研究生课程

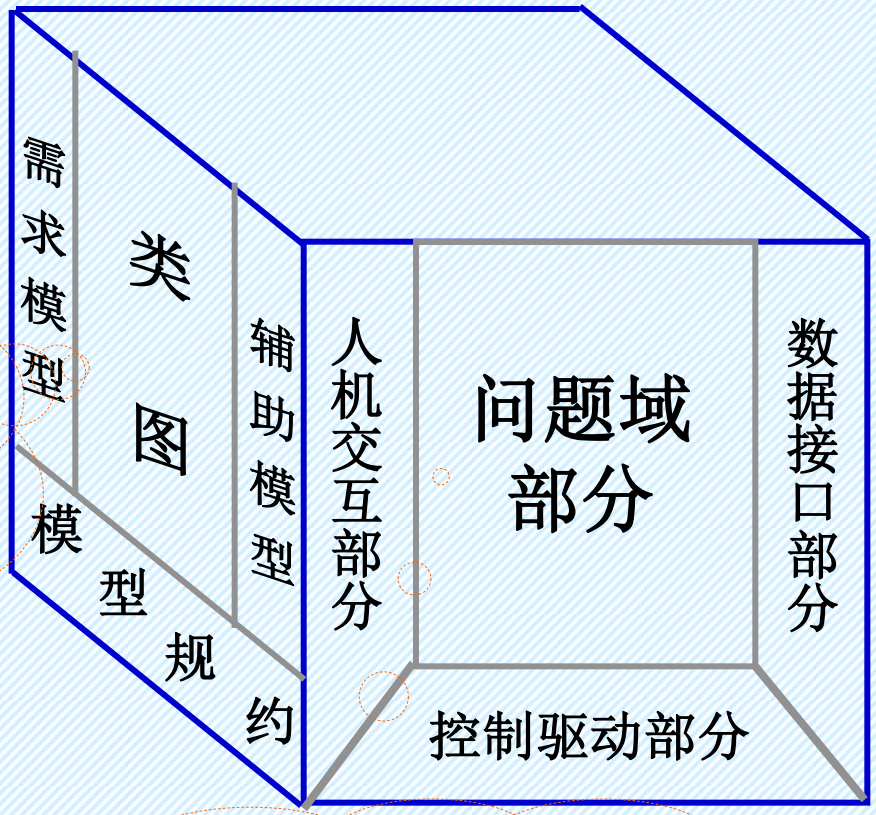
软件工程方法论

第三部分：设计篇

回顾

OOD模型框架

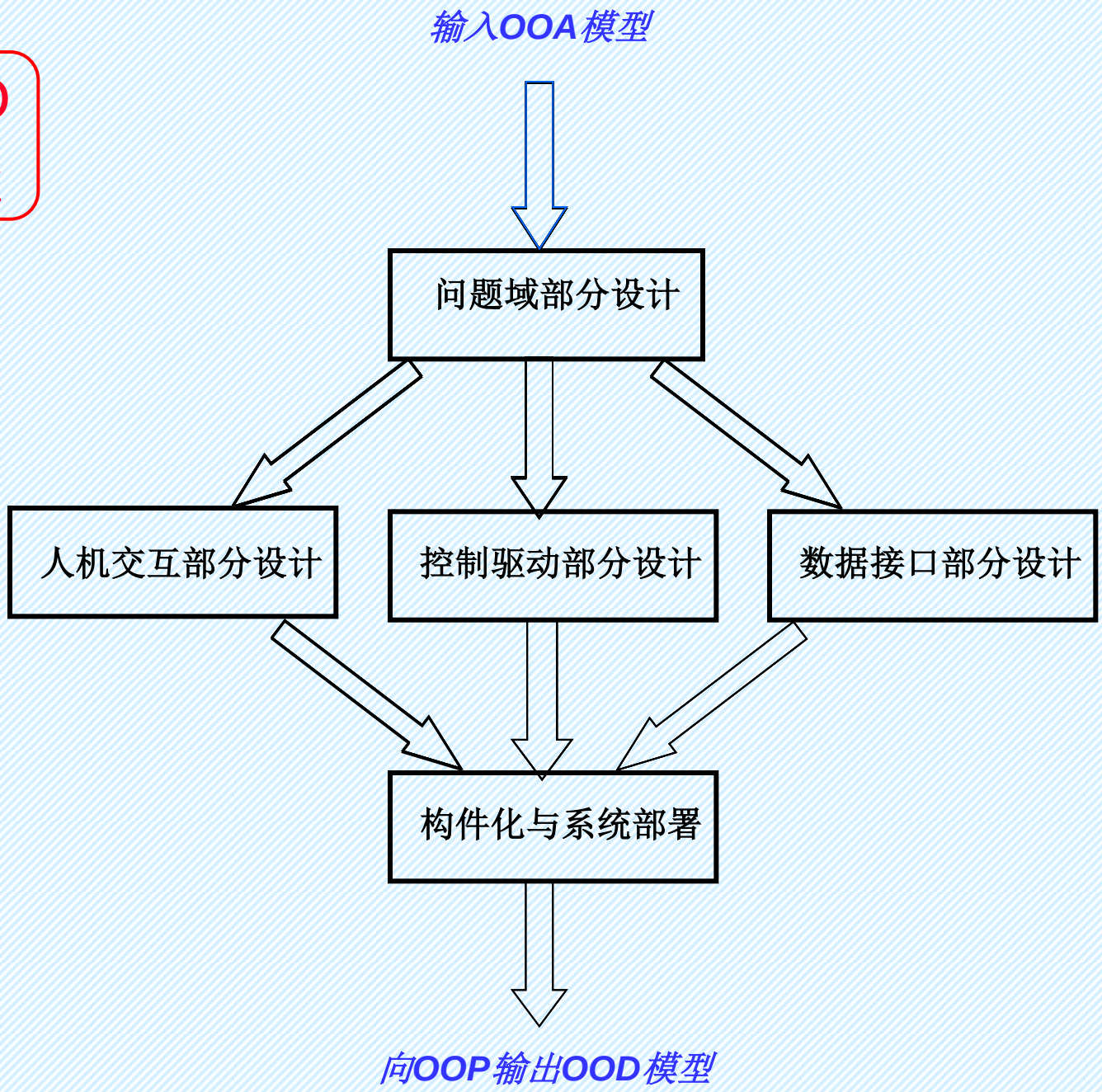
——从两个侧面来描述



从另一侧面看：
OOD模型每个部分
如何用OO概念表达？
采用与OOA相同的概念及
模型组织方式

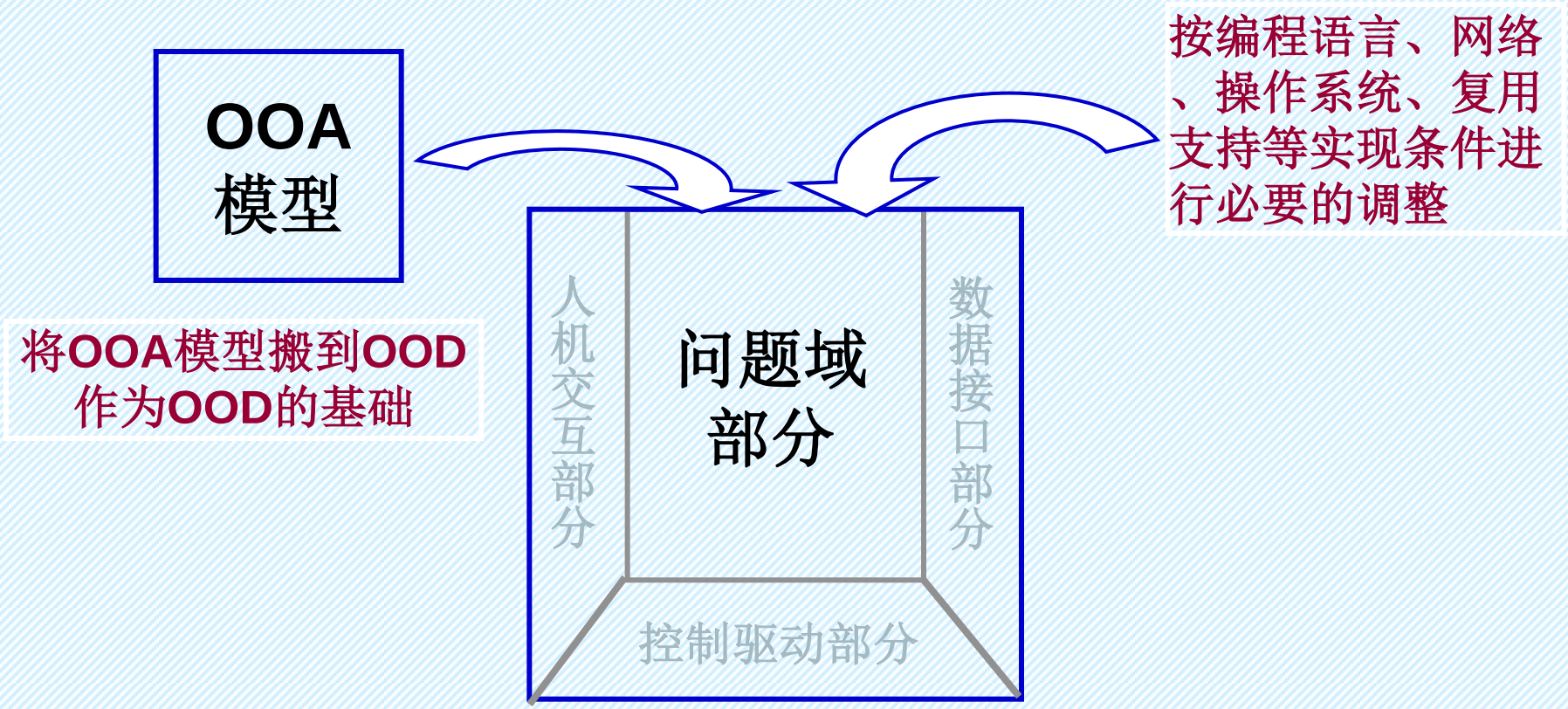
从一个侧面看：
OOD模型包括几个主要部分？
一个核心加三个外围

OOD过程

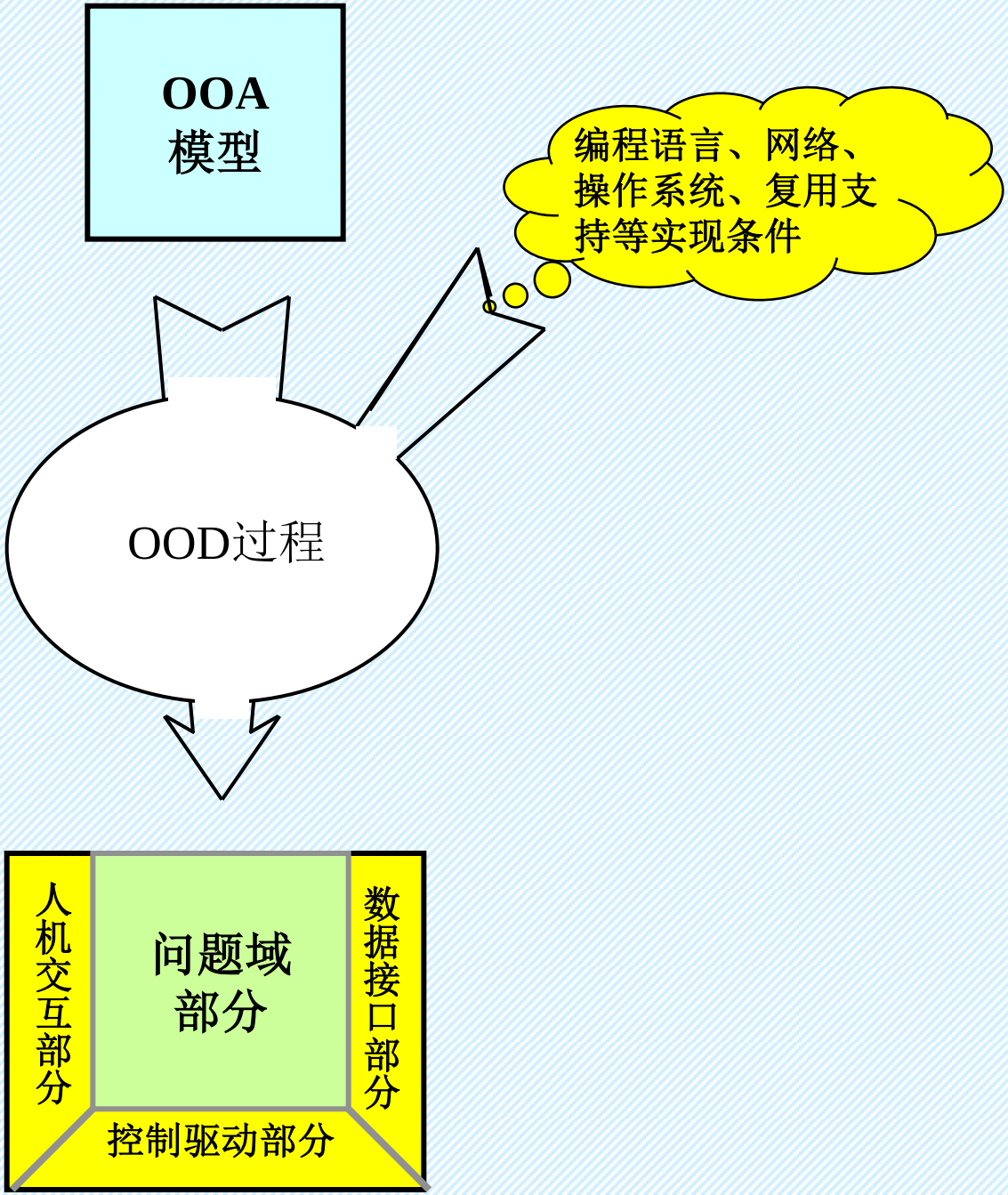

输入OOA模型
问题域部分设计
人机交互部分设计
控制驱动部分设计
数据接口部分设计
构件化与系统部署
向OOP输出OOD模型

11.1 什么是问题域部分

问题域部分是OOD模型的四个组成部分之一，由来自问题域的对象构成，是在OOA模型基础上，按照具体的实现条件进行必要的修改、调整和细节补充而得到的。



从MDA的观点看问题域部分的产生



11.2 实现条件对问题域部分的影响

编程语言

语言的实现能力

硬件、操作系统及网络设施

对象分布、并发、通信、性能

复用支持

根据复用支持对模型做适当调整，以实现复用

数据管理系统

为实现对象的持久存储，对问题域部分做某些修改

界面支持系统

问题域部分与人机界面之间的消息传输

11.3 设计过程

设计准备

保留OOA文档

复制OOA文档，作为OOD的输入

根据需求的变化和发现的错误进行修改

设计内容与策略（本节的重点）

针对编程语言支持能力的调整

增加一般类以建立共同协议

实现复用

提高性能

为实现对象持久存储所做的修改

完善对象的细节

定义对象实例

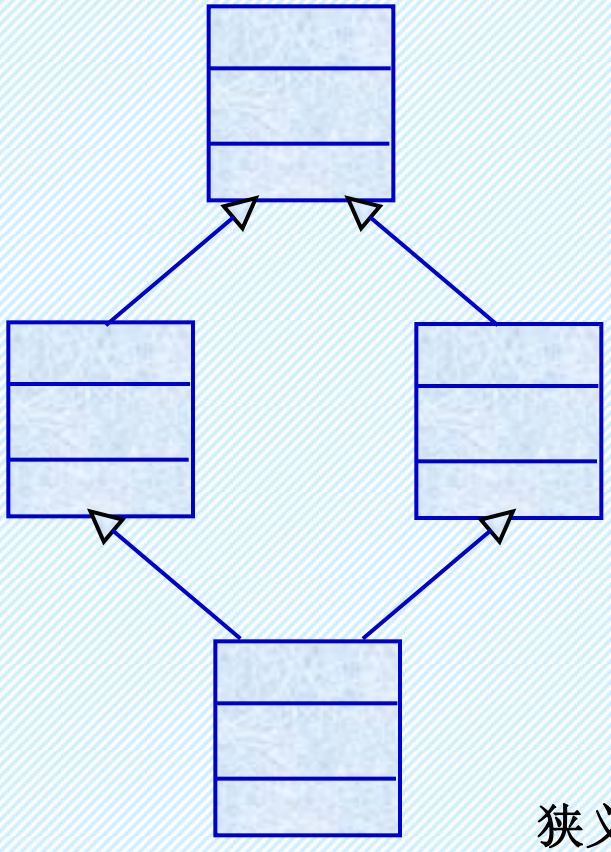
对辅助模型、模型规约的修改和补充

建立OOD文档与OOA文档的映射

1、按编程语言调整继承与多态

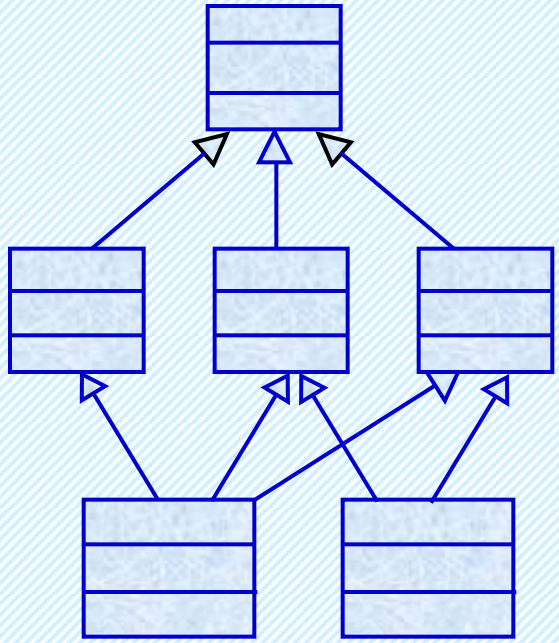
起因：OOA强调如实地反映问题域，OOD考虑实现问题，
如果语言不支持多继承或多态，就要进行对模型调整

（1）多继承化为单继承



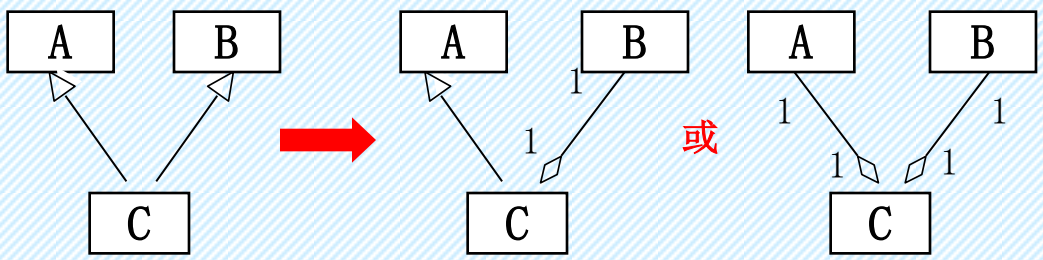
狭义菱形

多继承模式

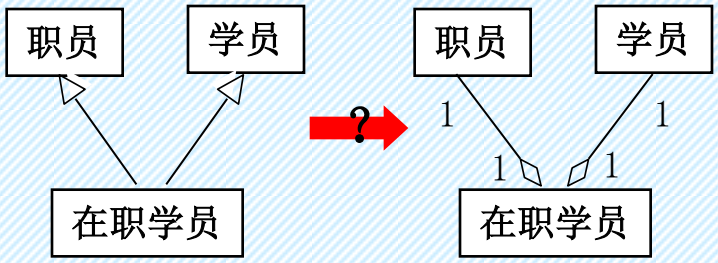


广义菱形

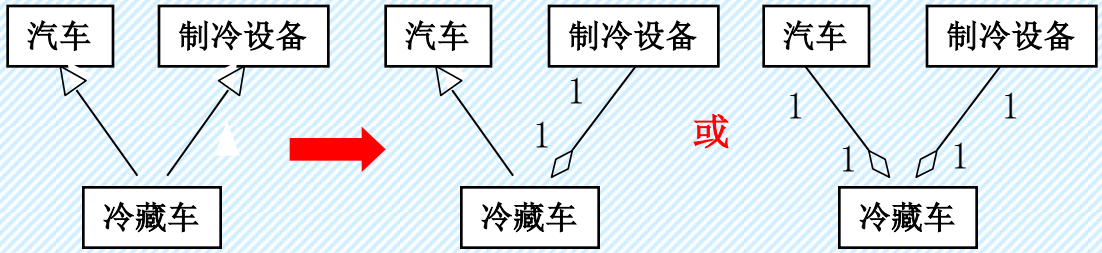
方法1：简单转换



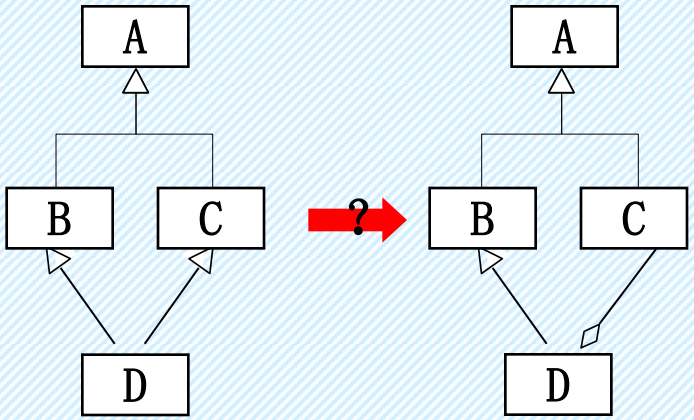
（a） 一般方法



（c） 不合适的例子

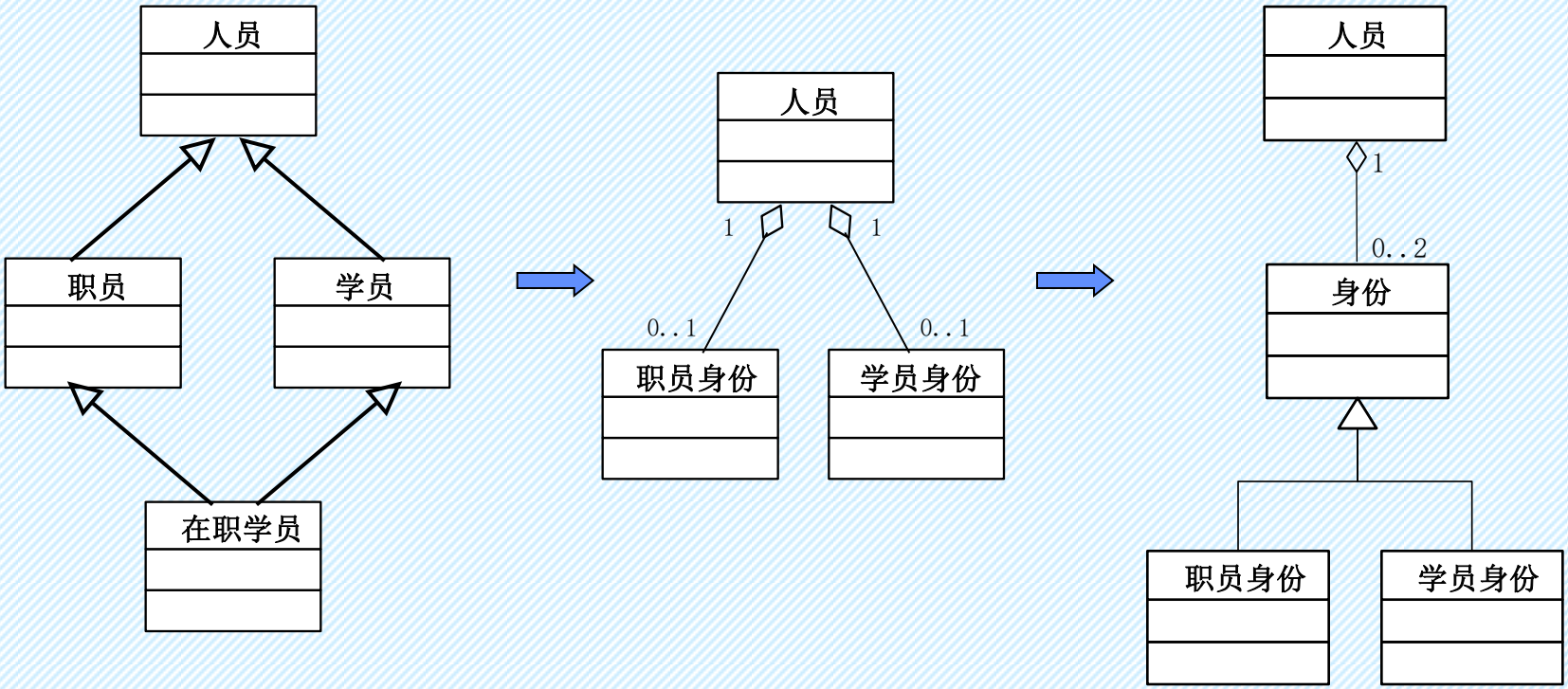


（b） 合适的例子

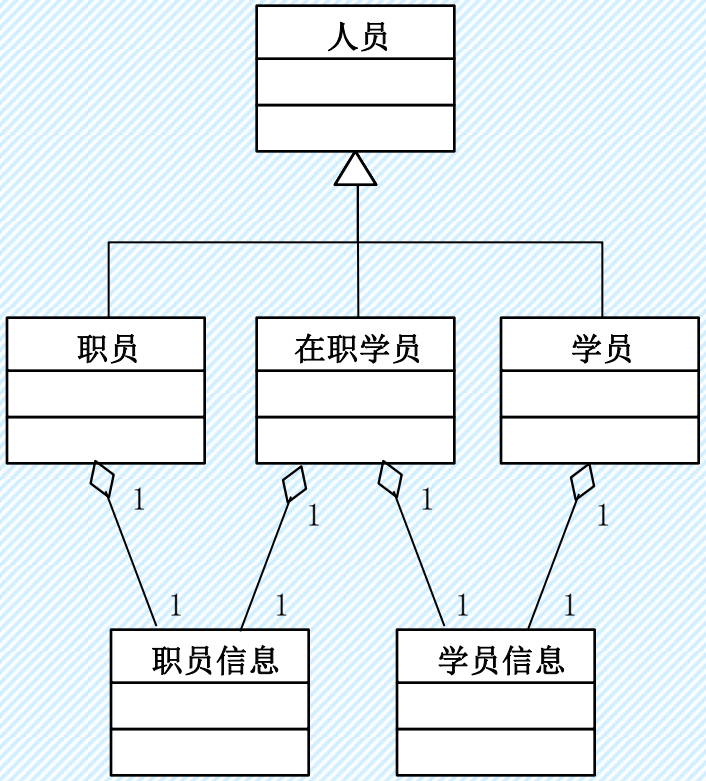


（d） 转换产生信息重复

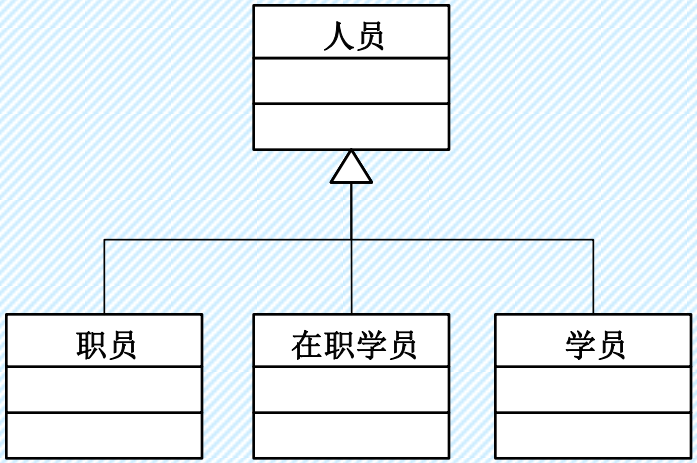
方法2：重新定义对象类，化解多继承



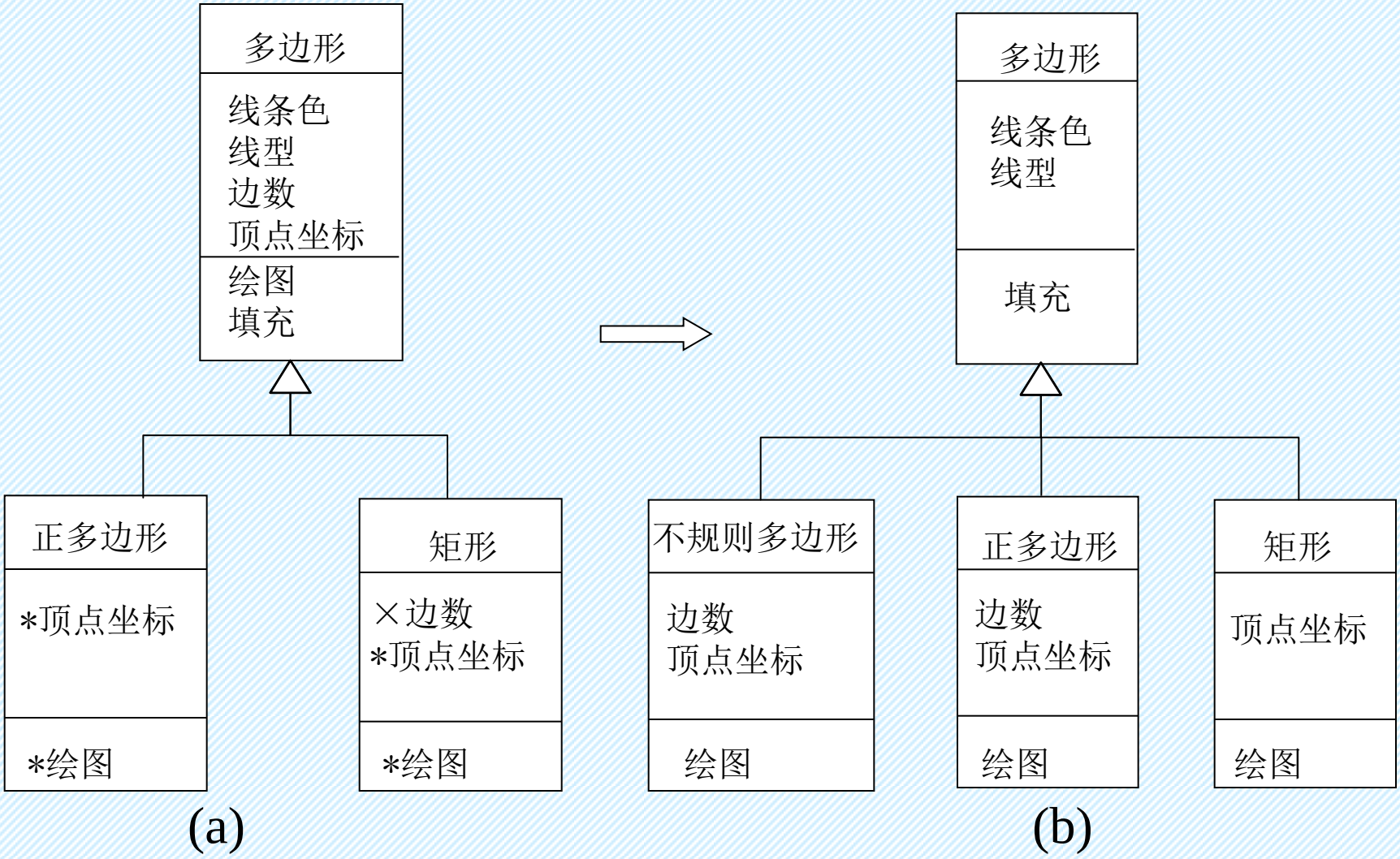
方法3：保持分类，剥离多继承信息



不适当的方法增加程序代码



（2）取消多态性


多边形
线条色
线型
边数
顶点坐标
绘图
填充
正多边形
*顶点坐标
*绘图
矩形
×边数
*顶点坐标
*绘图
(a)
多边形
线条色
线型
填充
不规则多边形
边数
顶点坐标
绘图
正多边形
边数
顶点坐标
绘图
矩形
顶点坐标
绘图
(b)

2、增加一般类以建立共同协议

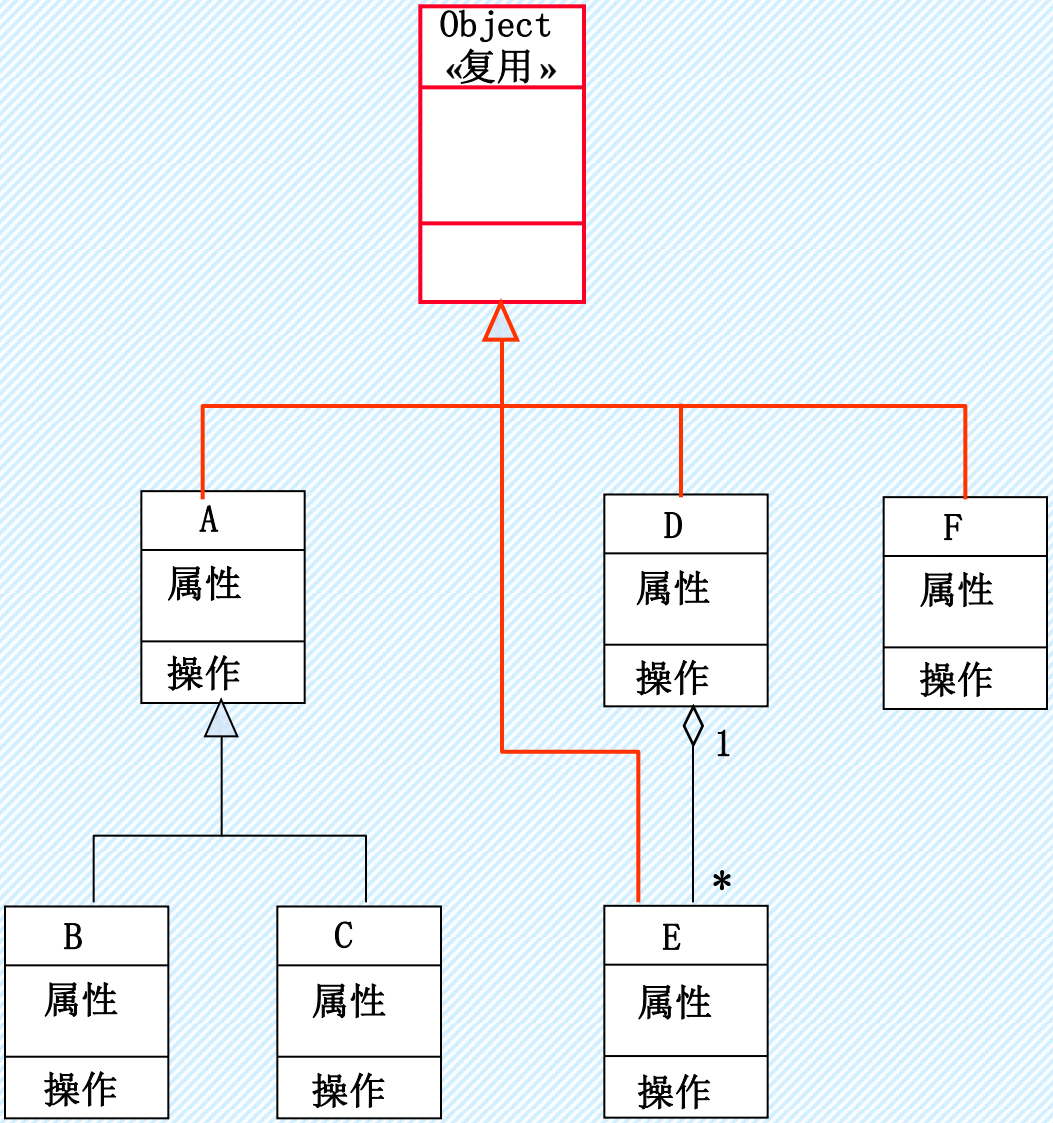
增加根类：将所有的类组织在一起
提供全系统通用的协议

例：提供创建、删除、复制等操作

增加其他一般类：提供局部通用的协议

例：提供持久存储及恢复功能

例：



3、实现复用的设计策略

如果已存在一些可复用的类，而且这些类既有分析、设计时的定义，又有源程序，那么，复用这些类即可提高开发效率与质量。

目标：尽可能使复用成分增多，新开发的成分减少

可复用类定义的信息 比 当前所需的类的信息

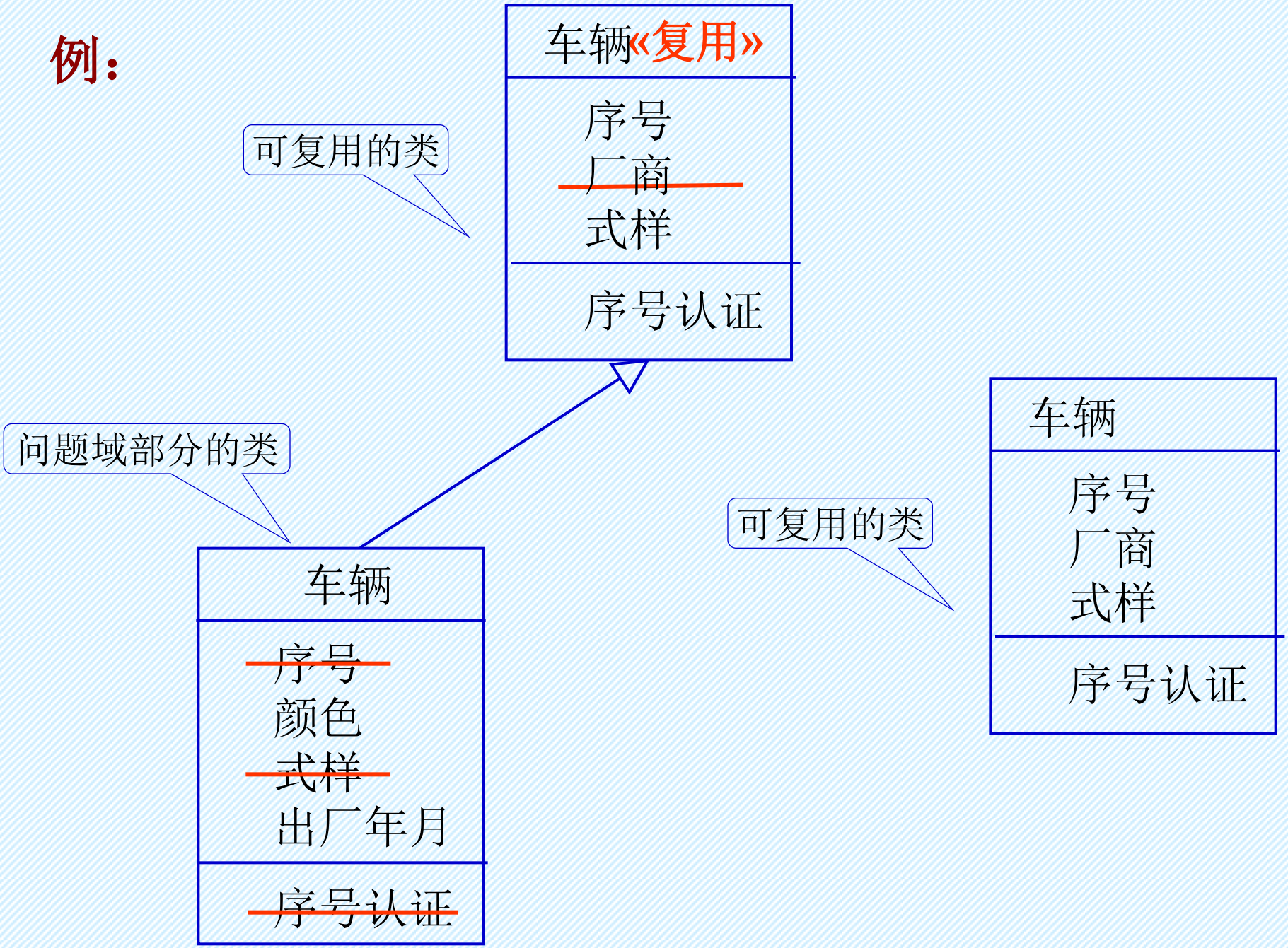
= 直接复用

< 通过继承复用

> 删除可复用类的多余信息

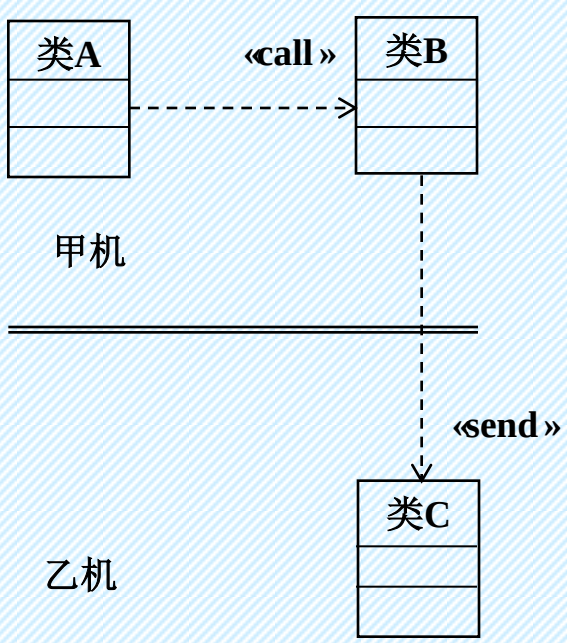
≈ 删除多余信息，通过继承而复用

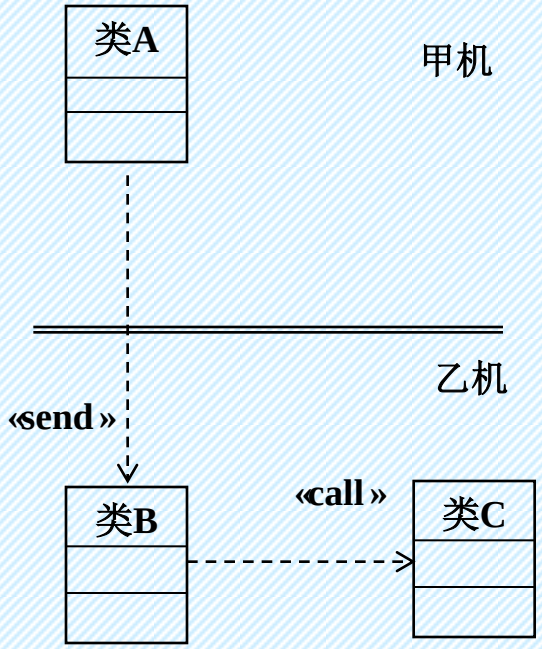
例：


车辆«复用»
序号
厂商
式样
序号认证
可复用的类
问题域部分的类
车辆
序号
颜色
式样
出厂年月
序号认证
可复用的类
车辆
序号
厂商
式样
序号认证

4、提高性能

（1）调整对象分布

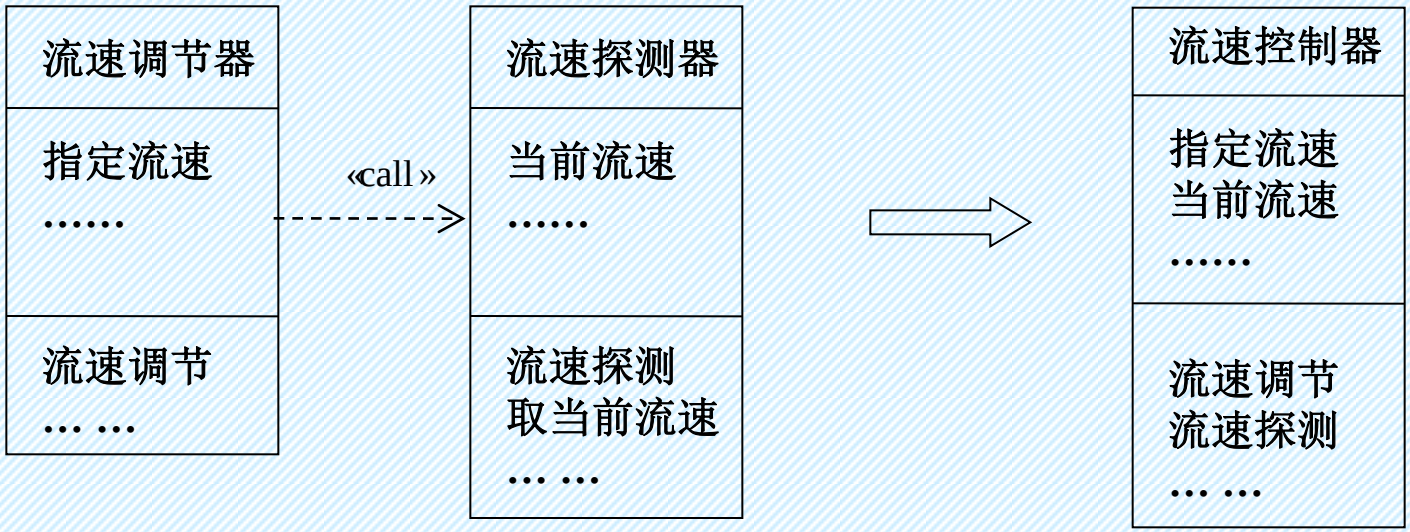




（2）缩短对象存取时间

设立缓冲区

（3）合并通讯频繁的类

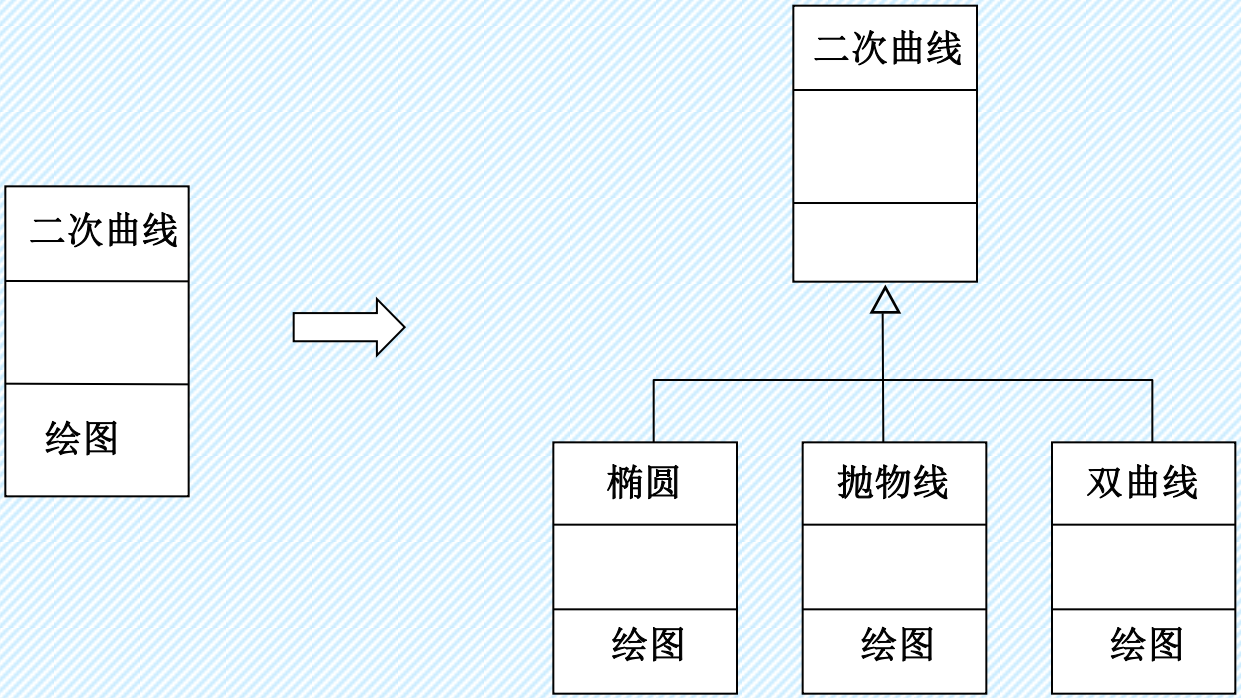


合并前　　合并后

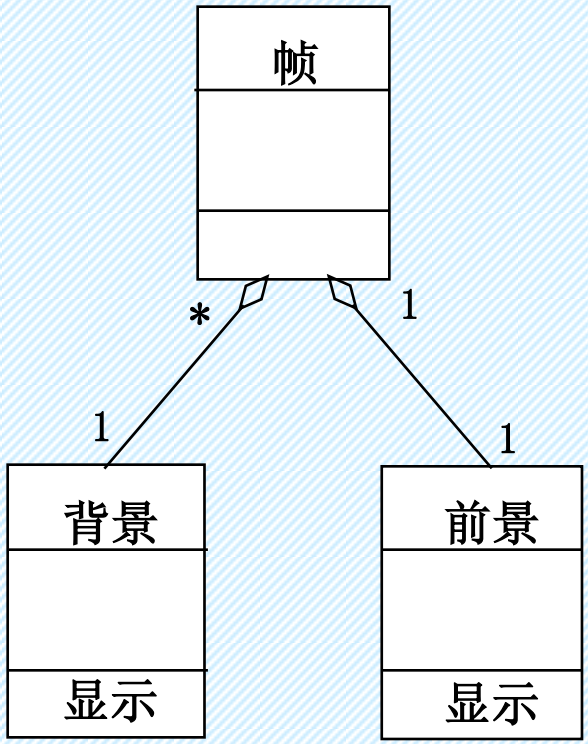
（4）增加属性以减少重复计算

（5）降低算法的计算复杂性

（6）细化对象的分类



（7）将复杂对象化为整体-部分结构



5、为数据存储管理增补属性与操作

在数据接口部分设计中介绍

6、完善对象的细节

OOD在OOA模型基础上所做的主要工作，不能用“细化”二字概括，但细化是不可缺少的

（1）完善与问题域有关的属性和操作

在OOA阶段允许不详尽，OOD必须加以完善

（2）解决OOA阶段推迟考虑的问题，包括：

因封装原则而设立的对象操作

与OOD模型其他部分有关的属性和操作

（3）设计类的每个操作

必要时用流程图或者活动图表示

（4）设计表示关联的属性

区分多重性的3种情况，决定属性设置在哪一端

（5）设计表示聚合的属性

区分组合与松散的聚合

对于组合，用嵌套对象实现

对于松散的聚合，采用与关联相同的策略

7、定义对象实例

在逻辑上，一个类的对象实例是：

问题域中所有可用这个类描述的实际事物

在物理上，一个类的对象实例可以是：

内存中的对象变量

文件的一个记录，或数据库表的一个元组

一个类的对象实例可以分布到不同的处理机上

对每一台处理机

说明在它之上创建的每一个（或组）内存对象

说明在它之上保存的外存对象

类的对象实例说明：

{

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>]}

}

8、修改或补充辅助模型及模型规约

包图

类的增减、拆分、合并以及各个类之间关系的变化

顺序图

操作与消息

活动图

操作流程

其他模型图

状态机图、定时图、交互概览图、组合结构图

模型规约

类的属性、操作及其对外关系的修改或细化

建立与OOA文档的映射

指出OOA模型中的哪个（或哪些）类演化为OOD模型中的哪个（或哪些）类

OOA 类与 OOD 类映射表

映射方式	OOA 类	OOD 类
1 = 1		
1 to 1		
1 to m		
m to 1		
m to m		
0 to 1		

图 3.13 OOA 类与 OOD 类的映射表

12.1 什么是人机交互部分

人机交互部分是OOD模型的外围组成部分之一，是系统中负责人机交互的部分。其中所包含的对象（称作界面对象）构成了系统的人机界面。

现今的系统大多采用图形方式的人机界面——形象、直观、易学、易用，远远胜于命令行方式的人机界面，是使软件系统赢得广大用户的关键因素之一。

但开发工作量大，成本高。近30年出现了许多支持图形用户界面开发的软件系统，包括：

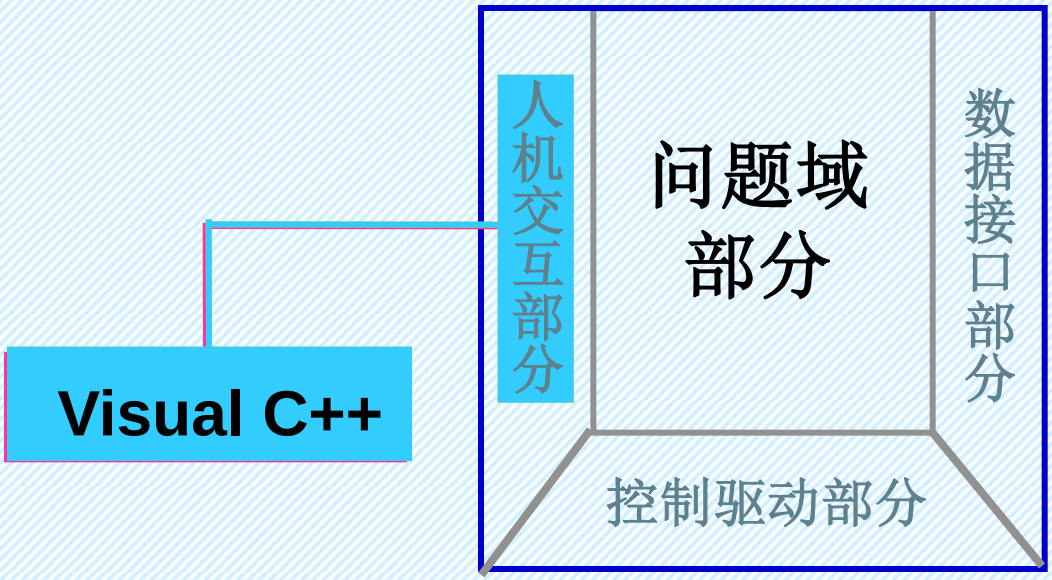
窗口系统（如X Window，News）；
图形用户界面（GUI）（如OSF/Motif，Open Look）；
可视化开发环境（如Visual C++，Visual Basic，Delphi）
——统称界面支持系统。

人机交互部分既取决于需求，又与界面支持系统密切相关。

人机界面的开发不仅是设计和实现问题，也包括分析问题——对人机交互需求的分析。

人机界面的开发也不纯粹是软件问题，它还需要心理学、美学等许多其它学科的知识。

把人机交互部分作为系统中一个独立的组成部分进行分析和设计，有利于隔离界面支持系统的变化对问题域部分的影响



12.2 人机交互部分的需求分析

对使用系统的人进行分析

以便设计出适合其特点的交互方式和界面表现形式

对人和机器的交互过程进行分析

核心问题是人如何命令系统，以及系统如何向人提交信息

（1）分析与系统交互的人（参与者）

人对界面的需求，不仅在于人机交互的内容，而且在于他们对界面表现形式、风格等方面的爱好。

前者是客观需求，对谁都一样

后者是主观需求，因人而异

分析工作包括

列举所有的人员参与者

调查研究

区分人员类型

统计（或估算）各类人员的比例

了解使用者的主观需求

（2）从用况分析人机交互

用况的构成

参与者的行为和系统行为按时间顺序交替出现，左右分明。形成交叉排列的段落。

每个段落至少含有一个输入语句或输出语句；
有若干纯属参与者自身或系统自身的行为陈述；
可能包含一些控制语句或括号。

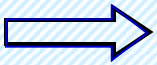
抽取方法：

删除所有与输入、输出无关的语句
删除不再包含任何内容的控制语句与括号
剩下的就是对一项功能的人机交互描述

例：

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
  通知供货员请求上货
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；
输入顾客交来的款数；
计算应找回的款数，
打印以上两个数目，
收款数计入账册。

（a）一个用况的例子



收款员·收款
输入开始本次收款的命令；

收款员．收款（人机交互）
输入开始本次收款的命令；
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；
打印商品编号、名称、数量、单价、总价；
end for；
打印应收款总数
输入顾客交来的款数
打印交款数及找回款数；

（c）得到人机交互描述

（b）删除与输入输出无关的陈述

人机交互的细化

输入的细化

- 输入步骤的细化
- 输入设备的选择
- 输入信息表现形式的选择

输出的细化

- 输出步骤的细化
- 输出设备的选择
- 输出信息表现形式的选择

输入与输出相比，输入在人机交互中起到主导作用

一次输入，广义地称为对系统的一条“命令”

（3）分析处理异常事件的人机交互

（4）命令的组织

不受欢迎的命令组织方式：

一条命令含有大量的参数和任选项

系统有大量命令，不加任何组织和引导

命令的组织措施——分解与组合

分解：将一条含有许多参数和选项的命令分解为若干命令步

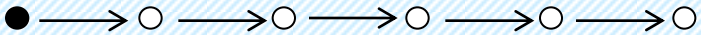
组合：将基本命令组织成高层命令，从高层命令引向基本命令

基本命令：使用一项独立的系统功能的命令。

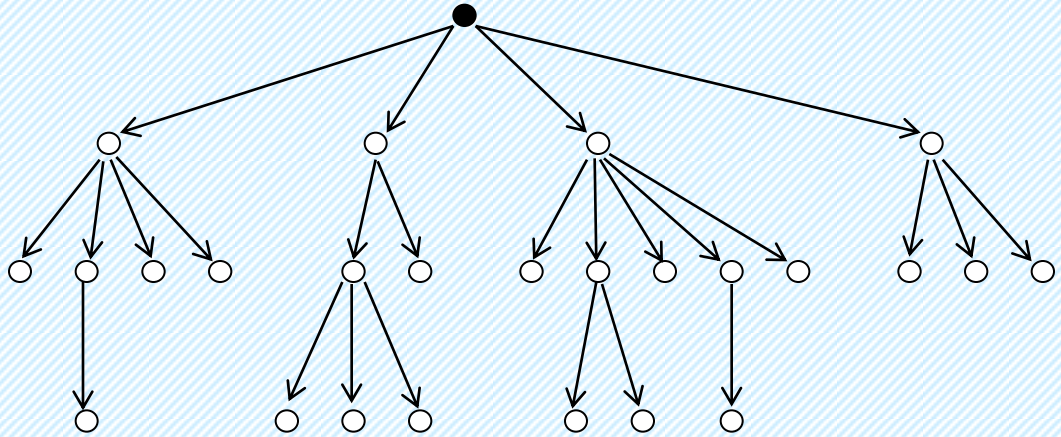
命令步：基本命令交互过程中所包含的具体输入步骤。

高层命令：由其他若干命令组合而成，起组织和引导作用

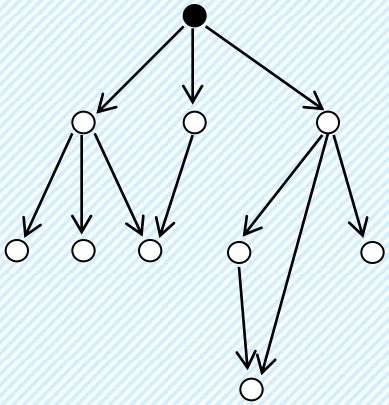
基本命令及其命令步的结构



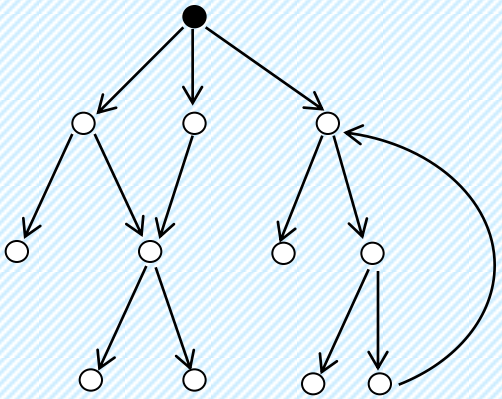
（a） 线性结构



（b） 树型结构

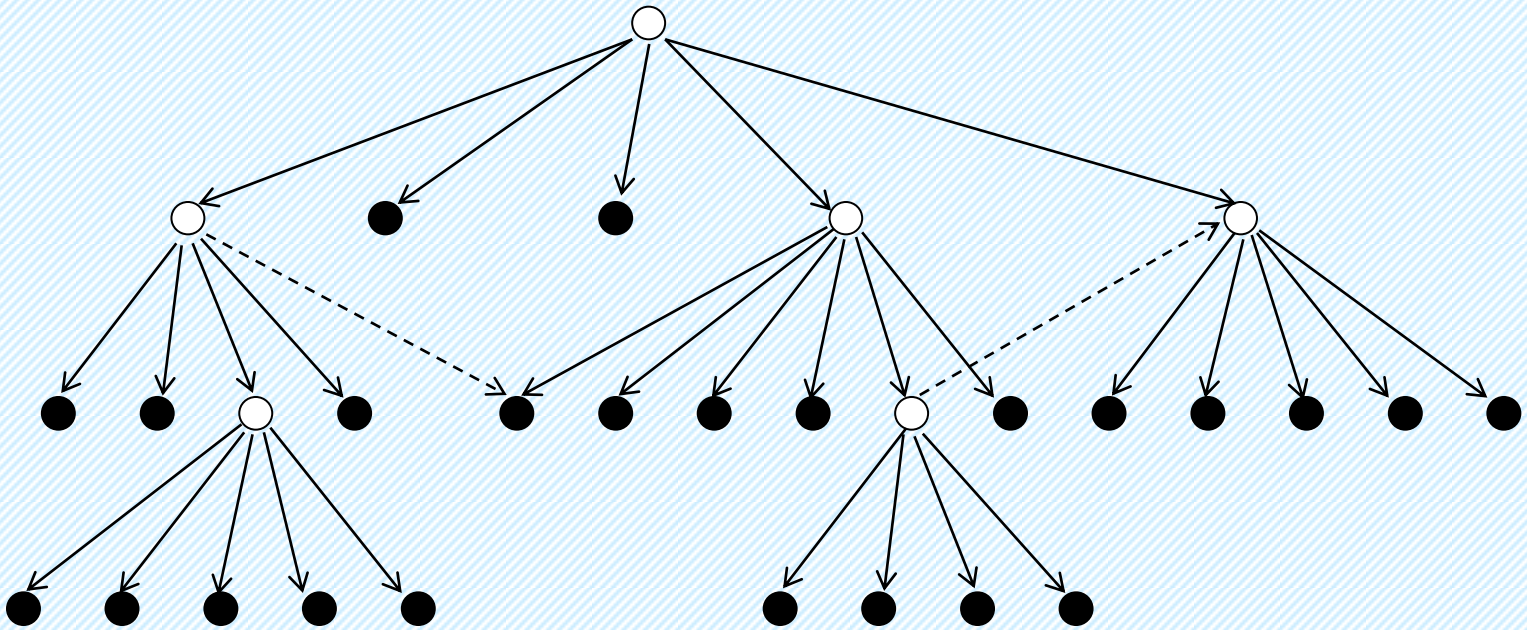


（c） 半序网状结构



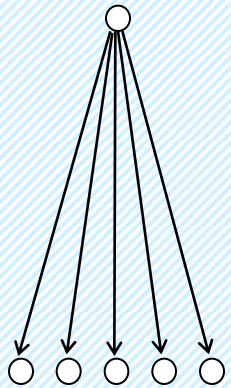
（d） 一般的网状结构

高层命令的组织结构

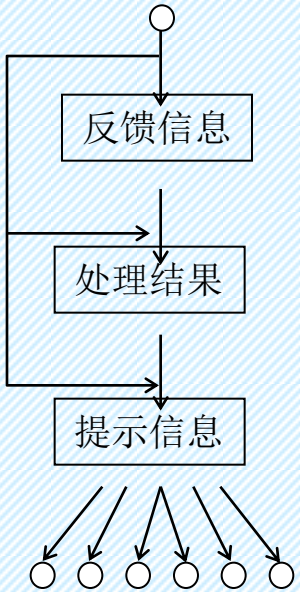


（5）输出信息的组织结构分析

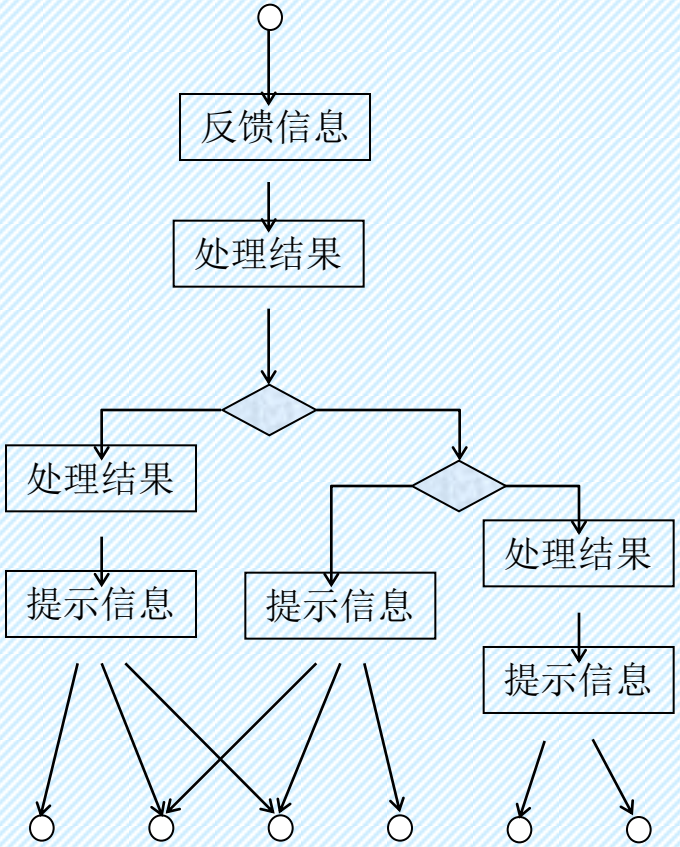
两层命令之间的输出信息结构



两层命令



典型的输出信息结构



复杂的输出信息结构

12.3 人机界面的设计准则

使用简便

一致性

启发性

减少人脑记忆的负担

减少重复的输入

容错性

及时反馈

其它：艺术性、趣味性、风格、视感……

12.4 人机界面的OO设计

（1）选择界面支持系统

窗口系统：“窗口系统是控制位映像显示器与输入设备的系统软件，它所管理的资源有屏幕、窗口、像素映像，色彩表、字体、光标、图形资源及输入设备。”

例：Smalltalk，Macintosh，X Window

图形用户界面（GUI）：在窗口系统之上提供了层次更高的界面支持功能，具有特定的视感和风格，支持应用系统界面开发的系统 。

例：OSF/Motif， Open Look

可视化编程环境：将窗口系统、GUI、可视化开发工具、编程语言以及类库结合为一体的可视化开发平台，支持用户以“所见即所得”的方式构造用户界面 。

例：Visual C++，Delphi，Visual Basic

考虑的因素：

硬件、操作系统及编程语言 ， 支持级别 ， 风格与视感

（2）根据人机交互需求选用界面元素

不同的界面支持系统提供不同的界面元素，常用的界面元素例如：窗口、菜单、对话盒、图符、滚动条等

系统的启动

选用实现主界面的界面元素，如框架窗口、对话框窗口

高层命令组织结构的实现

通过界面元素的构造层次体现高层命令的组织结构

例如：窗口—菜单—下级菜单……

基本命令的执行

通过高层命令引向基本命令

例如：窗口—菜单—菜单选项

详细交互过程的输入与输出

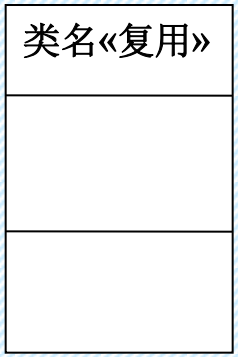
选择适当的界面元素完成每个命令步的输入与输出

异常命令的输入

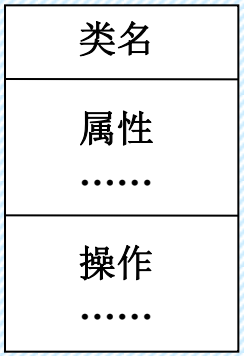
使用支持异常命令输入的界面功能，如鼠标右键菜单

（3）用OO概念表示界面元素

对象和类　尽可能使用界面类库中提供的可复用类



复用类库中的类



自定义的类

属性与 操作

用属性表示界面对象的静态特征

物理特征——如：位置、尺寸、颜色、立体效果

逻辑特征——聚合、关联

用操作表示界面对象的行为

例如：创建、激活、最大化、最小化、移动、选中、单击、双击 ……

整体-部分结构

表示界面元素之间的构成关系，例如：

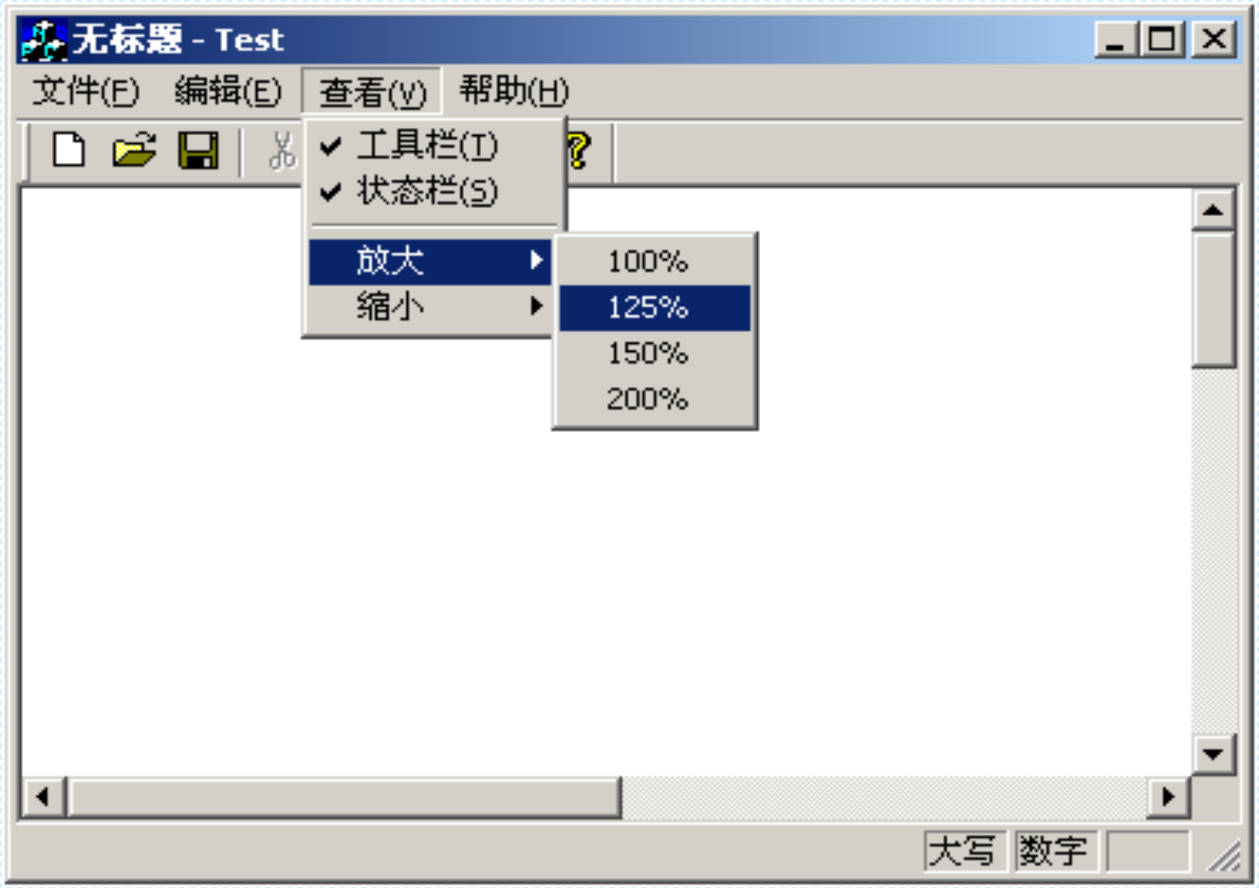
窗口 与 其中的菜单、按钮、图符、对话框、滚动条

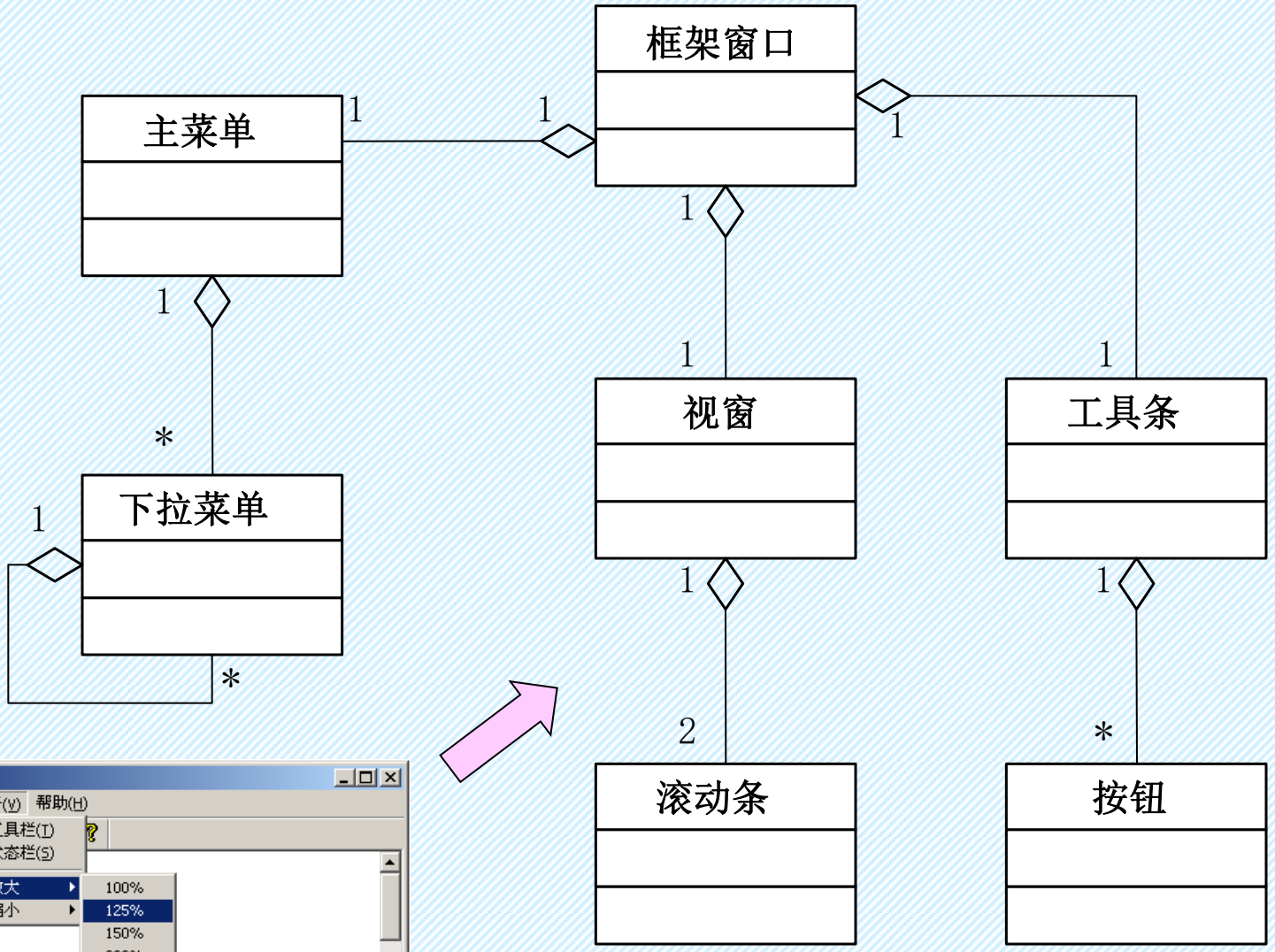
表示界面对象在操作中的逻辑层次

反映上、下两层命令之间的关系

例：

框架窗口



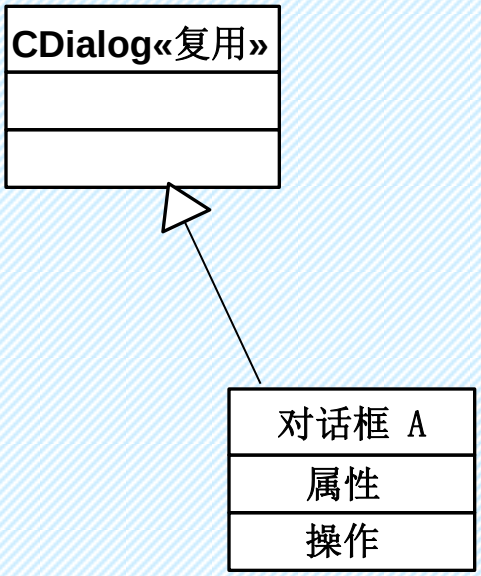

框架窗口
主菜单
下拉菜单
视窗
工具条
滚动条
按钮
1
*
2
无标题 - Test
文件(F) 编辑(E) 查看(V) 帮助(H)
工具栏(T)
状态栏(S)
放大
缩小
100%
125%
150%
200%
大写 数字

一般-特殊结构

表示较一般的界面类和较特殊的界面类之间的关系

自定义的类之间的一般-特殊关系

用一般-特殊结构特化可复用类



关联

表示界面类之间一个有特定意义的关系，例如：

消息

高层命令到低层命令——界面对象之间的消息
基本命令的执行——从界面对象向功能对象发消息
信息输出——从功能对象向界面对象发消息

12.5 可视化编程环境下的人机界面设计

（1）问题的提出

（2）所见即所得的界面开发

（3）设计的必要性

为实现提供依据

- 为了满足人机交互的需求，人机界面中要使用哪些界面对象？
- 交互过程中的各项输入和输出应由哪些界面对象完成？
- 如何通过界面对象类之间的各种关系体现人机交互命令的组织结构与层次？
- 如何通过界面对象和功能对象之间的消息实现它们之间的动态联系？

降低失败的风险

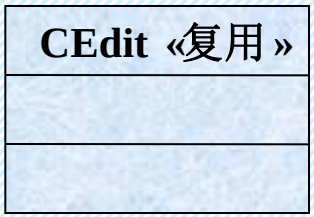
设计策略需要改进

类库的存在

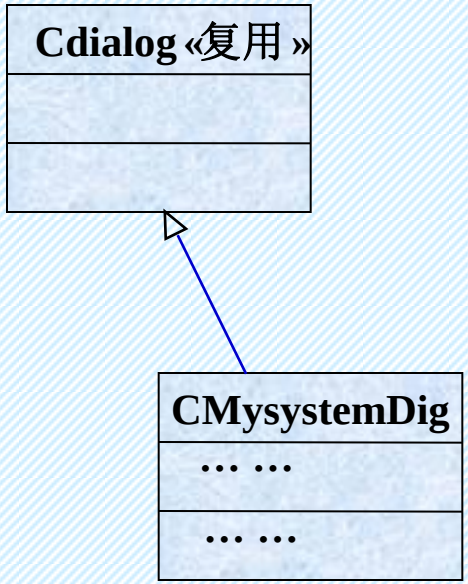
以所见即所得的定义界面对象的各种物理属性更为直接

（4）基于可视化编程环境的设计策略

类的设立——首先想到复用



直接复用



通过继承复用

属性——忽略物理特征，着重表示逻辑特征

设计阶段不必关心描述界面物理特征的属性

诸如：大小、形状、位置、颜色、边框、底纹、图案式样、三维效果等，

由实现人员去自主处理效果更好，效率更高

以主要精力定义描述界面逻辑特征的属性

表现命令的组织结构的属性、

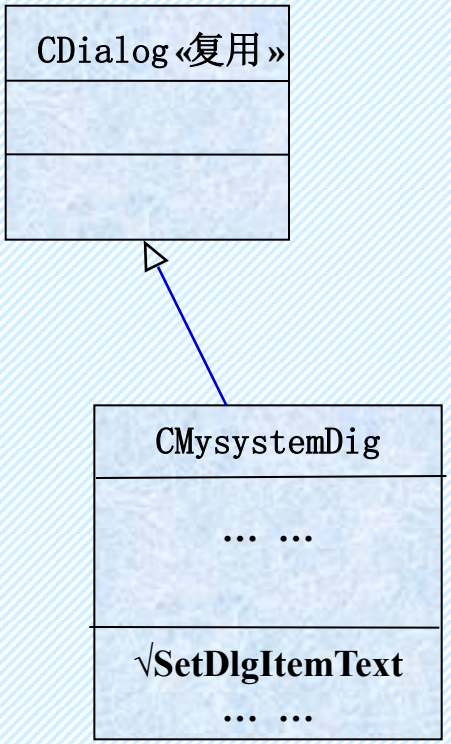
例如：菜单类的每个选项表示什么命令

表现界面元素之间组成关系和关联的属性

例如：对话框中包含哪些控件

操作——显式地表示从高层类继承的操作

例：



整体-部分结构——表现界面的组织结构和命令层次

通过整体-部分结构表现界面对象之间的组成关系和人机交互命令的层次关系——与采用其它界面支持系统的策略相同

区分界面对象的普通属性和它的部分对象

有些组成部分被作为对象的一个普通属性

——例如下拉菜单的选项，窗口的边框

有些组成部分则被作为一个部分对象

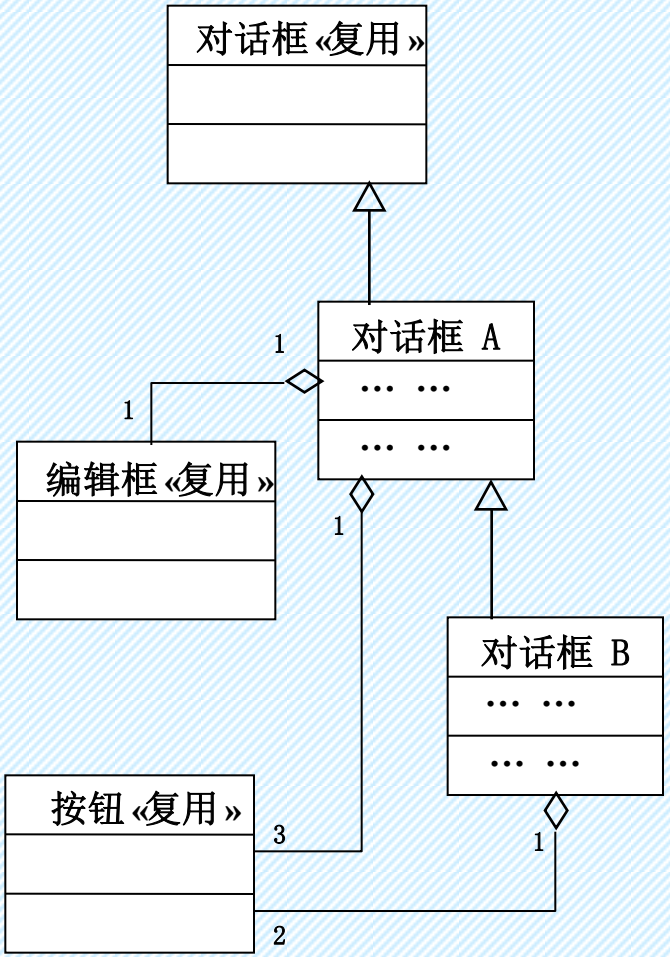
——例如对话框的一个下拉菜单或按钮

区分两种情况的依据

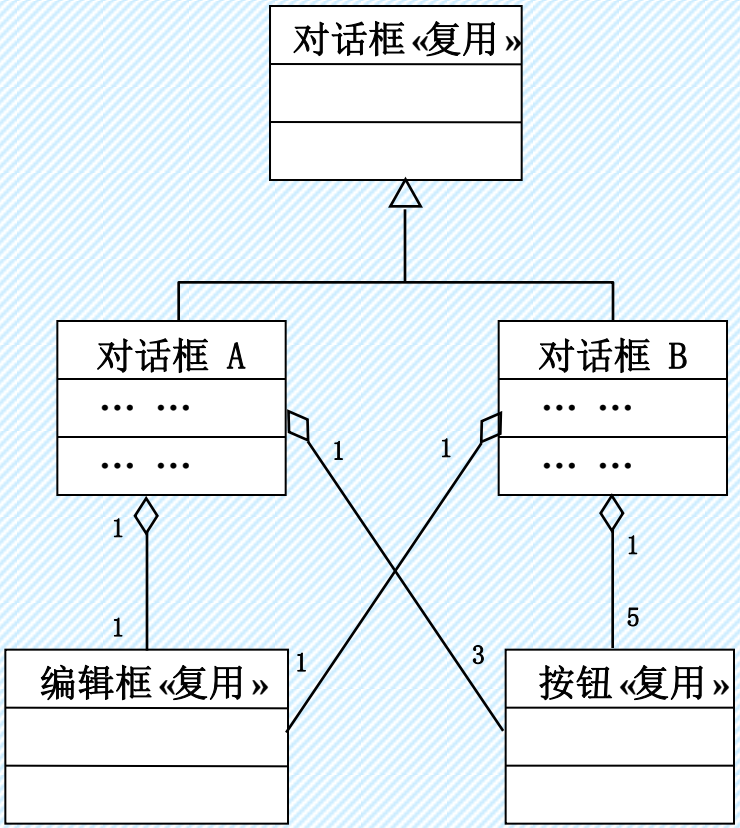
——环境类库有没有给出这种组成部分的类定义

一般-特殊结构——多从可复用类直接继承

例：



普通策略



直接继承可复用类的策略

消息——忽略自动实现的消息
注意需要编程实现的消息

1、界面对象接收到一个操作事件，通过它的一个操作向处理该事件的功能对象所发送的消息。

2、从功能对象向完成其输入/输出的界面对象发送的消息。

3、其它：凡是需要通过手工编程来实现的消息，都要在设计中加以表示。

13.1 什么是控制驱动部分

控制驱动部分是OOD模型的外围组成部分之一，由系统中全体主动类构成。这些主动类描述了整个系统中所有的主动对象，每个主动对象是系统中一个控制流的驱动者。

控制流（control flow）

——进程（process）和线程（thread）的总称

有多个控制流并发执行的系统称作并发系统（多任务系统）

为什么需要控制驱动部分

并发行为是现实中固有的

当前大量的系统都是并发系统（多任务系统），例如：

- 外围设备与主机并发工作的系统
- 有多个窗口进行人机交互的系统
- 多用户系统
- 多个子系统并发工作的系统
- 单处理机上的多任务系统
- 多处理机系统
- ……

多任务的设置

- 描述问题域固有的并发行为
- 表达实现所需的设计决策

隔离硬件、操作系统、网络的变化对整个系统的影响

13.2 相关技术问题

（1）由系统总体方案决定的实现条件：

计算机硬件

性能、容量和CPU数目

操作系统

对并发和通讯的支持

网络方案

网络软硬件设施、网络拓扑结构、通讯速率、网络协议等

软件体系结构（详后）

编程语言

对进程和线程的描述能力

其它商品软件

如数据管理系统、界面支持系统、构件库等

——对共享和并发访问的支持

（2）软件体系结构

抽象地说，软件体系结构描述了构成系统的元素、这些元素之间的相互作用、指导其组合的模式以及对这些模式的约束　——Mary Shaw

几种典型的软件体系结构风格

- 管道与过滤器风格（pipe and filter style）
- 数据抽象风格（data abstraction style）
- 面向对象风格（object-oriented style）
- 隐式调用风格（implicit invocation style）
- 层次风格（layered style）
- 仓库风格（repository style）
- 黑板风格（blackboard style）
- 解释器模型（interpreter model）
- 进程控制风格（process control style）
- 客户-服务器风格（client-server style）

（3）分布式系统的体系结构风格

主机+仿真终端体系结构

文件共享体系结构

客户-服务器体系结构

- 二层客户-服务器体系结构
- 三层客户-服务器体系结构
- 对等式客户-服务器体系结构
- 瘦客户-服务器体系结构

浏览器-服务器体系结构

（4）系统的并发性

进程（process）概念出现之前，并发程序设计困难重重

主要原因：

并发行为彼此交织，理不出头绪

与时间有关的错误不可重现

进程概念的提出使这个问题得到根本解决

进程的全称是顺序进程（sequential process），其基本思想是把并发程序分解成一些顺序执行的进程，使得：

每个进程内部不再包含并发行为

所以叫做顺序进程，其设计避免了并发问题

多个进程之间是并发（异步）执行的

所以能够构成并发程序

线程（Thread）

由于并行计算的需要，要求人为地在顺序程序内部定义和识别可并发执行的单位。
因此后来的操作系统大多支持线程概念。

线程与进程的区别：

进程既是处理机分配单位，也是存储空间、设备等资源的分配单位（重量级的控制流）；

线程只是处理机分配单位（轻量级的控制流）；

一个进程可以包含多个线程，也可以是单线程的。

控制流是进程和线程的总称。

应用系统的并发性

从网络、硬件平台的角度看：

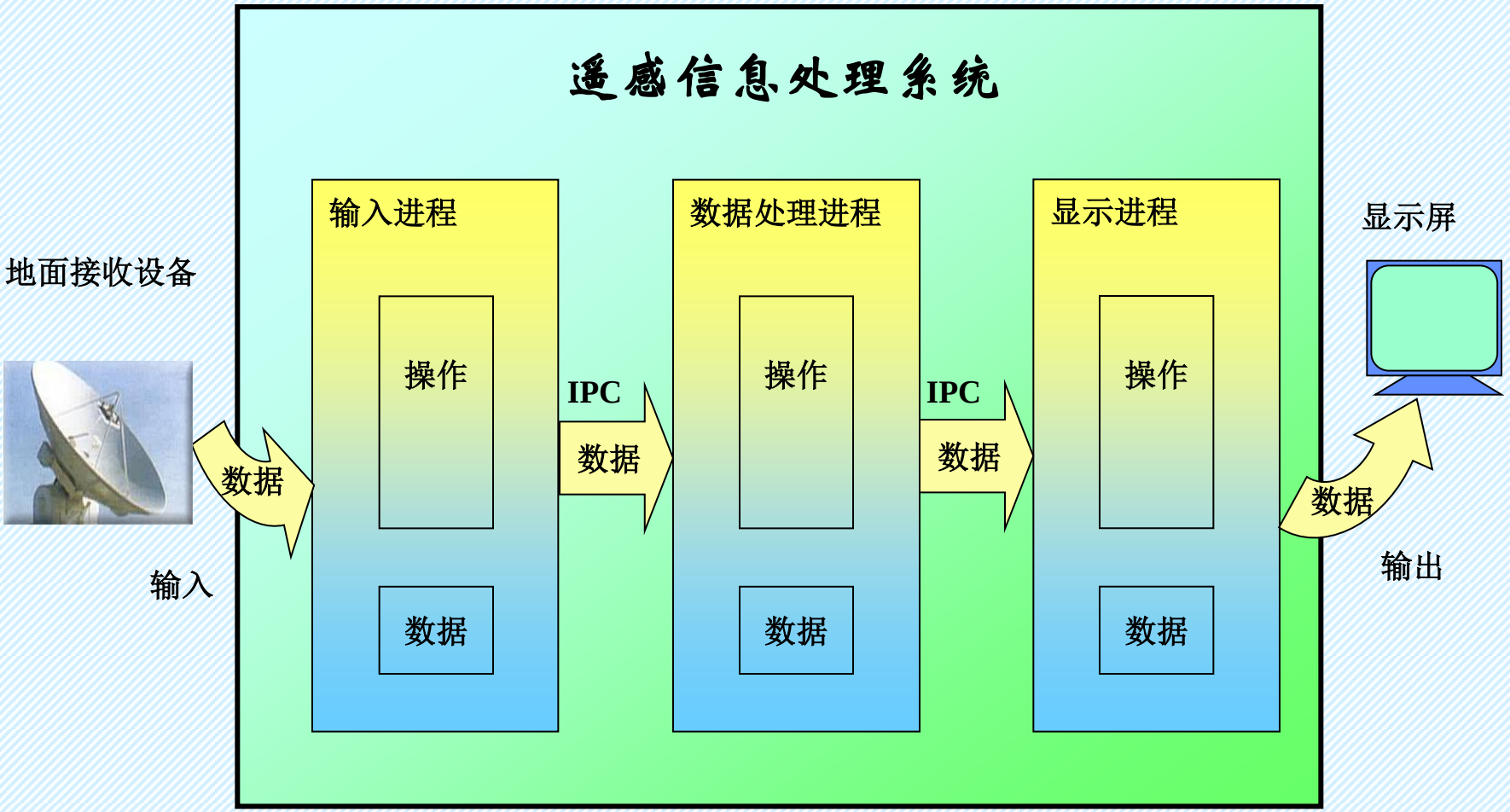
分布在不同计算机上的进程之间的并发
在多CPU的计算机上运行的进程或线程之间的并发
在一个CPU上运行的多个进程或线程之间的并发

从应用系统的需求看：

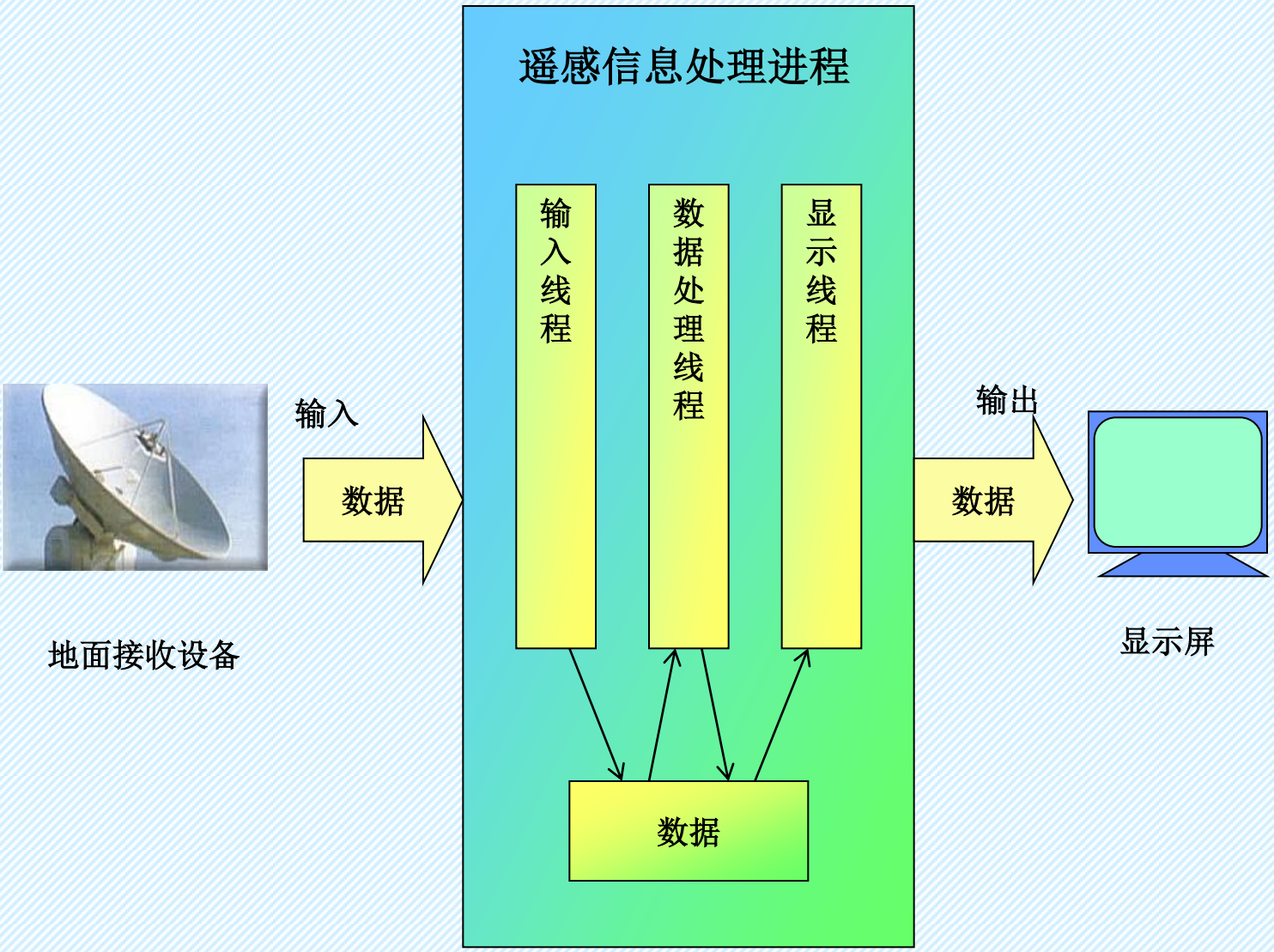
需要跨地域进行业务处理的系统
需要同时使用多台计算机或多个CPU进行处理的系统
需要同时供多个用户或操作者使用的系统
需要在同一时间执行多项功能的系统
需要与系统外部多个参与者同时进行交互的系统

处理应用系统并发性的例子

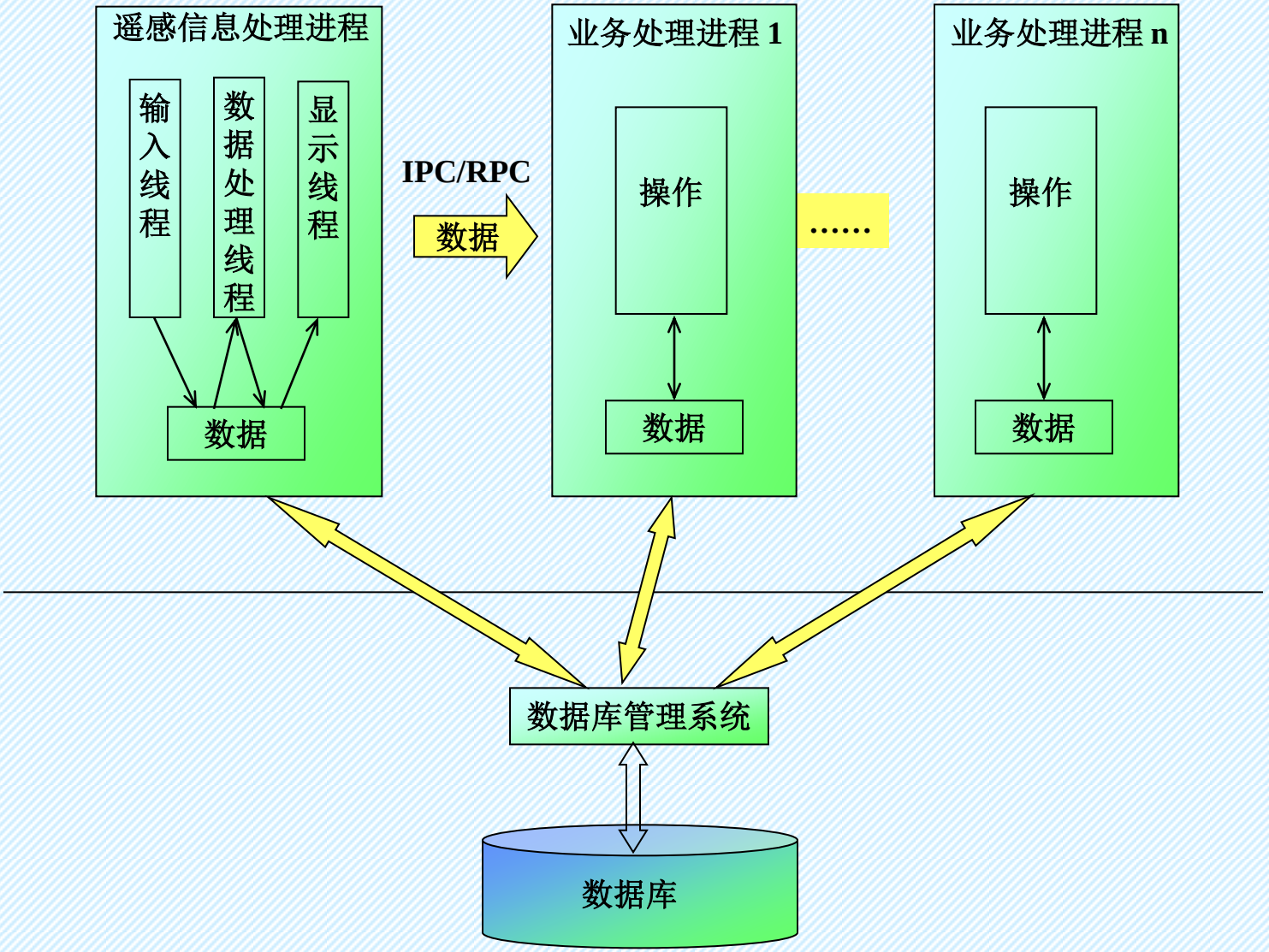
例：用多进程实现遥感信息的输入、处理和显示



例：用多线程实现遥感信息的输入、处理和显示



例：同时采用多进程和多线程



13.3 如何设计控制驱动部分

（1）选择软件体系结构风格

二层客户-服务器体系结构

（数据）服务器——客户机

三层客户-服务器体系结构

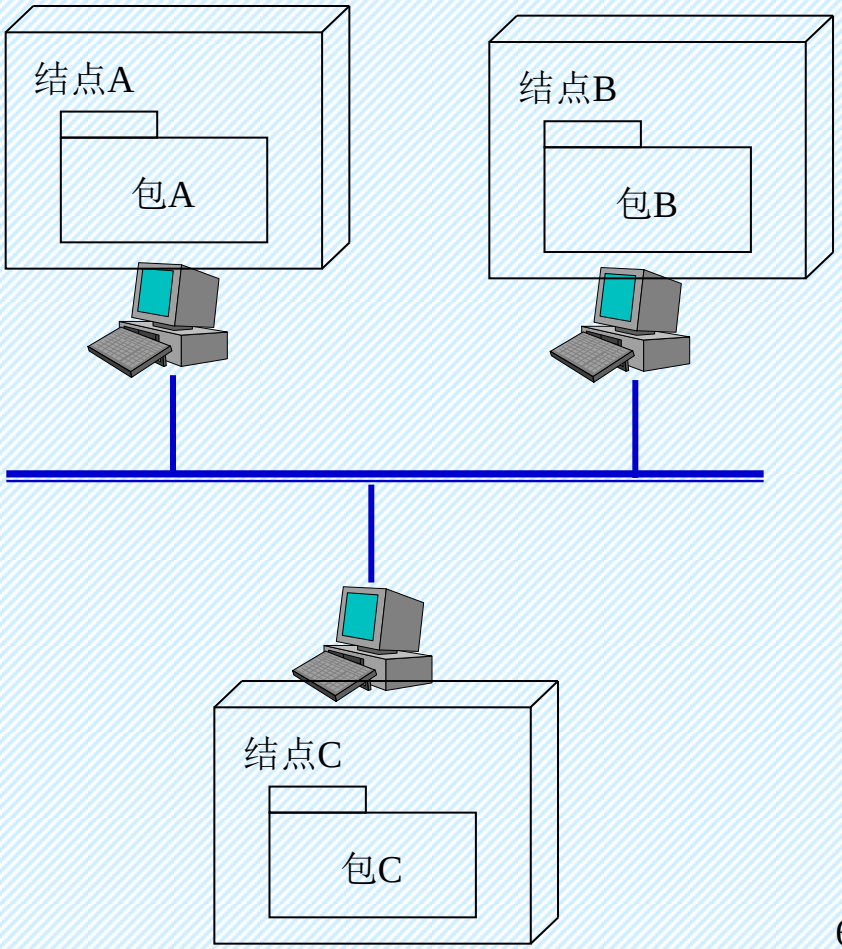
数据服务器——应用服务器——客户机

（2）确定系统分布方案

考虑分布方案之前，暂时将系统看作集中式的
确定分布方案之后，将对象分布到各个处理机上
以每台处理机上的类作为一个包

集中式
类图

分布到不同结点上



系统分布 包括功能分布和数据分布

在面向对象的系统中都体现于对象分布

原则：

减少远程传输，便于管理

对象分布

软件体系结构

系统功能在哪些结点提供

数据在哪些结点长期存储管理，在哪些结点临时使用

参照用况　把合作紧密的对象尽可能分布在同一结点

追踪消息　把一个控制流经历的对象分布在同一结点

类的分布：根据对象分布的需要

分布在每个结点上的对象，都需要相应的类来创建

（1）如果一个类只需要在一个结点上创建对象实例

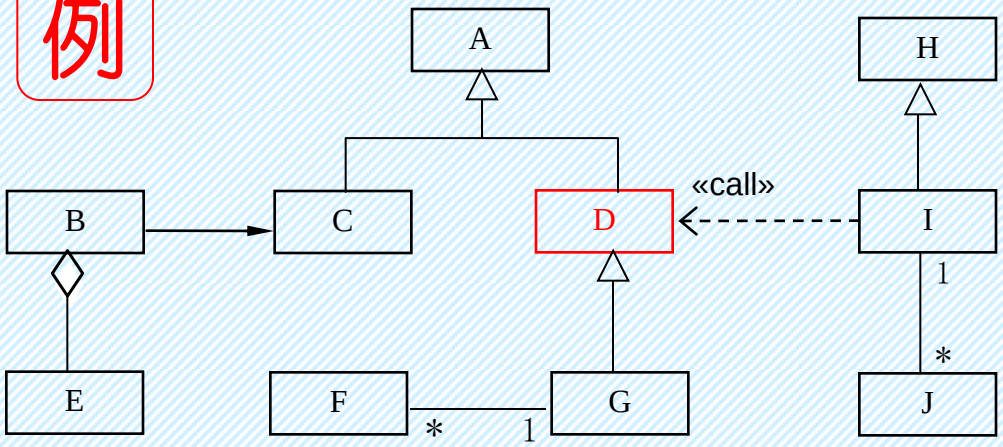
——把这个类分布在该结点上

（2）如果一个类需要在多个结点上创建对象实例

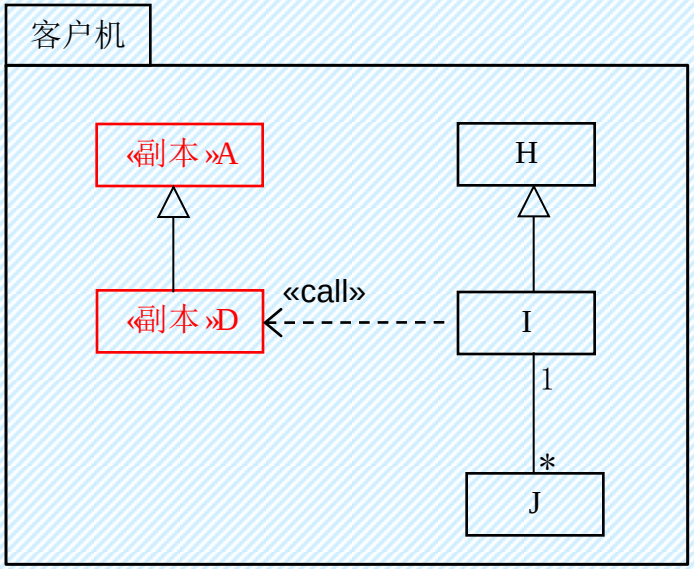
——把这个类分布到每个需要创建其实例的结点上

其中一个作为正本，其他作为副本

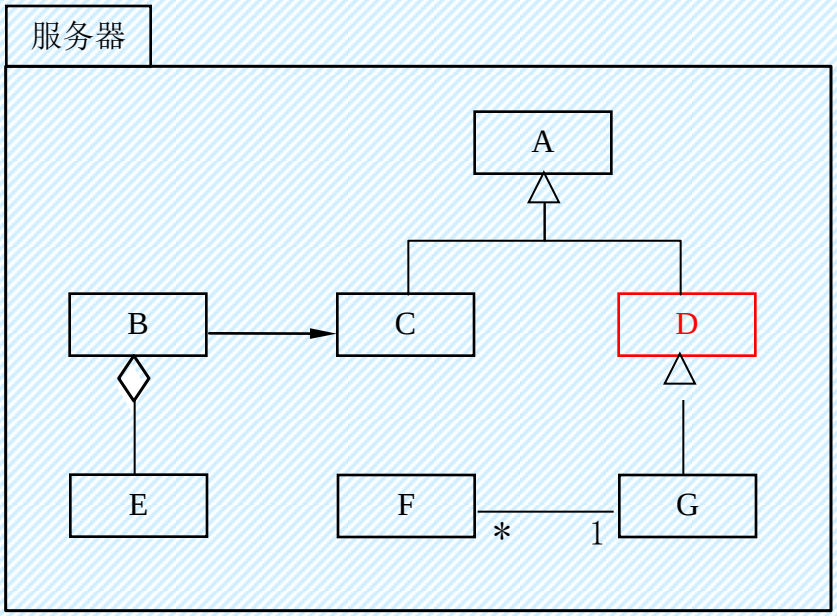
例



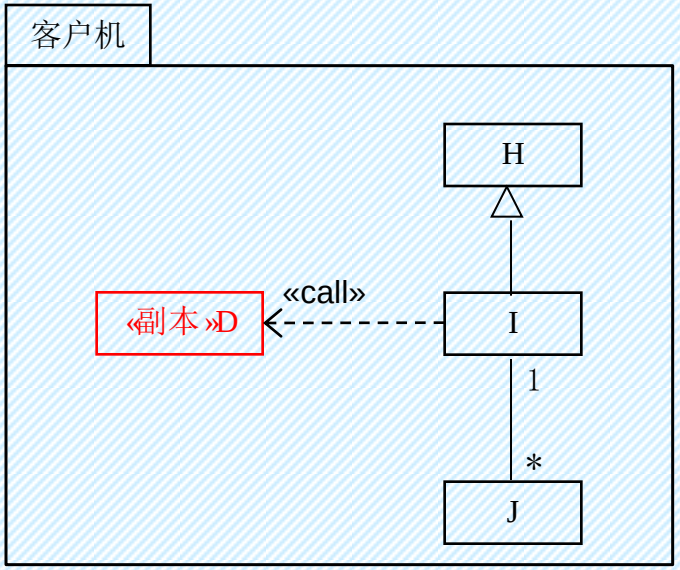
(a) 一个集中式类图



(c) 客户机包（第一种策略）



(b) 服务器包



(d) 客户机包（第二种策略）

13.3.3 识别控制流

（1）以结点为单位识别控制流

不同结点上程序的并发问题已经解决

考虑在每个结点上运行的程序还需要如何并发

（2）从用户需求出发认识控制流

有哪些任务必须在同一台计算机上并发执行

（3）从用况认识控制流 关注描述如下三类功能的用况

要求与其他功能同时执行的功能

用户随时要求执行 的功能

处理系统异常事件功能

（4）参照OOA模型中的主动对象

（5）为改善性能而增设的控制流

高优先级任务、低优先级任务、紧急任务

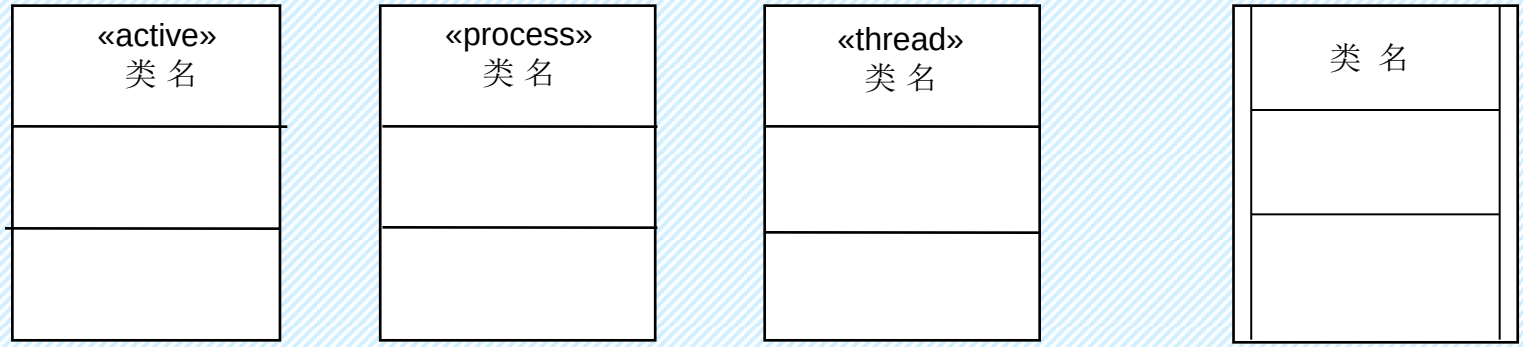
（6）实现并行计算的控制流（线程）

（7）实现结点之间通讯的控制流（进程）

（8）对其它控制流进行协调的控制流

13.3.4 用主动对象表示控制流

控制流是主动对象中一个主动操作的一次执行。其间可能要调用其他对象的操作，后者又可能调用另外一些对象的操作，这就是一个控制流的运行轨迹。

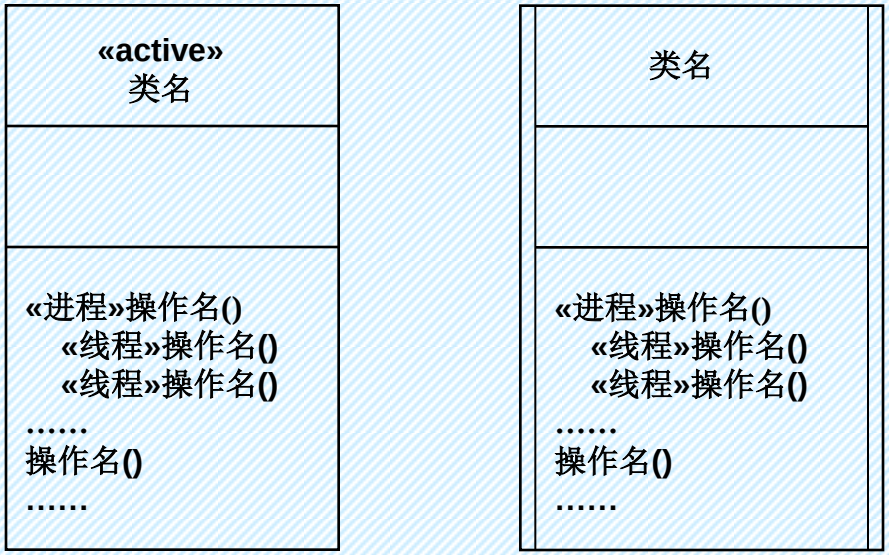


UML1的主动类表示法　　UML2的主动类表示法

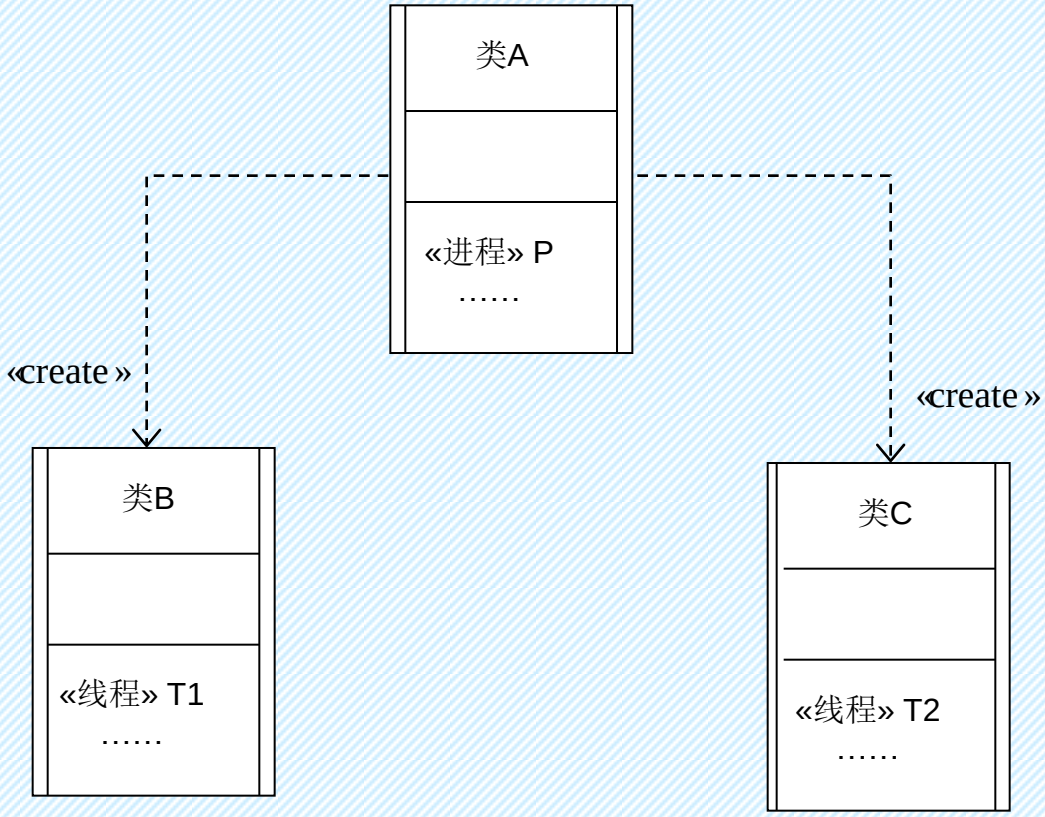
问题：

一个主动类可以有多个主动操作和若干被动操作，这种表示法不能显式地表示哪个（哪些）操作是主动操作。

用关键词表示主动操作



显示地表示由进程创建线程

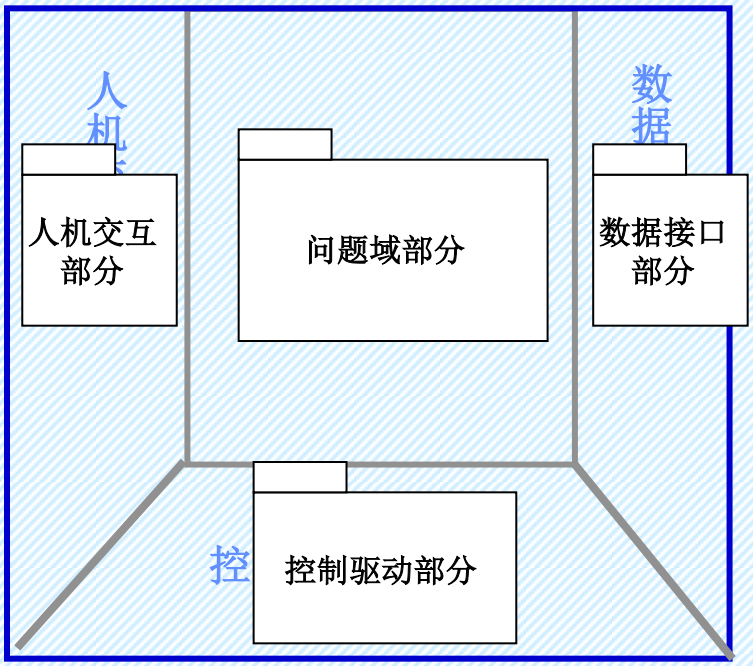


把控制驱动部分看成一个包

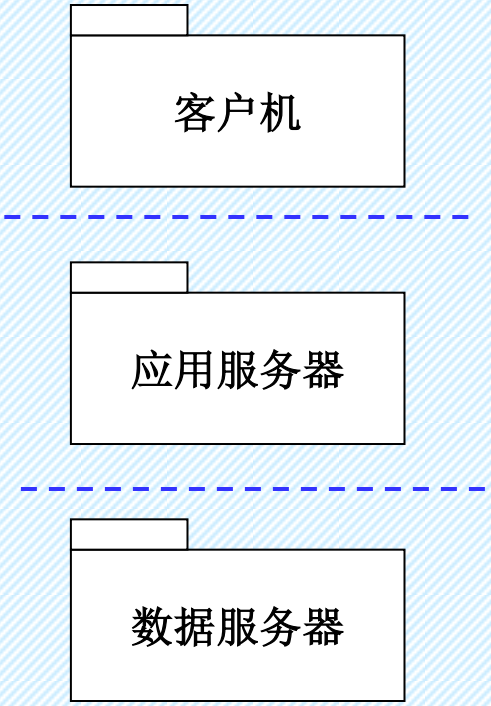
其中包含了系统中全部主动类

可以从多种观点把OOD模型划分成包

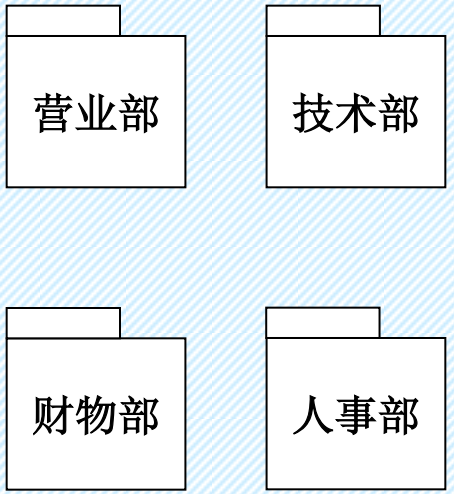
按OOD模型的四个组成部分



按分布的结点



按子系统



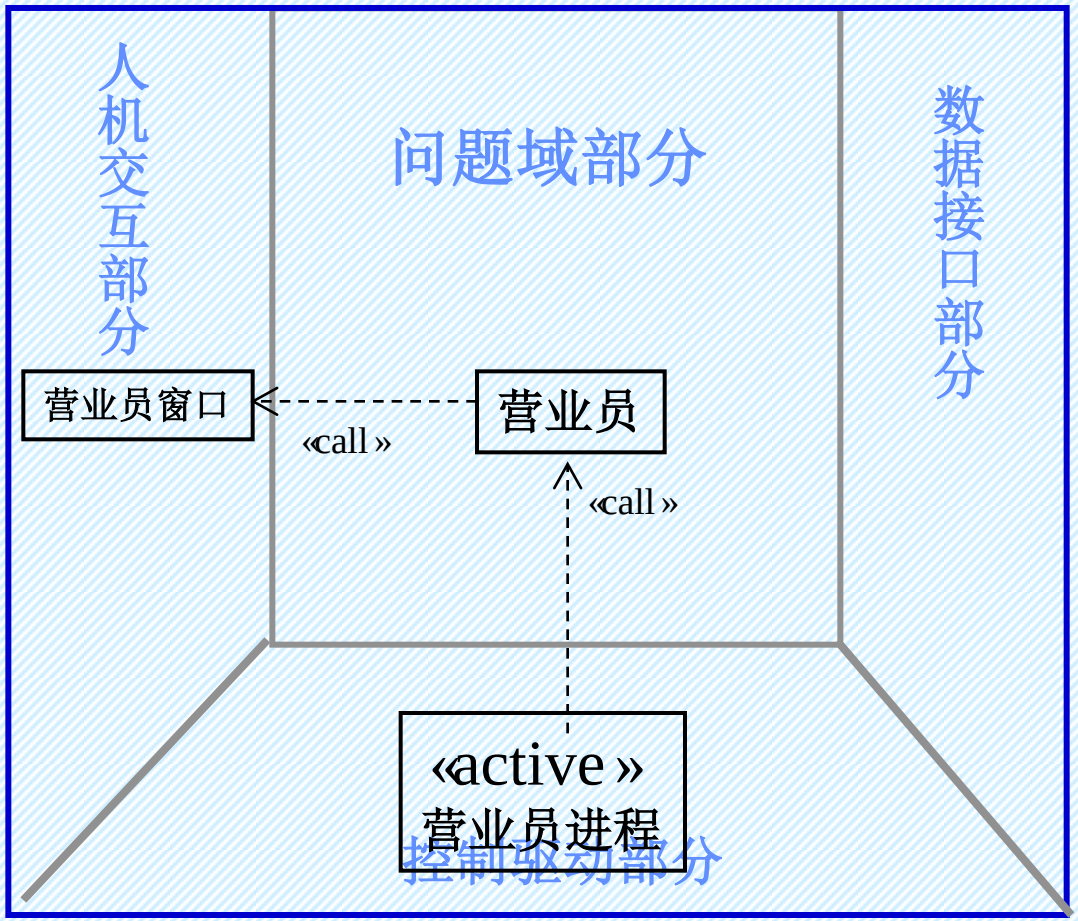
主动对象属于控制驱动部分，也可以同时属于其他部分

因为OOD模型的各个部分可以交叉

例：订单系统中营业员对象可以有不同的设计方案

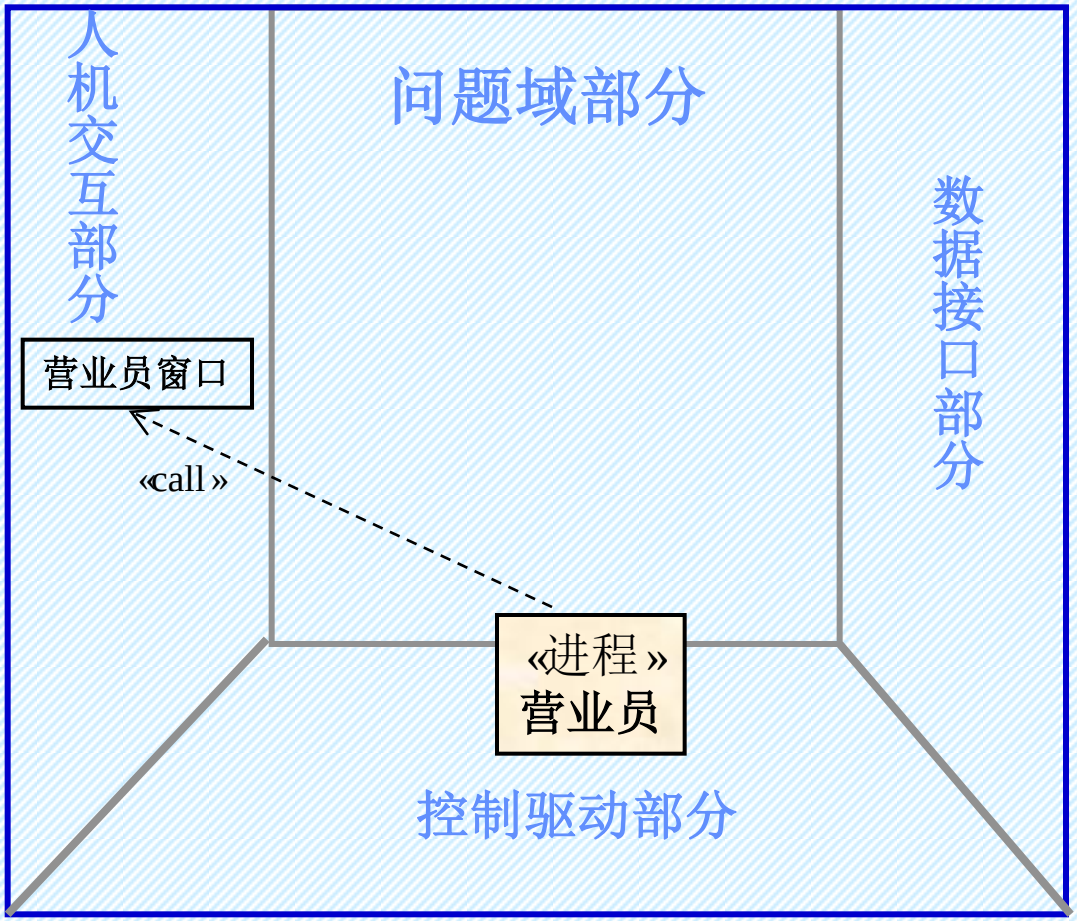
方案1：

无交叉



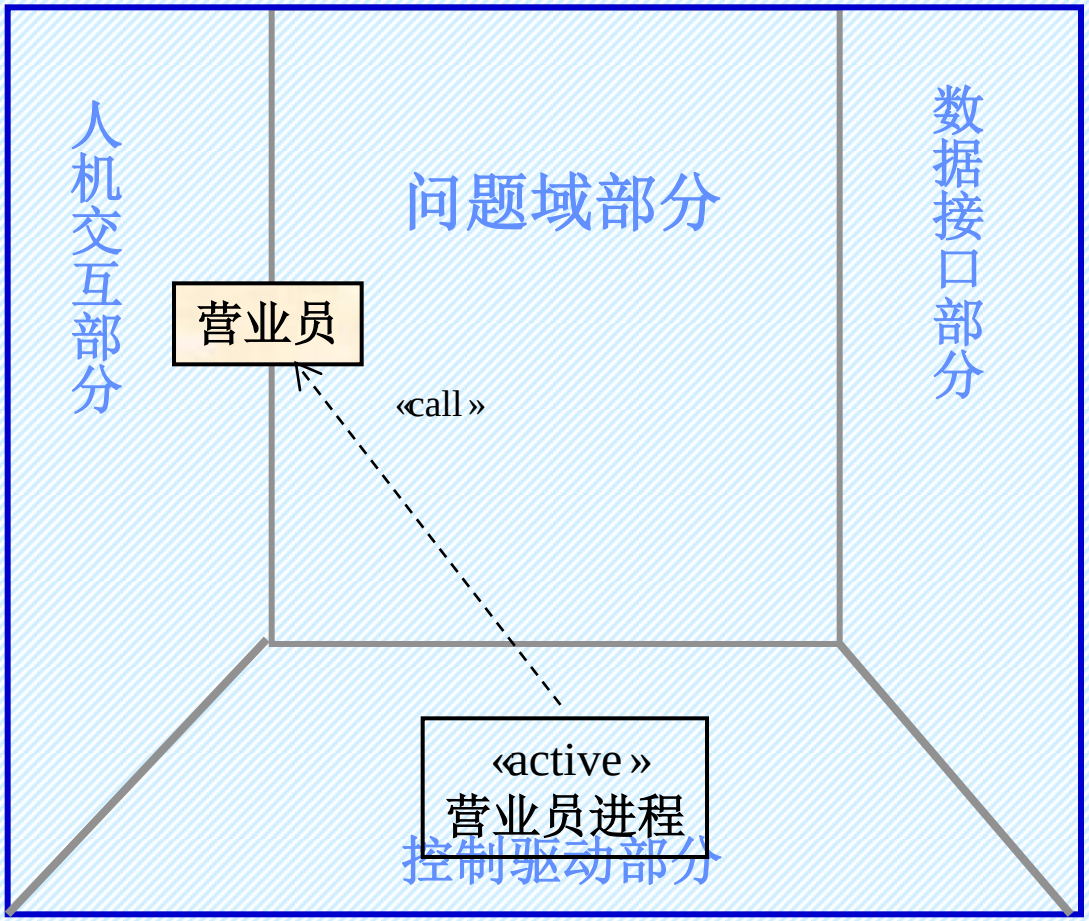
方案2：

问题域部分和控制驱动部分交叉



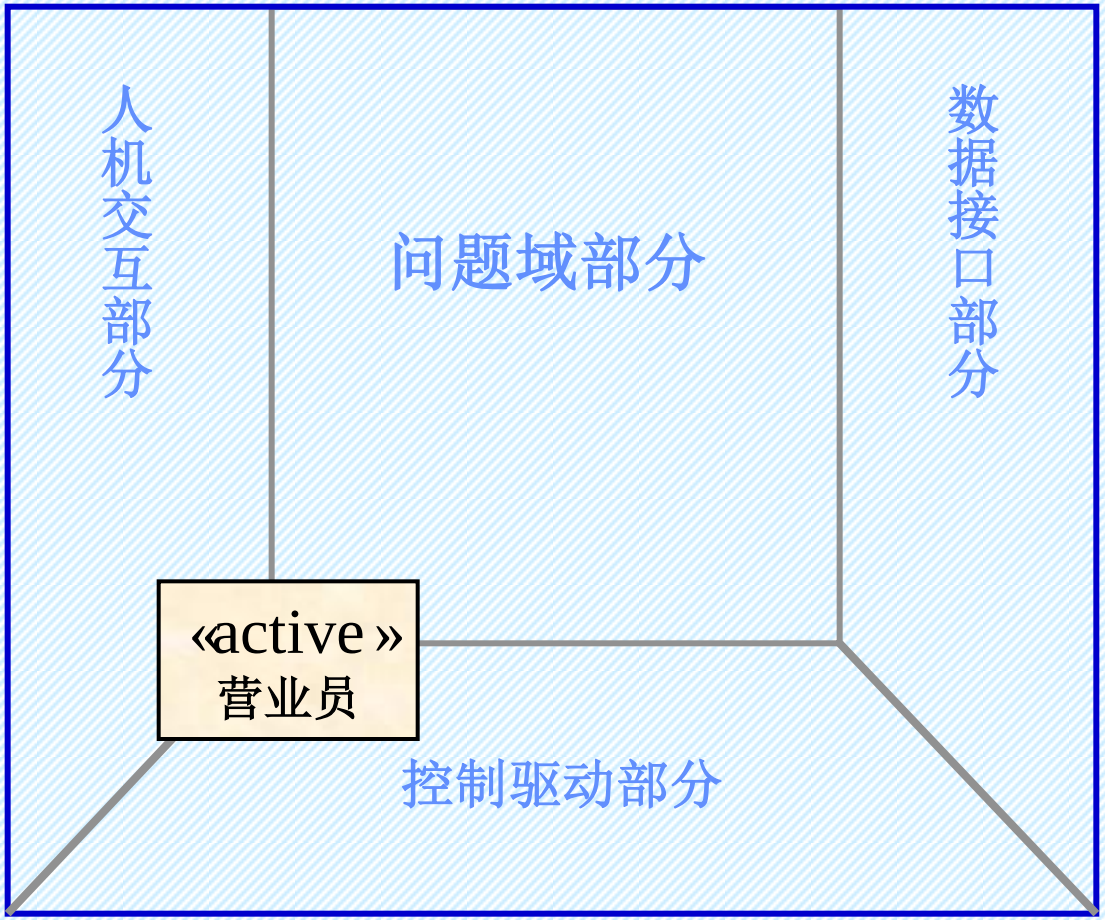
方案3：

问题域部分和人机交互部分交叉



方案4：

问题域部分、人机交互部分、控制驱动部分都交叉



提问：1、还有没有其他方案？

2、是否违背了隔离对问题域部分影响的初衷？

11.1 什么是数据接口部分

数据接口部分是OOD模型中负责与具体的数据管理系统衔接的外围组成部分，它为系统中需要长久存储的对象提供了在选定的数据管理系统中进行数据存储与恢复的功能。

大部分实用的系统都要处理数据的持久存储问题

- 数据保存于永久性存储介质
- 在数据管理系统的支持下实现其存储、检索和维护

在面向对象的系统中，数据的存储表现为对象存储

问题范围：

- 对象在永久性存储介质上的存储
- 只须存储对象的属性部分
- 可能只有一部分对象需要长久存储

不同的数据管理系统：

文件系统

关系型数据库管理系统

面向对象的数据库管理系统

——各有不同的数据定义方式和数据操纵方式

针对不同的数据管理系统，需要做不同设计

根据所选用的数据管理系统特点，设计一些专门处理其它对象的持久存储问题的对象

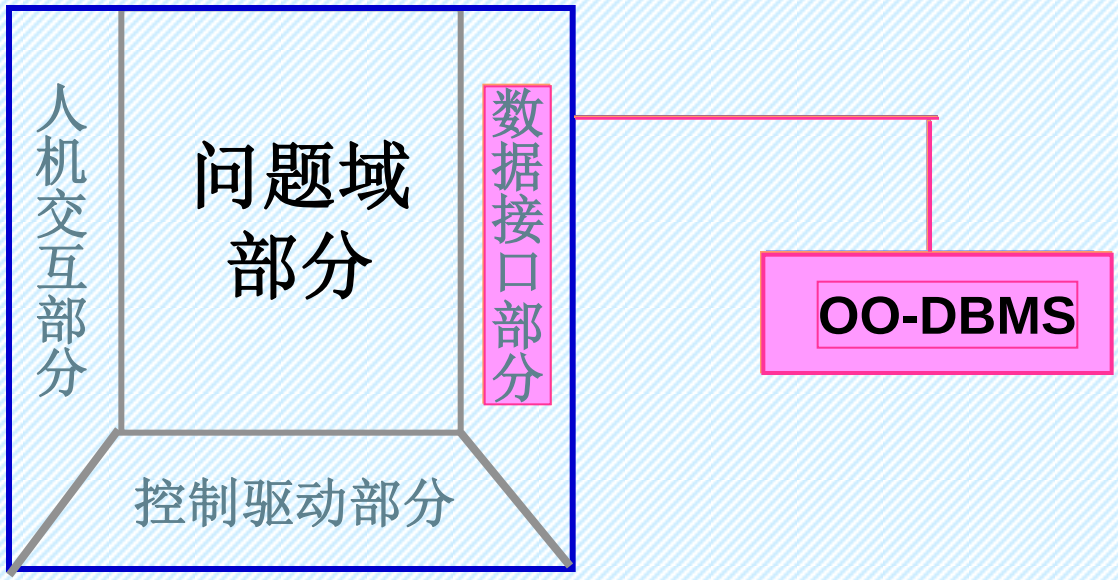
组织成一个独立的组成部分—— 数据接口部分

优点：

集中解决对象存储问题

隔离数据管理系统对其它部分的影响

选用不同的数据管理系统时，问题域部分变化较少



14.2 数据管理系统及其选择

数据管理系统

——实现数据存储、检索、管理与维护的系统

包括文件系统和数据库管理系统两大类

文件系统 **file system**

关系型数据库管理系统 **RDBMS**

面向对象的数据库管理系统 **OODBMS**

（一）文件系统

通常是操作系统的一部分

管理外存空间的文件数据

提供存储、检索、更新、共享和保护等功能

文件结构

物理结构

文件数据在存储空间的存放方法和组织关系

逻辑结构

呈现给用户的文件结构

如流式结构、记录式结构 等

文件系统提供的支持

在人机界面上进行操作的系统命令

在程序中使用的广义指令

创建、删除、打开、关闭、读、写、控制等

编程语言可以提供更方便的文件定义与使用方式

文件系统的优缺点

优点：

廉价，容易学习和掌握，对数据类型没有限制

缺点：

功能贫乏、低级

不容易体现数据之间的关系

只能按地址或者记录读写，

不能按属性进行数据检索与更新

缺少数据完整性支持

数据共享支持薄弱

（二）数据库管理系统

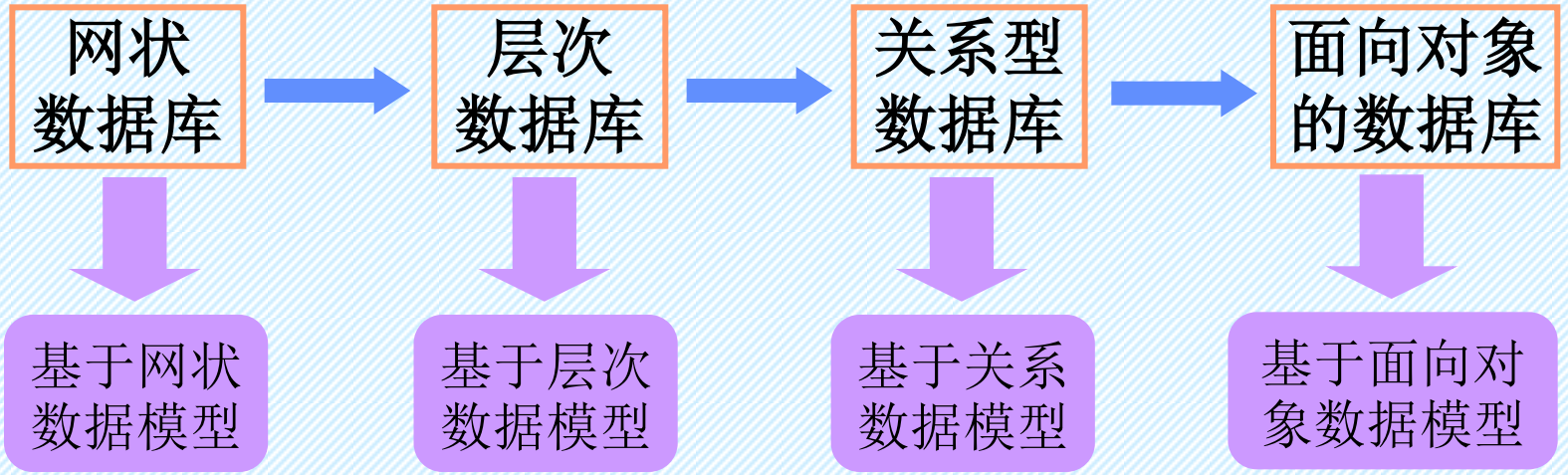
数据库：长期存储在计算机内、有组织、可共享的数据集合。其中的数据按一定的**数据模型**组织、描述和储存，具有较小的冗余度，较高的数据独立性和易扩展性，并可为各种用户共享。

数据库管理系统（DBMS）：用于建立、使用和维护数据库的软件。它对数据库进行统一的管理和控制，以保证数据库的安全性和完整性

数据模型：描述如何在数据库中定义数据及其操作，内容包括：

实体及实体间联系的数据结构描述，对数据的操作，完整性约束

不同的数据库基于不同的数据模型



1、关系数据库管理系统 RDBMS

关系模型

给定一组域D_1，D_2，…，D_n

其笛卡尔积$D_1 \times D_2 \times \ldots \times D_n$的一个子集就是一个关系，

又称二维表

基本要求：关系的每个属性必须是原子的

数据的组织：用二维表组织各类数据

既可存放描述实体自身特征的数据

也可存放描述实体之间联系的数据

每一列称作一个属性

每一行称作一个元组

数据的运算：

提供并、交、差等集合运算

以及选取、投影、联结等操作

关系数据库术语对照

数据库专业术语	开发者的习惯术语	用户习惯术语
关系 relation	文件 file	表 table
元组 tuple	记录 record	行 row
属性 attribute	字段，域 field	列 column

2、面向对象数据库管理系统 OODBMS

采用面向对象数据模型的数据库管理系统

背景：

越来越多的系统用OO技术开发（采用OO数据模型）

RDBMS的局限性（采用关系数据模型）

不能直接、有效地组织和存储对象数据，需要对数据模式进行转换，并提供相应的接口

因此出现了面向对象数据库管理系统——OODBMS

OODBMS的特征：

是面向对象的：

支持对象、类、对象标识、对象的属性与操作、封装、继承、聚合、关联、多态等OO概念。

具有数据库管理系统的功能：

数据定义与操纵语言、完整性保障、安全机制，并发控制、故障恢复、事务运行管理、可扩充

三种类型的OODBMS产品

1）在**OOPL**基础上增加数据库管理系统的功能

例如：GemStone 和 ObjectStore

2）对**RDBMS**进行扩充，使之支持面向对象数据模型

并向用户提供面向对象的应用程序接口

例如：Iris 和 POSTGRES

3）“全新的”**OODBMS**

即按照面向对象数据模型进行全新的设计

例如：O_2 和 DAMOKLES

目前状况：理论和技术上都有待完善

（三）数据管理系统的选择

理论上OODBMS最适合用OO方法开发的系统
实际上要权衡技术和非技术两方面的因素

非技术因素

与项目的成本、工期、风险、宏观计划有关的问题

产品的成熟性和先进性

价格

开发队伍的技术背景

与其它系统的关系

技术因素

考虑各种数据管理系统适应哪些情况，不适应哪些情况

文件系统的适应性

优点：可储任何类型的数据，包括具有复杂内部结构的数据和图形、图象、视频、音频等多媒体数据。以类和对象的形式定义的数据都可以用文件存储

适应：数据类型复杂，但对数据存取、数据共享、数据完整性维护、故障恢复、事务处理等功能要求不高的应用系统；

缺点：操作低级；数据操纵功能贫乏；缺少数据完整性支持；缺少多用户及多应用共享、故障恢复、事务处理等功能

不适应：数据操纵复杂、多样，数据共享及数完整性维护要求较高的系统。

关系数据库管理系统的适应性

优点：对数据存取、数据共享、数据完整性维护、故障恢复、事务处理等功能提供强有力的支持

适应：对这些功能要求较高的应用系统，以及需大量保存和管理各类实体之间关系信息的应用系统

问题：关系数据模型对数据模式的限制较多

当对象的内部结构较为复杂时，就不能直接地与关系数据库的数据模式相匹配，需要经过转换

更不适合图形、图象、音频、视频等多媒体数据 和经过压缩处理的数据

文件系统和RDBMS的优点和缺点形成了明显对照

但是对二者的选择却未必互相排斥，有时它们是互补的。某些应用系统可能同时采用RDBMS和文件系统，分别存储各自所适合的数据。

面向对象数据库管理系统的适应性

从纯技术的角度看，对用OO方法开发的系统采用OODBMS是最合理的选择，几乎没有不适合的情况。

如果某些项目不适合，主要是由于非技术因素，而不是技术因素

各种OODBMS采用的对象模型多少有些差异，与用户选用的OOA&D方法及OOPL中的匹配程度不尽一致，功能也各有区别，对不同的应用系统有不同的适应性

14.3 对象存储方案和数据接口的设计策略

针对三种
数据管理系统

文件系统

RDBMS

OODBMS

分别讨论

对象存储方案
——如何把对象映射到数据管理系统

数据接口部分的设计策略
——如何设计数据接口部分的对象类

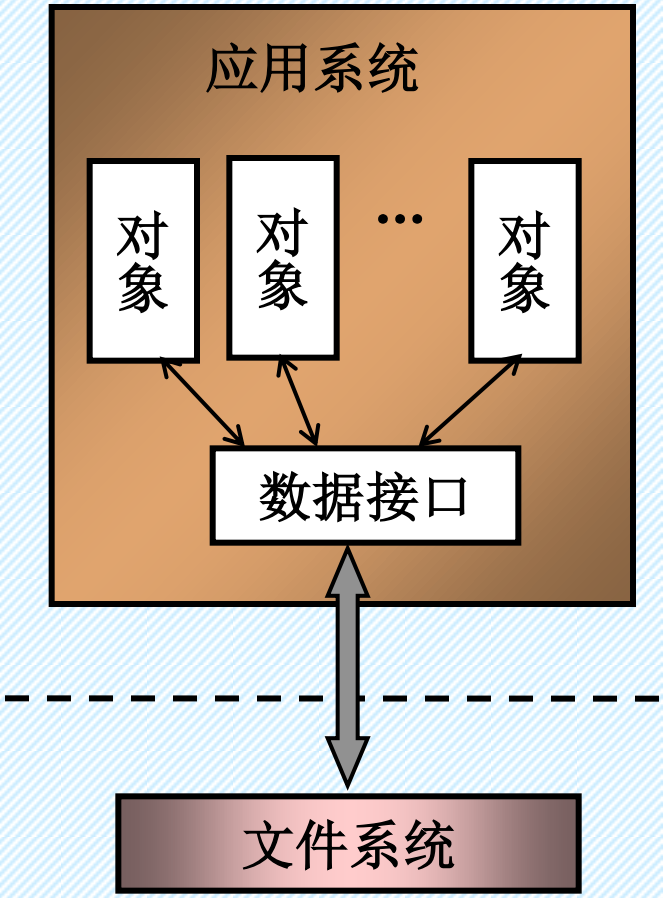
如何对问题域部分
做必要的修改

（一）针对文件系统的设计

1、对象在内存空间和文件空间的映像

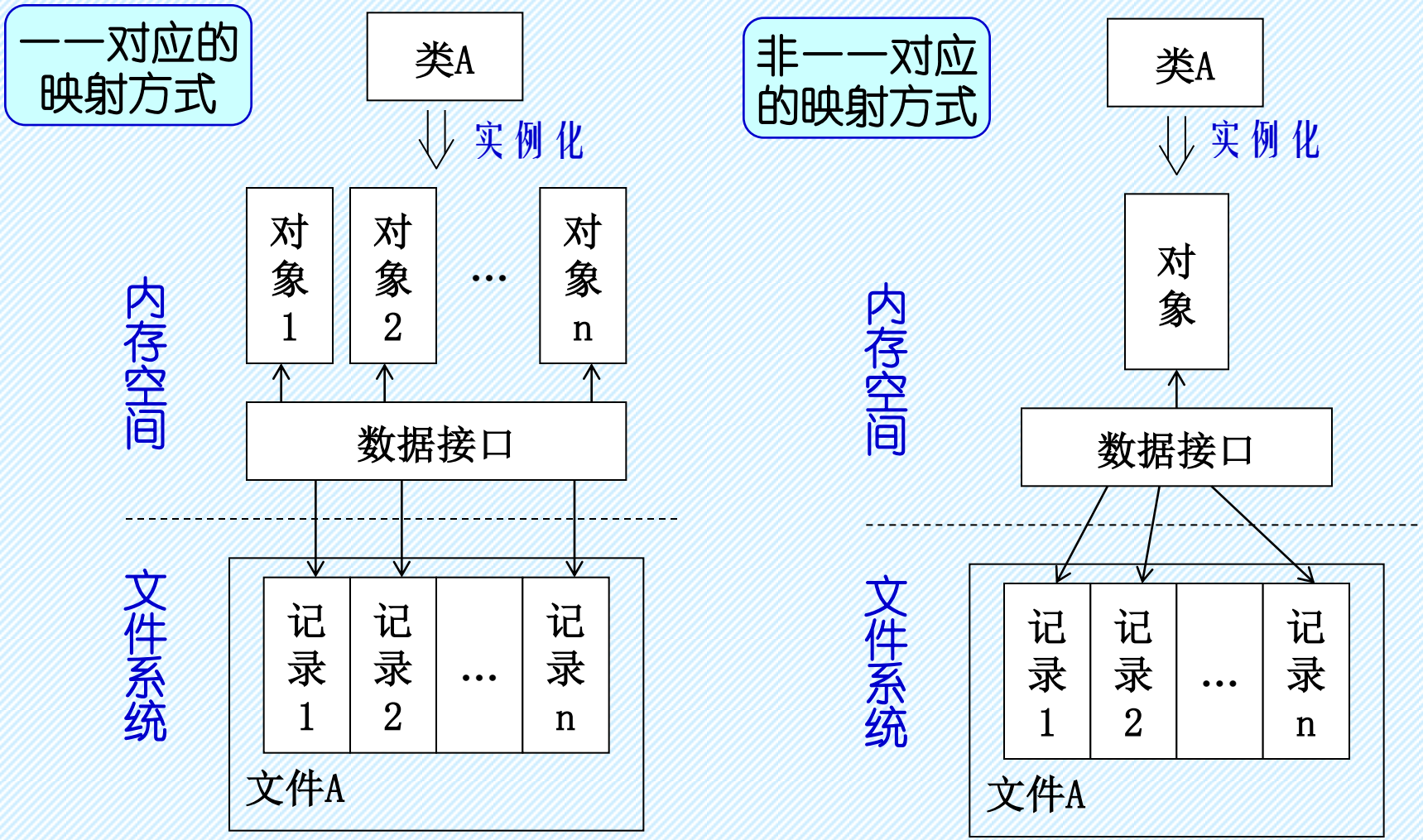
如何看待用文件系统存储对象

应用系统仍然是面向对象的



只是用文件系统存储对象的数据

从应用系统的对象到文件记录的不同映射方式



2、对象在文件中的存放策略

1）基本策略

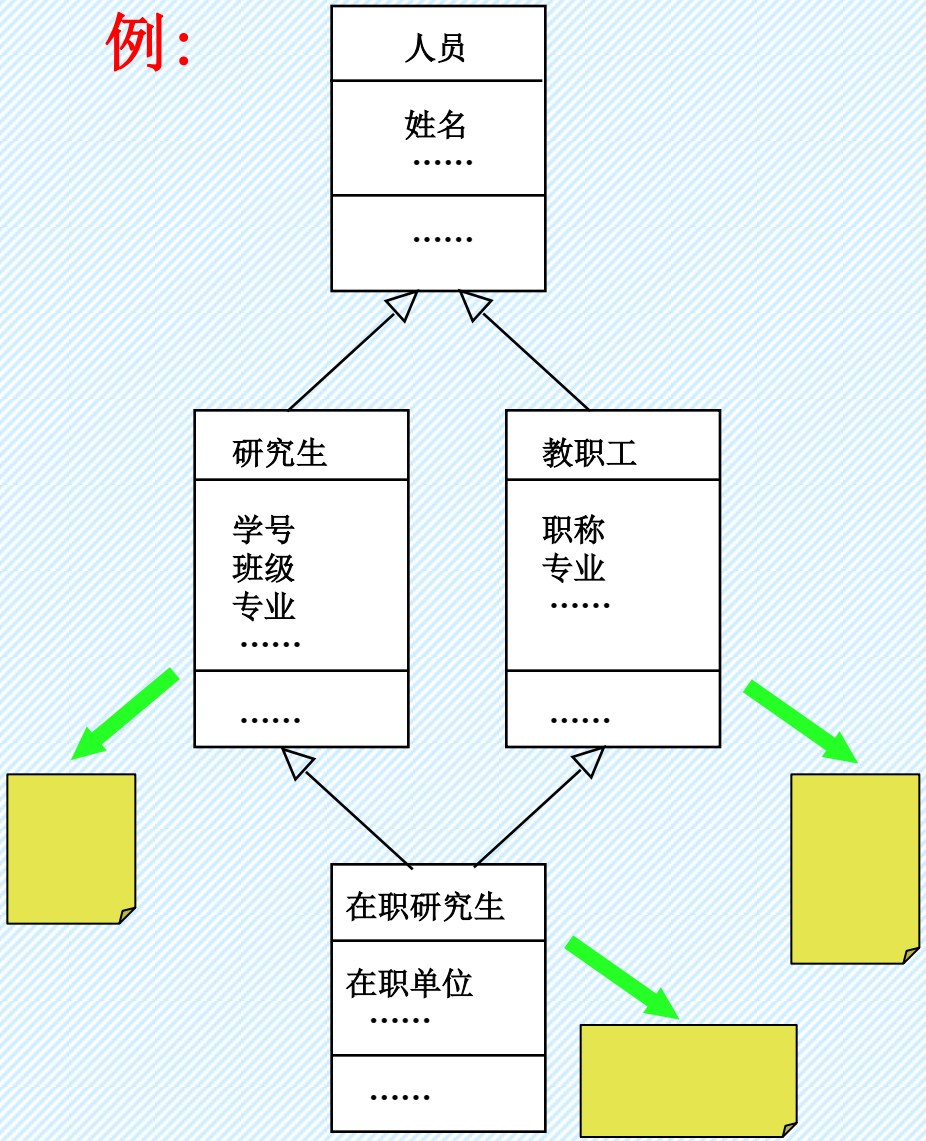
把由每个类直接定义、需要持久存储的全部对象实例存放在一个文件中；
每个对象实例的全部属性作为一个存储单元，占用该文件的一个记录。

如何理解“由一个类直接定义的”对象实例

另一种策略——
一个结构用一个文件

浪费空间
模糊了对象分类关系
使操作复杂化

例：



2）提高检索效率

——在对象和文件记录之间建立有规律的映射关系

对象名或关键字呈线性规律

- 按对象名或关键字的顺序形成文件记录
- 给出对象名称或关键字，快速地计算出它的存放位置

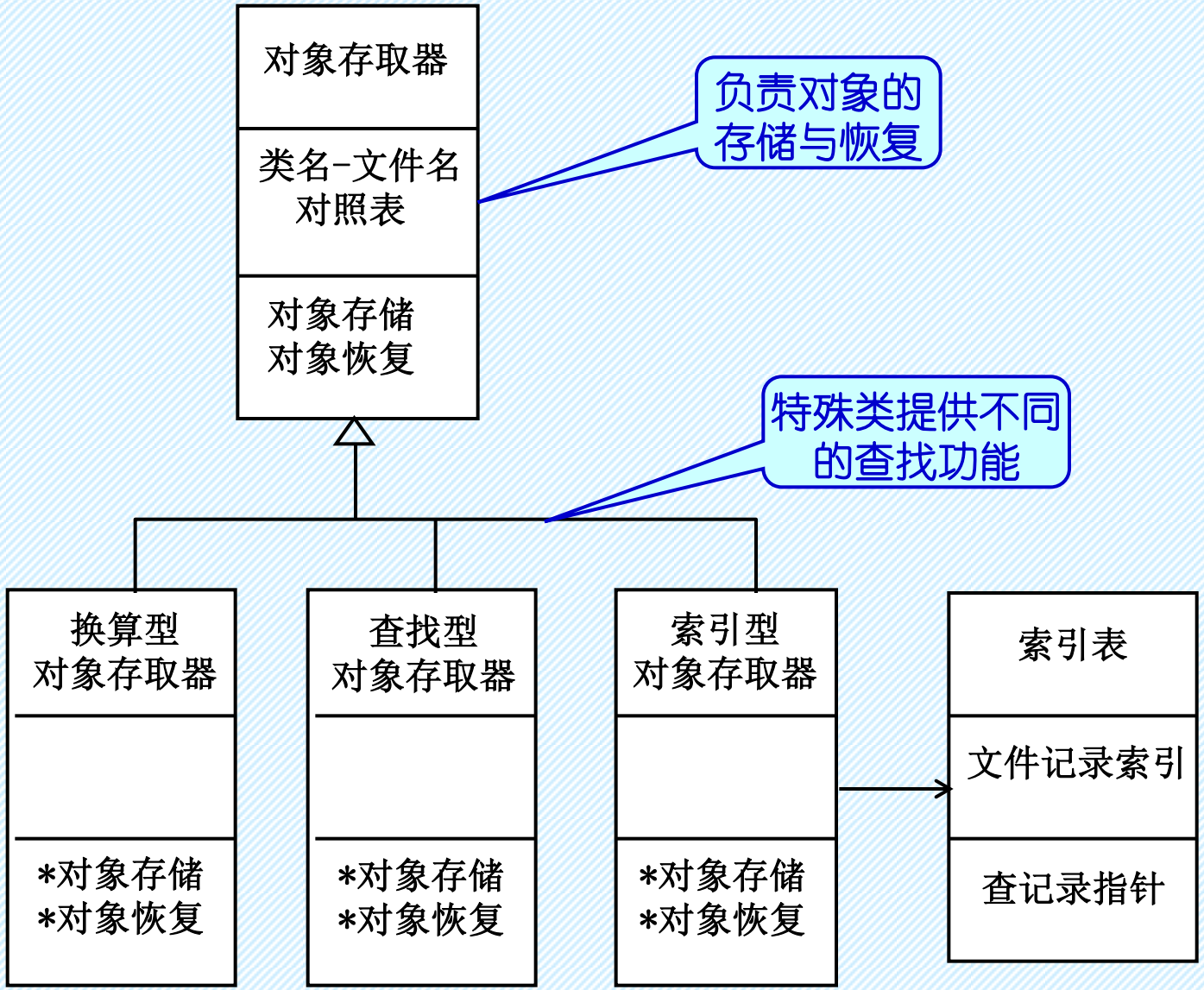
对象名称或关键字可以比较和排序

- 按关键字顺序安排记录，检索时采用折半查找法
- 建立按对象名称或者按关键字排序的索引表，通过该表中的记录指针找到相应的记录

其他措施

- 如散列表、倒排表、二叉排序树等等

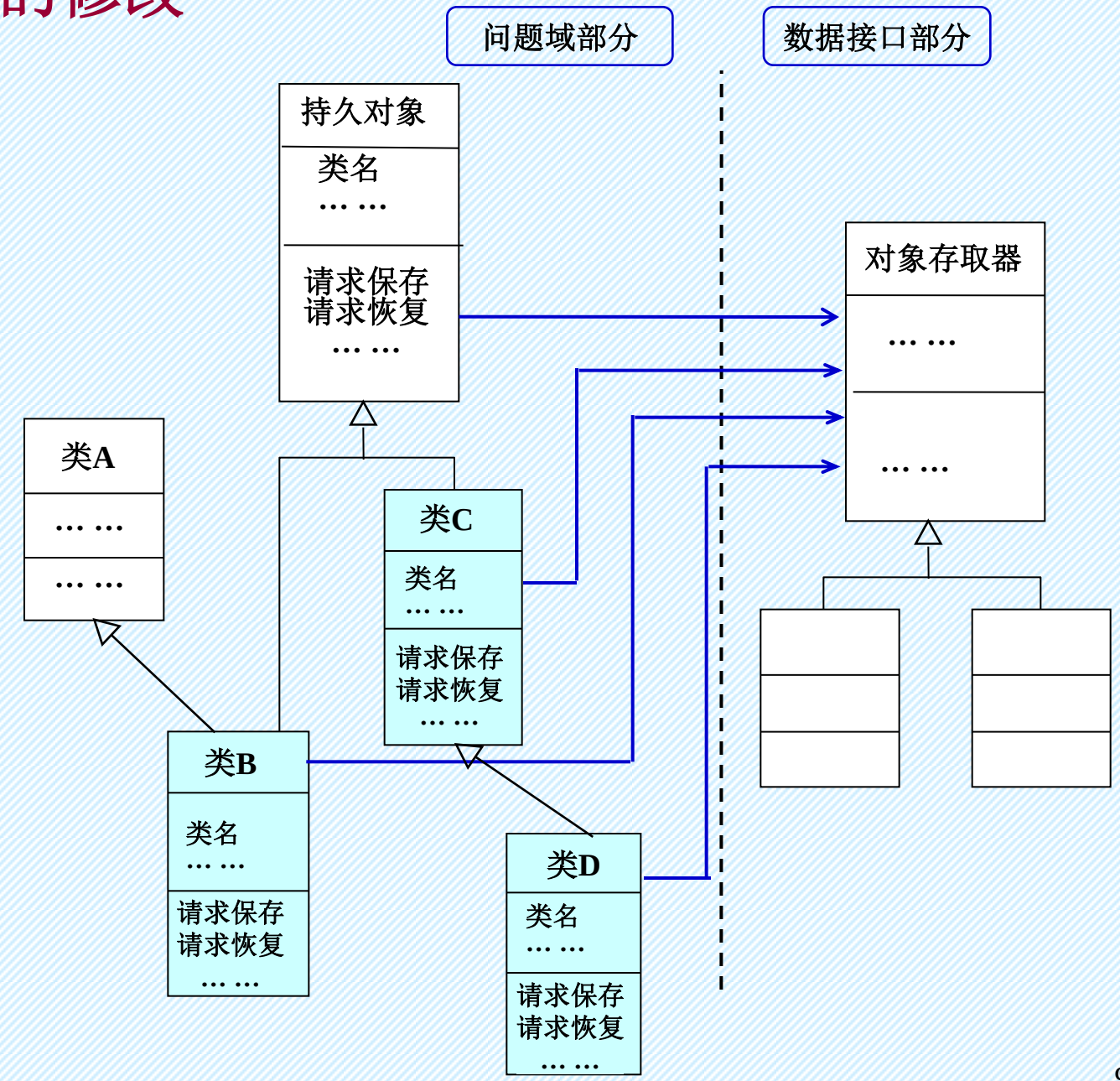
3、设计数据接口部分的对象类



4、问题域部分的修改

增加一个一般类来定义它们，作为共同协议，供所有的持久对象类继承

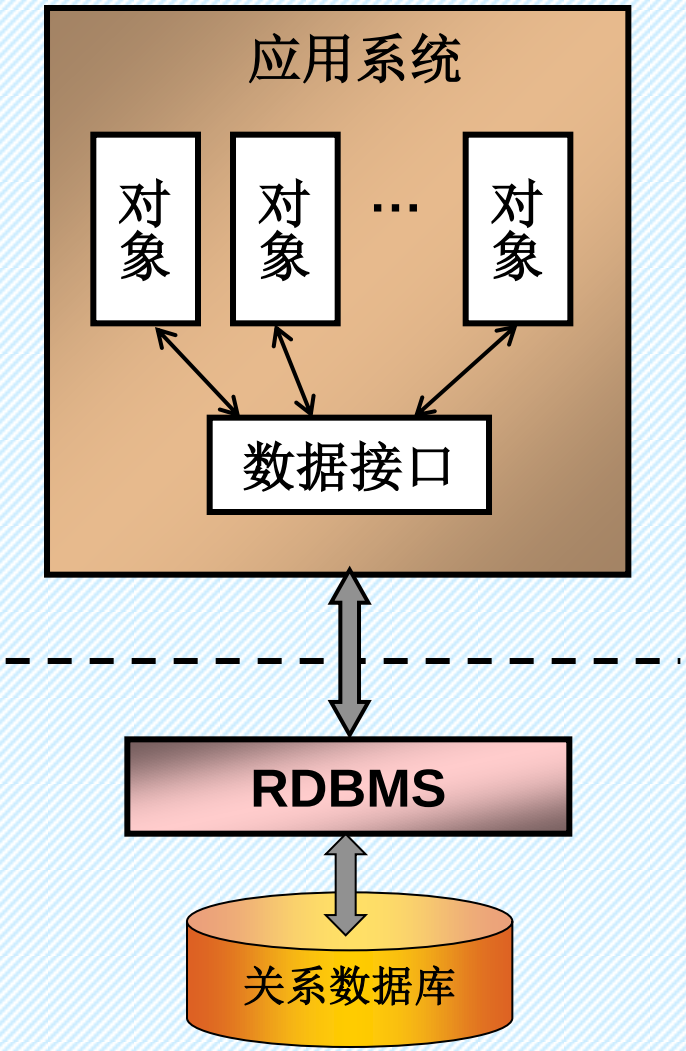
每个持久对象类都要增加请求存储和恢复所需的属性和操作，以便向数据接口部分发出请求



（二）针对RDBMS的设计

1、对象及其对数据库的使用

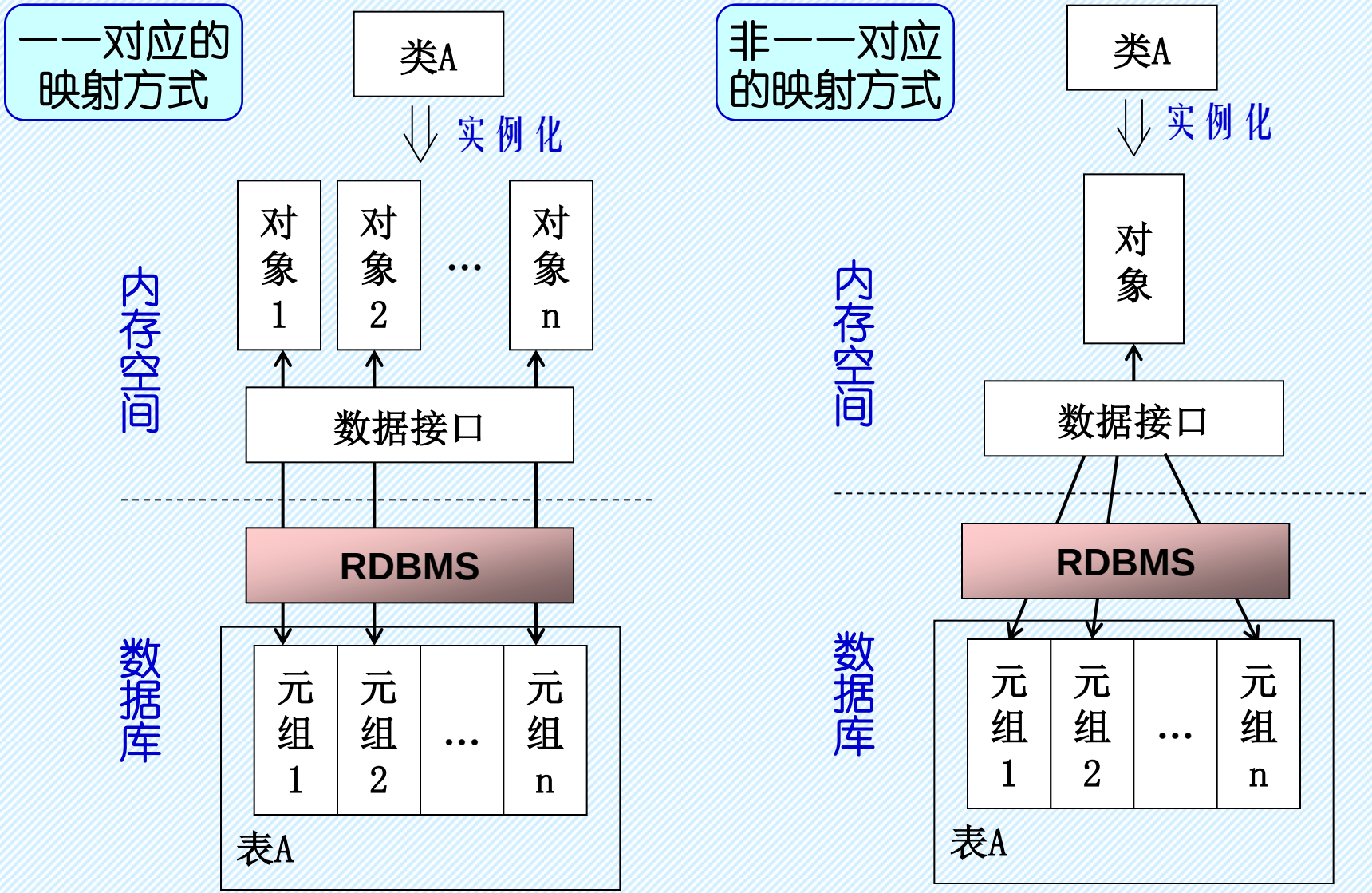
如何看待用 **RDBMS**存储对象



应用系统仍然是面向对象的

只是用关系数据库存储对象的数据

从应用系统的对象到数据库表元组的不同映射方式

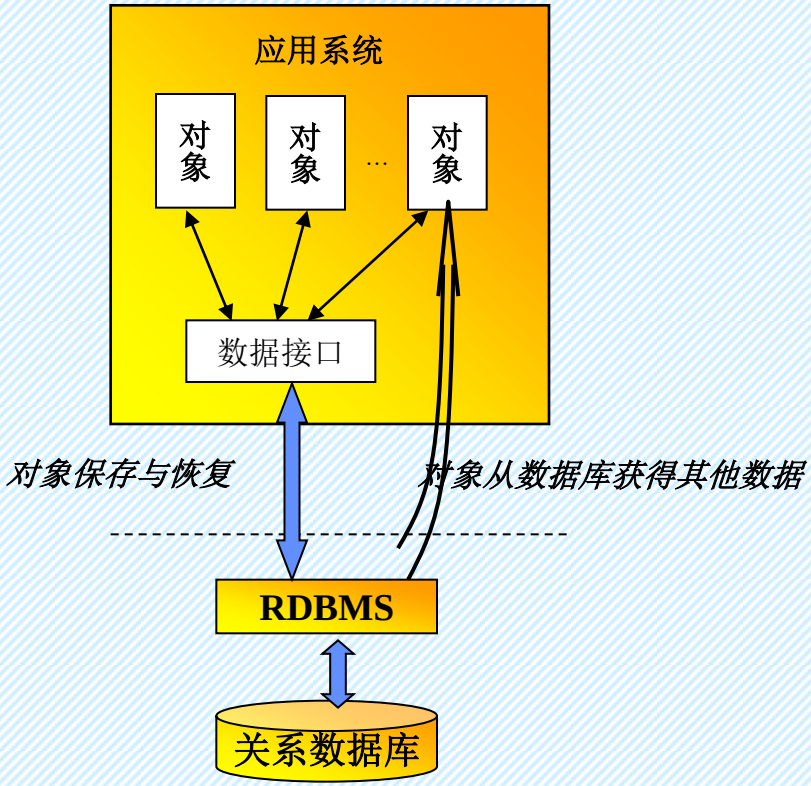


使用RDBMS和使用文件系统的不同

（1）系统以不同方式使用数据库中的数据

存储对象 或 使用其普通数据

——原因：数据库的共享性



（2）可能需要数据格式转换

原因：关系数据库对规范化的要求

2、对象在数据库中的存放策略

对象数据的规范化

修改类图

确定关键字

从类图映射到数据库表

- 类→表
- 类的属性→表的属性
- 对象实例→行
- 对一般-特殊结构、整体-部分结构、关联等OO概念的处理

1）对象数据的规范化

关系数据库要求存入其中的数据符合一定的规范，并且用范式衡量规范化程度的高低。

第一范式（1NF）：关系（表）的每个属性都必须是原子的。就是说，关系的每个属性都是单值的，它不再包含内部的数据结构。

是由关系数据模型决定的，是对一个关系的起码要求

第二范式（2NF）：如果一个关系的所有非关键字属性都只能依赖整个关键字（而不是依赖关键字的一部分属性），则该关系在第二范式中。

第三范式（3NF）：如果一个关系在第二范式中，而且没有传递依赖，则该关系在第三范式中。

Boyce-Codd范式（BCNF）：如果一个关系的每个决定因素都是候选关键字，则该关系在BCNF中。

主要为了解决关系中的函数依赖带来的更新异常问题。也可以减少数据冗余，但更新异常问题更为重要

第四范式（4NF）：如果一个关系在BCNF中，而且没有多值依赖，则该关系在第四范式中。

主要为了减少数据冗余

未必规范化程度越高越好

规范化的代价

——响系统的可理解性，增加了多表查询和连接操作

面向对象方法与关系数据库的规范化目标

既有相违的一面，又有相符的一面

对象的数据结构常常连1NF的要求都不能满足

以对象为中心组织数据与操作，恰恰有助于达到第2NF、3NF、BCNF和4NF要求的条件

例如“通信地址”属性

例1：一个不满足3NF的关系及其规范化

摘自：施伯乐等 译，[美] David M. Kroenke 著. 数据库处理. 北京：电子工业出版社，1998

HOUSING（SID，Building，Fee）
Key:SID
Functional Dependencies:
Building→Fee
SID→Building→Fee

SID	Building	Fee
100	Randolph	1200
150	Ingersoll	1100
200	Randolph	1200
250	Pitkin	1100
300	Randolph	1200

STU-HOUSING（SID，Building）
Key:SID

SID	Building
100	Randolph
150	Ingersoll
200	Randolph
250	Pitkin
300	Randolph

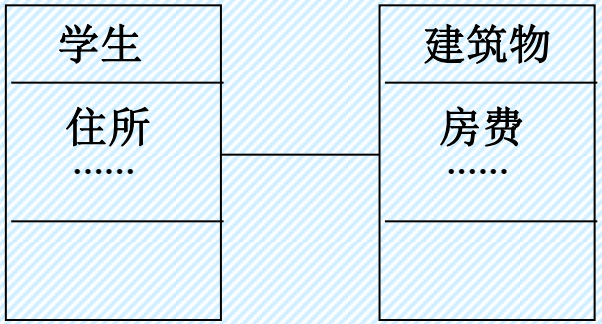
BLDG-FEE（Building，Fee）
Key:Building

Building	Fee
Randolph	1200
Ingersoll	1100
Pitkin	1100

分析原因：把描述不同事物的数据组织在一起

从面向对象的观点看应该定义“学生”和“建筑物”两个类

——恰好与规范化要求吻合



例2：一个不满足4NF的关系及其规范化

STUDENT （ SID， Major，Activity）
Key:（SID，Major，Activity）

SID	Major	Activity
100	Music	Swimming
100	Accounting	Swimming
100	Music	Tennis
100	Accounting	Tennis
150	Math	Jogging

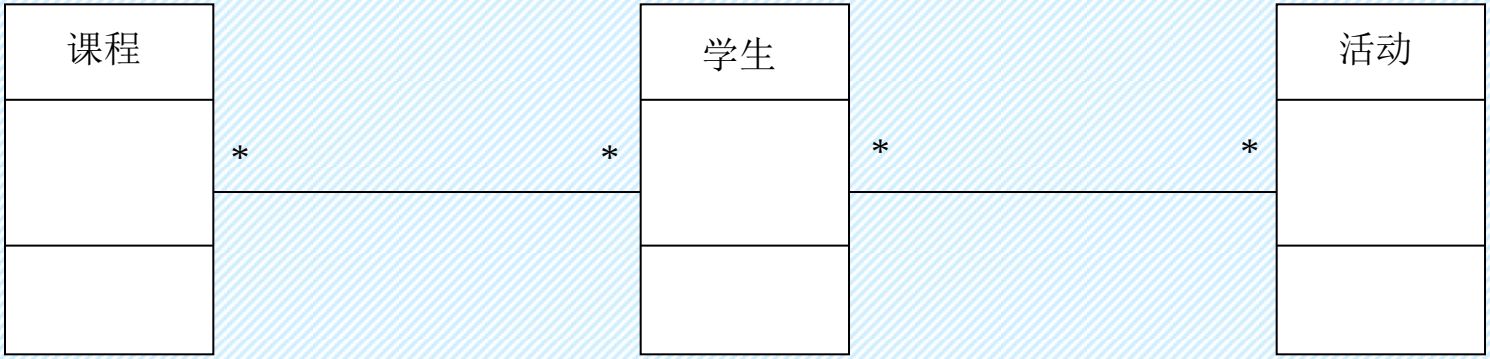
STU-MAJOR （ SID，Major）
Key:（SID，Major）

SID	Major
100	Music
100	Accounting
150	Math

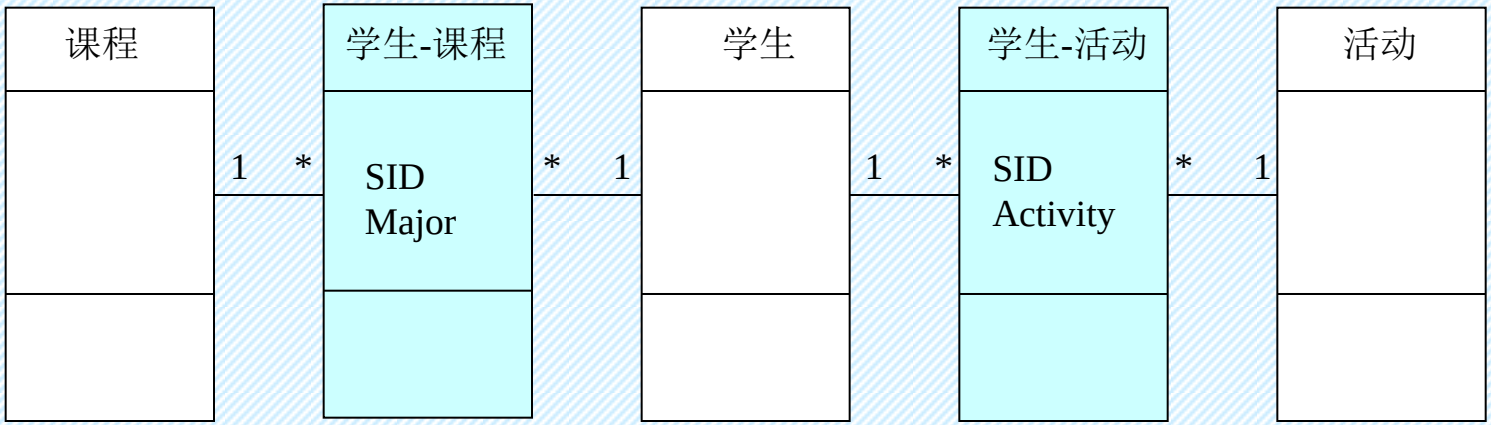
STU-ACT（SID，Activity）
Key:（SID，Activity）

SID	Activity
100	Skiing
100	Swimming
100	Tennis
150	Jogging

用面向对象方法得到的分类——有三类对象

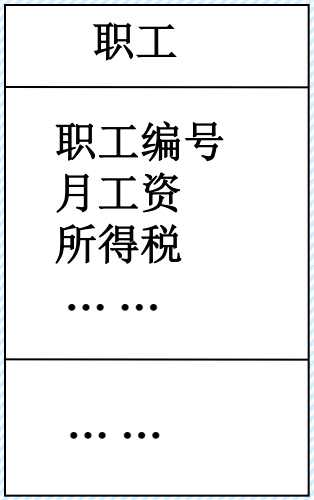


化解多对多关联之后的结果



为什么多出来3个类？

用面向对象方法未能避免函数依赖的例子



可以不通过规范化解决问题——讨论为什么

2）修改类图

规范化的两种策略

保持类图，对表规范化

缺点是对象的存储与恢复必须经过数据格式的转换

修改类图

对问题域的映射可能不像规范化之前那么直接。但是这个问题并不严重 ——利大于弊

采用第二种策略——按规范化的要求修改类图中的类

作为问题域部分的设计内容之一

3）确定关键字

用较少的属性作关键字，将为含关键字的操作带来方便

常用的技巧——引入编号

最终效果：

经过必要的规范化处理和关键字处理之后，得到一个符合数据库设计要求的类图，其中每个需要映射到数据库表的类，都满足如下条件：

至少满足第一范式

满足所期望的更高范式

有一组属性被确定为关键字

4）从类图到数据库的映射

（1）映射策略概要

对每个要在数据库中存储对象实例的类，都建立一个数据库表

类的每个属性（包括从所有祖先继承来的属性）都对应表的一个属性（列）

名称、数据类型完全相同

其中一组属性被确定为关键字

类的每个对象实例将对应表的一个元组（行）

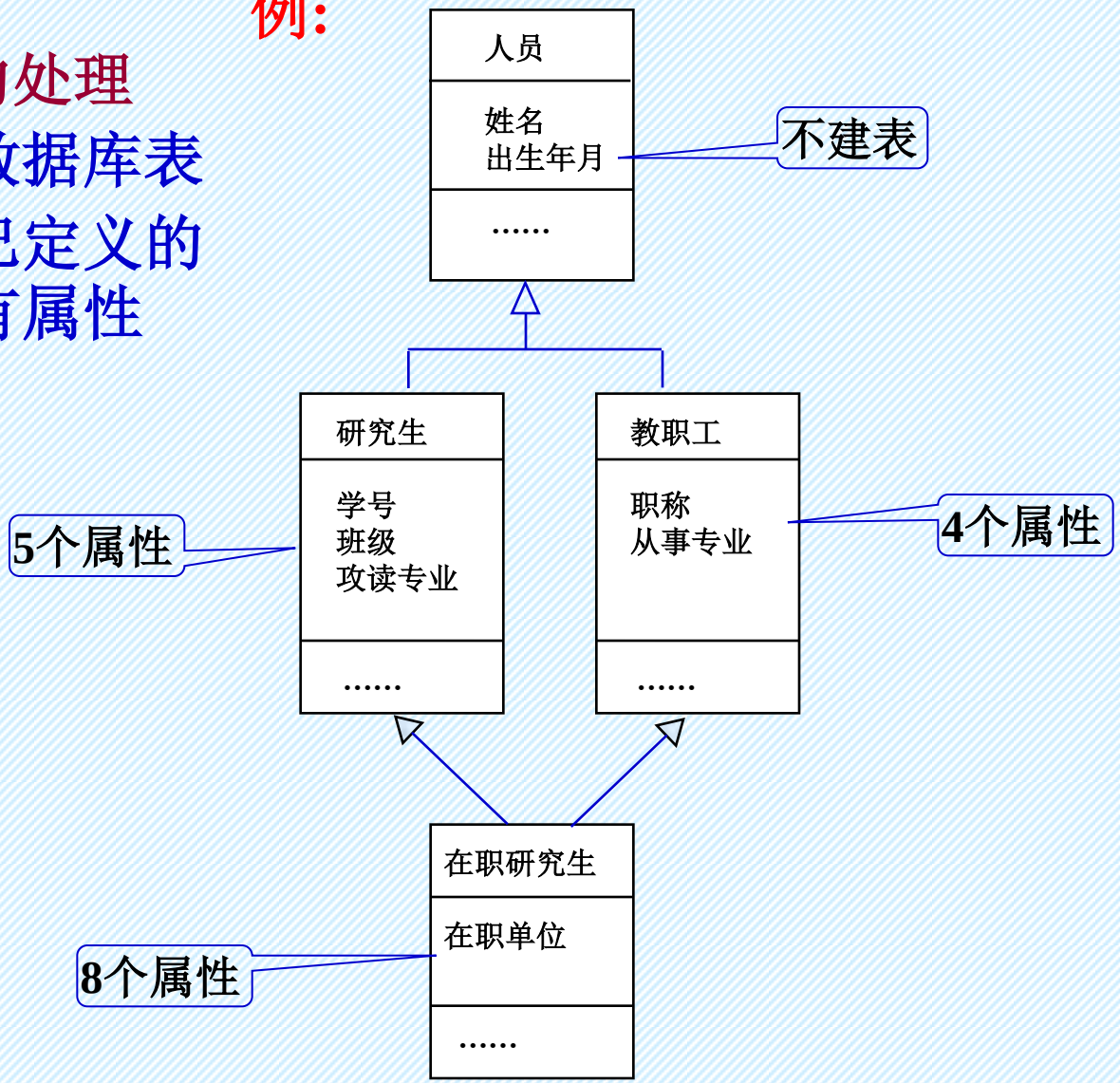
（2）对OO概念的处理

例：

对一般-特殊结构的处理

抽象类不对应数据库表

特殊类包括自己定义的和继承来的所有属性



对关联的处理

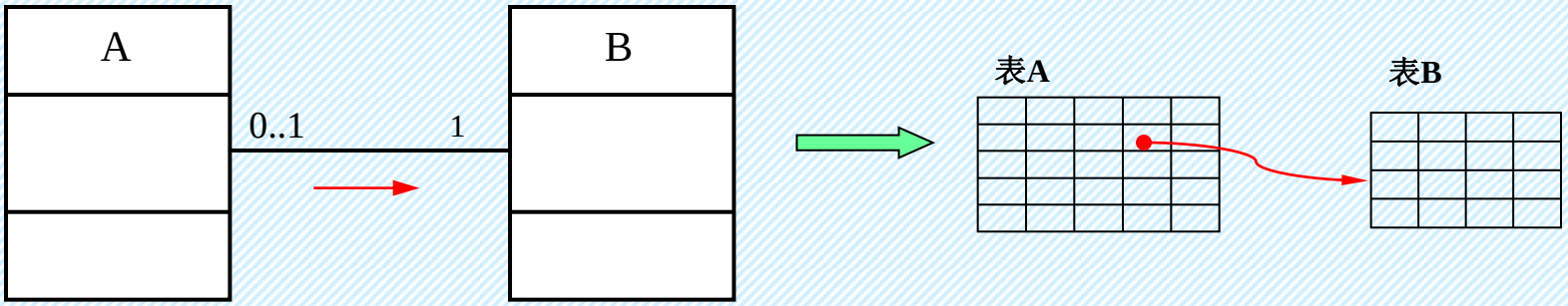
在关联连接线一端的类中定义一个（或一组）属性，表明另一端类的哪个对象实例与本端的对象实例相关联

该属性（属性组）应该和另一端的关键字相同

如果另一端的关键字包含多个属性，本端也要定义同样的多个属性

在对应的数据库表中，一个表以该属性（或属性组）作为外键，另一个表以它作为主键，使前者的元组通过其属性值指向后者的元组

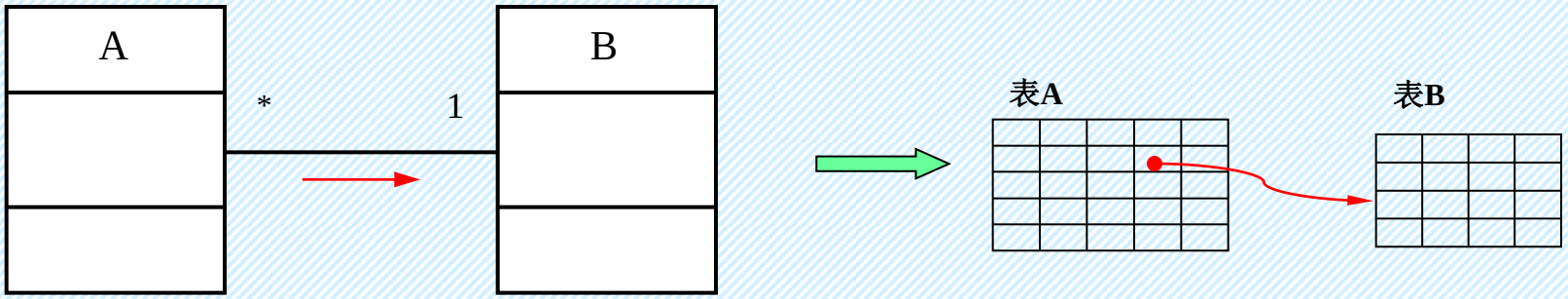
一对一的关联



从B端指向A端，则B表的外键对有些元组可能是空值（NULL）
从A端指向B端则不存在这一问题

一对多的关联

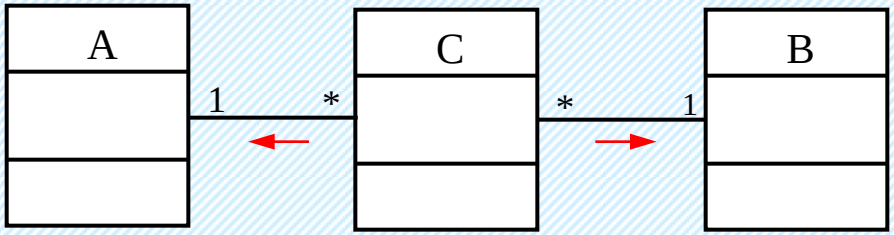
从多重性约束为“m”的一端指向多重性约束为 “1”的一端



映射为数据库表后，
A表以B表的主键作为自己的外键

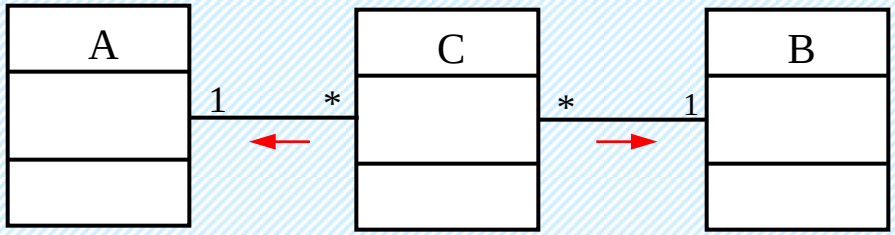
多对多的关联

先在类图中化为两个一对多的关联

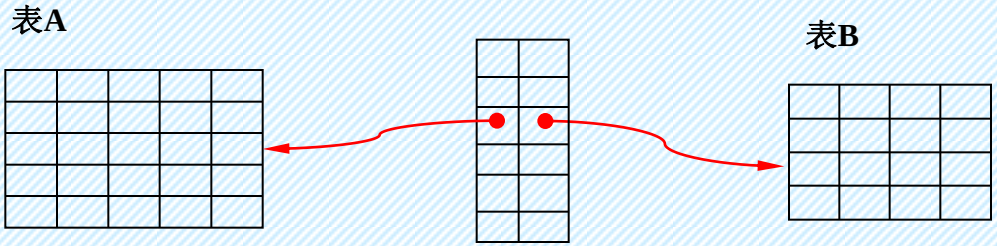


多对多的关联

先在类图中化为两个一对多的关联



然后将每个类映射到一个数据库表



C表含有两个外键，一个是A的主键，一个是B的主键

对象类转化为数据库表的三种情况：

①表中只包含描述本类事物自身特征的属性

②表中既包含描述本类事物自身特征的属性，
也包含作为外键指向另一个表的元组的属性

③表中只包含作为外键指向其它表的元组的属性

对整体-部分结构的处理

分为紧密、固定的方式　和　松散、灵活的方式

二者的区别将通过数据库表的设计体现出来

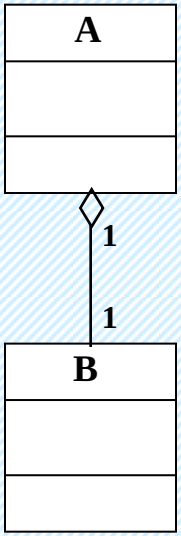
紧密、固定方式：

把部分对象类的属性合并到整体对象类中

松散、灵活方式：

整体对象类和部分对象类分别建立一个表

通过外键表现整体部分关系



(a)

紧密方式：

B的属性合并到A

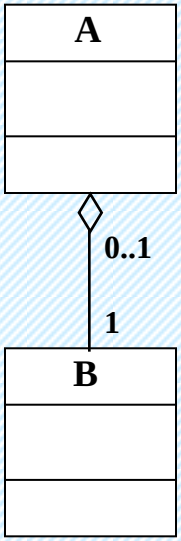
建立A表

松散方式：

建立A、B两个表

A指向B

或者B指向A



(b)

紧密方式：

B的属性合并到A

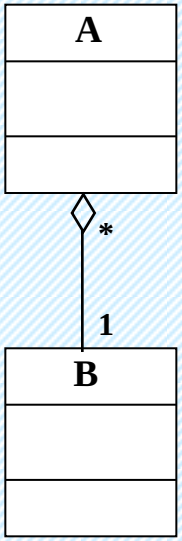
建立A表

还要建立B 表

松散方式：

建立A、B两个表

A指向B

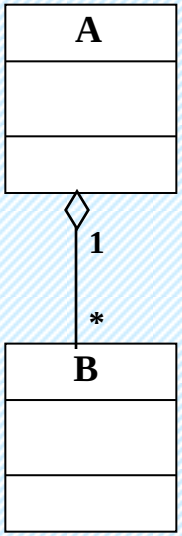


(c)

松散方式：

建立A、B两个表

A指向B

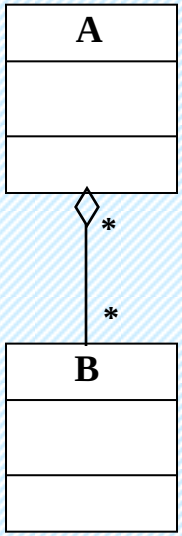


(d)

松散方式：

建立A、B两个表

B指向A



(e)

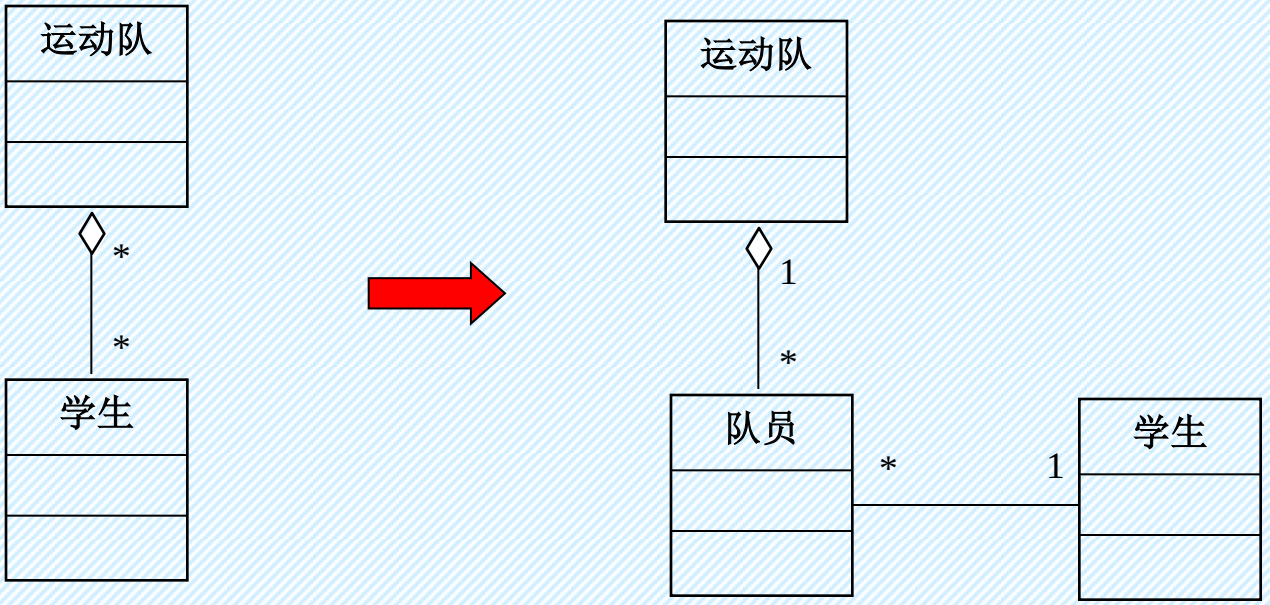
松散方式：

参考多对多关联解决办法

首先解决多对多问题

然后建立A、B两个表以及新增类的表

例：多对多的整体-部分结构的转化



3、数据接口部分类的设计

设计一个名为“对象存取器”的对象类，它提供两种操作

“对象保存”

将内存中一个对象保存到相应的数据库表中

“对象恢复”

从数据库表中找到对象所对应的元组，把它恢复成内存中的对象

执行这些操作需要知道对象的下述信息：

- 它在内存中是哪个对象

  为了知道从何处取得对象数据

  或者把数据恢复到何处

- 它属于哪个类

  为了知道该对象应保存在哪个数据库表中

- 它的关键字

  为了知道该对象对应数据库表的哪个元组

第一种设计方案

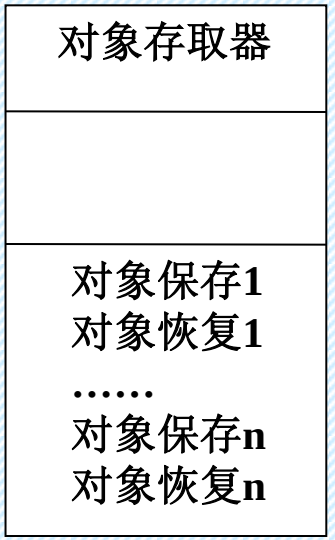
对每个要求保存和恢复的对象类，分别设计一个“对象保存”操作和一个“对象恢复”操作

每个操作只负责一类对象的存或取
因此下述信息是确定的：

- 对象存放在哪个数据库表
- 关键字所包含的属性数目与名称

在操作接口中只需传递如下参数：

- 内存中的一个对象变量
  - 用来提供或接收对象数据
- 对象关键字的值



优点：

- 每个操作都很容易实现，通常只需要一个数据操纵语句，（例如静态SQL语句）

缺点：

- 操作个数太多
- 很难在问题部分采用统一的消息协议

第二种设计方案

只设计一个“对象保存”操作和一个“对象恢复”操作供全系统所有要求保存和恢复的对象类共同使用

操作接口的参数应传送三项信息：

类名

指明被存取的对象属于哪个类

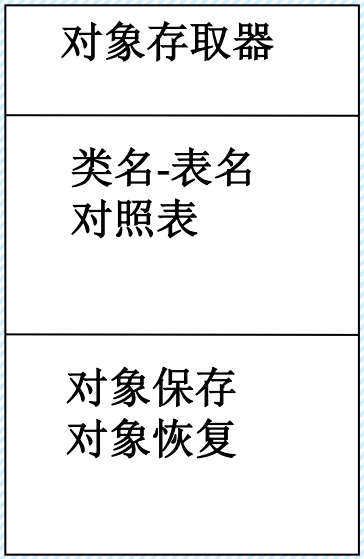
对象变量

提供或接受被存取的对象数据

关键字的值

指明是哪个对象实例要求保存或恢复

数据类型不能静态确定



优点：

操作少，消息协议统一

缺点：

实现难度大

——表的名称、关键字的构成、对象的类型不能在编程时确定

需要编程语言和数据操纵语言提供较强的支持

4、问题域部分的修改

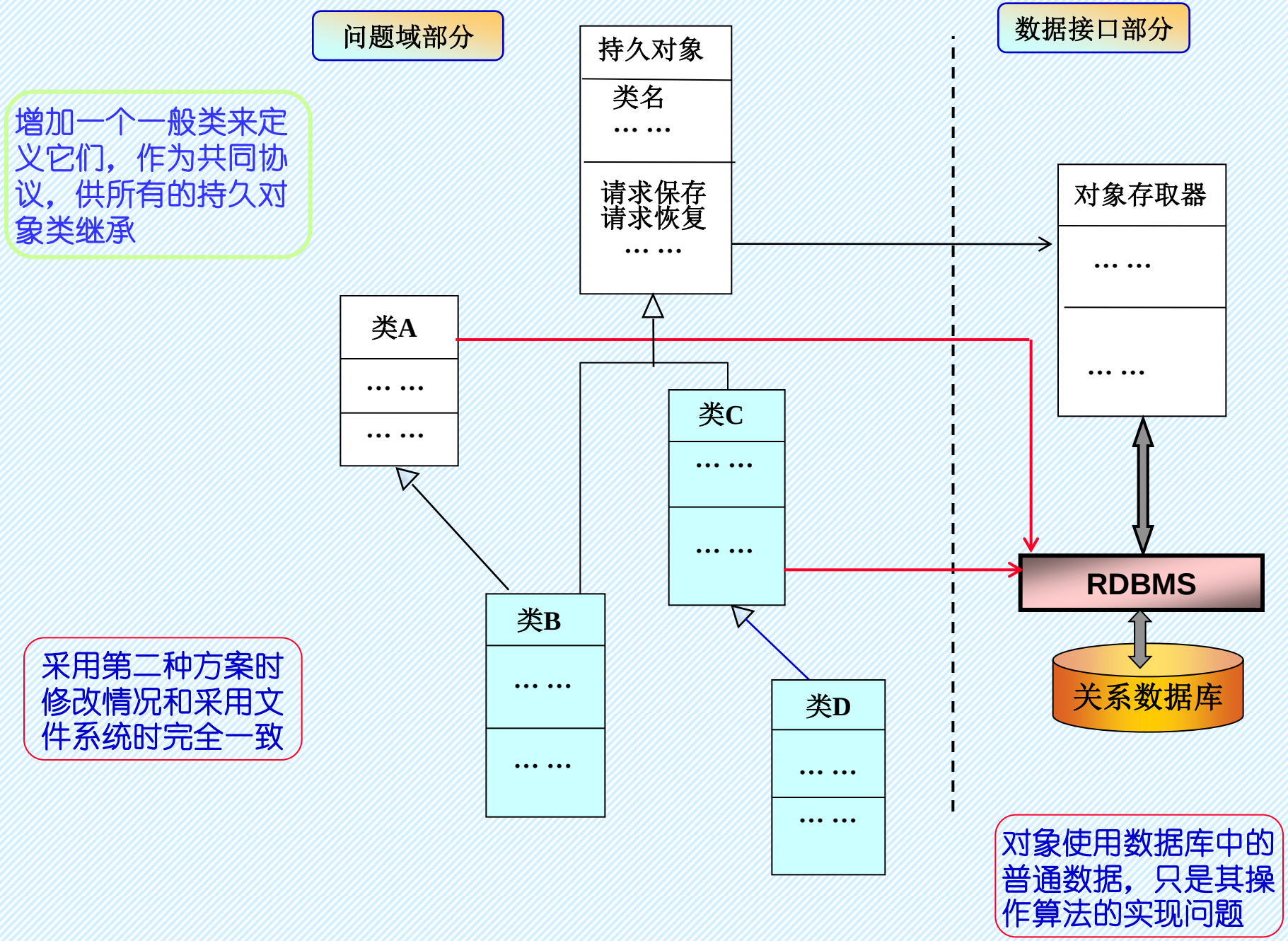
采用第一种方案时

问题域部分每个请求保存或恢复的类，都要使用不同的操作请求语句，这些请求只能分散到各个类中

采用第二种方案时

在问题域部分设计一个高层的类，提供统一的协议，供各个需要在数据库中存储其对象实例的类继承

可以做到和采用文件系统时的处理完全一致


问题域部分
数据接口部分
增加一个一般类来定义它们，作为共同协议，供所有的持久对象类继承
持久对象
类名
… …
请求保存
请求恢复
… …
对象存取器
… …
… …
类A
… …
… …
类C
… …
… …
类B
… …
… …
类D
… …
… …
RDBMS
关系数据库
采用第二种方案时修改情况和采用文件系统时完全一致
对象使用数据库中的普通数据，只是其操作算法的实现问题

（三）针对OODBMS的设计

从应用系统到数据库，从内存空间到外存空间，数据模型都是一致的。因此，几乎不要为此再做更多的设计工作。

类图中的类一般不需要类似于规范化的改造

也不需要专门设计专门负责对象保存与恢复的对象类

主要考虑：

如何用OODBMS提供的数据定义语言、数据操纵语言和其它编程语言来实现OOD模型——实现类和对象的定义和对数据库的访问

必要时要根据语言的功能限制对类图做适当的修改

15.1 面向对象方法与构件技术

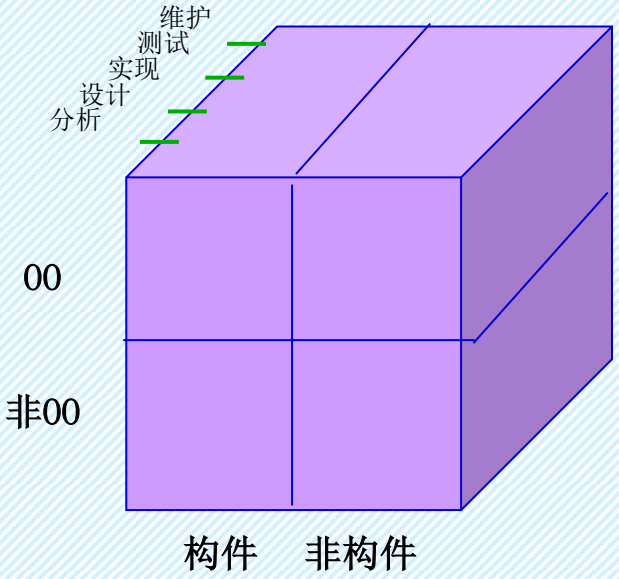
M. Fowler：

“面向对象界一直广为流传的一项争论就是构件和正常类之间的区别为何。”“‘什么是构件’这一问题是一个争论不休的题目。”

“要点是，构件代表可以独立购买与升级的软件片。因此，把一个系统分成若干构件既是一个技术抉择，又是一个销售抉择。”

——Fowler M and Scott K. UML Distilled 3rd edition

“基于构件”与“面向对象”并不是两种相互取代的方法或技术，它们是正交的、互补的关系



用OO方法开发的系统，既可以组织成构件，也可以不组织成构件

构件技术既可以用于OO软件开发
也可以用于非OO的软件开发

面向对象侧重于用什么概念来认识问题域，并把其中的事物以及其关系映射到软件系统中，是一种贯穿软件生命周期的软件方法学。

构件技术的侧重点是如何把系统组织成能够独立地进行生产、组装、复用、部署、发布、销售和升级的产品单位。

构件技术已经发展到软件生命周期的各个阶段——通过领域分析识别领域构件，在系统分析、设计和实现中形成分析级、设计级和实现级的构件，在构件生产线上进行构件的分析、设计和编程。但是这一切并不取代现有的分析与设计方法，正如它并不取代任何一种编程语言一样。

与其他软件工程方法和技术相比，面向对象方法与构件技术之间的配合最为紧密、融洽。

面向对象方法的抽象，继承，封装，聚合，多态等概念与原则对构件技术形成良好的支持。

构件一种比类粒度更大的系统单位。一个构件可以包括多个类，一个类不应该拆散到不同的构件。这意味着，构件的概念并不影响面向对象概念的语法和语义，

15.2 OO模型的构件化

构件化的意义

——支持基于构件的软件开发

OOA阶段：支持分析级的软件复用

OOD阶段：支持和计级的软件复用，并且支持以构件为单位进行编程实现

OOA阶段的构件通常不是最终结果，在OOD阶段会有变化

主要工作

把类图中的类组织成一些可以独立进行编程、发布、销售和升级的构件

基本原则

构件的粒度不宜过小，一个构件通常可以包含多个类，除非某些类本身就已经很庞大。

一个类可以在多个构件中复用，但是不把一个类拆分到多个构件中，即：把类看成一个原子的系统单位。

如何将类组织为构件

考虑的因素：

各个类之间关系的紧密程度
在问题域中所对应的事物
所提供的功能类别
彼此之间通信频繁程度
在系统中的分布与并发情况

这一切正是在面向对象建模中进行包的划分所考虑的因素。

结论：

以包作为组织构件的基本依据
必要时对包进行合并或拆分
兼顾软件的发布、销售等因素

15.3 系统部署

1、软件制品的组织

源文件制品

构件及其接口编程实现后的源文件

可执行文件制品

由源文件编译产生

数据库制品和数据文件制品

按部署的结点打包

模型文件制品

各种模型图及其规约

测试用例制品

按被测试的程序单位进行组织

其他制品

如产品说明书、用户手册、联机帮助文件等

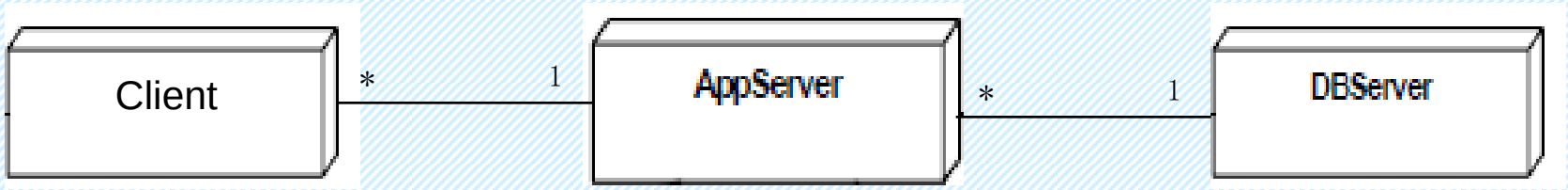
2、系统部署过程与策略

针对不同的目标确定不同的部署方案

例如：针对系统安装、售后服务和最终用户的不同方案

部署过程

（1）描述结点及通信路径　例如：



（2）配置结点的执行环境

操作系统、编译系统、DBMS、界面支持系统、中间件…

标准配置，自选配置

（3）把制品部署到结点上

包的组织策略、系统分布策略和构件组织策略

——决定了各个结点上应该有哪些构件

源文件制品和可执行文件制品

部署到相应的构件所在的结点上

模型文件制品和测试用例制品

根据模型文件和测试用例的作用范围

数据库制品和数据文件制品

根据应用范围和数据传输量较小的原则

产品说明书、用户手册和联机帮助文件等制品

根据使用范围

谢谢大家