体系结构模式

8种体系结构模式

- 层Layers
- 管道和过滤器 Pipes and Filters
- 黑板 BlackBoard
- 代理者 Broker
- 模型-视图-控制器 Model-View-Controller
- 表示-抽象-控制 Presentation-Abstraction-Control
- 微核 Microkernel
- 映像 Reflection

2.1 引言

- 体系结构模式
  - 模式系统中的最高等级模式
  - 有助于明确一个应用的基本结构

2.1 引言

- 分类
  - 从混沌到结构
    - 层、管道过滤器、黑板
  - 分布式系统
    - 代理者、微核、管道过滤器
  - 交互式系统
    - 模型-视图-控制器、表示-抽象-控制
  - 适应性系统
    - 映像、微核

2.1 引言

- 体系结构模式的选择
  - 由当前应用的一般属性来确定
  - 在决定一个特定的体系结构模式前多考察几种选择
  - 不同的体系结构模式导致不同的结果
- 绝大多数软件体系结构不能仅依据一个体系结构模式来构建
  - 代理者+MVC
- 体系结构模式的选择，或几个模式的结合，仅是设计一个软件系统的体系结构的第一步

2.2 从混沌到结构
From Mud to Structure

- 高层系统划分与“混沌球”

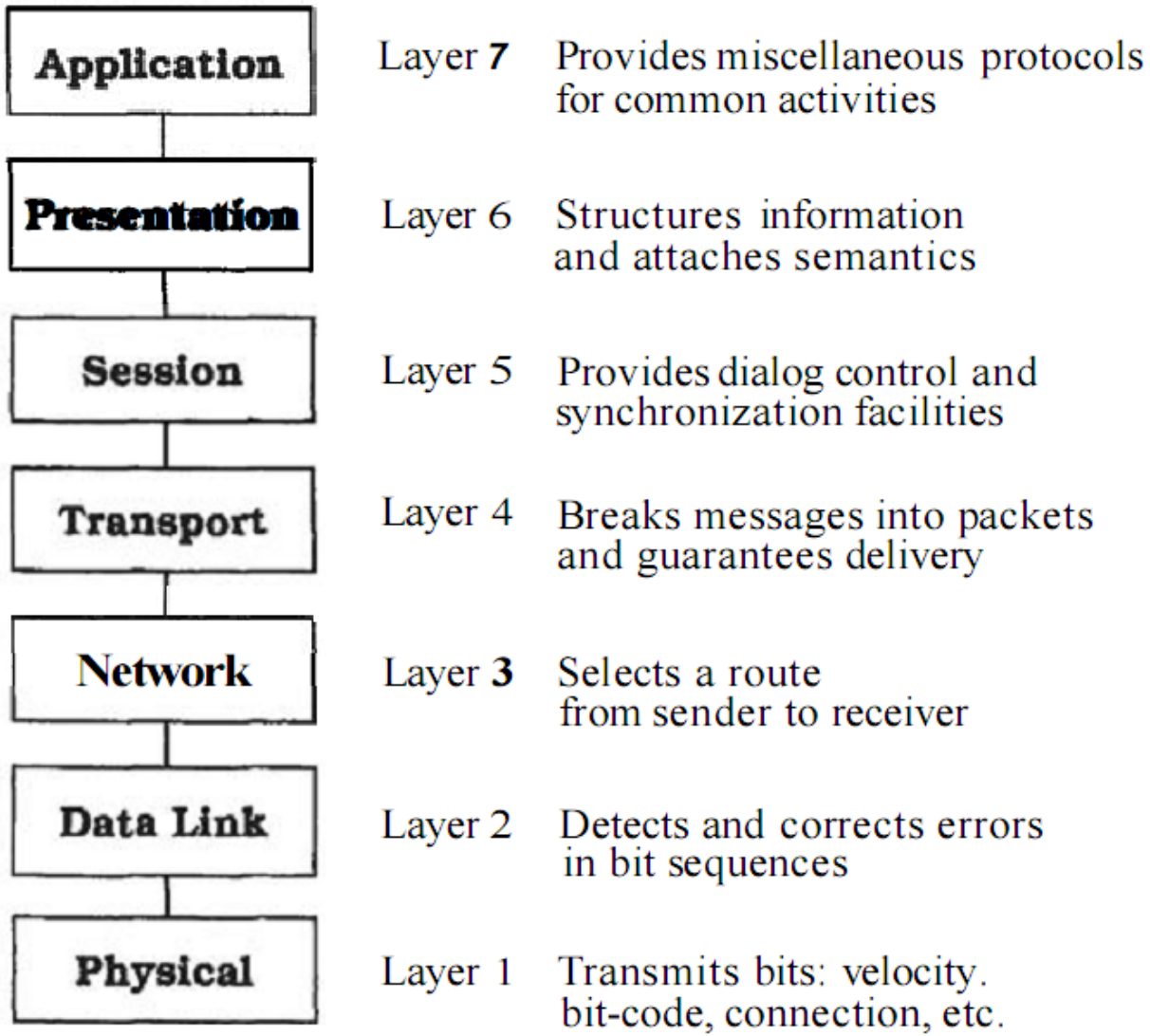
- 三种体系结构模式
  - 层
    - 应用可以分解成子任务组，每个子任务组处于一个特定的抽象层次上
  - 管道过滤器
    - 处理数据流的系统
  - 黑板
    - 无确定性求解策略的问题

2.2.1 层

- 层（Layer）体系结构模式有助于构造这样的应用：它能被分解成子任务组，其中每个子任务组处于一个特定的抽象层次上

The **Layers** architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

1.例子


Application
Layer 7
Provides miscellaneous protocols for common activities
Presentation
Layer 6
Structures information and attaches semantics
Session
Layer 5
Provides dialog control and synchronization facilities
Transport
Layer 4
Breaks messages into packets and guarantees delivery
Network
Layer 3
Selects a route from sender to receiver
Data Link
Layer 2
Detects and corrects errors in bit sequences
Physical
Layer 1
Transmits bits: velocity. bit-code, connection, etc.

2. 语境

- 一个需要分解的大系统

3. 问题

- 系统的显著特征是：
  - 混合了低层与高层问题
  - 高层操作依赖于低层操作
- 一个通信流的典型模式由…组成
  - 高层到低层移动的请求
  - 对请求的应答

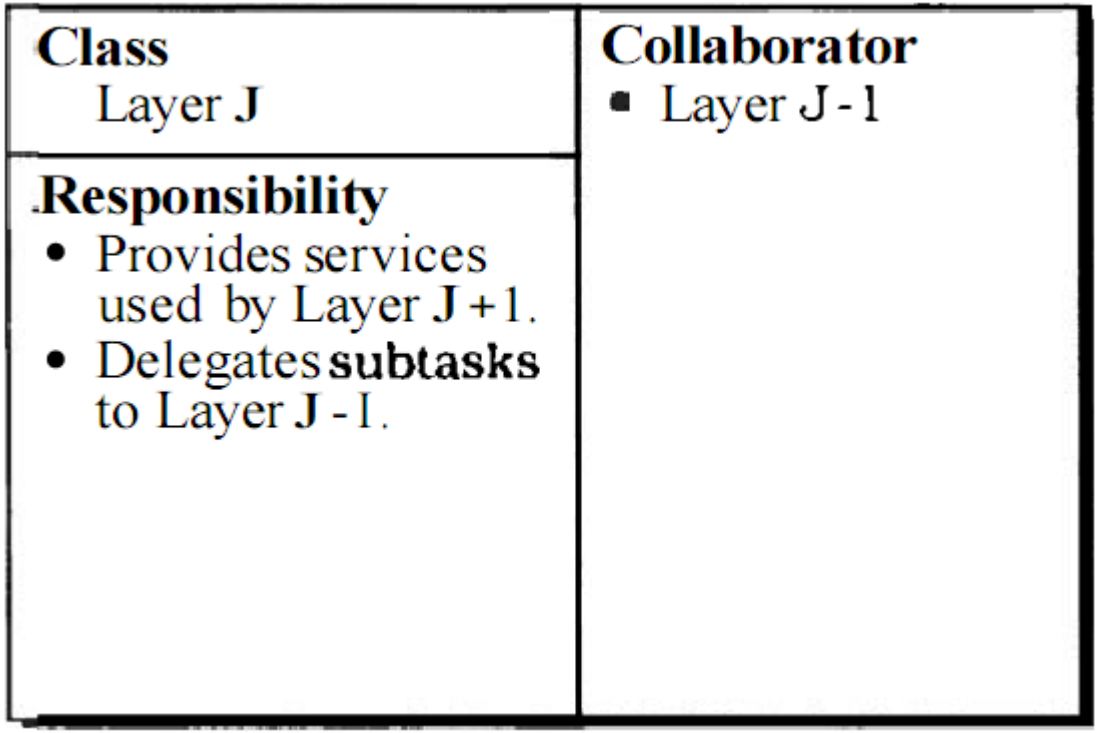
3. 问题

- 需要平衡下列条件：
  - 后期源代码的改动不应该影响到整个系统
  - 接口应该是稳定的，甚至可以用标准件来限定
  - 系统的各个部分应该可以替换
  - 以后可能有必要建立其他系统，这些系统与当前系统具有同样的低层问题
  - 相似职责应分组以提高可理解性和可维护性（内聚）
  - 没有“标准的”组件粒度
  - 复杂组件需要进一步分解
  - 跨组件边界可能影响性能
  - 系统可以由一个程序组来创建，工作界限必须划分清楚

4. 解决方案

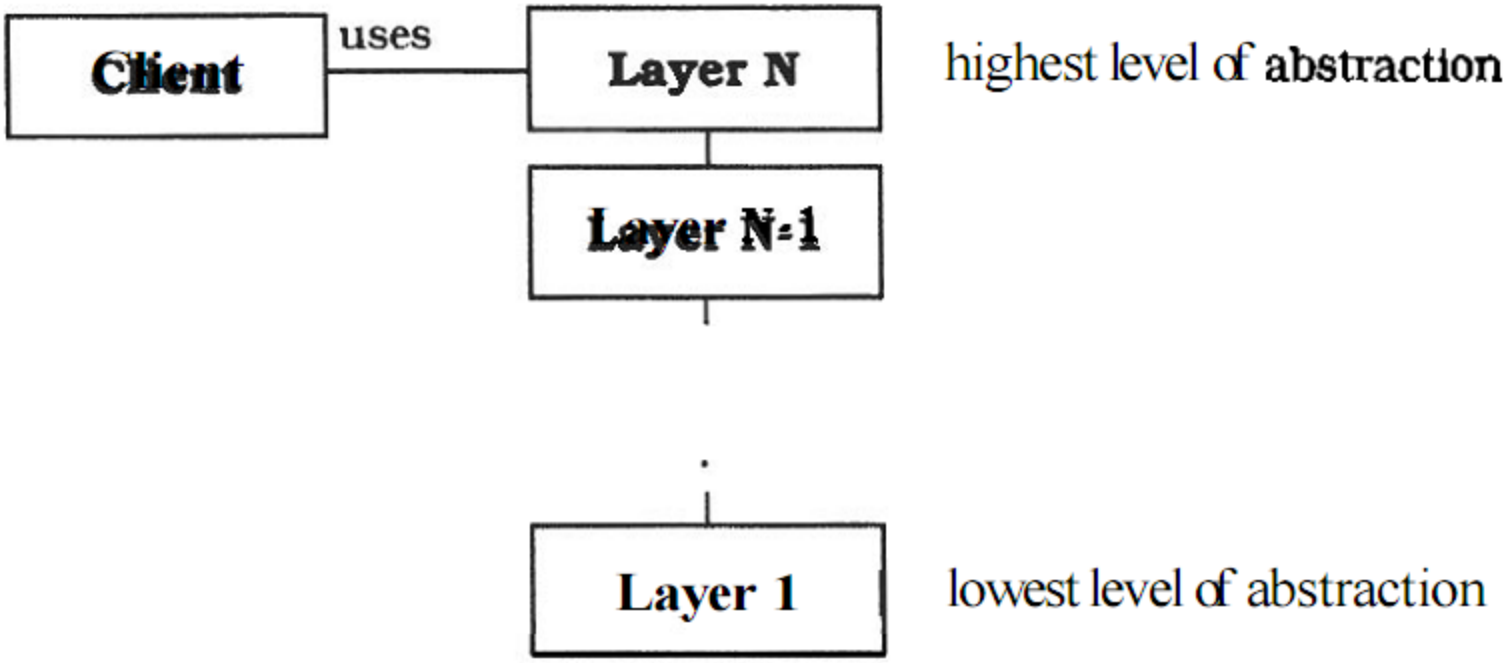
- 将系统分成适当层次，按适当次序放置
  - 从最低抽象层（第1层）到顶层（第N层）
- 层的内部实现不是本模式关注的问题
- 第J层提供的绝大多数服务是由第J-1层提供的服务组成
  - 实现每一层的服务的一种策略是有意义地组合低层的服务

5. 结构


Class
Layer J
Responsibility
• Provides services used by Layer J+1.
• Delegates subtasks to Layer J-1.
Collaborator
• Layer J-1

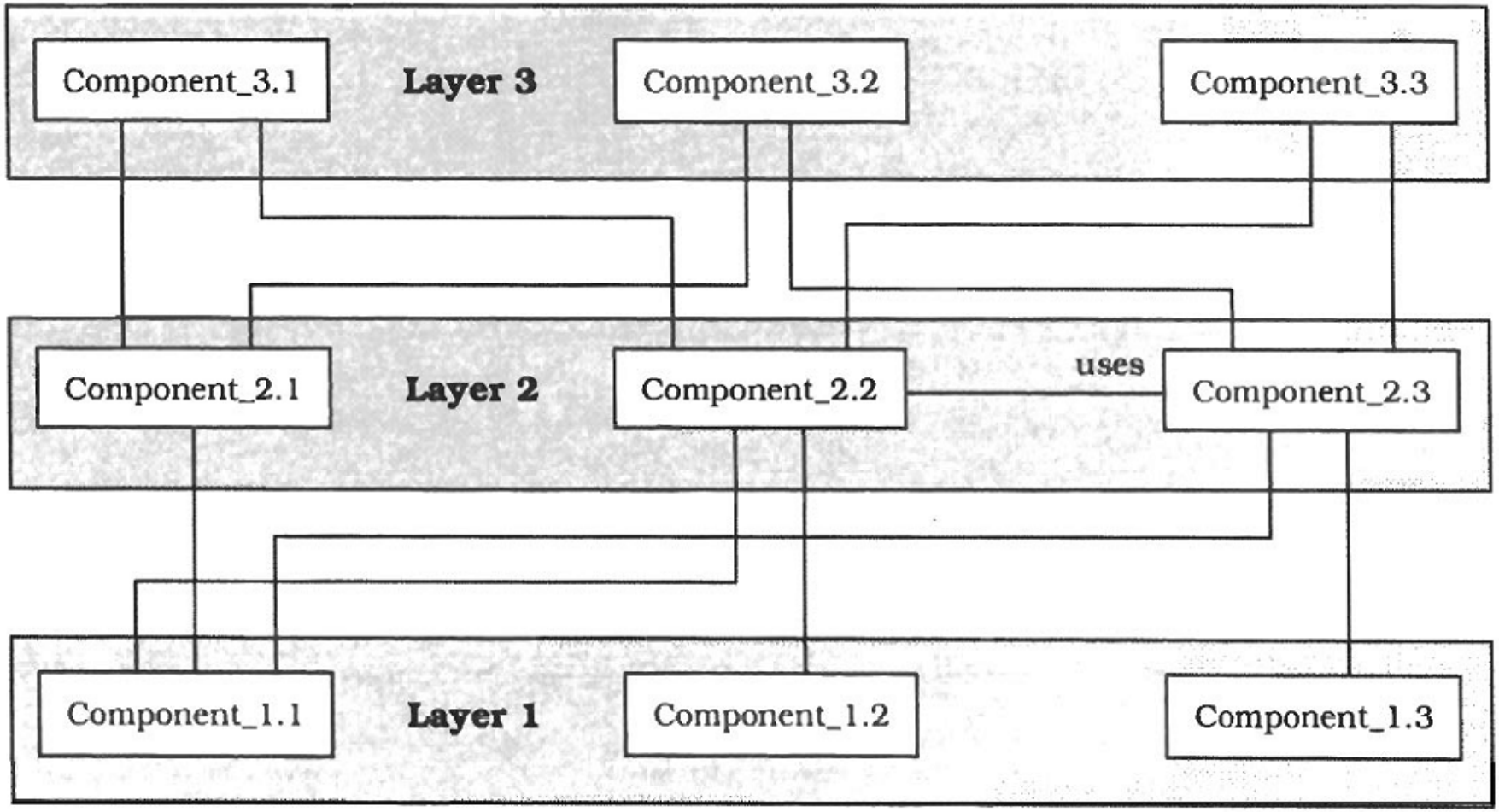
5. 结构

- 主要结构特征：
  - 第J层的服务只被第J+1层使用



5. 结构

- 每一层可以是由不同组件构成的复杂实体



6. 动态特性

- 场景I：自顶向下通信（请求）
  - 客户端向层N发出一个请求
  - 层N自身不能完成该请求，调用层N-1相应的子任务
  - 层N-1向层N-2发送进一步请求
  - …
  - 特点：
    - 层J往往会把层J+1的一个请求转换成几个请求发给层J-1

6. 动态特性

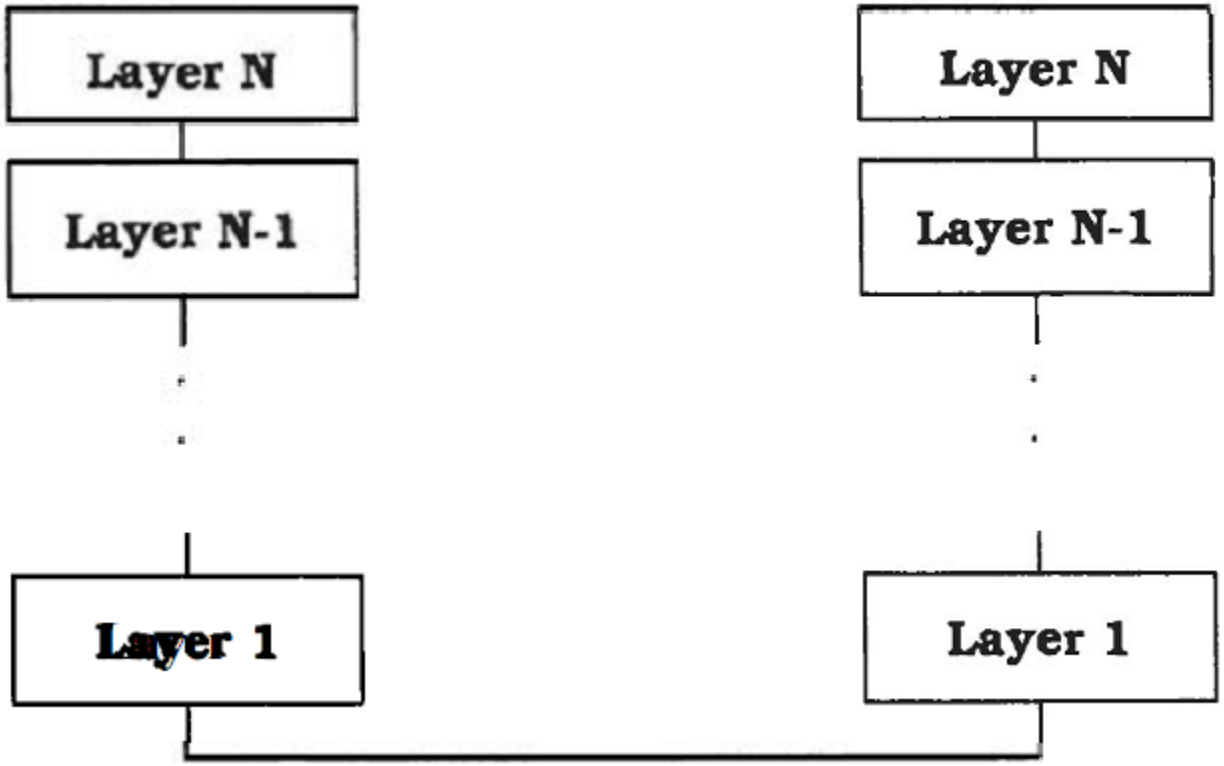
- 场景II： 自底向上通信（通知）
  - 从层1开始（e.g. 设备驱动程序）接收输入转换成内部格式，报告给层2
  - 层2解释，报告给层3
  - …
  - 几个自底向上的通知可能压缩成一个结构上更高的通知，或保持1:1的关系

6. 动态特性

- 场景III： 请求只经历层的一个子集
  - 层N-1可以满足请求的情况
    - E.g. 高速缓存命中的情况
- 场景IV： 通知只经历层的一个子集
  - 通知到层J后，不再向上传递
    - E.g. 通信协议中的重发请求

6. 动态特性

- 场景V：彼此能互相通信的两个层N的堆栈



7. 实现

1）Define the abstraction criterion for grouping tasks into layers

- 棋类游戏
  - Elementary units of the game, such as a bishop

    Basic moves, such as castling
  - Medium-term tactics, such as the Sicilian defense
  - Overall game strategies

7. 实现

1）Define the abstraction criterion for grouping tasks into layers

- 美式足球

In American Football these levels may correspond respectively to linebacker, blitz, a sequence of plays for a two-minute drill, and finally a full game plan.

7. 实现

1）Define the abstraction criterion for grouping tasks into layers

- User-visible elements
- Specific application modules
- Common services level
- Operating system interface level
- Operating system (being a layered system itself, or structured according to the Microkernel pattern (171))
- Hardware

7. 实现

2）根据抽象准则定义抽象层数

- 抽象层次到模式中的层不是一一映射的

3）给每个层命名并指定它们的任务

7. 实现

4）指定服务

- 层间要严格分离
  - 有争议的是
    - 由层J提供功能的错误类型
    - 层间共享模块放松了严格分层的原则
- 较多的服务放在高层往往要比低层好
  - 减少开发员需要学习的“原语”
  - “逆向重用金字塔”
    - 扩展高层以覆盖更宽的应用面
    - 基础层保持“细长”

7. 实现

5）细化分层

- 把最先的4个步骤执行若干次，直到进化出一种自然和稳定的分层

'Like almost all other kinds of design, finding layers does not proceed in an orderly, logical way, but consists of both top-down and bottom-up steps, and certain amount of inspiration…' [Joh95].

- “溜溜球”开发（yo-yo）
  - 交替实行自顶向下和自底向上步骤

7. 实现

6）为每个层指定一个接口

层J对层J+1是：

- 黑盒　　外观（Facade）对象
- 白盒
- 灰盒

7. 实现

7）构建独立层

- 如果一个独立层很复杂，则它应该被分解成几个独立组件
- 子划分可以采用更优粒状模式
- E.g.
  - 桥接模式 Bridge
  - 策略模式 Strategy

7. 实现

8）指定相邻层间的通信

- 常用机制:推模式(push model)、拉模式(pull model)
- 出版者-订阅者模式、管道过滤器模式
- 如要避免拉模式导致的低层与高层之间的依赖关系，使用回调

7. 实现

9）分离邻接层

- 高层关心相邻的低层，而低层不关心用户的身份
- 单路连接：层J的修改可以忽略层J+1的存在和标识
- 请求自顶向下传输（场景I）是单路连接
- 自底向上：使用回调函数
  - 高层注册低层的回调函数
  - 命令模式

7. 实现

```
#include <iostream.h>

class L1Provider {
public:
     virtual void L1Service() = 0;
};
class L2Provider {
public:
     virtual void L2Service() = 0;
     void setLowerLayer(L1Provider *l1) {level1 = l1;}
protected:
     L1Provider *level1;
};
class L3Provider {
public:
     virtual void L3Service() = 0;
     void setLowerLayer(L2Provider *l2) {level2 = l2;}
protected:
     L2Provider *level2;
};
```

7. 实现

```
class DataLink : public L1Provider {
public:
     virtual void L1Service(){
          cout << "L1Service doing its job" << endl;}
};
class Transport : public L2Provider {
public:
     virtual void L2Service() {
          cout << "L2Service starting its job" << endl;
          level1->L1Service();
          cout << "L2Service finishing its job" << endl;}
};
class Session : public L3Provider {
public:
     virtual void L3Service() {
          cout << "L3Service starting its job" << endl;
          level2->L2Service();
          cout << "L3Service finishing its job" << endl;}
};
```

7. 实现

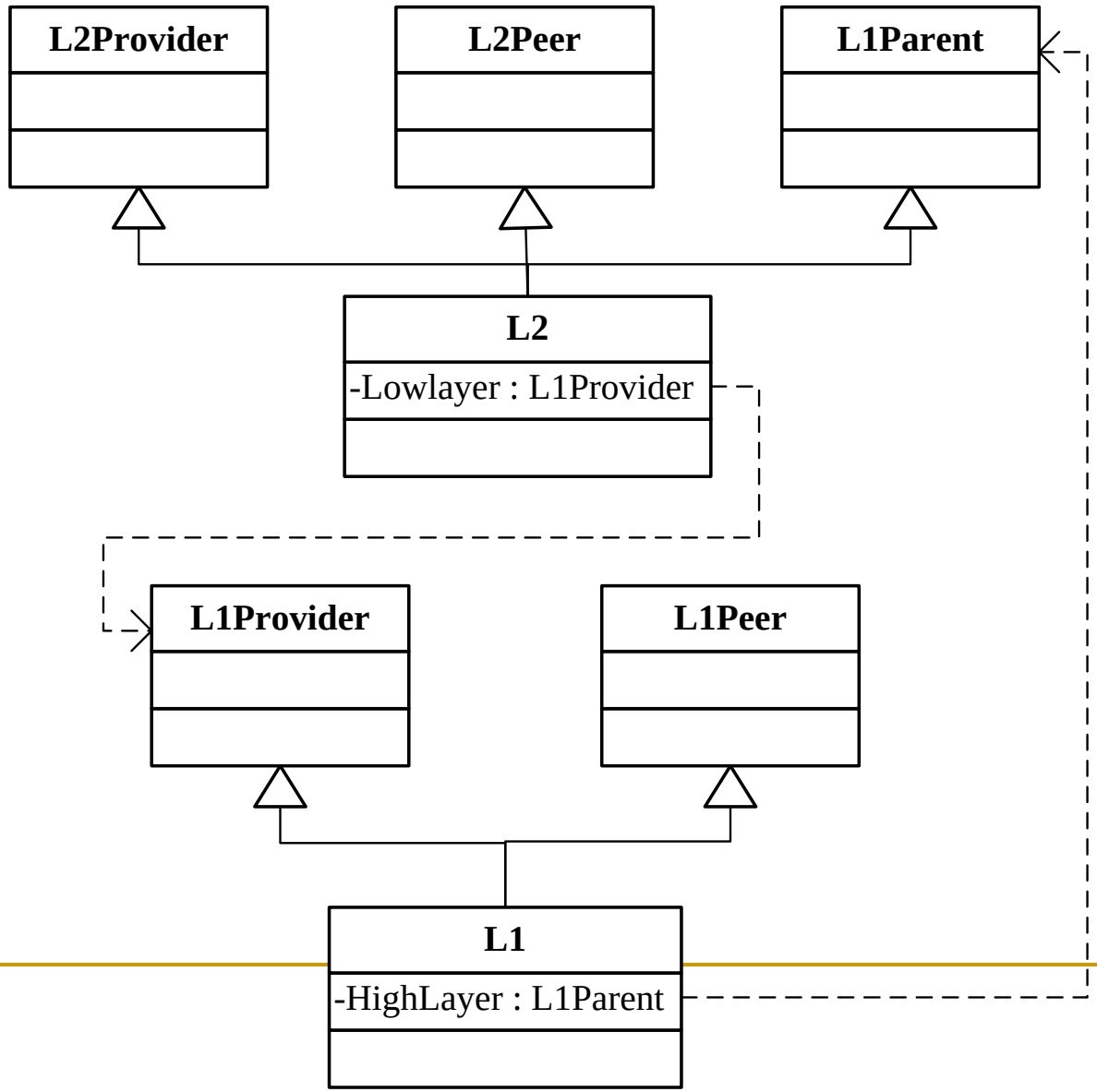
```
main() {
    DataLink dataLink;
    Transport transport;
    Session session;

    transport.setLowerLayer(&dataLink);
    session.setLowerLayer(&transport);

    session.L3Service();
}
```

```
L3Service starting its job
L2Service starting its job
L1Service doing its job
L2Service finishing its job
L3Service finishing its job
```

7. 实现


L2Provider
L2Peer
L1Parent
L2
-Lowlayer : L1Provider
L1Provider
L1Peer
L1
-HighLayer : L1Parent

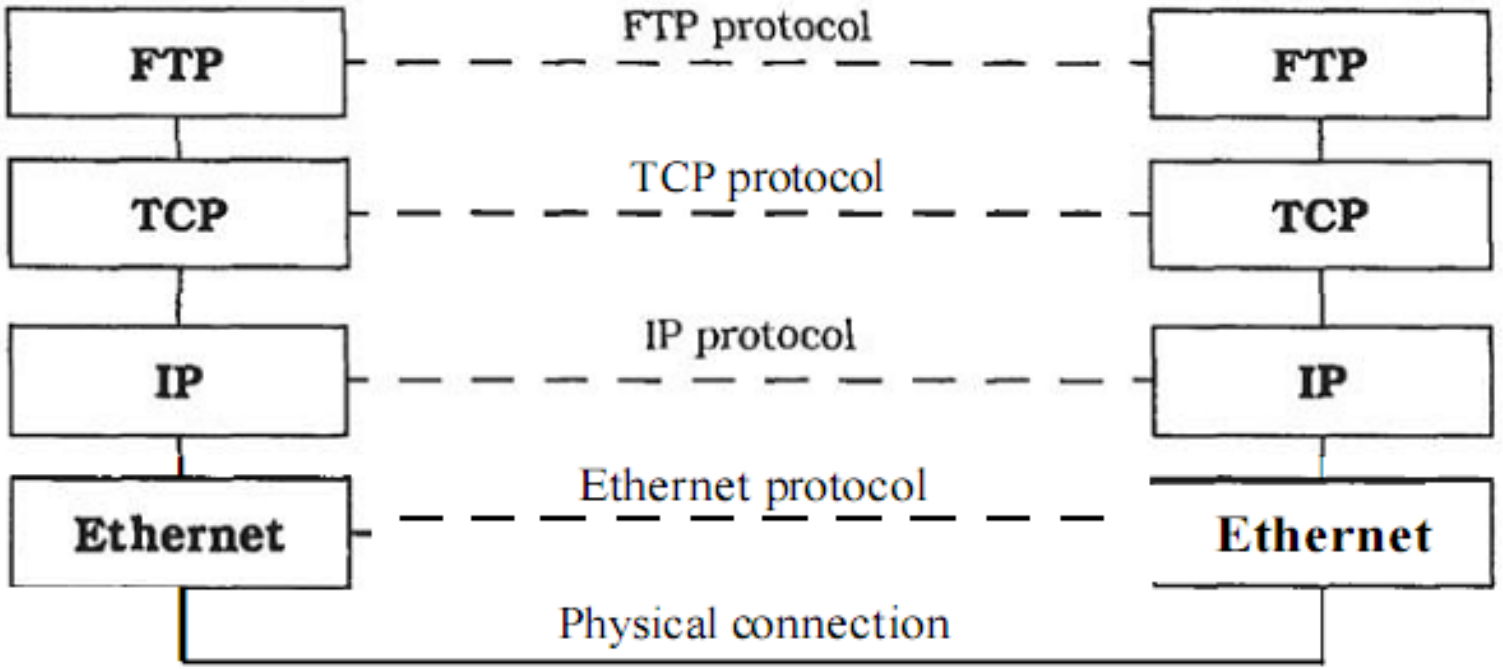
7. 实现

10）设计一种错误处理策略

- 对层式体系结构来讲，错误处理在处理时间和编程工作方面代价较大

- 错误可以在…处理
  - 错误出现的层内处理
  - or
  - 传输到下一个更高层

8. 已解决的例子


FTP
FTP protocol
FTP
TCP
TCP protocol
TCP
IP
IP protocol
IP
Ethernet
Ethernet protocol
Ethernet
Physical connection

9. 变体

- 松散分层系统 Relaxed Layered System
  - 每个层可以使用比它低的所有层的服务
  - 灵活性和性能++、可维护性--
  - 常见于基础结构系统
    - Unix操作系统
    - X Window系统

9. 变体

- 通过继承分层 Layering Through Inheritance
  - 低层作为基类实现
  - 高层（向低层请求服务的）从低层实现中继承

  - 优点：高层可以根据需要修改低层服务
  - 缺点：继承关系把高层和低层捆绑
    - 脆弱基类问题 fragile base class problem

10. 已知应用

- 虚拟机
  - E.g JVM
- API
  - 一个API是一个封装了低层常用功能的层
- 信息系统 IS（商务软件领域）
  - 数据库–应用
  - 数据库–领域层–应用
  - 数据库–领域层–应用逻辑–表示

10. 已知应用

- Windows NT
  - 系统服务
  - 资源管理器层
  - 内核
  - HAL（硬件抽象层）
  - 硬件

  - 内核和I/O管理器直接访问基础硬件

11. 效果

- 优点
  - 层的重用
  - 标准化支持
  - 局部依赖性
  - 可替换性
    - 硬件的替换或添加

11. 效果

- 缺点
  - 更改行为的重叠
    - E.g. 10M以太网 → 155M ATM
  - 降低效率
  - 不必要的工作
  - 难以认可层的正确粒度

参见

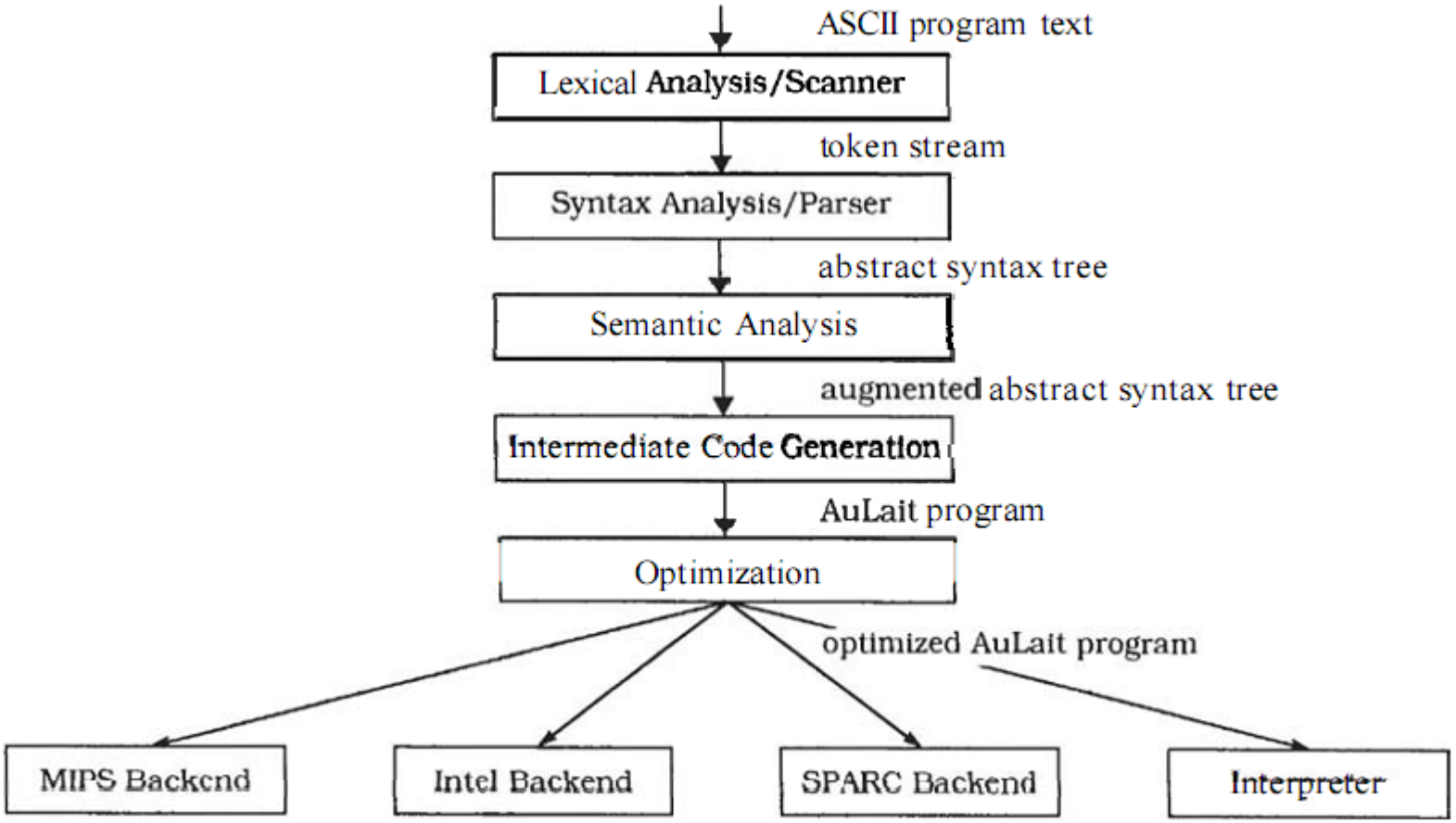
- 组合消息composite message
  - 通过层传输的消息的面向对象的封装
  - 组合模式的一个变种
- 微核
  - 可视为特殊的分层体系结构
- PAC
  - PAC也强调提高抽象层次
  - PAC是PAC节点的一颗树，而不是层次形的竖型结构

2.2.2 管道和过滤器

- 管道和过滤器（Pipes and Filters）体系结构模式为处理数据流的系统提供了一种结构。每个处理步骤封装在一个过滤器组件中。数据通过相邻过滤器之间的管道传输。重组过滤器可以建立相关系统族。

The ***Pipes*** and ***Filters*** architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

1. 例子 Mocha编译器



2. 语境

- 处理数据流

3. 问题

- 以单个组件来实现“处理或转换输入数据流的系统”是不容易的
- 灵活性要求：
  - 替换或重新排列处理步骤

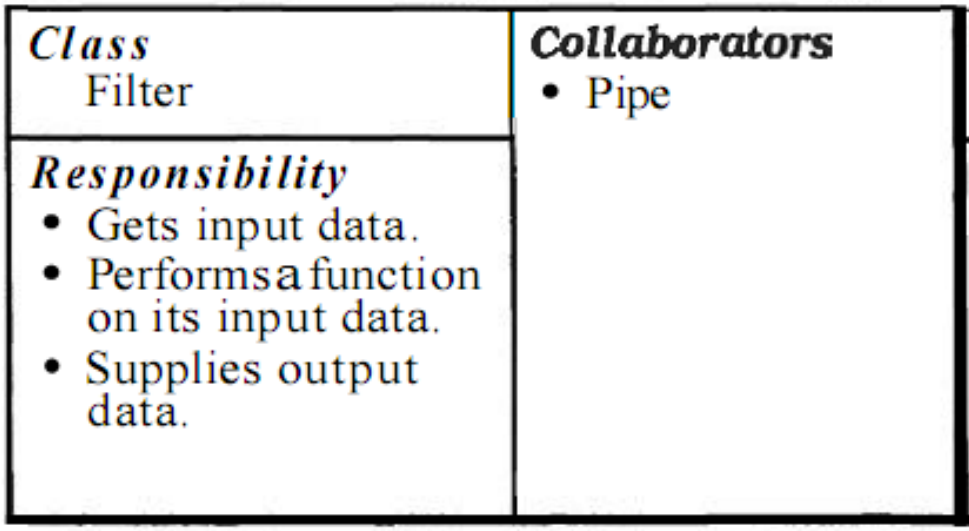
3. 问题

- 系统的设计（尤其是处理步骤的内部连接）要考虑：
  - Future system enhancements should be possible by exchanging processing steps or by recombination of steps, even by users.
  - Small processing steps are easier to reuse in different contexts than large components.
  - Non-adjacent processing steps do not share information.
  - Different sources of input data exist, such as a network connection or a hardware sensor providing temperature readings, for example.
  - It should be possible to present or store final results in various ways.
  - Explicit storage of intermediate results for further processing in files clutters directories and is error-prone, if done by users.
  - You may not want to rule out multi-processing the steps, for example running them in parallel or quasi-parallel.

4. 解决方案

- 将系统任务分为几个顺序的处理步骤
  - 过滤器：实现一个处理步骤
    - 过滤器对数据的增量式处理
  - 输入：数据源 data source
  - 输出：数据汇点 data sink
  - 管道：相连处理步骤间的数据流通
  - 处理流水线 Processing pipeline

5. 结构



Class Filter	**Collaborators**
Responsibility • Gets input data. • Performs a function on its input data. • Supplies output data.	• Pipe

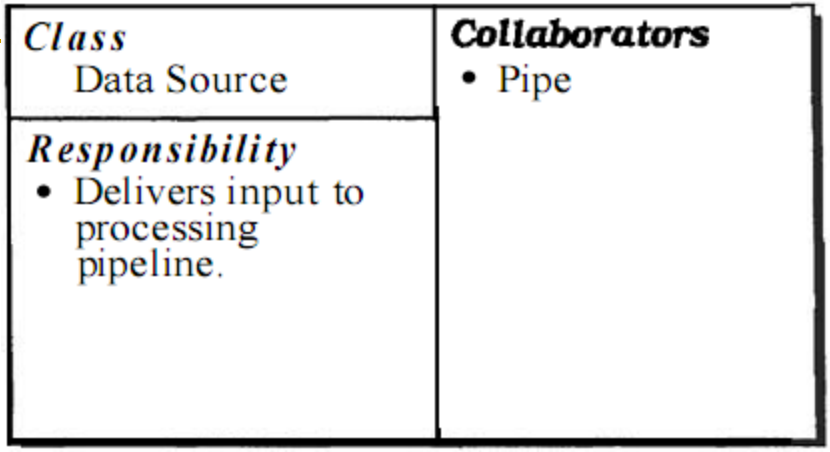
- Filter
  - 流水线的处理单元
  - Enrich、Refine、Transform data
- 由…触发动作
  - 后续单元pull　　被动过滤器
  - 前导单元push　　被动过滤器
  - 自己以循环方式工作，主动取得输入数据(pull)，产生输出数据(push)　　主动过滤器

5. 结构

Class Pipe	*Collaborators*
Responsibility • Transfers data. • Buffers data. • Synchronizes active neighbors.	• Data Source • Data Sink • Filter

- Pipe
  - 过滤器之间
  - Data source到过滤器
  - 过滤器到Data sink
- 工作方式
  - 连接的两个组件都是主动的
    - Pipe实现同步（FIFO缓冲区）
  - 由某个过滤器控制
    - Pipe调用另一个过滤器（被动过滤器）

5. 结构



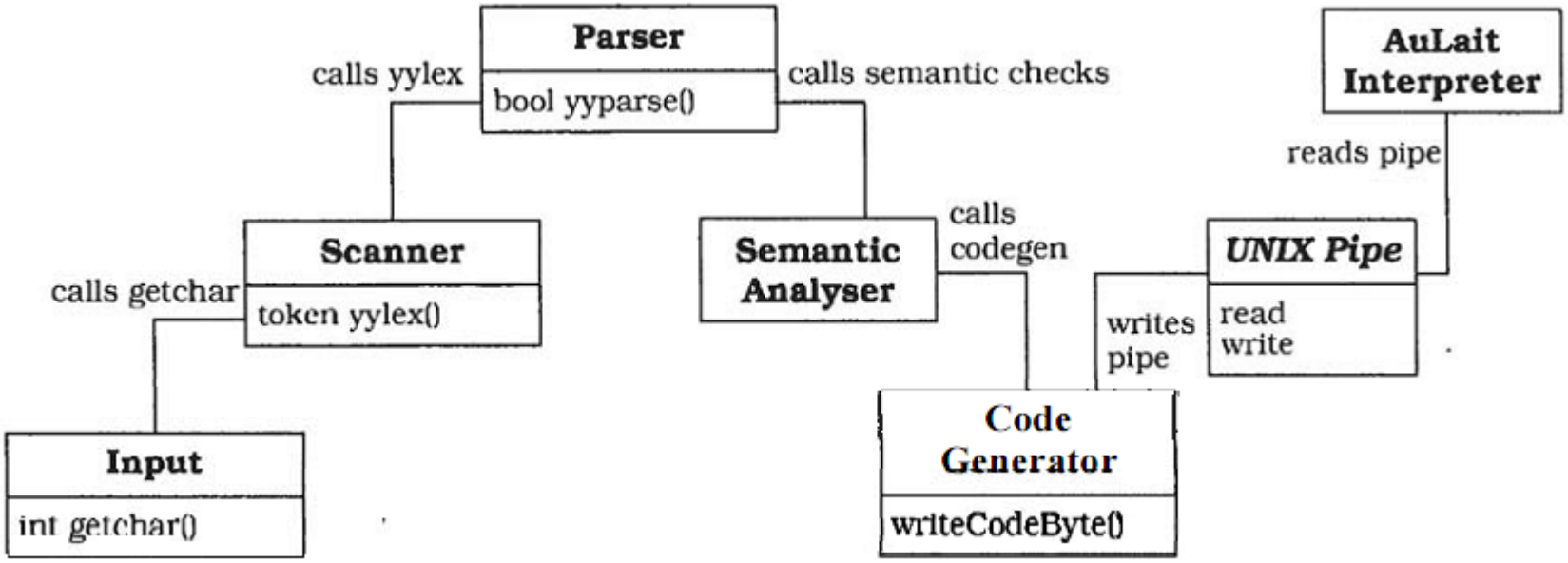
- Data Source
  - 提供一系列相同结构或类型的数值
  - E.g. 文本行组成的文件、发送数字序列的传感器
- 工作方式
  - 主动把数据推入第一个组件
  - Or
  - 第一个过滤器Pull数据时，被动地提供

5. 结构

Class	Collaborators
Data Sink	• Pipe
Responsibility	
• Consumes output.	

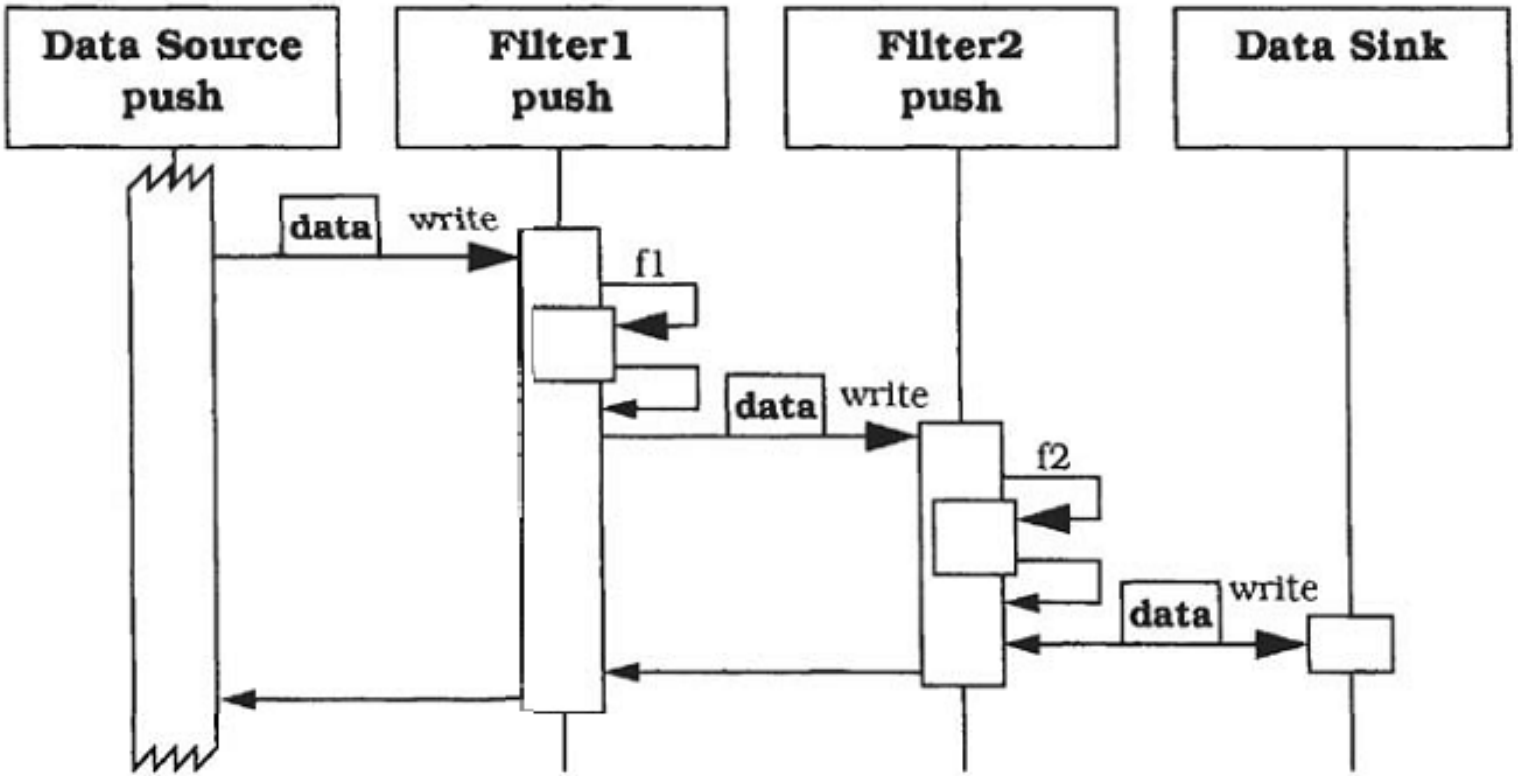
- Data sink
  - 收集来自流水线终端的结果
  - 主动数据汇点
    - Pull results out of the preceding processing stage
  - 被动数据汇点
    - Allows the preceding filter to push or write the results into it

Mocha编译器


Parser
bool yyparse()
calls yylex
calls semantic checks
AuLait Interpreter
reads pipe
Scanner
token yylex()
calls getchar
Semantic Analyser
calls codegen
UNIX Pipe
read
write
writes pipe
Input
int getchar()
Code Generator
writeCodeByte()

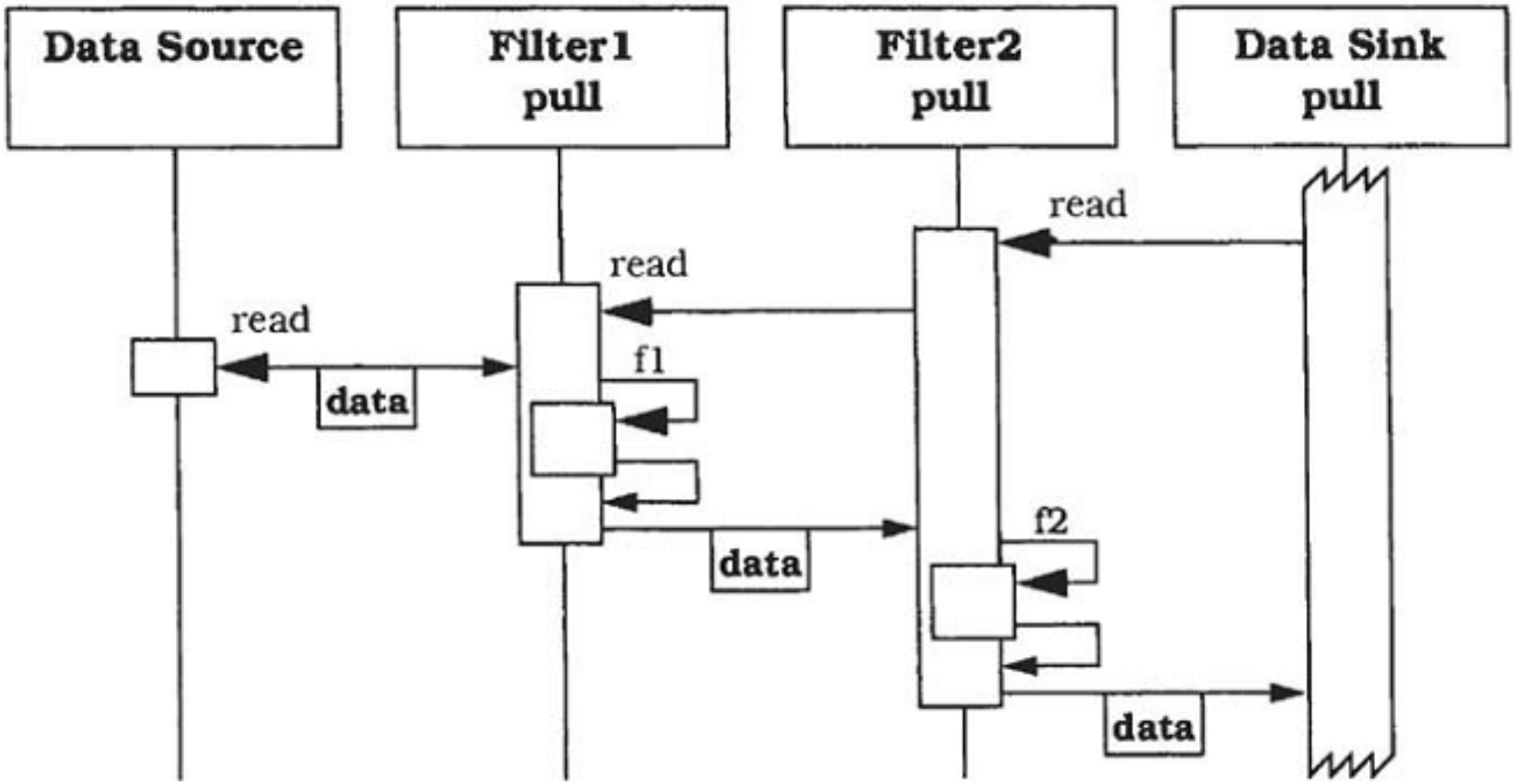
6. 动态特性

- 场景I　push



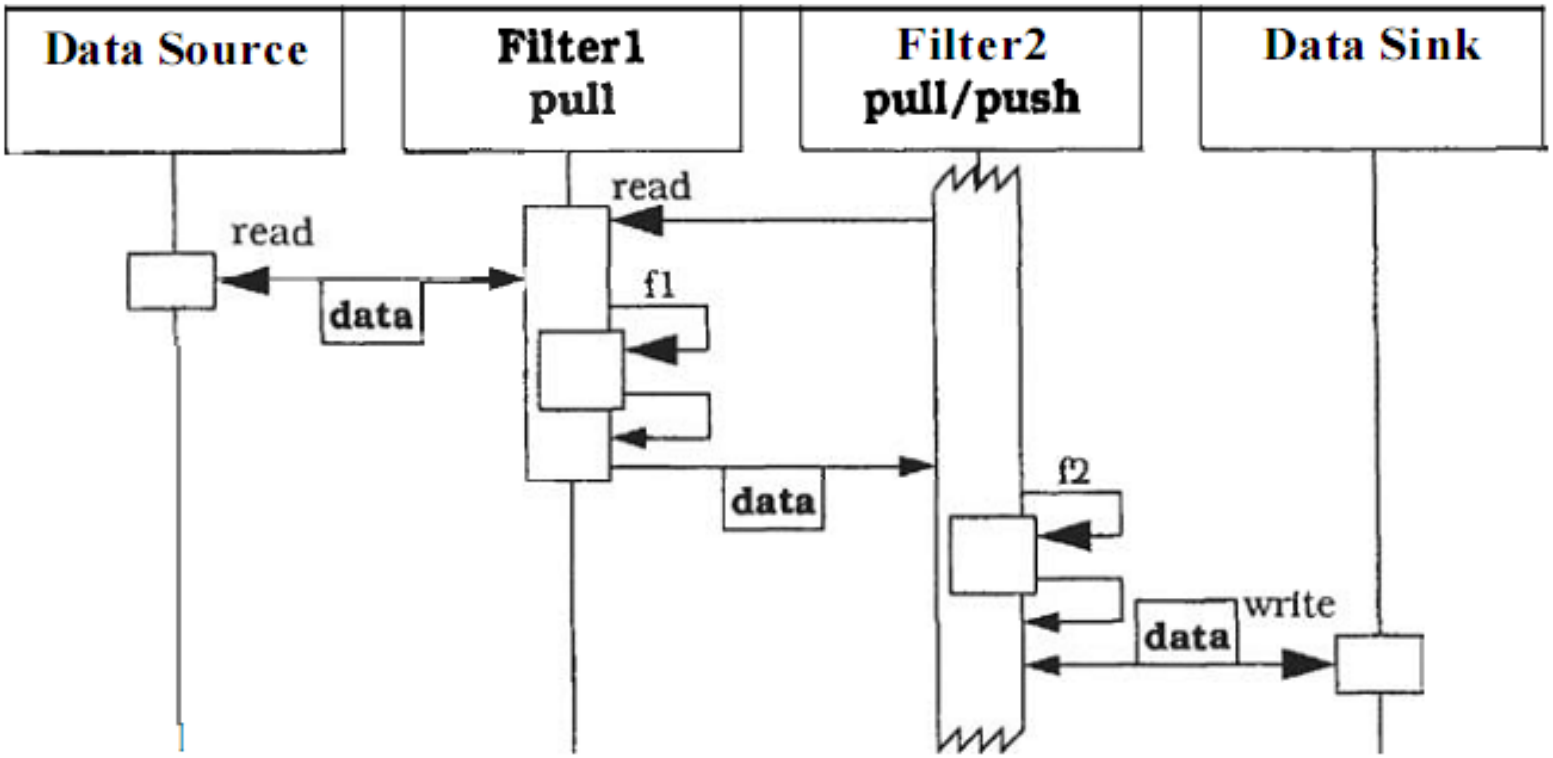
6. 动态特性

- 场景II pull



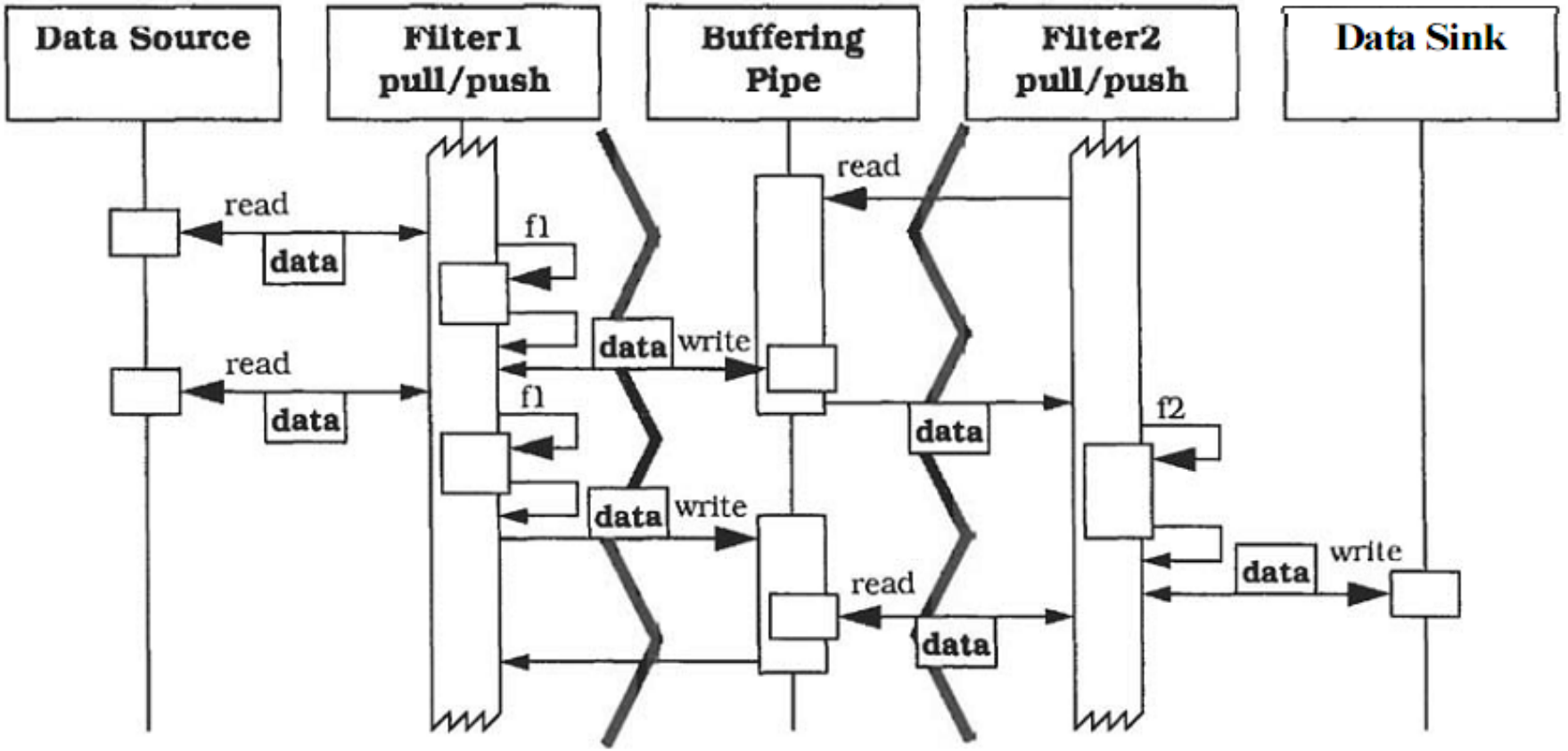
6. 动态特性

- 场景III Mixed



6. 动态特性

- 场景IV 主动过滤器

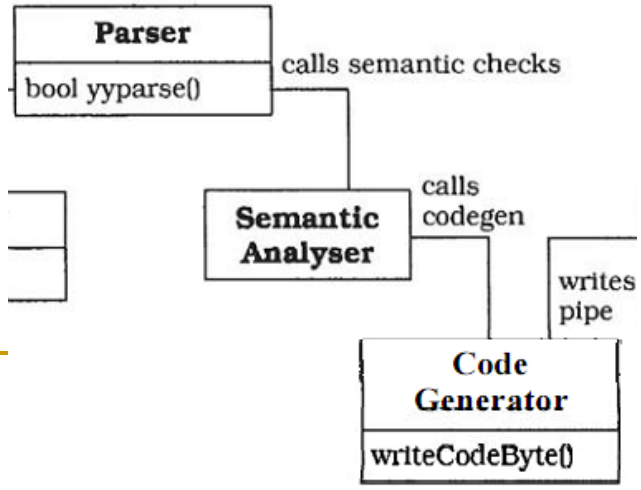


7. 实现

1）将系统任务分成一系列处理阶段

- 每个阶段必须只依赖其前一阶段的输出
- 通过数据流将所有阶段在概念上相连起来
- Mocha编译器
  - 主要分离处：Aulait的前端和后端之间
  - 前端：扫描器、语法分析器、语义分析器(sa)、代码生成器(cg)

```
addexpr   :    term
          |    addexpr '+' term
               { sa.checkCompat($1,$3); cg.genAdd($1,$3); }
          |    addexpr '-' term
               { sa.checkCompat($1,$3); cg.genSub($1,$3); }
```



7. 实现

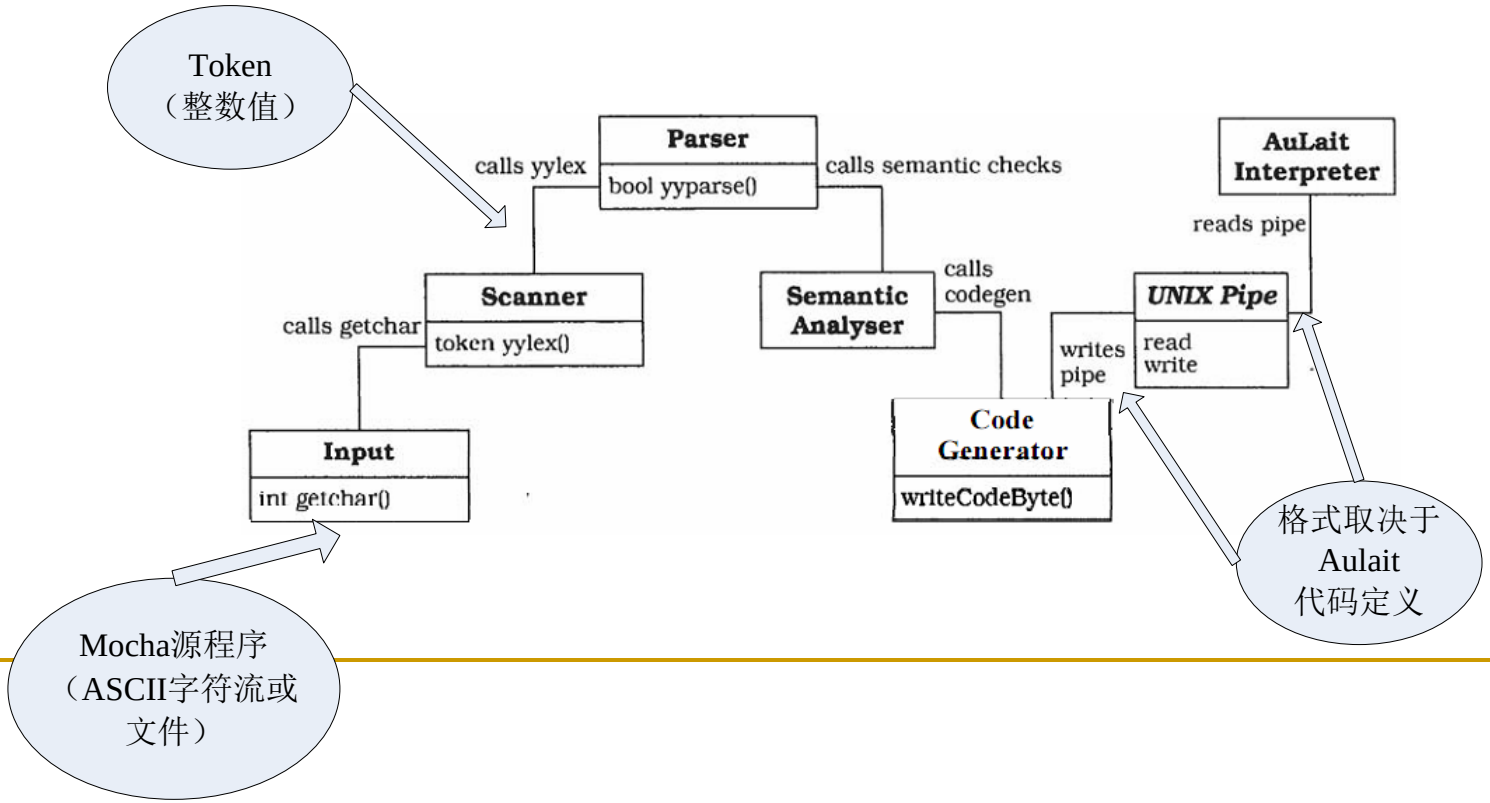
2）定义沿每个管道传输的数据格式

- 统一的格式：最大的灵活性，存在效率问题
- 使用转换过滤器组件
  - 对语义等价的数据进行转换
  - 既拥有灵活性，又能选择不同的数据表示
- 定义如何标识输入结束(E.g. Ctrl-z)

7. 实现

2）定义沿每个管道传输的数据格式

- Mocha编译器



7. 实现

3）决定如何实现每个管道连接

- 该决定将确定
  - 过滤器是主动or被动
  - 被动过滤器是由压入数据或拉出数据启动
- 最简单情况：相邻过滤器间压入或拉出一组数据值的直接调用
  - 缺点：重组或重用过滤器的组件时不方便
- 更灵活的分离管道机制：同步化相邻主动过滤器
- Mocha编译器
  - 前端和后端之间使用UNIX管道机制

7. 实现

4）设计和实现过滤器

- 基于其必须完成的任务以及相邻管道
- 实现被动过滤器
  - Pull动作可作为函数
  - Push动作可作为过程
- 实现主动过滤器
  - As processes or as threads in the pipeline program
- 对性能的影响
  - 过程之间的上下文转换
  - 在不同的地址空间中的数据拷贝
  - 管道的缓冲区大小
- 提高过滤器的重用性：向过滤器传递参数
  - 命令行参数（e.g UNIX过滤器程序）
  - 过滤器在执行时读取全局环境参数
  - 注意：考虑过滤器的灵活性和易用性之间的平衡

7. 实现

4）设计和实现过滤器

- Mocha编译器
  - 前端：
    - 从标准读入读程序源代码
    - 在标准输出上创建一个AuLait程序
    - 由直接调用来通信
  - 后端
    - 产生目标代码作为输出的过滤器阶段

7. 实现

5）设计出错处理

- 流水线组件不能共享任何全局状态，因此错误处理很难做到且往往被忽略
- 错误探测
  - 错误消息的输出管道（ e.g. stderr）
    - 过滤器并发执行时，一个单独的错误管道…
  - 过滤器在输入数据中探测到错误时
    - 可行的策略：忽略输入直到某种分隔符出现
  - 重启流水线
    - 流水线的再同步

7. 实现

5）设计出错处理

- Mocha编译器
  - 错误发送到标准错误管道stderr
  - 语法分析器：探测到语法错误时跳过标记，直到“；”

```
import    :    FROM identifier objidentlist ';'
           |   FROM error ';'
                    { mochaerror(errs[E_IMPRT]); yyerrok; }
```

7. 实现

6）建立处理流水线

- 如果系统处理单个任务
  - 可用一个main程序建立流水线并开始处理
- 通过提供外壳或其他终端用户能力为过滤器组件集创建各种流水线来增加灵活性

You can increase flexibility by providing a shell or other end-user facility to set up various pipelines from your set of filter components.

7. 实现

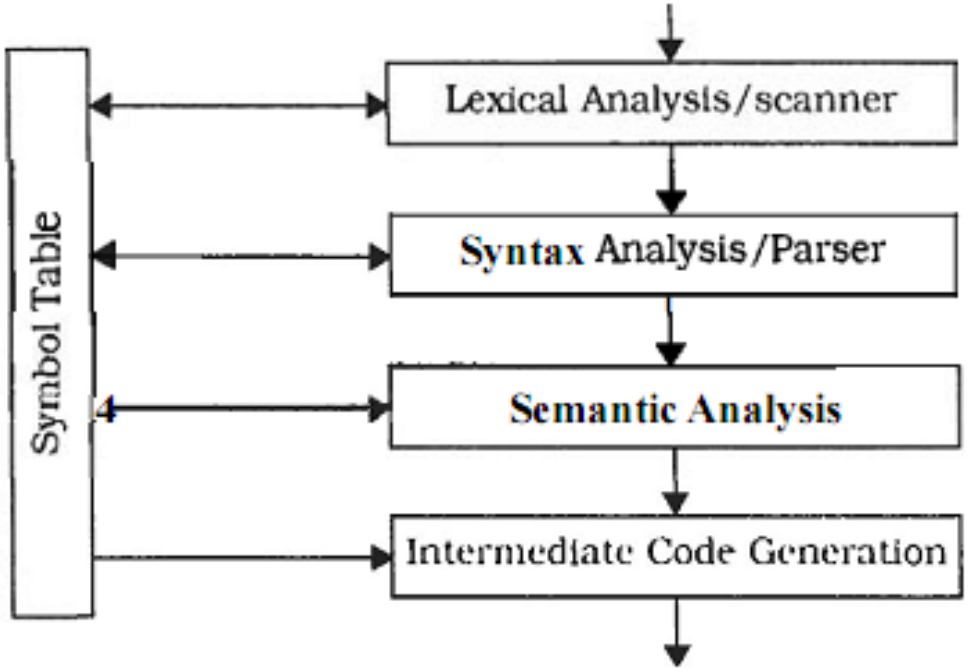
6）建立处理流水线

- Mocha编译器

```
# compile and optimize a Mocha program for a Sun
$ Mocha <file.Mocha | optauLait | auLait2SPARC >a.out

# interpret a Mocha program
$ Mocha <file.Mocha | cup
```

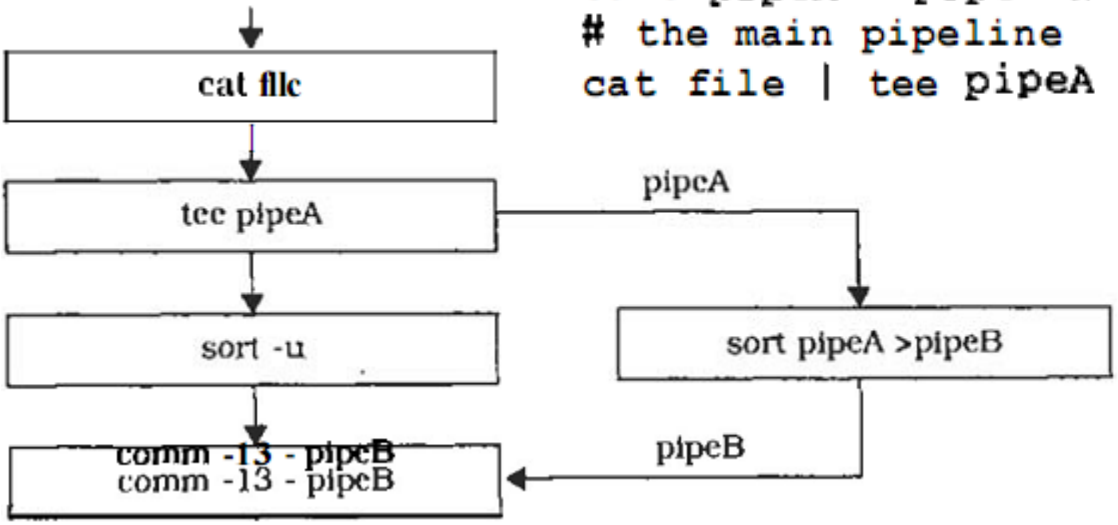
8. 已解决的例子


Lexical Analysis/scanner
Syntax Analysis/Parser
Semantic Analysis
Intermediate Code Generation
Symbol Table

9. 变体

- Tee and join pipeline systems

```
# first create two auxiliary named pipes to be used
rmknod pipeA p
mknod pipeB p
# now do the processing using available UNIX filters
# start side fork of processing in background:
sort pipeA > pipeB &
# the main pipeline
cat file | tee pipeA | sort -u | comm -13 - pipeB
```



10. 已知应用

- UNIX : Command shells
- CMS流水线
  - IBM大型机操作系统的扩展（支持管道和过滤器）
- LASSPTool
  - 支持数字化分析和图形的工具集

11. 效果 ---- 优点

- No intermediate files necessary, but possible
- Flexibility by filter exchange(替换)
- Flexibility by filter recombination(重组)
- Reuse of filter components
- Rapid prototyping of pipelines
- Efficiency by parallel processing

11. 效果 ---- 缺点

- 共享状态信息昂贵或不灵活
- 并行处理获得的效率往往是一种假象
  - 过滤器之间传输数据的代价比单个过滤器进行计算的代价高
  - 一些过滤器非增量式处理
  - 在单处理器上进行线程或进程切换
  - 同步化导致经常终止或启动过滤器
- 数据转换额外开销
- 错误处理

参见

- 层模式更适合需要可靠操作的系统，因为它比管道和过滤器模式更容易实现错误处理
- 层模式缺少对组件的易于重组和重用的支持，而这是管道和过滤器模式的主要特征

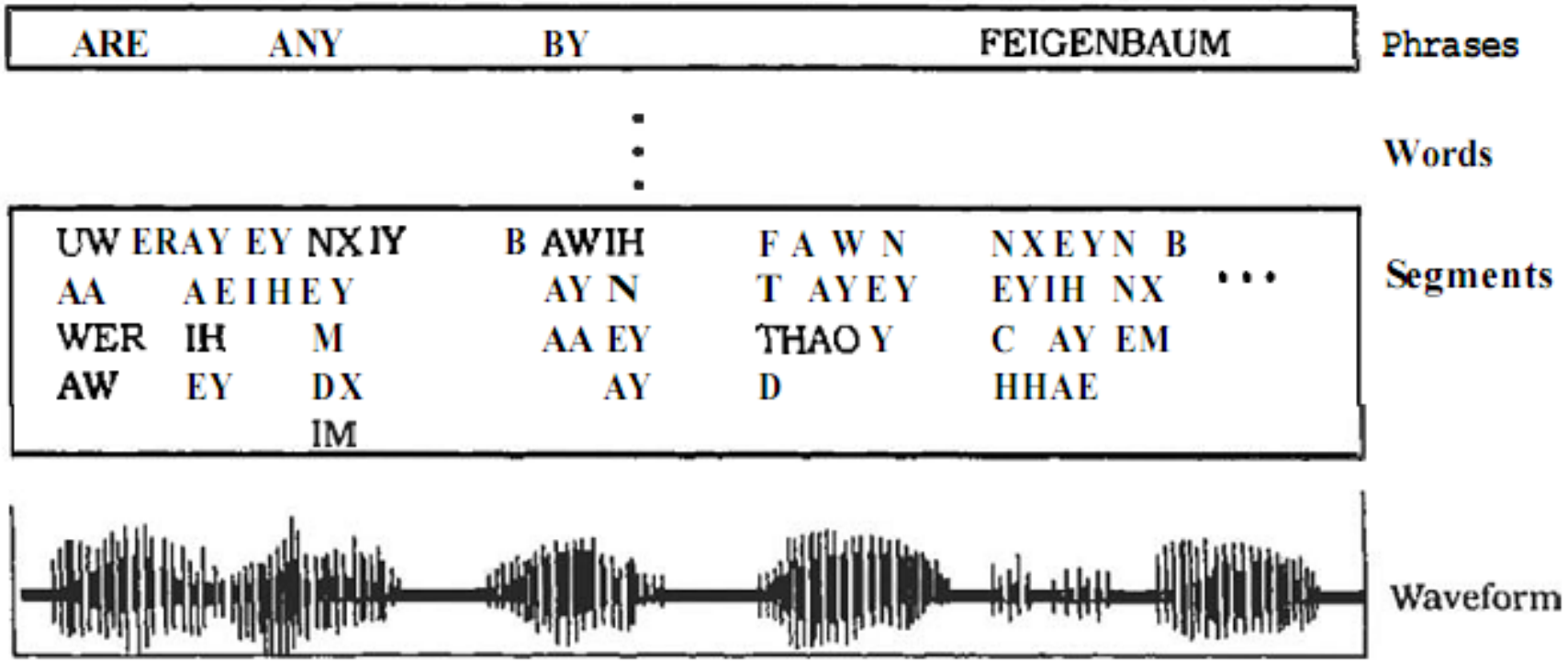
2.2.3 黑板

- 黑板(Blackboard)体系结构模式对于无确定性求解策略的问题比价有用。在黑板模式中有几个专用子系统收集其知识以建立一个可能的部分解或近似解。

The Blackboard architectural pattern is useful for problems for which no **deterministic** solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

1. 例子

- 输入：记录为一段波形的语音
- 输出：对应英文段落的机器表示



2. 语境

- 一个不成熟的领域，其中没有相近的已知方法或可行的方法

- ***An immature domain in which no closed approach to a solution is known or feasible.***

3. 问题

- 针对那些在把原始数据转换成高层数据结构（如图、表或英语短语等）方面没有可行的确定的求解方法的问题
  - E.g. 视觉、图像识别、语音识别和监视等领域

3. 问题

- 约束条件
  - *在合理的时间内，解空间的完全搜索是不可行的。*例如，在1000个单词的词汇量中考虑一个最多10个词的短语，单词排列数达到$\mathbf{1000^{10}}$
  - *由于领域还不成熟，所以需要有对同一个子任务进行不同算法的试验。*由于这个原因，单个模块应该是可替换的
  - *可以有求解部分问题不同的算法。*例如，波形中的语音切片的探测和基于词或词组的短语生成是无关的。
  - *输入，以及中间环节和最终结果，有不同的表示。*并且算法依据不同的范型来实现。

3. 问题

- 约束条件
  - *一个算法的输入往往是其他算法的输出*
  - *不确定数据和近似解也包含在内。*For example, speech often includes pauses and extraneous sounds. These significantly distort the signal. The process of interpretation of the signal is also error-prone. Competing alternatives for a recognition target may occur at any stage of the process. For example, it is hard to distinguish between 'till' and 'tell'. The words 'two' and 'too' even have the same pronunciation, as do many others in English.
  - *使用不相交算法引起潜在的并行性。*如可能，应避免严格顺序算法。

4. 解决方案

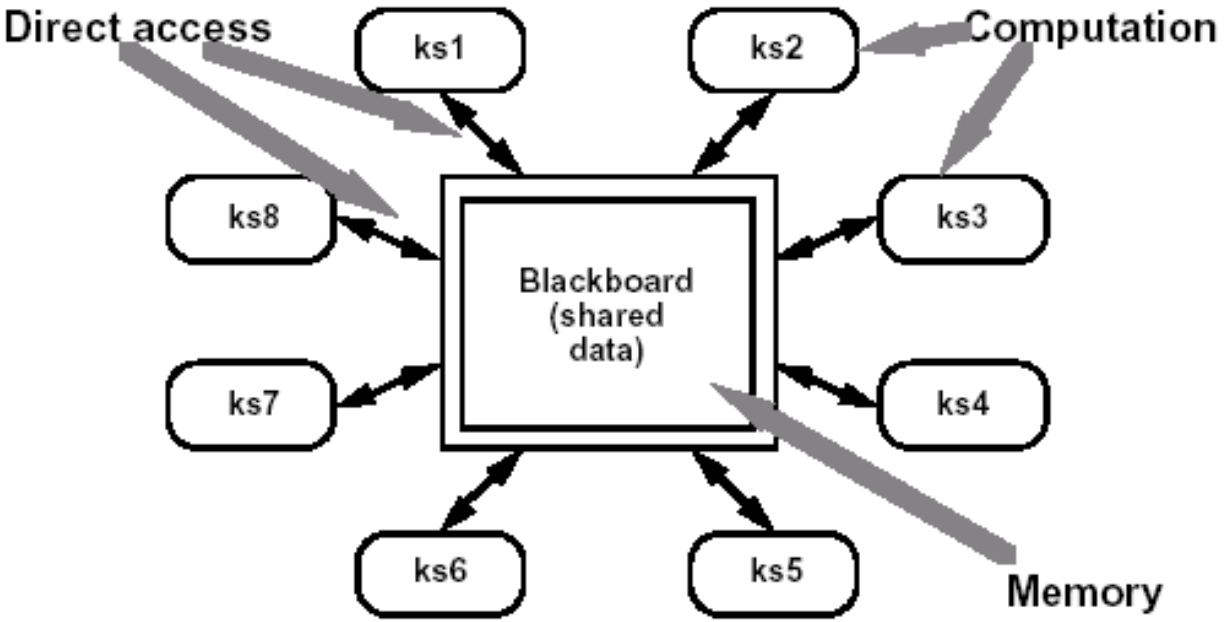
- 黑板体系结构背后的思想是*合作工作于一个公共数据结构上的独立程序集（a collection of independent programs that work cooperatively on a common data structure）*。每个程序专门用来解决整个任务中的一个特定部分，所有的程序一起工作以解决问题。
  - 这些程序彼此独立。彼此不互相调用。它们的活动也没有预先确定顺序
  - 系统的执行方向主要由当前状态确定。一个中心控制组件评估当前状态，协调专用程序
    - 允许采用不同算法进行试验
    - 允许启发算法

4. 解决方案

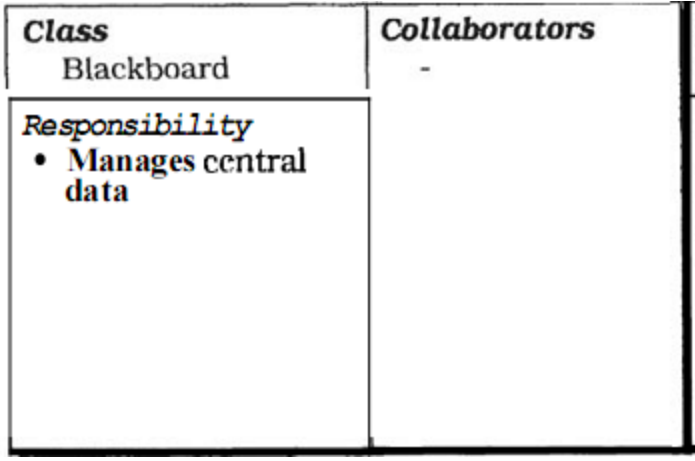
- 在问题—求解过程中系统通过部分求解方案的组合、更改和否定来工作
- 所有可能解的集合称为“解空间”，并被组织成几个抽象层次。
  - 解空间的最低层次由输入的一个内部表示构成
  - 整个系统任务的潜在解决方案处于最高层

5. 结构

- ***Divide your system into a component called blackboard, a collection of knowledge sources, and a control component.***



5. 结构



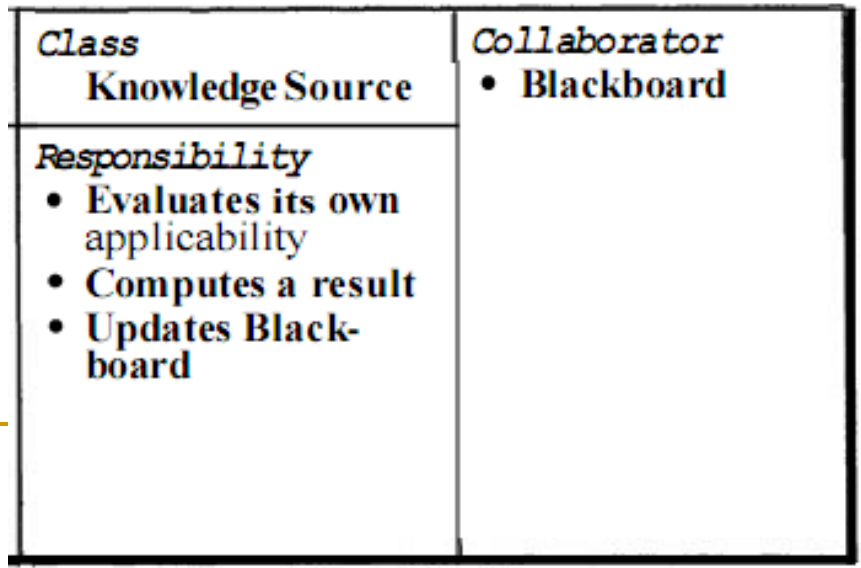
- 黑板
  - 中心数据仓库。解空间的元素和控制数据都存储于此。提供接口使得知识源可以读取和写入
  - 在问题求解过程中构造并放在黑板上的解，称为假设(hypothesis)或黑板条目(blackboard entry)
    - 后期被否决的假设需要从黑板中移除
    - 假设的属性：
      - 抽象层次 abstraction level
      - 真实度 Estimated degree of truth of the hypothesis
      - 时间区间 Time interval covered by the hypothesis

5. 结构

- 黑板
  - 语音识别的例子中
    - 解空间由…构成
      - Acoustic-phonetic and linguistic speech fragments
    - 抽象层次
      - 符号参数、声波-语音片断、音素、音节、单词和短语
    - 音节的真实度根据音节的理想音素序列和假设的音素间的匹配质量来估计

5. 结构

- 知识源 Knowledge source
  - 求解整个问题的特定方面，分离的独立的子系统
  - 知识源之间不直接交流
  - 知识源往往在两个抽象层次上工作
    - 前向推理：低层次➔高层次
    - 反向推理：高层次➔低层次
  - 知识源构成
    - 条件部分和行动部分

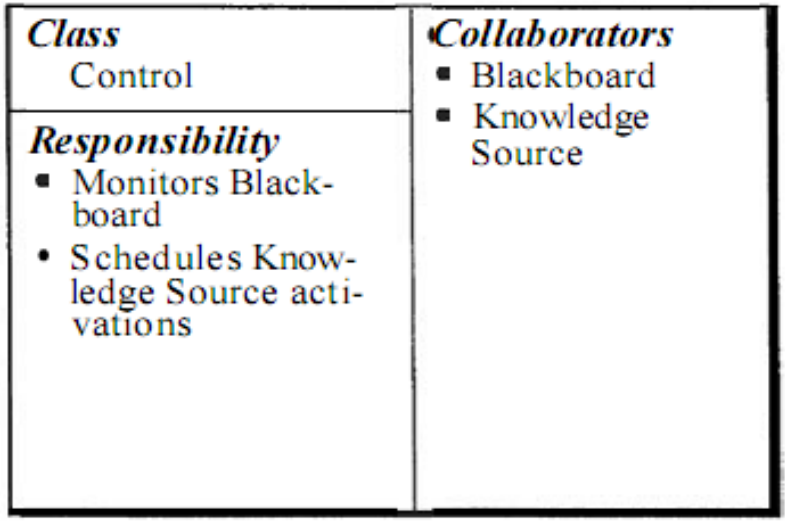


5. 结构

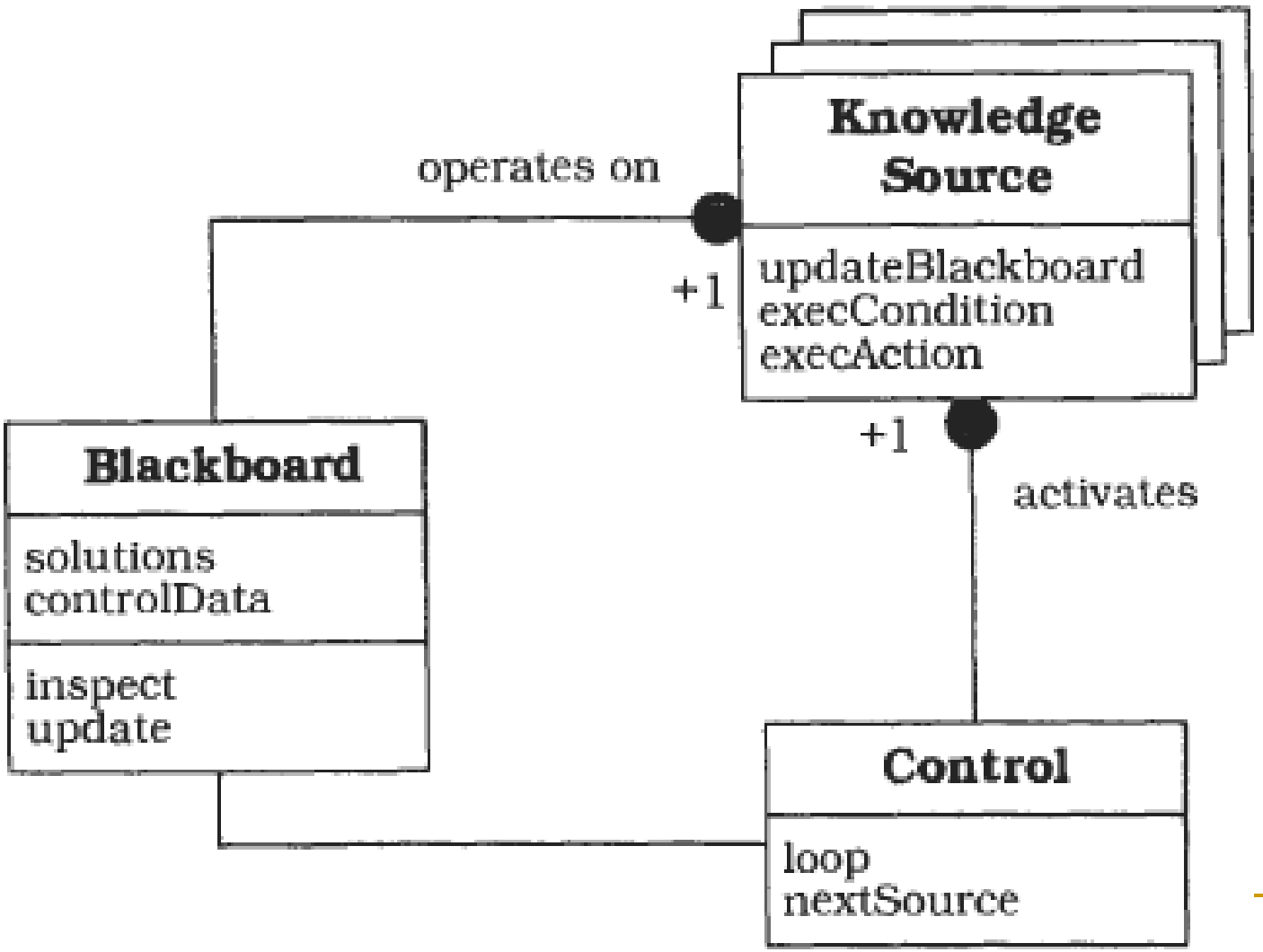
- 知识源 Knowledge source
  - 语音识别的例子中
    - 为以下部分问题明确给出求解方案
      - 定义声波-语音片断、创建音素、音节、词和短语
    - 上述每个问题，定义一个或几个知识源

5. 结构

- 控制组件
  - 循环运行。监视黑板上的改动，并决定下一步的动作。
  - 根据一个知识应用“策略”安排知识源评估和行动
    - 控制知识源 control knowledge source
  - 没有知识源可以应用的状态
    - 引入新的假设
  - 系统停止条件
    - 可接受的假设
    - Or
    - 系统空间或时间资源耗尽



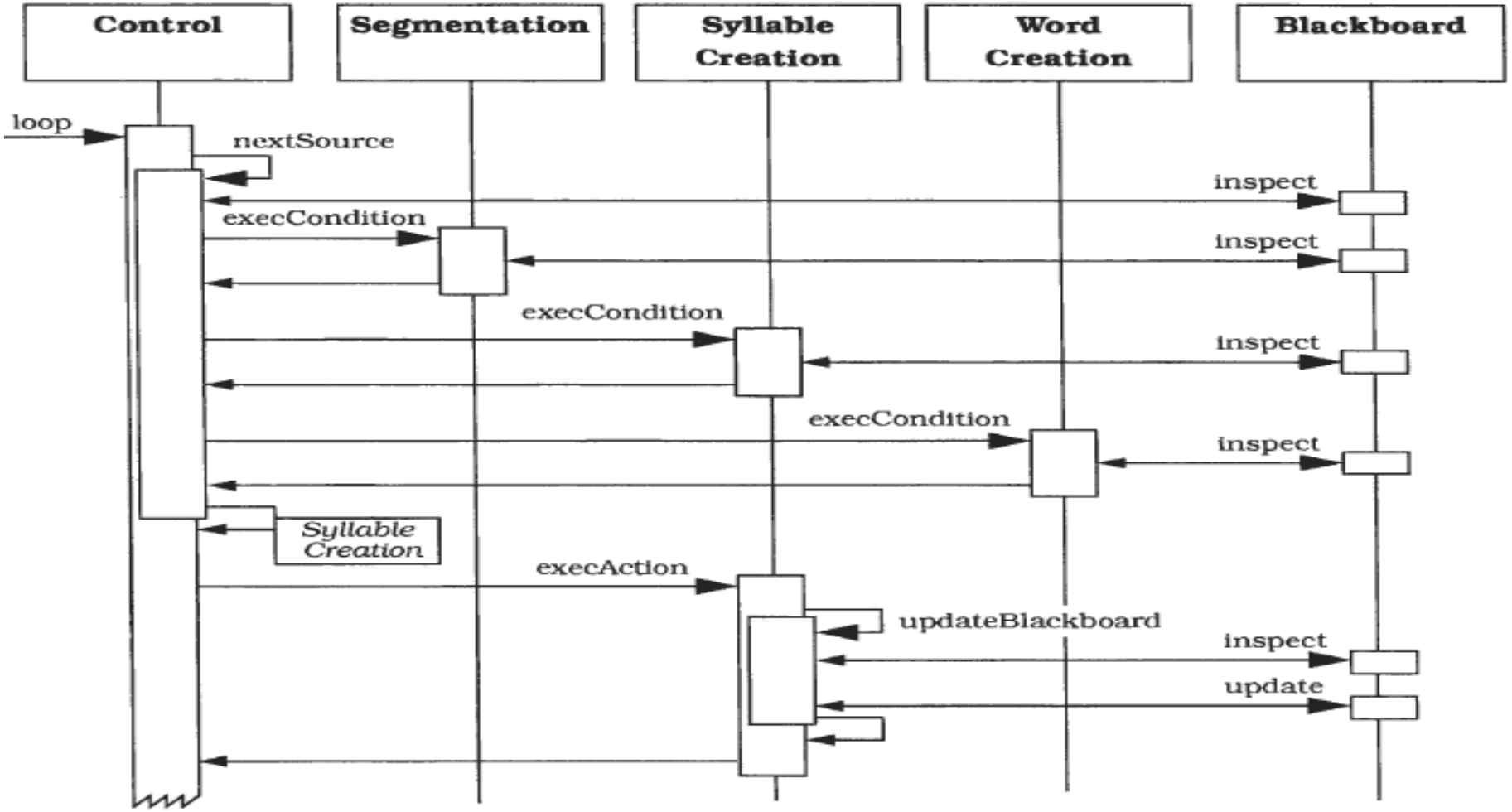
5. 结构


Knowledge Source
updateBlackboard
execCondition
execAction
operates on
+1
+1
activates
Blackboard
solutions
controlData
inspect
update
Control
loop
nextSource

6. 动态特性

- The main loop of the Control component is started.
- Control calls the **nextsource** ( ) procedure to select the next knowledge source.
- **nextsource** ( ) first determines which knowledge sources are potential contributors by observing the blackboard. In this example we assume the candidate Knowledge sources are Segmentation. Syllable Creation and Word Creation.
- **nextsource** ( ) invokes the condition-part of each candidate knowledge source. In the example, the condition-parts of Segmentation. Syllable Creation and Word Creation inspect the blackboard to determine if and how they can contribute to the current state of the solution.
- The Control component chooses a Knowledge source to invoke, and a hypothesis or a set of hypotheses to be worked on

6. 动态特性


Control
Segmentation
Syllable Creation
Word Creation
Blackboard
loop
nextSource
inspect
execCondition
inspect
execCondition
inspect
execCondition
inspect
Syllable Creation
execAction
updateBlackboard
inspect
update

7. 实现

1) 定义问题

- Specify the domain of the problem and the general fields of knowledge necessary to find a solution.
- Scrutinize the input to the system. Determine any special properties of the input such as noise content or variations on a theme that is, does the input contain regular patterns that change slowly over time?
- Define the output of the system. Specify the requirements for correctness and fail-safe behavior. If you need an estimation of the credibility of the results, or if there are cases in which the system should ask the user for further resources, record this.
- Detail how the user interacts with the system.

7. 实现

1) 定义问题

- 语音识别问题
  - 知识领域：声学、语言学和统计学
  - 输入：一个说话者的声音信号序列
  - 输出：对应所说短语的一个书面的英文短语
  - 容错性
    - 用于数据库查询接口时， 10%

7. 实现

2) 定义问题的解空间

- Specify exactly what constitutes a top-level solution.
- List the different abstraction levels of solutions.
- Organize solutions into one or more abstraction hierarchies.
- Find subdivisions of complete solutions that can be worked on independently, for example words of a phrase or regions of a picture or area.

7. 实现

2) 定义问题的解空间

- 语音识别问题
  - 完整的顶层解
    - 正确的短语（相对于已定义的词汇和语法来说）
  - 完整的中间解
    - 声波-语音序列
    - 描述整个口语片断的语言学元素序列
  - 部分解
    - 元素本身

7. 实现

3) 将求解过程分成几个步骤

- Define how solutions are transformed into higher-level solutions.
- Describe how to predict hypotheses at the same abstraction level.
- Detail how to verify predicted hypotheses by finding support for them in other levels.
- Specify the kind of knowledge that can be used to exclude parts of the solution space.

7. 实现

3)求解过程分成几个步骤

- 语音识别问题
  - 音节层的解转换单词层的解
    - 提供一个词典，把一个音节和所有发音中包含此音节的词相关联
  - 语法和统计学的知识对词序列搜索剪枝是有用的
    - E.g. 一个形容词后通常更另一个形容词或一个名词

7. 实现

4)把知识分成和特定子任务相关联的专门知识源

- 语音识别问题
  - 分片、音素创建、音节创建、词创建、短语创建、词预测、词验证等

5)定义黑板的词汇

- 语音识别问题

```
ABOUT+FEIGENBAUM+AND+FELDMAN+] (phrase)(48:225) (83)
```

7. 实现

6) **Specify the control of the system**

- Classifying changes to the blackboard into two types. One type specifies all blackboard changes that may imply a new set of applicable knowledge sources, the other specifies all blackboard changes that do not. After changes of the second type, the Control component chooses a knowledge source without another invocation of all condition-parts.
- Associating categories of blackboard changes with sets of possibly applicable knowledge sources.
- Focusing of control. The focus contains either partial results on the blackboard that should be worked on next, or knowledge sources that should be preferred over others.
- Creating a queue in which knowledge sources classified as applicable wait for their execution. By using a queue, you save valuable information about knowledge sources rather than discarding it after each change to the blackboard.

7. 实现

7) 实现知识源

- 根据控制组件的需要，把知识源分成条件部分和动作部分
- 为维护知识源的独立性和可替换性，不能做出任何与其他知识源或控制组件有关的假设
- 知识源自身可以根据各种体系结构模式或涉及模式来组织

8. 变体

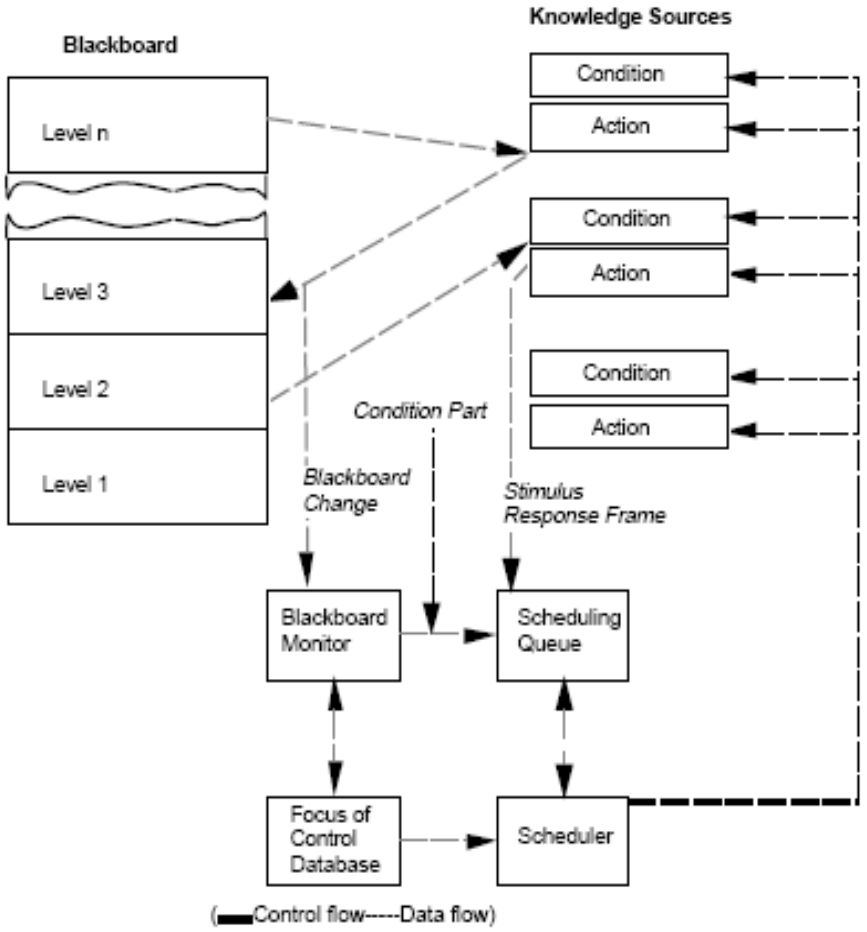
- 产生式系统 ***Production*** System
  - 子程序用“条件—动作”规则来表示
  - 当条件满足并选择了这条规则后才执行相应的动作
  - 选择由“冲突消解模块”做出的

8. 变体

- 仓库 Repository
  - This variant is a generalization of the Blackboard pattern. The central data structure of this variant is called a repository.
  - In a Blackboard architecture the current state of the central data structure, in conjunction with the Control component, finally activates knowledge sources.
  - In contrast, the Repository pattern does not specify an internal control. A repository architecture may be controlled by user input or by an external program.
  - A traditional database, for example, can be considered as a repository. Application programs working on the database correspond to the knowledge sources in the Blackboard architecture.

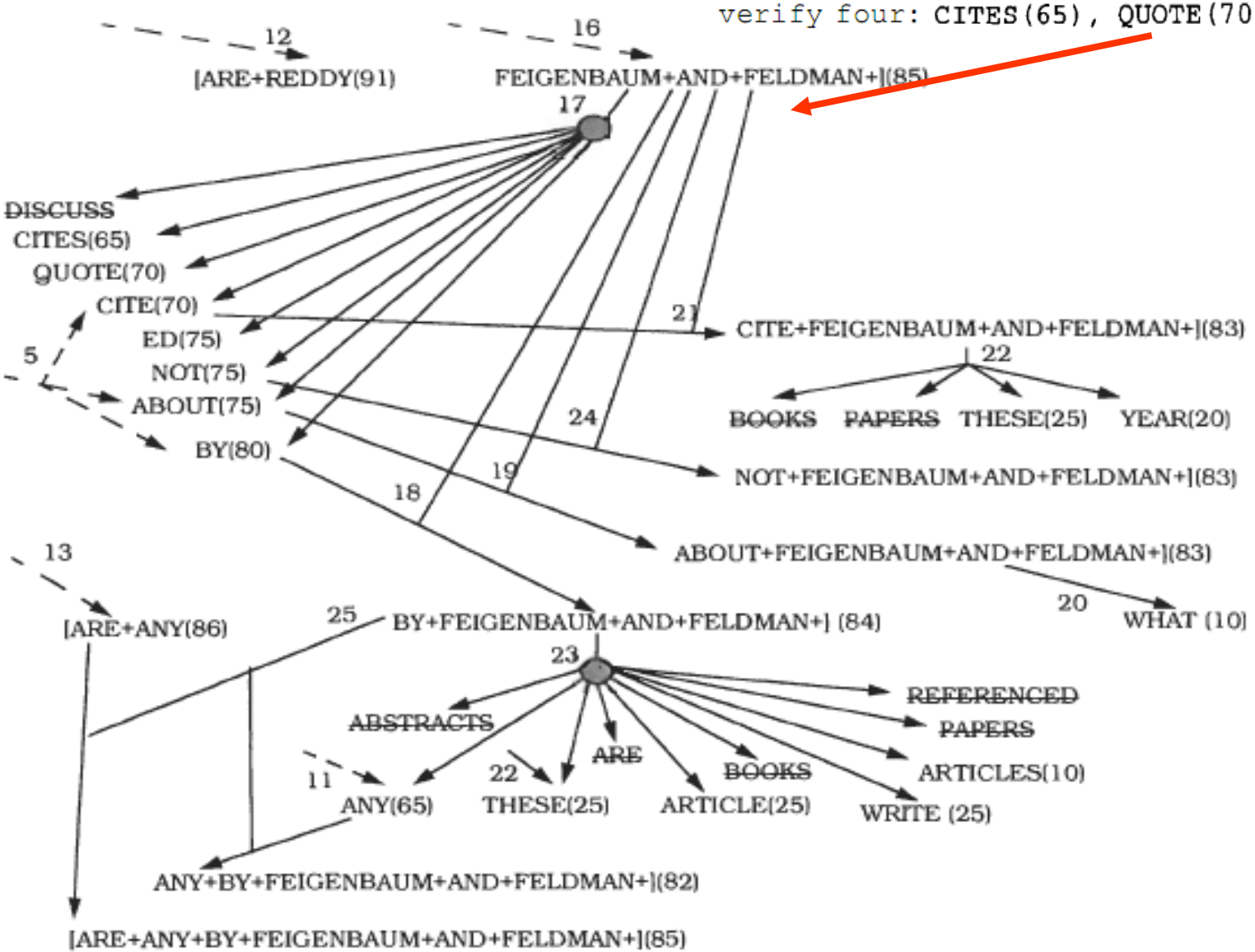
9. 已知应用

- **HEARSAY-II**
- **HASP/SIAP**
- **CRYSALIS**
- **TRICERO**
- **Generalization**
- **SUS**



10. 已解决的例子

Step 17: KS PREDICT&VERIFY
Stimulus: FEIGENBAUM+AND+FELDMAN+](85) (phrase).
Action: Predict eight preceding words;
reject one: DISCUSS;
find three already on the blackboard: CITE(70), ABOUT(75), BY(80);
verify four: CITES(65), QUOTE(70), ED(75), NOT(75).



11. 效果

- 优点
  - ***Experimentation.*** In domains in which no closed approach exists and a complete search of the solution space is not feasible, the Blackboard pattern makes experimentation with different algorithms possible, and also allows different control heuristics to be tried.
  - ***Support for changeability and maintainability***. The Blackboard architecture supports changeability and maintainability because the individual knowledge sources, the control algorithm and the central
  - ***Reusable knowledge sources.***
  - ***Support for fault tolerance and robustness.***

11. 效果

- 不足
  - ***Difficulty of testing***. Since the computations of a Blackboard system do not follow a deterministic algorithm, its results are often not reproducible. In addition, wrong hypotheses are part of the solution process.
  - ***No good solution is guaranteed***. Usually Blackboard systems can solve only a certain percentage of their given tasks correctly.
  - ***Difficulty of establishing a good control strategy.*** The control strategy cannot be designed in a straightforward way, and requires an experimental approach.
  - ***Low Efficiency***. Blackboard systems suffer from computational overheads in rejecting wrong hypotheses. If no deterministic algorithm exists, however, low efficiency is the lesser of two evils when compared to no system at all.
  - ***High development effort***. Most Blackboard systems take years to evolve. We attribute this to the ill-structured problem domains and extensive trial-and-error programming when defining vocabulary, control strategies and knowledge sources.
  - ***No support for parallelism***. The Blackboard architecture does not prevent the use of a control strategy that exploits the potential parallelism of knowledge sources. It does not however provide for their parallel execution. Concurrent access to the central data on the blackboard must also be synchronized.

2.3 分布式系统

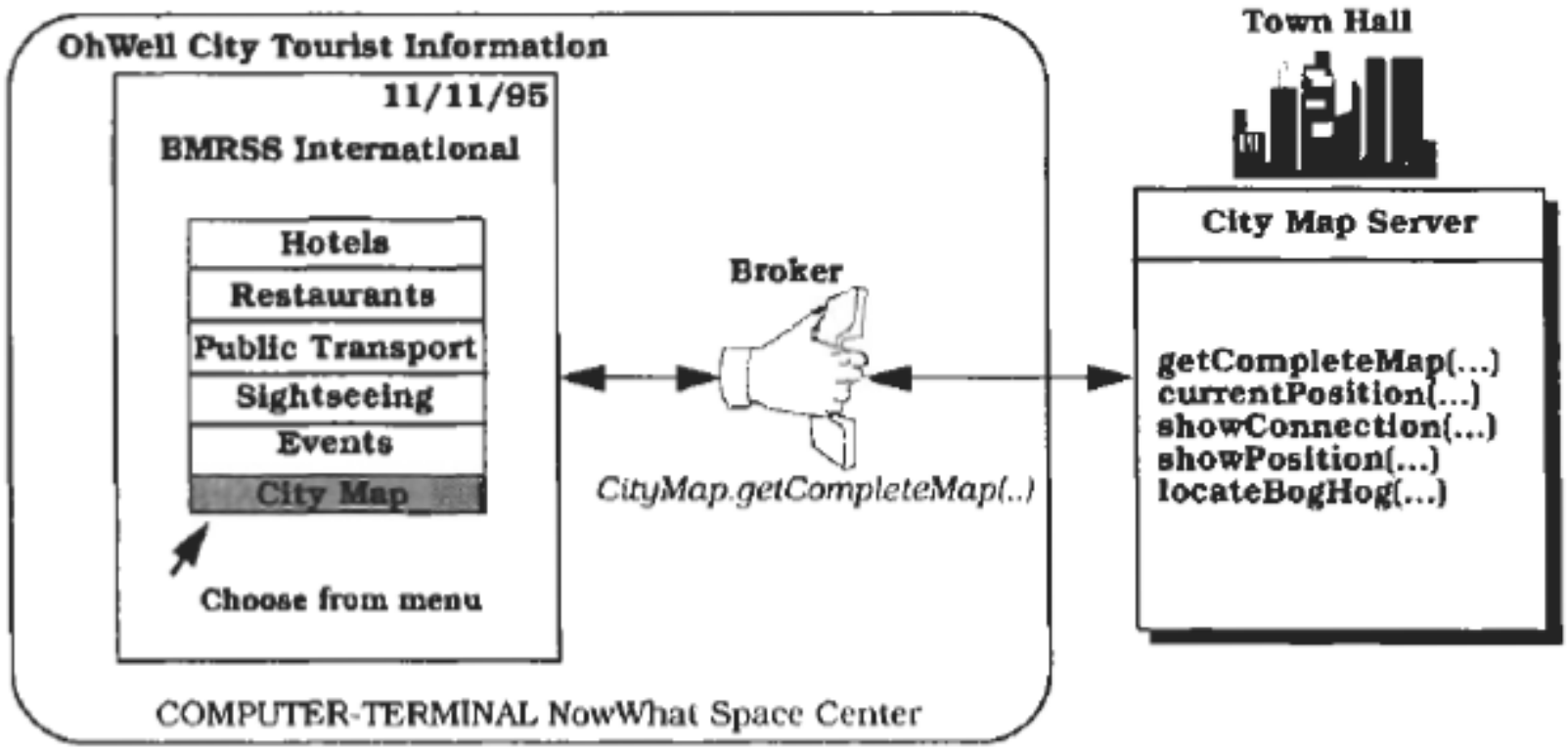
- 优点
  - *经济性* ***Economics***
  - *性能和可扩展性* ***Performance and Scaleability.***
  - *固有分布性* ***Inherent disbibution***
  - *可靠性* ***Reliabillty.***
- 三种与分布式系统相关的模式
  - 管道和过滤器 ***Pipes and Filters***
  - 微核 ***Microkernel***
  - 代理者 ***Broker***

2.3.1代理者 Broker

- 代理者（Broker）模式可以用于构建带有隔离组件的分布式软件系统，该软件通过远程服务调用进行交互。代理者组件负责协调通信，诸如转发请求，以及传送结果和异常。

- The Broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations.
- **A** broker component is responsible for coordinating communication, such as forwarding requests. as well as for transmitting results and exceptions.

1. 例子


OhWell City Tourist Information
11/11/95
BMRS8 International
Hotels
Restaurants
Public Transport
Sightseeing
Events
City Map
Choose from menu
COMPUTER-TERMINAL NowWhat Space Center
Broker
CityMap.getCompleteMap(..)
Town Hall
City Map Server
getCompleteMap(...)
currentPosition(...)
showConnection(...)
showPosition(...)
locateBogHog(...)

2. 语境

- 你的环境是带有独立协作组件的，分布式的并可能是异构的系统

  Your environment is a distributed and possibly heterogeneous system with independent cooperating components.

3. 问题

- ***Forces*** to balance
  - 组件应该能够访问其他组件通过远程、地点透明的服务调用提供的服务

    Components should be able to access services provided by others through remote, location-transparent service invocations.
  - 需要在运行期间交换、添加或移动组件

    You need to exchange, add, or remove components at run-time.
  - 该体系结构应该向用户隐藏特定系统和特定实现的细节

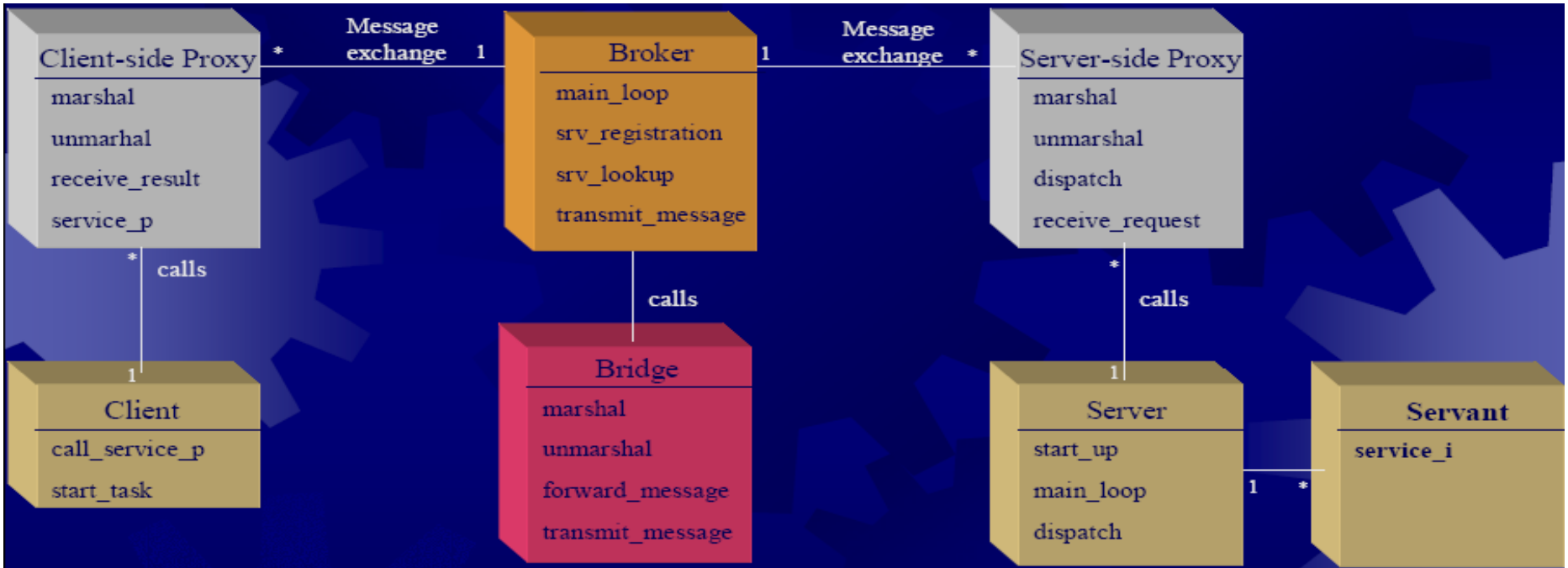
    The architecture should hide system- and implementation-specific details from the users of components and services.

4. 解决方案

- Introduce a ***broker*** component to achieve better decoupling of clients and servers. *Servers register themselves with the broker, and make their services available to clients through method interfaces.* Clients access the functionality of servers by sending requests via the broker. **A** broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

5. 结构

- 6种参与组件
  - 客户、服务器、代理者、桥接、客户端代理、服务器端代理



5. 结构

Class Client	**Collaborators**
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	• Client-side Proxy • Broker

Class Server	**Collaborators**
Responsibility • Implements services. • Registers itself with the local broker. • Sends responses and exceptions back to the client through a server-side proxy.	• Server-side Proxy • Broker

Class Broker	**Collaborators**
Responsibility • (Un-)registers servers. • Offers APIs. • Transfers messages. • Error recovery. • Interoperates with other brokers through bridges. • Locates servers.	• Client • Server • Client-side Proxy • Server-side Proxy • Bridge

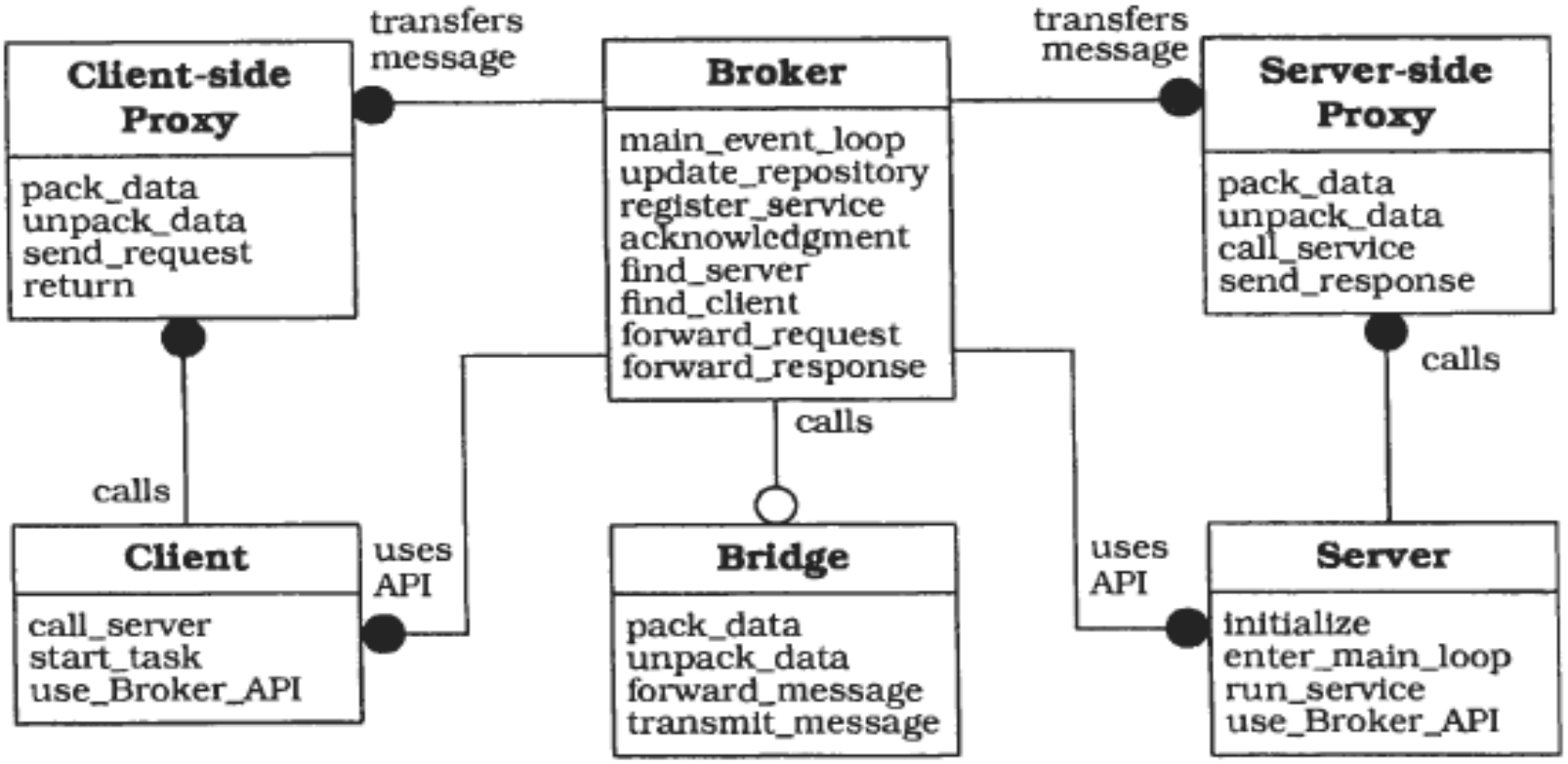
5. 结构

Class Client-side Proxy	Collaborators • Client • Broker
Responsibility • Encapsulates system-specific functionality. • Mediates between the client and the broker.	

Class Server-side Proxy	Collaborators • Server • Broker
Responsibility • Calls services within the server. • Encapsulates system-specific functionality. • Mediates between the server and the broker.	

Class Bridge	Collaborators • Broker • Bridge
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	

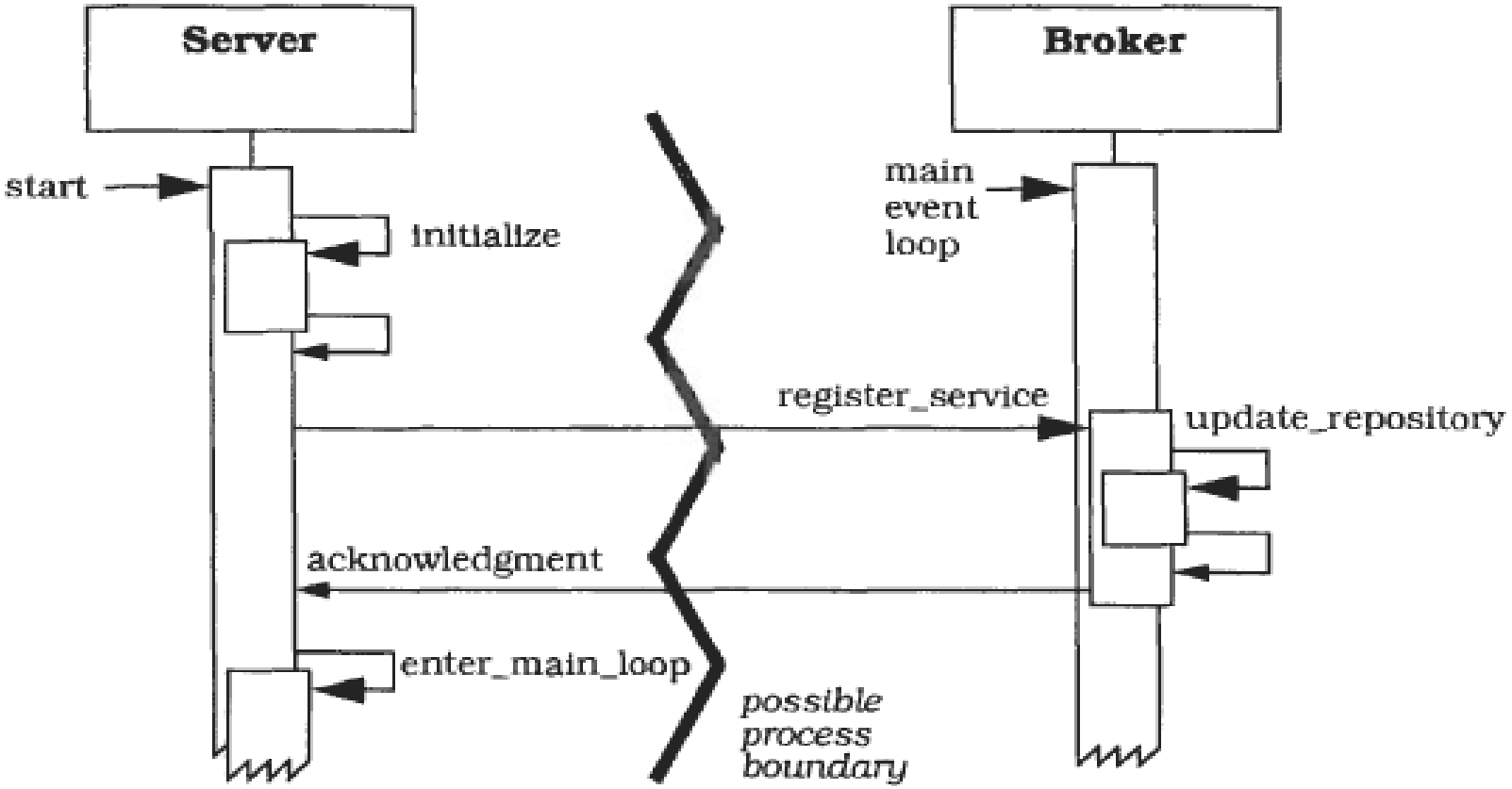
5. 结构


Client-side Proxy
pack_data
unpack_data
send_request
return
transfers message
Broker
main_event_loop
update_repository
register_service
acknowledgment
find_server
find_client
forward_request
forward_response
transfers message
Server-side Proxy
pack_data
unpack_data
call_service
send_response
calls
calls
calls
Client
call_server
start_task
use_Broker_API
uses API
Bridge
pack_data
unpack_data
forward_message
transmit_message
uses API
Server
initialize
enter_main_loop
run_service
use_Broker_API

6. 动态特性—场景I

- 服务器向本地代理者组件注册自己
  - The broker is started in the initialization phase of the system. The broker enters its event loop and waits for incoming messages.
  - The user, or some other entity, starts a server application. First, the server executes its initialization code. After initialization is complete, the server registers itself with the broker.
  - The broker receives the incoming registration request from the server. It extracts all necessary information from the message and stores it into one or more repositories. These repositories are used to locate and activate servers. An acknowledgment is sent back.
  - After receiving the acknowledgment from the broker, the server enters its main loop waiting for incoming client requests.

6. 动态特性—场景I


Server
Broker
start
initialize
main
event
loop
register_service
update_repository
acknowledgment
enter_main_loop
possible
process
boundary

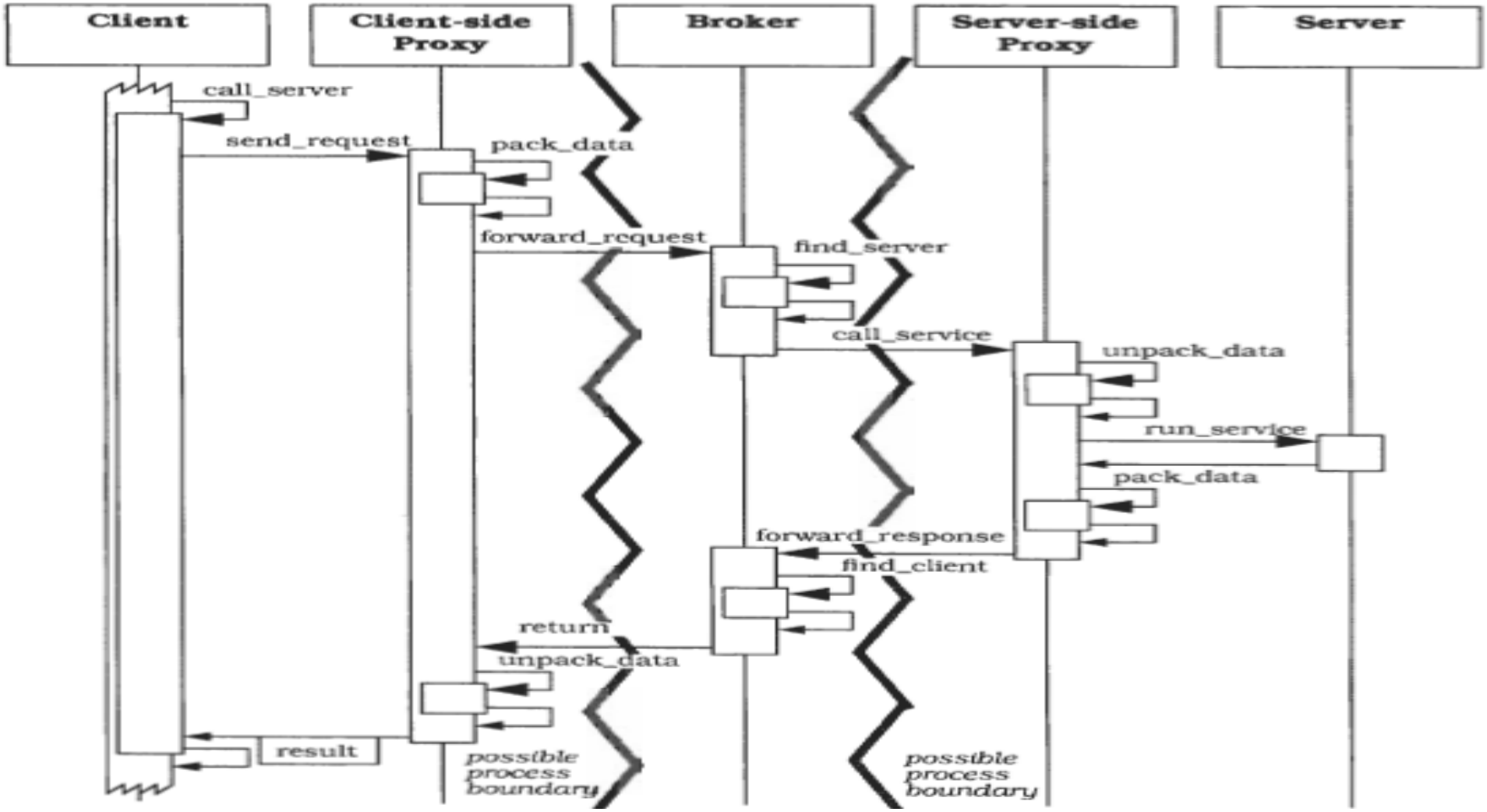
6. 动态特性—场景II

- 客户机向本地服务器发送请求
  - The client application is started. During program execution the client invokes a method of a remote server object.
  - The client-side proxy packages all parameters and other relevant information into a message and forwards this message to the local broker.
  - The broker looks up the location of the required server in its repositories. Since the server is available locally, the broker forwards the message to the corresponding server-side proxy. For the remote case, see the following scenario.
  - The server-side proxy unpacks all parameters and other information, such as the method it is expected to call. The server-side proxy invokes the appropriate service.

6. 动态特性—场景II

- After the service execution is complete, the server returns the result to the server-side proxy, which packages it into a message with other relevant information **and** passes it to the broker.
- The broker forwards the response to the client-side proxy.
- The client-side proxy receives the response, unpacks the result and returns to the client application. The client process continues with Its computation.

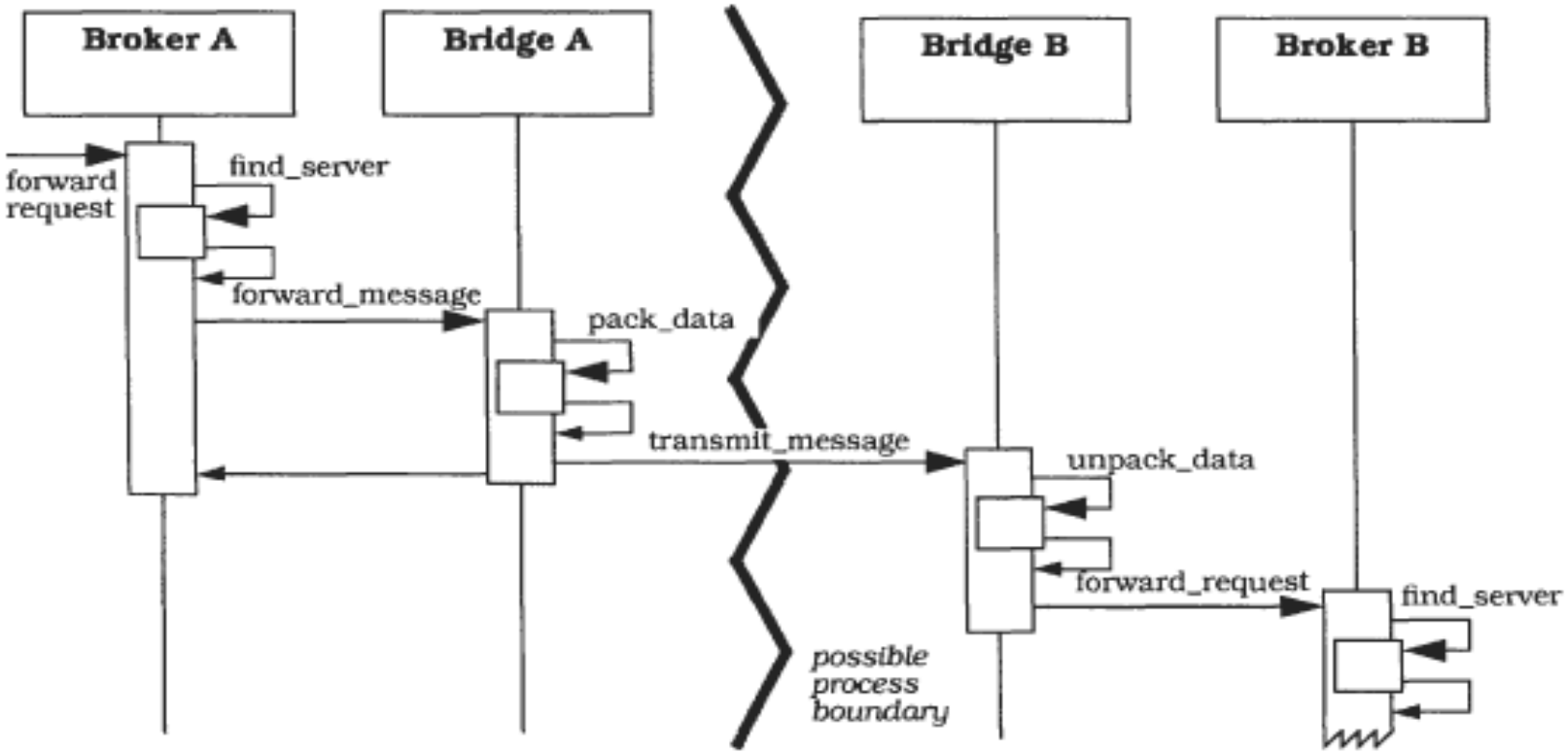
6. 动态特性—场景II


Client
Client-side Proxy
Broker
Server-side Proxy
Server
call_server
send_request
pack_data
forward_request
find_server
call_service
unpack_data
run_service
pack_data
forward_response
find_client
return
unpack_data
result
possible process boundary
possible process boundary

6. 动态特性—场景III

- 通过桥接组件不同代理者进行交互
  - **Broker A receives an incoming request**. It locates the server responsible for executing the specified service by looking it up in the repositories. Since the corresponding server is available at another network node, the broker forwards the request to a remote broker.
  - **The message is passed from Broker A to Bridge A.** This component is responsible for converting the message from the protocol defined by Broker **A** to a network-specific but common protocol understood by the two participating bridges. After message conversion, Bridge **A** transmits the message to Bridge B.
  - **Bridge B maps the incoming request from the network-specific format to a Broker B-specific format.**
  - ***Broker B performs all the actions necessary when a request arrives, as described in the &st step of this scenario***

6. 动态特性—场景III


Broker A
Bridge A
Bridge B
Broker B
forward request
find_server
forward_message
pack_data
transmit_message
unpack_data
forward_request
find_server
possible process boundary

7. 实现

- ***Define*** an object model, or use an existing model.
- ***Decide which kind of component-interoperability the system should offer.***
- Specify the APIs the broker component provides for collaborating with clients **and** servers.
- *Use proxy objects to hide implementation details from clients and servers.*
  - Client-side proxies package procedure calls into messages and forward these messages to the local broker component.
  - Server-side proxies receive requests from the local broker and call the methods in the interface implementation of the corresponding server.

7. 实现

- ***Design the broker component in parallel with steps 3 and 4.***
  - 5.1 Specify a detailed *on-the-wire* protocol for interacting with client-side proxies and server-side proxies.
  - 5.2 **A** local broker must be available for every participating machine in the network.
  - 5.3 When a client invokes a method of a server, the Broker system is responsible for returning all results and exceptions back to the original client.
  - 5.4 If the proxies (see step 4) do not provide mechanisms for marshaling and unmarshaling parameters and results, you must include that functionality in the broker component.
  - 5.5 If your system supports asynchronous communication between clients and servers, you need to provide *message buffers* within the broker or within the proxies for the temporary storage of messages.

7. 实现

- 5.6 Include a *directory service* for associating local server identifiers with the physical location of the corresponding servers in the broker. For example, if the underlying inter-process communication protocol is based on TCP/IP, you could use an Internet port number as the physical server location.
- 5.7 When your architecture requires system-unique identifiers to be generated dynamically during server registration, the broker must offer a *name service* for instantiating such names.
- 5.8 If your system supports *dynamic method invocation* (see step 3), the broker needs some means for maintaining type information about existing servers. A client may access this information using the broker APIs to construct a request dynamically. You can implement such type information by instantiating the Reflection pattern (193). In this, metaobjects maintain type information that is accessible by a metaobject protocol.

7. 实现

- **5.9** Consider the case in which something fails. In a distributed system two levels of errors may occur:
  - A component such as a server may run into an error condition. This is the same kind of error you encounter when executing conventional non-distributed applications.
  - The communication between two independent processes may fail. Here the situation is more complicated, since the communicating components are running asynchronously.
- Develop *IDL* compilers.

9. 变体

- 直接通信代理者系统Direct ***Communication Broker System***
  - *客户机可以直接与服务器通信*
- ***消息传送代理者系统Message Passing Broker System***
  - 适合于以数据传输为重点的系统
- ***交易器系统Trader System***
  - 客户机请求的目标是服务，而不是服务器
  - 客户端代理访问时使用服务器标识符，而不是服务器标识符
- 适配器代理者系统Adapter Broker System
  - 使用额外的适配器层将代理者组件的接口隐藏到服务器
- ***回调代理者系统Callback*** Broker System
  - 事件驱动的，对客户机和服务器不区分
  - 事件到来时，代理者将调用已注册组件的回调函数，对事件做出反应

10. 已知应用

- ***CORBA***
- ***IBM SOM/DSOM.***
- ***Microsoft's OLE 2.x***
- ***The World Wide Web***
- ***ATM-P***

11. 效果

- 优点
  - 定位透明性
  - 组件的可变性和可扩展性
  - 代理者系统的可移植性
  - 不同代理者系统之间的互操作性
  - 可重用性

11. 效果

- 不足
  - 效率受限
  - 容错性较差

- **benefits** as well as **liabilities**
  - 测试和调试

参见

- 转发器-接收器模式封装了两个组件之间的过程通信

  The *Forwarder-Receiver* pattern (307) encapsulates inter-process communication between two components.
- 代理模式有几种情况，远程情况是其中之一

  The *Proxy* pattern **(263)** comes in several flavors, the *remote* case being one of them.
- 客户机-分配器-服务器模式是直接通信代理者变体的一个轻量级模式

  The *Client-DispatcherServer* pattern **(323)** is a lightweight version of the Direct Communication Broker variant.