

## 软件工程核心思想

- 1 软件工程的本质：不同抽象层次之间的映射与转换
- 2 软件工程所关注的目标
- 3 软件开发中的多角色
- 4 软件工程 = 最佳实践
- 5 软件工程的四个核心理论概念

## 软件工程核心思想

- 1 软件工程的本质：不同抽象层次之间的映射与转换
- 2 软件工程所关注的目标
- 3 软件开发中的多角色
- 4 软件工程 = 最佳实践
- 5 软件工程的四个核心理论概念

## 一个小例子

- 你要开发一段程序，输入班级所有人的成绩，按成绩由高到低的次序进行排序
- 你该如何去做？

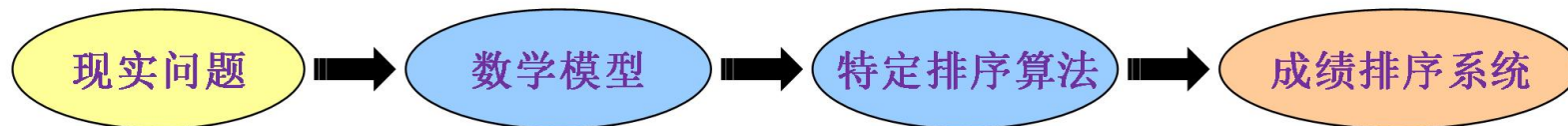
- 方法1：直接写程序



- 方法2：先设计算法，然后再用程序语言实现

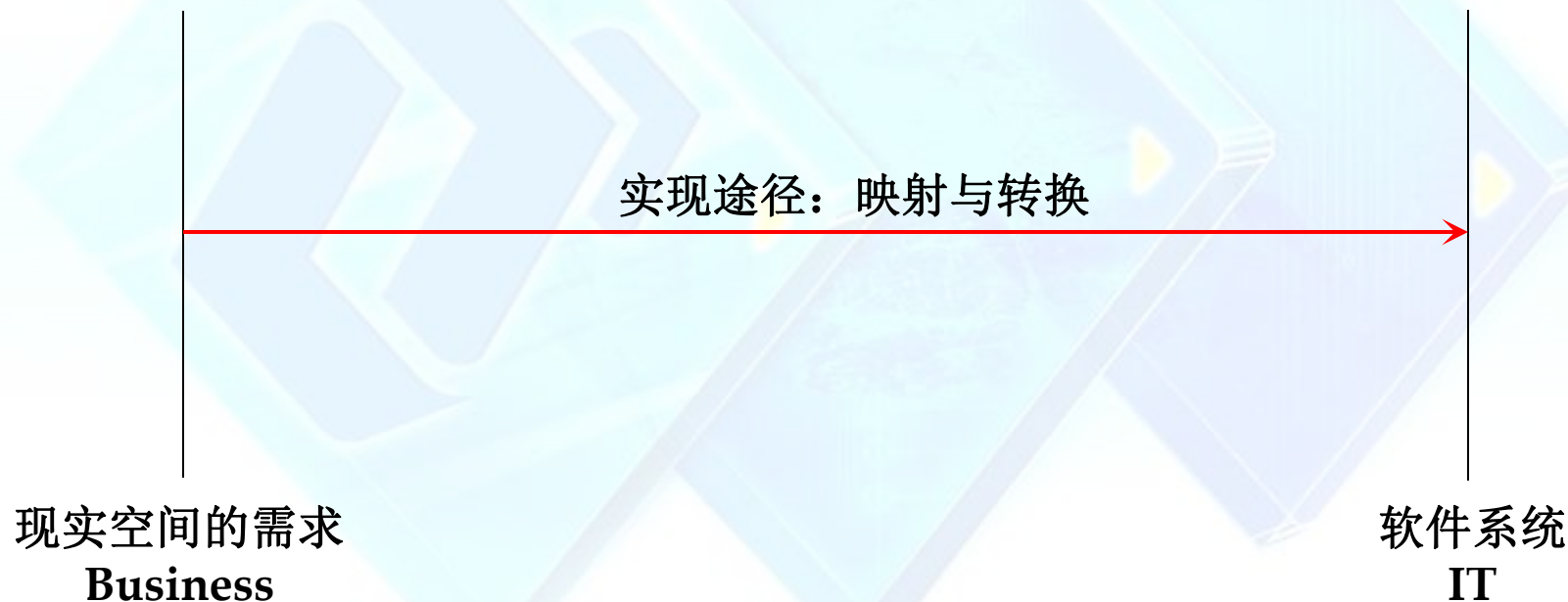


- 方法3：先建立数学模型，然后转换为算法，然后编程实现

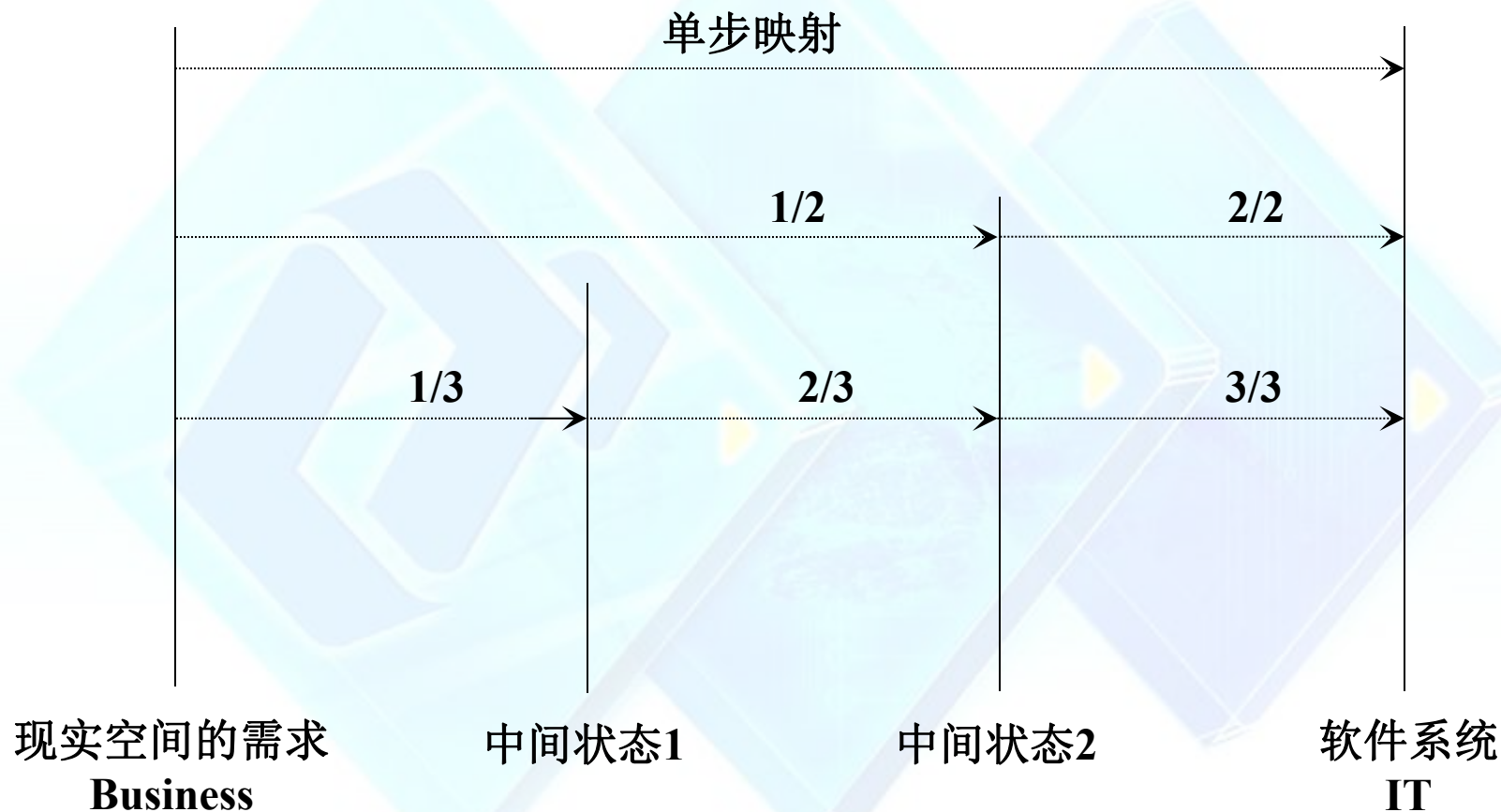


## 映射与转换

- 任何软件系统开发的共同本质在于：
  - 从现实空间的需求到计算机空间的软件代码之间的映射与转换

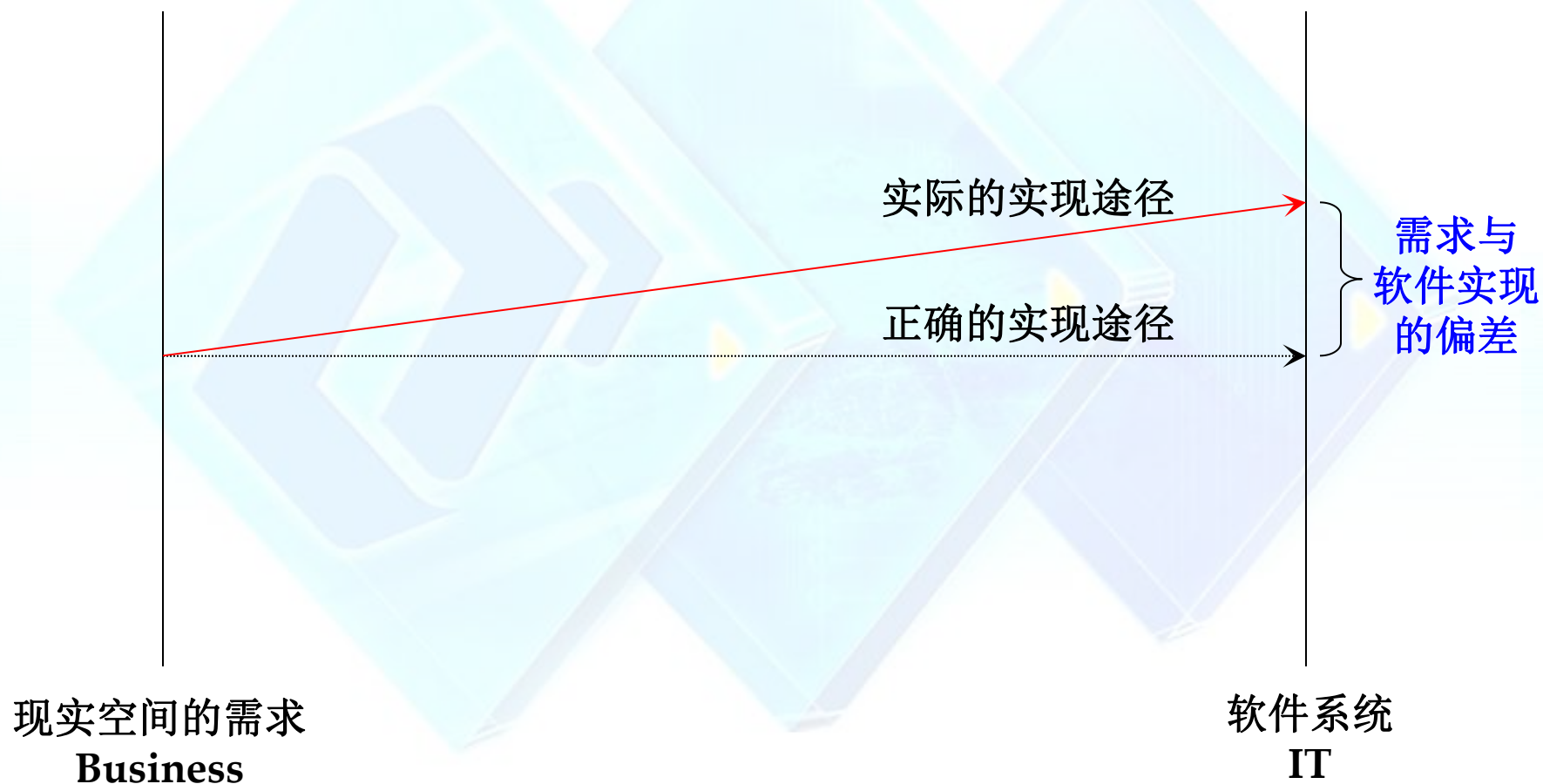


## 单步映射与多步映射



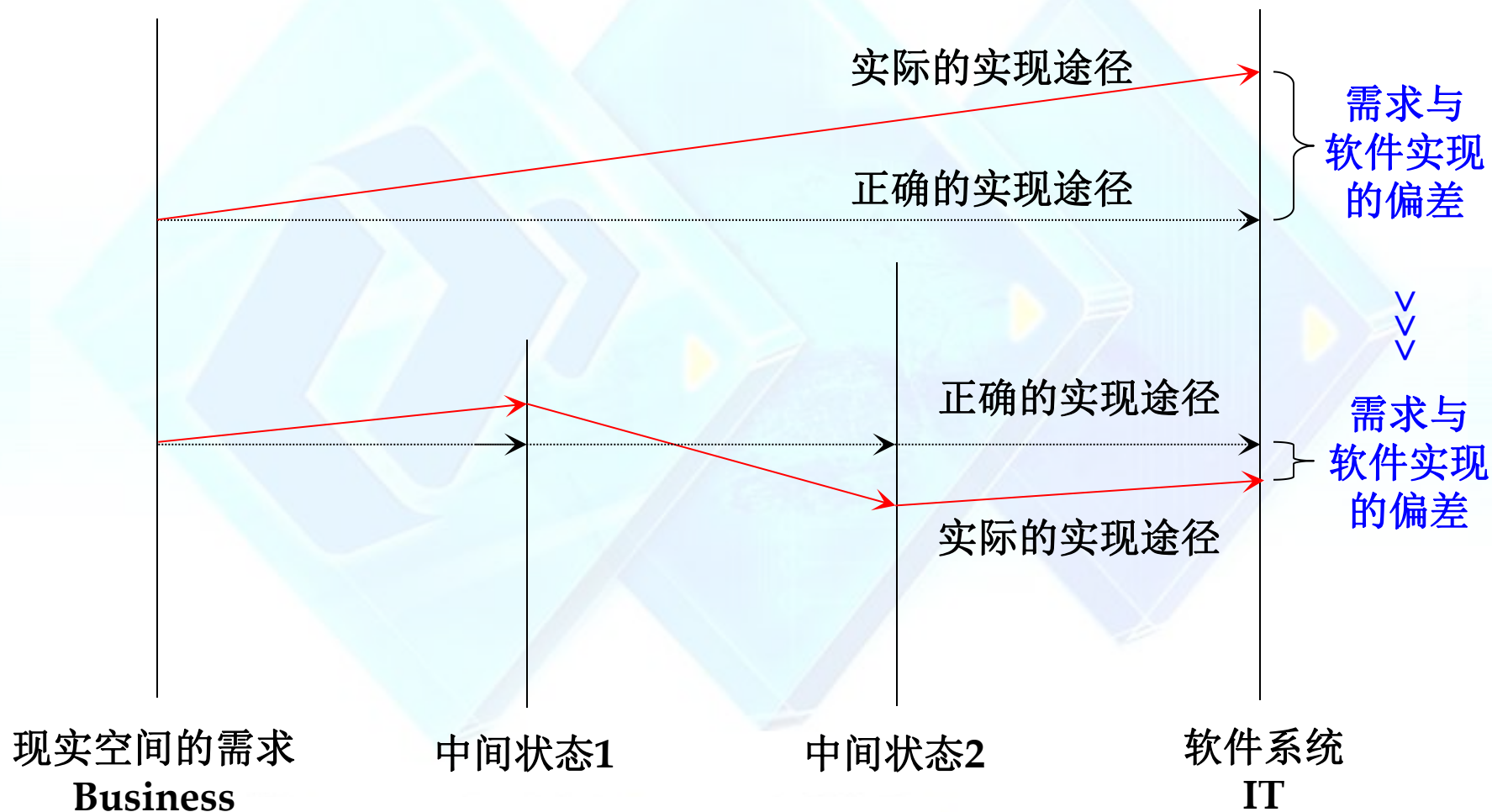
问题：单步映射与多步映射的优缺点分别都是什么？

## 单步映射与多步映射

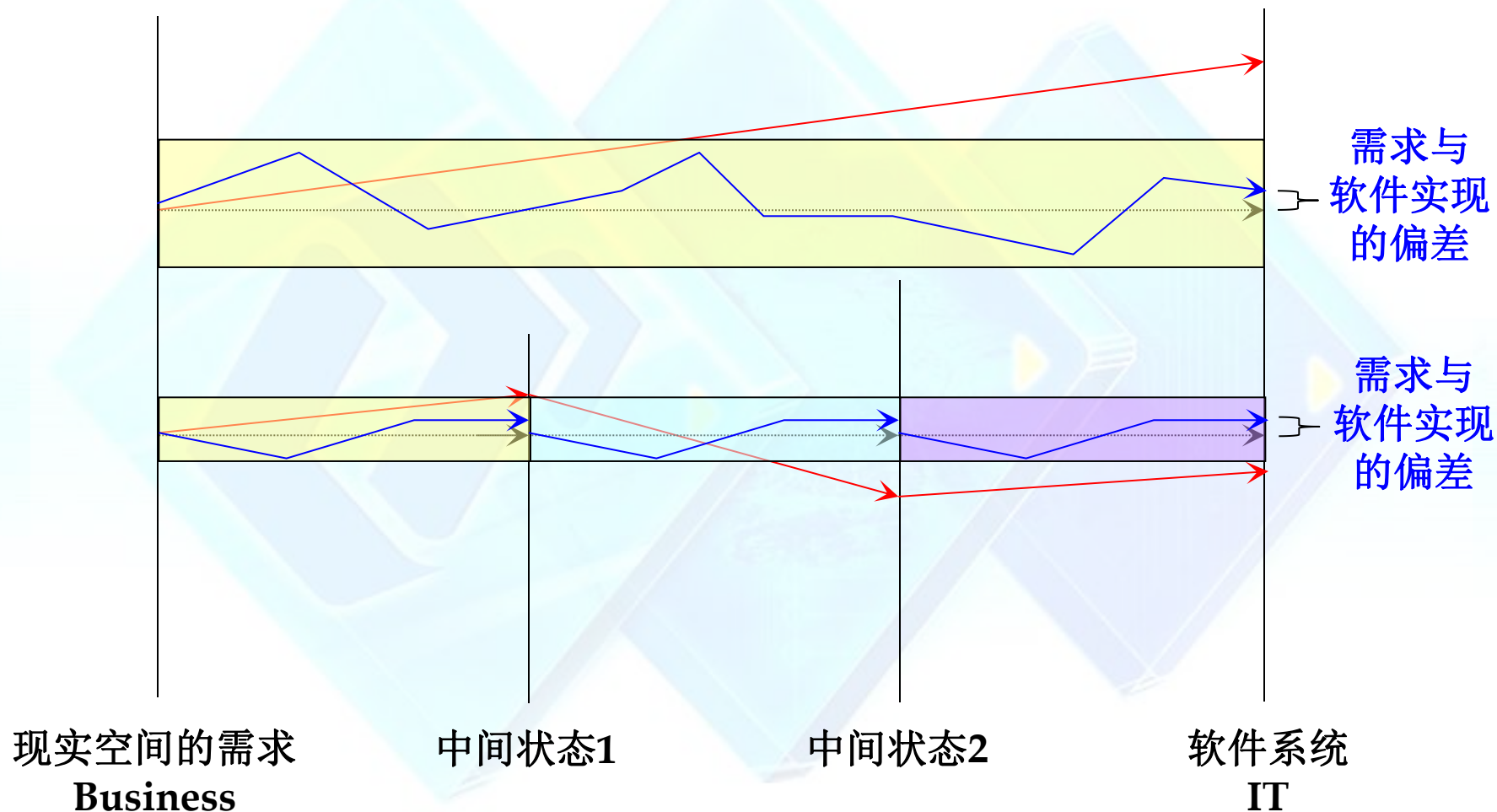




## 单步映射与多步映射



## 软件工程的作用



软件工程本质：用严格的规范和管理手段来缩小偏差，通过牺牲“时间”来提高“质量”



## 软件工程的两个映射之一：概念映射

- 概念映射：问题空间的概念与解空间的模型化概念之间的映射

- 例如：

- “学生” → **Class Student (No, Name, Dept, Grade)**

- “计算机学院大三学生张三” →

**Object Student(150310501,张三, 计算机,大三)**

- “学生成绩” →

**Struct StudentScore (StudentNo, CourseNo, Score)**

- “张三的软件工程课成绩为85分” →

**ZS\_SE\_SCORE (150310501, 软件工程, 85)**

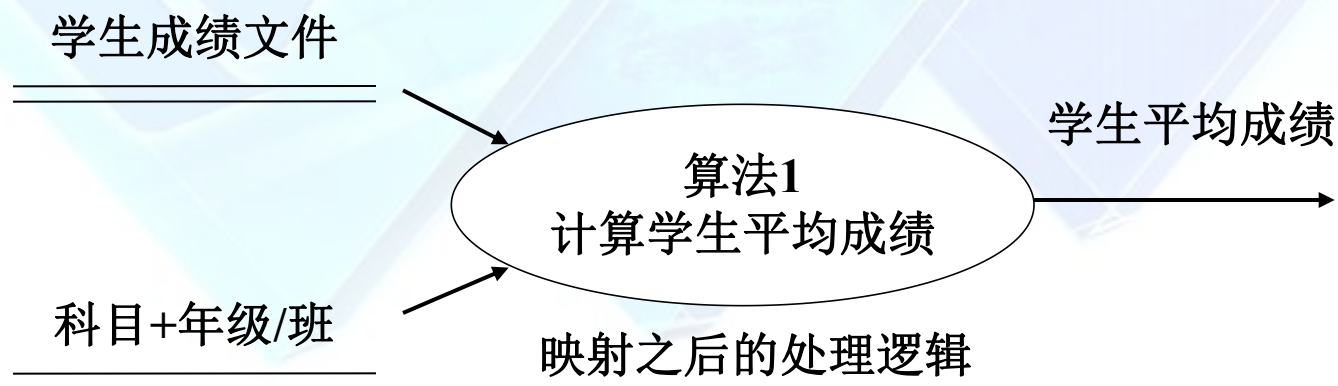
## 软件工程的两个映射之二：业务逻辑映射

- 业务逻辑映射：问题空间的处理逻辑与解空间处理逻辑之间的映射

- 例如：

— 计算某班学生的平均分数 →

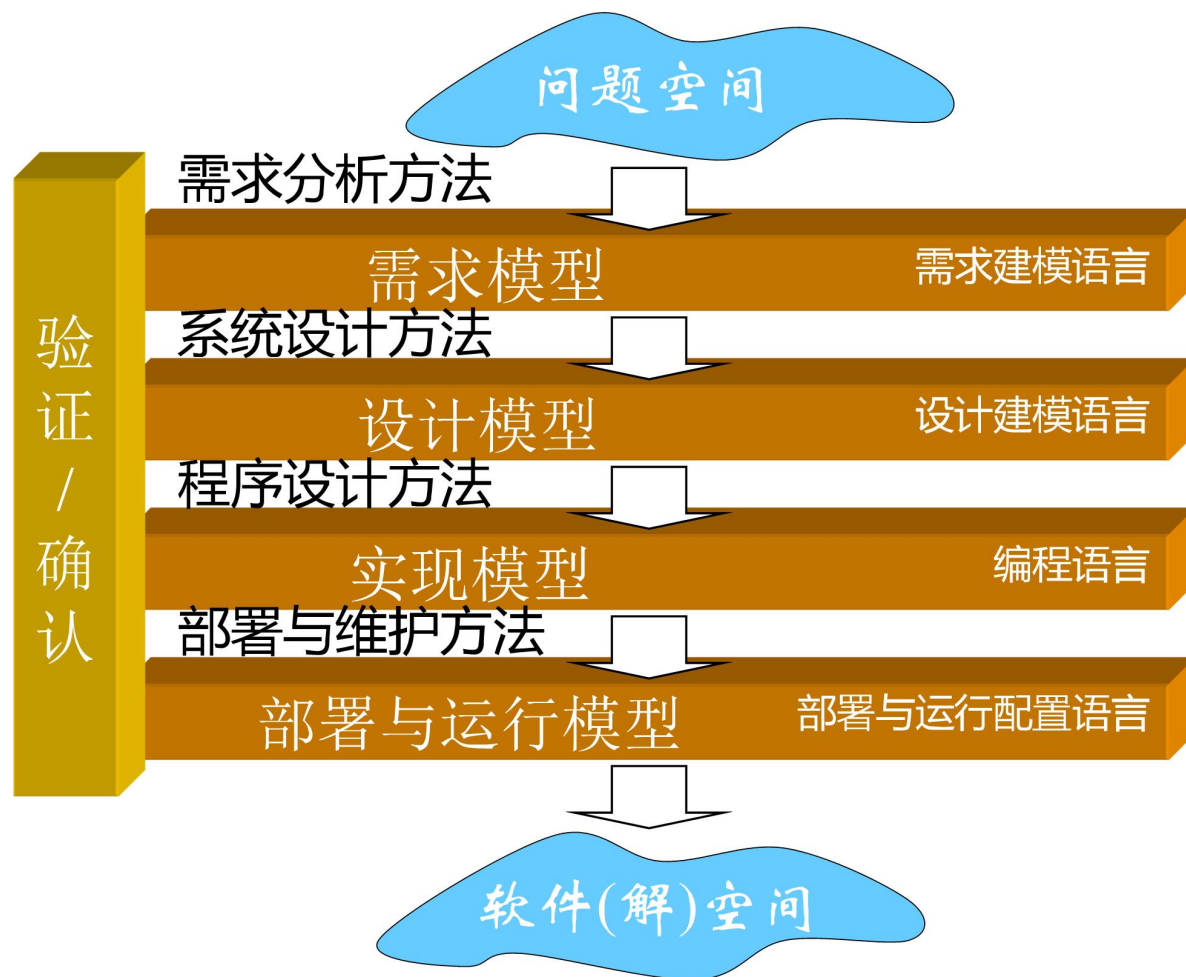
**double calculateAverageScore (Struct [] scores) {检索+平均算法;}**



## 软件工程的作用

- 为了实现以上两个映射，软件工程需要解决以下问题：
  - 需要设置哪些抽象层次——单步映射？多步映射？几步？
  - 每一抽象层次的概念、术语与表达方式——公式？图形？文字？
  - 相邻的两个抽象层次之间如何进行映射——需要遵循哪些途径和原则？

# 软件工程：不同抽象层次之间的映射过程



## 软件工程：不同抽象层次之间的映射过程

- 需求分析：在一个抽象层上建立需求模型的活动，产生需求规约 (**Requirement Specification**)，作为开发人员和客户间合作的基础，并作为以后开发阶段的输入

现实空间的需求 → 需求规约

- 软件设计：定义实现需求规约所需的系统内部结构与行为，包括软件体系结构、数据结构、详细的处理算法、用户界面等，即所谓设计规约 (**Design Specification**)，给出实现软件需求的软件解决方案

需求规约 → 设计规约



## 软件工程：不同抽象层次之间的映射过程

- 实现：由设计规约到代码的转换，以某种特定的编程语言，对设计规约中的每一个软件功能进行编码

设计规约 → 代码

- 验证/确认：一种评估性活动，确定一个阶段的产品是否达到前阶段确立的需求(**verification**)，或者确认开发的软件与需求是否一致(**validation**)

现实空间的需求  
需求规约  
设计规约  
代码



现实空间的需求  
需求规约  
设计规约  
代码

Bad smell

Good smell

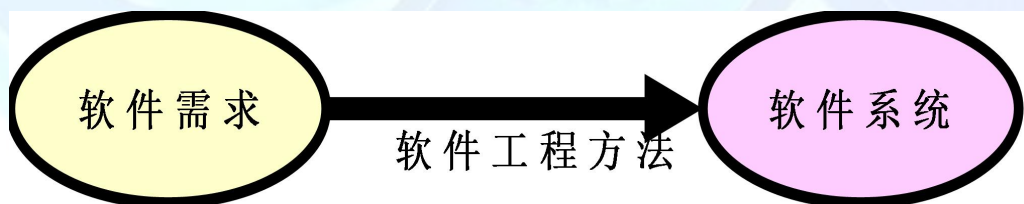


## 软件工程核心思想

- 1 软件工程的本质：不同抽象层次之间的映射与转换
- 2 软件工程所关注的目标
- 3 软件开发中的多角色
- 4 软件工程 = 最佳实践
- 5 软件工程的四个核心理论概念

## 软件工程所关注的对象

- **产品**：各个抽象层次的产出物
- **过程**：在各个抽象层次之间进行映射与转换
- 软件工程具有“**产品与过程二相性**”的特点，必须把二者结合起来去考虑，而不能忽略其中任何一方



## 软件工程所关注的目标

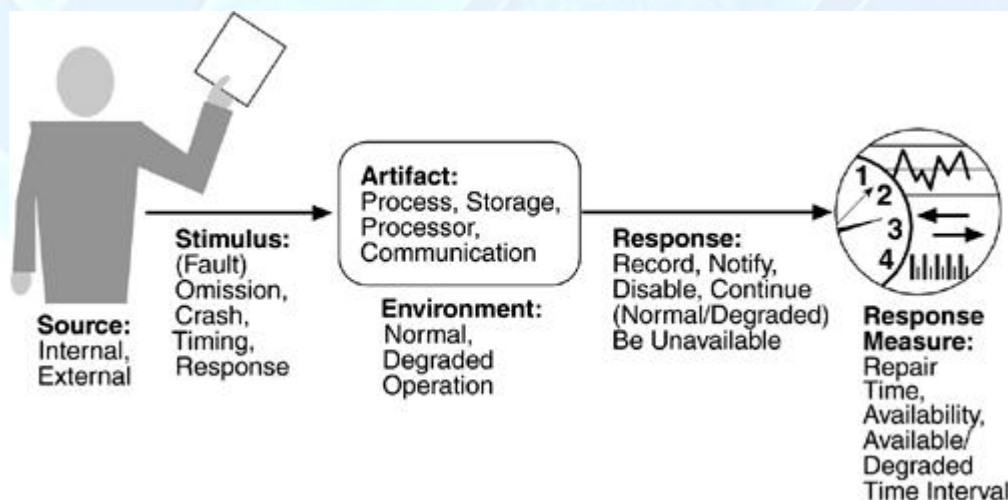
- 功能性需求(Functional Requirements): 软件所实现的功能达到它的设计规范和满足用户需求的程度
  - 功能1、功能2、...、功能n
  - 完备性: 软件能够支持用户所需求的全部功能的能力
  - 正确性: 软件按照需求正确执行任务的能力
    - 正确性描述软件在需求范围之内的行为
  - 健壮性: 在异常情况下, 软件能够正常运行的能力
    - 容错能力
    - 恢复能力
    - 而健壮性描述软件在需求范围之外的行为
  - 可靠性: 在给定的时间和条件下, 软件能够正常维持其工作而不发生故障的能力

## 软件工程所关注的目标

- 非功能性需求(Non-Functional Requirements): 系统能够完成所期望的工作的性能与质量
  - 效率: 软件实现其功能所需要的计算机资源的大小, “时间-空间”
  - 可用性: 用户使用软件的容易程度, 用户容易使用和学习
  - 可维护性: 软件适应“变化”的能力, 系统很容易被修改从而适应新的需求或采用新的算法、数据结构的能力
  - 可移植性: 软件不经修改或稍加修改就可以运行于不同软硬件环境(CPU、OS和编译器)的能力
  - 清晰性: 易读、易理解, 可以提高团队开发效率, 降低维护代价
  - 安全性: 在对合法用户提供服务的同时, 阻止未授权用户的使用
  - 兼容性: 不同产品相互交换信息的能力
  - 经济性: 开发成本、开发时间和对市场的适应能力
  - 商业质量: 上市时间、成本/受益、目标市场、与老系统的集成、生命周期长短等

## 典型NFR举例：可用性(availability)

- 当系统不再提供其规格说明中所描述的服务时，就出现了系统故障，即表示系统的可用性变差
- 关注的方面：
  - 如何检测系统故障、故障发生的频度、出现故障时的表现、允许系统有多长时间非正常运行、如何防止故障发生、发生故障后如何消除故障、等等





## 典型NFR举例：可用性(availability)

## ■ 错误检测(Fault Detection)

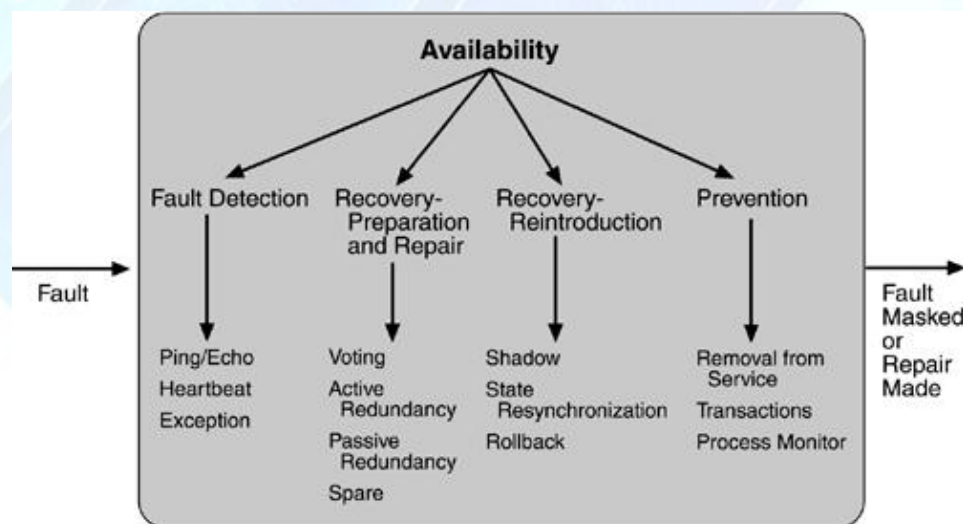
- 命令-响应机制(ping-echo)、心跳(heartbeat)机制、异常机制(exception)

## ■ 错误恢复(Recovery)

- 表决、主动冗余(热重启)、被动冗余(暖重启/双冗余/三冗余)、备件(spare)
- Shadow操作、状态再同步、检查点/回滚

## ■ 错误预防(Prevention)

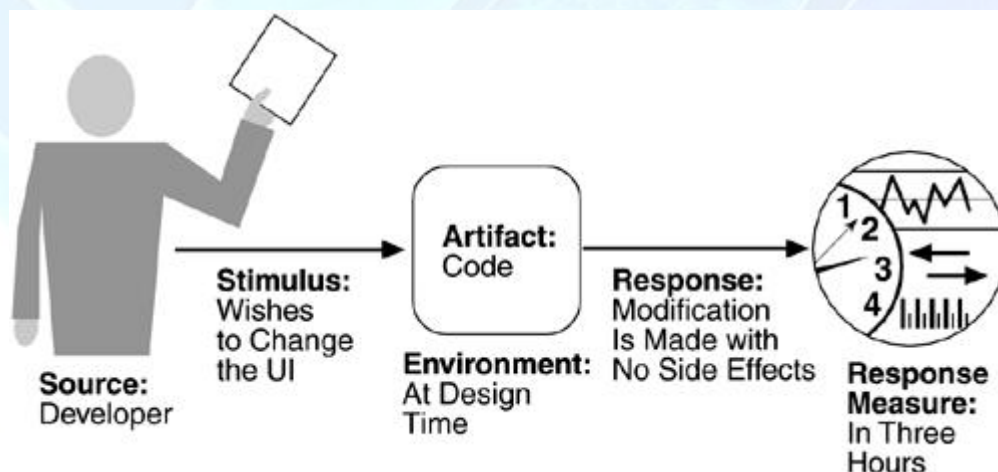
- 从服务中删除、事务、进程监视器





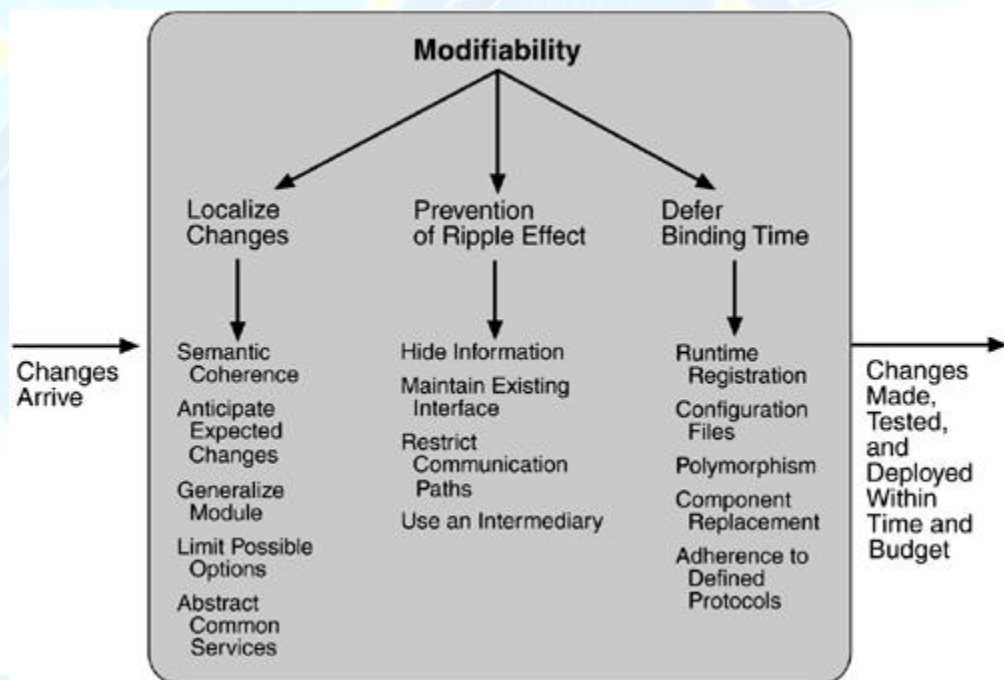
## 典型NFR举例：可维护性/修改性(modifiability)

- 可以修改什么——功能、平台(HW/OS/MW)、外部环境、质量属性、容量等
- 何时修改——编译期间、构建期间、配置期间、执行期间
- 谁来修改——开发人员、最终用户、实施人员、管理人员
- 修改的代价有多大？
- 修改的效率有多高？



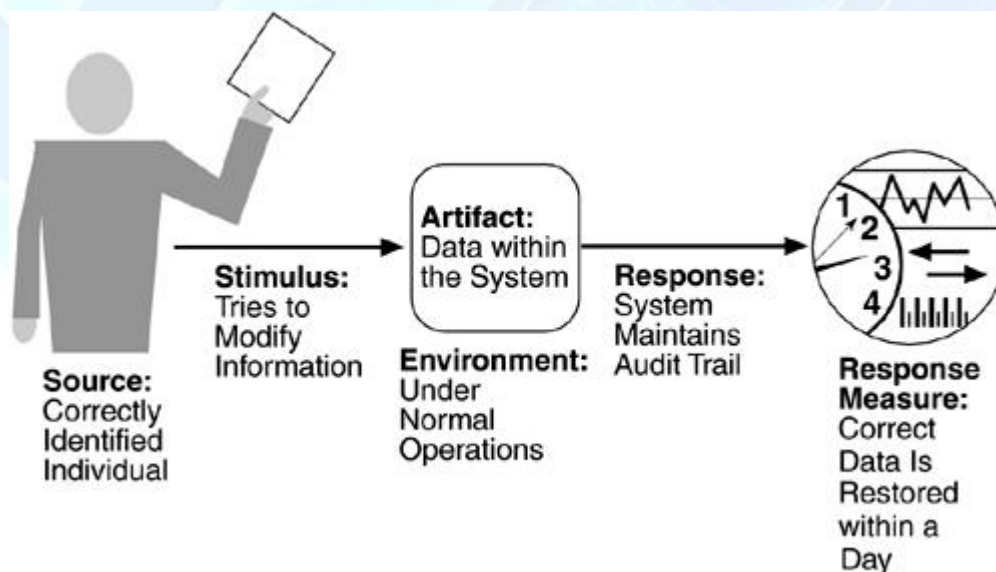
## 典型NFR举例：可维护性/修改性(modifiability)

- 目标：减少由某个修改所直接/间接影响的模块的数量
- 常用决策：
  - 高内聚/低耦合、固定部分与可变部分分离、抽象为通用模块、变“编译”为“解释”
  - 信息隐藏、保持接口抽象化和稳定化、适配器、低扇出
  - 推迟绑定时间 — 运行时注册、配置文件、多态、运行时动态替换



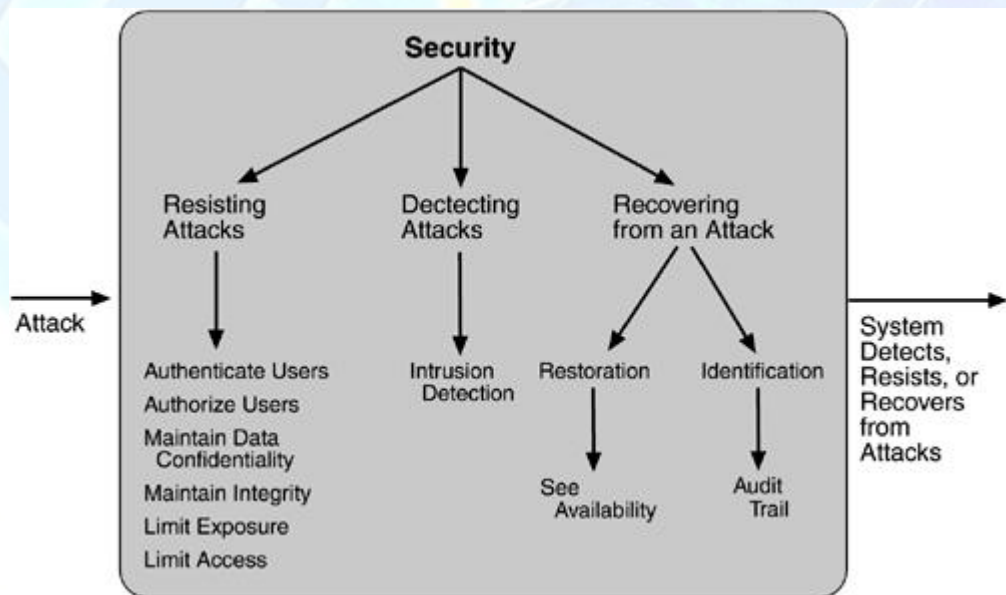
## 典型NFR举例：安全性(security)

- 安全性：系统在向合法用户提供服务的同时，阻止非授权使用的能力
  - 未经授权试图访问服务或数据
  - 试图修改数据或服务
  - 试图使系统拒绝向合法用户提供服务
- 关注点：抵抗攻击、检测攻击、从攻击中恢复



## 典型NFR举例：安全性(security)

- 抵抗攻击——对用户进行身份认证、对用户进行授权、维护数据的机密性、限制暴露的信息、限制访问
- 检测攻击——模式发现、模式匹配
- 从攻击中恢复——将服务或数据回复到正确状态、维持审计追踪



## 不同目标之间的关系—折中(tradeoff)

- 不同类型的软件对质量目标的要求各有侧重：
  - 实时系统：侧重于可靠性、效率
  - 生存周期较长的软件：侧重于可移植性、可维护性
- 多个目标同时达到最优是不现实的：
  - 目标之间相互冲突



## 软件工程核心思想

- 1 软件工程的本质：不同抽象层次之间的映射与转换
- 2 软件工程所关注的目标
- 3 软件开发中的多角色
- 4 软件工程 = 最佳实践
- 5 软件工程的四个核心理论概念



## 软件开发中的多角色

- 在软件开发过程中同样需要多种角色之间紧密协作，才能高质量、高效率的完成任务
- 客户单位(Client, 甲方):
  - 决策者(CXO)、终端用户(End User)、系统管理员(Administrator)
- 软件开发公司(Supplier, 乙方):
  - 决策者(CXO)
  - 软件销售与市场人员
  - 咨询师、需求分析师
  - 软件架构师、软件设计师
  - 开发人员：开发经理/项目经理、程序员
  - 维护人员

## 视角不同，需求各有不同

开发人员	易用性
开发经理	易调试性
销售人员	可修改性
顾客	可测试性
终端用户	结构清晰性
维护人员	功能性
系统管理员	价格
	开发成本
	按时交付
	性能
	稳定性与可维护性
	易安装性
	易集成性

不同的角色，他们所关心的非功能需求都有哪些？

不同角色的关注点之间，是否有重叠的情况？

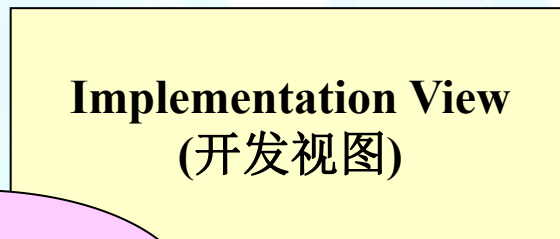
不同角色的关注点之间，是否有冲突的情况？

## 视角不同，需求各有不同

最终用户：功能需求

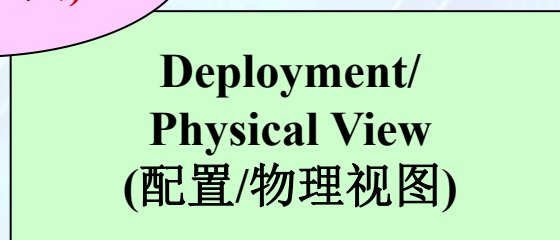
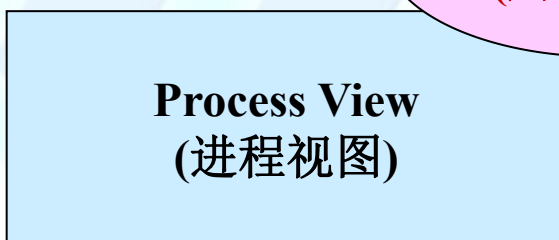


编程人员：静态软件模块(源代码、数据文件)的组织与管理



Use case View  
(用例视图)

架构师：体系结构的设计与实现



系统集成人员：运行时性能、可扩展性、吞吐量等

系统部署人员：运行时系统拓扑、安装、通信等



一人包打天下



团队协作完成任务



## 软件工程核心思想

- 1 软件工程的本质：不同抽象层次之间的映射与转换
- 2 软件工程所关注的目标
- 3 软件开发中的多角色
- 4 软件工程 = 最佳实践
- 5 软件工程的四个核心理论概念

## 软件工程=最佳实践

- 软件系统的复杂性、动态性使得：
  - 高深的软件理论在软件开发中变得无用武之地
  - 即使应用理论方法来解决，得到的结果也往往难以与现实保持一致
- 因此，软件工程被看作一种实践的艺术：
  - 做过越多的软件项目，犯的错误就越少，积累的经验越多，承接项目的成功率就越高
  - 对新手来说，要通过多实践、多犯错来积累经验，也要多吸收他人的失败与教训与成功的经验

——当你把所有的错误都犯过之后，你就是正确的了！



## 软件工程=最佳实践

- 在软件工程师试图解决“软件危机”的过程中，总结出一系列日常使用的概念、原则、方法和开发工具
- 这些实践经验经过长期的验证，已经被证明是更具组织性、更高效、更容易获得成功
- **大部分的这些实践都没有理论基础！**



## 最佳实践的例子

### ■ 软件工程的七条原理(B.W. Boehm, 1983)

- 用分阶段的生命周期计划严格管理
- 坚持进行阶段评审
- 实行严格的产品控制
- 采用现代程序设计技术
- 结果应能清楚地审查
- 开发小组的人员应少而精
- 不断改进开发过程



### ■ IBM RUP最佳实践原则:

- 迭代化开发
- 需求管理
- 使用基于构件的体系结构
- 可视化软件建模
- 持续质量验证
- 控制软件变更

### ■ 与客户沟通的最佳实践原则:

- 原则1: 倾听
- 原则2: 有准备的沟通
- 原则3: 需要有人推动
- 原则4: 最好当面沟通
- 原则5: 记录所有决定
- 原则6: 保持通力协作
- 原则7: 聚焦并协调话题
- 原则8: 采用图形表示
- 原则9: 继续前进原则
- 原则10: 双赢

但是...

在你自己展开实践之前，别人的任何经验对你来说都是概念——抽象、空洞、无物

## 最佳实践的例子：与客户沟通的最佳实践任务

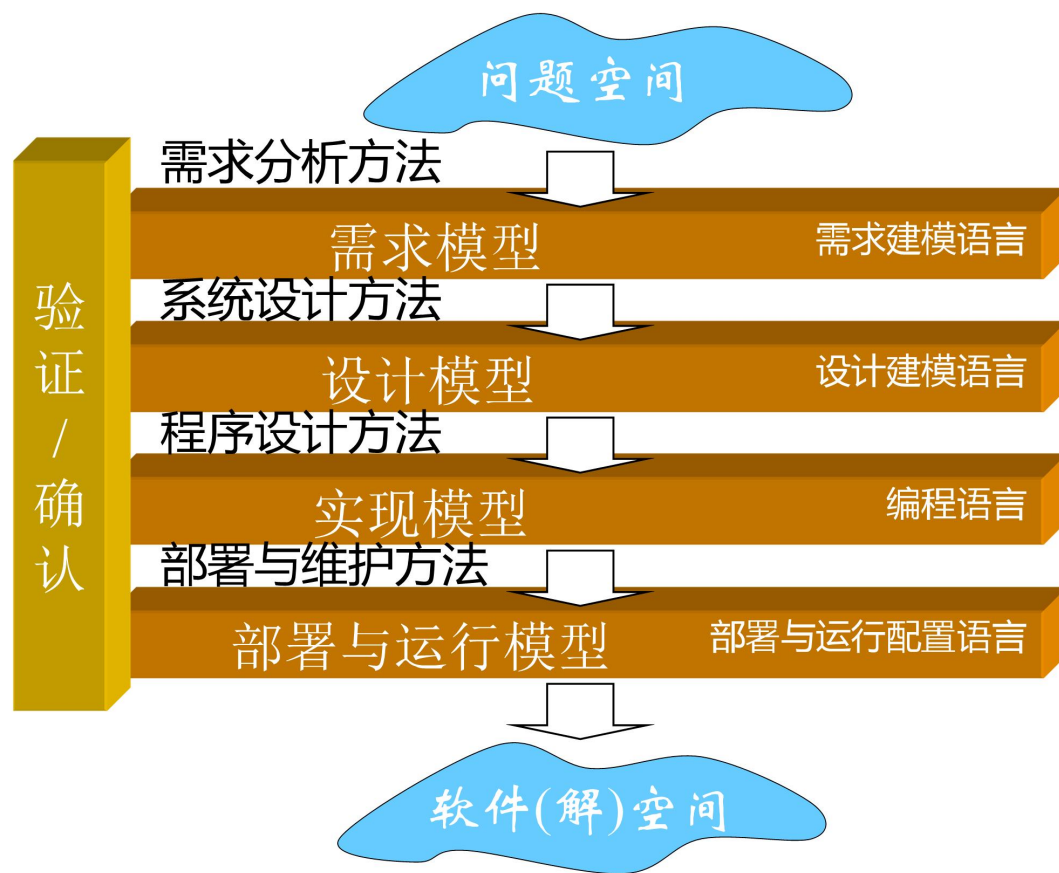
- “最佳实践” --沟通阶段应做的事情：
  - 识别出你需要与客户方的哪些人沟通
  - 找出沟通的最佳方式
  - 确定共同的目标、定义范围
  - 评审范围说明，并应客户要求作出修改
  - 确定若干典型场景，讨论系统应具备的功能/非功能
  - 简要记录场景、输入/输出、功能/非功能、风险等
  - 与客户反复讨论、交换意见，对上述内容进行细化
  - 与客户讨论，为最终确定的场景、功能、行为分配优先级
  - 评审最终结果
  - 双方签字

## 软件工程核心思想

- 1 软件工程的本质：不同抽象层次之间的映射与转换
- 2 软件工程所关注的目标
- 3 软件开发中的多角色
- 4 软件工程 = 最佳实践
- 5 软件工程的四个核心理论概念

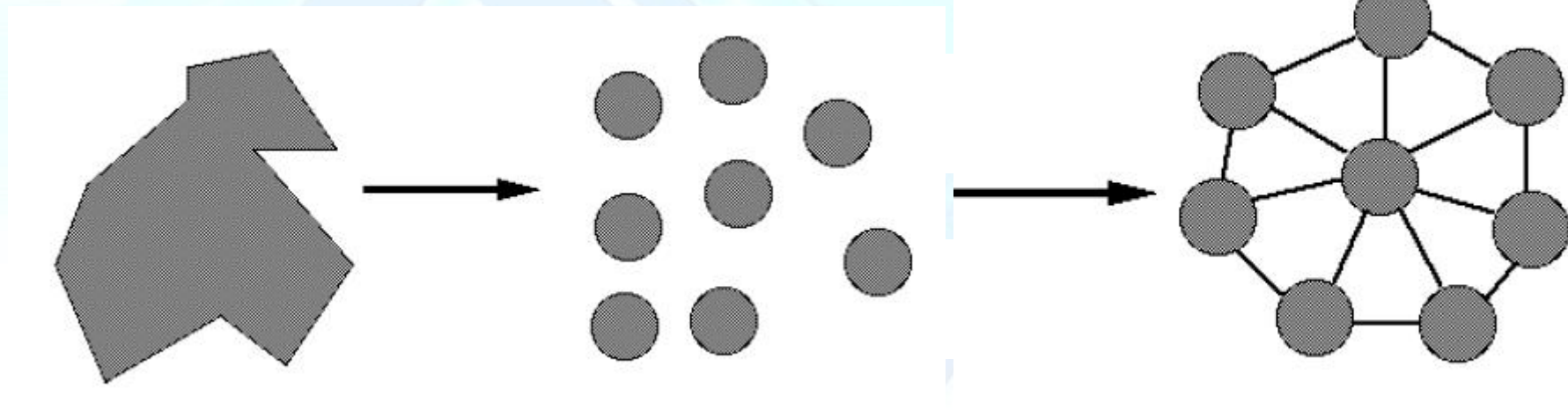
## 软件工程的核心理念

- 概念和形式模型
- 抽象层次
- 大问题的复杂性：分治
- 复用
- 折中
- 一致性和完备性
- 效率
- 演化



## 分而治之(Divide and Conquer)

- 将复杂问题分解为若干可独立解决的简单子问题，并分别独立求解，以降低复杂性
- 然后再将各子问题的解综合起来，形成最初复杂问题的解



**Rome was not built in a day !**

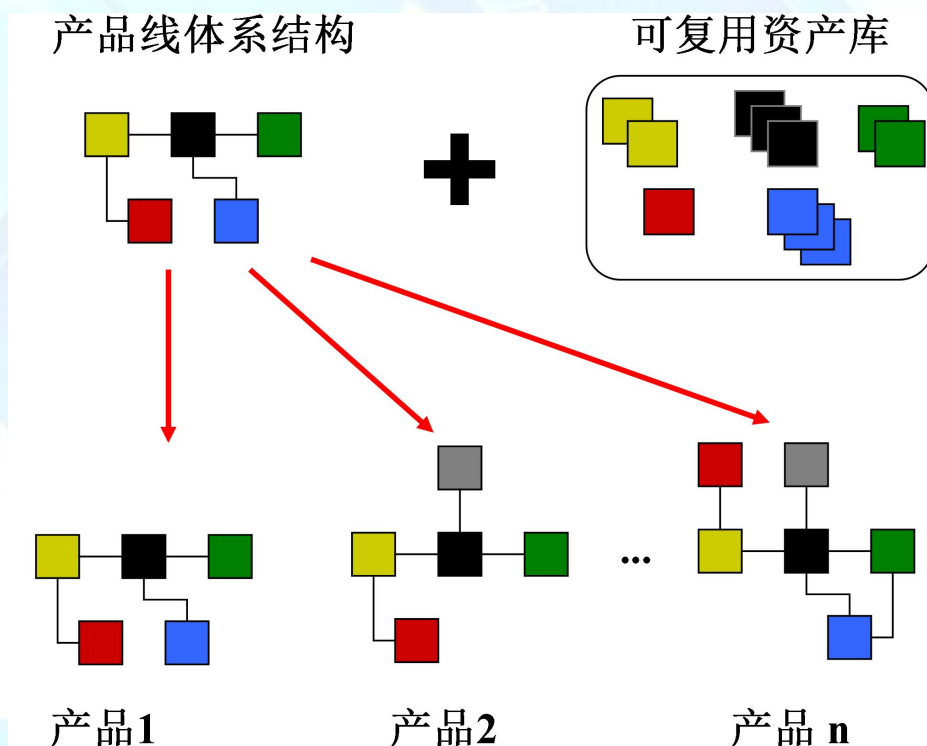
**核心问题：如何的分解策略可以使得软件更容易理解、开发和维护？**



## 复用(Reuse)

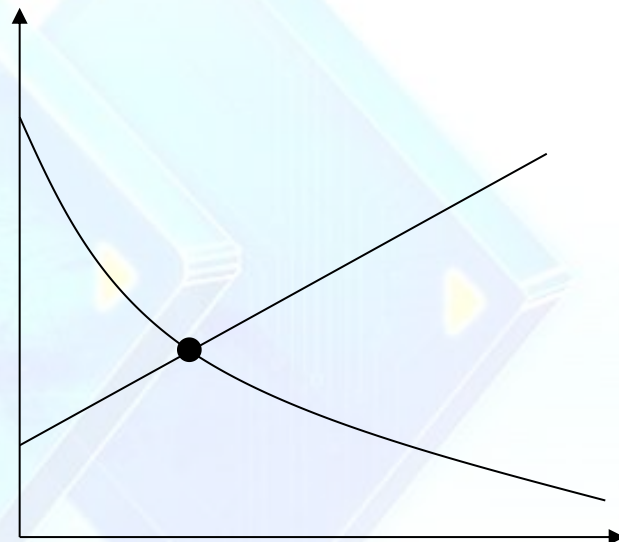
- 在一个新系统中，大部分的内容是成熟的，只有小部分内容是全新的
- 构造新的软件系统可以不必每次从零做起
- 复用已经成功使用的**架构**、**框架**、同类经验的**团队**
- 直接使用已有的**软构件**，即可组装成新的系统
- 复用已有的**功能模块**，既可以提高开发效率，也可以改善新开发过程中带来的质量问题

**Don't re-invent the wheel !**  
**Don't Repeat Yourself !**



## 折中(Trade-off)

- 不同的需求之间往往存在矛盾与冲突，需要通过折中来作出的合理的取舍，找到使双方均满意的点
- 例如：
  - 在算法设计时要考虑空间和时间的折中
  - 低成本和高可靠性的折中
  - 安全性和速度的折中



**The doctrine of the mean, the art of balance**

**核心问题：**如何调和矛盾（需求之间、人与人之间、供需双方之间、...）

## 演化(Evolution)

- 软件系统在其生命周期中面临各种变化
- **核心问题**：在设计软件的初期，就要充分考虑到未来可能的变化，并采用恰当的设计决策，使软件具有适应变化的能力
- **即**：可修改性、可维护性、可扩展性

### Continuing change and increasing complexity

