# 软件设计

### 历史

* 1. 着重点

转移到软件体系结构和可用于实现软件体系结构的设计模式。

* 1. 共同特征
     1. 一种用于将分析模型变换到设计模型的表示机制；
     2. 用于表示功能件构件及其接口的符号体系；
     3. 用于求精（优化）和划分的启发机制；

### 目标

* 1. **根据**“目标系统”的**逻辑模型确定**“目标系统”的**物理模型**，概括地描述系统如何实现用户所提出来的功能和性能等方面的需求
     1. 需要设计软件结构、处理方式、数据结构、数据存储、界面和可靠性
  2. 软件设计是**软件开发的基础和依据**，是软件工程过程中的**技术核心——时间占比很大**
  3. 取得主观上的最佳方案

### 过程

* 1. **将软件需求转换为功能模型、数据模型、行为模型**
  2. 分为
     1. **概要设计**：描绘总体框架
     2. **详细设计**：对结构进行细化，得到各功能模块的详细数据结构和算法，使得功能模块在细节上接近源程序的软件设计模型

### 软件的概要设计

* 1. 最终结果：生成概要设计规格说明书
  2. 需要完成的工作：
     1. 制定设计规范：
        1. 阅读和理解需求说明书、确定预算和现有技术条件下，能否满足用户需求；
        2. 确定功能的优先顺序，根据目标确定合适的分析方法；
        3. 规定文档标准
     2. 软件系统结构的总体设计：
        1. 确定基本的体系结构，确定活动和存储结构
        2. 确定模块--->确定模块功能。模块和功能要实现对应
        3. 确定模块之间的关联关系，结构是什么，为每个模块确定算法和评估算法
     3. 处理方式设计
     4. 数据结构设计：输入、输出、算法的数据结构
     5. 可靠性设计：容错性能、
     6. 界面设计：需求的直接表达
     7. 编写软件概要设计说明书：概要设计的最终结果
     8. 概要设计评审

### 软件详细设计

* 1. 需要完成的工作：
     1. 确定软件各个功能模块内的算法以及个功能模块的内部数据组织
     2. 选定某种表达形式来描述各种算法
     3. 编写软件详细设计说明书
     4. 进行详细设计的评审

### 软件设计模型

* 1. 软件设计既是过程又是模型
  2. 模型构成：
     1. **静态结构设计**：由**功能结构**和**数据结构**组成，展现软件系统能够满足所有需求的框架结构
     2. **动态结构设计**：表示功能响应用户请求时处理数据的**过程或条件**，用于进一步解释软件结构中**各功能之间如何协调工作**
  3. **评估设计过程的技术原则**：
     1. 设计过程应该是**可追踪和可回溯**的
     2. 实现分析模型中描述的所有**显式需求**
     3. 满足用户希望的所有**隐式需求**【安全性、功能性错误、数据安全、完整性要求】
     4. **设计说明文档**：可读的、可理解的、易于编程、测试、维护
  4. **评估设计模型的技术原则**
     1. **设计模型应该展现软件的全貌，**包括从实现角度可看到的数据、功能、行为。
     2. 设计模型应该是一个**分层结构**。该结构：
        1. 使用可识别的设计模式搭建系统结构；
        2. 由具备良好设计特征的构件构成；——模块设计
        3. **可以用演化的方式实现；——容纳变化**
     3. 设计应当**模块化**，即应当建立具有独立功能特征的构件。——软件设计的模块化和问题的分解相互呼应。
     4. 设计应当建立能够**降低**模块与外部环境之间**复杂连接的接口**。——接口过多不利于系统稳定性、不利于开发和维护
     5. 设计应当根据将要实现的对象和数据**导出合适的数据结构**。

### 软件模块化

* + 1. **模块的定义**：整个软件可被划分成若干个可单独命名且可编址组成部分，这些部分称之为模块。
    2. **模块的基本属性：**功能、逻辑、状态
    3. **模块的表示：**
       1. 外部特征：模块名、参数表、给程序甚至整个系统造成的影响
       2. 内部特征：完成功能的程序代码和仅供该模块内部使用的数据。
    4. 模块划分：
       1. **难点在于合理地划分模块——数量？**
          1. 如果模块是相互独立的（模块独立性），当模块变得越小，每个模块花费的工作量越低
          2. 但当模块数增加时，模块间的联系也随之增加（模块的耦合度），把这些模块联接起来的工作量也随之增加。
       2. **信息隐藏：——对应设计标准中的“可理解性、连续性、模块保护”**
          1. 每个模块的实现细节对于其它模块来说应该是隐蔽的。模块中所包含的信息（包括数据和过程）不允许其它不需要这些信息的模块使用。
          2. 其最大的好处就是**使得模块的修改和软件的维护所造成的影响可以局限在一个或几个模块范围内**。
          3. **模块之间的关系能够达到信息隐藏，就可以认为是合理的模块划分。**
  1. **模块设计的标准**
     1. 模块可分解性：可将系统按“问题／子问题”的分解原则分解成系统的模块层次结构;
     2. 模块可组装性：可利用已有的设计构件组装成新系统，不必一切从头开始。
     3. 模块可理解性：一个模块可不参考其他模块而被理解；
     4. 模块连续性：对软件需求的一些微小变更只导致对某个模块的修改而整个系统不用大动;
     5. 模块保护：将模块内出现异常情况的影响范围限制在模块
  2. **抽象化**
     1. 在软件设计过程中**按照自顶向下的设计原则，**可通过不同层次的抽象，逐步细化软件的结构。
     2. **过程抽象**：
        1. 软件计划阶段：软件被看作是一个相对宏观的系统元素
        2. 软件需求分析阶段：用“问题所处环境的为大家所熟悉的术语”来描述软件的解决方法。
        3. 概要设计阶段：使用**规定的符号**表示软件的轮廓和结构
        4. 详细设计阶段：使用**面向代码的符号**表示软件的内部结构
     3. **数据抽象：**数据抽象与过程抽象一样，允许设计人员在不同层次上描述数据对象的细节。
     4. **控制抽象：**可以包含一个程序控制机制而无须规定其内部细节。
  3. **模块独立性**
     1. 功能独立性是**抽象、模块化和信息隐藏的直接产物**。
     2. 如果一个模块能够**独立于其他模块被编程、测试和修改，而和软件系统中其它的模块的接口是简单的**，则该模块具有功能独立性。
     3. **两个度量准则**：
        1. 模块间的耦合
        2. 模块的内聚。

### 模块内聚

* + 1. **定义**：**模块内部各元素之间的联系程度**。一个模块内部各元素之间的联系越紧密，则它的内聚性就越高。
    2. **内聚是模块功能强度的度量**
    3. 联系得越紧密，**内聚性**越高，与其他模块的**耦合性**就会减低，而模块**独立性**就越强。**【联系紧密程度、内聚性、独立性成正比，内聚性和耦合性成反比】**
    4. 举例：国内能够自给自足（内聚度高），那么和其他国家之间的联系就会不稍微弱一点。
    5. **判断方法：**如果一个模块或者一个类，它完成的任务相同或者相似，则该模块或者类的内聚性比较高
    6. 内聚分类：
       1. **巧合内聚**
          1. 当几个模块内**凑巧**有一些**程序段代码相同**，又没有明确表现出独立的功能，为了**减少存储把这些代码独立出来**建立一个新的模块，这个模块就是巧合内聚模块。
          2. 它是**内聚程度最低**的模块。——绝对避免
       2. **逻辑内聚**
          1. 把几种相关的**功能组合**在一起，每次被调用时，由传送给模块的**判定参数**来确定该模块应执行哪一种功能。
          2. **该内聚的缺陷**：

调用模块需要的功能很简单，但是调用另外一个功能模块时，编译未用到的功能

违背了“消息隐藏”的原则

* + - * 1. **尽力避免**
      1. **时间内聚（经典内聚）**
         1. 含义：这种模块一般为**多功能模块**，但模块的各个功能的执行与时间有关，**通常要求所有功能必须在同一时间段内执行**。
         2. 内聚性相比于前两个较好，但是并不理想
      2. **过程内聚**
         1. 使用流程图做为工具设计程序时，把**流程图中的某一部分**划出组成模块，就得到过程内聚模块。
         2. 要求：程序执行有严格的时间顺序
      3. **通信内聚**
         1. 含义：一个模块内各功能部分都**使用了相同的输入数据**，**或产生了相同的输出数据**，则称之为通信内聚模块。
         2. **判定**：通信内聚模块可通过**数据流图**来判定。
      4. **信息内聚**
         1. 含义：模块**完成多个功能**，各功能都在**同一数据结构上**操作，每一项功能有一个唯一的入口点。
         2. 聚合度高
      5. **功能内聚**
         1. 含义：**一个模块中各个部分**都是完成某一具体功能**必不可少的**组成部分，或者说该模块中所有部分都是为了完成一项具体功能而**协同工作，紧密联系，不可分割的。**则称该模块为功能内聚模块。
         2. 聚合度很高，很难达到

巧合内聚、逻辑内聚、时间内聚、过程内聚、通信内聚、信息内聚、功能内聚

敲锣试过通信工

### 模块耦合性

* + 1. **定义**：是模块之间互相连接的紧密程度的度量。
    2. 接口复杂程度、调用模块的方式、模块数量——都影响了模块的耦合程度
    3. **模块之间的连接**越紧密，**联系越多，耦合性**就越高，而其模块**独立性**就越弱。
    4. 耦合度增加，独立性减弱，内聚度降低
    5. 耦合度过高的影响：？？、整合需要花费更多的时间、很难复用和测试
    6. 耦合分类
       1. **内容耦合（耦合度最高）**
          1. 含义：一个模块**直接访问另一个模块的内部数据**、不通过正常入口转到另一模块内部、两个模块有一部分**程序代码重迭**(汇编语言中)、一个模块有多个入口。
       2. **公共耦合**
          1. 定义：若**一组模块**都**访问同一个公共数据环境**，则它们之间的耦合就称为公共耦合。
          2. **公共的数据环境**可以是全局数据结构、共享的通信区、内存的公共覆盖区等。

公共耦合的复杂程度**随耦合模块的个数增加而显著增加**。

只是**两模块间**有公共数据环境，则**公共耦合有两种情况。**松散公共耦合和紧密公共耦合。

公共耦合度高引发的问题：

所有公共耦合模块都与某一个公共数据环境内部各项的物理安排有关，若**修改某个数据的大小，将会影响到所有**的模块。

**无法控制各**个模块**对公共数据的存取**，严重影响软件模块的可靠性和适应性。

**公共数据名的使用，明显降低了程序的可读性**。

* + - 1. **外部耦合**

含义：一组模块都**访问同一全局简单变量**而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息，则称之为外部耦合。

**和公共耦合的区别：**单一全局变量，而不是公共数据环境

* + - 1. **控制耦合**

含义：如果一个模块通过**传送开关、标志、名字等控制信息，明显地控制选择另一模块的功能**，就是控制耦合

控制模块+被控制模块

在单一接口上选择多功能模块的功能

控制模块必须知道所控制模块内部的一些逻辑关系，降低了模块的独立性。——**违背了“信息隐藏”的准则**

* + - 1. **标记耦合**

含义：**一组模块**通过**参数表（复杂的数据结构）**传递记录信息，就是标记耦合。

把数据结构上的操作全部集中在一个模块中，来消除这种耦合。

* + - 1. **数据耦合**

如果一个模块访问另一个模块时，彼此之间是通过**简单数据参数 (不是控制参数、公共数据结构或外部变量)**  来交换输入、输出信息的，则称这种耦合为数据耦合。

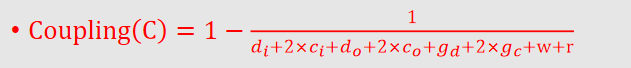
注意：和标记耦合进行比较

松散耦合，模块间的独立性强

* + - 1. **非直接耦合：**
         1. 耦合度最低，独立性最强，很难达到
         2. 如果两个模块之间没有直接关系，**它们之间的联系完全是通过主（上级）模块的控制和调用**来实现的，这就是非直接耦合。

内容公共外部控制标记数据非直接

内公外控彪淑妃

* + 1. **耦合度计算公式：**
       - 1. 
         2. **取值越大，模块之间的耦合度越大。**
         3. **0.67 （低耦合） ~ 1.0（高耦合）**
         4. 参数说明：

针对数据参数和控制参数的耦合项：

di：数据参数的输入个数；

ci：控制参数的输入个数；

do：数据参数的输出个数

co：控制参数的输出个数；

针对全局耦合项：

gd：作为数据的全局变量个数；

gc：作为控制的全局变量个数；

针对环境耦合项：

w：调用的模块数，也称为扇出数；

r：调用的模块数，也称为扇入数。

* + 1. 降低耦合度：
       - 1. 根据问题的特点**选择适当的耦合类型**

模块间的信息传递：数据信息和控制信息的选择；

模块间的调用方式：传送地址和传送判定参数的选择；

系统的错误处理模块：集中处理与分散处理的选择；

* + - * 1. **降低模块接口的复杂度**

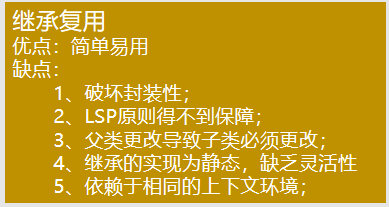
传送信息的数量：参数多和参数少的区别；

模块的调用方式：简单调用和直接引用的区别；

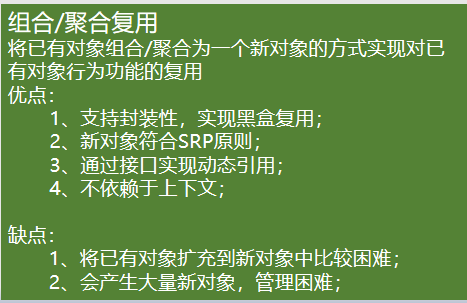
* + - * 1. 将**模块的通信信息放在缓冲区**中

### 面向对象的设计原则

* 1. **单一职责（Single Responsibility）**
     1. 定义：针对**类，应该只有一个**引起它变化的原因——**“职责”**，
     2. 如果有其他原因去改变一个类，那么这个类就具有其他的职责**。类具有多个职责，等于这些职责具有耦合关系。**
     3. 为了提高类的内聚度，**应将对象的不同职责分离至两个或多个类**中，确保引起该类变化的原因只有一个。
     4. **一个类应该只完成一个职责，**如果完成多个职责，会影响功能执行
  2. **开闭原则（Open Closed）**
     1. 软件实体（类、模块、函数）**可以扩展、不能修改**
        1. **对扩展开放，对更改封闭**——需求改变时进行代码拓展而不是代码修改
        2. OCP实现的关键是使用抽象，强调对变化的类进行抽象，
        3. 例子：不理解
  3. **里氏替换原则（Liskov Substitution）（LSP）**
     1. **子类应当可以替换父类**并出现在父类能够出现的任何地方。
     2. 原始定义：若对于每一个类型S的对象o1，都存在一个类型T的对象o2，使得在所有针对T编写的程序P中，用o1替换o2后，程序P的行为功能不变，则S是T的子类型。
     3. 所有**子类的行为功能必须和客户类对其父类所期望的行为功能保持一致。**
     4. 强调了实现抽象化的具体规范。
  4. **依赖倒置原则（Dependency Inversion）**
     1. **高层模块不应依赖于低层模块，二者都应依赖于抽象**；
     2. **细节依赖于抽象**，而不是抽象依赖于细节
     3. **传统的模块间的调用关系与该原则正好相反**，低层模块的改动使得高层模块不得不随之改动。
     4. 程序中所有的依赖关系应该终止于抽象类或者接口：
        1. 任何变量都不应该持有一个指向具体类的指针或者引用；
        2. 任何类都不应该从具体类派生，或者说继承自一个具体类；
        3. 任何方法都不应该覆盖它的任何基类中的已经实现了的方法。
     5. 每个较高层次都为它所需要的服务声明一个抽象接口，较低的层次实现了这些抽象接口
     6. **每个层次通过一个定义良好的、受控的接口向外提供了一组内聚的服务**
  5. **接口隔离原则（Interface Segregation）**
     1. 采用**多个与特定客户类的接口** 比 采用一个通用的涵盖多个业务方法的接口要好
     2. 服务器类应该为每一个客户类创建特定的业务接口，而不要为所有客户类提供统一的业务接口
     3. **接口隔离——多个特定接口，而不是统一接口**
  6. **组合/聚合复用（Composite/Aggregation Reuse）**
     1. 在一个新对象里面使用一些已有对象，使之成为新对象的一部分；新对象通过向已有对象委托（delegate）一部分责任而达到复用已有对象的目的。
     2. 继承复用



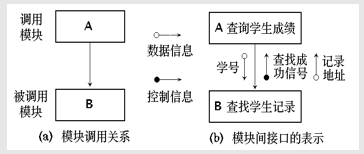
* + 1. 组合/聚合复用

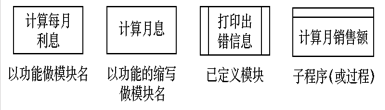


* 1. **迪米特法则（Law of Demeter）**
     1. 最少知识原则：一个对象应当可能少的了解其它对象。
     2. 只和某些对象进行沟通：
        1. 当前对象本身（this）；
        2. 以参量形式传入到当前对象方法中的对象；
        3. 当前对象的实例变量直接引用的对象；
        4. 当前对象的实例变量如果是一个聚集，那么聚集中的元素也都是朋友；
        5. 当前对象所创建的对象。
     3. 只与你直接的朋友们通信，不要跟“陌生人”说话：

### 软件设计基础

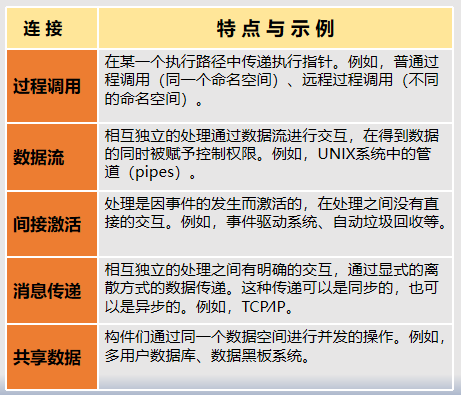
* 1. 自顶向下，逐步细化
  2. 系统控制结构
     1. 表明了程序构件（模块）的组织情况。
     2. 宽度
     3. 深度
     4. 扇入和扇出
  3. 结构划分和结构图
     1. **水平划分**：
        1. 按主要的程序功能来定义模块结构的各个分支。
     2. **垂直划分**：
        1. 用于程序的体系结构中
        2. 一般来说效果更好
     3. 功能结构图
        1. 模块用矩形框表示，用模块名来标记





* 1. 数据结构
     1. 应确定数据的组织、存取方式、相关程度、以及信息的不同处理方法。
  2. 软件过程
     1. 对模块内部细节的实现机制进行详细的描述
     2. 包括事件的顺序、正确的判定点、重复的操作直至数据的组织和结构提供精确的处理说明
     3. 程序结构和软件过程有关
     4. 软件过程是层次化的

### 软件体系结构

* 1. 根本目的：
     1. 根本目的是要解决软件重用、软件质量和软件维护问题
  2. 定义
     1. Booch：软件体系结构 = {组织，元素，子系统，风格}
     2. Bass：系统的一个或多个结构，包括软件构件（Components）、构件的外部可视属性（Properties）、构件之间的关系（Relationships）
     3. Shaw：结构模型、框架模型、动态模型、过程模型
     4. **Garlan & Shaw模型**：软件体系结构 = {构件，连接件，约束}
  3. **三要素**
     1. 程序构件（模块）的层次结构
     2. 构件之间交互的方式
     3. 数据的结构
  4. 软件构件的分类和调用方式
     1. 
     2. 
  5. 软件体系结构风格
     1. 软件体系结构风格：描述某一特定应用领域中系统组织方式的惯用模式
     2. 软件体系结构设计的一个**核心问题**：
        1. 是能否使用重复的体系结构模式。
     3. 四要素：
        1. 一个词汇表
        2. 定义一套配置规则
        3. 定义一套语义解释原则
        4. 定义对基于这种风格的系统所进行的分析
     4. **管道和过滤器风格**
        1. 最早出现在Unix系统中，它适用于对有序数据进行一系列已经定义的独立计算的应用程序。
        2. 构件：过滤器（filter）。【过滤器之间彼此独立】
        3. 它对输入流进行处理、转换，处理后的结果在输出端流出。而且，这种计算处理方式是递进的，所以可能**在全部的输入接受完之前就开始输出。**此外，系统中**可以并行地使用过滤器**。
        4. 连接件：连接件位于过滤器之间，起到信息流的导管作用，被称为管道（pipe）
        5. 不适合处理交互性数据应用



* + 1. **调用和返回风格**
       1. 调用/返回风格的体系结构在过去的30年之间占有重要的地位，是**大型软件开发**中的主流风格的体系结构。
          1. **主/子程序风格的体系结构**——面向过程

将程序划分为若干个小片段，从而使程序的可更改性大大提高。

这种风格有一定的层次性，主程序位于一层，下面可以再划分一级子程序，二级子程序甚至更多。

这种风格是单线程控制的，同一时刻只有一个孩子结点的子程序可以得到父亲结点的控制。

* + - * 1. **对象风格的体系结构**——面向对象

这种风格建立在数据抽象和面向对象的基础上，数据的表示方法和它们的相应操作封装在一个抽象数据类型或对象中。

对象是一种被称作管理者的构件，由它负责保持资源的完整性。对象是通过函数和过程调用来交互的。

封装性良好，透明性

* + - * 1. **分层风格的体系结构**

将系统组织成一个层次结构，每一层为上层提供服务，并作为下层的客户端。

这种风格支持基于可增加抽象层的设计。这样，允许将一个复杂问题分解成一个增量步骤序列的实现。

由于每一层最多只影响两层，同时只要给相邻层提供相同的接口，允许每层用不同的方法实现，同样为软件复用提供了强大的支持。

* + 1. **基于事件的风格**
       - 1. 构件不直接调用一个过程，而是声明或广播一个或多个事件。
         2. 适用于**设计低耦合构件集合的应用程序**，每个构件完成一定的操作，并可能触发其他构件的操作。
         3. 构件的接口不仅提供一个过程的集合，也提供一个事件的集合。这些过程既可以用一般的方式调用，也可能被注册为与某些事件相关。
         4. 构件可以声明或广播一个或多个事件，或者向系统注册用以表明它希望响应一个或多个事件。
    2. **客户端/服务器风格**
       - 1. 目标是达到可测量性的需求，并适用于应用程序的数据和处理分布在一定范围内的多个构件上，且构件之间通过网络连接。
    3. **解释器/虚拟机风格**
       1. 建立一种虚拟机去弥合程序的语义与作为计算引擎的硬件的差异。
       2. 适用于应用程序不能直接运行在最合适的机器上或不能直接以最适合的语言执行
    4. **仓库风格**
       1. 体系结构由两种构件组成：
          1. 中央数据结构，表示当前状态；
          2. 独立构件的集合，它对中央数据结构进行操作。
       2. 传统的数据和状态控制方法：输入事务选择进行何种处理
       3. 黑板体系结构：中央数据结构的当前状态决定进行何种处理
    5. **黑板风格**
       1. 用于在信号处理方面进行复杂解释的应用程序、松散的构件访问共享数据的应用程序
       2. 知识源：特定应用程序知识的独立散片
          1. 知识源之间的交互旨在黑板内部发生
       3. 黑板数据结构
       4. 控制器
    6. **MVC风格——使用较多**
       - 1. **模型-视图-控制器**
         2. **视图**：为用户显示模型信息。视图从模型获取数据，一个模型可以对应有多个视图。
         3. **模型**：模型是应用程序的核心，它封装内核数据与状态。对模型的修改将扩散到所有视图中。所有需要从模型中获取数据的对象都必须注册为模型的视图。
         4. **控制器**：是提供给用户进行操作的接口。每个视图与一个控制器构件相关联。控制器接受用户输入，输入事件转换成服务请求，传送到模型或视图。用户只通过控制其与系统进行交互
         5. 目的是将Model和View的实现代码分离，从而使同一个程序可以使用不同的表现形式。Controller存在的目的则是确保M和V的同步，一旦M改变，V应该同步更新。

