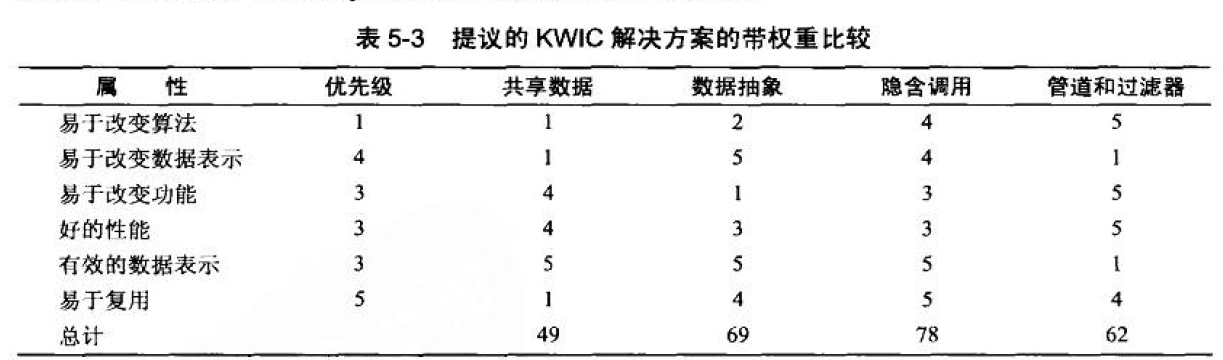
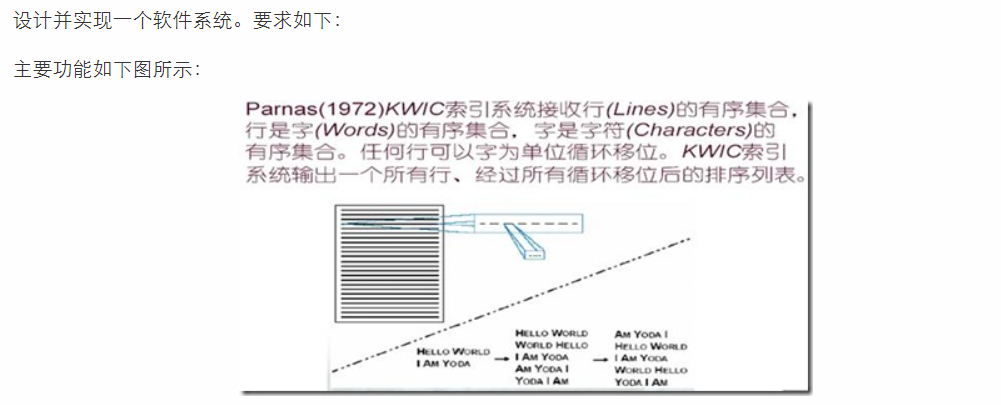
# 评价标准

我们可以参考教材第五章表5-3所示，对KWIC的解决方案和我们小组项目的设计方案进行打分。



# KWIC的解决方案

## 问题简述



该软件系统由命令行指定要进行处理的英文文本文件（.txt）的目录和文件名以及输出结果文件的目录和文件名。

## 解决方案

### **数据流风格：批处理序列；管道/过滤器**

管道-过滤器风格将系统的功能逻辑建立为部件集合。每个部件实例完成一个对数据流的独立功能处理，它接收数据流输入，进行转换和增量后进行数据流输出。

连接件是管道机制，它将前一个过滤器的数据流输出传递给后一个过滤器作为数据流输入。连接件也可能会进行数据流的功能处理，进行转换或增量，但连接件进行功能处理的目的是为了适配前一个过滤器的输出和后一个过滤器的输入，而不是为了直接承载软件系统的需求。

**优点：**

* 易于理解并支持变换的复用。
* 工作流风格与很多业务处理系统体系结构很匹配。
* 通过添加变换的方式进行进化是很显然的。
* 可以实现为顺序的系统，也可以实现为并发的系统。
* 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；
* 系统维护和增强系统性能简单。新的过滤器可以添加到现有系统中来；旧的可以被修改

**缺点:**

* 在通信变换间所传输的数据格式必须协商好。
* 每个变换必须解析它的输入并写成约定的格式输出。
* 增加了系统的负荷，意味着不可能复用实用不兼容数据结构的函数变换。

### 采用调用/返回风格:主程序/子程序、面向对象风格、层次结构

主程序/子程序风格（ Main Program/Subroutine Style ）将系统组织成层次结构,包括一个主程序和一系列子程序。主程序是系统的控制器，负责调度各子程序的执行。各子程序又是一个局部的控制器，调度其子程序的执行。

**优点：**

* 易于理解并支持变换的复用。
* 工作流风格与很多业务处理系统体系结构很匹配。
* 通过添加变换的方式进行进化是很显然的。
* 可以实现为顺序的系统，也可以实现为并发的系统。

**缺点：**

* 在通信变换间所传输的数据格式必须协商好。
* 每个变换必须解析它的输入并写成约定的格式输出.
* 增加了系统的负荷，意味着不可能复用实用不兼容数据结构的函数变换。

### **仓库风格：数据库系统、超文本系统、黑板系统**

数据共享风格也称为仓库风格。这种风格的典型代表有数据库系统、超文本系统、黑板系统。

该风格中，主要有两类部件：

1. 中心数据结构部件，又可称作“数据仓库”表示系统的当前状态。
2. 一组相对独立的部件集，它们可以以不同方式与数据仓库进行交互，这也就是数据共享

体系结构的技术实现基础。

根据所使用的控制策略不同，数据共享体系结构主要有两大分支：

1.如果系统输入业务流的类型是激发进程执行的主要原因，则数据仓库是黑板，其中黑板体

系结构风格主要应用于需要进行复杂解释的信号处理领域。

称为黑板的原因是： 它反映了信息共享， 如同教室里的黑板一样， 其模拟一组人类专家对于同一个问题或者是一个问题的各个方面， 每一位专家都根据自己的专业经验提出自己的看法，写在黑板上，其他人都能看到，随意使用，共同解决好这个问题。可以有多个人读上面的字，也可以有多个人在上面写字。

**优点：**

* 便于多客户共享大量数据，而不必关心数据是何时产生的、由谁提供的以及通过何种途径来提供。
* 便于将构件作为知识源添加到系统中来。

**缺点：**

* 容器是一个单个失败点， 因而容器中的问题会影响整个系统。在组织所有通过容器进行的通信时会比较低效，将容器分布到多个计算机上会很困难。

### 独立构件风格：进程通讯、事件系统

**事件驱动架构**的基本思想是：系统对外部的行为表现可以通过它对事件的处理来实现。一个基于事件驱动构架的应用程序系统， 各个功能设计为封装的、模块化的、可用于共享的事件服务组件，并在这些独立非耦合的组件之间将事件所触发信息进行传递。

**隐式调用**的思想是，不直接调用一个过程， 而是发布或广播一个或多个事件。系统中的其他构件通过注册与一个事件关联起来的过程， 来表示对某一个事件感兴趣。当这个事件发生时，系统本身会调用所有注册了这个事件的过程。这样一个事件的激发会导致其他模块中过程的隐式调用。比如在Field 系统中， 诸如编辑器和变量监视器等工具会注册调试器的中断点事件。

从体系结构的角度说， 隐式调用模式中的构件是模块，其接口不仅提供过程的集合（像抽象数据类型），也提供事件的集合。过程可能以一般的方式被调用,但构件可以将过程注册到与其相关联的系统事件中，这样，当事件发生时,过程会被间接调用。这种模式主要特点是事件发布者不知道哪些构件会受到事件的影响。因此，构件不能对事件的处理顺序，或者事件发生后的处理结果做任何假设。正因为这个原因，许多隐式调用系统也包括显式调用（比如正常的过程调用），以此作为构件交互的补充。

**优点：**

* 事件声明者不需要知道哪些构建会响应事件，因此,不能确定构件处理的先后顺序，甚至不能确定事件会引发哪些过程调用。
* 提高了软件重用能力， 只要在系统事件中注册构件， 就可以将该构件集成到系统中。
* 便于系统升级，只要构件名和事件中所注册的过程名保持不变，原有构件就可以被新构件所替代。

**缺点：**

* 构件放弃了自身对系统计算的控制。当一个构件发布一个事件，它不能保证其他构件会对其做出响应。即使它能够肯定该事件会被其他构件响应，它也不能依赖事件被处理的先后顺序。
* 涉及到数据交换。有时数据通过事件传递，但在某些情况下，事件系统必须依赖一个共享缓冲区，以便于数据的交换。这样，整体的性能和资源的管理可能成为关键性问题。
* 正确性验证，因为发布事件的过程的具体含义与事件激发的上下文有关。这和传统的过程调用验证不同， 当对调用功能行为进行验证时， 传统的过程调用只需考虑过程前和过程后的条件。

## 评分

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 属性 | 优先级 | 数据流风格 | 调用/返回 | 仓库风格 | 独立构件 |
| 易于改变算法 | 2 | 2 | 2 | 2 | 3 |
| 易于改变数据表示 | 4 | 2 | 2 | 4 | 2 |
| 性能 | 5 | 3 | 2 | 3 | 5 |
| 是否易于复用 | 3 | 4 | 4 | 3 | 4 |
| 结构是否灵活 | 2 | 4 | 3 | 1 | 3 |
| 总计 |  | 47 | 40 | 46 | 57 |

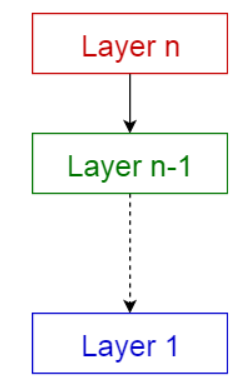
# BBS八哥论坛的设计

## 解决方案

### 分层模式

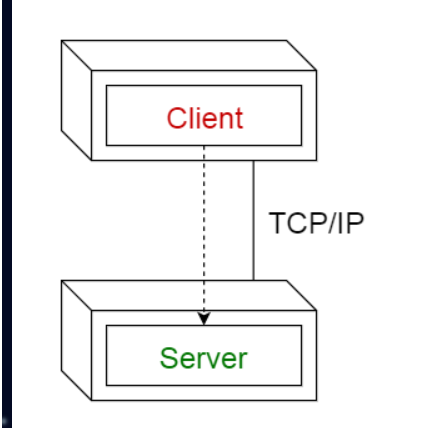
该模式用于构建可分解为多组子任务的程序，每个子任务都在某个抽象层，每个层对上一个更高层提供服务。一般信息系统中最常见的4层体系如下。

* 表示层（也叫 UI 层）
* 应用层（也叫服务层）
* 业务逻辑层（也叫领域层）
* 数据访问层（也叫持久层）



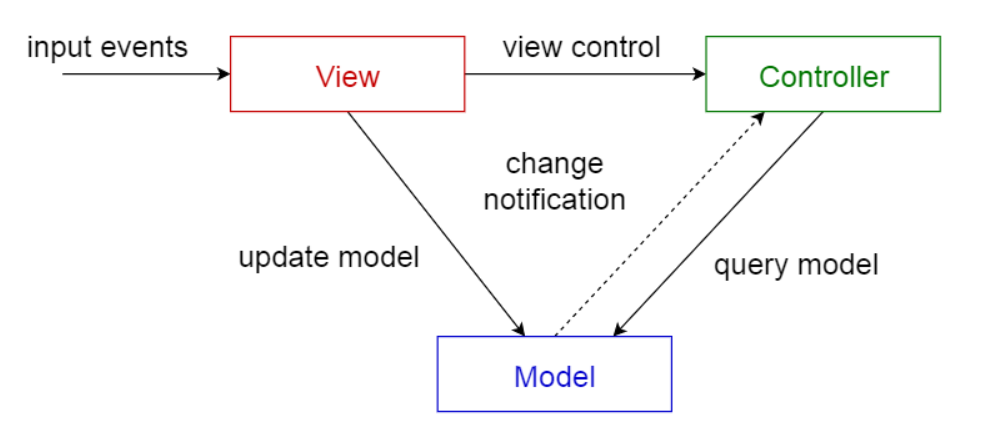
### 客户端-服务器模式

该模式由两部分构成：单个服务器端和多个客户端。服务器组件对多个客户端组件提供服务。客户端向服务器端请求服务，服务端提供对应服务给这些客户端。此外，服务器端继续监听客户端请求。



### 模型-视图-控制器模式

该模式也叫 MVC 模式，划分交互程序为3个部分：模型——包含核心功能和数据，视图——显示信息给用户（多个视图可被定义），控制器——处理用户输入。它通过分割用户信息的内部陈述和呈现、接受方式来实现，解耦组件并允许高效的代码复用。



### 评价

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性 | 优先级 | 分层模式 | B/S模式 | 模型-视图-控制器模式 |
| 易于改变算法 | 2 | 2 | 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 | 3 | 3 |
| 总计 |  | 41 | 65 | 58 |