

面向对象分析模型总结

主要概念

面向对象的概念包括以下两种情况：

- (1) 用来构成系统模型的某种基本成分，称为**建模元素**
- (2) 在建模中需要遵守的某种**原则**，不代表任何模型成分

主要建模元素

对象、类（所有的对象都通过类来表示）

属性、操作（类属性和实例属性，被动操作和主动操作）

一般-特殊关系，一般-特殊结构

整体-部分关系，整体-部分结构

关联（二元关联、多元关联）

消息（控制流内部的消息，控制流之间的消息）

主要原则

(1) 抽象

什么叫抽象？

OO方法广泛地运用抽象原则，例如：

- 系统中的对象是对现实世界中事物的抽象，
- 类是对象的抽象，
- 一般类是对特殊类的进一步抽象，
- 属性是事物静态特征的抽象，
- 操作是事物动态特征的抽象。

过程抽象

任何一个完成确定功能的操作序列，其使用者都可把它看作一个单一的实体，尽管实际上它可能是由一系列更低级的操作完成的。

数据抽象

根据施加于数据之上的操作来定义数据类型，并限定数据的值只能由这些操作来修改和观察。

(2) 分类

分类就是把具有相同属性和操作的对象划分为一类，用类作为这些对象的抽象描述。

不同程度的抽象可得到不同层次的类，形成一般-特殊结构（又称分类结构）。

强调：在类的抽象层次上建模

(3) 封装

(4) 继承

(5) 聚合

(6) 关联

(7) 消息通信

即要求对象之间只能通过消息进行通讯，而不允许在对象之外直接地存取对象内部的属性。

(8) 粒度控制

人们在研究问题时既需要微观的思考，也需要宏观的思考。因此需要控制自己的视野：考虑全局时，注重其大的组成部分，暂时不详察每一部分的具体的细节；考虑某部分的细节时则暂时撇开其余的部分。这就是粒度控制原则。

引入包（package）的概念，把模型中的类按一定的规则进行组合，形成一些包，使模型具有大小不同的粒度层次，从而有利于人们对复杂性的控制。

(9) 行为分析

- 以对象为单位描述系统中的各种行为
任何行为都归属于某个对象，用对象的操作表示。
对象的操作只作用于对象自身的属性。
- 通过消息描述对象之间的行为依赖关系
如果一个对象操作的执行需要另一个对象为它提供服务，则在模型中表现为前者向后者发送消息。
- 认识行为的起因，区分主动行为和被动行为
用主动对象的主动操作描述主动行为
用对象的被动操作描述被动行为
- 认识系统的并发行为
在分析阶段根据，根据系统的需求和事物的主动性来认识系统的并发行为。在设计阶段，根据具体的实现条件确定系统中需要设计哪些控制流。

模型及其规约

在分析阶段和设计阶段建立的系统模型分别称为**OOA模型**和**OOD模型**

正规理解：一个系统模型，应包括建模过程中产生的图形、文字等各种形式的文档。因为，所谓“模型”是指某一级别上的系统抽象描述，构成这种描述的任何资料都是模型的一部分。

习惯说法：目前大部分OOA/OOD著作谈到“模型”，一般是指OOA或OOD过程中产生的图形文档。

一般习惯——将**模型**和**模型规约**分别讨论

OOA和OOD模型包括**需求模型**、**基本模型**和**辅助模型**，通过**模型规约**做详细说明

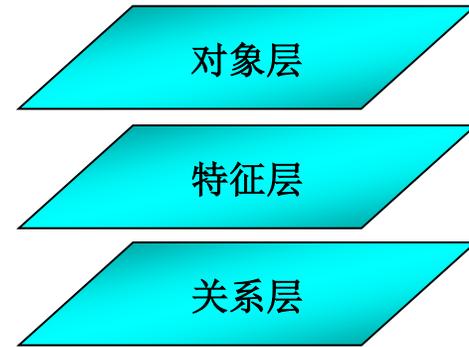
基本模型——类图

面向对象的建模中最重要、最基本的模型图

集中而完整地体现了面向对象的概念

为面向对象的编程提供了直接、可靠的依据

可以从三个层次来看



需求模型——用况图

每个用况是一项系统功能使用情况的说明，把每一类参与者对每一项系统功能的使用情况确切地描述出来，便全面地定义了系统的功能需求

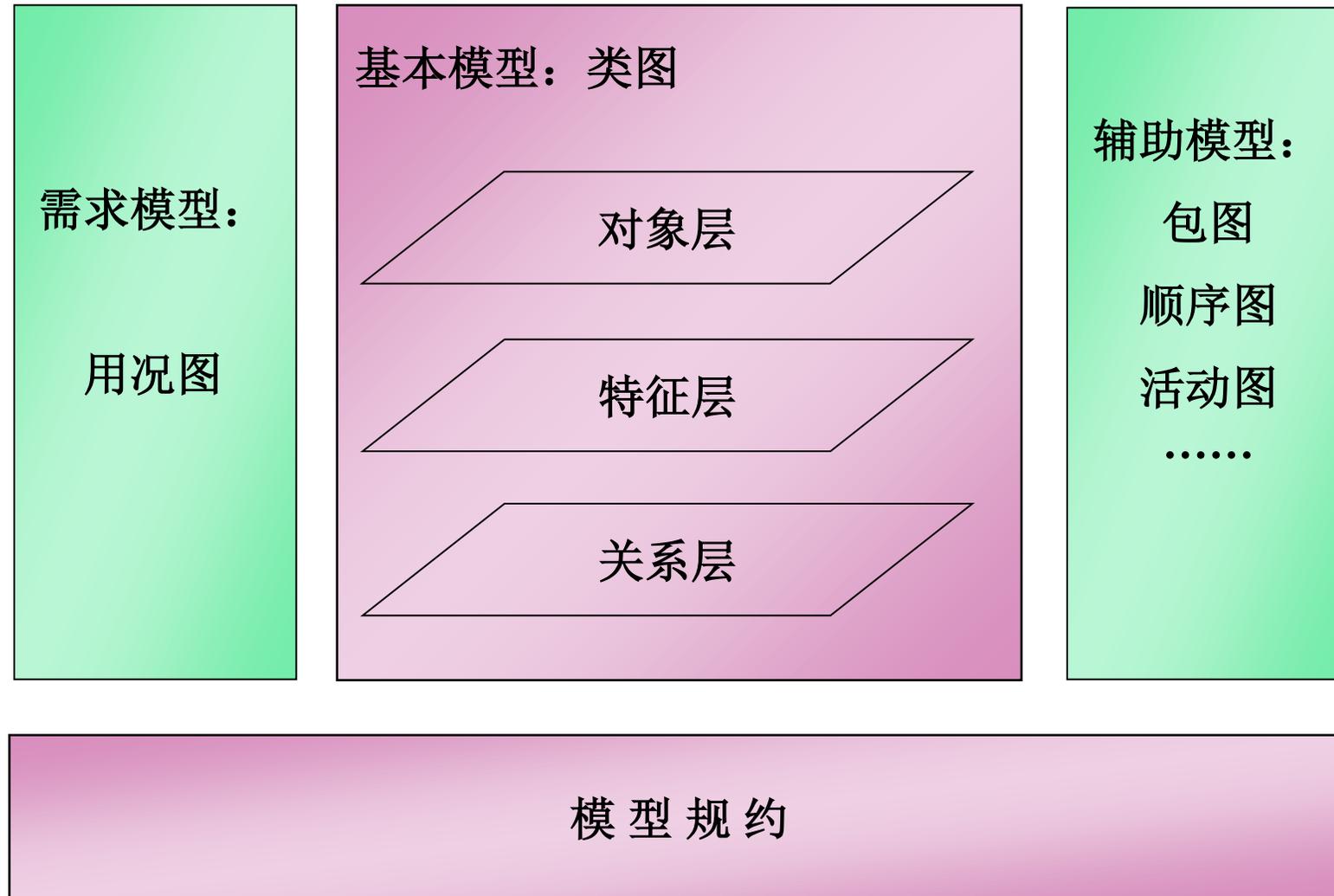
辅助模型——其他各种图

对类图起到辅助作用，提供更详细的建模信息，或者从不同的视角来描述系统。例如包图、顺序图、活动图等

模型规约

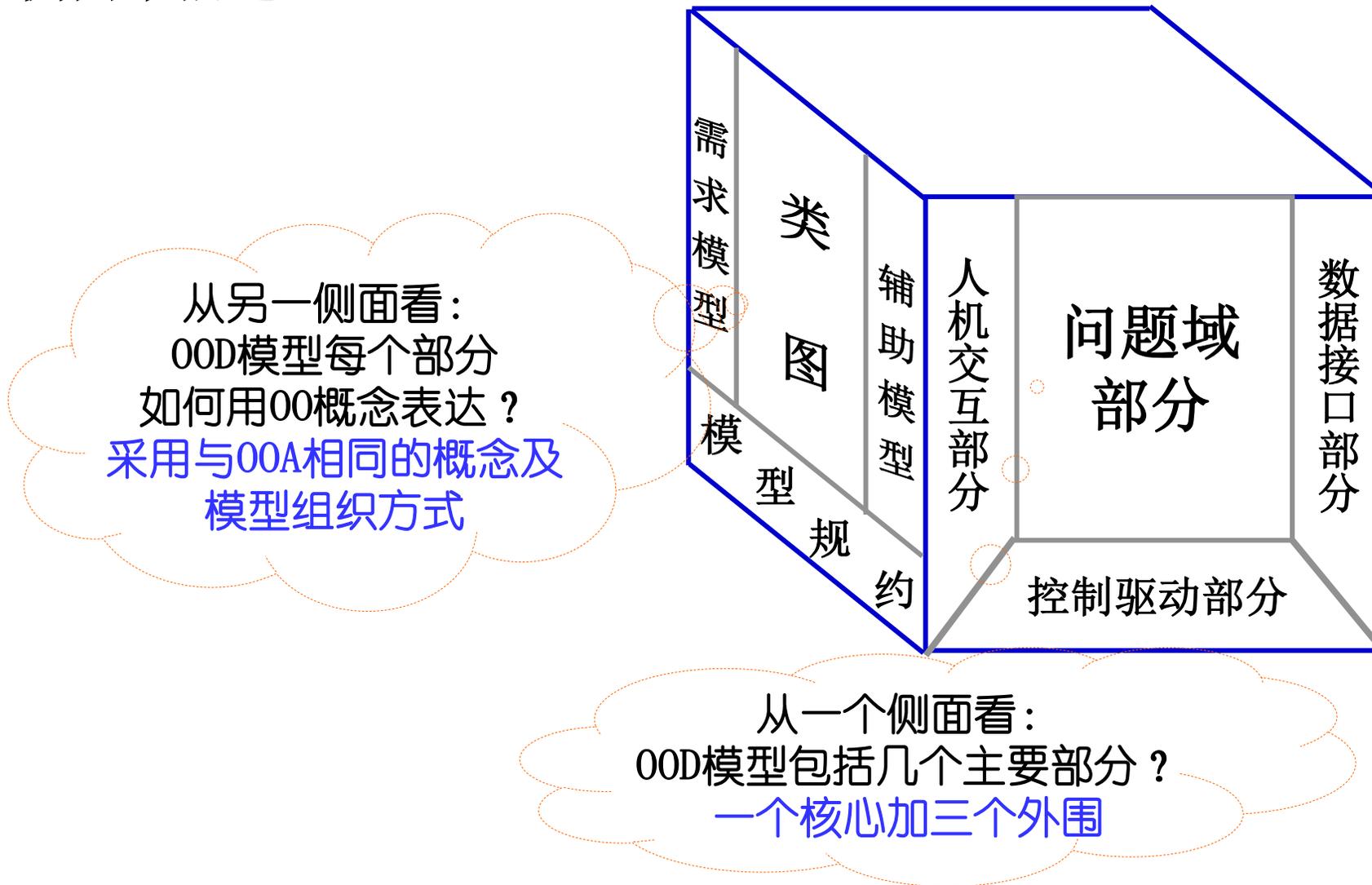
对上述各种模型图及其模型元素的详细而确切的定义和解释。

OOA模型框架

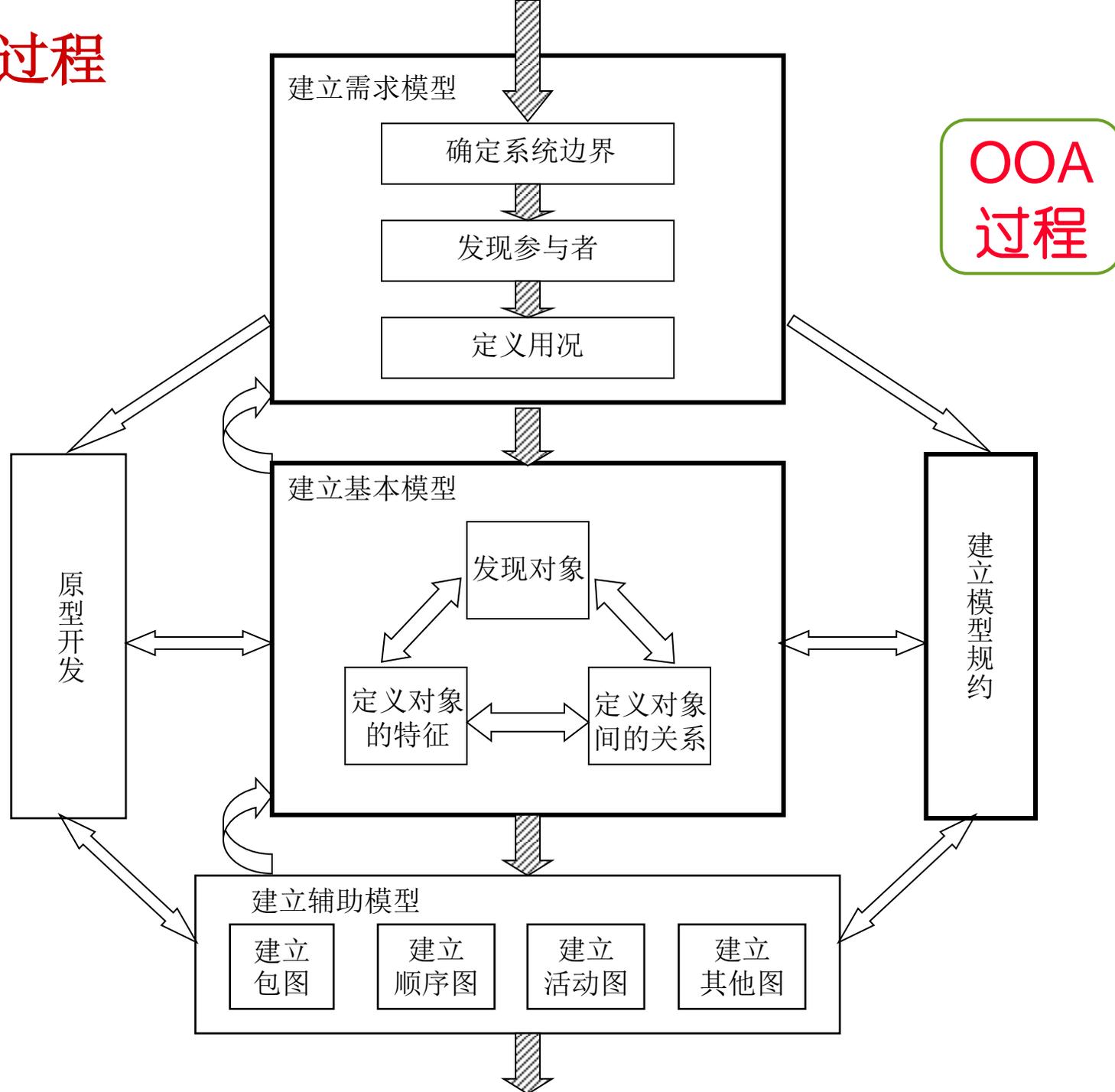


OOD模型框架

——从两个侧面来描述

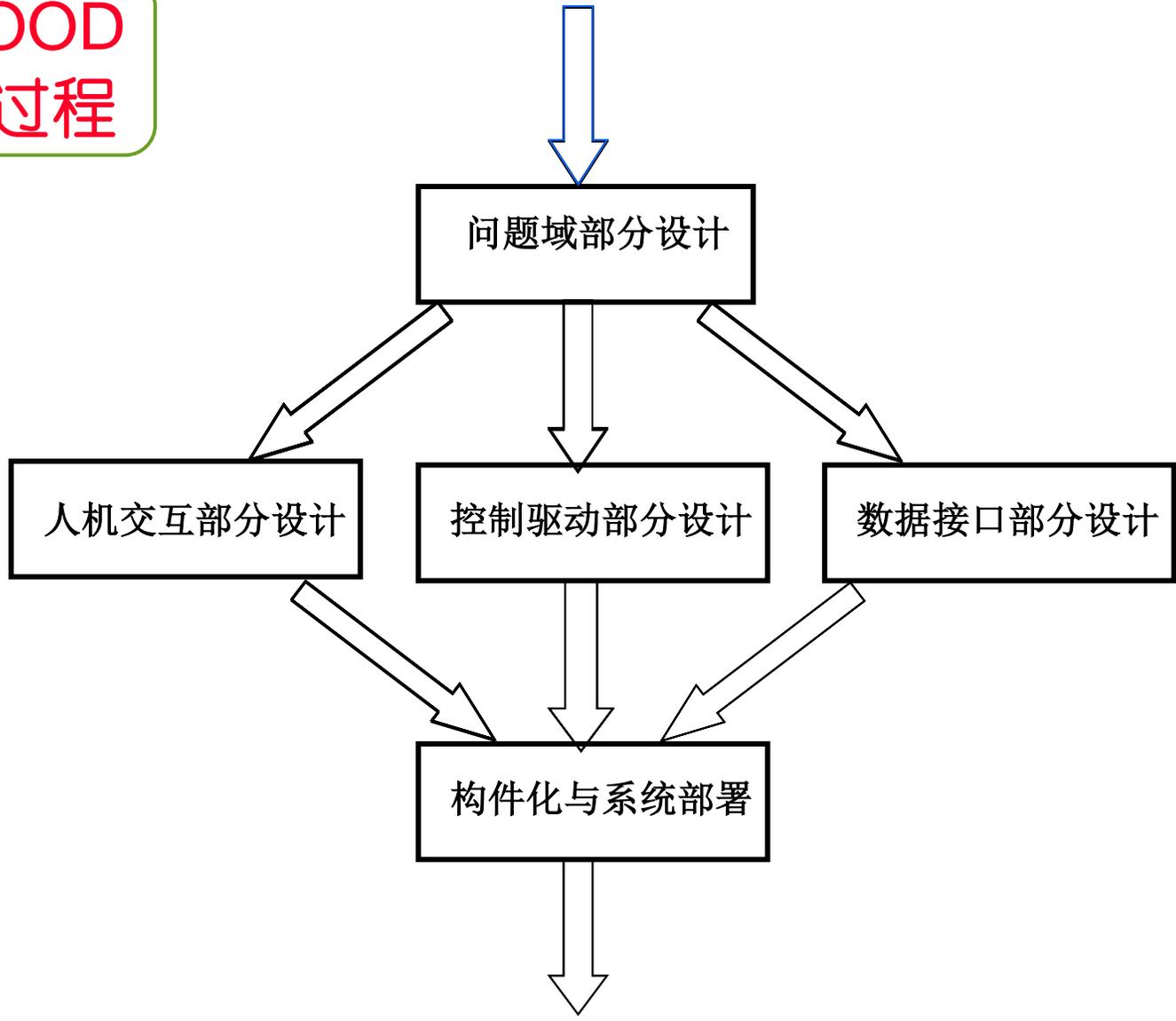


建模过程



OOD
过程

输入OOA模型



向OOP输出OOD模型

OOA与OOD的关系

一致的概念与表示法

OOA和OOD采用一致的概念和表示法，从而不存在分析与设计之间的鸿沟。

不同的内容、目标和抽象层次

OOA: 研究问题域和用户需求，运用面向对象的观点发现问题域中与系统责任有关的对象，以及对象的特征和相互关系。目标是建立一个直接映射问题域，符合用户需求的OOA模型。

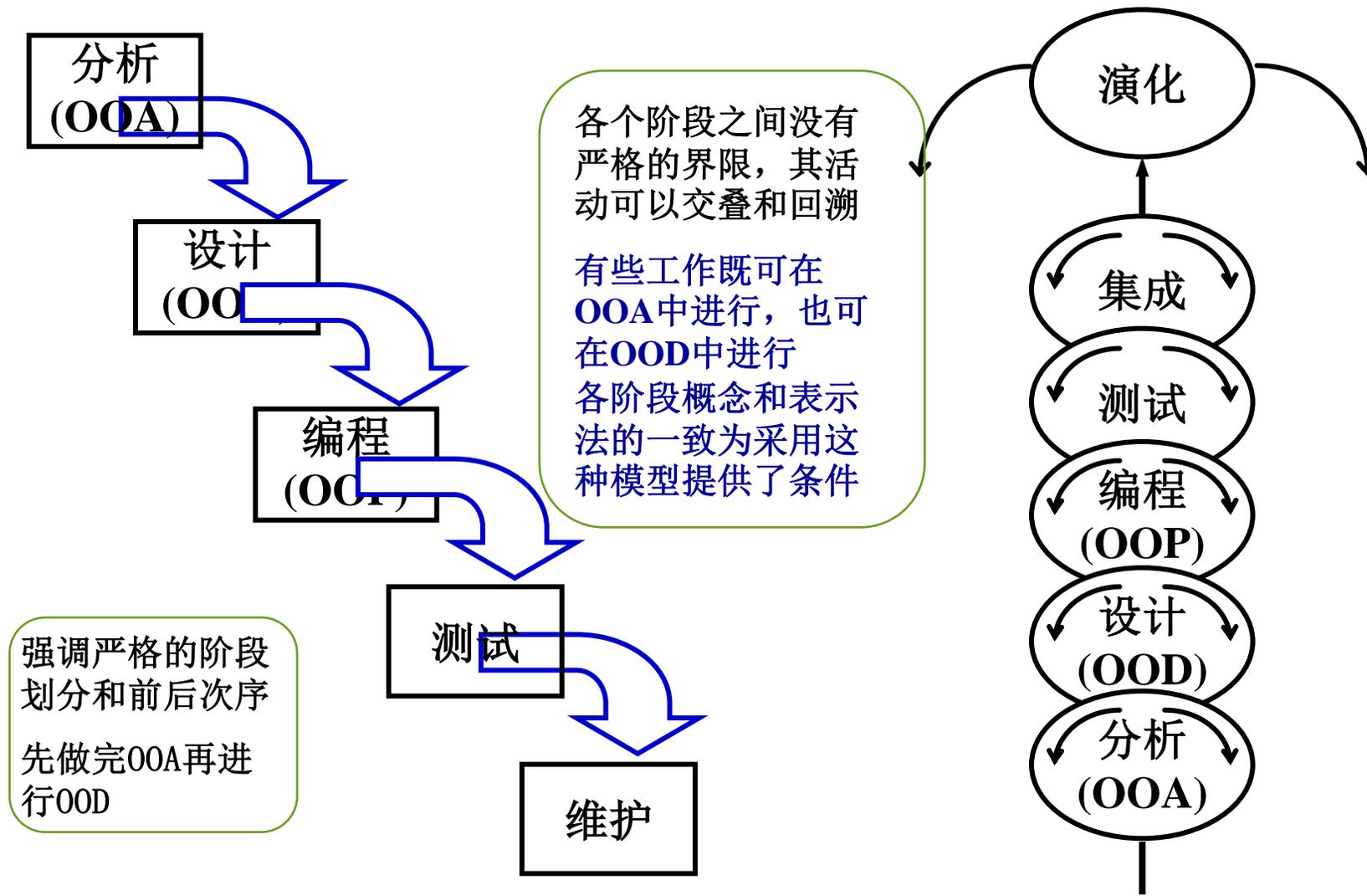
OOD: 在OOA模型基础上，针对选定的实现平台进行系统设计，按照实现的要求进行具体的设计，目标是产生一个能够在选定的软硬件平台上实现的OOD模型。

OOA模型: 抽象层次较高，忽略了与实现有关的因素

OOD模型: 抽象层次较低，包含了与实现平台有关的细节

在软件生存周期中的位置

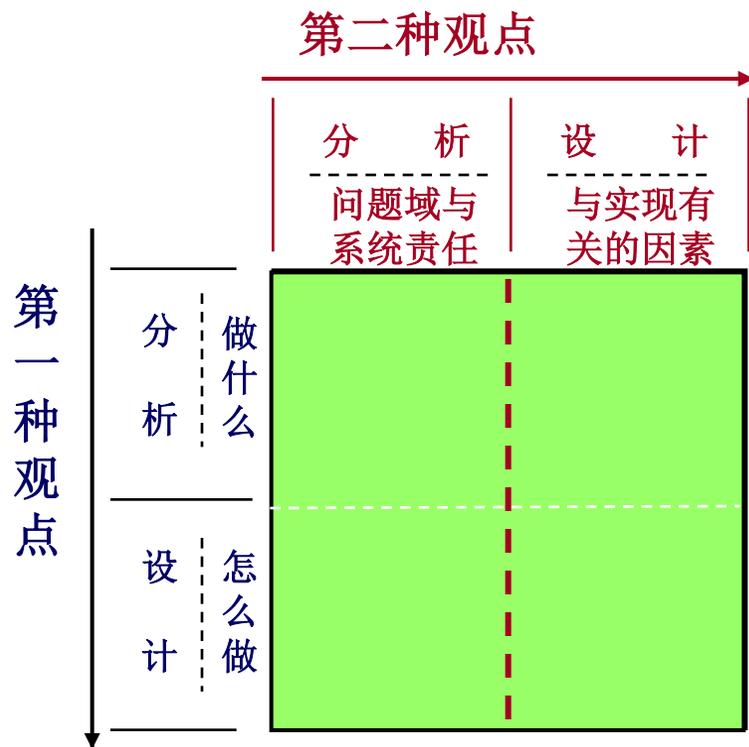
——可适应不同的生存周期模型



瀑布模型

喷泉模型

OOA与OOD的分工——两种不同的观点



关键问题：对象的特征细节（如属性的数据类型和操作流程），是在分析时定义还是在设计时定义？

第二种观点的理由：

(1) 过分强调“分析不考虑怎么做”将使某些必须在OOA考虑的问题得不到完整的认识。

(2) 把仅与问题域和系统责任有关的对象的描述在分析阶段一次完成，避免设计阶段重复地认识同一事物，减少了工作量总和。

(3) 对那些与问题域和系统责任紧密相关的对象细节，分析人员比设计人员更有发言权。

(4) 由于OOA和OOD概念和表示法的一致，不存在把细化工作留给设计人员的必然理由。

(5) OOA阶段建立平台无关的模型（PIM），OOD阶段针对不同的平台建立平台专用模型（PSM）可在最大程度上实现对OOA结果的复用。

从MDA看OOA与OOD的关系

模型驱动的体系结构（**model-driven architecture, MDA**）是**OMG**的一个技术规范，是一种加强模型能力的系统开发途径。

模型驱动（**model-driven**）：用模型来对系统的理解、设计、构造、部署、操作、维护和更改进行指导。

体系结构（**architecture**）：是对系统的部件和连接件以及这些部件通过连接件进行交互的规约。

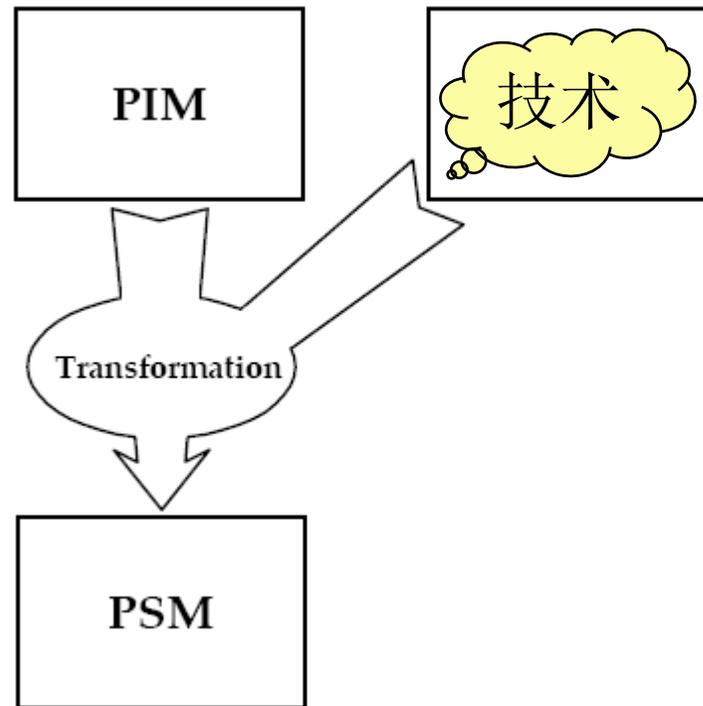
平台（**platform**）：是一组子系统和技術，它通过一些接口和专用规则提供了一个连贯的功能集合，任何由该平台支持的应用都可以使用平台所提供的功能而不必关心其实现细节。

平台无关模型（**platform independence model, PIM**）：独立于任何一种平台特征的模型。

平台专用模型（**platform specific model, PSM**）：与特定类型的平台特征有关的模型。

模型转换（**model transformation**）：由系统的一个模型转化成同一个系统的另外一个模型。它是**MDA**最为关键的部分，其核心问题是**从PIM转换到PSM**。

MDA提倡：在系统开发中首先建立平台无关模型（**PIM**），然后将它转换为平台专用模型（**PSM**）



把MDA的观点运用于OOA和OOD

OOA: 只针对问题域和系统责任，不涉及实现条件
因此可得到一个平台无关的**OOA**模型

OOD: 在**OOA**模型基础上针对特定实现条件进行设计
转换成一个平台专用的**OOD**模型

好处：使整个**OOA**模型可以在针对不同的实现平台的设计中得到复用

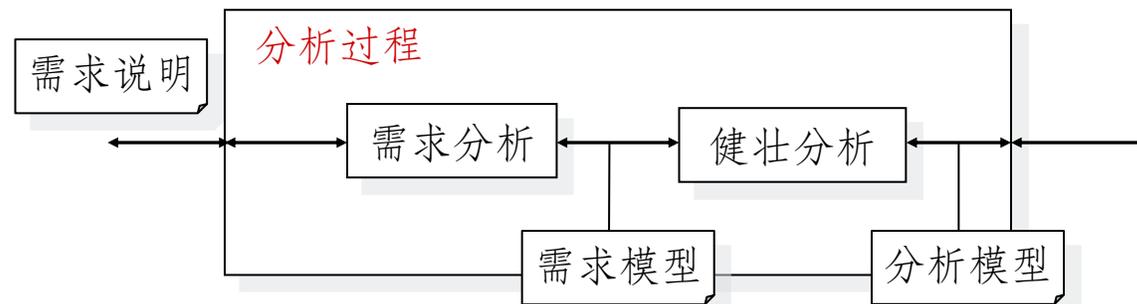
需求分析和系统分析

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。

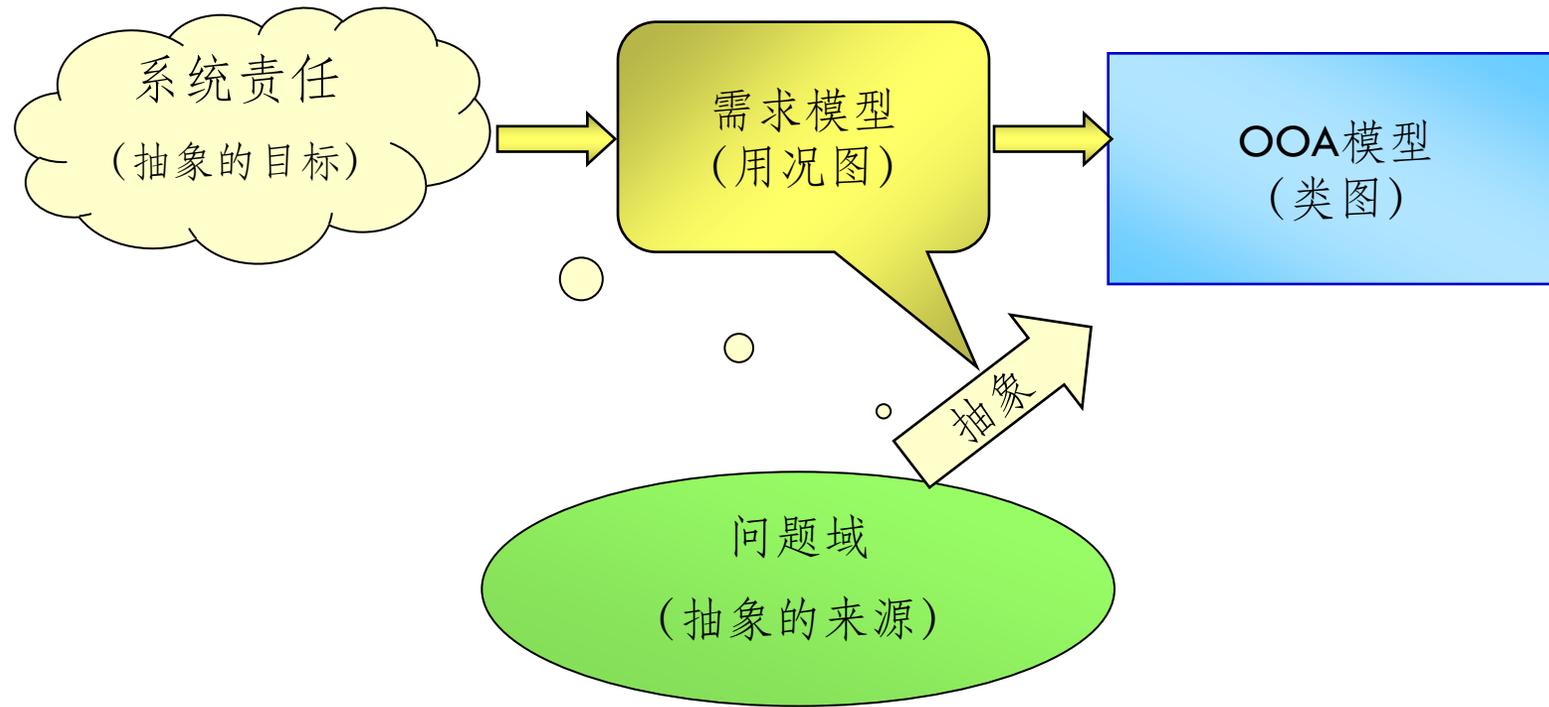
Jacobson方法（**OOSE**）提出用况（**use case**）概念，解决了对需求的描述问题，其分析过程如下：



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

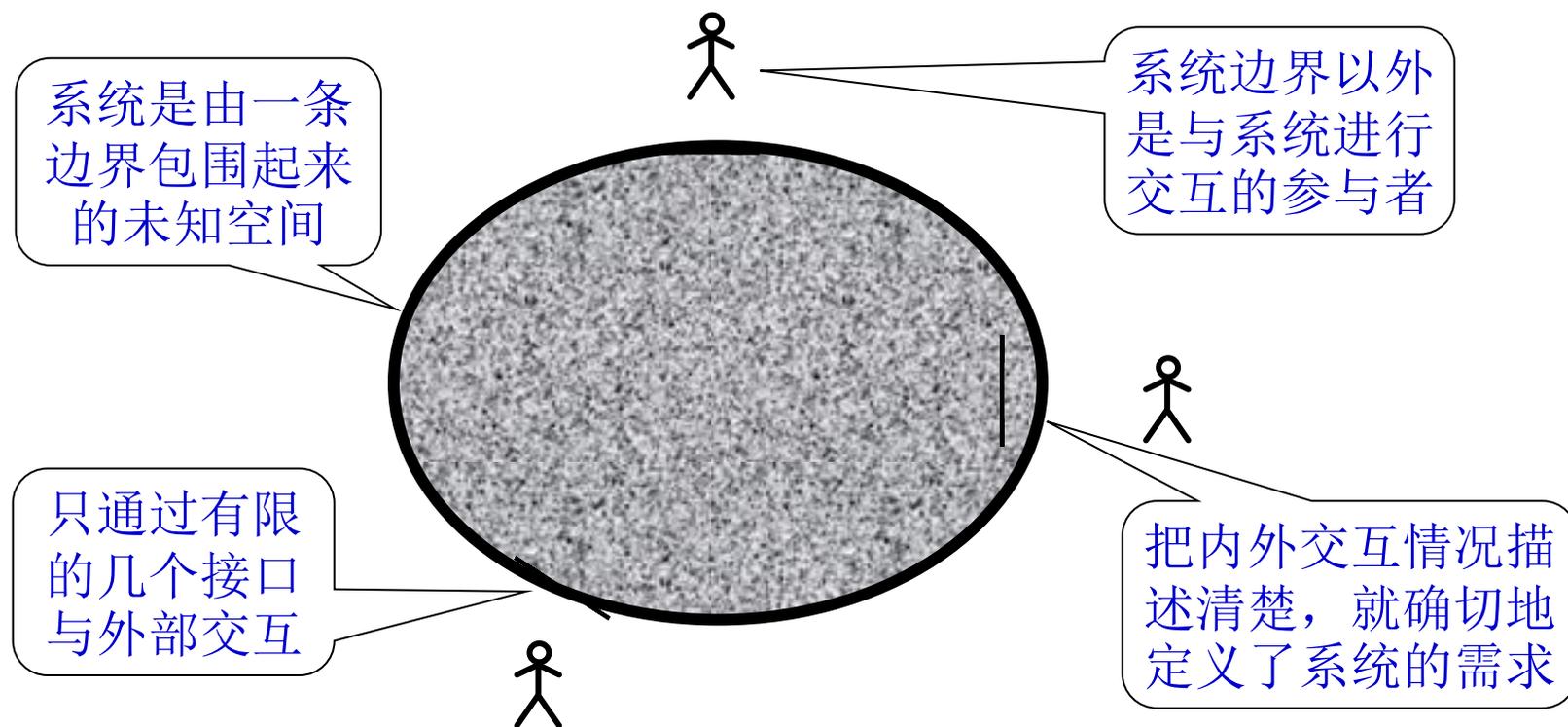
用况的概念解决了对需求的描述问题



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。



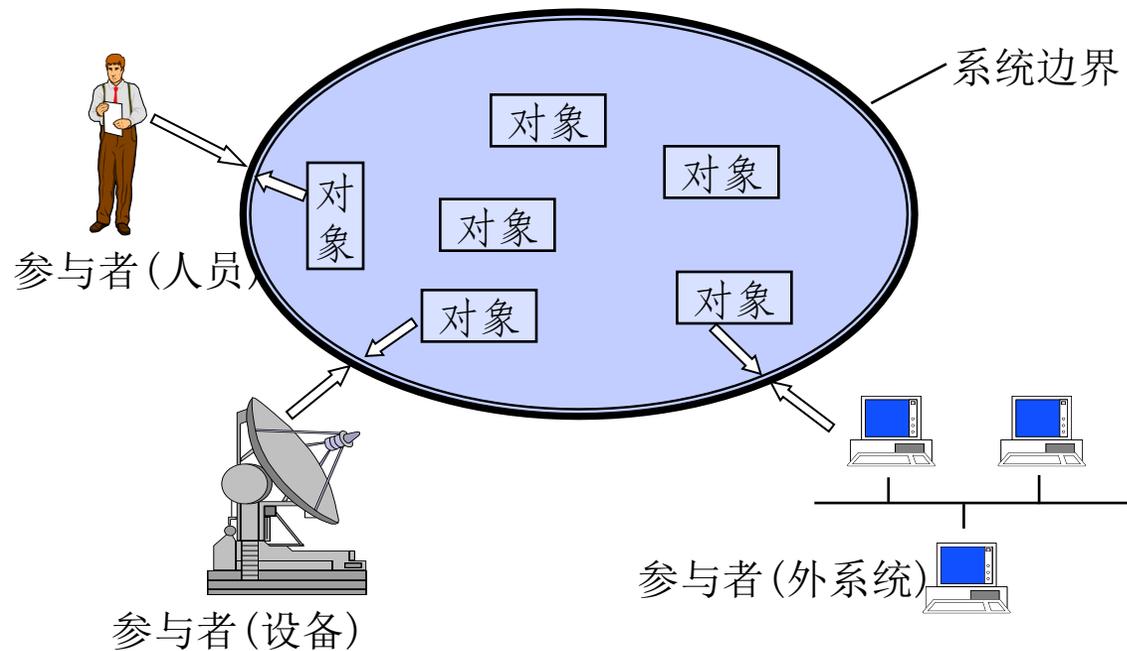
系统边界与参与者

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。

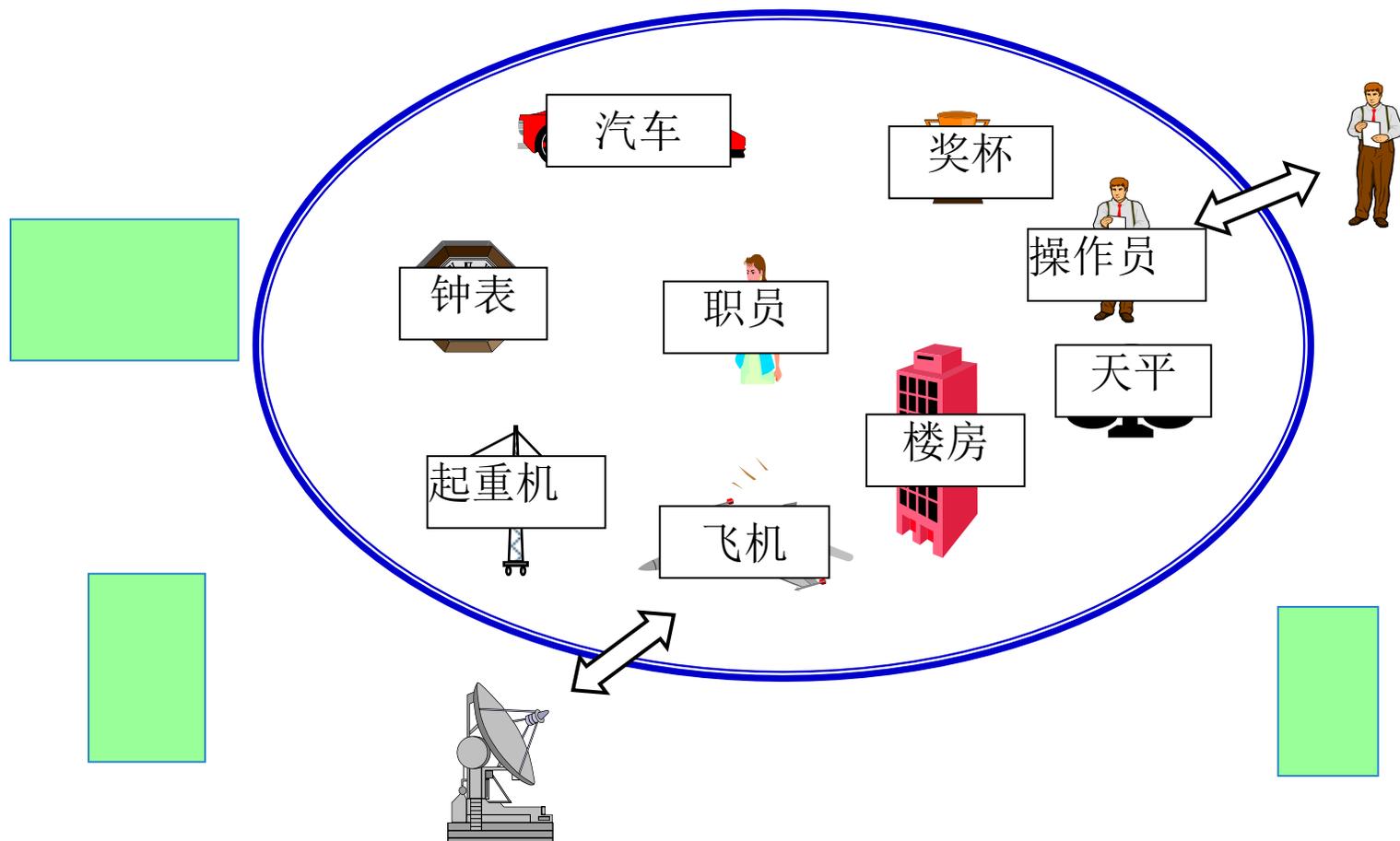
系统成分：在OOA和OOD中定义并且在编程时加以实现的系统元素——对象

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和系统交互
- (3) 既是系统中的对象，本身又作为参与者和系统交互
- (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

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。

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

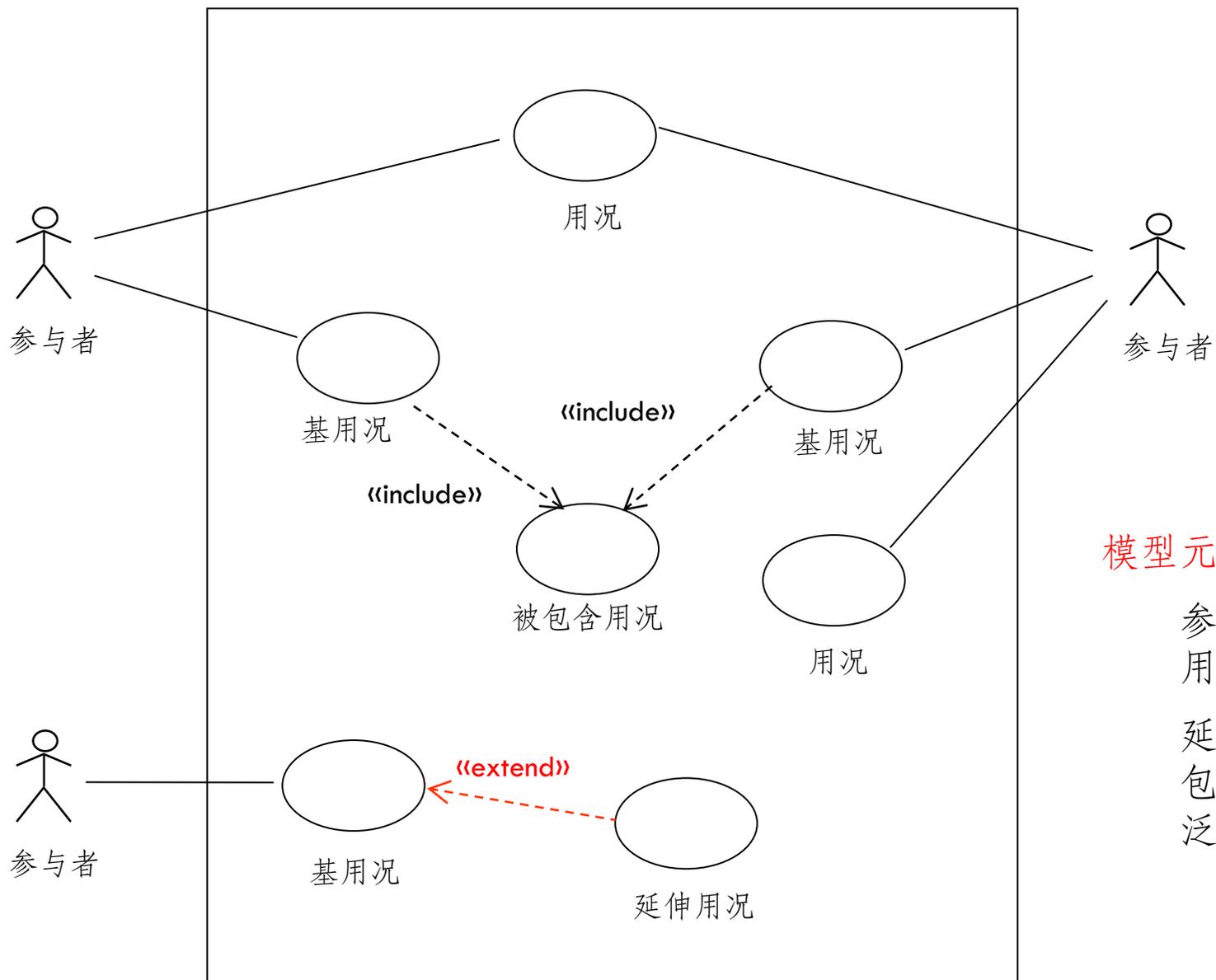
针对单个用况的描述策略：

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。

用况图



模型元素:

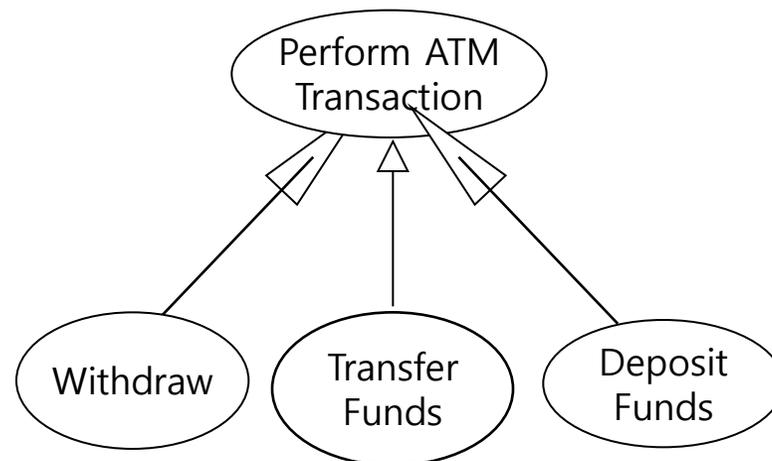
参与者
用况
延伸
包含
泛化

用况之间的关系

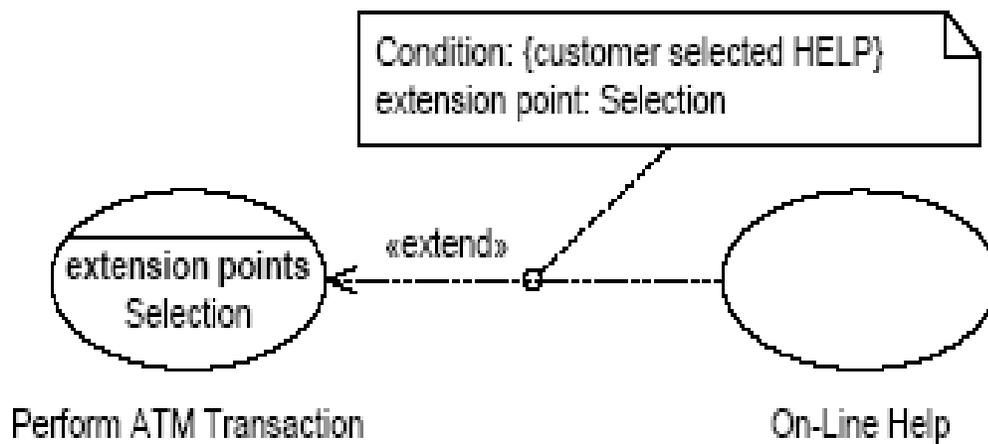
——包含、延伸、泛化



包含



泛化



延伸

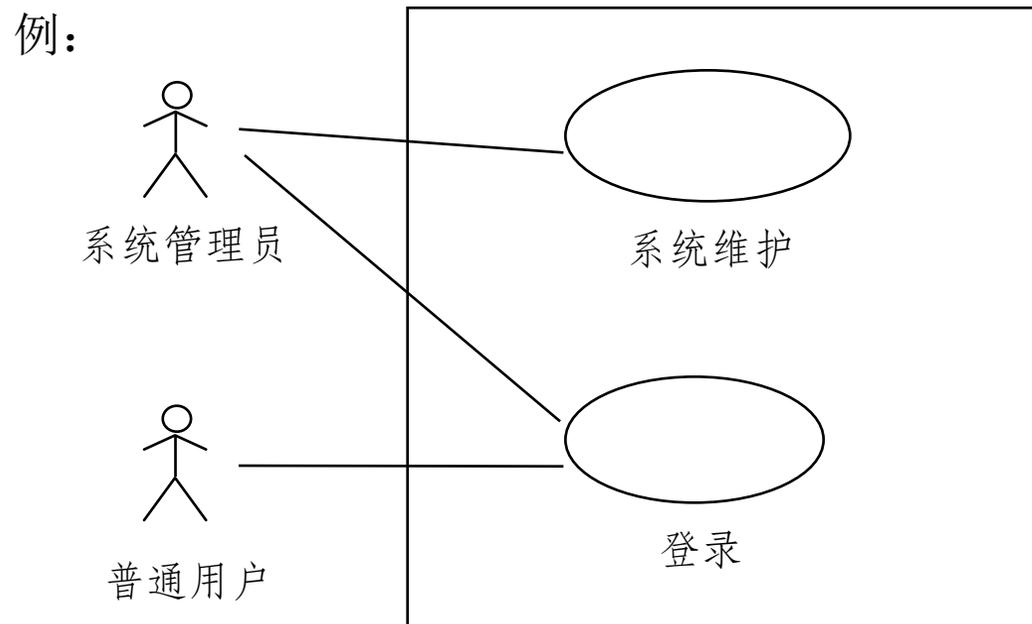
问题:

- 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

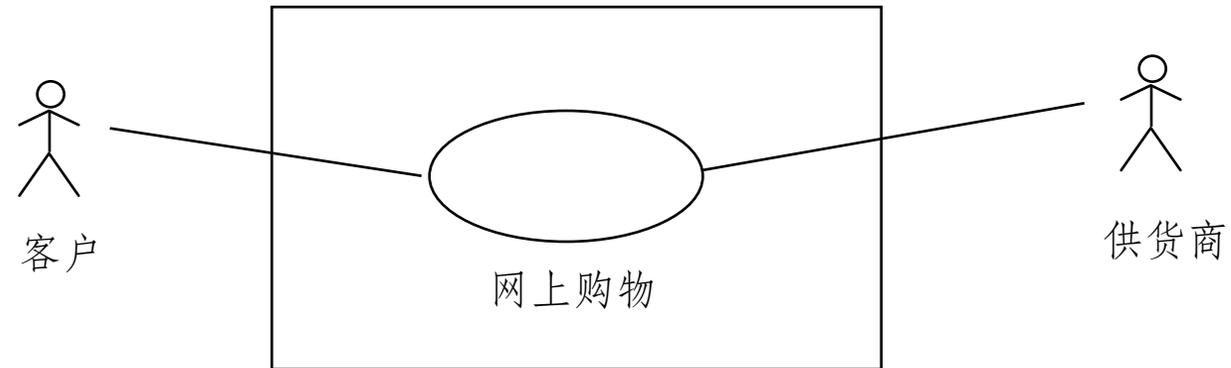
1、两个（或多个）参与者共享一个用况

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



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



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

发现对象，定义对象类

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

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

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

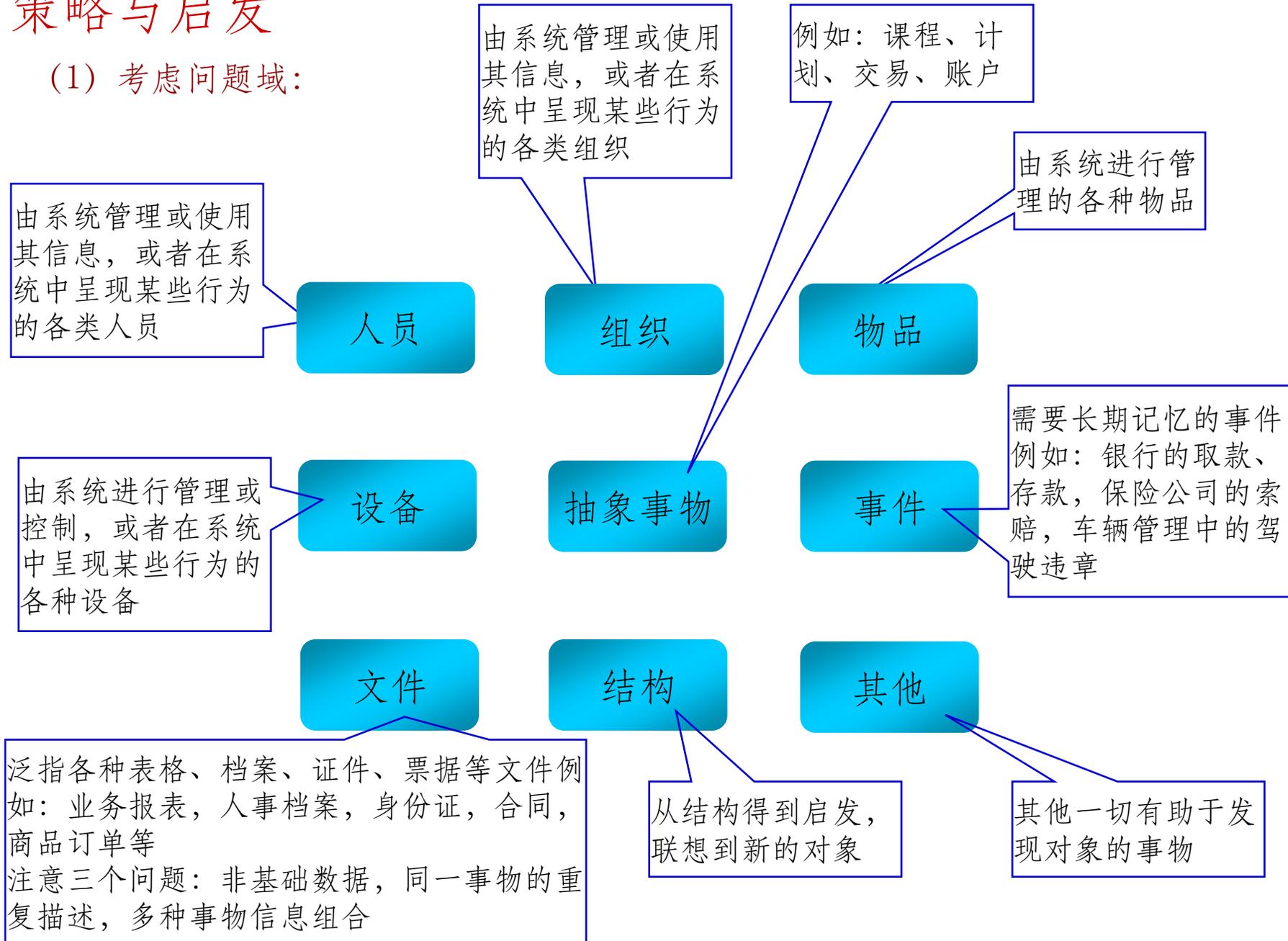
在书店管理系统中以一种书作为一个对象实例

如何发现各种有用的候选对象？

主要策略：从问题域、系统边界和系统责任3个方面考虑各种能够启发自己发现对象的因素，找出可能有用的候选对象。

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
考虑通过哪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？

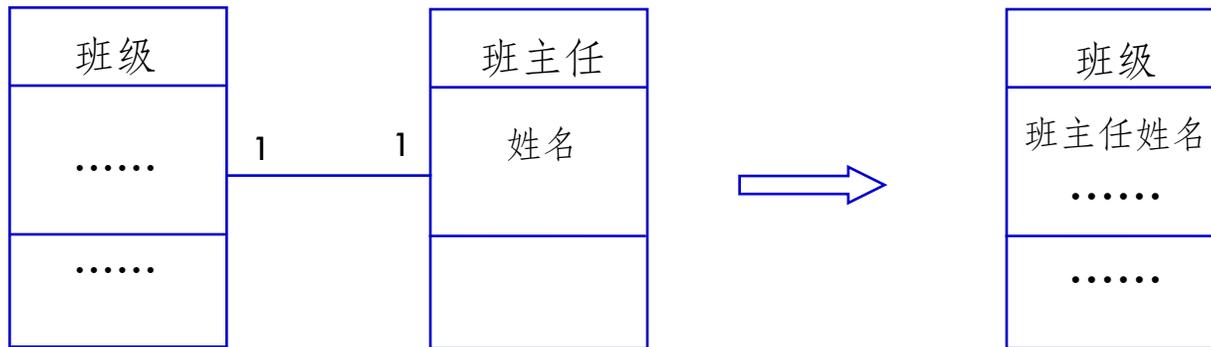
通过操作判断：

是否通过操作提供了某些有用的功能？

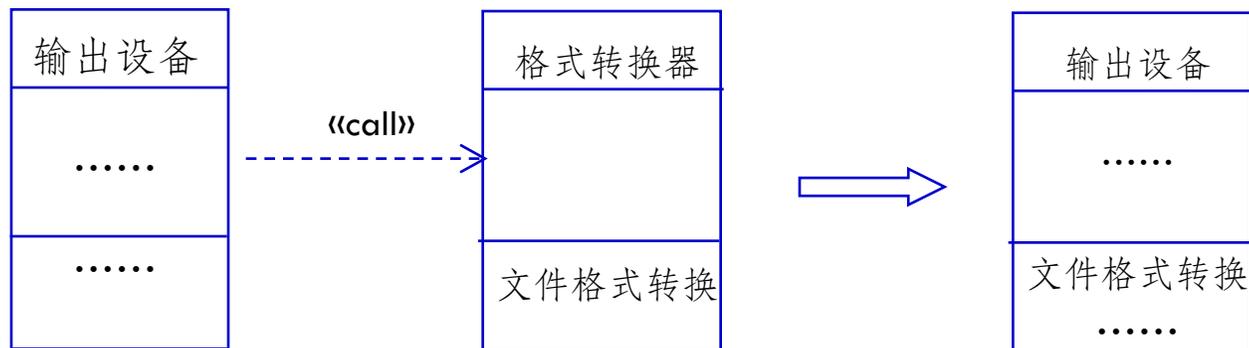
二者都不是——无用

(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：

图形用户界面 (GUI)

数据管理系统

硬件

操作系统 有关的对象

——推迟到OOD考虑

对象分类

(1) 将对象抽象为类，用类表示它的全部对象

为每一组具有相同属性和操作的对象定义为一个类，用一个类符号表示。

(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（服装和计算机）

——考虑能否合并为一个类

属性及操作相似的类

例如：“轿车”和“货车”

——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

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

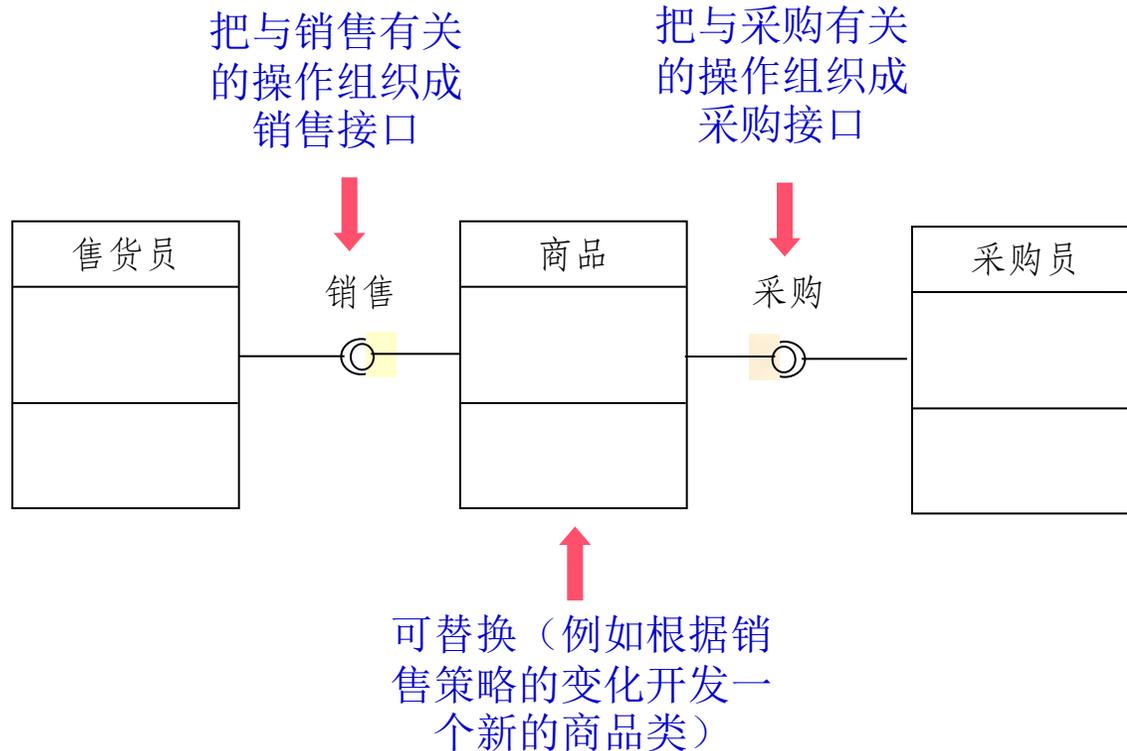
UML对接口的定义及解释：

“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) :

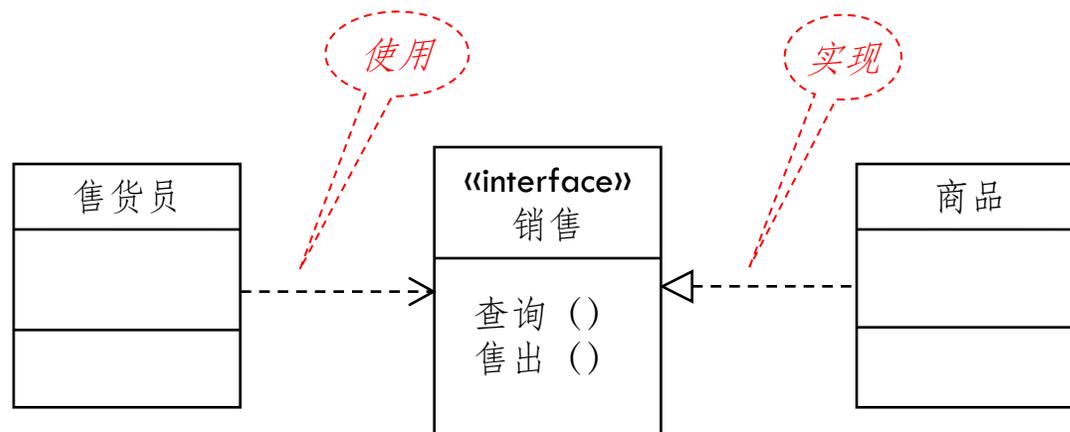


接口与类的关系

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）



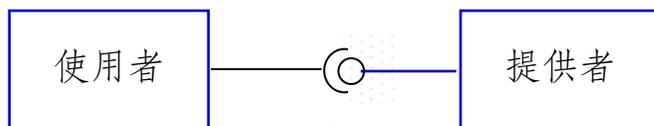
同一个接口

对实现者而言是**供接口**（provided interface）

对使用者而言是**需接口**（required interface）

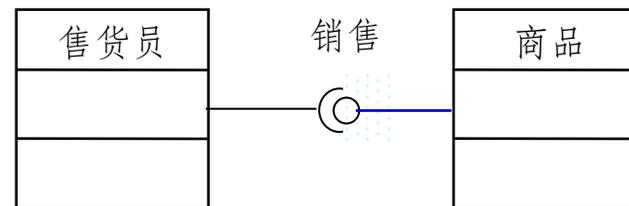
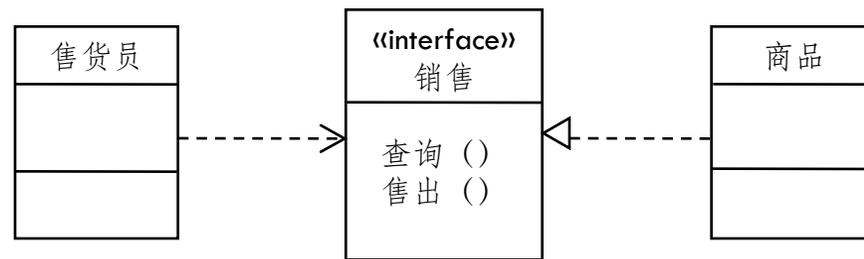
表示法（简略方式）：

——托球-托座

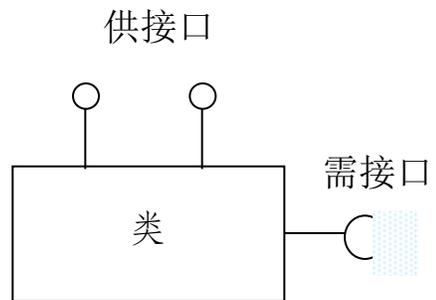


提供者的供接口（托球）
使用者的需接口（托座）

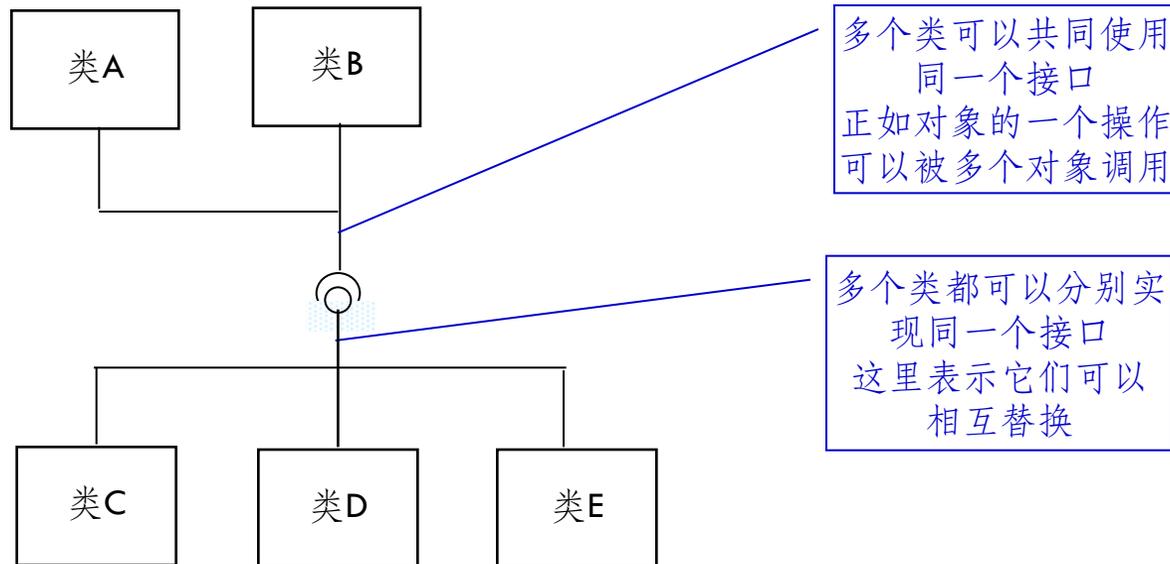
例：



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

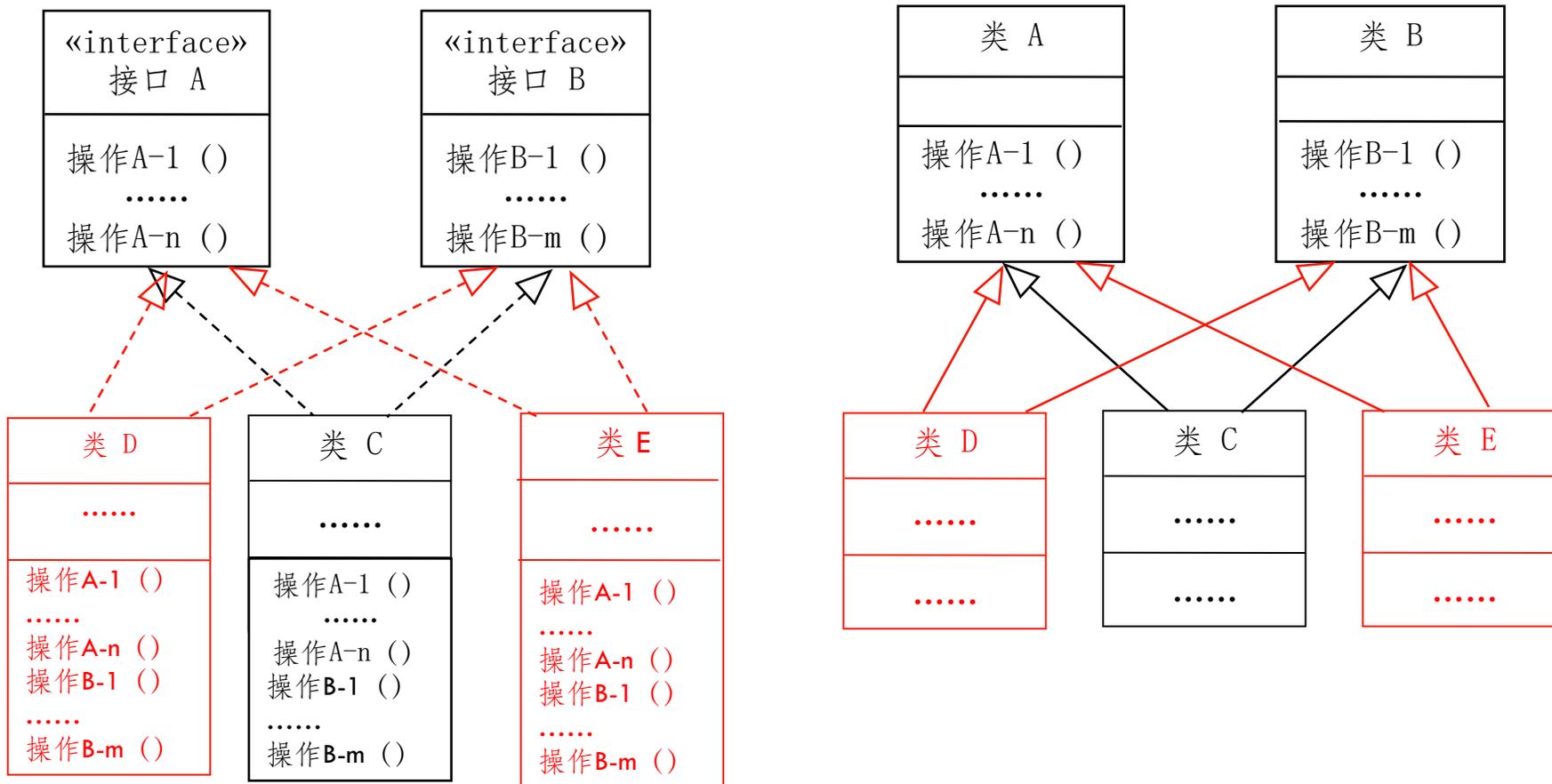
在接口的使用者和提供者之间建立了一种灵活的衔接机制，有利于对类、构件等软件成分进行灵活的组装和复用。

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

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

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。

相关概念：一般类、特殊类、继承、多继承、多态

语义：“is a kind of”

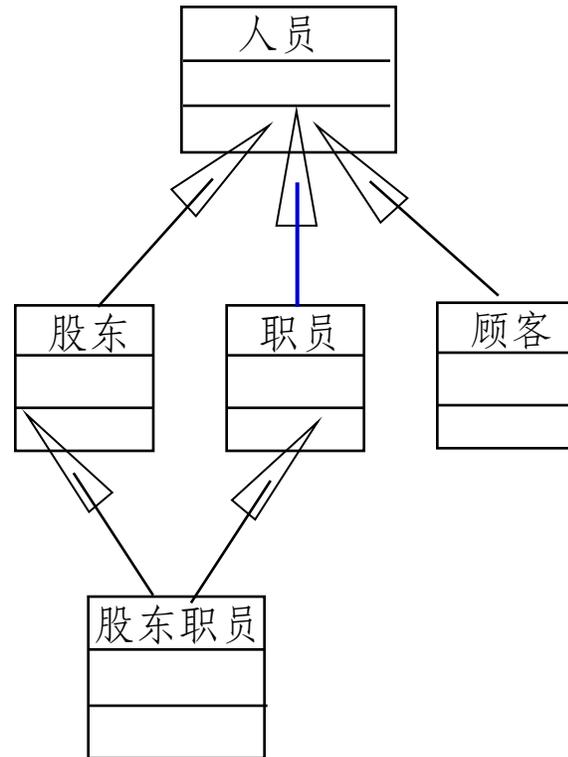
一般-特殊关系（继承关系）是类之间的一种二元关系

——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

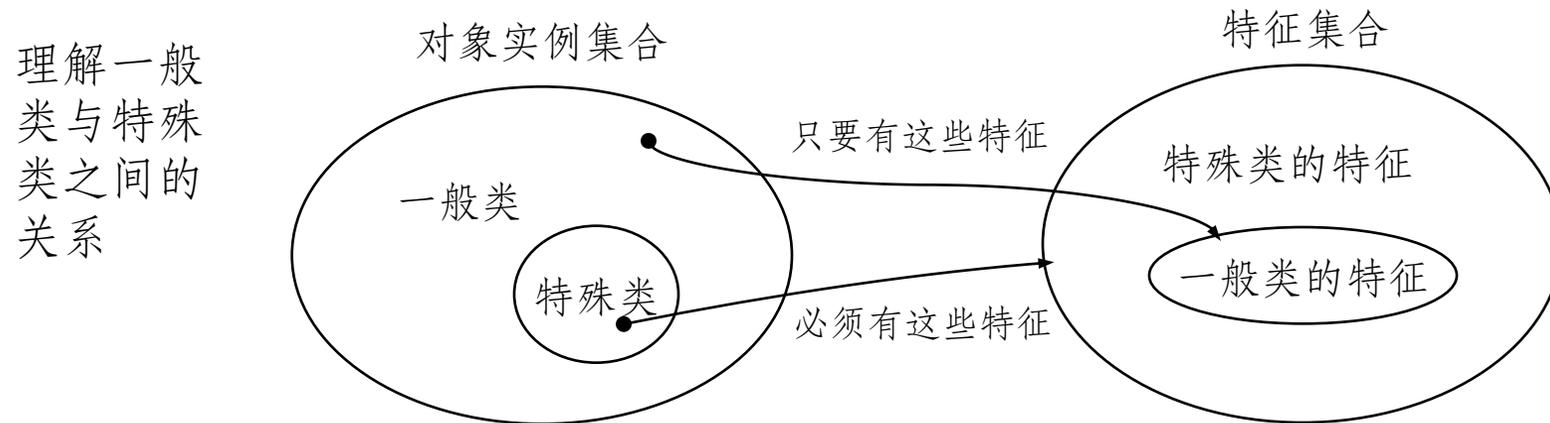
包含5个一般特殊关系

一般类和特殊类的两个定义

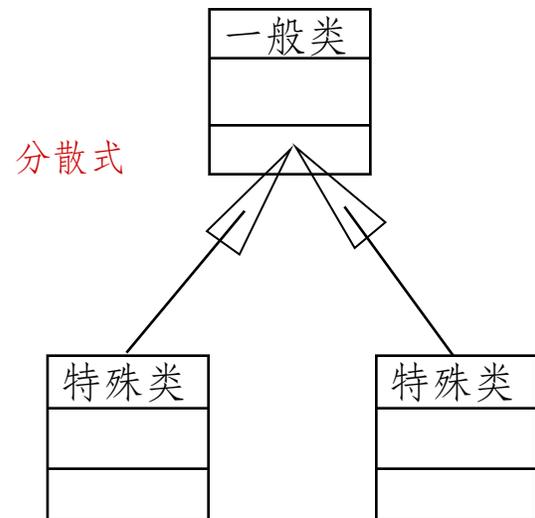
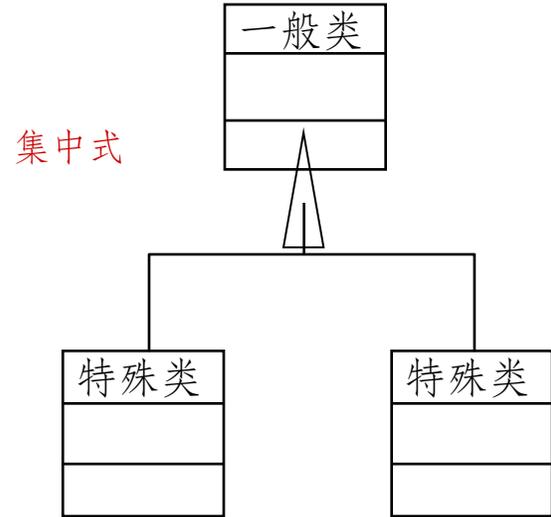
定义1: 如果类A具有类B的全部属性和全部操作，而且具有自己特有的某些属性或操作，则A叫做B的特殊类，B叫做A的一般类。一般类与特殊类又称父类与子类。

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。

——书中证明，以上两种定义是等价的

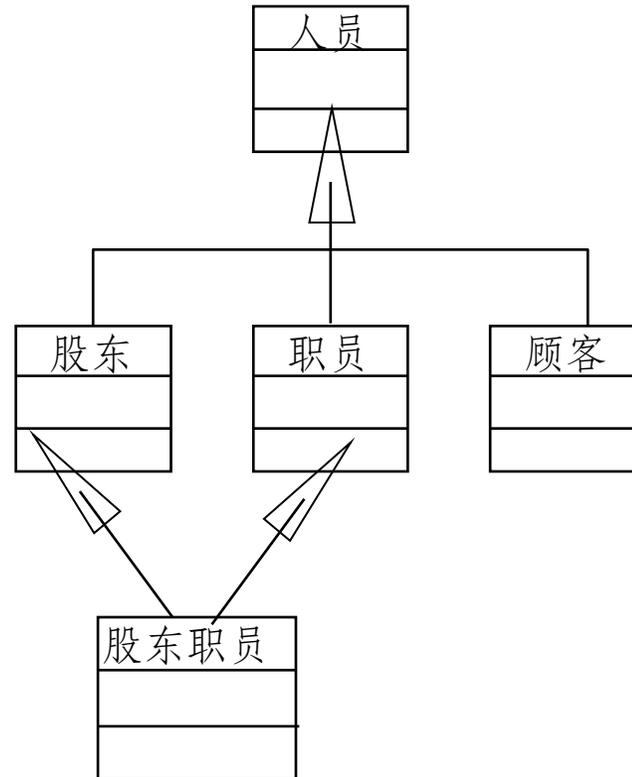


表示法



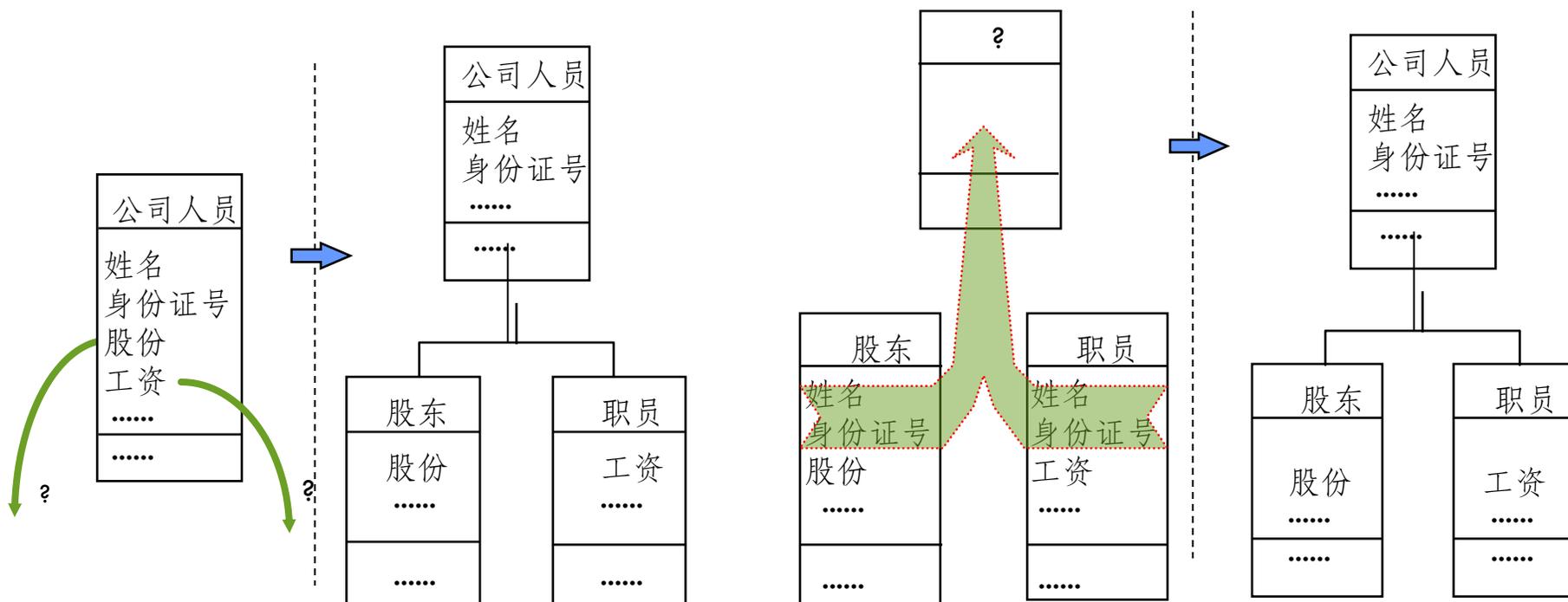
- * 对继承的属性或操作重新定义
 - × 拒绝继承
- 多态性的表示符号

例：

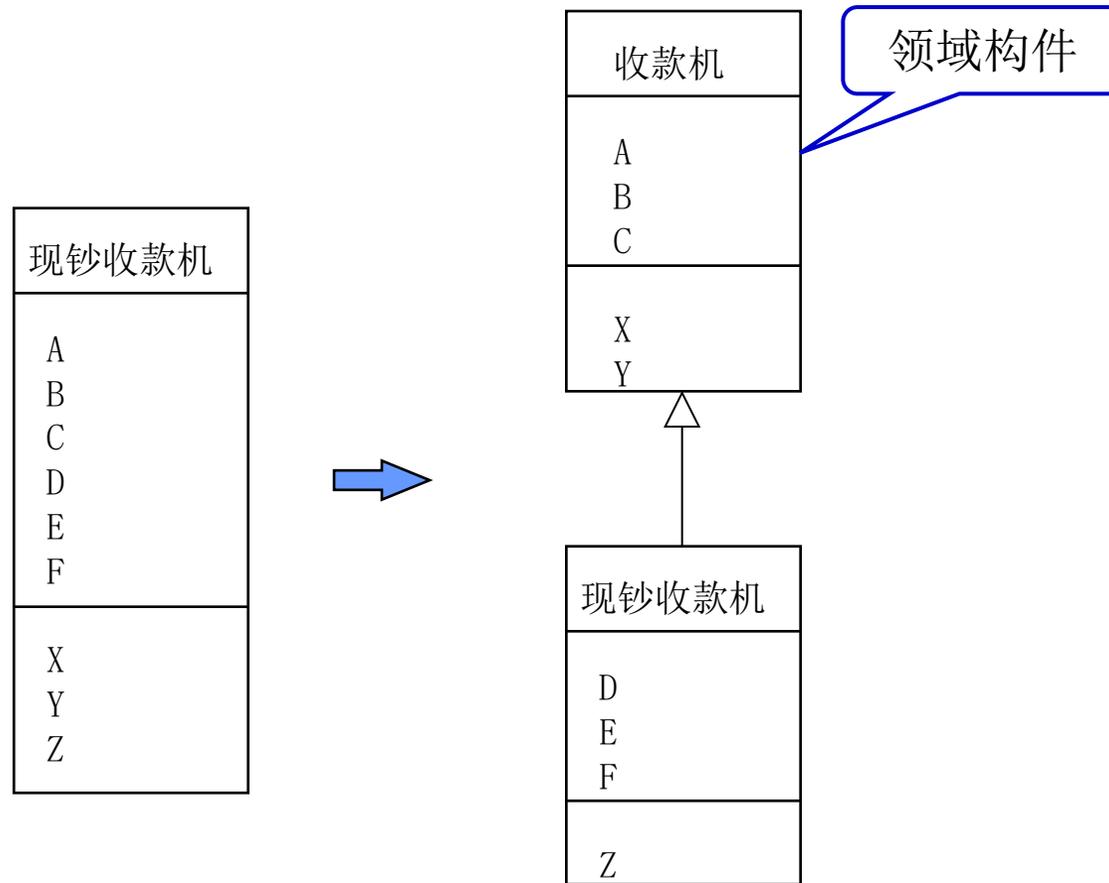


如何发现一般-特殊结构

- (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

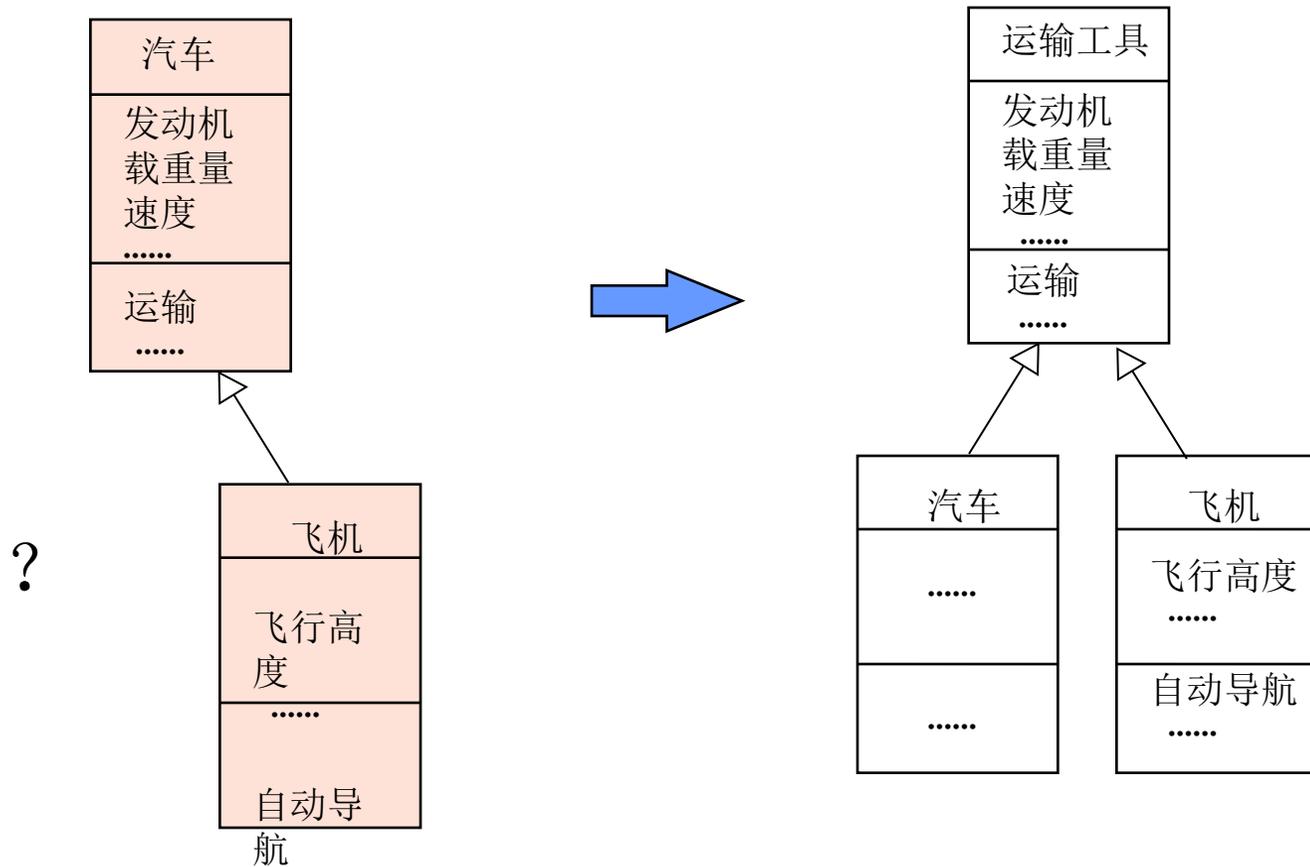


(5) 考虑领域范围内的复用



审查与调整

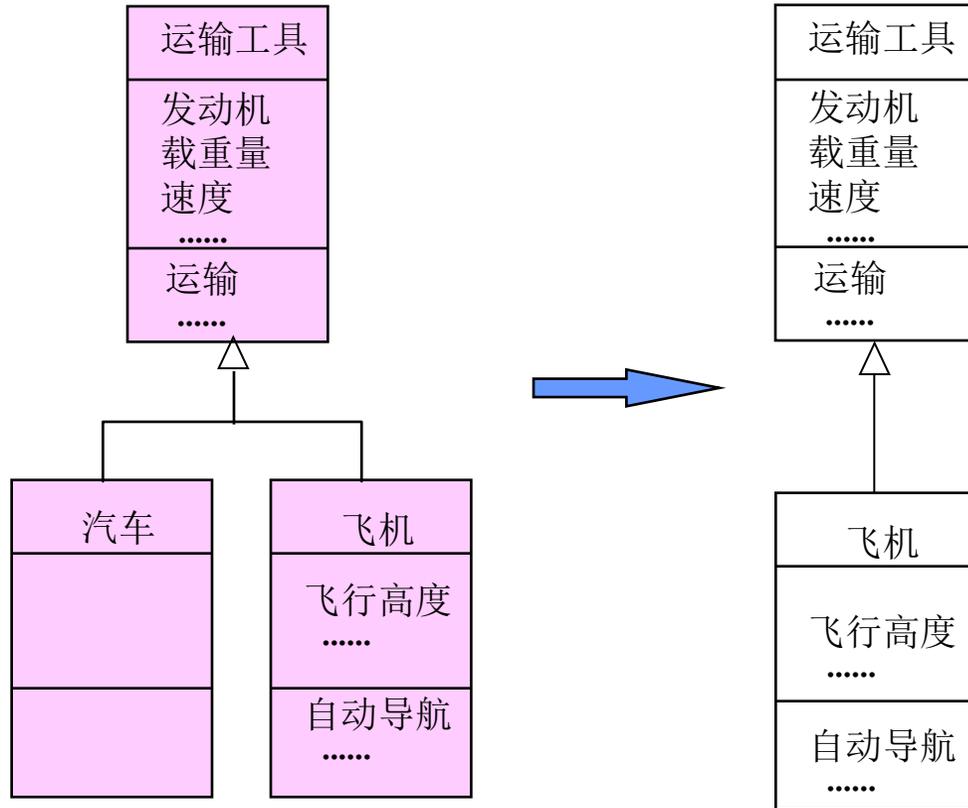
- (1) 问题域是否需要这样的分类？（例：书—线装书）
- (2) 系统责任是否需要这样的分类？（例：职员—本市职员）
- (3) 是否符合分类学的常识？（用“is a kind of”来衡量）



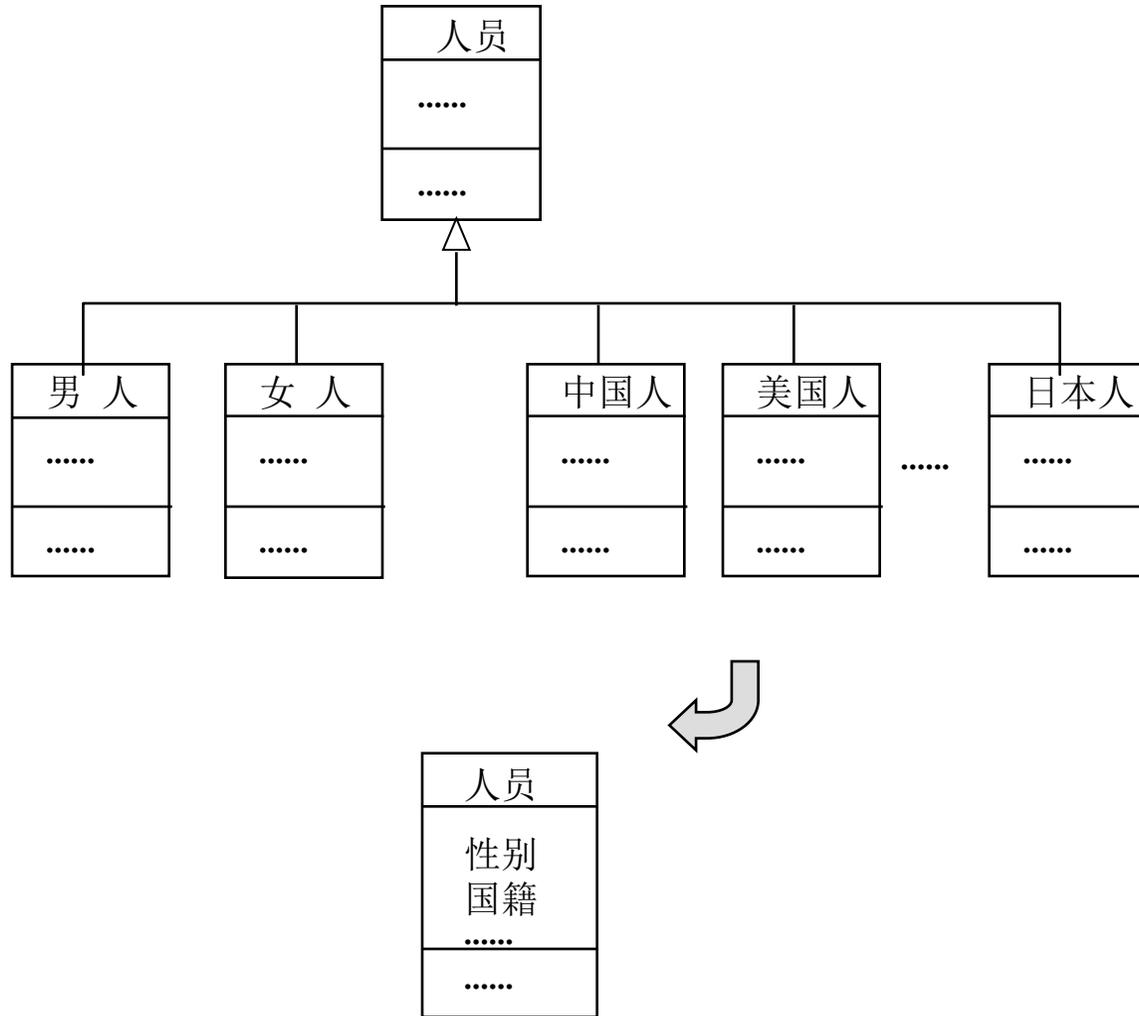
- (4) 是否真正的继承了一些属性或操作？

一般-特殊结构的简化

(1) 取消没有特殊性的特殊类



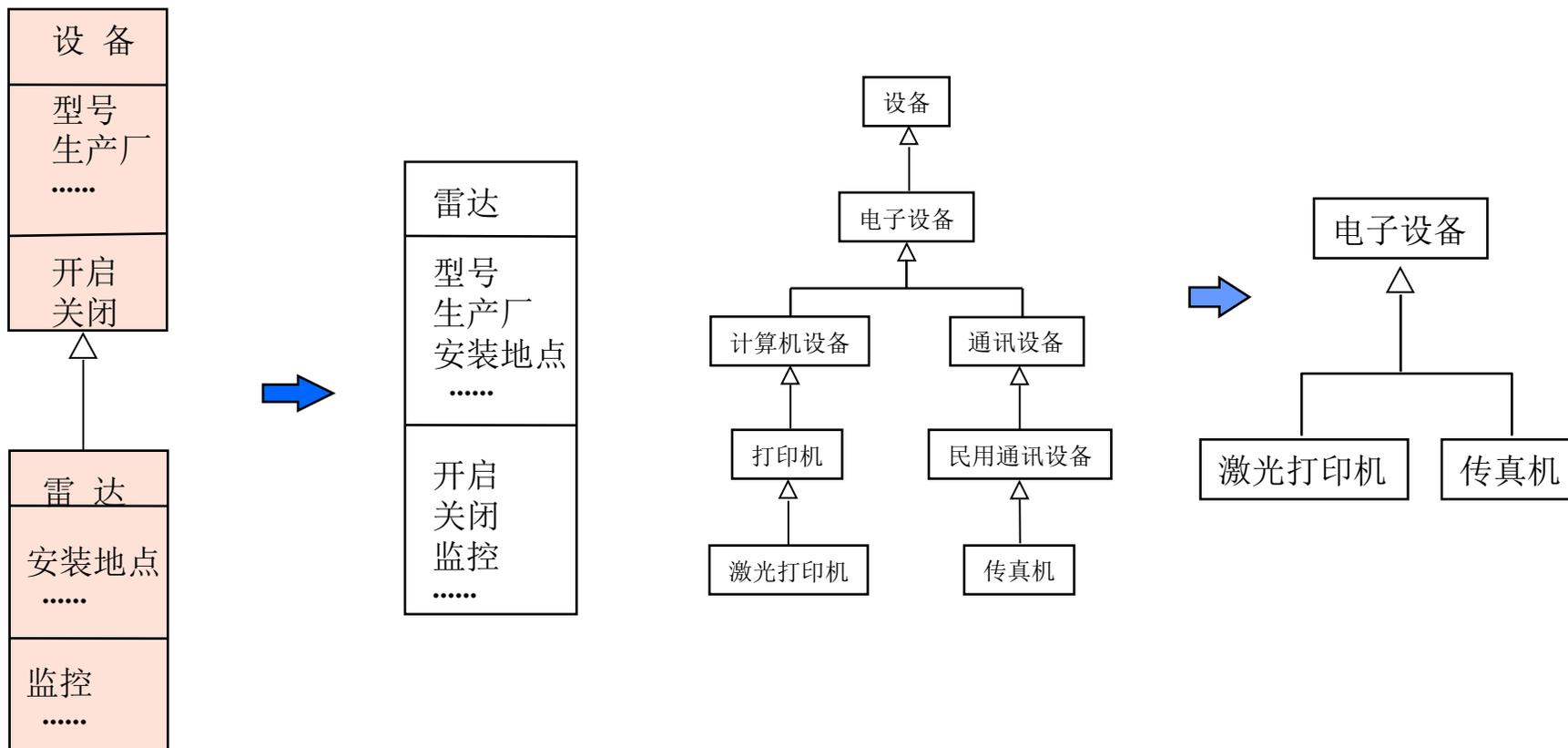
(2) 增加属性简化一般—特殊结构



(3) 取消用途单一的一般类，减少继承层次

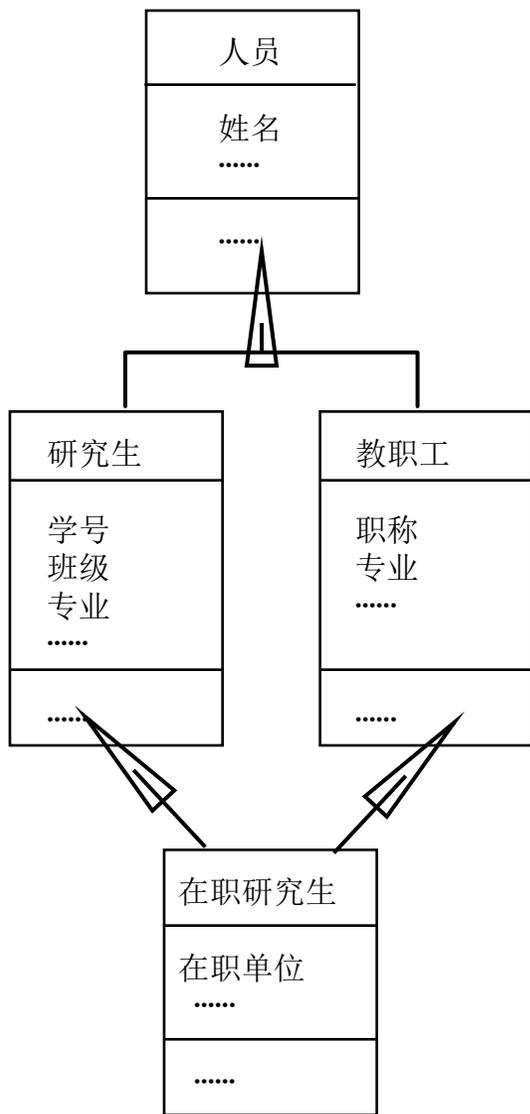
一般类存在的理由：

- * 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

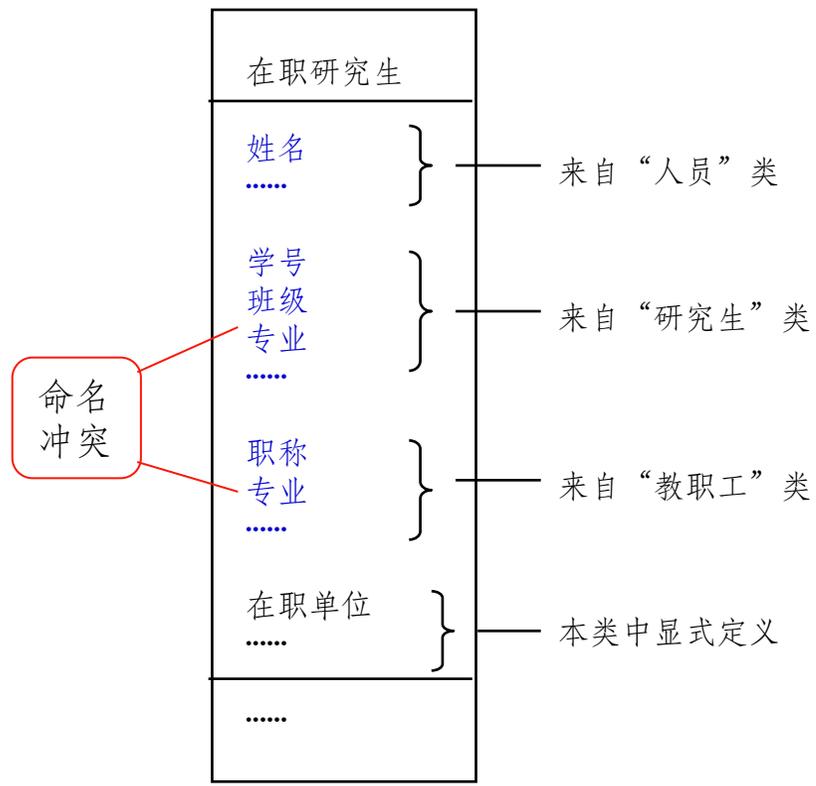


多继承：允许一个特殊类具有一个以上一般类的继承模式

例：

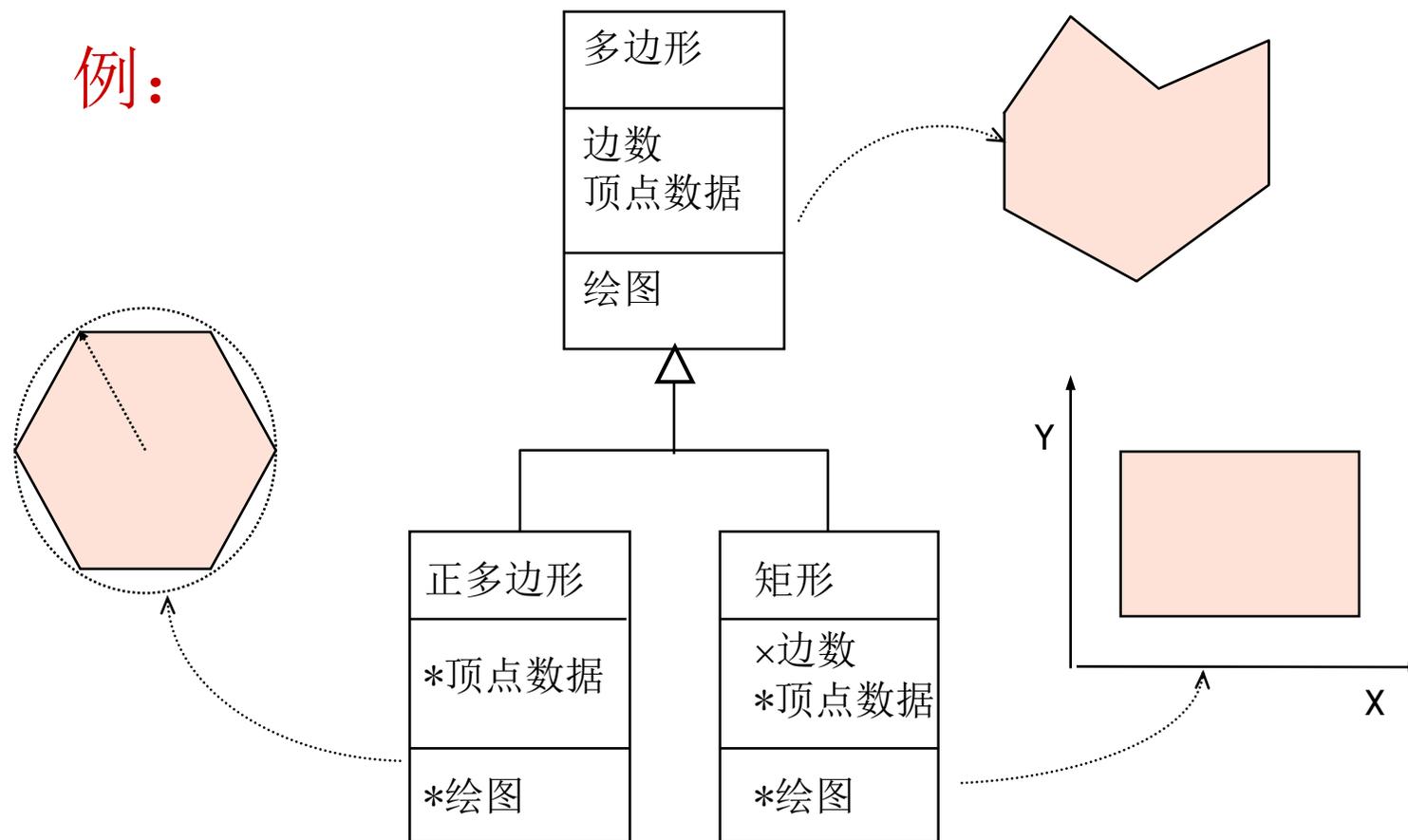


多继承特殊类的内部情况



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：



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

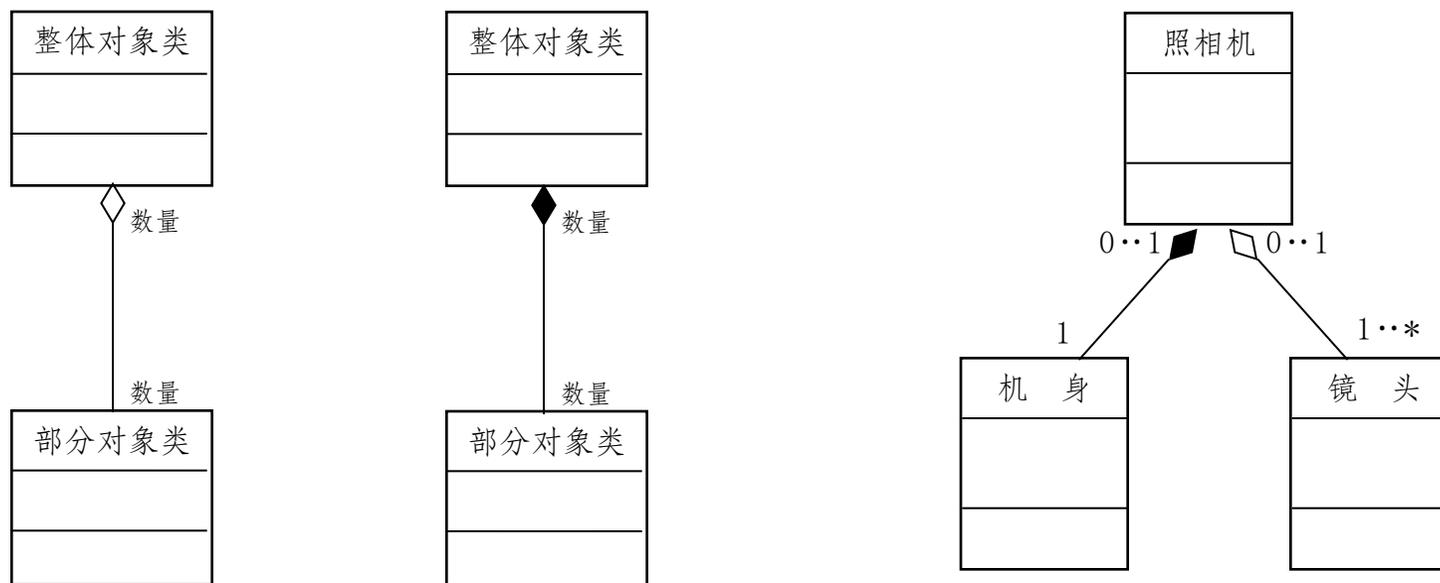
“类 A 是类 B 的一个组成部分”

——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

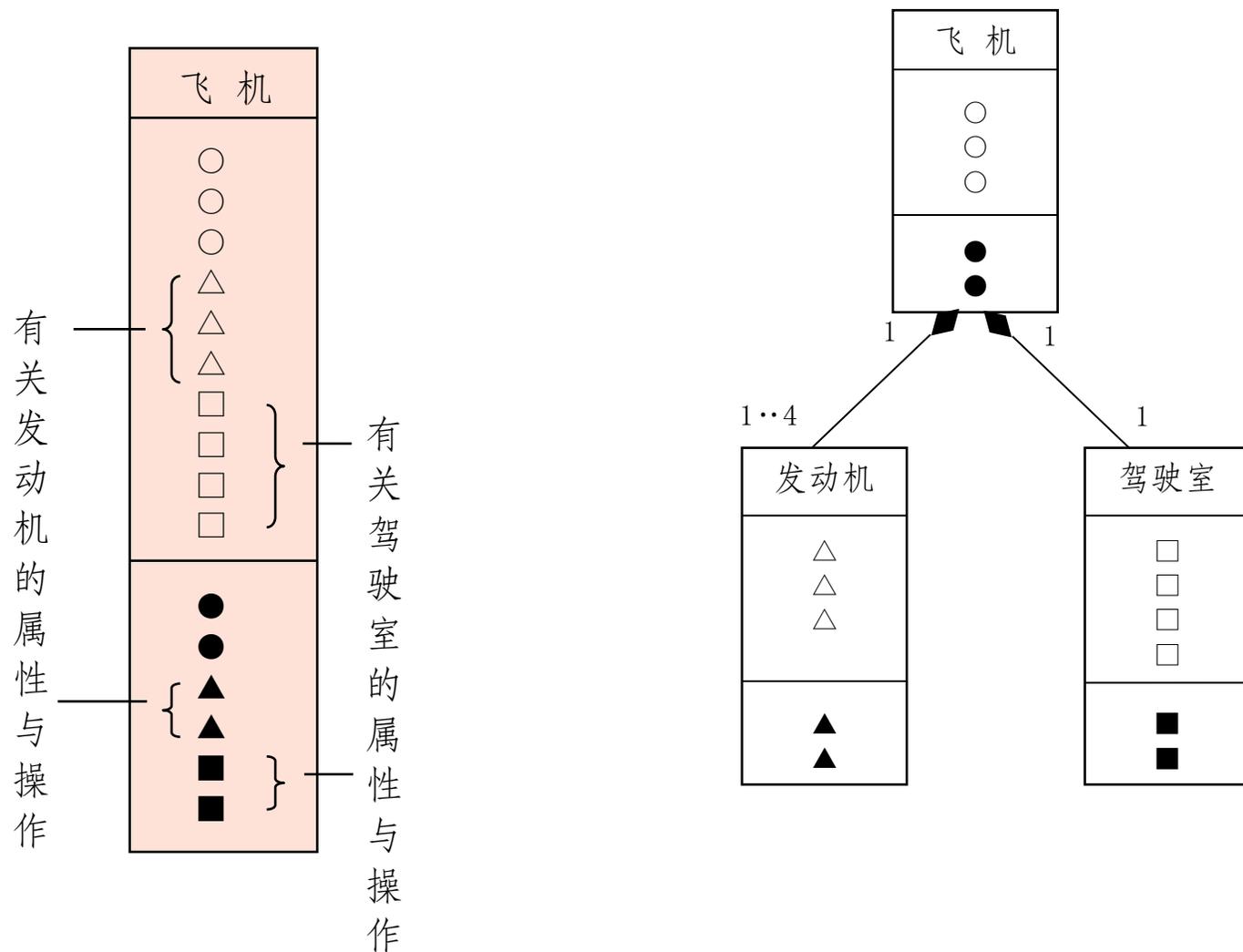
- (1) 物理上的整体事物和它的组成部分
例：机器、设备和它的零部件
- (2) 组织机构和它的下级组织及部门
例：公司与子公司、部门
- (3) 团体（组织）与成员
例：公司与职员
- (4) 一种事物在空间上包容其它事物
例：生产车间与机器
- (5) 抽象事物的整体与部分
例：学科与分支学科、法律与法律条款
- (6) 具体事物和它的某个抽象方面
例：人员与身份、履历

审查与筛选

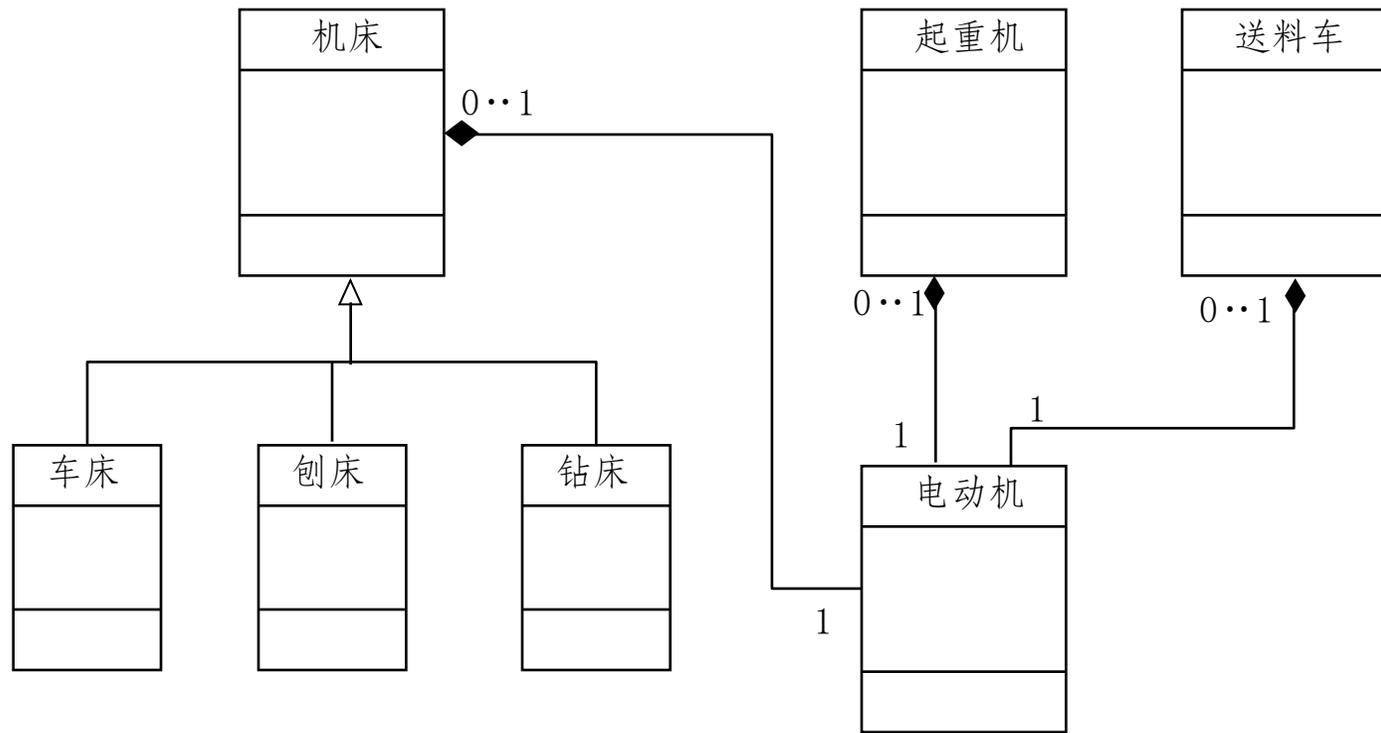
- (1) 是否属于问题域?
例：公司职员与家庭
- (2) 是不是系统责任的需要?
例：员工与工会
- (3) 部分对象是否有一个以上的属性?
例：汽车与车轮（规格）
- (4) 是否有明显的整体-部分关系?
例：学生与课程

整体-部分结构的高级应用技巧

(1) 简化对象的定义



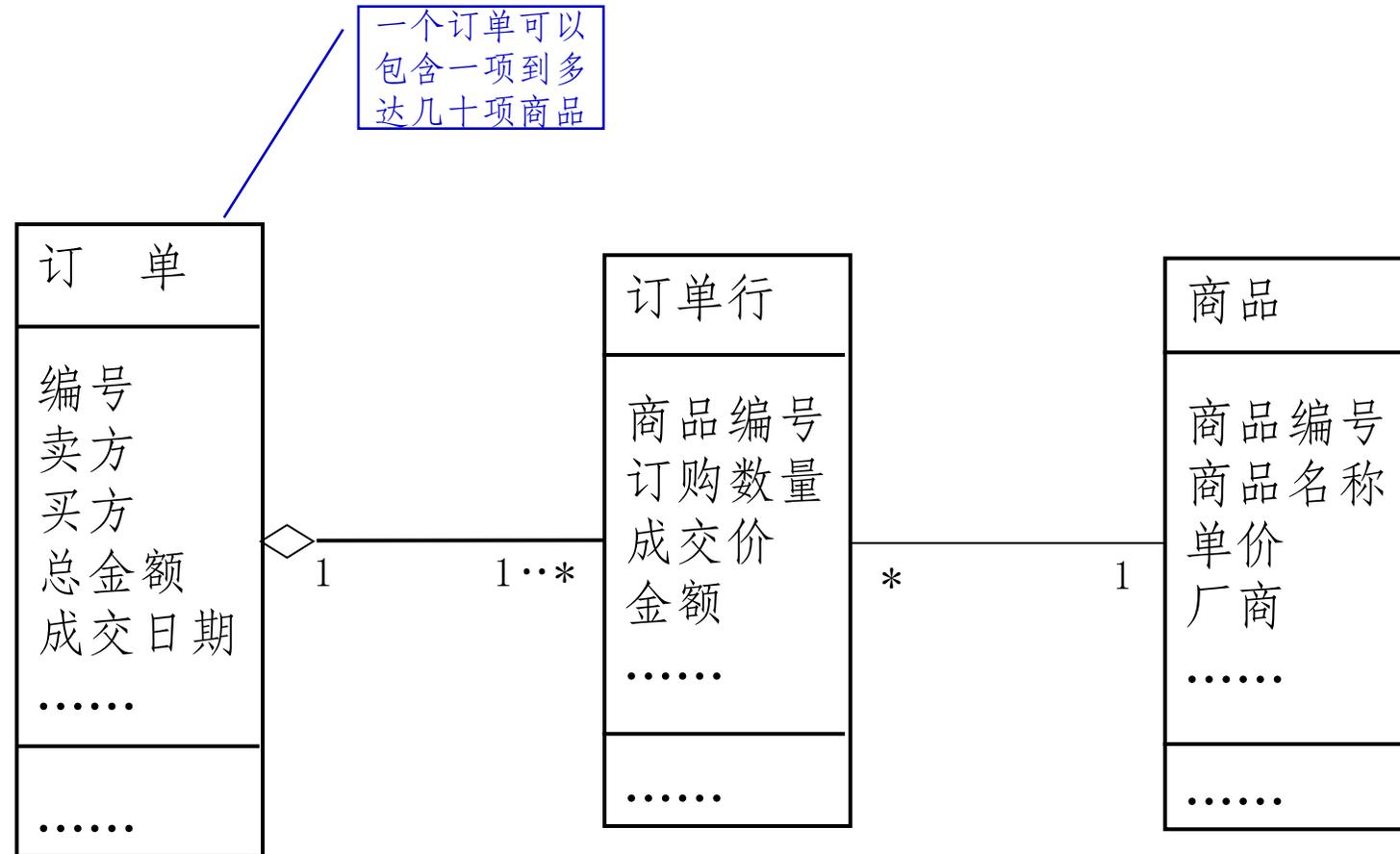
(2) 支持软件复用



(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?

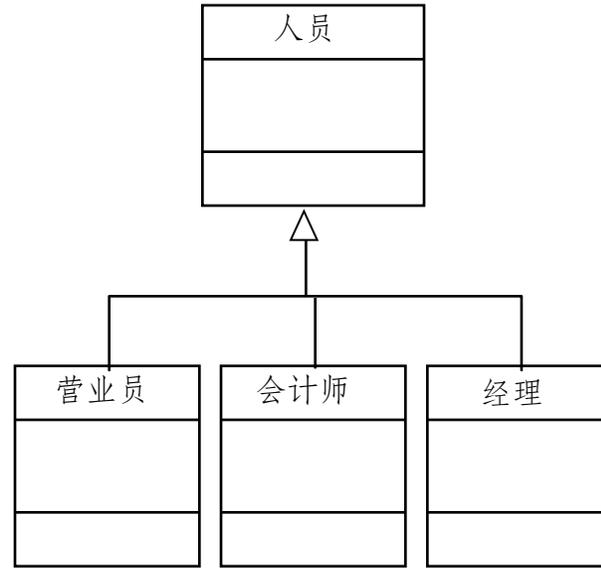


(4) 表示动态变化的对象特征

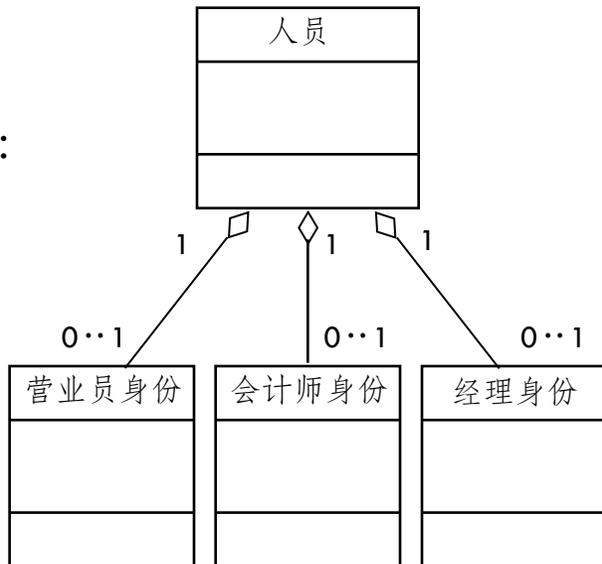
问题：对象的属性与操作定义在系统运行中动态变化，例如：

不理想的解决办法：

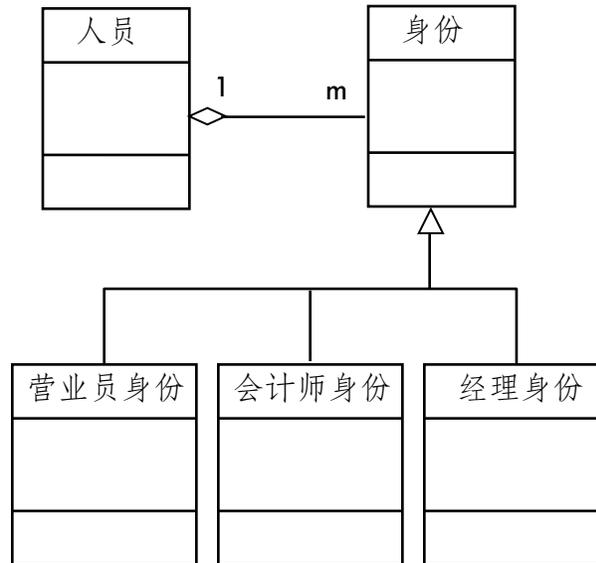
删除、重建
Shlaer/ Mellor的子类型迁移
“动态对象”



解：



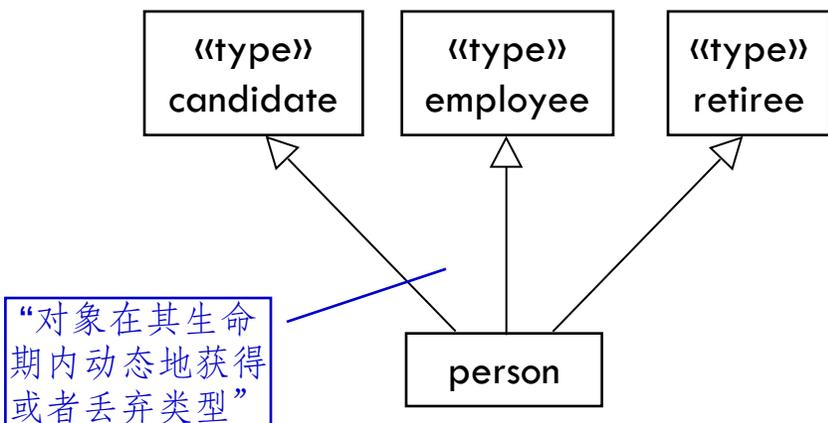
或



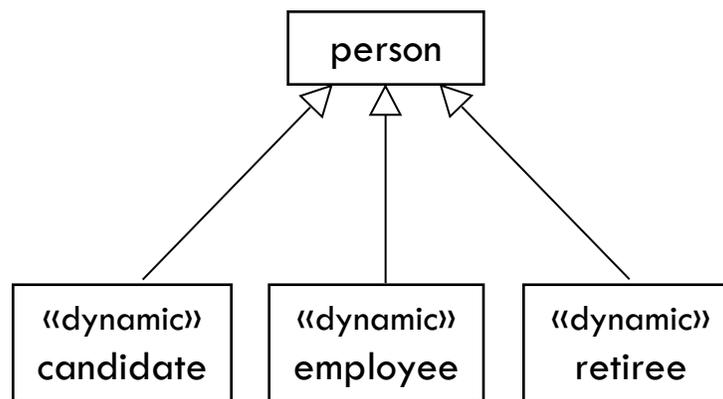
“三友”对问题的描述及解决方法

“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



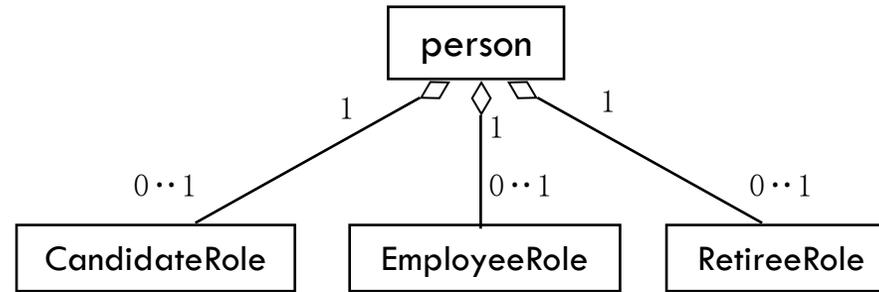
1999年第一版



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：

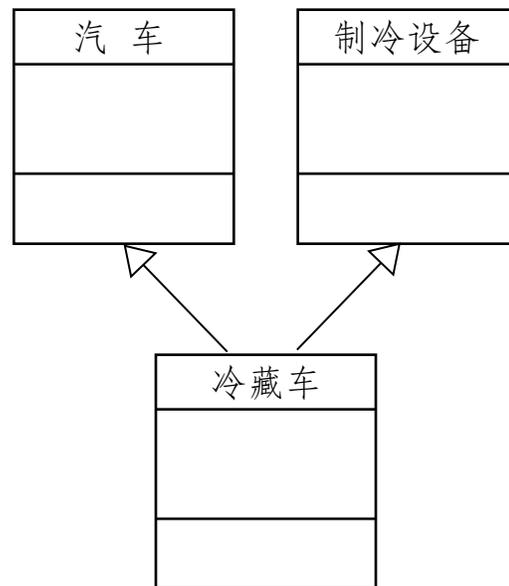
整体-部分结构有很强的表达能力

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

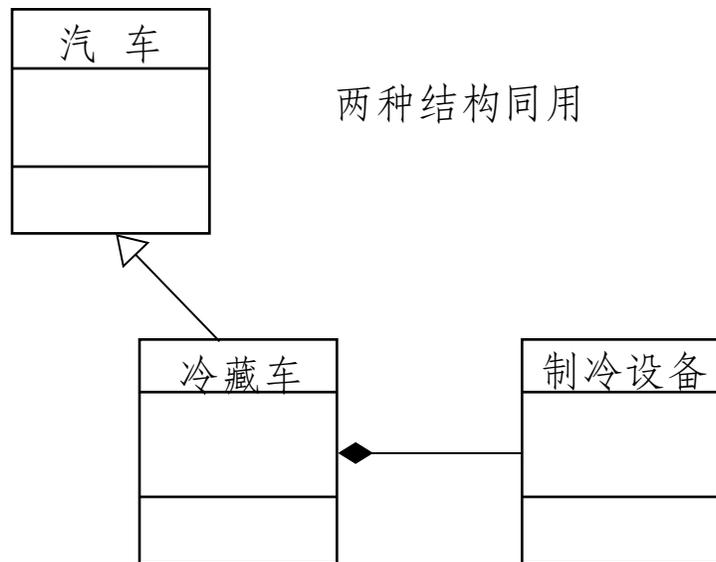
用一般—特殊结构



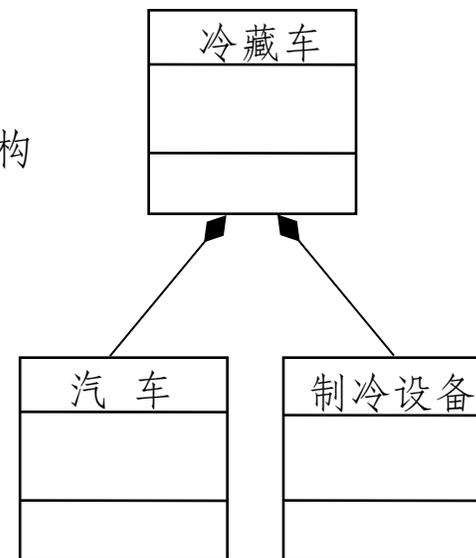
解释:

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



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： $\frac{1}{1}$

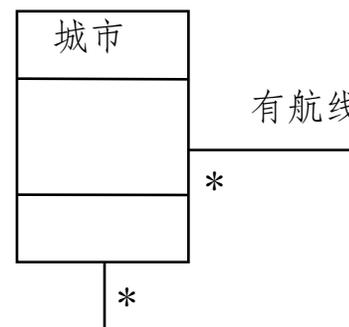
一对多： $\frac{1}{*}$

多对多： $\frac{*}{*}$

例子



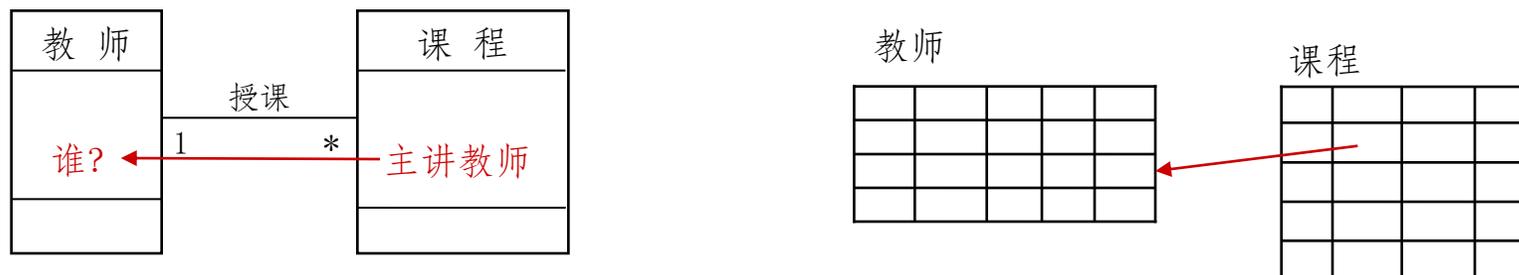
教师为学生指导论文



(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）。



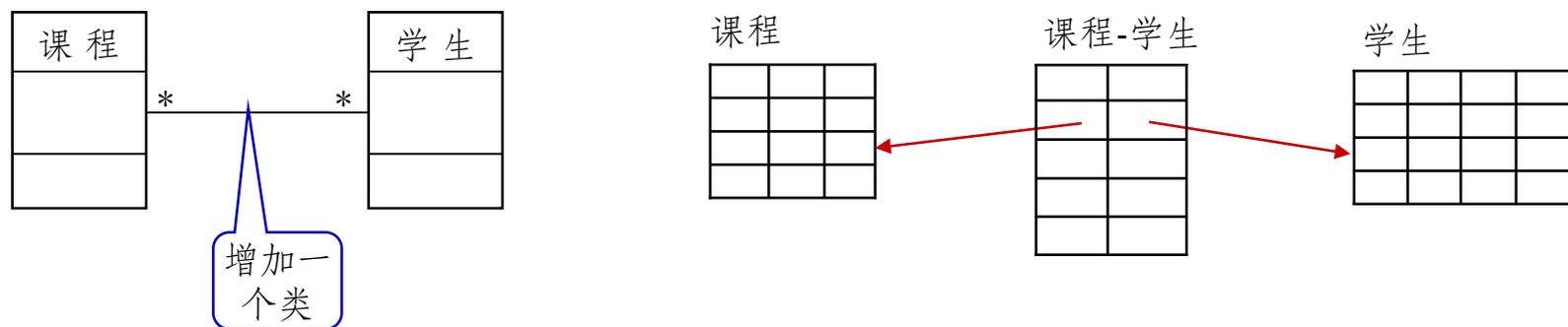
关系数据库：用两个数据库表分别实现关联两端的类；以数量约束为“1”的类对应的表的元组为目标，在关联另一端的类对应的表中设置一个指向该目标的外键（目标的主键）。

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。

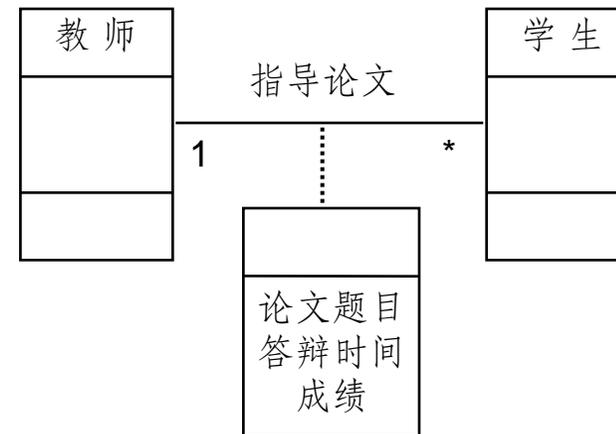
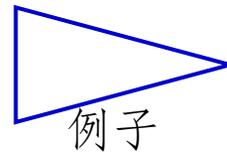
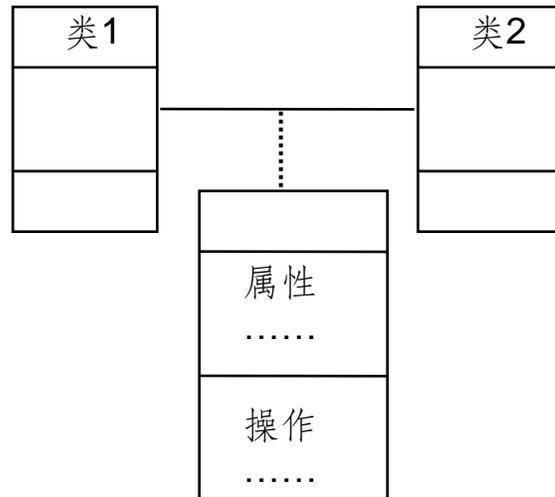
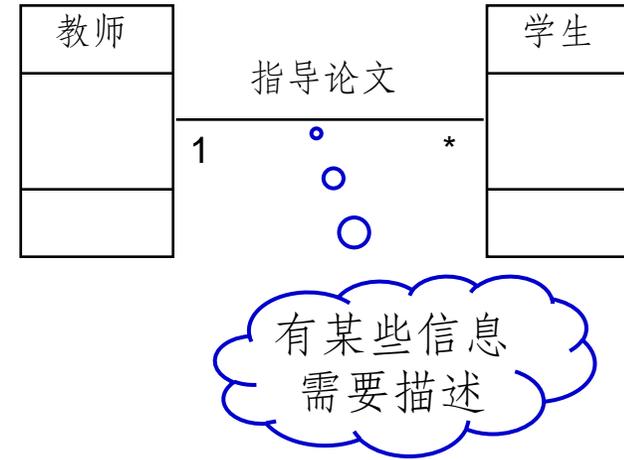
关系数据库：用两个数据库表分别实现关联两端的类，同时用另外一个数据库表来实现它们之间的关联。实现关联的数据库表含有两个属性，它们分别是指向两端的表的元组的外键。



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)



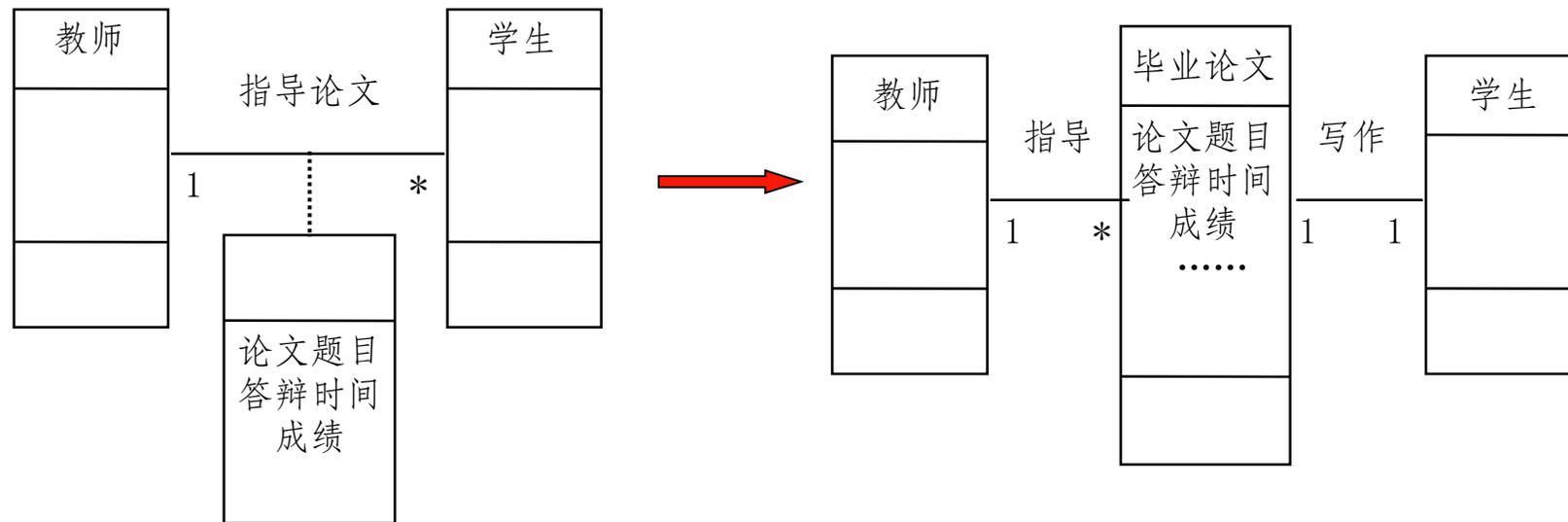
问题：增加了概念的复杂性，缺乏编程语言支持

换一种思路考虑问题：

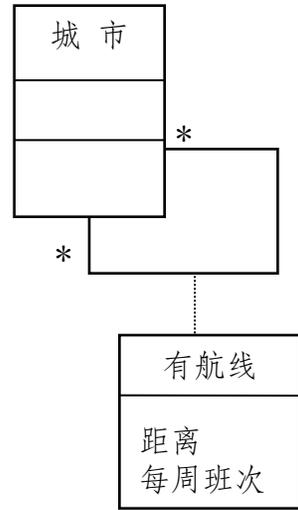
两类对象之间的关联带有某些复杂的信息，说明它们之间存在着某种事物（尽管可能是抽象事物）。

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。

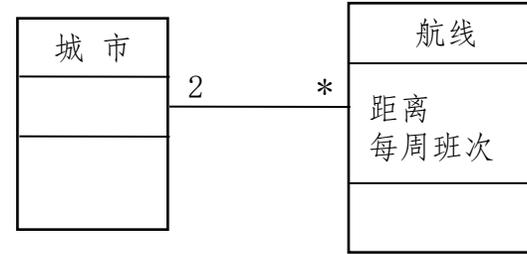
例1



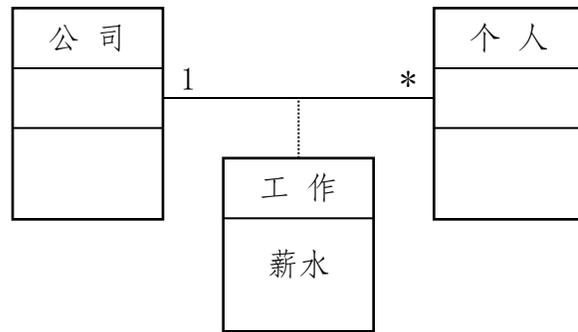
其他例子



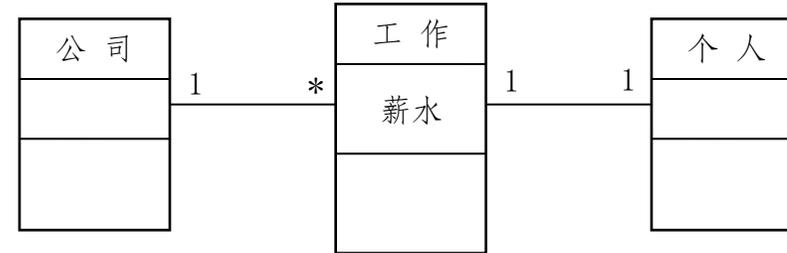
城市之间有航线



城市之间存在航线对象

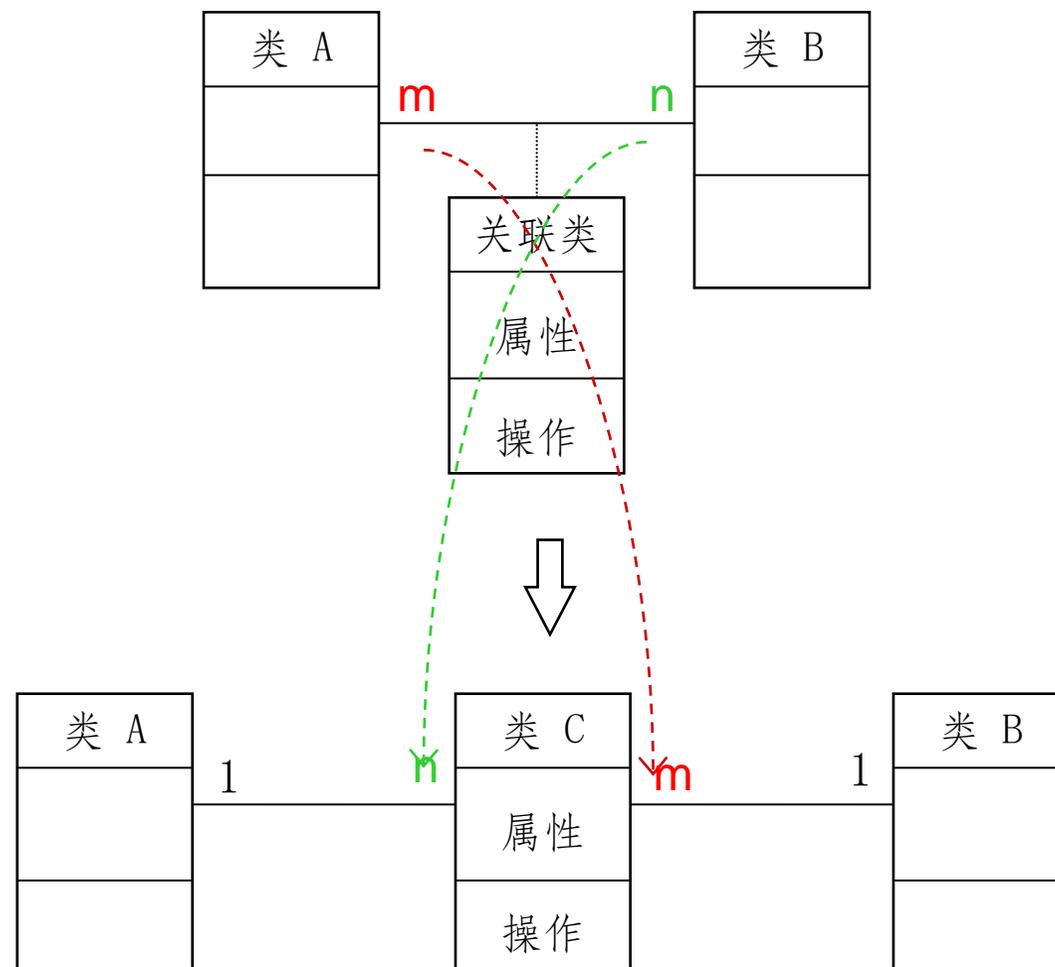


公司与个人



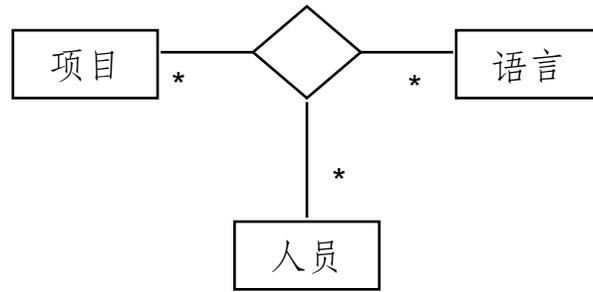
公司与个人之间存在工作对象

复杂关联表示法的转换

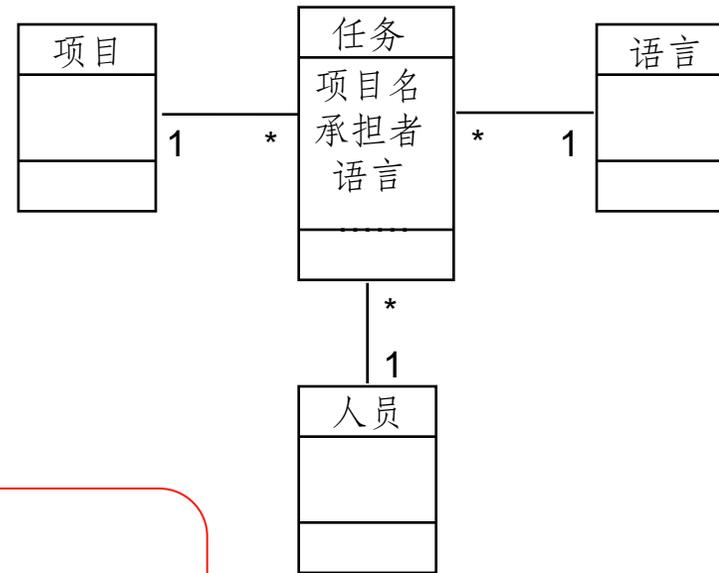


(2) n元关联

OMT的三元关联及其表示法



增设对象类表示多元关联



问题:

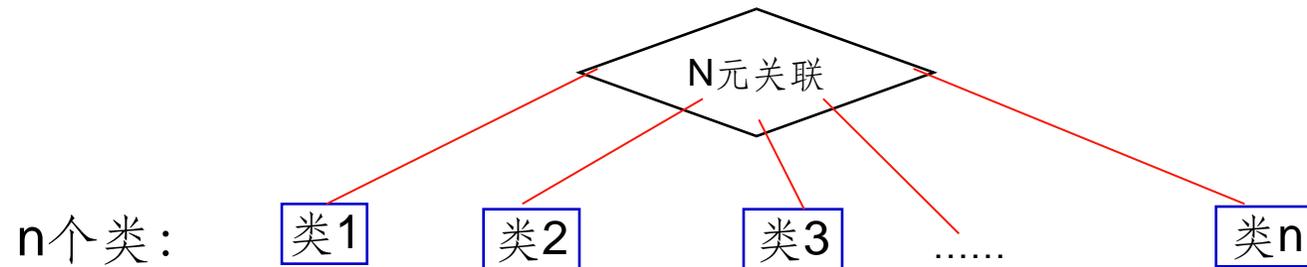
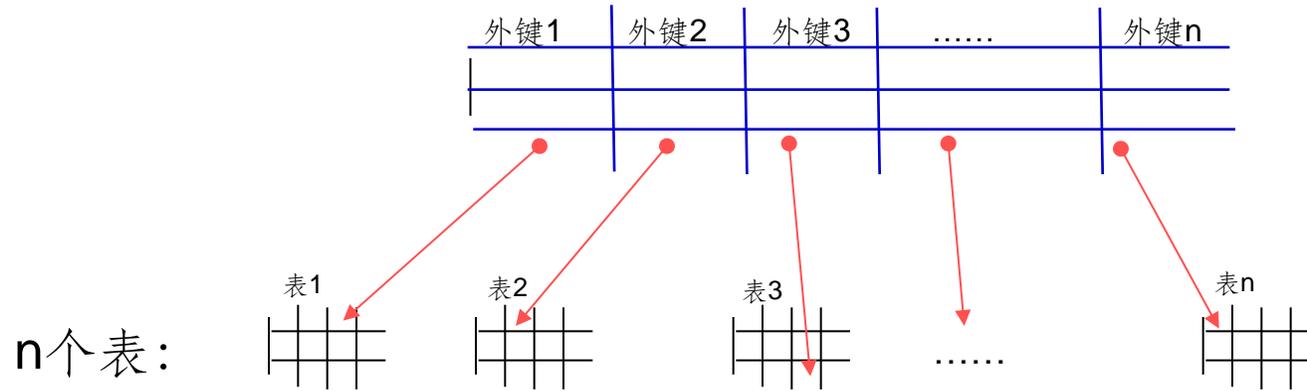
编程语言不能直接支持

可推广到n元关联, 是否要创造更多的符号?

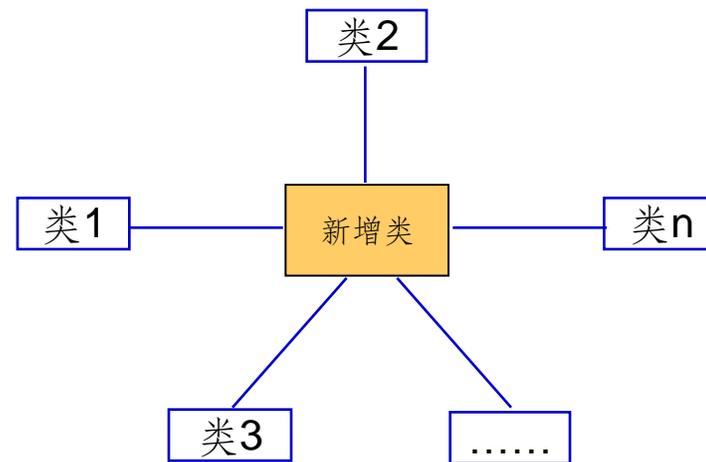
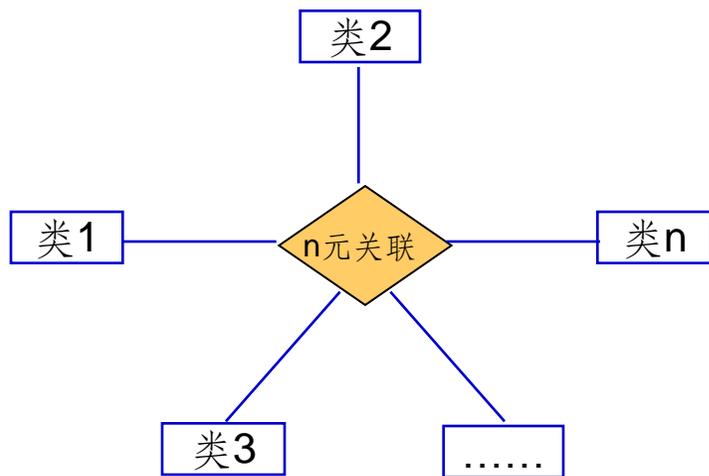
多重性表示的困难 (详后)

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

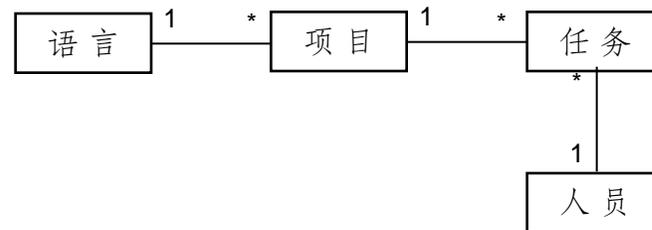
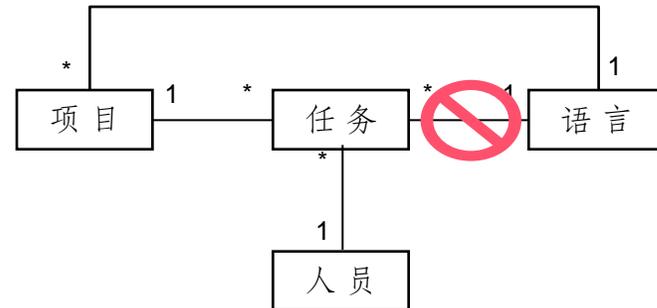
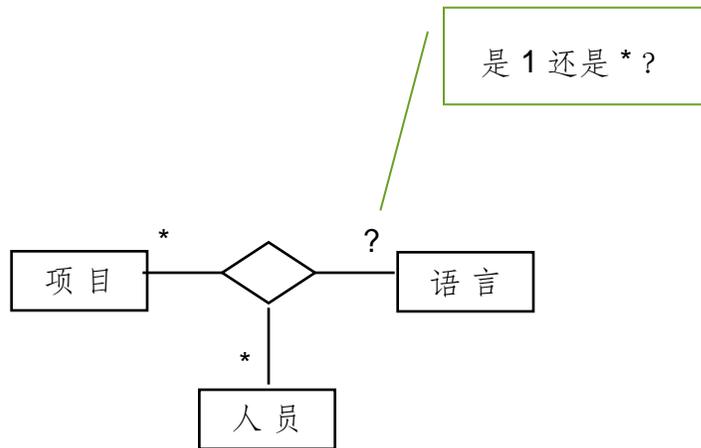
从实现的角度看，用类实现 n 元关联是最自然的选择
例如：用一个数据库表存放 n 元关联的全部 n 元组



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

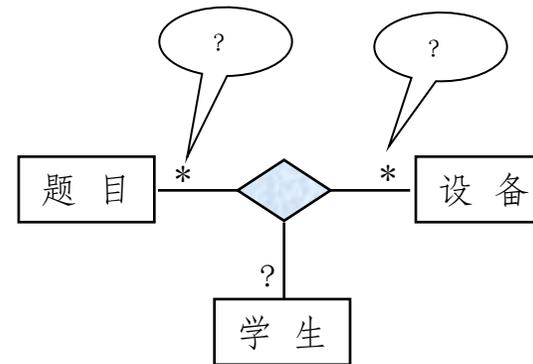
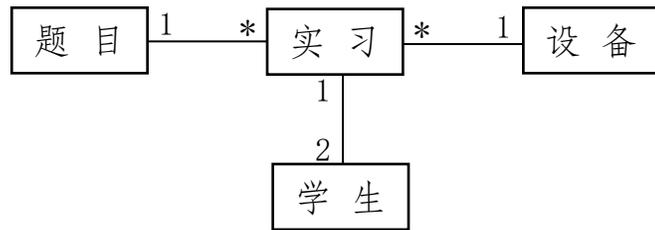
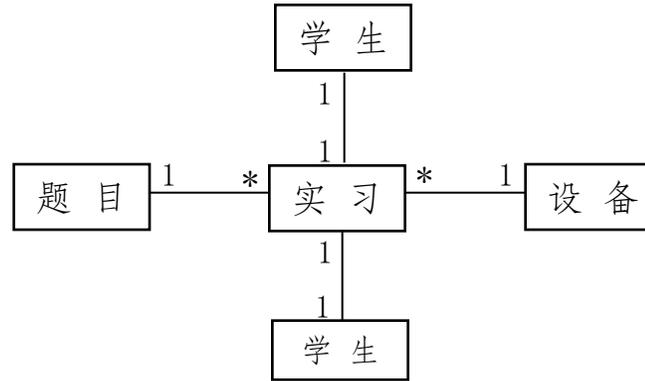
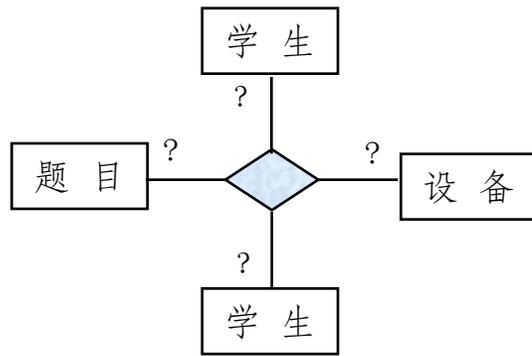
- 1) 若系统要求记录和查阅哪两名学生是合作者
建立学生类到它自身的关联（如同城市之间有航线）
是一个二元关联，其中学生类在关联中出现了两次
- 2) 如果还要记录每组学生的实习题目和使用的设备
建立学生、题目、设备三个类之间的4元关联
学生类在这个关联中出现了两次

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。



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

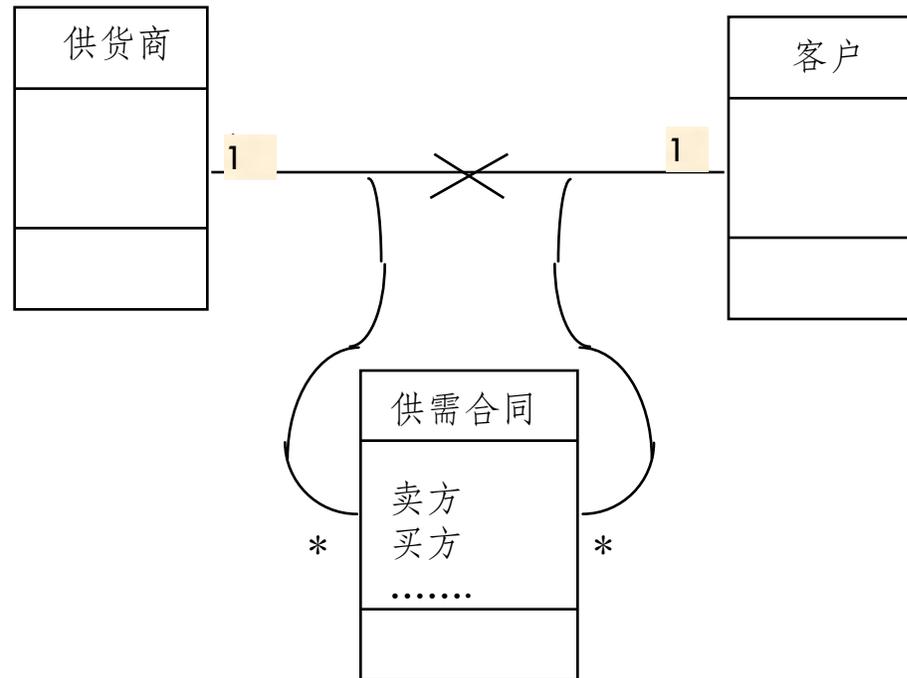
哪些类的对象实例之间存在着对用户业务有意义的关系？

- 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



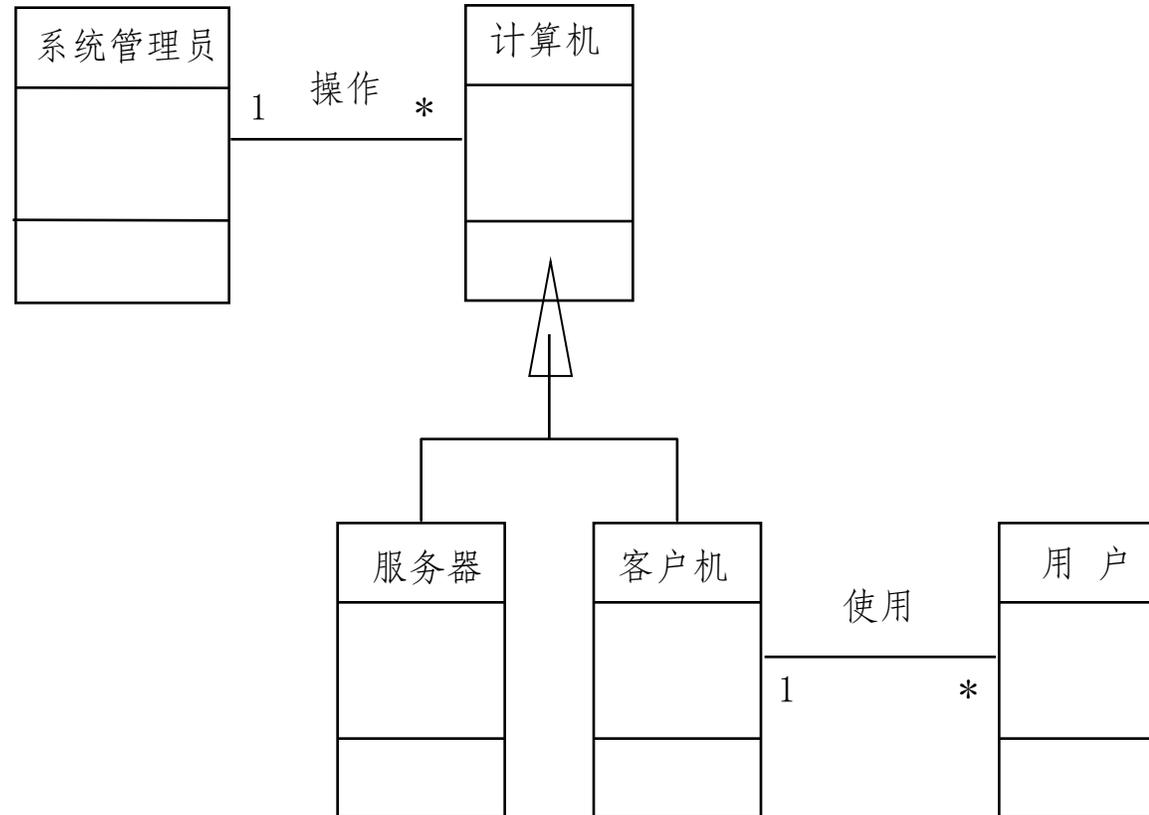
3. 为关联端点添加修饰

- 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——人或其他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

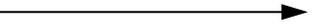
局部与全局问题

实际上类图中仍然包含动态信息

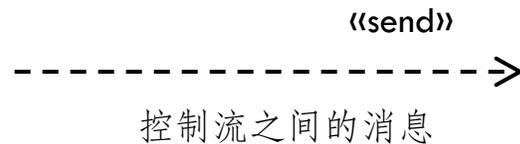
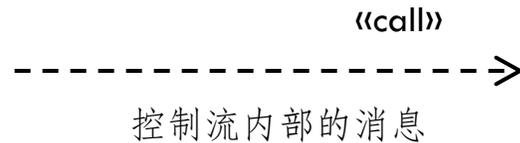
操作，调用 (**call**) 依赖

用什么符号表示消息

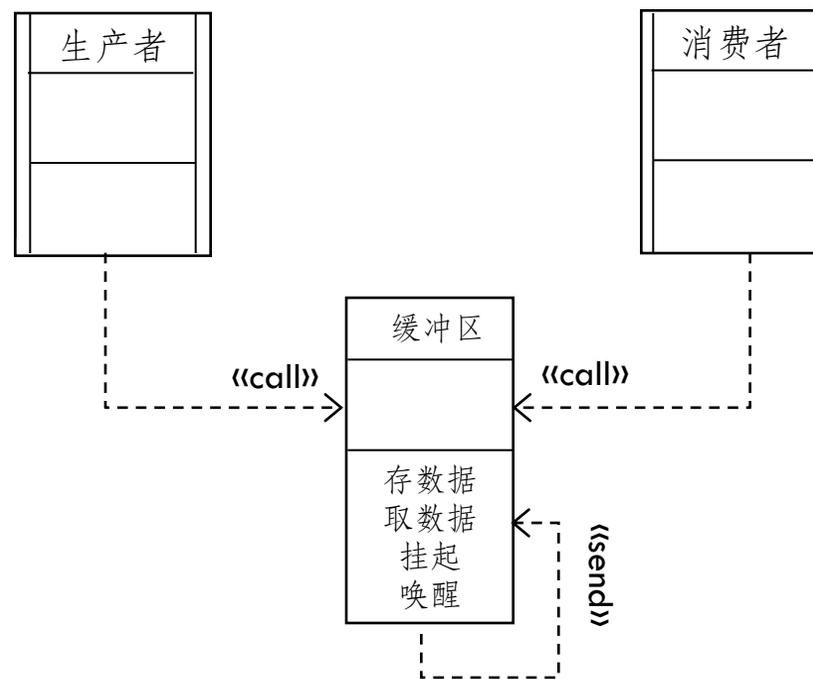
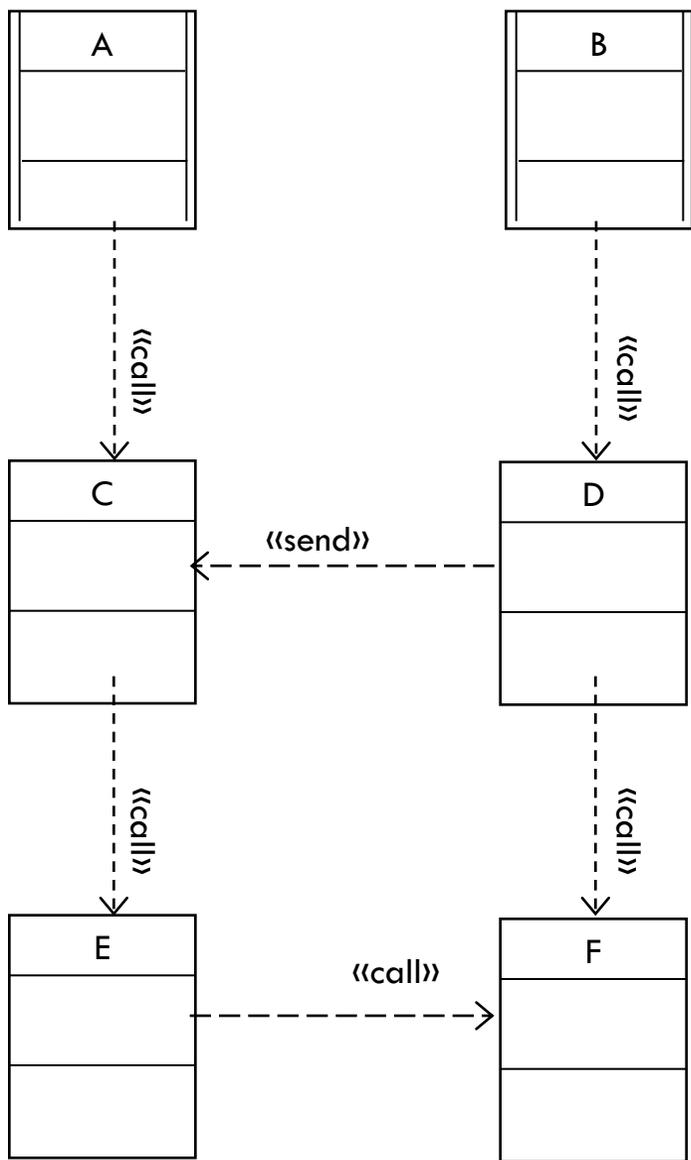
UML对各种箭头的用法

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

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
混杂了其他方法

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

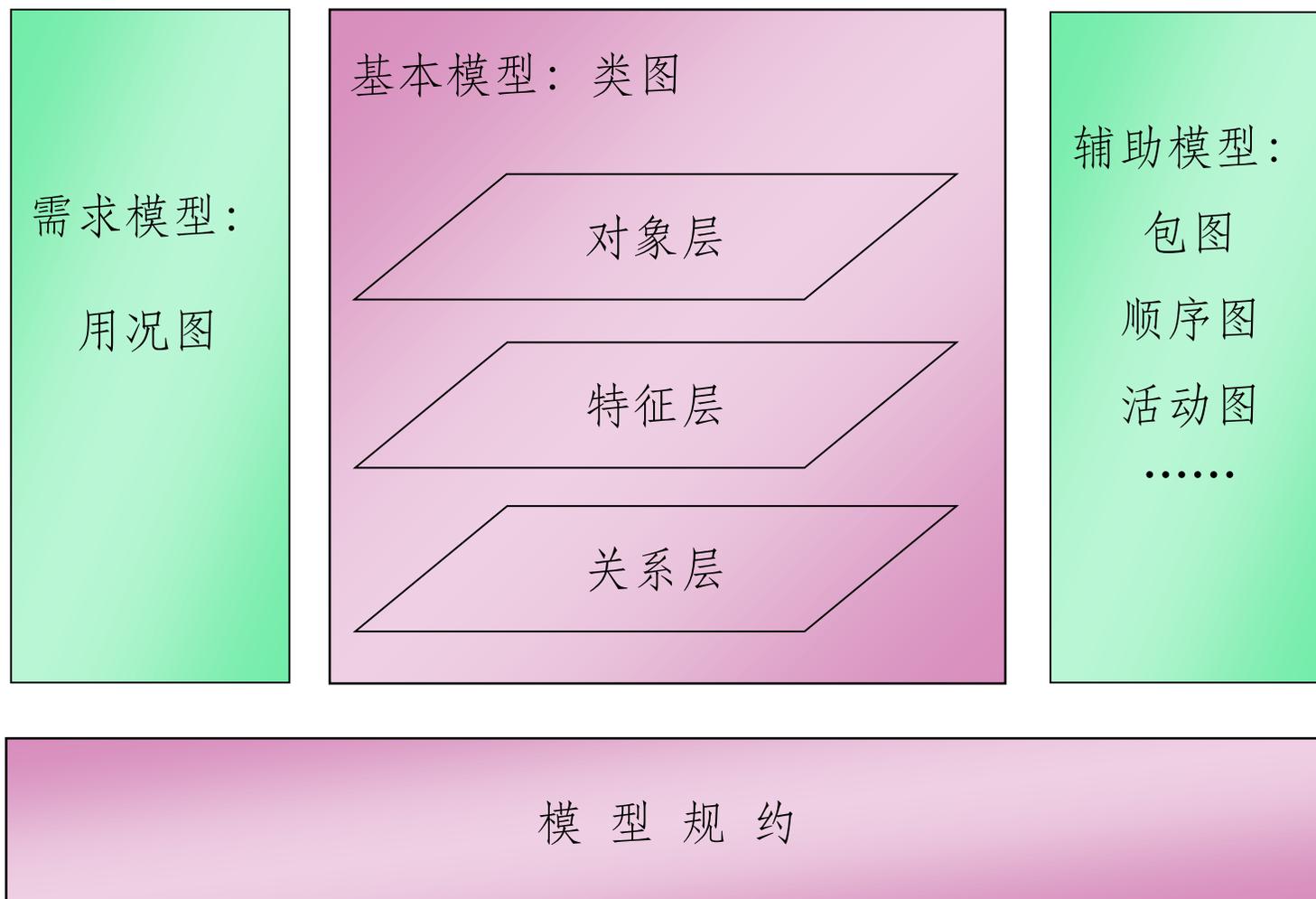
UML的状况和发展趋势

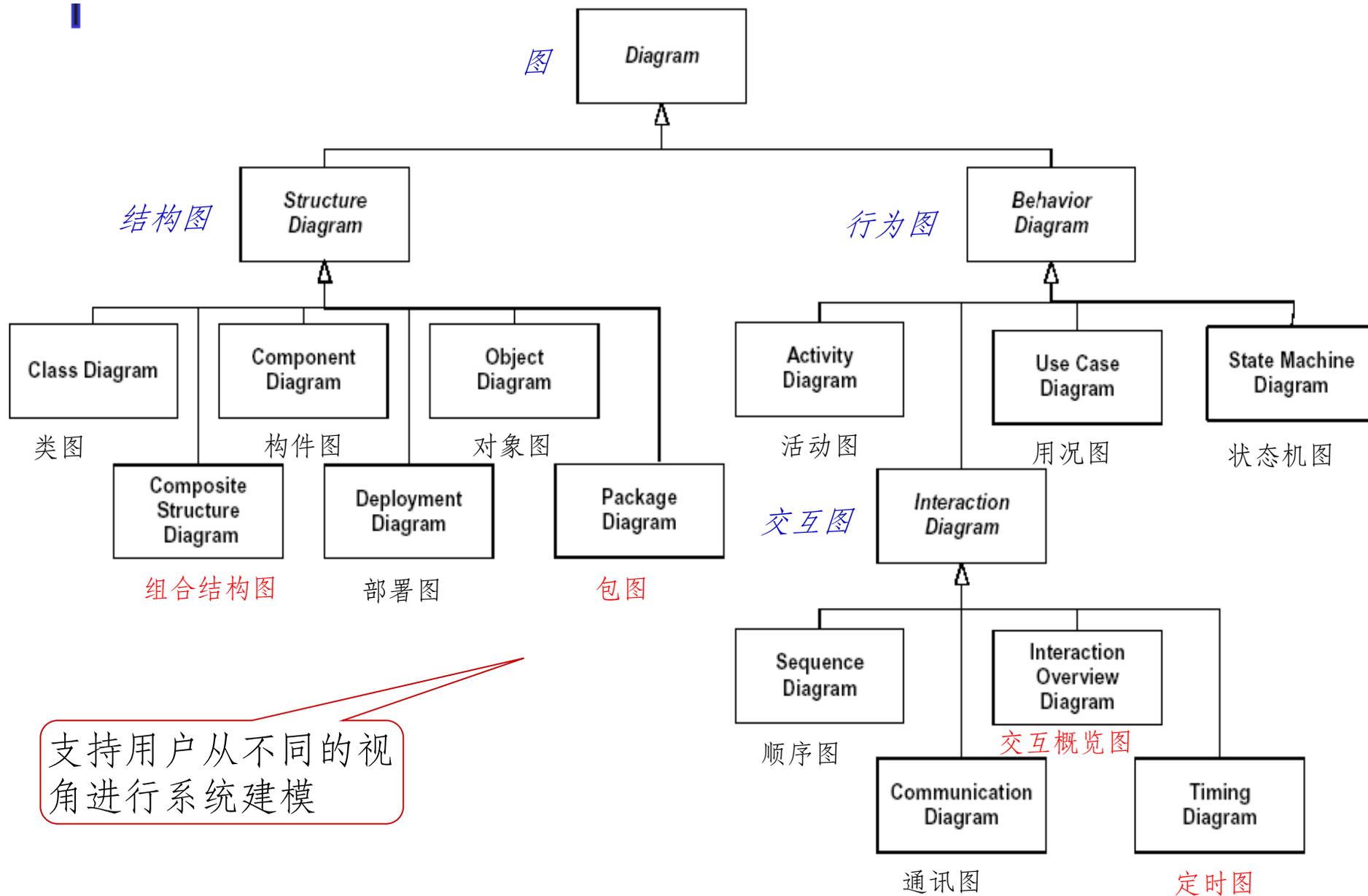
收集了大量的模型图，从9种发展到13种

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

在本书的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





支持用户从不同的视角进行系统建模

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。

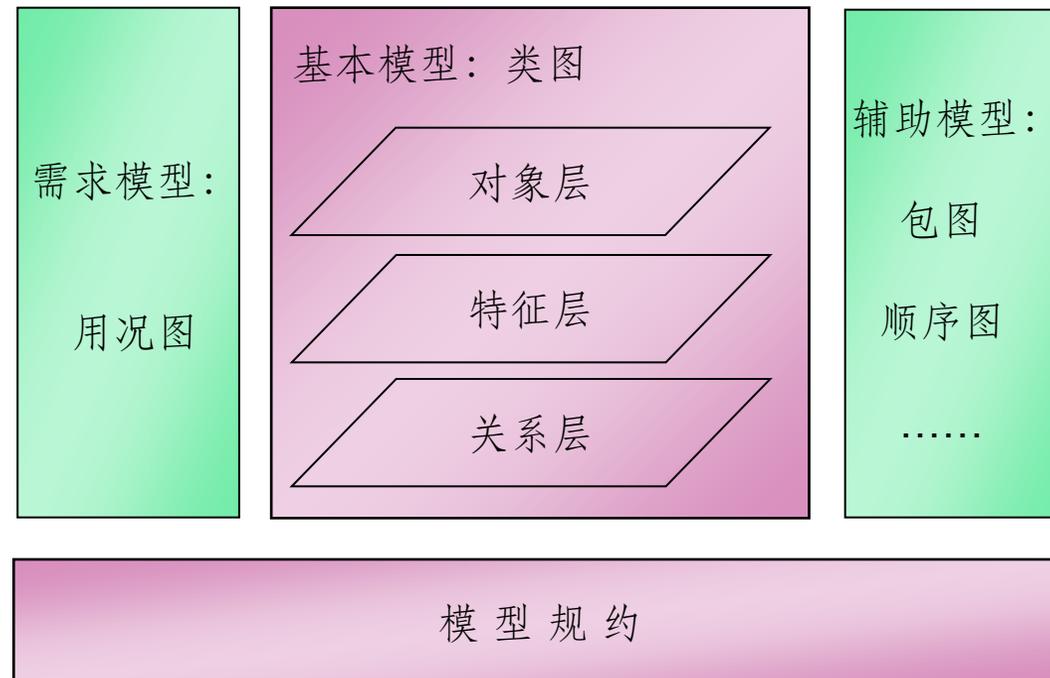
组合结构图、交互概览图、定时图：都可以作为辅助模型，无强烈建议。

对象图、通信图：建议不使用。

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

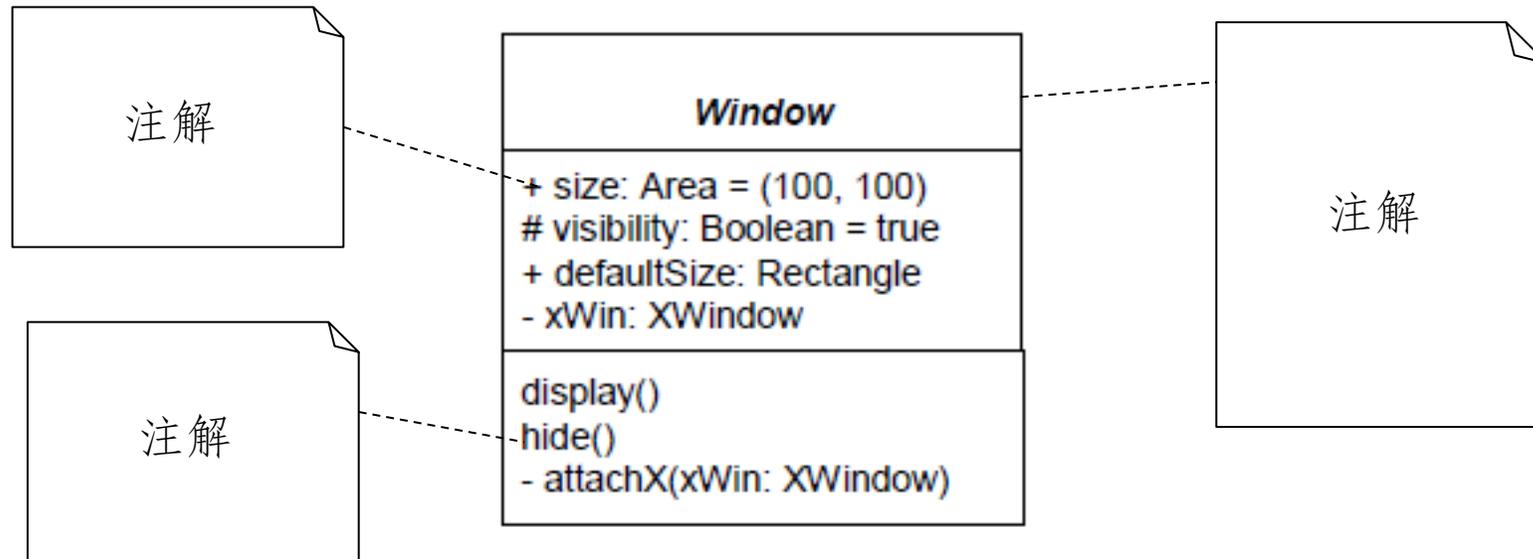
规约的组织

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

对一个面向对象的系统模型建立规约，关键问题是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地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。