- **什么是软件架构?**

是系统的一个或多个结构，它们由软件元素、这些元素的外部可见属性以及这些元素之间的关系构成。

（结构，元素，关系，设计）

- **软件构架师做什么?**

联络：与客户、技术团队、业务/需求分析者、管理和市场人员

软件工程：软件工程最好的实践

技术知识：对技术领域的深入理解

风险管理：设计、技术选择方面的风险

- **体系架构来自什么?**

NFRs，ASRs，质量属性，系统涉众，开发组织，技术环境，架构师的素质和经验

  - 架构和设计的关系：架构是设计的早期阶段，没有架构，则没有后续阶段。所有的架构都是软件设计，是一系列的设计决策，高层次的设计，反之不然

- **架构视图**

Rational 统一工程 4+1

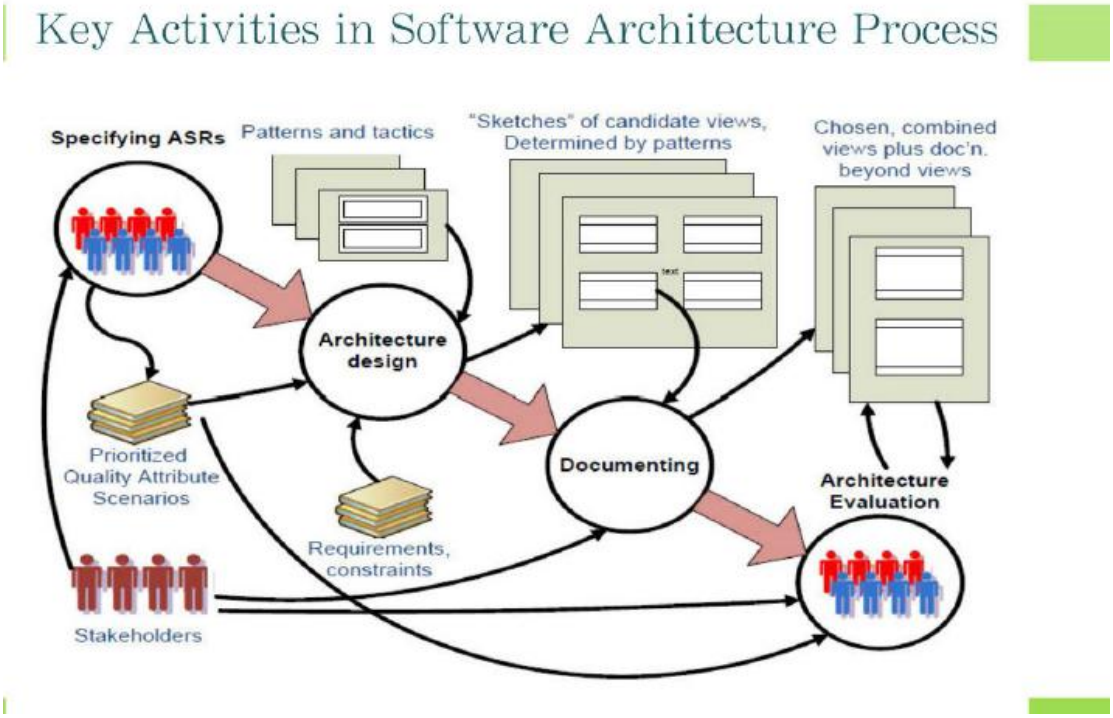
1. 逻辑视图：描述架构级别的重要元素和他们在设计开发时的逻辑关系。
   对象或对象类 模块视图 概念类图
2. 进程视图： 描述架构的并发和交流的元素，体现运行后的动态关系 ，比如数据流和控制流
   组件-连接器视图 顺序图
3. 开发视图：描述了开发环境中软件的静态组织结构，管理软件组件的内部组织（一个管理工具）
   分配视图 构件图
4. 物理视图：描述主要的进程和组件是如何与应用硬件相映射的，反应了分布式特性
   分配视图 部署图

架构用例：架构的描述可以围绕这四个视图来组织，再用一些用例或场景进行说明，形成了第五个视图

- **架构活动**

1. 为系统创建业务用例
2. 理解需求
3. 创建选择架构
4. 交流架构
5. 分析评估架构
6. 实现架构
7. 确保一致性

- **Architecture process**



关键步骤：

（1）确定 ASR:涉众识别，

（2）模式战术设计，以 ASR 作为架构设计的依据和约束条件，可以使用的工具是 patterns 和 tactics

（3）设计之后进行归档描述，生成不同的 view、view 之间的关系

（4）对架构设计进行评估，多个 stakeholder 参与，验证当前的架构是否满足了质量属性要求

Software architecture knowledge areas

软件设计基本概念

技术方面的关键问题：并发，控制，事件处理，分布，异常处理，交互式系统，持久性

软件结构和架构

软件设计的质量分析和评价

架构和设计注解

- **软件需求**

  功能需求，质量需求（NFR）,约束

- **质量属性场景**

刺激源

刺激

制品

环境

响应

响应度量

可用性

场景部分	可能的值
刺激源	系统内部、外部
刺激	错误：疏忽、崩溃、时间、响应
制品	系统的处理器、通信通道、持久性存储器、进程
环境	正常、降级模式
响应	系统检测到事件，进行以下活动之一记录故障，通知用户或系统；根据已定义的规则禁止故障源等
响应度量	系统修复时间，系统可以在降级模式下运行的时间间隔等

可修改性：

场景部分	可能的值
刺激源	开发人员、系统管理员、最终用户
刺激	希望修改功能，质量属性或系统容量
制品	系统用户界面、系统运行平台、环境或与目标系统交互的系统
环境	设计时、构建时、编译时、运行时
响应	查找构架中需要修改的位置，进行修改且不会影响其他功能，对所做的修改进行测试；部署所做的修改
响应度量	根据所影响的元素的数量的成本、资金；该修改对其他功能的影响

性能

场景部分	可能的值
刺激源	大量独立源中的一个，可能来自系统内部
刺激	定期、随机或偶然事件到达
制品	系统
环境	正常模式；超载模式
响应	处理刺激；改变服务级别
响应度量	等待时间、时间期限、吞吐量、抖动、缺失率、数据丢失

3

安全性

场景部分	可能的值
刺激源	授权或非授权用户；访问了有限的资源/大量资源
刺激	试图修改数据，访问系统服务
制品	系统服务、系统中的数据
环境	在线或离线、直接或通过防火墙入网
响应	对用户验证，阻止或允许访问数据或服务
响应度量	避开安全措施所需要的时间或资源；恢复数据/服务

可测试性

场景部分	可能的值
刺激源	单元开发人员、系统集成人员、系统验证人员、测试人员、用户
刺激	已完成的一个阶段，如分析、构架、　　类和子系统的集成，所交付的系统
制品	设计、代码段、完整的应用
环境	设计时、开发时、编译时、部署时
响应	可以控制系统执行所期望的测试
响应度量	已执行的可执行语句的百分比；最长测试链的长度，执行测试的时间，准备测试环境的时间

易用性

场景部分	可能的值
刺激源	最终用户
刺激	想要学习系统特性、有效使用系统、使错误的影响最低，适配系统
制品	系统
环境	在运行时或配置时
响应	上下文相关的帮助系统；数据和/或命令的集合，导航；撤销、取消操作，从系统故障中恢复；定制能力，国际化；显示系统状态
响应度量	任务时间，错误数量，用户满意度、用户知识的获得，成功操作的比例等

- **质量属性战术(tactics)：树形结构**

架构设计最基本的工具

从以下战术中列出四个

可用性战术

1. 错误检测　命令/响应　心跳　异常
2. 错误恢复　主动冗余(所有冗余组件都以并行的方式对事件作出相应)　被动冗余(一个组件对事件作出相应，并通知其他组件进行状态更新)　备件　检查点/回滚
3. 错误预防　从服务中删除　事务　进程监视器

可修改性战术

1. 局部性修改：维持语义一致性(语义的一致性指模块中责任之间的关系，目标是确保所有这些责任都能够协调工作，不需要过多依赖其他模块) 预期期望的变更 泛化该模块 限制可能的选择
2. 防止连锁反应：信息隐藏(把某个实体的责任分解成更小的部分，并选择哪些信息是公有的，哪些信息是私有的) 维持现有的接口 限制通信路径 仲裁者的使用
3. 推迟绑定时间：运行时注册 配置文件 多台 组件更换 遵守已定义的协议

性能战术
1、 资源需求：减少处理一个事物所需要的资源 减少所处理的事件的数量
2、 资源管理：引入并发(并行处理请求，可以减少闭锁时间) 维持多个副本 增加可用资源(选择速度更快的处理器、额外的处理器、额外的内存、速度更快的网络都可以减少等待时间)
3、 资源仲裁：调度策略

安全性战术
1. 抵抗攻击：对用户进行身份验证，对用户进行授权 限制访问
2. 检测攻击：入侵检测
3. 从攻击中恢复：查看可用性，审计追踪

可测试性战术
1. 输入/输出： 记录/回放 将接口与实现分离 特化访问路线/接口
2. 内部监视： 内置监视器

易用性战术
分离用户接口，支持用户主动 用户模型

1. 模块结构 系统被如何组织成一个代码单元集合的
2. 组件连接器结构 系统如何被组织成一个具有运行时行为和交互的元素集合的
3. 分配结构 系统如何与其环境中的非软件结构相关

- **怎么收集识别 ASR（architecture sinificant requirement）**

需求，面谈，理解商业目标，效用树

- **体系结构模式（patterns）**

实践中重复被发现的一系列设计决策
有已知的属性，允许重用，描述了一类架构

建立上下文，问题，解决方案之间的关系
由几个因素决定：一组元素类型，一组语义限制，一组交互机制

在实践中建立，没有完整的清单

- **模块模式**
  - 分层模式

定义：定义分层以及层之间单向调用的关系。
元素：层（一种模块） 层的描述应该定义层包括什么模块，以及层提供的服务的描述
关系：允许使用（依赖关系）设计应该定义层的用法和合法的异常
限制：每层软件被分配给一层；至少有两层；层的调用关系不能是双向的
缺点：层的增加会增加前期费用和系统复杂度；性能负担。

- **组件连接器模式**
  - 代理模式(去年考了这个模式，谁知道今年考哪个模式)

定义：定义一个 broker,在客户和服务器之间作为中介
元素：客户，服务器，broker,客户端代理，服务器端代理
关系：依赖关系关联了客户端，服务器端和代理
限制：客户只能和 broker(可能是客户端代理连接，服务器只能和 broker(可能是服务器端代理连接)
缺点：增加了一个间接层。客户和服务器之间有延迟，可能成为连接瓶颈。可能成为单点故障，可能成为攻击目标，可能难以测试。

  - MVC 模式

定义：把系统按功能分成三个组件：模型，视图，控制器
元素：model ,view, controller
关系：notifys 通知连接了 model，view，controller 的示例，相关状态改变通知给相关元素
限制：至少有一个实例 model 和 controller 不能直接接触
缺点：复杂度不适合简单用户接口

  - 管道过滤器模式

定义：数据从外部输入到输出要经过一系列的转换（由通过管道连接的过滤器执行）
元素：过滤器，管道
关系：attachment 将过滤器的输出和管道的输入连接起来，反之亦然
限制：管道连接过滤器的入口和出口。被连接的过滤器元素类型必须一致。
缺点：不适合交互系统。增加负担。不适合长期运行的计算。

  - 客户端-服务器模式

定义：客户端发起请求，服务器端响应返回结果
元素：客户端，服务器
关系：attachment
限制：客户端通过请求/响应连接器与服务器连接。服务器组件可以是其他服务器的客户端。
缺点：服务器可能成为性能瓶颈。服务器可能成为单点故障。复杂性，开销

  - 点对点

定义：点之间请求和提供服务来完成计算
元素：点，请求/响应连接器
限制：允许的连接数，寻找点的跳数，点之间了解哪些信息
缺点：管理安全性，数据一致性，可用性，备份和恢复更加复杂。小的点对点系统难以实现质量目标

  - SOA

定义：一系列提供和消费服务的合作组件来完成计算
元素：Components 服务提供者，服务消费者，ESB,服务注册处，编制服务器
Connectors: SOAP connector、REST connector、异步消息连接器
关系：attachment 连接不同种类的可用组件和连接器
限制：服务消费者和提供者相连，可能需要间接的组件
缺点：复杂。不控制独立服务的演化。服务可能成为性能瓶颈，通常不能保证性能。

- 发布订阅模式

定义：组件发布和订阅事件。当组件发布事件后，连接器把事件发布给已注册的订阅者。
元素：C&C 组件（有至少一个发布或订阅端口）。连接器，负责发布和监听
关系：attachment
限制：所有元素都与事件分发器相连接
缺点：增加了延迟，难以保证可拓展性和可预测性。对消息次序控制更少，消息传递难以保证

- 共享数据

定义：数据访问者的连接受到一个共享数据存储的调解。数据持久化。
元素：共享数据存储。数据访问组件。数据读写组件。
限制：数据访问者和数据存储交互
缺点：数据存储可能成为性能瓶颈，单点故障。生产者和消费者紧密耦合

- **Allocation 模式**
  - 多层

分层和多层区别?
缺点:前期投入，复杂度。

- Map-Reduce 模式（大数据处理中流行的架构） 映射规约

缺点：如果没有大的数据集合，不适合使用。如果不能把数据集合分入相似子集，也不适合。复杂性。

- **模式 vs 战术**

1. 战术比模式简单。
2. 模式把多个设计决策组合在一起。
3. 都是架构师的主要工具
4. 战术是 patterns 设计的基石
5. 大多模式包含几个不同的战术。例如分层模式包含增加内聚，降低依赖的战术

- **设计战略**

分解和迭代。生成和测试。

- ADD

输入和输出?
Input: requirements(书上写一组质量场景)
Output:
software element
role
responsibility

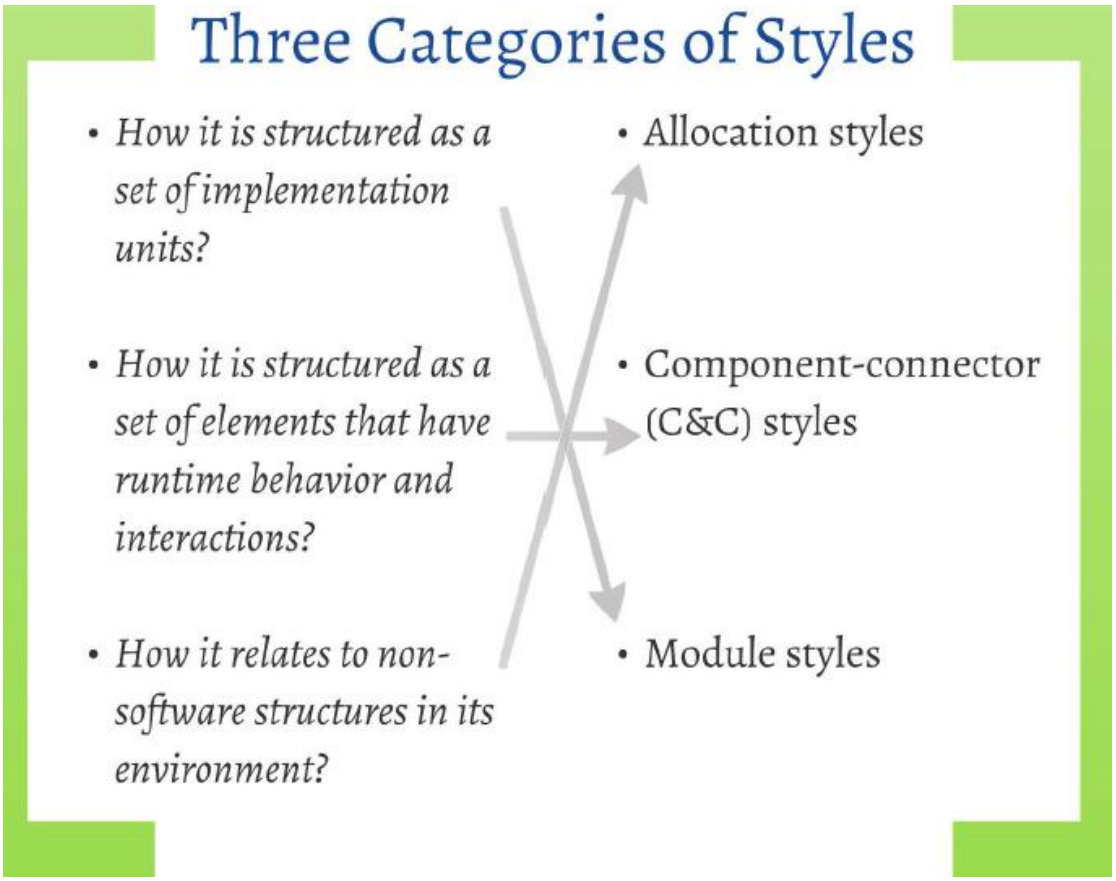
property
relationship

ADD 过程：

1. 确保有足够的需求信息
2. 选择一个系统元素做分解
3. 为选定元素识别 ASR
4. 选择一个满足 ASR 的设计
   4.1 识别设计关注
   4.2 列出可替换的模式/战术
   4.3 从列表中选择模式/战术
   4.4 决定模式/战术和 ASR 之间的关系
   4.5 获得初始架构视图
   4.6 评估设计，解决不一致
5. 初始化架构元素，分配职责
6. 为初始化的元素定义接口
7. 验证和精炼需求，使他们成为初始化的元素的约束
8. 重复直到所有的 ASR 被满足

- Styles,patterns,views
  Architecture style：元素和关系类型的专业化，以及其使用的约束
  Architecture patterns：表达了软件系统的基本组织结构
  Architecture views：视图是一系列系统元素和他们之间关系的表示

  Mapping



- **为什么要使用不同的 views**：
  不同的视图支持不同的目标和用户，突出不同的系统元素和关系
  不同的视图将不同的质量属性暴露出不同的程度
- 结构化视图：(第六题和第八题)
  模块视图：提供责任设置的实现单元
  包括：分解视图 使用视图 分层视图 数据模型视图 aspects view 等
  组件连接器视图：显示运行时的元素。例如进程、对象、客户端、服务器端、数据等等
  包括：管道-过滤器视图 客户端-服务器视图 点对点视图 SOA 视图 多层视图等
  分配视图：描述了软件单元和环境元素之间的映射
  包括：部署视图 安装视图 工作作业视图 其他部署视图
- 质量视图

- **文档化视图：**

1. 建立涉众/视图表 哪些 view 对哪些涉众来说重要
2. 合并视图 一个 view 可以体现架构多方面特征
3. 区分优先级

- **跨视图的文档**

把不同视图联系起来

- Documentation Beyond Views
  Documentation Roadmap(文档路线图)
  How a view is Documented(视图如何被归档)
  System overview(系统预览)

Mapping between views(视图之间的映射)
Rationale(合理性)
Directory---index, glossary, acronym list(目录)

- **构架评估 ATAM 构架权衡分析方法**

ATAM 一种进行构架评估的综合方法
参与人员：1、评估小组
2、项目决策者
3、构架涉众
不同阶段的输入输出
ATAM 结果
1、 一个简洁的构架表述
2、 表述清楚的业务目标
3、 用场景集合捕获的质量需求
4、 构架决策到质量需求的映射
5、 所决定的敏感点和权衡点集合
6、 有风险决策和无风险决策
7、 风险主题的集合

ATAM 四阶段
第 0 阶段（合作关系和准备）评估小组负责人和主要的项目决策者进行非正式会晤，以确定此次评估的细节。需要几周时间
第 1 阶段和第 2 阶段是评估阶段，每个人都开始认真考虑分析工作。
第 3 阶段是后续阶段，产生最终书面报告。

第 1 阶段
1、 ATAM 方法的表述
2、 商业动机的表述
3、 构架的表述
4、 对构架方法进行分类
5、 生成质量属性效用树
6、 分析构架方法(敏感点，权衡点，风险)
第 2 阶段 与 1 的涉众不同
1、 展示 ATAM 的结果
2、 集体讨论并决定场景的优先级
3、 分析构架方法
4、 结果的表述
第 3 阶段 跟进 生成评估报告

- 软件产品线

把一个产品分成两部分：core assets+ custom assets
共享 core assets
目的：实现高重用性，高可修改性
产品线之所以如此有效，是因为可以通过重用充分利用产品的共性，从而产生生产的经济性。而且可重用的范围很广。

产品线架构与普通架构的区别：

在产品线架构中有一组明确允许发生的变化，然而对于常规架构来说，只要满足了单个系统的行为和质量目标，几乎任何实例都是可以的。因此，识别允许的变化是架构责任的一部分，同时还需要提供内建的机制来实现它们。

产品线架构实现

重用：查找，理解，使用

变化：

不同的变化形式（6 种）*

How can you vary entities in a software system?

Include or omit elements 包括或忽略元素

Variable number of each element 每个元素不同个数

Interface Satisfaction 满足接口

Parameterisation 参数化

“Hook” interfaces

Reflection 反射

变化发生的位置（3 个层次） *

架构级别 设计级别 文件级别

变化发生的时间（5 个）

编码时、编译时、连接时、初始化时、运行时

变化点 variation points

SPL Practice Area and Patterns

通过把所有实践识别出来，分组

29 个 practice areas , 22 个 pattern(12 个，衍生出 10 个变组，共 22 个)

发生前，发生时，发生后改变架构

- 模型驱动开发 MDD model driven development

功能定义和实现分开

来自百度：MDA 的目的是分离业务建模与底层平台技术，以保护建模的成果不受技术变迁的影响。

1.具有平台无关性,MDA 是一个跨平台的规范,所以使用 MDA,可以做到与平台无关这样也提高了软件产品的可秀色可餐性.

2.提高了开发效率

3.使软件质量获得提升,使用模型自动生成代码可以大大减少代码的出错几率.

目标：高的可重用性，高的互操作性，可移植性

三个抽象层次：计算独立模型 CIM（定义系统需求，和周围环境关系）

平台无关模型 PIM（考虑到计算和算法）
平台相关模型 PSM（最底层）
OMG 的标准：UML, MOF, XMI, QVT（纵向的）

- SOA

基本思想：三角图 服务的提供者，消费者，注册之间的关系
SOA vs Web Service ：
SOA 是设计原则，web service 是实现技术
Soa 应用可以不使用 web service 建立
Web service 带来了依赖于平台的标准
Web service 允许更好的互操作性

SOA vs ESB ：
ESB 不实现 SOA，但为 soa 提供平台
大多数 ESB 提供者建立 ESB 来合并 SOA 规则增进销量

两种实现策略：
自下而上：标识(identify)并且描述服务的信息和操作 实现服务
自上而下：把已有的组件和程序暴露为服务 重用组件

SOA 基本原则：(来自百度，挑几个背吧)
1. 明确边界
2. 共享契约和架构，而不是类
3. 策略驱动
4.自治
5. 采用可传输的协议格式，而非 API
6.面向文档
7.松耦合
8.遵循标准
9. 独立于软件供应商
10.元数据驱动

SOA 对质量属性的影响：

	Availablity	Interoperability	Modifiability
Service-Oriented Pattern	SOA 架构中非常强调实体自我管理和恢复能力，可用性较好	通过服务之间既定的通信协议进行互操作，互操作性较好	服务之间是松耦合的，导致会话一端的软件可以在不影响另一端的情况下发生改变

	Performance	Security	Testability	Usability
Service-Orient	因为中间件的存	由于一个应用	无明显影响	无明显影响

ed Pattern	在, 性能不能得到保证，服务(service) 有可能是性能的瓶颈	软件的组件很容易与属于不同域的其他组件进行对话，所以安全性实现会比较复杂

简答题，问答题，分析设计题
基础内容（左边的） 70%
高阶内容（右边的，产品线，soa） 30%

期末考试 40%
平时 60%