**经典软件体系结构案例KWIC**

**案例研究：KWIC（上下文中的关键词）**

在他1972年的论文中，Parnas提出了以下问题：

KWIC[上下文中的键Word]索引系统接受一组有序的行，每一行都是一组有序的单词，每个单词都是一组有序的字符。任何一行都可以通过重复移除第一个单词并在行的末尾附加它，从而进行``循环移位。KWIC索引系统按字母顺序输出所有行的所有循环移位的列表。

Parnas利用这个问题来对比将系统分解为模块的不同标准。他描述了两种解决方案，一种基于具有对数据表示的共享访问的函数分解，另一种基于隐藏设计决策的分解。自引入以来，这一问题已成为人们众所周知的问题，并被广泛应用于软件工程的教学设备。加兰、凯泽和Notkin也使用这个问题来说明基于隐式调用的模块化方案。

虽然KWIC可以作为一个相对较小的系统来实现，但它不仅仅是教学兴趣。它的实际实例被计算机科学家广泛使用。例如，Unix Man页面的“排列”索引本质上就是这样一个系统。

从软件体系结构的角度来看，这个问题的吸引力在于它可以用来说明变化对软件设计的影响。Parnas表明，不同的问题分解在承受设计变化的能力上有很大的差异。他认为其中的变化包括：

处理算法的更改：例如，可以在从输入设备读取每行时，在读取所有行之后，或在字母排序需要一组新的移位行时按需执行行移位。

数据表示中的变化：例如，可以以各种方式存储行。类似地，循环位移可以显式或隐式存储（作为索引和偏移量对）。

加兰、凯泽和诺特金扩展了Parnas的分析，考虑到：

系统功能增强：例如，修改系统，使移线以消除以某些噪声词（如“a”、“an”、“和”等）开始的圆移。将系统更改为交互式系统，并允许用户从原始（或从循环移位）列表中删除行。

性能：空间和时间。

重用：组件在多大程度上可以作为可重用的实体。

我们现在概述了KWIC系统的四种架构设计。这四个方案都是基于已发布的解决方案（包括实现）。前两个是在帕纳斯的原始文章中考虑到的。第三个解决方案是基于隐式调用风格的使用，并代表了由Garlan、凯撒和Notkin所检查的解决方案的一个变体。第四个是受Unix索引实用程序启发的管道解决方案。

在介绍了每个解决方案并简要总结了其优缺点之后，我们对比了按照上面详细列出的五个设计维度组织的表中不同的架构分解。

**解决方案1：具有共享数据的主程序/子例程**

第一个解决方案根据所执行的四个基本函数对问题进行分解：输入、移位、按字母顺序排列和输出。这些计算组件由一个主程序协调为子程序，该主程序依次通过它们进行排序。数据通过共享存储（“核心存储”）在组件之间进行通信。计算组件和共享数据之间的通信是无约束读取编写协议。这是由于协调程序保证了对数据的连续访问。

使用这个解决方案可以有效地表示数据，因为计算可以共享相同的存储空间。该解决方案也具有一定的直观吸引力，因为不同的计算方面也被隔离在不同的模块中。

然而，正如Parnas所指出的那样，它在处理变化的能力方面有许多严重的缺陷。特别是，数据存储格式的改变将会影响到几乎所有的模块。类似地，整体处理算法的变化和系统功能的增强也不容易适应。最后，这个装饰位置并不特别支持重用。

**解决方案2：抽象的数据类型**

第二个解决方案将系统分解成一组相似的五个模块。然而，在这种情况下，数据不再由计算组件直接共享。相反，每个模块都提供了一个接口，它允许其他组件仅通过调用该接口中的过程来访问数据。（参见图7，它说明了每个组件现在如何有一组过程来确定系统中其他组件的访问形式。）

这个解决方案提供了与第一个解决方案相同的逻辑分解到处理模块中。然而，当考虑到设计更改时，它比第一个解决方案有许多优势。特别是，算法和数据表示都可以在单个模块中进行改变，而不影响其他模块。此外，它还比第一个解决方案更好地支持重用因为模块对与其交互的其他模块做的假设较少。

另一方面，正如加兰、凯泽和诺特金所讨论的那样，这个解决方案并不是特别适合用于增强。主要的问题是，要向系统添加新的功能，实现者或必须修改现有的模块——降低它们的简单性和完整性——或者添加新的导致性能损失的模块。

**解决方案3：隐式调用**

第三种解决方案使用了一种基于共享数据的组件集成形式。然而，这里有两个重要的区别。首先，到数据的接口更加抽象。将数据不是将存储格式公开给计算模块，而是抽象地访问（例如，作为列表或集合）。其次，在数据被修改时，系统会隐式地调用计算。因此，交互是基于主动数据模型的。例如，向线路存储中添加新线路的行为会导致事件发送到换档模块。这允许它产生循环移动（在一个单独的抽象共享数据存储中）。这反过来又会导致隐式地调用字母排序器，以便它可以按字母顺序排列这些行。

这个解决方案可以很容易地支持对系统的功能增强：通过将它们注册为在数据更改事件上调用，可以将附加模块附加到系统中。因为数据是被抽象地访问的，所以它还将计算与数据表示中的变化隔离开来。还支持重用，因为隐式调用的模块仅依赖于某些外部触发的事件的存在。这个解决方案可以很容易地支持对系统的功能增强：通过将它们注册为在数据更改事件上调用，可以将附加模块附加到系统中。因为数据是被抽象地访问的，所以它还将计算与数据表示中的变化隔离开来。还支持重用，因为隐式调用的模块仅依赖于某些外部触发的事件的存在。

但是，该解决方案是很难控制隐式调用模块的处理顺序。此外，由于调用是数据驱动的，这种分解的最自然的实现往往比以前考虑的分解使用更多的空间。

**解决方案4：管道和过滤器**

第四个解决方案使用了一个管道解决方案。在这种情况下，有四个过滤器：输入、移位、字母化和输出。每个过滤器都会处理这些数据，并将其发送到下一个过滤器。控件是分布式的：每个过滤器可以在任何时候它有要计算的数据时运行。过滤器之间的数据共享严格限制在在管道上传输的数据。

这个解决方案有几个很好的特性。首先，它保持了直观的处理流程。其次，它支持重用，因为每个过滤器都可以隔离地运行（只要上游过滤器以它期望的形式生成数据）。通过在处理序列中的适当点插入过滤器，可以很容易地将新的功能添加到系统中。第三，它支持易于修改，因为过滤器在逻辑上独立于其他过滤器。

另一方面，它也有一些缺点。首先，几乎不可能修改设计来支持交互式系统。例如，为了删除一行，必须有一些持久的共享存储，这违反了这种方法的一个基本原则。其次，该解决方案在使用空间方面效率很低，因为每个过滤器必须将所有数据复制到其输出端口。

**对比：**

可以通过将解决方案解决前面列出的设计考虑事项的能力进行比较。详细的比较必须考虑到有关系统的预期使用的一些因素：例如，它是批处理的还是交互式的、更新密集型的还是查询密集型的，等等。

图10根据前面介绍的建筑风格的讨论，提供了这种分析的近似值。正如Parnas所指出的，共享数据解决方案在对整体处理算法、数据表示和重用的更改的支持方面特别弱。另一方面，通过直接共享数据，它可以获得相对较好的性能。此外，添加新的处理组件（也访问共享数据）相对容易。抽象数据类型解决方案允许更改数据表示，并支持重用，而不一定会影响性能。但是，该解决方案中组件之间的交互是连接到模块本身中的，因此更改整体处理算法或添加新函数可能涉及对现有系统的大量更改。

隐式调用解决方案特别适合用于添加新的功能。然而，它存在共享数据方法的一些问题：对数据表示和重用的更改支持不足。此外，它可能会引入额外的执行开销。管道和过滤器解决方案允许在文本处理流中放置新的过滤器。因此，它支持处理算法的变化，功能的变化，和重用。另一方面，关于数据表示的决策将被连接到关于沿着管道传输的数据类型的假设中。此外，根据交换格式的不同，在解析和解解析管道上的数据时可能会涉及额外的开销。

