计算机学院软件工程与实践课程实验报告

|  |  |  |
| --- | --- | --- |
| 实验题目：软件体系结构设计及SAD（一） | | 学号：201900130107 |
| 日期：2022/5/2 | 班级： 计科19级1班 | 姓名： 郑晓旭 |
| Email：1250102275@qq.com | | |
| 实验目的：  1. 学习对比软件体系结构设计GB和IEEE最新SAD(Software Architecture Document)的标准  2. 研究经典软件体系结构案例  3. 给项目添加提高质量属性的设计 | | |
| 实验软件和硬件环境：  Windows10 | | |
| 实验步骤与内容：  1. 分工协作，参考国标“13 - 软件(结构)设计说明(SDD)”等资料，对比参考SAD最新标准IEEE-42010.pdf，针对自己的项目设计SAD初稿。  周四晚实验课上，经过讨论，我们各自领取了自己在SAD中的分工，初步分工如下：  赵一帆5；史子涵，刘诗婷，陈文盼 4；郑晓旭 其他。计划本周末前，每个组员将自己写的初稿发在qq群中，并针对自己在撰写中遇到的问题，也就是我们在前期分析中没有考虑周全的地方，提出问题后大家全体进行分析和解决。最终组长进行汇总，形成SAD的初稿。  2. 分工协作，学习、检索研究经典软件体系结构案例。  On-the-Criteria-To-Be-Used-in-Decomposing-Systems-into-Modules.pdf  <http://www.cs.cmu.edu/~ModProb/index.html、>  [软件体系结构案例分析 - 豆丁网 (docin.com)](https://www.docin.com/p-852016844.html)  [软件体系结构案例 - 百度文库 (baidu.com)](https://wenku.baidu.com/link?url=acV9L6-U8nxsjpJ8Ne2SrSObnh2R8E8gz8Ftj-PtceQvpW-7cuAf8EtTaj_MnAi5-NWvzC742NN4lpg4V3hnUdDuEEGl9KkfTAu062QpPac3jI4HkR-tNLsb_rr--erU1w7Mm3Ke4uSZLBeCTsQpBK)  [经典软件架构模式（完） - 云+社区 - 腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1050603)  [最经典的两种软件架构模式 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/136371486)  3. 学习、检索课本5.17参考文献及以下推荐的参考书或网上检索新的有关软件体系结构的资料。小组分工，每位成员选择自己关注的部分专题学习并写出学习报告（笔记）（附到最终提交的SAD）。  **An Introduction to Software Architecture学习报告（赵一帆）**  我们知道，随着软件系统规模的增大，计算的算法和数据结构不再是主要的设计问题。当系统由许多组件构成时，整个系统的组织——软件体系结构会出现一组新的设计问题。在这《An Introduction to Software Architecture》这本书中，作者介绍了软件架构这个新兴的领域。我们考虑的一些常见体系结构，目前都基于这些样式，并且书中也展示如何在一个设计中组合不同的样式。书的第四章节提出6个案例研究来说明架构表示如何提高我们对复杂软件系统的理解。书的最后对该领域的一些突出问题进行了综述，并考虑了一些有前景的研究方向。  本书的第一章节主要是对本书内容的一个概括，它强调了软件体系结构设计在软件系统设计中的重要性，甚至这个问题超越了计算的算法和数据结构。而且越来越清楚的是，有效的软件工程一定有着便利在软件结构设计中。关于这方面他提出了四点：首先，能够识别通用范例是很重要的，这样就可以理解系统之间的高级关系，并且可以将新系统构建为旧系统的变体；其次，获得正确的体系结构通常是软件系统设计成功的关键，错误的选择可能会导致灾难性的后果；第三，对软件架构的详细理解使工程师能够在设计方案中做出有原则的选择；第四，在分析和描述复杂系统的高级属性时，体系结构系统表示通常是必不可少的。  本书的第二章节是“从编程语言到软件架构”。为了正确地看待软件体系结构领域，作者首先让我们了解计算机科学中抽象技术的历史发展。第一小部分介绍了高级程序设计语言。最初刚出现的计算机的指令都是机器指令，即程序员把指令和数据分别显式地放在计算机的内存中，在程序中插入新指令可能需要对整个程序进行手动检查才能更新。之后人们认识到可以自动地对引用进行内存布局和更新，还可以将符号名用于操作代码和内存地址，于是产生了符号汇编器。不久之后出现了宏处理器，再之后便有了高级语言。  第二章节第二部分主要是讲抽象数据类型的作用。优秀的程序员明白：如果数据结构正确，那么开发程序的其他部分就会容易得多。抽象数据类型的设计水平提高到编程语言语句或单个算法的水平之上。这种形式的抽象导致了对服务于一个特定目的的整个模块的良好组织的理解。这涉及到以统一的方式组合表示、算法、规范和功能接口。  第二章节第三部分便讲了软件架构。就像好的程序员在20世纪60年代后期认识到有用的数据结构一样，好的软件系统设计人员现在认识到有用的系统组织。其中之一是基于抽象数据类型的理论，但这并不是组织软件系统的唯一方法。随着不断地发展，软件系统设计师这个词也出现了。我们现在可以清楚地识别许多体系结构模式或风格，它们目前构成了软件架构师的基本技能。  本书第三章讲了常见的架构风格。从第一小节到第八小节分别是管道和过滤器，数据抽象和面向对象的组织，基于事件的隐式调用，分层系统，存储库，表格驱动的解释器，其他架构，异构体系结构：  1.在管道和过滤器样式中，每个组件都有一组输入和一组输出。组件在其输入上读取数据流，并在其输出上生成数据流，以标准顺序交付结果的完整实例。这通常是通过对输入流应用本地转换和增量计算来实现的，以便在使用输入之前就开始输出。因此组件被称为“过滤器”。这种类型的连接器充当流的管道，将一个过滤器的输出传输到另一个过滤器的输入。因此，连接器被称为“管道”。  2.在数据抽象和面向对象的组织中，数据表示及其相关的原语操作被封装在抽象数据类型或对象中。这种样式的组件是对象—或者，如果愿意，是抽象数据类型的实例。对象是我们称为管理器的某种组件的示例，因为它负责保持资源的完整性。对象通过函数和过程调用进行交互。这种风格的两个重要方面是一个对象负责保持其表示的完整性，以及该表示对其他对象隐藏。  3. 传统上，在组件接口提供一组过程和函数的系统中，组件通过显式调用这些例程来相互交互。然而，最近人们对另一种集成技术非常感兴趣，这种技术被称为隐式调用、反应式集成和选择性广播。这种风格的历史根源在于基于参与者、约束满足、守护进程和包交换网络的系统。  4. 分层系统按层次结构组织，每一层为其上一层提供服务，并作为下一层的客户端。在一些分层的系统中，除了为导出而精心选择的某些功能外，内层对所有其他层都是隐藏的。因此，在这些系统中，组件在层次结构的某个层实现虚拟机。(在其他分层系统中，层可能只是部分不透明。)连接器由决定层如何交互的协议定义。拓扑约束包括限制相邻层的相互作用。  5. 在存储库风格中有两种截然不同的组件:中央数据结构表示当前状态，一组独立的组件对中央数据存储进行操作。存储库及其外部组件之间的交互在系统之间可能有很大的差异。  6. 在解释器组织中，虚拟机是由软件产生的。解释器包括被解释的伪程序和解释引擎本身。伪程序包括程序本身和解释器对其执行状态的模拟(激活记录)。解释引擎包括解释器的定义和执行的当前状态。因此，解释器通常有四个组成部分:执行工作的解释引擎、包含要解释的伪代码的内存、解释引擎的控制状态的表示，以及正在模拟的程序的当前状态的表示。  本书的第四章节通过6个例子来说明如何使用架构原则来增加我们对软件系统的理解。第一个示例展示了针对同一问题的不同体系结构解决方案如何提供不同的好处。第二个案例研究总结了为一系列工业产品开发特定于领域的体系结构风格的经验。第三个案例研究以全新的视角审视了我们熟悉的编译器体系结构，剩下的三个案例研究展示了异构体系结构的使用实例。  1.第一个例子是文章中的关键词，即经典的KWIC案例，这个在软件工程教材中的第五章也出现了。通过信息库，数据抽象，隐含调用，管道和过滤器四种不同的体系结构实现，可以更好地决定使用哪种方法最好。由于本案例我们小组也在实现，所以我就简略写了。  2.第二个例子为仪器软件。该项目的目的是开发一个可重复使用的示波器系统架构。示波器是一种采集电信号并在屏幕上显示其图像(称为跟踪)的仪器系统。另外，示波器对信号进行测量，并在屏幕上显示出来。虽然示波器曾经是简单的模拟装置，涉及的软件很少，但现代示波器主要依靠数字技术，软件相当复杂。现代示波器执行几十次测量、提供兆字节的内部存储、与工作站网络和其他仪器的接口，以及提供复杂的用户界面，包括带有菜单的触摸屏、内置的帮助设备和彩色显示器，这些都是很常见的。所以需要开发特定于领域的体系结构风格的经验。  3.第三个例子为编译器的新新视图。本书这一小节以全新的视角审视了我们熟悉的编译器体系结构。  4.剩下三个例子分别为带有不同风格的分层设计，对组件使用不同习语的解释器，被重新定义为解释器的全球范围黑板。这三个实例也通过相应的软件结构设计介绍，展示了异构体系结构的使用实例。  第五章节主要是全书内容一个小总结，然后说明了软件结构设计不断发展。随着人们对这个新兴领域的兴趣日益浓厚，我们可以预期，随着时间的推移，我们对软件体系结构的原则和实践的理解将会有相当大的提高。然而，正如我们已经说明的，即使我们现在有了基本的概念，软件架构级别的设计仍然可以为软件工程实践提供直接和实质性的好处。  通过阅读这本著作，我收获了很多，了解了软件体系结构发展的一些相关历史，并且知道了一些这方面的分类，也了解了一些实例。这让我对之后软件体系结构的深入学习产生了兴趣，也打下了好的基础。  **Documenting Software Architectures 学习报告（刘诗婷）**  这本书总计582页，内容丰富，但是我们软件工程实验并不需要全部熟读这本书。根据我们的实验要求，我选读了第四章节，其讲述的内容主要是与我们相关的软件体系结构风格。以下是学习报告：  **组件和连接器风格**component-and-connector types  组件和连接器(C&C)风格引入了一个特定的组件和连接器类型集，并指定规则关于如何组合这些类型的元素。此外，考虑到C&C视图捕获系统的运行时方面，C&C样式通常也与计算型相关联模型，规定了数据和控制如何在按这种风格设计的系统中流动。  C&C样式(或多个样式)的选择通常取决于系统中运行时结构的性质。例如，如果系统需要访问一组遗留数据库，则该样式可能基于共享数据样式。或者，如果系统要执行数据流转换时，可能会选择一种数据流样式。样式的选择也将取决于文档的预期用途。例如，如果高性能是一个关键属性，那么可能会选择这种风格来进行性能分析，这样就可以评估影响系统质量的权衡。许多C&C风格的存在。为了理解这些样式的空间，我们首先描述一些常用的C&C样式的广泛类别，然后在每个类别中更详细地考虑一个或多个示例样式。C&C风格的空间很大。例如，C&Cstyles在它们所支持的连接器类型方面可能有很大的不同。基于异步事件广播(例如发布-订阅)的样式与基于同步服务调用的样式有很大的不同。类似地，样式可能在它们允许的组件类型方面有所不同要求。  例如，有些样式需要一个数据库组件。其他样式可能需要注册表组件来允许组件在运行时查找其他样式。样式可能在拓扑限制方面有所不同，例如组件是否分配给层。它们也可能在领域特异性水平方面有所不同。例如，支持汽车控制系统的样式可能涉及连接器表示实时协调的特定协议。类似地，根据所处理的应用程序域的性质，有几十种客户机-服务器样式以微妙(或不那么微妙)的方式不同。例如，一些客户机-服务器样式允许服务请求的后期绑定，其中请求的接收者是动态确定的;其他样式则坚持在构建或部署系统时确定静态配置。  该体系结构有四个适用范例：  调用返回风格，组件交互的样式同步调用他人提供的功能组件；数据流样式，属性驱动的计算方式通过系统的数据流；基于事件的风格，组件交互的样式异步事件或消息；库风格，组件交互的样式持久共享数据的大集合。  **数据流风格**Data Flow Styles  数据流样式体现了一种计算模型，其中组件充当数据转换器和连接器，将数据从一个组件的输出传输到另一个组件的输入。数据流样式中的每个组件类型都有若干个输入端口和输出端口。它的工作是消耗其上的数据输入端口并将转换后的数据写入其输出端口。在实践中出现了各种各样的数据流样式。在早期的在计算的日子里，一种常见的数据流样式是“批处理”顺序的，“每个组件转换所有它的数据在下一个组件可以使用它的输出之前。后来发明了一种数据流风格，组件并发运行，数据增量处理:管道和过滤器风格。目前，数据流样式在流处理发生的域，在此可以将整个计算分解为一组转换步骤。  数据流风格衍生出来的比较通用的一种风格就是：管道-过滤器体系结构。管道-过滤器样式中的交互模式的特征是数据流的连续转换。数据到达过滤器的输入端口，进行转换，然后通过管道将其输出端口传递到下一个过滤器。一个过滤器可以从多个端口使用数据，也可以向多个端口生成数据。这类系统的现代示例有信号处理系统、使用UNIX管道构建的系统、Apache Web服务器的请求处理体系结构、map-reduce搜索引擎的典范，雅虎!用于处理RSS提要的管道，以及许多必须处理和分析大量实验数据流的科学计算系统。管道-过滤器样式的基本形式，如表4.1所示，提供单一类型的组件—过滤器—和单一类型的连接—管道。过滤器转换通过一个或多个管道接收的数据，并通过一个或多个管道传输结果。过滤器通常并发地、递增地执行。管道是一个连接器，它将数据流从一个过滤器的输出端口传输到另一个过滤器的输入端口。管道充当单向的管道，提供一个保存顺序的缓冲通信通道来传输过滤器生成的数据。在纯管道和过滤器样式中，过滤器仅通过管道进行交互。  因为管道在通信期间缓冲数据，所以过滤器可以异步并发地工作。此外，过滤器不需要知道其上游或下游过滤器的标识。因此，管道和过滤器系统具有良好的形式化特性，即可以将整个计算视为过滤器计算的功能组合，从而允许架构师将端到端行为作为部分行为的简单组合进行推理。  管道和过滤器风格适用那种情况呢？  符合管道和过滤器风格的系统通常用于数据转换系统，其中整个处理过程可以分解为一组独立的步骤，每个步骤负责其输入数据的增量转换。每个步骤所完成的处理的独立性支持重用、并行化和关于整体行为的简化推理。这些系统通常构成了信号处理应用程序的前端。这些系统通常在一组初始滤波器上接收传感器数据;每个过滤器都压缩数据并执行初始过滤。“下游”过滤器进一步减少数据，并对来自不同传感器的数据进行综合。最后一个过滤器通常将其数据传递给应用程序，例如，为建模或可视化工具提供输入。与管道和过滤器系统相关的分析包括派生由图提供的聚合转换关于系统性能的筛选和推理:输入/输出流延迟、管道缓冲区需求和吞吐量。  **调用-返回风格**call-return style  调用返回样式体现了一个计算模型，其中组件提供了一组可由其他组件调用的服务。调用服务暂停(或阻塞)的组件，直到该服务完成。因此，callreturn是编程语言中过程调用的架构类比。连接器负责将服务请求从请求者传递到提供者，并返回任何结果。  回调样式在许多方面都是不同的。有些变体在其连接器的行为方面有所不同。例如，某些调用返回样式中的连接器可能支持错误处理(例如当服务提供者不可用时)。其他差异与拓扑上的约束有关。一些回调架构被组织在层中。其他人将组件集划分为可以发出请求的独立组件集和可以为其提供服务的组件集。调用返回样式的例子包括客户机-服务器、对等和具象状态传输(representational state transfer, REST)样式。  客户端-服务器风格  与所有调用返回样式一样，客户机-服务器样式组件也是如此  通过请求其他组件的服务进行交互。请求程序称为客户机，服务提供者称为服务器，它们通过一个或多个端口提供一组服务。有些组件可以同时充当客户机和服务器。可能有一个中央服务器或多个分布式服务器。典型的客户端-服务器风格的系统包括:  •运行在本地网络上的信息系统客户端是GUI应用程序(如Visual Basic)和服务器是一个数据库管理系统(如Oracle)  •基于Web的应用程序，其中客户端运行在Web浏览器上，服务器是运行在Web服务器(如Tomcat)上的组件。  元素、关系和属性  在表4.2总结的客户机-服务器样式中，组件类型是客户机和服务器。客户机-服务器样式的主要连接器类型是用于调用服务的请求/应答连接器。当可以在同一连接器上请求多个服务时，通常使用协议规范来记录该连接器上可调用服务之间的订购关系。服务器具有描述其提供的服务的端口。客户端有描述他们需要的服务的端口。服务器可以通过从其他服务器请求服务来充当客户端。同时具有服务请求端口和服务应答端口的组件可以同时作为客户端和服务器。  因此，客户机必须知道服务的标识才能调用它，并且客户机启动所有交互。相反，服务器在服务请求之前不知道客户机的身份，必须响应发起的客户机请求。服务调用是同步的:服务的请求者等待或被阻塞，直到被请求的服务完成其操作，可能会提供返回结果。客户机-服务器样式的变体可能会引入其他连接器类型。例如，在一些客户机-服务器样式中，允许服务器对其客户机发起某些操作。这可以通过允许客户端注册服务器在特定时间调用的通知过程或回调来实现。在其他系统中，通过请求/应答连接器进行的服务调用由一个“会话”括起来，该“会话”描述了一组客户机-服务器交互的开始和结束。  客户机-服务器样式的用途  客户机-服务器样式提供了一个系统视图，该视图将客户机应用程序与它们使用的服务分开。这种风格通过分解公共服务来支持系统理解和重用。因为服务器可以被任意数量的客户端访问，所以向系统添加新客户端相对来说比较容易。类似地，可以复制服务器来支持可伸缩性或可用性。  **基于事件的风格** Event-Based Styles  基于事件的样式允许组件通过异步消息进行通信。此类系统通常组织为松散耦合的组件联合，这些组件通过事件触发其他组件中的行为。存在各种各样的事件样式。在某些事件样式中，连接器是点对点的，以类似于调用返回的方式传递消息，但是允许更多的并发性，因为事件发送方在接收方处理事件时不需要阻塞。在其他事件样式中，连接器是多方的，允许将事件发送到多个组件。这样的系统通常称为发布-订阅系统，其中事件通知程序被视为发布由其接收者订阅的事件。  发布-订阅风格  在表4.5中总结的发布-订阅风格中，组件通过发布事件进行交互。组件可以订阅一组事件。发布-订阅运行时基础结构的工作是确保将每个发布的事件交付给该事件的所有订阅者。因此，这种类型的连接器的主要形式是一种事件总线。组件通过通知将事件放置到总线上;连接器然后将这些事件交付给已经注册了对这些事件感兴趣的组件。发布-订阅样式的计算模型最好被认为是一个独立进程系统对象，该对象对其环境生成的事件作出反应，而作为其事件通知的副作用，该对象反过来又会在其他组件中引起反应。采用发布-订阅样式的系统示例如下:  •图形用户界面，用户的低层次输入动作被视为被路由到适当位置的事件输入处理程序  •基于模型-视图-控制器(MVC)的应用程序模式，其中视图组件在状态时得到通知模型对象的  •可扩展的编程环境，其中的工具通过事件协调  •邮件列表，其中一组订阅者可以注册感兴趣的内容在特定的主题  •社交网络，当发生变化时，“朋友”会收到通知出现在某人的Web站点上  **总结：**  这本书书名为<Documenting Software Architectures >，是一本记录软件体系结构的书，里面介绍了很多常用的软件体系结构模型，以及一些相关的模型和衍生模型，内容十分丰富。读完第四章之后，我已经有了不小的收获，对于如何建立自己的软件体系架构也有了几分把握。  这本书从各个方面介绍了常见软件体系风格，还援引多个范例来进行解说，方便人们能更加深刻的掌握这些架构，从而对自己的软件开发产生效益。  **架构之美学习报告（史子涵）**  《架构之美》这本书应该不是专门为软件架构而设的，该书的第一部分为论架构，通过实例讲述了架构的概念，然后开始涉及到各种不同的架构，1是企业级应用架构，主要讲述了伸缩性架构设计、记忆留存、面像资源的架构：在web中、数据增长：Facebook平台的架构。2是系统架构，讲述了Xen和虚拟化之美、guardian:一个容错操作系统环境、JPC：一个纯Java的x86 PC模拟程序、元循环虚拟机的力量：Jikes RVM。3是最终用户架构。4是语言与架构，这一部分才讲到了软件架构。  讲述了软件架构的两种方式：面向函数和面向对象。原来我以为软件架构和软件体系结构是一回事，现在发现虽然他们的英文翻译是相同的，但是有很大不同，例如软件体系结构的各种风格例如分层、客户机服务器等在书中都没有提及。网上有的回复有的认为软件体系结构是软件架构的一部分。  讲软件架构时，两种方式的描述思路都是相同的。先讲述方法，之后讲述方法的模块性评价。提出了软件架构美的定义：可靠性、可扩展性、可复用性。  书中通过用布丁问题代替“金融契约来”作为函数式软件架构示例，所形成的的树形结构和函数关系式如下所示：    函数式解决方案的模块性评价：从可扩展性来说，函数式方法对于架构所产生的直接好处就是添加新的合成方式简单；从函数式方式来说，其提供的程序结构很简单，是由一组定义组成的；函数式方法的四个重要优点便是1.表示法：定义简洁；2时表示法中定义对象的组合子表达式优雅；3是能够被高效处理；4是延迟计算，可以描述一个可能无限大的计算。  面向对象很好地利用我们编程语言中面像对象的概念，涉及到类的继承性和多态性。其模块化评价中由于继承使得可复用性好，对于可扩展性，表现在添加新的类型中。  表明了面向对象的设计，包含了函数式编程，在保留优点的同时又提供了更高层的抽象，更好的支持扩展和复用。  **Software Architecture in Practice学习报告（陈文盼）**  如何设计一个好的体系架构（过程方面）   1. 该架构应该是由一位技术负责人组成的架构师小组中的单个架构师或小型架构师的产品。这种方法使体系结构具有概念完整性和技术一致性。这个建议除了“传统”的架构同样也适用于敏捷和开源项目。这是一个在架构师和开发团队间强的联系，可以避免象牙塔设计不切实际。 2. 架构师（或架构团队）应在持续的基础上，将架构建立在一个明确指定的质量属性要求的优先列表上。这将为经常发生的权衡提供信息。 3. 架构应该使用视图进行记录。这个视图能支持项目时间表，能解决最重要利益人员的担忧。 4. 这个架构应该被评估传递系统重要质量属性的能力。这应该在生命周期的早期发生，当它返回最大的利益时，并在适当的时候重复，以确保对架构（或其预期的环境）的更改没有使设计过时。 5. 该体系结构应该适合于增量实现，以避免一次集成所有内容（几乎不起作用）以及及早发现问题。一种方法是创建一个“骨架”一种系统，在该系统中，通信路径被执行，但最初只有最小的功能。这个框架系统可以用于增量地“增长”系统，并根据需要进行重构。   **聊架构：洞察架构学习报告（郑晓旭）**  阐述了生命周期的概念：生命周期是一种行为或者一个事物从产生到消亡的全过程。  生命周期可以拆分成子生命周期：购买商品可以拆分成选购和购买两个过程。  核心生命周期和非核心生命周期：在周期的树上，位于主干节点的为核心生命周期。  架构的意义：以种粮吃饭为例。在分工前，每个人需要自行种粮、烧饭、吃饭。在分工后，可以请人种粮，请人烧饭，自己只需要完成核心生命周期——吃饭即可。  什么是架构：能够解决的人类问题，对目标的边界进行明确确定。能围绕目标生命周期进行分割。  认识概念是理解架构的基础。要做好架构首先必须具备的能力是能够正确的认识概念，能够发现概念背后所代表的问题，进而才能够认识目标领域所需要解决的问题。比如，什么是“杯子”？杯子这个概念并不是瓷器，回归作用：其实是为了解决“人需要一个可单手持握，但是希望避免直接接触所盛物体”这个问题。  如何做好架构？  第一、识别问题，找到问题的主体  如果把真正的问题找到，那么问题就已经解决了 80% 了。这个能力基本上就决定了架构师的水平。  找出问题的主体，是做架构的首要问题。我们要解决的问题，一定都是人的问题。更进一步，作为软件工程师或者架构师，我们大部分时候是要去解决别人的问题，“别人”是谁，是值得好好思考的。再进一步，我们一定要明白，任何找上架构师的问题，绝对都不是真正的问题。为什么呢? 因为如果是真正的问题的话，提问题过来的人肯定都能够自己解决了，不需要找架构师。架构师都要有这个自觉：发现问题永远都比解决问题来的更加重要。  一般来说，从问题暴露的点，一点点去溯源查找，一定会找出来谁的问题，以及是什么问题。最坏情况就是当我们时间或者能力有限，实在是无法定位出是谁的问题的时候，比如系统出故障，也就意味着我们无法根本解决问题。这时最好的办法就是去降低问题发生所带来的成本，尽量去隔离问题影响的范围。给我留出时间和空间去识别真正的问题。  总结一下，要正确的认识问题，需要问两个问题：  1、这是谁的问题？  2、有什么问题？  软件的本质，其实就是通过把人类的日常工作生活虚拟化，减少成本，提升单个人员的生产力，提升人类自己的利益。 随着软件的规模的变大，程序从早期由一个人完成，也逐渐变成了由很多不同角色的人共同合作来完成。  软件架构到底是要解决什么问题？  软件实际上就是把现实生活模拟到计算机中，并且软件是需要在计算机的硬件中运行起来的。要做到这一点需要解决两个问题：  一、业务问题  具体的现实生活状态下，没有软件的时候，所解决的问题的主体是谁，解决的是什么问题，是如何解决，如何运作的？  二、计算机问题  如何把现实生活用软件来模拟？模拟出来的软件，需要哪些硬件设施才能够满足要求？ 并且当访问量越来越大的时候，软件能否支持硬件慢慢长大，性能线性扩展？因为硬件是可能会失效的，软件如何在硬件失效的情况下，仍然能够保证可用性，让用户能够不中断的访问软件提供的服务？怎么收集软件产生的数据，为下一阶段的工作提供依据？  总之，软件架构包括了：代码架构，以及承载代码运行的硬件部署架构。实际上，硬件部署架构最终还是由代码的架构来决定。  架构师必须是一个组织的领导人，有权利调动这个组织的架构，才能够更好的发挥架构师的作用，更好的把利益的调整落到实处。 | | |
| 结论分析与体会：  本周的实验重点在于参考书的阅读和SAD初稿的设计。通过对老师推荐书目的阅读，我们写了读书笔记，并分享了自己的阅读感悟。计算机领域本来就是一个后浪推前浪、不断更新的领域，所以我们不能闭门造车，要善于思考善于学习。同时，这也是一个为其他领域服务的基础性学科，所以我们也需要了解其他领域的一定量的知识。 | | |