

第4章 软件设计

4.1 软件设计的概念

4.2 面向过程的设计

4.3 面向对象的设计

4.1.1 软件设计的概念与设计质量

软件设计的定义

分析

设计

编码

测试

维护

定义

在[IEEE610.12-90]中，软件设计定义为软件系统或组件的架构、构件、接口和其他特性的定义过程及该过程的结果。

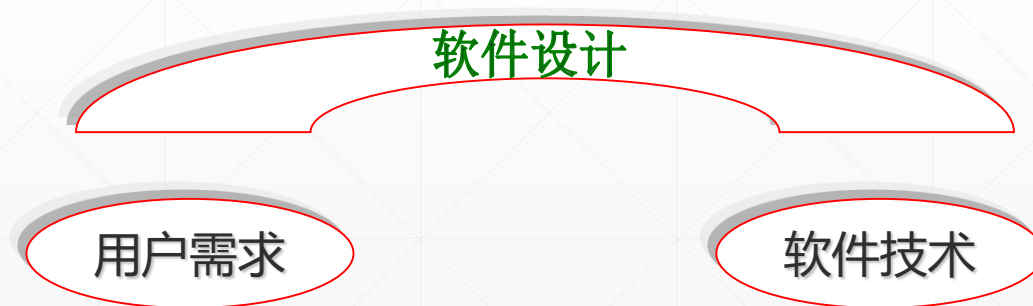
软件设计

软件生命周期
中的一个活动

进行软件编码
的基础

软件需求分析
被转化为软件
的内部结构

是连接用户需
求和软件技术
的桥梁



设计工程活动

软件架构设计（有时称为顶层设计）

- 描述软件的顶层架构和组织，划分不同的组件

软件详细设计

- 详细描述各组件以便能够编码实现

注意：

- 软件设计主要为分解设计D-design(Decomposition design);
- 可以包括系列模式设计FP-design(Family Pattern design);

分解设计：将软件映射为各组件

设计模型

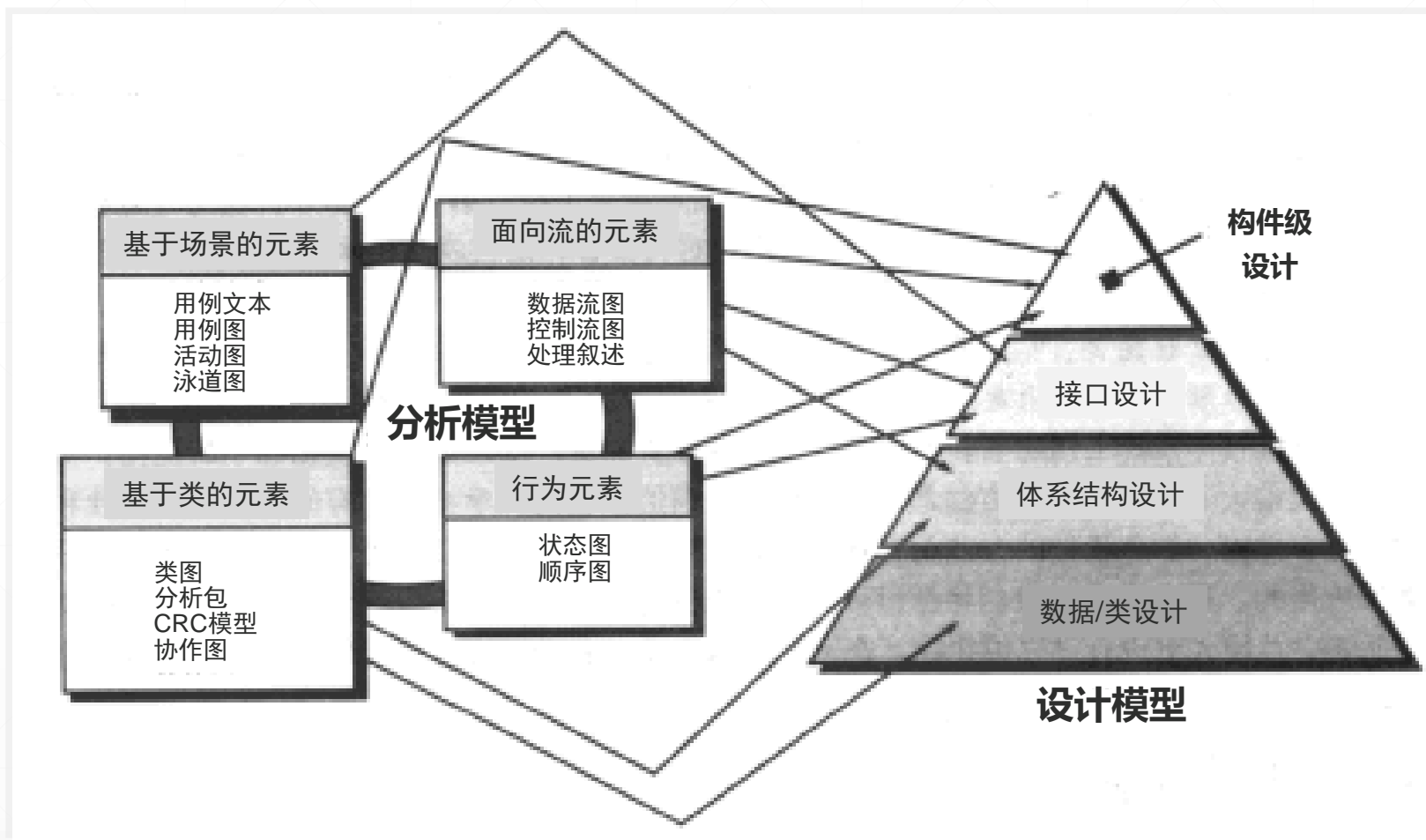
模型输入

- 软件需求的数据模型、功能模型和行为模型

分类

- 数据设计
- 架构设计
- 接口设计
- 组件设计

分析模型到设计模型的转化



好的设计应该具有如下三个特点



设计必须实现在分析模型中包含的所有明确要求，必须满足客户所期望的所有隐含要求；

设计必须对编码人员、测试人员及后续为维护人员是可读可理解的；

设计应提供该软件的完整视图，从实现的角度解决数据、功能及行为等各领域方面的问题

设计质量属性

功能性

易用性

可靠性

性能

可支持性

- 包含三个属性：扩展性、适应性、可维护性

设计指导原则

设计应该是一种架构

设计应该是模块化的

设计应该包含数据、体系结构、接口和组件各个方面

- 应该设计出系统所用的数据结构
- 应该设计出展现独立功能特性的各组件
- 应该设计出各组件与外部环境连接的各接口

设计由软件需求分析过程中获得信息驱动，采用可重复使用的方法导出

设计应该采用正确清楚的表示法

4.1.2 设计相关的八大概念

设计相关概念

抽象

体系结构

设计模式

模块化

信息隐藏

功能独立

精化

重构

概念1. 抽象

含义:

- 是“忽略具体的信息将不同事物看成相同事物的过程”

抽象机制:

- 参数化、规范化

规范化抽象

- 数据抽象: 描述数据对象的冠名数据集合
- 过程抽象: 具有明确和有限功能的指令序列

门

包含属性: 门的类型、转动方向、开门机关、重量和尺寸等

开

一系列过程: 走到门前, 伸出手并抓住把手, 转动把手并拉门, 离开打开的门等

概念2. 体系结构

定义：软件的整体结构和这种结构为系统提供概念上完整性的方式

体系结构设计可以使用大量的一种或多种模型来表达

- 结构模型
- 框架模型
- 动态模型
- 过程模型
- 功能模型

概念3. 设计模式

含义

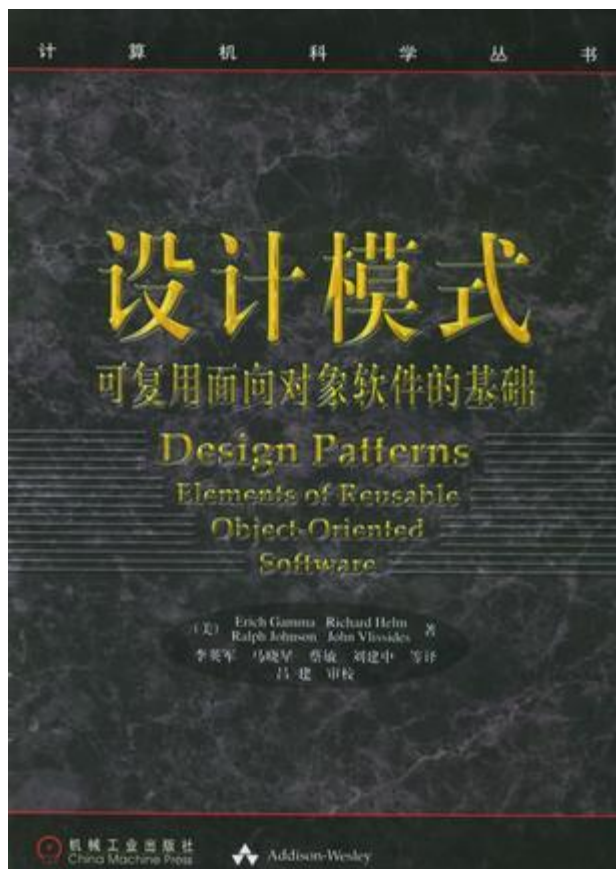
- 在给定上下文环境中一类共同问题的共同解决方案

微观结构

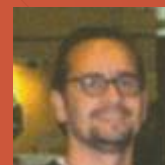
- 实体模式
- 结构模式
- 行为模式

《设计模式：可复用面向对象软件的基础》

- 面向对象的设计中精选出23个设计模式



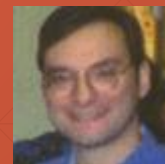
设计模式：可复用面向对象软件的基础



Erich Gamma



Richard Helm

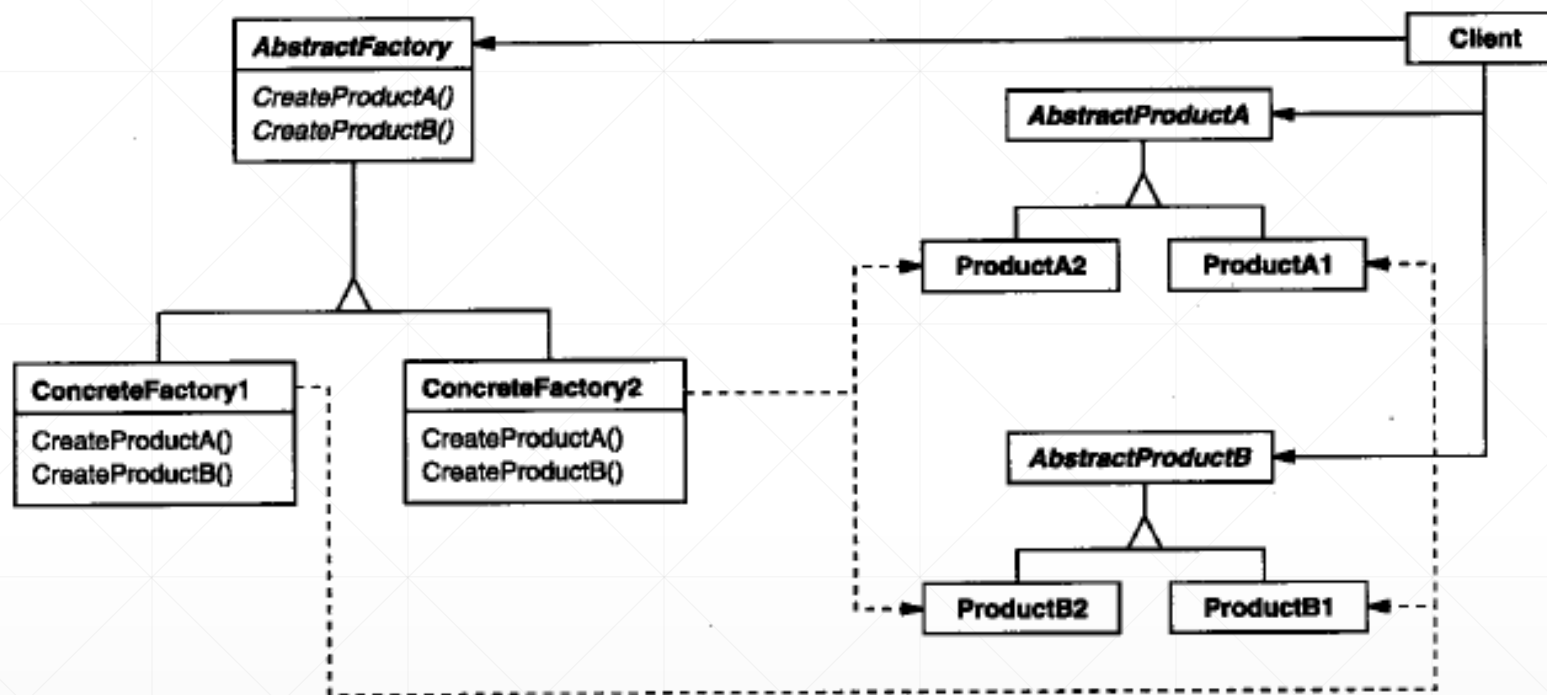


John Vlissides



Ralph Johnson

设计模式举例——抽象工厂



提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

概念4. 模块化

含义

- 软件被划分为命名和功能相对独立的多个组件（通常称为模块），通过这些组件的集成来满足问题的需求

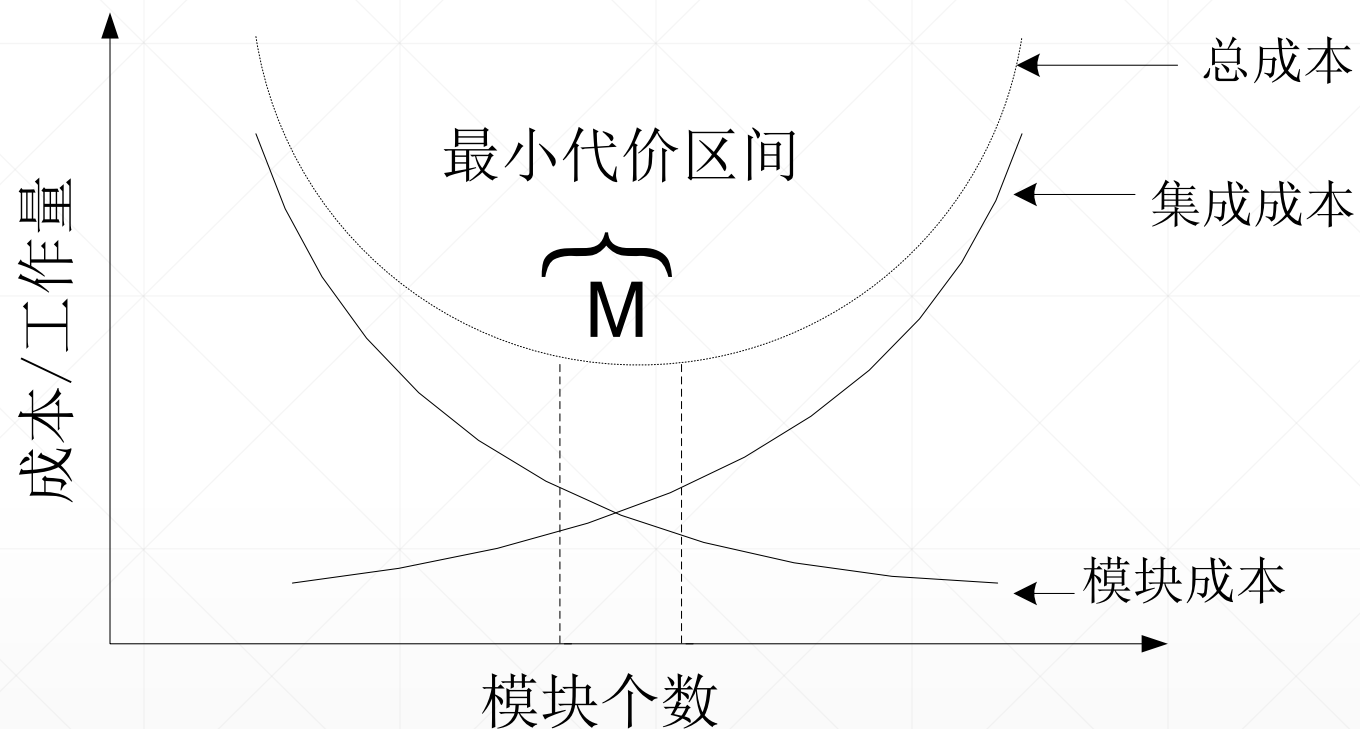
软件的模块性

- 程序可被智能管理的单一属性

模块化的理论依据

- 基于人类解决问题的观测数据

模块化和软件成本



问题：如何确定最小代价区间M??

模块化设计标准

模块化的分解性

- 可分解为子问题

模块化的组合性

- 组装可重用的组件

模块化的可理解性

- 可作为独立单元理解

模块化的连续性

- 需求小变化只影响单个模块

模块化的保护

- 模块内异常只影响自身

概念5. 信息隐藏

模块化基本问题

- 如何分解软件系统以达最佳的模块划分

信息隐藏原则

- 模块应该具有彼此相互隐藏的特性
- 即：模块定义和设计时应当保证模块内的信息（过程和数据）不可以被不需要这些信息的其他模块访问

特点

- 抽象有助于定义构成软件的过程（或信息）实体。
- 信息隐藏原则定义和隐藏了模块内的过程细节和模块内的本地数据结构。

概念6. 功能独立

含义

- 每个模块只负责需求中特定的子功能，并且从程序结构的其他部分看，该模块具有简单的接口

好处

- 易于开发：功能被划分，接口被简化
- 易于维护（和测试）：次生影响有限，错误传递减少，模块重用

定性衡量标准

- 内聚性：模块的功能相对强度
- 耦合性：模块之间的相互依赖程度
- 模块独立性强 = 高内聚低耦合

概念7. 精化

含义

- 逐步求精的过程

与抽象的关系

- 抽象使设计师确定过程和数据，但不局限于底层细节
- 精化有助于设计者在设计过程中揭示底层细节

概念8. 重构

含义

- 不改变组件功能和行为条件下，简化组件设计（或代码）的一种重组技术

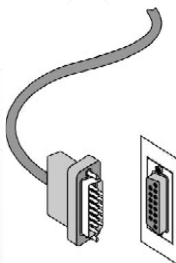
方法

- 检查现有设计的冗余情况、未使用的设计元素、无效或不必要的算法、较差的构建方式或不恰当的数据结构，或任何其他可被更改从而优化设计的问题

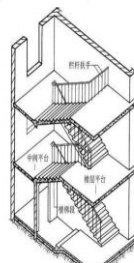
4.1.3 设计技术概要



数据设计



接口设计



架构设计



组件设计

设计技术之一：数据设计

含义

- 数据设计（有时也被称为数据架构）构建高层抽象（客户/用户的数据视图）的数据模型、信息模型

相关概念

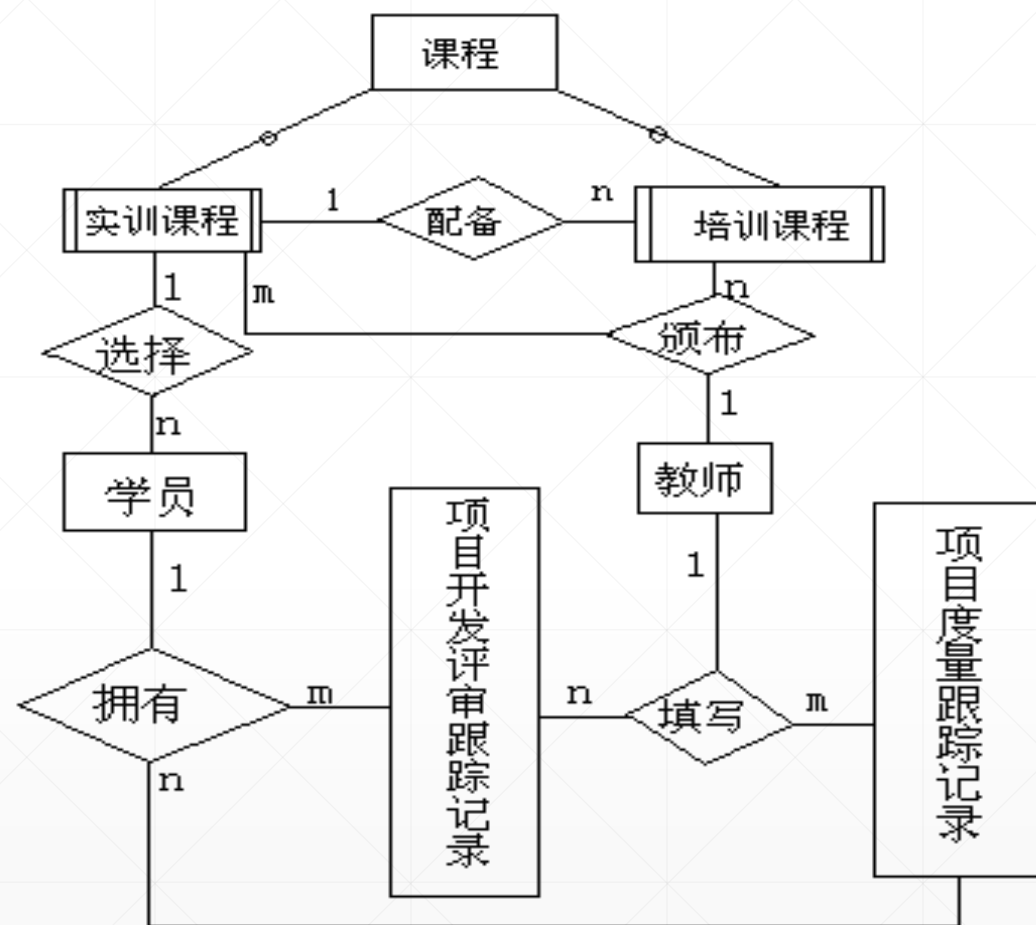
- 数据建模
 - 数据字典、E-R图、类图
- 数据结构
 - 计算机存储、组织数据的方式
- 数据库
 - 按照数据结构来组织、存储和管理数据的仓库
- 数据仓库

组件级别的数据设计

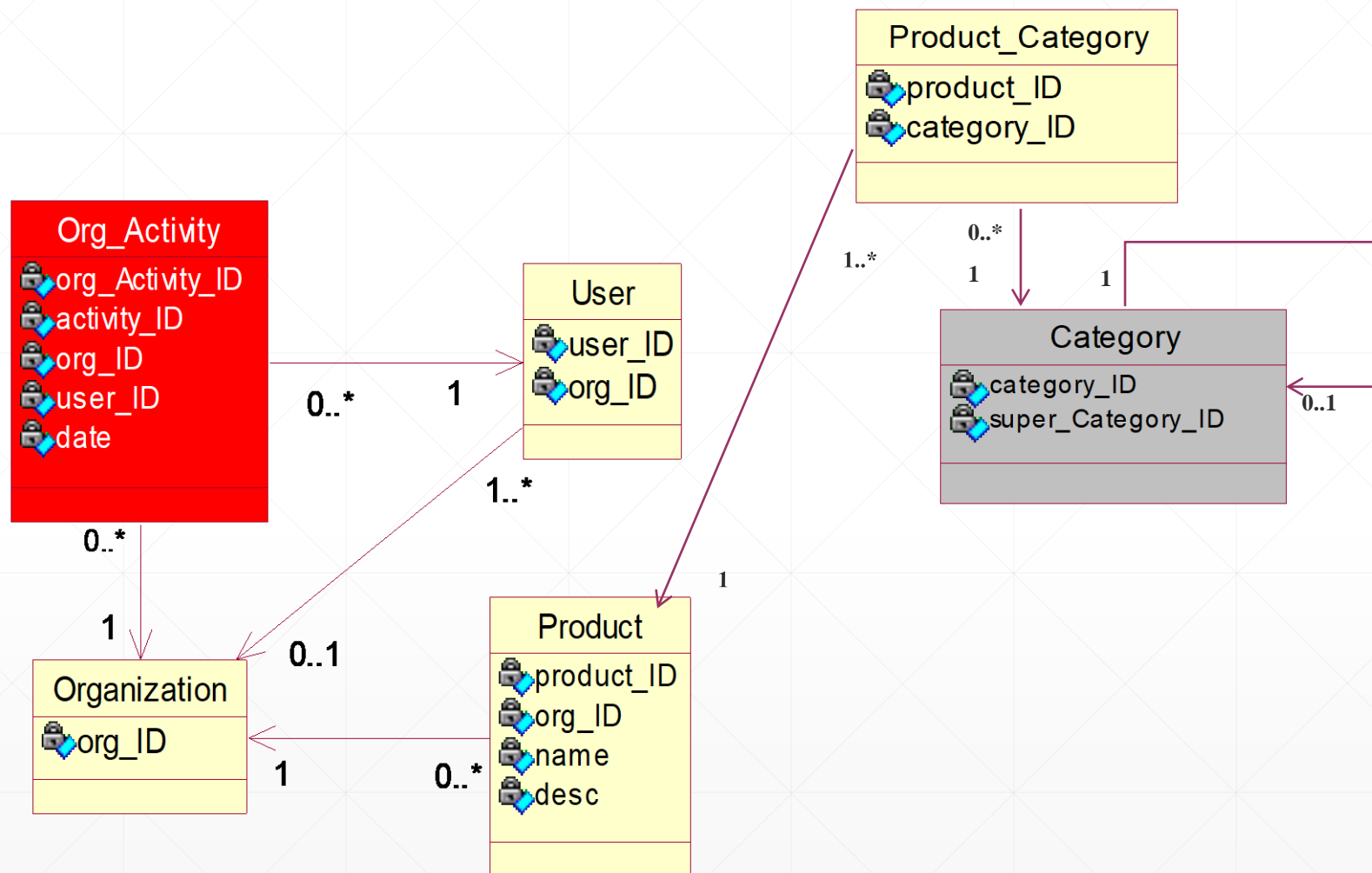
设计原则

- 应用于功能和行为系统分析的原则也应适用于数据设计
- 所有的数据结构及其对应的操作都应该确定
- 建立数据字典并在数据定义和程序设计中应用
- 低层次的数据设计应该推迟到设计的后期过程
- 数据结构的表示应该只对直接使用数据结构中数据的模块可见
- 开发有用的数据结构及其对应操作的程序库
- 软件设计和编程语言应该支持抽象数据类型的定义与实现

概念数据模型



物理数据模型



设计技术之二：体系结构设计



系统需要执行的函数功能组件集（如数据库、计算模块）

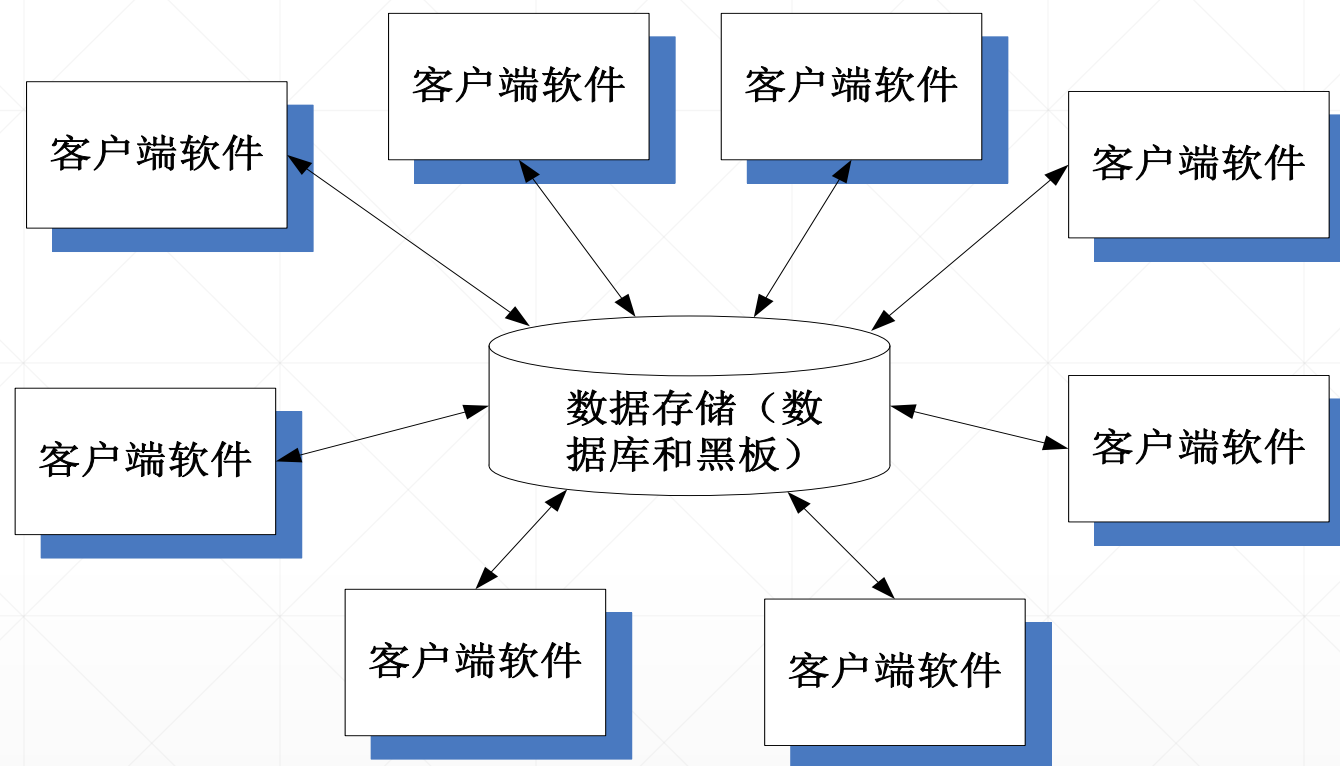
The diagram illustrates the components of system architecture design. It features a vertical line on the left with four white circles. Each circle is connected to a horizontal red bar containing text. The circles are connected to the bars by short vertical lines. The background has a light gray diamond pattern.

组件之间通信、协同和合作的连接

组件集成构成系统的约束

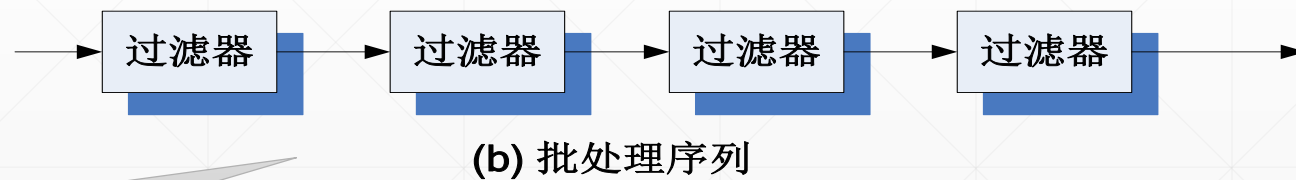
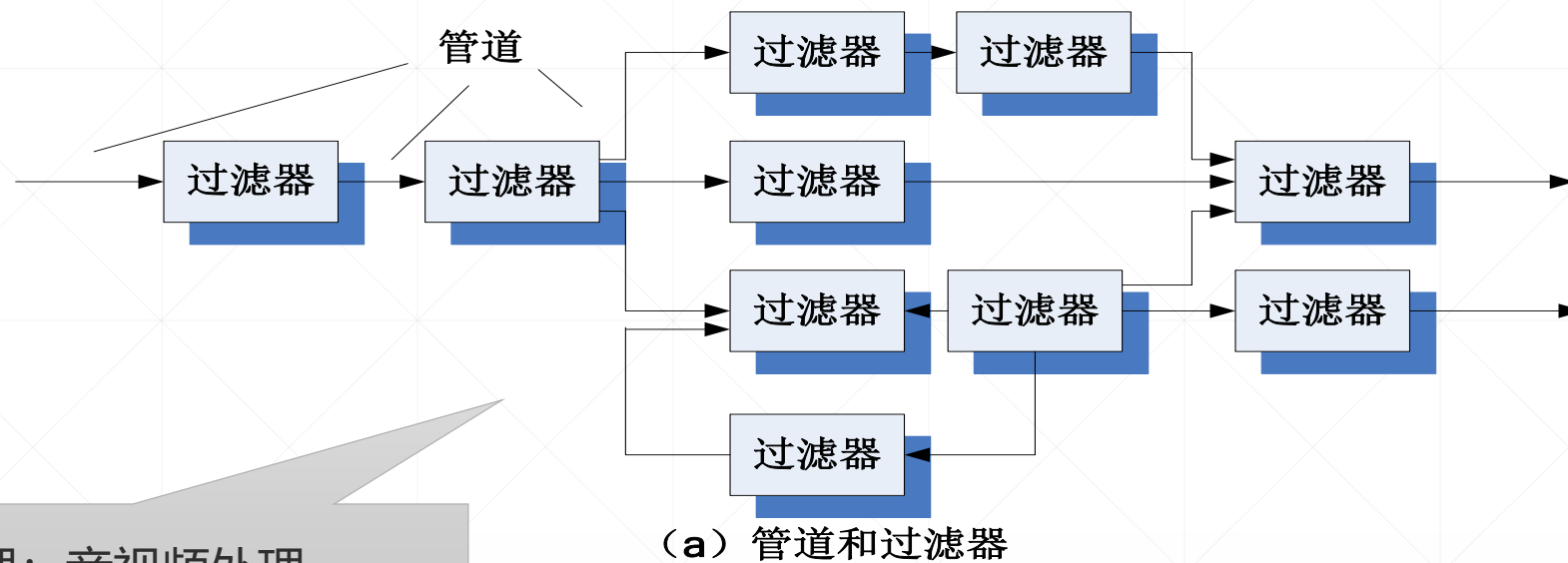
设计人员通过分析系统组成部分的已知特性，理解其整体特性的语义模型分析

风格和模式简要分类 - 数据中心架构



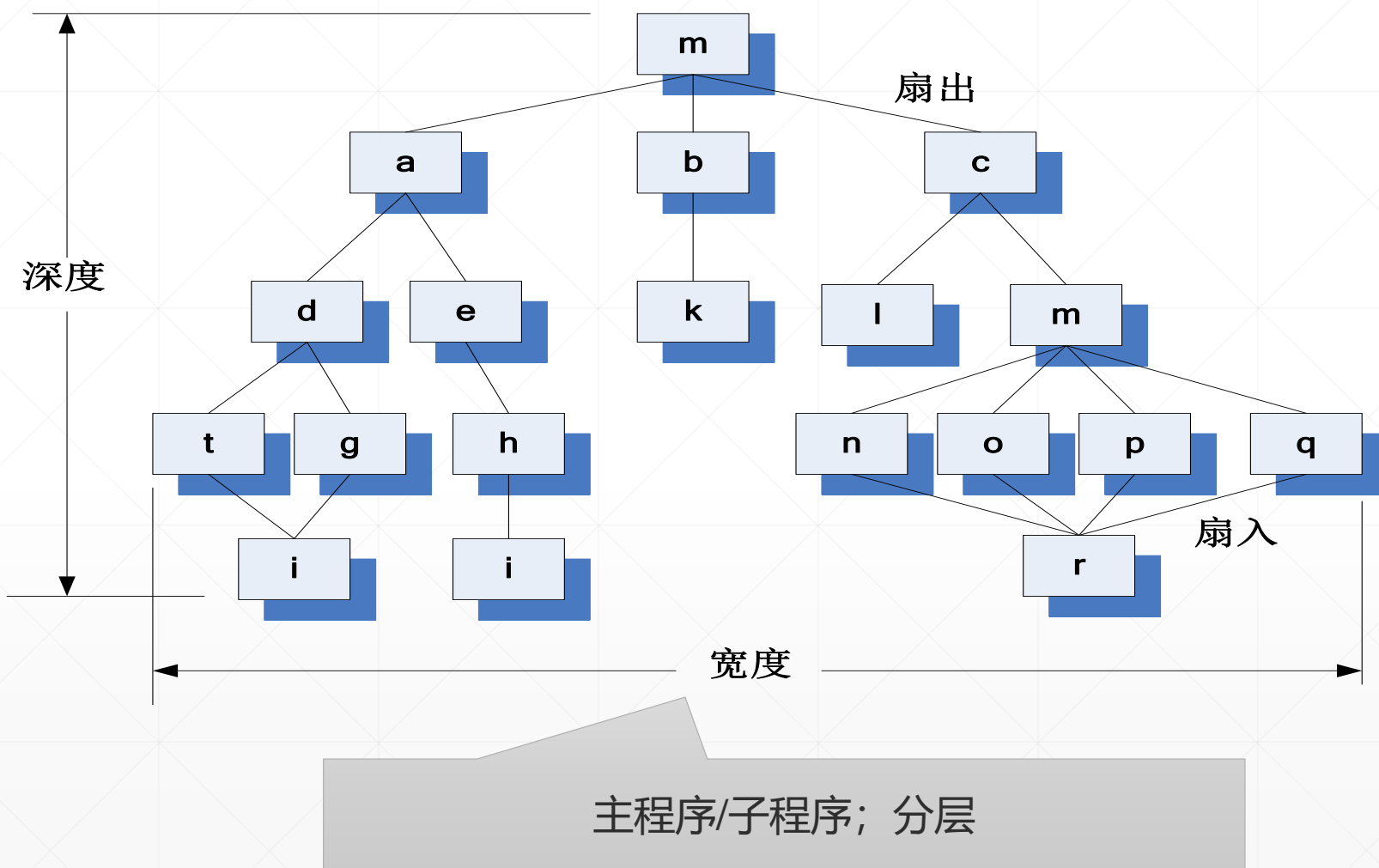
数据库系统；超文本系统；黑板系统

风格和模式简要分类 - 数据流体系架构

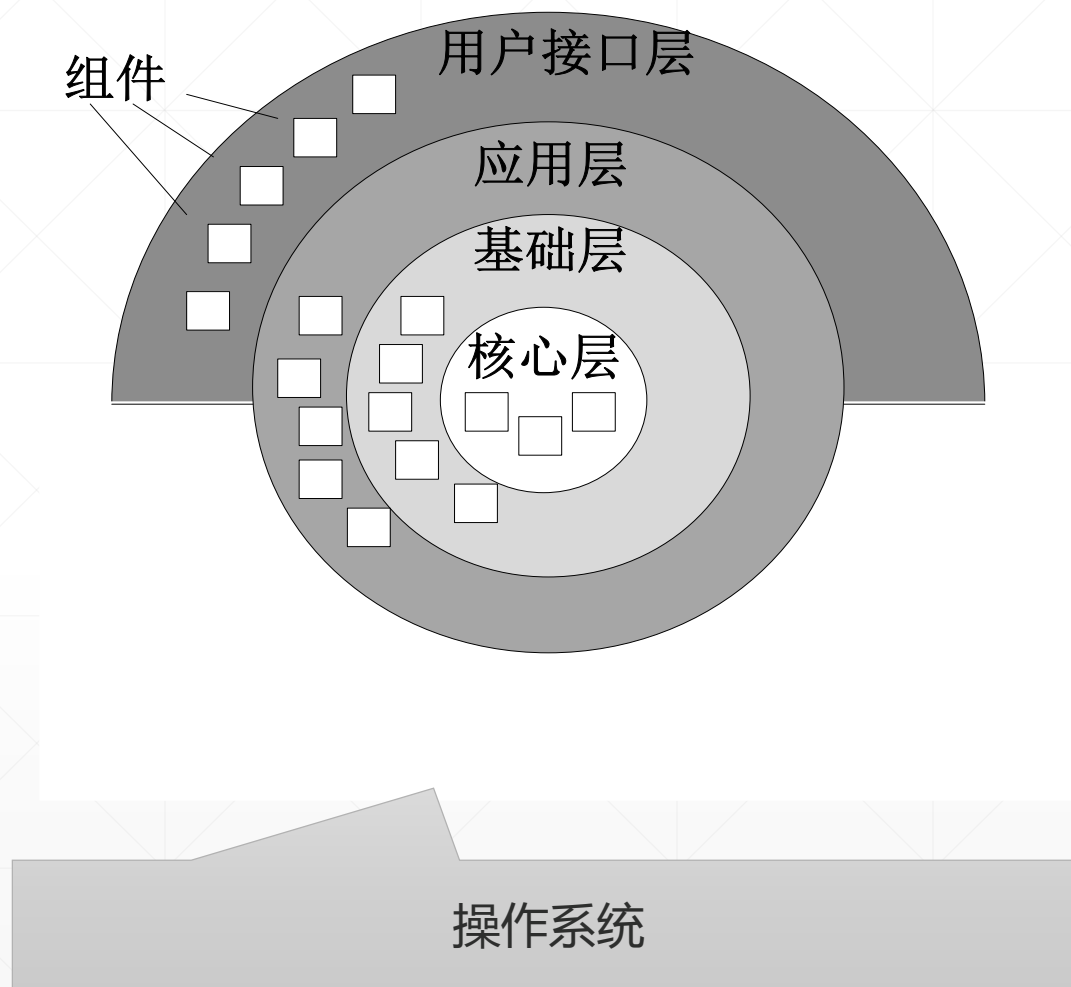


程序编译

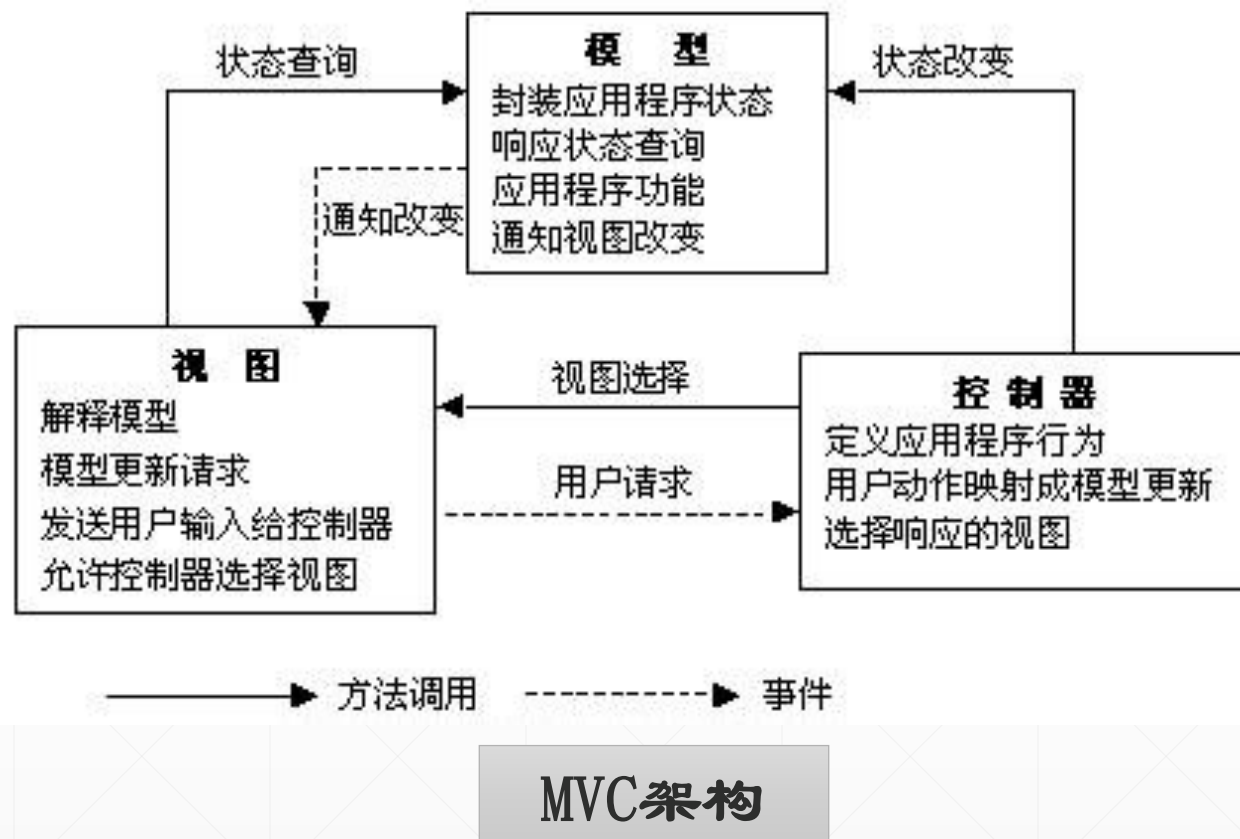
风格和模式简要分类 - 调用和返回架构



风格和模式简要分类 - 层次架构



风格和模式简要分类 - 面向对象架构



系统组件封装数据和处理该数据的操作。
组件之间的通信和协作通过消息传递实现。

两个基本问题

控制结构

- 在架构内部如何实现管理控制？是否有不同的控制架构存在？

数据传递

- 组件之间如何进行数据传递？数据流是否连续，或者传递给系统的数据对象是否零散？

部署设计

以部署环境创建开始，在整个生命周期阶段中处于逻辑设计和技术需求阶段

部署环境包含整个解决方案的逻辑架构和服务质量（QoS）需求

部署架构设计是一个反复迭代的过程，通常需要多次查看QoS要求和多次检查先前的设计，需要考虑了服务质量QoS需求的相互关系，平衡取舍相关问题成本以实现最佳解决方案，最终满足项目的业务目标

部署设计输出



部署设计方法



设计技术之三：接口设计（含界面设计）

高效用户界面设计有三条重要原则：

- 允许用户操作控制（用户为中心）
- 减少用户记忆负担
- 保持界面一致

环境分析确定了用户接口操作的物理结构和社会结构

推荐读物

- 《交互设计——超越人机交互》，电子工业出版社
- 《设计心理学》，唐纳德·A·诺曼
- 《情感化设计》，唐纳德·A·诺曼
- 《可用性工程》，尼尔森

设计技术之四：组件设计

面向过程的组件设计

- 函数与模块的设计

面向对象的组件设计

- 类与操作的设计
-

第4章 软件设计

4.1 软件设计的概念

4.2 面向过程的设计

4.3 面向对象的设计

4.2.1 面向过程的总体设计

结构化的总体设计方法

首先研究、分析和审查数据流图。从软件的需求规格说明中弄清数据流加工的过程，对于发现的问题及时解决。

然后根据数据流图决定问题的类型。数据处理问题典型的类型有两种：变换型和事务型。针对两种不同的类型分别进行分析处理。

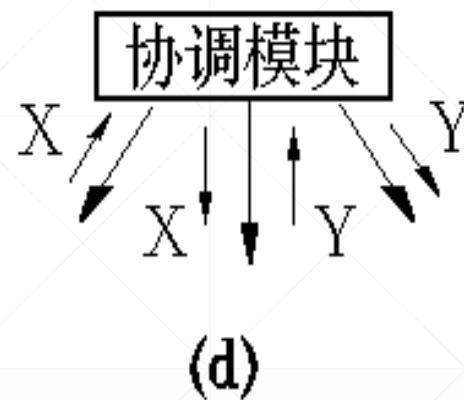
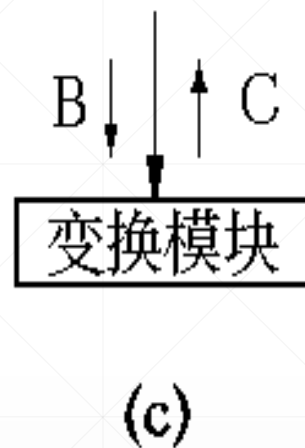
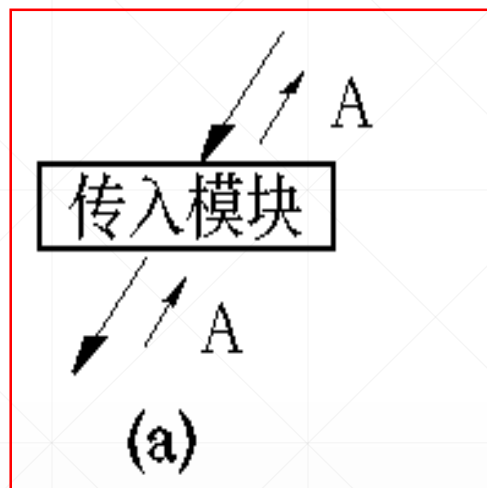
由数据流图推导出系统的初始结构图。

利用一些启发式原则来改进系统的初始结构图，直到得到符合要求的结构图为止。

修改和补充数据词典。

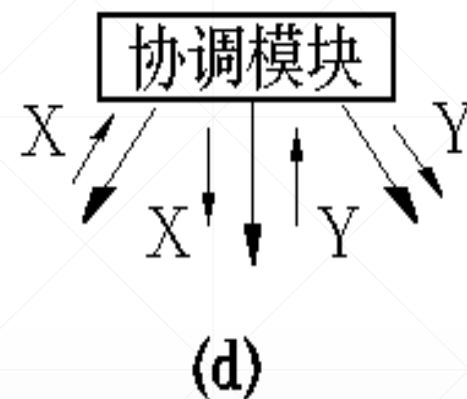
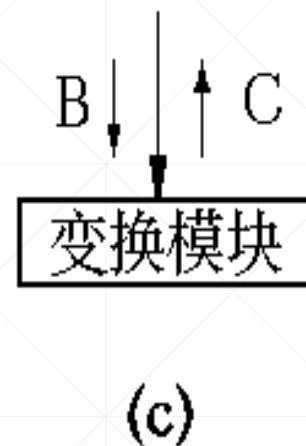
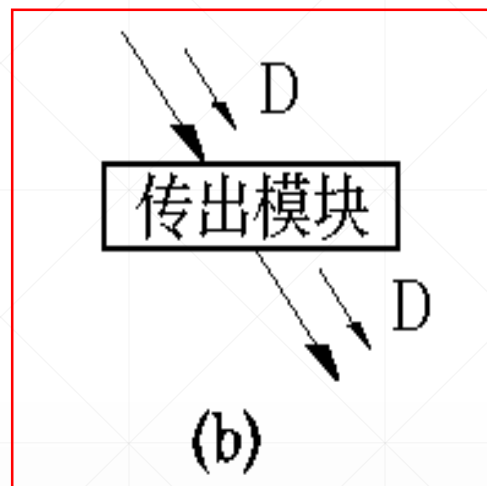
系统结构图

在系统结构图中的模块



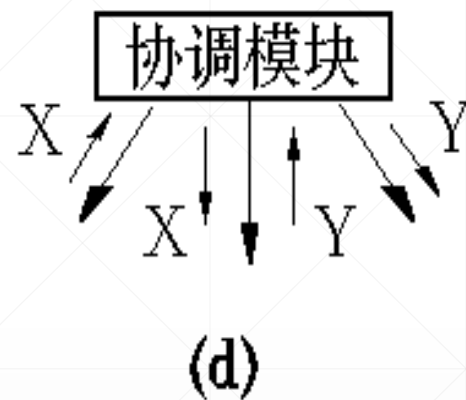
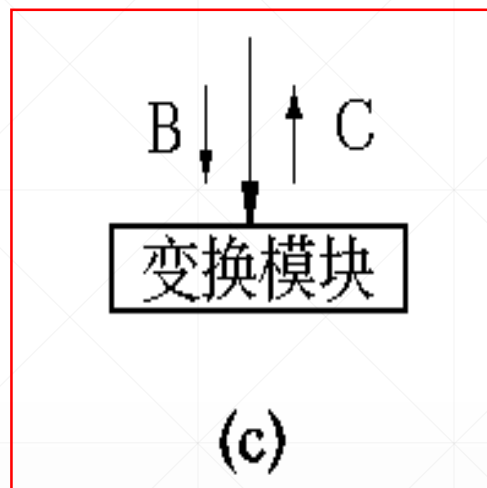
- 传入模块 — 从下属模块取得数据，经过某些处理，再将其传送给上级模块。它传送的数据流叫做逻辑输入数据流。

在系统结构图中的模块



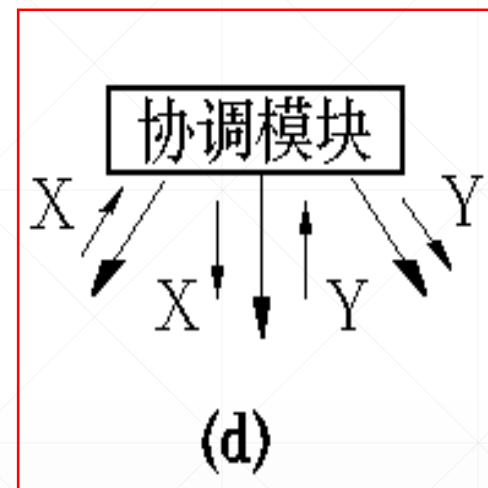
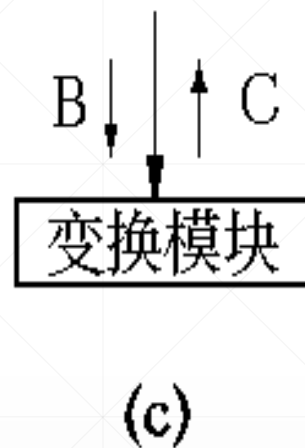
- 传出模块 — 从上级模块获得数据，进行某些处理，再将其传送给下属模块。它传送的数据流叫做逻辑输出数据流。

在系统结构图中的模块



- 变换模块 — 它从上级模块取得数据，进行特定的处理，转换成其它形式，再传送回上级模块。它加工的数据流叫做变换数据流。

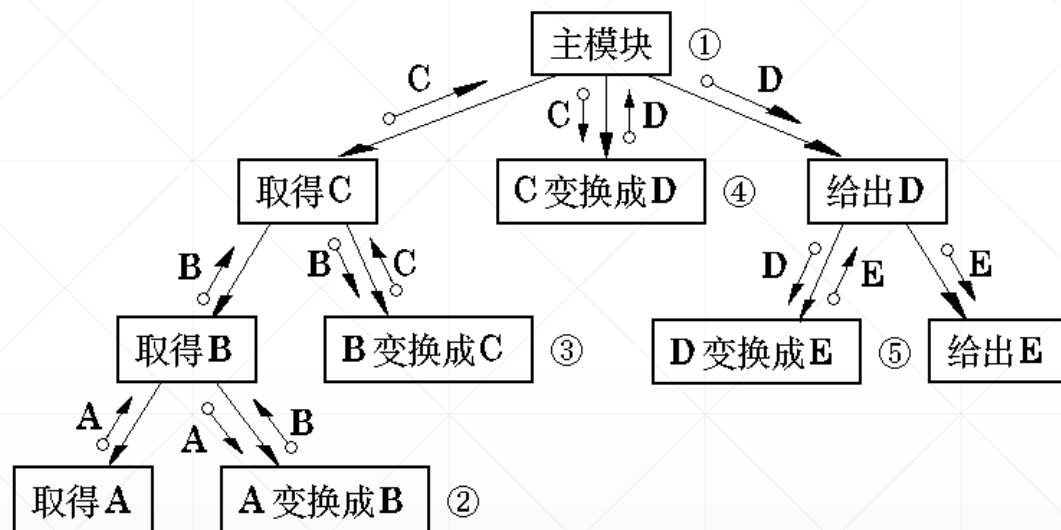
在系统结构图中的模块

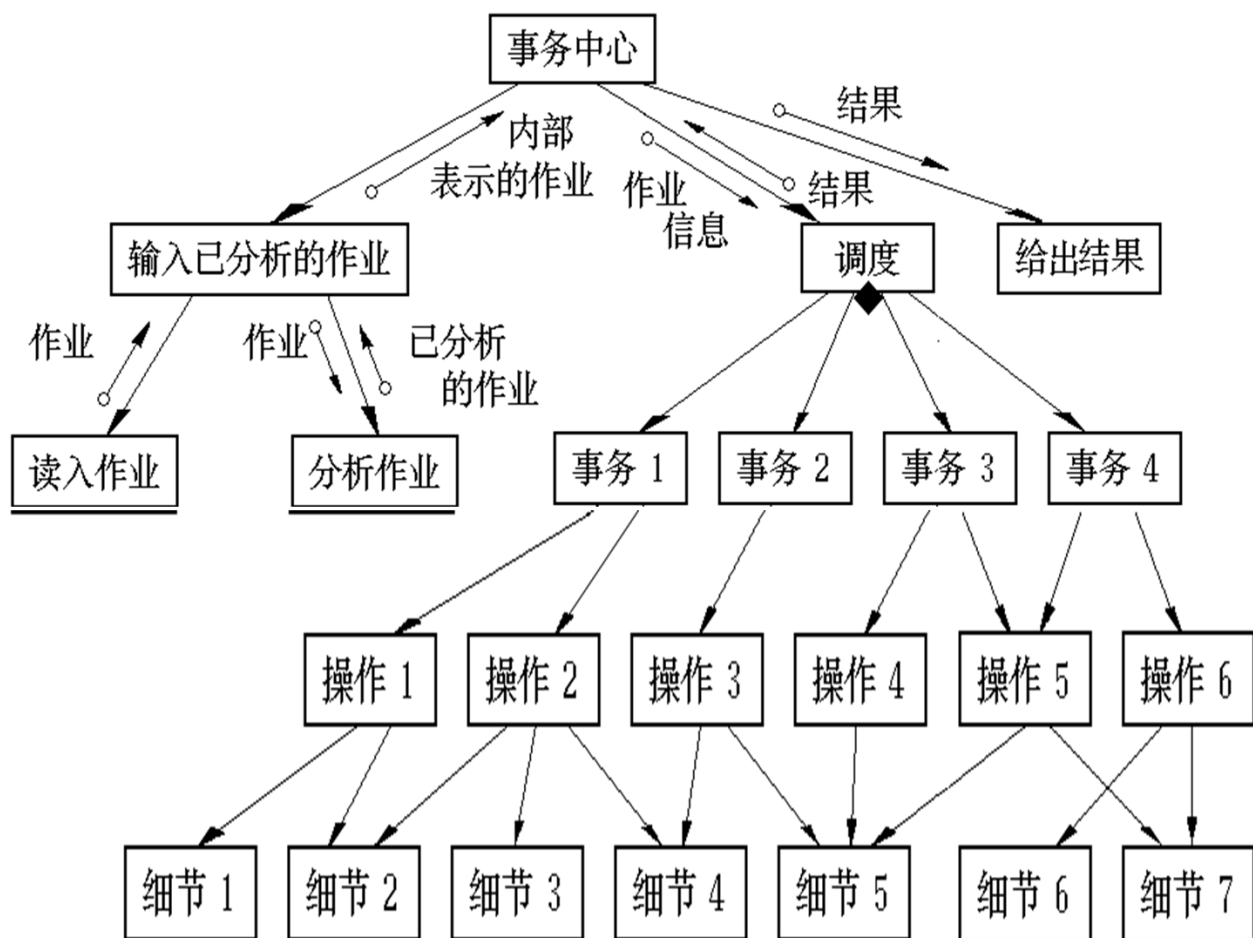


- 协调模块 — 对所有下属模块进行协调和管理的模块。

变换型系统结构图

- 变换型数据处理问题的过程大致分为三步，即取得数据，变换数据和给出数据。
- 相应于取得数据、变换数据、给出数据，变换型系统结构图由输入、中心变换和输出等三部分组成。





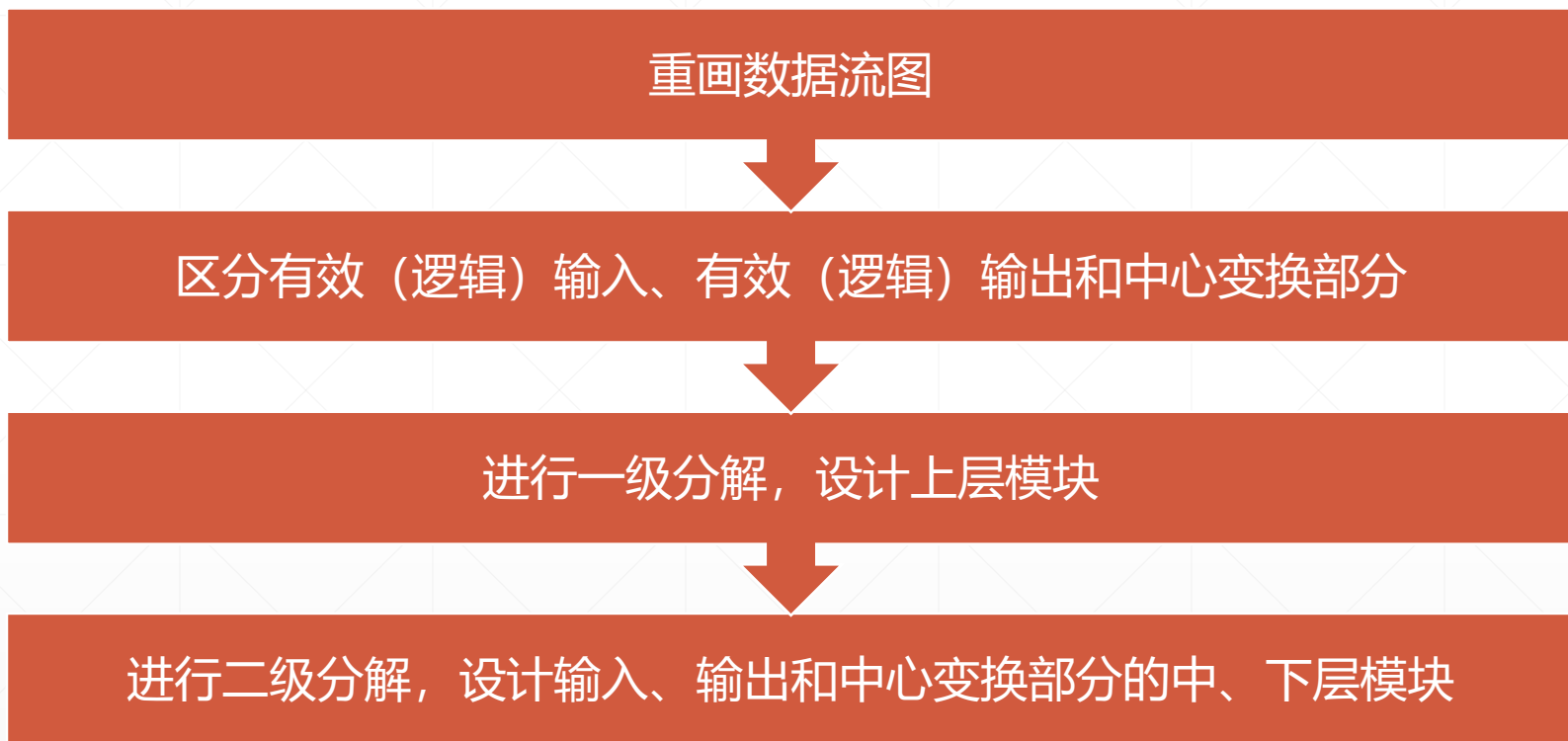
事务型系统结构图

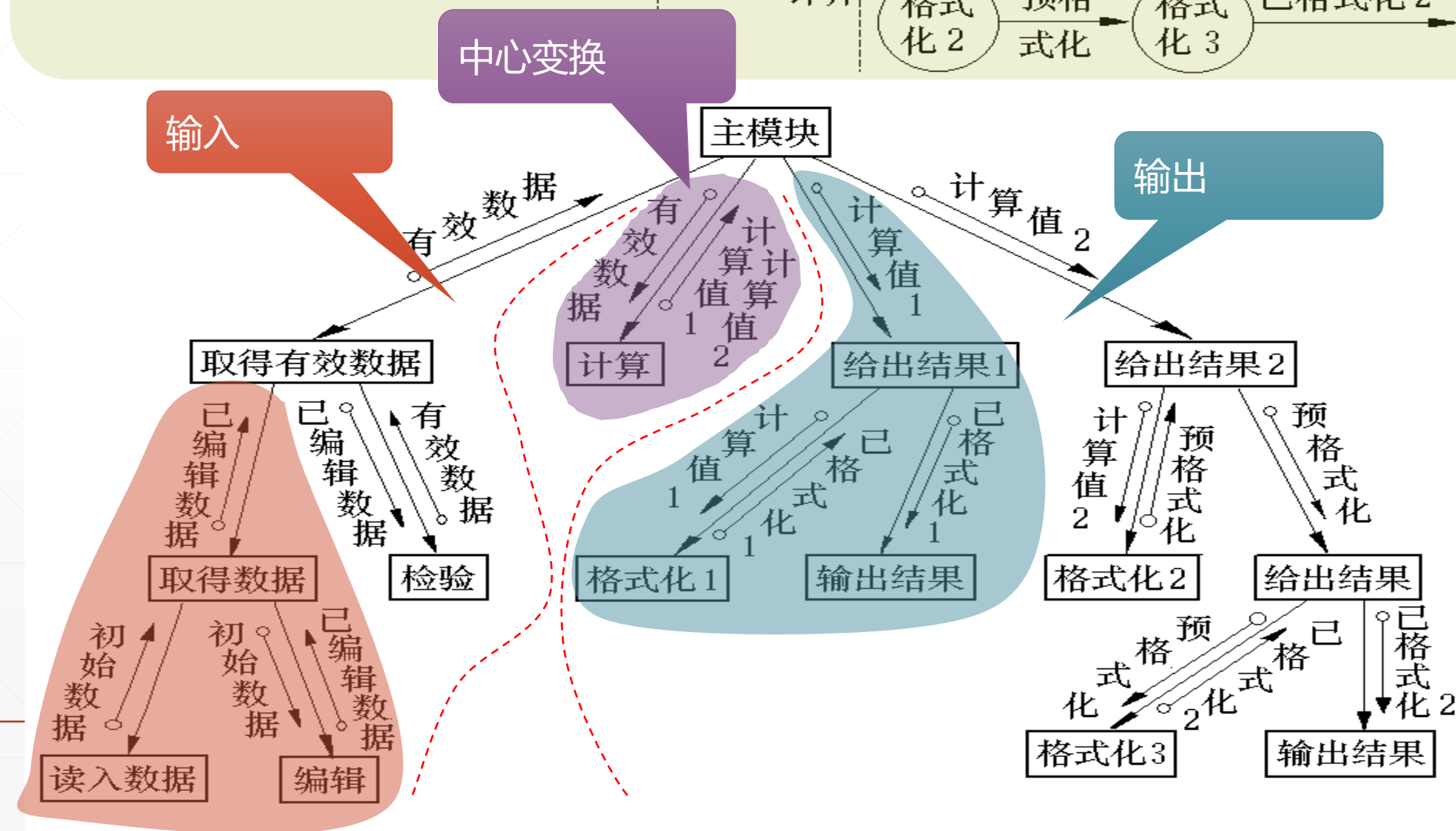
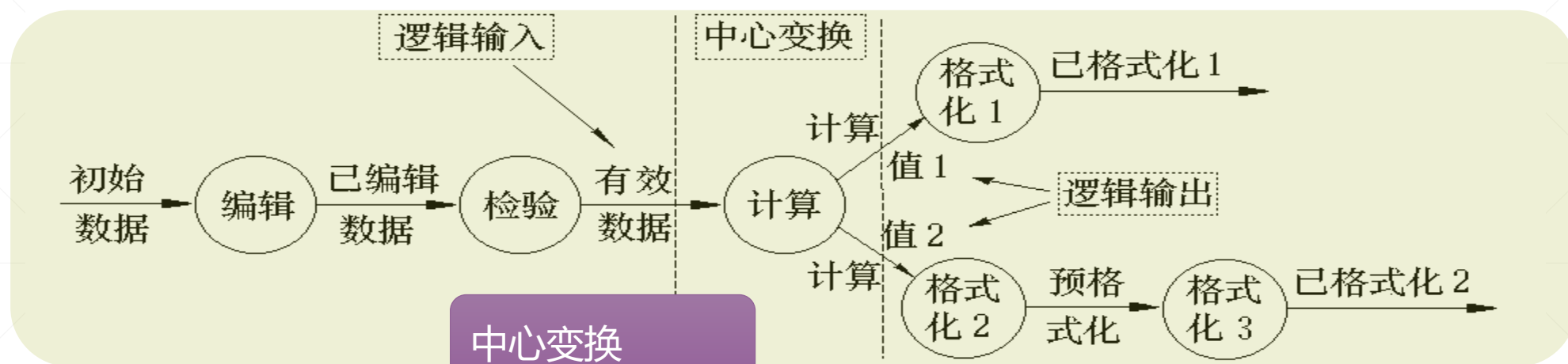
它接受一项事务，根据事务处理的特点和性质，选择分派一个适当的处理单元，然后给出结果。

在事务型系统结构图中，**事务中心模块**按所接受的事务的类型，选择某一事务处理模块执行。各**事务处理模块**并列。每个事务处理模块可能要调用若干个**操作模块**，而操作模块又可能调用若干个**细节模块**。

4.2.2 变换与事务分析

变换分析





变换分析注意事项

深度遍历模块设计

- 在选择模块设计的次序时，必须对一个模块的全部直接下属模块都设计完成之后，才能转向另一个模块的下层模块的设计。

耦合与内聚

- 在设计下层模块时，应考虑模块的耦合和内聚问题，以提高初始结构图的质量。

使用“黑箱”技术

- 在设计当前模块时，先把这个模块的所有下层模块定义成“黑箱”，在设计中利用它们时，暂时不考虑其内部结构和实现。在这一步定义好的“黑箱”，在下一步就可以对它们进行设计和加工。这样，又会导致更多的“黑箱”。最后，全部“黑箱”的内容和结构应完全被确定。

控制直接下属模块数

- 在模块划分时，一个模块的直接下属模块一般在5个左右。如果直接下属模块超过10个，可设立中间层次。

停止模块功能分解的情况

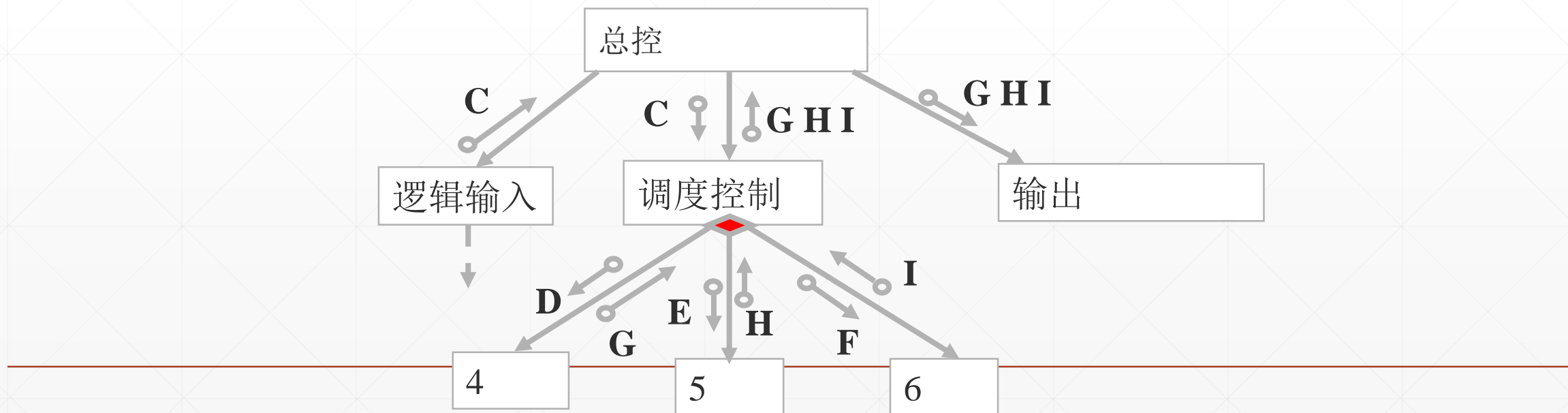
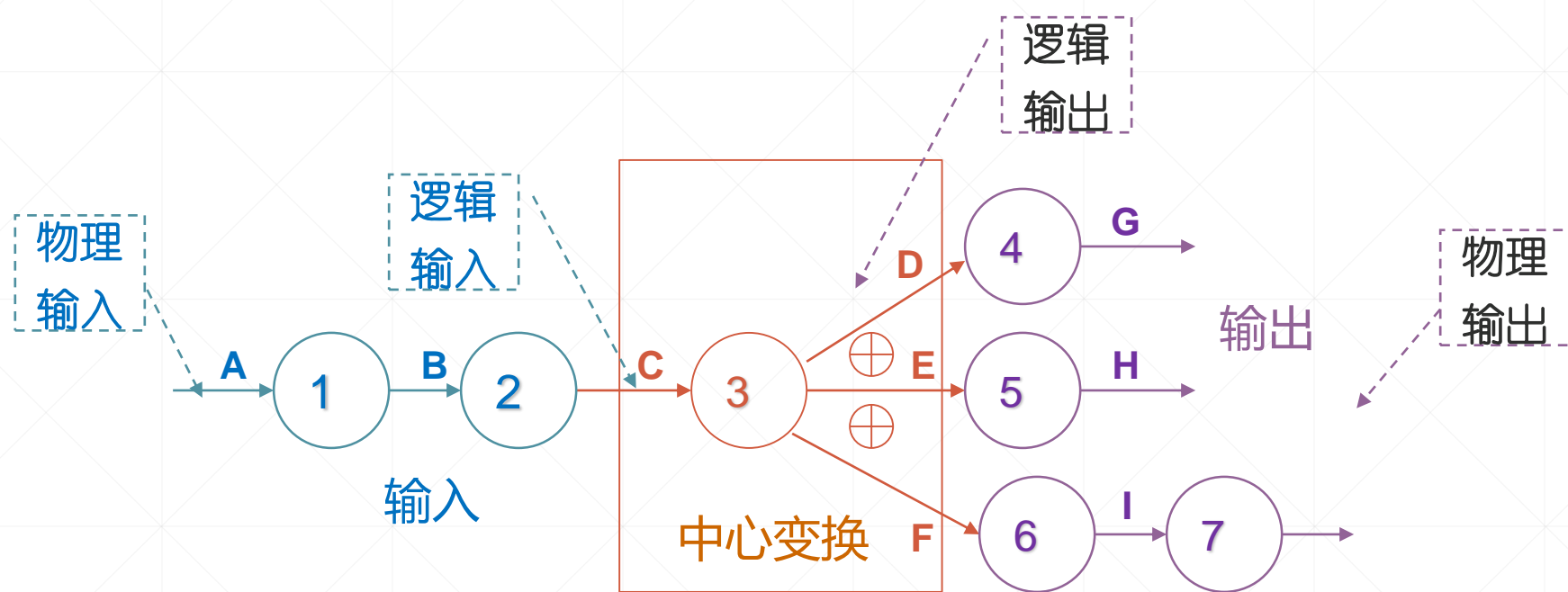
- 模块不能再细分为明显的子任务；
 - 分解成用户提供的模块或程序库的子程序；
 - 模块的界面是输入 / 输出设备传送的信息；
 - 模块不宜再分解得过小。
-

事务分析

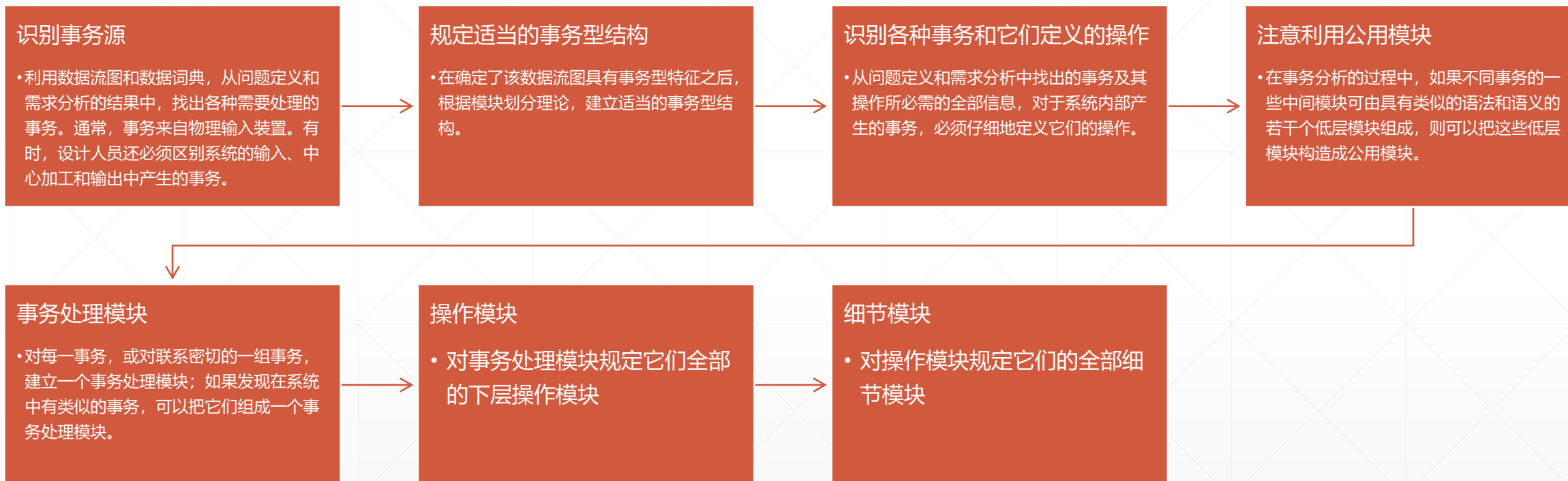


在很多软件应用中，存在某种作业数据流，它可以引发一个或多个处理，这些处理能够完成该作业要求的功能。这种数据流就叫做事务。

与变换分析一样，事务分析也是从分析数据流图开始，自顶向下，逐步分解，建立系统结构图。



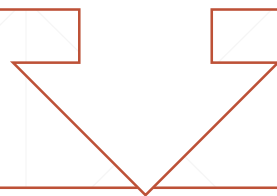
事务分析过程



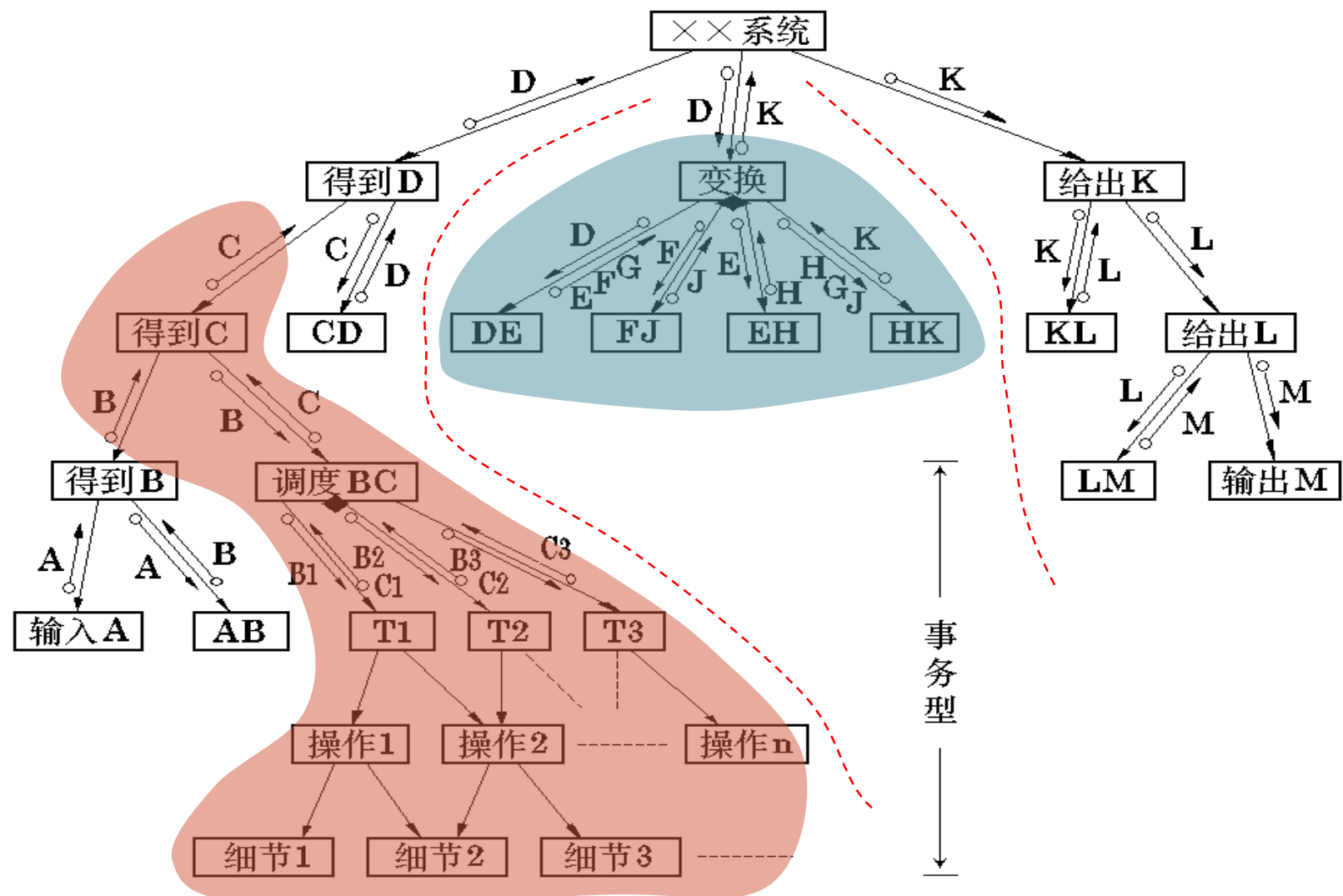
混合结构分析

变换分析是软件系统结构设计的主要方法。

一般，一个大型的软件系统是变换型结构和事务型结构的混合结构。



所以，我们通常利用以变换分析为主、事务分析为辅的方式进行软件结构设计。



4.2.3 面向过程的组件设计之流程图

结构化组件设计

组件级设计也称为过程设计、详细设计，位于数据设计、体系结构设计和接口设计完成之后

任何程序总可以用三种结构化的构成元素来设计和实现

- 顺序：任何算法规约中的核心处理步骤
- 条件：允许根据逻辑情况选择处理的方式
- 重复：提供了循环

详细设计工具可以分为以下三类：

- 图形设计符号：流程图、盒图等
 - 表格设计符号：决策表等
 - 程序设计语言：PDL等
-

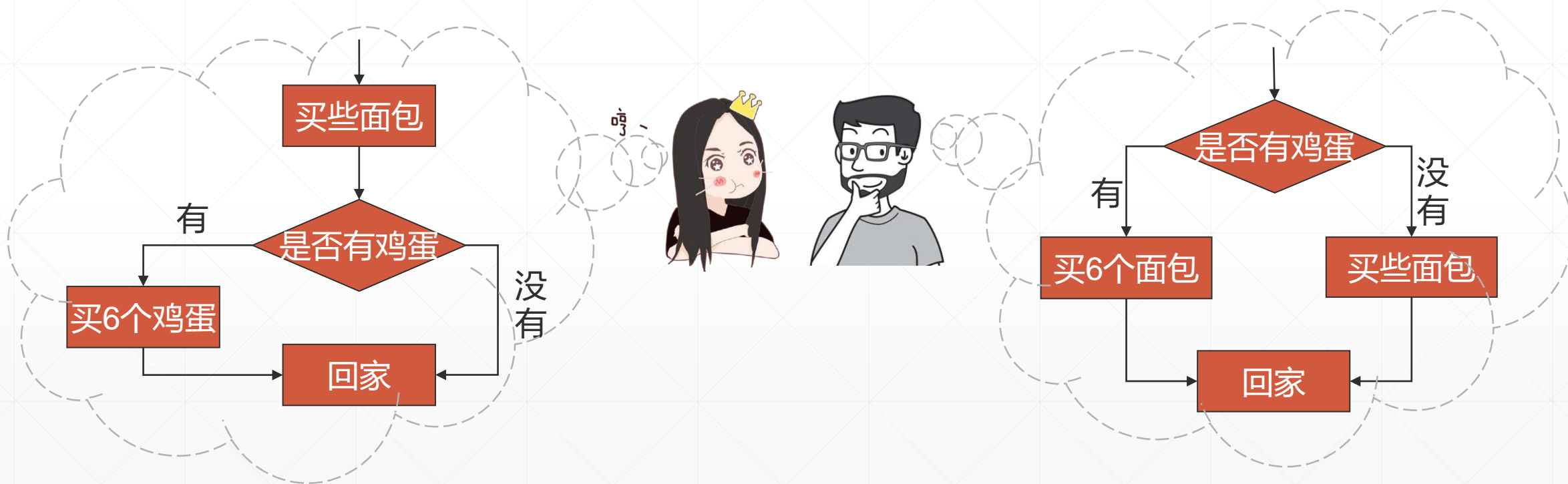
程序员的笑话

程序员的老婆吩咐他去商店买东西：你去附近的商店买些面包，如果有鸡蛋的话，买6个回来。

随后，老公买了6个面包回来。

他的妻子大吃一惊：你为什么买了6个面包？！

程序员回答：因为他们有鸡蛋。



流程图(flow chart)

利用各种方块图形、线条及箭头等符号来表达解决问题的步骤及进行的顺序；
是算法的一种表示方式。

“标准作业流程” (SOP, Standard operating procedure)

- 企业界常用的一种作业方法，其目的在使每一项作业流程均能清楚呈现，任何人只要看到流程图，便能一目了然，有助于相关作业人员对整体工作流程的掌握。

程序流程图(program flow chart)

- 表示程序中的处理过程。

制作流程图的优点



所有流程一目了然，工作人员能掌握全局。


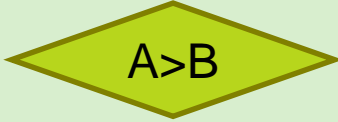

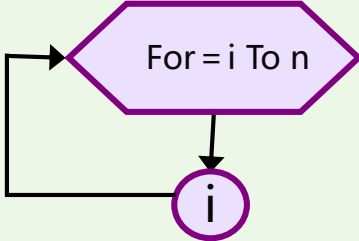

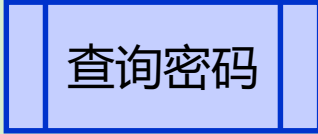


更换人手时，按图索骥，容易上手。

所有流程在绘制时，很容易发现疏失之处，可适时予以调整更正，使各项作业更为严谨。

流程图的基本符号

	名 称	意 义	范 例
	开始 (Start) 终止 (End)	表示程序的开始或结束	 
	路径(Path)	表示流程进行的方向	
	输入(Input) 输出(Output)	表示数据的输入或结果的输出	
	处理(Process)	表示执行或处理某一项工作	

流程图的基本符号

	名 称	意 义	范 例
	决策判断 (Decision)	针对某一条件进行判断	
	循环 (Loop)	表示循环控制变量的初始值及终值	
	子程序 (Subroutine)	用以表示一群已经定义流程的组合	
	文件 (Document)	指输入输出的文件	

流程图的基本结构

顺序结构 (Sequence)

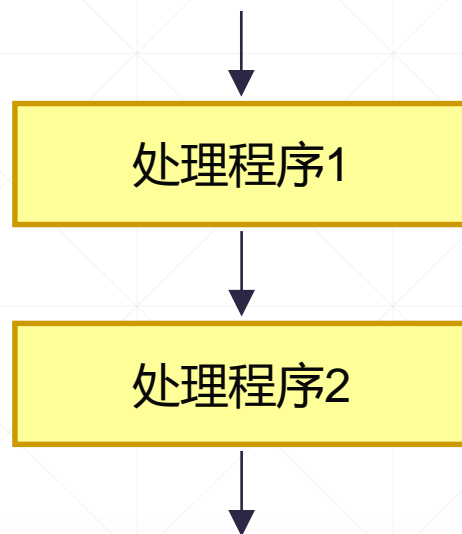
选择结构 (Selection)

- 二元选择结构 (基本结构)
- 多重选择结构

循环结构 (Iteration)

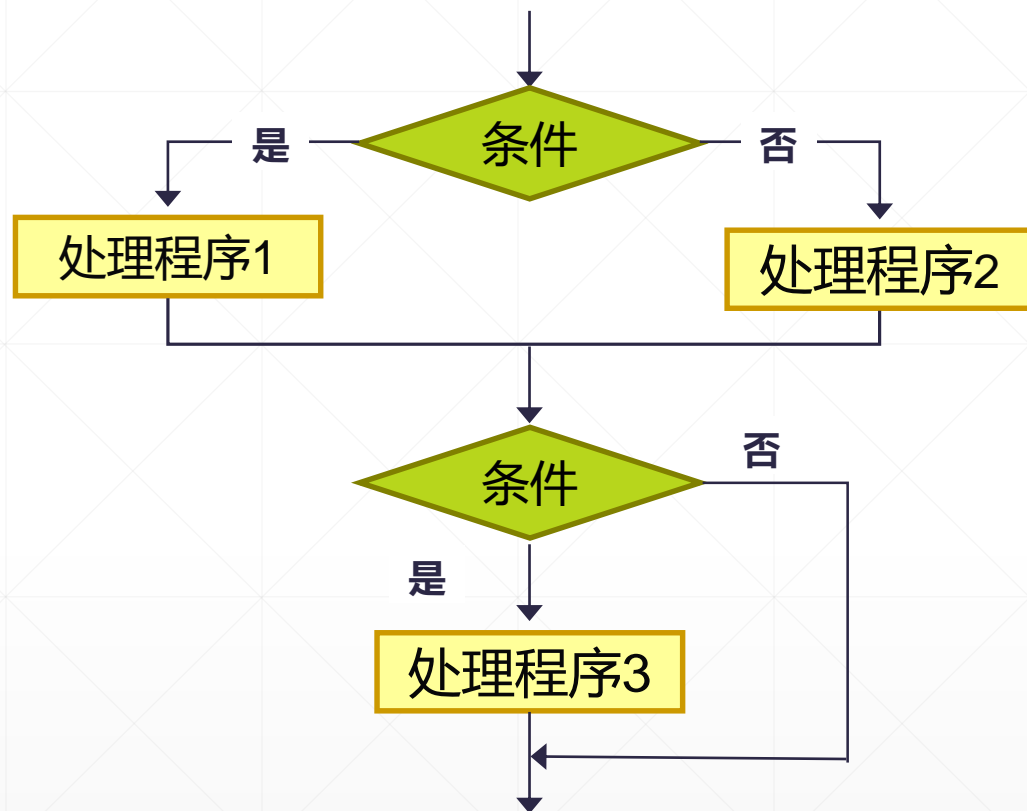
- while-do结构
 - do-while结构
-

顺序结构 (Sequence)



意义：处理程序顺序进行。

二元选择结构 (基本结构)

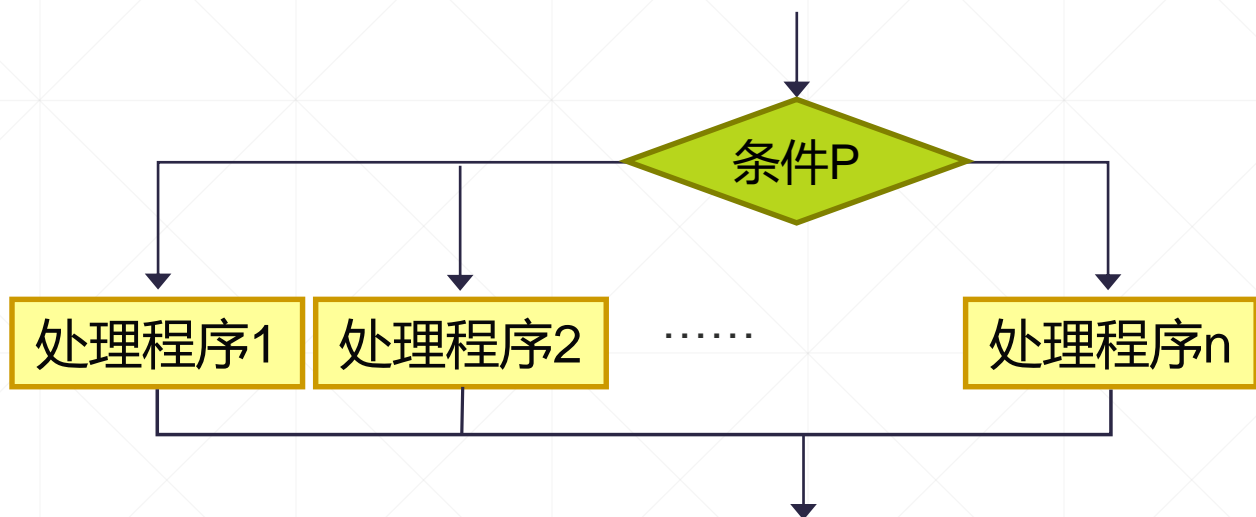


语法:

```
if (条件) {  
    处理程序1;  
}  
else {  
    处理程序2;  
}  
if (条件) {  
    处理程序3;  
}
```

意义：流程依据某些条件，依条件是否成立，分别进行不同处理程序。

多重选择结构



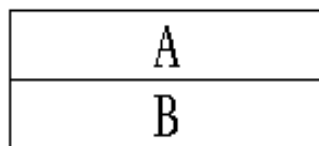
语法:

```
switch (条件) {  
  case p=1:  
    处理程序1;  
  case p=2:  
    处理程序2;  
  ...  
  case p=n:  
    处理程序n;  
}
```

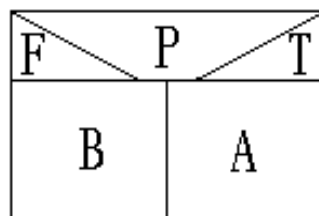
意义：流程依据某些条件，在不同的条件成立时，分别进行不同处理程序。
例如条件P=1时，进行处理程序1。条件P=n时，进行处理程序n。

4.2.4 面向过程的其他组件设计方法

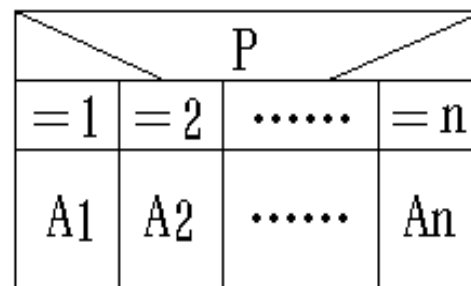
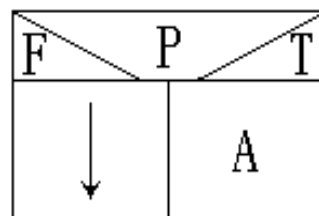
盒图 (N-S图)



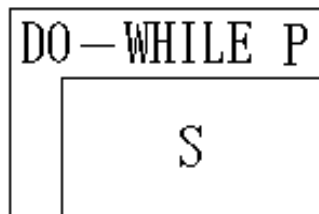
① 顺序型



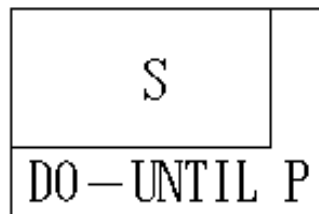
② 选择型



⑤ 多分支选择型
(CASE 型)



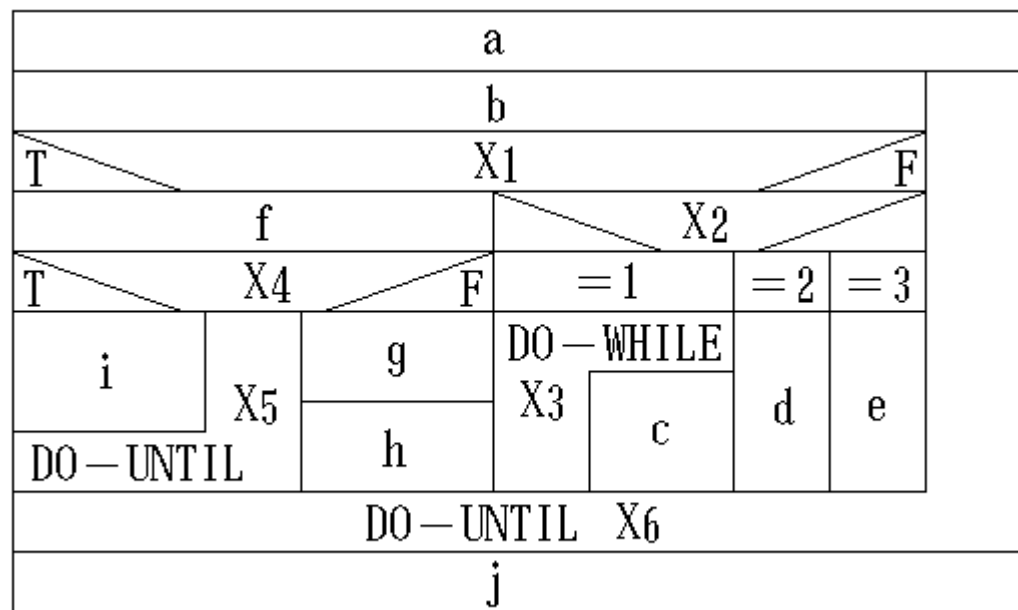
③ WHILE 重复型



④ UNTIL 重复型

五种基本控制结构由五种图形构件表示。

盒图示例



PDL (程序设计语言)



PDL是一种用于描述功能模块的算法设计和加工细节的语言。称为程序设计语言。它是一种伪码。

伪码的语法规则分为“外语法”和“内语法”。

PDL具有严格的关键字外语法，用于定义控制结构和数据结构，同时它的表示实际操作和条件的内语法又是灵活自由的，可使用自然语言的词汇。

PDL示例: 拼词检查程序

```
PROCEDURE spellcheck IS
BEGIN
    split document into single words
    load up words in dictionary
    display words which are not in dictionary
    create a new dictionary
END spellcheck
```



使用PDL语言
逐步求精

```
PROCEDURE spellcheck

BEGIN
    --* split document into single words
    LOOP get next word
        add word to word list in sortorder
    EXIT WHEN all words processed
    END LOOP

    --* look up words in dictionary
    LOOP get word from word list
        IF word not in dictionary THEN
            --* display words not in dictionary
            display word

            prompt on user terminal
            IF user response says word OK THEN
                add word to good word list
            ELSE
                add word to bad word list
            ENDIF
        ENDIF
    EXIT WHEN all words processed
    END LOOP

    --* create a new words dictionary
    dictionary := merge dictionary and good word list

END spellcheck
```

判定表 (决策表)

判定表用于表示程序的静态逻辑

在判定表中的条件部分给出所有的两分支判断的列表, 动作部分给出相应的处理

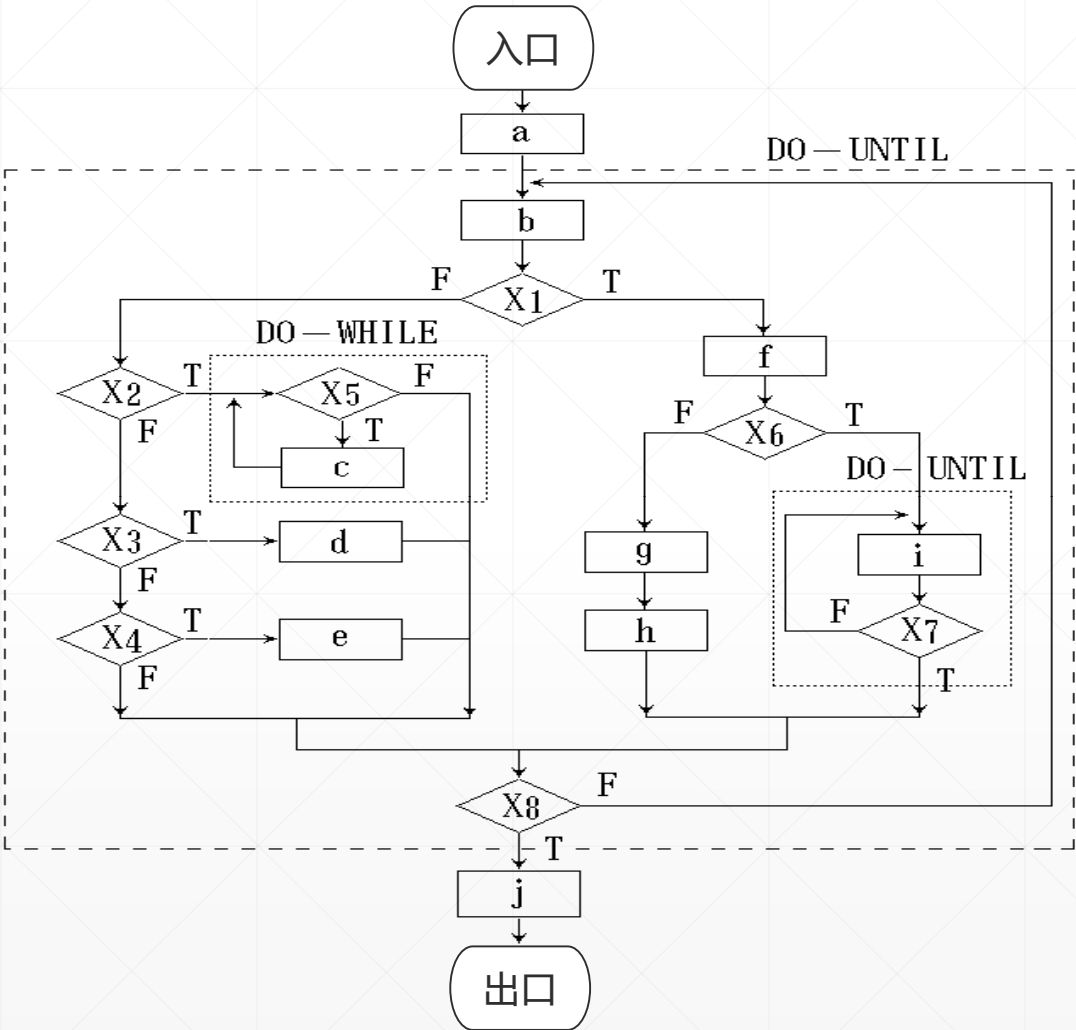
要求将程序流程图中的多分支判断都改成两分支判断

条件

动作

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
X1	T	T	T	T	T	F	F	F	F	F	F	F	F	F
X2	-	-	-	-	-	T	T	T	F	F	F	F	F	F
X3	-	-	-	-	-	-	-	-	F	F	T	T	F	F
X4	-	-	-	-	-	-	-	-	F	F	-	-	T	T
X5	-	-	-	-	-	T	F	F	-	-	-	-	-	-
X6	T	T	T	F	F	-	-	-	-	-	-	-	-	-
X7	T	T	F	-	-	-	-	-	-	-	-	-	-	-
X8	T	F	-	T	F	-	T	F	F	T	T	F	T	F
a	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
b	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
c	-	-	-	-	-	Y	-	-	-	-	-	-	-	-
d	-	-	-	-	-	-	-	-	-	-	Y	Y	-	-
e	-	-	-	-	-	-	-	-	-	-	-	-	Y	Y
f	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
g	-	-	-	Y	Y	-	-	-	-	-	-	-	-	-
h	-	-	-	Y	Y	-	-	-	-	-	-	-	-	-
i	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
j	Y	-	-	Y	-	-	Y	-	-	Y	Y	-	Y	-

无多分支流程图一>判定图



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
X1	T	T	T	T	T	F	F	F	F	F	F	F	F	F
X2	-	-	-	-	-	T	T	T	F	F	F	F	F	F
X3	-	-	-	-	-	-	-	-	F	F	T	T	F	F
X4	-	-	-	-	-	-	-	-	F	F	-	-	T	T
X5	-	-	-	-	-	T	F	F	-	-	-	-	-	-
X6	T	T	T	F	F	-	-	-	-	-	-	-	-	-
X7	T	T	F	-	-	-	-	-	-	-	-	-	-	-
X8	T	F	-	T	F	-	T	F	F	T	T	F	T	F
a	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
b	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
c	-	-	-	-	-	Y	-	-	-	-	-	-	-	-
d	-	-	-	-	-	-	-	-	-	-	Y	Y	-	-
e	-	-	-	-	-	-	-	-	-	-	-	-	Y	Y
f	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
g	-	-	-	Y	Y	-	-	-	-	-	-	-	-	-
h	-	-	-	Y	Y	-	-	-	-	-	-	-	-	-
i	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
j	Y	-	-	Y	-	-	Y	-	-	Y	Y	-	Y	-

第4章 软件设计

4.1 软件设计的概念

4.2 面向过程的设计

4.3 面向对象的设计

4.3.1 面向对象的架构设计

面向对象设计的活动



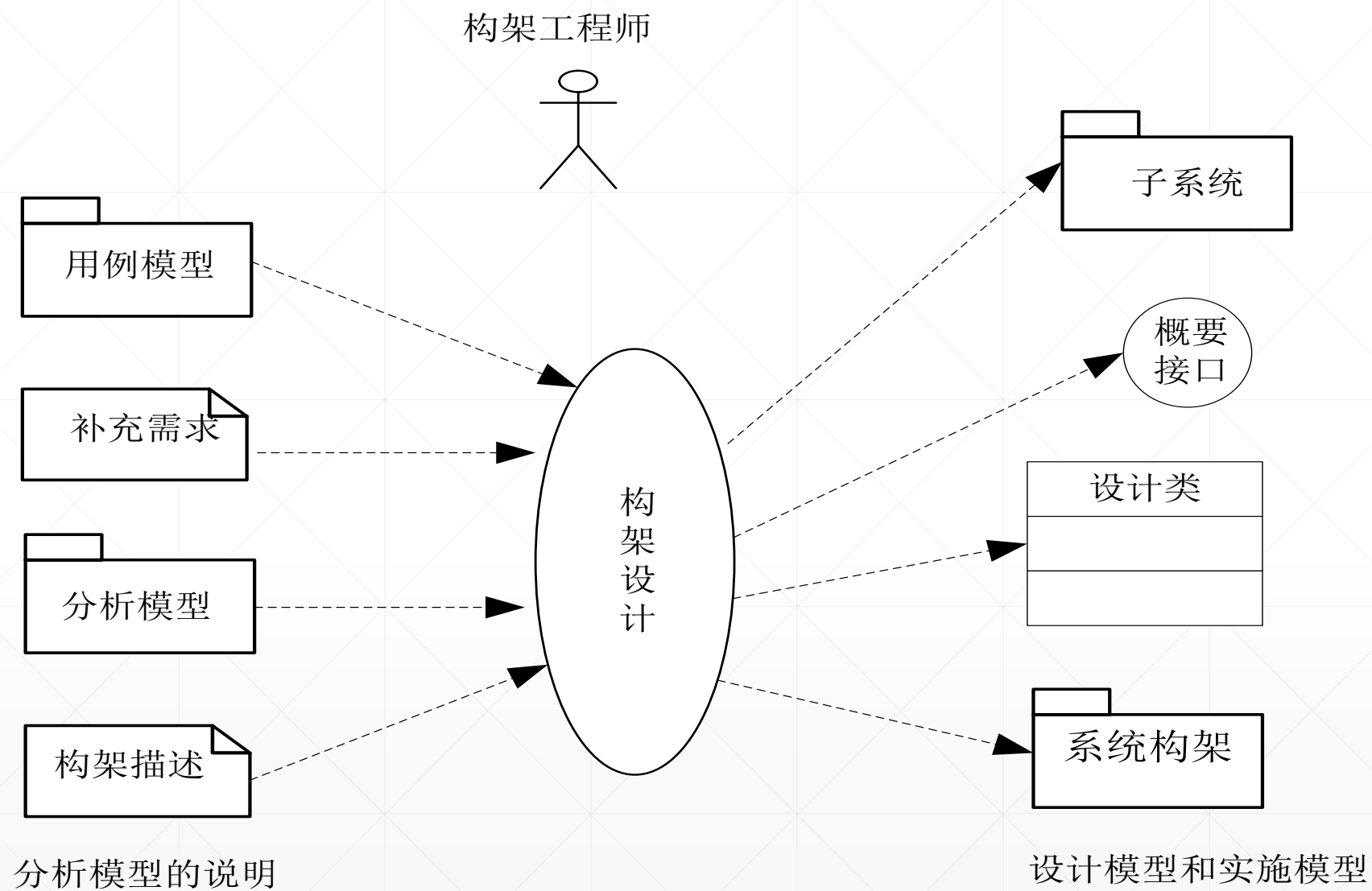
面向对象设计活动之一：架构设计



架构设计的目的是要勾画出系统的总体结构，这项工作由经验丰富的架构设计师主持完成。

输入：用例模型、分析模型。

输出：物理结构、子系统及其接口、概要的设计类。



架构设计第1步：构造系统的物理模型



首先用UML的配置图（部署图）描述系统的物理架构

将需求分析阶段捕获的系统功能分配到这些物理节点上。

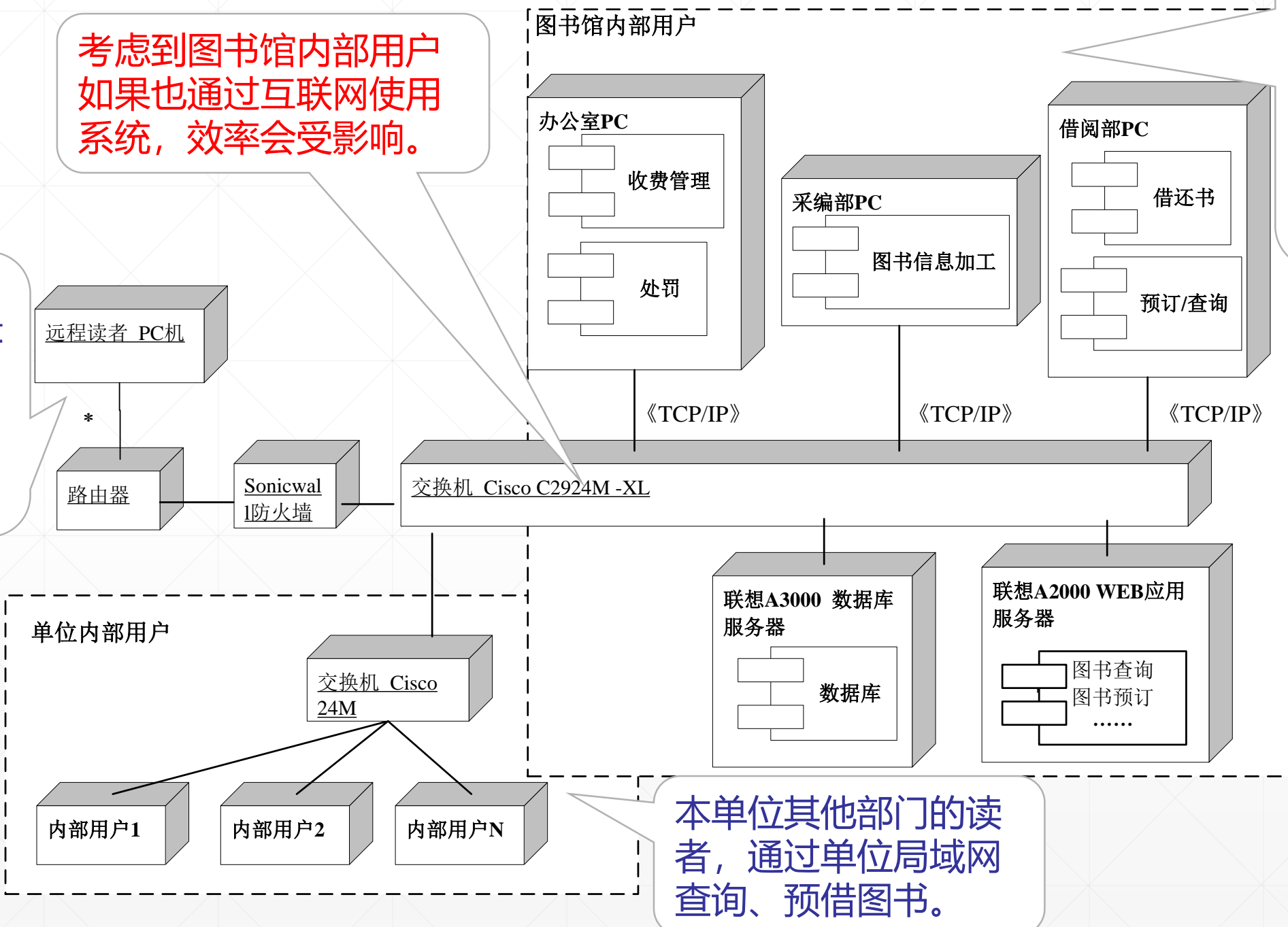
配置图上可以显示计算节点的拓扑结构、硬件设备配置、通信路径、各个节点上运行的系统软件配置、应用软件配置。

一个图书馆信息管理系统的物理模型如后图所示。

考虑到图书馆内部用户
如果也通过互联网使用
系统，效率会受影响。

图书馆内部
工作人员，
在局域网上
完成日常的
借还书、采
编、图书管
理等工作。

远程读者，
通过Internet
访问系统，
实现查询图
书、预借图
书的功能。



架构设计第2步：设计子系统

对于一个复杂的软件系统来说，将其分解成若干个子系统，子系统内还可以继续划分子系统或包，这种自顶向下、逐步细化的组织结构非常符合人类分析问题的思路。

每个子系统与其它子系统之间应该定义接口，在接口上说明交互信息，注意这时还不要描述子系统的内部实现。

可用UML组件图表示。

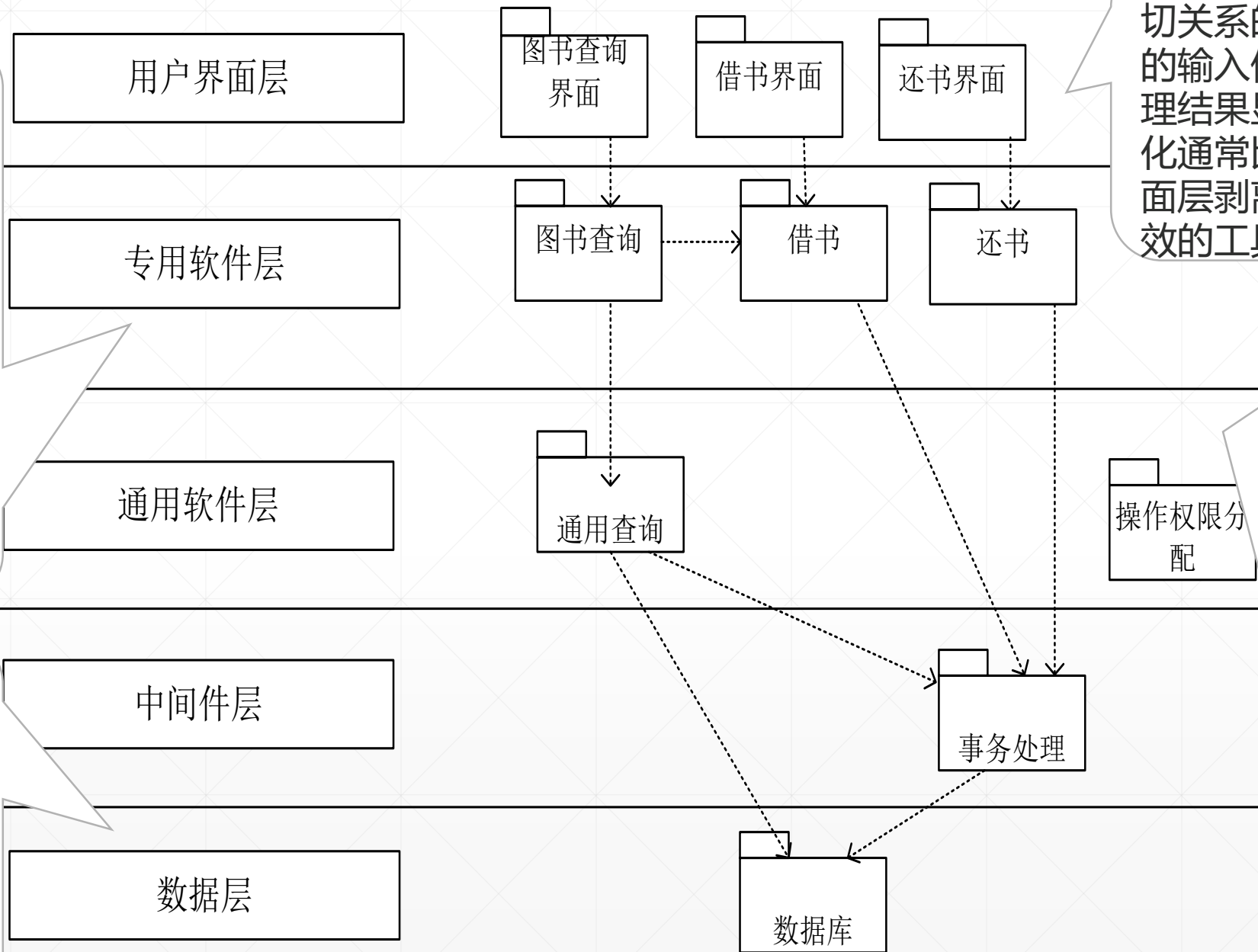
架构设计第2步：设计子系统

1) 划分各个子系统的方式：

- 按照功能划分，将相似的功能组织在一个子系统中；
 - 按照系统的物理布局划分，将在同一个物理区域内的软件组织为一个子系统；
 - 按照软件层次划分子系统，软件层次通常可划分为用户界面层、专用软件层、通用软件层、中间层和数据层，见后图。
-

专用软件层是每个项目中特殊的应用部分，它们被复用的可能性很小。在开发时可以适当地减小软件元素的粒度，以便分离出更多的可复用构件，减少专用软件层的规模。

数据层主要存放应用系统的数据，通常由数据库管理系统管理，常用的操作有更新、保存、删除、检索等。



用户界面层是与用户应用有密切关系的内容，主要接受用户的输入信息，并且将系统的处理结果显示给用户。这部分变化通常比较大，所以建议将界面层剥离出来，用一些快捷有效的工具实现。

通用软件层是由一些公共构件组成，这类软构件的可复用性很好。在设计应用软件时首先要将软件的特殊部分和通用部分分离，根据通用部分的功能检查现有的构件库。如果有可用的构件，则复用已有的构件会极大地提高软件的开发效率和质量。如果没有可复用的构件，则尽可能设计可复用的构件并且添加到构件库中，以备今后复用。

架构设计第2步：设计子系统

2) 定义子系统之间的关系：

划分子系统后，要确定子系统之间的关系。子系统之间的关系：

- “请求 - 服务”关系，“请求”子系统调用“服务”子系统，“服务”子系统完成一些服务，并且将结果返回给“请求”子系统。
- 平等关系，每个子系统都可以调用其它子系统。
- 如果子系统的内容相互有关联，就应该定义它们之间的依赖关系。在设计时，相关的子系统之间应该定义接口，依赖关系应该指向接口而不要指向子系统的内容。

注意：

如果两个子系统之间的关系过于密切，则说明一个子系统的变化会导致另一个子系统变化，这种子系统理解和维护都会比较困难。

解决子系统之间关系过于密切的办法基本上有两个：

- 重新划分子系统，这种方法比较简单，将子系统的粒度减少，或者重新规划子系统的内容，将相互依赖的元素划归到同一个子系统之中；
- 定义子系统的接口，将依赖关系定义到接口上；

架构设计第2步：设计子系统

3) 定义子系统的接口

- 每个子系统的接口上定义了若干操作，体现了子系统的功能，而功能的具体实现方法应该是隐藏的，其他子系统只能通过接口间接地享受这个子系统提供的服务，不能直接操作它。
-

架构设计第3步：非功能需求设计

分析阶段定义了整个系统的非功能需求，在设计阶段要研究这些需求，设计出可行的方案。

非功能需求包括：

- 系统的安全性、错误监测和故障恢复、可移植性和通用性等等。

具有共性的非功能需求一般设计在中间层和通用应用层，目的是充分利用已有构件，减少重新开发的工作量。

4.3.2 面向对象的用例设计与类设计

面向对象设计活动之二：进一步细化用例



根据分析阶段产生的高层类图和交互图，由用例设计师研究已有的类，将它们分配到相应的用例中。

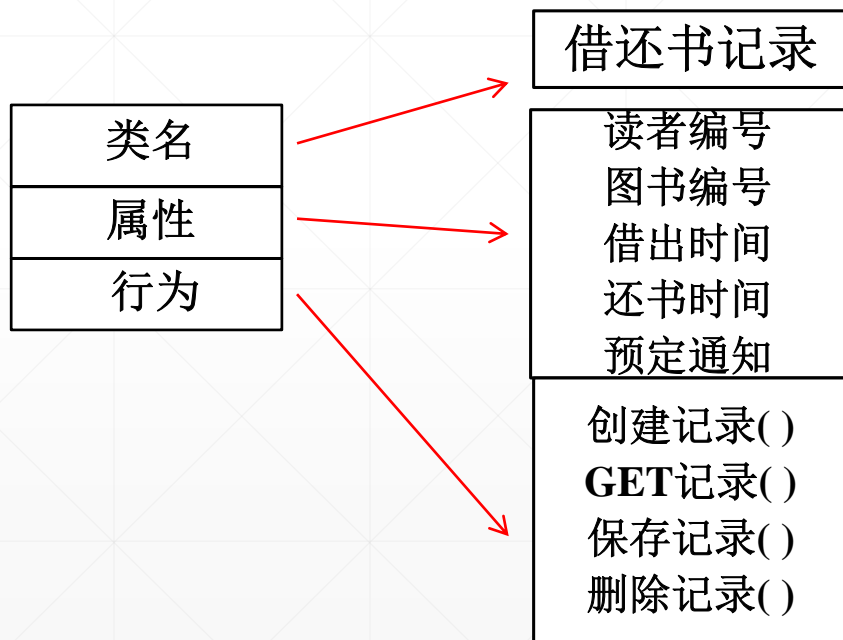
检查每个用例功能，依靠当前的类能否实现，同时检查每个用例的特殊需求是否有合适的类来实现。

细化每个用例的类图，描述实现用例的类及其类之间的相互关系，其中的通用类和关键类可用粗线框区分，这些类将作为项目经理检查项目时的重点。

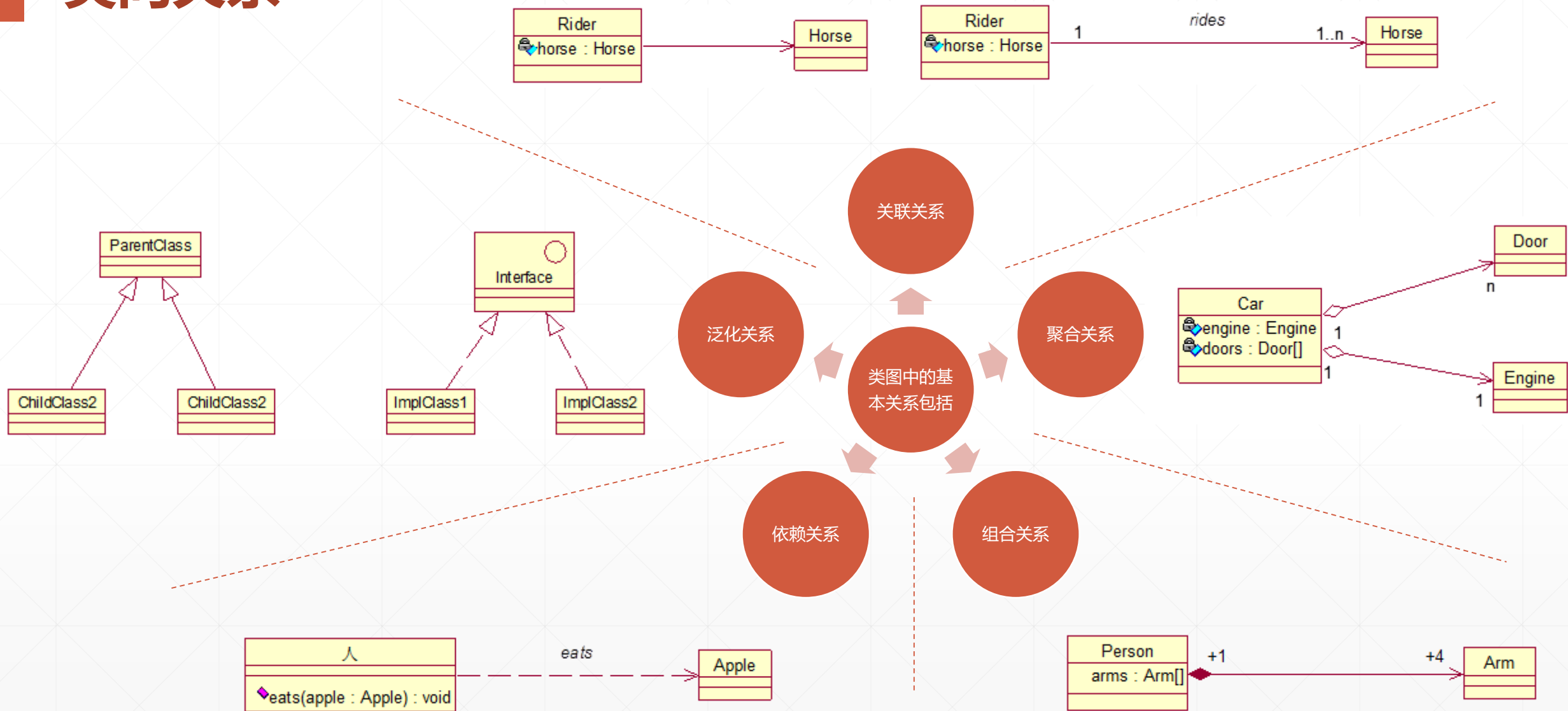
类

类是包含信息和影响信息行为的逻辑元素。类的符号是由三个格子的长方形组成，有时下面两个格子可以省略。

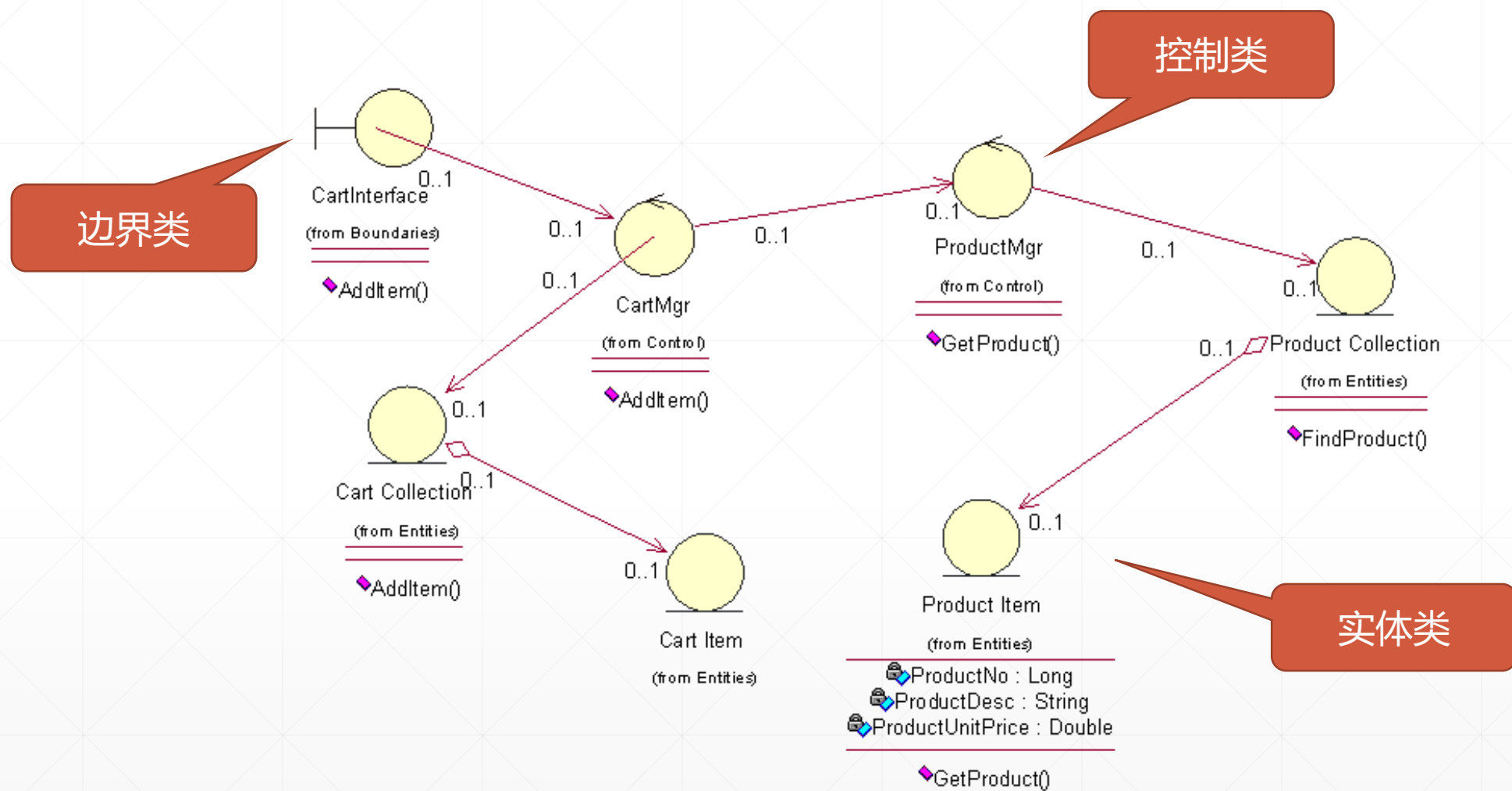
最顶部的格子包含类的名字，类的命名应尽量用应用领域中的术语，有明确的含义，以利于开发人员与用户的理解和交流。中间的格子说明类的属性。最下面的格子是类的操作行为。



类间关系



分析类图



如何找实体类?

实体类用于对**必须存储的信息和相关行为**进行建模

实体类源于业务模型中的**业务实体**，但出于对系统结构的优化，可以在后续的过程中被分拆、合并

如何找边界类?

参与者与用例之间应当建立边界类

用例与用例之间如果有交互，应当为其建立边界类

如果用例与系统边界之外的非人对象有交互，应当为其建立边界类

在相关联的业务对象有明显的独立性要求，即它们可能在各自的领域内发展和变化，但又希望互不影响时，也应当为它们建立边界类

如何找控制类?

控制类来源于对用例场景中动词的分析和定义

控制类主要起到协调对象的作用，例如从边界类通过控制类访问实体类，或者实体类通过控制类访问另一个实体类。

如果用例场景中的行为在执行步骤、执行要求或者执行结果上具有类似的特征，应当合并或抽取超类

细化用例

第1步：通过扫描用例中所有的交互图识别参与用例解决方案的类。在设计阶段完善类、属性和方法。例如，每个用例至少应该有一个控制类，它通常没有属性而只有方法，它本身不完成什么具体的功能，只是起协调和控制作用。

每个类的方法都可以通过分析交互图得到，一般地检查所有的交互图发送给某个类的所有消息，这表明了该类必须定义的方法。例如“借书控制”类向“读者”类发送“检查读者（读者编号）”消息，那么“检查读者”就作为“读者”类应该提供的方法。

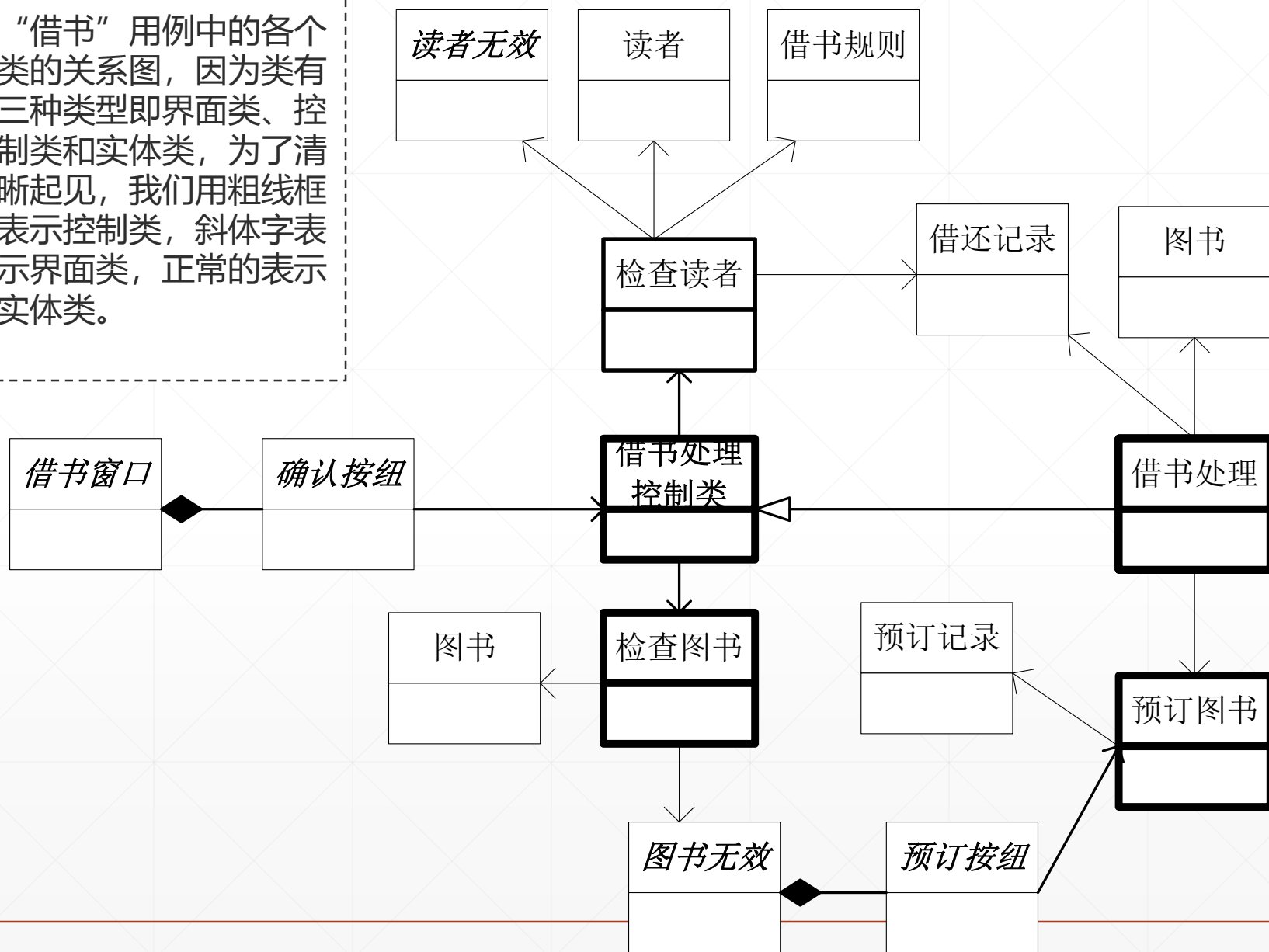


第2步：添加属性的类型、方法的参数类型和方法的返回类型。

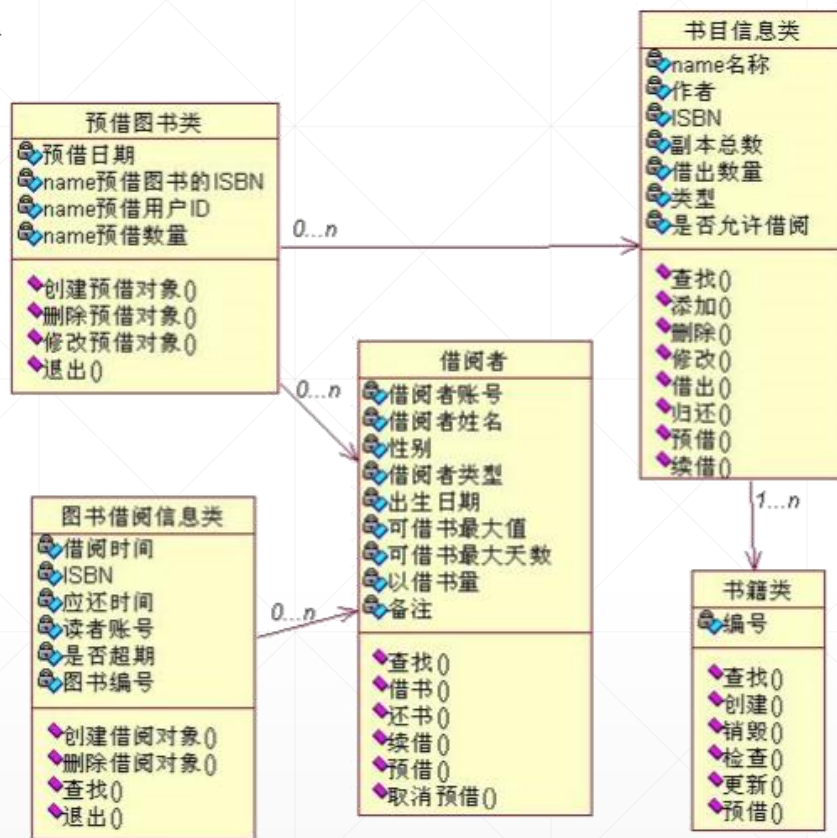


第3步：添加类之间的关系，包括关联、依赖、继承等。

“借书”用例中的各个类的关系图，因为类有三种类型即界面类、控制类和实体类，为了清晰起见，我们用粗线框表示控制类，斜体字表示界面类，正常的表示实体类。



面向对象设计活动之三：详细设计一个类



由构件工程师详细设计每个类的属性、方法和关系。

类设计第1步：定义类的属性

用所选择的编程语言定义每个类的属性。类的属性反映类的特性，通常属性是被封装在类的内部，不允许外部对象访问。

注意点：

- 分析阶段和概要设计阶段定义的一个类属性在详细设计时可能要被分解为多个，**减小属性的表示粒度**有利于实现和重用。但是一个类的属性如果太多，则应该检查一下，看能否分离出一个新的类。
- 如果一个类因为其属性的原因变得复杂而难于理解，那么就将一些属性**分离出来形成一个新的类**。
- 通常不同的编程语言提供的数据类型有很大差别，确定类的属性时要用编程语言来约束可用的属性类型。定义属性类型时**尽可能使用已有的类型**，太多的自定义类型会降低系统的可维护性和可理解性等性能指标。
- 类的属性结构要**坚持简单的原则**，尽可能不使用复杂的数据结构。

类设计第2步：定义类的操作

由构件工程师为每个类的方法设计必须实现的操作，并用自然语言或伪代码描述操作的实现算法。一个类可能被应用在多个用例中，由于它在不同用例中担当的角色不同，所以设计时要求详细周到。

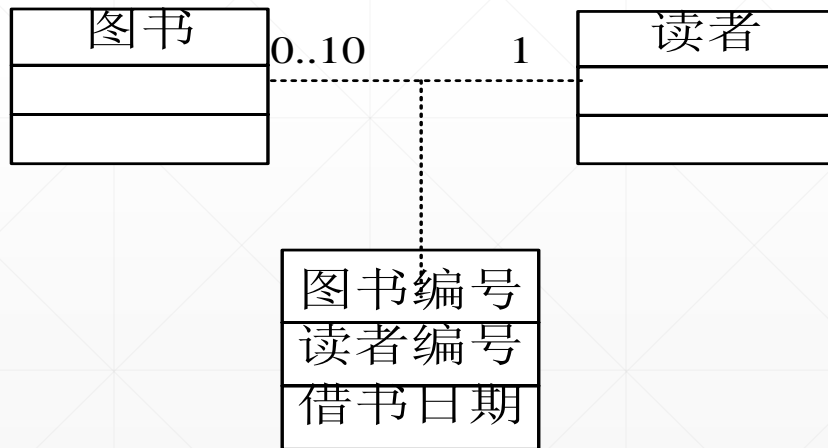
注意事项：

- 分析类的每个**职责**的具体含义，从中找出类应该具备的操作。
- 阅读类的**非功能需求**说明，添加一些必须的操作。
- 确定类的**接口**应该提供的操作。这关系到设计的质量，特别是系统的稳定性，所以确定类接口操作要特别小心。
- 逐个检查类在每个用例实现中是否合适，补充一些**必须的操作**。
- 设计时不仅要考虑到系统正常运行的情况，还要考虑一些**特殊情况**，如中断/错误处理等。

类设计第3步：定义类之间的关系

设置基数：一个类的实例与另一个类的实例之间的联系。在图书馆信息管理系统中，“图书”类和“读者”类关联，如果需求说明中有“一位读者可借图书的数量为0至10本”，那么它们之间的基数为1：0..10。

使用关联类：可以放置与关联相关的属性。例如“图书”类和“读者”类，如果要反映读者的借书情况，该如何处理呢？可以创建一个关联类，这个类中的属性是“借书日期”。



4.3.3 UML顺序图

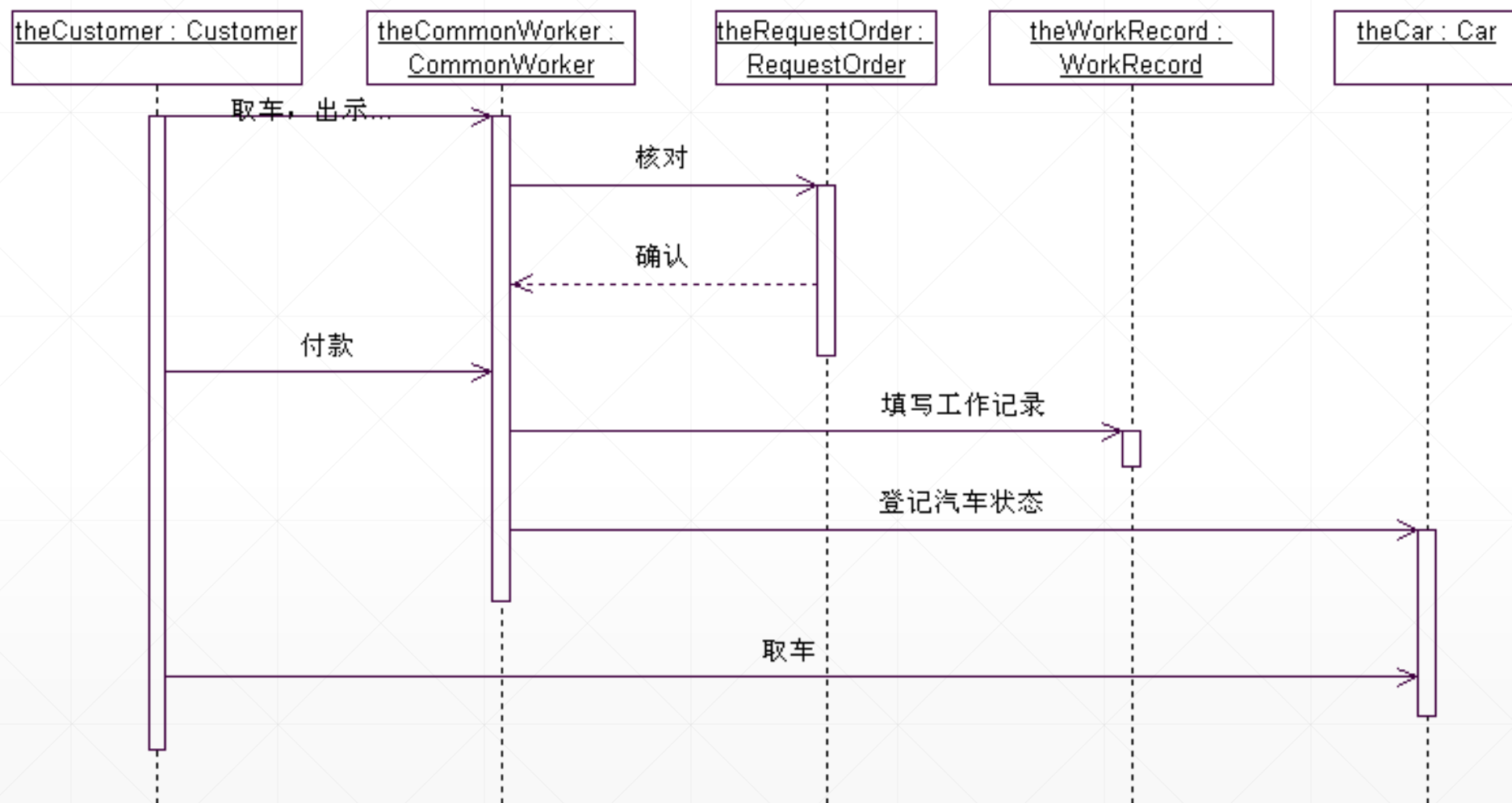
什么是顺序图？

顺序图是强调消息时间顺序的交互图。

顺序图描述了对象之间传送消息的时间顺序，用来表示用例中的行为顺序。

顺序图将交互关系表示为一个二维图。即在图形上，顺序图是一张表，其中显示的对象沿横轴排列，从左到右分布在图的顶部；而消息则沿纵轴按时间顺序排序。创建顺序图时，以能够使图尽量简洁为依据布局。

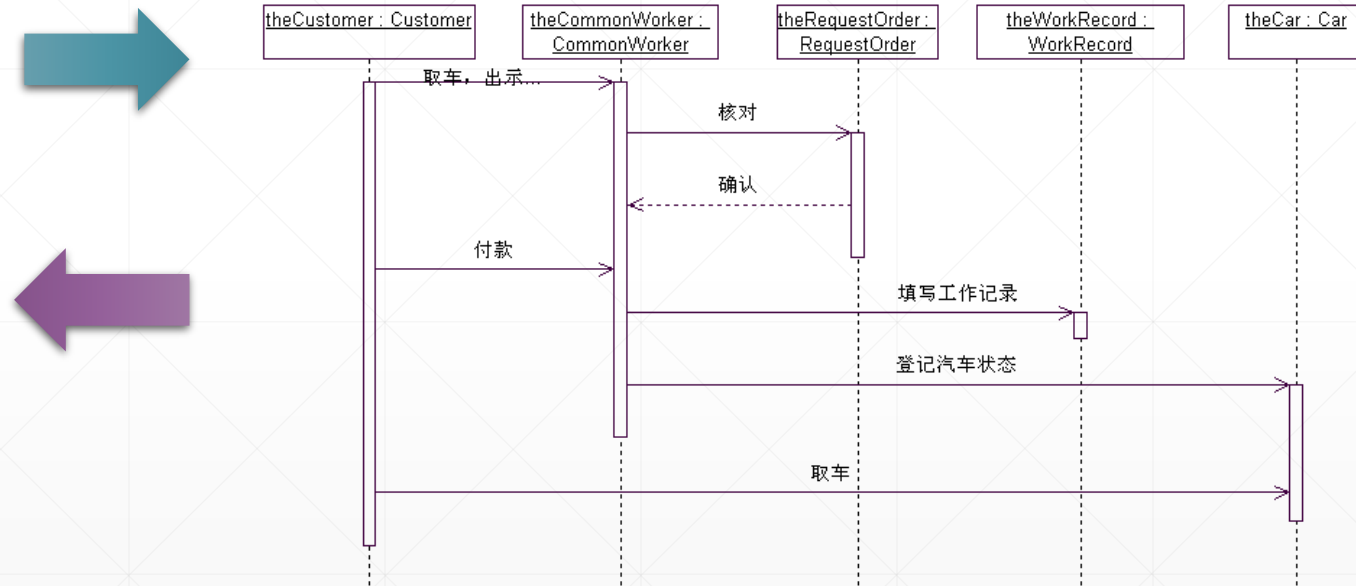
购买小车的顺序图示例



什么时候会用到顺序图?

用例

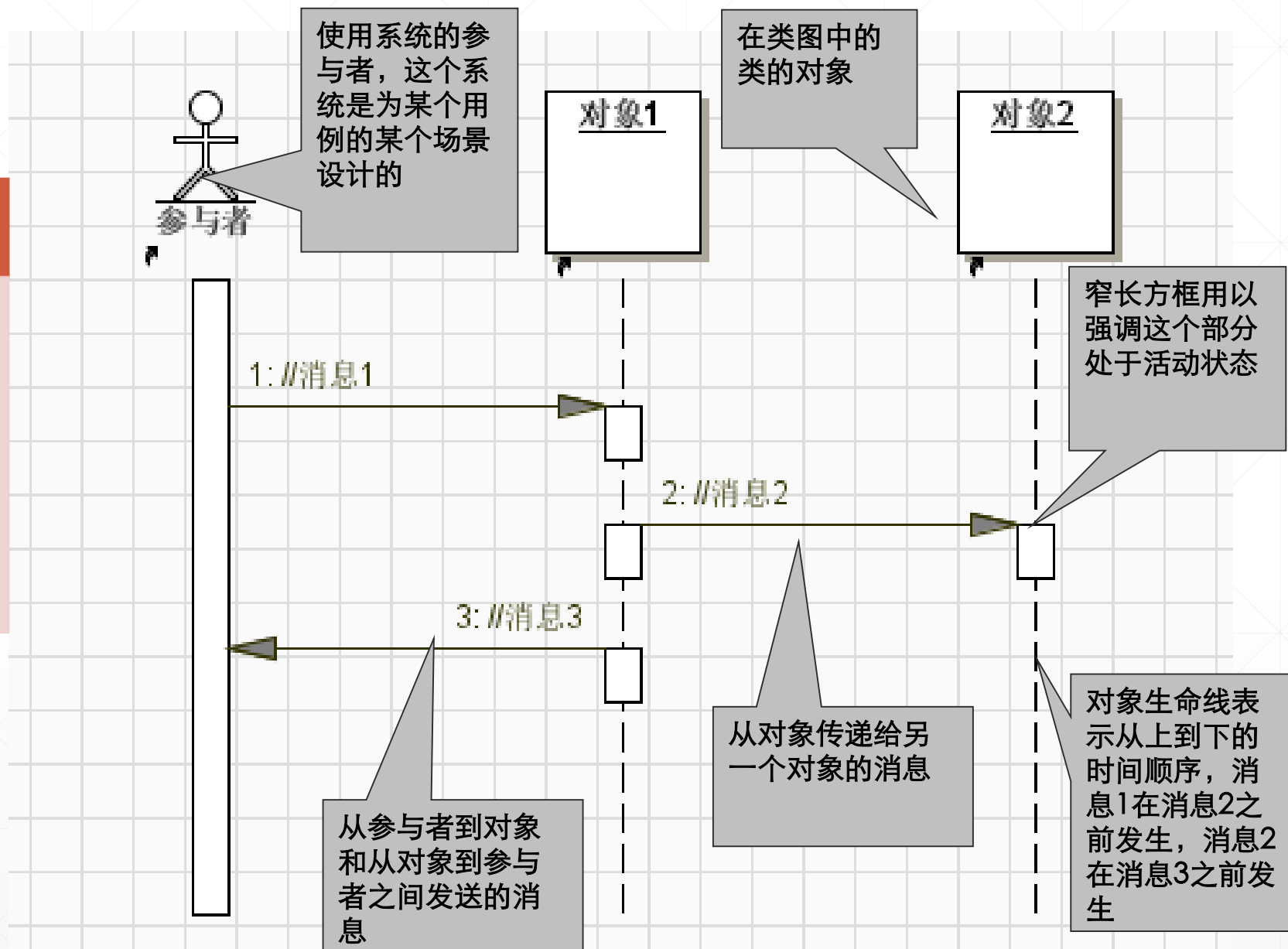
涉及多个类



顺序图的组成

顺序图包含4个元素：

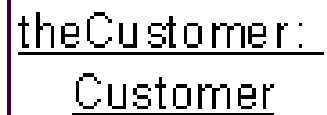
- 对象 (Object)
- 生命线 (Lifeline)
- 消息 (Message)
- 激活 (Activation)



对象

顺序图中对象的符号和对象图中对象所用的符号一样。

将对象置于顺序图的顶部意味着在交互开始的时候对象就已经存在了，如果对象的位置不在顶部，那么表示对象是在交互的过程中被创建的。



theCustomer:
Customer

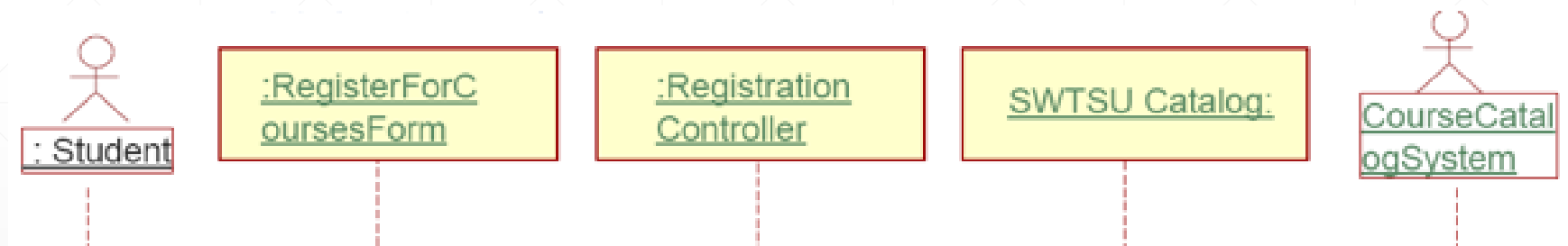
The image shows a UML object notation. It consists of a rectangular box with a thin black border. Inside the box, the text is split into two lines. The top line is 'theCustomer:' and the bottom line is 'Customer'. Both lines are underlined with a thin black line.

对象

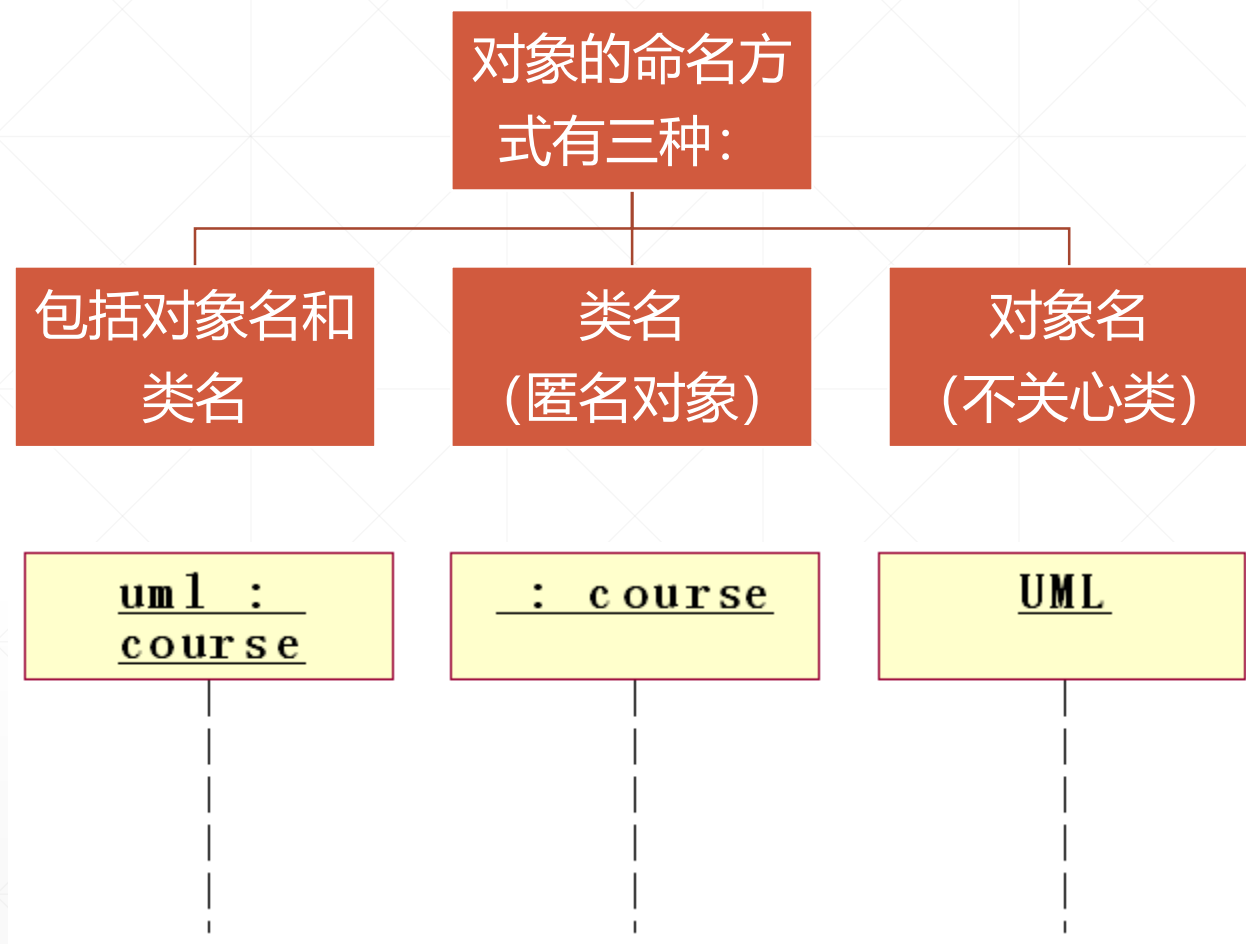
参与者和对象按照从左到右的顺序排列

一般最多两个参与者，他们分列两端。启动这个用例的参与者往往排在最左边；接收消息的参与者则排在最右端；

对象从左到右按照重要性排列或按照消息先后顺序排列。



对象



生命线

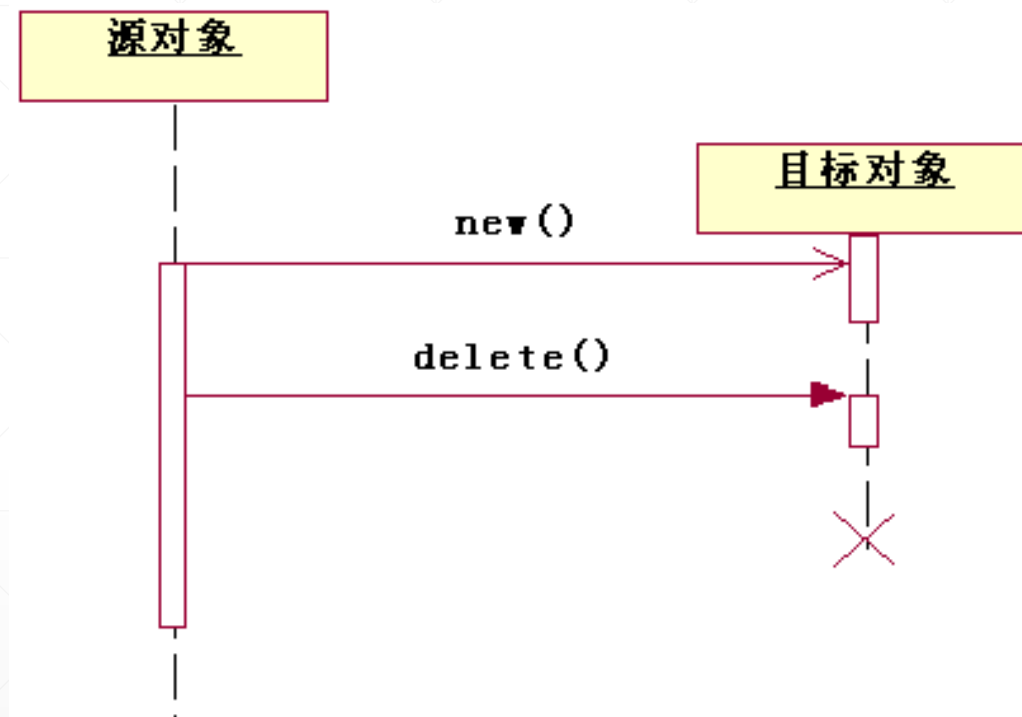
每个对象都有自己的生命线，用来表示在该用例中一个对象在一段时间内的存在

生命线使用垂直的虚线表示

如果对象生命期结束，则用注销符号表示

对象默认的位置在图顶部，表示对象在交互之前已经存在

如果是在交互过程中由另外的对象所创建，则位于图的中间某处。



激活

激活表示该对象被占用以完成某个任务，去激活指的则是对象处于空闲状态、在等待消息。

在UML中，为了表示对象是激活的，可以将该对象的生命线拓宽成为矩形。其中的矩形称为激活条(期)或控制期，对象就是在激活条的顶部被激活的，对象在完成自己的工作后被去激活。



激活期

当一条消息被传递给对象的时候，它会触发该对象的某个行为，这时就说该对象被激活了。

在UML中，激活用一个在生命线上的细长矩形框表示。

矩形本身被称为对象的激活期或控制期，对象就是在激活期顶端被激活的。

激活期说明对象正在执行某个动作。当动作完成后，伴随着一个消息箭头离开对象的生命线，此时对象的一个激活期也宣告结束。



消息

面向对象方法中，消息是对象间交互信息的主要方式。

- 在任何一个软件系统中，对象都不是孤立存在的，它们之间通过消息进行通信。
- 消息是用来说明顺序图中不同活动对象之间的通信。因此，消息可以激发某个操作、创建或撤销某个对象。

结构化程序设计中，模块间传递信息的方式主要是过程（或函数）调用。

- 对象A向对象B发送消息，可以简单地理解为对象A调用对象B的一个操作（operation）。

在顺序图中，消息是由从一个对象的生命线指向另一个对象的生命线的直线箭头来表示的，箭头上面还可以表明要发送的消息名及序号。

- 顺序图中消息编号可显示，也可不显示。协作图中必须显示。

顺序图中，尽力保持消息的顺序是从左到右排列的。在各对象之间，消息的次序由它们在垂直轴上的相对位置决定。

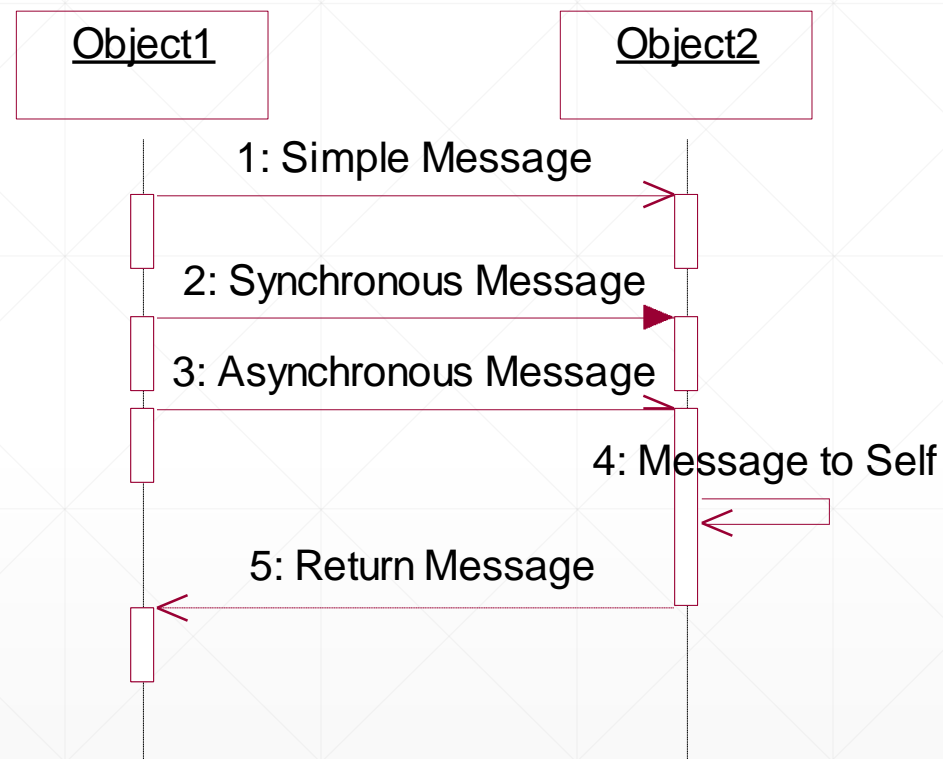
- 一个顺序图的消息流开始于左上方，消息2的位置比消息1低，这意味着消息2的顺序比消息1要迟。因为西方的阅读习惯是从左到右。

消息

在UML中，消息使用箭头来表示，箭头的类型表示了消息的类型。

进行顺序图建模时，所用到的消息主要包括以下几种类型：

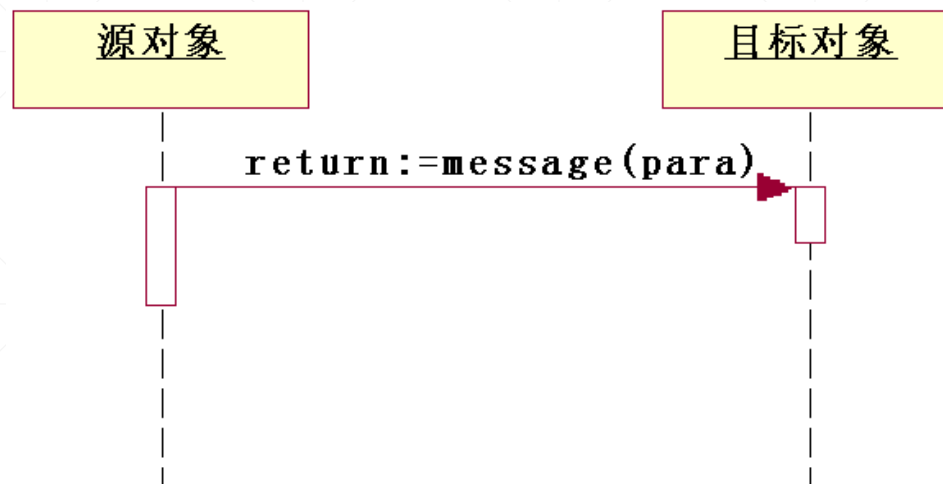
- 简单消息 (Simple Message)
- 同步消息 (Synchronous Message)
- 异步消息 (Asynchronous Message)
- 反身消息 (Message to Self)
- 返回消息 (Return Message)



同步消息

同步消息最常见的情况是调用，即消息发送者对象在它的的一个操作执行时调用接收者对象的一个操作，此时消息名称通常就是被调用的操作名称。

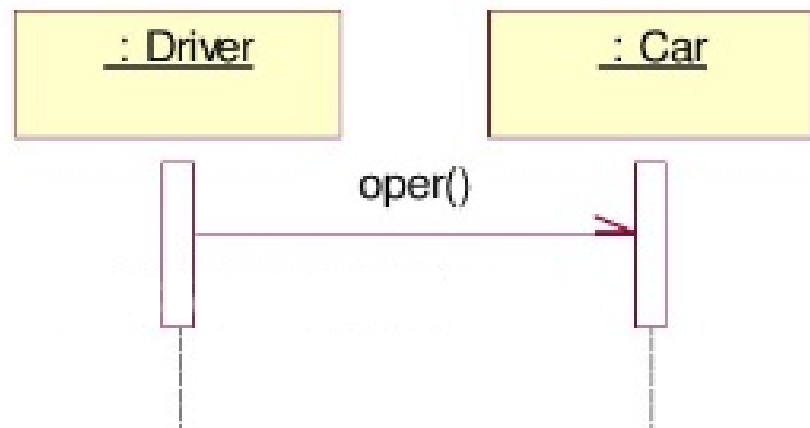
当消息被处理完后，可以回送一个简单消息，或者是隐含的返回。



异步消息

异步消息表示发送消息的对象不用等待回应的返回消息，即可开始另一个活动。

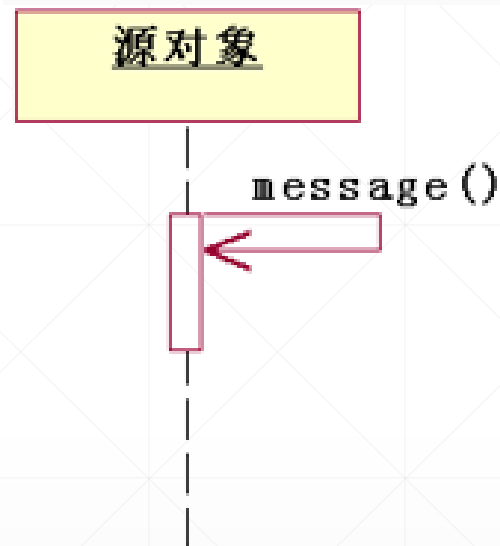
异步消息在某种程度上规定了发送方和接收方的责任，即发送方只负责将消息发送到接收方，至于接收方如何响应，发送方则不需要知道。对接收方来说，在接收到消息后它既可以对消息进行处理，也可以什么都不做。



反身消息

顺序图建模过程中，一个对象也可以将一个消息发送给它自己，这就是反身消息。

- 如果一条消息只能作为反身消息，那么说明该操作只能由对象自身的行为触发。
- 这表明该操作可以被设置为private属性，只有属于同一个类的对象才能够调用它。
- 在这种情况下，应该对顺序图进行彻底的检查，以确定该操作不需要被其他对象直接调用。

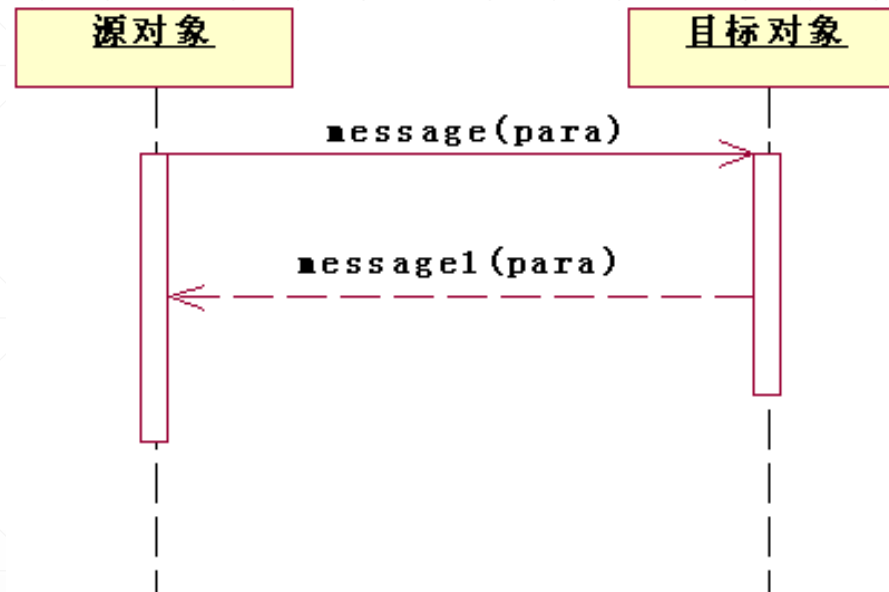


返回消息

返回消息是顺序图的一个**可选择部分**，它表示控制流从过程调用的返回。

返回消息**一般可以缺省**，隐含表示每一个调用都有一个配对的调用返回。

是否使用返回消息依赖于建模的具体/抽象程度。如果需要较好的具体化，返回消息是有用的；否则，主动消息就足够了。

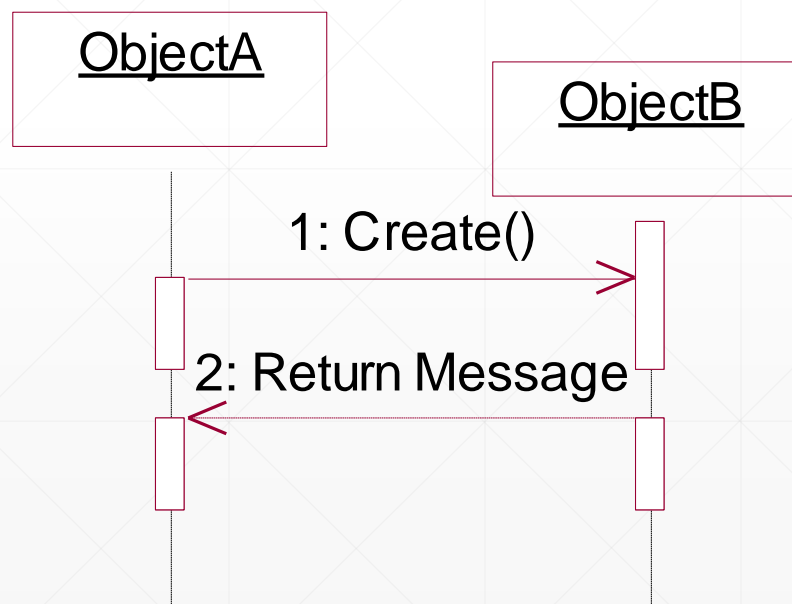
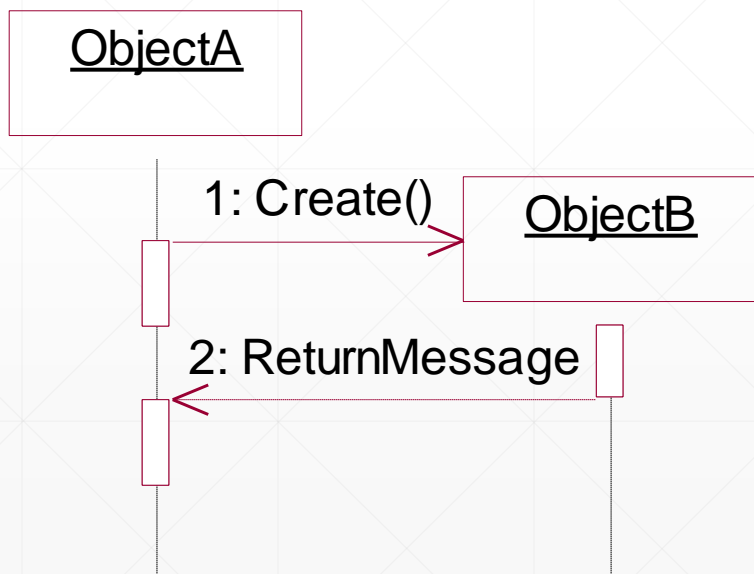


对象的创建和撤销

对象的创建有两种情况：

- 顺序图中的对象的默认位置是在图的顶部，如果对象在这个位置上，那么说明在发送消息时，该对象就已经存在了；
- 如果对象是在执行的过程中创建的，那么它的位置应该处在图的中间部分。

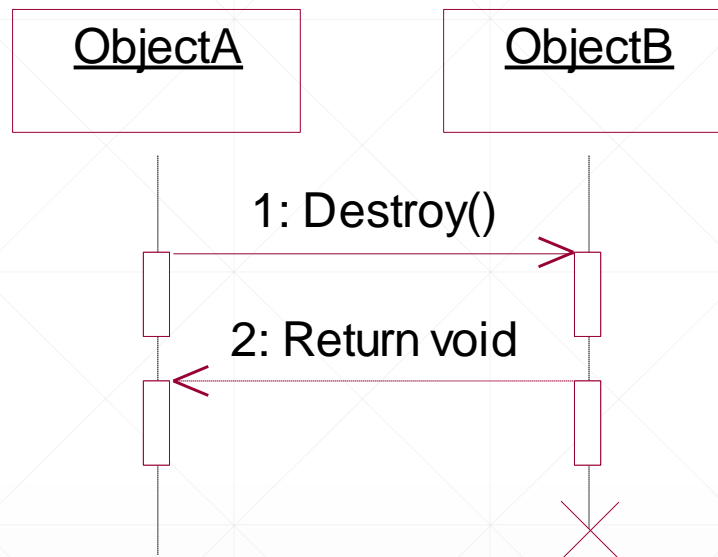
对象的创建有两种方法：



对象的创建和撤销

对象的撤销有两种情况：

- 在处理新创建的对象，或顺序图中的其他对象时，都可以发送 “destroy”消息来撤销对象。
- 要想说明某个对象被撤销，需要在被撤销对象的生命线末端放一个 “x”符号进行标识。

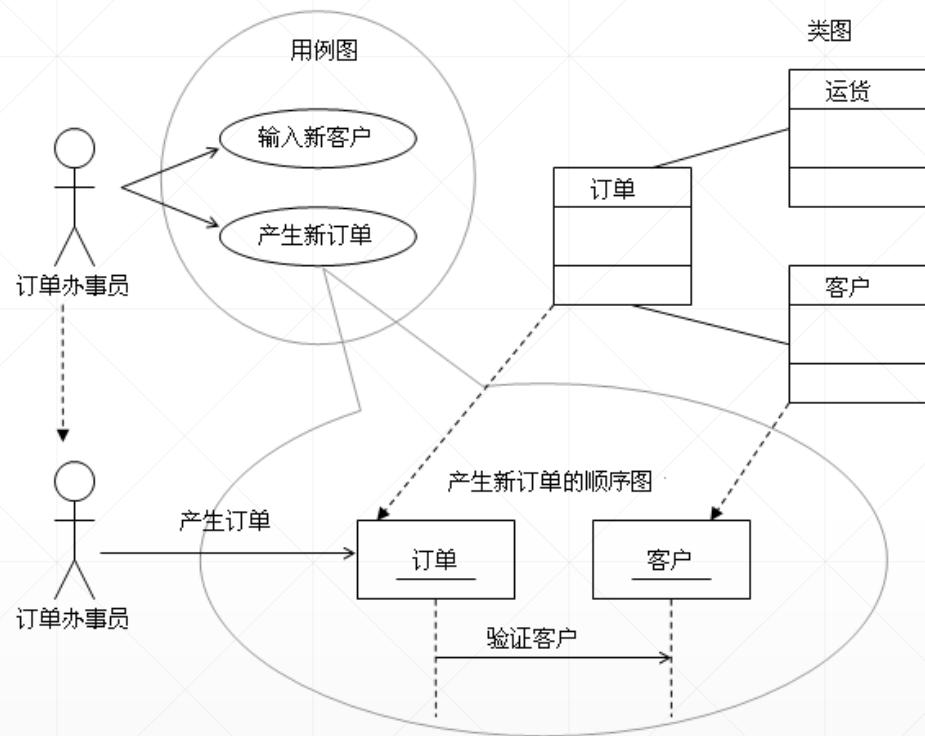


顺序图和用例

顺序图的主要用途之一是用来为某个用例的泛化功能提供其所缺乏的解释，即把用例表达的要求转化为更进一步的精细表达。

用例常常被细化为一个或多个顺序图。

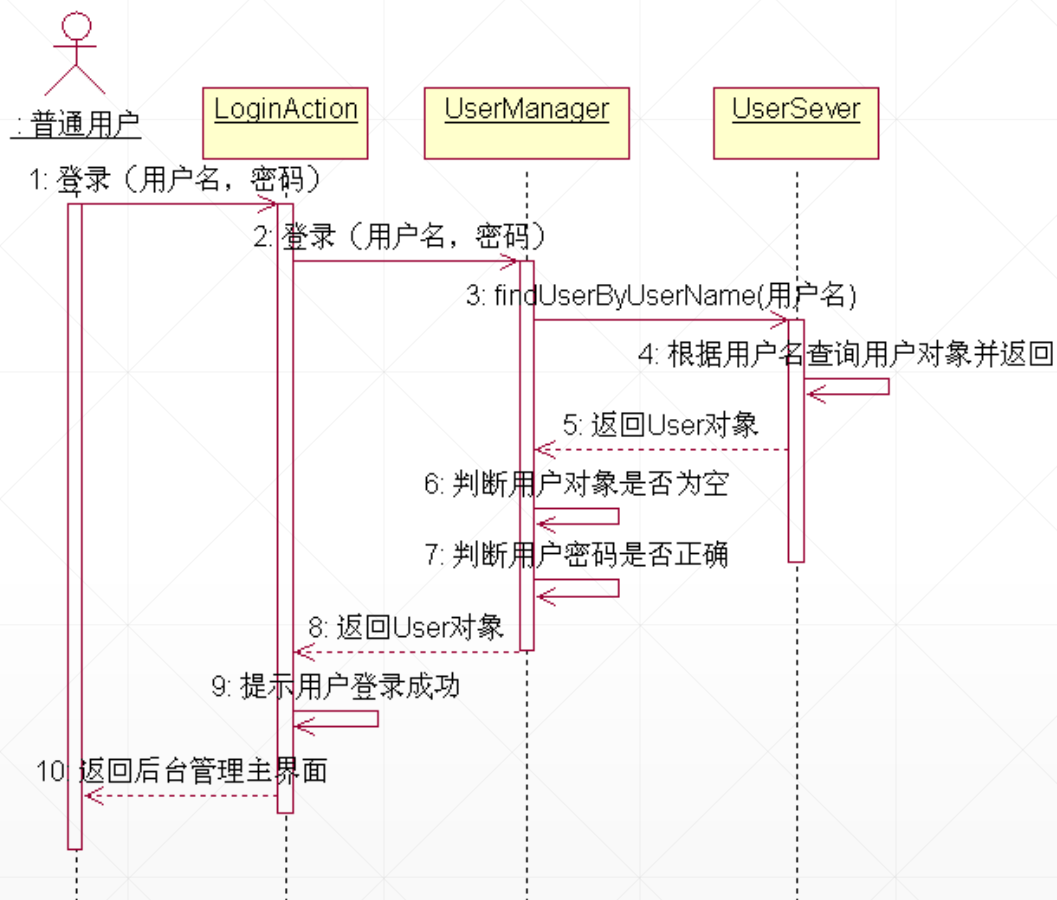
顺序图除了在设计新系统方面的用途之外，它还能用来记录一个存在于系统的对象现在如何交互。



顺序图和用例

登录用例：

- 用户将用户名和密码提交给LoginAction
- 由LoginAction调用UserManager
- UserManager到用户数据库User Server中查找用户对象并返回
- 由UserManager判断用户名是否为空、密码是否正确
- 然后将User对象返回
- 返回后台登录主界面。



顺序图建模

对系统动态行为建模的过程中，当强调按时间展开信息的传送时，一般使用顺序图建模技术。

一个单独的顺序图只能显示一个控制流。

- 一般情况下，一个完整的控制流是非常复杂的，要描述它需要创建很多交互图（包括顺序图和协作图），一些图是主要的，另一些图用来描述可选择的路径和一些例外，再用一个包对它们进行统一的管理。

顺序图建模参考策略

设置交互的语境

这些语境可以是系统、子系统、类、用例和协作的一个脚本。

对象从左到右

识别对象在交互语境中所扮演的角色，根据对象的重要性及相互关系，将其从左至右放置在顺序图的顶部。

设置每个对象的生命线

通常情况下，对象存在于整个交互过程中，但它们也可以在交互过程中创建和撤销。对于这类对象，在适当的时刻设置它们的生命线，并用适当的构造型消息显示地说明它们的创建和撤销。

消息自上而下

从引发某个消息的信息开始，在生命线之间画出从顶到底依次展开的消息，显示每个消息的内容标识。

设置对象的激活期

可视化消息的嵌套或可视化实际计算发生时的时间点。

时间和空间约束

如果需要设置时间或空间的约束，可以为每个消息附上合适的时间和空间约束。

前置和后置条件

如果需要形式化的说明某控制流，可以为每个消息附上前置和后置条件。

建立顺序图的步骤

确定交互的范围；

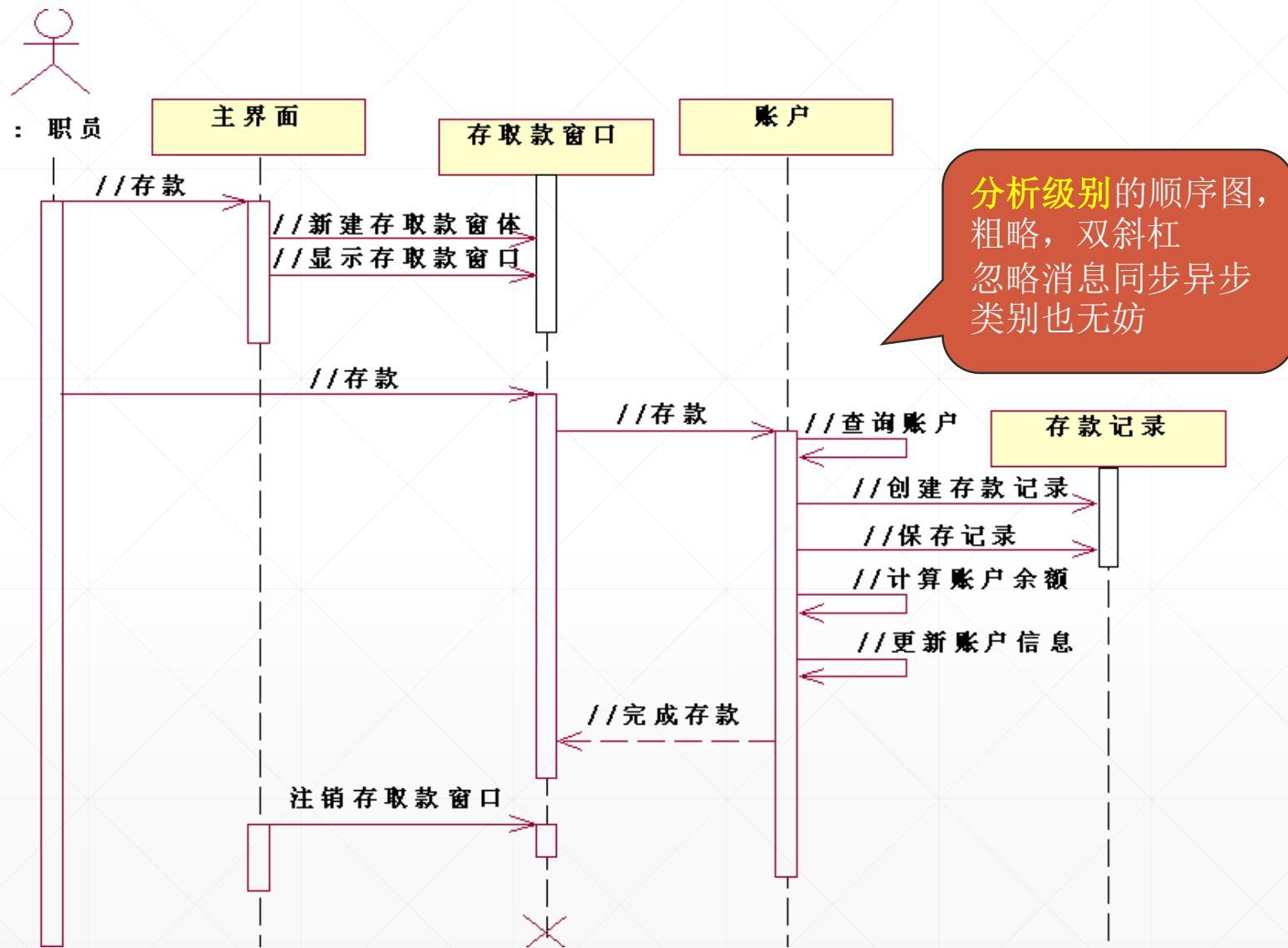
识别参与交互的
对象和活动者；

设置对象生命线
开始和结束；

设置消息；

细化消息。

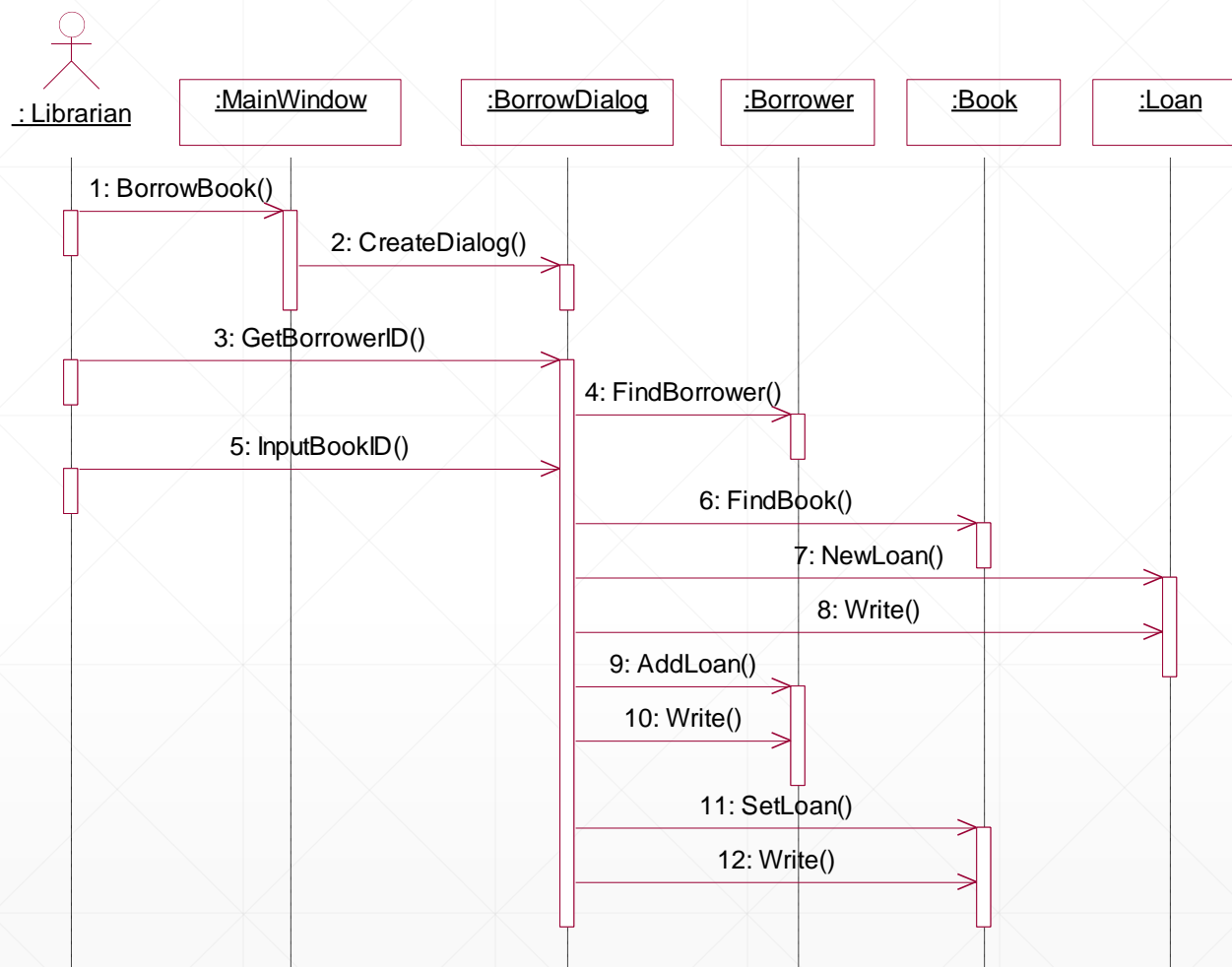
顺序图示例：存款用例的顺序图



顺序图示例：借阅图书用例的顺序图

借阅图书的过程为：

- 图书管理员选择菜单项“借阅图书”，弹出BorrowDialog对话框；
- 图书管理员在该对话框中输入借阅者信息
- 然后由系统查询数据库，以验证该借阅者的合法性
- 若借阅者合法，则再由图书管理员输入所要借阅的图书信息，系统记录并保存该借阅信息。



4.3.4 面向对象的设计原则

面向对象设计的特点



面向对象设计强调定义软件对象，并且使这些软件对象相互协作来满足用户需求。

面向对象分析和设计的界限是模糊的，从面向对象分析到面向对象设计是一个逐渐扩充模型的过程。分析的结果通过细化直接生成设计结果，在设计过程中逐步加深对需求的理解，从而进一步完善需求分析的结果。

分析和设计活动是一个反复迭代的过程。

面向对象方法学在概念和表示方法上的一致性，保证了各个开发阶段之间的平滑性。

面向对象设计的四个层次

确定系统的总体结构和风格，构造系统的物理模型，将系统划分成不同的子系统。



中层设计：对每个用例进行设计，规划实现用例功能的关键类，确定类之间的关系。



进行底层设计：对每个类进行详细设计，设计类的属性和操作，优化类之间的关系。



补充实现非功能性需求所需要的类。

注意点

面向对象设计与结构化设计的过程和方法完全不同，要设计出高质量的软件系统，记住：

- 对接口进行设计
- 发现变化并且封装它
- 先考虑聚合然后考虑继承

强内聚

类内聚——设计类的原则是一个类的属性和操作全部都是完成某个任务所必须的，其中不包括无用的属性和操作。

- 例如设计一个平衡二叉树类，该类的目的就是要解决平衡二叉树的访问，其中所有的属性和操作都与解决这个问题相关，其他无关的属性和操作在这里都是垃圾，应该清除。

弱耦合

在面向对象设计中，耦合主要指不同对象之间相互关联的程度。如果一个对象过多地依赖于其它对象来完成自己的工作，则不仅使该对象的可理解性下降，而且还会增加测试、修改的难度，同时降低了类的可重用性和可移植性。

对象不可能是完全孤立的，当两个对象必须相互联系时，应该通过类的公共接口实现耦合，不应该依赖于类的具体实现细节。

耦合方式

交互耦合

- 如果对象之间的耦合是通过消息连接来实现的，则这种耦合就是交互耦合。在设计时应该尽量减少对象之间发送的消息数和消息中的参数个数，降低消息连接的复杂程度。

继承耦合

- 继承耦合是一般化类与特殊化类之间的一种关联形式，设计时应该适当使用这种耦合。在设计时要特别认真分析一般化类与特殊化类之间继承关系，如果抽象层次不合理，可能会造成对特殊化类的修改影响到一般化类，使得系统的稳定性降低。另外，在设计时特殊化类应该尽可能多地继承和使用一般化类的属性和服务，充分利用继承的优势。

可重用性

软件重用是从设计阶段开始的，所有的设计工作都是为了使系统完成预期的任务，为了提高工作效率、减少错误、降低成本，就要充分考虑软件元素的可重用性。可重用性有两个方面的含义：

- 尽量使用已有的类，包括开发环境提供的类库和已有的相似的类；
- 如果确实需要创建新类，则在设计这些新类时考虑将来的可重用性。

设计一个可重用的软件比设计一个普通软件的代价要高，但是随着这些软件被重用次数的增加，分摊到它的设计和实现成本就会降低。

框架

框架是一组可用于不同应用的类的集合。框架中的类通常是一些抽象类并且相互有联系，可以通过继承的方式使用这些类。

- 例如，Java应用程序接口（API）就是一个成功的框架包，为众多的应用提供服务，但一个应用程序通常只需要其中的部分服务，可以采用继承或聚合的方式将应用包与框架包关联在一起来获得需要的服务。

一般不会直接去修改框架的类，而是通过继承或聚合为应用创建合适的GUI类。

谢谢！
