BUPT
TSEG

软件工程 模型与方法
Models & Methods of SE

软件设计

本章内容

- 软件设计的目标
- 软件的设计过程
  - 软件概要设计
  - 软件详细设计
- 软件设计模型
- 软件设计的一般性原则
  - 面向对象的设计原则
- 软件设计基础
- 软件体系结构简介

引言

- 相对于分析而言，设计是什么？为什么要进行设计？
- 软件设计在软件开发过程中处于什么位置？
- 经过软件需求分析，确定了系统必须“做什么”的功能，然而这些功能是如何实现用户需求的呢？
- 为了实现合理分配系统功能并尽可能提高处理用户需求的性能，系统中这些功能之间是什么关系呢？
- 围绕着核心功能的设计，界面和数据库表的设计如何考虑？
- ……

软件设计历史

- 早期的设计工作集中在模块化程序的开发标准和自顶向下求精软件结构的方法，称为结构化程序设计的理论。
- 之后提出了将数据流或数据结构转化为设计定义的方法，目前又提出了面向对象的方法进行软件设计。
- 如今，在软件设计方面的着重点已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。
- 各种软件设计方法都具有以下共同特征：
  - 一种用于将分析模型变换到设计模型的表示机制；
  - 用于表示功能件构件及其接口的符号体系；
  - 用于求精（优化）和划分的启发机制；

软件设计的目标

- 根据软件需求分析的结果，设想并设计软件，即根据“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?
  - 软件系统的结构设计、处理方式（性能）的设计；
  - 数据结构和数据存储的设计；
  - 界面和可靠性设计等
- 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
- 软件设计的最终目标是要取得最佳方案（Best Solution，主观）
  - “最佳”是指在所有候选方案中，就节省开发费用，降低资源消耗，缩短开发时间的条件，选择具有较高的生产率、较高的可靠性和可维护性的方案。

软件设计过程

- 软件设计是一个把软件需求变换成包含软件功能模型、数据模型以及行为模型的过程。从工程管理的角度，软件设计分成：
  - 概要设计：只需描绘出可直接反映功能、数据、行为需求的软件总体框架；
  - 详细设计：即过程设计，通过对软件结构进行细化，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非常接近于源程序的软件设计模型。

- 系统结构设计定义了软件系统各主要元素（主要指功能模块）之间的关系，其中包括软件的模块接口设计；
- 数据设计将软件各模块所需要处理的数据以及系统需要长久保存的数据进行数据结构和数据存储的设计；
- 过程设计主要是确定各功能模块内部结构的详细定义，包括模块主要算法逻辑和局部数据结构的定义。

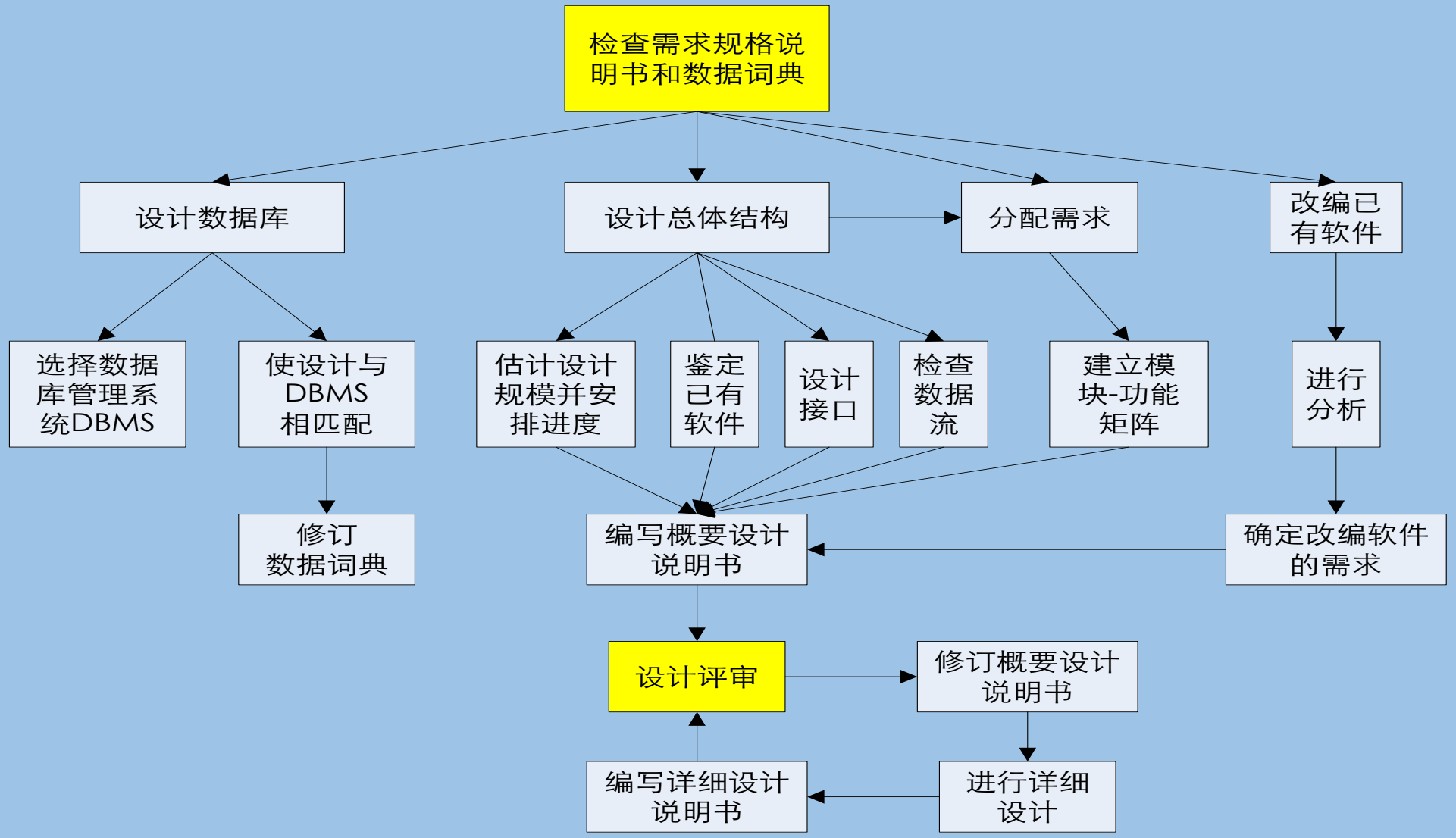
软件概要设计

- 制定设计规范
- 软件系统结构的总体设计
- 处理方式设计（性能设计）
- 数据结构设计
- 可靠性设计（质量设计）
- 界面设计（需求的直接表达方式）
- 编写软件概要设计说明书
- 概要设计评审

软件详细设计

- 确定软件各个功能模块内的算法以及各功能模块的内部数据组织。
- 选定某种表达形式来描述各种算法。
- 编写软件详细设计说明书
- 进行详细设计的评审。

软件设计的整体流程


检查需求规格说明书和数据词典
设计数据库
设计总体结构
分配需求
改编已有软件
选择数据库管理系统DBMS
使设计与DBMS相匹配
估计设计规模并安排进度
鉴定已有软件
设计接口
检查数据流
建立模块-功能矩阵
进行分析
修订数据词典
编写概要设计说明书
确定改编软件的需求
设计评审
修订概要设计说明书
编写详细设计说明书
进行详细设计

软件设计模型

- 软件设计既是过程又是模型。
- 软件设计模型由两个部分构成：
  - 动态结构设计：以某种方式表示功能响应客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是如何协调工作的机制。
  - 静态结构设计：由软件的功能结构和数据结构组成，展示软件系统能够满足所有需求的框架结构；
- 软件的设计活动
  - 系统结构设计及数据结构设计；
  - 接口设计和过程设计；
  - 界面设计、组件设计及优化设计等；

软件设计的一般性原则

- The design process should not suffer from ‘tunnel vision.’
- The design should be traceable to the analysis model.
- **The design should not reinvent the wheel.**
- The design should “minimize the intellectual distance” between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- **The design should be structured to accommodate change.**
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered. (异常保护机制)
- **Design is not coding, coding is not design.**
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

衡量设计过程的技术原则

- 设计过程应该是可追踪和可回溯的。
- 设计必须实现分析模型中描述的所有显式需求；
- 必须满足用户希望的所有隐式需求。
  - 所谓隐式需求，如系统的安全性要求，降低或消除功能性错误，数据安全和完整性要求等。
- 对于开发者和未来的维护者而言，设计说明文档必须是可读的、可理解的，使得将来易于编程、易于测试、易于维护。

衡量设计模型的技术原则

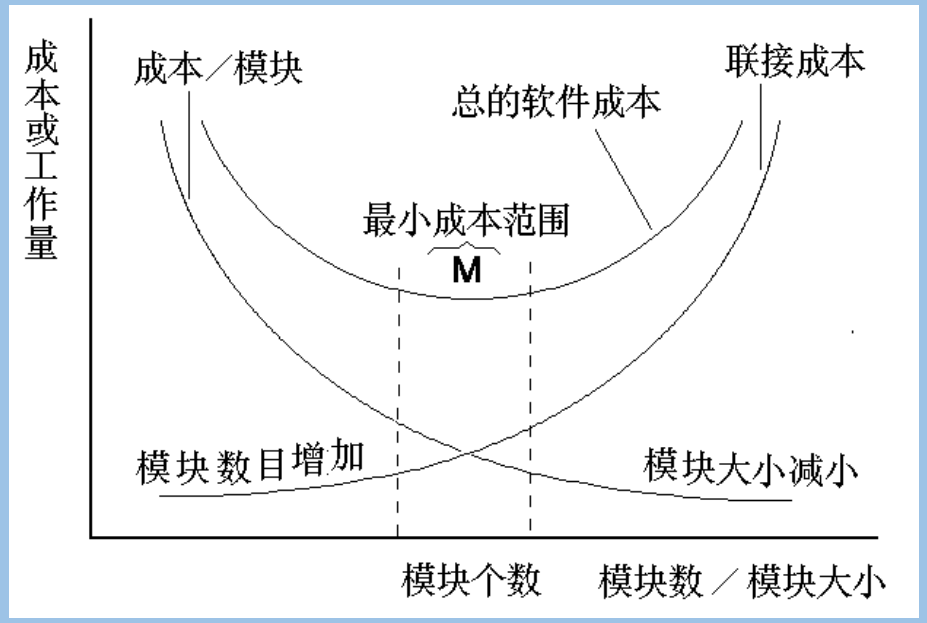
- 设计模型应该展现软件的全貌，包括从实现角度可看到的数据、功能、行为。
- 设计模型应该是一个分层结构。该结构：
  - 使用可识别的设计模式搭建系统结构；
  - 由具备良好设计特征的构件构成；
  - 可以用演化的方式实现；
- 设计应当模块化，即应当建立具有独立功能特征的构件。
- 设计应当建立能够降低模块与外部环境之间复杂连接的接口。
- 设计应当根据将要实现的对象和数据导出合适的数据结构。

软件的模块化

- 模块(module)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。
- 模块具有如下三个基本属性：
  - 功能：实现什么功能，做什么事情。
  - 逻辑：描述模块内部怎么做。
  - 状态：该模块使用时的环境和条件。
- 模块的表示
  - 模块的外部特性：是指模块的模块名、参数表、以及给程序以至整个系统造成的影响。
  - 模块的内部特性：是指完成其功能的程序代码和仅供该模块内部使用的数据。

软件模块的划分

- 按照自顶向下的设计原则，需将一个大规模的软件分解成若干相对独立的模块，然后分别对这些规模较小的模块进行处理。
- 难点在于合理的划分模块
  - 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
  - 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。



降低了系统的复杂性，使得系统容易修改;

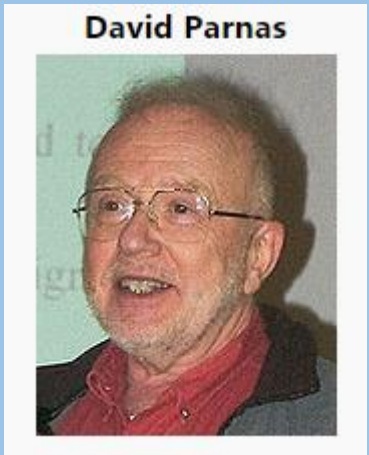
推动了系统各个部分的并行开发，从而提高了软件的生产效率。

模块设计的标准

- 模块可分解性：可将系统按问题 / 子问题分解的原则分解成系统的模块层次结构;
- 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。
- 模块可理解性：一个模块可不参考其他模块而被理解;
- 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;
- 模块保护：将模块内出现异常情况的影响范围限制在模块

信息隐藏

- **David Parnas** 指出，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。
- 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。
- 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。



David Lorge Parnas (born February 10, 1941) is a Canadian early pioneer of software engineering, who developed the concept of information hiding in modular programming, which is an important element of object-oriented programming today. He is also noted for his advocacy of precise documentation.

抽象化

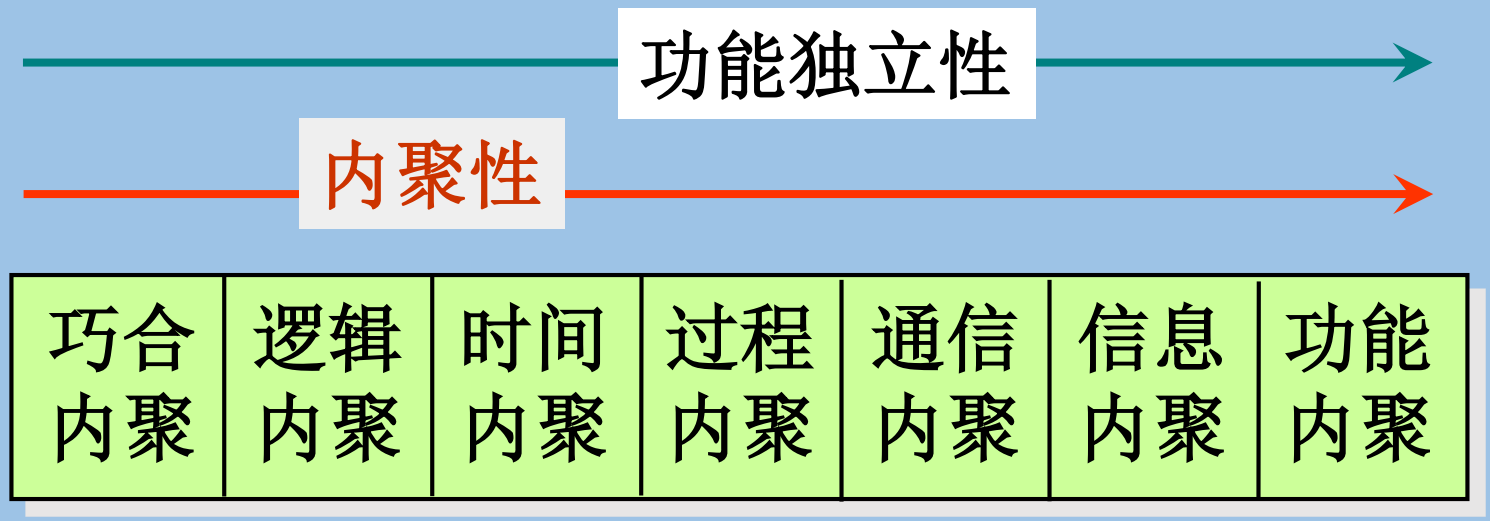
- 模型是人类抽象活动的一种表现形式。在软件设计过程中按照自顶向下的设计原则，可通过不同层次的抽象，逐步细化软件的结构。
  - 过程抽象：
    - 软件计划阶段：软件被看作是一个相对宏观的系统元素
    - 软件需求分析阶段：用“问题所处环境的为大家所熟悉的术语”来描述软件的解决方法。
    - 概要设计阶段：使用规定的符号表示软件的轮廓和结构
    - 详细设计阶段：使用面向代码的符号表示软件的内部结构
  - 数据抽象：数据抽象与过程抽象一样，允许设计人员在不同层次上描述数据对象的细节。
  - 控制抽象：可以包含一个程序控制机制而无须规定其内部细节。

模块的独立性

- 功能独立性是抽象、模块化和信息隐藏的直接产物。
- 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。
- 1978年Meyer提出了两个准则度量模块独立性，即模块间的耦合和模块的内聚。

模块的内聚性

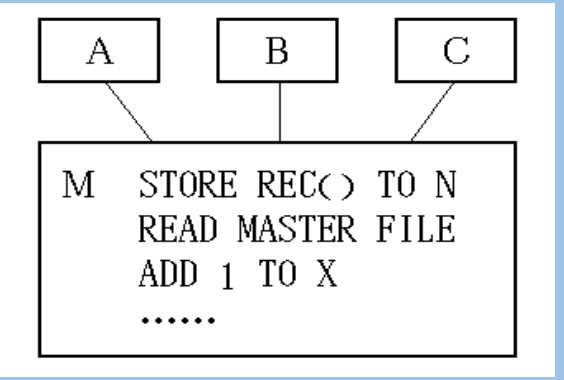
- 内聚是模块功能强度的度量，一个模块内部各元素之间的联系越紧密，则它的内聚性就越高，相对地，它与其他模块之间的耦合性就会减低，而模块独立性就越强。



模块的内聚性

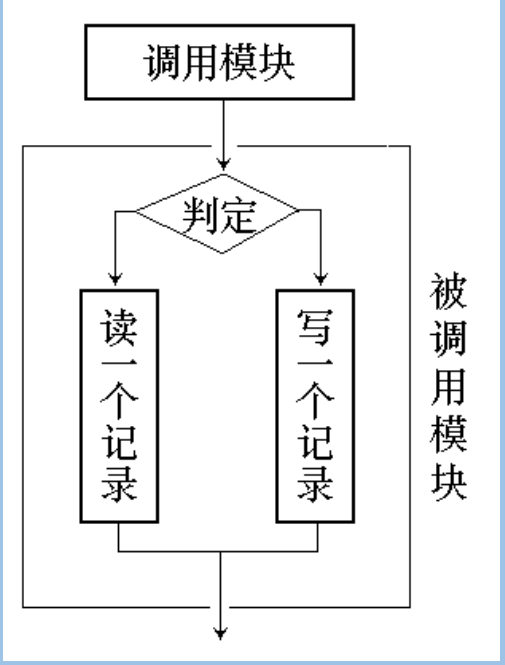
- 巧合内聚

  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。它是内聚程度最低的模块。



- 逻辑内聚

  这种模块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。

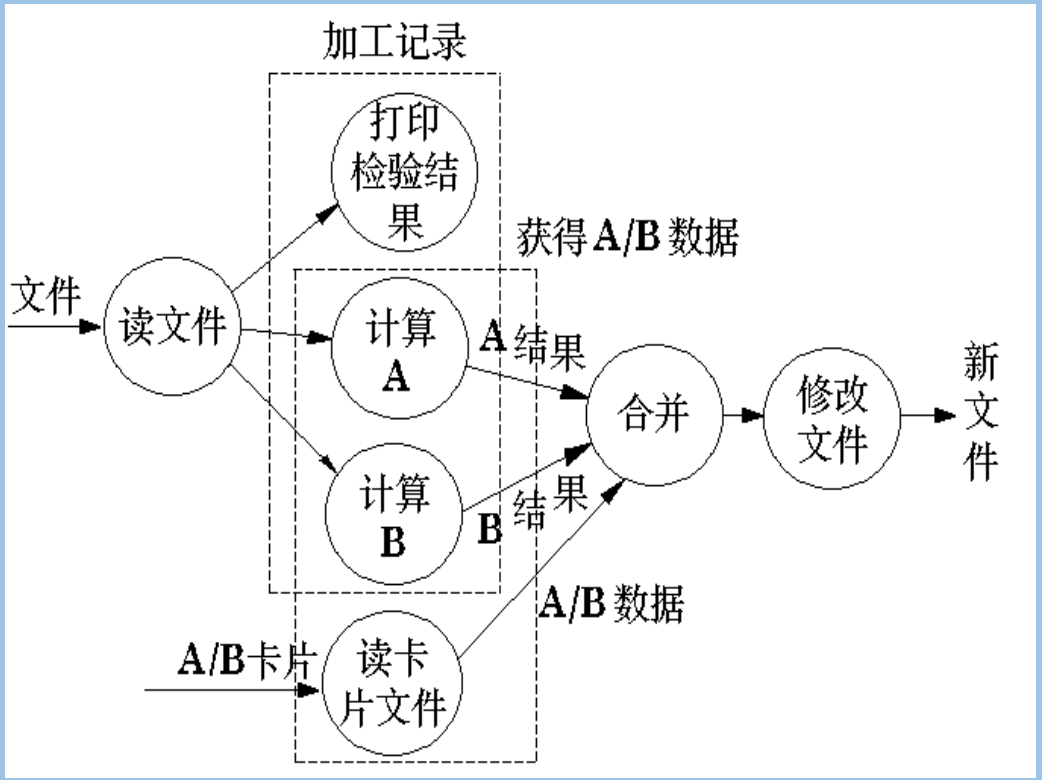


模块的内聚性

- 时间内聚
  - 时间内聚又称为经典内聚。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。
  - 例如初始化模块和终止模块。
- 过程内聚
  - 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。
  - 例如，把流程图中的循环部分、判定部分、计算部分分成三个模块，这三个模块就是过程内聚模块。

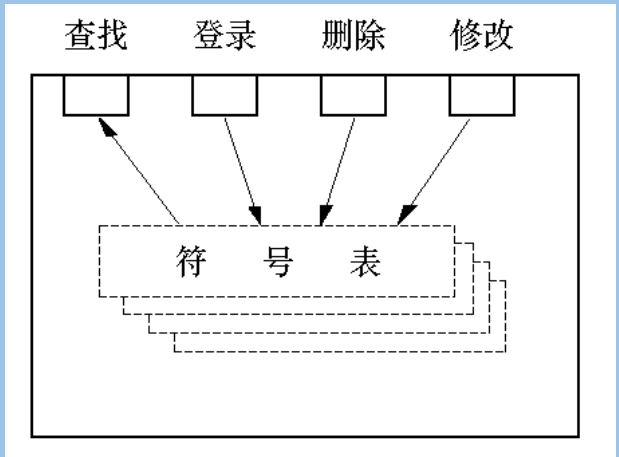
模块的内聚性

- 通信内聚
  - 如果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。通常，通信内聚模块可通过数据流图来判定。



模块的内聚性

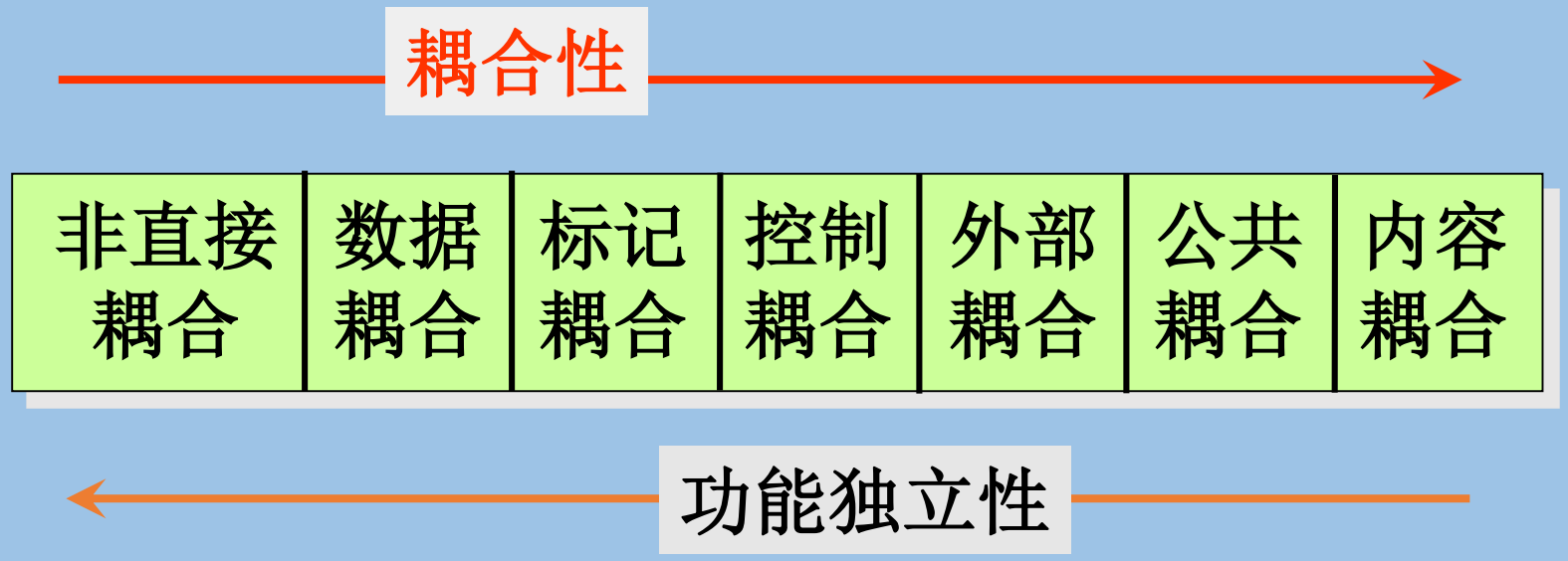
- 信息内聚
  - 这种模块完成多个功能，各功能都在同一数据结构上操作，每一项功能有一个唯一的入口点。
  - 这个模块将根据不同的要求，确定该执行哪一个功能。由于这个模块的所有功能都是基于同一个数据结构（符号表），因此，它是一个信息内聚的模块。
  - 信息内聚模块可以看成是多个功能内聚模块的组合，并且达到信息的隐蔽。



- 功能内聚
  - 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。

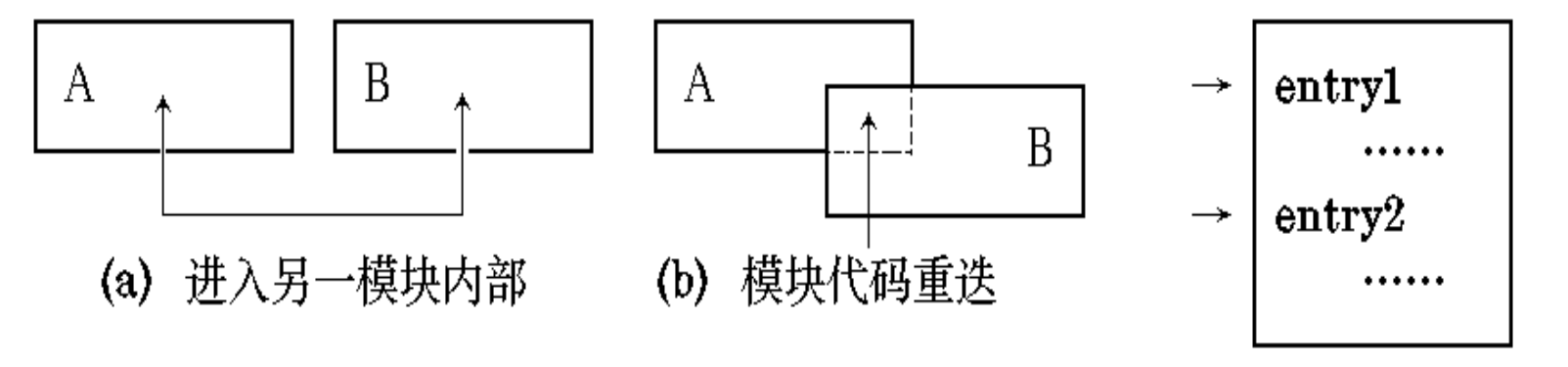
模块的耦合性

- 耦合是模块之间互相连接的紧密程度的度量。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。



模块的耦合性

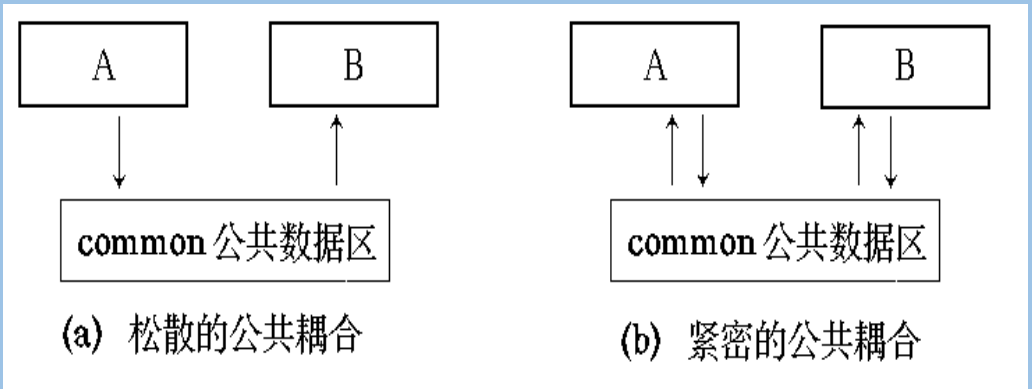
- 内容耦合
  - 一个模块直接访问另一个模块的内部数据;
  - 一个模块不通过正常入口转到另一模块内部;
  - 两个模块有一部分程序代码重迭(汇编语言中);
  - 一个模块有多个入口。



模块的耦合性

- 公共耦合（Common Coupling）
  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。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。若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。



公共耦合会引起下列问题：

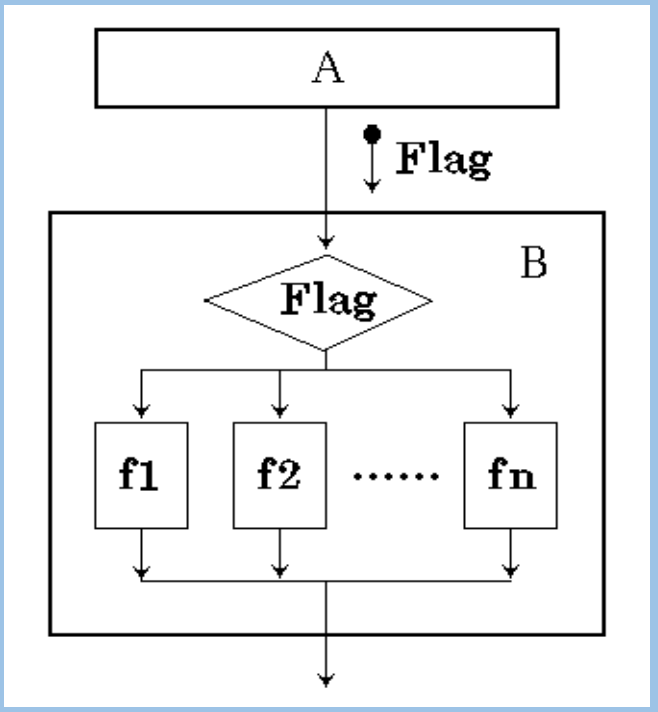
所有公共耦合模块都与某一个公共数据环境内部各项的物理安排有关，若修改某个数据的大小，将会影响到所有的模块。

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。

模块的耦合性

- 外部耦合
  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。
- 控制耦合
  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。
  - 控制耦合是在单一接口上选择多功能某块中的某项功能。
  - 对所控制模块的任何修改，都会影响控制模块
  - 且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。



模块的耦合性

- 标记耦合
  - 如果一组模块通过参数表传递记录信息，就是标记耦合。这个记录是某一数据结构的子结构，而不是简单变量。
  - 要求这一组模块都必须清楚该数据结构，并按结构的要求进行操作。它使在数据结构上的操作复杂化了，应该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。

p0 (x0, y0)　　p1 (x1, y1)

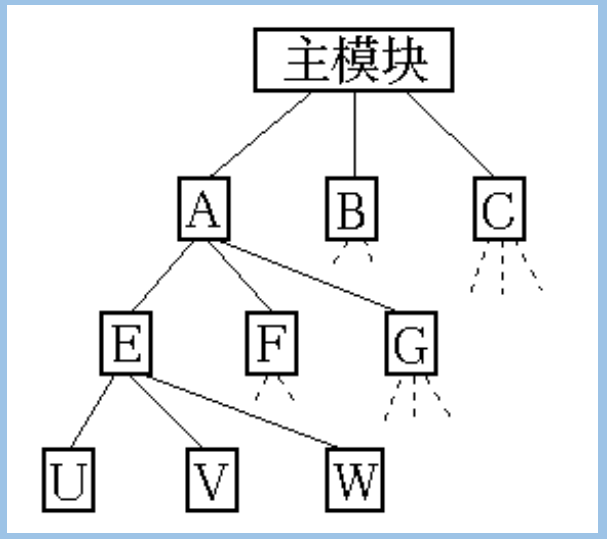
标记耦合　LINE (p0, p1)

模块的耦合性

- 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。
  - 由于限制了只通过参数传递数据，该方式开发的程序简单、安全可靠，因此它是一种松散的耦合，模块间独立性强。

p0 (x0, y0)　　p1 (x1, y1)

数据耦合　LINE (x0, y0, x1, y1)



- 非直接耦合
  - 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。这种耦合的模块独立性最强。

模块的耦合度计算

- $\text{Coupling(C)} = 1 - \frac{1}{d_i+2\times c_i+d_o+2\times c_o+g_d+2\times g_c+\text{w}+\text{r}}$
- 其中针对数据参数和控制参数的耦合项：
  - d_i：数据参数的输入个数；c_i：控制参数的输入个数；
  - d_o：数据参数的输出个数；c_o：控制参数的输出个数；
- 针对全局耦合项：
  - g_d：作为数据的全局变量个数；
  - g_c：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。

Coupling(c)取值越大意味着模块之间的耦合度越大。根据经验，该公式的取值范围应该大约在0.67（低耦合）到1.0（高耦合）

降低模块耦合度的方法

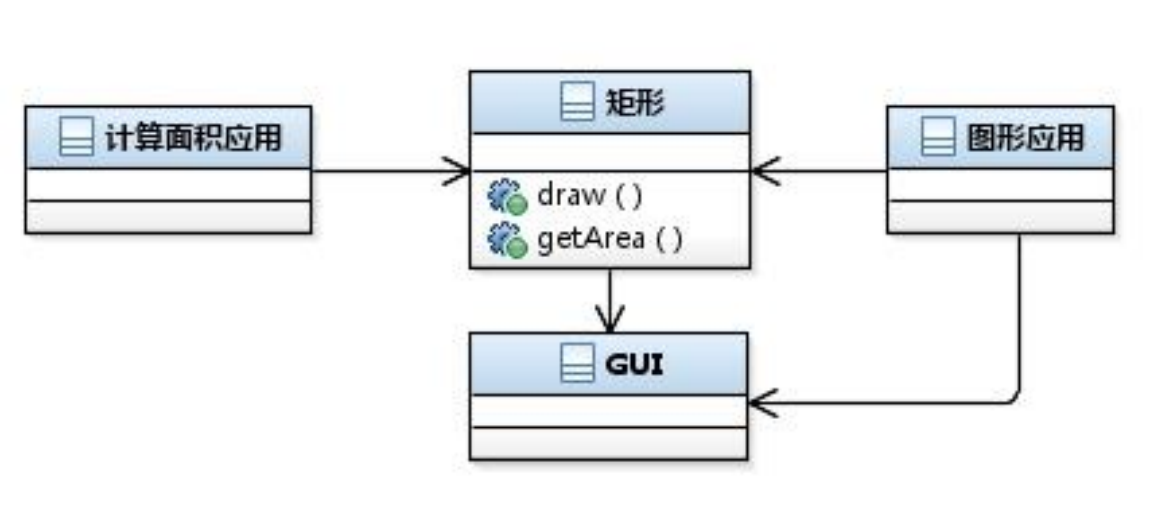
- 根据问题的特点选择适当的耦合类型
  - 模块间的信息传递：数据信息和控制信息的选择；
  - 模块间的调用方式：传送地址和传送判定参数的选择；
  - 系统的错误处理模块：集中处理与分散处理的选择；
- 降低模块接口的复杂度
  - 传送信息的数量：参数多和参数少的区别；
  - 模块的调用方式：简单调用和直接引用的区别；
- 将模块的通信信息放在缓冲区中

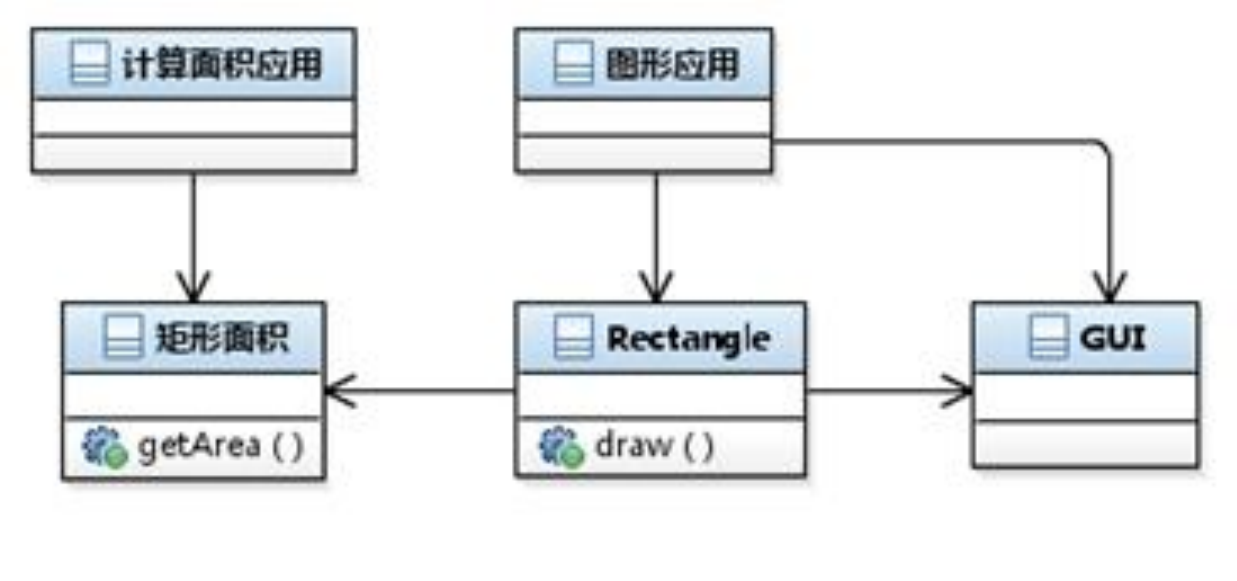
面向对象的设计原则

- 单一职责（Single Responsibility）
- 开闭原则（Open Closed）
- 里氏替换原则（Liskov Substitution）
- 依赖倒置原则（Dependency Inversion）
- 接口隔离原则（Interface Segregation）
- 组合/聚合复用（Composite/Aggregation Reuse）
- 迪米特法则（Law of Demeter）

单一职责（Single Responsibility）

- 定义：针对类，应该只有一个引起它变化的原因，“职责”定义为变化的原因。
  - 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。
  - 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。





里氏替换原则（Liskov Substitution）

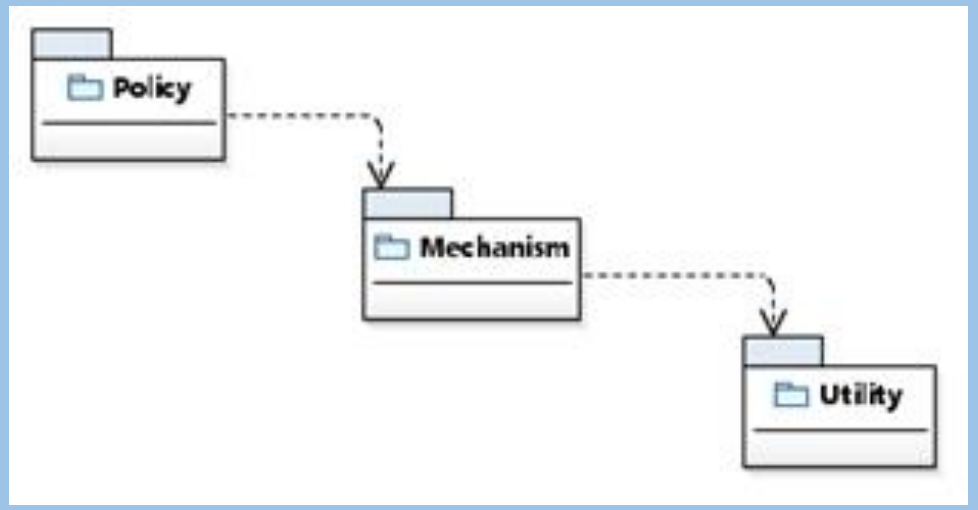
- **子类应当可以替换父类并出现在父类能够出现的任何地方。**
  - 原始定义如下：若对于每一个类型S的对象o1，都存在一个类型T的对象o2，使得在所有针对T编写的程序P中，用o1替换o2后，程序P的行为功能不变，则S是T的子类型。
  - LSP告诉我们：如果一个软件对象是父类型的，那么它出现的所有地方都可以被子类型对象替换。所有子类的行为功能必须和客户类对其父类所期望的行为功能保持一致。
  - OCP原则强调对变化的类进行抽象，LSP则强调了实现抽象化的具体规范。

依赖倒置原则（Dependency Inversion）

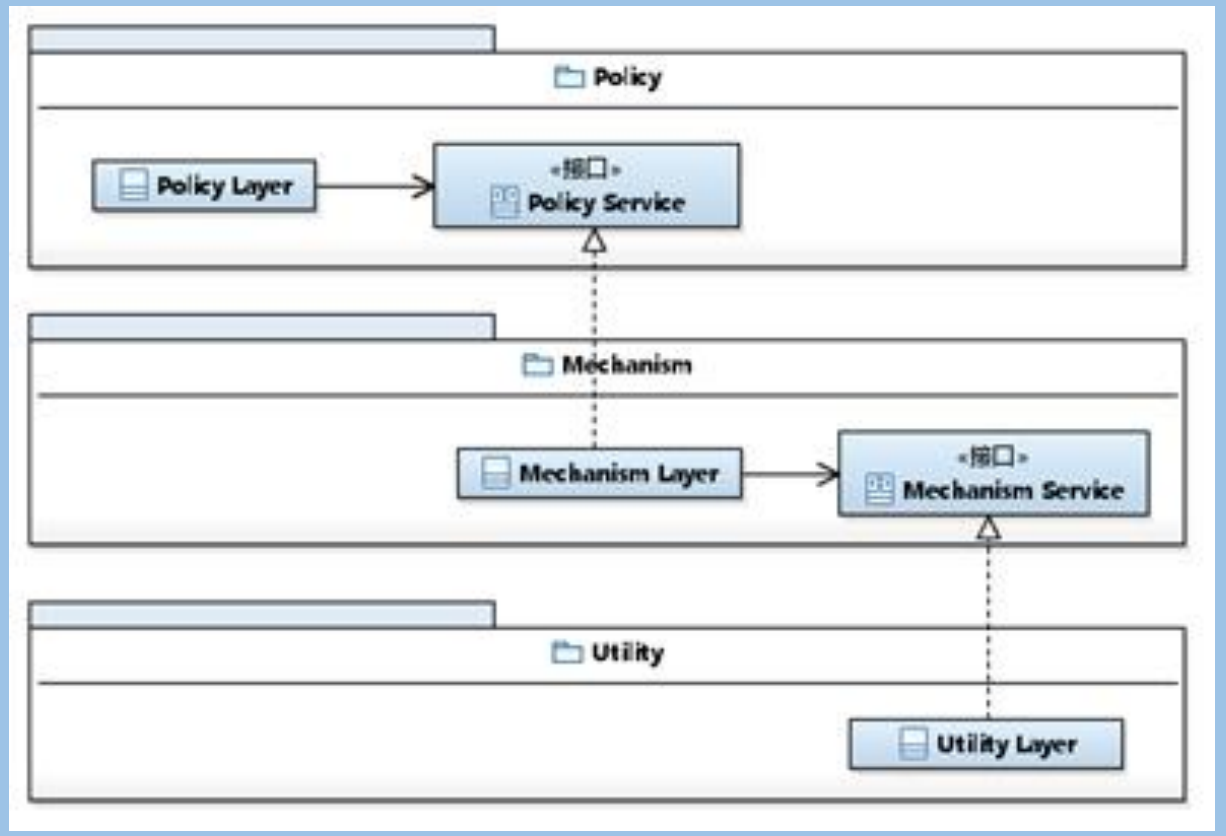
- 高层模块不应依赖于低层模块，二者都应依赖于抽象；抽象不应依赖于实现细节，细节应该依赖于抽象。然而，传统的模块间的调用关系与该原则正好相反，低层模块的改动使得高层模块不得不随之改动。
  - 高层模块包含应用程序的策略规则和业务逻辑，低层模块是具体的实现细节。
  - 低层模块的改动不应影响高层的逻辑；
  - 软件设计能够重用高层模块；
- 程序中所有的依赖关系应该终止于抽象类或者接口：
  - 任何变量都不应该持有一个指向具体类的指针或者引用；
  - 任何类都不应该从具体类派生，或者说继承自一个具体类；
  - 任何方法都不应该覆盖它的任何基类中的已经实现了的方法。

依赖倒置原则（Dependency Inversion）

“...所有结构良好的面向对象架构都具有清晰的层次定义，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”。对这个陈述的简单理解可能导致设计者设计出类似下图的结构分层包图。

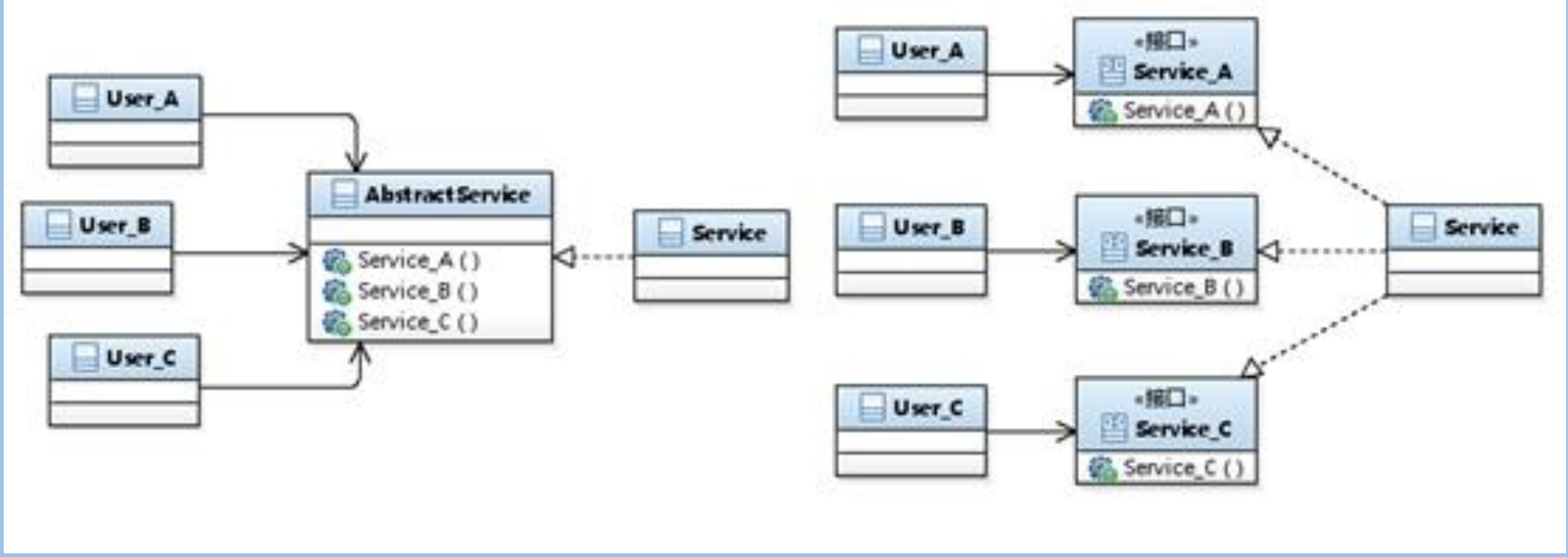


每个较高层次都为它所需要的服务声明一个抽象接口，较低的层次实现了这些抽象接口，每个高层类都通过该抽象接口使用下一层的服务。



接口隔离原则（Interface Segregation）

- **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。**
  - 如果一个服务器类为多个客户类提供不同的服务，则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，除非这些客户类请求的服务相同。

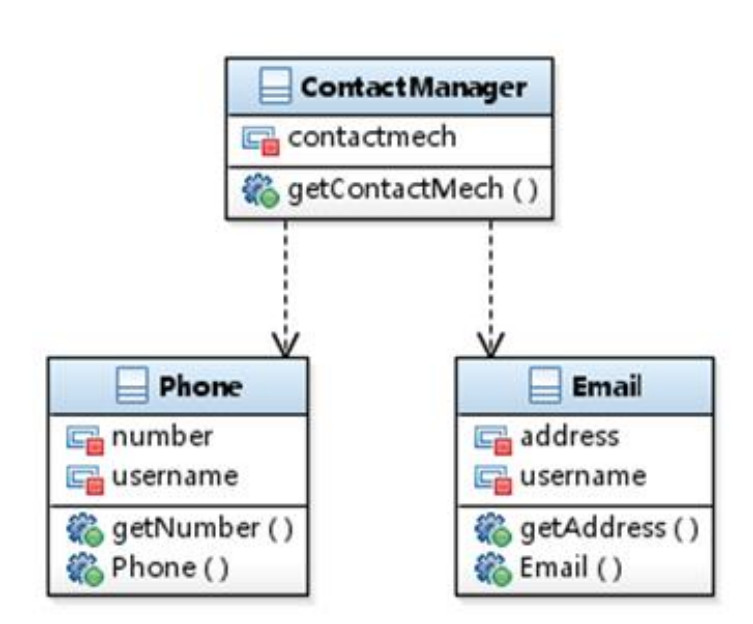


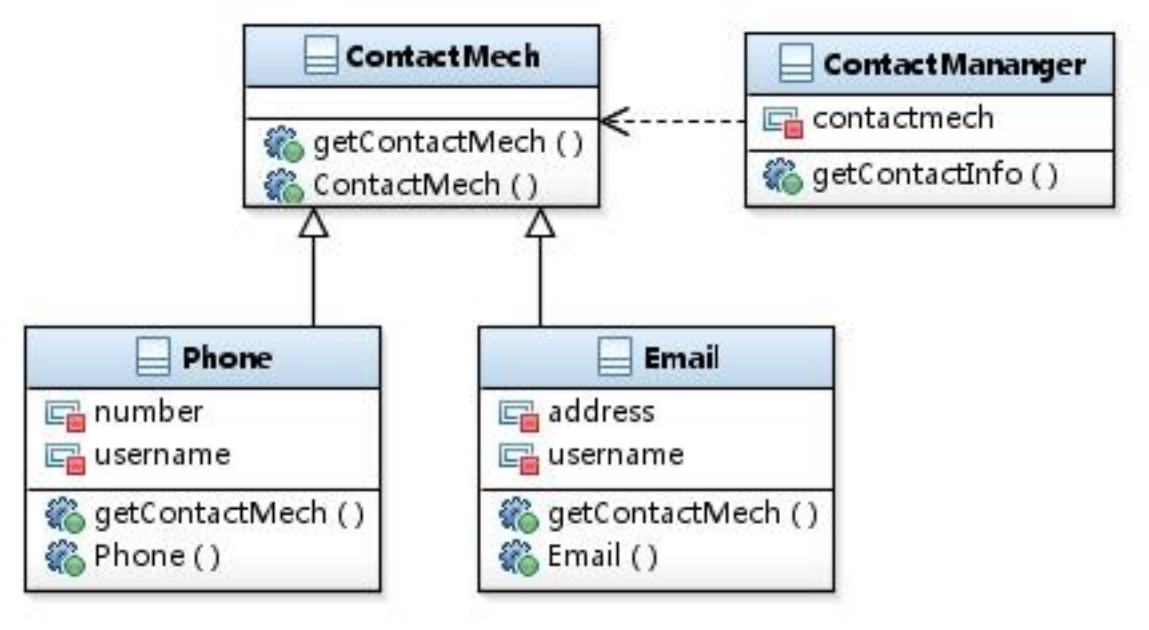
迪米特法则（Law of Demeter）

- 最少知识原则：一个对象应当可能少的了解其它对象。
- 只与你直接的朋友们通信，不要跟“陌生人”说话：
  - 当前对象本身（this）；
  - 以参量形式传入到当前对象方法中的对象；
  - 当前对象的实例变量直接引用的对象；
  - 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；
  - 当前对象所创建的对象。

开闭原则（Open-Closed）

- 软件实体（类、模块、函数）可以扩展，但不能修改。
  - 对于扩展是开放的（Open 4 extension），需求改变时，对实体进行扩展。
  - 对于更改是封闭的（Closed 4 modification），需求改变时，禁止对原来的代码进行修改。
  - 实现OCP的关键是使用抽象，识别不同类之间的共性和变化点，利用封装对变化点进行处理。





组合/聚合复用原则（Composite/Aggregation Reuse）

- 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。
- 两种复用的方式：

继承复用

优点：简单易用

缺点：

1、破坏封装性；
2、LSP原则得不到保障；
3、父类更改导致子类必须更改；
4、继承的实现为静态，缺乏灵活性
5、依赖于相同的上下文环境；

组合/聚合复用

将已有对象组合/聚合为一个新对象的方式实现对已有对象行为功能的复用

优点：

1、支持封装性，实现黑盒复用；
2、新对象符合SRP原则；
3、通过接口实现动态引用；
4、不依赖于上下文；

缺点：

1、将已有对象扩充到新对象中比较困难；
2、会产生大量新对象，管理困难；

软件设计基础

- 自顶向下，逐步细化
- 系统控制结构
- 结构划分和结构图
- 数据结构
- 软件过程

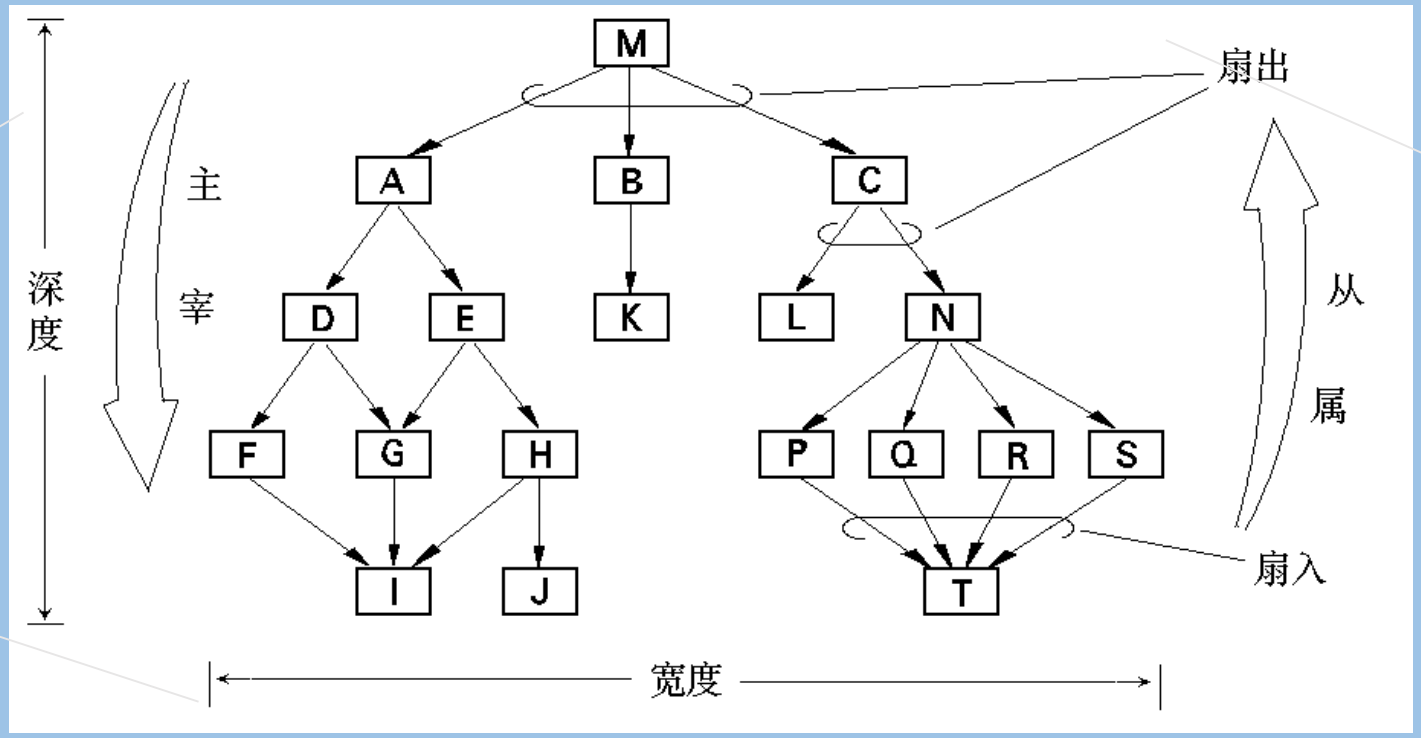
自顶向下逐步细化

- 这是Niklaus Wirth提出的设计策略。
- 将软件的体系结构按自顶向下方式，对各个层次的过程细节和数据细节逐层细化，直到用程序设计语言的语句能够实现为止，从而最后确立整个的体系结构。
  - 最初的说明只是概念性地描述了系统的功能或信息，但并未提供有关功能的内部实现机制或有关信息的内部结构的任何信息。
  - 对初始说明仔细推敲，进行功能细化或信息细化，给出实现的细节，划分出若干成份。

系统控制结构

- 系统控制结构表明了程序构件（模块）的组织情况。控制层次往往用程序的层次（树形或网状）结构来表示。

反映了程序结构的规模和复杂程度



多扇出意味着需要控制和协调许多下属模块。
而多扇入的模块通常是公用模块

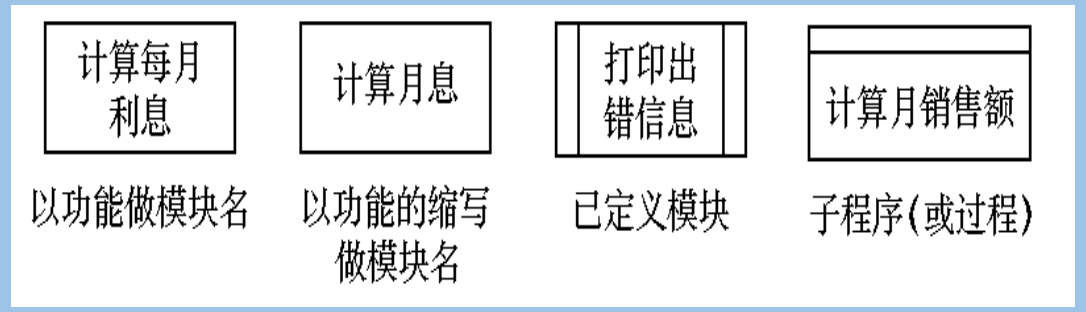
反映了程序结构的控制规模

结构划分

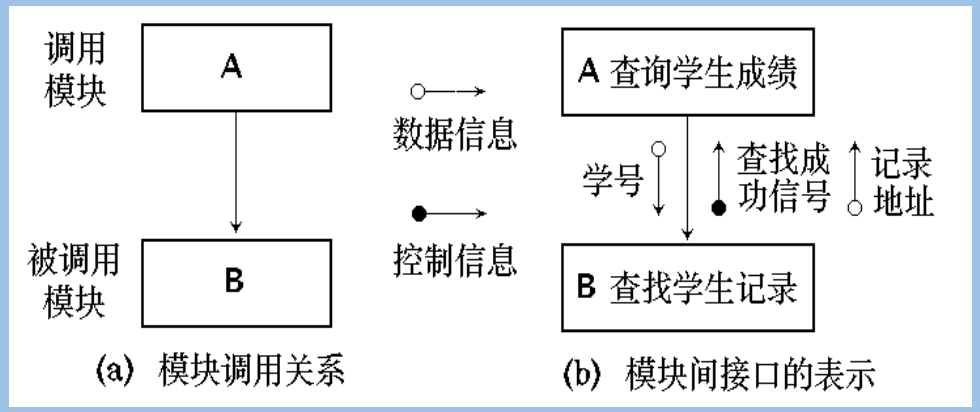
- 程序结构可以按水平方向或垂直方向进行划分（结构化程序设计）
  - 水平划分：水平划分按主要的程序功能来定义模块结构的各个分支。
    - 优点是：由于主要的功能相互分离，易于修改、易于扩充，且没有副作用。
    - 缺点是：需要通过模块接口传递更多的数据，使程序流的整体控制复杂化。
  - 垂直划分：也叫做因子划分，主要用在程序的体系结构中。
    - 优点是：对低层模块的修改不太可能引起副作用的传播

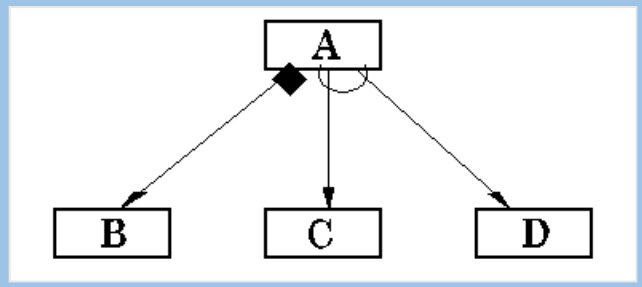
（结构化）功能结构图

- 在结构图中，模块用矩形框表示，并用模块的名字标记它。



- 模块的调用关系和接口

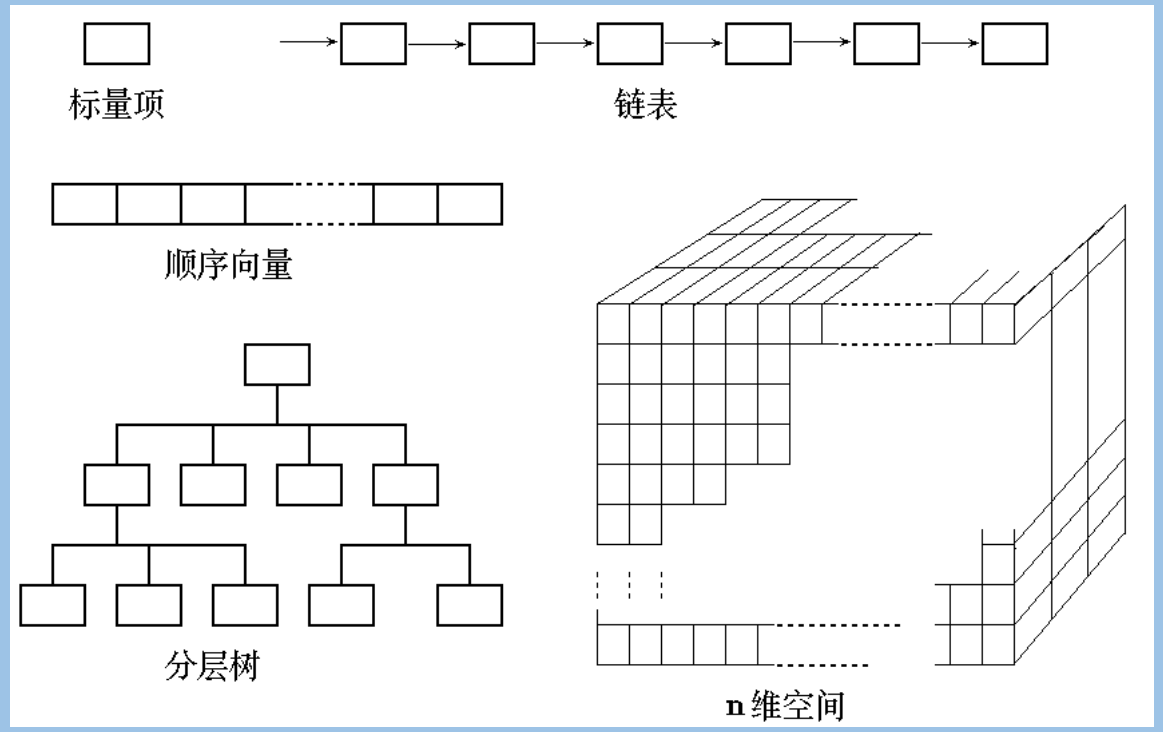




模块间的信息传递：当一个模块调用另一个模块时，调用模块把数据和控制信息传送给所调用模块，以使所调用模块能够运行。而在执行所调用模块的过程中又把它产生的数据或控制信息回送给调用模块。

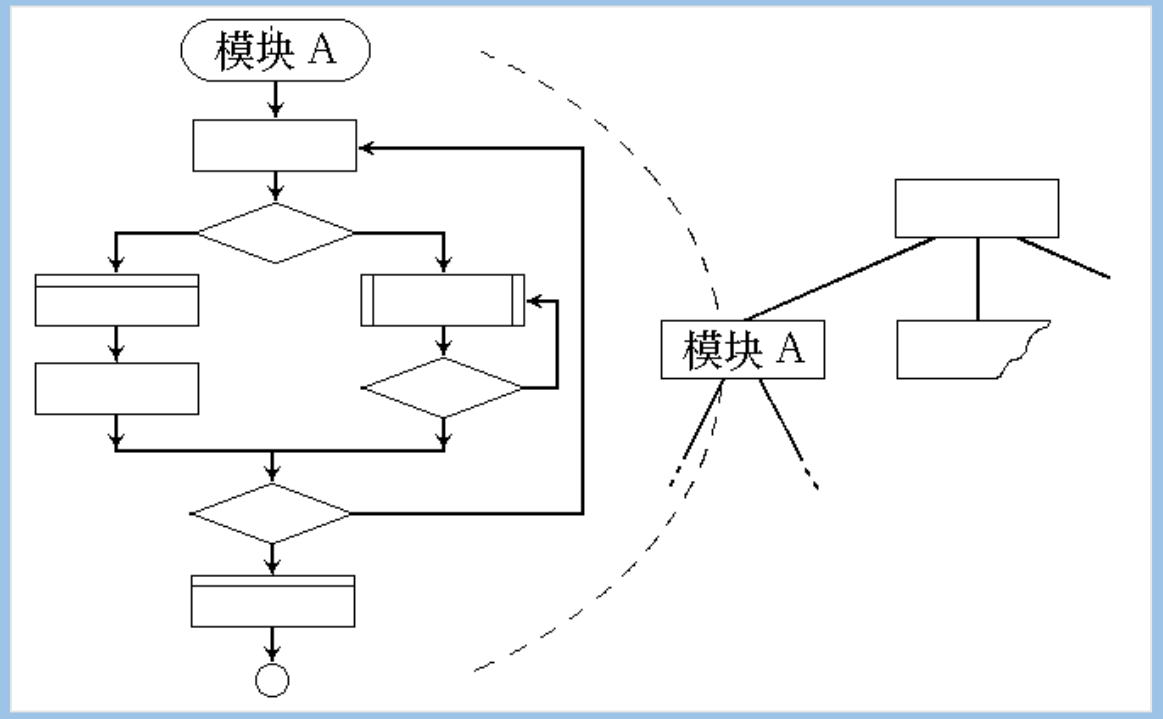
数据结构

- 数据结构是数据的各个元素之间的逻辑关系的一种表示。
- 数据结构设计应确定数据的组织、存取方式、相关程度、以及信息的不同处理方法。



软件过程

- 软件过程必须提供精确的处理说明，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等等。
- 程序结构与软件过程是有关系的。对每个模块的处理必须指明该模块所在的上下级环境。
- 软件过程遵从程序结构的主从关系，因此它也是层次化的。



软件体系结构简介

- 对于大规模的复杂软件系统来说，系统的总体结构设计和规格说明比数据结构和算法的选择重要的多。在这种背景下，人们认识到软件体系结构的重要性。
- 软件体系结构研究的主要内容涉及软件体系结构描述、软件体系结构风格、软件体系结构评价和软件体系结构的形式化方法等。
- 其根本目的是要解决软件重用、软件质量和软件维护问题。

软件体系结构定义

- Booch & Rumbaugh & Jacobson 定义
  - 软件体系结构 =
    {组织，元素，子系统，风格}
- Bass定义
  - 是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）

- **Shaw定义**
  - 结构模型：软件体系结构由构件、构件之间的连接和一些其它方面组词组成：
  - 框架模型：其重点在于整个系统的连贯结构，这与重视其组成恰好相反。框架模型常常以某种特定领域或某类问题为目标。
  - 动态模型：动态模型强调系统的行为质量。它可以指整个系统配置的变化，也可以是禁止预先激活了的通信或交互，还可以使计算中表现的动态特性等。
  - 过程模型：过程模型关注系统结构的构建及其步骤和过程。在这一观点下，体系结构是软件开发所进行的一系列过程的结果。

软件体系结构定义

- **Garlan & Shaw 模型**
  - **软件体系结构 = {构件，连接件，约束}**
    - 构件可是一组代码，也可是一个独立的程序，构件是一组对象集合，可实现某些计算逻辑。
    - 构件相对独立，仅通过接口与外部进行交互，可作为独立单元嵌入到不同的应用系统中。
    - 连接件可以是过程调用、管道、远程过程调用等，用于表示构件之间的相互作用。
    - 约束一般为对象连接时的规则，或指明构件连接的条件。
  - 他们认为软件体系结构是软件设计过程的一个层次，这一层次超越计算过程中的算法设计和数据结构设计。
  - 体系结构问题包括总体组织和全局控制、通信协议、同步、数据存取，给设计元素分配特定功能，设计元素的组织、规模和性能，在各设计方案间进行选择等。

软件体系结构三要素

- 软件设计的一个目标是建立软件的体系结构表示
- 软件体系结构的三要素是：
  - 程序构件（模块）的层次结构
  - 构件之间交互的方式
  - 数据的结构
- 在软件体系结构设计中应保持的几个性质：
  - 结构
  - 附属功能
  - 可复用

软件构件的分类与调用方式

构 件	特 点 和 示 例
纯计算构件	具有简单的输入/输出关系，没有运行状态的变化。例如，数值计算、过滤器（Filters）、转换器（Transformers）等。
存储构件	存放共享的、永久性的、结构化的数据。例如，数据库、文件、符号表、超文本等。
管理构件	执行的操作与运行状态紧密耦合。例如，抽象数据类型（ADT）、面向对象系统中的对象、许多服务器（Servers）等。
控制构件	管理其它构件运行的时间、时机及次序。例如，调度器、同步器等。
链接构件	在实体之间传递信息。例如，通信机制、用户界面等。

连 接	特 点 与 示 例
过程调用	在某一个执行路径中传递执行指针。例如，普通过程调用（同一个命名空间）、远程过程调用（不同的命名空间）。
数据流	相互独立的处理通过数据流进行交互，在得到数据的同时被赋予控制权限。例如，UNIX系统中的管道（pipes）。
间接激活	处理是因事件的发生而激活的，在处理之间没有直接的交互。例如，事件驱动系统、自动垃圾回收等。
消息传递	相互独立的处理之间有明确的交互，通过显式的离散方式的数据传递。这种传递可以是同步的，也可以是异步的。例如，TCP/IP。
共享数据	构件们通过同一个数据空间进行并发的操作。例如，多用户数据库、数据黑板系统。

软件体系结构风格

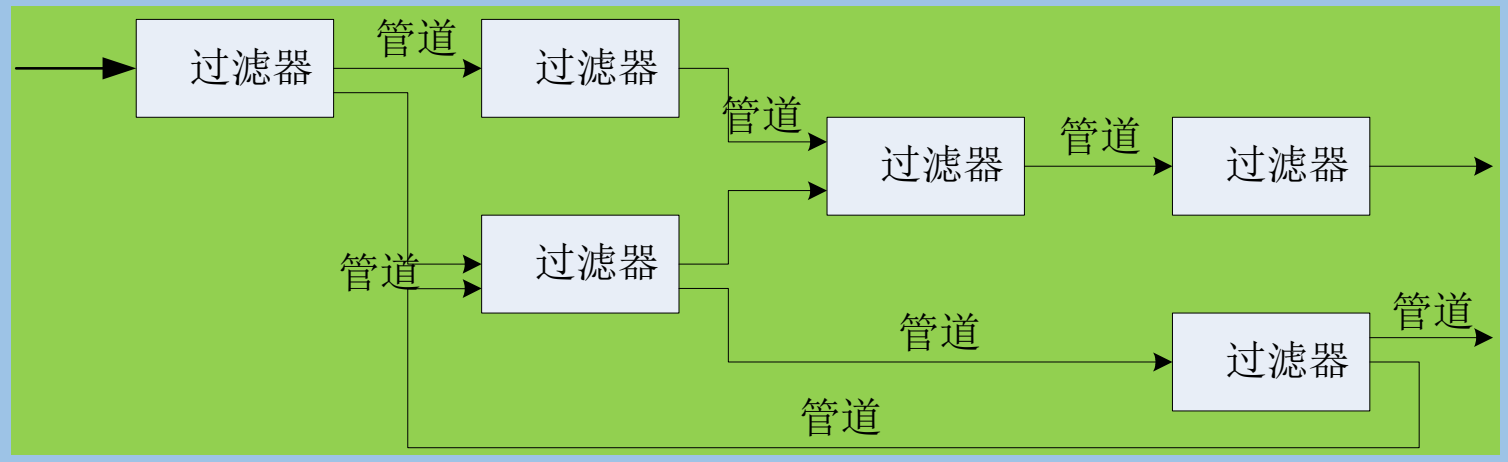
- 软件体系结构设计的一个核心问题是能否使用重复的体系结构模式。
- 基于这个目的，学者们开始研究和实践软件体系结构的风格和类型问题。
- 软件体系结构风格是描述某一特定应用领域中系统组织方式的惯用模式。

体系结构风格的四要素

- 体系结构风格具有四个主要元素，即提供一个词汇表、定义一套配置规则、定义一套语义解释原则和定义对基于这种风格的系统所进行的分析。
  - 反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整的系统。
  - 对软件体系结构风格的研究和实践促进了对设计的重用，一些经过实践证实的解决方案也能可靠地用于解决新的问题。体系结构风格的不变部分使不同的系统可以共享同一个实现代码。

管道和过滤器风格

- 在管道 / 过滤器（pipe and filter）风格最早出现在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。
  - 构件：在管道和过滤器风格中，构件被称为过滤器（filter）。
  - 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。
  - 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）。



调用和返回风格

- 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。
  - 主/子程序风格的体系结构：
  - 对象风格的体系结构：
  - 分层风格的体系结构：

主程序/子程序风格

- 主要目的是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。该风格的特点：
  - 由于单线程控制，计算的顺序得以保障。
  - 并且有用的计算结果在同一时刻只会产生一个。
  - 单线程的控制可以直接由程序设计语言来支持
  - 分层推理机制：子程序的正确性与它调用的子程序的正确性有关。

对象风格

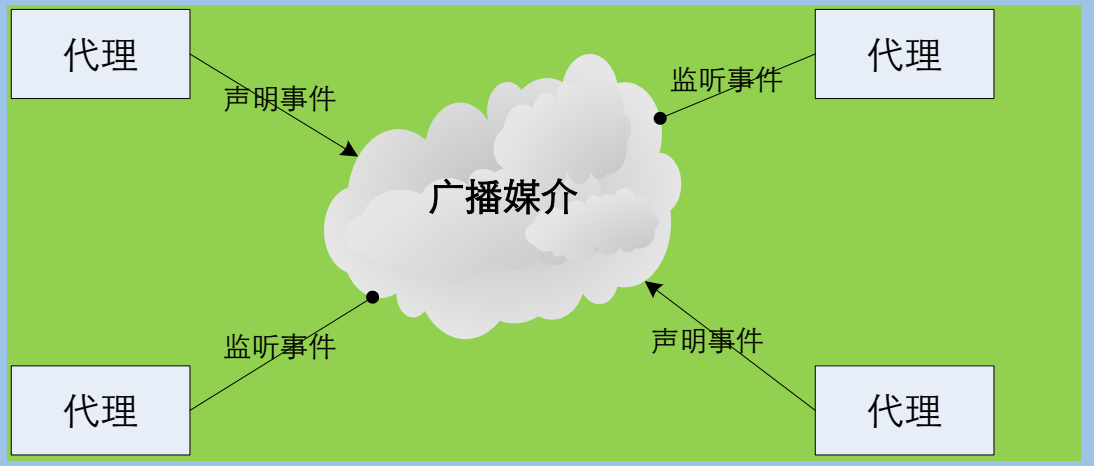
- 这种风格建立在数据抽象和面向对象的基础上，数据的表示方法和它们的相应操作封装在一个抽象数据类型或对象中。
- 对象是一种被称作管理者的构件，由它负责保持资源的完整性。对象是通过函数和过程调用来交互的。
  - 对象抽象使得构件和构件之间的操作以黑箱的方式进行。
  - 封装性使得细节内容对外部环境得以良好的隐藏。对象之间的访问是通过方法调用来实现的。
  - 考虑操作和属性的关联性，封装完成了相关功能和属性的包装，并由对象来对它们进行管理。
  - 使用某个对象提供的服务并不需要知道服务内部是如何实现的

分层风格

- 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。

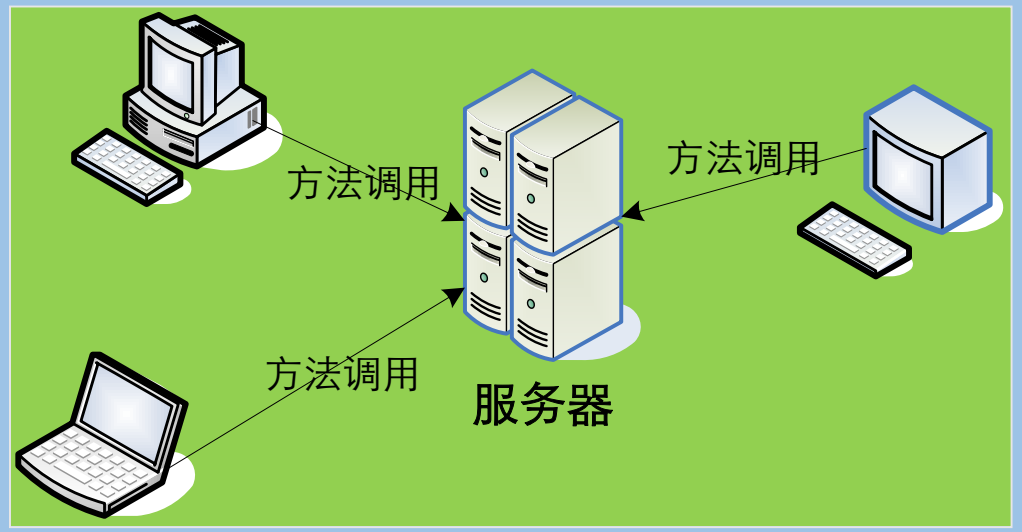
基于事件风格

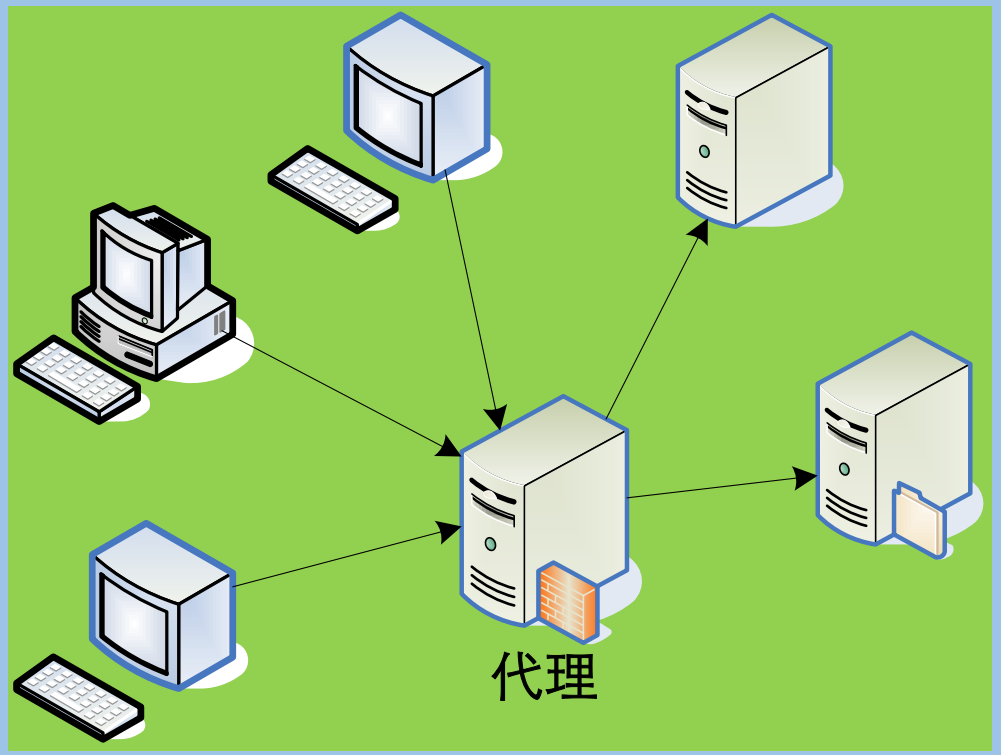
- 构件不直接调用一个过程，而是声明或广播一个或多个事件。适用于设计低耦合构件集合的应用程序，每个构件完成一定的操作，并可能触发其他构件的操作。
- 构件的接口不仅提供一个过程的集合，也提供一个事件的集合。这些过程既可以用一般的方式调用，也可能被注册为与某些事件相关。
- 构件可以声明或广播一个或多个事件，或者向系统注册用以表明它希望相应一个或多个事件。
-



客户端/服务器风格

- 客户端/服务器风格设计的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。





解释器(虚拟机)风格

- 该风格通常用于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。
- 由于解释器实际上创建了一个软件虚拟出来的硬件机器，所以被称为虚拟机风格。
- 这种风格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
  - 程序设计语言的编译器，如Java，Smalltalk等。
  - 基于规则的系统，比如专家系统领域的Prolog等。
  - 脚本语言，比如Awk，Perl等

仓库风格

- 仓库风格的体系结构由两种构件组成：
  - 中央数据结构，表示当前状态；
  - 独立构件的集合，它对中央数据结构进行操作。
- 传统的数据和状态控制方法
  - 由输入事务选择进行何种处理，并把执行结果作为当前状态存储到中央数据结构中，此时仓库是一个传统的数据库体系结构；
- 黑板体系结构
  - 由中央数据结构的当前状态决定进行何种处理

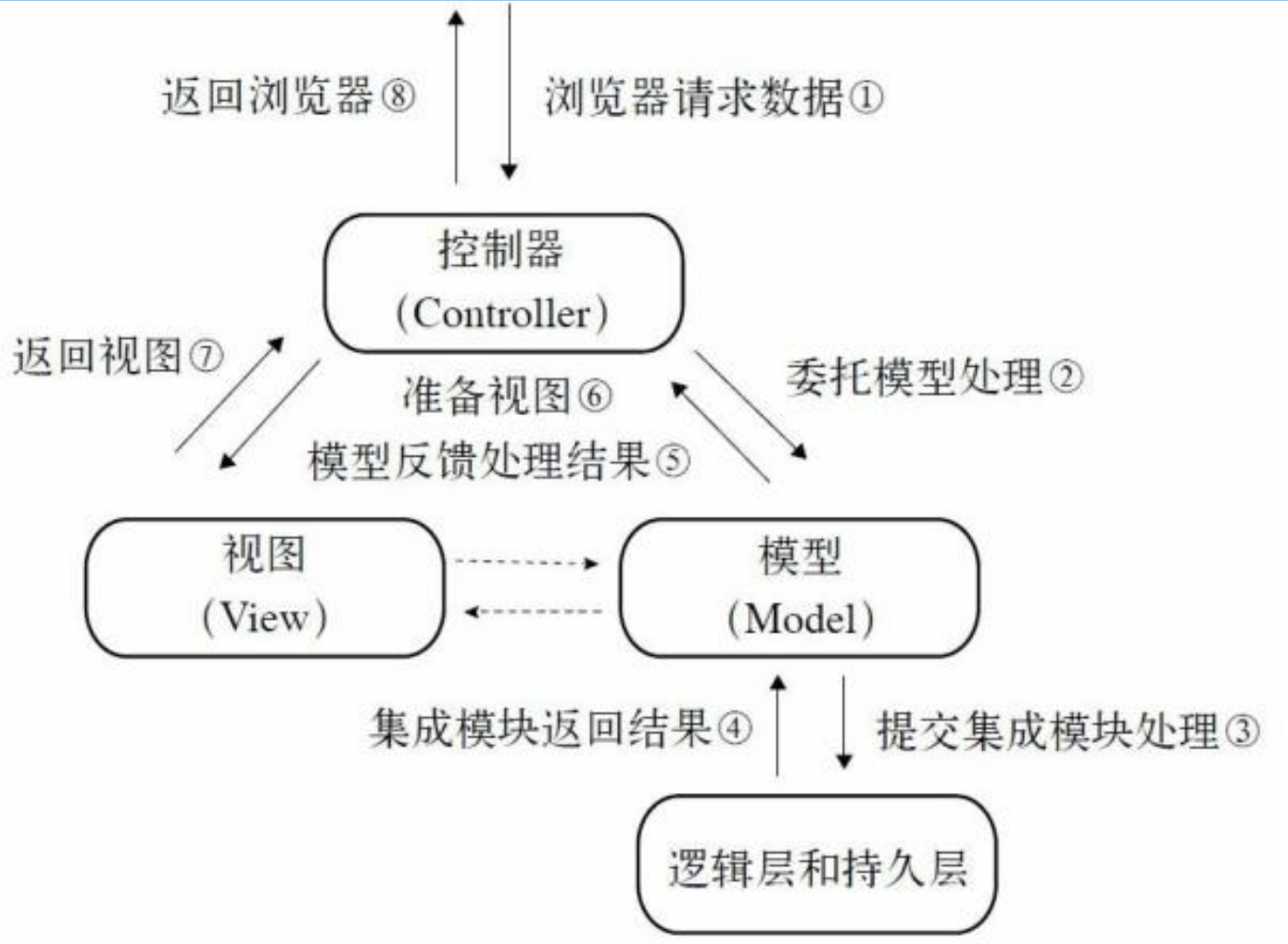
黑板风格

- 黑板系统通常被用于在信号处理方面进行复杂解释的应用程序，以及松散的构件访问共享数据的应用程序。
  - 知识源：是特定应用程序知识的独立散片。知识元之间的交互只在黑板内部发生。
  - 黑板数据结构：知识源不断地对黑板数据进行修改，直到得出问题的解答。黑板数据结构起到了知识源之间的通信机制的作用。
  - 控制器：控制是由黑板的状态决定的。一旦黑板数据的改变使得某个知识源成为可用的，知识源就会被控制模块激活。

MVC

- MVC是Xerox PARC在二十世纪八十年代为编程语言Smalltalk-80发明的一种软件设计模式，后来被推荐为Java EE平台的设计模式。
- 模型-视图-控制器风格通常简称为MVC（Model-View-Controller）风格。
- 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。
  - 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。
  - 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。
  - 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。

MVC运行机制示意


返回浏览器⑧
浏览器请求数据①
控制器
(Controller)
返回视图⑦
准备视图⑥
模型反馈处理结果⑤
委托模型处理②
视图
(View)
模型
(Model)
集成模块返回结果④
提交集成模块处理③
逻辑层和持久层