软件体系结构

公式

体系架构=组件+连接件+约束

SoftwareArchitecture=Components+Connectors+Constrains

风格决定因素

组件类型（例如：数据容器，过程，对象）

连接件类型/交互机制（例如：过程调用，事件，管道）

组件的拓扑分布

拓扑和行为的约束（例如：数据容器不能自己改变数据，管道不能是循环的）

风格的代价和益处（优缺点）

异质的风格 Heterogeneous style）： 一个系统是由不止一种风格构建的

几种软件体系结构风格

数据流 Data Flow:

实例：流水改卷

两种方法：

方式1：一位老师改完1份卷子，就传给下一位老师

方式2：一位老师改完整班卷子，再传给下一位老师

特点

由数据控制计算

系统结构由数据在处理之间的有序移动决定

数据流系统的结构是比较明显的

在纯数据流系统中，处理之间除了数据交换，没有任何其他的交互

风格

组件：

数据处理的步骤

组件的接口是输入端口还是输出端口

计算模型：从输入中读取数据，计算，然后写到出口

连接件：

数据

单向，通常是异步有缓冲的

系统：

任意的拓扑结构

不同组件完成不同的功能

模式：

我们主要研究近似**线性数据流**或者是在限度内的循环数据流

如果一个软件系统的数据流的流向无序很可能说明该系统不应采用数据流的体系结构

例子：

批处理

- 每个处理步骤是一个独立的程序
- 每一步必须在前一步结束后才能开始（有次序）
- 数据必须是完整的，以整体的方式传递
- 批处理可以做，管道过滤器做不了：对数据的整体访问，因为管道过滤器的数据分布在不同的组件上

管道过滤器

特性：

每个组件都有一组输入和输出，组件读取输入的数据流，经过内部处理，产生输出数据流。

这个过程通常通过对输入流的变换及**增量**计算来完成。

这里的组件称为过滤器，连接件像是对输入流传输的管道，将一个过滤器的输出传到另一个过滤器的输入。

- 管道过滤器的通用结构：
  - 管道：限制了系统的拓扑结构，只能是过滤器的线性序列
  - 有界管道：限制了在管道中能够容纳的数据量
  - 类型定义管道：要求定义在两个过滤器间传出的数据类型
- 过滤器的角色：
  - 读取数据流，输出处理后的数据流
- **执行流式的转换**
  - **递增地转换数据，数据边到来边处理，不是先收集好，再处理**
  - 不同过滤器之间是独立的
- 管道的角色
  - 移动数据，从一个过滤器的输出到另一个过滤器的输入
- 全部的操作
  - 数据传送引起系统动作
  - 当没有数据可用，没有更多的计算的时候，管道过滤器系统停止工作
- 读取与处理数据流的方式

    - 递增地读取和消费数据流
    - 在输入被完全处理之前，输出便产生了
- 优点
    - 使软件具有良好的**隐蔽性**和**高内聚，低耦合**的特点（过滤器可以看做是黑盒）
    - 可将整个系统的I/O特性，理解为各个过滤器**功能的简单合成。**（多个低级过滤器可以合并成一个高级过滤器）
    - 支持功能模块的**重用**（任意两个过滤器只要在相互所传输的数据格式上达成一致，就可以连接在一起）
    - 系统**易于维护和扩展**（新的过滤器容易加入到系统中，旧的过滤器也可以被改进的过滤器替换）
    - 支持某些特性属性的分析（入吞吐量和死锁检测）
    - 支持多个过滤器的并发执行（适合多核/多线程环境）

      理想情况下是组件之间只有数据依赖
- 缺点
    - 不适合在交互性很强的应用
    - 在数据传输上没有通用的标准，每个过滤器都增加了解析数据的工作
    - 处理两个独立但相关的数据流可能会遇到困难
    - 某些过滤器可能需要大尺寸的cache
- 批处理和管道过滤器这两种数据流风格，无法从拓扑结构上进行区分
- 比较：
    - 相同点：
        - 都是把任务分解为一系列固定顺序的计算单元（组件）
    - 不同点：
        -

批处理	**管道过滤器**
全部	增量
高延迟	立即有结果
输入随机访问	输入的处理是局部的
没有并发性	可能有反馈回路
没有交互性	有交互性

    - 怎么选择：
        - 任务是数据主导
        - 事先知道数据的确切流向
    - 考法：
        - 选择这个结构为什么可以满足需求，为什么这个结构的缺点是可容忍的

过程控制

- 适用场合：软硬件结合的系统，以硬件为主
- 开环控制
    - 系统无反馈
    - 由人进行反馈

- 闭环控制
  - 系统有反馈，不需要人进行反馈
  - 两种形式：
    - 反馈控制：根据受控变量的测量值来调整过程
    - 前馈控制：通过测量其他过程变量，来预计输入变量对被控变量将产生的影响。前馈控制这些变量来调整过程
- 当软件系统的运行受到外部干扰（软件不可见或不可控的力量或事件）的影响时，需要为软件体系结构考虑的一种过程控制方式。
- 和许多线性的数据流体系结构不同，控制环路体系结构需要有循环的拓扑结构
- **在什么情况下选择控制这种风格：**
  - 任务包含连续的动作、行为、状态的改变
  - 不适合人参与的情况
  - 一般是软硬件结合的系统（适用场合）

call/return 调用/返回 体系结构风格：

主程序/子程序

OO

分层：

特点

- 每层为上一层提供服务，使用下一层的服务，只能见到与自己邻接的层
- 大的问题分解为若干渐进的小问题，逐步解决，隐藏了很多复杂度
- 修改一层，最多影响两层，通常只会影响上层。若层之间接口稳固，则不会造成其他影响
- 上层必须知道下层的身份，不能调整层次之间的顺序
- 层层相调，影响性能。

Client/Server Style:

两层C/S

缺点：

1. 对客户端软硬件配置要求较高
2. 客户端程序设计复杂
3. 数据安全性不好，客户端程序可以直接访问数据库服务器
4. 信息内容和形式单一
5. 用户界面风格不一，使用繁杂，不利于推广使用
6. 软件维护与升级困难，每个客户机上的软件都需要维护

三层C/S

B/S(浏览器/服务器)

Data Centered/Shared Data:

repository 仓库

这种风格描绘很多系统

共同特点是共享数据

典型的例子：数据库、剪贴板、注册表

优点：

很容易增加数据的生产者和消费者

缺点：

种类：

1. 设计者预先定义好　　编译器
2. 输入流的信息类型决定　数据库事物系统
3. 系统其他部分的新信息决定　Scratchboard（刮板）
4. 机会主义：计算的状态决定　Blackboard

blackboard 黑板

什么是黑板?

黑板的作用相当于“共享内存”，如同多位不同专长的专家在同一黑板上交流思想，每个专家都可以获得别的专家写在黑板上的信息，同时也可以用自己的分析去更新黑板上的信息，从而影响其他专家。

解决无确定性求解策略问题

组成

- 知识源（专家）

  包含独立的、与应用程序相关的知识，知识源之间不直接进行通讯，他们之间的交互只通过黑板来完成，一个知识源只能解决问题的一部分

  - 提供解决问题的知识
  - 变现为过程、规则、逻辑断言
  - 仅能修改黑板，无法与其他知识源交互
  - 知道何时能发挥作用
  - 低耦合的子任务，或者有特别的能力

- 黑板数据结构

  按照与应用程序相关的层次结构来组织的解决问题的数据，知识源利用黑板的接口对黑板进行读写，通过不断地改变黑板数据来解决问题

  - 保存知识源要使用的数据
  - 保存来自解空间的数据
  - 分层；同层或者不同层的对象的之间的关联
  - **与仓库的区别**：
    - 黑板：黑板的状态触发进一步的操作
    - 仓库：操作的执行次序是预先确定的

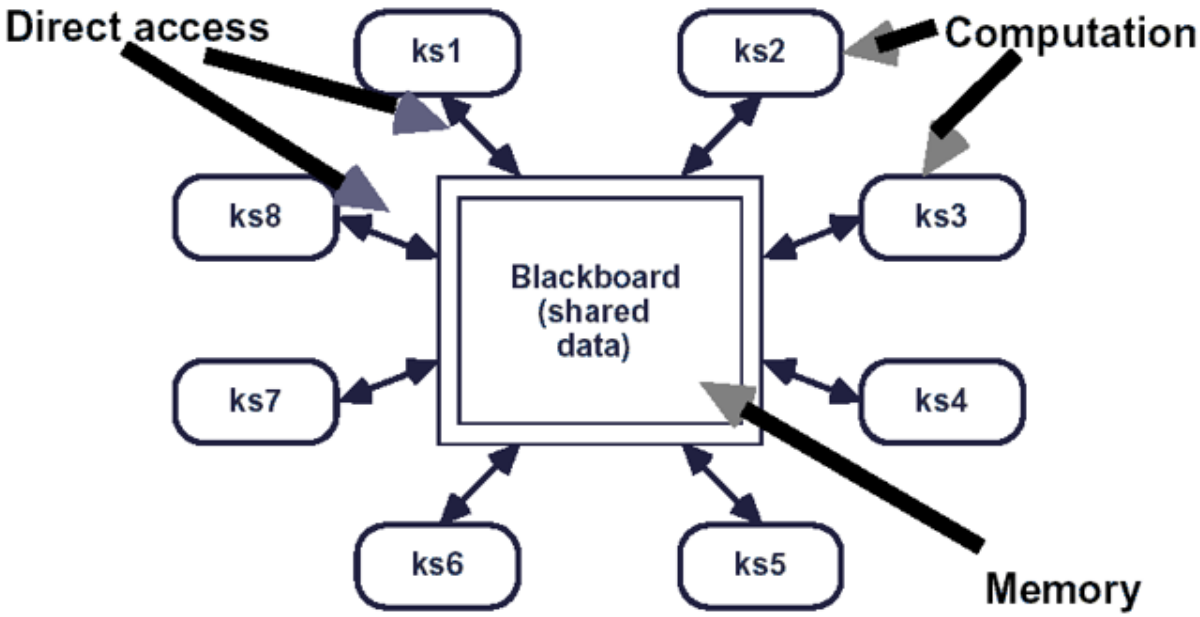
- 控制（仲裁者）

  控制完全由黑板的状态驱动，监控黑板的变化，决定下一步选哪个知识源进行工作

  - 让知识源响应偶然事件

- 连接各个知识源的能力，决策解决问题的步骤
- 控制机制是与时俱进、随机应变的

黑板的结构图：



黑板模型：

- Knowledge Sources
  - 把问题分成几个部分，每个部分独立计算
  - 对黑板的变化做出反应
- Blackboard Data Structure
  - 全局数据库包含解决方案的全部状态
  - 知识源互相关联的唯一媒介
- Control
  - 完全由黑板的状态驱动，黑板的状态的改变决定使用的特定知识
  - Knowledge sources respond "opportunistically"，让知识源响应偶然事件

Blackboard Problem Characteristics

- 没有直接的算法可解
  - 多种方法都可能解决问题
  - 需要多个领域的专门知识协作解决
- 不确定性 uncertainty
  - 数据和解决方法可能错误或变化
  - 数据中信噪比的变化
  - 算法接口的变化
- 问题没有唯一个解答，或者"正确"答案会变化

Virtual Machine

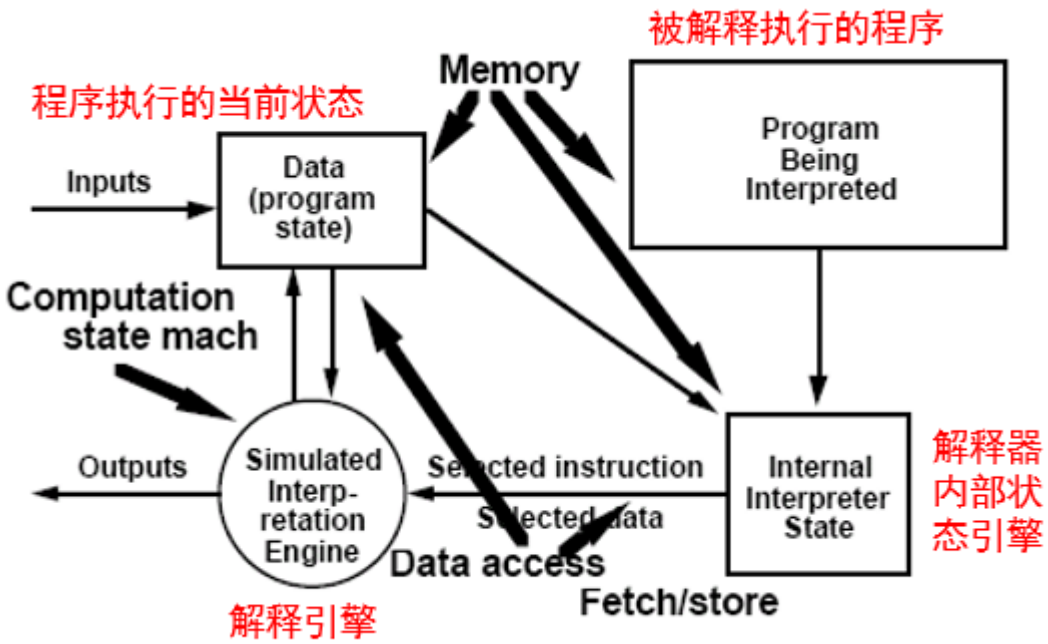
什么是虚拟机

- 一种软件
- 创建了虚拟的环境
- 屏蔽了底层平台
- 分类：
  - 系统级（硬件虚拟机）：将一台电脑虚拟为运行不同os的电脑
  - 进程级（应用程序虚拟机）：JVM
  - 机器耦合：云计算

分类

解释器（Interpreters）

结构图：



组件：一个状态机（解释引擎）、三个存储区

连接件：过程调用/对存储区的数据访问

优点：

- 功能性 Functionality
  - 可以仿真非本地的功能
- 测试性 Testing
  - 能够仿真灾难的模式
- 灵活性 Flexibility
  - 多用途的工具

缺点：

- 效率 Efficiency

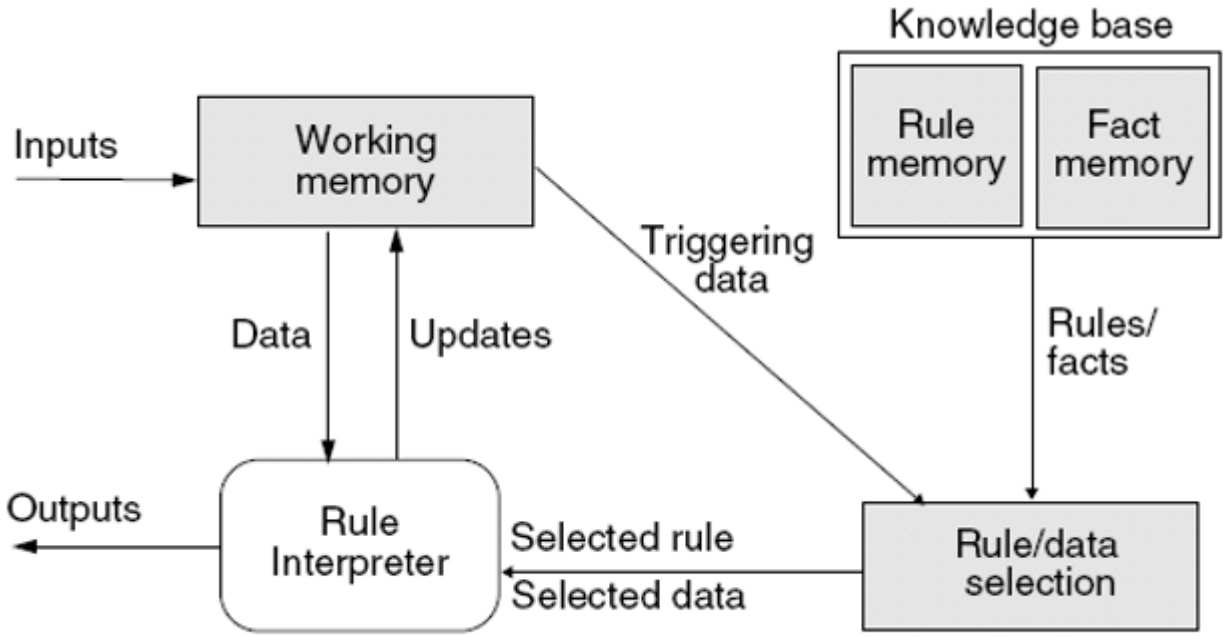
    - 比硬件效率慢的多
    - 比编译系统慢
- 测试性 Testing
    - 增加了软件要去验证的层数

解释器和编译器的区别：

- 解释器的执行速度慢于编译器产生的目标代码的执行速度，但低于“编译+链接+执行”的总时间
- 每次解释执行时，都需要分析程序结构，而编译器只需要一次性编译代码

规则系统 Rule-based system

结构：



解决的问题：

- 业务规则很复杂时，并且规则总是发生变化，不宜用if-else结构
- 按照OCP（开放/封闭原则），应把可变部分与不变部分进行分离，在前者变化时不影响后者‘

基于规则的系统：一种特殊的虚拟机，使用模式匹配搜索来寻找规则，并在正确时机应用正确的规则

特点：

- 1引擎
    - rule interpreter
- 3存储区
    - knowledge base
    - rule/data selection
    - working memory

优点：

- 降低修改业务逻辑的成本与风险
- 缩短开发时间

- 规则可在多个应用中共享

与解释器风格的不同：

- 解释器：在高级程序语言与OS/硬件平台间建立虚拟机
- 基于规则的系统：在自然语言/XML规则和高级程序语言建立虚拟机

常见的规则：

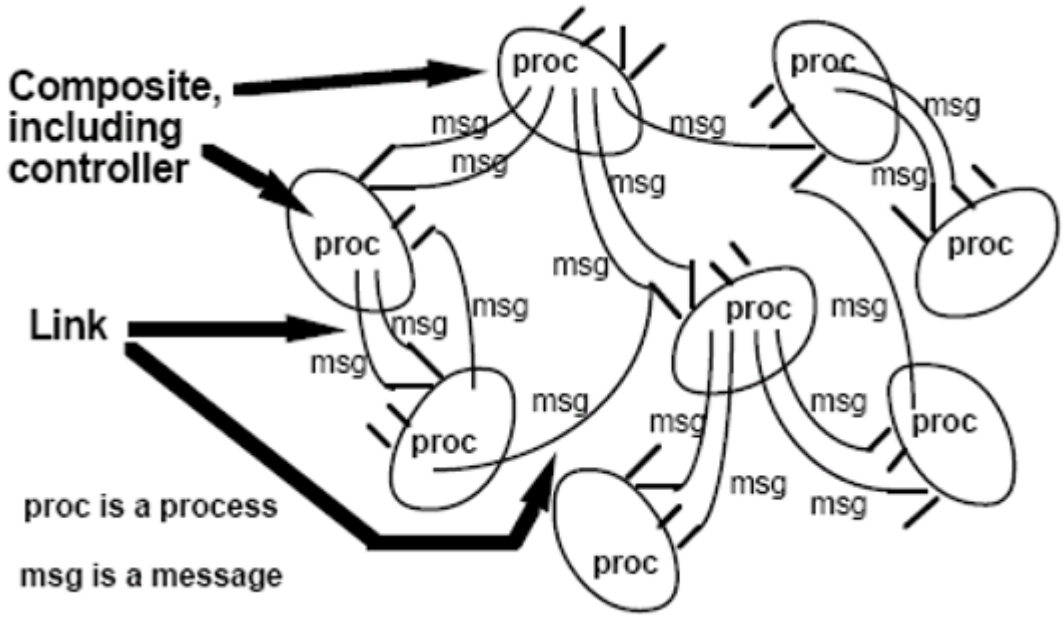
- 用户界面输入的合法性检查
- 安全规则/权限控制规则
- 业务策略（如VIP折扣策略等）

独立构件（Independent Component）

分类：

Communicating Processes

结构：



完成任务需要多个proc协同，proc间的协同通过msg完成，msg是**显性**的，即需要指明源和目的地。

组件间相对独立，依靠发消息通信。

模型：

- 解决方案
  - 系统模型：独立的通信的(communicating)进程
  - 组件：收发消息的过程
  - 连接件：消息
  - 控制结构：每一个进程有自己的控制线程
- 重要的变量
  - 通信网络的拓扑结构

    - 每条消息接收者的数量

Event Systems

> 组件对事件进行订阅，并在事件被触发时得到通知。而事件并不知道都有谁订阅了自己。

隐式调用：

- 解决方案：
    - 系统模型： 独立的反应过程
    - 组件：一个进程，不知道接收者信号的重要的事件
    - 连接件：自动地调用
    - 控制结构：分散的；没有意识到接受者信息的独立的构件
- 基于事件的隐式调用风格的思想：
    - 构件不直接调用一个过程，而是触发一个或者多个事件
    - 其他构件中的过程注册一个或多个事件，当一个事件被触发，系统自动调用在这个事件中注册的所有过程
    - 构件是一些模块（过程，或者事件的集合）
    - 主要特点是：事件的触发者并不知道哪些构件会被这些事件影响
        - 不能假定构件的处理顺序，甚至不知道哪些过程会被调用
        - 许多隐式调用的系统也包含显示调用作为构建交互的补充形式
- 优点：
    - 问题分解：
        - 为软件重用提供了强大的支持。当需要将一个构建加入现存系统中时，只需将它注册到系统的事件中。新的关注者不对现有的关注者构成任何影响
    - 系统维护和重用
        - 为改进系统带来了方便。当用一个构件代替另一个构件时，不会影响到其他构件的接口。
    - 性能
        - 调用可以是并行的
    - 鲁棒性
        - 一个构件的故障不会影响其它构件
    - 核心思想：系统分解为多个低耦合的模块
- 缺点：
    - 构件放弃了对系统计算的控制。一个构件触发一个事件时，不能确定其它构件是否会响应它。而且即使它知道事件注册了哪些构件的构成，它也不能保证这些过程被调用的顺序。
    - 数据交换的问题。有时数据可被一个事件传递，但另一些情况下，基于事件的系统必须依靠一个共享的仓库进行交互。在这些情况下，全局性能和资源管理便成了问题。
    - 既然过程的语义必须依赖于被触发事件的上下文约束，关于正确性的推理存在问题。

Other Familiar Styles

C2 Architecture Style

通过连接件绑定在一起的按照一组规则运行的并行构件网络。

系统组织规则：

- 系统中的构件和连接件都有一个顶部和一个底部
- 构件的顶部应连接到某连接件的底部，构件的底部则应连接到某连接件的顶部，而构件与构件之间的直接连接是不允许的
- 一个连接件可以和任意数目的其他构件和连接件连接
- 当两个连接件进行直接连接时，必须由其中一个的底部到另一个的顶部。

特点：

- 系统中的构件可实现应用需求，并能将任意复杂度的功能封装在一起
- 所有构件之间的通讯是通过以连接件为中介的异步消息交换机制来实现的
- 构件相对独立，构件之间依赖性较少。系统中不存在某些构件将在同一地址空间内执行，或某些构件共享特定控制线程之类的相关性假设。

异质的体系结构

（Heterogeneous Architectures）

组合方式

- 聚合方式
  - 组件的体系结构。例如：编译器中的文法分析组件
- 联合放式
  - 某一个组件或连接件成为两种以上体系机构联系的纽带
  - 可能是聚合方式的展开
  - 例如：视频点播系统
- 混合方式
  - 把多种体系结构的优点，混合使用。主要体现在连接件的混合使用。
  - 例如：事件驱动的CASE系统。

B/S与C/S混合软件体系结构

- 内外有别模型
- 查改有别模型

对等网（P2P）

特点：

- 整体稳定，不会因为一点的错误影响全局
- 资源共享，任务分摊
  - 我为人人，人人为我
- 每点还要承担一定量的中转负荷
- 对网络带宽占用极大
- 内容很难有效控制

质量属性：

质量属性场景：

- 刺激源

- 刺激
- 制品
- 环境
- 响应
- 响应量度

系统质量属性：U-STAMP

1. 可用性 availability
2. 可修改性 modifiability
3. 性能 performance
4. 安全性 security
5. 可测试性 testability
6. 易用性 usability

设计体系结构：

Tactics：

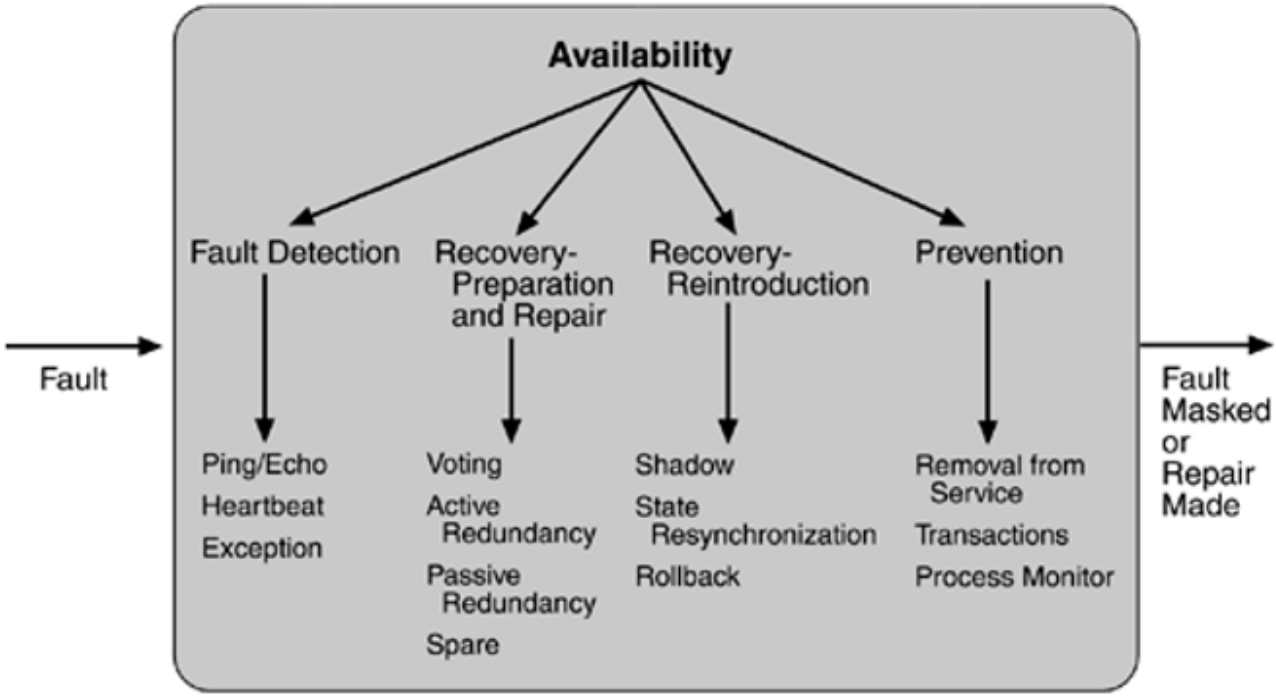
满足特定质量属性的具体设计手段

Availability tactics:

尽可能降低“故障”所造成的影响

1. 故障检测
2. 故障恢复：备份的方法
3. 错误避免：

Summary of availability tactics



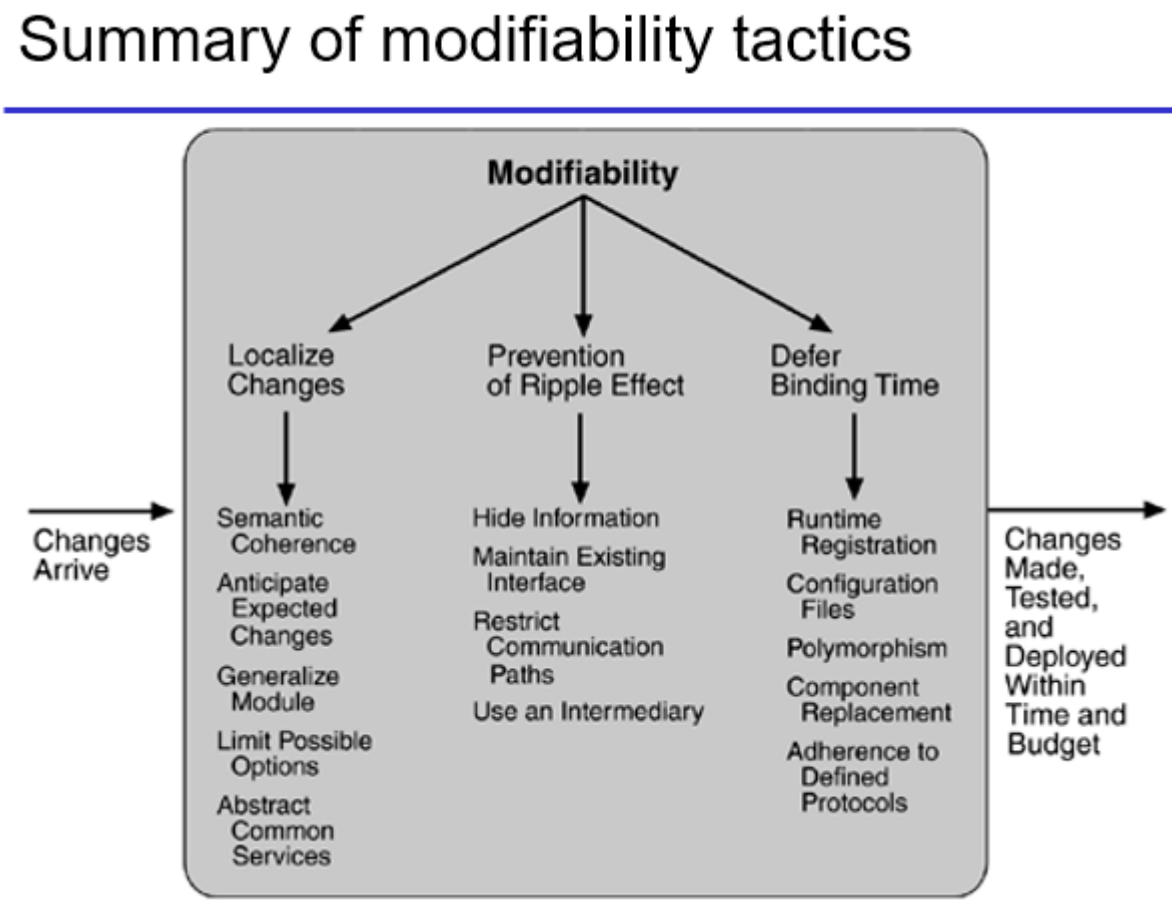
modifiability tactics:

关注的是:

- 时间和花费

1. 局部化修改
    1. 保持语义的一致性
    2. 预期期望的变更
    3. 一般化模块
    4. 限制可能的选项
2. 阻止连锁反应
    1. 隐藏信息
    2. 维护已存在的接口
    3. 限制通信路径
    4. 使用中间人
    5. location of A
        1. 每次搬家把新地址告诉父母，朋友联系不到你，也可以通过父母知道你的住址
    6. existence of A
    7. A的资源的行为/A的资源的控制
        1. 由资源管理器来调配资源
3. 延迟绑定时间
    1. 运行时注册

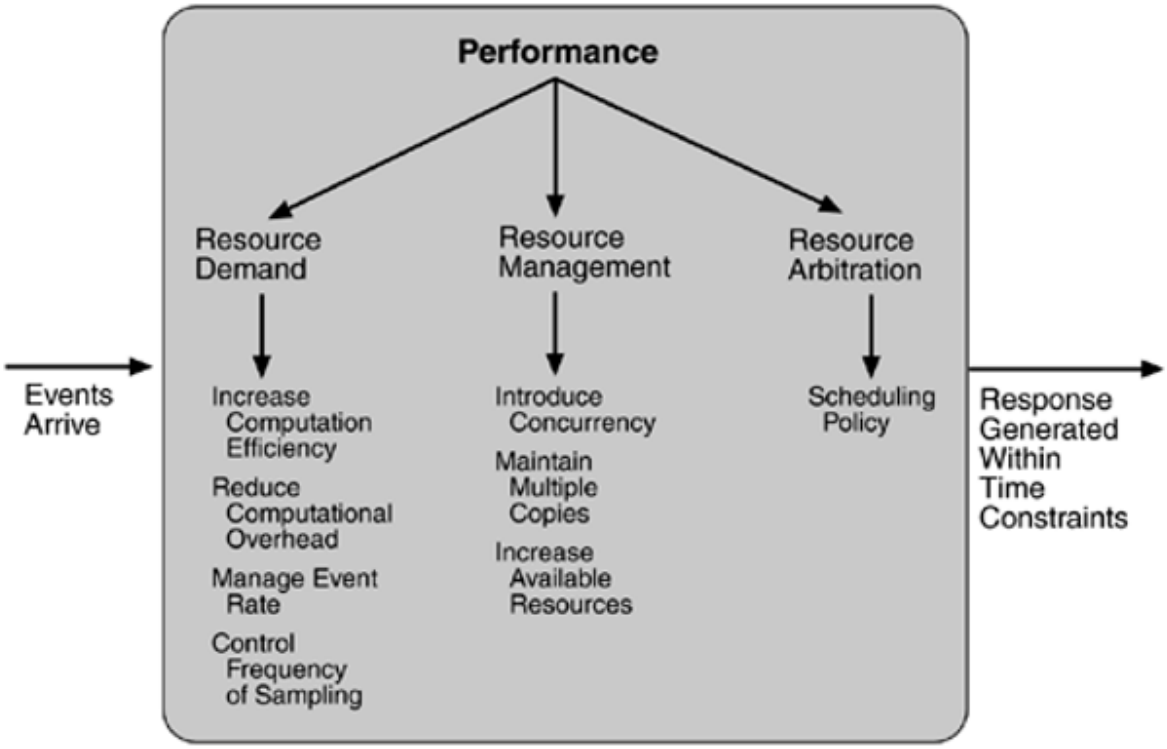
2. 发布/订阅
3. 配置文件
4. 多态
5. 构件更换
6. 定义协议



performance tactics:

1. 资源申请：干的事少一点
    1. 提高计算的效率
    2. 降低计算的开支
    3. 管理事件的速率
    4. 控制采样的频率
    5. 控制执行的时间
    6. 控制队列的等待人数的多少
2. 资源管理：牛一点
    1. 引入并发机制
    2. 提高可用的资源
3. 资源仲裁：获取资源快一点
    1. 先进/先出
    2. 固定优先级调度
    3. 动态优先级调度
    4. 静态调度（顺序不能改变）

Summary of performance tactics

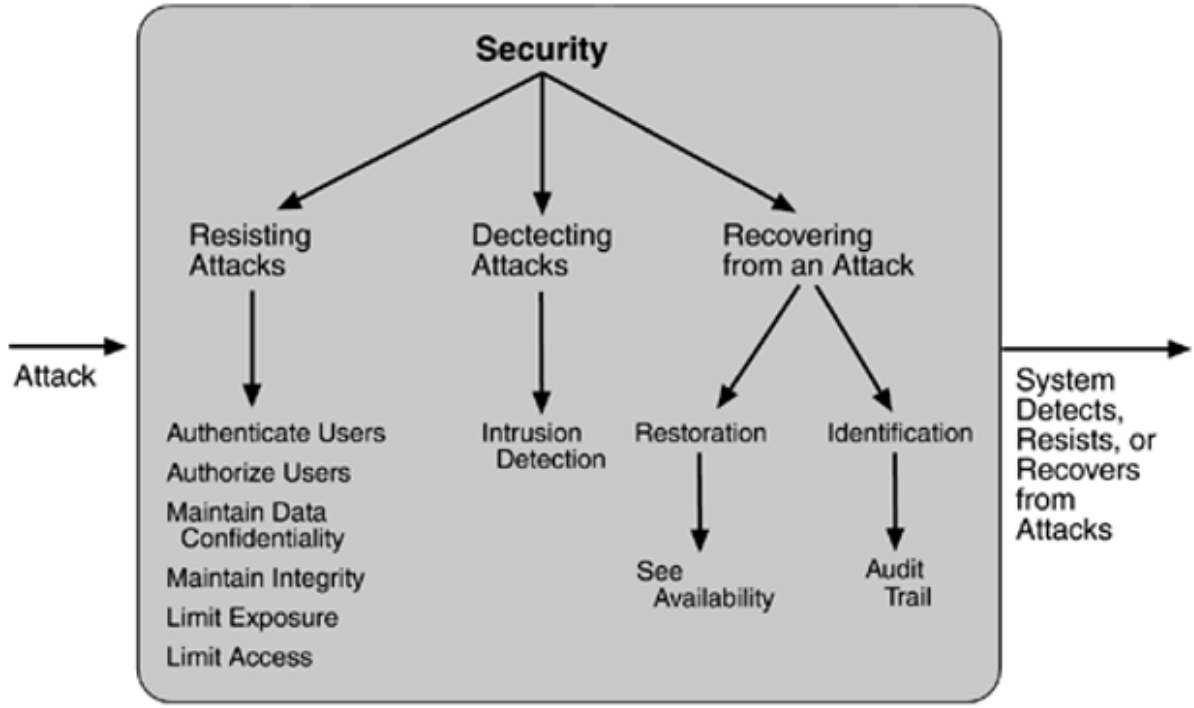


security tactics:

关注点：合法用户可以用，非法用户不能使用

1. 抵抗攻击
    1. 认证用户
    2. 对用户进行授权
    3. 维护数据的私密性
    4. 维护完整性
    5. 限制曝光
    6. 限制访问
2. 检测攻击
    1. 存储状态
    2. 攻击者的识别
3. 从攻击中恢复
    1. 审查跟踪

Summary of Tactics for Security

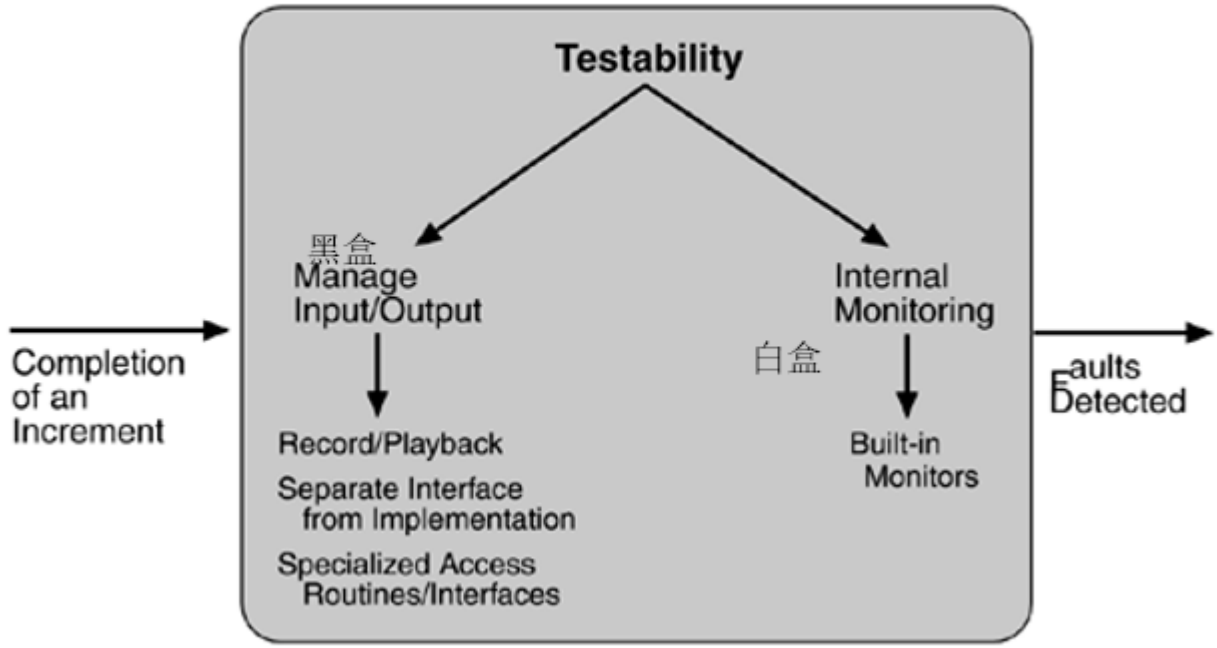


testability tactics:

关注点：测试成本，是不是容易在测试的时候，发现错误

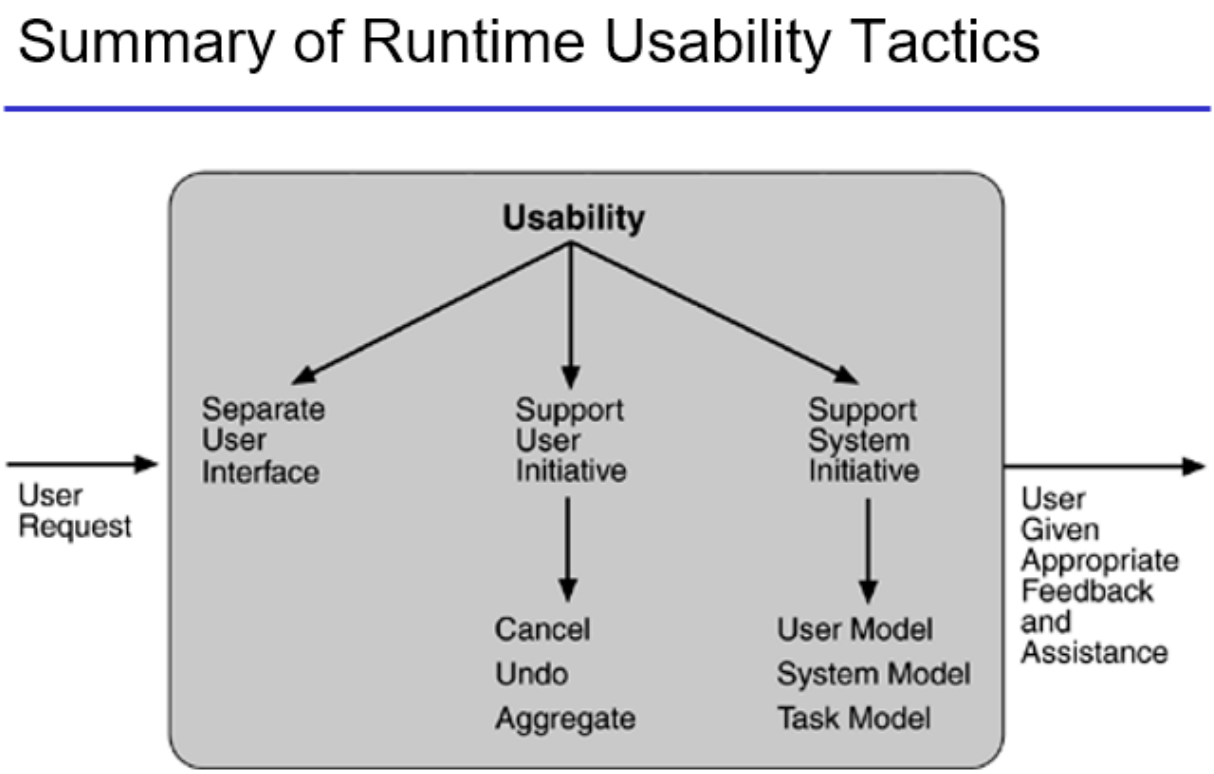
1. 黑盒
    1. 记录/重放
    2. 实现与接口分离
    3. 特殊化访问线路/接口
2. 白盒
    1. 设置断点

Summary of Testability Tactics



usability tactics：

1. 运行时
    1. 维护系统的模型
    2. 维护用户模型
    3. 维护任务的模型
2. 设计时
    1. 用户界面与其他部分分开



tactics summary:

- 从涉众处了解要达到的质量属性要求（如教务系统和银行关于安全的设置的不同）
- pattern/style和tatics分别处理高层和具体细节

软件体系结构的评估

ATAM (Architecture Trade-off Analysis Method)架构权衡分析法

目的：

- 风险：可能在将来损害某些质量属性的方案
- 非风险：可以提高质量、帮助实现目标的决策
- 关键点/敏感点(sensitivity points)：方案中一个小小的变化，就可能对某些质量属性有很大的影响
- 权衡点(tradeoffs)：影响一个以上质量的决策

步骤：

phase 0: 合伙和准备 partnership and prepartion

phase 1:初始化评估 Initial Evaluation

phase 2:比较评估 complete evaluation

phase 3:后续 follow-up

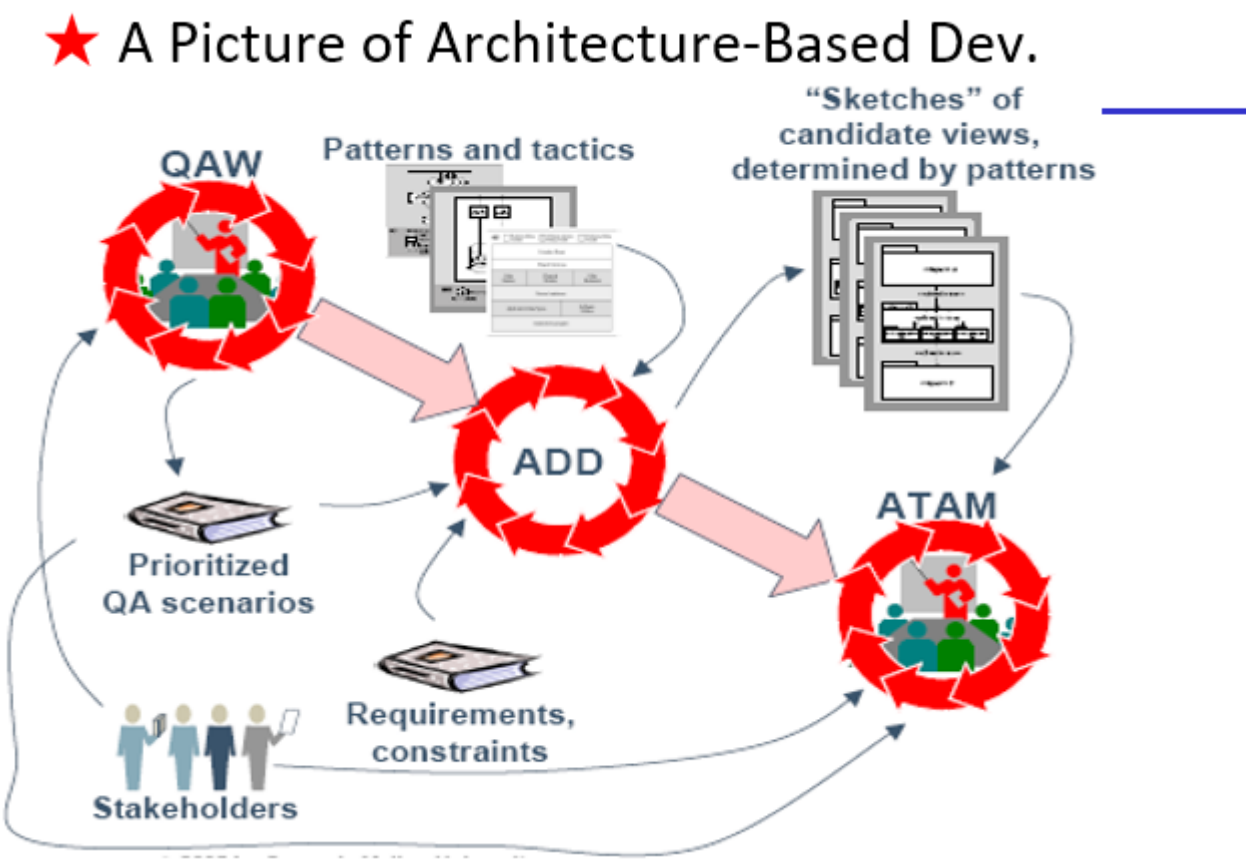
- 最重要的是画一个效用树：
  - 从两边往中间画

场景：

- 用例场景：系统的预期用途
- 成长场景：系统的预期变化
- 探索场景：系统为预料到的压力

软件体系结构的建模和文档

基于体系结构的开发的过程：



三个步骤所完成的主要任务和输入输出：

QAW（Quality Attributes Workshop质量属性会议）：

- 完成系统的质量属性，并进行优先级排序
- 输入： stakeholders的需求
- 输出：按优先级排序的质量属性

ADD（attribute-dirven design method）：

- 逐渐地有组织地为系统产生刚开始的体系结构的设计
- 输入：
  - 质量属性需求(quality attribute requirements)
  - 功能需求(functional requirements)
  - 约束(constraints)
  - 架构师所掌握的策略与风格
- 输出：

    - 初步设计的体系结构
-

ATAM(Architecture Tradeoff Analysis Method)：

- 主要任务：对产生的软件体系结构进行评估，判断其是否合理
- 输入：软件体系结构和业务需求
- 输出：绩效树、场景、风险点、非风险点、敏感点和权衡点

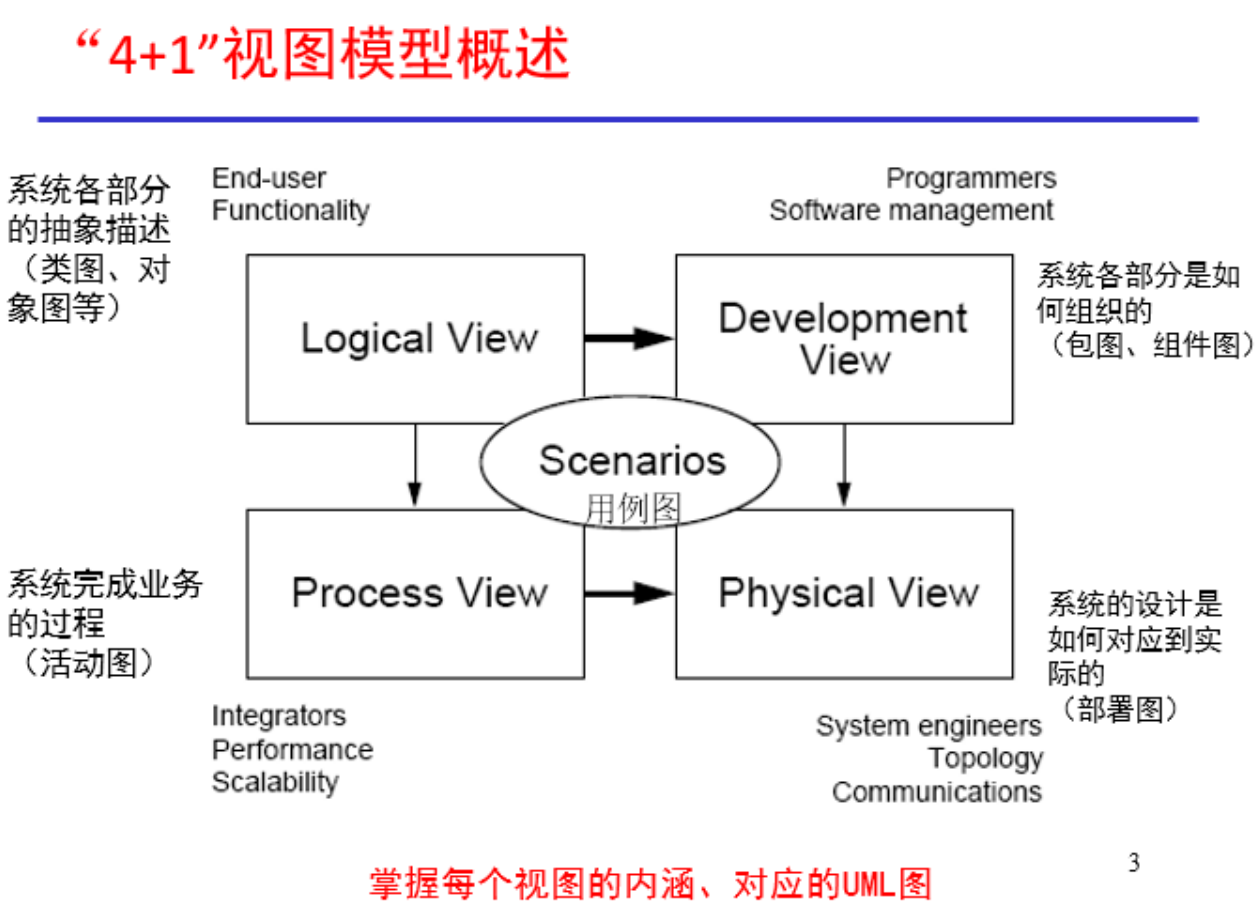
好的文档的七个原则：

1. 从读者的角度来写
2. 避免无意义的重复
3. 避免模糊
4. 使用标准的组织形式
5. 记录的合理性
6. 保持文档是当前的
7. 为了适当的目标审查文档

4+1视图模型

从5个角度来描述软件体系结构：逻辑视图、进程视图、物理视图、开发视图、场景视图

模型概述：



掌握每个视图的内涵、对应的UML图

每个模型的内涵：

- 逻辑视图：（类图、对象图、状态图、交互图）系统各部分的抽象描述
- 开发视图：（包图、组件图）系统各部分是如何组织的
- 进程视图：（活动图）系统完成业务的过程
- 物理视图：（部署图）系统的设计时如何对应到实际的
- 场景视图：（用例图）系统对外提供了什么“有价值的功能”

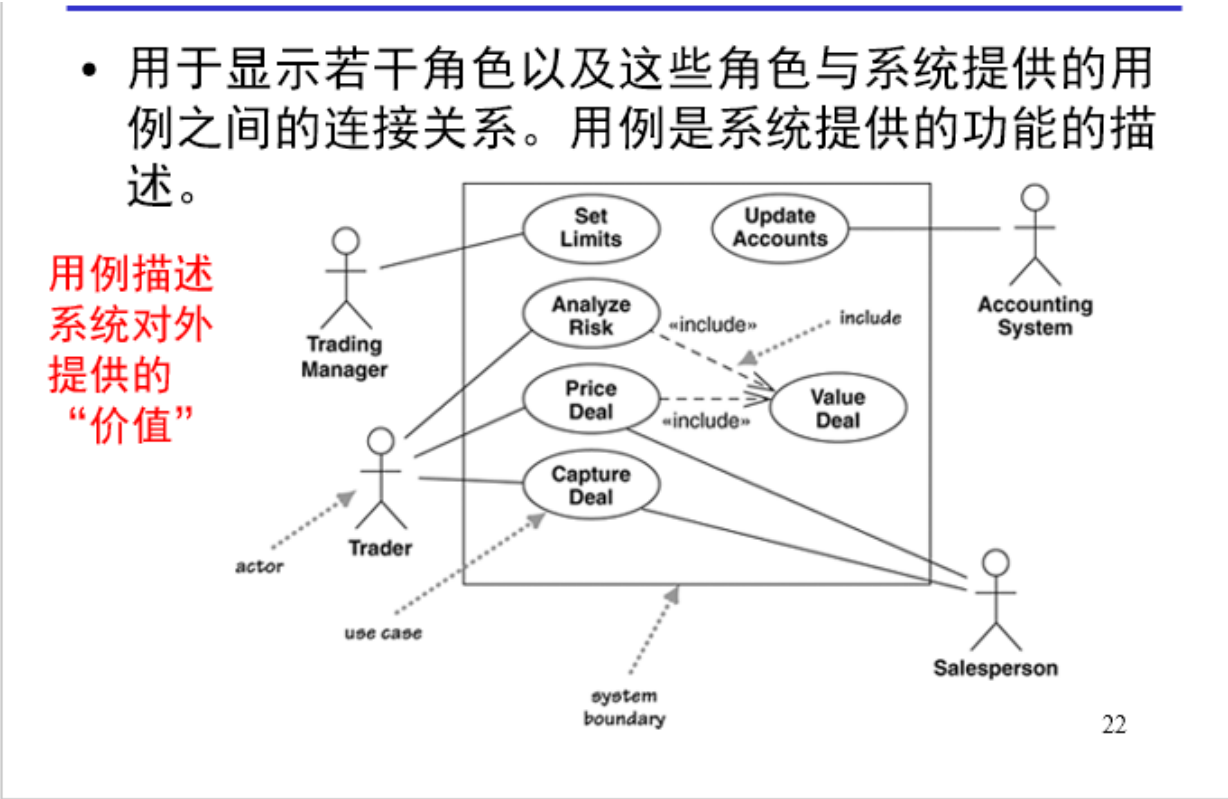
> 逻辑视图和开发视图描述系统的静态结构
>
> 进程视图和物理视图描述系统的动态结构

对于不同的软件系统来说，侧重的角度也不同

- 对于管理信息系统来说，侧重于从逻辑视图和开发视图来描述系统
- 对于实时控制系统来说，侧重于从进程视图和物理视图来描述系统

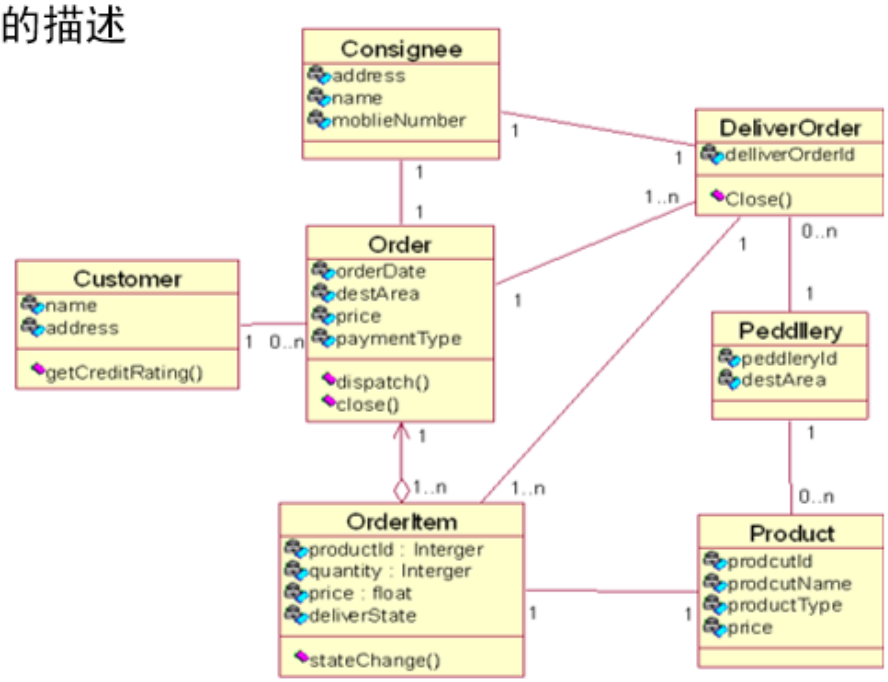
UML图：

- 用例图：



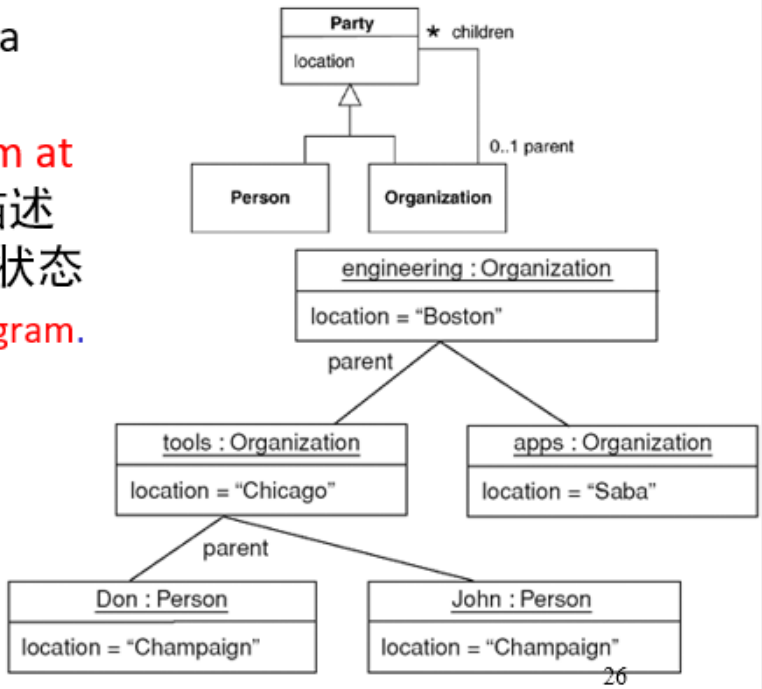
用例描述系统的“价值”
用例之间关系：包含、继承
小人：系统的使用者，代表一类人，一种身份和角色
圆圈：系统对外提供的有价值的功能
用例图是给用户看的，用来验证需求是否相同合理

  - 从外部视角来看系统，不关心系统内部（类似黑盒，只关心输入和输出）
- 类图：

- 表示系统中的类和类与类之间的关系，它是对系统静态结构的描述
  - 

- 对象图：
  - 图形像数据结构中的树

- object diagram is a snapshot of the objects in a system at a point in time. 描述系统运行时刻的状态
  - 也称instance diagram.



注意和类图的区别

- 包图：
  - 图形像一个一个文件夹

Package Diagram

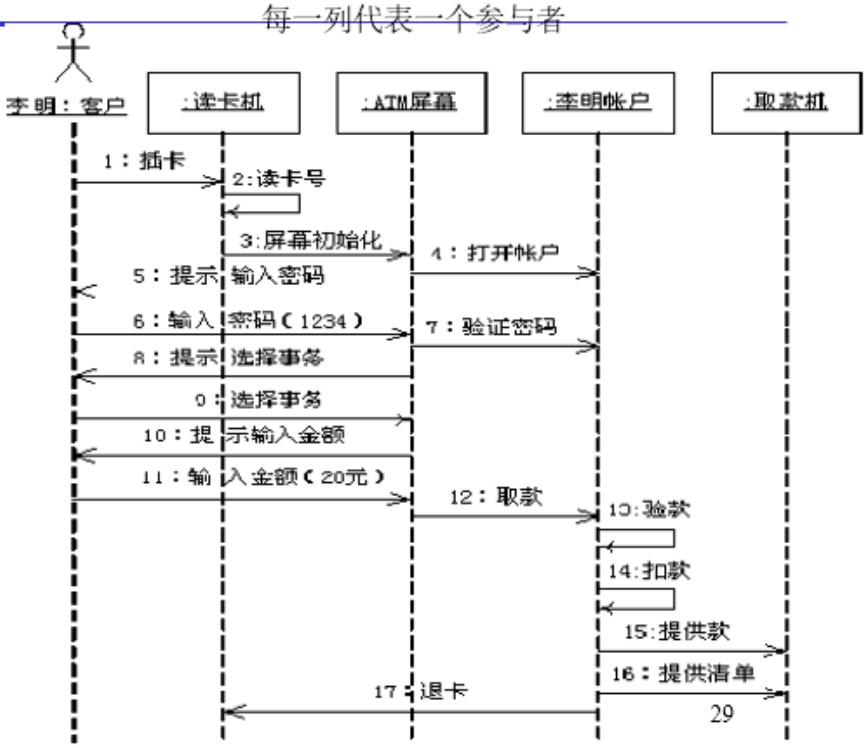
- package is a grouping construct that allows you to take any construct in the UML and group its elements together into higher-level units. you can use packages for every other bit of the UML.

- 序列图：
  - 描绘了多个对象如何协作来完成一个用例
  - 不适合精确定义对象的行为
  - 查看一个对象在多个用例之间的行为
  - 如果想查看多个用例或线程之间的行为，用活动图

Sequence Diagram 序列图

每一列代表一个参与者

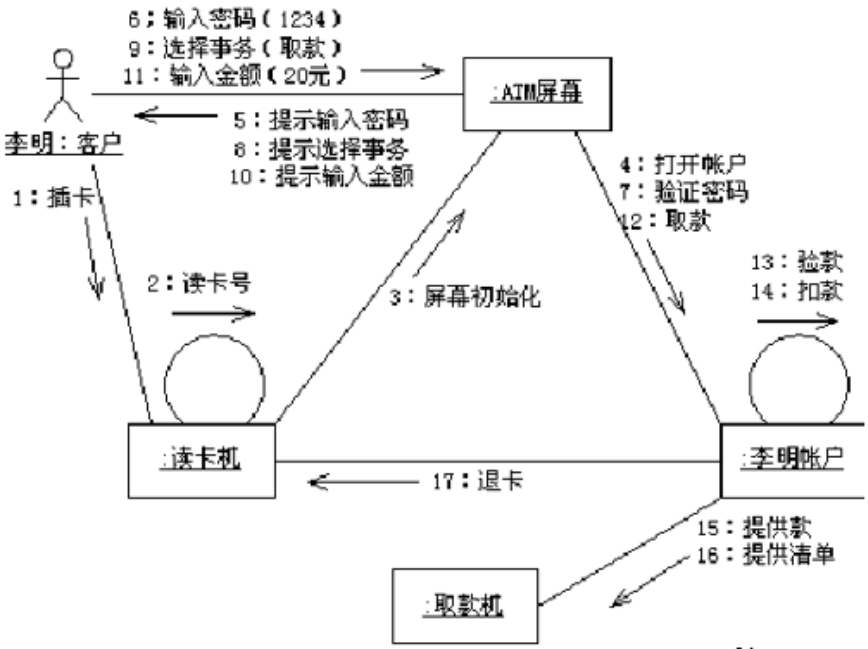
- 用来反映若干个对象之间的动态协作关系，即随着时间的推移，对象之间是如何交互的。
- 图中的箭头表示“一个类调用另一个类的方法”



- 通信图（协作图）：
  - 描述对象间的协作关系，跟序列图相似，显示对象间的动态合作关系。如果强调时间顺序，则使用序列图；如果强调联系，则选择协作图。

Communication Diagram

- 描述对象间的协作关系，协作图（通信图）跟序列图相似，显示对象间的动态合作关系。**如果强调时间和顺序，则使用序列图；如果强调联系，则选择协作图**



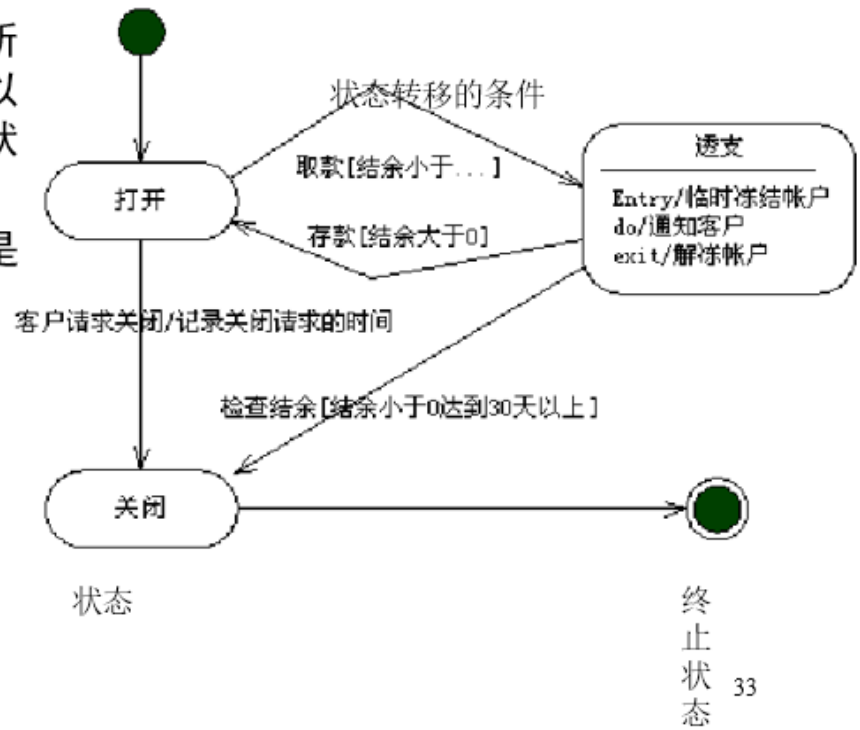
31

- 状态图：
  - 描述类的对象所有可能的状态以及事件发生时状态的转移条件。(状态图是对类图的补充)
  - > 描述的是一个类的多个状态，并且是在多个状态下的多个不同的行为。
  -

State Diagram

描述类的对象所有**可能的状态**以及事件发生时状态的**转移条件**。

通常，状态图是对类图的补充



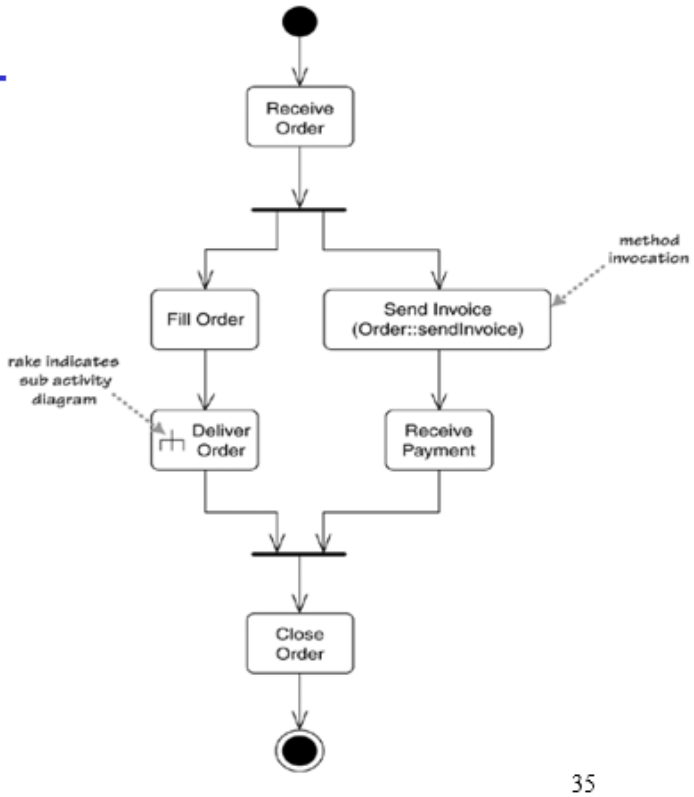
33

- 活动图：

- 描述满足用例要求所要进行的活动以及活动间的约束关系，有利于识别并行活动。
- 类似于流程图，没有选择

Activity Diagram

- 描述满足用例要求所要进行的活动以及活动间的约束关系，有利于识别并行活动

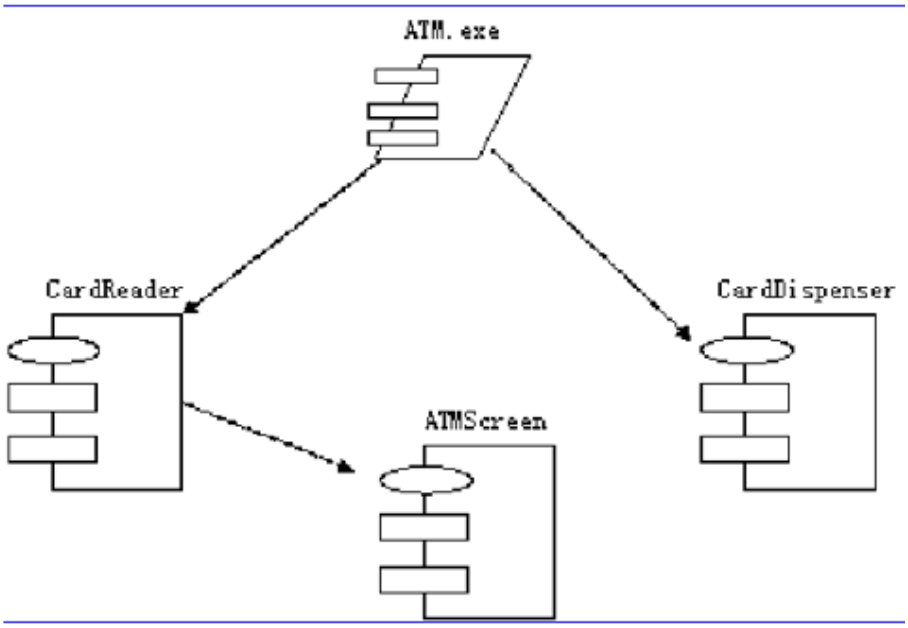


35

- 组件组：
  - 描述代码构件的物理结构之间的依赖关系

Component Diagram

- 描述代码构件的物理结构各构件之间的依赖关系

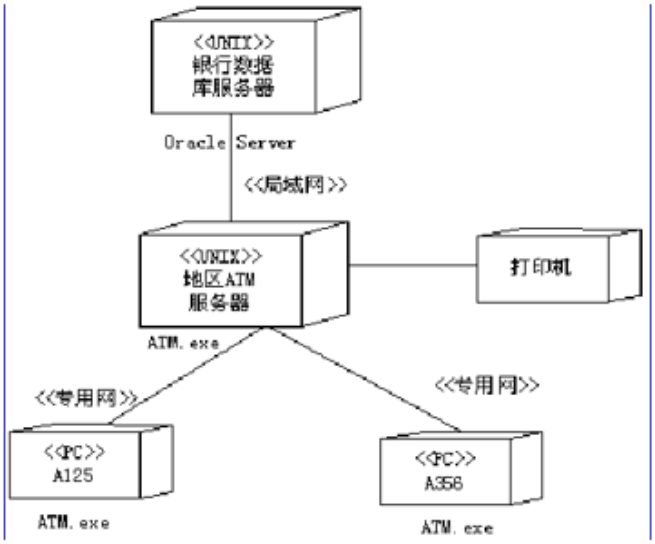


37

- 部署图（Deployment diagram）：
  - 定义系统中软、硬件的物理体系结构
  - 给部署人员使用，像网络中的拓扑结构图

Deployment Diagram

- 定义系统中软、硬件的物理体系结构



文档的一些关键问题

Some key documentation questions

- Who will use the documentation and for what purposes?
- What kind of information shall we record about an architecture?
- What languages and notations shall we use to record that information? 必考
- How shall we record and organize the information we've chosen, using the languages/notations we've chosen, to best meet the purposes we've identified?

考点：

会识别不同的UML图

要完成一件事情，需要使用什么图

10个选择： 6+2+2 策略 style UML