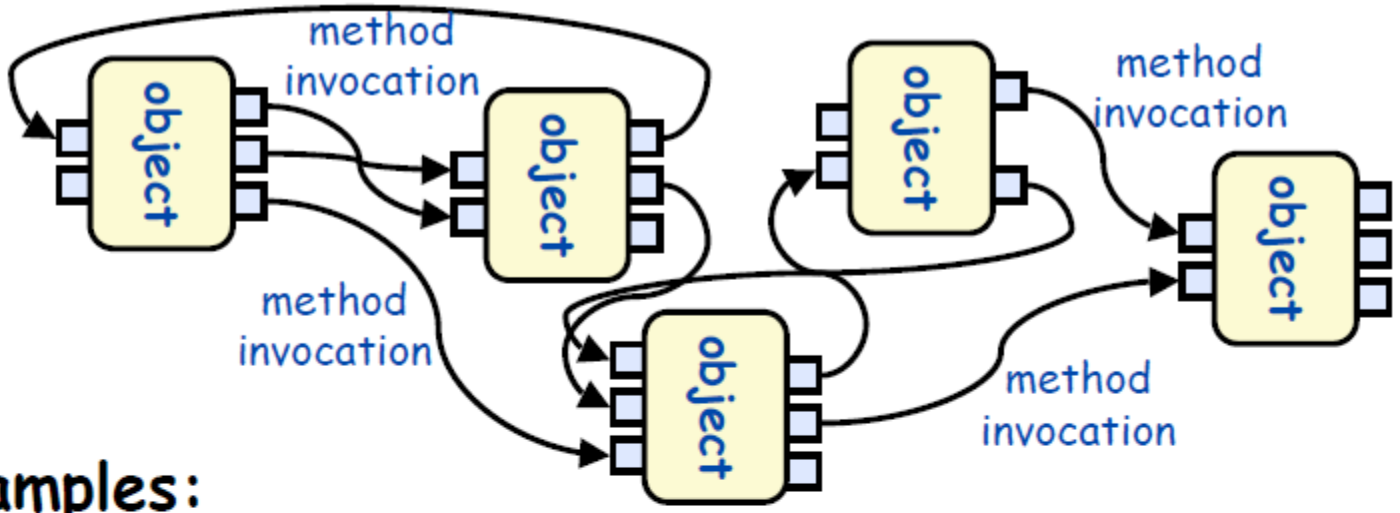
2. Software Architecture Style
（软件体系结构风格）

Call/Return

- Main program and subroutines
  - Classical programming paradigm-functional decomposition
- Object-Oriented/Abstract Data Types
  - Information (representation, access method) hiding
- Layered hierarchies
  - Each level only communicates with its immediate neighbors
- Other
  - Client-server
  - ……

Object-Oriented Architecture



→ Examples:

- abstract data types

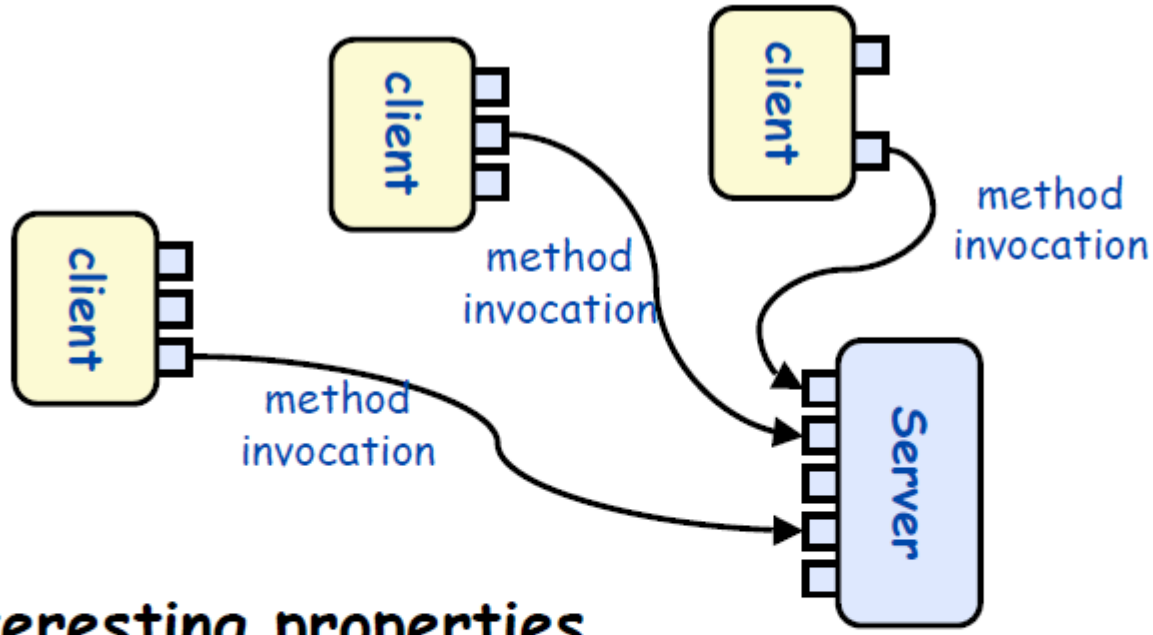
→ Interesting properties

- data hiding (internal data representations are not visible to clients)
- can decompose problems into sets of interacting agents
- can be multi-threaded or single thread

→ Disadvantages

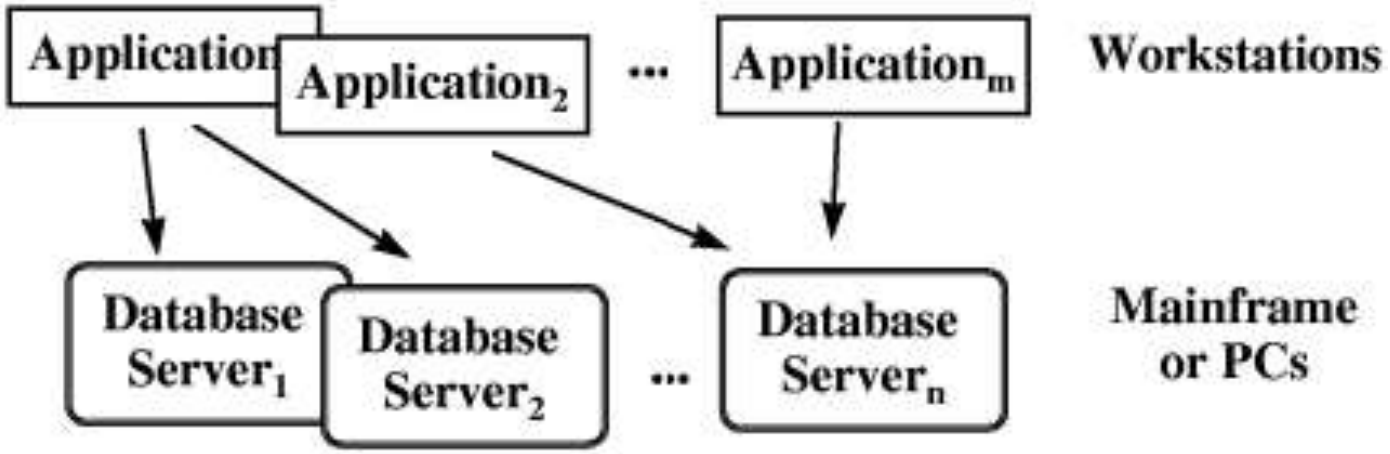
- objects must know the identity of objects they wish to interact with

Variant 1: Client/Server



- Interesting properties
  - Is a special case of the previous pattern object oriented architecture
  - Clients do not need to know about one another
- Disadvantages
  - Client objects must know the identity of the server

两层Client/Server



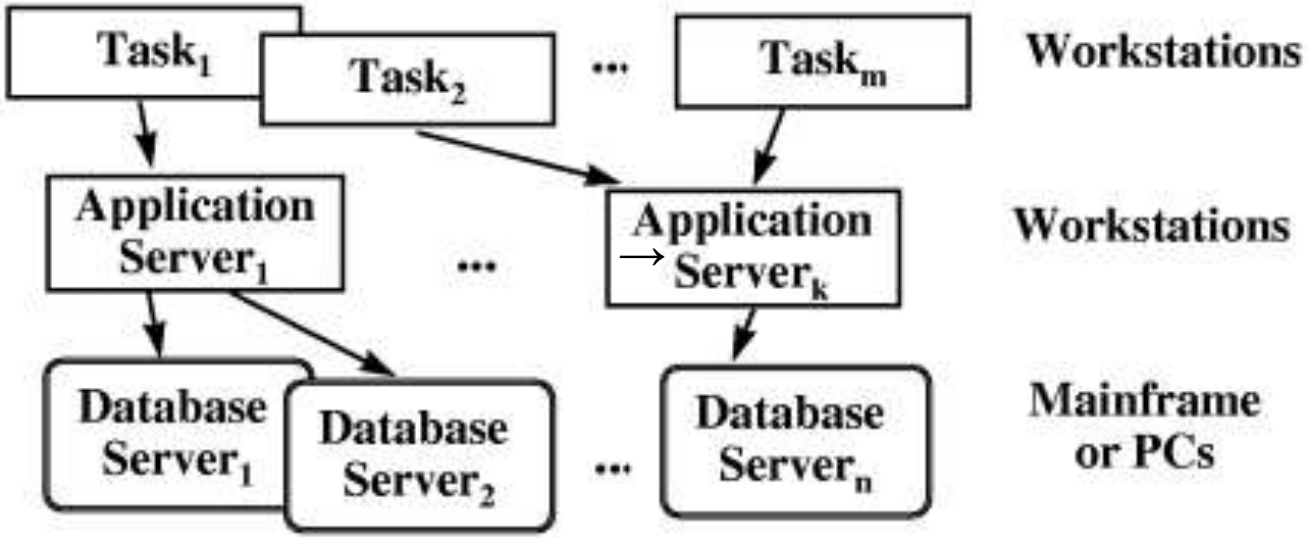
两层Client/Server

- C/S软件体系结构是基于资源不对等，且为实现共享而提出的，20世纪90年代成熟起来
- C/S体系结构有三个主要组成部分：数据库服务器、客户应用程序和网络
- 服务器（后台）负责数据管理，客户机（前台）完成与用户的交互任务。“胖客户机，瘦服务器”
- 缺点：
  - 对客户端软硬件配置要求较高，客户端臃肿
  - 客户端程序设计复杂
  - 数据安全性不好。客户端程序可以直接访问数据库服务器。
  - 信息内容和形式单一
  - 用户界面风格不一，使用繁杂，不利用推广使用
  - 软件维护与升级困难。每个客户机上的软件都需要维护

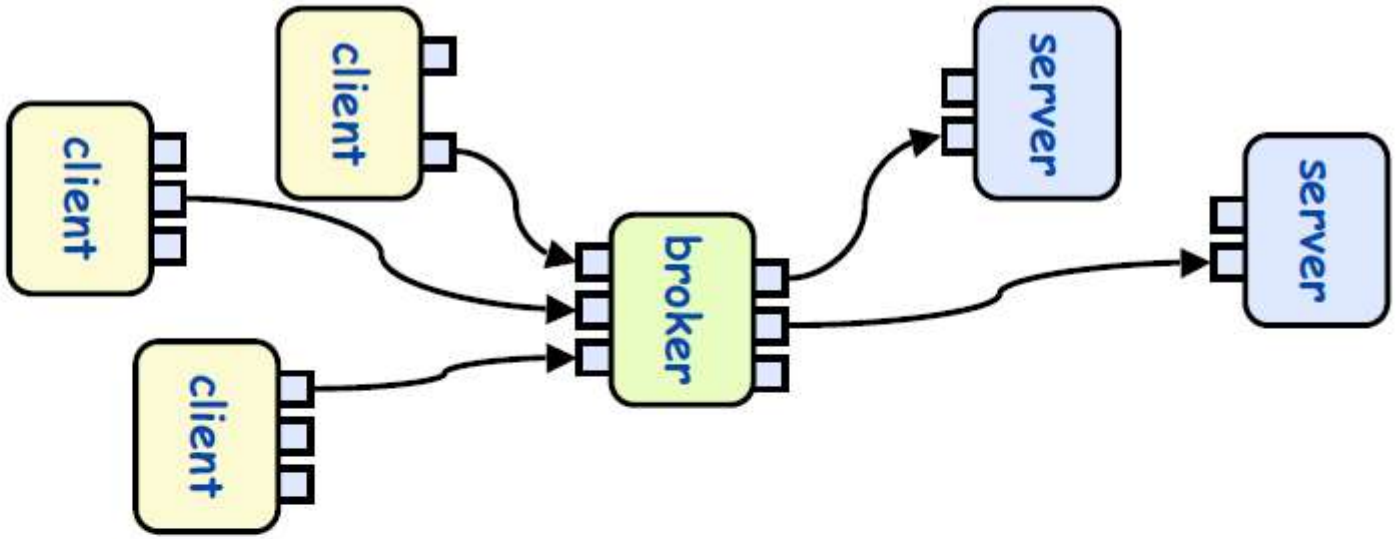
三层Client/Server→Application-Server

- 与二层C/S结构相比，增加了一个应用服务器。
- 整个应用逻辑驻留在应用服务器上，只有表示层存在于客户机上——“瘦客户机”。
- 应用功能分为表示层、功能层、数据层三层
  - 表示层是应用的用户接口部分。通常使用图形用户界面
  - 功能层是应用的主体，实现具体的业务处理逻辑
  - 数据层是数据库管理系统。
  - 以上三层逻辑上独立。
  - 通常只有表示层配置在客户机中

Client-Application-Server…



Variant 1: Object Broker



→ Interesting properties

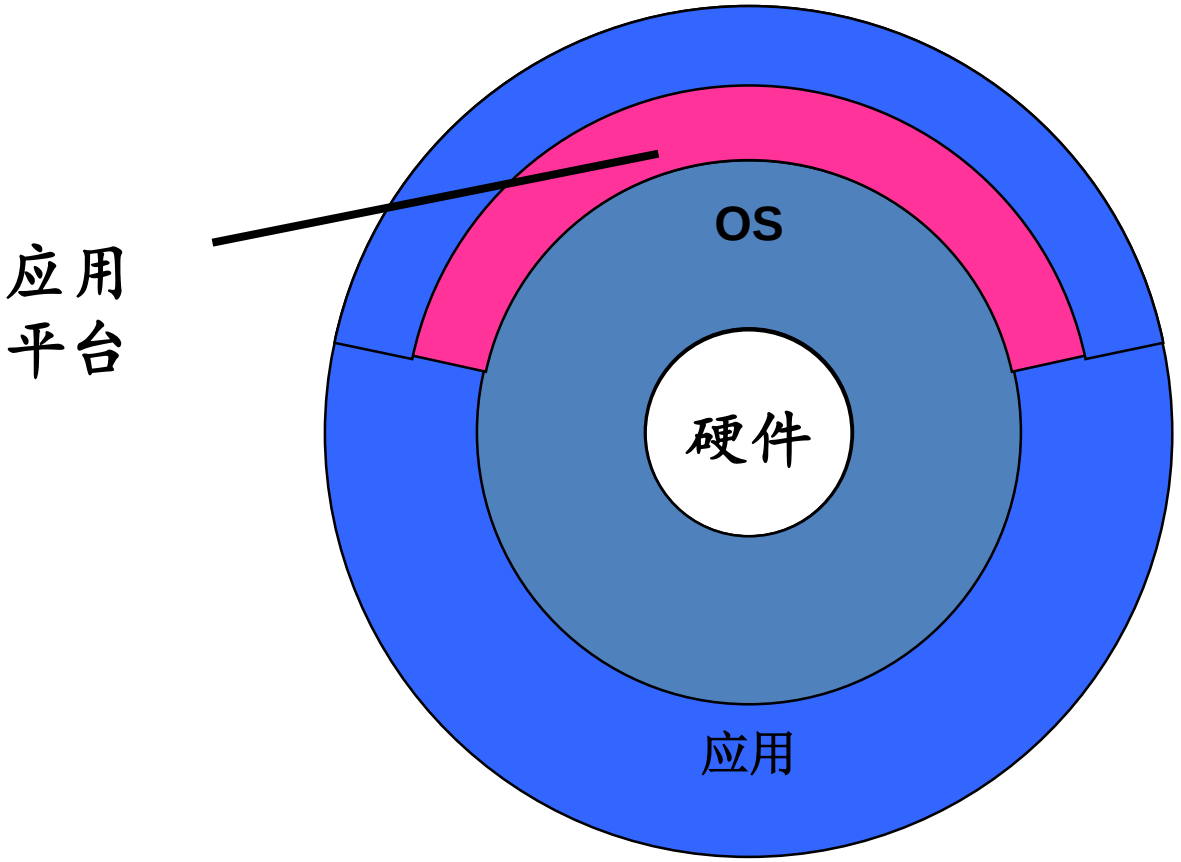
- Adds a broker between the clients and servers
- Clients no longer need to know which server they are using
- Can have many brokers, many servers.

→ Disadvantages

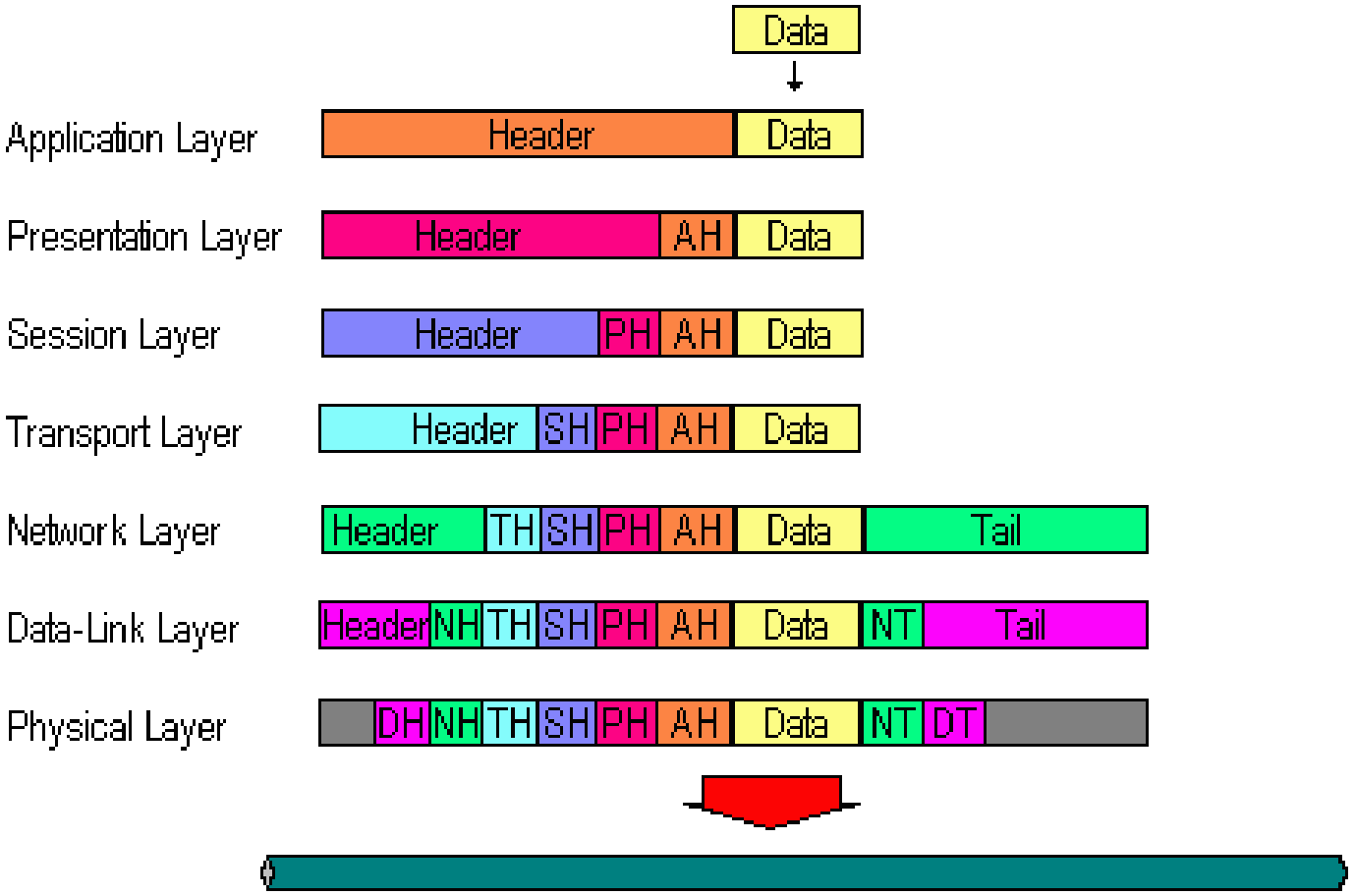
- Broker can become a bottleneck
- Degraded performance

Layered System

- 无处不在的层次结构



OSI 参考模型



Layered Architectural Pattern

- **Problem: This pattern is suitable for applications that involve distinct classes of services that can be arranged hierarchically. Often there are layers for basic *system-level services*, for *utilities* appropriate to many applications, and for specific *tasks* of the application.**
- **Context: Frequently, each class of service is assigned to a layer and several different patterns are used to refine the various layers. Layers are most often used at the higher levels of design, using different patterns to refine the layers.**

Layered Architectural Pattern

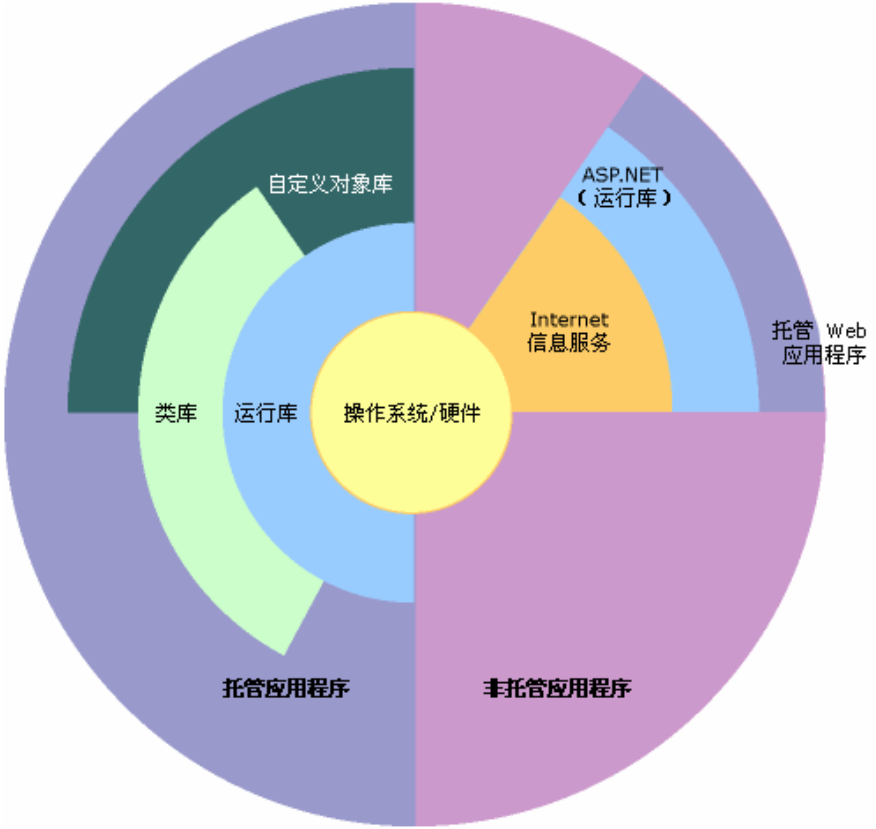
- **Solution:**
  - ***System model:*** **hierarchy of opaque（*不透明的*）layers**
  - ***Components:*** **usually composites; composites are most often collections of procedures**
  - ***Connectors:*** **depends on structure of components; often procedure calls under restricted visibility, might also be client/server**
  - ***Control structure:*** **single thread**

层次风格特点

- 优点：
  - 分层风格支持系统设计过程中的逐级抽象
  - 基于分层风格的系统具有较好的可扩展性
  - 分层风格支持软件复用
- 不足：
  - 并不是所有的系统都适合用分层风格来描述的
  - 对于抽象出来的功能具体应该放在哪个层次上也是设计者头疼的一个问题

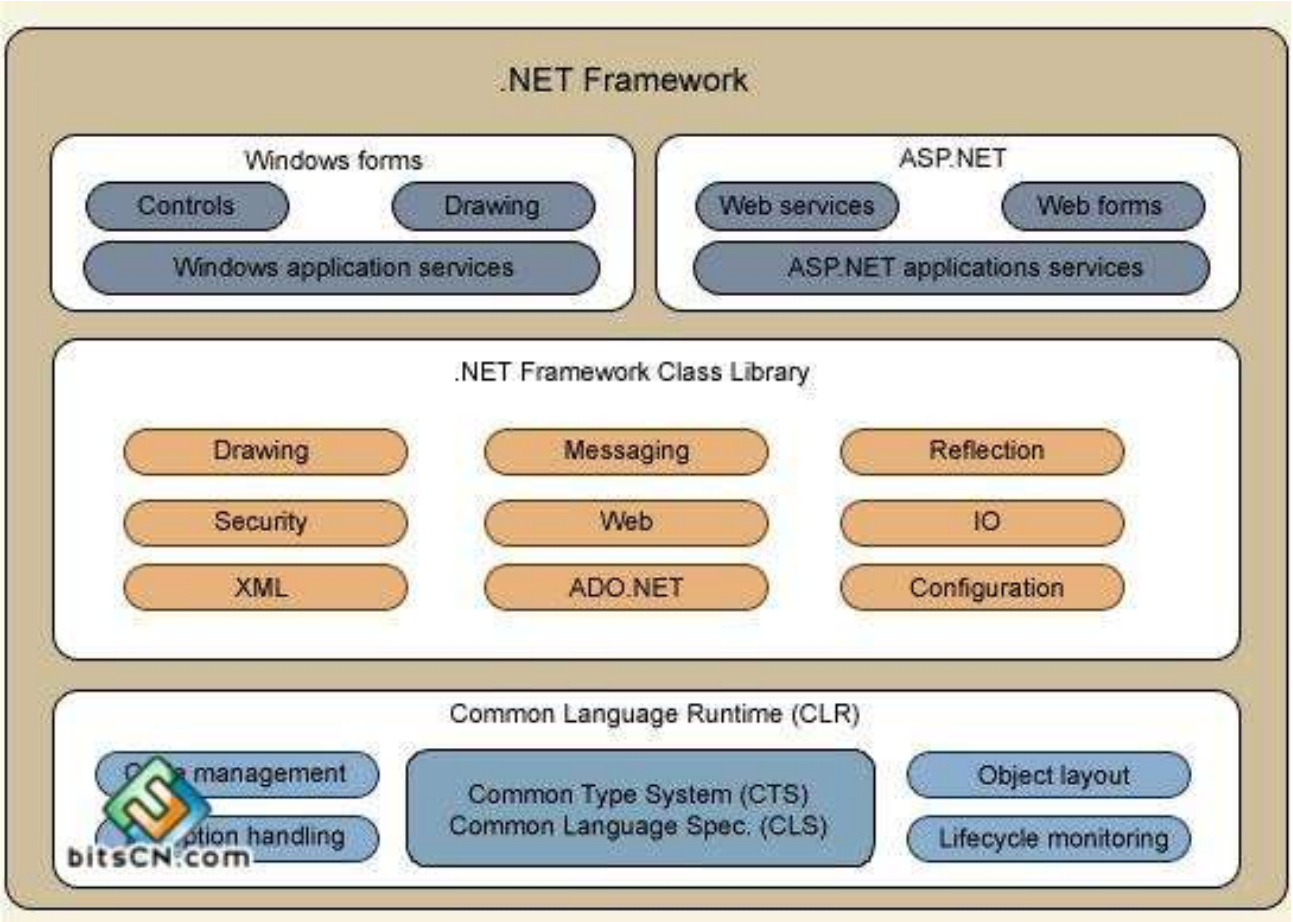
Example

- .Net平台也是一个明显的分层系统：



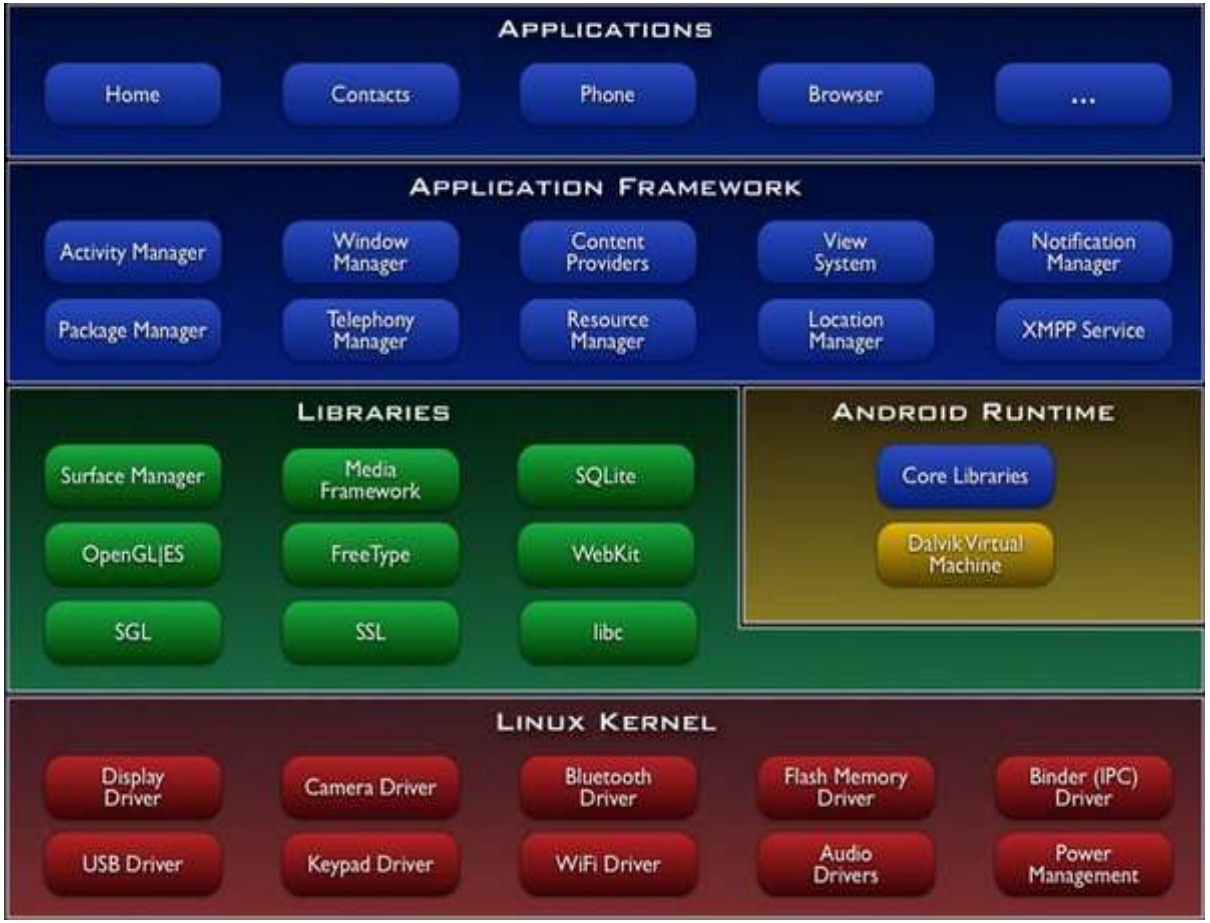
Example

- .Net 框架分层模图



Example

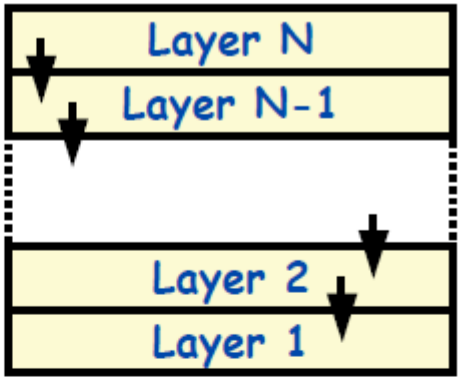
- **Android的内核架构**



Open vs. Closed Layered Architecture

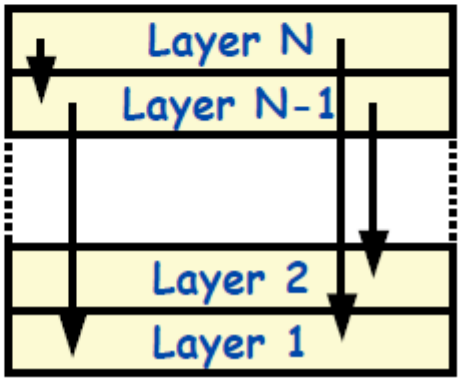
→ closed architecture

- each layer only uses services of the layer immediately below;
- Minimizes dependencies between layers and reduces the impact of a change.



→ open architecture

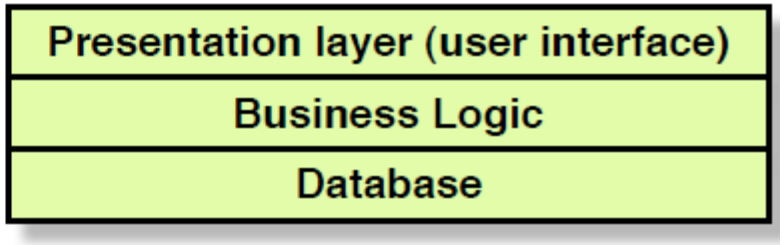
- a layer can use services from any lower layer.
- More compact code, as the services of lower layers can be accessed directly
- Breaks the encapsulation of layers, so increase dependencies between layers



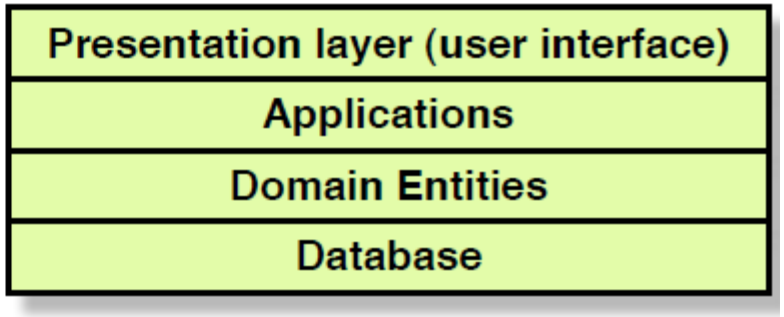
How many layers?

- 2-layers:
  - application layer
  - database layer
  - e.g. simple client-server model

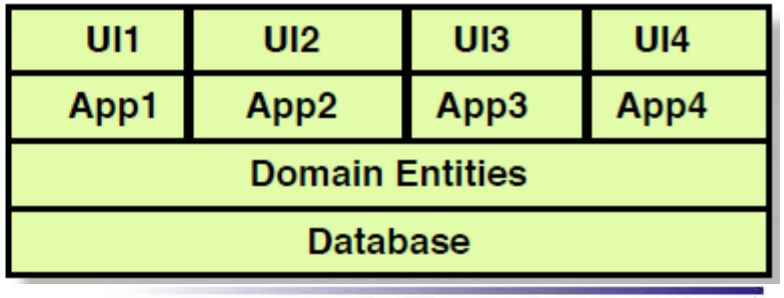
- 3-layers:
  - separate out the business logic
    - helps to make both user interface and database layers modifiable



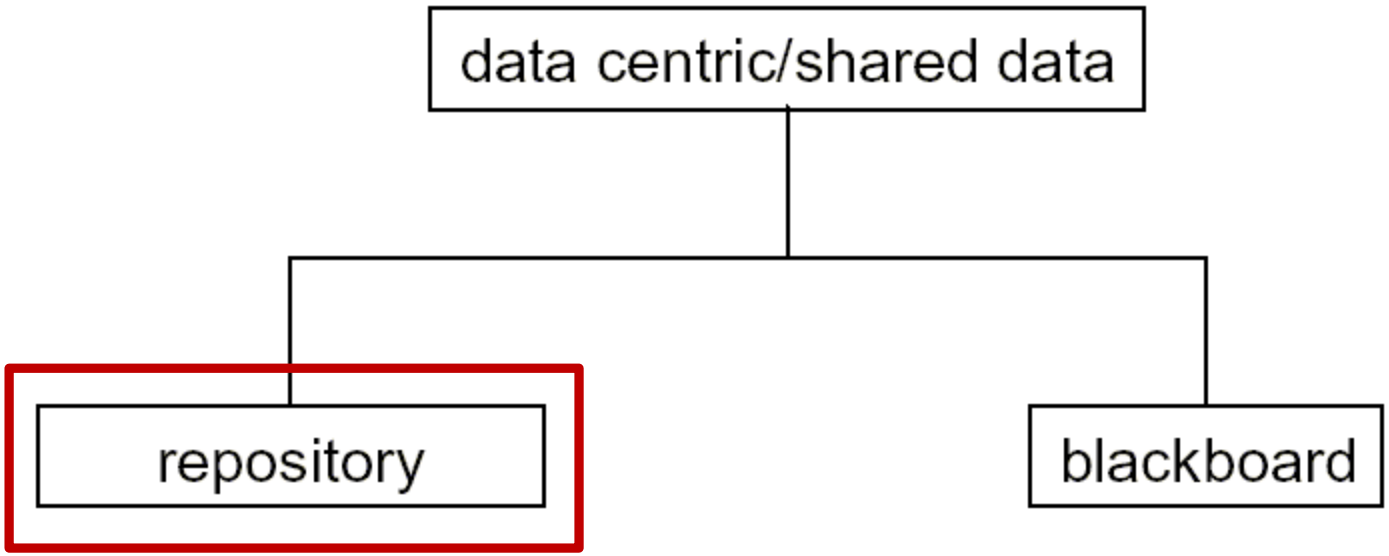
- 4-layers:
  - Separates applications from the domain entities that they use:
    - boundary classes in presentation layer
    - control classes in application layer
    - entity classes in domain layer



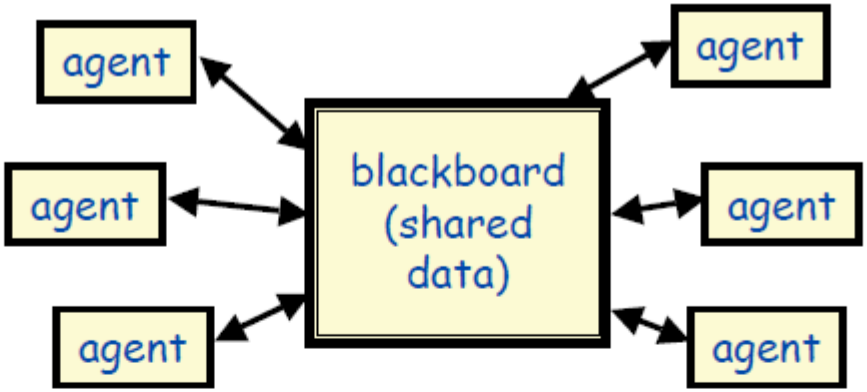
- Partitioned 4-layers
  - identify separate applications



Data Centered/Shared Data



Repository



→ Examples
- databases
- blackboard expert systems
- programming environments

→ Interesting properties
- can choose where the locus of control is (agents, blackboard, both)
- reduce the need to duplicate complex data

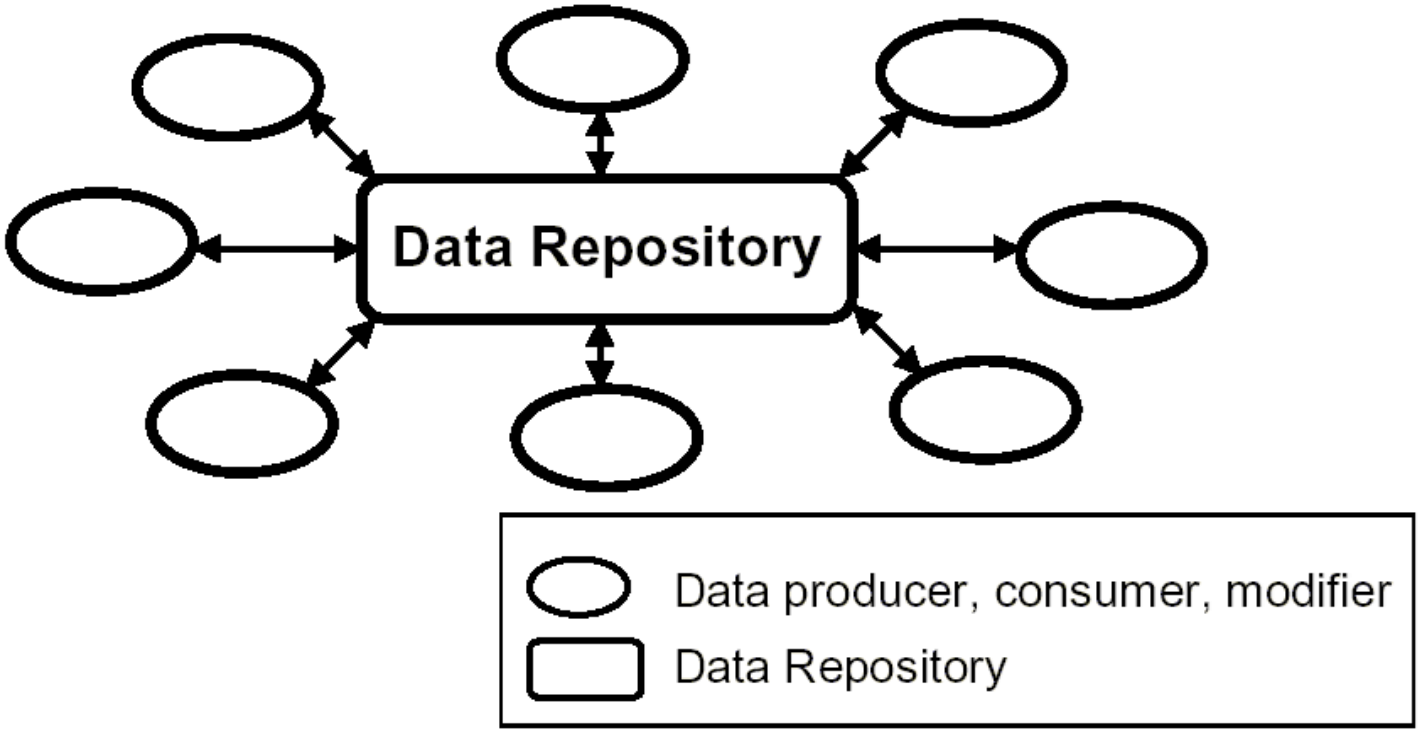
→ Disadvantages
- blackboard becomes a bottleneck

Shared Information Systems

- Style represents a large variety of systems
  - many variants depending on nature of shared data（共同特点是共享数据）
- Style addresses mechanisms for:
  - collecting, manipulating and preserving large bodies of data（收集、操作、保存大量的数据）
- Databases are a natural example, but not the only one
  - we'll see others in this lecture

Shared Information Systems

- High level view
  - what are the apparent features of this style?
  - what are some issues regarding this style?



Shared Information Systems

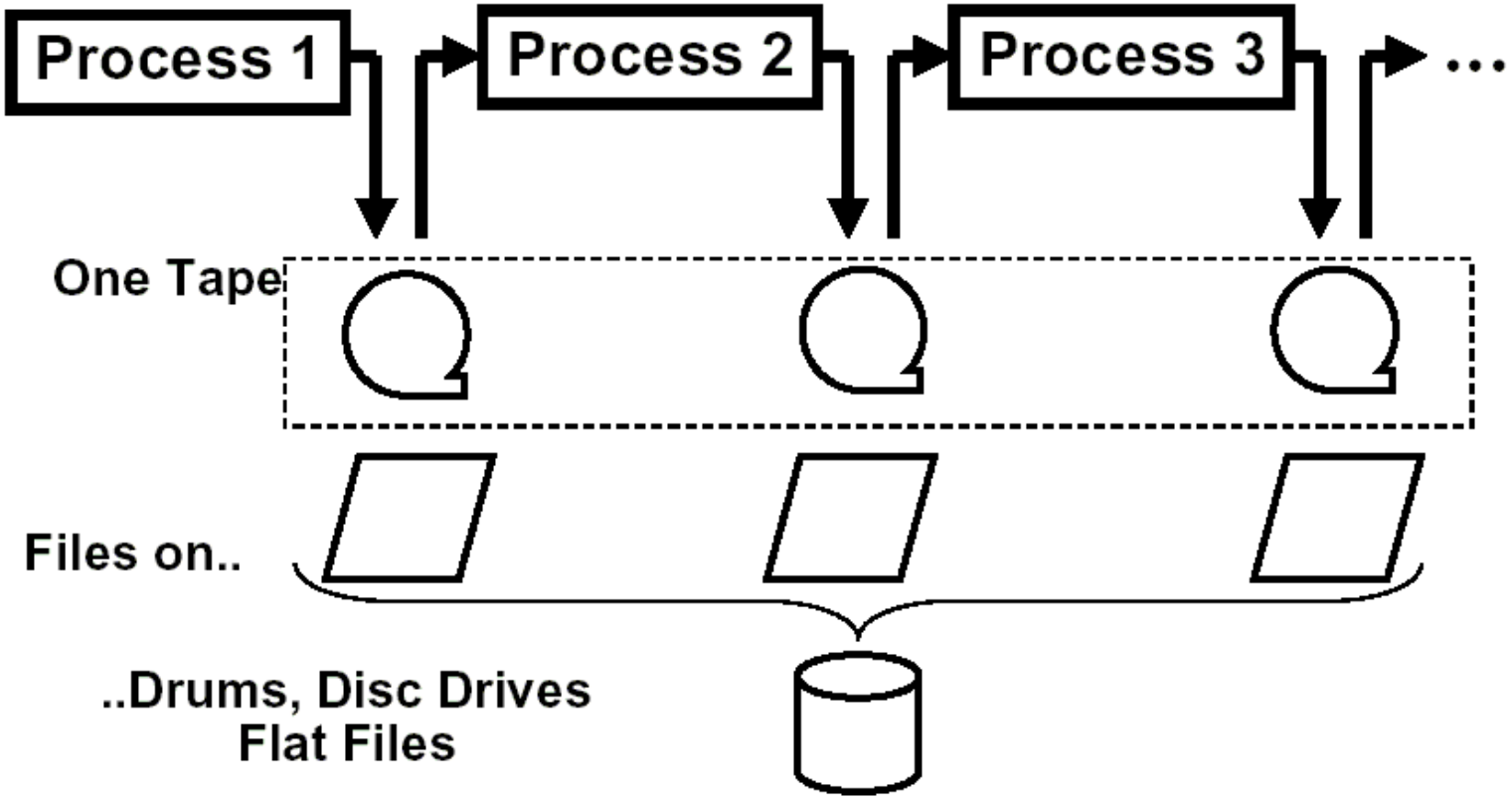
- **Advantages**
  - **easy to add consumers and producers of data（*很容易增加数据的生产者和消费者*）**
    - **How about modifier?**
- **Issues**
  - **synchronization（*同步*）**
  - **configuration and schema management（*配置和管理*）**
  - **atomicity（*原子性*）**
  - **consistency（*一致性*）**
  - **persistence（*持久性*）**
  - **performance（*性能*）**

Shared Information Systems

- Earliest repositories appear in batch sequential systems（批处理系统）
  - mainframes, drums（磁鼓）, magnetic tapes, disc drives
  - resources manually managed
- Pressure for on-line access to data
  - requirement to make access to data easy and instant
  - help to drive the shift from batch-sequential to interactive processing
- Today
  - shared information systems appear everywhere from the smallest business, to the most advanced scientific applications
  - many applications provide access mechanisms to shared data
  - the Web has become a giant distributed repository

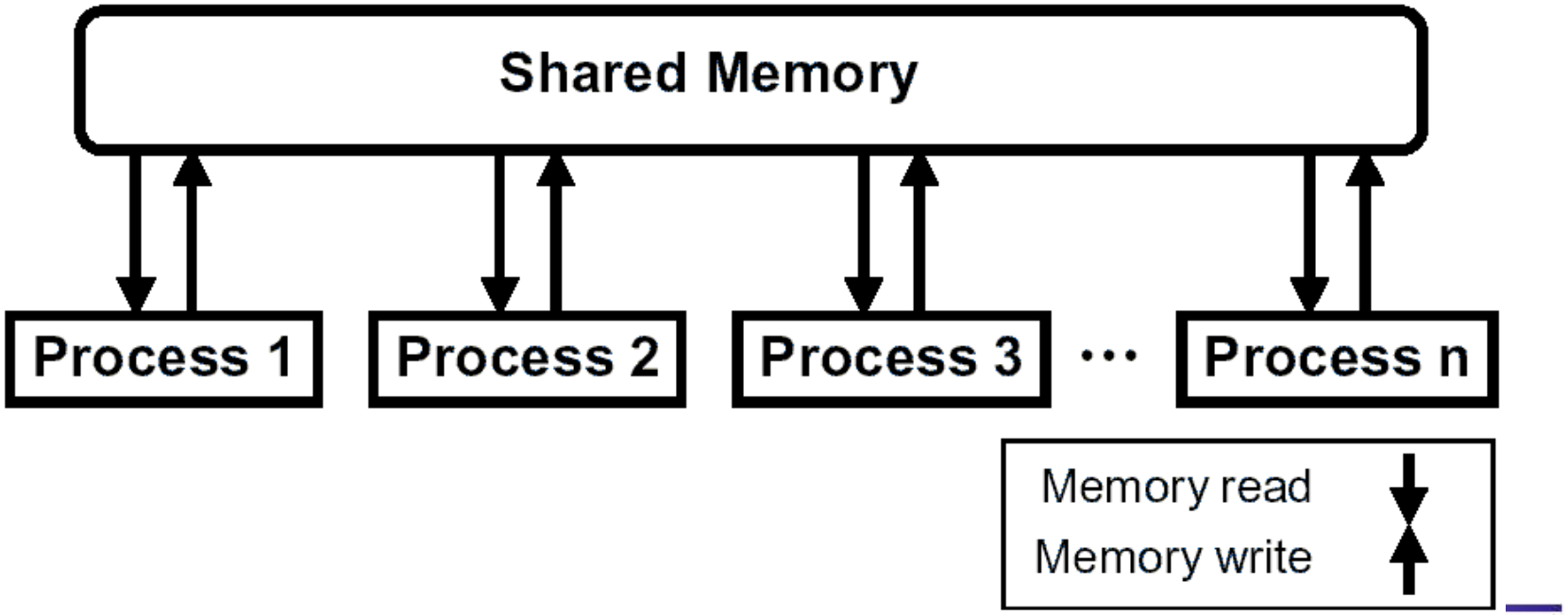
Evolution of Shared Information Systems

- **Batch Sequential Systems**
  - **flat file access (I/O)**

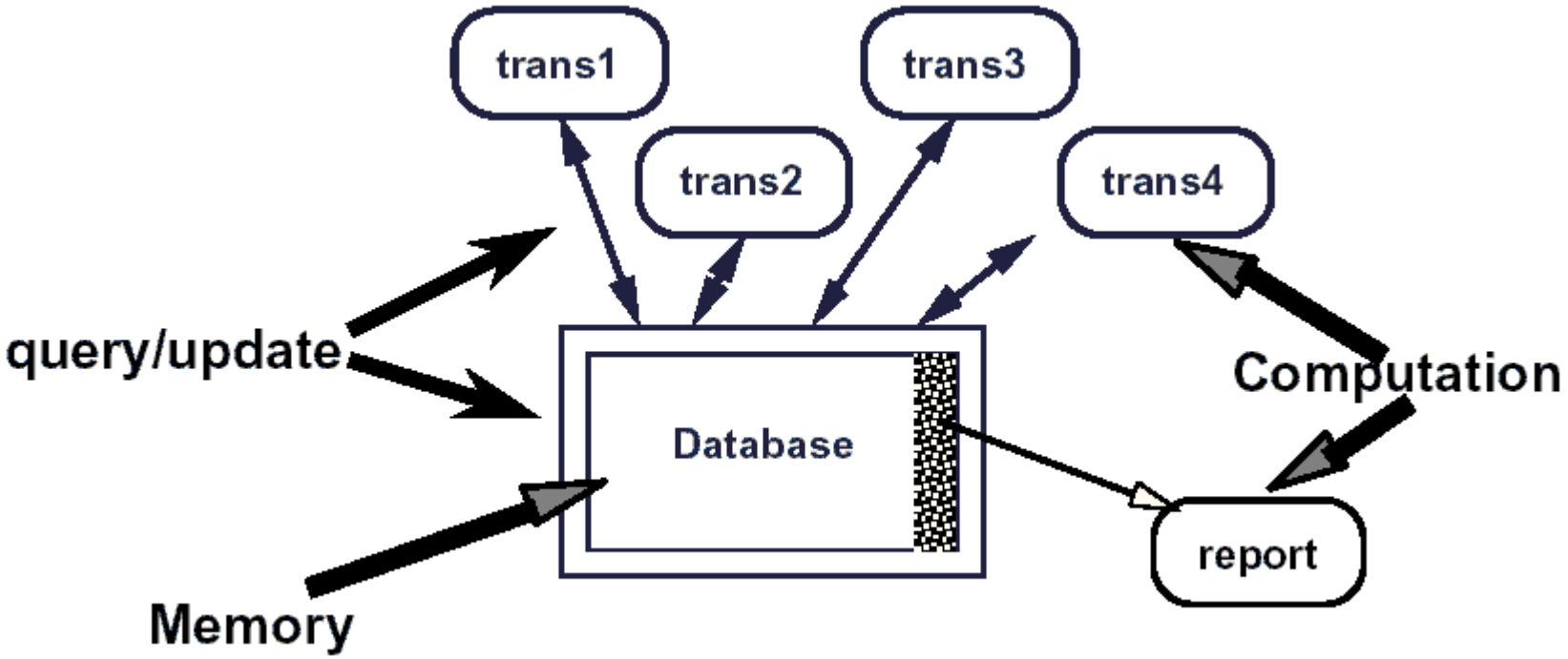


Evolution of Shared Information Systems

- **Datapool（数据池）(Shared Memory) Example:**
  - **Enabled by availability of RAM and languages to permit the sharing of common data (E.g., FORTRAN COMMON BLOCK)**
  - **Processes are not necessarily sequential**
  - **Gets messy without implementation rules - what are the issues?**

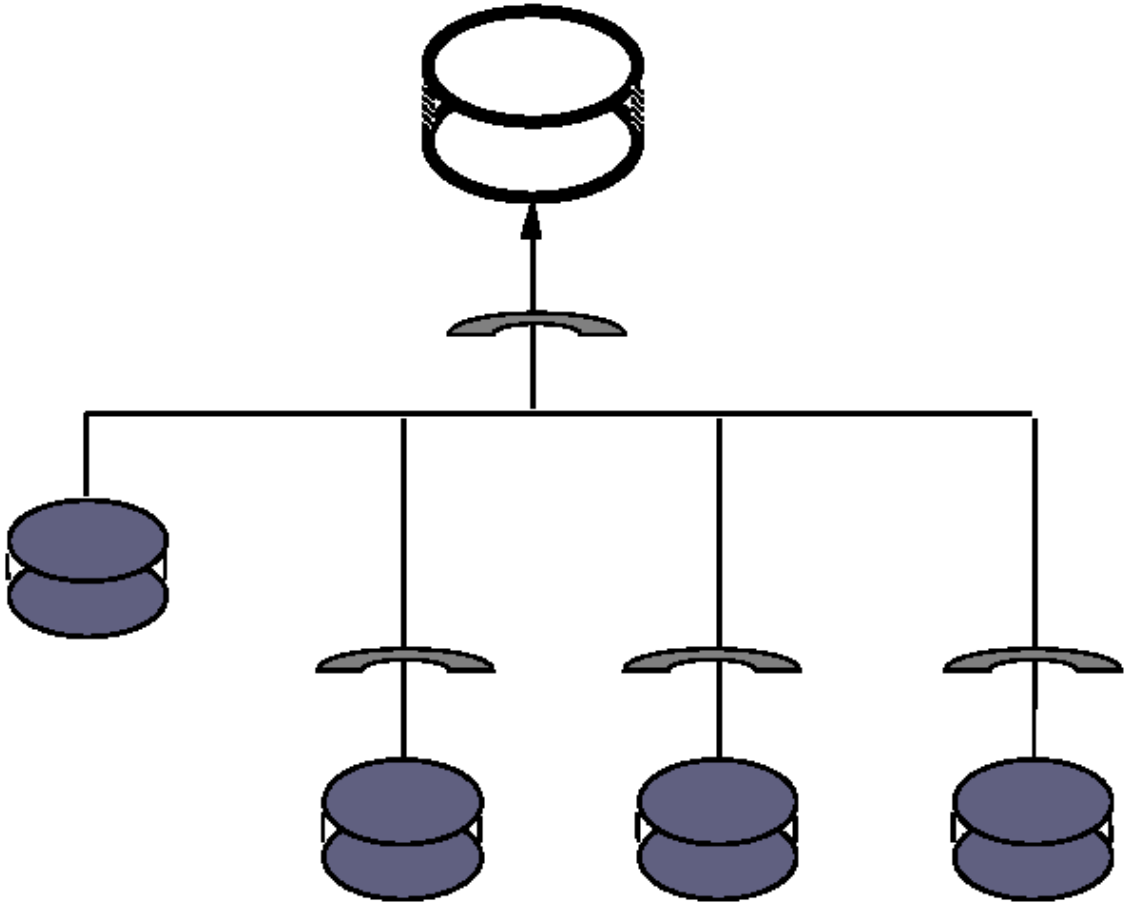


Repository Architecture

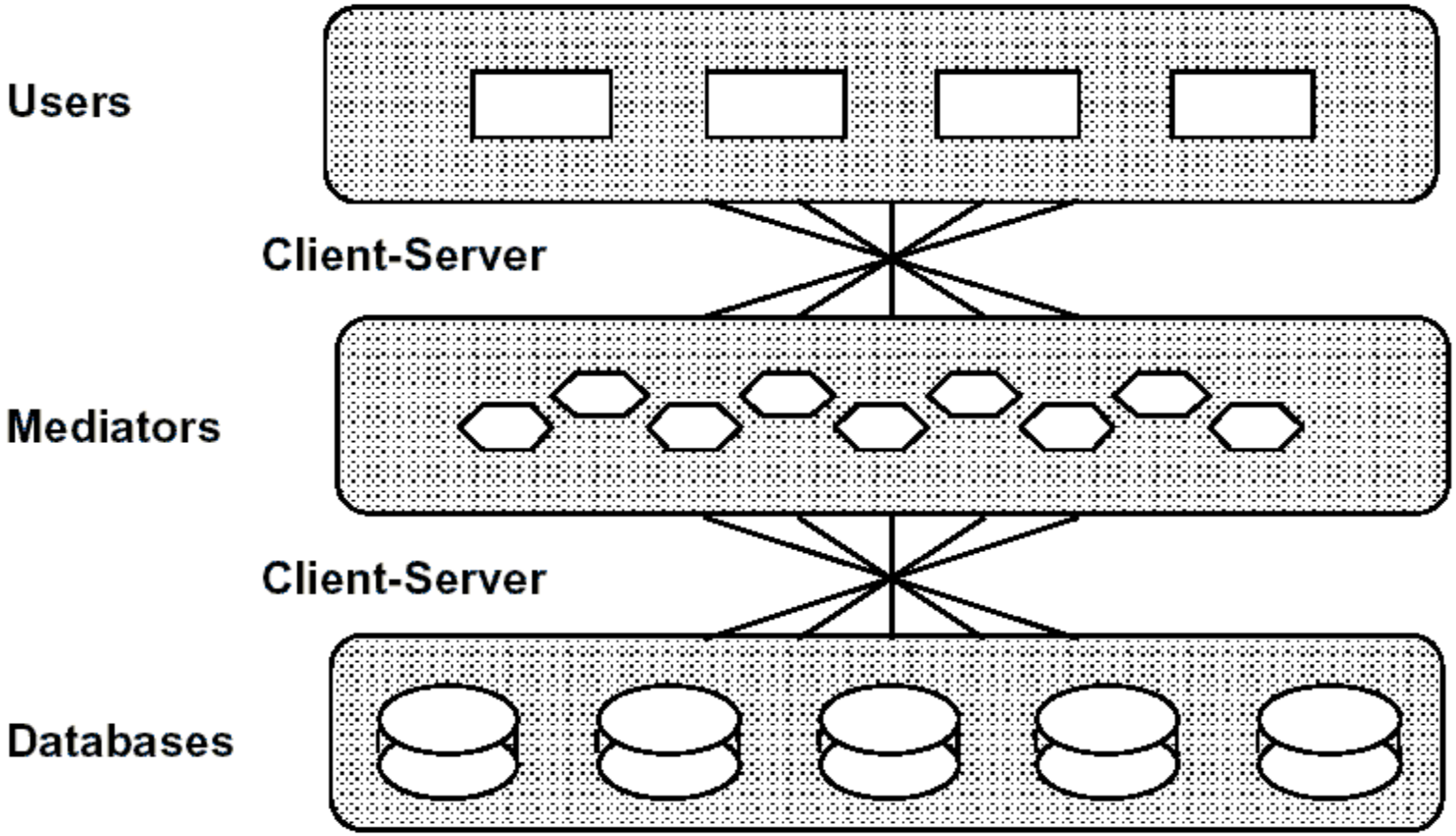


Unified Schemas for Integrating Database

- **Abstraction:**
  - **multiplex the databases; put filters on the query/update to match diverse views**
- 复合多个数据库；在查询/更新操作中增加过滤器来匹配不同的视图



Multi-Databases



Evolving Database Architecture

- **Batch processing:**
  - **Standalone programs** *（独立的程序）*
  - **results were passed from one to another on mag-tape** *（结果通过磁带从一个程序传到另一个程序）*
  - **batch sequential model** *（批处理风格）*
- **Interactive processing:**
  - **concurrent operation and faster updates preclude batching, so updates are out of synch with reports.** *（并行操作、更快的更新速度，但是使更新和报告难以保持同步。）*
  - **Repository model with external control** *（仓库风格）*

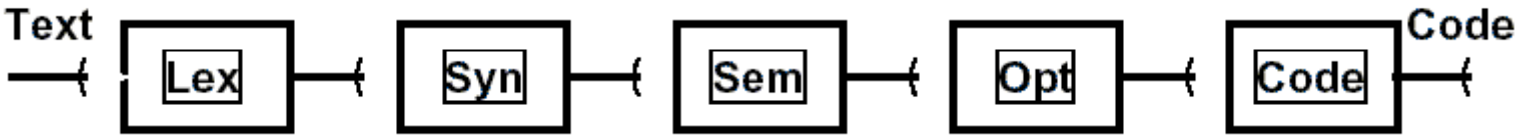
Computer-Aided Software Engineering

- **Initially just translation from source to object code: compiler, library, linker, make**（*起初，就是做从源代码到目标代码的转换*）
- **Grew to include design record, documentation, analysis, configuration control, incrementality**（*开始包含分析、设计、调试、测试、文档、配置管理、增量编译等功能*）
- **Integration demanded for 20 years, but not here yet.**

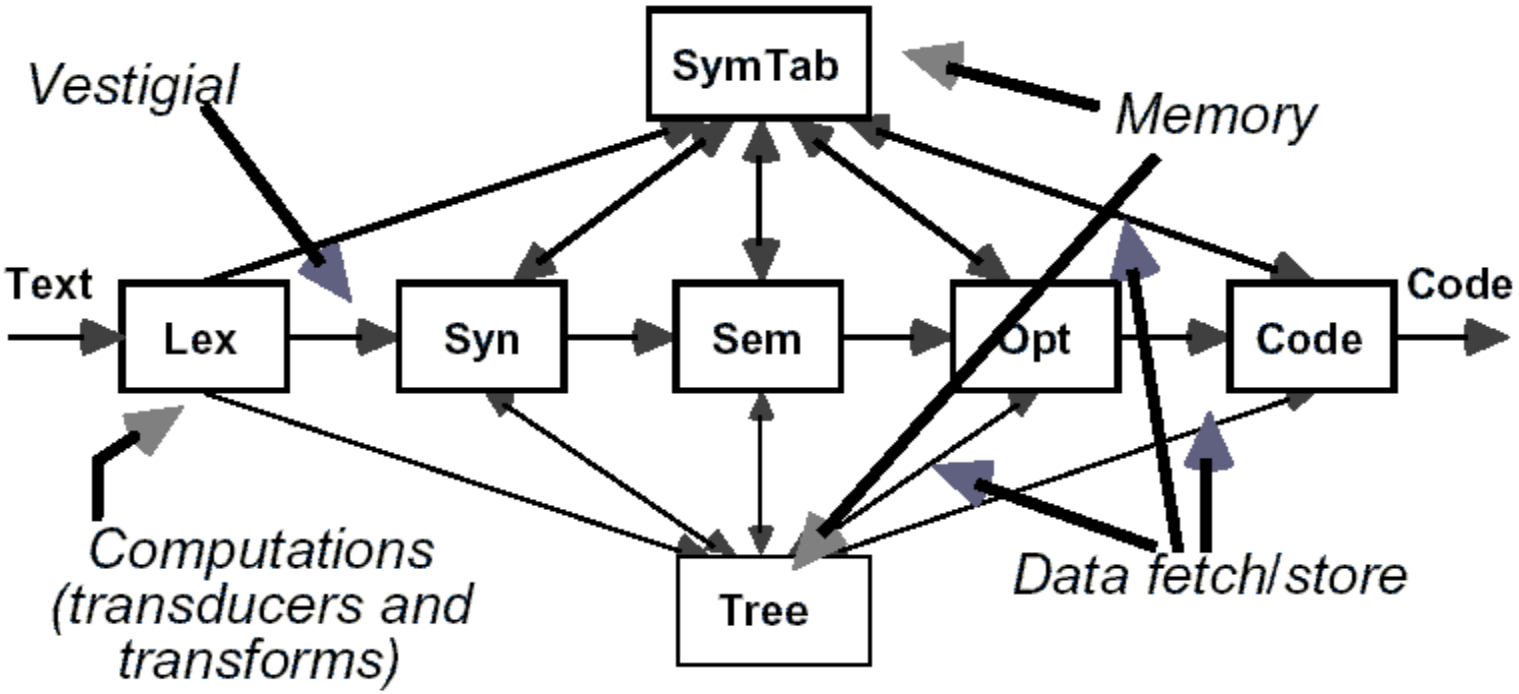
CASE vs DBMS

- As compared to databases, CASE has:
  - more types of data（更多的数据类型）
  - fewer instances of each type（更少的数据类型实例）
  - slower query rates（更慢的查询频率）
  - larger, more complex, less discrete information（更大，更复杂，更集中的信息）
  - but not shorter lifetime（生命周期没有更短）

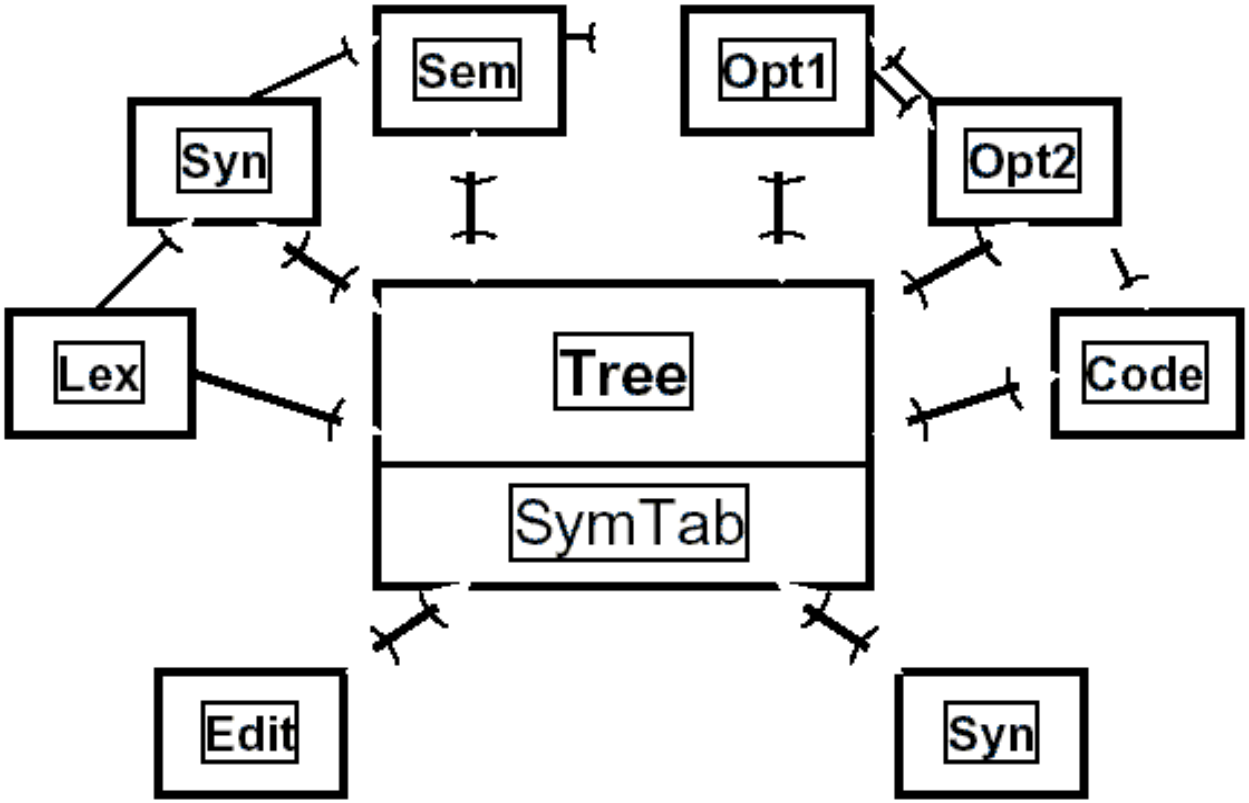
Traditional Compiler



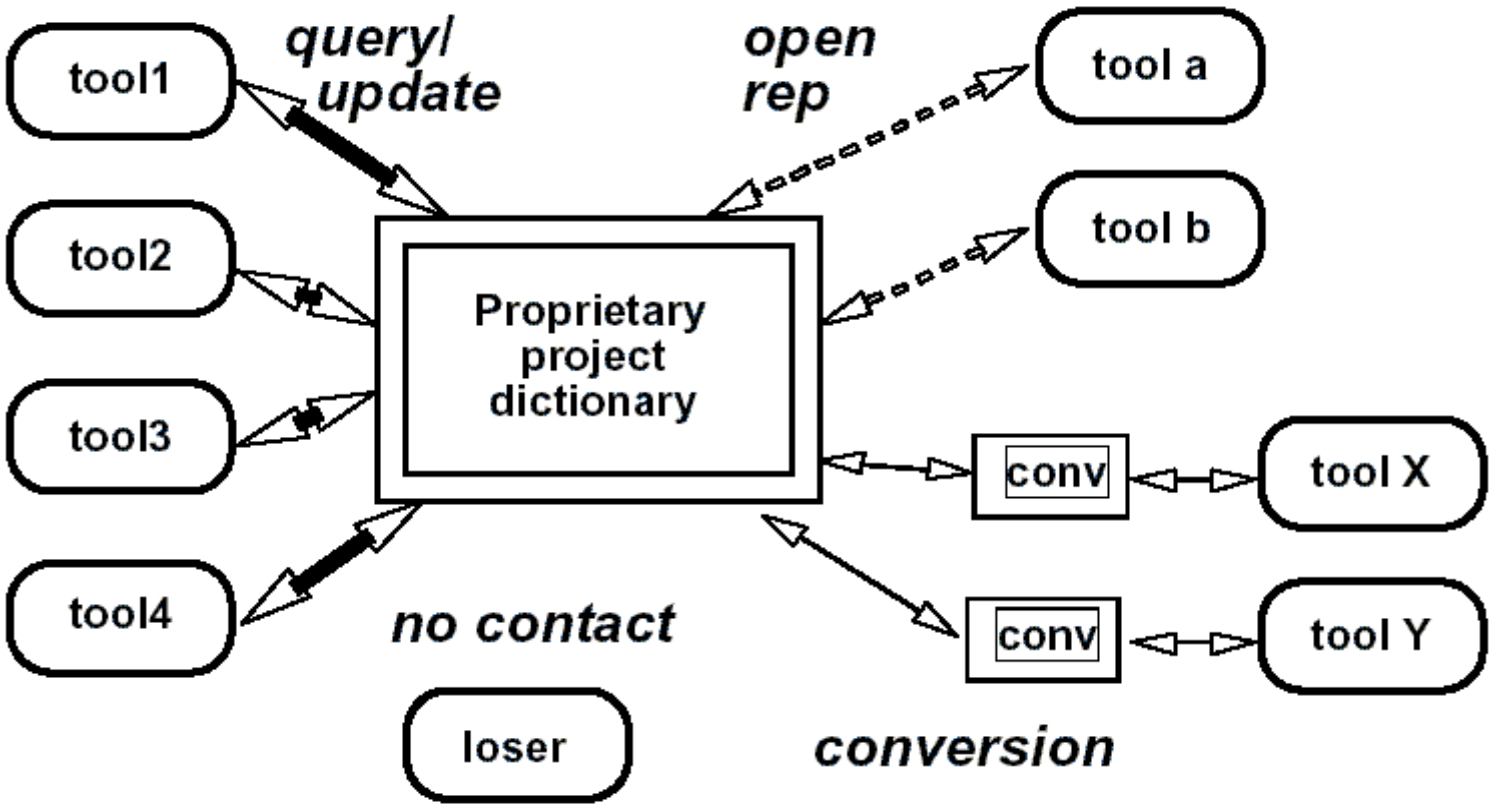
Example: Modern Canonical Compiler



Canonical Compiler



Software Tools with Shared Representation



Evolution of CASE Environments

- **Evolution is much like databases:**
  - **Interaction: batch --> interactive** *（交互：批处理→交互式）*
  - **Granularity: complete processing -->incremental** *（粒度：完全处理→增量）*
  - **Coverage: compilation --> full life cycle** *（覆盖：编译→全生命周期）*
  - **Like databases, started with batch sequential style** *（从批处理风格开始）*
  - **Integration needs led to repositories with rigid control, then to open systems in layers** *（对集成性的要求，促成了仓库风格被应用，让系统开始分层）*

Repository

- **Problem: This pattern is suitable for applications in which the central issue is establishing, augmenting, and maintaining a complex central body of information. Typically the information must be manipulated in a wide variety of ways. Often long-term persistence may also be required. Different variants support radically different control strategies.**
- **Context: Repositories often require considerable support, either an augmented runtime system (such as a database) or a framework or generator to process the data definitions.**

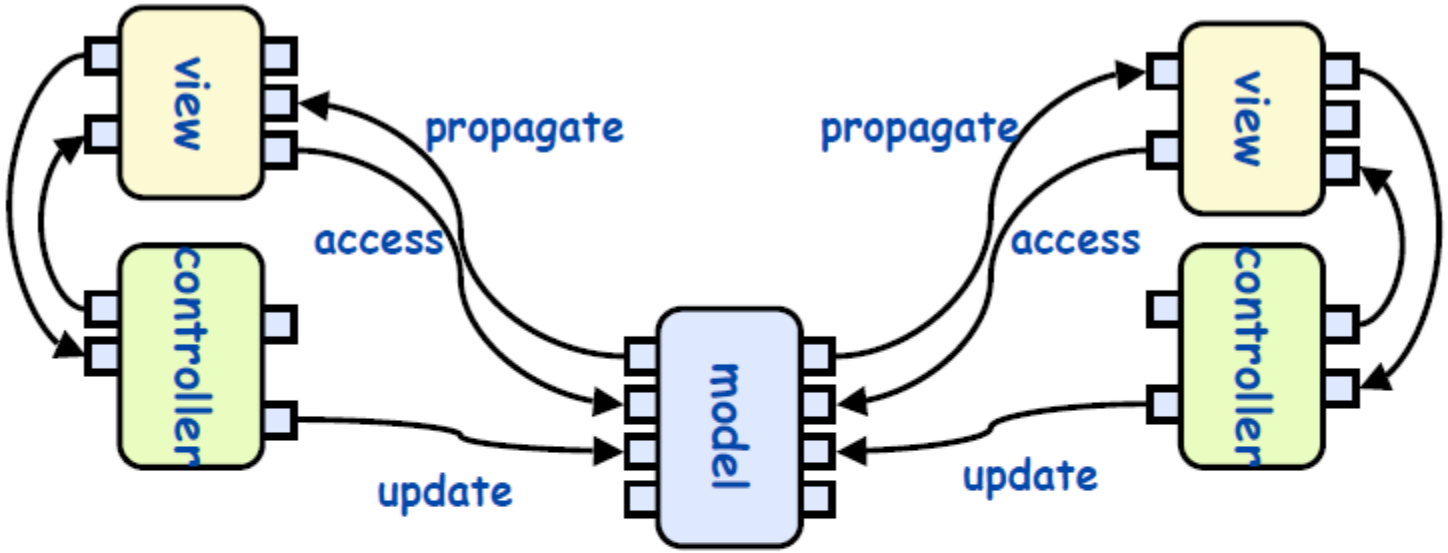
Repository

- **Solution:**
  - ***System model:*** **centralized data, usually richly structured**
  - ***Components:*** **one memory, many purely computational processes**
  - ***Connectors:*** **computational units interact with memory by direct data access or procedure call**
  - ***Control structure:*** **varies with type of repository; may be external (depends on input data stream, as for databases), predetermined (as for compilers), or internal (depends on the state of computation, as for blackboards)**

Varieties of Repositories

- **Discriminate on control strategy** *（在控制策略上的区别）*
  - **Predetermined by designer** *（设计者预先定义好）*
    - **Compilers**
  - **Driven by types of information in input stream** *（输入流的信息类型决定）*
    - **Database transaction system**
  - **Driven by availability of new information from other parts of the system** *（系统其他部分的新信息决定）*
    - **Scratchboard** *（刮板）*
  - **Opportunistic: driven by state of computation** *（机会主义：计算的状态决定）*
    - **Blackboard**

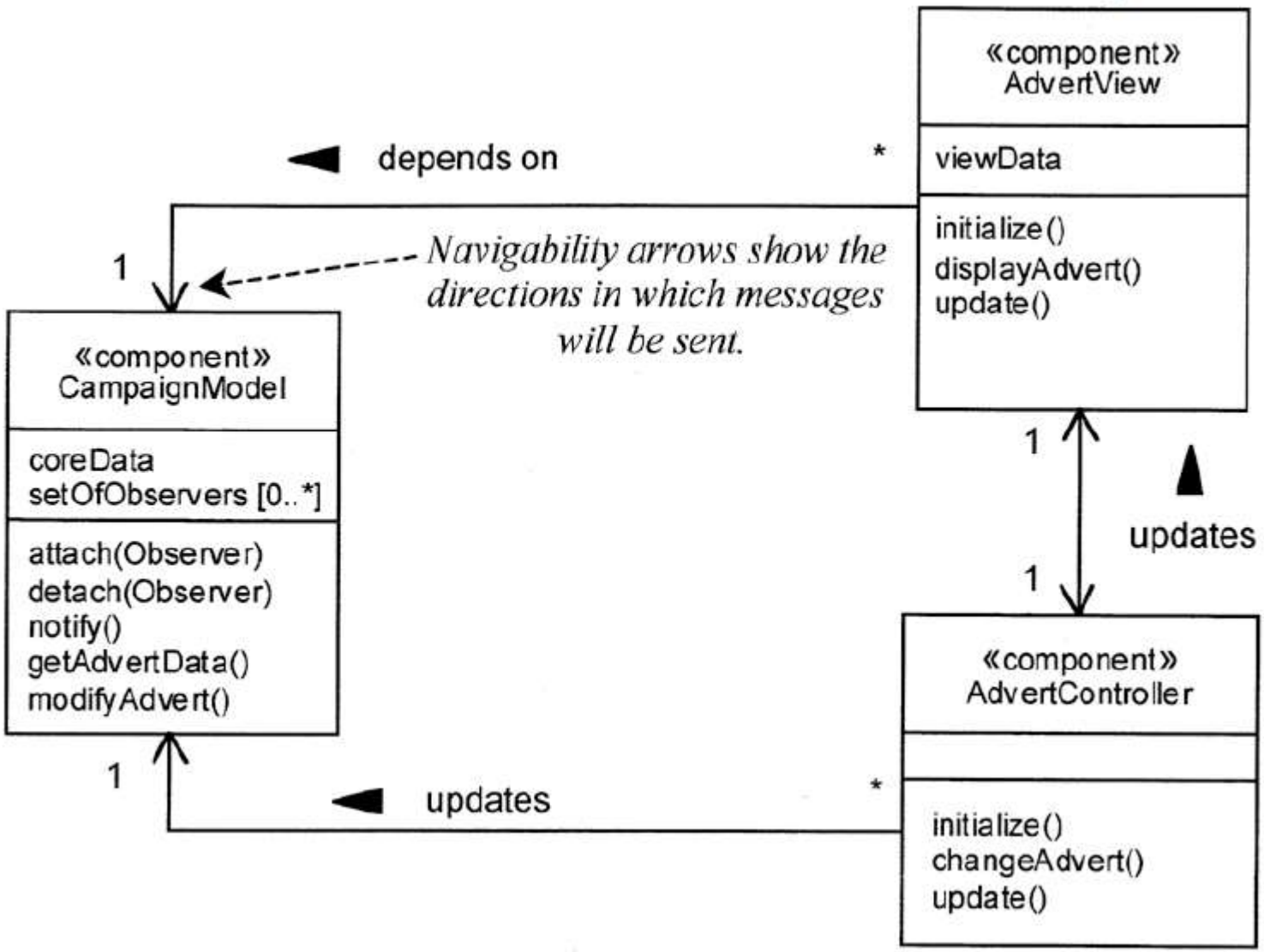
Variant: Model-View-Controller



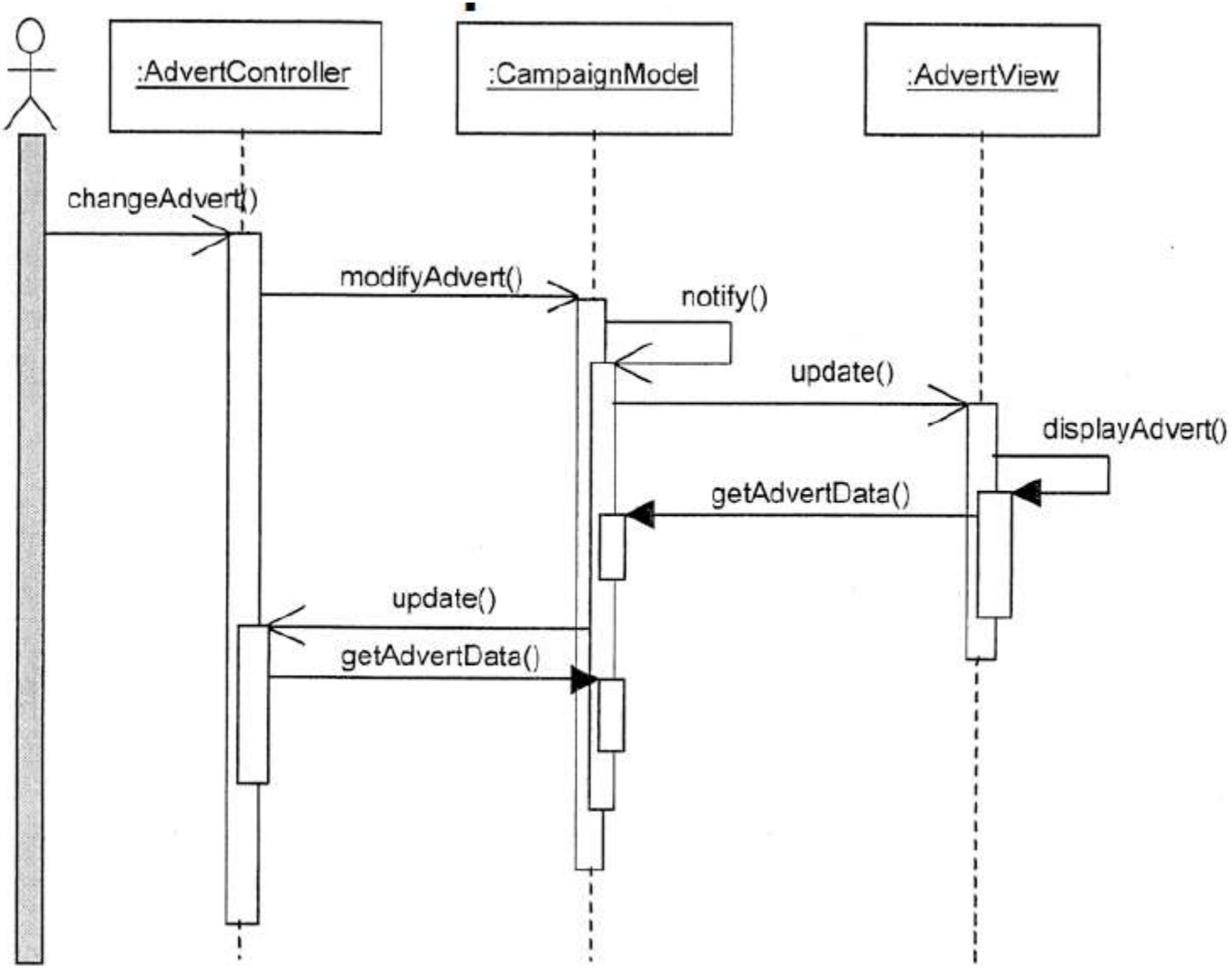
→ Properties

- One central model, many views (viewers)
- Each view has an associated controller
- The controller handles updates from the user of the view
- Changes to the model are propagated to all the views

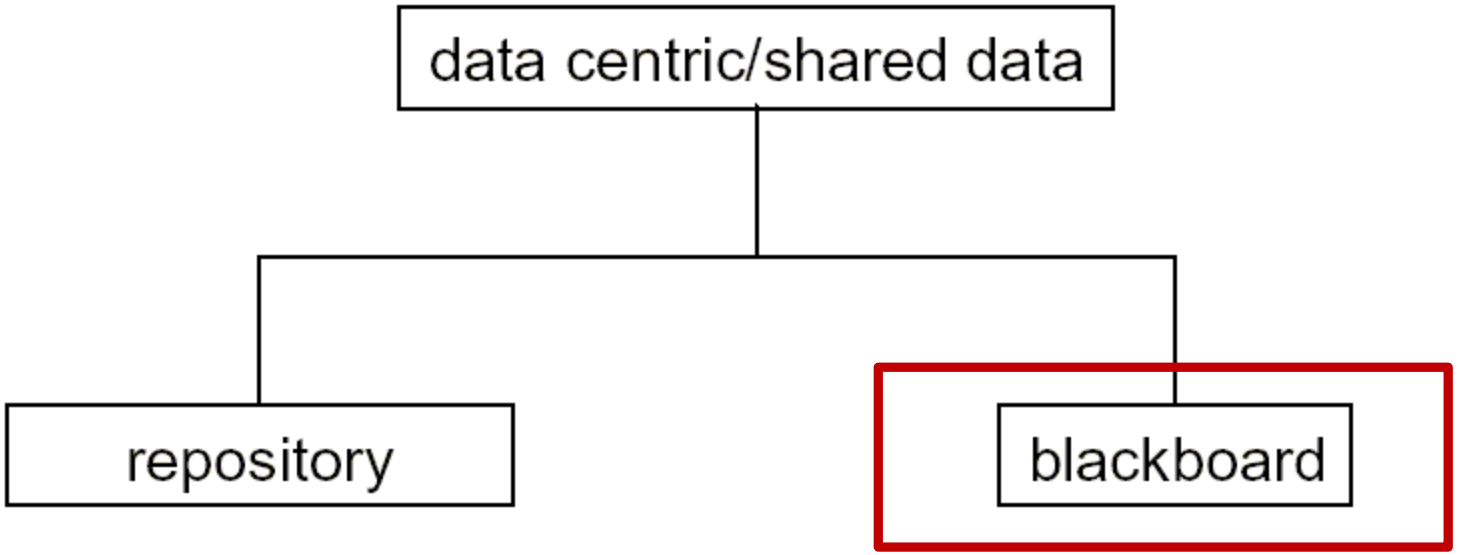
Variant: Model-View-Controller



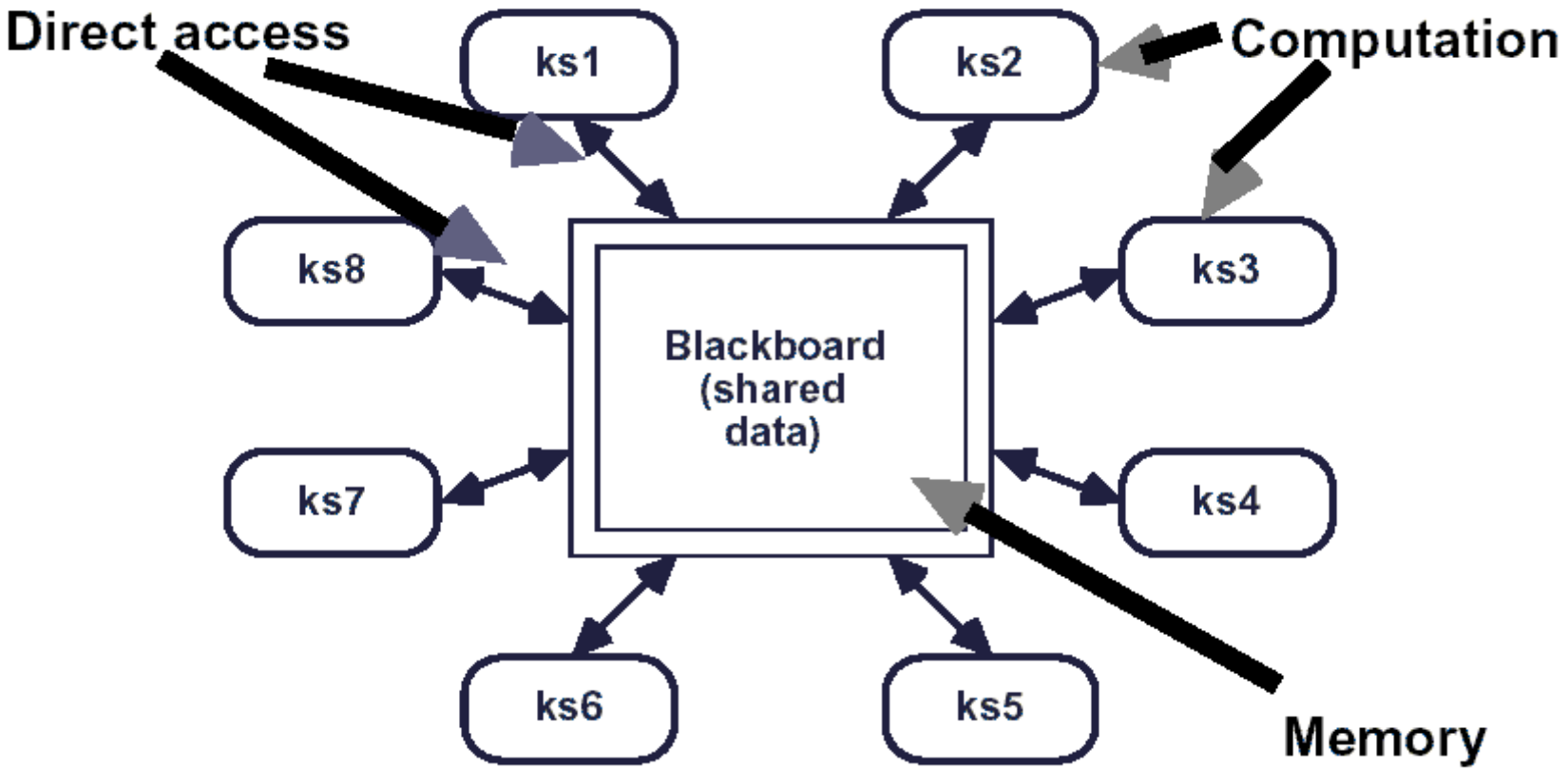
Variant: Model-View-Controller



Data Centered/Shared Data



Blackboard Style



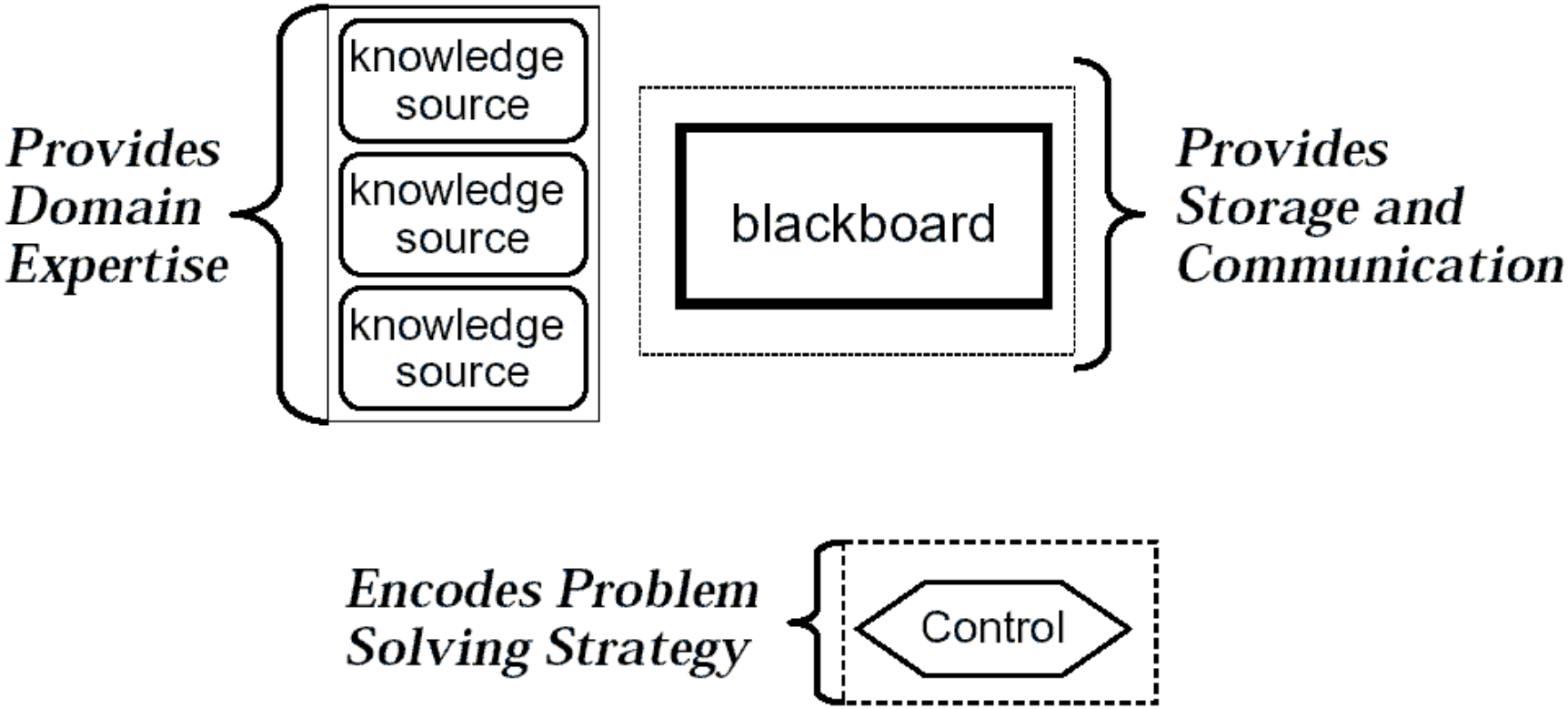
Blackboard Style

- 一个典型的黑板型数据共享系统包括以下三个部分：
  - 知识源：知识源中包含独立的、与应用程序相关的知识，知识源之间不直接进行通讯，它们之间的交互只通过黑板来完成。
  - 黑板数据结构：黑板数据是按照与应用程序相关的层次来组织的解决问题的数据，知识源通过不断地改变黑板数据来解决问题。
  - 控制：控制完全由黑板的状态驱动，黑板状态的改变决定使用的特定知识。
- 黑板模式对于无确定性求解策略的问题比较有用，在专家系统中，这种模式应用的比较广泛。

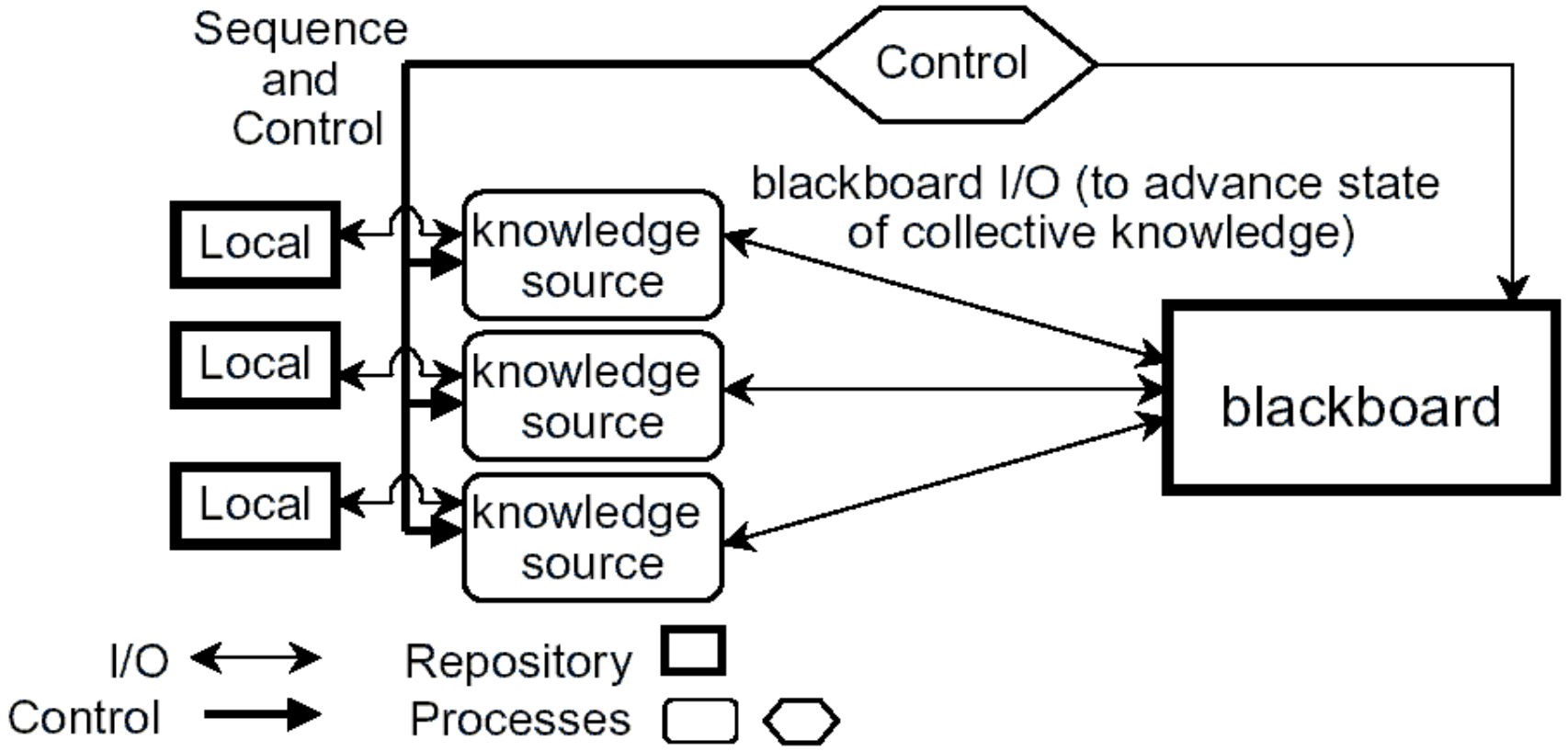
Blackboard Model

- **Historical examples:**
  - **Hearsay I, Hearsay II, HASP/SAIP, CRYSALIS, ATOME**
  - **Originally most were from signal processing, problem solving (planning, logistics, diagnostics)（信号处理、专家系统、模式识别领域经常采用）**

Blackboard Model



Blackboard Architecture



Knowledge Sources

- **Objective:**
  - **contribute knowledge that leads to solution** *（提供解决问题的知识）*
- **Representation:**
  - **procedures, sets of rules, logic assertions** *（过程、规则、逻辑断言）*
- **Action:**
  - **modify only the blackboard (or control data -- magic)** *（只修改黑板）*
- **Responsibility:**
  - **know when it's possible to help** *（知道何时能发挥作用）*
- **Selection:**
  - **loosely-coupled subtasks, or areas of specialization** *（低耦合的子任务，或者有特别的能力）*

Blackboard Data Structure

- **Objective:**
  - **hold data for use by knowledge sources** *（保存知识源要使用的数据）*
- **Representation:**
  - **stores objects from solution space, including** *（保存来自解空间的数据，包括）*
    - **input data, partial solutions, alternatives, final solutions, control data**
    - **objects and properties define the terms of the discourse**
    - **relationships are denoted by named links (“next-to”, “part-of”)**
- **Organization:**
  - **hierarchical, possibly multiple hierarchies; links between objects on same or different levels** *（分层；链接同层或不同层的对象）*

Control

- **Objective:**
  - **make knowledge sources respond opportunistically** *（让知识源响应偶然事件）*
- **Representation:**
  - **keeps various sorts of information about which knowledge sources could operate and picks a sequence that allows the solution process to proceed a step at a time** *（了解各个知识源的能力，决策解决问题的步骤）*
- **Remark:**
  - **the control mechanisms are thoroughly ad hoc** *（控制机制是彻底的与时俱进）*

Blackboard Problem Characteristics

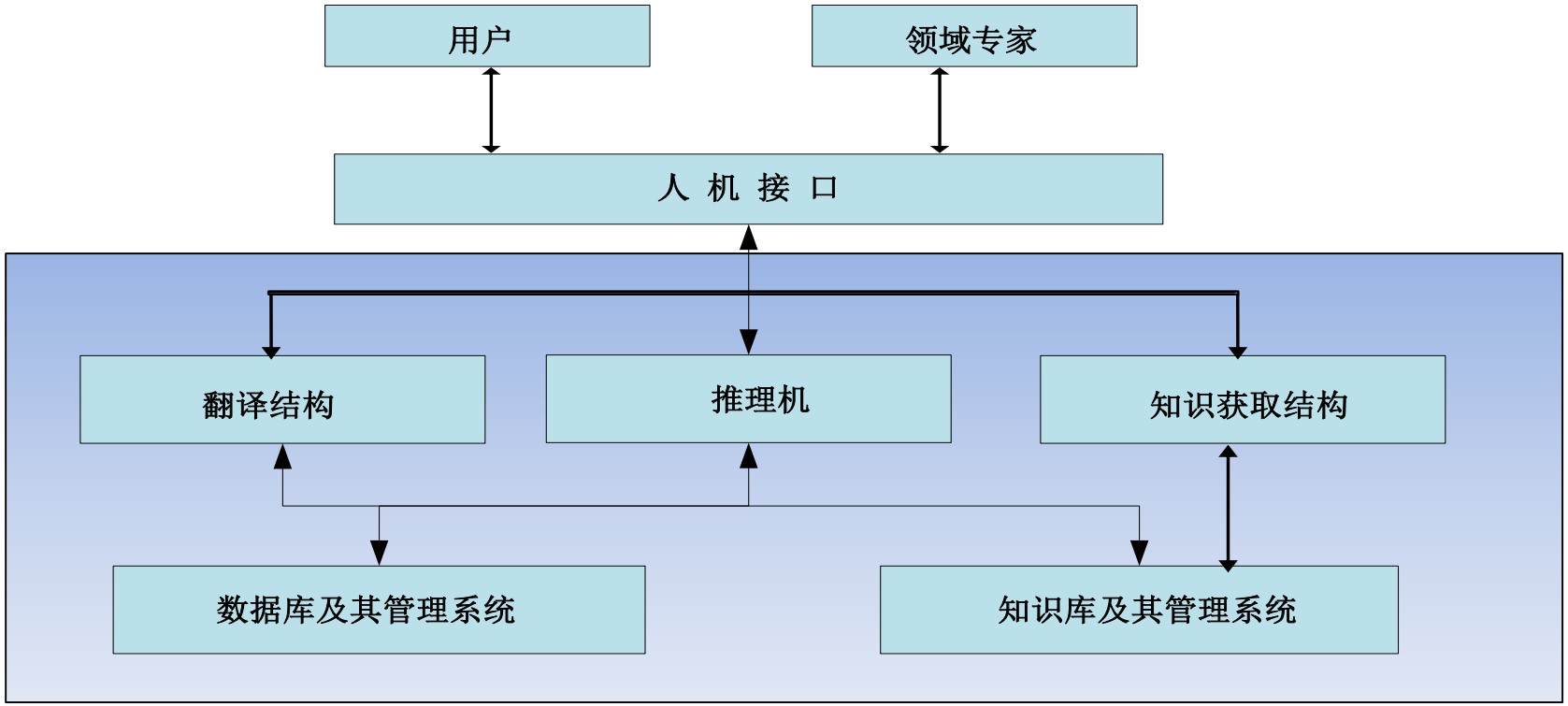
- no direct algorithmic solution（没有直接的算法可解）
  - multiple approaches to solving the problem（多种方法都可能解决问题）
  - various domain expertise required to solve the problem（需要多个领域的专门知识协作解决）
- uncertainty
  - error and variability in data and solution（数据和解决方法可能错误或变化）
  - moderate to low “signal-to-noise-ratio” in data（数据中信噪比的变化）
  - Uncertainty interferes with algorithmic solutions（算法接口的变化）
- Best-effort” or approximation is good enough
  - no single discrete answer to problem, or “right” answer may vary（问题没有唯一的解答，或者“正确”答案会变化）

Example

- 专家系统
  - 专家系统实质就是一组程序。
  - 从功能上：可定义为“一个在某领域具有专家水平解题能力的程序系统”，能像领域专家一样工作，运用专家积累的工作经验与专门知识，在很短时间内对问题得出高水平的解答。
  - 从结构上讲：可定义为“由一个专门领域的知识库，以及一个能获取和运用知识的机构构成的解题程序系统”。

Example

- 专家系统的一般结构



黑板风格的优点

- 解决问题的多方法性：
  - 对于一个专家系统，针对于要解决的问题，如果在其领域中没有独立的方法存在，而且对解空间的完全搜索也是不可行的，在黑板模式中可以用多种不同的算法来进行试验，并且也允许用不同的控制方法。
- 具有可更改性和可维护性：
  - 因为在黑板模式中每个知识源是独立的，彼此之间的通信通过黑板来完成，所以这使整个系统更具有可更改性和可维护性。
- 有可重用的知识源：
  - 由于每个知识源在黑板系统中都是独立的，如果知识源和所基于的黑板系统有理解相同的协议和数据，我们就可以重用知识源。
- 支持容错性和健壮性：
  - 在黑板模式中所有的结果都是假设的，并且只有那些被数据和其它假设强烈支持的才能够生存。这对于噪声数据和不确定的结论有很强的容错性。

黑板风格的不足

- 测试困难：
  - 由于黑板模式的系统有中央数据构件来描述系统的体现系统的状态，所以系统的执行没有确定的顺序，其结果的可再现性比较差，难于测试。
- 不能保证有好的求解方案：
  - 一个黑板模式的系统所提供给我们的往往是所解决问题的一个百分比，而不是最佳的解决方案。
- 效率低：
  - 黑板模式的系统在拒绝错误假设的时候要承受多余的计算开销，所以导致效率比较低。
- 开发成本高：
  - 绝大部分黑板模式的系统需要用几年的时间来进化，所以开发成本较高。
- 缺少对并行机的支持：
  - 黑板模式要求黑板上的中心数据同步并发访问，所以缺少对不并行机的支持。

Next Class

- **Architecture Styles**
  - **Virtual Machine**
  - **Independent Component**