软件架构

- SEI：程序或计算系统的软件架构是这个系统的结构，包括了其内部的软件部件（software element）及软件部件可见的外部属性及部件之间的关系。
- IEEE：一个系统的基本组织，包括了其内部的组件，组件与组件、组件与外部环境之间的关系，和设计与评估中遵循的准则。
- Software architecture = {Elements, Forms, Rationale/Constraints}
- vs 设计：
  - 架构关于设计。所有的架构都是设计，不是所有的设计都是架构；架构是设计的一个过程。
  - 视图。高层视图、一组设计决策、本地化。
  - 系统的结构/组织方式。元素（组件&连接件）、关系（静态、动态）
  - 属性：元素、元素组、整个系统
- vs 结构：
  架构定义了接口（结构）、交互方式（数据、控制流）、依赖、响应。
- 为什么重要：
  软件架构提供了交流
  软件架构展示了初期的设计决策
  软件架构会影响质量属性的实现（促进或阻碍）
  软件架构讨论了潜在的变化
  架构是可转移和可重用的的抽象
  系统可以通过架构集成独立的组件

软件师架构做了什么

- 联络：架构师扮演了多个联络角色。他们需要促进客户、技术团队、商业/需求分析团队之间的联络。他们需要时刻与管理、评估、设计、成本进行联络。大多数的时间里，这些联络主要是在不同的利益相关者之间通过简单的翻译和术语的解释来实施的。
- 软件工程：架构师必须促进良好的软件工程实践。他们的设计必须被充分的文档化和交流、他们的计划必须被明确和评估。他们必须了解他们设计下的潜在影响，从而和测试、文档化能合适的工作，减轻团队负担。
- 技术知识：架构师需要对跟他们工作应用相关的技术域有一个深入的了解。他们在评估和对第三方组件和技术上选择上是尤为重要的。好的架构师必须知道他们什么东西不知道。
- 风险管理：良好的架构师通常倾向于细致和谨慎。他们在不断的发现和评估设计和技术选择相关的风险，对这行风险进行文档化和管理。他们尝试保证不会有预期以外的灾难发生。

软件架构从何而来

- NFRs, ASRs, Quality Requirements; Stakeholders, Organisations, Technical Environments
- 不仅仅只从需求（功能性需求）
- 系统的利益相关者
  - 管理（低成本、高 ROI（投资回报率））
  - 市场（特点、一组竞争的产品）
  - 终端用户（易用性、可靠性、安全性）
  - 维护（可维护、灵活、易配置）
- 开发团队

- 现存的机制和外在的机遇
- 开发过程

- 架构师
  - 背景知识和经验
- 技术环境
  - 软件工程中的最佳实践或软件工程技术/工具

架构 4+1 视图

- 逻辑视图（Logic View）：描述了重要架构元素及其之间的关系。
- 进程视图（Process View）：描述了元素之间的并发和通信。
- 物理视图（Physical View）：描述了主要的进程和组件与应用硬件之间的映射关系。
- 开发视图（Development View）：关注软件组件内部的组织
- 架构用例（Architecture Use Case）：通常也叫 scenarios，描述了架构的需求；跟不只一个视图相关。

架构的活动和过程

- 活动

  为系统创建商业用例（场景）

  理解需求

  设计、选择架构

  对架构进行交流（利益相关者包括开发者）

  分析、评估架构

  实现架构

  保证一致性
- 过程

  识别 ASR

  架构设计

  架构文档化

  架构评估

架构知识域

- 软件设计基础概念
  - 设计概念常识
  - 上下文（Context）：软件开发生命周期：需求、设计、构造、测试
  - 设计过程
  - 在设计中使用技术
- 关键点：并发、控制和处理时间、分布式、异常处理、交互系统、持久化
- 软件结构和架构：
  - 架构的结构和 viewpoints
  - 架构的模式和风格
  - 软件设计方法：架构方法（ADD）、设计方法

功能性需求、质量需求、约束

- 功能需求：
  - 规定了系统必须做什么、为利益相关者提供了什么价值
  - 代表了系统的行为
  - 和架构独立，功能性可以单独以一个没有内部结构的系统存在

- 质量需求：（非功能性需求、架构需求）
  - 质量需求是系统需要提供的在功能性需求之上的特点
  - 是系统产品功能性需求的检验标准
  - 软件架构会根据质量需求将功能性需求分配到了不同的结构中
  - 分为可观测（外部，性能、安全性、可用性、易用性等）与不可观测（内部，可修改性、可重用性、可测试性等）
- 约束：
  - 具有 0 自由度的设计决策（必须遵守）
  - 约束提前规定了一些设计决策
  - 约束可以通过采取一些相关的设计决策来得到满足

质量属性的场景

- 刺激（Stimulus）：一个系统需要考虑到的情况，如发生错误、非法访问
- 刺激源（Source of Stimulus）：生成刺激的实体
- 响应（Response）：当刺激到来时系统做出的反应
- 响应度量（Response Measure）：响应应该在某种程度上被度量，以方便测试
- 环境（Enviroment）：刺激发生的系统环境，如过载、正常工作等
- 制品（Artifact）：实现需求的整个系统或系统的一部分

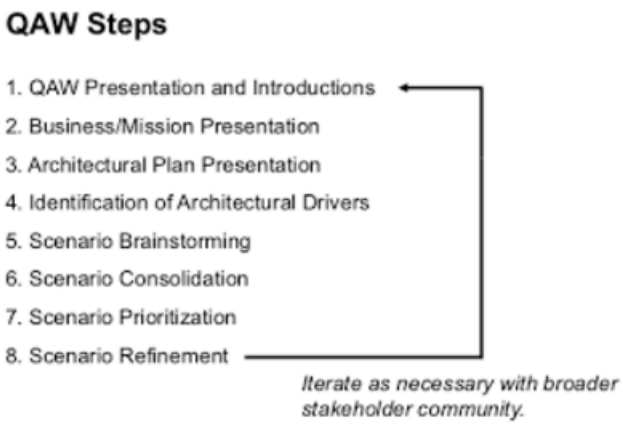
常见的质量属性

ASR（Architecturally Significant Requirments）

- ASR 是对系统有重要影响的需求——如果没有此需求系统完全不一样
- 质量需求越难、越重要，其更有可能影响系统，会成为一个 ASR

如何识别 ASR

- 从需求文档中获取
- 从与利益相关者的交流中获取（Qaulity Attribute Workshop，QAW）
  - system-centric
  - stakeholder focused
  - used before the software architecture has been created
  - scenario based



- 从理解商业目标中获取
- 从质量效用树中获取
- Persona-Based 方法

tactic

- Style or Patter 采取了 Tactics 来保证质量属性
- tactic 是一个影响控制质量属性响应的设计决策，如冗余
- tactic 的一个集合叫做策略（strategy）
- 同 patterns，tactic 也可以是由其他 tactic 组成
- tactics 可以被用作 tactics 树

常见的 Tactic：

- 可用性
  - 错误检测 ping/echo 心跳 异常 自检
  - 错误恢复 表决 主动冗余 被动冗余 备件 shadow 操作 状态再同步 检查点/回滚
  - 错误预防 从服务中删除 事务 进程监视器 异常预防
- 互操作性
  - 定位 服务发现（从一个已知的服务目录中定位服务）
  - 管理接口 Orchestrate Tailor Interface
- 可修改性
  - 减小模块大小：模块分解
  - 增加内聚：提高语义内聚性
  - 减小耦合：封装、使用中间者（代理）、限制依赖、重构、公共服务抽象
  - 推迟绑定时间
- 性能

        - 控制资源需求：管理样品率（Manage Sampling Rate）、限制事件响应、优先级事件、限定执行时间、增加资源的有效性
        - 资源管理：增加资源、引入并发、维护多个计算副本、维护多个数据副本、限定队列大小、调度资源
    - 安全性
        - 抵抗攻击：用户识别、用户授权、限定访问、限定公开信息、加密信息、实体分离、改变默认设置
        - 检测攻击：检测机制、检测信息时延、验证信息安全、检测服务、拒绝
        - 对攻击做出反应：锁定计算机、提醒用户、重新确认权限
        - 从攻击中恢复：重新恢复（查看可用性），审计追踪
    - 可测试性
        - 控制和观察系统状态：特殊接口、记录/回放、本地化状态仓库、抽象数据资源、沙盒、可执行断点
        - 限定复杂度：限定架构复杂度、限定非确定性。
    - 易用性
        - 支持用户自发操作：取消、撤销、暂停/继续、聚集
        - 支持系统自发操作：维护任务模型、维护用户模型、维护系统模型

架构模式（element+relations+constrains）

- 架构模式是在实践中重复出现的一组设计决策
- 通常可以重用，也描述了一类架构
- 建立了上下文、问题、解决方案之间的关系
- 模式是在实践中发现的（模式不能被发明，只能被发现），不会有一个完整的模式列表

各种架构模式

Module Pattern

Layer

Component-connector Pattern

Broker、MVC、pipefilter、C/S、P2P、SOA、publish/subscribe、share-data

Allocation Pattern

multi-tier、MapReduce

架构模式和架构风格(Pattern & Style)

- 架构模式（Pattern）会关注问题和其上下文，与问题如何在上下文中解决。
- 架构风格（Style）关注架构方法更强调说明一个架构风格有用或无用。
- 一个架构风格的描述不会包括详细的问题和上下文信息；但是模式会包含。

架构模式和架构策略(Pattern & Tactic)

- 区别于 Pattern，Tacitc 更简单，密度更小。
- Pattern 会互相包含和交叉。
- Pattern 通常包含多个设计方案。
- Pattern 和 Tactic 共同构建了整个架构基础的工具。
- Pattern 看的角度不同。

- Pattern 会包含一定的 Tactic。

架构设计思想

- 分解（Decomposition）、抽象（Abstraction）、分治（Divid and Conquer）、生成和测试（Generate and Test）、迭代（Iteration）、重用（Reuseable elements）
- 架构是一系列的设计决策。设计决策的范围：职责分配、协作模型、数据模型、资源管理、架构元素之间的映射、时间绑定决策、技术选择

ADD（结合作业的 ppt 看，这里不细列）

输入：质量属性需求、约束、功能需求

输出：高维度的模块分解、多种合适的系统视图、指定好功能和交互方式的组件集合

Part I	1. Confirm there is sufficient requirements information.
	2. Choose an element of the system to decompose.
	3. Identify candidate architectural drivers.
	4. Choose a design concept (patterns and tactics) that satisfies the architectural drivers.
Part II	5. Instantiate architectural elements and allocate responsibilities.
	6. Define interfaces for the instantiated elements.
	7. Verify and refine requirements and make them constraints for instantiated elements.
	8. Repeat these steps for the next element.

Documenting Architecture

为什么需要文档

- 用于交流
- 便于理解和实施设计决策
- 让设计者能回忆之前某些设计决策
- 训练人的架构设计能力
- 为地理上分布的团队提供方便

架构文档化挑战

- 没有一个普遍接受的软件架构文档化方法
- 对一个大系统进行架构文档化是一项耗费时间和艰难的工作
- 对用于文档化视图的数量和特点没有明确的概念——出于直接
- 对概念和工具理解不够

7 个架构文档化规则

- 从读者的角度书写文档
- 避免不必要的重复
- 避免模糊
- 使用标准
- 记录基本原理
- 保持文档随时更新

- 评审文档

Views And Beyond

结构视图（Structure View）

- 模块视图（Module）：一个模块是一个实现的代码单元；强调静态结构
  - Decomposition
  - Uses
  - Generalization
  - Layered
  - Aspects
  - Data Model
- 组件连接件（Component-connector）：组件是运行时的主要单元，连接件是其交互方式；强调动态（运行时）的结构
  - Pile-and-filter
  - Client-server
  - Peer-to-Peer
  - Service-oriented
  - Publish-subscribe
  - Shared-data
- 分配视图（Allocation）：分配视图描述了软件单元和环境之间的映射关系；可以是动态也可以是静态；强调系统与环境之间的关系
  - Deployment
  - Install
  - Work Assignment
  - Other Allocation

质量视图（Quality View）

Sucurity、Performance、Reliability、Communication、Exception（Error-Handing）

Beyond

- 架构文档信息
  - 文档的结构
  - 视图是怎么被记录的
- 架构信息
  - 系统概述
  - 视图之间的对应关系
  - 基本原理
  - 指引：目录、词汇表、缩写表等。

Evaluating Architecture

为什么需要评估

- 大型项目往往会超期交付或超出预算
- 项目往往需要在后期重新设计
- 架构评估可以解决以上问题
- 越早定位问题修复成本越低
- 普遍接受的最佳实践

- 良好的技术、信息管理手段

什么评估方法是有用的：

能识别风险、敏感点、权衡点

评估模式

设计者、伙伴、外界人士

ATAM（Architecture Tradeoff Analysis Method）

- 目的：
  - 根据质量属性和商业目标考察架构决策
  - 帮助利益相关者提出正确的问题
  - 发现风险
  - 权衡点可以被清楚的定义和文档化
  - 目的不是提供准确的分析
- 参与者：
  - 专业的评估团队
  - 架构的决策者（架构师、场景设计者、项目经历、用户）
  - 架构的利益相关者的代表
- 使用场景：
  - 架构已经被设计好了，可以没有或只有一点代码
  - 需要评估架构选择
  - 需要评估一个已经存在的系统的架构
- 流程：

phase-0：Partnership And Preparation

参与者：评估团队和一些做出关键决策的决策者

输入：架构文档

输出：评估计划（利益相关者列表，何时、何处、怎么评估，评估交付对象、评估报告应有什么内容）

phase-1：

Software Product Line

什么是 SPL

- **定义：**软件产品线是一个共享一个共有的、管理好的特性集合的软件密集型系统，这个特性集是满足了一个特定市场划分需求或任务，并且是从一组预先定义好的共有的核心集开发的。（我的理解就是为高层系统创建了很多可以复用的底层的单元，高层系统通过集成底层单元完成，类似微服务）

- Product=Core Asset(s) + Custom Asset(s)
- **核心资产（core assets）：**被 SPL 中许多（或全部）的产品复用、包含了领域知识、为变化的特性提供了变化点（variation points）、在 SPL 的范围内实现需求
- **自定义资产（custom assets）：**针对一个具体的产品、满足了自定义特点的需求、集成了产品需要的核心资产、实例化了核心资产的变化点
- **特点：**通常有大量变化点、为大规模的重用设计（不适合小型重用）、使用系统层次上的复用（不适用于单个系统内的重用）

为什么使用 SPL（基本上都是重用的优点）

- 大规模下更经济。将核心资产的成本分摊到了更多的产品中。特别是设计、编码和单元测试。
- 更简单。更容易发现哪些资产是商业的核心资产、不同产品之间是如何不同的、通过新的核心资产更新自定义产品
- 为项目工作带来产品特性。从产品中划分出自定义资产，在其他产品中重用这些资产。
- 提高产品质量。核心资产的 bug 只用修改一次。核心资产的 bug 修复会被多个项目共享。产品可以有不同的使用模式。
- 提高了产品的开发速度。对于一个新产品，不用再去开发核心资产。

重用和变化（Reuse & Variation）

- 代码复用是容易的
- 变化是大规模复用的关键点，要使代码更易变化，现在设计，将来复用。

产品线范围（Product Line Scope）

- 定义：产品线范围是用来描述可能构成产品线的产品或产品线能够包含的产品。
- 公司关注的只是一个特定的领域。
- 产品线范围是用来限定这种关注的。

产品线架构（Product Line Architecture）

- 为产品线范围里定义的变化提供支持。范围规定了一些列的产品和功能，PLA 为怎么去处理这些变化做出了实际的决策
- SPL 是在重用一系列架构基础
- PLA 可以看做是 SPL 的核心资产
- PLA 定义了统一（公用）的功能和变化的功能。
- 识别和支持变化点。

产品线生产计划（Product Line Production Plan）

- 定义：生产计划描述了产品是怎么通过核心资产生产出来的。
- Product=Core Asset(s) + Custom Asset(s)，SPL 生产计划类似于其中的“+”是怎么实施的
- 产品线工程框架（SPL Engineering Framework）
  - 领域工程（Domain Engineering）。领域工程是 SPL 中识别和定义 SPL 中的公共点和变化点的过程
  - 制品：Product Roadmap、Domain Variability Model、Domain Requirements、Domain Architecture、Domain Realisation Artefacts、 Domain Test Artefacts
  - 应用工程（Application Engineering）。应用工程是 SPL 中通过重用领域制品和应用产品线的变化来构造应用的过程
  - 制品： Application Variability Model、Application Requirements、Application Architecture、Application Realisation Artefacts、Application Test Artefacts

重用和变化的机制

重用机制：

发现、理解、使用

变化机制：

- 变化是通过变化的形式、变化的实体、和变化的时间来定义的（3 维的一个东西）
- 变化的形式：
  - 包含或忽略元素。（元素种类）
  - 同一个元素的不同数量。（元素数量）
  - 参数化。让产品从一个预定义好的功能中选择。
  - Interface Satisfaction。同一个接口的不同实现、子类、改变行为而不改变接口（我觉得就是多态）
  - 钩子接口。让产品自己提供变化的功能。
  - 反射。让元素发现其所在的上下文并作出反应。
- 变化的实体：
  - 架构层次：高层的抽象结构。框架、发布配置、系统结构。
  - 设计层次：中层的抽象结构。类、接口、方法。
  - 文件层次：底层的具体结构。项目结构、文件、代码
- 变化的时间：
  - 编码期。当你创建了一个代码工作区。
    工作区配置（IDE、版本控制工具）
  - 编译期。当你生成你的目标代码。
    编译配置、宏、文件替换（构建脚本）
  - 连接期。当你创建你的应用。
    连接目标，二进制替换（构建脚本）
  - 初始化。当你的应用开始工作。
    配置文件，注册表设置（已经设计好和编码好的初始化信息）
  - 运行时。在你的应用运行期间。
    用户、管理员、供应商定制（supplier customization）（设计和编码好的界面）

例子：

源代码模块

包括或忽略元素×文件层次×编码期

面对对象继承结构

Interface satisfaction×设计层次×编码期

SPL 的过程和实践

SEI 的产品线方法

- 3 个必须的活动
  核心资产开发（Core Asset Development）、产品开发（Product Development）、管理（Management）
- 29 个实践领域：内部通过不同的方式关联；均有限制；互相结合实现了 SPL 开发和商业价值
  - 软件工程：架构定义、架构评估、组件开发、测试等
  - 技术管理：配置管理、数据收集、技术风险管理等
  - 组织管理：建立商业用例、市场分析、培训等。
- 12（22）个 SPL 模式：展示了实践领域是如何结合、怎么结合来达到一系列不同的

目标

Assembly line、Cold Start、Curriculum、Each Asset、Essentials Coverage、Factory、In Motion、Process、Product Builder、Product Parts、What to Build

SPL 开发和变更控制

- 核心资产的变更会造成所有产品的变更
- 难题：负载的配置识别、不同产品发布的约束、为变化点进行变更、产品线退化
- 解决方法：SCM、Core Asset Change Control Board (CCB)、SPL Road Map、Higher-Quality Core Assets、Emergency “Fake” Core Changes in Custom Assets

Software Design Practices

设计策略（Design Stragegy）

- 设计方法可以分为两类：
  - 转化（transformation step）。修改系统模型的架构，从不同的关注点重新解释架构。需要设计师能将他们的想法在多个不同的特性集中桥接
  - 求精（elaboration step）。不改变关注点，在当前的关注点下重新架构或重新识别设计模型，添加更深入的信息。
- 设计策略
  - 分解方法（de-compositional method）。通常通过一个自上而下的顺序进行设计，将模型设计变为一系列子设计
  - 合成方法（compositional method）。从已经定义的实体或其他形式里构建模型。
  - 组织方法（organizational method）。
  - 模板方法（template-base method）。一个特定领域的问题可以通过一个标准的策略进行解决。

增量设计（Incremental Degisn）

- RAD
  - Rapid Application Development，快速应用程序开发，是指一种以最小幅度的规划并迅速地将原型完成的软件发展方法论。基于原型和迭代，早期原型测试、迭代、重用现存原型、持续集成、快速交付。
  - RAD 模型：业务模型、数据模型、处理模型（进程模型）、应用生成模型、测试转换模型
    https://en.wikipedia.org/wiki/Rapid_application_development
- DSDM：
  - Dynamic System Development Method，动态系统开发方法，是一种敏捷方法，
  - 阶段：
    - 可用性分析（Feasibility Study）。确定 DSDM 方法是否适合当前的项目
    - 商业分析（Business Study）。分析哪些会（将来会）被项目影响的商业过程组织并据此列出需求优先级
    - 功能性模型迭代（Functional Model Interation）。可以被看做开发系统的一个黑盒模型。
    - 设计和构建迭代（Design and Build Iteration）。提供了“白盒”解决方案的组合以及元素原型
    - 实现（Implementation）。提供最终的交付产品，包含了用户培训和评估。

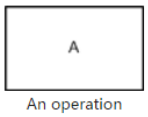
结构化系统分析和结构化设计（Structured System Analysis and Structured Degisn,SSA/SD，与 SSADM 有什么区别？）

(initial -> hierachy) DFD, Transaction analysis, transform analysis , structure chart

- SSA/SD 阶段
  - 初始化 DFD 来提供问题的高层描述
  - 对之前的 DFD 求精，形成一个有层次结构的 DFD，使用数据字典支持
  - 使用事务分析（Transaction Analysis）将 DFD 分解为易于管理的单元
  - 对每个事务的 DFD 进行转化分析（Transform Analysis），用于构建事务的结构图（Structure Chart）
  - 将结构图合并来创建一个基本的实现计划，并为他们添加必要的错误处理、初始化和其他成分。

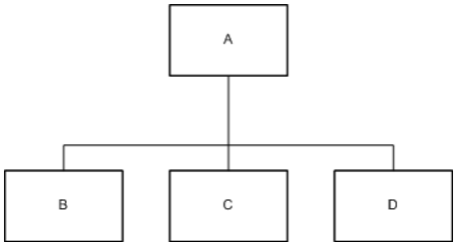
Jackson Strutured Programming

- 为结构化编程提供的一种方法。主要用于控制结构。（告诉我们按照怎样的规则写结构化的程序）
- 元素与结构图（Structure Diagram）规则
  - 操作（Operation）：操作使用一个简单的盒子表示

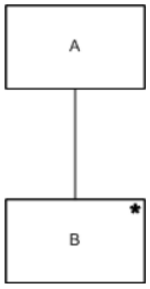


An operation

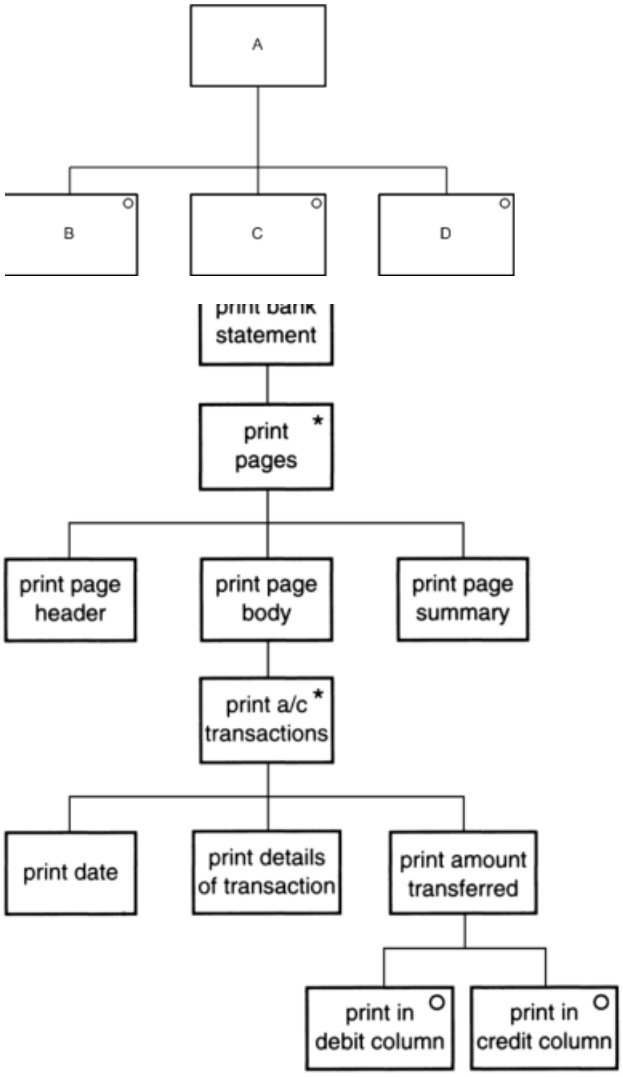
  - 序列（Sequence）：一个操作序列是通过盒子及其连线表示。且操作序列按照从左至右的顺序。表示 A 操作由 B、C、D 操作序列组成（严格按照 BCD 的执行顺序）



  - 迭代（Iteration）：迭代式通过带星号的盒子表示。表示 A 含有一个或多个 B 操作。（类似于结构化编程里的循环）



  - 选择（Selection）：选择类似于序列，但是是用一个带圆圈的盒子表示。A 由一个且只有一个 B、C、D 组成（执行顺序任意）

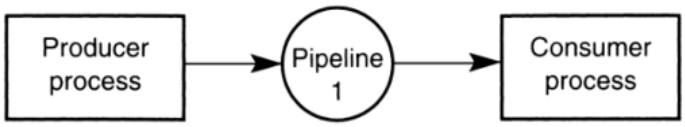


- 设计过程：
  - 为每一个输入和输出的流创建结构图
  - 合并全部结构图生成一个程序结构图
  - 列出程序所有的操作，并且将这些操作分配打具体的程序结构图中
  - 在没有任何特定的决策环境下将程序转化为文字（？）
  - 为每一个迭代和选择的操作添加条件（conditions）

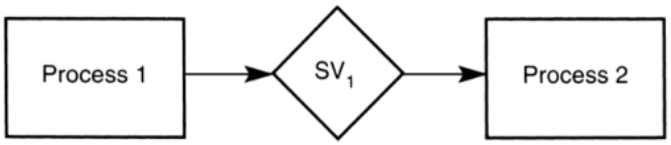
Jackson System Development

(Entity-)Structure diagram; Data-flow stream, State-vector, Modeling Network, Implementation

- 实体结构图（Entity Structure Diagram，类似于 Structure Diagram）。表示系统中各实体之间的关系及其动作。
- 数据流（data-flow stream）。两个相连的实体之间通过管道、异步的方式传递信息。程序 A（产生数据流的程序）程序主动提交数据给程序 B



- 状态向量（state vector）。程序 B（读取状态向量的程序）程序读取程序 A 产生的状态向量信息。



- 数据流链接和状态向量链接的差异在于主动的程序不同：在数据流中，产生数据流的程序 A 是主动程序，程序 A 依其选定的时间主动将数据发送给程序 B；在状态向量中，产生状态向量的程序 A 是被动程序，接收状态向量的程序 B 定期去读取程序 A 的状态向量。简单来说，数据流链接是消息传递的抽象化，状态向量是轮询（由一个程序主动定期读取其他许多程序的数据）的抽象化，状态向量也是数据库查询的抽象化。(from Wikipedia)
- 设计过程：
  - 建模阶段（modeling state）：分析，对问题进行分析和建模（基于组成的实体和他们的功能）。更关注创建一个问题的黑盒模型，不用关注具体的解决办法。实体分析，创建实体结构图（Entity Structure Diagram，类似于 Structure Diagram）。
  - 构建网络（network stage）：设计，对整个系统间的实体或者实体与外部系统添加详细的交互方法。
    - 初始化模型。对建模阶段生成的模型进行连接，形成初步的系统整体的模型
    - 求精。
      - P1：添加交互方式，识别会影响系统和外部系统不会产生的事件，并添加生成事件的过程。
      - P2：增加额外信息。添加生成整个系统需要的过程和数据流。
      - P3：确定系统时序。过程是怎么被调度的，依照怎样的规则，如何确定这些过程的基本层次的
  - 实现（implementation）：实现，将抽象设计映射为“物理”设计。当前相对抽象的解决方案模型是怎么映射到物理系统上。决定模型所有的状态向量并且在物理设计上实现。