# Software Architecture

### - principle and practice

---

# 第一讲　软件体系结构概述

- 一、什么是软件体系结构（**Software Architecture**）
  - 1.软件体系结构是软件工程的一门新兴学科
    - 现在，计算机软件工业界面临着巨大的挑战：对于日益复杂的需求和运行环境，如何生产一个灵活、高效的软件系统。
    - 随着软件系统的规模和复杂性的增加，软件结构的设计和规范变得越来越重要。对于一个小的程序，人们可以集中考虑其算法选择和数据结构的设计，以及使用哪一种的代码设计语言，可是，对于一个大的系统，软件的结构就显得更重要了，比如，系统由哪些组件（component)构成、组件间的关系如何、每个组件应完成的功能、组件运行的物理位置、如何通讯、同步/异步、数据访问等，这些内容就体现了软件系统结构的思想。
  - 2.在系统科学中，系统(system)的定义
    - 系统概念是系统理论的最基本的概念，它浓缩和概括了系统理论的最基本内容，然而，由于研究领域的不同、应用对象和理解角度的不同，对系统概念的定义也有不同，尚未有一个统一的定义。
    - 我国著名科学家钱学森认为“系统即由相互作用和相互依赖的若干部分(要素)结合成的具有特定功能的有机的整体，而且这个系统本身又是它所从属的一个更大系统的组成部分”
    - 系统的特性
      - 集合性
        - 系统由两个或两个以上的可以区别的相互区别的要素(对象)组成。
      - 相关性
        - 系统中，各个要素之间具有相互依赖、 相互作用的关系。
      - 结构性
        - 系统中，各个要素不是简单的排列，而是有一定的组成形式即结构。相同的几个要素，如果组织的结构不同，将构成不同的系统。
      - 整体性
        - 系统中各个要素根据特定的统一要求，共同协作，对外形成个整体。
      - 功能性
        - 各个要素完成特定的功能，它们的相互作用完成系统的功能，但系统的功能并不是它的各部分的功能的线性和，具有“整体大于各部分之和”
      - 环境适应性
        - 任何一个系统都存在于一定的环境之中，受外界的影响，具有开放性。
    - 系统的体系结构
      - 系统中各要素的组织方式和相互作用方式。
      - 常用的有整体性结构、层次结构等
  - 3.软件体系结构的定义（Mary Shaw, David Garlan)
    - 一个软件系统的体系结构是指构成这个系统的计算部件（computational components)、部件间的相互作用关系(interactions)。
    - 部件可以是客户（clients)、服务器（servers)、数据库（databases)、过滤器(filters) 、层等。
    - 部件间的相互作用关系（即连接器）可以是过程调用（procedure call)、客户/服务器、同步/异步、管道流(piped stream)等.
  - 4.软件设计的层次性
    - 结构级
      - 包括与部件相关联的系统的总体性能，部件是指模块，模块的相互关联通过多种方法处理。
    - 代码级
      - 包括算法和数据结构的选择，部件指程序语言中的数值、字符、指针等，相互关联是程序中的各种操作，合成如记录、数组、过程等。
    - 执行级
      - 包括存储器的映射、数据格式设置、堆栈和寄存器的分配等，部件是指硬件提供的位(bit)模式,相互关联由代码描述。
    - 以前，软件开发人员注意力主要集中在程序语言的层次上，现在，软件的代码和执行层次的问题已经得到很好的解决，而对结构级的理解一直都还停留在直觉和经验上，尽管一个软件系统通常都有文字和图表的说明，但所使用的句法和所表达语义的解释从来没有得到统一。而软件体系结构将在软件的较高层研究软件系统的部件组成和部件间的关系。

- 5.体系结构的类别
  - 服务于软件开发的不同阶段，体系结构可分为
  - 概略型
    - 是上层宏观结构的描述，反映系统最上层的部件和连接关系
  - 需求型
    - 对概略型体系结构的深入表达，以满足用户功能和非功能的需求。
  - 设计型
    - 从设计实现的角度对需求结构的更深层的描述，设计系统的各个部件，描述各部件的连接关系。这是软件系统结构的主要类别。
- 6.体系结构的重要性
  - 体系结构的重要性在于它决定一个系统的主体结构、基本功能和宏观特性，是整个软件设计成功的基础，其重要性表现为在项目的：
  - 规划阶段
    - 粗略的体系结构是进行项目可行性研究、工程复杂性分析、工程进度计划、投资规模预算、风险预测等的重要依据。
  - 需求阶段
    - 在需求分析阶段，需要从项目需求从发，建立更深入的体系结构，这时体系结构成为开发商与用户之间进行需求交互的表达形式，也是交互所产生的结果，通过它，可以准确地表达用户的需求，以及出对应需求的解决方案，并考察系统的各项性能。
  - 设计阶段
    - 需要从实现的角度，对体系结构进行更深入的分解和描述。部件的组成、各部件的功能、部件的位置、部件间的连接关系选择等的描述。
  - 实施阶段
    - 体系结构的层次和部件是建立人员的组织、分工、协调开发人员等的依据。
  - 测评阶段
    - 体系结构是系统性能测试和评价的依据。
  - 维护阶段
    - 对软件的任何扩充和修改都需要在体系结构的指导下进行，以维持整体设计的合理性和正确性以及性能的可分析性，并为维护升级和复杂性和代价分析提供依据。

## 二、模块及其设计

### 1.什么是模块

- 定义
  - 模块（module)是由一个或多个相邻的程序语句组成的集合。每个模块有一个名字，系统的其他部分可以通过这个名字来调用该模块。
  - 模块是一个独立的代码，能够按过程、函数或方法调用方式调用它。
  - 宏不是模块，过程、函数是模块，对象是模块，对象内的方法也是模块。
  - 一个软件产品可以分解成一些较小的模块。
  - 模块的重要性
    - 解决复杂问题的一种有效方法
      - Miller法则：一个人任何时候只能将注意力集中在7±2个知识块上。
      - 软件开发时，人们的大脑需要在一段时间内集中的知识块数往往远远多于7个。
      - 对一个复杂的问题解决转化为对若干个更小问题的解决来实现。
    - 集体分工协作的前提
      - 将一个产品分解成几个相对独立的模块，这些模块分配由几个不同的小组开发。
    - 产品维护的保障
      - 模块间的相对独立性使得修改其中一个模块的内部代码或数据结构不影响其他模块。不仅为运行时的维护提供了可行性，还减少维护的费用。

### 2.模块内聚性（module cohesion)

- 含义
  - 模块内聚性是指一个模块内相互作用的程度
- 内聚性的层次

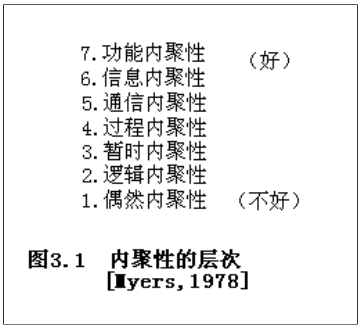


图3.1 内聚性的层次
[Myers, 1978]

- 偶然内聚性
  - 如果一个模块执行多个完全不相关的动作，那么这个模块就有偶然内聚性内。例如，一个模块在一个列表中增加一个新的项或删除一个指定的项。
  - 具有偶然内聚性的模块有两个严重的缺点：一是在改正性维护和完善性维护方面，这些模块降低了产品的可维护性；二是这些模块不能重用。
  - 造成的原因：“每个模块由35 - 50个可执行语句组成。”，将两个或多个不相关的小模块不得不组合在一起，从而产生一个具有偶然内聚性的大模块。另外，从管理角度认为在太大的模块内被分割出来的部分将来要组合在一起。
- 逻辑内聚性
  - 当一个模块执行一系列相关的动作,且其中一个动作是作为其他动作选择模块，称其有逻辑内聚性。例如，一个执行对主文件记录进行插入编辑、删除编辑和修改编辑操作的模块；
  - 一个模块有逻辑内聚性会带来两个问题：其一，接口部分难于理解，其二，多个动作的代码可能会缠绕在一起，从而导致严重的维护问题，甚至难于在其他产品中重用这样的模块。
- 暂时内聚性
  - 当一个模块执行在时间上相关的一系列动作时，称其具有暂时内聚性。例如，在一个执行销售管理的初始化模块中，其动作有：打开一个输入文件，一个输出文件、一个处理文件，初始化销售地区表，并读入输入文件的第一条记录和处理文件的第一条记录。这个模块的各个动作之间的相互关系是弱的，但是却与其他模块中的动作有更强的联系，就销售地区表来说，它在这个模块中初始化，而更新销售地区表和打印销售地区表这样动作是在其他模块中，因此，如果销售地区表的结构改变了，则若干模块将被改变，这样不仅可能有回归的错误（由于对产品的明显不相关的部分的改变而引起的错误），而且如果被影响的模块的数量很大时，则很可能会忽略一二个模块。而较好的作法是将有关销售地区表的所有操作放入一个模块。
  - 暂时内聚性的模块的缺点：一是降低了产品的可维护性；二是这些模块难以在其他产品中重用。
- 过程内聚性
  - 一个模块有过程内聚性，是指其执行与产品的执行序列相关的一系列动作。比如，从数据库中读出部分数据，然后更新日志文件中的维护记录。过程中的动作有一定的联系，但这种关系也还是弱的。
  - 这种模块比暂时内聚性模块有好些，但仍难以重用。
- 通信内聚性
  - 一个模块有通信内聚性，是指其执行与产品的执行序列相关的一系列动作，并且所有动作在相同数据上执行。
  - 这种模块比过程内聚性模块有好些因为其动作有紧密的联系，但仍难以重用。
- 信息内聚性
  - 如果一个模块执行一系列动作，每一动作有自己的入口点，每一个动作有自己的代码，所有的动作在相同的数据结构上执行，这样的模块称其为信息内聚性模块。

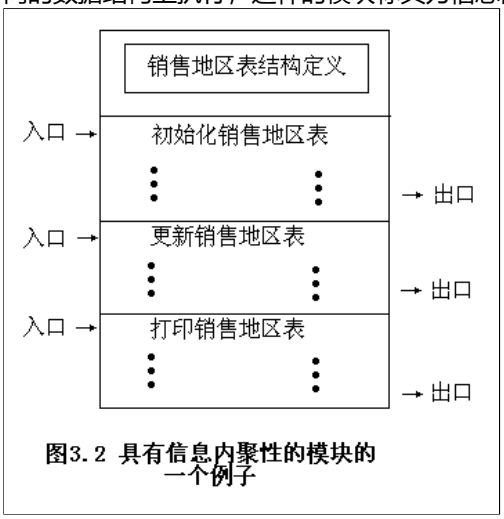


图3.2 具有信息内聚性的模块的一个例子

  - 对于面向对象的范型来说，信息内聚性是最理想的。信息内聚性的模块本质上是抽象数据类型的实现，而对象本质上就是抽象数据类型的一个实例。
  - 拥有抽象数据结构的优点。

- 功能内聚性
  - 一个只执行一个动作或只完成单个目标的模块有功能内聚性。
  - 一个有功能内聚性的模块可能经常被重用。因为它执行的那个动作经常需要在其他产品中执行。一个经过适当设计、彻底测试的并且具有良好文档的内聚性模块对一个软件组织来说是一个有价值的资产，应该尽可能多地重用。
  - 具有功能内聚性的模块也比较容易维护。首先，功能内聚有利于错误隔离；另外因为比普通的模块更容易理解从而简化了维护；同时还便于扩充（简单也抛弃旧模块用一个新模块来代替）。

◦ 3.模块耦合(module coupling)
- 含义
  - 模块耦合是指模块间的相互作用程度。
- 耦合的级别

5. 内容耦合（不好）
4. 共用耦合
3. 控制耦合
2. 特征耦合
1. 数据耦合 （好）

图3.3 耦合的级别

- 内容耦合
  - 如果一个模块p直接引用另一个模块q的内容，则模块p和q是内容耦合的。比如，模块p分支转向模块q的局部标号。
  - 在产品中，内容耦合是危险的，因为它们是不可分割的，模块q的改变，都需要对模块p进行相应的改变，并且，在新产品中，如果不重用模块q,则不可能重用模块p。
- 共用耦合
  - 如果两个模块都能访问同一个全局变量，则称它们为共用耦合。比如

```
while(global_varible==0)
{
 if(argument_xyz>25)
  module3();
 else
  module4();
}
```

  - 缺点
    - 代码不可读，与结构化编程的精神相矛盾；
    - 维护困难。如果在一个模块内，对某个全局变量的声明做了维护性修改，那么访问这个全局变量的每一个模块都必须修改，而且，所有的修改都必须是一致的。
    - 难以重用。因为重用这类模块时，必须提供相同的全局变量名。
    - 数据无法控制。作为共用模块的后果，一个模块也许会被它本身之外的数据改变，使得控制数据访问的努力变得无效。
- 控制耦合
  - 如果一个模块传递一个控制元素给另一个模块，则称这两个模块是控制耦合的。即一个模块明确地控制另一个模块的逻辑。

| | |
|---|---|
| model p<br>{<br>return integer_var;<br>} | module p<br>{ ...<br>call module q(0);<br>...<br>} |
| model q<br>{ ...<br>x=call module p; | module q(argument x)<br>{ ... |

```
switch(x){
 case 0:
   …
 case 1:
   …
 }
}
```

```
switch(x){
 case 0:
   …
 case 1:
   …
 }
 …
}
```

- 控制耦合所带来的问题是：两个模块不是相互独立的 - - 被调用模块必须知道模块p的内部结构和逻辑，降低了重用的可能性；另外，控制耦合一般是与逻辑内聚性的模块的有关，逻辑内聚性的问题也在控制耦合中出现。
- 特征耦合
  - 如果把一个数据结构当作参数传递，而被调用的模块只在数据结构的个别元素上操作，则称两个模块是特征耦合的。
  - 导致数据无法控制。
- 数据耦合
  - 如果两个模块的所有参数是同一类数据项，则称它们是数据耦合的，也一就是每一个参数要么是简单变量，要么是数据结构，而当是后者时，被调用的模块要使用这个数据结构中的所有元素。
- 一个好的模块设计应该使模块具有高内聚性和低耦合性。

◦ 4.重用

- 为了使一个产品在另一个编译器/硬件/操作系统上运行，如果修改这个产品比从头编写代码更容易的话，那么这个产品就是可移植的。而重用（reused)是指利用一个产品的构件以使开发另一个具有不同性能的产品更容易。可重用的构件不一定是一个模块或一个代码框架，它可能是一个设计、一本手册的一部分、一组测试数据或最后期限和费用的估计。
- 重用的两种类型
  - 偶然重用
    - 如果一个新产品的开发人员意识到以前所开发的产品中有一个构件在新产品中可重用，那么这种重用即为偶然重用。
  - 计划重用
    - 如果利用那些专门为在将来产品中重用而构件的软件构件，则称为计划重用。
  - 计划重用比偶然重用有一个潜在的优点，这就是，专门为在将来的产品中使用而构造的那个构件更容易于重用，而且重用也更加安全。因为这样的构件一般都有健全的文档，并做过全面的测试，另外，它们通常有统一的风格，从而易于维护。
  - 但另一方面在一个公司内实现计划的代价可能是很高的。对一个软件构件进行规格说明、设计、实现、测试和编制文档要花很多的时间，然而，这样一个构件是不是能重用，所投资的成本能否回收不无保证。
- 关于构件重用
  - 在计算机刚问世时，没有什么东西是可以重用的。每当开发一个产品时，所有项目都是从头开始构造的。然而不久为后，人们意识到这是相当大的工作浪费，于是人们构造了子程序库。这样，程序员在需要时就可直接调用这些以前编写好的例程，这些子程序库越来越成熟，并开始出现运行时（run-time)的支持程序。
  - 重用可以节省时间，缩短产品的开发期限同，使软件开发公司更有竞争力。开发人员既可以重用自己的例程，也可以重用各种类库或API，从而节省了大量的时间。相反，如果一个软件产品要花经费4年的时间才能进入市场，而一个竞争产品只用2年就交付使用，那么，不管它的质量有多高，它的销路也不会太好。开发过程的时间期限在市场经济中是至关重要的，如果产品在时间方面没有竞争优势，那么谈论怎样才能生产一个好的产品是不切实际的。
  - 软件重用是一项诱人的技术。毕竟，如果可以重用一个现有的构件，就不必再设计、实现、测试这个构件了。
- 重用的障碍
  - 据统计，对于任意软件产品来说，平均只有15%是完全服务于原始的产品目的的，产品的另外85%在理论上可以进行标准化，并在将来的产品中重用。在实际中，通常只能达到40%的重用率并且利用重用来缩短开发周期的组织也很少。因为重用有许多的障碍。
  - 自负
    - 从太多的软件专业人员宁愿从头重写一个程序而不愿重用一个由其他人编写的程序。他们认为，不是他们亲自编写的程序不可能是好程序。
  - 经济利益
    - 一些开发人员尽力避免编写那些太通用的例程，唯恐使自己失业。当然，从每个软件组织实际上有大量的积压任务的情况来看，这种担心是毫无根据的。
  - 检索
    - 一个组织可能有几十万个潜在的可重用的构件，为了提高检索效率，如何存储这些构件?
  - 代价

- 重用构件的成本：制作重用构件的成本、重用的成本、定义和实现一个可重用过程的成本。有统计表明，制作一个重用构件在理想情况下可其成本只增加11%，一般情况下增加60%，而有的重用可能会增加200%甚至480%。
- 版权
  - 根据客户和软件开发公司之间的合同，软件产品是属于客户的，当软件开发人员为另一个客户开发一个新产品时，如果重用了另一个客户产品中的一个构件，本质上是一个侵犯第一个客户的版权。

# 三、软件工程（Software Engineering)

## 1.软件危机（software crisis)

### 计算机硬件技术的发展迅速

- 在短短的几十年中，计算机技术成为了现代社会的高科技的核心，其中硬件的发展是其他领域不可比拟的。中央处理器功能、存储器的容量、集成工艺的提高、新兴材料的研制、网络等变革，使得计算机很快从实验室走向应用，进入各行各业。计算机是社会信息化的基础。
- 然而，在软件技术方面，虽然也以巨大的速度发展，但比起计算机的硬件发展，就是微不足道的了。特别是在应用领域，许多企事业单位、机关　团体中的计算机，其性能远远没有得到充分的发挥。
- 随着计算机硬件的飞速发展，对计算机软件的功能、结构和复杂性提出了更高的需求，在软件的设计中，软件的局部和整体系统的结构方面，已经越来越显出其重要性，甚至超过了软件算法和数据结构这些常规软件设计的概念。软件体系结构概念的提出和应用，说明了软件设计技术在高层次上的发展并走向成熟。

### 软件危机

- 解决软件的整体质量较低，以及最后期限和费用没有得到满足。由于这个危机长期以来仍然困扰着我们，所以也称为软件萧条（software depression)

- 开发大型软件过程中，难以汇集参与人员的设计理念然后提供给使用者一致的设计概念(conceptual integrity)，因而导致软件的高度复杂性，使得大型软件系统往往会进度落后、成本暴涨及错误百出，就是所谓的软件危机(software crisis)。
- The Mythical Man-Month
  - 二十多年前(1975)，IBM大型电脑之父——Frederick P. Brooks 出版的一本书。
  - 人月（man-month）"：熟悉软件项目管理的人员都清楚，人们常常根据人月来估计工作量（并相应收费），比如一个项目五人两月完成，那么总工作量就是10人月。
  - 称之为"神话"(Mythical)，其用意也并非完全否定作为计量方法的人月，而是要理清这个概念中隐含的种种错觉。文中论点主要包括：
    - 1. 人/月之间不能换算，换言之，两人做五个月完成，不等于说五人做两个月就能完成；
    - 2.在项目后期增加人手，只能使工期进一步推迟；
    - 3.项目越大，单位工作需要的人月越多。
  - 著名的Brooks法则
    - Adding manpower to a late software project makes it later(对于进度已落后的软件开发计划而言，若再增加人力，只会让其更加落后。)
    - "人月"概念可以线性化的神话：无论是开发人员的人数上，还是工作量本身上的变化，都可能导致最终完成时间的非线性变化。
- No Silver Bullet
  - 1986，Brooks发表了一篇著名的论文——"No Silver Bullet: Essence and Accidents of Software Engineering"。他断言：在10年内无法找到解决软件危机的根本方法（银弹）(There will be no silver bullet within ten years)。
  - Brooks认为软体专家所找到的各种方法皆舍本逐末，解决不了软体的根本困难——即概念性结构(conceptual structure)的复杂，无法达到概念完整性。
  - 软件开发的困难来自两个方面：本质的和偶然的。本质的困难是软件开发本身所固有的，无法用任何方式取消的，而偶然的困难是其中的非本质因素，可以通过引入新工具、方法论或管理模式来消除。关键在于，只要本质的困难在软件开发中消耗百分之十以上的工作量，则即使全部消除偶然困难也不可能使生产率提高10倍。
  - 软件的本质
    - 复杂性(complexity)——“复杂”是软件的根本特性，可能来自于程序员之间的沟通不良，而产生结构错误或时间延误；也可能因为人们无法完全掌握程序的各种可能状态；也可能来自新增功能时而引发的副作用等等。
    - 一致性(conformity)——大型软件开发中，各小系统之界面常会不一致，而且易于因时间和环境的演变而更加不一致。
    - 易变性(changability) ——软件的所处环境常是由人群、法律、硬体设备及应用领域等各因素融合而成的文化环境，这些因素皆会快速变化。
    - 不可见性(invisibility)——软件是看不见的，既使利用图示方法，也无法充分表现其结构，使得人们心智上的沟通面临极大的困难。
- There Is a Silver Bullet
  - 1990年，曾首先提出"Software IC"名词的OO大师——Brad Cox针对Brooks的观点而发表了一篇重要文章——"There Is a Silver Bullet"。说明他找到了尚方宝剑——即有些经济上的有利诱因会促使人类社会中的文化改变(culture change)，人们会乐于去制造类似硬体晶片(IC)般的软件组件(software component)，将组件内的复杂结构包装得完美，使得组件简单易用，由这些组件整合而成的大型软件，自然简单易用；软件危机于是化解了。

### 一个广为流传的故事 - - 零欠款事件

- 有一次，一个经理收到一张计算机打印出的账单，上面标出他欠了0.00美元。这位经理在和朋友们嘲笑“计算机傻瓜”之后，将账单扔掉了。一个月之后，又来了一张类似的帐单，这一次标明他已欠款30天。其后又来了第3次帐单。第4次帐单

在一个月之后又来了，帐单上提醒他，如果他不一次付清0.00美元的欠款，将可能承担法律责任。 第5次帐单标明他已欠款120天，上面没有其他任何提示，只有一条粗暴而直率的消息威胁他，如果不立即付款，将对他采取所有可能的法律行为。由于害怕他的公司在这台发疯的计算机控制下要付贷款利息，咨询了一位软件工程师朋友，这位朋友笑着告诉经理，只要发送一张0.00美元的支票即可。为一招果然有效，几天之后，经理收到了一张0.00美元的收据。

- 2.软件工程的复杂性
  - 建筑工程的经验对软件工程的启发
    - 桥墩有时会坍塌，但出现的次数远远小于操作系统崩溃的次数。
    - 两种故障的主要区别：土木工程领域和软件领域对崩溃事件的理解态度不同。当桥墩坍塌时人们几乎总是要对桥墩进行重新设计和重新建造，因为桥墩的坍塌说明该桥的最初设计有问题，这将威胁行人的安全，所以必须对设计作大幅度的改动，此外，桥墩坍塌后，几乎桥的所有结构被毁掉，所以唯一合理的做法是将桥墩残留的部分拆除再重新建造。更进一步地，其他所有相同设计的桥都必须仔细考虑，在最坏的情况下，要拆除重新建造。相比之下，操作系统的一次崩溃很少被认真考虑，人们很少立即对它的设计进行考察。当出现操作系统崩溃时，人们很可能只会重新启动系统，希望引起崩溃的环境设计不再重现。在多数情况下，没有去分析关于崩溃原因的证据，而且操作系统的崩溃引起的破坏通常中微不足道的。
    - 也许当软件工程师们以土木工程师们对待桥墩坍塌那样认真的态度来对待操作系统故障时，这种区别就会缩小。当然，人类关于桥梁的设计毕竟经历了几千年的历史，而设计操作系统的经历只不过短短50多年。随着经验的积累，将一定会像理解桥一样充分地理解操作系统，构造出无故障的操作系统。
  - 软件工程的复杂性
    - 软件在执行时处于离散状态。
      - 主存储器中一个比特位（bit)的改变就会引起软件执行的状态改变，而这种状态总数是巨大的，在设计初期无法完全测试。
    - 软件运行环境具有不可再现性。
      - 在采取多道程序设计后，一道程序在同一台机器的多次执行，其运行环境（比如，程序的代码被装入的主存空间的位置、程序执行过程的速度等）几乎不可重现。
    - 硬件的复杂性
      - 软件的功能要由硬件来实现，硬件的结构复杂多样性，软件测试的难度。
- 3.软件工程的内容
  - 软件工程是以软件系统为对象，合理地、创造性地采用软件系统所需的思想、理论、技术、方法对软系统进行研究分析、设计开发和服务，使软件最大限度地满足需求。
  - 软件工程的生命周期
    - 需求分析
    - 规格说明
    - 计划
    - 设计
    - 实现
    - 集成
    - 维护
    - 终结
-

## 四、软件体系结构的意义与目标

### 1.软件体系结构的意义

- 软件体系统结构是软件开发过程的初期产品，为以后的开发、集成、测试、维护等阶段提供保证。
- 与软件过程的其他设计活动相比，体系结构的活动成本和代价要要低得多。
- 软件体系结构的正确有效，给软件开发带来极大的便利。
- 这些在体系结构的重要性中有了更详细的说明。

### 2.软件体系结构的目标

- 主要目标：建立一个一致的系统及其部件的视图，并以提供能夠满足终端用户和后期设计需要的结构形式。
- 外向目标：建立满足终端用户要求的系统需求。了解用户需要系统应该做些什么，扩展或细化结构，澄清模糊，提高一致性。
- 内向目标：如何使系统满足用户需求。为些需要建立哪些软件模块、分析它们的结构、相互间的关系和规范。正是对这些软件上层部件的及其关系的规划，为以后的系统设计和实施活动提供基础和依据

## 五、软件体系结构的研究现状

### 1.软件体系统结构的发展

- 程序抽象
  - 20世纪50年代：汇编语言对机器语言的第一层抽象；
  - 20世纪60 - 70年代：高级语言对程序描述的抽象，算法和数据结构的概念从程序中获得抽象，软件设计理论获得根本发展。
  - 20世纪80年代：建立在抽象数据类型上的面向对象技术和理论。

        - 20世纪90年代：面向对象技术的广泛应用：OLE、动态链接、ODBC、组件、RPC、CORBA、浏览器等，面向网络、跨平的分布式环境等。
    - 软件工程
        - 软件工程（1967年）是对软件工程设计的方法、技术和管理等方面的研究，为了实现软件的工程设计，要在独立于程序语言之外建立软件构成的表达，就软件所解决的问题建立概念的关系和模型。
        - 20世纪70年代开始，提出和发展了软件的结构分析和设计方法，数据字典、数据流成为程序结构的主要描述手段，E - R图成了主要的信息概念模型甚至延用至今。
        - 80年代，软件工程设计方法了面向对象的分析和设计。
        - 90年代，面向对象方法的广泛发展，提出了诸如UML等多种面向对象的概念模型，并在软件工程中获得应用。
        - 从某种意义看，这些模型和表达也是软件体系结构的描述方法，只是它们更多地从信息处理的角度建立起来的。
    - 体系结构
        - 人们对软件结构的关心，在20世纪50年代就开始了，在机器语言的控制流概念方面，人们尚不知道循环和条件结构描述，而是通过测试分支指令来实现这些结构。
        - 高级语言的设计者更是认识到结构的用处，建立一套完整的程序结构描述，如循环、条件、过程调用，并通过开工形式化研究获得进一步的确认。现在几乎没有人认为还需要发明新的循环、条件或其他程序描述结构。
        - 直到90年代，软件体系结构才开始受到全面的关注，并形成了新的软件技术和工程的研究热点。
- 2.软件体系统结构的研究现状
    - 一个好的结构设计是决定一个软件系统成功的主要因素。
    - 体系统结构的研究现状其现状具体表现在：
        - 缺乏系统统一的概念和坚实的理论基础
            - 软件体系结构已经提出，从整体上把握软件设计的重要性，并在结构的部件、部件关系（连接）上取得一致的认识。但部件和关系的描述、体系结构的基础、体系结构与其他软件研究的关系、体系结构与需求分析的关系都没有取得全面统一的认识。
        - 缺乏工程知识的系统化和标准化
            - 现有的涉及体系结构部件和连接器标准标准化的规范，都是来自特别应用问题或领域的，没有来处建立在软件体系结构总体认识上。例如，许多有用的结构范例，如管道、层次、C/S等，还只能应用于特定的系统，软件系统的设计者还没有设计出一些公用的系统结构的原理供人们选择。
        - 缺乏形式化，没有建立统一的体系结构的工程描述方法
            - 在传统的设计中，人们很早就使用系统框图和非形式描述来表达软件结构，可是这些描述都太含糊了、因人、因事而异，只凭经验、直觉或个人爱好，没有一个标准，不便于交流。
            - 体系结构的形式化研究已经受到极大的关注并取得一定的成果，但它们都只是多某全侧面进行的，尚缺乏全面的适用性。
            - 在这方面Microsoft公司推出的.NET的公共语言规范（Common Language Specification)技术可能是一个重要发展。
    - 目前，软件体系结构已经成为软件工程的从业者一个重要研究领域。
    - 软件体系结构的研究内容
        - 风格(styles)
            - 研究部件间的相关关系，及合成和设计规则
        - 设计模式(design patterns)
            - 建立在结构化和面向对象的基础上，设计人员积累大量的经验，发现并抽象在众多的应用中普遍存在的软件的结构及其关系，以此为模板实现软件重用
        - 结构描述语言（ADL）
            - 研究各种表达软件构成的描述形式，作为软件设计的结构表达的一些规范。

返回

==========