是由一系列动作构成的，是对一项系统行为的描述，它不是活动图的模型元素，而是一个整体概念，对应着整个活动图。

如果要展开一个动作内部的细节，则：

定义为“子活动”——**UML1**

定义为“调用行为”动作——**UML2**

动作名称

一般动作

*
动作名称

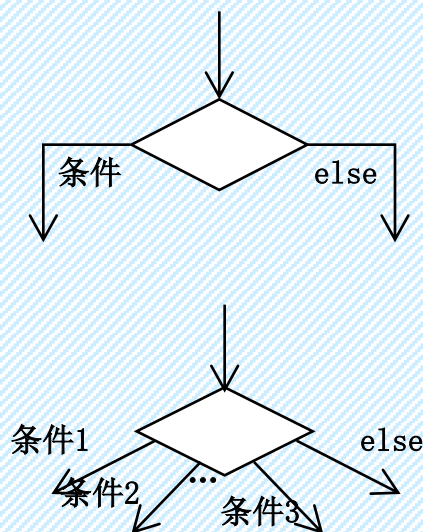
重复动作

判断与合并

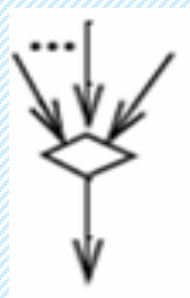
——是一对控制结点

判断 (decision) 表示执行到这一点时将判断是否满足某些条件，以决定从不同的分支选择下一个动作。

合并 (merge) 表示把多个分支合并到一起。



判断



合并



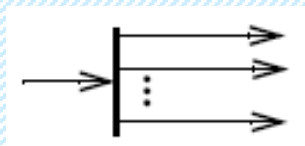
合并与判断结合

分岔与汇合

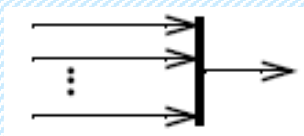
——另一对控制结点，用来表示并发行为

分岔 (fork) 表示一旦前面的动作结束而流入这个结点，它的每个流出边所指的动作都可以执行。

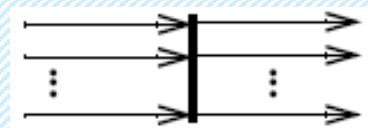
汇合 (join) 表示汇合点之前有多个控制流在汇合点上需要取得同步，并汇合为一个控制流。



分岔



汇合



汇合与分岔结合使用

判断以下说法是否正确：

M. Fowler: “分岔与汇合必须匹配。在最简单的情形，这便表示，当你有了一个分岔，就必须有一个汇合，后者把由该分岔开动的各个线程汇聚在一起。”

G. Booch: “汇合和分岔应该是平衡的，即离开一个分岔的流的数目应该和进入它所对应的汇合的流的数目相匹配。”

起点、活动结束和流结束

起点 (initial node) 表示由一个活动图所描述的整个活动的开始;

活动结束 (activity final) 表示活动图所描述的整个活动到此终结;

流结束 (flow final) 表示活动图中一个控制流的终结, 但并不是整个活动终结。



起点



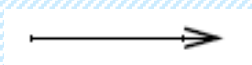
活动结束



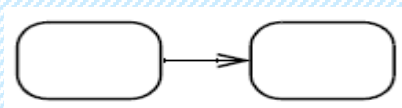
流结束

活动边

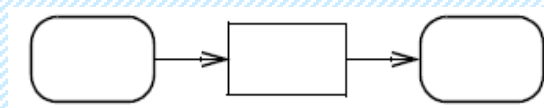
连接两个活动结点的有向边称为**活动边 (activity edge)**
包括**控制流 (control flow)** 和**对象流 (object flow)**



控制流

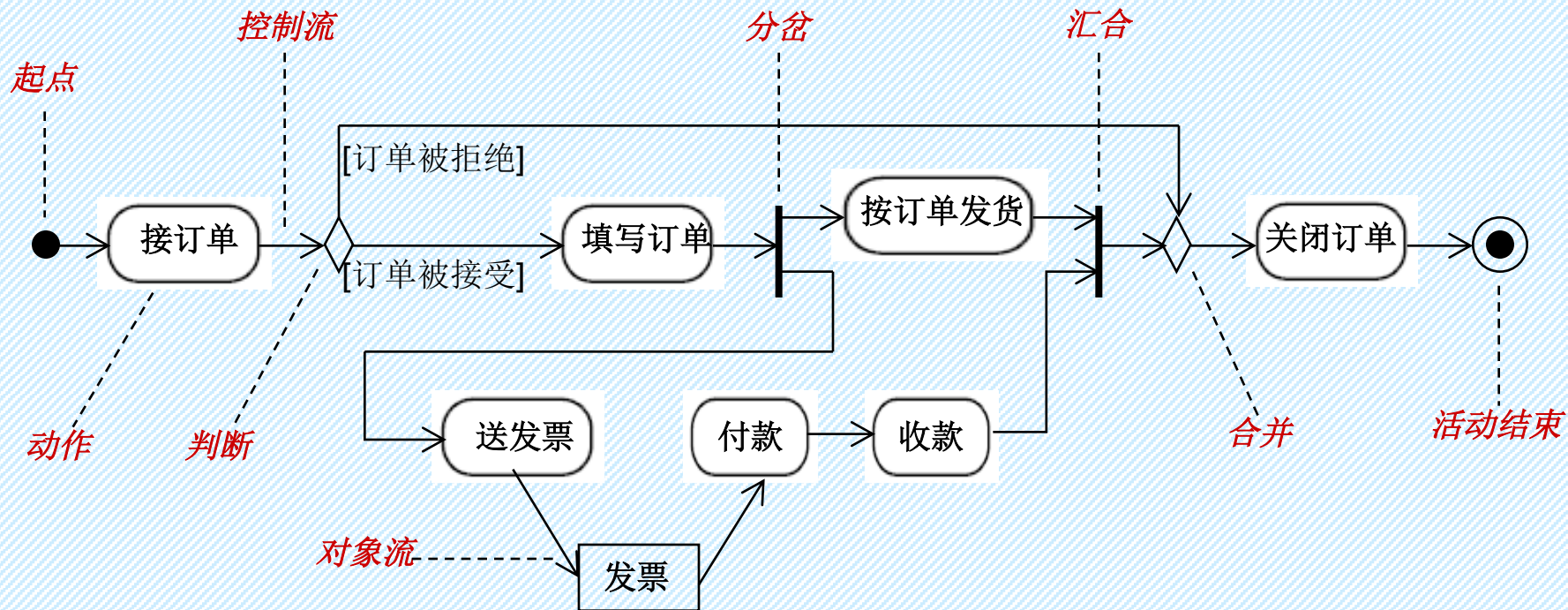


控制流及其连接的结点



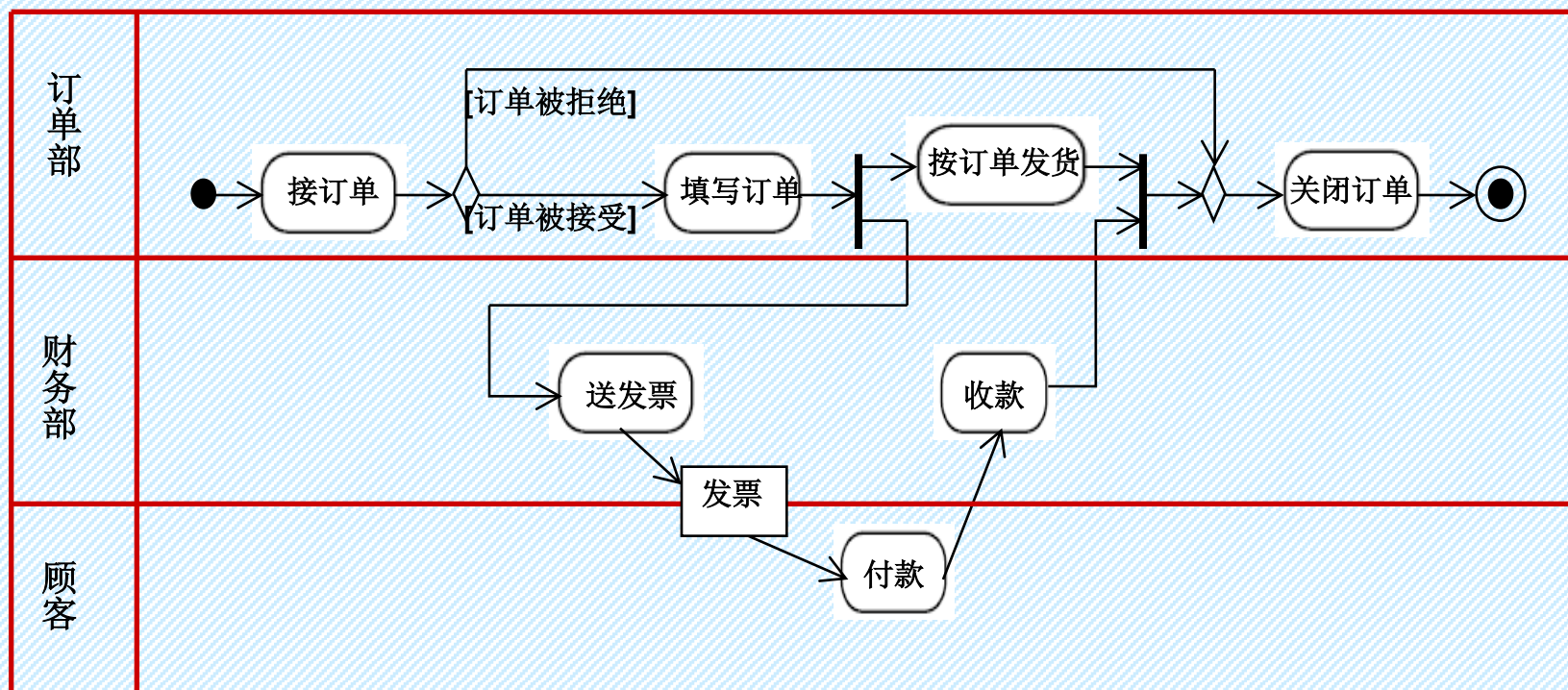
对象流

一个活动图的例子



泳道 (swim lane)

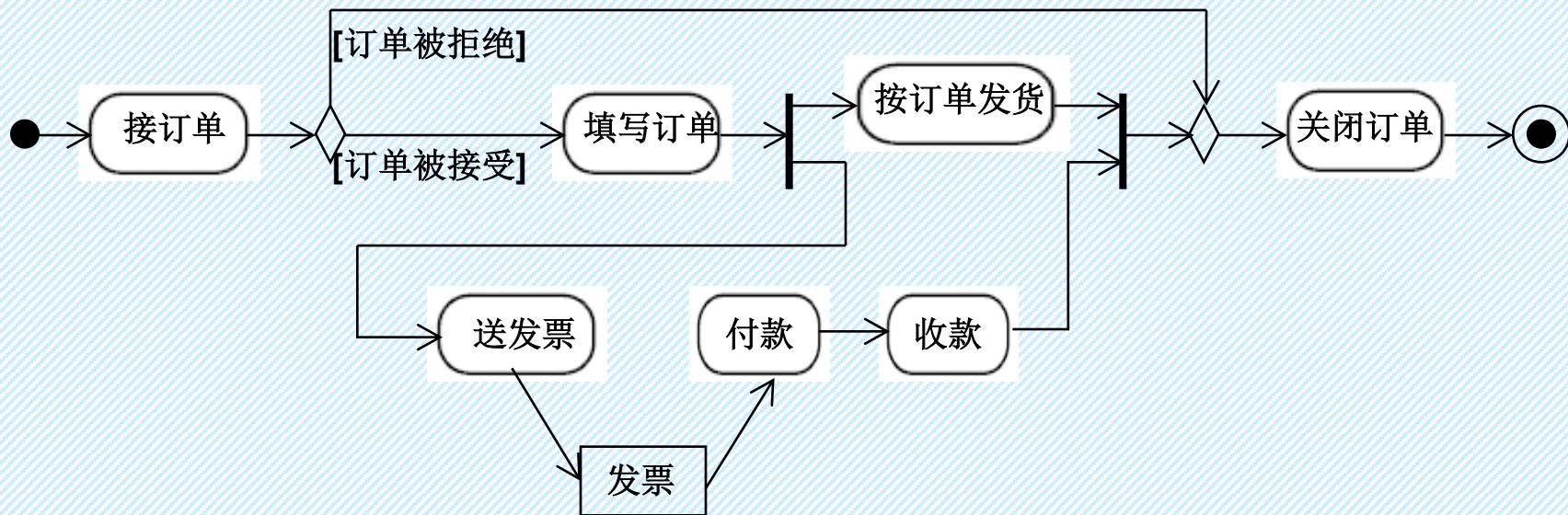
一种辅助机制，其作用是把活动图中的各个动作划分到与它们的执行者相关的若干区域中，从而清晰地表现出不同的执行者分别执行了哪些动作。



问题讨论

业务流程和执行过程的差异
并发描述上的误差
对象流问题
复杂性问题

UML2定义了大量的专用动作，包括：
发送信号动作、接受事件动作、
调用动作、接受调用动作、调用行为动作、调用操作动作、添加变量值动作、读变量动作、写变量动作、
创建对象动作、创建链动作、读链动作、读链对象端点动作、读结构特征动作.....



如何使用活动图

描述对象的操作流程

未必每个操作

未必十分详细

描述系统某些局部的行为

判断是否真正必要

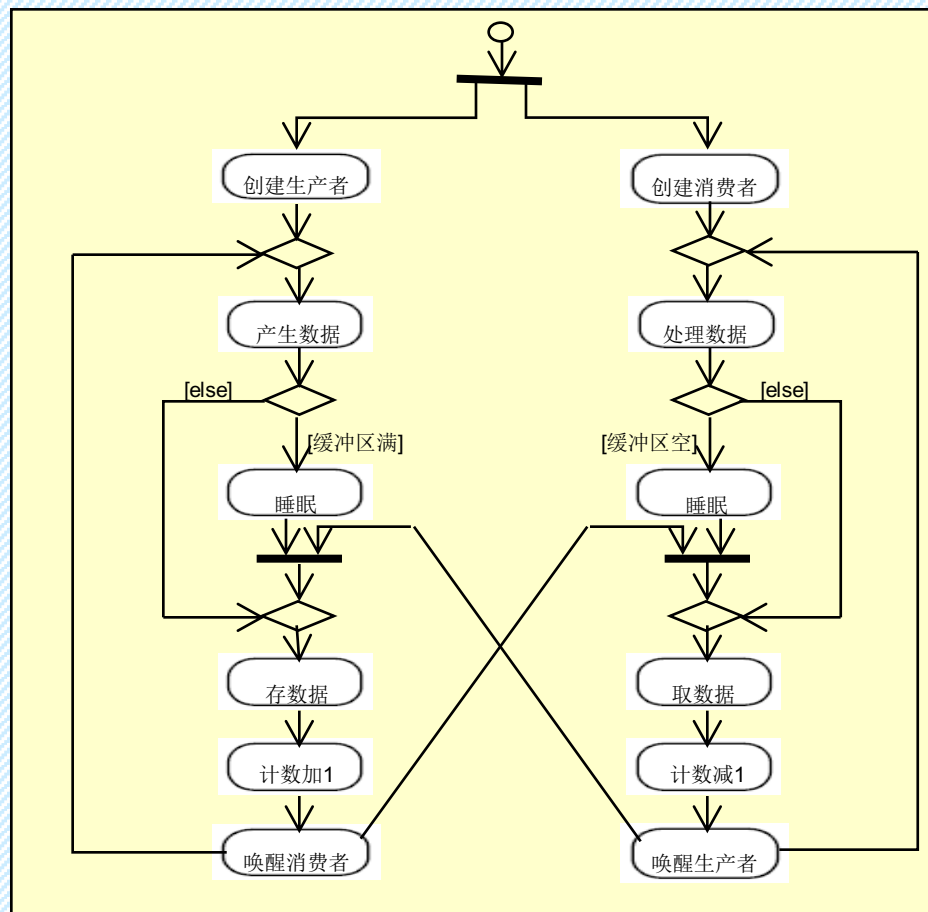
描述系统外部可见的行为

实际上是描述用况

如果用文字更清楚就用文字

描述系统的业务流程

注意业务流程和执行过程的差别和并发描述的误差



9.5 状态机图 (state machine diagram)

状态机图：是一种描绘系统中的对象（或者其他实体）在其生命期内所经历的各种状态，状态之间的转移，发生转移的动因、条件及活动的模型图。

别称：状态图 (state chart)

状态转移图 (state transition diagram, STD)

状态建模：通过分析系统（或其局部）所经历的状态和状态之间的转移，用状态、转移等概念来建立系统模型。

在某些领域可以作为一种独立的建模方法

在面向对象建模中可以起到一种辅助作用

长处：对状态复杂多变，并且在不同状态下呈现不同行为的对象，通过状态建模将有助于准确地认识和描述对象的行为。

局限性：一个状态机图通常只适合描述系统中一个或少数几个对象的状态及其转移情况，很难用于描述整个系统。

主要概念及表示法

状态 (state)

UML：“对象生命期中的一种条件或者情形，在此期间它满足某些条件，执行某些活动，或者等待某些事件。”

“状态是对一种状况的模型表示，在此期间保持了某些（通常是固有的）条件。”

《对象技术词典》的定义

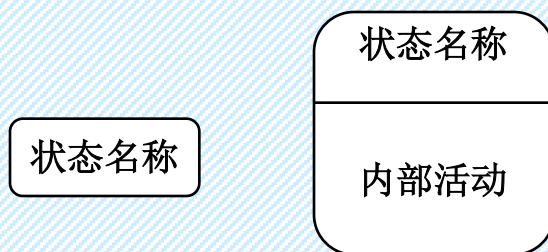
对象或者类的所有属性的当前值。

对象或者类的整体行为（例如响应消息）的某些规则所能适应的（对象或类的）状况、情况、条件、形式或生存周期阶段。

状态太多

识别状态
等价类

表示法



压缩方式

展开方式

转移 (transition)

“源顶点和目标顶点之间的一个有向的关系。”

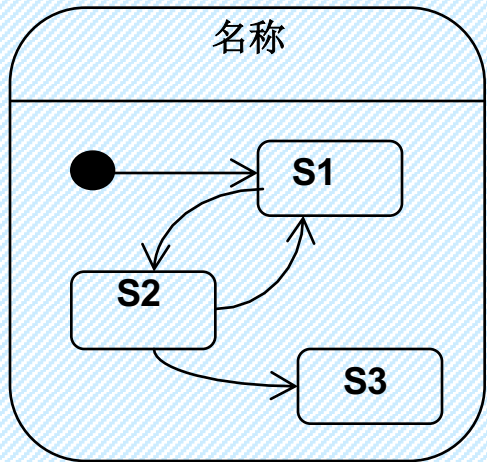


对转移的形式化描述:

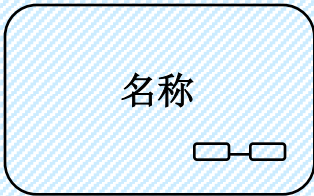
**<transition> ::= <trigger> [',' <trigger>]*
['[' <guard-constraint>']'] ['/ <behavior-expression>]**

组合状态 (composite state)

由若干状态组织在一起所形成的状态成为组合状态。
包含在组合状态内部的状态称为子状态
内部不包含其他状态的状态称为简单状态



展开方式



压缩方式

组合状态的作用：
使模型更清晰
便于复用
简化图的绘制

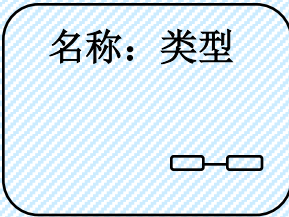
子状态机 (submachine)

可以作为一个状态单元在上层状态机中使用的状态机

子状态机状态 (submachine state)

作为一个状态单元使用 的子状态机

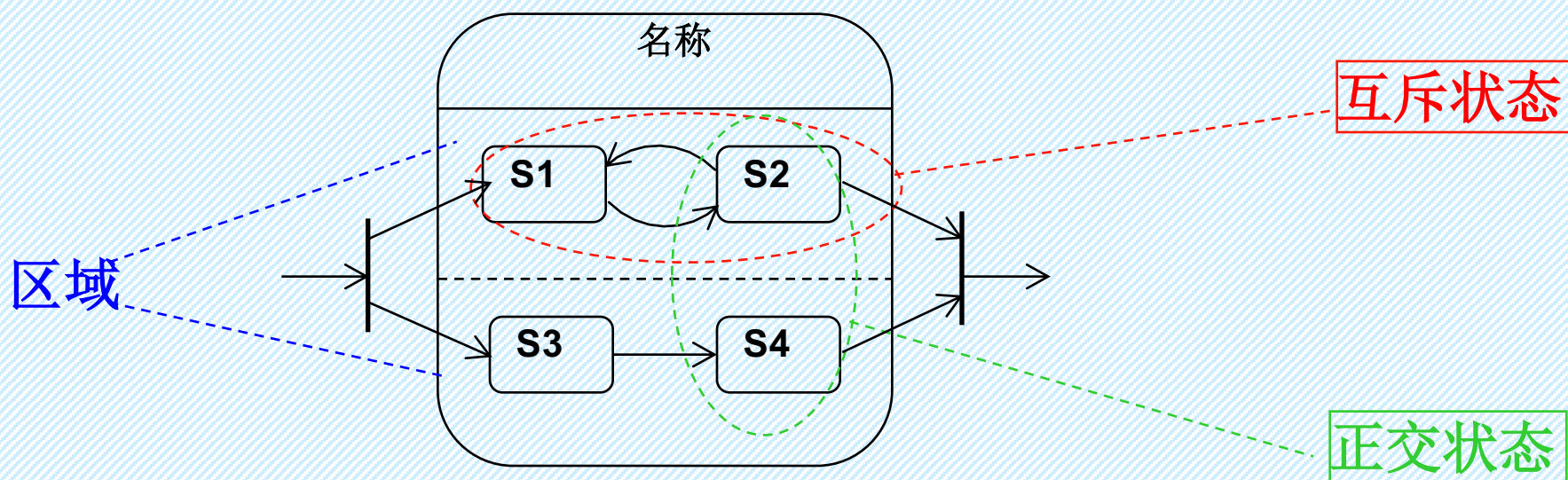
表示法



区域 (region)

互斥状态 (mutually state)

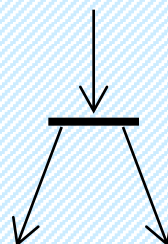
正交状态 (orthogonal state)



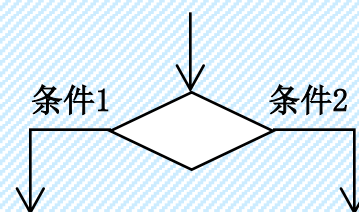
作用：表示并发转移

伪状态 (pseudo state)

伪状态实际上并不是一种状态，只是为了加强状态机图的可视化效果而引入的一些图形符号，都是结点（顶点）型的图形成分。



分岔

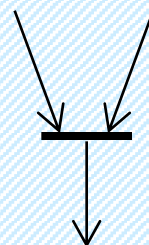


选择

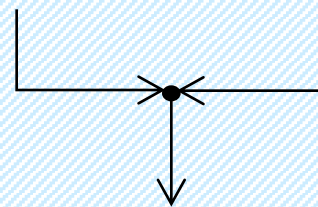


进入点

浅历史



汇合



接合



终止结点



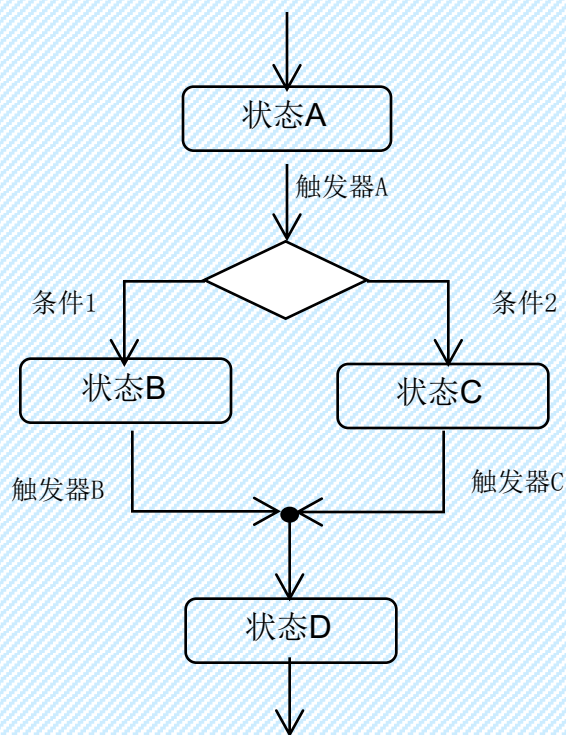
退出点



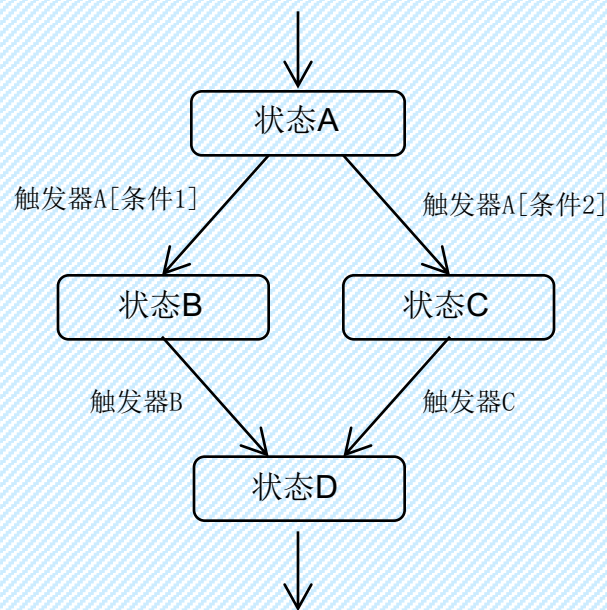
深历史

最终状态 \odot 和 终止结点 \times 的区别

选择 与 接合的用途 及 等价的表示法



用选择和接合表示



用监护约束表示

如何使用状态机图

在某些领域中，曾经以状态建模作为主要建模手段
在OO建模中，以状态机图作为认识对象行为的辅助手段

(1) 仅对状态对行为影响复杂的对象建立状态机图

(2) 使用状态机图的主要概念

状态、转移、组合状态、伪状态.....根据需要选用

(3) 识别对行为有不同影响的对象状态，达到如下效果：

在不同的状态下对象将呈现不同的行为规则

同一种状态下对象的行为规则是始终一致的。

(4) 认识和描述转移

触发器、监护约束、行为表达式

(5) 为类图提供有用的信息

状态——属性；转移——操作

9.6 构件图 (component diagram)

构件图是一种表示构件的组织结构和相互关系的图，用于表达在实现时如何将系统元素组织成构件，从而支持以构件为单位进行软件制品的实现和发布。

UML1的构件图没有太多地反映当时构件技术的发展，它只是着眼于把软件的逻辑蓝图转化为比特 (**bit**) 世界中的事物。

UML2为构件图增添了许多内容，能够表示构件技术领域的大部分常用的概念。

主要概念及表示法

构件（component） “一个构件表示系统中一个模块部件，它封装了它的内容，而其表现形式在其环境中是可替换的。”

可复用构件的简称，特指按某种构件标准设计，并可在多个系统中重复使用的软件构造块。

接口（interface） “接口是一种类目，它表示对一组紧凑的公共特征和职责的声明。一个接口说明了一个合约；实现接口的任何类目的实例必须履行这个合约。”

供接口（provided interface）——实现的接口

需接口（required interface）——使用的接口

端口（port） “类目的一个特征，指出类目与外部环境之间或者与内部的部件之间的一个明显的交互点。”

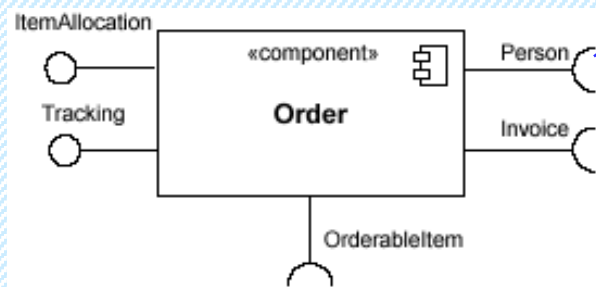
向内，可以连接到与外部交互的内部成分

向外，连接到供接口或者需接口，并因此而分别称为

供端口（provided port）

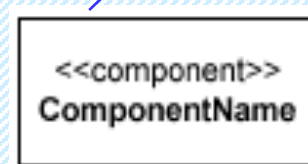
需端口（required port）

构件的表示法



压缩方式：给出构件的名字和对外接口，不显示其内容

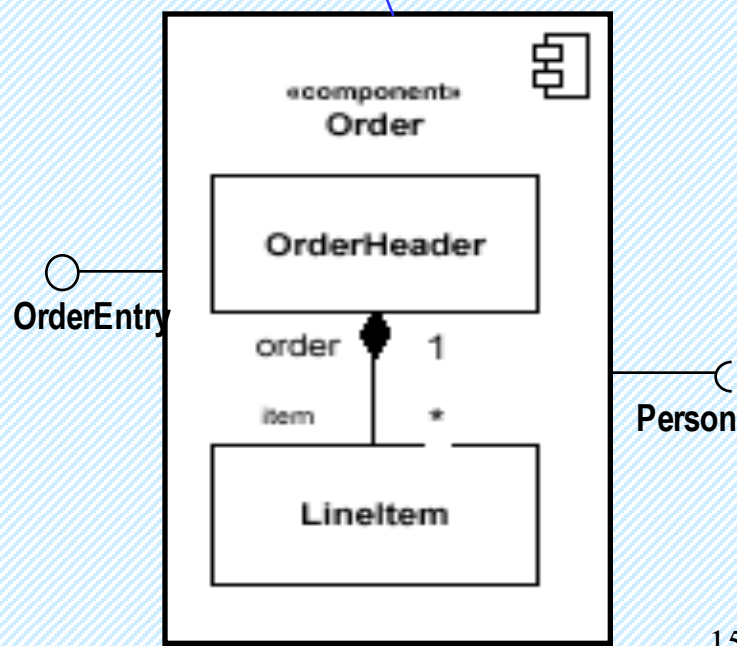
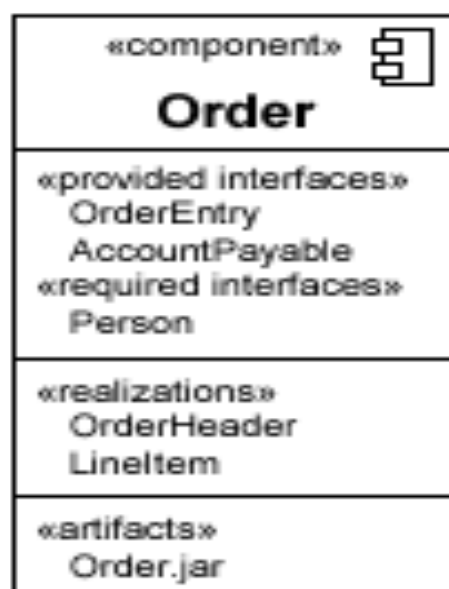
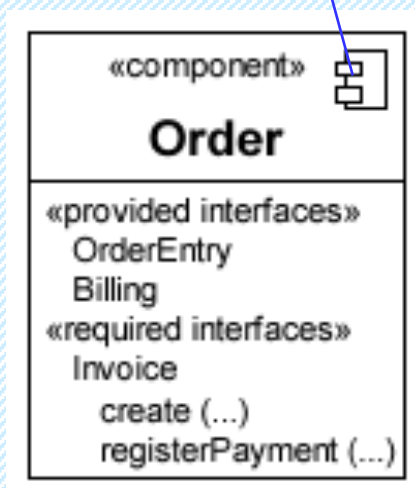
把构件表示为
衍型化类



展开方式：以图形的方式显示构件内部成分及其相互关系

白箱方式：列出构件接口、所包含的类，以及对应的软件制品

黑箱方式：在内部以列表的方式列出其对外接口，不显示内容



主要概念及表示法（续）

实现（realization）和使用（use）

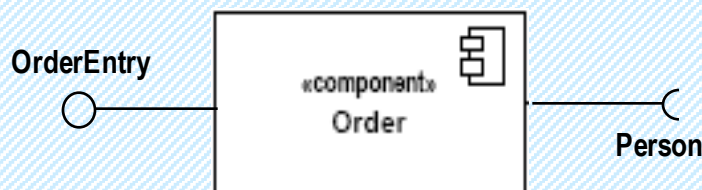
用于构件与它们的接口之间

构件与供接口之间为**实现**关系

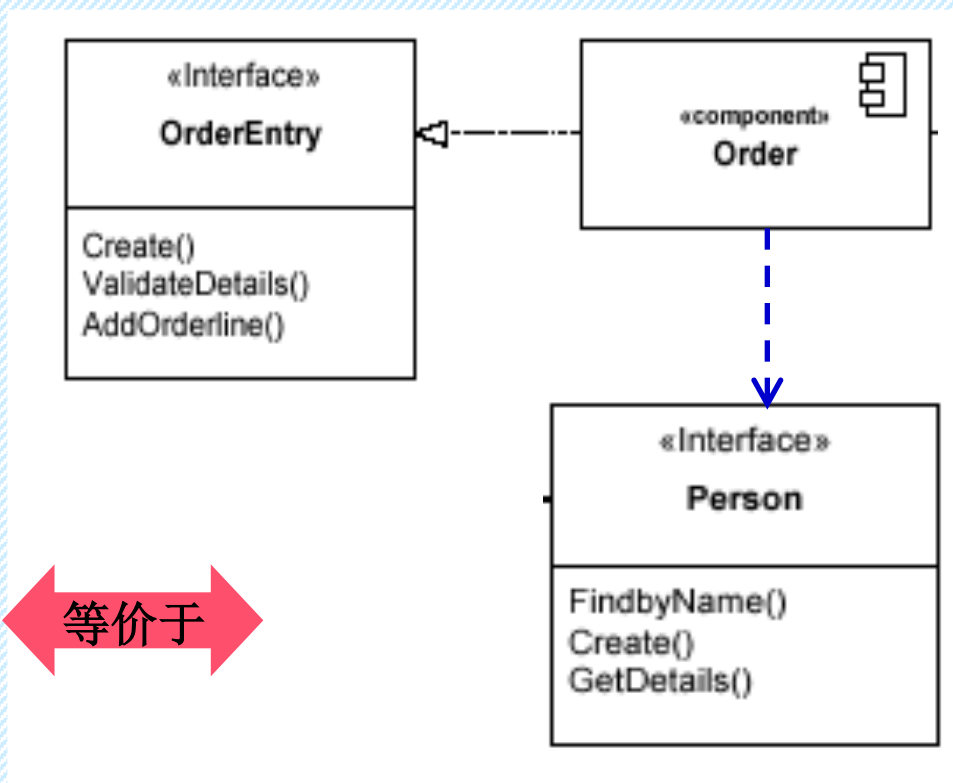
构件与需接口之间为**使用**关系

接口的简略表示方式是
“托球”与“托座”

。当需要展现接口的细节时，就需要表示出它与构件之间的实现关系和使用关系。



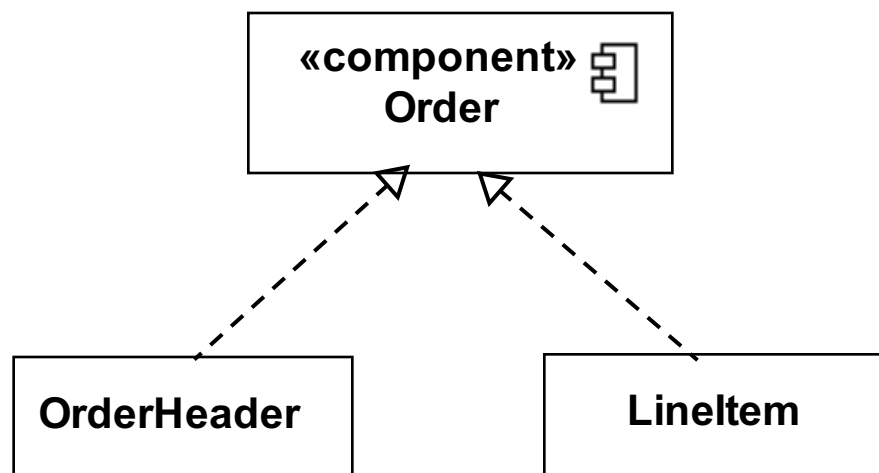
等价于



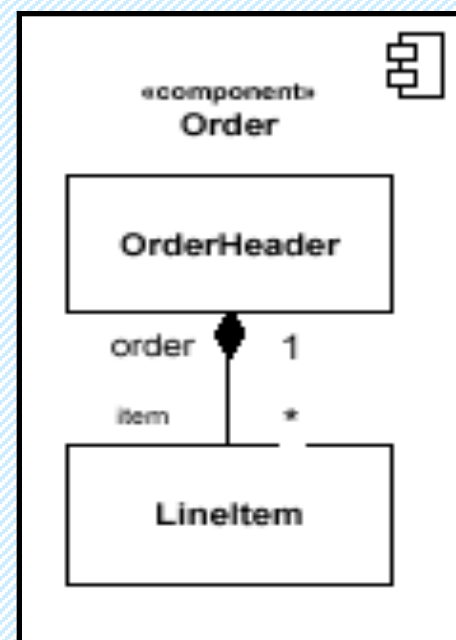
构件实现（ComponentRealization）

构件中的类与这个构件之间的关系
表明这些类实现了这个构件

UML2的表示法

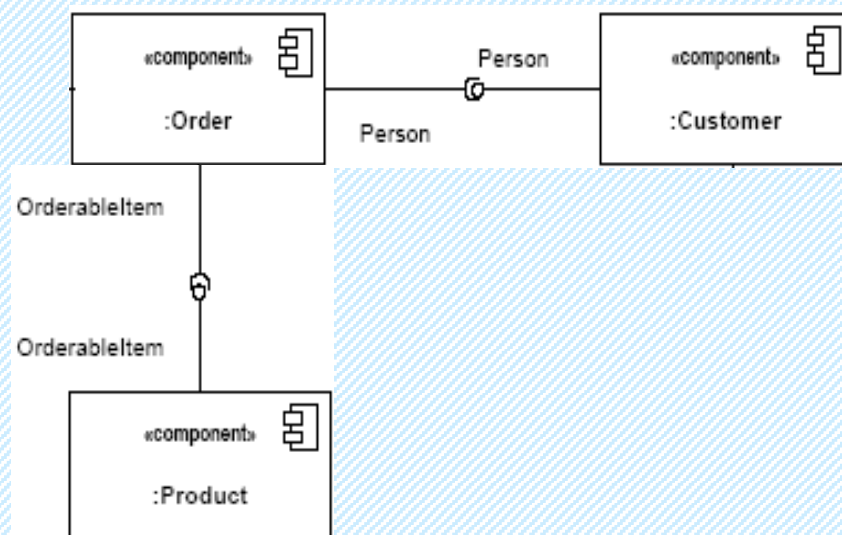
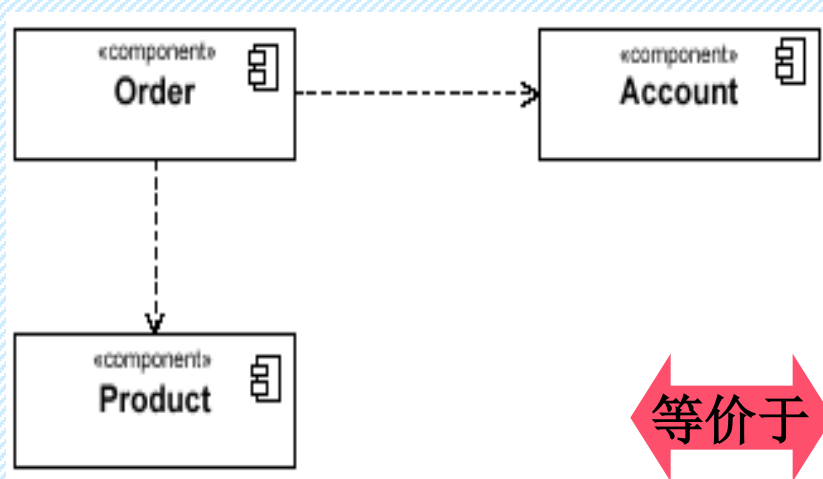


等价于



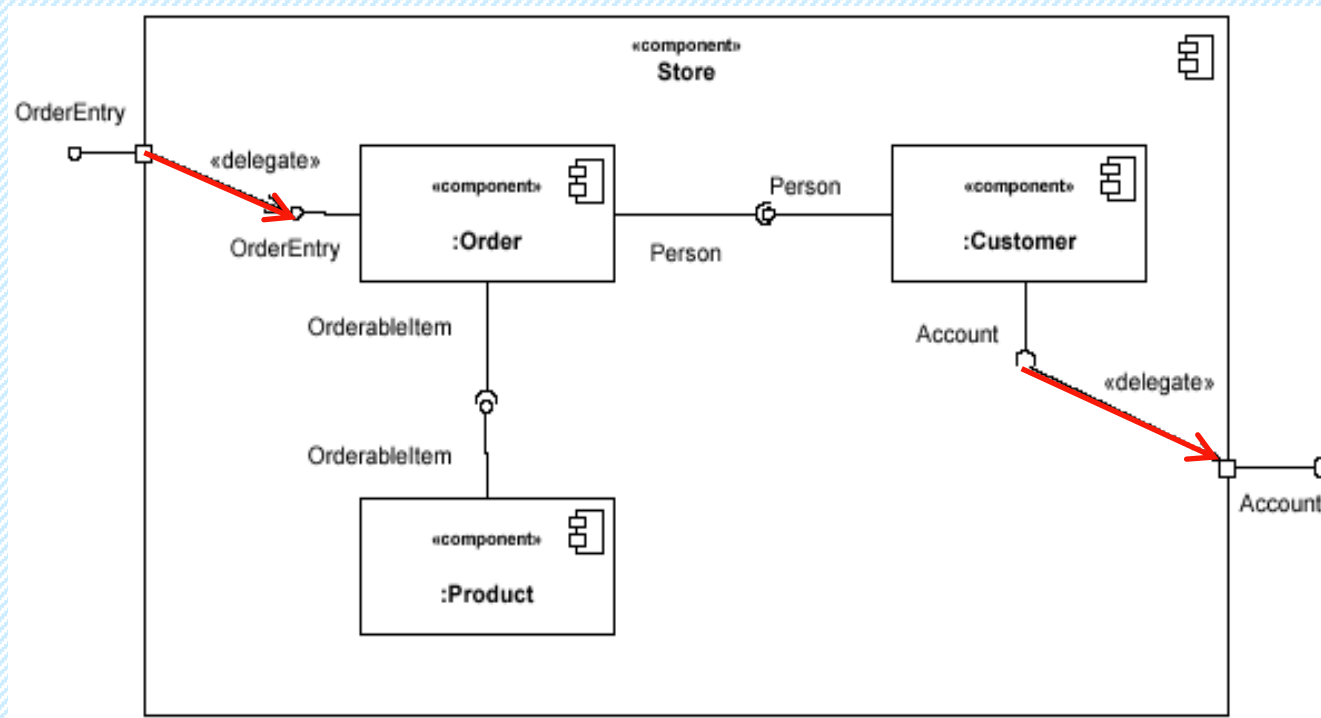
依赖（dependence）

除了«use»、«realization»等依赖关系之外，UML在构件图图中也使用了一般的依赖关系。用于构件之间，表示一个构件依赖另一个构件。

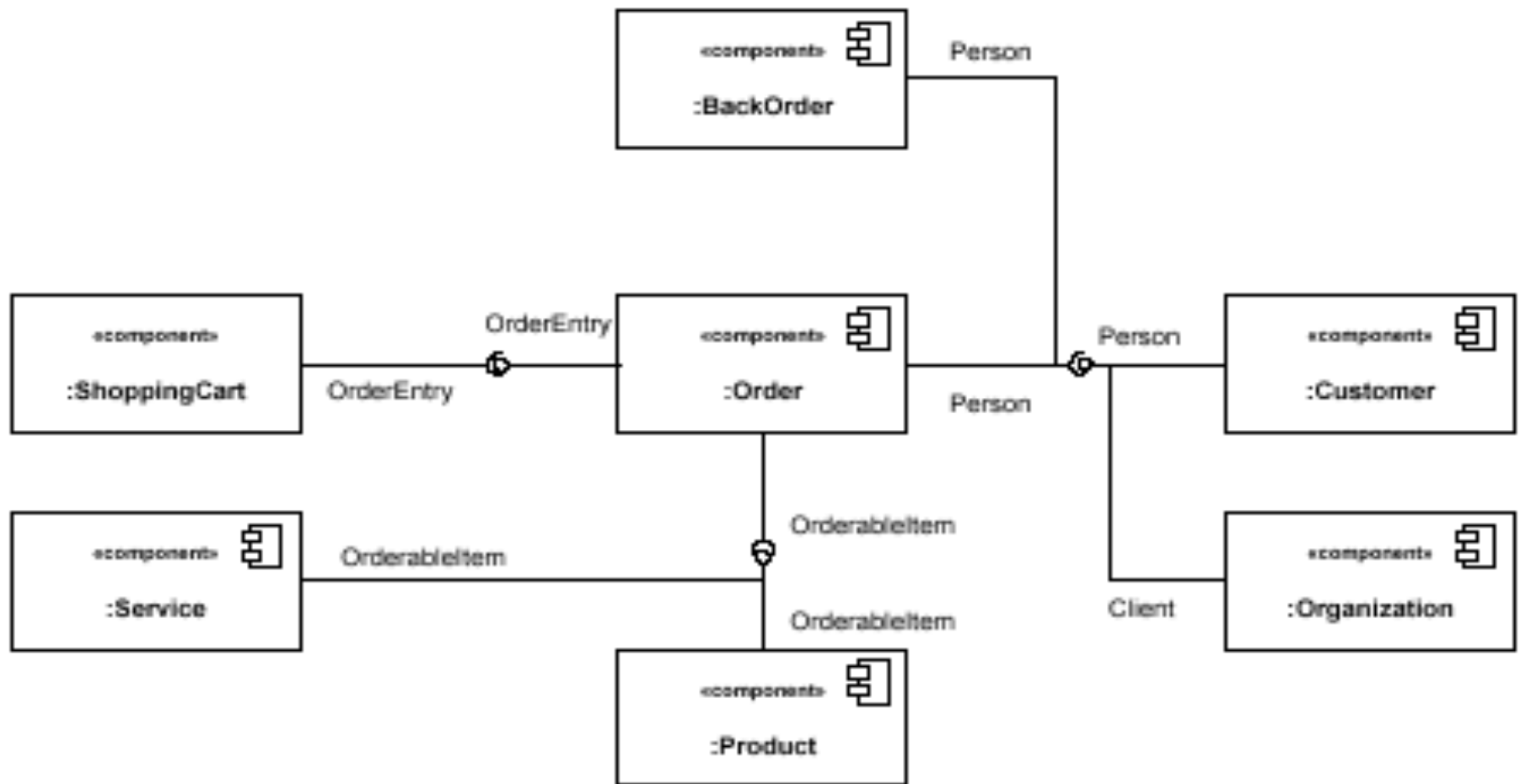


连接件（connector）

（1）委派连接件（**delegation connector**）是从构件的端口连接到构件内部成分的连接件，它“把构件外部的一个合约（由它的端口说明）链接到由构件中的部件对这个行为的内部实现。”



(2) 组装连接件 (assembly connector) 是两个构件之间的连接件，它表明一个构件提供了另一个构件所需要的服务。一个供接口和一个需接口之间的“托球与托座”衔接就表示一个组装连接件。



如何绘制构件图

用构件的图形符号表示每个构件

UML提供了多种构件表示方式，可以有选择地采用。

定义构件的接口

考察构件中各个类的操作，发现构件的供接口和需接口

定义连接件

考察各个构件的供接口和需接口，决定每个需接口应该与哪个供接口衔接。从而定义组装连接件。

如需表明构件接口与构件内部成分之间的衔接情况，可以使用委派连接件。

简要介绍

对象图 (**object diagram**)

组合结构图 (**composite structure diagram**)

通信图 (**communication diagram**)

交互概览图 (**interaction overview diagram**)

定时图 (**timing diagram**)

部署图 (**deployment diagram**)

对象图 (object diagram)

UML1.3：专门的章节和篇幅

UML1.4：“对象图是一种实例图，包括对象和数据的值。静态的对象图是类图的一个实例；它显示了在一个时间点上系统细节状态的一个快照。对象图的用处是很有限的，主要是展示数据结构的样子”。

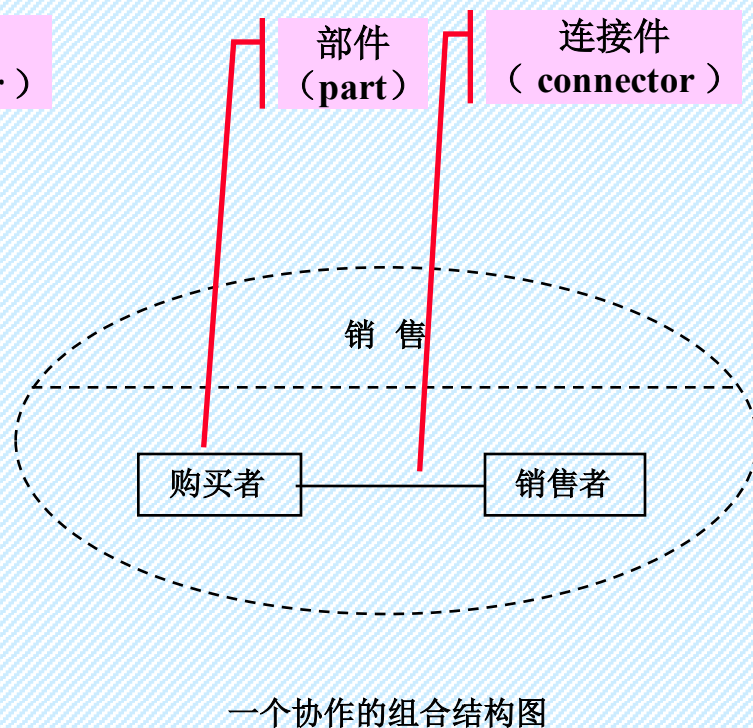
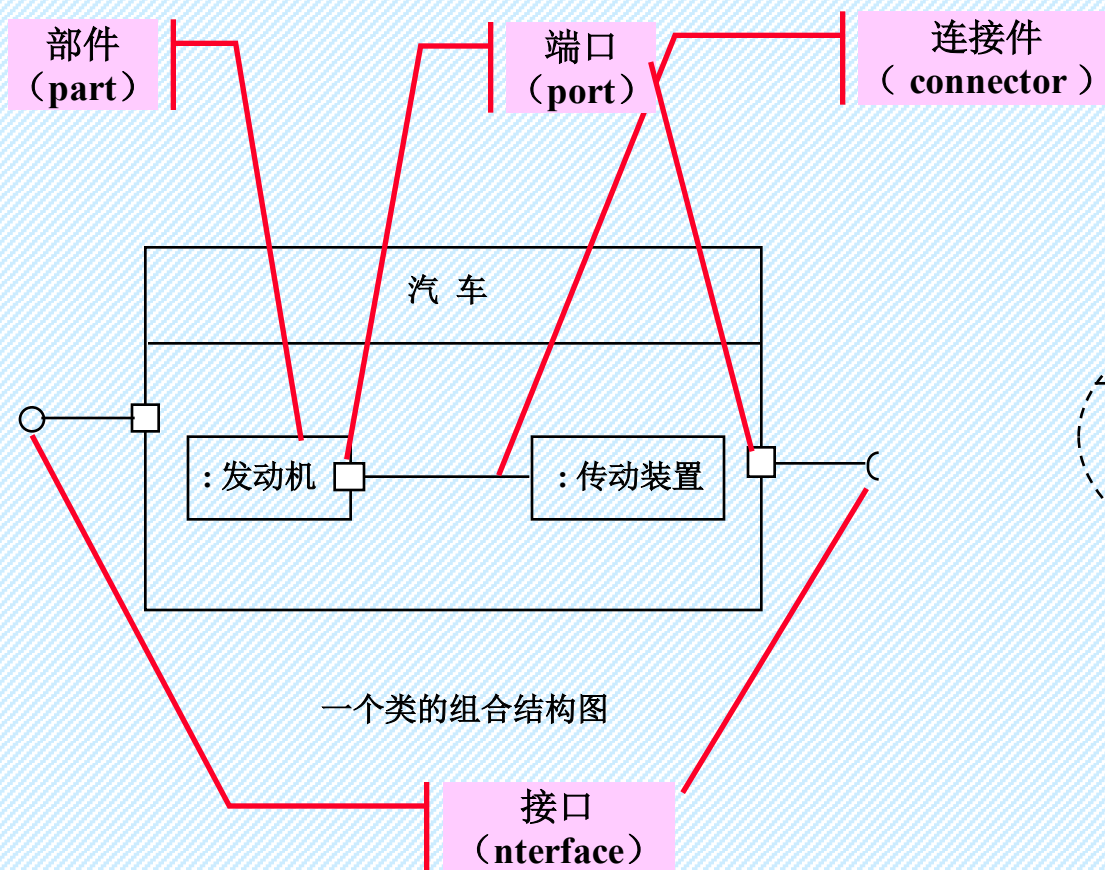
“工具不必支持单独形式的对象图，类图能包括对象，一个有对象而没有类的类图便是一个‘对象图’。不过这个术语对于刻画在各种方式下可能达到的特殊用法还是有用的”。

“一种含有在一个时间点上对象及其关系的图。一个对象图可以看成是一个类图或者协作图的特殊形式。”

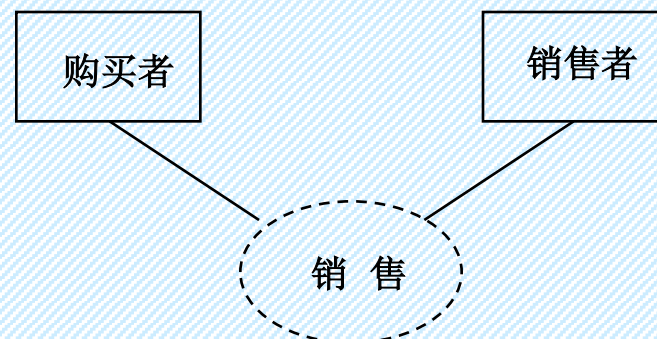
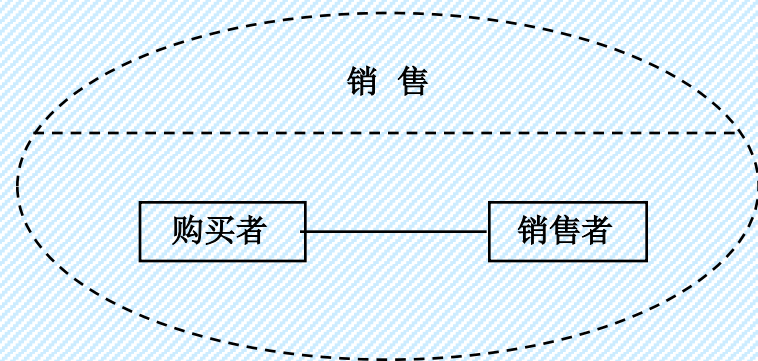
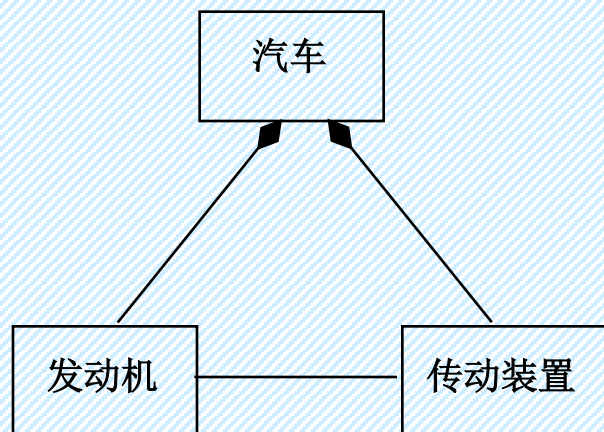
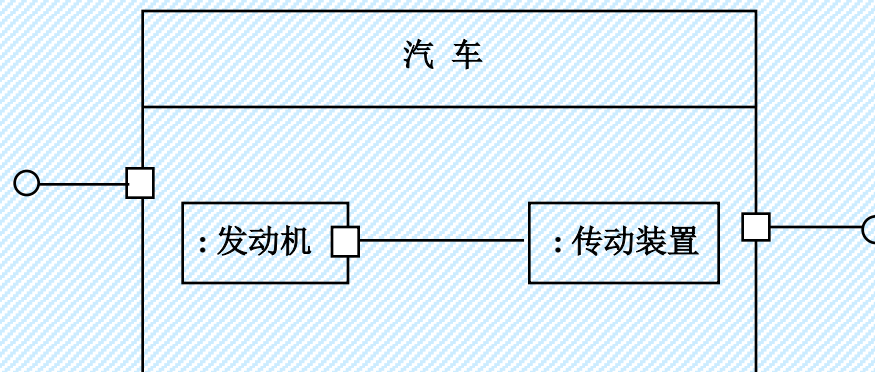
UML2：名存实亡。没有任何一句话介绍什么是对象图，也没有任何一句话说明对象图有什么用途。

组合结构图 (composite structure diagram)

组合结构图是UML2新增的图，用来表示类、构件、协作等模型元素的内部结构。



用其他表示方式代替组合结构图



通信图 (communication diagram)

通信图是UML2的4种交互图之一，在UML1被称为协作图（collaboration diagram），UML2改称通信图。

UML2对通信图的全部论述：

“通信图集中于生命线之间的交互，中心问题是其内部组织的体系结构以及如何与消息传输协调。消息的次序通过其序列号给出。”

“通信图与简单的顺序图是一致的，简单的顺序图是指，没有交互使用和组合片段等结构机制，并且假设不会发生消息超越（即在一个给定的消息集合内接收消息的次序与发送消息的次序不同）或者与之无关。”

通信图中的有关概念

问题:

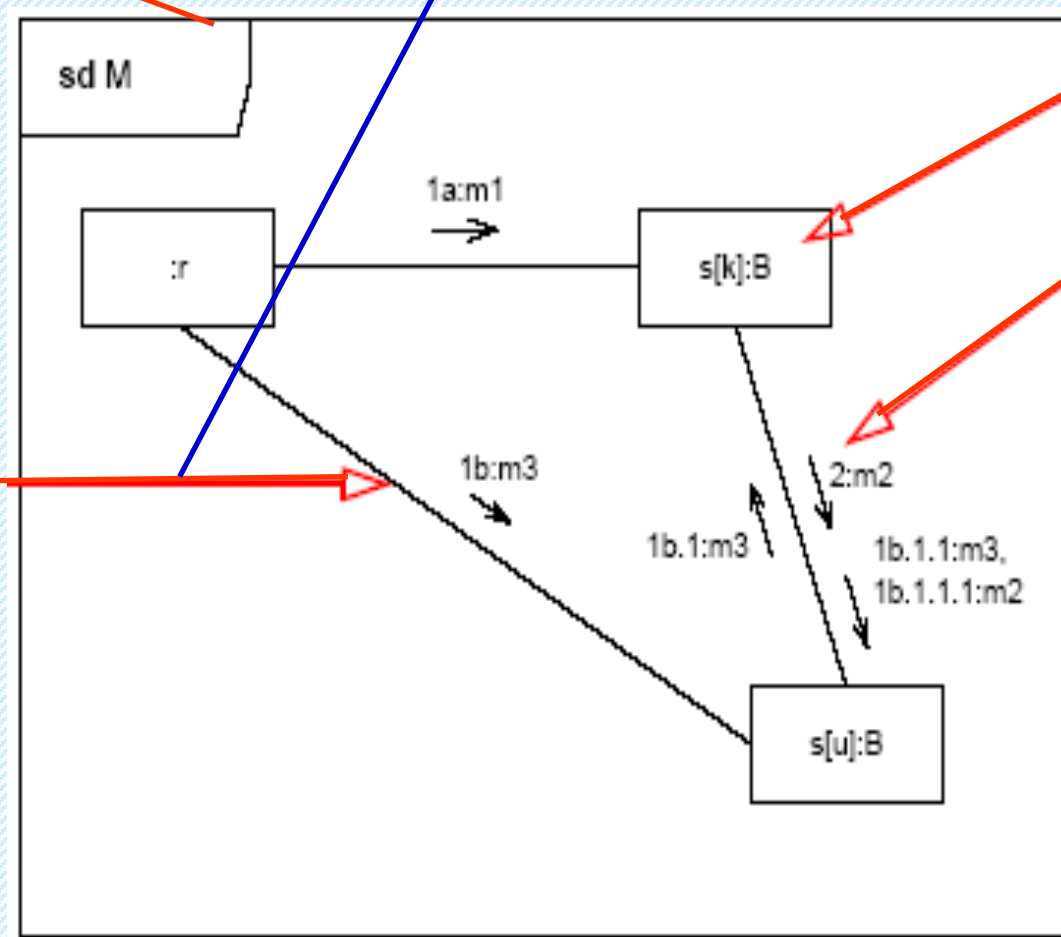
消息是否只能存在于具有关联的类之间? (见1.3.7节的讨论)

帧

消息

生命线
(对象)

带序号的
消息



交互概览图 (interaction overview diagram)

交互概览图是UML2新增加的图，是4种交互图之一。

交互概览图采用活动图的构造方式把多个交互组织到一个图中。它利用了活动图的各种控制结点，并把活动图的动作结点替换为一个交互或者交互使用。每个交互或者交互使用可以用一个顺序图来描述。通过这种方式，可以把许多零散的交互组织到一个图中。

“交互概览图通过活动图的一个变种，以提升控制流概览的方式来定义交互。交互概览图的焦点集中于控制流的概况，其中的结点是交互或者交互使用。生命线和消息不在概览的层次上出现。”——UML2

例:

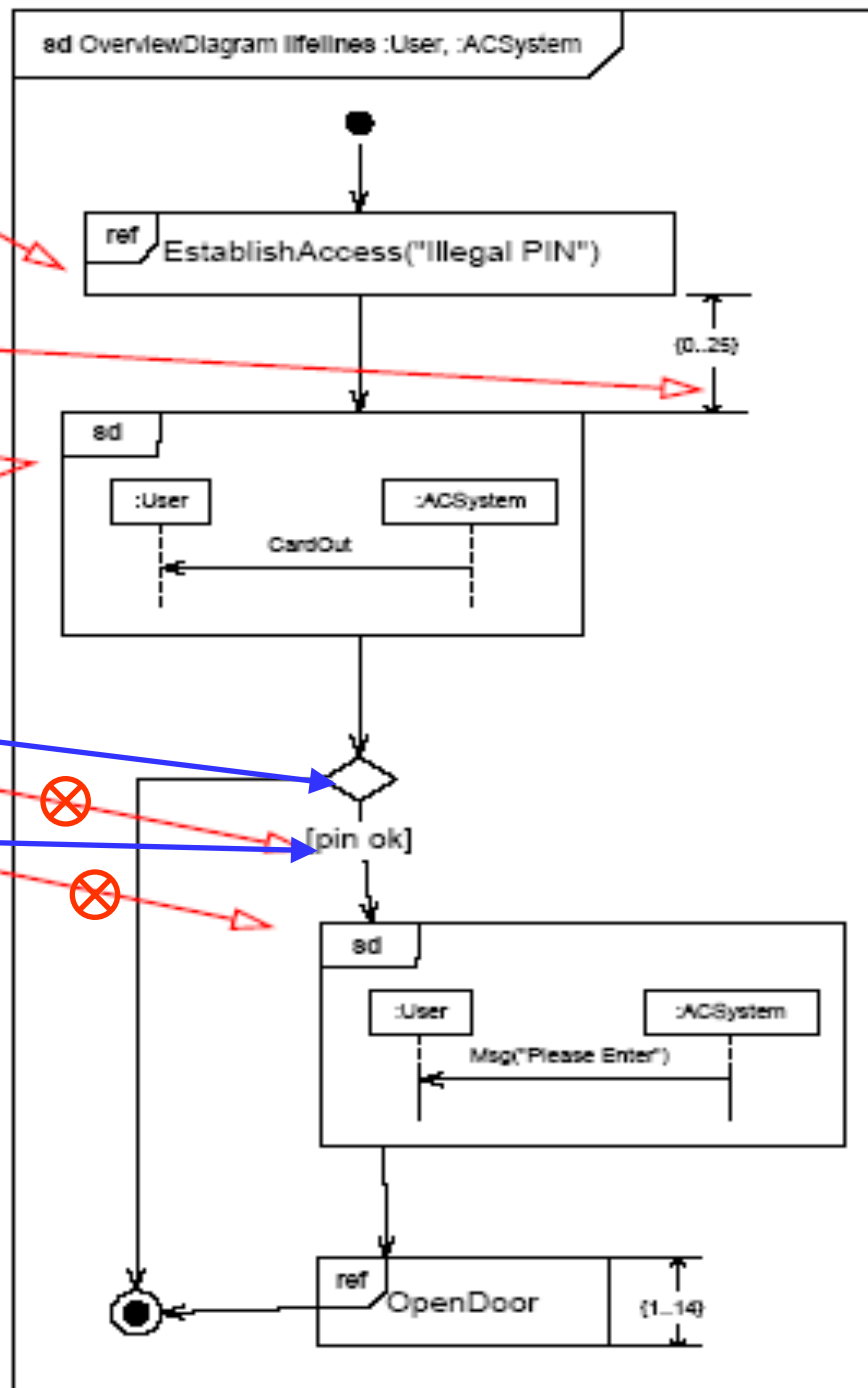
交互使用

时间约束

在线交互

判断

交互约束



定时图 (timing diagram)

定时图是UML2新增的图，是4种交互图之一。

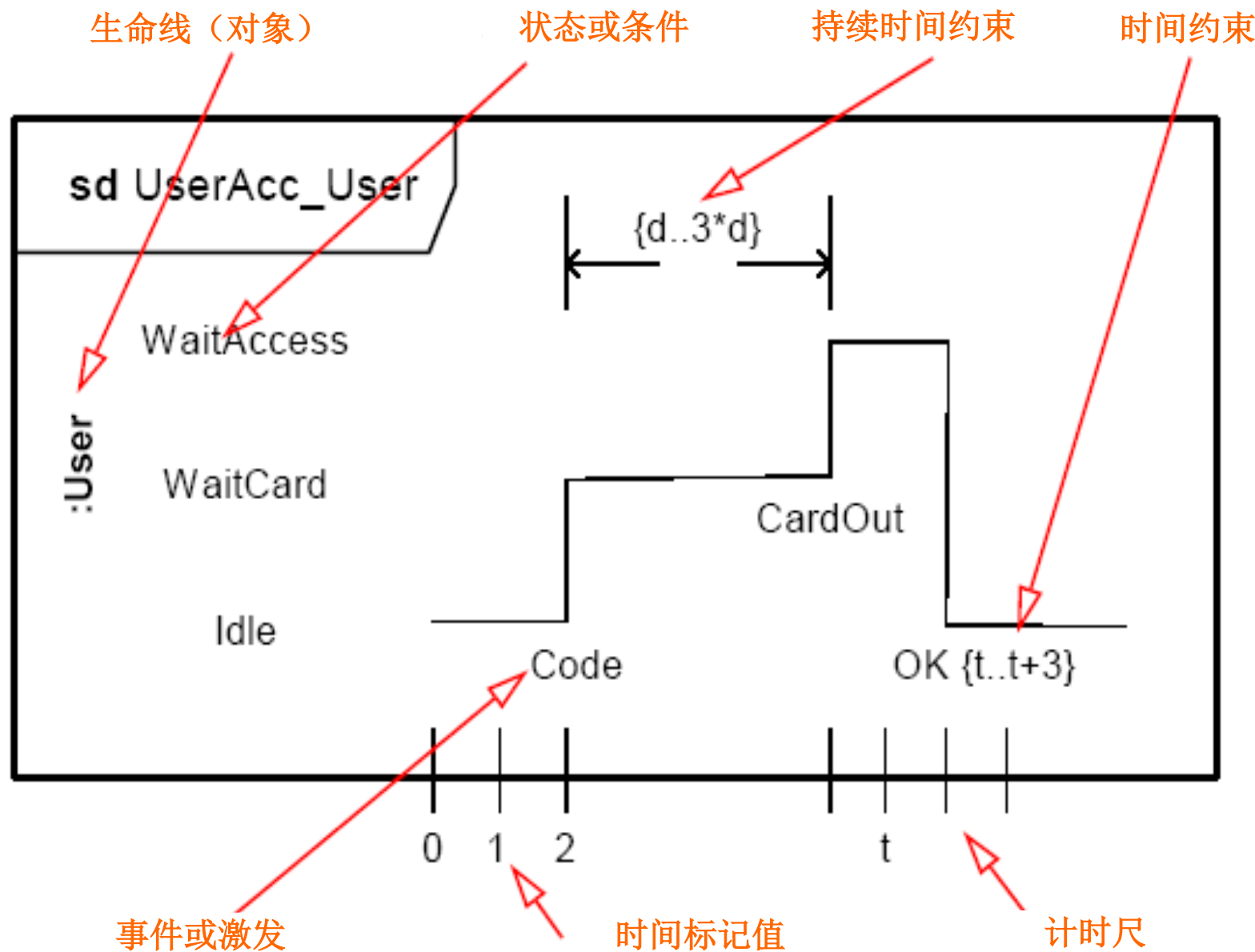
“定时图用于表示交互，此时的主要目的是考虑时间问题。定时图的焦点集中于生命线内部以及它们之间沿着时间轴的条件变化。”

“定时图既描述单个类目的行为，也描述类目间的交互，其焦点是引起模型中生命线条件变化的事件所发生的时刻”

一个对象在交互中可以处于不同的状态，或者遇到不同的条件。定时图可以把状态（条件）发生变化的时刻以及各个状态所持续的时间具体地表示出来。

如果把多个对象放在一个定时图中，还可以把它们之间发送和接收消息的时刻表现出来。

一个对象的定时图

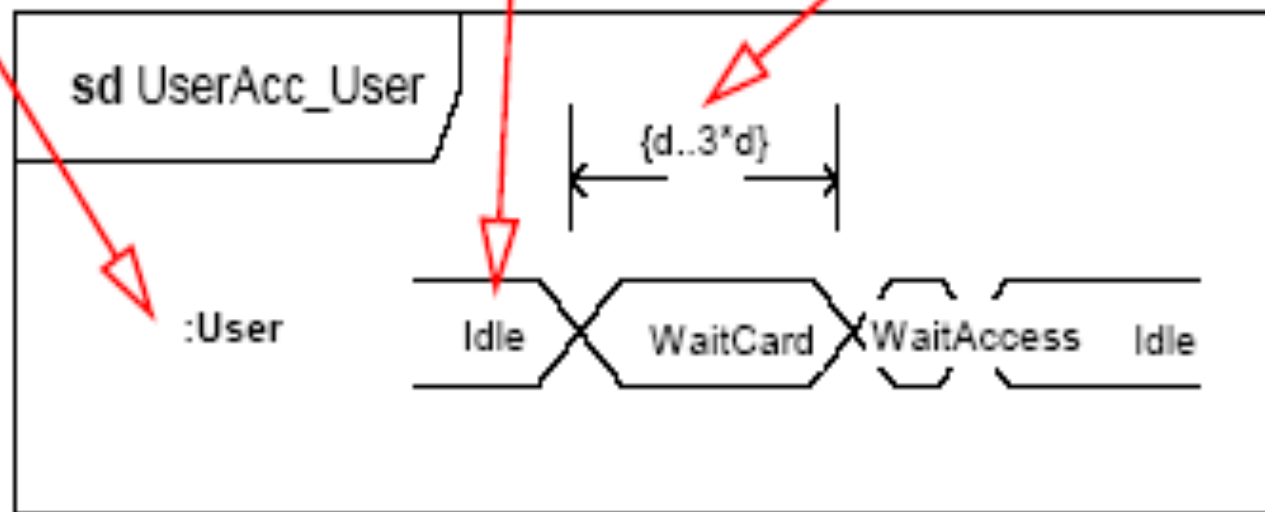


定时图的简略形式

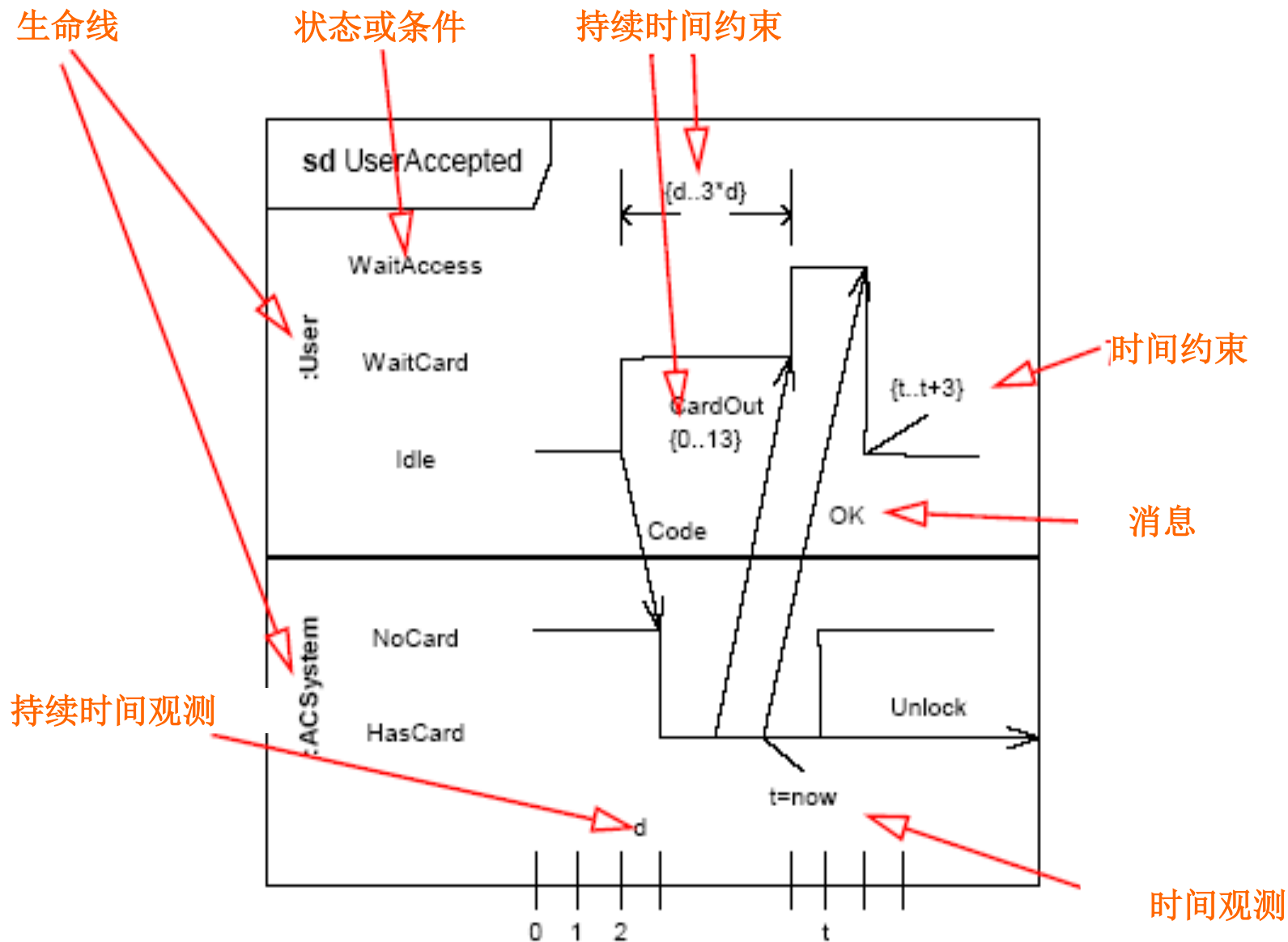
生命线

状态或条件

持续时间约束



多个对象的定时图



部署图 (deployment diagram)

UML1的定义：“一种显示运行时的处理结点以及在其上生存的构件、进程及对象配置的图。”

UML2的定义：“部署图展示了制品在结点上根据它们之间部署的定义所做的定位。”

变化：部署在结点上的软件成分不再是构件，而是**制品**。

主要概念

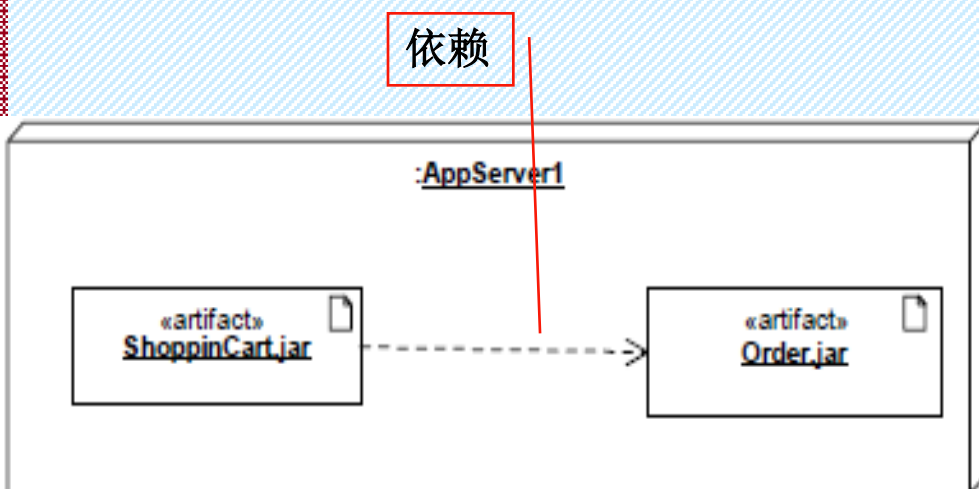
结点 (node) 表示一个硬件设备或执行环境，即被开发的软件制品将要部署于其上的宿主设备或环境。

制品 (artifact) “制品是对软件开发过程中或系统的部署与操作中所使用或产生的信息的物理块的说明。制品的例子包括模型文件、源文件、草稿、二进制可执行文件、数据库表、开发交付项、或者字处理的文档、邮件消息等。”

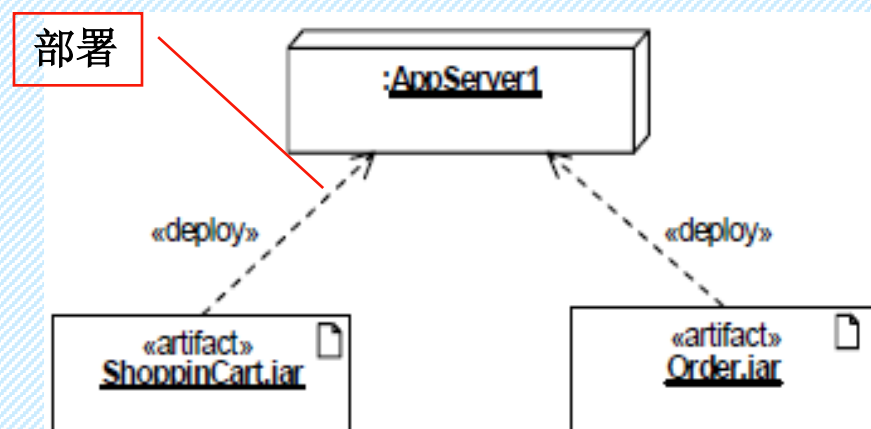
执行环境 (ExecutionEnvironment) “执行环境是一个结点，它为以可执行制品的形式部署于其上的各类构件提供了一个执行环境。”

各种关系

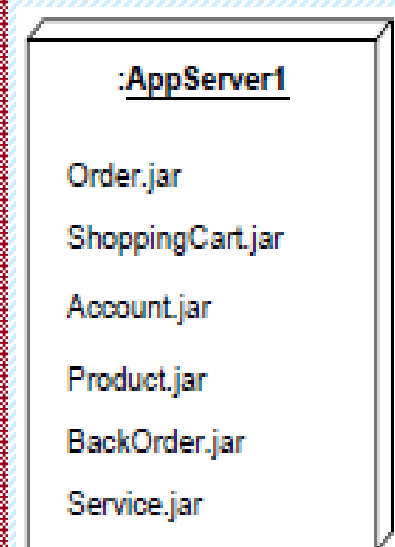
- 部署 (deploy) 依赖
- 制品之间的依赖
- 结点之间的关联



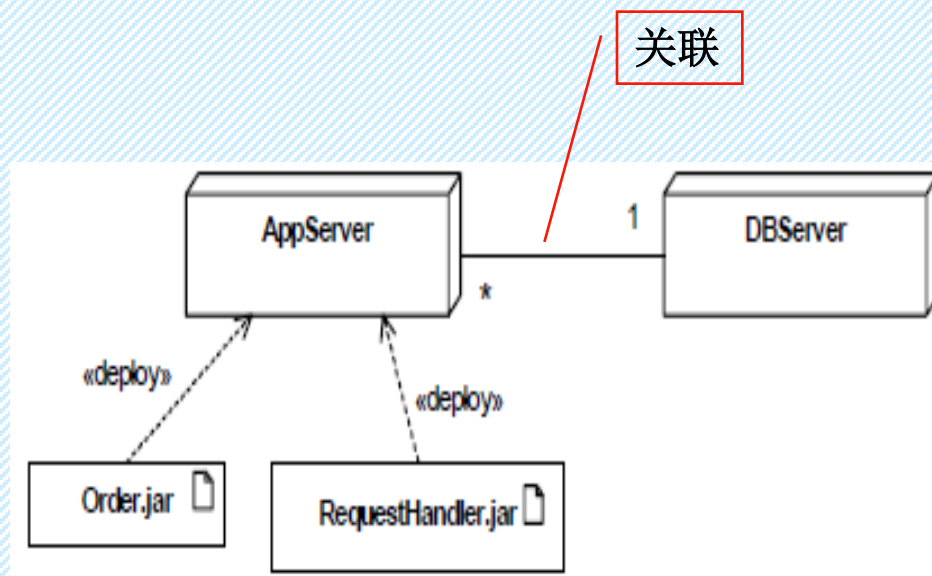
(a) 结点和制品



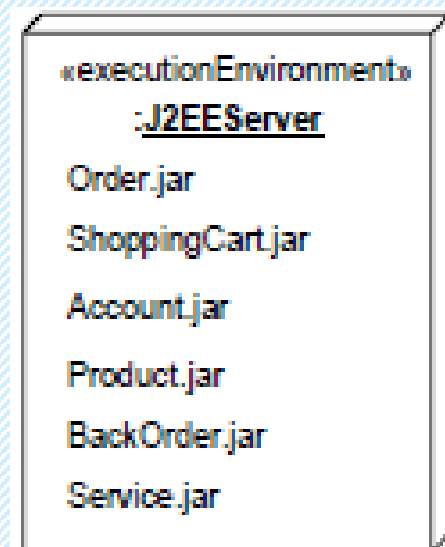
(b) 显式地表示部署



(c) 以列表方式表示部署



(d) 结点之间的关系

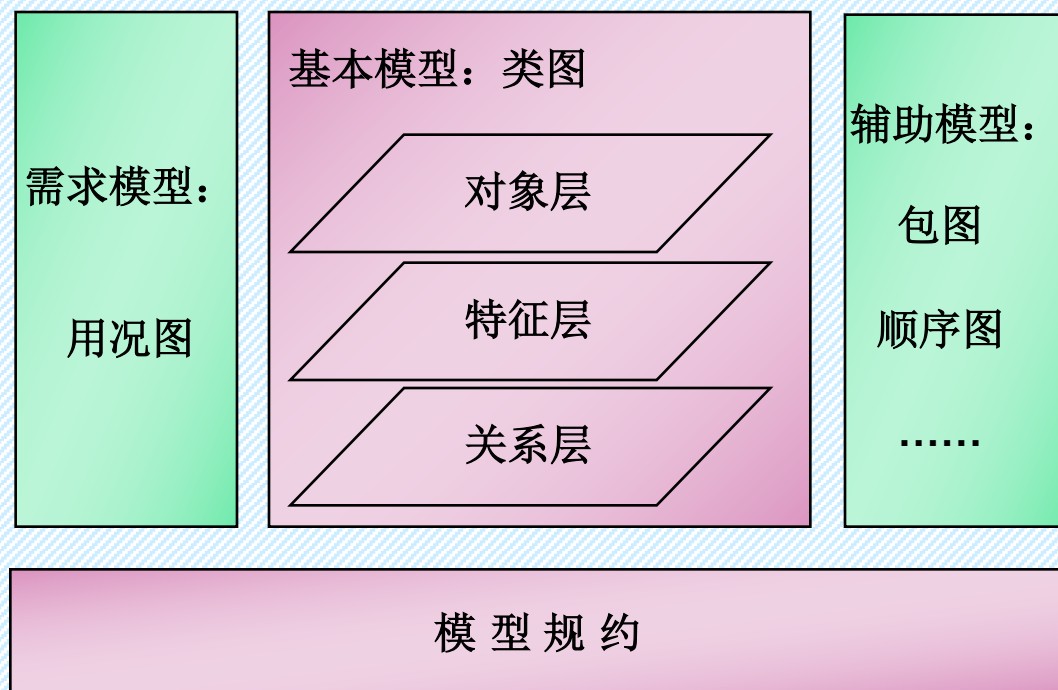


(e) 执行环境

模型规约（model specification）

——对系统模型的详细解释与说明。

仅靠图形文档还不足以详细、精确地表达建模阶段应该给出的全部系统信息。在软件工程中，各种分析与设计方法通常需要在以图形方式表达系统模型的同时，又以文字的方式对模型进行解释和说明。



问题讨论

规约是给谁看的

系统主要是由人开发，还是由计算机自动生成？

采用形式语言还是自然语言

“利用形式的表示法可以避免有歧义自然语言所引起的错误解释的风险。可是，它却引出由于作者和读者不真正了解OCL所引起的错误解释的风险。因此，除非你有一些对谓词演算感到舒服的读者，我愿意推荐使用自然语言。” ——M. Fowler

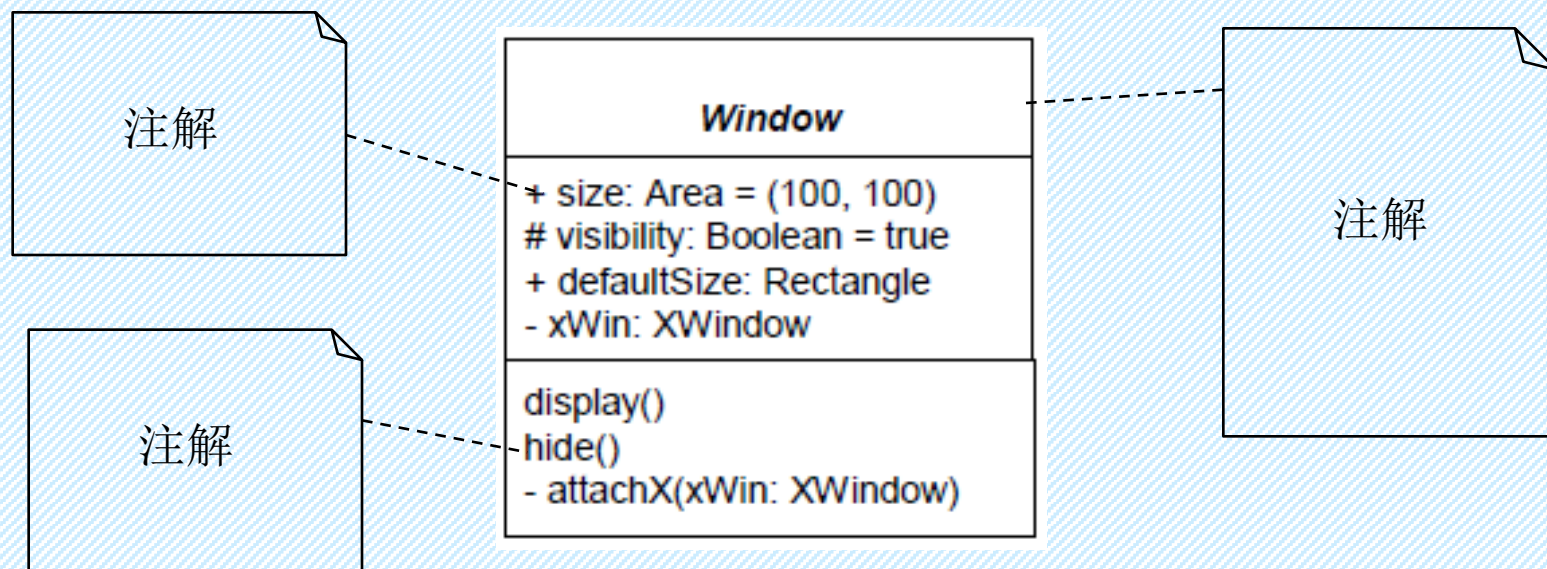
规约的组织

分离方式——把模型图和对模型的详细描述分别组织到不同的文档中。

混合方式——凡是与模型有关的信息，不分主次、不分巨细、不分级别，统统放到图中进行可视化表示。

UML的风格——混合方式

- 模型元素的基本表示法包含许多细节的描述
- 通过注解（**comment**）给出更多的细节



目前大部分建模工具支持分离方式，通过模型规约来描述模型元素的细节

例1: IBM Rational Rose对一个类的说明

The screenshot displays the IBM Rational Rose IDE interface. The main workspace shows a UML class diagram with two classes: **AbstractBusinessDelegateFactory** (an interface) and **BusinessDelegateFactory** (a concrete class). The interface has a method `createPiggyBankBusinessDelegate()`. The concrete class has a method `init()` and a reference to the interface. A dashed line indicates a generalization relationship from the concrete class to the interface.

Annotations in the diagram include:

- Pay attention to the abstraction level of your diagram. You can use class diagrams to model both the domain (high level) and the design (low level) of your system.**
- Model only the relevant details of your system and save the implementation details for developers.**
- Name your classes according to the domain of your model.**
- implementations vertically to indicate a hierarchy**
- When you describe an association, always put the owner on the left and the component on the right**

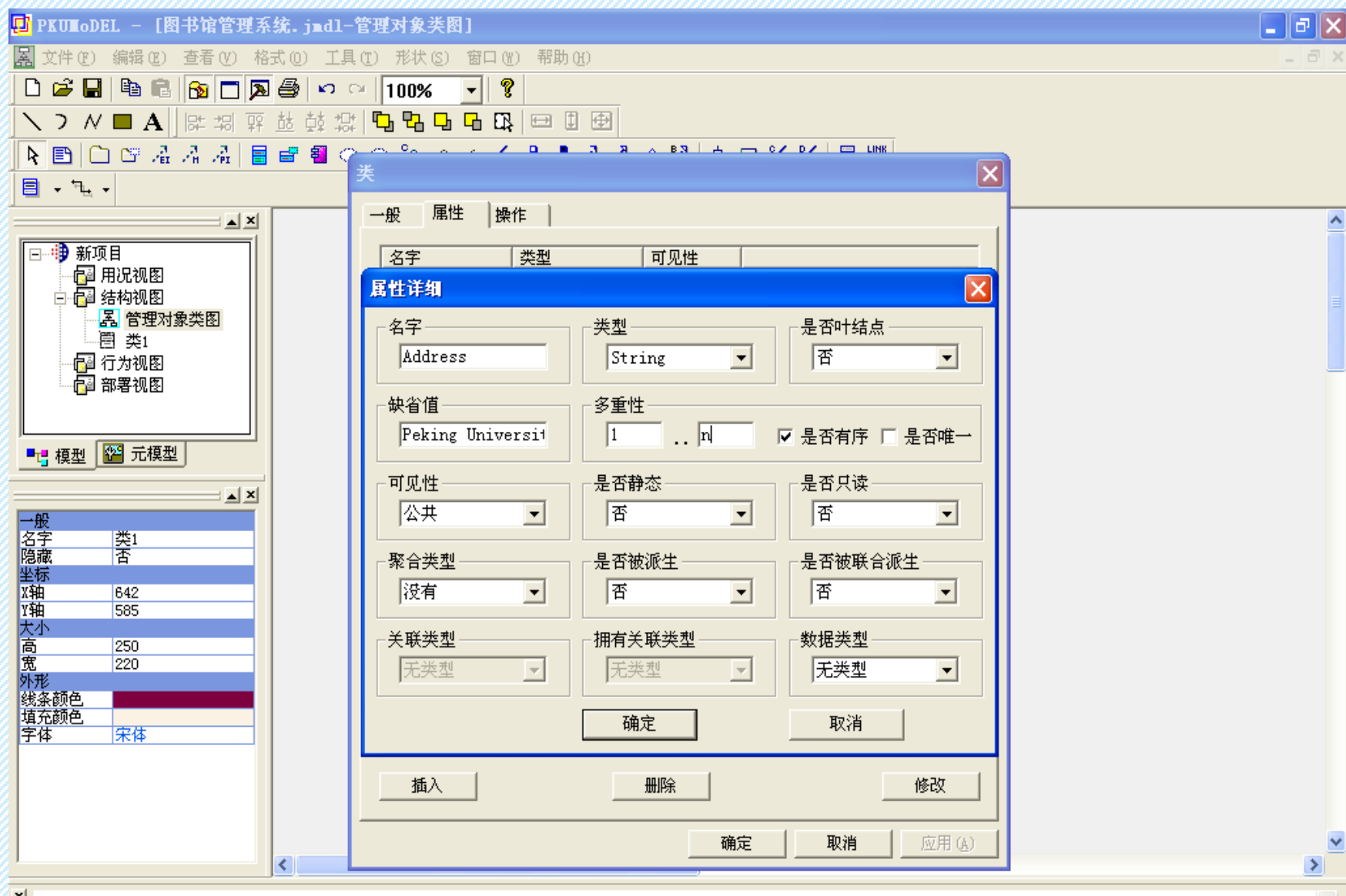
The left sidebar shows the **模型浏览器** (Model Browser) with a tree structure:

- SampleClassProject
 - Sample_class.emx
 - Design Model
 - Annotations
 - «透视图» Architectural Layer
 - Design-Level Use-Case Realization
 - itso.ad.business
 - itso.ad.citybank
 - itso.ad.common
 - «ModelLibrary» Model Building
 - delegate.ejb (UML2)

The bottom right pane shows the **属性** (Properties) view for the class **Design Model::itso.ad.business::factory::BusinessDelegateFactory**.

属性	值
构造型	
关键字	
集合	
可打包的元素的可视性	public
可视性	public
名称	BusinessDelegateFactory
名称表达式	
模板参数	
是抽象的	true
是活动的	false
是叶	false

例2：北京大学PKUModel4.0对一个属性的说明



建议的目标与措施

目标：

完整性——提供完整准确的模型信息

易读性——容易被人的阅读和理解

支持自动化处理——使尽可能多信息可自动转换为源程序

支持软件复用——规约中的信息容易在其他系统中复用

措施：

采用分离方式

采用格式化的描述方式

自然语言与形式化语言结合使用

在类的规约中定义类之间的关系

对于有向关系只在其源端的类中进行描述

对一个面向对象的系统模型建立规约，关键问题是为类图中的每个类给出详细、准确的定义。主要成分是**类规约**（**class specification**），即对每个类的详细说明。

类规约包括如下内容：

1. 类的总体说明
2. 属性说明
3. 操作说明
4. 对象实例说明

类规约组织格式

类的总体说明：

类名：<名字>

解释：[<文字描述>]

一般类：[<类名>] {,<类名>}

主动性：Yes | No

持久性：Yes | No

辅助模型：{<访问路径和名字>}

其他：.....

属性说明：{

名称与数据类型：<属性名>：<类型>

属性解释：[<文字描述>]

多态性：[*|×]

关联、聚合或组合：[关联|聚合|组合][<文字描述>]

其他：..... }

操作说明：{

特征标记：<操作名>（[<参数>：<类型>]
{,<参数>：<类型>}）[:<返回类型>]

操作解释：[<文字描述>]

主动性：主动[进程|线程]|被动

多态性：[*|×]

消息发送：[<类名>·<操作名>]{,<类名>·<操作名>}

操作流程：[<访问路径和名字>]

其他：.....

}

对象实例说明：{

处理机：<结点名>{,<结点名>}

内存对象：{<名称>[(n元数组)][<文字描述>]}

外存对象：{<名称>[<文字描述>]}

}

其他模型图的规约

原则上，对任何一种模型图都可以通过相应的模型规约给出更详细的信息，但是在实践中需要根据各种图的具体作用而区别对待，例如：

用况图——对每个用况的文字描述就是用况图的规约

包图——对每个包，必要时可以说明它的意义

活动图、状态机图、定时图——这些图通常是为了详细地描述各个类的行为或者状态。它们本身的作用就是对一个类做进一步的辅助说明，一般不需要再对它们做更多的说明。

模型规约的建立过程

与模型图同步进行，或者分别进行

OOA与OOD有不同的分工，但是没有严格的界限

原则上，描述问题域中固有的事物的对象及其特征和相互关系，应该在OOA阶段定义；与实现条件有关的对象及其特征和相互关系，应该在OOD阶段定义。

在这个原则下，有相当一部分工作既可以在OOA阶段进行，又可以在OOD阶段进行。