软件体系结构笔记

笔记本： 科软
创建时间： 2020/12/5 14:00 **更新时间：** 2021/5/8 14:00
作者： 栗子学姐

- 教师：丁箐
- 主页：http://staff.ustc.edu.cn/~dingqing/

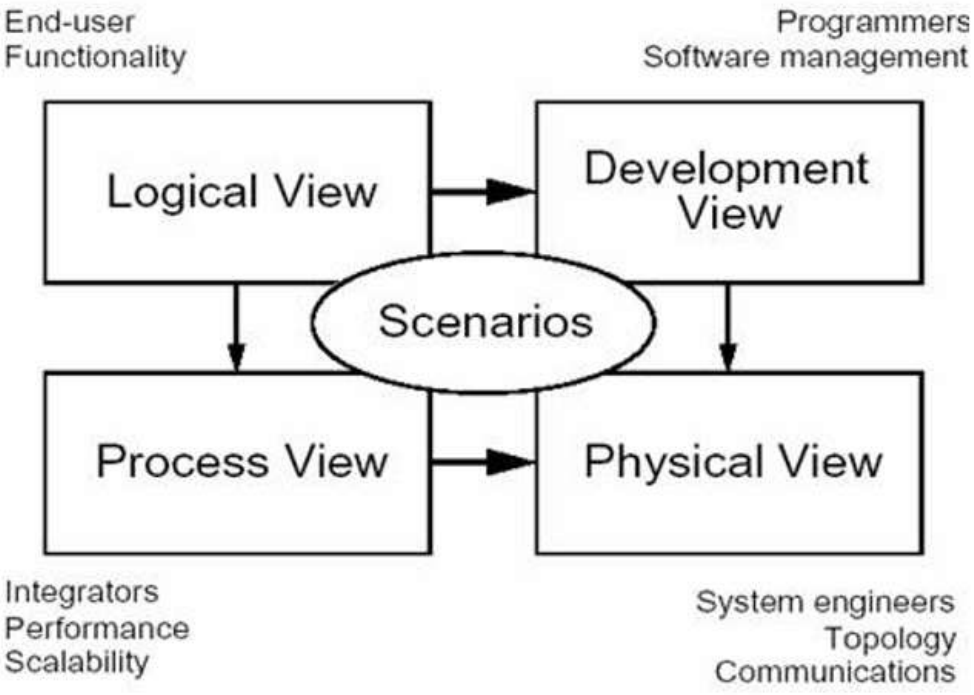
实验报告

1. 实验介绍（背景）
    - 目的
    - 简介
2. 系统分析（实验内容，4+1视图）
    - 实验内容（技术框架，接口）
    - 用户需求（功能性，非功能性）
    - 4+1视图
    - 质量属性
    - 设计模式（可选）
    - 系统架构（可选）
3. 实验结果（性能测试）
    - 运行结果（截图）
    - 性能展示（截图，图表）
4. 实验评价（优缺点）
    - 优点（非功能性需求，质量属性，等角度）
    - 缺点，可改进方面

考试重点

4+1视图

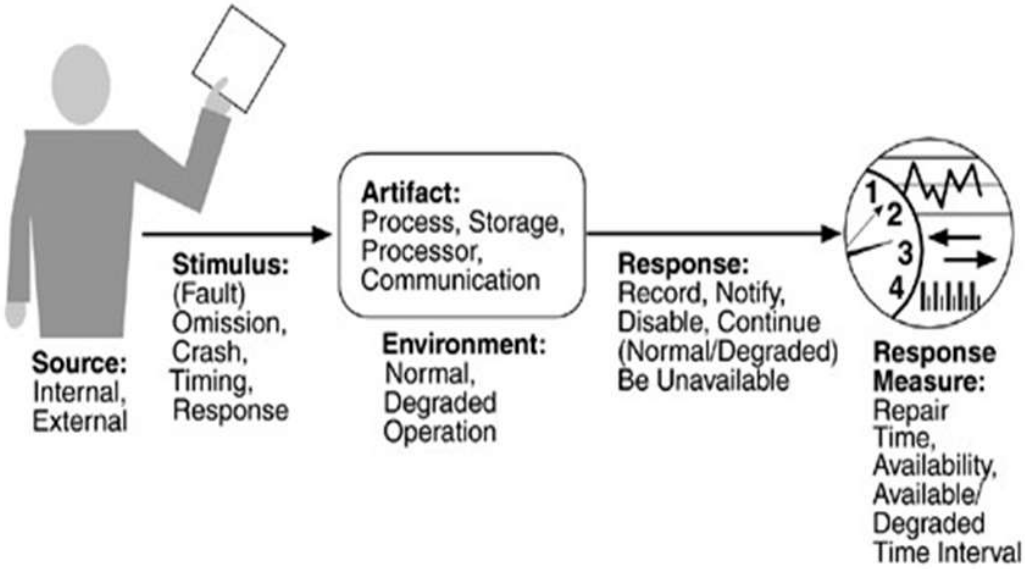
- logical view（唯一，关注功能性需求）
- development view（代码相关）
- process view（性能和扩展性需求）
- physical view（硬件拓扑逻辑）
- use case / scenarios （用例/场景）



	logic	process	development	physical	use case
视图	逻辑视图	过程视图	开发视图	物理视图	场景
组件	类	任务	模块、子系统	节点	步骤、脚本
连接工具	关联、继承、约束	会面、消息、广播、RPC 等	编译依赖性、“with”语句、“include”	通信媒体、LAN、WAN、总线等	
容器	类的种类	过程	子系统（库）	物理子系统	Web
涉众	最终用户	系统设计人员、集成人员	开发人员、经理	系统设计人员	最终用户、开发人员
关注点	功能	性能、可用性、S/W 容错、整体性	组织、可重用性、可移植性、产品线	可伸缩性、性能、可用性	可理解性

质量属性

- 性能performance
  - 响应时间、吞吐量、准确性、有效性、资源利用率等与系统完成任务效率相关的指标。
  - 架构设计策略：
    - 增加计算资源
    - 改善资源需求（减少计算复杂度）
    - 资源管理（并发，数据复制）
    - 资源调度（先进先出队列，优先级队列）
- 可用性availability
  - 可用性是指系统正常工作的时间所占的比例。可用性会遇到系统错误，恶意攻击等问题的影响，也就是在有情况发生下，能否稳定工作或尽快恢复
  - 常见情况
    - 日常维护
    - 物理层失效
    - 恶意攻击
  - 架构设计策略
    - Ping/Echo
    - 心跳，异常
    - 主动冗余，采用主从备份



- 安全性
  - 系统向合法用户提供服务并阻止非授权用户使用服务方面的系统需求。
  - 架构设计策略
    - 追踪审计
    - 抵御攻击（授权、认证、限制访问）
    - 攻击检测（入侵检测）
    - 从攻击中恢复（部分可用性策略）
- 可修改性
  - 架构设计策略
    - 信息隐藏
    - 接口-实现分离
    - 软件模块泛化
    - 限制模块之间通信
    - 使用中介和延迟绑定

软件架构

- MVC
  - Model —— 应用程序主体部分
    - JavaBean
    - 维护并保存持久性业务数据
    - 实现业务处理功能
    - 将业务数据的变化及时通知视图
  - View —— 用户看到并与之交互的界面
    - JSP
    - 呈现模型中包含的业务数据
    - 响应模型变化，更新呈现形式
    - 向控制器传递用户的界面动作
  - Controller —— 接受用户输入，完成用户需求
    - Servlet
    - 将用户的界面动作映射为模型的业务处理功能
    - 根据模型返回的业务处理结果，选择新的视图
- 软件架构风格
  - 定义
    - 组织方式——组成构件的组织方式
    - 惯用模式——共有的结构和语义
  - 数据流风格
    - 批处理
    - 管道-过滤器
      - 每个构件有一组输入和输出
      - 构件——过滤器，连接件——数据流传输的管道
  - 调用/返回风格
    - 主程序-子程序

            - 共享存储器交换数据
            - 主程序顺序调用子程序
            - 所有计算构件作为子程序协作工作
        - 层次架构
        - 面向对象架构
    - 虚拟机
        - 解释器架构
            - 描述语法和语义
            - 对语言进行解释
            - 生成对于的脚步语言程序
        - 规则系统
    - 以数据存储为中心架构
        - 特点
            - 支持交互式数据处理
            - 解耦功能间的依赖关系，灵活定义功能的逻辑顺序
            - 表示多种数据格式，支持格式间转换
        - 黑板架构
            - 解空间很大，求解过程不确定
        - 仓库
    - 隐式调用架构（过程控制）
        - 定义xx事件
        - 维护事件注册表，事件关联函数
        - 监听事件，当xx事件发生，查找注册表，并调用函数，实现自动定位
    - 独立构件
        - 通信架构
        - 事件驱动

架构风格-应用

- 管道-过滤器
    - 数据驱动
- 黑板——求解过程复杂，语音识别
- 仓库
    - 文件或模型驱动
- 隐式调用——事件驱动系统，做一件事触发一系列行为
- 规则系统——虚拟机风格，自定义交互
- 过程控制——不断采集当前状态
- C2——并行构件网络

设计模式

OO设计原则

- 单一责任原则 Single Responsibility Principle
- 开闭原则 Open for extension, closed for modification principle
- 里氏代换原则 Liskov
    - 子类可替换父类
- 依赖倒转原则 Dependence Inversion Principle
    - 高层模块，低层模块都应依赖抽象
    - 细节应依赖抽象，抽象不依赖细节
- 迪米特法则 Demeter
    - 最少知识原则
    - 若两个类不必互相通信，则不应直接相互作用，可通过第三者转达

创建型模式 creational patterns

用于创建对象，为设计类实例化新对象提供指南。

- 工厂模式 factory

    - 简单工厂：定义一个类，由该类封装实例化对象
    - 缺点：类的创建依赖工厂类；违背开闭原则：对扩展和修改开放
    - 工厂方法：定义一个抽象类（接口），**由子类决定要实例化的类**
    - 优点
        - 把实例化对象的代码从工厂抽象出来，形成一个个类
    - 缺点
        - 实例化对象，必须找不同需求的工厂子类
    - **应用场景**
        - 无法预知对象确切类别及其依赖关系
        - 用户能扩展你软件库或框架的内部组件
        - 复用现有对象来节省系统资源
    - 实现方法
        -
- 抽象工厂模式 abstract factory
    - 抽象工厂接口：可实例化多个产品
    - 提供一个创建一系列相关或互相依赖对象的接口，而无需指定具体的类
- 单例模式 Singleton
    - 窗口最多打开一个
    - 确保一个类最多只有一个实例，并提供一个全局访问点
        - 构造方法private
        - 实例方法public，调用private构造方法
    - 加锁，保证只实例化一个对象（判null）
    - 预加载：会造成内存浪费
    - 保证懒加载的线程安全
        - 把sychronized加在if(instance==null)判断语句里面，保证instance未实例化的时候才加锁
        - volatile保证对象实例化过程的顺序性。
    - 缺点
        - 线程安全问题，初始化对象和返回内存指针可调换顺序
- 建造者模式 Builder
    - 厨师炒菜
    - 将一个复杂对象的构建与它的表示分开
    - Director 创建builder：隔离用户与建造过程，控制建造过程
    - 对象建造顺序稳定，对象内部构建变化
- 原型模式 Prototype
    - 复制简历
    - 通过拷贝clone()原型创建新的对象

结构型模式 structural patterns

用于处理类或对象的组合，对类如何设计以形成更大的结构提供指南

- 适配器模式 Adapter
    - NBA翻译
    - 使由于接口不兼容的类可以一起工作
    - 通常在软件开发后期或维护期使用，**使用第三方组件**
- 装饰器模式 Decorator
    - 穿衣搭配
    - 动态地给一个对象添加一些额外地职责，比子类更灵活
    - 把类地核心职责和装饰功能区分开，去除重复地装饰逻辑
- 代理模式 Proxy
    - 为其他对象提供一种代理，以控制对这个对象地访问
    - 应用
        - 远程代理：WebService在.Net中的应用
        - 虚拟代理：需要创建开销很大的对象
        - 安全代理：对象有不同的访问权限
        - 智能指引：调用真实对象时，代理处理另一些事
- 外观模式 Facade
    - 基金——股票
    - 为子系统中的一组接口提供一个一致的界面——高层接口，使得子系统更容易使用

    - 应用
        - 设计初期，有意识**分层**
        - 开发阶段，增加Facade可减少众多子类之间的依赖
        - 维护遗留的大型系统，让新系统与Facade交互
- 桥接模式 Bridge
    - 手机软件何时统一
    - **抽象部分与实现部分分离**，使它们可以独立地变化
    - 实现系统可能有多角度分类，每种分类都有变化，把多角度分离出来
- 组合模式 Composite
    - 分公司
    - 用户对单个对象和组合对象的使用具有一致性
- 享元模式 Flyweight
    - 用户共享网址
    - 运用共享技术，有效支持大量细粒度的对象
    - UnsharedConcreteFlyweight 可以解决不需要共享对象的问题
    - 应用
        - 一个应用程序使用大量的对象
        - 对象有少数内部状态和大量外部状态（跳棋的颜色和位置）
        - String对象的引用

行为型模式 behavioral patterns

用于描述类或对象的交互以及职责的分配，提供指南

- 策略模式 Strategy
    - 商城打折
    - 定义算法家族，分别封装起来
- 模板方法模式 Template Method
    - 抄错题
    - 细节层次一致，个别步骤实现不同
- 观察者模式 Observer
    - 老板回来了
    - 抽象通知者——抽象观察者
        - 当一个对象的改变需要同时改变其他对象
        - 两个对象，一方依赖另一方，解除耦合
    - 事件委托
        - 可搭载多个方法，方法不需要属于同一个类
- 迭代器模式 Iterator
    - 有多种方式遍历时
- 责任链模式 Chain of Responsibility
    - 批请假条
- 命令模式 Command
    - 烤羊肉串
    - 将请求封装为一个对象
    - 优点
        - 容易设计一个命令队列
        - 容易将命令记入日志
        - 允许接收请求地一方决定是否要否决请求
        - 容易实现撤销和重做
        - 容易增加新的具体命令类
- 备忘录模式 Memento
    - 如果回到从前
    - 不破坏封装的前提下，捕获一个对象的内部状态，并在对象之外保存这个状态
    - Originator：获取Memento参数
    - Caretaker：保存Memento参数
    - **命令模式联动**
- 状态模式 State
    - 工作状态——加班
    - 一个对象的内在状态改变时，只改变其类
    - 消除分支结构
- 访问者模式 Visitor

    - 男人和女人
    - 适用于数据结构相对稳定的系统，把数据结构和作用于结构上的操作之间的耦合解脱开
- 中介者模式 Mediator
    - 联合国
    - 窗体Form，Web页眉aspx
- X 解释器模式
    - 正则表达式

课堂笔记

Some tricks

- 流程图不能显示并发程序，只有活动图能，两者区别 https://blog.csdn.net/li295214001/article/details/56667403
- 实验课：Software engineers collect requirements, code, test, integrate, configure，UML图
- 软件模式设计：找pattern匹配 -> 搭骨架 -> 修饰细节

导论

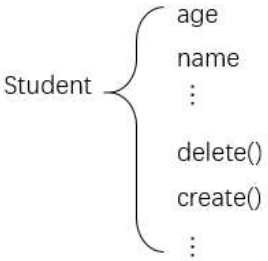
- why 软件体系结构/设计模式
    - 研究软件产品落地
    - 程序员 ——> 资深程序员（设计模式） ——> 架构师（架构模式）
- 课程内容
    - Overview of architecture and architect 结构和架构总览
    - Software Architecture Analysis and Design 软件架构分析与设计
    - Architectural views and styles 架构视图和模式
    - Service Architectures 服务架构
    - Microservices architecture 微服务架构
    - Serverless architecture 无服务架构
    - Quality Attributes of architecture 架构的质量属性
    - Design pattern 设计模式
- 工具
    - IDE: Eclipse/MyEclipse or NetBean
    - Web Server: Tomcat/GlassFish
    - EJB server: JBOSS/GlassFish
    - Database server: MySQL
- 推荐书目
    - dive into design patterns （老师认为讲设计模式最好的书）
- 软件工程需求
    - 功能性需求
    - 非功能性需求
    - 一旦架构系统确定，很少变动

架构模式类型

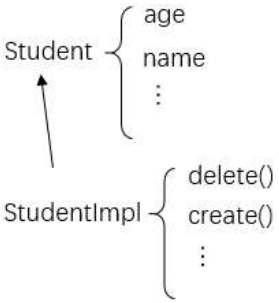
- C/S 模式
    - 架构放在client，数据库放server
    - 实现代码和数据分离
    - 缺点：C/S架构多为局域网架构（数据库安全），现在数据多用到公网访问
- B/S 模式（monopoly）

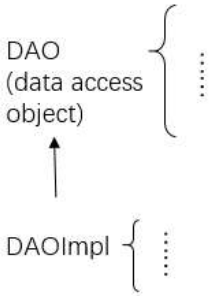
    - 数据库和应用都在server
    - 优点：数据库和代码联系一起，容易实现代码联合修改
    - 缺点：数据和代码捆绑一起，没有通用性
- P2P（peer to peer）
    - client和server在一起（每个结点同时都是client和server）
    - e.g. 区块链，自组织网络（ad hoc network)
    - 缺点：不方便管理数据，没有统一的server
- RoA(resource-oriented architecture) 面向资源架构
    - 轻量级web服务
- SoA(service-oriented architecture) 面向服务架构
    - 为了标准化和通用，主要用于B2B
    - remote process core 家族：用于corba架构，RMI远程方法调用，PCom
- Micro service，serveless 微服务
    - 用于云计算，降低成本
    - 细颗粒度应用，components之间交互多，可重用性好，启动和响应速度快
- 大对象/小对象

大对象



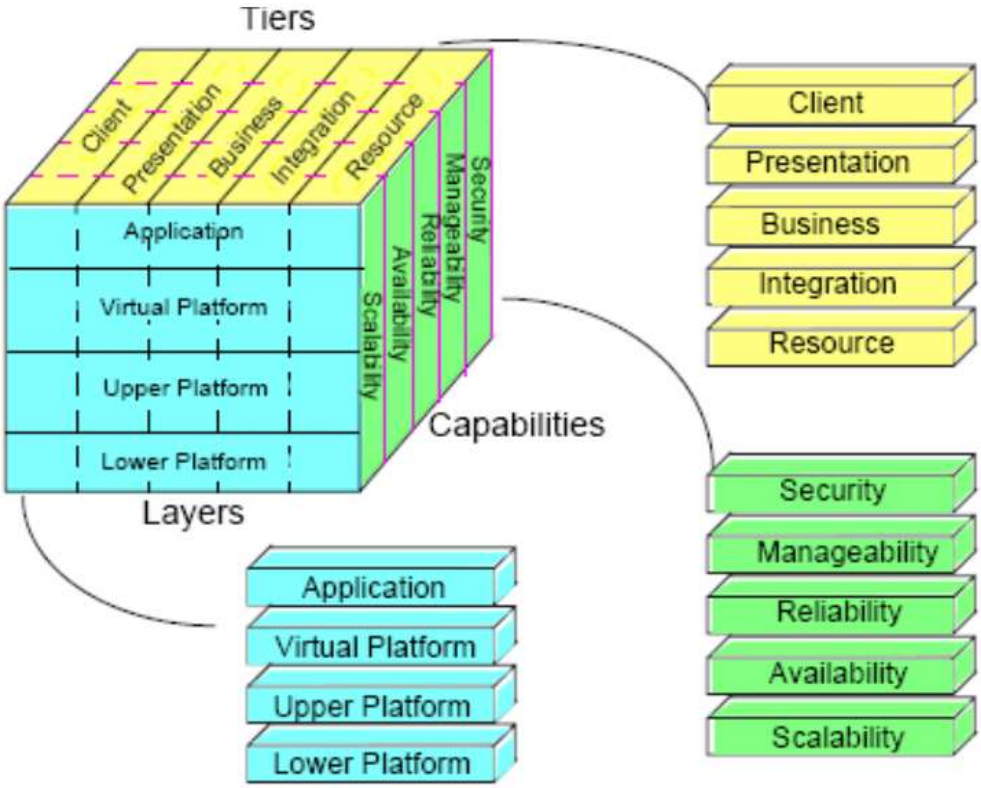
小对象





Architecture cube

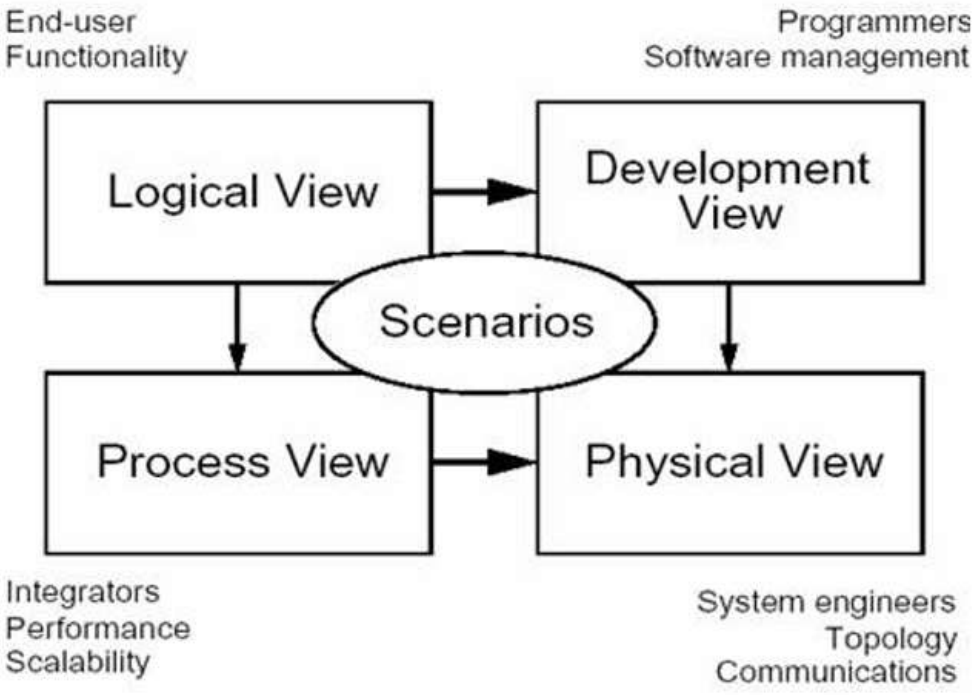
- Tiers
    - Tier通常指的是物理上的分层，由执行相同功能的一台（或者一组）server定义
- Layers
    - Layer通常指的是逻辑上的分层，对于代码的组织，比如：我们通常提到的"业务逻辑层。表现层，数据访问层"等等
- 层体系架构模式http://www.uml.org.cn/j2ee/j2ee031212001.htm



- System Quality
  - availability
    - 可以使用的（时间层面）
  - reliability 可靠性
    - 应用的稳定性（出错频率少）
  - Manageability
  - Flexibility：灵活性↑，管理性↓
  - modifiability 可修改性
  - performance 性能
    - 个人：end - end 性能
    - 系统：吞吐量
    - 大部分的优化，不是偏向performance
    - 分布式应用的花费瓶颈在通讯
  - capacity（并发性）：少用到
  - scalability（可伸缩性，人数，地域）
    - 指同一产品的复用能力，能服务更多的用户。
    - 垂直可扩成本低，天花板低
  - extensibility 可扩展性
    - 增加组件，扩充功能，减少处理耦合关系
  - security 安全性
  - testability 可测性

4+1视图 An architectural mode（4视图+1场景，重点）

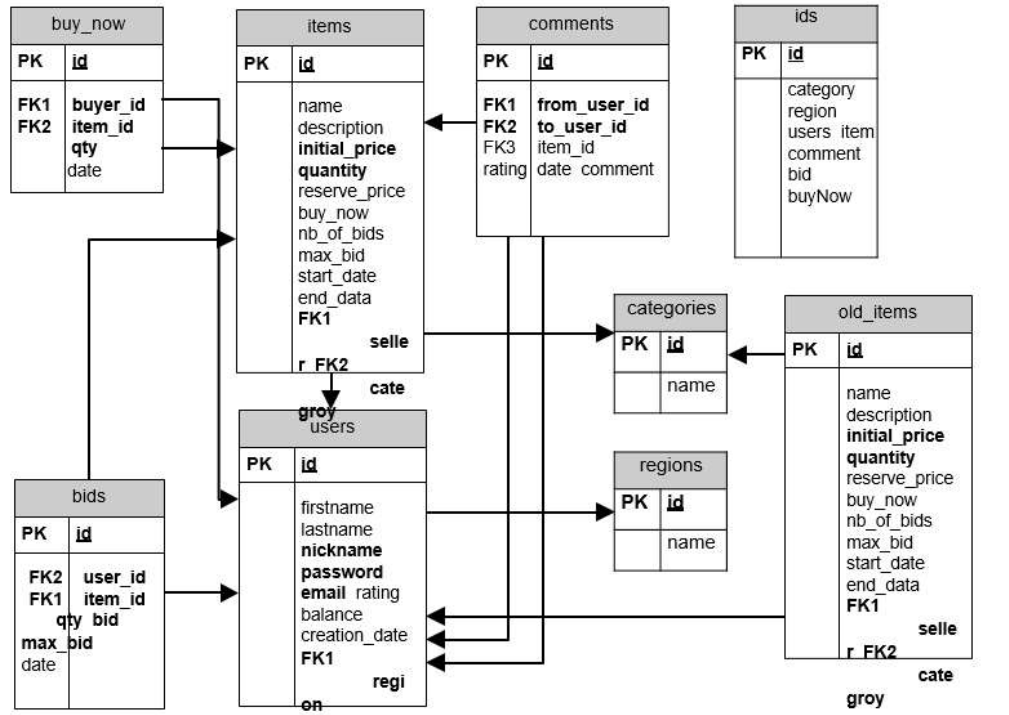
- logical view（唯一，关注功能性需求）
- process view（性能和扩展性需求）
- physical view（硬件拓扑逻辑）
- development view（代码相关）
- use case / scenarios （用例/场景）
- 功能理论：https://blog.csdn.net/degwei/article/details/51489444
- 实例说明：http://www.uml.org.cn/zjjs/201412262.asp

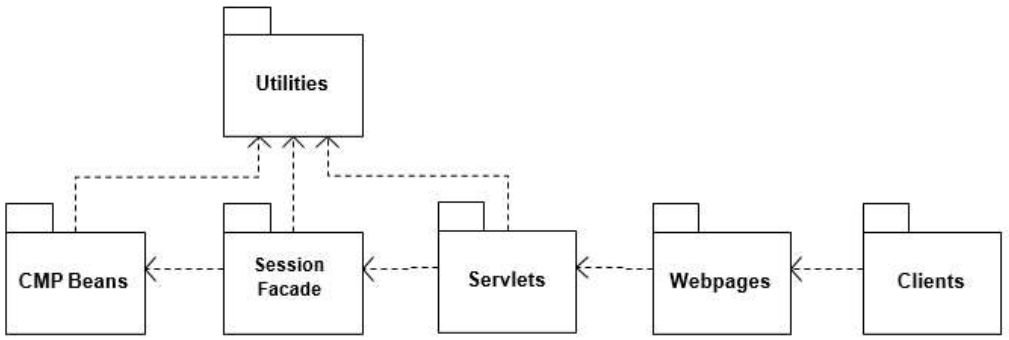


- 软件用例常见错误
  - 功能方面：从用户角色出发✓ 需求和功能一一对应×
  - 动词使用：写成名词× e.g. 学生管理系统是对学生信息的增删改查（动作），不是单纯信息
- Categorisation (将对象分成以下几类)
  - data
  - function
  - stackhoulder
  - system
  - abs.concept
- UML设计
  - 概要设计
    - 静态架构：功能模块，逻辑结构，布署——网络拓扑
    - 动态架构：活动图
  - 详细设计
    - 静态：类图
    - 动态：时序图
- components目标特性
  - 高内聚，低耦合

逻辑视图Logical view

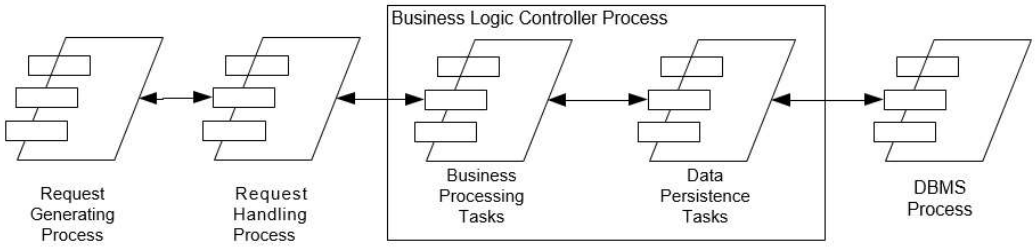
- 功能性需求
- 系统应向用户提供什么服务，关注layer划分，接口协作
- 通常用UML中类图和类模板来展示逻辑架构





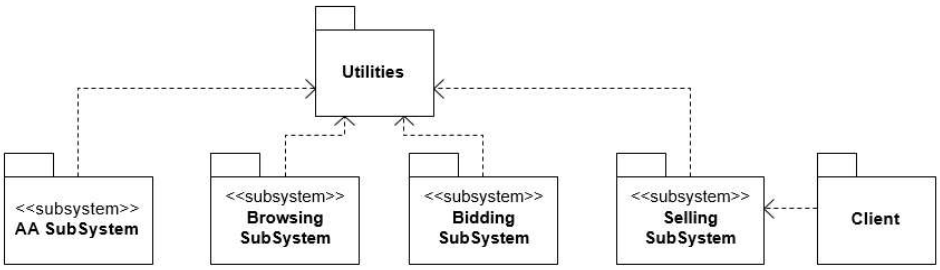
过程视图 Process view

- 非功能性需求
- 着重考虑运行期的属性（性能，可伸缩性，安全性），关注系统的并发与同步问题，考虑进程，线程，对象等运行时的概念
- 把软件分为一系列独立的任务，分为主要任务和次要任务，方框中为major tasks
- 对分布式系统，大部分时间花在通讯60%以上



开发视图 Development view

- 着重考虑开发期质量属性（可扩展性，可重用性，可移植性，易测试性）
- 关注软件模块实际组织方式（源文件，配置文件，程序包，类库，第三方库）
- 偏向于non-runtime的属性，如系统升级，系统管理
- 系统架构分为大对象和小对象



物理视图 Physical view

- 关注软件如何部署到物理机器
- 在UML中被称为Depolyment view 部署视图

经典架构风格

数据流模式 data-flow architecture

- 关注点：数据流，每个component任务不同（I/O—编解码—展示presentation）
- 数据和任务并行，数据流（时间有关，有ddl）区别于数据
- pipe and filters
    - 早期编译器采用该架构，要一步一步处理的，均可考虑采用此架构风格
    - AOP（aspect oriented programming): 为了代码纯洁性，加入cross-cutting，拦截代码（模块）执行，在调用方法前后，用pipe and filters管道过滤器
- batch sequential 类似批处理器，只能串行

数据中心架构 data-centered architecture

- 关注点：数据安全和wan'z
- repository仓库
    - 被动响应请求（由输入事务选择进行何种处理）
    - 短连接，通讯完断开
- blackboard
    - 应用在解决问题没有确定性算法的软件中（信号处理，问题规划，编译器优化）
    - 主动响应请求（由中央数据结构的当前状态决定何种处理）
    - 长连接：一直保持连接，成本高，可扩展性不好，服务器不友好
- 网络架构
    - flash：定时刷新，大多可能是无用功
    - 解决web不主动响应问题，可用web socket主动通知客户端
    - 微软soler 云数据库，广域网可访问（一般数据库只能局域网）

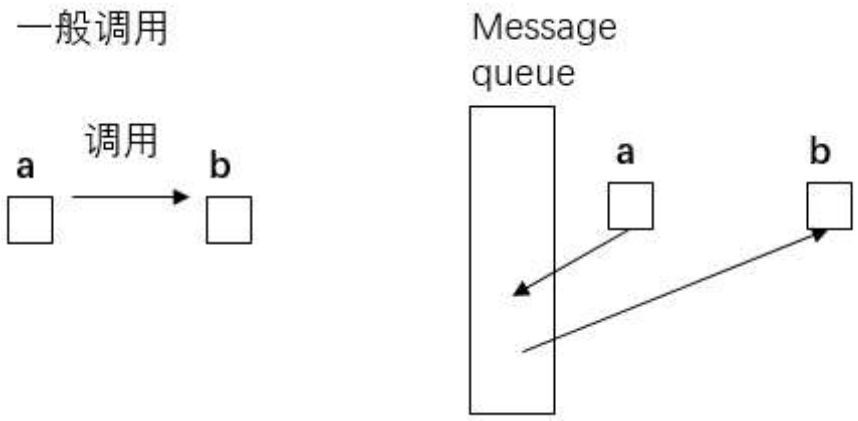
分层架构 layer architectures

- 越靠近底层，抽象级别越高
- 每一次最多只影响两层，一般为上层
- 分层：层与层之间有联系，系统关系更简单，通讯只发生在层之间not always but normally(跨层调用：无线网，I/O调用，上层通知下层结束)
- 逻辑架构一般分层，一般来说，presentation，application，data storage三层分开

通讯架构（严格来说是设计） notification architectures

- 异步通讯，低耦合，用于大规模系统，分布式节点多

- message queue消息队列
  - 异步，不存在阻塞
  - 便于管理，把多—多关系 ——> 多— 1 —多（云计算，用户多，多用异步调用）
  - 异步会导致可测试性下降，不知道事件什么时候来
  - service bus总线结构



Network-Centred Style

- Client-Server Style
- P2P Style

Remote Invocation Architectures 远程调用

- 优点：通过分布式计算提高性能
- 缺点：更加耦合，可寻址性的管理（收集对象的唯一标识）增加了通信开销

GUI architectures

- MVC模式：Javabean(M) JSP(V) servlet(C)
- 微内核（可靠性） 宏内核（易用性）

Adaptive Style 自适应风格

- Java的反射机制
  - Student s = new Master/PhD/Student();
  - Student s = Class.forName("Student");
  - 例如，IOC控制反转(inversion of control)，只用读配置文件，可完成依赖注入
- 优点：容易加入新元素和属性，支持各种修改
- 缺点：性能低下，增加了类的复杂性

Heterogeneous Architectures 异构架构

- 把几个架构组合起来

MVC、MVP和MVVC区别

http://www.ruanyifeng.com/blog/2015/02/mvcmvp_mvvm.html

基于组件的软件开发

- Application-specific components
  - Cargo, warehouse, vehicles

- Limited reuse components
  - Web server, clocks, connections
- Reusable components
  - GUI components, class and math libraries

Web Service Architectures

Competing Architectures

Resource-Oriented Architectures

- Addressability 可寻址性：作用域信息保留在URL中
- Uniform interface 统一接口：方法信息保存在HTTP方法中
- Statelessness 无状态：每个HTTP请求都与其他请求隔离
- Connectedness 连通性：将资源链接到资源网

RPC-Style Architectures

- 问题：RPC暗示一个API，这引入了处理开销

Service-Oriented Architecture 特点

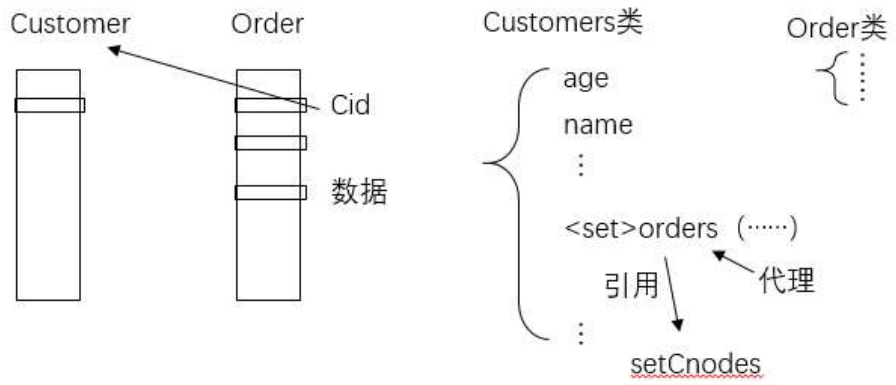
- 使用共享服务-无需“重新发明轮子”
- 松耦合-可以更新应用程序，而对调用它们的服务的影响最小
- 对上层隐藏-可以重新托管应用程序，而对调用它们的服务的影响最小
- 基于开放标准-减少了对供应商特定解决方案的依赖
- 不同组织间交互性的增强

- soa（略）
  - 每个service都运行在一个web容器里面
- 微服务
  - 分层也是小型化的过程
  - 要求部署时，连服务和代码一起部署
    - 带来问题：DB如何保证对应service访问权限，会产生小碎片
  - 相对独立的应用适合微服务
  - 细颗粒SOA，功能分解（既独立，又可管理）
  - 扁平化管理
- 无服务
  - 云计算，云厂商先提出
  - 适合短时间，无状态的事件驱动

质量属性

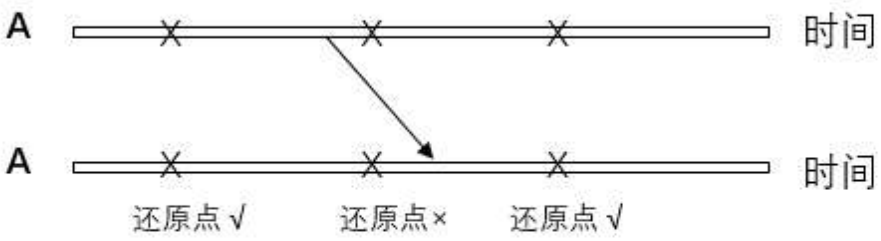
- runtime QA
- 数据库优化
  - 懒加载

懒加载



- usability易用性
  - 图形界面
  - 其他指标都会影响易用性
  - 比如安全性和易用性之间舍取（支付和密码验证）
  - 提高用户的可操作性，增加提示
- reliability可靠性
  - 高容错：时间冗余，计算冗余，数据冗余
  - 主动冗余（成本高，需解决一致性问题）：关键性领域用（供电，银行）
  - 时间冗余：回滚
- security安全性
  - 数据加密，加密通讯，限制权限，修改默认设置
- non-runtime QA
- maintainability可维护性
  - 难以测量
  - 代码注释
  - 面向对象的原则
  - 代码风格
  - 减小组件大小（现在编程越来越碎片化）
- testability可测试性
  - 写日志，检测系统状态
- Configurability可配置性
  - 系统越来越复杂，要求很多配置的东西
- scalability重点
  - 用户规模扩大，低成本的应用扩展
- Avalability
  - 区分：fault是隐性的，可观测叫failure
  - 周期性的fault较好解决，偶发性错误难测试，往往用冗余做运行，减轻（拜占庭？摆展型）/分布式一致性错误
  - 错误检测：
    - ping/echo, 用于优化
    - heartbeat, 检测+携带数据，用于性能优化，比如负载均衡
    - exceptions. 系统自检
  - 错误恢复
    - 所有数据备份
    - 主动冗余：多服务器，同时运算对外服务，互不影响。容易导致数据不一致
    - 被动冗余：只提供一个服务运算，容易实现数据一致性。服务器有主辅之分，数据在主辅服务器间复制备份，若主服务器坏了，无法做到热启动备份服务器，必须保证数据状态最新
    - check point还原点：分布式系统，打还原点很有难度

如何打check point



    - 错误预防
        - 移除服务（重启）
        - 连续步骤捆绑
        - 过程
- performance 性能
    - BT
    - tactics
        - **资源要求**
        - 减少处理事件流所需资源，提高计算效率，减少计算开销
        - 减少已处理时间的数量，管理事件发生率，控制采样频率
        - 控制资源使用，约束执行时间，约束队列大小
        - **资源管理**
        - 即使对资源的需求可能无法控制，但这些资源的管理仍会影响响应时间。一些资源管理策略是：引入并发。维护数据或计算的多个副本。增加可用资源。
        - **资源仲裁**
        - 常见的调度策略是：先进先出。
- modifiability
    - 带来时间/金钱成本
    - 大多设计模式，为了提高可维护性
    - 目标
        - 减少修改模块的数量
        - 限制对本地化模块的修改
        - 控制部署时间和成本
    - tactics
        - **本地化修改**
        - 保持语义一致
        - 低耦合，最小化更改
        - 更通用的模块
        - 限制可能的选择
        - **防止涟漪效应**
        - 双向关系，交给单向维护
        - 隐藏信息
        - 维护现有接口
        - **延迟绑定时间**
        - 多态允许方法调用的后期绑定
- security
    - 攻击/被攻击者 stimulus/artifact
    - 攻击次数，恢复困难程度
    - 检测攻击
        - 防护手段：加密（对称3DES，大信息量加密快/非对称RSA，加密对称密钥 | 通过数字证书和CA密钥公钥，hash码一样——检查数据是否破坏，公钥/密钥匹配——不可抵赖发过数据），完整性校验（checksums，hash results）
- testability
    - 可测试的覆盖率
    - 容易忽略对异常的测试
- usability

    - 用户
    - 系统预测下一步操作：预读

OO设计原则

- overview
    - 架构 自顶向下
    - 可重用 自底向上
    - SOLID原则
- 1 开闭原则 Open for extension, closed for modification principle
    - 有了需求变化，应扩展新的子类（继承+多态），不修改原来的类
    - 关键步骤：抽象化
    - 违背开闭原则
        - instanceof，有判断
- 2 里氏代换原则（Liskov Substitution Principle）
    - 对“开-闭”原则的补充，任何父类可能出现的地方，子类一定可以出现。
    - 在容器中表现为配置文件
    - 子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下4层含义：
        - 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
        - 子类中可以增加自己特有的方法。
        - 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。
        - 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。
- 3 单一责任原则 Single Responsibility Principle
    - 减少类的方法
- 4 迪米特法则（最少知识原则）（Law of Demeter）
    - 能调用自己类的成员和方法，不调用其他类的方法
    - 不和陌生人说话：前提：A->B, B->C　问题：A->C?　方法：A->B->C
    - 一个实体应当尽量少的和其他实体之间发生相互作用，使得系统功能模块相对独立。
- 5 分而治之 Divide and conquer
- 6 高类聚 high cohesion
    - 功能类聚，层类聚，通信类聚
- 7 低耦合 low coupling
    - 若耦合，危险度逐次降低
        - Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External
        - 比如静态变量，所有类可以访问，不好
        - 全局变量尽量不用
        - 封装实例变量，声明为private，提供get和set方法
- 补 依赖倒转原则（Dependence Inversion Principle）/ IOC
    - 这个是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖具体。
- 补 接口隔离原则（Interface Segregation Principle）/ 低内聚 高耦合
    - 使用多个隔离的接口，比使用单个接口更好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。
- 补 合成复用原则（Composite Reuse Principle）