第一部分

论架构

第 1 章

架构概述

John Klein
David Weiss

1.1 简介

建筑师、音乐家、作家、计算机设计师、网络设计师和软件开发者都在使用“架构”这个术语，其他人也用（你有没有听说过“食物架构”？），然而不同的用法其结果也不同。建筑与交响乐完全不同，但都有架构。而且，所有的架构师都在谈论他们工作中的美，以及因此而导致的结果。建筑师可能会说，一座建筑应该提供适合工作或生活的环境，而且它应该看起来很美。音乐家可能会说，音乐应该能演奏，包含能够辨明的主题，而且它应该听起来很美。软件架构师可能会说，系统应该对用户友好、响应及时、可维护、没有重大错误、易于安装、可靠，应该通过标准的方式与其他系统通信，而且也应该是美的。

这本书为你提供了一些美丽架构的详细例子，它们来自于各类计算机系统。相对来说，计算机是比较年轻的一个学科。因为年轻，所以不像建筑、音乐或写作等领域那样，有那么多的例子；也因为年轻，则需要更多的例子。我们希望这本书能满足这种需要。

在你开始研究这些例子之前，我们希望你考虑以下两个问题：1) 什么是架构？2) 美丽的架构都有哪些特性？你会在这一章中看到架构的不同定义，每个学科都有自己的定义，所以我们将首先探讨不同学科中的架构有何共同点，以及人们试图用架构解决哪些问题。具体来说，架构有助于确保系统能够满足其利益相关人的关注点，在构想、计划、构建和维护系统时，架构有助于处理复杂性。

然后我们将介绍架构的定义，展示如何将这个定义应用于软件架构，因为软件是本书后面大部分例子关注的核心。这个定义的关键在于，架构由一组结构组成，这些结构的设计目的是让架构师、构建者，以及其他利益相关人看到他们的关注点是如何得到满足的。

在本章末尾，我们将讨论美丽架构的特性，并引用一些例子。美的核心在于概念完整性——即一组抽象和规则，在整个系统中尽可能简单地应用它们。

在讨论中，我们将“架构”作为一个名词，它意味着一组工件，包括像蓝图和构建规范这样的文档。这些工件描述了要构建的对象，在这种描述中，该对象被视为一组结构。某些人也把“架构”作为一个动词，用来描述创建这些工件的过程，包括由此而导致的工作。然而，正如Jim Waldo和其他人曾指出的，没有什么过程可以保证你学了以后就能创造出好的系统架构，更不必说美的架构了（Waldo 2006），所以我们将更关注工件，而非过程。

> 架构：“建造的艺术或科学；特别是设计和建造人类使用的建筑时的艺术或实践，同时考虑到美学因素和实用因素。”
>
> ——《The Shorter Oxford English Dictionary》（小型牛津英语字典，第5版）

在所有学科中，架构都提供了一种方式来解决共同的问题：确保建筑、桥梁、乐曲、书籍、计算机、网络或系统在完成后具有某些属性或行为。换言之，架构既是所构建系统的计划，确保由此得到期望的特性，同时也是所构建系统的描述。维基百科上说：“根据这方面已知最早的著作，即Vitruvius的‘On Architecture’，好的建筑应该美观（Venustas）、坚固（Firmitas）、实用（Utilitas）；架构可以说是这三方面的一种平衡和配合，没有哪一个方面比其他方面更重要。”

> 我们谈到交响乐的“架构（architecture）”，反过来，又将架构（architecture）称为“凝固的音乐”。
>
> —— Deryck Cooke,《The Language of Music》（音乐的语言）

好的系统架构展示了架构完整性。也就是说，它来自于一组设计规则，这组规则有助于减少复杂性，并可以用于指导详细设计和系统验证。设计规则可能包含特定的抽象，这些抽象总是以同样的方式使用，诸如虚拟设备等。这些规则可能表现为一种模式，如管道和过滤器。在最理想的情况下，存在一些可以用于验证的规则，如“在设备失效时，所有某一类的虚拟设备都可以用任何其他同类的虚拟设备代替”，或“所有竞争同一资源的进程必须具有相同的调度优先级”。

当代的架构师可能会说，待构建的对象或系统必须具有以下特征：

- 具备客户要求的功能。
- 能够在要求的工期内安全地构建。

- 性能足够好。
- 可靠的。
- 可用的，并且使用时不会造成伤害。
- 安全的。
- 成本是可以接受的。
- 符合法规标准。
- 将超越前人及其竞争者。

> 我们将计算机系统的架构定义为一组最小的特征集，它们决定了哪些程序将运行，以及这些程序将得到什么结果。
>
> —— Gerrit Blaauw 和Frederick Brooks,《Computer Architecture》（计算机体系结构）

我们从来没有看到过一个复杂系统能够很好地满足上述特征。架构是一种折中 —— 决定改进其中一个特征常常会对其他特征产生负面影响。架构师必须确定怎样做是足够好的，方法就是发现特定系统的重要关注点，以及充分满足这些关注点的条件。

架构观点中的常见思想是结构，每种结构都由各种类型的组件及其关系构成：它们如何组合、相互调用、通信、同步，以及进行其他交互。组件可以是建筑中的支架横梁或内部腔室、交响乐中的旋律、故事中的章节或人物、计算机中的CPU和内存、通信栈中的层或连接到一个网络上的处理器、协作的顺序过程、对象、编译时的宏、构建时的脚本。每个学科都有自己的一套组件和组件间的相互关系。

> 从更大的范围来说，术语“ 架构 ”总是意味着“ 不变的深层次结构 ”。
>
> —— Stewart Brand,《How Buildings Learn》

面对不断增长的系统复杂性，以及它们内部和相互之间的交互，由一组结构形成的架构提供了对付复杂性的主要手段，目的是确保得到的系统具备所要求的特征。结构为我们提供途径，将系统化解为一些交互的组件。

每种结构都是为了帮助架构师理解如何来满足特定的关注点，如可变性或性能。展示某些关注点得到满足时，可能会影响到其他方面的关注点，但架构师必须能够说明所有关注点都已得到满足。

> 网络架构：构成一个网络的通信设备、协议和传输链路，以及它们的组织方式。
>
> —— *http://www.wtcs.org/snmp4tpc/jton.htm*

1.1.1 建筑师的角色

在设计、构建和修复建筑时，我们指定关键的设计师为“建筑师（architects）”，并赋予他们广泛的职责。建筑师准备建筑最初的草图，展示外观和内部布局，与客户讨论这些草图，直至所有相关方都达成一致意见，认为展示的就是他们想要的。这些草图是抽象：它们关注建筑中某些方面的适当细节，而忽略其他的内容。

当客户和建筑师在这些抽象上达成一致意见之后，建筑师会准备或监督准备更为详细的图纸，以及相关的文字规格说明。这些图纸和规格说明描述了建筑的许多“实质性”细节，如管道、壁板材料、窗户玻璃和电线等。

在极少的情况下，建筑师简单地将详细规划交给建造者，建造者将根据规划完成项目。对更重要一些的项目，建筑师会继续参与，定期检查工作，并且可能会建议变更，或接受来自建造者和客户的变更建议。如果建筑师监督项目，仅当他确认项目充分符合了规划和规格说明的要求，项目才算完工。

我们请一名建筑师是为了确保：1）设计满足客户的需要，包括前面提到的那些特征；2）设计具有概念完整性，处处运用了相同的设计原则；3）设计满足法规和安全的要求。建筑师职责的一个重要方面是确保设计概念在实现时得到一致的体现。有时候，建筑师也充当建造者和客户之间的协调人。哪些决定需要由建筑师做出，哪些决定由其他人做出，人们对这个问题常有不同意见，但我们清楚，建筑师将做出重要决定，包括所有对结构的可用性、安全性和可维护性产生影响的那些决定。

音乐作曲与软件架构

虽然人们常用建筑架构设计来类比软件架构，但音乐作曲可能是更好的类比。建筑师创建的是相对静止的结构（该架构必须考虑到人员和服务在建筑内的移动，以及承重结构）的静态描述（蓝图或其他图纸）。在音乐作曲和软件设计中，作曲家（软件架构师）创建一段音乐的静态描述（架构描述和代码），这段音乐以后将演奏（执行）许多次。在音乐和软件中，设计都依靠许多组件的交互来得到期望的结果，结果依赖于演奏者、演奏环境，以及演奏者所做的诠释。

1.1.2 软件架构师的角色

软件开发项目需要一些人在软件构建时扮演架构师的角色，就像构建或修复建筑时传统的建筑师的角色一样。但是，对于软件系统来说，从来就弄不清楚哪些决定属于架构师的职责范围，哪些决定要留给实现者。定义架构师在软件项目中做什么，比建筑师的类

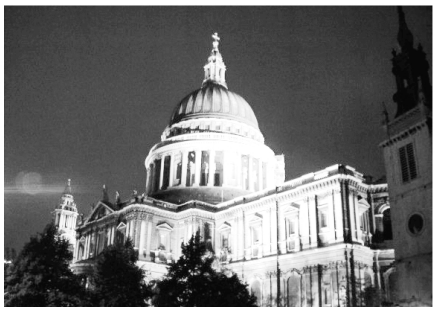
似定义更困难，原因有3个因素：缺少传统、产品无形性和系统复杂性。（参见Grinter[1999]，其中描述了软件架构师如何在一个大型软件开发组织中实现她的职责。）

具体来说：

- 建筑师可以回顾几千年的历史，看看过去的建筑师都做过些什么。他们可以参观并研究那些矗立了几百年的建筑，有时甚至有上千年历史的建筑，而它们仍在使用。在软件业，我们只有几十年的历史，并且我们的设计常常是不公开的。此外，建筑师拥有并利用标准来描述他们制作的图纸和规格说明，这让现在的建筑师能够从记录下来的架构历史中受益。
- 建筑是有形的产品，在建筑师制作的规划和工人修造的建筑之间存在着明显的区别。

架构复用

圣索菲亚大教堂（Hagia Sophia，上图），建造于公元6世纪，率先使用了所谓的“穹顶”结构来支撑巨大的圆形屋顶，它是拜占庭建筑之美的代表。在1100年之后，Christopher Wren使用了同样的设计来建造圣保罗大教堂的穹顶（St. Paul’s Cathedral，下图），它成为伦敦的地标性建筑。这两座建筑在今天仍在使用。



在大的软件项目中，常常会有许多架构师。某些架构师相当专注于特定领域，如数据库和网络，他们一般作为团队的一部分，但目前我们假定只有唯一一位架构师。

1.1.3 软件架构的含义

如果认为“架构”是一个简单的实体，能够用一份文档或一张图纸来描述，那就错了。架构师必须做出许多设计决定。要想有用，这些决定必须用文档记录下来，这样就能够进行复审、讨论、修改和批准，然后作为后续决定和构建时的约束。对于软件系统，这些设计决定包括行为上的和结构上的。

外部行为描述展示了产品如何与它的用户、其他系统和外部设备进行交互，这应该表现为需求。结构描述展示了产品如何划分为多个部分，以及这些部分之间的关系。我们还需要内部行为描述，用于描述组件之间的交互接口。结构上的描述常常展示相同部分的一些不同视图，因为不可能把所有信息以有意义的方式组织到一张图纸或一份文档中。一个视图中的组件，可能是另一个视图中一个组件的一个部分。

软件架构常常表现为分层的层次结构，这种层次结构将几种不同的结构放在一张图中。20世纪70年代，Parnas指出“层次结构”这个术语已经被滥用，然后精确地定义了它，并给出了几个不同结构的例子，它们在设计不同系统时实现了不同的目的（Parnas 1974）。将架构的结构描述为一组视图（view），每个视图关注不同的部分，现在已成为了广泛接受的标准架构实践（Clements等 2003; IEEE 2000）。我们将使用“架构”这个词来代指一组有标注的图纸和功能描述，它说明了设计和构建一个系统时所使用的结构。在软件开发社区中，针对这样的图纸和描述，人们使用并建议了许多不同的形式。在Hoffman和Weiss（2000，第14章和第16章）的著作中可以看到一些例子。

> 一个程序或计算系统的软件架构是系统的一种结构或一组结构，它包含软件元素、这些元素的外部可见的属性，以及元素之间的关系。
>
> “外部可见”的属性是其他元素对该元素可以做出的假定，诸如它提供的服务、执行时的特征、错误处理、共享资源的使用等。
>
> —— Len Bass、Paul Clements和Rick Kazman
>
> 《Software Architecture in Practice, Second Edition》

1.1.4 架构与设计

架构是系统设计的一部分，它突出了某些细节，并通过抽象省略掉另一些细节。所以，架构是设计的一个子集。关注实现系统组件的开发者可能不会特别关心所有组件如何装配在一起，而是主要关注少数组件的设计和开发，包括他们必须遵守的架构约束和可以应用的规则。因此，开发者和架构师面对的是系统设计的不同方面。

如果说架构关注的是组件之间的关系和系统组件外部可见的属性，那么设计还要关注这些组件的内部结构。例如，如果一组组件包含了一些信息隐藏的模块，那么这些外部可见的属性就构成了这些组件的接口，内部的结构与模块内的数据结构和控制流一同考虑（Hoffman和Weiss 2000，第7章和第16章）。

1.2 创建软件架构

到目前为止，我们已经讨论了一般意义上的架构，并分析了软件架构与其他领域的架构之间有何相似与差异。接下来我们将注意力转到“如何”设计软件架构。当架构师创建软件系统的架构时，她应该关注什么？

软件架构师的首要关注点不是系统的功能。

这是正确的——软件架构师的首要关注点不是系统的功能。

例如，如果我们请你来设计一个“基于Web的应用”，你首先问我们页面布局和导航树，还是问下面这些问题：

- 谁提供应用主机托管？托管的环境有什么技术限制吗？
- 你想运行在Windows服务器上还是在LAMP栈上？
- 你想支持多少并发用户？
- 应用需要怎样的安全性？有需要保护的数据吗？应用将运行在公网上还是在私有的内部网上？
- 你能为这些答案排列优先级吗？例如，用户数是否比响应时间更重要？

根据我们对这些问题和一些其他问题的回答，你就可以开始画出系统架构的草图。我们还没有谈到应用的功能。

好吧，我们承认耍了点计谋，因为我们问的是“基于Web的应用”，这是一个大家熟悉的领域，所以你已经知道了哪些决定会对你的架构产生最大的影响。类似地，如果我们问的是一个电信系统或一个航空电子控制系统，在这些领域有经验的架构师将考虑到一些功能需求。但是，你仍然可以不必过多担心功能就开始设计架构。你关注的是需要满足的品质。

品质关注点指明了功能必须以何种方式交付，才能被系统的利益相关人所接受，系统的结果包含这些人的既定利益。利益相关人有一些关注点，架构师必须重视。稍后，我们将讨论为了确保系统具有要求的品质，通常会提出的一些关注点。正如我们前面所说的，架构师的一项职责是确保系统设计能满足客户的需要，我们将利用品质关注点来帮助我们理解这些需要。

这个例子突出了成功架构师的两项关键实践：让利益相关人参与以及同时关注功能和品质。作为一名架构师，你首先问我们想从系统中得到什么，有怎样的优先级。在实际项目中，你会找出其他的利益相关人。典型的利益相关人和他们的关注点包括：

- 投资人，他们想知道项目是否能够在给定的资源和进度约束下完成。
- 架构师、开发人员和测试人员，他们首先考虑的是最初的构建和以后的维护与演进。
- 项目经理，他们需要组织团队，制定迭代计划。
- 市场人员，他们想通过品质特点实现与竞争者的差异化。
- 用户，包括最终用户、系统管理员，以及安装、部署、准备、配置人员。
- 技术支持人员，他们关注帮助平台电话呼入的数目和复杂性。

每个系统都有自己的品质关注点。有些关注点可能定义得很好，如性能、安全、可伸缩性等。但是，另一些同样重要的关注点却可能没有详细规定，如可变性、可维护性和可用性等。利益相关人希望把功能放到软件上，而不是放到硬件上，这主要是为了很容易、很快速地修改，然后通常在品质关注方面又对可变性轻描淡写。这很奇怪，不是吗？哪些改变能够迅速、容易地实现，哪些改变需要花时间并且很难实现，架构决定将对此产生重要影响。所以，架构师难道不应该在理解功能需求的同时，也理解利益相关人在“可变性”这样的品质方面的期望吗？

当架构师理解了利益相关人的品质关注点之后，接下来该做些什么？考虑折中。例如，对信息加密将加强安全性，但会损失性能。利用配置文件将增加可变性，但会降低可用性，除非我们能够验证配置是有效的。我们是否应该对这些文件使用标准的表示方式，如XML，还是使用自己发明的格式？创建系统的架构将涉及许多这样的艰难折中。

架构师的第一项任务，就是与利益相关人协作，理解这些品质关注点和约束，并为它们排列优先级。为什么不从功能需求开始？因为通常有许多种可能的系统分解方式。例如，从数据模型开始可能得到一种架构，而从业务处理模型开始则可能得到不同的架构。在极端的情况下，系统没有分解，被开发成单一的软件。这可能会满足所有的功能需求，但可能不会满足品质需求，如可变性、可维护性、可伸缩性等。架构师通常必须进行架构层面的系统重构，例如为了满足伸缩性或性能的要求，将单机部署迁移到分布式部署，从单线程转向多线程，或者将硬编码的参数移到外部配置文件中，因为原来从不改变的参数现在需要修改了。

尽管有许多架构都能满足功能需求，其中却只有一少部分能够满足品质需求。让我们回到Web应用的例子。请考虑提供Web页面的诸多方式 —— Apache和静态页面、CGI、Servlet、JSP、JSF、PHP、Ruby on Rails、ASP.NET等。选择其中的一种技术是一种架构决定，它将对你满足特定品质需求的能力产生重要影响。例如，像Ruby on Rails这样

的方式可能提供快速推向市场的好处，但可能更难维护，因为Ruby语言和Rails框架都在不断地快速发展。也许我们的应用是基于Web的电话，我们需要让电话“响铃”。如果你为了满足性能的要求，需要从服务器向Web页面发出真正异步的事件，那么基于Servlet的架构可能更容易测试和修改。

在真实的项目中，满足利益相关人的关注点需要做出更多的决定，而不仅是选择一个Web框架。你是否真的需要一个“架构”，并需要一名“架构师”来做出这些决定？谁将做出这些决定？是编程人员吗？他们可能会做出许多无意识的、隐含的决定。还是由架构师来做出这些决定？他们全面了解整个系统、利益相关人和系统的演进，然后做出明确的决定。不论哪种方式，你会有一个架构。它是否应该明确地形成并记入文档？或者它应该是隐式的，需要通过阅读代码才能发现？

当然，这种选择通常不是这么死板。但是，随着系统的规模、复杂度和开发人员数目的增长，这些早期决定以及它们的记录方式将产生越来越大的影响。

我们希望你现在已经理解，如果你的系统要满足其品质要求，架构决定是很重要的，你需要注意架构，有意识地做出这些决定，而不只是“让架构自动出现”。

如果系统非常大，情况会怎样？我们之所以运用“分而治之”这样的架构原则，一个原因就是为了降低复杂性，让工作能够并发进行。这让我们能够创建越来越大的系统。架构本身是否能够分解为多个部分，这些部分是否能由不同的人并行开发？考虑到计算机的架构，Gerrit Blaauw和Fred Brooks断定：

> ……如果，在采取了所有让任务能够由单人处理的方法之后，架构任务仍然巨大而复杂，不能由一人来完成，那么产品肯定是太复杂了，以致不实用且不应构建。换言之，单个用户必须能够理解计算机的架构。如果计划的架构不能由一个人设计，那它也不能被一个人理解。（1997）

你是否需要理解架构的所有方面，才能使用它？架构会分离关注点，所以在大多数情况下，利用架构来构建或维护系统的开发人员或测试人员，不需要一下面对全部的架构，而是只要面对必要的部分，就能完成指定的功能。这让我们能够创建超出个人可以理解的、更大的系统。但是，在我们完全忽略IBM System/360（最长寿的计算机架构之一）创造者的建议之前，让我们先来看看他们为什么这样说。

Fred Brooks说，概念完整性是架构最重要的特征：“最好是让系统……反映一组设计思想，而不是让系统包含许多好的思想，而这些思想却彼此独立而不协调”（1995）。正是这种概念完整性，让开发者在知道了系统的一部分之后，能够迅速理解系统的另一部分。概念完整性来自于处理问题的一致性，如分解的判据、设计模式的应用和数据模式。这让开发者运用在系统中的一部分工作的经验，来开发和维护系统的其他部分。同样的规

则应用于整个系统各处。当我们转向“众系统之系统”时，在集成了这些系统的架构中也必须保持概念完整性。例如，可以选择发布/订阅消息总线这样的架构风格，然后将这种风格统一地应用于“众系统之系统”的系统集成中。

架构团队的挑战在于，在创建架构时保持同一种思考方式和同一种哲学。让团队保持尽可能小，让他们在充分沟通、高度协作的环境工作，让一两个“首席架构师”担任仁慈的独裁者，最终做出所有决定。这种架构模式常见于成功的团队，不论是公司开发还是开源开发，由此而得到的概念完整性是美丽架构的一种特性。

好的架构师通常来自于更好的架构师提供的现场指导（Waldo 2006）。原因之一可能是有一些关注点几乎在所有项目中都会出现。我们已经提到过一些，但这里有一份更完整的清单。每个关注点都以问题的方式表述，架构师在项目过程中可能需要考虑它。当然，具体系统会有其他关键的关注点。

功能性（Functionality）

产品向它的用户提供哪些功能？

可变性（Changeability）

软件将来可能需要哪些改变？哪些改变不太可能发生，不需要特别容易进行这些改变？

性能（Performance）

产品将达到怎样的性能？

容量（Capacity）

多少用户将并发使用该系统？该系统将为用户保存多少数据？

生态系统（Ecosystem）

在部署的生态环境中，该系统将与其他系统进行哪些交互？

模块化（Modularity）

如何将编写软件的任务分解为工作指派（模块），特别是这些模块可以独立地开发，并能够准确而容易地满足彼此的需要？

可构建性（Buildability）

如何将软件构建为一组组件，并能够独立实现和验证这些组件？哪些组件应该复用其他的产品，哪些应该从外部供应商处获得？

产品化（Producibility）

如果产品将以几种变体的形式存在，如何开发一个产品线，并利用这些变体的共性？产品线中的产品以怎样的步骤开发（Weiss和Lai 1999）？在创建一条软件产品线时，要进行哪些投资？开发产品线中不同变体的选择，预期会得到怎样的回报？

特别是，是否可能先开发最小的有用产品，然后再添加（扩展）组件，在不改变以前编写的代码的情况下，开发产品线的其他成员？

安全性（Security）

产品是否需要用户认证，或者必须限制对数据的访问？数据的安全性如何得到保证？如何抵挡“拒绝服务”攻击或其他攻击？

最后，一个好的架构师会认识到，架构会影响组织机构。Conway指出，系统的结构会反映构建它的组织机构的结构（1968）。架构师可能会认识到，Conway法则可以反过来应用。换言之，一个好的架构可能对组织机构产生影响，让组织机构发生改变，从而更有效地从该架构构建出系统。

1.3 架构结构

那么，好的架构师如何来处理这些关注点？我们曾经提到过，需要将系统组织成一些结构，每种结构都定义了特定类型的组件之间的具体关系。架构师的主要关注点就是对系统进行组织，让每种结构有助于解答一个关注点所定义的问题。关键的结构决定将产品划分为组件，并定义了这些组件之间的关系（Bass、Clements和Kazman 2003; Booch、 Rumbaugh和Jacobson 1999; IEEE 2000; Garlan和Perry 1995）。对于任何产品，都有许多结构需要设计。每种结构都必须单独设计，这样它就表现为一个独立的关注点。在接下来的几节中我们会讨论一些结构，你可以利用它们来考虑前面列表中的关注点。例如，“信息隐藏结构”展示了如何将系统组织成一些工作指派。这种结构也可以用作改变的路线图，展示了建议的改变，以及哪些模块支持这些改变。针对每种结构，我们描述了一些组件及其之间的关系，正是这些组件和关系确定了这种结构。对照前面的列表，我们认为下面的结构是最重要的。

1.3.1 信息隐藏结构

组件与关系：主要组件是一些“信息隐藏模块”，每个模块都是针对一组开发人员的工作指派，每个模块都包含了一种设计决定。如果一项决定可以改变，同时又不影响任何其他模块，我们就说这项设计决定是一个模块的秘密（Hoffman和Weiss 2000，第7章和第16章）。模块间最基本的关系是“整体-部分”关系。如果“信息隐藏模块A”的秘密是“信息隐藏模块B”的秘密的一部分，那么A就是B的一部分。请注意，必须能够在改变A的秘密的同时，不改变B的其他部分。否则，根据我们的定义，A就不是B的一个子模块。例如，许多架构都有一些虚拟设备模块，它们的秘密是如何与特定的物理设备通

信。如果虚拟设备分成不同类型，那么每种类型可能构成该虚拟设备模块的一个子模块，其中每种虚拟设备类型的秘密将是如何与这种类型的设备进行通信。

每个模块都是一份工作指派，包含了一组要写的程序。根据不同的语言、平台、环境，“程序”可以是能在计算机上执行的方法、过程、函数、子程序、脚本、宏或其他指令序列。第二种信息隐藏模块结构是基于程序和模块之间的“包含”关系。如果模块M的一部分工作指派是要编写程序P，那么M就包含P。请注意，每个程序都包含在一个模块中，因为每个程序必然是某些开发人员的工作指派的一部分。

这些程序中的一些可以通过模块的接口来访问，而另一些则是内部的。模块也可能通过接口发生关系。A模块的接口是一组假定，这些假定包括该模块之外的程序可以对该模块做出的假定，也包括该模块中的程序对其他模块的程序和数据结构所做的假定。如果改变B的接口就要求A也发生改变，那么我们就说A“依赖”B的接口。

“整体-部分”结构是层次状的。在这个层次结构的叶节点上的模块不包含可识别的子模块。“包含”结构也是层次状的，因为每个程序都只包含在一个模块之中。“依赖”关系不一定是层次状的，因为两个模块可能互相依赖，要么是直接互相依赖，要么是通过一个较长的“依赖”关系形成的环。请注意，“依赖”不应该与后面小节中定义的“使用”混淆起来。

信息隐藏结构是面向对象设计方法的基础。如果一个信息隐藏模块设计为一个类，这个类的公有方法就属于该模块的接口。

满足的关注点：信息隐藏结构的设计应该能满足可变性、模块化和可构建性的要求。

1.3.2 使用结构

组件与关系：根据前面我们的定义，信息隐藏模块包含一个或多个程序（在上一小节中定义）。当且仅当两个程序共享一个秘密时，它们才属于同一个模块。“使用结构”（Uses Structure）的组件是一些可以单独调用的程序。请注意，程序可以相互调用，或被硬件调用（例如，被一个中断例程调用），调用也可能来自于不同命名空间的程序，如操作系统例程或远程过程。而且，调用发生的时间可以是任何时候，从编译时到运行时。

只有在相同绑定时间操作的程序之间，我们才考虑形成一种使用结构。首先只考虑运行时操作的程序可能最容易。以后，我们也可以考虑那些编译时或载入时操作的程序之间的使用关系。

如果程序B必须存在并满足其规格说明，程序A才能满足其规格说明，我们就说A使用了B。换言之，B必须存在且操作正常，A才能操作正常。使用关系有时候也称为“要求存在正确的版本”。进一步的解释和例子，参见（Hoffman和Weiss 2000）的第14章。

使用结构确定了我们可以构建并测试怎样的工作子集。在软件系统的使用结构中，期望的属性是它定义了一种层次结构，这意味着其中不出现环。如果在使用关系中出现环，那么环中所有程序都必须存在且正常工作，才能让其他的程序正常工作。由于也许不能够创建完全没有环的使用关系，架构师可能将使用环中的所有程序作为单一的程序，以这种方法来创建子集。子集必须要么包含全部程序，要么都不包含。

如果在使用关系中没有环，软件采用的就是一种层次结构。在最底层，即第0层，是所有不使用其他程序的程序。第n层包含了所有的程序，它们使用了第n - 1层或以下层的程序。这些层常常描绘为一系列的层次，每个层次表示了使用关系中的一个或几个层。在使用结构中对相邻的层分组，有助于简化表示，并允许在关系中出现小环的情况。进行这种分组有一个指导原则，即一个层次中的程序应该比它上一个层次中的程序执行速度快9倍，执行频率高9倍（Courtois 1977）。

具有层次使用结构的系统可以同时构造一层或几层。这些层次有时候称为“抽象层”，但这是一种错误的名称。因为这些组件是独立的程序，而不是完整的模块，它们不一定抽象（隐藏）了什么东西。

通常大型的软件系统包含太多的程序，这让程序间使用关系的描述不太容易理解。在这种情况下，使用关系可以用于程序的组合，如模块、类或包。这样的组合描述丧失了重要的信息，但有助于展示“全局”。例如，你有时候可以在信息隐藏模块之间建立使用关系，但是除非一个模块中所有的程序都属于实际使用层次的同一层，否则就会丧失重要的信息。

在某些项目中，系统的使用关系开始并没有完全确定，要到系统实现时才能确定，因为开发者会在实现过程中决定他们使用哪些程序。但是，系统的架构师可能在设计时创建一种“允许使用”关系，约束开发者的选择。今后，我们不会区分“使用”和“允许使用”。

定义良好的使用结构将创建系统的适当子集，可以用于驱动迭代式或增量式的开发循环。

满足的关注点：产品化和生态系统。

1.3.3 进程结构

组件与关系：信息隐藏模块结构和使用结构是静态的结构，存在于设计时和编码时。我们现在转向运行时结构。参与进程结构的组件是进程。进程是运行时的事件序列，由程序控制（Dijkstra 1968）。每个程序都作为一个或多个进程的一部分执行。一个进程中的事件序列的执行独立于另一进程中的事件序列，除非这两个进程彼此同步，例如一个进程等待来自另一个进程的信号或消息。进程由支持系统分配资源，包括内存和处理器时间。系统可能包含固定数量的进程，也可能在运行时创建和销毁进程。请注意，在

Linux和Windows操作系统中实现的线程也符合这个进程定义。进程是几种不同关系中的组件。下面是一些例子。

进程提供工作

一个进程可能会创建工作，该项工作必须由其他进程完成。这种结构在确定系统是否死锁时是很重要的。

满足的关注点：性能和容量。

进程取得资源

在动态分配资源的系统中，一个进程可能控制由另一个进程使用的资源，后者必须请求并归还这些资源。因为发起请求的进程可能从几个控制器那里请求资源，所以每项资源可能都有一个不同的控制进程。

满足的关注点：性能和容量。

进程共享资源

两个进程可能共享资源，如打印机、内存或端口等。如果两个进程共享一项资源，就需要通过同步来防止使用冲突。每一种资源可能有不同的关系。

满足的关注点：性能和容量。

进程包含在模块中

每个进程由一个程序控制，正如前面提到的，每个程序包含在一个模块之中。因此，我们可以认为进程包含在模块之中。

满足的关注点：性能和容量。

1.3.4 访问结构

系统中的数据可能划分成具有属性的段，如果程序对段中的任何数据拥有访问权，就对该段中的所有数据拥有了访问权。请注意，为了简化描述，我们应该让段的规模最大化，具体做法是添加一个条件，即如果两个段被同一组程序访问，这两个段就应该合并。数据访问结构包含两种类型的组件：程序和段。这种关系被命名“ 有权访问 ”，它是程序和数据段之间的关系。如果这种结构让程序访问的权限最小化，并且严格执行，我们就认为系统更安全。

满足的关注点：安全性。

1.3.5 结构小结

表1-1总结了前面的软件结构，包括它们的定义和它们满足的关注点。

表1-1：结构小结

结构	组件	关系	关注点
信息隐藏	信息隐藏模块	整体 - 部分 包含	可变性 模块化 可构建性
使用	程序	使用	产品化 生态系统
进程	进程（任务、线程）	提供工作 取得资源 共享资源 包含在模块中 ……	性能 可变性 容量
数据访问	程序和数据段	有权访问	安全性 生态系统

1.4 好的架构

我们曾提到，架构师玩的是折中的游戏。对于一组给定的功能需求和品质需求，没有唯一的正确架构和唯一的“正确答案”。我们从经验中得知，应该对架构进行评估，确定它是否满足其需求，然后再投入资金来构建、测试和部署这个系统。评估试图回答前面小节中讨论的一个或多个一般关注点问题，或回答特定系统的具体关注问题。

架构评估有两种常见的方式（Clements、Kazman和Klein 2002）。第一种评估方式是确定架构的属性，通常通过建模或模拟系统的一个或多个方面。例如，通过性能建模来评估吞吐量和伸缩性，通过失效树模型来评估可靠性和可访问性。其他类型的模型包括复杂性和耦合指标，用于评估可变性和可维护性。

第二种评估方式，也是最广泛使用的方式，就是通过对架构师提出质询来评估该架构。有许多结构化的质询方法。例如，贝尔实验室提出的软件架构复查委员会（Software Architecture Review Board，SARB）过程利用了组织机构之内的专家，以及他们在电信和相关应用中的深厚领域经验（Maranzano等 2005）。

质询方法的另一种变体是架构折中分析方法（Architecture Trade-off Analysis Method，ATAM）（Clements、Kazman和Klein 2002），它寻找架构不能满足品质关注点的风险。ATAM使用了场景分析，每种场景都描述了特定的利益相关人对系统的品质关注点。架构师然后解释该架构如何支持每一种场景。

主动复审是另一种质询方法，它改变了复审过程的开始方式，要求架构师向复审者提供架构师认为重要而需要回答的问题（Hoffman和Weiss 2000，第17章）。然后，复查者利

用已有的架构文档和描述来回答这些问题。最后，在网络上查找“ software architecture review checklist（软件架构复审检查清单）”，可以得到几十份检查清单，其中某些清单非常通用，另一些则是针对具体的应用领域或技术框架。

1.5 美丽的架构

所有前面的方法都有助于我们判断一个架构是否“ 足够好 ” —— 也就是说，是否有可能指导开发者和测试者构建一个系统，并满足系统的利益相关人的功能和质量关注点。在我们每天使用的系统中存在着许多好的架构。

但是，超越足够好的架构是怎样的呢？如果有一个“ 软件架构名人堂 ”，那会怎样？哪些架构会陈列在这个艺术馆的墙上？这个想法可能没有你想象的那么遥远 —— 在软件产品线领域，这样的“ 名人堂 ”的确存在。（注1）进入“ 软件产品线名人堂 ”的条件包括获得商业上的成功、影响其他产品线的架构（其他产品线可能“ 借用、复制、窃取 ”这个架构）、拥有足够的文档从而让其他人“ 不必通过道听途说 ”就能够理解该架构。

我们将为更一般的“ 架构名人堂 ”或“ 美丽架构艺术馆 ”的候选者设立怎样的条件呢？

首先我们应该认识到，这是一个软件系统的艺术馆，而不是其他艺术馆，我们的系统构建的目的是为了使用。所以，我们也许从一开始就应该关注该架构的实用性：它应该每天被许多人使用。

但是，在使用架构之前，它必须先构建，所以我们应该关注该架构的可构建性。我们会寻找那些具有定义良好的使用结构的架构，它们支持增量式构建，这样，通过每次构建迭代我们都能得到一个有用的、可测试的系统。我们也会寻找那些具有定义良好的模块接口、本来就很好测试的架构，这样，构建的过程就是透明的、可见的。

接来下，我们想寻找那些展示了持久性的架构 —— 也就是说，那些经过了时间考验的架构。我们生活在一个技术环境以从未有过的加速度变化的年代。美丽的架构应该预期到变更的需要，允许期望的修改能够容易而有效地进行。我们想寻找那些避免了“ 衰老地平线 ”（Klein 2005）的架构，超过了这条“ 衰老地平线 ”，维护将变得代价极大，以至于不可能进行。

最后，我们还想寻找这样一些架构，它们的特征让使用、构建、测试这些架构的开发人员和测试人员，以及由它而形成的系统的用户感到由衷的高兴。为什么开发人员会高兴？因为它让他们的工作变得容易，而且更有可能得到一个高品质的系统。为什么测试人员会高兴？作为测试过程的一部分，他们必须尝试模拟用户的行为。如果他们高兴，

注1： 参见 *http://www.sei.cmu.edu/productlines/plp_hof.html*.

可能用户也会感到高兴。如果厨师对他的烹调的菜品感到不高兴，那么品尝这些菜品的顾客也可能会感到不高兴。

不同的系统和应用领域为这些架构提供了许多机会，展示它们特有的令人高兴的特征，但概念完整性是一项跨越所有领域的特征，并且总是让人感到高兴。一致的架构学习起来更容易、更快，当知道了一点之后，你就可以开始预测其余的部分。不需要记住并处理特殊的情况，代码更干净，测试集更小。一致的架构不会为做同一件事情提供两种（或更多）方法，不会让用户浪费时间进行选择。正如Ludwig Mies van der Rohe所说的，好的设计，“少即是多”。爱因斯坦可能会说，美丽的架构就是尽可能简单，但不要过于简单。

有了这些判别条件，我们推荐第一批进入“美丽架构艺术馆”的候选者。

第一个是A-7E舰载飞行处理器（Onboard Flight Processor，OFP）的架构，它由海军研究实验室（Naval Research Laboratory，NRL）在20世纪70年代后期开发，在（Bass、 Clements和Kazman 2003)有介绍。尽管这个系统从未实现量产，但它满足了所有其他的判别条件。这个架构对软件架构的实践曾经产生了巨大的影响，它展示在真实世界的系统中，将设计时的信息隐藏模块和使用结构与运行时的进程结构分离。因为美国政府资助并开发了这个架构，所以所有项目文档都提供给了公共领域。（注2）这个架构具有定义良好的使用结构，促进了系统的增量式构建。最后，信息隐藏模块结构为分解系统提供了清晰一致的准则，实现了很强的概念完整性。作为嵌入式系统软件架构的榜样，A-7E OFP当然属于我们的艺术馆。

我们想放入艺术馆的另一个架构是朗讯5ESS电话交换机的软件架构（Carney等 1985）。5ESS取得了全球范围的商业成功，为世界各国的网络提供了核心电话网络交换。它成为性能和可靠性的标准，每个单元每小时能处理超过100万次的连接，平均每年非计划宕机时间少于10秒钟（Alcatel-Lucent 1999）。该架构的一些统一概念，如管理电话连接的“半通话模型”，已经成为电话和网络协议领域的标准模式（Hanmer 2001）。除了保持必须处理的通话类型的数目为2*n*（其中*n*是通话协议的数目）之外，半通话模式还在操作系统的进程概念和电话的通话类型概念之间建立起了联系，从而提供了简单的设计原则，引入了漂亮的架构一致性。在过去的25年中，开发团队涉及多达3000个人，他们发展并增强该系统。基于它的商业成功、持久性和影响，5ESS架构是我们艺术馆的一件好藏品。

还有一个我们想放入美丽架构艺术馆的系统，它就是万维网（World Wide Web，WWW）的架构。它由Tim Berners-Lee在CERN创建，在（Bass、Clements和Kazman 2003）中有介绍。万维网当然已经取得了商业上的成功，它转变了人们使用因特网的方式。即使创建了新的应用、引入了新的功能，它的架构仍然保持不变。该架构的整体简单性促成

注2： 参见CHoffman和Weiss2000）的第6章、第15章和第16章，或在NRL Digital Archives (*http://torpedo.nrl.navy.mil/tu/ps*)中查找“A-7E”。

了它的概念完整性，但有一些决定导致了该架构的完整性保持不变，如客户端和服务器端使用同一个库，创建分层架构以实现分离关注点等。核心万维网架构的持久性和它对新扩展、新功能持续支持的能力，使它当之无愧地进入了我们的艺术馆。

什么是建筑师？

夏天很热的一个日子里，一个外乡人沿着一条路在行走。他走着走着，来到一个人跟前，此人正在路边敲碎石头。

“你在做什么？”他问那个人。

那个人抬头看着他；“我在敲碎石头。你以为我看起来像在干什么？现在不要妨碍我，让我继续干活。”

这个外乡人继续沿着路走，不久他遇到了第二个在大太阳下敲碎石头的人。这个人正努力工作，汗滴如雨。

“你在做什么？”外乡人问道。

这个人抬头看他，露出微笑。

“我在为谋生而工作，”他说，“但这个工作太辛苦了。也许你能给我一份更好的工作？”

外乡人摇了摇头，继续前行。没多久，他遇到了第三个敲碎石头的人。太阳正是最炙热的时候，这个人非常卖力，汗流如注。

“你在做什么？”外乡人问道。

这个人停了一下，喝了一口水，微笑着抬起他的手，指向天空。

“我在建一座大教堂。”他喘着气说。

外乡人看了他一会儿，说：“我们正打算开一家新公司。你来做我们的总建筑师怎么样？”

我们的最后一个例子是UNIX系统，它展示了概念完整性，使用广泛，拥有巨大的影响力。管道和过滤器的设计是讨人喜欢的抽象，允许我们快速构建新的应用。

在描述架构、架构师的角色和创建架构时的考虑等方面，我们已经谈了很多，我们也简单介绍了一些美丽架构的例子。接下来我们邀请你阅读后续章节中详细的例子，这些例子来自于那些技艺精湛的架构师，本书介绍了他们创建并使用过的那些美丽架构。

致谢

David Parnas在几篇论文中定义了我们描述的许多结构，其中包括他的“术语滥用”论文（Parnas 1974）。Jon Bentley为这本书提供了创作灵感，他和Deborah Hill、Mark Klein对早期的草稿提出了许多有价值的建议。

参考文献

Alcatel-Lucent. 1999. “Lucent’s record-breaking reliability continues to lead the industry according to latest quality report.” *Alcatel-Lucent Press Releases. June 2. http://www.alcatel-lucent. com/ wps/ portal/ News Releases/ DetailLucent? LMSG_CABINET= Docs_and_ Resource_ Ctr&LMSG _CONTENT_FILE=News_Releases_LU_1999/LU_News_Article_007318.xml* (accessed May 15, 2008).

Bass, L., P. Clements, and R. Kazman. 2003. *Software Architecture in Practice*, Second Edition. Boston, MA: Addison-Wesley.

Blaauw, G., and F. Brooks. 1997. *Computer Architecture: Concepts and Evolution*. Boston, MA: Addison-Wesley.

Booch, G., J. Rumbaugh, and I. Jacobson. 1999. *The UML Modeling Language User Guide*. Boston, MA: Addison-Wesley.

Brooks, F. 1995. *The Mythical Man-Month*. Boston, MA: Addison-Wesley.

Carney, D. L., et al. 1985. “The 5ESS switching system: Architectural overview.” *AT&T Technical Journal*, vol. 64, no. 6, p. 1339.

Clements, P., et al. 2003. *Documenting Software Architectures: Views and* Beyond. Boston, MA: Addison-Wesley.

Clements, P., R. Kazman, and M. Klein. 2002. *Evaluating Software Architectures*. Boston: Addison-Wesley.

Conway, M. 1968. “How do committees invent.” *Datamation*, vol. 14, no. 4.

Courtois, P. J. 1977. *Decomposability: Queuing and Computer Systems*. New York, NY: Academic Press.

Dijkstra, E. W. 1968. “Co-operating sequential processes.” *Programming Languages*. Ed. F. Genuys. New York, NY: Academic Press.

Garlan, D., and D. Perry. 1995. “Introduction to the special issue on software architecture.” *IEEE Transactions on Software Engineering*, vol. 21, no. 4.

Grinter, R. E. 1999. “Systems architecture: Product designing and social engineering.” *Proceedings of ACM Conference on Work Activities Coordination and Collaboration (WACC '99)*. 11-18. San Francisco, CA.

Hanmer, R. 2001. “Call processing.” *Pattern Languages of Programming* (*PLoP)*. Monticello, IL. *http://hillside.net/plop/plop2001/accepted_submissions/PLoP2001/rhanmer0/PLoP2001_rhanmer0_1.pdf*.

Hoffman, D., and D. Weiss. 2000. *Software Fundamentals: Collected Papers by David L. Parnas*. Boston, MA: Addison-Wesley.

IEEE. 2000. “Recommended practice for architectural description of software intensive systems.” Std 1471. Los Alamitos, CA: IEEE.

Klein, John. 2005. “How does the architect's role change as the software ages?” *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Washington, DC: IEEE Computer Society.

Maranzano, J., et al. 2005. “Architecture reviews: Practice and experience.” *IEEE Software*, March/April 2005.

Parnas, David L. 1974. “On a buzzword: Hierarchical structure.” *Proceedings of IFIP Congress. Amsterdam*, North Holland. [Reprinted as Chapter 9 in Hoffman and Weiss (2000).]

Waldo, J. 2006. “On system design.” *OOPLSA '06*. October 22-26. Portland, OR.

Weiss, D., and C. T. R. Lai. 1999. *Software Product Line Engineering*. Boston, MA: Addison-Wesley.