Software Architecture Course Review

QA概念题

- What is software architecture and its components?

根据CMU，Mary Shaw给出的定义：

Software architecture=components+connectors+constraints

component：定义计算的位置，connector，构建进行通信协作的载体，传达构件间的交互，constraint：对模型拓扑结构与行为的约束。

比如在面向对象语言中，components是对象（类），connectors是过程调用（函数调用），constraints是对象内部隐藏信息的限制访问（比如C++中类只有内部函数能访问私有成员）

- What is software architecture style and its components?

软件体系风格描述了一类软件系统中通用的组织架构，且它被多次设计、应用，是若干设计思想的综合，具有熟知的特性且可以被复用。

组成：

1.一组组件类型，比如数据容器，过程，对象

2.一组连接件类型与交互机制，比如过程调用、事件、管道

3.这些组件的拓扑分布

4.一组对拓扑与行为的约束，比如数据容器不能自己改变数据，对象之间内部信息的访问限制

- Explain risks, non-risks, sensitivity, tradeoff points

risks: 有潜在问题的设计决策

non-risks: 良好的、可提高架构质量，帮助实现设计目标的设计决策，通常隐式地包含在架构中

sensitivity: 构件或构件间关系的某一特性，这种特性会影响一个特定质量属性

tradeoff: 影响多个质量属性的设计决策

- Explain quality attribute scenario

质量属性场景是对系统如何响应某一刺激的简短描述。

我们通常用六要素法来进行描述，将其分为六个部分。

source: 产生刺激的实体。

stimulus: 影响系统的条件

artifact: 系统被刺激物刺激的部分

environment: 刺激产生的条件

response: 系统收到刺激而产生的活动

response measure: 系统响应如何被度量

比如：用户在系统正常运行的情况时访问网页的响应时间不超过1s，在这个质量属性描述中，刺激源：用户，刺激物：访问网站请求，artifact：网站服务器，environment：正常运行，response：正确响应，返回网页数据，response measure：响应时间不超过1s。

Utility Tree

Utility tree

- performance:
  - latency e.g. data latency
  - throughput e.g. Transaction throughput

    - response time
    - capacity
- security:
    - confidentiality e.g. Data confidentiality
    - integrity e.g. Data integrity
- modifiability:
    - modify/create/delete… e.g. Change COTS, New product
    - reconfigure sth. e.g.
- availability:
    - failure(H/W,S/W, COTS)
    - data loss

P.S. (H,M) —>(importance, implementation difficulty)

Identifying sensitivity point, tradeoffs, risks, non-risks

- risk point: 有潜在的问题 设计决策**(architectural decisions** 下同)
- Non-risk point: 良好可提高质量、帮助实现目标的设计决策，通常隐式地包含在架构中
- sensitivity point: 显著影响特定质量属性的设计决策
- tradeoff point: 影响多个质量属性的设计决策

质量属性(Quality Attribute)

质量属性场景

> 质量属性场景是对一个系统如何响应特定刺激的简短描述.
>
> **A quality attribute scenario is a short description of how a system is required to respond to some stimulus.**
>
> 六个组成部分：
>
> 刺激源：产生刺激的实体
>
> 刺激物：影响系统的条件
>
> 制品：系统被刺激物刺激的部分
>
> 环境：刺激产生的条件
>
> 响应：因刺激产生的活动
>
> 相应度量：系统响应如何被度量

具体质量属性和**solutions**

> - Availability
>
> solutions: Ping/Echo, rollback, exception, heartbeat, voting
>
> - security
>
> authenticate users, maintain data confidentiality, limit access, limit exposure, intrusion detection
>
> - Usability
>
> Cancel, undo, aggregate, maintain user model, maintain system model, maintain task model

- Modifiability

Hide information, use an intermediary, anticipate expected changes, configuration files, maintain existing interface

- Performance

increase computation efficiency, reduce computational overhead, introduce concurrency, increase available resources, scheduling policy.

- Testability

Record/Playback, separate interface from implementation, specialized access routine/interface

软件架构风格

各风格

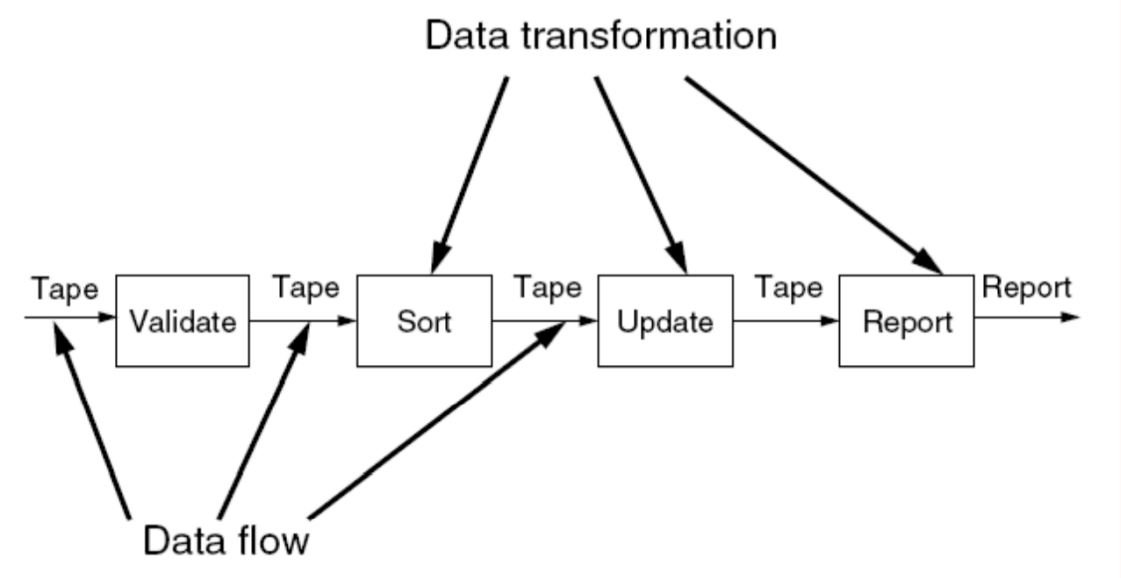
Data flow

数据控制计算、结构由数据在处理过程中的有序流动确定、系统结构显而易见

系统中除了数据交换没有其他任何交互、关注核心为计算顺序

Batch sequencail

- 每个步骤是独立的程序
- 每一步在前一步结束之后才能开始
- 数据必须是 完整地、整体地 方式传递



component: data transformation

connector: data stream

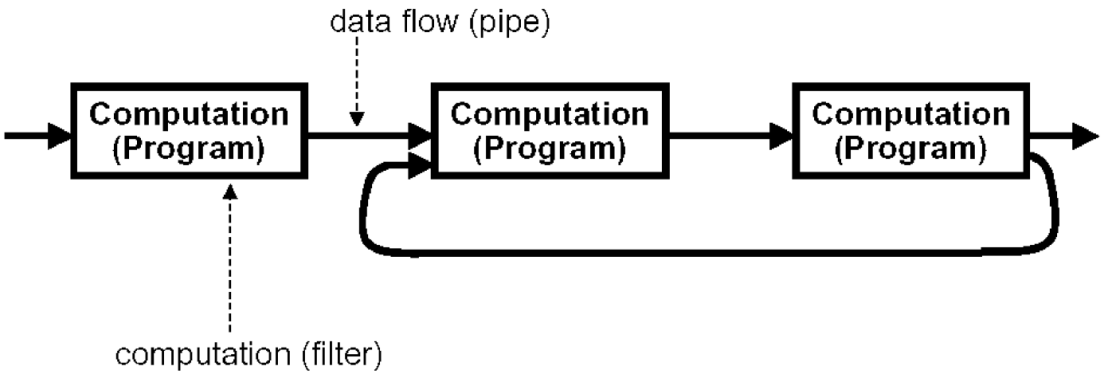
constraint: 执行顺序确定，数据必须完整传递

关键字：数据完整传递、每个处理步骤独立、按步进行

例子：编译器（早期），CASE，函数式编程

Pipe and filter

- 通过计算和增加信息来丰富数据
- 通过删减和浓缩来精炼数据
- 通过改变表现形式来转化数据



component: filter

connector: pipe

constraint: 过滤器各自独立，相互无联系

关键词：支持并发、数据有增减、高内聚低耦合、交互性差（好于批处理）、性能不高、模块重用、增量、可有反馈环

其他：开环 or闭环控制，数据 push or pull

优点	缺点
良好的隐蔽性，高内聚低耦合	不适合交互性强的应用
支持模块重用（传输数据格式一致）、支持并发执行	数据传输无标准，降低性能
便于设计者理解、易于维护和扩展	同步问题
支持特定属性分析（吞吐量，死锁）	

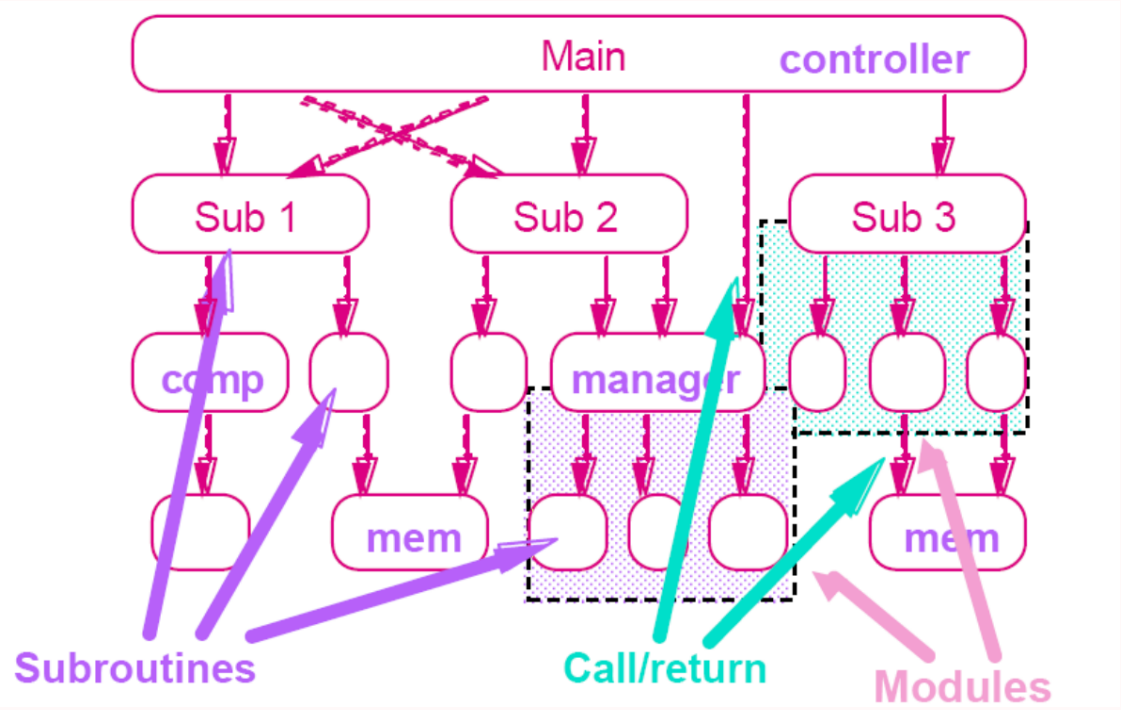
Batch sequential 与 Pipe/Filter对比

Batch Sequential	Pipe/Filter
•total	•incremental
•High latency (real-time is hard)	•Results start immediately
•Random access to input ok	•Processing localized in input
•No concurrency	•Feedback loops possible
•Non-interactive	•Often interactive, awkwardly

Call/Return

Main Program and Subroutine

适合计算过程可被定义为结构化过程的应用；例如编程语言提供嵌套过程和结构化调用方法



component：过程和显式可见数据

connector：过程调用和显式数据共享

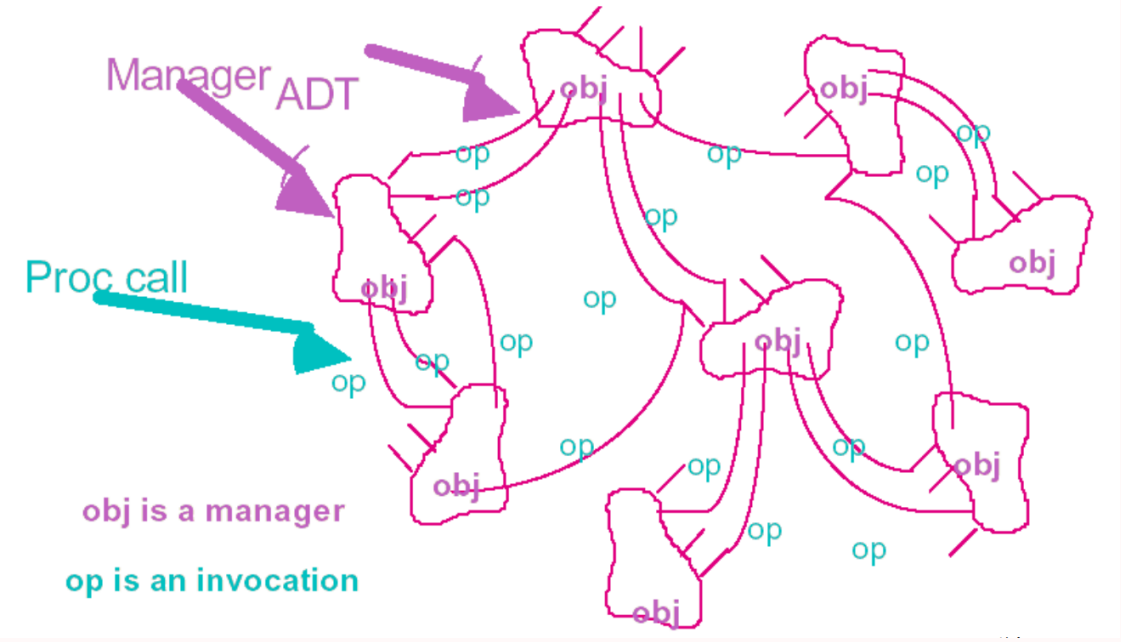
constraint：适合计算可被结构化定义的应用

关键字：显式可见数据、结构化过程、嵌套过程

优点	缺点
逐步分解问题,降低问题复杂性	只适合用于可定义为一系列步骤的问题
单线程控制	子系统的结构不清晰

OOP/ADT

适用于中心问题是识别和保护相关对象的信息特别是表示信息（**representation information**）



component: managers(servers，objects，ADT)

connector: procedure call

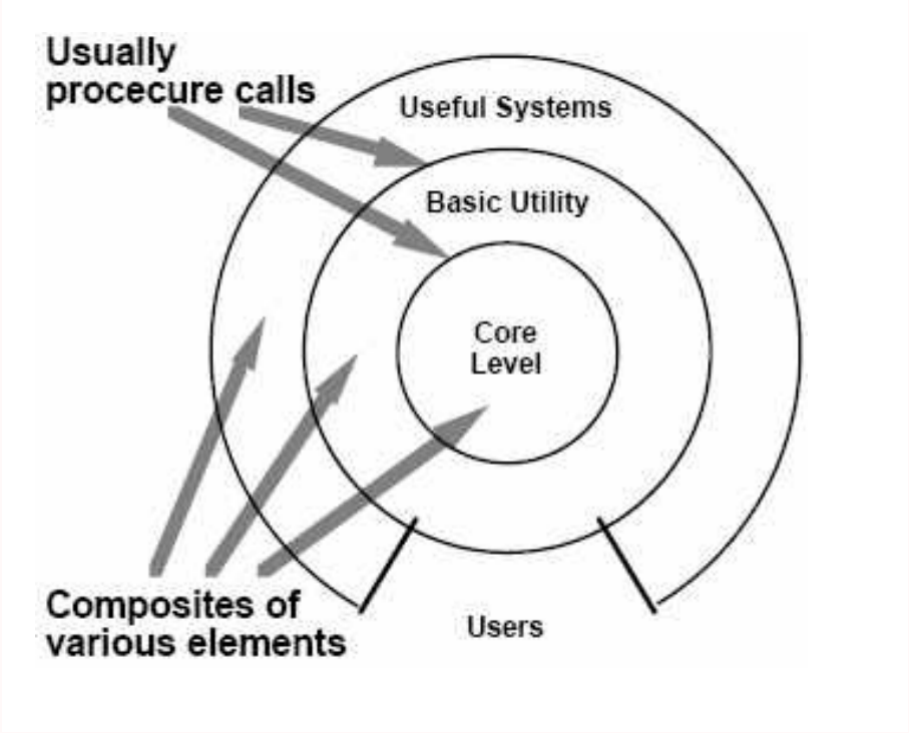
control structure: decentralized，usually single threaded

关键词：封装、交互、继承、复用和维护、保护相关信息、限制访问

优点	缺点
使用时不需要知道其内部具体实现，组件间操作以黑箱的方式进行，提高易用性	对象之间耦合较紧，交互时需要知道对象的标识，标识改变时，必须修改所有显式调用它的对象，性能降低
封装使内部细节对外部环境得以良好隐藏，安全性增强	单一共享接口能力有限且笨拙（故引出“友元”）
设计时分解出具体的类和对象，跟现实世界对应，便于理解	对象过多时需要额外的结构来容纳、维护，复杂性提高
	不同对象对统一资源访问时的状态的改变和维护问题

Layered System

适用于不同服务类别可被分层管理的应用



component: 各层内部构件（一般是一些过程）

connector: 层间交互协议，通常是限制可见性的过程调用

control structure: 单线程

优点	缺点
每层为上层提供服务，使用下层服务，只能见到相邻层	上层必须知道下层身份，不能调整层间顺序
大问题分解为若干个渐进的小问题，问题复杂性降低	层层相调，数据传输开销大，影响性能
修改一层，最多影响两层，降低耦合	层次大小难以清晰界定，不是每个系统都能清晰地划分层次
支持复用，层与层通过接口交互，接口不变即可重用	缺少合适的层次抽象方法。层次太少不能发挥可重用性、可更改性、可移植性上的潜力；层次太多，会引入复杂性和层间间隔冗余且，提升层间传输开销

关键词：分层、限制可见性、重用、低耦合

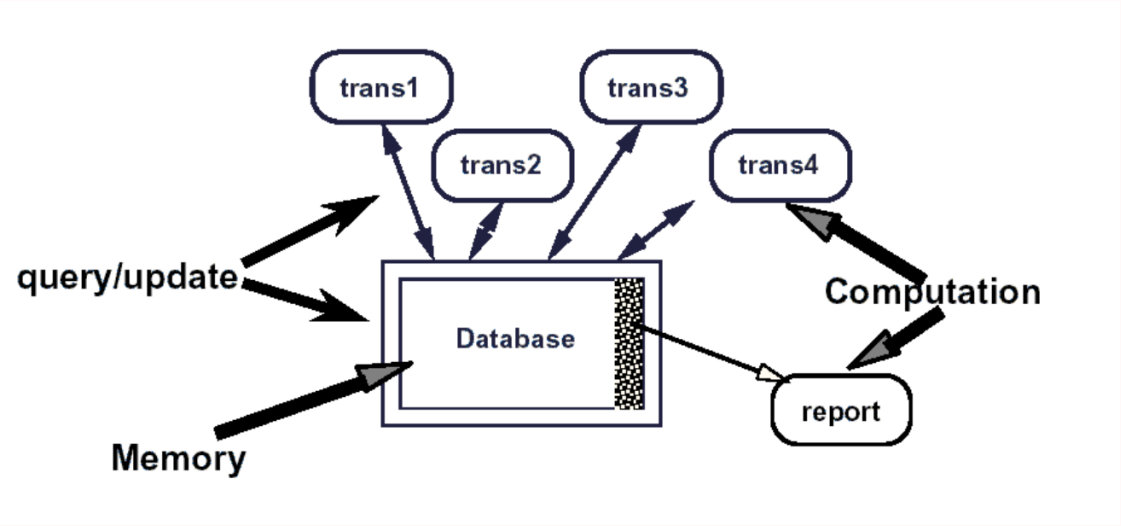
Pipe 与 procedure对比

	Pipes	Procedures
Control	Asynchronous, data-driven	Synchronous, blocking
Semantics	Functional	Hierarchical
Data	Streamed	Parameter / return value
Variations	Buffering, end-of-file behavior	Binding time, exception handling, polymorphism

Data centered

Repository

适用于中心问题是为了创建、增强和维护一个复杂的中心信息体的应用



component: 一个内存，多个纯计算进程

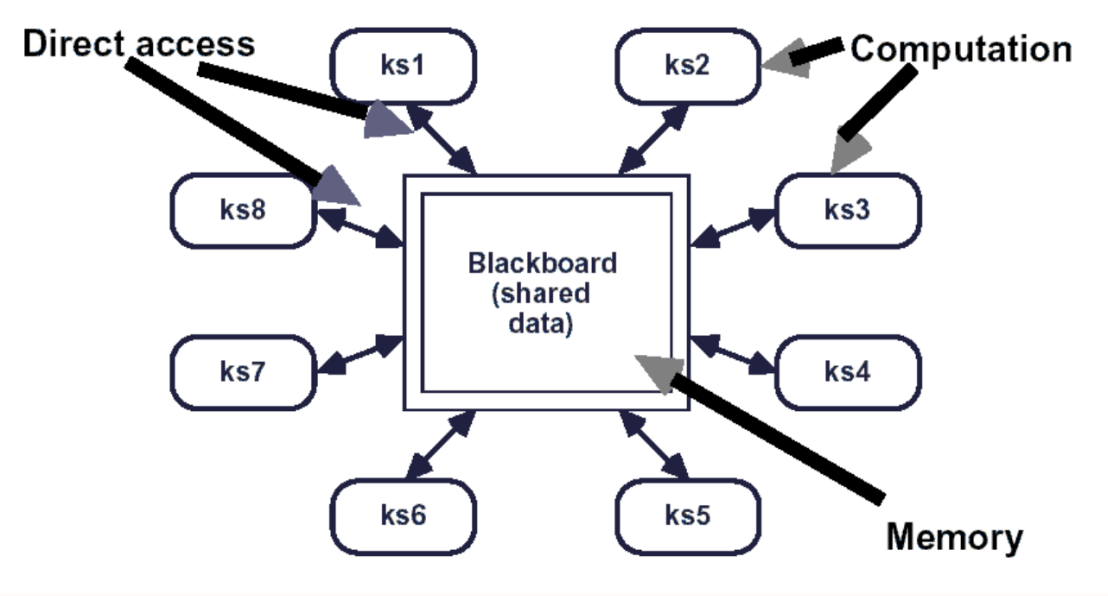
connector: direct memory access 或者procedure call来与内存模块进行交互的计算单元

constraint/control structure: predefined(compiler)/external(input data stream,database)/internal(state of computation, blackboard)

优点	缺点
容易增加数据的生产者和消费者	内存共享数据的同步问题
计算单元可并行，高性能	ACID、性能
交互性更强	配置和管理

关键字：中心信息体，共享内存，多个计算进程

Blackboard



component: knowledge source , blackboard(shared data)

connector: direct memory access

constraint/control structure: internal(driven by the state of the computation)

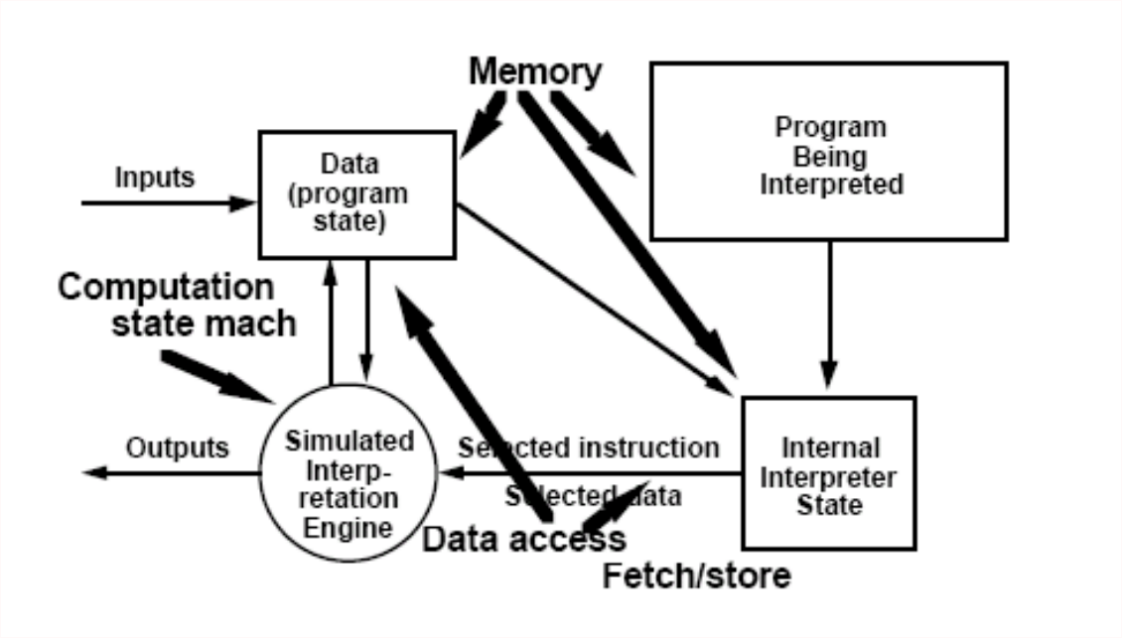
优点	缺点
知识源可以复用	数据和结果可能错误
规模伸缩性好(scalable)，方便添加新数据	测试困难，计算开销大，效率低
可用于非确定性问题的求解	不同数据源对共享数据结构需达成一致，使对黑板数据结构修改更加困难
	可并行处理，但需要同步/加锁机制来保证数据完整性和一致性，增加系统复杂度

关键字：知识源响应偶然事件，黑板状态驱动，状态决定知识使用，控制流只修改黑板状态

Virtual machine

Interpreter

适合目前没有合适语言/系统来执行解决方案的应用



component: 一个状态机（execution engine），三个内存（输入数据、被解释的代码、内部解释器状态）

connector: data access and procedure call

constraint/control structure: state-transition for execution engine; selection of what to interpret

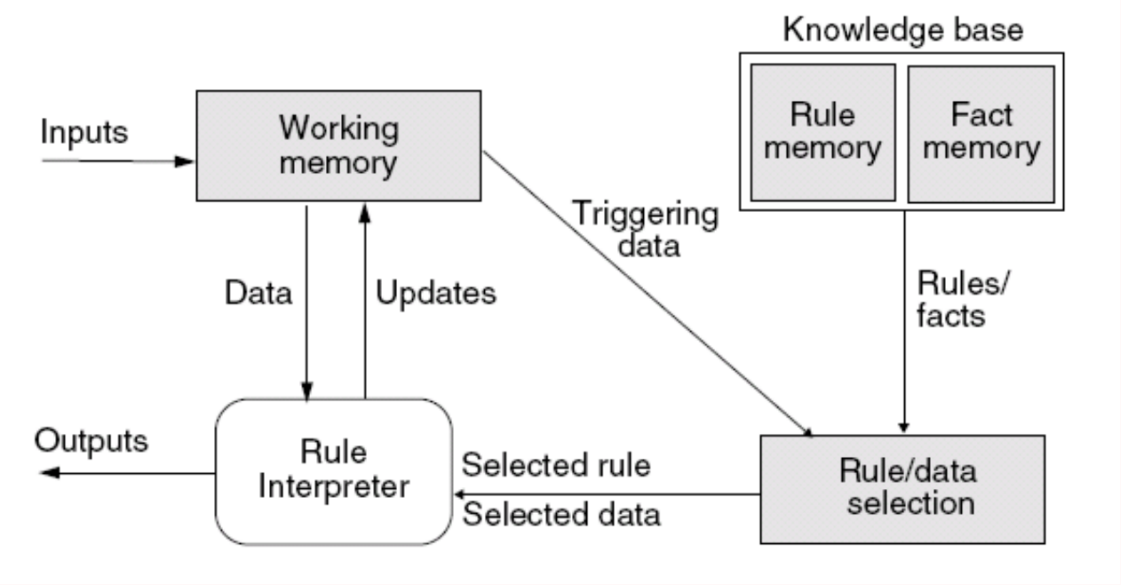
优点	缺点
可模拟非本地原生支持的功能	效率比硬件及原生支持的系统低
可模拟极端条件(disaster mode)来测试系统	额外软件层的正确性需要被验证
使用灵活，用途广泛	

关键字：模拟非原生功能，效率低

Rule-based system

Rule-based systems provide a means of codifying the problem-solving skills of human experts.

Rule-based systems make heavy use of pattern matching and context (currently relevant rules).



component: working memory, knowledge base, Rule/data selection, rule interpreter

connector: direct access and procedure call

constraint/control structure:

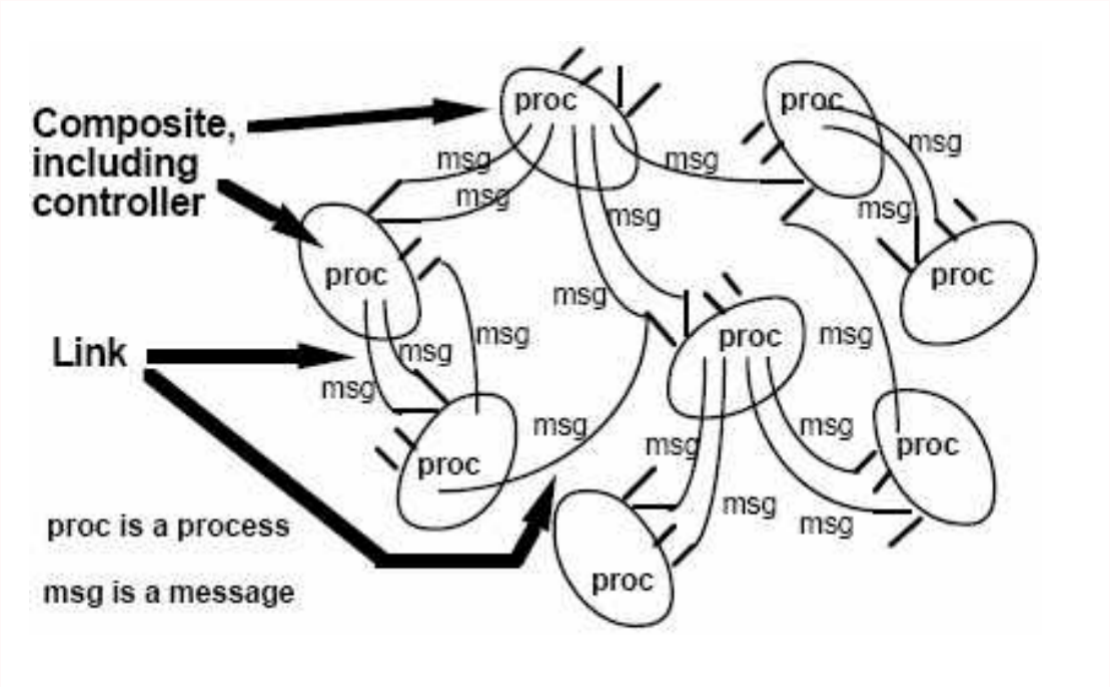
优点	缺点
产生式结构统一(if-then)，且产生式规则的语法让其有自释性	规则间关系不透明，不知道单个规则对于整体策略的影响，缺乏分层的知识表达
知识与处理分离，系统拓展性强，知识添加时灵活，不影响控制结构	低效的搜索策略。推理引擎周期性搜索所有策略，规则过多时，系统速度变慢
知识可以由自然语言表达被转化，灵活简单	没有学习能力，不能从已有经验中学习新的知识

关键字：知识，规则

Independent component

Communicating processes

Suitable for applications that involve a collection of distinct, largely independent computations whose execution should proceed independently



component: processes that send & receive messages from **explicitly** selected recipients

connector: 有已知通信对象的离散的消息（无数据共享），点对点，同步或异步

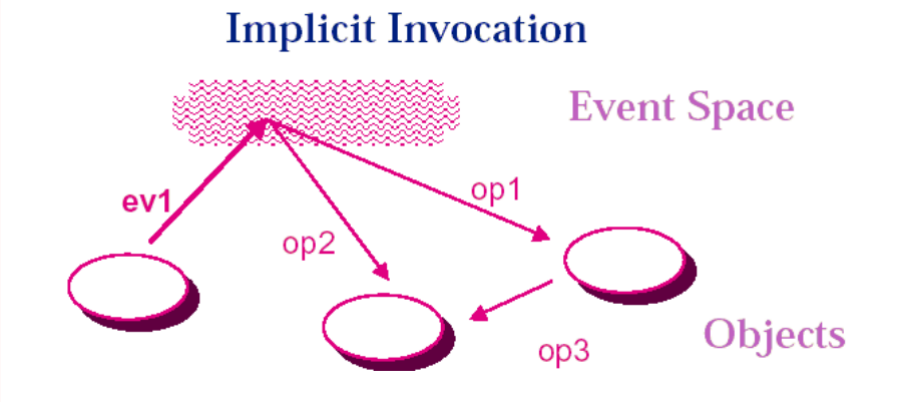
constraint/control structure: each has its own thread of control

优点	缺点
可多线程，性能高	

关键字：多线程

Implicit invocation

Suitable for applications that involve loosely coupled collection of components, each of which carries out some operation and may in the process enable other operations.



component: 发布事件但不知道接收方的进程

connector: 事件-过程绑定，对已注册该事件的进程进行自动调用

constraint/control structure: 去中心化，独立组件不知道消息接收方存在

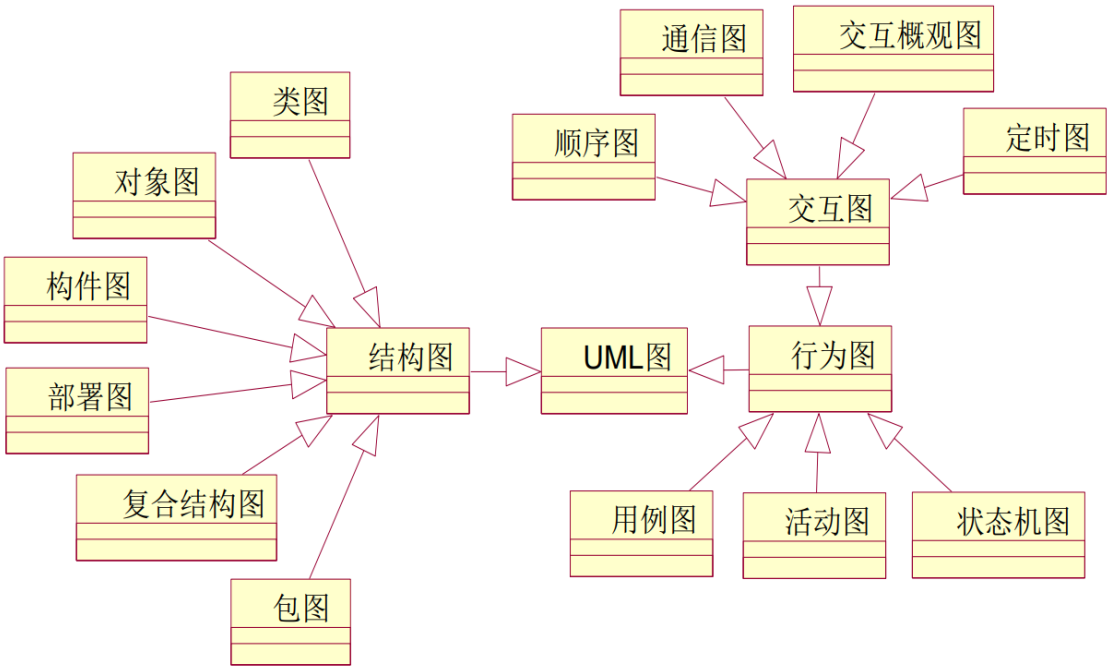
优点	缺点
软件重用性强。将某一构建加入系统时，只需要将其注册到系统事件中	对系统计算失去控制，不能确定具体调用进程与顺序，且过程调用可能有语义错误，可能成环造成循环调用
为改进系统带来方便。对旧构件进行替代时，不会影响其他构件的接口	事件处理跟其他运行机制难以交互（event control loops of X, RPC）
可并行执行，性能高	引入事件中心(event space)，间接通信有性能损失
单个组件的崩溃不会影响其他组件，鲁棒性强	数据交换问题：有时事件依赖共享数据的中心(储存事件，登记信息，调度策略的“黄页”)进行交互，此时，整体性能和资源管理可能会成为问题

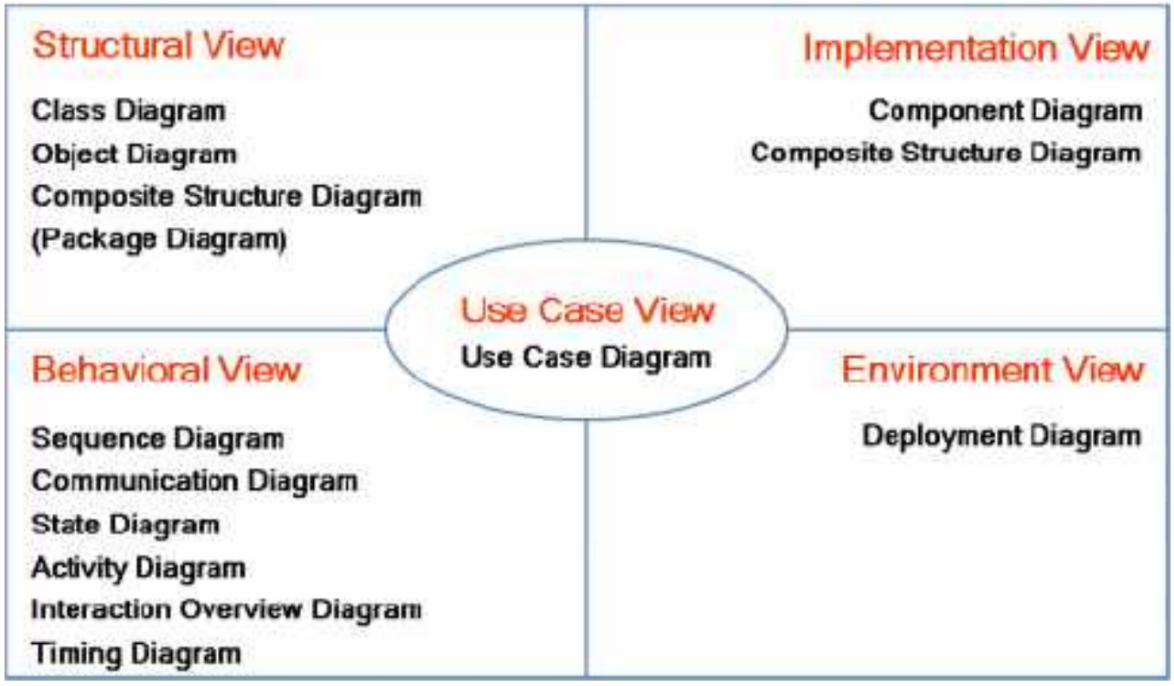
关键字：事件，事件过程绑定，隐式，并行

4+1 view & UML

概念

- 用例视图（Use case view）， 从外界来描述所建立系统的功能
- 逻辑视图（Logical View），系统部件的抽象描述。用来表示系统组成与部件的交互方式。
- 过程视图（Process View），描述系统中的过程(process)，有助于可视化系统具体功能。
- 物理视图（Physical View），描述了系统设计与现实实体对应，将抽象部件与最终系统部署相映射。
- 配置视图（Development View），描述了系统模块与组件如何组织。




Structural View
Class Diagram
Object Diagram
Composite Structure Diagram
(Package Diagram)
Implementation View
Component Diagram
Composite Structure Diagram
Use Case View
Use Case Diagram
Behavioral View
Sequence Diagram
Communication Diagram
State Diagram
Activity Diagram
Interaction Overview Diagram
Timing Diagram
Environment View
Deployment Diagram