**KWIC解决方案与评价**

**管道和过滤器**把系统任务分成为几个独立的功能模块。这些功能模块采用通过系统的数据流连接。

在管道-过滤器风格下，每个功能模块都有一组输入和输出。功能模块称作过滤器。功能模块间的连接可以看作输入、输出数据流之间的通路，所以称作管道。

管道-过滤器风格的特性之一在于过滤器的相对独立性，即过滤器独立完成自身功能，相互之间无需进行状态交互。

优点：

设计者可以将整个系统的输入、输出特性简单的理解为各个过滤器功能的合成。

基于管道-过滤器风格的系统具有较强的可维护性和可扩展性。

支持一些特定的分析，如吞吐量计算和死锁检测等。

管道-过滤器风格具有并发性。

缺点：

管道和过滤器常常导致批处理方式；在维护两个分离但相关的数据流时，有一定的困难；可能对数据传输标准有一定的要求，这就导致了为每一个过滤器进行数据语法分析和编码的额外工作，同时，在编写过滤器时增加了复杂性，降低了性能。

**面向对象**

面向对象的程序设计其实是一种观念，主要是完成对象的封装、继承、多态等特性。在这种结构中，数据的表示和与之相连的属性的基本构造都被封装在一个抽象的数据类型或对象中。这种结构的部件就是对象，或者说是抽象数据类型的实例。对象之间通过函数和过程调用发生相互作用。这里所说的对象就是C++，JAVA中的类，都是封装了特定功能的属性和事件。

优点：

对象的隐藏并不影响它的使用者，更改它的实现，不会对系统造成任何危害。

将它们所操纵的数据与访问这些数据的操作绑在一起，是设计者易于将问题分解成相关对象之间的相互作用。

缺点：

为了实现某个对象与其他对象相互作用，必须知道对象的标志。这与管道形成了鲜明的对比； 会产生连锁反应。比如A调B，C调B ，那么A对B的调用可能会对C造成影响。

**分层系统**

分层系统将整个软件系统组成一个层次结构, 系统的每个中间层既要为上次提供服务, 又要调用下层的功能。

表示层是系统的用户接口部分, 由用户界面代码组成, 它担负着用户与应用间的对话功能, 使用功能层提供的服务, 不直接访问后台数据库。它用于检查用户从输入设备发出的输入的数据, 并且显示输出的数据。为使用户能直观地进行操作, 一般使用图形用户接口, 操作简单、易学易用。在变更用户接口时, 只需改写显示控制和数据检查程序, 而不影响其他两层。检查的内容也只限于数据的形式和取值的范围, 不包括有关业务本身的处理逻辑。

功能层相当于软件系统的本体, 它是将具体的业务处理逻辑编入程序中。对于较大的系统, 功能层可以进一步分成若干层。表示层和功能层之间的数据交往要尽可能简洁。

数据层就是数据库管理系统, 负责管理对数据库数据的读写。数据库管理系统必须能迅速执行大量数据的更新和检索, 满足功能的需求。

分层风格有效支持了软件设计的逐级抽象, 具有较好的可扩展性, 支持软件复用, 但也要注意降低层次之间的耦合程度。

**评价**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性 | 优先级 | 管道过滤器 | 面向对象 | 分层系统 |
| 易于改变算法 | 2 | 2 | 2 | 2 |
| 易于改变数据表示 | 4 | 2 | 2 | 4 |
| 性能 | 5 | 3 | 3 | 3 |
| 是否易于复用 | 3 | 3 | 4 | 3 |
| 结构是否灵活 | 2 | 3 | 3 | 1 |
| 总计 |  | 42 | 45 | 46 |

**BBS八哥论坛解决方案与评价**

**B/S风格**

B/S体系结构主要利用不断成熟的WWW浏览器技术，特别是浏览器嵌入的多种脚本语言。B/S体系结构有三个主要组成部分：浏览器，Web服务器，数据库服务器。

优点：

维护和升级方式简单，交互性较强。

缺点：

在速度和安全性上需要花费巨大的设计成本，通常需要刷新页面（Ajax等可以一定程度缓解该问题），通信开销大。

**分层结构：**

分层结构采取层次化的组织方法 ,每一层向上一层提供服务，并利用下一层的服务。比较常见的有TCP/ IP 协议，OSI 七层结构等。在 OSI 形成的过程中 ，把通信协议有关的不同任务交给不同的层来完成，使每一层的函数既得到良好的定义又与其他层分开。服务的定义是从底层向顶层形成的。

优点：

层次结构风格支持系统设计过程中的逐级抽象。

基于层次结构风格的系统具有较好的可扩展性。

层次结构风格支持软件复用。

缺点：

并不是每个系统都可以很容易地划分为分层的模式

很难找到一个合适的、正确的层次抽象方法。

**模型-视图-控制器模式**

优点：

支持多个视图。因为视图与模型分离，而且模型与视图之间没有直接依赖性，所以用户界面可以同时显示同一数据的多个视图。例如，Web 应用程序中的多个页面可以使用同一模型对象。

适应更改。用户界面要求的更改往往比业务规则快。用户可能更喜欢新设备采用另一颜色、字体、屏幕布局和支持级别。因为模型不依赖于视图，所以将新类型的视图添加到系统中通常不会影响模型。

缺点：

复杂性。MVC 模式引入了新的间接级别，因此稍微增加了解决方案的复杂性。它还增加了用户界面代码的事件驱动特性，调试用户界面代码会变得更加困难。

频繁更新的成本。将模型与视图分离并不意味着模型的开发人员可以忽略视图的特性。例如，如果模型发生频繁更改，则它可能向视图发出大量更新请求。一些视图的显示可能需要一定时间。因此，视图可能滞后于更新请求。

**评价**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性 | 优先级 | 分层结构 | B/S风格 | 模型-视图-控制器模式 |
| 易于改变算法 | 2 | 3 | 3 | 2 |
| 易于改变数据表示 | 4 | 3 | 4 | 4 |
| 性能 | 5 | 3 | 5 | 4 |
| 是否易于复用 | 3 | 2 | 4 | 4 |
| 结构是否灵活 | 2 | 2 | 2 | 2 |
| 总计 |  | 43 | 63 | 56 |