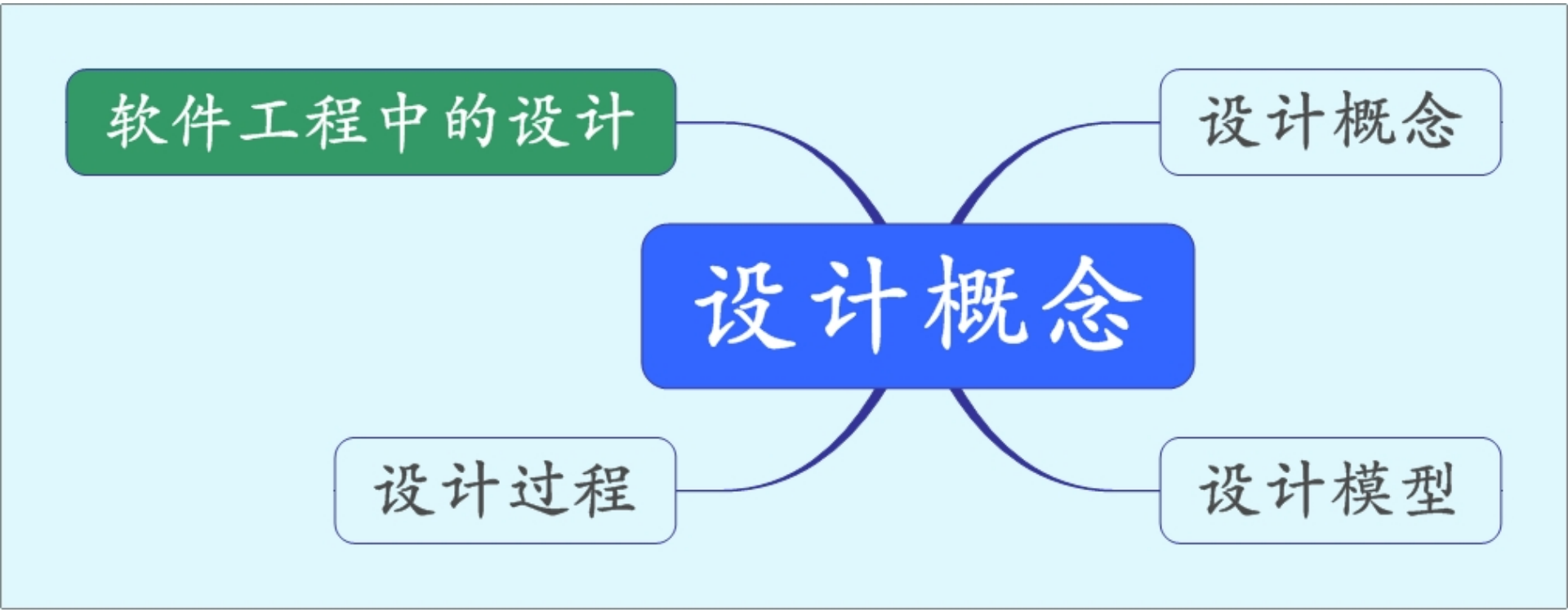
设计概念
(Design Concepts)

outline

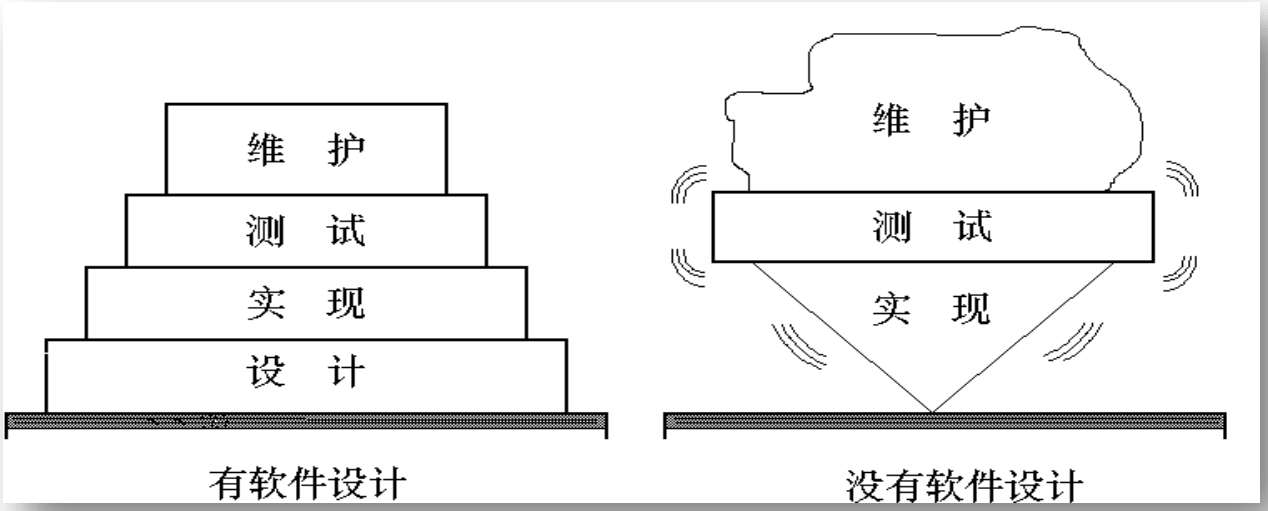


软件设计

- 软件设计是一个创造性的活动，需要基于客户需求识别软件构件及其关系
- 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
  - 设计实践指导随后的实现活动
- 考虑一个设计时通常都应该将实现问题考虑进来

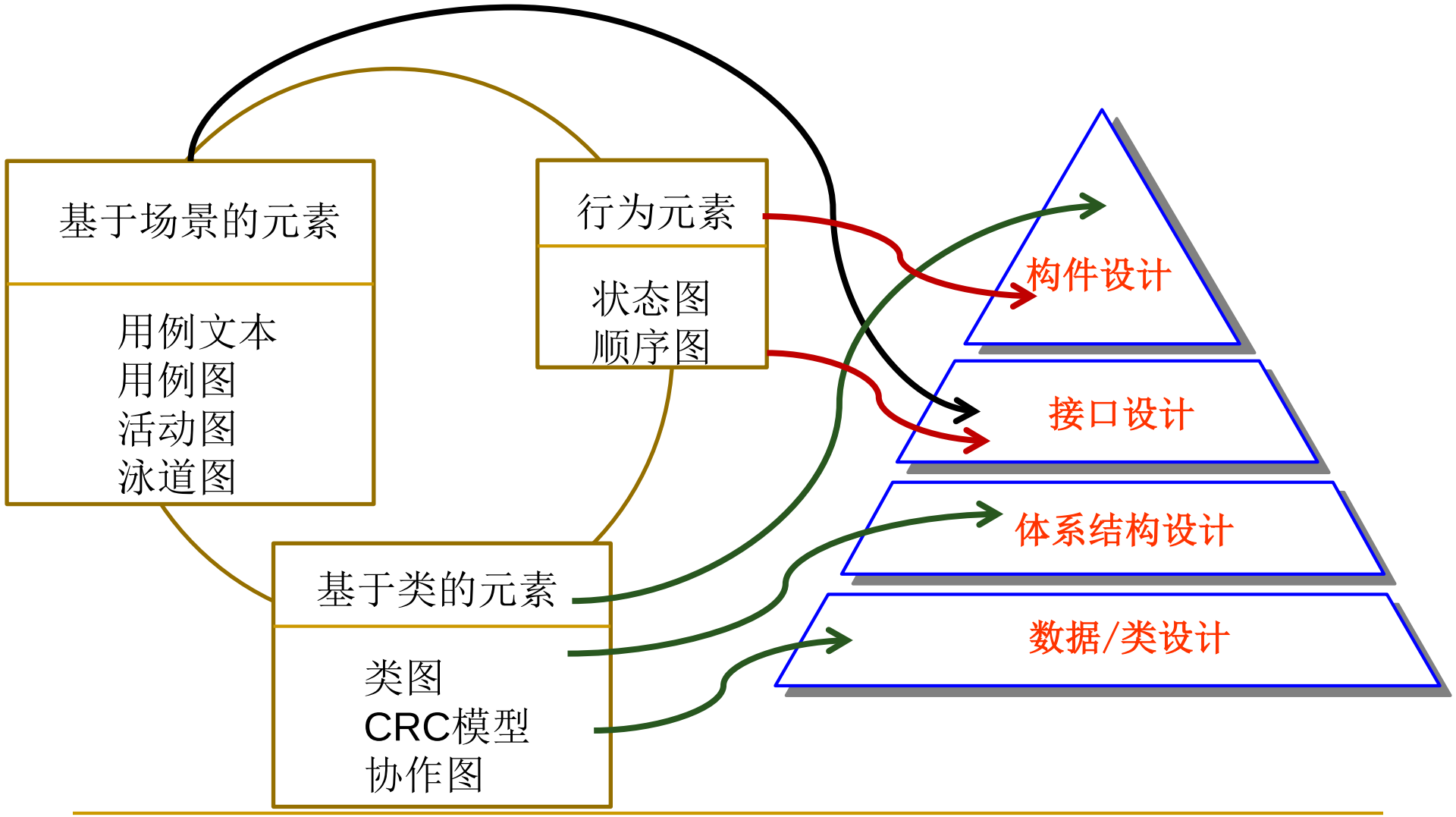
软件设计的重要性

- 设计过程中所做出的决策将最终影响软件构建的成功与否，会影响软件维护的难易程度
- 软件设计是质量形成的地方
  - 设计提供了可用于质量评估的软件表示



- 不稳定的系统
- 难以测试
- 直到后期才能评估其质量

分析模型→设计模型


基于场景的元素
用例文本
用例图
活动图
泳道图
行为元素
状态图
顺序图
基于类的元素
类图
CRC模型
协作图
构件设计
接口设计
体系结构设计
数据/类设计

设计分类

- 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

设计分类

- 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

outline

设计过程的目标

- 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

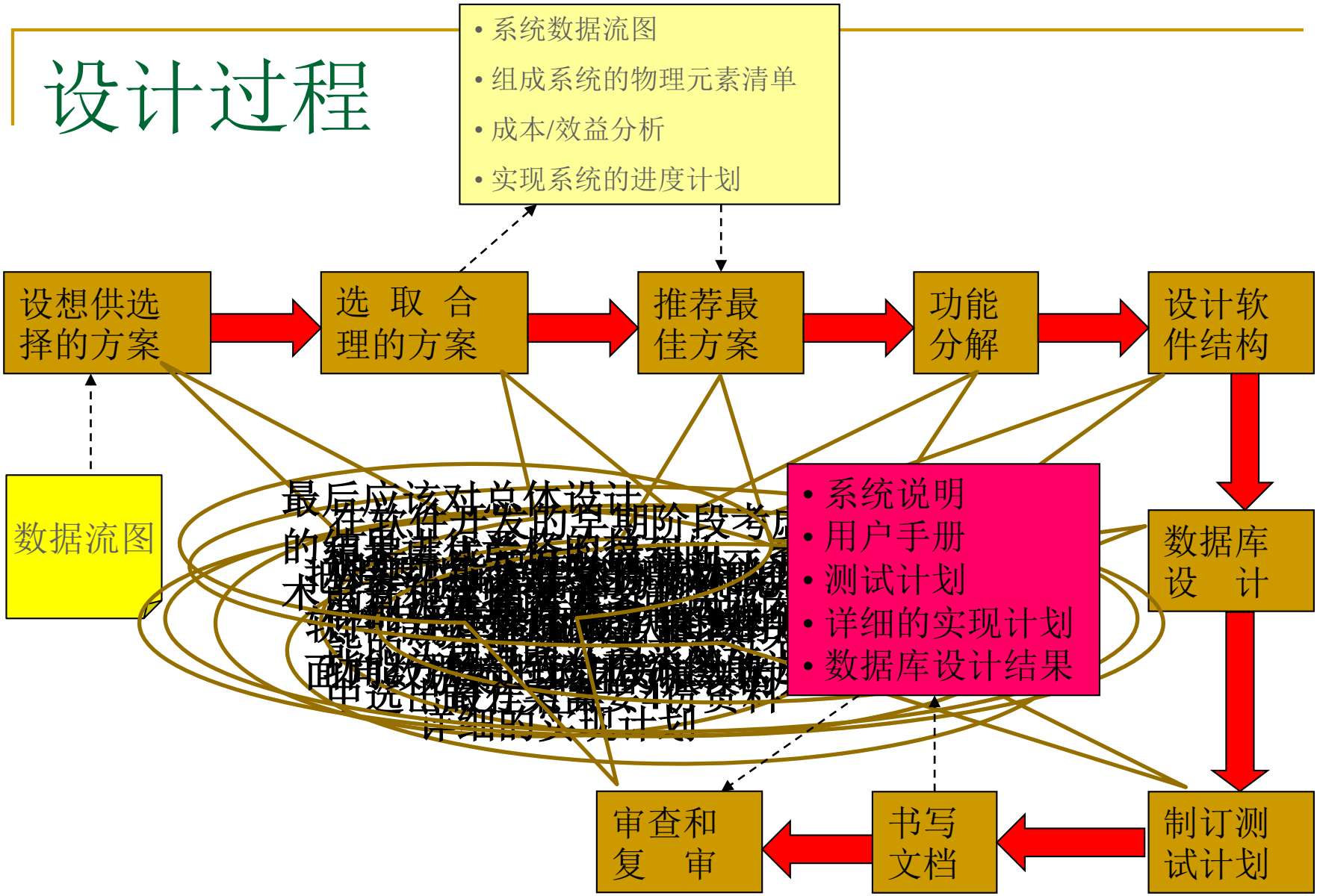
设计过程

从回答“做什么”到回答“怎样做”

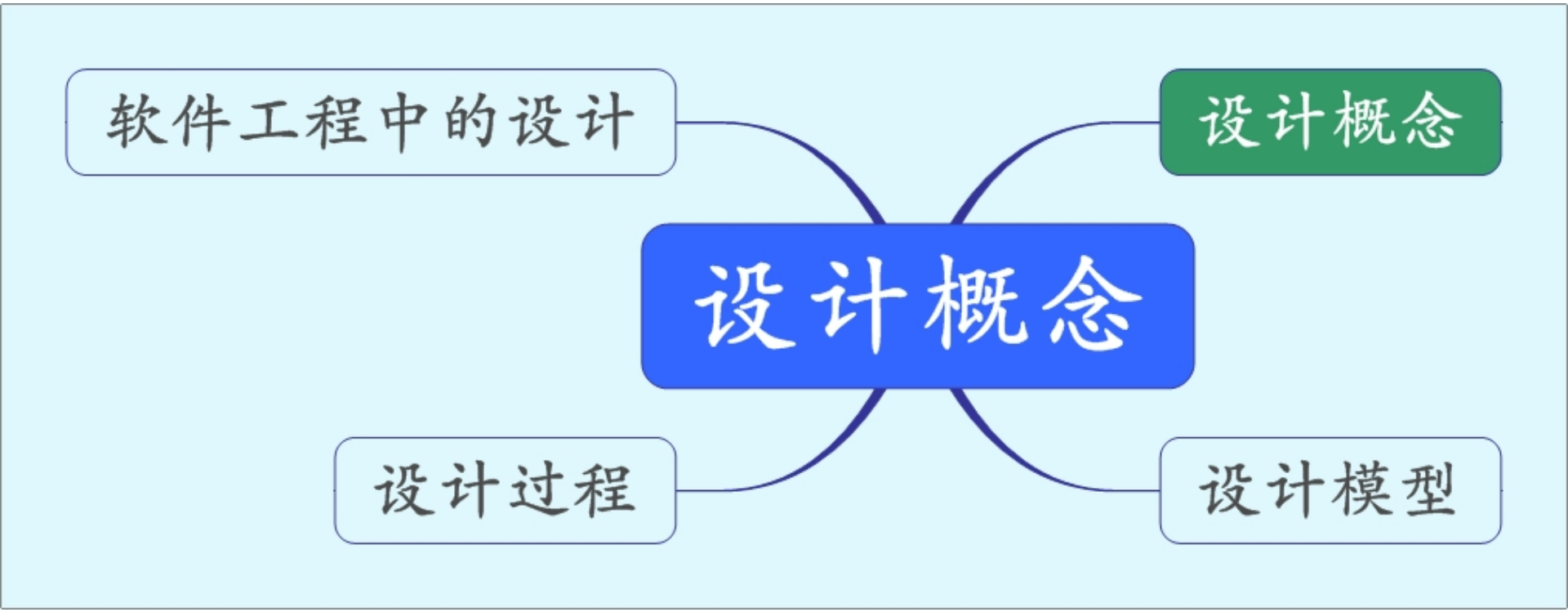
由两个主要阶段组成：

- 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


• 系统数据流图
• 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
设想供选择的方案
选 取 合理的方案
推荐最佳方案
功能分解
设计软件结构
数据流图
数据库设 计
• 系统说明
• 用户手册
• 测试计划
• 详细的实现计划
• 数据库设计结果
审查和复 审
书写文档
制订测试计划

outline



设计概念

- **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(*P*1)>C(*P*2)，显然E(P1)>E(P2)
  - 根据人类解决一般问题的经验，

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

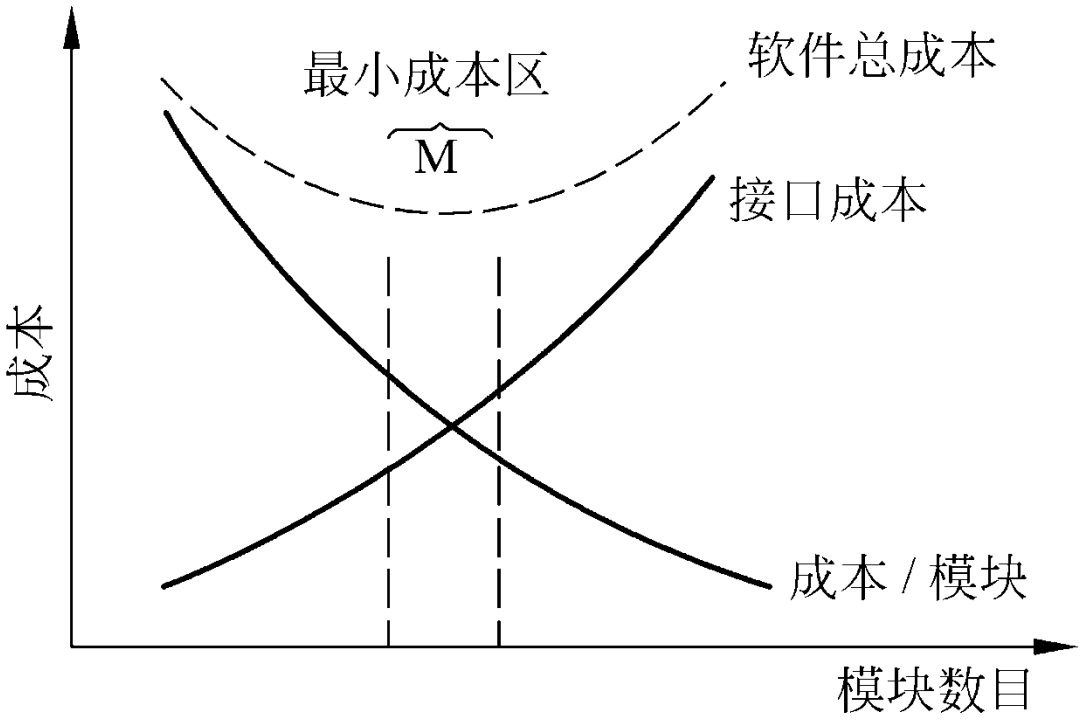
  - 综上所述，得到下面的不等式

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

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

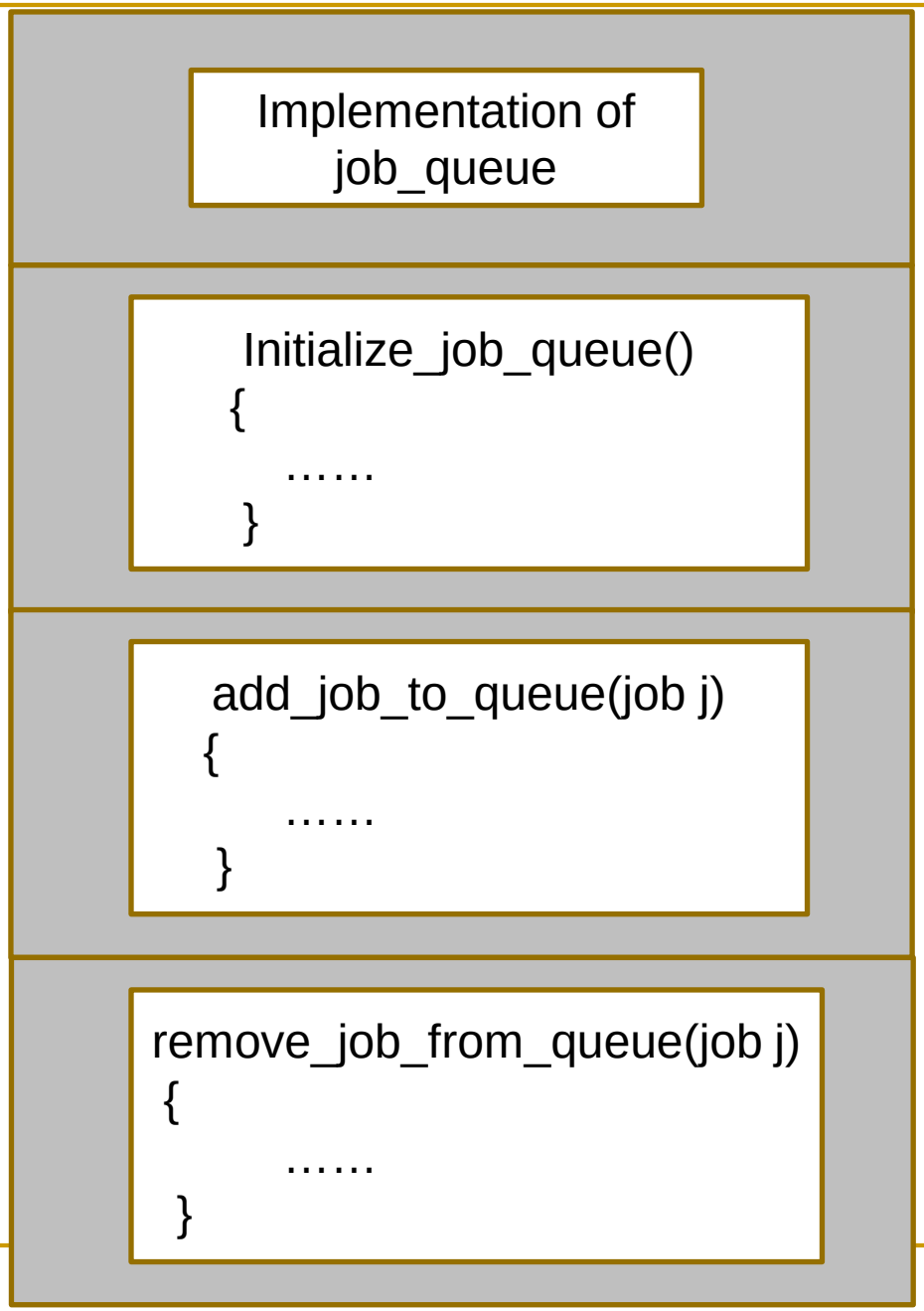
- 抽象：抽出事物的本质特性而暂时不考虑它们的细节。

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

- 数据抽象是描述数据对象的具名数据集合。
- 过程抽象是具有明确和有限功能的指令序列。



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 queque 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 queque 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();
  }
}
```

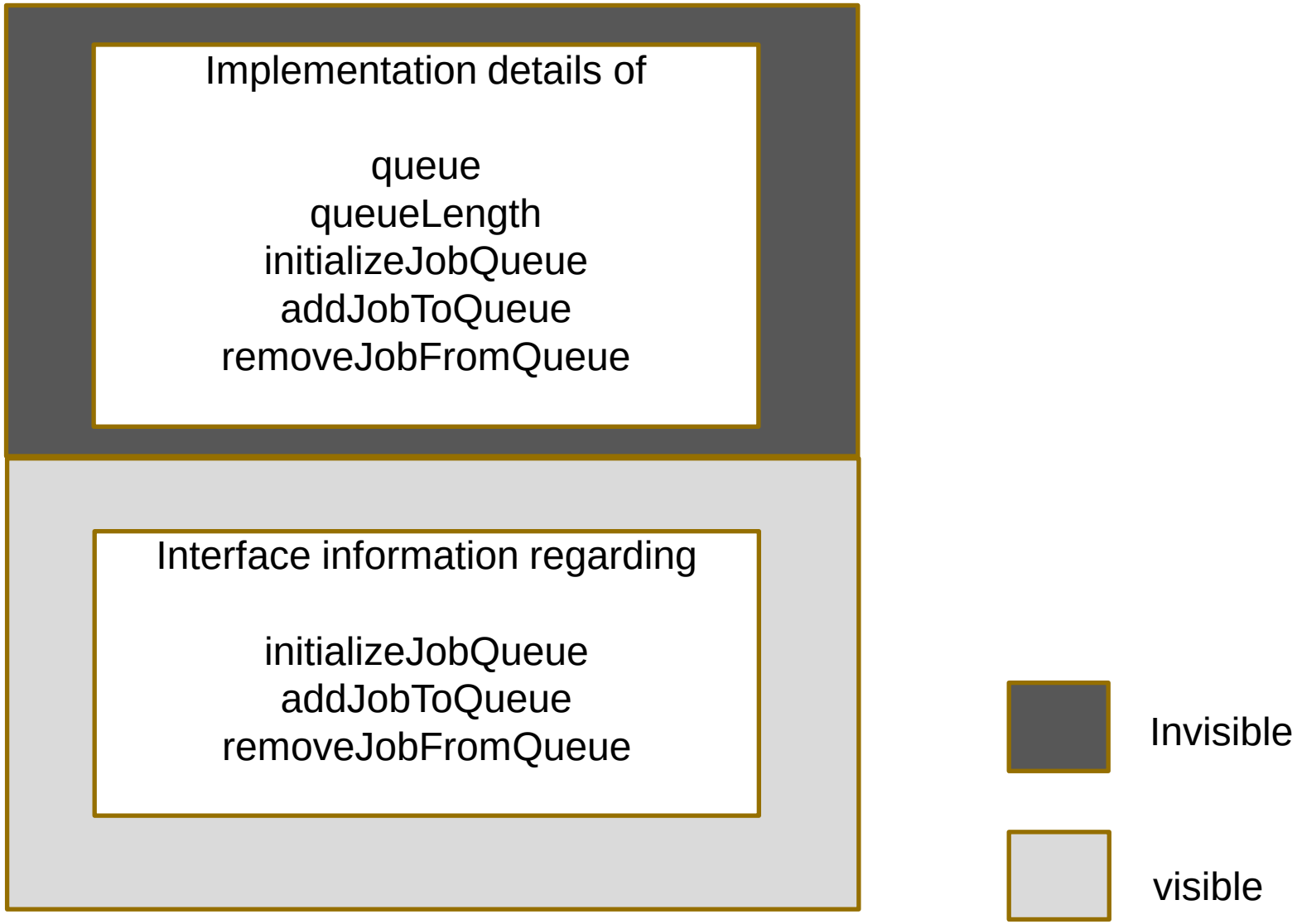
代码示例

```
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 queque 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



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：对一个软件结构内不同模块之间互连程度的度量
  - 紧密耦合的（tightly coupled）
  - 松散耦合的（loosely coupled）
- 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

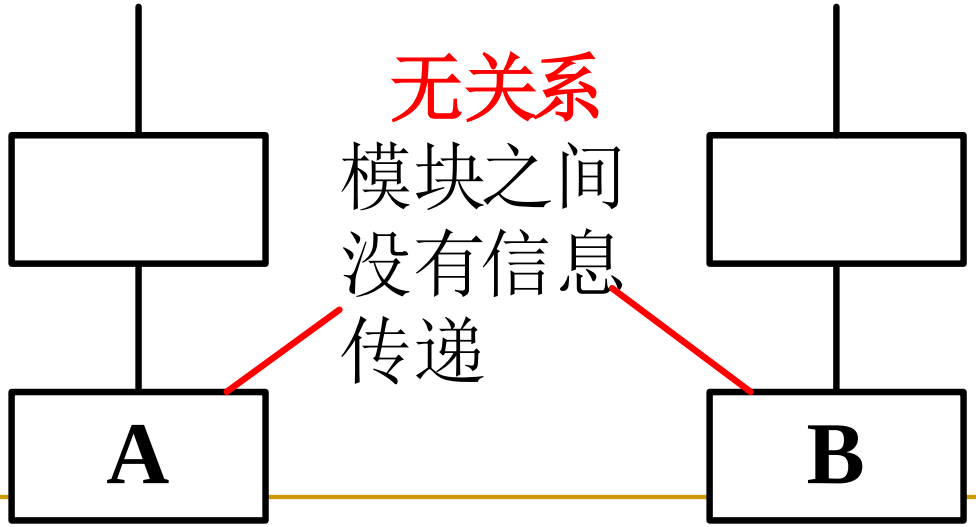
- 非直接耦合（Good）
- 数据耦合
- 标记耦合
- 控制耦合
- 外部耦合
- 公共耦合
- 内容耦合（Bad）

非直接耦合

☺ The most desirable.

(1) 非直接耦合/完全独立(no direct coupling)

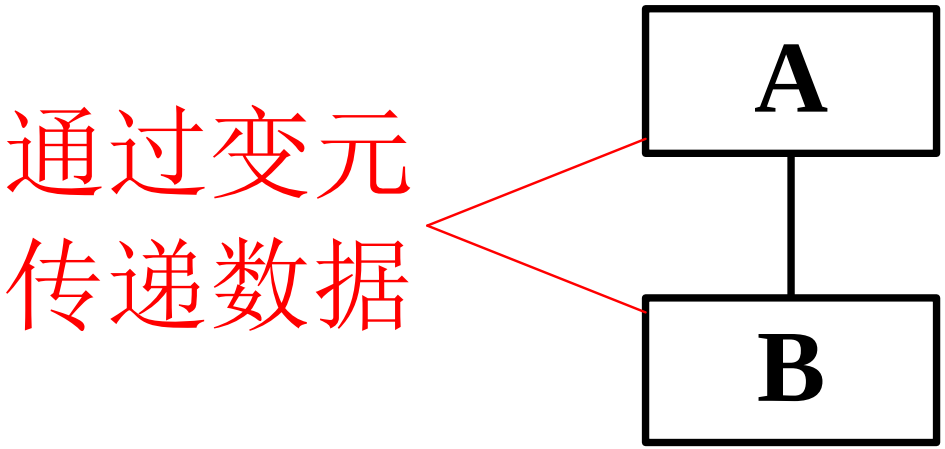
- 如果两个模块中的每一个都能独立地工作而不需要另一个模块的存在，那么它们完全独立
- 在一个软件系统中不可能所有模块之间都没有任何连接
- **没有必要使模块完全独立，只要尽可能减少模块之间的耦合度**



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。

标记耦合

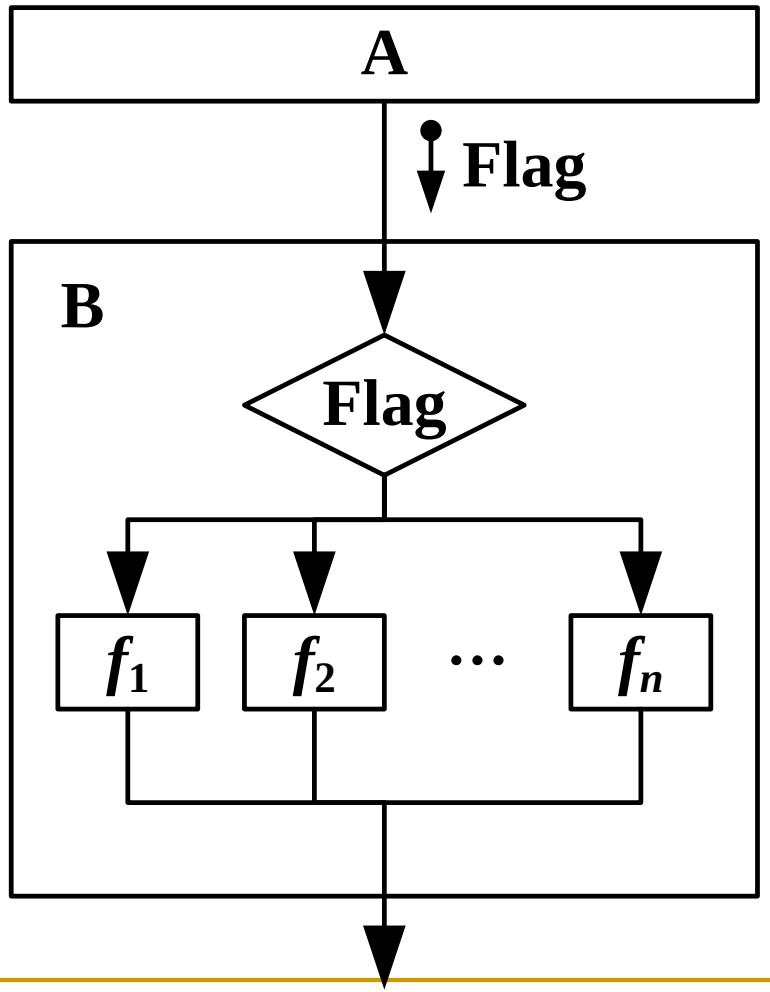
(3) 标记耦合(stamp coupling)

- 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
- 体现了模块之间更加复杂的接口
- 模块之间的数据的格式和组织方式必须匹配
- 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

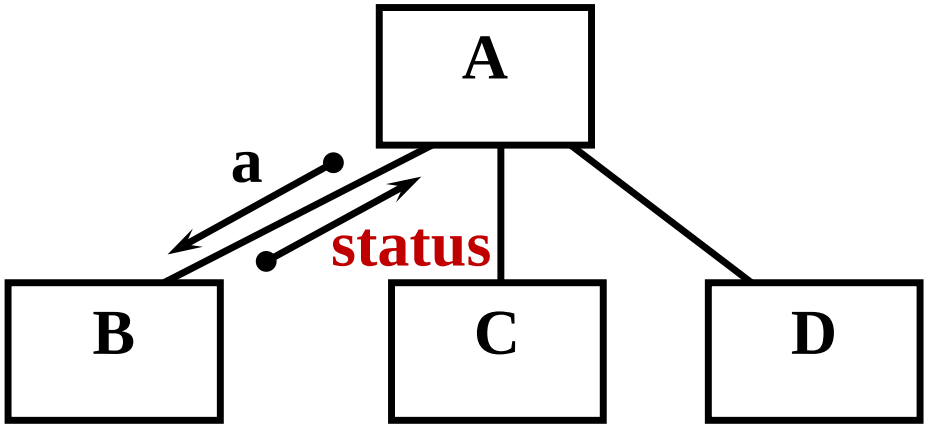
控制耦合

(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

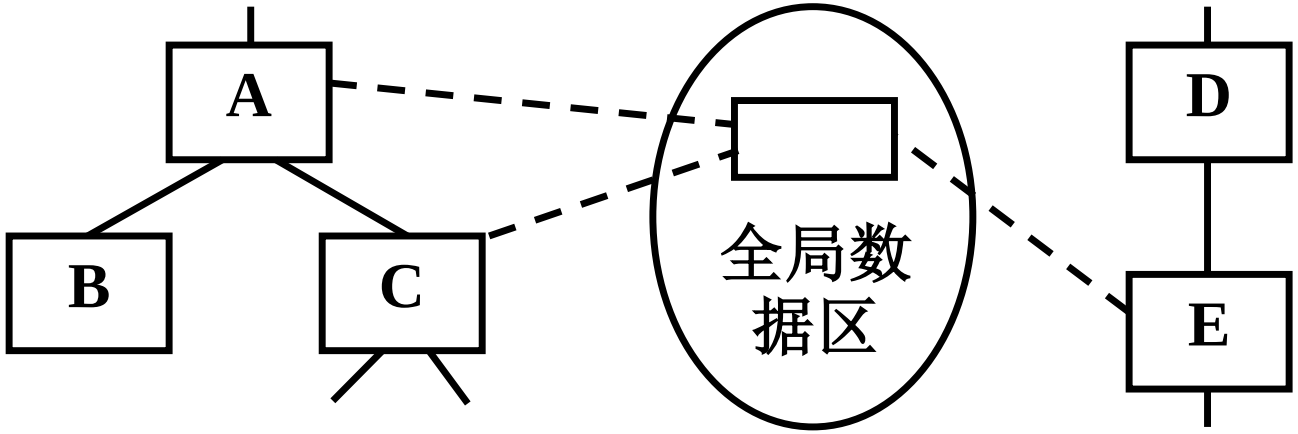
(5) 外部耦合(external coupling)

- 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)

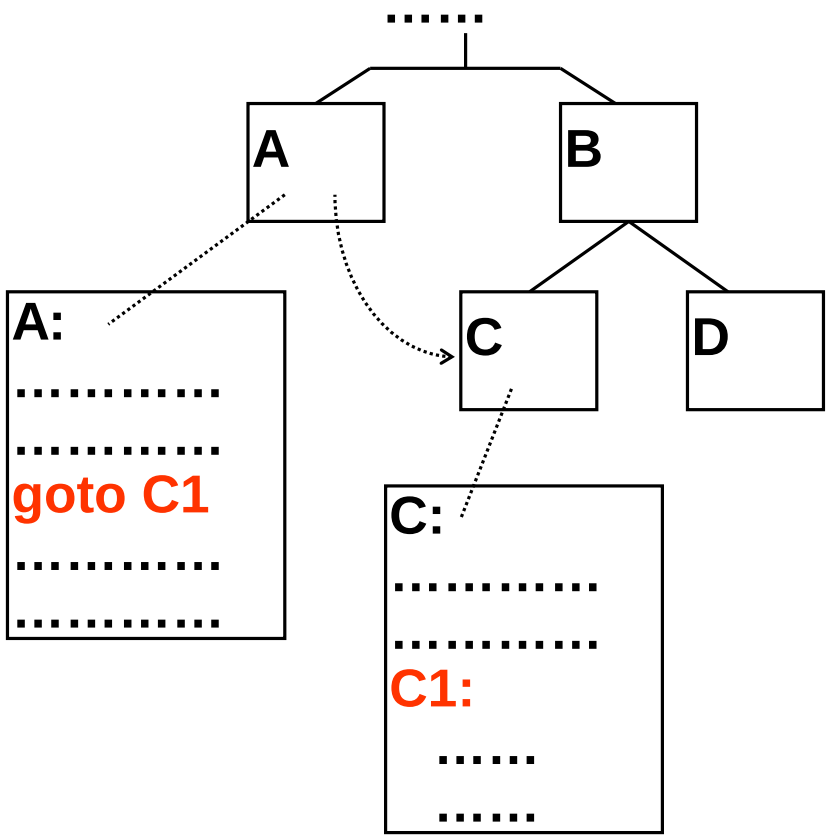
- 两个或多个模块通过一个公共数据环境相互作用
  - 访问全局变量
  - 读写同一个数据库的同一个记录



内容耦合

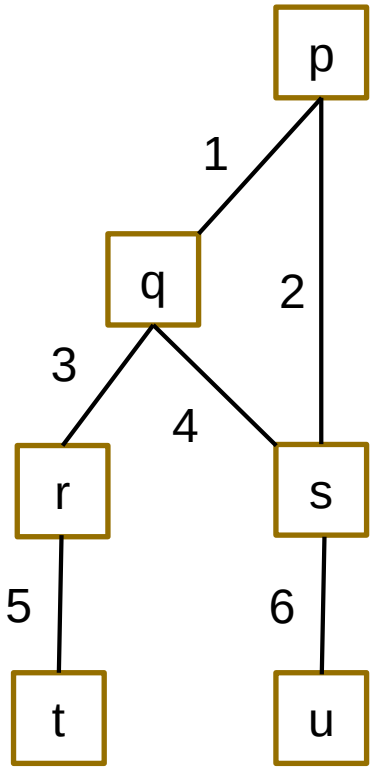
(7) 内容耦合(content coupling)

- 一个模块内的分支转移到另外一个模块中



☹ **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

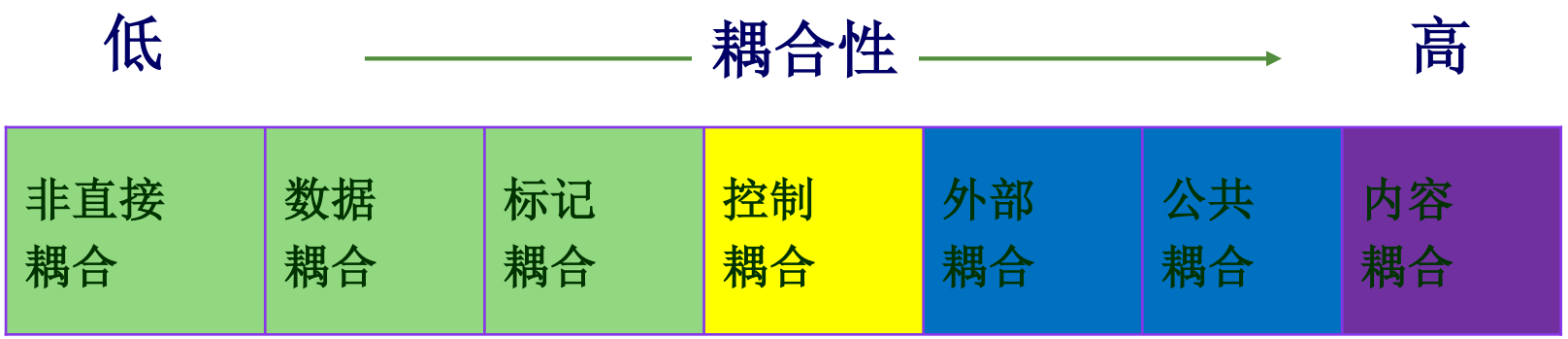
Number	In	Out
1	aircraft_type	Status_flag
2	list_of_aircraft_parts	-----
3	function_code	-----
4	list_of_aircraft_parts	-----
5	part_number	part_manufacturer
6	part_number	part_name

模块间耦合关系表

	q	r	s	t	u
p	数据	----	数据或标记	公共	公共
q		控制	数据或标记	----	----
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，

  少用控制耦合和标记耦合，

  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。

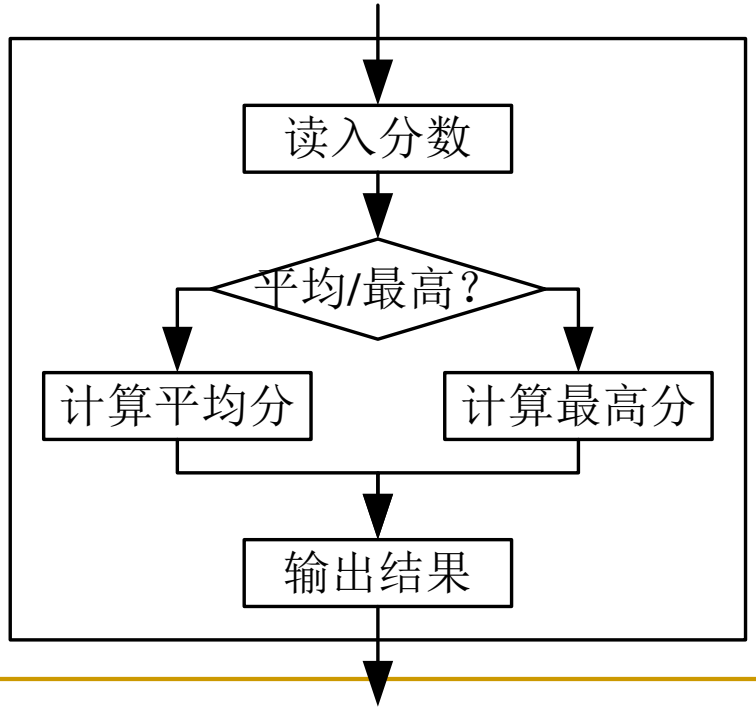
解决方案：

- 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。



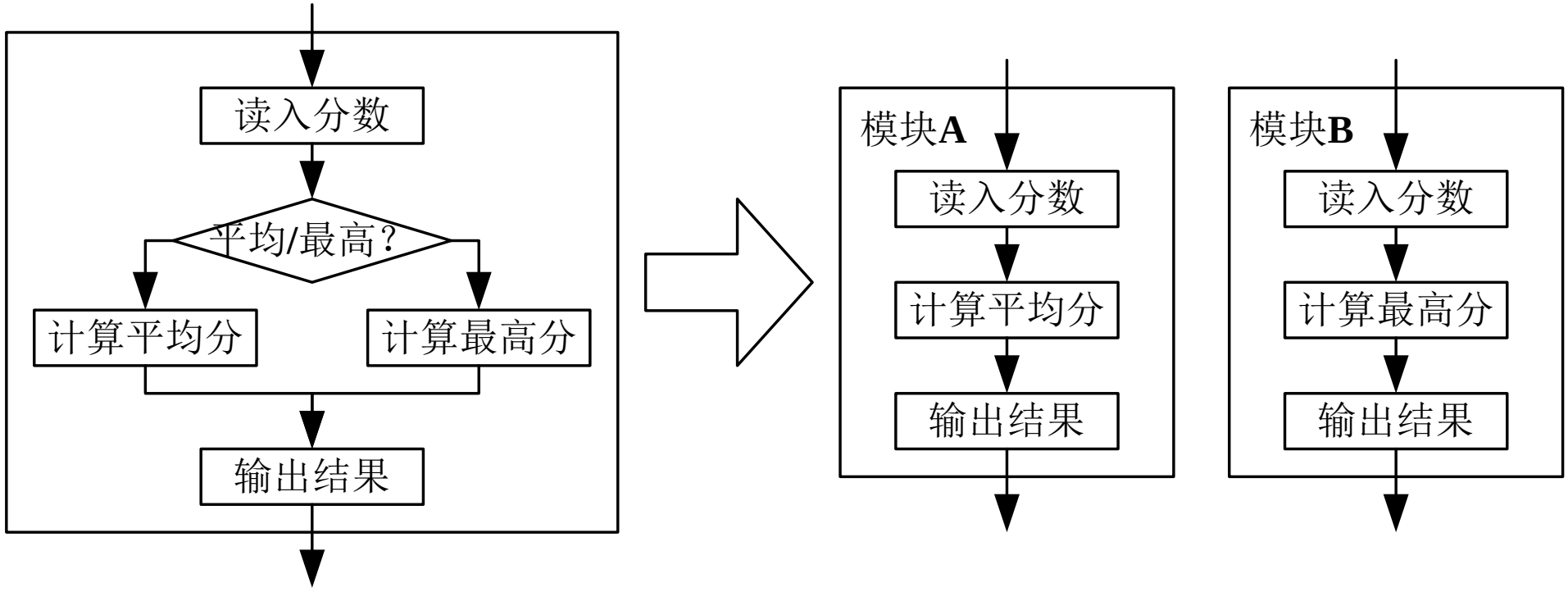
逻辑内聚

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。


读入分数
平均/最高?
计算平均分
计算最高分
输出结果
模块A
读入分数
计算平均分
输出结果
模块B
读入分数
计算最高分
输出结果

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

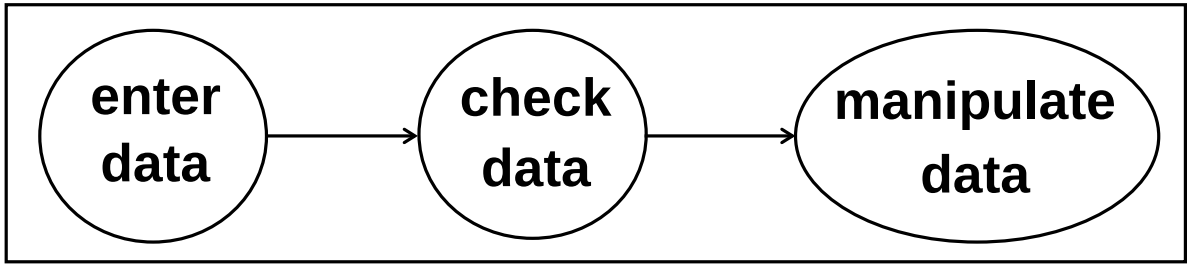
评价：

- 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。

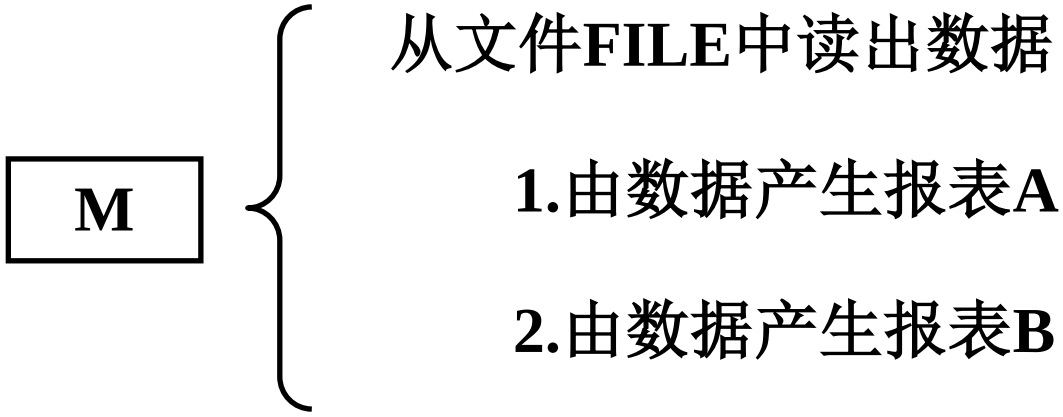
评价：

- 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：

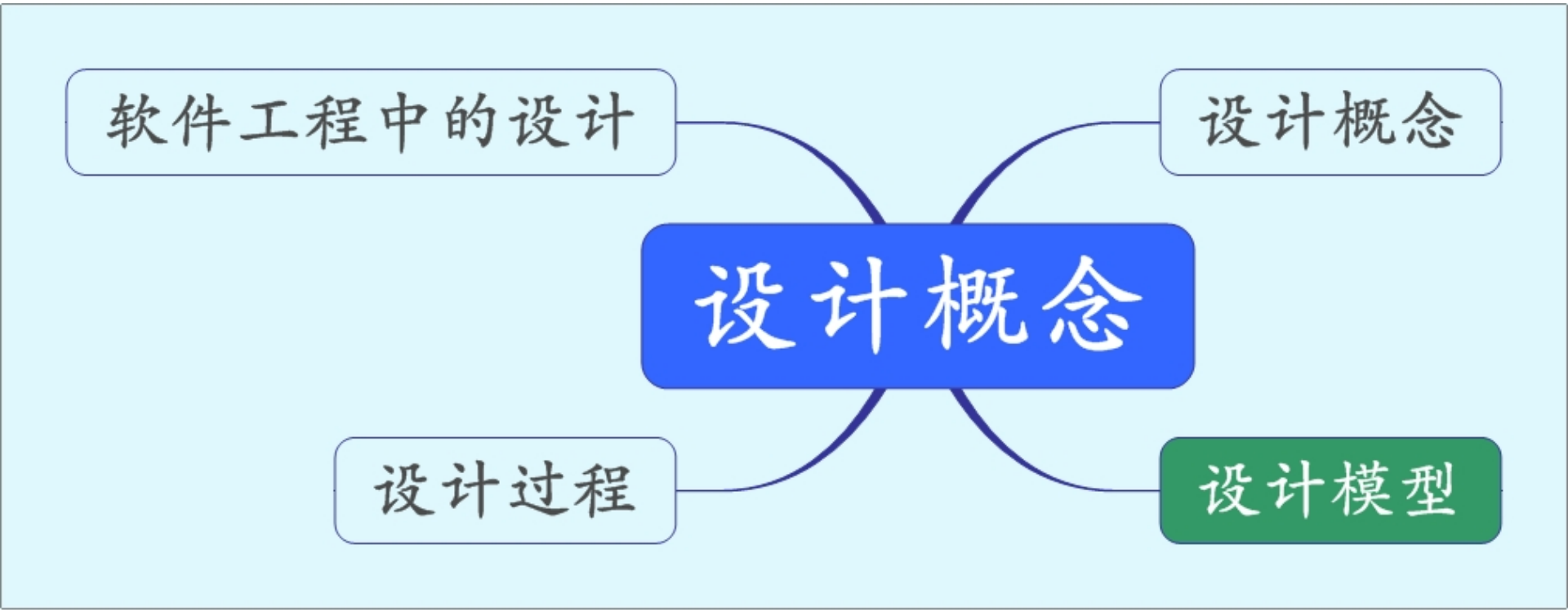
- 高内聚：功能内聚　　10分
  顺序内聚　　9分
- 中内聚：通信内聚　　7分
  过程内聚　　5分
- 低内聚：时间内聚　　3分
  逻辑内聚　　1分
  偶然内聚　　0分
- 设计时力争做到高内聚，并且能够辨认出低内聚的模块。

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。

outline



OOD概述

- 从OOA到OOD是一个逐渐扩充模型的过程
  - 对分析模型进行精化
  - 增加实现相关的细节
  - 考虑体系结构、构件和接口
- 在实际的软件开发过程中分析和设计的界限是模糊的。分析和设计活动是一个多次反复迭代的过程。

OOD准则1：弱耦合

- 对象之间的两类耦合：
  - 交互耦合：消息连接
    - 使交互耦合尽可能松散的准则：减少消息中包含的参数个数，降低参数的复杂程度，减少消息数。
  - 继承耦合：互为基类和派生类
    - 与交互耦合相反，应该提高继承耦合程度。
    - 继承是一般化类与特殊类之间耦合的一种形式。通过继承关系结合起来的基类和派生类，构成了系统中粒度更大的模块。彼此之间应该越紧密越好。

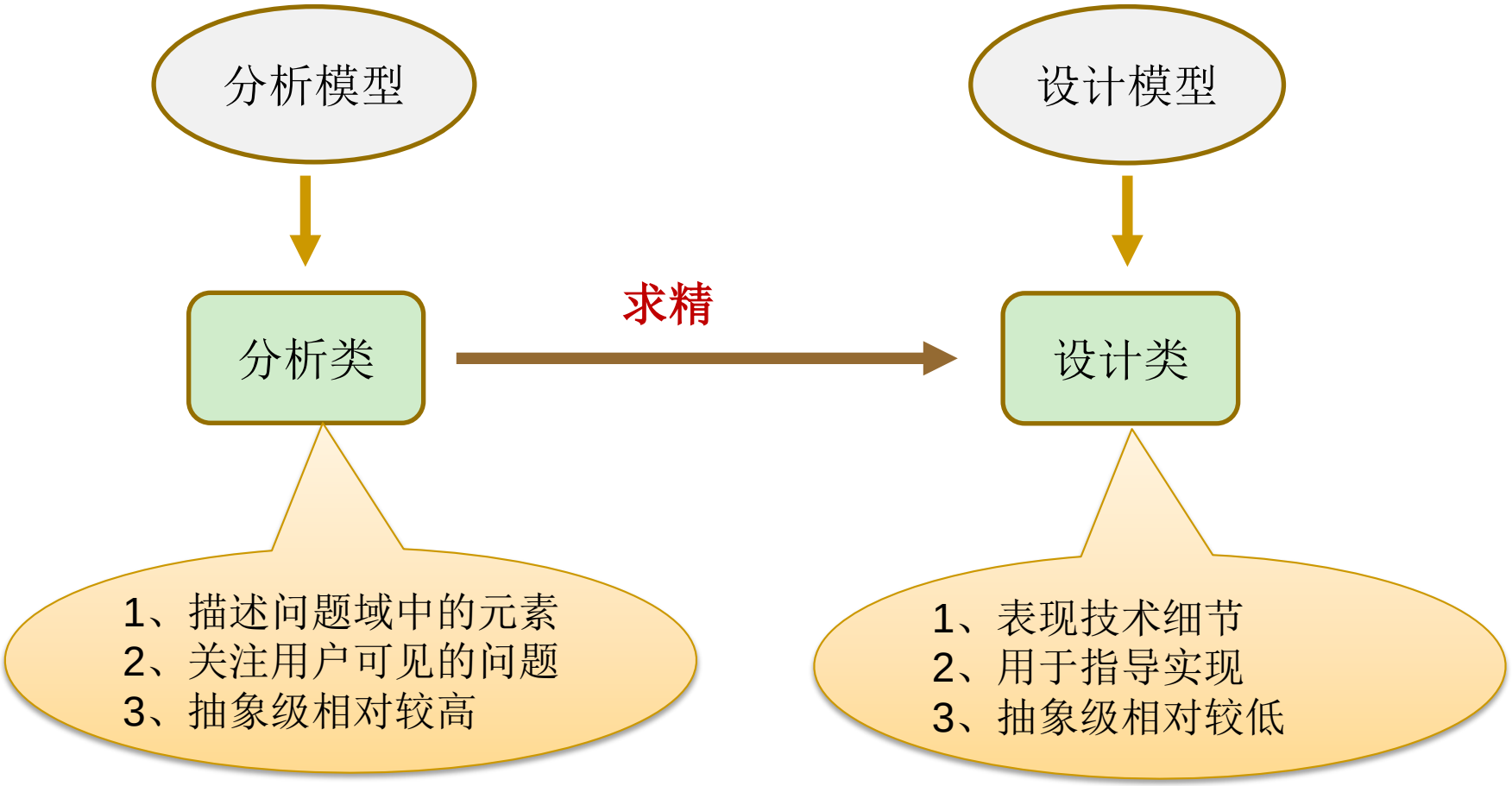
OOD准则2：强内聚

- 在面向对象设计中存在下述3种内聚：
  - 服务内聚。一个服务应该完成一个且仅完成一个功能。
  - 类内聚。一个类应该只有一个用途，它的属性和服务应该是高内聚的。如果某个类有多个用途，通常应该把它分解成多个专用的类。
  - 一般-特殊内聚。设计出的一般-特殊结构，应该符合多数人的概念，是对相应领域知识的正确抽取。

OOD准则3：复用

- 大多数现代的软件都是通过复用已有的构件或系统来构造的
- 在开发软件时，应该尽可能多地利用已有的代码
- 复用基本上从设计阶段开始
- 复用有两方面的含义：
  - **with reuse**：尽量使用已有的类（包括开发环境提供的类库，及以往开发类似系统时创建的类）
  - **for reuse**：如果确实需要创建新类，则在设计这些新类的协议时，应该考虑将来的可重复使用性。

分析类到设计类

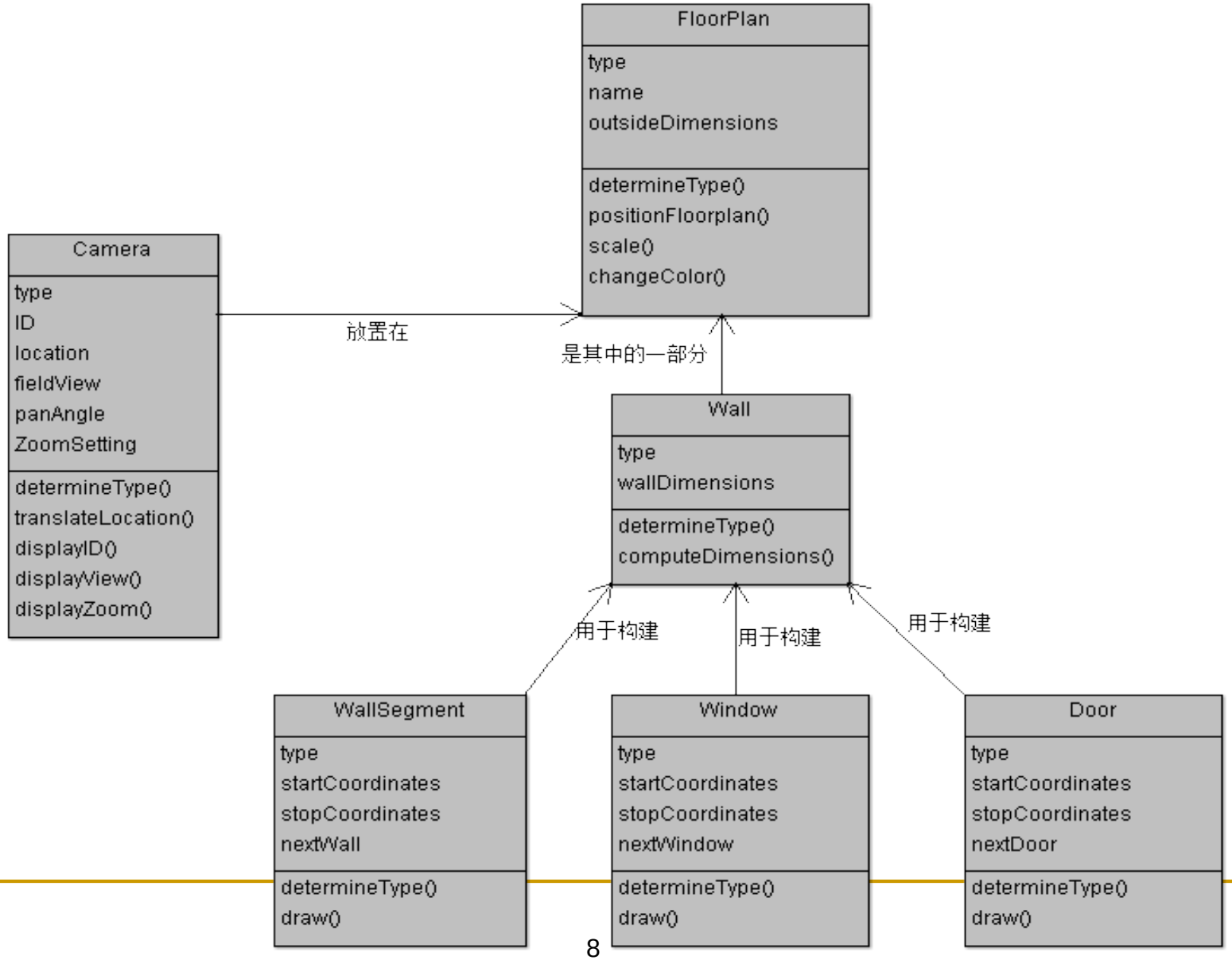

分析模型
设计模型
分析类
求精
设计类
1、描述问题域中的元素
2、关注用户可见的问题
3、抽象级相对较高
1、表现技术细节
2、用于指导实现
3、抽象级相对较低

设计类

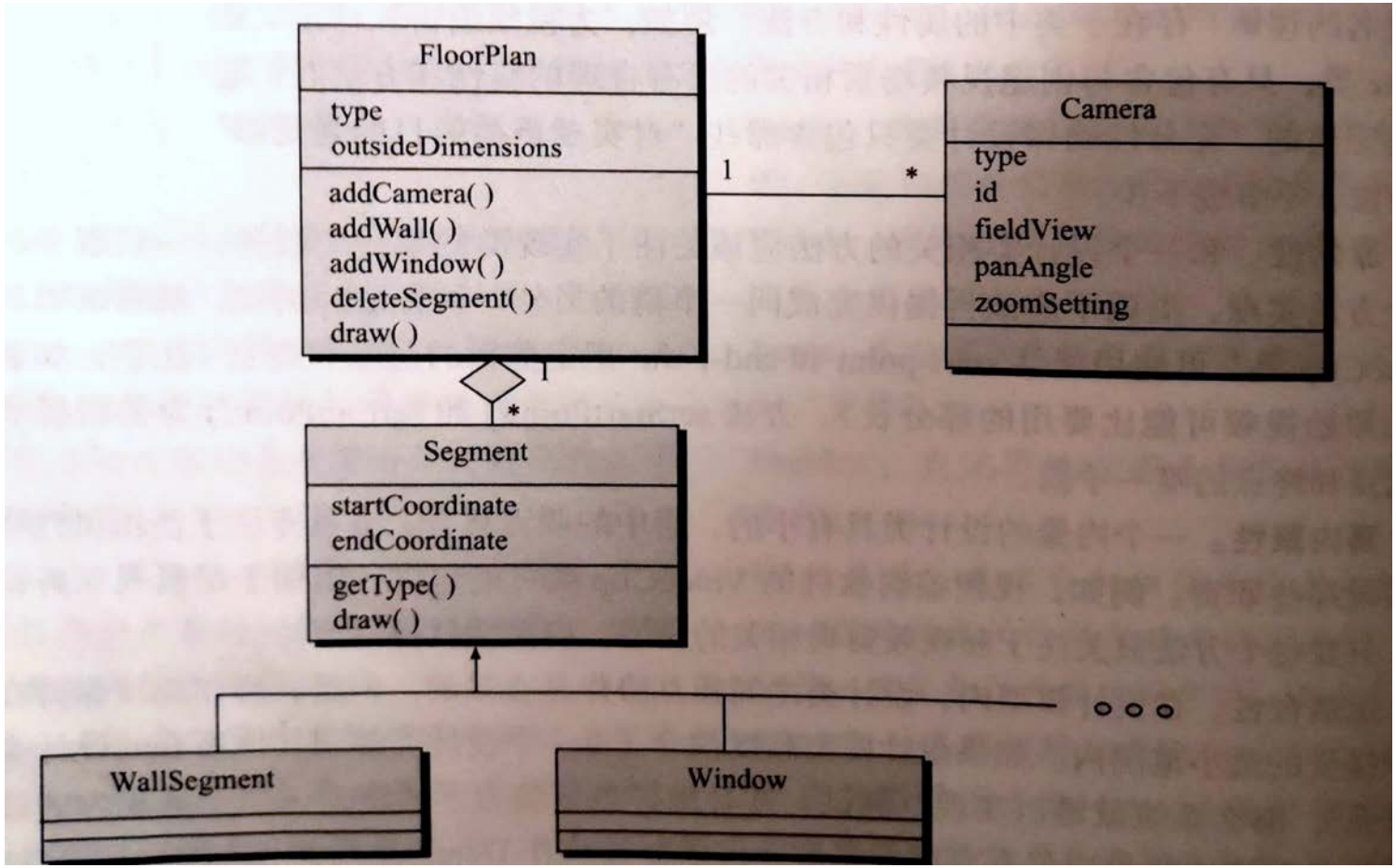
- 5种不同类型的设计类

用户接口类	业务域类	过程类	持久类	系统类
• 定义人机交互所必需的所有抽象	• 识别实现某些业务领域元素所必需的属性和服务	• 实现管理业务域类所必需的低层业务抽象	• 表示将在软件执行之外持续存在的数据存储（数据库）	• 实现软件管理和控制功能

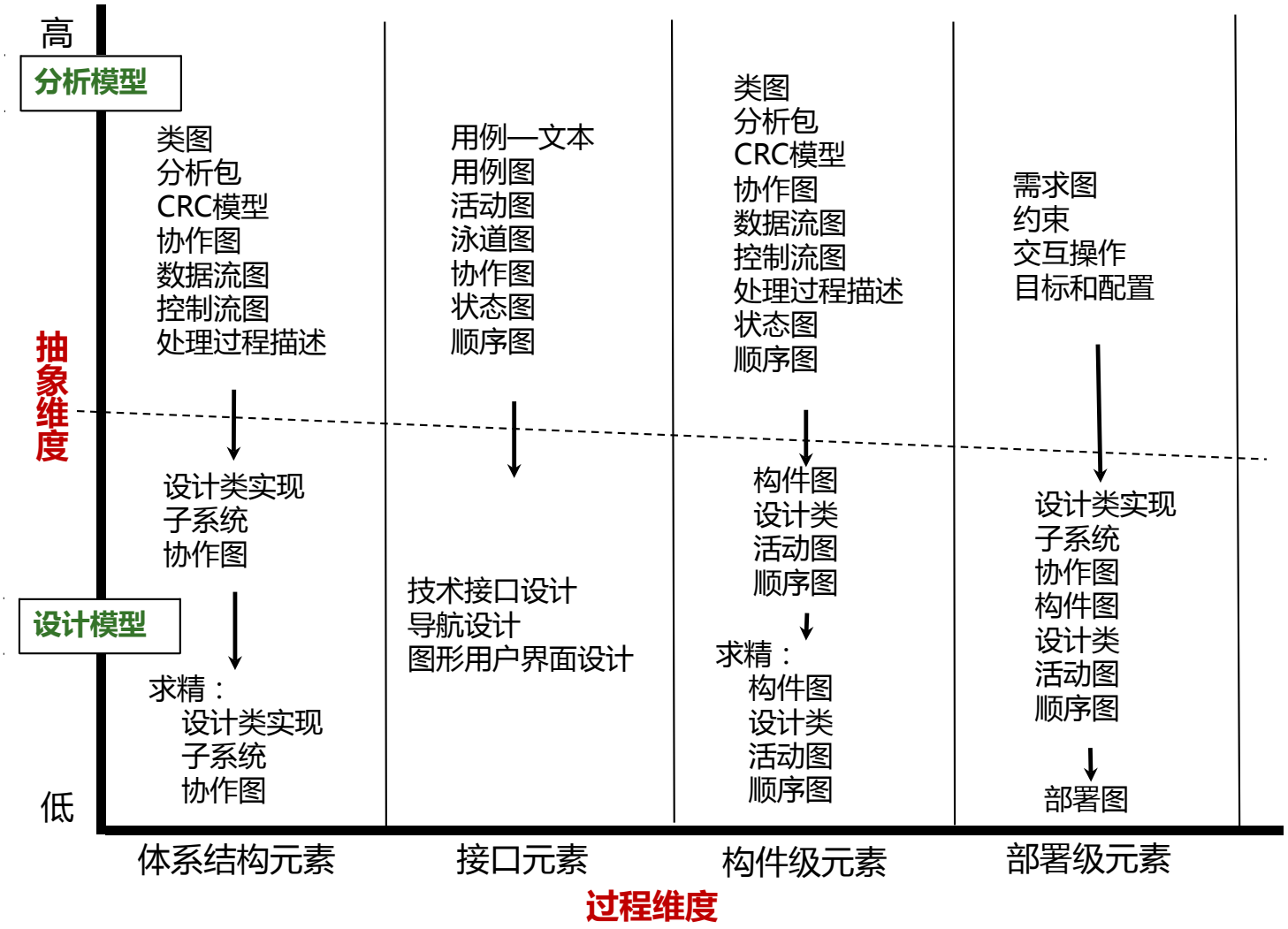
ACS-DCV平面设计图分析类



ACS-DCV平面设计图设计类



设计模型


高
分析模型
类图
分析包
CRC模型
协作图
数据流图
控制流图
处理过程描述
抽象维度
设计类实现
子系统
协作图
设计模型
求精：
设计类实现
子系统
协作图
低
用例—文本
用例图
活动图
泳道图
协作图
状态图
顺序图
技术接口设计
导航设计
图形用户界面设计
类图
分析包
CRC模型
协作图
数据流图
控制流图
处理过程描述
状态图
顺序图
构件图
设计类
活动图
顺序图
求精：
构件图
设计类
活动图
顺序图
需求图
约束
交互操作
目标和配置
设计类实现
子系统
协作图
构件图
设计类
活动图
顺序图
部署图
体系结构元素
接口元素
构件级元素
部署级元素
过程维度

设计模型元素

- 数据设计元素
- 体系结构设计元素
- 接口设计元素
- 构件级设计元素
- 部署级设计元素

数据设计元素

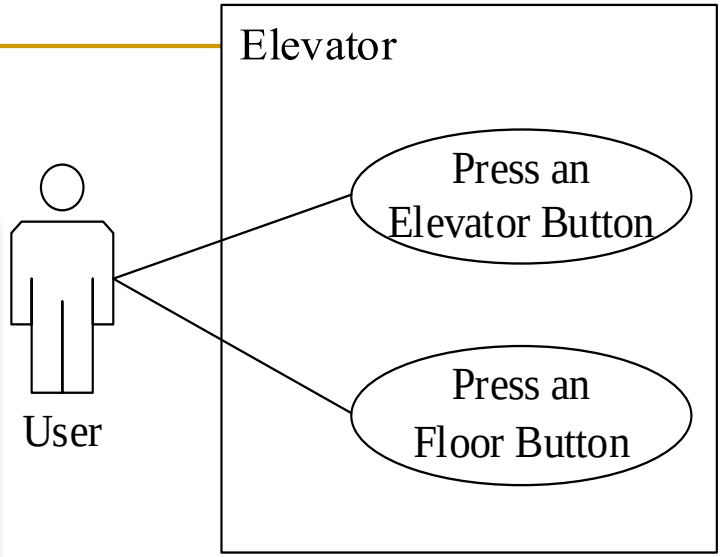
- 完成类图
  - 确定属性的格式
    - 通常由需求分析得到
    - 实际情况下，oo是迭代过程，越晚为类增加属性越好
    - 例如日期格式，美国为mm/dd/yyyy，欧洲为dd/mm/yyyy，中国为yyyy/mm/dd，长度为10个字符
  - 确定类的方法
    - 基于交互模型确定所有的方法
    - 确定方法属于哪个类
      - 信息隐藏：类的属性是私有的，属性上的操作在该类中
      - 职责驱动设计：如果一个客户端向一个对象发消息，则该对象就要为执行客户端的请求负责。

数据设计元素

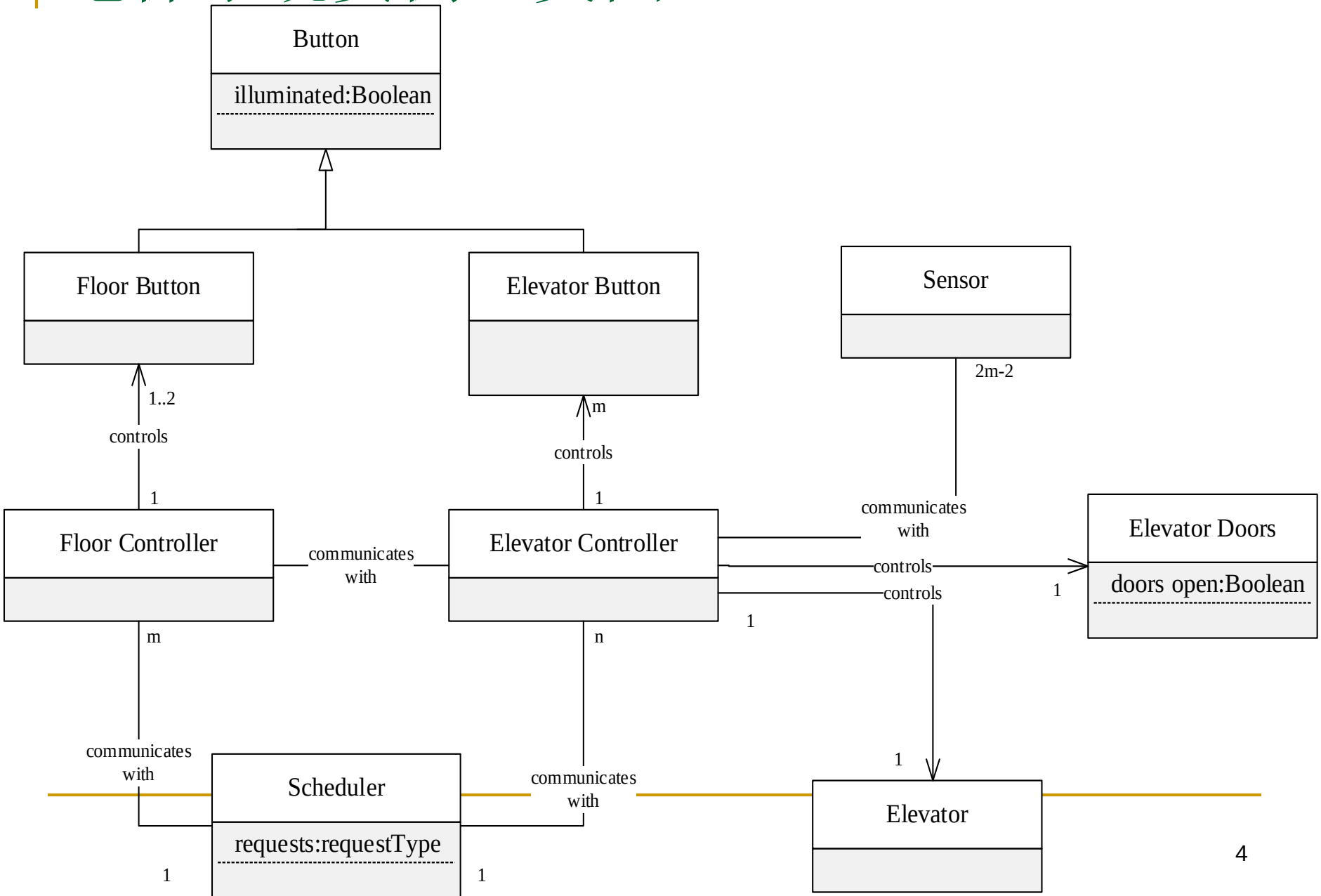
- **完成类的详细设计**
- **数据设计**
  - 在体系结构层上，设计数据库
  - 在构件层上，设计数据结构及其算法

电梯系统实例

1、用户A在3楼按了向上的楼层按钮，他想去7楼。
2、向上的楼层按钮灯亮了。
3、一部电梯到达3楼，用户B在电梯里，他从1楼进电梯，想去9楼。
4、电梯门开了。
5、计时器开始计时。
用户A进入电梯
6、用户A按下7楼的电梯按钮。
7、7楼的电梯按钮灯亮了。
8、计时时间到，电梯门关上了。
9、向上的楼层按钮灯灭了。
10、电梯上升至7楼。
11、 7楼的电梯按钮灯灭了。
12、电梯门开了。
13、计时器开始计时。
用户A走出电梯
14、计时时间到，电梯门关上了。
15、电梯载着用户B继续向9楼行进。



电梯系统实例（类图）


Button
illuminated:Boolean
Floor Button
Elevator Button
Sensor
1..2
controls
1
m
controls
1
2m-2
Floor Controller
communicates
with
Elevator Controller
communicates
with
controls
controls
Elevator Doors
doors open:Boolean
1
1
m
n
communicates
with
Scheduler
requests:requestType
communicates
with
1
Elevator
1
1

电梯系统实例（分配职责）

Class

Elevator Controller

Responsibility

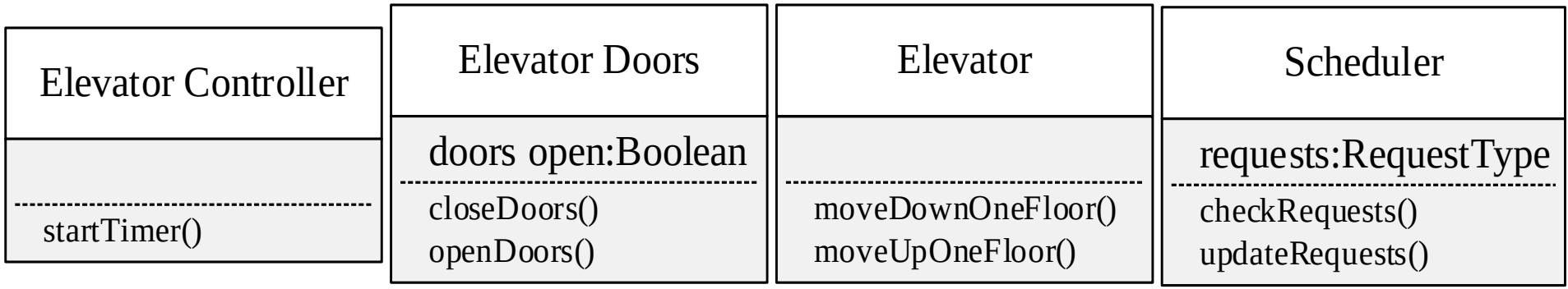
1. Send message to **Elevator Button** to check if it is turned on
2. Send message to **Elevator Button** to turn itself on
3. Send message to **Elevator Button** to turn itself off
4. Send message to **Elevator Doors** to open themselves
5. Start timer
6. Send message to **Elevator Doors** to close themselves after time out
7. Send message to **Elevator** to move itself up one floor
8. Send message to **Elevator** to move itself down one floor
9. Send message to **Scheduler** that a request has been made
10. Send message to **Scheduler** that a request has been satisfied
11. Send message to **Scheduler** to check if the elevator is to stop at the next floor
12. Send message to **Floor Controller** that elevator has left floor

电梯系统实例（增加方法）

- 职责驱动设计
  - 事件5表示**Elevator Controller**有启动计时器的职责，因此，**Elevator Controller**具有该方法
  - 其余11个事件
    - “Send a message to another class to tell it to do something.”
    - 为相应的类增加方法
- 所有12个事件都遵循信息隐藏原则

电梯系统实例（增加方法）

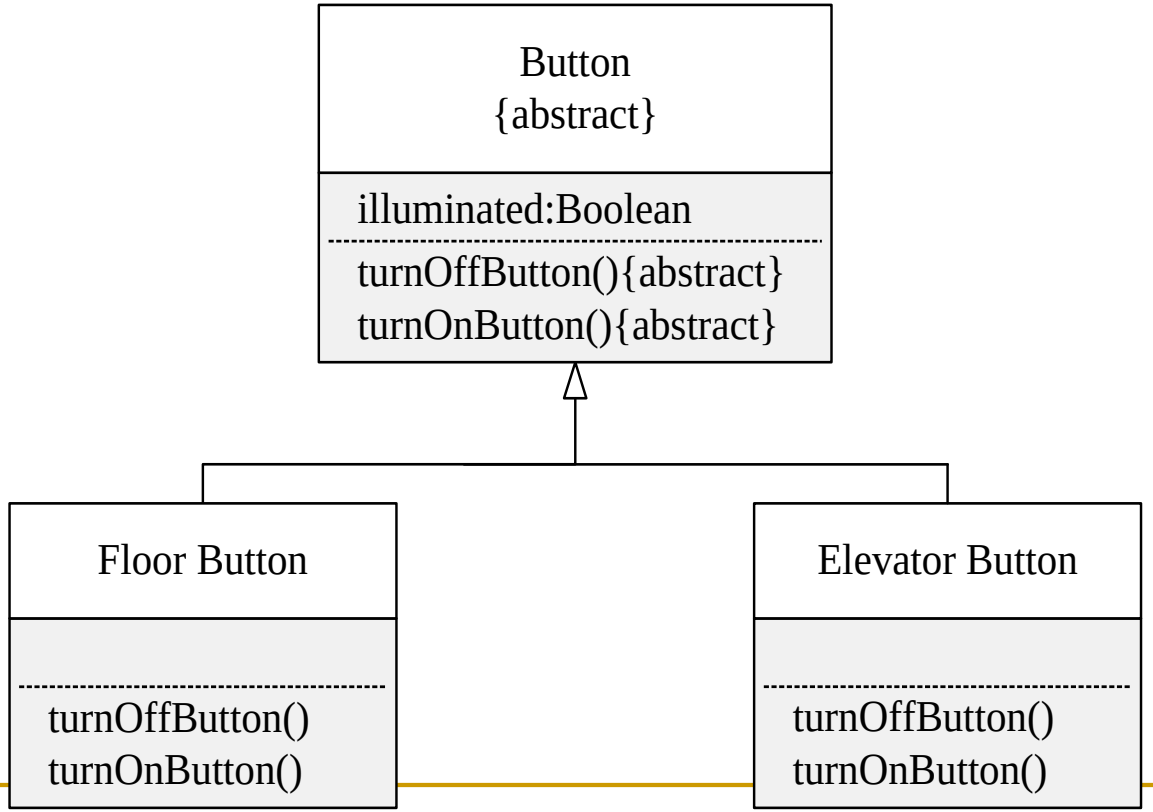
1. **Elevator Controller** starts timer.
2. **Elevator Controller** sends message to **Elevator Doors** to close or open themselves.
3. **Elevator Controller** sends message to **Elevator** to move itself down or up one floor.
4. **Elevator Controller** sends message to **Scheduler** to check if the elevator is to stop at the next floor.



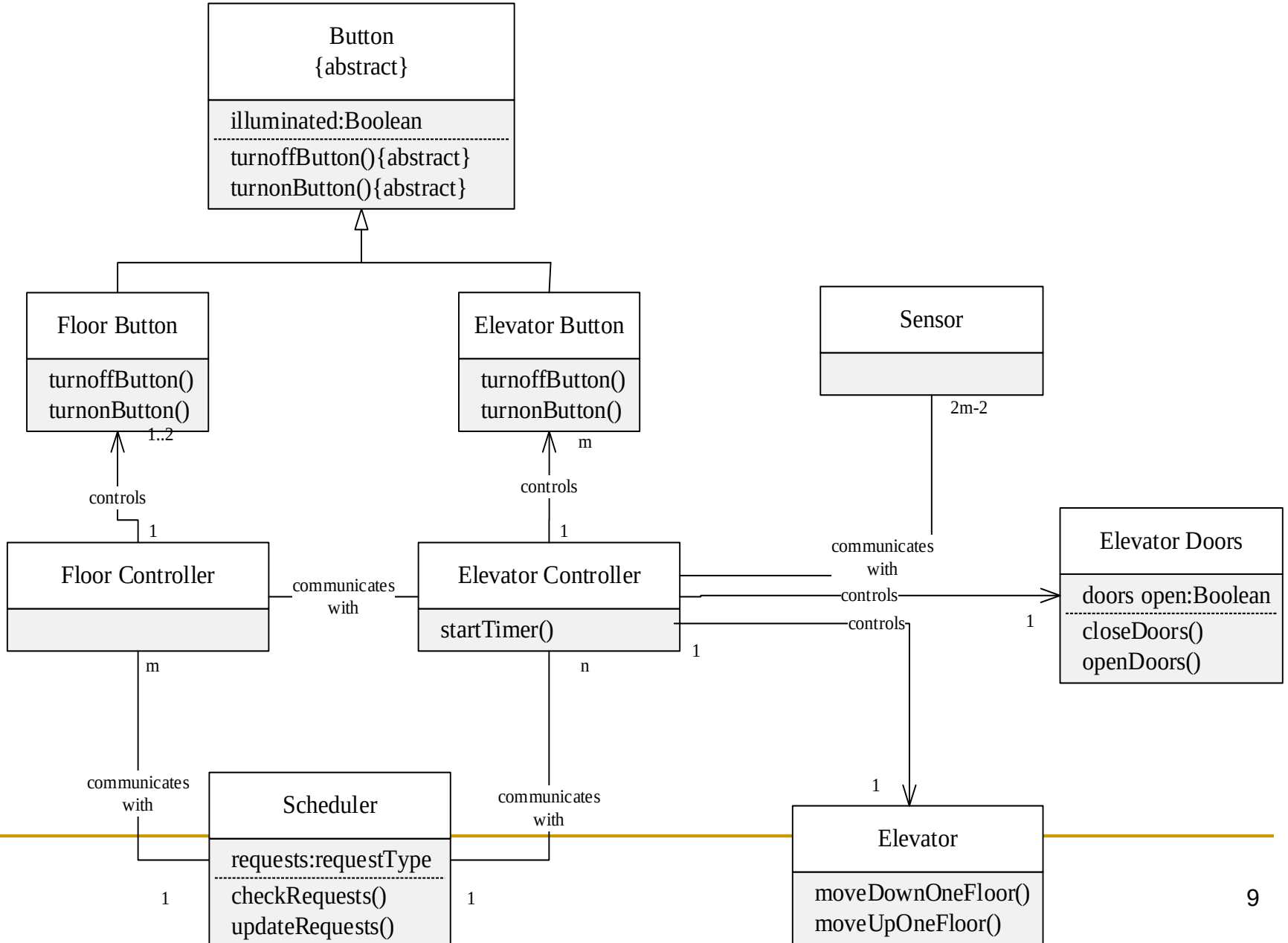
电梯系统实例（增加方法）

1. **Elevator Controller** sends message to **Elevator Button** to turn itself on or off.

2. **Floor Controller** sends message to **Floor Button** to turn itself on or off.



电梯系统实例


Button
{abstract}
illuminated:Boolean
turnoffButton(){abstract}
turnonButton(){abstract}
Floor Button
turnoffButton()
turnonButton()
Elevator Button
turnoffButton()
turnonButton()
Sensor
2m-2
1..2
m
controls
1
controls
1
Floor Controller
communicates
with
Elevator Controller
startTimer()
communicates
with
controls
controls
Elevator Doors
doors open:Boolean
closeDoors()
openDoors()
1
1
m
n
communicates
with
communicates
with
1
Scheduler
requests:requestType
checkRequests()
updateRequests()
1
1
Elevator
moveDownOneFloor()
moveUpOneFloor()

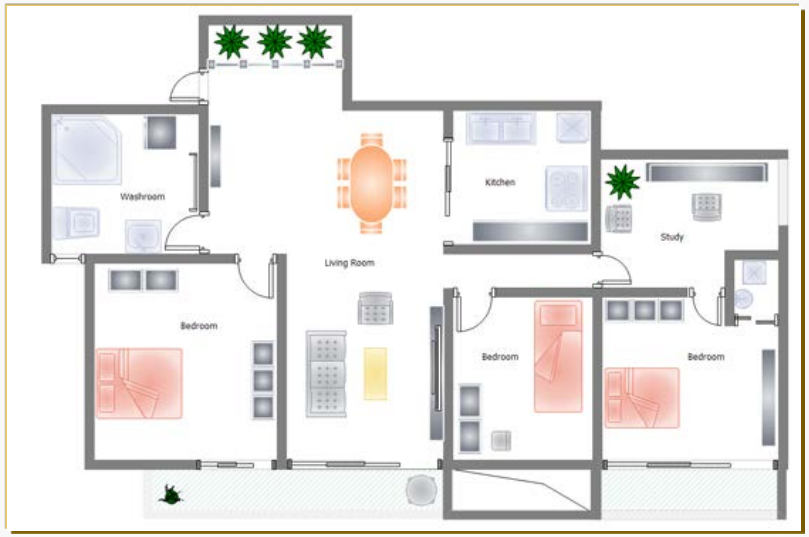
体系结构设计元素

- 类似于房屋的建筑平面图
  - 描绘房间的总体布局：大小、形状、位置关系、门、窗
  - 软件体系结构给出软件的全貌图
- 体系结构模型
  - 源于软件应用领域的信息
  - 源于需求模型：用例、类及其之间的关系
  - 源于已有的体系结构风格和模式
- 体系结构设计元素通常被描述为一组相互联系的子系统，且常常从需求模型中派生出来

体系结构设计元素

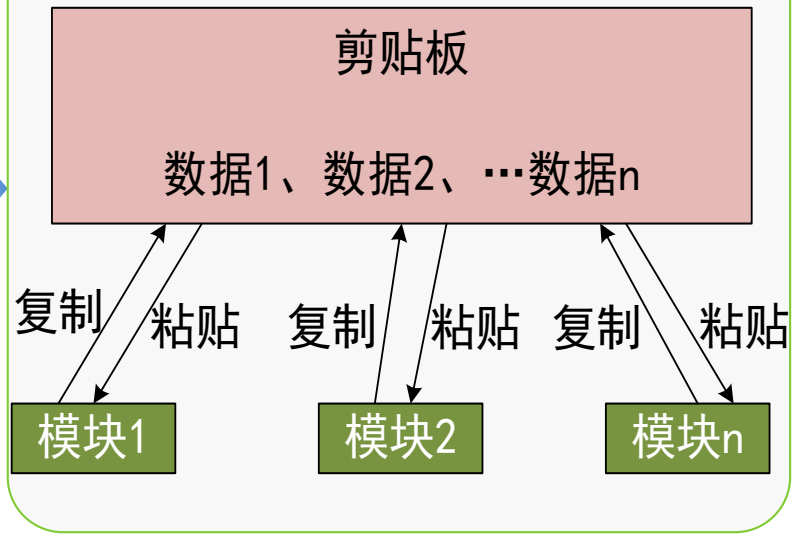
建筑设计

房屋的建筑平面图



软件设计

软件体系结构



从宏观的角度刻画组成目标软件系统的模块以及它们之间的关系

接口设计元素

- 类似于房屋的门窗及外部设施的详细绘图（说明）
  - 描绘人或物是如何进出房屋的，如何在房屋内走动
  - 软件接口描绘信息如何流入和流出系统，以及构件间是如何通信的
- 接口设计包括三个方面
  - 用户界面（UI）
    - 审美元素：布局、颜色、图表和交互机制
    - 人体工程学元素：信息布局、隐喻（metaphor）、UI导航
    - 技术元素：UI模式、可复用构件
  - 与外部系统的接口
    - 在需求工程阶段确定
  - 内部不同构件间的接口
    - 在构件设计阶段确定

接口设计元素

建筑设计

房屋门窗的设计图

- 人和物如何进出房屋

软件设计

接口设计

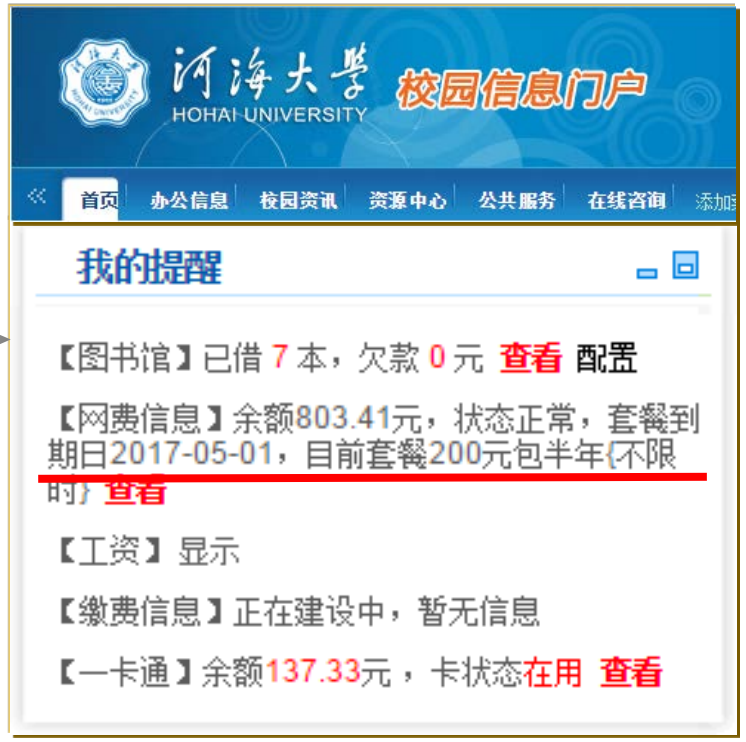
- 信息如何输入输出软件

用户界面设计

- 界面元素（菜单、按钮、图片等）的组织与布局
- 界面元素的预期响应行为
- 界面流转关系

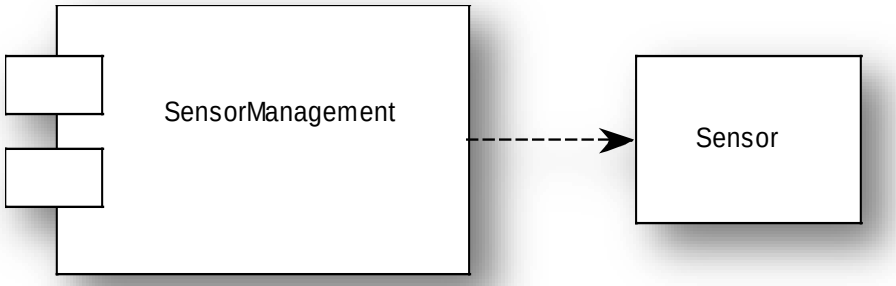
与外部系统的接口设计


DR.COM®
用户自助服务系统
帐户信息
账号：rzqi(戚荣志)
套餐：半年200元
余额：803.41元 (快速充值)
状态： 正常 在线
防伪信息：未设置
用户编号：900008121
到期日期：2019-05-01
退 出
您现在的位置：


河海大学
HOHAI UNIVERSITY
校园信息门户
首页
办公信息
校园资讯
资源中心
公共服务
在线咨询
我的提醒
【图书馆】已借 7 本，欠款 0 元 查看 配置
【网费信息】余额803.41元，状态正常，套餐到期日2017-05-01，目前套餐200元包半年{不限时} 查看
【工资】显示
【缴费信息】正在建设中，暂无信息
【一卡通】余额137.33元，卡状态在用 查看

构件级设计元素

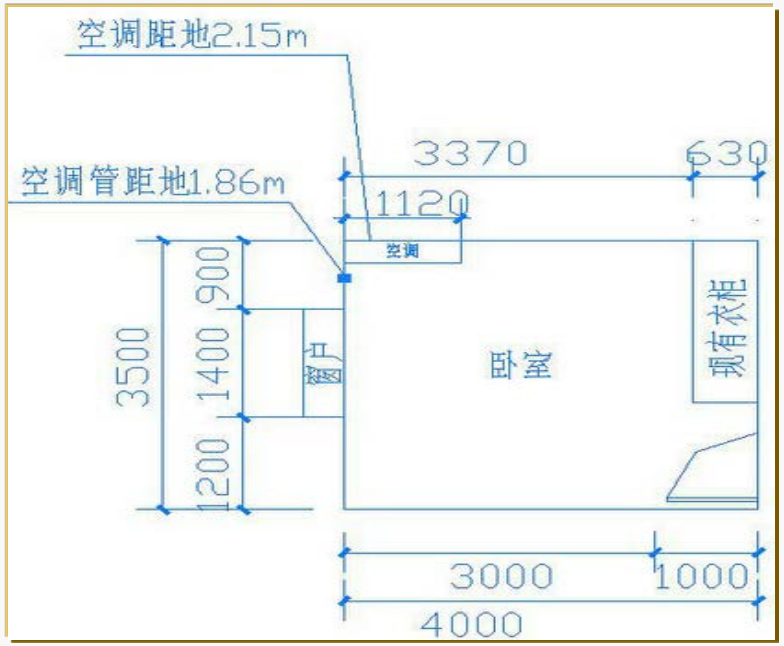
- 类似于房屋每个房间的详细绘图（说明）
  - 描绘房间的布线、管道，以及电插座、开关面板、水龙头、水槽、淋浴、浴缸、地漏、橱柜、衣帽间的位置等
  - 完整的描述每个构件的**内部细节**
- 构件设计包括三个方面
  - 数据结构
    - 伪代码
  - 算法
    - 伪代码
    - 图表工具（活动图、流程图、盒图、判定表等）
  - 访问所有服务的接口



构件级设计元素

建筑设计

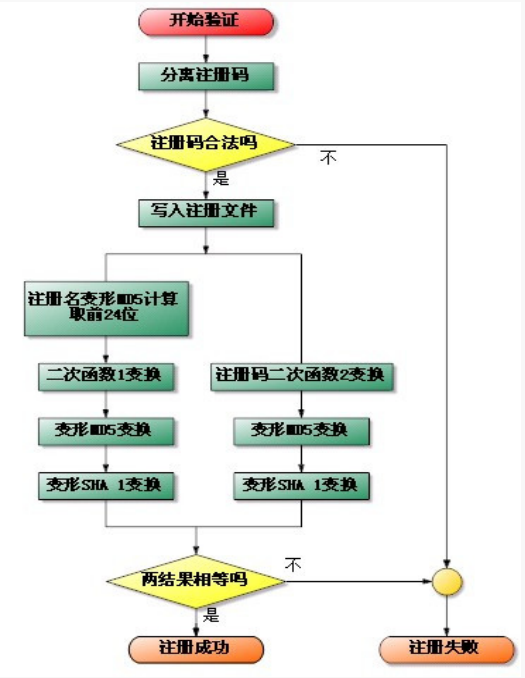
房间的详细设计图



- 管线、家具的位置等

软件设计

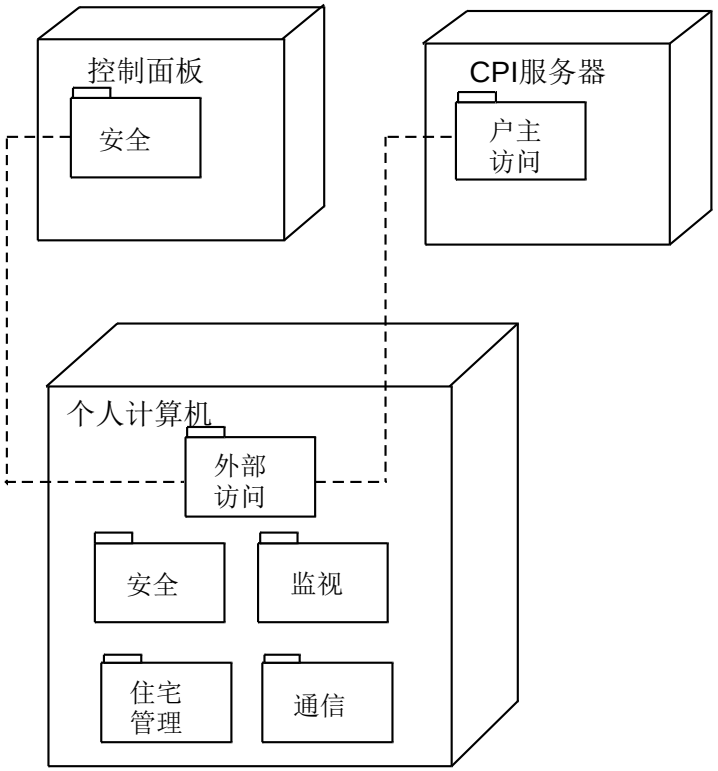
模块详细设计



- 算法
- 数据结构

部署级设计元素

- 指明软件功能和子系统将如何在支持软件的物理计算环境内分布
- UML部署图建模
- *描述符形式*的部署图显示了计算环境，但并没有明确地说明配置细节
- *实例形式*的部署图在后面阶段的设计中明确硬件配置细节



小结