软件体系结构概述

周颢

kitewind@ustc.edu.cn

什么是软件体系结构

- 软件体系结构的产生
  - 软件体系结构日益作为软件工程师所遵循的重要原则浮现（emerging）
    - 与传统的学科不同，作为软件系统的新概念，它并非成型的、有精确定义的
  - 而是在软件工程师们
    - 面对日益复杂的系统的设计与构造问题
    - 寻求理解软件系统的更好的方法
    - 寻求构造更大更复杂的软件系统的更有效的方法
  - 的过程中，作为设计抽象的自然演化而出现的。

什么是软件体系结构

- 软件功能要求　　简单→ 复杂
  个人英雄时代已经结束

- 结构性　VS　算法与数据结构
  再好的手工艺品也比不上机械化大生产

- 软件体系结构概念的提出与应用
  挑战与机遇

什么是软件体系结构

- 软件体系结构的重要性
  - 传统软件系统构造的着眼点
    - 算法的选择
    - 数据结构的设计
    - 数据库的构造
  - 当软件系统的规模与复杂度不断增大时，整个系统的设计与描述变得更加重要

什么是软件体系结构

- 软件体系结构层的设计目标：系统的结构
  - 将系统划分为若干组件
  - 全局控制结构
  - 通信、同步与数据交换协议
  - 设计元素之间的功能分配与合成
  - 物理分布
  - …
- 这些都面临一系列不同的选择，需要在考虑可扩展性与性能等问题的基础上作出选择，这就是设计问题。
- 一般意义上，设计问题总是在一些矛盾的要求之间寻找折衷点，这个过程不仅仅是软件设计的问题，同时涉及用户交流。

什么是软件体系结构

- SA的描述包含
  - 构成系统的各个部件的描述
  - 部件间的交互（interactions）
  - 部件构成与部件合成的模式（pattern）以及在这些模式上的约束

- 注：前两部分的描述，对于任意由不同部分构成的系统而言都是需要的，软件体系结构作为一门学科，是将此提升到设计层原则的高度；同时，用通过实践过程总结的模式（Pattern）作为设计的指导。

- Component的集合 + Component间的交互

什么是软件体系结构

- SA描述的现状：非正式、抽象
  - 通常用在长期的实践中非正式的浮现【即：没有通过理论抽象与统一的形式化描述】的特殊模型来描述
  - 通常用线/框图与一些文字描述
  - 线框体现出形象的结构，文字则描述符号的含义并提供在部件与其间的交互中所做的特定选择的理由

什么是软件体系结构

- 不规范的体系结构描述
  - We have chosen a *distributed, object-oriented* approach to managing information
  - The easiest way to make the canonical sequential complier into concurrent complier is to *pipeline* the execution of the compiler phases over number of processors…
- 基于自然语言的描述，有效但是不正式不严格。

什么是软件体系结构

- 问题
  - 为什么为产生这样非正式、抽象的描述？
  - 这样的体系结构的描述对软件工程是否有用？

什么是软件体系结构

- 【关于非正式】：对软件系统的组织而言，经过长期的积累，已有一系列习惯用语、模式、风格，构成共享的、语义丰富的（体系结构用）词汇

  - 例如，说一个系统是pipe-filter结构的风格，即该系统主要涉及流，且系统的功能可由构成系统的过滤器构成，系统延迟与吞吐量可相对直接的得到。即：该非正式的描述对软件工程师具有相当的系统结构信息。

什么是软件体系结构

- 【关于抽象】：虽然体系结构相对于具体的实现与组件构造的细节要抽象，但提供了更广的、结构层上理解系统层上的问题的自然的骨架
  - 这个骨架在系统没有形成的时候就已经在起作用
  - 例如：系统可能的演化轨迹、可扩展性、通讯的模式、数据组织模式等
  - 通过对主要系统需求（*此时主要是非功能需求，以及功能需求的共性构件*）对应描述，展现系统满足全部需求的能力

什么是软件体系结构

- 体系结构描述语言ADL

  Architecture Description Language

  - 需要更好的、更严格的符号系统、理论与分析技术
    - 模块互联语言
    - 用于某个特定领域的系统框架与模板
    - 模块集成的形式化描述等

  - 现状

    还没有统一的、为人们所接受的术语系统描述这些领域的共性元素

什么是软件体系结构

- 总体现状
  - 没有完整的软件体系结构理论
- 现有软件体系结构的基本内容
  - 一些体系结构描述的基本成分
  - 一些体系结构模式、风格、习惯语等

什么是软件体系结构

- 定义　　目前没有公认最好的定义
  - **Architecture of a software system defines that system in terms of computational components and interactions among those components . [Mary 98]**
  - **The Software Architecture of a program or a computing system is the structure or the structures of the system , which comprise software components , the externally visible properties of these components , and the relationships among them . [Bass98]**

什么是软件体系结构

- 系统的构成与结构关系
  - 系统的构成？ 部件
  - 部件如何拼接？ 连接器

- 部件Component（组件）

  部件通常是客户端、服务器、数据库、过滤器、层次系统中的某个层次、或系统中某种实体或功能的抽象。

  部件间的交互可能通过过程调用、共享数据等简单机制实现，也可能是CS协议、数据库存取协议、异步事件广播、管道流等复杂机制实现。

什么是软件体系结构

- 体系结构是系统的抽象，用抽象的部件、部件外部可见的性质、部件之间的关系来描述系统。
- 由于体系结构是抽象的，因此压缩了存局部信息，即部件的私有细节不属于体系结构的描述范围。
- 系统由许多结构构成，这些结构通常称为View(视图),一种视图只能描述从某个角度看到的系统，而非全部，同时，视图的集合不是固定的。

什么是软件体系结构

- 体系结构的四视图观点
  - 概念体系结构Conceptual Architecture
    - 部件、连接器、性能
    - 应用问题的分解和划分。
  - 模块体系结构Module Architecture
    - 子系统、模块、引入（Import）、引出（Exports）、模块的界面、管理、控制和一致性等
  - 代码体系结构 Code Architecture
    - 文件、目录、库、包含（includes）、软件的配置管理、系统建造等
  - 运行体系结构Execution Architecture
    - 任务、线程、进程、性能、调度、动态分配和不同执行系统之间的接口等

软件设计的层次

- 系统（不限于软件系统）设计划分为若干个层次，区分各个层次之间的区别，划分各层所需解决的设计问题是至关重要的。每个层次上定义
  - 部件：primitive and composite
    - 整个问题如何用小的部件完成
  - 合成规则：合成部件或系统的构造规则
    - 部件如何向上层组合
  - 行为规则：系统的语义
    - 提供了什么功能

- 这些规则在每一个层次上是不同的，每个层次上有各自的符号系统、设计问题、分析技术，由此，各个层次上的设计可以独立进行，而每一层是上一层的实现。

软件设计的层次

- 以计算机硬件系统的设计层次为例
  - 各层解决不同的问题
  - 各层由不同的部件依赖于本层的结构规则构成
  - 各层有自己的符号、分析技术用于解决本层的设计问题
  - 各层均允许子结构：抽象＋组合（abstraction and composition）即本层的部件与结构
  - 各层的primitive部件到下一层的部件之间转换

软件设计的层次

- 软件设计的层次性
  - 结构级：通过部件与部件间的协作达到系统功能。

    本层的部件是模块，模块之间的交互通过各种方式完成（调用、管道、共享、同步等）。指导子系统、模块组合的规则（非物理上强制）为一些设计模式（Design Pattern ）。

  - 代码级：包含算法与数据结构的设计。

    部件是编程语言的原子如数、字符、指针、线程等，连接的操作符是该语言的算术或数据操作符，组合机制包括记录、数组、过程等。

  - 执行级：解决内存映像、数据组织、栈的调用、寄存器分配等问题。

    部件为硬件所支持的位模式，用机器代码来表述操作符、组件的组合。

软件设计的层次

- 从60年代到80年代，软件开发人员集中在下面两层上，到现在已经彻底解决了。而结构级的理解停留在直觉和经验上。

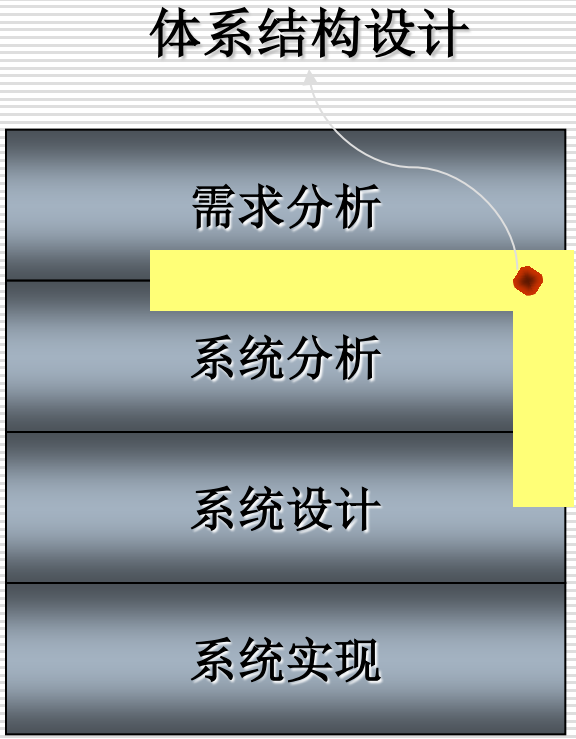
- 通常软件系统的描述都包含一些文字和图表说明，但没有统一的语法与语义，同时，这些描述与实际系统之间所需的模型往往还有很大的差距。

  【注：UML：Uniformed Modeling Language的出现就是为统一这些描述语言。】

软件设计的层次

- 结合软件工程角度

  软件的需求分析、系统分析、系统设计均可视为设计层，编码为实现层。体系结构设计位于需求分析与系统分析之间，但同时也包含了一些相当重要的设计问题。



体系结构与软件工程

软件工程

- 软件工程 Software Engineering
  - 出现于1968年
  - 目前包含一些管理方法、开发过程、软件工具等
  - 公认成功的软件开发必须以严格有序的项目管理、遵循设计良好的开发过程、以开发工具与一系列文档规范并提高效率
  - 我国的软件工业落后的原因，不在于不了解先进的方法，而在于如何在软件企业中有效的实施

软件工程

- 软件工程的目标
  - 基于使得软件开发可控，能以最小的代价开发出最成功的软件产品【又是矛盾的，从来无法都达到】
  - Creating cost-effective solutions to practical problems by applying scientific knowledge building things in the service of mankind .

软件工程

分解：

- **代价合理的解决方案：**工程问题不仅仅是要解决问题，而且要经济的使用资源，自然，最重要的就是钱（自然在严格规范的组织中，人力完全可以换算成钱），同样，包括时间（换算成效益还是钱）。

- **针对实际问题：**工程所面对的是在工程领域外的人们：客户所面临的问题，即使是面对软件领域自身的问题，如构造好的编译器，最终也是为了更好的解决客户的问题，否则无法生存下去。【这提出作为大多数软件体工程师必须面对问题：理解客户的行业。】

软件工程

- 分解
  - **运用科学知识：**成功的工程领域如硬件、建筑等无不充分利用科学特别是数学的知识。【补充：在工程领域中，科学知识往往是工具，如何有效的利用工具是更关键的问题，即：组织与管理问题。】
  - **构造实体：**工程强调产出的解是可解决实际问题的工具，而非方案。
  - **服务人类：**工程不仅仅服务于客户，更通过客户以及工程工程中形成的技术与经验支撑社会服务社会。【服务意识！】

软件工程

- 科学知识如何对工程有用？如何用？
  - 对特定的技术领域问题，将相关的科学知识*条例化*描述成对使用者直接有用的方式
  - 对实际中常遇到的问题给出直接的答案
  - 由此，一般的工程师可以*直接运用*这些手段解决问题
  - 工程师直接依赖于前人的经验解决过程

软件工程

- 示例：CMM软件开发流程
  - 需求规格说明书
  - 概要设计说明书
  - 详细设计说明书
  - 单元测试计划及报告
  - 系统测试计划及报告

软件工程

- 软件开发的历史包含成功与失败，成功者或有优异的性能或能稳定的工作，失败者原因则多
  - 对客户领域问题缺乏了解，需求阶段埋下祸根
  - 看对了病下错了药
  - 设计不完备
  - 从设计到实现的过程转化失败
- 这些原因可以归结到三个：
  - 个人素质不足
  - 组织、管理、规范不力
  - 缺乏相应的科学知识转化的组织管理规范
- 结果只有两个：不能工作或者超支超期没完没了的维护

常规 or 创新 ？

- *平凡的设计*与*创新的设计*之间的区别
  - Routine Design: 解决类似的问题，主要（大部分）使用以前的解决方案
  - Innovative Design: 对不熟悉的问题寻找新的解决方案
  - 显然，创新设计比平凡设计要少的多（需要的少，同时能做的人也少），所以
    - 后者是工程师们的bread and butter
    - *工程的作用就是普及平凡的设计，以及及时地将不平凡的创意变成平凡的例行公事*

常规 or 创新 ？

- 工程实践使得平凡的操作者能创造出复杂的系统，虽然不是特例独行引入入胜，但是能够工作。
  - 工作需要的不见得是天才，对工程规范强的公司而言，有创意但不能转化为效益的天才不如按部就班完成任务的螺丝钉样的傻子

常规 or 创新 ？

- 多数的工程原则的目标在于capture/organize/share设计知识，使得平凡的设计更容易更快速
  - 手册与指南常用于汇集这些信息
  - 但现阶段关于软件设计的符号与内容不足以记录与交流这些有效的设计（原则），因此还没有类似的手册出现

常规 or 创新 ？

- 由于下列原因，软件领域在多数情况下被当作原始的创新而不是例行工作，至少，比我们掌握和普及了已有的软件工程知识、组织原则的理想情况要多的多（在国内尤其如此，几乎是完全如此）

1.软件的抽象性，特别是从用户到软件开发人员之间存在的信息鸿沟使得目标系统往往难以用准确的语言描述出来（即使描述出来，也未必是想要的），传统产业中在组织中传递的实物变成抽象的文档、口述、代码，使得每多一个层次，不可控制性增加一层；

常规 or 创新 ？

2. 工程化要求企业的组织化，软件工程的成熟度与教育普及的程度不够形成不了组织力量。

3. 软件开发由高智商的人垄断。

4. 企业管理规范。比如，公司忙于完成任务，流动性大因而无法顾及总结、规范等严格的条例化的工作（e.g. 有源码，没文档）

常规 or 创新 ？

- 解决：标识例行设计并开发合适的支持
  - 从计算机科学的角度分析成功的设计，提取体系结构层的共性因素；
  - 完善并提取出完整的条例化的工程规则（自然，前提条件是有可描述这些设计规则的符号语言体系）；

常规 or 创新 ？

- 现有的复用的重点集中于寻找代码中，构造例程库 subroutine library 特别是系统调用和通用算术例程，这是多年以来的法宝。这些例程往往比较底层，离高层复用有相当的距离。

  （数据结构、算法等）

- 由此工程师们意识到需要有效的方式共享成功设计的经验

  （软件体系结构）

常规 or 创新 ？

- 一个前化学工程师的一段话：

  In Chem E , when I needed to design a heat exchanger , I used a set of reference to told me what the constants were … and the standard design equation ….

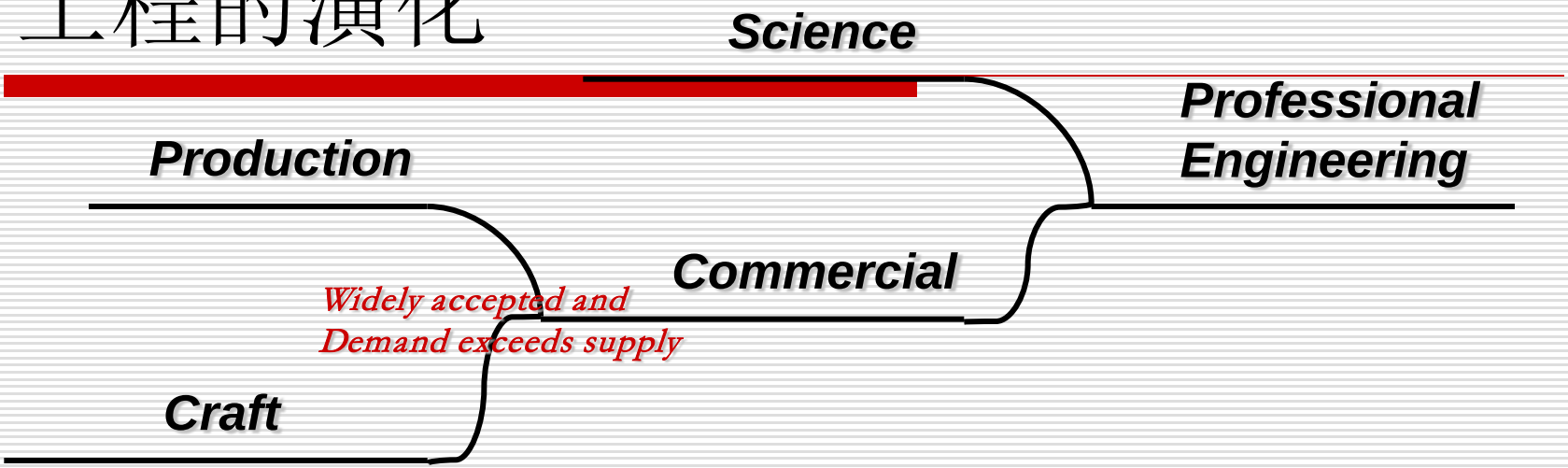
  In general , unless I ,or someone else in my engineering group , has read or remembers and makes known a solution to a past problem , I’m doomed to recreate the solution … I guess … the critical difference is the ability to put together little pieces of the problem that are relatively well known , without having to generate a custom solution for every application …

  I want to make it clear that I am aware of algorithm and code libraries , but they are incomplete solutions to what I am describing . (There is no Perry’s Handbook for the Software Engineering)

工程科学演进的模型

- 工程从特例的实践（ad hoc practice）中产生一般有两个阶段
  - 管理与制造技术的发展使得例行生产成为可能
  - 例行生产的问题刺激支持科学的发展，成熟的科学最终与已建立的业界实践共同完成专业工程。
- 例子：汽车工业流水线的发展

工程的演化



- Virtuosos and talented amateurs
- Intuition and brute force
- Haphazard progress
- Lack of communication
- Casual transmission
- Extravagant use of available materials
- Manufacture for use rather than sale

- Skilled craftsman
- Established procedure
- Pragmatic refinement
- Training in mechanics
- Economic concern for cost and supply of material
- Manufacture for sale

- Educated professionals
- Analysis and theory
- Progress relies on science
- Educated professional class
- Enabling new application through analysis
- Market segmentation by product variety

软件技术的现状

- 面对软件产业，工程化问题同样是：对实际问题提供有效的解决方案。据此判别是否能通过运用科学手段达到或接近此目标。
  - 业界实践＋相应的有效科学手段＝工程实践
  - 此时，科学所提供的结果必须成熟、丰富、有效，足以为实际问题建模，并提供一系列对实践行之有效的工具与手段。
  - 计算机科学已有一些支持实践的方法，但建模与组合使得它们能直接运用还不够。

支持科学的逐步成熟及其作用

- 尽管有批评认为计算机科学与实际软件开发几乎无关，事实上有一些好的成熟模型与理论（经过足够长的时间）已发挥了巨大的作用。
  - 如，算法与数据结构
  - 60年代早期，仅仅是各个程序的一部分，一些解决问题的方式非正式的传播；
  - 60年代中期，优秀的程序员开始认同一个直觉：数据结构正确的话，程序的余下的部分就简单了
  - 60年代晚期，数据结构从单个程序中抽象出来，其根本特性得到描述与分析，成为一门学科

支持科学的逐步成熟及其作用

- 70年代，稳步发展各种支持理论，如算法设计、性能分析、正确性验证等
- 同时，以建立在数据与算法抽象的上的抽象程序为基础，研究一系列问题，如
  - 描述【抽象模型、代数公理等】
  - 软件结构【算法的集成描述等】
  - 语言问题【模块、作用域、用户定义类型等】
  - 信息隐藏【保护信息的完整性】
  - 一致性约束【数据结构的不变量】
  - 组合的规则【声明等】
- 这些构成80年代后软件发展的基础。

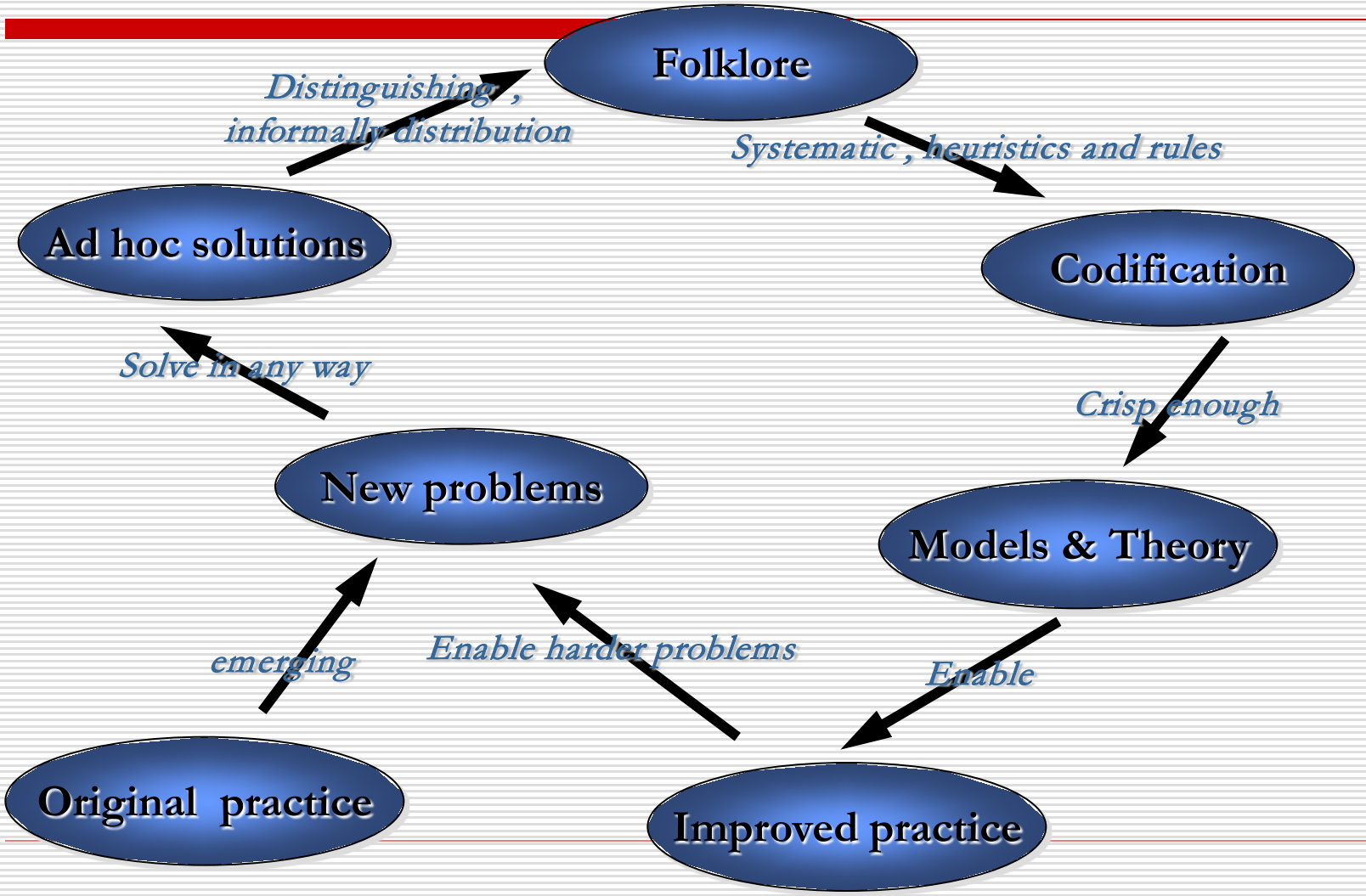
支持科学的逐步成熟及其作用

- 如，编译器的发展
  - 60年代，写编译器本身就是了不起的成就，人们不了解高级语言是什么样子。
  - Algol60 首先系统的使用形式化语法。
  - 自动处理工具（complier-compilers，或现称parser generators）在60年代中期开发出来，70年代实用化。
  - 70年代开始发展语义理论，80年代在编译器构造的自动化上取得重要进展。

支持科学的逐步成熟及其作用

- 这些问题起源于60年代并在80年代实用化，从理论学科的开始到对支持平凡的实际运用用了二十年左右的时间。
- 系统化方法和技术如结构化编程、面向对象设计等的发展也用了类似长的时间。
- 整个计算机工业与科学的历史只有四十年左右，尚有相当的理论处于研究前沿，软件体系结构就是其中之一。

科学与工程之间的相互作用

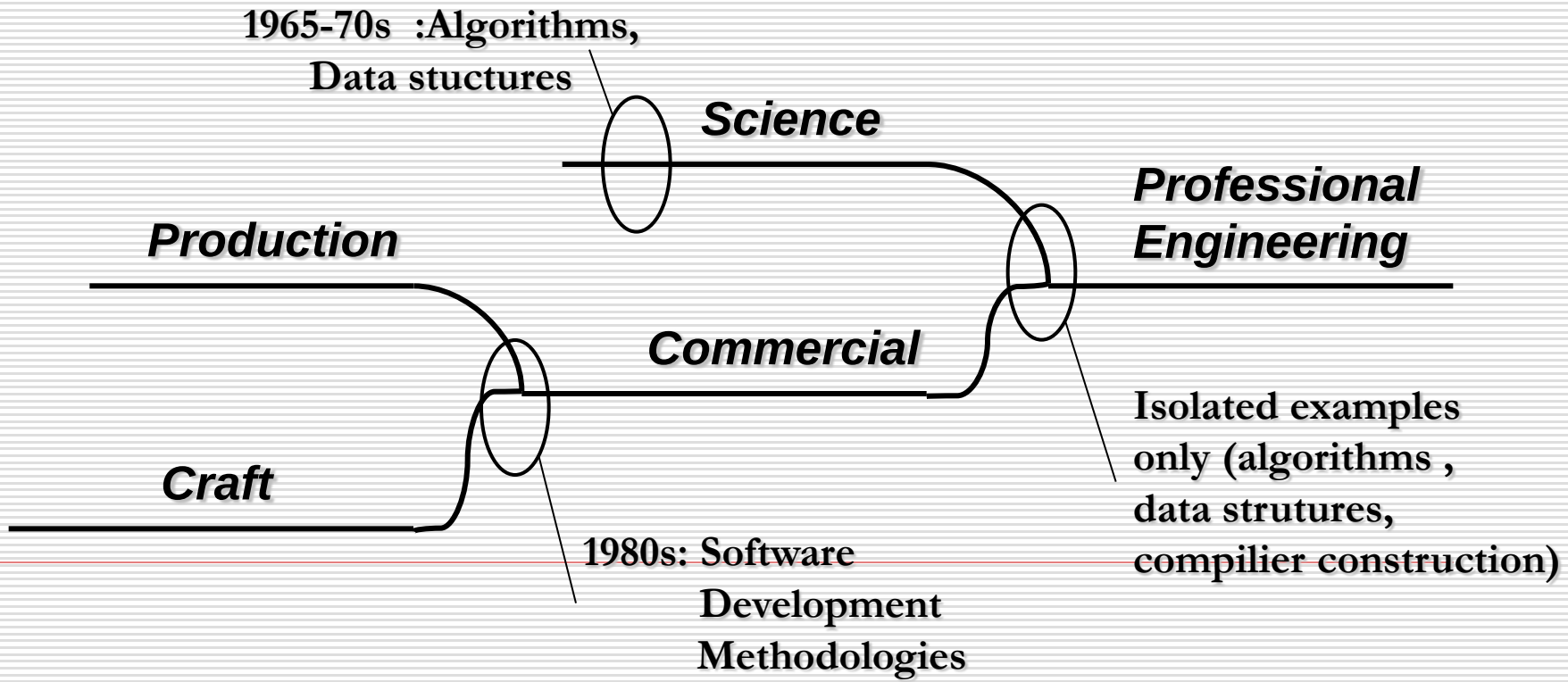


科学与工程之间的相互作用

- 例如，控制流的机器语言的发展
  - 50年代末60年代初，对循环与条件没有符号描述，只能用测试－分支指令独立的实现。
  - 一些有效的模式的出现，使得可以只用他们得到正确的结果。
  - 高级语言的设计者将它们提取出来设定特定的语法，如循环与分支。
  - 引入形式化语义描述从，理论上证明只需要几种循环与分支结构就可以达到所有的条件转移结果。
  - 现在人们使用while/if的时候不考虑底层的实现问题，认为这些结构的出现是理所当然的自然，即以成为一种例程。

工程的演化

- 当前的软件实践是在通向工程之路上吗？
  - 以计算机科学为基础的技术性的软件工程学是我们发展的目标。



抽象技术的历程

- 编程语言与工具发展过程中的一个特点
  - 软件设计者所使用的组件building block的抽象层和概念层在扩展，这是构成软件体系结构的工程基础之一。
  - 为此，我们考察计算机科学中抽象技术的发展过程。

抽象技术的历程

- 50年代数字计算机出现，软件用机器语言书写，指令与数据用二进制代码分别输入，必须明确描述内存分配。
- 最终发现内存分配与引用更新可以自动化，可以用符号命取代操作码和内存地址，于是产生汇编语言。为最早的软件抽象。
- 50年代后期，发现一些特定模式的执行是非常有用的，通常是算数表达式、过程调用、循环与条件等，于是用类似数学描述而非机器描述的符号体系描述它们，产生早期的高级语言，如Fortran。

抽象技术的历程

- 更高级的语言允许开发更复杂的程序，此时出现了使用数据的模式。高级语言主要开始面向程序员操作数据的需求，编译器同时进步，加强了对数据一致性等的检查，增强数据操作的安全性。

  - 数据类型机制出现
  - 并引入模块保护相关过程与数据结构
  - 将模块的描述与实现分开
  - 引入抽象数据类型。

抽象技术的历程

- 60年代晚期，优秀的程序员发现在正确的数据结构的基础上编程变得简单可靠。70年代，在此基础上通过一系列工作将这种直觉转化为了真正的理论：
  - 软件结构：通过封装基本操作符表示
  - 描述：用抽象模型或代数公理等数学化形式化表示
  - 语言：模块、作用域、用户定义类型
  - 结果完整性：用数据结构的不变量、前制条件与后制条件等保护
  - 组合规则：声明
  - 信息隐藏：保护不在描述中显式出现的属性

- 这些工作的结果是引入软件系统的设计层数据抽象概念抽象数据结构，同时人们意识到完整的模块应将数据表示、算法、描述、功能接口等集成描述。

抽象技术的历程

- 正如60年代优秀的程序员意识到数据结构的重要性，优秀的系统设计者开始意识到系统组织结构的重要性，由此往两个方面发展
  - 基于抽象数据结构的发展：数据驱动设计方法等以数据为中心的组织方法的提出
  - 基于模块抽象的发展：关注子系统如何交互构成大系统

抽象技术的历程

- 1975年提出模块互连语言MIL：
  - 模块导入导出资源，如类型定义、常量与变量、函数等；
  - 编译器通过模块间引用的类型检查保持一致性；
  - 70年代后期出现的MIL提供了软件构造、版本控制、系统簇等功能的支持；
  - 当前的MIL提供模块间通过过程调用、数据共享互连的方式，高层的互连结构隐含的实现。模块间遵循一定的约定，如命名匹配。
- 为建立真正可组合的系统，需要允许已有系统间弹性的高层互连，这种开放互联可以通过数据交换实现，如COBRA支持动态数据共享。

抽象技术的历程

- 大型软件系统需要分解机制使得系统设计实现可控，将一个系统划分为若干子系统，且通过了解子系统的属性了解系统的属性。传统上MIL/IDL(界面描述语言)用于描述（1）有良好接口的计算单元，（2）粘合这些单元的组合机制。

- 设计一个MIL/IDL的关键在于如何“粘合”，一般基于定义/使用的绑定（提供者与消费者），各个模块定义了一些可外部引用的功能并使用/请求其他模块定义的功能。通过寻找功能的定义点完成模块之间的绑定。

- 这样的结构的优点在于与现有的编程语言良好的结合、有利于编译器的构造与形式化推理（通过前制条件与后制条件）。

- 这些优点几乎是透明的，以至于很少有人怀疑其基本原则。然而的确存在一些严重的缺陷。

- 如，不能区分模块之间的“实现”与“交互”关系，前者对于理解一个模块是如何通过其他模块的功能构造的是必须的，而后者对于描述体系结构关系如计算单元之间的通信是必须的。

软件设计

面临的问题

- 问题 → 庞大、复杂化
- 环境条件 → 网络化、并行化、多操作系统环境。

软件的高效开发与维护

软件设计的必要性

- 设计
  - 低层次的软件代码设计
  - 软件体系结构的设计

- 小程序：从代码开始
- 大型软件：独立的软件体系结构设计阶段必不可少的。
  - 目标：良好的、易于维护的体系结构设计
  - 优点：

    降低设计风险、提高软件质量、保证开发进度

软件设计的目标

- 最终目标

  在时间和各类资源环境的限制下，最大限度地满足用户的需要。

能完成用户要求的软件就是好软件？

1. 便于维护和升级 → 模块化设计

- 目的：便于进行修改
  - 内部可变，接口不变（模块化设计思想的关键）

    软件的修改、维护、升级换代是必不可少的。

- 对开发工作的意义：分解模块、并行开发

  具体开发人员只需要了解自身的模块就可以了，对系统需求可以不必了解

- 便于修改

  将改动局限于某个模块内部，不致于使其蔓延开来

2.便于移植

- 移植

  不同环境下实现同样的功能需求。

  ps → pc，　　windows → linux，

  C/S → B/S

- 目标

  保证设计尽量可以重用。

  减少投入。

2.便于移植

- 软件的可移植性
  - 1）移植前后环境的差异程度
  - 2）软件描述语言的差异
  - 3）软件结构设计的优劣

e.g　C/S → B/S

3. 适应性

- 自扩展功能
  - 设计应该具有适应用户需求而变化的能力
  - 要求软件能够提供一部分功能和界面，运行用户按照需求独立定义和生成所需数据和功能
  - 设计和实现的代价较高，但可在相当程度上减缓用户需求的频繁变化。

- 示例：图书管理系统中自动报表、自动检索点的抽取
  - 自动报表：用户可以自行设计报表的样式，显示的内容等。
  - 自动检索点抽取：用户可以指定从书目的Marc码中抽取特定字段（如作者名、出版社、出版日期等）用于将来的查询。

- 现状

  通常仅局限于“满足用户需求”

4. 受到理性化的控制

- 理性化控制（Intellectual Control）
  - 一个正在发展的设计，不论结构多么复杂，如果都可以被其复杂正确性的人员深刻、全面地理解，就说明该设计是受到理性化控制的。

- 软件负责人员需要理解：
  - 理解系统的构成、部件的相互关联
  - 理解进行设计选择的理由和重要性
  - 修改可能造成的各种影响

- 如果设计者
  - 在实施前可以确定设计的正确性　　理性化控制
  - 仅靠实现后的测试判断核心设计的正确性　　非理性化控制

5. 概念的完整性

- 概念完整性（Conceptual Integrity）
  - 设计表现出整体的协调、一致和可预测性

- 具体表现在：
  - 内部结构的统一完整性

    有助于设计和维护人员进行正确的实施和良好的维护。

  - 外在表现界面的统一完整性

    便于用户学习和实践。

软件设计中出现的问题

- 设计对于需求的变化缺乏配合
  - 设计修改的必然性
    - 需求理解问题
    - 需求的变化
  - 用户与设计者 ←→ 不同的语言
    - 必须在需求分析阶段形成全面和成熟的系统结构

不要轻视需求分析阶段的工作量

软件设计中出现的问题

- 软件过程控制对于维持设计的正确性缺乏保障
  - 工程失控
  - 产生原因
    - 缺乏管理

      文档 + 评审　尽管麻烦但是有效
    - 缺乏全面的体系结构设计和设计规范说明

      缺乏整体系统的情况下，过于繁杂的实现细节往往超越人的控制能力。

软件设计中出现的问题

- 软件产品通常缺乏概念完整性
  - 对代码的随意修改
    - 设计者：

      不考虑整个系统要求，独立随意进行局部扩充或修改
    - 维护者：

      根据自身的认识，实施认为必要的修改。
  - 产生原因
    - 缺乏对系统整个结构和关系的认识
  - 如何保证完整性

    在设计初期建立一个完整的体系结构，使其成为理解、管理、控制整个系统开发的基础和核心出发点。

针对问题提出的软件设计思想

- 如何解决设计问题

  模块化设计

  → 面向对象设计

  → 软件体系结构

- 从宏观的层面上把握和控制软件复杂性

针对问题提出的软件设计思想

- 强调信息隐藏的单元概念
  - 面向对象思想的关键
- 操作与数据封装
  - 内部实现不是重点，功能才是关键
- 处理并发控制和分布系统
  - 对并发和分布的控制相当繁琐，以对象模式解决
- 基于模型的系统结构和设计方法
  - 在特定领域，从模型构造实际系统
- 明确软件体系结构的设计思想

软件体系结构的意义和目标

体系结构在软件开发中的意义

- 意义
  - 低成本、高回报
    - 体系结构的正确设计和选择为后续阶段（开发、集成、测试、维护）的成功提供了保证。
    - 此时的错误，意味着系统主体构成上的错误。
  - 对软件开发的便利
    - 为准确的程序设计规格说明提供了全面、可靠、可信的支持
    - 从而保证代码和规格说明的一致性
    - 进而保证设计实施的进度和正确性

体系结构在软件开发中的意义

- 软件体系结构的作用
  - 表达系统高层次的关系
  - 正确的体系结构是系统设计成功的关键
  - 帮助设计者解决复杂问题
  - 有助于进行复杂系统的高层次性能分析
  - 方便设计者之间、设计者与用户之间的交流
  - 方便进行系统维护、扩充与升级
- 设计文档的重要性
  - 特别在软件维护阶段

软件体系结构的目标

- 外向目标

  系统需求，建立满足终端用户需要（用户需要什么）

- 内向目标

  系统部件构成，满足后期设计者需要（如何使系统满足用户需求）

与其他软件设计活动的关系

- 软件开发：

  软件体系结构设计 + 软件设计

  - 起点不同
  - 用户←→结构分析者，结构分析者←→系统实现者
  - 处理的对象：系统部件/子系统←→变量,函数,过程等
  - 涉及的部门与组织
  - 体系结构设计的特点：贯串于整个工程活动过程中
    结构分析错误所造成的代价往往是很高的

软件体系结构的研究范畴

软件体系结构的研究范畴

- 研究范畴
  - 体系结构语言
  - 体系结构经验知识
  - 特殊应用领域体系结构框架
  - 基于体系结构的开发环境和工具
  - 体系结构形式化

风格、设计模式与框架

- 被公认的、多次使用的系统结构
  - 体系结构风格（Architecture Styles）
    e.g. 客户/服务器（Client / Server）
  - 设计模式（Design Patterns）
    e.g. 计数指针（Counted Pointer）
  - 框架（Framework）
    e.g. CAD系统

体系结构描述语言

- ADL（Architecture Description Language）
  - 对体系结构进行描述和规范的方法；对体系结构的理论认识的形式化描述。
  - 现状：

    关于什么是ADL？它应该描述体系结构的什么方面？这些基本问题尚未取得统一。
  - 特点：
    - 不同于需求语言

      需求语言描述问题；ADL描述问题的解决
    - 不同于建模语言

      建模语言注重于整体行为。ADL集中于部件的体现。
    - ADL提供抽象、结构和分析的能力

体系结构描述语言

- 现有ADL的共性
  - 提供形式化的句法和语义
  - 提供分布式系统建模
  - 除了通过一般的注释说明，很少为获得设计依据或历史提供支持
  - 把数据流和控制流作为互连机制处理
  - 有体现详细结构层次的能力

体系结构的形式化

- 形式化理论的重点

  通过形式化语言，对结构设计给出准确的描述

- 形式化语言分类

  1） 数学性语言

  Z语言、类属理论

  2） 专门语言

体系结构的工具和环境

- 为了支持基于体系结构的软件开发，需要体系结构描述语言和工程工具环境的支持

- 预期功能
  - 规范设计
  - 多视图表达
  - 性能分析
  - 分层细化
  - 代码生成
  - 动态行为描述

软件体系结构的现状

软件体系结构的现状

- 好的体系结构设计是决定系统成功的关键因素
  - 许多有用的体系结构模式如流水线、分层系统、客户端服务器模式等通常仅仅以习惯用语的方式理解，并特别地应用到不同系统中。
  - 从而软件系统设计者难于发掘系统体系结构中的共性因素，由此难于
    - 在不同的设计方案种作出选择
    - 将一般模式运用于特定领域和系统
    - 或与其他人交流自己的作品

软件体系结构的现状

- 人们意识到软件体系结构的重要性，从而在多方面开展研究
  - MIL/IDL模块交互语言与接口定义语言
  - 问题域相关的体系结构
  - 软件复用
  - 软件组织模式的条例化
  - 体系结构描述语言
  - 体系结构设计环境

软件体系结构的现状

- 主要关注的问题
  - 体系结构描述语言：用于交流并可通过工具分析的新的描述语言，如UML这样的通用语言
  - 体系结构经验的条例化：收集整理与重描述在软件开发过程中出现的各种体系结构性原则与模式
  - 特定领域的框架：为一些特定领域如航空控制系统、移动机器人、用户界面等整理体系结构框架，可用于简化该领域应用的设计与开发
  - 体系结构的形式化基础

软件体系结构的现状

- 现状
  - 缺乏系统统一的概念和坚实的理论基础
  - 缺乏工程知识的系统化和标准化
  - 缺乏形式化
  - 没有建立统一的体系结构的工程描述方法
  - 可应用的体系结构工程工具研究仍在实验室阶段

谢谢！