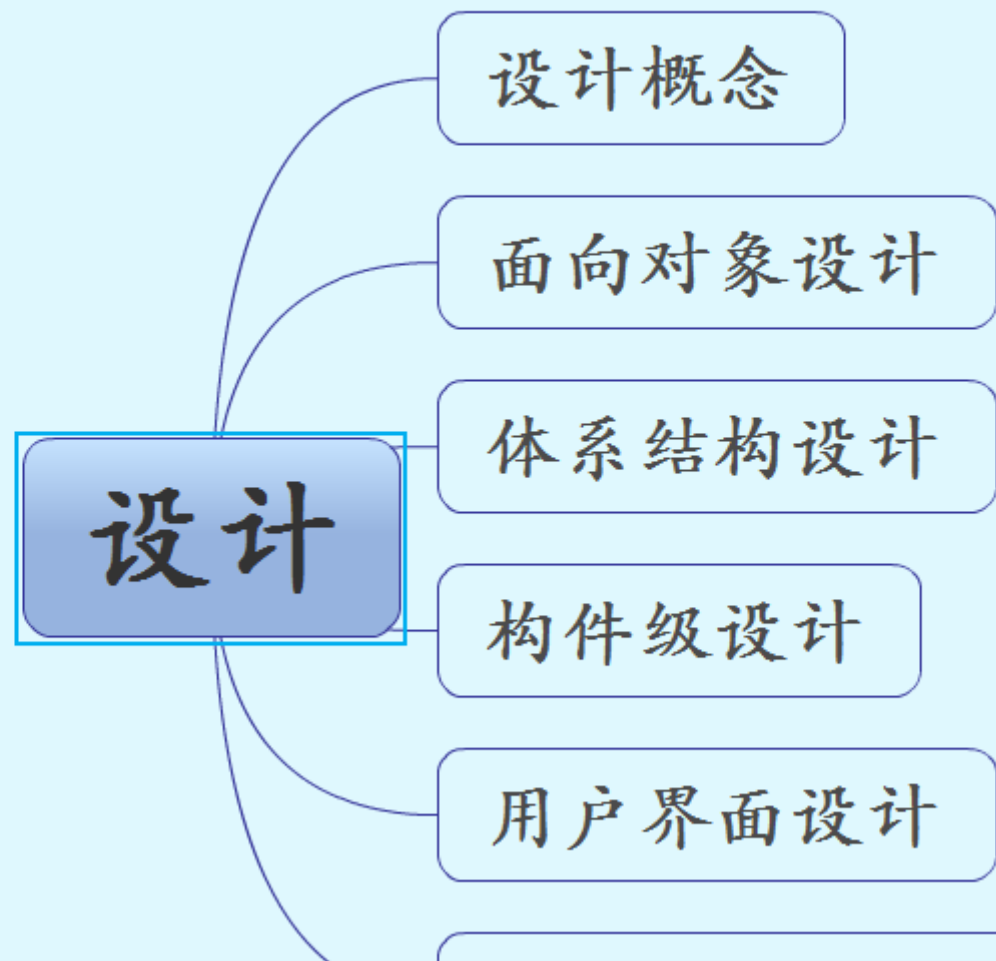


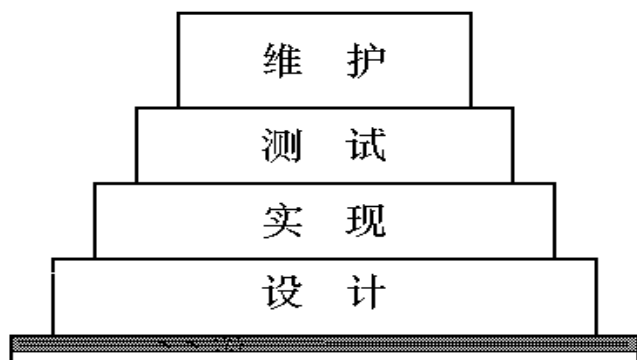
设计 (Design)

outline

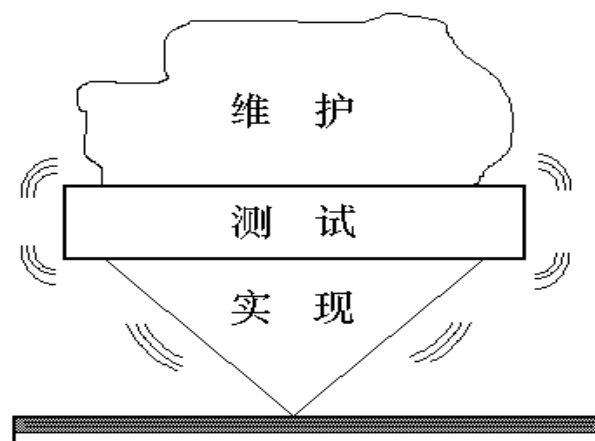


软件设计

- 软件设计包括一系列原理、概念和实践，指导高质量的系统或产品开发
 - 设计原理建立了指导设计工作的最重要原则
 - 在运用设计实践的技术和方法之前，必须先理解设计概念
- 设计是软件工程是否成功的关键

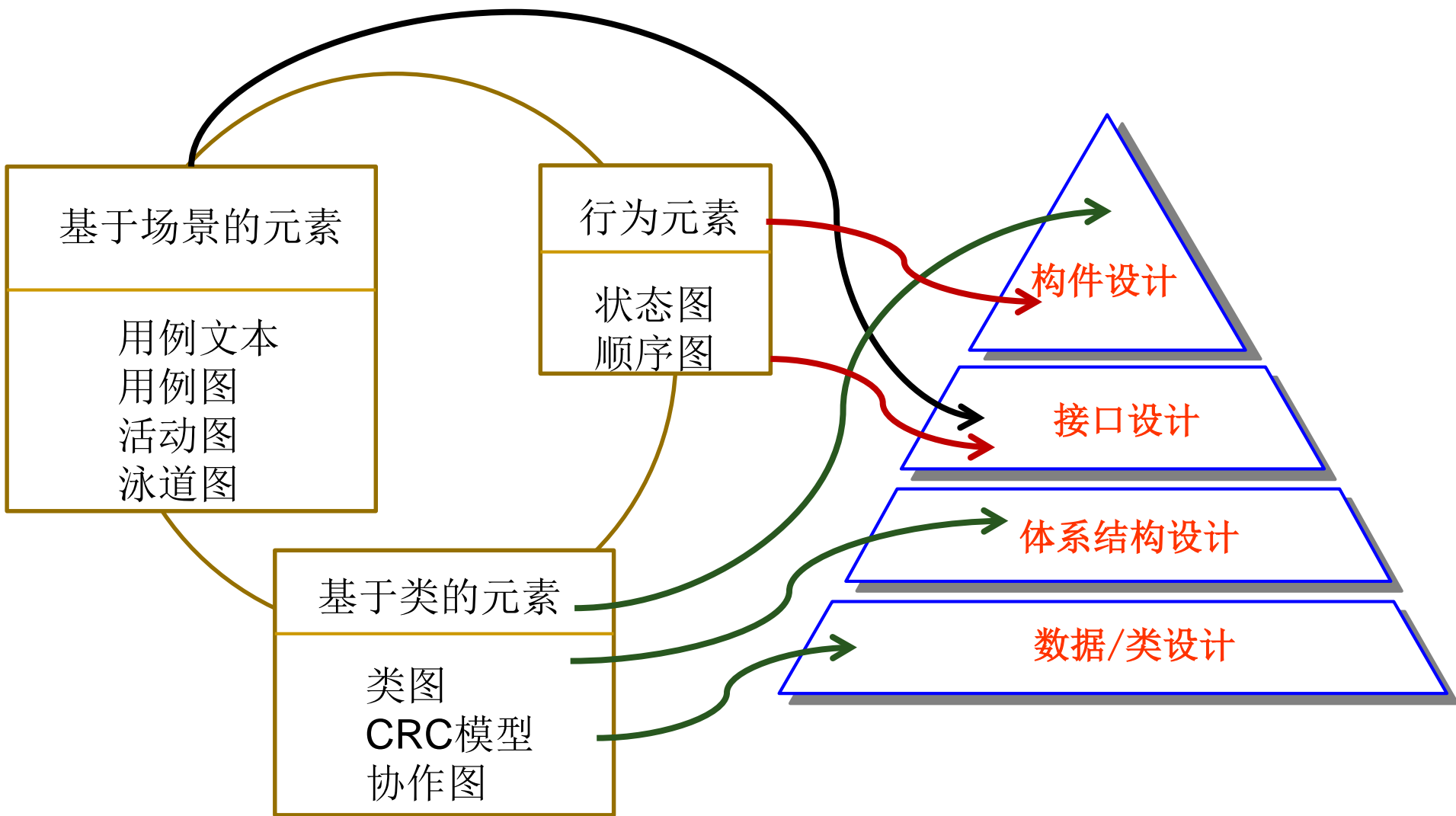


有软件设计



没有软件设计

分析模型 → 设计模型



设计概述

- 从技术的角度，面向对象方法将软件设计划分为**数据/类设计、体系结构设计、接口设计和构件设计**4部分。
 - 数据/类设计：将类模型转化为设计类的实现和必要的数据结构。
 - 体系结构设计：定义软件系统各主要成份之间的关系。
 - 接口设计：软件内部各成份之间、软件与其它协同系统之间及软件与用户之间的交互机制。
 - 构件设计：把软件系统各主要成份转换成软件构件的过程性描述。

设计概述

- 从工程管理的角度，可以将软件设计分为**总体设计**和**详细设计**。
 - 总体设计，将软件需求转化为数据结构和软件的系统结构。
 - 详细设计，通过对结构表示进行细化，得到软件的详细的数据结构和算法。

设计与质量

- 设计应当实现所有包含在需求模型中的明确需求，而且必须满足利益相关者期望的所有隐含需求。
- 对于编码、测试和维护人员，设计应当是可读的、可理解的指南。
- 设计应当提供软件的全貌，从实现的角度说明数据域、功能域和行为域。

软件质量指导原则

- 设计应展示出如下**体系结构**
 - 使用可识别的**体系结构风格**或模式创建
 - 由展示出良好设计特征的**构件**构成
 - 能够以演化的方式实现，从而便于实现和测试
- **设计应该模块化**；应将软件逻辑地划分为子系统。
- 设计应该包含数据、体系结构、接口和构件的清晰表示。

软件质量指导原则

- 设计应导出**数据结构**，这些数据结构适用于要实现的类，并从可识别的数据模式提取。
- 设计应导出显示独立功能特征的**构件**。
- 设计应导出**接口**，这些接口降低了构件之间以及与外部环境连接的复杂性。
- 设计的导出应根据软件需求分析过程中获取的信息采用可重复的方法进行。
- 设计应使用能够有效的表示法来传达其意义。

软件质量属性（FURPS）

- 功能性
 - 安全性
- 易用性
 - 人员因素、整体美感、一致性和文档
- 可靠性
 - 故障的频率和严重性、平均故障时间、故障恢复能力
- 性能
 - 处理速度、响应时间、效率
- 可支持性
 - 可维护性、可测试性、兼容性

软件质量属性注意点

- 不是每个软件质量属性都具有相同的权重
- 必须在设计开始时就考虑质量属性，而不是在设计完成后和构建开始时才考虑

设计过程

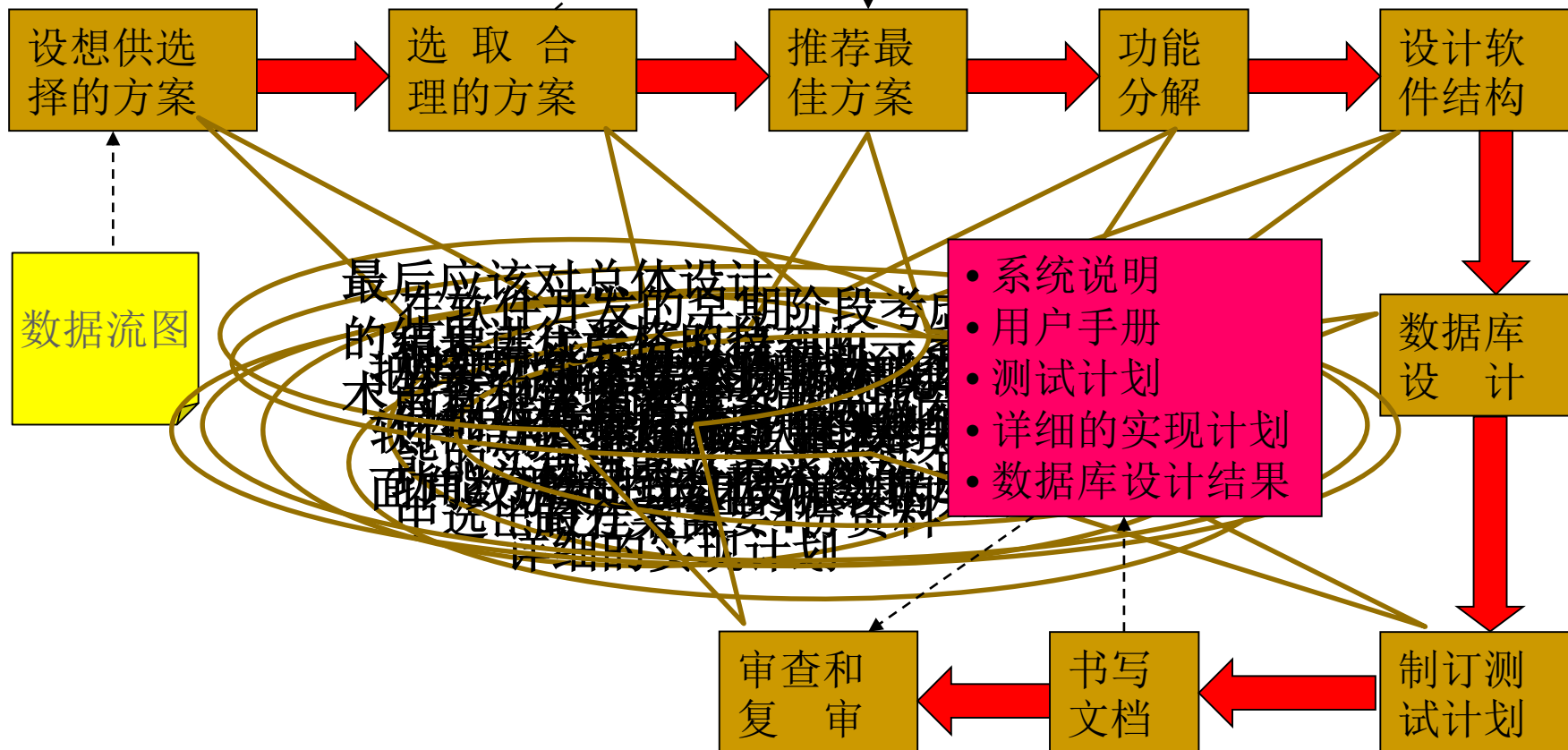
从回答“做什么”到回答“怎样做”

由两个主要阶段组成：

- **系统设计阶段**，确定系统的具体实现方案
 - 设想供选择的方案
 - 选取合理的方案
 - 推荐最佳方案
- **结构设计阶段**，确定软件结构
 - 功能分解
 - 设计软件结构
 - 设计数据库
 - 制定测试计划
 - 书写文档
 - 审查和复审

设计过程

- 系统流程图
- 组成系统的物理元素清单
- 成本/效益分析
- 实现系统的进度计划



设计原理

- **模块化(Modularity)**
- **抽象(Abstraction)**
- **逐步求精(Stepwise Refinement)**
- **信息隐藏(Information Hiding)**
- **模块独立**

模块化(Modularity)

- **模块**：独立命名的、可处理的构件。
- 模块是构成程序的基本构件。
- **模块化**：把程序划分成独立命名且可独立访问的模块，每个模块完成一个子功能，把这些模块集成起来构成一个整体，可以完成指定的功能满足用户的需求。
- 如果一个大型程序仅由一个模块组成，它将很难被人所理解，很难调试和维护。
- **问题**：对于特定的软件设计来说，多少数量的模块是“合适的”？

一般规律

- 设函数 $C(x)$ 定义问题 x 的复杂程度，函数 $E(x)$ 定义解决问题 x 需要的工作量（时间）。对于两个问题 $P1$ 和 $P2$ ，如果：

- 如果 $C(P1) > C(P2)$ ，显然 $E(P1) > E(P2)$
- 根据人类解决一般问题的经验，

$$C(P1+P2) > C(P1) + C(P2)$$

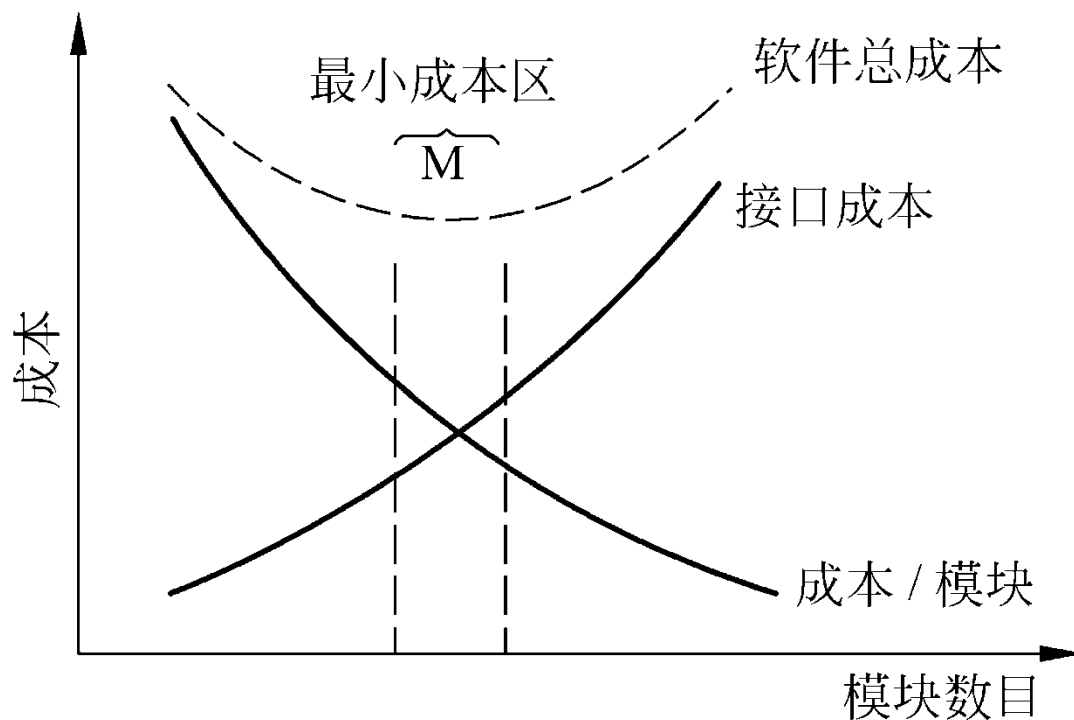
- 综上所述，得到下面的不等式

$$E(P1+P2) > E(P1) + E(P2)$$

如果无限制的划分软件，那么开发软件所需的工作量将会小到忽略不计吗？

总成本曲线

- 每个程序都相应地有一个最适当的模块数目 M ，使得系统的开发成本最小。



模块化和软件成本

抽象(Abstraction)

- **“Abstraction is one of the fundamental ways that we as humans cope with complexity.”**

——Grady Booch

- **抽象：**抽出事物的本质特性而暂时不考虑它们的细节。

一般抽象过程

- 处理复杂系统的有效方法是用层次的方式构造和分析它。
 - 在最高的抽象级上，使用问题所处环境的语言以概括性的术语描述解决方案。
 - 在较低的抽象级上，将提供更详细的解决方案说明。
 - 最后，在最低的抽象级上，以一种能直接实现的方式陈述解决方案。

抽象

- **数据抽象**是一个数据结构以及该数据结构上的操作。
- **过程抽象**是一个命名的指令序列，具有特定和有限的功能。

Implementation of
job_queue

```
Initialize_job_queue()  
{  
    .....  
}
```

```
add_job_to_queue(job j)  
{  
    .....  
}
```

```
remove_job_from_queue(job j)  
{  
    .....  
}
```

代码示例

```
Class JobQueueClass {  
    // attributes  
    public int queueLength;           // length of job queue  
    public int queue[] = new int[25]; // queue can contain 25 jobs  
    // methods  
    public void initializeJobQueue() {  
        queueLength = 0; // an empty job queue has length 0  
    }  
  
    public void addJobToQueue(int jobNumber) {  
        queue[queueLength] = jobNumber; // add the job to the end  
        queueLength = queueLength + 1;  
    }  
  
    public int removeJobFromQueue() {  
        int jobNumber = queue[0]; // set jobNumber to the head  
        queueLength = queueLength - 1; // remove the head  
        for (int k = 0; k < queueLength; k++)  
            queue[k] = queue[k + 1]; // move up the remaining jobs  
        return jobNumber;  
    }  
}
```

代码示例

```
Class SchedulerClass {  
    .....  
    public void queueHandler() {  
        int          jobA, jobB  
        JobQueueClass jobQueueJ = new JobQueueClass();  
        jobQueueJ.initializeJobQueue();  
        jobQueueJ.addJobToQueue(jobA);  
        jobB = jobQueueJ.removeJobFromQueue();  
    }  
}
```

逐步求精(Stepwise Refinement)

- 逐步求精是人类解决复杂问题时采用的基本方法，也是许多软件工程技术的基础。
- **逐步求精**：为了能集中精力解决主要问题而尽量推迟对问题细节的考虑。
- **Miller法则**：一个人在任何时候都只能把注意力集中在 (7 ± 2) 个知识块上。

G.A. Miller

Magical Number Seven, Plus or Minus Two, Some Limits on Our Capacity for Processing Information

The Psychological Review, 1956

Wirth本人对逐步求精策略的概括说明：

- 我们对付复杂问题的最重要的办法是抽象，因此，对一个复杂的问题不应该立刻用计算机指令、数字和逻辑符号来表示，而应该用较自然的抽象语句来表示，从而得出抽象程序。
- 抽象程序对抽象的数据进行某些特定的运算并用某些合适的记号（可能是自然语言）来表示。对抽象程序做进一步的分解，并进入下一个抽象层次，这样的精细化过程一直进行下去，直到程序能被计算机接受为止。这时的程序可能是用某种高级语言或机器指令书写的。

信息隐藏(Information Hiding)

- **信息隐藏：**应该这样设计和确定模块，使得一个模块内包含的信息(过程和数据)对于不需要这些信息的模块来说，是不能访问的。
- **细节隐藏**

代码示例

```
Class JobQueueClass {  
    // attributes  
    public int queueLength;           // length of job queue  
    public int queue[] = new int[25]; // queue can contain 25 jobs  
    // methods  
    public void initializeJobQueue() {  
        queueLength = 0;           // an empty job queue has length 0  
    }  
  
    public void addJobToQueue(int jobNumber) {  
        queue[queueLength] = jobNumber; // add the job to the end  
        queueLength = queueLength + 1;  
    }  
  
    public int removeJobFromQueue() {  
        int jobNumber = queue[0]; // set jobNumber to the head  
        queueLength = queueLength - 1; // remove the head  
        for (int k = 0; k < queueLength; k++)  
            queue[k] = queue[k + 1]; // move up the remaining jobs  
        return jobNumber;  
    }  
}
```

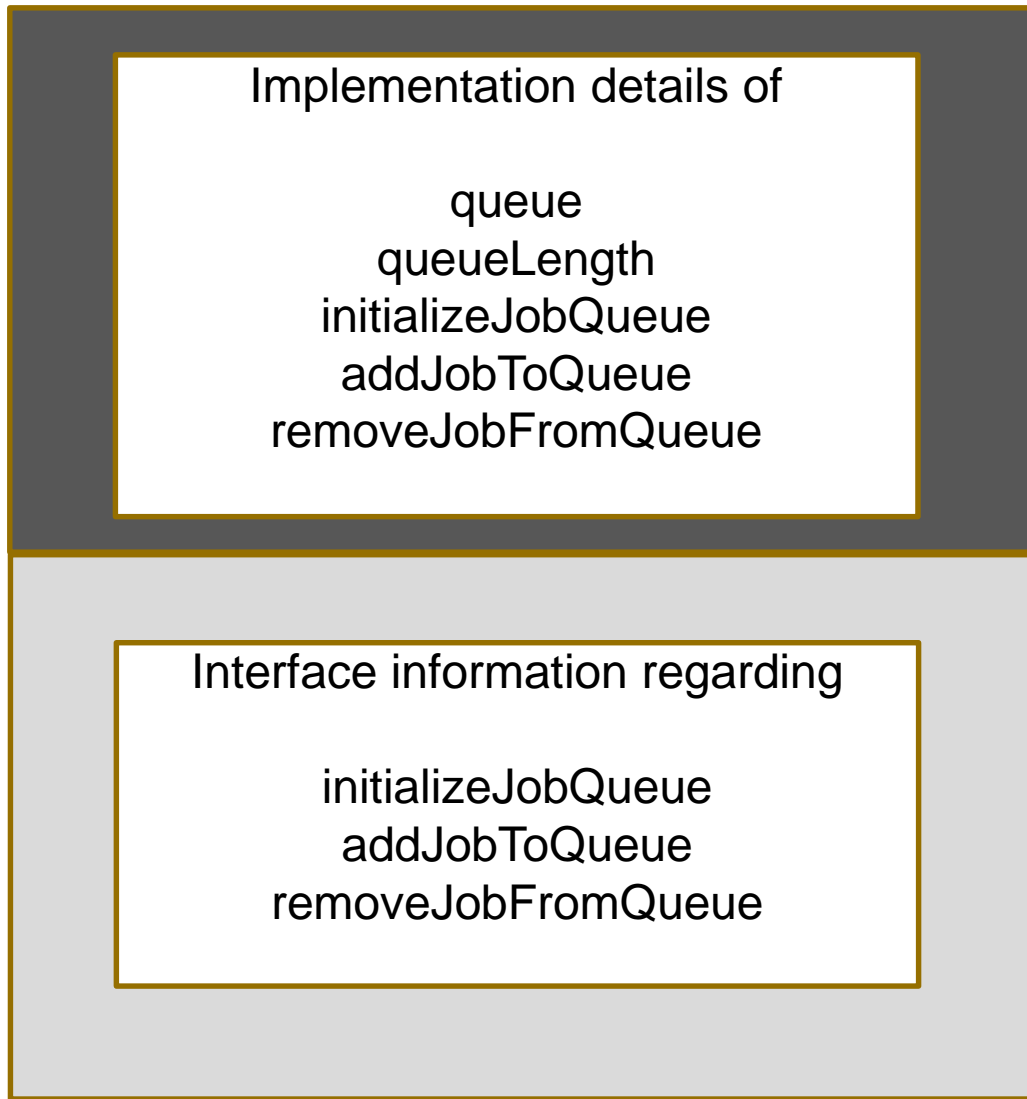
代码示例

```
Class SchedulerClass {  
    .....  
    public void queueHandler() {  
        int          jobA, jobB  
        JobQueueClass jobQueueJ = new JobQueueClass();  
  
        jobQueueJ.queue[7] = -5678;  
  
        jobQueueJ.initializeJobQueue();  
        jobQueueJ.addJobToQueue(jobA);  
        jobB = jobQueueJ.removeJobFromQueue();  
    }  
}
```

代码示例

```
Class JobQueueClass {  
    // attributes  
    private int queueLength;           // length of job queue  
    private int queue[] = new int[25]; // queue can contain 25 jobs  
    // methods  
    public void initializeJobQueue() {  
        queueLength = 0; // an empty job queue has length 0  
    }  
  
    public void addJobToQueue(int jobNumber) {  
        queue[queueLength] = jobNumber; // add the job to the end  
        queueLength = queueLength + 1;  
    }  
  
    public int removeJobFromQueue() {  
        int jobNumber = queue[0]; // set jobNumber to the head  
        queueLength = queueLength - 1; // remove the head  
        for (int k = 0; k < queueLength; k++)  
            queue[k] = queue[k + 1]; // move up the remaining jobs  
        return jobNumber;  
    }  
}
```

JobQueueClass



Invisible



visible

模块独立

- 模块独立的概念是模块化、抽象、信息隐藏概念的直接结果。
- 具有独立模块的软件较容易开发、测试和维护。
- **好设计的关键：**每个模块完成一个相对独立的子功能，并且和其他模块间的接口简单，即模块独立。

模块独立程度的两个定性标准度量：

- 耦合(Coupling)衡量不同模块彼此间互相依赖(连接)的紧密程度。耦合要低，即每个模块和其他模块之间的关系要简单；
- 内聚(Cohesion)衡量一个模块内部各个元素彼此结合的紧密程度。内聚要高，每个模块完成一个相对独立的特定子功能。

耦合

- **耦合**：是对一个软件结构内不同模块之间互连程度的度量。
- **要求**：在软件设计中应该追求尽可能松散耦合的系统。
 - 可以研究、测试或维护任何一个模块，而不需要对系统的其他模块有很多了解；
 - 模块间联系简单，发生在一处的错误传播到整个系统的可能性就很小；
 - 模块间的耦合程度强烈影响系统的可理解性、可测试性、可靠性和可维护性。

分类

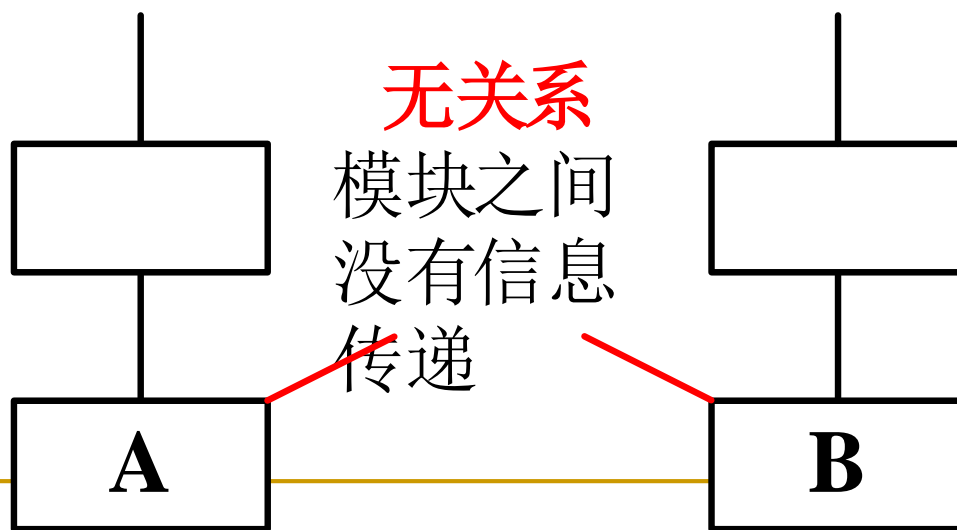
- 非直接耦合 (**Good**)
- 数据耦合
- 特征耦合
- 控制耦合
- 外部耦合
- 公共耦合
- 内容耦合 (**Bad**)

非直接耦合

😊 The most desirable.

(1) 非直接耦合/完全独立(no direct coupling)

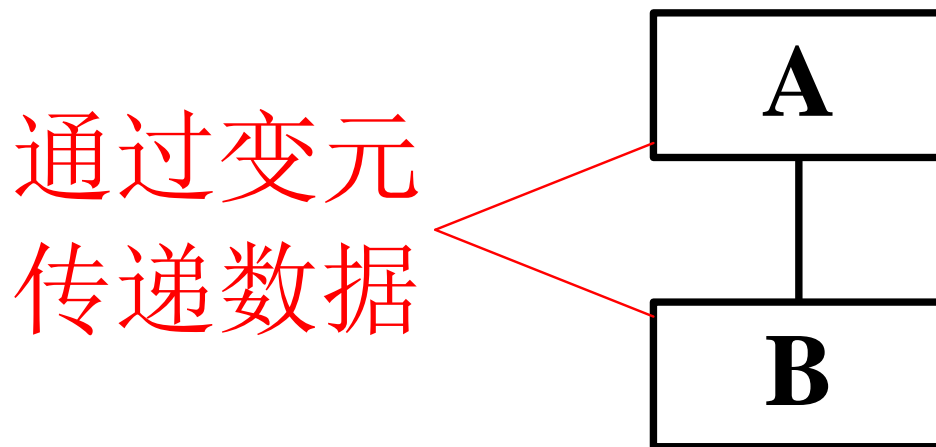
- 如果两个模块中的每一个都能独立地工作而不需要另一个模块的存在，那么它们完全独立。
- 在一个软件系统中不可能所有模块之间都没有任何连接。



数据耦合

(2) 数据耦合(data coupling)

- 如果两个模块彼此间通过参数交换信息，而且交换的信息仅仅是数据，那么这种耦合称为数据耦合。



数据耦合

评价：

- 数据耦合是低耦合。
- 系统中至少必须存在这种耦合。一般说来，一个系统内可以只包含数据耦合。
- 维护更容易，对一个模块的修改不会使另一个模块产生退化错误。

特征耦合

(3) 特征耦合(stamp coupling)

- 把整个数据结构（记录、数组等）作为参数传递而被调用的模块只需要使用其中一部分数据元素

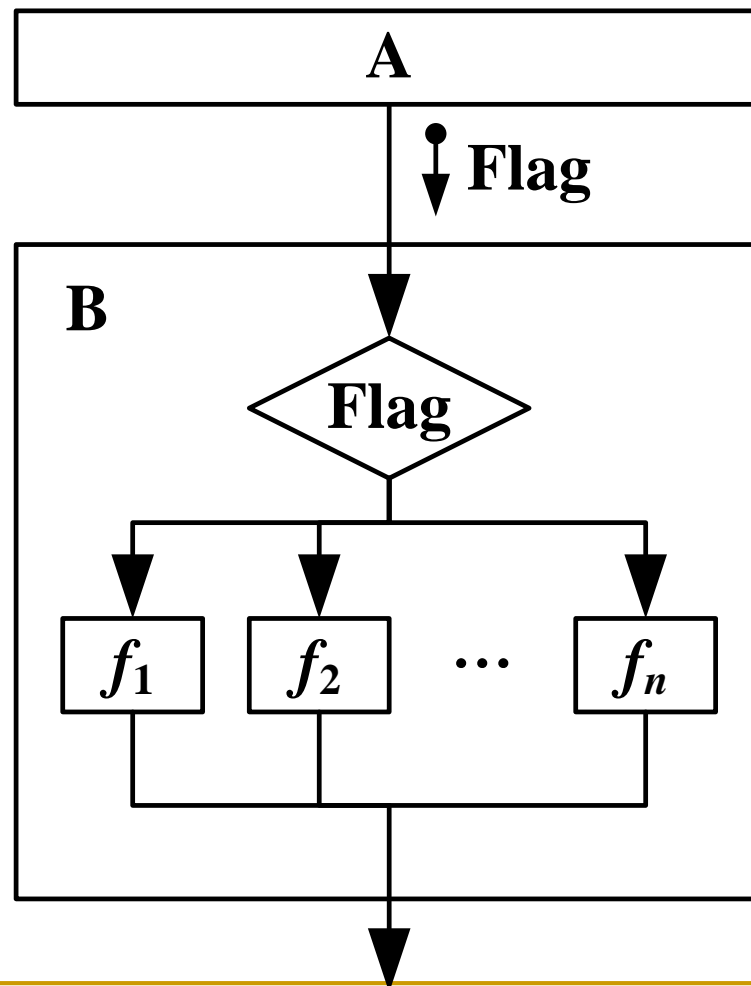
评价：

- 被调用的模块可使用的数据多于它确实需要的数据，这将导致对数据的访问失去控制。
- 无论何时把指针作为参数进行传递，都应该仔细检查该耦合。

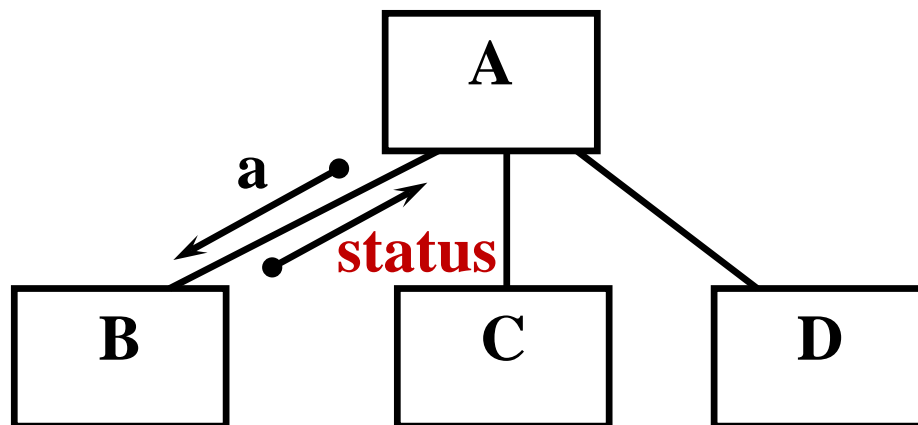
控制耦合

(4) 控制耦合(control coupling)

- 如果两个模块彼此间传递的信息中有控制信息，这种耦合称为控制耦合。
- One module passes parameters to control the activity of another module.



示例



图中模块A的内部处理程序判断是执行C还是执行D，要取决于模块B传来的信息状态（**status**）。

外部耦合

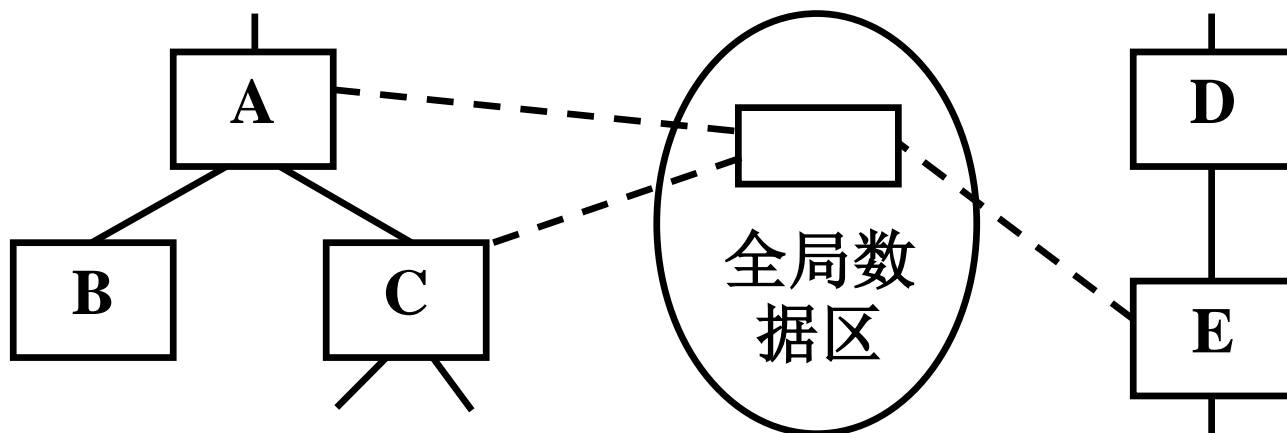
(5) 外部耦合(external coupling)

- 当模块与外部环境（操作系统函数、通信功能等）连接在一起，并受到约束时就出现较高程度的耦合。
- 在I/O中，把一个模块耦合到指定的设备、格式和通信协议上。
- 外部耦合是必不可少的，但必须加以限制。

公共耦合

(6) 公共耦合(common coupling)

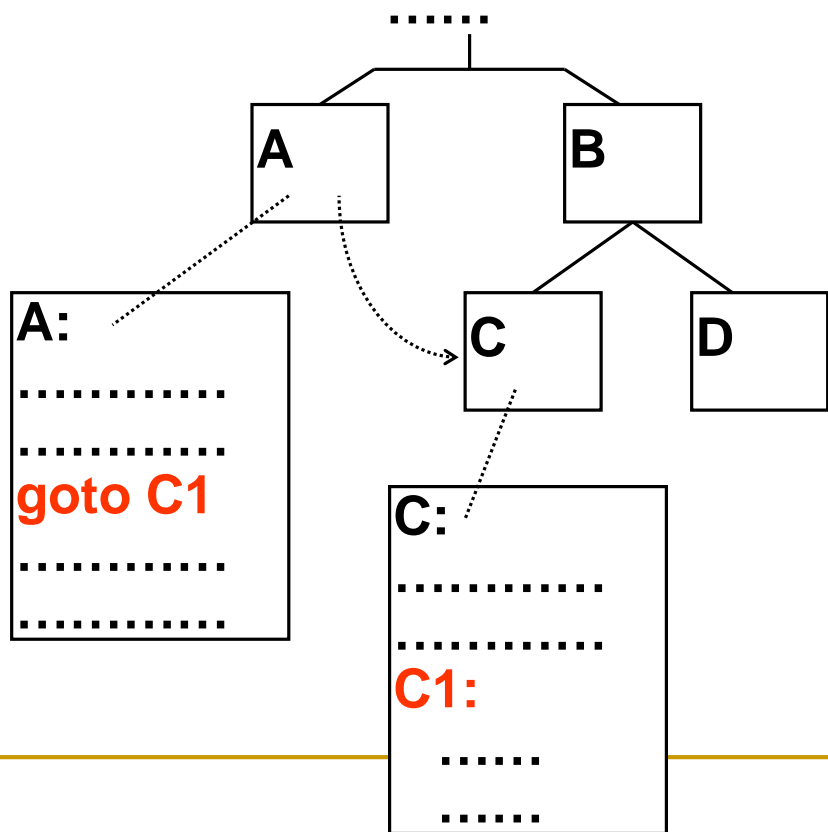
- 两个或多个模块通过一个公共数据环境相互作用
 - 访问全局变量
 - 读写同一个数据库的同一个记录



内容耦合

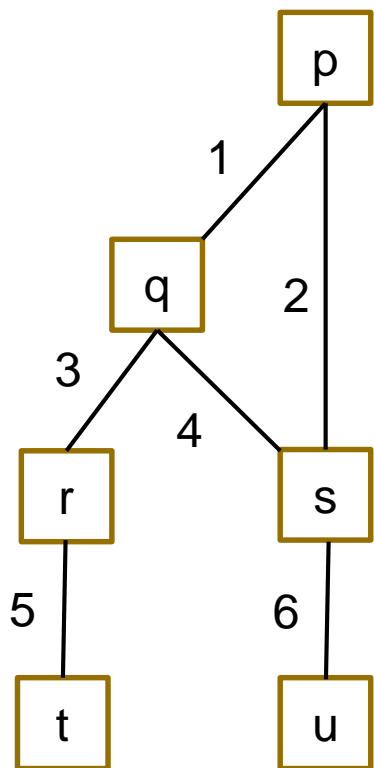
(7) 内容耦合(content coupling)

- ❑ 一个模块访问另一个模块的内部数据
- ❑ 一个模块不通过正常入口转到另一个模块的内部



☹ **The least desirable
must be avoided**

耦合实例



模块连线图

其中，**p, t, u**访问同一数据库

接口描述表

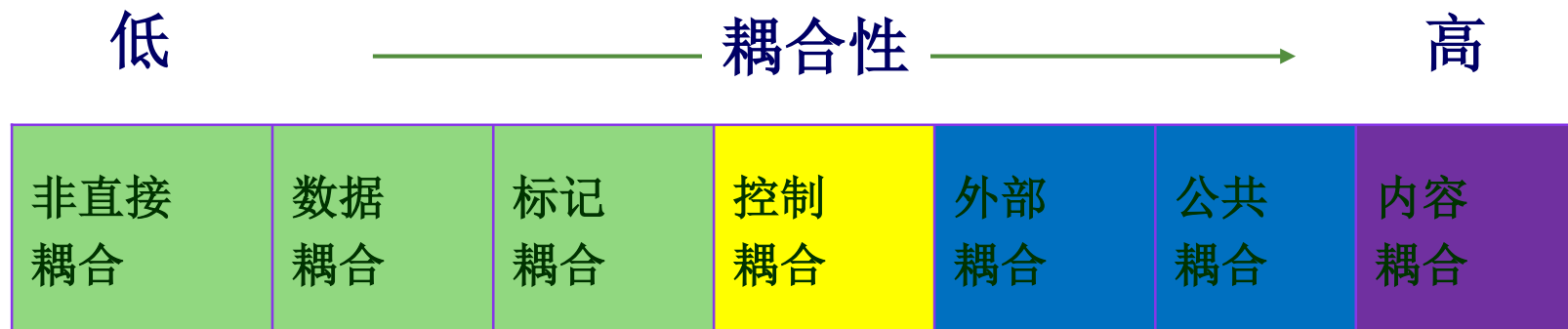
Number	In	Out
1	aircraft_type	Status_flag
2	list_of_aircraft_parts	-----
3	function_code	-----
4	list_of_aircraft_parts	-----
5	part_number	part_manufacturer
6	part_number	part_name

模块间耦合关系表

	q	r	s	t	u
p	数据	----	数据或特征	公共	公共
q		控制	数据或特征	----	----
r			----	数据	----
s				----	数据
t					公共

小结

- **Goal:** as loose as possible



小结

- 应该采取下述设计原则：

尽量使用数据耦合，

少用控制耦合和特征耦合，

限制公共环境耦合的范围，

完全不用内容耦合。

内聚

- **内聚：**标志一个模块内各个元素彼此结合的紧密程度。简单地说，理想内聚的模块只做一件事情。
- **要求：**设计时应该力求做到高内聚。
- 内聚和耦合是密切相关的，模块内的高内聚往往意味着模块间的松耦合。实践表明内聚更重要，应该把更多注意力集中到提高模块的内聚程度上。

分类

- 偶然内聚 (**bad**)
- 逻辑内聚
- 时间内聚
- 过程内聚
- 通信内聚
- 顺序内聚
- 功能内聚 (**good**)

偶然内聚

(1) 偶然内聚(**coincidental cohesion**)

- 一个模块执行多个完全不相关的操作

```
.....  
print_the_next_line  
reverse_the_string_of_characters_comprising_the_second_argument  
add_7_to_the_fifth_argument  
convert_the_fourth_argument_to_floating_point  
.....
```


偶然内聚

评价：

- 可理解性差，可维护性产生退化；
- 模块是不可重用的。

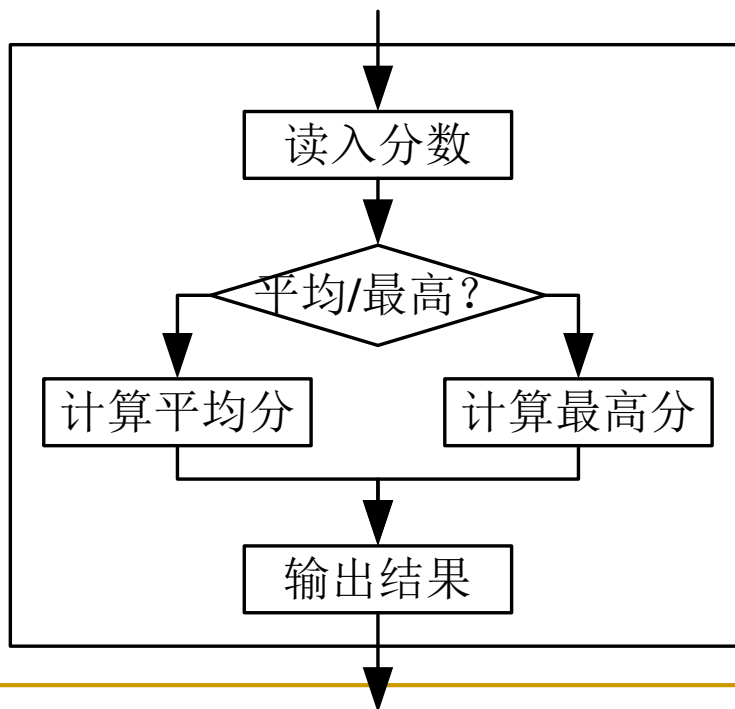
解决方案：

- 将模块分成更小的模块，每个小模块执行一个操作。

逻辑内聚

(2) 逻辑内聚(logical cohesion)

- 把几种功能组合在一起，每次调用时，则由传递给模块的判定参数来确定该模块应执行哪一种功能。



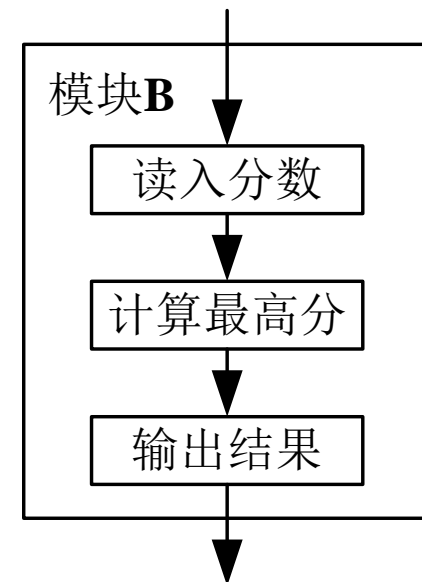
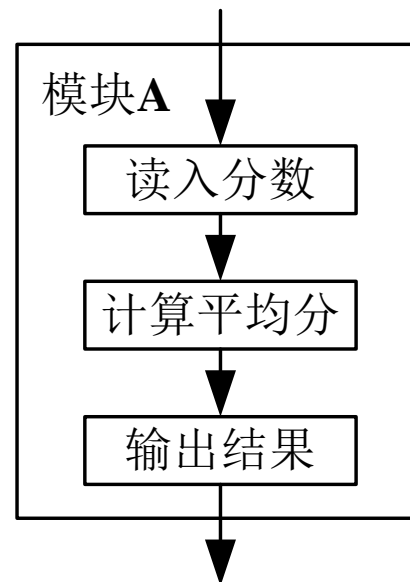
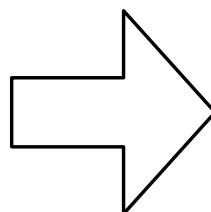
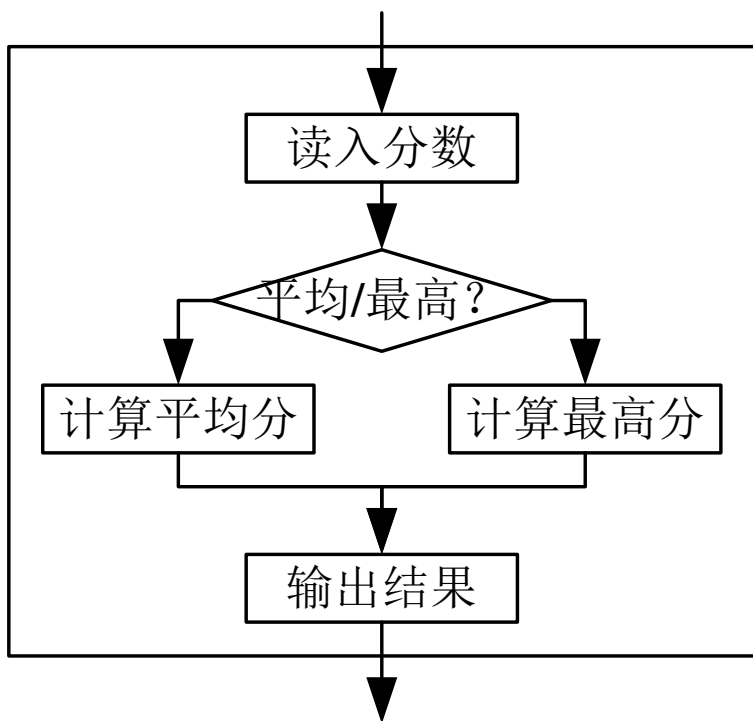
逻辑内聚

评价：

- 接口难以理解，造成整体上不易理解；
- 完成多个操作的代码互相纠缠在一起，即使局部功能的修改有时也会影响全局，导致严重的维护问题；
- 难以重用。

解决方案：

- 模块分解。



时间内聚

(3) 时间内聚(temporal cohesion)

- 一个模块包含的任务必须在同一段时间内执行
- 将多个变量的初始化放在同一个模块中实现
- 将需要同时使用的多个库文件的打开操作放在同一个模块中

时间内聚

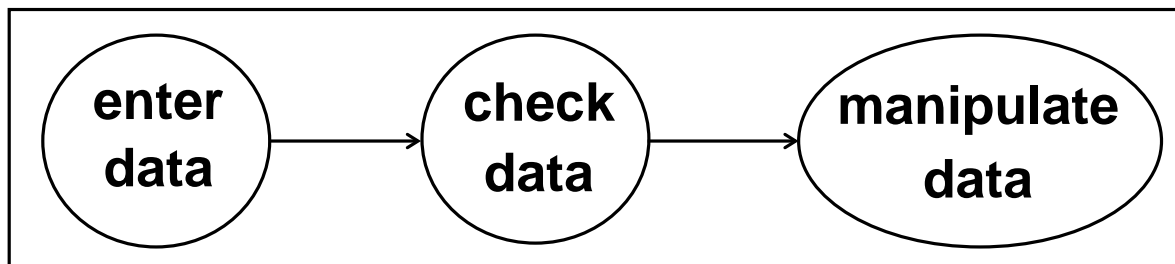
评价：

- 时间关系在一定程度上反映了程序某些实质，所以时间内聚比逻辑内聚好一些。
- 模块内操作之间的关系很弱，与其他模块的操作却有很强的关联。
- 时间内聚的模块不太可能重用。

过程内聚

(4) 过程内聚(procedural cohesion)

- 一个模块内的处理元素是相关的，而且必须以特定次序执行。



评价:

- 比时间内聚好，至少操作之间是过程关联的。
- 仍是弱连接，不太可能重用模块。

通信内聚

(5) 通信内聚(**communicational cohesion**)

- 如果模块中所有元素都使用同一个输入数据和(或)产生同一个输出数据，则称为通信内聚。即在同一个数据结构上操作。

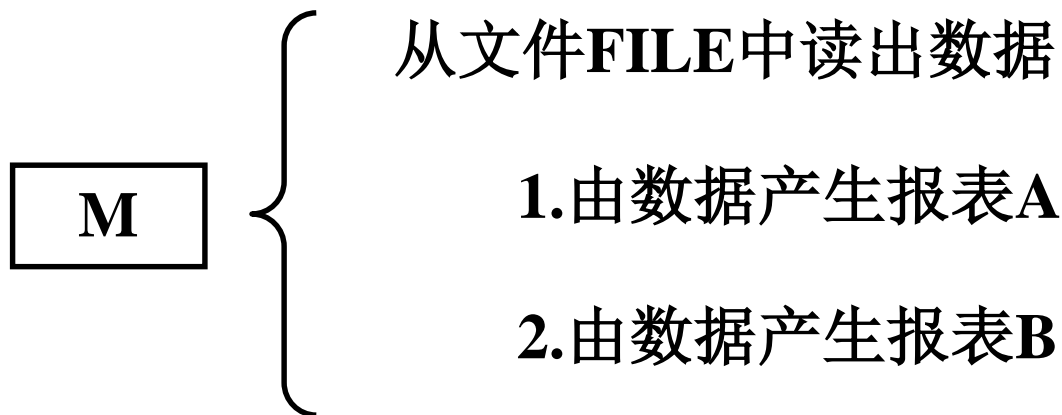
评价：

- 模块中各操作紧密相连，比过程内聚更好。
- 不能重用。

解决方案：

- 分成多个模块，每个模块执行一个操作。

例



- 模块M的处理单元将根据同一个数据文件FILE的数据产生不同的表格。

顺序内聚

(6) 顺序内聚(sequential cohesion)

- 如果一个模块内的处理元素和同一个功能密切相关，而且这些处理必须顺序执行，则称为顺序内聚。
- 通常，顺序内聚中一个处理元素的输出是另一个处理元素的输入。

例

- 有一个按给出的生日计算雇员年龄、退休时间的子程序。
 - 如果它是利用所计算的年龄来确定雇员将要退休的时间，那么它就具有顺序内聚性；
 - 如果它是分别计算年龄和退休时间的，但使用相同生日数据，那它就只具有通信内聚性。

功能内聚

(7) 功能内聚(functional cohesion)

- 如果模块内所有处理元素属于一个整体，完成一个单一的功能，则称为功能内聚。

评价：

- 模块可重用，应尽可能重用；
- 可隔离错误，维护更容易；
- 扩充产品功能时更容易。

例：

- 按给出的生日计算雇员年龄的子程序

小结

七种内聚的优劣评分结果：

- 高内聚：

功能内聚	10分
顺序内聚	9分
- 中内聚：

通信内聚	7分
过程内聚	5分
- 低内聚：

时间内聚	3分
逻辑内聚	1分
偶然内聚	0分
- 设计时力争做到高内聚，并且能够辨认出低内聚的模块。

小结

内聚与耦合的关系：

- 内聚和耦合是密切相关的，与其它模块存在高耦合的模块通常意味着低内聚，而高内聚的模块通常意味着与其它模块之间存在低耦合。
- 模块设计追求高内聚，低耦合。