Southeast University

软件工程导论

廖力

lliao@seu.edu.cn

课程结构

Unit1. 软件工程概述

Unit2. 软件工程技术

一、系统工程

二、需求工程

三、设计工程

（一）系统设计的任务和目标

（二）设计相关概念

（三）体系结构设计

（四）构件级设计

（五）设计建模工具

四、软件构建与测试

Unit3. 软件项目管理

（一）系统设计的任务和目标

1. 从需求分析到系统设计

- 需求分析阶段，我们明确了要“做什么”
  - 分析模型关注于说明必须的数据、功能和行为
- 系统设计阶段，我们将解决“怎么做”
  - 设计模型提供了软件的体系结构、数据结构、接口和构件的细节
  - 设计模型可以直接转化为系统实现

（一）系统设计的任务和目标

2. 软件工程中的设计

- 1）软件体系结构设计的任务
- 定义系统的主要结构元素及相互的关系
  - 结构化方法： 从数据流图出发对数据进行分析，得出软件的层次化的模块结构图
  - 面向对象方法：从分析模型出发划分子系统，在考虑通信、并发、 部署、复用等问题的基础上，建立系统层次结构

（一）系统设计的任务和目标

2. 软件工程中的设计

- 2）数据结构设计的任务：
- 设计软件实现所要求的数据结构

  - 结构化方法：从分析阶段得到的数据模型和数据字典出发，设计出相应的数据结构

  - 面向对象方法：数据作为类（通常为实体类）的一个属性，设计合适的数据结构，来表示这个属性

    – 对象的数据结构的设计，基本上已经属于构件级设计阶段

（一）系统设计的任务和目标

2. 软件工程中的设计

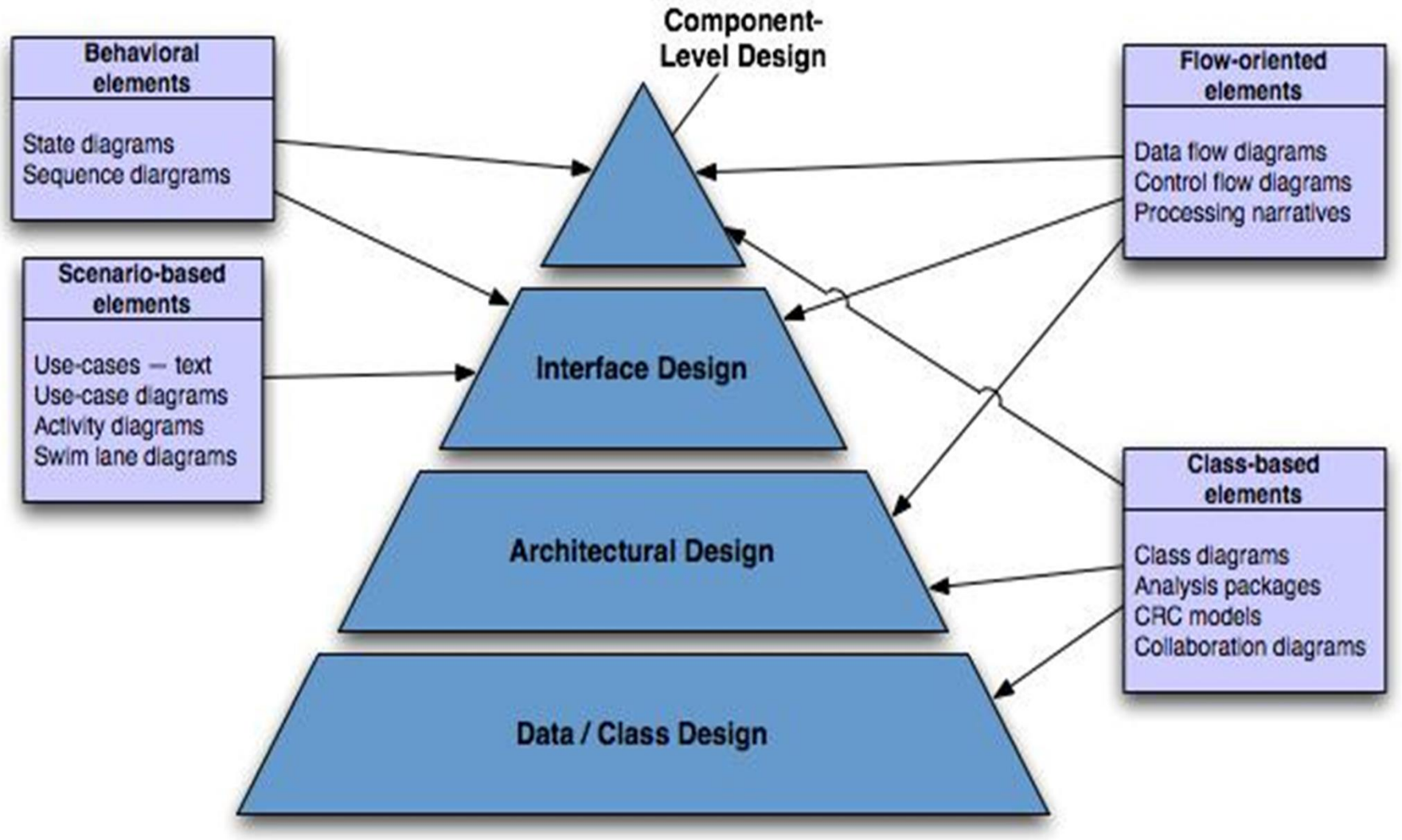
- 3）接口设计的任务：
- 描述系统内部、系统与系统之间以及系统与用户之间如何通信
  - 结构化方法：接口包括了信息交互和特定的行为，因此，数据流和控制是接口设计的基础
  - 面向对象方法：接口设计主要是消息设计，使用场景和行为模型为接口设计提供信息。
    - 消息设计的细化将在构件级设计阶段进行。

（一）系统设计的任务和目标

2. 软件工程中的设计

- 4）构件级设计的任务
- 将软件体系结构的结构元素变化为对软件构件的过程性描述
  - 结构化方法：从分析阶段获得的过程规格说明、控制规格说明和状态图出发，得到系统各个功能的过程化描述
  - 面向对象方法：从基于类的模型和行为模型出发，得到各个类和对象的方法以及实现细节的描述。

（一）系统设计的任务和目标



（一）系统设计的任务和目标

- 3.良好的系统设计必须具备的三个特征：
  - 设计要包含分析中获得的用户明显的和隐含的需求；
  - 对编码、测试和维护人员来说，设计是可读、可理解的；
  - 设计提供了系统完整的面貌，涉及从实现角度看的数据、功能、行为
- 以上三个方面，也就是系统设计的目标

（一）系统设计的任务和目标

- 4.系统设计基本步骤
- 1）制定设计规范
  - 阅读系统需求说明书，确定设计目标及其优先顺序
  - 确定设计方法：结构化、面向对象...
  - 设计文档的编制标准
  - 基本的实现规范：代码的信息形式、与硬件及操作系统的接口规约、命名规则

（一）系统设计的任务和目标

- 4.系统设计基本步骤
- 2）体系结构设计
  - 面向结构的设计：划分系统层次结构，确定功能模块，确定模块调用关系及接口。
  - 面向对象的设计：对子系统进行设计，定义出若干个一致的类与对象、关系、行为、功能的集合。

（一）系统设计的任务和目标

- **4.系统设计基本步骤**
  - **3）数据结构设计**
  - **4）可靠性设计（可靠性、质量、易于维护）**
  - **5）编写设计文档**
    - 概要设计说明书、数据库设计说明书、初步测试计划等。
  - **6）设计评审**
    - 可追溯性、接口、风险、实用性、技术清晰度、可维护性、质量、可选方案等方面
  - **7）构件设计**

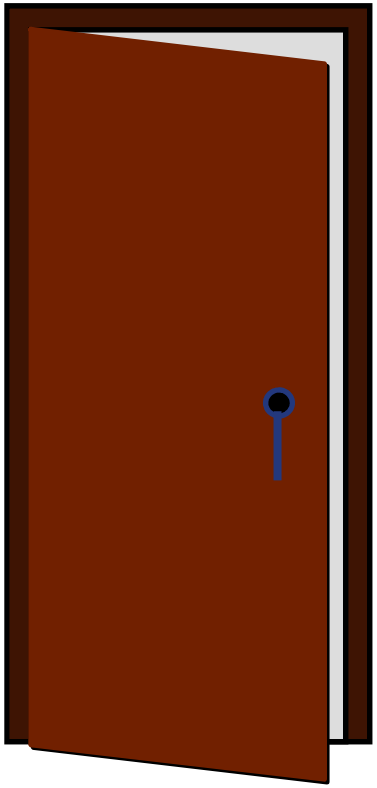
（二）设计相关概念

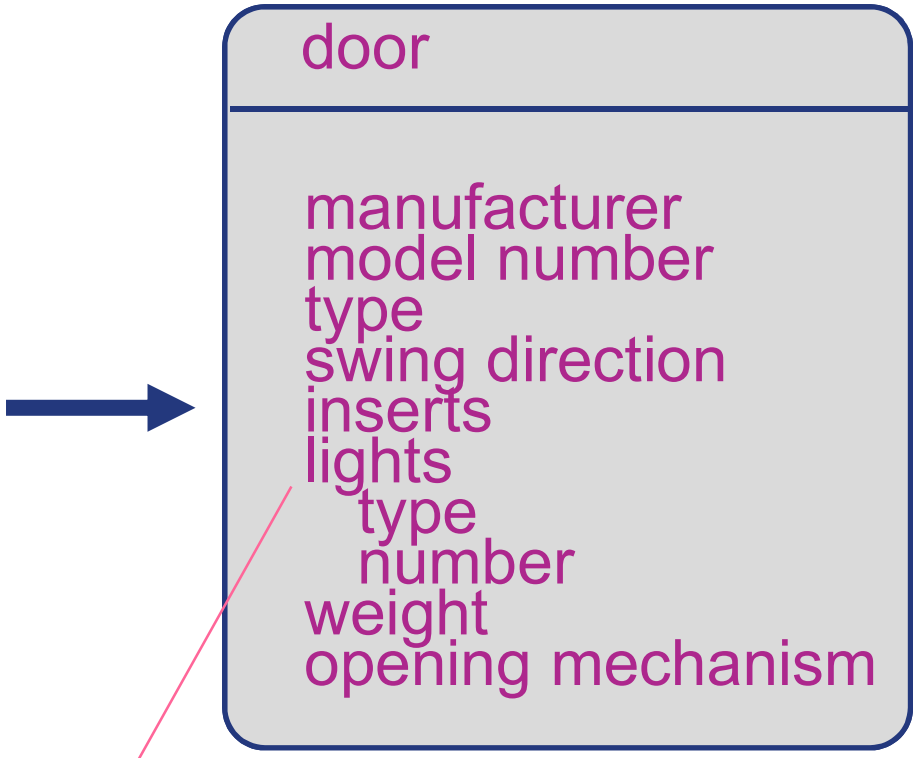
- 1、抽象（abstraction）
- 2、体系结构（architecture ）
- 3、模式（patterns）
- 4、模块化（modularity）
- 5、信息隐藏（information hiding）
- 6、功能独立（functional independence）
- 7、精化（refinement）
- 8、重构（refactoring）

（二）设计相关概念——1. 抽象 Next

- 数据抽象 Go
  - 定义数据类型和施加于该类型对象的操作
  - 限定了对象的取值范围，只能通过这些操作修改和观察数据
- 过程抽象：功能角度的抽象 Go
  - 将功能体作为单个功能看待
  - 该功能体实际上是由一系列更低级的功能或代码来实现的

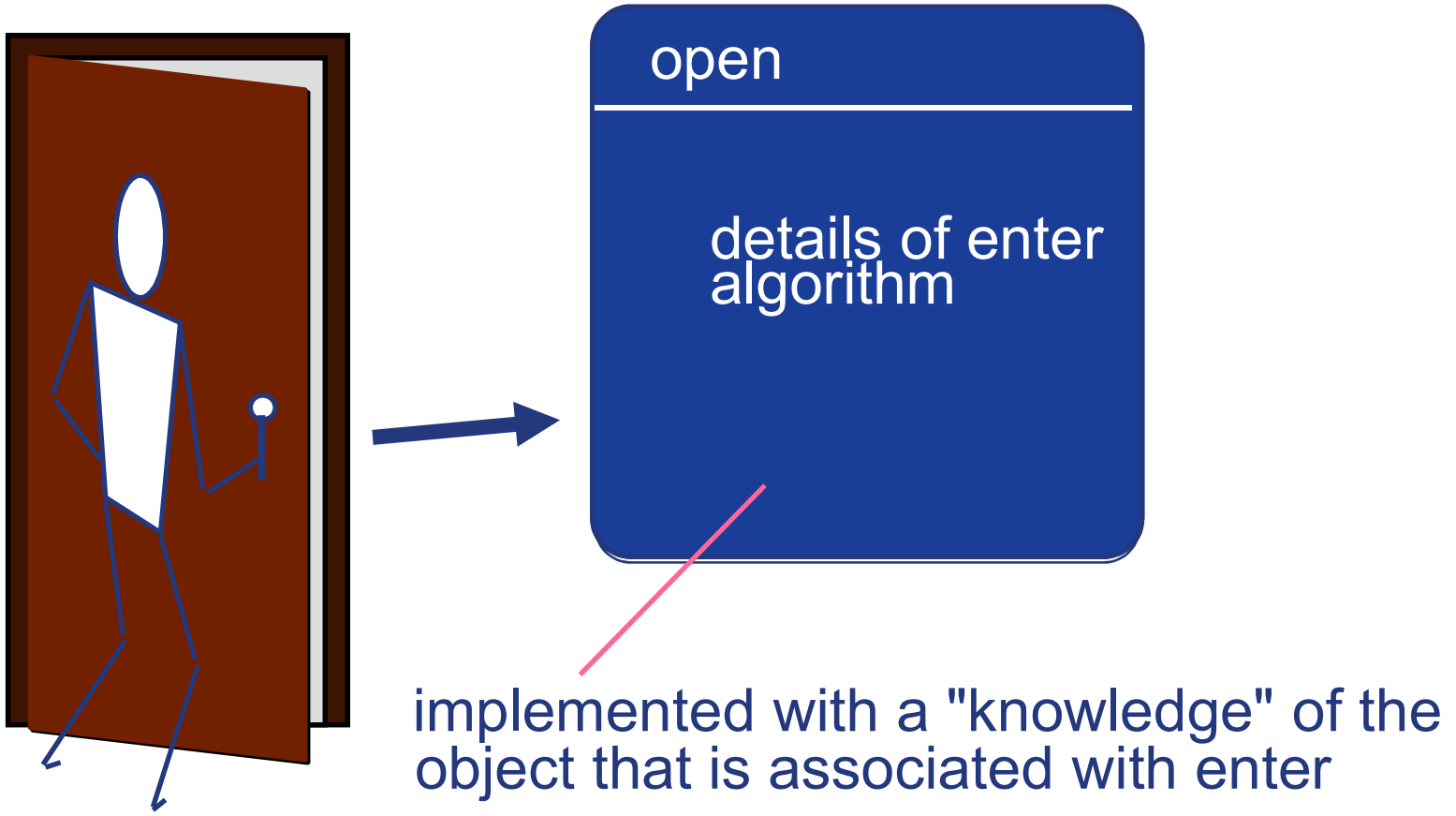
Data Abstraction Back





implemented as a data structure

Procedural Abstraction Back



（二）设计相关概念——2. 体系结构

- 软件体系结构关注系统的一个或多个结构，包含软件部件、部件对外可见的属性以及部件间的关系
- 体系结构的作用
  - 方便利益相关人员的交流
  - 有利于系统设计的前期决策
  - 可传递、易于理解的系统级抽象

（二）设计相关概念——2. 体系结构

- 体系结构反映了软件系统实现的高层方案
  - 软件系统越来越复杂，需要将其划分为若干部分分而治之—模块化
    - 不同的小组或开发者负责不同的部分
    - 然后在系统层面上进行集成
  - 负责不同部分的开发者对于其它模块需要了解的信息越少越好—抽象与信息隐藏
  - 这些部分之间还需要定义清晰、明确的接口—接口设计

（二）设计相关概念——3. 模式

- 每个模式都描述了一个在我们所处环境内反复发生的问题，描述了该问题的核心解决方案。
- 模式的定义便于复用。
- 模式的类型
  - 结构模式 （**Architecture Pattern**）
  - 设计模式 （**Design Pattern**）
  - 编码模式 （**Idiom**）

（二）设计相关概念——3. 模式

❖结构模式（Architectural Patterns）

- 表达了软件系统的基本结构组织形式或者结构方案
- 它包含一组预定义的子系统，规定了这些子系统的责任，同时还提供了用于组织和管理这些子系统的规则和向导
- 如：**MVC（Model View Controler）**
- 再如：软件体系结构风格

（二）设计相关概念——3. 模式

❖设计模式（Design Patterns）

- 为软件系统的子系统、组件或者组件之间的关系提供一个精炼之后的解决方案
- 它描述了在特定环境下，用于解决通用软件设计问题的组件以及这些组件相互通信时的可重现结构
- 如：适配器模式
  - 适配器模式使得原本由于接口不兼容而不能一起工作的那些东西可以一起工作。
  - 如：变压器

（二）设计相关概念——3. 模式

编码模式（Idiom）

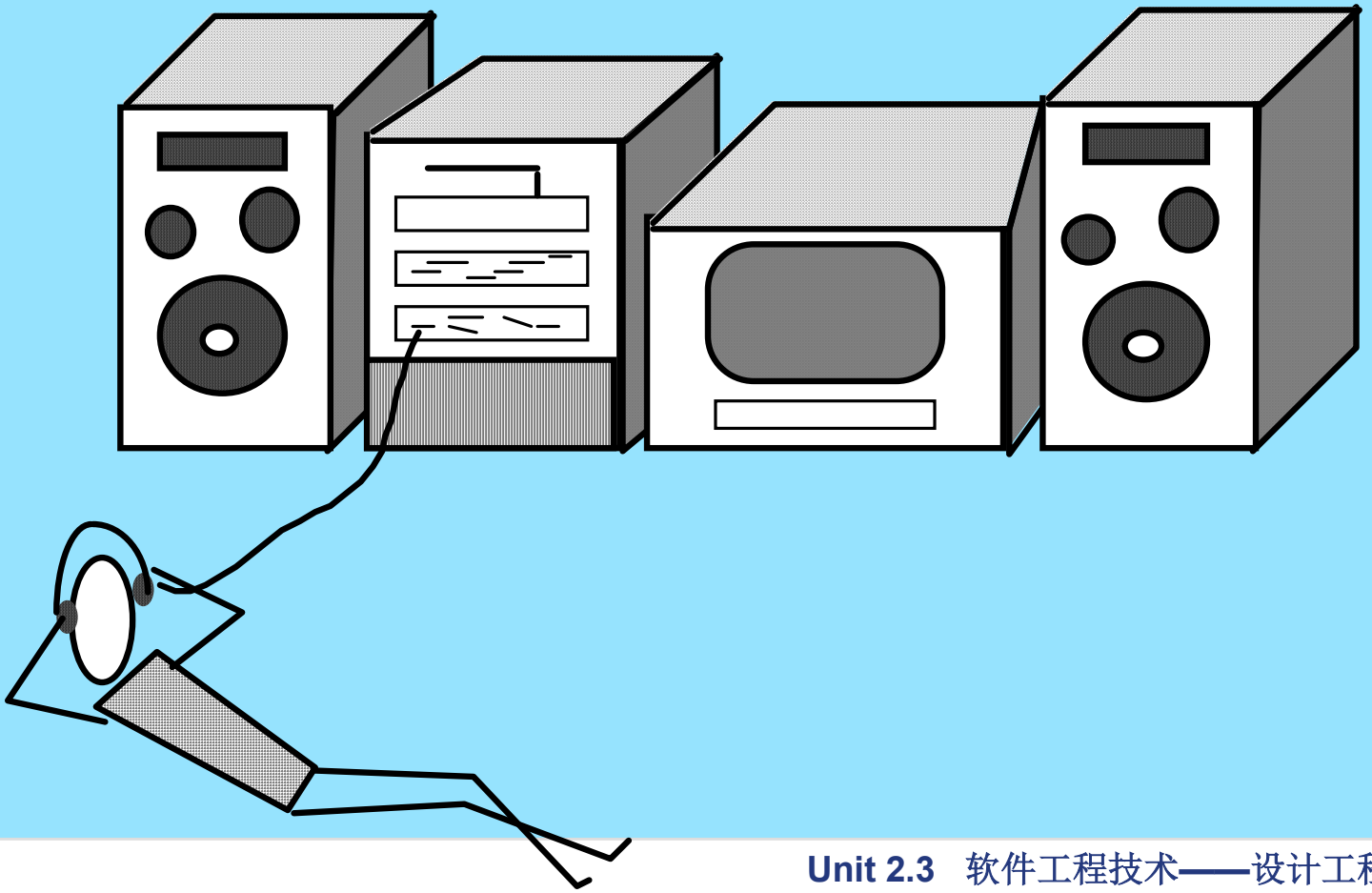
- **An idiom is a low-level pattern specific to a programming language.**
- **An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.**

（二）设计相关概念——4. 模块化

- 按照设计原则将系统划分为若干个较小的模块（构件）
  - 相互独立但又相互关联
  - 实际上是系统分解和抽象的过程
- 模块是相对独立的程序体
  - 独立命名的，并且可以通过名字来访问
    - 例如：过程、函数、子程序、宏等

（二）设计相关概念——4. 模块化

easier to build, easier to change, easier to fix ...



（二）设计相关概念——4. 模块化

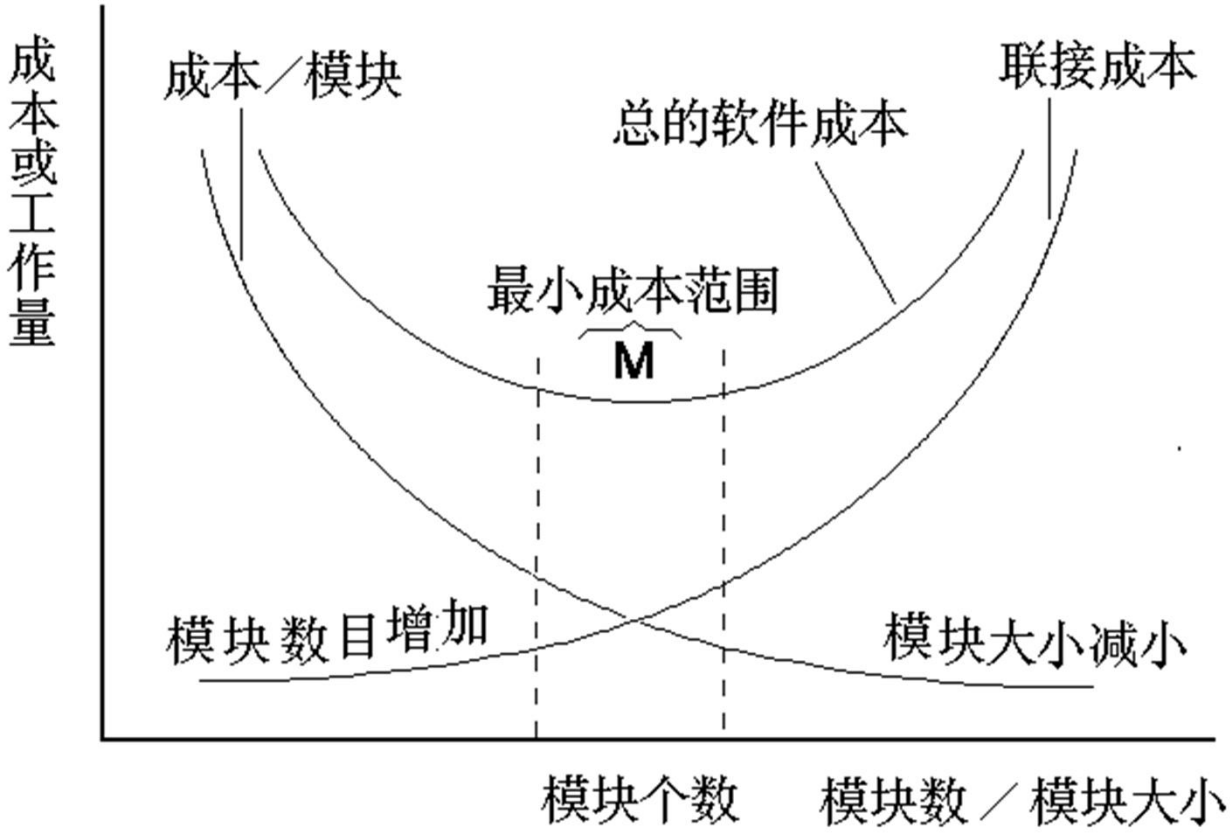
- 可通过模块化降低开发复杂度
  - **C(x)：问题x的复杂性**
  - **E(x)：解决问题x所需工作量**
  - 对于两个问题**p1** 和**p2**：
  - **1) 如果C(p1)>C(p2) 那么 E(p1)>E(p2)**
    - 问题越复杂解决问题所需要的花费更多
  - **2) C(p1+p2)>C(p1)+C(p2)因此E(p1＋p2)>E(p1)+E(p2)**
    - 将复杂问题分解成可以多个子问题分别解决会更加容易(模块化思想的依据)

（二）设计相关概念——4. 模块化

- 如果我们无限制地划分软件，开发它所需的工作量会变得小到可以忽略?
- 结论：**NO**
  - 模块数量增加时，只是使各个子模块的工作量之和有所减小
  - 然而开发工作量还有很大一部分来自：
    - 模块间的接口和集成
    - 人与人之间的沟通
  - 集成和沟通的开销到了一定程度就会成为开发工作量的主要部分

（二）设计相关概念——4. 模块化

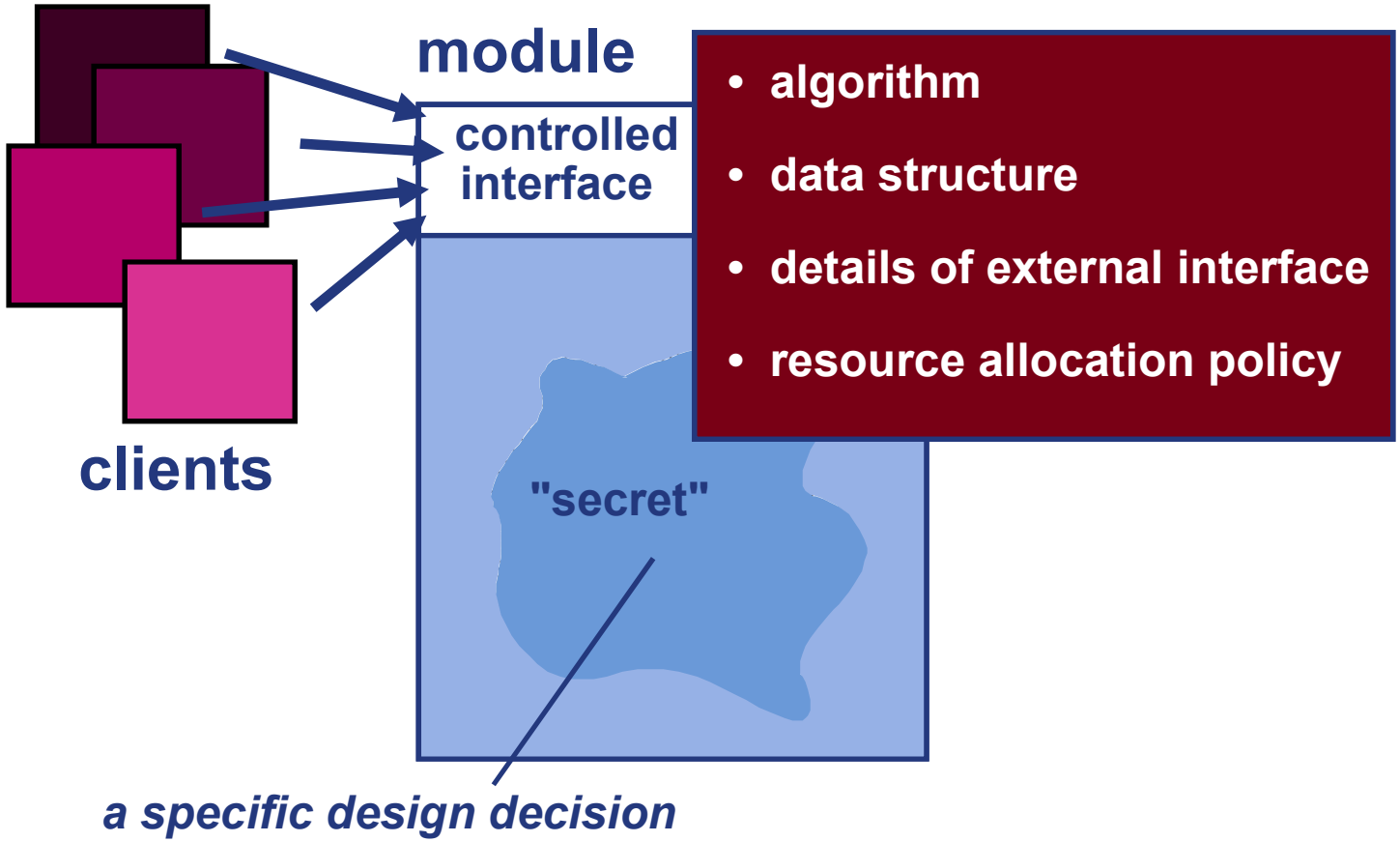
- 模块数增加时，模块间的关系也随之增加，接口和集成的工作量也随之增加
- 结论：寻找最佳模块化程度平衡点



（二）设计相关概念——5. 信息隐藏

- 每个模块都尽量对其他模块隐藏自己的内部实现细节
  - 模块内部的数据和过程不允许其它不需要这些信息的模块使用
  - 定义和实施对模块的过程细节和局部数据结构的存取限制
- 信息隐藏是实现抽象/模块化机制的基本支撑

（二）设计相关概念——5. 信息隐藏



（二）设计相关概念——6. 功能独立

- 1）功能独立（模块独立）是模块化的根本要求
  - 模块完成独立的功能：明确可辨识
  - 高内聚：内部结构紧密
  - 低耦合：模块间关联和依赖程度尽可能小，与其他模块的接口简单
  - 符合信息隐蔽和信息局部化原则

（二）设计相关概念——6. 功能独立

2）模块独立的重要性

- 模块的开发者专注于某一个相对独立的部分
- 不用过多关心其他模块
- 修改和**bug**所影响的范围被局部化
- 单个模块更容易复用
- 独立的模块更易于维护和测试

（二）设计相关概念——6. 功能独立

3）模块独立性的指标

- 内聚度与耦合度
- 内聚(cohesion)：一个模块内部各个元素彼此结合的紧密程度

  —尽量高
- 耦合(coupling)：模块之间相互关联的程度

  —尽量低

（二）设计相关概念——6. 功能独立

❖3）模块独立性的指标

- 内聚度 （由高到低排列）

高 ← 内聚度 ← 低

功能内聚	分层内聚	通信内聚	顺序内聚	过程内聚	时间内聚	实用内聚

强（功能单一）←——模块独立性 ←—— 弱（功能分散）

（二）设计相关概念——6. 功能独立

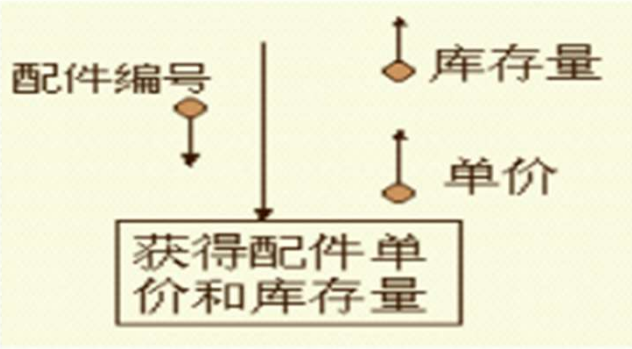
3）模块独立性的指标:内聚度

- 功能内聚 （**Functional cohesion**）
  - 指一个模块中各个部分都是为完成一项具体功能而协同工作，紧密联系，不可分割的(单个功能)
  - 判断一个模块是不是功能内聚：
    - 若需要完成的功能中含“和”，“或”等词，它一定不够功能内聚。
- 分层内聚（**Layer cohesion**）
  - 高层能访问低层的服务，但低层不能访问高层。

（二）设计相关概念——6. 功能独立

3）模块独立性的指标:内聚度

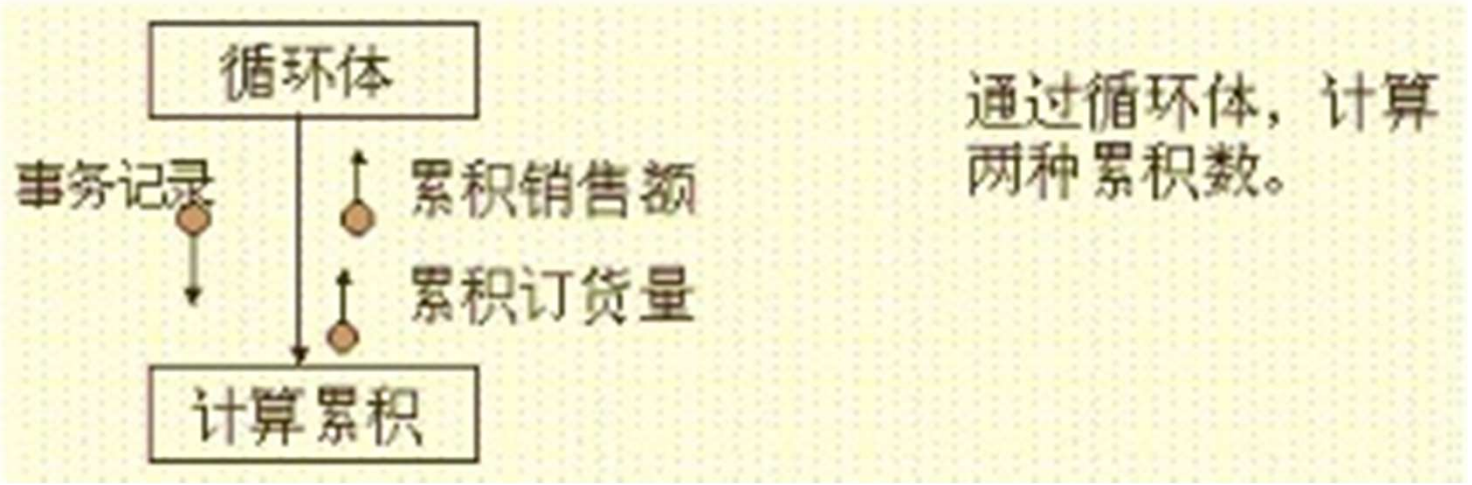
- 顺序内聚 （**Sequential cohesion**）
  - 一个模块内部各个组成部分必须顺序执行，前一个动作的输出是后一个动作的输入
- 通信内聚（**communicational cohesion**）
  - 模块内各组成部分的处理动作都使用相同的输入数据或者具有相同的输出数据



（二）设计相关概念——6. 功能独立

❖3）模块独立性的指标:内聚度

- 过程内聚 （**Procedural cohesion**）
  - 模块内的多个任务，即使动作各不相同，也没有相互联系，但它们都受一个控制流支配，必须按指定的过程执行



（二）设计相关概念——6. 功能独立

- 3）模块独立性的指标:内聚度
  - 暂时内聚（**Temporal cohesion**，时间内聚）
    - 模块内各部分的处理与时间相关
    - 时间内聚模块中的各项处理必须在一定时间内完成
    - 如：启动时要进行的全部操作
  - 实用内聚（**Utility cohesion**）
    - 在一个模块中，但是其他方面都不相关的构件、类或操作被分为一组。
    - 如：**Statistics**类中包含计算**6**个简单统计所需的所有属性和操作

（二）设计相关概念——6. 功能独立

❖3）模块独立性的指标

- 耦合度 （由高到低排列）

低 ← 耦合度 → 高

非直接耦合	数据耦合	标记耦合	控制耦合	外部耦合	共用耦合	内容耦合

强（功能单一）← 模块独立性 → 弱（功能分散）

（二）设计相关概念——6. 功能独立

❖3）模块独立性的指标:耦合度

- 内容耦合（**Content coupling**）
  - 一个模块可以直接访问另一个模块的内部数据或内部功能
  - 如：

```
模块A：
…
L1: …
…
```

```
模块B：
…
GOTO L1
…
```

（二）设计相关概念——6. 功能独立

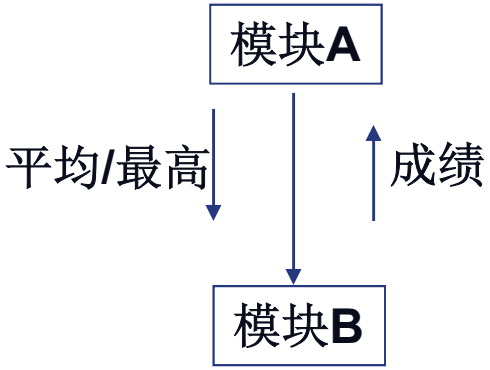
❖3）模块独立性的指标:耦合度

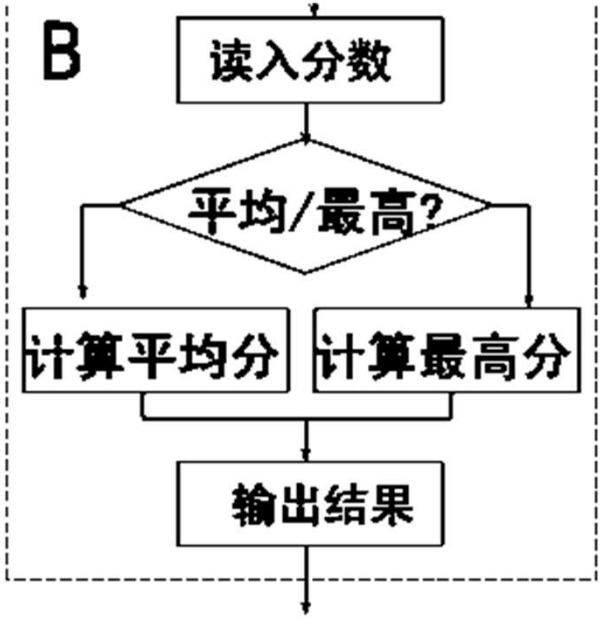
- 共用耦合（**Common coupling,**公共耦合）
  - 多个模块共同访问某些公共数据环境
  - 好处：便利性（如可设公共缺省值）
  - 缺点：当这种耦合需要变更时，无法判定有多少模块属于共用耦合，从而造成错误蔓延
- 外部耦合（**External coupling**）
  - 一组模块都访问同一全局简单变量而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息

（二）设计相关概念——6. 功能独立

❖3）模块独立性的指标:耦合度

- 控制耦合（Control coupling）
  - 模块间的交互参数包含控制信息，可影响另一个模块的执行逻辑
  - 控制信息不同于数据信息





调用逻辑性模块B时，须先传递控制信号(平均分/最高分)，以选择所需的操作。控制模块必须知道被控模块的内部逻辑，增强了相互依赖。

（二）设计相关概念——6. 功能独立

3）模块独立性的指标:耦合度

- 标记耦合（**stamp coupling**）
  - 模块间传递特定的数据结构
  - 如高级语言的数组名,记录名,文件名等这些名字（标记）,其实传递的是该数据结构的地址
  - 缺点：各模块都必须清楚该记录的结构，并按结构要求对该记录进行操作。
- 数据耦合（**data coupling**）
  - 模块间仅通过参数表传递简单数据
  - 但是要注意若参数列表过长则会带来测试和维护困难。

（二）设计相关概念——6. 功能独立

❖3）模块独立性的指标:耦合度

- 非直接耦合：两个模块可以相对独立工作
  - 例程调用耦合（**Routine call coupling**）
    - 一个操作调用另外一个操作
    - 通过减少例程总量来减少此种耦合
  - 类型使用耦合（**Type use coupling**）
    - 构件**A**使用了在构件**B**中定义的一个数据类型
    - 如：一个类将其某变量声明为另一个类的类型
  - 包含或导入耦合（**Inclusion or import**）
    - 构件**A**导入或者包含构件**B**的包或者内容

（二）设计相关概念——6. 功能独立

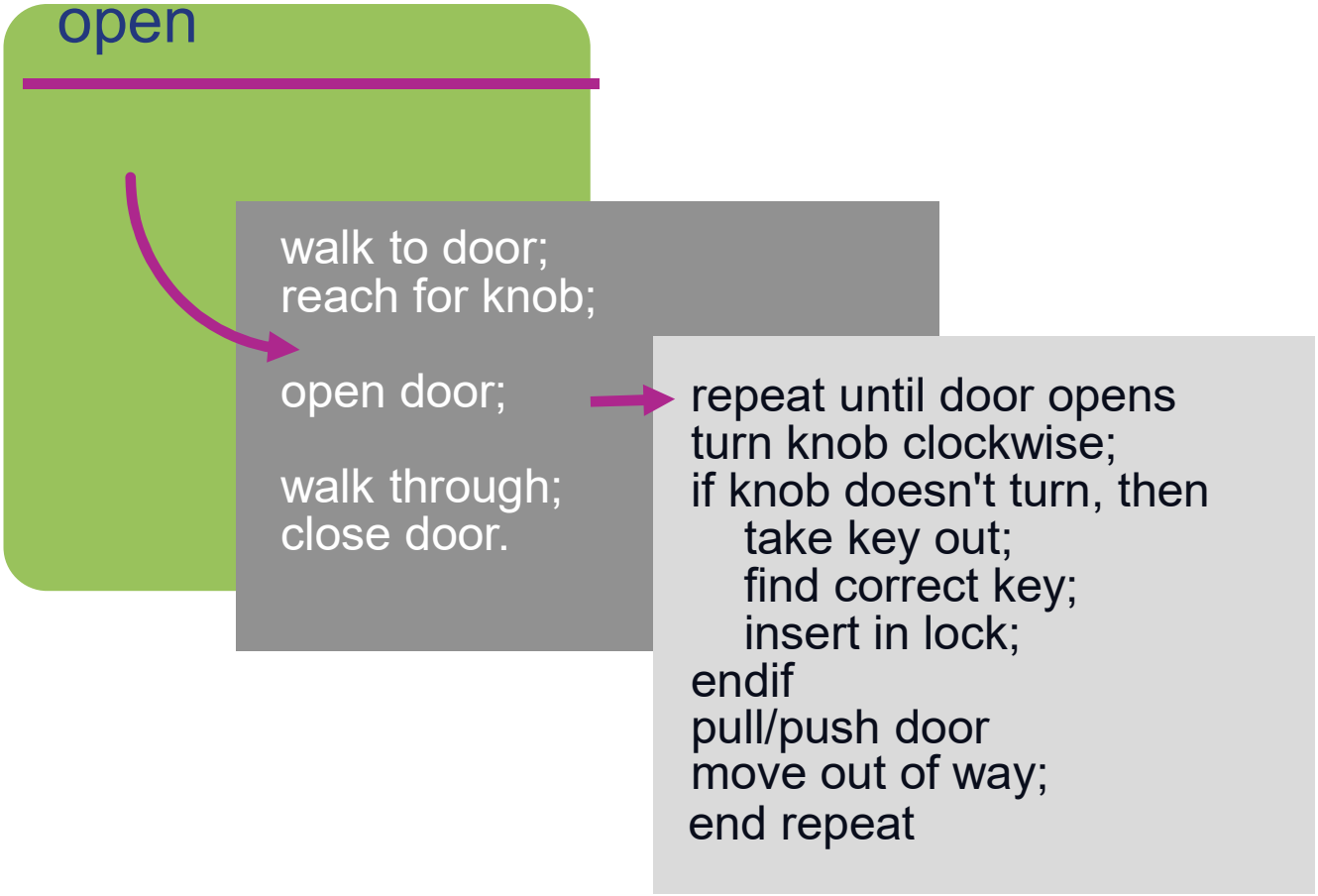
- 具有模块独立性的模块：高内聚低耦合的模块
  - 一个模块内部各个元素之间的联系越紧密，则它的内聚性就越高，模块独立性也越强
  - 模块之间的连接越紧密，联系越多，耦合性就越高，而其模块独立性就越弱

（二）设计相关概念——7. 精化

- 把问题的求解过程分解成若干步骤或阶段，每步都比上步更精化，更接近问题的解法
- 常与分层抽象的思想相结合
  - 抽象使得设计者能够描述过程和数据而忽略低层的细节
  - 求精有助于设计者在设计过程中揭示低层的细节
  - 高层抽象将在下层不断精化，最终得到软件实现

（二）设计相关概念——7. 精化

- 逐步精化（**Stepwise Refinement**）



（二）设计相关概念——8. 重构

- **"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”**

——Fowler

（二）设计相关概念——8. 重构

- 重构的目的是去发现和修改：
  - **redundancy**
  - **unused design elements**
  - **inefficient or unnecessary algorithms**
  - **poorly constructed or inappropriate data structures**
  - **or any other design failure that can be corrected to yield a better design.**

（二）设计相关概念——总结

- 1、抽象（abstraction）
- 2、体系结构（architecture ）
- 3、模式（patterns）
- 4、模块化（modularity）
- 5、信息隐藏（information hiding）
- 6、功能独立（functional independence）
- 7、精化（refinement）
- 8、重构（refactoring）

（三）体系结构设计

- 1. 什么是软件体系结构？
- **The software architecture of a program or computing system is the structure or structures of the system, which comprise the software components, the externally visible properties of those components, and the relationships among them.**

— Bass. et al.

（三）体系结构设计

2. 为什么要使用软件体系结构？

- **Architecture is a representation of a system that enables the software engineer to:**
  - **analyze the effectiveness of the design in meeting its stated requirements,**
  - **consider architectural alternatives at a stage when making design changes is still relatively easy, and**
  - **reduce the risks associated with the construction of the software.**

（三）体系结构设计

3. 体系结构设计的重要性

- 软件系统设计是开发者之间的分工和合作和基础，它能够方便利益相关人员的交流
- 有利于系统设计的前期决策，设计方案是决定系统质量的主要因素
- 体系结构构建了一个可传递、易于理解的系统级抽象

（三）体系结构设计

- 4. 软件体系结构风格
- 1）定义：体系结构风格定义了一系列系统的结构组织的模式，它是对一类具有相似结构的系统体系结构的抽象

（三）体系结构设计

4. 软件体系结构风格

1）定义

- 同一个问题，可以有不同的解决问题的模式，但我们根据经验，通常会采用特定的模式，这就是风格
- 软件风格是对软件构成带有整体性、普遍性、一般性的结构和结构关系的方法
- 因此，软件风格是一种特定的基本结构。

（三）体系结构设计

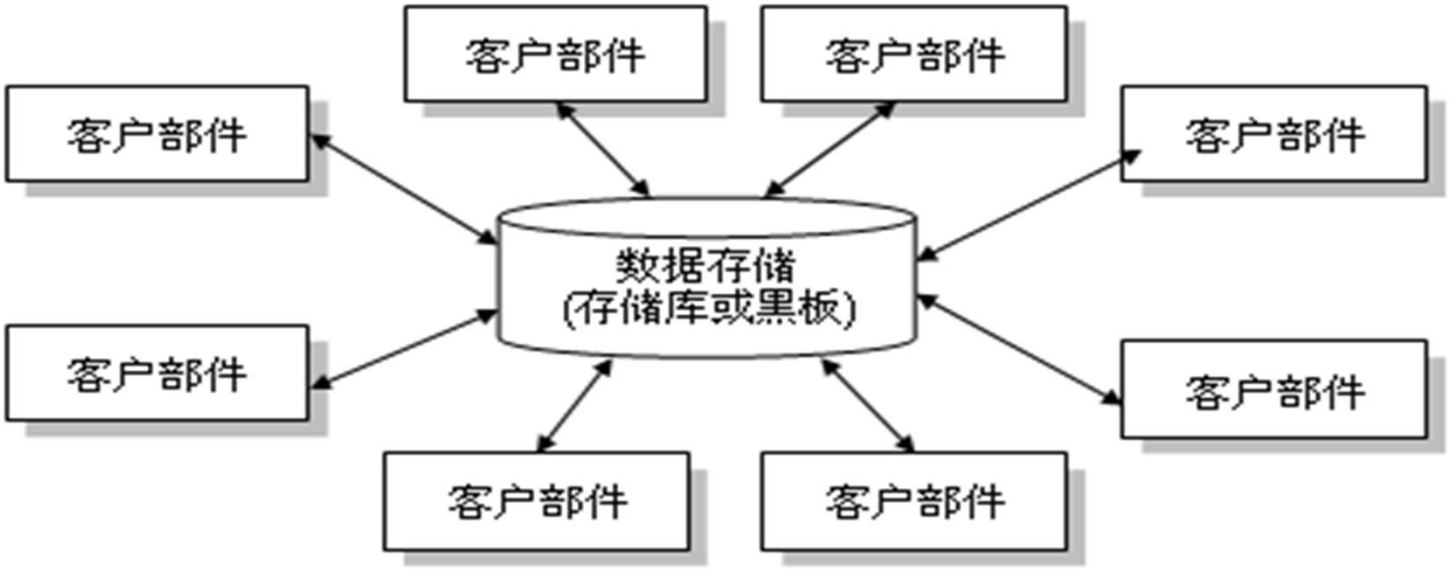
- 4. 软件体系结构风格
- 2）每种风格描述一种系统类别，其中包括：
  - 一些实现系统所需的功能的构件
  - 连接各个构件，负责构件间通信和协作的连接器
  - 定义构件之间怎样整合集成的系统约束
  - 使设计者能够理解整个系统属性并分析已知属性的语义模型

（三）体系结构设计

- **4.** 软件体系结构风格 Next
- **3）**体系结构风格的分类
  - 以数据为中心的体系结构 GO
  - 数据流体系结构 GO
  - 调用返回体系结构 GO
  - 面向对象体系结构 GO
  - 层次体系结构 GO

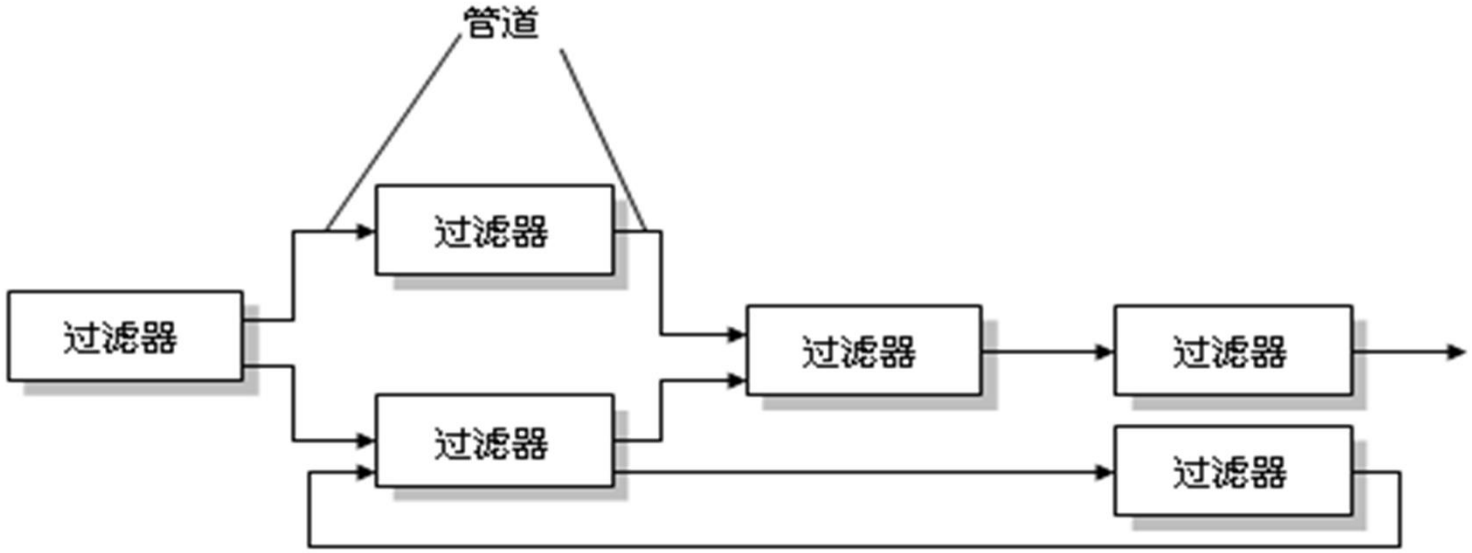
数据为中心体系结构 Back

- 一些数据(比如一个文件或者数据库)保存在整个结构的中心，并且被其他部件（构件）频繁地使用、添加、删除、或者修改



数据流风格的体系结构 Back

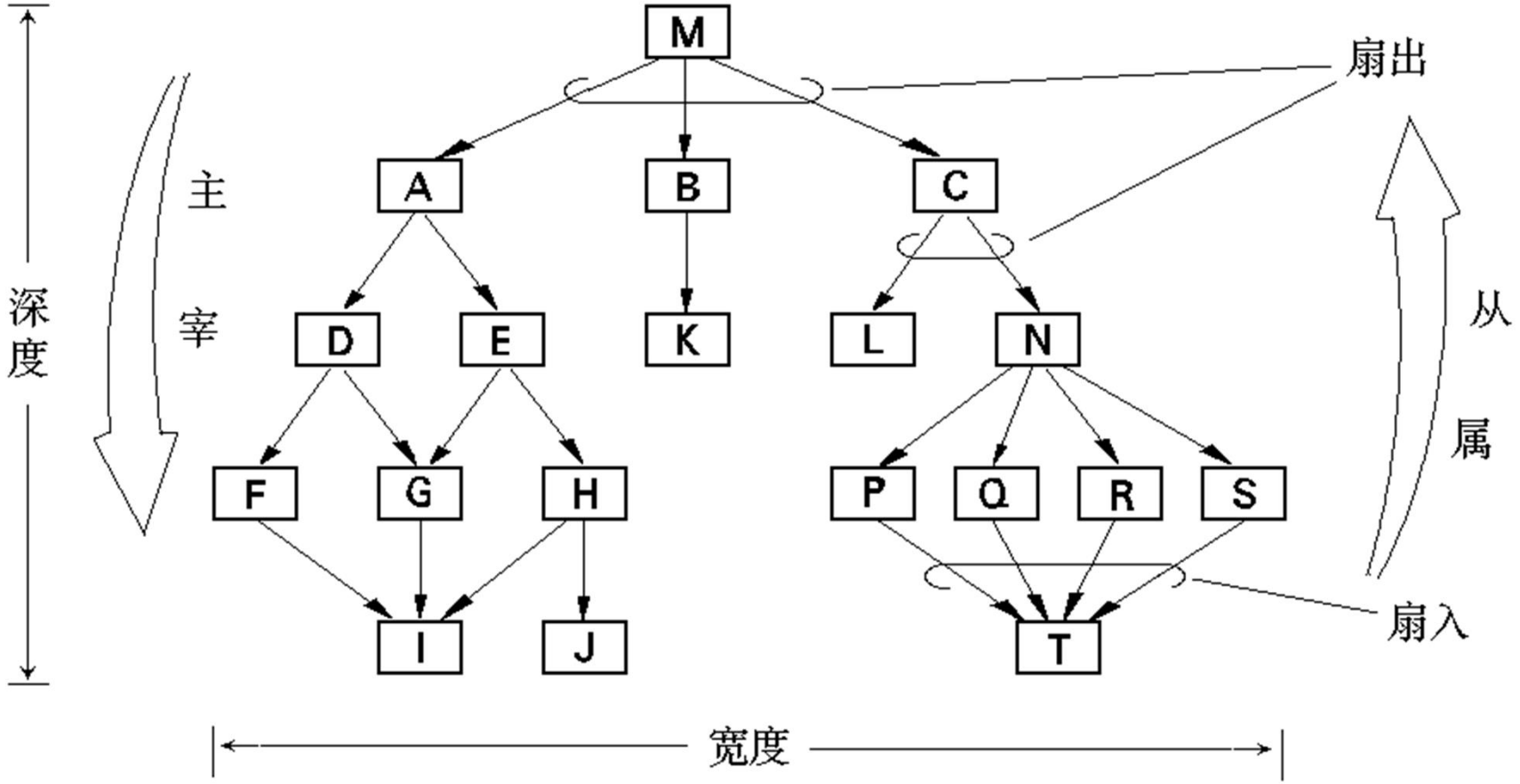
- 又称管道/过滤器体系结构风格
- 这种结构适用于输入数据被一系列的计算或者处理构件变换成输出数据。



调用和返回风格的体系结构 （1）

- 这种风格使一个软件设计者设计出非常容易修改和扩充的体系结构
  - 主程序/子程序风格体系结构
    - 传统程序结构，功能分解为控制层次
  - 远程过程调用风格的体系结构
    - 该体系结构的构件分布在网络的多个计算机上

调用和返回风格的体系结构（2）



调用和返回风格的体系结构（3） Back

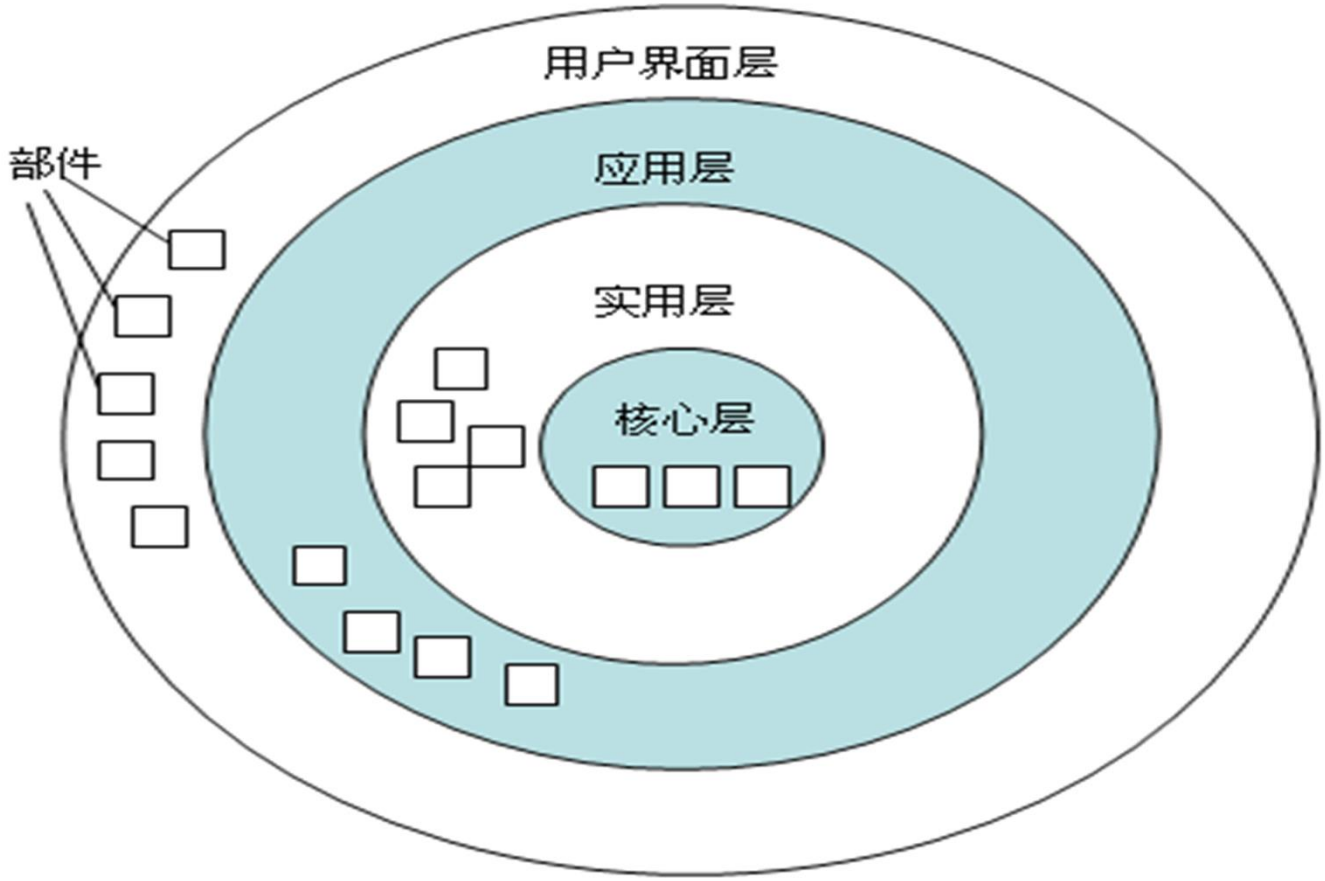
- 程序结构的深度：程序结构的层次数称为结构的深度。结构的深度在一定意义上反映了程序结构的规模和复杂程度。
- 程序结构的宽度：层次结构中同一层模块的最大模块个数称为结构的宽度。
- 模块的扇入和扇出：扇出表示一个模块直接调用（或控制）的其它模块数目。扇入则定义为调用（或控制）一个给定模块的模块个数。多扇出意味着需要控制和协调许多下属模块。而多扇入的模块通常是公用模块。

面向对象风格的体系结构 Back

- 系统部件封装数据和操作数据的方法
- 部件之间的交互和协调通过消息来传递

层次式风格的体系结构

- 在这种结构中，定义不同的层次，每层都完成了相对外层更靠近机器指令的操作



层次式风格的体系结构

- 例如：将层次式风格跟数据为中心的体系结构结合起来。
  - 用户界面层
    - 用于显示数据和接收用户输入的数据，提供交互式操作的界面
  - 应用层+实用层
    - 反映业务逻辑、业务流程的系统功能设计
  - 核心层
    - 数据访问层，负责数据库的访问

层次式风格的体系结构 Back

- 分层式结构也不可避免具有一些缺陷：
  - 降低了系统的性能。
    - 例如：如果不采用分层式结构，很多业务可以直接造访数据库，以此获取相应的数据，如今却必须通过中间层来完成。
  - 有时会导致级联的修改。
    - 例如：如果在界面层中需要增加一个功能，为保证其设计符合分层式结构，可能需要在相应的业务逻辑层和数据访问层中都增加相应的代码。

（三）体系结构设计

- 5. 软件体系结构设计方法
- 1）结构化设计
  - 结构化设计是将结构化分析的结果(数据流图)映射成软件的体系结构(结构图)，以系统化的途径实现软件结构的设计
  - 即：结构化的设计方法主要是解决如何将数据流图映射为软件体系结构。

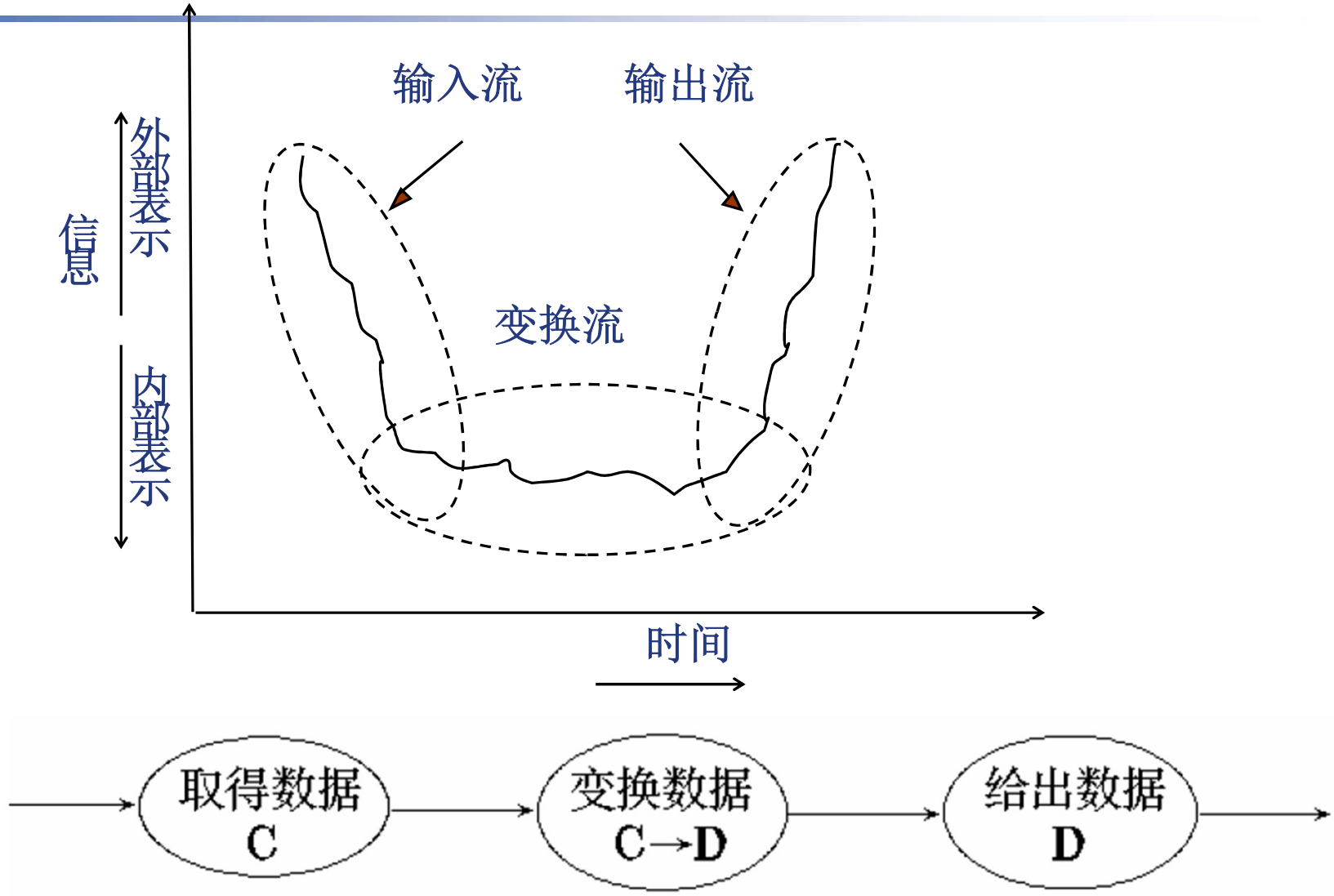
（三）体系结构设计

- 5. 软件体系结构设计方法
- 1）结构化设计 Next
- 数据流图→软件体系结构的主要任务
  - （1）复审和精化数据流图
  - （2）确定数据流图的类型(变换型 GO、事务型 GO)
  - （3）将DFD映射成初始结构图：
    - 采用变换分析(针对变换型数据流图) GO，
    - 或采用事务分析(针对事务型数据流图) GO
  - （4）改进初始结构图 GO

变换型数据流图

- 特征：数据流图可明显地分成输入、变换、输出三部分
- 信息沿着输入路径进入系统，并将输入信息的外部形式经过编辑、格式转换、合法性检查、预处理等辅助性加工后变成内部形式
- 内部形式的信息由变换中心进行处理
- 然后沿着输出路径经过格式转换、组成物理块、缓冲处理等辅助性加工后变成输出信息送到系统外

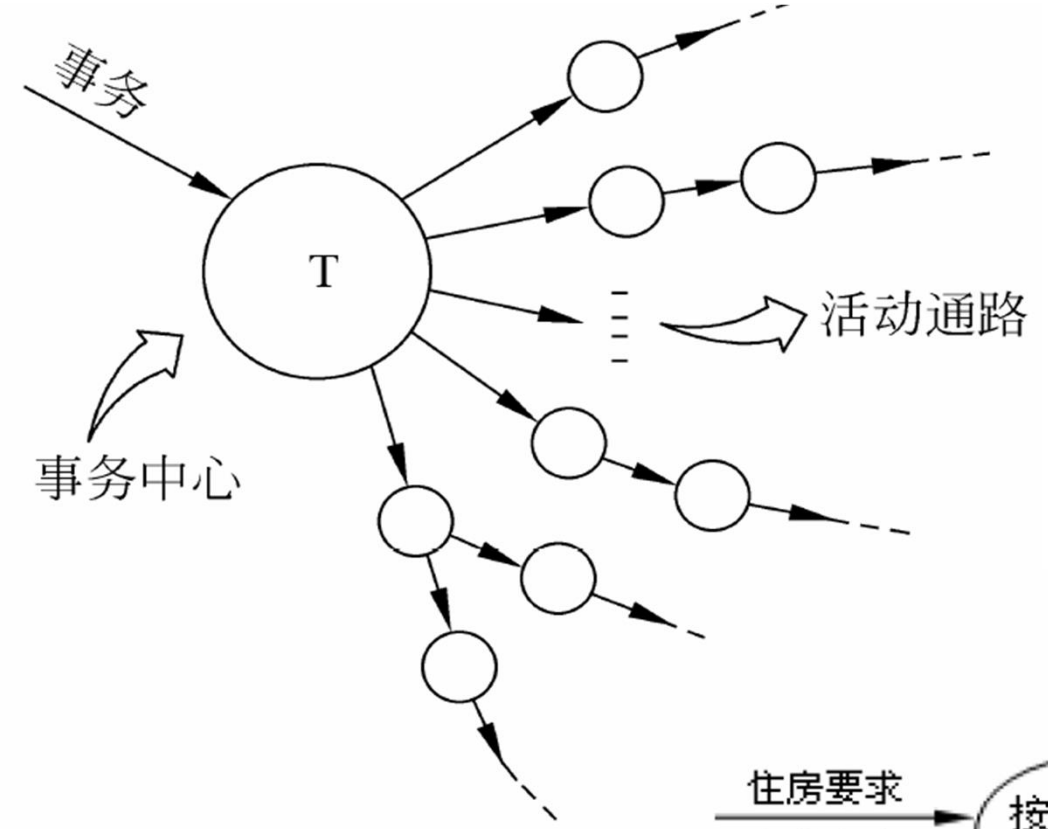
变换型数据流图 Back

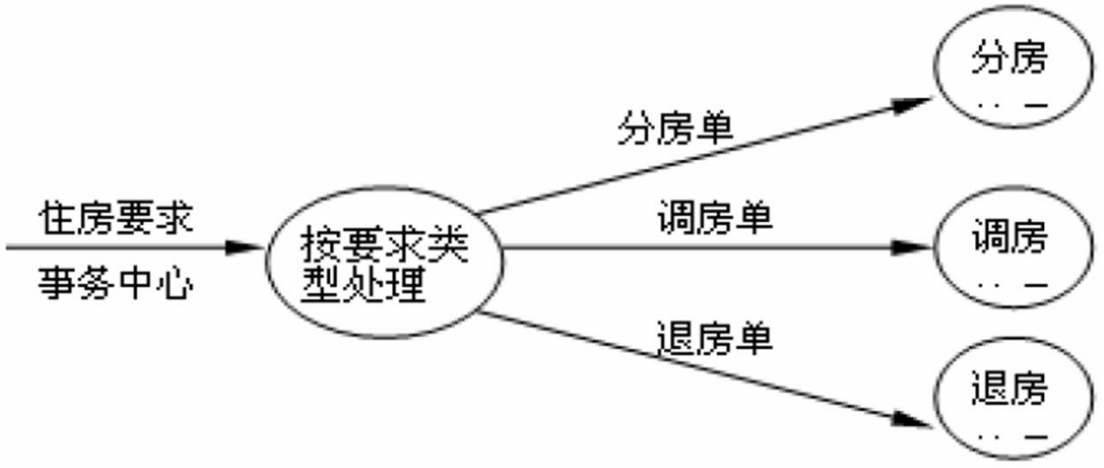


事务型数据流图

- 特征：
  - 数据流沿着输入路径到达一个事务中心，事务中心根据输入数据的类型在若干条活动通路中选择一条来执行
- 事务中心的任务是：决策
  - 接收输入数据(即事务)；
  - 分析每个事务的类型；
  - 根据事务类型选择执行一条活动通路

事务型数据流图 Back

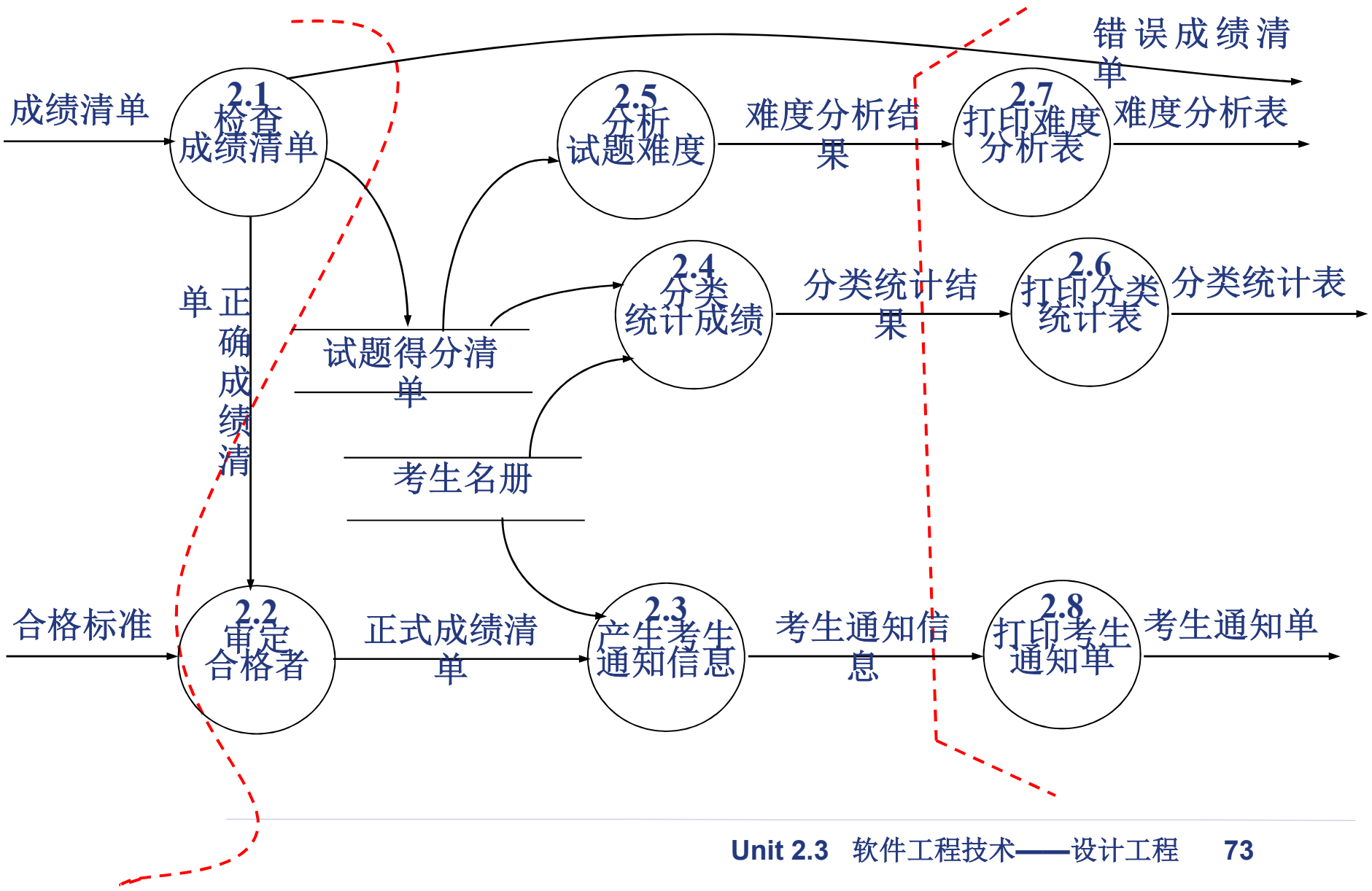




变换分析

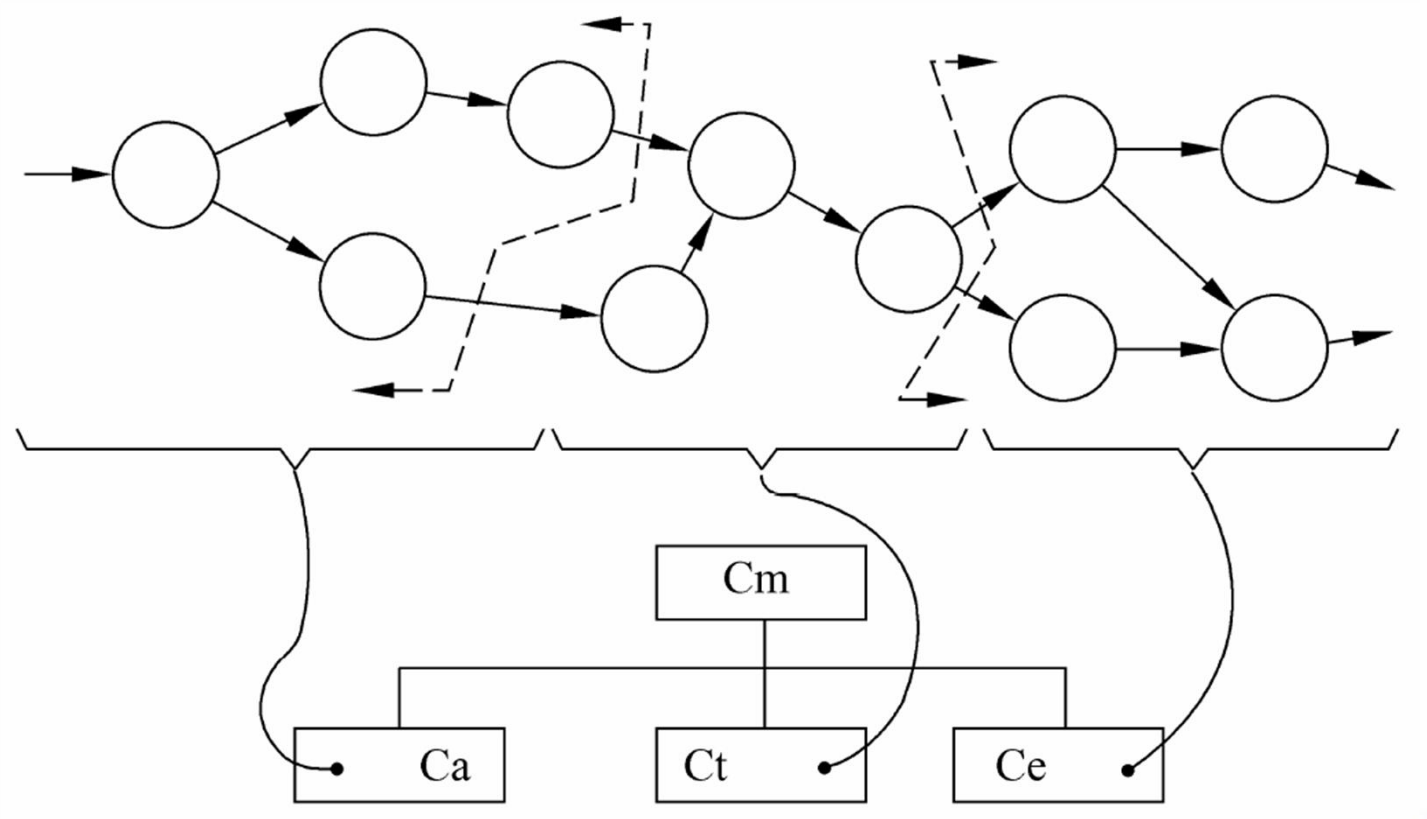
- 变换分析步骤如下：
  - 划定输入流和输出流的边界，确定变换中心
  - 进行第一级分解，设计上层模块
  - 进行第二级分解，设计中下层模块
  - 标注输入输出信息

示例：统计成绩子图的输入、输出流边界

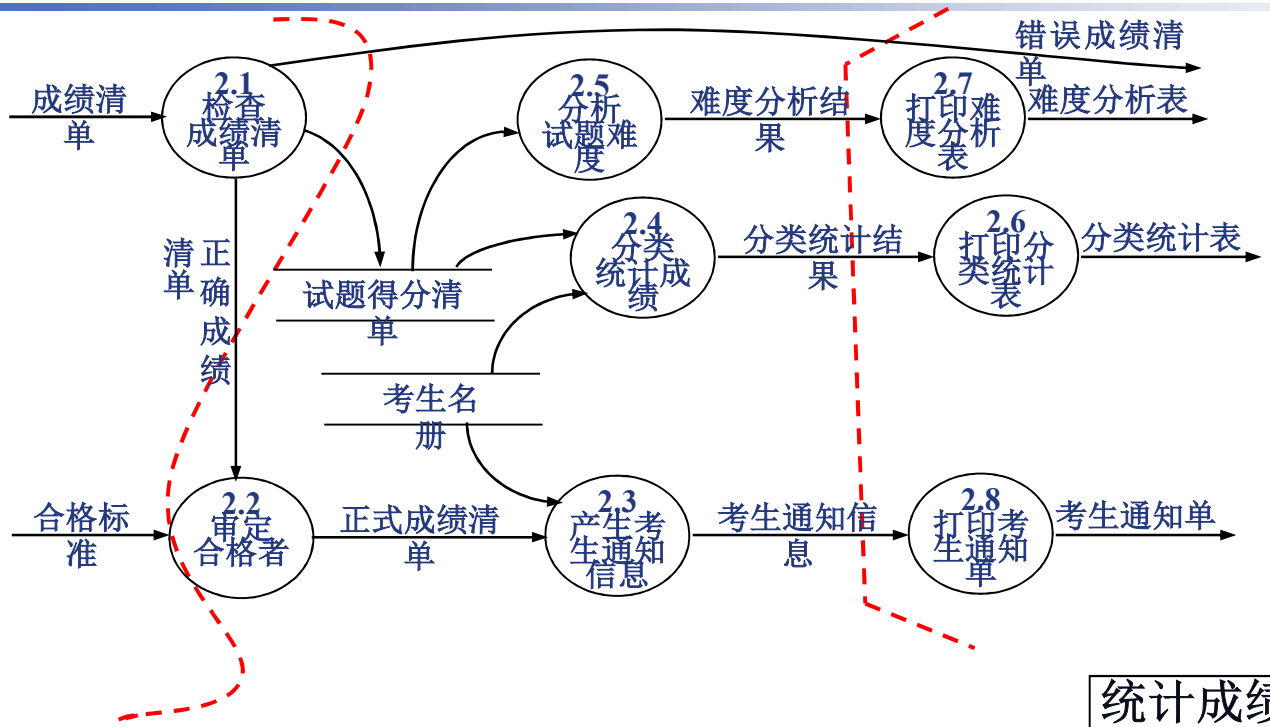


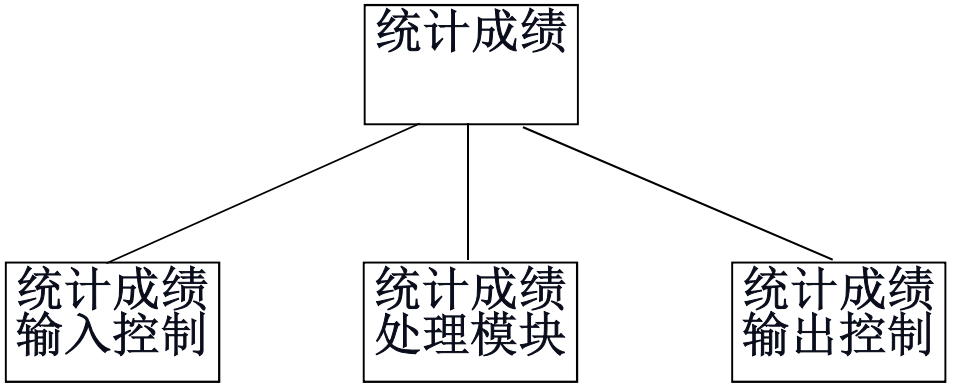
进行第一级分解

- 将**DFD**映射成变换型的程序结构：输入控制，变换控制，输出控制
- 大型的软件系统第一级分解时可多分解几个模块，以减少最终结构图的层次数



示例：统计成绩子图的第一级分解

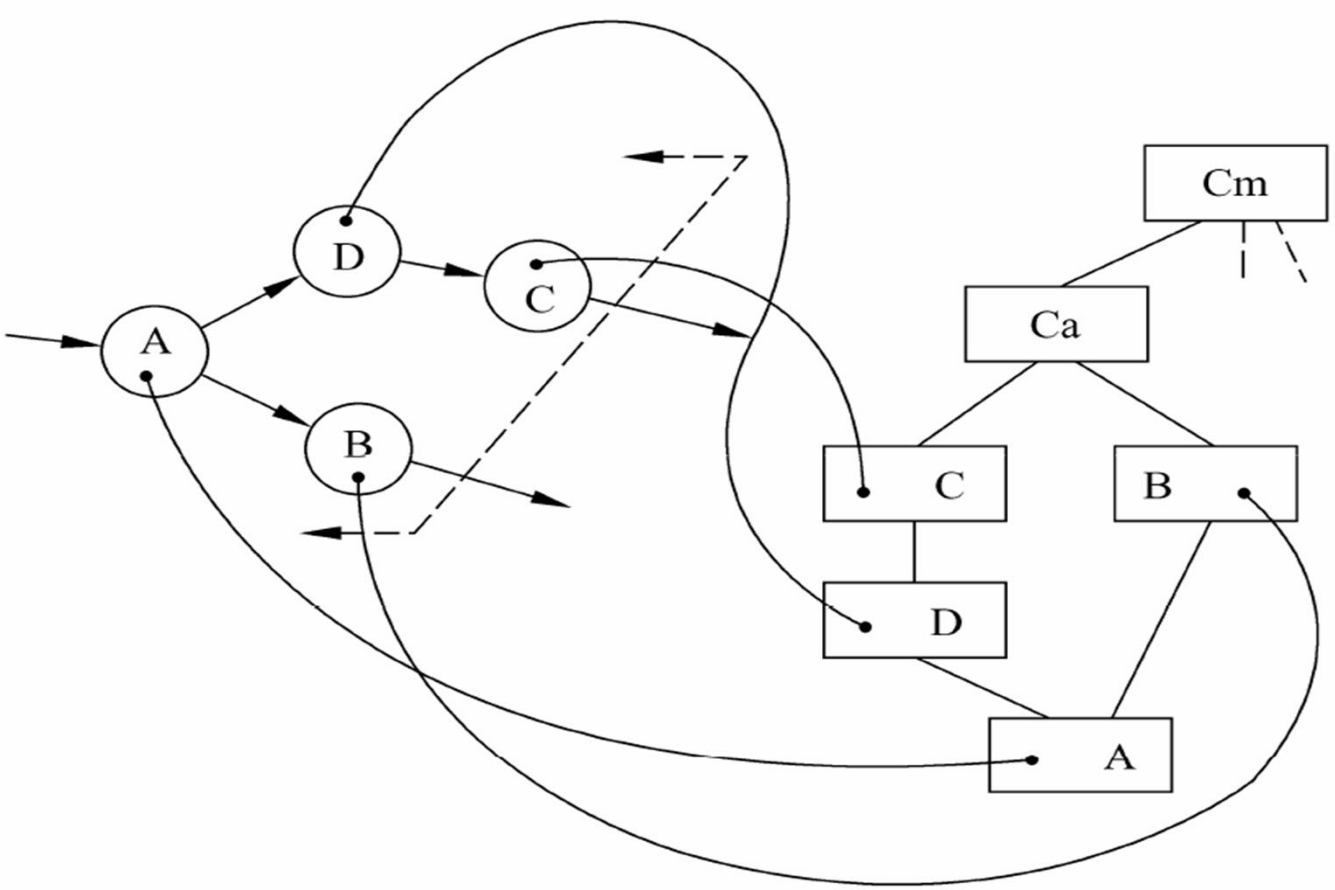




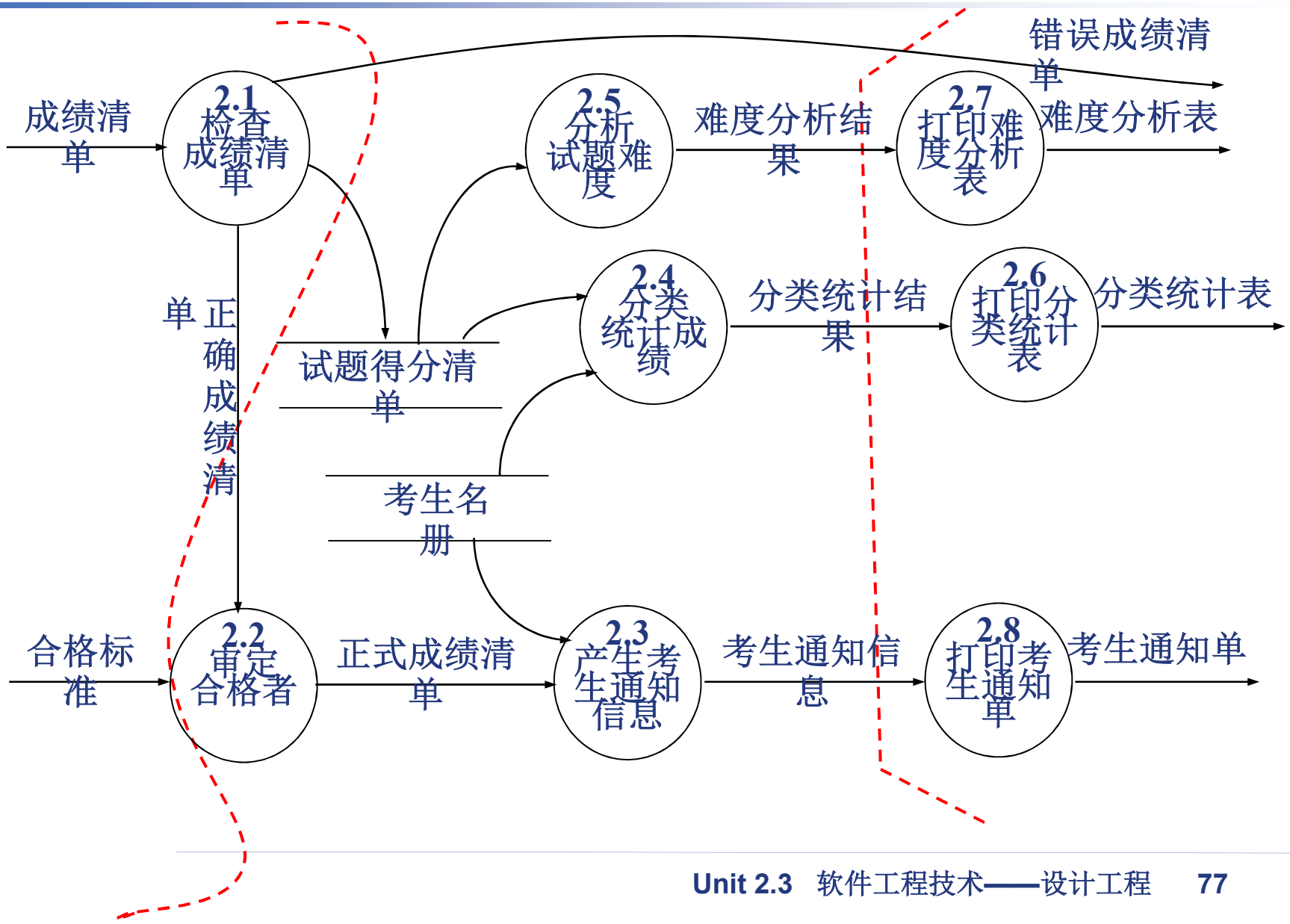
“统计成绩”第一级分解的结构图

进行第二级分解

- 将DFD中的加工映射成结构图中的一个适当的模块

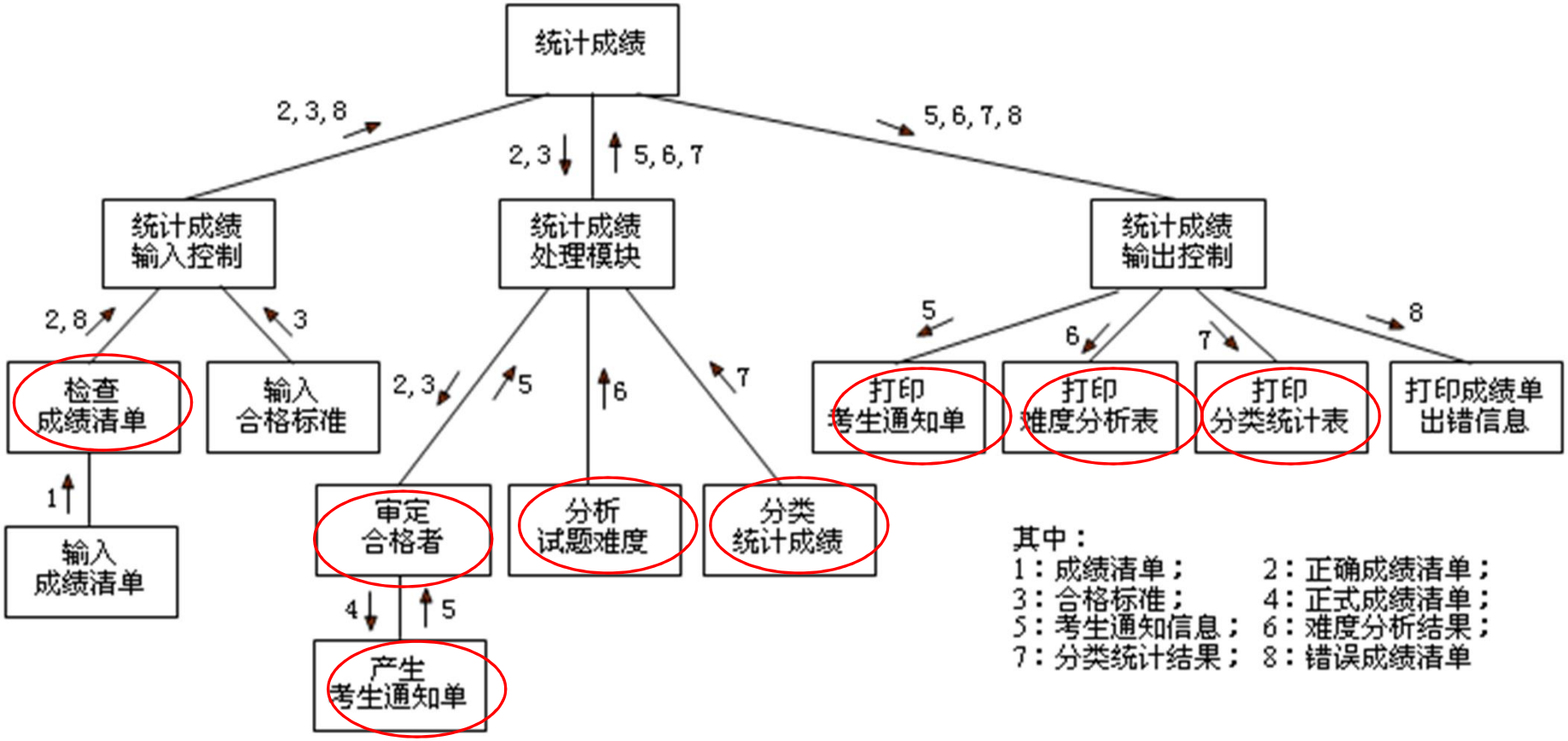


示例2：“统计成绩”数据流图



“统计成绩”第二级分解的结构图

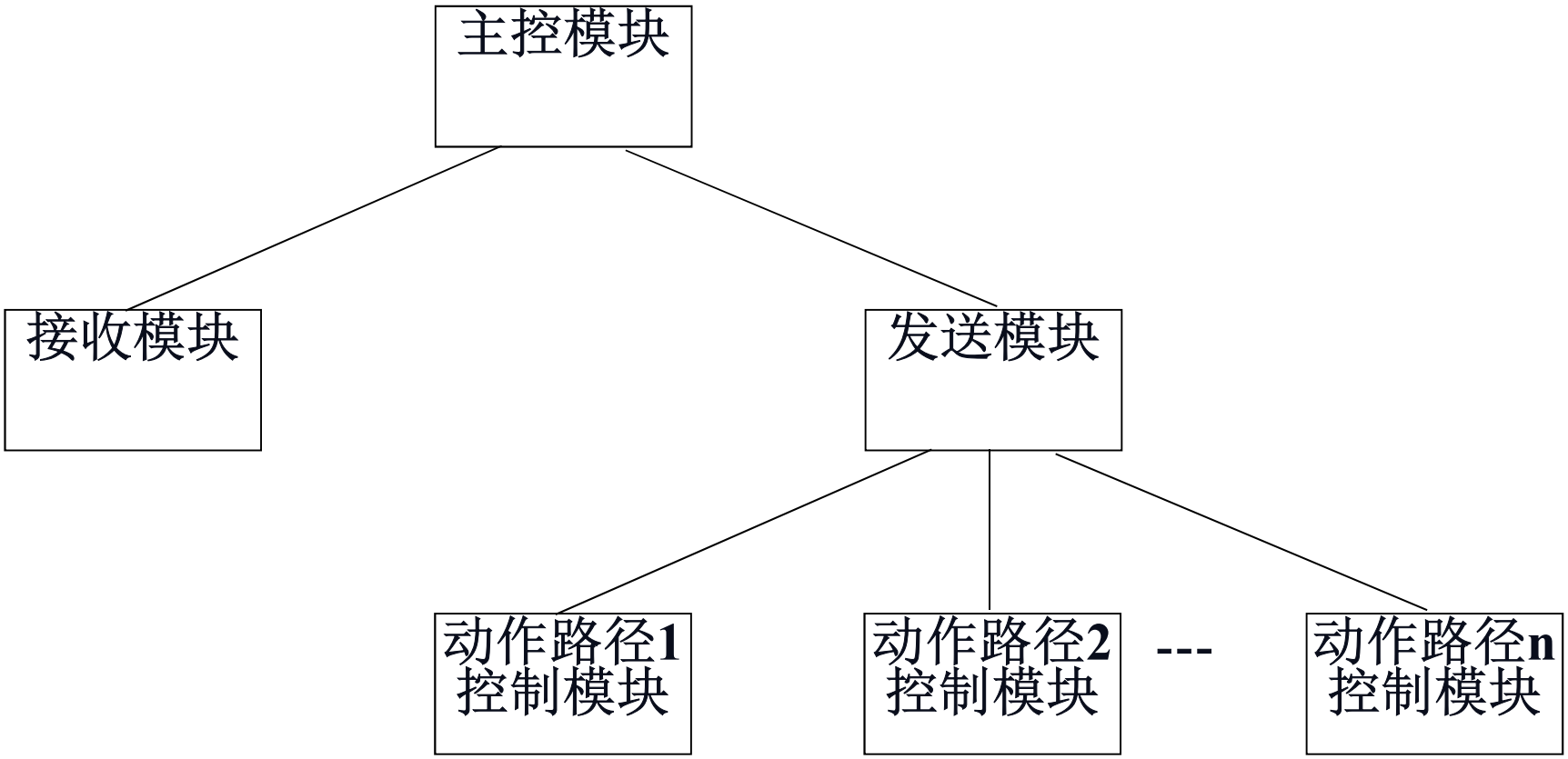
Back



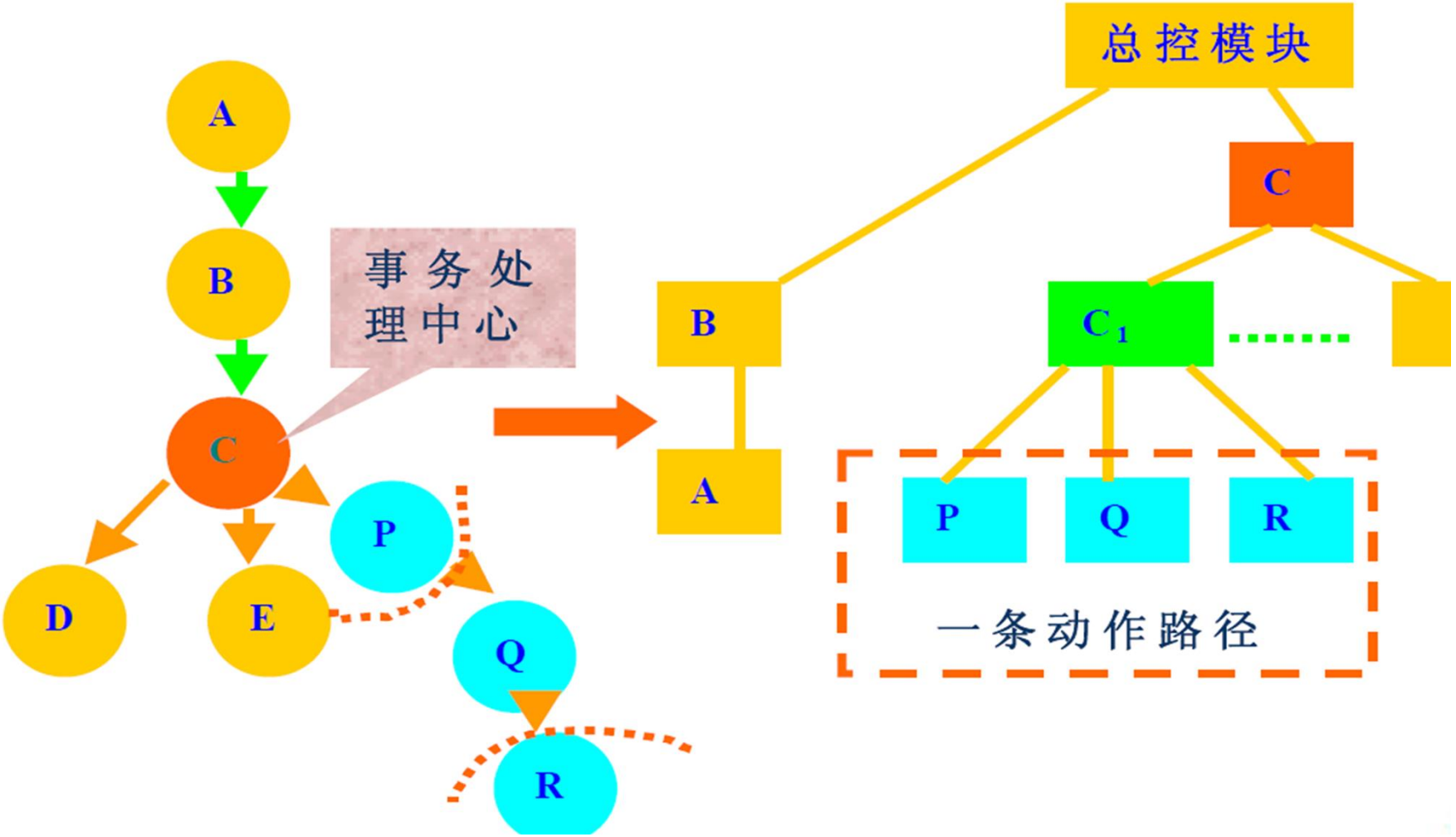
事务分析

- 将事务型DFD映射成初始的结构图
  - 处理事务的软件通常是接收一个事务，然后根据事务的类型执行一个事务处理的功能
  - 例如：银行业务中有存款、取款、查询余额、开户、转帐等多种事务
- 事务型软件体系结构通常包括：
  - 主控模块：完成整个系统的功能
  - 接收模块：接收输入数据(事务)
  - 发送模块：根据输入事务的类型，选择一个动作路径控制模块
  - 动作路径控制模块：完成相应的动作路径所执行的子功能

事务分析模块图



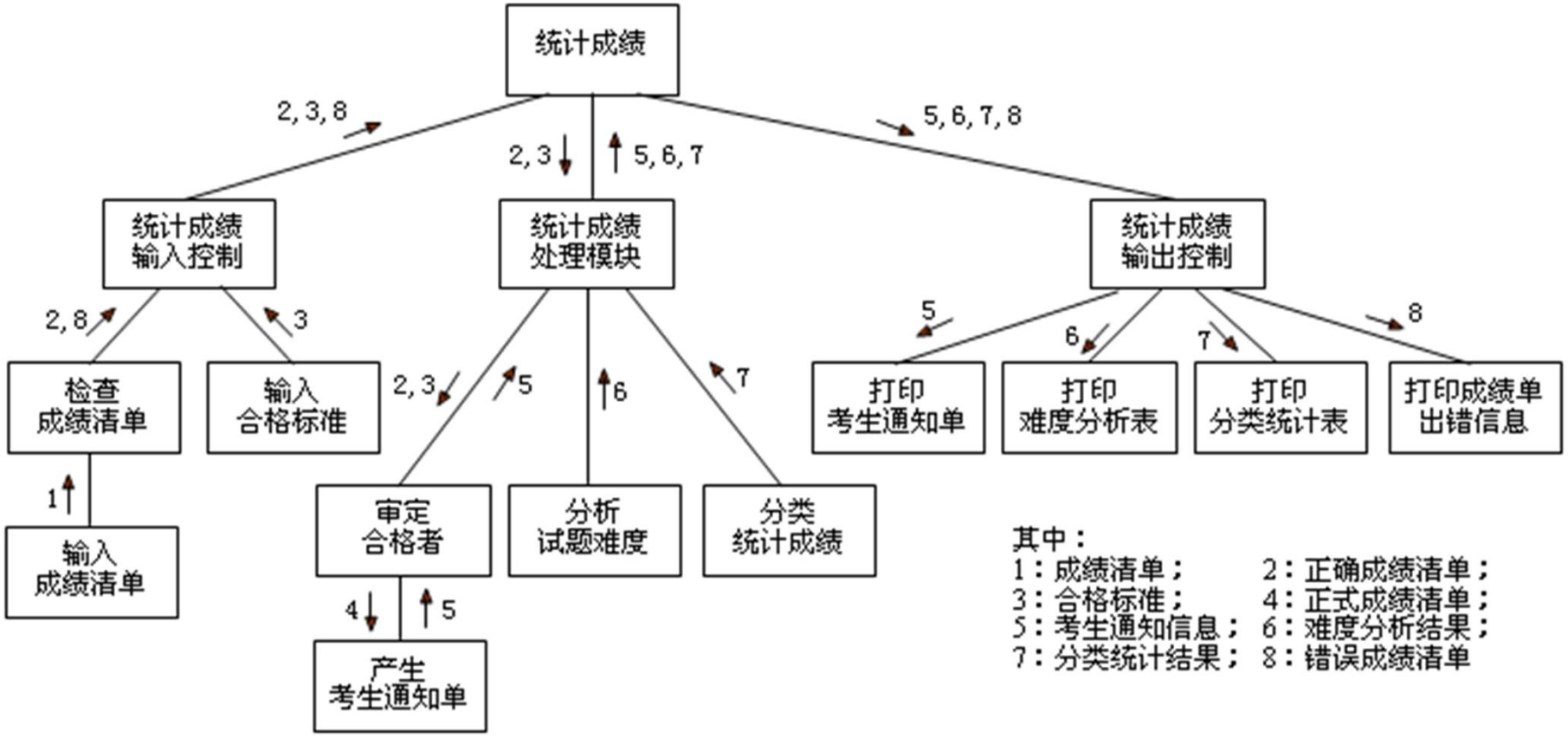
事务分析 Back



初始结构图的改进

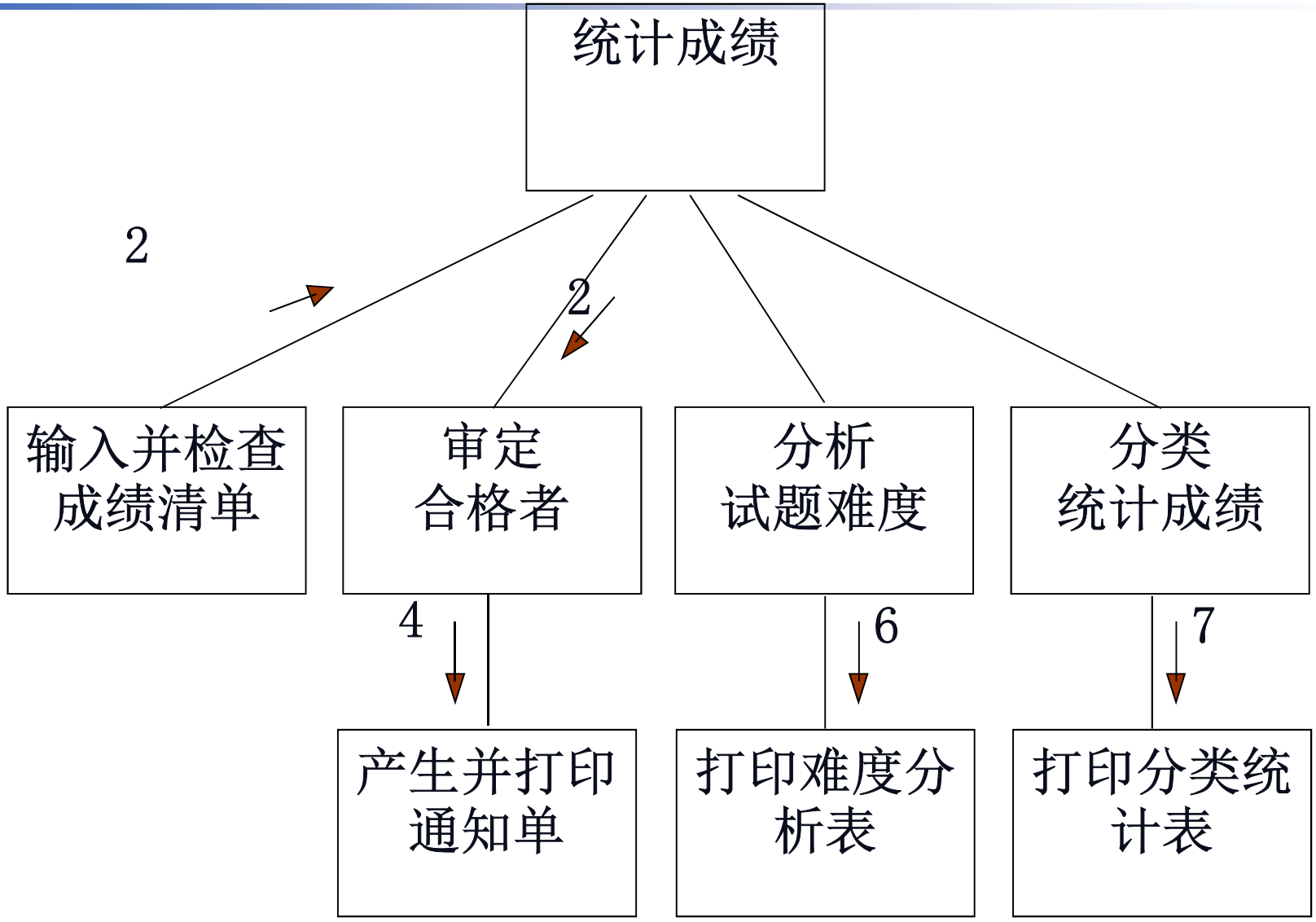
- 消除重复功能，改善软件结构
- 深度、宽度、扇出和扇入应适中
- 模块的影响范围应限制在该模块的控制范围内
- 减少模块间的耦合度
- 模块功能应该可以预测
- 模块大小适中

“统计成绩” 结构图

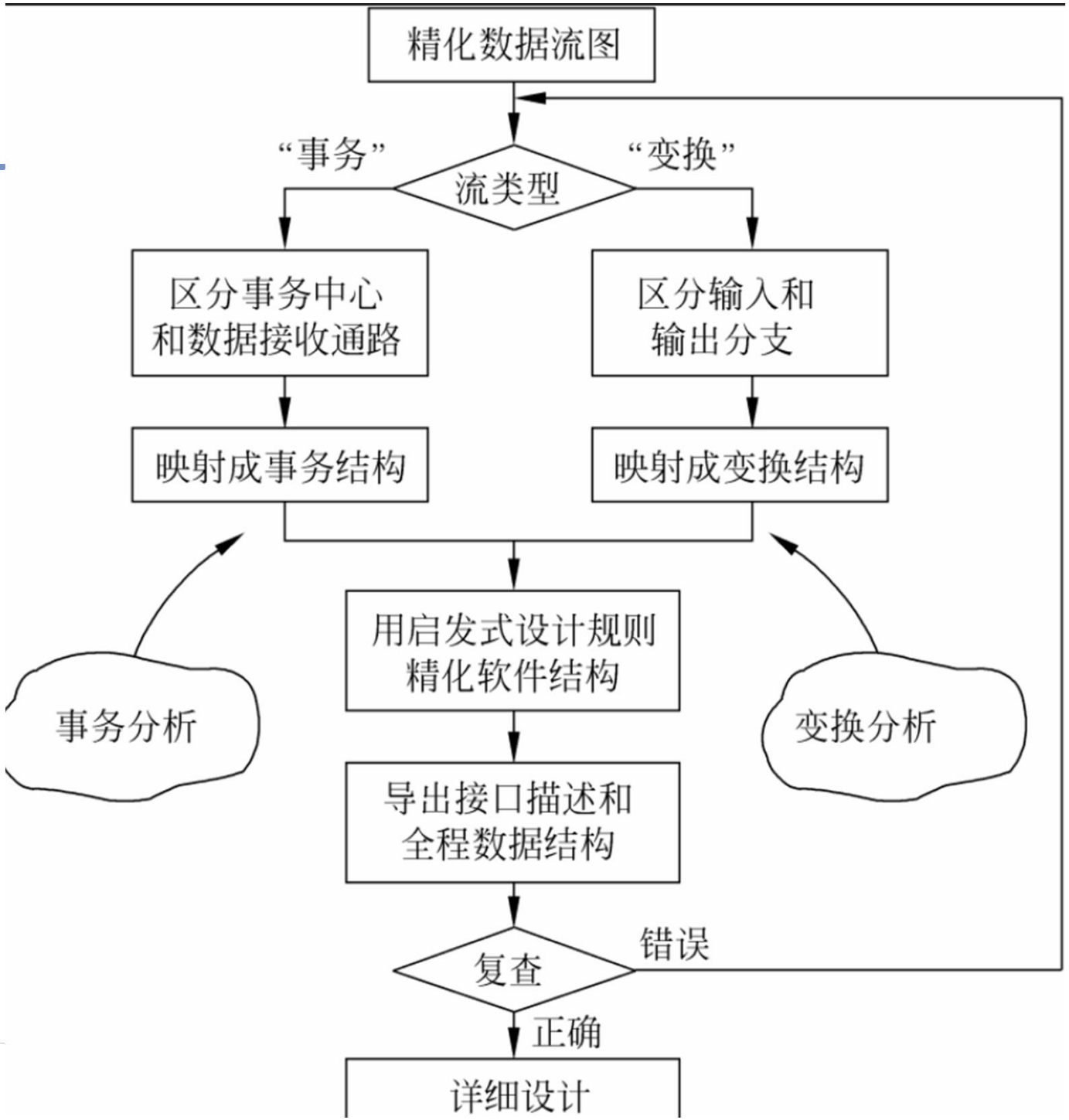

统计成绩
2, 3, 8
2, 3
5, 6, 7
5, 6, 7, 8
统计成绩
输入控制
统计成绩
处理模块
统计成绩
输出控制
2, 8
3
5
6
7
8
检查
成绩清单
输入
合格标准
2, 3
5
6
7
打印
考生通知单
打印
难度分析表
打印
分类统计表
打印成绩单
出错信息
1
输入
成绩清单
审定
合格者
分析
试题难度
分类
统计成绩
4
5
产生
考生通知单
其中：
1：成绩清单；
2：正确成绩清单；
3：合格标准；
4：正式成绩清单；
5：考生通知信息；
6：难度分析结果；
7：分类统计结果；
8：错误成绩清单

“统计成绩” 结构图改进

Back



设计步骤


精化数据流图
“事务”
流类型
“变换”
区分事务中心
和数据接收通路
区分输入和
输出分支
映射成事务结构
映射成变换结构
事务分析
用启发式设计规则
精化软件结构
变换分析
导出接口描述和
全程数据结构
复查
错误
正确
详细设计

（三）体系结构设计——5. 设计方法

- 2）面向对象设计
- （1）子系统设计
  - OOD方法也采取问题分解的处理方法，这样，系统设计就变成了子系统的设计和集成问题
  - 从分析模型开始，对数据（对应于对象模型）、行为（对应于行为模型）和功能（对应于功能模型）三个方面结合，进行问题分解
  - 由此，定义出若干个一致的类与对象、关系、行为、功能的集合。每一个这样的集合，就是一个子系统

（三）体系结构设计——5. 设计方法

- 2）面向对象设计
- （2）对象设计
  - 设计阶段的对象设计，就是指对每个对象中的数据部分的设计
  - 对象设计的数据设计，是对对象的进一步求精，得到每一个对象的更为准确的属性，然后设计出这些属性的相应的数据结构
  - 有关对象数据结构的设计，基本上已经属于构件级设计阶段

（三）体系结构设计——5. 设计方法

- 2）面向对象设计
- （3）消息设计
  - 消息设计就是描述每一个对象可以接收和发送消息的接口
  - 消息设计的来源是对象之间的关系
    - 在系统、子系统间的消息，是通过“通信类”对象实现的
    - 在子系统内部，根据事件跟踪图，获得对象间的消息信息
  - 具体的消息设计在构件级设计阶段完成

（三）体系结构设计——5. 设计方法

- 2）面向对象设计
- （4）方法设计
  - 方法设计是指在根据面向对象的行为模型和功能模型，进一步对对象的方法，进行求精
    - ✓找出分析中遗漏的地方
    - ✓定义方法过程化的细节
  - 方法设计，主要在构件级设计阶段完成

（三）体系结构设计——总结

- 1.软件体系结构概念
- 2.为什么要使用软件体系结构
- 3.软件体系结构设计的重要性
- 4.体系结构的风格
- 5.设计方法
  - 面向结构：数据流图→体系结构
  - 面向对象：子系统、对象、消息、方法

（四）构件级设计

- 1. 什么是构件？
- **“A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”**

— OMG UML Specification

（四）构件级设计

1. 什么是构件？

- **OO View – A component is a set of collaborating classes.**
- **Conventional View – A component is a functional element of a program that incorporates processing logic, the internal data structures required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.**

（四）构件级设计

- **2.** 什么是构件级设计？
- 体系结构设计确定了系统的总体结构
- 构件级设计
  - 是对体系结构设计结果的进一步细化
  - 是对目标的精确描述
  - 面向编码实现，可以直接翻译成计算机代码的

（四）构件级设计

- 2. 什么是构件级设计？
- 构件级设计
  - **Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each component**

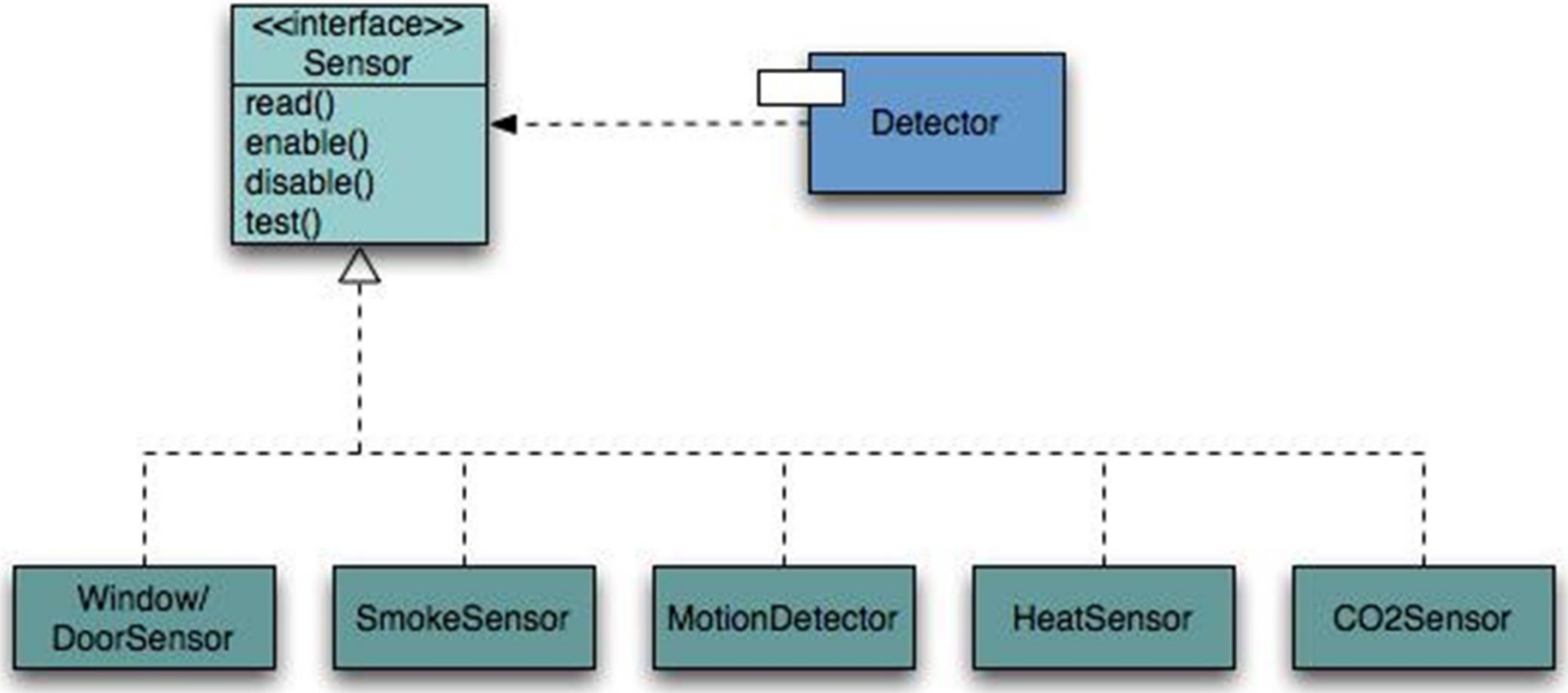
（四）构件级设计

3. 构件级设计的设计原则 Next

- 开关原则(Open-Closed Principle) GO
- 替换原则(Substitutability) GO
- 依赖倒置原则(Dependency Inversion) GO
- 接口分离原则(Interface Segregation) GO
- 高内聚，低耦合

Open-Closed Principle开关原则 Back

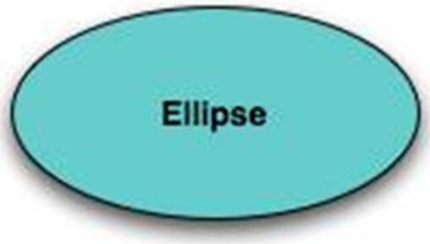
- A module should be open for extension but closed for modification.



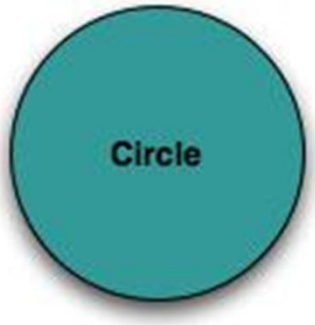
Substitutability 替换原则 Back

- **Subclasses should be substitutable for base classes**

Is a circle a kind of ellipse?



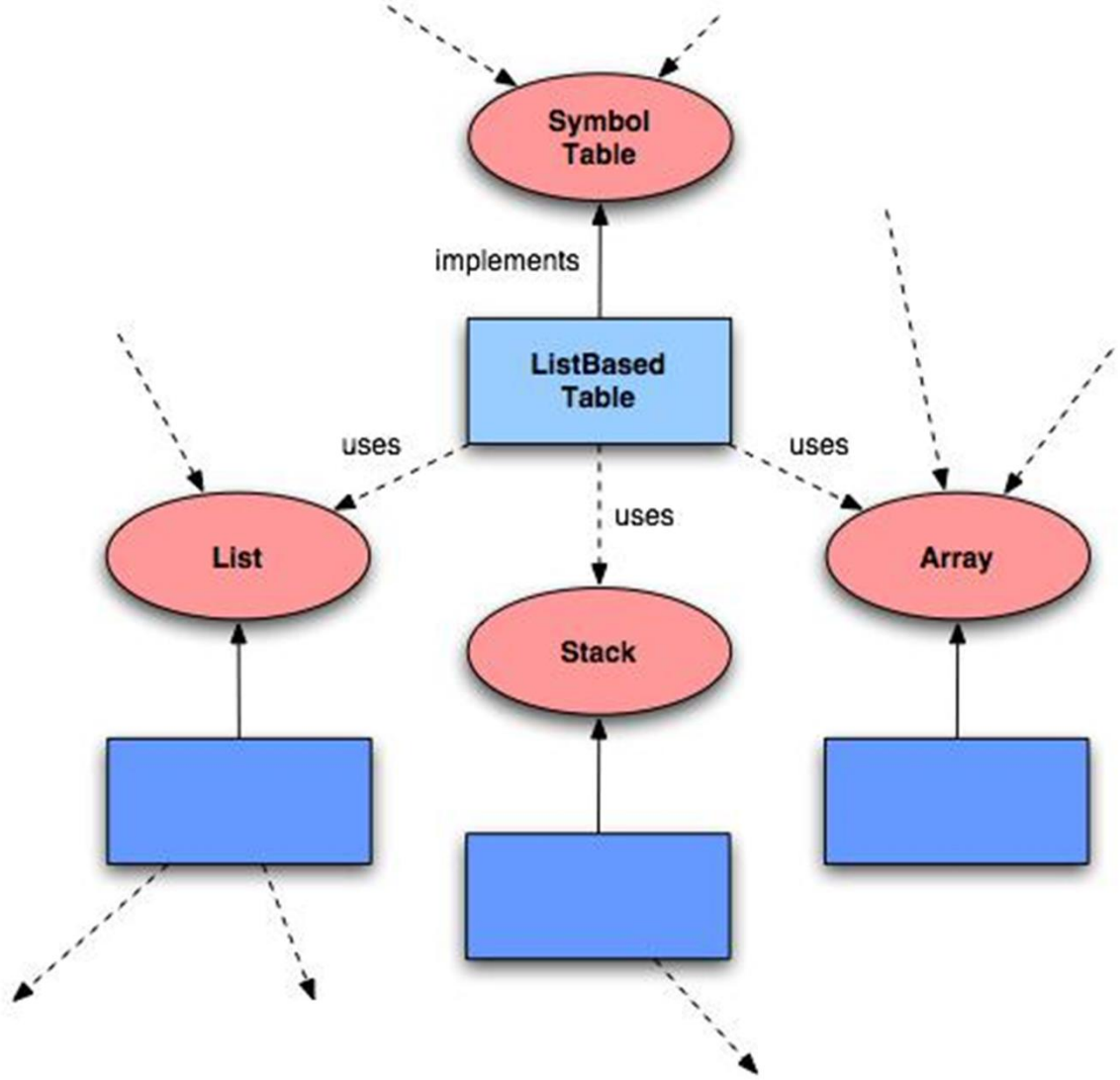
public void setSize(int x, int y);
requires nothing
ensures after the call, the ellipse is
x units wide and y units high



public void setSize(int x, int y);
requires x = y
ensures after the call, the ellipse is
x units wide and y units high

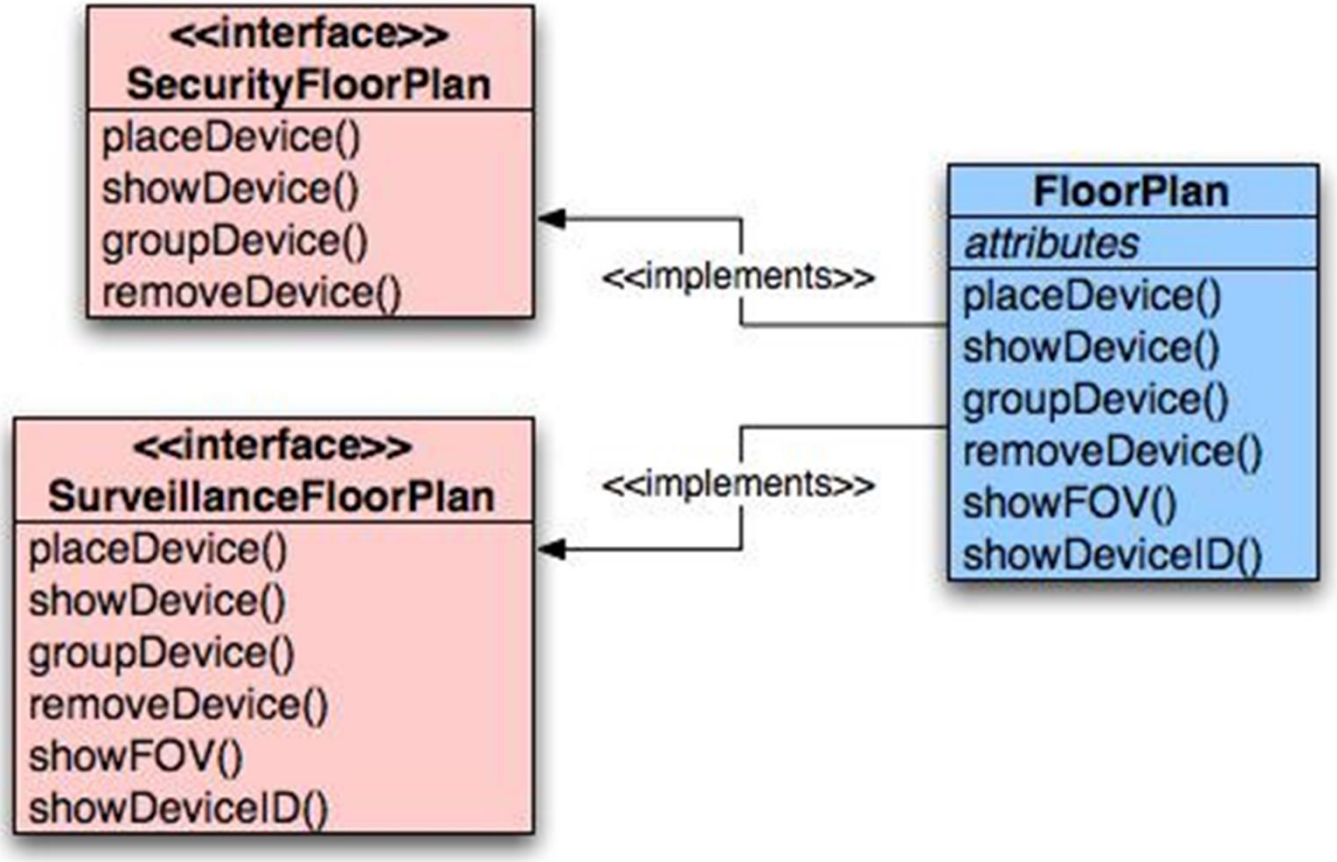
Dependency Inversion依赖倒置 Back

- Depend on abstractions. Do not depend on concretions.



Interface Segregation接口分离 Back

- **Many client-specific interfaces are better than one general purpose interface.**



（四）构件级设计

4. 构件级设计的设计任务

1）传统的设计

- （1）分析模型和体系结构构件。为每个构件确定采用的算法，选择某种适当的工具表达算法的过程，编写构件的详细过程性描述；
- （2）确定每一构件内部使用的数据结构；
- （3）在构件级设计结束时，应该把上述结果写入构件级设计说明书，并且通过复审形成正式文档，作为下一阶段（编码阶段）的工作依据

（四）构件级设计

- 4. 构件级设计的设计任务
  - 2）面向对象的设计
    - （1）标识类：标识出所有与问题域相对应的设计类
    - （2）标识基础类：确定所有与基础设施域相对应的设计类。这些类在前期分析和设计中通常忽略，如：GUI构件、操作系统构件等
    - （3）细化类：细化类的接口、属性和操作

（四）构件级设计

4. 构件级设计的设计任务

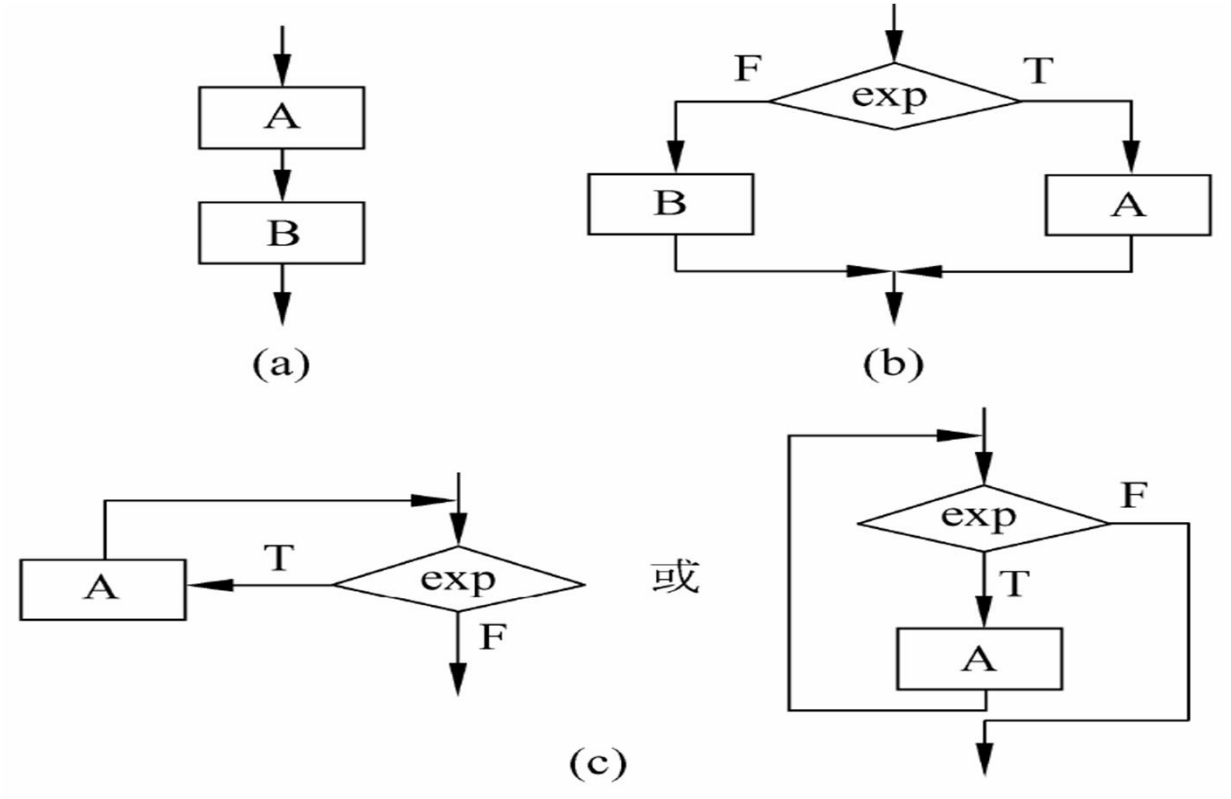
- 2）面向对象的设计
  - （4）说明持久数据源：确定数据库和文件，并确定管理数据源所需要的类
  - （5）开发并细化类或构件的行为表示：使用状态图对某些类的行为进行建模
  - （6）细化部署图：表示主要构件包的位置，软硬件部署等
  - （7）迭代重构：设计是一个迭代过程，设计过程中要不断重构构件，并选择其他的设计方案

（四）构件级设计

- **5. 算法设计方法** Next
- **Represents the algorithm at a level of detail that can be reviewed for quality**
  - **graphical （图形化方法）**
    - **Flowchart （程序流程图）** GO
    - **box diagram (N-S)（盒图，N-S图）** GO
  - **decision table （决策表法）** GO
  - **pseudocode (e.g., PDL)（伪码）** GO

程序流程图 （1）

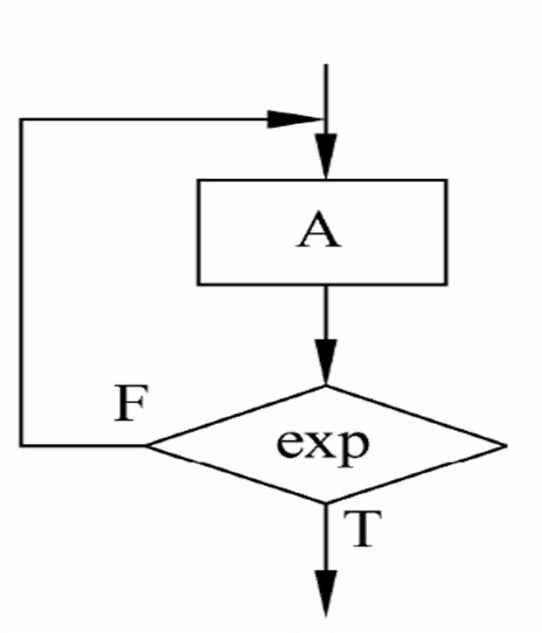
Sequence



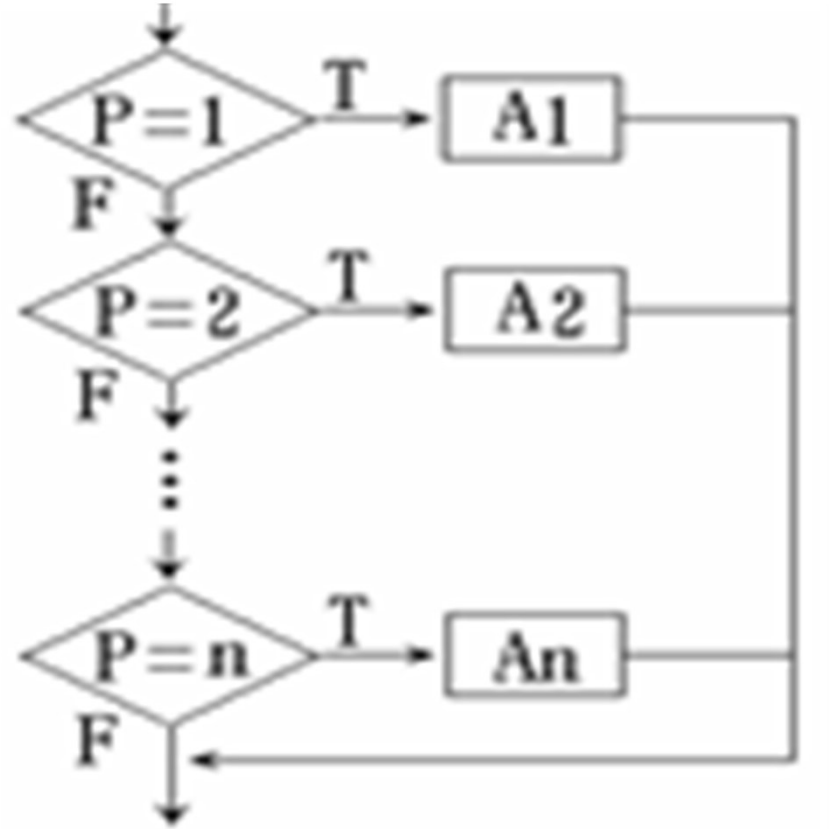
If …else…

While loop

程序流程图（2）

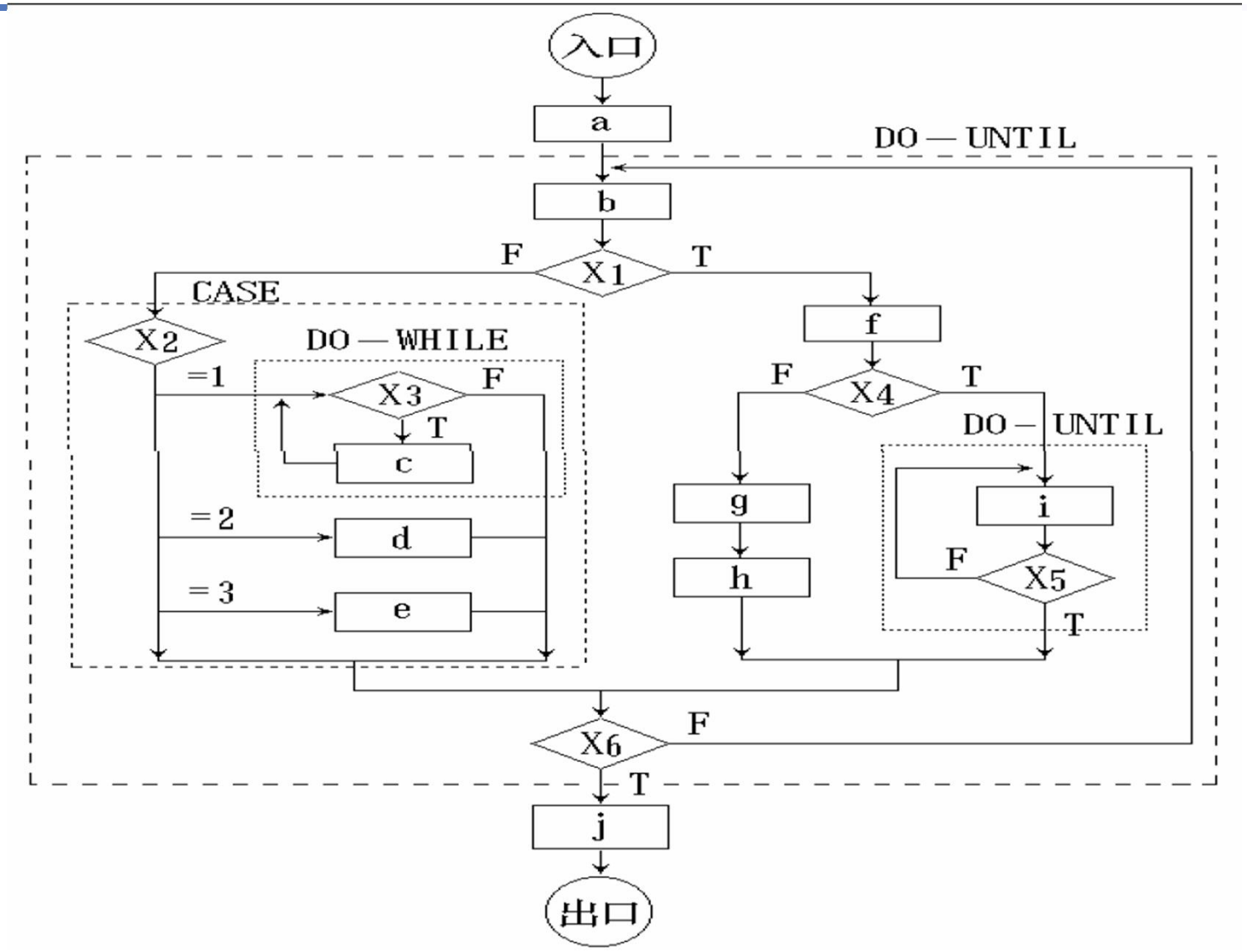


Do… Until



Switch

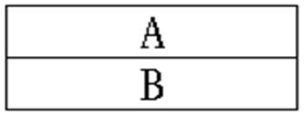
控制结构相互组合和嵌套的实例 Back



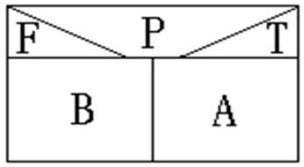
N-S图

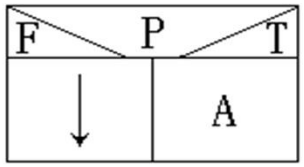
- Nassi和Shneiderman 提出了一种符合结构化程序设计原则的图形描述工具，叫做盒图，也叫做N-S图
- 盒图没有箭头，因此不允许随意转移控制。坚持以盒图为设计工具，可以使程序员逐步养成用结构化方式思考问题和解决问题的习惯。

N-S图的五种基本控制结构

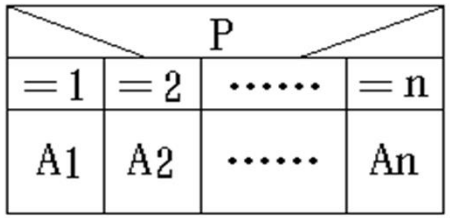


① 顺序型

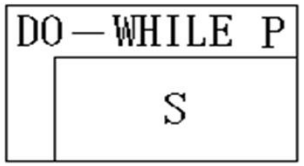




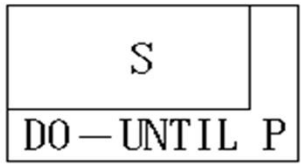
② 选择型



⑤ 多分支选择型
（CASE型）

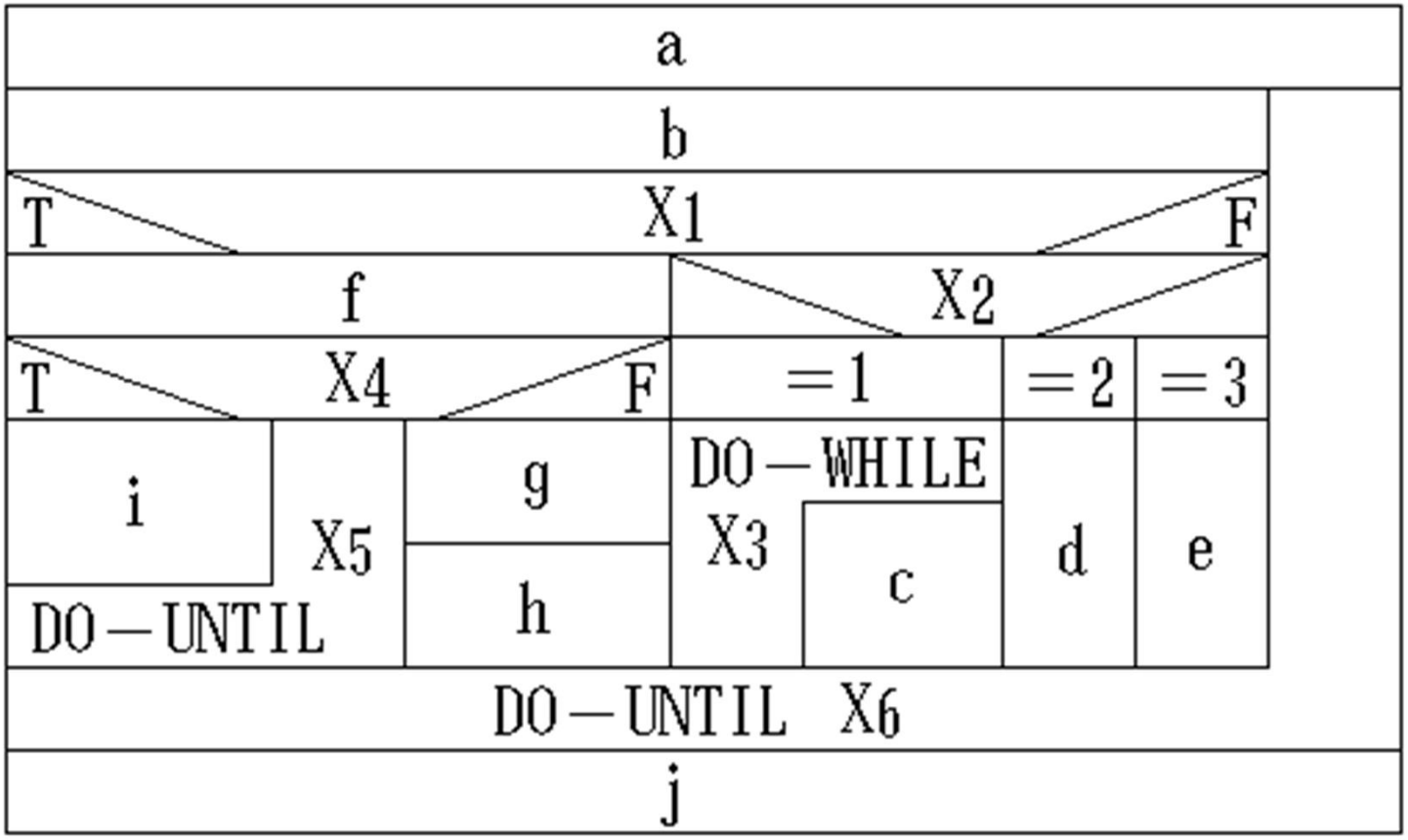


③ WHILE重复型



④ UNTIL重复型

控制结构相互组合和嵌套的实例 Back



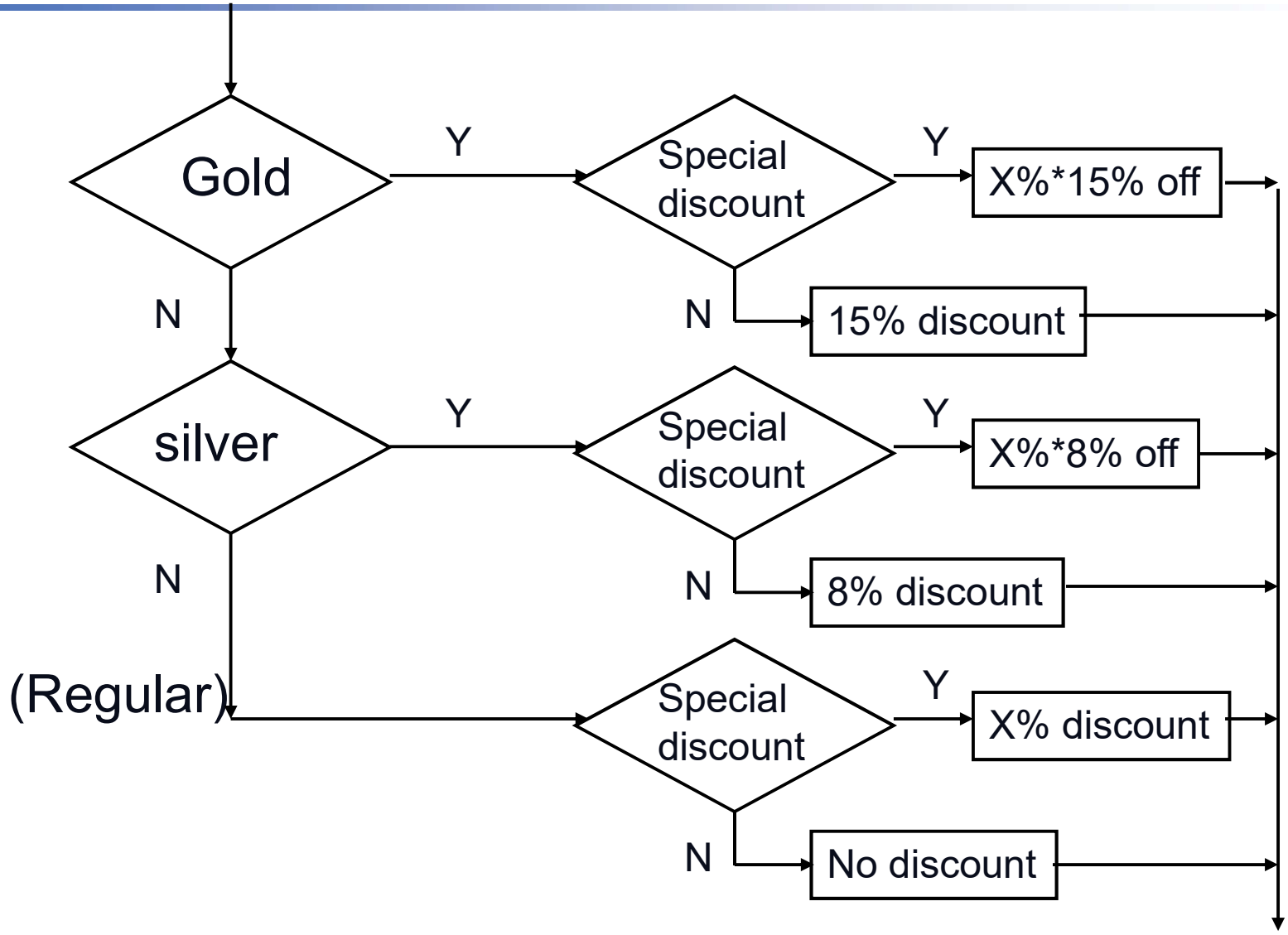
决策表/判定表（Decision Table）

- 一张决策表由四部分组成：
- （1）左上部列出所有条件；
- （2）左下部是所有可能做的动作；
- （3）右上部为各种可能组合条件，其中每一列表示一种可能组合；
- （4）右下部的每一列是和每一种条件组合所对应的应做的工作。

决策表/判定表（Decision Table）

- Example
  - Regular customer
    - No discount
  - Silver customer
    - 8% off
  - Gold customer
    - 15% off
  - Special discount
    - X% off

其程序流程图为


Gold
Y
Special discount
Y
X%*15% off
N
N
15% discount
silver
Y
Special discount
Y
X%*8% off
N
N
8% discount
(Regular)
Special discount
Y
X% discount
N
No discount

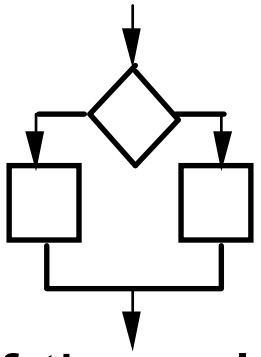
其决策表为 Back

	Rules					
Conditions	**1**	**2**	**3**	**4**	**5**	**6**
regular customer	T	T				
silver customer			T	T		
gold customer					T	T
special discount	F	T	F	T	F	T
Rules						
no discount	✓					
apply 8 percent discount			✓	✓		
apply 15 percent discount					✓	✓
apply additional x percent discount		✓		✓		✓

Program Design Language (PDL)

- **PDL**是一种用于描述功能部件的算法设计和处理细节的语言，称为设计性语言。它是一种伪码。
- 一般地，伪码的语法规则分为“外语法”和“内语法”
  - 外语法应当符合一般程序设计语言常用语句的语法规则；
  - 内语法可以用英语中一些简单的句子、短语和通用的数学符号，来描述程序应执行的功能

Program Design Language (PDL)



if-then-else

```
if condition x
  then process a;
  else process b;
endif
```

PDL

- easy to combine with source code
- machine readable, no need for graphics input
- graphics can be generated from PDL
- enables declaration of data as well as procedu
- easier to maintain

PDL Example Back

- PROCEDURE **spellcheck**
  BEGIN
  split document into single words
  look up words in dictionary
  display words which are not in dictionary

  END **spellcheck**

（四）构件级设计——总结

- 1.构件概念
- 2.构件级设计概念
- 3.构件级设计原则
- 4.构件级设计的任务
- 5. 算法设计方法

（五）建模和设计工具

- IBM Rational Software Modeler /Rose
- Telelogic TAU
- Borland Together
- Oracle Designer
- CA BPWin/ERWin
- Sybase PowerDesigner
- Miscrosoft Visio Professional

本章总结

（一）系统设计的任务和目标

（二）设计相关概念

（三）体系结构设计

（四）构件级设计

（五）设计建模工具

Southeast University

Thank You !

lliao@seu.edu.cn