1. Software Architecture

什么是软件架构?

- SEI：程序或计算系统的软件架构是这个系统的**结构**(Structure)，包括了其内部的**软件元素** (Software Elements)、软件元素的**外部可见属性**(Properties)及元素之间的**关系**(Relationship)。

- IEEE：一个系统的基本组织，包括了其**所有组件，组件之间的关系**、**环境**以及**指导该系统的设计(Design)和演化(Evolution)的原则(Principle)** 。

架构师做什么?

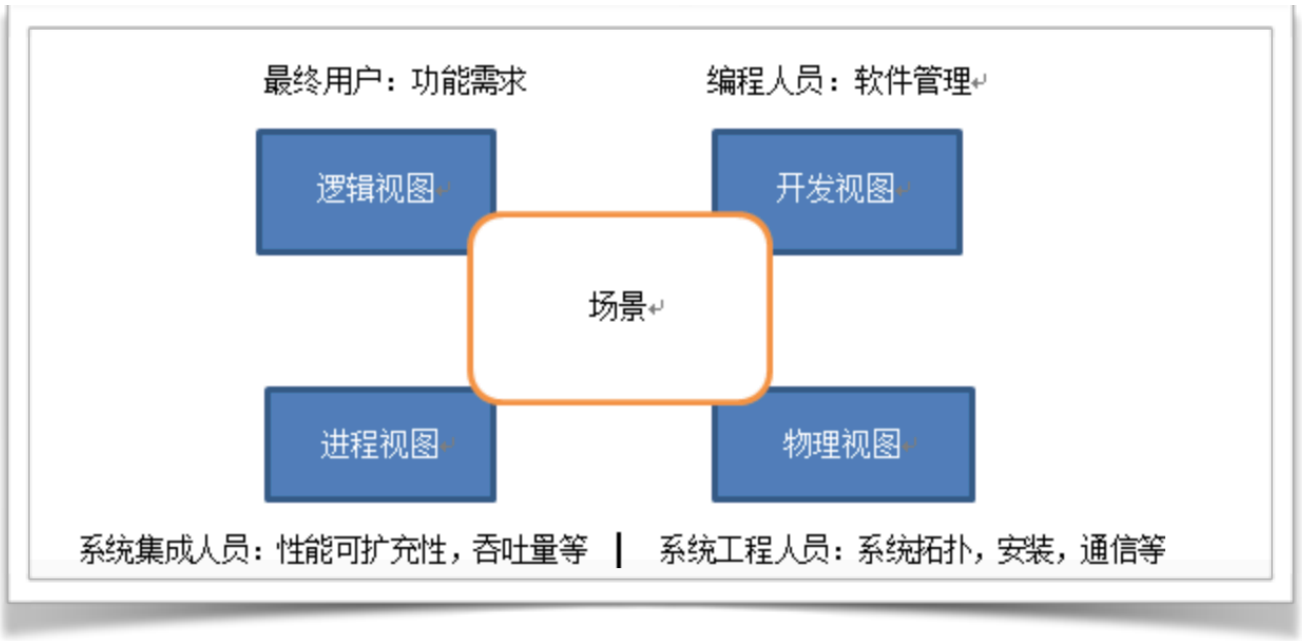
- 倾听客户，理解完整的需求
- 仔细检查可行性
- 形成一个实用的架构设想并制成蓝图
- 审查架构并确保与计划一致
- 引导设计中的变化、危机和歧义

架构设计从哪里来?

- 需求：质量需求（NFRs-非功能性需求, ASRs-架构重要需求）
- 系统的利益相关者（管理、市场、终端用户、维护）
- 开发团队
- 架构师
- 技术环境

架构视图

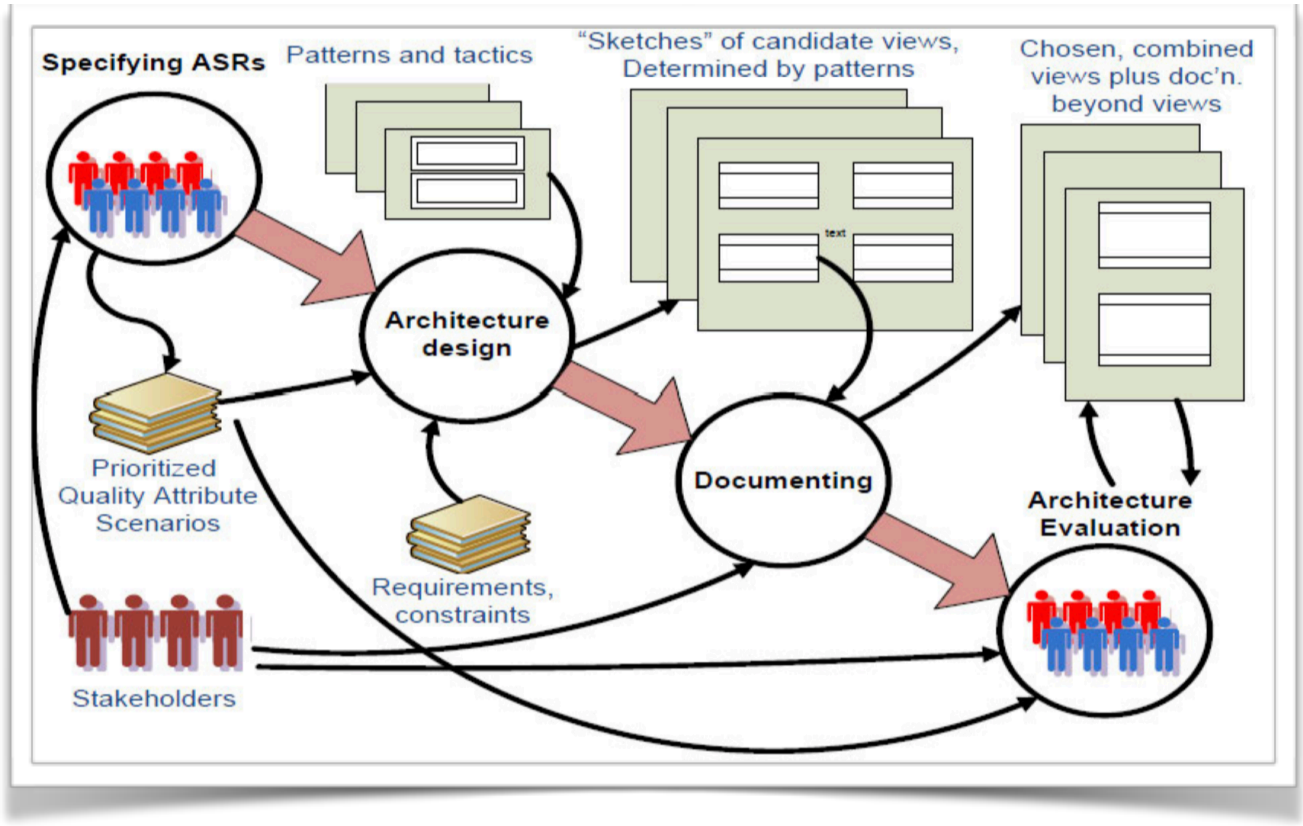
- 4+1视图模型：每一个视图只关心系统的一个侧面，5个视图结合在一起才能反映系统的软件架构的全部内容。



    - **逻辑视图(Logical View)**：从对象角度，构建对象模型，用以确立逻辑分层、模块划分、模块功能、模块间依赖关系等。其中，模块功能，既包括可见的业务功能，也包括不可见的系统功能（日志、权限、事务等）。
    - **过程视图(Process View)**：从过程角度，描述系统的并发和同步设计。旨在解决进程、线程、并发、同步、通信等方面的问题；
    - **物理视图(Physical View)**：负责从部署角度，描述软硬件的映射关系；，以及系统在分布/部署上的设计。旨在解决系统安装、系统部署、网络分布等问题。
    - **开发视图(DevelopmentView)**：从开发角度，描述软件在开发环境下的静态组织（程序包、应用的统一框架、引用的类库、SDK和中间件等），并规范和约束开发环境的结构。
    - **架构用例(Use case)**：通常也叫 Scenarios，负责从用户角度，识别业务需求，描述业务场景，是架构设计的起点和终点。
- 如何选择用什么视图?
    - 从业务目标出发
    - 涉众和其关注点
    - 哪些视图与这些关注点有关
    - 按优先级排序，有时会合并多个视图

架构活动(Activities)与过程(Process)

- 架构活动
  ① 创建系统的业务案例，理解需求
  ② 构造并选择架构
  ③ 与涉众交流架构
  ④ 分析或评估架构
  ⑤ 实现架构
  ⑥ 确保与架构一致
- 架构过程



  ① 发现ASRs
  ② 架构设计
  ③ 视图和文档
  ④ 架构评估
- 架构生命周期 Lifecycle
  ① 架构分析
  ② 架构评估

③ 架构实现
④ 架构合成
⑤ 架构维护

软件架构知识域

- 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
  - 设计的过程
  - 设计中使用的技术
- 关键点：并发、事件控制和处理、分布式、异常处理、交互系统、持久化
- 软件结构和架构:
  - 架构的结构和视点
  - 架构的模式和风格
  - 软件设计方法：架构方法、设计方法

2. Quality Attributes

软件需求

Functional requirements, Quality requirements (NFRs、ASRs), Constraints

- 功能需求
  - 描述了如何给利益相关者产生价值
  - 描述了系统的行为和职责
- 质量需求
  - 功能需求之上需要提供的整个系统需要的特征
  - 用于检测功能需求和整个产品的质量
- 约束
  - 是一个零自由度的设计决策

质量属性

- Modeling quality attribute scenarios:
  - 刺激源、刺激、制品、环境、响应、响应度量
- Availability, Interoperability, Modifiability, Performance, Security, Testability, Usability, X-ability...

质量属性

Availability：可用性。和整个程序的可靠性（Reliability）有关，是系统修复故障的能力。

- fault：错误。系统内在的缺陷，特定条件下才会触发的
- failure：故障。由fault导致的结果
- error：failure和fault之间的状态

Interoperability：互操作性。描述多个系统通过接口交换信息的属性，包括交换和正确推断信息。
Modifiability：可修改性。用于评估一个更改所消耗的时间和金钱。
Performance：性能。关于时间和软件系统满足时间需求的能力。响应时间=处理时间和等待时间
Security：安全性。衡量系统在向合法用户提供服务的同时，阻止非授权使用的能力。
Testability：可测试性。通过测试揭示软件缺陷的容易程度。
Usability：易用性。用户去完成一个特定任务的容易性程度和系统所提供的用户支持的种类。
X-ability：可变性、可移植性、可伸缩性、可部署性、可复用性...
Mobility 移动性，解决平台之间的移动和支持（大小、显示器类型、带宽、电池等）
Safety 安全性，与security相比，不是有意为之，关注自身安全，防止偶然可能造成破坏的对象。
Development Distributability：支持分布式软件开发
Variability：可变性，特殊的可修改性。
Portability：易于做改变，用于另一种平台
Scalibility 可伸缩性：是一种对软件系统计算处理能力的设计指标
Deployability 可部署性

-

- Tactics for quality attributes

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

架构重要需求(Architecturally Significant Requirements)

- ASR
  - 更难更重要、对系统有更重大影响的质量属性需求：重要且非功能性
- 如何收集和识别ASRs?
  - 从需求文档中发现
  - 访谈利益相关者
  - 理解业务目标
  - 效用树

3. Architecture Patterns

架构模式

Context, Problem, Solution: elements + relations + constraints （要知道每个架构模式的元素 关系 约束）

- 架构模式
  - 是一组适用于反复出现的设计问题的设计决策的集合
  - 是在不断尝试中找到的
  - 可以重用，描述一种架构的类别
  - 建立了上下文、问题、解决方案之间的关系
- 包含：
  - 上下文：现实生活中反复出现的、导致问题的情况
  - 问题：给定上下文中不断出现的问题

常见架构模式

  - 模块模式（Module Patterns）
    - 分层模式（Layered Patterns）——layer
  - 连接件模式(Component-Connector Patterns)
    - 代理模式（Broker Pattern）——Client,Server,Broker
    - MVC模式（Model-View-Controller Pattern）——M、V、C
    - 管道过滤器模式（Pipe-and-Filter Pattern）——pipe, filter

        - 客户端服务器模式（Client-Server Pattern）——C、S、request/reply connector
        - 点对点模式（Peer-to-Peer Pattern）——peer、request/reply connector
        - 面向服务的模式（Service-Oriented Pattern）
        - 发布-订阅模式（Publish-Subscribe Pattern）——C&C component with port
        - 共享数据模式（Shared-Data Pattern）——shared-data store、data accessor component、data reading and writing connector
    - 分配模式（Allocation Patterns）
        - 映射-规约模式（Map-Reduce Pattern）——map reduce infrastructure
        - 多层模式（Multi-Tier Pattern）——tier

Layered Pattern与 Multi-Tier Pattern：

- Layer通常指的是**逻辑**上的分层，对于代码的组织，例如：我们通常提到的“业务逻辑层，表现层，数据访问层”等等。
- Tier通常指的是**物理**上的分层，由执行同样功能的一台（或者一组）服务器定义。而
- Layered是**Module Pattern**，主要关注**静态部分**，Multi-Tier pattern是**Allocation Pattern**，关注**上下文环境**。
- Tier指代码运行的位置，多个Layer可以运行在同一个Tier上的，不同的Layer也可以运行在不同的Tier上，当然，前提是应用程序本身支持这种架构。

分层模式

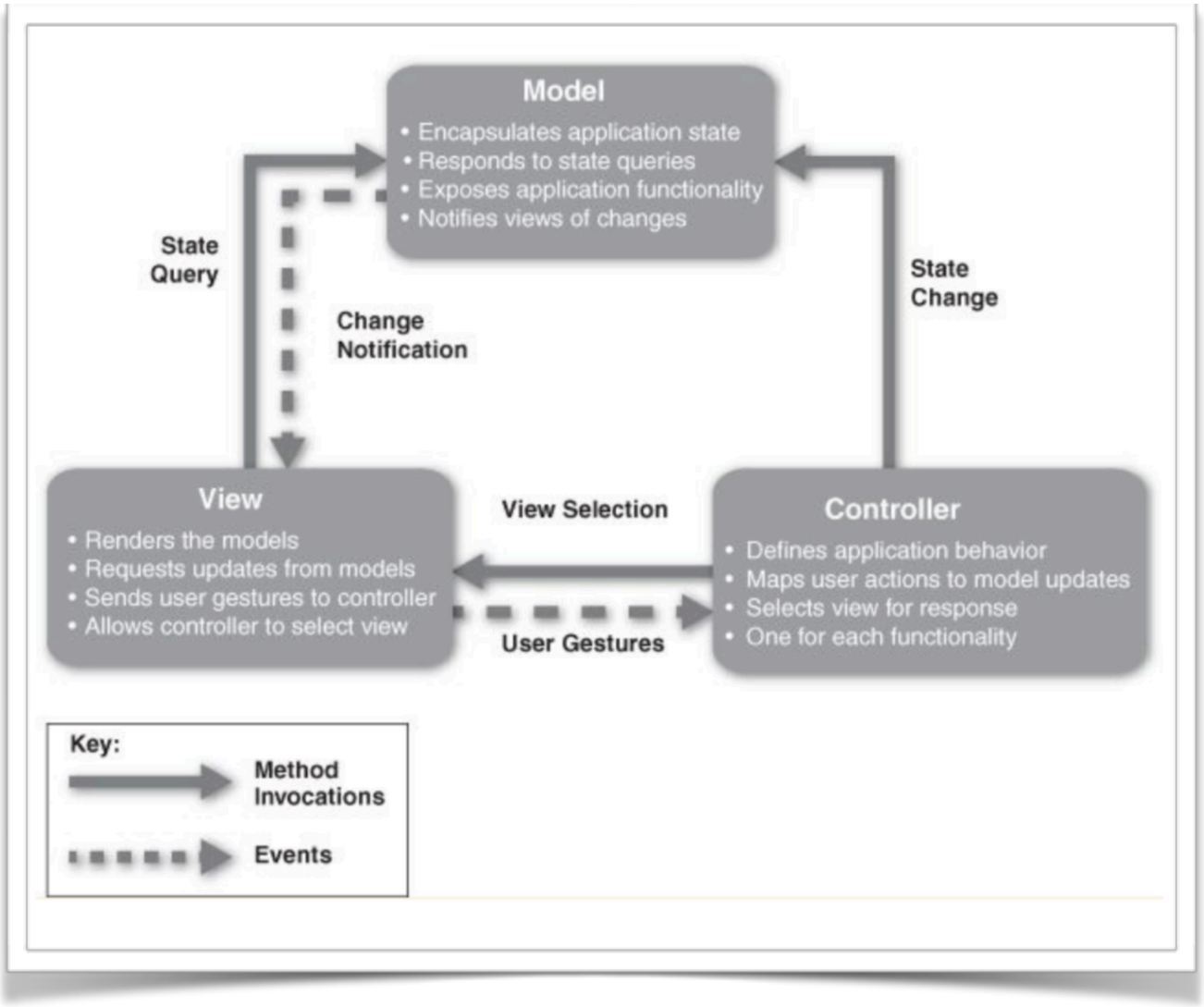
- 描述：
    - 只能单向调用，下层不能调用上层（也不能形成环）
    - 应该指定层间使用规则：如上层可使用任意下层 or 只能相邻层调用
- 缺点：
    - 增加了预付成本和项目复杂度
    - 造成性能下降
- 例子：网络传输7层OSI

代理（Broker）模式

- 描述
    - Broker作为客户端和服务端的中间人
- 元素：
    - Client：服务请求方
    - Server：服务提供方
    - Broker：为client定位一个合适的可以满足它的请求的server，将请求转发给server，并将结果传回client

    - Client-side Proxy：client通过它和Broker交流，负责将消息的编码、发送和解码
    - Server-side Proxy：…
- 约束：
    - Client和Server之间必须通过Broker交互
- 缺点：
    - Broke增加了一个间接层，可能造成两端通信延迟和交流瓶颈
    - Broke可能造成单点故障（即Broker出问题，整个系统都会无法运行）
        - Broke增加了预付成本
        - Broke可能会成为安全攻击的目标
        - Broke难以测试

MVC模式



- 描述：

    - Model：代表应用的数据或状态，包括（或提供接口）应用逻辑
    - View：UI，提供展示 & 允许某些形式的用户输入
    - Controller：管理model和view的交互，将用户的操作转换到model/view的变化上
- 联系：“通知关系”联系了model、view和controller的实例，通知相应元素状态的改变
- 约束：
    - model不能直接与controller交互，controller能单向通知model改变状态
- 缺点：
    - 对于简单的UI，MVC带来的复杂度是不值得的
    - MVC这种抽象可能不适用于一些UI工具集

模式 vs. 战术（Patterns vs. Tactics）

⑦ 战术比架构模式简单
⑧ 战术是一种设计决策，而模式是一组设计决策的集合
⑨ 它们都是架构师的基础工具
⑩ 战术是设计的建筑基石
⑪ 大多数模式都包含几个不同的战术。例如分层模式包含增加内聚，降低依赖的战术
⑫ 大多数模式也会包含其他模式

3. Designing Architecture

通用的设计策略

Abstraction, Decomposition, Divide & conquer, Generation and test, Iteration, Reuse

- 抽象
- 分解
- 分治
- 重用
- 生成和测试
- 迭代：逐步求精

属性驱动设计（ADD）

- 步骤
  ① 选择一个部分来设计

② 整理那个部分有关的所有ASRs
③ 为那个部分创建一个设计并测试
④ Input to and outputs of ADD

8-step process: 1. confirm requirements, 2. choose an element to decompose, 3. identify ASRs, 4. choose a design satisfying ASRs, 5. instantiate elements & allocate responsibilities, 6. define interface, 7. verify & refine requirements, 8. repeat step 2-7 until all ASRs satisfied

① 确认需求
② 选择一个系统元素分解
③ 为该元素确认ASRs
    - H, M, L的二维数组：（对利益相关者的重要性，对架构实现的影响）
    - 即（Importance, Difficulty）
④ 选一个符合ASRs的设计理念
    ④1. 确认设计关注点
    ④2. 为次要关注点列出不同的模式和策略
    ④3. 选择合适的模式和策略
    ④4. 决定ASRs和模式策略之间的关系
    ④5. 得到架构视图
    ④6. 评估并解决不一致问题
⑤ 列出架构元素并分配职责
⑥ 定义元素接口
⑦ 确认需求，为元素构造约束
⑧ 重复以上步骤直到满足所有的ASRs

4. Documenting Architecture

Views and Beyond

Views

- 三类风格
    - 模块风格
        - 如何被结构化为一系列**实现单元**的集合

    - 组件-连接器风格
        - 如何由一系列拥有运行时表现和交互的元素组织起来的
    - 分配风格
        - 如何与其环境中的非软件结构关联
- Style vs. Patterns vs. Views
    - Style:：关注组件和组件之间的交互和职责限制。更关注架构方法，不包含具体的上下文和问题。
    - Pattern：体现软件系统基本的结构组织模式。关注上下文问题，以及如何解决。
    - View：是系统中特定类型的元素及其关系的表示。不同的视图支持不同的目标和用户，强调不同的系统元素和关系，同时在不同程度上表示不同的质量属性
- 结构视图（每个Style要写四个视图！！！）
    - 模块视图：强调静态结构
        - 分解视图
        - 使用视图
        - 泛化视图
        - 分层视图
        - 切面视图
        - 数据模型视图
            - Decomposition view
            - Uses view
            - Generalization view
            - Layered view
            - Aspects view
            - Data model view
        -
    - 组件-连接器视图：强调动态（运行时）结构
        - 管道过滤器视图
        - 客户端服务器视图
        - 点对点视图
        - SOA视图
        - 发布订阅视图
        - 数据共享视图
        - 多层视图

- Pipe-and-filter view
- Client-server view
- Peer-to-peer view
- Service-oriented architecture (SOA) view
- Publish-subscribe view
- Shared-data view
- Multi-tier view

    - 分配视图：强调软件系统与环境之间的关系
        - 部署视图
        - 安装视图
        - 实现视图
        - 工作分配视图
        - 其它分配视图

- Deployment view
- Install view
- Work assignment view
- Other allocation views

- 质量视图
    - 安全视图
    - 性能表现视图
    - 可靠性视图
    - 交互视图
    - 异常视图/错误处理视图
    - …

文档化视图

选择视图的三个步骤

Step1：创建一个StakeHolders的视图表格：不同的StakeHolders对不同视图有不同的关注点

Step2：合并视图

Step2.1: 确定上表中的边缘视图

Step2.2: 将每个边缘视图与其它视图结合

e.g.. 不同的C&C视图之间、部署视图和SOA视图之间、分解视图和工作分配、使用或分层视图之间

Step3：确定优先级和排序：80/20原则

Beyond views

Beyond views: documentation info & architecture info (mapping between views)

- 架构文档信息
  - Roadmap
  - 版本控制
- 架构信息
  - 系统概述
  - 视图之间的映射
  - 基本理论
  - Guidance：目录、术语表、缩略词表等

Documentation package: views + beyond

架构 = 不同的Views + Documentation Beyond Views

5. Evaluating Architecture

ATAM(Architecture Tradeoff Analysis Method)：架构权衡分析方法

ATAM的涉众：

设计者、同事、外界人士

ATAM的过程：

- **Phase 0：准备阶段**
  - 参与者：评估小组组长和关键的项目决策者
  - 输出：评估计划——谁、什么时间、提供什么样子的评估报告
- **Phase1：评估(1)**
  - 参与者：评估小组和项目设计决策者（肯定包括了项目经理和架构师）

    - 输出：架构简要展示、业务目标、质量属性和相关场景、效用树、风险和非风险点、权衡点
        - **Step1: 介绍ATAM方法（评估小组组长）**
        - **Step2: 介绍商业动机（项目经理或系统客户）**
        - **Step3: 介绍架构（首席架构师）**
        - **Step4: 识别使用的架构方法（评估小组）**
        - **Step5: 生成Utility Tree（评估小组和项目设计决策者）**
            - 决定性的一步
        - **Step6: 分析架构方法（评估小组）**
            - 确保方法是正确的
            - 获得风险点、非风险点、敏感点和权衡点列表
- **Phase2：评估(2)**
    - 参与者：评估小组、项目设计决策者和架构涉众
    - 输出：从涉众群体获得的一个优先级场景列表、风险主题和商业动机
        - **Step1: 介绍ATAM方法和之前的结果（评估小组组长）**
            - 重复以确保涉众也知道方法并回顾分享之前2～6步的结果
        - **Step7: 头脑风暴、场景划分优先级（评估小组问涉众）**
            - 与质量属性效用树进行比对
        - **Step8: 分析架构方法（评估小组、架构师）**
            - 使用新产生的优先级靠前的场景、架构师解释与之相关的架构决定
        - **Step9: 展示结果（评估小组）**
- **Phase3：后续工作 跟进**
    - 参与者：评估小组和主要涉众
    - 输出：最终的评估报告

6. Software Product Lines

软件产品线Software Product Lines, **SPL** (Engineering)

什么是产品线?
软件产品线是指具有一组可管理的公共特性的软件密集性系统的合集，这些系统满足特定的市场需求或任务需求，并且按预定义的方式从一个公共的核心资产集开发得到。

Product = core assets + custom assets

- Core Assets:
  - 在 SPL 中被大量产品复用
  - 提供领域内的知识和经验，满足领域需求
  - 提供大量variation points以适应不同特性
- Custom Assets:
  - 集成产品所需的core assets
  - 满足用户特定的，SPL范围外的需求
  - 将core assets中的可变化的部分实例化

SPL Engineering = Core Asset Development(Domain Engineering)+Product Development(Application Engineering)

Reusability and Modifiability ？ ？ ？

产品线架构Product Line Architecture

Reuse 重用:

- 代码复用很容易，已经是开发者想要的了，不再仅是almost，所以复制之后不用再修改
- Reuse机制的核心是:
  - find code:源码控制工具或查找库/API
  - understand code:在workspace中阅读源码或阅读库/API文文档
  - use(invoke) code

Variation 变化:

Variation: forms of variation * software entity varied * binding time

- 可变性机制 Variation Mechanisms:
  - **Forms of Variation** 6种变化的形式?
    - 包括或省略元素

        - 改变元素的数量
        - 不同的实现满足一个统一的接口
        - 参数化。让客户从预先设定好的功能范围中选择
        - “Hook”接口。客户提供变化的功能
        - 反射。在程序运行过程中感知外部的变化，进行自身调节
    - **Software entity varied** 3个变化的位置?
        - 架构级别
        - 设计级别
        - 文件级别
    - **Binding time** 5个变化的时间?
        - Coding-time — workspace配置
        - Compile-time — 编译器设置
        - Link-time — 连接选项
        - Initialization-time —配置文件
        - Run-time —用户/管理员等定制化

Architecture: variation points? ? ?

SPL Practice Areas and Patterns

29 practice areas and 22 patterns

29个实践领域 Practice Areas (分为以下三类)：

- 软件工程
- 技术管理
- 组织管理

这3个区域是一个简单的分类：有用但有限

22个实践模式 Practice Patterns：

Curriculum、Essential Coverage 、Factory、Product Parts 、Each Asset 、Cold Start Pattern

7. 其它

Architecture vs. Design

Architecture vs. Structure

题目

1. **Where do software architecture come from? List five possible sources of software architecture.**
   - 需求：质量需求（NFRs-非功能性需求, ASRs-架构重要需求）
   - 系统的Stakeholders（管理、市场、终端用户、维护）
   - 开发团队
   - 架构师
   - **技术环境**

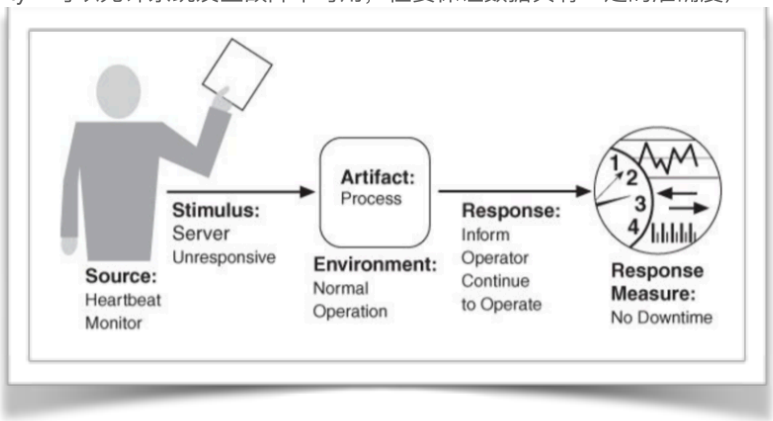
2. **What distinguishes an architecture for a software product line from an architecture for a single product?**

软件产品线架构 vs 单个产品架构

1) 软件产品线架构和单个产品架构的主要不同在于**关注点转移：**从单独的产品项目到一个产品线的项目，体现了从特定的项目开发到特定业务领域产品的愿景。产品线关注产品的特征，而单个产品架构更关注项目本身。
2) 产品线具有可重用和可变性两大特征。
   ① 产品线中的重用与单个产品架构中代码的重用相比，还包括了需求、业务等，几乎已经是开发者想要的了，而不用像单个产品架构中那样复制之后再修改；
   ② 产品线架构还拥有可变性的特点，除了像单个产品架构那样定义正常功能之外，还定义了可改变的功能，可以识别并支持变化点。
3) 总的来说，软件生产线架构与单个产品架构相比，可以减少成本、快速上市、减少风险、提高质量，更容易适应市场。

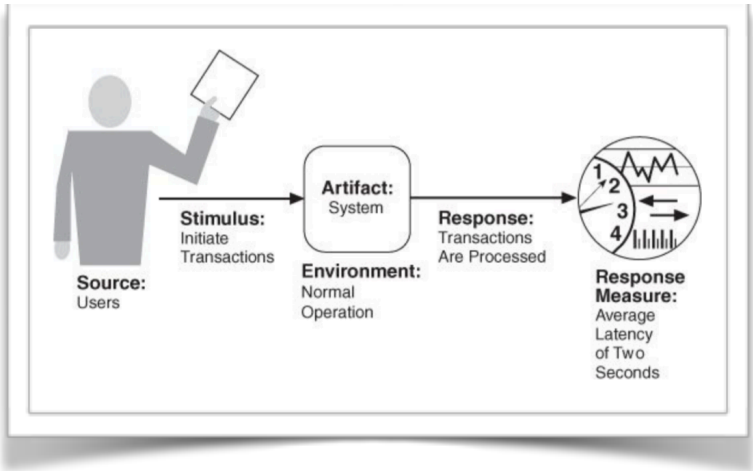
3. How to model quality attribute scenarios? Graphically model two quality attributes in "stimulus-response" format: availability and performance.

1) 步骤：
   ① 针对特定质量属性，找到特定场景
   ② 找到刺激源、刺激、制品、环境、响应，确定相应的响应度量，作出质量属性场景模型。
2) Graphically models:
   ① Availability（可用性：用来描述系统从故障中恢复的能力，可以允许出错、但要保证系统可用；可靠性Reliability：可以允许系统发生故障不可用，但要保证数据具有一定的准确度）



-
- 下面是我编的另一个例子
- 刺激：数据库运行故障
- 刺激源：数据库内部
- 制品：系统进程
- 环境：运行时
- 响应：记录错误：日志；恢复：使用备份数据恢复、暂时使用其他数据库、及时恢复该数据库工作
- 响应度量：数据恢复时间、系统重新恢复工作的时间

   ② Performance性能（负载 容量 实时）



-
- 下面是我编的另一个例子
- 刺激：大量数据更新
- 刺激源：用户
- 制品：系统进程
- 环境：运行时
- 响应：数据被写入数据库
- 响应度量：数据更新的总耗时

4. **Describe relationships between architecture patterns and tactics in architecture design. Give name of any four tactics that can be used during architecture design and describe the purpose of each of them.**
   1) 架构模式和战术在架构设计中的关系
      ① 战术比架构模式简单
      ② 战术是一种设计决策，而模式是一组设计决策的集合
      ③ 它们都是架构师的基础工具
      ④ 战术是设计的建筑基石
      ⑤ 大多数模式都包含几个不同的战术。例如分层模式包含增加内聚，降低依赖的战术
      ⑥ 模式之间也存在包含关系
   2) 4种战术及其作用
      ① ping/echo 错误检测（可用性）
      ② 心跳 错误检测（可用性）
      ③ 事务 错误预防（可用性）
      ④ 检查点/回滚 错误恢复（可用性）

5. **Briefly describe the general activities involved in a software architecture process.**

   自己写的不知道对不对

   软件架构包含以下几个主要过程：

① 发现ASRs
② 架构设计
③ 视图和文档
④ 架构评估

其中主要包含以下几个一般活动：

① 创建系统的业务案例
② 理解需求
③ 构造并选择架构
④ 与涉众交流架构
⑤ 分析或评估架构
⑥ 实现架构
⑦ 确保与架构一致

6. Explain the context, benefits and limitations of Broker Architecture Pattern.

- Context：
  - 由多个远程对象组成的分布式系统，这些对象同步或异步交互。
  - 异构环境
- Benefit：
  - 位置透明
  - 组件可更改、可扩展性提高
  - 代理系统移植性提高
  - 代理系统可互操作性提高
  - 可重用
- Limitation（因为不知道limitation是指局限性还是缺点，所以都答了）
  - 约束：Client和Server之间必须通过Broker交互
  - 缺点：
    - Broker增加了一个间接层，可能造成两端通信延迟和交流瓶颈
    - Broker可能造成单点故障（即Broker出问题，整个系统都会无法运行）
      - Broker增加了预付成本
      - Broker可能会成为安全攻击的目标
      - Broker难以测试

7. Why should a software architecture be documented using different views? Give the names and purposes of 4 example views.

1) 为什么要用不同的视图?

① 不同的视图支持不同的目标和用户，强调不同的系统元素和关系。
② 在不同程度上表示不同的质量属性。
③ 所以，如果要将软件架构文档化，必须要使用不同的视图来从不同的视角对架构进行描述。（我说的:)）

2) 结构视图——模块视图、组件—连接件视图、分配视图；质量视图——安全性视图、性能视图……

- 结构视图（描述系统结构）
  - 模块视图：强调系统静态结构，代码的建设蓝图、变化—影响分析、计划增量量开发。
  - 组件—连接件视图：强调动态，展示系统是如何运行的、指导运行时元素结构和行为的开发、帮助分析运行时系统的质量属性。
  - 分配视图：强调周围环境，展示系统如何部署、安装?
- 质量视图（描述系统质量需求？）

8. Briefly describe the fundamental principles of SOA and discuss the impact of SOA on quality attributes like interoperability, scalability and security.

今年考察**MicroService**

微服务架构风格是一种 将一个**单一应用程序** 开发为**一组小型服务程序**的方法，每个服务运行在自己的进程中，服务间通讯采用**轻量级通信机制**，这些服务围绕业务能力构建，并可以通过自动部署机制独立部署。
（分布式架构，也是面向服务架构的扩展）

因为服务之间独立，所以松耦合，所以有较强的：可修改性、可复用性
互用性 interoperability 好像不太行：虽然是面向服务架构的扩展，但它少了soa很重要的特性——异构互操作性。
可扩展性强 scalability 服务之间关联弱（但是这个可扩展性 项目到了很大的时候就会很复杂 因为服务之间的依赖过多
安全性 security（强调恶意）应该还不错2333

微服务架构的核心：

API网关（全部客户端的单一入口点，针对不同客户端提供出不同的API）、

熔断器（防止程序不断地尝试执行可能会失败的操作，也可以用来诊断错误是否已经修正，有闭合（可以直接引起方法调用）、断开（返回错误响应）、半断开状态（允许一定数量的请求可以调用服务，如果成功则说明修复、变回闭合状态；否则熔断器就变为断开状态））

微服务架构的特点：服务颗粒化、责任单一化、运行隔离化、责任自动化
微服务架构的挑战：运维要求高（微服务数量多）、发布复杂度（部署环境多样性）、部署依赖强（各个服务之间调用关系复杂）、通信成本高（跨进程调用比进程内调用消耗更多资源）

9. Describe outputs generated from each phase of ATAM process.

Phase0：参与者和准备阶段

输出：评估计划：谁，什么时间，提供什么样子的评估报告

Phase1：评估(1)

输出：架构简要展示、业务目标、质量属性和相关场景、效应树、风险和非风险点、敏感点、权衡点

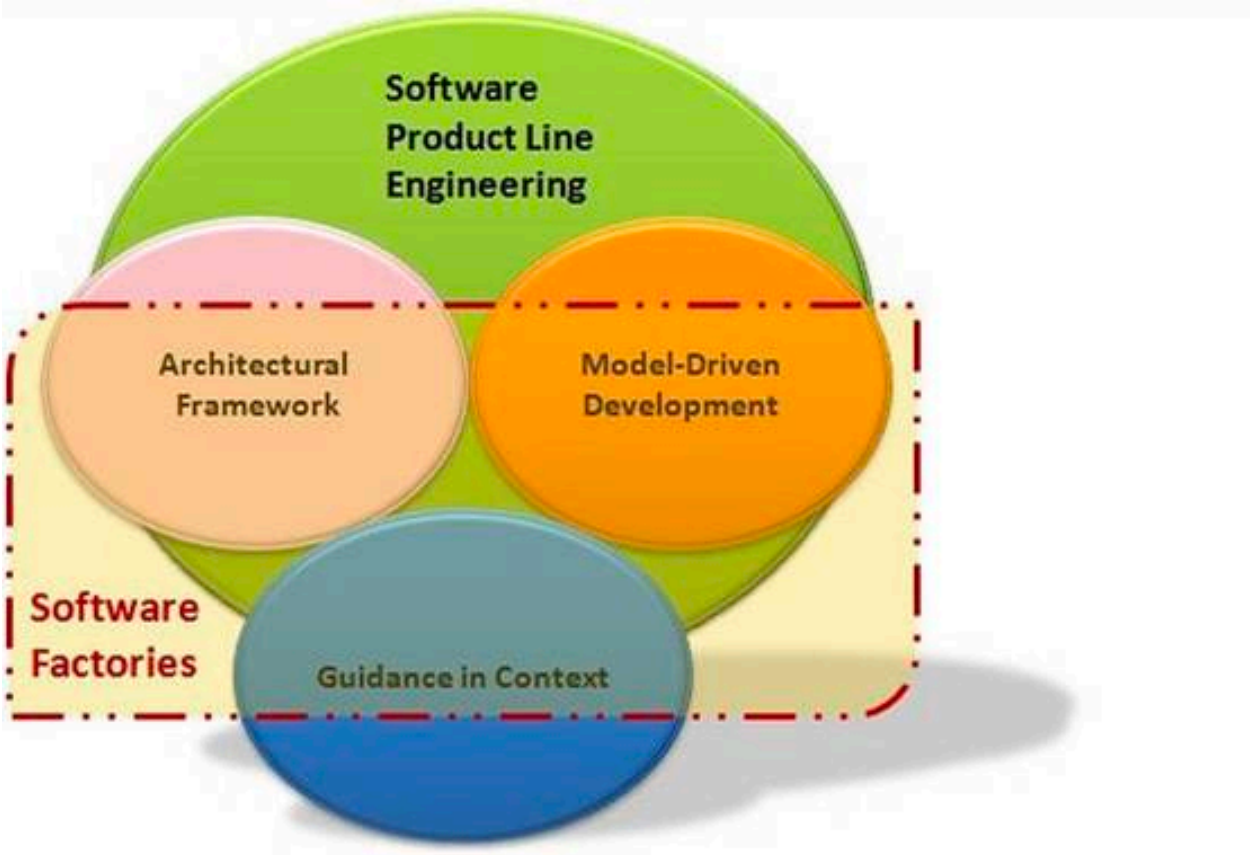
Phase2：评估(2)

输出：从涉众群体获得的一个优先级场景列表】风险主题和商业动机

Phase3：后续工作 跟进

输出：最终的评估报告

10. How do software product lines and model driven architecture archive a high reusability? Compare and discuss the commonalities and difference between two architectural approaches from various perspectives.（我不会我哭了）



- SPL：
  - 可复用的资产非常广，包括以下几点
    - 需求
    - 架构设计

        - 元素：元素复用不只是简单的代码复用，它旨在捕获并复用设计中的可取之处
        - 过程、方法和工具：有了产品线这面旗帜，企业就可以建立产品线级的工作流程、规范、标准、方法和工具环境，供产品线中所有产品复用。如编码标准就是一例。
        - 测试：采用产品线可积累大量的测试资源，即在考虑测试时不是以项目为单位，而是以产品线为单位，这样，整个测试环境都可以得到复用。

- MDA鼓励重用:
    - 模型
    - 最佳实践模型和模型转换映射
    - 创建应用程序家族—软件产品线
    - PIM平台独立模型映射到不同的PSM平台特定模型实现重用
    - 将可重用性作为目标
- 相同点：
- 不同点：
    - MDA将模块作为core assets,进行复用

11. 分析设计题