高级软件工程

软件设计

彭鑫

pengxin@fudan.edu.cn

www.se.fudan.edu.cn/pengxin

内容提要

- 软件设计概述
- 模型和视图
- 质量属性
- 软件体系结构
- 设计评审

内容提要

- 软件设计概述
- 模型和视图
- 质量属性
- 软件体系结构
- 设计评审

问题

Hello World 需要软件设计吗？为什么？

复杂性：软件开发的本质挑战

Year	Operating System	SLOC (Million)
1993	Windows NT 3.1	4-5[1]
1994	Windows NT 3.5	7-8[1]
1996	Windows NT 4.0	11-12[1]
2000	Windows 2000	more than 29[1]
2001	Windows XP	45[2][3]
2003	Windows Server 2003	50[1]

Operating System	SLOC (Million)
Debian 2.2	55-59[4][5]
Debian 3.0	104[5]
Debian 3.1	215[5]
Debian 4.0	283[5]
Debian 5.0	324[5]
OpenSolaris	9.7
FreeBSD	8.8
Mac OS X 10.4	86[6][n 1]
Linux kernel 2.6.0	5.2
Linux kernel 2.6.29	11.0
Linux kernel 2.6.32	12.6[7]
Linux kernel 2.6.35	13.5[8]
Linux kernel 3.6	15.9[9]

软件规模

团队规模

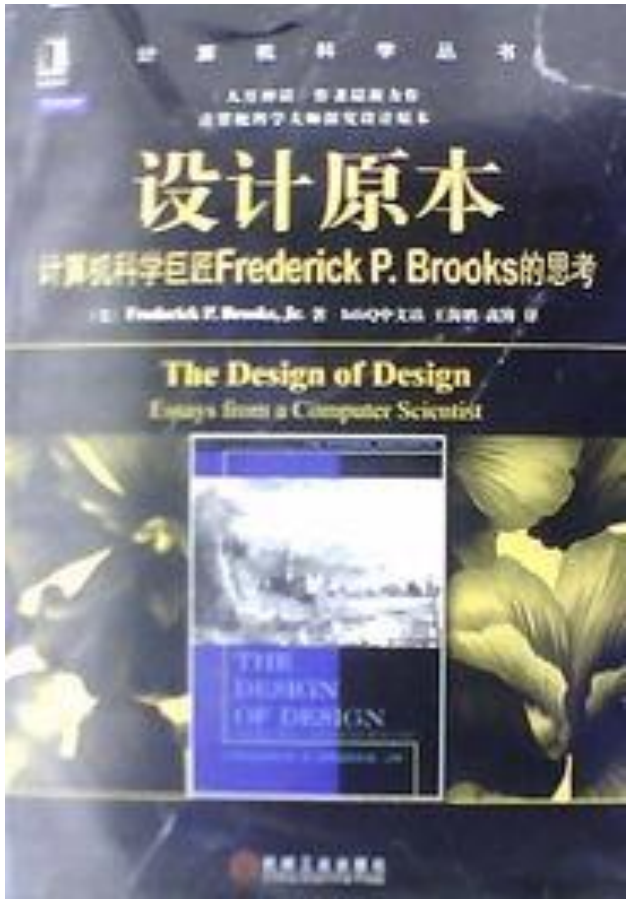
易变的需求

复杂的技术环境

数据来源
http://en.wikipedia.org/wiki/Source_lines_of_code

什么是设计?

什么是设计?



设计==搜索

根据问题的**目标**和**约束**，
在**解空间**中寻找**最优**解决方案的过程

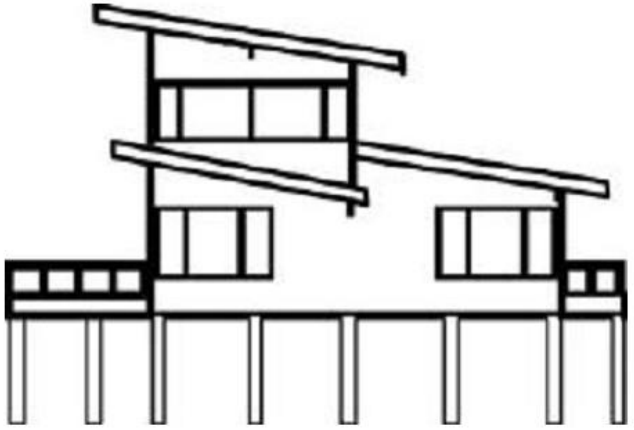
BROOKS海滨小屋的设计例子

海滨小屋的设计-1

- 目标
  - 面向大海的海滨小屋
  - 应该坚固，以抵御飓风，应具备至少14个人躺卧和就坐的空间，应为宾客提供难忘的风景
- 约束
  - 法律约束：小屋必须位于海滨场地的边界线后至少10英尺
  - 有一定弹性的约束：小屋应在春季前完工
  - 隐含的约束：小屋必须符合各种建筑法规

海滨小屋的设计-2

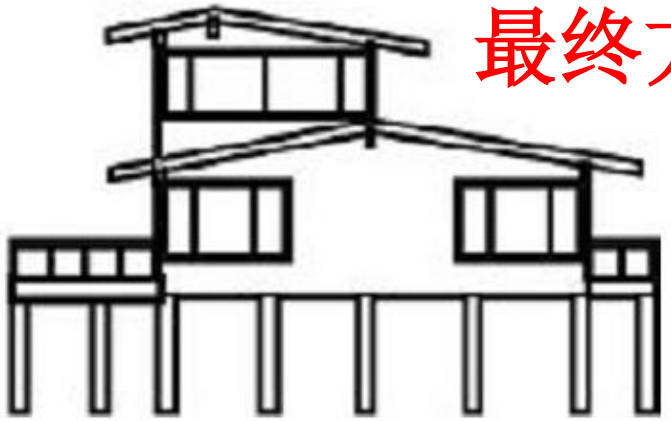
解空间（以屋顶的设计为例）



更多奇奇怪怪的屋顶…

海滨小屋的设计-3

- **如何选择最优解？**
  - 美观吗？
  - 房屋高度合适吗？
  - 容易漏雨吗？
  - 阻碍阳光吗？
  - …各种各样的效用评价



最终方案

设计的基本要素-1

- 目标
  - 短期目标：客户或项目价值
    - 明确的需求、隐含的需求
  - 长期目标：组织发展
    - 可理解、可维护、可扩展、可复用
- 约束
- 解空间
- 最优解决方案

设计的基本要素-2

- 目标
- **约束**
  - 商业约束-预算、竞争、项目期限…
  - 技术约束-技能、产品现状…

  思考：约束对设计而言都是坏事吗？

- 解空间
- 最优解决方案

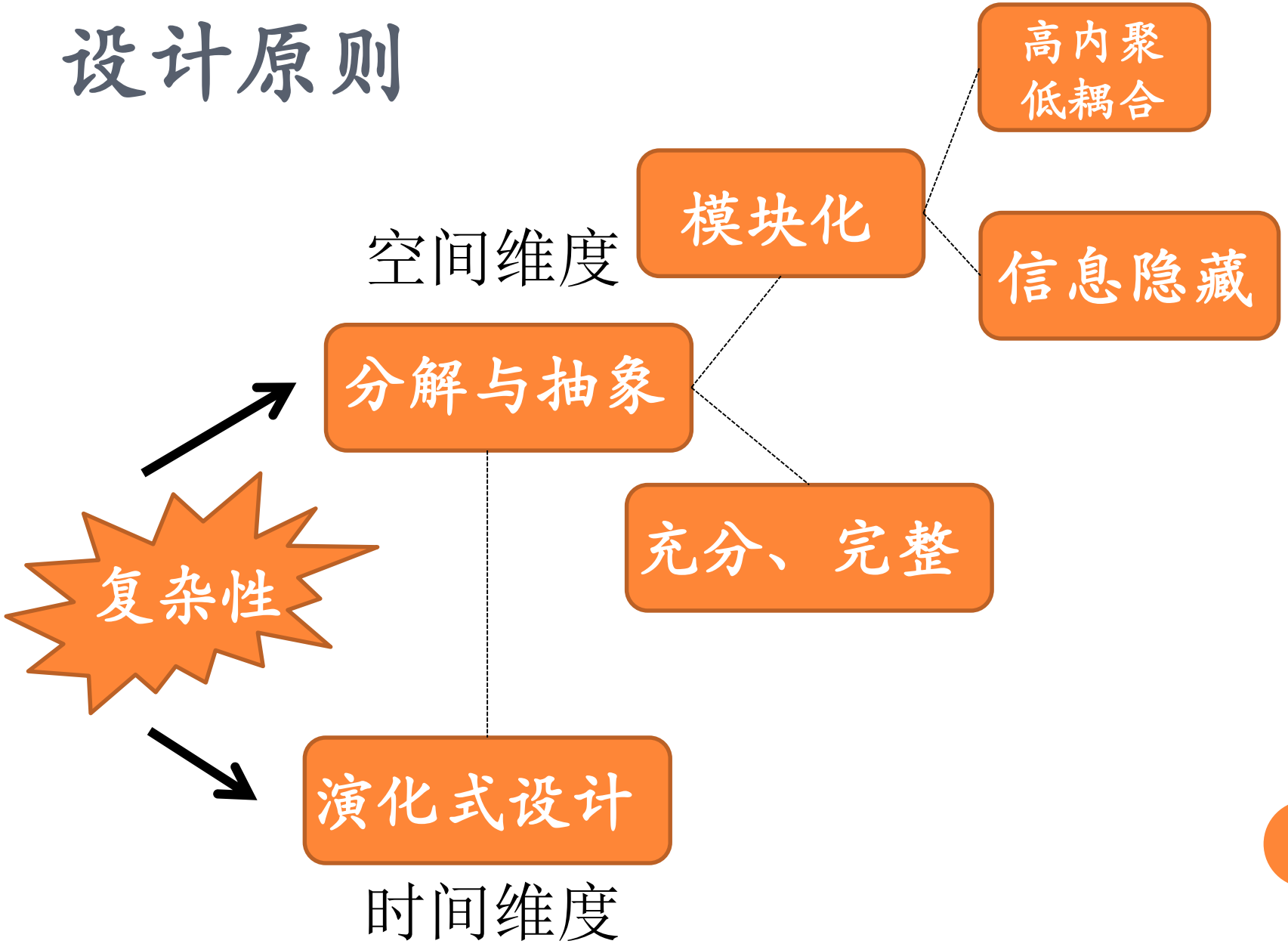
设计的基本要素-3

- 目标
- 约束
- **解空间**
  - 在设计中共通的东西占据了多数。
  - 即使是创新的设计，也是从前相似问题域的设计演化的结果
  - 好的设计者应该花费大量的时间来学习范例
- 最优解决方案

设计的基本要素-4

- 目标
- 约束
- 解空间
- **最优解决方案**
  - 效用函数描述了解决方案对目标的满足程度
  - 效用函数经常不是线性的
    - 例如，提高网站的响应性能对于改善用户的体验可能是有价值的
      提高更多的性能可能仍然能增加用户体验，但其对目标的满足程度逐步递减

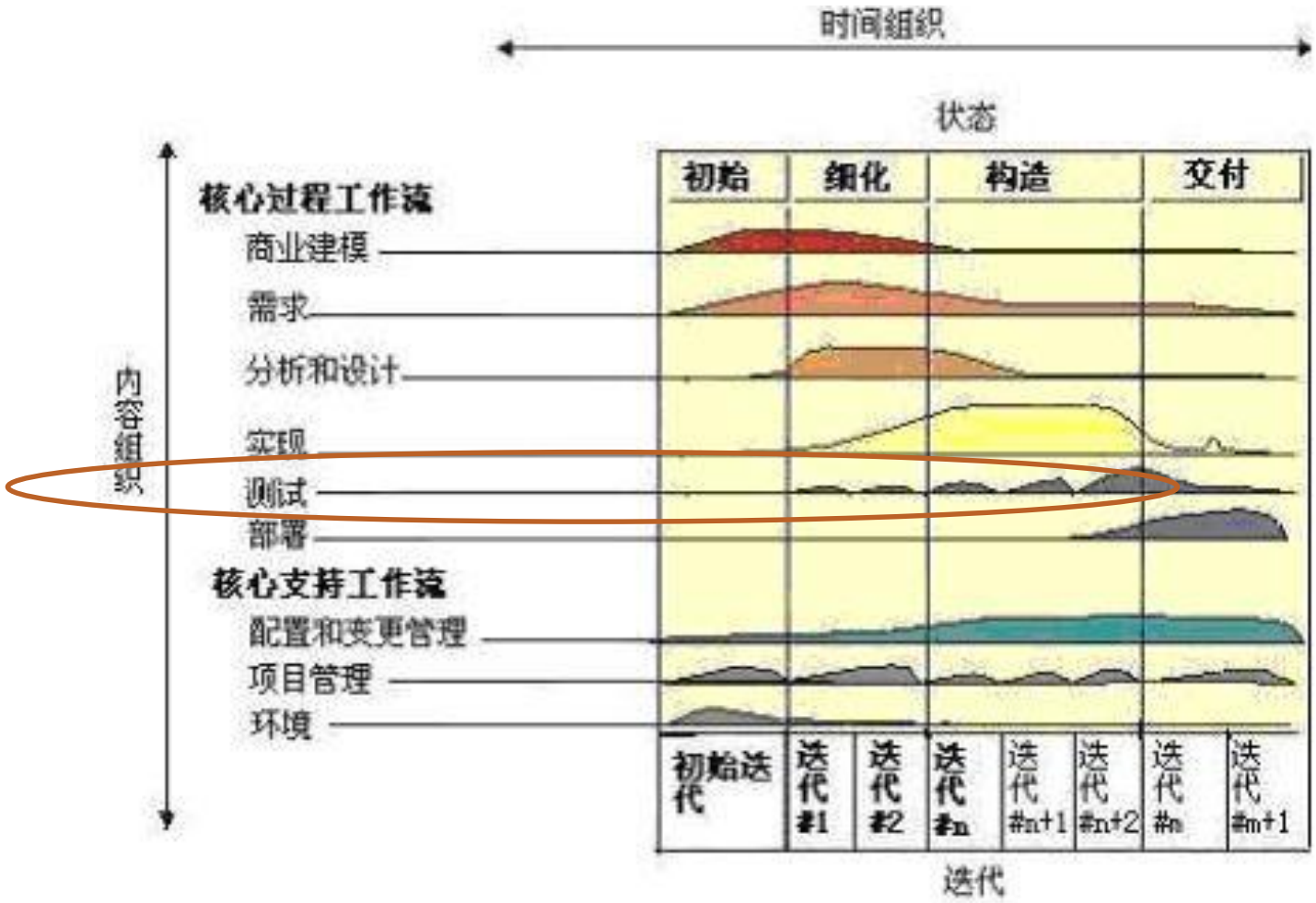
设计原则



过程模型-1

- 数据设计：软件系统的数据模型
- 体系结构设计：主要的构造元素之间的关系和设计约束
- 接口设计：软件和协作系统之间、软件和使用者之间的通信方式和接口
- 模块级设计：数据结构、算法、接口特征以及每个模块的具体实现方式

过程模型-2



UP过程模型：设计活动贯穿所有四个阶段的各个迭代

过程模型-3

- 敏捷方法
  - 反对BDUF (Big Design Up Front)
  - 强调LDUF（Little Design Up Front）
  - 强调设计的可演化特征：简单设计、重构、测试驱动开发、持续集成

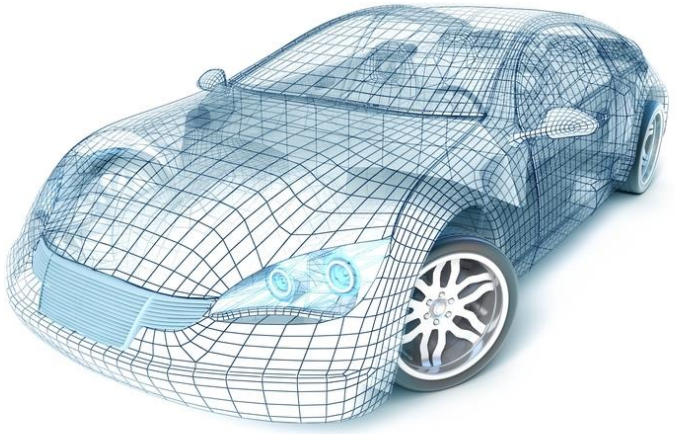
好的设计是演化出来的！

内容提要

- 软件设计概述
- 模型和视图
- 质量属性
- 软件体系结构
- 设计评审

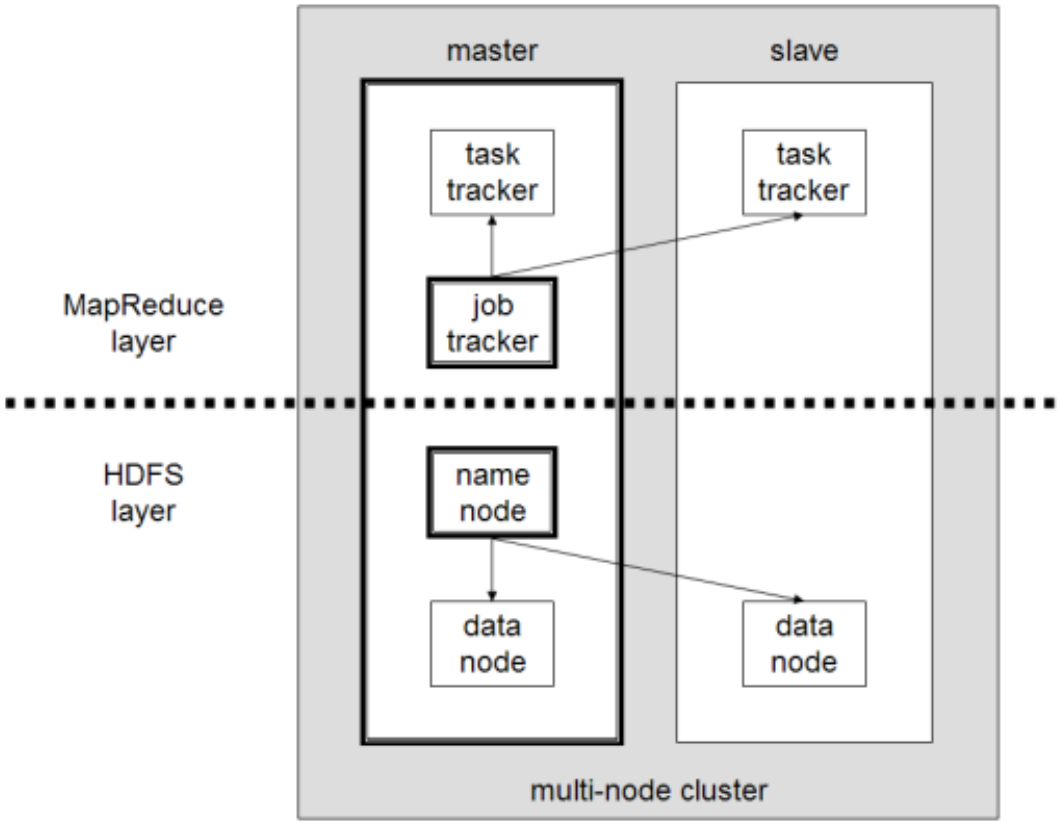
模型

- 按照需要对现实系统进行简化
- 模型的模拟功能
- 模型构造降低了系统的整体成本
  - 修改一个图纸或计算机上的模型，要比修改一个真实的系统要容易的多
  - 同时对多种可能的解决方案进行研究,发现最优解决方案
  - 涉众能够建立理解并且达成一致



案例研究：

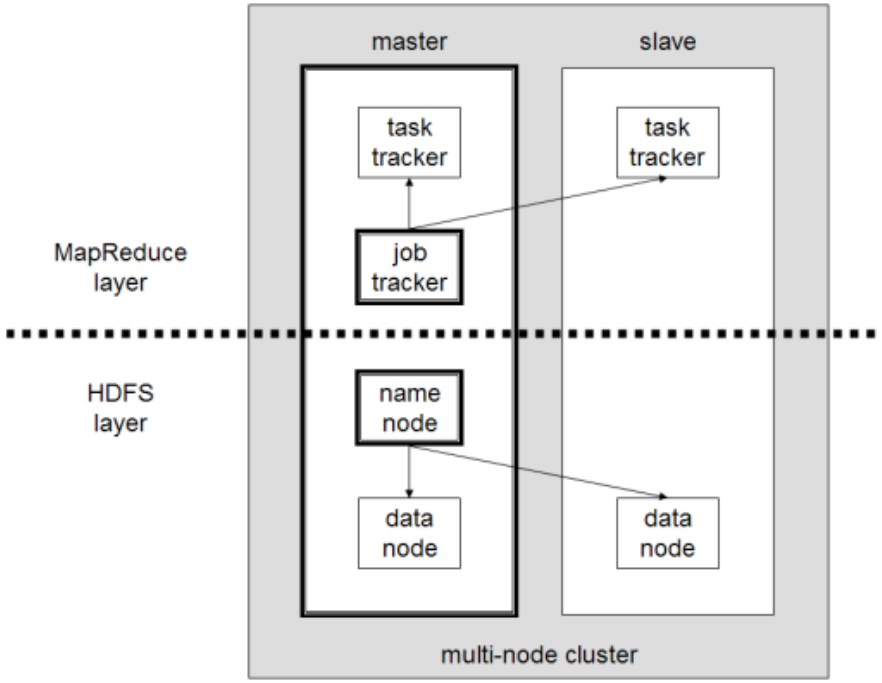
Hadoop是一个支持数据密集型应用的分布式系统框架，能够支持在成千上万个节点并行地对PB级的数据进行处理。



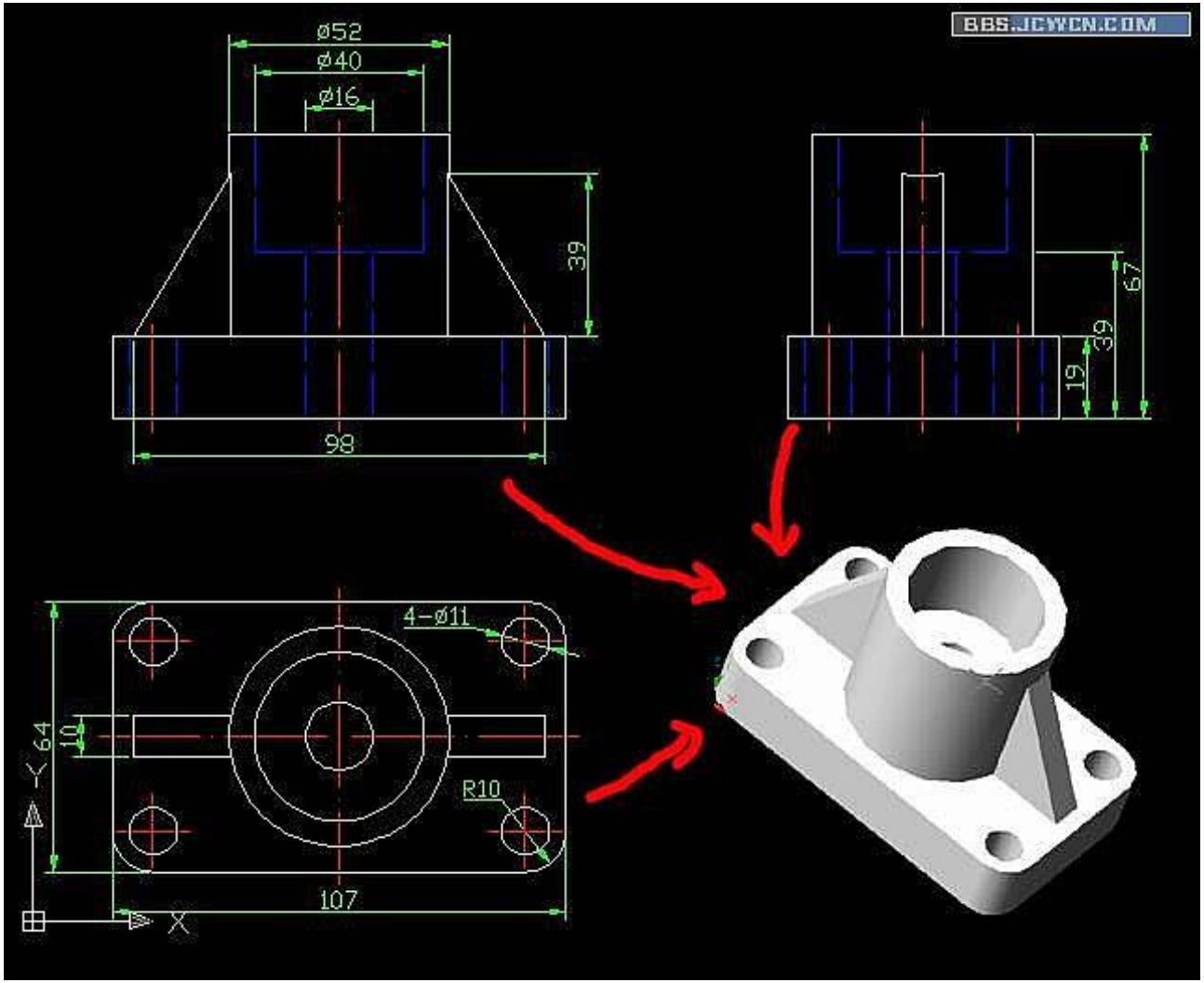
案例研究：

问题：

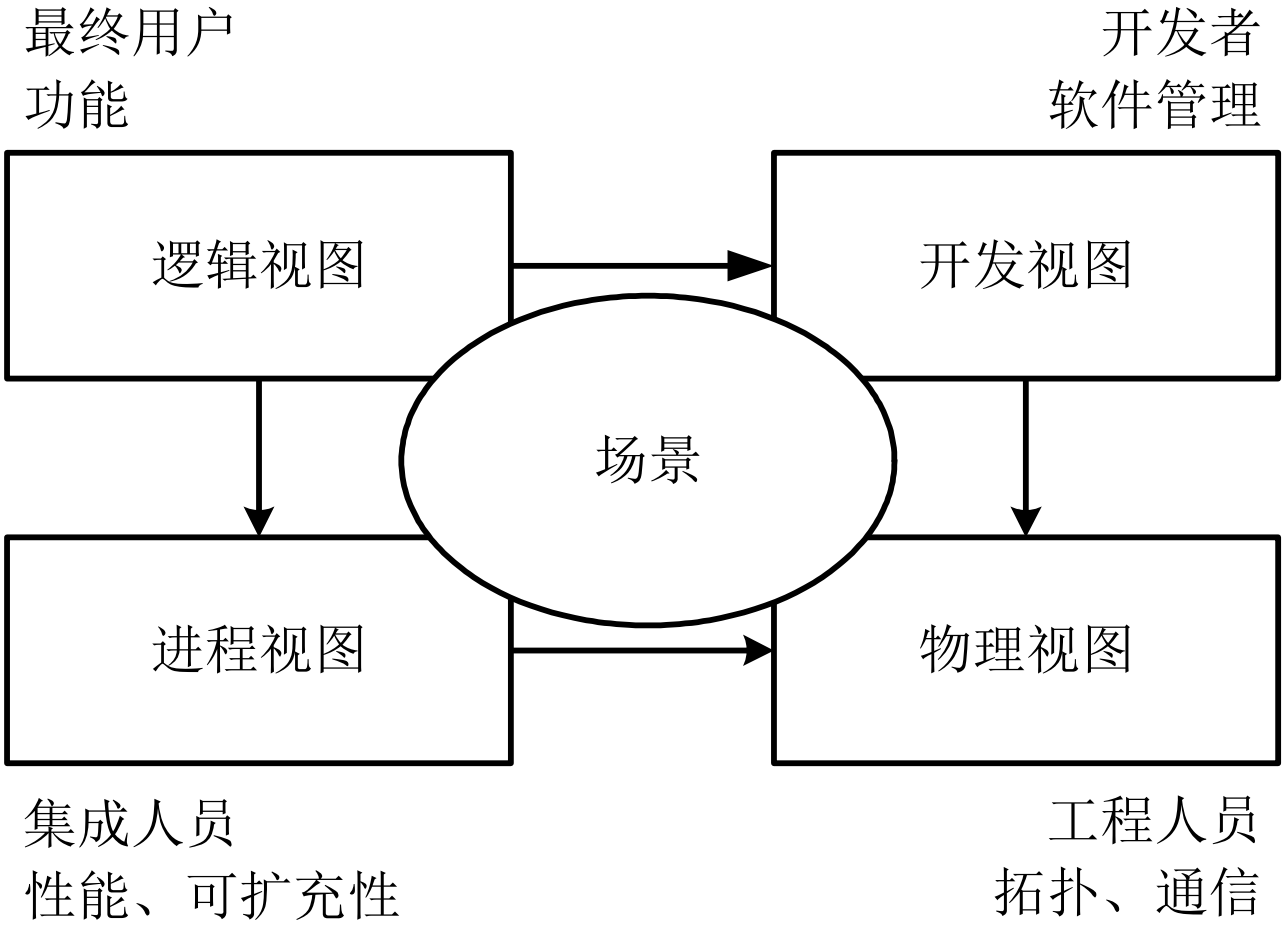
- 这张图说明了Hadoop的哪些部分的设计？
- 这张图哪些部分的说明是缺失的？
- 有哪些改进建议？



视图

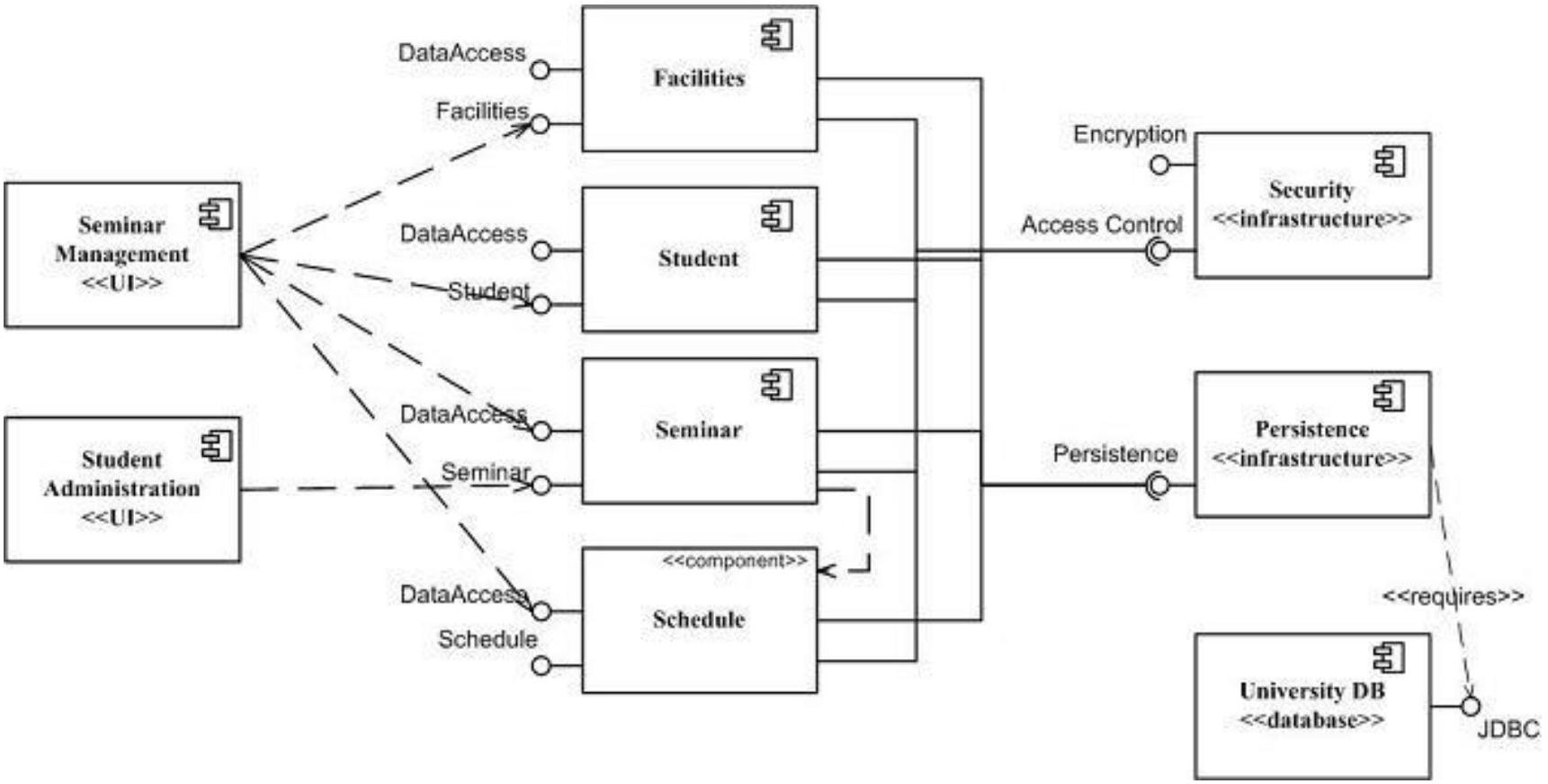


软件设计中的视图举例



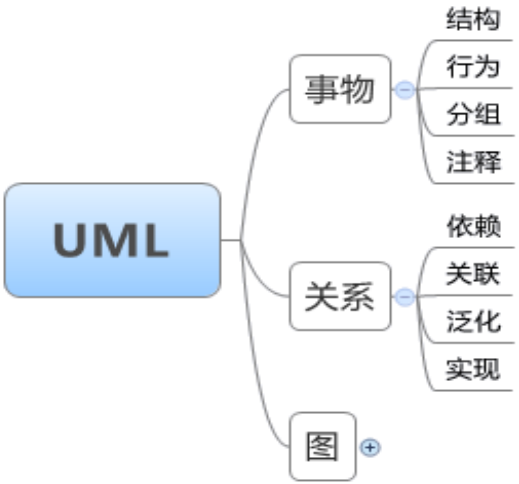
Kructen 4+1视图

UML建模语言

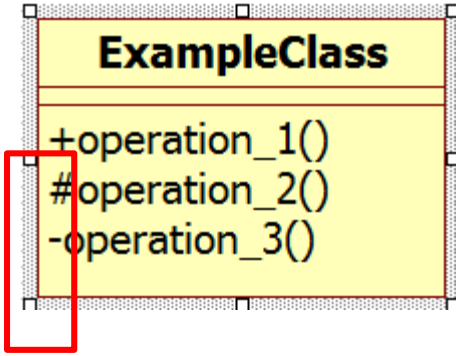

DataAccess
Facilities
Facilities
Seminar Management <<UI>>
DataAccess
Student
Student
Student Administration <<UI>>
DataAccess
Seminar
Seminar
DataAccess
Schedule
<<component>>
Schedule
Encryption
Access Control
Security <<infrastructure>>
Persistence
Persistence <<infrastructure>>
<<requires>>
University DB <<database>>
JDBC

UML不仅仅是图！

基本构造块

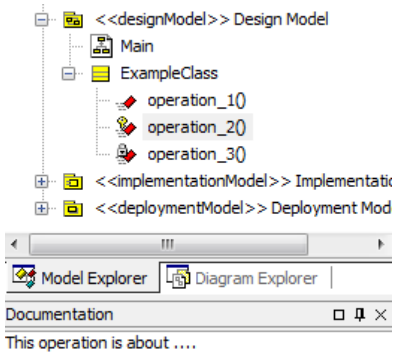


规则



命名
有效范围
可见性
完整性保证
异常

公共机制



详述
修饰
通用划分
扩展

UML基本构造块-事物

- 结构：描述模型中的静态部分，例如用况、类、接口、构件、结点
- 行为：描述模型中的动态部分，例如交互和状态
- 分组：用于组织UML模型，包括UML包及其变体（如子系统）
- 注释：描述、说明和标注UML模型的元素

UML基本构造块-关系

- 依赖（dependency）
- 关联（association）
- 泛化（generalization）
- 实现（realization）

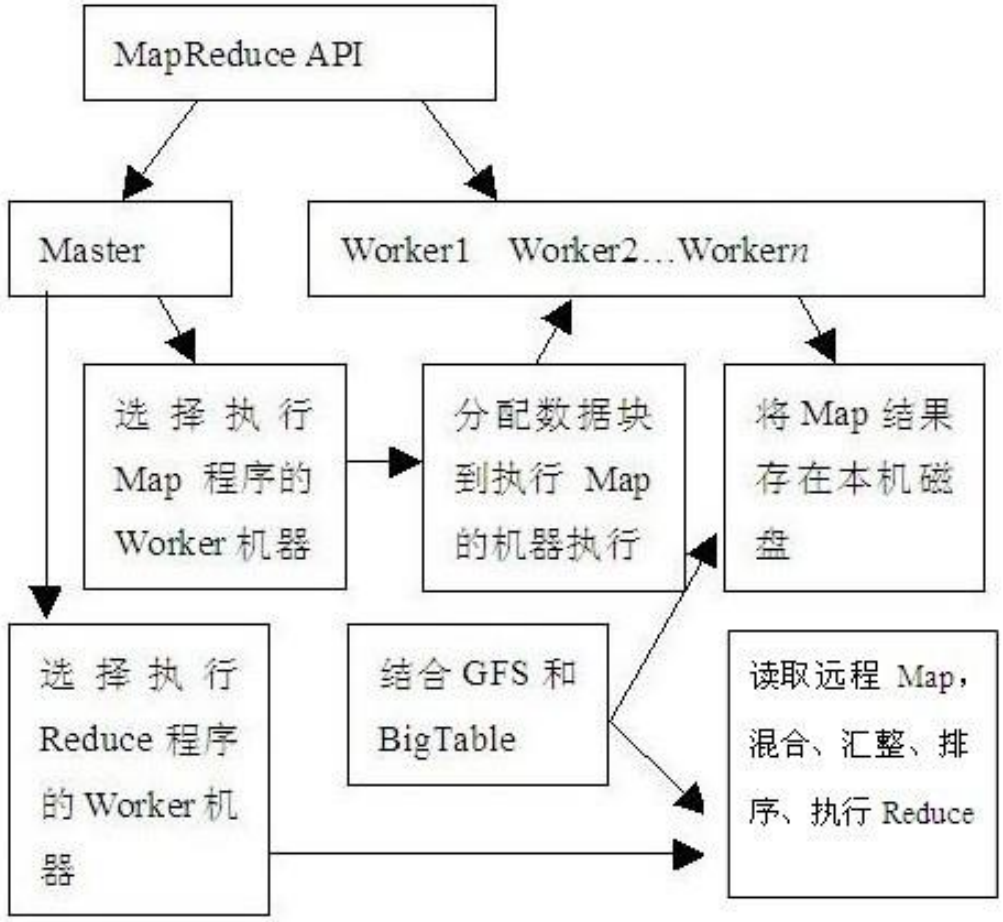
UML基本构造块-图

类型	图	主要模型元素	说明
结构	用例图	用例、执行者、关联、扩展、包含、泛化	描述需求
	类图	类、关联、泛化、依赖、实现、接口	描述需求的概念模型和设计的内部结构
	对象图	对象、链接	对象图是类图的特殊情况
	复合结构图	连接器、接口、部件、端口、角色、供给和需求接口	复合结构图和类图没有明显的界限
	组件图	组件、依赖、端口、供给和需求接口、实现、子系统	定义组件的类型、内部结构和依赖
	部署图	节点、组件、通信、依赖	显示物理实体或组件在计算资源（节点）上的物理部署

UML基本构造块-图

行为	通信图	协作、消息、角色、序号、守护条件	描述系统中的各种类型的对象之间的交互。通信图和顺序图的呈现焦点不同，但二者在语义上等价
	顺序图	交互、消息、信号、生命线、发生说明、执行说明、交互片段、交互操作域	
	定时图	状态、生命线、消息、时间单位	是顺序图的另外一种描述方式，用来刻画实时行为。
	状态机图	状态、时间、转换、效果、执行活动、触发	描述具有复杂状态的对象的状态变迁行为。
	活动图	动作、活动、控制流、控制节点、数据流、分叉、结合、异常、对象节点	描述了一个较大粒度的可执行行为的次级单元的协同执行过程。用于需求分析和内部实现的建模
	交互概览图	活动图元素和嵌套的序列图	是活动图和顺序图的混合物，用于将活动图中活动节点的控制流机制和序列图中的消息序列结合。
分组	包图	导入、包、模型	通过分组对模型进行管理

案例研究：



左图不是一个UML图。尝试将其分别改成UML顺序图和活动图。

内容提要

- 软件设计概述
- 模型和视图
- 质量属性
- 软件体系结构
- 设计评审

需求对设计的影响

- **功能性需求**
  - 容易发生变化
  - 要求具备良好可扩展性和适应性的软件设计，从而尽量保持设计（特别是高层体系结构）的稳定
- **非功能性需求**
  - 决定软件设计、特别是重要设计决策的首要因素
  - 非功能性需求的实现往往具有挑战性
  - 不同的非功能性需求之间经常存在冲突

思考：设计一个车票预订功能

- 设计一个基本的车票预订功能
- 设计一个能保证并发一致性且能支持百人规模的并发访问的车票预订功能
- 在有限的硬件资源情况下，设计支持百万级并发访问且保证高可靠性(99.99%)的车票预订功能

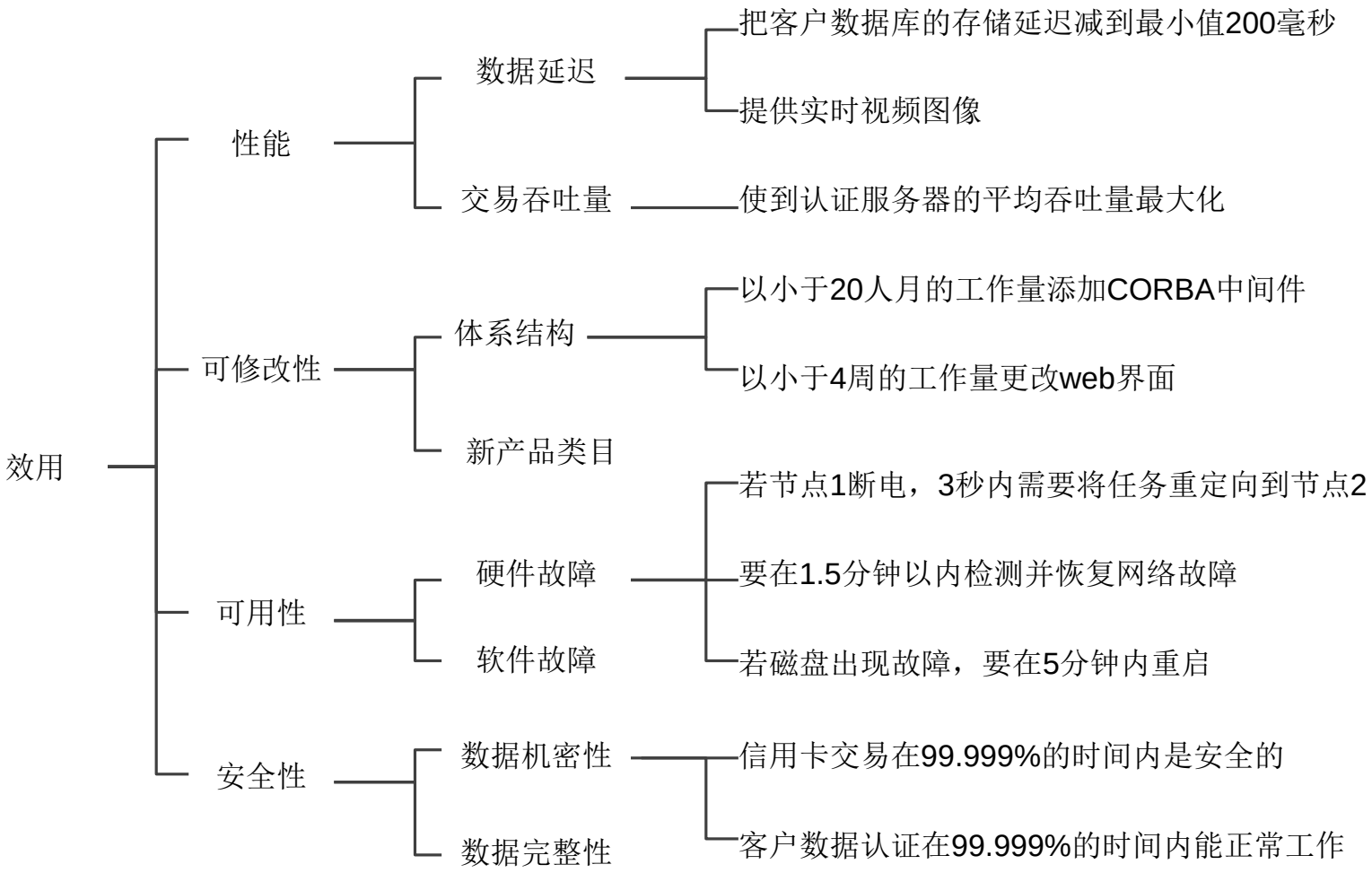
ISO/IEC25010

《系统和需求质量模型》

设计中的质量属性考虑

- 对该产品，有哪些关心的质量属性？
- 如何把这些质量属性说清楚？
- 当这些质量属性发生冲突的时候，如何排列优先级？

质量属性效用树


效用
性能
数据延迟
把客户数据库的存储延迟减到最小值200毫秒
提供实时视频图像
交易吞吐量
使到认证服务器的平均吞吐量最大化
可修改性
体系结构
以小于20人月的工作量添加CORBA中间件
以小于4周的工作量更改web界面
新产品类目
可用性
硬件故障
若节点1断电，3秒内需要将任务重定向到节点2
要在1.5分钟以内检测并恢复网络故障
软件故障
若磁盘出现故障，要在5分钟内重启
安全性
数据机密性
信用卡交易在99.999%的时间内是安全的
数据完整性
客户数据认证在99.999%的时间内能正常工作

三类质量属性场景

- 用例场景
- 变更场景
- 探索性场景

用例场景

用户期望的使用场景

在系统处理的高峰期，用户通过Web请求购买商品，应该能够在5秒钟内获得响应。

变更场景

- **在可预期的未来对系统的扩充或者更改，例如**
  - 预期的功能添加和修改、性能和可用性的变化
  - 向其他平台的移植、与其他软件进行集成

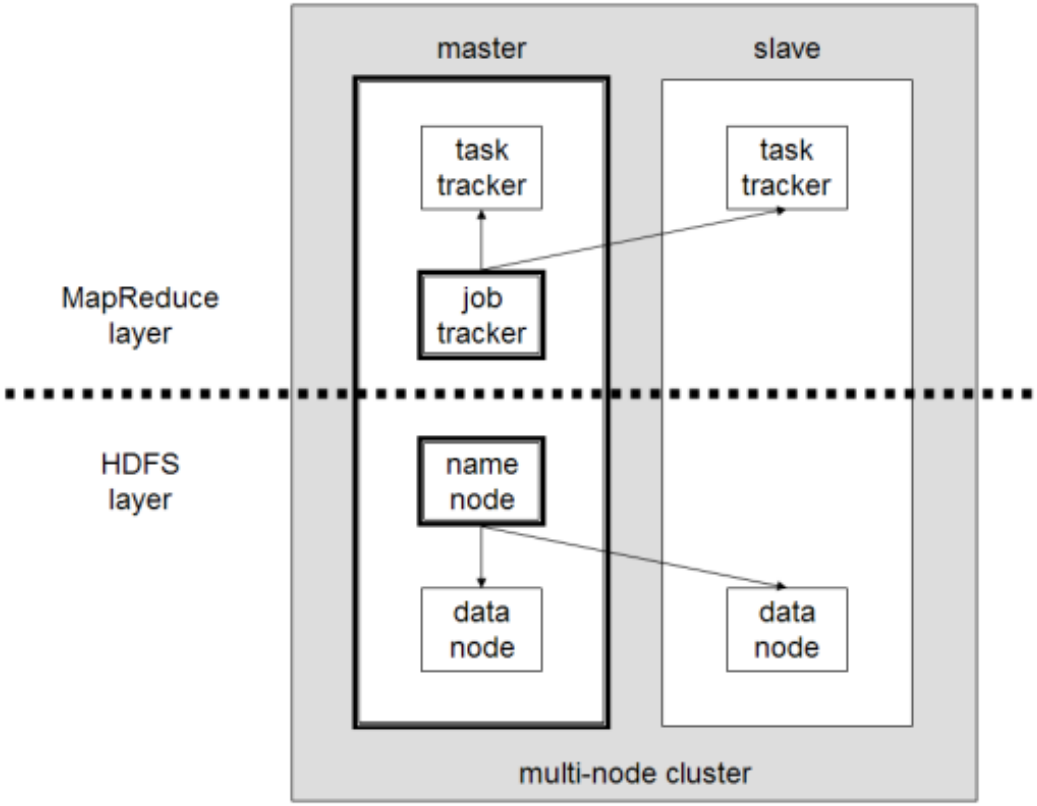
把当前系统的操作环境（Windows）迁移到Linux环境，系统作出修改所需的工作量应该小于1人年。

探索性场景

- 也是一种变更场景，发生的概率很低
- 主要用于探寻如果这种场景发生，当前的设计是否能够适应

案例研究：hadoop

- 问题：尝试列出Hadoop的一些重要质量属性和场景



内容提要

- 软件设计概述
- 模型和视图
- 质量属性
- 软件体系结构
- 设计评审

软件体系结构的定义

系统在某种环境下的一组最基本的概念和属性，包括了元素、关系以及设计和演化的原则。

ISO/IEC42010-2011：系统和软件工程-体系结构描述

软件体系结构表示了形成一个系统的重要设计决策。重要程度根据变化的成本来衡量。

Grady Booch

在项目中的价值

- 降低项目开发的复杂度和风险
- 支持项目的管理、跟踪和风险防范
- 支持系统的演化和复用

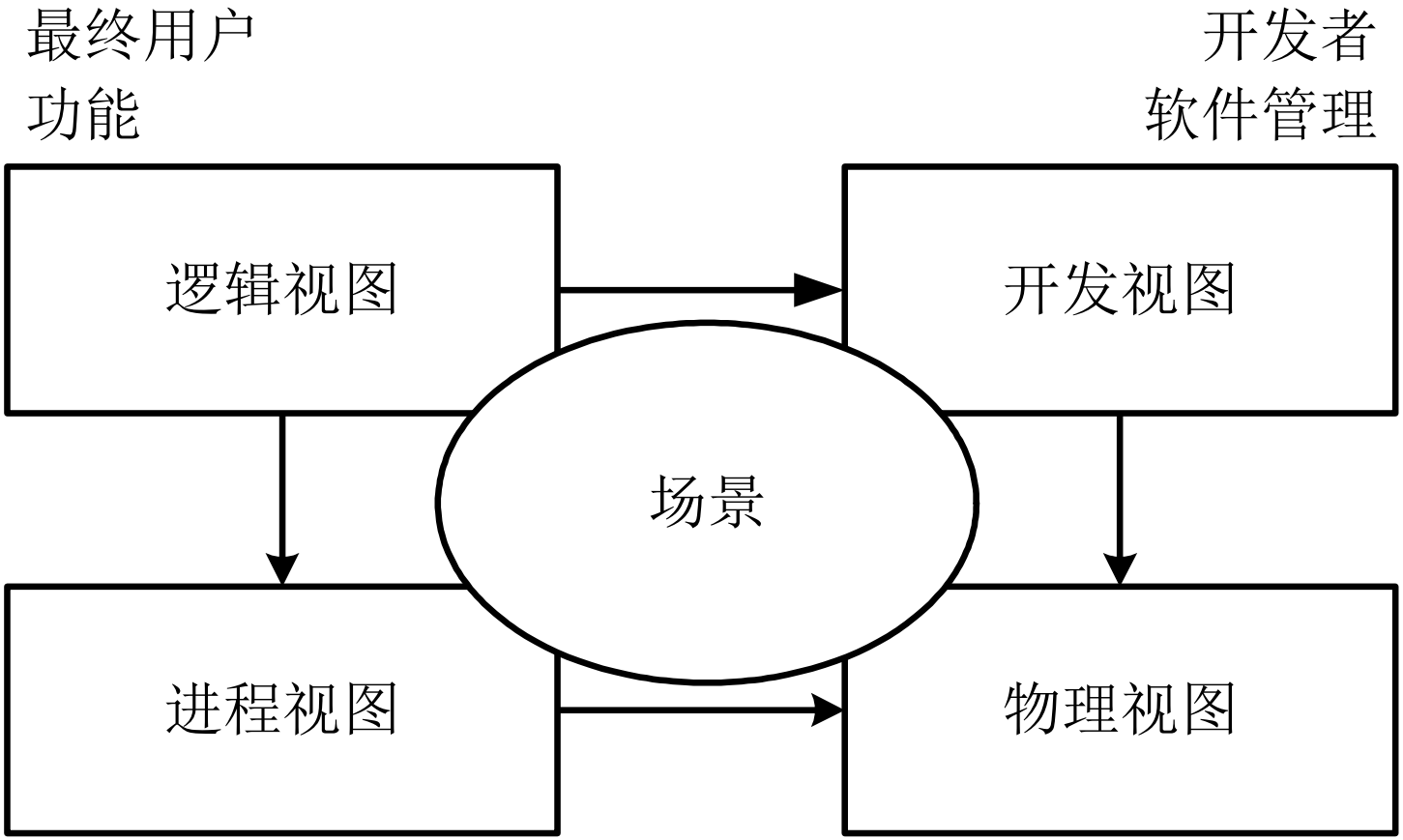
体系结构的核心内容-1

- **描述了系统的分解和抽象**
  - 软件体系结构是对软件元素、元素的外部可见属性，以及它们之间的关系的描述。
  - 降低了软件系统开发的复杂性
  - 更容易的对开发任务作出分解
- **定义了设计和演化的原则**
  - 例如，编写代码使用的语言、编程规范、采用的数据库类型、数据库访问的方法、如何进行出错处理、接口之间应该遵循的通信协议等

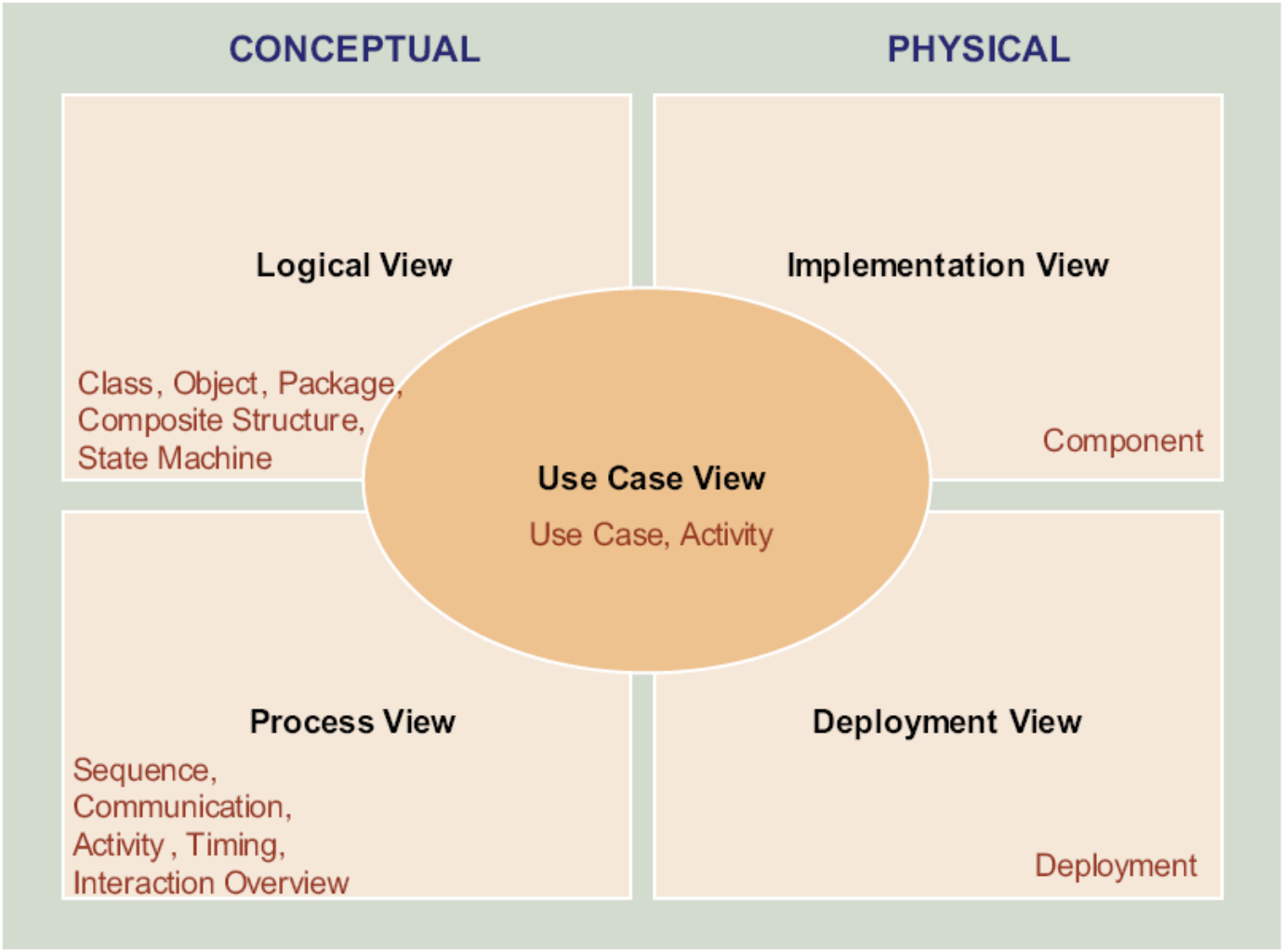
体系结构的核心内容-2

- **反映重要的设计决策**
  - “重要”与否和设计决策的影响面和变化成本相关
- **与特定的环境相关**
  - 商业环境、技术环境以及软件组织的环境等

体系结构的4+1视图

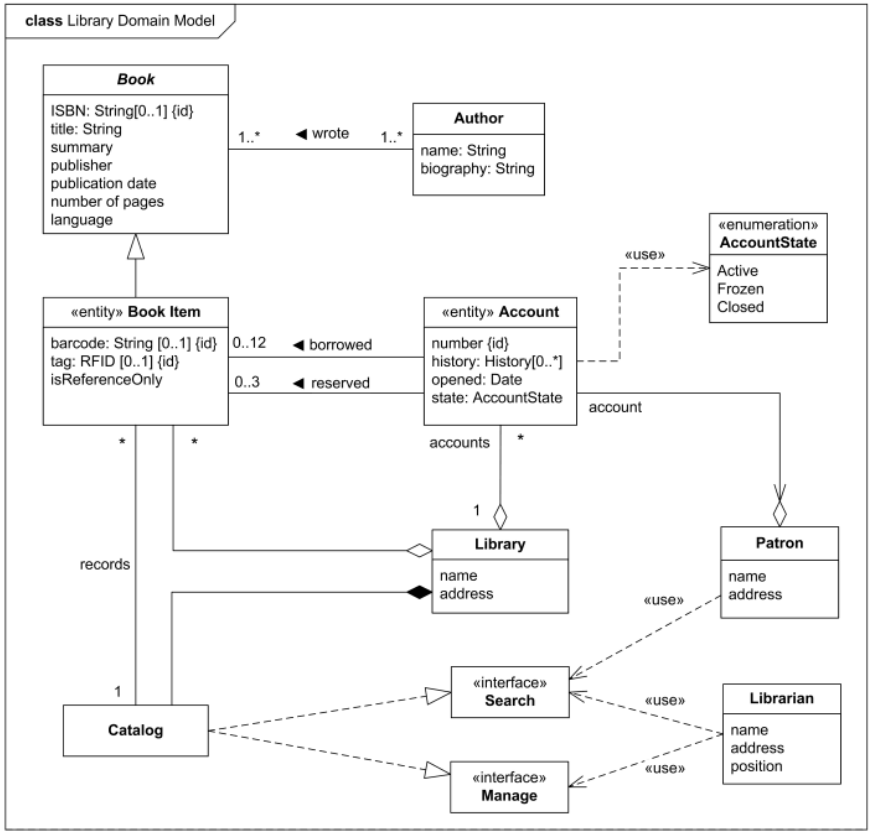


4+1视图的UML描述


CONCEPTUAL
PHYSICAL
Logical View
Class, Object, Package,
Composite Structure,
State Machine
Implementation View
Component
Use Case View
Use Case, Activity
Process View
Sequence,
Communication,
Activity , Timing,
Interaction Overview
Deployment View
Deployment

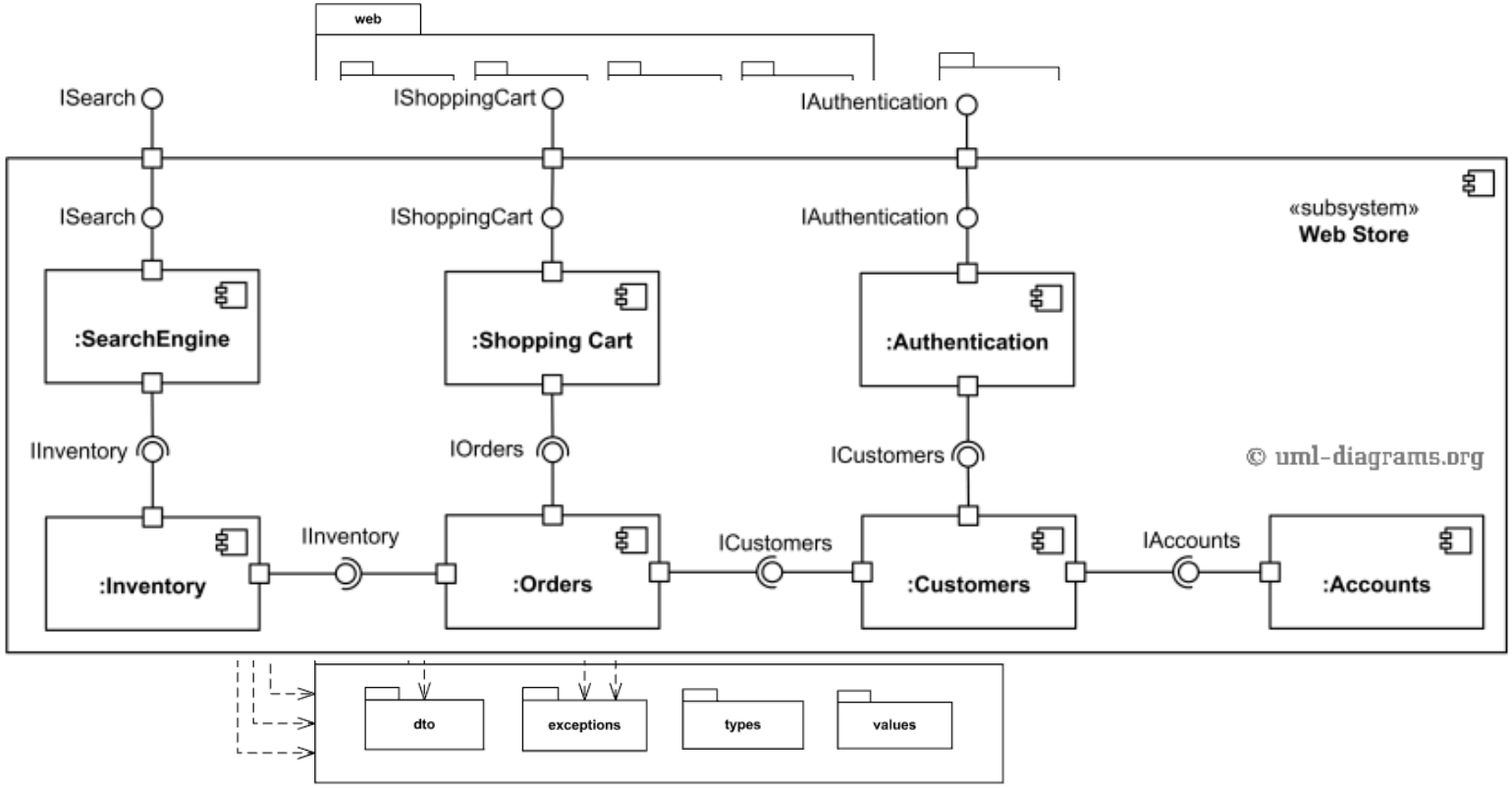
逻辑视图

- 从最终用户的角度描述系统应提供的功能
- 将系统系统需要支持的功能需求定义为一系列功能抽象以及它们之间的相互关系



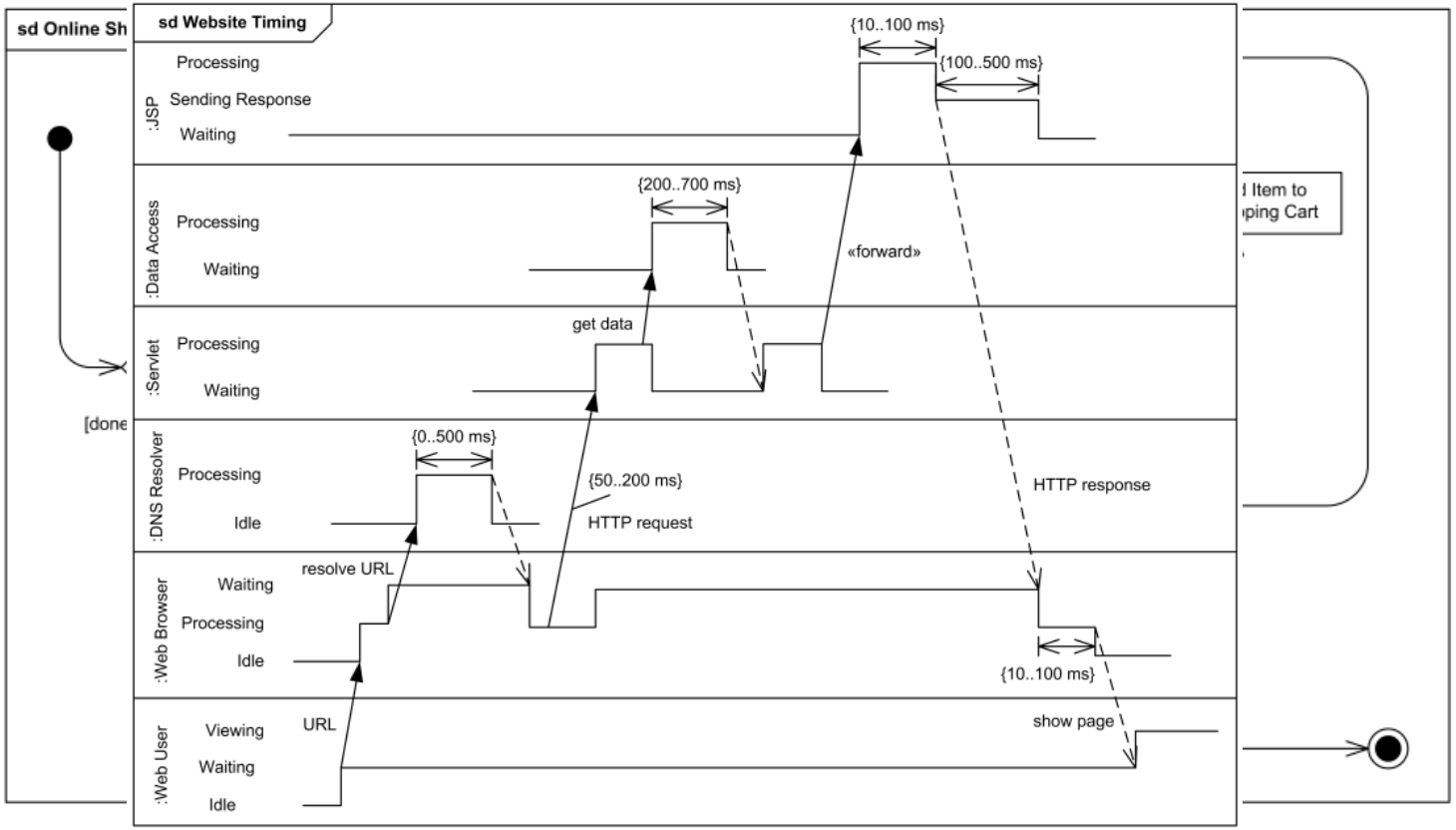
开发视图

- 描述了软件模块的组织和管理结构，作为后续软件开发的基础



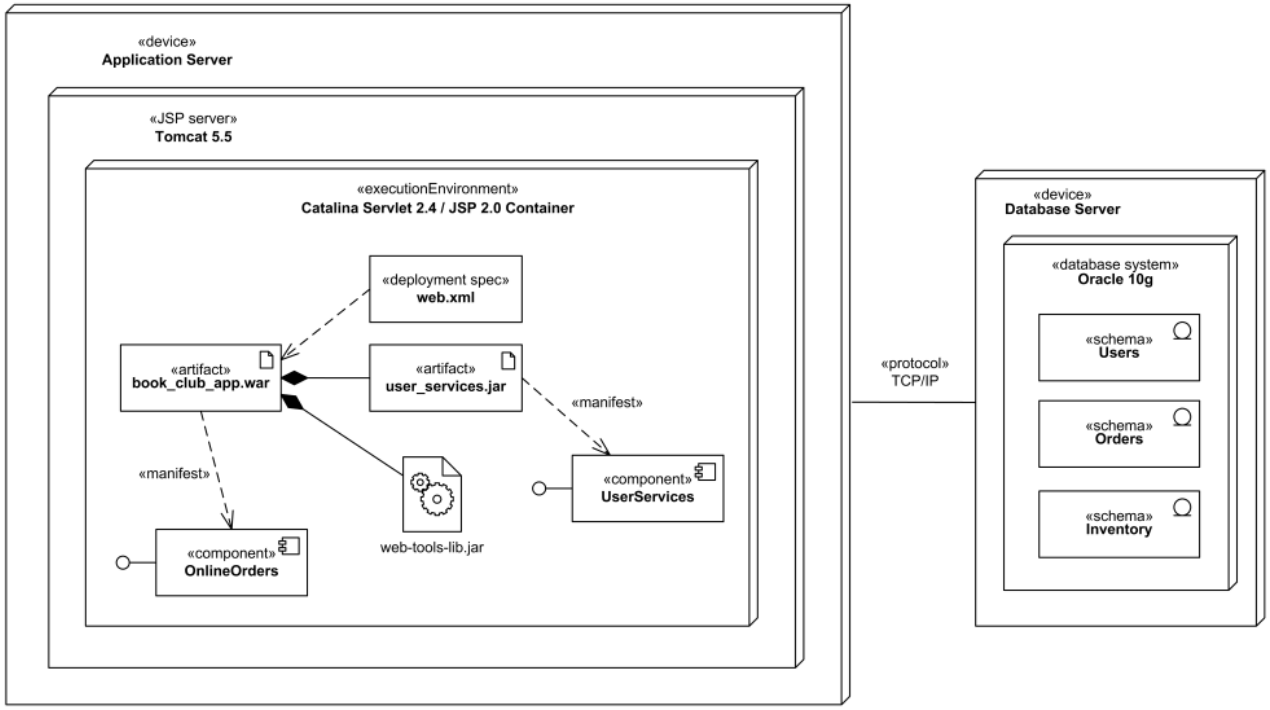
进程视图

- 从运行时角度描述软件系统的动态行为，特别是系统的并发和同步方面的设计



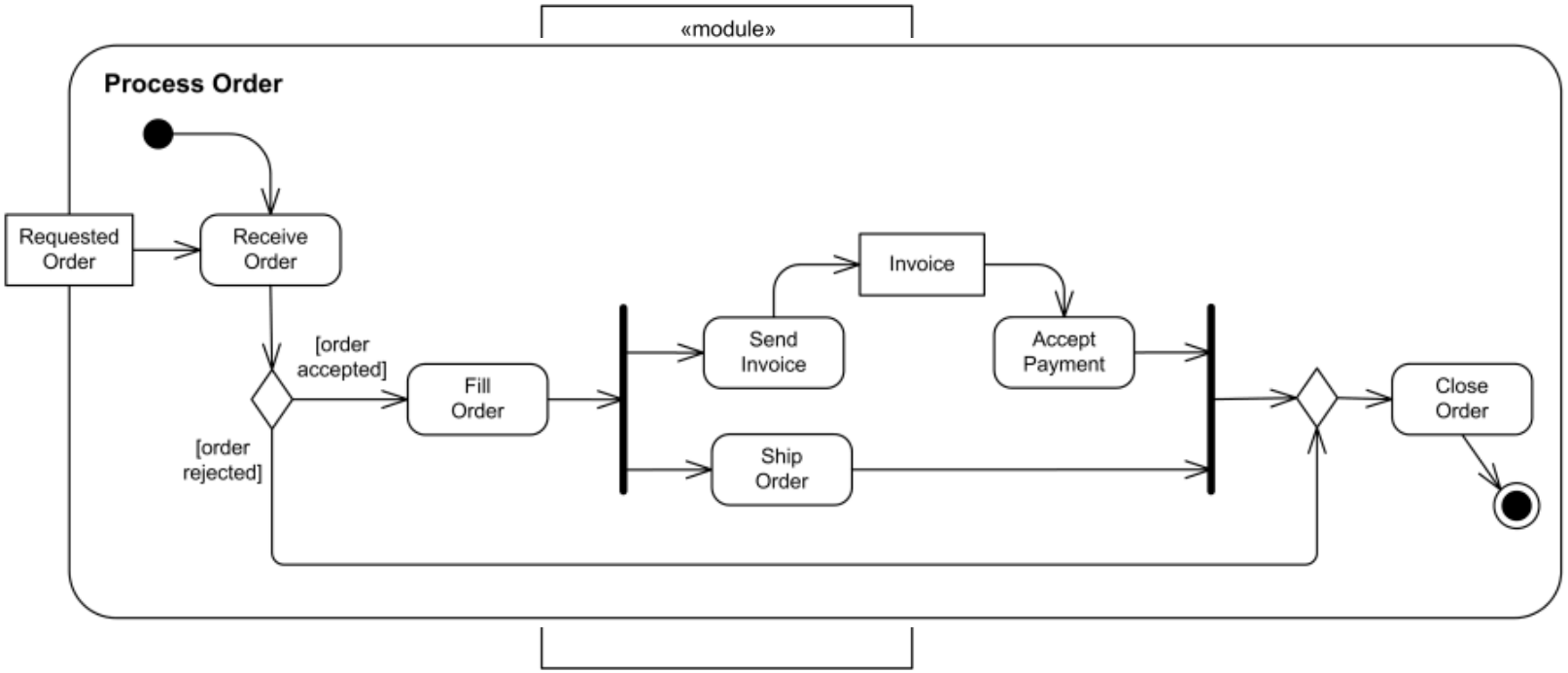
物理视图

- 也被称为部署视图
- 从系统工程人员的角度来描述系统的硬件部署结构，包括拓扑结构、系统安装及相互通信等



场景视图

- 主要系统活动的场景化描述，定义了在特定场景下对象与对象之间、进程与进程之间的交互关系
- 位于各视图的核心，将不同视图的联系起来，以场景为基础对各个体系结构视图进行分析



风格、模式和框架

风格

- 风格：定义了软件系统的某种特定结构，其组织方式有着鲜明的特点
- 典型的风格：管道和过滤器、分层、仓库、基于事件的隐式调用、面对对象、表驱动以及分布式处理、主程序-子进程、特定领域的体系结构、状态迁移、进程控制等

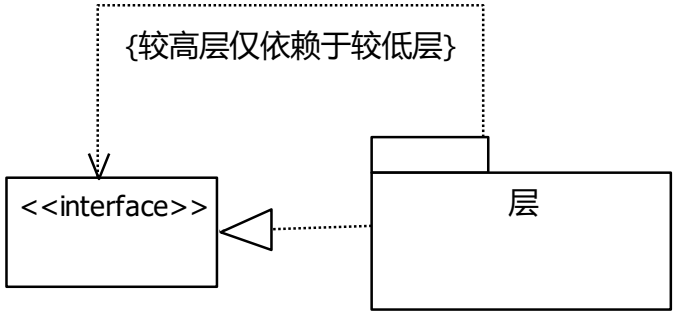
模式

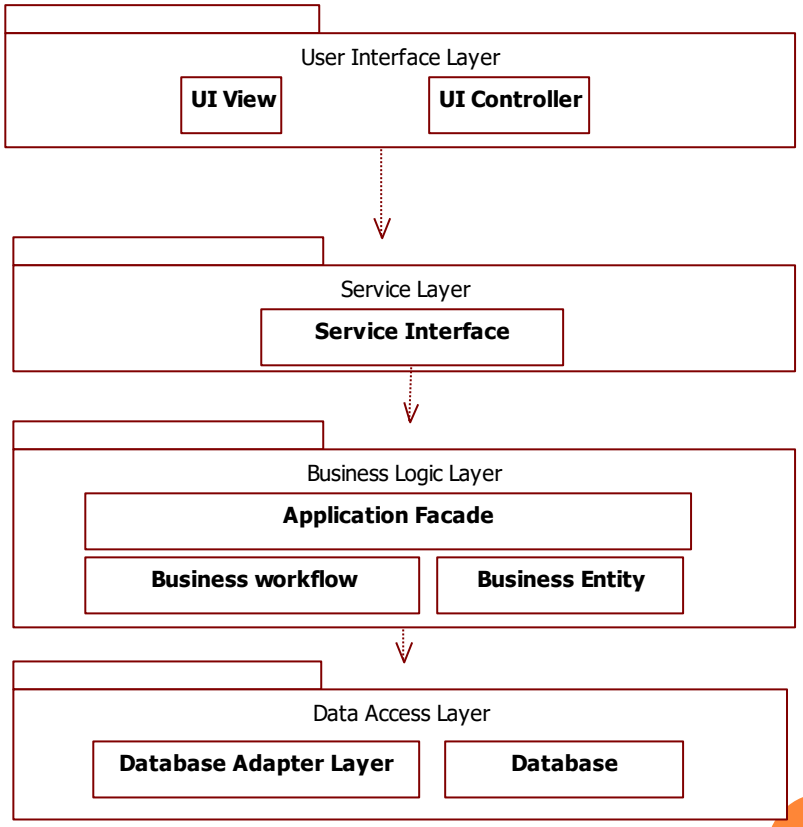
- 从要解决的问题和上下文的角度研究问题域的共通性
- 复用既有的解决方案
  - 体系结构模式
  - 分析模式
  - 设计模式
  - 实现模式等

框架

- 适用于特定领域的一种设计复用机制
- 让设计人员能够更多关注于和具体应用相关的设计，而不是处理那些通用的、较低层次的技术细节
- 往往能够提供包含通用部分的框架实现

分层体系结构模式-1

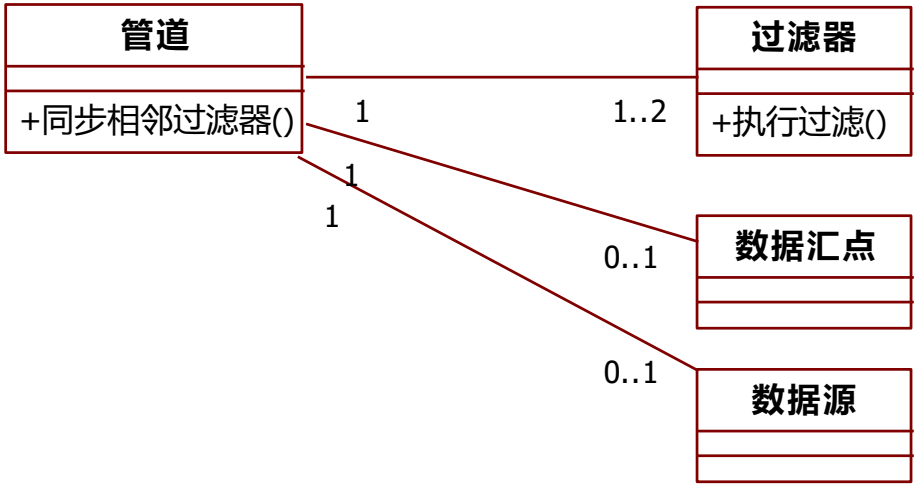

{较高层仅依赖于较低层}
<<interface>>
层

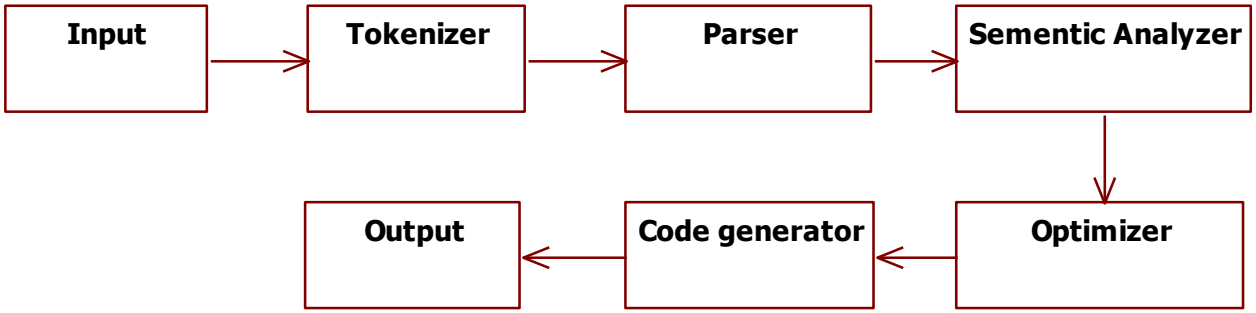

User Interface Layer
UI View
UI Controller
Service Layer
Service Interface
Business Logic Layer
Application Facade
Business workflow
Business Entity
Data Access Layer
Database Adapter Layer
Database

分层体系结构模式-2

- 提供了清晰定义的抽象层，可以降低软件开发任务的耦合
- 标准化的抽象接口实现了层间模块的可替换性
  - 通过标准化的层间接口，可以将由于硬件、操作系统、网络协议、数据库等因素造成的影响限制在一层以内
  - 提高了系统的可移植性和变更影响影响的局部化
- 由于标准化的层间接口的存在，可以在开发阶段分别对系统中的某一层进行独立测试，有助于在早期、以较低的成本发现设计和实现的缺陷

管道和过滤器模式-1

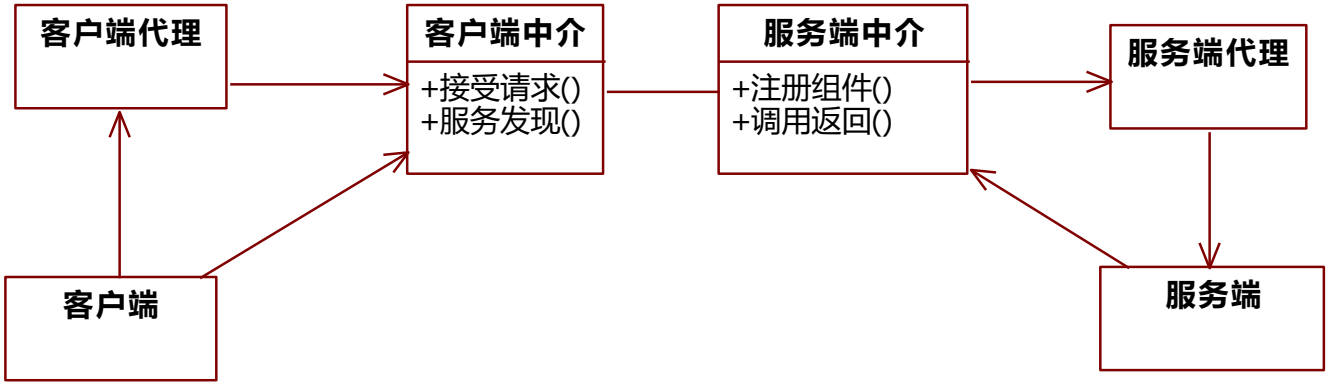

管道
+同步相邻过滤器()
过滤器
+执行过滤()
1
1..2
1
1
0..1
0..1
数据汇点
数据源

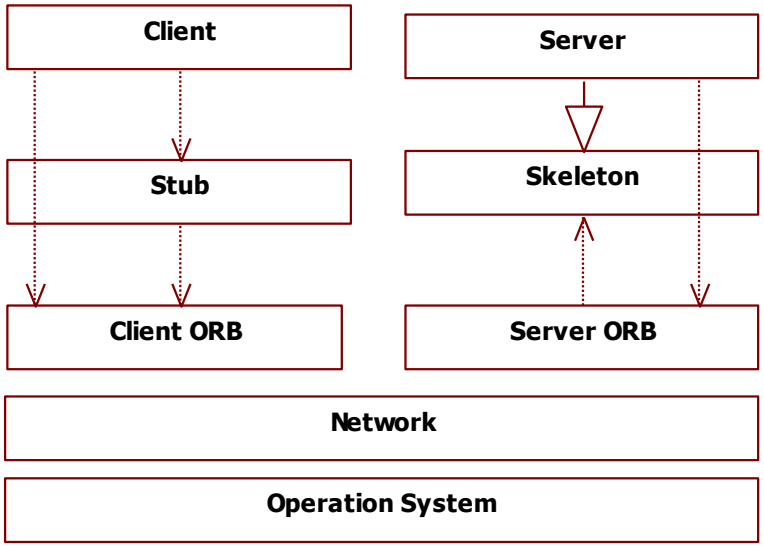

Input
Tokenizer
Parser
Sementic Analyzer
Optimizer
Code generator
Output

管道和过滤器模式-2

- **适用于数据流处理的环境**
  - 其特点是单向的数据流
  - 可以将系统的任务自然就分解成几个处理阶段
  - 系统的功能是多个任务的简单组合
- **组成单元**
  - 过滤器封装了对数据的具体的处理步骤，它顺序地处理每个数据，将未处理过的输入数据转换为经过加工的输出数据
  - 管道表示过滤器之间的连接。管道实现了相连的处理步骤之间的数据的流动
  - 数据源是数据流的输入
  - 数据汇点是数据处理的最终输出

中介模式-1

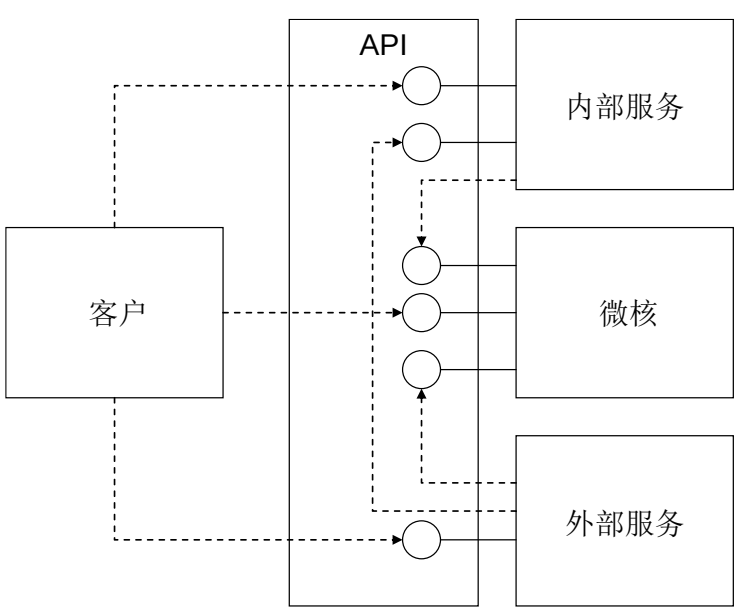

客户端代理
客户端中介
+接受请求()
+服务发现()
服务端中介
+注册组件()
+调用返回()
服务端代理
客户端
服务端

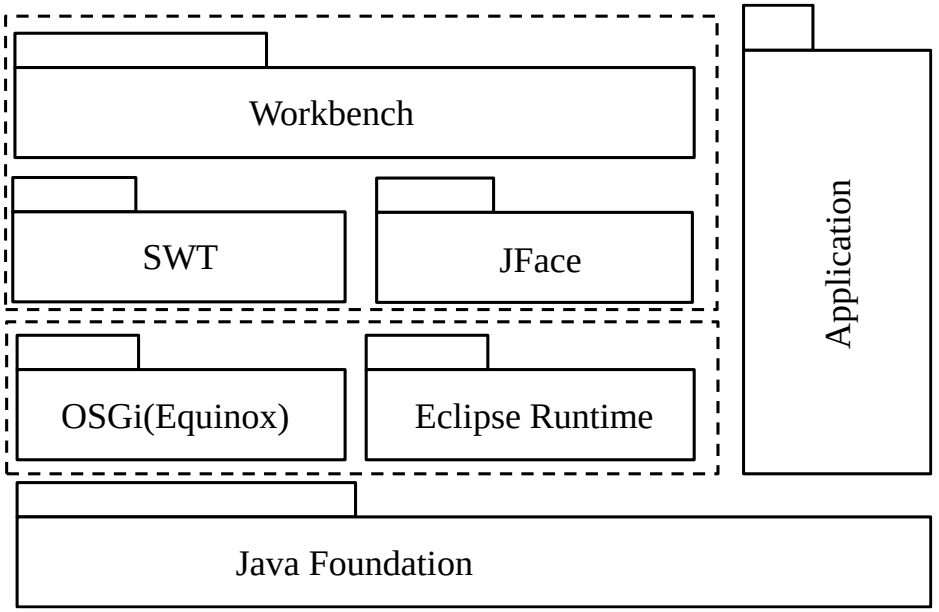

Client
Server
Stub
Skeleton
Client ORB
Server ORB
Network
Operation System

中介模式-2

- **降低了开发分布式应用程序的复杂性**
  - 应用程序组件可以简单地向合适的对象发出消息来调用分布式服务，无需把精力放在底层通信等无关的细节上
- **提供了定位的透明性**
  - 客户端并不需要知道服务器的位置
  - 服务端也不需要关心调用它的客户端的位置
  - 为灵活的部署提供了可能。
- **使得服务端可以灵活的修改和扩展：需要保证服务端接口设计的稳定性**

微内核模式-1





- 分离软件的核心部分和应用部分
- 支持应用部分的扩展和定制

Eclipse的微内核结构

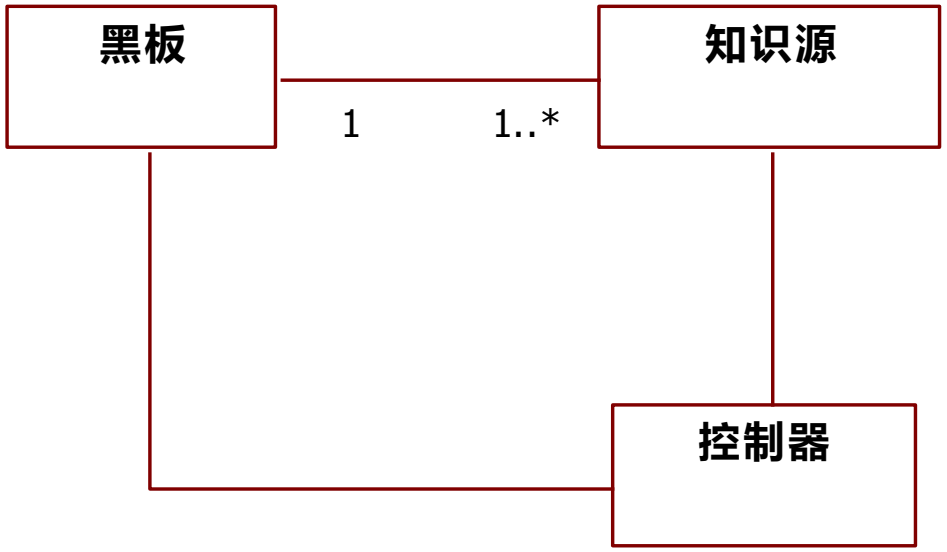
微内核模式-2

- 微内核：封装了系统提供的基本服务，是系统的最小服务子集，客户可以通过一组应用接口来访问内核服务
- 可使用微核模式的系统特点
  - 应用领域广
  - 生命周期长
  - 开发团队复杂多变

微内核模式-3

- 内部服务：和系统功能紧密相关的可选服务，将系统的可分离功能分配到内部服务组件中
- 外部服务：也是一组可选的服务，依照不同功能集合分组，提供不同的服务接口，可以由不同的团队进行开发
- 应用程序接口(API)：应用程序接口集成了来自于微核、内部服务和外部服务组件的接口
- 客户：客户代表一个应用，它通过应用程序接口使用微核、内部服务和外部服务提供的服务

黑板模式-1



知识源：问题求解模块，关注于解决问题的某些特定方面。
黑板：数据仓库，存储了协作问题求解的知识和控制数据。
控制器：负责监视黑板上的知识变化情况，根据知识应用策略协调知识源进行求解。裁决是继续求解还是终止求解。

黑板模式-2

- 使用场合
  - 问题的解空间很大，因此在有限时间内很难进行解空间的完全搜索
  - 领域尚不成熟，开发者仅有关于如何求解的零碎知识
  - 存在多个不同的求解方法，每种方法都有其优势和缺陷
  - 求解的某些步骤中存在不确定性，求解过程很难顺序化，也可能会基于其它步骤的求解结果继续求解
  - 设计者希望降低算法耦合，从而可以在后续开发中更新或添加算法模块

内容提要

- 软件设计概述
- 模型和视图
- 质量属性
- 软件体系结构
- 设计评审

设计评审和质量保证

- 通过尽早发现问题降低开发成本
- 通过评审过程改善沟通
- 通过改善设计质量改善项目计划

设计评审的常见问题

- 冗长低效的讨论过程
- 缺少建设性成果
- 没有达成沟通目标
- 缺乏时间、资源和精力投入

积极设计评审ADR

对比下列2组问题

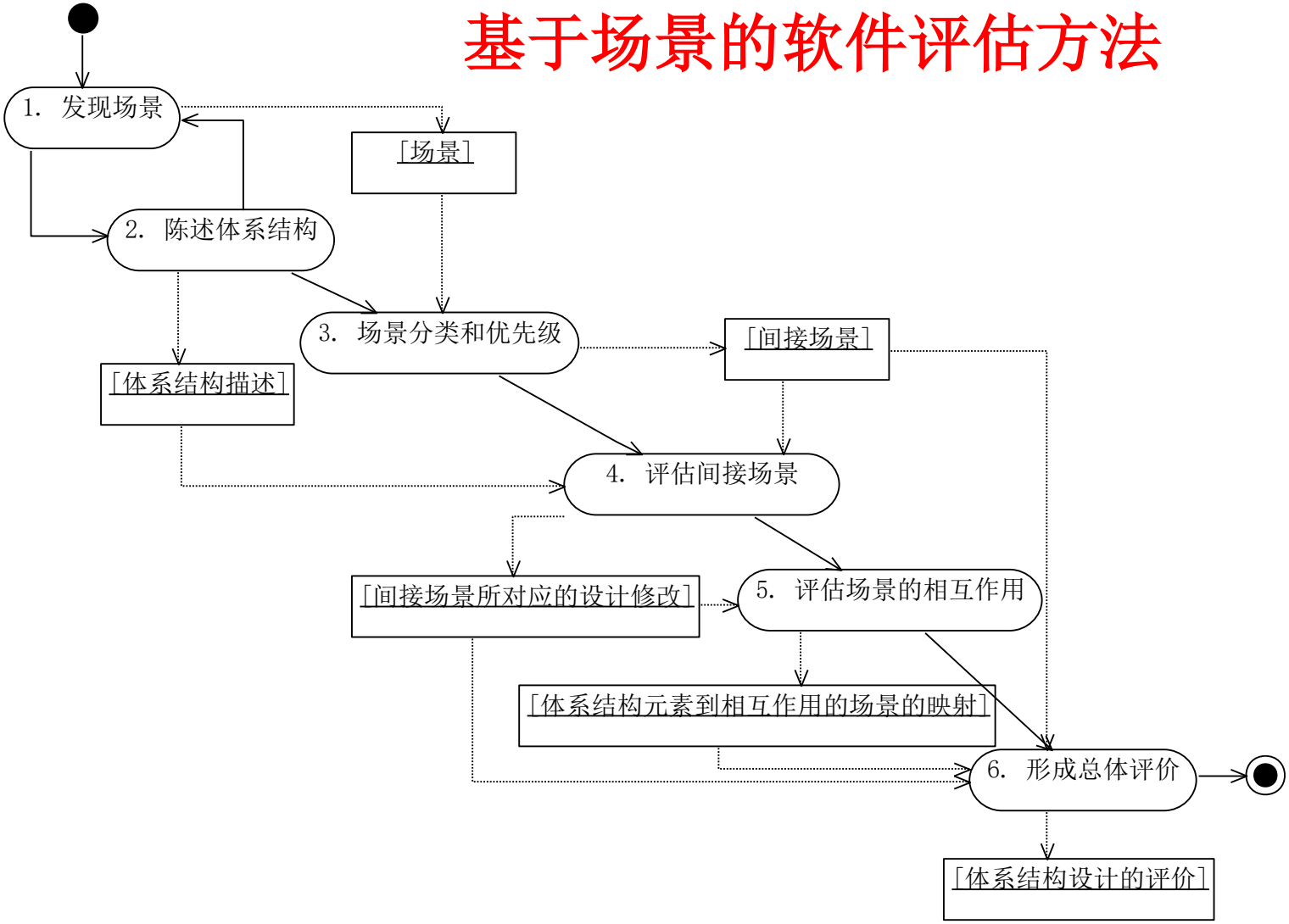
问题1	问题2
所有方法都定义了Exception吗？	请为每个方法写下可能发生的异常
每个方法定义的异常都正确吗？	请写下每个参数的有效值范围，并写下方法被非法调用时的状态
设计合理吗？	请根据本设计为程序编写伪代码
程序的性能合乎要求吗?	计算各场景运行的最大执行时间

体系结构评估方法

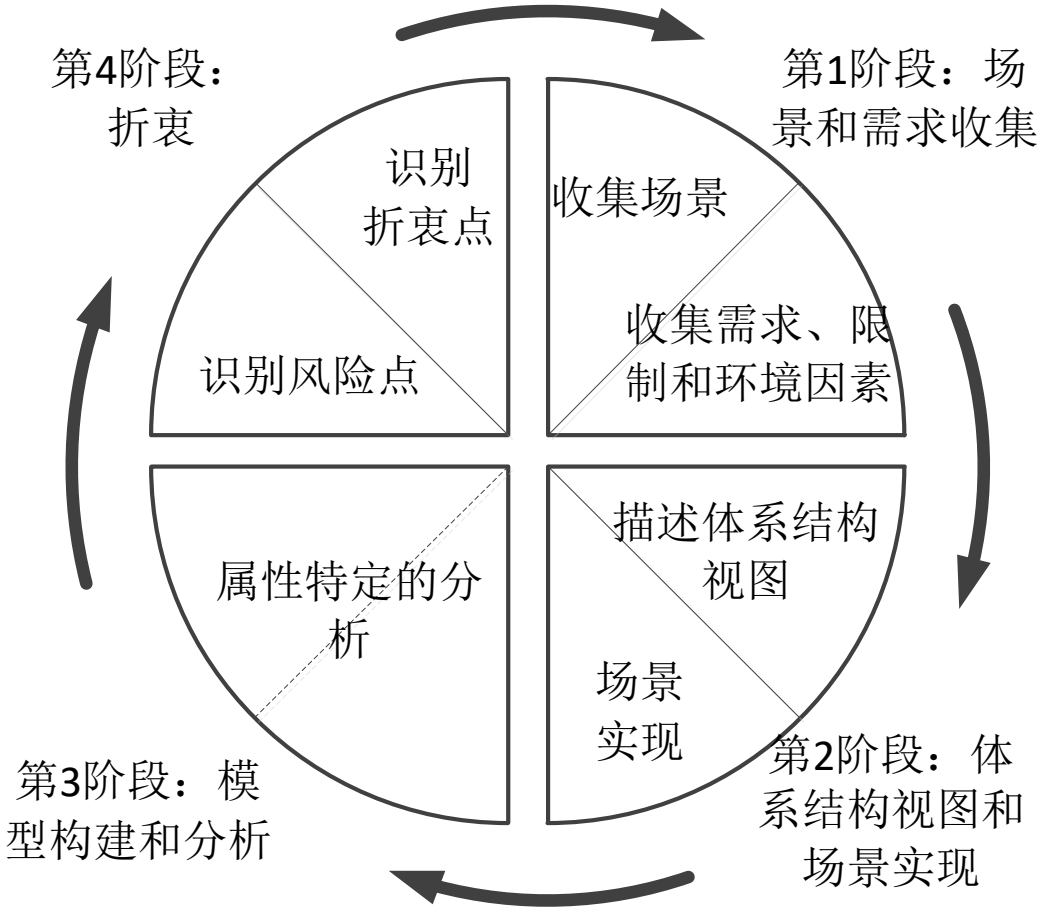
- **SAAM: Software Architecture Analysis Method**
- **ATAM: Architecture Tradeoff Analysis Method**
- **ARID: Active Reviews for Intermediate Design**

SAAM

基于场景的软件评估方法



ATAM



基于SAAM的扩展，更加强调质量属性在软件体系结构评估中的作用

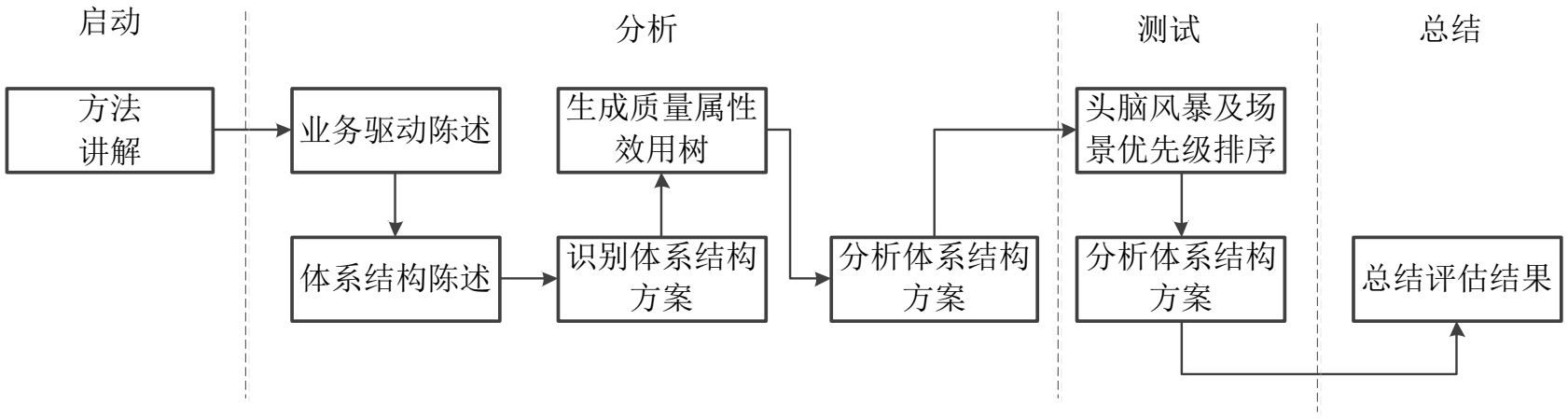
ATAM中四类重要的设计决策-1

- **风险**
  - 关注体系结构设计中尚未作出决策的重要决定（例如数据库类型）
  - 或是虽然设计团队已经作出了决定，但是并没有对该决定的后果建立完全理解的情况
- **非风险**
  - 已经做出的正确设计决策，但是该决策依赖于某些隐含的假设

ATAM中四类重要的设计决策-2

- **敏感点**
  - 描述了体系结构设计和某个质量属性之间的相关性
  - 安全性对加密算法的位数敏感
  - 消息处理的延迟对于涉及该消息的最低优先级进程敏感
- **折衷点**
  - 提示了多个质量属性之间的耦合性
  - 例如，通过允许系统在运行时动态载入插件可以提高可用性，但同时也提高了设计复杂度

ATAM


启动
分析
测试
总结
方法
讲解
业务驱动陈述
体系结构陈述
生成质量属性
效用树
识别体系结构
方案
分析体系结构
方案
头脑风暴及场
景优先级排序
分析体系结构
方案
总结评估结果

ARID

阶段	活动
第一阶段：准备	识别评审人员
	准备设计陈述
	准备种子场景
	准备评审会议
第二阶段：评审	介绍ARID方法
	陈述设计
	场景头脑风暴和优先级（关键）
	执行评审（关键）
	总结

ARID和ATAM的对比

ARID	ATAM
针对中间制品	针对完整设计
以开发人员为主体	涉及到的人员更广泛
强调积极设计评审，例如对关键方案进行实现	

高级软件工程

软件设计

The End