# Architecture of Enterprise Applications III Architectural Styles

**Haopeng Chen**

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

# Architecture and Program



| *Architecture* | *Program* |
| --- | --- |
| interactions among parts | implementations of parts |
| structural properties | computational properties |
| declarative | operational |
| mostly static | mostly dynamic |
| system-level performance | algorithmic performance |
| outside module boundary | inside module boundary |
| composition of subsystems | copy code or call libraries |

# Architecture Modeling



- Clarify Design Intent
  - Intended architecture is often lost. It's mostly informal, it's hard to communicate anyhow.
- Provide Analysis for Design
  - Engineering design entails performance prediction and design tuning. Routine practice.
- Improve maintainability
  - Over half of maintenance effort goes into figuring out just what's there.
- Answer Difficult Questions
  - Even without formal methods, explicit architectural modeling can uncover fuzzy requirements, thinking, and design approaches

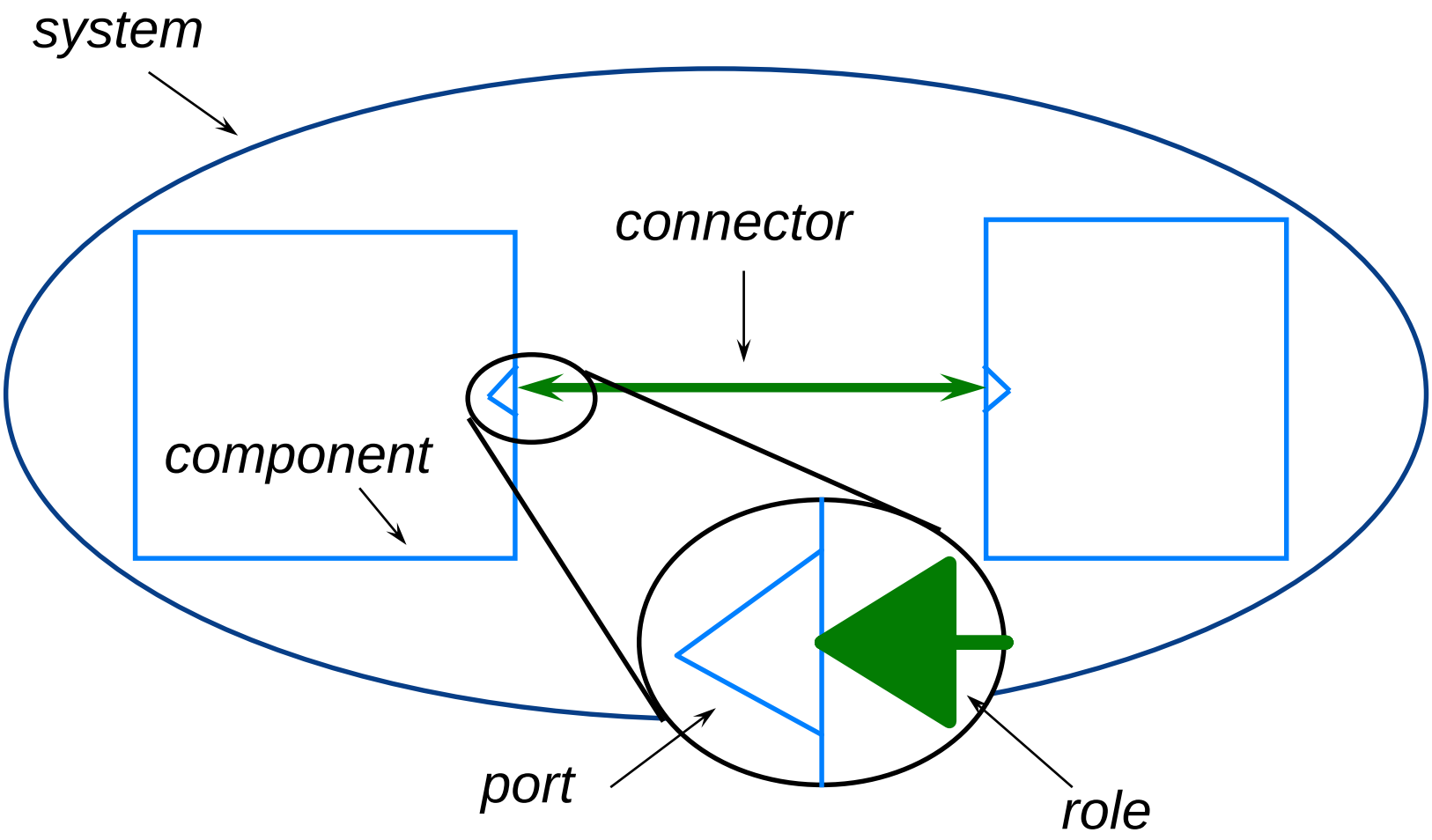
# Description of Architecture



- Architecture includes:
  - Components:
    - e.g.: filters, databases, objects, clients/servers
  - Connectors:
    - e.g.: procedure call, pipes, event broadcast
  - Properties:
    - e.g.: signatures, pre/post conds, RT specs

# Vocabulary of Architecture





# Example: Key Word In Context



Problem:

> Any line may be ‘circularly shifted’ by repeatedly removing the first word and appending it at the end of the line.
>
> The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order."

On the Criteria for Decomposing Systems into Modules. David Parnas. *CACM*, 1972

# KWIC: Key Word In Context



- Inputs: Sequence of lines

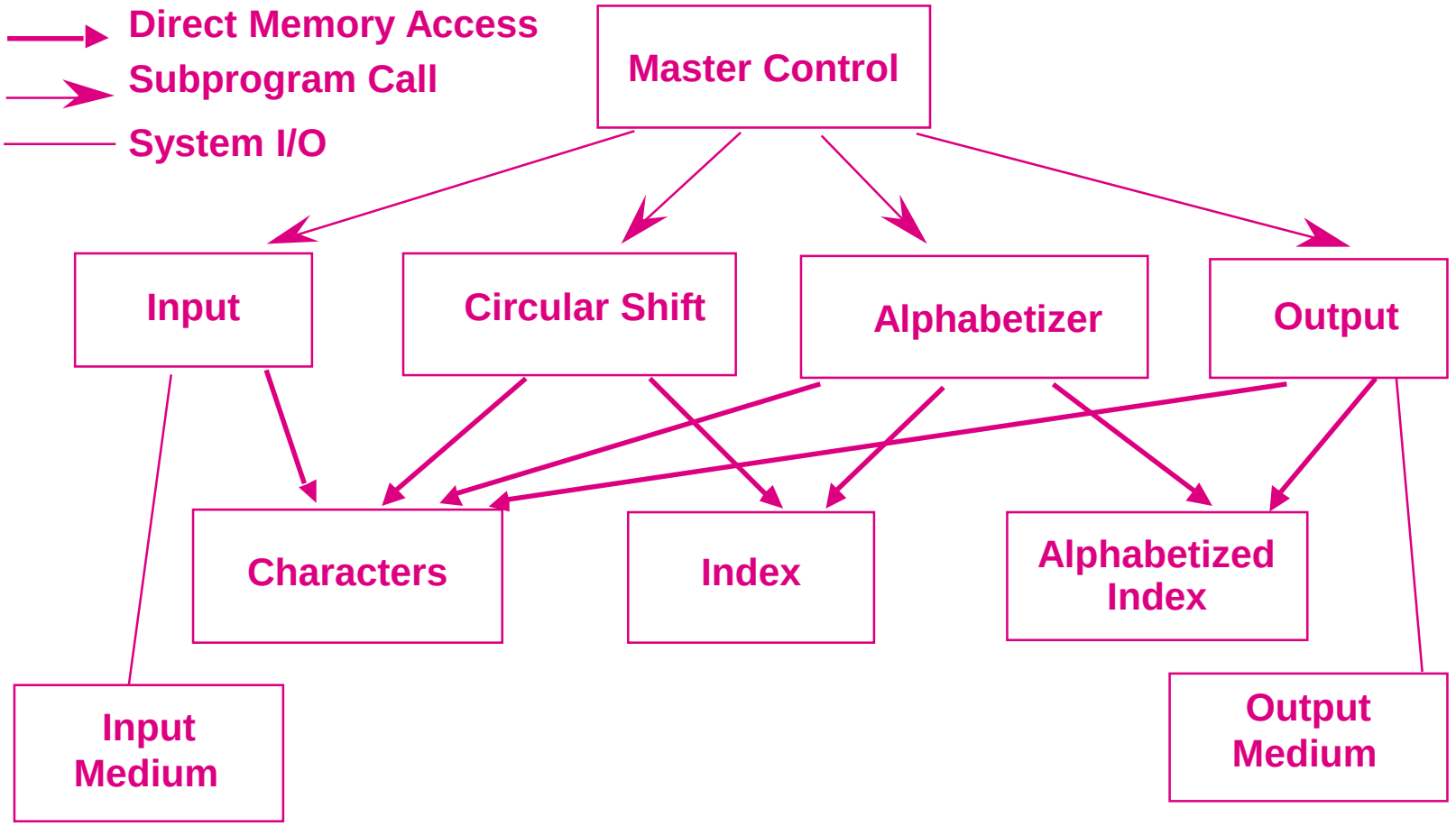
```
Pipes and Filters
Architectures for Software Systems
```

- Outputs: Sequence of lines, circularly shifted and alphabetized

```
and Filters Pipes
Architectures for Software Systems
Filters Pipes and
for Software Systems Architectures
Pipes and Filters
Software Systems Architectures for
Systems Architectures for Software
```

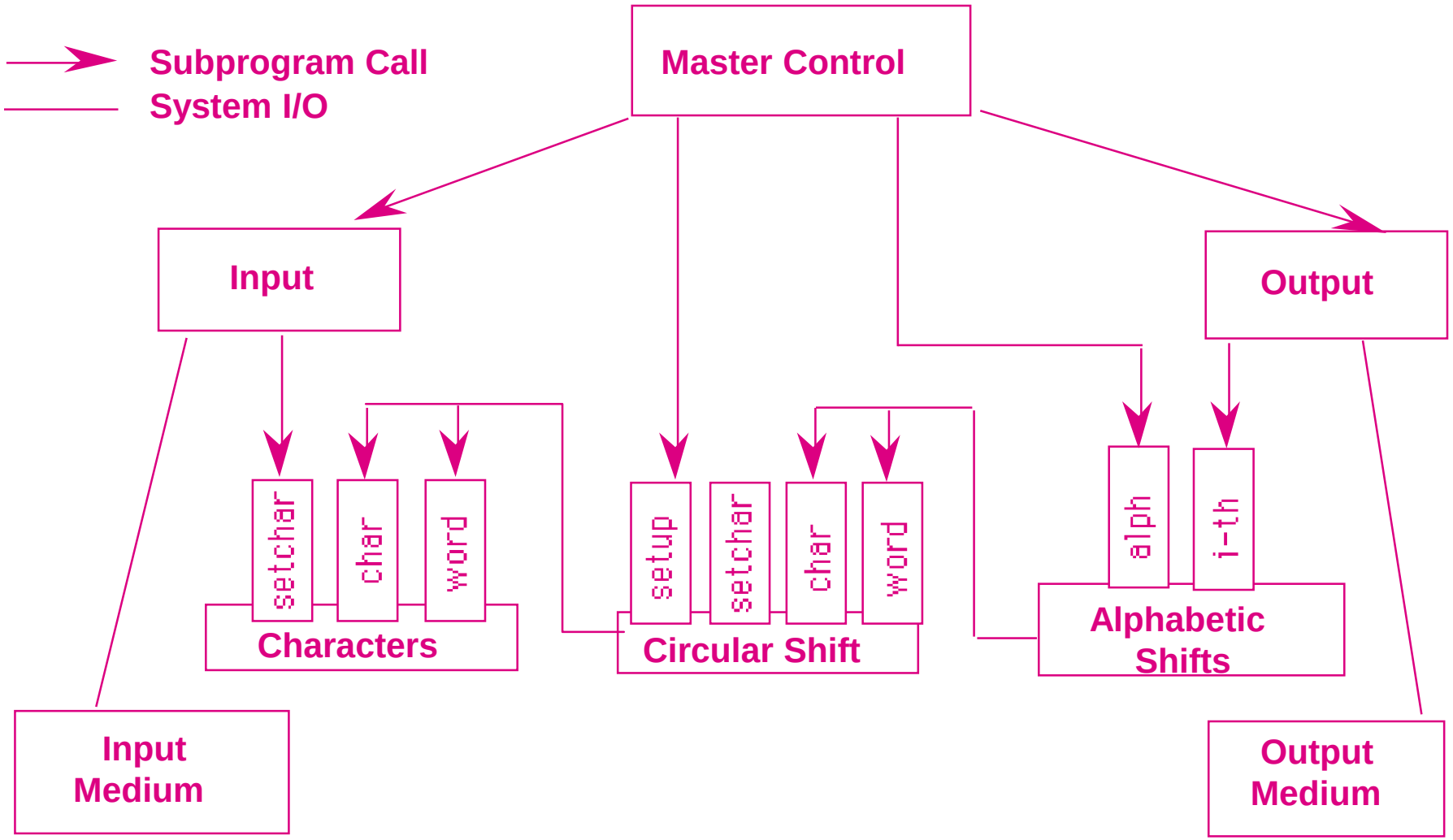
# KWIC: Shared Memory Architecture





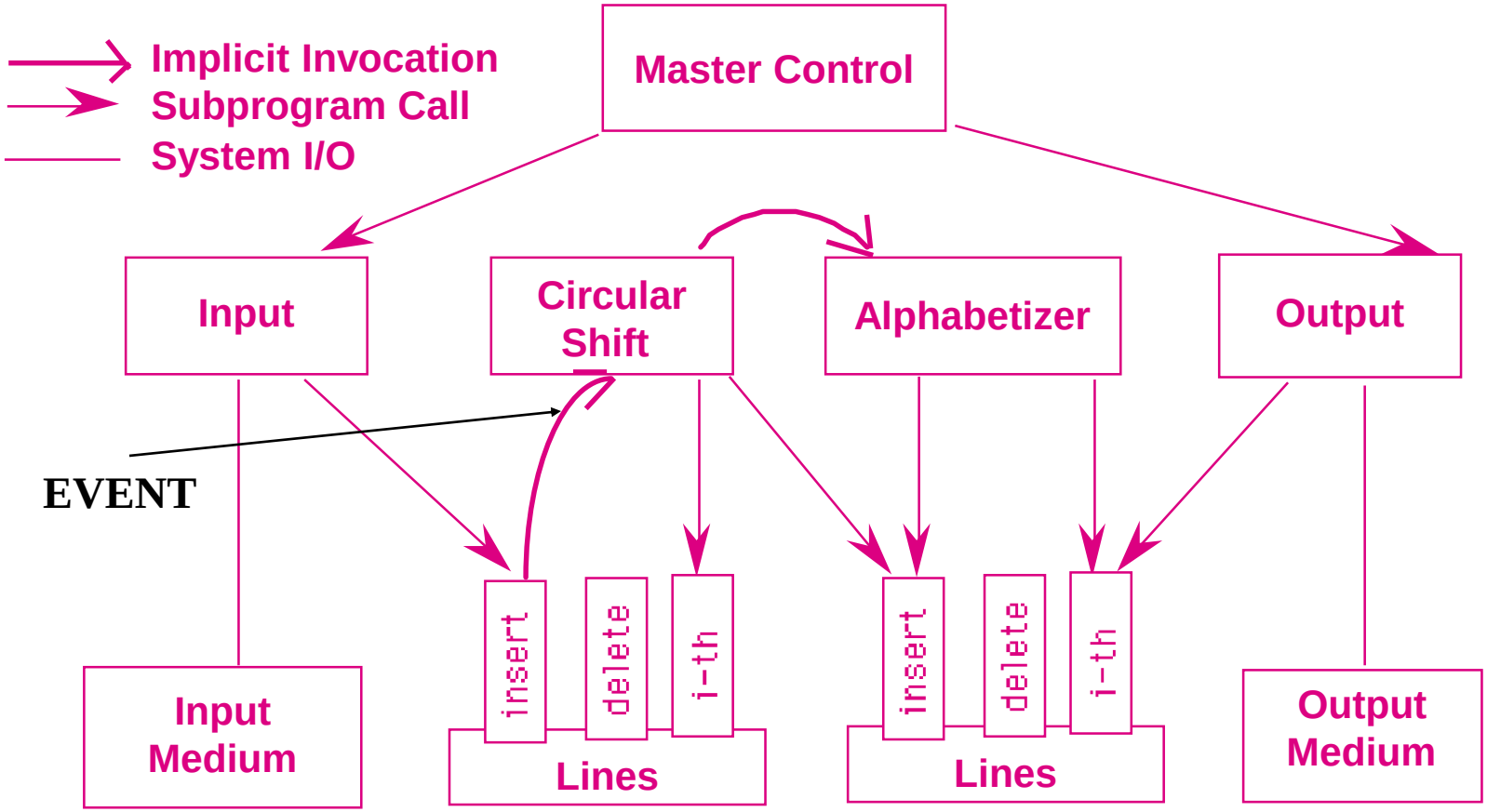
# KWIC: ADT Architecture





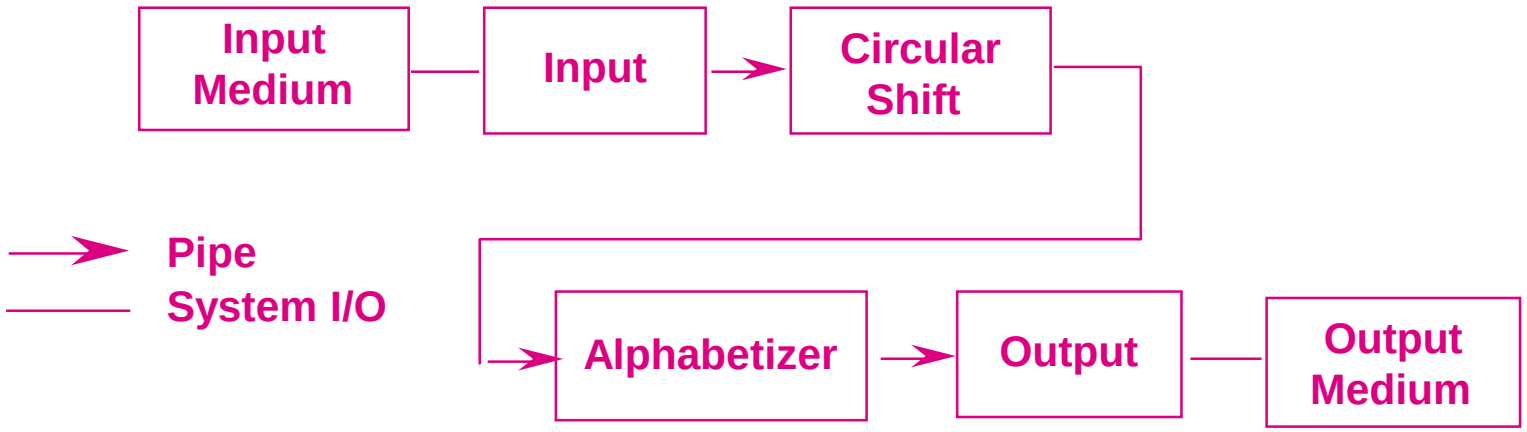
# KWIC: Event Architecture





# KWIC: Data Flow Architecture





# Design Consideration



- Change in Algorithm
  - Batch vs Increment
- Change in Data Representation
  - The number of characters in one line
- Change in Function
  - Omit the incorrect input
- Performance
  - Time and Space
- Reuse
  - Alphabetizer

# KWIC Comparisons

REin
REliable, INtelligent & Scalable Systems

| | Shared Memory | ADT | Events | Dataflow |
|---|---|---|---|---|
| Change in Algorithm | -- | -- | + | + |
| Change in Data Rep. | -- | + | -- | -- |
| Change in Function | + | -- | + | + |
| Performance | + | + | -- | -- |
| Reuse | -- | + | -- | + |

# Architectural Styles



- a. What are Architectural Styles?
  - An architectural style
    - defines a family of systems
    - in terms of a pattern of structural organization
    - provides a vocabulary of components and connector types
    - a set of constraints on how they can be combined
    - a semantic model may also exist which specify how to determine a system's overall properties from the properties of its parts

# Architectural Styles



| 类别 | 架构风格 |
| --- | --- |
| 通用结构 | 分层、管道与过滤器、黑板 |
| 分布式系统 | 客户端-服务器、三层架构、代理 |
| 交互式系统 | 模型-视图-控制器、表示-抽象-控制 |
| 自适应系统 | 微内核、反射 |
| 其他 | 批处理、解释器、进程控制、基于规则 |

# Heterogeneous Architectures



- The combination of several styles.
  - Components of a hierarchical system may have an internal structure developed using a different method.
  - Connectors may also be decomposed into other systems (e.g. pipes can be implemented internally as FIFO queues).
  - A single component may also use a mixture of architectural connectors.

- Example- Unix pipes-and-filter system
    - File system acts as the repository
    - Receives control through initialization switches
    - Interacts with other components through pipes.

# 通用结构 – 分层架构



- 分层架构风格针对大型系统的，将其抽象为不同的层次，从而提供了一种进行系统分解的模式。分层架构的结构特点是：
  - 每一层都提供了特定的设施，使其向高层暴露接口以提供服务，并对高层屏蔽低层；
  - 每一层都提供了与其他层有明确区分的功能，高层就像是依赖于低层运行的虚拟机，而低层并不依赖于高层，最底层就是硬件系统。

# 通用结构 – 分层架构



- 分层架构的优点包括：
  - 良好的可修改性
  - 良好的复用性
- 分层架构的缺点包括：
  - 有损于性能
  - 用户代码的控制力会被削弱
  - 分层质量受抽象的影响

# 通用结构 – 分层架构



- 在RUBiS系统中，就采用了分层架构，从底层到高层包括以下层：
  - 持久性层： RUBiS采用了关系型数据库来存储持久性状态，并通过JDBC来访问这些数据；
  - 领域层：使用实体Bean作为对象-关系映射工具，在应用服务器中创建领域层，实现对数据库的持久性操作；
  - 服务层：使用会话Bean和消息驱动Bean实现业务逻辑，包括竞价、出售、评论等功能；
  - 应用层：使用Servlet来控制用户请求的转发逻辑，从而实现完整的业务流程；
  - 表示层：实现页面显示逻辑，RUBiS的不同版本分别使用了JSP、PHP和Servlet等技术来实现这一层；
  - 客户端层：这一层是为了模拟客户端访问而设计的，它是用Java编写的应用程序。

# 通用结构 – 管道与过滤器



- 管道与过滤器架构是将软件系统分解成为若干个过滤器和管道，过滤器的作用是对其输入数据进行处理，并产生输出数据，而管道的作用是将过滤器连接在一起。管道和过滤器的结构特点如下：
  - 当过滤器对输入数据进行处理时，不会保留任何历史信息，即在对不同的数据进行处理时，不会保留任何状态信息。数据经历一系列过滤器的处理过程就是对数据进行增量式转变的过程。
  - 管道将过滤器连接在一起，构成了系统的通信链路，数据通过管道在过滤器之间传递。管道本身并不对数据进行任何转换处理。

# 通用结构 – 管道与过滤器



- 管道与过滤器架构的优点包括：
  - 有利于过程式的系统功能分解
  - 系统易于扩展和复用
  - 存在有利于性能的因素
- 管道与过滤器架构的缺点包括：
  - 交互式程序难以分解
  - 数据流格式有可能会降低系统性能
  - 存在有害于性能的因素

# 通用结构 – 管道与过滤器



- Linux的Shell程序可以看做是典型的管道与过滤器架构的例子，例如下面的Shell脚本：
  - $cat *TestResults* | sort | grep *Good*

会将TestResults文件的文本进行排序，然后找出其中包含单词Good的行，并显出在屏幕上。Shell命令cat、sort和grep依次执行，就构成了一个管道-过滤器架构。

# 通用结构 – 黑板架构

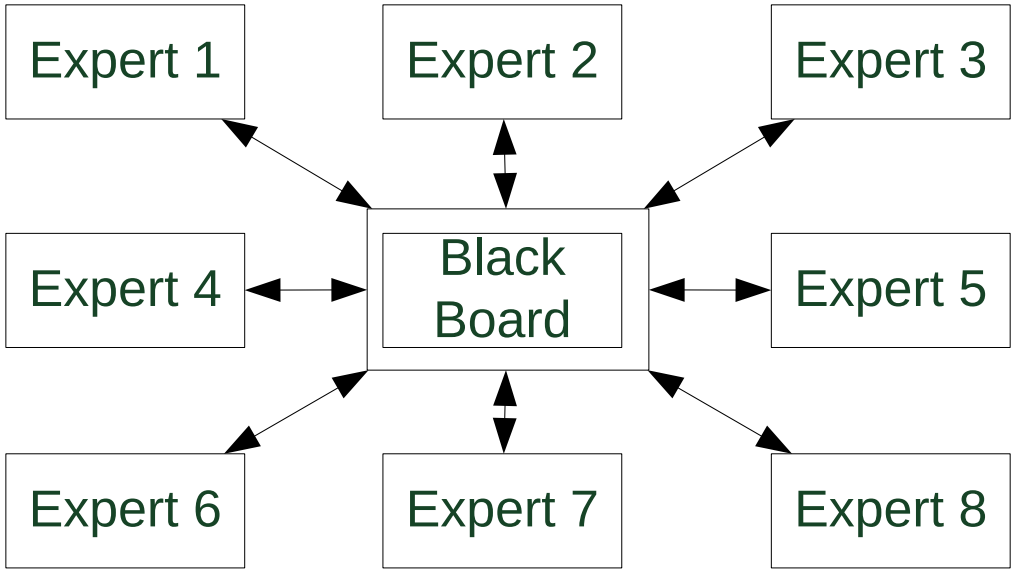


- 黑板架构为参与问题解决的知识源提供了共享的数据表示，这些数据表示是与应用相关的。在黑板架构中，控制流是由黑板数据的状态决定的，而并非按照某个固定的顺序执行。黑板架构的结构包括如下部分：
  - 知识源：是指彼此分离独立的与应用相关的知识包，知识源之间的交互都是通过黑板来完成的，它们彼此并不直接交互。
  - 黑板数据结构：按照应用相关的层次结构组织而成的问题解决过程中的状态数据。知识源会更新黑板数据，从而增量式地解决问题。
  - 控制流：完全有黑板状态驱动，知识源会在黑板数据更新时根据其状态做出相应的响应。

# 通用结构 – 黑板架构



- 黑板架构专门针对没有确定的解决方法的问题，例如信号处理和模式识别，它通过多个知识源的协作来解决问题，而这种协作完全是状态驱动的，因此各个知识源具有公平的机会获取并更新黑板中的状态数据。



# 分布式系统架构 – 客户端/服务器架构



- 客户端-服务器架构风格将软件系统分成了两层，客户端和服务器端分别运行于独立的进程中，通过远程方法调用来沟通。客户端-服务器架构的结构特点是：
  - 客户端通常运行的是应用程序，这种富客户端了解服务器程序标识，并可以通过多种网络通信协议以远程方法调用的方式与服务器进行通信；
  - 服务器可以支持客户端程序的并发访问，并提供安全、事务和消息等控制机制。通常服务器端无法预知在运行时访问它的客户端的数量和标识。

# 分布式系统架构 – 客户端/服务器架构



- 客户端-服务器架构的优点包括：
  - 有效降低服务器端的负荷
  - 能够提供复杂的用户界面
  - 能够传输和缓存大量的数据
- 客户端-服务器架构的缺点包括：
  - 软件升级代价大
  - 服务器端的控制减弱

# 分布式系统架构 – 三层架构



- 三层架构风格是在客户端和服务器端的数据库之间加入了一个中间层，有关业务逻辑的实现都放入这一层中，客户端程序只负责用户界面的呈现，它通过中间层与数据库进行交互。三层架构中的三层分别是：
  - 表示层：负责向用户呈现界面，并接收用户请求发送给业务逻辑层；
  - 业务逻辑层：负责执行业务逻辑以处理用户请求，并调用数据访问层提供的持久性操作；
  - 数据访问层：负责执行数据库持久性操作。

# 分布式系统架构 – 三层架构



- 三层架构的优点包括：
  - 有利于开发人员分工
  - 有利于系统维护和复用
- 三层架构的缺点包括：
  - 有些修改会贯穿所有层
  - 数据访问层潜在的问题

# 分布式系统架构 – 代理架构



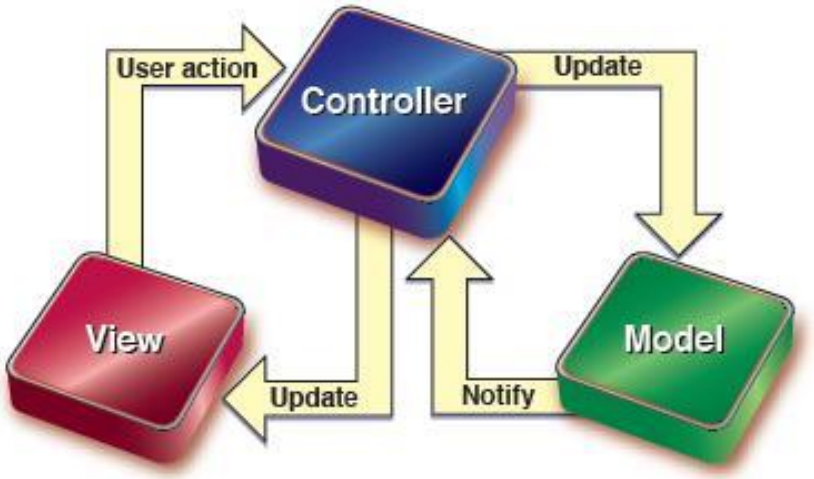
- 代理架构风格是在客户端程序和服务器端的业务逻辑执行程序之间插入了代理层，其目的是向用户屏蔽业务逻辑执行程序，并添加额外的控制逻辑。

- 我们可以在Portal技术中看到代理架构。

# 交互式系统架构 – MVC架构



- MVC风格将与三层架构风格很类似，它也强调了要将软件系统分成三个互相部分。
  - 视图：根据模型生成提供给用户的交互界面，不同的视图可以对相同的数据产生不同的界面。
  - 模型：管理系统中存储的数据和业务规则，并执行相应的计算功能。
  - 控制器：接收用户输入，通过调用模型获得响应，并通知视图进行用户界面的更新。



# 交互式系统架构 – MVC架构



- MVC架构和三层架构存在下列区别：
  - MVC架构的目标是将系统的模型、视图和控制器强制性地完全分离，从而使同一个模型可以使用不同的视图来表现，而计算模型也可以独立于用户界面；而三层架构的目标是将系统按照任务类型划分成不同的层次，从而将计算任务可以分布到不同的进程中执行，以提高系统的处理能力。
  - 模型包含了业务逻辑和数据访问逻辑，而在三层架构中这属于两个层的任务。

# 交互式系统架构 – PAC架构



- 表示-抽象-控制架构（PAC）与模型-视图-控制器（MVC）架构非常类似，我们大体可以认为表示与视图对应、抽象与模型对应、控制与控制器对应。
  - PAC是一种由智能体（Agent）构成的层次结构，每个智能体都包含表示、抽象和控制三个部分；
  - 智能体彼此间只能通过控制部件进行通信；
  - 在每个智能体内部，表示与抽象完全隔离，这使得它们可以在彼此隔离的多线程中运行，从而带给用户启动时间短的体验，因为用户界面（表示）可以在抽象完全初始化之前就显示出来

# 交互式系统架构 – PAC架构

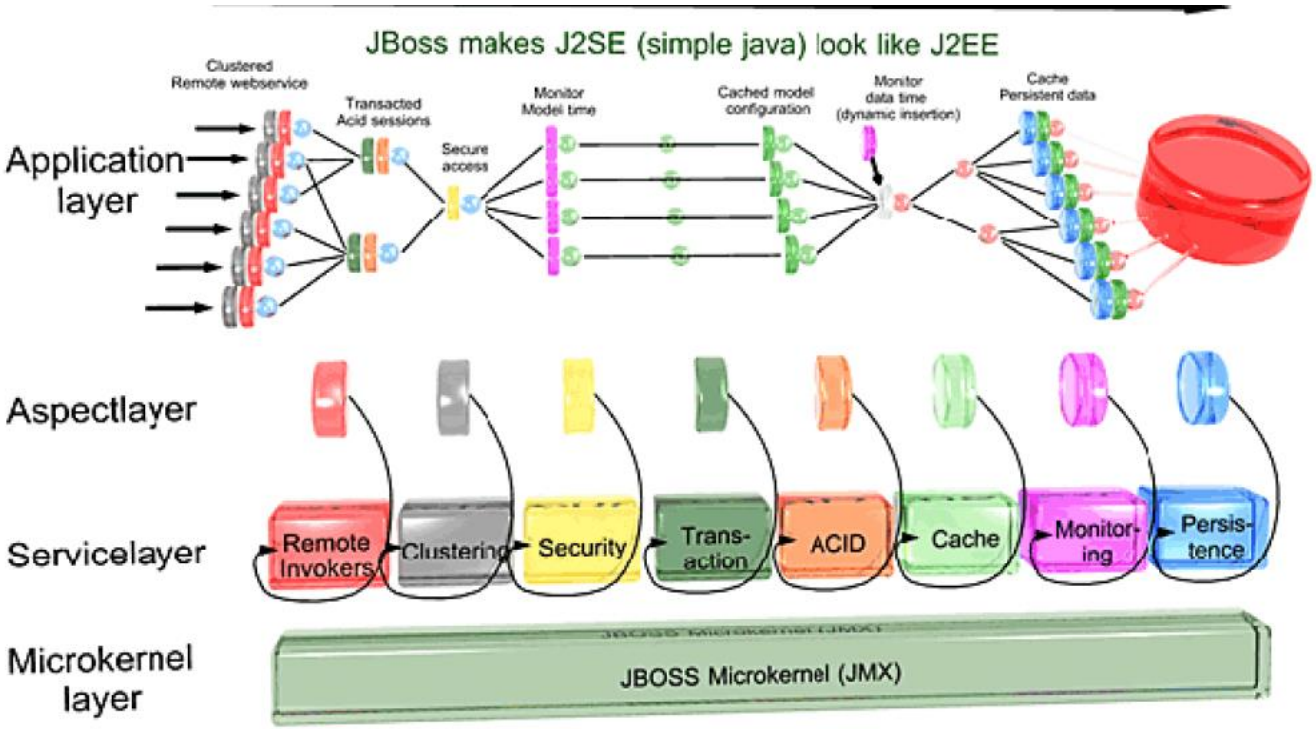


- 从MVC和PAC架构的结构我们可以发现，它们之间还是存在着一些差异的：
  - MVC的控制器关注于用户在视图上执行的输入操作和向用户产生的输出；而PAC的控制是抽象与表示，以及智能体之间的通信渠道，它关注于智能体之间，以及智能体内部从抽象到表示的通信和协调作用。
  - PAC把系统分解成了层次结构的、彼此互相协作的，但是又松散耦合的智能体；而MVC专注于模型与视图的隔离，并没有形成层次结构的若干个MVC智能体，因此其各个部分之间的关联更紧密。

# 自适应系统架构 – 微内核架构



- 微内核概念来源与操作系统领域。微内核是提供了操作系统核心功能的内核，它只需占用很小的内存空间即可启动，并向用户提供了标准接口，以使用户能够按照模块化的方式扩展其功能。现在大多数操作系统都采用了微内核架构。



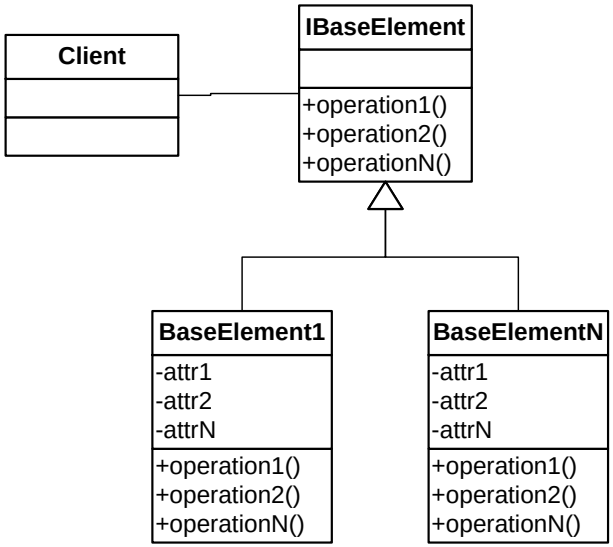
# 交互式系统架构 – 反射架构



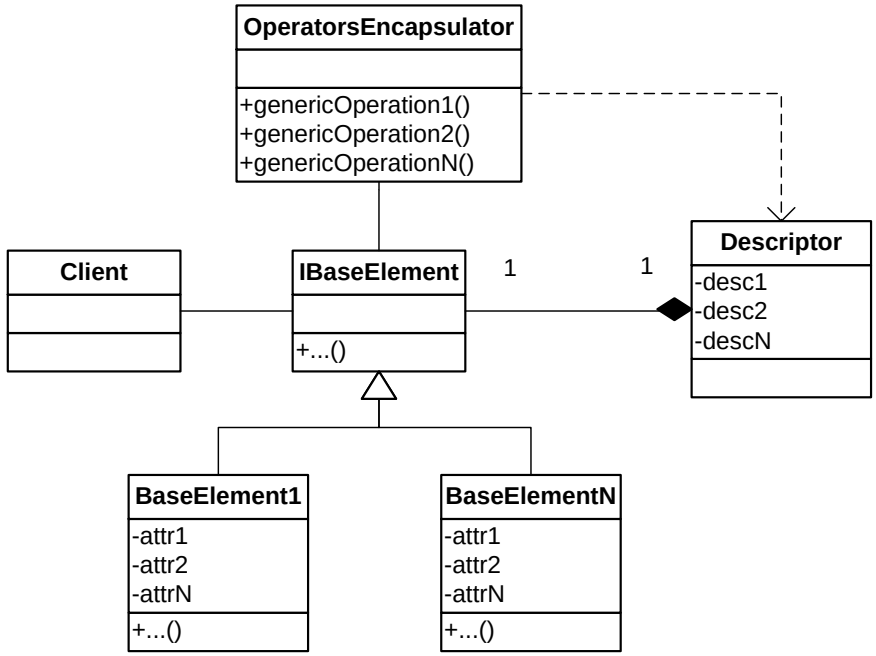
- 反射架构中包含一个被称为元信息（metainformation）的元素描述了系统中的其他元素，系统使用元信息作为基础元素（basic element）与那些被描述的元素进行交互，而这些被描述的元素通常称为基元素（base element）。这些基元素包含大量完全相同的属性，对于这些属性可以用统一的方式处理。

# 交互式系统架构 – 反射架构






（a）普通继承–实现架构




（b）反射架构

# 交互式系统架构 – 反射架构



- 反射架构的优点包括：
  - 对于添加元素和属性这样的变更非常容易实现
  - 它可以支持多种不同类型的变更
- 反射架构的缺点包括：
  - 效率不高
  - 类结构变得复杂

# 其他架构 – 批处理架构



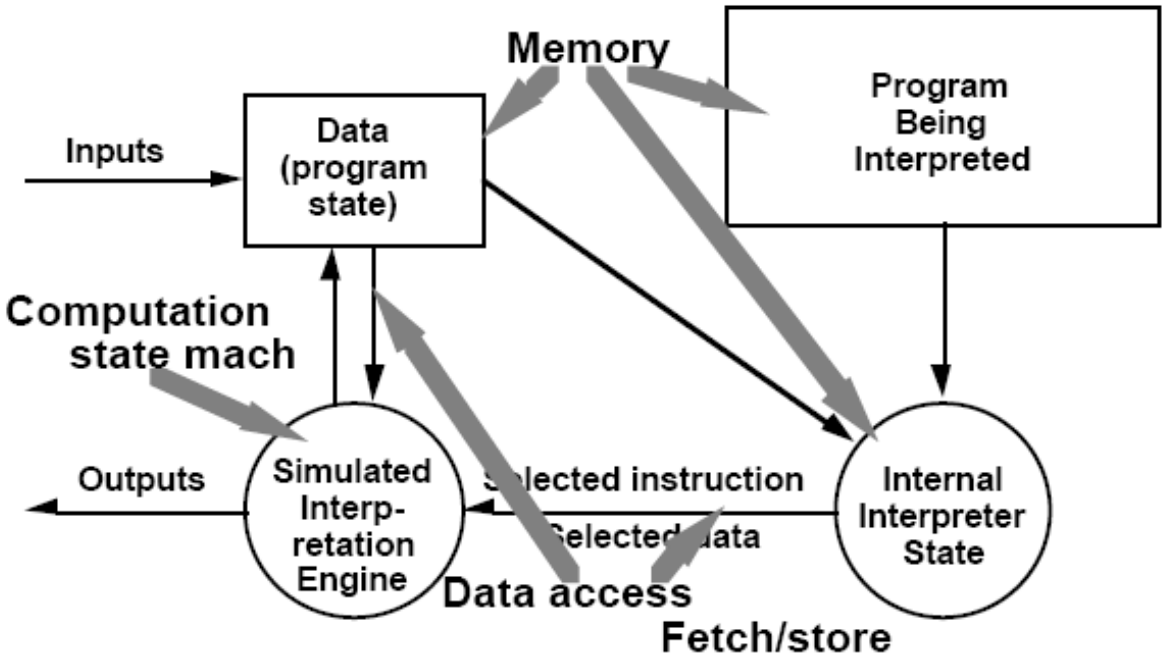
- 批处理架构实际上是管道-过滤器架构的一种降级形式，在批处理架构中，管道不再服务于提供数据流这个功能。批处理架构也强调数据流在不同的处理步骤之间流动，但是与管道-过滤器架构不同的是，在某个处理步骤处理完批量数据之前，是不会进去下一个处理步骤的，也就是说，数据是以块的方式整体在步骤之间进行传递的，而每个处理步骤都是互相独立的程序。

- 程序开发就是典型的批处理架构

# 其他架构 – 解释器架构



- 解释器架构用于仿真当前不具备的计算环境，通常包含四个组成部分：用来解释伪码程序的解释引擎、包含待解释程序的内存、解释引擎的控制状态，以及被仿真程序的当前状态：



# 其他架构 – 解释器架构



- 解释器架构的优点包括：
  - 可以使编写的程序独立于具体的运行环境
  - 可以仿真当前不具备的运行环境
- 解释器架构的缺点包括：
  - 对性能影响较大
  - 解释器本身需要经过验证

# 其他架构 – 进程控制架构



- 进程控制架构中，系统被分解成许多独立运行的进程，它们彼此之间通过同步或异步的方式，按照指定的协议通信，通信链路构成了整个系统的拓扑结构特性，例如环形结构或星型结构。
- 实际上，客户端-服务器架构就是一种进程控制架构，其中服务器和每个客户端也都运行于独立的进程中，它们通过远程方法调用进行通信。

# 其他架构 – 进程控制架构

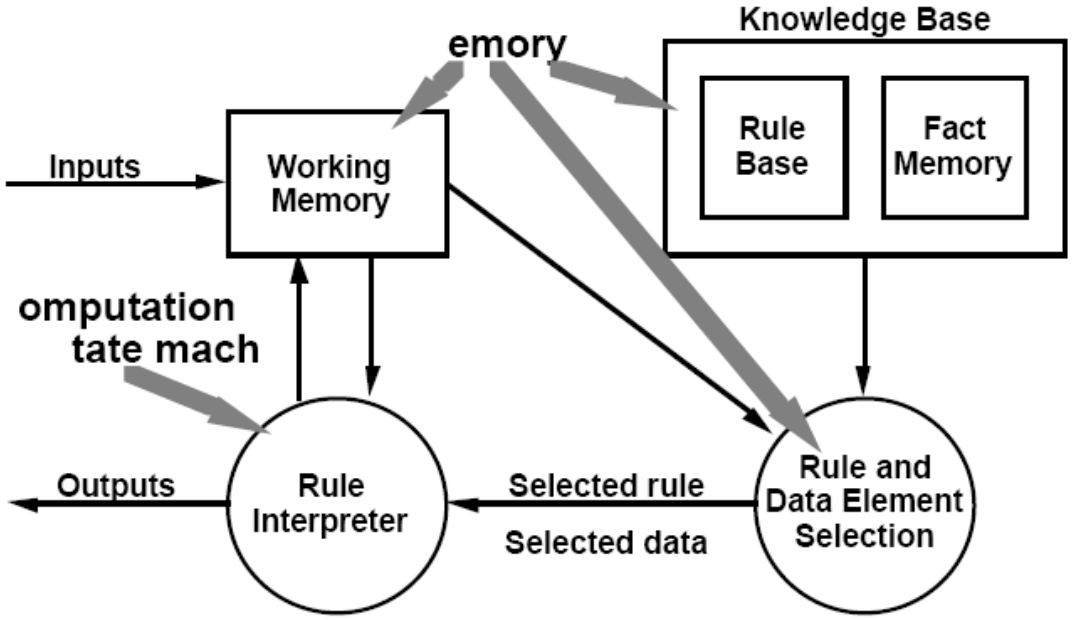


- 进程控制架构的优点包括：
  - 有利于问题分解
  - 有利于提高性能
  - 有利于提高可靠性
- 进程控制架构的缺点包括：
  - 进程间调用的时序控制困难
  - 进程间通信开销较大

# 其他架构 – 基于规则的架构



- 基于规则的架构是一种解释器架构风格，它将人类专家的问题解决知识编码成规则，这些规则在系统执行计算满足指定的条件时被执行或激活，通过规则不断地被执行和激活，最终使得问题被解决。由于这些规则不能被计算机系统直接执行，因此需要通过解释器来解释它们。



# Heterogeneous Architectures



- The combination of several styles.
- Local heterogeneous
- Hierarchy heterogeneous
- Parallel heterogeneous

# References



1. Indian Institute of Software Engineering's courseware of SA
2. SWEBOK, http://www.swebok.org
3. Software Architecture, http://en.wikipedia.org/wiki/Software_architecture
4. David Garlan & Mary Shaw, An Introduction to Software Architecture, http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf, 1994.1
5. Presentation-abstraction-control, http://en.wikipedia.org/wiki/Presentation-abstraction-control
6. JBoss AS 4.0 Datasheet, http://www.jboss.com
7. Reflective Architectures and Code Generation, http://www.moisesdaniel.com/wri/racgen.htm
8. F. Hayes-Roth, 'Rule-based systems," Communications of the ACM, vol. 28, pp. 921-932, September 1985.


# Thank You!