深圳大学
Shenzhen University

第2讲

软件体系结构概论

软件体系结构与设计模式

Software Architecture & Design Pattern

深圳大学计算机与软件学院

主要内容

- **1.4** 平凡设计与创新设计
- **1.5** 软件体系结构与抽象技术
- **1.6** 软件体系结构与软件设计
- **1.7** 软件体系结构的研究范畴

1.4 平凡设计与创新设计

- 平凡设计与创新设计之间的区别
- 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
- 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

1.4平凡设计与创新设计

- 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

1.4 平凡设计与创新设计

- 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

1.4 平凡设计与创新设计

- 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：

- 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；

1.4 平凡设计与创新设计

- 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）

1.4 平凡设计与创新设计

- 软件设计如何成为常规设计？

- 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）；

1.4 平凡设计与创新设计

- 底层复用向高层复用转换：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）

1.5 软件体系结构与抽象技术

- 编程语言与工具发展过程中，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。
- 回顾计算机科学中抽象技术的发展过程：
- 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。

1.5 软件体系结构与抽象技术

- 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。

- 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

1.5 软件体系结构与抽象技术

- 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。

1.5 软件体系结构与抽象技术

- 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

1.5 软件体系结构与抽象技术

- 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。

1.5软件体系结构与抽象技术

- 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。

1.6 软件体系结构与软件设计

- 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

1.6 软件体系结构与软件设计

- 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

1.6 软件体系结构与软件设计

- 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？

- 设计的5个目标：

（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

- 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。
- 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

（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**

1.6 体系结构在软件开发中的意义和目标

- 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

1.6 体系结构在软件开发中的意义和目标

- 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

1.6 体系结构在软件开发中的意义和目标

- **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）

1.7 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

1.7 软件体系结构的研究范畴

- 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）
- 框架（Framework）

谢谢