中国科学技术大学

University of Science and Technology of China

模式和软件体系结构

主要内容

- 模式
- 软件体系结构概念
- 软件体系结构风格（架构模式）
- 设计模式

模式

模式

- 人类专家在解决一个特定问题时，很少一上来就使用全新的解决方法，一般总是寻找是否有既有解决类似问题的方案，并试图将该方案的精髓用于解决新问题。
  - 如：建筑领域、经济领域、软件工程
- 从特定问题-解决方案中提炼出通用的因素即为模式Patterns
  - 每个模式都是由三部分组成的，描述了特定背景context、重复发生的问题problem和解决方案solution之间的关系

软件模式

- 软件模式描述了在特定设计情形下反复出现的设计问题，并提供了已得到充分证明的通用解决方案摘要。解决方案摘要描述模式的组件、组件的职责和关系，以及这些组件协作的方式。
- 模式的组成：
  - 背景（context）　　问题出现的背景
  - 问题（problem）　　该背景下反复出现的问题
  - 解决方案（solution）　经过实践检验的解决之道
- 例如MVC模式

软件模式的特征

- 模式阐述了在特定设计环境下反复出现的设计问题，并提供一个解决方案
- 模式记录了已得到充分证明的既有设计经验
- 模式描述了超越单个类、实例和组件层次上的抽象
- 模式为设计原则提供了一种公共的词汇和理解，使大家对设计原则有一致的认识，便于交流和讨论
- 模式提供一种将软件结构文档化的手段
- 模式有助于创建具有特定特征的软件，通过提供一个功能行为的基本骨架，有助于实现应用程序的功能
- 模式有助于建立一个复杂和异构的软件体系结构
- 模式有助于控制软件的复杂度

软件模式的分类

- 体系结构模式（Architectural Pattern）
  - An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
  - 描述了软件系统的基本结构概要（schema），提供了一组预定义的子系统、指定子系统职责，以及组织子系统的规则和指南
  - 是具体软件架构的模板，描绘了软件的系统级结构特征，也影响到子系统的架构设计

软件模式的分类

- 设计模式（Design Pattern）
  - A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.
  - 提供了对软件系统的子系统、组件和它们之间的关系进行精化的概要，它描述了交互组件经常出现的结构，这些组件用以解决特定背景下的一般性设计问题
  - 设计模式是中型模式，不依赖于编程语言和编程范式，对子系统的架构有很大的影响

软件模式的分类

- 惯用法（Idiom）
  - An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.
  - 是针对某种编程语言的低层模式，描述了使用给定语言的特征来实现组件、组件间关系的某些特定方面
  - 描述了某一或某些程序语言广泛接受的编程风格、习惯等

模式的应用

- 体系结构模式可以用在大粒度设计的开始
- 设计模式可以用在整个设计阶段
- 编码级模式一般在实现阶段使用

- 模式通常都不是原子的
  - 大多数的软件体系结构模式带来的问题可以用更小的模式来解决
  - 模式还可能是其他模式的变种

软件体系结构概念

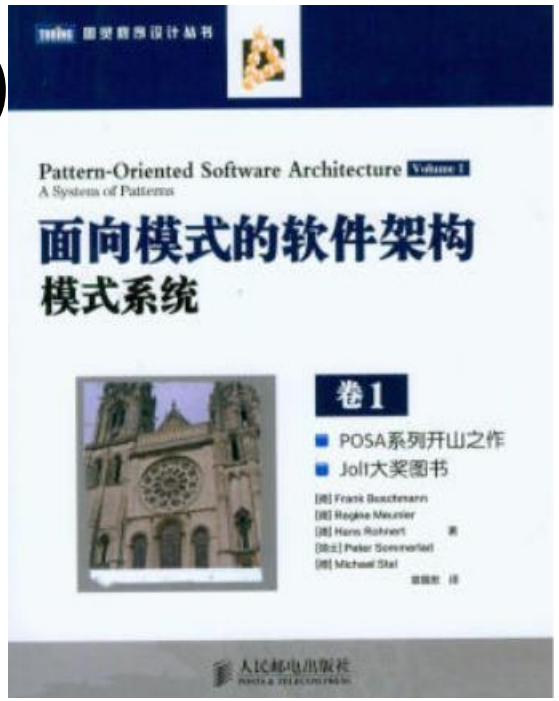
软件体系结构定义

- D. Garlan和M. Shaw：软件体系结构应包含对组成系统的组件的描述、组件之间的交互关系以及系统组合的模式和应该满足的约束等。
- D. E. Perry和A. L. Wolf：软件体系结构是由元素（element）、形式（form）和准则（rational）构成，其中，元素分成三类：处理元素（processing element）、数据元素（data element）和连接元素（connecting element），形式是由特性（property）和关系（relationship）组成。
- IEEE 610.12-1990软件工程标准词汇：软件体系结构是以组件、组件之间的关系、组件与环境之间的关系为内容的某一系统的基本组织结构，它还包括指导上述内容设计和演化的原理。

软件体系结构作用

- 经历了两个阶段：
  - 二十世纪八十年代左右，软件体系结构的描述随意性很大
  - 九十年代以来，软件体系结构已经正式作为软件开发中的一个设计活动，“架构设计师”也已成为了一个正式的技术职位，用于描述软件体系结构的语言和工具得到了迅速发展；出现了相应的不同体系结构标准
- 体系结构作用：
  - 增强对系统的理解
  - 支持软件复用
  - 便于系统构造
  - 便于软件进化
  - 为系统分析提供了新的途径和方法
  - 便于项目管理

软件体系结构风格
（架构模式）



软件体系结构风格经典分类

类别	名称
调用-返回 (Call and Return)	分层 (Hierarchical Layer)
	主程序-子程序 (main program and subroutine)
	面向对象(OO systems)
独立组件 (Independent components)	通讯进程(communicating processes)
	事件系统(event systems)
虚拟机 (Virtual Machine)	解释器(interpreters)
	规则为中心(rule-based systems)
数据流 (Dataflow)	批处理(batch sequential)
	管道-过滤器(pipes and filters)
数据为中心 (Data-centered)	数据库(database)
	黑板(blackboard)

架构模式的分类

类别	目的	名称	特点	例子
从混沌到结构 (From Mud to Structure)	支持把整个系统任务以可控方式分解成相互协作的子任务	分层(Layer)	.直观的分而治之方式 .层的重用 .依赖性局部化 .可替代性	通信协议栈 Java虚拟机 很多应用系统
		管道-过滤器 (Pipes and Filters)	.重用性好 .便于重组新策略 .交互性差 .适用于复杂处理	编译器 Unix Shell DirectShow 通信协议解析
		黑板(Blackboard)	.算法和数据分离 .耦合度大 .适用于某些AI系统	人工智能系统
分布式系统 (Distributed System)	为分布式应用提供完备的基础架构（涉及到微核、管道和过滤器模式）	代理者 (Broker)	.屏蔽复杂性 .支持互操作 .适用于分布式系统	CORBA RMI 交叉编译系统
交互式系统 (Interactive System)	支持具有人机交互特征的软件系统的构建	MVC (Model-View-Controller)	.灵活性 .广泛流行	很多
		PAC (Presentation-Abstraction-Control)	.灵活性 .复杂	人工智能系统
适应性系统 (Adaptive System)	支持应用的扩展以适应技术的进步和变需求的变更	微内核 (Microkernel)	.适应变化 .支持长生命周期 .复杂	OS JBoss Eclipse
		基于元数据 (Meta-Level-Architecture)	.适应变化 .支持长生命周期 .不易实现	反射框架 Spring MDA

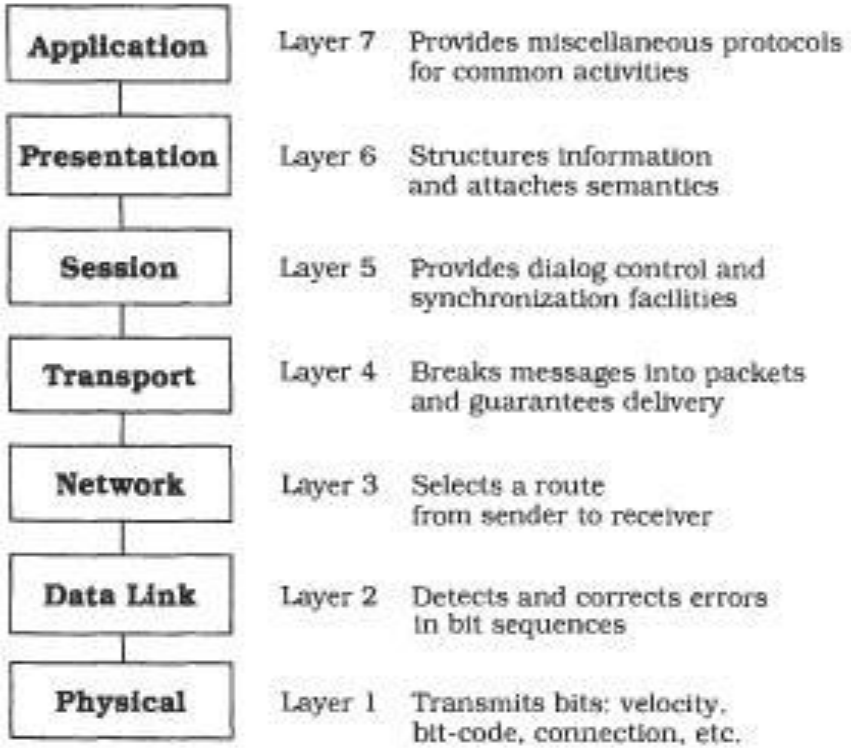
“从混沌到结构”模式

- 支持把整个系统以可控方式分解成相互协作的子任务

- 典型模式
  - Layers：有助于将应用划分为多组子任务，其中每组子任务都位于特定抽象层
  - Pipes and Filters：适合用于处理数据流的系统，每个处理步骤都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。通过重组过滤器，可以构建相关的系统族
  - Blackboard：用于解决未有确定解决策略的问题，若干专用子系统通过收集知识来求得部分解或近似解。

Layers模式

- 示例: OSI 7-Layer Model
- Benefits:
  - Aiding development by teams（多团队并行开发）
  - Incremental coding and testing（增量编码与测试）
  - Easier exchange of individual parts（易于替换独立部分）
  - Reuse of individual layers in different layers in different contexts（可以在不同场景下复用不同分层）



Layers模式

- Context:
  - A large system that requires decomposition（需要分解的大型系统）
- Problem:
  - Late source code changes should not ripple through the system (changeable)
  - Interfaces should be stable, and may even be prescribed by a standards body
  - Parts of the system should be exchangeable
  - It may be necessary to build other systems at a later date with the same low-level issues as the system you are currently designing (Reusable)
  - Similar responsibilities should be grouped to help understandability and maintainability. But each component should be coherent（内聚的）.
  - There's no 'standard' component granularity
  - Complex components need further decomposition
  - Crossing component boundaries may impede performance（影响性能）
  - System will be built by a team of programmers and work has to be subdivided along clear boundaries

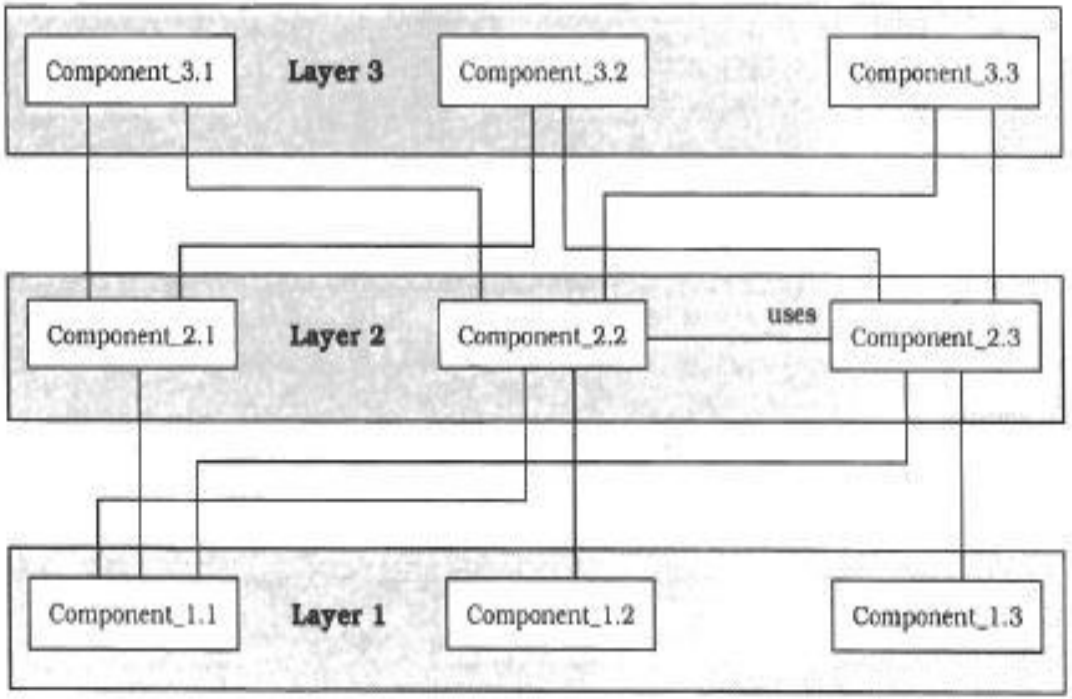
Layers模式

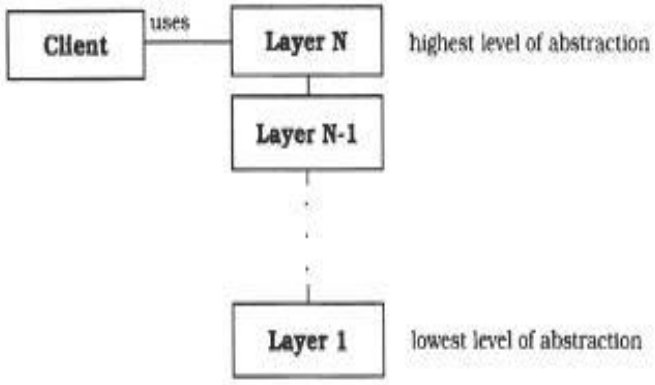
- Solution
  - Structure your system into an appropriate number of layers and place them on top of each other
  - Start at the lowest level of abstraction (Layer 1), which is the base of your system
  - Work your way up the abstraction ladder by putting Layer J on top of Layer J-1 until you reach the top level of functionality (Layer N)
  - Above does not prescribe the order in which to actually design layers, it just gives a conceptual view
  - Within an individual layer all constituent components work at the same level of abstraction
  - Most of the services that Layer J provides are composed of services provided by Layer J-1. The services of each layer implement a strategy for combining the services of the layer below in a meaningful way. In addition, Layer J's services may depend on other services in Layer J

Layers模式

- Structure: by CRC card
  - The services of Layer J are only used by Layer J+1 ---- There are no further direct dependencies between layers

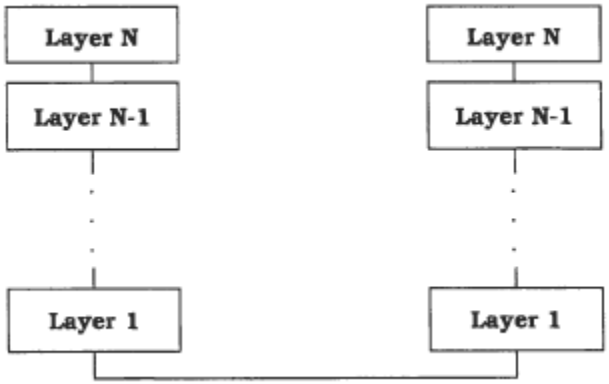
Class Layer J	Collaborator • Layer J-1
Responsibility • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1.	





Layers模式

- Dynamics
  - Case 1：Top-down communication: ‘requests’
  - Case 2：Bottom-up communication: ‘notifications’
  - Case 3：Requests only travel through a subset of the layers: ‘caching layers’
  - Case 4：Notifications only travel through a subset of the layers
  - Case 5：Two stacks of N layers communicating with each other



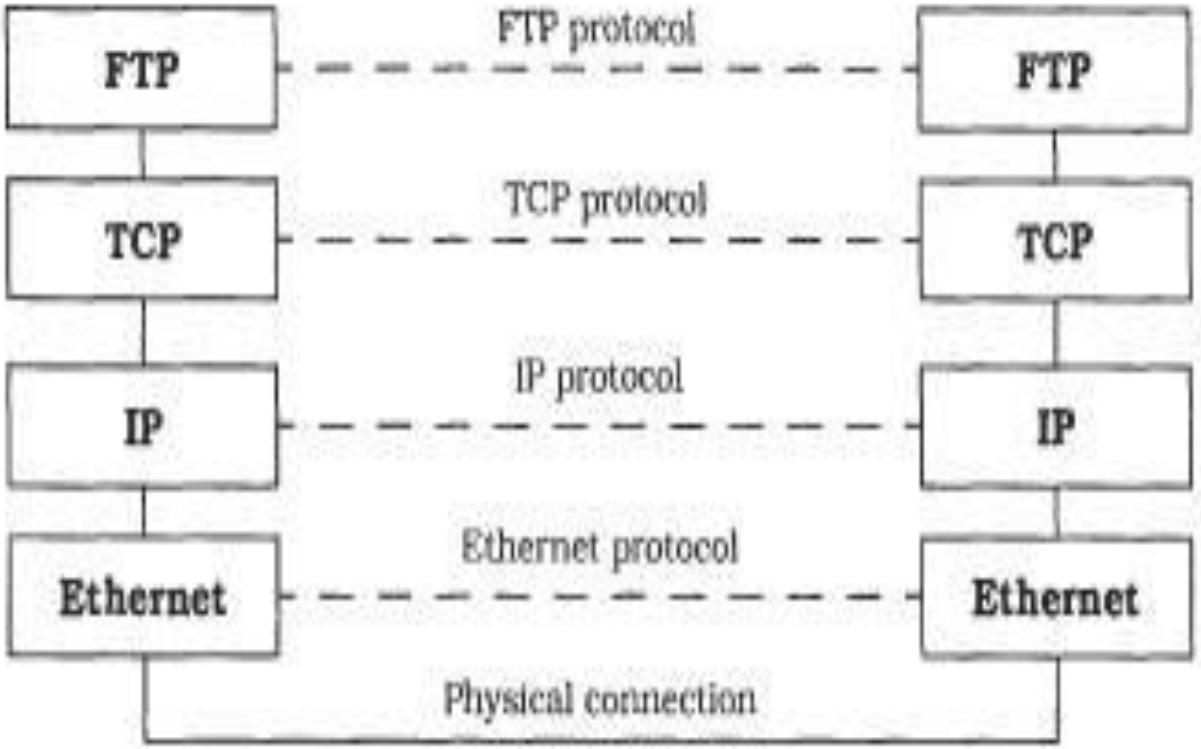
Layers模式

Implementation

1. Define the abstraction criterion for grouping tasks into layers (定原则)
2. Determine the number of abstraction levels according to your abstraction criterion (定分层)
3. Name the layers and assign tasks to each of them (分任务)
4. Specify the services (定服务)
5. Refine the layering (Iterate over steps 1 to 4) (迭代优化)
6. Specify an interface for each layer (定接口)
7. Structure individual layers (定层内结构)
8. Specify the communication between adjacent layers (定通信方式)
9. Decouple adjacent layers (层间解耦)
10. Design an error-handling strategy (定错误处理策略)

Layers模式

- Example resolved



Layers模式

- Variants
  - *Relaxed Layered System*: the gain of flexibility and performance is paid for by a loss of maintainability.
  - *Layering Through Inheritance*: higher layers can modify lower-layer services according to their needs. But such an inheritance relationship closely ties the higher layer to the lower layer.

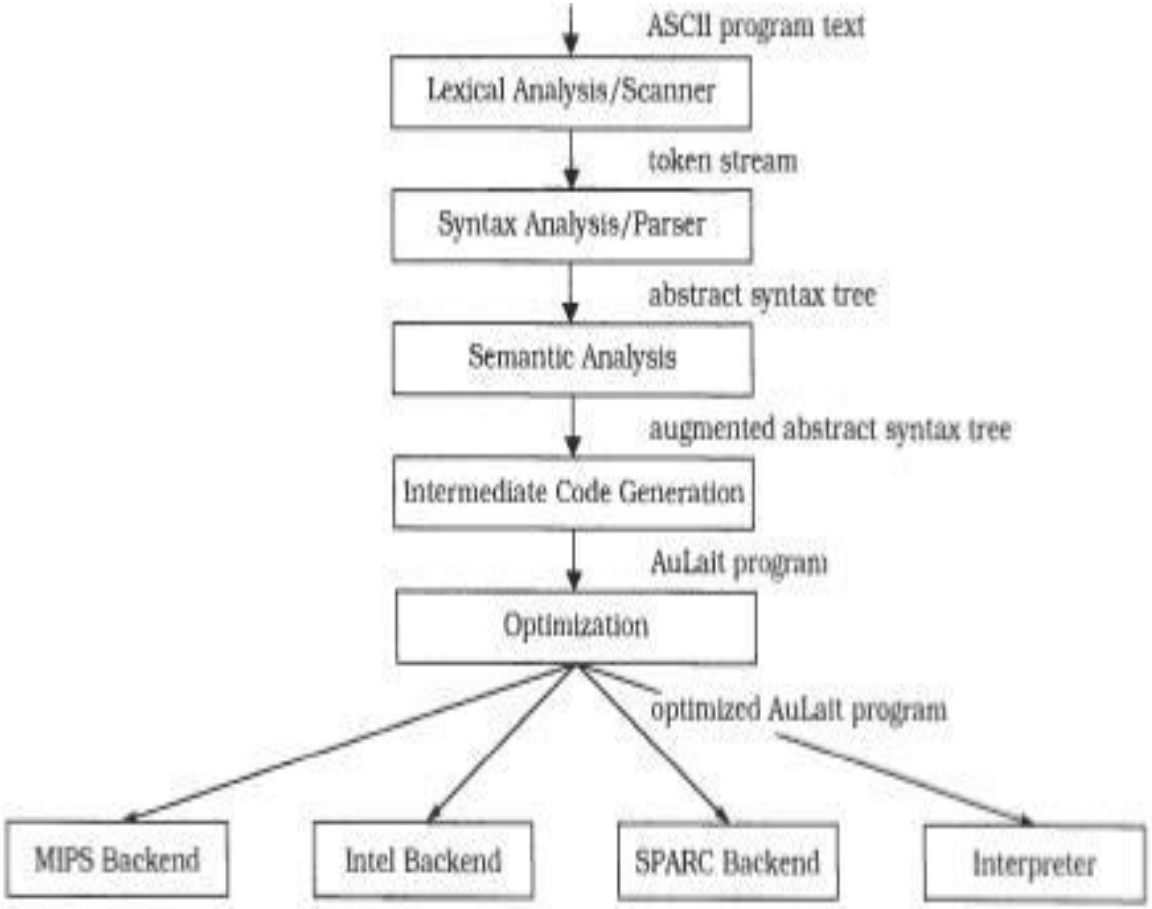
- Known Uses
  - Virtual Machines
  - APIs
  - Information Systems（C/S，三层架构）
  - Windows NT（微内核）

Layers模式

- 优点
  - 各层可重用
  - 支持标准化
  - 限制了依赖关系的范围
  - 可更换性
- 不足
  - 底层行为变化可能引起雪崩效应
  - 降低性能
  - 底层可能完成了高层所不需要多余的工作
  - 难以确定正确的层次粒度（层次数影响复用度、复杂度等因素）
- 相关模式
  - Composite Message Pattern
  - Microkernel architecture
  - PAC architectural pattern

Pipes & Filters模式

- 示例：编译器



Pipes & Filters模式

- Context
  - Processing data streams （处理数据流）

- Problem
  - Future system enhancements should be possible by exchanging processing steps or by recombination of steps, even by users （支持处理流程更改）
  - Small processing steps are easier to reuse in different contexts （步骤重用）
  - Non-adjacent processing steps do not share information（状态不共享）
  - Different sources of input data exist, such as a network connection or a hardware sensor providing temperature readings, e.g..（数据源）
  - It should be possible to present or store final results in various ways（数据池）
  - Explicit storage of intermediate results for further processing in files clutters directories and is error-prone, if done by users（中间结果不显式存储）
  - You may not want to rule out multi-processing the steps, for example running them in parallel or quasi-parallel.（避免并行化）

Pipes & Filters模式

- Solution
  - The pipes and filters architectural pattern divides the task of a system into several sequential processing steps.
    - These steps are connected by the data flow through the system
    - Each processing step is implemented by a *filter* component
    - A filter consumes and delivers data incrementally to achieve low latency and enable real parallel processing（流水线，data stream）
    - The input is provided by a *data source* such as a text file
    - The output flows into a *data sink* such as a file, terminal, animation program and so on.
    - The data source, the filters and the data sink are connected sequentially by pipes
    - Each pipe implements the data flow between adjacent processing steps
    - The sequence of filters combined by pipes is called processing pipeline

Pipes & Filters模式

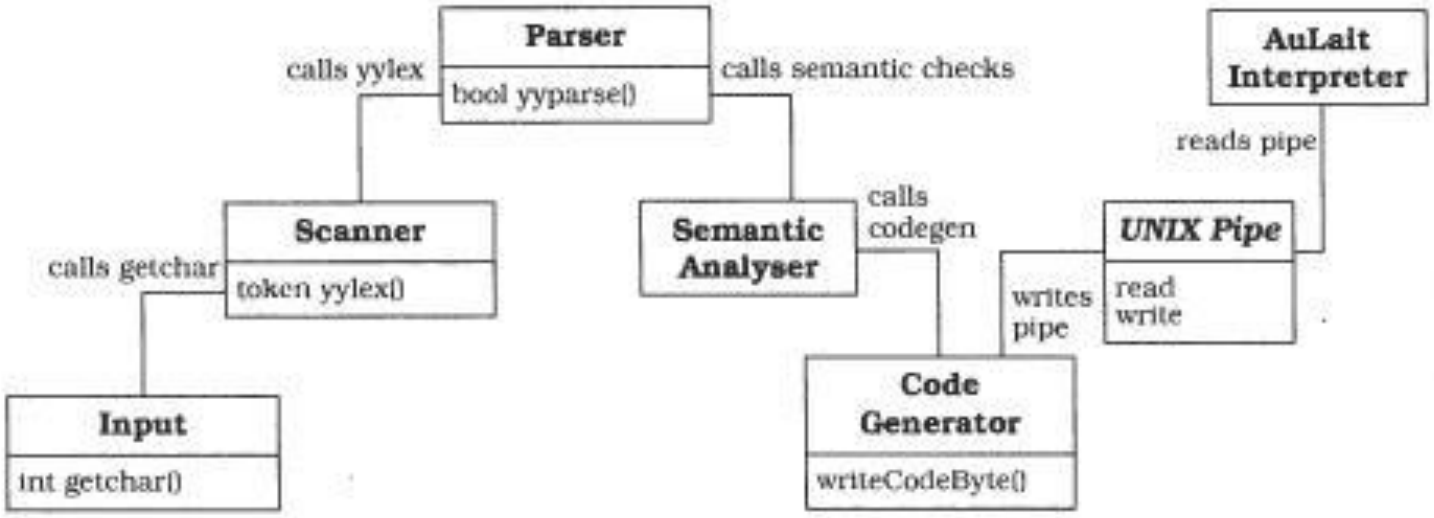
- Structure

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

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

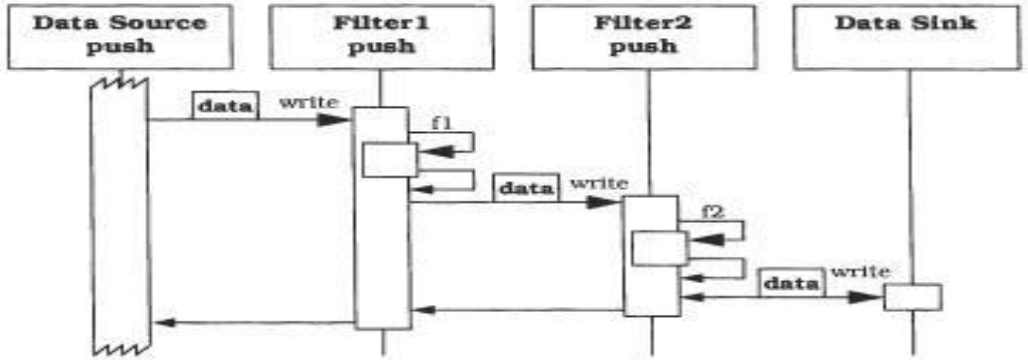
Class	Collaborators
Data Source	• Pipe
Responsibility	
• Delivers input to processing pipeline.	

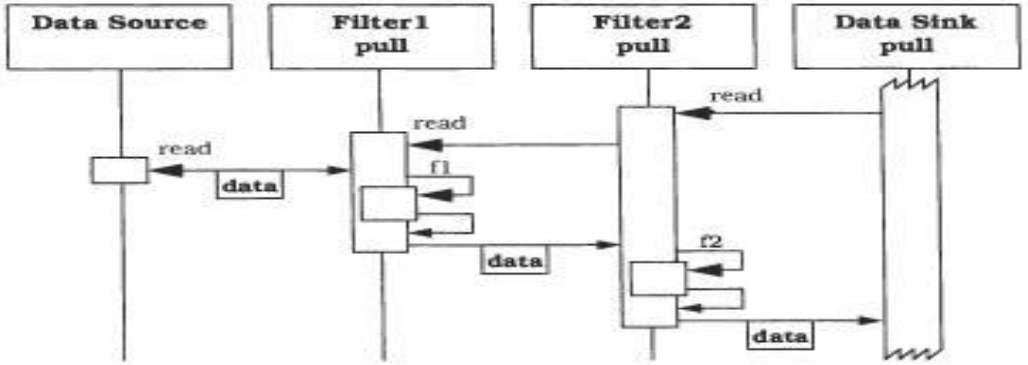
Class	Collaborators
Data Sink	• Pipe
Responsibility	
• Consumes output.	

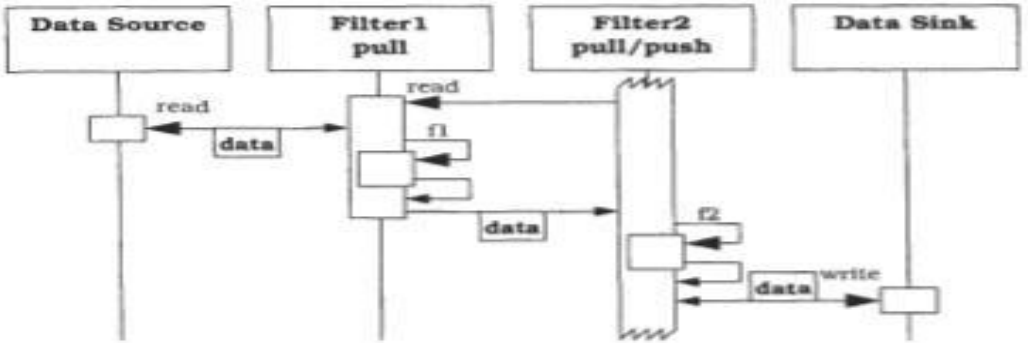


Pipes & Filters模式

- Dynamics
  - Passive filters
    1. Push pipeline
    2. Pull pipeline
    3. Push/pull pipeline

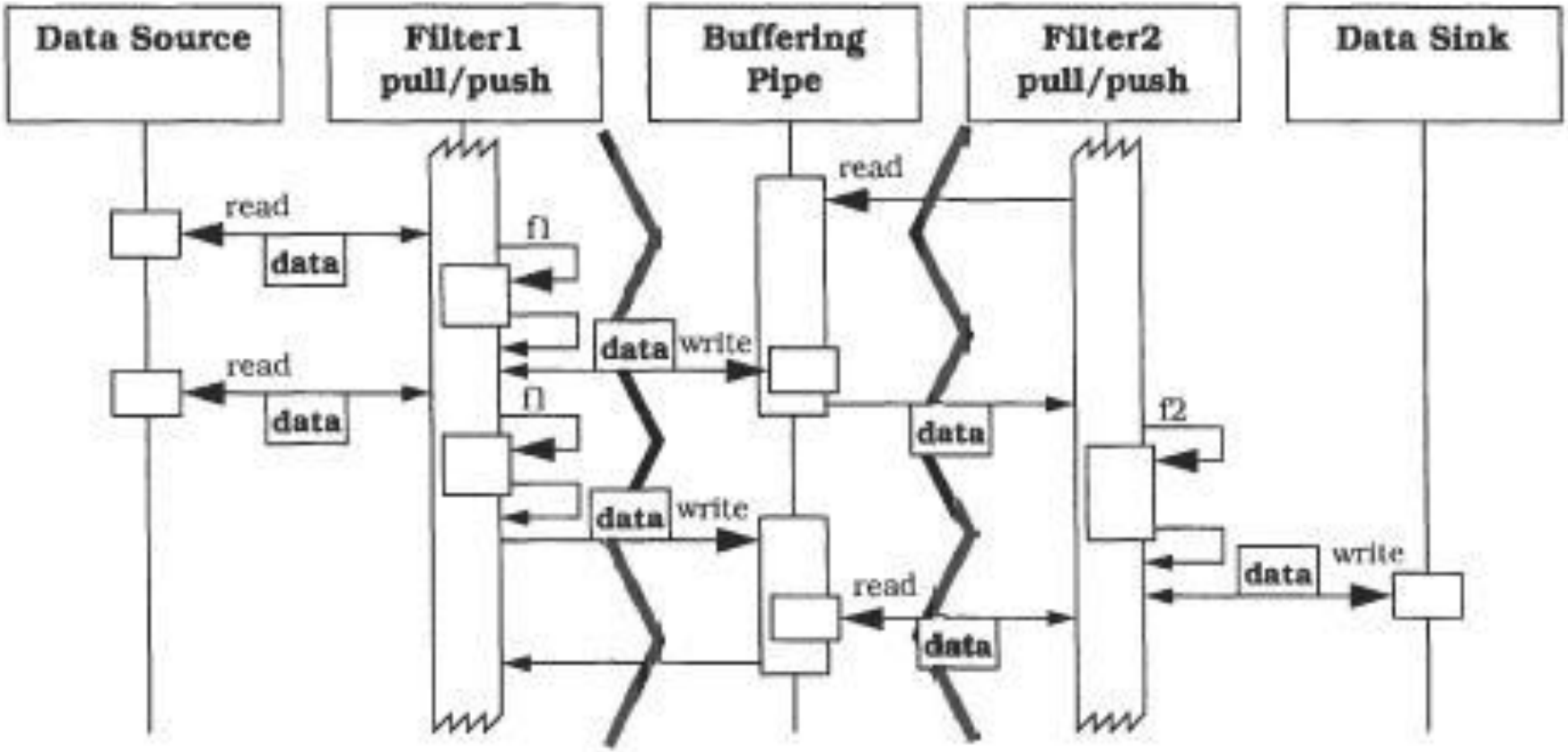






Pipes & Filters模式

- Dynamics
  - Active filters（主动过滤器）



Pipes & Filters模式

- Implementation
  - Divide the system's task into a sequence of processing stages;（将处理过程分段）
  - Define the data format to be passed along each pipe;（定义数据格式）
  - Decide how to implement each pipe connection;（定义管道连接）
  - Design and implement the filters;（实现过滤器）
  - Design the error handling;（实现错误处理）
  - Set up the processing pipeline;（搭建流水线）

Pipes & Filters模式

- Variants
  - Tee and join pipeline systems（过滤器有多个入口或出口）；
- Known Uses
  - UNIX pipes and filters;

Pipes & Filters模式

- 效果
  - 优点:
    - No intermediate files necessary, but possible;
    - Flexibility by filter exchange;
    - Flexibility by recombination;
    - Reuse of filter components;
    - Rapid prototyping of pipelines;
    - Efficiency by parallel processing;
  - 不足:
    - Sharing state information is expensive or inflexible;
    - Efficiency gain by parallel processing is often an illusion
      - The cost for transferring data between filters may be relatively high
      - Some filters consume all their input before producing any output
      - Context switching between threads or processes
      - Synchronization of filters via pipes may stop and start filters often
    - Data transformation overhead（统一管道传输类型带来数据传输开销大）
    - Error Handling

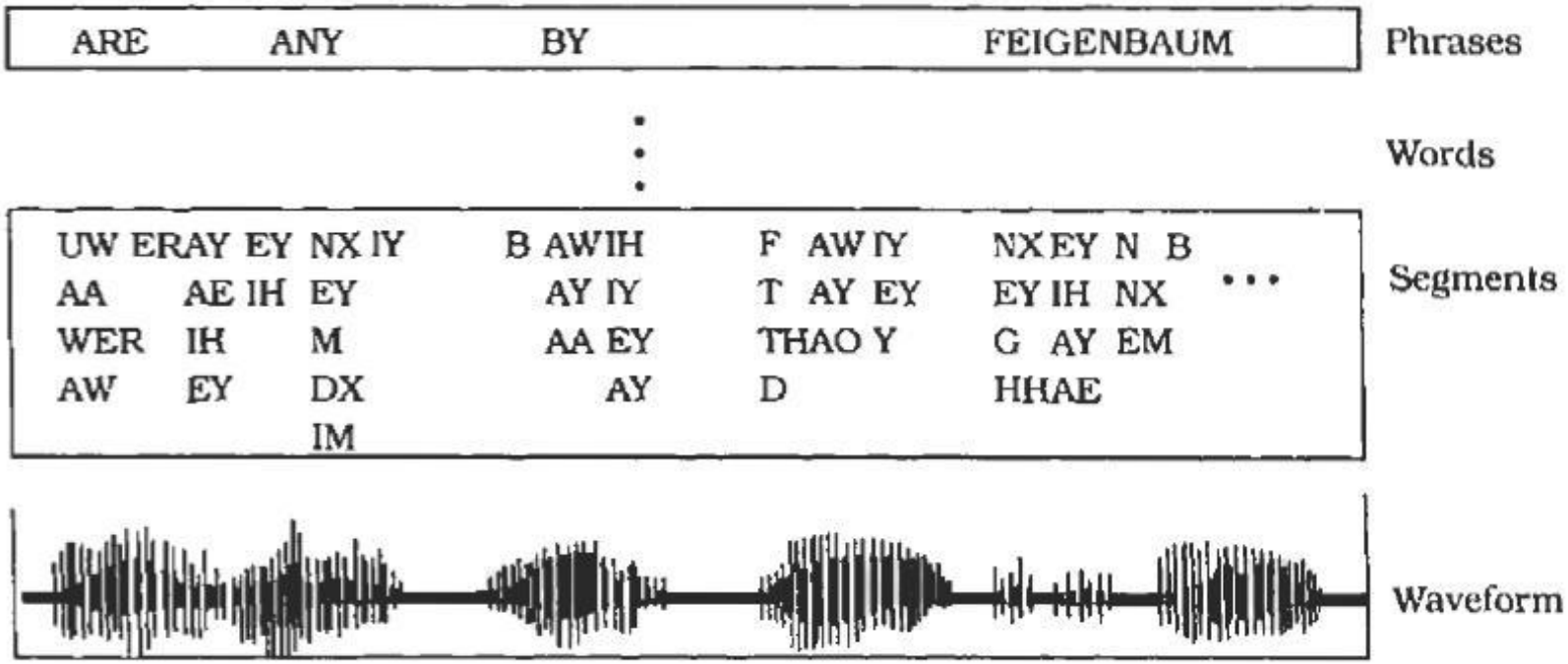
Pipes & Filters模式

- See also
  - The Layers pattern is better suited to systems that require reliable operation, because it is easier to implement error handling than with Pipes and Filters.

黑板Blackboard模式

- 示例：

  语音识别软件，输入声音波形数据，输出是识别出来的短语



黑板Blackboard模式

- 背景: 一个不成熟领域，该领域中没有已知或可行的解决方法

- 问题:
  - 不可能在合理的时间内遍历整个解空间。
  - 鉴于领域不成熟，可能需要对某一个子任务尝试不同的算法。因此，模块应该易于替换。
  - 解决子问题的算法是不同和无关的。
  - 输入、中间结果和最终结果的表达形式不同，而算法也是依据不同的范型（paragigms）来实现的。
  - 一个算法通常使用其他算法的结果。
  - 包含非确定数据和近似解。
  - 使用不相关算法带来潜在的并行性，尽量避免有严格顺序性的解决方案。

黑板Blackboard模式

- 解决方案

使用“黑板”，是因为该方案的思路是模拟若干不同领域的专家聚集在一起，将输入数据写在黑板上，每位专家根据黑板上的数据和自身的领域知识独立判断自己可能对问题解的贡献，对黑板上的信息进行修改、删除和添加，供其他专家共享使用，当然对黑板信息的操作要互斥进行，每次只能由一位专家进行。重复以上过程，直到问题解决。

- 一组独立的程序针对一个公共数据结构协同工作，每个程序专门用来解决整个任务的某一特定部分，所有程序一起来完成问题求解。
- 这里的协同关系不是程序间的直接调用关系，也不是顺序执行关系，而是由一个中心控制组件根据当前的状态，来协调执行某个程序。协调的策略也是可以调整的。
- 解决问题的过程中，系统就是通过合并、修正或否决部分解来完成求解的。每一个这样的解都表示了一个部分问题和某一个阶段的解，所有可能的解被称为问题的解空间，并按不同的抽象层次来组织。

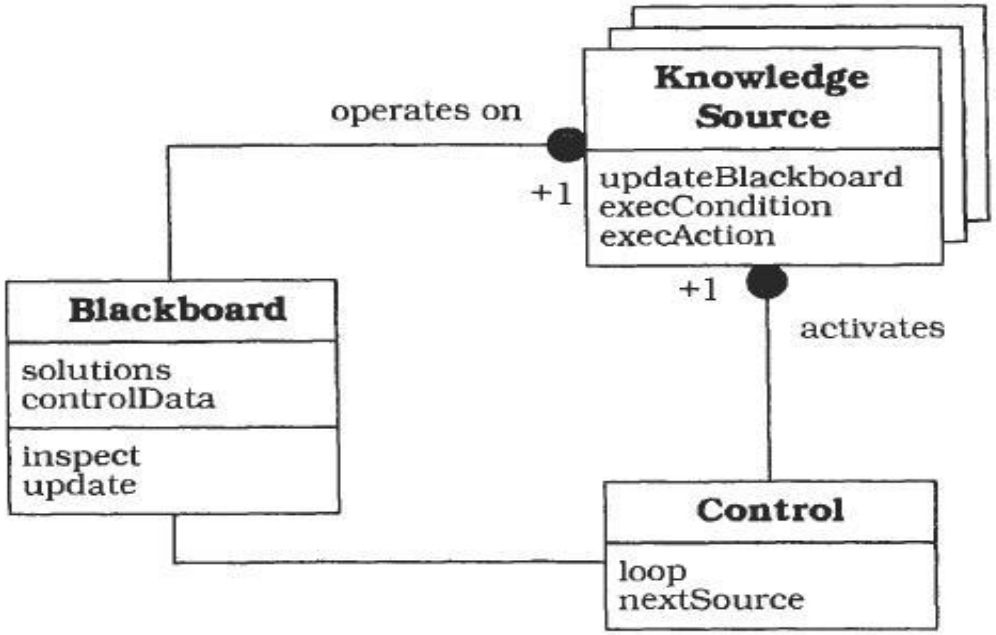
黑板Blackboard模式

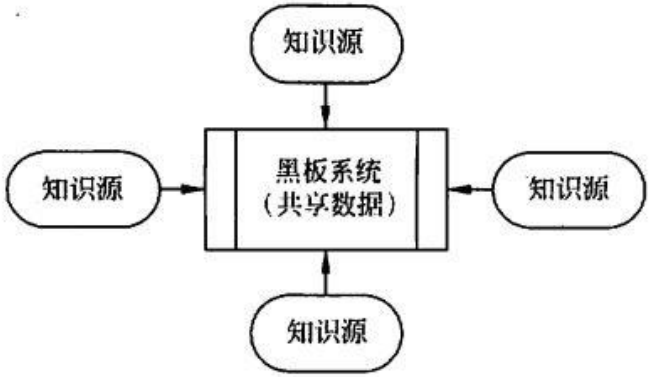
- 结构

系统包含了一个黑板组件blackboard, 一组知识源组件knowledge sources 和一个控制组件control component

- Blackboard：中央数据存储用于存放解空间元素和控制数据，并为knowledge sources组件提供读写接口。
- Knowledge sources：是一组解决系统特定方面问题的独立子程序，它们一起完成整个系统的建模，不能单独解决整个问题。它们不直接通信，而是通过对黑板读和写来间接通信。
- Control：是一个循环执行体，用来监控黑板的变化，并决定下一步要执行的动作。它根据知识应用策略（knowledge application Strategy）来调度知识源的评估和执行。

黑板Blackboard模式



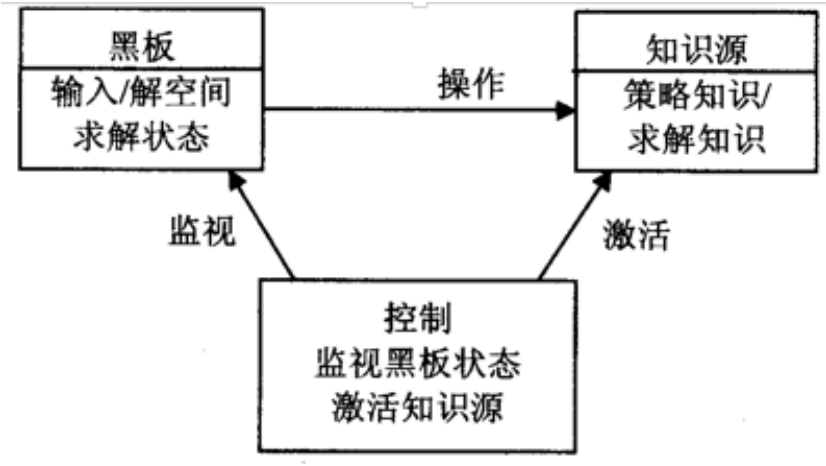


黑板Blackboard模式

Class Blackboard	Collaborators -
Responsibility • Manages central data	

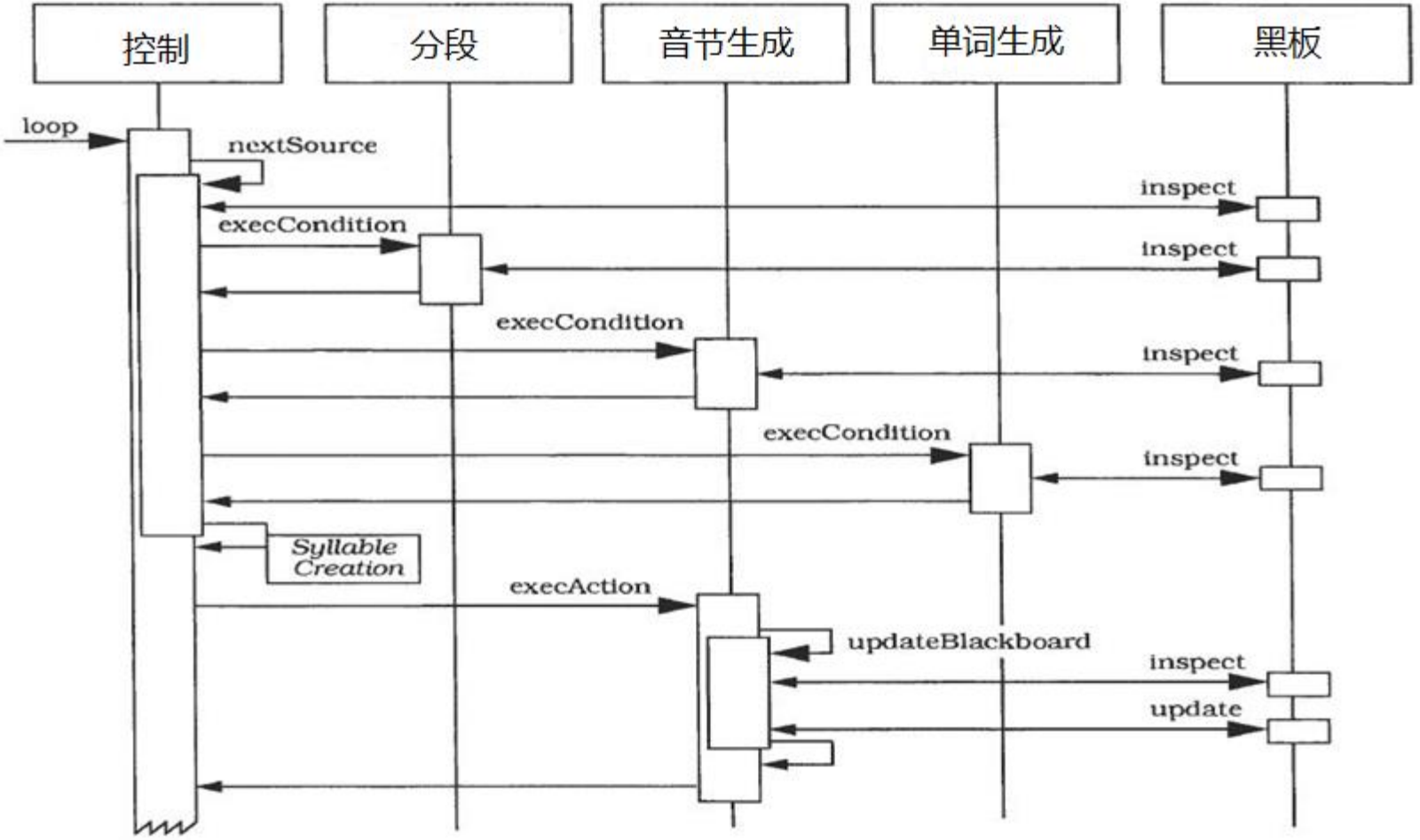
Class Knowledge Source	Collaborator • Blackboard
Responsibility • Evaluates its own applicability • Computes a result • Updates Blackboard	

Class Control	Collaborators • Blackboard • Knowledge Source
Responsibility • Monitors Blackboard • Schedules Knowledge Source activations	



黑板Blackboard模式

- 动态特性（以语音识别为例）



黑板Blackboard模式

- 实现
  1. 定义问题
     - 确定问题域以及解题所需的知识领域
     - 研究系统的输入，找出输入的特殊特征
     - 定义系统的输出
     - 详细描述用户如何与系统交互
  2. 定义问题的解空间
     - 将解分解为中间解和顶层解，顶级解为最高抽象层次，其他层级为中间解
     - 将解分解为部分解和完整解，完整解是整个问题的解
  3. 将求解过程分解为如下步骤
     - 定义解是如何转换成更高层级的
     - 描述在同一层级如何预测推测的（hypotheses）
     - 细化如何从其他层级找出证据来证实预测的推测
     - 指明可以排除部分解空间的知识类型

黑板Blackboard模式

4. 将知识划分为与子任务相关的专门知识源
5. 定义黑板的词汇表
6. 指定系统的控制器，控制器组件实现了机会主义的求解策略，以决定允许哪些知识源修改黑板。
7. 实现系统的知识源，根据控制器组件的要求，将知识源划分为条件部分和动作部分。知识源的实现是独立和不相关的。

黑板Blackboard模式

- 变种
  - 产生式系统[OPS language]（production system，rule-based system）
  - 仓库（Repository）
- 已知应用
  - HEARSAY-II
  - HASP / SIAP
  - CRYSALIS
  - TRICERO
  - Generalizations
  - SUS

黑板Blackboard模式

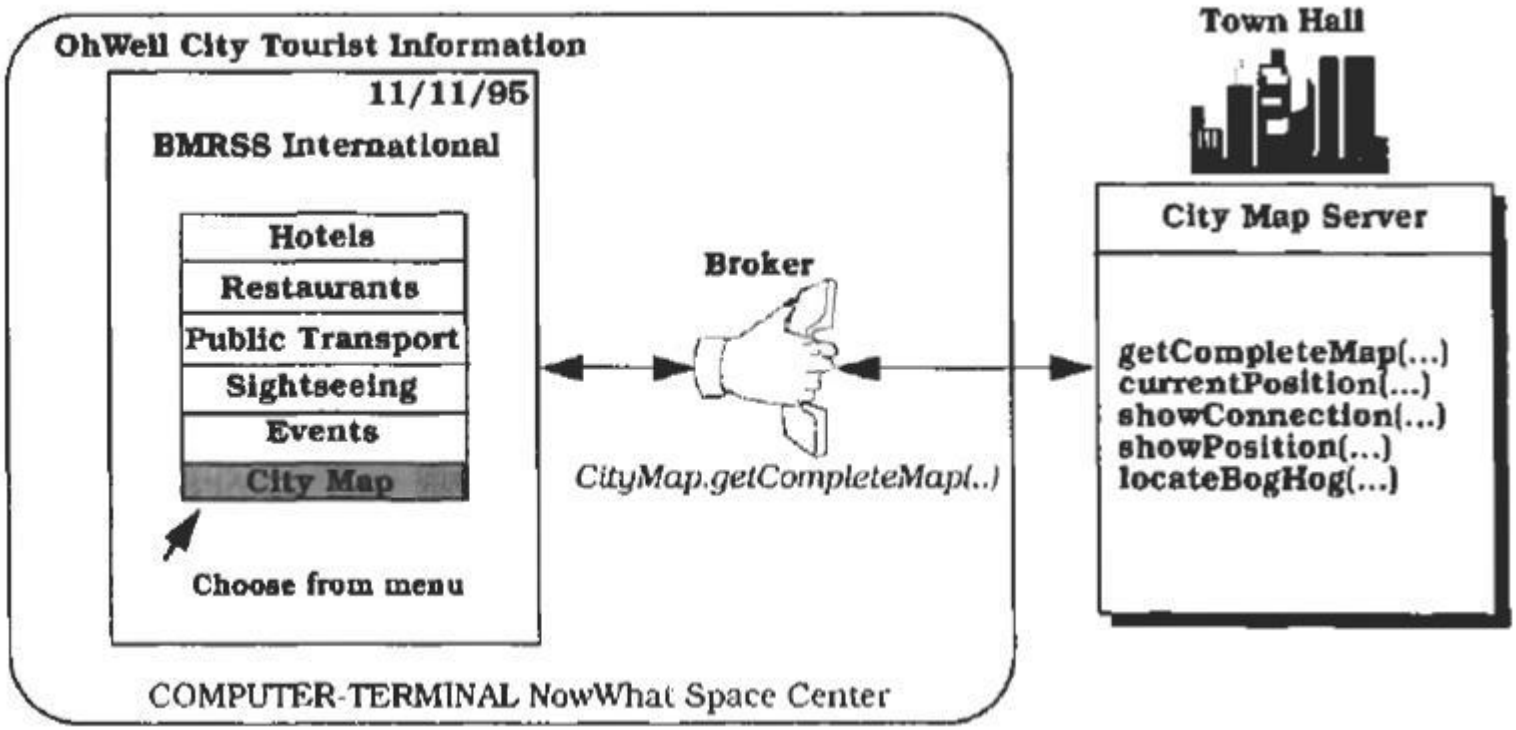
- 效果
  - 优点
    - 可以试验，试验不同的知识源（解题算法）和不同的控制策略
    - 耦合度低，提高了系统的可修改性和可维护性
    - 可重用知识源
    - 可提高系统的容错能力和鲁棒性，推错了问题也不大，只是可能影响效率
  - 不足
    - 不确定算法，难以测试
    - 不能保证提供好的解
    - 制定一个好的控制策略是困难的
    - 效率较低
    - 开发工作量大
    - 不支持并行性

“分布式系统”模式

- 分布式应用已成为主流系统形式，POSA1中与其相关的模式有三种
  - Pipes and Filters（参见“从混沌到结构”模式）
    用来解决数据流处理的问题
  - Microkernel（参见“适应性系统”模式）
    用来构建必须适应不断变化的系统
  - Broker
    用以实现远程服务调用的系统

代理Broker模式

- 示例（城市信息系统）



代理Broker模式

- 背景

  系统是由分布、甚至是异构的相互协作的组件构成的

- 问题

  通过将一个复杂的软件系统建成由一组分离且互操作的组件组成的软件，而非单个庞大的应用，从而获得更大的灵活性、可维护性和可修改性。通过将系统功能划分到各独立组件中，系统有望变为分布和可扩展的。

  - 组件可以通过位置透明的远程服务调用访问其他组件提供的服务
  - 可以在运行时更换、添加或删除组件
  - 该系统结构应向组件或服务的用户隐藏系统和实现有关的细节

代理Broker模式

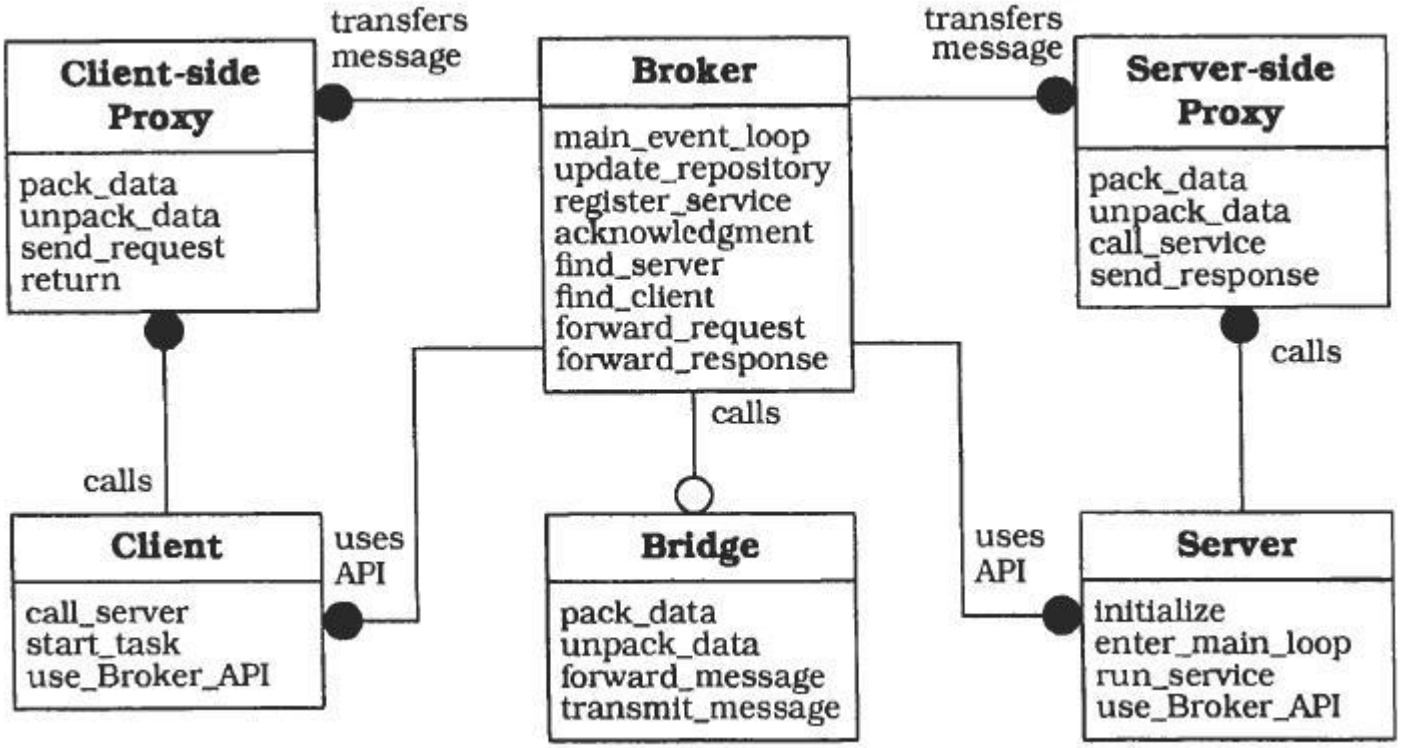
- 解决方案

引入一个代理（或称中介）组件（broker），降低客户端和服务器之间的耦合度。服务器向代理注册自己，使用方法接口向客户端提供服务；客户端向代理发出请求来访问服务器的功能；代理的任务就是定位合适的服务器，将请求发给该服务器，并将获得的结果或者异常传回给客户端。

代理Broker模式

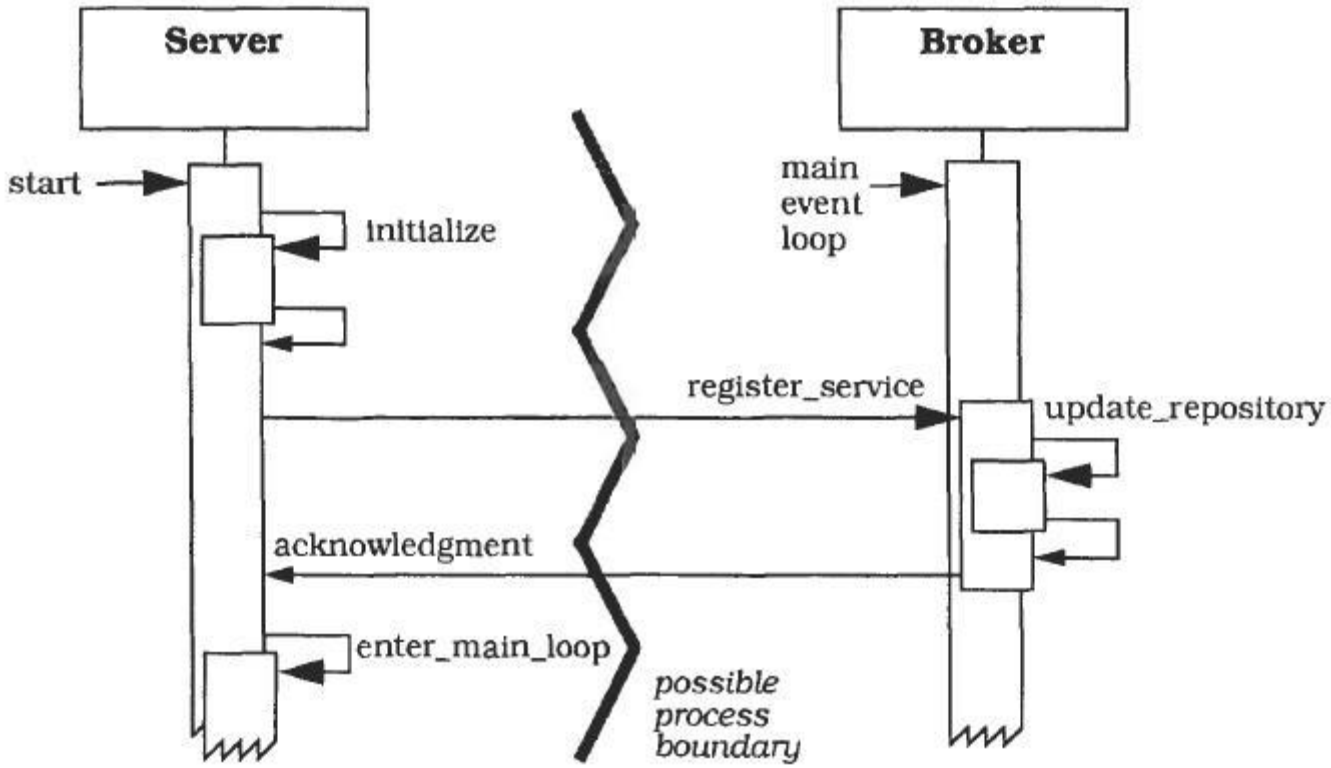
- 结构

Broker模式包含了6类组件：客户端client、客户端代理client-side proxy、代理（中介）broker、网桥bridge（可选组件）、服务器代理server-side proxy、服务器server



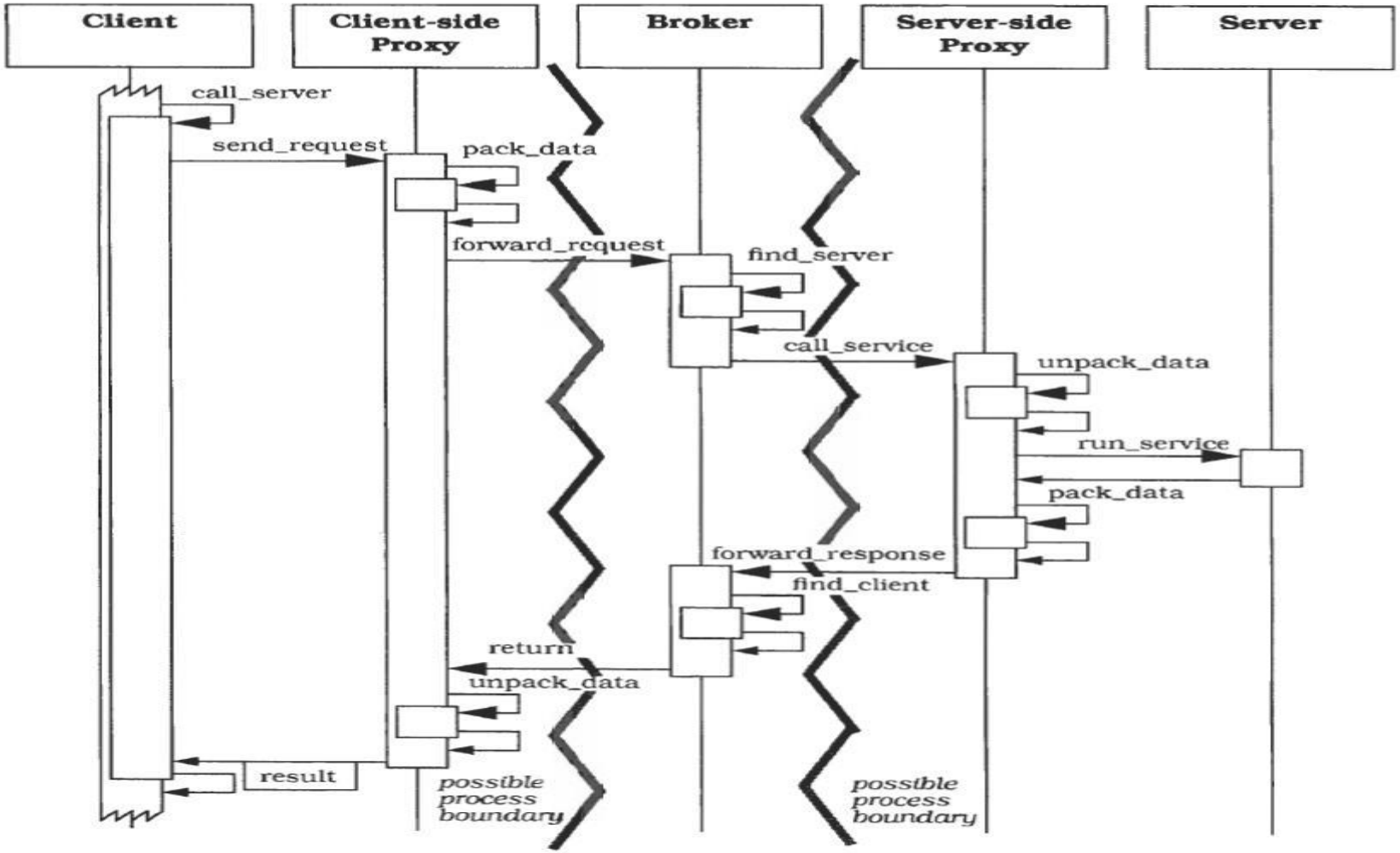
代理Broker模式

- 动态特性
  1. 服务注册



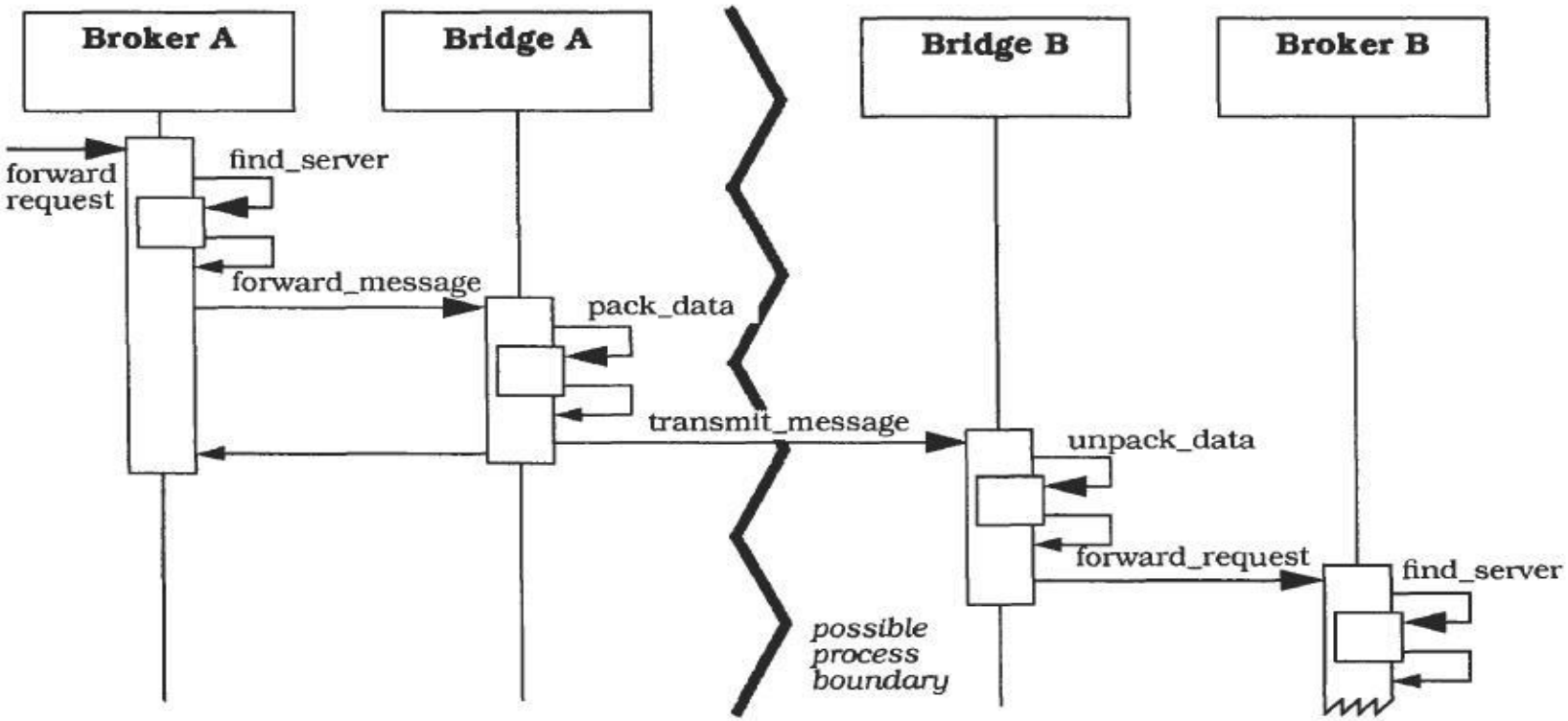
代理Broker模式

2. 客户端向服务器发送请求（同步或者异步）



代理Broker模式

3. 不同代理间通过网桥交互



代理Broker模式

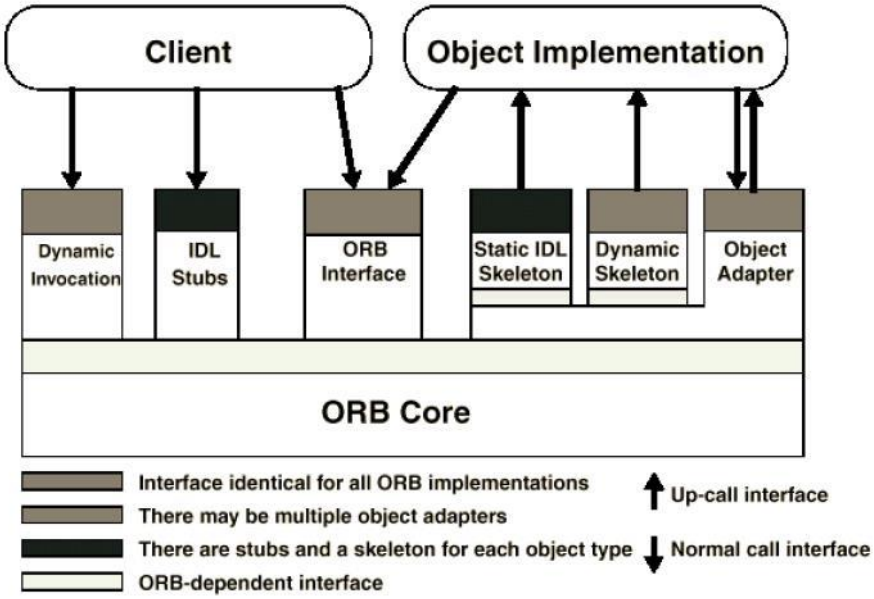
- 实现
  - 定义对象模型或使用现有模型model
  - 决定系统提供何种组件交互类型
  - 指定broker组件向客户端和服务器提供的API
  - 使用proxy对象对客户端和服务器隐藏实现细节
  - 设计broker组件
    - 给出client-side proxies和server-side proxies之间交互协议的细节
    - 对于网络中每一台参与的机器都必须提供一个本地broker供其使用
    - 客户端调用服务器的方法时，broker将所有结果和异常返回给客户端
    - 如果proxies没有提供调用参数和结果的包封marshaling和解封unmarshaling机制，那么broker就要提供这个功能
    - Broker应该包括一个目录服务，用来将服务器标识与服务器地址关联起来
    - Broker可能需要提供名字服务，为服务器生成唯一标识
    - 如果支持动态方法调用，则需要broker根据记住的服务器类型信息，动态生成调用
    - 实现相应的故障处理机制
  - 实现IDL编译器

代理Broker模式

- 变种
  - 直接通信Direct Communication Broker System
  - 消息发送Message Passing Broker System
  - 交易Trader System
  - Adapter Broker System
  - Callback Broker System
- 已知应用
  - CORBA
  - IBM SOM/DSOM
  - Microsoft’s OLE 2.x
  - WWW
  - ATM-P

代理Broker模式

- 效果
  - 优点
    - 位置透明
    - 组件可替换和可扩展
    - 可移植性好
    - 不同Broker系统之间的互操作
    - 可重用
  - 不足
    - 受限的效率，效率不高
    - 容错性差（broker可能成为故障点，导致所有依赖他的客户端、服务器都失效）
    - 测试和调试困难
- 相关模式
  - Forwarder-Receiver pattern
  - Proxy pattern
  - Client-Dispatcher-Server pattern
  - Mediator design pattern



CORBA：Common Object Request Broker Architecture

一个完整的典型请求过程

客户方	服务器方
1. 客户获取对象引用	
2. 向Stub发出请求	
3. Stub调用ORB接口，定位服务器	1.服务器方接收消息
4. ORB对请求进行编排	2.ORB进行反编排
5. 客户方ORB发送消息，等待服务器处理请求	3.OA定位（激活）实现
6.ORB接收结果	4.定位（激活）对象
7.客户处理结果	5.调用方法
	6.返回结果

“交互式系统”模式

图形用户界面的广泛使用，使得现有系统存在着大量的人机交互。构建具有人机交互特征的软件系统的挑战在于要确保功能核心独立于用户界面。这类系统的功能性需求通常比较稳定，而用户界面则经常需要修改和调整，往往还需要适应不同的设备和交互方式。

“交互式系统”模式希望构建一种架构，既允许修改用户界面，又要使得这种修改对底层的应用程序功能或数据模型带来的影响尽可能小。

典型模式：

- Model-View-Controller（MVC）

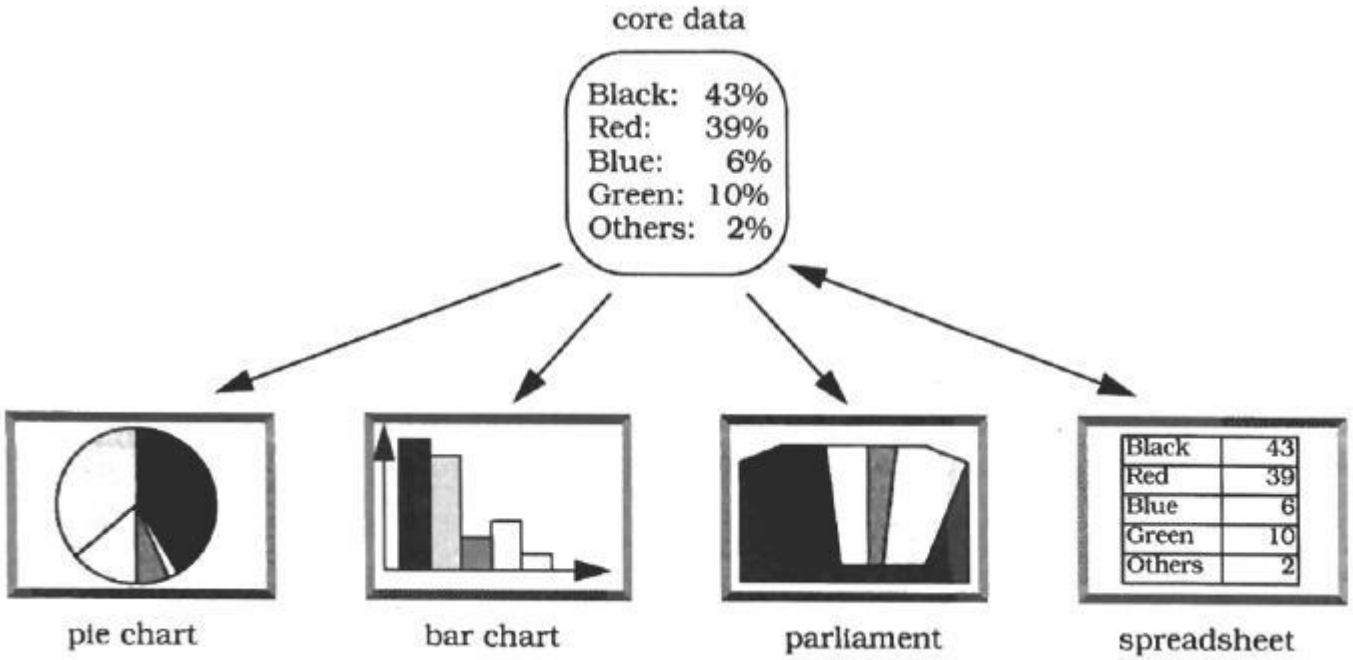
  最著名的一种交互式软件系统结构

- Presentation-Abstraction-Control（PAC）

  适用于由多个独立子系统组成的系统

MVC模式

- 示例


core data
Black: 43%
Red: 39%
Blue: 6%
Green: 10%
Others: 2%
Black 43
Red 39
Blue 6
Green 10
Others 2
pie chart
bar chart
parliament
spreadsheet

MVC模式

- 背景
  - 具有灵活人机界面的交互式应用

- 问题

用户界面变化频繁、不同用户对界面的要求可能相互矛盾、界面代码与功能核心耦合度高会导致开发和维护难度大大增加。考虑的因素有：

  - 同样的数据在不同的窗口有不同的表现形式，如饼图、棒图
  - 对数据的操作结果应该即时在应用的显示和行为上得到反映
  - 对用户界面的修改应该很容易，甚至可以在运行时修改
  - 支持不同的外观‘look and feel’标准，用户界面的一直不应影响到应用的核心代码

MVC模式

- 解决方案

MVC模式将一个交互应用分为三个方面：处理、输出、输入

- 处理: 模型（Model）组件封装了核心数据和功能，独立于特殊的输出表示和输入行为。
- 输出: 视图（View）组件从model组件获得数据，并向用户显示信息，每个view组件。一个模型可以对应有多个视图。
- 输入：每个view有一个相关联的控制器（Controller）组件负责接收输入，输入通常表现为鼠标和键盘动作的事件，而事件被转换成对模型或视图的服务请求，用户只通过控制器与系统交互。
- 通过某个控制器修改了模型，则对所有与该模型关联的视图都会产生影响。这种变更传播机制可以通过发布-订阅（Publisher-Subscriber）模式来实现。

MVC模式

结构

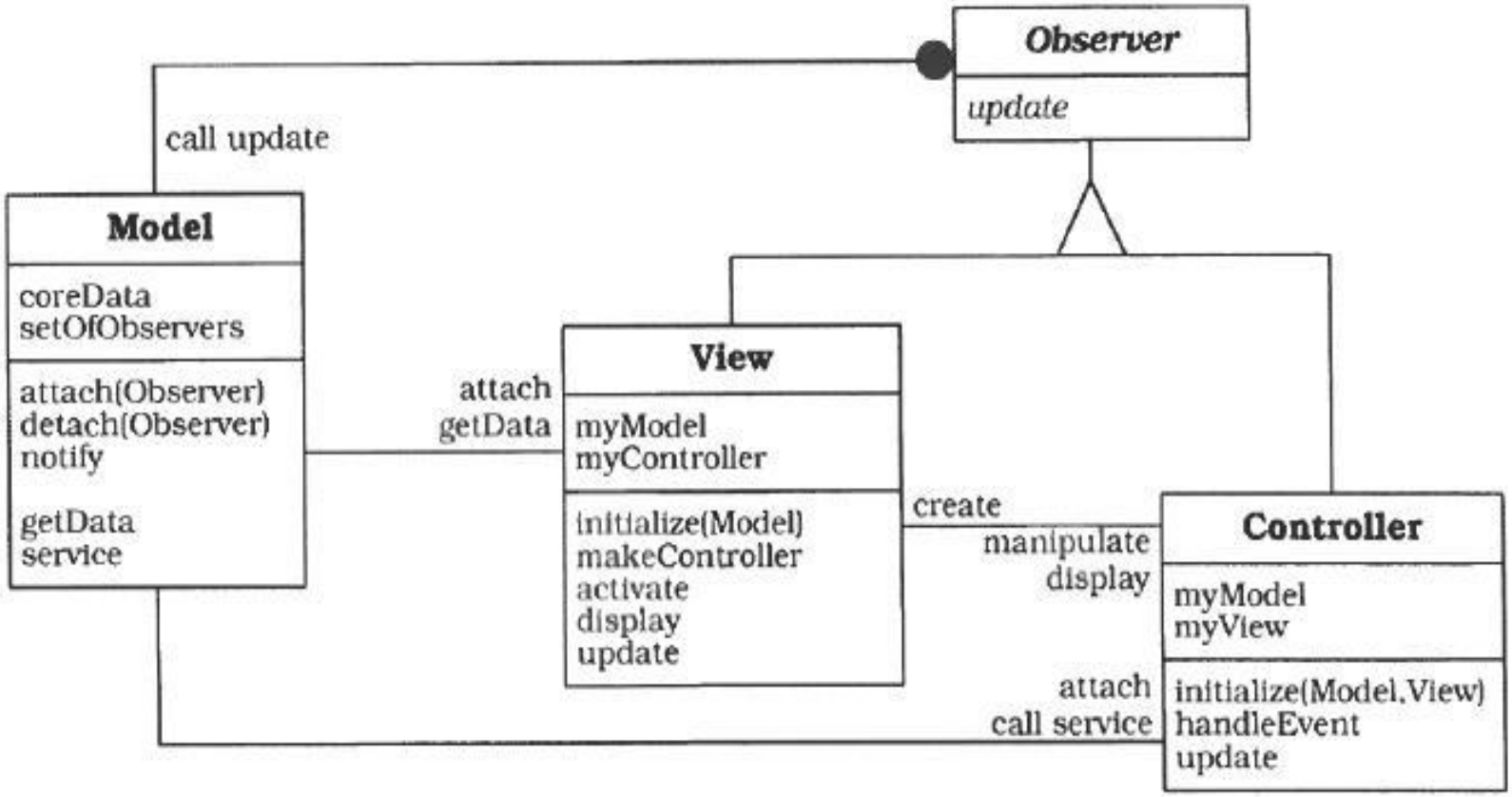
类 模型	协作者 • 视图 • 控制器
职责 • 提供应用程序的功能核心 • 注册相关的视图和控制器 • 将数据变更通知有关依赖的组件	

类 视图	协作者 • 模型 • 控制器
职责 • 创建并初始化相关联的控制器 • 向用户显示信息 • 实现更新过程 • 从模型处获得数据	

类 控制器	协作者 • 模型 • 视图
职责 • 接收用户输入事件 • 将事件转换成对模型的服务请求或者对视图的显示请求 • 根据需要实现更新过程	

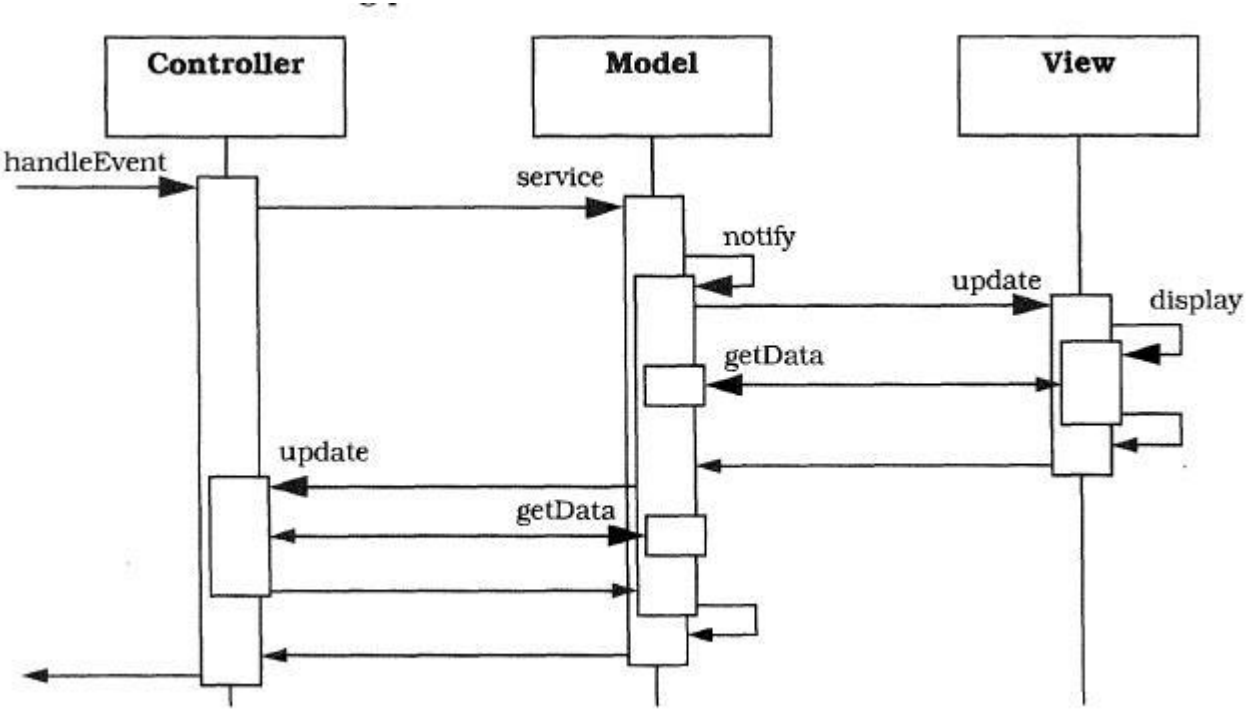
MVC模式

- 结构（C++实现，引入Observer为了实现变更传播）



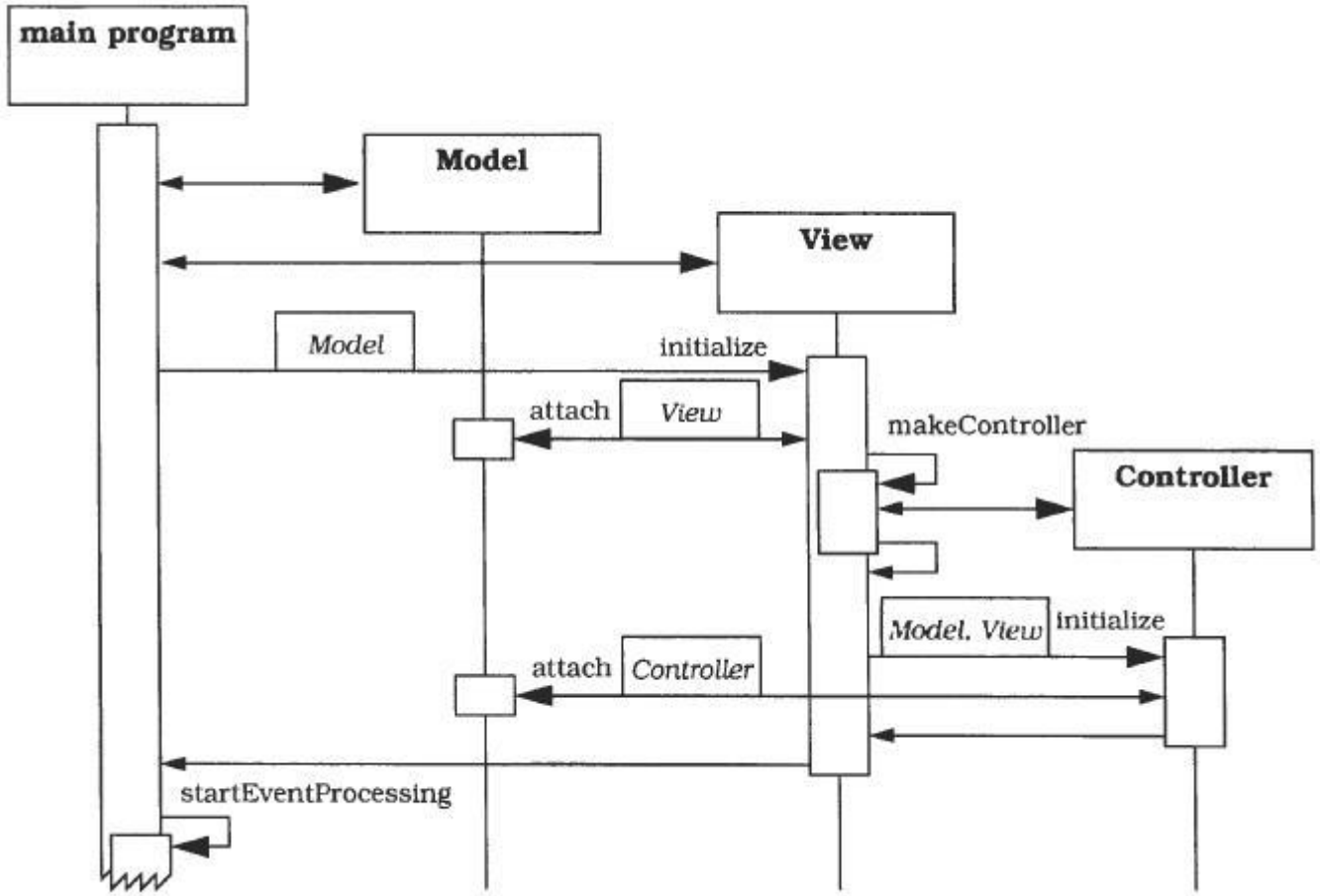
MVC模式

- 动态行为
  - 情景1: 用户输入导致模型发生变化，触发变更传播机制



MVC模式

- 情景2: 初始化MVC模式中的组件（通常在MVC组件之外）



MVC模式

- 实现
  1. 将人机交互与核心功能分离
  2. 实现变更传播机制
  3. 设计并实现视图组件
  4. 设计并实现控制器
  5. 设计并实现视图-控制器关系
  6. 实现MVC的初始化和启动代码
  7. 如果支持动态创建视图，此事需要提供一个视图管理组件
  8. 可以实现可插入‘Pluggable’控制器来动态变换视图关联的控制器
  9. 层次化视图和控制器的基础结构
  10. 进一步降低系统的依赖性

1-6步是实现MVC的基本步骤，7-10可选，通常用于构建一个MVC框架

MVC模式

- 变种
  - Document-View 没有将视图和控制器分离
- 已知应用
  - Smalltalk
  - MFC
  - ET++

MVC模式

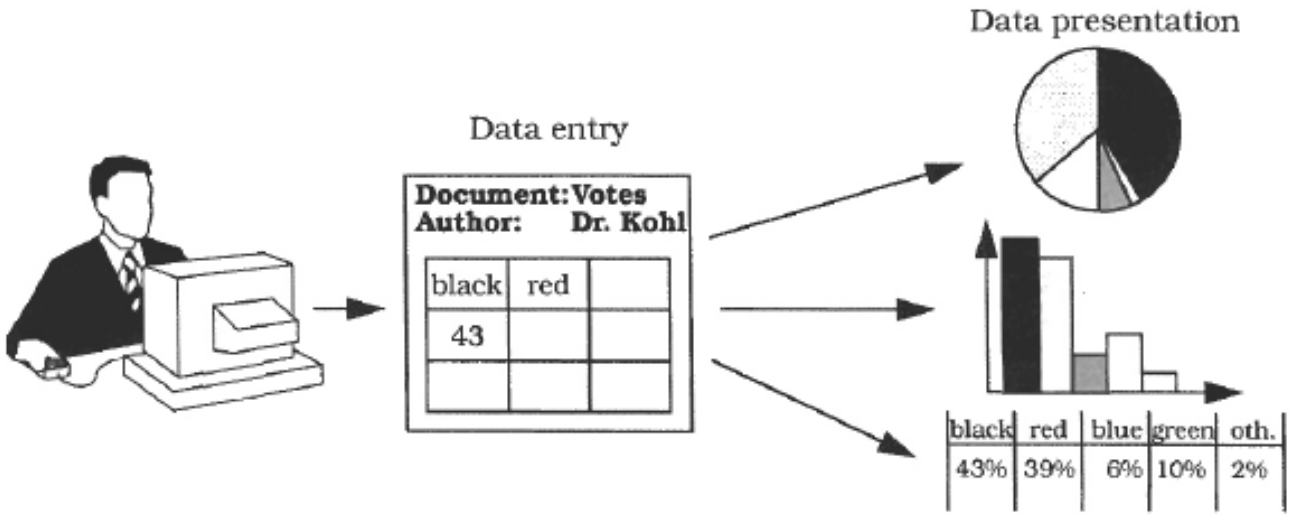
- 效果
  - 优点:
    - 同一模型多个视图
    - 视图间是同步的
    - 可插入的（Pluggable）视图和控制器
    - 外观可换
    - 可实现为框架
  - 不足：
    - 增加了复杂度
    - 更新可能过度频繁，并非每个视图都需要对所有变更做出响应，可通过忽略某些更新来解决
    - 视图和控制器联系紧密
    - 视图与控制器同模型紧耦合，模型接口修改将导致相应视图和控制器代码修改
    - 视图中数据访问的效率往往不高
    - 移植时需要修改视图和控制器
    - 将MVC与现代用户界面工具一起使用时会有困难
- 参见
  - The Presentation-Abstraction-Control pattern（PAC模式）

表示-抽象-控制PAC模式

➢示例（Presentation-Abstraction-Control模式）

这类系统是由相互协作的智能体组成的层次结构，每个智能体都负责应用的一些特定功能，包含有表示、抽象和控制三个组件，通过这种方式将智能体的人机交互同功能核心和通信相分离。

选举投票结果显示：



PAC模式

- 背景

  在智能体（agent）支持下开发交互式应用。

- 问题

  一大类交互系统可以由一组相互协作的智能体构成，有些用于处理人机交互，有些用于维护系统数据模型和提供相关功能服务，有些用于错误处理、通信等等。构建这类系统，考虑以下因素：

  - 智能体通常维护自身的状态和数据，为了与其他智能体协同工作，它需要一种交换数据、消息和事件的机制。
  - 由于智能体进行人机交互的方式有很大不同，交互智能体需要提供自己的用户界面
  - 系统发生演化，尤其是表示方面会经常发生变化，单个智能体的改变或者系统中加入新的智能体，不应该影响到整个系统。

PAC模式

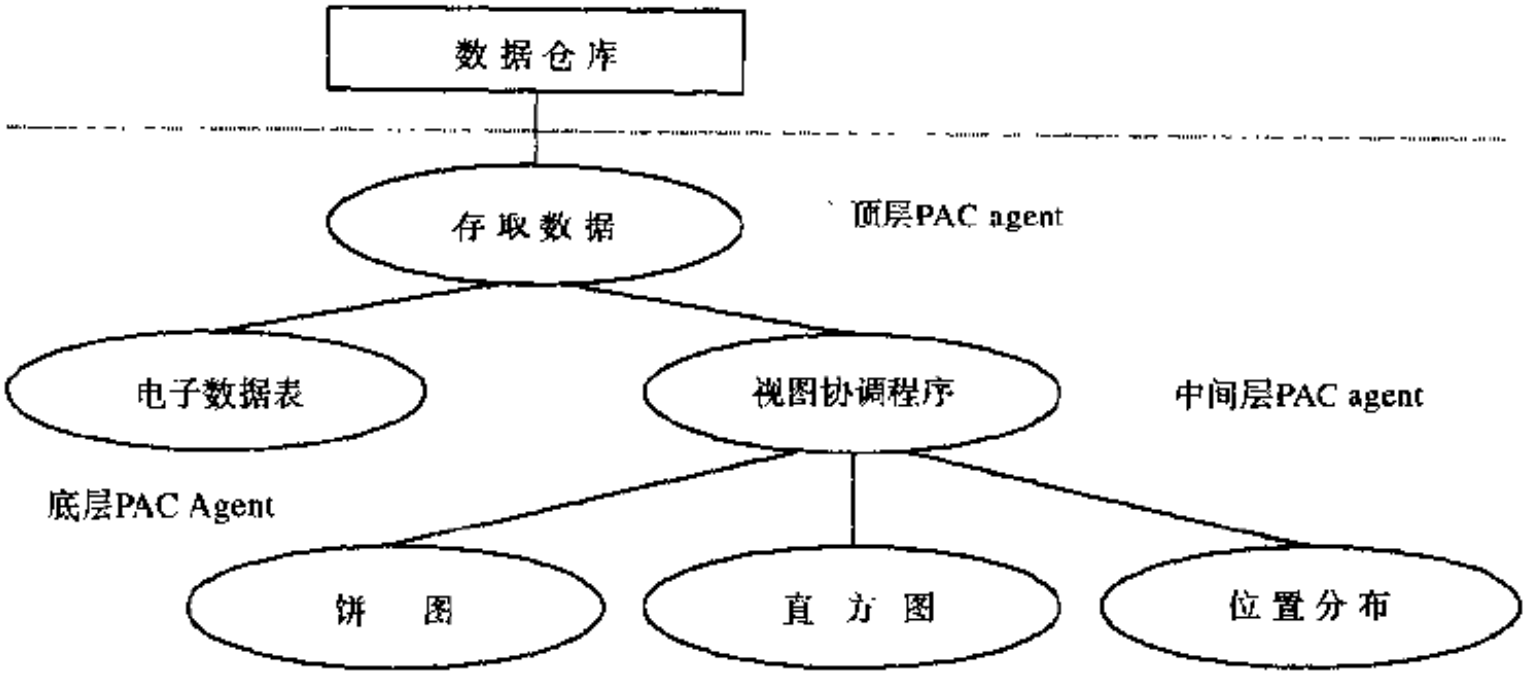
- 解决方案

  将交互式应用组织成由PAC智能体构成的树状层次结构

  - 有一个顶层智能体、若干中间层智能体和更多的底层智能体。每个智能体应用的某一方面功能，包含了表示、抽象和控制三个组件。整个层次反映了智能体之间的传递依赖关系，每个智能体依赖于其上一层到顶部的所有智能体。
  - 智能体的表示（presentation）组件提供智能体的可视行为；抽象（abstraction）组件维护其数据模型和提供操作这些数据的功能；控制（control）组件连接表示组件和抽象组件，提供与其他智能体通信的功能。
  - 顶层智能体提供了系统的功能核心，底层的智能体则代表了系统用户可操作的独立语义概念，中间层的智能体则是下一层智能体的复合或者相互关系。

PAC模式

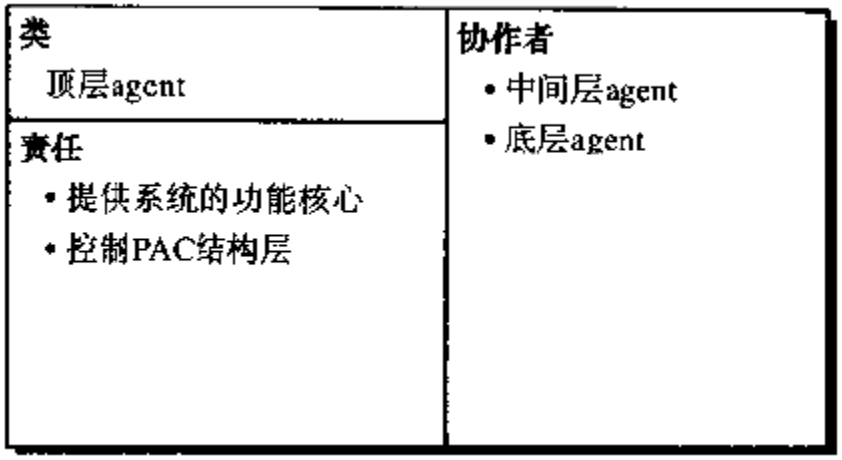
- 示例：选举系统的智能体层次结构图



PAC模式

➢结构

顶层智能体



抽象组件：提供应用的全局数据模型，以及操作数据模型和返回数据模型中的信息

表示组件：职责较为简单，通常是整个应用要用到的一些界面元素。有些应用的顶层智能体可以无此组件

控制组件：1.为下一层智能体提供对顶层智能体服务的访问，转发请求到抽象和表示组件

2.维护顶层与下一层智能体之间的关联信息，协调智能体层次结构

3.维护有关用户与系统交互的信息

PAC模式

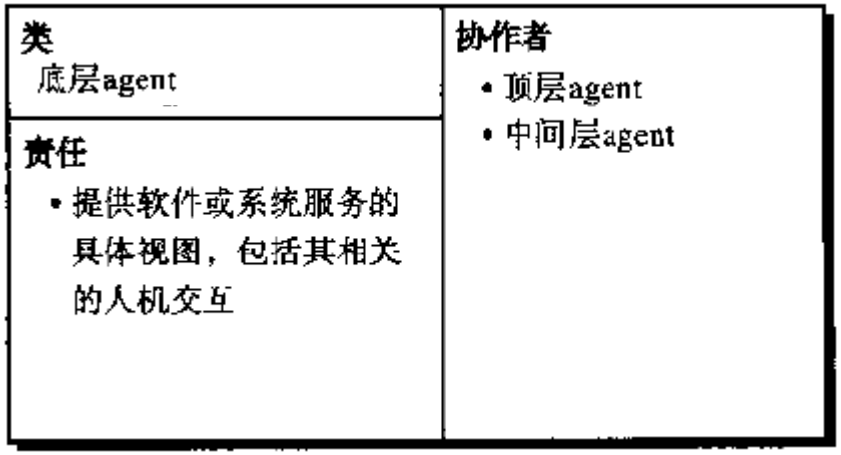
底层智能体

抽象组件：维护当前智能体的专用数据

表示组件：显示相应语义概念的特定视图，维护该视图的信息，并提供用户可对该视图操作的函数

控制组件：1.保证抽象组件和表示组件一致，避免彼此依赖

2.同上一层通信，以交换事件和数据



PAC模式

中间层智能体

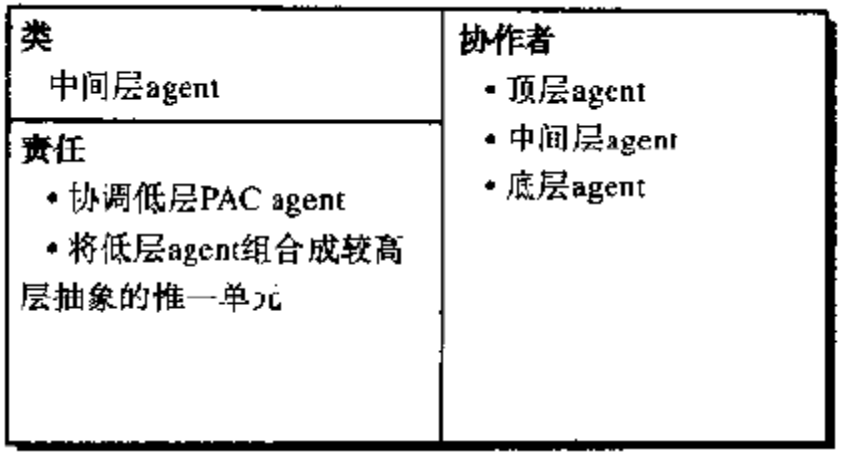
一是将底层智能体复合成复杂智能体

二是协调下层智能体保持一致性，如协调多个智能体视图

抽象组件：维护当前智能体的专用数据

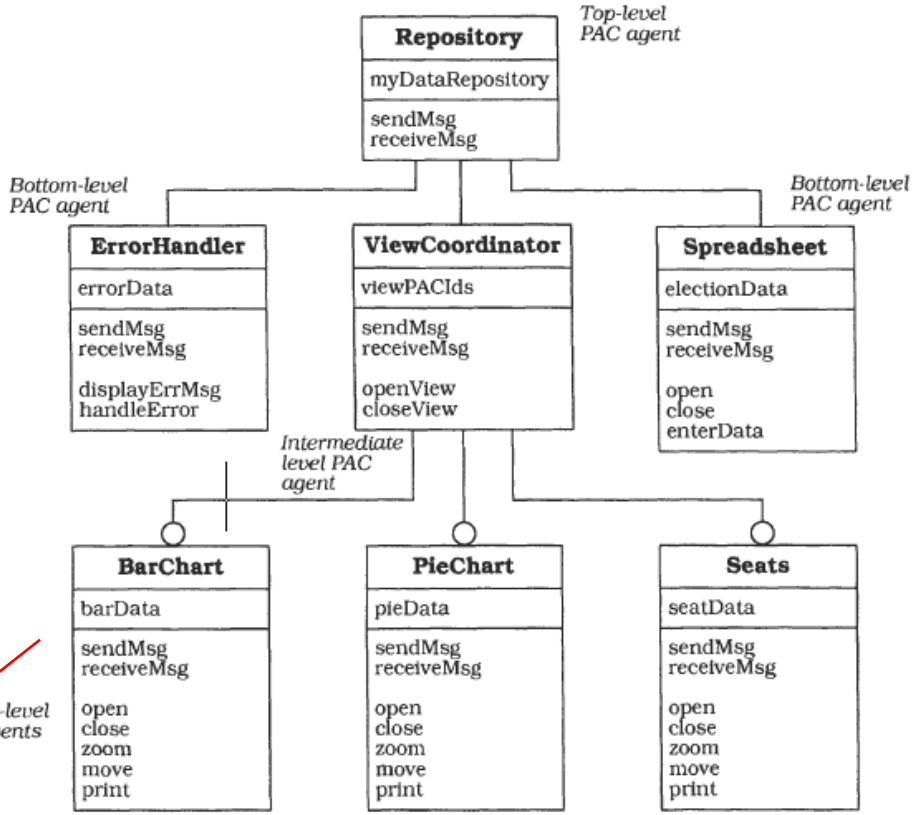
表示组件：实现该智能体的用户界面

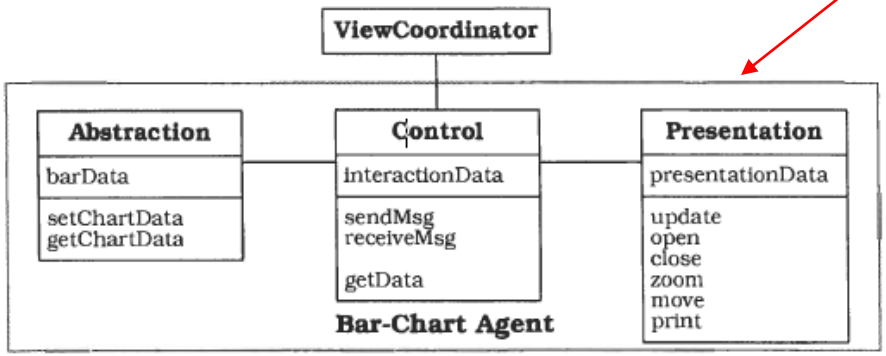
控制组件：与其他智能体相同



PAC模式

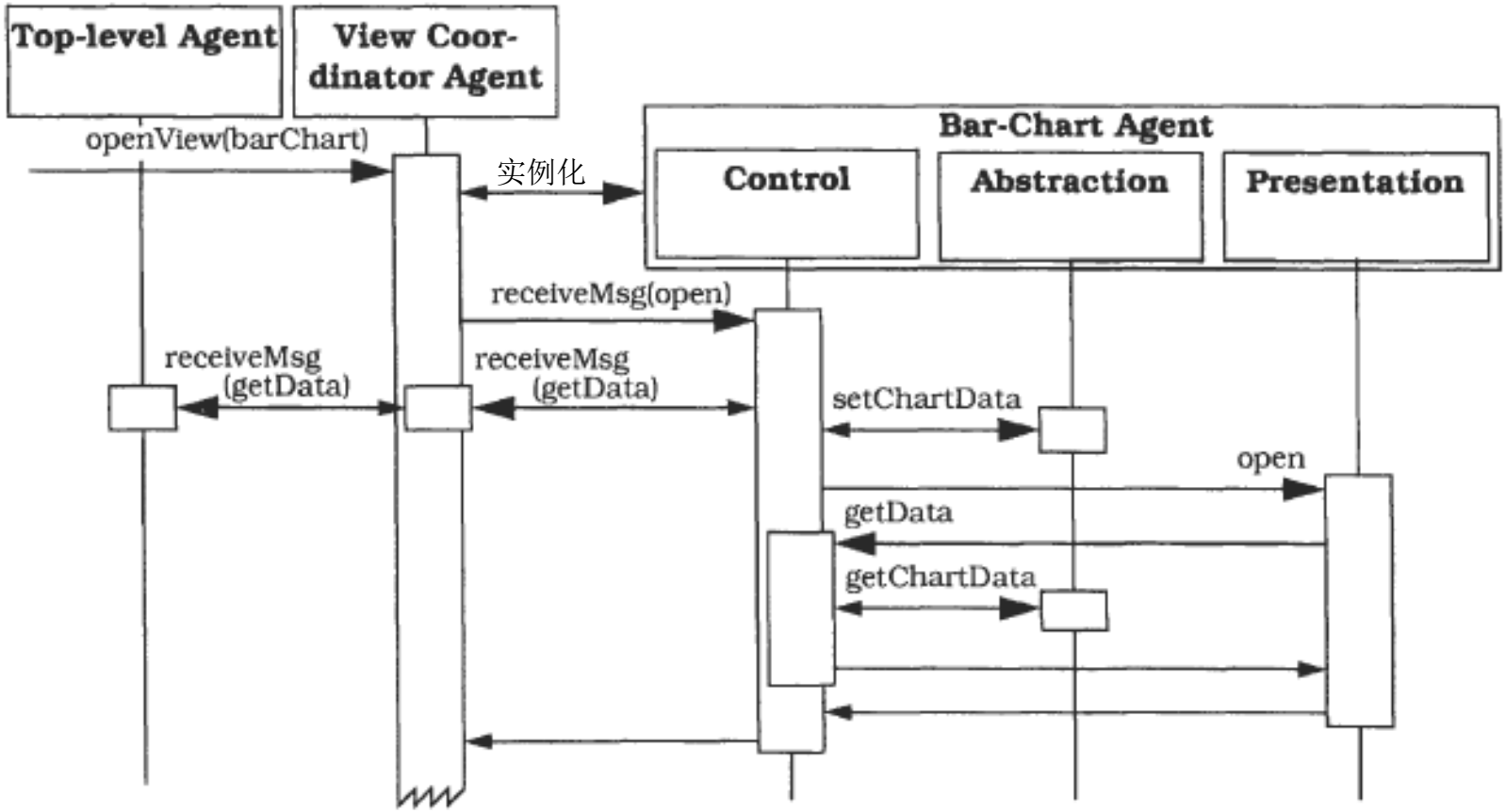
- 示例 选举系统





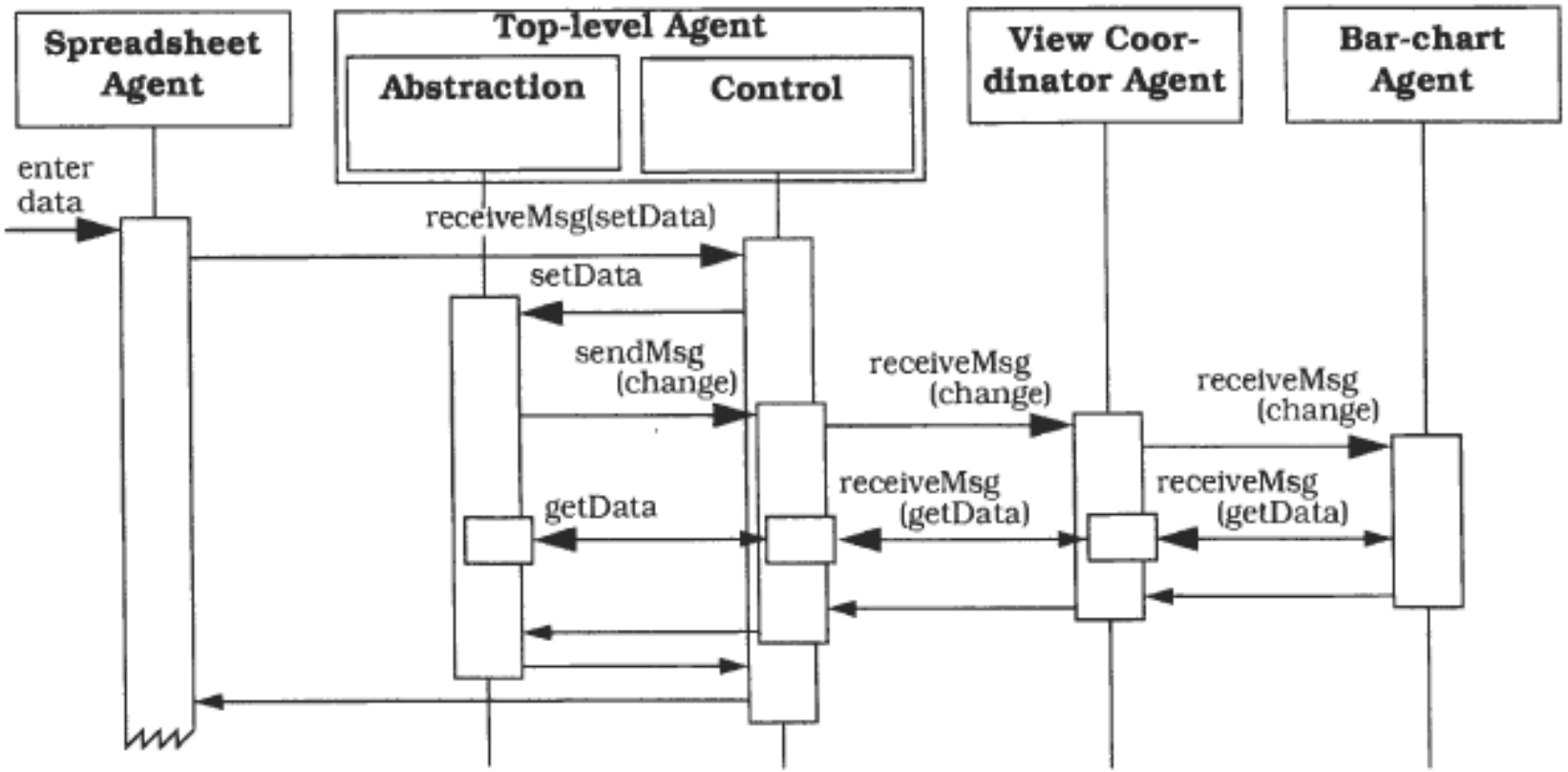
PAC模式

- 动态性（以选举系统为例）
  - 情景1: 描述了用户向“视图协调器”智能体的表示组件请求打开新的选举柱状图时，不同智能体之间的协作和内部行为。



PAC模式

- 情景2: 显示了输入新的选举数据后系统的行为，详细给出了顶层智能体的内部行为

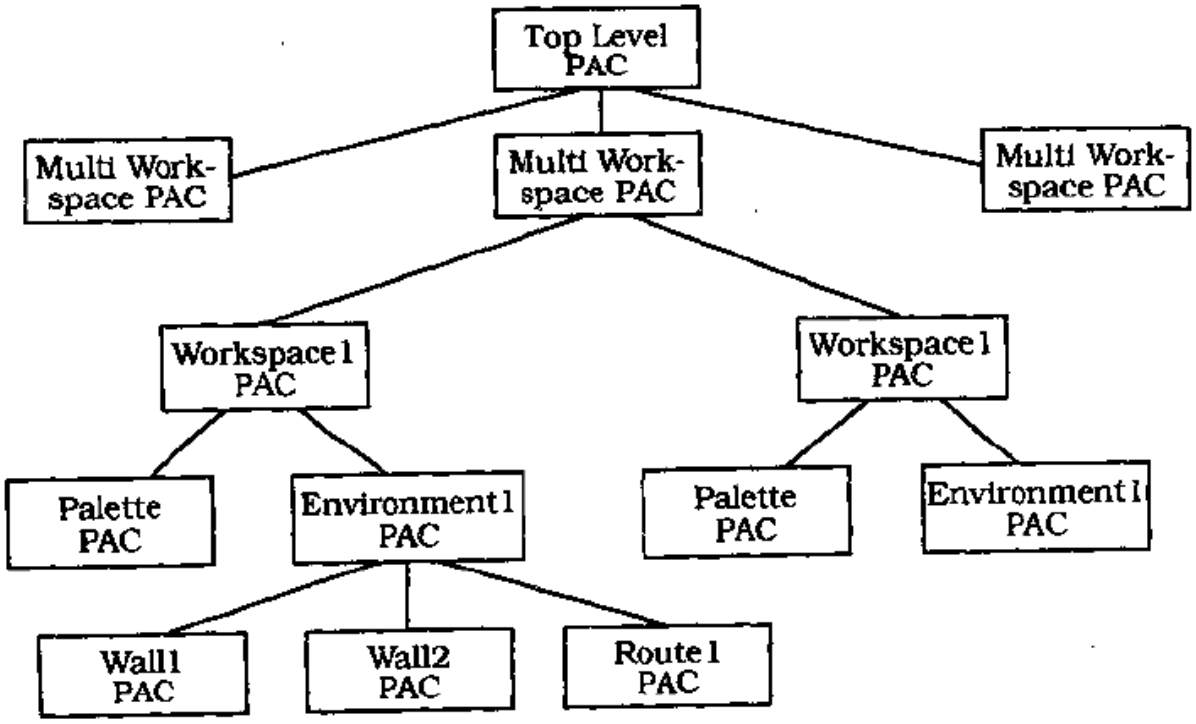


PAC模式

- 实现
  - 定义应用模型
  - 制定PAC层次结构的总体组织策略
  - 确定顶层智能体
  - 确定底层智能体
  - 确定底层用于系统服务的智能体
  - 确定复合下一层智能体的中间层智能体
  - 确定协调下一层智能体的中间层智能体
  - 将核心功能与人机交互分离
  - 提供外部接口
  - 将各层连接在一起

PAC模式

- 变种
  - PAC智能体为主动对象(multi-threading)
  - PAC智能体为进程 (进程间IPC或者分布式)
- 已知应用
  - 网络流量管理
  - 移动机器人



PAC模式

- 效果
  - 优点:
    - 分离关注点
    - 支持修改和扩展
    - 支持多任务，很容易实现智能体多线程、多进程和分布式
  - 不足
    - 增加了系统的复杂度
    - 复杂的控制组件
    - 效率问题
    - 可应用性，当语义概念过小过多时，PAC模式将变得复杂和难以维护，效率也低，就不再适用。

- 参见
  - MVC模式

“适应性系统”模式

设计支持修改和扩展的应用架构，以适应硬件、基础软件系统、用户界面以及客户要求的新特性。

典型模式

- Microkernel模式

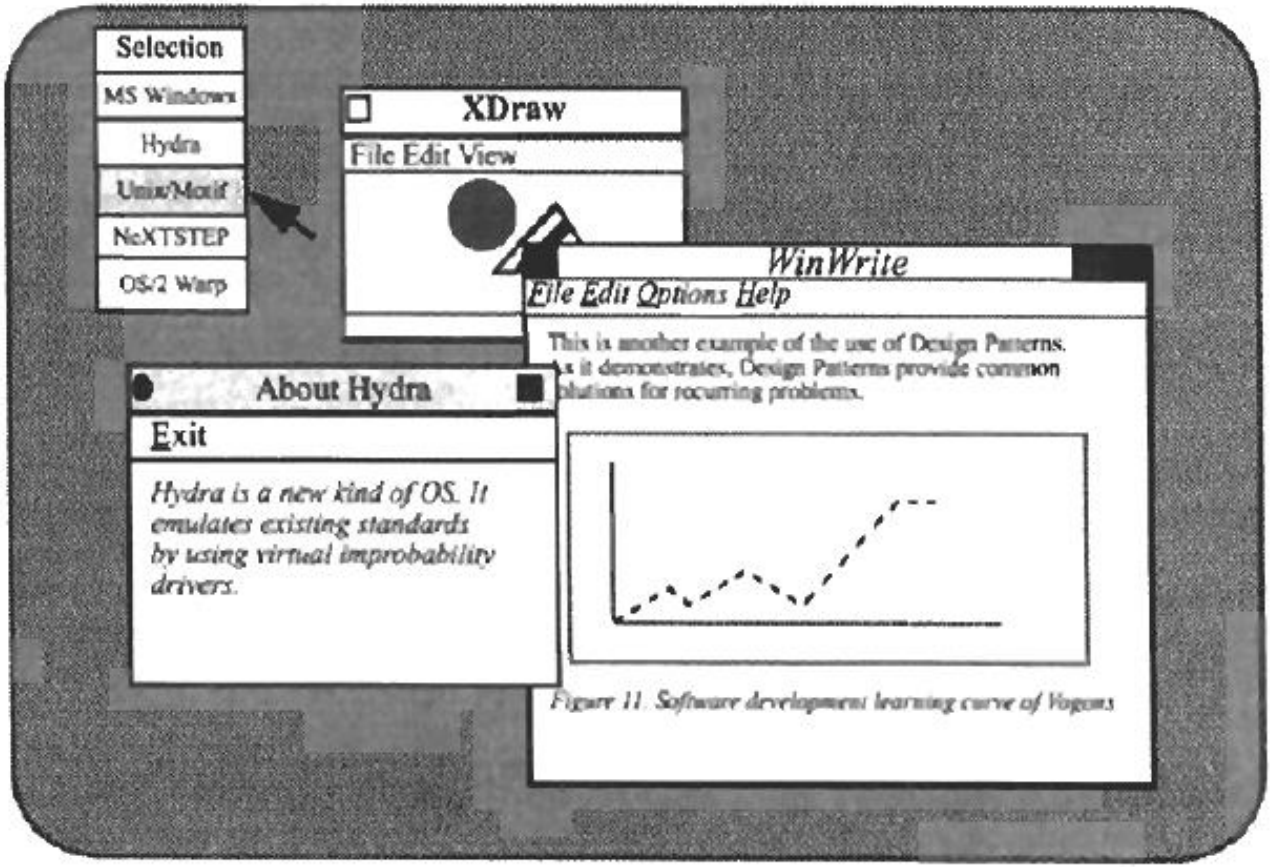
  将最基本的功能核心与扩展功能、特定客户部分分离开，微核还充当插座（socket）以插入扩展功能和协调它们之间的协作。

- Reflection模式

  提供了一种动态修改软件系统的结构和行为的机制。

Microkernel模式

- 示例（Hydra操作系统OS）



Microkernel模式

- 背景

  使用相似的、依赖于相同核心功能的编程接口开发多个应用

- 问题

  需考虑以下因素

  - 所开发应用的平台必须应对软硬件的持续演化
  - 为了便于集成新兴技术，应用平台应该是可移植、可扩充和可适应的
  - 在用户领域中需要支持不同的、但是相似的应用平台
  - 应用可以按组分类，它们按照不同的方式使用相同的功能核心，这要求底层的应用平台要模拟现有的标准
  - 应用平台的功能核心应被分离为一个占用尽可能少内存的组件和一些消耗尽可能少处理能力的服务

Microkernel模式

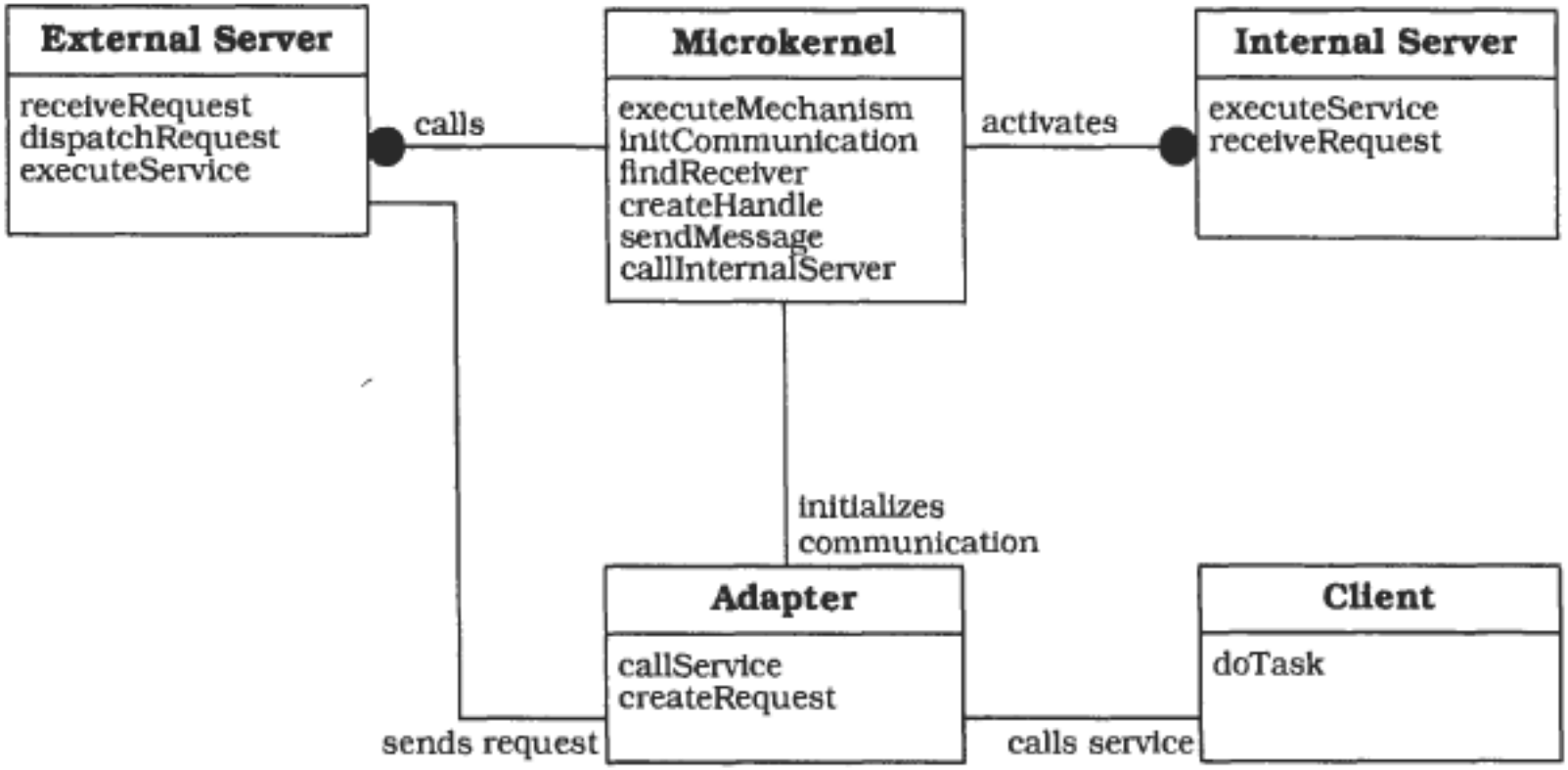
- 解决方案
  - 将应用平台的基本服务封装在一个微核组件中，微核包含了支持其他组件进程间通信、维护系统级资源（如文件和进程）和访问其功能的API（microkernel）
  - 不可避免会增加微核的大小和复杂度的核心功能应该被分离到内部服务器中（internal servers）
  - 外部服务器通过微核接口提供的机制实现了底层微核的视图，每个视图是一个独立的进程，也就是一个应用平台（external servers）
  - 客户端使用微核提供的通信工具与外部服务器通信（clients）

Microkernel模式

- 结构
  - 微核（microkernel）实现通信工具和资源管理等中心服务，封装了很多与系统相关的细节，为其他组件提供了一个或多个接口来访问这些基本服务。
  - 内部服务器（internal server）扩展了内核提供的功能，通常封装一些与底层硬件或软件系统相关的细节
  - 外部服务器（external server）使用微核实现其底层应用领域视图，并通过接口暴露其功能
  - 客户端（client）与单个外部服务器相关联的应用
  - 适配器（adapter）客户端和外部服务器之间的接口，避免客户端直接依赖于外部服务器

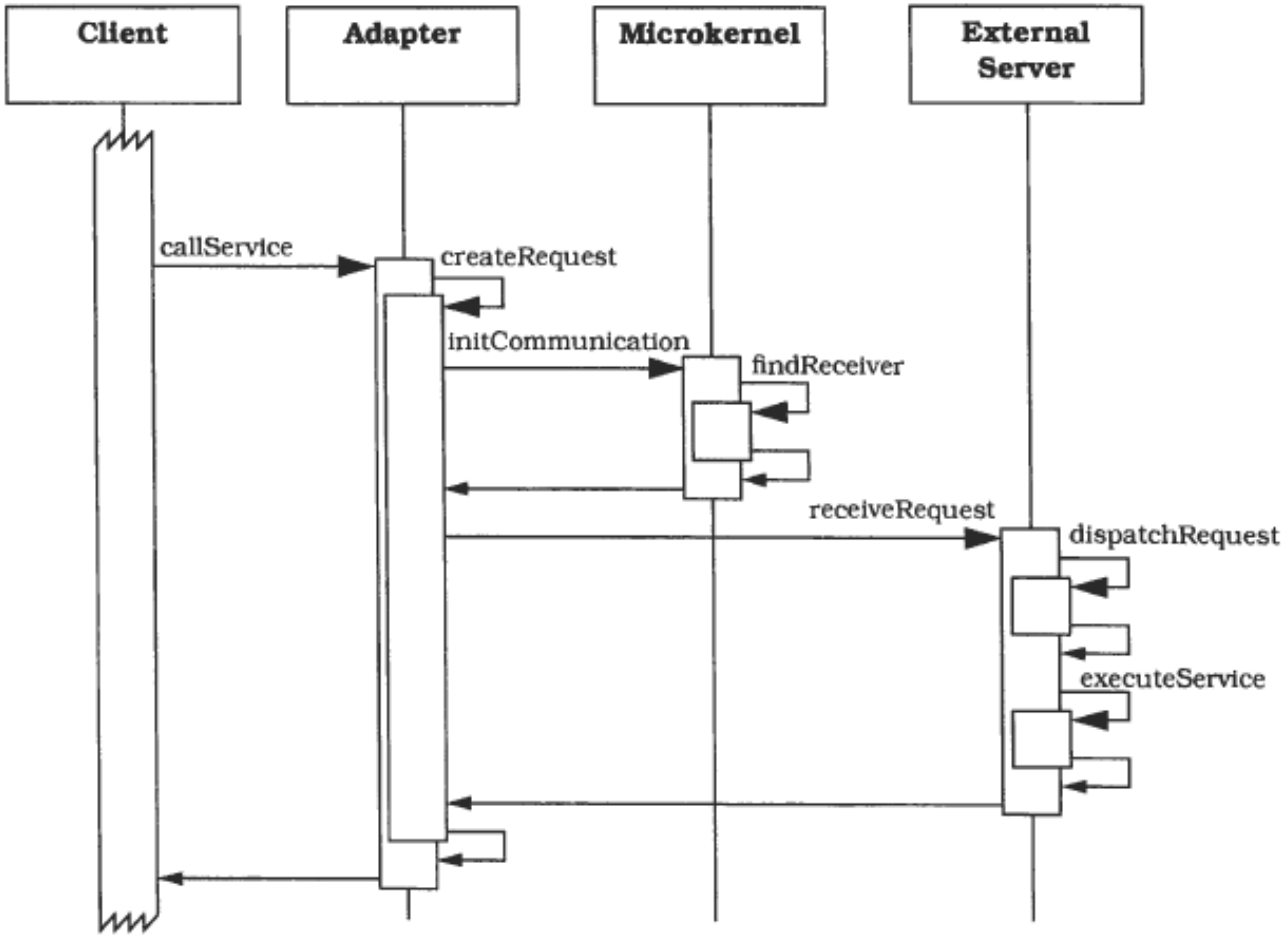
Microkernel模式

- 结构



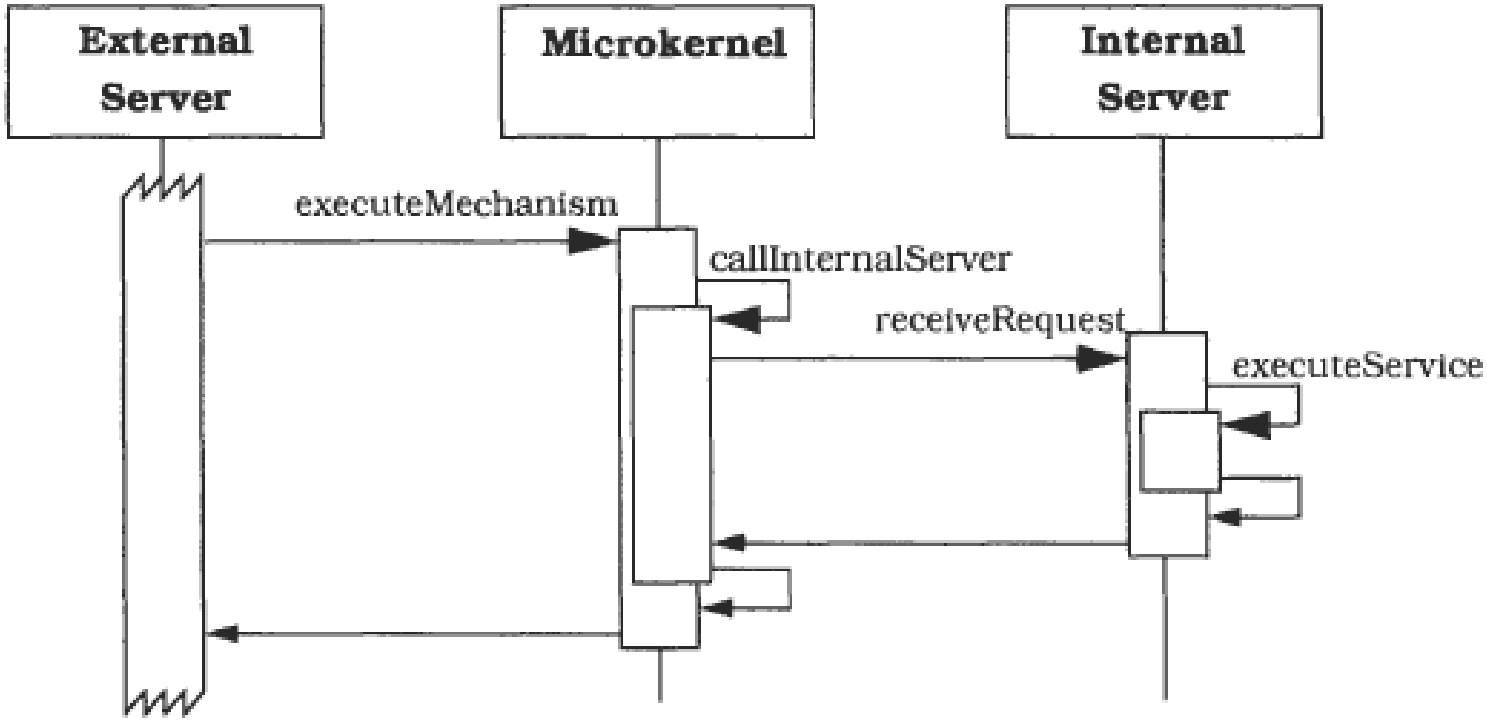
Microkernel模式

- 动态性
  - 情形1: 客户端调用相应外部服务器服务时的行为



Microkernel模式

- 情形2: 外部服务器通过微核调用内部服务器提供的服务



Microkernel模式

- 实现
  1. 分析应用领域
  2. 分析外部服务器
  3. 服务分类，找出与应用领域没有直接关系的基础服务
  4. 划分类别
  5. 为每一种类别找出其完整一致的操作和抽象集合
  6. 确定请求传输和返回的策略
  7. 确定微核组件的结构
  8. 描述微核的编程接口
  9. 微核负责管理所有的系统资源
  10. 内部服务器可以设计为独立进程或者共享库
  11. 实现外部服务器
  12. 实现适配器
  13. 开发客户端或直接使用现有客户端

Microkernel模式

- 变种
  - Microkernel System with indirect Client-Server connections
  - Distributed Microkernel System
- 已知应用
  - 操作系统Mach、Amoeba、Chorus、Windows NT
  - MKDE 数据库引擎
  - Eclipse

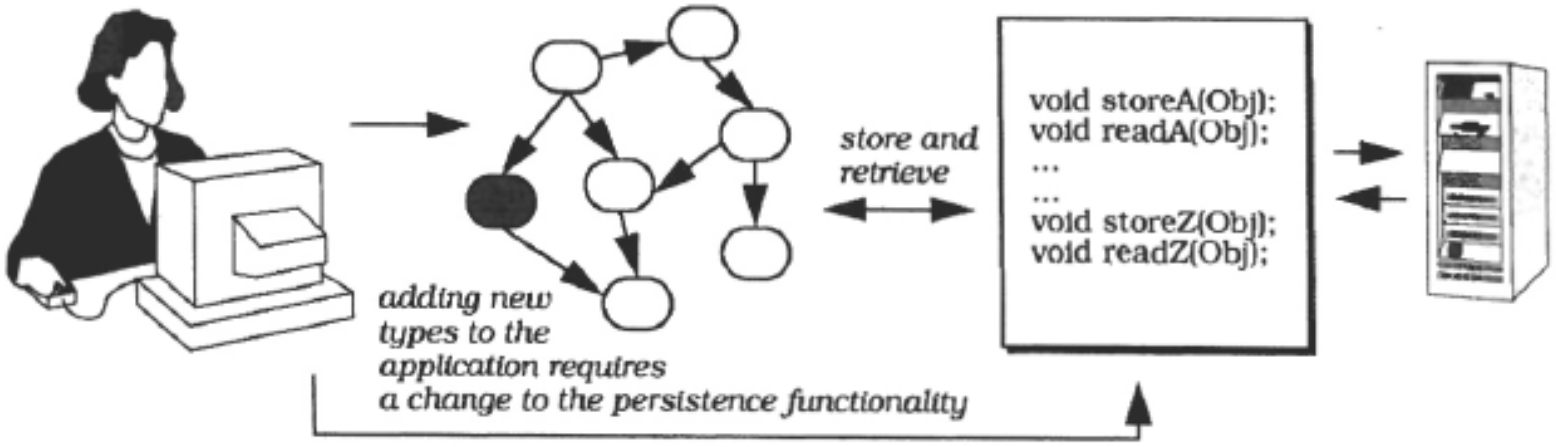
Microkernel模式

- 效果
  - 优点:
    - Portability
    - Flexibility and Extensibility
    - Separation of policy and mechanism
    - Scalability
    - Reliability
    - Transparency
  - 不足
    - Performance
    - Complexity of design and implementation
- 参见
  - Broker模式
  - Reflection模式
  - Layers模式

Reflection模式

提供了一种动态修改软件结构和行为的机制，程序分为元层（Meta level）和基层（Base level），元层描述系统属性的信息，让软件可以自己了解自己，而基层则包含了应用程序逻辑，实现依赖于元层，修改元层信息将影响基层的行为。

➢示例（对象的持久化）



Reflection模式

- 背景

  构建天生支持自修改的系统

- 问题
  - 修改软件是件繁琐、容易出错、代价昂贵的工作
  - 可适应软件系统通常内部结构复杂
  - 保持系统可修改的技术（如参数化、子类化、混入、复制和粘贴）使用的越多，系统的修改就变得越别扭和复杂
  - 修改规模可以从修改命令快捷键到修改某个客户的应用框架
  - 甚至是系统的基本部分，如组件之间的通信机制，都可能需要修改

Reflection模式

- 解决方案

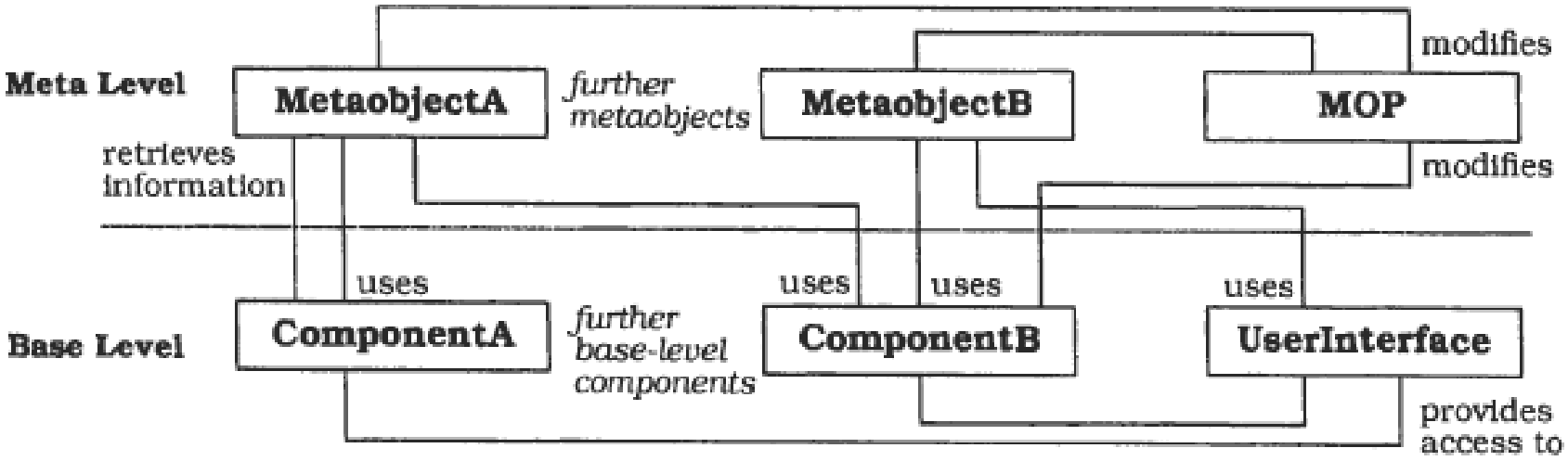
  软件需要可以自我感知，并且其结构和行为的某些方面可以被调整和修改，为此将软件分为两个主要部分: 元层（meta level）和基层（base level）

  - 元层提供了软件的自表示，让软件可以自己了解其结构和行为。元层由元对象（metaobject）组成，封装了类型结构、算法和函数调用机制等
  - 基层定义了应用逻辑。通过使用元对象来实现基层，可以保持那些未来可能变化的方面的独立性。
  - 使用称为MOP（metaobject protocol）的接口操作元对象，允许客户端描述特定的修改。例如修改函数调用机制。通过MOP对元对象所做的修改都会影响到基层对象的后续行为。

Reflection模式

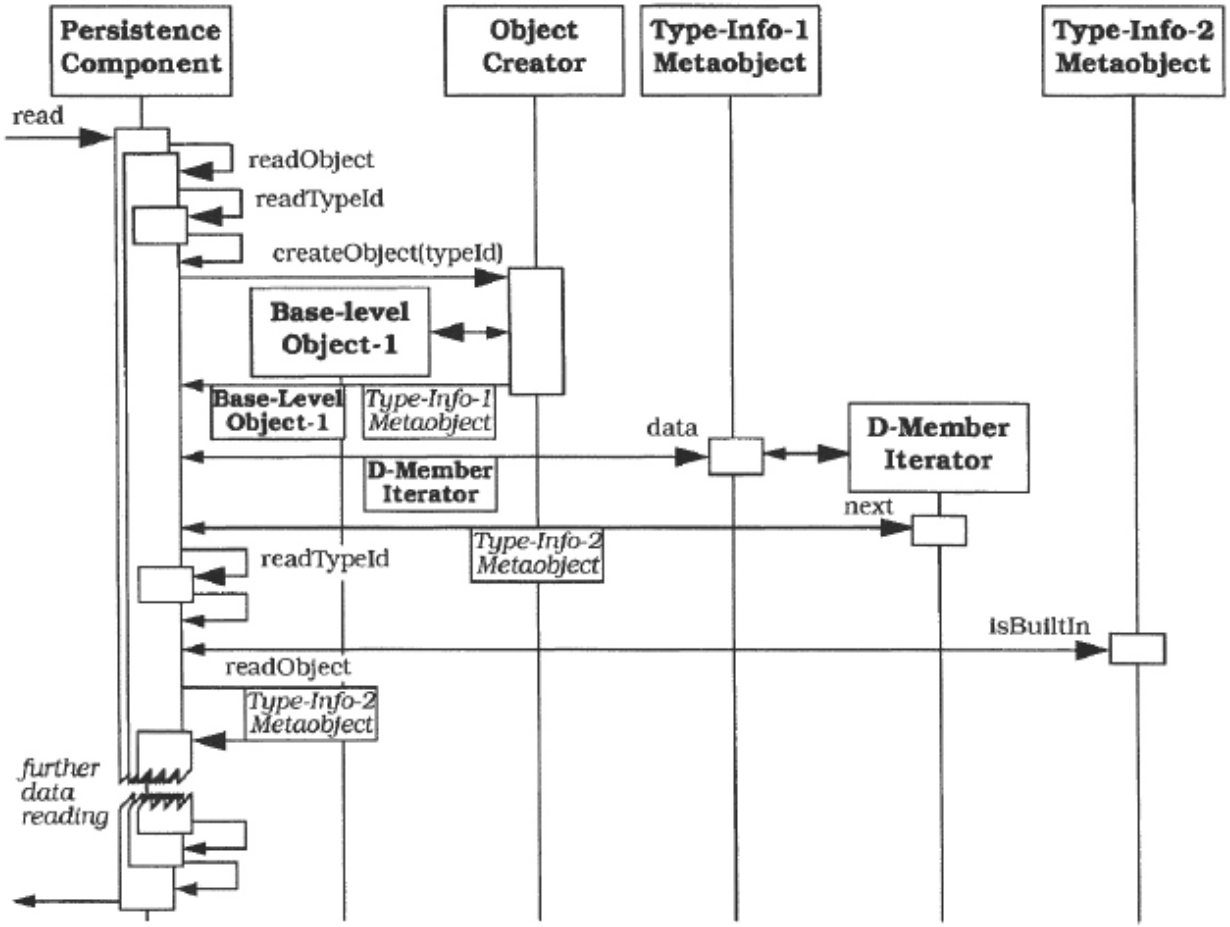
- 结构
  - 元层，包含一组元对象，每个元对象都封装了有关基层的某个结构、行为或状态方面的信息。所有的元对象一起构成了应用的自我表示。用元对象提供什么取决于系统需要适应哪些方面的变化，元对象应该只封装那些会发生变化的系统细节。
  - 基层实现了软件的应用逻辑，其组件提供了各种系统服务及其底层数据模型
  - 元对象协议MOP向外部提供访问元层的接口，以确定的方式访问反射系统实现

Reflection模式



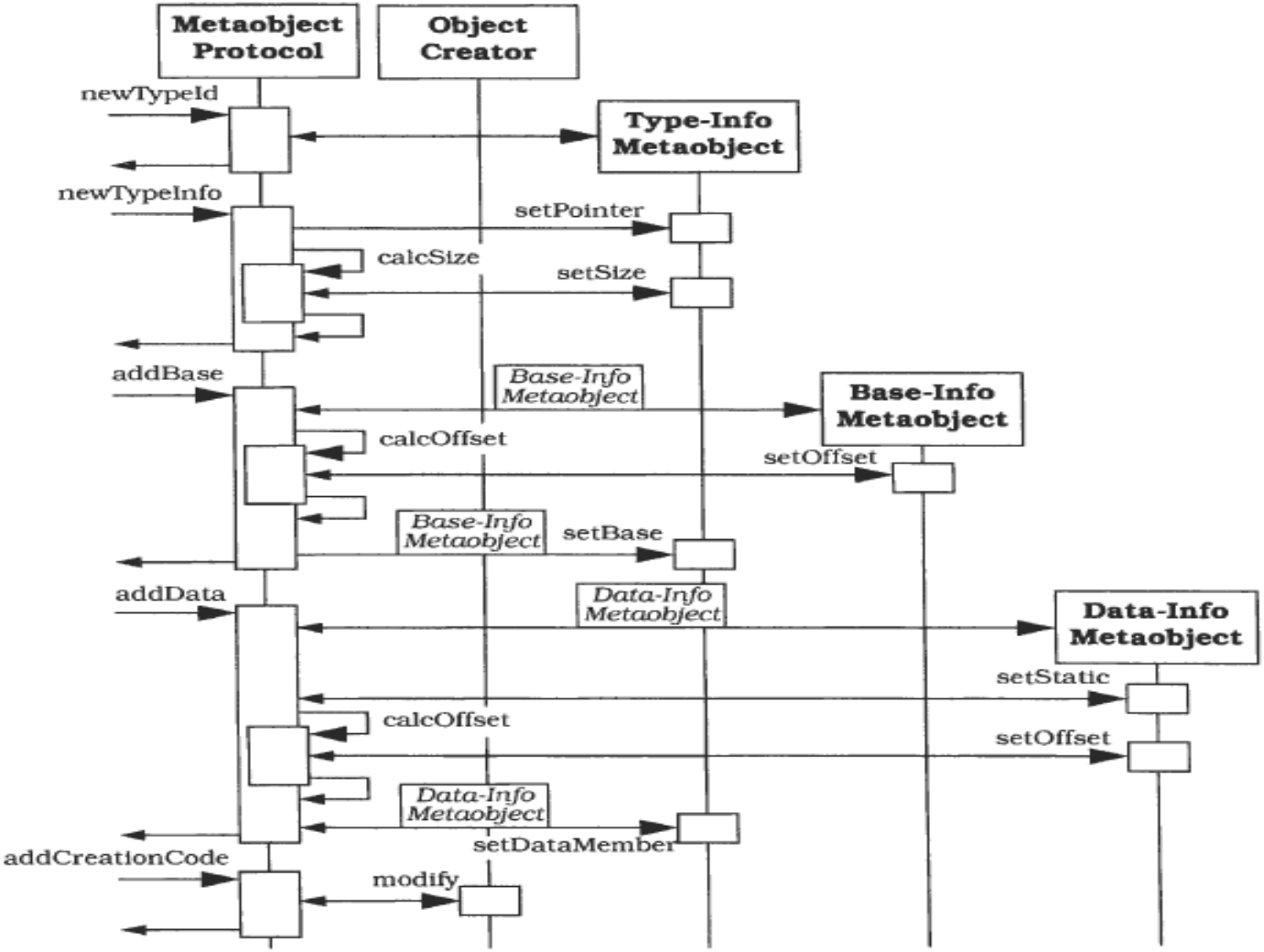
Reflection模式

- 动态（以C++对象持久化为例，使用了C++的RTTI机制
  - 情形1: 读取磁盘文件中对象时基层和元层的协作行为



Reflection模式

- 情形2: 使用MOP向元层添加类型信息



Reflection模式

- 实现
  1. 定义应用的模型
  2. 识别变化的行为
  3. 识别发生变化时，不应影响基层实现的系统的结构化方面
  4. 识别系统服务，这些服务支持第2步中应用服务的变化和第3步中结构细节的独立部分
  5. 定义元对象
  6. 定义MOP
  7. 定义基层.

Reflection模式

- 变种
  - 包含多个元层的Reflection模式
- 已知应用
  - CLOS
  - MIP
  - OLE 2.0

Reflection模式

- 效果
  - 优点
    - No explicit modification of source code
    - Changing a software system is easy
    - Support for many kinds of change
  - 不足
    - Modifications at the meta level may cause damage. The robustness of a metaobject protocol is therefore of great importance.
    - Increased number of components.
    - Lower efficiency
    - Not all potential changes to the software are supported
    - Not all languages support reflection, e.g C++
- 参见
  - Microkernel模式

其它风格或模式

- 客户/服务器体系结构
- 正交软件体系结构
- 基于层次消息总线的体系结构
- 异构体系结构
- SIS体系结构
- 对象管理体系结构
- Web Services
- 面向服务体系结构
- 对等计算体系结构
- 特定领域体系结构

- 每种风格有自己的特点和适应性，需要根据目标问题的实际情况，选择合适的一种或多种风格，并有机地整合起来