软件体系结构期末复习

第一章—软件体系结构概述

1、软件体系结构的主要思想

- 软件架构是一个软件系统的设计图，并不局限于软件系统的总体结构，还包含一些质量属性以及功能与结构之间的映射关系，即设计决策。
- 软件架构的两个主要焦点集中于系统的总体结构以及需求和实现之间的对应。
- 软件架构的主要思想是将注意力集中在系统总体结构的组织上。

2、软件架构的特征

特征	实现方式	作用
注重可重用性	组件及架构级重用	提高软件质量
利益相关者众多	满足各利益相关者需求	平衡需求
关注点分离	分而治之、模块化	简化复杂性
质量驱动	使用软件架构来处理质量属性需求、控制复杂性	由功能、数据流驱动向质量驱动转变
概念完整性	强调设计决策是一个持续的过程	每个决策都要在其前面设计决策的基础上进行
循环风格	架构风格、架构模式	用标准方法来处理反复出现的问题

3、软件架构的发展阶段

- **（1）无体系结构**

  高级语言出现

- **（2）基础研究阶段（1968-1994）**

  面向过程开发

- **（3）核心技术形成（1991-2000）**

  面向对象开发

  1）软件架构作为一个独立的研究领域出现

  2）软件架构核心技术的发展

  3）软件组件技术

- **（4）理论体系丰富（1996-1999）**

  面向服务开发

- **（5）理论体系完善及普及应用（1999-至今）**

  云和移动服务、智能化软件开发

第二章—软件架构定义

1、概述

- 组成派关注于软件本身，将软件架构看做组件和交互的集合。
- 决策派关注于架构中的实体(人)，将软件架构视为一系列重要设计决策的集合。

2、组成派定义

依据：软件架构主要反映系统由哪些部分组成，以及这些部分是如何组成的，强调系统的整体结构和配置。

（1）1992 —Dewane Perry & Alexander Wolf

软件架构 = {元素、组成、原理}

- 架构元素：，具有一定形式的结构元素，包括处理元素、数据元素、连接元素
- 架构组成：
  - 加权的属性：约束架构的选择
  - 关系：约束架构元素的放置
- 架构原理：捕获在选择架构风格、架构元素和架构形式的选择动机。

（2）Mary Shaw & David Carlan

软件架构包括组件、连接件和约束三大要素。

- 组件：可以是一组代码，也可以是独立的程序。
- 连接件：可以是过程调用、管道和消息等，用于表示组件间的相互关系。

- 约束：组件连接时的条件。

（3）2011年 ISO/IEC/IEEE标准

软件架构是某一系统的基本组织结构，其内容包括软件组件、组件间的联系、组件与其环境间的关系，以及指导上述内容设计与演化的原理。

3、决策派定义

依据：软件架构是软件设计的一部分，软件设计实际上是开发人员意志和决策在软件开发过程中的体现，更是高层领导和软件架构师意志和决策的体现。强调设计决策，更加注重架构风格和模式选择。

4、参考定义框架

组件（Component）、连接件（Connector）、配置（Configuration）、端口（Port）、角色（Role）

- 组件：具有某种功能的可重用的软件模块单元。
- 连接件：表示了组件之间的交互。
- 配置：表示了组件和连接件的拓扑逻辑和约束。
- 端口：组件的接口由一组端口组成，每个端口表示了组件和外部环境的交汇点。
- 角色：连接件的接口由一组角色组成，连接件的每个角色定义了该连接件表示的交互的参与者。
    - 二元连接件有两个角色
    - 有的连接件有多于两个的角色

第三章—软件架构模型

至今没有一种建模方法能够满足软件架构建模的所有要求。

1、软件架构的五类建模方法（过程、缺点）。

①基于非规范的图形表示的建模方法

图形可视化是将软件架构按照图形的方式进行表达，需要便于涉众阅读、理解和交流，使之不会因图形过于复杂而难以把握架构的概况

- 非正式图形表示：盒线图
- 正式图形表示：
    - 树形结构：

        （1）是显示层次性软件架构的理想方法。

        （2）难以处理复杂的问题。
    - 树地图（TreeMap）

        底层盒子往往用于表示方法，组合盒子往往用于表示类。

        （1）是展示整个软件层次架构的有效方法。

        （2）实质是一种空间填充方法。

- 改进的树地图
- 冰块图（Icicle Plot）

  每一行代表树的一个层次，按照子节点的数量进行分割。

  （1）有助于理解结构化的关系。

  （2）对于大型系统的层次化架构，这种可视化技术的扩展性和导航性存在问题。
- 旭日图（SunBurst）

  （1）具有较好的弹性：图中元素的角度和颜色。

  （2）与树地图相比，更易学习且更令人舒适。
- 双曲图（Hyperbolic）

  （1）双曲空间比欧几里得空间有更多的显示空间。

②基于UML的建模方法

用例图、类图、状态图、协作图、序列图、活动图、包图、组件图、部署图、复合结构、交互概述图、时序图

架构元素	UML模型组件
组件	分类器（如类、组件、节点、用例等）
接口	接口
关系（连接器）	关系（如泛化、关联、依赖等）
约束（规则）	规则

用UML建模的三种方法：

- （1）将UML看作是一种软件架构描述语言直接对架构建模。
- （2）通过扩展机制约束UML的元模型以支持软件架构模型的需要。
- （3）对UML的元模型进行扩充。

③基于形式化的建模方法

Z语言、Petri网、过滤器模式、B语言、VDM、CSP

④基于UML形式化的方法

UML不是一种形式化的语言

形式化与UML结合的建模过程：

需求分析—>需求文档规格说明—>UML建模—>形式化描述—>程序编码—>测试变量（形式规范自动生成）—>软件产品

- 注意：前四步占全部工作量的60-70%

⑤其他建模方法（文本语言、MDA）

文本语言建模方法

- 文本语言建模是通过文本文件描绘架构，这些文本文件通常需要符合某些特殊的句法格式。
- 可用方法：语法高亮显示、文本的静态检查、自动补全、代码折叠
- 种类：XML文本建模方法、xADLite文本建模方法
- 优势：
  （1）单个文档中描述整体架构，并且存在众多文本编辑器方便用户与文本文档的交互
  （2）许多工具能够生成程序库来对使用该语言的文本文档进行句法分析和检查。
  （3）许多编辑器附带额外的开发支持工具。
- 缺点：
  （1）用文本语言建模方法表示类图形结构就不易理解。
  （2）文本编辑器通常限于显示满屏的文本，很难以另外的方式组织文本。

MDA

- MDA不是一个实现分布式系统的软件架构，而是一个模型技术进行软件开发的方法
- MDA将模型区分为PIM和PSM
  - 平台无关模型（PIM）：PIM是一个系统的形式化规范，它与具体的实现技术无关。
  - 平台相关模型（PSM）：PSM基于某一具体目标平台的形式化规范。
- 它的核心思想是抽象出与实现技术无关、完整描述业务功能的平台独立模型。

2、“4+1”模型

（1）逻辑视图：支持行为要求。

- 描述系统各部分的抽象描述。用于建模系统的组成部分以及各组成部分之间的交互方式。
- 通常包括类图、对象图、状态图和协作图。

（2）过程视图：解决并发和分发。

- 描述系统中的进程。 当可视化系统中一定会发生的事情时，此视图特别有用。
- 该视图通常包含活动图、顺序图等。

（3）开发视图：组织软件模块，库，子系统，开发单元。

- 描述系统的各部分如何被组织为模块和组件。
- 该视图通常包含包图和组件图。
- 管理系统体系结构中的层非常有用。

（4）物理视图：将其他元素映射到处理和通信节点。

- 描述如何将前三个视图中所述的系统设计实现为一组现实世界的实体
- 该视图通常包含部署图，展示了抽象部分如何映射到最终部署的系统中。

（5）用例视图：将其他视图映射到重要的用例（这些用例被称作场景）上对体系结构加以说明。

- 从外部世界的角度描述正在建模的系统的功能。
- 需要使用此视图来描述系统应该执行的操作。 所有其他视图都依靠用例视图（场景）来指导，这就是将模型称为4 + 1的原因。
- 该视图通常包含用例图，描述和概述图。

第四章—软件架构风格

1、什么是软件架构风格?

- 软件体系风格描述了一类体系结构，独立于实际问题，强调了软件系统中通用的组织结构，在实践中被多次应用，是若干设计思想的综合，具有已经被熟知的特性，并且可以被复用。
- 软件架构风格又称软件架构惯用模式，是描述某一特定应用领域中系统组织方式的惯用模式，作为“可复用的组织模式和习语”，为设计人员的交流提供了公共的术语空间，促进了设计复用与代码复用。

2、使用架构风格的好处。

（1）可以极大的促进设计的重用性和代码的重用性，并且使得系统的组织结构易被理解。

（2）使用标准的架构风格可较好地支持系统内部的互操作性以及针对特定风格的分析。

3、经典体系结构风格的特点、优缺点、适用范围。（组件、连接件、约束）

3.1、管道过滤器风格

- 基本组件：过滤器（功能模块）
  - 每个过滤器组件中都封装了一个处理步骤
  - 数据源点和数据终点可以看作是特殊的过滤器
- 连接件：管道（数据流）
- 过滤器Filter：
  - 作用：将源数据变换为目标数据
  - 变换方式：（丰富）增加、（精炼）删除、（转换）改变、分解、合并等
  - 特性：独立性
    - （1）过滤器独立完成自身功能，相互之间无需进行状态交互
    - （2）过滤器自身无状态
    - （3）过滤器对其上下游的过滤器“无知”
- 管道Pipe：
  - 作用：将数据从一个过滤器的输出口转移到另一个过滤器的输入口
  - 过滤器是单向移动的。
  - 过滤器可以有缓冲区。
- **结果的正确性不依赖于各个过滤器运行的先后顺序**

- 优点：
  - （1）由于每个组件行为不受其他组件的影响，整个系统的行为易于理解。
  - （2）管道-过滤器风格支持功能模块的复用。
  - （3）具有较强的可维护性和可拓展性
  - （4）支持一些特定的分析，如吞吐量计算和死锁检测等。
  - （5）具有并发性。
- 缺点：
  - （1）往往导致系统处理过程的成批操作。
  - （2）增加了过滤器具体实现的复杂性，系统性能不高。
  - （3）交互式处理能力弱。
- 应用场景：数据源源不断的产生，系统需要对这些数据进行若干处理。

3.2、主程序/子程序风格

- 组件：程序和明确可见的数据（程序=数据结构+算法）
- 连接件：程序调用和数据共享
- 约束：单线程
- 优点：
  - （1）结构化程序设计的典型风格，相对于非结构化设计逻辑清晰，易理解。
  - （2）开发过程采用逐步细化，将大系统分解为若干模块。
- 缺点：
  - （1）对数据存储格式的变化将会影响几乎所有的模块。
  - （2）在规模变大时将会难理解。
  - （3）难以支持有效的复用。
- 应用场景：它适用于可以通过过程定义的层次结构适当地定义计算的应用程序。

3.3、面向对象风格

- 特点：
  - （1）对象负责维护其表示的完整性；
  - （2）对象的表示对其他对象而言是隐蔽的。抽象数据类型的使用，以及面向对象系统的使用已经非常普遍。
- 应用场景：它适用于中心问题是识别和保护相关信息体（尤其是表示信息）的应用程序。
- 组件：管理器（例如，服务器、对象、抽象数据类型）
- 连接件：程序调用
- 约束：分散的，通常是单线程
- 优点：
  - （1）对象隐藏了其实现细节，使得对象的使用变得简单方便，而且具有很高的安全性和可靠性。
  - （2）设计者可将一些数据存取操作的问题分解成一些交互的代理程序的集合。
- 缺点：
  - （1）管理多个对象
  - （2）管理许多交互
  - （3）行为的分布式责任，难以理解
  - （4）捕获相关设计的族

3.4、层次化风格

- 特点：
  - （1）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。
  - （2）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。
  - （3）修改一层，最多影响两层，而通常只能影响上层。接口稳固，见谁都不影响。
  - （4）上层必须知道下层的身份，不能调整层次之间的顺序。
- 应用场景：它适用于涉及可以分层排列的不同服务类的应用程序。
- 组件：通常是复合的，例如程序的集合。
- 连接件：取决于组件的结构；通常是在受限的可见性下进行的过程调用。
- 约束：单线程
- 优点：
  - （1）支持基于可增加抽象层的设计，允许将一个复杂问题分解成一个增量步骤序列的实现。
  - （2）支持扩展。【维护、修改比较容易】
  - （3）支持重用。【接口稳定】
- 缺点：
  - （1）不是所有系统都容易用这种模式来构建；
  - （2）定义一个合适的抽象层次可能会非常困难，特别是对于标准化的层次模型。
  - （3）层层相调，影响性能。

3.5、事件驱动风格

- **基本思想**：组件不直接调用一个过程，而是发布或广播一个或多个事件。
- **特点**：事件的触发者并不知道哪些构件会被这些事件影响，相互保持独立。
- **事件分派策略**：广播式、选择广播式
- **优点**：
  - （1）组件之间关联较弱，一个组件出错将不会影响其他构件。
  - （2）提高软件的复用能力。
  - （3）系统便于升级。
- **缺点**：
  - （1）组件放弃了对计算机的控制权，完全由系统决定。
  - （2）存在数据交换问题。
  - （3）该风格中，正确性验证成为一个问题。

3.6、解释器风格

- **解释器&编译器**
  - 编译器不会执行输入的源程序代码，而是将其翻译为另一种语言，并输出到文件中以便随后链接为可执行文件并加以执行；在解释器中程序源代码被解释器直接加以执行。
  - 解释器的执行速度要慢于编译器产生目标代码的执行速度，却低于编译器“编译+链接+执行”的总时间。
- **解释器的三种策略**
  - 传统解释器:纯粹的解释执行
  - 基于字节码的解释器：编译为字节码—>解释执行

    - JIT编译器：编译||解释执行（模糊了解释器、字节码解释器和编译器之间的边界与区分）
- **优点**：
    - （1）有利于实现程序的可移植性和语言的跨平台能力。
    - （2）可以对未来的硬件进行模拟和仿真，降低测试所带来的复杂性和昂贵花费。
- **缺点**：
    - （1）额外的间接层次导致了系统性能的下降。

3.7、基于规则的系统风格

- **核心思想**
    - 将业务逻辑中可能频繁发生变化的代码从源代码中分离出来。
- **优点**：
    - （1）降低了修改业务逻辑的成本。
    - （2）缩短了开发时间。
    - （3）将规则外部化，可在多个应用之间共享。
    - （4）对规则的改变将会非常迅速并且具有较低的风险。
- **缺点**：
    - （1）额外的间接层次导致了系统性能的下降。

3.8、仓库风格

- **应用场景**：应用于核心问题是建立、扩充和维护一个复杂的中央信息体的情况。
    - 数据处理，主要需要用传统的数据库来搭建业务决策系统。
    - 软件开发环境，主要需要表示和操作相关的程序和设计。
- **组件**：
    - 中心数据结构组件，表示当前数据的状态。
    - 相对独立的组件集合，各个功能模块（子系统）等。
- **连接件**：数据仓库与独立组件之间的交互
    - 由输入流中事务触发系统相应的进程执行—>数据库型知识库。
    - 由中心数据结构的当前状态触发系统相应的进程执行—>黑板知识库。
- **优点**：
    - （1）便于模块间的数据共享。
    - （2）方便模块的添加、更新和删除。
    - （3）避免了知识源的不必要的重复存储等。
- **缺点**：
    - （1）对于各个模块，需要一定的同步/加锁机制保证数据结构的完整性和一致性等。

3.9、黑板系统风格

- **组成部分**
    - 知识源：是描述某个独立领域问题的知识及其处理方法的知识库。
    - 黑板数据结构：
    - 控制器：时刻监视黑板状态变化
- **优点**：
    - （1）便于多客户共享大量数据。

- （2）即便于添加新的作为知识源代理的应用程序，也便于扩展共享的黑板数据结构。
- （3）知识源可重用。
- （4）支持容错性和健壮性。

- **缺点**：
  - （1）不同的知识源代理对于共享数据结构要达成一致，这也造成了对黑板数据结构的修改较为困难—要考虑到各个代理的调用。
  - （2）需要一定的同步/加锁机制保证数据结构的完整性和一致性，增大了系统复杂度。

3.10、C2风格

通过连接件绑定在一起的按照一组规则运作的并行组件网络。该规则规定了所有组件之间的交互必须是通过异步消息机制来实现。【不能跨层连接】

- **优点**：
  - （1）可使用任何编程语言开发组件，组件重用和替换易实现。
  - （2）组件相互独立，依赖较小，具有一定的扩展能力，可支持不同粒度的组件。
  - （3）组件不需要共享地址空间。
  - （4）可实现多个用户和多个系统之间的交互。
  - （5）可使用多个工具集和多媒体类型，动态更新系统框架结构。
- **缺点**：
  - （1）不太适合大规模流式风格系统，以及对数据库使用比较频繁的使用。

3.11、C/S风格

两层C/S架构

- 优点：
  - (1)客户机组件和服务器组件分别运行在不同的计算机上，有利于分布式数据的组织和处理。
  - (2)组件之间的位置是相互透明的。
  - (3)客户机程序和服务器程序可运行在不同的操作系统上，便于实现异构环境和多种不同开发技术的融合。
  - (4)软件环境和硬件环境的配置具有极大的灵活性，易于系统功能的扩展。
  - (5)将大规模的业务逻辑分布到多个通过网络连接的低成本的计算机上，降低了系统的整体开销。
- 缺点：
  - (1)开发成本较高(客户机的软硬件要求高)。
  - (2)客户机程序的设计复杂度大，客户机负荷重。
  - (3)信息内容和形式单一。
  - (4)C/S架构升级需要开发人员到现场更新客户机程序，对运行环境进行重新配置，增加了维护费用。
  - (5)两层C/S结构采用了单一的服务器，同时以局域网为中心，难以扩展到Internet。
  - (6)数据安全性不高，客户端程序可以直接访问数据库数据。

三层C/S架构

- 相比于两层的优点
  - (1)合理地划分三层结构的功能，可以使系统的逻辑结构更加清晰，提高软件的可维护性和可扩充性。
  - (2)在实现三层C/S架构时，可以更有效地选择运行平台和硬件环境，从而使每一层都具有清晰的逻辑结构、良好的负荷处理能力和较好的开放性。

    - (3)在C/S架构中，可以分别选择合适的编程语言并行开发。
    - (4)系统具有较高的安全性。
- 需要注意的问题
    - (1)如果各层之间的通信效率不高，即使每一层的硬件配置都很高，系统的整体性能也不会太高。
    - (2)必须慎重考虑三层之间的通信方法、通信频率和传输数据量，这和提高各层的独立性一样也是实现三层C/S架构的关键性问题。

3.12、B/S风格

- **优点**
    - （1）客户端只需要安装浏览器，操作简单，能够发布动态信息和静态信息。
    - （2）运用HTTP标准协议和统一客户端软件，能够实现跨平台通信。
    - （3）开发成本比较低，只需要维护Web服务器程序和中心数据库。客户端升级可以通过升级浏览器来实现。
- **缺点**:
    - （1）个性化程度比较低，所有客户端程序的功能都是一样的。
    - （2）客户端数据处理能力比较差，加重了Web服务器的工作负担，影响系统的整体性能。
    - （3）在B/S架构中，数据提交一般以页面为单位，动态交互性不强，不利于在线事务处理。
    - （4）B/S架构的可扩展性比较差，系统安全性难以保障。
    - （5）B/S架构的应用系统查询中心数据库，其速度要远低于C/S架构。

3.13、平台/插件风格

将待开发的目标软件分为两个部分：（1）程序的主体框架，可定义为平台。（2）功能扩展或补充模块，可定义为插件。

- **优点**:
    - （1）降低系统各模块之间的互依赖性。
    - （2）系统模块独立开发、部署、维护。
    - （3）根据需求动态的组装、分离系统。
- **缺点**:
    - （1）插件是别人开发的可以用到某主程序中的，只服务于该主程序，可重用性差。

3.14、面向Agent风格

认为事物的属性，特别是动态特性在很大程度上受到与其密切相关的人和环境的影响，将事务的主观与客观特征相结合抽象为系统的agent，作为系统的基本构成单位，通过agent之间的合作实现系统的整体目标。

- **优点**:
    - （1）面向Agent的软件工程方法对于解决复杂问题是一种好的技术，特别是对于分布开放异构的软件环境。
- **缺点**:
    - （1）大多数结构中Agent自身缺乏社会性结构描述和与环境的交互。

3.15、面向方面软件架构风格

尽量分离“技术问题实现”和“业务问题实现”。允许开发者能够对横切关注点进行模块化设计。

- **优缺点分析**
  - （1）可以定义交叉的关系，并将这些关系应用于跨模块的、彼此不同的对象模型。
  - （2）AOP同时还可以让我们层次化功能性而不是嵌入功能性，从而使代码由更好的可读性和易维护性。
  - （3）它会和面向对象编程可以很好地合作，互补。

3.16、面向服务架构风格

服务请求者、服务提供者、服务注册中心

具有基于标准、松散耦合、共享服务和粗粒度等优势，易于集成现有系统、具有标准化的架构、提升开发效率、降低开发维护复杂度。

- **优点**：
  - （1）灵活性，根据需求变化，重新编排服务。
  - （2）对IT资产的复用。
  - （3）使企业的信息化建设真正以业务为核心。业务人员根据需求编排服务，而不必考虑技术细节。
- **缺点**：
  - （1）服务的划分很困难。
  - （2）服务的编排是否得当。
  - （3）如果选择的接口标准有问题，会带来系统额外开销和不稳定性。
  - （4）对IT硬件资产还谈不上复用。
  - （5）主流实现方式接口很多，很难统一。
  - （6）主流实现方式只局限于不带界面的服务的共享。

3.17、正交架构风格

是一种以垂直线索组件族为基础的层次化结构，其基本思想是把应用系统的结构按功能的正交相关性，垂直分割为若干个线索，线索又分为几个层次，每个线索由多个具有不同层次功能和不同抽象级别的组件构成。

- **特点**：
  - （1）由完成不同功能的n（n>1）个线索（子系统）组成。
  - （2）系统具有m（m>1）个不同抽象级别的层。
  - （3）线索之间是相互独立（正交）的。
  - （4）系统有一个公共驱动层（一般为最高层）和公共数据结构（一般为最底层）。
- **优点**：
  - （1）结构清晰，易于理解。
  - （2）易修改，可维护性强。
  - （3）可移植性强，重用粒度大。
- **缺点**：
  - （1）在实际应用中，并不是所有软件系统都能完全正交化，或者有时完全正交化的成本太高。因此，在进行应用项目的软件架构设计是，必须反复权衡进一步正交化的额外开销与所得到的更好的性能之间的关系。

3.18、异构风格

- **优点**：
  - （1）可以实现遗留代码的重用。
  - （2）在某一单位中，规定了共享软件包和某些标准，但仍会存在解释和表示习惯上的不同。而异构风格可以解决这一问题。
- **缺点**：
  - （1）不同风格之间的兼容问题很难解决。

3.19、基于层次消息总线的架构风格

基于层次消息总线、支持组件的分布和并发，组件之间通过消息总线进行通讯。

- **优点**
  - （1）构件接口是一种基于消息的互联接口，可以较好的支持架构设计。降低了构件之间的耦合性，增强了构件之间的重用性。
  - （2）支持运行时的系统演化，主要体现在可动态增加和删除构件，动态改变构件所响应的消息以及消息过滤这三个方面。
- **缺点**
  - （1）可重用要求高，可重用性差。

3.20、模型-视图-控制器风格

MVC结构主要包括模型(封装问题的核心数据、逻辑关系和计算功能，提供处理问题的操作过程)、视图（提供交互界面）和控制器（处理用户与系统之间的交互）。

- **应用场景**：用户交互程序的设计中
- **优点**：
  - （1）多个视图和一个模型相对应，便于维护。
  - （2）具有良好的移植性。
  - （3）当功能发生变化时，改变其中的一部分就能满足要求。
- **缺点**：
  - （1）增加了系统设计和运行的复杂性。
  - （2）视图和控制器连接的过于紧密，妨碍了二者的独立重用。
  - （3）视图访问模型的效率比较低。
  - （4）频繁访问未变化的数据，也将降低系统性能。

第六章—软件架构与敏捷开发

1、敏捷开发的基本理念。

（1）强调个体和互动比强调过程和工具更好。

（2）强调获得可运行的软件比强调完成详尽的文档好。

（3）强调与客户合作比强调进行详细的合同谈判好。

（4）强调响应变化比强调遵循既定的计划好。

2、敏捷开发与架构设计的关系。

（1） 软件架构与敏捷开发的出发点是一致的。

- 软件架构与敏捷开发都是一个权衡的过程：软件架构设计需要权衡涉众们的各种需求，在众多的解决方案中确定唯一的架构设计；敏捷开发是在软件开发过程混沌和大量开发管理活动加入的两个极端中做出的一种权衡。
- 软件架构与敏捷开发目的都是为了提高软件开发效率、提高软件质量、降低软件成本，将开发团队的价值最大化。

（2）敏捷开发也需要重视软件架构。

- 软件架构设计对于敏捷开发来说也是必要的。两者在软件开发实践中能够共同存在，且互相促进。

（3）敏捷开发改变了软件架构的设计方式。

- 敏捷开发将详细架构设计转移到Code编码阶段、重构阶段、单元测试阶段等。

3、敏捷开发中如何改变了软件架构的设计方式?

敏捷开发把传统软件开发前期的详细架构设计，分散到了整个敏捷开发软件过程中，以达到提高效率、减少风险的目的。

- **需求分析**

  敏捷开发中的需求分析引入了架构设计的理念，分为初始阶段需求分析和迭代阶段需求分析。

- **初始设计**

  初始设计需要对软件系统的设计进行全局抽象层次上的考虑。包括系统的基本处理流程、系统的组织结构、模块划分、功能分配等

- **迭代过程**

  迭代设计、重构、确定架构、客户交流

敏捷开发的思想在软件架构设计中最主要的体现就是团队设计和简单设计这两种设计理念。

- **团队设计**
  - 优点：其结论要比个人决策更加完整，避免个人遗漏，相对稳定、周密。
  - 缺点：需要额外付出沟通成本、决策效率低、责任不明确。
- **简单设计**
  - 简单体现在两个方面：表达方式的简单和现实抽象的简单化。
  - 简单设计可以降低开发成本、提升沟通效率、增强适应性和稳定性。

第八章—软件架构设计和实现

1、成功的软件架构应具有的品质。

（1）良好的模块化。

（2）适应功能需求的变化，适应技术的变化 。

（3）对系统的动态运行有良好的规划

（4）对数据的良好规划。

（5）明确、灵活的部署规划。

2、基于体系结构的软件设计方法。

基于体系结构的软件设计（ architecture-based software design, ABSD）方法为软件系统的概念体系结构提供构造方法，概念体系结构描述了系统的主要设计元素及其关系。概念体系结构代表了在开发过程中做出的第一个选择，它是达到系统质量和业务目标的关键，为达到预定功能提供了基础。

ABSD方法基础

- 功能分解：ABSD方法使用已有的基于模块的内聚和耦合技术；
- 通过选择体系结构风格来实现质量和业务需求。
- 软件模板的使用：利用一些软件系统的结构。

ABSD方法的步骤

（1）功能分解：分解的目的是使每个组在体系结构内代表独立的元素。

（2）选择体系结构风格

（3）为风格分配功能

（4）细化模板

（5）功能校验

（6）创建并发视图

（7）创建配置视图

（8）验证质量场景

（9）验证约束

3、将软件架构的概念和原则引入软件需求阶段有什么好处？不引入可能会引起什么问题?

- 若把架构概念引入需求分析阶段，有助于保证需求规约、系统设计之间的可追踪性和一致性，有效保持软件质量。
- 将软件架构概念和原则引入需求分析，也可以让我们获得更有结构性和可重用的需求规约。
- 用传统的方法产生需求规约，不考虑软件架构概念和原则，则在软件架构设计阶段建立需求规约与架构的映射将相对困难。

4、软件架构和软件需求是如何协同演化的?

- 软件需求和软件架构两者是相辅相成的关系，一方面软件需求影响软件架构设计，另一方面软件架构帮助需求分析的明确和细化。
- 需求与架构的互相影响可以看作一个螺旋的过程，也是一个双峰的过程。在一个反复的过程中，产生更详细的需求规约和设计规约，最终把交织在软件开发过程中的设计规约和需求规约分离开来。

5、将软件架构映射到详细设计经常遇到什么问题？如何解决?

问题:

（1）缺失重要架构视图，片面强调功能需求。

（2）不够深入，架构设计方案过于笼统，基本还停留在概念性架构的层面，没有提供明确的技术蓝图。

（3）名不副实的分层架构，缺失层次之间的交互接口和交互机制，只进行职责划分。

（4）在某些方面过度设计。

解决方法:

（1）对于缺失重要架构视图问题，可以针对遗漏的架构视图进行设计。

（2）对于不够深入问题，需要将设计决策细化到和技术相关的层面。

（3）对于名不副实的分层架构问题，需要步步深入，明确各层之间的交互接口和交互机制。

（4）虽然我们必须考虑到系统的扩展性，可维护性等，但切忌过度设计。

6、MDA的基本思想、过程，应用MDA的好处。

MDA——model driven architecture，基于模型驱动软件架构。

基本思想:

将软件系统分成模型和实现两部分：模型是对系统的描述，实现是利用特定技术在特定平台或环境中对模型的解释。模型仅仅负责对系统的描述，与实现技术无关。这是模型的实现技术无关性。

过程:

（1）用计算无关模型CIM 捕获需求;

（2）创建平台无关模型PIM;

（3）将PIM转化成为一个或多个平台特定模型PSM，并加入平台特定的规则和代码;

（4）将PSM 转化为代码等。

好处:

将模型与实现分离后，能够很好的适应技术易变性。由于实现往往高度依赖特定技术和特定平台，当技术发生迁移时，只需针对这种技术作相应的实现，编写相应的运行平台或变换工具。所以，能够比较好的应对实现技术发展带来的挑战。

第十五章—软件体系结构评估

1、质量属性、（质量）场景。

质量属性：

（1）可修改性：度量软件系统变化的成本。

（2）可用性：是指软件能够正常运行的时间比例。

（3）性能：性能表示软件系统的响应速度或者由响应速度决定的其它度量。

（4）可测试性：软件系统在多大程度上容易被测试检查出缺陷。

（5）易用性：表明软件系统完成后用户的体验和效率。

（6）安全性：代表软件对未授权和非法操作的防卫能力。

场景：

- 在进行体系结构评估时，一般首先要精确地得出具体的质量目标，并以之作为判定该体系结构优劣的标准。我们把为得出这些目标而采用的机制叫做场景。
- 场景是从风险承担者的角度对与系统的交互的简短描述。
- 在体系结构评估中，一般采用刺激、环境和响应三方面来对场景进行描述。

2、体系结构权衡分析方法（ATAM）的相关概念（敏感点、权衡点、质量效用树）、评估过程（步骤）、质量效用树的构建、优缺点。

ATAM-Architecture Tradeoff Analysis Method

（1）概念：

- **敏感点：**敏感点是一个或者多个构件的特征，可以使设计师搞清楚实现质量目标时应该注意什么。
- **权衡点：**权衡点是影响多个质量属性的特征；是多个质量属性的敏感点；权衡点需要进行权衡。
- **质量效用树：**效用树为我们提供了一种直接而有效地将系统的业务驱动因素转换为具体的质量属性场景的机制，该步骤的输出结果是对具体质量属性需求（以场景形式实现）的优先级的确定。
- **风险承担者、涉众、牵涉到的人：**体系结构设计师、开发人员、维护人员、集成人员、测试人员、标准专家、性能工程师

（2）ATAM评估步骤：

ATAM主要部分包括4组，共9个步骤。

- **1）陈述，包括通过它进行的信息交流**
  - ①ATAM方法的陈述
  - ②商业动机的陈述
  - ③SA的陈述
- **2）调查与分析，包括对照体系结构方法评估关键**
  - ④确定体系结构方法
  - ⑤生成质量效用树
  - ⑥分析体系结构方法：
- **3）测试，包括对照所有相关人员的需求检验最新结果**
  - ⑦集体讨论并确定场景优先级
  - ⑧分析体系结构方法

- **4）形成报告，包括陈述ATAM的结果**
    - ⑨结果的表述

（3）质量效用树的构建

- 效用树中质量属性细化为场景
- 确定最重要的质量属性目标，并设置优先级
- 效用树设置优先级标准

（4）优缺点

优点：

- 考虑了所有与系统相关的人员对质量的要求。
- 涉及到的基本活动包括确定应用领域的功能和软件体系结构之间的映射，设计用于体现待评估质量属性的场景以及分析软件体系结构对场景的支持程度。

缺点：

- 基于场景的评估方式是特定领域的，对一个领域适合的场景设计在另一领域未必适合。
- 实施者一方面需要有丰富的领域知识以对某质量需求设计出合理的场景，另一方面，必须对待评估的软件体系结构有一定的了解以判断是否支持场景描述的一系列活动。

软件架构相关课题

1、软件架构演化与维护

软件架构演化就是为了维护软件架构自身的有用性。

（1）对象演化 （2）消息演化（3）复合片段演化（4）约束演化

针对软件架构的演化过程是否处于系统运行时期，可以将软件架构演化分为静态演化和动态演化。

- 静态演化：发生在软件架构设计、实现和维护过程中，软件系统还未运行或者处于运行停止状态。
- 动态演化：发生在软件系统运行过程中。

2、架构腐蚀

软件架构腐蚀（software architecture erosion）是指预期软件架构或概念软件架构与实际软件架构之间的偏离。它意味着最终的实现并没有完全满足预定的计划或违背了系统的约束和规则。这种偏离更多的是源自日常的软件修改，而非人为的恶意。架构腐蚀会导致软件演化过程中出现工程质量的恶化。

预防方法：

- 腐蚀最小化
- 腐蚀预防
- 腐蚀修补

3、架构技术债

技术债是指开发人员为了加速软件开发，或是由于自身经验的缺乏，有意或无意的在应该采用最佳方案的时候进行了妥协，使用了短时期能加速软件开发的方案，从而在未来给自己带来额外的开发负担。

分类：

- 代码债
- 设计债
- 测试债
- 文档债

4、架构坏味道

- 如果程序中某一段代码是不稳定的或者有一些潜在的问题，那么该段代码往往会包含一些明显的不太好的痕迹。我们称这些痕迹为代码坏味道。架构坏味道定义和代码坏味道类似，只是架构坏味道在系统粒度下出现的层次要高于代码坏味道。
- 架构坏味道是一种通常使用的，可以对系统生命周期特性产生消极影响的架构设计。它可能是由于在不适当的环境下应用了一个不适合的解决方案或者在错误的粒度层次下应用了某个设计抽象等产生的，会对系统的可理解性、可测试性、可扩展性以及可重用性等产生负面影响。
- **典型的架构坏味道：**连接件嫉妒、过度分散的功能、模糊接口、无关的相邻连接件、砖关注过载、砖使用过载、砖循环依赖、未使用接口、重复的组件功能、组件嫉妒、连接件链。

5、架构脆弱性

- 软件（系统）架构设计存在一些明显的或者隐含的缺陷，攻击者可以利用这些缺陷攻击系统，或者当受到某个或某些外部刺激时，系统发生性能下降、稳定性下降、可靠性下降、安全性下降等等。如果软件架构具备这类缺陷，我们认为该软件架构是脆弱的，也就是软件架构脆弱性。
- 软件架构脆弱性通常与软件架构的风格和模式有关，不同风格和模式的软件架构，脆弱性体现和特点有很大不同。