体系结构

Chapter 1

- architecture & structure
  - architecture
    - 动态关系 定义了一个组件的职责
    - 组件之间的相互依赖和交流关系(数据传输和流控制)
    - 定义了NFR(no functional requirements) 系统运行好不好(技术 商业 质量属性)
    - 隐藏实现细节

  - structure： 静态组合关系 是一个architecture的具体实现方案

- 设计策略 Design Strategies
  - 分解：将复杂事务分解，实现每一部分，降低复杂度
  - 抽象：突出显现问题
  - 逐步渐进、分而治之：逐步解决问题
  - 生成测试：生成测试，看是否符合需求
  - 迭代
  - 重用element

- **经典4+1视图模型** 4+1 View Model(4个视图，架构用例，结合4个视图)
  - 逻辑视图：描述了重要架构元素及其之间的**关系**
  - 进程视图：描述了元素之间的**交流和并发**问题 强调了动态运行时的关系
  - 物理视图：描述了主要进程和组件与硬件之间的映射关系
  - 开发视图：关注软件内部的组织。对软件系统各个组件有一定了解，能够展现一个系统的开发过程以及在该过程中对开发团队的组织和管理
  - 架构用例：通常也叫scenarios，描述了架构需求，跟不止一个视图相关

- 软件设计过程 Software Design Process
  - 需求规格说明
  - 体系结构设计
  - 实现细节**逻辑**化
  - 详细的设计决定
  - 具体开发过程

- 架构活动 Architecture Activities
  - 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

- 架构过程 Architecture Process
    - 识别 ASR
    - 架构设计
    - 架构文档化
    - 架构评估

- 架构生命周期 Architecture Lifecycle
    - 架构分析
    - 架构评估
    - 架构实现
    - 架构维护

- 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

Chapter 2 Quality Attributes

- 功能性需求
    - 系统行为，系统更够提供的价值
    - 系统完成他必须完成的工作
    - 功能性需求很大程度上取决于structure

- 质量属性 非质量属性 架构需求(可以分成可见的(performance security availability usability)；不可见的(modifiability Protability reusability testability))
    - 在功能性**需求之上**提供的系统特性
    - 衡量了**质量**上的合格程度
    - 对功能实现所需要的方式和structure进行限制

- 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

- Availability 可用性
    - 最重要的需求
    - 一般情况下用系统可用时间比例来描述
    - 和系统可靠性相关
    - 系统不可用的衡量标准：发现错误的时间 纠正错误的时间 重启系统的时间
    - 高可用性方案：避免单点失效；复制和故障转移；自动检测和重启
    - 系统恢复：重新回到正常运行状态并且恢复受损数据
    - 可用性计算：

      • MTBF (mean time between failures)
      • MTTR (mean time to repair)

      $$\frac{MTBF}{(MTBF + MTTR)}$$

    - outstage failure fault error
        - outstage 系统不能用(可能是意外 也可能是计划中的)
        - failure 问题展现导致系统出现问题
        - fault 导致系统故障的问题
        - error 系统中的问题，但可能被catch

    - 错误检测
        - ping/echo：来回 双方通讯连接
        - heartbeat：单向 需要频繁建立连接 双方通讯连接
        - exception：异常 在一个process中
        - 自我测试
        - 异常检测
        - 对结果合理性进行检查
        - 通过发送和接收的时间差进行检查

    - 错误恢复
        - Voting 投票机制

    - active redundancy 多台设备并行处理，占用资源
    - passive redundancy 两台设备，一台正常运行，保持数据同步，一个出现问题马上启动另一台设备
    - spare 通过重启恢复到上一个正常运行时间
    - state re-synchronisation 状态同步
    - shadow operation 影子的形式监测刚刚修正的组件
    - checkpoint rollback 检查点 回滚
    - remove from service 将某个组件移除服务
    - 错误处理
    - 软件升级
    - 重试
    - 降级
    - 重写配置
    - 将不能处理的问题抛给别人
    - 扩大重启范围

- Interoperability 互操作性
    - 正确交换和处理数据的能力
    - 发现(识别接口或者请求)+处理(发送 接收 转发)
    - 转发：安排一个中心控制单元来协调各个部分

- Modifiability 可修改性
    - 系统需要修改时需要付出的代价，需要在设计时考虑
    - tactic
        - 减少模块大小：模块划分
        - 增加内聚性：提高模块内聚性
        - 减少耦合：封装 使用中间层 限制依赖 重构 抽象服务层
        - 延迟绑定

- Performance 性能
    - 系统响应时间 满足时间上的要求 包括处理时间和阻塞时间
    - 控制资源**需求**：管理采样率(减少高频率采样) 限制时间处理 确定时间重要程度(优先级) 降低overhead(工作之外的时间和精力) 限制处理时间 提高资源使用效率(平衡资源 合理资源调度策略)
    - 管理**资源**：增加资源 并行处理 维护多个计算的copy(把计算分配到不同计算单位) 限制队列大小 资源调度

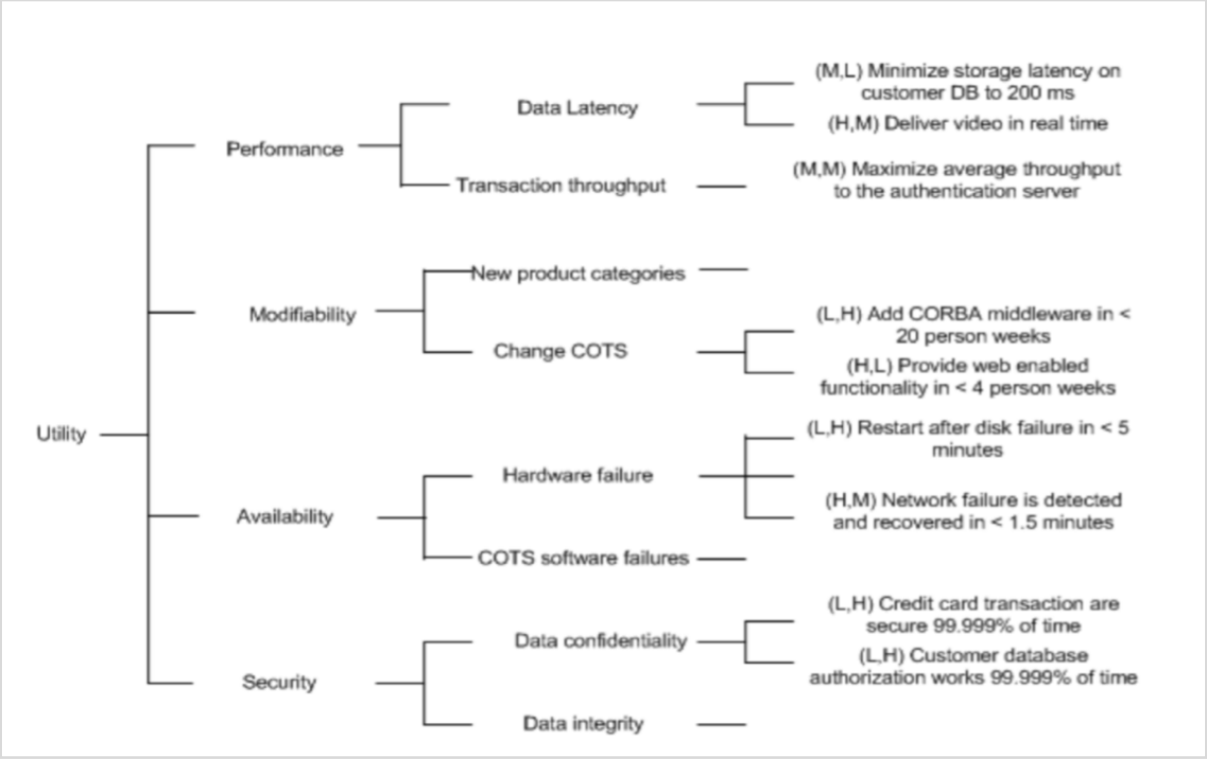
- Security 安全性
    - 指的是防止系统因为恶意破坏而产生数据安全问题，也可能是因为用户没有权限

    - 特点：保密性(保护数据不可达)、完整性(不允许未授权操作)、可用性(合理使用)
    - 识别攻击
    - 阻止攻击：识别攻击者 合理授权 数据加密 实体分离 修改默认设置
    - 对攻击的反应：阻止 对系统上锁 通知攻击者
    - 恢复系统

- Testability 可测试性
    - 系统对测试的支持程度，可以通过输入看到输出并比对结果
    - 控制和观察系统状态：控制输入 错误复现 沙盒测试
    - 限制复杂度

- Usability 可用性
    - 用户学习的难易程度，对系统功能的支持程度
    - 学习难易程序 效率 错误率 适应用户需求 提高系统满意度
    - 支持用户：各种操作 取消 暂停……
    - 支持系统：主要是系统能否满足客户需求 维护模型……

- **Architecturally Significant Requirements(ASR)** 系统中重要并且实现难度相对较大的质量需求
    - 来源：需求文档 与利益相关者的交流 了解商业目标 效用树，下图为效用树



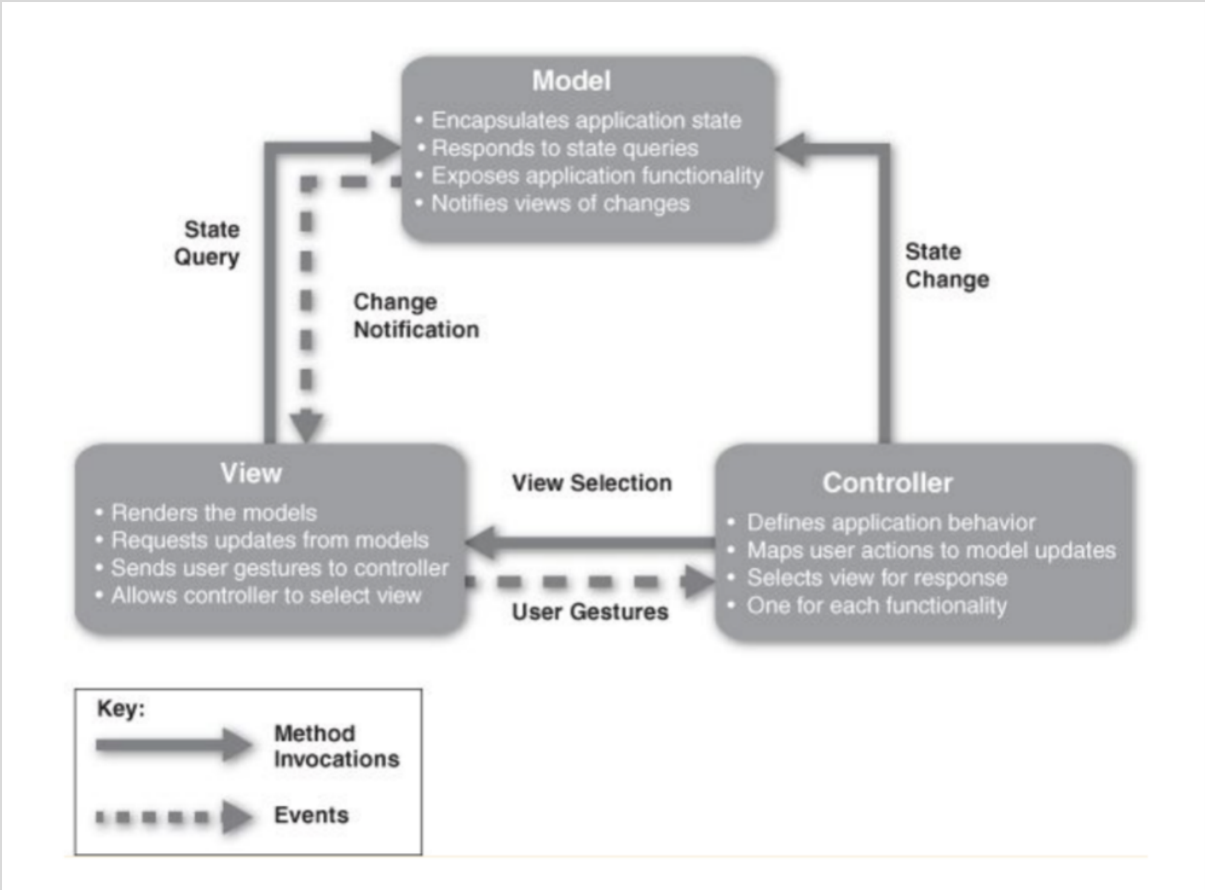
Chapter 3 architectural patterns

- 基本目的：不同系统中复用，解决一类问题
- 主要分类：module C&C(component and connector) allocation
- layered pattern 分层模型 —— module
  - 严格分层，不能有闭环，只能下层依赖上层
  - 缺点：增加系统复杂度 影响系统性能

- Broker pattern —— C&C
  - 组件
    - broker：运行中的组件，作为服务器和客户端之间的中间连接者，对请求和数据进行分发
    - server
    - client
    - server-side-proxy
    - client-side-proxy

  - 缺点：增加延迟 测试困难 broker可能变成单点失效部分 broker可能面向安全攻击从而导致系统瘫痪

- MVC pattern —— C&C
  - 缺点：系统复杂 model不能和controller进行直接通讯



- Pipe-and-Filter Parttern —— C&C
  - pipe connector 将filter处理得到的数据进行传输
  - filter component 并行进行数据处理
  - 适合用在计算密集、多阶段处理的系统中
  - 不适合放在单次计算需要消耗大量时间的系统中
  - 对交互不友好

- Client-Server Pattern —— C&C
  - 缺点：server会成为性能瓶颈、server会成为单点失效、系统耦合程度高

- Peer-to-Peer Pattern —— C&C
  - 每个组件都是网络间的一个节点：每个节点可以直接点对点发送请求、节点寻找
  - 有些特殊的组件可以提供路由功能
  - 缺点：节点之间相互知道彼此的信息、数据同步、数据安全性、数据可恢复性、影响testability 无法测试

- Service-Oriented Pattern —— C&C
  - service-provider
  - service-consumer
  - ESB 扮演中间角色

    - registry of services：服务提供者注册他的服务让消费者发现并使用
    - orchestration server：服务指挥者 基于商业流程提供规定工作流
    - connector：SOAP、REST、异步消息机制
    - 限制、缺点：消费者不一定直接和服务提供者相连，可能会有中间者；性能不能整体评估

- Publish-Subscribe Pattern —— C&C
    - 缺点：信息分发不确定、延迟

- Shared-Data pattern —— C&C
    - 缺点：单点失效、高耦合、数据存储那边成为性能瓶颈
    - scalability availability 降低

- Map-Reduce Pattern —— allocation
    - 支持大量并行计算
    - 缺点：前提是需要把大量数据集进行拆分，大量map-reduce操作难以控制复杂度

- Multi-Tier Pattern —— C&C
    - tier 物理上、架构上相关联组件的集合

- pattern VS. tactic
    - pattern：打包多个设计 解决单个问题
    - tactic：改善某个质量属性

Chapter 4 design architecture

- design strategies
    - 分解：把质量属性分解到单个组件上，满足约束要求，实现质量和商业上的需求
    - 生成测试：利用已有的系统 初步的设计设计测试用例 不断迭代生成测试
    - ADD Attribute-Driven Design(第一个迭代8个步骤，后面的迭代7个步骤 不需要第一个步骤：确定需求)
        1. 明确需求的信息足够多
            - 使用刺激-响应描述需求
            - 根据优先级进行排序

        2. 选择一个系统组件进行分解
            - 第一个迭代对整个系统进行分解
            - 后面的迭代选择一个组件即可

3. 识别选定组件的ASR
    - 使用(H,H)进行标注每个需求，第一个字母表示需求的重要程度，第二个字母表示实现的难易程度

4. 设计
    - 识别设计关注点——如何满足ASR
    - 列出可能的pattern/tactic——分析pattern/tactic的重要性和价值
    - 选择一个pattern/tactic
    - 研究pattern/tactic和ASR之间的关系
    - 创建初步架构视图
    - 评估架构设计冲突或者不一致的地方并解决

5. 实例化并将责任分配到单个组件上
6. 定组件之间交互的接口
7. 验证和确认目前实例化的组件是否满足系统需求
8. 迭代

- ADD输入：需求
- ADD输出：组件、每个组件的角色和职责、属性、组件之间的关系
- TODO chapter4 后面的ADD例子

Chapter 5 Documenting Architecture

- 文档化原因
    - 有利于交流
    - 有利于理解
    - 维持记忆
    - 训练人进行结构设计
    - 支持地理上分布较广的团队

- 文档用处
    - 架构设计分析
    - 分工
    - 维护

- 文档包含内容
    - 接口和依赖
    - 限制
    - 测试用例和场景

    - 上下文信息

- 注意事项
    - 从读者的角度撰写
    - 避免不必要的重复
    - 避免二义性
    - 使用标准格式
    - 合理记录
    - 保证文档语言的易读性(不要太落伍，也不要让别人看不懂)
    - 针对特定目的审查文档的合理性

- Style VS. Pattern
    - Style: 组件之间联系的方式
    - pattern：一个系统中组件的组织结构

- Architecture style
    - module：提供完整功能的单位
    - component and connector：组件根据运行时提供的功能划分；组件之间需要专门的连接件
    - allocation：软件单元和物理环境的映射关系，包含软件单元和物理组件
    - quality：从质量属性出发

Chapter 6 Evaluating Architecture

- ATAM
    0. partnership & preparation
        - 分析架构文档，给出评估计划
    1. evaluation 1
        - 评估组展示他们对项目的理解，以及一个预先评估的结果展示
        - 负责人展示项目的商业相关内容：功能性需求 技术 管理 财务 政策 限制 商业目标 主要利益相关者 主要质量属性
        - 架构组展示他们详细的架构设计：技术限制 系统交互 架构方法
        - 评估组识别架构方法(style pattern tactic)
        - 评估组生成效用树
        - 根据效用树分析架构方法
    2. evaluation 2
        - 重复1~6 不需要效用树了
        - 利益相关者根据自身角色进行头脑风暴，想出一些和个人相关的场景，并对场景进行优先级排序

    - 分析架构方法
    - 展示结果：架构文档 含有优先级的场景 效用树 敏感点和权衡点 风险
3. follow-up
    - 让关键利益相关者评审，产出最终评估文档

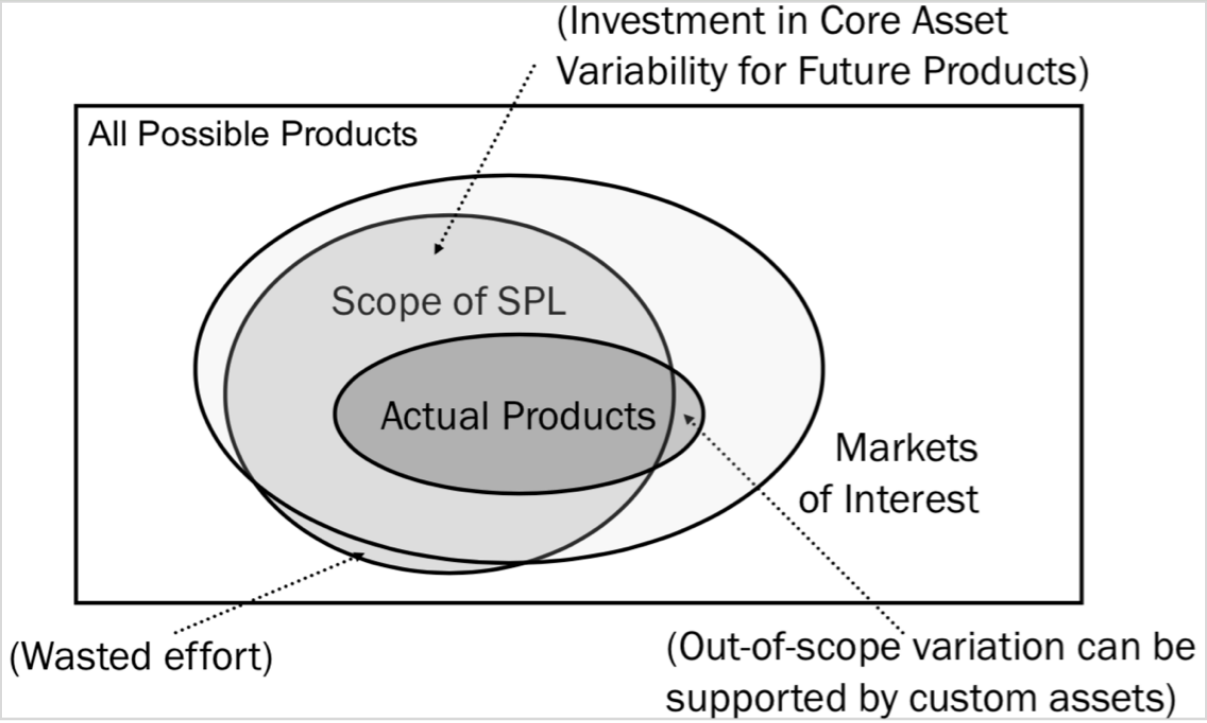
【最终产出】：架构的详细展示 关键商业目标 通过质量属性场景描述的按优先级需求列表排序的质量需求 效用树 风向 风险主题 结构和质量需求的对应关系 敏感点和权衡点 评估报告

- risk: 架构设计中可能会影响质量属性的方面
- sensitive point：架构设计中对某个质量属性比较敏感的地方
- tradeoff：架构设计中会影响多个质量属性的地方

Chapter 7 Software Product Line(SPL)

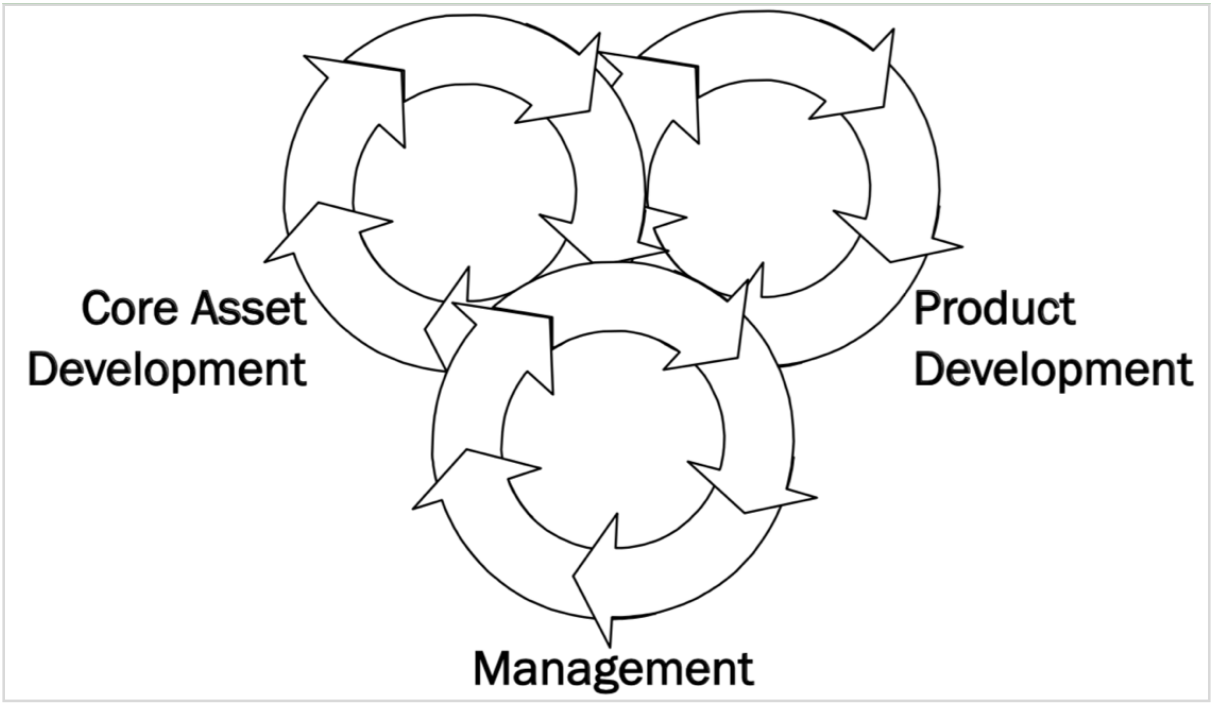
- Software Product Line 软件产品线
    - 定义：软件产品线是指为满足特定市场需求开发的一系列产品组件和系统
    - 共享core assets
    - core assets + custom assets(核心资产+自定义资产)
    - core assets: 被许多产品重用、包含领域知识和经验、为不同的特性提供变化点、在SPL生命周期中实现需求
        - custom assets：对特定产品有效、满足客户需求、在SPL生命周期中实现需求、结合core assets，将其放在产品需要的地方、实例化core assets提供的变化点
    - 目的：实现高重用性，高可修改性
    - 特点
        - 有计划 有策略的大范围重用
        - 多个同时可变点
        - 使用系统级可变点
        - 可以有一些支持重新配置的组件或者架构
        - 可以是基于组件的开发
        - 可以使用一些技术标准和平台
    - 为什么需要软件产品线开发
        - 经济效益：将核心资产重复使用，减少从头开发的代价
        - 简化：容易发现什么是最核心的东西，产品之间的差别是什么，便于升级
        - 把项目级工作变成产品级特性
        - 提高产品质量：核心资产相对bug更少，一旦改成一个bug就可以减少很多相关问题
        - 增加推广时间
    - 重用的好处
        - 代码量减少
    - 产品线架构实现
    - 重用：查找，理解，使用
    - **可变性是代码可供大范围重用的关键**

- 产品线范围



  - 关注点集中在某个领域
  - 利益最大化
  - 成本最低化
- 产品线架构 PLA（Product Line Architecture）
  - 支持软件产品线范围的可变性特点
  - 基于架构层面的重用
  - 产品线架构可以看成SPL的core assets
- 变化：不同的变化形式（Include or omit elements、Variable number of each element、Interface Satisfaction、Parameterisation、“Hook” interfaces、Reflection）*变化发生的位置（架构层次 设计层次 文件层级） *变化发生的时间、绑定时间（code阶段 编译阶段 链接阶段 初始化阶段 运行阶段）=90
- 变化点 variation points

- SPL Process & Practice



Chapter 8 Micro-service Architecture 微服务架构

- 分层架构模式特点
  - 结构简单
  - 易于组织开发
  - 便于独立测试、维护
  - 不易实现持续发布、部署
  - 性能代价
  - 可扩展性差
- SOA 面向服务架构

  特点
  - 服务自身高内聚、服务间松耦合，最小化开发维护中的相互影响
  - 良好的互操作性，符合开放标准
  - 模组化，高重用性
  - 服务动态识别、注册、调用
  - 系统复杂度性提高
  - 难以测试验证
  - 各独立服务的演化不可控
  - 中间件易成为性能瓶颈

  实现原则
  - 服务解耦：服务之间的关系最小化，只是互相知道接口
  - 服务契约：服务按照描述文档所定义的服务契约行事
  - 服务封装：除了服务契约所描述内容，服务将对外隐藏实现逻辑
  - 服务重用：将逻辑分布在不同的服务中，以提高服务的重用性

    - 服务组合：一组服务可以协调工作，组合起来形成定制组合业务需求
    - 服务自治：服务对所封装的逻辑具有控制权
    - 服务无状态：服务将一个活动所需保存的资讯最小化
- ESB 企业服务总线
- Orchestration Engine 服务编排引擎
- 构建云原生应用程序SaaS的12个要素
    - 基准代码：一份基准代码，多份部署，且保持同步
    - 依赖：显式声明依赖关系
    - 配置：在环境中存储配置
    - 后端服务：把后端服务当作附加资源
    - 构建、发布、运行：严格区分
    - 进程：以一个或多个无状态进程运行应用
    - 端口绑定：通过绑定端口提供服务
    - 并发：通过进程模型进行扩展
    - 易处理：快速启动和优雅终止
    - 开发环境与线上环境等价
    - 日志：把日志当成事件流
    - 管理进程：后台管理任务当作一次性进程运行
- 微服务架构

  特点 （服务颗粒化 责任单一化 运行隔离化 管理自动化）
    - 通过服务组件化
        - 组件是一个可独立替换或升级的软件单元，微服务架构实现组件话的方式是分解成服务
        - 服务是一种进程外的组件，它通过web服务请求或远程过程调用（RPC）机制通信
        - 使用服务作为组件的主要原因是服务是可独立部署的
        - 使用服务作为组件产生更加明确的组件发布接口
    - 内聚和耦合
        - 提高内聚性，尽可能解耦，使得每个部分拥有各自的领域逻辑
    - 去中心化
        - 去中心化治理，构建可以有自己的技术栈
        - 去中心化数据存储，每个服务管理自己的数据库
        - 去中心化数据管理，强调服务间的无事务协作，需要权衡更大一致性的业务损失与修复错误的代价
    - 基础设施自动化，降低构建、部署、运维微服务的操作复杂度
    - 服务设计
        - 高可用性
        - 变更与演化

  挑战
    - 运维要求高：微服务数量多，部署与监控要求高

- 发布复杂度：部署环境多样化，网络性能、系统容错、分布式事务等挑战
- 部署依赖强：服务间相互调用关系复杂，存在部署顺序依赖
- 通信成本高：跨进程调用比进程内调用消耗更多资源

技术选择：

- 开发服务：Spring boot
- 封装服务：Docker
- 部署服务：Jenkins
- 注册服务：ZooKeeper
- 调用服务：Node.js

核心模式：

- 服务注册与发现
  - 微服务启动时将自己的地址等信息注册到服务发现组件
  - 服务消费者可以从服务发展组件查询服务提供者的网络地址和调用接口
  - 各个微服务与服务发现组件使用一定机制通信，如果无法通信即立即注销该实例
  - 微服务数量、地址、接口等发生变更时，会重新注册到服务发现组件，无需人工修改
- API网关
- 熔断器

#学习/2018-2019第2学期