OOA

纲要

- OOA概述
- 标示实体—领域模型
- 职责分配的一般性原则
- 分析类和用例实现
- **架构分析**

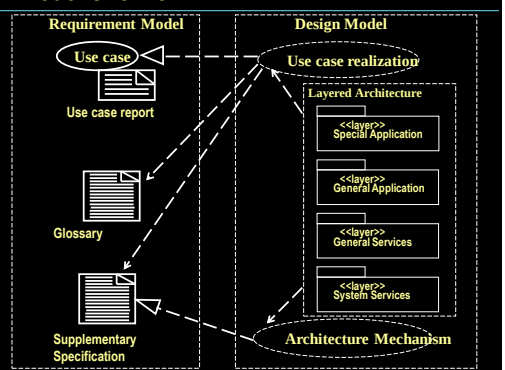
架构分析

- 架构分析关注在一定的功能需求的上下文环境里系统的非功能需求的识别的解决方案。
- 在UP里，架构分析包括架构调研和架构设计

架构分析的目的

- 基于经验定义系统的候选架构
- 定义架构模式，关键机制和模型要素
- 定义重用策略
- 为规划计划提供输入，作为决策依据

Design Model Overview



架构分析的一般性步骤

- 标识和分析影响系统架构的**非功能性需求**
- 对于对系统架构有重要影响的非功能性需求，确定解决方案和分析备选方案

非性能对系统架构的影响

- 响应时间：一个系统要求在网络负载正常情况下，界面的平均响应时间小于1.5秒，在线查询和提交操作的处理时间小于2秒。为了满足这一性能需求，架构师可能需要设计**优先级队列、增加计算资源、减少计算开销、引入并发机制、采用资源调度等策略。**
- **吞吐量**：如系统需要支持大量的用户请求和并发用户浏览功能，架构师需要设计能够**扩展和伸缩**的架构，以确保系统在高负载下仍能稳定运行。

- 能够扩展和伸缩的软件架构：
  - ✓ **分层架构（Layered Architecture）**：
    - 描述：将软件系统分为若干层次，每个层次都有特定的功能和职责。分层架构可以提高系统的可维护性和可扩展性，因为每个层次可以独立开发、测试、维护和扩展。
    - 优点：结构清晰，便于管理和维护；各层之间可以独立开发，降低了系统的复杂性。
  - ✓ **微服务架构**：
    - 描述：将应用程序划分为一组小的、松散耦合的服务，每个服务实现特定的业务功能，并通过轻量级通信机制（如HTTP RESTful API）相互协作。
    - 优点：高度可扩展性和灵活性；服务之间独立部署和扩展，降低了系统的风险；便于技术栈的多样化和持续集成/持续部署（CI/CD）。
  - ✓ **服务器架构（Client-Server Architecture）**：
    - 描述：分为客户端（请求服务）和服务器（提供服务）两部分，二者通过网络进行通信。
    - 优点：实现了分布式计算，提高了系统的性能和可靠性；客户端和服务器可以独立开发和扩展。

- 能够扩展和伸缩的软件架构：
  - ✓ **对等网络架构（Peer-to-Peer Architecture, P2P）**：
    - 描述：每个节点既是客户端又是服务器，节点之间直接进行资源分享和通信，没有中央服务器。
    - 优点：实现了分布式计算和资源共享；提高了系统的可扩展性和可靠性；降低了对中央服务器的依赖。
  - ✓ **容器化架构（Containerization Architecture）**：
    - 描述：通过使用容器技术（如Docker）来实现应用程序的部署和管理。
    - 优点：提供了隔离和资源管理的功能；简化了系统的部署和运维工作；提高了系统的可扩展性和可移植性。
  - ✓ **无服务器架构（Serverless Architecture）**：
    - 描述：开发者编写的代码运行在无状态的计算容器中，由云服务提供商动态管理机器资源。
    - 优点：降低了基础设施的维护成本；提高了系统的可扩展性和灵活性；便于快速开发和部署小型服务。
  - ✓ **分布式架构（Distributed Architecture）**：
    - 描述：系统的不同部分分布在网络上的多个节点上，这些节点协同工作，对外表现为一个统一的系统。
    - 优点：提供了高可用性、可扩展性和容错能力；能够处理大量数据和高并发请求。

- **持续服务能力**：系统需要保证7x24小时持续提供服务，且在出现故障时能够迅速恢复。为了满足这一需求，架构师可能会采用冗余技术、负载均衡等策略，以及**设计双机热备或双机互备环境，以确保系统的高可用性和容错能力。**
- **故障恢复时间**：例如，系统要求在网络失效或主机断电的情况下，3分钟内恢复正常服务。这要求架构师在设计时考虑**故障切换和恢复机制，以及备份和恢复策略。**

- **安全性需求影响架构设计**
  - ✓ **访问控制**：系统需要实现严格的权限访问控制，确保用户只能访问其权限范围内的数据和操作。这要求架构师在设计时考虑身份认证、授权管理、安全审计等功能，以及采用加密技术保护数据传输和存储的安全性。
  - ✓ **抵抗恶意攻击**：系统需要能够经受来自互联网的一般性恶意攻击，如病毒、口令猜测、黑客入侵等。为了满足这一需求，架构师可能需要设计防火墙、入侵检测系统、安全漏洞扫描等安全措施。

- **可维护性需求影响架构设计**
  - ✓ **代码可维护性**：架构师在设计时需要考虑代码的可读性、可理解性和可测试性，以便在后续的开发和维护过程中能够轻松地修改和扩展系统。例如，通过采用抽象、信息隐藏、限制通信路径等设计原则，以及使用单元测试、回归测试等测试策略来提高代码的可维护性。
  - ✓ **系统变更能力**：系统需要能够适应不断变化的业务环境，快速实现业务的新增和修改。为了满足这一需求，架构师可能会采用服务化设计、模块化设计等策略，将系统拆分成若干独立的服务或模块，降低系统耦合度，提高系统的可修改性和可扩展性。