Software Architecture

- principle and practice

第二讲　软件体系结构风格

- 一、什么是软件体系结构风格（Architecture Styles）
  - 1.什么是结构风格（Architecture Styles）
    - 在许多工程学科，设计模式、风格的利用是非常普遍的。事实上，衡量一个工程领域是否成熟的一个重要指标，是看该工程领域是否就建就一套共享的通用设计形式。在软件上也是这样。在结构上，与之联系的有，客户/服务器系统(client-server system，C/S)， 管道/过滤器设计(pipe-filer design，P/F), 层次结构（layered architecture),在设计方法上的面向对象（object-oriented)、数据流（dataflow)等。
    - 在软件设计的实践中，人们发现某些特殊组织结构（如C/S结构，B/S结构、P/F结构）在许多的软件系统中频繁出现，这些特殊结构有很高的应用价值。于是人们就设想能否将这些特殊结构进一步发展规范，使它们成为一种相对固定的设计结构，并应用于新软件产品。
    - 关于一个软件系统的结构风格定义为组织该系统可用的各种结构模式。具体地说，就是系统的组件（component)、连接器(connector)，以及它们的组合的约束条件(constraint)。
    - 一个软件系统的结构包括构成系统的所有计算组件（computational component)、组件间相互作用(interacton)即连接器(connector)。用数学中的图的概念描述，一个系统结构图就是由节点(node)和边(edge)组成，这里节点代表组件，边表示组件间的连接器。
  - 2.体系结构风格的分类
    - Garlan和Shaw对通用体系结构风格的分类
      - 数据流系统（Dataflow systems)
        - 批处理序列(Batch sequential)
        - 管道/过滤器(Pipes and filters)
      - 调用/返回系统(Call-return systems)
        - 主程序/子程序(Main program and subroutine)
        - 面向对象(OO system)
        - 层次结构(Hierarchical layers)
      - 独立组件(Independent components)
        - 进程通讯(Communicating processes)
        - 事件系统(Event systems)　　;
      - 虚拟机(Virtual machines)
        - 解释器(Interpreters)
        - 基于规则的系统(Rule-based systems)
      - 仓库(Repositories)
        - 数据库系统(Databases)
        - 超文本系统(Hypertext system)
        - 黑板系统(Blackboards)
- 二、常用结构风格（Architecture Styles)
  - 1.管道/过滤器（Pipes and Filters,P/F)
    - 系统组织
      - 在一个管道/过滤器组织的系统中，每个组件有一个输入集(input set)和一个输出集(output set)，并从输入集中读取数据流，处理后产生的数据流送向输出集。组件称为过滤器（Filters），数据流称为管道（Pipes）。过滤器间按预定的顺序工作，即一个过滤器的的输出作为下一个过滤器的输入。从第一个过滤器的输入开始的，数据被逐步地处理直到完成。
      - 从整个系统的输入和输出关系看，各过滤器可以对其输入进行局部的独立处理并产生部分的计算结果，过滤器的活动可以按以下方法激活:
        - 后续的组件从过滤器中取出数据;
        - 前续的组件向过滤器推入新数据;
        - 过滤器处于活跃状态，不断地从前续组件取出、并向后续组件推入数据。
      - 前两种情况产生的是被动式过滤器(passive filter),最后的是主动式过滤器（active filter)。被动式过滤器是通过函数或过程调用的，而主动式过滤器是作为独立的程序或线程任务激活的。
      - 管道（Pipe)是过滤器之间的连接器(Connector)，如果两个主动式过滤器连接在一起，管道将对它们实施同步控制，管道是一个先进先出(FIFO)的数据缓冲区。

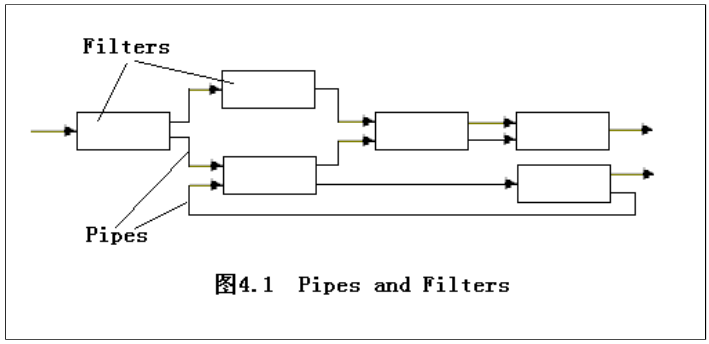


图4.1 Pipes and Filters

- 过滤器是独立的运行组件。除了输入和输出外，其独立性具体表现在：
  - 过滤器（filters)之间不受任何其他的过滤器运行的影响，非相邻的过滤器之间不共享任何状态，甚至对于多次加工而言，过滤器自身也是无状态的即每次加工后回到原始的等待状态；
  - 一个过滤器对其前继或后续过滤器的设计和使用没有任何的限制，惟一关心的是其输入的到来形式、加工处理的逻辑和产生的输出形式；
  - 在整个结果的正确不依赖于各个过滤器运行的先后次序。对于原始的输入，尽管其最终输出形式的获得需要经过特定的加工、并符合加工的顺序要求，但在系统工作时，各过滤器只在输入具备后独立完成自己的计算，完整的计算过程包含在各个过滤器之间的拓扑结构（topology)中。
- 过滤器的拓扑结构（topology)
  - 线性顺序的管道（linear sequences filters)
  - 界限管道（bounded pipes): 对管道中的驻留的数据的限制。
  - 类型管道（typed pipes)：对两个过滤器间的数据有严格的类型定义。
- 优点（advantage)
  - 使得软构件具有良好的隐蔽性和高内聚、低耦合的特点；
  - 允许设计者将整个系统的输入/输出行为看成是多个过滤器的行为的简单合成；
  - 支持软件重用。重要提供适合在两个过滤器之间传送的数据，任何两个过滤器都可被连接起来；
  - 支持快速原型系统的实现。
  - 具有自然的并发特性。过滤器可以独立顺序运行，也可以同时并发运行，从而增加了系统运行的灵活性，也使提高运行效率成为可能，尤其是网络环境下的管道。
  - 由于是通过独立的过滤器的组合，系统具有清晰的拓扑结构，因而容易进行某些性能方面的分析，例如，数据吞吐量(throughput)、死锁(deadlock)、计算正确性等。
- 缺点(disadvantage)
  - 通常导致进程成为批处理的结构。这是因为虽然过滤器可增量式地处理数据，但它们是独立的，所以设计者必须将每个过滤器看成一个完整的从输入到输出的转换。。
  - 不适合于需要共享大量数据的应用设计。
  - 不适合处理交互的应用。当需要增量地显示改变时，这个问题尤为严重。
  - 过滤器之间通过特定格式的数据进行工作，数据格式的设计和转换是系统设计的主要方面，为了确保过滤器的正确性，必须对数据的句法和语义进行分析，这增加了过滤器设计的复杂性。
  - 并行运行获得高效率往往行不通。原因，第一，独立运行的过滤器之间的数据传送的效率会很低，在网络传送时尤其如此；第二，过滤器通常是在消耗了所有输入后才产生输出；第三，在单处理器的机器上进程的切换代价是很高的；第四，通过管道对过滤器的同步控制可导致频繁启动和停止过滤器工作。
- 2.数据抽象与面向对象组织（Data abstraction and Object-oriented Organization)
  - 系统组织
    - 抽象数据类型概念对软件系统有着重要作用，目前软件界已普遍转向使用面向对象系统。这种风格建立在数据抽象和面向对象的基础上，数据以及施加于它们之上的操作封装在一个抽象数据类型或对象中。这种风格的系统中组件（component)是对象(object)，或者说是抽象数据类型的实例。对象是一种被称作管理者(manager)的组件，因为它负责资源的表示（representation)。对象间的相互作用是通过函数(founction)和过程(procedure)调用（invocation)实现。连接器（connector)则是激活对象方法（method)的消息（message)。

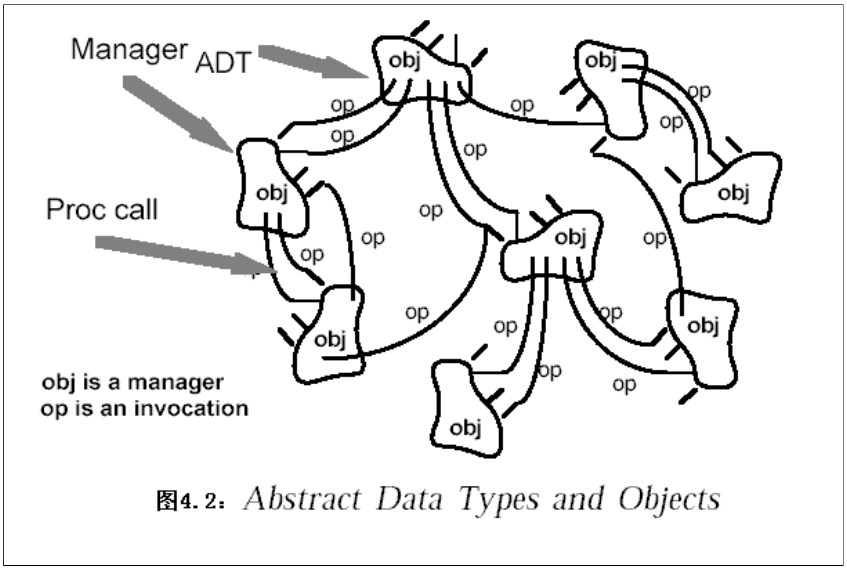


图4.2：*Abstract Data Types and Objects*

- 优点
  - 因为对象对其它对象隐藏它的表示，所以可以改变一个对象的表示，而不影响其它的对象。
  - 设计者可将一些数据存取操作的问题分解成一些交互的代理程序的集合。
  - 允许多个对象实现并发（concurrent)任务，对象间可有多种接口。
- 缺点
  - 为了使一个对象和另一个对象通过过程调用等进行交互，必须知道对象的标识。只要一个对象的标识改变了，就必须修改所有其他明确调用它的对象。与此相比，管道/过滤器系统中一个过滤器就不必知道另一个过滤器而只需将处理结果按一定的形式输出即可。因此，在面向对象（object-oriented)的系统中,一个对象（组件）的标识的改变，需要修改所有与之相互的对象（组件）。
  - 必须修改所有显式(explicitly)调用它的其它对象，并消除由此带来的一些副作用（side-effect)。例如，如果A使用了对象B，C也使用了对象B，那么，C对B的使用所造成的对A的影响可能是料想不到的。

- 3.事件及隐含激活（Event-Based, Implicit Inovaations)
  - 系统组织
    - 在面向对象的系统中，组件是一组对象，组件间的相互关系是通过直接的过程调用实现，这种方法的不足是组件（对象）标识必须是公开的，众所周知（well-known)的。因此，人们考虑是否可以隐含激活（implicit invocation)或作用集成（reactive integration)。
    - 在一个基于事件的隐含激活系统中，组件不直接调用一个过程，而是触发(announce)或广播(broadcast)一个或多个事件（event)。系统中的组件事先注册（register)它们感兴趣的事件及对应的过程，将来，当一个事件被触发（inovaation），系统自动调用在这个事件中注册的所有过程，这样，一个事件的触发就导致了另一模块中的过程的调用。
    - 从体系结构上说，这种隐含激活风格的组件是一些模块，这些模块是一些过程和一些事件的集合。过程可以用通常的方式调用，也可以在系统事件中注册一些过程，当发生这些事件时，过程被隐含调用。
    - 基于事件的隐含激活风格的主要特点是事件的触发者并不知道哪些组件会被这些事件影响。这样不能假定组件的处理顺序，甚至不知道哪些过程会被调用，因此，许多隐含调用的系统也包含显式调用作为组件交互的补充形式。
    - 支持基于事件的隐含激活的应用系统很多。例如，在数据库管理系统中确保数据的一致性(consistency)约束，在用户界面系统中管理数据，以及在编辑器中支持语法检查。例如在编程环境中，编辑器和变量监视器可以登记相应Debugger的断点(breakpoint)事件。当Debugger在断点处停下时，它声明该事件，由系统自动调用处理程序，程序代码自动滚动（scroll)到到断点，变量监视器刷新变量数值。而Debugger本身只声明事件，并不关心哪些过程会启动，也不关心这些过程做什么处理。
  - 优点
    - 为软件重用(reuse)提供了强大的支持。当需要将一个组件加入现存系统中时，只需将它注册到系统的事件中。
    - 为扩充系统带来了方便。当用一个组件代替另一个组件时，不会影响到其它组件的接口。
  - 缺点
    - 组件放弃了对系统计算的控制。一个组件触发一个事件时，不能确定其它组件是否会响应它。而且即使它知道事件注册了哪些组件的构成，它也不能确保这些过程被调用的顺序。
    - 数据交换的问题。有时数据可被一个事件传递，但另一些情况下，基于事件的系统必须依靠一个共享的仓库进行交互，在这些情况下，全局性能和资源管理便成了问题。
    - 难以保证正确性。
- 4.层次系统（Layers Systems)
  - 系统组织
    - 层次系统组织成一个层次结构，每一层为上层提供若干服务(service)，并作为下层的客户（client)。在一些层次系统中，除了一些精心挑选的输出函数外，内部的层只对相邻的层可见。这样的系统中组件在一些层实现了虚拟机（在另一些层次系统中层是部分不透明的）。连接器通过决定层间如何交互的协议来定义，拓扑约束限制了对相邻层间交互。

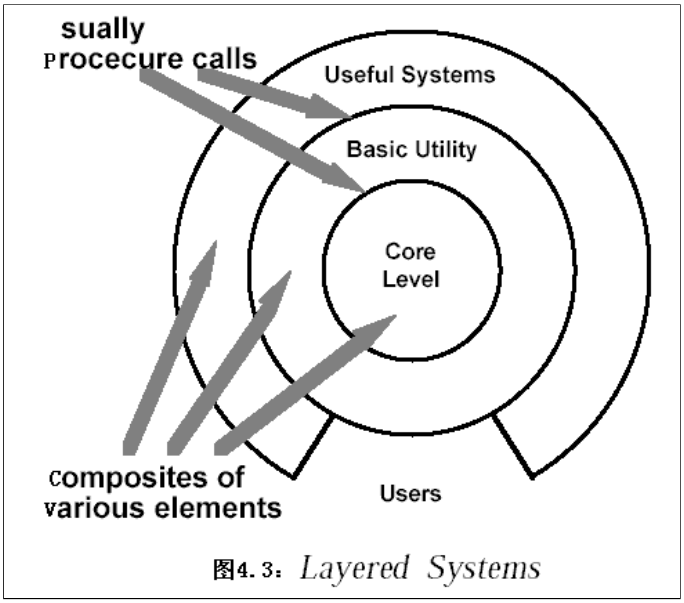


图4.3：*Layered Systems*

- 层次系统（Layers Systems)最广泛的应用是分层通信协议（如，OSI/RM)。在这一应用领域中，每一层提供一个抽象的功能，作为上层通信的基础。较低的层次定义低层的交互，最低层通常只定义硬件物理连接。

- 优点
  - 支持基于层递增的系统设计，使设计者可以把一个复杂系统按递增的步骤进行分解。
  - 支持(enhancement)功能扩充，与管道/过滤器系统一样，因为每一层至多和相邻的上下层交互，因此功能的改变最多影响相邻的上下层。
  - 支持重用（reuse)。只要提供的服务接口定义不变，同一层的不同实现可以交换使用。这样，就可以定义一组标准的接口，而允许各种不同的实现方法。
- 缺点
  - 并不是每个系统都可以很容易地划分为分层的模式，甚至即使一个系统的逻辑结构是层次化的，出于对系统性能的考虑，系统设计时不得不把一些低级或高级的功能综合起来。
  - 很难找到一个正确层次的抽象方法。如，在通信协议中，虽然OSI/RM定义了7层协议，可是TCP/IP，IPX/SPX也都不是7层。

- 5.仓库(Repositories)
  - 系统组织
    - 在仓库(Repositories)风格的系统中，有两种不同的组件：中央数据结构表示当前状态，一组独立组件执行在中央数据上，仓库与外部组件间的相互作用在系统中会有大的变化。
    - 控制原则（control discipline)的选取导致这种风格的两个主要的子类。如果由输入的事务类型触发进程执行，则仓库风格的系统是一种传统的数据库系统；如果由中央数据结构的当前状态触发进程执行，则仓库是一个黑板系统。

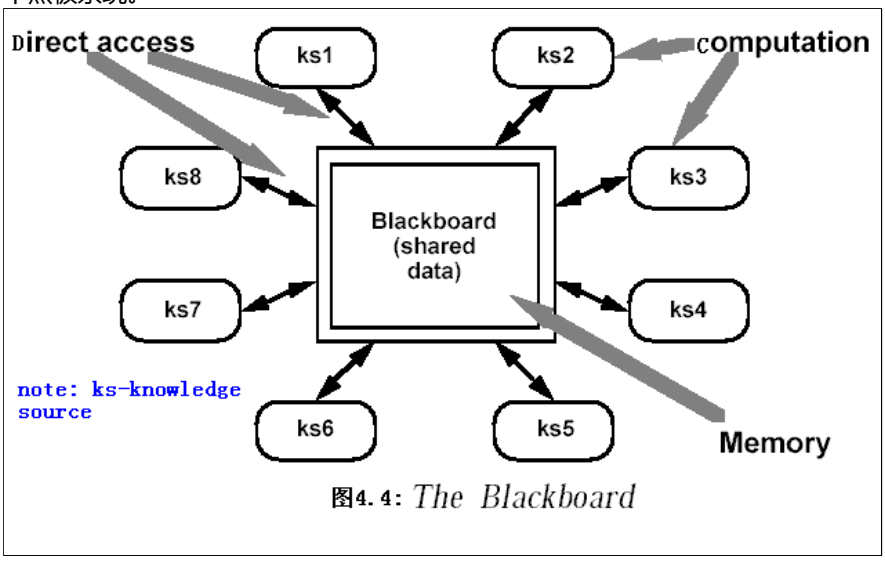


图4.4：*The Blackboard*

    - 黑板系统实现的基本出发点是已经存在一个对公共数据结构进行协同操作的独立程序集合。每个程序专门解决一个子问题，但需要协同工作完成整个问题的求解，这些专门的程序是相互独立的，它们之间不存在互相调用，也不存在右事先确定的操作顺序，操作次序是由问题求解的进程状态决定的。
    - 在黑板系统中，有一个中心操作组件，即黑板，它是一个数据驱动或状态驱动的控制机制。它保存着系统的输入、问题求解各个阶段的中间结果和反映整体问题求解进程的状态，这些是由系统的输入和各个求解程序写入的，因此被称为黑板。
    - 在问题求解过程中，黑板上保存了所有部分解，它们代表了问题求解的不同阶段。形成了问题的可能解空间，并以不同的抽象层次表达出来，其中，最底层的表达就是系统的原始输入，最终的问题求解在抽象的最高层

次。
- 黑板系统主要由三部分组成
  - 知识源（knowledge source)
    - 知识源中包含独立的、与应用的问题求解相关的知识(knowledge)，知识源之间不直接进行通讯，它们之间的交互只通过黑板来完成。
    - 知识源包含参与问题求解的条件的执行的操作，条件部分对黑板的信息和求解进程的状态做出评估，在条件得到满足时执行相应的操作；执行的操作可以是产生新的假设，也可以是对黑板上的数据结构的变换处理，操作的结果可能导致黑板上数据和状态的变化，并引起进一步的处理。
  - 黑板数据结构(blackboard data structure)
    - 问题求解(problem-solving)的状态数据（state data)，是按照与求解进程的相关的层次来组织。通过黑板中知识源的不断改变来体现一个问题的逐步解决。
  - 控制(control)
    - 控制黑板的数据和状态的变化，并根据变化决定采取的行动。黑板状态的改变决定使用的特定知识。
    - 其中有一类特殊的知识源，它们用来确定问题求解的最终目标和终止求解的条件。
- 黑板系统的传统应用是信号处理领域，如语音和模式识别（speech and pattern recognition)。另一应用是松耦合代理(loosely coupled agent)数据共享存取。一些管道/过滤器系统（如编译器）也可以设计为黑板系统。另个黑板系统也是人工智能广泛使用的系统结构。

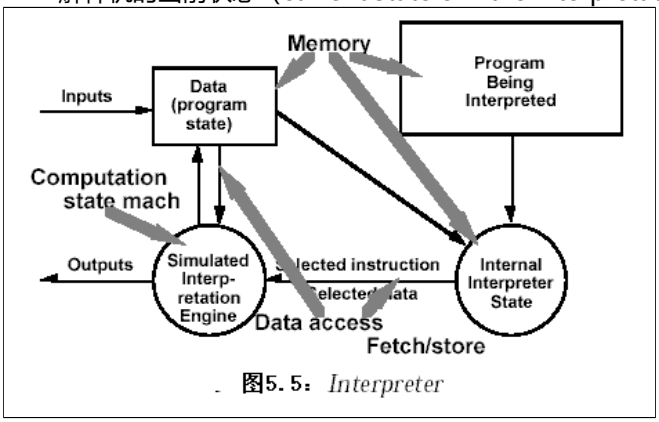
- 6.解释器（Interpreters)
  - 系统组织
    - 解释器（interpreters)广泛用于建立虚拟机（virtual machine),以减少程序的语义所期望的计算机器与硬件所提供的有效计算机器的差距。JVM - java语言虚拟机。
    - 一个解释器系统由一个程序伪代码(pseudoprogram)和解释机（interpretation engine)。其中程序伪代码包含程序代码和执行的解释中间代码，解释机包含解释器的定义和执行的当前状态的定义。
    - 解释器的由个部件组成
      - 解释机（interpretation engine)：完成解释工作；
      - 程序源码：待解释的程序；
      - 伪代码（pseudocode): 用来解释的中间代码。
      - 解释机的当前状态（current state of　the interpretation engine)。
    - 

图5.5：*Interpreter*

三、其他结构风格

- 1.分布式处理（Distributed Processes)
  - 分布式系统（distributed systems)多处理器系统的一个组织，客户/服务器（client-server,C/S）结构则是组织分布式系统的主要方式。服务器（server)表现为一个进程,它向其他进程（client)提供服务或数据，通常服务器事先不知道访问它的客户的数量和标识(identity)，客户（client)则通过远程过程调用（RPC）或消息传递（message passing)方式请求服务。
  -
- 2.主程序/子过程调用（Main program/subroutine organiztions)
  - 系统由一个主程序（main program）和一组子过程（subroutines）组成，由主程序驱动调用所需要的子过程，通常主程序是运行一个循环顺序地调用各个子过程。
- 3.确定域结构(Domain-specific)
  - 一种分布式系统的实现方式，服务程序用域（domain)管理，提供对象引用（reference)给客户程序使用。
- 4.状态变迁系统（State transition system)
  - 系统有一组状态(state)集、一组状态转换函数，其中状态集包括初始状态集和终止状态。系统接受一个输入集，根据初始状态依次驱动对应的状态转换函数。

四、案例分析（Case Studies)

- 以上，我们介绍了几种软件体系结构风格，不同的风格有不同的处理能力、优点和弱点，要为系统选择或设计某一个体系结构风格，必须根据特定项目的具体特点，进行分析比较后再确定，体系结构风格的使用几乎完全是特化的。下面，我们引用分析Shaw和Garlan教材中的一个案例。

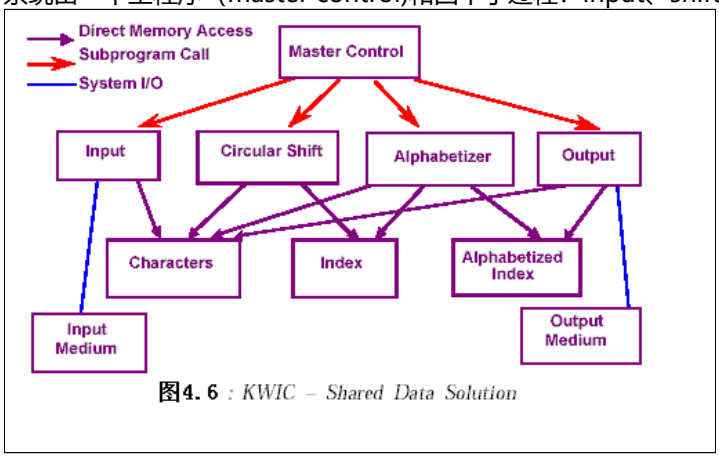
- 1.KWIC问题描述
  - KWIC - Key Word Index in Contex（关键字索引）,由Parnas在1972年提出的，问题描述如下：
  - The KWIC [Key Word in Context] index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a list of all circular shifts of all lines in alphabetical order.
  - 例子
顺序输入	输出结果
Pipes and Filters Architectures for Software Systems	and Filters Pipes Architectures for Software Systems Filters Pipes and for Software Systems Architectures Pipes and Filters Software Systems Architectures for Systems Architectures for Software
- 2.方法1：主程序/子过程调用（Main program/subroutine）
  - 系统由一个主程序（master control)和四个子过程：input、shift、alphabetize和output组成。
  - 

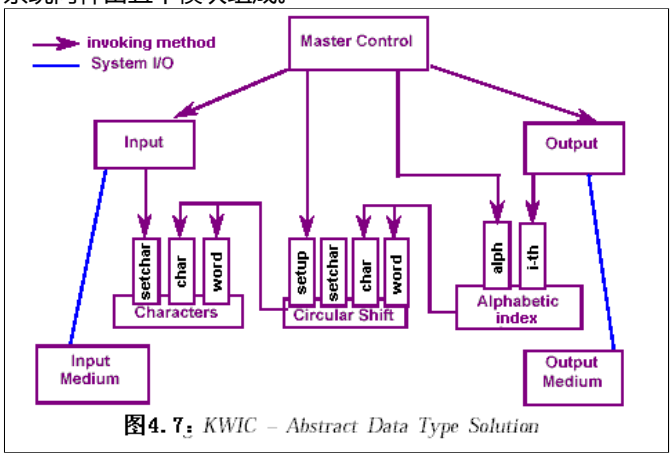
    图4.6：KWIC – Shared Data Solution
  - 四个子过程在主程序的控制下依次执行，子过程间的数据传递是通过共享的存储区（磁盘)。
  - 在这种方法中，子程序间共享存储区域，使得数据的表示效率高，节省了存储空间；但这也带来一些不足，数据结构的改变将影响到所有的四个子过程，不支持重用（reuse)。
- 3.方法2：数据抽象与面向对象组织（Data abstraction and Object-oriented Organization)
  - 系统同样由五个模块组成。
  - 

    图4.7：KWIC – Abstract Data Type Solution
  - 但模块间不再直接共享数据存储区，每个模块定义了一些方法，供其他模块通过激活方法调用。
  - 在这种方法中，算法和数据表示封装在模块内，对它们的修改不影响其他模块；支持重用。但不适合于功能的扩充，当增加一个新功能时，必须修改现有的模块。
- 4.风格3：事件及隐含激活（Event-Based, Implicit Inovaations）
  - 33

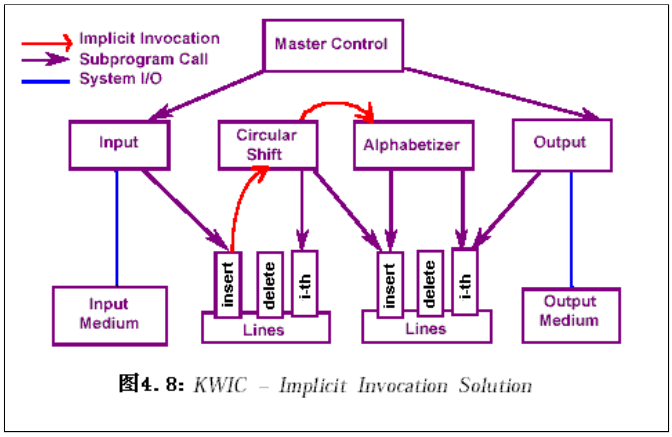


图4. 8: *KWIC – Implicit Invocation Solution*

-
-

- 5.风格4：管道/过滤器（Pipes and Filters,P/F)
  - 3

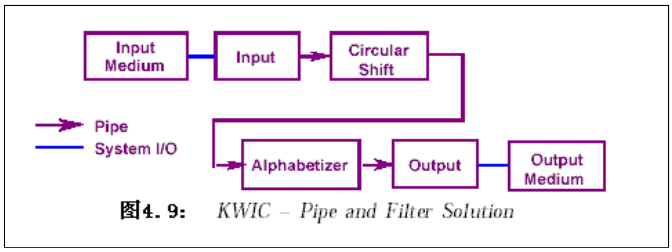


图4. 9: *KWIC – Pipe and Filter Solution*

  -
  - 8

返回

==========