#### 软件工程

###### 系统测试（评审）

* 尽早、不断的进行测试。
* 程序员避免测试自己设计的成勋。
* 你要选择有效、合理的数据，也要选择无效不合理的数据。
* 修改后应进行回归测试。
* 尚未发现的错误与该程序发现的错误数成正比。

###### 测试类型

动态测试：在计算机上实际运行程序进行软件测试，包括黑盒测试、白盒测试。

* 黑盒测试-功能测试-数据驱动。
* 白盒测试-结构测试-逻辑驱动。
* 灰盒测试-介于两者之间。

静态测试：程序不在机器上运行，采用人工检测代码质量，保罗桌前检查、代码走查、代码审查。

* 桌前检查-程序员自查。
* 代码走查-程序员与架构师共同检查。
* 代码审查-正规审查/轻量级审查（非正式）/结对编程。

###### 测试用例

驱动模块--前--被测模块--后--桩模块。

黑盒测试用例：

* 等价类划分：把输入字段内容划分不同类型，每个类型用一个代表性的输入字段即可。两个测试用例的设计原则：①设计一个新的测试用例，使其更多的覆盖有效等价类。②设计一个新的测试用例，使其覆盖一个尚未被覆盖的无效等价类。
* 边界值分析：把参数范围内的参数取边界值，测试极端参数的情况。
* 错误推测：凭经验、直觉测试。
* 因果图：由结果反推原因。

白盒测试用例（基本路径测试、循环覆盖测试、逻辑覆盖测试）：

* 语句覆盖SC：逻辑代码里面所有语句都执行一遍，覆盖层级最低。
* 判定覆盖DC：判断语句条件真假分支都覆盖一次。
* 条件覆盖CC：每个判断条件每个独立条件都要执行一遍真和假（一个判断里面可能有多个条件）。
* 条件判定组合覆盖CDC：同时满足判定覆盖和条件覆盖。
* 路径覆盖：把所有可行路径都覆盖，覆盖层级最高。

###### 测试阶段

* 单元测试：针对一个小的功能模块测试。测试依据是软件详细设计说明书。
* 集成测试：检查模块之间的问题，测试依据是软件概要设计文档。
* 系统测试：目的是验证软件是否和系统有联系，偏向于性能测试。
* 确认测试：验证软件系统的各个功能是否有问题，测试依据是需求文档。
* 内部确认测试：由软件开发内部人员测试，按SRS（需求规格说明书）进行测试。
* Alpha测试：用户在开发环境下测试。
* Beta测试：用户在实际使用环境下测试。
* 验收测试：交付前以用户为主进行的测试。
* 回归测试：针对改正的bug进行测试，测试bug是否改掉，并且测试bug是否导致了其他的bug。

###### 非功能性测试（负载测试、强度测试、容量测试）

* 恢复测试。
* 安全性测试。
* 压力测试。
* 性能测试。
* 可靠性测试。
* 可用性测试。
* 可维护性测试。
* 安装测试。

###### 测试策略

* 自底向上：从底层模块开始测试，最后完成整个系统的测试。
* 自顶向下：先测试整个系统，再往下测试单个模块。
* 混合式：既有第一种，也有第二种。缺点是测试工作量大。

###### 面向对象的测试

* 算法层(单元测试)：包括等价类划分测试、组合功能测试（基于判定表的测试）、递归函数测试、多态消息测试。
* 类层（模块测试）：包括不变式边界测试、模态类测试。
* 模板层/类数层（集成测试）：包括多态服务测试和展平测试。
* 系统层（系统测试）。

###### 测试自动化优点

效率上：

* 提高测试执行的速度。
* 提高运行效率。
* 保证测试结果的准确性。
* 连续运行测试脚本。
* 模拟现实环境下受约束的情况。

成本上：

* 所有测试活动都可以自动完成。
* 减少人力成本。
* 可以免费获得降低测试工作量。

###### 软件调试

调试是找出错误的代码和原因

软件调试方法：

* 蛮力法：主要思想是”通过计算机找错”，低效，耗时。
* 回溯法：从出错处人工沿控制流程往回追踪，直至发现出错的根源。复杂程 序由于回溯路径多，难以实施。
* 原因排除法：主要思想是演绎和归纳，用二分法实现。

软件调试与测试的区别：

* 测试的目的是找出存在的错误，而调试的目的是定位错误并修改以修正错误。
* 调试是测试之后的活动,测试和调试在目标、方法和思路上都有所不同。
* 测试从一个已知的条件开始,使用预先定义的过程，有预知的结果；调试从一个未 知的条件开始,结束的过程不可预计。
* 测试过程可以事先设计,进度可以事先确定;调试不能描述过程或持续时间。

###### 验证与确认

* 验证是指在软件开发周期中的一个给定阶段的产品是否达到在上一阶段确立的需求过程。
* 确认是指在软件开发过程结束时对软件进行评价以确定它是否和软件需求相一致的过程。
* 测试是指通过执行程序来有意识的发现程序中的设计错误和编码错误的过程，测试是验证和确认的手段之一。

###### 软件度量

软件的两种属性：

* 外部属性：面向用户的属性，一般是性能指标。可直接测量。
* 内部属性：软件本身的属性，如可靠性等。只能间接测量。

McCabe度量法：又称为环路复杂度，假设有向图中有向边数为m,节点数为n,计算有向图的环路复杂度为m-n+2。

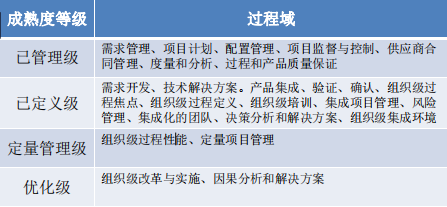
###### 软件质量

质量保证与质量控制（QC/QC）(关注过程/关注结果)（事中控制/事后检查）

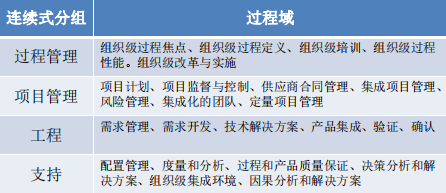
* 质量保证一般是每隔一定时间（例如每个阶段末）进行的，主要通过系统的质量审计和过程分析来保证项目的质量。
* 质量控制是实时监控项目的具体结果，以判断他们是否符合相关质量标准，制定有效方案，以消除产生质量问题的原因。
* 一定时间内质量控制的结果也是质量保证的是质量审计对象，质量保证的成果又可以指导下一阶段的质量工作，包括质量控制和质量改进。

###### 软件过程改进（CMMI）

* 阶段式--组织能力成熟度。



* 连续式--软件过程能力。



###### 配置管理

* IEEE对配置下的定义为硬件、软件或二者兼有的集合，为配置管理指定的，在配置管理过程中作为一个单独的实体对待，可作为配置项管理的有：外部交付的软件产品和数据、指定的内部软件工作产品和数据、指定的用于创建或支持软件产品的支持工具、供方/供应商提供的软件和客户提供的设备软件。
* 典型配置项，包括项目计划书、需求文档、设计文档、源代码、可执行代码、测试用例、运行软件需要的各种数据，他们经评审和检查通过后进入软件配置管理。
* 开发库（动态库，程序员库，工作库；动态系统，开发者系统，开发系统，工作空间）。
* 受控库（主库、系统库；主系统、受控系统）。
* 产品库（备份库、静态库、软件仓库；静态系统）。

###### 软件开发环境与工具

软件开发支持环境、软件生存期支持环境。

环境机制：

* 环境信息库。
* 过程控制和消息服务。
* 用户界面规范。

工具集：

* 系统规划工具。
* 建模工具。
* 分析设计工具。
* 编程工具。
* 测试工具。
* 项目管理工具。

###### 系统转换计划

遗留系统：计算修改和演化也不能满足新的业务需求的信息系统。

特点：业务不能满足需求，技术落后，大型软件系统维护工作困难，很难管、开发、理解。

新旧系统的转换策略：

* 直接转换策略-成本低，风险高-小系统。
* 并行转换策略-成本高，风险低-一般系统。
* 分段转换策略-区域：试点-推广；大型系统：子系统-逐步转换。

数据转换与迁移：旧数据库-抽取-转换-装载-新数据库（ETL）三种方式：

* 系统切换前通过工具迁移。
* 系统切换前采用手工录入。
* 系统切换后通过新系统生成。

###### 系统维护

软件维护是生命周期的个完整部分，可以将软件维护定义为需要提供软件支持的全部活动，这些活动包括在交付前完成的活动，以及交付后完成的活动。交付前完成的活动包括交付后运行的计划和维护计划等;交付后的活动包括软件更改,培训、帮助 资料等。

系统运行和维护包括：硬件维护、软件维护、数据维护。

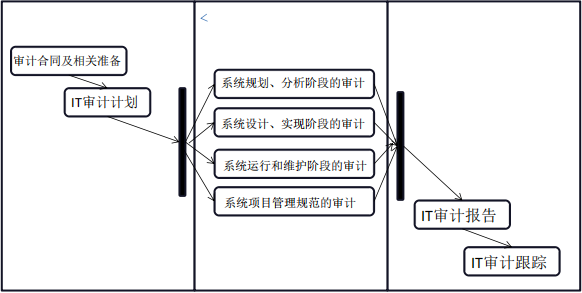
系统的可维护性：为维护人员理解，改正，改动和改进这个软件的难易程度。

系统评价指标：易分析性、易改变行、易测试性、稳定性。

软件维护类型：

* 改正性维护（25%）：正确性，发现bug而修改。
* 适应性维护（20%）：由于外部环境，被动惊醒软件的修改和升级。
* 完善性维护（50%）：基于用户主动对软件提出更多需求。
* 预防性维护（5%）预防未来可能会发生的bug。

###### 系统审计



###### 系统评价

系统性能评价、系统效益评价、系统建设评价。

系统评价流程：

* 确定评价对象，下达评价通知书，组织成立评价工作组合专家咨询组。
* 拟定评价工作方案，收集基础资料。
* 评价工作组实施评价，征求专家意见和反馈给企业，撰写评价报告。
* 评价工作组将评价报告报送专家咨询组复核，向委托人送达评价报告和选 择公布评价结果建立评价项目档案。

系统评价分类：

* 立项评价：系统开发前的评价，分析是否立项开发，做可行性分析。
* 中期评价：项目开发中的阶段评审。
* 结项评价：系统投入使用后，了解系统是否达到预期要求。

#### 软件架构设计

###### 软件架构的概念

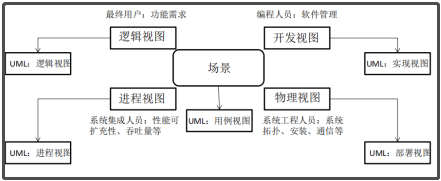
需求分析到软件设计之间的过渡过程称为软件架构。架构设计就是需求分配，将满足需求的职责分配到组件上。

* 软件架构为软件系统提供一个结构，行为和属性的高级抽象。由构建的描述，构建的相互作用（连接件），指导构建集成的模式以及这些模式的约束组成。
* 软件架构不仅制定了系统的组织结构和拓扑结构，并且显示了系统需求和构建之间的对应关系，提供了一些设计决策的基本原理。
* 解决好软件的复用，质量和维护问题，是研究软件架构的根本目的。
* 软件架构设计包括提出架构模型，产生架构设计和进行设计评审等活动，是一个迭代的过程。
* 架构设计主要关注软件组件的结构，属性和交互作用，并通过多种视图全面描述特定系统的架构。
* 软件架构是项目干系人进行交流的手段，明确了对系统实现的约束条件，决定了开发和维护组织的组织结构，制约着系统的质量属性。
* 软件架构使推理和控制的更改更加简单，有助于循序渐进的原型设计，可以作为培训的基础。是可传递和可复用的模型，通过研究软件架构可以预测软件的质量。

###### 软件架构的发展史

* 无架构阶段-汇编语言。
* 萌芽阶段-程序结构设计。
* 初级阶段-统一建模语言。
* 高级阶段-4+1视图。

4+1视图



4：

逻辑视图：针对功能需求，主要描述类，对象，构件之间的关系的，主要针对用户看的。

开发视图：也称为模块视图，侧重于软件模块的组织和管理，主要针对开发人员。

进程视图：涉及到性能相关，涉及多线程，并发，主要针对系统集成人员。

物理视图：就是系统部署视图，把软件映射到硬件上，强调软件如何部署，主要针对物理实施人员。

1：

场景视图：对应UML的用例视图，描述整个软件的功能，设计不同的使用场景。

###### 软件架构建模

* 结构建模：以构件，连接件来刻画结构。反应整个系统结构细节。
* 框架建模：侧重于系统整体结构，不在意具体细节。针对具体的问题设计问题模型。
* 动态建模：对结构模型和框架模型的补充，描述系统的演化。
* 过程建模：构件系统的步骤和过程。
* 功能建模：由一组功能构件层次组成，下层向上层提供服务。

###### 软件架构风格

基本架构风格

* 数据流风格：面向数据流，从前向后执行程序，代表风格有批处理序列（按脚本顺序一件一件的执行处理），管道-过滤器（前一个命令的输出是后一个命令的输入）。
* 调用/返回风格：构件之间相互调用，显式调用。代表风格有主程序/子程序（主程序调用子程序），面向对象（对象之间相互调用方法），层次结构（相互调用形成层次关系）。
* 独立构件风格：构件之间是独立的，通过某个事件的触发异步的方式来执行。代表风格：进程通信（通过公共缓冲区来判断进程是否调用，进行数据通信），事件驱动系统。
* 虚拟机风格：自定义了一套规则供使用者使用，常用的就是JVM，使用者要基于规则来开发，这样就能支持跨平台运行。代表风格有解释器，基于规则的系统。
* 仓库风格：又称数据共享风格，以数据为中心，所有操作都基于数据中心来进行。代表风格有数据库仓库，超文本系统，黑板系统。

数据流风格

* 批处理序列：每个处理步骤都是一个完成的程序，每一步需要前一步结束后才能开始。数据必须是完整的。
* 管道-过滤器：每个构件都有一组输入和输出，前后数据流关联，过滤器就是构件，连接件就是管道。。

上面两者区别：

* 批处理的数据必须完整，要前一步完成后才能进行下一步。
* 管道-过滤器虽然也是按顺序执行，但没有限制前一步一定完成后才能进行后一步，如前一步产生了数据，不管这个数据是否完整就可以开始下一步了
* 早期编译器就是采用数据流风格。

调用/返回风格

* 主程序/子程序：单线程控制，就是构件之间相互调用。
* 面向对象：构件是对象，想调用类里面的属性和方法，需要先定义一个这个类的对象，通过对象来调用。
* 层次结构：构件组成一个层次结构，相邻的层次之间才能调用，每一层为自己的上一层提供服务。所以一般修改某一层的内容，最多影响自己的上层和下层（一般会影响到上层，因为下层是为它提供服务的）。

层次结构优点：

* 允许将一个复杂问题分解成逐步增量的问题的实现。
* 不同层次越靠近底层越抽象，越与数据相关。越靠近顶层越具体，越与业务相关。
* 每一层最多影响两层，软件复用性更好。

层次结构缺点：

* 很难找到一个合适的层次数量，不是层次越多越好，层次越多可能性能越差。
* 不是每个系统都能容易划分层次结构。

独立构件风格

* 进程通信：构件是独立的进程，连接件是消息传递。消息传递方式可以点对点，异步和同步方式，以及远程调用等。
* 事件驱动系统（隐式调用）：构件不直接调用一个过程，而是触发或广播一个或多个事件。某个事件被触发时，系统自动调用这个事件的过程。

事件驱动系统优点：为软件复用提供了强大的支持，为构件的维护和演化带来方便。

事件驱动系统缺点：构件放弃了对系统计算的控制。

虚拟机风格

* 解释器：包括完成解释工作的解释引擎，代码存储区，记录当前工作状态的数据结构和记录源代码被解释执行的进度数据结构。
* 基于规则的系统：包括规则集，规则解释器，规则/数据选择器和工作内存。智能匹配结果供人选择，和人打交道。

上面两种区别：基于规则的系统通常用于人工智能，DSS中。

仓库风格

* 数据库系统：构件有两大类，一类是中央共享数据源，保存数据。另一类是多个独立处理单元，对数据进行操作。
* 黑板系统：包括知识源，黑板和控制三部分。知识源响应黑板的变化，只修改黑板。黑板是一个全局的数据库，是知识源相互作用的唯一媒介。
* 超文本系统：构件以网状链接方式相互连接，通常应用在互联网领域。

闭环风格（过程控制风格）-有反馈-嵌入式

当软件被用来操作一个物理系统时，软件与硬件之间可以粗略地表示为一 个反馈循环，这个反馈循环通过接受一定的输入，确定一系列的输出，最 终使环境达到一个新的状态。适合于嵌入式系统，涉及连续的动作与状态。

###### 层次架构风格

* 两层C/S架构：

包含表示层（客户端）和数据层（服务器）。客户端和服务器都有处理逻辑功能。

* 三层C/S架构：

包括表示层（显示给用户看的前端，相当于APP），数据层（数据库的数据处理）和功能层（业务逻辑结构）。

* 三层B/S架构：

包括表示层（展示的前端，相当于浏览器页面），数据层和功能层。又称为0客户端架构。

C/S架构特点：

* 开发成本较高。
* 客户端程序设计复杂。
* 信息内容和形式单一。
* 用户界面风格不一。
* 软件移植困难。
* 软件维护和升级困难。
* 新技术不能轻易应用。

B/S架构特点：

* B/S架构缺乏对动态页面的支持能力， 没有集成有效地数据处理功能。
* B/S架构的安全性难以控制。
* 采用B/S架构的应用系统，在数据查询 等相应速度上，要远远低于C/S架构。
* B/S架构的数据提交一般以页面为单位， 数据的动态交互性不强，不利于OLTP应用。

混合架构风格：

* 内外有别模型：企业内部用C/S，外部人员访问用B/S。
* 查改有别模型：采用B/S查询，采用C/S修改。
* 混合架构实现困难，成本高。

富互联网应用RIA（AJAX）：弥补三层B/S架构存在的问题，RIA是一种用户接口，比用HTML实现接口更好，并且有可视化内容，本质还是网站模式，还是0客户端。例如小程序。

* RIA结合了C/S架构反应速度快、交互性强的优点，以及B/S架构传播范围广及容易传播的特性 。
* RIA简化并改进了B/S架构的用户交互 。
* 数据能够被缓存在客户端，从而可以实现一个比基于HTML的响应速度更快且 据往返于服务器的次数更少的用户界面。

###### 面向服务的架构风格（SOA）

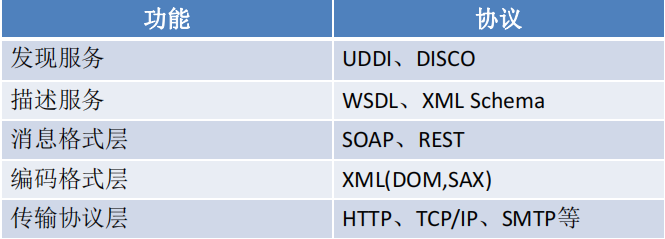
三个标准：松散耦合、粗粒度、标准化接口

* SOA是一种粗粒度，松散耦合服务架构。服务之间通过定义的接口进行通信，不涉及底层内容。
* 在SOA中，服务是一种为了满足某项业务需求的操作，规则的逻辑组合，包含一系列有序活动的交互。
* 从基于对象到基于构件，到基于服务，架构越来越松散耦合，粒度越来越粗，接口越来越标准。

基于服务的构件和传统构件的区别：

* 服务构件粗粒度，传统构件细粒度居多。
* 服务构建接口是标准的，主要是WSDL接口，传统构件经常以具体API出现。
* 服务构建与语言无关，传统构件通常绑定某种具体语言。
* 服务构件通过构件容器提供Qos服务，传统构件由程序代码控制。

SOA关键技术协议：



发现服务：

* UDDI：用于web服务注册和服务查找，描述了服务的概念，定义了编程的接口，提供给其他企业来调用（发现服务）。
* DISCO：发现公开服务的功能及交互协议。

描述服务：

* WSDL：网络服务描述语言，用来描述web服务的接口和操作功能（描述服务）。

消息格式层：

* SOAP：建立服务和服务之间的请求（实现服务远程调用）。
* REST：只使用HTTP和XML进行基于web通信的技术。

编码格式层：

* 扩展标记语言（XML）：标记电子文件使其具有结构性的标记语言。

传输协议层：

* HTTP，TCP/IP，SMTP等。

SOA实现方式：

* WEB Service：采用服务提供者，服务注册中心和服务请求者。服务提供者将服务发布到注册中心，请求者查找后，请求者绑定查找结果。底层传输层、服务通信协议层、服务描述层、服务层、业务流程层、服务注册层。
* 服务注册表：本质与Web Service类似，只是使用了注册表来代替服务注册中心。
* 企业服务总线ESB：包含客户端（请求者），基础架构服务（中间件，注册中心），核心集成服务（服务提供者）。

服务注册表：

* 服务注册：应用开发者（服务提供者）向注册表公布服务的功能。
* 服务位置：服务使用者（服务应用开发者），帮助他们查询注册服务，寻找符合自身要求的服务。
* 服务绑定：服务使用者利用检索到的服务接口来编写代码，所编写的代码将注册的服务绑定、调用注册的服务，以及与它们实现互动。

企业服务总线ESB特点：

* 提供位置透明性的消息路由和寻址服务。
* 提供服务注册和命名的管理功能。
* 支持多种的消息传递范型。
* 支持多种可以广泛使用的传输协议。
* 支持多种数据格式及其相互转换。
* 提供日志和监控功能。

微服务（粒度更小）：

* 可以独立部署。
* 轻量级的通信机制。
* 小且专注于一件事情。

微服务优势：

* 技术的异构性。
* 弹性。
* 易于扩展性。
* 简化的部署。
* 与对应的组织机构相匹配。
* 可组织性
* 可替代性的优化。

需要克服的困难（挑战）：

* 分布式系统的复杂行。
* 运维成本高。
* 部署需要自动化。
* 开发运维与组织架构。
* 服务间的依赖测试。
* 服务间的依赖管理

微服务与SOA的区别（来源百度，供参考）：

* 微服务剔除SOA中复杂的ESB企业服务总线，所有的业务智能逻辑在服务内部处理，使用Http（Rest API）进行轻量化通讯。
* SOA强调按水平架构划分为：前、后端、数据库、测试等，微服务强调按垂直架构划分，按业务能力划分，每个服务完成一种特定的功能，服务即产品。
* SOA将组件以library的方式和应用部署在同一个进程中运行，微服务则是各个服务独立运行。
* 传统应用倾向于使用统一的技术平台来解决所有问题，微服务可以针对不同业务特征选择不同技术平台，去中心统一化，发挥各种技术平台的特长。
* SOA架构强调的是异构系统之间的通信和解耦合；（一种粗粒度、松耦合的服务架构）
* 微服务架构强调的是系统按业务边界做细粒度的拆分和部署。

###### 架构描述语言ADL（管道-过滤器）

ADL是一种形式化语言，为软件系统概念体系结构建模提供了具体的语法和概念框架。

ADL基本构成要素：

* 构件和构件接口：计算单元或数据存储单元，是计算和状态存储的场所。
* 连接件：连接构件的，用于构件之间交互建模的体系结构。
* 架构配置：描述架构的构件和连接件的连接图。

主要的架构描述语言

* Aesop：支持体系结构风格的应用；
* MetaH：为设计者提供了关于实时电子监控软件系统的设计指导；
* C2：支持基于消息传递风格的用户界面系统的描述；
* Rapide支持体系结构设计的模拟并提供了分析模拟结果的工具；
* SADL：提供了关于体系结构加细的形式化基础；
* Unicon：支持异构的构件和连接类型并提供了关于体系结构的高层编译器；
* Wright：支持体系结构构件之间交互的说明和分析。

###### 特定领域软件架构DSSA

DSSA是专用于一类特定类型任务的，在整个领域能有效使用的，成功构造应用系统限定了标准的组合结构的软件构件的集合。（就是针对特定的领域，需要使用什么类型架构来满足需求）

DSSA是一个特定的问题领域中支持一组应用的领域模型，参考需求，参考架构等组成的开发基础，目标是支持在一个特定领域中多个应用的生成。

* 垂直域：在一个特定领域中的通用的软件架构，是一个完整的架构。
* 水平域：在不同领域间相同的需求（如购物和教育系统都有收费系统，收费系统就是水平域）。

DSSA三个基本活动：

* 领域分析：主要目的是获得领域需求和领域模型，识别信息源，基于获取到的数据确定哪些需求是领域中广泛共享的，建立领域模型。
* 领域设计：主要目的是获得DSSA，DSSA描述在领域模型中的需求的解决方案，不是单个系统的表示，而是能适应领域中多个系统需求的高层次的设计。
* 领域实现：主要目的是基于领域需求和DSSA开发出一些可以重用的构件。

参与DSSA的角色人员：

* 领域专家：该领域中有经验的用户，很了解这个领域的人。主要提供关于这个领域的知识。
* 领域分析人员：相当于系统分析师，控制整个领域分析过程，进行知识获取，将得到的知识组织到领域模型中。
* 领域设计人员：相当于软件设计师，根据领域模型开发出DSSA。
* 领域实现人员：详单与程序人员，根据领域模型和DSSA，开发出构件。

建立DSSA过程：

* 定义领域范围：领域中的应用要满足用户的需求。
* 定义领域特定的元素：建立领域的字典，归纳术语，识别与该领域有关的元素。
* 定义领域特定的设计和实现需求的约束：构件和连接件之间如何连接，识别领域中的所有约束。
* 定义领域模型和架构：产生领域的架构，描述构件说明。
* 产生、搜集可复用的产品单元：为DSSA增加复用的构件，使其可用于新系统。

以上过程是并发的，递归的，反复的，螺旋型的。

DSSA三层模型：

* 领域开发环境：由领域架构师来决定核心架构，产出参考结构，参考需求，架构，领域模型和开发工具等。
* 领域特定的应用开发环境：应用工程师根据具体环境将核心架构实例化。
* 应用执行环境：操作员实现实例化后的架构。

基于架构的软件开发ABSD

ABSD是架构驱动，强调由业务，质量和功能需求的组合驱动架构设计。强调采用视角和视图来描述软件架构，采用用例和质量属性场景来描述需求。用例描述的是功能需求，质量场景描述的是质量需求（或侧重于非功能需求）。

使用ABSD方法，设计活动可以从项目总体功能框架明确就开始，意味着需求获取和分析还没有完成就开始了软件设计。

ABSD有三个基础：

第一个是功能的分解，使用已有的基于模块的内聚和耦合技术。

第二个是通过选择架构风格来实现质量和业务需求。

第三个是软件模板的使用，软件模板利用了一些软件系统的结构。

ABSD方法是递归的，每个步骤都是清晰定义的，不管设计是否完成，架构总是清晰的。

ABSD开发过程：

* 体系结构需求：就是架构需求，重在掌握标识构件三步。
* 体系结构设计：就是架构设计，将需求阶段的标识构件映射成构件。
* 体系结构文档化：主要产出两种文档，架构（体系结构）规格说明书和测试架构（体系结构）需求的质量设计说明书。
* 体系结构复审：就是架构复审，对文档进行复审，由外部人员（如用户代表，领域专家）进行复审，看架构是否满足需求。
* 体系结构实现：用实体来显示出架构，实现构件，构件组装成系统。
* 体系结构演化。

###### 软件架构评估（非功能）

质量属性

* 性能：系统的响应能力，如响应时间，吞吐量。设计策略（如何提高性能）：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。
* 可靠性：软件在遇到错误时，是否还有维持软件系统功能的基本能力。如MTTF，MTBF。设计策略：（如何提高可靠性）：心跳，Ping/Echo，冗余，选举。
* 可用性：系统能正常运行的时间都比例，经常用两次故障之间的时间长度或出现故障恢复的速度来表示，如故障间隔时间。设计策略：心跳，Ping/Echo，冗余，选举。
* 安全性：系统在给合法用户提供服务时能阻止非授权用户使用，如保密性，完整性，不可抵赖性，可控性。设计策略：入侵检测，用户认证，用户授权，追踪审计，信息隐藏。
* 可修改性：是否可以很高的性价比和方便程序修改系统代码内容。基于功能的改变。设计策略：接口-实现分类，抽象。
* 功能性：系统能完成的工作的能力。
* 可变性：体系结构变更称为新的体系结构的能力。基于架构的改变。
* 互操作性：系统组成部分不是独立存在的，经常和其他系统相关作用，要有良好的API供系统之间相互调用。

软件架构评估：

* 敏感点：为了实现某一种质量属性，一个或多个构建所具有的特性。
* 权衡点：影响多个质量属性的特性，是多个质量属性的敏感点。（ps：题目中涉及多个质量属性的就是权衡点，只有一个质量属性的就是敏感点）
* 风险点和非风险点：只是一个常规概念，某个做法如果有隐患，可能会导致一些问题，称为风险点。某件事可以操作，称为非风险点。

软件架构评估是在架构设计之后，系统设计之前的。主要目的是为了评估所采用的架构是否能解决软件系统的需求，但不是单纯的确定是否满足需求。

软件架构评估方式：

* 基于调查文件（检查表）的方式：类似需求获取中的问卷调查方式，只不过是对架构方面的问卷，要求评估人员对领域熟悉。
* 基于度量的方式：制定定量的属性来度量架构，如代码行数。
* 基于场景的方式：是主要方法，确定应用领域的功能和软件架构结构之间的映射，设计场景，看架构对场景的支持程度。



从三个方面对场景进行设计：

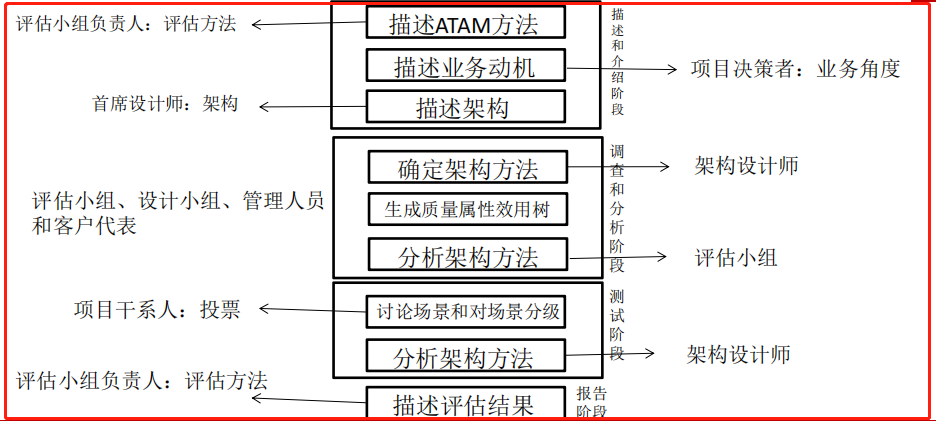
* 外部事件刺激。
* 环境（事件发生的环境）。
* 响应（架构响应刺激的过程）。

架构评估方法：

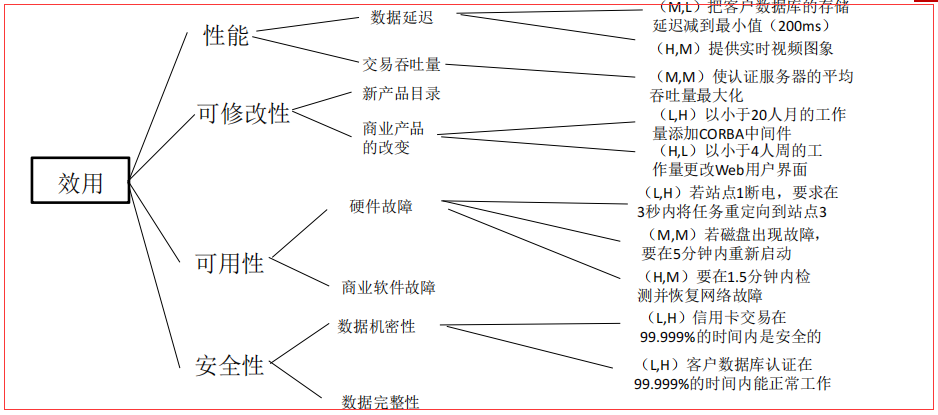
* 架构权衡分析法（ATAM）
* 基于场景的软件架构分析法（SAAM）
* 成本效益分析法（CBAM）

架构权衡分析法（ATAM）

让架构师明确如何权衡多个质量属性，参与者有评估小组，项目决策者和其他项目相关人



质量效用树：



基于场景的软件架构分析法（SAAM）

一种非功能质量属性的架构分析方法，主要输入是问题描述，需求说明和架构描述。步骤：

* 形成场景
* 描述架构
* 对场景分类和确定优先级
* 对场景进行单个评估
* 评估场景的相互作用
* 形成总体评价

场景分类：直接场景（直接可以实现的）和间接场景（要做修改才能实现的）

成本效益分析法（CBAM）

* 整理场景。
* 对场景进行求精。
* 确定场景的优先级。
* 分配效用。
* 形成“策略-场景-响应级别”的对应关系。
* 确定期望的质量属性响应级别的效用。
* 计算各架构策略的总收益。
* 根据受成本限制影响的投资报酬率选择架构策略。

###### 软件产品线

软件产品线是一个产品集合，这些产品共享一个公共的，可管理的特征集。这个特征集能满足特定领域的特定需求。

核心资源：包括所有产品所共用的软件架构，通用的构建，文档等。

产品集合：产品线中的各种产品。

软件复用：将已有的软件知识创建新的软件，缩减软件开发和维护的花费。早起的软件复用是代码级的复用，后来扩大到领域知识，开发经验，体系结构，需求，设计，文档等方面。

双生命周期模型：

* 领域工程生命周期：领域分析-领域设计-领域实现（类似于DSSA）。形成的是核心资源。
* 应用工程生命周期：需求分析-系统设计-系统实现。形成的是新系统。

SEI模型：

核心资源开发-产品开发-管理。

三生命周期模型：

企业工程-领域工程-应用工程。

软件产品线建立方式：

###### 

* 将现有产品演化为产品线。
* 用软件产品线替代现有产品集。
* 全新软件产品线的演化。
* 全新软件产品线的开发。

组织架构

* 设立独立的核心资源小组。
* 不设立独立的核心资源小组。
* 动态的组织结构。

形成条件：

* 对该领域具备长期和深厚的经验。
* 一个用于构建产品的好的核心资源库。
* 好的产品线架构。
* 好的管理（软件资源、人员组织、过程）支持。

###### 中间件技术

* 中间件是一种独立的系统软件或服务程序（非应用软件），可以帮助分布式软件在不同技术之间共享资源。JVM就是典型的中间件。
* 充当客户机和服务器之间的转化平台。
* 屏蔽硬件，操作系统，网络和数据库的差异。

中间件分类：

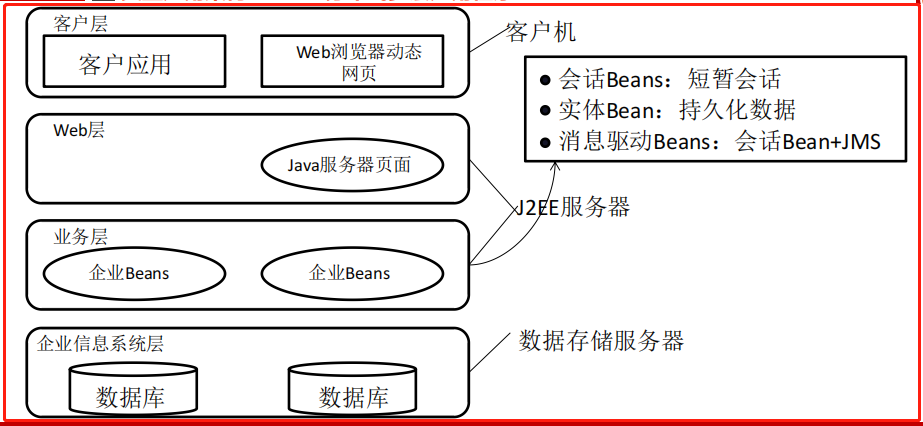
* 远程过程调用RPC：客户进程向服务器进程提出调用请求，客户端和服务器直接通信，是同步的。
* 对象请求代理ORB：对象之间通信是通过代理总线进行的。
* 远程方法调用RMI：服务器创建远程对象的方法，客户端调用这些方法。
* 面向消息的中间件MON：消息传递与排队算法。
* 事务处理监控器TPM：介于客户端和服务器之间，处理失败操作的中间件。

中间件技术CORBA（公共对象请求代理体系结构）：核心就是对象请求代理ORB，中间有一个中间件做请求代理，两边客户端通信，可以把逻辑处理部分放在中间件。

###### 典型应用架构

J2EE：分布式多层应用程序

四层架构：客户层，web层，业务层，企业信息系统层。



轻量级：SSH，性能，安全要求比较低。重量级：EJB，性能，安全要求比较高。

.NET：

* JVM与CLR：处理操作系统和应用语言之间，只适用于微软系统，而J2EE支持跨平台。
* 对多层分布式应用的支持：都支持。
* 安全性：都可以。
* 应用程序的部署：IIS/TOMCAT。
* 可移植性：J2EE更有优势。
* 外部支持：J2EE更有优势。

## 系统安全分析与设计

保密、防篡改（完整性）、防抵赖、身份认证。

###### 加密技术

###### 对称加密技术

###### 非对称加密技术