软件工程——实践者的研究方法

作者简介

Roger S.Pressman 是软件工程领域国际知名的咨询专家和作者。他以优等成绩从 Connecticut 大学获得学士学位，从 Bridgeport 大学获得硕士学位，从 Connecticut 大学获得工学博士学位。已有超过 25 年的产业经验。主要从事工程产品软件和系统软件的开发技术工作和管理工作。

作为产业的实践者和管理者，Pressman 博士主要从事的是航空航天应用中高级工程和制造的 CAD/CAM 系统的开发，他也从事科学及系统程序设计方面的工作。

除了他的产业经验之外，Pressman 博士还是 Bridgeport 大学计算机工程系的兼职副教授和该大学的计算机辅助设计和制造中心的主任。

Pressman 博士是 R.S.Pressman& Associates，Inc 公司的总裁，这是一家专门从事软件工程方法和培训的咨询公司。他是公司主要的咨询专家，专门负责帮助其他公司建立有效的软件工程方法。他开发了 RSP＆A 软件工程评估方法，该方法采用独特的数量和质量分析混合的方式，帮助客户评估他们软件工程实践的当前状况。

除了给 500 多个客户提供咨询服务外，R.S.Pressman & Associates，Inc 公司还提供大量的软件工程培训及过程改善服务。公司开发了一个艺术式的录像课程“Essential SoftwareEngineering”，它全面地讲述了产业界关于这一主题的内容。另一个产品“Process Advisor”是指导企业软件工程改进的自测系统。

Pressman 博士还在产业期刊上发表了许多技术论文，是企业期刊的特约撰稿人并出版了 6 本书。除了本书外，还有：“Making Software Engineering Happen（Prentice Hall 出版公司出版）”，这是第一本涉及到改善软件工程实施过程中的主要管理问题的书籍；“Software Shock（DorsetHouse 出版公司出版）”，该书叙述了软件及其对商业和社会的影响；“A Manager′ s Guide toSoftware Engineering（McGraw－Hill 出版公司出版）”，该书使用独特的 Q＆A 方式表示了创立和理解技术的管理指南。Pressman 博士是杂志 American Programmer（美国程序员)和 IEEESoftware(IEEE 软件)的编委，是 IEEE Software 的 Manager（管理员）专栏的编辑。他还是 ACM、IEEE、Tau Beta Pi、Phi Kappa Phi、Eta Kappa Nu 和 Pi Tau Sigma 的会员。

译者序

20 世纪末发生在我们这个星球上的最大变化之一无疑是席卷全球的信息技术（IT）革命，人们将这场革命视为 21 世纪——知识经济时代的前奏曲。在这场 IT 革命中，软件无疑扮演了极其重要的角色。软件产业作为一个独立形态的产业，正在全球经济中占据越来越举足轻重的地位。而软件工程正是软件产业健康发展的关键技术之一。

从 1968 年软件工程概念的正式提出到现在，软件工程已有逾 30 年的发展，出现了大量的研究成果，也进行了大量的技术实践。正是由于学术界和产业界的共同努力，软件工程正在逐步发展为一门成熟的专业学科，以解决软件生产的质量和效率问题为宗旨，在软件产业的发展中起到了重要的技术保障和促进作用。

本书是一本系统而全面地介绍软件工程理论、技术和实践的专著，是北美学术界和产业界的畅销书之一。本书作者 Roger S.Pressman 是软件工程领域国际知名的咨询专家和作者，著有多本学术专著，本书已是其第四版。本书第二版曾在国内翻译出版，并被很多学校选为软件工程教材，在我国软件工程研究、教学和实践中起到了很好的借鉴和参考作用。而第四版并不仅仅是简单的修订，而是被完全重构以适应软件工程领域快速的增长并着重于新的、重要的软件工程方法。从早期版本保留的章节被全面地修订，并加入了 12 章新内容，以提供对当代趋势和技术的完整讨论。书中还加入了很多新例子、思考题、推荐阅读文献及其它参考信息源。本书的翻译出版旨在向国内软件工程领域的研究、教学、管理和技术人员提供一个全面的参考文献、教材或实践指南。

本书由黄柏素、梅宏组织翻译，其中梅宏负责第二部分 7、8、9 章和第三部分的翻译工作，黄柏素翻译了其余部分并负责全书的统稿工作。同时译者希望向参与了部分章节翻译工作的李克勤、张路、袁望洪、常继传、郭立峰、谢涛、郭耀、马黎等，以及参与了插图绘制和参考文献录入工作的徐松青、沈璞、刘洋、孟祥文等表示诚挚的感谢。

由于译者自身的知识局限及时间的仓促，译稿中难免存在错误和遗漏。谨向读者及原书作者致以歉意，并欢迎指正。

黄柏素、梅宏

黄柏素（女），博士，北京大学计算机科学技术系副教授。1993 年于西北工业大学获得博士学位。同年进入北京大学计算机科学技术系博士后流动站。1995 年出站后留校工作。主要研究方向为软件工程、软件开发环境及工具、面向对象技术、用户界面管理系统等。承担了软件工程课程教学工作。目前已发表学术论文 20 余篇，并获得多项国家及部委科技成果奖和个人奖。

梅宏，博士，北京大学计算机科学技术系教授。1992 年于上海交通大学获工学博士学位，1994 年从北京大学计算机科学技术系博士后出站。研究、教学工作主要涉及软件工程及软件开发环境、软件复用及软件构件技术、（分布）对象技术、软件工业化生产技术及支持系统、新型程序设计语言等。已在国内外学

术刊物及国际、全国学术会议上发表学术论文 60 余篇。并多次获得国家及部委级科技成果奖，以及其他个人荣誉奖。

前　言

软件工程已进入到目前的第四代，它已具有许多优势，虽然它仍存在同时代人曾经历的某些弱点，但其早年的天真和热情已被更合理的经历多年培育的期望（以及甚至善意的嘲讽）所替代，软件工程正带着许多成就步入中年，然而还有大量工作需要去做，今天，它已被公认为一个重要学科，值得认真地研究、细心地学习和热烈地争论。在整个产业界，“软件工程师”已经替代“程序员”成为更受欢迎的工作头衔。产业应用软件中已广泛而成功地采用了软件过程模型、软件工程方法以及软件工具。管理者和实践者均认识到，需要一个更严谨的软件方法来支持手头的工作。

但是，在本书的早期版本中很多讨论的问题仍然存在，很多个人和公司仍然在随意地开发软件，很多专业人员和学生不知道现代方法，最终，我们生产的软件仍然存在大量质量问题。此外，关于软件工程方法的真实性质的争论仍在继续。然而，今天软件工程已成为研究的热点，人们对它的态度已有很大变化，它的发展也很明显，但是，要使软件工程最终发展成为一个完全成熟的学科还需做大量工作。

本书的第 4 版试图成为正逐步走向成熟的软件工程学科的一个指南。和前面三版一样，第 4 版的主要读者群仍然是学生和实践者，而且在写作风格上我们力图仍然保持前面各版的格式和风格。本书的基本目标仍然是：作为产业界专业人员的指南以及作为高年级大学生和一年级研究生的软件工程的全面导论。

我们在第 4 版中并不仅仅简单地修订了原稿，为适应本领域快速的增长我们完全重新组织了书中的内容，并着重讨论了新的重要的软件工程方法，还全面地修订了从早期版本保留的章节，加入了 12 章新内容，以提供对当代趋势和技术的完整讨论。加入了很多新例子、思考题，每一章中还增补了推荐阅读文献及其他信息搜索地址，包括数百个新的出版站点以及超过 160 个 WWW 信息站地。

第 4 版由 5 个部分共 30 章构成。这样做的目的是按专题安排内容，并使那些没有时间在一个学期内完成书中内容教学任务的老师，可以按需取用。第一部分：产品和过程，简介软件工程的相关语境，引出书中主要内容，并着重介绍了以后章节用到的概念；第二部分：管理软件项目，讨论那些与计划、管理和控制软件开发项目的人员相关的话题；第三部分：传统软件工程方法，讨论那些被视为传统软件工程不同“学派”的分析、设计和测试方法；第四部分：面向对象软件工程，讨论跨越整个软件工程过程的面向对象方法，包括分析、设计和测试方法；第五部分：软件工程高级课题，分章专门讨论形式化方法、净室软件工程、复用、再建工程、客户/服务器软件工程和 CASE。

第 4 版比以前版本更多地强调了度量和测度方面的相关技术。有三章和软件度量相关，分别是：软件过程和项目的度量、软件的技术度量、面向对象系统的技术度量。

本书的组织使得老师可以根据时间和学生需要安排授课话题。一个学期可选择一个或多个部分。例如，“设计课程”可能只需要第三或第四部分，“方法课程”可能只需第三、第四和第五部分的部分章节，“管理课程”可能只需要第一和第二部分。按这种方式组织本书第 4 版，目的是给老师提供灵活的教学选择。

第 4 版的写作工作已成为我生活中持续最长的技术计划。即使当写作停止时，从各种技术文献中提炼、组织信息的工作也一直在进行，为此，我要感谢许多书籍、论文和文章的作者，以及新一代的电子媒体（新闻组和 WWW）的贡献者们，他们在过去的 15 年中给我提供了大量的信息资源、思想和评注，很多信息资源已在每章的参考文献中列出，他们在这个快速发展的领域中的贡献是值得称道的。我还要感谢第 4 版的审阅者：Wayne State University 的 Frank H.Westervelt、The University of Connecticut 的 Steven A.Demurjian、California State PolytechnicUniversity 的 Chung Lee、University of Colorado 的 Alan Davis、QSM Associates 的 Michael C.Mah、University of California－Irvine 的 Richard N.Taylor、Virginia Tech.的 Osman Balci、AuburnUniversity 的 James H.Cross、Portland State University 的 Warren Harrison、NortheasternUniversity 的 Mieczyslaw M.Kokar，他们的评注和批评是无价的。

本书第 4 版内容的成型有赖于许多曾经使用过本书以前版本的产业界专业人员、大学教授和学生，他们花了很多时间和我通信交流他们的建议、批评和思想，我要感谢他们中的每一位。

此外，我也要向我们的在北美和欧洲的许多产业客户表示感谢，他们教我的比我教他们的要多。

Roger S.Pressman

第一部分　产品和过程

在本书的这一部分中我们主要讨论什么是工程产品和如何为工程技术提出一个框架的过程。在下面的章节中，我们主要解决下列问题：

- 到底什么是计算机软件？
- 为什么我们不断努力要建造高质量的基于计算机的系统？
- 我们如何对计算机软件的应用领域分类？
- 关于软件仍存在什么样的神话？
- 什么是软件过程？
- 是否存在一般性的方法评价一个过程的质量？

• 软件开发中可以应用什么过程模型？

• 线性过程和迭代过程有何区别？

• 它们的优点和缺点是什么？

• 在软件工程中可以建议什么更高级的过程模型？

一旦回答了这些问题，读者就能够更好地理解本书其余部分给出的工程原则的管理和技术方面的知识。

第 1 章　产品

本书的第 1 版在 80 年代初出版后不久，Business Weekly(《商业周刊》)杂志在头版给出如下的大标题：“软件：新的驱动力”。编辑们当时并没有意识到他们的预见是多么的正确。那时，大多数人对软件还是一无所知。大软件公司，如微软公司，还不存在；拥有 15000 平方英尺专门出售包装好的软件的计算机超市闻所未闻；在电视上为计算机操作系统做 60 秒钟商业广告的想法是可笑的；而互联网仅为个别研究者和高等学校学生所知。但是，在不到 20 年的时间里，所有这些(甚至更多)已经成为现实。

计算机软件已经成为一种驱动力。它是进行商业决策的引擎；它是现代科学研究和工程问题解决的基础；它也是区分现代产品和服务的关键因素。它在各种类型的系统中应用，如交通、医药、通讯、军事、产业化过程、娱乐、办公……难以穷举。软件在现代社会中的确是必不可少的。而且当我们进入 21 世纪，软件将成为从基础教育到基因工程的所有各领域新进展的驱动器。

所有这一切已经改变了软件的常见概念。计算机软件是无所不在的，人们把软件看作是生活中现实的技术。在很多情况下，人们把他们的工作、他们的舒适、他们的安全、他们的娱乐、他们的决策、甚至他们的整个生活都依赖于计算机软件。软件千万可不能出错。

本书介绍的若干技术是那些想要建造正确的计算机软件的人们需要用到的。这些技术包括一个过程，一组方法和一系列工具，我们称之为软件工程。

1.1 软件的发展

今天，软件担任着双重角色。它是一种产品，同时又是开发和运行产品的载体。作为一种产品，它表达了由计算机硬件体现的计算潜能。不管它是驻留在蜂窝电话中，还是操作在主机上，软件就是一个信息转换器——产生、管理、获取、修改、显示或转换信息，这些信息可以很简单，如一个单个的位(bit)，或很复杂，如多媒体仿真信息。作为开发运行产品的载体，软件是计算机控制(操作系

统)的基础、信息通信(网络)的基础,也是创建和控制其他程序(软件工具和环境)的基础。

许多人相信 21 世纪最重要的产品是——信息，软件充分体现了这一观点。它处理个人数据(如个人的金融事务)，使得这些数据在局部范围中更为有用；它管理商业信息增强了商业竞争力；它提供了通往全球信息网络(如 Internet)的途径；它也提供了以各种形式获取信息的手段。

计算机软件的角色在 20 世纪后半叶发生了很大的变化。硬件性能的极大提高，计算机体系结构的不断变化，内存和硬盘容量的快速增加，以及大量输入输出设备的多种选择,均促进了更为成熟和更为复杂的基于计算机的软件系统的出现。如果一个系统是成功的,那么这种成熟性和复杂性能够产生出奇迹般的结果,但是它们也给建造这些复杂系统的人员带来很多的问题。

在 70 年代和 80 年代出版的受欢迎的书对于计算机、软件和它们对我们文化的影响等方面提供了有用的历史的视角。Osborne[OSB79]称之为一次“新的工业革命”。Toffler [TOF80] 称微电子的发展是人类历史上的“第三次浪潮”，Naisbitt [NAI82] 则预言了从工业社会向“信息社会”的转变。Feigenbaum 和 McCorduck [FEI83] 认为由计算机控制的信息和知识将是 20 世纪中表现能力的焦点,Stoll[STO89]则提出由网络和软件产生的“电子社会”将是全球知识交换的关键。

进入 90 年代，Toffler [TOF90] 描述了“权利的转移”，因为计算机和软件导致了“知识的民主化”，因而旧的权利结构(政府，教育，工业，经济，及军事)将要瓦解。Yourdon [YOU92] 担心美国公司在软件相关的业务中会失去竞争力，并预言“美国程序员的衰落和下降”。Hammer 和 Champy [HAM93] 提出信息技术将在“公司的再工程”中起到很关键的作用。在 90 年代中期，计算机和软件的流行产生了大量“新劳工运动”的书籍(例如：由 James Brook 和 Iain Boal 编辑的“抵制虚拟的生活”，以及 Stephen Talbot 写的“未来不是计算”)。这些作者把计算机看成是魔鬼，强调了其合法性，而忽略了已被人们意识到的巨大的利益 [LEV95] 问题。

在计算机发展的早期阶段，大多数人把软件看成是不需预先计划的事情。计算机编程很简单，没有什么系统化的方法。软件的开发没有任何管理，一旦计划延迟了或成本提高了，程序员才开始手忙脚乱地弥补，而他们的努力一般情况下也会取得成功。

在通用的硬件已经非常普遍的时候，软件却相反，对每一类应用均需自行再设计，应用范围很有限。软件产品还在婴儿阶段，大多数软件均是由使用它们的人员或组织自己开发的，如你写软件，使其运行，如果它有问题，你负责改好。工作的可变性很低，管理者必须得到保证：一旦发生了错误你必须在那里。因为这种个人化的软件环境，设计往往仅是人们头脑中的一种模糊想法，而文档就根本不存在。

在早期，我们了解了很多关于计算机系统的实现，但对于计算机系统工程几乎一无所知。但是公平地讲，我们应该感谢这个时期开发的许多卓越的计算机系统，其中不少一直到今天还在使用，并继续发挥着巨大的作用。

计算机系统发展的第二阶段跨越了从60年代中期到70年代末期的十余年(如图1－1)。多道程序设计、多用户系统引入了人机交互的新概念。交互技术打开了计算机应用的新世界，以及硬件和软件配合的新层次。实时系统能够从多个源收集、分析和转换数据，从而使得进程的控制和输出的产生以毫秒而不是分钟来进行。在线存储的发展导致了第一代数据库管理系统的出现。

第二阶段还有一个特点就是软件产品的使用和“软件作坊”的出现。软件被开发，使得它们可以在很宽的范围中应用。主机和微机上的程序能够有数百甚至上千的用户。来自工业界、政府和学术界的企业家们纷纷开始开发各类软件包，并赚了大笔钱财。

早期	第二阶段	第三阶段	第四阶段
• 面向批处理	• 多用户	• 分布式系统	• 强大的桌面系统
• 有限的分布	• 实时	• 嵌入“智能”	• 面向对象技术
• 自定义软件	• 数据库	• 低成本硬件	• 专家系统
	• 软件产品	• 消费者的影响	• 人工神经网络
		• 并行计算	
		• 网络计算机	

随着计算机系统的增多，计算机软件库开始扩展。内部开发的项目产生了上万行的源程序，从外面购买的软件产品加上几千行新代码就可以了。这时，一团黑云出现在地平线上，当发现错误时需要纠正所有这些程序(所有这些源代码)；当用户需求发生变化时需要修改；当硬件环境更新时需要适应。这些活动统称为软件维护。在软件维护上所花费的精力开始以惊人的速度消耗资源。

更糟糕的是，许多程序的个人化特性使得它们根本不能维护。“软件危机”出现了。

计算机系统发展的第三阶段始于70年代中期并跨越了整整十年。分布式系统——多台计算机，每一台都在同时执行某些功能，并与其他计算机通讯——极大地提高了计算机系统的复杂性。广域网和局域网、高带宽数字通讯以及对“即时”数据访问需求的增加都对软件开发者提出了更高的要求。然而，软件仍然继续应用于工业界和学术界，个人应用很少。

第三阶段的主要特点是微处理器的出现和广泛应用。微处理器孕育了一系列的智能产品——从汽车到微波炉，从工业机器人到血液诊断设备——但那一个也

没有个人计算机那么重要，在不到十年时间里，计算机真正成为大众化的东西。①

计算机系统发展的第四个阶段已经不再是着重于单台计算机和计算机程序，而是面向计算机和软件的综合影响。由复杂的操作系统控制的强大的桌面机，广域和局域网络，配合以先进的软件应用已成为标准。计算机体系结构迅速地从集中的主机环境转变为分布的客户机/服务器环境。世界范围的信息网提供了一个基本结构，使得学者和政治家可以同样考虑“信息高速公路”和“网际空间连通”的问题。事实上，Internet 可以看作是能够被单个用户访问的“软件”。

软件产业在世界经济中不再是无足轻重的。由产业巨子如微软做的一个决定可能会带来成百上千亿美元的风险。随着第四阶段的进展，一些新技术开始涌现。面向对象技术(本书第四部分)在许多领域中迅速取代了传统软件开发方法。虽然关于“第五代”计算机的预言仍是一个未知数，但是软件开发的“第四代技术”确实改变了软件界开发计算机程序的方式。专家系统和人工智能软件终于从实验室里走了出来，进入了实际应用，解决了现实世界中的大量问题。结合模糊逻辑应用的人工神经网络软件揭示了模式识别和类似人的信息处理能力的可能性。虚拟现实和多媒体系统使得与最终用户的通讯可以采用完全不同的方法。“遗传算法”[BEG95]则提供了可以驻留于大型并行生物计算机上的软件的潜在可能性。

但是，一系列软件相关的问题在计算机系统的整个发展过程中一直存在着，而且这些问题还会继续恶化：

1.硬件的发展一直超过软件，使得我们建造的软件难以发挥硬件的所有潜能。

2.我们建造新程序的能力远远不能满足人们对新程序的需求，同时我们开发新程序的速度也不能满足商业和市场的要求。

3.计算机的普遍使用已使得社会越来越依赖于可靠的软件。如果软件失败，会造成巨大的经济损失，甚至有可能给人类带来灾难。

4.我们一直在不断努力建造具有高可靠性和高质量的计算机软件。

5.拙劣的设计和资源的缺乏使得我们难以支持和增强已有软件。

为了解决这些问题，整个产业界开始采用了软件工程实践。

1.1.1 产业的观点

在计算机发展的早期，计算机系统是采用面向硬件的管理方法来开发的。项目管理者着重于硬件，因为它是系统开发中最大的预算项。为了控制硬件成本，管理者建立了规范的控制和技术的标准。他们要求在真正开始建造系统之前，进行详尽的分析和设计，他们度量过程，以发现哪里还可以进一步改进，他们坚持

质量控制和质量保证，他们设立规程，以管理变化。简言之，他们应用了控制、方法和工具，我们可以称之为硬件工程。但遗憾的是软件只不过是事后才考虑的事情。

在早期，程序设计被看作是一门“艺术”。几乎没有规范化的方法，也没有人使用它们。程序员往往从试验和错误中积累经验。建造计算机软件的专业性和挑战性，使其披上了一种神密的面纱，管理者们很难了解它。软件世界真是完全无序——这是一个开发者的为所欲为的时代。

今天，计算机系统开发成本的分配发生了戏剧性的变化。软件，而不是硬件，是最大的成本项。在近二十年里，管理者和很多开发人员在不断地问以下的问题：

• 为什么需要那么长时间才能结束开发？

• 为什么成本如此之高？

• 为什么我们不能在把软件交给客户之前就发现所有的错误？

• 为什么在软件开发过程中我们难以度量其进展？

这些问题以及其他许多问题都表明了对软件及其开发的方式是必须关注了——这种关注最终导致了软件工程实践的出现。

1.1.2 老化的软件工厂

在 50 和 60 年代，许多评论家批评美国的钢铁产业缺少对其工厂的投入。工厂开始恶化，现代化的方法很少被采纳，最终产品的质量和成本难以容忍，外来的竞争开始赢得市场份额。这些企业在管理中确定把主要的投资投入到其主营业务中，以保持竞争力。随着时间的推移，美国的钢铁产业蒙受了巨大损失，大量市场份额被外来竞争者占领——这些企业拥有新工厂，采用更为现代化的技术，并且得到政府的资助，使得他们极具价格优势。

在那个时期，羽翼未丰的计算机产业中的很多人都以蔑视的态度评价钢铁产业，“如果它们不愿在自己的业务上投入，那当然会失去市场份额”。这些话现在轮到说我们自己了。

听上去很戏剧化，今天的软件产业就像五、六十年代的钢铁产业，无论大公司还是小公司，都有一个老化的“软件工厂”——有成千上万的重要的基于软件的应用程序急待更新：

• 20 年前开发的信息系统应用程序经过了几十次的修改，已经真正不可维护了。即使是最小的修改也会引起整个系统失败。

• 一些用于生成关键设计数据的工程应用程序，由于不断的修改和老化，已经没有人真正了解其内部结构。

• 嵌入式系统(有成千上万的这类应用程序，其中包括核电站控制、航空调度和工厂管理)表现出奇怪的有时甚至是无法解释的行为，但又不能不用它们，因为目前还没有其他方法可以替代它们。

有问题就“打补丁”，并给这些应用程序一个时髦的界面，仅仅如此是不够的。软件工厂的许多构件需要再生产，否则它们就不再具有竞争力了。但不幸的是，许多企业的管理者并不愿投入资源去进行再生产，他们辩解：“这些应用程序仍能工作，投入资源去使得它们更好是不经济的”。

1.1.3 软件的竞争

许多年来，大、小公司雇佣的软件开发人员仅仅在公司内部服务，而且他们也愿意这样。因为每一个计算机程序都是自行开发的，这些“自家”的软件人员控制着成本、进度和质量。今天，所有这一切都改变了。

软件目前是一个竞争很强的行业。曾经要自行开发的软件现在可以在货架上买到，许多公司过去雇佣了大量的程序员开发特定的软件，现在它们大部分的软件工作已交给第三方厂商去完成 [MIN95] 。

成本、进度和质量将是未来若干年中导致软件激烈竞争的主要因素。美国和西欧有很成熟的软件产业，而远东(如韩国，新加坡)、亚洲(如印度、中国)和东欧的一些国家拥有大量的有天份、受过良好教育且相对较低廉的专门人才 [ECO94]。这种压力导致了必须迅速采用现代化的软件工程实践的需要，同时软件开发也是一个必须认真考虑的因素，因为世界范围的软件从业人员都在追求尽量少的开发费用。

在关于信息服务对美国及世界的影响的论著中，Feigenbaum 和 McCorduck[FEI83]作了如下陈述：

知识就是力量，而计算机是这种力量的倍增器……美国的计算机产业是创新的、充满活力的和成功的。在某种程度上，它是一个理想的产业。它通过转化知识分子的脑力劳动来产生价值，而几乎不需要什么能源和原材料。今天，我们在这个最重要的现代技术上领导着世界的想法和市场，但明天会怎样哪？

是的，明天会怎样哪？计算机硬件已经成为一种商品，可从很多渠道得到。但软件仍然是美国保持着“创新的、充满活力的和成功的”一个产业。但我们还会继续保持领先吗？至少答案的一部分就在这里：我们将采用什么样的方法去建造下一代计算机系统的软件。

1.2 软件

在 1970 年，只有不到 1%的人能够比较准确地描述出什么是“计算机软件”。而现在，大多数专业人士和许多业外公众都认为他们了解了什么是软件。但他们真的了解吗？

关于软件，教科书上一般是如下定义的：软件是(1)能够完成预定功能和性能的可执行的指令(计算机程序)；(2)使得程序能够适当地操作信息的数据结构；(3)描述程序的操作和使用的文档。毫无疑问，也可以给出其他更详细的定义。但我们不只是需要一个形式上的定义。

1.2.1 软件特征

要理解软件的含义(以及对软件工程有一个全面的理解)，首先要了解软件的特征是很重要的，据此能够明白软件与人类建造的其他事物之间的区别。当建造硬件时，人的创造性的过程(分析、设计、建造、测试)能够完全转换成物理的形式。如果我们建造一个新的计算机，初始的草图、正式的设计图纸和面板的原型一步步演化成为一个物理的产品(VLSI 芯片、线路板、电源等等)。

而软件是逻辑的而不是物理的产品。因此，软件具有与硬件完全不同的特征：

1.软件是由开发或工程化而形成的，而不是传统意义上的制造产生的。

虽然在软件开发和硬件制造之间有一些相似之处，但两者本质上是不同的。这两者，都可以通过良好的设计获得高质量，但硬件在制造过程中可能会引入质量问题，这种情况对于软件而言几乎不存在(或是很容易改正)。软件成为产品之后，其制造只是简单的拷贝而已；两者都依赖于人，但参与的人和完成的工作之间的关系不同；两者都是建造一个产品，但方法不同(见第 3 章)。

软件成本集中于开发上，这意味着软件项目不能象硬件制造项目那样来管理。

在 80 年代中期，“软件工厂”的概念被正式引入［MAN84］、［YAJ84］。应该注意到这个术语并没有把硬件制造和软件开发认为是等价的。而是通过软件工厂这个概念提出了软件开发中应该使用自动化工具(见第 5 部分)。

2.软件不会“磨损”。

图 1－2 刻划了随着时间的改变硬件故障率的变化曲线图。其关系，常常被称作“浴缸曲线”，表明了硬件在其生命初期有较高的故障率(这些故障主要是由于设计或制造的缺陷)；这些缺陷修正之后，故障率在一段时间中会降到一个稳定的曲线上(很低)。随着时间的改变，故障率又提升了，这是因为硬件构件由于种种原因会不断受到损害，例如灰尘、振动、滥用、温度的急剧变化以及其他许多环境问题。简单讲，硬件已经开始磨损了。

软件并不受到这些引起硬件磨损的环境因素的影响。因此，理论上讲，软件的故障率曲线呈现出如图 1－3 所示的形式。隐藏的错误会引起程序在其生命初期具有较高的故障率。但这些错误改正之后(我们假设理想情况下改正过程中并不引入其他错误)，曲线就趋于平稳，如图所示。图 1－3 给出了实际软件故障模型的一个总的简化图(第 8 章将给出更多信息)。其意义很清楚——软件不会磨损，不过它会退化。

这个说法表面上似乎是矛盾的，我们可以通过图 1－4 来解释清楚。在其生命期中，软件会经历修改(维护)，随着这些修改，有可能会引入新的错误，使得故障率曲线呈现为图 1－4 所示的锯齿形。在该曲线能够恢复到原来的稳定状态的故障率之前，又需要新的修改，又引起一个新的锯齿。慢慢地，最小故障率就开始提高了——软件的退化由于修改而发生了。

关于磨损的另一个侧面也表明了硬件和软件之间的不同。当一个硬件构件磨损时，可以用另外一个备用零件替换它，但对于软件就没有备用零件可以替换了。每一个软件故障都表明了设计或是将设计转换成机器可执行代码的过程中存在错误。因此，软件维护要比硬件维护复杂得多。

3.大多数软件是自定的，而不是通过已有的构件组装而来的。

我们先看一看一个基于微处理器的控制硬件是如何设计和建造出来的。设计工程师画一个简单的数字电路图，做一些基本的分析以保证可以实现预定的功能，然后查阅所需的数字零件的目录。每一个集成电路(通常称为“IC”或“芯片”)都有一个零件编号、固定的功能、定义好的接口和一组标准的集成指南。每一个选定的零件，都可以在货架上买到。

而软件设计者就没有上述这种荣幸了。几乎没有软件构件。有可能在货架上买到的软件，它本身就是一个完整的软件，而不能作为构件再组装成新的程序[①]。虽然关于“软件复用”已有大量论著，但这种概念的成功实现还只是刚刚开始。

1.2.2 软件构件

随着工程化的发展，大量标准的设计构件产生了。标准螺丝和货架上的集成电路芯片仅仅是成千上万的标准构件中的两种，机械和电子工程师在设计新系统时会用到它们。这些可复用构件的使用使得工程师们能够集中精力于设计中真正有创造性的部分(如设计中那些新的成分)。在硬件中，构件复用是工程化的必然结果。而在软件中，它还仅仅是在小范围内取得一定应用。

可复用性(Reusability)是高质量软件构件的一个重要特征(第 26 章将给出更详细的讨论)，一个软件构件应该被设计和实现为能够在多个不同程序中复用。在 60 年代，我们建造了科学计算子程序库，它们能够在很多工程和科学应用中复用，这些子程序库可以以一种高效的方式复用，定义明确的算法，但其应用范围有限。今天，我们已经扩展了复用的概念，不仅是算法，还可以是数据结构。现代的可复用构件包含了数据以及应用这些数据的处理过程，使得软件工程师能

够从已有可复用构件中创建新的应用[②]。例如，现在交互界面都是通过可复用构件建造的，你可以使用它们创建图形窗口、下拉式菜单和各种交互机制。建造用户界面所需的数据结构和处理细节均包含在一个可复用的界面建造构件库中。

软件构件使用某种程序设计语言实现，该语言具有一个有限的词汇表、一个明确定义的文法及语法和语义规则。在最底层，该语言直接反映了硬件的指令集；在中层，程序设计语言，如 Ada 95、C 或 Smalltalk 可用于创建程序的过程化描述；在最高层，该语言可使用图形化的图标或其他符号去表示关于需求的解决方案。由于可执行代码就自动生成了。

机器级语言是 CPU 指令集的一个符号表示。当一个好的软件开发者在开发一个可维护、文档齐全的程序时，使用机器语言能够很高效地利用内存并优化该程序的执行速度。当程序设计得很差且没有文档时，机器语言就是一场恶梦。

中层语言使得软件开发者和程序可独立于机器。如果使用了很好的翻译器，一个中层语言的词汇表、文法、语法和语义都能够比机器语言高级得多。事实上，中层语言的编译器和解释器的输出就是机器语言。

虽然目前有成百上千种的程序设计语言，但只有不到 10 种中层的程序设计语言在工业界广泛使用。一些语言，如 COBOL 和 FORTRAN 从它们发明至今已经流行了 30 余年，更多的现代程序设计语言，如 Ada95、C、C＋＋、Eiffel、Java 和 Smalltalk 也各自有一大批热心的追随者。

机器代码，汇编语言(机器级语言)和中层程序设计语言通常被认为是计算机语言的前三代。因为这些语言中的任何一种，都需程序员既要关心信息结构的表示，又要考虑程序本身的控制。因此这前三代语言被称为是过程语言。

第四代语言，也称非过程语言，使得软件开发者更加独立于计算机硬件。使用非过程语言开发程序，不需要开发者详细说明过程化的细节，而仅仅“说明期望的结果，而不是说明要得到该结果所需要的行为”[COB85]。支撑软件会把这种规约自动转换成机器可执行的程序。

1.2.3 软件应用

软件可以应用于任何场合，只要定义了一组预说明的程序步骤(如一个算法，但也有例外，如专家系统和人工神经网络)。信息的内容和确定性是决定一个软件应用的特性的重要因素。内容指的是输入和输出信息的含义和形式，例如，许多商业应用使用高结构化的输入数据(一个数据库)，且产生格式化的输出“报告”。而控制一个自动化机器的软件(如一个数控系统)则接受限定结构的离散数据项，并产生快速连续的单个机器命令。

信息的确定性指的是信息的处理顺序及时间的可预定性。一个工程分析程序接受预定顺序的数据，不间断的执行分析算法，并以报告或图形格式产生相关的数据。这类应用是确定的。而一个多用户操作系统，则接受可变化内容和任意时

序的数据，执行可被异常条件中断的算法，并产生随环境功能及时序而变化的输出。具有这些特点的应用是非确定的。

在某种程度上讲我们难以对软件应用给出一个通用的分类。随着软件复杂性的增加，其间已没有明显的差别。下面给出一些软件应用领域，它们可能是一种潜在的应用分类：

系统软件：系统软件是一组为其他程序服务的程序。一些系统软件(如编译器、编辑器和文件管理程序)处理复杂的但也是确定的信息结构。其他的系统应用(如操作系统、驱动程序和通讯进程等)则处理大量的非确定的数据。不管哪种情况，系统软件均具有以下特点：与计算机硬件频繁交互；多用户支持；需要精细调度、资源共享及灵活的进程管理的并发操作；复杂的数据结构；及多外部接口。

实时软件：管理、分析、控制现实世界中发生的事件的程序称为实时软件。实时软件的组成包括：一个数据收集部件，负责从外部环境获取和格式化信息；一个分析部件，负责将信息转换成应用时所需要的形式；一个控制/输出部件，负责响应外部环境；及一个管理部件，负责协调其他各部件，使得系统能够保持一个可接受的实时响应时间(一般从 1 毫秒到 1 分钟)，应该注意到术语“实时”不同于“交互”或“分时”。一个实时系统必须在严格的时间范围内响应。而一个交互系统(或分时系统)的响应时间可以延迟，且不会带来灾难性的后果。

商业软件：商业信息处理是最大的软件应用领域。具体的“系统”(如工资表、帐目支付和接收、存货清单等)均可归为管理信息系统(MIS)软件，它们可以访问一个或多个包含商业信息的大型数据库。该领域的应用将已有的数据重新构造，变换成一种能够辅助商业操作和管理决策的形式。除了传统的数据处理应用之外，商业软件应用还包括交互式的和客户机/服务器式的计算(如 POS 事务处理)。

工程和科学计算软件：工程和科学计算软件的特征是“数值分析”算法。此类应用含盖面很广，从天文学到火山学；从汽车压力分析到航天飞机的轨道动力学；从分子生物学到自动化制造。不过，目前工程和科学计算软件已不仅限于传统的数值算法。计算机辅助设计、系统仿真和其他交互应用已经开始具有实时软件和系统软件的特征。

嵌入式软件：智能产品在几乎每一个消费或工业市场上都是必不可少的，嵌入式软件驻留在只读内存中，用于控制这些智能产品。嵌入式软件能够执行很有限但专职的功能(如微波炉的按钮控制)，或是提供比较强大的功能及控制能力(如汽车中的数字控制，包括燃料控制、仪表板显示，刹车系统等)。

个人计算机软件：个人计算机软件市场是在过去十年中萌芽和发展起来的。字处理、电子表格、计算机图形、多媒体、娱乐、数据库管理、个人及商业金融应用、外部网络或数据库访问，这些仅仅是成百上千这类应用中的几种。

人工智能软件：人工智能(AI)软件利用非数值算法去解决复杂的问题，这些问题不能通过计算或直接分析得到答案。一个活跃的 AI 领域是专家系统，也称为基于知识的系统。AI 软件的其他应用领域还包括模式识别(图象或声音)、定理证明和游戏。最近，AI 软件的一个新分支，称为人工神经网络，得到了很大进展。神经网络仿真人脑的处理结构(生物神经系统的功能)，这有可能导致一个全新类型的软件登场，它不仅能够识别复杂的模式，而且还能从过去的经验中自行学习进步。

1.3 软件：地平线上的危机

许多产业界观察者(包括本书早期版本的作者)都把与软件开发相关的问题称为“危机”。但实际上我们的问题与这个术语真正的含义可能并不相同。

“危机”这个词在韦氏字典中定义为“任何事情过程中的一个转折点；决定性的或危急的时刻、阶段或事件”。而对于软件而言，没有“转折点”，没有“决定性时刻”，只有延迟或进展上的变化。在软件产业中，“危机”已经伴随我们走过了近 30 年，这在术语上是很矛盾的。

在字典中查找“危机”这个词，我们还可以发现另外一个定义：“疾病过程中的一个转折点，能够确定病人是生是死”。这个定义多少可以反映出软件开发中面临的问题的真正含义。

我们已经到了计算机软件的危机阶段，实际上我们真正的问题应该是一种“慢性的苦恼”[①]。“苦恼”这个词定义为“引起痛苦或不幸的任何事情”。但形容词“慢性的”这个词的定义能够更加贴切地反映问题：“持续很长时间或经常重犯；不确定地延续”。这很准确地描述了过去 30 年我们所经历的问题是一种慢性的苦恼，而不是危机。并没有一种灵丹妙药可以完全治愈这种病疼，但在我们正努力去发现解决方案的同时会得到很多缓解痛苦的方法。

不管我们称之为软件危机还是软件苦恼，该术语都是指在计算机软件开发中所遇到的一系列问题。这些问题不仅局限于那些“不能正确完成功能的”软件，还包含那些与我们如何开发软件、我们如何维护大量已有软件以及我们的开发速度如何跟上目前对软件越来越大的需求等相关的问题。

虽然关于危机甚至是苦恼的介绍可能看起来有些戏剧化，但这些段落对于表明在软件开发的所有领域中所遇到的真正的问题起到很大作用。

1.4 软件神话

引起软件危机的诸多原因可以追溯到软件开发的早期阶段产生的神话。它不象古代的神话那样可以给人以经验和教训，软件神话使人产生了误解和混乱。软件神话具有一些特征使得它们很有欺骗性：例如，它们表面上看很有道理(有时

含有一定真实的成分)；它们符合人的直觉；它们常常是有经验的实践者发布出来的。

今天，大多数专业人员已经认识到这些神话误导了人们，给管理者和技术人员都带来了严重的问题。但是，旧的观念和习惯难以改变，软件神话仍被不少人相信着。

管理者的神话：负责软件的管理者象大多数其他行业的管理者一样，都有巨大的压力，要维持预算、保持进度及提高质量。就像溺水者抓住一根救命稻草，软件管理者常常抓住软件神话不放，如果这些神话能够缓解其压力的话(哪怕是暂时的)。

神话：我们已经有了关于建造软件的标准和规程的书籍，难道它们不能给人们提供所有其需要知道的信息吗？

事实：不错关于标准的书籍已经存在，但真正用到了它们吗？软件实践者知道它们的存在吗？它们是否反映了现代软件开发的过程？它们完整吗？很多情况下，对于这些问题的答案均是“不”。

神话：我们已经有了很多很好的软件开发工具，而且，我们为它们买了最新的计算机。

事实：为了使用最新型号的主机、工作站和 PC 机去开发高质量的软件，我们已经投入太多的费用。实际上，计算机辅助软件工程(CASE)工具相比起硬件而言对于获得高质量和高生产率更为重要，但大多数软件开发者并未使用它们。

神话：如果我们已经落后于计划，可以增加更多的程序员来赶上进度(“有时称为蒙古大夫概念”)。

事实：软件开发并非象制造一样是一个机械过程。用 Brooks [BR075] 的话来说，“给一个已经延迟的软件项目增加人手只会使其更加延迟”。看起来，这句话与人的直觉正好相反。但实际上，增加新人，原来正在工作的开发者必须花时间来培训新人，这样就减少了他们花在项目开发上的时间。人手可以增加，但只能是在计划周密、协调良好的情况下。

用户的神话:需要计算机软件的用户可能就是邻桌的人,或是另一个技术组，也可能是市场/销售部门，或另外一个公司。在许多情况下，用户相信关于软件的神话，因为负责软件的管理者和开发者很少去纠正用户的错误理解。神话导致了用户过高的期望值，并引起对开发者的极端不满意。

神话:有了对目标的一般描述就足以开始写程序了——我们可以以后再补充细节。

事实：不完善的系统定义是软件项目失败的主要原因。关于待开发项目的应用领域、功能、性能、接口、设计约束及确认标准的形式化的、详细的描述是必需要的。这些内容只有通过用户和开发者之间的通信交流才能确定。

神话：项目需求总是在不断变化，但这些变化能够很容易地满足，因为软件是灵活的。

事实：软件需求确实是经常变化的，但这些变化产生的影响会随着其引入的时间不同而不同。如果我们很注重早期的系统定义，这时的需求变化就可被很容易地适应。用户能够复审需求，并提出修改的建议，这时对成本的影响会相对较小。当在软件设计过程中才要求修改时，对成本的影响就会提高得很快。资源已经消耗了，设计框架已经建立了，这时的变化可能会引起大的改动，需要额外的资源和大量的设计修改，例如，额外的花费。实现阶段(编码和测试阶段)功能、性能、接口及其他方面的改变对成本会产生更大的影响。当软件已经投入使用后再要求修改，这时所花的代价比起较早阶段做同样修改所花的代价可能是几何级数级的增长。

开发者的神话：那些至今仍被软件开发者相信的神话是由几十年的程序设计文化培植起来的。正如我们在本章一开始就提到的，在软件的早期阶段，程序设计被看作是一门艺术。这种旧的观念和方式是很难改变的。

神话：一旦我们写出了程序并使其正常运行，我们的工作就结束了。

事实：有人说过：“越早开始写程序，就要花越长时间才能完成它”，产业界的数据[LIE80]表明在一个程序上所投入的 50％到 70％的努力是花费在第一次将程序交给用户之后。

神话：在程序真正运行之前，没有办法评估其质量。

事实：从项目一开始就可以应用的最有效的软件质量保证机制之一是正式的技术复审。软件复审(见第 8 章)是“质量的过滤器”，比起通过测试找到某类软件错误要有效得多。

神话：一个成功项目唯一应该提交的就是运行程序。

事实：运行程序仅是软件配置的一部分，软件配置包括：程序、文档和数据。文档是成功开发的基础，更重要的是，文档为软件维护提供了指导。

许多软件专业人士认识到上述这些神话是错误的。但令人遗憾的是旧的观念和方法培植了拙劣的管理和技术习惯，虽然现实情况已经需求有更好的方法。对软件现实的认识是形成软件开发的实际解决方案的第一步。

1.5 小结

软件已经成为基于计算机的系统及产品的关键组成成分。在过去 40 年中，软件已经从特定的问题解决和信息分析工具演化为一门独立的产业。但早期的“程序设计”文化和历史产生了一系列至今还存在的问题，软件已经成为计算机系统演化过程中的阻碍因素。软件是由程序、数据和文档组成。这些条目构成了软件工程过程中的配置项，软件工程的目的就是为建造高质量的软件提供一个框架。

思考题

1.1 软件在许多基于计算机的系统或产品中具有不同的特点，试举 2 到 3 个产品以及至少 1 个系统，说明软件而非硬件在其中的差别。

1.2 在五、六十年代，计算机程序设计是一门艺术，其学习环境就象在工厂当学徒。早期阶段对今天的软件开发有何影响？

1.3 很多作者已经探讨了“信息时代”的影响。试举若干实例(正面的和反面的)说明软件对当今社会的影响。参考 1.1 节中任—90 年代以前的参考文献，指出其中作者的哪些预见是正确的，哪些是错误的？

1.4 试举一个特定应用并说明(a)它属于哪一个软件应用类型(见 1.2.3 节)；(b)与该应用相关的数据内容；(C)该应用的信息确定性。

1.5 随着软件的普及，公众所冒的危险(由于软件的错误引起的)受到越来越多的关注。试举一个因为计算机程序的失败而带来巨大灾难的(对人类或经济均可)实际可能发生的场景。

1.6 精读 Internet 上的新闻组 comp.risks 的内容，并对最近正在讨论的软件带来的危险作一个总结。[另一个可选的资料来源是：Software Engineering Notes(软件工程注释)，由 ACM(theAssociation of Computing Machinery(计算机协会))出版]。

1.7 写一篇文章总结某个前沿软件应用领域的最新进展。推荐可选的领域包括：人工智能，虚拟现实、人工神经网络、高级人机界面和智能代理。

1.8　1.4 节陈述的神话随着时间的推移已经慢慢淡化了，但另外一些新神话替代了它们，试着为每一类神话增加一二个“新”神话。

推荐阅读文献及其他信息源

关于计算机软件有成千上万的书籍，大多数是探讨程序设计语言或软件应用的，但也有一部分是讨论软件本身的。关于该话题的一个非正式的探讨可在［PRE91］中找到。Negroponte 的畅销书(Being Digital，数字化，Alfred A.Knopf,Inc.,1995)阐述了计算的观点及其对 21 世纪的影响。DeMarco(Why Does

Software Cost So Much? 为什么软件的成本如此之高，Dorset House，1995)对于软件及其开发过程进行了精辟的和有远见的论述。

对计算机软件和软件工程深刻理解的基础是计算机科学，这个话题涉及面太宽，超出了本书讨论的范围。“计算机科学文献汇总”(Computer Science Bibliography Collection)中收集了近 600 个计算机科学领域的文献目录。在其中的 300 000 个参考文献中几乎每一个都与软件工程相关：

http：//www.pilgrim.umass.edu/pub/misc/bibliographies/index.html

“计算机科学指导”(the World Guide to Computer Science)中包含了很多大学的研究机构在计算机科学领域取得的进展：

http：//www.worldwidenews.net/subjects/htm

软件方面的术语词，包括缩略词可在以下网站上找到：

http：//dxsting.cern.ch/sting/glossary.html

“工程的国际互联”(Internet Connection for Engineering)给出了国际上工程程序正在研究的方向索引：

http：//www.englib.cornell.edu/ice/ice-index.html

① 这种状况目前正在迅速改变。面向对象技术的广泛使用已导致了软件构件的生产，本书第 19 至 29 章将有详细讨论。

② 本书第四部分介绍了面向对象技术的使用和它们对构件复用形成的影响。

① 该术语由Michigan大学的DanielTiechrow教授于 1989 年 4 月，在SwitzerlndGeneva提出。

第 2 章　过程

软件过程是过去十年中人们关注的焦点。但准确讲什么是软件过程呢？在本书中，我们定义软件过程为建造高质量软件需要完成的任务的框架。“过程”与软件工程同义吗？答案是“是也不是”。一个软件过程定义了软件开发中采用的方法，但软件工程还包含该过程中应用的技术——技术方法和自动化工具。

更重要的一点，软件工程是有创造力、有知识的人在定义好的、成熟的软件过程框架中进行的。本章的目的就是探讨软件过程的研究现状，并为在本书以后的章节中更详细地讨论关于管理和技术方面的话题提供指导。

2.1 软件工程——一种层次化技术

虽然有很多作者都给出了软件工程的定义，但 Fritz Bauer[NAU69]在 NATO 会议上给出的定义仍是进一步展开讨论的基础：

软件工程　是为了经济地获得可靠的和能在实际机器上高效运行的软件而建立和使用的好的工程原则。

几乎每一个读者都忍不住想在这个定义上增加点什么。它没有提到软件质量的技术层面，也没有直接谈到用户满意度或按时交付产品的要求，它忽略了测度和度量的重要性，甚至没有阐明一个成熟的过程的重要性。但 Bauer 的定义给我们提供了一个基线。什么是可以应用到计算机软件开发中的“好的工程原则”？我们如何“经济地”建造软件使得其可靠性高？如何才能创建出能够在多个、而不是一个不同的实际机器上“高效运行”的程序？这些都是进一步挑战软件工程师的问题。

IEEE[IEE93]给出了一个更加综合的定义：

软件工程：(1)将系统化的、规范的、可度量的方法应用于软件的开发、运行和维护的过程，即将工程化应用于软件中。(2)(1)中所述方法的研究。

2.1.1 过程、方法和工具

软件工程是一种层次化的技术(如图 2－1 所示)。任何工程方法(包括软件工程)必须以有组织的质量保证为基础。全面的质量管理和类似的理念刺激了不断的过程改进，正是这种改进导致了更加成熟的软件工程方法的不断出现。支持软件工程的根基就在于对质量的关注。

软件工程的基层是过程层。软件工程过程是将技术层结合在一起的凝聚力，使得计算机软件能够被合理地和及时地开发出来。过程定义了一组关键过程区域的框架(KPAs)[PAY93]，这对于软件工程技术的有效应用是必须的。关键过程区域构成了软件项目的管理控制的基础，并且确立了上下各区域之间的关系，其中规定了技术方法的采用、工程产品(模型、文档、数据、报告、表格等)的产生、里程碑的建立、质量的保证及变化的适当管理。

软件工程的方法层提供了建造软件在技术上需要“如何做”。方法涵盖了一系列的任务：需求分析、设计、编程、测试和维护。软件工程方法依赖于一组基本原则，这些原则控制了每一个技术区域，且包含建模活动和其他描述技术。

软件工程的工具层对过程和方法提供了自动的或半自动的支持。当这些工具被集成起来使得一个工具产生的信息可被另外一个工具使用时，一个支持软件开发的系统就建立了，称为计算机辅助软件工程(CASE)。CASE 集成了软件、硬件和一个软件工程数据库(一个仓库，其中包含了关于分析、设计、编程和测试的重要信息)，从而形成了一个软件工程环境，它类似于硬件的 CAD/CAE(计算机辅助设计/工程)。

2.1.2 软件工程的一般视图

工程是对技术(或社会)实体的分析、设计、建造、验证和管理。抛开要工程化的实体，下列问题是必须首先回答的：

- 要解决的问题是什么？
- 要用于解决该问题的实体具有什么特点？
- 如何实现该实体(解决方案)？
- 如何建造该实体？
- 采用什么方法去发现该实体设计和建造过程中产生的错误？
- 当该实体的用户要求修改、适应和增强时，如何支持这些活动？

本书全文只针对一个实体——计算机软件。要适当地建造一个软件，软件开发过程是必须定义的。本节给出了软件过程的一般性特点，本章的以后几节进一步阐述了特定的过程模型。

如果不考虑应用领域、项目规模和复杂性，与软件工程相关的工作可分为三个一般的阶段。每一个阶段回答了上述的一个或几个问题。

定义阶段集中于“做什么”。即在定义过程中，软件开发人员试图弄清楚要处理什么信息，预期完成什么样的功能和性能，希望有什么样的系统行为，建立什么样的界面，有什么设计约束，以及定义一个成功系统的确认标准是什么。即定义系统和软件的关键需求。虽然在定义阶段采用的方法取决于使用的软件工程范型(或范型的组合)，但在某种程度上均有三个主要任务：系统或信息工程(见第 10 章)，软件项目计划(第 3 章到第 7 章)，和需求分析(第 11，12 和 20 章)。

开发阶段集中于“如何做”。即在开发过程中，软件工程师试图定义数据如何结构化，功能如何转换为软件体系结构，过程细节如何实现，界面如何表示，设计如何转换成程序设计语言(或非过程语言)，测试如何执行。在开发阶段采用的方法可以不同，但都有三个特定的任务：软件设计(第 14、15 和 21 章)，代码生成，和软件测试(第 16、17 和 22 章)。

维护阶段集中于“改变”，与以下几种情况相关：纠正错误；随着软件环境的演化，而要求的适应性修改；及由于用户需求的变化而带来的增强性修改。维护阶段重复定义和开发阶段的步骤，但却是在已有软件的基础上发生的。在维护阶段可能遇到四类修改要完成：

纠错：即使有最好的质量保证机制，用户还是有可能发现软件中的错误。纠错性维护是为了改正错误而使软件发生变化。

适应：随着时间的推移，原来的软件被开发的环境(如 CPU、操作系统、商业规则、外部产品特征)可能发生了变化。适应性维护是为了适应这些外部环境的变化而修改软件。

增强：随着软件的使用，用户可能认识到某些新功能会产生更好的效益。完善性维护是由于扩展了原来的功能需求而修改软件。

预防：计算机软件由于修改而逐渐退化，因此，预防性维护，常常称为软件再工程，就必须实行，以使软件能够满足其最终用户的要求。本质上讲，预防性维护对计算机程序的修改，可使得能够更好地纠错软件的错误，提高软件的适应性和增强软件的需求。

今天，“老化的软件工厂”(见 1.1.2 节)已经迫使许多公司不得不实行软件再工程(第 27 章)。从一个全面的观点来看，软件再工程常常被视为商业过程再工程的一个组成部分。

还有很多保护性活动可用来补充在软件工程的一般视图中(本节)所阐述的阶段和相关步骤。典型的活动包括：

- 软件项目追踪和控制。
- 正式的技术复审。
- 软件质量保证。
- 软件配置管理。
- 文档的准备和产生。
- 可复用管理。
- 测试。
- 风险管理。

这些活动贯穿于整个软件过程中，将在本书的第 2 和第 5 部分探讨。

2.2 软件过程

一个软件过程可以表示为图 2—2 所示的形式。一个公共过程框架，是通过定义若干框架活动来建立的，如果不考虑其规模和复杂性这些活动适用于所有软件项目。若干任务集合——每一个集合都由软件工程工作任务、项目里程碑、软件工程产品和交付物以及质量保证点组成——使得框架活动适应于不同软件项目的特征和项目组的需求。最后是保护性活动——如软件质量保证，软件配置管理，和测试度(这些内容在以后的章节中还要详细讨论)——它们贯穿于整个过程模型之中。保护性活动独立于任何一个框架活动，且贯穿于整个过程。

近年来，“过程成熟度”成为人们关注的焦点(参见文献［PAU93］)。软件工程研究所(SEI)提出了一个综合模型，定义了当一个组织达到不同的过程成熟

度时应该具有的软件工程能力。为了确定一个组织目前的过程成熟度，SEI 使用了一个评估调查表和一个五分的等级方案，这个等级方案与能力成熟度模型［PAU93］一致，该模型定义了在不同的过程成熟度级别上所需要的关键活动。SEI 的模型提供了衡量一个公司软件工程实践的整体有效性的方法，且建立了五级的过程成熟度级别，其定义如下：

第一级：初始级——软件过程的特征是无序的，有时甚至是混乱的。几乎没有过程定义，成功完全取决于个人的能力。

第二级：可重复级——建立了基本的项目管理过程，能够追踪费用、进度和功能。有适当的必要的过程规范，使得可以重现以前类似项目的成功。

第三级：定义级——用于管理和工程活动的软件过程已经文档化、标准化，并与整个组织的软件过程相集成。所有项目都使用文档化的、组织认可的过程来开发和维护软件。本级包含了第二级的所有特征。

第四级：管理级——软件过程和产品质量的详细度量数据被收集，通过这些度量数据，软件过程和产品能够被定量地理解和控制。本级包含了第三级的所有特征。

第五级：优化级——通过定量的反馈，进行不断的过程改进，这些反馈来自于过程或通过测试新的想法和技术而得到。本级包含了第四级的所有特征。

SEI 定义的这五个级别是根据 SEI 基于 CMM 的评估调查表得到的反馈而产生的结果。调查表的结果被精确化而得到单个的数字等级，表示了一个组织的过程成熟度。

SEI 将关键过程区域(KPAs)与每一个成熟度级别联系起来。KPAs 描述了要达到某一特定级别必须满足的软件工程功能(如软件项目计划，需求管理)，每一个 KPA 均通过标识下列特征来描述：

• 目标——KPA 要达到的总体目的。

• 约定——要求(组织必须遵守的)。这些要求是要达到目标就必须满足的，或是提供了是否实现目标的考察标准。

• 能力——使得组织能够满足约定要求的那些事物(组织的或技术的)。

• 活动——为了完成 KPA 的功能所需要的特定任务。

• 监控实现的方法——活动在实现过程中被监控的方式。

• 验证实现的方法——KPA 的活动能够被验证的方式。成熟度模型中定义了 18 个 KPAs(每一个都用上述的结构来描述)，它们映射到过程成熟度的不同级别。下面给出了在每个过程成熟度级别上应该实现的 KPAs(注意 KPAs 是叠加的。例如，过程成熟度第三级包含了第二级的所有 KPAs 加上第三级特有的 KPAs)：

过程成熟度第二级

软件配置管理

软件质量保证

软件子合同管理

软件项目追踪和查错

软件项目计划

需求管理

过程成熟度第三级

详细复审

组内协调

软件产品工程

集成的软件管理

培训程序

组织的过程定义

组织的过程焦点

过程成熟度第四级

软件质量管理

定量的过程管理

过程成熟度第五级

过程变化管理

技术变化管理

错误预防

每一个 KPA 都由一组用于满足其目标的关键行为来定义。这些关键行为是在一个关键过程区域完全建立之前必须完成的策略、规程和活动。SEI 定义了关键

指示器：“那些关键行为或关键行为的构件，能够很好地判定一个关键过程区域的目标是否达到”。用于评估的问题被设计来探查关键指示器的存在(或缺少)。

2.3 软件过程模型

为了解决产业环境中的实际问题，一个软件工程师或一组工程师必须综合出一个开发策略，该策略能够覆盖 2.1.1 节所述的过程、方法和工具三个层次以及 2.1.2 节所讨论的一般阶段。这个策略常常被称为过程模型或软件工程范型。软件工程过程模型的选择是基于项目和应用的特点、采用的方法和工具以及需要的控制和交付的产品。在一篇关于软件过程本质的有趣文章中，L.B.S.Raccoon(参见文献[RAC95])使用了分级几何表示，作为讨论软件过程本质的基础。

所有软件开发都可看成是一个问题循环解决过程(图 2－3a)，其中包含四个截然不同的阶段：状态描述，问题定义，技术开发和方案综述。状态描述“表示了事物的当前状态”［RAC95］；问题定义标识了要解决的特定问题；技术开发通过应用某些技术来解决问题；方案综述提交结果(如文档、程序、数据、新的商业功能、新产品)给那些从一开始就需要方案的人。2.1.2 节定义的软件工程的一般阶段和步骤可以很容易地映射到这些阶段上。

上述的问题循环解决过程可以应用于软件工程的多个不同开发级别上。它可以用于考虑整个系统的宏观阶段；开发程序构件的中级阶段；甚至是代码行编写阶段。因此，可以使用分级几何表示(最早是在几何学的表示中提出的。定义一个模式，然后在连续的更小的规模上递归地应用它；一个模式套着一个模式)来提供关于过程的理想化的视图。在图 2－3b 中，问题循环解决过程的每一个阶段又包含一个相同的问题循环解决过程，该循环还可以再包含另一个问题循环解决过程(这可以一直继续下去直到一个合理的边界，对于软件而言，是代码行)。

实际上，要想像图 2－4 那样清楚地划分活动是很困难的，因为阶段内部和阶段之间的活动常常是交叉的，但这个简化的视图产生了一个重要的思想：对于一个软件项目而言，不管选择了什么过程模型，所有的阶段——状态描述，问题定义，技术开发和方案综述——在某个细节的级别上都是同时存在的。给出了递归性质的图 2－3b，上面讨论的四个阶段既可以用于一个完整应用的分析也可以用于一小段代码的生成。

Raccoon(参见文献[RAC95])建议了一种“无序模型”，它描述“软件开发是从用户到开发者到技术的一个连续过程”。随着向一个完整系统的逐步进展，上述的阶段递归地应用于用户的需求和开发者的软件技术说明形成中。

在下节中，讨论了多个不同的软件工程过程模型，其中每一个都代表了一种将本质上无序的活动有序化的企图。重要一点的是记住每个模型都具有能够帮助实际软件项目的控制及协调的特征。但在他们的核心中，这些模型又表现了无序模型的特点。

2.4 线性顺序模型

图 2-4 表明了软件工程的线性顺序模型，有时也称“传统生命周期”或“瀑布模型”，线性顺序模型提出了软件开发的系统化的、顺序的方法(虽然由 Winston Royce [ROY70] 提出的最早的瀑布模型支持带有反馈的循环，但大多数使用该过程模型的组织均把它视为是严格线性的)，从系统级开始，随后是分析、设计、编码、测试和维护。借鉴了传统的工程周期，线性顺序模型包含以下活动：

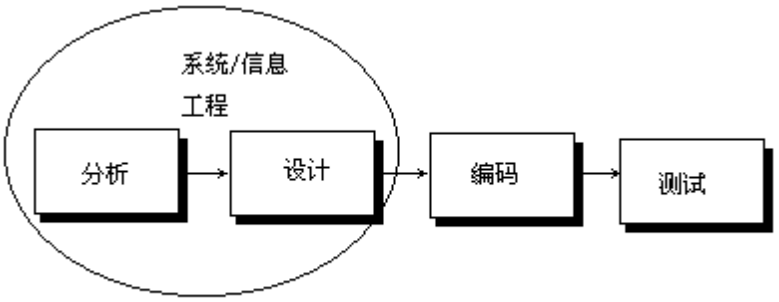


图 2-4 线性顺序模型

系统/信息工程和建模：因为软件总是一个大系统(或商业)的组成部分，所以一开始应该建立所有系统成分的需求，然后再将其中某个子集分配给软件。整个系统基础是，以软件作为其他成分如硬件、人及数据库的接口。系统工程和分析包括了系统级收集的需求，以及一小部分顶层分析和设计。信息工程包括了在战略商业级和商业领域级收集的需求。

软件需求分析：需求收集过程特别集中于软件上。要理解待建造程序的本质，软件工程师(“分析员”)必须了解软件的信息领域(见第 11 章)，以及需求的功能、行为、性能和接口。系统需求和软件需求均要文档化，并与用户一起复审。

设计：软件设计实际上是一个多步骤的过程，集中于程序的四个完全不同的属性上：数据结构、软件体系结构、界面表示及过程(算法)细节。设计过程将需求转换成软件表示，在编码之前可以评估其质量。象需求一样，设计也要文档化，并且是软件配置的一部分。

代码生成：设计必须转换成机器可读的形式。代码生成这一步就是完成这个任务的。如果设计已经表示得很详细，代码生成可以自动完成。

测试：一旦生成了代码，就可以开始程序测试。测试过程集中于软件的内部逻辑——保证所有语句都测试到，以及外部功能——即引导测试去发现错误，并保证定义好的输入能够产生与预期结果相同的输出。

维护：软件在交付给用户之后不可避免地要发生修改(一个可能的例外是嵌入式软件)。在如下情况下会发生修改：当遇到错误时；当软件必须适应外部环境的变化时(例如，因为使用新的操作系统或外设)；或者当用户希望增强功能或

性能时。软件维护重复以前各个阶段，不同之处在于它是针对已有的程序，而非新程序。

线性顺序模型是最早，也是应用最广泛的软件工程范例。但是，对于该模型的批评使得即使是最积极的支持者也开始怀疑其功效［HAN95］。在使用线性顺序模型过程中有时会遇到如下一些问题：

1.实际的项目很少按照该模型给出的顺序进行。虽然线性模型能够容许迭代，但却是间接的。结果，在项目组的开发过程中变化可能引起混乱。

2.用户常常难以清楚地给出所有需求，而线性顺序模型却要求如此，它还不能接受在许多项目的开始阶段自然存在的不确定性。

3.用户必须有耐心。程序的运行版本一直要等到项目开发晚期才能得到。大的错误如果直到检查运行程序时才被发现，后果可能是灾难性的。

4.开发者常常被不必要地耽搁。在对实际项目的一个有趣的分析中，Bradac［BRA94］发现传统生命周期的线性特征会导致“阻塞状态”，其中某些项目组成员不得不等待组内其他成员先完成其依赖的任务。事实上，花在等待上的时间可能会超过花在开发工作上的时间!阻塞状态经常发生在线性顺序过程的开始和结束。

这些问题都是真实存在的。但不管怎样，传统的生命周期范型在软件工程中仍占有肯定的和重要的位置。它提供了一个模板，使得分析、设计、编码、测试和维护的方法可以在该模板的指导下展开。传统的生命周期模型仍然是软件工程中应用最广泛的过程模型。虽然它确实有不少缺陷，但很显然它比起软件开发中随意的状态要好得多。

2.5 原型模型

常有这种情况，用户定义了软件的一组一般性目标，但不能标识出详细的输入、处理及输出需求；还有一些情况，开发者可能不能确定算法的有效性、操作系统的适应性或人机交互的形式。在这些及很多其他情况下，原型范型可能是最好的选择。

原型范型(如图 2-5 所示)从需求收集开始。开发者和用户在一起定义软件的总体目标，标识出已知的需求，并规划出进一步定义的区域。然后是“快速设计”，快速设计集中于软件中那些对用户/客户可见的部分的表示(如输入方式和输出格式)。快速设计导致原型的建造。原型由用户/客户评估，并进一步精化待开发软件的需求。逐步调整原型使其满足客户的要求，同时也使开发者对将要做的事情有更好的理解，这个过程是迭代的。

理想上，原型可以作为标识软件需求的一种机制。如果建立了可运行原型，开发者就可以在此基础上试图利用已有的程序片断或使用工具(如报表生成器、窗口管理器等)来尽快生成工作程序。

但当原型已经完成了上述目的之后，我们将如何处理它们呐？ Brooks [BRO75]给出了一个答案：

在大多数项目中，建造的第一个系统很少是可用的。它可能太慢，太大，难以使用或三者都有。没有其他选择，只能重新开始，虽然痛苦，但会得到更好的结果。建造一个经过重新设计的版本，解决了上述的问题……。当使用了新的系统概念或新技术时，你应该建造一个抛弃型的系统，因为即使是最好的计划也不可能是无所不知的，第一次就能完全正确。因此，管理上的问题不是你是否要建造一个指导系统，然后抛弃它，你必须这么做。唯一的问题是：是否需要事先计划好建造一个抛弃型系统，或是承诺要将抛弃型系统交付给用户。

原型可以作为“第一个系统”，就是 Brooks 建议我们抛弃的那一个。但这可能是一种理想化的看法。用户和开发者确实都喜欢原型范型，用户能够感受到实际的系统，开发者能够很快地建造出一些东西。但由于如下的原因原型仍存在问题：

1.用户似乎看到的是软件的工作版本，他们不知道原型只是“用口香糖和打包绳”拼凑起来的；不知道为了使原型很快能够工作，我们没有考虑软件的总体质量和长期的可维护性。当被告知该产品必须重建，才能使其达到高质量时，用户叫苦连天，会要求做“一些修改”，使原型成为最终的工作产品。如此，软件开发管理常常就放松了。

2.开发者常常需要实现上的折衷，以使原型能够尽快工作。一个不合适的操作系统或程序设计语言可能被采用，仅仅因为它是通用的和有名的；一个效率低的算法可能被使用，仅仅为了演示功能。经过一段时间之后，开发者可能对这些选择已经习惯了，忘记了它们不合适的所有原因。于是这些不理想的选择就成为了系统的组成部分。

虽然会出现问题，原型仍是软件工程的一个有效范型。关键是如何定义一开始的游戏规则，即用户和开发者两方面必须达成一致：原型被建造仅是为了定义需求，之后就该被抛弃(或至少部分抛弃)，实际的软件在充分考虑了质量和可维护性之后才被开发。

2.6 RAD 模型

快速应用开发(RAD)是一个线性顺序的软件开发模型，强调极短的开发周期。RAD模型是线性顺序模型的一个“高速”变种，通过使用基于构件的建造方法获得了快速开发。如果需求理解得很好，且约束了项目范围[①]， RAD过程使得一个开发组能够在很短时间内(如 60 到 90 天)创建出“功能完善的系统”[MAR91]。RAD方法主要用于信息系统应用软件的开发，它包含如下几个开发阶段[KER94]：

业务建模：业务活动中的信息流被模型化，以回答如下问题：什么信息驱动业务流程？生成什么信息？谁生成该信息？该信息流往何处？谁处理它？业务建模将在第 10 章详细描述。

数据建模：业务建模阶段定义的一部分信息流被精化，形成一组支持该业务所需的数据对象。标识出每个对象的特征(称为属性)，并定义这些对象间的关系。数据建模将在第 12 章讨论。

处理建模：数据建模阶段定义的数据对象变换成为要完成一个业务功能所需的信息流。创建处理描述以便增加、修改、删除或获取某个数据对象。

应用生成：RAD 假设使用第四代技术(见 2.9 节)。RAD 过程不是采用传统的第三代程序设计语言来创建软件，而是复用已有的程序构件(如果可能的话)或是创建可复用的构件(如果需要的话)。在所有情况下，均使用自动化工具辅助软件建造。

测试及反复：因为 RAD 过程强调复用，许多程序构件已经是测试过的，这减少了测试时间。但新构件必须测试，所有接口也必须测到。

RAD 过程模型如图 2-6 所示。很显然，加之于一个 RAD 项目上的时间约束需要有“一个可伸缩的范围”［KER94］。如果一个商业应用能够被模块化，使得其中每一个主要功能均可以在不到三个月时间内完成(使用上述的方法)，它就是 RAD 的一个候选件。每一个主要功能可由一个单独的 RAD 组来实现，最后再集成起来形成一个整体。

像所有其他过程模型一样，RAD 方法也有其缺陷：

- 对于大型的、但可伸缩的项目，RAD 需要足够的人力资源以创建足够的 RAD 组。

- RAD 要求承担必要的快速活动的开发者和用户在一个很短的时间框架下完成一个系统。如果两方中的任何一方没有完成约定，都会导致 RAD 项目失败。

并非所有应用软件都适合使用 RAD。如果一个系统难以被适当地模块化，那么建造 RAD 所需的构件就会有问题；如果高性能是一个指标，且该指标必须通过调整接口使其适应系统构件才能赢得，RAD 方法就有可能失败了；RAD 不适合技术风险很高的情况，当一个新应用要采用很多新技术，或当新软件要求与已有计算机程序有高可互操作性时，这种情况就会发生。

RAD 强调可复用程序构件的开发。可复用性(见第 26 章)是对象技术的基础(本书的第四部分)，在 2.7.3 节讨论的构件组装过程模型中还会提到。

2.7 演化软件过程模型

人们已经越来越认识到软件就象所有复杂系统一样要经过一段时间的演化(参见文献 [GIL88])。业务和产品需求随着开发的发展常常发生改变，想找到最终产品的一条直线路径是不可能的；紧迫的市场期限使得难于完成一个完善的软件产品，但可以先提交一个有限的版本以对付竞争的或商业的压力；只要核心产品或系统需求能够被很好地理解，而产品或系统的细节部分可以进一步定义。在这些情况及其他类似情况下，软件工程师需要一个过程模型，以便能明确设计，同时又能适应随时间演化的产品的开发。

线性顺序模型(见 2.4 节)是支持直线开发，本质上，瀑布方法是假设当线性序列完成之后就能够交付一个完善的系统。原型模型(见 2.5 节)目的是帮助用户(或开发者)理解需求，总体上讲，它并不是交付一个最终产品系统。而软件的变化特征在这些传统的软件工程范型中都没有加以考虑。

演化模型是利用一种迭代的思想方法。它的特征是使软件工程师渐进地开发，逐步完善的软件版本。

2.7.1 增量模型

增量模型融合了线性顺序模型的基本成分(重复地应用)和原型的迭代特征。如图 2-7 所示，增量模型采用随着日程时间的进展而交错的线性序列。每一个线性序列产生软件的一个可发布的“增量”(参见文献 [McDE93])。例如，使用增量范型开发的字处理软件，可能在第一个增量中发布基本的文件管理、编辑和文档生成功能；在第二个增量中发布更加完善的编辑和文档生成能力；第三个增量实现拼写和文法检查功能；第四个增量完成高级的页面布局功能。应该注意：任何增量的处理流程均可以结合进原型范型。

当使用增量模型时，第一个增量往往是核心的产品，即实现了基本的需求，但很多补充的特性(其中一些是已知的，另外一些是未知的)还没有发布。核心产品交用户使用(或进行更详细的复审)，使用和/或评估的结果是下一个增量的开发计划。该计划包括对核心产品的修改，使其能更好地满足用户的需要，并发布一些新增的特点和功能。这个过程在每一个增量发布后不断重复，直到产生最终的完善产品。

增量过程模型，像原型(见 2.5 节)和其他演化方法一样，具有迭代的特征。但与原型不一样，增量模型强调每一个增量均发布一个可操作产品。早期的增量是最终产品的“可拆卸”版本，但它们确实提供了给用户服务的功能，并且提供了给用户评估的平台。

增量开发是很有用的，尤其是当配备的人员不能在为该项目设定的市场期限之前实现一个完全的版本时。早期的增量可以由较少的人员实现。如果核心产品很受欢迎，可以增加新的人手(如果需要的话)实现下一个增量。此外，增量能够有计划地管理技术风险，例如，系统的一个重要部分需要使用正在开发的且发布时间尚未确定的新硬件，有可能计划在早期的增量中避免使用该硬件，这样，就可以先发布部分功能给用户，以免过分地延迟系统的问世时间。

2.7.2 螺旋模型

螺旋模型，最早是由 Boehm［BOE88］提出来的，是一个演化软件过程模型，它将原型的迭代特征与线性顺序模型中控制的和系统化的方面结合起来，使得软件的增量版本的快速开发成为可能。在螺旋模型中，软件开发是一系列的增量发布。在早期的迭代中，发布的增量可能是一个纸上的模型或原型；在以后的迭代中，被开发系统的更加完善的版本逐步产生。

螺旋模型被划分为若干框架活动，也称任务区域(本节描述的螺旋模型是 Boehm 提出的模型的变种。最初的螺旋模型的相关信息参见［BOE88］。一般情况下，有三到六个任务区域，图 2-8 刻划了包含六个任务区域的螺旋模型：

- 用户通信——建立开发者和用户之间有效通信所需要的任务。
- 计划——定义资源、进度及其他相关项目信息所需要的任务。
- 风险分析——评估技术的及管理的风险所需要的任务。
- 工程——建立应用的一个或多个表示所需要的任务。
- 建造及发布——建造、测试、安装和提供用户支持(如文档及培训)所需要的任务。
- 用户评估——基于在工程阶段产生的或在安装阶段实现的软件表示的评估，获得用户反馈所需要的任务。

每一个区域均含有一系列适应待开发项目的特点的工作任务。对于较小的项目，工作任务的数目及其形式化程度均较低。对于较大的、关键的项目，每一个任务区域包含较多的工作任务，以得到较高级别的形式。在所有情况下，均需应用 2.2 节提到的保护性活动(如软件配置管理和软件质量保证)。

随着演化过程的开始，软件工程项目组按顺时针方向沿螺旋移动，从核心开始。螺旋的第一圈可能产生产品的规格说明；再下面的螺旋可能用于开发一个原型；随后可能是软件的更完善的版本。经过计划区域的每一圈是为了对项目计划进行调整，基于从用户评估得到的反馈，调整费用和进度。此外，项目管理者可以调整完成软件所需的计划的迭代次数。

与传统的过程模型不同，它不是当软件交付时就结束了，螺旋模型能够适用于计算机软件的整个生命周期。图 2-9 中定义了项目入口点的轴线。沿轴线放置的每一个小方块都代表了一个不同类型项目的开始点。一个“概念开发项目”从螺旋的核心开始一直持续到概念开发结束(沿着中心阴影区域限定的螺旋线进行多次迭代)。如果概念被开发成真正的产品，过程从第二个小方块(“新产品开发项目”入口点)开始，一个新的开发项目启动了。新产品的演化是沿着比中心区

域略浅的阴影区域所限定的螺旋线进行若干次迭代。其他类型的项目也遵循类似的过程。

本质上，具有上述特征的螺旋是一直运转的直到软件退役。有时这个过程处于睡眠状态，但任何时候出现了改变，过程都会从合适的入口点开始(如产品增强)。

对于大型系统及软件的开发来说，螺旋模型是一个很现实的方法。因为软件随着过程的进展演化，开发者和用户能够更好地理解和对待每一个演化级别上的风险。螺旋模型使用原型作为降低风险的机制，但更重要的是，它使开发者在产品演化的任一阶段均可应用原型方法。它保持了传统生命周期模型中系统的、阶段性的方法，但将其并进了迭代框架，更加真实地反映了现实世界。螺旋模型要求在项目的所有阶段直接考虑技术风险，如果应用得当，能够在风险变成问题之前降低它的危害。

但像其他范型一样，螺旋模型也不是包治百病的灵丹妙药。它可能难以使用户(尤其在合同情况下)相信演化方法是可控的；它需要相当的风险评估的专门技术，且其成功依赖于这种专门技术。如果一个大的风险未被发现和管理，毫无疑问会出现问题；最后，该模型本身相对比较新，不像线性顺序范型或原型范型那样被广泛应用。在这个重要的新范型的功效能够被完全确定之前，还需要经历若干年的时间。

2.7.3 构件组装模型

对象技术(第 19 章)为软件工程的基于构件的过程模型提供了技术框架。面向对象范型强调了类的创建，类封装了数据和用于操纵该数据的算法。如果经过合适的设计和实现，面向对象的类可以在不同的应用及基于计算机的系统结构中复用。

构件组装模型(如图 2-10 所示)融合了螺旋模型的许多特征。它本质上是演化的(参见文献 [NIE92])，支持软件开发的迭代方法。但是，构件组装模型是利用预先包装好的软件构件(有时称为“类”)来构造应用程序的。

开发活动从候选类的标识开始。这一步通过检查将被应用程序操纵的数据及用于实现该操纵的算法来完成(这是对类定义的一个简单的描述，更详细的讨论见第 19 章)。相关的数据和算法封装成一个类。

以前的软件工程项目中创建的类(在图 2-10 中称为构件)被存储在一个类库或仓库中(第 29 章)。一旦标识出候选类，就可以搜索该类库，确认这些类是否已经存在。如果已经存在，就从库中提取出来复用。如果一个候选类在库中并不存在，就采用面向对象方法(第 20～22 章)开发它。之后就可以利用从库中提取出来的类以及为了满足应用程序的特定要求而建造的新类，来构造待开发应用程序的第一个迭代。过程流程而后又回到螺旋，并通过随后的工程活动最终再进入构件组装迭代。

构件组装模型导致软件复用，而可复用性给软件工程师提供了大量的可见的益处。基于可复用性的研究，QSM 联合公司的报告称：构件组装降低了 70％的开发周期时间；84％的项目成本；相对于产业平均指数 16.9，其生产率指数为 26.2。虽然这些结果依赖于构件库的健壮性，但毫无疑问构件组装模型给软件工程师提供了意义深远的好处。

2.7.4 并发开发模型

并发开发模型，有时也称并发工程，David 和 Sitaram [DAV94] 是这样描述它的：

试图根据传统生命周期的主要阶段来追踪项目的状态的项目管理者是根本不可能了解其项目的状态的。这就是使用过于简单的模型追踪非常复杂的活动的示例。注意，虽然一个项目正处在编码阶段，但同时可能有一些项目组人员在参与涉及开发的多个阶段的活动，例如，……项目组人员在写需求、在设计、在编码、在测试和集成测试，所有这些活动可能在同时进行。Humphrey 和 Kellner [HUM89，KEL89] 提出的软件工程过程模型表达了这种任一阶段的活动之间存在的并发性。 Kellner 最近的工作中 [KEL91] 使用了状态图(一个加工的状态的表示法，见第 15 章)来表示与一个特定事件(如，在开发后期需求的一个修改)相关的活动之间存在的并发关系，但它不能捕获到贯穿于一个项目中所有软件开发和管理活动的大量并发……。大多数软件开发过程模型均为时间驱动的；越到模型的后端，就越到开发过程的后一阶段。而一个并发过程模型是由用户要求、管理决策和结果复审驱动的。

并发过程模型可以被大致表示为一系列的主要技术活动、任务及它们的相关状态。例如，螺旋模型(见 2.7.2 节)定义的工程活动(任务区域)是通过执行下列任务来完成的：原型和/或分析建模，需求说明，以及设计(应该注意到：分析和设计是非常复杂的任务，需要更进一步的讨论。本书的第三部分和第四部分给出了关于它们的更详细的探讨)。图 2-11 给出了并发过程模型中一个活动的图形表示。该活动(分析)在任一给定时间可能处于任一状态(状态是行为的某个外部可见的模式)。同样的，其他活动(如设计或用户通信)也能够用类似方式来表示。所有活动并发存在，但处于不同的状态。例如，在项目开发早期，用户通信活动(图中并未显示)已经完成它的第一次迭代，并处于“等待修改”状态。而分析活动(当最初的用户通信活动完成时，它正处于“开始”状态)现在则转移到“开发”状态。如果用户表示必须做某些需求上的修改，那么分析活动就从“开发”状态转移到“等待修改”状态。

并发过程模型定义了一系列事件，对于每一个软件工程活动，它们触发从一个状态到另一个状态的转移。例如，在设计的早期阶段，发现了分析模型中的一个不一致，这产生了事件“分析模型修改”，该事件触发了分析活动从“开始”状态转移到“等待修改”状态。

并发过程模型常常被用于作为客户机/服务器[①]应用(第 28 章)的开发范型。一个客户机/服务器系统由一组功能构件组成。当应用于客户机/服务器系统时，

并发过程模型在两维上定义活动［SHE94］：一个系统维和一个构件维。系统维包含三个活动：设计、组装和使用。构件维包含两个活动：设计和实现。并发提供两种方式得到：(1)系统维和构件维活动同时发生，并能使用上述的面向状态方法建模；(2)一个典型的客户机/服务器应用系统是通过多个构件实现的，其中每个构件均可以并发地设计和实现。

实际上，并发过程模型可用于所有类型的软件开发，并能够提供关于一个项目的当前状态的准确视图。该模型不是将软件工程活动限定为一个顺序的事件序列，而是定义了一个活动网络。网络上的每一个活动均可与其他活动同时发生。在一个给定的活动中或活动网络中其他活动中产生的事件将触发一个活动中的状态的转移。

2.8 形式化方法模型

形式化方法模型包含了一组活动，它们带来了计算机软件用数学说明描述的方法。形式化方法使得软件工程师能够通过采用一个严格的、数学的表示体系来说明、开发和验证基于计算机的系统。这种方法的一个变种，称为净室软件工程［MIL87，DYE92］，目前已被一些软件开发组织采用。

当在开发中使用形式化方法时(第 24 章和 25 章)，它们提供了一种机制，能够消除使用其他软件工程范型难以克服的问题。二义性、不完整性和不一致性能被更容易地发现和纠正——不是通过专门的复审，而是通过数学分析。当在设计中使用形式化方法时，它们能作为程序验证的基础，从而使得软件工程师能够发现和纠正在其他情况下发现不了的错误。

形式化方法模型虽然不是主流的方法，但可以产生正确的软件。不过，它在商业环境中的可用性还需要考虑：

1.形式化模型的开发目前还很费时和昂贵。

2.因为很少有软件开发者具有使用形式化方法所需的背景知识，所以尚需多方面的进行培训。

3.难以使用该模型作为与对其一无所知的用户进行通信的机制。

尽管存在这些顾虑，但形式化方法可能会赢得一批拥护者，例如那些必须建造安全的关键软件(如航空电子及医疗设备的开发者)的软件开发者，以及那些如果发生软件错误会遭受严重的经济损失的开发者。

2.9 第四代技术

术语“第四代技术”(4GT)包含了一系列的软件工具，它们都有一个共同点：能使软件工程师在较高级别上说明软件的某些特征。之后工具根据开发者的说明

自动生成源代码。毫无疑问软件在越高的级别上被说明，就能越快地建造出程序。软件工程的 4GT 范型的应用关键在于说明软件的能力——它用一种特定的语言来完成或者以一种用户可以理解的问题描述方法来描述待解决问题的图形来表示。

目前，一个支持 4GT 范型的软件开发环境包含如下部分或所有工具：数据库查询的非过程语言，报告生成器，数据操纵，屏幕交互及定义，以及代码生成；高级图形功能；电子表格功能。最初，上述的许多工具仅能用于特定应用领域，但今天，4GT 环境已经扩展，能够满足大多数软件应用领域的需要。

像其他范型一样，4GT 也是从需求收集这一步开始。理想情况下，用户能够描述出需求，而这些需求能被直接转换成可操作原型。但这是不现实的，用户可能不能确定需要什么；在说明已知的事实时，可能出现二义性；可能不能够或是不愿意采用一个 4GT 工具可以理解的形式来说明信息。因此，其他范型中所描述的用户——开发者对话方式在 4GT 方法中仍是一个必要的组成部分。

对于较小的应用软件，使用一个非过程的第四代语言(4GL)有可能直接从需求收集过渡到实现。但对于较大的应用软件，就有必要制订一个系统的设计策略，即使是使用 4GL。对于较大项目，如果没有很好地设计，即使使用 4GT 也会产生不用任何方法来开发软件所遇到的同样的问题(低的质量、差的可维护性、难以被用户接受)。

应用 4GL 的生成功能使得软件开发者能够以一种方式表示期望的输出，这种方式使得可以自动生成产生该输出的代码。很显然，相关信息的数据结构必须已经存，在且能够被 4GL 访问。

要将一个 4GT 生成的功能变成最终产品，开发者还必须进行测试，写出有意义的文档，并完成其他软件工程范型中同样要求的所有集成活动。此外，采用 4GT 开发的软件还必须考虑维护是否能够迅速实现。

像其他所有软件工程范型一样，4GT 模型也有优点和缺点。支持者认为它极大地降低了软件的开发时间，并显著提高了建造软件的生产率。反对者则认为目前的 4GT 工具并不比程序设计语言更容易使用，这类工具生成的结果源代码是“低效的”，并且使用 4GT 开发的大型软件系统的可维护性是令人怀疑的。

两方的说法中都有某些对的地方，这里我们对 4GT 方法的目前状态进行一个总结：

1.在过去十余年中，4GT 的使用发展得很快，且目前已成为适用于多个不同的应用领域的方法。与计算机辅助软件工程(CASE)工具和代码生成器结合起来，4GT 为许多软件问题提供了可靠的解决方案。

2.从使用 4GT 的公司收集来的数据表明：在小型和中型的应用软件开发中，它使软件的生产所需的时间大大降低，且使小型应用软件的分析和设计所需的时间也降低了。

3.不过，在大型软件项目中使用 4GT，需要同样的甚至更多的分析、设计和测试(软件工程活动)才能获得实际的时间节省，这主要是通过编码量的减少赢得的。

总结一下，第四代技术已经成为软件开发的一个重要方法。当与构件组装方法(见 2.7.3 节)结合起来时，4GT 范型可能成为软件开发的主流方法。

2.10 过程技术

前面几节讨论的过程模型必须适合于软件项目组的使用。为了满足这点，开发了过程技术工具以帮助软件组织分析它们当前的过程，组织工作任务，控制和监管进度，以及管理技术质量(参见文献［BAN95］)。

过程技术工具使得软件组织能够建造一个自动模型，该模型包含 2.2 节讨论的通用过程框架、任务集合及保护性活动。该模型一般表示成一个网络，对其加以分析，就能够确定典型的工作流程，考查可能导致降低开发时间和成本的可选的过程结构。

一旦创建了一个可接受的过程，就可以使用其他过程技术工具来分配、监管、甚至控制过程模型中定义的所有软件工程任务。软件项目组的每一个成员均能使用这些工具产生一个清单，包括：要完成的工作任务，要开发的工作产品，以及要实现的质量保证活动。过程技术工具还能用于协调适合某一特定工作任务的其他计算机辅助软件工程工具(第 29 章)的使用。

2.11 产品和过程

如果过程很弱，最终产品不可避免会出问题。但过分依赖过程也是很危险的。在一篇短文中，Margaret Davis［DAV95］评论了产品和过程的二元性：

大约每隔 5 至 10 年，软件界就会重定义“问题”，将其焦点从产品转移到过程。这样，我们在结构化程序设计语言(产品)之后有了结构化分析方法(过程)，之后又有了数据封装(产品)，再后是目前的重点——软件工程研究所的软件开发能力成熟度模型(过程)。

正如钟摆的自然倾向是停在两个极端之间的中间点，软件界的焦点也是在不断转移，因为当上一次摆动停止后，就要加新的力。这些摆动是有害的，因为它们彻底改变了完成工作的方法，使普通的软件开发人员迷失方向，更不用说要很好地使用它了。这些摆动也不能解决“问题”，因为它们注定是要失败的，只要产品和过程变成是二分的而不再是二元的。

当观察到的矛盾无法用一个权威的理论或另一种方式完全解释时。科学界率先提出了二元性的概念，光的二元特性——似乎同时是粒子和波，从本世纪 20 年代 Louis de Brogile 提出时起就已经被广泛接受了。我相信软件及其开发表

现出了产品和过程之间的一个基本的二元性。当你仅仅从过程或是仅仅从产品角度来看，你永远也不可能得到或理解整个软件，如它的范围、使用、含义和价值。

所有人类的活动可能都是一个过程，但我们每一个人从其中得到了自我价值的体现，这些活动产生的一个表示或示例，或可被很多人反复使用，或可用于某些其他未考虑到的背景上。即，我们的产品能被自己或他人复用，从其中我们得到了一种满足感。

因此，在软件开发中迅速普及复用的目标，可能会提高软件开发者从他们的工作中得到的满足感，也增加了接受产品和过程的二元性的紧迫感。仅从产品或仅从过程考虑一个可复用的事物，或者会模糊了它的使用范围和方式，或者会隐藏了某个事实：每一次使用它均会产生新的产品，该产品反过来又可用作某个其他的软件开发活动的输入。仅考虑其中一方面，会急剧降低复用的机会，也就失去了提高工作满足感的机会。

人们从创造的过程中得到了从最终的产品中同样的甚至更多的满足感。一个艺术家从挥笔的过程和完成的画中会得到同样的享受。一个作家从冥思苦想一个合适的比喻和写完的书中会得到同样的享受。一个有创造性的软件专业人员也会从过程和最终的产品中得到同样的满足感。

软件人员的工作随着时间的推移会发生改变。产品和过程的二元性是一个重要的因素，使得当从程序设计最终转移到软件工程的过程中，能够一直保持人的创造力。

2.12　小结

软件工程在计算机软件的开发中集成了过程、方法和工具。本章给出了若干不同的软件工程过程模型，每一个都有其优点和弱点，但它们均具有一系列共同的一般阶段。使得我们能够实现在本书的其余部分将要一一探讨的过程的原则、概念和方法。

思考题

2.1　图 2-1 中将三个软件工程层次置于一个名为“质量焦点”的层次之上，该焦点是指一个质量程序，如全面质量管理。做一些研究并给出一个全面质量管理程序的主要原则的概要。

2.2　是否存在一种情况，软件工程过程的一般阶段不适用？如果有，描述出来。

2.3　SEI 的能力成熟度模型是一个演化的文档。做一些研究，并确定在本书出版之后是否有新的 KPAs 增加？

2.4 “无序模型”认为一个问题循环解决过程能被用于开发的任何级别上。讨论在下列情况下你如何应用这个循环过程：(1)理解一个新的字处理产品的需求；(2)为字处理器开发一个高级拼写/文法检查构件；(3)产生一个程序模块的源代码，该模块用于确定英语语句中的主语、谓语和宾语。

2.5 你认为本章给出的哪一个软件工程范型最有效？为什么？

2.6 给出5个可以采用原型方法的软件开发项目的实例。举出两或三个难以使用原型的应用。

2.7 RAD模型一般与CASE工具结合使用。研究一些文献，并总结出一个典型的支持RAD的CASE工具。

2.8 举出一个可以采用增量模型的特定的软件项目。并给出在软件中应用该模型的某个场景。

2.9 当沿着螺旋模型的过程流路径向外移时，你认为正在开发或维护的软件发生了什么变化？

2.10 许多人相信极大地提高软件质量和生产率的唯一途径是运用构件组装。找出3或4篇关于该题目的最新文章，并给出总结。可在班里进行讨论。

2.11 用自己的话描述并发开发模型。

2.12 举出第四代技术的三个实例。

2.13 哪一个更重要——产品还是过程？

推荐阅读文献及其他信息源

软件工程的最新进展能够从一些月刊中得到，诸如IEEE Software(IEEE软件)，Computer(计算机)，和IEEE Transactions on Software Engineering(IEEE软件工程学报)。产业界的一些期刊如Application Development Trends(应用开发趋势)和Software Development(软件开发)，它们常常刊登软件工程方面的文章。每年Proceedings of the International Conference onSoftware Engineering(国际软件工程会议论文集)中都会有一个有关该专题的“摘要”(该会议由IEEE和ACM赞助)，并且在一些杂志上也对此专题进行深入的讨论，如ACM Transactions onSoftware Engineering and Methodology(ACM软件工程和方法学学报)，ACM SoftwareEngineering Notes(ACM软件工程评论)有Annals of Software Engineering(软件工程年报)。

近年来很多软件工程方面的书籍出版了。其中一些给出了关于整个过程的综述，而另外一些则深入研究了前者未探讨的若干重要的专题。下面三个文集包含了范围很广的软件工程专题：

Keyes, J.(ed.), Software Engineering Productivity Handbook, McGraw-Hill, 1993.

Marchiniak, J.J.(ed.), Encyclopedia of Software Engineering, Wiley, 1994.

McDermid, J.(ed.), Software Engineer's Reference Book, CRC Press, 1993.

Gautier(Distributed Engineering of Software, Prentice-Hall, 1996)给那些必须跨地域开发软件的组织提出了一些建议和指南。

Robert Glass 的书(Software Conflict, Yourdon Press, 1991)给出了关于软件和软件工程的有趣的和引起争议的问题论述。Pressman 和 Herron(Software Shock, Dorset House, 1991)描述了软件及其对个人、商业和政府的影响。

软件工程研究所(SEI 位于 Carnegie-Mellon 大学内)被授权负责发起一组软件工程专刊系列的创刊。来自产业界、政府和学术界的实践者都为此做出了重要的新贡献。另外, SoftwareProductivity Consortium(软件生产力协会)会定期发表关于软件工程方面的出版物。

在过去十余年中, 发布了很多软件工程的标准和规范。 IEEE Software EngineeringStandards(IEEE 软件工程标准)中包含了几乎覆盖了所有重要技术层面的多个不同的标准。ISO9000-3 指南为那些申请 ISO 9001 质量标准认证的组织提供了指导。国防部、 FAA 及其他政府和非营利机构也发布了若干软件工程标准。Fairclough(Software Engineering Guides, Prentice-Hall, 1996)给出了关于欧洲太空署(European Space Agency, ESA)发布的软件工程标准的详细参考。

在 Internet(因特网)上有大量关于软件工程和软件过程的资源。从一些技术协会和组织还可以得到更多软件工程的信息:

ACM	http://www.acm.org
European SPI Initiative	http://www.sea.uni-linz.ac.at/espiti/home.html
IEEE	http://www.computer.org
NASA Software Engineering Lab	http://fdd.gsfc.nasa.gov/seltext.html
Research Access Clearing House	http://www.rai.com/Home Page.html
Software Engineering Institute	http://www.sei.cmu.edu/
Software Engineering Lab, EPFL	http://lglWWW.epfl.ch/

Software Engineering Web Sites	http：//wwwsel.iit.nrc.ca/favourites.html
Software Productivity Center, Canada	http：//www·spc·ca/spc/
Software Productivity Consortium	http：//software.software.org/vcoe /Home.html

关于软件工程的另一个有用的信息来源是一个时事通讯季刊，可在下面的网址中找到它：

http：// www.tcse.org

软件工程研究中心(SERC)有一个技术报告系列给出最新的研究成果。可以在下面的网址上找到：

http：//www.cs.purdue.edu/tech-reports/sere-orders.html

关于经常提出的问题(FAQ)的一个概要已经放在 Internet 新闻组 comp.software-eng 上，其网址为：

http：//www.qucis.queensu.ca/Software-Engineering/reading.html

一个最新的与软件过程相关的WWW参考文献目录可在下面的网址中找到：http：//www.rspa.com.

① 这些条件很难保证。事实上，许多软件项目在开始时需求定义得很不充分的。在这种情况下，原型（见2.5节）或演化方法（见2.7节）是更好的选择。（见参考文献［REI95］）

① 在客户机/服务器应用软件中，软件功能被划分成两部分客户机端（通常是PC机）和服务器（通常是更强大的计算机）端功能，服务器典型任务是维护一个中心数据库。

第二部分　软件项目的管理

在本书的这一部分中我们主要考虑计划、组织、监管和控制软件项目所需要的管理技术。在下面的章节中，我们主要解决下列问题：

- 在一个软件项目中如何管理人员、问题和过程？
- 什么是软件度量？如何使用它们管理软件过程和过程指导下的项目？
- 什么度量能够辅助管理者评估开发的产品的质量以及使用的过程的有效性？
- 一个软件项目组如何对工作量、成本和项目时间进行可靠的评估？
- 一个组织何时应该建造软件？何时应该获取软件？何时应该请求外援？
- 采用什么技术评估来影响项目成功的风险？

• 一个软件项目管理者如何为特定项目选择合适的软件工程工作任务集？

• 如何创建一个项目进度计划？

• 如何定义质量使得软件项目组能够控制它？

• 什么是软件质量保证？如何使用它作为项目控制机制？

• 为什么正式的技术复审那么重要？

• 在计算机软件开发之中以及它被交付给用户之后如何进行变化管理？

认真回答这些问题使你能够以一种更好的方式管理软件，以便按时交付高质量的产品。

第 3 章　项目管理的概念

在关于软件项目管理的论著的序言中，　　Meiler Page-Jones [PAG85] 给出了一段引起许多软件工程顾问共鸣的陈述：

我拜访了很多商业公司(好的和不好的)，我也观察了很多数据处理的管理者(好的和不好的)。我常常恐惧地看到这些管理者徒劳地与恶梦般的项目斗争着，在根本不可能完成的最后期限下苦苦挣扎，或是在交付了使其用户极为不满的系统之后，又继续花费大量的时间去维护该系统。

Page-Jones 所描述的正是源于一系列管理和技术问题而产生的症状。不过，如果事后再剖析一下每一个项目，很有可能发现一个共同的问题：项目管理太弱。

在本章和之后的六章中，我们将探讨进行有效的软件项目管理的关键性概念。本章主要给出基本软件项目管理的概念和原则。第 4 章将阐述过程和项目度量，这是进行有效的管理决策的基础。第 5 章将讨论用于评估成本和资源需求，并建立有效的项目计划的技术。第 6 章会给出进行有效的风险监控、缓解和管理的管理活动。第 7 章会讨论定义项目任务，并建立一个可操作的项目进度计划所需的活动。最后，第 8 章和第 9 章将探讨在项目开发过程中保证质量及在应用的整个生命期中控制变化所需的技术。

3.1　管理的范围

有效的项目管理集中于三个 P 上：人员(people)、问题(problem)和过程(process)。其顺序不是任意的。任何管理者如果忘记了软件工程是人的智力密集的劳动，他就永远不可能在项目管理上得到成功；任何管理者如果在项目开发早期没有支持有效的用户通信，他有可能为错误的问题建造一个不错的解决方案。最后，对过程不在意的管理者可能冒把有效的技术方法和工具插入到真空中的风险。

3.1.1 人员

培养有创造力的、技术水平高的软件人员是从60年代起就开始讨论的话题(如［COU80，DeM87，WIT94］)。事实上，“人因素”非常重要，以致于软件工程研究所专门开发了一个人员管理能力成熟度模型(PM—CMM)，旨在“通过吸引、培养、鼓励和留住改善其软件开发能力所需的人才增强软件组织承担日益复杂的应用程序开发的能力，”(参见文献［CUR94］)。

人员管理成熟度模型为软件人员定义了以下的关键实践区域：招募，选择，业绩管理，培训，报酬，专业发展，组织和工作计划，以及团队精神/企业文化培养。在人员管理上达到较高成熟度的组织，更有可能实现有效的软件工程开发。

PM－CMM与软件能力成熟度模型(第2章)相配合，后者指导组织完成一个成熟的软件过程的创建。与软件项目的人员管理及结构相关的问题将在本章后面的几小节中详细论述。

3.1.2 问题

在进行项目计划之前，应该首先明确该项目的目的和范围，考虑可选的解决方案，定义技术和管理的约束。没有这些信息，就不可能进行合理的(准确的)成本估算；有效的风险评估；适当的项目任务划分；或是给出了意义明确的项目进度的标志的项目管理计划。

软件开发者和用户必须一起定义项目的目的和范围。在很多情况下，这项活动是作为系统工程的一部分开始的(第10章)，持续到作为软件需求分析的第一步(第11章)。目的说明该项目的总体目标，而不考虑这些目标如何实现。范围说明给出与问题相关的主要数据、功能和行为，更为重要的是，它以量化的方式约束了这些特性。

一旦了解了项目的目的和范围，就要开始考虑可选的解决方案了。虽然这一步并不讨论细节，但它使得管理者和开发者可以选择一条“最好的”途径，并根据产品交付的期限、预算的限制、可用的人员、技术接口及各种其他因素，给出项目的约束。

3.1.3 过程

软件过程(第2章)提供了一个框架，在该框架下可以建立一个软件开发的综合计划。若干框架活动适用于所有软件项目，而不在乎其规模和复杂性。若干不同的任务集合——每一个集合都由任务、里程碑、交付物以及质量保证点组成——使得框架活动适应于不同软件项目的特征和项目组的需求。最后是保护性活动——如软件质量保证，软件配置管理，和测度——它们贯穿于整个过程模型之中。保护性活动独立于任何一个框架活动，且贯穿于整个过程之中。

3.2 人员

在 IEEE 发表的一项研究中[CUR88]，三个大型的技术公司的主管工程的副总裁被问到一个成功软件项目中最重要的因素是什么。他们的回答如下：

第一位：我想如果必须在我们的环境中挑出一项最重要的因素，我必须承认它不是我们所用的工具，而是人。

第二位：一个项目成功的最重要的因素是有聪明的人……我想不出其他的因素……你为一个项目所做的最重要的事情是选择人员……软件开发组织的成功与其招募优秀人才的能力密切相关。

第三位：我在管理上唯一的准则是保证我有优秀的人员——真正优秀的人员，同时我也培养优秀的人员——我提供培养优秀人员的良好环境。

事实上，这是对软件工程过程中人的重要性的强有力的证明。不过，我们所有的人，从高级工程副总裁到低级的开发人员，常常认为人员是不成问题的。管理者表态说(就象上面给出的)人员是主要的，但他们的行为有时与说话不符。本节我们讨论参加软件过程的人员，以及如何组织他们实现有效的软件工程的方式。

3.2.1 项目参与者

参与软件过程(及每一个软件项目)的人员可以分为以下五类：

1.高级管理者：负责确定商业问题，这些问题往往对项目产生很大影响。

2.项目(技术)管理者：必须计划、刺激、组织和控制软件开发人员。

3.开发人员：负责开发一个产品或应用软件所需的专门技术人员。

4.客户：负责说明待开发软件的需求的人员。

5.最终用户：一旦软件发布成为产品，最终用户是直接与软件进行交互的人。

每一个软件项目都有上述的人员参与。为了获得很高的效率，项目组的组织必须最大限度地发挥每个人的技术和能力。这是项目负责人的任务。

3.2.2 项目负责人

项目管理是集中于人的活动，因此，能胜任开发的人员常常有可能是拙劣的项目负责人。他们完全没有管理人的技能。如 Edgemon 所说：“很不幸而且是很

经常地，似乎人们碰巧落在项目管理者的位置上，也就意外地成为了项目管理者”［EDG95］。

当我们要选择某个人来领导一个软件项目时，我们应该考虑什么呢？在一本优秀的关于领导能力的论著中，Jerry Weinberg［WEI86］给出了一个答案，他提出领导能力的MOI模型：

刺激(Motivate)：鼓励（通过“推或拉”）技术人员发挥其最大能力的一种能力。

组织(Organization)：融合已有的过程(或创造新的过程)的一种能力，使得最初的概念能够转换成最终的产品。

想法(Ideas)**或创新**(Innovation)：鼓励人们去创造，并感到有创造性的一种能力，即使他们其实必须工作在为特定软件产品或应用软件建立的约束下。

Weinberg提出：成功的项目负责人应采用一种解决问题的管理风格。即，软件项目经理应该集中于理解待解决的问题，管理新想法的交流，同时，让项目组的每一个人知道(通过言语，更重要的是通过行为)质量很重要，不能妥协。

关于一个有效的项目经理应该具有什么特点的另一个观点［EDG95］则强调了以下四种关键品质：

解决问题：一个有效的软件项目经理应该能够准确地诊断出技术的和管理的问题；系统地计划解决方案；适当地刺激其他开发人员实现解决方案；把从以前的项目中学到的经验应用到新的环境下；如果最初的解决方案没有结果，能够灵活地改变方向。

管理者的身份：一个好的项目经理必须掌管整个项目。他在必要时必须有信心进行控制，必须保证让优秀的技术人员能够按照他们的本性行事。

成就：为了提高项目组的生产率，项目经理必须奖励具有主动性和做出成绩的人。并通过自己的行为表明约束下的冒险不会受到惩罚。

影响和队伍建设：一个有效的项目经理必须能够“读懂”人；他必须能够理解语言的和非语言的信号，并对发出这些信号的人的要求做出反应。项目经理必须在高压力的环境下保持良好的控制能力。

3.2.3 软件项目组

软件开发的组织结构几乎与开发软件的组织一样多。不管怎么说，组织结构不能轻易改变。关心组织改变所产生的实际的及政策上的影响，并不是软件项目管理者的责任范围。但是，在一个新的软件项目中直接涉及到的人员的组织，则是项目管理者的职责。

下面给出为一个项目分配人力资源的若干可选方案，该项目需要 n 个人工作 k 年：

1.n 个人被分配来完成 m 个不同的功能任务，相对而言几乎没有合作的情况发生；协调是软件管理者的责任，而他可能同时还有六个其他项目要管。

2.n 个人被分配来完成 m 个不同的功能任务(m＜n)，建立非正式的“小组”；指定一个专门的小组负责人；小组之间的协调由软件管理者负责。

3.n 个人被分成 t 个小组；每一个小组完成一个或多个功能任务；每一个小组有一个特定的结构，该结构是为同一个项目的所有小组定义的；协调工作由小组和软件项目管理者共同控制。

虽然对于上述的每一种方法都可以找到其优点和缺点，但越来越多的证据表明正式的组织小组(策 3 种方法)是生产率最高的。

“最好的”小组结构取决于组织的管理风格、组里的人员数目及他们的技术水平和整个问题的难易程度。Mantei [MAN81] 提出了三种一般的小组组织方式：

民主分权式(Democratic Decentralized，DD)：这种软件工程小组没有固定的负责人。“任务协调者是短期指定的，之后就由其他协调不同任务的人取代”。问题和解决方法的确定是由小组讨论决策的。小组成员间的通信是平行的。

控制分权式(Controlled Decentralized，CD)：这种软件工程小组有一个固定的负责人，他协调特定的任务及负责子任务的二级负责人关系。问题解决仍是一个群体活动，但解决方案的实现是由小组负责人在子组之间进行划分的。子组和个人间的通信是平行的，但也会发生沿着控制层产生的上下级的通信。

控制集权式(Controlled Centralized，CC)：顶层的问题解决和内部小组协调是由小组负责人管理的。负责人和小组成员之间的通信是上下级式的。

Mantei 还给出了计划软件工程小组的结构时应该考虑的七个项目因素：

- 待解决问题的困难程度。
- 要产生的程序的规模，以代码行或者功能点来衡量。
- 小组成员需要待在一起的时间(小组生命期)。
- 问题能够被模块化的程度。
- 待建造系统所要求的质量和可靠性。
- 交付日期的严格程度。
- 项目所需要的社交性(通信)的程度。

表 3－1［MAN81］总结了项目特性对小组组织的影响。因为集中式的结构能够更快地完成任务，因此最适合处理简单问题。而分散式的小组比起个人而言能够产生更多更好的解决方案，因此，这种小组在处理复杂问题时成功的可能性更大。因为 CD 小组是集中式地解决问题，所以 CD 或 CC 小组结构能够成功地用来解决简单的问题。而 DD 结构则适于解决难度较大的问题。

因为小组的性能与必须进行的通信量成反比，所以很大的项目最好采用 CC 或 CD 结构的小组组织方式，如果子组能够很容易地协调的话。

表 3－1 项目特性对小组结构的影响［MAN81］

小组类型：	DD	CD	CC
难易程度			
高	×		
低		×	×
规模			
大		×	×
小	×		
小组生命期			
短		×	×
长	×		
模块化程度			
高		×	×
低	×		
可靠性			
高	×	×	
低			×
交付日期			
紧			×
松	×	×	
社交性			
高	×		
低		×	×

小组“在一起”的时间的长短影响小组的士气。我们发现 DD 小组结构能够产生较高的士气和工作满意度，因此适合生命期较长的小组。

DD 小组结构最适于解决模块化程度较低的问题，因为它需要更多的通信。如果有可能要较高的模块化程度(这时人们自己做自己的事情)，则 CC 或 CD 结构更加合适。

CC 和 CD 小组已被发现能够产生比 DD 小组更少的缺陷，但这与小组所采用的质量保证活动密切相关。分散式结构通常需要比集中式结构更多的时间来完成一个项目，但如果要求高社交性，它是最适合的。

Constantine [CON93] 提出了软件工程小组的四种“组织范型”：

1.封闭式范型：按照传统的权利层次来组织小组(类似 CC 小组)。这种小组在开发与过去已经做过的产品类似的软件时十分有效，但在这种封闭式范型下难以进行创新式的工作。

2.随机式范型：松散地组织小组，并依赖于小组成员个人的主动性。当需要创新或技术上的突破时，按照这种随机式范型组织的小组很有优势。但当需要“有次序的执行”才能完成工作时，这种小组组织范型就会陷入困境。

3.开放式范型：试图以一种，既具有封闭式范型的控制性，又包含随机式范型的创新性的方式来组织小组。工作的执行结合了大量的通信和基于小组一致意见的决策。开放式范型小组结构特别适于解决复杂问题，但可能不象其他类型小组那么效率高。

4.同步式范型：依赖于问题的自然划分，组织小组成员各自解决问题的片断，他们之间没有什么主动的通信需要。

从历史角度看，最早的软件小组是控制集权式(CC)结构，原来称为主程序员小组。这种结构由 Harlan Mills 首先提出，并由 Baker [BAK72] 描述出来。小组的核心是由以下人员组成的：一个高级工程师(“主程序员”)，负责计划、协调和复审小组的所有技术活动；技术人员(一般 2 到 5 个人)，执行分析和开发活动；以及一个后备工程师，支持高级工程师的活动，并能在项目进行过程中，以最小的代价取代高级工程师的工作。

主程序员可以由一个或多个专家(如电讯专家，数据库设计者)、支持人员(如技术文档写作者，行政人员)和软件资料员来担当。资料员为多个小组服务，执行以下功能：维护和控制所有软件配置(如文档，源程序，数据和磁介质)；帮助收集和格式化软件生产数据；分类和索引可复用软件模块；辅助小组进行研究、评估及文档准备。资料员的重要性不能过分强调。资料员充当了软件配置的控制者、协调者及潜在的评估者。

不考虑小组的组织，每一个项目管理者的目标都是帮助建立一个有凝聚力的小组。在“人”的论著中，DeMarco 和 Lister [DeM87] 讨论了这个问题：

我们在商业中随便使用小组这个词，把任何被分配在一起工作的一组人都称为一个“小组”。但很多这样的组并不象小组，它没有统一的对于成功的定义，没有任何可标识的团队精神，它们所缺少的是一种很珍贵的东西，我们称之为凝聚力。

一个有凝聚力的小组是一组团结紧密的人，他们的整体力量大于个体力量的总和……

一旦一个小组具有凝聚力，成功的可能性就大大提高。这个小组不可阻挡，成为成功的象征……他们不需要按照传统的方式进行管理，也不需要去激励。他们已经有了动力。

DeMarco 和 Lister 认为有凝聚力小组的成员比起一般的小组而言，具有更高的生产率和更大的动力。他们共享一个共同的目标、共同的文化，而且在很多情况下，“精英的感觉”使得他们独一无二。

3.2.4 协调和通信问题

有很多原因会使软件项目陷入困境。许多开发项目规模宏大，以致于使小组成员间的关系复杂性高、混乱、难以协调。不确定性是经常存在的，它会引起困扰项目组的一连串的改变。互操作性已成为许多系统的关键特性。新的软件必须与已有的软件通信，并遵从系统或产品所加诸的预定义约束。

现代软件的这些特性——规模、不确定性和互操作性——都是现实的问题。为了有效地处理它们，软件工程小组必须建立有效的方法，以协调参与工作的人员之间的关系。要完成这项任务，必须建立小组成员之间及多个小组之间的正式的和非正式的通信机制。正式的通信是通过“文字、会议及其他相对而言非交互的和非个人的通信渠道”来实现［KRA95］。非正式的通信则更加个人化。软件工程小组的成员在一个特别的基础上共享想法，出现问题时相互帮助，且每天互相交流。

Kraul 和 Streeter［KRA95］调查了一系列的项目协调技术，并将其分为以下几类：

正式的、非个人的方法：包括软件工程文档和交付物(如源程序)、技术备忘录、项目里程碑、进度和项目控制工具(第 7 章)、修改请求及相关文档(第 9 章)、错误跟踪报告、和中心库数据(第 29 章)。

正式的、个人间的规程：集中表现于软件工程工作中产品的质量保证活动中(第 8 章)。包括状态复审会议及设计和代码检查。

非正式的、个人间的规程：包括信息传播、问题解决及“需求和开发人员配置”的小组会议。

电子通信：包括电子邮件、电子公告栏、Web 站点以及基于视频的会议系统。

个人间的网络：与项目组之外的人进行的非正式的讨论，这些人可能有足够的经验或见解，能够帮助项目组成员。

为了评估这些项目协调技术的功效，Kraul 和 Streeter 调研了 65 个软件项目，其中包括上百个技术人员。图 3－1［KRA95］中描述的变种)表明了上述协调技术的价值及使用。在图中给出了不同的协调和通信技术的认知价值(总分为

7)及它们在项目中的使用频率之间的关系。落在回归线之上的技术“被判定为相对而言价值较高，已给出了它们被使用的总量”［KRA95］。落在线之下的技术被认为价值较低(相对于使用频率而言)。有趣的是，个人间网络(对等讨论)被评为具有最高的协调和通信价值的技术。还有一点很重要的是，早期的质量保证机制(需求和设计复审)被认为比起后期的源代码评估(代码检查)更具价值。

3.3 问题

软件项目管理者从软件工程项目一开始就面临着进退两难的局面。需要定量的估算成本和有组织的计划项目的进展，但却没有可靠的信息可以使用。对软件需求的详细分析可以提供必要的估算信息，但分析常常要花数周甚至数月的时间才能完成。更糟糕的是，随着项目的进展经常发生改变，需求可能是不固定的。不过，无论如何计划仍是“现在”就需要的!

3.3.1 软件范围

软件项目管理的第一个活动是软件范围的确定。范围是通过回答下列问题来定义的：

背景：待建造的软件如何适应于大型的系统、产品或商业的背景，在该背景下要加什么约束？

信息目标：软件要产生什么样的客户可见的数据对象(第 11 章)来作为输出使用？需要什么样的数据对象作为输入？

功能和性能：软件要执行什么样的功能使得输入数据才能变换成为输出数据？需要满足什么特殊的性能特征吗？

软件项目范围在管理层和技术层都必须是无二义性的和可理解的。对软件范围的描述必须是界定的。即，明确给出定量的数据(如并发用户数目、邮件列表的大小、允许的最大响应时间)；说明约束和/或限制(如产品成本、内存大小)；描述其他的特殊因素(如要用的算法能够很好地理解，并写成 C++程序)。

3.3.2 问题分解

问题分解，有时称为划分，是一个软件需求分析(第 11 章)的核心活动。在确定软件范围的活动中并没有完全分解问题。分解一般用于两个主要领域：(1)必须交付的功能；(2)交付所用的过程。

面对复杂的问题人类常常采用分而治之的策略。简单讲，就是将一个复杂的问题划分成若干较易处理的小问题。这是项目计划开始时所采用的策略。在估算

开始之前，范围中所描述的软件功能必须被评估和精化，以提供更多的细节。因为成本和进度估算都是面向功能的，所以某种程度的分解是很有用的。

例如，考虑我们要建造一个新的字处理产品的项目。该产品的特殊功能包括：连续的语音输入和键盘输入；高级的“自动拷贝编辑”功能；页面布局功能；自动建立索引和目录；以及其他功能。项目管理者必须首先建立软件范围描述，规定这些功能(以及其他的通用功能，如编辑、文件管理、文档生成等等)。例如，连续语音输入功能是否需要对用户进行专门的产品“培训”？拷贝编辑功能提供了哪些能力？页面布局功能要到什么程度？

随着范围描述的进展，自然产生了第一级划分。项目组研究市场部与潜在用户的交谈资料，并找出自动拷贝编辑应该具有下列功能：

- 拼写检查。
- 语句文法检查。
- 大型文档的参考书目关联检查(如对一本参考书的引用是否能在参考书目列表中找到？)。
- 大型文档中章节的参考书目关联的验证。

其中每一项都是软件要实现的子功能。同时，如果分解可以使计划更简单，则每一项又可以进一步精化。

3.4 过程

软件过程的一般阶段(定义、开发和维护)适用于所有软件项目。问题在于如何选择一个适合项目组要开发的软件的过程模型。在第 2 章中，讨论了多种软件工程范型：

- 线性顺序模型。
- 原型模型。
- RAD 模型。
- 增量模型。
- 螺旋模型。
- 构件组装模型。
- 并发开发模型。
- 形式化方法模型。

• 第四代技术模型。

项目管理者必须决定哪一个过程模型最适合待开发项目，然后基于公共过程框架活动集合，定义一个初步的计划。一旦建立了初步的计划，便可以开始进行过程分解，即必须建立一个完整的计划，以反映框架活动中所需要的工作任务。我们在下面的小节中简要讨论了这些活动，在第 7 章中将给出更详细的信息。

3.4.1 合并问题和过程

项目计划开始于问题和过程的合并。软件项目组要开发的每一个功能都必须通过为软件组织定义的框架活动集合来完成。假设组织采用了如下的框架活动集合(第 2 章)：

一般过程框架活动\用户通信\计划\风险分析\工程\

软件工程任务\\\\\

产品功能\\\\\

正文输入\\\\\

编辑及格式设计\\\\\

自动拷贝编辑\\\\\

页面布局\\\\\

自动建立索引及目录\\\\\

文件管理\\\\\

文档生成\\\\\

图 3－2 合并问题和过程

• 用户通信——建立开发者和用户之间有效通信所需要的任务。

• 计划——定义资源、进度及其他相关项目信息所需要的任务。

• 风险分析——评估技术的及管理的风险所需要的任务。

• 工程——建立应用的一个或多个表示所需要的任务。

• 建造及发布——建造、测试、安装和提供用户支持(如文档及培训)所需要的任务。

• 用户评估——基于在工程阶段产生的或在安装阶段实现的软件表示的评估，获得用户反馈所需要的任务。

开发每一个功能的项目组成员都要将每一个框架活动应用于该功能的实现上。实质上，产生了一个类似图 3－2 所示的矩阵。每一个主要的问题功能(图中列出了前述的字处理软件的功能)被列在左手边的一列中。框架活动则列在顶端的一行中。软件工程工作任务(对于每一个框架活动)列在紧接着的行中(应该注意到工作任务必须适应于项目的特定需要。框架活动总是一样的，但工作任务则要根据一系列的适应性标准来选择)。项目管理者(和其他组员)的工作是估算每一个矩阵单元的资源需求、与每一个单元相关的任务的开始和结束日期、及每一个单元所产生的工作产品。这些问题将在第 5 和第 7 章探讨。

3.4.2 过程分解

软件项目组在选择最适合项目的软件工程范型、以及选定的过程模型中所包含的软件工程任务时，应该有很大的灵活度。一个相对较小的项目如果与以前已开发过的项目相似，可以采用线性顺序模型。如果时间要求很紧，且问题能够被很好地划分，RAD 模型可能是正确的选择。如果时间太紧，不可能完成所有功能时，增量模型可能是最合适的。同样地，具有其他特性(如不确定的需求、突破性的技术、困难的用户、明显的复用潜力等)的项目将导致选择其他过程模型(回忆一下项目的特性，对开发该项目的小组的结构也有巨大影响，见 3.2.3 节)。

一旦选定了过程模型，公共过程框架(Common Process Framework， CPF)应该适于它。本章较早讨论的 CPF——用户通信，计划，风险分析，工程，建造及发布，用户评估——能够适用于范型。它可以用于线性模型，还可用于迭代和增量模型、演化模型、甚至是并发或构件组装模型。CPF 是不变的，充当一个软件组织所执行的所有软件工作的基础。

但实际的工作任务却是可变的。当项目管理者问到下面的问题时过程分解就开始了：“我们如何完成这个 CPF 活动？”。例如，一个小型的、相对简单的项目在用户通信活动中可能需要下列工作任务来完成：

1.列出需澄清问题的清单。

2.与用户见面说明需澄清的问题。

3.共同给出范围描述。

4.就所关心的问题复审范围描述。

5.根据需求修改范围描述。

这些事件可能在不到 48 小时的时间内发生。它们代表了适于小型的、相对简单的项目的一种过程分解。

现在，我们考虑一个复杂些的项目，它具有更广的范围和更重要的商业影响。这样一个项目在用户通信活动中可能需要下列工作任务来完成：

1.复审用户要求。

2.计划和安排与用户进行正式的、卓有成效的会议。

3.研究如何定义推荐的解决方案和已有的方法。

4.为正式的会议准备一份“工作文档”和议程。

5.召开会议。

6.共同制订能够反映软件的数据、功能和行为特性的小的规格说明书。

7.复审每一份小的规格说明书，确认其正确性、一致性和无二义性。

8.把这些小的规格说明书组装起来形成一份范围文档。

9.就所关心的问题复审范围文档。

10.根据需求修改范围文档。

两个项目都执行了我们称之为用户通信的框架活动，但第一个项目组只执行了第二个项目组一半的软件工程工作任务。

3.5 项目

疲惫不堪的产业专家们在讨论特别困难的软件项目时，常常提及 90—90 规则：一个系统的第一个 90％花费了所分配工作量和时间的 90％。系统最后的 10％也会花费所分配工作量和时间的 90％［ZAH94］。这段陈述告诉了我们很多关于一个项目陷入困境的情况：

• 评估进度所采用的方法是有缺陷的。（很显然，如果 90—90 规则是真的，90％的完成度就不是一个准确的指标！）

• 没有办法测定进度，因为没有可用的量化的度量。

• 项目计划在项目结束时没有考虑协调所需要的资源。

• 没有明确地考虑风险，没有建立缓解、监控和管理风险的计划。

• 进度计划是(1)不现实的，或(2)有缺陷的。

为了克服这些问题，在项目开始时必须花时间建立一个现实的计划，在项目进行中监控该计划，并在项目整个过程中控制质量和变化。花费这些时间的活动将在下一章中详细讨论。

3.6 小结

软件项目管理是软件工程的保护性活动。它先于任何技术活动之前开始，且持续贯穿于整个计算机软件的定义、开发和维护之中。

三个 P 对软件项目管理具有本质的影响——人员、问题和过程。人员必须被组织成有效率的小组，激发他们进行高质量的软件工作，并协调他们实现高效的通信。问题必须由用户与开发者交流，划分(分解)成较小的组成部分，并分配给软件小组。过程必须适应于人员和问题。选择一个公共过程框架，采用一个合适的软件工程范型，并挑选一个工作任务集合来完成项目的开发。

所有软件项目中最关键的因素是人员。软件工程师可以按照不同的小组结构来组织，从传统的控制层到“开放式范型”的小组。可以采用多种协调和通信技术来支持项目组的工作。一般而言，正式的复审和非正式的个人间通信对开发者最有价值。

项目管理活动包含测度和度量、估算、风险分析、进度安排、跟踪和控制。这些题目在下面的章节中将进一步探讨。

思考题

3.1 基于本章给出的信息和自己的经验，列举能够增强软件工程师能力的“十条戒律”。即，列出 10 条指导原则，使得软件人员能够在工作中发挥其全部潜力。

3.2 软件工程研究所的人员管理能力成熟度模型(PM—CMM)定义了培养优秀软件人员的“关键实践区域”。你的老师将为你指派一个关键实践区域，请你分析和总结该区域。

3.3 描述三种现实生活中的实际情况，其中客户和最终用户是相同的人。也描述三种他们是不同的人的情况。

3.4 高级管理者所做的决策会对软件工程项目组的效率产生重大的影响。请列举五个实例来说明这是真的。

3.5 复习一下 Weinberg 的书 [WEI86]，并写出一份二三页的总结，说明在使用 MOI 模型时应该考虑的问题。

3.6 在一个信息系统组织中，你被指派为项目管理者。你的工作是建造一个应用程序，类似于你的小组以前已经做过的项目，虽然这一个规模更大且更复杂一些。需求已经由用户写成文档。你会选择哪种小组结构？为什么？你会选择哪(些)种软件过程模型？为什么？

3.7 你被指派为一个小型软件产品公司的项目管理者。你的工作是建造一个具有突破性的产品，该产品结合了虚拟现实的硬件和高超的软件。因为家庭娱乐市场的竞争非常激烈，完成这项工作的压力很大。你会选择哪种小组结构？为什么？你会选择哪(些)种软件过程模型？为什么？

3.8 你被指派为一个大型软件产品公司的项目管理者。你的工作是管理该公司已被广泛使用的字处理软件的新版本的开发。因为竞争激烈，已经规定了严格的期限，并对外公布。你会选择哪种小组结构？为什么？你会选择哪(些)种软件过程模型？为什么？

3.9 在一个为遗传工程领域服务的公司中，你被指派为项目管理者。你的工作是管理一个新软件产品的开发，该产品能够加速基因分类的速度。这项工作是面向研究及开发(R&D)的，但其目标是在下一年内生成产品。你会选择哪种小组结构？为什么？你会选择哪(些)种软件过程模型？为什么？

3.10 基于图 3－1 中给出的研究结果，文档被认为其价值远比实用性要差。你认为为什么会出现这种情况？如何做才能使“文档”的数据点移到图中回归线之上？即，如何做才能改进文档的认知价值。

3.11 你被要求开发一个小型应用软件，它的作用是分析一所大学提供的每一门课程，并输出这门课的平均成绩(对于一个给定的学期)。写出该问题的范围描述。

3.12 给出 3.3.2 节讨论的页面布局功能的第一级分解。

推荐阅读文献及其他信息源

由 Weinberg 所著的 3 册一套的系列丛书(Quality Software Management，高质量软件管理，Dorset，1992，1993，1994)介绍了基本的系统思想和管理概念，解释了如何有效地使用测度，并说明了“一致的行为”，即建立管理者的需要、技术人员的需要及商业的需要之间的配合能力。它给新的及有经验的管理者都提供了有用的信息。Brooks(The Mythical Man－Month，神话般的人月，Anniversary Edition，Addison－Wesley，1995)更新了他的杰作，给出了关于软件项目及管理问题的新见解。Purba 和 Shah(How to Manage a Successful Software Project，如何管理一个成功的软件项目，Wiley，1995)提出了很多案例研究，指出为什么一些项目能成功，而另外一些则失败。Bennatan(Software Project Management in a Client/Server Environment，客户机/服务器环境中的软件项目管理，Wiley，1995)讨论了与客户机/服务器系统的开发相关的特殊的管理问题。

Weinberg 的另外一本优秀的论著 [WEI86] 是每一个项目管理者和每一个小组负责人必读的书籍。它能指导你更加有效地进行工作。House(The Human Side of Project Management, Addison－Wesley, 1988)和 Crosby(Running Things: The Art of Making Things Happen, McGraw－Hill, 1989)为必须处理人员及技术问题的管理者提供了实用的建议。DeMarco 和 Lister [DeM87] 及 Weinberg(Understanding the Professional Programmer, Dorset House, 1988)的论著中对软件人员及其管理方法提供了有用的建议。

以下一些书中也给出了关于项目管理的实用指南： Wysocki et al.(Effective ProjectManagement,Wiley,1996),Metzger 和 Boddie(Managing a Programming Project:Processes andPeople,3rd edition,Prentice－Hall, 1995), 以及 O'Connell(How To Run Successful Projects, Prentice－Hall, 1994)。Constantine(Constantine on Peopleware, Prentice－Hall, 1995)也论述了软件领域中的项目管理。McCarthy(Dynamics of Software Development, Microsoft Press, 1995)写了一本有见解的书，探讨了按照进度计划交付“大型软件”的规则。

一个很好的关于项目管理的资源——其中包括有有用的信息及丰富的书籍、工具和培训参考——可在下面的网址上找到：

http://www.cis.ohio-state.edu/hypertext/faq/usenet/proj-planfaq/faq.html

Project 2000(PS 2000)是欧洲的一个研究及开发关于项目管理的程序。大量有用的信息和指南可在如下网址上找到：

http: //www.ntnu.no/ps2000/welcome.html

项目管理研究所的站点上包含了关于项目管理教育计划、出版物、特殊兴趣小组及工具的信息：

http://www.pmi.org/

一个最新的关于软件项目管理的参考文献目录可在下面的网址上找到：
http://www.rspa.com

第 4 章　软件过程和项目的度量

测度对于任何工程学科而言都是基本的，软件工程也不例外。Lord Kelvin 曾经说过：

当你能够测度你所说的，并将其用数字表达出来，你就对它有了一些了解；但当你不能测度，不能用数字表达它时，你对它的了解就很贫乏、很不令人满意：它可能是知识的开始，但你在思想上还远没有进入科学的阶段。

在过去十年中，软件工程界终于开始认可了 Lord Kelvin 的话。但并非没有挫折，也不是只有一点点的争论。

软件度量是指计算机软件中范围广泛的测度。测度可以应用于软件过程中，目的是在一个连续的基础上改进它。测度也可以用于整个软件项目中，辅助估算、质量控制、生产率评估、及项目控制。最后，软件工程师使用测度能够帮助评估技术工作产品的质量，且在项目进行中辅助决策。

在软件项目管理中，我们主要关心生产率和质量的度量——根据投入的工作量和时间对软件开发“输出”的测度，对产生的工作产品的“适用性”的测度。为了达到计划及估算的目的，我们的兴趣主要放在历史上。在过去的项目中软件开发生产率是怎样的呢？产生的软件的质量是怎样的？如何从过去的生产率及质量数据推断出现在的状况？过去的信息如何帮助我们更加准确地计划和估算呢？

在本章中，我们主要探讨用于过程和项目级的软件度量。在后面的几章中，我们将给出软件工程师在技术环境下使用的度量方法。

4.1 测度、度量和指标

虽然术语“measure”(测量)、“measurement”(测度)和“metrics”(度量)经常被互换地使用，但注意到它们之间的细微差别是很重要的。因为“measure”(测量)和“Measurement”(测度)即可以作为名词也可以作为动词，所以它们的定义可能会混淆。在软件工程领域中，“measure”(测量)对一个产品过程的某个属性的范围、数量、维度、容量或大小提供了一个定量的指示。“Measurement”(测度)则是确定一个测量的行为。IEEE 的软件工程术语标准辞典(IEEE Standard Glossary of Software Engineering Terms)[IEE93]中定义“metric”(度量)为“对一个系统、构件或过程具有的某个给定属性的度的一个定量测量”。

当获取到单个的数据点(如在一个模块的复审中发现的错误数)时，就建立了一个测量。测度的发生是收集一个或多个数据点的结果(如调研若干个模块的复审，以收集每一次复审所发现的错误数的测量)。软件度量在某种程度上与单个的测量相关(如每一次复审所发现的错误的平均数，或复审中每人/小时所发现的错误的平均数)。

软件工程师收集测量结果并产生度量，这样就可以获得指标“indicator”。指标是一个度量或度量的组合，它对软件过程、软件项目或产品本身提供了更深入的了解[RAG95]。指标所提供的更深入的理解，使得项目管理者或软件工程师能够调整开发过程、项目或产品，这样使事情进行得更顺利，能被更好地完成。

例如，四个软件小组共同完成一个大型软件项目。每一个小组必须进行技术复审，但允许其自行选择所采用的复审类型。检查度量结果——每人/小时所发现的错误数，项目管理者注意到采用更加正式的复审方法的两个小组，每人/小

时所发现的错误数比起另外两个小组高 40%。假设所有其他参数都相同，这就给项目管理者提供了一个指标：正式的复审方法比起其他复审方法在时间投资上能得到更大的回报。他可能会决定建议所有小组都采用更加正式的方法。度量给管理者提供了更深入的理解，而更深入的理解会产生更严谨、更正确的决策。

4.2 过程和项目领域中的度量

测度在工程界中是常事。我们测量动力消耗、重量、物理体积、温度、电压、信号—噪音比……不胜枚举。不幸的是，在软件工程界测度还远未普及。我们对于要测量什么及如何评估收集到的度量结果尚没有达成一致意见。

应该收集度量，以确定过程和产品的指标。过程指标使得软件工程组织能够洞悉一个已有过程的功效(如范型、软件工程任务、工作产品、及里程碑)。它们使得管理者和开发者能够评估哪些部分可以运作，哪些部分不行。过程度量的收集贯穿整个项目之中，并历经很长的时间。它们的目的是提供能够导致长期的软件过程改善的指标。

项目指标使得软件项目管理者能够(1)评估正在进行的项目的状态；(2)跟踪潜在的风险；(3)在问题造成不良影响之前发现问题；(4)调整工作流程或任务；以及(5)评估项目组控制软件工程工作产品的质量的能力。

在某些情况下，可以采用同样的软件度量来确定项目的过程的指标。事实上，项目组收集到的并被转换成项目度量的测量数据，也可以传送给负责软件过程改进的人们。因此，很多同样的度量既用于过程领域又用于项目领域。

4.2.1 过程度量和软件过程改进

改进过程的唯一合理的方法是测量过程的特定属性，基于这些属性开发一组有意义的度量，而后使用这组度量来提供引导改进战略的指标。但是，在我们讨论软件度量及它们对软件过程改进的影响之前，必须注意到过程仅是众多“改进软件质量和组织性能的控制因素”中的一种［PAU94］。

在图 4－1 中，过程位于三角形的中央，并连接了三个对软件质量和组织性能有重大影响的因素。人员的技能和激励被认为是对质量和性能最有影响的因素［BOE81］。产品的复杂性对质量和小组性能产生相当大的影响。过程中采用的技术(如软件工程方法)也有影响。除此之外，过程三角形存在于环境条件的圆圈之内，这些条件包括：开发环境(如 CASE 工具)、商业条件(如交付期限，商业规则)、及用户的特性(如通信的容易程度)。

我们间接地测量一个软件过程的功效。即，我们基于从过程中获得的结果导出一组度量。这些结果包括：在软件发布之前发现的错误数的测量，交付给最终用户并由最终用户报告的缺陷的测量，交付的工作产品的测量，花费的工作量的测量，花费的时间的测量，与进度计划是否一致的测量，以及其他测量。我们还

通过测量特定软件工程任务的特性来导出过程度量。例如，我们可能测量第 2 章中描述的保护性活动及一般软件工程活动所花费的工作量和时间。

Grady [GRA92] 认为不同类型的过程数据可以分为“私有的和公用的”。因为某个软件工程师可能对在其个人基础上收集的度量的使用比较敏感，这是很自然的，这些数据对此人应该是私有的，并成为仅供此人参考的指标。我们列举若干私有的度量数据：

- 缺陷率(个人的。)
- 缺陷率(模块的)。
- 开发中发现的错误。

“私有过程数据”的观点与 Humphrey [HUM95] 所建议的个人软件过程(Personal SoftwareProcess)方法相一致。Humphrey 这样描述该方法：

个人软件过程(PSP)是一个过程描述、测度和方法的结构化集合，能够帮助工程师改善其个人性能。它提供了表格、脚本和标准，以帮助工程师估算和计划其工作。它显示了如何定义过程及如何测量其质量和生产率。一个基本的 PSP 原则是：每个人都是不同的，对于某个工程师有效的方法不一定适合另一个。这样，PSP 帮助工程师测量和跟踪他们自己的工作，使得他们能够找到最适合自己的方法。

Humphrey 认识到软件过程改进能够、也应该开始于个人级。私有过程数据是个体软件工程师改进其工作的重要驱动力。

某些过程度量对软件项目组是私有的，但对所有小组成员是公用的。例如，主要软件功能(由多个开发人员完成)的缺陷报告、正式技术复审中发现的错误、以及每个模块和功能的代码行或功能点(参见 4.3.1 和 4.3.2 节中关于 LOC 及功能点度量的详细讨论)。这些数据可由小组进行复查，以找出能够改善小组性能的指标。

公用度量一般吸取了原本是个人的或小组的私有信息。项目级的缺陷率(肯定不能归因于某个个人)、工作量、时间及相关的数据被收集和评估，以找出能够改善组织的过程性能的指标。

软件过程度量对于一个组织提高其总体的过程成熟度，能够提供很大的帮助。不过，就像其他所有度量一样，它们也可能被误用，产生比它们解决的问题更多的问题。Grady [GRA92] 提出了一组“软件度量规则”，可用于管理者建立过程度量计划：

- 解释度量数据时使用通用的观念，并考虑组织的感受性。
- 对收集测量和度量的个人及小组提供定期的反馈。

• 不要使用度量去评价个人。

• 与开发者和小组一起设定清晰的目标及达到这些目标的度量。

• 不要用度量去威胁个人或小组。

• 指出某个问题的度量数据不应该被看成是“否定的”含义。这些数据仅仅是过程改进的指标。

• 不要被某个与其他重要度量不符合的度量迷惑。

随着一个组织更加得心应手地收集和使用过程度量，简单的指标获取被更加精确的方法所替代，该方法称为统计软件过程改进(statistical software process improvement，SSPI)。本质上，SSPI使用软件故障分析方法，来收集应用软件、系统或产品的开发及使用中所遇到的所有错误及缺陷的信息[①]。故障分析采用如下方式：

1.根据来源分类所有的错误和缺陷(如，规格说明中的错误，逻辑错误，与标准不符的错误等)。

2.记录修改每个错误和缺陷的成本。

3.统计每一类错误和缺陷的数目，并按降序排列。

4.计算每一类错误和缺陷的总成本。

5.分析结果数据，找出造成组织最高成本的错误和缺陷类型。

6.产生修正过程的计划，目的是消除(或降低其出现的频率)成本最高的错误和缺陷类型。

由上述的第 1 步和第 2 步，能够得到一个简单的缺陷分布情况(如图 4－2 所示)[GRA94]。图中的饼图显示了 8 种错误和缺陷的成因及来源(用阴影表示)。Grady 提出一种鱼骨图的开发 [GRA92] 以帮助诊断饼图中的出错原因。图 4－3 中其脊柱(中间那条线)表示要考虑的质量因素(本例中是指占总数 25％的规约缺陷)。与脊柱相连的每一根肋骨(斜线)表示质量问题的一个潜在原因(如遗漏了需求，有二义性的规约，不正确的需求，修改了的需求等)。而后，每一根主要的肋骨上又可以加上新的脊柱和肋骨，以进一步标注错误产生的原因。图 4－3 中仅给出“不正确的需求”这一项的发展过程。

过程度量的收集是创建鱼骨图的前提。通过分析一个完整的鱼骨图，可以导出使得软件组织能够改进其过程以降低错误和缺陷率的指标。

4.2.2 项目度量

软件过程度量主要用于战略的目的。软件项目度量则是战术的。即，项目管理者和软件项目组经过使用项目度量及从其中导出的指标，可以改进项目工作流程和技术活动。

大多数软件项目中项目度量的第一个应用是在评估时发生的(第 5 章)。从过去的项目中收集的度量可用来作为评估现在软件项目的工作量及时间的基础。随着项目的进展，所花费的工作量及时间的测量可以和预评估值(及项目进度)进行比较。项目管理者使用这些数据来监督和控制项目的进展。

随着技术工作的开始，其他项目度量也开始有意义了。生产率根据文档的页数、复审的时间、功能点、及交付的源代码行数来测量。除此之外，对每一个软件工程任务中所发现的错误也会加以跟踪。软件在从规格说明到设计的演化中，需要收集技术度量(第 18 章)，以评估设计质量，并提供若干指标，这些指标会影响代码生成及模块测试和集成测试所采用的方法。

项目度量的目的是双重的。首先，这些度量能够指导进行一些必要的调整以避免延迟，并减少潜在问题及风险，从而使得开发时间减到最少。其次，项目度量可在项目进行的基础上评估产品质量，并且可在必要时修改技术方法以改进质量。

随着质量的提高，错误会减到最小，而随着错误数的减少，项目中所需的修改工作量也会降低。这就导致整个项目成本的降低。

软件项目度量的另一个模型［HET93］建议每个项目都应该测量：

- 输入——完成工作所需的资源(如人员，环境)的测量。
- 输出——软件工程过程中产生的交付物或工作产品的测量。
- 结果——表明交付物的效力的测量。

实际上，这个模型既可以用于过程也可以用于项目。在项目范畴中，该模型能够在每个框架活动发生时递归地使用。这样，一个活动的输出是下一个活动的输入。结果度量能够用于提供关于工作产品有用性的指标，当这些产品从一个框架活动(或任务)流动到下一个时。

4.3 软件测量

测量在现实世界中可分为两类：直接测量(如螺钉的长度)和间接测量(如生产的螺钉的“质量”，由计算其次品率来测量)。软件测量也可以这样分类。

软件工程过程的直接测量，包括花费的成本和工作量。产品的直接测量，包括产生的代码行(lines of code，LOC)、执行速度、内存大小及某段时间内报告

的缺陷。产品的间接测量，包括功能、质量、复杂性、有效性、可靠性、可维护性及许多其他在第 18 章中讨论的“能力”。

我们已经把软件度量领域划分为过程、项目和产品度量。我们也已经注意到：对个人私有产品的度量常常被结合起来，以形成对软件项目组公用项目的度量。之后，项目度量又被联合起来产生对整个软件组织公用过程的度量。但是，一个组织如何将来自不同个人或项目的度量结合起来呐？

为了说明这个问题，我们看一个简单的例子。分开两个不同项目组的个人记录，并分类他们在软件工程过程中所发现的所有错误。之后，个人的测量被结合起来产生项目组的测量。在发布前，项目组 A 在软件工程过程中发现了 342 个错误，项目组 B 则发现了 184 个错误。所有其他情况都相同，哪个组在整个过程中能够更加有效地发现错误呢？因为我们不知道项目的规模或复杂性，所以我们不能回答这个问题。不过，如果度量采用规范化的手段，就有可能产生能够在更大的组织范围内进行比较的软件度量。

4.3.1 面向规模的度量

面向规模的软件度量是通过规范化质量和/或生产率的测量而得到的，这些测量基于所生产软件的“规模”。如果一个软件组织保持简单的记录，就会产生一个如图 4－4 所示的面向规模测量的表。该表列出了在过去几年中完成的每一个软件开发项目及其相关的测量。参考项目 alpha 表项(图 4－4)：24 个人月的工作量，成本为 168000 美元，产生了 12100 行代码。应该注意到表中记录的工作量和成本含盖了所有软件工程活动(分析、设计、编码及测试)，而不仅仅是编码。项目 alpha 的更进一步的信息包括：产生了 365 页的文档；在软件发布之前，发现了 134 个错误；软件发布给用户之后运行的第一年中遇到了 29 个缺陷；3 个人参加了项目 alpha 的软件开发工作。

项目	LOC(代码行)	工作量	成本(千美元)	文档页数	错误	缺陷	人员
alpha	12100	24	168	365	134	29	3
beta	27200	62	440	1224	321	86	5
gamma	20200	43	314	1050	256	64	6
.	.	.	.	.			
.	.	.	.	.			
.	.	.	.	.			

图 4 － 4 面向规模的度量

为了产生可以与其他项目中同类度量相比较的度量，我们选择代码行作为规范化值。根据表中所包含的基本数据，能够为每个项目产生一组简单的面向规模度量：

- 每千行代码(KLOC)的错误数。

• 每千行代码(KLOC)的缺陷数。

• 每行代码(LOC)的成本。

• 每千行代码(KLOC)的文档页数。

除此之外，还能够计算出其他有意义的度量：

• 每人月错误数。

• 每人月代码行(LOC)。

• 每页文档的成本。

面向规模的度量并不被普遍认为是测量软件开发过程的最好方法[JON86]。大多数的争议都是围绕着使用代码行(LOC)作为关键的测量是否合适。LOC 测量的支持者们声称 LOC 是所有软件开发项目的“生成品”，并且很容易进行计算；许多现有的软件估算模型使用 LOC 或 KLOC 作为关键的输入，并且已经有大量的文献和数据涉及到 LOC。另一方面，反对者们则认为 LOC 测量依赖于程序设计语言；它们对设计得很好，但较小的程序会产生不利的评判；它们不适用于非过程语言；而且它们在估算时需要一些可能难以得到的信息(如，早在分析和设计完成之前，计划者就必须估算出要产生的 LOC)。

4.3.2 面向功能的度量

面向功能的软件度量使用软件所提供的功能的测量作为规范化值。因为“功能”不能直接测量，所以必须通过其他直接的测量来导出。面向功能度量是由 Albrecht [ALB79] 首先提出来的，他建议一种称为功能点的测量。功能点是基于软件信息领域的可计算的(直接的)测量及软件复杂性的评估而导出的。

功能点是通过完成图 4－5 所示的表计算出来的。其中确定了五个信息域特性，并在表中合适的位置提供计算。信息域值按下列方式定义：(实际上，信息域值的定义及计算方式很复杂。有兴趣的读者可以参考文献 [IFP94])。

用户输入数：计算每个用户输入，它们向软件提供面向应用的数据。输入应该与查询区分开来，分别计算。

用户输出数：计算每个用户输出，它们向用户提供面向应用的信息。这里，输出是指报表、屏幕、出错信息，等等。一个报表中的单个数据项不单独计算。

用户查询数：一个查询被定义为一次联机输入，它导致软件以联机输出的方式产生实时的响应。每一个不同的查询都要计算。

文件数：计算每个逻辑的主文件(如数据的一个逻辑组合，它可能是某个大型数据库的一部分或是一个独立的文件)。

外部接口数：计算所有机器可读的接口(如磁带或磁盘上的数据文件)，利用这些接口可以将信息从一个系统传送到另一个系统。

一旦已经收集到上述数据，就将每个计数与一个复杂度值(加权因子)关联上。采用功能点方法的组织建立一个标准以确定某个特定的条目是简单的、平均的还是复杂的。不过，复杂度的确定多少有些主观。

我们采用下面的方式计算功能点：

$$FP=总计数值\times[0.65+0.01\times\Sigma F_i]$$

其中“总计数值”是从图 4-5 得到的所有条目的总和。

Fi(i=1 到 14)是基于对图 4-6 中问题的回答而得到的“复杂度调整值”(0 到 5)。等式中的常数和信息域值的加权因子是根据经验确定的。

F_i：

1.系统需要可靠的备份和复原吗？

2.需要数据通信吗？

3.有分布处理功能吗？

4.性能很关键吗？

5.系统是否在一个已有的、很实用的操作环境中运行？

6.系统需要联机数据项吗？

7.联机数据项是否需要在多屏幕或多操作之间切换以完成输入？

8.需要联机更新主文件吗？

9.输入、输出、文件或查询很复杂吗？

10.内部处理复杂吗？

11.代码需要被设计成是可复用的吗？

12.设计中需要包括转换及安装吗？

13.系统的设计支持不同组织的多次安装吗？

14.应用的设计方便用户修改和使用吗？

一旦计算出功能点，则该以类似 LOC 的方法来使用它们，以规范软件生产率、质量及其他属性的测量：

- 每个功能点(FP)的错误数。
- 每个功能点(FP)的缺陷数。
- 每个功能点(FP)的成本。
- 每个功能点(FP)的文档页数。
- 每人月完成的功能点(FP)数。

4.3.3 扩展的功能点度量

功能点度量最初主要是用于商业信息系统应用软件中。为了适应这类应用软件的需要，数据维(前面讨论的信息域值)强调了排除功能维及行为(控制)维。因此，功能点度量不适合用于很多工程及嵌入式系统(它们强调了功能及控制)。为了解决这种情况，又提出了许多对功能点度量的扩展。

其中一个功能点扩展，称为特征点［JON91］，是功能点度量的超集，能够用于系统及工程软件应用程序中。特征点度量适用于算法复杂性较高的应用程序。实时系统、过程控制软件及嵌入式软件都有较高的算法复杂性，因此适合用特征点度量。

为了计算特征点，还要进行 4.3.2 节所述的信息域值的计算及加权。除此之外，特征点度量增加了一个新的软件特性，即算法。算法定义为“特定计算机程序中所包含的一个特定的计算问题”［JON91］。转置矩阵、一个位解码串、或中断处理都是算法的例子。

另一个为实时系统和工程产品的功能点扩展是由 Boeing 提出的。Boeing 的方法是将软件的数据维与功能维及行为维集成起来考虑，以提供一个面向功能的测量，称为 3D 功能点度量[WHI95]，它适用于强调功能及控制能力的应用软件。这三个软件维的特性被“计算、定量及变换”成测量值，以提供软件的功能指标。

数据维的评估基本采用 4.3.2 节描述的方法。计算获得数据(内部的程序数据结构，如文件)和外部数据(输入、输出、查询、及外部引用)，并与复杂度测量(加权因子)结合起来，导出数据维的计算。

功能维的测量是考虑“把输入变换成输出数据所需要的内部操作数”［WHI95］。为计算 3D 功能点，一个“变换”被视为一系列由一组语义表示的约束的加工步骤。一般情况下，一个变换是由一个算法来完成的，在处理输入数据，并将其变换成输出数据的过程中，它导致输入数据的根本改变。仅从一个文件中

获取数据并简单地将其放到内存中的加工步骤并不是一个变换，因为数据本身并没有发生根本的改变。

为每个变换赋复杂度值是加工步骤数及控制加工步骤的语义语句数的一个函数。表 4-1 给出了在功能维中分配复杂度值的指导原则。

表 4-1　确定 3D 功能点中一个变换的复杂度值

加工步骤 \ 语义语句	1～5	6～10	11+
1～10	低	低	平均
11～20	低	平均	高
21+	平均	高	高

控制维的测量是计算状态之间的变迁数(关于行为维的更详细的讨论将在第 12 章中给出，包括状态及状态变迁)。一个状态代表某种外部可见的行为模式，一个变迁是某个事件的结果，该事件引起软件或系统改变其行为模式(即改变状态)。例如，蜂窝电话包含支持自动拨号功能的软件。要从 resting(休眠)状态进入 auto-dial(自动拨号)状态，用户需按下键盘上的 Auto(自动)键。这个事件将产生一个 LCD 显示,提示输入呼叫方的号码。输入号码并按下 Dial(拨号)键(另一个事件)，蜂窝电话软件就产生一个到 dialing(正拨号)状态的变迁。当计算 3D 功能点时，变迁并不被赋予复杂度值。

采用下面的方法计算 3D 功能点：

$$3D\ 功能点指数=I+O+Q+F+E+T+R$$

其中 I、O、Q、F、E、T 及 R 分别代表前面讨论的元素的复杂度加权值：输入、输出、查询、内部数据结构、外部文件、变换及变迁。每一个复杂度加权值采用下面的方法计算：

$$复杂度加权值=N_{i1}W_{i1}+N_{ia}W_{ia}+N_{ih}W_{ih}$$

其中N_{i1}、N_{ia}和N_{ih}表示元素i(如输出)在每一个复杂度级别上(低、平均、高)发生的次数；W_{i1}、W_{ia}和W_{ih}则表示相应的权值。整个 3D功能点的计算如图 4-7 所示。

应该注意：功能点、特征点及 3D 功能点都代表同一种事物——软件提供的“功能”或“效用”。事实上，如果仅考虑应用软件的数据维的话，这些度量都得到同一个结果。对于更为复杂的实时系统，特征点计算的结果一般比仅仅使用功能点计算得到的结果高出 20％～35％。

功能点(及其扩展)，像 LOC 度量一样，也有很大争议。支持者们认为 FP 与程序设计语言无关，使得它既适用于传统的语言，也可用于非过程语言；它是基

于项目开发初期就有可能得到的数据，因此 FP 作为一种估算方法更具吸引力。反对者们则声称该方法需要某种“人的技巧”，因为计算是基于主观的而非客观的数据，信息域(及其他维)的计算可能难以收集事后信息，FP 没有直接的物理含义，它仅仅是一个数字而已。

4.4 调和不同的度量方法

代码行和功能点度量之间的关系依赖于实现软件所采用的程序设计语言及设计的质量。很多研究试图将 FP 和 LOC 联系起来考虑。引用 Albrecht 和 Gaffney［ALB83］的话说：

本研究的论点是：应用(程序)所提供的功能数能够从所使用的数据的主要组成部分的详细记录中估算出来，或是直接从记录中得到。更进一步，功能的估算应该与要开发的 LOC 数及开发所需的工作量关联起来。

表 4-2［ALB83，JON91］给出了在不同的程序设计语言中建造一个功能点所需的平均代码行数的一个粗略估算：

表 4-2 程序设计语言代码行粗算

程序设计语言	LOC/P(平均值)	程序设计语言	LOC/FP(平均值)
汇编语言	320	面向对象语言	30
C	128	第四代语言(4GLs)	20
Cobol	105	代码生成器	15
Fortran	105	电子表格	6
Pascal	90	图形语言(图标)	4
Ada	70		

查看上表可知，Ada 的一个 LOC 所提供的“功能性”大约是 Fortran 的一个 LOC 的 1.4 倍(平均讲)。而 4GL 的一个 LOC 所提供的“功能性”大约是传统程序设计语言的一个 LOC 的 3 到 5 倍。FP 与 LOC 之间关系的更详细的数据可以参见文献［JON91］，且这些数据可以用来从已有的程序中“逆向”确定 FP 度量(即，在已经知道交付的 LOC 数时来计算功能点数)。

LOC 和 FP 度量常常用于导出生产率度量。这就引起关于这类数据使用的争论。是否应该将某个组的 LOC/人月(或 FP/人月)与另一个组的同类数据进行比较？管理者是否应该根据这些度量来评价个人的表现？这些问题的答案毫无疑问是一个“不”字。这个回答的理由在于很多因素都会影响生产率，进行“苹果与桔子”的比较(不能比较)很容易产生曲解。

Basili 和 Zelkowitz［BAS78］定义了五个影响软件生产率的重要因素：

人因素：开发组织的规模和专业技能。

问题因素：待解决问题的复杂性及设计约束或需求的改变次数。

过程因素：使用的分析及设计技术，可用的语言及 CASEE 工具，以及复审技术。

产品因素：基于计算机系统的可靠性及性能。

资源因素：CASE 工具、硬件及软件资源的可用性。

对于一个给定项目，如果其中任何一个生产率因素高于平均值(有利的)，则软件开发生产率将明显高于该因素低于平均值的情况。

4.5 软件质量度量

软件工程的最高目标就是产生高质量的系统、应用软件或产品。为了达到这个目标，软件工程师必须掌握在成熟的软件过程背景下有效的方法及现代化的工具的应用。除此之外，一个优秀的软件工程师(及优秀的软件工程管理者)必须评估是否能够达到高质量的目标。

一个系统、应用软件或产品的质量依赖于问题需求的描述、解决方案的建模设计、可执行程序的编码的产生、以及为发现错误而运行软件的测试。一个优秀的软件工程师使用度量来评估软件开发过程中产生的分析及设计模型、源代码和测试用例的质量。为了实现这种实时的质量评估，工程师们必须采用技术度量(第 18 章和 23 章)来客观地评估质量，而不能采用主观的方法进行评估。

在项目进展过程中，项目管理者也必须评估质量。个体软件工程师所收集的私有度量可用于项目级信息的提供。虽然可以收集到很多质量测量，在项目级最主要的还是错误和缺陷测量。从这些测量中导出的度量能够提供一个关于个人及小组的软件质量保证指标及控制活动效率的指标。

复审中每小时所发现的错误及测试中每小时所发现的错误，使我们能够洞悉度量所指的那些活动的功效。错误数据也能够用于计算每个过程框架活动的缺陷排除效率(DRE，DRE 也能够用于评估在整个过程中质量保证活动的影响)。DRE 将在 4.5.3 节讨论。

4.5.1 概述影响质量的因素

二十多年前，McCall 和 Cavano [MCC78] 定义了一组质量因素，这是走向用软件质量度量开发的第一步。这些因素从三个不同的视点来评估软件：(1)产品的操作(使用它)，(2)产品的修改(改变它)，及(3)产品的转换(修改它使之能够用于另一个环境，即“移植”它)。在他们的工作中，作者给出了这些质量因素(他们称之为“框架”)与软件工程过程其他一些方面之间的关系：

首先，框架为项目管理者提供了一个机制，以标识哪些质量因素最重要。除了软件功能的正确性和性能之外，这些质量因素也都是软件的属性，在生命周期中有重要意义。一些因素，诸如可维护性及可移植性，近年来已显示出对整个生命周期的成本有重要影响。

其次，框架提供了定量评估相对于已建立的质量目标而进行的，开发进展得如何的方法。

再者，框架在整个开发工作中，提供了更多的 QA(质量保证)人员之间的交互。

最后，质量保证人员能够利用低质量的指标去帮助确定将来要增强的更好的标准。

对 McCall 和 Cavano 提出的框架的更详细的讨论，以及其他质量因素，将在第 18 章给出。有趣的是，自从 McCall 和 Cavano 在 1978 年完成他们的最初工作之后，几乎关于计算的每一个方面都发生了根本的改变，但提供软件质量指标的属性仍然没有改变。

这意味着什么呢？如果一个软件组织采用一组质量因素作为评估软件质量的一个“清单”，那么就有可能今天建造的软件一直到 21 世纪的头几十年仍展现出良好的质量。即使计算的体系结构发生了根本的改变(事实上也肯定会)，在操作、修改和转换上表现出高质量的软件仍会继续给用户提供很好的服务。

4.5.2 测量质量

虽然有很多软件质量的测量方法，但对软件进行正确性、可维护性、完整性、及可用性的测量为项目组提供了有用的技术指标。Gilb [GIL88] 给出了这些属性的定义及测量。

正确性：一个程序必须能够正确操作，否则对于用户就没有价值了。正确性是软件完成所需的功能的程度。关于正确性的最常用的测量是每千行(KLOC)的缺陷数，这里缺陷定义为验证出的与需求不符的地方。

可维护性：软件维护所占的工作量比任何其他软件工程活动都大。可维护性是指遇到错误时程序能被修改的容易程度；环境发生变化时程序能够适应的容易程度，用户希望改变需求时程序能被增强的容易程度。可维护性无法直接测量；因此，我们必须采用间接测量。一个简单的面向时间的度量是平均修改时间(mean-time-to-change，MTTC)，即分析改变的需求设计合适的修改方案实现修改测试，并将修改后的结果发布给用户所花的时间。一般情况下，可维护的程序与不可维护的程序相比，有较低的 MTTC(相对于同类修改而言)。

Hitachi [TAJ81] 使用一种面向成本的可维护性度量，称为“损坏度”——在软件发布给其最 终用户之后修改所遇到的缺陷的成本。如果用损坏度与整个

项目成本(对于许多项目)的比作为时间的函数，管理者就能够据此来确定一个软件开发组织所产生的软件的总体可维护性是否有所改进。而后管理者便可以根据这个信息采取相应的行动。

完整性：在黑客及病毒横行的现在，软件完整性已变得日益重要。这个属性测量系统在安全方面的抗攻击(包括偶然的和蓄意的)能力。攻击可能发生在软件的三个主要成分上：程序、数据及文档。

为了测量完整性，必须定义两个附加的属性：威胁和安全性。威胁是某个特定类型的攻击在给定时间内发生的可能性(能够根据经验估算或推断出来)。安全性是某个特定类型的攻击将被击退的可能性(也能够根据经验估算或推断出来)。一个系统的完整性可以定义为：

$$完整性=\Sigma\ [1—威胁\times(1—安全性)]$$

这是威胁及安全性针对每种类型的攻击求和。

可用性：在对软件产品的讨论中，“用户友好性”这个词已经是普遍存在的。如果一个程序不是“用户友好的”，那么它是注定会失败的，即使它所完成的功能很有价值。可用性试图量化“用户友好性”，并根据四个特性来测量：(1)学会一个系统所需的体力的和/或智力的投入；(2)在系统的使用上达到中等效率所需的时间；(3)当系统由某个具有中等效率的人使用时，测量到的生产率的净增长(与被该系统替代的老系统相比)；以及(4)用户对系统的态度的一个主观评估(有时可以通过调查表获得)。关于本话题的更详细的探讨参见第 14 章。

上述的四个因素仅仅是被建议作为软件质量测量的众多因素中的一个样板。第 18 章将给出更多的信息。

4.5.3 缺陷排除效率

缺陷排除效率(DRE)在项目级和过程级都能提供有益的质量度量。本质上，DRE 是对质量保证及控制活动的过滤能力的一个测量，这些活动贯穿于整个过程框架活动。

当把一个项目作为一个整体来考虑时，DRE 按如下方式定义：

$$DRE=E/(E+D)$$

其中 E=软件交付给最终用户之前所发现的错误数

D=软件交付之后所发现的缺陷数

最理想的 DRE 值是 1，即软件中没有发现缺陷。现实中，D 会大于 0，但随着 E 值的增加，DRE 的值仍能接近 1。事实上，随着 E 的增加，D 的最终值可能会降低(错误在变成缺陷之前已经被过滤了)。如果 DRE 作为一个度量，提供关于

质量控制和保证活动的过滤能力的衡量指标，则 DRE 鼓励软件项目组采用先进技术，以便在交付之前发现尽可能多的错误。

DRE 也能够用来在项目中评估一个小组发现错误的能力，在这些错误传递到下一个框架活动或软件工程任务之前。例如，需求分析任务产生了一个分析模型，可以复审该模型以发现和改正错误。在对分析模型的复审中未被发现的错误会传递给设计任务(在这里它们有可能被发现，也有可能没被发现)。在这种情况下，我们定义 DRE 为：

$$DRE_i = E_i 1 (E_i + E_{i+1})$$

其中E_i=在软件工程活动i中所发现的错误数

E_{i+1}=在软件工程活动i＋1 中所发现的错误数，这些错误来源于软件工程活动i中未能发现的错误。

一个软件项目组(或单个软件工程师)的质量目标是使DRE_i接近 1。即，错误应该在传递到下一个活动之前被过滤掉。

4.6 在软件过程中集成度量

大多数软件开发者仍然没有进行测量，更可悲的是，他们中大多数根本没有开始测量的愿望。正如我们在本章开始所说的，这是文化的问题。试图收集过去从来没有人收集的测量常常会遇到阻力。迷惑不解的项目管理者问“为什么我们要做这个？”。工作过度的开发者抱怨“我看不出这样做有什么用”。

为什么测量软件工程过程及其产生的产品(软件)如此重要？答案其实很明显。如果我们不进行测量，就没法确定我们是否在改进。如果我们没有改进，那就等于失败了。

测量是可以帮助解决第 1 章中所述的“软件苦恼”的若干“良药”中的一种。它能够提供战略级、项目级及技术级上的利益。

通过请求和评估生产率及质量的测量，高级管理者能够建立有意义的目标来改进软件工程过程。在第 1 章，我们注意到软件对于许多公司而言是一个战略性的商业事务。如果开发软件的过程能够有所改进，就会对基层产生直接的影响。但要建立改进的目标，必须了解软件开发的当前状态。因此，用测量建立一个过程的基线，并基于此来评估改进的效用。

软件项目工作的日益繁重使得几乎没有时间去进行战略性的思考。软件项目管理者更关心现实的问题(当然这也同样重要)：有意义的项目估算；高质量系统的产生；产品按时交付等。通过使用测量来建立项目基线，使得这些问题更加容易管理。我们已经知道基线是估算的基础，此外，质量度量的收集使得一个组织能够“调整”其软件工程过程，以排除引起对软件开发有重大影响的缺陷的“致

命”原因，(这些想法已被归纳为一个方法，称为统计软件质量保证，将在第 8 章详细讨论)。

在项目级和技术级，软件度量能够提供立即可见的好处。软件设计完成后，大多数开发者都急于知道以下问题的答案：

- 哪些用户需求最有可能发生改变？
- 本系统中哪些模块最易于出错？
- 对于每一个模块需要设计多少测试？
- 当测试开始时，预计会有多少错误(特定类型的)？

如果度量已经被收集并被用作一项技术指南，这些问题的答案就能够确定了。在后面的章节中我们还要讨论这是如何做到的。

建立基线的过程如图 4-8 所示(度量的基线是由从以前的软件开发项目中收集的数据所组成的，这些数据可以像图 4-4 给出的表那么简单，也可以是复杂的综合数据库，其中包含几十个项目测量及从其导出的度量)。理想情况下，建立基线所需的数据是以一种渐进的方式收集的。但遗憾的是，很少能做到这样。因此，数据收集需要对以前的项目进行历史的调查，以重建所需的数据。一旦收集好了测量(无疑这是最困难的第一步)，就有可能进行度量计算。依据所收集的测量的广度，度量可以跨越 LOC 或 FP 度量及其他面向质量和项目的度量。最后，度量必须在估算、技术工作、项目控制及过程改善中加以评估和使用。度量评估主要是分析产生所得到的结果的原因，并生成一组指导项目或过程的指标。

4.7 小结

测量使得管理者和开发者能够改善软件过程；辅助软件项目的计划、跟踪及控制；评估产生的产品(软件)的质量。对过程、项目及产品的特定属性的测量被用于计算软件度量。分析这些度量可产生指导管理及技术行为的指标。

过程度量使得一个组织能够从战略级洞悉一个软件过程的功效。项目度量是战术的，使得项目管理者能够以实时的方式改进项目的工作流程及技术方法。

产业界普遍使用面向规模和面向功能的度量。面向规模的度量使用代码行作为其他测量，如人月或缺陷的规范化因子。功能点则是从信息域的测量及对问题复杂度的主观评估中导出的。

软件质量度量，如生产率度量，集中于过程、项目及产品上。通过建立和分析质量的度量基线，一个组织能够采取行动来纠正引起软件缺陷的那些软件过程区域。本章中讨论了四个质量度量——正确性、可维护性、完整性及可用性。其他的质量度量将在本书后面的章节中探讨。

测量导致文化的改变。如果打算开始进行度量，则数据收集、度量计算及度量评估是必须执行的三个步骤。通过创建一个度量基线(一个包含过程及产品测量的数据库)软件工程师及管理者能够更好地了解他们所做的工作及所开发的产品。

思考题

4.1 给出可以用以评估一辆汽车的三个测量、三个度量及相应的指标。

4.2 给出可以用以评估一个汽车经销商的服务部门的三个测量、三个度量及相应的指标。

4.3 用自己的话描述过程和项目度量之间的不同。

4.4 为什么某些软件度量是“私有的”？给出三个私有度量的例子，并给出三个公用度量的例子。

4.5 阅读［HUM95］，并写出一个 1 到 2 页的总结简述 PSP 方法。

4.6 Grady 提出了一组“软件度量规则”。你能够在 4.2.1 节所列的规则中再增加三个吗？

4.7 试着完成图 4-3 所示的鱼骨图。即，按照“不正确的”规格说明所使用的方法，为“遗漏的”需求，“有二义性的”规格说明，及“改变的”需求提供类似的信息。

4.8 什么是间接测量？为什么在软件度量工作中经常用到这类测量？

4.9 产品交付之前，小组 A 在软件工程过程中发现了 342 个错误。小组 B 发现了 184 个错误。对于项目 A 和 B 还需要做哪些额外的测量，才能确定哪个小组能够更有效地排除错误？你建议采用什么度量来帮助做决定？哪些历史数据可能有用？

4.10 给出一个反对代码行作为软件生产率度量的论据。当考虑几十个或是几百个项目时，你说的情况还成立吗？

4.11 根据下面的信息域特性值，计算项目的功能点值：

用户输入数：32

用户输出数：60

用户查询数：24

文件数：8

外部接口数：2

假设所有的复杂度调整值都是“平均”。假设共有 14 个算法，计算特征点值。

4.12 计算一个嵌入式系统的 3D 功能点值，它具有下面的特性值：

内部数据结构：6

外部数据结构：3

用户输入数：12

用户输出数：60

用户查询数：9

外部接口数：3

变换：36

变迁：24

假设以上计数的复杂度平均分布为低、平均和高。

4.13 用于控制先进的影印机的软件需要 32，000 行 C 语言代码和 4200 行类 4GL 描述语言。估算该影印机软件的功能点数。

4.14 McCall 和 Cavano(见 4.5.1 节)为软件质量定义了一个“框架”。根据本书及其他书籍中提供的信息，分别将三个主要的“视点”展开为一组质量因素及度量。

4.15 为正确性、可维护性、完整性、及可用性建立你自己的度量(不要使用本章给出的度量)。确定它们能够转换成定量的值。

4.16 当每千行缺陷数降低时，“损坏度”有可能提高吗？为什么？

4.17 当使用第四代语言时，LOC 测量还有意义吗？为什么？

4.18 写一篇文章概述软件生产率研究的结果(参见推荐文献)。这些结果之间有共同之处吗？

4.19 收集 2 到 5 个你参加的项目的软件度量。基本的信息应该包括：成本，LOC 或功能点，工作量(人月)，复杂度指标(从 1 到 10)，及完成时间。将你的数据与其他同学或同事的信息结合起来，建立并检查班级的度量基线。

4.20 收集能够用于向管理部门证明测量是有价值的活动的资料，写出一份报告，注意使用定量的论据，并将在班上报告结果。

推荐阅读文献及其他信息源

软件过程改进近年来得到极大关注。Humphrey [HUM95]、Yeh(软件过程控制，SoftwareProcess Control，McGraw-Hill，1993)，Hetzel [HET93]，以及Grady [GRA92] 的著作都探讨了如何使用软件度量来提供必要的指标，以改善软件过程。Putnam 和 Myers(Executive Briefing：Controlling Software Development，IEEE Computer Society，1996)，以及 Pulford 和其同事们(AQuantitative Approach to Software Management，Addison-Wesley，1996)则从管理的视点探讨了过程度量及其使用。

Weinberg(Quality Software Management，Volume 2：First Order Measurement，Dorset House，1993)提出了一个有用的模型，能够观察软件项目，确定观察结果的含义，并决定其对战略及战术决策的重要性。Garmus 和 Herron(Measuring the Software Process，Prentice-Hall，1996)讨论了过程度量，重点放在功能点分析上。软件生产力协会，(The Software Measurement Guidebook，Thomson Computer Press，1995)对于如何建立有效的度量方法提出了有用的建议。

在过去十年中，国防部及政府/企业的研究中心出版了很多企业的报告及度量指南。主要包括：

Andres，D.，Software Project Management Using Effective Process Metrics：The CCPDS-R Experience，TRW-TS-89-01，TRW Systems Engineering & Development Division，November 1989.

Carleton，A.D.et al.， Software Measurement for DoD Systems：Recommendations for Initial Core Measures，CUM/SEI-92-TR-19，Carnegie-Mellon University，Software Engineering Institute， September 1992.

Florac，W.A.,Software Quality Measurement：A Framework for Counting Problems and Defects，CMU/SEI-92-TR-22，Carnegie-Mellon University，Software Engineering Institute， September 1992.

Metrics Reporting Guidebook，Version 1.1，Defense Information Systems Agency， prepared by Mitre Corp.，May 1994.

Metrics Starter Kit and Guidelines，Software Technology Support Center，Hill AFB，UT，1994.

在 Internet 上，关于软件度量的报告及信息索引可在软件工程实验室(Software EngineeringLaboratory)的网站上找到：

http：//fdd.gsfc.nasa.gov/seltext.html

美国军方的软件度量系统网站(Software Metrics System Web)上包含很多关于过程度量的信息：

http：//www.army.mil/optec-pg/homepage.htm

关于过程度量的大量教科书及文章的目录可在下面的网址上找到：

http：//www.rai.com/soft-eng/sme.html

已经建立了一个 Listserv 邮件列表，能够获得功能点度量的信息。如果想订阅，发 mail 给：cim@crim.ca

SUBJECT：“”(这个域必须是空的)

CONTENT：SUB FUNCTION.POINT.LIST“你的名字”

一个最新的关于软件过程度量的WWW参考目录可在下面的网址上找到：http：//www.rspa.com

① 如我们在第 8 章所讨论的，错误是软件工程工作产品或交付物中的某个瑕疵，在该软件交付给最终用户之前已被软件工程师发现。而缺陷则是交付给最终用户之后所发现的瑕疵。

第 5 章　软件项目计划

软件项目管理过程从一组被称为项目计划的活动开始。这些活动中的第一个是估算。无论何时进行估算，我们都是在预测未来，并会接受某种程度的不确定性。引用 Frederish Brooks [BRO75] 的话来说：

我们的估算技术发展缓慢。更为严重的是，它们隐含了一个很不正确的假设，即“一切都会好的”……；因为我们对自己的估算没有把握，软件管理者常常缺少让人们得到一个好产品的信心。

虽然估算是一门科学，但它更是一门艺术，可这个重要的活动不能以随意的方式来进行。对时间及工作量进行评估的有用技术确实存在。而且因为估算是所有其他项目计划活动的基础，而项目计划又提供了通往成功的软件工程的行车图，因此，没有它我们就会搭错车。

5.1 对估算的观

一位总经理曾经被问到：在选择一个项目管理者时，什么特质是最重要的。他的回答是：“具有在错误真正发生之前就能知道它的能力”。我们还可以加上：“在未来还是一团迷雾的时候就有勇气进行估算”。

估算一个软件开发工作的资源、成本及进度需要经验、需要了解以前的有用信息、以及当仅存在定性的数据时进行定量测量的勇气。估算具有与生俱来的风险(风险分析技术在第 6 章讨论)，而正是这种风险导致了不确定性。

项目复杂性对计划中固有的不确定性产生重大影响。不过，复杂性是一个受到对以前工作的熟悉程度影响的相对的测量。实时应用对于一个以前仅仅开发过批处理应用的软件项目组而言，可能被认为是“非常复杂的”。同样的实时应用对于一个经常开发高速处理控制的软件项目组而言，则可能被认为是“小菜一碟”。目前已经有一些定量的软件复杂性测量(第 18 章和 23 章)。这类测量主要用于设计级及代码级，因此难以在软件计划中(它在设计及编码之前)使用。不过，关于复杂性的其他一些更为主观的评估(如，第 4 章描述的功能点复杂度调整因子)可以在早期的计划过程中建立。

项目规模是另一个影响估算准确性的因素。随着规模的增长，软件中各个元素之间的相互依赖性也迅速增加(项目规模的增长会对项目的成本及进度产生几何级数级的影响 [MAH96] 。估算中采用的重要方法——问题分解，也因为分解出来的元素仍然很大而变得更为困难。改写 Murphy 定律：“所有可能出错的地方都会出错”——如果有更多的部分可能失败，那就会有更多的部分失败。

结构不确定性的程度也会对估算的风险产生影响。在这里，结构是指：需求能被确定的程度，功能能被分解的容易程度，以及必须要加工的信息的层次性。

历史信息的可用程度也决定了估算的风险。Santayana 曾经说过：“不记得过去的人必将重蹈覆辙”。通过回顾过去，我们能够效法好的地方，且避免再出现同样的问题。当存在大量可用的关于过去项目的软件度量时(第 4 章)，估算就会有更大的保证；能够建立进度计划，以避免以前遇到过的困难；总体风险也会降低。

风险是由为资源、成本及进度建立的定量估算中存在的不确定性来测量的。如果对项目范围理解很差或项目需求不断变化，不确定性及风险就会很高。软件计划者应该要求功能、性能及接口定义(包含在系统说明中)的完备性。计划者，尤其是用户，应该认识到软件需求的变化意味着成本及进度的不稳定。

作为对估算的最后一个观察，我们引用亚里斯多德(公元前 330 年)的话：

记住：应该满足于事物的本性所能容许的精确度，当只能近似于真理时，不要去寻求绝对的准确……

项目管理者不应该被估算所困扰。现代软件工程方法(如，演化软件过程模型)支持开发的迭代视图。在这类方法中，当用户改变需求时，有可能会重新审查估算(在知道更多信息后)并修改之。[①]

5.2 项目计划目标

软件项目计划的目标是提供一个框架，使得管理者能够对资源、成本及进度进行合理的估算。这些估算是软件项目开始时在一个限定的时间框架内所做的，并且随着项目的进展不断更新。此外，估算应该定义“最好的情况”及“最坏的情况”，使得项目的结果能够限制在一定范围内。

项目计划的目标是通过一个信息发现的过程实现的，该过程最终导致能够进行合理的估算。在以下各节中，我们讨论了与软件项目计划相关的每一个活动。

5.3 软件范围

软件项目计划的第一个活动是确定软件范围。在系统工程阶段(第 10 章)应该对分配给软件的功能及性能加以评估，以建立一个项目范围，该范围在管理级及技术级均是无二义性的和可理解的。

软件范围描述了功能、性能、约束条件、接口及可靠性。在范围说明中给出的功能被评估，并在某些情况下被进一步精化，以便在估算开始之前提供更多的细节。因为成本及进度估算都是面向功能的，所以某种程度上的分解常常是很有用的。性能方面要考虑包括加工及响应时间在内的要求。约束条件标识了外部硬件、可用内存或其他已有系统等对软件的限制。

5.3.1 获取定义软件范围所需的信息

在软件项目开始时，事情总是有某种程度的模糊不清。已经定义了要求，并确立了基本的目标及目的，但定义软件范围所需的信息(这是估算的前提)却还没有被定义。

在用户和开发者之间建立通信的桥梁，并使通信过程顺利开始的最常用的技术是进行一个初步的会议或访谈。软件工程师(分析员)和用户之间的第一次会议可能就像青年男女的第一次约会那么尴尬。两方都不知道说什么或问什么；两方都担心他们所说的话会被误解；两方都在想会有什么结果(有可能他们的期望完全不同)；两方都希望事情赶快结束；但同时，两方都希望能够成功。

不管怎样，通信必须要开始。Gause 和 Weinberg [GAU89] 建议分析员开始时可以问一些与项目无关的问题。即，一组使你对总体情况有一个基本了解的问题，如，需要解决方案的人，所期望的解决方案的性质，以及对第一次见面的效果的评价等。

第一组与项目无关的问题主要集中于用户、总体目标及利益上。例如，分析员可能会问：

• 谁提出了关于这项工作的要求？

• 谁将会使用这个解决方案？

• 成功的解决方案将获得什么利益？

• 是否有其他的解决方案？

下一组问题使得分析员能够对问题有一个更好的理解，使得用户能够谈出他或她对于解决方案的想法：

• 你(用户)认为一个成功的解决方案所产生的“好的”输出应该具有什么特征？

• 这个解决方案针对什么问题？

• 你能给我演示(或描述)一下该解决方案将被使用的环境吗？

• 是否有什么特殊的性能问题或约束会影响该解决方案被实现的方式？

最后一组问题主要集中于会议的效果。Gause 和 Weinbers 称这些为“元问题”，并推荐了下面的问题列表(简略的)：

• 你是回答这些问题的最合适的人选吗？你的回答是否是“正式的”？

• 我提的问题与你要解决的问题相关吗？

• 我是否问了太多的问题？

• 是否还有其他人能够提供更多的信息？

• 是否还有其他我应该问你的问题？

这些问题(及其他问题)能够帮助“打破僵局”，并开始了建立项目范围所必须的通信活动。但这种问答会议的形式并不是一定会成功的。事实上，Q&A(问答)形式仅应该用于第一次会面，之后应该被问题解决、协商及规约等多种方式相结合的会议形式所取代。

有很多独立调查者提出了一种收集需求的面向小组方法，能够帮助建立项目的范围(应该注意，这些技术经常被视为是需求分析的第一步，在第 11 章将给出更详细的讨论)。一种被称为便利的应用规约技术(facilitated application specification techniques，FAST)的方法鼓励建立由用户及开发者共同组成的联合小组在一起工作，来标识问题、建议解决方案、商议不同的方法、并说明初步的所有需求。

5.3.2 一个范围定义的例子

与用户的通信使得我们可以定义数据、功能、必须实现的行为、性能、界定系统的约束、及相关的信息。举一个例子，假定要开发驱动一个传送带分类系统(a convey line sorting system，CLSS)的软件。对 CLSS 的范围说明如下：

传送带分类系统(CLSS)将沿传送带移动的盒子进行分类。每一个盒子由一个包含零件号的条形码来标识，并在传送带末端分送到六个箱子中的一个。这些盒子要通过一个由条形码阅读器及一台 PC 所组成的分类站。分类站的 PC 连接到一个分流器上，它把盒子分送到不同的箱子中。盒子以随机的顺序通过，且其间的距离相同。传送带以每分钟 5 英尺的速度移动。CLSS 如图 5-1 所示。

CLSS 软件以和传送带速度一致的时间间隔接受来自条形码阅读器的信息。条形码数据被解码成盒子的标识格式。软件将在最多可容纳 1000 个条目的零件号数据库中进行检索，以确定当前在阅读器(分类站)位置的盒子应该放到哪个箱子中。该箱子的信息被传送到分流器，以把盒子放进合适的箱子中。每一个盒子所放进的箱子的记录均被保存起来，以供以后提取及报告。CLSS 软件同时也接受来自脉冲流速计的输入，用于使控制信号与分流器同步。根据分类站和分流器之间产生的脉冲数，软件将产生一个控制信号给分流器，以适当地定位盒子。

项目计划者检查范围说明，并提炼出所有重要的软件功能，这个过程，称为分解，曾在第 3 章讨论过，它将产生如下功能(实际上，功能分解在系统工程阶段进行，见第 10 章，计划者利用从系统规约中得到的信息来定义软件功能)：

- 读条形码作为输入。
- 读脉冲流速计。
- 解零件编码数据。
- 检索数据库。
- 确定合适的箱子。
- 产生分流器的控制信号。
- 保存盒子目的地的记录。

在本例中，性能取决于传送带的速度。对于每个盒子的处理必须在下一个盒子到达条形码阅读器之前完成。CLSS 软件受以下条件约束：必须使用的硬件(条形码阅读器、分流器、PC)，可用的内存，以及整个传送带的结构(等距离的盒子)。

功能、性能及约束必须放在一起评估。在不同的性能约束下，相同的功能可能在开发工作量上有数量级的巨大差别。因此，如果保持功能相同(如，将盒子放进箱子中)而改变性能，开发 CLSS 软件所需的工作量及成本会产生戏剧性的变

化。例如，如果传送带的平均速度增长 10 倍，且盒子不再是等距的(一个约束)，软件就会复杂得多，因此需要更多的工作量。功能、性能及约束是紧密相关的。

软件会与基于计算机系统的其他组成成分之间进行交互。计划者考虑每一个接口的性质及复杂性，以确定它们对开发资源、成本及进度的影响。接口的概念是指(1)运行软件的硬件(如处理器、外设)及不直接由软件控制的设备(如机器、显示器)；(2)已有的且必须与新软件连接的软件(如数据库访问例程、可复用软件构件、操作系统)；(3)通过键盘或其他 I/O 设备使用软件的人；以及(4)在软件之前或之后共同作为一个顺序操作系列的程序，在每种情况下，通过接口传送的信息必须能被清楚地理解。

关于软件范围的最不精确的方面是对可靠性的讨论。软件可靠性测量确实已经存在(第 8 章)，但它们很少在项目的这一阶段中使用。典型的硬件可靠性特性如平均失败间隔时间(mean-time-between-failure，MTBF)难以转换到软件领域使用。不过，软件的一般性质可能引发某些特殊的考虑来保证“可靠性”。例如，航空交通控制系统或穿梭号宇宙飞船(两个都是关系到人类生命安全的系统)的软件一定不能失败，否则就会出人命。而一个库存控制系统或字处理软件原则上也不应该失败，但如果失败，其影响要小得多。虽然我们不可能在范围说明中精确地量化软件可靠性，但我们可以利用项目的性质来辅助工作量及成本的估算，以保证可靠性。

如果已经适当地建立了系统规约(见第 10 章)，那么，在软件项目计划开始之前，描述软件范围所需的几乎所有信息已经存在，且文档化了。如果还没有建立系统规约，计划者就必须担当系统分析员的角色，以确定影响估算工作的属性及约束。

5.4 资源

软件计划的第二个任务是估算完成软件开发工作所需的资源。图 5-2 将开发资源表示成一个金字塔。开发环境——硬件及软件工具——处于资源金字塔的底层，提供支持开发工作的基础。再高一层是可复用软件构件——软件建筑块，能够极大地降低开发成本，并提早交付时间。在金字塔的顶端是主要的资源——人员。每一类资源都由四个特征来说明：资源描述、可用性说明、需要该资源的时间、及该资源被使用的持续时间。后两个特征可以看成是时间窗口。对于一个特定的窗口而言，资源的可用性必须在开发的最初期就建立起来。

5.4.1 人力资源

计划者在开始评估范围及选择完成开发所需的技术。对于组织的职位(如管理者、高级软件工程师，等等)及专业技能(如，电讯、数据库、客户机/服务器)等都要加以描述。对于相对比较小的项目(六个人月或更少)，一个人就可以完成所有软件工程步骤，如果需要的话可以咨询专家。

一个软件项目所需的人员数目在完成了开发工作量的估算之后就能够确定。估算工作量的技术将在本章后面加以探讨。

5.4.2 可复用软件资源

如果没有对可复用性的认识，任何关于软件资源的讨论都将是不完整的，可复用性是指软件建筑块的创建及复用［H0091］。这类建筑块必须被分类，才能方便查找；被标准化才能方便应用；被确认，才能方便集成。

Bennatan［BEN92］建议了在计划进行过程中应该考虑的四种软件资源分类是：

可直接使用的构件：已有的，能够从第三方厂商获得或已经在以前的项目中开发过的软件。这些构件已经经过验证及确认且可以直接用在当前的项目中。

具有完全经验的构件：已有的为以前类似于当前要开发的项目建立的规约、设计、代码、或测试数据。当前软件项目组的成员在这些构件所代表的应用领域中具有丰富的经验。因此，对于这类构件进行所需的修改其风险相对较小。

具有部分经验的构件：已有的为以前与当前要开发的项目相关的项目建立的规约、设计、代码、或测试数据，但需做实质上的修改。当前软件项目组的成员在这些构件所代表的应用领域中仅有有限的经验，因此，对于这类构件进行所需的修改会有相当程度的风险。

新构件：软件项目组为满足当前项目的特定需要而必须专门开发的软件构件。

当可复用构件作为一种资源时，以下的指导原则是软件计划者应该考虑的：

1.如果可直接使用的构件能够满足项目的需求，就采用它。获得和集成可直接使用的构件所花的成本一般情况下总是低于开发同样的软件所花的成本(当在项目中使用已有的软件构件时，总体成本可能会急剧降低。事实上，产业数据表明：成本、上市时间以及缺陷数均会降低)。此外，风险也相对较小。

2.如果具有完全经验的构件可以使用，一般情况下，修改和集成的风险是可以接受的。项目计划中应该反映出这些构件的使用。

3.如果具有部分经验的构件可以使用，则必须详细分析它们在当前项目中的使用。如果这些构件在与软件中其他成分适当集成之前需要做大量修改，就必须小心行事。修改具有部分经验的构件所需的成本有时可能会超过开发新构件的成本。

具有讽刺意味的是，可复用软件构件的使用在计划阶段经常被忽视，仅在软件过程的开发阶段才变成最主要的关心对象。最好能够尽早说明软件资源需求。这样才能进行可选方案的技术评估，并及时获得所需的构件。

5.4.3 环境资源

支持软件项目的环境，通常被称为软件工程环境(software engineering environment，SEE)，集成了硬件及软件两大部分。硬件提供了一个支持工具(软件)平台，这些工具是产生通过良好的软件工程实践而得到的工作产品所必需的(其他硬件——目标环境——是软件已经交付给最终用户之后将要在其上运行的计算机)。因为大多数软件组织中均有多个小组需要使用 SEE，因此，项目计划者必须规定硬件及软件所需的时间窗口，并验证这些资源是否可用。

当要开发一个计算机系统(集成特定的硬件及软件)时，软件项目组可能需要用到由其他工程小组开发的硬件成分。例如，在某一类机械工具上使用的数控(numerical control，NC)软件可能需要某个特定的机械工具(如数控机床)来作为测试步骤确认的一部分；自动排版的软件项目可能在开发过程中的某个阶段需要用到照片排版机。软件项目计划者必须说明需要的每一个硬件成分。

5.5 软件项目估算

在计算机发展的早期，软件成本在整个计算机系统的成本中仅占很小的百分比。软件成本估算中即使出现了数量级的误差也几乎没有什么影响。今天，在大多数计算机系统中，软件已经变成开销最大的成分。一个大的成本估算误差会造成营利及亏损间的巨大差别。而超支对于开发者而言是一场灾难。

软件成本及工作量估算永远不会是一门精确的科学。太多的变化——人员、技术、环境、策略——影响了软件的最终成本及开发所需的工作量。不过，软件项目估算可以从神秘的技巧向一系列系统化的步骤的转变的过程中，估算出可接受的风险。

为了可靠地估算成本及工作量，有以下几种选择可以考虑：

1.将估算拖延到项目的最后阶段(显然，如果在项目完成之后进行估算，我们能够赢得百分之百的准确率)。

2.基于已经完成的类似的项目进行估算。

3.使用简单的“分解技术”来进行项目成本及工作量的估算。

4.使用一个或多个经验模型进行软件成本及工作量的估算。

不幸的是，第一个选择，不管它有多吸引人，是不现实的。成本估算必须“预先”提出。然而，我们应该认识到我们等的越久，知道的越多，而我们知道的越多，就越可能在估算中不会产生严重错误。

如果当前项目与以前的工作非常相似，且其他的项目影响因素(如用户的特性、商业条件、SEE、及交付期限等)也相同，第二个选择能够运作得很好。不幸的是，过去的经验并不总是未来结果的好的指示器。

其余的两个选择是软件项目估算的可用方法。理想情况下，这两种选择技术可以同时使用，互相进行交叉检查。分解技术采用“分而治之”的策略进行软件项目估算。将项目分解成若干主要的功能及相关的软件工程活动，通过逐步求精的方式进行成本及工作量估算。经验估算模型可用于补充分解技术，并提供一种潜在有价值的估算方法。该模型是基于经验(历史数据)来进行的，可以用下面的公式表示：

$$d=f(v_i)$$

其中d是要估算的值(如工作量、成本、项目持续时间)，V_i是选择出来的独立参数(如被估算的LOC或FP)。

自动估算工具实现一种或多种分解技术或经验模型。如果与交互式人机界面结合起来，在进行估算的时候自动工具将是一种很有吸引力的选择。在这类系统中，要描述开发组织的特性(如经验、环境)及待开发软件的性质。成本及工作量估算可由这些数据导出。

每一个选择来估算可用软件成本的参量取决于用于估算的历史数据。如果没有历史数据存在，成本估算也就建立在一个很不稳定的基础之上。在第 4 章中，我们已经考察了一些软件度量的特性，它们为如何提供历史估算数据奠定了基础。

5.6 分解技术

软件项目估算是一种解决问题的形式。在大多数情况下，如果将待解决的问题(即，为软件项目建立一个成本及工作量估算)作为一个整体来考虑则太过复杂了。因此，我们要分解问题，把问题重新划分成一组较小的(也更易管理的)问题。

在第 3 章中，从两个不同的视点讨论了分解方法：问题分解及过程分解。估算可以利用其中一种或两种划分形式。但在进行估算之前，项目计划者必须了解待建造软件的范围，并对其“规模”有相应的估算。

5.6.1　软件规模估算

软件项目估算的准确性取决于若干因素：(1)计划者适当地估算待建造产品的规模的程度；(2)把规模估算转换成人的工作量、时间、及成本的能力(一个来自以前项目的可靠软件度量的可用性函数)；(3)项目计划反映软件项目组能力的程度；以及(4)产品需求的稳定性及支持软件工程工作的环境。

在本节中，我们考虑软件规模估算的问题。因为项目估算的好坏取决于要完成的工作的规模估算，因此，规模估算是项目计划者面临的第一个挑战。在项目计划中，规模是指软件项目的可量化的结果。如果采用直接的方法，规模可以用LOC(代码行)来测量。如果选择间接的方法，规模可以用FP(功能点)来表示。

Putnam 和 Myers [PUT92] 建议了四种估算问题规模的方法：

“模糊逻辑”法：这个方法使用了模糊逻辑基础的近似推理技术。要使用这个方法，计划者必须说明应用软件的类型，建立其定性的规模估算，之后在最初的范围内精化该估算。虽然可以利用个人的经验，但计划者也应该访问项目的历史数据库(见 5.9 节，简单地讨论了支持模糊逻辑规模估算及本节中讨论的其他技术的工具)，使得估算能够与实际的经验加以比较。

功能点法：计划者对第 4 章讨论过的信息域特性进行估算。

标准构件法：软件由若干不同的“标准构件”组成，这些构件对于一个特定的应用领域而言是通用的。例如，一个信息系统的标准构件是子系统、模块、屏幕、报表、交互程序、批程序、文件、LOC、以及对象级的指令。项目计划者估算每一个标准构件的出现次数，然后使用历史项目数据来确定每个标准构件交付时的大小。为了说明这点，让我们以一个信息系统为例。计划者估算将产生 18 个报表，历史数据表明每一个报表需要 967 行 Cobol [PUT92] 代码。这使得计划者估算出报表构件需要 17000LOC。对于其他标准构件也可以进行类似的估算及计算，将它们合起来就得到最终的规模值(以统计方式调整)。

修改法：该方法主要用于这种情况：一个项目中包含对已有软件的使用，但该软件必须做某种程度的修改才能作为该项目的一部分。计划者要估算必须完成的要修改的数目及类型(如，复用、增加代码、修改代码、删除代码)。对于每一类修改使用“工作比率”，即可估算出修改的规模。

Putnam 和 Myers 建议：上述每种估算规模的方法所产生的结果可以在统计上进行结合，以产生一个三点或期望值估算。这可以通过下述方式实现：建立关于规模的乐观、可能、及悲观值，并使用将在下一节描述的等式(5.1)将它们结合起来。

5.6.2 基于问题的估算

在第 4 章，已描述了基于代码行(LOC)和功能点(FP)的测量，从它们可以计算出生产率度量。LOC 和 FP 数据在软件项目估算中有两种作用：(1)作为一个估

算变量，用于估算软件中每个成分的规模；(2)作为从以前的项目中收集来的，并与估算变量结合使用的基线度量，以建立成本及工作量估算。

LOC 和 FP 估算是两种不同的估算技术，但两者之间有共同之处。项目计划者从界定的软件范围说明开始，并根据该说明将软件分解为可以被单独估算的功能问题。然后，估算每一个功能的 LOC 或 FP(估算变量)。当然，计划者也可以选择其他元素进行规模估算，诸如类及对象、修改、或受到影响的业务过程。

下一步将基线生产率度量(如，LOC/pm 或 FP/pm，其中 pm 代表人月)用于变量估算中，从而导出每个功能的成本及工作量。将所有功能估算合并起来，即可产生整个项目的总体估算。

应该注意到：对于一个组织而言，其生产率度量常常是多样化的，这使得只使用一个单一的基线生产率度量来做决定非常令人怀疑。一般情况下，平均的 LOC/pm 或 FP/pm 应该按项目领域来计算，即，项目应该根据项目组的大小、应用领域、复杂性、以及其他相关的参数进行分类，之后才计算各个子领域的生产率平均值。当估算一个新项目时，首先应将其对应到某个领域上，然后，才使用合适的生产率的领域平均值进行估算。

LOC 和 FP 估算技术在分解所要求的详细程度上及划分的目标上有所差别。当 LOC 被用作估算变量时，分解(一般情况下，问题功能会被分解。但也有可能使用一组标准构件取而代之，见 5.6.1 节)是绝对必要的，而且常常需要分解到非常精细的程度。分解的程度越高，就越有可能建立合理的准确的 LOC 估算。

对于 FP 估算，分解则是不同的。它并不是集中于功能上，而是要估算每一个信息域特性——输入、输出、数据文件、查询、和外部接口——及十四个复杂度调整值，这在第 4 章已经讨论过。这些估算结果则用于导出 FP 值，该值可与过去的数据比较，并可用于产生项目估算。

不管使用哪种估算变量，项目计划者都要从估算每个功能或信息域的范围值开始。利用历史数据或直觉(当其他方法都不可行时)，计划者为每个功能或每个信息域值的计数值都估算出一个乐观的、可能的、及悲观的规模值。当确定了一个范围值时，就暗示了不确定性的程度。

接着，计算三点或期望值。估算变量(规模)的期望值，EV(expected value)，可以通过乐观值(S_{opt})、可能值(S_m)、及悲观值(S_{pess})估算的加权平均值来计算：

$$EV=(S_{opt}+4_{Sm}+S_{pess})/6 \qquad (5.1)$$

其中给予“可能值”估算以最大的权重，并遵循 β 概率分布。

我们假设实际的规模值落在乐观值和悲观值区间之外的概率极小。采用标准的统计技术，我们就能够计算出估算的偏差。然而，应该注意到：基于不确定的(估算的)数据的偏差必须被谨慎地使用。

一旦确定了估算变量的期望值，就可以开始使用历史的 LOC 或 FP 生产率数据，这种估算正确吗？对于这个问题唯一合理的答案就是：“我们不能确定”。任何估算技术，不管它有多高明，都必须与其他方法交叉使用。即使这样，直觉和经验也是必不可少的。

5.6.3 一个基于 LOC 估算的例子

为了说明 LOC 和 FP 估算技术，让我们来看一个例子：假设估算的是一个计算机辅助设计(CAD)应用开发软件包。复审系统规约，表明该软件将要运行于一个工程工作站上，且必然与各种计算机图形外设如鼠标、数字化仪、高分辨率彩色显示器、以及激光打印机有接口。

以系统规约为指导，能够建立一个初步的软件范围说明：

CAD 软件接受来自工程师的二维或三维几何数据。工程师通过用户界面与 CAD 系统进行交互，并控制它，该界面应表现出良好的人机界面设计的特征。所有几何数据及其他支持信息都保存在一个 CAD 数据库中。开发设计分析模块，以产生所需的输出，这些输出将显示在各种不同的图形设备上。软件在设计中要考虑与外设进行交互并控制它们，包括鼠标、数字化仪和激光打印机。

上述关于范围的说明是初步的——它并没有规定边界。每一句说明都应该进一步扩展，以提供具体的细节及定量的边界。例如，在估算开始之前，计划者必须确定“良好的人机界面设计的特征”是什么含义，或“CAD 数据库”的大小及复杂度是怎样的。

为了达到最终的目的，我们假设已经做了进一步的精化，并标识出了以下的主要软件功能：

- 用户界面及控制机制(UICF)。
- 二维几何分析(2DGA)。
- 三维几何分析(3DGA)。
- 数据库管理(DBM)。
- 计算机图形显示机制(CGDF)。
- 外设控制(PC)。
- 设计分析模块(DAM)。

遵照 LOC 的三点估算技术，能够建立如表 5－1 所示的表。例如，三维几何分析功能的 LOC 估算范围是：

乐观值：4600

可能值：6900

悲观值：8600

应用等式 5.1，得到三维几何分析功能的期望值是 6800LOC。这个值被输入到表中。其他估算也可以通过类似的方式获得。将 LOC 估算列的所有值相加，即得到该 CAD 系统的规模估算

表 5-1　LOC 方法的估算表

功能	LOC 估算
用户界面和控制机制(UICF)	2300
二维几何分析(2DGA)	5300
三维几何分析(3DGA)	6800
数据库管理(DBM)	3350
计算机图形显示机制(CGDF)	4950
外设控制(PC)	2100
设计分析模块(DAM)	8400
总代码行数估算	33200

对历史数据的复审表明：这类系统的平均生产率是 620LOC/pm。如果一个劳动力价格是每月 8000 美元，则每行代码的成本约为 13 美元。根据 LOC 估算及历史生产率数据，总的项目成本估算是 431000 美元，工作量估算是 54 个人月(估算的单位是 1000 美元及人月。取到元或十分之一人月的数学精确度是不现实的)。

5.6.4 一个基于 FP 估算的例子

基于 FP 估算的分解是集中于信息域值，而不是软件功能。根据图 4－5 给出的功能点计算表，项目计划者估算 CAD 软件的输入、输出、查询、文件、及外部接口。为了达到这个估算目的，我们假设复杂度加权因子都是平均的。表 5－2 给出了估算的结果

表 5－2 估算信息域值

信息域值	乐观值	可能值	悲观值	估算计数	加权因子	FP 计数
输入数	20	24	30	24	4	96
输出数	12	15	22	16	5	80
查询数	16	22	28	22	4	88
文件数	4	4	5	4	10	40
外部接口数	2	2	3	2	7	14
总计数值						318

接着，估算 14 个复杂度加权因子，并计算复杂度调整因子(如第 4 章所述)，表 5－3 给出了因子值。

表 5-3 计算复杂度调整因子

因子	值	因子	值
备份和复原	4	信息域值复杂度	5
数据通信	2	内部处理复杂度	5
分布式处理	0	设计成可复用的代码	4
关键性能	4	设计中的转换及安装	3
现有的操作环境	3	多次安装	5
联机数据登录	4	方便修改的应用设计	5
多屏幕输入切换	5	复杂度调整因子	1.17

最后，得出 FP 的估算值：

$$FP_{estimated}=总计数值\times[0.65+0.01\times\Sigma F_i]$$

$$FP_{estimated}=372$$

使用功能点进行规范化的历史数据表明：这类系统的组织平均生产率是 6.5FP/pm。如果一个劳动力价格是每月 8000 美元，则每个 FP 的成本约为 1230 美元。根据 FP 估算及历史生产率数据，总的项目成本估算是 457000 美元，工作量估算是 58 个人月。

5.6.5 基于过程的估算

估算一个项目的最常用的技术是基于使用的过程进行估算，即，将过程分解为相对较小的活动或任务，再估算完成每个任务所需的工作量。

与基于问题的估算一样，基于过程的估算也是开始于从项目范围中得到的软件功能描述。对于每一个功能，都必须执行一系列的软件过程活动。功能及相关的软件过程活动可以表示成类似于图 3－2 所示的表。

一旦建立了问题功能及相关的过程活动，计划者就可以估算完成每一个软件功能的软件过程活动所需的工作量(如人月)。这些数据包含在图 3-2 所示的表中的矩阵中。然后，将平均劳动力价格(成本/单位工作量)用来每一个软件过程活动的估算工作量，得到其成本估算。很有可能对于每一个任务而言，平均劳动力价格是不同的。高级职员一般从事项目早期的活动，且费用要高于低级职员，低级职员通常参与后期的设计任务、编码、及测试。

每一个功能及软件过程活动的成本及工作量在最后一步进行计算。如果基于过程的估算是独立于 LOC 或 FP 估算而进行的，则我们到目前为止已经有了两或三种成本及工作量的估算，它们之间可以进行比较和结合。如果两组估算基本一致，则有理由相信估算是可靠的。否则，如果这些技术得到的结果几乎没有相似性，则必须进行更进一步的调研及分析。

5.6.6 一个基于过程估算的例子

为了阐明基于过程估算的使用，我们还是考虑 5.6.3 节中所介绍的 CAD 系统。系统配置及所有软件功能都保持不变，且由项目范围来说明。

表 5－4 所示的已完成的基于过程的表显示了为每个 CAD 系统软件功能(为了简化进行了省略)所提供的软件工程活动的工作量估算(人月)。工程和建造及发布活动被划分为如图所示的主要软件工程任务。用户通信、计划、及风险分析的总的工作量估算被直接给出。这些数值都列在表底部的“合计”行中。水平的和垂直的合计为分析、设计、编码、及测试所需的工作量提供了一个估算的指标。应该注意到：53％的工作量是花费在“前端”的工程任务上(需求分析和设计)，说明了这些工作的重要性。

表 5-4　基于过程的估算表

活动→	用户通信	计划	风险分析		工程		建造发布	用户评估	总和
任务→				分析	设计	编码	测试		
功能									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
DSM	0.50	3.00	1.00	1.50	n	a	6.00	n/a	
CGDF				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
总和	0.25	0.25	0.25	3.50	20.50	4.75	16.50		46.00
% 工作量	1%	1%	1%	8%	45%	10%	36%		

如果平均一个劳动力价格是每月 8000 美元，则总的项目成本估算是 368000 美元，工作量估算是 46 个人月。如果需要的话，每一个软件过程活动均可关联不同的劳动力价格。

CAD 软件的总估算工作量的范围从最低的 46 个人月(使用基于过程的估算方法导出)到最高的 58 个人月(使用 FP 估算方法导出)。平均估算值是 53 个人月。与平均估算值的最大偏差约为百分之 13。

如果不同的估算方法得到的结果之间差别很大，是什么原因呢？这个问题的答案需要对估算所使用的信息进行再评估。估算之间差别很大一般能够追溯到下面两个原因中的一个：

1.项目的范围未能被充分理解，或被计划者误解。

2.基于问题的估算技术中所使用的生产率数据对于该应用是不合适的，或是太陈旧了(因为它已经不能正确地反映组织的情况)，或是被误用了。

计划者必须确定引起差别的原因，并调和各个估算结果。

5.7 经验估算模型

计算机软件的估算模型使用由经验导出的公式来预测工作量，工作量是 LOC 或 FP 的函数。LOC 或 FP 的值使用 5.6.2 节和 5.6.3 节所描述的方法进行估算。但不使用前述的表，而是将 LOC 或 FP 的值插入到估算模型中。

支持大多数估算模型的经验数据是来源于一个有限的项目样品集。因此，没有任何估算模型能够适用于所有类型的软件及所有开发环境。所以，从这种模型

中得到的结果必须谨慎地使用(一般讲，一个估算模型应能根据当前项目情况加以调整。该模型是根据已完成项目的结果导出。由该模型预测的数据应该与实际的结果进行比较，并针对当前情况评估该模型的功效。如果两个数据之间有较大偏差，则模型的指数及系数必须使用当前项目的数据进行重新计算)。

5.7.1 估算模型的结构

一个典型的估算模型是通过对以前的软件项目中收集到的数据进行回归分析而导出的。这种模型的总体结构具有下列形式[MAT94]：

$$E=A+B\times(ev)^{C} \qquad (5.2)$$

其中，A、B 和 C 是由经验导出的常数，E 是以人月为单位的工作量，而 ev 则是估算变量(LOC 或 FP)。除了等式(5.2)所标明的关系之外，大多数估算模型均有某种形式的项目调整成分，使得 E 能够根据其他的项目特性(如，问题的复杂性、开发人员的经验、开发环境等)加以调整。

在文献中提出了许多面向 LOC 的估算模型：

$E= 5.2\times(KLOC)^{0.91}$ Walston－Felix模型

$E=5.5+0.73\times(KLOC)^{1.16}$ Bailey-Basili模型

$E=3.2\times(KLOC)^{1.05}$ Boehm的简单模型

$E=5.288\times(KLOC)^{1.047}$ Doty模型，在KLOC＞9 的情况下

同样，也提出了许多面向 FP 的估算模型。主要包括：

$E=-13.39+0.0545FP$ Albrecht 和 Gaffney 模型

$E=60.62\times7.728\times10^{-8}FP^{3}$ Kemerer模型

$E=585.7+5.12FP$ Maston、Barnett 和 Mellichamp 模型

从上述的模型中可以很快看出：每一个模型对于相同的 LOC 或 FP 值，会产生出不同的结果(部分原因是因为这些模型一般都是仅从若干应用领域中相对很少的项目中推导出来的)。其含义很清楚，估算模型必须根据当前项目的需要进行调整。

5.7.2 COCOMO 模型

在其经典著作“软件工程经济学”(software engineering economics)中，BarryBoehm[BOE81]介绍了一种软件估算模型的层次体系，称为COCOMO(构造性成本模型，COnstructive COst MOdel)。Boehm的模型层次具有以下形式：

模型1：基本COC OMO模型，将软件开发工作量(及成本)作为程序规模的函数进行计算，程序规模以估算的代码行来表示。

模型2：中级COC OMO模型，将软件开发工作量(及成本)作为程序规模及一组“成本驱动因子”的函数来进行计算，其中“成本驱动因子”包括对产品、硬件、人员、及项目属性的主观评估。

模型3：高级COCOMO模型，包含了中级模型的所有特性，并结合了成本驱动因子对软件工程过程中每一个步骤(分析、设计等)的影响的评估。

为了阐明COCOMO模型，我们给出基本模型和中级模型的总体介绍。要想了解更详细的信息，可以参考[BOE81](COCOMO模型在本书写作期间正在进行修改。该模型可能会扩展以便包含FP测量，并以更多的项目为基础进行改进)。

COCOMO模型是为三种类型的软件项目而定义的。使用Boehm的术语来说，它们是：(1)组织模式——较小的、简单的软件项目，有良好应用经验的小型项目组，针对一组不是很严格的需求开展工作(如，为一个热传输系统开发的热分析程序)；(2)半分离模式——一个中等的软件项目(在规模和复杂性上)，具有不同经验水平的项目组必须满足严格的及不严格的需求(如，一个事务处理系统，对于终端硬件和数据库软件有确定需求)；(3)嵌入模式——必须在一组严格的硬件、软件及操作约束下开发的软件项目(如，飞机的航空控制系统)。

基本COCOMO模型具有以下形式：

$$E=a_b KLOC^{bb}$$

$$D=C_b E^{db}$$

其中，E是以人月为单位的工作量，D是以月表示的开发时间，KLOC是估算的项目代码行数(以千行为单位)。表5－5列出系数a_b和c_b及指数b_b和d_b。

表5-5　基本COCO■O模型

软件项目	a_b	b_b	c_b	d_b
组织模式	2.4	1.05	2.5	0.38
半分离模式	3.0	1.12	2.5	0.35
嵌入模式	3.6	1.20	2.5	0.32

为了考虑一组“成本驱动因子属性”[BOE81]扩展了基本模型，它们可以被分为四个主要类型：产品属性、硬性属性、人员属性及项目属性。这些类型共有

15 个属性，在 6 个级别上取值，从“很低”到“很高”(根据重要性或价值)。根据这个取值级别，可以从 Boehm[BOE81]提供的表中来确定工作量乘数，且所有工作量乘数的乘积就是工作量调整因子(effort adjustment factor，EAF)。EAF 的典型值是从 0.9 到 1.4 之间。

中级 COCOMO 模型具有以下形式：

$$E=a_i KLOC^{bi} \times EAF$$

其中，E 是以人月为单位的工作量，KLOC 是估算的项目代码行数(以千行为单位)。系数 ai 及指数 bi 由表 5-6 给出。

表 5-6　中级 COCOMO 模型

软件项目	a_i	b_i
组织模式	3.2	1.05
半分离模式	3.0	1.12
嵌入模式	2.8	1.20

COCOMO 代表了软件估算的一个综合经验模型。然而，Boehm 自己关于 COCOMO(可以扩展到所有模型)的评论 [BOE81] 应该引起注意：

今天，一个软件成本估算模型如果能够达到以下结果就相当不错了：估算的软件开发成本与实际的成本相差不到 20%，时间估算相差不到 70%，而且是在它自己的地盘上(即，是它适用的项目类型)……这可能不象我们所期望的那么精确，但已经足以在软件工程经济分析及决策中提供很大的帮助了。

为了阐明 COCOMO 模型的使用，我们采用基本模型对本章前面所描述的 CAD 软件范例进行估算。根据表 5－1 中给出的 LOC 估算和表 5－5 中所标注的系数及指数，我们利用基本模型可以得到：

$$E=2.4(KLOC)^{1.05}$$

$$=2.4(33.2)^{1.05}$$

=95 个人月

这个值比 5.6 节导出的几个估算值要高得多。因为 COCOMO 模型假设了比 5.6 节低得多的 LOC/pm，所以结果并不令人吃惊。为了适用于本范例中的问题，COCOMO 模型应该做适当调整。

要计算项目的持续时间，我们使用上面描述的工作量估算值：

$$D=2.5E^{0.35}$$

$=2.5(95)^{0.35}$

=12.3 个月

项目持续时间的估算值使得计划者能够为该项目确定一个合适的人员数目，N：

N=E/D

=95/12.3

=约 8 人

实际上，计划者可能决定只用四个人，并相应地延长项目的持续时间(应该注意到：工作量和时间之间的关系并不是线性的。因此，将人员减少至四人并不意味着项目需要增加一倍的时间。进一步的探讨可以参见文献[PUT92]。

5.7.3 软件方程式

软件方程式[PUT92]是一个多变量模型，它假设在软件开发项目的整个生命周期中的一个特定的工作量分布。该模型是从 4000 多个当代的软件项目中收集的生产率数据中导出的。基于这些数据，估算模型具有以下形式：

$$E=[LOC\times B^{0.333}/P]3\times(1/t^4) \qquad (5.3)$$

其中，E=以人月或人年为单位的工作量

t=以月或年表示的项目持续时间

B=“特殊技术因子”，它随着“对集成、测试、质量保证、文档、及管理技术的需求的增长”而缓慢增加。对于较小的程序(KLOC=5 到 15)，B=0.16。对于超过 70KLOC 的较大程序，B=0.39。

P=“生产率参数”，它反映了：

- 总体的过程成熟度及管理水平
- 良好的软件工程实践被使用的程度
- 使用的程序设计语言的级别
- 软件环境的状态
- 软件项目组的技术及经验

• 应用的复杂性

对于实时嵌入式软件的开发，典型值是 P=2000；对于电讯及系统软件，P=10000；而对于商业系统应用， P=28000。当前项目的生产率参数可以通过从以前的开发工作中收集到的历史数据中导出。

应该注意到：软件方程式有两个独立的参数：(1)规模的估算值(以 LOC 表示)，及(2)以月或年表示的项目持续时间。

为了简化估算过程，并将该模型表示成更为通用的形式，Putnam 和 Myers[PUT92]又提出了一组方程式，它们均从软件方程式中导出。最小开发时间被定义为：

$t_{min}=8.14(LOC/PP)^{0.43}$，以月表示，对于$t_{min}$＞6 个月的情况(5.4a)

$E=180Bt^3$，以人月表示，对于E≥20 的情况(5.4b)

注意等式(5.4b)中的 t 是以年表示的。

对本章前面所讨论的 CAD 软件使用等式(5.4)时，P=12000(对科学计算软件的推荐值)：

t_{min}=8.14(33，200/12，000)0.43

t_{min}=12.6 个月

E=180×0.28×(1.05)2

E＝58 个人月

由软件方程式得到的结果与 5.6 节的估算值非常符合。

5.8 自行开发或购买的决策

在许多软件应用领域中，直接获取(购买)计算机软件常常比自行开发的成本要低得多。软件工程管理者面临着是自行开发还是购买的决策，且因为有多种可选的获取方案使得决策更加复杂：(1)购买可直接使用的软件(或被授权使用)；(2)购买“具有完全经验”或“具有部分经验”的软件构件(见 5.4.2 节)，然后进行修改和集成，以满足特定的需求；或(3)软件可以由一个外面的承包商根据买方的规约定制开发。

软件获取的步骤根据对要购买的软件的要求程度及最终价格来定义。在某些情况下(如，低成本的 PC 软件)，直接购买并试验比起对可选的软件包进行冗长的评估要便宜得多。而对于比较昂贵的软件产品，可采用下列指导原则：

1.建立所需软件的功能及性能规约，定义任何可能的可测量特性。

2.估算内部开发的成本及交付日期。

3a.选择三到四个最符合你的需求的候选软件。

3b.选择能够有助于建造所需软件的可复用软件构件。

4.建立一个比较矩阵，对关键功能进行仔细比较。或者，进行基准测试，以比较候选软件。

5.根据以前产品的质量、开发商的支持、产品的方向、以及其名声，来评估每个候选软件包或构件。

6.联系该软件的其他用户并询问其意见。

在最后的分析中，自行开发或是购买的决策是根据以下条件决定的：(1)软件产品的交付日期是否比内部开发要快？(2)购买的成本加上定制的成本是否比内部开发该软件的成本要低？(3)外部支持的成本(如，一个维护合约)是否比内部支持的成本要少？这些条件可以用于上述的每一个可选的获取方案中。

5.8.1 创建决策树

前述的步骤可以使用统计技术进行扩充，如决策树分析[BOE89]。例如，图5－6刻画了一个软件系统X的决策树。在这个例子中，软件工程组织能够(1)从头开始建造系统X；(2)复用已有的“具有部分经验”的构件来构造系统；(3)购买现成的软件产品，并修改它，以满足当前项目的需要；或(4)将软件开发承包给外面的开发商。

如果从头开始建造系统，那么这项工作是困难的占百分之七十的概率。使用本章前面所讨论的估算技术，项目计划者预计：一个困难的开发工作将需要$450000的成本；而一个“简单的”开发工作则需要$380 000。沿决策树的任一分支进行计算，得到成本的预期值如下：

$$\text{预期成本}=\Sigma(\text{路径概率})i\times(\text{估算的路径成本})i$$

其中，i是决策树的某个路径。对于“建造系统”这条路径而言：

$$\text{预期成本}_{build}=0.30(\$380K)+0.70(\$450K)=\$429K(K：千元)$$

沿着决策树的其他路径，“复用”、“购买”、及“承包”情况下的预期成本也可被给出。这些路径的预期成本如下：

$$\text{预期成本}_{reuse}=0.40(\$275K)+ 0.60[0.20(\$310K)+0.80(\$490K)]=\$382K$$

$$\text{预期成本}_{buy}=0.70(\$210K)+0.30(\$400K)=\$267K$$

预期成本$_{contract}$=0.60($350K)+ 0.40($500K)=$410K

根据图 5－3 给出的路径概率及估算成本，可以知道“购买”方式具有最低的预期成本。

不过，应该注意到：在决策过程中许多标准(而不仅仅是成本)必须加以考虑。可用性、开发人员/厂家/承包商的经验、与需求的一致性、当前项目的“策略”、以及修改的可能性等，这些都是可能影响最终决定，使用建造、复用、购买或承包方式的标准，当然这也仅仅是其中一部分标准而已。

5.8.2 外购

每一个开发计算机软件的公司或迟或早都会问到一个基本问题：“是否有什么方法使得我们能够以较低的成本得到所需的软件和系统？”。这个问题的答案不是唯一的，但对于这个问题的情绪化的回答经常是一句话：“外购”。

在概念上，外购是非常简单的。软件工程活动被承包给第三方厂商，他们能够以较低的成本和较高的质量来完成这项工作。公司内部需要做的软件工作已经降至仅仅是合同管理活动。

Edward Yourdon[YOU92]在讨论对于外购的不断增长的趋势时说：

如果你已经培养了一种文化理念，认为美国的软件产业是世界的领袖，那么我只是想问你是否还记得仅在几年前对我们的汽车产业也有同样的想法……软件开发可能会从美国移至其他许多国家的软件工厂中，那里的人们也受过良好教育，成本较低，且更加投入地提高质量和生产率。

多么有力的陈述！而且现在已经成为事实。

外购的决策可以从战略及战术两个级别进行考虑。在战略级，商业管理要考虑是否软件工作的绝大部分可以承包给其他厂商。在战术级，项目管理者要确定是否项目的部分或全部能够通过外包部分软件工作的方式而被很好地完成。

不考虑其他很多因素，外购的决策常常是财政的决策。关于外购的财政分析的详细探讨超出了本书的范围，最好留给其他人去研究［MIN95］。不过，从正反两方面考察一下外购的决策是值得的。

从正面来看，通过减少软件人员及相应设备(如，计算机、基础设施等)通常能够节约成本。而从反面来看，公司失去了对其所需软件的部分控制权。因为软件是一种技术，使得它区别于公司的系统、服务、及产品，所以，公司冒着将其命运交到第三方厂商手中的风险。

外购的趋势毫无疑问会持续下去。减缓这个趋势的唯一方法是认识到：在21世纪，软件工作在所有层次上都会有激烈竞争。生存的唯一方法是使自己变得和外购的厂商一样具有竞争力。

5.9 自动估算工具

前面几节中所描述的分解技术和经验估算模型已在很多软件工具中得以实现。这些自动估算工具使得计划者能够估算成本及工作量，并对重要的项目变量诸如交付日期或人员进行“What－if”(什么-假如)分析。虽然已有许多自动估算工具存在，但它们都具有相同的一般特性，且都需要以下的一种或几种数据：

1.对于项目规模(如LOC)或功能(如功能点数据)的定量估算。

2.定性的项目特性，如复杂性、所要求的可靠性、或商业上的紧迫度。

3.对于开发人员和/或开发环境的描述。

根据这些数据，由自动估算工具所实现的模型就能够提供关于以下信息的估算：完成该项目所需的工作量、成本、人员配置、持续时间、以及在某些情况下，开发进度及相关的风险。

Martin[MAR88]在这些自动估算工具之间进行了有趣的比较。若干不同的工具被用于同一个项目中。得到的估算结果之间存在很大偏差，这并不令人吃惊。更重要的是，估算值有时与实际值之间有显著的差异。这再次说明了估算工具的输出仅应该作为一个“数据点”，从中导出估算值——而不是作为估算的唯一来源。

5.10 小结

软件项目计划者在项目开始之前必须先估算三件事：需要多长时间、需要多少工作量、以及需要多少人员。此外，计划者必须预测所需要的资源(硬件及软件)和包含的风险。

范围说明能够帮助计划者使用一种或多种技术进行估算，这些技术主要分为两大类：分解和经验建模。分解技术需要划分出主要的软件功能，接着估算实现每一个功能所需的程序规模或人月数。经验技术使用根据经验导出的公式来预测工作量和时间。可以使用自动工具实现某一特定的经验模型。

精确的项目估算一般至少会用到上述三种技术中的两种。通过比较和调和使用不同技术导出的估算值，计划者更有可能得到精确的估算。软件项目估算永远不会是一门精确的科学，但将良好的历史数据与系统化的技术结合起来能够提高估算的精确度。

思考题

5.1 假设你是一家为用户开发软件的公司的项目管理者。你承接了一个家庭保安系统的软件开发项目。写一个范围说明来描述该软件，确定你的范围说明是界定的。如果你对家庭保安系统不熟悉，在你开始写作之前先做一些调研工作。你也可以选择其他你感兴趣的问题，而不做家庭保安系统。

5.2 在 5.1 节简要地讨论了软件项目的复杂性。建立一个表，列出可能会影响项目复杂性的软件特性(如，并发操作、图形输出等)。按其对项目影响的程度排列这些特性。

5.3 在计划过程中，性能是一个重要的考虑因素。讨论针对不同的软件应用领域，是如何以不同的方式解释性能的。

5.4 根据你在问题 5.1 中所述的家庭保安系统的软件范围，进行功能分解。以 LOC 估算每个功能的规模。假设你所在的公司平均生产率是 450LOC/pm，且平均劳动力价格是每人月 7000 美元，使用 5.6.3 节讨论的基于 LOC 的估算技术来估算建造该软件所需的工作量及成本。

5.5 使用第 4 章所述的 3D 功能点测量，计算家庭保安系统软件的 FP 值，并使用 5.6.4 节所讨论的基于 FP 的估算技术导出工作量及成本的估算值。

5.6 使用 COCOMO 模型估算家庭保安系统软件。假设采用基本模型。

5.7 使用“软件方程式”估算家庭保安系统软件。假设采用等式(5.4)且 P＝8000。

5.8 比较从问题 5.4 到问题 5.7 所得到的工作量估算值。使用三点估算法导出该项目的单一估算值。求出标准偏差，它对你关于估算的确信程度有何影响？

5.9 使用从问题 5.8 中得到的结果，确定是否有可能期望该软件能在 6 个月之内完成，及要完成该工作需要多少人员？

5.10 建立一个电子表格模型，实现本章所述的一个或多个估算技术。

5.11 建立一个项目组：开发一个软件工具，实现本章所述的每一个估算技术。

5.12 有一点似乎很奇怪：成本和进度估算是在软件项目计划期间完成的——在详细的软件需求分析或设计进行之前。你认为为什么会这样？是否存在不需要这样做的情况？

5.13 假设每个分支的概率均为 50-50，重新计算图 5-6 中决策树上所标注的预期值。这会改变你的最终决定吗？

5.14 在行业文献中(如，Computerworld，Software Magazine 等)查阅最近的关于外购的文章。写一个 2 到 3 页的文章描述你从中学到了什么。

推荐阅读文献及其他信息源

Bennatan [BEN92]，Wellman(Software Costing，Prentice-Hall，1992)，和 Londeix(CostEstimation for Software Development，Addison-Wesley，1987)等人的书中包含了很多关于软件项目计划及估算的有用信息。 Lederer 和 Prasad(“Nine Management Guidelines for Better CostEstimating”，Communications of the ACM，February 1992)提供了一组有用的统计信息，可以作为进行更好的成本估算的指南。

Putnam 和 Myer 的关于软件成本估算的详细论述 [PUT92] 以及 Boehm 的关于软件工程经济学的书 [BOE81] 都描述了软件的经验估算模型。两本书中都提供了对于来自成百上千个软件项目的数据的详细分析。 DeMarco 的一本优秀论著(Controlling Software Projects， Yourdon Press，1982)提供了关于软件项目的管理、测量及估算的有价值的观点。 Sneed(Software EngineeringManagement，Wiley，1989)和 Macro(Software Engineering：Concepts and Management，Prentice-Hall，1990)非常细致地探讨了如何进行软件项目估算。

代码行成本估算是产业界中使用得最广泛的一种方法。不过，面向对象范型(见第四部分)的影响可能会使某些估算模型无效。Lorenz 和 Kidd(面向对象的软件度量，Object-OrientedSoftware Metrics，Prentice-Hall，1994)探讨了针对面向对象系统的估算。

COCOMO 模型的新版本，在 90 年代进行了更新，可以在下面的网址上得到：

http：//sunset.usc.edu/COCOMO2.0/Cocomo.html

经常提出的问题(FAQ)及项目管理术语表可在以下站点找到：

http：//www.wst.com/projplan/proj-plan.FAQ.html

http：//smurfland.cit.buffalo.edu/NetMan/FAQs/proj-plan.FAQ.html

http：//www.wst.com/projplan/proj-plan.glossary.html

有关软件项目管理工具的信息可在下面的网址上获得：

http：//www.wst.com/projplan/proj-plan.reviews.html

关于项目管理资源的列表可以在下面的网址得到：

http：//www.pmi.org/pmi/mem_prod/pmmisc.htm

项目成本分析的详细报告及大量参考文献可在下面的网址上获得：

http：//mijuno.larc.nasa.gov/dfc/pca.html

一个最新的关于软件项目估算的WWW参考目录可在下面的网址上找到：
http：//www.rspa.com

① 这并不意味着修改最初的估算一定是要发生的。一个成熟的软件组织及其管理者会认识到不是可以随意修改计划的。许多用户要求（这是不正确的）一旦做了一个估算就必须一直维持它，而忽略了环境的变化。

第 6 章　风险管理

Robert Charette [CHA89] 在他的关于风险分析及管理的书中给出了风险概念的定义是：

首先，风险关注未来将要发生的事情。今天和昨天已不再被关心，如同我们已经在收获由我们过去的行为所播下的种子。问题是：我们是否能够通过改变我们今天的行为，而为一个不同的、充满希望的、更美好的明天创造机会。其次，这意味着，风险涉及改变，如思想、观念、行为、或地点的改变……第三，风险涉及选择及选择本身所包含的不确定性。因此，就象死亡和税收一样，风险是生活中最不确定的元素之一。

当在软件工程领域考虑风险时，Charette 的三个概念定义是显而易见的。未来是我们所关心的——什么样的风险会导致软件项目彻底失败呢？改变也是我们所关心的——用户需求、开发技术、目标计算机、以及所有其他与项目相关的因素的改变将会对按时交付和总体成功产生什么影响呢？最后，我们必须抓住选择机会——我们应该采用什么方法及工具？需要多少人员参与工作？对质量的要求要达到什么程度才是“足够的”？

Peter Drucker [DRU75] 曾经说过：“当没有办法消除风险，甚至连试图降低该风险也存在疑问时，这些风险就是真正的风险了”。在我们能够标识出软件项目中的“真正风险”之前，识别出所有对管理者及开发者而言均为明显的风险是很重要的。

6.1 被动和主动的风险策略

被动风险策略被戏称为“印地安那·琼斯学派的风险管理”[THO92]。印地安那·琼斯在以其名字为影片名的电影中，每当面临无法克服的困难时，总是一成不变地说：“不要担心，我会想出办法来的！”。印地安那·琼斯从不担心任何问题，直到它们发生，再做出英雄式的反应。

遗憾的是，一般的软件项目管理者并不是印地安那·琼斯，且软件项目组的成员也不是他的可信赖的伙伴。大多数软件项目组还是仅仅依赖于被动风险策

略。被动策略最多不过是针对可能发生的风险来监督项目，直到它们变成真正的问题时，才会拨出资源来处理它们。更普遍的情况是，软件项目组对于风险不闻不问，直到发生了错误，这时，项目组才赶紧采取行动，试图迅速地纠正错误。这常常被称为“救火模式”。当这样的努力失败后，“危机管理”［CHA92］接管一切，这时项目已经处于真正的危机中了。

对于风险管理的一个更聪明的策略是主动式的。主动策略早在技术工作开始之前就已经启动了。标识出潜在的风险，评估它们出现的概率及产生的影响，且按重要性加以排序，然后，软件项目组建立一个计划来管理风险。主要的目标是预防风险，但因为不是所有的风险都能够预防，所以，项目组必须建立一个意外事件的计划，使其在必要时能够以可控的及有效的方式作出反应。在本章其余部分，我们将讨论风险管理的主动策略。

6.2 软件风险

虽然对于软件风险的严格定义还存在很多争议，但在风险中包含了两个特性这一点上是已达成了共识的［HIG95］：

• 不确定性——刻划风险的事件可能发生也可能不发生；即，没有 100％发生的风险(100％发生的风险是加在项目上的约束)。

• 损失——如果风险变成了现实，就会产生恶性后果或损失。

进行风险分析时，重要的是量化不确定性的程度及与每个风险相关的损失的程度。为了实现这点，必须考虑不同类型的风险。

项目风险威胁到项目计划。也就是说，如果项目风险变成现实，有可能会拖延项目的进度，且增加项目的成本。项目风险是指潜在的预算、进度、人力(工作人员及组织)、资源、客户、及需求等方面的问题以及它们对软件项目的影响。在第 5 章中，项目复杂性、规模、及结构不确定性也被定义为项目(估算)风险因素。

技术风险威胁到要开发软件的质量及交付时间。如果技术风险变成现实，则开发工作可能变得很困难或根本不可能。技术风险是指潜在的设计、实现、接口、验证、和维护等方面的问题。此外，规约的二义性、技术的不确定性、陈旧的技术、及“先进的”技术也是风险因素。技术风险的发生是因为问题比我们所设想的更加难以解决。

商业风险威胁到要开发软件的生存能力。商业风险常常会危害项目或产品。五个主要的商业风险是：(1)开发了一个没有人真正需要的优秀产品或系统(市场风险)；(2)开发的产品不再符合公司的整体商业策略(策略风险)；(3)建造了一个销售部门不知道如何去卖的产品；(4)由于重点的转移或人员的变动而失去了高级管理层的支持(管理风险)；以及(5)没有得到预算或人力上的保证(预算风

险)。应该注意到的很重要的一点是：简单的分类并不总是行得通。某些风险根本无法事先预测。

另一种常用的分类方式是由 Charette [CHA89] 提出的。已知风险是通过仔细评估项目计划、开发项目的商业及技术环境、以及其他可靠的信息来源(如，不现实的交付时间，没有需求或软件范围的文档、恶劣的开发环境)之后可以发现的那些风险。可预测风险能够从过去项目的经验中推断出来(如，人员调整、与客户之间无法沟通、由于需要进行维护而使开发人员精力分散)。不可预测风险就象纸牌中的大王，它们可能、也会真的出现，但很难事先识别出它们来。

6.3 识别风险

识别风险是试图系统化地确定对项目计划(估算、进度、资源分配)的威胁。通过识别已知的和可预测的风险，项目管理者已经迈出了第一步——在可能时避免这些风险，且当必要时控制这些风险。

在 6.2 节中提出的每一类风险又分为两个不同的类型：一般性风险和特定产品的风险。一般性风险对每一个软件项目而言都是一个潜在的威胁。特定产品的风险只有那些对当前项目的技术、人员、及环境非常了解的人才能识别出来。为了识别特定产品的风险，必须检查项目计划及软件范围说明，并给出以下问题的答案：“本项目中有什么特殊的特性可能会威胁到我们的项目计划？”

一般性风险和特定产品的风险都应该被系统化地标识出来。 Tom Gilb [GIL88] 很贴切地表达了这点：“如果你不主动攻击风险，风险就会主动攻击你”。

识别风险的一个方法是建立风险条目检查表。该检查表可以用于识别风险，并使得人们集中来识别下列常见子类型中的已知的及可预测的风险：

• 产品规模——与要建造或要修改的软件的总体规模相关的风险。

• 商业影响——与管理或市场所加诸的约束相关的风险。

• 客户特性——与客户的素质以及开发者和客户定期通信的能力相关的风险。

• 过程定义——与软件过程被定义的程度以及它们被开发组织所遵守的程度相关的风险。

• 开发环境——与用以建造产品的工具的可用性及质量相关的风险。

• 建造的技术——与待开发软件的复杂性及系统所包含技术的“新奇性”相关的风险。

• 人员数目及经验——与参与工作的软件工程师的总体技术水平及项目经验相关的风险。

风险条目检查表能够以不同的方式来组织。与上述每个话题相关的问题可以由每一个软件项目来回答。这些问题的答案使得计划者能够估算风险产生的影响。我们也可以采用另一个不同的风险条目检查表，它仅仅列出与每一个常见子类型有关的特性。最后，列出一组“风险元素和驱动因子”［AFC88］以及它们发生的概率。关于性能、支持、成本、及进度的驱动因子将在以后讨论。

6.3.1 产品规模风险

有经验的管理者几乎都对下面的陈述没有异议：项目风险是直接与产品规模成正比的。下面的风险检查表中的条目标识了与产品(软件)规模相关的常见风险：

• 是否以 LOC 或 FP 估算产品的规模？

• 对于估算出的产品规模的信任程度如何？

• 是否以程序、文件或事务处理的数目来估算产品规模？

• 产品规模与以前产品的规模平均值的偏差百分比是多少？

• 产品创建或使用的数据库大小如何？

• 产品的用户数有多少？

• 产品的需求改变多少？交付之前有多少？交付之后有多少？

• 复用的软件有多少？

在每一种情况下，待开发产品的信息必须与过去的经验加以比较。如果出现了较大的百分比偏差，或者如果数字相近但过去的结果很不令人满意，则风险较高。

6.3.2 商业影响风险

有一个大型软件公司的工程经理在他的墙上挂了一个镜框，上面写着：“上帝给了我头脑使我成为一个优秀的项目管理者，同时每当销售部门设定项目的最后期限时，也让我经历了地狱般的煎熬”。销售部门是受商业驱动的，而商业考虑有时会直接与技术现实发生冲突。下面的风险检查表中的条目标识了与商业影响相关的常见风险：

• 本产品对公司的收入有何影响？

- 本产品是否得到公司高级管理层的重视？
- 交付期限的合理性如何？
- 将会使用本产品的用户数及本产品是否与用户的需要相符合？
- 本产品必须能与之互操作的其他产品/系统的数目？
- 最终用户的水平如何？
- 必须产生并交付给用户的产品文档的量与质如何？
- 政府对本产品开发的约束？
- 延迟交付所造成的成本消耗是多少？
- 产品缺陷所造成的成本消耗是多少？

对于待开发产品的每一个回答都必须与过去的经验加以比较。如果出现了较大的百分比偏差，或者如果数字相近但过去的结果很不令人满意，则风险较高。

6.3.3 客户相关的风险

并非所有客户都是一样的。Pressman 和 Herron [PRE91] 在讨论这个话题时曾经说过：

客户有不同的需要。一些人知道他们需要什么；而另一些人知道他们不需要什么。一些客户希望进行详细讨论，而另一些客户则满足于模糊的承诺。

客户有不同的个性。一些人喜欢享受客户的身份——紧张、谈判、一个好产品带来的心理满足；而另一些人则根本不喜欢作为客户。一些人会高兴地接受几乎任何交付的产品，并能充分利用一个不好的产品；而另一些人则会对质量差的产品猛烈抨击。一些人会对质量好的产品表示他们的赞赏；而另一些人则不管怎样都会抱怨不休。

客户和他们的供应商之间也有各种不同的通信方式。一些人非常熟悉产品及生产厂商；而另一些人则可能素未谋面，仅仅通过信件往来和几个匆忙的电话与生产厂商沟通。

客户常常是矛盾的。他们希望昨天的一切工作都是免费的。生产厂商经常陷入客户自己的矛盾之中。

一个“不好的”客户可能会对一个软件项目组能否在预算内按时完成项目产生很大的影响。对于项目管理者而言，不好的客户是对项目计划的巨大威胁和实际的风险。下面的风险检查表中的条目标识了与客户特征相关的常见风险：

• 你以前是否曾与这个客户合作过？

• 该客户是否很清楚需要什么？他能否花时间把需求写出来？

• 该客户是否同意花时间召开正式的需求收集会议(第 11 章)，以确定项目范围？

• 该客户是否愿意建立与开发者之间的快速通信渠道？

• 该客户是否愿意参加复审工作？

• 该客户是否具有该产品领域的技术素养？

• 该客户是否愿意让你的人来做他们的工作，即，当你的人在做具体的技术工作时，该客户是否会坚持在旁边监视？

• 该客户是否了解软件过程？

如果对于这些问题中的任何一个的答案是否定的，则需要进行进一步的调研，以评估潜在的风险。

6.3.4 过程风险

如果软件过程(第 2 章)定义得不清楚；如果分析、设计、及测试以无序的方式进行；如果质量是每个人都认为很重要的概念，但没有人切实地采取行动来保证它，那么，这个项目就处于风险之中。以下问题摘自一次由 R.S.Pressman & Associates，Inc.［PRE95］建立的对软件工程实践活动进行评估的研讨会。这些问题已经在软件工程研究所(SEI)的过程评估调查表中进行了改编。

过程问题

• 你的高级管理层是否支持一份已经写好的政策综述，该综述中强调了软件开发标准过程的重要性吗？

• 你的组织是否已经建立了一份已经成文的、用于本项目的软件过程的说明？

• 开发人员是否“签约”同意按照文档所写的软件过程进行开发工作，并自愿使用它？

• 该软件过程是否可以用于其他项目？

• 你的组织是否已经为管理者及技术人员开设了一系列的软件工程培训课程？ • 是否为每一个软件开发者和管理者都提供了印好的软件工程标准？

• 是否为作为软件过程一部分而定义的所有交付物建立了文档概要及示例？

• 是否定期地对需求规约、设计和编码进行正式的技术复审？

• 是否定期地对测试过程和测试情况进行复审？

• 是否对每一次正式技术复审的结果要建立了文档，其中包括发现的错误及使用的资源？

• 是否有什么机制来保证软件工程标准确认的方案指导的工作开展正常？ • 是否使用配置管理来维护系统/软件需求、设计、编码及测试用例之间的一致性？

• 是否使用一个机制来控制用户需求的变化及其对软件的影响？

• 对于每一个承包出去的子合同，是否有一份文档化的工作说明、一份软件需求规约及一份软件开发计划？

• 是否有一个可遵循的规程，来跟踪及复审子合同承包商的工作？

技术问题

• 是否使用方便易用的规格说明技术来辅助客户与开发者之间的通信？

• 是否使用特定的方法进行软件分析？

• 是否使用特定的方法进行数据和体系结构的设计？

• 是否百分之 90 以上的代码都是采用高级语言编写的？

• 是否定义及使用特定的规则进行代码编写？

• 是否使用特定的方法进行测试用例设计？

• 是否使用软件工具来支持计划和跟踪活动？

• 是否使用配置管理软件工具来控制和跟踪软件过程中的变化活动？

• 是否使用软件工具来支持软件分析和设计过程？

• 是否使用工具来创建软件原型？

• 是否使用软件工具来支持测试过程？

• 是否使用软件工具来支持文档的生成和管理？

• 是否收集所有软件项目的质量度量值？

• 是否收集所有软件项目的生产率度量值？

如果对于上述问题中大多数的答案是否定的，则软件过程是薄弱的，且风险很高。

6.3.5 技术风险

突破技术的限制是极具挑战性且令人兴奋的，这是几乎每一个技术人员的梦想，因为这迫使开发人员使出他的或她的浑身解数，但这也是很有风险的。Murphy定律似乎对开发工作中的这一部分有了控制，使得我们难以预测风险，更不用说对它们进行计划了。下面的风险检查表中的条目标识了与建造的技术相关的常见风险：

• 该技术对于你的组织而言是新的吗？

• 客户的需求是否需要创建新的算法或输入、输出技术？

• 软件是否需要使用新的或未经证实的硬件接口？

• 待开发软件是否需要与开发商提供的未经证实的软件产品接口？

• 待开发软件是否需要与其功能及性能均未在本领域中得到证实的数据库系统接口？

• 产品的需求中是否要求采用特定的用户界面？

• 产品的需求中是否要求开发某些程序构件，这些构件与你的组织以前所开发过的构件完全不同？

• 需求中是否要求使用新的分析、设计、或测试方法？

• 需求中是否要求使用非传统的软件开发方法，如形式化方法、基于 AI 的方法、以及人工神经网络？

• 需求中是否有过份的对产品的性能约束？

• 客户能确定所要求的功能是“可行的”吗？

如果对于这些问题中的任何一个的回答是肯定的，则需要进行进一步的调研，来评估潜在的风险。

6.3.6 开发环境风险

如果一个木匠被要求用弯曲的、钝的手锯制作一件好家具，则最终产品的质量肯定是令人怀疑的。即使是熟练的开发者，不适当的或没有效率的工具也会阻碍工作的进行。软件工程环境支持项目组、过程及产品。但是，如果环境有缺陷，它就可能成为重要的风险源。下面的风险检查表中的条目标识了与开发环境相关的常见风险(第 29 章讨论了本检查表中所列的工具种类)：

- 是否有可用的软件项目管理工具？
- 是否有可用的软件过程管理工具？
- 是否有可用的分析及设计工具？
- 分析及设计工具是否支持适用于待建造产品的方法？
- 是否有可用的编译器或代码生成器，且适用于待建造产品？
- 是否有可用的测试工具，且适用于待建造产品？
- 是否有可用的软件配置管理工具？
- 环境是否利用了数据库或仓库？
- 是否所有软件工具都是彼此集成的？
- 项目组的成员是否已经接受过关于每个工具的培训？
- 是否有相关的专家能够回答关于工具的问题？
- 工具的联机帮助及文档是否适当？

如果对于上述问题中大多数的回答是否定的，则软件开发环境是薄弱的，且风险很高。

6.3.7 与人员数目及经验相关的风险

Boehm [BOE89] 建议了以下问题可用于评估与人员数目及经验相关的风险：

- 是否有最优秀的人员可用？
- 人员在技术上是否配套？
- 是否有足够的人员可用？
- 开发人员是否能够自始自终地参加整个项目的工作？

• 项目中是否有一些人员只能部分时间工作？

• 开发人员对自己的工作是否有正确的期望？

• 开发人员是否接受过必要的培训？

• 开发人员的流动是否仍能保证工作的连续性？

如果对于这些问题中的任何一个的回答是否定的，则需要进行进一步的调研，以评估潜在的风险。

6.3.8 风险因素和驱动因子

美国空军［AFC88］写了一本小册子，其中包含了如何很好地识别和消除软件风险的指南。

他们所用的方法要求项目管理者标识影响软件风险因素的风险驱动因子，这些因素包括性能、成本、支持和进度。在本讨论中，风险因素是以如下的方式定义的：

• 性能风险——产品能够满足需求且符合于其使用目的的不确定的程度。

• 成本风险——项目预算能够被维持的不确定的程度。

• 支持风险——软件易于纠错、适应及增强的不确定的程度。

• 进度风险——项目进度能够被维持且产品能按时交付的不确定的程度。

每一个风险驱动因子对风险因素的影响均可分为四个影响类别——可忽略的、轻微的、严重的及灾难性的。表 6－1［BOE89］指出了由于错误而产生的潜在影响(标为 1 的行)或没有达到预期的结果所产生的潜在影响(标为 2 的行)。影响类别的选择是以最符合表中描述的特性为基础的。

表 6 － 1　影响评估［ BOE89 ］

因素 / 类别		性能	支持	成本	进度
灾难的	1	无法满足需求而导致任务失败		错误将导致进度延迟和成本增加，预计超支$500K	
	2	严重退化使得根本无法达到要求的技术性能	无法做出响应或无法支持的软件	严重的资金短缺，很可能超出预算	无法在交付日期内完成
严重的	1	无法满足需求而导致系统性能下降，使得任务能否成功受到质疑		错误将导致操作上的延迟，并使成本增加， 预计超支$100K 到$500K	
	2	技术性能有些降低	在软件修改中有少量的延迟	资金不足，可能会超支	交付日期可能延迟
轻微的	1	无法满足需求，而导致次要任务的退化		成本、影响和或可恢复的进度上的小问题预计超支$1K 到$100K	
	2	技术性能有较小的降低	较好的软件支持	有充足的资金来源	实际的、可完成的进度计划
可忽略的	1	无法满足需求而导致使用不方便或不易操作		错误对进度及成本的影响很小，预计超支少于$1K	
	2	技术性能不会降低	易于进行软件支持	可能低于预算	交付日期将会提前

注： 1.未测试出的软件错误或缺陷所产生的潜在影响。

2.如果没有达到预期的结果所产生的潜在影响。

6.4 风险预测

风险预测，又称风险估算，试图从两个方面评估每一个风险——风险发生的可能性或概率，以及如果风险发生了，所产生的后果。项目计划者，以及其他管理人员和技术人员，一起执行四个风险预测活动：(1)建立一个尺度，以反映风险发生的可能性；(2)描述风险的后果；(3)估算风险对项目及产品的影响；(4)标注风险预测的整体精确度，以免产生误解。

6.4.1 建立风险表

风险表给项目管理者提供了一种简单的风险预测技术(风险表应该采用电子表格来实现，这样使得表中的内容易于操纵及排序)。风险表的样本如图 6－2 所示。

项目组一开始要在表中的第一列列出所有风险(不管多么细微)。这可以利用 6.3 节所述的风险检查表条目来完成。每一个风险在第二列上加以分类(如，PS 指产品规模风险，BU 指商业风险)。每个风险发生的概率则输入到第三列中。每个风险的概率值可以由项目组成员个别估算，然后将这些单个值求平均，得到一个有代表性的概率值。下一步是评估每个风险所产生的影响。使用表 6-1 所述的特性评估每个风险因素，并确定其影响的类别。对四个风险因素——性能、支持、

成本、及进度——的影响类别求平均可得到一个整体的影响值(如果其中一个风险因素对项目特别重要，也可以使用加权求平均值)。

表 6-2　分类前的风险表样本

风险	类别	概率	影响	RMMM
规模估算可能非常低	PS	60％	2	
用户数量大大超出计划	PS	30％	3	
复用程度低于计划	PS	70％	2	
最终用户抵制该系统	BU	40％	3	
交付期限将被紧缩	BU	50％	2	
资金将会流失	CU	40％	1	
用户将改变需求	PS	80％	2	
技术达不到预期的效果	TE	30％	1	
缺少对工具的培训	DE	80％	3	
人员缺乏经验	ST	30％	2	
人员流动比较频繁	ST	60％	2	
▪				
▪				
▪				

影响类别取值：

1 一灾难的
2 一严重的
3 一轻微的
4 一可忽略的

一旦完成了风险表的前四列内容，就要根据概率及影响来进行排序。高发生概率、高影响的风险放在表的上方，而低概率风险则移到表的下方。这样就完成了第一次风险排序。

项目管理者研究已排序的表，并定义一条中止线。该中止线(表中某一点上的一条水平线)表示：只有那些在线之上的风险才会得到进一步的关注。而在线之下的风险则需要再评估以完成第二次排序。

风险影响及概率从管理的角度来考虑，是起着不同的作用的(见图 6－1)。一个具有高影响但发生概率很低的风险因素不应该花费太多的管理时间。而高影响且发生概率为中到高的风险、以及低影响且高概率的风险，应该首先列入管理考虑之中。

所有在中止线之上的风险都必须进行管理。标有 RMMM 的列中包含了一个指示器，指向为所有中止线之上的风险所建立的风险缓解、监控、及管理计划(Risk Mitigation，Monitoringand Management Plan)。RMMM 计划将在 6.5 节讨论。

风险概率的确定可以通过先做个别估算而后求出一个有代表性的值来完成。虽然该方法是可行的，不过仍存在很多其他确定风险概率的更加复杂的技术

［AFC88］可供使用。风险驱动因子的评估是以一个定性的概率尺度：不可能、不一定、可能和极可能为基础，然后，根据每一个定性值相关的数学概率值(如，概率为 0.7 到 1.0 表示极可能发生的风险)来计算的。

6.4.2 评估风险影响

如果风险真的发生了所产生的后果有三个因素可能会受影响：风险的性质，范围，及时间。风险的性质是指当风险发生时可能产生的问题。例如，一个定义得很差的与客户硬件的外部接口(技术风险)会妨碍早期的设计及测试，也有可能导致项目后期阶段的系统集成问题。风险的范围结合了严重性(即风险有多严重？)及其整体分布情况(项目中有多少部分受到影响或有多少用户受到损害？)。最后，风险的时间主要考虑何时能够感到风险及持续多长时间。在大多数情况下，项目管理者希望“坏消息”越早出现越好，但在某些情况下，越迟越好。

让我们再回到美国空军提出的风险分析方法上来［AFC88］。以下的步骤被建议用来确定风险的整体影响：

1.确定每个风险元素发生的平均概率。

2.使用表 6－1，基于其中列出的标准来确定每个因素的影响。

3.按照前面几节给出的方法完成风险表，并分析其结果。

6.4.1 节和 6.4.2 节所述的风险预测和分析技术可以在软件项目进展过程中迭代使用。项目组应该定期复查风险表，再评估每一个风险，以确定新的情况是否引起其概率及影响发生改变。这个活动的结果可能需要在表中添加一些新风险，删除一些不再有影响的风险，并改变风险的相对位置。

6.4.3 风险评估

在风险管理中的这一步，我们建立了如下形式的一系列三元组［CHA89］：

$$[r_i, l_i, x_i]$$

其中r_i表示风险，l_i表示风险发生的概率，x_i则表示风险产生的影响。在风险评估过程中，我们进一步审查在风险预测阶段所做的估算的精确度，试图为所发现的风险排出优先次序，并开始考虑如何控制和/或避免可能发生的风险。

要使评估发生作用，必须定义一个风险参考水平值［CHA89］。对于大多数软件项目而言，前面所讨论的风险因素——性能、成本、支持、及进度——也代表示了风险参考水平值。即，对于性能下降、成本超支、支持困难、或进度延迟(或者这四种的组合)，都有一个水平值的要求，超过它就会导致项目被迫终止。

如果风险的组合所产生的问题引起一个或多个参考水平值被超过，则工作将会停止。在软件风险分析中，风险参考水平值存在一个点，称为参考点或临界点，在这个点上决定继续进行该项目或终止它(问题太大了)都是可以接受的。

图 6－2 以图形方式表示了这种情况。如果风险的组合产生问题而导致成本超支及进度延迟，则会有一个水平值，即图中所示的曲线，当超过它时会引起项目终止(阴影区域)。在临界点上，决定继续进行或终止项目都是可以的。

实际上，参考水平很少能表示成如图所示的一条光滑曲线。在大多数情况下，它是一个区域，其中存在很多不确定性(即，基于参考值的组合进行管理决策常常是不可能的)。

因此，在风险评估过程中，我们执行以下步骤：

1.定义项目的风险参考水平值。

2.建立每一组［r_i，l_i，x_i］与每一个参考水平值之间的关系。

3.预测一组临界点以定义项目终止区域，该区域由一条曲线或不确定区域所界定。

4.预测什么样的风险组合会影响参考水平值。

更详细的讨论参见专门探讨风险分析的论著(如文献［CHA89、ROW88］)。

6.5 风险缓解、监控和管理

这一步的所有风险分析活动都只有一个目的——辅助项目组建立处理风险的策略。一个有效的策略必须考虑三个问题：

- 风险避免。
- 风险监控。
- 风险管理及意外事件计划。

如果软件项目组对于风险采用主动的方法，则避免永远是最好的策略。这可以通过建立一个风险缓解计划来达到。例如，假设频繁的人员流动被标注为一个项目风险，r_0。基于以往的历史及管理经验，人员频繁流动的概率l_0被估算为0.7(70%，相当高)，而影响x_0被预测为对于项目成本及进度有严重的影响(见图6－1)。

为了缓解这个风险，项目管理必须建立一个策略来降低人员流动。可能采取的策略如下：

• 与现有人员一起探讨一下人员流动的原因(如，恶劣的工作条件，低报酬，竞争激烈的劳动力市场)。

• 在项目开始之前，采取行动以缓解那些在管理控制之下的原因。

• 一旦项目启动，假设会发生人员流动并采取一些技术以保证当人员离开时的工作连续性。

• 对项目组进行良好组织，使得每一个开发活动的信息能被广泛传播和交流。

• 定义文档的标准，并建立相应的机制，以确保文档能被及时建立。

• 对所有工作进行详细复审，使得不止一个人熟悉该项工作。

• 对于每一个关键的技术人员都指定一个后备人员。

随着项目的进展，风险监控活动开始进行了。项目管理者监控某些因素，这些因素可以提供风险是否正在变高或变低的指示。在人员频繁流动的例子中，应该监控下列因素：

• 项目组成员对于项目压力的一般态度。

• 项目组的凝聚力。

• 项目组成员彼此之间的关系。

• 与报酬和利益相关的潜在问题。

• 在公司内及公司外工作的可能性。

除了监控上述因素之外，项目管理者还应该监控风险缓解步骤的效力。例如，前述的一个风险缓解步骤中要求定义“文档的标准，并建立相应的机制，以确保文档能被及时建立”。如果有关键的人员离开了项目组，这是一个保证工作连续性的机制。项目管理者应该仔细地监控这些文档，以保证每一个文档内容正确，且当新员工加入该项目时，能为他们提供必要的信息。

风险管理及意外事件计划假设缓解工作已经失败，且风险变成了现实。继续前面的例子，假定项目正在进行之中，有一些人宣布将要离开。如果按照缓解策略行事，则有后备人员可用，因为信息已经文档化，有关知识已经在项目组中广泛进行了交流。此外，项目管理者还可以暂时重新将资源调整到那些人员充足的功能上去(并调整项目进度)，从而使得新加入人员能够“赶上进度”。同时，应该要求那些要离开的人员停止工作，并在最后几星期进入“知识交接模式”。这可能包括：基于视频的知识捕获，“注释文档”的建立和/或与仍留在项目组中的成员进行交流。

值得注意的是，RMMM 步骤将导致额外的项目开销。例如，花费时间去“备份”每一个关键的技术人员是需要花钱的。因此，风险管理的部分任务是评估何时由 RMMM 步骤所产生的效益低于实现它们所花费的成本。本质上是讲，项目计划者执行一个典型的成本-效益分析来估算项目的开销变化情况。如果对于频繁人员流动风险的缓解步骤将会增加 15%的项目成本及持续时间，而主要的成本因素是“备份后备人员”，则管理者可能决定不执行这一步骤。另一方面，如果风险缓解步骤仅增加 5%的成本及 3%的持续时间，则管理者极有可能将这一步骤付诸实现。

对于一个大型项目，可能标出 30 或 40 种风险。如果为每种风险定义三至七个风险管理步骤，则风险管理本身就可能变成一个“项目”！因此，我们将 Pareto 的 80—20 规则用于软件风险上。经验表明：整个软件风险的 80%(即，可能导致项目失败的 80%的潜在因素)能够由仅仅 20%的已标出风险来说明。早期风险分析步骤中所实现的工作能够帮助计划者确定哪些风险在所说的这 20%中。因此，一些已经标出、评估过及预测过的风险可能并不纳入 RMMM 计划之中——它们不属于那关键的 20%(具有最高项目优先级的风险)。

6.6 安全性风险和危险

风险并不仅限于软件项目本身。在软件已经被成功开发并交付给客户之后，仍有可能发生风险。这些风险一般与领域中的软件失败相关。

虽然一个良好的系统发生错误的概率很小，但在基于计算机的控制及监督系统中未被发现的错误可能会导致巨大的经济损失，或者更加严重，造成人员伤害或丧失生命。不过，基于计算机的控制及监督系统所产生的成本和功能效益常常超过这种风险。今天，计算机硬件及软件已经大量用于有规律地控制安全性很重要的系统。

当软件被用做控制系统的一部分时，复杂性会以数量级增加。由于人的错误所引起的微小的设计缺陷(这在基于硬件的传统控制系统中能够被发现并消除)当使用软件时会变得难以发现。

软件安全和危险分析是属于软件质量保证活动(第 8 章)，它主要是来标识和评估可能对软件产生负面影响并使整个系统失败的潜在危险。如果能够在软件工程的早期阶段标出危险，则可以指定软件设计特征来消除或控制潜在的危险。

6.7 RMMM 计划

风险管理策略可以包含在软件项目计划中，或者风险管理步骤也可以组织成一个独立的风险缓解、监控和管理计划(RMMM 计划)。RMMM 计划将所有风险分析工作文档化，并由项目管理者作为整个项目计划中的一部分来使用。RMMM 计划的大纲如下：

Ⅰ.引言

1.文档的范围和目的

2.主要风险综述

3.责任

a.管理者

b.技术人员

Ⅱ.项目风险表

1.中止线之上的所有风险的描述

2.影响概率及影响的因素

Ⅲ.风险缓解、监控和管理

n.风险＃n

a.缓解

i.一般策略

ii.缓解风险的特定步骤

b.监控

i.被监控的因素

ii.监控方法

c.管理

i.意外事件计划

ii.特殊的考虑

Ⅳ.RMMM 计划的迭代时间安排表

Ⅴ.总结

一旦建立了 RMMM 计划，且项目开始启动，则风险缓解及监控步骤也开始了。正如我们前面讨论过的，风险缓解是一种问题避免活动。风险监控则是一种项目

跟踪活动，它有三个主要目的：(1)评估一个被预测的风险是否真正发生了；(2)保证为风险而定义的缓解步骤被正确地实施；以及(3)收集能够用于未来风险分析的信息。在很多情况下，项目中发生的问题可以追溯到不止一个风险，风险监控的另一个任务就是试图在整个项目中确定“起源”(什么风险引起了什么问题)。

6.8 小结

当对软件项目期望很高时，一般都会进行风险分析。不过，即使他们进行这项工作，大多数软件项目管理者都是非正式地和表面上地完成它。花在标识、分析、和管理风险上的时间可以从多个方面得到回报：更加平稳的项目进展过程；较高的跟踪和控制项目的能力；由于在问题发生之前已经做了周密计划而产生的信心。

风险分析需要占用大量项目计划的工作量。标识、预测、评估、管理、及监控都要花费时间。但这是值得的。引用中国 2500 多年前的兵法家孙子的话：“知己知彼，百战不殆”。对于软件项目管理者而言，这个“彼”指的就是风险。

思考题

6.1 举出五个其他领域的例子来说明与被动风险策略相关的问题。

6.2 描述“已知风险”和“可预测风险”之间的差别。

6.3 在本章给出的所有风险检查表中的条目中增加三个额外的问题或主题。

6.4 你被要求建造一个软件以支持一个低成本的视频编辑系统。该系统将录像带作为输入设备，将视频信息存在磁盘上，并允许用户对已经数字化的视频信息进行各种编辑。对这类系统做一些调研，然后列出当你开始启动该项目以建造视频编辑系统时，你将面临的技术风险是什么。

6.5 你是一家大型软件公司的项目管理者。你被要求领导一个小组开发“下一代”字处理软件(见 3.3.2 节的简单描述)。为该项目建立一个风险表。

6.6 描述风险因素和风险驱动因子之间的不同。

6.7 为项目风险驱动因子建立一个加权方案。

6.8 为图 6－2 中的三个风险建立风险缓解策略及特定的风险缓解活动。

6.9 为图 6－2 中的三个风险建立风险监控策略及特定的风险监控活动。确认你已经标识出了你需要监控的因素，并确定风险发生的可能性是否在变高或变低。

6.10 为图 6—2 中的三个风险建立风险管理策略及特定的风险管理活动。

6.11 你能否想到一种情况：一个高概率、高影响的风险并不纳入 RMMM 计划的考虑之中？

6.12 参照图 6－4 所示的风险参考水平值，该曲线是否总是如图显示的那么匀称光滑，或者是否会有该曲线扭曲变形的情况存在？如果有，请举出一个例子。

6.13 做一些关于软件安全方面的研究，并写一份简单的报告。可以参考［LEV95］做为起点。

6.14 描述五个软件安全和危险分析是主要关心对象的软件应用领域。

第 7 章　项目进度安排及跟踪

在六十年代后期，一位热情的青年工程师[①]受命为一个自动制造应用软件项目“编写”计算机程序。选择他的原因非常简单，因为在整个技术小组中他是唯一参加过计算机编程培训班的人。这位工程师对汇编语言的IN和OUT指令以及Fortran语言略知一二，但是却根本不懂软件工程，更不用说项目进度安排和跟踪了。

他的老板给了他一大堆相关的手册，以及需要做些什么的口头描述。年轻人被告知该项目必须在两个月之内完成。

他阅读了这些手册，想好了解决方法，就开始编写代码。两周之后，老板将他叫到办公室询问项目进展情况。

“非常好，”工程师以年轻人的热情回答道，“这个项目远比我想象的简单。我差不多已经完成了 75％的任务。”

老板笑了，说道：“真是太棒了。”然后他嘱咐年轻人继续努力工作，准备好一周后再汇报一次工作进度。

一周之后老板将年轻人叫到办公室，问他说：“现在进度如何？”

“一切顺利，”年轻人回答说，“但是我遇到了一些小麻烦。我会排除这些困难，很快就可以回到正轨上来。”

“你觉得在最后期限之前能否完成？”老板问道。

“没有问题，”工程师答道。“我差不多已经完成了 90％。”

如果读者在软件领域中工作过几年，你一定可以将这个故事写完。毫不奇怪，青年工程师在整个项目工期内始终停留在90%的进度上，(在别人的帮助下)直到交付期限之后一个月才做完。

在过去的30年间，这样的故事被不同的软件开发者重复了成千上万次。我们不禁要问："为什么？"

7.1 基本概念

虽然软件延期交付的原因很多，但是大多数都可以追溯到下面列出的一个或多个根本原因上：

- 一个不现实的截止期限，由软件工程组以外的人所设立并强加给软件工程组内的管理者和项目开发者。
- 客户需求发生变化，而需求的变化没有能够反映在项目进度的变化上。
- 对工作量和/或完成该工作所需的资源数量估计不足。
- 在项目开始时，没有将可以预测的和/或不可预测的风险考虑在内。
- 事先无法预计的技术困难。
- 事先无法预计的人力困难。
- 由于项目组成员之间的交流不畅而导致的延期。
- 项目管理者未能发现进度拖后，也未能采取行动解决这一问题。

在软件行业中，人们对过于乐观的(即"不现实的")项目完成期限已经司空见惯。有时候设定截止时间的人认为这样的截止期限是合理的，但是常识告诉我们，合理与否还必须由完成工作的人来判断。

7.1.1 关于"延迟"的评注

拿破仑曾经说过："任何同意执行一个他本人都认为有缺点的计划的指挥官都应该受到指责；他必须提出自己的反对理由，坚持修改这一计划，最终甚至提出辞职而不是使自己的军队遭受惨败。"这句话掷地有声，值得软件项目管理者们深思。

在第5和第6章中讨论的估算和风险分析活动，以及本章中涉及的进度安排技术，通常都需要在一个定义好的截止期限的约束之下实现。如果最乐观的估算都表明截止期限是不现实的，一个胜任的项目管理者就应该"保护其队伍免受不

适当的进度安排的压力并将这种压力反映给施加压力的一方”［PAG85］。

不妨举例说明，假定一个软件开发小组的任务是构造一个医疗诊断仪器的实时控制器，该控制器需要在 9 个月之内推向市场。在进行了仔细的估算和风险分析之后，软件项目管理者得到的结论是在现有人员条件下，需要 14 个月的时间才能完成这一软件。这位项目管理者下一步该怎么办？

闯进客户的办公室(这里的客户非常可能是市场或销售人员)并要求修改交付日期似乎不太现实。外部市场压力决定了交付日期，届时必须发布产品。而(从事业前途的角度出发)拒绝这一项目同样是鲁莽的。那么应该怎么办呢？

在这种情况下，推荐以下的处理步骤：

1.使用从以前的项目中得到的数据，进行详细的估算。确定项目的估算工作量和持续时间。

2.使用增量过程模型(参见第 2 章)，制定一个软件开发策略，以能够在规定的交付日期提供关键功能，而将其他功能的实现推到以后。将这一计划做成文档。

3.与客户会谈并(用详细估算结果)来解释为什么规定的交付日期是不现实的，一定要指出所有这些估算都是基于以往的项目实践，而且一定要指出为了在目前规定的交付期限完成项目，与以往相比在工作效率上必须提高的百分比[①]。做如下解释将是恰如其分的：

“我认为在 XYZ 控制器软件的交付日期方面存在一个问题，我已经将一份以往项目中生产率的简化明细分类表和以多种不同方式进行的项目估算提交给各位，你会注意到我已经假设与以往的生产率相比有 20％的提高，但是我们仍然只能得到 14 个月而不是 9 个月的交付时间。”

4.将增量开发策略作为可选计划提交给客户。

“我们有几个选择，而我希望各位能够在这些选择的基础上做出决策。首先，我们可以增加预算，并引入额外的资源，以便使我们能够在 9 个月时间内完成这一工作。但是应该知道由于时间限制过于苛刻，这样做将会增加质量差的风险[②]。第二个方案是，去掉一部分需求中所列出的软件功能和特性。由此得到功能稍弱的产品的最初版本，但是我们可以对外宣布全部功能，并在总共 14 个月的时间内发布这些功能。第三种选择是不顾现实条件的约束，而希望项目能够在 9 个月时间内完成。结果是我们竭尽全力，但是却无法向用户提供任何功能。我想你们会同意，第三种选择是不可接受的。过去的历史和我们最乐观的估算都表明这是不现实的，是在选择一场灾难。”

尽管这样做会有些抱怨，但如果你给出了基于准确的历史数据的可靠估算，那么最终的谈判结果将可能是选择 1 或者选择 2。不现实的交付期限就不存在了。

7.1.2 基本原则

曾经有人请教著名的《神秘的人月》(The Mythical Man－Month，见文献[BRO95])一书的作者 Fred Brooks，“软件项目的进度是如何延迟的？”他的回答即简单又深刻：“一天一次。”

技术性项目(不论它是涉及到水力发电厂建设，还是开发一个操作系统)的现实情况是，在实现一个大目标之前必须完成数以百计的小任务。这些任务中有些是处于主流之外，其实现不会影响到整个项目的完成时间；其他任务则位于“关键路径”[①]之上，如果这些“关键”任务的进度拖后，则整个项目的完成日期就会受到威胁。

项目管理者的目标是定义所有项目任务，识别关键任务，然后跟踪关键任务的进展以保证“一天一次”的发现进度拖延情况。为了做到这一点，管理者必须建立一个具有一定详细程度的进度表，使得项目管理者能够监督进度，并控制整个项目。

软件项目进度安排是一种活动，它通过将工作量分配给特定的软件工程任务，而将所估算的工作量分布于计划好的项目持续时间内。但是，进度是随着时间的改变而不断演化的，注意到这一点至关重要。在项目计划的早期，首先建立一个宏观的进度安排表。该进度表标识所有主要的软件工程活动和这些活动影响到的产品功能。随着项目的进展，宏观进度表中的每个条目都被精化成一个“详细进度表”。于是(完成一个活动所必须实现的)特定软件任务被标识出来，并进行进度安排。

可以从两个不同的视角考察软件开发项目的进度安排。第一个视角，基于计算机的系统的最终发布日期已经确定(而且不能更改)。软件开发组织在这一约束下将工作量分布在预先确定的时间框架内。第二个视角，假定大致的时间界限已经讨论过，但是最终发布日期是由软件开发组设定的，工作量以一种能够最好地利用资源的方式加以分布，且在对软件进行仔细分析之后才定义最终发布日期。但不幸的是，第一种情况发生的频率远远高于第二种情况。

同软件工程的所有其他领域一样，有一些基本原则能够指导软件项目的进度安排：

划分：项目必须被划分成若干可以管理的活动和任务。为了实现项目的划分，对产品和过程都需要进行分解(参见第 3 章)。

相互依赖性：各个被划分的活动或任务之间的相互关系必须是确定的。有些任务必须顺序发生；而其他的则可以并发进行。有些活动只有在其他活动产生的工作产品完成时才能够开始，而其他的则可以独立进行。

时间分配：必须为每个被调度的任务分配一定数量的工作单位(例如，若干人天的工作量)。此外，必须为每个任务指定开始和结束日期，这些日期是与工

作完成的方式相互依赖的(全职还是兼职)是工作方式的函数。

工作量确认：每个项目都有预定数量的人员参与。在进行时间分配时，项目管理者必须确保在任意时段中分配给任务的人员数量不会超过项目组中的人员数量。例如，一个项目分配了 3 名员工参加(即，每天可分配的工作量为 3 人天[②])。在某一天中，需要完成 7 项并发的任务，每个任务需要 0.50 人天的工作量，在这种情况下，所分配的工作量就大于可用于分配的工作量。

定义责任：每个被调度的任务都应该指定某个特定的小组成员来负责。

定义结果：每个被调度的任务都应该有一个定义好的结果。对于软件项目而言，结果通常是一个工作产品(例如一个模块的设计)或某个工作产品的一部分。通常将多个工作产品组合成“可交付产品”。

定义里程碑：每个任务或任务组都应该与一个项目里程碑相关联。当一个或多个工作产品经过质量复审(参见第 8 章)并且得到认可时，标志着一个里程碑的完成。

随着项目进度的发展，上述每一条原则都会被用到。

7.2 人员与工作量之间的关系

在一个小型软件开发项目中，只需一个人就可以完成需求分析、设计、编码和测试。随着项目规模的增长，必然涉及到更多的人员参与(我们几乎不可能负担让一个人工作十年来完成 10 人年工作量的奢侈！)。

许多负责软件开发工作的管理者仍然坚信一个普遍存在的荒诞想法(参见第 1 章)：“即使进度拖后，我们也总可以增加更多的程序员，并在后期跟上进度”。不幸的是，在项目后期增加人手通常产生一种破坏性影响，其结果是使进度进一步拖延。后来增加的人员必须学习这一系统，而培训他们的人员正是过去一直在工作着的那些人，当他们进行教学时，就不能完成任何工作，而项目也进一步被拖延。

除去学习系统所需的时间之外，新加入人员将会增加人员之间通信的路径数量和整个项目中通信的复杂度。虽然通信交流对于一个成功的软件开发项目而言绝对是必不可少的，但是每增加一条新的通信路径就会增加额外的工作量，从而需要更多的时间。

7.2.1 一个例子

让我们来考虑一个有 4 名软件工程师的项目，在单独完成项目时，每个工程师的工作能力都是每年生产 5000 LOC，而当这 4 位工程师被编入一个项目组时，有 6 条可能的通信路径。每条路径都将花费原本可以用于软件开发的工作时间。

由于增加了通信成本，我们将假定每增加一条通信路径将会使项目组的生产率(以LOC计算)降低 250 LOC/年。因此项目组的生产率为 20000－(250×6)＝18500LOC/年，比期望数字低了 7.5%[①]。

上述项目组承担的为期一年的项目现在只剩下两个月时间，但是已经落后于进度，这时又加入了 2 名工程师。于是通信路径激增为 14，在交付之前剩下的两个月时间里，新增加成员的生产能力为 840×2＝1680 LOC。而目前的项目组生产率为 20000＋1680－(250×14)＝18180LOC/年。

尽管上述例子仅仅是对现实情况的粗略简化，但是它足以揭示另一个关键性的问题：参加软件项目的工作人员数量与整体生产率之间的关系不是线性的。

那么基于上述的人员-工作关系，是不是说项目组会降低生产效率呢？如果通信可以提高软件质量的话，答案断然是“不”。实际上，由软件工程小组进行的正式技术复审(参见第 8 章)将得到更好的分析和设计，更重要的是，这样可以降低直到测试阶段才能发现的错误数量(从而减小测试的工作量)。因此，如果以项目完成时间和客户满意程度来衡量，生产率和质量都将确实提高。

7.2.2 一个经验关系

请回忆一下在第 5 章介绍的“软件方程式”[PUT92]，我们可以看到在完成项目的时间与投入项目中的人员的工作量之间存在着高度非线性关系。交付的代码(源代码语句)行数 L 与工作量和开发时间之间的关系可以用下面的公式表示：

$$L = P \times (E/B)^{1/3} t^{4/3}$$

其中 E 是以人月为单位的开发工作量；P 是一个生产率参数，它反映了导致高质量软件工程工作的各种因素的综合效果(P 通常取值在 2，000 到 28，000 之间)；B 是特殊技术因子，在 0.16 到 0.39 之间取值，是所生产软件的规模的函数；t 是以月为单位的项目持续时间。

将上述软件方程式重排，可以得到关于开发工作量 E 的计算公式：

$$E = L^3/(P^3 t^4) \qquad (7.1)$$

其中 E 是在软件开发和维护的整个生命周期内所需的工作量(以人年计算)，t 是以年计算的开发时间。通过引入平均劳动力价格因素($/人年)，开发工作量的计算公式还能够与开发成本相关联。

这一方程式引出了一些有趣的结果。考虑一个复杂的实时软件项目，估计需要 33，000 LOC，12 个人年的工作量。如果为项目组分配 8 个人，项目大约可以用 1.3 年时间完成。但是如果将交付时间延长到 1.75 年，则公式(7.1)中的模型

所具有的高度非线性特性将导致如下结果：

$$E = L_3/(P^3t^4) \sim 3.8 \text{ 人年}$$

这意味着通过将截止时间推迟 6 个月，我们可以将项目组人数从 8 降低到 4 人！这一结果的有效性有待考证，但是其含意却十分清楚：通过在略为延长的时间内使用较少的人员，可以实现同样的目标。

7.2.3 工作量分布

在第五章中讨论的每种软件项目估算技术最终都归结为对完成软件开发所需人月(或者人年)的估算。一种推荐的在定义和开发阶段之间的工作量分布通常被称为“40-20-40 规则”[①]。40%或者更多的工作量分配给前端的分析和设计任务。类似比例的工作量用于后端测试。你可以推断出编码工作(大约 20%的工作量)没有被着重强调。

这种工作量分布只应被当作指导原则。各个项目的特点决定了其工作量分布。用于项目计划的工作量很少超过 2%到 3%，除非提交给组织的项目计划费用极大，而且具有高风险。需求分析大约占用 10%到 25%的工作量，用于分析或原型开发的工作量应该与项目规模和复杂度成正比的增长。通常有 20%到 25%的工作量用于软件设计，用于设计复审和随之而来的迭代开发也必须列入考虑之中。

因为在软件设计时投入了相当的工作量，随后的编码工作就变得相对简单。15%到 20%的工作量就可以完成这一工作。测试和随之而来的调试工作将占用 30%到 40%的软件开发工作量。软件的重要性决定了所需测试工作的份量，如果软件系统是人命相关的(即，软件错误可能使人丧命)，就应该考虑分配更高的测试工作量比例。

7.3 为软件项目定义任务集合

在第二章中，我们描述了多种不同的过程模型。这些模型为软件开发提供了不同的范型。如果不考虑一个软件项目组选择的是线性顺序范型、迭代开发模型、演化开发模型、并发开发模型还是组合使用这些模型，则过程模型都是由一个任务集合组成的，它使得软件项目组能够定义、开发和最终维护计算机软件。

没有一个普遍适用于所有软件项目的任务集合。适用于大型复杂系统的项目任务集合可能对于小型且相对简单的项目而言就过于复杂。因此一个有效的软件过程应该定义一组“任务集合”，各个任务集合的设计可以满足不同类型项目的要求。

一个任务集合包括一组软件工程工作任务、里程碑和交付产品，为了完成某一特定项目就必须完成这些任务。一个项目所选择的任务集合必须为最终获得高

质量的软件产品提供充分的规程要求，但同时又不能让项目组负担不必要的工作。

任务集合的设计应该可以应用于不同类型的项目和不同的严格度。尽管很难建立一个全面详尽的分类结构，但是大多数软件组织接手的项目一般属于下述类型：

概念开发项目：项目的目的是为了探索某些新的商业概念或者某种新技术的应用。

新应用开发项目：根据特定的客户需求而承担的项目。

应用增强项目：对现有软件进行最终用户可察觉的功能、性能或界面的修改。

应用维护项目：以一种最终用户不会立即察觉到的方式对现有软件进行纠错、适应或者扩展。

再工程项目：为了全部或部分重建一个现有(遗留)系统而承担的项目。

即使在单一的项目类型中，也会有许多因素影响任务集合的选择。当将这些因素综合考虑时，就会构成一个称为“严格度”的指示量，它将应用于所采用的软件过程中。

7.3.1 严格度

即使只考虑某种特定类型的项目，所采用的软件过程的严格度也会相当不同。严格度是众多项目特性的函数。例如，小型的非主要商业性质的项目的严格度一般可以小于大型复杂的主要业务应用程序。但是应该注意到，所有项目都必须以一种能够按时得到高质量的发布产品的方式来实施。可以定义如下的 4 种不同程度的严格度：

随意的：使用了所有过程框架活动(参见第 2 章)，但只需要一个最小的任务集合。一般情况下，将保护性任务最小化，并将文档需求降低。所有基本的软件工程原则仍然都是适用的。

结构化的：在项目中将使用过程框架。框架活动和适用于这种项目类型的相关任务，以及为保证高质量所需的保护性活动将得到应用。SQA、SCM、文档和度量任务将以一种经过优化的有效方式进行。

严格的：整个过程将按照一种能够确保高质量的严格规程要求应用于项目之中。所有保护性活动都将被采用，且要建立健壮的文档。

快速反应的：该项目将使用过程框架，但是由于某种紧急情况[①]的出现，只应用了那些为保持软件系统质量所必须完成的任务。在应用程序/产品交付给客

户之后再完成“回填工作”(即开发一套完整的文档，进行额外的复审)。

项目管理者必须开发一种系统化的方法用以选择适用于特定项目的严格度。为了做到这一点，需要定义项目适应性准则并计算“任务集合选择因子”的值。

7.3.2 定义适应性准则[①]

适应性准则用于确定一个项目中使用软件过程的严格度。我们为软件项目定义了以下 11 条适应性准则：

- 项目的规模。
- 潜在的用户数量。
- 任务的关键性。
- 应用程序的寿命。
- 需求的稳定性。
- 客户与开发者之间通信的容易程度。
- 应用技术的成熟度。
- 性能约束。
- 嵌入式/非嵌入式特性。
- 项目人员配置。
- 再工程因素。

每一条适应性准则都被赋予一定的等级分数，取值在 1 到 5 之间。其中 1 代表需要使用较小子集的过程任务，且整体的方法学及文档需求为最小的项目；而 5 则代表需要采用全部过程任务，且整体的方法学及文档需求相当高的项目。

7.3.3 计算任务集合选择因子的值

为一个项目选择适当的任务集合，应该按照下述几个步骤进行：

1.复审 7.3.2 节中给出的每个适应性准则，根据项目特点为它们赋予适当的等级分数(从 1 到 5)。这些等级分数应该输入到表 7－1 中。

2.复审赋予每个适应性准则的加权因子。加权因子的取值在 0.8 到 1.2 之间，

表明了对在当前环境中开发的软件类型而言特定适应性准则的相对重要性。如果需要，则可以进行修改，以反映当前的环境。

3.将表 7.1 中的等级分数与加权因子相乘，再乘以表示当前项目类型的条目点乘数。条目点乘数在 0 到 1 之间取值，表示该适应性准则与项目类型之间的相关程度。对应于各个适应性准则的相乘结果：

等级分数×加权因子×条目点乘数

分别放入表 7－1 的“乘积”栏中。

4.计算“乘积”栏中所有条目的平均值，并将结果放入标记着“任务集合选择因子(TSS)”的空格中。这个值将用于帮助你选择最适用于当前项目的任务集合。

表 7－1　计算任务集合选择因子

适应性准则	等级分数	权重	条目点乘数					乘积
			概念	新应用	增强	维护	再工程	
项目规模	—	1.20	0	1	1	1	1	—
用户数量	—	1.10	0	1	1	1	1	—
业务关键性	—	1.10	0	1	1	1	1	—
寿命	—	0.90	0	1	1	0	0	—

适应性准则	等级分数	权重	条目点乘数					乘积
			概念	新应用	增强	维护	再工程	
需求稳定性	—	1.20	0	1	1	1	1	—
易于通信	—	0.90	1	1	1	1	1	—
技术成熟度	—	0.90	1	1	0	0	1	—
性能约束	—	0.80	0	1	1	0	1	—
嵌入式/非嵌入式	—	1.20	1	1	1	0	1	—
项目人员配置	—	1.00	1	1	1	1	1	—
互操作	—	1.10	0	1	1	1	1	—
再工程因素	—	1.20	0	0	0	0	1	—
任务集合选择因子(TSS)							—	

7.3.4 解释 TSS 值并选择任务集合

一旦计算好任务集合选择因子，就可以使用下述的指南帮助你选择一个适用

于项目的任务集合：

任务集合选择因子取值	严格度
TSS＜1.2	随意的
1.0＜TSS＜3.0	结构化的
TSS＞2.4	严格的

两个推荐任务集合之间的 TSS 取值的重叠是有意设定的，这用于说明在进行任务集合的选择时，定义出精确的边界是不可能的。在进行最后的分析时，应该将任务集合选择因子的取值、以往的经验以及常识都作为项目任务集合的选择因素。

表 7－2 显示了在一个假想的项目中如何计算 TSS 的情况。项目管理者在“等级分数”一栏中选择等级分值，项目类型是“新应用开发”。因此其条目点乘数应该从“新应用”一栏中选择。“乘积”一栏中条目的计算使用如下公式

等级分数×加权因子×条目点乘数

TSS 的取值(“乘积”一栏中所有条目的平均值)是 2.8。使用上述的准则，管理者可以选择使用结构化的或严格的任务集合。在考虑了所有项目因素之后，可以作出最终决策。

表 7－2　计算任务集合选择子：一个例子

适应性准则	等级分数	权重	条目点乘数					乘积
			概念	新应用	增强	维护	再工程	
项目规模	2	1.20	—	1	—	—	—	2.4
用户数量	3	1.10	—	1	—	—	—	3.3
业务关键性	4	1.10	—	1	—	—	—	4.4
寿命	3	0.9	—	1	—	—	—	2.7
需求稳定性	2	1.2	—	1	—	—	—	2.4
易于通信	2	0.9	—	1	—	—	—	1.8
技术成熟度	2	0.9	—	1	—	—	—	1.8
性能约束	3	0.8	—	1	—	—	—	2.4
嵌入式/非嵌入式	3	1.2	—	1	—	—	—	3.6
项目人员配置	2	1.0	—	1	—	—	—	2.0
互操作	4	1.1	—	1	—	—	—	4.4
再工程因素	0	1.2	—	0	—	—	—	2.8
任务集合选择因子(TSS)								2.8

7.4 选择软件工程任务

为了制定项目进度安排，必须将任务集合分布在项目时间表上。如 7.3 节所述，任务集合将根据项目类型和严格度的不同而有所不同。7.3 节中的每种项目类型都可以使用线性顺序、迭代(如，原型模型)或者演化(如，螺旋模型)等过程模型来实现。在某些情况下，项目类型可以从一种形式平滑的转换为另一种形式。例如，成功的概念开发项目通常会演化成为新应用开发项目，而新应用开发项目结束之后，可能又开始了一个应用增强项目。这种进程是自然且可预测的，而不论是采用何种过程模型来组织。作为一个例子，下面我们将考虑概念开发项目的主要软件工程任务。

概念开发项目是在必须探索某些新技术是否可行时发起的。这种技术是否可行尚不可知，但是某个客户(如营销部门)相信其潜在利益的存在。概念开发项目的完成需要应用下面所述的主要任务：

确定概念范围：确定项目的整体范围。

初步的概念计划：确定组织的能力，以承担项目范围所规定的工作。

技术风险评估：评估与将要作为项目范围的一部分而被实现的技术相关联的风险。

概念证明：阐明新技术在软件环境中的生命力。

概念实现：以一种可以由客户方复审的方式实现概念的表示，且当概念必须被卖给其他客户或管理者时能够用于“推销”的目的。

客户对概念的反应：向客户索取对新技术概念的反馈，并以特定的客户应用作为目标。

快速浏览上述主要任务，你应该不会有何诧异。实际上概念开发项目的软件工程任务流程(以及其他所有类型的项目)与人们的常识相差无几。

软件开发组必须知道哪些任务是必须完成的(项目范围)；项目组(或管理者)必须确定目前是否有人能够进行这一工作(计划)；考虑与工作相关联的风险(风险评估)；以某种方式证明这种技术(概念的证明)；并且将这种技术用原型的方式实现出来，从而使客户能够对其进行评价(概念实现和客户评价)。最后，如果这一概念是可行的，则必须将其产品版本开发出来(技术转化)。

有非常重要的一点需要注意，概念开发任务本质上是迭代的。这就是说，一个实际的概念开发项目也许会通过多个有计划的增量步骤得以实现，每个步骤都被设计成能够产生一个可供客户评价的可发布版本。

如果选择使用线性过程模型，则每一个增量都被定义为如图 7－1 所示的一

个重复序列。在每个序列中，将使用保护性活动(参见第 2 章中的描述)；监控软件质量，且在每个序列的末尾，将产生一个可交付产品。随着逐次迭代的过程，可交付产品应该向着在概念开发阶段已经定义的最终产品收敛。如果选择的是演化开发模型，则从任务 1.1 到 1.6 的分布应该如图 7－2 中所示。可以以类似的方式定义和使用关于其他项目类型的主要软件工程任务。

7.5 主要任务的求精

在 7.4 节中所描述的主要任务可以被用于定义项目的宏观进度表。例如，概念开发项目中描述的任务可以用于定义该项目的任务网络(参见 7.6 节)。

我们在本章前面曾经指出，必须将宏观进度表精化来创建一个详细的项目进度表。精化工作始于将每个主要任务分解为一组子任务(以及相关的工作产品和里程碑)。

作为任务分解的例子，我们考虑在 7.4 节中讨论的“确定概念范围”任务。任务精化可以使用大纲格式完成，但是在本书中将使用一种过程设计语言[①]来阐明“确定概念范围”这一活动的流程：

任务 I.1　确定概念范围

I.1.1 标出需求、效益和潜在的客户；

I.1.2 定义所希望的输出/控制和驱动应用程序的输入事件；

Begin 任务 I.1.2

I.1.2.1FTR：[①]复审需求的书面说明

I.1.2.2 导出一个客户可见的输出/输入列表

case of：mechanics

mechanics＝质量功能配置

与客户会谈，以划分主要的概念需求；

会见最终用户；

观察现有的解决问题的方法和当前的问题解决过程；

复审以往的要求和抱怨；

mechanics＝结构化分析

列出主要数据对象；

定义对象之间的关系；

定义对象的属性；

mechanics＝对象视图

列出问题类；

确定类层次和类连接；

定义各个类的属性；

endcase

I.1.2.3FTR：与客户一起复审输出/输入，并在需要时进行修改；

endtask 任务 I.1.2

I.1.3 为每项主要功能定义操作/行为；

Begin 任务 I.1.3

I.1.3.1 FTR：复审在任务 I.1.2 中得到的输出和输入数据对象；

I.1.3.2 导出功能/行为模型；

case of：mechanics

mechanics＝质量功能配置

与客户会谈，以复审主要的概念需求；

会见最终用户；

观察现有的解决问题的方法和当前的问题解决过程；

建立一个功能/行为的层次结构概要；

mechanics＝结构化分析

导出一个系统级别的数据流程图；

精化数据流程图，以便给出更多的细节；

为最底层精化图的功能书写处理过程说明；

mechanics＝对象视图

定义与各个类相关的操作/方法；

endcase

I.1.3.3 与客户一起复审功能/行为模型，并在需要时进行修改；

endtask 任务 I.1.3

I.1.4 把需要在软件中得到实现的技术要素分离出来；

I.1.5 研究现有软件的可用性；

I.1.6 定义技术可行性；

I.1.7 对系统规模进行快速估算；

I.1.8 创建一个“范围定义”；

endtask 任务 I.1

任务和用过程设计语言进行精化时标注的子任务共同构成了制定“确定概念范围”这一活动的详细进度表的基础。

7.6 定义任务网络

任务和子任务之间基于其间顺序而存在相互依赖关系。而且当有多个人参与软件工程项目时，多个开发活动和任务并行进行的可能性很大。在这种情况下，必须协调多个并发任务，以保证它们能够在后继任务需要其工作成果之前完成。

“任务网络”是一个项目的任务流程的图形表示。该网络有时被用作在自动项目进度安排工具中输入任务序列和依赖关系的机制。任务网络的最简单形式(当创建宏观进度表时使用)刻划了软件工程主要任务。图 7－3 显示了一个概念开发项目的任务网络示意图。

软件工程活动的并发本质导致了若干重要的调度需求。由于并行任务是异步发生的，计划者必须确定任务之间的依赖关系，以保证项目朝着最终完成的方向持续发展。另外，项目管理者应该注意那些位于关键路径之上的任务，也就是说，为了保证整个项目的如期完成，就必须保证这些任务能够如期完成。在本章的后面部分，我们将详细讨论这些问题。

图 7－3 中所示的任务网络是宏观的，注意到这一点非常重要。在一个详细

的任务网络(详细进度表的前驱)中，应该对图 7－3 所示的各个活动加以扩展。例如，应该扩展任务 I.1，以显示 7.5 节所述的精化中的所有详细任务。

7.7 进度安排

软件项目的进度安排与任何其他多任务工程工作的进度安排几乎没有太多的差别。因此，通用的项目进度安排工具和技术不必做太多修改就可以应用于软件项目。

“程序评估和复审技术(PERT)”和“关键路径方法(CPM)”[MOD83]就是两种可以用于软件开发的项目进度安排方法。两种技术都是由较早的项目计划活动中已经产生的信息来驱动的，这些信息包括：

- 工作量的估算。
- 产品功能的分解。
- 适当的过程模型的选择。
- 项目类型和任务集合的选择。

任务之间的依赖关系可以使用任务网络来定义。任务，有时也称作项目的“工作细分结构”(WBS)，其定义可以是针对整个产品，也可以是针对单个功能进行的。

PERT 和 CPM 两种方法都提供项目工作定量划分的工具，能支持软件计划者(1)确定关键路径一决定项目持续时间的任务链；(2)通过使用统计模型为单个任务建立最有可能的时间估算；(3)计算为特定任务定义其时间“窗口”的边界时间。

边界时间的计算对软件项目进度的安排十分有用。比如说，某个功能的设计的推迟可能进一步导致其他功能开发的拖后。Riggs[RIG81]描述了一些能够从 PERT 或 CPM 网络中得到的重　　谋呓缡奔测海*1)某个任务的最早开始时间是当其所有前驱任务在最短的可能时间中完成时；(2)某个任务的最晚开始时间是在不延迟项目最小完成时间的前提下，最晚启动该任务的时间；(3)最早结束时间——最早开始时间加上任务持续时间；(4)最晚结束时间——最晚开始时间加上任务持续时间；以及(5)总浮动量——在保证进度的前提下，调度任务时所允许的富余时间或回旋时间的总和。通过边界时间的计算可以确定关键路径，并为管理者提供当任务完成时评估进展的量化方法。

PERT 和 CPM 方法都已经在多种自动工具中得到实现，这些工具在几乎所有个人电脑上都可以找到[THE93]。这类工具易于使用，且使得每一位软件项目管理者都可以使用前述的进度安排方法。

7.7.1 时间表

在创建软件项目进度表时，计划者将从一组任务(工作细分结构)入手。如果使用自动工具，就可以用任务网络或者任务大纲的方式输入工作细分结构。然后为每一项任务输入工作量、持续时间和开始时间。此外，每一项任务都必须被分配给特定的人员。

上述输入的结果之一是产生“时间表(Timeline Chart)”，也叫做“甘特图(Gantt Chart)”。可以为整个项目建立一个时间表，也可以为各个项目功能或各个项目参与者分别开发各自的时间表。

表 7－3 给出了时间表的格式，该图描述了一个新的字处理软件产品中“确定概念范围”任务(参见 7.5 节)的软件项目进度安排。所有的项目任务(对于“确定概念范围”)都在左边的栏中列出。水平条表示每个任务的持续时间。当日历中同一时段中存在多个水平条时，就代表任务之间存在并发。图中的菱形表示里程碑。

一旦输入了生成时间表所需的信息，大多数软件项目进度安排工具都会生成“项目表”——一个表格式的列表，列出所有项目任务、其计划的及实际的开始与结束日期、以及各种相关信息(参见图 7－5)。项目表与时间表一同使用，使得项目管理者能够跟踪项目的进展情况。

7.7.2 跟踪进度

项目进度为软件项目管理者提供了一张进度路线图。如果被正确制定，项目进度表中应该

定义在项目进展过程中必须被跟踪和控制的任务及里程碑。项目跟踪可以通过以下方式得以实现：

- 定期举行项目状态会议，由项目组中的各个成员分别报告进度和问题。
- 评估所有在软件工程过程中所进行的复审的结果。
- 确定正式的项目里程碑(表 7-3 中的菱形)是否在预定日期内完成。
- 比较项目表(表 7-4)中列出的各项任务的实际开始日期与计划开始日期。
- 与开发者进行非正式会谈，获取他们对项目进展及可能出现的问题的客观评估。

实际上，有经验的项目管理者都会使用所有这些跟踪技术。

软件项目管理者使用“控制”的方法来管理项目资源、处理问题和指导项目

参与者。如果一切顺利(即，项目在预算范围内按进度进行，复审结果表明的确取得了实际进展，达到了各个里程碑)，则几乎不必施加控制。但是如果出现问题，项目管理者就必须施加控制，以便尽快解决问题。当问题得到诊断之后[①]，可能需要增加额外的资源以解决问题：如可能需要雇佣新员工，或者需要重新定义项目进度。

表 7—4　一个项目表的例子

	工作任务	计划开始	实际开始	计划结束	实际结束	人员分配	工作量分配	附注
I.1.1	标出需求和效益							
	会见客户	wk1,d1	wk1,d1	wk1,d2	wk1,d2	BLS	2p-d	
	标出需求和项目约束	wk1,d2	wk1,d2	wk1,d2	wk1,d2	JPP	1p-d	
	建立产品说明	wk1,d3	wk1,d3	wk1,d3	wk1,d3	BLS/JPP	1p-d	
	里程碑：定义产品说明	wk1,d3	wk1,d3	wk1,d3	wk1,d3			
I.1.2	定义希望的输出/控制							
	输入(OCI)							
	确定键盘功能	wk1,d4	wk1,d4	wk2,d2		BLS	1.5p-d	
	确定语音输入功能	wk1,d3	wk1,d3	wk2,d2		JPP	2p-d	
	确定交互模式	wk2,d1		wk2,d3		MLL	1p-d	
	确定文档诊断	wk2,d1		wk2,d2		BLS	1.5p-d	
	确定其他 WP 功能	wk1,d4	wk1,d4	wk2,d3		JPP	2p-d	
	做 OCI 文档	wk2,d1		wk2,d3		MLL	3p-d	
	FTR：与客户复审 OCI	wk2,d3		wk2,d3		all	3p-d	
	按要求修改 OCI	wk2,d4		wk2,d4		all	3p-d	
	里程碑：定义 OCI	wk2,d5		wk2,d5				
I.1.3	定义功能	行为						
	…	…	…	…	…	…	…	

在面对交付期限的巨大压力时，有经验的项目管理者有时会使用一种称为“时间盒”[ZAH95]的项目进度安排和控制技术。时间盒策略认识到完整的产品可能难以在预定时间内交付，因此，应该选择增量软件开发范型，并为每个增量的交付定义各自的进度。

接着，对与每个增量相关的任务实行时间盒技术。这意味着通过对增量的交付日期倒退着推算，来调整每项任务的进度。在每项任务前后加上了一个“盒子”，当任务触及其时间盒边界时(正负 10%的范围内)，则该项任务停止，下一任务开始。

对于时间盒方法的第一个反应通常是反面的：“如果工作尚未完成，我们该如何继续？”答案就隐藏在工作的完成方式中。在遇到时间盒的边界时，很可能任务的 90%已经完成[②]。余下的 10%工作尽管重要，但是可以(1)被推迟到下一个增量中，或者(2)在以后需要时再完成。项目朝着交付日期逐步进展，而不是被某个任务“卡住”。

7.8 项目计划

软件工程过程中的每一步骤都应该产生可以被复审并能够作为后续步骤的基础的工作产品。软件项目计划在计划任务结束时产生，它给出了基线的成本及进度信息，这些信息将会在整个软件工程过程中被使用。

软件项目计划是一种面向广大读者的相对简短的文档。它必须能够(1)在软件管理者、技术人员和客户之间传达项目范围和资源信息；(2)定义风险并提出有关风险管理技术的建议；(3)定义管理复审的成本和进度；(4)为与项目相关的所有人员提供软件开发的整体方法；(5)概述如何保证质量及变化的管理。表 7-5 给出了一个软件项目计划的大纲。

针对不同的读者，所显示的成本和进度可能有所不同。如果计划只作为内部文档使用，则应该给出各种成本估算技术所得到的结果。当计划在组织以外发布时，应该给出一份经过整合的成本细分表(结合了所有成本估算技术的结果)。类似的，进度信息的详细程度也应该随读者和计划正式程度的不同而有所不同。

软件项目计划不必过于冗长复杂，其目的是帮助确立软件开发工作的有效性。该计划集中于“做什么”的一般性说明和特定的“多少资源”与“多长时间”的说明。软件工程过程中的后续步骤将集中来完成项目的定义、开发和维护。

表 7－5　软件项目计划

Ⅰ.引言	3.风险管理(意外事件计划)
A.计划的目的	Ⅳ.进度
B.项目范围和目标	A.项目工作细分结构
1.范围说明	B.任务网络
2.主要功能	C.时间表(甘特图)
3.性能问题	D.资源表
4.管理及技术约束	Ⅴ.项目资源
Ⅱ.项目估算	A.人员
A.用于估算的历史数据	B.硬件和软件资源
B.估算技术	C.特殊资源
C.工作量、成本和持续时间的估算	Ⅵ.人员组织
Ⅲ.风险管理策略	A.项目组结构(如果需要)
A.风险表	B.管理报告
B.对需管理的风险的讨论	Ⅶ.跟踪及控制机制
C.每个风险的 RMMM 计划	A.质量保证和控制
1.风险缓解	B.变化管理和控制
2.风险监控	Ⅷ.附录

7.9 小结

进度安排是计划活动的首要任务，而计划活动则是软件项目管理的首要组成部分。与估算方法和风险分析相结合，进度安排将为项目管理者建立起一张行路图。

进度安排始于过程的分解。项目的特性被用于为需要完成的工作选择一个适当的任务集合。任务网络刻划了各项软件工程任务、各项任务与其他任务之间的依赖关系、以及任务的持续时间。任务网络可以用于计算项目的关键路径、时间表和各种项目信息。用进度表作为指导，项目管理者可以跟踪和控制软件工程过程中的每一个步骤。

思考题

7.1“不合理的”项目截止日期是软件行业中存在的现实情况。当你遇到这种情况时应该如何处理？

7.2宏观进度表和详细进度表之间的区别是什么？是否有可能只依据所制定的宏观进度表来管理一个项目？为什么？

7.3是否存在这种情况：一个软件项目里程碑没有与某个复审相连？如果有，请给出一个或更多个例子。

7.4 在 7.2.1 节中，我们给出了在一个多人参与的软件项目中可能出现的“通信成本”的例子。请给出一个反例，说明一群精通良好的软件工程实践并使用正式技术复审的工程师能如何提高项目组的生产率(当与个人生产率的总和相比时)。［提示：可以假定复审能够减少返工的工作量，而返工将占用个人工作时间的 20％到 40％。］

7.5 尽管为延迟的软件项目增加人手可能会进一步拖延工期，但是否在某些情况下并不如此呢？请加以描述。

7.6 在人员和时间之间的关系是高度非线性的。使用 Putnam 的软件公式(见 7.2.2 节)，编制一个表，反映软件项目中人员数量与项目持续时间之间的关系，该项目需要 50，000 LOC 和 15 人年的工作量(生产率参数为 5000，且 B=0.37)。假定该软件必须在 24 个月和加减 12 个月的时间期限内交付。

7.7 假定你要为一所大学开发一个联机课程登记系统(OLCRS)。首先请扮演客户的角色(如果你是一名学生就很简单了！)，并指出一个好系统应该具有的特性。或者你的老师会为你提供一组初步的系统需求。

使用第五章所介绍的 COCOMO 模型，估算 OLCRS 系统的开发工作量和持续时

间。建议你如下进行：

定义 OLCRS 项目中的并行工作活动

将工作量分布到整个项目中

建立项目里程碑

7.8 以 7.3 节为指导，计算 OLCRS 的 TSS。确证你考虑到了所有的工作。选择一种项目类型并由此为该项目导出一个适当的任务集合。

7.9 为 OLCRS 或者为其他感兴趣的软件项目定义任务网络。确证你已将任务和里程碑显示出来，并为各项任务附上了估算的工作量和持续时间。如果可能的话，使用一个自动进度安排工具来完成这一工作。

7.10 如果可以使用自动进度安排工具，请为问题 7.8 中定义的任务网络确定关键路径。

7.11 使用进度安排工具(如果有条件)或者纸笔(如果需要)，制定 OLCRS 项目的时间表。

7.12 依照 7.5 一节中对“确定概念范围”任务进行精化的方式，对“计划软件项目”这一任务进行精化。

7.13 建议一些实用的方法，使得项目管理者能够监控项目成本和进度是否与项目计划相符合。

① 这个故事是我的自传。

① 如果提高的百分比为 10%到 25%，则实际上是有可能如期完成这一项目的。但更有可能的是，所需的队伍效率的提高百分比要高于 50%。这是不切实际的期望。

② 你还可以补充说，增加更多的人手不会成比例的缩短完成项目所需的时间。

① 在本章中后面将详细讨论“关键路径”。

② 实际上由于与工作无关的会议、病假、休假和各种其他原因，可供分配的工作量要少于 3 人天。但是在本书中，我们将假定员工时间是 100%可用的。

① 也可以做出以下辩驳：如果通信是高效的，就可以提高所进行的工作的质量，从而降低返工工作量，并提高每个项目组成员的生产率。辩驳成立！

① 现在对于大型软件开发项目而言，通常多于 40%的全部项目工作量用于分析和设计任务。因此从严格意义上讲，“40-20-40”，的名称已经不再适用。

① 紧急情况应该非常罕见（在软件工程的讨论范围内，发生这种情况的工作不应该超过 10%到 20%）。紧急情况与项目工期紧张是不同的。

① 出现在本节中的适应性准则与后面一节出现的TSS计算都是根据《自适应的过程模型》改编的。其复制得到了R.S.Pressman&Associates，Inc.的许可。欲知有关详情，请发电子邮件给info@rspa.com.

① 过程设计语言与 14 章中讨论的程序设计语言在语法上是类似的。

① FTR表示在此需要进行一次正式的技术复审（参见第 8 章）

① 千万注意：进度延迟是某种潜在问题的症状。项目管理者的角色是论断出潜在的问题，并采取行动改正之。

② 爱挖苦人的人也许会想起一句谚语：“完成系统的前 90％需要 90%的时间，完成剩下的 10%也要用 90%的时间”。

第 8 章　软件质量保证

本书中所描述的软件工程方法的唯一目标是：生产出高质量的软件。但是许多读者会遇到这种难题的挑战：“什么是软件质量？”

Philip Crosby ［CRP79］在他关于质量的划时代著作中为上述问题提供了一个谎谬的答案：

质量管理的问题不在于人们不知道什么是质量，问题在于人们认为他们自己知道什么…

在这一点上，质量与性共性颇多。每个人都需要它(当然，是在某种条件之下)。每个人都觉得自己理解它(尽管人们不愿意解释它)。每个人都认为实行它只需遵从自然的趋势(毕竟我们不管怎样都还做得不错)。当然，大多数人认为这一领域的问题都是由他人引起的(假设只要他们花了时间就能把事情做好)。

有些软件开发者仍然相信软件质量是在编码之后才应该开始担心的事情。这太谎谬了！“软件质量保证”(SQA)是一种应用于整个软件过程的保护性活动(参见第 2 章)。SQA 包括：(1)一种质量管理方法，(2)有效的软件工程技术(方法和工具)，(3)在整个软件过程中采用的正式技术复审，(4)一种多层次的测试策略，(5)对软件文档及其修改的控制、(6)保证软件遵从软件开发标准的规程(在适用时)，以及(7)度量和报告机制。

在本章中，我们将集中讨论与软件组织“在正确的时间、以正确的方式、做正确的事情”相关的管理问题和特定过程活动。有关质量的量化讨论将在第 18 章中给出。

8.1 质量概念 ①

据说没有两片雪花是完全相同的。当然在我们望着雪花飘落时，很难想象雪花的不同，更不会想到每一片雪花都具有独一无二的结构。为了观察雪花之间的差异，我们必须非常仔细地检查各个标本，也许要用放大镜才行。实际上观察得越仔细，发现的不同之处就越多。

这种现象称为“样本间差异”，不但适用于自然界的万物，而且适用于人类的一切产品。例如，如果非常仔细的观察两个“相同”的电路板，就可以观察到电路板上的铜线路在几何形状、位置和厚度上都有所不同，而且电路板上钻孔的位置和直径也各不相同。

所有的工程和制造产品都会表现出差异。也许不借助于精密仪器对几何特征、电路特征、或者其他零件属性进行测量，不同样本之间的差异将非常不明显。但是在足够精密的仪器帮助下，我们就会得到这样的结论：没有任何物品的两个样本是完全一样的。

这一物理世界中的规律同样适用于软件吗？想象一个程序，在它执行过程中的某一点上需要按照某一关键字的升序对若干记录进行排序。这些记录的性质并不重要，有可能是雇员记录、客户数据库、实时航空控制系统的地图坐标、或者随便什么。

编写这一排序例程(或者从一个可复用构件库中选择例程)的程序员决定使用“快速排序”来解决这一问题。最终产品的观察者能否将这一软件与一个除了使用(比如说)“冒泡排序”之外与前者完全相同的产品区别开来呢？也许可以，但是可能会需要更多信息，甚至精密的工具来区分两个系统的不同。

差异控制是质量控制的核心。制造商希望尽可能减小生产的产品之间的差异，即使进行复制软盘这样相对简单的工作也不例外。我们希望尽可能缩小任何一对所谓完全相同的磁盘之间的差异。当然这并不成问题——软盘复制仅仅是一项微不足道的制造操作，我们可以保证总是能够创建完全相同的软件副本。

但是，我们真的能够做到吗？我们需要保证软盘上的磁道具有某一特定的耐久性，以保证绝大多数的软驱能够正确读出这些软盘。而且必须保证区分 0 和 1 的磁通量足够大，使得读写头能够正确探测 0 与 1。软盘复制机可能的确会磨损和超出忍耐范围。因此，象软盘复制这样“简单”的过程也会遇到样本差异问题。

那么软件开发组织控制差异的需要可能是怎样的呢？对于每个不同的项目，我们希望尽可能减小完成项目预计需要的资源和实际使用的资源之间的差异，包括人员、设备和时间。一般来说，我们希望测试程序能够覆盖软件的不同发布版本之间的某个已知百分比。我们不仅希望尽可能缩小发布产品中的缺陷数量，而且要保证不同版本之间的错误数量差异也保持最小。(如果产品的第三个发布版本中错误数量十倍于此前的版本，我们的客户将会感到失望。)我们希望自己的热线技术支持在解决不同客户的问题时，速度和准确程度差异尽可能减小。这样的举列可以无穷无尽。

8.1.1 质量

American Heritage Dictionary(《美国传统字典》)中对质量的定义是：“某一事物的特征或属性”。作为一个事物的属性，质量指的是可以度量的特征——那些可以与已知标准进行比较的东西，如长度、颜色、电的性质、可延展性等等。但是软件，很大程度上是一种知识实体，其特征的定义远比物理对象要困难得多。

然而，程序特征的度量的确存在。这样的属性包括循环复杂度、内聚力、功能点、代码行数和其他许多在第 18 章和第 23 章中讨论的属性。在根据对象的可度量特征考察一个对象时，可以有以下两种不同的质量：设计质量和符合质量。

设计质量：是指设计者为一件产品规定的特征。材料等级、耐久性、及性能的规约都属于设计质量。当规定使用更高级别的材料、要求达到更强的耐久性和更高层次的性能时，如果产品能够依照规约进行制造，则产品的设计质量便会提高。

符合质量：是指在制造过程中符合设计规格的程度。同样，符合程度越高，符合质量也就越高。

在软件开发时，设计质量包括系统的需求、规约和设计。符合质量则主要关注实现问题。如果实现了符合设计、得到的系统满足系统需求和性能目标，则符合质量较高。

8.1.2 质量控制

差异控制可以等同于质量控制。但我们如何实现质量控制呢？“质量控制”是为了保证每一件工作产品都满足对它的需求而应用于整个开发周期中的一系列审查、复审和测试。质量控制在创建工作产品的过程中包括一个反馈循环。度量和反馈相结合，使得我们能够在得到的工作产品不能满足其规约时调整开发过程。这种方法将质量控制视为整个制造过程的一部分。

质量控制活动可以是全自动的、全人工的，也可以是自动工具与人员交互的结合。质量控制中的关键概念之一是所有工作产品都具有定义好的和可度量的规约，我们可以将每个过程的产品与这一规约进行比较。反馈循环的引入对于最小化产生的缺陷至关重要。

8.1.3 质量保证

“质量保证”由管理层的审计和报告功能构成。质量保证的目标是为管理层提供为获知产品质量信息所需的数据，从而获得产品质量是否符合预定目标的认识和信心。当然如果质量保证所提供的数据发现了问题，则管理层负责解决这一问题，并为解决质量问题分配所需的资源。

8.1.4 质量的成本

质量成本包括所有由质量工作或者进行与质量有关的活动所导致的成本。质量成本研究的开展能够为当前质量成本设定基线，标识降低质量成本的机会，并提供一种规范化的比较基础。规范化的基础几乎全都以“元”(钱)计算。一旦我们将质量成本以“元”为单位进行了规范化，我们就拥有了必要的数据来评估能够在何处改进现有过程。而且，还可以进一步评估那些基于“元”的项在改变时所产生的影响。

质量成本可以被划分为与预防、鉴定及失败相关的成本。“预防成本”包括：

• 质量计划。

• 正式技术复审。

• 测试设备。

• 培训。

“鉴定成本”包括为深入了解“首次通过”各个过程时产品的状态而开展的那些活动。鉴定成本的例子如下：

• 过程内和过程间审查。

• 设备校准和维护。

• 测试。

“失败成本”是指如果在将产品交付给客户之前已经消除了缺陷时就不会存在的成本。失败成本可以进一步划分为内部失败成本和外部失败成本。“内部失败成本”是指在产品交付之前发现错误而引发的成本。内部失败成本包括：

• 返工。

• 修复。

• 失败模式分析。

“外部失败成本”是指与产品交付给客户之后所发现的缺陷相关的成本。外部失败成本的例子如下：

• 解决客户的抱怨。

• 退换产品。

• 求助电话支持。

• 保修工作。

正如我们所预料的，发现和修改一个缺陷的成本将随着我们从预防到检测、从内部失败到外部失败工作的开展而急剧增加。图 8-1，根据 Boehm [BOE81] 所收集的数据，阐述了这一现较蟆*

Kaplan 及其同事[KAP95]报告了更多近期的数据，该报告以 IBM 的 Rochester 开发部门的工作为基础：

审查 200000 行代码总共用了 7053 个小时，结果是预防了 3112 个潜在的缺陷。假定雇佣一名程序员的成本为每小时 40 美元，预防 3112 个缺陷的总成本为 282120 美元，约为每个缺陷 91 美元。

下面将这些数字与产品交付给客户之后消除缺陷的成本加以比较。假定没有进行代码审查，但是程序员编码格外小心，而在交付的产品中每 1000 行代码中只有 1 个缺陷漏网［大大优于产业界的平均水平］。这意味着在客户的操作环境中仍然有 200 个缺陷需要改正。估算改正每个缺陷的成本为 25000 美元，则总成本将高达 5 百万美元，大约比进行缺陷预防的产品的总成本高出 18 倍。

当然，IBM 生产的软件被数以万计的客户所使用，因此，有可能交付后改正软件缺陷的成本要高于平均水平，但这并不能否定上面所说的结果。即使普通软件组织的缺陷改正成本仅仅为 IBM 的 25％(大多数组织并不知道这项成本究竟是多少！)，但用于质量控制和保证活动而节约的成本仍然令人叹服。

8.2 质量运动

今天，整个工业界中的公司资深管理者都认识到产品的高质量将导致成本下降和底线的提高。但是在以前，情况并不如此。质量运动始于本世纪 40 年代 W.Edwards Deming［DEM86］的开创性工作，第一次真正的实验则是在日本进行的。以Deming的想法为基础，日本人开发了一种系统化的方法来从根本上消除造成产品缺陷的原因。从 70 年代到 80 代，他们的工作被移植到西方，有时被称作“全面质量管理(TQM)”[①]。尽管不同公司和不同作者那里的术语略有不同，但通常采用的都是 4 个步骤的过程，且该过程构成了任何一个好的TQM项目的基础。

第一步被称为“kaizen”，是指一个连续的过程改进系统。Kaizen 的目标是开发一个看的见的、可重复的和可度量的过程(在这里是指软件过程)。

第二步被称为“atarimae hinshitsu”，只能在 kaizen 完成之后才可启动。这一步将检查影响过程的无形因素，并优化这些因素对过程的影响。例如，软件过程可能受到高层职员流动的影响，而这本身又是由公司内部不断重组而引起的。由于一个稳定的公司组织可能会对软件质量的提高有很大的帮助，所以 Atarimae hinshitsu 可以帮助管理者对公司重组方式提出建议。

前面两个步骤关注的是过程，下一个称为“kansei”的步骤(意思是“第五感觉”)则关注产品的用户(这里的产品是指软件)。Kansei 就是通过检查用户使用产品的方式，而导致产品本身的改进和(潜在地)改进产品的生产过程。

最后一个步骤称为“miryokuteki hinshitsu”，它将管理者的注意力从当前的产品上移开并拓宽。这是一个面向商业的步骤，通过观察产品在市场上的用途，寻找产品在可以识别的相关领域中的发展机会。在软件世界中，miryokuteki hinshitsu 可以被视为一种发现有利可图的新产品或寻找当前计算机系统的副产品用途的努力。

对于大多数公司而言，都应该立即关心 kaizen 步骤。在建立一个成熟的软件过程(第 2 章)之前，进入后面步骤的意义不大。

8.3 软件质量保证

即使最疲惫的软件开发者也同意，生产高质量的软件是一个十分重要的目标。但我们将如何定义质量呢？有一个笑话说：“每个程序都能够做好一些事情，不过不是我们希望它做到的那些而已。”

文献中对软件质量的定义有很多种。在我们这里对软件质量做如下定义：

明确声明的功能和性能需求、明确文档化过的开发标准、以及专业人员开发的软件所应具有的所有隐含特征都得到满足。

显然上述定义还可以修改或扩充。实际上，对软件质量的确定性定义可以无限制地争论下去。在本书中，上述定义强调了以下三个重要方面：

1.软件需求是进行“质量”度量的基础。与需求不符就是质量不高。

2.指定的标准定义了一组指导软件开发的准则。如果不能遵照这些准则，就极有可能导致质量不高。

3.通常有一组“隐含需求”是不被提及的(如对易维护性的需求)。如果软件符合了明确的需求却没有满足隐含需求，软件质量仍然值得怀疑。

8.3.1 背景

对于任何为他人生产产品的企业，质量保证都是必不可少的活动。在 20 世纪之前，质量保证曾经只由生产产品的工匠承担。第一个正式的质量保证和控制职能于 1916 年在贝尔实验室出现，此后迅速风靡整个制造行业。

在计算机发展的早期(50 和 60 年代)，质量保证曾经只由程序员承担。软件质量保证的标准是 70 年代首先在军方的软件开发合同中出现的，此后迅速传遍整个商业领域［IEE94］。通过扩颊骨懊蒌　档闹柿慷丁澹　砑　柿勘Ⅴさ亩丁迨俏　吮Ⅴと砑　柿慷　匾璧摹坝屑苹　暮拖低郴　男卸　J 健保篼 CH87］。在今天，这一定义的含义是在一个组织中有多个机构负有保证软件质量的责任——包括软件工程师、项目管理者、客户、销售人员和 SQA 小组的成员。

SQA 小组充当客户在公司内部的代表。这就是说，SQA 小组的成员必须以客户的观点看待软件。软件是否充分满足第 18 章中的各项质量因素？软件开发是否依照预先设定的标准进行？作为 SQA 活动的一部分的技术规程是否恰当的发挥了作用？SQA 小组的工作将回答上述的及其他的问题，以确保软件质量得到维护。

8.3.2 SQA 活动

软件质量保证由各种任务构成，这些任务分别与两种不同的参与者相关——做技术工作的软件工程师和负责质量保证的计划、监督、记录、分析及报告工作的 SQA 小组。

软件工程师通过采用可靠的技术方法和措施、进行正式的技术复审、执行计划周密的软件测试来考虑质量问题(并保证软件质量)。在本章中将只讨论复审问题。本书中的第 3 到第 5 部分将讨论有关的技术问题。

SQA 小组的职责是辅助软件工程小组得到高质量的最终产品。SEI [PAU93]推荐了一组有关种柿勘Ⅴ ぶ械募苹　12.喽健12.锹肌 7.治黾氨ù 娴腔 QA 活动。这些活动将由一个独立的 SQA 小组执行(或协助)：

为项目准备 SQA 计划：该计划在制定项目计划时制定，由所有感兴趣的相关部门复审。该计划将控制由软件工程小组和 SQA 小组执行的质量保证活动。在计划中要标识以下几点(参见图 8-5)：

- 需要进行的评价。
- 需要进行的审计和复审。
- 项目可采用的标准。
- 错误报告和跟踪过程。
- 由 SQA 小组产生的文档。
- 为软件项目组提供的反馈数量。

参与开发该项目的软件过程描述——软件工程小组为要进行的工作选择一个过程。SQA 小组将复审过程说明，以保证该过程与组织政策、内部软件标准、外界所订标准(如 ISO 9001)以及软件项目计划的其他部分相符。

复审各项软件工程活动、对其是否符合定义好的软件过程进行核实——SQA 小组识别、记录和跟踪与过程的偏差，并对是否已经改正进行核实。

审计指定的软件工作产品、对其是否符合定义好的软件过程中的相应部分进行核实——SQA 小组对选出的产品进行复审；识别、记录和跟踪出现的偏差、对是否已经改正进行核实、定期将工作结果向项目管理者报告。

确保软件工作及工作产品中的偏差已被记录在案，并根据预定规程进行处理——偏差可能出现在项目计划、过程描述、采用的标准或技术工作产品中。

记录所有不符合的部分，并报告给高级管理者——不符合的部分将受到跟踪直至问题得到解决。

除进行上述活动之外，SQA 小组还需要协调变化的控制和管理(参见第 9 章)，并帮助收集和分析软件度量信息。

8.4 软件复审

软件复审是软件工程过程中的“过滤器”。复审被用于软件开发过程中的多个不同的点上，起到发现错误(进而引发排错活动)的作用。软件复审起到的作用是“净化”分析、设计和编码中所产生的软件工作产品。Freedman 和 Weinberg 在 [FRE90] 中对复审的讨论如下：

技术工作需要复审的理由就象铅笔需要橡皮——“人非圣贤，孰能无过”。我们需要复审的第二个理由是：尽管人善于发现自己的某些错误，但是犯错误的人自己对许多种错误的发现能力远小于其他的人。因此复审过程正好回答了 Robert Burns 的祈祷者的祈求：

“赐予我们力量吧！让我们能够像别人看我们那样看我们自己”

一次复审(任何复审)是一种借助一组人的差异性来达到目的的方法：

1.指出一个人或小组生产的产品所需进行的改进。

2.确定产品中不需要或者不希望改进的部分。

3.得到与没有进行复审相比更加一致、或者至少更可预测的技术工作的质量，从而使得技术工作更易于管理。

在软件工程过程中可以进行的复审有许多种，它们各有用处。在咖啡机旁讨论技术问题的非正式会议是一种复审；将软件设计正式介绍给客户、管理层和技术人员也是一种复审方式。但是在本书中，我们将集中讨论正式的技术复审——有时称为“走查”(Walkthrough)。从质量保证的角度出发，正式的技术复审是最有效的过滤器。由软件工程师(或其他人)对软件工程师进行的正式技术复审(FTR)是一种提高软件质量的有效方法。

8.4.1 软件缺陷对成本的影响

《IEEE 电气和电子标准术语字典》(IEEE Standard Dictionary of Electrical and ElectronicsTerms， IEEE Standard 100-1992)中对“缺陷”(Defect)一词的定义是“产品异常”。在硬件领域中，“失败”(Failure)一词的定义可以参见 IEEE Standard 610.12-1990：

(a)硬件设备或部件的缺陷：比如线路短路或断路；(b)在计算机程序中的一个不正确的步骤、过程或者数据定义。注意：本定义主要用于容错学科。在通常的用法中，术语“错误”(Error)和“臭虫(小毛病)”(Bug)也用来表达同样的意思。参见：数据敏感性失败(data-sensitive fault)、程序敏感性失败(program-sensitive fault)、等价失败(equivalent faults)、失败屏蔽(fault masking)、间断性失败(intermittent fault)。

在软件过程范围中，术语“缺陷”和“失败”是同义词，它们都表示在软件交付给最终用户之后发现的质量问题。在前面的章节中，我们使用“错误”描述在软件交付之前由软件工程师发现的质量问题。

正式技术复审的主要目标是在此过程中发现错误，以便使缺陷在软件交付之前变成错误。正式技术复审的明显优点是较早发现错误，防止错误被传播到软件过程的后续阶段。

产业界的大量研究(TRW、Nippon Electric 和 Mitre Corp.以及其他公司)表明设计活动引入的错误占软件过程中出现的所有错误(和最终的缺陷)数量的50％到 65％。而现有研究［JON86］表拿鳎　　郊际醺瓷笞诜㈣稚杓拼砦蠓矫孀罡叽锏*75％的有效性。通过检测和排除大量设计错误，复审过程将极大降低后续开发和维护阶段的成本。

为了说明尽早发现错误对成本的影响，我们将根据从大型软件项目中收集到的实际数据研究一系列的相对成本［IBM81］。假定在设计阶段发现的错误的改正成本为 1 个货币单位，在测试开始之前发现一个错误的改正成本为 6.5 个货币单位，在测试时发现一个错误的改正成本为 15 个货币单位，而在发布之后发现一个错误的改正成本为 60 到 100 个货币单位。

8.4.2 缺陷的放大和消除

可以用“缺陷放大模型”［IBM81］来说明在软件工程过程中的概要设计、详细设计和编码越锥沃写砦蟮牟　　凹觳狻 8 媚 P 腿缤*8-2 所示。图中方块表示软件开发步骤。在这一步骤中，错误可能因为疏忽而产生；复审过程可能没有发现新产生的以及来自此步骤之前的错误，从而导致一定数量的错误通过当前步骤；在有些情况下，从前面步骤传下来的错误在当前步骤会放大(放大因子为 X)。将方块进一步划分以表示上述特点及错误检测的有效性百分比，后者是复审完善程度的函数。

图 8-3 所示的是一个假设在没有复审的软件开发过程中缺陷放大的例子。如图所示，假设每一个测试步骤都发现和改正 50％的输入错误，而又不引入新的错误(一种乐观的估计)。10 个概要设计阶段的错误在开始测试之前已经放大成为 94 个。12 个隐藏的缺陷则随软件发布到客户现场。图 8-4 所示的情况与之同样，只是在设计和编码过程中将复审作为每个软件过程步骤的一部分。在这种情况下，最初的 10 个概要设计错误在开始测试之前放大成为 24 个。只有 3 个隐藏的缺陷。回忆一下与发现和改正错误相关的相对成本，由此可以确定总开发成本

(在我们假设的例子中无论有没有复审的情况下)。在表 8-1 中，在进行了复审的情况下，开发和维护的总成本是 782 个成本单位。而在不进行复审的情况下，总成本是 2177 个成本单位。后者几乎是前者的 3 倍。

表 8-1　开发成本比较

错误发现时机	数量	成本单位	总计
进行复审			
设计期间	22	1.5	33
测试之前	36	6.5	234
测试期间	15	15	315
发布之后	3	67	201
			783

(续)

错误发现时机	数量	成本单位	总计
不进行复审			
测试之前	22	6.5	143
测试期间	82	15	1230
发布之后	12	67	804

为了进行复审，开发人员必须花费时间和工作量，开发组织必须花费金钱。但是上述例子的结果证明，我们面临的是一种“现在付出、否则以后付出更多”的情况。(设计和其他技术活动中的)正式技术复审提供了显而易见的成本效益。因此，应该进行复审活动。

8.5 正式技术复审

正式技术复审(FTR)是一种由软件工程师进行的[①]软件质量保证活动。FTR的目标是(1)在软件的任何一种表示形式中发现功能、逻辑或实现的错误；(2)证实经过复审的软件的确满足需求；(3)保证软件的表示符合预定义的标准；(4)得到以一种一致的方式开发的软件；(5)使项目更易于管理。由于FTR的进行使大量人员对软件系统中原本并不熟悉的部分更为了解，因此，FTR还起到了提高项目连续性和培训后备人员的作用。

FTR 实际上是一类复审方式，包括“走查”(Walkthrough)、“审查”(Inspection)、“轮查”(Round-robin Review)以及其他软件小组的技术评估。每次 FTR 都以会议形式进行，只有经过适当的计划、控制和参与，FTR 才能获得成功。在后面的段落中，我们给出了类似于“走查”[FRE90，GIL93]的典型正式技术复审的指南。

8.5.1 复审会议

不论选择何种 FTR 形式，每个复审会议都应该遵守下面的约束：

- 复审会议(通常)应该在 3 到 5 个人之间进行。
- 应该进行提前准备，但是每人占用工作时间应该少于 2 小时。
- 复审会议时间应该不超过 2 小时。

在上述约束之下，显然 FTR 应该关注的是整个软件中的某个特定(且较小)部分。例如，不要试图复审整个设计，而是对每个模块或者一小组模块进行走查。当 FTR 的关注范围较小时，发现错误的可能性更大。

FTR 的焦点是某个工作产品——软件的一部分(如一部分需求规约、一个模块的详细设计、一个模块的源代码清单)。开发这一产品的个人(即“生产者”)通知项目管理者工作产品已经完成，需要进行复审。项目管理者与“复审主席”联系，主席负责评估工作产品是否准备就绪，创建副本，并将其分发给两到三个“复审者”以便事先准备。每个复审者应该花 1 到 2 个小时复审工作产品、做笔记或者用其他方法熟悉这一工作。与此同时，复审主席也对工作产品进行复审、并制定复审会议的日程表，通常安排在第二天开会。

复审会议由复审主席、所有复审者和生产者参加。其中一个复审者作为“记录员”，负责记录在复审过程中发现的所有重要问题。FTR 将从介绍会议日程开始，并由生产者做简单的介绍。然后生产者将“遍历”工作产品、作出解释，而复审者将根据各自的准备提出问题。当发现问题或错误时，记录员逐个加以记录。

在复审结束时，所有 FTR 的与会者必须做出以下决定中的一个：(1)工作产品可以不经修改而被接受(2)由于严重错误而否决工作产品(错误改正后必须再次进行复审)(3)暂时接受工作产品(发现必须改正的微小错误，但是不再需要进一步复审)。作出决定之后，所有 FTR 与会者需要“签名”，以表示他们参加了此次 FTR 并且同意复审小组所做的决定。

8.5.2 复审报告和记录保存

在 FTR 期间，一名复审者(记录员)主动记录所有被提出的问题。在复审会议结束时，对这些问题进行小结，并生成一份“复审问题列表”。此外，还要完成一份简单的“复审总结报告”。复审总结报告将回答以下问题：

1. 复审什么？

2. 由谁复审？

3. 发现了什么，结论是什么？

复审总结报告通常是一页纸大小(可能还有附件)。它是项目历史记录的一部分，有可能被分发给项目管理者和其他感兴趣的参与方。

复审问题列表有两个作用：(1)标识产品中存在问题的区域(2)用作“行动条目”检查表以指导生产者进行改正。通常在总结报告中将问题列表作为附件。

建立一个跟踪规程，以保证问题列表中的每一条目都得到适当的改正，这一点非常重要。只有做到这一点，才能保证提出的问题真正得到控制。一种方法是将跟踪的责任指派给复审主席。更为正式的方法是将这一责任分配给一个独立的SQA 小组。

8.5.3 复审指南

进行正式技术复审之前必须建立复审指南，分发给所有复审者，并得到大家的认可，然后才能依照它进行复审。不受控制的复审，通常比没有复审更加糟糕。

下面给出了正式技术复审指南的最小集合：

1.复审产品，而不是复审生产者。FTR 涉及到别人和自我。如果进行得恰当，FTR 可以使所有参与者体会到温暖的成就感。如果进行得不恰当，则可能陷入一种审问的气氛之中。应该温和的指出错误，会议的气氛应该是轻松的和建设性的；不要试图贬低或羞愧别人。复审主席应该引导复审会议，以保证会议始终处于恰当的气氛和态度之中，并在讨论失去控制时应立即休会。

2.制定日程，并且遵守日程。各种类型的会议都具有一个主要缺点：放任自流。FTR 必须保证不要离题和按照计划进行。复审主席被赋予维持会议程序的责任，在有人转移话题时应该提醒他。

3.限制争论和辩驳。在复审者提出问题时，未必所有人都认同该问题的严重性。不要花时间争论这一问题，这样的问题应该被记录在案，留到会后进一步讨论。

4.对各个问题都发表见解，但是不要试图解决所有记录的问题。复审不是一个问题解决会议。问题的解决通常由生产者自己或者在其他人的帮助下来完成。问题解决应该放到复审会议之后进行。

5.作书面笔记。有时候让记录员在黑板上做笔记是个好注意，这样在记录员记录信息时，其他的复审者可以推敲措辞，并确定问题的优先次序。

6.限制参与者人数，并坚持事先做准备。两个人的脑袋好过一个，但是 14 个脑袋未必就好过 4 个。将复审涉及的人员数量保持在最小的必需值上。但是所有的复审组成员都必须事先作好准备。复审主席应该向复审者要求书面意见(以表明复审者的确对材料进行了复审。)

7.为每个可能要复审的工作产品建立一个检查表[①]。检查表能够帮助复审主席组织FTR会议，并帮助每个复审者将注意力集中在重要问题上。应该为分析、设计、编码、甚至测试文档都建立检查表。

8.为 FTR 分配资源和时间。为了让复审有效，应该将复审作为软件工程过程中的任务加以调度。而且要为由复审结果必然导致的修改活动分配时间。

9.对所有复审者进行有意义的培训。为了提高效率，所有复审参与者都应该接受某种正式培训。培训要强调的不仅有与过程相关的问题，而且应该涉及复审的心理学因素。Freedman 和 Weinberg [FRE90] 为每 20 个人有效的参与复审而估算了一份 1 个月的学习曲线。

10.复审以前所作的复审。听取汇报对发现复审过程本身的问题十分有益。最早被复审的工作产品本身可能就会成为复审指南。

由于成功的复审涉及到许多变数(如，开发者数量、工作产品类型、时间和长度、特定的复审方法等)，软件组织应该在实验中决定何种方法最为适用。Porter [POR95] 及其同事为这类实验提供了良好的指南。

8.6 SQA 的形式化方法

在前面的几节中，我们认为软件质量是大家的工作，且通过完整的分析、设计、编码和测试，以及采用正式技术复审、多层次测试策略、对软件文档及其改变进行更好的控制、及广为接受的应用软件的开发标准，有可能获得软件的高质量。另外，质量可以根据多种质量因素来定义，并可以使用各种指标和度量进行测量。

在过去的 20 年中，在软件界中有一群虽然很少但是很坚决的人们，提出软件质量保证应该采用一种更为形式化的方法。一个计算机程序可以看作一个数学对象，对于每一种程序设计语言都能够定义一套严格的语法和语义，且对于软件需求说明也出现了一种类似的严格方法(第 24 章)。一旦需求模型(说明)以一种严格的方式被表达出来，就可以采用程序正确性的数学证明来说明程序是否严格符合它的说明。

程序正确性证明不是一个新的思路。Dijkstra [DIJ76] 和 Linger 等人 [LIN79]，以及其他很多人都支持程序正确性的数学证明，并将它与结构化程序设计概念联系在一起(参见第 14 章)。今天，已经提出了各种程序正确性的形式化证明方法，我们将在第 24 和 25 章中详细讨论。

8.7 统计质量保证

统计质量保证反映了一种在产业界不断增长的趋势：质量的量化。对于软件而言，统计质量保证包括以下步骤：

1.收集和分类软件缺陷信息。

2.尝试对每个缺陷的形成原因(例如，不符合规约、设计错误、违背标准、与客户通信不力等)进行追溯。

3.使用 Pareto 规则(80％的缺陷的 20％成因有可能可以追溯到)，将这 20％(少数重要的)分离出来。

4.一旦标出少数重要的原因，就可以开始纠正引起缺陷的问题。

这一相对简单的概念代表的是为创建适合的软件工程过程的一个重要步骤，在该过程中将进行修改，以改进那些引入错误的过程要素。

为了说明这一过程，假定软件开发组织收集了为期一年的缺陷信息。有些错误是在软件开发过程中发现的，其他缺陷则是在软件交付给最终用户之后发现的。尽管发现了数以百计的不同类型的错误，但是所有错误都可以追溯到下述原因中的一个或几个：

- 说明不完整或说明错误(IES)
- 与客户通信中所产生的误解(MCC)
- 故意与说明偏离(IDS)
- 违反编程标准(VPS)
- 数据表示有错(EDR)
- 模块接口不一致(IMI)
- 设计逻辑有错(EDL)
- 不完整或错误的测试(IET)
- 不准确或不完整的文档(IID)
- 将设计翻译成程序设计语言中的错误(PLT)
- 不清晰或不一致的人机界面(HCI)
- 杂项(MIS)

为了使用统计质量保证方法，需要建一张表 8-2 那样的表格。表中显示 IES、MCC 和 EDR 是造成所有错误中的 53％的“少数重要的”原因。但是需要注意，在只考虑严重错误时，应该将 IES、EDR、PLT 和 EDL 作为“少数重要的”原因，一旦确定了什么是重要的少数原因，软件开发组织就可以开始采取改正行动了。例

如，为了改正 MCC 错误，软件开发者可能要采用方便的应用软件说明技术，以提高与客户的通信及说明的质量。为了改正 EDR，开发者可能要使用 CASE 工具进行数据建模，并进行更为严格的数据设计复审。随着少数重要原因的不断改正，新的候选错误原因也将被提到改进日程上来。

表 8-2　统计 SQA 的数据收集

	总计		严重		一般		微小	
错误	数量	百分比%	数量	百分比%	数量	百分比%	数量	百分比%
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
IMI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
总计	942	100%	128	100%	379	100%	4335	100%

当与缺陷信息集合结合使用时，软件开发者可以为软件工程过程中的每个步骤计算“错误指标”(Error index)[IEE94]。在经过分析、设计、编码、测试和发布之后，将收集到以下数据：

Ei=在软件工程过程中的第 i 步中发现的错误总数

S_i=严重错误数

M_i=一般错误数

T_i=微小错误数

PS=第 i 步的产品规模(LOC、设计说明、文档页数)

W_s、W_m、W_t分别是严重、一般、微小错误的加权因子，其推荐取值为Ws=10、Wm=3 和Wt=1。随着过程的进展，每个阶段的加权因子取值逐渐变大。也就是说，尽早发现错误的组织得益较多。

在软件工程过程中的每一步中，分别计算各个阶段的阶段指标PI_i：

$$PI_i=W_s(S_i/E_i)+W_m(M_i/E_i)+W_t(T_i/E_i)$$

错误指标E_i通过计算各个PI_i的加权效果得到，在软件工程过程中后面的步骤中遇到的错误的权值要高于在前面阶段遇到的错误权值。

$$EI=\sum(i\times PI_i)/PS=(PI_1+2PI_2+3PI_3+iPI_i)/PS$$

错误指标与表 8-2 中收集的信息相结合，可以得出软件质量的整体改进指标。

统计 SQA 的应用软件及 Pareto 规则可以用一句话概括：将时间集中用于真正重要的地方，但是首先你必须知道什么是重要的。有经验的业界人士都同意下面的观点：大多数真正麻烦的缺陷都可以追溯到数量相对有限的根本原因上。实际上大多数业界人士对软件缺陷的“真正”原因都有一种直觉，但是很少有人花时间收集数据以验证他们的感觉。通过统计 SQA 中的基本步骤，产生缺陷的少数重要原因就被分离出来，从而可以得到改正。

有关统计 SQA 的详尽讨论超出了本书的范围，感兴趣的读者请参见文献［SCH87］、荆路 AP95］和[KAN95]。

8.8 软件可靠性

计算机程序的可靠性无疑是软件整体质量的一个重要因素。如果程序在执行时总是频繁地、重复地失败，那么其他软件质量因素是否可以接受也就无从谈起。

与其他因素不同，软件可靠性可以使用历史数据和开发数据来测量、标示和估算出来。用统计术语定义的软件可靠性是“在特定环境和特定时间内，计算机程序不失败地运行的概率”［MUS87］。举例说明，程序 X 在 8 个处理小时内的可靠性估计为 0.96；也就是说，如果程序 X 执行 100 次，每次运行 8 个小时(执行时间)，则 100 次中正确运行(不失败)的次数可能是 96 次。

任何时候讨论软件可靠性，都会引出一个主要问题：“失败”一词是什么含义？在讨论软件质量和可靠性时，失败意味着与软件需求的不符。但是这一定义也存在着等级之分。失败可以仅仅是令人厌烦，也可以是造成灾难。有的失败可以在几秒钟之内得到改正，有的则需要几个星期甚至几个月的时间才能改正。让问题更加复杂的是，改掉一个失败实际上可能引入其他的失败，而这些失败最终又会引入其他的失败。

8.8.1 可靠性和可用性的度量

早期的软件可靠性测量工作中试图将硬件可靠性的数学理论(参见文献［ALV64］)外推来进行软件可靠性的预测。大多数与硬件有关的可靠性模型依据的是由于“磨损”，而不是由于设计缺陷而导致的失败。在硬件中，由于物理磨损(如温度、腐蚀、震动的影响)导致的失败远比与设计缺陷相关的失败更多。但

不幸的是，软件恰好相反。实际上所有软件失败都可以追溯到设计或实现的问题上；磨损在这里根本不存在(参见第 1 章)。

文献可靠性理论中的核心概念与它们在软件中的可用性之间的关系仍然被争论着(参见文献 [LIT89] 和[ROO90])。尽管在两种系统之间尚未建立不可辩驳的联系，但是考虑少数几个同时适用于这两种系统的简单概念却很有必要。

当我们考虑一个基于计算机的系统时，可靠性的简单度量是“平均失败间隔时间”(MTBF)，其中：

$$MTBF=MTTF+MTTR$$

(MTTF 和 MTTR 分别是“平均失败时间”和“平均修复时间”的首字母缩写)。

许多研究人员认为 MTBF 是一个远比“缺陷数/KLOC”更为有用的度量指标。简而言之，最终用户关心的是失败，而不是总缺陷数。由于一个程序中包含的每个缺陷所具有的失败率不同，总缺陷数难以表示系统的可靠性。比如考虑一个已经投入运行 14 个月的程序，程序中的许多缺陷需要经过几年的运行时间才能暴露。这种隐藏缺陷的 MTBF 可能是 50 到 100 年。还有一些尚未被发现的缺陷的失败率可能是 18 到 24 个月。即使全部排除第一种缺陷(具有较长 MTBF)，对软件可靠性的影响也微乎甚微。

除可靠性度量之外，我们必须开发一个“可用性”度量。软件可用性是指在某个给定时间点上程序能够按照需求执行的概率。其定义为：

$$可用性=MTTF/(MTTF+MTTR)\times 100\%$$

MTBF 可靠性度量对 MTTF 和 MTTR 同样敏感。而可用性度量在某种程度上对 MTTR 较为敏感，MTTR 是软件可维护性的间接度量。

8.8.2 软件的安全和危险的分析

Leveson [LEV86] 讨论了软件对安全要求非常高的系统的影响，她写道：

在对安全要求比较高的系统中使用软件之前，它们通常是由传统(不可编程)的机械和电子设备来控制的。为此在系统中设计了一套安全技术来处理那些[不可编程]中的随机失败现象。由于假定所有由于人的错误而引起的失败可以被完全防止或者在交付和运行之前得到排除，因此人的设计错误在此不被考虑。

当软件被用来对系统进行部分控制时，系统复杂性将增加一个数量级甚至更多。当使用软件时，由于人为错误而引入的微小设计错误——在基于硬件的传统控制系统中可能被发现和排除——会变得更加难以发现。

“软件的安全和危险分析”主要是解决如何标出潜在的危险(这些危险将负面影响软件系统，并导致整个系统失败)和评估潜在的危险可能对系统形成的负面影响和引起整个系统完全瘫痪的可能性。如果能够在软件工程过程的早期标出这些危险，则可以指定软件设计特性来排除或控制潜在的危险。

建模和分析过程可以带来部分软件安全。首先，根据是否关键和有风险来标识和分类危险。例如，与基于计算机的汽车驾驶控制相关的危险可能有：

- 导致不受控制的加速难于停止运动。
- 当刹车踏板踩下后不能制动。
- 开关打开后不能启动。
- 加速或减速缓慢。

一旦这些系统级的危险被标识出来，就可以使用分析技术指定这些危险发生的严重性和概率[①]。为了真正有效，软件应该被放置于整个系统中进行分析。例如，一个微小的用户输入错误(人也是系统组成成分)可能被软件错误放大，产生将机械设备置于不正确位置的控制数据，此时如果一组外部环境条件满足(而且仅当满足这些条件时)，机械设备的不正确位置将引发灾难性的失败。错误树分析[VES81]、实时逻辑[JAN86]或Petri网模型[LEV87]等分析技术可以用于预测可能引起危险的事件链，以及事件链中的各个事件出现的概率。

“错误树分析”将建立一个事件组合的顺序及并发关系的图形模型，这些事件组合可能导致危险的事件发生或系统状态出现。使用一个开发得很好的错误树，有可能对在不同系统构件中发生的一连串相互关联的失败所产生的后果进行观察。“实时逻辑”(RTL)通过说明事件以及响应事件的行动来建立系统模型，可以使用逻辑操作来分析事件-行为模型，以对有关系统构件及其时序关系的安全性进行测试。“Petri 网”模型可以用于确定哪些错误最具有危险性。

危险标识和分析完成之后，就可以进行软件中与安全相关的需求说明了。即，这个需求说明可以包括一张不希望发生的事件列表、以及针对这些事件所希望产生的系统响应。也就指明了软件在管理不希望发生的事件方面所起的作用。

尽管软件可靠性与软件安全性相互之间关系紧密，但是理解它们之间的微妙差异更为重要。软件可靠性使用统计分析方法确定软件失败发生的可能性；而失败的发生未必导致危险或灾难。软件安全性则考察失败会导致灾难发生的条件。也就是说，失败不是被置于真空之中，而是被放在整个计算机系统范围内加以考虑。

有关软件安全和危险分析的详尽讨论超出了本书的范围。感兴趣的读者可以参见 Leverson 的专著 [LEV95] 。

8.9 SQA 计划

“SQA 计划”为建立软件质量保证提供了一张行路图。该计划由 SQA 小组和项目组共同制定，充当了每个软件项目中的 SQA 活动的模板。

表 8-3 所示的是由 IEEE［IEE94］推荐的 SQA 计划大纲。开始部分描述目的和文档范围，并指出质量保证所覆盖的软件过程活动。所有在 SQA 计划中提到的文档都被列出来，且所有可应用的标准都专门注明。计划的“管理”部分描述 SQA 在组织结构中的位置，SQA 任务和活动、及它们在整个软件过程中的位置，以及与产品质量有关的组织角色和责任。

表 8-3　ANSI/IEEE Std.730-1984 和 983-1986 软件质量保证计划

Ⅰ.计划的目的	a.软件需求复审
Ⅱ.参考文献	b.设计复审
Ⅲ.管理	C.软件验证和确认复审
1.组织	d.功能审计
2.任务	e.物理审计
3.责任	f.过程内部审计
Ⅳ.文档	g.管理复审
1.目的	Ⅶ.测试
2.所需的软件工程文档	Ⅷ.问题报告和改正行动
3.其他文档	Ⅸ.工具、技术和方法学
Ⅴ.标准、实践和约定	Ⅹ.代码控制
1.目的	Ⅺ.媒体控制
2.约定	Ⅻ.供应商控制
Ⅵ.复审和审计	XIII.记录收集、维护和保留
1.目的	XIV.培训
2.需求复审	XV.风险管理

“文档”一节描述的是软件过程各个部分所产生的各种工作产品，包括：

- 项目文档(例如项目计划)。
- 模型(例如 ERD 模型、类层次模型)。
- 技术文档(例如说明、测试计划)。
- 用户文档(例如帮助文件)。

另外在这一节中还定义了为获得高质量产品所能接受的工作产品的最小集合。

在“标准、实践和约定”中列出了所有在软件过程中采用的合适的标准和实践方法(例如，文档标准、编码标准和复审指南)。此外，还列出了作为软件工程工作的组成部分而收集的所有项目、过程及(在某些情况下)产品度量信息。

计划中的“复审和审计”一节标识了软件工程小组、SQA 小组和客户进行的审计和复审活动。它给出了各种复审和审计方法的总览。

“测试”一节中列出了软件测试计划和过程(参见第 17 章)。它还定义了测试记录保存的需求。“问题报告和改正行动”中定义了错误及缺陷的报告、跟踪和解决规程，这些活动的组织责任也被标识出来。

“SQA 计划”的其他部分标识了支持 SQA 活动与任务的工具和方法；给出了控制变化的软件配置管理过程；定义了一种合同管理方法；建立了组装、保护、维护所有记录的方法；标识了为满足这一计划所需的培训；定义了标识、评估、监控和控制风险的方法。

8.10 ISO 9000 质量标准

“质量保证系统”可以被定义成用于实现质量管理的组织结构、责任、规程、过程和资源［ANS87］。 ISO 9000 标准以一种能够适用于任何行业(不论提供的是何种产品或服务)的一般术语描述了质量保证的要素。

为了登记成为 ISO 9000 中包含质量保证系统模型中的一种，一个公司的质量系统和操作应该由第三方审计者仔细检查，查看其与标准的符合性以及操作的有效性。成功登记之后，这一公司将收到由审计者所代表的登记实体颁发的证书。此后每半年进行一次的检查性审计将持续保证该公司的质量系统与标准是相符的。

8.10.1 ISO 对质量保证系统的方法

ISO 9000 质量保证模型将一个企业视为一个互联过程的网络。为了使质量系统符合 ISO 标准，这些过程必须与标准中给出的区域对应，并且必须按照描述文档化和实现。对一个过程文档化将有助于组织的理解、控制和改进。正是理解、控制和改进过程网络的机会为设计和实现符合 ISO 的质量系统的组织提供了(也许是)最大的效益。

ISO 9000 以一般术语描述了一个质量保证系统的要素。这些要素包括用于实现质量计划、质量控制、质量保证和质量改进所需的组织结构、规程、过程和资源。但是 ISO 9000 并不描述一个组织应该如何实现这些质量系统要素。因此，真正的挑战在于如何设计和实现一个能够满足标准并适用于公司的产品、服务和文化的质量保证系统。

8.10.2 ISO 9001 标准

ISO 9001 是应用于软件工程的质量保证标准。这一标准中包含了高效的质量保证系统必须体现的 20 条需求。因为 ISO 9001 标准适用于所有的工程行业，

因此，为了在软件过程的使用中帮助解释该标准，而专门开发了一个 ISO 指南的子集(即 ISO 9000-3)。由 ISO 9001 描述的 20 条需求所面向的是以下问题：1. 管理责任。

2. 质量系统。

3. 合同复审。

4. 设计控制。

5. 文档和数据控制。

6. 采购。

7. 对客户提供的产品的控制。

8. 产品标识和可跟踪性。

9. 过程控制。

10. 审查和测试。

11. 审查、度量和测试设备的控制。

12. 审查和测试状态。

13. 对不符合标准产品的控制。

14. 改正和预防行动。

15. 处理、存储、包装、保存和交付。

16. 质量记录的控制。

17. 内部质量审计。

18. 培训。

19. 服务。

20. 统计技术。

软件组织为了通过 ISO 9001，就必须针对上述每一条需求建立相关政策和过程，并且有能力显示组织活动的确是按照这些政策和过程进行的。有关 ISO 9001 的进一步信息，感兴趣的读者可以参见文献 [SCH94] 和[ESE95]。

8.11 小结

软件质量保证是在软件过程中的每一步都进行的“保护性活动”。SQA 包括对方法和工具有效应用的规程、正式技术复审、测试策略和技术、变化控制规程、保证与标准符合的规程、以及度量和报告机制。

软件质量的复杂本质——这是计算机程序的一种属性，其定义是“与明确地和隐含地定义的需求的符合程度”——使 SQA 很复杂。但是当被更为一般地考虑时，软件质量包括了许多不同的产品和过程因素及其相关的度量。

软件复审是最为重要的 SQA 活动之一。复审的作用是作为软件过程的过滤器，在发现及改正错误的成本相对较小时就排除错误。正式技术复审或走查是一种典型的复审会议，在实践中这种形式对于发现错误极其有效。

为了正确的进行软件质量保证，必须收集、评估和发布软件工程过程的数据。统计 SQA 有助于改进产品和软件过程本身的质量。软件可靠性模型将度量加以扩展，能够由所收集的缺陷数据推导出项目失败率和可靠性估计。

总而言之，让我们借用 Dunn 和 Ullman [DUN82] 的一句话：“软件质量保证就是将质量保证的管理对象和设计原则映射到适用的软件工程管理和技术空间上”。质量保证的能力是成熟的工程学科的量尺。当上述映射成功实现时，其结果就是成熟的软件工程。

思考题

s8.1 在本章前面我们提到“差异控制是质量保证的核心”。因为每个程序都与其他程序不同，我们应该寻找什么样的差异？应该如何控制差异？

8.2 如果客户不断改变要做什么的想法，是否有可能评估软件质量？

8.3 质量和可靠性是相关的概念，但却有一些基本的不同。请就此进行讨论。

8.4 一个程序能够既正确又不可靠吗？请加以解释。

8.5 一个程序能够既正确又不表现出高质量吗？请加以解释。

8.6 为什么软件工程小组与独立的软件质量保证小组之间的关系经常是紧张的？这种紧张关系是否是正常的？

8.7 假设你被赋予改进组织中的软件质量的职责。你要做的第一件事是什么？然后呢？

8.8 除了可以对错误计数之外，还有哪些可以计数的软件特征具有质量意义？它们是什么？能否直接度量？

8.9 仅当每个参与者都进行了事先准备时，一次正式技术复审才是有效的。你怎样能够发现复审参与者是否进行了准备？如果你是复审主席，你该怎么办？

8.10 有人认为 FTR 应该主要针对编程风格和正确性。这种主意好吗？为什么？

8.11 复审表 8-2，并从中选择导致严重和一般错误的 4 个“少数重要的”原因。用其他章节中给出的信息，提出相应的改正行动。

8.12 一个组织采用了 5 步的软件工程过程，其中错误的发现情况如下表中的百分比分布所示：

步骤	发现错误的百分比
1	20%
2	15%
3	15%
4	40%
5	10%

用表 8-2 中的信息和上述百分比分布，计算该组织的整体缺陷指标(假定 PS 为 100，000)。

8.13 研究软件可靠性的文献，写一篇文章描述一个软件可靠性模型(一定要给出一个例子)。

8.14 关于软件 MTBF 概念的争论仍然进行着。为什么，你能够想到几个原因吗？

8.15 给出两个安全性非常关键的计算机控制系统。为每个系统列出至少 3 条与软件失败直接相关的危险。

8.16 获取一份ISO9001 和ISO 9000-3 的副本。准备一个演示作业，讨论 3 个ISO 9001 需求及其在软件行业中如何应用。

① 本节由MichaelStovsky编写，根据《ISO9000 基本原理》改编。后者是由R.S.Pressman&Associates，Inc.开发的业务手册性质的录像教程《软件工程要点》中的一部分，其复制得到了R.S.Pressman&Associates，Inc.的许可。

① 参见文献［ART93］中关于TQM及其在软件世界中的应用的详尽讨论，参见[KAP95]中关于在软件世界中应用aldridgeAward准则的讨论。

① 在有些情况下，其他参与方（如客户、市场人员、技术支持人员等）也可能参加正式技术复审。

① 在WWW的FTRArchive上可以找到大量有关复审检查表的信息。有关详细信息参

见“推荐阅读文献及其他信息源”。

① 这种方法类似于在第 6 章的软件项目管理中描述的风险分析方法。两者的主要不同在于前者关心的是技术问题，而不是与项目有关的问题。

第 9 章　软件配置管理

当建造计算机软件时，变化(change)是不可避免的，并且，变化使得共同工作在某一项目中的软件工程师之间的彼此不理解程度更加增高。当变化进行前没有经过分析、变化实现前没有被记录、没有向那些需要知道的人报告变化、或变化没有以可以改善质量及减少错误的方式被控制时，则不理解性将会产生。Babich [BAR86] 对此有如下陈述：

协调软件开发以减少不理解性到最小程度的技术称为配置管理。配置管理是对正在被一个项目组建造的软件的修改标识、组织和控制的技术，其目标是通过最大限度地减少错误，来最大限度地提高生产率。

软件配置管理(SCM)是贯穿于整个软件过程中的保护性活动。因为变化可能发生在任意时间，SCM 活动被设计来(1)标识变化，(2)控制变化，(3)保证变化被适当地实现，以及(4)向其他可能有兴趣的人员报告变化。

明确地区分软件维护和软件配置管理是很重要的。维护是发生在软件已经被交付给客户，并投入运行后的一系列软件工程活动，而软件配置管理则是当软件项目开始时就开始，并且仅当软件退出运行后才终止的一组跟踪和控制活动。

软件配置管理的主要目标是使改进变化可以更容易地被适应，并减少当变化必须发生时所需花费的工作量。本章，我们将讨论在管理变化中必须发生的特定活动。

9.1 软件配置管理

软件过程的输出信息可以分为三个主要的类别：(1)计算机程序(源代码和可执行程序)，(2)描述计算机程序的文档(针对技术开发者和用户)，以及(3)数据(包含在程序内部或在程序外部)。这些项包含了所有在软件过程中产生的信息，总称为软件配置。

随着软件过程的进展，软件配置项(Software Configuration Items，SCI)迅速增长。系统规约产生了软件项目计划和软件需求规约(以及硬件相关的文档)，这些然后又产生了其他的文档，从而建立起一个信息层次。如果每个 SCI 仅仅简单地产生其他 SCI，则几乎不会产生混淆。不幸的是，另一个变量进入到过程中，即变化，为了任意的理由，变化可能随时发生。事实上，正如第一本系统工程法 [BER80] 中所说：不管你在系统生命期的什么地方，系统都将会发生变化，并且对变化的希望将持续于整个生命期中。

变化的起源是什么呢？对这个问题的回答就象变化本身一样那么多变。然而，有四种基本的变化源：

• 新的商业或市场条件，引起产品需求或业务规则的变化。

• 新的客户需要，要求修改信息系统产生的数据、产品提供的功能、或基于计算机的系统提供的服务。

• 改组和/或企业规模减小，导致项目优先级或软件工程队伍结构的变化。

• 预算或进度的限制，导致系统或产品的重定义。

软件配置管理是一组用于在计算机软件的整个生命期内管理变化的活动。SCM 可被视为应用于整个软件过程的软件质量保证活动。在下面几节中，我们考察能够帮助我们管理变化的主要的 SCM 任务和重要的概念。

9.1.1 基线

变化是软件开发中必然的事情。客户希望修改需求，开发者希望修改技术方法，管理者希望修改项目方法。这些修改是为什么？回答实际上相当简单，随着时间的流逝，所有相关人员也就知道更多信息(关于他们需要什么、什么方法最好、以及如何实施并赚钱)，这些附加的知识是大多数变化发生的推动力，并导致这样一个对于很多软件工程实践者而言难于接受的事实：大多数变化是合理的!

基线是一个软件配置管理的概念，它帮助我们在不严重阻碍合理变化的情况下来控制变化。IEEE(IEEE Std.610.12-1990)定义基线如下：

已经通过正式复审和批准的某规约或产品，它因此可以作为进一步开发的基础，并且只能通过正式的变化控制过程的改变。

一种类比方式描述基线的方式是：

考虑某大饭店的厨房门，为了减少冲突，一个门被标记为“出”，其他门被标记为“进”，门上有机制，允许它们仅能朝适当的方向打开。

如果某侍者在厨房里拿起一盘菜，将它放在托盘上，然后，他意识到他拿错了盘子，他可以迅速而非正式地在他离开厨房前改变成正确的盘子。

然而，如果他已经离开了厨房，并将菜给了顾客，然后被告知他犯了一个错误，那时，他就必须遵循下面一组规程：(1)查看帐单以确定是否错误已经发生；(2)向顾客道歉；(3)通过“进”门返回厨房；(4)解释该问题，等等。

基线类似于饭店中从厨房里传送出去的盘子，在软件配置项变成基线前，变化可以迅速而非正式地进行，然而，一旦基线已经建立，我们就得象通过一个单

向开的门那样，变化可以进行，但是，必须应用特定的、正式的规程来评估和验证每个变化。

在软件工程的范围内，基线是软件开发中的里程碑，其标志是有一个或多个软件配置项的

交付，且这些 SCI 已经经过正式技术复审而获得认可(第 8 章)。例如，某设计规约的要素已经形成文档并通过复审，错误已被发现并纠正，一旦规约的所有部分均通过复审、纠正、然后认可，则该设计规约就变成了一个基线。任何对程序体系结构(包含在设计规约中)进一步的变化只能在每个变化被评估和批准之后方可进行。虽然基线可以在任意的细节层次上定义，但最常见的软件基线如图 9-1 所示。

产生基线的事件的进展如图 9-2 所示。软件工程任务产生一个或多个 SCI，在 SCI 被复审并认可后，它们被放置到项目数据库(也称为项目库或软件中心库)中。当软件工程项目组中的某个成员希望修改某个基线 SCI 时，该 SCI 被从项目数据库拷贝到工程师的私有工作区中，然而，这个提取出的 SCI 只有在遵循 SCM 控制(在本章后部讨论)的情况下才可以被修改。图 9-2 中的虚线箭头说明了对某个作为基线的 SCI 的修改路径。

9.1.2 软件配置项

我们已经将软件配置项定义为部分软件工程过程中创建的信息，在极端情况下，一个 SCI 可被考虑为某个大的规约中的某个单独段落，或在某个大的测试用例集中的某种测试用例，更实际地，一个 SCI 是一个文档、一个全套的测试用例、或一个已命名的程序构件(例如，C++函数或 Ada95 软件包)。

以下的 SCI 成为配置管理技术的目标并形成一组基线：

1. 系统规约

2. 软件项目计划

3. 软件需求规约

a. 图形分析模型

b. 处理规约

c. 原型

d. 数学规约

4. 初步的用户手册

5.设计规约

a.数据设计描述

b.体系结构设计描述

c.模块设计描述

d.界面设计描述

e.对象描述(如果使用面向对象技术)

6.源代码清单

7.测试规约

a.测试计划和过程

b.测试用例和结果记录

8.操作和安装手册

9.可执行程序

a.模块的可执行代码

b.链接的模块

10.数据库描述

a.模式和文件结构

b.初始内容

11.联机用户手册

12.维护文档

a.软件问题报告

b.维护请求

c.工程变化命令

13.软件工程的标准和规程

除了上面列出的 SCI，很多软件工程组织也将软件工具列入配置管理之下，即，特定版本的编辑器、编译器和其他 CASE 工具被“固定”作为软件配置的一部分。因为这些工具被用于生成文档、源代码和数据，所以当对软件配置进行改变时，必然要用到它们。虽然问题并不多见，但有可能某工具的新版本(如，编译器)可能产生和原版本不同的结果。为此，工具，就象它们辅助生产的软件一样，可以被基线化，并做为综合的配置管理过程的一部分。

在现实中，SCI 被组织成配置对象，它们有自己的名字，并被归类到项目数据库中。一个配置对象有名字、属性，并通过关系和其他对象“联结”。在图 9-3 中，配置对象的“设计规约”、“数据模块”、“模块 N”、“源代码”和“测试规约”被分别定义。然而，每个对象如图中箭头所示和其他对象关联。曲线箭头指明组装关系，即“数据模块”和“模块 N”是对象“设计规约”的一部分。直线的双箭头指明相互关系，如果“源代码”对象发生变化，相互关系使得软件工程师能够确定哪些其他对象(和 SCIs)可能被影响。

9.2 SCM 过程

软件配置管理是软件质量保证的重要一环，其主要责任是控制变化。然而，SCM 也负责个体 SCI 和软件的各种版本的标识、软件配置的审计(以保证它已被适当地开发)、以及配置中所有变化的报告。

任何关于 SCM 的讨论均涉及一系列复杂问题：

- 一个组织如何标识和管理程序(及其文档)的很多现存版本，以使得变化可以高效地进行？
- 一个组织如何在软件被发布给客户之前和之后控制变化？
- 谁负责批准变化，并给变化确定优先级？
- 我们如何保证变化已经被恰当地进行？
- 采用什么机制去告知其他人员已经实行的变化？

这些问题导致我们对五个 SCM 任务的定义：标识、版本控制、变化控制、配置审计和报告。

9.3 软件配置中对象的标识

为了控制和管理软件配置项，每个配置项必须被独立命名，然后用面向对象的方法组织。有两种类型的对象可以被标识[CHO89]：基本对象和聚集对象(已有人建议将聚集对象作为一个代表软件配置完整版本的机制)。基本对象是软件工程师在分析、设计、编码或测试中创建的“文本单元(unit of text)”，例如，

一个基本对象可能是需求规约的一个段落、模块的源程序清单或一组用于测试代码的测试用例。一个聚集对象是基本对象和其他聚集对象的集合，在图 9-3 中，“设计规约”是一个聚集对象，在概念上，它可被视为一个已命名的(标识的)指针表，指向基本对象“数据模块”和“模块 N”。

每个对象均具有一组唯一地标识它的独特的特征：名字、描述、资源表、以及“现实”。对象名是无二义性地标识对象的一个字符串；对象描述是一个数据项的列表，它们标识：

• 该对象所表示的 SCI 类型(如，文档、程序、数据)

• 项目标识符；以及变化和/或版本信息；

资源是“由对象提供、处理、引用或需要的实体”[CHO89]，例如，数据类型、特定函数、或甚至变量名也可以作为对象资源；现实是一个指针，对基本对象而言指向“文本单元”，对聚集对象而言则指向 null。

配置对象的标识也必须考虑存在于命名对象之间的关系，一个对象可被标识为某聚集对象的〈part-of〉，关系〈part-of〉定义了一个对象层次，例如，使用下面的简单表示：

E-R diagram 1.4 〈part-of〉 data model;

Data model 〈part-of〉 Design Specification;

这样，我们创建了一个 SCI 的层次结构。

假定在一个对象层次中的唯一关系是沿着层次树的路径是不现实的，在很多情况下，对象跨越对象层次分支的相互关联，例如，“数据模块”和数据流程图(假定使用结构化分析)彼此关联，也和一组针对某特定等价类的测试用例相关联。这些交叉的结构关系可以用下面的方式表示：

data model 〈interrelated〉 data flow model;

data model 〈interrelated〉 test case class m;

在第一种情况下，相互关系是存在于组装对象之间，而第二种关系则是在聚集对象(datamodel，数据对象)和基本对象(测试用例类 m)之间。

配置对象之间的相互关联关系可以用模块互联语言(module interconnection language，MIL)[NAR87]表示，AMIL 描述配置对象间的相互依赖，并能够自动建立系统的任意版本。

对于软件对象的标识模式必须认可对象在整个软件过程中的演化，在一个对象被确定为基线前，它可能会变化很多次，甚至在基线已经建立后，变化也可能经常发生。有可能为任意对象创建一个演化图(evolution graph)[GUS89]，演

化图描述了对象的变化历史，如图 9-4 所示。配置对象 1.0 经过修改，变成对象 1.1，小的纠正和变化导致版本 1.1.1 和 1.1.2，随后的较大更新得到对象 1.2。对象 1.0 演化继续产生 1.3 和 1.4，但是同时，一个对于对象的大修改导致了新的演化路径，版本 2.0。两个版本当前都支持。

变化有可能对任意版本进行，但是没必要对所有版本进行。开发者如何引用版本 1.4 的所有模块、文档和测试用例？市场部门如何知道当前使用版本 2.1 的客户是哪些？我们如何确定对版本 2.1 源代码的修改已适当地反应在对应的设计文档中？对上面所有问题的回答的关键在于标识。

一系列自动的 SCM 工具(如，CCC、RCS、SCCS、Aide-de-Camp)已经开发出来以辅助标识(及其他 SCM)任务，在某些情况下，工具被设计为仅仅保持最新版本的完整拷贝，为了得到(文档或程序的)早期版本，要从最新版本中“减去”变化(由工具进行分类)［TIC82］。这个模式使得当前的配置立即可用，而其他版本也易于得到。

9.4 版本控制

版本控制结合了规程和工具以管理在软件工程过程中所创建的配置对象的不同版本。Clemm［CLE89］给出了在 SCM 中版本控制的如下描述：

配置管理使得用户能够通过对适当版本的选择来指定可选的软件系统的配置，这一点的实现是通过将属性关联到每个软件版本上，然后通过描述一组所期望的属性来指定(和构造)配置的。

上面提到的“属性”可以简单到赋给每个对象的特定版本号，或复杂到用以指明系统中特定类型的功能变化的布尔变量(开关)串［LIE89］。

一个系统的不同版本的一种表示方式是如图 9-4 的演化图，图中每个结点均是聚集对象，即，软件的完整版本。软件的每个版本是一组 SCI(源代码、文档、数据)的集合，并且每个版本可能由多种不同的变体(variant)组成。为了阐明这个概念，考虑一个由构件 1、2、3、4 和 5 组成的简单程序的版本(图 9-5)(此处，“构件”可以是 SCI 基线的，构件 4 仅仅当软件用彩色显示器实现时才被使用，构件 5 仅当单色显示器可用时才被实现，因此，可以定义该版本的两个变体：(1)构件 1、2、3 和 4；(2)构件 1、2、3 和 5。

为了构造某程序的给定版本的适当变体，可以为每个构件赋上一个“属性元组”——一个特征表，当某软件版本的特殊变体被构造时，它们将定义是否应该使用该构件。对每个变体赋上一个或多个属性，例如，color 属性用于定义支持当彩色显示器时，应该使用哪个构件。

另一种将构件、变体和版本(修改版)之间的关系概念化的方法是将它们表示成对象池(object pool)［REI89］，如图 9-6 所示，配置对象与构件、变体和版本之间的关系可以表示为一个三维空间。一个构件由一组在同一修改层次上的对

象组成；变体是同一修改层次上的一组不同对象，因此，是和其他变体平行共存的；当对一个或多个对象进行了较大的修改时，就定义了一个新的版本。

在过去十年中，已经提出了一系列不同的版本控制的自动方法，这些方法之间的主要不同在于用于构造系统的特定版本及变体的属性的复杂程度，以及构造过程的机制。在早期的系统中，如 SCCS［ROC75］，属性只取数字值；在后来的系统，如 RCS[TIC82]，使用了符号化的修改关键字；现代系统，如 NSE 或 DSEE［ADA89］，则建立了可用于构造变体或新版本的版本规约。这些系统也支持基线概念，因此，排除了对特定版本进行无控制修改(或删除)的可能性。

9.5 变化控制

对于大型的软件开发项目，无控制的变化将迅速导致混乱，变化控制结合人的规程和自动化工具以提供一个变化控制的机制。变化控制过程如图 9-7 所示，一个变化请求被提交和评估，以评价技术指标、潜在副作用、对其他配置对象和系统功能的整体影响、以及对于变化的成本预测。评估的结果以变化报告的形式给出，该报告被变化控制审核者(change control authority，CCA)——对变化的状态及优先级作最终决策的人或小组——使用。对每个被批准的变化生成一个工程变化命令(engineering change order，ECO)，ECO 描述了将要进行的变化、必须注意的约束、以及复审和审计的标准。将被修改的对象从项目数据库“提取(check out)”出来，进行修改，并应用于合适的 SQA 活动，然后，将对象“提交(check in)”进数据库，并使用合适的版本控制机制(9.4 节)去建立软件的下一个版本。

“提交”和“提取”过程实现了两个主要的变化控制要素——访问控制和同步控制。访问控制管理哪个软件工程师有权限去访问和修改某特定的配置对象，同步控制帮助保证由两个不同的人员完成的并行修改不会互相覆盖[HAR89]。

访问和同步控制流程如图 9-8 所示。基于一个经过批准的变化请求和 ECO，软件工程师提取出配置对象，访问控制功能保证该软件工程师有权限提取该对象，而同步控制对项目数据库中的该对象加锁，使得当前提取出的版本在被放回以前不能对它作任何其他修改。注意，可以提取出其他的备份，但是，不能进行其他修改。基线对象的备份，称为“提出版本(extractedversion)”，由软件工程师修改，在经过恰适的 SQA 和测试后，提交对象修改后的版本，且解锁新的基线对象。

某些读者可能开始对变化控制过程的描述中所蕴含的繁文缛节感到不舒服，这个感觉并不是少见的。没有合适的安全措施，变化控制可能会阻碍进展，并产生不必要的繁琐手续。大多数拥有变化控制机制的软件开发者(不幸的是，很多人没有)已经建立了一些控制层来帮助避免上面提到的问题。

在 SCI 变成基线之前，只需要进行非正式的变化控制。配置对象(SCI)的开发者可以进行任何被管理和技术需求证明是合适的修改(只要修改不会影响到在开发者工作范围之外的更广的系统需求)，一旦对象已经经过正式的技术复审并

已被认可，则创建了一个基线。一旦 SCI 变成基线，则项目级的变化控制就开始实施了。这时，为了进行修改，开发者必须获得项目管理者的批准(如果变化是“局部的”)，或 CCA 的批准(如果该变化影响到其他 SCI)。在某些情况下，变化请求、变化报告和 ECO 的正式生成可以省略，然而，需要管理对每个变化的评价，并对所有变化进行跟踪和复审。

当软件产品发布给客户时，正式的变化控制就开始实施了，正式的变化控制规程已在图 9- 7 中概要给出。

变化控制审核者(CCA)在第二和第三层控制上扮演了活跃的角色，依赖于软件项目的规模和性质，CCA 可能包含一个人——项目管理者——或一组人(如，来自软件、硬件、数据库工程、支持、市场等方面的代表)。CCA 的角色是从全局的观点来评估变化对 SCI 之外的事物的影响：变化将如何影响硬件？变化将如何影响性能？变化将如何改变客户对产品的感觉？变化将如何影响产品的质量和可靠性？这些和很多其他的问题需被 CCA 处理。

9.6 配置审计

标识、版本控制和变化控制帮助软件开发者维持秩序，否则情况可能将是混乱和不固定的。然而，即使最成功的控制机制也只能在 ECO 产生后才可以跟踪变化。我们如何保证变化被合适的实现呢？回答是两方面的：(1)正式的技术复审；和(2)软件配置审计。

正式的技术复审(在第 8 章中已经详细讨论过)关注已经被修改的配置对象的技术正确性，复审者评估 SCI 以确定它与其他 SCI 的一致性、遗漏、及潜在的副作用，正式的复审应该对所有变化进行，除了那些最琐碎的变化之外。

软件配置审计通过评估配置对象的通常不在复审中考虑的特征，而形成正式复审的补充。审计询问并回答如下问题：

1.在 ECO 中说明的变化已经完成了吗？加入了任意附加的修改吗？

2.是否已经进行了正式的技术复审，以评估技术的正确性？

3.是否适当地遵循了软件工程标准？

4.变化在 SCI 中被“显著地强调(hightlighted)”了吗？是否指出了变化的日期和变化的作者？配置对象的属性反应了变化吗？

5.是否遵循了标注变化、记录变化并报告变化的 SCM 规程？

6.所有相关的 SCI 被适当修改了吗？

在某些情况下，审计问题被作为正式的技术复审的一部分而询问，然而，当 SCM 是一个正式的活动时，SCM 审计由质量保证组单独进行。

9.7 状态报告

配置状态报告(Configuration status reporting，有时称为 status accounting)是一个 SCM 任务，它回答下列问题：(1)发生了什么事？(2)谁做的此事？(3)此事是什么时候发生的？(4)将影响别的什么吗？

配置状态报告(CSR)的信息流如图 9-7 所示，每次当一个 SCI 被赋上新的或修改后的标识时，则一个 CSR 条目被创建；每次当一个变化被 CCA 批准时(即，一个 ECO 产生)，一个 CSR 条目被创建；每次当配置审计进行时，其结果作为 CSR 任务的一部分被报告。CSR 的输出可以放置到一个联机数据库中［TAY85］，使得软件开发者或维护者可以通过关键词分类访问变化信息。此外，CSR 报告被定期地生成，并允许管理者和开发者评估重要的变化。

配置状况报告在大型软件开发项目的成功中扮演了重要角色，当涉及到很多人员时，有可能会发生“左手不知道右手在做什么”的综合症。两个开发者可能试图以不同的或冲突的意图去修改同一个 SCI；软件工程队伍可能花费几个月的工作量针对过时的硬件规约建造软件；能认识到被建议的修改有严重副作用的人并不知道该修改已经进行。CSR 通过改善所有相关人员之间的通信，而帮助排除这些问题。

9.8 SCM 标准

在过去 20 年中，已经提出了一系列的软件配置管理标准。很多早期的 SCM 标准，如 MIL—STD—483，DOD—STD—480A，和 MIL—STD—1521A，主要用于为军事用途而开发的软件。然而，最近的 ANSI/IEEE 标准，如 ANSI/IEEE Std.No.828-1983，No.1042-1987，和 Std.No.1028-1988［IEE94］，可应用于商业软件，并被向大型的和小型的软件工程组织推荐。

9.9 小结

软件配置管理是应用于整个软件过程中的保护性活动。SCM 标识、控制、审计和报告在软件开发过程中及在它已被发布给客户之后发生修改。所有作为软件过程的一部分而产生的信息成为软件配置的一部分，配置被适当地组织，使得可以进行有秩序的变化控制。

软件配置由一组相关联的对象构成，也称为软件配置项，它们作为某些软件工程活动的结果而产生。除了文档、程序和数据外，用于创建软件的开发环境也被置于配置控制之下。

一旦某配置对象已被开发和复审，它就变成基线，对基线对象的修改将导致该对象的新版本的建立。程序的演化可以通过检查所有配置对象的修改历史进行

跟踪。基本对象和聚集对象形成一个对象池，变体和版本可从中产生。版本控制是为了管理这些对象而使用的一组规程和工具。

变化控制是一个规程活动，它能在对配置对象进行修改时保证质量和一致性。变化控制过程从变化请求开始，然后决定是否拒绝该变化请求，最后，对将被修改的 SCI 进行可控制的更新。

配置审计是一个 SQA 活动，它有助于确保进行修改时仍能维持质量。状况报告提供关于每个变化的信息给那些需要知道的人。

思考题

9.1 为什么第一本系统工程法是真的？它将如何影响我们对软件工程范型的理解？

9.2 用你自己的话讨论使用基线的理由。

9.3 假定你是某个小项目的管理者，你将为项目定义什么基线？以及如何控制它们？

9.4 设计一个项目数据库系统，它使得软件工程师能够存储、交叉引用、跟踪、更新、修改所有重要的软件配置项。数据库将如何处理相同程序的不同版本？源代码的处理会与文档的处理有所不同吗？两个开发者将如何避免在相同时间内对相同的 SCI 进行不同的修改？

9.5 研究面向对象数据库，并撰写一篇描述它们如何被用于 SCM 中的文章。

9.6 使用 E-R 模型(第 12 章)描述在 9.1.2 节中列出的 SCI(对象)间的相互关系。

9.7 研究某现有的 SCM 工具，并描述它如何实现对于版本、变体及配置对象的控制。

9.8 关系〈part-of〉和〈interrelated〉表示了配置对象之间的简单关系，描述 5 种其他的可能在项目数据库中有用的关系。

9.9 研究某现有的 SCM 工具，并描述它如何实现版本控制的机制。此外，阅读 2-3 提到的参考篇在本章中论文，并描述用于版本控制的不同数据结构和引用机制。

9.10 用图 9.7 作为指南，为变化控制建立一个更详细的工作细分表。描述 CCA 的角色并给出变化请求、变化报告和 ECO 的格式。

9.11 开发一个在配置审计中使用的检查表。

9.12 SCM 审计和正式的技术复审之间有什么不同？它们的功能可以被放置在一个复审中吗？正反两面的观点各是什么？

第三部分　传统软件工程方法

在本书的这一部分，我们考虑那些可应用于计算机软件的分析、设计和测试的技术概念方法和测度。下面章节中，我们将涉及下列问题：

• 如何在一个大型系统的范围内设计软件？产品工程和信息工程在什么地方发挥作用？ • 可应用于软件需求分析的基本概念和原则是什么？

• 什么是结构化分析？它的各种模型如何使得软件工程师能够理解数据、功能和行为？ • 软件设计活动中使用的基本概念和原则是什么？

• 如何创建数据、体系结构、过程和界面等设计模型？

• 实时系统有什么独特的特性？这些特性如何影响这种系统的分析和设计方式？ • 可应用于软件测试的基本概念和原则是什么？

• 如何使用黑盒和白盒测试方法来设计有效的测试用例？

• 软件测试的策略是什么？

• 什么技术度量可用于评估分析和设计模型、源代码、以及测试用例？一旦这些问题得到回答，你将了解如何使用严格的工程方法去建造软件。

第 10 章　系统工程

480 年前，Machiavelli 说过：“没有任何东西比在引入新的事物秩序的过程中领头更难于承担、更险于管理或对成功更具不确定性”。在 20 世纪的最后四分之一阶段，基于计算机的系统已经引入了新的秩序，虽然技术自 Machiavelli 的话问世以来已经有了巨大的发展，但他的话仍然是有用的。

软件工程是作为称为系统工程的过程的结果而发生的。不是仅仅着重于软件，系统工程关注于一系列元素，关注于如何按一个系统分析、设计和组织那些元素，该系统可以是针对信息变换或控制的产品、服务或技术。

当工程工作的前后相关环境着重于商业企业时，系统工程过程被称为信息工程，当一个产品[①]被建造时，该过程称为产品工程[②]。

信息工程和产品工程均试图为基于计算机的系统的开发带来秩序。虽然各自被应用于不同的应用领域，二者均努力将软件放置于前后相关的环境中，即，信息工程和产品工程均致力于分配计算机软件的角色，并建立结合软件和基于计算机的系统的其他元素之间的连接。

10.1 基于计算机的系统

词“系统”可能是在技术词典中使用最为过度和滥用的术语。我们谈论政治系统和教育系统、航空电子设备系统和制造系统、银行系统和地铁系统。这个词告知了我们很少的东西，我们使用“系统”的修饰词来理解该词所使用的语境。Webster 字典定义“系统”为“事物的集合或排列，被相互关联以使得形成一个联合或有机的整体，…一组事实、原理、规则等，被按有序的形式分类和排列以使得能够显示连接各个部分的逻辑设计图，…一种分类或排列的方法或计划，…一种确定的做某种事情、方法、规程的方式，…”在字典中还有 5 种其他定义，然而没有提出任何精确的同义词。“系统”是一个特殊的词。

借用 Webster 的定义，我们定义基于计算机的系统为：

元素的集合或排列，这些元素被组织在一起，以便通过处理信息完成某些预定义的目标。

其目标可能是支持某些业务功能或开发可被销售以产生业务收入的产品，为了达到该目标，基于计算机的系统使用一系列系统元素：

软件。计算机程序、数据结构和相关的文档，它们被用于实现所需的逻辑方法、规程或控制。

硬件。提供计算能力的电子设备和提供外部世界功能的电子机械设备(如，传感器、马达、抽水泵)。

人员。硬件和软件的用户和操作者。

数据库。通过软件访问的大型的有组织的信息集合。

文档。手册、表格和其他描述性信息，它们描绘系统的使用和/或操作。

规程。一序列步骤，定义每个系统元素的特定使用或系统驻留的过程性语境。

这些元素按不同的方式组合可构成不同的信息。例如，市场部门将原始的销售数据经组合转变成为典型的产品购买者能理解的图表，机器人将包含特定指令的命令文件变换为一组导致某些特定物理动作的控制信号。创建一个信息系统来帮助市场部门和控制软件来支持机器人均需要系统工程。

基于计算机的系统的一个复杂特征是：构成一个系统的一组元素可能也表示了某个更大的系统的宏元素，宏元素是一个基于计算机的系统，它是某更大的基于计算机的系统的一部分。例如，我们考虑“工厂自动化系统”，它实质上是如图 10－1 所示的层次结构。在结构的最低层，我们有数控机器、机器人和数据输入设备，每台设备自身均是基于计算机的系统。数控机器的元素包括电子的和电子机械的硬件(如，处理器和内存、马达、传感器)；软件(用于通信、机器控制

和内插校正)；人员(机器操纵者)；数据库(存储的 NC 程序)；以及文档和规程。类似的分解可应用于机器人和数据输入设备，每个均是基于计算机的系统。

在层次(图 10-1)的再上一层，定义了制造车间(manufacturing cell)，制造车间是一个基于计算机的系统，它可以有自己的元素(如计算机、固定设备)，并且也集成我们称为数控机器、机器人和数据输入设备的宏元素。

所以，制造车间和它的宏元素各自由一组通用标记的系统元素——软件、硬件、人员、数据库、规程和文档——构成。在某些情况下，宏元素可以共享这些通用元素，例如，机器人和 NC 机器可能均被单个操纵者管理(人员元素)。在其他情况下，这些通用元素是只对某系统的。

系统工程师的角色是在系统的整体层次(宏元素)的语境内为特定的基于计算机的系统定义一系列元素。在下面几节，我们检测组成计算机系统工程的任务。

10.2 系统工程层次结构

不管关注的领域是什么，系统工程包含一组自顶向下和自底向上的方法来导出如图 10-2 所示的层次结构。系统工程过程通常从“整体视图(world view)”开始，即，检查完整的业务或产品域以保证能够建立适当的业务或技术语境。精化整体视图，以更完全地关注特定的兴趣域，在特定的领域内，分析目标系统元素(如，数据、软件、硬件、人员)的需要，最后，开始分析、设计和构造目标系统元素。在结构的顶层，建立了一个非常广的语境，而在底层，进行导出由相关工程方法(如硬件或软件工程)完成的详细技术活动。[①]

以稍微形式化的方式来描述，整体视图(WV)包含若干个领域■(Di)，它们本身可以是一个系统或系统的系统。

$$WV= \{D_1, D_2, D_3, \cdots, D_n\}$$

每个领域由特定的元素(E_j)构成，每个元素代表了完成领域的实体和目标。

$$D_i= \{E_1, E_2, E_3, \cdots, E_m\}$$

最后，通过刻划每个元素是实现完成元素必要功能的技术构件。

$$E_j= \{C_1, C_2, C_3, \cdots, C_k\}$$

在软件语境内，构件应该是计算机程序、可复用的程序构件、模块、类或对象、或甚至是程序设计语言语句。

重要的是要注意，系统工程师当他或她沿上面描述的层次从上向下开展工作时，工作的关注区域越变越窄，然而，整体视图在适当的语境内清楚地描绘了整个功能的定义，从而使工程师能理解整个领域、并最终理解系统或产品。

10.2.1 系统建模

系统工程是一个建模过程，不管关注点是整体视图或详细视图，工程师创建模型，该模型可以［MOT92］：

- 定义可满足所考虑视图的需要的过程
- 表示过程的行为和行为基于的假设
- 显式地定义模型的外部(exogenous)和内部(endogenous)输入[①]
- 表示将使得工程师更好的理解视图的所有连接(包括输出)

为了构造系统模型，工程师应该考虑一系列控制因素：

1.假设能减少可能的排列和变化数量，这样使得模型以合理的方式反映问题。例如，考虑一个由娱乐产业用于创建现实动画的 3 维绘图产品，产品的一个领域是要能够表示 3D 人形，对该领域的输入包含从活“演员”、视频、或通过图形模型的创建来输入运动的能力。系统工程师给出某些关于可允许的人体运动范围的假设(如，腿不能卷曲在躯体上)，使得可限制输入的范围和处理。

2.简化，使得能够以及时的方式创建模型。为了说明问题，假设以一个办公产品公司为例，它销售和服务范围很宽，包括复印机、传真机和相关设备。系统工程师正在为服务组织的需要建模，并正在为理解产生服务工单的信息流而工作。虽然很多情况都可能产生服务工单需要，但工程师仅将它们分为两类：内部需要和外部请求，这使得可以很简单地划分生成工单所需的输入。

3.限制帮助定义系统边界。例如，正在为下一代飞机的一个飞机航空电子设备系统建模，因为飞机将设计为双引擎，故要为所有针对推动力的监控域建模，以便它们能够适应最多两个引擎，并有必要的相关联的冗余系统。

4.约束，将指导用什么方式来建立模型，以及用什么方法来实现模型。例如，上面描述的三维绘图系统的技术基础设施是基于 PowerPC 的处理器，因此，必须约束，问题的计算复杂性，以便该处理器的处理能力能够满足要求。

5.优先选择指明，所有数据、函数和技术优先选择的体系结构。优先选择的解决方案有时会和其他控制因素冲突，然而，客户的满意程度经常由优先选择方法的实现程度相匹配。

(在任意视图中)为了完全自动得到一个、或半自动得到一个解决方案、或手工导出一个方法，可能会调用二个已有的系统模型。事实上，用手工方式为某个问题选用另一个解决方案时，考查每种类型模型的特征经常是很有可能的，实质上，系统工程师只要简单地修改不同系统元素(人员、硬件、软件)的相关影响，从而导出每种类型的模型。

10.2.2 信息工程：概述

信息工程(IE)的目标是为业务能够有效地使用信息，而定义体系结构，此外，信息工程试图为实现那些体系结构创建一个整体计划［SPE93］。有三种不同的体系结构必须在业务目标的语境内加于分析和设计：

- 数据体系结构。
- 应用软件体系结构。
- 技术基础设施。

数据体系结构为业务或业务功能需要的信息提供框架，体系结构的个体建造块是被业务使用的数据对象(在第 12 章讨论)，数据对象在业务功能间流动，并被组织在数据库中，并被变换，以提供服务于业务需要的信息。而且在为了满足业务需要，提供服务时被不断变换。

应用软件体系结构包含那些为了某些业务目的而在数据体系结构中变换数据对象的系统元素，在本书的范围内，我们通常考虑的应用软件体系结构是指为完成这种变换而设计的程序(软件)系统，然而，在更广的范围内，应用软件体系结构可能包括人员(他们是信息变换者和用户)和尚未自动化的业务规程。

技术基础设施为数据和应用软件体系结构提供基础，这些基础设施包括用于支持应用软件和数据的硬件和软件，包括计算机和计算机网络、通信连接、存储技术和为实现这些技术而设计的体系结构(如，客户/服务器)。

为了给前面描述的系统体系结构建模，要定义一个信息工程活动层次结构，如图 10－3 所示，整体视图为信息策略计划(information strategy planning，ISP)[①]，ISP将整个的业务视为一个实体，并孤立成为整体企业具有重要性的业务领域(如，工程、制造、市场、财会、销售)。ISP定义企业级别中可见的数据对象、它们的关系、以及它们如何在业务领域间流动［MAR90］。

领域视图由称为业务域分析(business area analysis，BAA)的 IE 活动来组成，Hares［HAR93］以如下方式描述 BAA：

BAA 关注于详细地标识在 ISP 过程中定义的选定业务领域(domain)中的数据(以实体［数据对象］类型的形式)和功能需求(以处理的形式)，并确定它们的交互(以矩阵形式)，它仅仅以特定的方式来关注什么是业务域中所需要的。

当信息工程师开始 BAA 时，则焦点缩小到特定的业务领域。BAA 中将业务域视为一个实体，并孤立看待业务域能够满足其目标的业务功能和规程。BAA，和 ISP 一样，定义数据对象、它们的关系、以及数据如何流动，但是在这一层上，这些特征均被局限在被分析的业务域内。BAA 的结果是一个独立的机会区域，在该区域中信息系统可能支持业务领域。

一旦为了进一步的开发而独立出信息系统，IE 开始就转变为软件工程。通过启动业务系统设计(business system design，BSD)步骤，可为特定信息系统的基本需求建模，并且使这些需求映射到数据体系结构、应用软件体系结构和技术基础设施上去。

最后的 IE 步骤——构造和集成(construction and integration， C&I)主要关注实现细节。通过构造合适的数据库和内部数据结构、通过用程序构件建造应用软件、以及通过选择合适的技术基础设施元素来支持在 BSD 中创建的设计，可以实现体系结构和基础设施。然后必须集成每一个系统构件，以形成完整的信息系统或应用软件。集成活动也将新的信息系统放置到业务域语境内，完成所有用户培训和后勤支持，从而实现平稳转变。

10.2.3 产品工程：概述

产品工程的目标是将客户对一组已定义的能力的希望映射为可工作的产品[①]。为了达到这个目标，产品工程，和信息工程一样，必须导出体系结构和基础设施。体系结构包括四种不同的系统构件：软件、硬件、数据(和数据库)、和人员。建立一个支撑体系，并包括连接这些构

件在一起所需的技术和用于支持这些构件的信息(如，文档、CD－ROM，录象)。

如图 10－4 所示，整体视图由系统分析来完成。从客户处得到产品的整体需求，这些需求包括信息和控制需要、产品功能和行为、整体产品的性能、设计和界面约束，以及其他特殊需要。一旦知道了这些需求，系统分析的工作是给上面提到的四种构件的每一种分配功能和行为。

一旦分配完成，构件工程开始。构件工程实际上是一组并发活动，它们分别处理每一种系统构件，这些活动包括软件工程、硬件工程、人机工程和数据库工程。每一个这些工程方法是一个特定领域视图，但是，重要的是要注意，工程方法必须建立和保持与另一个方法之间的活跃通信，系统分析角色的一部分工作是建立该事将发生的接口机制。

产品工程的元素视图(element view)本身是被用来分配构件的工程方法，对软件工程而言，这实际是分析和设计建模活动(后面章节将详细讨论)以及包括代码生成、测试和支持步骤的构造和集成活动。分析建模将需求分解成数据、功能和行为，设计将分析模型映射成软件的数据、体系结构、界面和过程设计。

10.3 信息工程

当商业自动化在 60 年代早期被首次引入时，公司寻找机会区域，并简单地自动化那些以前用手工方式完成的业务功能。随时间流逝，单个的计算机程序被组合，来包含业务应用软件，这些应用软件被组合在一起成为服务于特定业务区域的大的信息系统。虽然该方法是可用的，但它产生了一些问题：很难完成，系

统的相互“连接”；到处留下冗余数据；很难计划对服务于某业务域的应用软件的修改影响，甚至难于实现修改；当旧程序已超越了使用期时，由于资源的缺乏，而导致它们将跨越它们的限制时期，而长期使用。

Hammer 和 Champy [HAM93] 在它们的书籍“reengineering the corporation”中提到：

信息技术在业务再工程中扮演了重要的角色，但是又很容易被不恰当的使用。现代的高级信息技术是任何再工程努力的一部分，是基本的使能者，它允许公司再工程其业务过程。但是，仅仅将计算机(和软件)投资于现存的业务问题并不会导致它被再工程。

信息工程的整个目标是将业务的整个需要的方式的最好服务应用于“信息技术”中，为了达到这个目标，IE 必须从分析业务目标开始，理解必须一起工作以完成这些目标的很多业务域、然后必须定义每个业务域以及整体业务的信息需求。仅仅在这些工作完成后，IE 才能转变为软件工程技术中要求最高的领域——信息系统、应用软件和程序被分析、设计和建造的过程。

10.4 信息策略计划

第一个信息工程步骤是信息策略计划(ISP)，ISP 的整体目标是：(1)定义战略性的业务目的与目标(objectives and goals)，(2)独立使业务能够达到这些目标的关键成功因素，(3)分析技术和自动化对这些目标的影响，(4)分析现存的信息，以确定其在达到目标中的所扮演角色。ISP 也创建业务级的数据模型，它定义了关键数据对象以及它们相互的关系及和各种业务域的关系。

术语“目的(objectives)”和“目标(goals)”在 ISP 中具有特定的意义，目的是对方向的一般性陈述，例如，蜂窝电话厂商的业务目的可能是减少产品的制造成本。而目标则定义定量的动作过程，为了达到上面提到的目的，制造商可能建立下面的目标：

- 在 9 个月内使不合格率下降 20%。
- 从供应商处获得 10%的价格让步。
- 对键盘进行再工程以减少 30%的组装成本。
- 使手工的部件组装自动化。
- 实现一个实时的生产控制系统。

目的往往是战略性的，而目标往往是战术性的。

关键成功因素(Critical success factors，CSF)可能与某个目的或单个目标有关，如果需要达到目的或目标，必须提出CSF，因此，管理计划必须与该CSF相适应。例如，对上面提到的制造目的的CSF可能是：

- 对制造组织的全部的质量管理策略。
- 对工人的培训和鼓励。
- 使用高可靠性的机器。
- 使用高品质的零件。
- 说服供应商降低价格的“销售计划”。
- 启用优质的工程人员。

技术影响分析检查目标，并指明哪些技术将对成功地完成目标有直接或间接的影响。信息工程师需回答下面的问题：技术对业务目标的完成是如何起关键性作用的？技术当前可用吗？技术将改变业务管理的方式吗？直接和间接的成本是什么？业务应该如何有效地修改或扩展目标，以适应技术的需要？

因为每个业务域使用不同的信息技术，ISP还必须标识当前存在的技术以及如何使用它可达到当前的目标。业务过程再工程(BPR)是一个活动，它检查现存的系统，试图再工程它们，以更好地满足业务需要。BPR在第27章讨论。

10.4.1 企业建模

企业建模创建业务的三维视图，第一维表达组织结构和在由组织结构定义的业务域中完成的功能；第二维分解业务功能以独立使功能发生的处理工作；最后，第三维将目的和目标以及CSF与组织及其功能相关联。此外，企业建模创建业务级的数据模型，该模型定义数据对象和它们与其他企业模型元素的关系。

用传统的业务单元层次结构(如，“组织图”)来定义业务组织(图10－5)，在组织图中的每个方框表示公司的一个业务域，和所有层次结构一样，通常有可能要精化组织图中的方框，直至分成为小的工作组或甚至某个个体，然而，根据ISP的目的，业务域是不可缺少的。

标识业务功能并且定义实现业务功能所需的处理，然后将每个业务功能与对其有责任的业务域(图10－5)相关联。通常，一个业务功能是某个正在进行的活动，必须完成它来支持整体业务，通常可将它描述为一个名词短语。一种业务处理是一个变换，它接收特定的输入并产生特定的输出，通常可将它描述为一个动词短语。

为了说明一个业务功能如何被精化为一组支持该功能的处理，以图10－5中显示的市场分析功能为例，其处理求精如下：

市场分析：

• 收集关于所有销售调查的数据。

• 收集关于所有销售的数据。

• 分析关于调查和销售的数据。

• 开发顾客图表。

• 比较图表和人口统计学的研究。

• 研究产业购买趋势。

• 建立重点小组，以确定最好的销售消息。

• 设计粗略的销售材料。

• 测试销售材料、并求精。

• 最后确定销售方法。

这里列出的每个处理步骤可以进一步精化，以提供完成业务功能所需的详细的计划图。

在 ISP 过程中，信息工程师并没有专门去关心存在自动化机会的区域，其意图仅仅是简单地理解业务并对其建模。

10.4.2 业务级数据建模

业务级数据建模[SCH92]是一个企业建模活动，它关注的是为完成在 10.4.1 节中提到的业务功能所需的数据对象(也称实体)。在业务级，典型的数据对象包括信息的生产者和消费者(如，客户)、事物(如，报告)、出现的事件(如，销售会议)、组织角色(如，工程副总裁)、组织单元(如，销售和市场部门)、位置 (如，制造车间)、或信息结构(如，雇员文件)。每个数据对象包含一组属性，它们定义了被描述对象的某些方面、质量、特征或被描述的数据描述体，例如，在企业建模过程中，信息工程师可能定义了数据对象 customer，为了更完善地描述 customer，定义了如下属性：

对象： Customer

属性：

名字

公司名

工作分类和购买权限

业务地址和联络信息

产品利润

过去购买史

上次联络的日期

联络状况

一旦定义了一组数据对象，就标识了它们的关系，关系指明对象间如何相互连接。例如，以对象customer、product A和salesperson为例，信息工程师创建了一张图(图 10－6)来描述这些关系[①]，根据该图，关系蕴含了数据对象间的连接。通常，可从两个方向理解关系，如，customer购买了product A和product A被customer购买了均可。在现实中，提供的其他信息是数据模型的一部分，但是，我们到第 12 章对它们再作讨论。

ISP 活动的顶点是创建一系列交叉引用矩阵，它们建立了组织(及其业务域)、业务目的和目标、业务功能、以及数据对象间的全面关系，这种矩阵的例子如图 10－7 显示。

10.5 业务域分析

Martin [MAR90] 在其关于信息工程的书中如此描述业务域分析：

业务域分析为建造基于信息的企业建立详细的框架，它一次处理一个业务域，并详细分析之。它使用图为企业中的数据和活动建模、并用矩阵记录企业中的数据和活动，且给出对企业中信息方面的精细和微妙的相互关联方式的清楚的说明。

在 BAA 中，我们的重点从整体视图移向领域视图，为了对“企业中信息方面的精细和微妙的相互关联方式”进行建模，信息工程师必须描述如何在每个业务域中使用和变换数据对象(在 ISP 中描述，并在 BAA 中精化)以及如何在每个业务域中的业务功能和处理中变换这些数据对象。实质上是，如何分析每个业务域的外部和内部数据并对其建模。

为了完成该工作，BAA 使用一系列不同的模型：

- 数据模型(现在被精化到业务域层)。

- 处理流模型。

• 处理分解图。

• 一系列交叉引用矩阵。

精化在 ISP 中定义的数据对象以便用于每个业务域中，例如，销售部门使用在前面节描述的数据对象 customer，在评估销售部门的需要后(销售域分析)，为满足销售的需要 customer 的初始定义被进一步精化：

对象：Customer

属性：

名字

公司名→对象：Company

工作分类和购买权限

业务地址和联络信息

产品利润

过去购买史

上次联络的日期→联络记录

联络状况→上次联络状况

→下次联络日期

→推荐的联络性质

属性公司名被修改为指向另一个称为 Company 的对象，该对象不仅包含公司名，而且包括公司规模、它的购买需求、其他的联络名等信息，这些信息在销售域中将是有用的。还修改和加入了其他属性。

10.5.1 处理建模

在业务域中完成的工作包括一组业务功能，它们被进一步精化为业务处理，为了阐明如何精化，以在 10.4.2 节讨论的销售功能的简化版本为例，为了完成销售而发生的处理是：

销售功能：

• 建立客户联系。

• 提供产品文献和相关信息。

• 解答问题和关心的内容。

• 提供评估用的产品。

• 接收销售定单。

• 检查订购配置的有效性。

• 准备可交付的订货。

• 和客户确认配置、价格和交货日期。

• 传送可交付的订货到执行部门。

• 跟踪客户。

可为这个处理序列开发处理流程图(如图 10－8 所示)。请注意，每个和业务域相关的业务功能均可以以类似的方式精化。

10.5.2 信息流建模

处理流模型和数据模型组合起来可以指明信息如何在业务域中流动。显示每个处理的输入和输出数据对象，指明处理如何变换信息(图 10－9)来完成业务功能。

一旦创建了一个完全的处理流模型集合，信息工程师(和其他人员一起)检查现存的处理过程可以如何被再工程(如文献[HAM93]和[JAY94])，以及现存信息系统或应用软件的什么地方可以被用更高效的信息技术来修改和替代。修订的处理模型被用作新的或修订的支持业务功能的软件的规约的基础。

在 BAA 中建立的领域视图作为业务系统设计、构造和集成的基础——这些 IE 步骤实际上是软件工程过程的一部分。这些步骤将在后面章节中讨论。

10.6 产品工程

产品工程(也称系统工程)是一个问题求解活动，揭示分析希望的产品数据、功能和行为，并分配给单个的工程构件。系统工程师从客户为产品定义的目标开始 进而以分配它们到一组工程构件——软件、硬件、数据(和数据库)和人员——的方式对这些需求建模，建立起这些构件与支撑基础设施——集成构件和被用于支持这些构件的信息(如文档、CD－ROM、录象)所需的技术——之间的连接。

大多数新产品和系统的起源从一个相当朦胧的希望功能的概念开始，因此，s 系统工程师必须通过标识希望的功能和性能范围来限制产品需求，例如，仅仅说在制造自动化系统中的机器人的控制软件对“在零件托盘空时快速反应”来说是不够的，系统工程师必须定义(1)什么时候指示机器人的零件托盘空，(2)以什么来限制期望的软件反应的精确时间(以秒计)，(3)必须说明采取什么形式的反应。即，系统工程师必须描述驱动机器人行为的事件、行为的性质、以及为行为设置的数量限制。

一旦限制了功能、性能、约束和接口，系统工程师开始转向任务分配，在分配中，将功能赋给一个或多个工程构件。通常会建议和评估可选择的分配，为了说明分配过程，我们以工厂自动化系统的一个宏元素为例——在第 5 章引入的传送带分类系统(CLSS)，下面是为系统工程师提出的 CLSS 目标的陈述(还稍有些朦胧)：

必须开发 CLSS，以使得将传送带上移动的盒子识别出来并分类传送到传送带末端的 6 个箱柜之一中去。盒子通过分类站，在这里它们被标识出来。基于打印在盒子旁边的标识号(以等价的条形码形式表示)，盒子将被分流到合适的箱柜中。盒子之间的顺序是随机的，并且相互间间距是相同的，传送带慢速移动。

在图 5－1 中用图形方式描述了 CLSS。在继续下一步工作前，如果你是系统工程师请列出一系列你将问的问题。

下面是一些很多将被问和答的问题：

1.必须处理多少不同的标识号？它们的形式是什么？

2.什么是传送带的速度(以每秒英尺计)？什么是盒子间的间距(以英尺计)？

3.分类站离箱柜有多远？

4.箱柜间相距多远？

5.如果盒子没有标识号或有错误的标识号，会发生什么？

6.当箱柜装满时，会发生什么？

7.是否关于盒子目的地和箱柜内容的信息将传送到工厂自动化系统的其他地方？是否需要获取实时的数据？

8.容许什么样的错误/失败率？

9.传送带系统的什么部分已经存在并在运行中？

10.有什么样的进度和预算约束？

注意，上面的问题着重于功能、性能、以及信息流和内容，系统工程师并不问客户任务如何完成，而是问需要什么。

假定回答的合理，系统工程师可以开发出一组可选择的分配方案。注意，功能和性能在每个分配方案中被赋予不同的常见系统元素。

分配方案 1。培训一个分类操纵者，并安置他工作在分类站位置上，他/她辨认盒子，并将盒子放入合适的箱柜。

分配方案 1 表示了对 CLSS 问题的纯粹的手工解决方案(但是，确实是奏效的)，主要的工程构件是人员(分类操纵者)，人完成所有的分类功能。可能需要某些文档(以表格的形式将标识号贴到箱柜上，以及针对操纵者培训的规程说明)，因此，这个分配仅仅有人员和文档元素。

分配方案 2。一个条形码阅读器被放置在分类站上，输出的条形三被传送给可编程控制器，它控制机械的分流机制，分流器将盒子滑动到合适的箱柜。

对分配方案 2，硬件(条形码阅读器、可编程控制器、分流硬件设备等)、软件(针对条形码阅读器和可编程控制器的)和数据库(将盒子 ID 和箱柜位置相对应的查找表)构件被用于提供实现自动化操作。有可能每个构件有对应的手册和其他文档，这是另一个构件。

分配方案 3。条形码阅读器和控制器被放置在分类站上，输出的条形码被传送给机器人手臂，它抓住盒子，并将盒子放入合适的箱柜中。

分配方案 3 使用了一个宏元素——机器人，和分配方案 2 一样，该分配方案使用了硬件、软件、数据库和文档作为工程构件，机器人是 CLSS 的宏元素，并且本身包含一组工程构件。

通过检查三个可选择的 CLSS 分配方案，显然，相同的功能可以分配给不同的构件。为了选择最有效的分配方案，应该对方案选择应用一组权衡标准。

下面的权衡标准用来对基于特定的常用系统元素的功能和性能而产生的分配方案的选择：

项目考虑。该分配方案的配置可以在预先确定的成本和进度限制内被建立吗？与成本和进度估算相关的风险是什么？

业务考虑。该方案的配置代表了最有利可图的解决方案吗？它在市场上能成功吗？最后的回报能够证明开发风险是值得的吗？

技术分析。开发所有系统元素的所需的技术存在吗？功能和性能能否有保证？方案配置可以恰当地保持吗？技术资源存在吗？和技术相关的风险是什么？

制造评估。制造设施和装备可用吗？缺少必要的构件吗？质量保证能够适当地完成吗？

人的问题。对开发和制造有可用的培训过的人员吗？是否存在政治问题？客户理解将要完成什么样的系统吗？

环境接口。建议的方案配置和系统的外部环境有合适的接口吗？是否机器和机器、人和机器间的通信以智能的方式来处理？

法律的考虑。该方案配置会引入不适当的责任风险吗？产权方面可以被适当地保护吗？是否存在潜在的侵权。

我们将在本章后部更详细地对这些问题中的某些进行讨论。

重要的是要注意，系统工程师应该考虑对客户问题的可购买的解决方案，是否已经有等价的系统存在？解决方案的大部分部件可以从第三方购买吗？

权衡标准的应用软件将导致特定系统配置及分配对硬件、软件(和固件)、人员、数据库、文档和规程的功能和性能规约的选择，实质上，功能和性能的内容被分配给产品的每个工程构件，硬件工程、软件工程、人机工程和数据库工程的角色是精化该内容，并产生可运行的与其他构件适当地集成使用的产品构件。

10.6.1 系统分析

系统分析的进行应该将下面的目标置于脑中：(1)说明客户的需要；(2)评估系统概念的可行性；(3)完成经济和技术分析；(4)分配功能到硬件、软件、人员、数据库、和其他系统元素；(5)建立成本和进度约束；(6)创建形成所有后续工程工作基础的系统定义。为了成功地达到上面的目标，需要专门的硬件和软件技术(以及人机和数据库工程)。

10.6.2 说明客户需要

系统分析过程的第一步涉及对需要的说明。分析员(系统工程师)会见客户和终端用户(如果用户不同于客户)，客户可能是外面公司、分析员所在公司的市场部门(当定义产品时)、或另一个技术部门(当内部系统将被开发时)的代表，和信息工程一样，其意图是了解产品的目的，并定义满足该目的所需的具体目标。

一旦确定了全部目标，分析员转向补充信息的评估：建造系统的技术是否存在？将需要什么特殊的开发和制造资源？对成本和进度有什么限制？如果新系统实际上是一个为向很多客户销售而开发的产品，还应该问下面的问题：产品的潜在市场是什么？该产品与竞争产品的比较情况如何？在公司的整个产品线中该产品占据什么位置？

在系统概念文档中说明在需要标识步骤时收集的信息，初始的概念文档有时是由客户在会见分析员前准备的，客户—分析员通信将总会导致对文档的修订。

10.6.3 可行性研究

所有项目均是可行的——给定无限的资源和无限的时间！不幸的是，基于计算机的系统或产品的开发是更可能受资源不足和困难的交付时间(如果不是完全现实的)的折磨，在尽可能早的时间评估项目的可行性既是必要的也是应该的，如果在定义阶段较早地识别出一个错误构思的系统，那么，可以避免成月或成年的工作量、数千或百万美元的投资、以及数不清的专业困窘。

可行性和风险分析是与很多方式相关的，如果项目风险过大(由于在第 6 章讨论的任意理由)，生产高质量软件的可行性将会降低。然而，在产品工程阶段，我们主要对 4 个主要利润域更加关注：

经济可行性。评估开发成本相对于最后的从开发的系统或产品获得的收入或收益。

技术可行性。研究可能影响完成一个可接受系统的能力的功能、性能和约束。

法律可行性。确定可能产生自系统开发的任何侵权、妨碍或责任。

可选择性。评估系统或产品开发的其他可选方法。

可行性研究，不保证系统的经济回报是明显的、技术风险是低的、几乎不预测法律问题、以及不考虑选择的合理性，但是，如果任何前提条件失败，就应该对该域进行研究。

经济回报通常是大多数系统考虑的“底线”(有时只有国防系统、法律授权的系统、和如空间程序等高技术应用的领域才有例外存在)[①]，经济回报考虑的非常广泛，如，成本—收益分析、长期的公司收入策略、对其他利润中心或产品的影响、开发所需的资源成本、以及潜在的市场增长。

技术可行性经常是在产品工程过程阶段最难于评估的区域，因为目标、功能和性能均稍微有些模糊的，如果“正确的”假设被给定，任何事似乎都是可能的。分析和定义的过程与技术可行性评估并行进行是很重要的，采用这种方式，可以对具体的规约在它们被确定时加于判断。

通常和技术可行性相关的该考虑问题包括：

开发风险。可以设计的系统元素，能在使得必要的功能和性能在分析中揭示的约束的范围内完成吗？

资源可用性。有熟练的职员可用于系统元素的开发吗？其他必要的资源(软件和硬件)对建造系统可用吗？

技术。相关的技术已经进展到可以支持系统的状态吗？

基于计算机的系统的开发者本性是乐天派(其他有什么人能有勇气敢于去尝试我们经常承担的事情？)，然而，在技术可行性评估阶段，应该保持批评的态度，因为在此阶段的错误判断将是灾难性的。

法律可行性要考虑的范围也是很广泛的，它们包括合同、责任、侵权、和技术人员不知道的无数其他的陷阱，关于法律问题和软件的讨论已超越本书范围，有兴趣的读者可参见文献［SC089］。

对可选择性所考虑的程度经常受成本和时间约束所限制，然而，不应该忽视合理的方案变体。

可行性研究可以文档化为独立的递交给高层管理者的报告，并且可作为系统规约的附录。虽然可行性研究的格式可能变化，但下面提供的大纲覆盖了大多数主要话题。

Ⅰ.引言 A.可选择的系统配置

A.问题的说明 B.用于选择最后方法的标准

B.实现环境Ⅳ.系统描述

C.约束 A.内容的简略说明

Ⅱ.管理总结和见意 B.被分配元素的可行性

A.重要的发现Ⅴ.成本收益分析

B.注释Ⅵ.技术风险评估

C.见意Ⅶ.法律问题

D.影响 Ⅷ.其他项目特定的话题

Ⅲ.可选择性

可行性研究首先由项目管理层复审(评估内容可靠性)和高层管理者复审(评估项目状况)，研究将导致一个“进行/不进行”的决策。应该注意，其他的进行/不进行决策将在硬件和软件工程的计划、规约、和开发阶段给出。

10.6.4 经济分析

包含在可行性研究中的最重要的信息之一是成本—收益分析——对基于计算机的系统项目的经济回报的评估。成本—收益分析描绘项目开发的成本，并将它们和系统的实际的(即，可直接用美元测度的)和无形的收益相比较。

成本—收益分析由于一些评估标准而复杂化，这些标准随将被开发的系统的特征、项目的相对规模、以及作为公司的战略计划的一部分希望的投资期望回报变化而发生变化。此外，很多从基于计算机的系统导出的收益是无形的(如，通过迭代优化而带来的更好的设计质量、通过可编程的控制而增加的客户满意程度、以及通过重新格式化和预先分析的销售数据而带来的更好的业务决策)，直接的数量比较可能难于实现。

正如我们上面提到，收益的分析将依赖于系统特征而产生差异。为了说明该问题，以表 10—1 中显示的管理信息系统 [KIN78] 的收益为例。大多数数据处理系统的开发是以“更好的信息数量、质量、合时、或组织”作为主要目标，因此，在表 10-1 中提到的收益着重于信息访问和它对用户环境的影响。与工程-科学分析程序或基于计算机的产品相关的收益可能会有实质性的差异。

表 10－1 可能的信息系统收益［KIN78］

来自对计算和打印任务所做努力的收益
在计算和打印方面的每单位成本的降低(CR)
改善的计算任务精度(ER)
在计算程序中快速改变变量和值的能力(IF)
在计算和打印方面极大增加的速度(IS)
来自对记录保持任务所做努力的收益
从记录“自动地”收集和存储数据的能力(CR、IS、ER)
更完整的和系统化的记录保持(CR、ER)
在空间和成本方面对记录保持增加的容量(CR)
记录保持的标准化(CR、IS)
每个记录可以存储的数据量的增加(CR、IS)
改善的记录存储安全性(ER、CR、MC)
改善的记录可移植性(IF、CR、IS)
来自对记录搜索任务所做努力的收益
记录的快速检索(IS)
改善的从大型数据库访问记录的能力(IF)
改善的在数据库中修改记录的能力(IF、CR)
通过远程通信连接需要搜索的地点的能力(IF、IS)
改善的创建被访问的记录的记录的能力(ER、MC)
审计和分析记录搜索活动的能力(MC、ER)
来自对系统重构能力所做努力的收益
同时修改全部记录类的能力(IS、IF、CR)
移动大批数据文件的能力(IS、IF)
通过合并其他文件的某些方面而创建新文件的能力(IS、IF)
来自对分析和仿真能力所做努力的收益
快速完成复杂的、同时计算的能力(IS、IF、ER)
创建复杂现象的仿真以回答“是什么？”问题的能力(MC、IF)
聚合大量的对计划和决策有用的数据的能力(MC、IF)
来自对过程和资源控制所做努力的收益
在过程和资源控制方面需要减少的工作量(CR)
改善的“微调”过程(如组装线)的能力(CR、MC、IS、ER)
改善的保持对资源的连续监控的能力(MC、ER、IF)

注：CR=成本降低或避免；ER=错误减少；IF=增加的灵活性；IS=增加的活动速度；MC=管理计划和控制的改善。

和基于计算机的系统的开发相关的成本［KIN78］被列在表 10-2 中，分析员可以估算每个成计划和控制的本，然后使用开发和运行成本来确定投资回报、收支平衡点、以及偿还期。

表 10-2 可能的信息系统成本

获得成本
咨询成本
实际的设备购买或租赁成本
设备安装成本
装修设备地点的成本(空调、安全等)
资金的成本
涉及获取过程的管理和人员的成本
启动成本
操作系统软件成本
通信设备安装成本(电话线、数据线等)
启动人员成本
人员搜索和雇佣活动的成本
对组织的其余部分中断的成本
指导启动活动所需的管理成本
项目相关的成本
应用软件购买成本
修改软件以符合本地系统的成本
来自内部应用开发的人员、一般管理费用等的成本在应用软件使用
中培训用户人员的成本
数据收集和安装数据集过程的成本
准备文档的成本
开发管理成本
运行成本
系统维护成本(硬件、软件和设施)
租赁成本(电力、电话等)
硬件折旧成本
涉及信息系统管理、运行和计划活动的人员的成本

下面的摘录 [FRI77] 可能最恰当地说明了成本—收益分析：

和选举后的政治许诺一样，成本—收益分析可能在项目实现开始后被忘记，然而，它是极端重要的，因为它是得到管理批准的手段。

仅仅通过花费时间去评估可行性，我们减少了在系统项目的后期阶段极端困窘(或更糟)的机会，对建议的项目被取消而花费在可行性分析上的工作量并不是浪费。

10.6.5 技术分析

在技术分析中，分析员评估系统概念的技术优点，同时收集关于性能、可靠性、易维护性和生产率的附加信息，在某些情形下，该系统分析步骤也包括有限数量的研究和设计。

技术分析从对系统的技术生存力的建议评估开始，要完成系统功能和性能需要什么技术？需要什么新材料、方法、算法或处理，并且它们的开发风险是什么？这些技术问题将如何影响成本？

技术分析可用的工具可从数学建模和优化技术、概率和统计、排队论和控制论中导出[①]，然而，重要的是要注意，并不总是可以进行的分析评估。建模(数学的或物理的)是对基于计算机的系统的有效的技术分析机制。

Blanchard 和 Fabrycky［BLA81］，定义了在系统的技术分析中为模型使用的一组标准：

1.模型应该能动态地表示系统配置的评估，这种配置要求很容易理解和操纵、并且与现实操作足够接近的方式产生结果。

2.模型应该关注那些和手边的问题最相关的因素，并且抑制(通过判断)那些不很重要的因素。

3.模型应该尽可能全面的包括所有相关的因素，并且应该使结果的可重复性尽可能可靠。

4.模型设计应该足够简单，以允许在问题求解中适时的实现。除非分析员或管理者可以以适时和高效的方式使用模型，否则它是没有价值的。如果模型很大，且高度复杂，就可能需要开发一系列更小的模型(其中一个模型的输出当成另一个模型的输入)。而且，有可能希望独立于其他元素来评估系统的某特定元素。

5.模型设计应该将易于修改和/或扩展的能力相结合，以允许需要时对其他附加因素的评估。在整体目标被满足前成功的模型开发经常包括一系列试验，最初的意途主要是忽视一些不被立即注意的数据，随后再去将它们加上，从中获得更高的效益。

从技术分析得到的结果形成了关于系统的另一个“进行/不进行”决策的基础，如果技术风险很严重，如果模型指明希望的功能或性能不能被完成，如果各部分将不能被平稳地结合在一起——它将回到最初状态，从头开始！

10.7 系统体系结构建模

每个基于计算机的系统可用输入-处理(加工)-输出的体系结构来为信息变换建模，Hatley 和 Pirbhai［HAT87］为了包括两个附加的系统特性——用户界面处理以及维护和自测试处理——扩展了该视图。虽然没有对每个基于计算机的系统提出这些附加的特性，但是，它们是非常常见的，并且它们的规约使得系统

模型更健壮。使用输入、处理、输出、用户界面处理和自测试处理的表示方法，系统工程师可以创建一个系统构件模型，从而建立了在每种工程方法中后期的需求分析和设计步骤的基础。

为了开发系统模型，要使用一个体系结构模板［HAT87］。系统工程师给模板内的5个处理区域的每一个分配系统元素：(1)用户界面，(2)输入，(3)系统功能和控制，(4)输出，(5)维护和自测试。体系结构模板的格式如图10－10所示。

与用于系统和软件工程的几乎所有建模技术一样，体系结构模板使得分析员能够创建详细的层次结构。体系结构语境图(ACD)驻留于层次的最顶层。

语境图“建立了实现的系统和系统将运行的环境间的信息边界”［HAT87］，即，ACD定义了系统使用信息的所有外部生产者、系统创建信息的所有外部消费者、以及所有通过界面通信或完成维护和自测试的实体。

为了说明ACD的使用，以本章前面讨论的传送带分类系统(CLSS)的扩展版本为例，该扩展版本在分类站位置使用PC机。PC机执行所有CLSS软件，和条形码阅读器接口，以读取盒子上的零件号，和传送带监控设备接口，以获取传送带速度，存储所有已分类的零件号，和分类站操纵者交互，以产生一系列报告和诊断，发送控制信号给分流硬件，以对盒子进行分类，以及和中央工厂自动化主机通信。扩展CLSS的ACD如图10-11所示。

显示在图10-11中的每个方框表示一个外部实体——即，来自系统的信息的生产者或消费者，例如，条码阅读器产生信息，它是CLSS系统的输入。完整系统的符号(或在较低层的主要子系统)是具有圆角的矩形，因此，CLSS被表示在ACD中部的处理和控制区域。在ACD中显示的标记箭头表示在外部环境和CLSS系统间流动的信息(数据和控制)。外部实体条码阅读器产生标记为条码的输入信息，在本质上，ACD将任何系统放置于其外部环境的语境中。

系统工程师通过更详细地考虑图10－11中的阴影矩形，而精化体系结构语境图。使得传送带分类系统能够在ACD定义的语境中标识主要运行子系统，在图10－12中，主要子系统被定义在体系结构流程图(architecture flow diagram，AFD)中，它是从ACD导出的。跨越ACD的区域的信息流用于指导系统工程师开发AFD——对CLSS的更详细的图形表示，体系结构流程图显示了主要的子系统和重要的信息(数据和控制)的流程。此外，体系结构模板将处理子系统划分为前面讨论的5个处理区域，在此阶段，每个子系统可包含一个或多个系统元素(如，硬件、软件、人员)，到底包含谁，由系统工程师分配。

初始体系结构流程图(AFD)变成AFD层次的顶层结点，在最初的AFD中的每个圆角矩形可被扩展为另一个专门描述它的体系结构模板。这个过程如图10－13所示，系统的每个AFD可被用作来描述子系统的后续工程步骤的起始点。

子系统和在它们间流动的信息可被用来指定后续工程的工作(的限制)，每个子系统的说明性描述和在子系统间流动的所有数据的定义变成了系统规约的重要元素。

10.8 系统建模和仿真

大约 30 年以前，R.M.Graham [GRA69] 对我们建造基于计算机的系统的方式给出了一个悲观的评论：“我们建造系统就象 Wright 兄弟建造飞机一样——建造整个事物，将它推出悬崖，让它坠毁，而后又从头开始”。事实上，至少有一类系统——受刺激有反应的系统，我们今天仍在这么做。

很多基于计算机的系统和现实世界以一种受刺激有反应的方式交互，即现实世界事件由构成基于计算机的系统的硬件和软件监控，并且基于这些事件，系统实施对机器、过程、以及甚至导致事件发生的人员的控制。实时和嵌入式系统经常归类为受激反应系统范畴。

不幸的是，受激反应系统的开发者有时努力去使它们能适当地工作，直至最近，在建造它们之前要预测这样的系统的性能、效率、和行为仍是很困难的。作为非常真实的感受，很多实时系统的构造是一次“飞行”的冒险，直至系统被建造完并被“推出悬崖”，令人震惊不已的现象(大多数是不愉快的)才被发现。如果系统由于不正确的功能、不合适的行为或很差的性能而“坠毁”，我们只好捡起碎片，又从头开始。

最高级体系结构流程图

在受激反应范畴的很多系统是用来控制机器和/或过程(如，商业航班或石油精炼厂)，它们必须以极其高的可靠性运行，如果系统失败，就可能造成巨大的经济或人员损失。为此，Graham 描述的这类问题是痛苦的和危险的。

今天，针对系统建模和仿真的 CASE 工具正被用于帮助消除当建造受激反应的基于计算机的系统时的不良后痪。这些工具被用在系统工程过程中，且由工具来指定硬件和软件的角色、数据库和人员。建模和仿真工具使得系统工程师能够去“测试驱动”系统的规约。如何才能来测试驱动的技术细节和特殊建模技术在第 29 章有概略地讨论。

10.9 系统规约

系统规约是一份文档，它是硬件工程、软件工程、数据库工程和人机工程的基础。它描述了基于计算机的系统的功能和性能以及将支配其开发的约束。规约限制了每个分配的系统元素，例如，它为软件工程师指明了软件在系统的整体语境内的角色，并且指明了在体系结构流程图中描述的不同的子系统，。系统规约也描述了系统输入和输出的信息(数据和控制)。

下面是推荐了一个系统规约的大纲，然而，应该注意，这仅仅是很多可被用于定义系统描述文档的大纲之一，实际的格式和内容可以由软件或系统工程标准或本地的习惯和喜好规定。

Ⅰ. 引言

A.文档的范围和目的

B.概述

1.目标

2.约束

Ⅱ.功能和数据描述

A.系统体系结构

1.体系结构语境图

2.ACD 描述

Ⅲ.子系统描述

A.子系统 N 的体系结构图规约

1.体系结构流程图

2.系统模块叙述

3.性能问题

4.设计约束

5.系统构件分配

B.体系结构字典

C.体系结构互连图和描述

Ⅳ.系统建模和仿真结果

A.用于仿真的系统模型

B.仿真结果

C.特殊性能问题

Ⅴ.项目问题

A.计划的开发成本

B.计划的进度

Ⅵ.附录

10.10 小结

高技术系统包含一系列构成成分(构件)：软件、硬件、人员、数据库、文档和过程。系统工程帮助将客户的需要映射到使用一个或多个这些构件的系统。

系统工程从“整体视图”开始，分析业务域或产品，以建立所有的基本需求。然后，关注点缩小到“领域视图”，这里单独地分析每个系统元素，给每个元素分配给一个或多个工程构件，它们然后由相关的工程方法处理。

信息工程是一种系统工程方法，它被用来定义使业务能够有效地的使用信息的体系结构。信息工程的意图是导出全面的数据体系结构、应用体系结构和技术基础设施，它们将一起满足业务策略的需要和每个业务域的目的和目标。信息工程包括信息策略计划(ISP)、业务域分析(BAA)、以及特定应用的分析(它实际上是软件工程的一部分)。

产品工程是一种从系统分析开始的系统工程方法，系统工程师标识客户的需要、确定经济和技术可行性、以及分配功能和性能到软件、硬件、人员和数据库——这些是关键的工程构件。生成系统或产品的体系结构模型，并且开发每个主要子系统的表示，最后，系统工程师可以创建受激反应系统模型，它可被用作性能和行为仿真的基础。系统工程任务以系统规约的创建为最后的终结，系统规约是形成后续的所有工程工作的基础文档。

系统工程要求在客户和信息或系统工程师间的密切通信，客户必须了解系统目标，并能够清楚地说清它们，工程师必须知道问什么问题、给什么建议、以及做什么研究。如果通信成功，并且创建完整的系统模型，则建立了系统构造的坚固基石。

思考题

10.1 尽你的可能，找出词“系统”的尽可能多的单个同义词。

10.2 针对大型系统建造类似于如图 10－1 所示的“系统的系统”(不能使用图中的例子)，你的层次应该沿至少一个“树”分支向下扩展到简单的系统元素(硬件、软件等)。

10.3 选择你熟悉的任何大型系统或产品，定义描述系统或产品的整体视图的一组领域，描述构成一个或两个领域的元素集合，对一个元素，标识必须开发的技术构件。

10.4 选择你熟悉的任何大型系统或产品，陈述为了建造有效的(和可实现的)系统模型而必须给出的假设、简单要求、限制、约束和优先选择。

10.5 信息工程试图定义数据和应用软件体系结构以及技术基础设施，描述这些术语的含义，并给出一个例子。

10.6 信息策略计划从目的和目标定义开始，给出来自业务域的例子。

10.7 你已经决定开展计算机软件的邮购业务，因为想高效地运行业务，你决定进行信息工程。请从 ISP 开始，建造简单的企业模型，包括组织图、业务功能和业务处理的大纲、以及业务级的数据模型。

10.8 让我们假定你对软件邮购业务(思考题 10.7)标识的业务域之一是电话订货处理，对该业务域进行 BAA，以开发更详细的数据模型和处理流程图。

10.9 系统工程师可来自三种人员之一：系统开发者、客户或某些外部组织，讨论每种人员的优势和劣势，描述“理想的系统工程师”。

10.10 对 10.6 节中的 CLSS 系统的列表加入至少 5 个其他附加问题，并提出两种其他的 CLSS 系统分配方案。

10.11 你的老师将分别描述某基于计算机的系统或产品的高层结构。

a.给出一组你作为系统工程师应该问的问题。

b.基于老师对你的问题的回答，建议至少两种系统的分配方案。

c.在班级上，比较你的分配方案和其他同学的方案。

10.12 开发一个当系统或产品的“可行性”将被评估时考虑的属性的检查表，讨论属性间的相互影响，并提供一种分级方法，使得开发出可量化的“可行性数”。

10.13 研究用于基于计算机的系统(它将需要某些硬件制造和组装)的详细的成本-收益分析的记帐技术，写出一个技术管理者可应用的指南的“食谱”集。

10.14 为你选择基于计算机的系统(或你老师赋予的题目)开发体系结构语境图(ACD)和体系结构流程图(AFD)。

10.15 撰写系统模块说明，它包括在为思考题 10.14 开发的 AFD 中定义的一个或多个子系统的体系结构图规约中。

10.16 研究关于CASE工具的文献，并撰写一篇概略性论文，描述建模和仿真工具是如何工作的。(或者，收集来自两个或多个销售建模和仿真工具的CASE厂商的文献，并评估相似处和不同处。)

10.17 基于你老师提供的文档，为下面的基于计算机的系统之一开发简略的系统规约：

a.非线性的、数字视频编辑系统

b.个人计算机的数字扫描仪

c.电子邮件系统

d.大学注册系统

e.Internet访问提供商

f.交互式酒店预定系统

g.本地感兴趣的系统

确定已经创建了描述在10.8节中的体系结构模型。

10.18 存在不能在系统工程活动中建立的系统特征吗？描述这些特征，并解释为什么对它们的考虑必须延迟到后面的工程步骤。

10.19 存在形式化的系统规约可以被简略或完全排除的情况吗？解释之。

推荐阅读文献及其他信息源

Martin(Information Engineering，3 volumes，Prentice－Hall，1989，1990，1991)对信息工程话题进行了全面讨论。Hares[HAR93]、Spewak[SPE93]，以及Flynn和Fragoso－Diaz(InformationModeling：An Interntaional Perspective，Prentice-Hall，1996)等书籍也详细地讨论了该主题。

因为它是交叉学科话题，产品工程是一个可能的主题。Armstong和Sage(Introduction toSystems Engineering，Wiley，1997)，Martin(Systems Engineering Guidebook，CRC Press，1996)、Wymore(Model－Based Systems Engineering，CRC Press，1993)、Lacy(System EngineeringManagement，McGraw－Hill，1992)、Aslaksen和Belcher(Systems Engineering，Prentice－Hall，1992)、Athey(Systematic System Approach，Prentice－Hall，1982)，以及Blanchard和Fabrycky[BLA81]等书籍讨论了系统工程过程(具有各自不同的工程强调)并提供了有价值的指南。

Thayer 和 Dorfman 的优秀的 IEEE 教程(System and Software Requirements Engineering，IEEEComputer Society Press，1990)讨论了在系统和软件级别需求分析问题间的关系，同一作者的另一部相关书籍(Standards，Guidelines and Examples：System and Software RequirementsEngineering，IEEE Computer Society Press，1990)对分析工作的标准和指南进行了全面讨论。

Robertson 和 Robertson(Complete Systems Analysis,Dorset House,1994)、Silver 和 Silver(Systems Analysis and Design，Addison—Wesley，1989)，Modell(A Professiona′ s Guide toSystems Analysis，McGraw—Hill，1988)，以及 McMenamin 和 Palmer(Essential Systems Analysis，Yourdon Press，1984)提供了对用于信息系统世界的系统分析任务的有用的讨论，每个讨论包含了对实例的研究，阐明了在系统分析中要面对的问题、方法、和解决方案。

对那些活跃地涉及到系统工作或对更高级的话题感兴趣的读者来说，Gerald Weinberg 的书(An Introduction to General System Thinking，Wiley—Interscience，1976，和 on the Design of stableSystems，Wiley—Interscience，1979)已经变成经典文献，并对“一般系统”思维(general systemsthinking)”(它隐式地导向了一般性的系统分析和设计方法)给出了很好的讨论。Weinberg 最近的书籍(General Principles of Systems Design，Dorset House,1988,和 Rethinking Systems Analysisand Design,Dorset House，1988)继承了他早期工作的传统。

可在下面网址上得到关于“系统科学”信息的其他有关资料：

http：//www.sea.uni—linz.ac.at

下面的 WWW 网址讨论了系统和信息工程的研究和实际应用：

系统理论和信息工程：

http：//www.cast.uni—linz.ac.at/st/

信息工程初步：

http：//www2.echo.lu/programmes/en/fact_sheets/elpub2001.html

系统工程主页：

http：//rs712b.gsfc.nasa.gov/704/704home.html

关于信息工程和产品工程的WWW文献最新消息可在http：//www.rspa.com找到。

① 在此，产品一词包括从单部电话到一种高技术系统。

② 实际中，于此常常用到“系统工程”一词，但，在本书中，“系统工程”一词是通用的，它包括信息工程和产品工程。

① 但是在有些情况下，系统工程首先考虑的是单个系统元素和/或详细的需求，用这种方法，通过首先考

虑子系统的详细构件的样式从而自低向上地描述了子系统。

① 外部输入与用同级别或其他级别中的其他元素给出的视图的一个元素有关，而内部输入则与特定视图中的元素的单个构件有关。

① 应该注意到图 10－3 中的技术术语词与一般的文学上的休词方法不同，但是，通过所有提到的主题可以找到每个IE活动暗示的区域。

① 在某种高技术系统的开发中也用到产品工程［如航空交通控制系统］。

① 该图，称为整体关系模型，在第 12 章将有详细描述。

① 这些方面现在也在改变，以便使政府“预算”能够缩小。今天，任何系统都应该考虑投资回报。

① 称为原型和仿真工具的许多CASE工具可以来辅助完成大部分的技术分析，第 28 章将讨论这些工具。

第 11 章　分析概念和原则

对软件需求的完全理解对软件开发工作的成功是至关重要的，不论我们设计得如何好、编码得如何好，未很好地分析和编写的程序将只会给用户带来失望，并给开发者带来烦恼。

需求分析任务是发现、求精、建模和规约的过程。包括详细地精化初始由系统工程师建立并在软件项目计划中精化的软件范围，创建所需数据、信息和控制流以及操作行为的模型，此外还有分析可选择的解决方案，并将它们分配到各软件元素中去。

在需求分析和规约中，软件开发者和客户均扮演了积极的角色。客户必须尽力将有时有些模糊的软件功能和性能概念重新具体详细地描述出来，而开发者则是软件功能的询问者、咨询顾问和问题解决者。

需求分析和规约看起来可能是相当简单的任务，但其实并不是这样。用户与开发者之间需要通信的内容非常大，其中可能会存在错误解释或误传的可能性，或含糊性。软件工程师可能要面临的进退两难的局面可通过重复某客户的陈述而得到最好的理解：“我知道你认为你已经理解了我所说的内容，但是我并不能肯定你已认识到你所听到的并不是我所想要的。”

11.1 需求分析

需求分析是一种软件工程活动，它在系统级软件分配和软件设计间起到桥梁的作用(如图 11－1 所示)。需求分析使得系统工程师能够刻划出软件的功能和性能、指明软件和其他系统元素的接口、并建立软件必须满足的约束。需求分析允许软件工程师(在这种角色中经常称为分析员)精化软件分解模块，并建造将被软件处理的数据、功能、和行为模型。需求分析为软件设计者提供了可被翻译成数据、体系结构、界面和过程设计的模型，最后，需求规约为开发者和客户提供了软件建造完后质量评估的依据。

软件需求分析可被划分成 5 个工作阶段：(1)问题分析，(2)问题评估和方案综合，(3)建模，(4)规约，和(5)复审。

初始时，分析员研究系统规约(如果存在的话)和软件项目计划，并在系统语境内理解软件和复审，从而生成计划软件范围的估算。接着，必须建立针对分析的相互通信方式，以使得问题分析得到保证。分析员的目标是对用户/客户认识到的基本问题要素进行识别。

问题评估和方案综合是分析工作的下一个主要关注点，分析员必须定义所有外部可观察到的数据对象，评估信息流和内容；定义并详细阐述所有软件功能；在影响系统的事件的语境内理解软件行为；建立系统界面特征；以及揭示其他设计约束。这些任务中的每一个旨在描述问题，以便可以综合出全面的方法或解决方案。

例如，汽车零件的主要供应商需要一个库存控制系统，分析员发现与当前的手工系统相关的问题包括：(1)不能快速地获得部件的状况；(2)更新卡片文件需要 2 或 3 天的工作量；(3)由于没有办法查找相关厂商的部件信息而使得对同一厂商同一货品多次再订货，等等。一旦问题被标识出来，分析员将确定新系统该产生什么信息，以及将提供什么信息给系统[①]，例如，客户希望得到指明什么零件被从库存中取出以及还剩余多少相似零件的日报表。客户指明一旦当该零件离开仓库时库存管理员就该记载每个零件的标号。

通过对当前问题和希望的信息(输入和输出)进行的评估，分析员开始综合一个或多个解决方案。为了便于开始，必须详细地定义系统的数据、处理功能和行为。一旦已经建立这些信息，就该考虑针对实现的基本体系结构。客户/服务器方法似乎是合适的，但是，它确实属于在软件计划中概括的范围吗？似乎需要一个数据库管理系统，但是，该数据库系统真的是用户/客户需要的吗？继续评估和综合的过程，直至分析员和客户均确信针对后面的开发步骤软件确实已被适当地刻划了。

贯穿整个评估和综合过程，分析员的主要焦点是“什么(what)”，而不是“怎么做(how)”，系统会产生和使用什么数据？系统必须完成什么功能？将定义什么界面？会应用什么约束？等。[②]

在问题评估和综合解决方案的活动中，分析员创建系统模型，以便可以更好地理解数据和控制流、处理功能和操作行为、以及信息内容。模型是软件设计的基础，也是创建软件规约的基础。

重要的是要注意，在本阶段要得到详细的规约是不可能的。客户可能并不能精确地肯定需要什么，开发者可能不能肯定可用什么特定的方法来适当地完成功能和性能。由于这些以及许多其他理由，可以采用一种可选择的需求分析方法——原型法，(第 2 章)，本章后面我们将讨论原型法。

11.2 通信技术

软件需求分析中的相互通信总是要在两方或多方间进行。客户有某个问题需要基于计算机的解决方案，开发者要要针对客户的请求提供帮助，这时便产生了

通信需求。但是，正如我们已经提到，从相关通信、交流到理解的道路经常是充满坎坷的。

11.2.1 过程的启动

客户和开发者之间最常用的交流方式以及开始相互通信过程的技术是进行预备会议或访谈。在软件工程师(分析员)和客户间的第一次会议可与两个青年间的第一次约会的笨拙程度相比较，没有人知道该说什么或问什么；双方均担心他们所说的会被误解；双方均在考虑最终谈话将导向何处(双方在这里可能有完全不同的期望)；双方均希望能控制事情的进程；而同时，双方均希望能够获得成功。

然而，相互交流活动必须被启动。Gause 和 Weinberg [GAU89] 建议分析员从寻问一组无关的问题(context free questions)开始比较好，即，首选该问问问题的性质、需要解决方案的人数及能力、希望的解决方案的性质，以及第一次可能遭遇什么问题。第一组问及的问题主要是关注客户、整体目标和收益，例如，分析员可能问：

- 谁是本工作的最初请求者？
- 谁将使用该解决方案？
- 成功的解决方案的经济收益是什么？
- 存在另一个需要解决的问题吗？

下一组问题该使得分析员能够对问题更好的理解，并使得客户能够表达其关于解决方案的感觉：

- 如何刻画将由某成功的解决方案所产生的“好的”输出？
- 该解决方案强调了什么问题？
- 能向我显示(或描述)解决方案所应用的环境吗？
- 存在影响解决方案的特殊性能问题或约束吗？

最后一组问题该关注于会议的效果。Gause 和 Weinberg [GAU89] 将它们称为元问题，并给出了下面的简略问题列表：

- 你是回答这些的问题的合适人员吗？你的回答是“正式的”吗？
- 我的提问和你想解决的问题相关吗？
- 我是否问了太多的问题？

• 还有其他人员可以提供附加信息吗？

• 还有其他我应该问你的问题吗？

这些问题(和其他问题)将帮助“打破坚冰”，并启动对成功的分析至关重要的相互通信活动，但是，提问和回答的会议形式并不是一定会取得成功的方法，事实上，Q&A 会议应该仅仅用于第一次相遇的时候，然后该用包含问题求解、商议和规约等活动的会议形式来取代。在下节中将给出这种会议形式。

11.2.2 便利的应用规约技术

客户和软件工程师经常有无意识的“我们和他们”的区分，不是按工作的需要来将一支队伍标识和精化，而是各自定义自己的“版图”并通过一系列备忘录、正式的意见书、文档以及提问和回答会议来相互通信。历史已经证明，这种方法不能很好地奏效，会产生大量的误解、忽略重要的信息、以及无法建立成功的工作关系。

正是由于这些问题的存在，一系列独立的研究者开发了一种面向团队的需求收集方法，该方法被应用在分析和规约的早期阶段，被称为便利的应用规约技术(FAST)，该方法鼓励建立客户和开发者队伍之间的合作，他们共同工作来标识问题、提出解决方案的要素、商议不同的方法、以及刻划出初步的解决方案需求［ZAH90］。今天，FAST 已经成为信息系统界使用的主流技术，该技术为改善所有各种应用中的相互通信提供了潜在可能。

已经提出了很多不同的FAST方法[①]，各自使用的场景稍有不同，但是，所有方法均是在下面的基本原则之上进行某些修改的：

• 在中立的地点举行会议，由开发者和客户出席。

• 建立准备和参与会议的规则。

• 建议一个足够正式的议程以便可以进行所有重要点的，而又是足够非正式的，鼓励思维的自由交流。

• 一个“协调者”(他可以是客户、开发者或其他外人)控制会议。

• 使用一种“定义机制”(它可以是工作表、图表、墙上胶黏纸或墙板)。

• 目标是标识问题、提出解决方案的要素、商议不同的方法、以及在有利于完成目标的氛围中刻划出初步的解决方案需求。

为了更好地理解典型的 FAST 会议中所发生的事件流，我们提出了一个概略的场景，该场景概述了作为会议所准备的事件序列、在会议中发生的事件序列、以及在会议之后的事件序列。

当举行了开发者和客户间的初始会议(11.2.1 节)并且基本的提问和回答帮助建立了问题的范围和解决方案的整体感觉。在初始会议之外，开发者和客户要写下一页或者二页的“产品请求”，选择 FAST 的会议地点、时间和日期，并选举协调者。邀请来自开发方和客户方组织的双方代表出席会议，并在会议日之前将产品请求发布给所有的与会者。

当在会议前复审请求时，要求每个 FAST 出席者列出一组环绕系统环境的一部分对象、产生将系统的其他对象、以及完成系统功能的对象。此外，要求每个出席者列出另一个服务(处理或功能)列表，这些服务操纵对象或与对象的交互。最后，开发出约束列表(如，成本、规模大小、重量)和性能标准列表(如，速度、精度)。通知出席者：不期望这些列表是穷尽的，但是，希望每套表反应的是每个人对系统的感觉。

例如[①]，假定对消费产品公司工作的FAST团队提供了下面的产品描述：

我们的研究表明，家庭安全系统的市场正以每年 40%的比率增长，我们希望能进入该市场，并试图建造基于微处理器的家庭安全系统，该系统将保护和/或识别一系列出人意料之外的“情况”，如非法进入、火警、水灾或其他。该产品，暂时称为 SafeHome，产品将使用合适的传感器来检测各种情况，具体使用时房主可按需要编程,并且当系统检测到情况时,会自动地给监控机构拨打电话。

现实中，在本阶段将提供大量更多的信息，但是，即使有了附加的信息，仍存在问题的含糊性、仍存在问题被忽略的可能、并且仍可能发生错误。就现在而言，上面的“产品描述”已是足够了。

FAST 团队由来自市场、软件和硬件工程以及制造方的代表组成，并选择外来人员作为协调者。

FAST 团队(图 11-2)的每个人给出上面产品描述的列表。为 SafeHome 描述的对象可能包括：若干烟雾检测器、若干窗口和门传感器、若干运动检测器、一个警报器、一个事件(启动某传感器)、一个控制模板、一个显示器、一串电话号码、一次电话拨号等等。服务的列表可能包括：设置警报器、监控传感器、电话拨号、控制面板编程、以及读显示器(注意，服务作用于对象)。以类似的方式，每个 FAST 出席者将开发约束列表(如，系统的制造成本必须低于 200 万美元，界面必须是友好的，以及必须是和标准电话线接口)和性能标准列表(如，应该在一秒钟之内识别传感器事件，应该实现事件优先级模式)。

在提出某话题的单个列表后，团队将创建一个组合的列表，该组合列表将删去冗余项，并加入在表达过程中出现的新思想，但是不删除任何其他东西。在建好所有话题的组合列表后，开始讨论活动(有协调者主持)。缩短、加长或重新措词组合列表以适当地反应将被开发的产品/系统。讨论活动的目标是要为每个话题(对象、服务、约束和性能)开发出一个意见一致的列表，这些列表然后被用于后面的活动中。

一旦创建了意见一致的列表，该将团队分为更小的小组，每个小组力图为每个列表中的一个或多个项开发出小型规约，小规约是对包含在列表中的单词或短语的精细化。例如，SafeHome 中的对象“控制面板”的小规约可能是：

- 安装在墙纸上。
- 大小大约为 9×5 英寸。
- 包含标准的 12 键键盘和特殊键。
- 包含 LCD 显示，操作如草图所示(未在此给出)。
- 所有的客户交互通过键盘发生，键盘被用于启动或关闭系统。
- 连接提供交互指南、回显等的软件到所有的传感器。

每个小组然后将他们开发的每个小规约提交给所有的 FAST 出席者讨论，进行添加、删除或进一步的精化等工作，在某些情形中，小规约的开发将可能发现新的对象、服务、约束或性能需求，它们将被添加到原列表中。在所有讨论过程中，团队可能提出某些不能在会议过程中解决的问题，此时要保留问题列表以使得这些思想在以后的活动中产生作用。

在小规约完成后，每个 FAST 的出席者提出一个针对产品/系统的确切标准列表，并将该列表提交给团队，然后创建一个意见一致的确定标准列表，最后，一个或多个参与者(或外来者)被赋予撰写完整的规约草案的任务(用来自 FAST 会议的结果为输入)。

FAST 并不是解决在早期需求收集阶段遇到的问题的万能药，但是，团队方法提供了集中很多不同观点的好处、即时的讨论和求精、以及具体的规约开发步骤。

11.2.3 质量功能部署

质量功能部署(QFD)是一种质量管理技术，它将客户的需要翻译为软件的技术需求。该方法初始在日本开发，并于 70 年代早期首先被用于 Kobe Shipyard of Mitsubishi Heavy Industrues，Ltd。QFD“集中来最大限度地让客户满意”［ZUL92］，为了达到这个目标，QFD 强调理解什么是对客户有价值的，然后在整个工程活动中部署这些价值。

QFD 标识三类需求［ZUL92］：

正常的需求。在与客户的会谈中，所陈述的针对某产品或系统的目标。如果提出这些需求，则将满足客户需要。正常需求的例子可能是：要求按图形方式显示特定的系统功能、以及自定义的性能级别。

期望的需求。这些需求对产品或系统是隐式的，并且可能是非常基础的，以致于客户没有显式地陈述它们。它们的缺席将导致很多不满意。期望的需求的例子有：人机交互的容易性、整体的操作正确性和可靠性、以及软件安装的容易性。

兴奋的需求。这些特征在客户的期望范围之外，并且当其存在时将是非常令人满意的。例如，字处理软件要求标准的特征，所交付的产品包含一系列页面布局能力，这将是令人愉快的和出人意料的。

在现实中，QFD 贯穿于整个工程过程［AKA90］，然而，很多 QFD 概念可用于软件工程师需求分析的早期阶段来面对与客户的相互通信问题，下面的段落中，我们仅仅给出这些概念(适用于计算机软件)的概述。

在和客户的会谈中，功能部署被用于确定系统所需的每个功能的价值；信息部署标识系统必须使用和产生的数据对象和事件，这些数据对象和事件和功能有联系；最后，任务部署在合适的语境内检查系统或产品的行为；价值分析确定在上面提到的三个部署中确定的需求的相对优先级别。

QFD 使用客户访谈和观察、调查、以及历史数据(如，问题报告)的检查来作为需求收集活动的原始数据，这些数据然后被翻译为需求表——客户意见表(customer voice table)——需要和客户一起复审该表。然后一系列图表、矩阵、以及评估方法被用于抽取期望的需求，并力图导出令人感兴趣的需求［BOS91］。

11.3 分析原则

在过去 20 年，研究者已经标出了若干分析问题和它们的原因，并开发了一系列建模符号体系和对应的启发规则来克服这些问题，每个分析方法有独特的观点，然而，所有分析方法与一组操作原则相关联：

1.必须表示和理解问题的信息域。

2.必须定义软件将完成的功能。

3.必须表示软件的行为(作为外部事件的结果)。

4.必须划分描述信息、功能和行为的模型，从而使得可以以层次的方式揭示细节。

5.分析过程应该从要素信息移向细节实现。

通过应用这些原则，分析员系统地处理某问题。检查信息域以使得功能可以被更完整地理解，使用模型以使得可以以简洁的方式交流功能和行为的特征，应用划分以减少问题的复杂性。在这些处理过程中软件的要素和视图实现对适当由处理需求带来的逻辑约束和由其他系统元素带来的物理约束是必需的。

除了上面提到的操作性分析原则，Davis [DAV95a]，建议了一组针对“需求工程”的指导性原则：[①]

• 在开始建立分析模型前先理解问题。人们通常总存在急于求成的倾向，甚至在问题被很好地理解前，这经常会导致产生一个解决错误问题的优美软件的诞生。

• 开发原型，使得用户能够了解将如何发生人机交互。因为人们一般对软件质量的感觉经常基于对界面“友好性”的感觉，因此，强力推荐使用原型方法(以及相应产生的迭代)。

• 记录每个需求的起源及原因。这是建立回溯到客户的可追踪性的第一步。

• 使用多个需求视图。建立数据、功能和行为模型，为软件工程师提供三种不同的视图，这将减少忽视某些东西的可能性，并增加识别不一致性的可能性。

• 给需求赋予优先级。过短的时限可能使每个软件需求得于实现的可能性减小，如果采用增量过程模型(第 2 章)，必须标识那些将在第一个增量中要交付的需求。

• 努力删除含糊性。因为大多数需求以自然语言描述，存在含糊性的可能，正式的技术复审是发现并删除含糊性的一种方法。

软件工程师将这些原则牢记在心，则有可能开发出为软件设计提供优秀基础的软件规约。

11.3.1 信息域

所有的软件应用程序均可被称为数据处理。有趣的是，这个术语包含了一个我们理解软件需求的关键性含义，软件被建造来处理数据，将数据从一种形式变换为另一种形式，即接收输入，以某种方式操作之，并产生输出。这个基本的策略始终是不变的陈述，不管我们是开发针对工资系统的批处理软件，还是开发控制汽车发动机油流的实时嵌入式软件。

然而，重要的是要注意，软件也处理事件，一个事件表示了系统控制的某些方面，实质上仅仅是一个布尔值数据——或者开或者关、或者真或者假、或者存在或者不存在。例如，一个压力传感器检测压力是否超过安全值，并发送警报到监控软件，警报信号是一个事件，该事件控制系统的行为，因此，数据(数值、字符、图象、声音等)和控制(事件)二者均驻留于问题的信息域内。

第一条操作性分析原则需要对信息域进行检查，信息域包含三个不同的数据和控制视图，每个视图被一个计算机程序处理：(1)信息内容和关系，(2)信息流，(3)信息结构。为了完全理解信息域，应该考虑每个视图。

信息内容表示了单个数据和控制对象，它们构成了某个更大的由软件变换的信息集合。例如，数据对象“工资”是一组重要数据体的组合：领款人的姓名、净付款数、付款总额、扣除额等等，因此，创建“工资”的内容以确定它所需的属性定义。类似地，控制对象“系统状态”的内容可能由一个位串定义，每个位表示一个单独的信息项，它指明是否某特殊的设备是在线或是离线。

数据和控制对象可和其他的数据和控制对象相关联，例如，数据对象“工资”和对象“时间卡、雇主、银行”及其他对象有一个或多个关系，在信息域的分析过程中，应该定义这些关系。

信息流表示了数据和控制在系统中流动时的变化方式，如图 11-3 所示，输入对象被变换为中间信息(数据和/或控制)，它们被进一步变换为输出。沿着这个(或这些)变换路径，可能从现存的数据存储(如，磁盘文件或内存缓冲区)中引入附加的信息。数据的变换是程序必须完成的功能或子功能，在两个变换(功能)间流动的数据和控制定义了每个功能的接口。

信息结构表示了各种数据和控制项的内部组织，数据或控制项将被组织为 n 维表还是层次树形结构？在结构的语境内，什么信息是和其他信息相关的？所有的信息是包含在单个结构中，还是使用不同的结构？在某信息结构中的信息如何和在另一个结构中的信息相关？通过对信息结构的估价可回答这些问题以及其他问题，应该注意，数据结构(在本书后面讨论的一个相关概念)指明了信息结构的设计和实现。

11.3.2 建模

我们创建模型可获得对将要建造的实际实体的更好的理解。当实体是物理的事物时(如建筑物、飞机、机器)，我们可以建造在形式和形状上相同、但在规模上要小的模型，然而，当所建造的实体是软件时，我们的模型必须采用不同的形式，从而使它们能够对软件变换的信息进行建模、对变换发生的功能(和子功能)进行建模、以及对变换发生时的系统行为进行建模。

在软件需求分析阶段，我们创建将要建造的系统的模型，这些模型着重于描述系统必须做什么、而不是如何去做系统。在很多情形下，我们使用图形符号体系创建模型，将信息、处理、系统行为和其他特征描述为不同的、可识别的符号，模型的其他部分可能是完全文字的，可使用自然语言或某特殊的专用于描述需求的语言来提供信息描述。

第二条和第三条操作性分析原则需要我们建立功能和行为模型。

功能模型。软件变换信息，为了达到此目标必须至少完成三个常见功能：输入、处理和输出。当创建应用的功能模型时，软件工程师只关注于问题特定的功能。功能模型从单个的语境层模型(即，将被建造的软件的名字)开始，经过一系列的迭代，将提供越来越多的功能细节，直至得到所有系统功能的完全描绘。

行为模型。大多数软件对来自外界的事件做出反应，这种刺激-反应特征构成了行为模型的基础。一个计算机程序总是存在于某个状态——一种外部可观测到的行为模式(如，等待、计算、打印、投票)，仅当某事件发生时才被改变。例如，软件将保持等待状态直至(1)某内部时钟指明某个时间段已经过去，(2)某外部事件(如，鼠标移动)产生一个中断，或(3)某外部系统通知该软件以某种方式动作。行为模型创建了软件状态的表示，以及导致软件状态变化的事件的表示。

在需求分析阶段创建的模型扮演了一系列重要的角色：

• 模型帮助分析员理解系统的信息、功能和行为，因此，使得需求分析任务更容易实现结果更系统化。

• 模型变成了复审的焦点，因此，也成为确定规约的完整性、一致性和精确性的重要依据。

• 模型变成了设计的基础，为设计者提供了软件要素的表示视图，该表示可被转化到实现的语境中去。

在第 12 和 20 章讨论的分析方法实际上是建模方法，虽然使用的建模方法经常取决于个人(或组织)的喜爱，但对好的分析工作而言，建模活动是最基础的行为。

11.3.3 划分

问题经常太大而且复杂，以致于难于进行整体理解，为此，我们往往将这样的问题划分(partition)为易于理解的子问题，并建立各子问题间的接口以使得可以完成整个的功能。第四条操作性分析原则建议划分软件的信息、功能和行为域。

在本质上，划分将问题分解为其构成成分。在概念上，我们建立信息或功能的层次表示，然后划分最上层的元素，通过(1)在层次上垂直向下移动而显露更多的细节，或(2)在层次上水平移动而分解问题。为了阐明这些划分方法，让我们重新考虑描述在 11.2.2 节的 SafeHome 安全系统，SafeHome 的软件划分(作为系统工程和 FAST 活动的结果)描述如下：

SafeHome 软件使得房主能够在安装时自行设置安全系统，监控所有和安全系统连接的传感器，以及通过包含在 SafeHome 中的控制面板(如图 11-4 所示)的键盘和功能键和房主进行信息交互。

在安装过程中，SafeHome 控制面板被用来“编程”和配置系统，每个传感器被赋予一个编号和类型，编写主人密码以启动和关闭系统，当传感器事件发生时将输入电话号码进行拨号。

当某传感器事件被识别出时，软件激活一个附于系统上的可发声的警报，在一定的延迟时间后(由房主在系统配置活动中指定)，软件拨出监控服务的电话号码并报告关于位置和被检测到的事件的性质等信息，电话号码将每 20 秒重拨一次，直至电话接通。

和 SafeHome 的所有交互由用户交互子系统管理，该子系统读入通过键盘和功能键提供的输入，在 LCD 显示屏上显示提示信息和系统状况。键盘交互采用下面的形式…。

SafeHome 软件的需求可以通过划分产品的信息、功能和行为域而分析，为了阐明问题，可划分问题的功能域，图 11-5 给出了 SafeHome 软件的水平分解，通过在功能层次中水平地移动，表出 SafeHome 软件的功能构成成分，并将问题分解，三个主要的功能记录在层次的第一层。

和某主要的 SafeHome 功能相关联的子功能可以通过在层次中垂直地显示出细节而得于检查，如图 11-6 所示。沿着监视传感器功能下的单条路径向下移动，产生垂直的划分，以显示出增加的功能细节的级别。

我们应用于 SafeHome 功能的划分方法也可以应用于信息域和行为域，事实上，信息流和系统行为的划分(在第 12 章讨论)将提供对系统需求的额外观察。当问题被划分时，将导出功能间的接口，跨接口移动的数据和控制项应该至少包括用于完成所陈述的功能所需的输入以及其他功能或系统元素所需的输出。

11.3.4 基本视图和实现视图

软件需求的基本视图(essential view)给出了将要完成的功能和将要处理的信息，而不管实现细节。例如，SafeHome 功能“读监视器状态”的基本视图本身并不关心数据的物理形式或使用的传感器的类型，事实上，可以说“读状态”将是本功能更合适的名字，因为它根本不考虑关于输入机制的细节。类似地，在本阶段可表示数据项“电话号码”(由功能“拨电话号码”暗含)的基本数据模型，而不管用于实现该数据项的底层数据结构。在需求分析的早期阶段，通过将注意力着重集中于问题的基本要求，我们可以在后期的需求规约和软件设计中对刻划细节的实现留下更多的选择。

软件需求的实现视图给出了处理功能和信息结构的现实世界表示下，在某些情形下，在软件设计中，物理表示的开发被作为第一步，然而，大多数基于计算机的系统通过规定适应某些实现细节的方式来刻划。SafeHome 的一个输入设备是边界传感器(不是看门狗、警卫人员、或陷阱)，该传感器通过感知电路的中断而检测进入是否非法，该传感器的一般性特征应该作为软件需求规约的一部分被记录，分析员必须识别由于预定义的系统元素(该传感器)而带来的约束，并在合适的时候考虑功能和信息的实现视图。

我们已经提到，软件需求分析应该着重于软件将完成什么，而不是处理将如何实现，然而，实现视图不一定被解释为表示“如何做”，而是表示实现模型操

作的当前模式，即对所有系统元素现存的或建议性的划分。(功能或数据的)基本模型是一种，明显地指明功能实现的基本模型。

11.4 软件原型

应该不管应用的软件工程范型而进行软件分析，然而，分析采用的形式是不同的，在某些情形下，有可能应用操作性分析原则，并导出可从该模型开发设计出的软件模型。在其他情况下，为客户和开发者的评估而构造的需求收集(通过FAST、QFD、或其他“大脑风暴”技术[JOR89])被进行，分析原则被应用，以及软件模型被建造——称为原型。最后，存在一种需要在分析开始时构造原型的环境，因为该模型是唯一的可以有效地导出需求的手段，该模型然后演化为产品软件。

Boar[BOA84]以如下方式评判原型技术：

当前大多数推荐的定义业务系统需求的方法被设计为：在系统被设计、构造、被用户看见或试用前，建立一个最终的、完整的、一致的和正确的需求集合。常见的和多次的产业经验表明，尽管使用了严格的技术，在很多情况下用户仍因为最终产品的不正确性和不完整性而拒绝应用它，结果是导致需要进行昂贵的、耗时的重新工作，以协调初始的规约和实际操作需求的最后测试，在最坏的情形下，不是重新修订交付的系统，而是将系统丢弃。开发者可以针对规约进行系统的建造和测试，但用户是根据当前的和实际的操作现实而确定是接收或是拒绝使用系统。

虽然上面的叙述代表了一种极端的观点，它的基本论据却是正确的，在很多(但不是所有)情形下，原型的构造，可能结合系统化的分析方法，是一种有效的软件工程方法。

11.4.1 选择原型方法

原型范型可以是封闭结束的或开放结束的，封闭结束的方法经常称为丢弃型原型方法，使用该方法，原型仅仅粗略展示需求，然后原型被丢弃，再使用不同的范型来开发软件。开放结束的方法称为演化型原型方法，使用原型作为继续进入设计和构造的分析活动的第一部分，软件的原型是最终系统的第一次演化。

在选择封闭结束的或开放结束的方法之前，有必要确定是否将要建造的系统是否适合于原型方法，可定义一系列候选原型方法的因素[BOA84]：软件应用领域、软件应用复杂性、客户特征、以及项目特征。[①]

通常，任何要创建动态可视显示、和人员用户有很多的交互、或要求必须以演化方式开发的算法或组合处理等的应用软件是原型方法的候选者，然而，这些软件应用领域必须针对应用软件复杂性进行权衡，如果一个候选的应用软件(它具有上面提到的特征)在任何演示功能被完成前需要开发数万行代码，那么有可

能采用原型方法会太复杂[②]。然而，如果可以分解复杂性，仍有可能对软件的各部分采用原型方法。

因为客户必须在稍后的步骤中和原型交互信息，因此，如下两个基本点是：(1)用原型评估和精化的客户资源，(2)能够以即时的方式作出需求决策的客户。最后，开发项目的性质将对原型方法的功效有很大影响，项目管理愿意并能够使用原型方法吗？有原型工具可用吗？开发者有使用原型方法的经验吗？

Andriole[AND92]建议了一组 6 个问题，它们将有助于原型方法的选择：

- 客户已和开发者已很好地理解将被建造的软件的应用域了吗？
- 要求解的问题支持建模吗？
- 客户能清楚地确定基本的系统需求吗？
- 需求被很好地建立并有可能相当稳定吗？
- 有任何需求是含糊的吗？
- 在需求中存在矛盾吗？

表 11－1　选择适当的原型方法

问题	丢弃型原型法	演化型原型法	需要的其他预备性工作
应用域已被建模码？	是	是	否
问题可被建模吗？	是	是	否
客户能确定基本需求吗？	是/否	是/否	否
需求已被建立并稳定吗？	否	是	是
有任何需求是含糊的吗？	是	否	是
需求中有矛盾吗？	是	否	是

11.4.2 原型方法和工具

为了使软件原型方法有效，必须快速地开发原型以使得客户可以评估结果和建议做修改，为了导出快速原型，三个基本的方法和工具类(如参考文献[AND92]和[TAN89])是可用的：第四代技术、可复用软件构件、形式化规约和原型环境。

第四代技术。第四代技术(4GT)包括广泛的数据库查询和报表语言、程序和应用软件生成器以及其他非常高级的非过程语言。因为 4GT 使得软件工程师能够快速地生成可执行代码，因此，它们是理想的快速原型工具。

可复用软件构件。另一种快速原型方法是使用一组现存的软件构件来装配，而不是构造原型。一个软件构件可以是数据结构(或数据库)或软件体系结构构件

(即程序)或过程构件(即模块)，在每种情形中，必须设计软件构件以使得它可以在不知道其内部工作细节的条件下被复用。

混合型方法和程序构件复用将只有在一个可使得存在的构件可以被分类和检索的库系统已被开发的情况下才可以有效地工作，虽然一系列工具已经被开发来满足该需要(如参考文献[ARN87])，但在此领域仍有大量工作需要做。

应该注意，现存的软件产品可以被用作“新的、改进的”竞争性产品的原型，在某种意义下，这也是一种软件原型复用的形式。

形式化规约和原型环境。在过去 20 年里，已经开发了一系列形式化规约语言和工具，它们是自然语言规约技术的替代品，今天，这些形式化语言的开发者正在开发交互式的环境，该环境(1)使得分析员能够交互地创建基于语言的系统或软件规约，(2)激活自动工具将基于语言的规约翻译为可执行代码，(3)使得客户能够使用原型可执行代码去精化形式化需求。

11.5 规约

毫无疑问，规约的模式与解决方案的质量有很大关系，曾经被迫使用不完整的、不一致的或易误解的规约进行工作的开发者，均曾经历过由此产生的失败和混乱，这是以软件的质量、时间和完整性为代价的。

11.5.1 规约原则

规约可被视为一个表示过程，不管我们完成规约的模式有多么不同，需求以最终导向成功的软件实现的方式来表示。Balzer 和 Goldman 给出了如下的一系列规约原则：

1.从实现中分离出功能性

2.开发一个系统希望的行为模型，该模型包含了系统对来自环境的各种刺激的数据和功能反应。

3.建立软件操作的语境，通过它刻划其他系统构件和软件交互的方式。

4.定义系统运作的环境，并指明“一组高度缠绕在一起的代理如何对环境中由其他代理产生的刺激(对对象的变化)作出反应”。

5.创建认知模型而不是设计或实现模型，该认知模型按用户感觉系统的方式来描述系统。

6.认识清楚“规约必定是不完整的和可增加的”。规约总是某个通常相当复杂的现实的(或想象的)情形的一个模型——一个抽象，因此，它将是不完整思维，并将存在多个细节层次。

7.建立规约的内容和结构，并使得它将能够适应未来的变化。

上面给出的这个基本规约原则的列表提供了表示软件需求的基础，然而，原则必须被翻译为实现，在下一节，我们将给出一组创建需求规约的指南。

11.5.2 表示

图 11-7 是一个好规约的经典例子，该图来自于 Galileo 的著作(大约在 1638)，在著作中该例子用于描述横梁的力度分析的文字说明，即使在没有伴随的说明文字的情况下，该图也能帮助我们理解必须做什么。

我们已经看到，软件需求可以用一系列方式来刻划，然而，如果需求被用纸张或电子表示媒体提交(它们几乎总是应该如此)，一系列的简单指南是值得注意的：

表示格式和内容应该和问题相关。可以为软件需求规约的内容开发一般性的大纲，然而，包含在规约中的表示形式有可能随软件应用领域的变化而发生变化，例如，制造自动化系统的规约将使用和程序设计语言编译器的规约不同的符号、图和语言。

包含在规约中的信息应该是嵌套的。表示应该展示信息的层次性以使得读者能够转移到需要的细节级别，段落和图的编号模式应该指明它所表示的细节层次，有时，值得在不同的抽象层次中表示相同的信息，以辅助理解规约。

图和其他符号应该在数量上有所限制，并在使用上一致。混乱的或不一致的符号体系，不管是图形的还是符号的，均会妨碍理解，并导致错误。

表示应该是可修订的。规约的内容将会被修改，理想地情况下，CASE 工具应该可用于更新被每个修改所影响的所有表示。

研究者们对和规约相关联的人的因素已经进行了很多研究(如参考文献[HOL95]和[CUR85])，毫无疑问符号体系和符号排列将影响人们对问题的理解，然而，软件工程师似乎都有自己对特定符号和图形的喜好，熟悉经常是人的喜好的根源，但是，其他更切实的因素如空间排列、易于识别的模式、以及形式化程度经常主宰了对所用符号的选择。

11.5.3 软件需求规约

软件需求规约是分析任务的最终产物，作为系统工程一部分的，且分配给软件的功能和性能通过建立完整的信息描述、详细的功能和行为描述、性能需求和设计约束的说明、合适的校验校准、以及其他和需求相关的数据而被精化。国家标准局、IEEE(标准号 830－1984)以及美国防御部门均已经提出了软件需求规约

(以及其他软件工程文档)的候选格式，然而，为了达到我们的目的，表 11－2 给出的简化大纲可被用作为规约的框架。

表 11－2　软件需求规约的框架

Ⅰ.引言	4.设计约束口
A.系统参考文献	5.支撑图
B.整体描述	C.控制描述
C.软件项目约束	1.控制规约
Ⅱ.信息描述	2.设计约束
A.信息内容表示	Ⅳ.行为描述
B.信息流表示	A.系统状态
1.数据流	B.事件和动作
2.控制流	Ⅴ.校验和校准
Ⅲ.功能描述	A.性能范围
A.功能划分	B.测试的类
B.功能描述	C.期望的软件响应
1.处理说明	D.特殊的考虑
2.限制/局限	Ⅵ.参考书目
3.性能需求	Ⅶ.附录

“引言”陈述的软件目标，可在基于计算机的系统语境内进行描述。实际上，引言仅仅是计划文档中关于软件范围的描述。

“信息描述”给出软件必须解决的问题的详细描述，并记录了信息内容和关系、流和结构，此外针对外部系统元素和内部软件功能描述了硬件、软件和人机界面。

解决问题所需的每个功能的描述在“功能描述”中给出，这其中为每个功能说明了一个处理过程；叙述并证明了设计约束；叙述了性能特征；以及用一个或多个图形来图形地表示的软件的整体结构和软件功能与其他系统元素间的相互影响。规约的“行为描述”部分检查作为外部事件和内部产生的控制特征的软件的操作。

可能最重要，但是又是具有讽刺意义的是，软件需求规约中最经常被忽视的内容是“校验标准”，我们如何识别是实现成功的？为了校验功能、性能和约束，必须进行什么类型的测试？我们忽视这些内容是因为要完成它需要对软件需求的全面理解——有时这是我们在本阶段做不到的事。然而，校验标准的规约实际上是对其他需求的隐式复审。将时间和注意力集中到该内容是非常重要的。

最后，软件需求规约包括“参考书目”和“附录”，参考书目包含了对所有和该软件相关的文档的引用，这些内容包括其他的软件工程文档、技术参考文献、厂商文献以及标准。附录包含了规约的补充信息，表格数据、算法的详细描述、图表以及其他材料均在附录中给出。

在很多情形下，软件需求规约可能伴随有可执行的原型(它在某些情况下可以替代规约)、纸上原型、或初步的用户手册。初步的用户手册将软件表示为一个黑盒，即重点是放在用户的输入和结果输出。手册可作为发现人机界面中的问题的有价值的工具。

11.6 规约复审

软件需求规约(和/或原型)的复审是由软件开发者和客户一起进行的，因为规约构成了设计和以后的软件工程活动的基础，在进行复审时必须给予特别的重视。

复审首先在宏观的(macroscopic)级别上进行，在该层次上，复审者试图保证规约是完整的、一致的、精确的。下面的问题会被提出：

- 叙述的软件目标和系统的目标是否保持一致？
- 对已经描述了所有系统元素的重要接口吗？
- 已合适的定义了问题域的信息流和结构吗？
- 图是否清楚？每个图可以没有文字补充而单独存在吗？
- 是否主要的功能保留在范围中，并且均已适地描述？
- 软件的行为和它所必须处理的信息及必须完成的功能是否一致？
- 设计约束是现实的吗？
- 是否考虑过开发的技术风险？
- 是否考虑过其他可选的软件需求？
- 校验标准是否被详细地陈述？它们对成功描述的系统是合适的吗？
- 是否存在不一致性、是否信息被忽略或冗余？
- 和客户全面接触过吗？
- 是否用户复审过初步的用户手册或原型？
- 计划的估算受到什么样的影响？

为了开发对上面很多问题的答案，复审可能着重于更详细的层次，这里，我们的关注点是规约的措词，我们试图去发现隐藏在规约内容中的问题。下面的指南是针对详细的规约复审而提出的：

• 着重于说服性的连接词(例如，当然、因此、明确地、显然地、紧随的)，并问“为什么”。

• 观察含糊的术语(例如，一些、有时、经常、通常、一般地、大多数、大多数的)，并进行澄清。

• 当给出了不完整的列表时，确定已理解了所有的项。关键是查找“等等、如此这样”。

• 确定陈述的范围没有包含未陈述的假设(例如，从 10 到 100 的校验代码范围究竟是整数？实数？还是复数？)

• 小心含糊的动词如“处理、拒绝、处制、跳过、限制”等，它们可以以很多方式来解释。

• 小心含糊的代词(例如，I/O 模块与数据校验模块通信，且设置它的控制标志，那么标志是谁的？)

• 查找蕴含了确定性的语句(例如，总是、每次、所有、无、永不)，然后要求证明它们。

• 当某术语被明确地定义在某地方，力图用该定义去替换其他的出现术语。

• 当用语句描述某结构，画出图以帮助理解。

• 当刻划计算时，至少试验两个例子。

一旦复审完成，软件需求规约被客户和开发者双方“终止”，规约变成了软件开发的“合约”。在规约完成后并不被排除请求修改需求，但是，客户应该注意，每个事后的修改是对软件范围的扩展，因此可能增加成本和/或延长项目进度。

即使采用最好的复审规程，一系列常见的规约问题仍然存在。规约是难于“测试”的，因此，不一致性或信息的忽略可能不会被注意到。在复审过程中，对规约的修改可被推荐，但要评估修改的全局影响是极端困难的，即在一个功能中的某个修改如何影响其他功能的需求是很难评估的？现代软件工程环境(第 29 章)使用 CASE 工具来帮助解决这些问题。

11.7 小结

需求分析是软件工程过程的第一个技术步骤，在此阶段，一般性的软件范围陈述被精化为具体的规约，它成为后面的所有软件设计活动的基础。

分析必须关注于问题的信息、功能和行为域，为了更好地理解需要什么，必须创建模型、划分问题、以及描述需求要素和表示以后要开发的实现细节的。

在很多情况下，不可能在早期阶段完整地刻划问题，原型方法提供了一种可选的方法，它产生软件的可执行的模型，并从该模型精化出需求。为了适当地使用原型方法，需要特殊的技术和工具。

软件需求规约作为分析的结果而被开发。复审对保证开发者和客户有相同的系统感觉是非常重要的。不幸的是，即使采用最好的方法，依然存在的问题是：问题总是在不断变化的。

思考题

11.1 软件需求分析无疑是软件工程过程中最强调相互通信的步骤，为什么通信路径经常中断？

11.2 当软件需求分析(和/或系统分析)开始时，经常遇到严重的行政上的反弹，例如，工人可能感觉到工作安全性受到了新的自动化系统的威胁。什么原因导致了这样的问题？分析任务可以进行得使行政问题减少到最小程度吗？

11.3 讨论你对系统分析员的理想培训和背景的理解。

11.4 贯穿本章，我们均涉及“客户”，描述信息系统开发者的“客户”、基于计算机的产品建造者的“客户”、以及系统建造者的“客户”。这里要小心，对该问题可能有比你第一次想象更多的东西。

11.5 开发一个便利的应用规约技术(FAST)“工具箱”，该工具箱应该包括一组指导 FAST 会议的指南、可用于帮助创建列表的材料、以及任何其他的可能对需求定义有帮助的事项。

11.6 你的老师将把班级分成 4—6 个学生的小组，一半小组将扮演市场部门的角色，而另一半将扮演软件工程的角色，你们的工作是定义本章所描述的 SafeHome 安全系统的需求，用本章给出的指南来指导 FAST 会议。

11.7 说“初步的用户手册是一种原型形式”对吗？解释你的回答。

11.8 分析 SafeHome 的信息域，表示(使用任何合适的符号体系)系统的信息流、信息内容和任何相关的信息结构。

11.9 划分 SafeHome 的功能域，首先完成水平划分，然后进行垂直划分。

11.10 创建 SafeHome 系统的基本视图和实现视图的表示。

11.11 建造 SafeHome 系统的纸上原型(或真实原型)，并描述房主和整个系统功能的交互。

11.12 力图标识 SafeHome 系统中可能被其他产品或系统“复用”的软件构件，力图对这些构件进行分类。

11.13 用在图 11－9 中提供的大纲开发 SafeHome 系统的规约。(注：你的老师将建议此时完成的章节)。找出应用了针对规约复审描述的问题。

11.14 你的需求如何不同于其他人为 SafeHome 开发的需求？谁建造了“Chevy”？谁建造了“Cadillac”？

推荐阅读文献及其他信息源

需求分析是一种强调相互通信的活动，如果通信失败，甚至最好的技术方法也将难以实现。Gause 和 Weinberg(Are Your Light On？，Dorset House，1991)，Davis[DAV93]，以及 Kilov 和 Ross(Information Modeling：An Object-Oriented Approach，Prentice-Hall，1994)的书籍包含了对需求分析原则和概念的很好讨论，Jackson 的书(Software Requirements and Specifications，Addison-Wesley，1995)给出了来自该领域的某个专家的观点和指导。

对一种需求收集方法，联合应用软件设计，的全面讨论被 Wood 和 Silver(Joint ApplicationDesign，2nd edition，Wiley，1995)给出，Cohen(Quality Function Deployment，Addison-Wesley，1995)、Gause 和 Weinberg[GAU89]，以及 Zahniser[ZAH90]给出了进行 FAST 会谈的有价值的指南，作者讨论了有效会谈的技巧、集体讨论的方法、可以用于澄清结果的途径、以及一系列其他有用的问题。Martin 的书籍(User Centered Requirements Analysis，Prentice-Hall，1988)也讨论了对有效的客户—开发者通信的需要。

信息域分析是需求分析的基本原则，Mattison(The Object-Oriented Enterprise，McGraw-Hill，1994)、Tillman(A Practical Guide to Logical Data Modelling，McGraw-Hill，1993)，和 Modell(Data analysis，Data Madeling and Classification，McGraw-Hill，1992)等书籍介绍了该重要主题的各个方面。Gehani 和 McGettrick(Software Specification Techniques，Addison-Wesley，1986)编辑了一本规约软件分析话题的重要论文专集，内容从规约的基本原则到高级的规约和设计环境。更新的研究结果可从 IEEE 主办的 Proceedings of the International Symposium onRequirements Engineering 中得到。

Boar 的关于应用原型方法的书籍[BOA84]和 Connell 和 Shafer 的书籍(Structured RapidPrototyping，Prentice-Hall，1989)针对强调的信息系统特征讨论了这种重要的分析技术，然而，作者所讨论的很多话题可跨应用域使用。IEEE 主办的 Proceedings of the InternationalConference on Rapid Prototyping 给出了在快速原型方法方面进展的很好概述。

The Requirements Engineering Newsletter(一个在线出版物)可从下面网址上得到：

http：//web.cs.city.ac.uk/homes/acwf/rehome.html

关于需求工程的扩展参考书目可从下面网址上得到：

http//www.ida.liu.se/labs/aslab/people/joaka/re_bib.html

其他关于需求工程、企业建模、和相关话题的信息可从下面网址上得到：

http：//mijuno.larc.nasa.gov/dfc/re.html

关于软件需求分析活动的最新WWW参考文献列表可在：http://www.rspa.com找到。

① 实际中，如果这些问题已有答案，那么它们中的大部分将作为信息工程行为（第10章有详述）的一个组成部分。

② Davis［DAV93］中只是粗略地讨论了术语“什么”和“怎么做”，有兴趣了解对它们的详细讨论请参阅Davis编写的专著。

① FAST两个更常用的方法——联合应用软件开发（JAD），由IM开发的；和METHOD，由PerformaceResource，Inc开发；已广为大家使用。

① 该例子［有扩展和变型］在后面的许多章节中被用于说明重要的软件工程方法，用它来作为一个练习，使大家在得出自己的一系列FAST会谈和开发列表也是可取的。

① 这儿仅涉及了Davis需求工程原则的很少一部分内容，如想详细了解这些原则，请参阅［DAV95a］。

① 许多人用术语“逻辑的”和“物理的”视图，它们是同一个概念。

① 在[DAV956]中可找到其他候选因素——“什么时候用原型法”——的有效讨论信息。

② 在某些情况下，通过使用第四代技术或复用软件构件可以更快速地构造出更复杂的原型。

第12章　分析建模

在技术层次上，软件工程是从一系列建模任务开始的，这些任务导致了对被建立的软件的完整的需求规约和全面的设计的表示。分析模型——实际上是一组模型，是系统的第一个技术表示。在过去的数年中，人们提出了许多种分析建模的方法，然而，其中两种在分析建模领域占有主导地位，第一个是“结构化分析”，这是传统的建模方法，将在本章进行描述。另一种方法——“面向对象的分析”，将在第20章详细介绍。12.8节将对其他普遍使用的分析方法进行简要的介绍。

结构化分析是一种建立模型的活动。通过使用满足第11章描述的操作性分析原则的符号体系，我们创建描述信息(数据和控制)内容和流的模型、依据功能和行为对系统进行划分、并描述必须要建立的要素。结构化分析不是被所有的使用者一致使用的单一方法，相反，它是发展了几乎20多年的一个混合物。

可能没有其他的软件工程方法引起了如此多的兴趣、被如此众多的人试用(经常是被抛弃，然后再试用)、激起如此多的批评、以及引发如此多的辩论，但这种方法成功了，并在软件工程界获得了大量的追随者。

在这个学科的创始书籍中，Tom DeMarco[DEM79]这样描述结构化分析：

回顾被识别的问题和分析阶段的失败，我建议对分析阶段的目标进行以下的增补。

• 分析的产品必须是高度可维护的。这特别被应用于目标文档(软件需求规约)。

• 必须使用有效的划分方法控制问题的规模。维多利亚时代小说式的规约是不行的。

• 尽可能地使用图形符号。

• 必须区分逻辑的(本质的)和物理的(实现)考虑。

至少，我们需要：

• 某种方法来帮助我们划分需求，并在规约前用文档记录该划分。

• 某种跟踪和评价接口的手段。

• 比叙述性的正文更好的新工具来描述逻辑和策略。

这样，Tom DeMarco 建立了在世界上使用最广泛的分析方法的主要目标。在本章，我们将考察这种方法及其扩展。

12.1 简史

象许多对软件工程的重要贡献一样，结构化分析并不是由里程碑式的明确地涉及这个主题的一篇文章或一本书引入的。分析建模的早期工作开始于 1960 年代后期和 1970 年代初期，但结构化分析方法的第一次出现是作为另一个重要课题——结构化设计的附属品而出现的。研究者(如参考文献[STE74]和[YOU78]需要一种图形符号体系来表示数据和对数据进行变换的处理(加工)，这些处理将最终被映射到体系结构设计中去。

术语“结构化分析”，最初由 Douglas Ross 提出，由 DeMarco[DEM79]进行了推广。在他关于这个主题的书中，DeMarco 引入并命名了使得分析员可以创建信息流模型的主要图形记号、提出了使用这些符号的启发信息、建议“数据字典”和“处理说明”可以作为信息流模型的补充、并给出了说明如何使用这种新方法的大量实例。在以后的几年中，Page-Jones[PAG80]、Gane 和 Sarson[GAN82]等人提出了结构化分析方法的一些变种，在这些变种中，方法关注于信息系统的应用，而没有提供足够的符号来表示实时工程问题中的控制和行为方面的问题。

在 1980 年代中期，结构化分析(当试图应用于面向控制的应用中时)的不足明显地表现了出来。Ward 和 Mellor[WAR85]以及以后的 Hatly 和 Pirbhai[HAT87]实时引入了“扩展”版结构化分析，这些扩展版导致了可以有效地应用于工程问题的一个更加强壮的分析方法。人们还进行了开发一致符号体系的努力[BRU88]以及适应 CASE 工具使用的现代处理[YOU89]方法。

12.2 分析模型的元素

分析模型必须达到三个主要目标：(1)描述客户的需要；(2)建立创建软件设计的基础；(3)定义在软件完成后可以被确认的一组需求。为了达到这些目标，在结构化分析中导出的分析模型采用图 12—1 所描述的形式。

在模型的核心是“数据字典”——包含了软件使用或生产的所有数据对象描述的中心库。围绕着这个核心有三种图，“实体—关系图”(ERD)描述数据对象间的关系，ERD 是用来进行数据建模活动的记号，在 ERD 中出现的每个数据对象的属性可以使用“数据对象描述”来描述。

“数据流图”(DFD)服务于两个目的：(1)指明数据在系统中移动时如何被变换；(2)描述对数据流进行变换的功能(和子功能)。DFD 提供了附加的信息，它们可以被用于信息域的分析，并作为功能建模的基础。在 DFD 中出现的每个功能的描述包含在“加工规约”(PSPEC)中。

“状态—变迁图”(STD)指明作为外部事件的结果，系统将如何动作，为此，STD 表示了系统的各种行为模式(称为“状态”)以及在状态间进行变迁的方式，STD 是行为建模的基础。关于软件控制方面的附加信息包含在“控制规约”(CSPEC)中。

分析模型包含了图 12－1 中提到的各种图、规约、描述和字典。以下各节将对分析模型中的这些元素进行更加详细的讨论。

12.3 数据建模

数据建模回答与任何数据处理应用相关的一组特定问题：系统处理哪些主要的数据对象？每个数据对象的组成如何，哪些属性描述了这些对象？这些对象当前位于何处？每个对象与其他对象有哪些关系？对象和变换它们的处理之间有哪些关系？

为回答这些问题，数据建模使用实体—关系图(ERD)，本节将详细地描述 ERD。ERD 使得软件工程师可以使用图形符号来标识数据对象及它们之间的关系。在结构化分析的语境中，ERD 定义了应用中输入、存储、变换和产生的所有数据。

“实体—关系图只是关注于数据(这样就满足了第一条操作性分析原则)，表示了存在于给定系统中的“数据网络”。ERD 对于数据及其之间的关系比较复杂的应用特别有用。与数据流图不同(数据流图用来表示数据如何被变换，将在 12.4 节讨论)，数据建模独立于变换数据的处理来考察数据。

12.3.1 数据对象、属性和关系

数据模型包含三种互相关联的信息：数据对象、描述数据对象的属性和数据对象相互连接的关系。

数据对象

“数据对象”是几乎任何必须被软件理解的复合信息的表示。“复合信息”是指具有若干不同的特征或属性的事物。因此，“宽度”(单个的值)不是有效的数据对象，但坐标系(dimensions)(包括高度、宽度和深度)可以被定义为一个对象。

数据对象可能是一个外部实体(例如，生产或消费信息的任何事物)、一个事物(例如，报告或显示)、一次行为(例如，一个电话呼叫)或事件(例如，一个警报)、一个角色(例如，销售人员)、一个组织单元(例如，统计部门)、一个地点(例如，仓库)或一个结构(例如，文件)。例如，人或车(图 12－2)可以被认为是数据对象，因为它们可以用一组属性来定义。“数据对象描述”则包含了数据对象及其所有属性。

数据对象是相互关联的，例如，人可以“拥有”车，“拥有”关系意味着人和车之间的一种特定的连接。关系总是由被分析的问题的语境定义的。

数据对象只封装了数据，在数据对象中没有指向作用于数据的操作的引用[①]。因此，数据对象可以表示为如图 12－3 所示的一个表，表头反映了对象的属性。在这个例子中，车是通过制造商、模型、ID＃、实体类型、颜色和拥有者的定义。表格式的实体表示了数据对象的特定实例。例如，Chevy Corvette是数据对象车的一个实例。

属性

属性定义了数据对象的性质，它可以具有以下三种不同的特性之一，它们可以用来：(1)为数据对象的实例命名；(2)描述这个实例；(3)建立对另一个表中的另一个实例的引用。另外，一个或多个属性应被定义为“标识符”，即当我们要找到数据对象的一个实例时，标识符属性成为一个“关键性属性”。在有些情况下，标识符的值是唯一的，尽管这不是必须的。在数据对象车的例子中，ID＃可以是一个合理的标识符。

特定数据对象的一组合适的属性是通过对问题语境的理解而确定的。以上描述的车的属性可以很好地用于机动车辆部门的应用系统，但这些属性对于需要制造控制软件的汽车公司来说是无用的。在后一种情况下，车的属性可能也包括ID＃、实体类型和颜色，但为了使车在制造控制的语境中成为一个有用的对象，必须增加许多其他的属性(如内码、驱动训练类型、掠过软件设计者、传递类型)。

关系

数据对象可以以多种不同的方式互相连接。考虑数据对象“书”和“书店”。这些对象可以用图 12－4a 中的简单符号表示。由于这两个对象是相关的，

在 book 和 bookstore 之间建立了连接。但这个关系是什么呢？为确定该答案，我们必须理解在将要建立的软件的语境中书和书店的角色。我们可以定义一组“对象——关系对”来定义有关的关系。例如：

- 书店订购书。
- 书店陈列书。
- 书店储存书。
- 书店销售书。
- 书店返还书。

关系“订购”、“陈列”、“储存”、“销售”、“返还”定义了书和书店间的相关连接。图 12－4b 以图形的方式显示了这些对象—关系对。

重要的是要注意，对象—关系对是双向的，即它们可以在两个方向读——书店订购书或书被书店订购否[①]。

12.3.2 基数和形态

数据建模的基本元素——数据对象、属性和关系提供了理解问题的信息域的基础，然而，也必须理解与这些基本元素相关的其他元素。

我们已经定义了一组对象，并表示了把它们绑定在一起的对象—关系对，但是，简单地说实体 X 与实体 Y 相关，并没有为软件工程的目的提供足够的信息，我们必须理解实体 X 多少次出现与实体 Y 的多少次出现相关。这引出了称为“基数(cardinality)”的数据建模概念。

基数

数据模型必须能够表示在一个给定的关系中，对象出现的次数。Tillmann[TIL93]如下定义了对象—关系对的基数：

基数是关于一个对象可以与另一个对象的出现次数相关联的出现次数的规约。基数通常简单地表达为‘一’或‘多’。例如，一个丈夫只能有‘一’个妻子(在多数文化中)，而一对父母可以有‘多’个孩子。考虑到‘一’和‘多’的所有组合，两个对象可以如下关联：

- 一对一(1：1)：(对象)A 的一次出现可以并且只能关联到(对象)B 的一次出现，B 的一次出现只能关联到 A 的一次出现。例如，一个丈夫只能有一个妻子，一个妻子也只能有一个丈夫(至少在新泽西州是如此)。

• 一对多(1∶N)：(对象)A 的一次出现可以关联到(对象)B 的一次或多次出现，但 B 的一次出现只能关联到 A 的一次出现。例如，一个母亲可以有多个孩子，但一个孩子只能有一个母亲。

• 多对多(M∶N)：(对象)A 的一次出现可以关联到(对象)B 的一次或多次出现，同时 B 的一次出现也可以关联到 A 的一次或多次出现。例如，一个叔叔可以有多个侄子，一个侄子可以有多个叔叔。

基数定义了“可以参与在一个关系中的对象关联的最大数目”[TIL93]，然而，它没有指出一个特定的数据对象是否必须参与在关系中。为说明这种信息，数据模型为对象—关系对增加了“形态(modality)”。

形态

如果对关系的出现没有明显的需要或关系是可选的时，关系的形态是 0。如果关系必须出现一次，则形态是 1。为说明这一点，我们考虑本地电话公司用于处理区域服务的软件。一个客户指出了一个问题，如果诊断出这个问题是相对简单的，就发生一次简单的修理行为，如果问题是复杂的，就需要多次修理行为。图 12－5 说明了数据对象“客户”和“修理行为”间的关系、基数和形态。

在图中，建立了一个“一对多”的关系。即可以向一个客户提供零个或多个修理行为[①]。在关系连接上距数据对象矩形最近的符号[②]表示的是基数。竖短线表示 1，三叉表示多。形态是用距数据对象矩形较远的记号表示的。左边的第二个竖短线指示为发生一次修理行为，必须有一个客户，右边的圆圈表示对客户报告的问题类型可能不需要修理行为。

12.3.3 实体—关系图

对象—关系对(在 12.3.1 节中进行了讨论)是数据模型的基础，这些对可以使“实体—关系图”(ERD)[③]以图形的方式进行表示。ERD最初是由Peter Chen[CHE77]为关系数据库系统的设计提出的，并被其他人进行了扩展。ERD标识了一组基本的构件：数据对象、属性、关系和各种类型指示符。ERD的主要目的是表示数据对象及其关系。

基本的 ERD 符号已经在 12.3 节进行了介绍，带标记的矩形表示数据对象，连接对象的带标记的线表示关系。在 ERD 的某些变种中，连接线包含一个标记关系的菱形。数据对象的连接和关系使用各种表示基数和形态的特殊符号来建立。

数据对象“车”和“制造商”之间的关系可以表示为如图 12－6 所示。一个厂商生产一辆或多辆汽车。在这张 ERD 所示的语境中，数据对象“车”的规约(图 12－6 中的数据对象表)与较早的规约(图 12－3)有根本的不同。通过考察对象间连接线端的记号，可以看到两处出现的形态都是必须的(竖线)。

扩展这个模型，我们绘制了一张大大简化了的关于汽车业务销售元素的ERD(见图 12－7)，其中引入了新的数据对象“货主”和“销售关系”。另外，新的关系——运输、合同、许可证和货栈指明了图中所示的数据对象如何互相关联。ERD 中包含的每个数据对象的表必须依据本章介绍的规则进行开发。

除了在图 12－6 和 12－7 中介绍的基本 ERD 符号之外，分析员还可以表示“数据对象类型层次”。在许多情况下，数据对象可能实际上表示信息的类或分类，例如，数据对象“车”可以被分类为国内的、欧洲的或亚洲的。图 12－8 所示的 ERD 记号以层次的形式表示了这种分类。

ERD 符号还提供了表示对象间关联性的机制。“关联数据对象”如图 12－9 所示。在图中，为单个子系统建模的每个数据对象都被关联到数据对象“车”。

数据建模和实体—关系图向分析员提供了一种简明的符号体系，从而可以方便对数据处理应用语境中的数据考察。在多数情况下，数据建模方法用来创建部分分析模型，但它也可以用于数据库设计，并支持任何其他的需求分析方法。

12.4 功能建模和信息流

当信息“流”过基于计算机的系统时会被变换。系统以多种形式接受输入；应用硬件、软件以及人的元素将输入变换为输出；并以多种形式产生输出。输入可能是由传感器传输的一个控制信号、由操作员键入的一列数字、通过网络连接传输的一个信息包、或从 CD-ROM 提取的大量的数据文件。变换可以包括单个的逻辑比较、复杂的数值算法或专家系统中的规则—推理方法。输出可能是使一个 LED 发光、或产生 200 页的报告。实际上，我们可以为任何基于计算机的系统产生“流模型”，不管其规模与复杂性。

结构化分析开始是作为信息流建模技术的。基于计算机的系统被表示为图 12－10 所示的信息变换流程图。系统的整个功能被表示为单个的信息变换流，即图中的“泡泡”。表示为带标记的箭头的一个或多个输入来自于表示为方框的外部实体，输入驱动了变换并产生传递给外部实体的输出信息(也表示为带标记的箭头)。要注意的是，这个模型可以应用于整个系统，也可以只应用于软件元素。关键是表示变换所接受和产生的信息。

12.4.1 数据流图

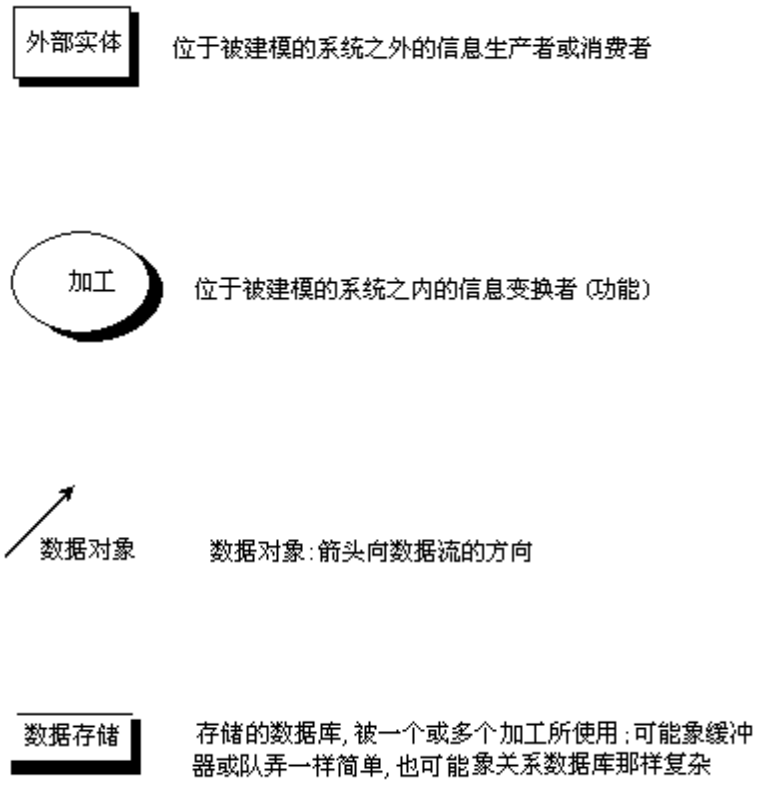
当信息在软件中移动时，它会被一系列变换所修改。“数据流图”(DFD)是描述信息流和当数据从输入移动到输出时被应用的变换的图形化技术。数据流图的基本形式如图 12－10 所示。DFD 也被称为“数据流图表”或“泡泡图”。

数据流图可以被用来抽象地表示系统或软件。事实上，DFD 可以被划分为内容在不断增加的多种信息流和功能细节的表示图，因此，DFD 既提供了功能建模

的机制，也提供了信息流建模的机制，从而，它满足了第 11 章讨论的第二条操作性分析原则(即创建功能模型)。

第 0 层的 DFD，也被称为基本系统模型或语境模型，将整个软件元素表示为一个具有分别由进入的和离开的箭头表示的输入和输出数据的泡泡。当第 0 层的 DFD 被划分来揭示更多的细节时，附加的加工(泡泡)和信息流路径也被表示出来。例如，第 1 层的 DFD 可能包括 5 或 6 个具有互相连接箭头的泡泡。表示在第 1 层的每个加工代表的是在语境模型中的整个系统的子功能。

用于建立DFD的基本符号体系[①]如图 12－11 所示。矩形用来表示外部实体，即为软件中的变换产生信息或接受软件产生的信息的系统元素(例如硬件、人、另一个程序)或另一个系统。圆圈代表应用于数据(或控制)并以某种方式改变它的“加工”或“变换”。箭头代表一个或多个数据项或数据对象。数据流图中所有的箭头应具有标记。双线代表“数据存储”——被软件所使用的要保存的信息。DFD记号的简单性是结构化分析技术被最广泛地使用的原因之一。



重要的是要注意，图中没有提供对加工顺序的明显的指示。过程或顺序在图中可以是隐式的，但是，明显的过程表示通常延迟到软件设计中才表示。

正如我们前面提到的，每个泡泡可以通过描述更多的细节来精化或层次化。图 12－12 说明了这个概念，系统 F 的一个基本模型指出主要的输入是 A，最终的输出是 B。我们将模型 F 精化为变换 f1 到 f7,注意必须保持“信息流连续性”，即必须保持每次精化的输入和输出相同，这个概念，有时称为“平衡”，对于开

发一致的模型非常重要。对 f4 的进一步精化，以变换 f41 到 f45 的形式描述了细节，输入(X， Y)和输出(Z)仍然没有变化。

数据流图在软件需求分析中是非常有价值的图形工具，但如果将它的功能与流程图混淆就有可能对图进行错误的解释，数据流图描述信息流，不明显地表示过程逻辑(如条件或循环)，它不是具有圆形边的流程图。

用于开发DFD的基本符号自身并不能充分地描述软件的需求，例如，DFD中的箭头代表输入到加工中或从加工中输出的数据对象，数据存储表示有组织的数据的集合，但是，箭头所意味的或存储所描述的数据的“内容”是什么？如果箭头(或存储)代表数据的集合，那么数据是什么？应用结构化分析的另一部分基本符号——“数据字典”[①]可以回答这些问题，数据字典的形式和使用在本章的以后部分有介绍。

最后，图 12－11 中的图形符号必须用叙述性的正文来说明。“加工规约”(PSPEC)可以用来说明 DFD 中的泡泡蕴含的加工细节，加工规约描述了功能的输入、施加于输入的算法、产生的输出。另外，PSPEC 指示了加工(功能)的约束和限制、与加工相关的性能特征、以及影响加工实现方式的设计约束。

12.4.2 针对实时系统的扩展

许多应用软件是依赖于时间的，并将越来越多的面向控制的信息作为数据进行处理。实时系统必须在现实世界所规定的时间框架内与现实世界进行交互。飞机电子设备、制造过程控制、客户产品以及工业仪器仅仅是数百种实时软件应用中的一部分。

为适应实时系统的分析，人们提出了对结构化分析的基本符号的一些扩展。由 Ward 和 Mellor [WAR85]以及由 Hatley 和 Pirbhai[HAT87]给出的扩展如图 12－13 和图 12－14 所示，这些扩展使得分析员可以在表示数据流和加工的同时，表示控制流和控制加工。

12.4.3 Ward 和 Mellor 扩展

Ward 和 Mellor[WAR85]对基本的结构化分析符号体系进行了扩展，从而使得这些符号体系可以适应实时系统提出的以下要求：

- 在时间连续的基础上收集或产生信息流。
- 传遍系统的控制信息以及相关的控制处理。
- 同一个变换的多个实例，这在多任务的条件下可能遇到。
- 系统状态以及导致状态变迁的机制。

在相当比例的实时系统应用中，系统必须监控由一些现实世界处理产生的“时间连续”的信息。例如，汽油涡轮发动机的实时测试监控系统需要监控涡轮速度、燃烧室温度和连续探测到的各种压力，传统的数据流符号不能区分离散的数据和时间连续的数据。对基本结构化分析符号体系的一个扩展，如图 12－15 所示，提供了表示时间连续的数据流的机制，双头箭头用来表示时间连续的流，单头箭头指示离散的数据流。在图中，“监控温度”被连续地测量，同时提供了“温度设置点”的单个值，图中的加工产生了一个时间连续的输出“已校正的值”。

离散的和时间连续的数据流的区分对于系统工程师和软件设计者都有重要的影响。在创建系统模型时，系统工程师最好能够将性能关键的加工分离出来(通常时间连续的数据的输入和输出是关键性性能)。创建物理或实现模型时，设计者必须建立搜集时间连续的数据的机制，很明显，数字系统使用诸如高速轮流检测等技术以半连续的方式来搜集数据。符号指明了哪里需要模拟数字硬件，以及哪些变换需要高性能的软件。

在传统的数据流图中，控制或“事件流”没有显式地表示出来，事实上，分析员被特别地警告，在数据流图中排除控制流的表示。但当考虑实时应用时，这种排除太严格了，为此，开发了代表事件流和控制加工的特殊符号。数据流继续使用传统的数据流图中的实线箭头表示，“控制流”表示为虚线箭头或阴影箭头，只处理控制流的加工，称为“控制加工”，类似地表示为虚线泡泡。

控制流可以直接输入到一个传统的加工或控制加工中，图 12－16 以Ward和Mellor的符号说明控制流和加工，该图说明了一个制造车间[①]的数据和控制流的顶层视图。当要被机器人组装的部件放在工作固件上时，就设置指示每个部件存在或空缺的“部件状态缓冲器”(一个控制存储件)的某个状态位，包含在“部件状态缓冲器”中的事件信息作为“位串”传递给加工“监控固件和操作接口”，该加工只在控制信息“位串”指示所有的工作固件都有部件时才读“操作命令”。一个事件标志“开始/停止”发送给“机器人初始化控制”，它是一个使得进一步的命令处理可以进行的控制加工。作为发送给“处理机器人命令”的“处理活动”事件的后继结果，产生了其他数据流。

在某些情况下，在实时系统中可能出现同一个控制或数据变换加工会引发多个后续工作的实例。当任务作为内部加工或外部事件的结果产生时，这种情况可能在多任务的环境中发生，例如，多个“部件状态缓冲器”可能被监控，这样在适当的时间可以用信号通知不同的机器人开始工作。另外，每个机器人可能有自己的机器人控制系统，图 12－13 显示了用 Ward 和 Mellor 符号来表示相同加工的“多个对等实例”。

12.4.4 Hatley 和 Pirbhai 扩展

Hatley 和 Pirbhai[HAT87]对基本结构化分析符号体系的扩展对创建新的图形符号关注较少，对软件的面向控制方面的规约表示关注的较多。在图 12－14 中，虚线再次用来表示控制或事件流。与 Ward 和 Mellor 不同，Hatley 和 Pirbhai

建议虚符号和实符号分开来表示，这样，就定义了“控制流图”(CFD)。CFD 包含与 DFD 相同的加工，但显示控制流而不是数据流。与在流模型中直接表示控制加工相反，使用了对“控制规约”(CSPEC)的一个引用符号(实短线)，实质上，实短线可以看作是进入到“执行者”(CSPEC)的一个窗口，该执行者基于通过窗口传送的事件而控制表示在 DFD 中的加工(功能)。将在 12.6.4 节中描述的 CSPEC，用来指明：(1)当事件或控制信号被感知时软件如何行动；(2)作为事件发生的结果，哪些加工被激活。加工规约用来描述表示在流图中的加工的内部工作。

使用图 12－11 和图 12－14 所示的符号，以及包含在 PSPEC 和 CSPEC 中的附加信息，Hatley 和 Pirbhai 创建了实时系统的模型。数据流图用来表示数据和操作数据的加工；控制流图表示事件在加工间的流动，说明导致各种加工被激活的外部事件。加工和控制模型的关联显示在图 12－17 中，加工模型通过“数据条件”连接到控制模型，控制模型通过包含在 CSPEC 中的激活信息的加工连接到加工模型。

当一个加工的数据输入导致了一个控制输出时，数据条件发生，这种情况如图 12－18 所示，这是部分石油精炼中压力容器自动监控和控制系统的流模型。加工“检测＆转换压力”实现了在 PSPEC 伪码中算法的描述，当绝对油箱压力大于允许的最大值时，产生了一个“压力过大”事件。当使用 Hatley 和 Pirbhai 符号时，数据流是 DFD 的一部分，而控制流则独立地构成控制流图的一部分。为了确定当事件发生时，会出现什么事情，我们需要检查 CSPEC。

控制规约(CSPEC)包含了一系列重要的建模工具，“加工激活表”(在 12.6.4 节描述)用来指明什么加工被给定的流过竖短线的事件激活，例如，图 12－18 中的加工激活表(PAT)指示“压力过大”事件将导致加工“减小容器压力”(图中未显示)被激活。除了 PAT 以外，CSPEC 还包括“状态—变迁图”(STD)，STD 是依赖于一组“系统状态”的定义的行为模型，将在下节中介绍。

12.5 行为建模

行为建模是所有需求分析方法的操作原则。然而，只有结构化分析的扩展版本(见参考文献[WAR85]和[HAT87])才提供了这种建模的符号。状态—变迁图[①]通过描述状态以及导致系统改变状态的事件来表示系统的行为，另外，STD指明了作为特定事件的结果，要执行哪些行为(例如，活动加工)。

状态是可观察的行为模式，例如，在 12.4.4 节中所描述的压力容器监控和控制系统的状态可能包括“监视状态”、“报警状态”、“压力释放状态”等，每个状态代表系统的一种行为模式。状态变迁图指明了系统如何在状态间移动。

为了例举 Hatley 和 Pirbhai 的控制和行为扩展的使用，以一台办公室复印机中嵌入的软件为例。复印机要执行包含在图 12-19 所示的第 1 层 DFD 中的若干功能，应该注意数据流和每个数据项的定义(使用数据字典)需要进一步的精化。

复印机软件的控制流显示在图 12－20 中。控制流显示了进入和离开单个加工以及 CSPEC“窗口”的状态，例如，事件“进纸状态”和“开始/停止”流入了 CSPEC 短线，这意味着每个事件将导致 CFD 中的某些加工被激活。考察 CSPEC 的内部，“开始/停止”事件将激活/撤销“复印管理”加工。类似地，“卡纸”事件(“进纸状态”的一种)将激活“完成问题诊断”活动。应该注意，CFD 中所有的竖短线指向相同的 CSPEC。

事件流可以直接被输入到加工中，如“重新处理失败”所示。然而，这个流并没有激活加工，而是为加工的算法提供了控制信息，数据流箭头为了举例的目的被轻微地加上了阴影，但是在现实中，它们不是作为控制流图的一部分显示的。

以上描述的复印机软件的一个简化状态的变形图如图 12－21 所示。矩形代表系统状态，箭头代表状态间的“变迁”。每个箭头用规则表达式标记，箭头上的说明的“分子”值指明导致变迁发生的事件，“分为”值指明作为事件的结果发生的行为，因此，当纸匣是“满”并且“开始”按钮被按下时，系统由“读命令”状态进入“复印”状态。状态不必与加工一一对应，例如，状态“复印”包含了图 12－20 所示的加工“复印”和“向用户显示过程”。

12.6 结构化分析的技巧

在前面的一节中，我们讨论了结构化分析的基本和扩展符号体系。为了在软件需求分析中有效地应用这些符号必须与一组使得软件工程师可以导出良好分析模型的启发性信息相结合。为了阐明这些启发性信息的使用，在本章的其余部分，我们将采用 Hatley 和 Pirbhai[HAT87]对基本结构化分析符号体系的扩展的一个修改版本来说明问题。

在以下的几节中，我们将考察为使用结构化分析开发完整的和准确的模型所需要的每个步骤。在这些讨论中，将使用 12.4 节介绍的符号体系，并详细地给出前面间接地介绍的其他符号形式。

12.6.1 创建实体—关系图

实体—关系图使得软件工程师可以完整地刻划系统输入和输出的数据对象、定义这些对象的性质的属性、以及对象间的关系。与分析模型中的其他元素一样，ERD 是以迭代的方式构造的。可以采用以下的方法：

1.在需求搜集的过程中，要求客户列出应用软件或业务过程涉及到的“事物”。这些“事物”演化为一组输入和输出的数据对象，以及生产和消费信息的外部实体。

2.一次考虑一个对象，分析员和客户定义这个对象和其他对象间是否存在连接(在这个阶段没有命名)。

3.当连接存在时，分析员和客户应创建一个或多个对象—关系对。

4.对每个对象—关系对，考察其基数和形态。

5.迭代地进行步骤 2 到步骤 4，直至定义了所有的对象—关系对。在这个过程进展中发现遗漏是很正常的。当进行若干次迭代时，将总是不断地增加新的对象和关系。

6.定义每个实体的属性。

7.形式化并复审实体—关系图。

8.重复步骤 1 到步骤 7，直到数据建模完成。

为说明这些基本指南的使用，我们将使用第十一章讨论的 SafeHome 安全系统的例子。再次描述 SafeHome 的处理如下：

SafeHome 软件使得房主能够在安装时配置安全系统，监控所有和安全系统连接的传感器，以及通过包含在 SafeHome 控制面板(controlpanel)(如图 11－4 所示)中的键盘和功能键和房主进行信息交互。

在安装过程中，SafeHome 控制面板被用于“编程”和配置系统，每个传感器被赋予一个编号和类型，编置主人密码以启动和关闭系统，当传感器事件发生时将输入电话号码并拨号。

当某传感器事件被识别出时，软件激活一个附于系统上的可发声的警报，在一定的延迟时间后(由房主在系统配置活动中指定)，软件拨出监控服务的电话号码，并报告关于位置和被检测到的事件的性质等信息，电话号码将每 20 秒重拨一次，直至电话接通。

和 SafeHome 的所有交互由用户交互子系统管理，该子系统读入通过键盘和功能键提供的输入，在 LCD 显示屏上显示提示信息和系统状况。键盘交互采用下面的形式…。

分析员和客户间的讨论指出以下一组(部分)“事物”是与问题相关的：

- 房主。
- 控制面板。
- 传感器。
- 安全系统。
- 监控服务。

一次只考虑一个“事物”，才能发现连接。为此，画出每个对象，并标记连接对象的线，例如，图 12－22 显示了“房主”和“控制面板”、“安全系统”、“监控服务”间存在着直接连接，“传感器”和“安全系统”间也存在着单一连接，等等。

一旦定义了所有的连接后，针对每个连接标识一个或多个对象—关系对，例如，可确定“传感器”和“安全系统”间的连接具有以下对象—关系对：

security system monitors sensor 安全系统“监控”传感器

security system enables/disables sensor 安全系统“允许/不允许”，传感器起作用

security system tests sensor 安全系统“检测”传感器

security system programs sensor 安全系统“编程”传感器

分析以上的每个对象—关系对以确定基数和形态，例如，在对象—关系对安全系统“监控”传感器中，安全系统和传感器间的基数是一对多，形态是安全系统的一次出现(必须的)和传感器的至少一次出现(必须的)。使用 12.3 节引入的 ERD 记号，安全系统和传感器间的连接被修改为如图 12－23 所示。类似的分析也要应用于其他所有数据对象。

考察每个对象从而确定其属性。由于我们考虑的软件必须支持 SafeHome，属性应集中于为了使系统运行而必须存储的数据，例如，传感器对象应有以下属性：传感器类型、内部标识号、区域位置、和警报级别。

12.6.2 创建数据流模型

数据流图(DFD)使得软件工程师可以同时开发信息域和功能域的模型。当 DFD 被精化到较细的级别时，分析员对系统进行了隐式的功能分解，这样完成了第四条操作性分析原则。同时，当数据流过体现应用的加工时，DFD 的精化导致了数据的相应精化。

一些简单的指南在导出数据流图时会有所帮助：(1)第 0 层的数据流图应将软件/系统描述为一个泡泡；(2)应仔细地标记主要的输入和输出；(3)通过隔离要表示在下一层中的候选加工、数据对象和存储而开始精化过程；(4)所有的箭头和泡泡应使用有意义的名称标记；(5)当从一个级别到另一个级别时要维护“信息流连续性”；(6)一次精化一个泡泡。经常存在一种使数据流图过分复杂的自然趋势，当分析员试图过早地显示过多的细节或在信息流中表示软件的过程时，会发生这种情况。

再来考虑 SafeHome 产品，图 12－24 显示了系统第 0 层的 DFD。主要的外部实体(方盒)产生了系统所使用的信息及系统产生的消费信息，带标记的箭头代表

数据对象或数据对象类型层次。例如，用户命令和数据包括了所有的配置命令、所有的激活/撤销命令、所有各色各样的交互以及所有修饰或扩展某命令的输入数据。

第 0 层的 DFD 现在要扩展到第 1 层，但我们应怎样进行呢？一个简单而有效的方法是对描述语境级的泡泡的处理进行“语法扫描”描述，即将叙述中的所有名词(和名词短语)和动词(和动词短语)隔离出来。为进行说明，我们再重述一次处理的描述方法，对其中第一次出现的名词加下划线，对第一次出现的动词采用斜体。(忽略同义词的或与建模过程没有直接关系的名词和动词。)

SafeHome 软件使得房主能够在安装时配置安全系统，监控所有和安全系统连接的传感器，以及通过包含在 SafeHome 控制面板(control Panel)(如图 11－4 所示)中的键盘和功能键和房主进行信息交互。

在安装过程中，SafeHome 控制面板被用于“编程”和配置系统，每个传感器被赋予一个编号和类型，编置主人密码以启动和关闭系统，当传感器事件发生时将输入电话号码并拨号。

当某传感器事件被识别出时，软件激活一个附于系统上的可发声的警报，在一定的延迟时间后(由房主在系统配置活动中指定)，软件拨出监控服务的电话号码，并报告关于位置和被检测到的事件的性质等信息，电话号码将每 20 秒重拨一次，直至电话接通。

和 SafeHome 的所有交互由用户交互子系统管理，该子系统读入通过键盘和功能键提供的输入，在 LCD 显示屏上显示提示信息和系统状况。键盘交互采用下面的形式…。

当考察这种语法扫描时，我们看到出现了一种模式，所有的动词都是 SafeHome 的加工，即它们最终将被表示为后来的 DFD 中的泡泡，所有的名词是外部实体(方盒)或数据或控制对象(箭头)或数据存储(双线)，进一步注意到名词和动词可以互相连起来(例如，传感器被赋予一个编号和类型)。因此，通过对任何 DFD 层次中某泡泡的处理的描述可进行语法扫描，我们可以产生许多关于如何精化到下一个层次的有用的信息。使用这些信息，第 1 层的 DFD 如图 12－25 所示 图 12－24 所示的语境层次的加工通过对语法扫描的考察被扩展为 7 个加工，类似地，第 1 层加工间的信息流也通过扫描而导出。

应该注意，在第 0 层和第 1 层之间保持了信息流连续性，在第 0 层和第 1 层 DFD 中的输入和输出的内容的说明推迟到 12.7 节。

第 1 层 DFD 中的加工可以被进一步精化到更低的级别，例如，加工“监控传感器”可以被精化为如图 12－26 所示的第 2 层 DFD，再次注意级别之间保持了信息流连续性。

DFD 的精化可以连续进行，直至每个泡泡只执行一个简单的操作，即直至每个泡泡所代表的加工执行一个可以很容易地实现为程序组成部分的功能。在第

13 章，我们将讨论“内聚”的概念，它可以用来评估给定功能的简单性，现在，我们只是努力地精化，直到每个泡泡都是简单的。

12.6.3 创建控制流模型

对于很多数据处理应用的类型来说，数据模型和数据流模型对于获得关于软件需求的有意义的考察均是必要的。然而，正如我们前面提到的，存在这样的一大类应用软件——它们是事件驱动的，而不是数据驱动的；产生控制信息，而不是报告或显示值；处理信息时非常关注时间和性能。这些应用软件在数据流建模以外还需要使用控制流建模。

创建控制流图(CFD)所需的图形记号在 12.4.4 节中已说明，为了回顾创建 CFD的方法，从数据流模型中“剥去”所有的数据流箭头[①]，然后向图中加入事件和控制项(虚线箭头)，并显示一个到控制规约的“窗口”(竖短线)。但如何选择事件呢？

我们已经提到事件或控制项实现为布尔值(例如，true 或 false，on 或 off，1 或 0)或条件的离散列表(缺纸、卡纸、满)。为选择潜在的候选事件，有以下指南：

- 列出所有被软件“读取”的传感器。
- 列出所有的中断条件。
- 列出所有被操作者开动的“开关”。
- 列出所有的数据条件。
- 回忆对加工叙述进行的名词—动词扫描，回顾所有作为可能的 CSPEC 的输入/输出的“控制项”。
- 通过标识其状态描述系统的行为；标识这些状态是如何达到的，并定义状态间的变迁。
- 关注可能的省略——这是在刻划控制时非常普遍的错误(例如，问“有什么其他的途径可以达到这个状态或从它离开吗？”)。

SafeHome 软件的第 1 层 CFD 如图 12－27 所示。在这些事件和控制项中要注意传感器事件(即一个传感器已经出错)、闪烁标志(在 LCD 显示上闪烁的一个信号)和开/关切换(关闭或打开系统的信号)。从外部世界流入 CSPEC 窗口的事件意味着 CSPEC 将激活一个或多个显示在 CFD 中的加工，当控制项从一个加工发出，流入 CSPEC 窗口时，意味着某些其他加工或外部实体的控制和激活。

12.6.4 控制规约

控制规约(CSPEC)在两个不同的方面表示系统的行为(在它被引用的层次上)，CSPEC 包括一个状态变迁图(STD)，它是行为的“顺序规约”；CSPEC 还包括加工激活表(PAT)——行为的“组合规约”。CSPEC 的基本属性在 12.4.4 节进行了介绍，现在到了看看结构化分析这种重要记号的一个例子的时候。

图 12－28 显示了 SafeHome 的第 1 层流模型的“状态—变迁图”。带标记的变迁箭头指明当系统定义在这个层次上的四个状态间来回移动的时候，如何对事件作出响应。通过研究 STD，软件工程师可以确定系统的行为，更重要的是，可以确定被描述的行为中是否存在“空洞(hole)”，例如，这张 STD(图 12－28)指明当遇到开/关切换事件时，从状态“读用户输入”出发的唯一变迁发生，并到达状态“监控系统”，然而，除了发生“传感器事件”，没有方法使得系统回到状态“读用户输入”，这是规约中的一个错误，是希望在复审中发现并修改的。请考察 STD，以确定是否有其他异常之处。

一种不同的表示行为的模式是“加工激活表”(PAT)，PAT 在加工的语境中，而不是在状态的语境中，表示 STD 中包含的信息，即这张表指明当事件发生时，流模型中的哪些加工(泡泡)被激活。PAT 可以作为那些必须确立执行者来控制在该层次上表示的加工的设计者的指南。SafeHome 软件的第 1 层流模型的 PAT 如图 12－29 所示。

CSPEC 描述了系统的行为，但它没有提供关于作为行为的结果被激活的加工的内部工作的任何信息，提供这种信息的建模记号在下一节讨论。

12.6.5 加工规约

加工规约(PSPEC)用来描述出现在求精过程的最终层次的所有流模型加工。加工规约的内容可以包括叙述性正文、加工算法的“程序设计语言”(PDL)描述[①]、数学方程、表、图或图表。为流模型中的每个泡泡提供了PSPEC，软件工程师就创建了一个“小规约”，它可以作为创建软件需求规约的第一步，并作为对实现加工的程序成分进行设计的指南。

为了例举 PSPEC 的使用，以一个软件应用为例，其中要分析各种几何对象的维度，以识别对象的形状。语境层次的数据流图被持续地精化直到产生第 2 层的加工，其中一个，三角分析，如图 12－30 所示。三角分析的 PSPEC 首先如图写成英语的叙述。如果在这个阶段希望增加算法细节，一个程序设计语言的表示(图 12－31)也可以作为 PSPEC 的一部分，然而，多数人认为，PDL 的版本应推迟到设计开始时。

12.7 数据字典

分析模型中包含对数据对象、功能和控制的表示。在每种表示中，数据对象和/或控制项都扮演一定的角色，因此，有必要提供一种有组织的方式来表示每个数据对象和控制项的特性，这是由数据字典来完成的。

数据字典是为描述在结构化分析中定义的对象的内容而作为半形式化的语法被提出的，这个重要的建模记号被定义如下[YOU89]：

数据字典是对所有与系统相关的数据元素的一个有组织的列表，以及精确的、严格的定义，使得用户和系统分析员对于输入、输出、存储成分和[甚至]中间计算有共同的理解。

今天，数据字典几乎总是作为 CASE“结构化分析与设计工具”的一部分。尽管各工具中字典的形式各不相同，但都包含以下信息：

• 名称——数据或控制项、数据存储或外部实体的主要名称

• 别名——第一项的其他名字

• 何处使用/如何使用——使用数据或控制项的加工列表，以及如何使用(例如，加工的输入、加工的输出、作为存储、作为外部实体)

• 内容描述——表示内容的符号

• 补充信息——关于数据类型、预设值(如果知道)、限制或局限等的其他信息

一旦数据对象或控制项的名称和别名被输入了数据字典，就要保持命名的一致性，即如果一个分析组的成员决定将一个新导出的数据项命名为 xyz，而 xyz 已经存在于字典中，支持字典的 CASE 工具应发出警告，指出重名，这改进了分析模型的一致性，有助于减少错误。

“何处使用/如何使用”的信息是从流模型中自动记录的，当一个字典项被创建时，CASE工具扫描DFD和CFD，确定哪些加工使用了该数据或控制信息、以及如何使用。尽管这看起来并不重要，但这事实上是字典的最重要的益处之一。在分析中，总是存在连续的修改流，对于大型的项目来说，确定修改的影响常常是很困难的，很多软件工程师会问：“这个数据对象在哪里被使用了？如果修改了它还有哪里会被影响？这个修改的整体影响如何？”因为数据字典可以看作一个数据库[①]，分析员可以询问“何处使用/如何使用”的问题，并获得回答。

用来开发内容描述的符号如表 12－1 所示，使得分析员可以以三种基本的构造方式之一来表示“复合数据”(即数据对象)：

1.作为数据项的序列。

2.作为从一组数据项中的选择。

3.作为数据项的重复分组。

每个表示序列、选择或重复的一部分的数据项自身可能是另一个数据对象，需要在字典中进一步精化。

表 12－1　数据字典的内容描述符号

数据结构	记号	意义
	=	由……构成
顺序	+	和
选择	[|]	或
重复	$\{\}^n$	N 次重复
	()	可选择的数据
	* *	限定的注释

为说明数据字典的使用和表 12－1 所示的内容描述符号，我们回到图 12－26 所示的 SafeHome 中“监控系统”加工的第 2 层 DFD。在该图中，数据项“电话号码”被刻划为输入，但严格地讲，电话号码是什么呢？它可以是 7 位本地电话、4 位的分机电话，或 25 位的长途载波序列，数据字典提供了 DFD 中“电话号码”的精确定义。另外，它还指出了何处和如何使用这个数据项，以及相关的任何其他补充信息，数据字典中该项的定义如下：

名称　　电话号码

别名　　无

何处使用　　如何使用　预计设置(输入)

拨号(输出)

描述　　电话号码=当地分机号|外地号码

以上的内容可以读作：电话号码或者是当地分机号的组合，或者是外地号码。当地分机号和外地号码表示复合数据，必须在其他的内容描述陈述中进一步精化。继续该内容描述：

电话号码=[当地分机号|外地号码]

当地分机号=[2001|2002…|2999]

外地号码=9+[当地号码|长途号码]

当地分机号=前缀+访问的号码

长途号码=(1)+区号+当地号码

前缀=[795|799|874|877]

访问的号码=*任意四位串号码*

内容描述被扩展直至所有的复合数据项(数据对象)都被表示为元素项(不需要进一步扩展的项)，或所有的数据对象用众所周知的对所有读者没有歧义的方式(例如,区号通常被理解为不以 0 或 1 开头的 3 位数字)表示。重要的是要注意，基本数据的规约常常限制系统，例如，前缀的定义指明在本地只可以访问 4 个分支交换机。

数据字典明确地定义了信息项，尽管我们假定图 12－26 所示的 DFD 中的电话号码可以包括 25 位的长途载波数字，数据字典内容描述告诉我们，这样 25 位长的数字不是可以使用的数据。

对于大型的基于计算机的系统，数据字典的规模和复杂性迅速地增长，事实上，手工地维护数据字典是非常困难的，这就是使用 CASE 工具的原因。

12.8 其他传统分析方法的概述

多年以来，在工业界已使用了许多其他有价值的软件需求分析方法。它们都遵循第 11 章讨论的操作性分析原则，每种方法都引入了构造分析模型的不同符号体系和启发信息。在本节，我们对三种使用较为普遍的方法进行简要的概述。为获得进一步的信息，有兴趣的读者可以去读参考文献[①]。

12.8.1 数据结构化系统开发

“数据结构化系统开发”(DSSD)，也被称为 Warnier－Orr 方法，是由 J.D.Warnier[WAR74，WAR81]进行的信息域分析的开创性工作演化而来。Warnier 开发了使用顺序、选择和重复三种结构表示信息层次的符号体系，并说明软件结构可以由数据结构直接导出。

Ken Orr[ORR77,ORR81]扩展了 Warnier 的工作，使得它可以包含信息域的更广泛的视图，这样演化成为数据结构化系统开发。DSSD 包含信息流和功能特征以及数据层次。

12.8.2 Jackson 系统开发

Jackson 系统开发(JSD)由 M.A.Jackson(见参考文献[JAC75 和 JAC83])关于信息域分析及其与程序设计和系统设计的关系方面的工作演化而来。在某些方面与 Warnier 的方法和 DSSD 类似,JSD 关注“现实世界”信息域的模型,用 Jackson 的话来说[JAC83]：“开发者首先创建系统涉及的、提供系统主题的现实模型……”

为实施 JSD，分析员应采用以下步骤：

实体动作步骤：采用与第 20 章介绍的面向对象分析技术非常类似的方法，标识“实体”(系统需要用于生产或使用信息的人、对象或组织)和“动作”(发生在现实世界中影响实体的事件)。

实体结构步骤：影响实体的动作依据时间排序并用“Jackson 图”(一种树型的符号)表示。

初始建模步骤：实体和动作被表示为处理模型；定义模型和现实世界间的连接。

功能步骤：刻划与被定义的动作相对应的功能。

系统时间步骤：估算和刻划处理的调度特性。

实现步骤：刻划作为设计的硬件和软件。

JSD 中最后三步与系统或软件设计有密切的联系。

12.8.3 SADT

“结构化分析和设计”(SADT)是作为系统定义、处理表示、软件需求分析和系统/软件设计的符号体系而被广泛地应用的技术(见参考文献[ROS77 和 ROS85])。SADT 包括允许分析员对软件(或系统)功能进行分解的过程、对信息间的关系和软件中的功能进行通信的图形符号——SADT“活动图和数据图”、以及应用此方法的项目控制指南。

SADT 方法包括了支持分析过程的自动化工具和应用工具的良好定义的组织装备，刻划说明了复审和里程碑，以允许对开发者—客户之间的通信进行确认。

12.9 小结

结构化分析是最广泛使用的需求建模方法，它是依赖于数据建模和流建模来创建全面的分析模型的基础。使用实体—关系图，软件工程师创建了系统中所有重要的数据对象的表示。数据和控制流图用来作为表示数据和控制转换的基础。同时，这些模型用来创建软件的功能模型，并为功能划分提供机制。使用状态变迁图创建行为模型，使用数据字典开发数据内容。加工和控制规约提供了附加的细节说明。

结构化分析的初始记号是为传统的数据处理应用开发的，扩展使得该方法适用于实时系统。很多 CASE 工具支持结构化分析，它们辅助创建模型中的元素，并帮助保证一致性和正确性。

思考题

12.1 至少获取 12.1 节中的三篇引用并写一篇简要的文章，给出对结构化分析的理解随着时间变化的纲要。作为结论，提出你认为这种方法在将来的发展方向。

12.2 你被要求建立以下的系统之一：

a.你所在大学的基于网络的课程注册系统；

b.计算机软件产品的直接邮件广告客户的订货处理系统；

c.小型业务的简单发货系统；

d.代替 Rolodex 的软件产品；

e.自动食谱产品

选择一个你感兴趣的系统，开发描述数据对象、关系和属性的实体—关系图。

12.3 基数和形态的区别是什么？

12.4 为思考题 12.2.中列出的五个系统之一绘制语境层次模型(第 0 层 DFD)。为系统撰写语境层次的加工描述。

12.5 使用思考题 12.4.中开发的语境层次 DFD，开发第 1 层和第 2 层数据流图，以对语境层次加工描述进行“语法扫描”作为开始，通过为泡泡之间的箭头加标记刻划所有的信息流，为每个变换使用有意义的名称。

12.6 为你在思考题 12.2.中选择的系统开发 CFD、CSPEC、PSPEC 和数据字典，努力使你的模型完整。

12.7 信息流连续性是否意味着在第 0 层作为输入的流箭头，在以后的层次中必须作为输入出现？讨论你的答案。

12.8 使用 Ward 和 Mellor 扩展，重新绘制图 12－19 和图 12－20 中的流模型。你如何包括隐含在图 12.20 中的 CSPEC？Ward 和 Mellor 并没有使用这个符号。

12.9 使用 Hatley 和 Pirbhai 扩展，重新绘制图 12－16 中的流模型。你如何包括隐含在图 12.16 中的控制加工(虚泡泡)？(Hatley 和 Pirbhdi 没有使用这个符号)

12.10 用你自己的语言描述事件流。

12.11 为 12.5 节讨论的复印机软件开发一个完整的流模型。你可以使用 Ward 和 Mellor 或 Hatley 和 Pirbhai 扩展，保证为系统开发一个详细的状态—变迁图。

12.12 为图 12－25 所示的 SafeHome 软件的分析模型完成处理说明，描述用户和系统间的交互机制。你增加的信息是否改变了 SafeHome 的流模型？如果改变了，是如何改变的？

12.13 一个大城市的公共工作部门决定开发一个“计算机化的”坑洼跟踪和修理系统(PHTRS)。当报告有坑洼时，它们被赋予一个标识号，并依据街道地址、大小(1 到 10)、地点(路中或路边等)、区域(由街道地址确定)和修理优先级(由坑洼的大小确定)储存起来。工单数据被关联到每个坑洼，其中包括地点和大小、修理队标识号、修理队的人数、被分配的装备、修理所用的时间、坑洼状况(正在工作、已被修理、临时修理、未修理)、使用填料的数量和修理的开销(由使用的时间、人数、使用的材料的装备计算得到)，最后，产生一个损害文件包括关于坑洼的被报告的损害信息，并包括居民的姓名、地址、电话号码、损害的类型和损害的钱数。PHTRS 是一个联机系统；请求可以交互式地进行。使用结构化分析为 PHTRS 建模。

12.14 要开发一个字处理软件。对这个应用领域进行几个小时的研究，并与你的同学进行 FAST 会谈(参看第 11 章)以开发需求。使用结构化分析为系统建立需求模型。

12.15 要开发汽油涡轮发动机的实时测试监控系统的软件。象思考题 12.14 一样进行。

12.16 要开发汽车组装工厂的生产控制系统的软件。象思考题 12.14 一样进行。

12.17 要开发一个视频游戏的软件。像思考题 12.14 一样进行。

12.18 与 4 到 5 个结构化分析 CASE 工具的厂商取得联系，复审他们的文献，撰写一篇简要的文章，总结将各个工具区别开来的一般性特征。

推荐阅读文献及其他信息源

已出版了很多关于结构化分析的书籍。大多数都全面地阐述了这个主题，但只有很少做的工作真正出色。DeMarco 的书[DEM79]仍然是对基本符号体系的最好的介绍。Hoffer et al.(Modern Systems Analysis and Design，Benjamin－Cummings，1996)、Modell(Systems Analysis，2nd edition，McGraw－Hill，1996)，Robertson 和 Robertson(Complete systemsAnalysis，twovolumes，Dorset House，1994)，Page－Jones(The Practical Guide to Structured systems Design，2nd edition，Prentice-Hall，1988)的书籍是有价值的参考资料。Yourdon 的书[YOU89]到今天仍然最广泛地阐述了这个课题。Kirner(“A Toolfor Analysis of Real－Time Specification Methods，”Software Engineering Notes，July 1993)提供了关于实时分析方法的出色的参考书目。Internet 新闻组 comp.realtime 常常包括对实时分析和规约方法的讨论。

在过去十年中结构化分析出现了很多变种。Cutts(Structured Systems Analysis and DesignMethodology，VanNostrandReinhold，1990)和Hares(SSADMfortheAdvancedPractitioner,Wiley，1990)描述了SSADM，这是在英国和欧洲广泛使用的结构化分析方法的一个变种。

Reingruber和Gregory(Data Modeling Handbook，Wiley，1995)以及Tillman[TIL93]提出了创建工业质量数据模型的详细教程。 Kim和Salvatore(“Comparing Data Modeling Formalisms，”Communications of the ACM，June 1995)对数据建模方法进行了出色的比较。Hay的一本有趣的书(Data Modeling Patterns，Dorset House，1995)提供了在很多不同的业务中出现的典型的数据模型“模式”。行为建模的详细处理可以在Kowal的书(Behavior Models: Specifying User′ sExpectations，Prentice—Hall，1992)中找到。

软件技术支持中心(STSC)提供了对于可用于分析建模的CASE工具的全面研究(Requirements Analysis and Design technology Report，March 1994)。STSC是位于犹他州Hill AirForce Base的政府资助的组织。

关于需求分析方法的简短参考书目和指向其他关于功能分析和质量功能部署的参考书目的指针可以在以下地址中找到：

http：//mijuno.larc.nasa.gov/dfc/biblio/reqengBiblio.html

许多CASE工具的生产者提供了包含结构化分析工具信息的Web站点。一组广泛的指针可以从以下地址中获得：

http：//www.qucis.queensu.ca/Software—Engineering/toolcat.html

结构化分析的简短教程可以参看：

http：//cscmosaic.albany.edu/～gangolly/ssal.html

关于软件分析建模的最新WWW参考文献列表可在http://www.rspa.com找到。

① 这种区别可将数据对象与作为本书第四部分要讨论的面向对象范型的部分的“类”或“对象”区分开来。

① 为避免不明确性，要考虑标记关系的方式。例如，如果没有考虑双向关系的语境，图12.4b可能被错误地解释为意味着书订购书店。在这种情况下，改述是必要的。

① 在某些情况下可能不需要修理行为。

② 请注意，为表示关系、基数和形态，已提出了许多不同的记号。12.3.4节将出现另一种记号。

③ 今天，术语“数据实体”或“实体”已被“数据对象”所取代。然而，术语“实体”仍是对象—关系对的图形记号名称的一部分。

① 基本记号的扩展将在12.4.2节讨论。

① 也称为“需求字典”。

① 生产单元用于工厂自动化应用中。它包括计算机和自动化的机器（例如机器人、NC机器、特定的工作固件），在计算机的控制下执行离散的生产操作。

① 也可以使用状态变迁表来替代图。附加的信息请参看[HAT87]。

① 为说明的清晰，数据流箭头被加了阴影，并保留在图中。在实践中，它们都要被去掉。

① 程序设计语言经常被用作过程设计记号，将在第 14 章详细描述。

① 事实上，数据字典可以看作一个更大的CASE库的一个元素。这将在第 29 章讨论。

① 软件需求规约的形式化方法在第 24 章详细讨论。

第 13 章　设计概念和原则

在任何工程化产品或系统的开发阶段中，设计是第一步。它可以定义为“为了能够足够详细地定义一种设备、一个处理或一个系统，以便保证其物理实现，而应用各种技术和原则的过程。”[TAY59]。

设计者的目标是生成一个随后要构造的实体的一种模型或表示。开发模型的过程综合了基于构造类似实体的经验的直觉和判断、一系列指导模型演化路径的原则和直观推断、一系列判断质量的标准以及导出最终设计表示的迭代过程。

象其他学科中的设计方法一样，软件设计随着新的方法、更好的分析和更广泛的理解的引入而不断地变化着。与机械或电子设计不同的是，软件设计在它的演化过程中处于一种相对早期的阶段。我们对软件设计(相对于“编程”或“代码书写”)给予正式考虑仅仅才三十年，因此，软件设计方法学缺少那些更经典的工程设计学科所具有的深度、灵活性和定量性。但是，软件设计的方法是存在的；设计质量的标准是能够获得的；设计符号体系也是能够应用的。在本章中，我们探讨可以应用于所有软件设计的基本概念和原则。第 14 章和第 21 章考察各种软件设计方法。

13.1 软件设计和软件工程

软件设计处于软件工程过程中的技术核心位置，并且它的应用不考虑所使用的软件过程模型。软件设计开始于对软件需求进行分析和规约之后，它是构造和验证软件所需的三项技术活动—设计、代码生成和测试—之一，每一项活动都最终导致经过验证的计算机软件的方式变换信息。

分析模型(第 12 章)的每一个元素均提供了创建设计模型所需的信息。软件设计中的信息流表示在图 13－1 中，通过数据、功能和行为模型展示的软件需求被传送给设计阶段，使用许多设计方法(在后面章节中讨论)中的一种，设计阶段产生数据设计、体系结构设计、接口设计和过程设计。

数据设计将分析时创建的信息域模型变换成实现软件所需的数据结构。在实体—关系图中定义的数据对象和关系以及数据字典中描述的详细数据内容为数据设计活动奠定了基础。

结构设计定义了程序的主要结构元素之间的关系。这种设计表示—计算机程序的模块框架—可以从分析模型和分析模型中定义的子系统的交互导出。

接口设计描述了软件内部、软件和协作系统之间以及软件同人之间如何通信。一个接口意味着信息流(如数据和/或控制流)，因此，数据和控制流图提供了接口设计所需的信息。

过程设计将程序体系结构的结构元素变换为对软件构件的过程性描述。从PSPEC，CSPEC 和 STD 获得的信息是过程设计的基础。

我们在设计时作出的决策最终将会影响软件构造是否成功，更重要的是会决定，软件维护的难易程度，但是，为什么设计如此重要呢？

软件设计的重要性可以用一个词来表达—质量。设计是在软件开发中形成质量的地方，设计为我们提供了可以用于质量评估的软件表示，设计是我们能将用户需求准确地转化为完整的软件产品或系统的唯一方法。软件设计作为所有软件工程和软件维护步骤的基础，没有设计，我们将冒构造出不稳定系统的风险—稍作改动就会失败;难于测试的系统;直到软件工程过程后期才能评估系统的质量，到那时时间已不够并且已经花销很多经费。

13.2 设计过程

软件设计是一个迭代的过程，通过它需求被变换为用于构造软件的“蓝图”。初始时，蓝图描述了软件的整体视图，也就是说，设计在高的抽象层次上表示—在该层次可以直接追踪到特定数据、功能和行为需求。随着数据迭代的开始,后续的精化将导致更低抽象级别的设计表示,这些表示仍然能够追踪到需求，但是连接更微妙了。

13.2.1 设计和软件质量

在整个设计过程中,演化的设计的质量可以通过在第 8 章讨论的一系列正式技术复审或设计追踪审查来评估。McGlanghlin[McG91]提出了可以指导良好设计演化的三个特征：

• 设计必须实现所有包含在分析模型中的明显需求,并且必须满足客户希望的所有隐式需求。

• 对于那些生成代码和那些进行测试并随后维护软件的人而言，设计必须是可读的，可理解的。

• 设计应该提供软件的完整面貌，这与从某个实现视角看到的数据、功能、和行为域有关。

这些特征中的每一个实际上都是设计过程的目标。但是如何达到这些目标呢？

为评价一项设计表示的质量，我们必须建立良好的设计技术标准，在本章的后面部分，我们将详细讨论设计质量标准，现在，我们给出下面的指南：

1.设计应该展示一种层次性组织，从而使得可以有指导性地使用软件元素间的控制。[①]

2.设计应该模块化，也就是说软件应该逻辑地划分成完成特定功能和子功能的构件。

3.设计应该既包含数据抽象，也包含过程抽象。

4.设计应该导出具有独立功能特征的模块(例如，子例程或过程)。

5.设计应该导出降低模块和外部环境间复杂连接的接口。

6.设计应该通过使用由软件需求分析过程中获得的信息导出要驱动的可重复的方法。

这些标准不是偶然获得的，软件设计过程通过应用基本设计原则、系统化的方法学和完全的复审来促进良好的设计。

13.2.2 软件设计的演化

软件设计的演化是一个历经以往三十年的连续过程。早期的设计工作集中在模块化程序的开发标准[DEN73]和自顶向下求精软件结构的方法[WIR71]，设计定义的过程方面发展成一种称为结构化程序设计的理论[DAH71，MIL72]，以后的工作提出了将数据流[STE74]或数据结构[JAC75，WAR74]转化为设计定义的方法。近期的设计方法(例如参考文献[JAC92]和[GAM95])提出一种面向对象方法导出的设计。

在上述工作中出现的许多设计方法，正在被整个产业界应用。同第 12 章中提到的分析方法一样，每一种软件设计方法引入了独特的启发信息和符号体系，以及有些狭隘的关于什么特性会影响设计质量的观点。然而，这些方法都具有一些共同特性：(1)一种用于将分析模型变换到设计表示的机制，(2)用于表示功能性构件及其接口的符号体系，(3)用于求精和划分的启发信息，以及(4)质量评价的指南。

不管使用的是什么设计方法，软件工程师应该在数据、体系结构、接口和过程设计方面应用一系列基本原则和概念。

13.3 设计原则

软件设计既是过程又是模型。设计过程是一系列迭代的步骤，它们使设计者能够描述要构造的软件的所有侧面，然而，要注意的是，设计过程不仅仅是一本

菜谱，创造性的技能、以往的经验、对于什么能形成“良好”软件的感觉、以及对质量的全部责任是设计成功的关键因素。

设计模型和建筑师的房屋设计图是类似的，它首先表示出要构造的事物的整体(例如，房屋的三维表示)，然后逐渐精化事物，以提供构造每个细节(例如，管道布置)的指南，类似地，软件的设计模型提供了计算机程序的一系列不同视图。

基本的设计原则使软件工程师能够为设计过程导航。Davis[DAV95]提出了一系列[①]软件设计的原则。下面的列表中对它们作了些修改和扩充：

• 设计过程不应该受“隧道视野”的限制。一名好的设计者应该考虑替代的手段，根据问题的要求，可用 13.4 节提到的设计概念来判断完成工作的资源。

• 设计对于分析模型应该是可跟踪的。因为设计模型的单独一个元素经常会跟踪到多个需求上，所以对设计模型如何满足需求进行的追踪是必要的。

• 设计不应该从头做起。系统是使用一系列设计模式构造的，很多模式很可能在以前就遇到过。这些模式通常被称为可复用设计构件，应该总是作为一切都从头开始的方法的一种替代选择。时间短暂而资源有限！设计时间应该投入到表示真正的新思想和集成那些已有模式上去。

• 设计应该缩短软件和现实世界中问题的“智力距离”[DAV95]，也就是说，软件设计的结构应该(尽可能)模拟问题域的结构。

• 设计应该表现出一致性和集成性。如果一项设计整体上看上去象是一个人完成的，那它就是一致的。在设计工作开始之前，设计小组应该定义风格和格式的规则，如果注意定义了设计构件之间的接口，那么，设计就是集成的。

• 设计应该构造以适应修改。下一节中讨论的许多概念使设计能实现这项原则。

• 设计应该构造以使得即使遇到异常的数据、事件或操作条件时也能够平滑、轻巧地降级。设计良好的计算机程序应该从不“彻底崩溃”，它应该设计为适应异常的条件，并且即使它必须中止处理时，也要采用优雅的方式。

• 设计不是编码，编码也不是设计。即使在为程序构件构造详细的过程设计时，设计模型的抽象级别也比源代码要高，在编码级别上作出的唯一设计决策是描述能使过程性设计被编码的小的实现细节。

• 在创建设计时就应该能够评估质量，而不是在事情完成之后。有许多设计概念(13.4 节)和设计方法(第 19 和 23 章)可以帮助设计者评估质量。

• 应该复审设计以减少概念性(语义性)错误。有时人们在复审设计中倾向于注重细节，只见树木不见森林。在关注设计模型的语法之前，设计者应该确保已经检查过设计的主要概念性元素(忽略、含糊性、不一致性)。

正确应用上述设计原则时，软件工程师创建的设计就会展现出外部和内部的质量因素[MEY88]。外部质量因素是那些用户能轻易观察到的软件特性(例如，速度、可靠性、正确性、可用性)[①]；内部质量因素对软件工程师是重要的，它们能导致技术角度上的高质量设计。要取得内部质量因素，设计者必须理解基本的设计概念。

13.4 设计概念

在过去 30 年中发展起来了一套基本的软件设计概念。尽管对于每一种概念的感兴趣程度在几十年中一直变化着，但它们都经历了时间的考验，每一种概念都为设计者提供了应用软件更加复杂的设计方法的基础。它们可以帮助软件工程师回答下述问题：

• 能使用什么标准将软件划分为单个构件？

• 如何将功能或数据结构与软件的概念性表示分离开？

• 是否存在定义软件设计的技术质量的统一标准？

M.A.Jackson 曾经说过：“软件工程师的智慧的体现开始于对程序工作和程序正确性之间的区别的识别”[JAC75]，基本的软件设计概念为“使程序正确”提供了必要的框架。

13.4.1 抽象

当我们考虑任何问题的模块化解决方案时，可以给出许多抽象级别。在最高的抽象级别中，解决方案使用问题环境的语言来进行概括性的术语描述，在低一些的抽象级别中，会有更加面向过程的倾向。为了描述解决方案，要一同使用面向问题的术语和面向实现的术语，最后，在最低的抽象级别中，用能够直接实现的方式描述解决方案。Wasserman[WAS83]提供了一种有用的定义：

“抽象”的心理学观念使人能够集中于某个一般性级别上的问题，而不去考虑无关的低层细节；使用抽象还能使人能用问题环境中熟悉的概念和术语工作，而不必将它们转换成一种不熟悉的结构…

软件工程过程中的每一个步骤都是软件解决方案抽象级别上的求精。在系统工程时，软件划分为基于计算机的系统的一种元素；在软件需求分析时，使用“问题环境中熟悉的”术语来描述软件解决方案；当我们在设计过程中时，降低了抽象级别，最终，当源代码生成时，就到达了最低的抽象级别。

当我们在不同的抽象级别中移动时，我们努力去创建抽象的过程和数据。抽象过程是一个命名的指令序列，它具有特定和有限的功能，抽象过程的一个例子是单词“开”(门)。“开”蕴含了一长串过程性步骤(例如，走向门；伸出手并抓住把手；转动把手并拉门；离开移动的门等)。抽象数据是命名的数据集合，它描述了一个数据对象。

在上面提到的抽象过程“开”的情形中，我们可以定义一个称作“门”的抽象数据。同任何抽象数据一样，门的抽象数据会包含一组描述门的属性(例如，门的类型、转动方向、打开机制、重量和维度)，抽象过程“开”将使用抽象数据“门”的属性中包含的信息。

许多编程语言(例如，Ada，Modula，CLU)提供了创建抽象数据类型的机制，例如，Ada 的 Package(包)就是一种程序设计语言机制，它提供了对数据和过程抽象的支持。初始的抽象数据类型被用作能够被实例化的其他数据结构的模板或类属数据结构。

抽象控制是软件设计中使用的第三种抽象形式。同抽象过程和抽象数据类似，抽象控制蕴含了不刻划内部细节的程序控制机制，抽象控制的一个例子是用来在操作系统中协调活动的同步信号量[KAI83]。

13.4.2 求精

逐步求精是由 Niklaus Wirth[WIR71]最初提出的一种自顶向下设计策略，程序的体系结构是通过过程细节的逐步层次精化开发的，层次结构通过逐步地分解功能的宏观声明，直至形成程序设计语言的语句而开发出来。Wirth[WIR71]给出了该概念的简要定义：

在(求精的)每一步中，给定程序的一条或几条指令被分解成更详细的指令，这种连续的对规约的分解或求精，直至所有指令都用某种底层的计算机或程序设计语言表达时才中止…。当任务被精化时，数据可能也要被精化、分解或结构化，而且，并行地精化程序和数据规约是很自然的。

每一步求精都蕴含了某些设计决策，对于程序员而言，注意根本性的标准(针对设计决策)和可替换的解决方案是重要的…

Wirth 提出的程序精化过程同需求分析时所用的过程精化和划分是类似的，区别在于所考虑的是在实现细节级别上，而不是方法。

求精实际上是一个推敲的过程。我们从高抽象级别定义的功能描述(或信息描述)开始，也就是说，该描述概念性地说明了功能或信息，但没有提供有关功能内部工作的情况或信息的内部结构。求精使设计者去推敲原始声明，并在后续的求精(推敲)活动中提供越来越多的细节。

抽象和求精是互补的概念，抽象使得设计者能够刻划过程和数据而同时忽略低层细节，求精有助于设计者在数据过程中揭示低层的细节，两个概念均帮助设计者在设计演化中构造出完整的设计模型。

13.4.3 模块化

计算机软件的模块化概念已经被采纳了将近四十年，软件体系结构(在13.4.4节中描述)体现了模块化；即软件被划分成独立命名和可独立访问的被称作模块的构件，它们集成到一起满足问题需求。

有人提出“模块化是软件的单个属性，它使程序能被理性地管理”[MYE78]。读者无法简单地掌握单块集成电路式的软件(例如，由单一模块构成的大程序)，控制路径的数量、引用的跨度、变量的数量和整体的复杂性会使对软件的理解接近于不可能。为说明这一点，考虑下面的论据，它们是基于对人解决问题的观察而提出的。

设C(x)是描述问题x复杂性的函数，E(x)是定义解决问题x所需工作量(按时间计算)的函数。对于两个问题p_1和p_2，如果

$$C(p_1)>C(p_2) \qquad (13.1a)$$

那么

$$E(p_1)>E(p_2) \qquad (13.1b)$$

作为普遍的情况，这个结构直观上是显然的。解决困难问题需要花费更多时间。

通过对人解决问题的实验又发现了另一个有趣的特性。即，

$$C(p_1+p_2)>C(p_1)+C(p_2) \qquad (13.2)$$

方程式(13.2)意味着p_1和p_2组合后的复杂性比单独考虑每个问题时的复杂性要大。考虑方程式(13.2)和方程式(13.1)隐含的条件，可以得出

$$E(p_1+p_2)E(p_1)+E(p_2) \qquad (13.3)$$

这引出了“分而治之”的结论—将复杂问题分解成可以管理的片断会更容易。不等式(13.3)表达的结果对模块化和软件有重要意义，事实上，它是模块化的论据。

从不等式(13.3)可能得出的结论是：如果我们无限制地划分软件，那么开发它所需的工作量会变得小到可以忽略！不幸的是，其他因素开始起作用，使得这个结论(遗憾地)无效了。如图13－2所示，开发单个软件模块所需的工作量(成本)的确随着模块数量的增加而下降，给定同样的需求，更多的模块意味着每个

模块的尺寸更小，然而，随着模块数量的增长，集成模块所需的工作量(成本)也在增长。这些特性形成了图中所示的总成本或工作量曲线。存在一个模块数量—M—可以导致最小的开发成本，但是，我们无法确切地预测 M。

图 13－2 所示的曲线为考虑模块化提供了有用的指导，我们应该进行模块化，但应注意保持在 M 附近，应该避免过低或过高的模块性，但是，我们如何知道“M 的附近”呢？我们应该怎样将软件划分成模块呢？回答这些问题需要理解本章后面提出的其他设计概念。

在考虑模块化时，出现了另一个重要问题：我们如何定义给定大小的一个合适模块？答案在于用来在系统中定义模块的方法(有关设计方法将在第 14 和第 21 章中讨论)。Meyer[AEY88]定义了五个标准，它们用我们可以根据定义有效的模块化系统的能力来评价一种设计方法：

模块可分解性。如果一种设计方法提供了将问题分解成子问题的系统化机制，它就能降低整个系统的复杂性，从而实现一种有效的模块化解决方案。

模块可组装性。如果一种设计方法使现存的(可复用的)设计构件能被组装成新系统，它就能提供一种不一切从头开始的模块化解决方案。

模块可理解性。如果一个模块可以作为一个独立的单位(不用参考其他模块)被理解，那么它就易于构造和修改。

模块连续性。如果对系统需求的微小修改只导致对单个模块，而不是整个系统的修改，则修改引起的副作用就会被最小化。

模块保护。如果模块内出现异常情况，并且它的影响限制在模块内部，则错误引起的副作用就会被最小化。

最后，要注意的是：即使系统的实现必须是“单块集成电路式的”，它也可以采用模块化的设计。有些情况(例如，实时软件，嵌入式软件)下，由子程序(例如，子例程，过程)造成的即使是比较小的速度和内存开销也是无法接收的。在这种情况下，软件能够而且也应该采用模块化设计作为一种指导思想，可以“内嵌地(in－line)”开发代码。尽管程序源代码看上去可能没有模块性，但这种思想依然有效，而且程序对提供模块化系统非常有益。

13.4.4 软件体系结构

软件体系结构意指“软件的整体结构和这种结构提供给系统在概念上的整体性的方式”[SHA95a]。以最简单的形式为例，体系结构是程序构件(模块)的层次结构、构件间交互的方式、以及构件使用的数据的结构，然而，在更广泛的意义上讲，“构件”可以被推广来代表主要的系统元素和它们的相互交互。[①]

软件设计的一个目标是导出系统的体系结构表示，这个表示作为一个框架，指导更详细的设计活动，一组体系结构模式使得软件工程师能够复用设计层的内容。

Shaw 和 Garlan[SHA95a]描述了一组性质，它们应该是体系结构设计中的一部分：

结构性质。体系结构设计表示的这一方面定义了系统的构件(例如，模块、对象、过滤器)以及这些构件被打包和相互交互的方式，例如，将对象打包以便封装数据和操纵数据的处理，并且通过方法调用进行交互(第 19 章)。

附加的功能性质。体系结构设计描述应该说明设计的体系结构如何实现对性能、功能、可靠性、安全性、适应性、和其他系统特征的需求。

相关系统族。体系结构设计应该基于可复用的模式进行描述，这些模式通常会在类似系统族的设计中遇到，重要的是，设计应该能够复用体系结构的构造块。

如果给定这些性质的规约，体系结构设计可以使用一种或多种不同模型[CAR95]来表示。结构模型将体系结构作为程序构件的有组织的集合来表示；框架模型通过试图标识类似应用软件中遇到的可复用的体系结构设计框架(模式)而提高设计抽象级别；动态模型强调程序体系结构的行为侧面，指明结构或系统配置作为外部事件的函数将如何变化；过程模型注重系统必须适应的业务或技术过程；最后，功能模型可以用来表示系统的功能层次结构。

人们已经开发出一系列不同的体系结构描述语言(ADL)来表示上面提到的模型[SHA95b]，尽管已经提出了许多不同的 ADL，大多数都提供了描述系统构件和构件互联方式的机制。

13.4.5 控制层次

控制层次，也称作程序结构，代表了程序构件(模块)的组织(常常是结构化的)，并表示了控制的层次结构，它并不表示软件的过程性，例如进程序列、事件/决策的顺序、或重复的操作。

许多不同的标记被用来表示控制层次，最常用的是表示层次结构的树形图(图 13－3)，然而，像Warnier－Orr[ORR77]和Jackson[JAC83]图一类的其他符号体系也可以同样有效地拿来使用[①]。为了有助于后面有关结构的讨论，我们定义一些简单的方法和术语。在图 13－3 中，宽度和深度分别提供了对控制级别的数量和整体控制跨度的指示，扇出(fan－out)衡量的是被一个模块直接控制的其他模块的数量，扇入(fan－in)指出有多少模块直接控制一个给定模块。

模块间的控制关系是通过下述方法表达的：控制其他模块的模块被称作上级模块；相对的被其他模块控制的模块被称作控制者的从属模块[YOU79]，例如，如图 13－3 所示，模块 M 是模块 a、b 和 c 的上级模块，模块 h 是模块 e 的从属

模块和最终是模块 M 的从属模块。宽度方向的关系(例如，模块 d 和 e 之间)，尽管在实际中可能有表达，但这种关系是不需要用明显的术语来定义。

控制层次还代表了两种略有不同的软件体系结构特征：可见性和连接性，可见性指明可以被调用或被给定构件用作数据的一组程序构件，即使是通过间接方式实现的。例如，在面向对象系统中的一个模块可以访问它所继承的很多属性，但其中只有一小部分可以拿来用，所有这些属性对该模块都是可见的。连接性指明被给定构件直接调用或用作数据的一组构件，例如，直接导致另一个模块开始执行的模块是连接到该模块的。②

13.4.6 结构划分

程序结构应该被水平和垂直地划分，如图 13—4a 所示，水平划分为每个主要程序功能定义了分离的模块结构分支，用深色阴影表示的控制模块被用来协调程序功能之间的通讯和运行。水平划分的最简单方法定义了三个划分—输入、数据变换(通常称作处理)和输出。对体系结构进行水平划分获得有关她的许多特殊的优点：

- 产生易于测试的软件。
- 产生易于维护的软件。
- 产生副作用更小的传播。
- 产生易于扩展的软件。

由于主要的功能相互分离，修改变得更加简单，对系统的扩展(常见情况)往往变得更加容易完成，且没有副作用。从消极的方面看，水平划分常常通过模块接口传递更多的数据，因而可能会使程序流的整体控制复杂化(如果处理需要从一个功能快速移动到另一个功能)。

垂直划分(图 13－4b)，常常称作因子划分，要求发生在程序体系结构中(决策)，且工作应该自顶向下分布，顶层模块应该执行控制功能，而少作实际处理工作，在层次结构中位于低层的模块应该是工作者，它们完成所有的输入、计算和输出任务。

程序体系结构的变化的性质验证了垂直划分的必要性，(高层的)控制模块的变化很可能将副作用传播到下层的从属模块，对工作者模块的修改，假定它在结构中位于下层，就不太可能引起副作用的传播。通常情况下，对计算机程序的修改会在输入、计算或变换和输出的修改间循环，程序的整体控制结构(例如，它的基本行为)不太可能修改，由于这个原因，垂直划分的体系结构在作修改时更不容易受到副作用的影响，因而更加易于维护－这是一项关键的质量因素。

13.4.7 数据结构

数据结构是单个数据元素之间逻辑关系的表示。因为信息结构必将会影响最终的过程设计，对于软件体系结构的表示而言，数据结构同程序结构同样重要。

数据结构决定了信息的组织、访问方法、关联程度、可替换的处理方法，有关这些专题已经有完整的文章做了介绍(例如参考文献[AHO3]、[KRU84]和[GAN89])，而完整的讨论已超出本书范围，然而理解可用的组织信息的经典方法和信息层次结构的基础概念是重要的。

数据结构的组织和复杂性仅仅受设计者创造性的局限，不过，已存在有限数量的经典数据结构构成了更复杂结构的基本构造块。

标量项是最简单的数据结构，就象名字的含义一样，标量项代表一个单独的信息元素，它可以通过一个标识符来引用；即可以通过指定一个单独的存储地址来实现对它的访问。

当标量被组织成序列或相邻的组时，就形成了一个顺序向量，向量是数据结构中最常用的，并且开启了信息变量索引之门。

当顺序向量扩展到两维、三维、最终是任意维时，就创建了一个 n 维空间，最常见的 n 维空间是二维的矩阵，在大多数编程语言中，n 维空间被称作数组。

项、向量和空间可以按多种格式组织，链表是将不相邻的标量项、向量或空间以某种方式(称为节点)组织起来的数据结构，这种方式使得可以象处理列表一样处理结点，每一个节点包含适当的数据组织(例如，一个向量)和一个或更多个指示，列表中下一个节点存储位置的指针。在列表中的任意处都可以加入新节点，方法是重新定义指针以适应新的列表项。

使用上述的基本数据结构进行组合或构造可以得出其它的数据结构，例如，层次数据结构通过包含的标量项、向量和可能是 n 维的空间的多链链表来实现。层次结构通常在需要信息分类和关联的应用中将遇到，分类蕴含了根据某种类属种类(例如，所有 64 位微处理器)将信息分组。

关联意味着从不同类别中关联信息的能力，例如，在微处理器类中找出所有价格低于 100 美元(价格子类)，运行在 100MHz(时钟周期子类)以内并且有美国厂商制造(供应商子类)的条目。

要注意的是，数据结构象程序结构一样可以在不同的抽象级别上表示出来，例如，一个堆栈就是一个概念性的模型，它既可以作为向量实现，也可以作为链表实现。根据设计细节的层次，可能指定堆栈的内部工作机制，也可能不作指定。

13.4.8 软件过程

程序结构定义了控制层次，而不管处理和决策的顺序。软件过程着重于每个模块单个的处理细节，过程必须提供处理的精确定义，包括事件的顺序、准确的决策点、循环操作、甚至还有数据组织/结构。

当然，结构和过程之间有关系，为每个模块指明的处理必须包括对所有从属模块的引用，也就是说，软件的过程表示是分层的，如图 13－5 所示。

13.4.9 信息隐蔽

模块化的概念将所有软件设计者引向一个问题：“如何分解软件的解决方案，以便获得最佳的模块集？”信息隐藏的原则[PAR72]提出模块的“特征在于每个模块都对其他模块隐蔽的设计决策”，换句话说，模块应该设计成其中包含的信息(过程和数据)对不需要这些信息的其他模块是不可访问的。

隐蔽的含义是：有效的模块化可以通过定义一组独立模块来实现，这些模块相互之间只交流实现软件功能必需的信息。抽象有助于定义组成软件的过程(或信息)实体。隐蔽定义并加强了对模块内部过程细节或模块使用的任何局部数据结构的访问约束[ROS75]。

将信息隐藏用作模块化系统的设计标准为在测试及以后维护中需作修改时提供了极大的方便，因为大多数数据和过程对软件其他部分是隐蔽的，修改时无意中引入的错误就不太可能传播到软件中其他位置。

13.5 有效的模块设计

前面章节中描述的基本设计概念都是用来阐述模块化设计的，事实上，模块化已经成为所有工程学科中广为接收的方法。模块化设计降低了复杂性(参见 13.4.3 节)、方便了修改(软件可维护性的关键方面)、并且通过鼓励系统的不同部分的并行开发简化了实现。

13.5.1 功能独立性

功能独立性是模块化和抽象及信息隐藏概念的直接产物。Parnas[PAR72]和 Wirth[WIR71]在关于软件设计的里程碑性文献中提到了促进模块化的求精技术，以后，Stevens，Myers 和 Constantine[STE74]的工作巩固了这个概念。

功能独立性是通过开发具有“专用”功能的模块和“反对”同其他模块的过分交互而实现的，换句话说，我们希望将软件设计成每个模块完成需求的一个特定子功能，并且从程序结构的其他部分观察时具有简单的接口。有理由询问为什么独立性是重要的，因为功能可以被划分，并且接口简化了(考虑由小组完成开发时的分支情况)，有效模块化的软件(例如，独立模块)更易于开发。因为由设计/编码修改引起的副作用受到局限，错误传播被减小，以及模块复用成为可

能，因此，独立的模块更易于维护(及测试)。总而言之，功能独立性是良好设计的关键，而设计又是软件质量的关键。

独立性是通过两项质量标准来衡量的：内聚和耦合。内聚是模块相对功能密度的度量，耦合是模块间相对独立性的度量。

13.5.2 内聚

内聚是 13.4.9 节描述的信息隐蔽功能的自然扩展。内聚的模块在软件过程中完成单一的任务，同程序其他部分执行的过程交互很少，简而言之，内聚模块(理想情况下)应该只完成一件事。

内聚可以用图 13－6 所示的“谱系”来表示，尽管谱系的中段通常也是可以接收的，但我们总是尽量争取高内聚。内聚的刻度是非线性的，也就是说，低端内聚比中段内聚要“差”得多，而中段内聚几乎和高端内聚一样“好”，在实际上，设计者不必关心对特定模块的内聚分类，而应该理解整体概念并且在设计模块时应该避免低内聚。

为了说明(有些开玩笑的)谱系的低端，我们讲述下面的故事：

在六十年代后期，大多数 DP 经理开始认识到模块化的价值，不幸的是，许多现有的程序都是单块的－例如，20000 行没有文档的 Fortran 程序加上 2 500 行子程序！为了将一个大型计算机程序达到一种艺术境界，一位经理命令她的员工对程序进行模块化，而这些工作要在“你的业余时间”完成。

在压力下，一名员工(天真地)询问每个模块的合适长度，回答是“75 行代码”，然后，他得到了一只红色的笔和一把尺子，测量 75 行源代码的线性距离，随后在源程序清单上一条接一条地划上红色的线，每一条红线表示一个模块的边界。这种技术就类似于去开发具有偶然内聚的软件！

执行一组彼此松散相关的任务的模块就称作偶然内聚，执行逻辑相关的任务模块(例如，产生所有输出而不管类型的模块)就是逻辑内聚，当模块包含着由于必须在相同时间段内执行而相关的任务时，该模块就表现为时间内聚。

以一个低内聚为例子，如有一个工程分析包错误执行处理的模块，当计算的数据超出预定义的边界时调用该模块，它执行下列任务：(1)根据初始计算的数据，计算补充数据；(2)在用户的工作站上生成错误报告(以图形方式)；(3)执行用户要求的跟踪计算；(4)更新数据库；以及(5)使选择后续处理的菜单有效。虽然前面的任务是松散相关的，但是，每一项都是独立的功能实体，并且最好作为独立的模块完成。将这些功能组合在单一的模块中只会起到在修改上述任一处理任务时增加错误传播可能性的作用。

中度内聚在模块独立程度上是彼此接近的。当模块的处理元素相关，并必须按指定顺序执行时，就存在过程内聚；当所有处理元素集中在某个数据结构的一块区域上时，通信内聚就产生了。高内聚是以执行单独过程任务的模块为特征的。

如前所述，确定内聚的精确级别是不必要的，重要的该是尽量争取高内聚和识别低内聚，这样就可以修改软件设计来实现功能独立。

13.5.3 耦合

耦合是程序结构中模块相互连接的测度。同内聚类似，耦合可以用图 13－7 所示的谱系表示，耦合依赖于模块间接口的复杂性、引用或进入模块所在的点、以及什么数据通过接口传递。

在软件设计中我们力争尽可能低的耦合。模块间的简单连接导致软件易于理解，并且当错误发生于某个位置并在系统中传播时更少受“涟漪效应”的影响。

图 13－8 提供了不同类型耦合模块的例子。模块 a 和 b 是不同模块的子模块，相互之间无关，因而没有直接耦合发生；模块 c 是模块 a 的子模块，并通过常规的参数表来访问，数据通过该列表传递，只存在简单的参数表(例如，传递简单数据；存在项的一对一对应)，这部分结构中就体现了低耦合(谱系中的数据耦合)；当数据结构的一部分(而不是简单的参数)通过模块接口传递时就会发现数据耦合的一个变体，被称作印记(stamp)耦合，这种情况出现在模块 b 和 a 之间。

在中度耦合级别上，耦合的特性是在模块间传递控制。控制耦合在大多数软件设计中非常普遍，如图 13－8 所示，在这里，“控制标记”(在从属或上级模块中控制决策的变量)在模块 d 和 e 之间传递。

当模块连接到软件外部环境上时会发生相对高度的耦合，例如，I/O 将模块耦合到特定设备、格式和通信协议上。外部耦合是重要的，但应该局限在结构中少量的模块上。高耦合还发生在许多模块引用一个全局数据区时。共用耦合，就象这种模式的名称一样，显示在图 13－8 中，模块 c、g 和 k 都访问一块全局数据区中的数据项(例如，一个磁盘文件、 Fortran COMMON 或 C 语言中的外部数据类型)，模块 c 初始化该数据目，以后模块 g 重新计算并更新该数据目，让我们设想出现了错误，并且 g 错误地更新了该数据目，再以后的处理中，模块 k 读取该数据目，试图处理它并失败了，引起软件异常中止。引起软件异常中止的表面原因是模块 k，实际的原因在于模块 g。在具有很多共用耦合的结构中诊断错误是费时和困难的，然而，这并不意味着使用全局变量就一定是“坏的”，它的含义是软件设计者应该注意共用耦合的潜在后果，并特别注意防范这些后果。

耦合的最高层次，内容耦合，出现在一个模块使用在另一个模块的边界中维护的数据或控制信息的情况下，其次，内客耦合出现在分支并跳转到模块中间时，这种模式的耦合能够并且应该避免。

上面讨论的耦合模式的出现是由于设计决策是在开发程序结构时做出的，然而，各种各样的外部耦合也可能在编码时引入，例如，编译器耦合将源代码同编译器的特定(常常是不标准的)属性联系到一起；操作系统(OS)耦合将设计和结果代码同操作系统的“钩子”联系到一起，而当 OS 发生变化时这会造成严重的破坏。

13.6 针对有效模块化的设计启发

一旦开发了程序结构，就可以通过应用本章前面介绍的设计概念实现有效的模块化。程序体系结构是根据本节描述的一组启发方法(指南)来处理的。

Ⅰ.评估程序结构的“第一次迭代”以降低耦合并提高内聚。一旦开发了程序结构，为了增强模块独立性可以对模块进行向外或向内的突破，一个向外突破后的模块变成最终程序结构中的两个或多个模块，一个向内突破的模块是组合两个或多个蕴含待处理产物的模块。

当两个或多个模块中存在共同的处理构件时，可以将该构件重新定义成一个内聚的模块，这时常常形成向外突破的模块。在期望高耦合时，有时可以将模块向内突破，从而减少控制传递、对全局变量的引用和接口的复杂性。

Ⅱ.试图用高扇出使结构最小化；当深度增加时争取提高扇入。图 13－9 中阴影中所示的结构没有有效地使模块因子化，所有的模块都“平铺”在单个控制模块下，而上面非阴影中的结构显示出通常更合理的控制分布，结构采用椭圆外形，指明一系列控制层次以及低层的高度实用性的模块。

Ⅲ.将模块的影响限制在模块的控制范围内。模块 e 的影响范围定义成所有受模块 e 中决策影响的其他模块，模块 e 的控制范围是模块 e 的所有从属及最终的子模块。如图 13－9 所示，如果模块 e 作出的决策影响了模块 r，这就违反了规则Ⅲ，因为模块 r 位于模块 e 的控制范围之外。

Ⅳ.评估模块接口以降低复杂度和冗余并提高一致性。模块接口太复杂是软件错误的首要原因，接口应该设计成简单地传递信息并且应该同模块的功能保持一致，接口不一致(看上去是指无关的数据通过参数表或其他技术进行传递)是低内聚的表现。有问题的模块应该重新评估。

Ⅴ.定义功能可以预测的模块，但要避免过分限制的模块。当模块可以作为黑盒对待时就是可预测的；即同样的外部数据可以在不考虑内部处理细节的情况下生成。[①]具有内部“存储器”的模块可能是不可预测的，除非使用时加以注意。

将处理限制在单个子函数中的模块体现出高内聚，而且为设计者支持。然而任意限制局部数据结构大小、控制流内选项或外部接口模式的模块将不可避免地需要维护来清除这些限制。

Ⅵ.**力争“受控入口”模块，避免“病态连接”。**这条设计原则针对内容耦合提出警告，当模块接口受到约束和控制时，软件易于理解，因而易于维护。病态连接是指指向模块中间的分支或引用。

Ⅶ.**根据设计约束和可移植性需求，对软件进行打包。**打包的含义是指用来为特定处理环境组装软件的技术。设计约束有时要求程序在内存中“覆盖”自己，当这种情况必须发生时，设计结构可能必须重新组织，以便根据重复的程度、访问的频度和调用的间隔将模块分组，此外，可选的或“只用一次”的模块可以在结构中分离出来，以便有效的覆盖它们。

13.7 设计模型

本章讨论的设计原则和概念为创建包括数据、体系结构、接口和过程的设计模型建立了基础。同前面的分析模型类似，在设计模型中每种设计表示都同其他表示相关联，而且都可以追踪到软件需求。

在图 13－1 中，设计模型表示成金字塔，这种形状的象征意义是重要的，金字塔是极为稳固的物体，它具有宽大的基础和低的重心。象金字塔一样，我们希望构造坚固的软件设计，我们通过用数据设计建立宽广的基础，用体系结构和接口设计建立坚固的中部，以及应用过程设计构造尖锐的顶部，从而创建出不会被修改之风轻易“吹倒”的设计模型。

有意思的是，有些程序员继续在隐晦地设计，从而导致过程设计发生在编码时，这就类似将金字塔倒立着放在地上——一种非常不稳固的设计结果，最微小的修改也可能导致金字塔(和这个程序)倾覆。

引入创建设计模型的方法在第 14 章和 21 章(关于面向对象系统)描述，每种方法都使设计者能够创建出符合基本原则并形成高质量软件的稳固设计。

13.8 设计文档

表 13－1 所示的文档可以用作设计规约的模板，每一个编号的段落都描述设计模型的不同侧面，在设计者精化他或她的软件表示时，设计规约的章节编号也就完成了。

表 13－1　设计规约大纲

Ⅰ.范围
　A.系统目标
　B.主要软件需求
　C.设计约束，限制
Ⅱ.数据设计
　A.数据对象和形成的数据结构
　B.文件和数据库结构
　　1.外部文件结构
　　a.逻辑结构
　　b.逻辑记录描述
　　c.访问方法
　　2.全局数据
　　3.文件和数据交叉索引
Ⅲ.体系结构设计
　A.数据和控制流复审
　B.得出的程序结构
Ⅳ.接口设计
　A.人机界面规约
　B.人机界面设计规则
　C.外部接口设计
　　1.外部数据接口
　　2.外部系统或设备接口
　D.内部接口设计规则
Ⅴ.(每个模块的)过程性设计
　A.处理说明
　B.接口描述
　C.设计语言(或其他)描述
　D.使用的模块
　E.内部设计结构
　F.注释/约束/限制
Ⅵ.需求交叉索引
Ⅶ.测试部分
　　1.测试方针
　　2.集成策略
　　3.特殊考虑
Ⅷ.特殊注解
Ⅸ.附录

第Ⅰ节(节号指设计规约大纲)中描述了设计工作的整体范围。这一节中包含的大部分信息来自系统规约和分析模型(软件需求说明)。

第Ⅱ节说明数据设计，描述了外部文件结构、内部数据结构以及连接数据对象到特定文件的交叉索引。第Ⅲ节是体系结构设计，它说明程序体系结构如何从分析模型导出，结构图(程序结构的表示)用来表示模块的层次结构。

第Ⅳ节和第Ⅴ节在开始接口和过程设计时出现。这里描述了外部和内部的程序接口以及人—机交互界面的详细设计，对模块—可独立引用的软件成分，例如子例程、函数或过程—使用英语的处理说明进行的描述，处理说明解释了模块的过程功能，然后过程设计工具用来将该说明翻译成结构化的描述。

设计规约的第Ⅵ节包含需求交叉索引，这个交叉索引矩阵的目的是(1)确保软件设计满足所有需求，和(2)指出哪些模块对特定功能的实现是关键的。

设计文档的第Ⅶ节包含了开发测试文档的第一个阶段。一旦软件结构和接口建立以后，我们就可以开发用来测试单个模块和集成整个包的指南。在有些情况下，详细的测试规程规约和设计平行出现，在这种情况中，这一节可以从设计规约中删除。

设计约束，例如物理内存限制或对特殊外部接口的需求，会对软件组装或打包提出特殊要求。由对程序覆盖、虚存管理、高速处理或其他因素的需求而引起

的特殊考虑会导致对从信息流或结构导出的设计的修改。第Ⅷ节描述了对软件打包的需求和考虑，其次，这一节描述了用来向客户传送软件的方法。

设计规约的第Ⅸ节包含着补充数据。算法描述、可替换的过程、表格数据、其他文档的摘要和其他相关信息作为特殊注解或独立的附录描述来出现。建议开发一份初步操作/安装手册，并将它包含在设计文档的附录中。

13.9 小结

设计是软件工程的技术核心。在设计时，对数据结构、程序体系结构、接口和过程细节进行了逐步的求精、复审和文档描述。设计导致能够评估质量的软件的表示。

过去三十年中提出了一系列基本软件设计原则和概念。设计原则在设计过程进行中指导软件工程师开展工作。设计概念提供了有关设计质量的基本标准。

(程序和数据的)模块化和抽象的概念使设计者能够简化并复用软件构件。求精提供了表示连续层次功能细节的机制。程序结构和数据结构对软件体系结构的整体视图做出了贡献，而过程提供了算法实现的必要细节。信息隐蔽和功能独立性为实现有效的模块化提供了途径。

我们用 Glenford Myers[MYE78]的话总结我们有关设计基本原理的讨论：

我们试图匆忙完成设计过程，以便在项目结尾时节省出足够的时间去发现由于我们匆忙完成设计过程而造成的错误来解决问题…。

基本准则是：不要匆忙地完成它！设计是值得花费精力的。

我们还没有为设计的讨论作结论，在下面一章中会讨论设计方法，这些方法同本章中的基本原理以及第 21 章中的其他方法相结合，就构成了软件设计的完整基础。

思考题

13.1 你在“编写”程序时对软件进行过设计吗？软件设计和编码的区别是什么？

13.2 开发三个其他的设计原则以便添加到 13.3 节中提到的原则中去。

13.3 提供三个抽象数据以及可以用来对它们进行操作的抽象过程的例子。

13.4 应用“逐步求精方法”开发下面一个或多个程序的三种不同层次的抽象过程：

a.开发一个支票打印机。给定美元数额，以支票上通常要求的格式打印出数额。

b.超越(transcendental)方程式根的迭代解法。

c.为操作系统开发一个简单的 round－robin 调度算法。

13.5 不等式(13.2)有没有可能取非真值？这样的情况对模块化的论证有何影响？

13.6 何时应该将模块设计实现成一个单片集成电路式的软件？如何做到这一点？性能是不是实现单片集成电路式软件的唯一要求？

13.7 为下面软件问题之一开发至少五层的抽象：

a.消费者银行应用软件

b.计算机图形应用软件的 3 维变换包

c.BASIC 语言解释器

d.两个自由的机器人控制器

e.你和你的导师同意的任何问题

随着抽象层次的降低，你的注意力会逐步集中，这样在最后的层次中(源代码)只需要描述单个任务。

13.8 获取 Parnas 的原始文章[PAR72]并总结用来例举将系统分解为模块的软件例子。在实现分解的过程中是怎样隐蔽应用信息的？

13.9 讨论下面两个概念的区别：作为有效模块化特性的信息隐蔽和模块独立性。

13.10 复审你最近的一些软件开发工作，并对每个模块分级(按照从 1－低到 7－高的刻度)。举出你最好和最差的工作。

13.11 许多高级程序设计语言将内部过程作为一种模块构件。这种构件怎样影响耦合？怎样影响信息隐蔽？

13.12 耦合和软件可移植性的概念有何关系？提供例子支持你的讨论。

13.13 讨论结构划分怎样能帮助提高软件可维护性？

13.14 开发因子化的软件体系结构的目的是什么？

13.15 用你自己的话描述信息隐蔽概念。

13.16 为什么将模块的影响范围限制在它的控制范围内是个好主意？

推荐阅读文献及其他信息源

由 Freeman 和 Wasserman(Software Design Techniques，4th edition，IEEE 1983)编写的文选中包含了对有关软件设计的重要文章的出色的历史性概述。这本指南再版了许多已经形成当今软件设计主流的基本经典文章。在 Myers[MYE78]，Peters[PET81]，Macro(Software Engineering：Concepts and Management，Prentice－Hall，1990)和 Sommerville(Software Engineering，Addison－Wesley，5th edition，1996)的书中可以找到有关软件设计基本原理的有益讨论。McConnell(CodeComplete，Microsoft Press，1993)对设计高质量计算机软件的实际方面作了精彩的讨论。

在 Martin 和 McClure 的书(Diagramming Techniques for Analysts and Programmers，Prentice－Hall，1985)中可以找到不同设计符号体系的介绍。Stevens(Software Design：Concepts andMethods，Prentice－Hall，1990)探讨了数据、结构和过程设计。Witt 和他的同事们(SoftwareDesign，Van Nostrand Reinhold，1993)也透彻的讨论了这个问题。Shaw 和 Garlan(SoftwareArchitectures，Prentice－Hall，1995)以及 Coplien 和 Schmidt 编写的一本文选(Pattern Languages ofProgram Design，Addison－Wesley，1995)中论述了围绕创建有效软件体系结构的设计问题。

对计算机软件和设计基本原理的严格的数学研究可以在 Jones(Software Development：ARigorurous Approach，Prentice－Hall，1980)，Wulf(Fundamental Structures of Computer Science，Addison－Wesley，1981)以及 Brassard 和 Bratley(Fundamentals of Algorithmics，Prentice－Hall，1995)的书中找到。这些文献都为我们对计算机软件的理解提供了必要的理论基础。Kruse(Data Structures and Program Design，Prentice－Hall，1994)和 Tucker et al.(Fundamental of Computing II：Abstraction，Data Structures，and Large Software Systems，McGraw－Hill，1995)提供了关于数据结构的有价值的信息。Card 和 Glass(Measuring Software Design Quality，Prentice－Hall，1990)从技术和管理两个方面讨论了设计质量的测度。

对设计问题的一般讨论可以在 Internet 新闻组 comp.software－eng 和其他许多新闻组中找到。

在下述站点上有对软件设计集成环境的讨论：

http：//www-ksl.stanford.edu/KSL Abstracts/KSL－91－73.html

各种有关软件设计的联机文章以及其他最新信息可以从软件设计协会得到：

http：//pcd.stanford.edu/asd/index.html

对软件体系结构已经进行了实质性研究，在下面的站点可以找到关于这个专题的出色性的技术指导：

http：//www.stars.reston.paramax.com/arch/guide.html

有关当前工作的链接可以在下面站点中找到：

http：//WWW.cs.cmu.edu/～shaw/Shawparts/ArchPubs.html

http：//WWW.sei.cmu.edu/technology/architecture

http：//WWW2.umassd.edu/SECenter/SAResources.html

有关软件设计的最新WWW文献列表可以在：http：//WWW.rspa.com中找到。

① 要注意的是良好的面向对象设计不必表现出这项特点。参见第 19 章和 21 章的详细信息

① 这里只提到Davis的设计原则的部分子集。有关更多的信息，参见文献[DAV95]。

① 有关质量因素的更详细讨论在第 18 章。

① 例如，客户/服务器系统的结构性构件在不同的抽象级别上有表示。有关详细信息参见第 28 章。

① 对于面向对象的设计（第 21 章），程序结构的概念不太明显。

① “黑盒”模块是一种过程抽象。

第 14 章　设计方法

设计通常被描述为一个多步的过程，其主要任务是从信息需求中综合出数据结构的表示、程序结构、接口特征和过程细节。Freeman 在[FRE80]中对设计有较为详细的描述：

设计是一项主要考虑进行重要决策的活动，这些决策通常都与结构有关。设计与编程都要考虑抽象信息表示，但其详细程度与编程有很大的不同。设计的结果是一个一致的、合理计划的程序表示，主要描述高层各部分的相互关系和低层所需的逻辑操作…

在前一章我们已经指出，设计是由信息驱动的。各种软件设计方法主要考虑分析模型中的三个域，因此数据、功能和行为三个域是整个设计创建活动的指南。

本章将讨论多种用于创建设计模型(见图 13－1)的各个层次的方法，本章的目标是提供一个系统地完成设计的方法，设计的结果就是构造软件的蓝图。

14.1 数据设计

数据设计是实施软件工程中的四个设计活动的第一个(有人也认为是最重要的一个)。由于数据结构对程序结构和过程复杂性都有影响，数据结构对软件质量的影响是很深远的。信息隐蔽和抽象数据的概念为数据设计提供了基础。

Wasserman 在参考文献[WAS80]中总结了数据设计的过程：

数据设计的主要活动是选择对需求定义和规约过程中找出来的数据对象(数据结构)的逻辑表示。选择过程可以包括对候选结构进行算法分析，以决定出效率最高的结构；选择过程也可以只使用一组模块(一个包)，在对象的某种表示上提供需要的操作。

设计中的另一个相关的活动是标识要直接作用于逻辑数据结构的程序模块，这样，各个数据设计决策的影响域就受到了约束。

无论采用哪种设计技术，好的数据设计将改善程序结构和模块划分，降低过程复杂性。

Wasserman[WAS80]提出了一组用于数据规约和设计的原则。在实际应用中，数据设计在创建分析模型(见第 12 章)就已经开始了，考虑到需求分析和设计经常要重叠，我们主要考虑以下一组数据规约原则[WAS80]：

1.用于功能和行为的系统分析原则也应用于数据。我们通常要在导出、复审和刻画功能需求和初步设计上花很多时间和工作量；数据对象及其关系、数据流和内容的表示也应该按步骤进行开发和复审，其他可选的数据组织结构也应加以考虑，数据模型对于软件设计的影响也应得到正确的评估，例如，一个多环链表可能可以很好地满足数据需求，但它也可能导致过于复杂的软件设计，而其他替代的数据组织结构可能会得到更好的结果。

2.应该标识所有的数据结构以及其上的操作。设计一个高效的数据结构必须考虑其上的操作(见参考文献[AHO83])，例如，考虑一个由不同数据元素组成的数据结构，在许多重要的软件功能中都要操作这个数据结构。通过评估该数据结构上的操作，可定义一个抽象数据类型，以便在以后的软件设计中使用。抽象数据类型的规约将大大简化软件设计。

3.应当建立数据字典，并用于数据设计和程序设计。数据字典的概念在第 12 章中已经介绍，数据字典明确表示了数据对象间的关系以及对数据结构中的元素的约束。如果有一个类似字典的数据规约存在，那些必须利用某些特定关系的优秀算法的定义将得到简化。

4.低层的设计决策应该推迟到设计过程的后期。数据设计可以采用逐步求精的过程，也就是说，总体的数据组织可以在需求分析阶段定义，在概要设计中进行精化，并在以后的设计迭代中进行详细描述。在数据设计中应用自顶向下方法的优点与在软件设计中应用自顶向下方法的优点类似：主要的结构属性要首先进行设计和评估，以便建立数据的体系结构。

5.只有那些需要直接使用数据结构内部数据的模块才能看到该数据结构的表示。信息隐蔽的概念以及相关的耦合概念为软件设计质量的评估提供了依据。本原则不但强调了这两个概念的重要性，还强调了“将数据对象的逻辑视图和物理视图分开的重要性”[WAS80]。

6.应该开发一个由有用的数据结构和应用于其上的操作组成的库。数据结构和操作都应被看作可用于软件设计的资源，数据结构的设计可以考虑到复用。数据结构模板(抽象数据类型)库可以减少数据规约和设计的工作量。

7.软件设计和程序设计语言应该支持抽象数据类型的规约和实现。如果没有办法对已有的数据结构直接进行规约，复杂数据结构的实现(以及对应的设计)将变得非常困难。例如，如果目标语言是 Fortran 的话，实现(或设计)一个链表或多层异构数组将是非常困难的，因为 Fortran 不支持直接对这些数据结构进行规约。

以上这些原则为数据设计提供了基础，它们既可以应用在软件工程的定义阶段，也可以应用在开发阶段。在本书的其他部分我们已经指出，清晰的信息定义是软件开发成功的关键。

14.2 体系结构设计

体系结构设计的主要目标是开发一个模块化的程序结构，并表示出模块间的控制关系。此外，体系结构设计将程序结构和数据结构相结合，为数据在程序中的流动定义了接口。

为了理解体系结构设计的重要性，这里给出一个日常生活的小例子：

你存了一笔钱，买了一块土地，准备盖一幢自己梦想中的房子。由于没有这方面的经验，你拜访了一位建筑师，向建筑师解释了自己的要求：房间的大小和数目、流行的式样、温泉、教堂式的屋顶、大量的玻璃窗等等。建筑师仔细的听了，问了几个问题，然后表示需要花几个星期进行设计。

你在焦急的等待中一直在想象自己的房子会是什么样，脑海里已经有了好几幅画面(当然，都非常昂贵)。最后建筑师来了电话，你就立刻赶往建筑师的办公室。

建筑师打开自己的文件夹，拿出第二层浴室的设计图纸，并进行了详细的解释。

“可是，总体的设计是怎样的？”你问。

“不用着急，”建筑师回答，“这个问题我们以后再说。”

建筑师的做法是不是有些不正常？你是不是对建筑师最后的回答感到满意？当然不满意。你需要看到房子的草图、楼层平面图以及其他一些可以提供体系结构视图的信息。然而，许多软件开发人员的做法和例子中的建筑师是类似的，他们注重具体的设计(过程细节和代码)，而忽略了软件体系结构。本节给出的设计方法鼓励软件工程师在考虑具体的设计之前先集中考虑体系结构设计。

14.2.1 贡献者

体系结构设计(和通常的软件设计)起源于早期的强调模块化的设计概念[DEN73]、自顶向下的设计[WIR71]和结构化程序设计[DAH72，LIN79]。Steven、Myers 和 Constantine[STE74]是基于数据在系统中流动的软件设计的最早支持者。这些工作经过精化后归纳在 Myers[MYE78]以及 Yourdon 和 Constantine[YOU79]的书中。

有关软件体系结构设计的一些新工作则对此进行了更细致的研究。Shaw 和 Garlan[SHA95]的书以及 Dean[DEA95]和 Moriconi[MOR95]的文章是有代表性的关于用形式化方法表示体系结构设计模型和模式的工作。

14.2.2 应用域

每种软件设计方法都有其长处和短处，选择设计方法的一个重要因素是其应用的范围。面向数据流的设计可以用于大量的应用领域，事实上，由于所有的软件都可以用数据流图表示，从理论上讲，使用这些数据流图的设计方法可以应用到各种软件开发中。面向数据流的设计在那些信息需要顺序进行处理，并且基本没有层次数据结构的应用中特别有效，例如，微处理器控制；复杂的数值分析过程；过程控制；以及其他一些属于这一类的工程和科学应用软件。面向数据流的设计技术还可以用于数据处理应用，也可以有效地用于有层次数据结构的应用。

然而，在某些应用中对数据流的考虑最好作为一个次要方面，在这些应用中(比如数据库系统、专家系统、面向对象用户界面)，其他的设计方法可能会更有效。

14.3 体系结构设计过程

面向数据流的设计是一种体系结构设计方法，它可以方便地从分析模型转换到程序结构的设计描述。这种从信息流(用数据流图表示)向结构的变迁是通过以下五步过程的某些部分来完成的：(1)建立数据流的类型；(2)指明流的边界；(3)将 DFD 映射到程序结构；(4)用“因子化”的方法定义控制的层次结构；以及(5)用设计测度和启发信息对结构进行求精。其中，信息流的类型是第三步中的映射驱动因素。下面，我们考查两种流类型。

14.3.1 变换流

在基本的系统模型(第 0 层数据流图)中，信息必须以“外部世界”信息的形式进出软件。例如，键盘输入的数据、电话线上的语音信号以及计算机显式器上的图形都是外部世界的信息。为了处理方便，外部的数据形式必须转化成内部的形式，图 14-1 表示了数据的时间历史。信息可以通过各种路径进入系统，并被标识为输入流，信息在这个过程中由外部数据变换成内部的形式。在软件的核心，有一个重要的变换，输入数据通过了“变换中心”，并沿各种路径流出软件，这些流出的数据称为输出流。整个的数据流动以一直顺序的方式沿一条或几条路径进行。如果一部分数据流图体现了这些特征，这就是变换流。

14.3.2 事务流

由于基本的系统模型隐含着变换流，可以把所有的数据流都归为这一类。然而，信息流经常可以被描述成有一个称为事务的单个数据项，它可以沿多条路径之一触发其他数据流。如果数据流图类似于图 14-2 的形式，则就是事务流。

事务流的特征是数据沿某输入路径流动，该路径将外部信息转换成事务，估计事务的价值，根据其价值，启动沿很多“动作路径”之一的流。其中发射出多条动作路径的信息流中心被称为事务中心。

需要指出的是，在一个大系统的 DFD 中，变换流和事务流可能会同时出现，例如，在一个面向事务的流中，动作路径上的信息流可能会体现出变换流的特征。

14.4 变换映射

变换映射是一组设计步骤，可以将具有变换流特征的 DFD 映射为一个预定义的程序结构模板。本节通过将这些设计步骤应用于一个实例系统来说明变换映射，该实例系统是前面章节提到的 SafeHome 安全软件的一部分。

14.4.1 一个实例

前面提到的 SafeHome 安全软件是目前仍在使用的许多基于计算机的产品和系统的代表，该产品监视现实世界，并对遇到的变化做出反应，它也可以通过一系列相应的输入和字母数字显示器与用户交互。图 14-3 是 SafeHome 的第 0 层数据流图，与第 12 章相同。

在需求分析阶段，应该为SafeHome建立更详细的流模型，另外，还应建立控制和加工规约、数据字典和各种行为模型。[①]

14.4.2 设计步骤

上面的例子将被用于说明变换映射的各个步骤，这些步骤首先都要对需求分析中完成的工作进行重新评估，然后转向程序结构的开发。

步骤 1 复审基本系统模型。基本系统模型包括第 0 层 DFD 和支持信息，在实际应用时，这一步骤需要评估系统规约和软件需求规约，这两个文档在软件接口级描述了信息流和结构，图 14-3 和 14-4 描述了 SafeHome 软件的第 0 层和第 1 层数据流。

步骤 2 复审和精化软件的数据流图。需要对从包含在软件需求规约中的分析模型中获得的信息进行精化，以便得到更多的细节。例如，检查第 2 层的“监控传感器”的 DFD(图 14-4 和图 14-5)，导出了第 3 层数据流图(见图 14-6)。在第 3 层，数据流图中的每个变换都展示了高的内聚性(见第 13 章)，也就是说，变换所代表的加工执行单一、独立的功能，该功能可由 SafeHome 软件中的一个模块来实现，因此，图 14-6 中的 DFD 包含了用于设计“监控传感器”子系统的程序结构所需的足够多的细节信息，不需要再进一步精化。

步骤 3 确定 DFD 含有变换流还是事务流特征。总的来说，系统里的信息流总可以表示为变换流，但如果其中有明显的事务流特征(见图 14-2)，最好采用另一种设计映射。在这一步骤中，设计人员根据前面介绍的 DFD 特征确定全局(整个软件)的流特征。此外，也应隔离出局部的变换流和事务流，这些局部流可以用于精化按照全局 DFD 特征导出的程序结构。下面就针对图 14-6 中的“监控传感器”子系统的数据流进行讨论。

通过对 DFD(图 14-6)的评估，我们可以看出，数据通过一条输入路径进入软件，沿三条输出路径流出，没有明显的事务中心(虽然变换“询问警报条件”也可以被看作事务中心)，因此，整个 DFD 具有变换流特征。

步骤 4 划分输入和输出流的边界，隔离变换中心。前面已经指出，输入流被描述为信息从外部形式变换为内部形式的路径;输出流是信息从内部形式变换为外部形式的路径，但对输入流和输出流的边界并未加以说明。实际上，不同的设计人员在选择流边界时可能不尽相同，不同的流边界选择也将导致不同的设计方案。虽然在选择流边界时要加以注意，但是，沿流路径的某个泡泡的变化对最终的程序结构的影响并不会太大。

例子中的流边界由两条纵向的曲线确定(见图 14-6)，包括变换中心在内的变换(泡泡)位于两条边界曲线之间。另一种调整方案是，将输入流的边界定在“读传感器”和“询问响应信息”之间。本步骤的重点在于选择合理的边界，而不是关于边界位置的冗长迭代。

步骤 5 完成“第一级因子化”。程序结构表示了控制自顶向下的分布，因子化的作用是得到一个顶层模块完成决策，低层模块完成大多数输入、计算和输出工作的程序结构，中层的模块既完成一部分控制，又完成适量的工作。

对于变换流，DFD 将被映射成一个能为信息的输入、变换和输出提供控制的特定结构。图 14-7 表示了对“监控传感器”子系统进行第一级因子化的结果，

主控模块(图中称为“监控传感器执行程序”)位于程序结构的顶端，负责协调以下的子控制功能：

输入信息处理控制模块(图中称为传感器输入控制)，负责协调接收所有输入数据。

变换流控制模块(图中称为警报条件控制)，负责管理采用内部形式的数据上的所有操作(如，一个调用多个数据变换过程的模块)。

输出信息处理控制模块(图中称为警报输出控制)，负责产生输出信息。

虽然图 14-7 蕴含了一种三叉的结构，但是，大型系统的复杂数据流图可能会要求为上述的每个控制功能提供两个或多个模块。第一层模块的数量应为既完成控制功能又维持良好的耦合和内聚特征的最少模块数。

步骤 6 完成“第二级因子化”。第二级因子化是将 DFD 中的每一个变换(泡泡)映射为程序结构中的模块。从变换中心的边界开始，沿输入路径和输出路径向外，将变换依次映射到子层的软件结构中去。图 14-8 表示了 SafeHome 的第二级因子化的一般结果。

虽然图 14-8 表示了 DFD 变换和软件模块间的一对一映射，其他的映射方式也经常采用。两个甚至三个变换可以合并在一起，用一个模块表示(有时会产生内聚问题)；一个变换也可以扩展成两个或多个模块。第二级因子化的结果往往是现实考虑和设计质量要求的折衷，以后的复审和精化可能会导致这个结构的改变，但这个结构仍然应作为第一次的迭代结果。

对于输入流的第二级因子化，方法也是类似的，因子化是从变换中心与输入流的边界开始沿输入流反方向进行。子系统软件的变换中心的映射略有不同，DFD 变换中心部分的每个代表转换或计算的变换应该被映射为变换控制模块的子模块。图 14-9 是一个完整的第一次迭代的程序结构。

按以上方法映射出来的模块(见图 14-9)表示了最初的程序结构设计结果。虽然模块的名字已经可以体现其功能，我们仍然需要为每个模块提供简要的处理说明(可以经修改分析建模时创建的 PSPEC 得到)。处理说明应该包含以下内容：

- 模块输入和输出的信息，即接口描述。
- 模块要保留的信息，例如，在局部数据结构中存储的数据。
- 过程描述，指明主要的决策点和任务。
- 有关限制和特殊特性的讨论(例如，文件 I/O、与硬件相关的特征、特殊的时间要求)。

这些描述可以作为第一代的设计规约，以后的设计过程中还要对此进行不断的求精。

步骤 7 用提高软件质量的启发信息，精化第一次迭代得到的程序结构。应用模块独立性的概念(见第 13 章)总能对第一个程序结构进行精化。对模块进行“外突破”或“内突破”，可以得到合理的因子化、好的内聚、低的耦合的程序结构，最重要的是易于实现、测试和维护的程序结构。

求精往往是对现实进行考虑和常识的要求，例如，有时输入数据流的控制模块完全没有必要；有时输入处理需要在变换控制模块的子模块中完成；有时全局数据的存在使得高耦合不可避免；有时优化结构不可能达到。软件需求加人工的判断是最终的依据。

对于为 SafeHome 的监控传感器子系统开发的第一次迭代结果来说，可以进行很多修改：(1)由于只有一条输入路径，输入控制模块就变得没有必要了，因此可以去掉；(2)由变换流生成的子结构可以内突破成模块“建立报警条件”(其中现在包括了“选择电话号码”所代表的处理)，变换控制模块就不再需要了，由此导致的内聚性的较小降低是可以容忍的；(3)模块格式化显示和模块产生显示可以合并成一个新的模块显示过程(我们认为数据的格式是非常简单的)。图 14-10 表示了精化以后的监控传感器程序结构。

以上七个步骤的目的是开发一个全局的软件表示，即一旦结构被定义，我们就可以将其视为一个整体，并据此对软件体系结构进行评估和求精。虽然以后的修改仍然需要做一些工作，但这部分的工作对软件的质量和可维护性具有深远的影响。

读者可以停下来考虑一下以上设计步骤和编写程序的区别。如果只有代码作为软件的表示，开发人员就很难在全局的层次上对软件进行评估和求精，容易导致“只见树木不见森林和后果”。

14.5 事务映射

在许多软件应用中，一个单个数据项触发一条或多条信息流，每条信息流代表一个由数据项的内容蕴含的功能，这个数据项被称为事务。其相应的流特征在 14.3.2 节中已做了讨论，在本节我们主要考虑处理事务流的设计步骤。

14.5.1 一个实例

为了阐明事务映射，可以考虑 SafeHome 软件的用户交互子系统。该子系统的第 1 层的数据流图如图 14-4 所示，对此图进行精化后得到第 2 层数据流图(见图 14-11)，相应的数据字典、CSPEC 和 PSPEC 也应被创建出来。

如图所示，用户命令流入系统，沿三条动作路径之一产生其他的信息流，单个数据项命令类型导致数据流从中心扇出，因此整个数据流的特征是面向事务的。

应该注意，沿这三条动作路径中的两条路径的信息流还需要其他的输入流，例如，系统参数和数据是动作路径配置的输入。三条动作路径都流入变换显示信息和状态。

14.5.2 设计步骤

事务映射的步骤与变换映射的步骤很相似，有时甚至是相同的，主要区别在于将 DFD 映射成的软件结构不同。

步骤 1 复审基本系统模型。

步骤 2 复审和精化软件的数据流图。

步骤 3 确定 DFD 含有变换流还是事务流特征。这三个步骤与变换映射中的对应步骤是一致的。图 14-11 中的 DFD 具有典型的事务流特征，但从变换与“命令处理”相关发出的两条动作路径上的流具有变换流的特征，因此必须为这两种流建立流边界。

步骤 4 标识事务中心和每条动作路径上的流特征。事务中心的位置可以从 DFD 上直接识别出来，事务中心位于几条动作路径的起始点上，在图 14-11 来看，变换与“命令处理相关”就是事务中心。

输入路径(就是接受事务的路径)和所有的动作路径都必须被隔离出来。图 14-11 中也标识出了接受路径和动作路径的边界。对于每条动作路径来说，还应确定它们自己的流特征。例如，路径密码(在图 14-11 中用阴影标出)具有变换流特征，对其又要找出输入、变换和输出流及其边界。

步骤 5 将 DFD 映射到一个适合于进行事务处理的程序结构上。事务流应被映射到包含一个输入分支和一个分类处理分支的程序结构上，输入分支结构的开发与变换流中采用的方法是类似的，从事务中心开始，沿输入路径的变换都被映射成模块。分类处理分支结构又包含一个分类控制模块，它控制下面的动作模块。DFD 的每一个动作流路径应映射成与其自身的流特征一致的结构，图 14-12 表示了这一过程。

考虑用户交互子系统的数据流，本步骤的第一级因子化如图 14.13 所示，变换读用户命令和变换激活/非激活系统可以直接映射到程序结构中，而不需要中间的控制模块；事务中心与“命令处理”相关直接映射成同名的子控制模块；对系统配置和密码处理的控制模块的映射如图 14—12 所示。

步骤 6 因了化并精化该事务结构和每条动作路径的结构。每条动作路径的数据流图有自己的信息流特征，它可以是变换流也可以是事务流。与动作路径相关的子结构可以根据本部分和上部分讨论的设计步骤进行开发。

作为一个例子，考虑图 14-11 中的密码处理信息流(阴影部分)，它具有典型的变换流特征。密码作为输入流，它被送到变换中心与以前记录的密码进行比较，如果比较结果不匹配的话，将产生报警和警告信息。配置路径也是类似地按变换流处理。图 14-14 是最后的程序结构。

步骤 7 用提高软件质量的启发信息，精化第一次迭代得到的程序结构。这一步与对应的变换映射的步骤是一样的。在这两种方法中，模块相关性、实用性(实现和测试的有效性)以及可维护性的标准在修改程序结构时必须加以认真的考虑。

14.6 设计的后处理

应用变换和事务映射后，还应增加作为体系结构设计一部分的所需文档。程序结构开发出来并进行精化以后，还需要完成以下任务：

- 为每个模块开发处理说明。
- 为每个模块提供接口描述。
- 定义局部和全局数据结构。
- 标出所有各种设计限制/局限。
- 进行设计复审。
- 进行优化(如果需要的话)。

处理说明应该是无二义的，它应正确描述模块内的处理过程，其内容主要包括处理任务、决策和 I/O 的描述。

接口描述需要对内部模块接口、外部系统接口和人机界面进行设计，这些内容在 14.8 节讨论。

数据结构的设计对程序结构和每个模块的过程细节都有深远的影响。第 12 章中介绍的技术可以用来建立基础数据模型和标识各种重要数据对象，它们可以作为设计局部和全局数据结构的基础。

应该记录下对模块的限制和/或局限，典型的讨论话题包括对数据类型和格式的限制、对内存和时间的局限、对数据结构的量和边界值的限制、没有考虑的特殊情况和个体模块的独特特征。记录这些限制/局限的目的是减少因对模块的功能理解不准确而引入的错误的数量。

每个模块的设计文档开发完毕后，就要对设计进行复审(复审的指南见第 8 章)。复审强调软件需求的可跟踪性、程序结构的质量、接口的描述、数据结构的描述、实现和测试的实用性以及可维护性。

14.7 体系结构设计优化

在对设计优化进行讨论之前，应首先记住这句话：“不能实施的‘优化设计’是毫无价值的”。软件设计人员应该始终考虑开发一个能满足所有功能和性能需求以及设计质量要求的软件表示。

应该鼓励设计阶段早期对程序结构多做精化，软件表示在开发出来以后应该不断精化和评估，以达到“最好”。便于优化也是开发软件体系结构表示的一个重要因素。

重要的是要注意，结构上的简单往往反映出程序的优雅和高效。设计优化应在满足模块化要求的前提下尽量减少模块数量，在满足信息需求的前提下尽量减少复杂数据结构。

对于对性能要求很高的应用来说，可能还需要在设计的后期甚至编码阶段进行优化。软件工程师应该注意，一小部分程序(通常占 10%～20%)往往占用大部分的处理时间(50%～80%)。因此，以下的方法对于对性能要求很高的应用来说并非是不合理的：

1.开发和精化程序结构，且不考虑关键性能而进行的优化。

2.使用可以提高运行性能的 CASE 工具来孤立低效的部分。

3.在后期设计中，对有可能最消耗时间的模块的算法进行时间优化。

4.用适当的程序设计语言编码。

5.孤立出那些大量占用处理器时间的模块。

6.如果需要，用依赖机器的语言重新设计和编码，以提高效率。

这一方法遵循了下面的格言：“先使其工作起来，再设法使其更好地工作”。后面我们还要对此进一步讨论。

14.8 接口设计

体系结构设计为软件工程师提供了程序结构的全局视图，就像房子的蓝图一样，如果不画出门、窗、水管、电线和电话线(更不要说有线电视的电缆)，整个设计是不完整的。对于计算机软件来说，这些“门、窗和布线”就构成了系统的接口设计。

接口设计主要包括三个方面：(1)设计软件模块间的接口；(2)设计模块和其他非人的信息生产者和消费者(比如外部实体)的接口；以及(3)设计人(用户)和计算机间的接口。

14.8.1 内部和外部接口设计

内部程序接口的设计有时也被称为模块间的接口设计，它是由模块间传递的数据和程序设计语言的特性共同导致的。[①]一般来说，分析模型中包含了足够的信息用于模块间的接口设计。数据流图(见第 12 章)描述了数据对象在系统中流动时发生的变换，DFD中的变换(泡泡)被映射到程序结构的模块中(见 14.4 节和 14.5 节)，因此，每个DFD变换的输入和输出箭头(数据对象)必须被映射到与该变换对应的模块接口上。

外部接口设计起始于对分析模型的 DFD 中的每个外部实体的评估。外部实体的数据和控制需求确定下来以后，就可以设计外部接口了，例如，前面讨论的 SafeHome 软件需要与各种传感器连接，每个传感器的外部接口设计由传感器所需的特定数据和控制项决定。

内部和外部接口设计必须与模块内的数据验证和错误处理算法紧密相关，由于副作用往往是由程序接口进行传播的，必须对从某模块流向另一个模块(或流向外部世界)的数据进行检查，以保证符合需求分析时要求的确定。

14.8.2 用户界面设计

在 Ben Shneiderman[SHN87]专门介绍用户界面设计的书中，Shneiderman 指出：

对于许多计算机化的信息系统的用户来说，挫折和焦虑是他们日常生活的一部分。他们努力地学习命令语言和菜单选择系统以帮助他们更好地完成工作。有些人甚至对计算机、终端和网络产生了恐惧，因而刻意地回避计算机化的系统。

Shneiderman 提到的问题并不是危言耸听，我们也都遇到过难学、难用、令人迷惑、对人过于苛刻，总之使人感到沮丧的用户界面。然而，仍然有人花费时间和精力来开发这样的界面，看起来，他们并不是有意地在制造问题。

用户界面的设计要求在研究技术问题的同时对人加以研究。用户是什么样的人？用户怎样学习与新的计算机系统交互？用户怎样解释系统产生的信息？用户对系统有那些期望？这些问题仅仅是需要在用户界面设计时必须询问和回答的部分问题

14.9 人机界面设计

设计用户界面的过程应该从创建不同的系统功能模型(从外部对系统的视图)开始。从此时开始，用以完成系统功能的任务被分为了面向人的和面向计算机的；它们在要应用到接口设计的各种设计问题中得到考虑。各种工具可被用于建造原型和最终实现设计模型；最后从质量的角度对结果进行评估。

14.9.1 界面设计模型

设计人机界面(HCI)时要考虑四种模型：软件工程师创建“设计模型”；人员工程师(或者也是软件工程师)建立“用户模型”；终端用户在脑海里对界面产生的映象称为“用户的模型”或“系统感觉”；系统的实现者创建“系统映像”[RUB88]。不幸的是，这四种模型可能会相去甚远，界面设计人员的任务就是消除这些差距，导出一致的表示界面。

整个系统的设计模型包括对软件的数据、体系结构、界面和过程的表示，需求规约可以建立一定的约束以有助于定义系统的用户，界面的设计往往是设计模型附带结果[①]。

用户模型描述了系统终端用户的特点。为了建立有效的用户界面，“开始设计之前，必须对用户加以了解，包括年龄、性别、身体状况、教育、文化和种族背景、动机、目的以及性格”[SHN87]。此外，用户可以分类为：

• 新手：对系统没有任何“语法了解”[②]，对该应用程序或计算机的一般用法几乎没有“语义了解”[③]；

• 对系统有了解的中级用户：对该应用程序有一定的合理的语义了解，但对使用界面所必需的语法信息的了解还比较少；

• 对系统有了解的常用用户：对该应用程序有很好的语义和语法了解(这经常导致“强力用户综合症”)，这些用户经常寻找更简洁和快速的交互方式。

系统感觉(用户的模型)是终端用户在脑海里对系统产生的印象，例如，请某种特殊字处理程序的用户描述其功能，系统感觉是用户回答的依据，准确的回答取决于用户的特点(新手只能做简要的回答)和用户对同领域软件的了解。一个对字处理软件有深刻了解，但刚刚开始使用这种字处理程序的用户可能比已经使用该字处理程序好几个星期的新手回答得更详细。

系统映像包括计算机系统的外在表示(界面的观感)和用来描述系统语法和语义的支撑信息(书、手册、录像带)。如果系统映像和系统感觉是一致的，用户就会对软件感到很舒服，使用起来效率就很高。为了将这些模型融合起来，设计模型必须包括用户模型中的一些信息，系统映像必须准确地反映接口的语法和语义信息。这几种模型的关系如图 14-15 所示。

这些模型是对“用户在使用交互式系统时的所作所为、或是用户想象中的所作所为、或是他人想象中的用户的所作所为的抽象”[MON84]。从本质上看，这些模型使得界面设计人员能够满足界面设计中的最重要的原则：“了解用户，了解任务。”

14.9.2 任务分析和建模

任务分析和建模有助于理解人们同时执行的任务(以手工或半自动的方式)，并将它们映射成在 HCI 环境中实现的一组类似的任务(并不需要完全一样)。这一过程可以通过研究已有的计算机解决方案的规约，导出能够表示用户模型、设计模型和系统感觉的一组任务。

无论通过什么渠道进行任务分析，人员工程师必须首先定义任务并对任务分类，一种办法是分多步进行(见第 13 章)。例如，一个小软件公司想要为室内设计人员建立一个计算机辅助设计系统，通过对设计人员工作的观察，工程师注意到室内设计主要包括以下一些活动：家具布局、材料选择、墙面和窗面的选择、向用户的展示、商定价格和购买。其中每项任务又可分成子任务，例如，家具布局可以分为：(1)基于房间格局画出楼层平面图；(2)将门窗放在适当的位置；(3)用家具模板在平面图上画出家具轮廓；(4)将家具轮廓放到最合适的位置；(5)标记出所有家具轮廓；(6)画出标尺，以确定位置；以及(7)画出客户的视图。对于其他任务也可以进行类似划分。

这七个子任务还可以进一步细分，前六个子任务可以通过用户界面来操纵信息和执行动作，第七个子任务则由软件自动完成，基本不需要用户干预。界面的设计模型应该以一种与用户模型(“典型的”室内设计人员的视图)和系统感觉(室内设计人员对自动系统的期望)一致的方式完成这些任务。

另一种任务分析方法采用了面向对象的观点[①]。人员工程师观察室内设计人员使用的物理对象以及施加在每个对象上的动作，例如，家具模板就应是这种任务分析方法中的一个对象，室内设计人员可以“选择”适当家具模板，将其“移动”到合适的位置，“画出”家具模板的轮廓等等。界面的设计模型不必描述每个动作的实现细节，但必须定义出完成最后结果的用户任务(在这里就是“在平面图上画出家具轮廓”)。

定义好每一项任务或动作后，界面设计就开始了。界面设计过程最初的步骤[NOR86]可以按以下步骤进行：

1.建立任务的目标和意图。

2.为每个目标或意图制订特定的动作序列。

3.按在界面上执行的方式对动作序列进行规约。

4.指明系统状态，即执行动作时的界面表现。

5.定义控制机制，即用户可用的改变系统状态的设备和动作。

6.指明控制机制如何影响系统状态。

7.指明用户如何通过界面上的信息解释系统状态。

14.9.3 设计问题

在进行用户界面设计时，几乎总会遇到以下四种问题：系统响应时间、用户帮助设施、错误信息处理和命令交互。不幸的是，许多设计人员往往在很晚的时候才注意到这些问题(有时在操作原型已经建立起来后问题才出现)，这往往会导致不必要的反复、项目滞后和用户的挫折感，最好的办法是在设计的初期就将这些作为设计问题加以考虑，因为此时修改比较容易，代价也低。

系统响应时间是交互式系统中用户经常抱怨的地方。一般来说，系统响应时间是指从用户开始执行动作(比如按“回车”键或点鼠标)到软件给出预期的响应的等待时间。

系统响应时间包括两方面的属性：长度和易变性。如果系统响应时间过长，用户就会感到不安和沮丧，过快的系统相应时间有时也会成为问题，因为这会迫使用户加快操作节奏，从而导致错误。

系统响应时间的易变性是指相对于平均响应时间的偏差，这往往更重要。即使响应时间比较长，低的响应时间易变性也有助于用户建立稳定的节奏。例如，稳定在 1 秒的响应时间比从 0.1 到 2.5 秒不定的响应时间要好。用户往往比较敏感，他们总是关心界面后面是否发生了异常。

交互式系统的用户总是或多或少地需要帮助。有时问一下同事就可以解决问题，但细致的问题需要查看用户手册，在多数情况下，现代的软件均提供联机帮助，用户可以不离开界面就解决自己的问题。

常见的帮助设施有两种：集成的和附加的[RUB88]。集成的帮助设施是一开始就设计在软件里面的，它通常与语境相关，用户可以直接选择与所要执行操作相关的主题。显然，这可以缩短用户获得帮助的时间，增加界面的友好性。附加的帮助设施是在系统建好以后再加进去的，在多数情况下，这种帮助是一种查询能力较弱的联机帮助，用户必须自己在成百上千条主题中查找所需主题，经常会有无用的查找和得到无关信息，因此，人们普遍认为集成的帮助优于附加的帮助。

考虑到帮助设施，设计时又要考虑一系列问题[RUB88]：

• 在进行系统交互时，是否总能得到各种系统功能的帮助？有两种选择：提供部分功能的帮助和提供全部功能的帮助。

• 用户怎样请求帮助？有三种选择：帮助菜单、特殊功能键和 HELP 命令。

• 怎样表示帮助？有三种选择：在另一个窗口中、指出参考某个文档和在屏幕特定位置的简单提示。

• 用户怎样回到正常的交互方式？有两种选择：屏幕上的返回键和功能键或控制序列。

• 怎样构造帮助信息？有三种选择：平面式(所有信息均通过一个关键词来访问)、分层式(用户可以进一步查询得到更详细的信息)和超文本式。

出错信息和警告是指出现问题时系统给出的坏消息。如果做不好的话，出错信息和警告会给出无用或误导的信息，反而增加了用户的沮丧感。很少有用户没有遇到类似以下的出错信息：

SEVERE SYSTEM FAILURE—14A

应该在某处有对错误 14A 的解释，否则设计者为什么会指出 14A 呢？但出错信息并没有指出是什么错误或从何处可以找到进一步的信息。类似上面的出错信息既不能减轻用户的焦虑感也不能解决问题。

通常，交互式系统给出的出错信息和警告应具备以下特征：

• 信息以用户可以理解的术语描述问题。

• 信息应提供如何从错误中恢复的建设性意见。

• 信息应指出错误可能导致哪些不良后果(比如破坏数据)，以便用户检查是否出现了这些情况，或帮助用户进行改正。

• 信息应伴随着视觉或听觉上的提示。即显示信息时应该伴随警告声，或信息用闪烁方式显示，或信息用明显的表示错误的颜色显示。

• 信息不能带有判决色彩，即不能指责用户。

由于没有人喜欢坏消息，无论出错信息代表了什么也很少有用户喜欢出错信息，但从道理上讲，在出现问题时有效的出错信息能够提高交互式系统的质量、减少用户的沮丧感。

命令行曾经是用户和系统交互的主要方式，并广泛用于各种应用程序中。现在，面向窗口的采用点击(point)和拾取(pick)方式的界面减少了用户对命令行的依赖，但许多高级用户仍然喜欢面向命令的交互方式。在许多软件中，用户既可以从菜单中选择一个命令，也可以用键盘输入命令序列。

在提供命令交互方式时，必须考虑以下问题：

• 每一个菜单选项是否都有对应命令？

• 以何种方式提供命令？有三种选择：控制序列(比如 ^ P)、功能键和键入命令。

• 学习和记忆命令的难度有多大？命令忘了怎么办？(见本节前面有关帮助的讨论)。

• 用户是否可以定制和缩写命令？

在越来越多的应用中，界面设计者提供“命令宏机制”，用用户定义的名字代表一个常用的命令序列，用户不必分别输入这些命令，只需输入命令宏就可以执行其代表的命令序列。

在理想情况下，所有的应用程序应有通用的命令使用方法。如果在一个应用中 ^D 表示复制一个图形对象，而在另一个应用中 ^D 表示删除一个图形对象，这就会使用户感到困惑，并往往会导致错误，潜在的错误是明显的。

14.9.4 实现工具

界面设计是一个迭代的过程，设计模型先被实现成一个原型，[①]由用户进行检查(用户最能体会用户模型)，然后根据用户的意见进行修改。为了适应这种迭代的过程，用于界面设计和原型开发的工具也应运而生。这些工具被称为用户界面工具箱或用户界面开发系统(UIDS)，它们为简化窗口、菜单、设备交互、出错信息、命令以及很多其他交互环境元素的创建提供了各种例程和对象。

由于采用了已经开发好的可以被设计者、实现者和用户界面直接使用的软件包，UIDS 提供了以下的固有机制[MYE89]：

- 管理输入设备(比如鼠标和键盘)。
- 确认用户输入。
- 处理错误和显示出错信息。
- 提供反馈(比如自动的输入响应)。
- 提供帮助和提示。
- 处理窗口、域和窗口内的滚动。
- 建立应用软件和界面间的连接。
- 将应用程序与界面管理分离。
- 允许用户定制界面。

以上功能既可以通过基于语言的方式也可以通过基于图形的方式来实现。

14.9.5 设计评估

建立好操作性用户界面原型后，必须对其进行评估，以确定是否满足用户的需求。评估可以很不正式，比如用户可以临时提供一些反馈；也可以非常正式，比如按照统计学的方法向一定数量的用户发放问题评估表。

用户界面评估的周期如图 14-16 所示。完成初步设计后就开始实现第一级的原型；用户对该原型进行评估，直接向设计者提供有关界面功效的建议，如果采用正式的评估技术(比如问题表)，设计者需要从调查结果中得到需要的信息(比如 80％的用户不喜欢其中的保存数据文件的机制)；针对用户的意见对设计进行修改，完成下一级的原型。评估过程不断进行下去，直到不需要再修改为止。但是，在建立原型以前是否就可以对用户界面的质量进行评估呢？如果能够及早地发现和改正潜在的问题，就可以减少评估周期执行的次数，从而缩短开发时间。

界面的设计模型完成以后，就可以运用下面的一系列评估标准[MOR81]对设计进行早期复审：

1.书面的系统规约和界面规约的长度和复杂性在一定程度上表示了用户学习系统的难度。

2.命令或动作的个数、以及命令的平均参数个数或动作中单个操作的数量在一定程度上表示了系统交互的时间和系统总体的效率。

3.设计模型中动作、命令和系统状态的数量反应了用户学习系统时所要记忆的内容的多少。

4.界面风格、帮助设施和错误处理协议在一定程度上表示了界面的复杂度和用户的接受程度。

原型完成以后，设计者就可以收集到一些定性和定量的数据帮助进行界面评估。问题表可以用于收集定性的数据，问题的答案可以是(1)是/否、(2)数字、(3)程度(主观的)或(4)百分比(主观的)。例如：

1.该命令是否容易记忆？(是/否)

2.你使用了多少条命令？

3.学习基本系统操作的容易程度？(程度为 1 至 5)

4.与其他的界面相比，你对该界面的评价如何？(前 1％，前 10％，前 25％，前 50％，后 50％)

如果需要定量的数据，就必须进行某种形式的定时研究分析，观察用户对界面交互的使用，记录以下数据：在标准时间内正确完成任务的数量、使用命令的频度、命令序列、用于看屏幕的时间、出错的数目、错误的类型和错误恢复时间、使用帮助的时间、标准时间段内查看帮助的次数，这些数据可以用于指导界面修改。

有关用户界面评估方法的详细论述已超出了本书的范围，有兴趣的读者可以参考文献[LEA88]。

14.10 界面设计指南

界面的设计很大程度上依赖于设计者的经验，许多文献(如，[DUM88]也给出了设计友好、高效的界面的指南，本部分则指出一些更为重要的 HCI 设计指南。

下面分三部分讨论这些指南：一般交互、信息显示和数据输入。

14.10.1 一般交互

一般交互的指南经常跨越边界进入信息显示、数据输入和整体系统控制，因此这些指南是全局性的，忽略它们将承担较大风险。以下就是一些一般交互的指南：

一致性。在 HCI 中的菜单选择、命令输入、数据显示以及无数其他功能都应使用一致的格式。

提供有意义的反馈。向用户提供视觉和听觉的反馈，以保证在用户和界面间建立双向联系。

在执行有较大破坏性的动作前要求确认。如果用户要删除一个文件，或覆盖一些重要信息，或请求停止一个程序，应该给出类似“您确实要…？”的信息。

允许取消大多数操作。UNDO 或 REVERSE 功能使成千上万的用户免受成百万小时的挫折。每个交互式应用都应允许取消已完成的操作。

减少在动作间必须记忆的信息数量。不应期望用户能记住一大串数字或名字，以便在下一步的功能中使用，记忆量应尽量减少。

在对话、移动和思考中提高效率。击键次数应尽量减少，设计屏幕布局时应考虑鼠标移动的距离，用户问“下面怎么办？”的时候应尽量减少。

允许错误。系统应保护自己不受致命错误的破坏。

按功能对动作分类，并据此安排屏幕布局。下拉菜单的一个优点就是按类型组织命令。实际上，设计者应努力提高命令和动作组织的内聚性。

提供语境相关的帮助机制。见 14.9.3 节。

命令用简单的动词或动词短语命名。过长的命令名难以识别和记忆，也会占据过多的菜单位置。

14.10.2 信息显示

如果 HCI 显示的信息是不完整的、含糊的或难以理解的，应用软件就难以满足用户的需求。信息可以以多种不同方式显示：用文字、图片和声音；按位置、运动和大小；使用颜色、分辨率和甚至省略。下面是一些信息显示的指南：

只显示与当前语境环境相关的信息。用户在获取有关某特定系统功能的信息时，不必看到其他的数据、菜单和图形。

不要用数据将用户包围，使用便于用户迅速吸取信息的方式表现数据。可以用各种图形取代巨大的表格。

使用一致的标记、标准的缩写和可预测的颜色。显示信息的含义应该非常明确，用户不必再参照其他信息源。

允许用户维持可视化的语境。如果图形表示可以伸缩，原来的图像应一直显示着(比如以缩小的形式放在屏幕的一角)，以使得用户可以知道自己观察的部分在原图中的相对位置。

产生有意义的出错信息。见 14.9.3 节。

使用大小写、缩进和文本来辅助理解。HCI 显示的大部分信息是文字，文字的布局和形式对用户从中吸取信息影响很大。

使用窗口分隔不同类型的信息。窗口使得用户可以方便地保存多种不同类型的信息。

如果采用“模拟”的明显方式表示信息，更容易被用户理解，则应采用模拟方式。例如，显示炼油厂储油罐的压力时，简单的数字表示难以被用户理解。如果用类似温度计的方式来表示，用垂直的运动和颜色变化来表示危险的压力状况，就比较容易理解，因为这样为用户提供了绝对和相对两方面的信息。

高效地使用显示器的显示空间。如果采用多窗口，应该为每个窗口都留一些显示空间。此外，屏幕的大小应该适合于应用程序(这实际上是一个系统工程的问题)。

14.10.3 数据输入

用户的大部分时间是花在选择命令、键入数据或者提供系统输入等方面。在许多应用中，键盘是主要的输入介质，鼠标、数字化仪甚至语音识别系统正在成为重要的输入手段。下面是一些数据输入的指南：

减少用户输入动作的数量。最主要的是减少击键的数量，这可以用以下方式实现：用鼠标选择菜单代替击键；用滑动标尺输入一定范围内的值；使用宏，用一次击键代表复杂的输入数据。

维护信息显示和数据输入的一致性。显示的视觉特征(比如文字的大小、颜色和位置)应与输入域一致。

允许用户自定义输入。专家用户可能需要定义用户命令或省去警告信息及动作确认。HCI 应该允许这样做。

交互应该是灵活的，并可调整到用户最喜欢的输入方式。用户模型有助于确定用户喜欢的输入模式。书记员可能很喜欢键盘输入，经理可能会喜欢鼠标一类的点击设备。

在当前动作的语境中使不合适的命令不起作用。这使得用户不会使用哪些肯定会导致错误的动作。

让用户控制交互流。用户应该能够跳过不必要的动作、改变所需动作的顺序(如果应用的语境环境允许的话)以及在不退出系统的情况下从错误状态恢复。

为所有的输入动作提供帮助。见 14.9.3 节。

消除冗余输入。不要要求用户指定工程输入的单位(除非不这样会产生含混)；不要要求用户在整钱数后加.00、可能的话要提供缺省值、以及绝不要让用户提供程序中可以自动获得或计算出来的信息。

14.11 过程设计

过程(Procedural)设计应该在数据、体系结构和界面设计完成之后进行。在理想情况下，用以定义算法细节的过程规约应该用自然语言表达(比如英语)。软件开发组织内的人员都说自然语言，组织外的人员也更容易理解自然语言，自然语言也不需要重新学习。

不幸的是，过程细节的规约必须是无二义的，而没有二义的自然语言就不再是自然语言了。采用自然语言，一组过程步骤可以用多种方式表达，往往需要根据语境环境来确定真正的含义。用自然语言书写过程步骤时，我们往往假定可以与读者进行对话，这实际上是不可能的。基于这些原因，对过程细节的表达必须有很强的约束。

14.11.1 结构化程序设计

过程设计的基础在六十年代初已经形成，在Edsgar Dijkstra等人的工作中又得到了进一步的完善(见参考文献[BOH66]、[DIJ65]和[DIJ76])，在六十年代末，Dijkstra等人提出，所有的程序都可以建立在一组已有的逻辑构成元素上，这一组逻辑构成元素强调了“对功能域的维护”，其中每一个逻辑构成元素有可预测的逻辑结构，从顶端进入，从底端退出，读者可以很容易地理解过程流。[①]

这些逻辑构成元素包括顺序、条件和重复。顺序实现了任何算法规约中的核心处理步骤；条件允许根据逻辑情况选择处理的方式；重复提供了循环。这些逻辑构成元素是结构化程序设计的基础，而结构化程序设计是过程设计的一种重要技术。

结构化的构成元素使得软件的过程设计只采用少数可预测的操作。复杂性度量的理论(见第 18 章)表明，结构化的构成元素减少了程序复杂性，从而增加了可读性、可测试性和可维护性。使用少量的构成元素也符合心理学家所谓的“成块理解”的过程。要说明这一过程，可以考虑阅读一页书的过程，读者不是阅读单个的字母，而是识别由字母组成的单词和短语。结构化的构成元素就是一些逻辑块，读者可以用它来识别模块中的过程成份，而不必逐行阅读设计或代码，遇到容易识别的逻辑结构就能提高理解度。

任何程序，无论是哪个应用领域，无论技术上有多复杂，总可以用这三种结构来设计和实现。需要指出的是，完全只使用这三种结构可能会带来实际的困难，14.11.2 节进一步考虑了这方面的问题。

14.11.2 图形设计符号

“一幅图胜过一千句话”，但也要看是哪幅图和哪一千句话。毫无疑问，诸如流程图和盒状图等图形工具为描述过程细节提供了很好的图形模式，然而，如果错误地使用了图形工具，错误的图形也将导致错误的软件。

流程图曾是最广泛使用的过程设计的图形表示方法，但它也是最广泛地被滥用的方法。

流程图画起来很简单，方框表示处理步骤，菱形表示逻辑条件，箭头表示控制流。图 14—17 表示了 14.11.1 节中讨论的三种结构化构成元素。顺序由两个表示处理的方框以及连接两者的控制线表示；条件也称为 if-then-else 结构，由一个菱形表示，如果值为真则进行 then 部分，如果值为假则进行 else 部分；重复可由两种略有不同的结构表示，do-while 结构首先测试条件，然后重复执行循环任务，只要测试条件为真就不停止，repeat-until 结构首先执行循环任务，然后再测试条件，只要不满足测试条件就不停止。图中的选择结构(也称为 select-case 结构)实际上是 if-then-else 的扩展，参数被连续地测试，直到有一次测试为真，其对应的处理步骤将被执行。

结构化的构成元素可以相互嵌套(见图 14－18)。在图中，一个 repeat-until 结构作为一个 if-then-else 结构的 then 部分(用虚线框标出)；另一个 if-then-else 结构则作为 else 部分；最后，整个条件作为顺序结构中的第二个步骤。用嵌套的方法可以表示复杂的逻辑结构。需要指出的是，图 14－18 中的每一个方框均可以引用其他的模块，从而表出程序结构中的过程分层。

一般来说，如果需要从一组嵌套的循环或条件中退出，完全依赖结构化的构成元素将导致效率降低。更重要的是，退出路径上的复杂的逻辑测试将影响软件的控制流，增加出错的可能，降低可读性和可维护性。如何解决这一问题呢？

设计人员有两种选择：(1)重新设计过程表示，保证内层嵌套的控制流中不需要退出分支；或(2)在一定的范围内破坏结构化的构成元素，设计一条特殊的退出路径。第一种选择显然是最理想的，但第二种选择也不破坏结构化编程的精神。

另一种图形化设计工具是盒状图，它的目标是开发一种不破坏结构化构成元素的过程设计表示。盒状图由 Nassi 和 Shneiderman[NAS73]开发，Chapin[CHA74]对其进行了扩展，因此盒状图又称为 Nassi-Shneiderman 图，或 N—S 图，或 Chapin 图。盒状图有以下特征：(1)功能域(即重复和 if-then-else 的作用域)定义明确、表示清晰；(2)不允许随意的控制流；(3)局部和全局数据的作用域很容易确定；(4)表示递归很方便。

使用盒状图的结构化框架的图形表示见图 14－19，盒状图的最基本的成份是方盒，两个方盒上下相连表示顺序；条件盒加上 then 部分的方盒和 else 部分的方盒表示 if-then-else 结构；用边界部分将处理过程包围起来表示重复(do-while 和 repeat-until)；表示选择的盒状图在图 14－19 的右下角。

和流程图一样，盒状图也可以画成分层结构，以表示模块的精化，对子模块的调用可以表示为一个方盒，并将模块名写在一个椭圆中。

14.11.3 表格设计符号

在许多软件应用中，模块需要对复杂的组合条件求值，并根据该值选择要执行的动作。决策表可以将对处理过程的描述翻译成表格，该表很难被误解，并且可以被计算机识别。NedChapin 对决策表的综合评价是[HUR83]：

一些旧的工具和技术可以与新的软件工程技术结合，决策表就是个很好的例子。决策表比软件工程早出现十年，但它与软件工程非常吻合，好像专为软件工程所设计。

决策表的组织见图 14－20，该表分为四个部分，左上部列出了所有的条件，左下部列出了所有可能的动作，右半部构成了一个矩阵，表示条件的组合以及特定条件组合对应的动作，因此矩阵的每一列可以解释成一条处理规则。

下面是开发决策表的步骤：

1.列出与特定过程(或模块)相关的所有动作。

2.列出执行该过程时的所有条件(或决策)。

3.将特定的条件组合与特定的动作相关联，消除不可能的条件组合；或者找出所有可能的条件排列。

4.定义规则，指出一组条件应对应哪个或哪些动作。

为了例举决策表的使用，考虑下面一段从一个用电收费系统中摘录的处理说明：

如果用固定比率收费，月耗电少于100度的用户负担的费用为某个固定值，其余用户应按照表A中的比率计算。如果用可变比率收费，月耗电少于100度的部分用表A计费，其余部分用表B计费。

图14－21是上述处理说明的决策表。五条规则中的每一个都代表五个条件中的一个(用户不可能既按固定比率收费又按可变比率收费)。一般来说，决策表可以作为其他过程设计方法的有效辅助方法。

14.11.4 程序设计语言

程序设计语言(PDL)也称为结构化的英语或伪码，它是“一种混合语言，采用一种语言的词汇(即英语)和另一种语言的语法(即一种结构化程序设计语言)”[CAI75]。在本章，PDL作为设计语言的附属参考。

PDL看起来像现代的编程语言，其区别在于PDL允许在自身的语句间嵌入叙述性文字(如英语)，由于在有语法含义的结构中嵌入了叙述性文字，PDL不能被编译，但PDL处理程序可以将PDL翻译成图形表示，并生成嵌套图、设计操作索引、交叉引用表以及其他一些信息。

程序设计语言可以由编程语言(比如Ada或C)变换得来，也可以是专门买来用于过程设计的产品。无论怎样得来，设计语言必须有以下特征：

·有为结构化构成元素、数据声明和模块化特征提供的固定的关键词语法；

·是自由的自然语言语法，能用以描述处理特性；

·数据声明机制应既可以说明简单数据结构(如标量和数组)，也可以声明复杂数据结构(如链表和树)；

·支持各种接口描述的子程序定义和调用技术。目前，常用一种高级编程语言作为PDL的基础，例如，Ada-PDL是Ada程序员们常用的设计工具，其中Ada语言的结构和格式与英语叙述混合构成了这种设计语言。

基本的PDL语法应包括：子程序定义、接口描述和数据声明；以及针对块结构、条件结构、重复结构和I/O结构的技术。这些PDL构成元素的格式和语义将在下一部分介绍。

值得注意的是，可以对 PDL 进行扩展，以支持多任务、并发处理、进程间同步以及其他一些特性，使用 PDL 的应用设计应采用该 PDL 的最终格式。

14.11.5 一个 PDL 实例

为了说明 PDL 的使用方法，我们以前面提到的 SafeHome 安全系统的过程设计为例。SafeHome 系统监控火、烟、盗贼、水(洪水)和温度(比如冬天房主外出时炉子熄火)等；产生报警信号；调用监控服务，发出合成的语音信息。在下面的 PDL 中，我们可以找到许多 14.11.4 节中提到的重要构成元素。

考虑到 PDL 不是编程语言，设计人员可以随便修改而不必担心语法错误，然而，监控软件的设计应该在编写代码之前进行复审(你发现了什么问题吗？)和精化。下面的 PDL 定义了“安全监控”过程的设计。

```
PROCEDURE security.monitor;

INTERFACE RETURNS system.status;

TYPE signal IS STRUCTURE DEFINED

nameIS STRING LENGTH VAR;

address IS HEX device location;

bound.value IS upper bound SCALAR;

message IS STRING LENGTH VAR;

END signal TYPE;

TYPE system.statusISBIT(4);

TYPE alarm.type DEFINED

smoke.alarm ISINSTANCEOFsignal;

fire.alarm IS INSTANCE OF signal;

water.alarm IS INSTANCE OF signal;

temp.alarm ISINSTANCEOFsignal;

burglar.alarm IS INSTANCEOFsignal;

TYPE phone.number IS area code+7-digit number;
```

•

•

•

initialize all system ports and reset all hardware;

CASE OF control.panel.switches(cps):

WHEN cps=“test”SELECT

CALL alarm PROCEDURE WITH

“on” for test.time in seconds;

WHEN cps=“alarm-off” SELECT

CALL alarm PROCEDURE WITH

“off”;

WHEN cps=“new.bound.temp” SELECT

CALL keypad.inputPROCEDURE;

WHEN cps=“burglar.alarm.off” SELECT

deactivate signal[burglar.alarm];

DEFAULT none;

ENDCASE

REPEATUNTILactivate.switchis turnedoff

reset all signal.values and switches;

DO FOR alarm.type=smoke, fire, water, temp, burglar;

READ address[alarm.type]signal.value;

IF signal.value＞bound [alarm.type]

THEN phone.message=message[alarm.type];

set alarm.bell to“on” for alarm.timeseconds;

```
PARBEGIN
CALL alarm PROCEDURE WITH “on”, alarm.time in seconds;
CALL phone PROCEDURE WITH message[alarm.type],
phone.number;
ENDPAR
ELSE skip
ENDIF
ENDFOR
ENDREP
END security.monitor
```

注意到， security.monitor 过程的设计者采用了一个新的结构：PARBEGIN…ENDPAR，它表示并行的块，其中的所有任务都以并行方式执行。在本例中没有考虑实现细节。

程序设计语言经常与 CASE 设计工具结合使用，这样可以在过程设计表示中嵌入图形化的成份。例如控制结构图(CSD)[CRO96]可以用于编程语言的源代码和 PDL 中。用图形符号表示重要的结构化编程构成元素和特殊的语言形式可以增强设计文字的表达能力。CSD 记号见图 14—2。

“何种设计符号体系最好？”是很自然的问题。这个问题的答案因人而异，也有待于进一步探讨，然而，似乎程序设计语言提供了最好的功能组合，还可以把 PDL 嵌入文档，降低设计维护的难度。可以用任何文本编辑器或字处理系统对 PDL 进行编辑；也可以用自动的处理程序进行处理，其“自动代码生成”的前景也很不错。

然而，这并不说明其他的设计符号体系就比 PDL 差，图形化的方式使得使用流程图和盒状图的设计人员更容易看清控制流，因而为许多设计人员所喜爱；决策表内容精确，是表驱动应用的理想工具；本书没有涉及的其他设计表示[PET81][SOM89]也各有其优点，总之，对设计工具的选择可能更取决于人的因素而不是技术因素。

14.12 小结

软件设计包括四个独立又相互联系的活动：数据设计、体系结构设计、接口设计和过程设计，这四个活动完成以后就得到了全面的软件设计模型。

数据设计将分析模型中的数据对象转换成数据结构。数据对象的属性、数据对象间的关系以及它们在程序中的使用都会影响数据结构的选择。

本章中的体系结构设计方法使用分析模型中信息流的特征来导出程序结构。可以用两种方法将数据流图映射成程序结构：变换映射和事务映射。变换映射适用于具有明确的输入和输出流边界的数据流图，DFD 对应的程序结构包括三个控制模块：输入、加工和输出。事务映射适用于单一数据项驱动多条动作路径的数据流图，DFD 对应的程序结构有一个控制模块处理事务的获取和求值，另一个控制模块处理基于事务的动作。

接口(界面)设计包括内部和外部程序接口，以及用户界面设计。内部和外部接口设计由分析模型中获得的信息作为指导。用户界面设计由任务分析和建模开始，用逐步分解或面向对象的方法定义用户任务和动作。其中讨论了一些设计问题(比如响应时间、命令格式、出错处理和帮助设施)，并精化了系统的设计模型。有许多工具都可以用于构造用户界面的原型。本章还讨论了有关一般交互、信息显示和数据输入的指南。

设计符号体系和结构化编程概念结合使得设计者可以用易于翻译成代码的方式表示过程细节。有图形、表格和文字三种符号体系用于表示设计。

本章给出的设计方法可以用于构造设计模型，可以用于开发数据结构、建立程序体系结构、定义模块和建立接口，设计方法也是以后实现设计模型的蓝图软件工程活动的基础。

思考题

14.1 写一篇三至五页的文章，给出基于问题的性质选择数据结构的指南。首先描述在软件中经常遇到的一些经典数据结构，然后针对某几类问题描述选择特殊类型的数据结构的标准。

14.2 有一些设计人员认为所有的数据流都可以当作面向变换的流。试指出将面向事务的流当作面向变换的流会对软件结构有什么影响。请举例说明要点。

14.3 完成第 12 章中的思考题 12.13，试用本章中的设计方法开发 PHTRS 的程序结构。

14.4 试提出一种用面向数据流的技术设计实时软件应用的方法。在开始讨论之前，请列出实时系统中导致面向数据流的设计方法不能直接应用的问题(比如中断驱动)。

14.5 试用数据流图和处理说明描述一个具有明显变换流特征的计算机系统。用 14.4 节中介绍的技术确定流的边界，并将 DFD 映射成软件结构。

14.6 试用数据流图和处理说明描述一个具有明显事务流特征的计算机系统。用 14.5 节中介绍的技术确定流的边界，并将 DFD 映射成软件结构。

14.7 试用课堂讨论时得到的需求完成 14.4 节和 14.5 节中 SafeHome 例子的 DFD 和体系结构设计。评估所有模块间的功能依赖性，并为设计做文档。

14.8 对于有一定编译器设计背景的读者，试开发出一个简单编译器的 DFD，评价它的流特征，并用本章中的技术导出程序结构，最后为每个模块完成处理说明。

14.9 递归模块(自己调用自己)的概念如何适应本章提出的设计原则和技术？

14.10 试将图 14－23 中的 DFD 应用于事务分析，并导出程序结构。整个图的流特征应是事务流(事务中心是变换 c)， Ⅰ区是变换流；Ⅱ区是事务流，其子变换流如图所示； Ⅲ区是变换流。在你的程序结构中，除了有必要导出一组控制模块以外，其余的模块应该与图中的变换一一对应。

14.11 试讨论在下列领域中应用面向数据流的设计方法的优点和难点：

a.嵌入式微处理器内置应用软件

b.工程/科学分析

c.计算机图形

d.操作系统设计

e.商业应用软件

f.数据库管理系统设计

g.通讯软件设计

h.编译器设计

i.进程控制应用软件

j.人工智能应用软件

14.12 由老师给出一组需求(或当前工作中某个问题的一组需求)，开发一个包括所有设计文档的完整的设计。进行设计复审(见第 8 章)，评估该设计的质量。本问题可以由一个小组完成，而不是某个人。

14.13 试举出所遇到过的一个不好的用户界面，用本章中的概念评价它；试举出所遇到过的最好的用户界面，用本章中的概念评价它。

14.14 以下列交互式应用软件之一：

a.桌面出版系统

b.计算机辅助设计系统

c.室内设计系统(如 14.9.2 节)

d.大学自动课程注册系统

e.图书馆管理系统

f.下一代选举投票箱

g.家庭银行系统

h.老师布置的一个交互式系统

为例写出其中一个系统开发用户界面的设计模型、用户模型、系统映像和系统感觉。

14.15 对思考题 14.14 中的任何一个系统进行详细的任务分析。可以用逐步分解或面向对象的方法。

14.16 继续思考题 14.15，应用本章给出的七个步骤完成界面设计过程。

14.17 针对思考题 14.15 和 14.16 得到的结果，试描述自己的帮助设施。

14.18 试举出一些例子，说明为什么响应时间的易变性会成为问题。

14.19 试开发一种可以将出错信息与用户帮助设施自动集成的方法，即系统自动识别出错信息，并在帮助窗口中提供修改建议。请完成一个相对完整的软件设计，要考虑相应的数据结构和算法。

14.20 试开发一个界面评估问题表，其中包括可以适用于大多数界面的 20 个问题。针对一个大家都用的交互系统，让 10 位同学完成该问题表。总结结果并向全班汇报。

14.21 试为 14.10 节中讨论的每一类中增加 5 条设计指南。

14.22 已有很多关于结构化程序设计的文献。试写一篇短文，指出有关完全依赖结构化构成元素进行编程的正反两方面的论据。

思考题 23 至 31 必须使用任何一种(或多种)本章给出的过程设计符号体系。具体选用何种符号体系由老师针对问题指定。

14.23 试为以下几种排序算法进行过程设计： Shell－Metzner 排序；堆排序；BSST(树)排序。如果对这几种排序算法不熟悉，可以参考有关数据结构的书。

14.24 试为基本收入税信息查询的交互界面进行过程设计。自己指定需求，并假定所有税收计算由其他模块完成。

14.25 试为针对可变分区的内存管理模式的垃圾回收功能进行过程设计。在设计表示中要定义所有需要的数据结构，其他信息可参考有关操作系统的书。

14.26 试为一个以任意长的文本为输入，以文本中所有单词及其出现频率为输出的程序进行过程设计。

14.27 试为一个求函数 f 在 a 到 b 上的数学积分的程序进行过程设计。

14.28 试为一个以四元组为输入，按指定进行输出的推广的图灵机进行过程设计。

14.29 试为解决 Hanoi 塔问题的程序进行过程设计。许多人工智能的书都对 Hanoi 塔问题进行了详细的讨论。

14.30 试为编译器中 LR 扫描器的主要或所有部分进行过程设计。可以参考有关编译方面的书。

14.31 试选择一种加密/解密算法进行过程设计。

推荐阅读文献及其他信息源

大多数探讨软件工程的书中都涉及软件设计，最严格的观点见 Feijs(Formalization ofDesign Methods，Prentice－Hall，1993)，Witt(Software Architecture and Design Principles，Thomson Publishing，1994)和 Budgen(Software Design，Addison－Wesley，1994)的专著。

全面的面向数据流的设计见 Myers、Yourdon 和 Constantine、Buhr(System Design With Ada，Prentice－Hall，1984)和 Page－Jones(The Practical Guide to Structured systemsDesign，2ndedition，Prentice－Hall，1988)的专著。这些书针对设计，并提供大量的教学实例。

有关人机界面和人为因素的文献在过去的十年里增长很快。 Thimbley(The User InterfaceDesign Book，Addison－Wesley，1989)，Barfield(The User lnterface： Concepts and Design，Addison-Wesley，1993)，Nielsen(Usability Engineering，Academic Press，1993)，Preece(A Guideto Usability， Addison－Wesley，1993)和 Lee(Object－Oriented GUI Application Development，Prentice－Hall，1994)都对这一问题进行了探讨。 Rubinstein 和 Hersh(The Human Factor，DigitalPress，1984)，Dumas[DUM88]，Helander(Ha ndbook of

Human－Computer Interaction，ElsevierScience Publishers，1988)和Laurel(The Art of Human－Computer Interface Design， Addison－Wesley，1990)则给出了界面设计的指导原则。Norman(The Psychology of Everyday Things， Basic Books，1988)用日常生活中的例子来说明好的设计和坏的设计，他认为好的界面设计很好地考虑到了人们的感知能力，坏的界面设计则相反。

Linger，Mills 和 Witt 的工作(Structured Programming－TheoryandPractice，Addison－Wesley，1979)仍然是这方面的基础，其中包括了一个不错的 PDL，并对结构化编程的各种流派进行了讨论。其他关于过程设计的书还有 Bentley 的(Programming Pearls，Addison－Wesley，1986 和MoreProgramming Pearls,Addison－Wesley,1988)以及 Dahl,Dijkstra 和 Hoare 的(StructuredProgramming，Academic Press，1972)。

有关软件设计方法的文章和讨论可以从 Association forSoftware Design 的站点上得到：

http：//www－pcd.stanford.edu/asd/info/articles/

有关软件体系结构技术的书目和讨论以及“软件体系结构技术指南”可由以下站点得到：

http：//www.stars.reston.unisysgsg.com/arch/guide.html

http ：//www.sci.cmu.edu/technology/architecture/bibliography.html

Geo WiederHold 的经典著作 Data base Design,3rd edition,McGraw-Hill,1988 的在线版本可由下面站点得到：

http：//db.stanford.edu/pub/gio/dbd/intro.html

简单的教学资料“Comments on Human Computer Interface Design”可由下面站点得到：

http：//infolabwww.kub.nl：2080/w3thesis/Hci/user_centered.html

Lewis 和 Rieman 提供了一本联机教材，题目为 Task Centered User Interface Design，可由下面站点得到：

ftp://ftp.cs.colorado.edu/pub/cs/distribs/clewis/HCl-Design-Book/

WWW上最新的关于软件设计方法的文献列表可从http ：//www.rspa.com上得到。

① 在阅读本章之前应先阅读第 11 章和第 12 章。

① 常规的编程语言通常采用参数表示进行接口上的数据传递；面向对象语言使用消息实现数据传递。在这两种情况下设计要考虑的问题是不同的。

① 当然，有时也并不是这样。对于交互式系统来说，界面设计和数据、体系结构、过程的设计一样重要。

② 这里的“语法了解”是指有效使用界面所需的交互机制。

③ “语义了解”是指该应用程序的内在含义，即对执行的功能、输入输出的含义、系统的目的理解。

① 面向对象的观点在本书的第 4 部分有进一步的讨论。

① 值得注意的是，在某些情况下（飞机驾驶舱显示器）第一步应足在界面上驱动显示设备，而不是用驾驶舱显示器建立原型。

① 结构化编程的一个很少使用但非常重要的特性是“正确性证明”。可以在过程设计完成后就证明其正确性。有关设计的正确性验证的介绍见第 25 章。

第 15 章　实时系统的设计

实时计算系统的设计是一个软件工程师所能从事的最有挑战性和最为复杂的任务。由于这个特点，用于实时系统的软件需要其他应用领域没有涉及到的分析、设计和测试技术。

实时软件是外部世界高度耦合的，也就是说，实时软件必须在问题域规定的时间框架内对该问题域(现实世界)作出响应。由于实时软件必须在严格的性能约束下操作，因此软件的设计常常是被硬件及软件体系结构、操作系统特性、应用需求和编程语言的变化所驱动的。

Robert Glass[GLA83]在他关于实时软件的书中，对实时系统主题给出了一段很有帮助的介绍：

数字计算机在我们所有人的日常生活中正变得日益普及。计算机不但可以让我们玩游戏，还可以报时、优化最新一代汽车的汽油里程以及控制我们的家用电器……[在工业上，计算机可以控制机器、协调过程，并逐渐用自动化系统和“人工智能”来代替手工操作和人的参与。]

所有上述的计算——不论是有帮助的还是强行引入的——都是实时计算的例子。计算机正在控制着某些东西与外部世界的实时交互，事实上，时间是交互的核心……反应迟钝的实时系统比完全没有系统还要糟糕。

就在十年前，实时软件开发还被认为是一种黑色的艺术，它的从事者是那些充满嫉妒地保卫着他们封闭世界的巫师们。如今，巫师们已经供不应求了！然而，实时软件的开发毫无疑问需要特殊的技能。在本章里我们将对实时软件进行研究，并讨论建造实时软件所需的某些开发技能。

15.1 系统考虑

与其他任何基于计算机的系统一样，一个实时系统必须将硬件、软件、人力和数据库元素集成起来，以恰当地实现一组功能和性能需求。在第 10 章中，我们探讨了基于计算机的系统任务分配，并指出，系统工程师必须对系统元素分配功能和性能。实时系统的问题在于恰当的分配。实时性能常常与功能一样重要，但却很难有把握地作出与性能相关的分配决策。一个处理算法能满足严格的时间

约束吗，或者说我们应该建造特殊的硬件来完成这个工作？一个购买来的操作系统能够满足我们进行高效的中断处理、多任务和通信的需求吗，或者说我们应该使用自定义的执行程序？与推荐的软件配对的特定硬件能够满足性能标准吗？所有这些以及其他许多问题都需要由实时系统工程师来回答。

对实时系统所有成分的详细讨论已经超出了本书的范围，在[SAV85]、[ELL94]和[SEL94]等文献中有大量好的有关这方面的信息，但是，在讨论软件分析和设计问题前我们对实时系统的各个元素能有所理解还是非常重要的。

Everett[EVE95]定义了实时软件开发不同于其他软件工程的三个特征：

• 实时系统的设计是受资源约束的。时间是实时系统的首要资源，关键是要在指定数目的 CPU 周期内完成一个定义好的任务，除此以外，其他系统资源，如内存大小等，在实现系统目标时都有可能和时间进行折衷。

• 实时系统是紧凑而复杂的。尽管一个复杂的实时系统可能包含上百万行的代码，但软件中有关时间标准的代码一般只占很小一部分。这一小部分代码是最为复杂的(从算法的角度来说)。

• 实时系统的运行常常不需要用户的参与。因此，实时软件必须能检测到导致故障的问题，并在对数据和控制环境造成破坏前改正这些问题。

在下面一节中，我们将探讨一下实时系统不同于其他类型计算机软件的一些关键性属性。

15.2 实时系统

实时系统产生某种动作以响应外部世界。为了完成这个功能，它们能高速地获取数据，并在严格的时间和可靠性约束控制下。由于这些约束是如此苛刻，实时系统通常只用于满足单个的应用。

实时系统广泛地用于各种应用领域，包括军用的命令与控制系统、消费者电器、过程控制、工业自动化、医疗和科学研究、计算机图形、局域和广域通信、航天系统、计算机辅助测试以及大量的工业仪器。

15.2.1 集成和性能问题

为了将一个实时系统组织在一起，系统工程师需要作出困难的硬件和软件决策。(实时系统与硬件相关的分配问题超出了本书的范围；更多的信息请参阅文献[SAV85]。)一旦分配好软件成分，就要建立详细的软件需求，并必须开发出一个基本的软件设计。许多实时设计关心的是实时任务间的协调、系统中断的处理、保证不丢失数据的 I/O 处理、指定系统的内部和外部时间约束、以及确保数据库的准确度等。

实时设计关注的每个部分都必须应用到系统性能这个语境中。在大多数情况下，一个实时系统的性能是由一个或多个与时间相关的特征来测度的，但也可能用容错性之类的指标来测度。

某些实时系统是设计来用于那些只关注反应时间或数据传输率的应用中，其他实时应用还需要对峰值负载条件下的这两个参数进行优化，而且，实时系统必须在执行一系列并发任务时处理它们的峰值负载。

由于一个实时系统的性能主要由系统响应时间和它的数据传输率决定，所以理解这两个参数是很重要的。系统的响应时间是从系统检测到一个内部或外部事件到发出响应动作这段时间，事件检测和反应生成常常是很简单的，对事件信息进行处理以判断合适的反应往往会涉及到复杂耗时的算法。

语境切换和中断等待时间是影响反应时间的两个关键参数。语境切换包括在任务间切换的时间和系统开销，而中断等待时间是实际的切换发生之前的延迟时间。计算速度和大容量存储介质的存取速度也会影响到系统的反应时间。

数据传输率指的是串行/并行数据以及模拟/数字信号进出系统的速度。硬件厂商常常援引时间和容量值作为性能指数，但是，硬件的性能规范通常是单独衡量的，而且一般对整个实时系统的性能影响甚微。因此，I/O 设备的性能、总线延迟、缓冲区大小、磁盘性能及其他一系列因素，尽管很重要，但也只占实时系统设计的一小部分。

实时系统常常被用来处理一个连续的输入数据流，设计必须保证数据不会丢失，除此以外，实时系统必须要响应异步事件，因此，到达序列和数据容量是很难事先预测的。

尽管所有软件都必须要求可靠，但实时系统在可靠性、重启动和故障恢复方面有更为特别的要求。由于它监视和控制着现实世界，因此失去监视或控制在许多情况下是不可忍受的(例如航空交通控制系统)。于是，实时系统都包含有重启动和故障恢复机制，并通常有用于备份的内置冗余。

但是，对可靠性的要求已经引发了一场关于联机系统(如机票预订系统和自动银行出纳等)是否也可以算作实时系统的争论。一方面，这样的联机系统必须在规定的时间内以秒级时间响应外部中断，另一方面，如果联机系统无法满足响应需求，并不会有任何大灾难发生，只是会造成系统性能的降低。

15.2.2 中断处理

中断处理是实时系统不同于其他任何类型系统的一大特性。实时系统在外部世界规定的时间框架内响应外部刺激——中断。由于多种刺激(中断)常常并存，必须建立优先级和优先级中断，换句话说，最重要的任务必须比其他事件优先在预定时间内得到服务。

中断处理不仅需要保存信息，以使计算机能够重新开始被中断的任务，而且要避免死锁和无限循环，图 15－1 说明了中断处理的整个过程。正常的处理流是被处理器硬件检测到的一个事件“中断”，由硬件或软件产生的任何需要立即服务的要求都称为事件，中断程序的状态被保存起来(即所有寄存器的内容、控制块等等)，控制被传递给一个中断服务例程，它将控制转移到一个适当的处理中断的软件，一旦完成中断服务，将恢复机器的状态，继续进行正常的处理流。

在许多情况下，一个事件的中断服务自身可能会被另一个更高优先级的事件所中断，可以建立中断优先级(图 15－2)。如果一个低优先级的进程偶然被允许中断一个更高优先级的进程，再用正确顺序重新启动进程就可能很困难了，并且可能导致无限循环。

为了在满足系统时间约束的前提下处理中断，许多实时操作系统进行动态计算，以决定是否能满足系统目标。这些动态计算是基于事件的平均出现频率、占用的服务时间(如果它们被服务到的话)以及可能会中断它们和暂时阻止它们服务的例程。

如果动态计算证明，在满足时间约束的前提下不可能处理系统中发生的事件，系统就必须决定一个动作计划。一种可能的办法是先将数据送入缓冲区，等到系统准备就绪时再来快速处理它们。

15.2.3 实时数据库

象许多数据处理系统一样，实时系统常常具有数据库管理功能，但是，在实时系统中分布式数据库似乎是首选，因为实时系统中多任务是很常见的，并且数据常常是并行处理。如果数据库是分布式的，各个任务就可以更快更可靠地访问到库中的数据，瓶颈也比中心式数据库少。在实时应用中使用分布式数据库，将划分开输入/输出“交通”，并缩短等待访问数据库的任务队列。而且，如果有内置的冗余的话，一个数据库发生故障很少会引起整个系统的瘫痪。

通过使用分布式数据库可以提高性能，但也有数据分区和复制造成的潜在问题。尽管数据冗余通过提供多个信息源提高了响应时间，但分布式文件的复制需求也造成了系统的开销，因为所有的文件副本都必须被更新。此外，使用分布式数据库还引入了并发控制问题，并发控制涉及到数据库的同步，以便所有副本都有正确的、相同的信息可供访问。

并发控制的传统方法是基于锁定和时间戳。系统定期执行下列任务：(1)“锁住”数据库，以保证控制并发；不允许 I/O；(2)必要时进行更新；(3)给数据库解锁；(4)确认文件以确保正确进行了所有更新；(5)认可完整的更新。所有上锁的任务都由一个主时钟监控着(即时间戳)，这些过程产生的延迟以及避免不一致的更新和死锁的问题，影响了分布式数据库的广泛使用。

但是已经开发了一些用于加速更新和解决并发问题的技术，其中有一个被称为唯一写者协议(exclusive－writer protocal)，它通过只允许一个唯一的写任

务更新文件来维护被复制文件的一致性，从而避免了上锁或时间戳过程的大量开销。

15.2.4 实时操作系统

一些实时操作系统(RTOS)被应用到大量的系统配置中，而有些操作系统则被固化在一块特定的电路板甚至微处理器上，不用考虑周围的电子环境。RTOS 通过将软件特性和(逐渐增多)在硬件中实现的各种微码功能结合起来实现了自己的功能。

现在，在实时应用中有两大类操作系统：(1)专为实时应用设计的专用操作系统，(2)增加了实时功能的通用操作系统。实时执行程序(executive)的使用使得通用操作系统的实时性能变得可行，执行程序在行为上类似于一个应用软件，它负责执行大量的操作系统功能——尤其是那些影响实时性能的功能——它比通用操作系统更快、更有效率。

所有操作系统都必须有一个优先级调度机制，但 RTOS 必须提供一个允许高优先级中断低优先级中断的优先级机制，而且，由于中断响应是异步的、不可再现的事件，因此它们必须被服务，而不应该先花时间将程序从磁盘存储调换到内存。为了保证所需的响应时间，实时操作系统必须有一个内存锁定的机制——也就是说，在主存中常驻一些程序以避免调换的开销。

为了决定哪类操作系统最适合于一个应用需要，定义和评价 RTOS 的测度可以。语境切换时间和中断等待(前面讨论的)决定了中断处理的能力，这是一个实时系统最重要的方面。语境切换时间是操作系统用来保存计算机状态和寄存器内容，以便在中断服务结束后能返回到正在处理的任务的时间。

中断等待，是系统着手去切换一个任务之前的最大延迟时间，这是因为在操作系统中经常有不可再入的或关键的处理路径，它们必须在中断处理前被完成。

系统处理中断前的这些路径的长度(指令数目)代表着最坏情况下的时间延迟。最坏的情况是，当系统刚进入一个中断和中断服务之间的关键路径，这时又发生了更高优先级的中断，如果时间太长，系统可能会丢失那些不可恢复的数据，因此，设计者知道这个延迟时间是很重要的，这样才能让系统对此作出补偿。

许多操作系统会执行多任务[WOO90]和并发处理，这是实时系统的另一个重要需求，但为了让实时操作变得可行，系统开销相对于切换时间和占用的内存空间来说必须很低。

15.2.5 实时语言

由于实时系统对性能和可靠性的特殊需求，程序设计语言的选择也是很重要的。许多通用的程序设计语言(如 C、Fortran、Modula－2)对实时应用是有效的。

然而，有一类所谓的“实时语言”(如 Ada、Jovial、HAL/S、Chill 等)经常被用在专门的军事和通信应用上。

各种特点的结合使得实时语言有别于通用语言，这包括多任务能力、直接实现实时功能的结构和帮助保证程序正确性的现代程序设计特征。

直接支持多任务的编程语言是很重要的，因为实时系统必须响应异步事件。尽管许多 RTOS 提供了多任务能力，但内嵌的实时软件经常独立于操作系统而存在，而内嵌的应用软件是用为实时程序执行提供足够运行时刻支持的语言编写的，运行时刻支持比操作系统需要的内存更少，它可以针对应用来裁剪，从而增强了性能。

15.2.6 任务同步和通信

一个多任务系统必须具有在任务间传递消息和保证任务同步的机制，为此，操作系统和具有支持运行时刻的语言一般使用信号量队列、邮箱或消息系统来完成同步。信号量使得并发任务能够同步，它提供同步和信号发出，但不包含信息。消息与信号量的不同之处在于消息带有相关的信息，而邮箱，则不发出信号，仅包含信息。

信号量排队(queuing semaphores)是帮助管理通信的软件原语，它们提供了一种管理多个队列的方法——例如，等待资源、数据库访问和设备的任务队列，以及资源和设备队列。信号量负责协调(同步)正在等待的任务，不论它们在等待什么，任务和资源之间都不会互相干扰。

在实时系统中，信号量常用来实现和管理邮箱，邮箱用于暂时存放一个进程发送给另一个进程的消息(也称为消息池或缓冲区)，进程产生一条消息后，将它放入邮箱中，接着用信号通知一个要用消息的进程：邮箱中有一条消息可供使用。

一些实时操作系统或支持的运行时刻系统将邮箱看作进程间效率最高的通信方式。一些实时操作系统有一个发送和接收邮箱中数据指针的地方，这样就避免了传送所有的数据——从而节省了时间和资源开销。

进程间通信和同步的第三种方法是消息系统，一个进程可以利用消息系统发送消息给另一个进程，接着后者被自动运行时支持系统，或由操作系统激活来处理这条消息，这样的系统带来了开销问题，因为它需要传送实际的信息，但它也提供了更大的灵活性并易于使用。

15.3 实时系统的分析和仿真

在前面一节中，我们讨论了一组与实时系统的功能需求不可分离的动态属性：

• 中断处理和语境切换。

• 响应时间。

• 数据传输率和吞吐量。

• 资源分配和优先级处理。

• 任务同步和任务间通信。

所有这些性能属性都可以被详细说明，但要验证是否系统成分会达到满意的响应时间、是否有足够的系统资源来满足计算需求、或处理算法的执行速度是否足够快，这些都是极为困难的。

对实时系统的分析需要建模和仿真，才能使系统工程师能够访问到“时间长短和资源大小”的问题。尽管在文献中提出了大量的分析技术(如参考文献[LIU90]、[WIL90]和[ZUC89])，但公正地说，对实时系统分析和设计的分析方法还处在它们的初级阶段。

15.3.1 实时系统分析的数学工具

Thomas McCabe[MCC85]已提出了一组数学工具，使得系统工程师能够对实时系统元素进行建模，并评价时间长短和资源大小问题。这组数学工具大致根据数据流分析技术(第 12 章)，McCabe 的方法使分析者能够对实时系统的硬件和软件元素建模；以概率的方式表示控制；并应用网络分析、排队论、图论和一个 Markovian 数学模型[GRO85]来推导出系统时间和资源大小。不幸的是，这里牵涉的数学知识已超出了本书的范围，要详细阐明 McCabe 的工作就变得很困难了，但是，对这项技术的综述将提供一个对实时系统工程的分析方法很有价值的认识视角。

McCabe 的实时分析技术依据的是实时系统的数据流模型，但是，McCabe[MCC85]没有使用传统的 DFD 方式，而是主张一个 DFD 中的加工(泡泡)可以表示为一个加工状态的 Markov 链(一种概率排队模型)，数据流自身表示加工状态之间的变迁。分析者可以给每条数据流路径赋予变迁概率，如图 15－3 所示，可以为每条流路径指定一个值，

$$0 < p_{ij} \leqslant 1.0$$

这里p_{ij}表示加工pi和Pj之间的流发生的概率。这些加工对应于DFD中的信息变换(泡泡)。

可以给定 DFD 类模型中的每个加工一个“单价”，表示执行它的功能估计需要的(或实际需要的)执行时间，还可以有一个“入口值”，用于描述该加工对应的系统中断数量。接着要使用一组数学工具来分析该模型，工具要计算(1)对一

个加工期望的访问次数，(2)从一个特定加工开始处理时所花的系统时间，(3)系统所花的总时间。

为了用一个实际的例子说明McCabe技术，我们以图 15－4 中一个电子对抗系统的DFD为例。数据流用的是标准形式，但数据流的标识被p_{ij}替换了。图 15－5 中显示了一个从DFD导出的队列网络模型，数值λ_i对应于每个加工的到达率(每秒到达数)，根据遇到的队列的类型，分析者必须决定各种统计信息，如平均服务率(每个加工的平均运行时间)、服务率的方差、到达率的方差等等。

每个加工的到达率是由流路径的概率p_{ij}和系统的到达率λ_{in}决定的。可导出一组流平衡方程来计算经过每个加工的流，对图 15－5 所示的例子，有如下的流平衡方程 [MCC85]

$$\lambda_1 = \lambda_{in} + \lambda_4$$

$$\lambda_2 = p_{12}\lambda_1$$

$$\lambda_3 = p_{13}\lambda_1 + p_{23}\lambda_2$$

$$\lambda_4 = p_{64}\lambda_6$$

$$\lambda_5 = P_{25}\lambda_2 = \lambda_3$$

$$\lambda_6 = \lambda_5$$

$$\lambda_7 = P_{67}\lambda_6$$

根据图中所示的P_{ij}以及到达率，$\lambda_{in}=5$，以上方程可以得到解出：

$$\lambda_1 = 8.3$$

$$\lambda_2 = 5.8$$

$$\lambda_3 = 5.4$$

$$\lambda_4 = 3.3$$

$$\lambda_5 = 8.3$$

$$\lambda_6 = 8.3$$

$$\lambda_7 = 5.0$$

计算出到达率后，可以使用标准排队论来计算系统时间，每个子系统(队列Q和服务器S)可以利用队列类型对应的公式来评价。对 m/m/1 队列 [KLI75]：

利用率：ρ＝λ/μ

期望的队列长度：$N_q=\rho^2/(1-\rho)$

期望的子系统数：$N_s=\rho_1/(1-\rho)$

队列期望的时间：$T_q=\lambda/(\mu(\mu-\lambda))$

子系统期望的时间：$T_s=1/(\mu-\lambda)$

μ 是完成率(完成数/秒)。应用标准的排队网络归约规则，如图 15-6 所示，从数据流图(图 15-4)导出的最初的网络队列可以用图 15-7 所示的步骤来简化，系统所花的总时间是 2.37 秒。

很明显，McCabe 分析方法的精度只能依赖于流概率、到达率和完成率的估计水准，但是，通过在分析时采用更具时间系统分析观点的方法来进行分析，确实可以达到显著的效益。引用 McCabe 的话如下［MCC85］：

通过改变象到达率、中断率、拆分概率、优先级结构、队列规程、配置、需求、物理实现和方差等变量，我们可以很容易地向程序管理者展示：影响这些变量的因素将影响手边的系统。这些迭代的方法论对于填补实时规约建模的空白是必要的。

(a)串行规则——被子系统串行服务的每个到达

(b)并行规则——被子系统并行服务的每个到达

$$T_{并行}=\frac{p_1T_1+p_2T_2}{p_1+p_2}$$

P_1=进入服务器 1 系统的概率

P_2=进入服务器 2 系统的概率

(c)循环规则——延迟为 T(λ μ)的服务器和循环概率为 P 的“反馈”循环

15.3.2 仿真和建模技术

实时系统的数学分析代表了一种可用于理解项目性能的方法，然而，越来越多的实时软件开发者使用了仿真和建模工具，这些工具不仅能分析系统的性能，还能使软件工程师建立一个原型、执行它并由此获得对系统行为的理解。

对实时系统仿真和建模的基本原理曾由 I－LogiX(一个为系统工程师开发工具的公司)给出过如下讨论［ILO89］：

在一个项目的设计、实现和测试阶段，经常需要通过反复的试验和排错来理解系统在其环境中的时间行为。Statemate(一个用于仿真和建模的系统工程工具)方法提供了一种对这个昂贵过程的替换方法。它允许你建立一个全面的系统模型，这个模型足够精确可靠，足够清晰有用。这个模型涉及到通常的功能和流问题，但也涉及了一个系统动态行为。可以用 Statemate 分析和检索工具来测试这个模型，这些工具提供了检查和调试规约并从中检索信息的扩展机制，通过测试实现模型，系统工程师可以看到被规约描述的系统在实现后的行为。

LOgiX方法 [HAR90]使用了一种组合系统三种不同视图的符号体系：活动图、模块图和状态图，下面一段将描述用于实时系统仿真和建模的i－LogiX方法。[①]

概念视图

功能问题是用表示系统处理能力的活动来体现的。在航空订票系统中处理客户的确认请求以及在一个航空电子系统中更新飞机的位置都是活动的实例。活动可以被嵌套，从而形成了系统功能分解的层次。信息项，如到目的地的距离或客户的名字，一般会在活动之间流动，也有可能保存在数据库中。活动图体现了系统的功能视图，它类似于传统的数据流图。

动态行为问题，通常涉及控制方面，是用状态图来体现的，这种符号体系是由 Harel 和他的同事开发的 [HAR88] [HAR92] 。在这里，状态(或模式)可以被用一系列表示顺序或并发行为的方式来嵌套和链接，例如，一台承担航空电子使命的计算机可能有三种状态：空对空、空对地和导航，同时它必须处于自动或手工飞行控制状态。状态间的变迁通常由事件触发，事件还可以有限制条件，例如，在风门上敲击一个特定的开关，是一个会引起从导航状态到空对地状态的变迁事件，但条件是飞机有可用的空对地导航。再举个简单的例子，如图 15－8 中的数字手表，图 15－9 显示了手表的状态图。

一个系统的这两种视图是按下面的方式构集的。活动图的每层通常关联了一个被称为控制活动的状态图，它的角色是控制该层的活动和数据流(这在某些方面类似于第 12 章所说的流模型和 CSPEC)。状态图能够对活动施加控制，例如，它能指导活动的开始和终止，以及暂停和恢复，它能改变变量的值，并因此影响活动的处理，它还能向其他活动发送信号并因此使它们改变自己的行为。除了能够产生行为外，控制状态图还可以检测到其他状态图正在实施的行为，例如，如果一个状态图开始了一个活动或增加了一个变量的值，另一个状态图就可以检测到这个事件并利用它来触发一个变迁。

认识到活动图和状态图紧密关联是很重要的，但它们并非同一事物的不同表示。作为系统模型，活动图自身是不完整的，因为它没有涉及到行为；状态图也是不完整的，因为如果没有活动它就没有可控制的东西。总的来说，一个详细的活动图和它的控制状态图提供了概念模型，活动图是模型的主干；它的系统能力分解构成了规约的主要层次，而它的控制状态图是系统行为背后的驱动力。

物理视图

一个以概念模型形式使用活动图和状态图的规约是一个很好的基础，但它不是一个真正的系统，它缺少一种从物理(实现)视角描述系统的方式——一种可以确信系统的实现，使该规约有效的方式。物理视角描述的一个重要方面是描述系统的物理分解以及它与概念模型的关系。

Statetmate 中是用模块图语言来体现物理特征的。术语“物理”和“模块”一般用于表示系统的构件，可以是硬件、软件或者二者的混合。同活动图中的活动一样，模块是以层次方式排列的，表示一个系统被分解为它的构件和子构件。模块用流线连接，流线可以看作是模块间信息的载体。

分析和仿真

我们建立了由一个活动图和它的控制状态图构成的概念模型后，就可以对它进行彻底的分析和测试了。模型可能描述的是整个系统，向下一直到最底层的细节，或者它只是一个部分的规约。

我们首先必须确信，这个模型在语法上是正确的。这引发了许多比较简单的测试：例如，各种图在表面上应该是完整的(如不会缺少标记或名字，不会有悬浮的箭头等)；事件和条件等非图形元素的定义，只能使用合法的操作等等。语法检查也会涉及到更加精细的测试。如输入和输出的正确性，例如，测试在状态图中使用过但既不是输入也不被内部影响的元素。如开电源事件，它会引起状态图中的一个变迁，但并没有在活动图中被定义为输入。所有这些通常被称为一致性和完整性测试，其中的大部分类似于编译器在实际编译一个语言之前执行的检查。

运行样例

一个语法正确的模型精确地描述了某个系统，但它也可能不是我们想要的系统。事实上，被描述的系统可能存在严重的缺陷——语法正确并不能保证功能或行为的正确。分析这个模型的真正目的，是看它是否真的描述了我们想要的系统。分析应该使我们能更深入地了解被构造的系统，检验基于它的系统的行为，并验证它是否真的满足需要。我们还需要一个语法更具形式化的建模语言，需要创建模型的系统也能识别形式化语义(Semantics)。

如果这个模型是基于一个形式的语义，系统工程师就可以执行这个模型。工程师能够创建并运行一个样例，允许他在系统实际开发出来之前，就可以“按按钮”并观察模型的行为。例如，要试验一个自动取款机(ATM)模型，步骤如下：(1)创建一个概念模型；(2)工程师扮演客户和银行电脑的角色，产生诸如插入一张银行卡、按下按钮及新的结余信息到达等事件；(3)监控系统对这些事件的反应；(4)记录下行为的不一致之处；(5)修改概念模型以反映正确的行为；(6)重复以上步骤直到演化出所需的系统。

系统工程师运行样例，并用图形化的方式观察系统的反应。模型的“活动”元素(如系统此刻所处的状态和当前的活动)被高亮度显示，动态的执行结果以生动的模型表示出来。对样例的执行模拟了实时运行的系统，并记录了依赖于时间

的信息。在执行中的任意一点，工程师都能够看到任何非图形元素的状态，如一个变量或一个条件的值。

编程的仿真

样例使得系统工程师能够交互式地试验模型，但有时还需要更广泛的仿真。可能需要评估典型和非典型情况时随机条件下的性能。面对实时系统需要更广泛的仿真的情况，仿真控制语言(SCL)使得工程师能对执行进度保持整体的控制，同时利用工具的能力来接管许多细节。

用 SCL 能做的最简单的事情之一就是从一个批处理文件中读取事件列表，这意味着要事先准备好冗长的样例再自动执行，这些都能被系统工程师遵从。系统工程师也可以用 SCL 来编程设置端点并监控特定的变量、状态或条件，例如，在运行一个航空电子系统的仿真时，工程师可以让 SCL 程序在雷达锁定目标时停下来，切换到交互模式，一旦遇到“锁定”，工程师就交互式地接管过来，这样就能更加详细地检查这个状态。

使用样例和仿真也能使工程师收集到关于要开发的系统的操作的有意义的统计数字。例如，我们可能想知道，在飞机一次飞行中，雷达有多少次丢失了锁定的目标。由于系统工程师可能很难组起一次单一的、完全封装的飞行样例，因此可以开发一个模拟的程序，利用其他样例汇集的结果来获得平均情况下的统计数字。仿真控制程序根据预先定义的概率产生随机事件，因此，很少发生的事件(比如说，飞行中的座位弹射)可以被赋予很低的概率，而其他事件可以被赋予较高一些的概率，从而使事件的随机选择成为了现实。为了收集所需的统计数字，我们在 SCL 程序中插入了适当的断点。

自动翻译为代码

系统模型建立起来后，利用原型函数可以将它整个翻译为可执行的代码。活动图和它们的控制状态图可以被翻译成一种高级的程序设计语言，如 Ada 或 C。如今，产生的结果代码的主要用处是在尽可能接近现实世界的环境下观察系统的运行情况，例如，原型代码可以在目标环境成熟的仿真器中或在最终环境中执行。这类 CASE 工具产生的代码应该被看作是“原型化的”，而不是产品或最终代码，它可能不一定反映了所需系统精确的实时性能，但是，可以用它在接近于现实的环境中测试系统的性能。

15.4 实时设计

实时软件的设计必须具体体现高质量软件所具有的所有基本概念和原则(第 13 章)，除此以外，实时软件还给设计者带来了以下一系列独特的问题：

- 中断和语境切换的表示。
- 多任务和多处理所需的并发。

• 任务间通信和同步。

• 数据和通信速率大幅度的变化。

• 时间约束的表示。

• 异步处理。

• 与操作系统、硬件及其他外部系统元素间必要和不可避免的耦合。

为实时系统的设计给出一组专门的设计原则是很有价值的，Kurki－Suono [KUR93] 讨论了实时(“受激反应式”)软件的设计模型：

所有的推理，不论是形式化的还是直觉的，都由一些抽象来完成，因此，理解哪类特性在抽象中是可表达的就很重要了。以一个受激反应式系统为例，该产品强调对形式化方法的更直接需要和强调，它应该使用的模型还没有一个公论的事实。严格的形式化方法既包括进程代数和时序逻辑，也包括具体的状态模型和 Petri 网，不同的学派仍在不停地争论它们各自的优点。

接着，他定义了在实时软件设计中应该考虑的一些“建模原则 [KUR93]：

显式的原子性。在实时设计模型中明显地定义“原子操作”是必要的。一个原子操作或事件是一个约束严格的、有限的功能，它可以被一个单一的任务执行，也可以被几个任务并发执行。一个原子操作只能被那些需要它的任务(“参与者”)激活，而且它的执行结果只会影响到那些参与者，系统的其他部分不会受到影响。

交错(interleaving)。尽管处理可以并发，但某些计算的历史应该用线性操作序列的方式刻划出来。从初始状态开始，第一个操作被启动和执行，这个操作改变了状态，于是第二个操作发生了。由于在一个给定的状态里多个操作都可能会发生，因此从同一个初始状态可能产生出不同的结果(历史)。“这种非确定性是并发交错建模的本质” [KUR93]。

非终止的历史和完美性。受激反应式系统的处理历史被假设为无限的，也就是说，处理无限继续，或者“原地不动(stutter)”，直到某个事件使它继续处理下去。完美性避免了系统在任意一点中止。

封闭的系统原则。实时系统的设计模型应该包含了软件和软件驻留的环境，“动作因此能划分为那些系统自身可以负责的部分，以及那些环境执行的部分” [KUR93]。

状态的结构。一个实时系统可以被模型描述为一组对象，其中每个对象都具有自身的状态。

在实时系统的设计演化中，软件工程师应该考虑上面提到的每个概念。

在过去的二十年里，为了解决上面提到的问题，出现了大量的实时软件设计方法。一些实时设计方法扩展了第 14 章和第 21 章中讨论的方法(如数据流［WAR85］［HAT87］、数据结构［JAC83］或面向对象［LEV90］)方法。还有一些引入了一整套独立的方法，利用有限状态机模型、消息传递系统、Petri 网或一种专门的语言作为设计的基础。对实时系统软件设计的深入讨论已超出了本书的范围。要进一步了解细节，请参考文献［LEV90］、［SHU92］、［SEL94］和[GOM95]。

15.5 小结

实时软件的设计不仅包含了传统软件设计的所有方面，还引入了一套新的设计标准和关注点。由于实时软件必须在外部事件规定的时间范围内对事件作出响应，因此各种类型的设计将变得更为复杂。

将软件设计与更大的面向系统的问题孤立开来是很困难、而且常常是不现实的。实时软件设计者必须考虑硬件和软件的功能和性能，因为实时软件要么是时钟驱动，要么是事件驱动。中断处理和数据传输率、分布式数据库和操作系统、专门的程序设计语言和同步方法都只是实时系统设计者关注的某个部分。

对实时系统的分析既包括数学模型，又包括仿真。排队和网络模型使得系统工程师能够评价总体响应时间、处理率以及其他时间和大小问题。形式化的分析工具为实时系统提供了仿真机制。

实时系统的软件设计可基于传统的设计方法学来扩展，其中通过提供一种强调实时系统特征的符号体系和方法，对面向流或面向对象的设计进行了扩展。设计方法也可以利用独特的符号体系或应用专门的语言来实现。

实时系统的软件设计仍然是一项挑战。有所进步的，方法也确实存在，但对目前发展水平的现实的评价表明，还有许多事情需要去做。

思考题

15. 1 列出五个基于计算机的实时系统的例子。说明系统有哪些“刺激(stimuli)”，以及系统控制或监视了什么环境。

15.2 获取关于商用实时操作系统的信息(RTOS)，并写一篇短文来讨论 RTOS 的内核。它有什么特殊的特征？中断处理是怎样的？RTOS 是如何实现任务同步的？

15.3 写一篇对程序设计语言 Ada 和 Modula－2 的实时构成成分进行比较的文章。这些构成成分提供了显著的超越 C 或 Pascal 的优点吗？

15.4 给出三个信号量作为任务同步机制的例子。

15.5 15.3 节介绍的实时系统分析技术假定了排队模型的知识，使用指定的参考文献进行研究并：

a. 描述图 15－5 如何从图 15－4 导出。

b. 说明流平衡方程如何从图 15－5 导出。

15.6 获取一个或多个实时系统形式化分析工具的信息(15.3.2 节)，写一篇文章，简述每个工具在实时系统规约和设计中的使用。

① 以H. J. Hinden和W..RauschHinden所著的“实时系统”[HIN83]为基础，该书的使用权已由电子设计和Heyden出版社授予。

① 以下描述Statemate的文字改编自［ILO89］，由i－Logix公司授权使用。

第 16 章　软件测试技术

软件测试的重要性及其对软件质量的好坏的预意是非常重要的。下面这段话引自 Deutsch［DEU79］：性及其对软件质量的好坏的预意是非常重要的。下面这段话引自 Deutsch[DEU79]：

软件系统的开发包括一系列生产活动，其中由人带来的错误因素非常多。错误可能出现在程序的最初…，其时目标可能是错误的或描述不完整，也可能在后期的设计和开发阶段…，因为人们不能完好无缺地工作和交流，软件开发过程中必须伴有质量保证活动。

软件测试是软件质量保证的关键元素，代表了规约、设计和编码的最终检查。

软件作为系统元素的可见性不断增加软件故障带来的代价太高使得人们注重于规划良好的彻底测试，软件开发组织将 30％—40%的项目精力花在测试上并不为怪。另一方面，人命悠关的软件(如飞行控制和核反应堆)测试所花的时间往往是其他软件工程活动时间之和的三到五倍。

本章讨论软件测试基础和设计软件测试用例的技术。软件测试基础定义了软件测试的目标，测试用例的设计讨论符合整体目标的测试用例创建技术。第 17 章讨论测试策略和软件调试。

16.1 软件测试基础

测试为软件工程师带来了很有趣的意外。在软件过程的早期，软件工程师试图由抽象概念到具体实现来建立软件，现在来了测试，工程师创建测试用例试图“摧毁”已经建立的软件。事实上，在软件工程过程中，测试可以看成(至少心理上)摧毁性的而不是建设性的。

软件开发者就其本性而言是建设者，测试要求开发者放弃刚开发的软件是正确的观念，并克服发现错误时的心理矛盾。Beizers [BEI90] 如下描述了这种情况：

如果我们真正擅长编程，就应当不会有错误，但这只是一个神话。如果我们真的很认真，如果每个人都使用结构化方法，自顶向下设计而且使用决策表，如果程序是用 SQUISH 写的，如果我们有合适的银弹，就不会有错误了，这样，神话就不存在。因为我们并不擅长所做的事，所以有错误，如果不擅长，就应当感到内疚。因此，测试和测试用例设计是对失误的承认，它注入了一针内疚剂。测试的枯燥是对我们错误的处罚，为什么处罚？为了人？为什么内疚？为了没能达到人类的完美境界？为了没有区别另一个程序员所想的和所说的？为了没有心灵感应？为了没有解决人类四千年来尚未解决的相互通信问题？

测试真的应当注入内疚感？测试真的是摧毁性的？这些问题的回答是“不！”，然而，测试的目标可能和我们所期待的不同。

16.1.1 测试目标

Glen Myers [MYE79] 在他的软件测试著作中陈述了一系列关于测试目标的规则：欢　　馑?

1.测试是一个为了寻找错误而运行程序的过程。

2.一个好的测试用例是指很可能找到迄今为止尚未发现的错误的用例。

3.一个成功的测试是指揭示了迄今为止尚未发现的错误的测试。

上述目标蕴含了一个观点上的戏剧性变化，他们转向通常的观点，即一个成功的测试是指没有找到错误的测试。我们的目标是设计这样的测试，它们能够系统地揭示不同类型的错误，并且耗费最少时间与最小工作量。

如果成功构造了测试(根据上述目标)，则能够在软件中揭示错误。测试的第二个好处在于它证实了软件依据规约所具有的功能及其性能需求，此外，构造测试时的数据收集提供了软件可靠性以及软件整体质量的一些信息。但是，有一件事测试无法完成：

测试无法说明错误不存在，它只能表示软件错误已经出现。

在构造测试时必须牢记这一点。

16.1.2 测试原则

在设计有效的测试用例之前，软件工程师必须理解软件测试的基本原则。Davie [DAV95] 提出了一组[①]测试原则：

• 所有的测试都应追溯到用户需求。正如我们所知，软件测试的目标在于揭示错误。而最严重的错误(从用户角度来看)是那些导致程序无法满足需求的错误。

• 应该在测试工作真正开始的前较长时间内就进行测试计划。测试计划(第17章)可以在需求模型一完成就开始，详细的测试用例定义可以在设计模型被确定后立即开始，因此，所有测试可以在任何代码被产生前进行计划和设计。

• Pareto原则应用于软件测试。简单而言， Pareto原则暗示着测试发现的错误中的80%很可能起源于程序模块中的20%。当然，问题在于如何孤立这些有疑点的模块并进行彻底的测试。

• 测试应从“小规模”开始，逐步转向“大规模”。最初的测试通常把焦点放在单个程序模块上，进一步测试的焦点则转向在集成的模块簇中寻找错误，最后在整个系统中寻找错误(第17章)。

• 穷举测试是不可能的。甚至一个大小适度的程序，其路径排列的数量也非常大(进一步讨论参见第16.2节)，因此，在测试中不可能运行路径的每一种组合，然而，充分覆盖程序逻辑，并确保程序设计中使用的所有条件是有可能的。

• 为了达到最佳效果，应该由独立的第三方来构造测试。“最佳效果”指最可能发现错误的测试(测试的主要目标)。由于本章中已经介绍过、并将在第17章进一步讨论的那些原因，创建系统的软件工程师并不是构造软件测试的最佳人选。

16.1.3 可测试性

在理想的情况下，软件工程师在设计计算机程序、系统或产品时应该考虑可测试性，这就使得负责测试的人能够更容易地设计有效的测试用例，但是，什么是“可测试性”呢？ JamesBach[②]这样描述可测试性：

软件可测试性就是一个计算机程序能够被测试的容易程度。因为测试是如此的困难，因此，需要知道做些什么才能理顺测试过程。有时，程序员愿意去做对测试过程有帮助的事，而一个包括可能的设计点、特性等等的检查表对他们是很有用的。

肯定存在可用于在很多方面测度可测试性的度量，有时，可测试性被用来表示一个特定测试集覆盖产品的充分程度。在军方还用它来表示工具被检验和修复的容易程度。这两种意义都略不同于“软件可测试性”。下面的检查表提供了一组可测试软件的特征：

可操作性。“运行得越好，被测试的效率越高。”

• 系统的错误很少(错误加上测试过程中的分析和报告开销)。• 没有阻碍测试执行的错误。

• 产品在功能阶段的演化(允许同时的开发和测试)。

可观察性。“你所看见的就是你所测试的。”

• 每个输入有唯一的输出。

• 系统状态和变量可见，或在运行中可查询。

• 过去的系统状态和变量可见，或在运行中可查询(例如：事务日志)。• 所有影响输出的因素都可见。

• 容易识别错误输出。

• 通过自测机制自动侦测内部错误。

• 自动报告内部错误。

• 可获取源代码。

可控制性。“对软件的控制越好，测试越能够被自动执行与优化。”• 所有可能的输出都产生于某种输入组合。

• 通过某种输入组合，所有的代码都可能被执行。

• 测试工程师可直接控制软件和硬件的状态及变量。

• 输入和输出格式保持一致且有结构。

• 能够便利地对测试进行说明、自动化和再生。

可分解性。“通过控制测试范围，能够更快地分解问题，执行更灵巧的再测试。”• 软件系统由独立模块构成。

• 能够独立测试各软件模块。

简单性。“需要测试的内容越少，测试的速度越快。”

• 功能简单性(例如：特性集是满足需求所需的最小集合)。

• 结构简单性(例如：将体系结构模块化以限制错误的繁殖)。

• 代码简单性(例如：采用代码标准为检查和维护提供方便)。

稳定性。“改变越少，对测试的破坏越小。”

- 软件的变化是不经常的。
- 软件的变化是可控制的。
- 软件的变化不影响已有的测试。
- 软件失效后能得到良好恢复。

易理解性。“得到的信息越多，进行的测试越灵巧。”

- 设计能够被很好地理解。
- 内部、外部和共享构件之间的依赖性能够被很好地理解。
- 设计的改变被通知。
- 可随时获取技术文档。
- 技术文档组织合理。
- 技术文档明确详细。
- 技术文档精确性稳定。

软件工程师可运用 James Bach 提出的这些属性来开发可测试的软件配置(即程序、数据和文档)。

但是关于测试本身呢？ Kaner， Falk 和 Nguyen [KAN93] 给出了“好”测试的一些属性：文 1.一个好的测试发现错误的可能性很高。为了达到这个目标，测试者必须理解软件，并尝试设想软件如何才能失败，理想，被探测的错误类别，例如，在 GUI(图形用户界面)中有一种潜在的错误，即错误识别鼠标位置。应该设计一个测试集来验证鼠标位置识别的错误。

2.一个好的测试并不冗余。测试的时间和资源是有限的，没有必要构造一个与其他测试用途完全相同的测试，每一个测试都应该有不同的用途(哪怕是细微的差异)。例如，软件SafeHome[①]中有一个模块被用来识别用户密码以决定是否启动系统，为了测试密码输入的错误，测试者设计了一系列的输入密码测试。在不同的测试中输入有效与无效密码(四个数字)，然而，每一个有效/无效密码将检测一种不同错误模式，例如，一个将 8080 作为有效密码的系统将不会接受非法密码 1234，如果接收 1234，将产生错误，另一个测试输入 1235，与 1234 的测试意图相同，因此是冗余的，然而，非法输入 8081 或 8180 就有些细微的差异，即对与有效密码相近但并不相同的密码该进行测试。

3.一个好的测试应该是“最佳品种” [KAN93]。在一组目的相似的测试中，时间和资源的限制可能只影响其某个子集的执行，此时，应该使用最可能找到所有错误的测试。

4.一个好的测试既不会太简单，也不会太复杂。虽然有时会将一组测试组合到一个测试用例中，其副作用可能屏蔽错误，通常，每一个测试应该独立执行。

16.2 测试用例设计

软件和其他工程产品的测试设计与产品本身的设计一样具有挑战性，然而由于已经讨论过的一些原因，软件工程师经常将测试作为一种事后的措施，开发一些“感觉上正确”但是缺乏完整保证的测试用例。再回头看看测试目标，我们必须设计出最可能发现最多数量的错误、并耗费最少时间和最小代价的测试。

在过去的 20 年，出现了大量的测试用例设计方法，为开发人员进行测试提供了系统的方法。更重要的是，方法提供了一种有助于确保完全测试的机制，并提供了揭示软件错误的最高可能性。

能够采用以下两种方法之一对任何工程化产品(以及大多数其他东西)进行测试：(1)若了解产品的特定功能，则构造测试，以证实各功能完全可执行，同时在各功能中寻找错误；(2)若了解产品的内部构造，则构造测试，以确保“所有齿轮吻合”，即内部操作依据规约执行，而且所有的内部构件被充分利用。第一种测试方法被称为黑盒测试，第二种则被称为白盒测试。

如果考虑计算机软件，黑盒测试指在软件界面上进行的测试，虽然设计黑盒测试是为了发现错误，它们却被用来证实软件功能的可操作性；证实能很好地接收输入，并正确地产生输出；以及证实对外部信息完整性(例如：数据文件)的保持。黑盒测试检验系统的一些基本特征，很少涉及软件的内部逻辑结构。

软件的白盒测试依赖对程序细节的严密检验，提供运用特定条件和/与循环集的测试用例，对软件的逻辑路径进行测试，在不同的点检验“程序的状态”以判定预期状态或待验证状态与真实状态是否相符。

一眼看去，可能认为全面的白盒测试将产生“百分之百正确的程序”，需要我们做的只是定义所有的逻辑路径、开发相应的测试用例，并评估结果，简而言之，详尽地生成用例以测试程序逻辑。不幸的是，穷举测试带来了必然的计算问题，即使是很小的程序，可能的逻辑路径数量也非常大，例如，考虑 100 行C语言程序，在一些基本的数据声明之后，程序包含两个嵌套循环，根据输入的条件分别执行 1 到 20 次，在内部循环中，需要四个if-then-else结构，该程序中大约有 10^{14}条可能路径！

为了正确表达这个数值，我们假设开发了一个有魔力的测试处理器(“有魔力”是因为不存在这样的处理器)进行穷举测试。该处理器能在一毫秒内开发一个测试用例、进行运行并评估结果，如果每天运行 24 小时，每年运行 365 天，

则需要 3170 年的时间来测试这个程序。不可否认，这将导致大多数开发进度表的混乱，对大型软件系统不可能进行穷举测试。

然而，白盒测试不应该被抛弃，可选择有限数量的重要逻辑路径进行测试，检测重要数据结构的有效性，可以综合黑盒测试和白盒测试的属性提供一种方法，以验证软件界面，并有选择地保证软件内部工作的正确性。

16.3 白盒测试

白盒测试，有时称为玻璃盒测试，是一种测试用例设计方法，它使用程序设计的控制结构导出测试用例。使用白盒测试方法，软件工程师能够产生测试用例(1)保证一个模块中的所有独立路径至少被使用一次;(2)对所有逻辑值均需测试 true 和 false; (3)在上下边界及可操作范围内运行所有循环; (4)检查内部数据结构以确保其有效性。

此时可能会提出一个合理的问题："我们应该更注重于保证程序需求的实现，为什么要花费时间和精力来担心(和测试)逻辑细节？"换一种说法，我们为什么不将所有精力用于黑盒测试呢？答案在于软件自身的缺陷(例如参考文献［JON81］)：

• 逻辑错误和不正确假设与一条程序路径被运行的可能性成反比。当我们设计和实现主流之外的功能、条件或控制时，错误往往开始出现在我们的工作中。日常处理往往被很好地了解(和很好地细查)，而"特殊情况"的处理则难于发现。

• 我们经常相信某逻辑路径不可能被执行，而事实上，它可能在正常的基础上被执行。程序的逻辑流有时是违反直觉的，这意味着我们关于控制流和数据流的一些无意识的假设可能导致设计错误，只有路径测试才能发现这些错误。

• 印刷上的错误是随机的。当一个程序被翻译为程序设计语言源代码时，有可能产生某些打印错误，很多将被语法检查机制发现，但是，其他的会在测试开始时才会被发现。打印错误出现在主流上和不明显的逻辑路径上的可能性是一样的。

上述任何一条原因都是该进行白盒测试的论据，黑盒测试，不管它多么全面，都可能忽略前面提到的某些类型的错误。正如 Beizer 所说［BEI90］："错误潜伏在角落里，聚集在边界上"。白盒测试更可能发现它们。

16.4 基本路径测试

基本路径测试是 Tom McCabe［MCC76］首先提出的一种白盒测试技术，基本路径测试方法上"。允许测试用例设计者导出一个过程设计的逻辑复杂性测度，

并使用该测度作为指南来定义执行路径的基本集。从该基本集导出的测试用例保证对程序中的每一条语句至少执行一次。

16.4.1 流图符号

在介绍基本路径方法之前，必须先介绍一种简单的控制流表示方法，即流图或程序图[①]。流图使用图 16-1 中的符号描述逻辑控制流，每一种结构化构成元素(第 14 章)有一个相应的流图符号。

为了说明流图的用法，我们采用图 16-2a中的过程设计表示法，此处，流程图用来描述程序控制结构。图 16-2b将流程图映射到一个相应的流图(假设流程图的菱形决定框中不包含复合条件)。在图 16-2b中，每一个圆，称为流图的节点，代表一个或多个语句。一个处理方框序列和一个菱形决测框可被映射为一个节点，流图中的箭头，称为边或连接，代表控制流，类似于流程图中的箭头。一条边必须终止于一个节点，即使该节点并不代表任何语句(例如：参见 if-else-then结构的符号)。由边和节点限定的范围称为区域。计算区域时应包括图外部的范围[①]。

任何过程设计表示法都可被翻译成流图，图 16-3 显示了一个程序设计语言(PDL，ProgramDesign Language)片段及其对应的流图，注意，对 PDL 语句进行了编号，并将相应的编号用于流图中。

程序设计中遇到复合条件时，生成的流图变得更为复杂。当条件语句中用到一个或多个布尔运算符(逻辑 OR，AND，NAND，NOR)时，就出现了复合条件。图 16-4 中，将一个 PDL 片段翻译为流图，注意，为语句 IF a OR b 中的每一个 a 和 b 创建了一个独立的节点，包含条件的节点被称为判定节点，从每一个判定节点发出两条或多条边

16.4.2 环形复杂性

环形复杂性是一种为程序逻辑复杂性提供定量测度的软件度量，将该度量用于基本路径方法，计算所得的值定义了程序基本集的独立路径数量，并为我们提供了确保所有语句至少执行一次的测试数量的上界。

独立路径是指程序中至少引进一个新的处理语句集合或一个新条件的任一路径。采用流图的术语，即独立路径必须至少包含一条在定义路径之前不曾用到的边。例如，图 16-2b 中所示流图的一个独立路径集合为：

路径 1：1-11

路径 2：1-2-3-4-5-10-1-11

路径 3：1-2-3-6-8-9-10-1-11

路径 4：1-2-3-6-7-9-10-1-11

注意，每一条新的路径都包含了一条新边。路径 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 不是独立路径，意味它只是已有路径的简单合并，并未包含任何新边。

上面定义的路径 1，2，3 和 4 包含了图 16-2b 所示流图的一个基本集，简而言之，如果能将测试设计为强迫运行这些路径(基本集)，那么程序中的每一条语句将至少被执行一次，每一个条件执行时都将分别取 true 和 false。应该注意到基本集并不唯一，实际上，给定的过程设计可派生出任意数量的不同基本集。

如何才能知道需要寻找多少条路径呢？对环形复杂性的计算提供了这个问题的答案。环形复杂性以图论为基础，为我们提供了非常有用的软件度量。可用如下三种方法之一来计算复杂性：

1.流图中区域的数量对应于环形的复杂性。

2.给定流图 G 的环形复杂性——V(G)，定义为 V(G)=E-N+2，E 是流图中边的数量，N 是流图节点数量。

3.给定流图 G 的环形复杂性——V(G)，也可定义为 V(G)=P+1，P 是流图 G 中判定节点的数量。

再回到图 16-2b。可采用上述任意一种算法来计算环形复杂性。

1.流图有 4 个区域。

2.V(G)=11 条边-9 个节点+2=4。

3.V(G)=3 个判定节点+1=4。

因此，图 16-2b 的环形复杂性是 4。

更重要的是，V(G)的值提供了组成基本集的独立路径的上界，并由此得出覆盖所有程序语句所需的测试设计数量的上界。

16.4.3 导出测试用例

基本路径测试方法可用于过程设计或源代码生产。本节中，我们将基本路径测试表示为一系列步骤，图 16-5 中 PDL 所描述的过程“求平均值”将被用于阐明测试用例设计方法中的各个步骤。注意，“求平均值”虽然是一个非常简单的算法，但是仍然包含了复合条件和循环。

1.以设计或代码为基础，画出相应的流图。使用符号和第 16.4.1 节中的构造规则创建一个流图，参考图 16-5 中“求平均值”的 PDL。创建流图时，要对将被映射为流图节点的 PDL 语句进行标号，图 16-6 显示了对应的流图。

2.确定结果流图的环形复杂性。可采用上一节中的任意一种算法来计算环形复杂性——V(G)。应该注意到，计算 V(G)并不一定要画出流图，计算 PDL 中的所有条件语句数量(过程求平均值中复合条件语句计数为 2)，然后加 1 即可得到环形复杂性。在图 16-6 中，

V(G)=6 个区域

V(G)=18 条边-14 个节点+2=6。

V(G)=5 个判定节点+1=6。

3.确定线性独立的路径的一个基本集。V(G)的值提供了程序控制结构中线性独立的路径的数量，在过程求平均值中，我们指定六条路径：

路径 1：1-2-10-11-13

路径 2：1-2-10-12-13

路径 3：1-2-3-10-11-13

路径 4：1-2-3-4-5-8-9-2-……

路径 5：1-2-3-4-5-6-8-9-2-……

路径 6：1-2-3-4-5-6-7-8-9-2-……

路径 4、5 和 6 后面的省略号(……)表示可加上控制结构其余部分的任意路径。通常在导出测试用例时，识别判定节点是很有必要的。本例中，节点 2、3、5、6 和 10 是判定节点。

4.准备测试用例，强制执行基本集中每条路径。测试人员可选择数据以便在测试每条路径时适当设置判定节点的条件。满足上述基本集的测试用例如下：

路径 1 测试用例：

value(k)=有效输入，其中 k＜i

value(i)=-999，其中 2≤i≤100

期望结果：基于 k 的正确平均值和总数

注意：路径 1 无法独立测试，必须作为路径 4、5 和 6 测试的一部分。

路径 2 测试用例：

value(1)=-999

期望结果：求平均值=-999；其他按初值汇总

路径 3 测试用例：

试图处理 101 或更大的值

前 100 个数值应该有效

期望结果：与测试用例 1 相同

路径 4 测试用例：

value(i)=有效输入，其中 i＜100

value(k)＜最小值，其中 k＜i

期望结果：基于 k 的正确平均值和总数

路径 5 测试用例：

value(i)=有效输入，其中 i＜100

value(k)＞最大值，其中 k≥i

期望结果：基于 k 的正确平均值和总数

路径 6 测试用例：

value(i)=有效输入，其中 i＜100

期望结果：基于 k 的正确平均值和总数

执行每个测试用例，并和期望值比较，一旦完成所有测试用例，测试者可以确定在程序中的所有语句至少被执行一次。

重要的是要注意，某些独立路径(如，例子中的路径 1)不能以独立的方式被测试，即，穿越路径所需的数据组合不能形成程序的正常流，在这种情况下，这些路径必须作为另一个路径测试的一部分来进行测试。

16.4.4 图矩阵

导出流图和决定基本测试路径的过程均需要机械化，为了开发辅助基本路径测试的软件工具，称为图矩阵(graph matrix)的数据结构很有用。

图矩阵是一个正方形矩阵，其大小(即列数和行数)等于流图的节点数。每列和每行都对应于标识的节点，矩阵项对应于节点间的连接(边)，图 16-7 显示了一个简单的流图及其对应的图矩阵［BEI90］。

该图中，流图的节点以数字标识，边以字母标识，矩阵中的字母项对应于节点间的连接，例如，边 b 连接节点 3 和节点 4。

这里，图矩阵只是流图的表格表示，然而，对每个矩阵项加入连接权值(link weight)，图矩阵就可以用于在测试中评估程序的控制结构，连接权值为控制流提供了另外的信息。最简单情况下，连接权值是 1(存在连接)或 0(不存在连接)，但是，连接权值可以赋予更有趣的属性：

- 执行连接(边)的概率。
- 穿越连接的处理时间。
- 穿越连接时所需的内存。
- 穿越连接时所需的资源。

举例来说，我们用最简单的权值(0 或 1)来标识连接，图 16-7 所示的图矩阵重画为图 16-8。字母替换为 1，表示存在边(为清晰起见，没有画出 0)，这种形式的图矩阵称为连接矩阵(linkmatrix)。图 16-8 中，含两个或两个以上项的行表示判定节点，所以，右边所示的算术计算就提供了另一种环形复杂性计算(参 16.4.2 节)的方法。

Beizer［BEI90］提供了可用于图矩阵的其他数学算法的处理，利用这些技术，设计测试用例的分析就可以自动化或部分自动化。

16.5 控制结构测试

16.4 节所述的基本路径测试技术是控制结构测试技术之一。尽管基本路径测试简单高效，但是，其本身并不充分。本节讨论控制结构测试的其他变种，这些测试覆盖并提高了白盒测试的质量。

16.5.1 条件测试

条件测试是检查程序模块中所包含逻辑条件的测试用例设计方法。一个简单条件是一个布尔变量或一个可能带有 NOT(“¬”)操作符的关系表达式。关系表达式的形式如：

E_1＜关系操作符＞E_2

其中E_1和E_2是算术表达式，而＜关系操作符＞是下列之一：“＜”，“≤”，“=”，“≠”（“┐=”），“＞”，或“≥”。复杂条件由简单条件、布尔操作符和括弧组成。我们假定可用于复杂条件的布尔算子包括OR“|”，AND“&”和NOT“┐”，不含关系表达式的条件称为布尔表达式。

所以条件的成分类型包括布尔操作符、布尔变量、布尔括弧(括住简单或复杂条件)、关系操作符或算术表达式。

如果条件不正确，则至少有一个条件成分不正确，这样，条件的错误类型如下：

- 布尔操作符错误(遗漏布尔操作符，布尔操作符多余或布尔操作符不正确)。
- 布尔变量错误。
- 布尔括弧错误。
- 关系操作符错误。
- 算术表达式错误。

条件测试方法注重于测试程序中的条件。本节后面讨论的条件测试策略主要有两个优点，首先，测度条件测试的覆盖率是简单的，其次，程序的条件测试覆盖率为产生另外的程序测试提供了指导。

条件测试的目的是测试程序条件的错误和程序的其他错误。如果程序 P 的测试集能够有效地检测 P 中的条件错误，则该测试集可能也会有效地检测 P 中的其他错误，此外，如果测试策略对检测条件错误有效，则它也可能有效地检测程序错误。

已经提出了几个条件测试策略。分支测试可能是最简单的条件测试策略，对于复合条件 C，C 的真分支和假分支以及 C 中的每个简单条件都需要至少执行一次［MYE97］。

域测试(Domain testing)［WHI80］要求从有理表达式中导出三个或四个测试，有理表达式的形式如：

E_1＜关系操作符＞E_2

需要三个测试分别用于计算E_1的值是大于、等于或小于E_2的值［HOW82］。如果＜关系操作符＞错误，而E_1和E_2正确，则这三个测试能够发现关系算子的错误。为了发现E_1和E_2的错误，计算E_1小于或大于E_2的测试应使两个值间的差别尽可能小。

有n个变量的布尔表达式需要2^n个可能的测试(n＞0)。这种策略可以发现布尔操作符、变量和括弧的错误，但是只有在n很小时实用。

也可以派生出敏感布尔表达式错误的测试［FOS84，TAI87］。对于有n个布尔变量(n＞0)的单布尔表达式(每个布尔变量只出现一次)，可以很容易地产生测试数小于2^n的测试集，该测试集能够发现多个布尔操作符错误和其他错误。

Tai［TAI89］建议在上述技术之上建立条件测试策略，称为BRO(branch and relational 试集 operator)的测试保证能发现布尔变量和关系操作符只出现一次而且没有公共变量的条件中的分支和条件操作符错误。

BRO策略利用条件C的条件约束。有n个简单条件的条件C的条件约束定义为$(D_1, D_2, \cdots, D_n)$，其中D_i(0＜i≤n)表示条件C中第i个简单条件的输出约束。如果C的执行过程中C的每个简单条件的输出都满足D中对应的约束，则称条件C的条件约束D由C的执行所覆盖。

对于布尔变量B，B输出的约束说明B必须是真(t)或假(f)。类似地，对于关系表达式，符号＜、=、＞用于指定表达式输出的约束。

作为简单的例子，考虑条件

$$C_1：B_1 \& B_2$$

其中B_1和B_2是布尔变量。C_1的条件约束式如(D_1, D_2)，其中D_1和D_2是“t”或“f”，值(t，f)是C_1的条件约束，由使B_1为真B_2为假的测试所覆盖。BRO测试策略要求约束集｛(t，t)，(f，t)，(t，f)｝由C_1的执行所覆盖，如果C_1由于布尔算子的错误而不正确，至少有一个约束强制C_1失败。

作为第二个例子，考虑

$$C_2：B_1 \& (E_3=E_4)$$

其中B_1是布尔表达式，而E_3和E_4是算术表达式。C_2的条件约束形式如(D_1, D_2)，其中D_1是“t”或“f”，D_2是＜，=或＞。除了C_2的第二个简单条件是关系表达式以外，C_2和C_1相同，所以可以修改C_1的约束集｛(t，t)，(f，t)，(t，f)｝，得到C_2的约束集,注意$(E_3=E_4)$的“t”意味着“=”,而$(E_3=E_4)$的“f”意味着“＞”或“＜”。分别用(t，=)和(f，=)替换(t，t)和(f，t)，并用(t，＜＼＜＞)和(t，＞)替换(t，f)，就得到C_2的约束集｛(t，=)，(f，=)，(t，＜)，(t，＞)｝。上述条件约束集的覆盖率将保证检测C_2的布尔和关系算子的错误。

作为第三个例子，考虑

C_3：$(E_1＞E_2) \& (E_3=E_4)$

其中E_1、E_2、E_3和E_4是算术表达式。C_3的条件约束形式如(D_1，D_2)，其中D_1和D_2是＜、=或＞。除了C_3的第一个简单条件是关系表达式以外，C_3和C_2相同，所以可以修改C_2的约束集得到C_3的约束集，结果为

{(＞，=)，(=，=)，(＜，=)，(＞，＞)，(＞，＜)}

上述条件约束集能够保证检测C_3的关系操作符的错误。

16.5.2 数据流测试

数据流测试方法按照程序中的变量定义和使用的位置来选择程序的测试路径。已经有不少关于数据流测试策略的研究(如参考文献［FRA88］、[NTA88]和[FRA93])。肪丁Ｒ丫　胁簧*

为了说明数据流测试方法，假设程序的每条语句都赋予了独特的语句号，而且每个函数都不改变其参数和全局变量。对于语句号为 S 的语句，

DEF(S)={X｜语句 S 包含 X 的定义}

USE(S)={X｜语句 S 包含 X 的使用}

如果语句 S 是 if 或循环语句，它的 DEF 集为空，而 USE 集取决于 S 的条件。如果存在从 S 到 S′的路径，并且该路径不含 X 的其他定义，则称变量 X 在语句 S 处的定义在语句 S′仍有效。

变量 X 的定义—使用链(或称 DU 链)形式如［X，S，S′］，其中 S 和 S′是语句号，X 在 DEF(S)和 USE(S′)中，而且语句 S 定义的 X 在语句 S′有效。

一种简单的数据流测试策略是要求覆盖每个 DU 链至少一次。我们将这种策略称为 DU 测试策略。已经证明 DU 测试并不能保证覆盖程序的所有分支，但是，DU 测试不覆盖某个分支仅仅在于如下之类的情况：if-then-else 中的 then 没有定义变量，而且不存在 else 部分。这种情况下，if 语句的 else 分支并不需要由 DU 测试覆盖。

数据流测试策略可用于为包含嵌套 if 和循环语句的程序选择测试路径，为此，考虑使用 DU 测试为如下的 PDL 选择测试路径：

```
proc x

B1;

do while C1

if C2
```

```
then
if C4
then B4;
else B5;
endif;
else
if C3
then B2;
else B3;
endif;
endif;
enddo;
B6;
end proc;
```

为了用DU测试选择控制流图的测试路径，需要知道PDL条件或块中的变量定义和使用。假设变量X定义在块B1，B2，B3，B4 和B5 的最后一条语句之中，并在块B2，B3，B4，B5 和B6 的第一条语句中使用。DU测试策略要求执行从每个B(0＜i≤5)到B_j(0＜j≤6)的最短路径(这样的测试也覆盖了条件C_1，C_2，C_3和C_4中的变量使用)。尽管有 25 条X的DU链，只需 5 条路径覆盖这些DU链。原因在于可用 5 条从B_i(0＜i≤5)到B6 的路径覆盖X的链，而这 5 条链包含循环的迭代就可以覆盖其他的DU链。

注意如果要用分支测试策略为上述的 PDL 选择测试路径，并不需要另外的信息。为了选择 BRO 测试的路径，只需知道每个条件和块的结构。(选择程序的路径之后，需要决定该路径是否实用于该程序，即是否存在执行该路径的至少一个输入)。

由于变量的定义和使用，程序中的语句都彼此相关，所以数据流测试方法能够有效地发现错误，但是，数据流测试的覆盖率测度和路径选择比条件测试更为困难。

16.5.3 循环测试

循环是大多数软件实现算法的重要部分，但是，在软件测试时却很少注意它们。

循环测试是一种白盒测试技术，注重于循环构造的有效性。有四种循环[BEI90]：简单循环，串接循环，嵌套循环和不规则循环(如图 16-9 所示)。

简单循环。下列测试集应当用于简单循环，其中 n 是允许通过循环的最大次数。

1.整个跳过循环。

2.只有一次通过循环。

3.两次通过循环。

4.m 次通过循环，其中 m＜n。

5.n-1，n，n+1 次通过循环。

嵌套循环。如果要将简单循环的测试方法用于嵌套循环，可能的测试数就会随嵌套层数成几何级增加，这会导致不实际的测试数目，Beizer [BEI90] 提出了一种减少测试数的方法：*

1.从最内层循环开始，将其他循环设置为最小值。

2.对最内层循环使用简单循环测试，而使外层循环的迭代参数(即循环计数)最小，并为范围外或排除的值增加其他测试。

3.由内向外构造下一个循环的测试，但其他的外层循环为最小值，并使其他的嵌套循环为“典型”值。

4.继续直到测试完所有的循环。

串接循环。如果串接循环的循环都彼此独立，可以使用嵌套循环的策略测试串接循环。但是，如果两个循环串接起来，而第一个循环的循环计数是第二个循环的初始值，则这两个循环并不是独立的。如果循环不独立，则推荐使用嵌套循环的方法进行测试。

不规则循环。尽可能的情况下，要将这类循环重新设计为结构化的程序结构(参第 14 章)。

16.6 黑盒测试

黑盒测试注重于测试软件的功能性需求，也即黑盒测试使软件工程师派生出执行程序所有功能需求的输入条件。黑盒测试并不是白盒测试的替代品，而是用于辅助白盒测试发现其他类型的错误。

黑盒测试试图发现以下类型的错误：(1)功能不对或遗漏，(2)界面错误，(3)数据结构或外部数据库访问错误，(4)性能错误和(5)初始化和终止错误。

白盒测试在测试的早期执行，而黑盒测试主要用于测试的后期(参第 17 章)。黑盒测试故意不考虑控制结构，而是注意信息域。测试用于回答以下问题：

- 如何测试功能的有效性？
- 何种类型的输入会产生好的测试用例？
- 系统是否对特定的输入值尤其敏感？
- 如何分隔数据类的边界？
- 系统能够承受何种数据率和数据量？
- 特定类型的数据组合会对系统产生何种影响？

运用黑盒测试，可以导出满足以下标准的测试用例集［MYE79］：(1)所设计的测试用例能够减少达到合理测试所需的附加测试用例数，和(2)所设计的测试用例能够告知某些类型错误的存在或不存在，而不是仅仅与特定测试相关的错误。

16.6.1 基于图的测试方法

黑盒测试[①]的第一步是理解软件所表示的对象[②]及其关系，然后，第二步是定义一组保证“所有对象与其他对象都具有所期望的关系”［BEI95］的测试序列，换而言之，软件测试首先是创建对象及其关系图，然后导出测试序列以检查对象及其关系，并发现错误。

为了完成这些步骤，软件工程师首先创建一个图，节点代表对象，连接代表对象间的关系，节点权值描述节点的属性(如特定的数据值或状态行为)，连接权值描述连接的特点[①]。

图的符号表示如图 16-10a 所示。节点表示为圆，而连接有几种，有向连接(有箭头表示)表明关系只在一个方向上存在。双向连接，也称为对称连接，表示关系适于两个方向。如果节点间有几种联系，就使用并行边。

举一个简单的例子，考虑部分字处理应用程序图，如图 16-10b 所示：

对象＃1=新建文件菜单选择

对象＃2=文档窗口

对象＃3=文档文本

如图所示，选择菜单“新建文件”产生一个文档窗口，文档窗口的节点权值提供窗口产生时所期望的属性集，连接权值表明窗口必须在 1.0 秒之内产生，一条无向边在“选择菜单新建文件”和“文档文本”之间建立对称联系，并行连接表明“文档窗口”和“文档文本”间的联系，事实上，要产生测试用例还需要更加详细的图。软件工程师遍历图，并覆盖所显示的联系就可以导出测试用例，这些用例用于发现联系之间的错误。

Beizer [BEI95] 描述了几个使用图的行为测试方法：

事务流建模。节点代表事务的步数(如使用联机服务预订航空机票的步数)，连接代表步骤之间的连接关系(如 flighgt.information.input 后跟 validation/availabilty.processing)。数据流图(参见第 12 章)可用于辅助产生这种图。

有限状态建模。节点代表不同用户可见的软件状态(如订票人员处理订票时的各个屏幕)，而连接代表状态之间的转换(如 order-information 在 inventory-availability-look-up 时验证并后跟 customer-billing-information-input)。状态变迁图(参见第 12 章)可用于辅助产生这种图。

数据流建模。节点是数据对象，而连接是将数据对象转换为其他对象时发生的变换。例如，节点 FICA.tzx.withheld(FTW)由 gross.wagess(GW)利用关系 FTW=0.062×GW 计算而来。

时间建模。节点是程序对象，而连接是对象间的顺序连接。连接权值用于指定程序执行时所需的执行时间。

基于图的测试方法的详细讨论超出了本书的范围，感兴趣的读者可以阅读(见参考文献 [BEI95])一书。但是，大致了解一下基于图的测试还是值得的。

基于图的测试开始定义节点和节点权值，也即标识对象及其属性，数据模型(参见第 12 章)可以作为起始点，但是要注意很多节点是程序对象(不在数据模型中时显表示出来)，为了标识图的起点和终点，可以定义入点和出点。

标识节点以后，就可以建立连接及其权值，连接一般应当命名，但是当代表程序对象间控制流的连接时除外。

很多情况下，图模型可能有循环(如图的路径含有环)，循环测试(参见 16.5.3 节)也可用于行为(黑盒)测试，图可用于标识需要测试的循环。

分别研究每个关系，以导出测试用例。研究顺序关系的传递性可以发现关系在对象间传播的影响。举例说明，有三个对象 X，Y 和 Z。考虑如下关系：

计算 Y 需要 X

计算 Z 需要 Y

所以，X 和 Z 之间有传递性：

计算 Z 需要 X

基于这种传递性，测试 Z 的计算时要考虑 X 和 Y 的各种值。

关系(图连接)的对称性也是设计测试用例的重要考虑，如果关系是双向(对称)的，就要测试这种性质。很多应用程序的 UNDO 功能［BEI95］实现了有限的对称性，应该彻底测试并标识鸵　馑　械囊斐＃m床荒苁褂肮 NDO 的地方)。最后，图的每个节点都应当有到自己的关系，本质上是“空操作”循环，自反性也应当进行测试。

开始设计测试用例时，第一个目标是节点的覆盖度，这意味着测试不应当遗漏某个节点，而且节点的权值是正确的。

接着，考虑连接的覆盖率，基于属性测试每个关系，例如，测试对称关系以表明它的确是双向的，测试传递关系以表明存在传递性，测试自反关系以表明存在空操作。指明连接权值时，要设计测试以展示权值是否有效，最后，加入循环测试(参见 16.5.3 节)。

16.6.2 等价划分

等价划分是一种黑盒测试方法，将程序的输入域划分为数据类，以便导出测试用例。理想的测试用例是独自发现一类错误(如字符数据的处理不正确)。等价划分试图定义一个测试用例以发现各类错误，从而减少必须开发的测试用例数。

等价划分的测试用例设计基于输入条件的等价类评估。使用前面章节介绍的概念，如果对象由具有对称性、传递性或自反性的关系连接，就存在等价类［BEI95］。等价类表示输入条件的一组有效或无效的状态。典型地，输入条件通常是一个特定的数值，一个数值域，一组相关值或一个布尔条件。可按照如下指南定义等价类：

1.如果输入条件代表一个范围，可以定义一个有效等价类和两个无效等价类。

2.如果输入条件需要特定的值，可以定义一个有效等价类和两个无效等价类。

3.如果输入条件代表集合的某个元素，可以定义一个有效等价类和一个无效等价类。

4.如果输入条件是布尔式，可以定义一个有效等价类和一个无效等价类。

作为例子，考虑自动银行应用软件所维护的数据，用户可以用自己的微机拨号到银行，提供六位数的密码，并遵循一序列键盘命令以触发各种银行功能。银行应用程序的软件可以接受如下格式的数据：

区号——空或三位数字。

前缀——三位数字，但不是 0 和 1 开始。

后缀——四位数字。

密码——六位字母或数字。

命令——“检查”，“存款”，“付款”等。

与银行应用程序各种数据元素相关的输入条件可以表示为：

区号：输入条件，布尔——区号存在与否。

输入条件，范围——定义在 200 和 999 之间的数值，少数例外。

前缀：输入条件，范围——大于 200 不含 0 的数值。

后缀：输入条件，值——四位数字。

密码：输入条件，布尔——密码存在与否。

输入条件，值——六位字符串。

命令：输入条件，集合——包含上述命令。

利用上述导出等价类的指南，就可以为每个输入域的数据项开发并执行测试用例，测试用例的选择最好是每次执行最多的等价类属性。

16.6.3 边界值分析

由于某些未被完全知道的原因，输入域的边界比中间更加容易发生错误，为此，可用的边界值分析(boundary value analysis，BVA)可作为一种测试技术。边界值分析选择一组测试用例检查边界值。

边界值分析是一种补充等价划分的测试用例设计技术。BVA 不是选择等价类的任意元素，而是选择等价类边界的测试用例，BVA 不仅注重于输入条件，而且也从输出域导出测试用例［MYE79］。

BVA 的指南类似于等价划分：

1.如果输入条件代表以 a 和 b 为边界的范围，测试用例应当包含 a、b、略大于 a 和略小于 b 的值。

2.如果输入条件代表一组值，测试用例应当执行其中的最大值和最小值，还应当测试略大于最小值的值和略小于最大值的值。

3.指南 1 和 2 也适用于输出条件，例如，工程分析程序要求输出温度和压强的对照表，测试用例应当能够创建包含最大值和最小值的项。

4.如果程序数据结构有预定义的边界(如数组有 100 项)，要测试其边界的数据项。

大多数软件工程师会在某种程度上自发地执行 BVA，利用上述指南，边界测试会更加完整，从而更可能发现错误。

16.6.4 比较测试

有些情况下(如航空电子设备、核电厂控制)，软件的可靠性绝对重要，此时，需要冗余的硬件和软件，以减小错误的可能性。开发冗余软件时，另外的软件工程师队伍利用相同的需求开发应用程序的另外版本，这样，可用相同的测试数据引进测试以产生相同的输出，接着，并行执行所有版本并进行实时结果比较以保证一致性。

利用从冗余系统中学到的经验，研究者(如参考文献［BRI87］)建议对关键应用程序开发不同的软件版本，即便最后只使用一个版本，不同的版本成了称为比较测试或背靠背测试参考文献［KNI89］的黑盒测试技术的基础。

同样的需求有不同的实现时，利用其他黑盒技术(如等价分割)设计的测试用例可以作为另一个版本的输入，如果每个版本的输出相同，就可以假定所有的实现都正确，如果输出不同，就要调查各个版本，以发现错误所在。大多数情况下，可用自动化工具进行输出比较。

比较测试并不能够保证无错，如果需求本身有错，所有的版本都可能反映错误，另外，如果各个版本产生相同但却错误的结构，条件测试也无法发现错误。

16.7 针对专门环境和应用的测试

随着计算机软件变得更为复杂，对特殊测试方法的需求也增加了。16.5 和 16.6 两节所讨论的白盒和黑盒测试方法可以用于所有的环境、体系结构和应用程序，但是有时还是需要专门的指南和方法。本节讨论用于软件工程师常见的特定环境、体系结构和应用程序的测试指南。

16.7.1 GUI 测试

图形用户界面(GUI)对软件工程师提出了有趣的挑战，因为 GUI 开发环境有可复用的构件，开发用户界面更加省时而且更加精确，同时，GUI 的复杂性也增加了，从而增加了设计和执行测试用例的难度。

因为现代 GUI 有相同的观感，已经有一序列标准的测试。下列问题可以作为常见 GUI 测试的指南：

对于窗口：

- 窗口能否基于相关的输入或菜单命令适当地打开？
- 窗口能否改变大小、移动和滚动？
- 窗口中的数据内容能否用鼠标、功能键、方向箭头和键盘访问？
- 当被覆盖并重调用后，窗口能否正确地再生？
- 需要时能否使用所有窗口相关的功能？
- 所有窗口相关的功能是可操作的吗？
- 是否有相关的下拉式菜单、工具条、滚动条、对话框、按钮、图标和其他控制可为窗口可用，并适当地显示？
- 显示多个窗口时，窗口的名称是否被适当地表示？
- 活动窗口是否被适当地加亮？
- 如果使用多任务，是否所有的窗口被实时更新？
- 多次或不正确按鼠标是否会导致无法预料的副作用？
- 窗口的声音和颜色提示和窗口的操作顺序是否符合需求？
- 窗口是否正确地关闭？

对于下拉式菜单和鼠标操作：

- 菜单条是否显示在合适的语境中？
- 应用程序的菜单条是否显示系统相关的特性(如时钟显示)？
- 下拉式操作能正确工作吗？
- 菜单、调色板和工具条是否工作正确？
- 是否适当地列出了所有的菜单功能和下拉式子功能？
- 是否可以通过鼠标访问所有的菜单功能？
- 文本字体、大小和格式是否正确？
- 是否能够用其他的文本命令激活每个菜单功能？
- 菜单功能是否随当前的窗口操作加亮或变灰？
- 菜单功能是否正确执行？
- 菜单功能的名字是否具有自解释性？
- 菜单项是否有帮助，是否语境相关？
- 在整个交互式语境中，是否可以识别鼠标操作？
- 如果要求多次点击鼠标，是否能够在语境中正确识别？
- 如果鼠标有多个按钮，是否能够在语境中正确识别？
- 光标、处理指示器和识别指针是否随操作恰当地改变？

对于数据项：

- 字母数字数据项是否能够正确回显，并输入到系统中？
- 图形模式的数据项(如滑动条)是否正常工作？
- 是否能够识别非法数据？
- 数据输入消息是否可理解？

除了上述智能以外，有限状态建模图(参见 16.6.1 节)也可以用于导出 GUI 相关的数据和程序对象的测试集。

因为 GUI 操作相关的排列数很大，所以应当用自动化工具进行测试。近几年来市场上已有不少的 GUI 测试工具，关于详细情况，请参见第 29 章。

16.7.2 客户/服务器体系结构的测试

客户/服务器体系(C/S)结构(如参考文献[BER92]和[VAS93])对软件测试人员提出了很大挑战。客户/服务器的分布式特性、事务处理相关的性能、不同硬件平台存在的可能性、网络通讯的复杂性、由中心(或分布式)数据库为多个客户服务和服务器的协调需求一起使得 C/S 体系结构及其软件的测试更为困难。事实上，最近的工业研究表明开发 C/S 环境时显著增加了测试时间和成本。有关客户/服务器测试的详细信息，请参见第 28 章。

16.7.3 测试文档和帮助设施

术语“软件测试”造成一种假象，即测试用例是为程序及其操纵的数据准备的。回忆一下本书第一章所讲的软件定义，要注意到测试必须扩展到软件的第三个元素——文档[①]。

文档错误会同数据和代码错误一样给程序带来灾难性后果。没有什么比精确地按照用户指南，但得到的结果却不符合文档的描述更令人气恼的了。为此，文档测试也是每个软件测试中有意义的一部分。

文档测试可以分为两个步骤。第一步为正式的技术复审(见第 8 章)，检查文档的编辑错误；第二步是活性测试(live testing)，结合实际程序的使用而使用文档。

活性测试可使用类似于 16.6 节所述的黑盒测试方法，基于图的测试可用于描述程序的使用。等价划分和边界值分析可以定义输入类型及其相关的交互，然后就可以按照文档跟踪程序的使用：

- 文档是否精确描述了如何使用各种使用模式？
- 交互顺序的描述是否精确？
- 例子是否精确？
- 术语、菜单描述和系统响应是否与实际程序一致？
- 是否能够很方便地在文档中定位指南？
- 是否能够很方便地使用文档排除错误？
- 文档的内容和索引是否精确完整？

• 文档的设计(布局、缩进和图形)是否便于信息的理解？

• 显示给用户的错误信息是否有更详细的文档解释？

• 如果使用超级链接，超级链接是否精确完整？

回答这些问题最可行的方法是让第三个组(如选择用户)按照程序的使用测试文档。标出所有错误和模糊的地方，以便重写。

16.7.4 实时系统测试

很多实时系统(第 15 章)的时间依赖性和异步性给测试带来新的困难——时间。测试用例的设计者考虑的不仅是白盒和黑盒测试用例，而且包括事件处理(如中断处理)，数据的时间安排以及处理数据的任务(进程)的并发性。很多情况下，提供的测试数据有时使得实时系统在某状态下可以正常运行，而同样的数据在系统处于不同状态时有时又会导致错误。

例如，控制复印机的实时软件在机器复印时接收操作员的中断(如操作员按某些键如“reset”或“darken”)不会产生错误，但是如果在夹纸时，按同样的键就会产生一个诊断代码，指明夹纸的位置信息将被丢失。

另外，实时系统的软件和硬件之间的密切关系也会导致测试问题，软件测试必须考虑硬件故障对软件处理的影响，这种故障很难实时仿真。

实时系统的综合性测试用例设计方法还有待进一步发展，但是，仍然已有了大致的四步策略：

任务测试 测试实时系统的第一步是独立地测试各个任务。也即对每个任务设计白盒和黑盒测试用例，并在测试时执行每个任务。任务测试能够发现逻辑和功能错误，但是不能发现时间和行为错误。

行为测试 利用 CASE 工具(参见 15.3 节)创建的软件模型，就可能仿真实时系统，并按照外部事件的序列检查其行为，这些分析活动可作为创建实时系统时设计测试用例的基础。使用类似于等价划分的技术(参见 16.6.2 节)，可以对事件(如中断、控制信号和数据)分类测试，例如，复印机的事件可能是用户中断(如重置计数)、机器中断(如卡纸)、系统中断(如缺粉)和故障模式(如过热)。每种事件都可以独立测试，并且检查可执行系统的行为以检测是否有与事件处理相关的继发性错误。对系统模型(在分析活动时开发)的行为和可执行软件进行符合性比较，看是否一致。测试每种事件以后，以随机顺序和随机频率将事件传给系统，检查系统行为看是否有行为错误。

任务间测试 在隔离了任务内部和系统行为错误以后，测试就要转向时间相关的错误。用不同的数据率和处理负载来测试与其他任务通讯的异步任务，看任

务间的同步是否会产生错误。另外，测试通过消息队列和数据存储进行通讯的任务，以发现这些数据存储区域大小方面的错误。

系统测试 集成软件和硬件，并进行大范围的系统测试(参见第 17 章)，以发现软件/硬件接口间的错误。

很多实时系统处理中断，所以，测试布尔事件的处理尤其重要。利用状态变迁图和控制规约(参见第 12 章)，测试者可开发一系列可能的中断及其将发生的处理，设计测试用例以验证如下的系统特性：

- 是否能够正确赋予和处理中断的优先权？
- 每个中断的处理是否正确？
- 中断处理的性能(如处理时间)是否符合需求？
- 关键时刻有大量中断时，是否会导致功能和性能上的问题？

另外，也测试应当作为中断处理一部分的传输信息的全局数据区域，以评估潜在的副作用。

16.8 小结

测试用例设计的主要目标是导出有可能发现软件错误的测试集，为此，有两种不同的测试用例设计技术：白盒测试和黑盒测试。

白盒测试注重于程序控制结构。测试用例要保证测试时程序的所有语句至少执行一次，而且检查了所有的逻辑条件。基本路径测试利用程序图(或图矩阵)导出保证覆盖率的线性无关的测试集。条件和数据流测试进一步检测程序逻辑，而循环测试补充其他的白盒测试技术，来对不同复杂度的循环进行测试。

Hetel [HET84] 将白盒测试描述为“小规模测试”，意味着本章所讨论的白盒测试一般用于小的程序构件的测试(如模块和小模块组)。黑盒测试扩大了测试焦点，因而称为“大规模测试”。

黑盒测试发现功能需求错误，而不考虑程序的内部工作。黑盒测试技术注重于软件的信息域，划分程序的输入域和输出域而导出测试用例。基于图的测试方法检查程序对象的行为及其之间的联系。等价划分将输入域划分成数据类，每类执行特定的软件功能。边界值分析则检查程序处理边界数据的能力。

特定的测试方法包括广泛的软件功能和应用区域，图形用户界面，客户/服务器结构，文档和帮助功能以及实时系统都需要专门的测试指南和技术。

有经验的软件开发者经常说：“测试永不终止，只是从软件工程师转移到客户。客户每次使用程序时，都是一次测试。”通过设计测试用例，软件工程师可以进行更广泛的测试，从而在“客户测试”之前发现并修改尽可能多的错误。

思考题

16.1 Myers [MYE79] 用以下的程序作为对测试能力的自我评估：某程序读入三个整数值，这三个整数值表示三角形的三条边长。该程序打印信息表明三角形是不等边三角形、等腰三角思考题形或等边三角形。开发一个测试用例集测试该程序。

16.2 设计和实现思考题 16.1 描述的问题(带有适当的错误处理)。从程序中导出流图，并用基本路径测试方法开发保证测试所有程序语句的测试用例。执行测试用例，并显示结果。

16.3 请列举出 16.1.1 节中没有讨论的测试目标。

16.4 用基本路径测试技术测试实现思考题 14.23 到 14.31 的程序。

16.5 规约、设计和实现一个计算所选语言环形复杂性的软件工具。利用图矩阵作为设计的数据结构。

16.6 阅读 Beizer [BEI95] 一文并尝试使思考题 16.5 中开发的程序如何扩展以适应各种连接权值。扩展你的工具以处理执行概率或连接处理时间。

16.7 使用 16.5.1 节描述的条件测试方法为思考题 16.2 中的程序设计测试用例集。

16.8 使用 16.5.2 节描述的数据流测试方法列出题 16.2 中的程序定义——使用链。

16.9 设计一个自动化工具，使其能够识别循环并按照 16.5.3 节所描述的方法分类。

16.10 扩展思考题 16.9 描述的工具，可以为每种遇到的循环类型设计测试用例。有必要让测试人员交互式使用该工具。

16.11 列出至少三个例子，使得黑盒测试能够给出“一切正常”的印象，而白盒测试可能发现错误。列出至少三个例子，使得白盒测试给出“一切正常”的印象，而黑盒测试可能发现错误。

16.12 穷尽测试(即便可能是对非常小的程序)是否能够保证程序 100％正确？

16.13 使用等价划分方法为本书前面描述的 SafeHome 系统导出测试用例集。

16.14 使用边界值分析方法为思考题 12.13 所描述的 PHTRS 系统导出测试用例集。

16.15 选择熟悉的 GUI 并为之设计一组测试用例集。

16.16 测试某个经常使用的软件手册(或帮助)，以发现文档中的错误。

① 此处只列出了Davis需求工程原则的小部分。有关详细信息，请参见［DAV95］。

② 后面的几段由Jamesach拥有版权，并摘自新闻组comp.software-eng.内容的使用经过了允许。

① SafeHome是前面章节已经使用的家庭安全系统例子。

① 事实上，没有流图也可以执行基本路径测试。但是，用流图可以很好地理解控制流程和阐明方法。

① 第 16.6.1 章详细讨论了图及其在测试时的用法。

① 16.5.1 节和 16.5.2 节经K.C.Tai教授的同意摘自［TAI89］。

① 黑盒测试有时也称为行为测试或分隔测试。

② 在此，术语“对象”包括在第 11 和 12 章讨论的数据对象和程序对象，如模块和编程语言句集。

① 如果上述概念看起来相当熟悉，请回忆 16.4.1 节讨论的为基本路径测试方法创建的图。程序节点代表过程序设计表示或源代码，有向边表示程序对象间的控制流。此处，也将图用于黑盒测试。

① 在此，文档指打印的手册和联机帮助。

第 17 章　软件测试策略

软件测试策略把软件测试用例的设计方法集成到一系列已经周密计划过的步骤中去，从而使得软件的开发得以成功的完成。同样重要的是，软件测试策略为软件开发人员、质量保证组织、和客户提供了一个路线图——这个路线图描述了测试的步骤，以及当这些步骤在计划和实施的过程中，需要多少工作量、时间、和资源。因此，任何测试策略都必须和测试计划、测试用例设计、测试执行、还有测试结果数据的收集与分析结合在一起。

一种软件测试策略应当具备足够的灵活性，这样在必要的时候它能够有足够的创造性和可塑性来应付所有的大软件系统。与此同时，软件测试策略还必须保证足够的严格，这样才能保证对项目的整个进程进行合理的计划和跟踪管理。Shooman［SHO83］对这个问题进行了探讨：

在许多情况下，测试是一个独立的过程，不同的测试类型的数量和不同的开发方法是一样多。许多年以来，我们对付程序出错的唯一武器就是谨慎的设计，以及程序员个人的智慧。我们现在处于这样的一个时代——现代设计技术(和正式的技术复审)正在帮助我们减少代码中存在的初始错误。类似地，不同的测试方法正在开始聚合为有限的几种方法和思想。

这些方法和思想就是我们所说的策略。在第 16 章中，我们已经介绍了软件测试技术①。在本章中，我们将会把注意力放在软件测试策略上。

17.1 软件测试的策略途径

测试是一系列可以事先计划并且可以系统地进行管理的活动。正是由于这个原因，应当为软件工程过程定义一个软件测试的模板——即我们可以把特定的测试用例设计方法放置进去的一系列步骤。

人们已经提出了许多软件测试策略，所有这些策略都为软件开发人员提供了一个供测试用的模板，而且它们都包含下列的类属特征：

- 测试开始于模块层[②]，然后“延伸”到整个基于计算机的系统集合中。
- 不同的测试技术适用于不同的时间点。
- 测试是由软件的开发人员和(对大型系统来说)独立的测试组来管理的。
- 测试和调试是不同的活动，但是调试必须能够适应任何的测试策略。

软件测试策略必须提供可以用来检验一小段源代码是否得以正确实现的低层测试，同时也要提供能够验证整个系统的功能是否符合用户需求的高层测试。一种策略必须为使用者提供指南，并且为管理者提供一系列的重要的程碑。因为测试策略的步骤是在软件完成的最终期限的压力已经开始出现的时候才开始进行的，所以测试的进度必须是可测量的，而且问题要尽可能早的暴露出来才好。

17.1.1 验证和确认

软件测试是我们通常所讲的一个更为广泛的话题验证和确认(Verification and Validation，V&V)的一个部分。验证指的是保证软件正确地实现了某一特定功能的一系列活动。确认指的则是保证软件的实现满足了用户需求的一系列活动。Boehm [BOE81] 是用另外一种方法来解释这两者的区别的：

验证：“我们是否正确地完成了产品？”

确认：“我们是否完成了正确的产品？”

V&V 的定义还包含了许多我们称作软件质量保证(SQA)的许多活动。

回忆一下我们在第 8 章中对软件质量的讨论。为了获取软件质量而必需的活动可以看作是图 17-1 中所描绘的一些组成部分。软件工程方法提供了质量的基础，分析、设计和构造(编码)方法通过提供一致的技术和可预测的结果而帮助提高质量，正式的技术复审(跟踪检查)有助于保证作为每一个软件工程步骤的结果而产生的工作产品的质量。在这些过程当中，测度和控制被应用于软件配置的每一个元素中。标准和规程也有助于保证一致性，而一个形式化的 SQA 过程保证了“整套质量思想”的实现。

测试是质量可以被评估——更实际点说，错误可以被发现——的最后堡垒，但是，测试不应当被视为一个安全网。象人们所说的那样，“你不能测试质量。如果你开始测试的时候它不在那里，那么当你完成测试的时候它仍然不会在那里”。质量在软件的整个过程中都和软件结合在一起。方法和工具的正确使用，有效的正式技术复审和可靠的管理与测度都可以导致在测试过程中得以认可的质量。

Miller [MIL77] 把软件测试和质量保证联系在一起：“程序测试的内在动机是使用对大规模系统和小规模系统都能节约地并且有效地应用的方法来认可软件的质量。”

需要重点加以注意的是，验证和确认包含了范围很广的 SQA 活动，其中包括正式技术复审、质量和配置审查、性能监控、仿真、可行性研究、文档复审、数据库复审、算法分析、开发测试、质量测试和安装测试 [WAL89] 。虽然测试在 V&V 中发挥着非常重要的作用，但是其他的活动也是必要的。

17.1.2 软件测试的组织

对每一个软件项目来说，在测试开始的时候总会产生一些固有的利益冲突。开发软件的人们现在开始被要求对软件进行测试。这本身来说似乎是无害的：毕竟，谁能比开发人员更了解这个软件呢？不幸的是，这些开发人员有很高的兴趣要急于证明他们的程序是毫无错误的，是按照用户的需求开发的，而且完全能够按照预定的进度和预算完成。这些兴趣和认真地测试是相互冲突的。

从心理学的角度上来讲，软件分析和设计(包括编码)是建设性的工作。软件工程师构造一个计算机程序、程序文档、还有相关的数据结构。和其他任何建设者们一样，软件工程师也对自己的“大厦”感到非常骄傲，而对任何试图摧毁其“大厦”的人嗤之以鼻。当测试开始的时候，就会存在一种微妙的、但确实存在着的、试图要“摧毁”软件工程师建立起来的东西的企图。从开发者的观点来看，测试可以被看作是(从心理上来说)破坏性的。所以开发者只是简单地设计和进行能够证明程序正确性的测试，而不是去尽量发现错误。不幸的是，错误是确确实实地存在着的，而且如果软件工程师不能找到错误，那么客户就会找到它们。

通过上面的这些讨论，常常会导致人们产生如下的误解：(1)软件的开发人员根本不应当参与测试；(2)软件应当给那些会无情地挑毛病的陌生人来测试；(3)测试者只有在测试的步骤即将开始的时候才参与项目。这些想法都是错误的。

软件开发者总是负责程序的单个单元(模块)的测试，保证每个单元能够完成设计的功能。在很多情况下，开发者也进行集成测试——进行完整的程序结构构造(和测试)的步骤。仅仅在软件体系结构完成后，独立测试组织才开始介入。

独立测试组织(ITG)的功能就是为了避免让开发者来进行测试时会引发固有问题。独立地测试可以消除可能存在的利益冲突。毕竟，独立组织中的人员是靠找错误来拿工资的。

然而，软件开发人员并不能把程序交给 ITG 就一走了之，开发人员和 ITG 在软件项目中应当紧密合作，以保证测试顺利进行。而且在测试进行过程中，那么开发人员必须可以去修改测试过程中发现的错误。

ITG 从需求说明过程开始，参与了一个大项目的整个过程(计划和确定测试过程)，从这种意义上来说，它是软件项目组中的一部分。然而，在许多情况下，ITG 是直接向软件质量保证组织负责的，这样它就获得它作为软件开发组织的一部分而可能得不到的独立性。

17.1.3 一种软件测试策略

软件工程的过程可以看作是如图 17－2 所示的一个螺旋结构。最初，系统工程定义了软件的功能，从而引出了软件需求分析，建立了软件的信息域、功能、行为、性能、约束和确认标准。沿着螺旋向内前进，经过设计阶段，最终到达了编码。为了开发计算机软件，我们沿着流线的螺旋前进，每一圈都会降低软件的抽象层次。

软件测试策略也可以放在螺旋的语境里来考虑(图 17-2)。单元测试从螺旋的漩涡中心开始，它着重于软件以源代码形式实现的各个单元；测试沿着螺旋向外前进就到了集成测试，这时的测试则着重于对软件的体系结构的设计和构造；再沿着螺旋向外走一圈，我们就遇到了确认测试，我们要用根据软件需求分析得到的需求对已经建造好的系统进行验证；最后，我们要进行系统测试，也就是把软件和其他的系统元素放在一起进行测试。为了对计算机软件进行测试，我们沿着螺旋的流线向外，每转一圈都扩宽了测试的范围。

从过程的观点来考虑测试的整个过程的话，在软件工程环境中的测试事实上是顺序实现的四个步骤的序列，这些步骤表示在图 17-3 中。最开始，测试着重于每一个单独的模块，以确保每个模块都能正确执行，所以，我们把它叫做单元测试，单元测试大量地使用白盒测试技术，检查每一个控制结构的分支以确保完全覆盖和最大可能的错误检查；接下来，模块必须装配或集成在一起形式完整的软件包，集成测试解决的是验证与程序构造的双重问题，在集成过程中使用最多的是黑盒测试用例设计技术，当然，为了保证覆盖一些大的分支，也会用一定数量的白盒测试技术；在软件集成(构造)完成之后，一系列高级测试就开始了，确认标准(在需求分析阶段就已经确定了的)必须进行测试，确认测试提供了对软件符合所有功能的、行为的和性能的需求的最后保证，在确认过程中，只使用黑盒测试技术。

最后的高级测试步骤已经跳出了软件工程的边界，而属于范围更广的计算机系统工程的一部分，软件，一旦经过验证之后，就必须和其他的系统元素(比如硬件、人员、数据库)结合在一起。系统测试要验证所有的元素能正常地啮合在一起，从而完成整个系统的功能/性能。

17.1.4 测试完成的标准

每当讨论软件测试的时候都会引发一个经典问题的讨论：“我们什么时候做测试呢？我们又怎样才能知道我们的测试已经足够了呢？”很遗憾的是，到目前为止对这个问题还没有一个确定性的答案，但是还是有一些在经验引导下的实际的答案和早期的尝试。

对上面问题的一个答复是：“你永远也不可能完成测试，这个重担将会简单地从你(或者开发人员)身上转移到你的客户身上”。每次客户/用户执行一个计算机程序的时候，程序就是在一个新的数据集合下经受测试的考验，这个清醒的事实使得其他的软件质量保证活动更加重要。对上面问题的另一个答复是(有些讽刺，但无疑是准确的)：“当你时间不够或者资金不够用的时候，就完成了测试。”

虽然很少有人会对这些答复产生异议，但是作为一个软件工程师，需要有更严格的标准来决定是否已经进行了足够的测试。Musa 和 Ackerman [MUS89] 提出了一个基于统计标准的答复：“不，我们不能绝对地认定软件永远也不会再出错，但是相对于一个理论上合理的和在试验中有效的统计模型来说，如果一个在按照概率的方法定义的环境中，1000 个 CPU 小时内不出错的操作概率大于 0.995 的话，那么我们就有 95％的信心说我们已经进行了足够的测试。”

使用概率论模型和软件可靠性理论，可以建立一种作为执行时间的函数软件故障(在测试过程中发现的错误)模型 [MUS89] 。一个称为对数泊松执行时间模型(logarithmic Poissonexecution-time model)的软件故障模型为：

$$f(t)=(1/p)\ln(l_0pt+1) \qquad (17.1)$$

其中 f(t)=软件在一定的测试时间 t 后，可能会发生故障的预期累计数目。

l_0=在测试刚开始时的初始软件故障密度(单位时间内的故障数)。

p=错误被发现和修正的过程中故障密度的指数递减值。

瞬时的故障密度，l(t)可以使用 f(t)的导数得出，

$$l(t)=l0/(l0pt+1) \qquad (17.2)$$

使用等式(17.2)中给出的关系，测试人员可以预测测试进程中错误的急剧减少。实际的错误密度可以画在预测曲线上(图 17-4)。如果在测试过程中实际收集的数据和对数泊松执行时间模型能够在大量数据下都相当好地接近的话，那么这个模型就可以用来预测为了达到一个可以接收的低故障密度，整个测试过程所需要的时间。

通过在软件测试过程中收集数据和利用现有的软件可靠性模型，就可能得到回答“测试什么时候完成”这种问题的有意义的指导原则。毫无疑问的是，在测试的量化规则建立之前仍然需要大量的进一步工作，但是，现有经验理论总比直觉要好不少。

17.2 策略问题

在本章的后面部分，我们会探讨一个软件测试的系统化策略。但是，如果无视一些重要的问题的话，那么即使是最好的策略也会失败。 Tom Gilb[GIL95]指出如果要实现一个成功的软件测试策略的话，下面的问题是必须涉及的：

在着手开始测试之前较长时间内，就要以量化的形式确定产品的需求。虽然测试的主要目的是找错误，一个好的测试策略同样可以评估其他的质量特性，比如可移植性、可维护性和可用性(第 18 章)。这些应当用一种可以测度的方式来表示，从而保证测试结果是不含糊的。

明显地指出测试目标。测试的特定目标应当用可以测度的术语来描述。比如，测试有效性、测试覆盖率、故障出现的平均时间、发现和改正缺陷的开销、允许剩余的缺陷密度或出现频率、以及每次回归测试的工作时间都应当在测试计划中清楚地说明[GIL95]。

了解软件的用户并为每一类用户建立相应档案通过着重于测试产品的实际用途。“使用实例”——描述每一类用户的交互情况图(第 20 章)研究可以减少整个测试的工作量。

建立一个强调“快速循环测试”的测试计划。Gilb[GIL95]建议软件工程队伍“学会以对客户有用的功能添加或/和质量改进的快速循环测试方法(用百分之二的项目开销)进行测试”。从这些快速循环测试中得到的反馈可以用来控制质量的级别和相应的测试策略。

设计一个能够测试自身是否“强壮”的软件。软件可以使用反调试技术(第 17.3.1 节)的方法来设计，这就是说，软件应当能够诊断特定类型的错误，另外，设计应当能够包括自动测试和回归测试。

使用有效的正式技术复审作为测试之前的过滤器。正式技术复审(第 8 章)在发现错误方面可以和测试一样有效，由于这个原因，复审可以减少为了得到高质量软件所需的测试工作量。

使用正式技术复审来评估测试策略和测试用例本身。正式的技术复审可以发现在测试过程中的不一致性、遗漏和完全的错误。这样可以节省时间，同时也能够提高产品的质量。

为测试过程建立一种连续改善的实现方法。测试策略必须进行测量，在测试过程中收集的度量数据应当被用作软件测试的统计过程控制方法的一部分。

17.3 单元测试

单元测试完成对最小的软件设计单元——模块的验证工作。使用过程设计描述作为指南，对重要的控制路径进行测试以发现模块内的错误。测试的相关复杂度和发现的错误是由单元测试的约束范围来限定的。单元测试通常情况下是面向白盒的，而且这个步骤可以针对多个模块并行进行。

17.3.1 单元测试考虑

作为单元测试的一部分而出现的测试在图 17-5 中用图形的方式说明。对模块接口的测试保证在测试时进出程序单元的数据流是正确的，对局部数据结构的检查保证临时存储的数据在算法执行的整个过程中都能维持其完整性，对边界条件的测试保证模块在极限或严格的情形下仍然能够正确执行，在控制结构中的所有独立路径(基本路径)都要走遍，以保证在一个模块中的所有语句都能执行至少一次，最后，要对所有处理错误的路径进行测试。

对穿越模块接口的数据流的测试需要在任何其他测试开始之前进行，如果数据不能正确地输入和输出的话，所有的其他测试都是没有实际意义的，Myers[MYE79]在他关于软件测试的文章中提出了接口测试的一个清单：1.输入的形参数目是否等于实参的数目？

2.实参和形参的属性是否匹配？

3.实参和形参的单元系统是否匹配？

4.传递给被调用模块的实参数目是否等于形参的数目？5.传递给被调用模块的实参属性是否等于形参的属性？6.传递给被调用模块的实参的单元系统是否等于形参的单元系统？7.传递给内置函数的数值属性和参数顺序是否正确？

8.任何对参数的引用和当前入口点是否有关联？

9.只输入的参数是否被改变了？

10.跨模块的全局变量定义是否一致？

11.约束条件是否作为参数传递？

当一个模块执行外部 I/O 操作的时候，必须进行附加的接口测试，下面的列表仍来自 Myers[MYE79]：

1.文件属性是否正确？

2.OPEN/CLOSE 语句是否正确？

3.格式规约是否和 I/O 语句匹配？

4.缓冲区大小是否和记录大小匹配？

5.文件是否在打开之前被使用？

6.是否处理了文件结束条件？

7.是否处理了 I/O 错误？

8.在输出信息里是否有文本错误？

模块的局部数据结构是经常出现的错误源。应当设计测试用例以发现下列类型的错误：

1.不正确或者不一致的类型描述。

2.错误的初始化或缺省值。

3.不正确的(拼写错误的或被截断的)变量名字。

4.不一致的数据类型。

5.下溢、上溢和地址错误。

除了局部数据结构，全局数据对模块的影响在单元测试过程中也应当进行审查。

在单元测试过程中，对执行路径的选择性测试是最主要的任务。测试用例应当能够发现由于错误计算、不正确的比较、或者不正常的控制流而产生的错误。基本路径和循环测试是发现更多的路径错误的一种有效技术。

其他常见的错误有：(1)误解的或者不正确的算术优先级；(2)混合模式的操作；(3)不正确的初始化；(4)精度不够精确；(5)表达式的不正确符号表示。比较和控制流是紧密地耦合在一起的(比如，控制流的转移是在比较之后发生的)，测试用例应当能够发现下列错误：(1)不同数据类型的比较；(2)不正确的逻辑操作或优先级；(3)应该相等的地方由于精度的错误而不能相等；(4)不正确的比较或者变量；(5)不正常的或者不存在的循环中止；(6)当遇到分支循环的时候不能退出；(7)不适当地修改循环变量。

好的设计要求错误条件是可以预料的，而且当错误真的发生的时候，错误处理路径被建立，以重定向或者干净地终止处理。Yourdon[YOU75]把这种方法叫做反调试(antidebugging)，不幸的是，存在一种把错误处理过程加到软件中去，但从不进行测试的倾向。这里有一个现实生活中的故事可以说明这个问题：

一个交互式设计系统按照合同进行开发。在一个事务处理模块中，开发人员将错误处理信息“错误！你不可能到达这里！”加入到调用各种控制流分支的一系列条件测试之后。这个“错误信息”在用户培训过程中被一个客户发现了！

在错误处理部分应当考虑的潜在错误有下列情况：

1.对错误描述的莫名其妙。

2.所报的错误与真正遇到的错误不一致。

3.错误条件在错误处理之前就引起了系统异常。

4.例外条件处理不正确。

5.错误描述没有提供足够的信息来帮助确定错误发生的位置。

边界测试是单元测试的最后(而且可能也是最为重要的)一个步骤。软件通常是在边界情况下出现故障的，这就是说，错误往往出现在一个 n 元数组的第 n 个元素被处理的时候，或者一个 i 次循环的第 i 次调用，或者当允许的最大或最小数值出现的时候。使用刚好小于、等于和刚好大于最大值和最小值的数据结构、控制流、数值来作为测试用例就很有可能发现错误。

17.3.2 单元测试规程

单元测试通常看成为是编码步骤的附属品。在源代码级的代码被开发、复审、和语法正确性验证之后，单元测试用例设计就开始了。对设计信息的复审可能能够为建立前面讨论过的每一类错误的测试用例提供指导，每一个测试用例都应当和一系列的预期结果联系在一起。

因为一个模块本身不是一个单独的程序，所以必须为每个单元测试开发驱动器或/和稳定桩(stub)。单元测试的环境如图 17-6 所示。在绝大多数应用中，一个驱动只是一个接收测试数据，并把数据传送给(要测试的)模块，然后打印相关结果的“主程序”。毫无错误的子程序桩的功能是替代那些隶属于本模块(被调用)的模块。一个毫无错误的子程序桩或“空子程序”可能要使用子模块的接口，才能做一些少量的数据操作，并验证打印入口处的信息，然后返回。

驱动器和稳定桩都是额外的开销，这就是说，两种都属于必须开发但又不能和最终软件一起提交的软件，如果驱动器和稳定桩很简单的话，那么额外开销相对来说是很低的。不幸的是，许多模块使用“简单”的额外软件是不能进行足够的单元测试的。在这些情况下，完整的测试要推迟到集成测试步骤时再完成。

当一个模块被设计为高内聚时，单元测试是很简单的。当一个模块只表示一个函数时，测试用例的数量就会降低，而且错误也就更容易被预测和发现。

17.4 集成测试

一个在软件世界里初出茅庐的年轻人可能在所有的模块都已经完成单元测试之后会问这样一个似乎很合理的问题：“如果它们每一个都能单独工作得很好，那么你为什么要怀疑把它们放在一起就不能正常工作呢？”当然，这个问题

就在于“把它们如何放在一起？”——接口。数据可能在通过接口的时候丢失；一个模块可能对另外一个模块产生无法预料的副作用；当子函数被联到一起的时候，可能不能达到期望中的功能；在单个模块中可以接受的不精确性在联起来之后可能会扩大到无法接受的程度；全局数据结构可能也会存在问题——还有很多，很多。

集成测试是通过测试发现和接口有关的问题来构造程序结构的系统化技术，它的目标是把通过了单元测试的模块拿来，构造一个在设计中所描述的程序结构。

通常存在进行非增量集成的倾向，也就是说，使用一步到位的方法来构造程序。所有的模块都预先结合在一起，整个程序作为一个整体来进行测试，然后的结果通常是混乱不堪！会遇到许许多多的错误，错误的修正也是非常困难的，因为在整个程序的庞大区域中想要分离出一个错误是很复杂的。一旦这些错误被修正之后，就马上会有新的错误出现，这个过程会继续下去，而且看上去似乎是个无限循环的。

增量集成是一步到位的方法的对立面。程序先分成小的部分进行构造和测试，这个时候错误比较容易分离和修正；接口也更容易进行彻底地测试；而且也可以使用一种系统化的测试方法。在下面的章节中，将要讨论许多不同的增量集成策略。

17.4.1 自顶向下集成

自顶向下的集成是一种构造程序结构的增量实现方法。模块集成的顺序是首先集成主控模块(主程序)，然后按照控制层次结构向下进行集成。隶属于(和间接隶属于)主控模块的模块按照深度优先或者广度优先的方式集成到整个结构中去。

如图 17-7 所示，深度优先的集成首先集成在结构中的一个主控路径下的所有模块。主控路径的选择是有些任意的，它依赖于应用程序的特性，例如，选择最左边的路径，模块M_1，M_2，和M_5，将会首先进行集成，然后是M_8或者M_6(如果对M_2的适当的功能是必要的)，然后，开始构造中间的和右边的控制路径。广度优先的集成首先沿着水平的方向，把每一层中所有直接隶属于上一层模块的模块集成起来，从图中来说，模块M_2，M_3和M_4首先进行集成，然后是下一层的M_5，M_6，然后继续。

集成的整个过程由下列五个步骤来完成：

1.主控模块作为测试驱动器，所有的稳定桩替换为直接隶属于主控模块的模块。

2.根据集成的实现方法(如深度或广度优先)，下层的稳定桩一次一个地被替换为真正的模块。

3.在每一个模块集成的时候都要进行测试。

4.在完成了每一次测试之后，又一个稳定桩被用真正的模块替换。

5.可以用回归测试(第 17.4.3 节)来保证没有引进新的错误。

整个过程回到第 2 步循环继续进行，直至这个系统结构被构造完成。

自顶向下的集成策略在测试过程的早期主要验证控制和决策点。在一个好的程序结构中，决策的确定往往发生在层次结构中的高层，因此首先会被遇到。如果主控制的确存在问题，尽早地发现它是很重要的。如果选择了深度优先集成，软件的某个完整的功能会被实现和证明，例如，考虑一个经典的事务性结构(第 14 章)，在这个结构中，有一系列复杂的交互式输入要通过一条输入路径请求、获得和验证，这条输入路径就可以用自顶向下的方式来进行集成。早期的对功能性的验证对开发人员和客户来说都是会增加信心的。

自顶向下的策略似乎相对来说不是很复杂，但是在实践过程中，可能会出现逻辑上的问题。最普通的这类问题出现在当高层测试需要首先对较低层次的足够测试后才能完成的时候。在自顶向下的测试开始的时候，稳定桩代替了低层的模块，因此，在程序结构中就不会有重要的数据向上传递，测试者只有下面的三种选择：(1)把测试推迟到稳定桩被换成实际的模块之后再进行，(2)开发能够实现有限功能的用来模拟实际模块的稳定桩，或者(3)从层次结构的最底部向上来对软件进行集成。图 17-8 给出了典型的几种稳定桩类型，从最简单的(稳定桩 A)到最复杂的(稳定桩 D)。

第一种实现方法(把测试推迟到稳定桩被换成实际的模块之后再进行)使我们失去了对许多在特定测试和特定模块组合之间的对应性的控制，这样可能导致在确定错误发生原因时的困难性，并且会违背自顶向下方法的高度受限的本质。第二中方法是可行的，但是会导致很大的额外开销，因为稳定桩会变的越来越复杂。第三种方法，也就是自底向上的测试，将在下一节加以讨论。

17.4.2 自底向上集成

自底向上的测试，就象它的名字中所暗示的一样，是从原子模块(比如在程序结构的最低层的模块)开始来进行构造和测试的，因为模块是自底向上集成的，在进行时要求所有隶属于某个给定层次的模块总是存在的，而且也不再有使用稳定桩的必要。

自底向上的集成策略可以使用下列步骤来实现：

1.低层模块组合成能够实现软件特定子功能的簇。

2.写一个驱动程序(一个供测试用的控制程序)来协调测试用例的输入输出。

3.对簇进行测试。

4.移走驱动程序，沿着程序结构的层次向上对簇进行组合。

这样的集成遵循在图 17-9 中说明的模式，首先把所有的模块聚集成三个簇 1，2，3，然后用对每一个簇使用驱动器(图中的虚线框)进行测试，在簇 1 和簇 2 中的模块隶属于M_a，把驱动D_1和D_2去掉，然后把这两个簇和M_a直接连在一起。类似地，驱动器D_3也在模块M_b集成之前去掉。M_a和M_b最后都要和模块M_c一起进行集成，驱动器的不同种类如图 17-10 所示。

当测试在向上进行的过程中，对单独的测试驱动器的需求减少了，事实上，如果程序结构的最上两层是自顶向下集成的，那么所需的驱动数目就会明显的减少，从而对簇的集成会变得非常简单。

17.4.3 回归测试

每当一个新的模块被当作集成测试的一部分加进来的时候，软件就发生了改变。新的数据流路径建立了起来，新的 I/O 操作可能也会出现，还有可能激活了新的控制逻辑。这些改变可能会使原本工作得很正常的功能产生错误。在集成测试策略的环境中，回归测试是对某些已经进行过的测试的某些子集再重新进行一遍，以保证上述改变不会传播无法预料的副作用。

在更广的环境里，(任何种类的)成功测试结果都是发现错误，而错误是要被修改的，每当软件被修改的时候，软件配置的某些方面(程序、文档、或者数据)也被修改了，回归测试就是用来保证(由于测试或者其他原因的)改动不会带来不可预料的行为或者另外的错误的活动。

回归测试可以通过重新执行所有的测试用例的一个子集人工地来进行，也可以使用自动化的捕获回放工具来进行。捕获回放工具使得软件工程师能够捕获到测试用例，然后就可以进行回放和比较。回归测试集(要进行的测试的子集)包括三种不同类型的测试用例：

- 能够测试软件的所有功能的代表性测试用例。
- 专门针对可能会被修改影响的软件功能的附加测试。
- 针对修改过的软件成分的测试。

在集成测试进行的过程中，回归测试可能会变的非常庞大。因此，回归测试集应当设计为只包括那些涉及在主要的软件功能中出现的一个或多个错误类的那些测试，每当进行一个修改时，就对每一个程序功能都重新执行所有的测试是不实际的而且效率很低的。

17.4.4 关于集成测试的讨论

关于自顶向下和自底向上的集成测试的相对优缺点有许多人进行了探讨(比如：[BEI84])，总的来说，一种策略的优点差不多就是另一种策略的缺点。自顶向下的方法的主要缺点是需要稳定桩和与稳定桩有关的附加测试困难。和稳定桩有关的问题可以被主要控制功能的尽早测试来抵消。自底向上的集成的主要缺点就是“直到最后一个模块被加进去之后才能看到整个程序的框架”[MYE79]，该缺点由简单的测试用例设计和不用稳定桩来弥补的。

选择一种集成策略依赖于软件的特性，有的时候还有项目的进度安排。总的来说，一种组合策略(有时候被称作是三明治测试)可能是最好的折衷：在程序结构的高层使用自顶向下策略，而在下面的较低层中使用自底向上策略。

当集成测试进行时，测试人员应当能够识别关键模块。一个关键模块具有一个或多个下列特性：(1)和好几个软件需求有关；(2)含有高层控制(位于程序结构的高层)；(3)本身是复杂的或者是容易出错的；(4)含有确定性的性能需求。关键模块应当尽可能早地进行测试，另外，回归测试也应当集中在关键模块的功能上。

17.4.5 集成测试文档

软件集成的总体计划和详细的测试描述要按照测试规约来写入文档。规约是软件工程过程中的重要文件，已经成为了软件配置的一部分。表 17-1 给出了一个测试规约的大纲，可以用作这一类文档的框架。

表 17-1　测试规约大纲

Ⅰ.测试范围	1.对模块 n 的测试描述
Ⅱ.测试计划	2.额外软件的描述
A.测试阶段和结构	3.期望的结果
B.进度	C.测试环境
C.额外的软件	1.特殊的工具和技术
D.环境和资源	2.额外软件的描述
Ⅲ.测试过程 n(对结构 n 的测试的描述)	D.测试用例数据
A.集成顺序	E.建立 n 所期望的结果
1.目的	Ⅳ.实际的测试结果
2.要测试的模块	Ⅴ.参考文献
B.对结构中模块单元的测试	Ⅵ.附录

“测试的范围”总结了要进行测试的特定功能的、性能的、和内部设计的特征；界定了测试工作量；描述了每一个测试阶段完成的标准；并把对进度的约束写入文档。

“测试计划”部分描述了集成的总体策略，把测试划分为涉及软件的特定功能和行为特征的阶段和结构。例如，对一个面向图形的 CAD 系统的集成测试可以被划分为下面的测试阶段：

- 用户交互(命令选择、图形生成、显示的表示、错误处理和表示)。
- 数据操作和分析(符号创建、维数转化、旋转、物理属性的计算)。
- 显示的处理和生成(二维显示、三维显示、图表)。
- 数据库管理(访问、更新、完整性、性能)。

所有这些阶段和子阶段(在括号中说明的)勾勒出了在软件内部的比较大的功能类别，所以通常来说可以和程序结构的一个特定领域联系起来，因此，对每一个阶段都建立了相应的程序结构(程序模块集)。

下面的标准和相应的测试被应用于所有的测试阶段：

接口完整性。在每一个模块集成到整个结构中去的时候，要对其内部和外部接口进行测试。

功能有效性。进行以发现功能性错误为目的的测试。

信息内容。进行以发现和局部或全局数据结构相关的错误为目的的测试。

性能。在进行软件设计的过程中，设计用来验证性能边界的测试。

这些标准与和它们相联系的测试都在测试规约的这一节中讨论。

集成的进度、额外的软件和相关的话题也在“测试计划”一节中讨论。确定每一阶段的开始和结束日期，定义测试模块单元的可用性窗口，对额外软件(稳定桩和驱动器)的简单描述着重于可能会需要特殊工作量的特征，最后，描述测试环境和资源、特殊的硬件配置、奇异的仿真器、特殊的测试工具和技术也是在这一节中要讨论的众多话题。

为了完成在“测试过程”一节中描述的测试计划，需要有一个详细的测试过程。在测试规约大纲的第Ⅲ条中，描述了每一个测试步骤的集成顺序和相应的测试。其中还含有全部测试用例(和次要引用注释)和期望有结果。实际测试结果、问题或特性的历史值记录在测试说明的第四部分。软件维护时该部分的信息是很有用的核心，在最后两节中给出的是相应的参考文献和附录。

象软件配置的其他所有元素一样，可以根据软件开发组织的具体需要裁剪测试规约格式，然而，需要指出的是，在测试计划中的集成策略和在测试过程中描述的测试细节是最基本的成分，因而是必须出现的。

17.5 确认测试

当集成测试结束的时候，软件就全部组装到一起了，接口错误已经被发现并修正了，而软件测试的最后一部分(确认测试)就可以开始了。确认可以通过多种方式来定义，但是，一个简单(虽然很粗糙)的定义是当软件可以按照用户合理地期望的方式来工作的时候，确认即算成功。从这一点上，一个爱挑毛病的软件开发人员可能会提出抗议：“谁或者什么来作为合理的期望的裁定者呢？”

合理的期望在描述软件的所有用户可见的属性文档——软件需求规约中(第12章)被定义。这个规约包含了标题为“确认标准”的一节内容，在这节中的信息就形成了确认测试方法的基础。

17.5.1 确认测试的标准

软件确认通过一系列证明软件功能和需求一致的黑盒测试来达到。测试计划列出了要进行的测试种类，定义了为了发现和需求不一致的错误而使用的详细测试实例的测试过程。计划和过程都是为了保证所有的功能需求都得到了满足；所有性能需求都达到了；文档是正确且合理的；还有其他的需求也都满足了(比如，可移植性，兼容性，错误恢复，可维护性)。

在每个确认测试实例进行时，会出现以下两种可能的条件之一：(1)和需求说明一致的功能或性能特性是可接受的，或者(2)和需求说明的偏差被发现时，要列出问题清单。一个项目中在这个阶段所发现的偏差或者错误是无法在原定进度下得到修改的。和客户协商一下解决这些缺陷的方法往往是有必要的。

17.5.2 配置复审

确认过程中的一个重要元素是配置复审。复审的目的是保证软件配置的所有元素都已进行正确地开发和分类，而且有支持软件生命周期维护阶段的必要细节。配置复审，有时候被称为审计(audit)，在第 9 章中已经进行了详细的讨论。

17.5.3 Alpha 和 Beta 测试

软件开发者想要预见到用户是如何实际地使用程序实质上是不可能的。使用的命令可能会被误解；还可能经常有奇怪的数据组合出现；输出对测试者来说似乎是很清晰的，但对在这个领域里的用户可能会是无法理解的。

如果软件是给一个客户开发的，需要进行一系列的接收测试来保证客户对所有的需求都满意。接收测试是由最终用户而不是系统开发者来进行的，它的范围从非正式的“测试驱动”直到有计划的系统化进行的系列测试。事实上，接收测

试可以进行几个星期或者几个月，因此可以发现随着时间流逝可能会影响系统的累积错误。

如果一个软件是给许多客户使用的，那么让每一个用户都进行正式的接收测试是不切实际的。大多数软件厂商使用一个被称作 alpha 测试和 beta 测试的过程来发现那些似乎只有最终用户才能发现的错误。

alpha 测试是由一个用户在开发者的场所来进行的，软件在开发者对用户的“指导”下进行测试，开发者负责记录错误和使用中出现的问题，alpha 测试是在一个受控的环境中进行的。

beta 测试是由软件的最终用户在一个或多个用户场所来进行的，不象 alpha 测试，开发者通常来说不会在场，因此，beta 测试是软件在一个开发者不能控制的环境中的“活的”应用。用户记录下所有在 beta 测试中遇到的(真正的或是想象中的)问题，并定期把这些问题报告给开发者，在接到 beta 测试的问题报告之后，开发者对系统进行最后的修改，然后就开始准备向所有的用户发布最终的软件产品。

17.6 系统测试

在本书的开始，我们就强调过软件只是一个大的计算机系统的一个构成成分这个事实。在最后，软件要和其他的系统成分(比如，新的硬件、信息)集成起来，然后要进行系统集成和确认测试。这些测试不属于软件工程过程的研究范围，而且也不只是由软件开发人员来进行的。然而，在软件设计和测试阶段采用的步骤能够大大增加软件成功地在大的系统中进行集成的可能性。

一个经典的系统测试问题是“互相指责”。当一个错误出现时，每一个系统构件的开发者都会指责其他构件的开发者要对此负责。软件工程师不应当沉溺于这种无谓的斗嘴，而应当能够预料到潜在的接口问题和(1)设计测试所有从系统的其他元素来的信息的错误处理路径；(2)在软件接口处进行一系列仿真错误数据或者其他潜在错误的测试；(3)记录测试的结果作为当“互相指责”时出现的“证据”；以及(4)参与系统测试的计划和设计来保证系统进行了足够的测试。

系统测试事实上是对整个基于计算机的系统进行考验的一系列不同测试。虽然每一个测试都有不同的目的，但所有都是为了整个系统成分能正常地集成到一起以完成分配的功能而工作的。在下面的几节中，我们将会讨论对基于计算机的系统有用的系统测试类型[BEI84]。

17.6.1 恢复测试

许多基于计算机的系统必须在一定的时间内从错误中恢复过来，然后继续运行。在有些情况下，一个系统必须是可以容错的，这就是说，运行过程中的错误

必须不能使得整个系统的功能都停止。在其他情况下，一个系统错误必须在一个特定的时间段之内改正，否则就会产生严重的经济损失。

恢复测试是通过各种手段，让软件强制性地发生故障，然后来验证恢复是否能正常进行的一种系统测试方法。如果恢复是自动的(由系统本身来进行的)，重新初始化、检查点机制、数据恢复和重启动都要进行正确验证。如果恢复是需要人工干预的，那么要估算修复的平均时间是否在可以接受的范围之内。

17.6.2 安全测试

任何管理敏感信息或者能够对个人造成不正当伤害的计算机系统都是不正当的或非法侵入的目标。侵入包括了范围很广的活动：只是为练习而试图侵入系统的黑客；为了报复而试图攻破系统的有怨言的雇员；还有为了得到非法的利益而试图侵入系统的不诚实的个人。

安全测试用来验证集成在系统内的保护机制是否能够在实际中保护系统不受到非法侵入。引用 Beizer 的话来说[BEI84]：“系统的安全当然必须能够经受住正面的攻击——但是它也必须能够经受住侧面的和背后的攻击。”

在安全测试过程中，测试者扮演着一个试图攻击系统的个人角色。就是这样！测试者可以尝试去通过外部的手段来获取系统的密码，可以使用可以瓦解任何防守的客户软件来攻击系统；可以把系统“制服”，使得别人无法访问；可以有目的地引发系统错误，期望在系统恢复过程中侵入系统；可以通过浏览非保密的数据，从中找到进入系统的钥匙；等等。

只要有足够的时间和资源，好的安全测试就一定能够最终侵入一个系统。系统设计者的任务就是要把系统设计为想要攻破系统而付出的代价大于攻破系统之后得到的信息的价值。

17.6.3 压力测试

在较早的软件测试步骤中，白盒和黑盒技术对正常的程序功能和性能进行了详尽的检查。压力测试(strees testing)的目的是要对付非正常的情形。在本质上说，进行压力测试的人应该这样问：“我们能够将系统折腾到什么程度而又不会出错？”

压力测试是在一种需要反常数量、频率或资源的方式下执行系统。例如，(1)当平均每秒出现 1 个或 2 个中断的情形下，应当对每秒出现 10 个中断的情形来进行特殊的测试；(2)把输入数据的量提高一个数量级来测试输入功能会如何响应；(3)应当执行需要最大的内存或其他资源的测试实例；(4)使用在一个虚拟的操作系统中会引起颠簸的测试实例；或者(5)可能会引起大量的驻留磁盘数据的测试实例。从本质上来说，测试者是想要破坏程序。

压力测试的一个变种是一种被称为是敏感测试的技术。在有些情形(最常见的是在数学算法中)下，在有效数据界限之内的一个很小范围的数据可能会引起极端的甚至是错误的运行，或者引起性能的急剧下降，这种情形和数学函数中的奇点相类似。敏感测试就是要发现在有效数据输入里的可能会引发不稳定或者错误处理的数据组合。

17.6.4 性能测试

在实时系统和嵌入系统中，提高符合功能需求但不符合性能需求的软件是不能接受的。性能测试就是用来测试软件在集成系统中的运行性能的。性能测试可以发生在测试过程的所有步骤中，即使是在单元层，一个单独模块的性能也可以使用白盒测试来进行评估，然而，只有当整个系统的所有成分都集成到一起之后，才能检查一个系统的真正性能。

性能测试经常和压力测试一起进行，而且常常需要硬件和软件测试设备，这就是说，在一种苛刻的环境中衡量资源的使用(比如，处理器周期)常常是必要的。外部的测试设备可以监测执行的间歇，当出现情况(比如中断)时记录下来。通过对系统的检测，测试者可以发现导致效率降低和系统故障的情况。

17.7 调试的技巧

软件测试是一个可以系统地进行计划的过程，可以指导测试用例的设计，定义测试策略，测试结果可以和预期的结果进行对照评估。

成功的测试之后需要调试，也就是说，调试就是在测试发现一个错误后消除错误的过程。尽管调试可以也应该是一个有序的过程，但是仍然还有许多特别的技术。软件工程师评估测试结果时，常常会看到问题的症状。错误的外部表现和它的内部原因并没有明显的关系，调试就是发现问题症状原因的、尚未很好理解的智力过程。

17.7.1 调试过程

调试并不是测试，但总是发生在测试[①]之后。如图 17-11 所示，调试过程从执行一个测试例子开始，得到执行结果并且发生了预期结果和实际结果不一致的情况，在许多情况下，这种不一致表明还有隐藏的问题。调试过程试图找到症状的原因，从而使得能够改正错误。

调试过程总会有以下两种结果之一：(1)发现问题的原因并将之改正及消除，(2)未能发现问题的原因。在后一种情况下，调试人员应假设一个错误原因，设计测试例子帮助验证此假设，并重复此过程最后改正错误。

为什么调试会如此困难呢？在很大困难程度上，人类心理(参见下一节)比软件技术更多地涉及对该问题的答案，不过，错误的以下特征提供了一些线索：

1.症状和原因可能是相隔很远的。也就是说，症状可能在程序的一部分出现，而原因实际上可能在很远的另一个地方。高度耦合的程序结构(第 14 章)加剧了这种情况。

2.症状可能在另一个错误被纠正后消失或暂时性的消失。

3.症状可能实际上并不是由错误引起的(如舍入误差)。

4.症状可能是由不太容易跟踪的人工错误引起的。

5.症状可能是和时间相关的，而不是处理问题。

6.很难重新产生完全一样的输入条件(如一个输入顺序不确定的实时应用)。

7.症状可能是时有时无的。这在那些不可避免的耦合硬件和软件的嵌入式系统中特别常见。

8.症状可能是由分布在许多不同任务中的原因引起的，这些任务运行在不同的处理器上[CHE90]。

在调试过程中，我们会遇到恼人的小错误(比如，不正确的输出格式)到灾难性的大错误(比如系统失效，导致严重的经济和物质损失)。错误越严重，相应的查找错误原因的压力也越大。通常情况下，这种压力会导致软件开发人员修正一个错误的同时引入两个甚至更多的错误。

17.7.2 心理考虑

不幸的是，有证据表明，调试的本领是属于一种个人的先天本领。有些人精于此道，而其他人就不行。虽然关于调试的试验证据表明对此可以有多种解释，但是对于具有相同教育和试验背景的程序员来说，他们的调试能力还是有很大的差别的。

Shneiderman[SHN80]是这样来评价调试的人为因素的：

调试是一种更容易让人感到沮丧的编程工作。它包含解决问题或智力测验，而同时最恼人的是认识到你犯下了错误。高度的焦虑和不愿接受可能发现的错误，会增加这项任务的难度。幸运的是，当错误最终……被改正的时候，你还能感受到强烈的放松。

虽然“学会”调试可能是很困难的，还是有一些办法可以使用的。我们将在下一节中讨论这些方法。

17.7.3 调试方法

无论调试使用什么样的方法，它都有一个最主要的目标：寻找错误的原因并改正之。这个目标是通过有系统的评估、直觉和运气一起来完成的。Bradley[BRA85]如此描述调试的过程：

调试是对过去 2，500 年间一直在发展的科学方法的直接应用。调试是以根据对将被检查的新值的预测的假设通过二分法定位问题源[原因]为基础。

拿一个简单的和软件无关的例子来说：我屋里的一个台灯不亮了。如果整个屋子都没电了，那么一定是总闸或者是外面坏了；我出去看是否邻居家也是黑的。如果不是，我把台灯插到好的插座里试试，或者把别的正常的电器插到原来插台灯的插座里检查一下。这就是假设和检测的过程。

总的来说有三种调试的实现方法(见参考文献[MYE79])：

- 蛮力法(brute force)。
- 回溯法(backtracKing)。
- 原因排除法(cause elimination)。

蛮力法的调试可能是为了找到错误原因而使用的最普通但是最低效的方法了。当所有其他的方法都失败的情形下，我们才会使用这种方法。根据“让计算机自己来寻找错误”的思想，进行内存映象，激活运行时的跟踪，而且程序里到处都是 WRITE 语句。我们希望在这么多的信息的海洋里能够发现一点儿有助于我们找到错误原因的线索。虽然大量的信息可能最终导致成功，但是更多的情况下，只是浪费精力和时间。你必须首先进行思考！

回溯是在小程序中经常能够奏效的相当常用的调试方法。从发现症状的地方开始，开始(手工地)向回跟踪源代码，直到发现错误原因。不幸地是，随着源代码行数的增加，潜在的回溯路径可能会多到无法管理的地步。

第三种调试方法——原因排除法，是通过演绎和归纳，以及二分法来实现的。对和错误发生有关的数据进行分析可寻找到潜在的原因。先假设一个可能的错误原因，然后利用数据来证明或者否定这个假设。也可以先列出所有可能的原因，然后进行检测来一个个地进行排除。如果最初的测试表明某个原因看起来很象的话，那么就要对数据进行细化来精确定位错误。

上面的每一种方法都可以使用调试工具来辅助完成。我们可以使用许多带调试功能的编译器、动态的调试辅助工具(“跟踪器”)、自动的测试用例生成器、内存映象工具、以及交叉引用生成工具。然而，工具是不可能代替基于完整的软件设计文档和清晰代码的人为的细心评价的。

任何对调试方法和工具的讨论都是不完整的，所以我们不得不提到另外一个最有力的助手，那就是：其他人！我们每个人可能都有过好多小时或好几天一直在为一个错误头疼的经历。这时候，有一个同事正好路过，在绝望中我们对他描述了我们遇到的问题，几乎是在刹那间，错误就被他找到了，这个同事洋洋得意地笑着走了。在许多个小时的沮丧的阴云笼罩下，一个新鲜的观点可能创造奇迹。所以对调试的最后的箴言应该是：“如果什么办法都失败了，那么就问问别人！”

一旦找到了错误，那么就必须纠正。但是我们已经提醒过了，修改一个错误可能会带来其他的错误，因此，做得过多将害大于利。 Van Vleck[VAN89]提出了每一个软件工程师在进行排除错误发生原因的“修改”之前都必须问的三个问题：

1.这个错误原因在程序的其他地方也产生过吗？在许多情形下，一个程序错误是由错误的逻辑模式引起的，而这种逻辑模式可能会在别的地方出现。对这种逻辑模式的仔细考虑可以帮助发现其他的错误。

2.我将要进行的修改可能会引发的“下一个错误”是什么？在进行修改之前，需要认真地研究源代码(最好包括设计)的逻辑和数据结构之间的耦合，如果是在修改高度耦合的程序段的话，那么就应当格外的小心。

3.为了防止这个错误，我们首先应当做什么呢？这个问题是建立统计的软件质量保证方法的第一个步骤(第 8 章)。如果我们不仅修改了产品，还修改了过程，那么我们就不仅排除了现在的程序错误，还避免了所有今后的程序可能出现的错误。

17.8 小结

软件测试在软件过程中占有最大百分比的技术工作量，而我们只是刚刚开始理解系统化测试的计划、执行和控制的一些皮毛。

软件测试的目的是发现错误。为了完成这个目标，需要计划和进行一系列的测试步骤——单元、集成、确认和系统测试。单元测试和集成测试侧重于验证一个模块的功能和把模块集成到程序结构中去。确认测试用来验证软件需求，系统测试在软件集成为一个大的系统时才进行。

每一个测试步骤都是通过一系列有助于测试用例设计的系统化测试技术来完成的。在每一步测试中，软件考虑的抽象层次都提高了。

和测试不一样(一个系统化、有计划的活动)，调试则必须被看作是一种艺术。从一个问题的症状开始，调试活动要去追寻错误的原因。在调试过程中可利用众多资源中，最有价值的是和其他软件工程师的协商。

更高质量的软件则需要更系统化的测试方法。引用 Dunn 和 Ullman[DUN82]的话来说：

我们所需要的是贯穿整个测试过程的一个整体策略，而且在方法学上应当象基于分析、设计和编码的系统化软件开发一样地进行周密的计划。

在本章中，我们详细的探讨了测试策略的问题，考虑了达到主要测试目标最可能需要的步骤：以一种有序的和有效的方法来发现并纠正错误。

思考题

17.1 用你自己的话来描述一下验证和确认的区别。它们两个都要使用测试用例设计方法和测试策略吗？

17.2 列出一些可能和独立测试组织有关的问题。ITG 和 SQA 组织是由同样的人来组成的吗？

17.3 使用在 17.1.3 节中所描述的测试步骤序列来建立测试软件的策略总是可能的吗？对于嵌入式系统来说，可能会有那些复杂性呢？

17.4 如果你只能在单元测试过程中选择三种测试用例设计方法，那么应该选择哪几种，为什么呢？

17.5 向在 17.3.1 节中列出的单元测试检查点的每个部分添加三个另外的问题。

17.6“反调试”(17.3.1 节)的概念是当发现错误之后提供内置的调试帮助的非常有效的手段。

a.为反调试建立一些指南。

b.讨论一下使用这种技术的优点。

c.讨论一下缺点。

17.7 为思考题 14.23 到 14.31 实现的任何一个系统建立相应的集成测试策略。定义测试阶段，说明集成的顺序，指出需要的额外测试软件，然后证明你的集成顺序是可行的。假设所有的模块都已经经过了单元测试，并且都是可用的。(注意：你可能需要先进行一点儿设计工作)。

17.8 项目的进度是如何影响集成测试的？

17.9 在所有的情况下单元测试都是可能的或者说是值得做的吗？提供可以证明你的回答的实例。

17.10 谁来进行确认测试？是软件开发人员还是软件最终用户？为什么？

17.11 为在本书前面讨论过的 SafeHome 系统建立一份完整的测试策略。并依据测试规约完成文档。

17.12 作为一个班级项目，为你们的程序开发建立一份调试指南，其中应当提供面向你们在学校里学过的语言和系统的提示。从一个经过全班和老师复审过的大纲开始做起。并把这个指南介绍你周围的其他人。

推荐阅读文献及其他信息源

对测试策略的详细讨论可以在 Kit(Software Testing in the Real World, Addison-Wesley, 1995), Evans(Productive Software Test Management, Wiley-Interscience,1984), Hetzel(TheComplete Guide to Software Testing, QED Information Sciences, 1984), Beizer [BEI84], Ould 和 Unwin(Te sting in Software Development,Cambridge University Press,1986),Marks(Testing VeryBig Systems,McGraw-Hill,1992),和 Kaner 等(Testing Computer Software, 2nd Eition, VanNostrand Reinhold, 1993)的书中找到。每一本书中都描绘了一种有效的策略的各个步骤，提供了一系列的技术和指南，并提出了控制和跟踪测试过程的方式。 Hutcheson(Software TestingMethods and Metrics, McGraw-Hill, 1996)不仅给出了测试的方法和策略，还提供了对如何为了获得有效的测试而使用度量手段的详细讨论。

对开发商品软件的个体来说，Gunther 的书(Management Methodology for Software ProductEngineering, Wiley-Interscience, 1978)对建立和管理有效的测试策略是一本很有用的指南。另外，Perry(How to Test Software Package, Wiley,1986)提供了对产品开发者和购买者都很有用的信息。Mosley(Real World Issues in Client-Server Software Testing, Prentice-Hall, 1996)提出了对基于客户/服务器的应用的测试策略和技术。

和调试有关的内容在一本由 Dunn 写的书(Software Defect Removal, McGraw-Hill, 1984)中可以找到。Beizer[BEI84]也给出了有助于建立测试计划的有效方法的一套很有意思的“错误分类学”书籍。McConnell(Code Complete, Microsoft Press, 1993)给出了包括单元测试、集成测试，和调试在内的很实用的建议。

关于软件测试的Internet上的信息源在第 16 章和第 22 章的“推荐阅读文献和其他信息资源”一节中可以找到。

① 面向对象的测试将在第 22 章中讨论。

② 对面向对象系统，测试开始于类似或对象层。细节参见第 22 章。

① 而向对象的集成测试将在第 22 章中讨论。

① 此处我们考虑最广义的测试，不仅包括软件发布之前开发人员的测试，也包括用户每次使用软件时对软件的测试。

第 18 章　软件的技术度量

测度是任意一个工程过程中的重要元素。运用测度，我们能更好的理解我们所建立的模型的属性，但最重要的是，我们还可以运用测度来评价我们所建立的工程化产品或系统的质量。

不象其他的工程学科，软件工程并不是建立在基本物理定量规律上的。绝对测度，如电压、重量、速度、或者温度，在软件界中并不普遍。相反，我们是要设法获取一套间接测度方法来提供对软件质量的表示。因为软件测度不是绝对的，所以它们是可以讨论的。Fenton[FEN91]在讨论这个问题时谈到：

测度是把数字或符号分配给现实世界实体的属性，根据明确定义的规则来定义它们……在物理科学、医学、经济和较新的社会科学方面，现在我们能够测度我们原先认为不能测度的属性……。当然，这些测度并不象在物理科学中定义的一些测度那么完美……，但是它们确实存在并且往往基于它们可作出重要的决策。为了提高我们对特定实体的理解，我们有责任尝试去测度所谓不可测度的东西，这种责任在软件工程中和在其他学科中是一样重要。

但是有一些软件业人员继续争论道软件是不是可测度的或者说测度的尝试应该推迟到我们能更好地理解软件和用以描述软件的属性的时候。这是个错误的说法。

虽然计算机软件的技术度量并不是绝对的，但是它们为我们提供了基于一套清晰定义规则的一种系统的方法来评价质量。它们同时还为软件工程提供了一种现场的而不是事后的洞察，这使得工程师能在潜在的问题变成灾难性错误前发现和纠正它们。

在第 4 章我们讨论了运用在过程和项目级别的软件度量。在这一章，我们的重点转移到当产品被工程化的时候，可以用来评价产品的质量的度量。这些产品内部属性的测度为软件工程提供了对分析、设计和编码模型的有效性；测试用例的有效性以及要开发的软件整体质量的一个实时的指示。

18.1 软件质量

甚至最老练的软件开发者都会同意高质量的软件是一个重要的目标，但是我们如何定义质量呢？一句谐趣话曾经这样说道，“每一个程序能正确地做某件事，但是这并不是我们想要它做的事情。”

在第 8 章我们提出了许多不同的方法来看待软件质量并介绍了一个定义，它强调了与清晰描述的功能和性能需求的符合性、明显的文档的开发标准、以及被认为是所有专业开发的软件所应具备的隐式特征。

毫无疑问上述定义可以被无休止地修改、扩展或讨论。针对本书的目的，定义强调了以下三个重点：

1.软件需求是质量测度的基础。需求符合性的缺乏也就是缺乏质量。[①]

2.特定的标准定义了一套开发标准，用以指导软件开发的方式。如果标准未能够遵守，那么缺少质量就几乎是肯定的结论。

3.要有一套经常未被提及的隐式需求(例如，对好的可维护性的期望)。如果软件符合其显式的需求，但是未能满足隐式需求，软件质量仍是值得怀疑的。

软件质量是一个多因素的复杂混合，这些因素随着不同的应用和需要它们的用户而变化。以下章节标识了软件质量因素，以及描述了用以获取它们的人类活动。

18.1.1 McCall 的质量因素

影响软件质量的因素可以分为两大类：(1)可以直接测度的因素(例如，每个功能点的错误)和(2)只能间接测度的因素(例如，可用性和可维护性)。在每种情况下测度都必须发生。我们必须对软件(文档、程序、数据)和一些数据作一些比较，并获得质量的指示。

McCall 和他的同事［MCC77］提出了对影响软件质量的因素的有用的分类。这些软件质量因素，如图 18－1 所示，集中在软件产品的三个重要方面：它的操作特性、它承受改变的能力、以及对新环境的适应能力。

对于在图 18－1 提到的因素，McCall 提供了如下的描述：

正确性。一个程序满足它的需求规约和实现用户任务目标的程度。

可靠性。一个程序期望以所需的精确度完成它的预期功能的程度。更完整的对可靠性的定义已经在前面提出过了(见第 8 章)。

功效。一个程序完成其功能所需的计算资源和代码的数量。

完整性。对未授权人员访问软件或数据的可控制程度。

可用性。学习、操作、准备输入和解释程序输出所需的工作量。

可维护性。定位和修复程序中一个错误所需的工作量。(这是一个十分局限的定义。)

灵活性。修改一个运作的程序所需的工作量。

可测试性。测试一个程序以确保它完成所期望的功能所需的工作量。

可移植性。把一个程序从一个硬件和/或软件系统环境移植到另一个环境所需的工作量。

可复用性。一个程序［或一个程序的一部分］可以在另外一个应用程序中复用的程度——这和程序完成的功能的包装和范围相关。

互操作性。连接一个系统和另一个系统所需的工作量。

很难，在一些情况下也不可能，去开发一个对以上的质量因素的直接测度，因此，定义一组度量，并被用于按照下面的关系为每个因素开发表达式：

$$F_q = c_1 \times m_1 + c_2 \times m_2 + \cdots + c_n \times m_n$$

这里F_q是一个软件质量因素，c_n是回归系数，m_n是影响质量因素的度量值。不幸的是许多McCall定义的度量值只能主观地测度。度量可以用检查表的形式，来给软件的特定属性进行评分［CAV78］。由McCall提出的评分方案是从0(低)到10(高)的范围。以下是用在评分方案中的度量：

能听度(audibility)。和标准的符合性可被检查的容易程度。

准确度(accuracy)。计算和控制的准确度。

通讯公用度(communication commonality)。标准界面、协议和带宽的使用程度。

完全性(completeness)。所需功能完全实现的程度。

简洁度(conciseness)。以代码行数来评价程序的简洁程度。

一致性(consistency)。在软件开发项目中一致的设计和文档技术的使用。

数据公用性(data commonality)。在整个程序中对标准数据结构和类型的使用。

容错度(error tolerance)。当程序遇到错误时所造成的损失。

执行效率(execution efficiency)。一个程序的运行性能。

可扩展性(expandability)。结构、数据或过程设计可被扩展的程度。通用性(generality)。程序构件潜在的应用广度。

硬件独立性(hardware independence)。软件独立于其运行之上的硬件的程度。

检自性(instrumentation)。程序监视它自身的操作并且标识产生的错误的程度。

模快性(modularity)。程序部件的功能独立性(第13章)。

可操作性(operability)。程序操作的容易度。

安全性(security)。控制和保护程序和数据的机制的可用度。

自包含文档度(self－documentation)。源代码提供有意义的文档程度。

简单性(simplicity)。一个程序可以没有困难的被理解的程度。

软件系统独立性(software system independence)。程序独立于非标准编程特性、操作系统特性、和其他环境限制的程度。

可追溯性(traceability)。从一个设计表示或实际程序部件追溯到需求说明的能力。

可培训性(training)。软件支持使得新用户使用系统的能力。

软件质量因素的关系和上述的度量在图 18－2 中有所描述。值得注意的是分给每个度量的权值依赖于本地的产品和考虑。

18.1.2 FURPS

McCall 和他的同事提出的质量因素 [MCC77] 代表了被提出的众多软件质量“检查表”之一。Hewlett－Packard [GRA87] 提出了一套考虑软件质量的因素，简称为 FURPS——功能性(functionality)、可用性(usability)、可靠性(reliability)、性能(performance)、和支持度(supportability)。FURPS 质量因素是从早期工作中的得出的，五个主要因素每一个都定义了如下评估方式：

• 功能性：通过评价特征集和程序的能力、交付的函数的通用性、和整体系统的安全性来评估。

• 可用性：通过考虑人的因素(第 14 章)、整体美学、一致性、和文档来评估。

• 可靠性：通过测度错误的频率和严重程度、输出结果的准确度、平均失效间隔时间(MTBF)、从失效恢复的能力、程序的可预测性等来评估。

• 性能：通过测度处理速度、响应时间、资源消耗、吞吐量、和效率来评估。

• 支持度：包括扩展程序的能力(可扩展性)、可适应性、和服务性(这三个属性代表了一个更一般的概念——可维护性)，以及可测试性、兼容度、可配置性[组织和控制软件配置的元素的能力(第 9 章)]、一个系统可以被安装的容易程度、问题可以被局部化的容易程度。

FURPS 质量因素和上述描述的属性可以用来为软件过程中的每个活动建立质量度量。

18.1.3 到量化视图的变迁

在前面的章节里，讨论了一套软件质量测度定性因素。我们设法开发精确的软件质量的测度，但有时又会被活动的主观性质所困惑。Cavano 和 McCall[CAV78]讨论了该情形：

决定质量在日常事件(葡萄酒品尝比赛、运动赛事[例如体操]、智力竞赛等等)中是一个关键因素。在这些情形下，质量是以最基本和最直接的方式来判定的：在相同的条件和预先决定的概念下并列对比物体。葡萄酒可以根据清澈度、颜色、花束、味道等。但是，这种类型的判定是十分主观的；为了最终得到某一个值，它必须由一个专家来判定。

主观性和特殊性同样应用于确定软件质量。为了帮助解决这个问题，一个对软件质量更为精确的定义是必需的，同样，为了客观的分析，需要一个方法来导出软件质量的定量测度…，因为这样一件事情不是绝对知识的，不能期望去很精确地测度软件质量，因为每一个测度方法都是不完美的。Jacob Bronkowski 这样来描述这个知识的矛盾：“年复一年，我们设计更为精确的仪器来更好地观察自然，而当我们看到观察资料时，我们十分沮丧地看到它们仍然很模糊，并且我们感觉到它们和以往一样仍然不确定。

在接下来的章节里，我们检查了一组软件度量，它们可以应用到软件质量的定量评价。在所有的场合里，度量代表着间接测度；也就是说，我们从来没有真正地测度质量，而是测度一些质量的表现。复杂的因素在于所测度的变量和软件质量间的准确关系。

18.2 软件技术度量框架

如我们在这章的介绍中所说的，测度分配数字或符号给现实世界中的实体的属性。为了达到这个目的，需要一个包含一组一致规则的测度模型。尽管测度理论(例如，[KYB94]及其在计算机软件的应用(例如参考文献[DEM81]和[BRI96])这些话题不在本书的内容范围，但是，仍然值得去建立一个基本框架和一组软件的技术度量的测度的基本原则。

18.2.1 技术度量的挑战

在过去二十年中，许多研究者尝试着开发能提供软件复杂度的全面测度的单一度量。Fenton [FEN94] 把这种研究看成是对“不可能的圣杯”的搜寻。尽管已经提出了很多的复杂度测度 [ZUS90] ，但是每一种都对复杂度是什么以及是什么系统属性导致的复杂性持有不同的看法。类比而言，考虑一个评价有吸引力的汽车的度量，一些观察者可能强调车身的设计，另外一些会考虑机械特性，而有人会考虑成本、性能或燃料经济性、或当汽车需丢弃时的回收能力。因为这些特性中的任意一个都有可能和其他的产生不一致，这样很难给吸引力一个单一的值。对计算机软件而言也会发生同样的问题。

但是，仍然有必要去测度和控制软件复杂度。并且如果这个“质量度量”的单一值难以获取的话，应该有可能去开发不同程序内部属性的度量(例如，有效模块度、功能独立性和在 13 章讨论的其他属性)，这些测度和从它们导出的度量可被用作分析和设计模型的质量的独立指示。但是，在这里又出现了问题，Fenton［FEN94］这样说道：

尝试去寻找标识那么多不同属性的测度的危险是度量不可避免地不得不满足有冲突的目标。这是和测度的代表性理论相冲突的。

尽管 Fenton 的论述是正确的，许多人争论道在软件过程的早期阶段采取的技术测度给软件工程提供了评估质量的一个一致和客观的机制。

但是，询问技术度量有多么正确是很合理的。也就是说，度量和基于计算机的系统的长期的可靠性及质量的符合程度有多少？ Fenton[FEN91]用下面的方法解决了这个问题：

尽管软件产品的内部结构[技术度量]和它的外部产品和过程属性间存在直觉的联系，实际上，仍然几乎没有科学的尝试来建立特定的关系。为什么这样有许多原因；最普遍的说法是进行相关实验是不实际的。

上述的每一个挑战都是一个值得警惕的原因，但是并不是摒弃技术度量的原因。[①]如果要获得质量，那么测度是很重要的。

18.2.2 测度原则

前面我们介绍了一系列的技术度量，它们(1)辅助评价分析和设计模型，(2)提供了过程设计和源代码的复杂度指示，以及(3)辅助设计更为有效的测试。理解基本的测度原则是很重要的，Roche[ROC94]建议了一个测度过程，它以下面五个活动为特征：

- 简洁表示——导出适合于所考虑软件的表示的软件测度和度量。
- 收集——用以积累导出简洁度量所需的数据的机制。
- 分析——计算度量值且应用数学工具。
- 解释——为了获得对所表示的质量的洞察，对度量结果进行评价。
- 反馈——把对技术度量的解释获得的建议递交给软件队伍。

可以和技术度量的表示相关联的原则如下(见参考文献[ROC94])：

- 应该在数据收集开始前确定测度的目标。
- 每一个技术度量应该以无二义的方式进行定义。

• 度量应该基于应用领域是正确的理论之上而导出(例如，设计度量应该基于基本的设计概念和原则而导出，并且设法提供被认为是需要的属性存在的指示)。

• 度量应该被剪裁以最适应特定的产品和过程(见参考文献[BAS84])。

尽管简洁表示是一个关键的出发点，然而收集和分析是推进测度过程的活动。Roche(见参考文献[ROC94])针对这些活动建议了以下原则：

• 任何时候应尽可能使得收集和分析自动化。

• 应该应用正确的统计技术来建立内部产品属性和外部质量特性的关系(例如，结构复杂度层次和产品使用中报告的错误数是不是有关联？)。

• 应该给每个度量建立解释性的指南和建议。

除上面提到的原则以外，度量活动的成功也和管理支持紧密相关，如果要建立和维持一个技术度量计划，资金、培训和能力提高均应该加以考虑。

18. 2. 3 有效软件度量的属性

对计算机软件，已经提出过有几百个度量，但是，并不是所有的都对软件工程有实际的支持，有些度量太复杂了，其他一些太深奥以至于很少现实世界的专业人员能理解它们，另外一些则违反了高质量软件的实际的基本直觉概念。

Ejiogu [EJI91] 定义了一组应该由有效软件度量包含的属性，导出的度量及导致它的测度如下：

• 简单的和可计算的。学习如何导出度量值应该是相对简单的，并且它的计算不应该要求过多的工作量和时间。

• 经验和直觉上有说服力。度量应该满足工程师对于所考虑的产品的直觉概念(例如，一个测度模块内聚性的度量值应该随着内聚度的提高而提高)。

• 一致的和客观的。度量应该总是产生非二义性的结果。一个独立的第三方使用该软件的相同信息能够得到相同的度量值。

• 在其单位和维度的使用上是一致的。度量的数学计算应该使用不会导致奇异单位组合的测度。例如，把项目队伍的人员乘以程序中的编程语言的变量会引起一个直觉上没有说服力的单位组合。

• 编程语言独立的。度量应该基于分析模型、设计模型、或程序本身的结构。它们不应该依赖于不同的编程语言的句法和语法。

• 质量反馈的有效机制。度量应该给软件工程师提供能导致更高质量的最终产品信息。

尽管许多软件度量都满足上述的属性，一些普遍应用的度量也可能会不满足其中一到两个属性。例如对功能点方法(在第 4 章讨论过，这章又要再讨论)，可以争辩[①]说一致性和客观性属性不满足，因为一个独立的第三方不可能获得和一个同事用相同的软件信息得到的同样的功能点。难道我们因此就拒绝使用功能点度量吗？回答是“当然不！”即使它不能很好地满足一个属性，功能点却提供了有用的洞察方法，且提供了清晰的值。

18. 3 分析模型的度量

软件工程的技术性工作开始于分析模型的创建[①]。在这个阶段可以导出需求分析，并建立设计的基础，所以，提供了对分析模型质量的洞察的技术度量是有必要的。

尽管在文献里很少出现分析和规约度量，但是仍然有可能把针对项目应用导出的度量作适应性修改后用于这个语境中。这些度量以预测结果系统的大小为目的，来检查分析模型，有可能大小和设计复杂度将被直接相关联。

18.3.1 基于功能的度量

功能点(FP)度量(第 4 章)可以用来作为预测从分析模型得到的系统大小的手段。为了说明FP度量在该语境的使用，我们考虑一个简单的分析模型，见图18－3，在图中描述了SafeHome软件[②]的一个功能数据流图(第 12 章)，该功能管理用户交互，接收一个用户密码来启动或关闭系统，并且允许对安全区状态和不同安全传感器进行查询。该功能显示了一系列的提示信息且发送合适的控制信号到安全系统的不同部件。

为了确定用以计算功能点度量所需的关键测度，对数据流图加以评估(第 4 章)：

• 用户输入数。

• 用户输出数。

• 用户查询数。

• 文件数。

• 外部接口数。

三个用户输入：密码、莫名奇妙的按键、和激活/非活动在图中有所显示，另外还有两个查询：零查询和传感器查询。还显示有一个文件(系统配置文件)。

还有两个用户输出(信息和传感器状态)和四个外部接口(测试传感器、零设置、激活/非激活、及报警警报)。这些数据以及合适的复杂度在图 18－3 中显示。

在图 18－4 中显示的总计数必须用公式(4.1)调整：

$$FP=总计数\times(0.65+0.01\times\Sigma F_i)$$

这里总计数是所有从图 18－3 中获得的FP项的总和，F_i(i＝1 到 14)是“复杂度调整值。”对于这个例子，我们假设ΣF_i是 46(一个适度复杂的产品)，所以，

$$FP=50\times[0.65+(0.01\times46)]=56$$

基于从分析模型得到的项目 FP 值，项目队伍可以估计 SafeHome 用户交互功能的整体实现后的大小。假设过去的数据表明一个 FP 转换成 60 行源代码(使用面向对象语言)且每个人月的工作量产生 12FP，这些历史数据给项目经理提供了基于分析模型而不是初步估计的重要的计划信息。

18.3.2 “撞击值”度量

象功能点度量一样，bang度量可以由分析模型得到对将要实现的软件的大小的指示。“撞击值”(bang)度量由Tom DeMarco [DEM892] 提出，它是“一个实现独立的系统大小的指示。”为了独立计算bang，软件工程师必须首先评价一组原语——在分析层次不能再划分了的分析模型的元素。原语是通过评价分析模型和开发以下项的计数来决定的[①]：

功能原语(FuP)。在数据流图(第 12 章)中最低层次的变换(泡泡)。

数据元素(DE)。数据对象的属性，数据元素不是复合数据且在数据字典里出现。

对象(OB)。在第 11 章里描述的数据对象。

关系(RE)。在第 12 章里描述的数据对象间的联系。

状态(ST)。在状态变迁图(第 12 章)中用户可观察的状态数量。

变迁(TR)。在状态变迁图(第 12 章)中用户状态变迁的数量。

除了上述的六个原语，另外如下的计数也需确定：

修改的手工功能原语(FuPM)。在系统边界之外且必须为了适应新系统而必须修改的功能。

输入数据元素(DEI)。输入到系统的数据元素。

输出数据元素(DEO)。从系统输出的数据元素。

存储数据元素(DER)。被系统存储的数据元素。

数据记号(TCi)。存在第 I 个功能原语(为每一个原语评价)的边界上的数据记号(在一个功能原语内不能再分割的数据项)。

关系连接(REi)。在数据模型中连接第 i 个对象和其他的对象的关系。

DeMarco[DEM82] 建议大多数软件可以划分为以下两个领域之一，功能很强型或数据复杂型，这依赖于比率 RE/FuP。功能很强型应用程序(一般在工程和科学应用程序中遇到的多)强调数据的变换且通常没有复杂的数据结构。数据复杂型的应用程序(一般在信息系统应用程序中遇到的多)往往有复杂的数据模型。

RE/FuP＜0.7 意味着一个功能很强型应用程序

0.8＜RE/FuP＜1.4 意味着混合型应用程序

RE/FuP＞1.5 意味着数据复杂型应用程序

因为不同的分析模型将模型分成或大或小的细化程度，DeMarco 建议了一个对每个原语的平均记号计数

$$TC_{avg} = \Sigma TCi/FuP$$

它被用来控制在某应用程序领域中跨越很多不同模型划分的一致性。

为了计算功能很强型应用软件的“撞击值”，可以用以下算法：

```
set initial value of bang＝0;
do while functional primitives remain to be evaluated
compute token－count around the boundary of primitive i;
compute corrected FuP increment(CFuPI);
Allocate primitive to class;
Assess class and note assessed weight;
Multiply CFuPI by the assessed weight;
bang＝bang＋weighted CFuPI;
enddo
```

通过确定在原语中有多少分离的记号是“可见的”[DEM82]来计算记号计数。如果数据元素可以没有任何内部变换地从输入移到输出，记号的数目和数据元素的数目不相同是有可能的。纠正的 CFuPI 可以从 DeMarco 公开的表中得出。一个节选版本如下所示：

Tc_i	CFuPI
2	1.0
5	5.8
10	16.6
15	29.3
20	43.2

上述算法提到的评价加权由 DeMarco 定义的 16 个不同的功能原语类导出。一个权值依赖于原语的类别分配，其范围从 0.6(简单数据路由)到 2.5(数据管理功能)。

对于数据复杂型应用程序，撞击值用以下算法来计算：

```
set initial value of bang＝0;
do while objects remain to be evaluated in the data model;
compute count of relationships for object i;
compute corrected OB increment(COBI);
bang＝bang＋COBI;
enddo
```

纠正的 COBI 也可以从 DeMarco 公布的表格确定，一个节选的版本如下所示：

RE_i	COBI
1	1.0
3	4.0
6	9.8

一旦撞击值计算后，过去的历史可以把它和大小和工作量关联起来。DeMarco建议一个组织建立它自己的CFuPI和COBI表版本，以用于从已完成软件项目来校准信息。

18.3.3 规约质量的度量

Davis及其同事[DAV93]提出了一列可以用来评价分析模型和相应的需求规约质量的特征：明确性(无二义性)、完全性、正确性、可理解性、可验证性、内部和外部一致性、可完成性、简洁性、可追踪性、可修改性、精确性和可复用性。此外，作者还提到高质量的规约是电子存储的、可执行的或至少可解释的、对相对重要性和稳定性进行注释的、并在适当的详细级别提供版本化、组织、交叉引用和表示。

尽管上述的许多特性在性质上看起来象是定性的，Davis et al.[DAV93]建议每一个可以用一到多个度量来表示。[①]例如，我们假设在一个规约中有n_r个需求，所以

$$n_r=n_f+n_{nf}$$

其中n_f是功能需求的数目，n_{nf}是非功能需求数目(例如，性能)。

为了确定需求的确定性(无二义性)，Davis et al.建议了一种基于复审者对每个需求的解释的一致性的度量方法：

$$Q_1=n_{ui}/n_r$$

其中n_{ui}是所有复审者都有相同解释的需求数目。当需求的模糊性越低时，Q的值越接近1。

功能需求的完整性可以通过计算以下比率获得：

$$Q_2=n_u/(n_i\times n_s)$$

其中n_u是唯一功能需求的数目，n_i是由规约定义或包含的输入(刺激)的个数，n_s是被表示的状态的个数。Q_2比率测度了一个系统所表示的必需的功能百分比，但是它并没有考虑非功能需求，为了把这些非功能需求结合到整体度量中以求完整，我们必须考虑已经需求已经被确认的程度。

$$Q_3=n_c/(n_c+n_{nv})$$

其中n_c是已经确认为正确的需求的个数，n_{nv}是尚未被确认的需求的个数。

18.4 设计模型的度量

很难想象一个新的飞机、一个新的计算机芯片、或一个新的办公楼的设计可以在没有定义设计测度、确定设计质量的各个方面的度量、和使用它们来指导设计演化方式的情况下开始进行。然而，基于复杂软件的系统设计实际上经常在没有测度的情况下进行。有讽刺意义的是软件的设计度量是可以获得的，但是很大部分的软件工程师却继续忽视它的存在。

和所有其他的软件度量一样，计算机软件的设计度量并不是完美的。对于它们的功效和它们应该如何应用的争论一直在继续。许多专家争论道在设计测度可以使用之前需要进一步的试验，但是，没有测度的设计是一个难以接受的选择。

在接下来的章节里我们讨论一些计算机软件更为常见的设计度量，尽管其中没有一个是完美的，但是所有的都给设计者提供了改善的洞察方式，且所有的都可以帮助设计演化到一个更高质量的水平。

18.4.1 高层设计度量

高层次的设计度量集中于程序体系结构(第 14 章)的特征上，它强调了体系结构上的结构和模块的有效性，这些度量在某种意义上是黑盒的，它们不需要系统某一个特定模块的内部运作的知识。

Card 和 Glass [CAR90] 定义了三个软件设计复杂度测度：结构复杂度、数据复杂度和系统复杂度。一个模块 i 的结构复杂度，S(i)，按如下方式定义：

$$S(i)=f^2_{out}(i) \qquad (18.1)$$

其中$f_{out}(i)$是模块i的扇出[①]。

数据复杂度，D(i)，提供了一个模块 i 的内部接口的复杂度的指示，定义如下：

$$D(i)=v(i)/[f_{out}(i)+1] \qquad (18.2)$$

其中 V(i)是传入传出模块 i 的输入输出变量的个数。

最后，系统复杂度，C(i)，定义为结构复杂度和数据复杂度的总和，如下定义：

$$C(iO=S(i)+D(i) \qquad (18.3)$$

当其中的任何一个复杂度的值增大，整体的系统体系结构复杂度也随着提高，这导致集成和测试开销也跟着上升的可能性增大。

由 Henry 和 Kafura(见参考文献 [HEN81])提出的一个早期高层次的体系结构设计度量也使用了扇入扇出。作者定义了如下形式的复杂性度量：

$$HKM=length(i)\times[f_{in}(i)+f_{out}(i)]^2 \qquad (18.4)$$

其中length(i)是在模块i中编程语言语句的数目，$f_{in}(i)$是模块i的扇入。Henry和Kafura扩展了在本书的扇入扇出概念，包括了不仅是模块控制连接(模块调用)，而且还包括当模块i所读取(扇入)或更新(扇出)另外模块的数据时，这些模块的数目。为了在设计中计算HKM，必须用过程设计来估计模块i的编程语言语句的数目。如同Card和Glass上面提及的度量一样，在Henry－Kafura度量值的增大时会导致模块集成和测试开销也跟着上升的可能性也增大。

Fenton [FEN91] 建议了一些简单的形态(例如，外形)度量，使得不同的程序体系结构可以用一套简单的维度来加以比较。在图 18－5 中，可以定义下面的度量：

size=n+a

其中 n 是节点(模块)的数目，a 是弧线(控制线)的数目。对于图 18-5 所示的体系结构，

size=17+18=35

深度=从根节点到页节点的最长路径。对于图 18-5 所示的体系结构，深度=4。

广度=在体系结构的任意一层的最大节点数。对于图 18-5 所示的体系结构，广度=6。

弧和节点的比率，r=a/n，

测度了体系结构的连接密度且对体系结构的耦合提供了一个简单的指示。对于图 18-5 所示的体系结构，r=18/17=1.06。

美国空军系统司令部[USA87]基于计算机程序的可测度设计特性开发了一组软件质量指示。空军使用了类似于在 IEEE Std.982.1-1988[IEEE94]中提出的概念，使用了从数据和体系结构设计中获得的信息而导出了一个范围从 0 到 1 的设计结构质量指标(DSQI)。为了计算 DSQI，以下值必须要获得[CHA89]：

S_1=在程序体系结构中定义的模块总数

S_2=其正确功能依赖于数据输入源或产生在其他地方使用的数据的模块数[通常，控制模块(在其他模块之中)将不会被计为S_2的一部分]

S_3=其正确功能依赖于前导处理的模块数

S_4=数据库中的项目数(包括数据对象和所有定义对象的属性)

S_5=独特的数据库项目的总数

S_6=数据库段的数目(不同的记录或单个对象)

S_7=有单个入口和出口的模块数目(异常处理不被看作是多重出口)

一旦一个计算机程序的S_1到S_7的值确定后，以下中间值可以计算出来：

程序结构：D_1，其中D_1如下定义：如果体系结构设计是用一个明显的方法(例如，面向数据流设计或面向对象设计)来开发的，那么D_1=1，否则D_1=0。

模块独立性：$D_2=1-(S_2/S_1)$

模块不依赖于前导处理：$D_3=1-(S_3/S_1)$

数据库大小：$D_4=1-(S_5/S_4)$

数据库分区：$D_5=1-(S_6/S_4)$

模块出/入口特性：$D_6=1-(S_7/S_1)$

当这些中间值确定后，DSQI 以下面的方式来计算：

$$DSQI=\sum w_i D_i \qquad (18.5)$$

其中i=1 到 6，w_i是每个中间值的重要性的相对权值，$\sum w_i=1$(如果所有D_i平均地加权，那么w_i=0.167)。

过去设计的 DSQI 值可以确定下来并和目前正在开发的设计相比较。如果 DSQI 明显低于平均值，意味着需要进一步的设计和考虑，同样，如果将要对一个现存的设计做重要的改动，这些改动对 DSQI 的影响也可以被计算出来。

18.4.2 构件级设计度量

构件级的设计度量集中于软件构件的内部特性且包括模块内聚、耦合和复杂度的度量。这三个属性可以帮助软件工程师判定一个构件级的设计。

在本章节讨论的度量在某种意义上是玻璃盒，它需要所考虑的模块的内部运作知识。一旦一个过程设计已经开发完，构件级设计度量就可以被应用。另外，他们也可以延迟到有源代码时才用。

内聚度量 Bieman 和 Ott[BIE94]定义了一组给模块内聚性(第 13 章)提供指示的度量。该度量以五个概念和测度来定义：

数据 0 片。简单的说，一个数据片是一个对模块进行回溯跟踪检查，找到的在影响可查开始处的模块位置的数据值。值得一提的是程序片(主要集中在语句和条件)和数据片都可以加以定义。

数据表征。为模块定义的变量可以被定义为模块的数据表征。

胶合表征。位于一个或多个数据片的数据表征集合。

超胶合表征。在一个模块里每个数据片都公有的数据表征。

粘度。一个胶合表征的相对粘度是和它所绑定的数据片的数目直接成比例。

Bieman 和 Ott 开发了强功能内聚(SFC)、弱功能内聚(WFC)、以及附着性(胶合表示号绑定数据片的相对程度)的度量，这些度量可以用下面方式来解释[BIE94]：

所有这些内聚度量值范围在 0 到 1 之间。当一个过程有多于一个输出且没有展示任何由某一特定的度量指示的内聚属性时具有值 0。没有超胶合表征，也就是没有一个为所有数据片所公有的表征的过程具有零强功能内聚——即没有对所有输出有贡献的数据表征。一个没有胶合表征的过程，即没有对多于一个数据片公有的表征(在具有多于一个的数据片的过程中)，展示了零弱功能内聚和零附着性——即没有对多于一个输出有贡献的数据表征。

当 Bieman 和 Ott 度量取得最大值 1 时，出现了强功能内聚和附着性。

一个详细的关于 Bieman 和 Ott 度量的讨论最好是留给作者[BIE94]，但是，为了解释这些度量的特性，考虑强功能内聚的度量：

$$SFC(i)=SG(SA(i))/tokens(i) \quad (18.6)$$

其中 SG(SA(i))指超胶合表征——位于一个模块 i 的所有数据片的数据表征集合。当超胶合表征比上模块 i 中的所有的表征的总和上升到最大值 1 的时候，模块的功能内聚也增加。

耦合度量。模块耦合提供了一个对一个模块与其他模块、全局数据、和外部环境的连接的指示。在第 13 章，耦合是以定性的角度讨论的。

Dhama[DHA95]提出了一个包含数据和控制流耦合、全局耦合、和环境耦合的对模块的度量。计算模块耦合需要的度量按上述的三种耦合类型的每一种来定义。

对数据和控制流耦合：

d_i=输入数据参数的个数

c_i=输入控制参数的个数

d_o=输出数据参数的个数

c_o=输出控制参数的个数

对全局耦合：

g_d=用作数据的全局变量的个数

g_c=用作控制的全局变量的个数

对环境耦合：

w=被调用模块的个数(扇出)

r=调用所考虑的模块的模块数(扇入)

使用这些度量，一个模块耦合指示器，m_c，以下面方式定义：

$$m_c=k/M$$

其中k=1，一个比例常数。[①]

$$M=d_i+a\times c_i+d_0+b\times c_0+g_d+c\times g_c+w+r$$

其中

m_c的值越高，整体模块耦合越低。例如，如果一个模块有一个单一的输入和输出数据参数，没有访问全局数据，且只被一个单一的模块调用：

$$m_c=1/(1+0+1+0+0+0+1+0)=1/3=0.33$$

我们将预计到这样一个模块将有低耦合，所以一个m_c=0.33 的值意味着低耦合。另外，如果一个模块有 5 个输入和 5 个输出数据参数，一个相等数目的控制参数，访问了 10 个条目的全局数据，且有 3 个扇入和 4 个扇出，

$$m_c=1/(5+2\times 5+5+2\times 5+10+0+3+4)=0.02$$

其意味着高耦合。

为了让耦合度量随着耦合度的上升而上升，一个改进的耦合度量定义成：

$$C=1-m_c$$

其中耦合度在最小值 0.66 到接近 1 的最大值之间非线性上升。

复杂度度量一系列软件度量可以计算出来以确定程序控制流的复杂度，其中许多是基于一个称做流图的表示形式来计算的。如我们在第 16 章所讨论的一样，一个图是由节点和链接(也称为边)构成的表示形式，当链接(边)是有方向时，流图是一个有向图。

McCabe[MCC94]指出了复杂度度量的一系列重要应用：

复杂度度量可以用来预计关于从源代码[或过程设计信息]的自动分析得到的软件系统的可靠性和可维护性。复杂度度量同时也在软件项目中提供反馈以帮助控制[设计活动]。在测试和维护中，他们提供了模块的详细信息以帮助指出潜在的不稳定的地方。

计算机软件最广泛使用的(且最广泛被讨论的)复杂度度量是环形计数复杂度，它最先是由 Tomas McCabe[MCC76，MCC89]提出，且在 16.4.2 节里有详细讨论。

McCabe 度量提供了对测试困难度和最终可靠性标识提供了一个定量的度量。试验研究表明 McCabe 度量和在源代码中存在的错误数、以及需要发现和纠正这些错误的时间是有很强关联的。

McCabe 也提出环形计数复杂度也可以用来提供一个最大模块大小规模的定量指示。通过从许多实际的编程项目中收集数据，他发现值为 10 的环形计数复杂度似乎是一个实际的模块大小的上限。当模块的环形计数复杂度超过这个数，要充分测试一个模块就变得特别难。把环形计数复杂度作为对白盒测试用例设计的指导可见第 16 章。

Zuse[ZUS90]给出了一个对不少于 18 种不同的软件复杂度度量的全面讨论。作者对每一个种类中的度量给出了基本定义(例如，有许多环形计数复杂度度量的变种)且对每一个都进行了分析和评论。Zuse 的工作是至今为止最为全面的。

18.4.3 界面设计度量

尽管在人机界面设计方面有许多重要的文献(见第 14 章)，但是，对提供对界面的质量和可用性洞察的度量的信息倒是相对较少。

Sear[SEA93]建议布局恰当性(layout appropriateness，LA)是一个人机界面有价值的设计度量。一个典型的 GUI 使用布局实体——图标、文本、菜单、窗口等——来辅助用户完成任务。为了使用 GUI 完成一个给定的任务，一个用户必须从一个布局实体移动到另一个布局实体，每一个布局实体的绝对和相对位置、及其被使用的频率、和从一个布局实体变迁到下一个布局实体的“开销”将影响界面的恰当性。

对于一个特定的布局(例如，一个特定的 GUI 设计)，根据下面的关系，开销可以分配给每个动作序列：

$$\text{开销}=\Sigma[\text{变迁频率}(k)\times\text{变迁开销}(k)] \quad (18.7)$$

其中 k 是当一个特定任务完成时，从一个布局实体到下一个实体的特定变迁。对完成某应用功能所需的一个特定任务或任务集的所有变迁形成一个总和。

开销可以用时间、处理延迟、或其他合理的值，例如一个鼠标在布局实体之间移动的距离，来标识。

布局恰当性定义成：

$$LA=100\times[(\text{LA 最优布局开销})/(\text{提出的布局开销})] \qquad (18.8)$$

其中对于一个最优布局 LA=100。

为了计算 GUI 的最优布局，界面区域(屏幕的区域)被分成方格，每个方格代表着一个布局实体可能的位置。对于一个有 N 个可能位置的且放置 K 个不同的布局实体的方格，可能的布局数以下面方式给出[SEA93]：

$$\text{可能的布局数}=[N!/(K!\times(N-K)!)]\times K! \qquad (18.9)$$

当布局位置数上升，可能的布局数增长得十分大。为了发现一个最优(最低开销)的布局，Sears[SEA93]提出了一个树搜索算法。

LA 用来评价不同的建议 GUI 布局和一个特定的布局对任务描述的变化的敏感度(例如，序列和/或变迁频率的改变)。界面设计者可以使用布局恰当性的改变，△LA，作为给一个特定应用选择最好的 GUI 布局的指导。

值得指出的是选择 GUI 设计可以用度量，例如 LA，来指导，但是最后的裁定应该是基于 GUI 原型的用户反馈。Nielsen 和 Levy[NIE94]说道“如果谁在仅仅从基于用户的观点的界面[设计]中选择，他就有相当大的机会获得成功。用户的平均任务性能和他们对一个 GUI 的主观满意程度是紧密相关的。”

18.5 源代码度量

Halstead的软件科学理论[HAL77]可能是“最著名的和最完全的…(软件)复杂度的复合度量”[CUR80]。软件科学提出了第一个计算机软件的分析。“定律”。[①]

软件科学把定量定律赋予了计算机软件的开发。Halstead 的理论是从一个基本的假设得来的[HAL77]：“人脑遵循一套比大家已知道的还要严格的规则…”，软件科学使用的一组基本测度可以在代码产生后或一旦设计完成后对代码进行估算得到，列举如。

n_1——在一个程序中出现的不同操作符

n_2——在一个程序中出现的不同操作数

N_1——操作符出现的总数

N_2——操作数出现的总数

为了解释这些初始的度量是如何获得的，可参见在图 18-6 所示的简单的排序程序[FIT78]。

Halstead 使用了基本测度来为以下几方面开发表达式：整体程序长度；一个算法的潜在的最小体积；实际体积(表示一个程序所需的位数)；程序层次(一种软件复杂度测度)；语言层次(针对某给定语言的一个常量)；和其他的特征如开发工作量、开发时间、甚至在软件中被计划的错误数。

Halstead 展示了长度 N 可以这样来估计：

$$N=n_1\log_2 n_1+n_2\log_2 n_2 \quad (18.10)$$

程序体积可以如下定义：

$$V=N\log_2(n_1+n_2) \quad (18.11)$$

值得提出的是 V 会随着编程语言的不同而不同，且它代表了写一个程序所需的信息量(以位数计)。对于图 18-6 所示的排序模块，可以看到 Fortran 版本的体积是 204[FIT78]。对于等价的汇编语言版本的体积是 328。如我们所预料的，用汇编语言来写程序要化更多的工作量。

理论上，一个特定的算法存在一个最小体积。Halstead 定义了一个体积比率 L，作为程序最简洁形式的体积比上实际程序的体积。实际上，L 一定总是小于 1。以基本测度而言，体积比率可以写成：

$$L=2/n_1\times n_2/N_2 \quad (18.12)$$

Halstead 提出每个语言都可以按语言层次 1 来分类，1 可以在语言中改变。Halstead 理论总结道，语言层次对于一个给定的语言是常量，但是其他的一些工作[ZEL81]指出语言层次是语言和编程者的函数。以下语言层次值已经由实践获得：

语言平均值	1
英语散文	2.16
PL	11.53
ALGOL	68 2.12
FORTRAN	1.14
汇编	0.88

看起来语言层次蕴含了在过程规约中的一个抽象层次。高层次语言允许代码规约比汇编语言(面向机器)具有更高层次的抽象。

Halstead 的工作易于接受试验的验证，而且大量的研究已经针对软件科学进行。对该工作的讨论已不在本书的讨论范围，但是，可以说在分析性预测和试验结果之间已经发现了很好的一致性。对于进一步的信息，见参考文献[ZUS90]和[FEN91]。

18.6 对测试的度量

尽管在软件测试度量方面已写有不少东西(例如参考文献[HET93])，但是提出的大部分度量都集中在测试的过程，而不是测试本身的技术特性。通常而言，测试者在测试用例的设计和执行上必须依赖于分析、设计和代码度量来指导他们。

基于功能的度量(18.3.1 节)可以用来作为整体测试工作量的指示器。以往项目的不同项目层次特性(例如，测试工作量和时间、未发现的错误、产生的测试用例的数目)可以收集起来且和项目队伍产生的 FP 数关联。队伍可以为将来的项目的这些特征预测“所期望”的值。

“撞击值”度量可以通过检查在 18.3.2 节讨论的基本测度而提供所需的测试用例数目的指示。功能原语(FuP)、数据元素(DE)、对象(OB)、关系(RE)、状态(ST)、和变迁(TR)可被用来为软件设计黑盒和白盒测试的数目和类型。例如，和人机界面关联的测试数可以通过检查以下项来估计：(1)包含在 HCI 的状态变迁表示图中的变迁(TR)的数目以及评价检测每个转换所需的测试，(2)穿过界面的数据对象(OB)数目，和(3)输入或输出的数据元素的数目。

高层次的设计度量提供了和集成测试(第 17 章)相关联的难易信息以及对专用的测试软件(例如，桩和驱动器)的需求。环形计数复杂度(一种构件设计度量)位于基本路径测试的核心，这是在第 16 章提出的测试用例设计方法，另外，环形计数复杂度还可以用来定位模块作为广泛的单元测试的候选(第 17 章)，环形计数复杂度高的模块可能比环形计数复杂度低的模块更易出错。由于这个原因，测试者应该在模块集成进系统之前花费超过平均工作量的时间来发现模块中的错误。测试工作量也可以使用 Halstead 度量(18.5 节)来估算。使用程序体积的定义 V、和程序层次 PL，软件科学工作量 e，可以如下计算：

$$PL=1/[(n_1/2)\times(N_2/n_2)] \qquad (18.13a)$$

$$e=V/PL \qquad (18.13b)$$

将被分配给模块 k 的整体测试工作量百分比可以用下面的关系来估算：

$$\text{测试工作量百分比}(k)=e(k)/\sum e(i)$$

其中 e(k)是用公式(18.13)为模块 k 计算的且在公式(18.14)的分母是系统所有模块的软件科学工作量的总和。

当测试进行时，有三个不同的测度提供了一个对测试完全性的指示。测试广度的测度提供了多少需求(在所有需求的数目中)已经被测试，这样给测试计划完全性提供了指示。测试深度是对被测试覆盖的独立基本路径占在程序中的基本路径的总数的百分比的测度，基本路径数目的相当精确的估算可以通过累加所有程序模块的环形计数复杂度而计算得到。最后，当测试进行时且错误数据收集起来时，收集的错误值可以用来对未发现的错误进行优先级和分类处理。优先级指明问题的严重性，错误类别提供了错误的描述以使得可以进行统计错误分析。

18.7 对维护的度量

所有在本章介绍的度量可以用来开发新的软件和维护现存的软件，但是，人们也提出了明确针对维护活动设计的度量。

IEEE Std.982.1-1988[IEE94]建议了一个软件成熟度指标(SMI)，它提供了对软件产品的稳定性的指示(基于为每一个产品的发布而做的变动)，以下信息可以确定：

M_T=当前发布中的模块数

F_c=当前发布中已经变动的模块数

F_a=当前发布中已经增加的模块数

F_d=当前发布中已删除的前一发布中的模块数

软件成熟度指标以下面的方式计算：

$$SMI=[M_T-(F_a+F_c+F_d)]/M_T \qquad (18.15)$$

当 SMI 接近 1.0 的时候，产品开始稳定。SMI 也可以用作计划软件维护活动的度量。产生一个软件产品的发布的平均时间可以和 SMI 关联起来，且也可以开发一个维护工作量的经验模型。

18.8 小结

软件度量提供了一个定量的方法来评价产品内部属性的质量，因此可以使得软件工程师能够在产品完成之前进行质量评估。度量提供了创建有效的分析和设计模型、可靠的代码、和完全的测试必需的洞察。

为了在现实世界环境中有用，一个软件度量必须简单和可计算、有说服力、以及一致和客观。它应该是独立于编程语言的，且给软件工程师提供了有效的反馈。

分析模型的度量集中于功能、数据和行为——分析模型的三个元素。功能点和撞击击度量各自给评价分析模型提供了定量的方法。设计度量考虑了高层次、构件层次、和界面设计问题。高层次设计度量考虑了设计模型的体系结构和结构方面。构件层次设计度量通过建立内聚、耦合、和复杂度的间接度量提供了模块质量的指示。界面设计度量给 GUI 的布局恰当性提供了指示。

软件科学提供了一组在源代码级别的令人感兴趣的度量。使用在代码中出现的操作符和操作数，软件科学提供了可以用来评价程序质量的各种度量。

很少有技术度量提出来是直接用于软件测试和维护的，但是，许多其他的技术度量可以用来指导测试过程和作为一种机制来评价计算机程序的可维护性。

思考题

18.1 测度理论是一个对软件度量有重大意义的高级话题。使用参考文献[FEN91]、[ZUS90]和[KYB84]，或其他信息来源，写一篇短论文概括测度理论的主要原则。个人项目：写一个关于这个主题的介绍，并在班上进行交流。

18.2 McCall 的质量因素是在七十年代提出的。自从它们被提出来后几乎计算的每个方面改动都很大，但是，McCall 的因素继续应用到现代软件。你能根据这个事实得出什么结论吗？

18.3 为什么没有一个单一的、全包容的对程序复杂度或程序质量的度量？

18.4 复审你作为思考题 12.3 建立的分析模型。使用在 18.3.1 节提出的指南，对和 PHTRS 相关联的功能点数作一个估算。

18.5 复审你作为思考题 12.3 建立的分析模型。使用在 18.3.2 节提出的指导原则，对撞击值度量开发一个基本计数。请问 PHTRS 系统是功能很强型的还是数据复杂型的？

18.6 使用你在思考题 18.5 中得到的测度来计算撞击值度量的值。

18.7 为你老师提出的系统创建一个完整的设计模型。使用在 18.4.1 节描述的度量来计算结构和数据复杂度。同时也为设计模型计算 Henry-Kafura 和形态度量值。

18.8 一个主要的信息系统有 1140 个模块，其中有 96 个模块执行控制和协调功能，490 个模块其功能依赖于前导处理，系统处理大约 220 个数据对象，每个对象平均有三个属性，有 140 个独特的数据库条目和 90 个不同的数据库，600 个模块有单一的入口和出口点。为这个系统计算 DSQI 值。

18.9 研究 Bieman 和 Ott[BIE94]的论文并设计一个完整的例子来解释它们内聚度量的计算。注意指明数据片、数据表征、和胶合及超胶合表征是如何确定的。

18.10 选择现有计算机程序的五个模块。使用在 18.4.2 描述的 Dhama 度量来计算每个模块的耦合值。

18.11 开发一个软件工具用以计算一个编程语言模块的环形计数复杂度。你可以任选一种语言。

18.12 开发一个软件工具用以计算 GUI 的布局恰当性。工具能使你在布局实体间赋予变迁开销。[注意：认识到当可能的方格位置数目增长时，其他布局方案的潜在数量的大小增长得很快。]

18.13 开发一个小型的软件工具用来对你选择的编程语言源代码进行 Halstead 分析。

18.14 研究文献并撰写一篇关于 Halstead 和 McCabe 的软件质量度量(以错误数来测度)关系的论文。这些数据是不是引人注意？推荐应用这些度量的指南。

18.15 研究关于支持测试用例设计的度量的近期文献，并在班上宣读你的发现。

18.16 一个遗产系统有 940 个模块，最新发布中需要其中的 90 个模块被改动，另外，40 个新模块被加入其中，且 12 个旧模块被删除。为该系统计算软件成熟度指标。Press， 1981.

Sheppard， M.，Software Engineering Metrics， McGraw-Hill， 1992.

由 Denvir，Herman，和 Whitty 编辑的论文集讨论了软件测度理论(Proceedings of theInternational BCSFACS Workshop.Formal Aspects of Measurement， Springer-Verlag，1992)。Shepperd(Foundations of Software Measurement， Prentice-Hall， 1996)也详细讨论了测度理论。

在[IEE94]中提出了对众多有用的软件度量的完整总结。通常而言，每个度量的讨论都已经被提练到计算度量所需的关键的“基本”(测度)和影响计算的适当的关系，它的附录中提供了一些讨论和许多参考书。

一个覆盖了过程、项目、和产品度量的广泛的软件度量书目已经由 Software MeasurementLaboratory at the University of Magdeburg 所收集，可以在下面网址上找到：

http：//irb.cs.uni-magdeburg.de/se/metrics_eng.html

许多咨询公司专门研究软件度量。它们的网址包含了有用的信息：

Software Productivity Research， Inc.http：//www.spr.com/

Quantitative Software Management， Inc. http：//www.qsm.com

美国国防部和其他政府机构也有在如下网址中描述的正在进行的度量项目：

http：//www.army.mil/optec-pg/homepage.htm

http：//vislab-www.nps.navy.mil/～sm/metrics.shtml

加拿大 Software Productivity Center 也有正在进行的度量项目，下面网址提供了工具以及手册和其他有用的信息：

http：//www.spc.ca/spc/metrovrv.htm

一批剪裁后以适应工业和学术特定需要的教育材料，称为 METKIT，在以下网址上可得到：

http：//www.sbu.ac.uk/～csse/metkit.html

一个有 500 个条目的关于技术度量的数据库已经由 Horst Zuse 教授建立，他还写了联机的关于软件度量发展历史的文章，并包括了广泛的参考书目：

http：//www.cs.tu-berlin.de/～zuse/3-hist.html

关于技术度量的最新WWW文献列表可以在httP：//www.rspa.com上找到。

① 值得指出质量可扩展到分析，设计和编码模型的技术属性。展示高质量的模型（在技术角度）将使得软件从用户角度展示出高的质量。

① 已经出现了软件度量的许多专著（例如，见参考文献[FEN94]，[ROC94]可得更广泛的文献目录），并且对特定的度量的批评也很普通。但是，许多批评集中在深奥的话题上，且忽略了现实世界中的度量的主要目标——帮助工程师建立一个系统的和客观的方法来获得对他的工作的洞察且用来提高产品的质量。

① 请注意也可以同样做出严厉的相反论点。这是软件度量的性质。

① 分析模型包括数据、功能、行为的表示。见第 11 和 12 章有详细的信息。

② SafeHome是一个在前些章节用作应用程序例子的家庭保安系统。

① 在下面原语后面括号中的首字母缩写是用来指示特定原语的计数的；例如，FuP表示在分析模型中的功能原语的数目。

① 一个完整的规格质量度量的讨论不在本章的范围。详细信息见［DAV93］。

① 如在第 13 章的讨论，扇出指示了直接从属与模块i的模块个数，也就是说，被模块I直接调用的模块数。

① 作者在[DHA95]提到当进行更多的实验性验证时，k和a，b，和c（在下一个公式讨论）的值可以进行调整。

① 应该指出Halstead的“定律”已经产生了很大的争议且并不是每个人都同意其根本的理论是正确的。但是，已经对许多编程语言对Halstead的发现进行了试验验证。（例如，[FEL89]）

第四部分　面向对象的软件工程

在本书的这一部分，我们讨论那些应用于面向对象软件的分析、设计和测试的技术概念、方法和测度。下面章节中，我们将涉及下列问题：

• 什么是应用于面向对象思维的基本概念和原则？

• 如何计划和管理面向对象的软件项目？

• 什么是面向对象的分析？它的各种模型如何能使软件工程师理解类及它们的关系和行为？

• 什么是“使用实例”？它如何被用于分析系统的需求？

• 传统的和面向对象的方法有何不同？

• 什么是面向对象设计模型的构成成分？

• 如何将“模式(pattern)”用来创建面向对象设计？

• 什么是应用于面向对象软件的测试的基本概念和原则？

• 当考虑面向对象软件时，测试策略和测试用例的设计方法将如何改变？

• 什么技术度量可用于评估面向对象软件的质量？

一旦这些问题得到回答，你将了解如何使用面向对象的范型去分析、设计、实现和测试软件。

第 19 章　面向对象的概念和原则

我们生活在对象的世界，这些对象存在于自然中、人造实体中、商业中、以及我们使用的产品中，它们可以被分类、描述、组织、组合、操纵和创建。因此，为计算机软件的创建提出面向对象的观点是毫不奇怪的，这是一种模型化世界的抽象方法，它可以帮助我们更好地理解和探索世界。

软件开发的面向对象方法首先于 60 年代后期提出，然而，花了几乎 20 年的时间对象技术才开始被广为使用。在 90 年代的前半部，面向对象软件工程变成了很多软件产品建造者、以及不断增长的信息系统和工程专业人员的首选范型。随时间的流逝，对象技术正在取代传统的软件开发方法。一个重要的问题是“为什么”？

这个问题的答案(就象很多其他软件工程问题的答案一样)并不简单。某些人将争辩说软件专业人员只是向往“新”技术，但是这个观点过分简单化，面向对象技术确实导致了一系列内在的优点，在管理和技术层次均提供了优势。

对象技术导致复用，而(程序构件的)复用导致更快的软件开发和高质量的程序。面向对象软件易于维护，因为它的结构是内在松耦合的，这样，当进行修改时，产生较少的副作用，也对软件工程师和客户来产生较小的挫折感。此外，面

向对象系统易于进行适应性修改及伸缩(即，通过组装可复用子系统而可以创建大的系统)。

在本章，我们介绍形成理解对象技术的基础的基本原理和概念，在第四部分的其他剩余部分，我们考虑形成创建面向对象产品和系统的工程途径的基础的方法。

19.1 面向对象的范型

曾经有很多年，“面向对象”(OO)被用于指使用一系列面向对象程序设计语言(如，Ada95，C++，Eiffel，Smalltalk)的软件开发方法。今天，OO 范型包含完整的软件工程观点。EdwardBerard 有如下陈述：

如果在软件工程过程的早期和全程采用面向对象技术，则该技术将产生更多的优势。那些考虑面向对象技术的人们必须评估它对整个软件工程过程的影响。仅仅使用面向对象程序设计(OOP)将不会产生最好的结果，软件工程师及其管理者必须考虑面向对象需求分析(OORA)、面向对象设计(OOD)、面向对象领域分析(OODA)、面向对象数据库系统(OODBMS)和面向对象计算机辅助软件工程(OOCASE)等。

熟悉传统软件工程方法(本书第三部分讨论)的读者可能会对上面的陈述不以为然：“当我们使用传统方法开发软件时，我们也使用分析、设计、编程、测试、和其他相关技术，为什么 OO 应该有任何不同呢？”确实，为什么 OO 应该有任何不同呢？简而言之，它不应该有不同！

在第 2 章，我们讨论了一系列软件工程的不同的过程模型，虽然这些模型的任意一个均可以适用于 OO 技术，但是，最好的选择应该认识到：OO 系统往往随时间演化，因此，演化过程模型结合鼓励构件组装(复用)的方法是 OO 软件工程的最好范型。在图 19-1 中，构件组装过程模型(第 2 章)已被剪裁以适应 OO 软件工程。

OO 过程沿演化的螺旋迭代，从用户通信起步，在这里，问题域被定义，并且定义基本的问题类(本章后面将讨论)；计划和风险分析阶段建立 OO 项目计划的基础和 OO 软件工程关联的技术工作遵循在阴影方框中显示的迭代路径，OO 软件工程强调复用，因此，类在被建造前，先在(现存的 OO 类)库中“查找”，当在库中没有找到时，软件工程师应用面向对象分析(OOA)、面向对象设计(OOD)、面向对象程序设计(OOP)、和面向对象测试(OOT)来创建类及从类导出的对象，新的类然后又被放入库中，使得可以在将来被复用。

面向对象的观点要求演化的软件工程方法，就如我们将在本章及下面章节中看到的那样，要在一次单个迭代中为主要的系统或产品定义出所有必需的类是极端困难的，当 OO 分析和设计模型演化时，对附加类的需要就变得明显化。正因为如此，上面描述的范型特别适合于 OO。在 19.4.1 节对 OO 过程模型进一步讨论。

19.2 面向对象概念

任何对面向对象软件工程的讨论必须从解释术语“面向对象”开始。什么是面向对象的观点？为什么一个方法被认为是面向对象的？什么是对象？很多年来，存在很多关于这些问题的正确回答的不同的观点(如参考文献[BER93]、[TAY90]、[STR88]及[BOO86])，在下面的讨论中，我们试图综合这些观点中最公共的部分。

为了理解面向对象的观点，考虑一个现实世界对象的例子——你现在正坐在上面的东西——椅子(chair)，chair 是某个称为家具(furniture)的更大的对象类的一个成员(也使用术语“实例”)。一组类属属性和类 furniture 中的每个对象关联，例如，所有家具在其很多可能的属性中，有 cost(价格)、dimension(尺寸)、weight(重量)、Location(位置)和 color(颜色)等，无论我们谈论桌子或椅子、沙发或衣橱，这些属性总是可用的。因为 chair 是类 furniture 的成员，椅子继承了为类定义的所有属性。该概念在图 19-2 中以图形的方式说明。

一旦类被定义，当新的类的实例被创建时，属性可以被复用，例如，假定我们要定义一个新的称为 chable(在椅子(Chair)和桌子(table)间的东西)的对象，它也是类 furniture 的成员，chable 继承了 furniture 的所有属性。

我们试图通过描述类的属性给出了它的定义，但是某些东西并未考虑。在类 furniture 中的每个对象可以被一系列不同的方式操纵，它可以被买和卖、物理地修改(如，你可以锯去一条腿或漆上紫色)或从一个地方移到另一个地方。这些操作(也称“服务”或“方法”)将修改对象的一个或多个属性，例如，位置(location)属性是一个复合数据项，定义如下(使用第 12 章的数据字典符号)：

位置=大厦+楼层+房号

则命名为 move(移动)的操作将修改构成属性位置的一个或多个数据项(大厦、楼层或房子)。为了完成该操作，“移动”必须“知道”这些数据项。操作“移动”可用于桌子或椅子，只要二者是类 furniture 的实例。所有对类 furniture 的合法操作(如，买，卖，重量)被“联结”对象的定义中，如图 19-3 所示，并且被类的所有实例继承。

对象 chair(和所有一般的对象)封装数据用于(定义椅子的属性值)、操作(用以修改对象 chair 的属性的动作)、其他对象(可定义复合对象[EVB89])、常量(给定值)、以及其他相关信息。封装意味着所有信息被包装在一个名字下，可以作为一个规约或程序构件复用。

既然我们已经介绍了一些基本概念，面向对象的更正式的定义将是更有意义的。Coad 和 Yourdon[COA91]给出该术语的如下定义：

面向对象(object-oriented)=对象(objects)+分类(classification)

+继承(inheritance)+通信(communication)

已经介绍了其中的三个概念，我们将在后面再讨论通信。

19.2.1 类和对象

导致高质量设计(第 13 章)的基本概念对使用传统方法和面向对象的方法的系统开发均有效。为此，计算机软件的 OO 模型必须展示导致有效模块化的数据和过程抽象。类是一个 OO 概念，它封装了对描述某些现实世界实体的内容和行为所需的数据和过程抽象。Taylor[TAY90]使用如图 19-4 右边所示的符号来描述类(和从类导出的对象)。

描述类的数据抽象(属性)被一个能够以某种方式操纵数据的过程抽象(称为操作、方法或服务)的“墙”所包围，唯一能接触到属性(及操作属性)的方式是通过构成“墙”的其中一种方法。因此，类封装数据(在“墙”内)和操纵数据的过程(构成墙的方法)，这样到达隐蔽信息和减少与变化相伴生的副作用的影响。因为方法往往只操纵有限的属性。它们是结合一起的，并且因为通信仅仅通过构成“墙”的方法进行，所以类往往和系统中其他元素间松耦合。所有这些设计特征(第 13 章)均导致高质量的软件。

以另一种方式陈述，类是一个对一组相似对象的一般性描述(如，模板、模式或蓝图)。通过定义，存在于类中的所有对象继承其属性和用于操纵属性的操作。超类是类的集合，子类是类的实例。

这些定义蕴含了类层次的存在，超类的属性和操作被子类继承，而子类也可加入自己“私有的”属性和方法。类 furniture 的类层次如图 19-5 所示。

19.2.2 属性

我们已经看到，属性依附于类和对象，并且以某种方式描述类或对象。Champeaux 及其同事给出了如下的关于属性的讨论[CHA93]：

现实的实体经常用指明其稳定特性的词来描述。大多数物理对象具有形状、重量、颜色和材料类型等特性；人具有生日、父母、名字和眼睛颜色等特性。特性可被视为在类和某确定域之间的二元关系。

上面提到的二元关系蕴含着属性可以取由一个枚举域定义的值。大多数情况下，域只是某些特定值的集合。例如，假定类自动具有属性颜色，颜色的值域是{白，黑，银，灰，蓝，红，黄，绿}；在更复杂的情况下，域可以是类的集合，继续上面的例子，类自动也有属性 power train，它包含下面值域：{16 valve economy option， 16 valve sport option， 24 valve sport option， 32valve luxury option}，上面提到的每个“选项”具有一组自己的特定属性。

特性(域的值)可以通过赋予某属性一个缺省值(特性)而被扩展，例如，上面提到的 powertrain 属性被赋予 16 valve sport option 的缺省值。对某特殊的特性通过赋予一个＜值，概率＞对而关联一个概率也可能是有用的，考虑 automobile 的 color 属性，在某些应用(如，制造计划)中，可能有必要为每种颜色赋予一个概率(黑和白是最有可能作为汽车颜色的)。

19.2.3 操作、方法和服务

对象封装数据(表示为属性集合)和处理数据的算法，这些算法称为操作、方法或服务，并且可被视为传统意义上的模块。

被对象封装的每个操作提供了对象的一种行为，例如，对象 automobile 的操作 GetColor 将提取存放在 color 属性中的颜色，该操作的存在暗示类 automobile 被设计为可接收要求类中某特定实例的颜色的激励[JAC92](我们将激励称为“消息”)。一旦当对象接收到激励，它就开始某种行为，这可以简单到查询汽车的颜色，也可以复杂到启动一系列将在不同的对象中传递的激励。对后一种情形考虑一个例子，对象 1 接收的初始激励导致生成被发送给对象 2 和对象 3 的两个其他激励，对象 2 和对象 3 所封装的操作对激励作出反应，向对象 1 返回必要的信息，对象 1 使用返回的信息来满足初始激励要求的行为。

19.2.4 消息

消息是对象间交互的手段，使用在前面节中引入的术语，消息刺激接收对象产生某种行为，通过操作的执行来完成相应行为。

图 19-6 以图形的方式说明了对象间的交互作用，在 sender object 中的操作产生如下形式的消息：

message：[destination， operation， parameters]

这里 destination 定义被消息刺激的接收者对象(receiver object)，operation 指接收消息的方法，parameters 提供操作成功所必需的信息。

作为在 OO 系统中消息传递的例子，考虑图 19-7 所示的对象，四个对象 A、B、 C、D 通过消息传递相互通信，例如，如果对象 B 要求对象 D 配合处理操作 op10，它将向 D 传递如下消息：

message：[D，op10，＜data＞]

作为执行 op 10 的一部分，对象 D 可能传递如下消息给对象 C：

message： [C， op08，＜data＞]

C 找到操作 op08，完成该操作，然后将合适的返回值传送给 D。操作 op10 完成其执行并将返回值传送给 B。

Cox[COX86]以如下方式描述对象间的交互：

对象通过接收到一个告知它做什么的消息而被请求完成某一操作。接收者(对象)对消息的反应是：首先选择实现该消息名的操作，执行该操作，然后返回控制给调用者。

消息传递将面向对象系统连在一起，消息提供了对个体对象的行为和 OO 系统整体的洞察。

19.2.5 封装、继承和多态

在 19.2.1 到 19.2.4 节中引入的结构和术语区分了 OO 系统和传统的系统不同，面向对象系统的三个特征使它们得以区分。正如我们已经注意到，OO 的类和从类导出的对象封装数据和数据上的操作在同一个包中，这提供了一系列重要的益处：

• 数据和过程的内部实现细节对外界隐蔽(信息隐蔽)，这减少当变化发生时副作用的传播。

• 数据结构和对它们的操作被合并在单个名字的实体(类)中，这将便利于构件复用。

• 简化被封装对象间的接口。发送消息的对象不需要关心接收对象的内部数据结构，因此，接口被简化，系统耦合度被降低。

继承是传统系统和 OO 系统间的关键区别之一。子类 Y 继承其超类 X 的所有属性和操作，这意味着，所有原本针对 X 设计和实现的数据结构和算法，不需要进行进一步的工作立即可被 Y 使用。复用被直接实现。

对包含在超类中的数据或操作的任何修改立即被继承该超类的所有子类继承，因此，类层次变成了一种机制，通过它，(在高层的)变化可以立即传播到系统的其他部分。

必须注意，在类层次的每个层上，新的属性和操作可以被加到那些从其高层继承来的属性和操作中，事实上，一旦要创建新的类，软件工程师可有一系列选择：

• 可以从头开始设计和建造类，即，不使用继承。

• 搜索类层次以确定是否某祖先类包含了所需的属性和操作中的大部分。

• 新的类继承祖先类，并根据需要加入新的内容。

• 重新构造类层次，使得需要的属性和操作可被新的类继承。

• 现存类的特征可以被重载，并对新的类实现私有版本的属性或操作。

为了阐明如何对类层次重构以导致希望的类，考虑图 19-8 所示的例子。图 19-8a 所示的类层次使得我们能够导出类 X3 和 X4，各自具有特性 1、2、3、4、5 和 6，以及 1、2、 3、 4、5 和 7。现在，假设需要一个仅具有特性 1、2、3、4 和 8 的新类，本例中称为类 x2b，为了导出这个类，类层次可以如图 19-8b 所示重构。必须注意，类层次的重构可能是困难的，为此，有时需要使用重载。

在本质上，当属性和操作被以正常的方式继承时重载发生，但然后被修改以满足新类的特定需要时。如 Jacobson 所说，当使用重载时，“继承是不可传递的”[JAC92]。

在某些情形，可能试图从某类继承某些属性和操作，而从另一个类继承其他的属性和操作，这称为多继承，这是一个有争议的特性。通常，多继承使类层次复杂化，并产生在配置控制(第 9 章)方面的潜在问题，因为多继承序列更难于跟踪，对位于类层次高层的类定义的改变可能产生对低层的类的未预料的影响。

多态是大大减少于扩展现存 OO 系统所需的工作量的特性。为了理解多态，考虑一个传统的应用，它必须画四种不同类型的图：线图、饼图、柱状图和 Kiviat 图。理想情况，一旦收集到某类图形所需的数据，图将自己画出。为了在传统应用中完成此功能(并保持模块内聚性)，必须为每种图形类型开发一个画图模块，那么，在对每种图形类型的设计中，必须嵌入类似于下面的控制逻辑：

```
case of graphtype:

if graphtype=linegraph then DrawLineGraph(data);

if graphtype=piechart then DrawPieChart(data);

if graphtype=histogram then DrawHisto(data);

if graphtype=kiviat then DrawKiviat(data);

end case;
```

虽然这个设计是相当直接的，但是，加入新的图形类型将可能很棘手。对每种图形类型必须创建一个新的画图模块，并且对每种类型的控制逻辑将必须更新。

在 OO 系统中解决这一问题，上面提到的每种图形均变成了一个一般类 graph 的子类，使用重载概念 [TAY90] ，每个子类定义一个操作 draw，一个对象可以传递 draw 消息给任意子类的任意实例对象，接收消息的对象将激活它自己的 draw 操作来创建合适的图形，因此，上面的设计简化为：

graphtype draw

当子系统中加入一种新的图形类型时，创建一个具有自己的 draw 操作的子类，但是对任何希望画图的对象不需做任何修改，因为消息 graphtype draw 是不变的。总而言之，多态使得一系列不同的操作具有相同的名字，这使得对象间相互松耦合，相互更加独立。

19.3 标识对象模型的元素

对象模型的元素——类及对象、属性、操作、和消息——已在前一节中一一定义和讨论，但是，我们如何对一个实际问题标识这些元素？下面的几节将提出一系列非正式的指南，以帮助对象模型的元素的标识。

19.3.1 标识类和对象

如果环视房间一圈，你可以发现一组易于标识、分类和定义(使用术语属性操作)的物理对象。但是，当你“环视”软件应用的问题空间时，对象可能是较难理解的。

我们可以通过检查问题陈述或(使用 12 章中的术语)对将被建造的系统的过程叙述进行“语法分析”来开始标识对象。通过在每个名词或名词短语下面划线并将其输入简单表格中而确定对象，必须注意同义词。如果对象是对实现解决方案必须的，则它是解空间的一部分，否则，如果仅仅是对描述解决方案必需的，则它是问题空间的一部分。但是，一旦所有名词已经被孤立出来，我们将查找什么？

对象以如图 19－9 所示的方式之一展示，对象可以是：

• 外部实体(如，其他系统、设备、人员)，它们生产或消费被基于计算机的系统使用的信息。

• 事物(如，报告、显示、文字、信号)，它们是问题的信息域的一部分。

• 发生的事情或事件(如，性质变迁或完成一系列的遥控的运动)，它们出现在系统运行的环境内。

• 角色(如，管理者、工程师、销售人员)，由和系统交互的人员所扮演。

• 组织单位(如，分支、小组、小队)，它们和应用相关。

• 位置(如，制造场所或装载码头)，它们建立问题和系统整体功能的环境。

• 结构(如，传感器、四轮交通工具或计算机)，它们定义一类对象，或者在极端情况下，定义对象的相关类。

上面的分类仅仅是在文献中提出的很多分类的一种，例如，Budd [BUD96]建议了一种类的分类学，它包含数据的生产者(源)和消费者(潭)、数据管理者、视图或观察者类、以及帮助者类。

我们也必须注意到对象不是什么。通常，一个对象决不能有“命令型过程名”[CAS89]，例如，如果医学图像系统的软件开发者定义一个名为“图像反转”的对象，他们可能正在犯一个微妙的错误。从软件得到的 image 当然是一个对象(它是一种事物，是信息域的一部分)，图象的反转(inversion)是应用到该对象的一个操作，inversion 可被定义为对象 image 的一个操作，但是它不可定义为单独的蕴含反转图像的对象。如 Cashman [CAS89] 所说，“面向对象的意图是封装数据和数据上的操作，但仍然保持分离”。

为了阐明如何在分析的早期阶段定义对象，我们回到在本书前面引入的 SafeHome 安全系统的例子，在第 12 章，我们完成了对 SafeHome 系统的过程叙述的“语法分析”过程叙述被重引用如下：

SafeHome 软件使得房主能够在安装时配置安全系统、监控所有和安全系统连接的传感器、以及通过键盘及包含在 SafeHome 控制面板(如图 11—4 所示)中的功能键和房主交互。

在安装过程中，SafeHome 控制面板被用于“编程”和配置系统，每个传感器被赋予一个编号和类型，用于启动和关闭系统的主要码被编程，并且传感器事件发生时所拨电话的电话号码被输入。

当接收到传感器事件后，软件激活附于系统上的警铃，在一段延时(由房主在系统配置阶段规定)后，软件拨监控服务的电话号码，提供关于位置的信息，报告被检测到的事件的性质。此号码将每 20 秒重拨一次，直至电话接通为止。

所有和 SafeHome 的交互由用户交互子系统管理，它读取通过键盘和功能键的输入，在 LCD 显示板上显示提示信息和系统状况，键盘交互采用如下形式…

抽取其中的名词，我们可以建议一组潜在对象：

潜在对象	类	一般分类
房主	角色或外部实体	
传感器	外部实体	

(续)

潜在对象/类	一般分类
控制面板	外部实体
安装	发生的事情(事件)
系统(别名安全系统)	事物
编号、类型	非对象，传感器的属性
主密码	事物
电话号码	事物
传感器事件	发生的事情(事件)
警铃	外部实体
监控服务	组织单位或外部实体

上面表格可以继续补充，直至过程叙述中的所有名词全被考虑到。注意，我们称在表中的每个项为潜在对象，在最后决定前，我们必须进一步考察。

Coad 和 Yourdon[COA91]建议了 6 个选择特征，它们应用于分析员考虑每个潜在对象是否应包含在分析模型中：

1.保留的信息——仅当关于潜在对象的信息必须被记住才能使系统可以工作时，则它在分析中是有用的；

2.需要的服务——潜在对象必须拥有一组可标识的操作，它们可以以某种方式修改对象属性的值；

3.多个属性——在需求分析阶段，关注点应该是“较大的”信息(仅具有单个属性的对象在设计时可能是可用的，但是在分析阶段，最好将它表示为另一个对象的属性)；

4.公共属性——可以为潜在对象定义一组属性，这些属性适用于对象每一次发生的事件；

5.公共操作——可以为潜在对象定义一组操作，这些操作适用于对象每一次发生的事件；

6.必要的需求——出现在问题空间的外部实体以及对系统的任何解决方案的操作都是必要的生产或消费信息将几乎总是被定义为需求模型中的对象。

只有一个潜在的对象满足所有(或几乎所有)上面提到的特征，才可能会被考虑成为包含在需求模型中的合法对象。决定潜在对象是否包含在分析模型中具有某种主观性，后面的评估可能使得某对象被丢弃或恢复。然而，OOA 的第一步必须定义对象并进行决策(即使是主观的)。记住以上所述，我们将选择特征应用到 SafeHome 的潜在对象表上：

潜在对象/类	适用的特征编号
房主	拒绝：1、2失败，即使6适用
传感器	接受：所有均适用
控制面板	接受：所有均适用
安装	拒绝
系统(别名安全系统)	接受：所有均适用
编号、类型	拒绝：3失败，传感器的属性
主密码	拒绝：3失败
电话号码	拒绝：3失败
传感器事件	接受：所有均适用
警铃	接受：2、3、4、5、6适用
监控服务	拒绝：1、2失败，即使6适用

应该注意到：(1)上面的表并不是全包含的，可能必须加入其他的对象才能完成模型；(2)某些被拒绝的潜在对象将变成被接受的对象的属性(如，编号和类型是传感器 sensor 的属性，主人密码和电话号码可能变成 system 的属性)；(3)对问题的不同陈述可能导致不同的“接受或拒绝”决策(如，如果每个房主有自己的密码或通过声音纹来识别，则房主对象 homeowner 将满足特征 1 和 2，从而被接受)。

19.3.2 表示属性

属性描述已经被选择包含在分析模型中的对象。在本质上，正是属性定义了对象——它们阐明了在问题空间中对象意味着什么，例如，如果我们将建造一个用于跟踪职业棒球运动员的统计数据的系统，对象 Player 的属性和用于职业棒球运动员退休金系统中的相同对象的属性间就存在较大差异，对前者，名字、位置、平均击球数、接球百分率、已打球年数、以及已参加球赛场数等属性是相关的；对后者，这些属性中的某些是有意义的，但是其他将被平均工资、信用度、选择的退休金计划选项、通信地址等属性所替代(或增强)。

为了为对象开发有意义的属性集，分析员可以再次研究对问题的过程叙述(或范围陈述)并选择那些应“属于”该对象的事物，此外，对每个对象应回答下列问题：“在当前的问题范围内，什么数据项(复合的和/或基本的)完整地定义了该对象”？

为了阐明这点，我们考虑为 SafeHome 定义的 system 对象，在前面我们已提到房主可以配置安全系统以反映传感器信息、警报反应信息、激活/非激活信息、标识信息等等，使用在第 12 章讨论的数据字典中的内容描述符号，我们可以以如下方式表示这些复合数据项：

传感器信息=传感器类型+传感器编号+警报阀值

警报反应信息=延迟时间+电话号码+警报类型

激活/非激活信息=主密码+可允许的尝试次数+临时密码

标识信息=系统 ID+验证电话号码+系统状况

在等号右边的每个数据项可以进一步被定义到基本层次，但是对我们的目的来说，它们构成了 system 对象的合理的属性列表(图 19-10 中阴影部分)。

19.3.3 定义操作

操作定义对象的行为并以某种方式修改对象的属性，更专业地说，操作修改包含在对象中的一个或多个属性值。因此，操作必须“知道”对象属性的性质，并且必须以一种使得它可以操纵从属性导出的数据结构的方式来实现。

虽然存在很多不同类型的操作，但总体来说它们可被分为三类：(1)以某种方式操纵数据的操作(如，增、删、重格式化、选择)；(2)完成某种计算的操作；(3)为控制事件的发生而监控对象的操作。

作为对分析模型的对象导出操作集的第一次递进，分析员可以再次研究问题的过程叙述(或范围的陈述)并选择那些应属于对象的操作。为了到达这个目标，可再次进行语法分析，从中分离出动词。部分动词将是合法的操作并可以容易地和特定对象相联系，例如，从本章前面的 SafeHome 过程叙述中，我们看见“传感器(sensor)被赋予(is assigned)一个编号和类型”或“主密码被编程(programmed)以用于启动(arming)和关闭(disarming)系统”这样的句子，这两个句子表明了下面的事情：

- 赋予(assign)操作和传感器(sensor)对象相关。

- 编程(Program)操作适用于 system 对象。

- 启动(arm)和关闭(disarm)操作可以适用于 system 对象也可能使用数据字典符号，最终定义了系统状态(system status)，如下。

系统状况=[启动态(armed)关闭态(disarmed)]

通过进一步研究，有可能操作 program 将被分解成一系列用于系统配置的更特定的子操作，例如，program 蕴含着指定电话号码、配置系统特征(如，创建传感器表，输入警报特征)以及输入密码，但是现在，我们定义 program 为一个单一的操作。

除了语法分析，我们可以通过考虑发生在对象间的通信而获得对其他操作的进一步了解，如我们已经看见的那样，对象间通过传递消息来相互通信，对象间传递的消息暗示了一组必须存在的操作。

19.3.4 完成对象定义

对操作的定义是完成刻划对象的最后一步，在 19.3.3 节中，操作通过对系统过程叙述的语法分析而被精选出来，其他的操作可通过考虑对象的“生命历史”[COA91]和考虑在系统的对象间传递的消息而确定。

可以通过对对象必须被创建、修改、以某种方式操纵或读出、可能被删除的认识来定义对象的类属生命历史。对 system 对象，这可以扩展到在其生命期内(在此情况下，指 SafeHome 保持运行的区间)对发生的已知活动的反应。某些操纵可以从对象间可能的通信来确定，例如，传感器事件(sensor event)将发送消息给 system 以显示(display)事件位置和编号；控制面板(control panel)将发送复位(reset)消息以更新系统状况(一个属性)；警铃 (audible alarm)将发送查询(query)消息；控制面板(control panel)将发送修改(modify)消息以在不重新配置整个 system 对象前提下改变一个或多个属性；传感器(sensor)也将发送消息去拨打包含在对象中的电话号码。也可考虑其他消息并导出相应操作，最终的对象定义如图 19-10 中所示。

类似的方法可以用于在 SafeHome 中定义每个对象，在对当前确定的每个对象的属性和操作的定义完成后，就已创建完一个 OOA 的初始模型。更详细的对分析模型(作为 OOA 的一部分而被创建)的讨论在第 20 章。

19.4 面向对象软件项目的管理

如我们在本书第二部分所讨论的那样，现代软件项目管理可以被分解为如下活动：

1. 建立项目的公共过程框架。

2. 使用框架和历史度量信息来进行工作量和时间估算。

3. 规定工作产品和里程碑，以使得进展可以被测度。

4. 定义用于质量保证和控制的检查点。

5. 管理当项目进展中不时发生的变化。

6. 跟踪、监控和控制项目进展。

从事面向对象项目的技术管理者运用这 6 项活动，但是由于面向对象软件的独特性质，这些管理活动每个均有微妙的不同并必须以不同的思路来加以运用。

在下面几节中，我们考察对面向对象项目的软件项目管理，管理的基本原则是相同的，但是必须采用适当的技术来有效地管理 OO 项目。

19.4.100 的公共过程框架

公共过程框架 common process framework(CPF)定义了组织的软件开发和维护途径，它标识了用于建造和维护软件的软件工程范型以及需要的任务、里程碑和可交付的产品。它确立了用于不同种类的项目开发的管理严格程度。

CPF 总是可以进行适应性的修改，使得它可以满足每个项目组的需要，这是它最重要的特征。

对 00 项目的有效的 CPF 不是线性的顺序模型。线性顺序模型，即广为所知的生命期或瀑布模型，假定在项目的开始定义要求，并且工程活动以线性顺序的模式进展。就其本身性质而言，面向对象软件工程必须应用鼓励递进发表的范型，即，00 软件通过一系列循环周期而演进。用于管理 00 项目的公共过程框架必须本质上是演进的，在 19.1 节中给出了一个 00 过程模型。

Ed Berard[BER93]和 Grady Booch[B0091]建议使用“递归/并行模型”来进行面向对象软件开发，在本质上，递归/并行模型以如下方式工作：•进行足够的分析以分离出主要的问题类和联系。

- 进行少量的设计以确定类和联系是否可以用实际的方式实现。
- 从库中提取可复用对象建造大致的原型。
- 通过测试发现原型中的错误。
- 获取客户对原型的反馈。
- 基于从原型、少量设计和客户反馈学到的知识来修改分析模型。
- 精化设计以适应你的修改。
- 开发特殊的对象(从库中得不到的)。
- 用来自库中的和你新创建的对象组装新的原型。
- 进行测试以发现原型的错误。
- 获取客户对原型的反馈。

该方法一直继续，直至原型发展为可用的产品。

递归/并行模型和本章前面讨论的 00 过程模型是非常相似的，即项目递进地进展，但递归/并行模型在下面两点认识上不同：(1)对 00 系统模型的分析和设计不能在相同的抽象层次上进行；(2)可以对系统中相互独立的构件同时进行分析和设计。

Berard[BER93]以如下方式描述该模型：

• 将问题系统地分解为高度独立的构件。

• 对每个独立构件进一步进行分解(此即递归部分)。

• 对所有构件的分解工作同时进行(此即并行部分)。

• 继续此过程直至完成。

必须注意，当分析员/设计者认识到所需的构件或子构件在复用库中存在时上面的分解过程即不再继续。

为了控制递归/并行过程框架，项目管理者必须认识到进展要被增量式地计划和测度，即，项目任务和项目进度同每个“高度独立的构件”紧密联系的，要针对每个构件进行独立的进度测量。

递归/并行过程的每次递进需要计划、工程(分析、设计、类提取、原型开发和测试)和评估活动(如图 19-11 所示)。在计划阶段，对每个和独立程序构件相关联的活动进行计划、制定进度(注：针对每次递进，要进行同本次迭代的变化相适应的进度安排的调整)；为了分离出 OO 分析和设计模型中的所有重要元素，在工程的早期阶段要迭代地进行分析和设计工作；随着工程工作不断开展，产生软件的增补版本；通过评估、复审、客户评价和针对每个增补的性能测试，得到反馈信息，这将影响下一次计划和随后的增补。

19.4.2 面向对象项目的度量和估算

传统的软件项目的代码行(LOC)或功能点(FP)的估算作为项目估算的基本元素。对 OO 项目而言，一个非常重要的目标是复用，所以代码行估算的意义就不是很大，但 FP 估算可以被有效地应用，因为所需的信息域计数可很容易地从问题陈述中得到。虽然 FP 分析可以运用于 OO 项目估算，但是当我们进行递归/并行范型的迭代时，FP 测度没有提供进度安排和工作量调整所需的足够的粒度。

Lorenz 和 Kidd[LOR94]建议了下面一组项目度量：

场景脚本的数量。场景脚本是描述用户和应用间交互的详细的步骤序列，每个脚本被组织为如下形式的三元组：

{initiator，action， participant}（{发起者、动作、参与者}）

这里 initiator 是请求某种服务的对象(它启动一个消息)；action 是请求的结果； participant 是满足请求的服务器对象。

场景脚本的数量同应用的大小和当系统完成时必须开发的测试用例的数量直接相关。

关键类的数量。关键类是在 OOA 早期定义的“高度独立的构件”[LOR94]。因为关键类是问题域的核心，所以它的数量是开发该软件所需的工作量的一个指标，也是在系统开发中将应用的潜在的复用量的一个指标。

支持类的数量。支持类是实现系统所需要的，但不是和问题域直接相关的，例如：GUI(图形用户界面)类、数据库访问和操纵类、以及通信类。此外，可以为每个关键类开发支持类，且在整个递归/并行过程中迭代地定义支持类。

支持类的数量是开发软件所需的工作量的一个指标，也是在系统开发过程中将应用的潜在的复用量的一个指标。

每个关键类的平均支持类数量。通常，关键类在项目的早期被定义，而支持类在整个过程中被定义。如果对某个给定问题域已知每个关键类的平均支持类数量，则估算(基于全部类的数量)将是非常简单的。

Lorenz 和 Kidd 提出，对带有 GUI 的应用，支持类是关键类的 2-3 倍，对不带 GUI 的应用，支持类的数量最多是关键类的 2 倍。

子系统的数量。子系统是支持对系统的终端用户可见的某功能的类的集合。一旦标识出子系统，则易于安排一个合理的进度，子系统的工作可基于此进度在项目组内合理地划分。

19.4.3 一种 OO 估算和进度安排方法

软件项目估算是一门艺术，而不是一门科学，然而，这并不妨碍采用系统化的方法。为了得到合理的估算，开发多个数据点是非常重要的，即，估算应该采用不同的技术进行。对传统软件开发使用的工作量和时间估算方法(第 5 章)对 OO 也可使用，但是对很多组织来说，OO 项目的历史数据库相对较小，因此，以明显的设计用于 OO 软件的方法对传统软件的成本估算进行补充是值得考虑的。 Lorenz 和 Kidd[LOR94]提出了下面的方法：

1.使用工作量分解、FP 分析以及任何其他可应用于传统应用的方法来进行估算。

2.使用 OOA(第 20 章)开发场景脚本并确定脚本数量。注意，场景脚本的数量可能随项目进展而改变。

3.使用 OOA，确定关键类的数量。

4.划分应用的界面类型并确定支持类的倍数：

界面类型	倍数
非 GUI	2.0
基于字符的用户界面	2.25
GUI	2.5
复杂的 GUI	3.0

将关键类的数量(第 3 步)和上面的倍数相乘可得到支持类的数量的估算。

5.将全部类的数量(关键类+支持类)和每个类的平均工作单元相乘，Lorenz 和 Kidd 建议每个类以 10-15 个人日计算。

6.通过乘上每个场景脚本的平均工作单元，对基于类的估算进行交叉检查。

面向对象项目的进度安排由于其过程框架的递进性质而变得复杂，Lorenz 和 Kidd 建议了一组可以帮助进行项目进度安排的度量方法：

主要的递进的次数。回想 OO 的过程模型，一次主要的递进相应于一个弹簧的一次 360° 旋转。递归/并行过程模型将衍生出一系列小的螺旋(局部递进)，它们在主要递进的进展过程中产生。Lorenz 和 Kidd 建议 2.5 到 4 个月进行一次递进是最容易跟踪和管理的。完成的合约的数量。一份合约是“由子系统和类向其客户提供的一组相关的公共责任”[LOR94]，一份合约是一个最好的里程碑，至少一份合约应该和每个项目的递进相关联。项目经理可以将完成的合约作为 OO 项目进展的一个很好的指标。

19.4.4 面向对象项目的跟踪过程

虽然递归/并行过程模型是针对 OO 项目的最好框架，但是，任务并行性使得项目的跟踪变的困难。因为一系列不同的事物同时发生。项目经理可能难于为 OO 项目建立有意义的里程碑，通常，可以认为已“完成”了如下的主要里程碑，当满足给定的标准时：

技术里程碑：OO 分析完成

- 已经定义和复审了所有类和类层次。
- 已经定义和复审了和类关联的类属性和操作。
- 已经建立和复审了类关系(第 20 章)。
- 已经建立和复审了行为模型(第 20 章)。
- 已经注释了可复用类。

技术里程碑：00 设计完成

- 已经定义和复审了子系统集(第 21 章)。
- 分配类到子系统并通过复审。
- 已经建立和复审了任务分配。
- (第 21 章)已标识了责任和协作。
- 已经设计和复审了属性和操作。
- 已经创建和复审了消息模型。

技术里程碑：00 编程完成

- 已经根据设计模型编码实现了每个新类。
- 已经集成了提取出的类(从复用库)。
- 已经建造原型或增量。

技术里程碑：00 测试

- 已经复审了 00 分析和设计模型的正确性和完整性。
- 已经开发和复审了类—责任—协作网络(第 22 章)。
- 设计了测试用例，并且(第 22 章)已经对每个类进行了类—级的测试。
- 设计了测试用例，并且完成集簇测试(第 22 章)，完成类的集成。
- 完成系统级的测试。

回忆在本章前面讨论的递归/并行模型，重要的是要注意：当将不同的增补交付给客户时，可能会再次重复每一个里程碑。

19.5 小结

对象技术反应了对世界的自然视图。对象按类和类层次被分类，每个类包含一组描述它的属性和一组定义其行为的操作。对象几乎模拟了问题域的可标识的全部方面：外部实体、事物、发生的事情或事件、角色、组织单位、位置、以及可以表示为对象的结构。很重要的一点，对象(及其从中导出的类)封装了数据和处理。处理操作是对象的一部分，并且被传递给该对象一个消息所引发。类定义，

一旦定义完成，形成了在建模、设计和实现不同级别上复用的基础。新对象可从类中通过实例化而产生。

三个重要的概念区分了 OO 方法和传统的软件工程方法。封装将数据和操纵数据的操作包装到单个命名的对象中；继承使得类的属性和操作可以被通过实例化产生的它的所有的子类和对象所继承；多态使得一系列不同的操作具有相同的名字，减少实现系统所需的代码行数并方便修改。

面向对象的产品和系统使用演进模型来开发，有时又称为递归/并行模型。OO 软件迭代地演进，并且管理时必须认识到最终产品须通过一系列的增补来实现。

思考题

19.1 面向对象范型是一个“热点”。关于它的在不同技术复杂层次上的文章已有很多，到图书馆去查找三篇当前的非技术文章(不是在工程杂志或期刊上发表的，但可以是在 PC 杂志上发表的)，并撰写一篇文章概述它们的内容。

19.2 在本章中我们没有考虑这种情形，新对象需要一个在它继承属性和操作的类中没有的属性或操作，你认为应该如何处理这个问题？

19.3 进行研究，找到思考题 19.2 的真正答案。

19.4 使用你自己的话和一些例子，定义术语“类”、“封装”、“继承”和“多态”。

19.5 复审为 SafeHome 系统定义的对象，你认为在开始建模时还应该定义其它的对象吗？

19.6 考虑典型的图形用户界面(GUI)，为典型地 GUI 中出现在的界面实体定义一组类(和子类)。确保定义了合适的属性和操作。

19.7 给出一个复合对象的例子。

19.8 对一项开发新的字处理软件的工作，已经标识了一个名为 document 的类，定义和 document 相关的属性和操作。

19.9 研究两种不同的面向对象程序设计语言并找出如何在语言的语法中如何实现消息的，给出每种语言的一些例子。

19.10 给出一个具体的类层次重构的例子(在图 19-8 中讨论过)。

19.11 给出一个多继承的具体例子，研究几篇关于该主题的文章，给出关于多继承的正反两面的论点。

19.12 给出一个指定在系统的开发范围的陈述，使用语法分析分离出系统的候选类、属性和操作。使用在 19.3.1 节中讨论的选择标准来确定在分析模型中是否使用这些类。

19.13 用你自己的话，描述为什么递归/并行过程模型适用于 OO 系统。

19.14 给出 3 到 4 个在 19.4.2 节中讲述的关键类和支持类的例子。

第 20 章　面向对象分析

当开发一个新的产品或系统时，我们如何以遵从 OO 软件工程的方式来刻划它？什么是相关的对象？它们如何相互关联？对象如何在系统的范围内工作？我们应如何对问题刻划或建模以使得我们可有效地进行设计？

所有这些问题涵盖在面向对象分析(OOA)的范围内，OOA 是作为 OO 软件工程的第一项技术活动来实现的。OOA 基于在第 11 章引入的一组基本原则，为了建立一个分析模型，要运用如下 5 个基本原则：(1)建模信息域；(2)描述模块功能；(3)表示模型行为；(4)分解以模型显示更多细节；(5)早期模型表示问题的本质，而后期模型提供实现细节。这些原则形成了在本章中讨论的 OOA 方法的基础。

OOA 的意图是定义所有和被求解的问题相关的类(及同类关联的关系和行为)，为了达到这个目标，必须完成以下任务显示：

1.必须在客户和软件工程师之间沟通了解基本的用户需求。

2.必须标识类(即，定义属性和方法)。

3.必须刻划类层次。

4.表示对象—对象关系(对象连接)。

5.必须建模对象行为。

6.任务 1 到 5 递进地反复使用，直至完成建模。

代替用传统的输入—加工—输出(信息流)模型或仅从层次信息结构唯一地导出模型考察问题的方式，OOA 引入了一系列新概念。这些新概念看上去可能有一些不寻常，但是它们实际是相当自然的。Coad 和 Yourdon［COA91］对该问题有如下论述：些不寻常，但是它们实际是相当自然的。Coad 和 Yourdon[COA91]对该问题有如下论述：

OOA(面向对象分析)是基于我们在幼儿园首先学到的概念：对象和属性、类和成员、整体和部分。为什么我们花了这么长的时间才将这些概念应用于信息系统的分析和规约——也许是由于我们忙于在结构化分析的全盛时期“跟随潮流”而未曾考虑其他的选择。

重要的是要注意：对作为 OOA 基础的“概念”，不存在一致的共识，但是一些关键思想重复地出现，这些正是我们将在本章中考虑的。

20.1 面向对象的分析

面向对象的分析的目标是开发一系列模型，这些模型被用来描述以满足一组客户需求的计算机软件。OOA 和在第 12 章描述的传统分析方法一样，建造一个多部分的分析模型以满足这个目标。分析模型(在第 19 章描述的对象模型的元素的语境内)描述信息、功能和行为。

20.1.1 传统方法和 OO 方法

面向对象分析是否确实不同于在第 12 章讨论的结构化分析方法？虽然争论仍在继续，Fichman 和 Kemerer [FIC92] 正面地阐述了这个问题：够　治龇椒ǎ克淙徽　攵栽诩绦　*

我们的结论是面向对象分析方法代表了相对于面向过程的方法学(如结构化分析)的根本性变化，但相对于面向数据的方法学(如信息工程方法)仅仅是适当增补。面向过程的方法学在建模过程中的关注点不是对象的内在性质，从而导致了和面向对象的三个基本原则(封装、对象分类和继承)相正交的问题域模型。

简而言之，结构化分析对需求采用独特的“输入-加工-输出”视角，数据被脱离数据的变换过程而单独考虑，系统行为虽然是重要的，但在结构化分析中往往扮演第二位的角色。结构化分析方法着重于功能分解的使用(数据流图的划分，第 12 章)。

Fichman 和 Kemerer [FIC92] 建议了 11 个“建模维数(modeling　dimensions)”，可用它们来比较各种传统的和面向对象的分析方法：

1. 实体的标识/分类[①]。

2. 一般到特殊以及整体到部分的实体关系。

3. 其他实体关系。

4. 实体属性的描述。

5. 大型模型的划分。

6. 状态和状态间的变迁。

7. 功能的详细刻划。

8.自上向下的分解。

9.端到端的处理序列。

10.排它性服务的标识。

11.实体通信(通过消息或事件)。

因为结构化分析方法及OOA方法(见20.1.2节)均有很多种，所以很难在两种方法间进行一般性的比较。然而，可以确定，建模维数8和9总是在结构化分析中出现，而很少在OOA中使用。

20.1.2 OOA概述

对象技术的流行已经衍生出许多的OOA方法[②]，每个方法都引入一个产品或系统分析的过程、一组随过程演化的模型、以及使得软件工程师能够以一致的方式创建模型的符号体系。在下面篇幅中，概略地讨论了某些流行的OOA方法[③]，其意图是提供一个由方法的作者建议的OOA过程[④]的概览。

Booch方法

Booch方法[B0094]包含“微开发过程”和“宏开发过程”两个过程，微级别定义了一组在宏过程中的每一步反复应用的分析任务，因此，演进途径得以维持。Booch方法得到了一系列自动工具的支持。Booch的OOA宏观开发过程概述如下：

·标识类和对象

提出候选对象

进行行为分析

标识相关场景

为每个类定义属性和操作

·标识类和对象的语义

选择场景并分析

赋予责任以完成希望的行为

划分责任以平衡行为

选择一个对象，枚举其角色和责任

定义操作以满足责任

寻找对象间的协作

• 标识类和对象间的关系

定义对象间存在的依赖

描述每个参与对象的角色

通过走查场景进行确认

• 进行一系列精化

对上面进行的工作制作适当的图解

定义合适的类层次

完成基于类共性的聚合

• 实现类和对象(在 OOA 的语境中，这意味着分析模型的完成)

Coad 和 Yourdon 方法

Coad 和 Yourdon 方法 [COA91] 经常被视为最容易学习的 OOA 方法之一。建模符号相当简单且开发分析模型的指引是直接明了的。Coad 和 Yourdon 的 OOA 过程概述如下：

• 使用“寻找什么(what to look for)”标准来标识对象

• 定义一般—特殊结构

• 定义整体—部分结构

• 标识主题(子系统构件的表示)

• 定义属性

• 定义服务

Jocobson 方法

也称为 OOSE(面向对象软件工程)，Jocobson 方法 [JAC92] 是 Jocobson 开发的专有的 Objectory 方法的一个简化版本。该方法和其他方法的不同点是特别强调使用实例——用以描述用户和产品或系统间如何交互的场景。Jocobson 的 OOA 过程概述如下：

• 标识系统的用户和他们的整体责任

• 建造需求模型

定义参与者(actor)和他们的责任

为每个参与者标识使用实例

准备系统对象和关系的初步视图

应用使用实例作为场景去复审模型以确定有效性

• 建造分析模型

使用参与者交互的信息来标识界面对象

创建界面对象的结构视图

表示对象行为

分离出每个对象的子系统和模型

使用使用实例作为场景去复审模型以确定合法性

Rambaugh 方法

Rambaugh [RAM91] 及其同事开发了用于分析、系统设计和对象级设计的对象建模技术(Object Modeling Technique，OMT)。分析活动创建三个模型：对象模型(对象、类、层次和关系的表示)、动态模型(对象和系统行为的表示)、以及功能模型(高层的类似 DFD 的时系统信息流的表示)。Rambaugh 的 OOA 过程概述如下：

• 开发对问题的范围陈述

• 建造对象模型

标识和问题相关的类

定义属性和关联

定义对象链接

用继承来组织对象类

• 开发动态模型

准备场景

定义事件并为每个场景开发一个事件轨迹

构造事件流图解

开发状态图解

复审行为的一致性和完整性

• 构造系统的功能模型

标识输入和输出

使用数据流图表示流变换

为每个功能开发 PSPECs(第 12 章)

规定约束和优化标准

应该注意，合并 Booch 和 Rambaugh 方法的工作正在进行，其产生的符号和启发信息集，称为统一方法(Unified Method)[BOO96][LOC96]，组合了 Rambaugh 方法对数据的重视以及 Booch 方法对行为和操作的重视。

Wirfs－Brock 方法

Wirfs－Brock 方法 [WIR90] 并没有明确地区分分析和设计任务，而是从对客户规约的估价到设计完成的一个连续过程。Wirfs－Brock 的可分析相关的任务概述如下：

• 评估客户规约

• 使用语法分析从规约中抽取候选类

• 组合类以试图标识超类

• 为每个类定义责任

• 为每个类赋予责任

• 标识类之间的关系

• 定义类之间基于责任的协作

• 构造类的层次表示以显示继承关系

• 构造系统的协作图

虽然这些 OOA 方法的术语和过程步骤各有差异，但整体的 OOA 过程是非常相似的。为了完成面向对象分析，软件工程师应该完成如下的类属步骤：

• 获取客户对 OO 系统的需求

标识场景或使用实例

建造需求模型

• 使用基本的需求作为指引来选择类和对象

• 为每个系统对象标识属性和操作

• 定义组织类的结构和层次

• 建造对象—关系模型

• 建造对象—行为模型

• 用使用实例/场景来复审 OO 分析模型

将在 20.3 和 20.4 节中更详细地讨论这些类属步骤。

20.2 领域分析

面向对象系统的分析可以在不同的抽象层次上进行。在商业或企业级，OOA 技术可以同信息工程方法，(第 10 章)结合，来定义模拟全部业务的类、对象、关系和行为，这个层次的 OOA 类似于信息策略计划；在业务范围层次，可以定义一个描述某特殊的业务范围(或某产品或系统范畴)的工作的对象模型；在应用层次，对象模型着重于特定的客户需求，因为那些需求将影响应用的实现。

在最高抽象层次(企业级)的 OOA 已超出本书范围，有兴趣的读者可参见文献［DUE92］、［MAT94］、［SUL94］和［TAY95］。在最低抽象层次的 OOA 着重于面向对象软件工程的一般的范围，将在本章其他几节重点讨论。在本节，我们集中讨论在中间抽象层次进行的 OOA，该活动称为领域分析，在某组织希望创建可以广泛地用于整个应用范畴的可复用类(构件)库时进行。

20.2.1 复用和领域分析

对象技术通过复用产生杠杆作用。考虑一个简单的例子，对一个新应用的需求分析指明需要 100 个类，两个项目组被委派去实现该应用，各自将设计和构造一个最终产品，每个组由具有相同的技能级别和经验的人构成。

组 A 不访问类库，这样它必须从头开发所有 100 个类；组 B 使用一个强健的类库并从中找到 55 个类，则最可能发生的事情是：

1.组 B 将比组 A 快得多地完成项目。

2.组 B 的产品成本将大大低于组 A 的产品成本。

3.组 B 的产品将比组 A 的产品有更少的错误。

虽然对组 B 的工作将超出组 A 的工作的利润差数仍然存在争论，但很少有人会否认复用为组 B 带来了实质性的优势。

但是“强健的复用类库”从何而来？并且如何确定在库中的项是否适用于新应用中？为了回答这些问题，创建和维护库的组织必须采用领域分析方法。

20.2.2 领域分析过程

Firesmith [FIR93] 以如下方式描述了领域分析：方法。

软件的领域分析是在特定应用领域中标识、分析和规约公共需求，典型地是在应用领域中多个项目间的复用。面向对象领域分析是以公共对象、类、子集合和框架等形式在特定应用领域中标识、分析和规约公共的可复用的能力。

“特定应用领域”可以涵盖从航空电子设备到银行，从多媒体视频游戏到计算机 X 射线轴向分层造影扫描机(CAT scanner)中的软件。领域分析的目标是直接明了的：发现或创建那些可广泛应用的类，使得它们可以被复用。

使用在本书前面引入的行话，领域分析可以被视为软件过程的一个全程活动，其含义是指领域分析是一个进行中的软件工程活动，它不和任何的软件项目相联系。在某些方面，领域分析的角色类似于制造环境中的工具制造者，工具制造者的工作是设计和制造可用于很多相似工作(但不一定是相同的工作)的工具。领域分析员的工作是设计和建造可复用构件，它们可以用于很多相似的(但不一定是相同的)应用开发工作。

图 20－1 [ARA89] 指明了领域分析过程的关键输入和输出。考察领域知识源以试图标识可在领域中指明复用的对象，在本质上，领域分析和知识工程相当类似，知识工程师调查特定的兴趣域，试图抽取出可用于创建专家系统或人工神经网络的关键事实。在领域分析过程中，抽取出对象(和类)。

领域分析过程可以用起始于标识被调查的领域到刻划领域的对象和类的规约的一系列活动来表示。Berard [BER93] 建议了下列活动：

定义将被调查的领域。为了完成此工作，分析员必须首先隔离感兴趣的业务范围、系统类型或产品范畴。接着，必须从中抽出 OO 和非 OO 的“项”。OO 项包括：现存 OO 应用的类的规约、设计和代码；支持类(如，GUI 类或数据库访问

类)；和领域相关的 COTS(commerical off－theshelf)构件库以及测试用例。非 OO 项包括：政策、步骤、规程、计划、标准和指南；现存非 OO 应用的部件(包括规约、设计和测试信息)；度量；以及 COTS 非 OO 软件。

分类将从领域中抽取出来的项分类。这些项被分类，并且定义种类的一般定义特征。提出种类的分类模式，并为每个项定义命名惯例。在合适的时候，建立分类层次。

收集领域中应用的代表性样本。为了完成该活动，分析员必须保证正被讨论的应用包括满足已被定义的种类的项。Berard［BER93］提到在使用对象技术的早期阶段，一个软件组织几乎没有任何 OO 应用，因此，领域分析员必须“在每个(非 OO)应用中标识概念的(相对物理的)对象”。

分析样本中的每个应用。［BER93］ 在领域分析中进行如下步骤：念的(相对物理的)对象”。

- 标识候选的可复用对象。
- 指明对象被标识为可复用的理由。
- 定义对对象的适应性修改(可能也是可复用的)。
- 估算在领域中可利用对象复用的应用的百分率。
- 用名字标识对象，并运用配置管理技术(第 9 章)来控制它们。

此外，一旦对象已被定义，要估算典型应用可使用该可复用对象来构造的百分率。

为对象开发分析模型。分析模型将作为设计和构造领域对象的基础。

除了以上步骤，领域分析员也应该创建一组复用指南并给出一个例子，以指说明如何应用领域对象来创建新的应用。

领域分析是被称为领域工程(第 26 章)的更大的学科中的第一项技术活动。当业务、系统或产品域被定义为长期的业务策略，则可以展开持续的创建强健的可复用库的工作，其目标是能够在领域中以非常高的可复用构件率来创建软件。低成本、高质量和改善的项目所需时间是支持领域工程的论据。

20.3 OO 分析模型的类属成分

面向对象分析过程遵从在第 11 章讨论的基本的分析概念和原则，虽然术语、符号体系和活动 OOA 与传统方法有所不同，但 OOA(在其核心)也强调相同的根本目标。Rambaugh 等［RAM91］有如下论述：

分析…涉及到建立真实世界的精确的、简明的、易理解的、和正确的模型…。面向对象分析的目的是以可理解的方式模拟真实世界。为了达到这个目标，必须检查需求，分析含义，并重新严格地加以陈述。必须首先抽象出真实世界的特征，而将小的细节推迟到以后考虑。

为了开发“真实世界的精确的、简明的、易理解的和正确的模型”，软件工程师必须从一系列OOA符号体系和过程中选择一种。如在 20.1.2 节中所说，每种OOA方法(存在许多OOA方法)有独特的过程和不同的符号体系，然而，所有方法均遵从一定的类属过程步骤(20.1.2 节)，并且均提供了实现一组OO分析模型[①]的类属成分的符号体系。

Monarchi 和 Puhr［MON92］定义了一组出现在所有 OO 分析模型中的类属表示成分，静态成分在本质上是结构性的，它指明了在应用的整个运行生命期中不变的特征，这些特征区分一个对象和其他对象。动态成分关注于控制并且对时序和事件处理敏感，它们定义了对象如何和其他对象在一定时间区间内交互。［MON92］标识出如下的模型描述成分：

语义类的静态视图。典型的类的分类法如第 19 章所述，作为分析模型的一部分评价需求并且抽取(并表示)类。这些类在整个应用的生命期内存在并且是基于客户需求的语义而被导出的。

属性的静态视图。每个类必须被显式地描述，和类关联的属性对描述类并且暗示了和类相关的操作。

关系的静态视图。对象以一系列方式相互“联接”，分析模型必须表示出这些关系以便于标识操作(它们影响这些连接)及完成消息序列的设计。

行为的静态视图。上面所述的关系定义了一组适应于系统的使用场景(使用实例)的行为，通过定义完成这些行为的一系列操作来实现。

通信的动态视图。对象间必须相互通信，并且基于导致系统从一个状态过渡到另一个状态的一系列事件来完成通信工作。

控制和时序的动态视图。必须描述导致状态变迁的事件的性质和时序。

如图 20－2 所示， de Champeaux 及其同事［CHA93］定义了一个稍有不同的 OOA 表示视图，针对对象内部和对象之间的表示来标识静态和动态成分。对象内部的动态视图可被刻划为对象生命历史(object life history)，即，当对对象的属性执行各种操作，对象随时间发生的状态变迁。

	对象内部	对象之间的表示
静态成分	类属性操作	对象关系状态变迁
动态成分	对象生命历史	通信时序

图 20-2 OOA 表示［ CHA93 ］通信时序

20.4 OOA 过程

OOA 过程并不是从考虑对象开始，而是从理解系统的使用方式开始，如果系统是人机交互的，则考虑被人使用的方式；如果系统是涉及过程控制的，则考虑被机器使用的方式；或者如果系统协调和控制应用，则考虑被其他程序使用的方式。定义了使用场景后，即开始软件的建模过程。

下面的几节定义一系列用于收集基本的客户需求并定义面向对象系统的分析模型的技术。

20.4.1 使用实例

需求收集总是任何软件分析活动的第一步。如在第 11 章讨论的那样，需求收集可以采用 FAST 会议的形式，客户和开发者在一起定义基本的系统和软件需求。基于这些需求，软件工程师(分析员)可以创建一组场景，每个场景标识系统的一个使用序列。场景，经常称为使用实例 [JAC92]，描述系统将如何运作。

为了创建使用实例，分析员必须首先标识使用该系统或产品的不同类型的人(或设备)，这些参与者(actor)真实地代表了当系统运行时人(或设备)所扮演的角色，更正式地定义，参与者是存在于系统之外和系统或产品通信的任何事物。

重要的是注意参与者和用户并不是相同的概念；一个典型的用户可能在使用系统时扮演一系列不同的角色，而一个参与者表示一个外部实体类(经常，但并不总是人)，它只扮演一个角色。例如，某机器操纵者(用户)，和某制造设备的控制计算机交互工作，该设备包含了一组机器人和数值控制的机器。在仔细的研究需求后，控制计算机的软件需要四种不同的交互模式(角色)：编程模式、测试模式、监控模式和排除故障模式，因此，须定义四个参与者：程序员、测试员、监控者和故障排除者。在某些情况下，机器操纵者可以扮演所有这些角色，在其他情况，不同的人可能扮演不同的参与者的角色。

和 OO 分析模型的其他方面一样，并不是在第一次递进时标识所有参与者，有可能在第一次递进时标识出主要参与者 [JAC92]，当更多的认识了系统后再标识出次要的参与者。主要参与者交互工作以达到所需的系统功能并从系统导出所需要的收益，它们直接地、频繁地和软件一起工作。次要参与者用以支持系统，以使得主要参与者能够完成它们的工作。

一旦标识参与者后，使用实例就可以开发，使用实例描述了参与者和系统交互的方式。Jacobson [JAC92] 提出了一组应该在使用实例中回答的问题。参与者和系统交互的方式。

- 参与者执行的主要任务或功能是什么？
- 参与者将获取、生产或改变什么系统信息？

• 参与者是否必须通知系统关于外部环境的改变？

• 参与者希望从系统中得到什么信息？

• 参与者是否希望得到关于未预料的改变的通知？

通常，使用实例仅仅是一段关于参与者和系统发生交互时所处的角色的描述。

用在前几章讨论的 SafeHome 安全系统来说明如何开发使用实例。回忆系统的基本需求，我们可定义三个参与者：房主(用户)、传感器(连接于系统的设备)以及监控和反应子系统(监控中心)。在此，我们仅考虑房主参与者，房主和产品以以下一系列不同的方式交互：

• 输入密码以允许所有其他交互进行。

• 查询安全区域的状况。

• 查询传感器的状况。

• 在紧急情况时按下紧急按钮。

• 启动/关闭安全系统。

下面是系统启动的使用实例：

1.房主观察 SafeHome 系统的控制面板(图 11－4)以确定系统是否已准备好接收输入，如果未准备好，房主必须关闭窗户/门，以使系统就绪。(未准备好的指示器表明某传感器是开着的，即，某道门或窗户是开着的。)

2.房主使用键盘键入四位密码，和存放在系统中的合法密码比较，如果密码不符，控制面板将鸣叫一次并复位等待再次输入。如果密码正确，控制面板等待进一步的动作。

3.房主选择并键入 stay 或 away(见图 11－4)以启动系统，stay 仅仅启动外围传感器(不启动内部的运动检测传感器)，away 启动所有传感器。

4.当启动进行时，房主可以观察到一个红色警灯。

以类似的方式开发其他的房主交互的使用实例。要注意，必须仔细地复审每个使用实例。如果交互的某些元素是含混不清的，对使用实例的复审将可能指出问题。

每个使用实例提供了参与者和软件间交互的明确的场景，它也用于刻划时序需求或对场景的其他约束。例如，在上面提到的使用实例中，需要在按 stay 或 away 键 30 秒后，启动系统。可将该信息附加到使用实例中。

使用实例描述了对不同参与者产生不同感觉的场景，Wyder［WYD96］建议可以使用质量功能部署(quility function deployment，QFD[①])来为每个使用实例确定加权优先级，为了完成此工作，对使用实例以系统定义的所有参与者的观点进行评估，每个参与者[②]对每个使用实例赋予不同的优先级(例如，值从 1 到 10)。然后计算平均优先级，指明每个使用实例的主观的重要性。当将迭代或增量过程模型使用于面向对象软件工程时，优先级将影响先交付哪个系统功能。

20.4.2 类—责任—协作者建模

一旦系统的基本使用场景确定后，则要开始标识候选类并指明它们的责任和协作，类—责任—协作者(Class－responsibility－collaborator，CRC)建模［WIR90］提供了一种简单的标识和组织与系统或产品需求相关的类的手段。Ambler［AMB95］对 CRC 建模的描述如下：

CRC 模型实际上是一组表示类的标准的索引卡片的集合。卡片被分成三个部分，在卡片的顶部为类的名字，在卡片体的左边列出类的责任，在右边列出协作者。

实际上，CRC 模型可以使用真实的或虚拟的索引卡片，其目的是开发一个有组织的类的表示法。责任是和类相关的属性和操作，简单地说，责任是“类知道或做的任何事情”［AMB95］。协作者是为某类提供完成责任所需要的信息的类，通常，协作蕴含着对信息的请求或对某种动作的请求。

类

在第 19 章已讨论过标识类和对象的基本指南。总的来说，对象以一系列不同的形式展示出来(19.3.1 节)：外部实体、事物、发生的事情或事件、角色、组织单位、位置或结构。在软件问题的语境内标识这些对象的一种技术是对系统的过程叙述进行语法分析，将所有名词变成潜在的对象，然而，并非每个潜在对象都会成为最终对象。在第 19 章中定义了 6 条选择特征：保留的信息、需要的服务、多个属性、公共属性、公共操作以及基本的需求，只有满足所有这 6 条选择特征的潜在对象，才被考虑包含在 CRC 模型中。

Firesmith［FIR93］扩展了上述的类的分类法，提出了如下类型：求，只有满足所有这 6 条选择特征的潜在对象，才被考虑包含在 CRC 模型中。

设备类。模拟外部实体，如传感器、发动机和键盘等。

属性类。表示问题环境的某些重要性质(例如，在抵押贷款应用语境中的信用级别)。

交互类。模拟在其他对象间发生的交互(例如，购买或执照)。

此外，对象和类可以按以下特征进行分类：

有形性(tangibility)。类是表示了有形的事物(如，键盘或传感器)，还是表示了抽象的信息(如，某预期的输出)？

包含性(inclusiveness)。类是原子的(即，不包含任何其他类)，还是聚合的(包含至少一个被嵌套对象)？

顺序性(sequentiality)。类并发的(即，拥有自己的控制线程)，还是顺序的(被外面的资源控制)？

持续性(persistence)。类是短暂的(即，在程序运行中被创建和删除)、临时的(在程序运行中被创建，在程序终止时被删除)还是永久的(存放在数据库中)？

完整性(integrity)。类是易被侵害的(即，没有保护资源不受外界的影响)、还是被保护的(类加强对其资源访问的控制)？

基于上述类的分类，要对作为 CRC 模型一部分的“索引卡片”扩展以包含类的类型及其特征(如图 20－3)。

责任

在第 19 章也讨论了标识责任(属性和操作)的基本指南。总而言之，属性表示类的稳定特性，即为了完成客户规定的软件目标所必须保持的类的信息，一般可以从对问题的范围陈述中抽取出或通过对类的本质的理解而辨识出属性。可以通过对系统的过程叙述进行语法分析而抽取出操作，动词作为候选的操作，每个为类选择的操作展示了类的某种行为。

Wirfs-Brock 及其同事 [WIR90] 给出了将责任分配到类的 5 条指南：

1.应该平均地分布系统智能。每个系统均包含一定程度的智能，即系统知道什么会做什么。智能可以以一系列不同的方式在类间分布，“废物(dump)”类(几乎没有责任的类)作为“聪明(smart)”类(有很多责任的类)的服务提供者。虽然该方法使系统中的控制流程直接明了，但它也有一些缺点：

- 它将所有智能集中在一些类中，使修改更为困难。

- 需要更多的类从而导致需要更多的开发工作量。

因此，系统智能应该在应用的类之间平均分布。因为每个对象仅仅知道并做少量的事情(它们通常是被很好聚焦的)，从而系统的内聚性得到改善。此外，因为系统智能已经被分布在很多对象间，由于变化而引起的副作用将被减弱。

为确定系统智能是否已被平均分布，须评估每个 CRC 模型索引卡片上的责任列表，确定是否某个类具有特别长的责任列表，这表明存在智能的集中。此外，每个类的责任应该有相同的抽象层次，例如，在一个称为 checking account 的聚合类的操作列表中，有两个责任：balance－the－account 和

check-off-cleared-checks，第一个操作(责任)蕴含了复杂的数学和逻辑过程，第二个操作是一个简单的书记性活动。因为这两个操作在不同的抽象层次，Check-off-cleared-checks 应该包含在聚合类 checking account 中的类 Cheek－enty 的责任中。

2.责任应该尽可能通用性地加以陈述。这个指南指出通用性责任(属性和操作)应存在于类层次的高层(因为它们是类属的，它们适用于所有子类)。此外，多态(第 19 章)应该用来定义总体上适用于超类但在每个子类中有不同实现的操作。

3.信息和与其相关的行为应该存在同一类中。这体现了我们称为封装的 00 原则(第 19 章)，数据和操纵该数据的处理应该作为内聚的单元来封装。

4.关于一个事物的信息应该包含在单个类中，而不是分布在多个类中。单个类应该承担存储和操纵特定信息类型的责任，通常，该责任不应该由一组类分享。如果信息被分布，软件将变得更难于维护及测试。

5.当适当时候，在相关类间分享责任。在很多情况下，一系列相关对象必须同时展示相同的行为。作为一个例子，考虑必须显示下列对象的视频游戏：player、player－body、player－arms、player-head，每个对象均有自己的属性(例如，position、orientation、color、speed)，并且当用尸操纵游戏杆时，所有对象必须被更新和显示。因此责任 update 和 display 必须被上面的每个对象分享，当某些东西改变时，player 要知道所发生的改变并进行更新，它和其他对象协作以完成新的位置或方向，但是每个对象控制自己的显示。

协作者

类以以下两种方式之一完成它们的责任：(1)类使用它自己的操作去操纵它自己的属性，从而完成某一特定责任；(2)类可以和其他类协作。

Wirfs-Brock [WIR90] 及其同事对协作定义如下：

协作表示了为完成客户的责任，对客户服务器的请求。协作是在客户和服务器间合约的具体体现…。如果为了完成某责任，一个对象需要向其他对象发送任何消息，则我们说该对象和另一个对象协作。单个协作流是单方向的——表示从客户到服务器的请求。从客户的观点来看，它的每个协作是和服务器实现某特殊责任相关联的。

协作标识了类之间的关系。当一组类一起协作以完成某需求时，可将它们组织为子系统(这是一个设计问题)。

通过确定类是否可以自己完成每个责任来标识协作，如果不能，则它需要和另一个对象交互，因此，产生协作。

作为一个例子，考虑SafeHome系统[①]。作为启动过程的一部分(见20.4.1节中针对启动的使用实例)，控制面板(control panel)对象必须确定是否任一传感器是打开的，相应定义一名为determine－sensor-status的责任。如果传感器是打开的，控制面板对象必须将状况(status)属性设置为“未准备好”。可以从sensor对象获取传感器的信息。因此，仅当control panel和sensor协作时才可以完成责任determine-sensor-status。

为了帮助标识协作者，分析员可以检查类之间的三种不同的类属关系[WIR90]：(1)部分(is－part－of)关系，(2)获知(has－knowledge－of)关系，(3)依赖(depends－upon)关系。通过创建一个类-关系图(20.4.4节)，分析员开发出标识这些关系所必需的连接。在下面的篇幅中，概略地讨论三种类属关系。

所有是某聚合类的一部分的类通过is－part－of关系同该聚合类连接。考虑前面提到的为视频游戏定义的类,类player－body同类player是is－partof关系，player－arms、player－legs和player－head与player间也是同样的关系。

当一个类必须从另一个类获取信息时，就建立has－knowledge－of关系。前面提到的determine－sensor－status责任是has－knowledge－of关系的一个例子。

depends－upon关系表示着两个类间存在由has－knowledge－of或is-part－of所不能完成的依赖关系。例如，player－head必须总是和player－body相连接(除非该视频游戏是特别暴力的),但每个对象仍可以在不直接知道另一个对象的情况下存在。player-head对象的一个称为center-position的属性，将根据player－body对象的center－position属性来确定,该信息是通过第三个对象player从对象player－body获取的，因此，player－headdepends－uponplayer－body。

在所有情形,将协作者类名记录在CRC模型的索引卡片上衍生出该协作的责任的旁边，因此，索引卡片包含一组责任和使得责任能够被完成的相应的协作。

当已建好了一个完整的CRC模型后,来自客户和软件工程组织的代表可以使用下面的方法[AMB95]遍览该模型：

1.给所有参加(CRC模型)复审的人一个CRC模型索引卡片的子集，有协作关系的卡片要分开(即，没有任何复审者持有两张有协作关系的卡片)。

2.应该将所有使用实例场景组织为种类。

3.复审的负责人仔细地阅读使用实例，当复审负责人遇到一个命名对象时，他将令牌传送给持有对应的类索引卡片的人员。例如，在20.4.1节中讲述的使用实例包含以下叙述：

1.房主观察 SafeHome 控制面板(图 11－4)以确定系统是否已准备好接收输入，如果尚未准备好，房主必须关闭窗户/门，以使系统就绪。(未准备好的指示器表明某传感器是开着的，即，某道门或窗户是开着的。)

当复审负责人在 we case 叙述中遇到“控制面板”，则将令牌传送给持有“控制面板”卡片的人。短语“表明某传感器是开着的”需要一个包含可确认此状态的责任(责任 determine－sensor－status 达到此目的)的索引卡片。在索引卡片上紧靠责任的地方是协作者 sensor，则令牌又被传送给 sensor 对象的索引卡片。

4.当传送令牌时，类卡片的持有者要描述卡片上记录的责任，小组将确定是否一个(或多个)责任满足使用实例需求。

5.如果索引卡片上的责任和协作不能适应使用实例，则需对卡片进行修改，这可能包括定义新类(及相应的 CRC 索引卡片)或在现存卡片上刻划新的或修订的责任或协作。这种做法持续至使用实例完成。

CRC 模型是第一个关于 OO 系统的分析模型的表示，可以通过进行从系统导出的被使用实例所驱动的复审来进行测试。

20.4.3 定义结构和层次

一旦已经使用 CRC 模型标识了类和对象，分析员开始关注类模型的结构及由类和子类所引致的类层次。Coad 和 Yourdon [COA91] 建议应该从已标识的类中导出其一般—特殊(gen－spec)结构。

例如，考虑为 SafeHome 定义的 sensor 对象，如图 20－4 所示。这里，一般类 sensor 被细分为一组特殊类-entry sensor、smoke sensor 和 motion sensor。我们可以建立一个简单的类层次。

在其他情形，一个在初始模型中表示的对象可能实际上是由一组成员部件(它们本身可能被定义为对象)构成，这些聚合的对象可以表示为整体一部分(whole－part)结构 [COA91]，并用在图 20－5 中的标志法来定义。三角表示组装关系。要注意，可以用附加的符号(图中未显示)来增强连接线以表示基数。这些是从第 12 章讨论的实体—关系建模符号借用过来的。

结构表示为分析员提供了划分 CRC 模型并图形地表示划分的工具，每个类的扩展为复审和后继的设计提供了所需的细节。

20.4.4 定义主题和子系统

复杂系统的分析模型可以包含成百个类和许多个结构，为此，有必要定义一种简洁的表示法，它是前面描述的 CRC 和结构模型的摘要。

当类的某个子集相互协作以完成一组内聚的责任时，它们常被称为主题(subjects)［COA91］ 或子系统［WIR90］。主题和子系统都是一种抽象，提供了指向分析模型中更细节内容的引用或指针。当从外界观察时，主题或子系统可被视为黑盒子，它包含了一组责任并且有自己的(外界)协作者。子系统和其外界协作者间实现一份或多份合约(contract)，一份合约是协作者可以对子系统[①]进行的一组特定请求的清单。

子系统可以在CRC建模的语境内通过创建一个子系统索引卡片来表示。子系统索引卡片标明子系统的名字、子系统必须服务的合约以及支持合约的类或(其他)子系统。

主题在目的和内容上和子系统是相同的，但是主题可被图形化地表示，例如，假定SafeHome的控制面板远远比图20－5所示的控制面板复杂，包含了多个显示区域、复杂的键盘安排以及其他特性，它可被以如图20－6所示的整体—部分结构的方式建模。如果整体需求模型包含许多这种的结构(SafeHome不存在这种情况)，要一次给出完整的表示是困难的。通过定义如图20－6所示的主题引用，整个结构可以通过单个图符(矩形)来加以引用。通常对多于5到6个对象的任何结构创建主题引用。

在最抽象的层次，OOA模型将只包含类似图20－7顶部所示的主题引用，每个引用将被扩展为一个结构。控制面板(control panel)和传感器(sensor)对象的结构(图20－6和图20－4)已经给出，如果system、sensor event和audible alarm对象需要多于5到6个的分类或组装对象，则也可创建相互的结构。

在图20－7顶部所示的双向箭头表示在主题引用内部的对象间通信(消息)的路径，这是作为在下一节所述的建模活动的结果而导出的。

20.5 对象—关系模型

在前几节中使用的CRC建模方法已经建立了类和对象关系的首要因素。建立关系的第一步是理解每个类的责任，CRC模型索引卡片包含了一系列责任。第二步是定义那些有助于完成责任的协作者类，这建立了类间的“连接”。

关系存在于任意两个相连接[①]的类之间，因此，协作者总是以某种方式相关的。最常见的关系类型是二元关系——在两个类之间存在的连接。当在OO系统的语境内讨论时，二元关系有确定的方向[①]，这是根据哪个类扮演客户角色及哪个类作为服务器而定义的。

Rambaugh及其同事［RAM91］建议可以通过检查对系统的范围或使用实例的陈述中的动词或动词短语而导出关系。使用语法分析，分析员分离出如下动词：指明物理位置的动词(nextto、part of、contained in)、指明通信的动词(transmits to、acquires from)、指明所有权的动词(incorporated by、is composed of)以及指明条件满足的动词(manages、coordinates、controls)，这些提供了对关系的暗示。

对对象的关系模型已经提出了一系列不同的图形符号(如参考文献[COA91]、[RAM91]、[EMB92]、[BOO94])，虽然各自使用自己的符号体系，但所有均是自第 12 章讨论的实体—关系建模技术演变而来。本质上，对象通过指定的关系和其他对象连接。规定连接的基数(cardinality)(见第 12 章)并建立整体的关系网络。

对象—关系模型(如实体—关系模型一样)可通过以下三个步骤导出：

1.利用 CRC 索引卡片，可以画出协作者对象的网络。图 20－8 表示了 SafeHome 对象间的类连接，首先对象被用无标记的线连接(图中未显示)，这表示在被连接的对象之间存在某种关系。

2.复审 CRC 模型索引卡片，评估责任和协作者，命名未标记的连接线。为了避免含混不清，用箭头指明关系的方向(图 20－8)。

3.一旦已经建成命名的关系，对每个端评估以确定基数(图 20－8)。存在四种选项：0—1、1—1、 0—多、或 1—多，例如，SafeHome 系统包含一个的控制面板(1：1 基数符号指明这个关系)，至少有一个被控制面板检测的传感器，然而，可能存在很多传感器(1：m 符号指明这个关系)，一个传感器可以识别从 0 到很多传感器事件(如，烟雾被检测到或闯入事件发生)。

持续进行以上步骤，直至得到一个完全的对象—关系模型。

通过建立对象—关系模型，分析员为整体的分析模型增加了另一维。不仅标识了对象之间的关系，而且定义了所有重要的消息路径(第 19 章)。在对图 20－7 的讨论中，我们提到了连接主题符号的箭头，这些也是消息路径，每个箭头蕴含了模型中子系统之间的消息交换。

20.6 对象—行为模型

CRC 模型和对象—关系模型表示了 OO 分析模型中的静态元素，现在我们转向讨论 OO 系统或产品的动态行为。为达到此目标，我们必须将系统的行为表示为特定事件和时间的函数。

对象—行为模型指明 OO 系统如何相应外部事件或激励。为了创建该模型，分析员必须完成下面几个步骤的工作：

1.评估所有的使用实例(20.4.1 节)以完全地理解系统中交互的序列。

2.标识驱动交互序列的事件，理解这些事件如何和特定的对象相关联。

3.为每个使用实例创建事件轨迹［RAM91］。

4.为系统建造状态—变迁图。

5.复审对象—行为模型以验证精确性和一致性。

上面的每个步骤将在以下节中讨论。

20.6.1 用使用实例标识事件

如在 20.4.1 节中所述，使用实例(use case)表示了涉及参与者和系统的一系列活动。通常，当 00 系统和参与者(记住，参与者可以是人、设备甚或外部系统)交换信息时，引发一个事件。回忆在第 12 章的讨论，重要的是注意到事件是布尔量，即，事件不是已经被交换的信息，而是信息已经被交换这样一个事实。

为指出信息交换检查使用实例，例如，回忆在 20.4.1 节中描述的使用实例：

1.房主观察 SafeHome 控制面板(图 11－4)以确定系统是否已准备好接收输入，如果系统未准备好，房主必须地关闭窗户/门，以使系统就绪。(未准备好的指示器表明某传感器是开着的，即，某道门或窗户是开着的。)

2.房主使用键盘键入四位密码，和存放在系统中的合法密码比较，如果密码不符，控制面板将鸣叫一次并复位等待再次输入。如果密码正确，控制面板等待进一步的动作。

3.房主选择并键入 stay 或 away(见图 11-4)以启动系统，stay 仅仅启动外围传感器(不启动内部的运动检测传感器)，away 启动所有传感器。

4.当启动进行时，房主可以观察到一个红色警灯。

上面给出的使用实例场景中带下划线的部分表示事件。应该对每个事件标识一个参与者，给出被交换的信息并且指明任何条件和约束。

作为典型事件的例子，考虑有下划线的使用实例短语房主使用键盘键入四位密码。在 00 分析模型的语境内，参与者 homeowner 发送一个事件给对象 control panel，这个事件可被称为 password entered；传送的信息是构成密码的四位数字，但这不是行为模型的本质构成部分。要注意：某些事件对使用实例的控制流有直接的影响，而其他一些事件对控制流无直接影响。例如，事件 password entered 对使用实例的控制无直接改变，但是事件 compare password(从将密码与存在系统中的合法密码进行比较的交互过程中导出)的结果将对 SafeHome 软件的信息和控制流有直接影响。

一旦标识完所有事件，它们即被分配给所涉及的对象。参与者(外部实体)和对象负责生成事件(例如，homeowner 生成 password entered 事件)或识别发生在其他地方的事件(例如，controlpanel 识别 compare password 事件的二元结果)。

20.6.2 状态表示

在 OO 系统语境内，须考虑两种不同的状态特征：

• 当系统执行其功能时每个对象的状态。

• 当系统执行其功能时从外界观察到的系统的状态。

一个对象的状态有被动(passive)和主动(active)两种特征 [CHA93] 。被动状态仅是简单的对象所有属性的当前状况，例如，聚合对象 player(在前面讨论的视频游戏应用中)的被动状态包括 player 的当前位置(position)和方位(orientation)(对象属性)以及和游戏相关的其他特征(如，指明 magic wishs remaining 的属性)。主动状态是当对象经历连续的变换或处理时的当前状况，对象 player 可能有如下主动状态：moving、at rest、injured、being cured、trapped、lost 等等。必须发生某一事件(有时称为触发)来促使对象从一个主动状态过渡到另一个主动状态。对象—行为模型的组成成分仅仅对对象的主动状态及促使对象于不同的主动状态间转变的事件(触发)的简单表示。图 20－9 给出了在 SafeHome 系统中 controlpanel 对象的主动状态的简单表示。

图 20－9 中的每个箭头代表了从对象的一个主动状态到另一个主动状态的转变，每个箭头的标记代表触发转变的事件。虽然主动状态模型提供了对对象的“生命历史”的有用的深入了解，但仍有可能刻划其他附加信息以更深入地理解对象的行为。除了刻划导致转变发生的事件外，分析员还可以规定保护条件(guard)和动作(action) [CHA93] 。保护条件是一个布尔变量是转变要发生所必须满足的前提条件，例如，图 20－9 中从“at rest”状态到“comparing”状态的转变的保护条件可以通过检查使用实例来确定：

如果(password input=4 digits)那么发生到 comparing state 的转变；

通常，转变的保护条件依赖于对象的一个或多个属性的值，换句话说，保护条件依赖于对象的被动状态。

动作和转变并发或作为转变的结果而发生，并且通常涉及对象的一个或多个操作(责任)，例如，和 password entered 事件相(图 20－9)连接的动作是输入 password 对象并逐位进行比较以校验输入的密码的操作。

OOA 行为表示的第二种类型考虑对整体产品或系统的状态表示，它包含一个用以指明事件如何引起从对象到对象的转变的简单的事件轨迹模型 [RAM91] ，以及一个描述每个对象的处理行为的状态转变图。

一旦已经为使用实例标识出事件后，分析员要创建一个对事件如何导致从一个对象到另一个对象的流的表示，称为事件轨迹(event trace)，该表示是使用实例的简化版本，它表示了引致行为从对象流向对象的关键对象和事件。

图 20－10 给出了 SafeHome 系统的部分事件轨迹，每个箭头表示一个事件(从某使用实例导出)并指明事件如何在 SafeHome 对象之间引导行为。第一个事件 system ready 是从外部环境导出的并引导行为到 homeowner，房主输入密码，事

件 initiates beep 和 beep sounded 指明如果密码不符时如何引导行为，合法的密码导致返回 homeowner。其余的事件和轨迹遵从系统被启动或关闭时的行为。

一旦确定完全的事件轨迹后，所有引致系统对象间的转变的事件可以被分为输入事件集和输出事件集(从一个对象)，这可以使用事件流图(event flow diagram)来表示[RAM91]。所有流入和流出某对象的事件以图 20－11 所示的方式标注。可以用状态—变迁图(第 12 章)来表示和每个类的责任相关联的行为。①

和第 12 章提出的结构化分析图一样，图形表示形成了 OO 分析模型的基础，并且为软件需求规约的创建打下了良好基础。

20.7 小结

面向对象分析方法使得软件工程师能够通过对对象、属性和操作(作为主要的建模成分)的表示来对问题建模。在文献中已经提出了大量的不同的面向对象分析方法，但是所有方法均有一组共同的特征：(1)类和类层次的表示；(2)对象—关系模型的创建；(3)对象—行为模型的导出。

面向对象系统的分析发生在很多不同的抽象层次，在业务或企业层，和 OOA 关联的技术可以同信息工程方法相结合，该技术通常称为领域分析；在应用层，对象模型着重于特定的客户需求，因为这些需求影响将被建造的应用。

OOA 过程从定义使用实例(描述 OO 系统如何被使用的场景)开始；然后应用类—责任—协作者(CRC)建模技术来为类和它们的属性与操作建立文档，这也提供了在对象间的协作的初始视图；再下一步是对象的分类和类层次的创建。可使用子系统(主题)可用于封装相关的对象，对象—关系模型提供了对象间如何相互连接的标记，而对象—行为模型指明了个体对象的行为和 OO 系统的整体行为。

思考题

20.1 选择 5 种面向对象分析方法和结构化分析方法(第 12 章)并用 Fichman 和 Kemerer [FIC92] (20.1.1 节)提出的建模维数对它们进行比较。12 章)并用 Fichman 和 Kemerer

20.2 为在 20.1.2 节中讨论的 OOA 方法之一准备一份讲课稿。结合一个简单的例子讨论该方法，并标注那些使用独特的符号或方法的地方。

20.3 对下列领域之一进行概略的领域分析：

a.大学学生成绩记录系统

b.联机服务

c.银行的客户服务

d.视频游戏开发

e.老师建议的应用领域

分离出那些可以被领域中一组应用所使用的类。

20.4 用你自己的话描述 OO 系统的静态和动态视图间的不同。

20.5 为在本书中讨论的 SafeHome 系统写出使用实例，使用实例应该强调定义安全区域所需的场景。安全区域包含一组可以作为整体而不是以个体方式来定位、启动和关闭的传感器。可以定义多达 10 个安全区域。以发挥你的创造力，但是仍需在本书前面定义的控制面板的范围内。

20.6 为在思考题 12.13 中引入的 PHTRS 系统开发一组使用实例，你必须给出假设一组用户和系统交互的方式。

20.7 为表 20－1 或你的老师建议的产品或系统开发一组使用实例；对应用领域进行研究并和你的同学举行 FAST 会议(第 11 章)以确定基本需求(老师将帮助进行协调)。

表 20－1　为思考题 20.7 到 20.12 所建议的应用

在思考题 12.13 引入的 PHTRS 应用
个人计算机的字处理系统的软件
煤气涡轮机的实时测试监控系统

(续)

在思考题 12.13 引入的 PHTRS 应用
制造控制系统的软件
你选定的视频游戏的软件
电子表格应用
基于 PC 的 3D 图形包
老师建议的系统

20.8 为在思考题 20.7 中选择的产品或系统开发一组完全的 CRC 模型索引卡片。

20.9 和同学一起，对思考题 20.8 中开发的 CRC 模型索引卡片进行复审，作为复审的结果，加入了多少附加的类、责任和协作者？

20.10 为在思考题 20.7 中选择的产品或系统开发类层次。

20.11 为在思考题 20.7 中选择的产品或系统开发一组子系统。

20.12 为在思考题 20.7 中选择的产品或系统开发对象—关系模型。

20.13 为在思考题 20.7 中选择的产品或系统开发对象—行为模型。确保列出了所有事件，提供了事件轨迹，开发了事件流图，为每个类并定义了状态图。

20.14 用你自己的话，描述如何确定类的协作者。

20.15 你建议是什么策略来为一组类定义子系统？

20.16 在对象—关系模型的开发中，基数扮演了什么角色？

20.17 对象的被动状态和主动状态间有什么不同？

推荐阅读文献及其他信息源

已有很多书籍涉及面向对象分析。在选择书籍之前，读者应该首先选择适合自己需要的 OOA 方法，然后再选择关于该方法的书籍。对可用方法的很好的概述可见 [MON92] 和需要的 [FIC92] 。

对 OOA 的详细讨论可见文献 [BER93] ，[BOO94] ，[CHA93] ，[COA91] ，[FIR93] ，[JAC92] ，[RAM91] 和 [WIR90] 。此外， Shlaer 和 Mellor(Object－Oriented Systems Analysis：Modeling the World in States，Prentice－Hall， 1992)，Martin 和 Odell(Object－Oriented Analysis andDesign，Prentice－Hall，1992)，Goldstein 和 Alger(Developing Object－Oriented Software for theMacintosh，Addison－Wesley，1992)，和 Lorenz(Object－Oriented Software Development，Prentice－Hall，1993)等的书籍包含了有用的讨论。Yourdon 和 Argila(Case Studies in Object－OrientedAnalysis and Design，Prentice－Hall，1995)给出了两个现实的实例研究，详细讨论了 OOA 过程。Connell 和 Shafer(Object－Oriented Rapid Prototyping，Prentice－Hall，1994)以及 Krief(Using ObjectOriented Languages for Rapid Prototyping，Prentice－Hall，1996)对原型的 OO 技术进行了有意义的讨论。

和第 19 章相同的 Internet 信息源也可用于 OOA。Internet 新闻组 comp.object 是关于 OOA 方法和工具的信息、讨论和争论的有用的信息源。该新闻组的 FAQ 包含了 OOA 方法的概述以及到其他参考文献的指针，其网址如下：

http：//www.cs.cmu.edu/Web/Groups/AI/html/faqs/lang/oop/faq.html

有时可在 comp.software－eng 和 comp.specification 找到对 OOA 问题的讨论，可以找到 EdwardBerard 的 A Comparison of Object－Oriented Development Methodologies，可在下面网址选择“on－line documents”：

http：//WWW.toa.com

其他的关于 OOA 和对象技术的一般性信息可在下面网址找到：

http：//www.rational.com

很多关于 OOA 论文和书籍的参考文献可在下面网址找到：

http：//www.cera2.com/object.htm

关于面向对象分析的 WWW 文献的最新列表可在

http：//www.rspa.com找到。

① 在这，上下文中的“实体”或者指数据对象（在结构分析的情况中），或者指对象（在OOA情况中）。

② 这些方法及它们之间的区别的讨论已超出本书的范围，有兴趣的读者可以参阅erard [ER92] 和Graham [GRA94]，从中可见到详细的比较。

③ 通常，OOA方法以方法开发者的名字来命名，即使方法已有一个给定的名字或缩略词代号。

④ 下面描述的过程的多个步骤的许多详细含义将在第 20.3 和 20.4 节详细讨论。第 21 章中也将主要讨论这些方法中的每一步里提到的设计构件。

① 作者 [MON92] 还提供了 23 种OOA方法和它们如何查找这些构件的说明。

① 第 11 章中主要讨论QFD。

② 理论上，这些估算应该是由作为参与者的各个组织或商业功能代表未完成的。

① 关于SafeHome的各个细节定义参见第 19.3 章。

① 回顾一下与客户/服务器方法有关的分类原则，在此子系统是指服务器和外部与客户的协作者。

①“关系（Relationship）”术语的其他说法是“相关（association）”[RAM91]和“连接（connection）”[COA91]。

① 第 12 章的数据模型中关系的自然双向特性的区分是非常值得注意的问题。

第 21 章　面向对象设计

面向对象设计(OOD)将用面向对象分析(第 20 章)所创建的分析模型转变为将作为软件构造的蓝图的设计模型和传统软件设计方法不同，OOD 实现一个完成一系列不同的模块性等级的设计。主要的系统构件被组织为称为子系统的系统级“模块”，数据和操纵数据的操作被封装为对象——一种作为 OO 系统的构造块的模块形式。此外，OOD 必须描述属性的特定数据组织和个体操作的过程细节，这些表示了 OO 系统的数据和算法片，从而实现整体模块性。

面向对象设计的独特性在于其基于四个重要的软件设计概念——抽象、信息隐蔽、功能独立性和模块性(第 13 章)建造系统的能力。所有的设计方法均力图建造有这些基本特征的软件，但是，只有 OOD 提供了使设计者能够以较少的复杂性和折衷达到所有这四个特征的机制。

软件设计者的工作可能是使人畏缩的，Gamma [GAM95] 及其同事给出了对 OOD 的适当的精确的描述：

设计面向对象的软件是困难的，设计可复用的面向对象的软件更加困难。你必须找到适当的对象、以适当的粒度将它们转化为类的因子、定义类接口和继承层次以及建立它们之间的关键关系。你的设计应该针对于手边的问题，但也应足够通用化以适应将来的问题和需求。你也应避免重复设计，至少应使重设计减少到最小程度。有经验的面向对象设计者将告诉你虽然不是不可能在第一次就达到目标，但可复用的灵活的设计是困难的。在设计完成前，他们通常尝试复用几次，并每次做一些修改。

面向对象设计、面向对象编程和面向对象测试是构造 00 系统的活动，在本章，我们考虑其中的第一步。

21.1 面向对象系统的设计

在第 13 章，我们介绍了针对传统软件的设计金字塔的概念，四个设计层次——数据、体系结构、界面和过程——被一一定义和讨论。对面向对象系统，我们也可以定义一个设计金字塔，但是层次略有不同，如图 21－1 所示，00 设计金字塔的四个层次是：

子系统层。包含每个子系统的表示，这些子系统使得软件能够满足客户定义的需求，并实现支持客户需求的技术基础设施。

类和对象层。包含类层次，它们使得系统能够以通用化方式创建并不断逼近特殊需求，这层也包含了每个对象的设计表示。

消息层。包含使得每个对象能够和其协作者通信的细节，本层建立了系统的外部和内部接口。

责任层。包含针对每个对象的所有属性和操作的数据结构和算法的设计。

设计金字塔着重于特定产品或系统的设计，然而，应该注意，存在另一个设计“层”，该层形成了金字塔的基础。基础层着重于领域对象(在本章后面称为设计模式)的设计，领域对象通过提供对人机界面活动、任务管理和数据管理的支持，在建造 00 系统的基础设施方面扮演了关键角色。领域对象也可以被用于应用系统本身的设计。

21.1.1 传统方法和 00 方法

传统的软件设计方法使用清楚的符号和一组启发规则将分析模型映射为设计模型，回忆图 13－1，传统分析模型的每个元素被映射到设计模型的一个或多个层次内。和传统软件设计一样，00D使用数据设计(当属性被表示时)、接口设计(当开发消息模型时)以及过程设计(在操作的设计中)。然而，体系结构的设计是不同的，和使用传统软件工程方法导出的体系结构设计不一样，00设计并不展

示层次化的控制结构[①]，事实上，OO设计的“体系结构”更多地关心伴随着控制流程的对象间的协作。

虽然在传统设计模型和OO设计模型间确实存在相似性，我们已经选择重命名设计金字塔的层次以更精确地反应OO设计的本质。图 21－2 给出了OO分析模型(第 20 章)和将其导出[②]的设计模型之间的关系。

子系统设计通过考虑整体客户需求(用使用实例表示)和外部可观察的事件和状态(对象—行为模型)而导出的类和对象设计通过对包含在 CRC 模型中的属性、操作和协作的描述的映射得来的；消息设计由对象—关系模型导出；责任设计利用 CRC 模型中的属性、操作和协作导出。

Fichman 和 Kemerer [FIC92] 提出了可用来比较各种传统的和面向对象的设计方法的 10 种设计建模成分：

1. 模块层次的表示。

2. 数据定义的规约。

3. 过程逻辑的规约。

4. 端到端处理序列的指明。

5. 对象状态和转变的表示。

6. 类及层次的定义。

7. 将操作赋予类。

8. 详细的操作定义。

9. 消息连接的规约。

10. 独有服务的标识。

因为有很多传统的以及面向对象的设计方法，所以很难在两种方法间进行一般性的比较。然而，我们可以说，建模维数 5 到 10 是结构化方法(第 14 章)或其派生方法所不支持的。

21.1.2 设计问题

Bertrand Meyer [MEY90] 建议了 5 种标准来判断设计方法的模块化的能力，并将它们和面向对象设计相联系：

• 分解性(decomposability)——一种设施，设计方法利用它帮助设计者将一个大型问题分解为易于求解的子问题。

• 组装性(composability)——一种级别，当设计方法保证当程序构件(模块)一旦被设计和建造到此级别后，即可被复用去创建其他系统。

• 易理解性(understandability)——一种简单度，程序构件在此简单度下，不参考其他信息或其他模块即可被理解。

• 连贯性(continuity)——在程序中进行小的修改的能力以及使这些修改在仅仅一个或很少的几个模块中发生对应修改下展示自己的能力。

• 保护性(protection)——一个体系结构特征，它将在给定模块中发生错误时减少副作用的传播。

根据这些标准，Meyer [MEY90] 建议了可为模块化体系结构导出的 5 种基本设计原则：(1)语义模块单元；(2)很少的接口；(3)小的接口(弱耦合)；(4)显式的接口；以及(5)信息隐蔽。

对应于所用语言中的语法单元；模块被定义为语义模块单元(linguistic modular units)， [MEY90] ，即，所使用的程序设计语言应该有支持被直接定义的模块性的能力，例如，如果设　普叽唇丁桓蛰永　蹋　词故谴　车某绦蛏杓朴镅裕匚纾現 ortaran、C、Pascal)也可以实现它为一个语法单元，但是，如果定义一个包含数据结构和过程并将它们标识为单个单元的包(package)，则需要如 Ada 的语言(或其他面向对象语言)以便可直接在语言句情中表示这种类型的模块。

为了达到低耦合(在第 13 章中引入的设计概念)，模块间接口的数量应该最小化(很少的接口)并且在接口间传递的信息量也应该最小化(小的接口)。当模块通信时，它们应该以明显的和直接的方式进行(显式的接口)。例如，如果模块 X 和模块 Y 通过一个全局数据区域进行通信(在第 13 章我们称为公共耦合)，它们便违反了显式接口的原则，因为模块间的通信对外部观察者来说是不明显的。最后，当关于模块的所有信息对外界访问是隐蔽的时，我们达到了信息隐蔽的原则，除非该信息被特别地定义为“公共信息”。

在本节中提出的设计标准和原则可以被应用于任意设计方法(如，我们可以应用它们于结构化设计)，然而，如我们将看到，面向对象设计方法比其他方法能更高效地达到每个标准，并产生模块化的体系结构以允许我们更有效地满足每条模块性标准。

21.1.3 OOD 概述

在第 20 章我们给出了一组流行OOA方法的概述，每个方法都有对应的OOD过程，它衍生出一组设计表示和符号体系以使得软件工程师能够以一致的方式创建

设计模型。在下面篇幅中，概括介绍对应的OOD方法[①]，目的是提供方法作者[②]建议的OOD过程的概述。

Booch 方法

我们已在第 20 章中提到，Booch 方法［B0094］包含“微开发过程”和“宏开发过程”，微层次定义了可在宏过程的每一步被复用的一组设计任务。因此，演进的进程得以维持。对 Booch 的 OOD 微开发过程的概述如下：

体系结构计划：

- 将相似的对象聚集于独立的体系结构区间。
- 根据抽象层次对对象分层。
- 标识相关的场景。
- 创建设计原型。
- 通过将设计原型应用到使用场景来校验设计原型。

战术的设计：

- 定义领域独立的方针(即，管理操作和属性的使用的“规则”)。
- 为存储管理、错误处理和其他基础设施功能定义领域特定的方针。
- 开发一个描述这些方针的语义的场景。
- 为每个方针创建一个原型。
- 装备并精化原型。
- 复审每个方针以保证“它传播了它的体系结构设想”［B0094］。

发布计划：

- 根据优先级组织在 OOA 中开发的场景。
- 将对应的体系结构发布分配给场景。
- 增量地设计和构造每个体系结构发布。
- 根据需要调整增量发布的目标和进度。

Coad 和 Yourdon 方法

Coad 和 Yourdon 的 OOD 方法 [COA91]是通过研究“有效的面向对象设计者”如何完成他们的设计而开发的，该方法不仅强调应用而且强调应用的基础设施。对 Coad 和 Yourdon 的 OOD 过程概述如下：问题域部分：

- 组合所有的领域特定类。
- 为应用类设计适当的类层次。
- 当适当时，简化继承。
- 精化设计以改善性能。
- 开发和数据管理构件的接口。
- 当需要时，精化并加入低层的对象。
- 复审设计并审查对分析模型的任何增补。人机交互部分：
- 定义人员参与者。
- 开发任务场景。
- 设计用户命令层次。
- 精化用户交互序列。
- 设计相关的类和类层次
- 适当时集成 GUI 类。

任务管理部分：

- 标识任务的类型(如，事件驱动、时钟驱动)。
- 建立优先级。
- 标识作为其他任务的协调者的任务。
- 为每个任务设计合适的对象。

数据管理部分：

- 设计数据结构和布局。
- 设计管理数据结构所需的服务。

• 标识可以协助实现数据管理的工具。

• 设计适当的类和类层次。

Jacobson 方法

OOSE(object－oriented software engineering)[JAC92]的设计活动是 Jacobson 开发的专属的 Objectory 方法的简化版本，该设计模型强调对 OOSE 分析模型的可追踪性。对 Jacobson 的 OOD 过程概述如下：

• 适应性修改以使得理想化的分析模型适合现实世界环境的要求。

• 创建块(blocks)作为主要设计对象[①]。

定义块去实现相关的分析对象。

标识界面块、实体块和控制块。

描述在执行中块如何通信。

标识在块之间传送的刺激及它们的通信顺序。

• 创建一个显示了刺激如何在块间传送的交互图。

• 组织块为子系统。

• 复审设计工作

Rambaugh 方法

对象建模技术(OMT)[RAM91]包含了一个设计活动，它鼓励在两个不同的抽象级别上进行设计。系统设计着重于构造一个完全的产品或系统所需的构件的布局，对象设计强调个体对象的详细布局。对 Rambaugh 的 OOD 过程概述如下：

• 进行系统设计

划分分析模型为子系统。

标识由问题所指令的并发性。

将子系统分配到处理器和任务。

选择实现数据管理的基本策略。

标识全局资源及访问它们所需的控制机制。

为系统设计适当的控制机制。

考虑如何处理边界条件。

复审并考虑权衡。

• 进行对象设计

从分析模型选择操作。

为每个操作定义算法。

选择适应于算法的数据结构。

定义任意的内部类。

修订类组织以优化对数据的访问并改善计算效率。

设计类属性。

• 实现在系统设计中定义的控制机制。

• 调整类结构以加强继承性。

• 设计消息序列以实现对象关系(关联)。

• 包装类和关联为模块。

我们已在第 20 章提到，合并 Booch 和 Rambaugh 方法的工作正在进行中[BOO96][LOC96]。

Wirfs-Brock 方法

Wirfs-Brock [WIR90] 定义了技术任务的连续统一体，分析通过它无缝地过渡到设计。对 Wirfs-Brock 方法的设计相关的任务概述如下：

• 为每个类构造协议[①]。

精化对象间的合约为精化的协议。

设计每个操作(责任)。

设计每个协议(接口设计)。

• 为每个类创建设计规约。

详细描述每个合约。

定义私有责任。

为每个操作刻划算法。

标注特殊的考虑和约束。

• 为每个子系统创建设计规约。

标识所有被封装的类。

详细描述子系统是服务器的合约。

标注特殊的考虑和约束。

虽然这些 OOD 方法的术语和过程步骤各不相同，但是，整体的 OOD 过程基本是一致的。为了完成面向对象设计，软件工程师要执行下列的类属步骤：

• 以可实现的方式描述每个子系统。

将子系统分配到处理器和任务。

选择实现数据管理、界面支持和任务管理的设计策略。

为系统设计合适的控制机制。

复审并考虑权衡。

• 对象设计：

在过程级别设计每个操作。

定义任意内部类。

为类属性设计内部数据结构。

• 消息设计：

使用对象间的协作和对象—关系模型，设计消息模型。

• 复审设计模型并在需要时迭代。

要注意，本节讨论的设计步骤是递进的，即，它们可以被增量地执行，伴随着附加 OOA 活动，直至完成完整的设计。

21.2 OO设计模型的类属成分

有时很难明确区分面向对象分析(第20章)和面向对象设计[①]。本质上，面向对象分析(OOA)是一个分类活动，即分析问题力图确定在开发解决方案时可应用的对象类，同时确定对象关系和行为。面向对象设计(OOD)使得软件工程师能够确定从类中导出的对象，以及这些对象如何相互关联，此外，OOD描述了对象间的关系如何达到，行为如何实现以及对象间通信如何实现。

从分析到设计的类属过程流如图21-3所示。一旦已经开发完一个合理的完整的分析模型后，[②]软件工程师开始关注于系统的设计，这是通过描述要实现客户需求及实现需求所必需的支持环境所需的子系统的特征来完成的。

当子系统被定义后，它们必须在客户需求的整体语境内相互协调，哪个子系统负责什么客户需求？在OOA中定义的对象驻留在哪个子系统内？哪些子系统必须并发运行以及什么系统构件协调和控制它们？全局资源如何被子系统管理？

在子系统设计过程中，软件工程师必须定义四种重要的设计构件[COA91]：

- 问题域——那些直接负责实现客户需求的子系统；
- 人机交互——实现用户界面的子系统(包括可复用的GUI子系统)；
- 任务管理——负责控制和协调并发任务的子系统，这些任务可能被包装在一个子系统中或在不同的子系统间；
- 数据管理——负责对象的存储和检索的子系统。

这些类属构件的每一个都可能被用一系列类以及必备的关系和行为来建模(在OOA过程中)，此外，设计构件通过定义协议[WIR90]来实现，协议正式地描述每个构件的消息模型。

一旦已经定义了子系统，并且开始设计上述的每个构件，则重点开始转移到对象设计，在这个层次，CRC模型(第20章)的元素被转换为设计实现，实质上，进行图21-4中所示的转换。

21.3 系统设计过程

虽然很多研究者提出了OO系统设计的过程模型，但Rambaugh及其同事[RAM91]提出的活动序列是比较完善的处理方法之一，在21.1.3节中给出的概述中，定义了如下的设计步骤：

- 将分析模型划分为子系统。

• 标识问题本身的并发性。

• 将子系统分配到处理器和任务。

• 选择实现数据管理的基本策略。

• 标识全局资源及访问它们所需的控制机制。

• 为系统定义合适的控制机制。

• 考虑边界条件应该如何处理。

• 复审并考虑权衡。

在下面几节，将详细地讨论和这些步骤相关的设计活动。

21.3.1 划分分析模型

基本的分析原则之一(第 11 章)是划分。在 00 系统设计中，我们划分分析模型以定义类、关系和行为的内聚集合，这些设计元素被包装为子系统。

通常，子系统的所有元素共享某些公共的性质，它们可能均涉及完成相同的功能；它们可能驻留在相同的产品硬件中；或它们可能管理相同的类和资源。子系统由它们的责任所刻划，即，一个子系统可以通过它所提供的服务来标识［RAM91］。当应用在 00 系统设计的语境中时，服务是完成特定功能(如，管理字处理器文件、生成三维渲染、将模拟视频信号转换为压缩的数字图像)的一组操作。

在定义(和设计)子系统时，应该遵从下面的设计标准：

• 子系统应该具有良好定义的接口，通过接口和系统的其余部分通信。

• 除了少数的“通信类”，子系统中的类应该只和该子系统中的其他类协作。

• 子系统的数量不应太多。

• 可以在子系统内部划分以降低复杂性。

当两个子系统相互通信时，它们可以建立客户/服务器连接或端对端(peer-to-peer)连接［RAM91］。在客户 /服务器连接方式中，每个子系统只承担一个角色，服务只是单向地从服务器流向客户端。在端对端连接方式中，服务可以双向流动。

可以用数据流图(第 12 章)来表示上述的通信和信息流，在这种情形， DFD 中的每个“泡泡”便是一个子系统。

21.3.2 并发性和子系统分配

对象—行为模型的动态方面提供了对对象间(或子系统间)并发性的指示，如果对象(或子系统)不是同时活动的，则不需要并发处理。这意味着对象(或子系统)可以在同一个处理器硬件上实现。另一方面，如果对象(或子系统)必须同时异步地作用于事件，则它们被视为并发的，当子系统间是并发时，有以下两种分配方案：

- 将每个子系统分配到独立的处理器。
- 将子系统分配到相同的处理器并通过操作系统特性提供并发支持。

通过检查每个对象的状态图来定义并发任务［RAM91］，如果事件和转换流指明在任何时刻只有单个对象是活动的，则建立一个控制线程(thread of control)。即使当一个对象向另一个对象发送消息，只要第一个对象在等待回应，则控制线程一直持续。然而如果第一个对象在发送消息后继续处理，则控制线程分叉。

在 OO 系统中通过分离出控制线程来设计任务，例如，当 SafeHome 安全系统正在监控其传感器时，它也可以拨号到中心监控站以验证连接情况。因为涉及这两个行为的对象是同时活动的，每个表示一个独立的控制线程并且每个可被定义为独立的任务。如果监控和拨号活动顺序地发生，则可实现单个任务。

为了确定上述的哪种处理器分配方案是合适的，设计者必须考虑性能需求、成本和处理器间通信所带来的花销。

21.3.3 任务管理构件

Coad 和 Yourdon［COA91］建议了如下的设计管理并发任务的对象的策略：

- 确定任务的特征。
- 定义协调者任务和关联的对象。
- 集成协调者和其他任务。

通过理解任务如何初始化来确定任务的特征，事件驱动和时钟驱动任务是最常遇见的，二者均由中断激活，但是前者接收来自某些外部源的中断(如，另一个处理器、某传感器)，而后者由系统时钟控制。

除了任务初始化方式外，也必须确定任务的优先级和关键程度，高优先级任务必须能够立即访问系统资源，高关键度的任务即使在资源可用性减少或系统处于退化状态下时也必须能够继续运行。

一旦任务的特征已经确定后，就定义为完成和其他任务的协调和通信所需的属性和操作，基本任务模板(任务对象的)采用如下形式［COA91］：

任务名——对象的名字。

描述——对对象目的的叙述。

优先级——任务优先级(如：低、中、高)。

服务——一组作为是对象责任的操作。

由…协调——对象行为被激活的方式。

通过…通信——和任务相关的输入和输出数据值。

然后模板描述可被转换为任务对象的标准设计模型(属性和操作的综合表示)。

21.3.4 数据管理构件

数据管理包括两个不同的关注区域：(1)对应用本身关键的数据管理；(2)创建用于对象存储和检索的基础设施。通常，数据管理设计为层次的模式，其思想是分离操纵数据结构的低层需求和处理系统属性的高层需求。

在系统语境中，数据库管理系统常被用作所有子系统的公共数据仓库，操纵该数据库所需的对象是通过领域分析(第 20 章)标识的可复用类的成员或直接由数据库厂商提供。对OO系统的数据库设计的详细讨论超出了本书范围。[①]

数据管理构件的设计包括管理对象所需的属性和操作的设计，相关的属性被附加于问题域中的每个对象，并提供回答下列问题的信息：我如何存储自身？Coad 和 Yourdon［COA91］建议创建一个对象服务器(object-server)类，“其服务将(1)告知每个对象去存储自身，以及(2)检索被存储的对象以供其他设计构件使用”。

作为对 SafeHome 安全系统中讨论的 sensor 对象的数据管理的一个例子，设计将定义一个称为 sensor 的平坦文件，每个字段将对应 sensor 的一个指定的实例并且将包含该指定实例的每个属性的值。在 object-server 类中的操作将使得能够存储在系统需要时任一特定的对象。检索对更复杂的对象，可能必须定义一个全关系数据库或面向对象数据库来完成同样的功能。

21.3.5 资源管理构件

对 OO 系统或产品有一系列不同的有用资源，并且在很多情况下，子系统同时竞争这些资源。全局的系统资源可以是外部实体(如，磁盘驱动器、处理器或通信线)或抽象(如，数据库，对象)，不管资源的性质如何，软件工程师应该为其设计一个控制机制。Rambaugh 及其同事［RAM91］建议每个资源应该由某“保护者对象(guardian object)”拥有，保护者对象是该资源的门卫，控制对资源的访问并协调对资源请求的冲突。

21.3.6 人机界面构件

虽然人机界面(HCI)构件在问题域的语境内实现，但是，界面本身对大多数现代应用而言是一个非常重要的子系统。OO 分析模型(第 20 章)包含了使用场景(称为使用实例)和对用户在和系统交互时所扮演的角色(称为参与者)的描述，这些被作为 HCI 设计过程的输入。

一旦定义了参与者及其使用场景，则标识了一个命令层次。命令层次定义了主要的系统菜单类别(菜单条或工具调色板)以及在主要系统菜单类别(菜单窗口)内可用的所有子功能。命令层次被递进地精化，直至通过探索功能层次可实现所有 use case。

因为已经有大量的 HCI 开发环境(如 MacApp 或 Windows)，GUI 元素的设计不是必要的。对于窗口、图符、鼠标操作和大量的其他交互功能已有可以复用的类(具有合适的属性和操作)，实现者只需要针对问题域的要求实例化具有合适特征的对象即可。

21.3.7 子系统间通信

一旦已经定义了每个子系统后，有必要定义子系统间的协作关系。我们使用的“对象到对象协作”模型可被扩展到用于子系统，图 21-5 提供了协作的图形表示。如我们在本章前面所述，可以通过建立客户/服务器连接或端对端连接进行通信。如图所示，我们必须确定存在于子系统间的合约。如前所述，合约提供了一个子系统和另一个子系统交互的方式。

可运用下面的设计步骤来为子系统确定合约［WIR90］：

1.列出可以被子系统的协作者提出的每个请求，按子系统组织这些请求并在一个或多个合约中定义，确定已标注了从超类继承的合约。

2.对每个合约，标注实现该合约蕴含的责任所需的操作(继承的和私有的)，确定将操作和子系统内的特定类相关联。

3.一次考虑一个合约，创建如图 21-6 所示的表格，对每份合约，要创建如下的表项：

类型——合约的类型(即，客户/服务器或端对端)

协作者——作为合约伙伴的子系统的名字

类——支持合约蕴含的服务的类(包含在子系统中)的名字

操作——实现服务的操作(在类中)的名字

消息格式——实现协作者间交互所需的消息格式

对于子系统间的每个交互草拟一份合适的消息描述。

4.如果子系统间的交互模式是复杂的，则可以创建如图 21-7 所示的子系统协作图。协作图在形式上类似于第 20 章讨论的事件流图，同时表示出每个子系统及其与其他子系统之间的交互。在交互中被激活的合约如图中所标注，交互的细节通过查找在子系统协作表(图 21-6)中的合约而确定。

21.4 对象设计过程

借鉴在本书前面所述的比喻，OO 系统设计可被视为房子的平面图，平面图刻划了每个房间的用途，以及房间和房间、房间和外部环境间连接的机制。现在到了提供建造每个房间所需的细节的时候了。

在 OOD 的语境内，对象设计着重于“房间”。我们必须开发构成每个类的属性和操作的详细设计，以及连接类和其协作者的完整的消息规约。

21.4.1 对象描述

对象(类或子类的一个实例)的设计描述可以采用以下形式之一［GOL83］：

1.协议描述，通过定义对象可以接收的每个消息和当对象接收到消息后完成的相关操作来建立对象的接口。

2.实现描述，显示由传送给对象的消息所蕴含的每个操作的实现细节，实现细节包括关于对象私有部分的信息，即关于描述对象的属性的数据结构的内部细节及描述操作的过程细节。

协议描述仅仅是一组消息和对消息的注释，例如，对对象 motion sensor(前面所述)的协议描述的片断可能是：

MESSAGE(motion sensor)→read：RETURNS sensor ID，sensor status；

它描述了读传感器所需的消息。类似地，

MESSAGE(motion sensor)→set：SENDSsensorID，sensorstatus；

设置或复位传感器的状况。

对有很多消息的大型系统，一般有可能创建消息类别，例如，对 SafeHome 的 system 对象的消息类别可能包括系统配置消息、监控消息、事件消息等等。

对象的实现描述提供了内部的(“隐藏的”)细节，它是实现所需要的，但不是调用所必需的。即，对象的设计者必须提供一个实现描述并创建对象的内部细节，然而，使用该对象或该对象的某个其他实例的另一个设计者或实现者所需要的仅仅是协议描述而不是实现描述。

实现描述包含了以下信息：(1)对象的名字的定义和类的引用；(2)指明数据项和类型的私有数据结构的定义；(3)每个操作的过程描述或指向这样的过程描述的指针。实现描述必须包含用以对在协议描述中所描述的所有消息的适当处理的足够的信息。

Cox[COX85]用服务的“用户”和“提供者”来刻划包含在协议描述中的信息和包含在实现描述中的信息的不同，对象所提供的“服务”的用户必须熟悉调用该服务的协议，即，刻划想要什么。服务的提供者(对象本身)必须考虑如何将服务提供给用户，即，考虑实现细节。这就是在第 19 章引入“封装”的目标的概念。

21.4.2 设计算法和数据结构

包含在分析模型和系统设计中的一系列表示提供了对所有操作和属性的定义，使用与传统软件工程所讨论的数据设计和过程设计方法略有不同的方法来设计算法和数据结构。

创建算法以实现每个操作的规约，在很多情况下，算法是可以自我包含的软件模块来实现的简单的计算或过程序列，然而，如果操作的规约是复杂的，有可能必须将操作模块化，可运用传统的过程设计技术来达到此目的。

数据结构被和算法并行地设计。因为操作总是要操纵类的属性，所以，最好的反应了属性的数据结构设计将对相应操作的算法设计具有重要意义。

虽然存在很多不同的操作类型，它们通常可以分成三大类：(1)以某种方式操纵数据的操作(如，加入、删除、重格式化、选择)；(2)执行计算的操作；(3)为控制事件出现监控对象的操作。

例如，SafeHome 的过程化叙述包含了语句：“传感器被赋予一个编号和类型”和“主密码被编程以用于启动和关闭系统”，这两个语句指明了如下事情：

- 赋值(assign)操作和 sensor 对象相关。

• 编程(program)操作将被应用于 system 对象。

• 启动(arm)和关闭(disarm)是应用于 system 的操作，且系统状况可最终定义为(使用数据字典符号)。

system status＝ [armed| disarmed]

在OOA阶段分配操作program，但是在对象设计过程中它将被细分为一组更特定的配置系统所需要的操作。例如，在和产品工程师、分析员、也可能和市场部门讨论后，设计者可能精化初始的叙述并对编程(program)[①]操作描述如下：

编程(program)使得 SafeHome 用户能够在安装时配置系统，用户可以(1)设置(install)电话号码；(2)定义(define)警报延迟时间；(3)建造(build)一个它包含每个传感器 ID、其类型、以及其位置的传感器表；(4)装载(load)主密码。

即，设计者已将单个操作 program 精化为一组操作 install、define、build 和 load，这些新操作成为 system 对象的一部分，知道实现对象属性的内部数据结构，并且通过发送如下形式的消息而被调用：

MESSAGE(system)→install：SENDS telephone number;

它表明为系统提供一个紧急电话号码，install 消息将被发送给 system 对象。

动词意味着动作或事件，在进行对象设计形式化时，我们不仅考虑动词，而且考虑描述性动词短语和谓词(如，“等于”)作为潜在操作，递归地应用语法分析，直至全部操作已经被精化到最详细的层次。

一旦创建了基本对象模型后，应该进行优化。Rambaugh 及其同事 [RAM91] 建议了 OOD 优化的三个主要的切入点：

• 复审对象—关系模型以保证已实现的设计可带来对资源的高效使用并容易实现，必要时加入冗余。

• 修订属性数据结构和对应的操作算法以提高处理效率。

• 创建新的属性以存放导出的信息，以避免重复计算。

对 OO 设计优化的详细讨论已超出本书范围，有兴趣的读者可参见 [RAM91] 和 [DEC93] 。

21.4.3 程序构件和接口

软件设计质量的一个重要方面是模块性——即，对被用来组合以形成完整程序的程序构件(模块)的定义。面向对象的方法定义对象为和其他构件(如，私有

数据、操作)连接的程序构件，但是仅定义对象和操作是不够的，在设计过程中，我们还必须标识存在于对象间的接口和对象的整体结构(在体系结构的意义上考虑)。

虽然程序构件是一种设计抽象，但是它应该在实现该设计的程序设计语言的语境内被表示出来，为了适应 OOD，所用的程序设计语言应该能够创建下面的程序构件：

PACKAGE program-component-name IS

TYPE specification of data objects

•

•

•

PROC specification of related operations…

PRIVATE

data structure details for objects

PACKAGE BODY program-component-name IS

PROC operation.1(interface description) IS

•

END

PROC operation.n(interface description) IS

•

•

•

END

END program-component-name

在上面的类似 Ada 的 PDL(program design language)中，通过指明数据对象和操作来刻划程序构件。构件的规约部分指明所有的数据对象(用 TYPE 语句声

明)和作用于它们之上的操作(PROC 用于过程声明)，构件的私有部分(PRIVATE)提供数据结构和处理的隐藏的细节。我们在前面讨论的，PACKAGE 在概念上类似于本章讨论的对象。

被标识的第一批程序构件应该是最高层的模块，所有的处理发源于它们并且所有数据结构从其演化而来。再一次考察 SafeHome 例子，我们可以定义最高层的程序构件为：

PROCEDURE SafeHome software

SafeHome 软件构件可以和对下列的包(对象)的初步设计相结合：

```
PACKAGE system IS

TYPE system data

PROC install,  define,  build, load

PROC display,  reset,  query,  modify,  call

PRIVATE

PACKAGE BODY system IS

PRIVATE

system.id IS STRING LENGTH(8);

verification phone number, telephone number, …

IS STRING LENGTH(8);

sensor table DEFINED

sensor type IS STRING LENGTH(2),

sensor number, alarm threshold IS NUMERIC;

PROC install RECEIVES(telephone number)

 {design detail for operation install}

•

•
```

•

END system

PACKAGE sensor IS

TYPE sensor data

PROC read, set, test

PRIVATE

PACKAGE BODY sensor IS

PRIVATE

sensor.id IS STRING LENGTH(8);

sensor status IS STRING LENGTH(8);

alarm characteristics DEFINED

threshold, signal type, signat level IS NUMERIC,

hardware interface DEFINED

type, a/d.characteristics, timing.data IS NUMERIC,

END sensor

•

•

•

END SafeHome software

为 SafeHome 软件的每个程序构件定义数据对象和对应的操作，对象设计过程的最后一步是完成对完全实现包含在包的 PRIVATE 部分中的数据结构和类型所需的所有信息，以及包含在 PACKAGE BODY 部分中的所有过程细节。

为了阐明程序构件的详细设计，我们考虑上面描述的 sensor 包，因为已经定义了传感器属性的数据结构，所以，第一步是定义附于 sensor 对象的每个操作的接口：

PROC read(sensor.id, sensor status: OUT);

PROC set(alarm characteristics, hardware interface: IN);

PROC test(sensor.id, sensor status, alarm characteristics: OUT);

下一步要对和 sensor 包关联的每个操作逐步求精，为了说明求精过程，我们给出对 read 的过程叙述(一个非正式的策略)：

当传感器对象接收到 read 消息时，激活 read 进程，该进程确定接口和信号类型、查询传感器接口、转换 A/D 特征为内部的信号电平、并将内部信号电平和阀值比较，如果超出了阀值，传感器状况被设置为“事件”，否则，传感器状况被设置为“无事件”，如果在查询传感器时检测到错误，则传感器状况被设置为“错误”。

对上述过程描述，可以给出 read 处理的 PDL 描述：

```
PROC read(sensor.id, sensor.status: OUT);

raw.signal IS BIT STRING

IF(hardware.interface.type＝"S" &

alarm.characteristics.signal.type＝"B")

THEN

GET(sensor, exception: sensor status: ＝error)raw.signal;

CONVERT raw.signal TO internal.signal.level;

IF internal.signal.level＞threshold

THEN sensor status:  ＝"event";

ELSE sensor status: ＝"no event";

ENDIF

ELSE {processing for other types of s interfaces would be specified}

ENDIF

RETURN sensor.id, sensor status;

END read
```

read 操作的 PDL 表示可以被翻译为合适的实现语言，假定可在实时库中找到函数 GET 和 CONVERT。

21.5 设计模式

在任何领域的最好的设计者均具有这样的奇特能力，能够看见刻划问题的模式和可被组合以创建解决方案的对应的模式，Gamma 及其同事［GAM95］讨论该问题如下：

在许多面向对象系统中，你将发现类和通信对象的重复出现的模式。这些模式解决特定的设计问题，并使得面向对象的设计更灵活、优美并最终可复用。它们通过将新的设计基于以前的经验之上来帮助设计者复用成功的设计，熟悉这样的模式的设计者可以立即应用它们到设计问题中，而不需重新发现它们。

贯穿整个 OOD 过程，软件工程师应该去寻找每个复用现存设计模式的机会，并且，如果复用不可能时，则力图创建新的设计模式。

21.5.1 描述设计模式

成熟的工程学科使用成千的设计模式。例如，机械工程师使用一个两步键轴作为设计模式，该模式的固有性质是属性(轴的直径、键沟的维数，等)和操作(如，轴旋转、轴连接)；电子工程师使用集成电路(一个极端复杂的设计模式)来解决新问题的特定元素。所有设计模式均可以通过刻划四个信息而描述［GAM95］：

- 模式的名字。
- 模式通常被应用的问题。
- 设计模式的特征。
- 应用设计模式的结果。

设计模式名是它传达关于其适用性和意图的有意义的信息的一个抽象；问题描述指明使得设计模式可以被应用所必须存在的环境条件。模式特征指明设计中可被调整以使得模式能够适应一系列问题的属性，这些属性表示了可以被用来搜索(通过数据库)以找到合适的模式设计的特征；最后，和使用设计模式相关联的结果指明了设计决策的分叉。

应该仔细选择对象和子系统(潜在的设计模式)的名字。如我们在第 26 章中所讨论，软件复用中的关键技术问题之一就是不能在成百上千候选模式中找到合用的可复用模式。对“合适”模式的搜索可从有意义的棱式名字及一组帮助区分对象的特征［PRE95］中得到不可限量的帮助。

21.5.2 在设计中使用设计模式

在面向对象系统中，可以运用两种不同的机制来使用设计模式[①]：继承和复合。继承是基本的OO概念，在第 19 章中有详细描述，使用继承，现存的设计模式变成了新子类的模板，存在于模式中的属性和存在成为子类的一部分。

复合是导致聚合对象的概念，即，一个问题可能需要具有复杂功能的的对象(在极端情形，用子系统完成这些要求)，复杂的对象可以通过选择一组设计模式并复合适当的对象(或子系统)而被组装而成，每个设计模式被作为黑盒，在模式间的通信仅仅通过良好定义的接口进行。

Gamma 及其同事［GAM95］建议当两种选择并存时，对象复合应该优于继承。不是去创建大型的、有时不可管理的类层次(过分使用继承的结果)，复合采用针对一个目标的小的类层次和对象。复合以不修改的方式使用现存的设计模式(可复用构件)。

21.6 面向对象编程

虽然对象技术的所有区域均已经受到软件界的强烈关注，但是，还没有一个主题比面向对象编程(OOP)产生了更多的书籍、更多的讨论和更多的争论。已经有超过 100 本涉及 C++程序设计的书，成打的涉及 Smalltalk 的书籍，正在增多的涉及 Ada95 的书，及很多涉及并不广泛使用的 OO 语言的书籍。

软件工程的观点强调 OOA 和 OOD，而认为 OOP(编码)是重要的但只是第二位的活动，它是分析和设计的产物。理由纸简单，当系统复杂性增加时，最终产品的设计体系结构比使用的程序设计语言对其成功有更强的影响，然而，“语言战争”仍然风行。

OOP 的细节最好留给专门涉及这一主题的书籍，有兴趣的读者可以参阅在本章尾部的“推荐阅读文献及其他信息源”中列出的一本或多本 OOP 书籍。

21.7 小结

面向对象设计将现实世界的 OOA 模型转换为可以用软件实现的实现—特定的模型。OOD 过程可以被描述为有四个层次的金字塔，基础层着重于实现主要系统功能的子系统的设计；类层刻划了实现系统所需的整体对象体系结构和类层次；消息层指明如何实现对象间的协作；责任层标识用以刻划类特征的属性和操作。

和 OOA 一样，有很多不同的 OOD 方法。虽然各自互不相同，但所有均遵从设计金字塔，并且所有均通过两个层次的抽象完成设计过程——系统和子系统的设计以及个体对象的设计。

在子系统设计过程中，涉及四类构件：问题域构件、人机交互构件、任务管理构件和数据管理构件。问题域构件实现 OO 应用的客户需求，其他的构件提供了使得应用能够有效运行的设计基础设施。对象设计过程关注于对实现类属性的数据结构、实现操作的算法及支持协作和对象关系的消息的描述。

面向对象程序设计扩展设计模型到可执行域。使用 OO 程序设计语言将类、属性、操作和消息转换为可被机器执行的形式。

面向对象设计代表了一种独特的软件工程方法，引用 Tom Love [LOV85] 的话如下：

软件问题的根可能在于时我们的产业——数据处理的描述中的最传统的术语，我们被告知数据和数据处理是构成我们的业务基础的两种不同的“事物”。该划分所带来的危害可能远比我们认识到的多得多。

OOD 为我们提供了打破这种数据和处理间的“划分”的手段，这样做，可改善软件质量。

思考题

21.1 OOD 的设计金字塔和为传统软件设计描述的金字塔(第 13 章)略有不同，讨论两个金字塔的不同和相似之处。

21.2 OOD 和结构化设计如何不同？这两种设计方法在什么方面是相同的？

21.3 复审在 21.1.2 节讨论的有效模块性的 5 个标准，使用在本章后部描述的设计方法，展示如何达到的这 5 个标准。

21.4 选择在 21.1.3 节描述的 OOD 方法之一并为你的班级准备一小时的教程，确信显示了作者建议的所有图形的建模约定。

21.5 选择未在 21.1.3 节中描述的一种 OOD 方法(如 HOOD)，并为你的班级准备一小时的教程，确信显示了作者建议的所有图形的建模约定。

21.6 讨论使用实例如何可以作为设计的重要信息源。

21.7 研究某种 GUI 开发环境并显示在现实世界中如何实现人机交互构件。提供什么设计模式以及它们运用？

21.8 OO 系统的任务管理可能是相当复杂的，对实时系统的 OOD 方法(如参考文献 [BIH92])进行研究，并确定在该语境中，任务管理是如何完成的。

21.9 讨论在典型的 OO 开发环境中，数据管理构件是如何实现的。

21.10 撰写 2 到 3 页的关于面向对象数据库的论文，并讨论它们可如何被用于开发数据管理构件。

21.11 设计者如何识别必须并发的任务？

21.12 应用在本章中讨论的 OOD 方法来加强 SafeHome 系统的设计，定义所有相关的子系统并对重要类进行对象设计。

21.13 将本章中讨论的 OOD 方法应用于思考题 12.13 中描述的 PHTRS 系统。

21.14 描述一个视频游戏并应用在本章讨论的 OOD 方法来表示其设计。

21.15 你正负责一个用在 PC 网络上的电子邮件(e－mail)系统的开发工作，该 e－mail 系统将使得用户能够创建将被邮给另一个用户或邮给某特定地址表的信件，信件可被读、拷贝、存储等。该 e－mail 系统将使用现存的字处理能力来创建信件。使用这个描述作为起点，导出一组需求并应用 OOD 技术来创建该 e－mail 系统的顶层设计。

21.16 某小岛国决定为机场建造一个航空交通控制系统(airtrafficcontrol，ATC)，系统被刻划如下：

所有停放在机场的飞机必须装有异频雷达收发机，它将飞机类型和航班数据以高密度压缩的格式传送给 ATC 地面站。ATC 地面站可以查询飞机的特殊信息。当 ATC 地面站接收到数据，它将数据解包并存储到飞机数据库中。从存储的信息来创建一个计算机图形显示，并显示给航空交通控制员，该显示每 10 秒钟更新一次。分析所有信息以确定是否有“危险情况”出现。航空交通控制员可以查询数据库，获得显示在屏幕上的任意飞机的特定信息。

使用 OOD，为 ATC 系统做一个设计。不要试图去实现它！

推荐阅读文献及其他信息源

面向对象设计的文献正在迅速地扩展。 Reil(Object－Oriented Design Through Heuristics,Addison－Wesley,1996),Gamma 等[GAM95],Booch[BOO94]、Jacobson[JAC92]，Rambaugh 等 [RAM91]，Coad 和 Yourdon[COA91]，以及 Wirfs－Brock、Wilkerson 和 Weiner[WIR90]等书籍提供了对这个重要领域的深入的调察，此外，下列书籍也值得一看：

Berard,E.V.,Essays on Object－Oriented Software Engineering,Addison－Wesley，1993.Champeaux，D.，D.Lea，and P.Faure，Object－Oriented System Deveolpment，Addison－Wesley，1993.

Coad,P.,D.North,and M.Mayfield,Object Models:Strategies,Patterns,and Applications，Prentice－Hall，1995.

Coleman，D.et al.，Object－Oriented Development：The FUSION Method，Prentice－Hall，1994.

Hutt，T.F.(ed.)，Object Analysis and Design：Description of Methods，Wiley，1994.

Meyer，B.，Reusable Software，Prentice－Hall，1995.

Page－Jones，M.，What Every Programmer Should Know about Object－Oriented Design，DorsetHouse， 1995.

Schlaer，S.，and Mellor，S.，Object Life Cycles： Modeling the World in States，Prentice－Hall，1992.Wilkie，G.，Object－Oriented Software Engineering，Addison－Wesley，1993.

Yourdon，E.，Object－Oriented Systems Design：An Integrated Approach，Prentice－Hall，1994.

Yourdon，E.et al.，Mainstream Objects，Prentice－Hall，1995.

Buschmann 和 Meunier(Pattern－OrientedSoftwareArchitecture，Wiley，1996)显示了对大型应用的开发设计模式有何帮助。Hutt(Object Analysis and Design，Wiley，1994)编辑了一本专辑，显示并比较了 16 种不同的面向对象分析和设计方法。

在面向对象编程(OOP)方面已经出版了上百的书籍，关于 OOP 语言的书籍举例如下：

Ada95：Barnes，J.，Programming in Ada95，Addison－Wesley，1995.

C＋＋：Eckel，B.，C＋＋InsideandOut，McGraw－Hill，1993.

Lakos，J.，Large－ScaleC＋＋SoftwareDesign，1996.

Eiffel：Rist，R.S.，and R.Terwillinger，Object－OrientedProgramminginEiffel，1995.

Meyer，B，Object－OrientedSoftwareConstruction，2ndedition，Prentice－Hall，1995.

Java：Arnold，K.，and J.Gosling，The Java Programming Language，Addison－Wesley，1996.

Manger，J.，Essential Java，McGraw－Hill，1996.

SmallTalk：LaLonde，W.R.，and J.R.Pugh，Programming in Smalltalk，Prentice－Hall，1995.Klimas，E.et al.，Smalltalk with Style，Prentice－Hall，1995.

Selek 及其同事(Real－Time Object－Oriented Modeling，Wiley，1995)讨论了对实时系统的特殊需求。 Lewis 及其同事(Object－OrientedApplicationFrameworks，IEEEComputerSocietyPress，1995)考虑使用“应用框架(application frameworks)”作为设计机制。

和第 19、20 章相同的 Internet 信息源对 OOD 也是可用的，Internet 新闻组 comp.object 和针对 OO 程序设计语言的新闻组(如，comp.lang.C＋＋和 comp.lang.smalltalk)是有用的关于 OOD 方法、程序设计技术和工具的信息、讨论和争论源。

下面的资源包含有用的关于 OOD 和 OOP 的信息，包括其他的参考文献、文章和信息：

http：//www.omg.org/

http：//osm7.cs.byu.edu

http：//www.mcs.com/～ woodsman/otek.html

http：//wwwtrese.cs.utwente.nl/index.html

关于面向对象设计的WWW文献的最新列表可在http：//www.rspa.com找到。

① 请回顾一下第 13 章中讨论的各主要控制结构的表示。

② 值得注意的是导出不总是直接完成的。详细讨论请见［DAV95］。

① 通常，OOD方法以方法开发者的名字来命名，即使方法已有一个给定的名字或缩略词代号。

② 过程描述中的多个步骤的许多详细含义将在第 21.3 和 21.4 中讨论。

① 块是一个设计抽象；它可能是代表某个集合对象。

① 协议是对到响应类的信息的格式描述。

① 尚未读过第 20 章的读者请快点学习它吧！

② 记住OOA是一组迭代活动。分析工作又在设计工作之后出现是完全可能的。

① 有兴趣的读者可参阅［RO91］，［TAY92]或[RA094]。

① 已对单个重要的动词添加了下划线。

① Gamma和他的同事们［GAM95］已经发布了可用于OO系统的 23 个设计模式。

第 22 章　面向对象测试

在第 16 章，我们学习了测试的目标，简单地说，在现实可行的时间跨度内应用可管理的工作量去发现尽可能多的错误。虽然对面向对象软件而言，这个基本目标仍保持不变，但是 OO 程序的性质改变了测试的策略和测试战术。

有人可能争辩说，随着 OOA 和 OOD 的成熟，更多的设计模式复用将减轻 OO 系统的繁重测试量。准确地说，其反面才是真的。Binder [BIN94b] 讨论该问题如下：

每次复用是一个新的使用语境，要谨慎地重新测试。为了获得面向对象系统的高可靠性，似乎可能将需要更多、而不是更少的测试。

为了充分地测试 OO 系统，必须做好三件事：(1)测试的定义必须扩大包括用于 OOA 和 OOD 模型的错误发现技术；(2)单元和集成测试策略必须有很大的改变；(3)测试用例的设计必须考虑 OO 软件的独特特征。

22.1 扩大测试的视角

面向对象软件的构造从分析和设计模型(第 19、20 章)的创建开始。因为 OO 软件工程范型的演化性质，模型从对系统需求相对非正式的表示开始，逐步演化为详细的类模型、类连接和关系、系统设计和分配、以及对象设计(通过消息序列的对象连接模型)。在每个阶段，测试模型，以试图在错误传播到下一次递进前发现错误。

可以争辩说时 OO 分析和设计模型的复审是特别有用的，因为相同的语义结构(如，类、属性、操作、消息)出现在分析、设计和代码阶段，因此，在分析阶段发现的类属性定义中的问题将遏止当问题直至设计或编码阶段(或甚至到下一次分析迭代)才被发现所带来的副作用。

例如，考虑一个类，在第一次 OOA 迭代时定义了其中一系列属性，一个外来的无关属性被附于该类(由于对问题域的错误理解)，然后定义两个操纵该属性的操作。在复审时，一个领域专家指出该问题。通过在本阶段删除该无关属性，可在分析过程中避免下面的问题和不必要的努力：

1.特殊的子类可能已经被生成以适应不必要的属性或它的例外。涉及不必要的子类的创建工作可被避免。

2.类定义的错误解释可能导致不正确的或无关的类关系。

3.系统或它的类的行为可能被不适当地刻划以适应该无关属性。

如果问题未在分析过程中被发现并进一步向前传播，在设计中可能产生下面的问题(如尽早复审，应已避免的)：

1.在系统设计阶段可能将类不合适的分配到子系统和/或任务。

2.可能花费不必要的工作去创建针对无关属性的操作的过程设计。

3.消息模型将是不正确的(因为必须为无关的操作设计消息)。

如果问题在设计阶段仍未被检测到，并传送到编码活动中，则将花费大量的工作努力和精力去生成那些实现一个不必要的属性、两个不必要的操作、驱动对象间通信的消息以及很多其他相关问题的代码。此外，类的测试将花费更多的时间。一旦问题最终被发现，必须对系统进行修改，从而引致由于修改而产生到作用的很大的潜在可能性。

在它们的开发的后面阶段，OOA 和 OOD 模型提供了关于系统的结构和行为的实质性信息，为此，这些模型应该在生成代码前经受严格的复审。

所有面向对象模型应该被测试(在这个语境内，术语“测试”代表正规的技术复审)，以保证在模型的语法、语义和语用 [LIN94] 的语境内的正确性、完整性和一致性[MCG94]。)，以保证在模型的语法、语义和语用[LIN94]的语境内的正确性、完整性和一致性。

22.2 测试 OOA 和 OOD 模型

分析和设计模型不能进行传统意义上的测试，因为它们不能被执行。然而，正式的技术复审(第 8 章)可被用于检查分析和设计模型的正确性和一致性。

22.2.1 OOA 和 OOD 模型的正确性

用于表示分析和设计模型的符号体系和语法将是和为项目选定的特定分析和设计方法联系的，因此，语法正确性基于符号是否合适使用，而且对每个模型复审以保证保持合适的建模约定。

在分析和设计阶段，语义正确性必须基于模型对现实世界问题域的符合度来判断，如果模型精确地反应了现实世界(到这样一个细节程度：它对模型复审时所处的开发阶段是合适的)，则它是语义正确的。为了确定是否模型确实在事实上反应了现实世界，它应该被送给问题域专家，专家将检查类定义和类层次以发现遗漏和含混。评估类关系(实例连接)以确定它们是否精确地反应了现实世界的对象连接。[①]

22.2.2 OOA 和 OOD 模型的一致性

对 OOA 和 OOD 模型的一致性判断可以通过“考虑模型中实体间的关系。一个不一致的模型在某一部分有表示，但未在模型的其他部分正确地反应”[MCG94]。

为了评估一致性，应该检查每个类及其和其他类的连接。可运用类—责任—协作者(CRC)模型和对象—关系图。如我们在第 20 章提到，CRC 模型由 CRC 索引卡片构成，每个 CRC 卡片列出类名、类的责任(操作)、以及其协作者(其他类，类向它们发送消息并依赖于它们完成自己的责任)。协作蕴含了在 OO 系统的类之

间的一系列关系(即，连接)，对象关系模型提供了类之间连接的图形表示。所有这些信息可以从 OOA 模型(第 20 章)得到。

为了评估类模型，推荐采用以下面步骤［MCG94］：)得到。

1.再次考察 CRC 模型和对象-关系模型，进行交叉检查以保证由 OOA 模型所蕴含的协作适当地反应在二者中。

2.检查每个 CRC 索引卡片的描述以确定是否某被受权的责任是协作者的定义的一部分，例如，考虑为某 POS 结账系统定义的类，称为 credit sale，该类的 CRC 卡片如图 22-1 所示。

对于这组类和协作，我们问如果某责任(如，read credit card)被委托给指定的协作者(creditcard)，该责任是否将被完成。即，类 credit card 是否具有一个操作以使得它可以被读。在本例的情形，我们的回答是“是”。遍历对象关系以保证所有这样的连接是有效的。

3.反转该连接以保证每个被请求服务的协作者正在接收来自合理源的请求，例如，如果 credit card 类接收来自 credit sale 类的 purchase amount 请求，则将会有问题，因为 credit card 并不知道 purchase amount。

4.使用在第 3 步检查的反转连接，确定是否可能需要其他的类或责任是否被合适地在类间分组。

5.确定是否被广泛请求的责任可被组合为单个的责任，例如，read credit card 和 getauthorization 在每种情况均发生，它们可以被组合为 validate credit request 责任，它结合了获取信用卡号及获得授权。

6.步骤 1 到 5 被迭代地应用到每个类，并贯穿 OOA 模型的每次演化。

一旦已经创建了设计模型(第 21 章)，也应该进行对系统设计和对象设计的复审。系统设计描述了构成产品的子系统、子系统被分配到处理器的方式、以及类到子系统的分配。对象模型表示了每个类的细节和实现类间的协作所必需的消息序列活动。

通过检查在 OOA 阶段开发的对象—行为模型，和映射需要的系统行为到被设计用于完成该行为的子系统来进行系统设计的复审。也在系统行为的语境内复审并发性和任务分配，评估系统的行为状态以确定哪些行为并发地存在。

对象模型应该针对对象—关系网络来测试，以保证所有设计对象包含为实现为每张 CRC 索引卡片定义的协作所必须的属性和操作。此外，使用传统的检查技术来复杂操作细节的详细规约(即，实现操作的算法)。

22.3 面向对象的测试策略

传统的测试计算机软件的策略是从“小型测试”开始，逐步走向“大型测试”。用软件测试的行话来陈述(第 17 章)，我们从单元测试开始，然后逐步进入集成测试，最后是有效性和系统测试。在传统应用中，单元测试集中在最小的可编译程序单位——子程序(如，模块、子例程、进程)，一旦这些单元均被独立测试后，它被集成进程序结构中，这时要进行一系列的回归测试以发现由于模块的接口所带来的错误和新单元加入所导致的副作用，最后，系统被作为一个整体测试以保证发现在需求中的错误。

22.3.1 在 OO 语境中的单元测试

当考虑面向对象软件时，单元的概念发生了变化。封装驱动了类和对象的定义，这意味着每个类和类的实例(对象)包装了属性(数据)和操纵这些数据的操作(也称为方法或服务)。而不是个体的模块。最小的可测试单位是封装的类或对象，类包含一组不同的操作，并且某特殊操作可能作为一组不同类的一部分存在，因此，单元测试的意义发生了较大变化。

我们不再孤立地测试单个操作(传统的单元测试观点)，而是将操作作为类的一部分。作为一个例子，考虑一个类层次，其中操作 X 针对超类定义并被一组子类继承，每个子类使用操作 X，但是它被应用于为每个子类定义的私有属性和操作的环境内。因为操作 X 被使用的语境有微妙的不同，有必要在每个子类的语境内测试操作 X。这意味着在面向对象的语境内在真空中测试操作 X(即传统的单元测试方法)是无效的。

对OO软件的类测试等价于传统软件[①]的单元测试。和传统软件的单元测试不一样，它往往关注模块的算法细节和模块接口间流动的数据，OO软件的类测试是由封装在类中的操作和类的状态行为所驱动的。

22.3.2 在 OO 语境中的集成测试

因为面向对象软件没有层次的控制结构，传统的自顶向下和自底向上集成策略就没有意义，此外，一次集成一个操作到类中(传统的增量集成方法)经常是不可能的，这是由于“构成类的成分的直接和间接的交互”［BER93］。

对 OO 软件的集成测试有两种不同策略［BIN94a］，第一种称为基于线程的测试(thread－basedtesting)，集成对回应系统的一个输入或事件所需的一组类，每个线程被集成并分别测试，应用回归测试以保证没有产生副作用。第二种称为基于使用的测试(use－based testing)，通过测试那些几乎不使用服务器类的类(称为独立类)而开始构造系统，在独立类测试完成后，下一层的使用独立类的类，称为依赖类，被测试。这个依赖类层次的测试序列一直持续到构造完整个系统。序列和传统集成不同，使用驱动器和桩(stubs)(第 16 章)作为替代操作是要尽可能避免的。

集群测试(cluster testing)［MCG94］ 是 OO 软件集成测试的一步，这里一群协作类(通过检查 CRC 和对象—关系模型而确定的)通过设计试图发现协作中的错误的测试用例而被测试

22.3.3 在 OO 语境中的有效性测试

在有效性或系统层次，类连接的细节消失了。和传统有效性一样，OO 软件的有效性集中在用户可见的动作和用户可识别的系统输出。为了协助有效性测试的导出，测试员应该利用作为分析模型一部分的使用实例(第 20 章)，使用实例提供了在用户交互需求中很可能发现错误的一个场景。

传统的黑盒测试方法(第 16 章)可被用于驱动有效性测试，此外，测试用例可以从对象—行为模型和作为 OOA 的一部分的事件流图中导出。

22.4 OO 软件的测试用例设计

OO 软件的测试用例设计方法还正处于成型期，然而，Berard 已建议了对 OO 测试用例设计的整体方法［BER93］：

1.每个测试用例应该被唯一标识，并且和将被测试的类显式地相关联。

2.应该陈述测试的目的。

3.对每个测试应该开发一组测试步骤，应该包含［BER93］：

a.将被测试的对象的一组特定状态。

b.将作为测试的结果使用的一组消息和操作。

c.当测试对象时可能产生的一组例外。

d.一组外部条件(即，为了适当地进行测试而必须存在的软件的外部环境的变化)。

e.辅助理解或实现测试的补充信息。

和传统测试用例设计不同，传统测试是由软件的输入—加工—输出视图或个体模块的算法细节驱动的，面向对象测试关注于设计合适的操作序列以测试类的状态。

22.4.100 概念的测试用例设计的含义

如我们已经看到的，00 类是测试用例设计的目标。因为属性和操作是被封装的，对类之外操作的测试通常是徒劳的。虽然封装是 00 的本质设计概念，但是它可能会成为测试的小障碍，如 Binder [BIN94a] 所说，“测试需要对对象的具体和抽象状态的报告”，然而，封装却使得这些信息在某种程度上难于获得。除非提供了内置操作来报告类属性的值，否则，对对象的状态快照是难于获得的。

继承也造成了对测试用例设计者的挑战。我们已知道，即使是彻底复用的，对每个新的使用语境也需要重测试。此外，多重继承[①]增加了需要测试的语境数量 [BIN94a] 从而 使测试进一得步复杂化。如果从超类导出的子类被用于相同的问题域，有可能对超类导出的测试用例集可以用于子类的测试，然而，如果子类被用于完全不同的语境，则超类的测试用例将没有多大用处，必须设计新的测试用例集。

22.4.2 传统测试用例设计方法的可用性

在第 16 章描述的白盒测试方法可用于对为类定义的操作的测试，基本路径、循环测试或数据流技术可以帮助保证已经测试了操作中的每一条语句，然而，很多类操作的简洁结构导致某些人认为：将用于白盒测试的工作量用于类级别的测试可能会更好。

黑盒测试方法就象对传统软件工程方法开发的系统和对 00 系统同样适用的，如我们在本章前面看到的，use cases 可以为黑盒及基于状态的测试的设计提供有用的输入。

22.4.3 基于故障的测试

在 00 系统中基于故障的测试的目标是设计最有可能发现似乎可能的故障的测试。因为产品或系统必须符合客户需求，因此，完成基于故障的测试所需的初步计划是从分析模型开始。

测试员查找似乎可能的故障(即，系统实现中有可能产生错误的方面)，为了确定是否存在这些故障，设计测试用例以测试设计或代码。

考虑一个简单的例子。[③]软件工程师经常在问题的边界处犯错误，例如，当测试SQRT操作(该操作对负数返回错误)时，我们尝试边界：一个靠近零的负数和零本身，“零本身”用于检查是否程序员犯了如下错误：

```
if(x>0) calculate_the_square_root();
```

而不是正确的：

```
if(x>= 0) calculate_the_square_root();
```

作为另一个例子，考虑布尔表达式：

if(a＆＆bllc)

多条件测试和相关的用于探查在该表达式中可能存在的故障的技术，如：

“＆＆”应该是“ll”

“！”在需要处被省去

应该有括号包围“！bll”

对每个可能的故障，我们设计迫使不正确的表达式失败的测试用例。在上面的表达式中，(a＝0，b＝0，c＝0)将使得表达式得到预估的“假”值，如果“＆＆”已改为“ll”，则该代码做了错误的事情，有可能分叉到错误的路径。

当然，这些技术的有效性依赖于测试员如何感觉“似乎可能的故障”，如果00系统中的真实故障被感觉为“难以置信的”，则本方法实质上不比任何随机测试技术好。然而，如果分析和设计模型可以提供对什么可能出错的深入洞察，则，基于故障的测试可以以相当低的工作量花费来发现大量的错误。

集成测试在消息连接中查找似乎可能的故障，在此语境下，会遇到三种类型的故障：未期望的结果、错误的操作/消息使用、不正确的调用。为了在函数(操作)调用时确定似乎可能的故障，必须检查操作的行为。

集成测试，对象的“行为”通过其属性被赋予的值而定义，测试应该检查属性以确定是否对对象行为的不同类型产生合适的值。

应该注意，集成测试试图在客户对象，而不是服务器对象中发现错误，用传统的术语来说，集成测试的关注点是确定是否调用代码中存在错误，而不是被调用代码中。用调用操作作为线索，这是发现实施调用代码的测试需求的一种方式。

22.4.4 00编程对测试的影响

面向对象编程可能对测试有几种方式的影响，依赖于OOP的方法，

- 某些类型的故障变得几乎不可能(不值得去测试)
- 某些类型的故障变得更加可能(值得进行测试)
- 出现某些新的故障类型

当调用一个操作时，可能很难确切知道执行什么代码，即，操作可能属于很多类之一。同样，也很难确定准确的参数类型/类，当代码访问参数时，可能得到一个未期望的值。

可以通过考虑如下的传统的函数调用来理解这种差异：

x＝func(y)；

对传统软件，测试员需要考虑所有属于 func 的行为，其他则不需考虑。在 00 语境中，测试员必须考虑 base：：func()、 of derived：：func()等行为。每次 func 被调用，测试员必须考虑所有不同行为的集合，如果遵循了好的 00 设计习惯并且限制了在超类和子类(用 C++的术语，称为基类和派生类)间的差异，则这是较为容易的。对基类和派生类的测试方法实质上是相同的，所不同的仅是簿记之一。

测试 00 的类操作类似于测试一段代码，它设置函数参数，然后调用该函数。继承是一种方便的生成多态操作的方式，在调用点，关心的不是继承，而是多态。继承确实使得对测试需求的搜索更为直接。

由于 00 系统的体系结构和构造，是否某些类型的故障更加可能，而其他类型的故障则几乎不可能吗？对 00 系统而言，回答是“是”。例如，因为 00 操作通常是较小的，往往存在更多的集成工作和更多的集成故障的机会，集成故障变得更加可能。

22.4.5 测试用例和类层次

如本章前面所述，继承并没有排除对所有派生类进行全面测试的需要，事实上，它确实使测试过程变得复杂。

考虑下面情形，类 base 包含了操作 inherited 和 redefined，类 derived 时 redefined 重定义以用于局部语境中，毫无疑问，必须测试 derived：：redefined()操作，因为它表示了新的设计和新的代码。但是，必须重测试 derived：：inherited()操作吗？

如果 derived：：inherited()调用 redefined，而 redefined 的行为已经改变，derived：：inherited()可能错误地处理这新行为，因此，即使其设计和代码没有改变，它需要被重新测试。然而，重要的是要注意，仅仅必须执行 derived：：inherited()的所有测试的一个子集。如果 inherited 的设计和代码部分不依赖于 redefined(即，不调用它或任意间接调用它的代码)，则不需要在 derived 类中重测试该代码。

base：：redefined()和 derived：：redefined()是具有不同规约和实现的两个不同的操作，它们各自具有一组从规约和实现导出的测试需求，这些测试需求探查似乎可能的故障：集成故障、条件故障、边界故障等等。但是，操作可能是相似的，它们的测试需求的集合将交迭，00 设计得越好，交迭就越大，仅仅需要对那些不能被 base：：redefined()测试满足的 derived：：redefined()的需求来导出新测试。

小结一下，base：：redefined()测试被应用于类 derived 的对象，测试输入可能同时适合于 base 和 derived 类，但是，期望的结果可能在 derived 类中有所不同。

22.4.6 基于场景的测试设计

基于故障的测试忽略了两种主要的错误类型：(1)不正确的规约，和(2)子系统间的交互。当和不正确的规约关联的错误发生时，产品不做客户希望的事情，它可能做错误的事情，或它可能省略了重要的功能。在任一情形下，质量(对要求的符合度)均受到影响。当一个子系统建立环境(如事件、数据流)的行为使得另一个子系统失败时，发生和子系统交互相关联的错误。

基于场景的测试关心用户做什么而不是产品做什么。它意味着捕获用户必须完成的任务(通过使用实例)，然后应用它们或它们的变体作为测试。

场景揭示交互错误，为了达到此目标，测试用例必须比基于故障的测试更复杂和更现实。基于场景的测试往往在单个测试中处理多个子系统(用户并不限制他们自己一次只用一个子系统)。

例如，考虑对文本编辑器的基于场景的测试的设计，下面是使用实例：

使用实例：确定最终草稿

背景：打印“最终”草稿、阅读它并发现某些从屏幕上看是不明显的恼人错误是常见的。该使用实例描述当此事发生时产生事件的序列。

1.打印完整的文档。

2.在文档中移动，修改某些页面。

3.当每页被修改后，打印它。

4.有时打印一系列页面。

该场景描述了两件事：测试和特定的用户需要。用户需要是明显的：(1)打印单页的方法；以及(2)打印一组页面的方法。当测试进行时，有需要在打印后测试编辑(以及相反)。测试员希望发现打印功能导致了编辑功能的错误，即，此两个软件功能不是合适的相互独立的。

使用实例：打印新拷贝

背景：某人向用户要求文档的一份新拷贝，它必须被打印。

1.打开文档。

2.打印文档。

3.关闭文档。

测试方法也是相当明显的，除非该文档未在任何地方出现过，它是在早期的任务中创建的，该任务对现在的任务有影响吗？

在很多现代的编辑器中，文档记住它们上一次被如何打印，缺省情况下，它们下一次用相同的方式打印。在“确定最终草稿”场景之后，仅仅在菜单中选择“Print”并对话盒里点击“Print”按钮，将使得上次修正的页面再打印一次，这样，按照编辑器，正确的场景应该是：

使用实例：打印新拷贝

1.打开文档。

2.选择菜单中的“Print”。

3.检查你是否将打印一系列页面，如果是，点击以打印完整的文档。

4.点击“Print”按钮。

5.关闭文档。

但是，这个场景指明了一个潜在的规约错误，编辑器没有做用户希望它做的事。客户经常忽略在第 3 步中的检查，当他们走到打印机前发现只有一页，而他们需要 100 页时，他们将是烦恼的。烦恼的客户指出这一规约错误。

测试用例的设计者可能在测试设计中忽略这种依赖，但是，有可能在测试中问题会出现，测试员则将必须克服可能的反应，“这就是它工作的方式”。

22.4.7 测试表层结构和深层结构

表层结构指 OO 程序的外部可观察的结构，即，对终端用户立即可见的结构。不是处理函数，而是很多 OO 系统的用户可能被给定一些以某种方式操纵的对象。但是不管接口是什么，测试仍然基于用户任务进行。捕获这些任务涉及到理解、观察以及和代表性用户(以及很多值得考虑的非代表性用户)的交谈。

在细节上一定存在某些差异。例如，在传统的具有面向命令的界面的系统中，用户可能使用所有命令的列表作为检查表。如果不存在执行某命令的测试场景，测试可能忽略某些用户任务(或具有无用命令的界面)。在基于对象的界面中，测试员可能使用所有的对象列表作为检查表。

当设计者以一种新的或非传统的方式来看待系统时，则可以得到最好的测试。例如，如果系统或产品具有基于命令的界面，则当测试用例设计者假设操作

是独立于对象的，将可以得到更彻底的测试。提出这样的问题：“当使用打印机工作时，用户有可能希望使用该操作(它仅应用于扫描仪对象)吗？”不管界面风格是什么，针对表层结构的测试用例设计应该同时使用对象和操作作为导向被忽视任务的线索。

深层结构指 00 程序的内部技术细节，即，通过检查设计和/或代码而理解的结构。深层结构测试被设计用以测试作为 00 系统的子系统和对象设计(第 21 章)的一部分而建立的依赖、行为和通信机制。

分析和设计模型被用作深层结构测试的基础。例如，对象—关系图或子系统协作图描述了在对象和子系统间的可能对外不可见的协作。那么测试用例设计者会问：“我们已经捕获了某些测试任务，它测试在对象—关系图或子系统协作图中记录的协作？如果没有，为什么？”

类层次的设计表示提供了对继承结构的误入洞察，继承结构被用在基于故障的测试中。考虑如下一种情形：一个命名为 caller 的操作只有一个参数，并且该参数是到某基类的引用。当 caller 被传递给派生类时将发生什么事情？可能影响 caller 的行为有什么差异？对这些问题的回答可能导向特殊测试的设计。

22.5 在类级别上可用的测试方法

在第 16 章，我们提到软件测试从“小型”测试开始，慢慢进展到“大型”测试。对 00 系统的小型测试着重于单个类和类封装的方法。随机测试和划分是在 00 测试中测试类的方法 [KIR94] 。宰胖赜诘ジ隼嗪屠凿庾暗姆椒ād K婊　馑院突　质窃负 0 测试中测试类的方法

22.5.1 对 00 类的随机测试

为了提供时这些方法的简略性说明，考虑一个银行应用，其中 account 类有下列操作：open，setup，deposit，withdraw balance，summarize，creditLimit，和 close [KIR94] ，每一个操作均可应用于 account，但是，该问题的本质包含了一些限制(如，帐号必须在其他操作可应用前被打开，在所有操作完成后才关闭)。即使有了这些限制，也存在操作的很多排列。一个 account 实例的最小的行为生命历史包括下面操作：

open • setup • deposit • withdraw • close

这表示了对 account 的最小测试序列，然而，在下面序列中可能发生大量的其他行为：

open • setup • deposit • [deposit | withdraw | balance | summarize | creditLimit]n • withdraw • close

一系列不同的操作序列可以随机产生，例如：

测试用例＃r1：

open • setup • deposit • deposit • balance • summarize • withdraw • close

测试用例＃r2：

open • setup • deposit • withdraw • deposit • balance • creditLimit • withdraw • close

执行这些和其他的随机顺序测试以测试不同的类实例生命历史。

22.5.2 在类级别上的划分测试

采用和划分测试(partition testing)可以来减少测试类所需的测试用例的数量。传统软件的等价划分(第 16 章)基本相同的方式输入和输出被分类，测试用例被设计以处理每个类别。但是，划分类别如何导出的呢？

基于状态的划分是根据类操作改变类的状态的能力来划分类操作的。再次考虑 account 类，状态操作包括 deposit 和 withdraw，而非状态操作包括 balance、summarize 和 creditLimit。测试被以这样一种方式来设计：分别独立测试改变状态的操作和不改变状态的操作，因此，

测试用例＃p1：
open • setup • deposit • deposit • withdraw • withdraw • close

测试用例＃p2：
open • setup • deposit • summarize • creditLimit • withdraw • close

测试用例＃p1 改变状态，而测试用例＃p2 测试不改变状态的操作(除了那些在最小序列中的操作)。

基于属性的划分是根据操作使用的属性来划分类操作。对 account 类，用属性 balance 和 credit limit 来定义划分，操作被分为三个类别：(1)使用 credit limit 的操作，(2)修改 credit limit 的操作，和(3)不使用或不修改 credit limit 的操作。然后对每个划分设计测试序列。

基于类别的划分是根据各自完成的类属函数来划分类操作。例如，在 account 类中的操作可被分类为初始化操作(open、setup)、计算操作(deposit、withdraw)、查询操作(balance、summarize、creditLimit)和终止操作(close)。

22.6 类间测试用例设计

当 OO 系统的集成开始后，测试用例的设计变得更复杂。正是在此阶段，必须开始对类间的协作测试。为了说明类间测试用例生成［KIR94］，我们扩展在 22.5 节中引入的银行例子，使⑺　　缤*22－2 所示的类和协作，图中箭头的方向指明消息的传递方向，标注则指明被作为消息所蕴含的一系列协作来调用的操作。

和个体类的测试一样，类协作测试可通过应用随机和划分方法以及基于场景的测试和行为测试来完成。

22.6.1 多个类测试

Kirani 和 Tsai［KIR94］建议采用下面的步骤序列以生成多个类随机测试用例：

1.对每个客户类，使用类操作符列表来生成一系列随机测试序列，操作符将发送消息给其他服务器对象。

2.对所生成的每个消息，确定在服务器对象中的协作者类和对应的操作符。

3.对在服务器对象(已经被来自客户对象的消息调用)中的每个操作符确定传递的消息。

4.对每个消息，确定下一层被调用的操作符并结合这些操作符到测试序列中。

为了说明［KIR94］，考虑 bank 类相对于 ATM 类的(图 22－2)的操作序列：

verifyAcct • verifyPIN • ［[verifyPolicy • withdrawReq］
┃depositReq┃acctlnfoREQ]n

对 bank 类的随机测试用例可能是：

测试用例＃r3：verifyAcct • verifyPIN • depositReq

为了考虑涉及到该测试的协作者，要考虑同在测试用例 r3 中提到的每个操作相关联的消息。bank 必须和 validationInfo 协作以执行 verifyAcct 和 verifyPIN，bank 必须和 account 协作以执行 depositReq，因此，测试上面提到的协作的新的测试用例是：

测试用例＃r4：　$verifyAcct_{Bank}$ • ［$validAcct_{ValidationInfo}$］ • $verifyPIN_{Bank}$ • ［$validPIN_{ValidationInfo}$］ depositReq • ［$deposit_{account}$］

多个类划分测试的方法类似于单个类划分测试的方法，如在 22.5.2 节讨论那样划分单个类，然而，扩展测试序列以包括那些通过发送给协作类的消息而激活的操作。另一种方法基于特殊类的接口来划分测试，如图 22－2 所示，bank

类接收来自 ATM 和 cashier 类的消息，因此，可以通过将 bank 中的方法划分为服务于 ATM 和服务于 cashier 的操作来测试。基于状态的划分(22.5.2 节)可用于进一步精化划分。

22.6.2 从行为模型导出的测试

在第 20 章，我们讨论了使用状态—变迁图(STD)作为表示类的动态行为的模型。类的 STD 可被用于帮助导出测试类(和那些与其协作的类)的动态行为的测试序列。图 22－3 [KIR94] 给出了前面讨论的 account 类的 STD，根据该图，初始变迁经过了 empty acct 和 setup acct 状态，类的实例的大多数行为发生在 working acct 状态，最终的 withdrawal 和 close 使得 account 类分别向 non－working acct 或 dead acct 状态变迁。

所设计的测试应该涵盖所有的状态[KIR94]，即，操作序列应该致使 account 类产生经过所有允许的状态的变迁：

测试用例＃s1：
open • setupAccnt • deposit(initial) • withdraw(final) • close

应该注意，该序列等同于 22.5.1 节讨论的最小测试序列，加入其他测试序列到最小序列中：

测试用例＃s2：
open • setupAccnt • deposit(initial) • deposit • balance • credit • withdraw(final) • close

测试用例＃s3：
open • setupAccnt • deposit(initial) • deposit • withdraw • accntlnfo • withdraw(final) • close

仍然可以导出更多的测试用例以保证已经适当的测试了类的所有行为，在类行为导致与一个或多个类协作的情况下，使用多个 STD 去跟踪系统的行为流。

可以按“宽度优先的方式”遍历状态模型 [MCG94] ，在本语境内宽度优先指明：一个测试用例测试单个变迁，并且当测试新的变迁时，仅使用以前被测试的变迁。

考虑在 22.2.2 节讨论的 credit card 对象，credit card 的初始状态是 undefined(即，没有提供信用卡号)，通过在销售中读信用卡，对象进入 defined 状态，即定义属性 card number 和 expirationdate 以及银行特定的标识符。当发送请求授权时，信用卡被提交(submitted)；当授权被接收时，信用卡被核准(approved)。creditcard 从一个状态到另一个状态的的变迁可以通过导出引致变迁发生的测试用例来测试。对这种测试类型的宽度优先的方法将不会在测试

undefined 和 defined 之前测试 submitted，如果这样做了，它将使用了以前尚未测试的变迁，因此违反了宽度优先准则。

22.7 小结

面向对象测试的整体目标——以最小的工作量发现最多的错误——和传统软件测试的目标是一致的，但是 OO 测试的策略和战术有很大不同。测试的视角扩大到包括复审分析和设计模型，此外，测试的焦点从过程构件(模块)移向了类。

因为 OO 分析和设计模型以及产生源代码是语义耦合的，测试(以正式的技术复审的方式)在这些活动进行中开始，为此，CRC、对象—关系、和对象—行为模型的复审可视为第一阶段的测试。

一旦已经完成 OOP，可对每个类进行单元测试。类测试使用一系列不同的方法：基于故障的测试、随机测试和划分测试。每种方法均测试类中封装的操作。设计测试序列以保证相关的操作被处理。类的状态，由其属性的值表示，检查以确定是否存在错误。

集成测试可使用基于线程或基于使用的策略来完成。基于线程的测试集成一组相互协作以对某输入或事件作出回应的类。基于使用的测试按层次构造系统，从那些不使用服务器类的类开始。集成测试用例设计方法也可以使用随机和划分测试。此外，基于场景的测试和从行为模型导出的测试可用于测试类及其协作者。测试序列跟踪跨越类协作的操作流。

OO 系统有效性测试是面向黑盒的并可以通过应用对传统软件讨论的相同的黑盒方法来完成。然而，通过使用使用实例作为有效性测试的主要驱动基于场景的测试主宰了 OO 系统的有效性。

思考题

22.1 用你自己的话，描述为什么类是 OO 系统中测试的最小合理单位。

22.2 为什么我们必须重测试从某现存类导出的子类，即使已经完全地测试过现存类？我们是否可以使用为现存类设计的测试用例？

22.3 为什么“测试”应该从 OOA 和 OOD 活动开始？

22.4 为 SafeHome 导出一组 CRC 索引卡片，并进行在 22.2.2 节提到的步骤以确定是否存在不一致性。

22.5 用于集成测试的基于线程的和基于使用的策略间有什么不同？集群测试如何适合它？

22.6 对你在思考题 21.12 中生成的 SafeHome 系统的设计中定义的三个类运用随机测试和划分。制作指明将被调用的操作序列的测试用例。

22.7 对 SafeHome 的设计应用多个类测试和从行为模型导出的测试。

22.8 用在思考题 22.6 和 22.7 中提到的方法，为在思考题 12.13 中描述的 PHTRS 系统导出测试用例。

22.9 用在思考题 22.6 和 22.7 中提到的方法，为在思考题 21.14 中描述的视频游戏导出测试用例。

22.10 用在思考题 22.6 和 22.7 中提到的方法，为在思考题 21.15 中描述的 e－mail 系统导出测试用例。

22.11 用在思考题 22.6 和 22.7 中提到的方法，为在思考题 21.16 中描述的 ATC 系统导出测试用例。

22.12 用在思考题 22.6 和 22.7 中提到的每个方法，为在 22.5、22.6 节中描述的银行应用导出四个其他的测试用例。

推荐阅读文献及其他信息源

面向对象测试的文献才刚刚开始出现。Jorgensen(Software Testing：A Craftsman's Approach,CRC Press,1995),McGregor 和 Sykes(Object－Oriented Software Development，Van NostrandReinhold，1992)讨论了这一主题。Beizer(Black－Box Testing，Wiley，1995)讨论了一系列适合于 OO 语境的测试用例设计方法，Binder(Testing Object－Oriented Systems，Addison－Wesley，1996)和 Marick［MAR94］给出了对 OO 测试的详细讨论。此外，在第 16 章提到的很多信息源通常也适用于 OO 测试。

Communications of the ACM 的 1994 年第 9 期是 OO 测试专集，它给出了一系列有价值的论文和覆盖 OO 测试的较广范围的研究参考书目。

Internet 新闻组 comp.object 也时常有关于 OO 测试的内容，不幸的是，该新闻组的 FAQ(虽然在很多方面是全面的)当前尚未涉及 OO 测试。对 OO 测试的进一步讨论有时可在 comp.software－eng 和 comp.testing 找到。

下面的网址包含了关于 OO 测试的参考书目和其他信息：

http：//www.rbsc.com/pages/ootbib.html

http：//donkey.cs.arizona.edu：1994/bib/Object

下面网址包含了 OO 测试的讨论：

http：//www.cs.washington.edu/homes/gmurphy/testSTApp.html

http：//www.stlabs.com/marick/root.htm

http：//www.toa.com

关于面向对象测试的WWW最新文献列表可在http：//www.rspa.com找到。

① 对OO系统来说，在与真实世界不同的情况下，对使用实例（usecases）。进行的分析跟踪和模型设计是没有什么价值的。

① OO类的测试实例设计方法的讨论贯穿于第 22.4 到 22.6 节。

① 在大的实例中要用到OO设计概念。

② 第 22.4.3 到 22.4.7 中采用了rianMarick发表在Internet新闻组comp.testing中的文章。这的引用已获作者的许可，关于这些主题的进一步讨论，可参［MAR94］。

③ 在此的代码及下一节中使用了C＋＋语法，有关C＋＋的详细内容可参阅任何一本C＋＋资料，只要它好用

第 23 章　面向对象系统的技术度量

在本书前面我们提到，测度和度量是任何工程学科的关键构成成分——面向对象软件工程也不例外。可悲的是，对 00 系统的度量的使用比其他 00 方法的使用的进展要慢得多。EdBerard［BER95］提到如下对测度的讽刺：

软件人员似乎对度量有爱—恨关系。一方面，他们轻视和不信任任何听起来或看起来象是测度的东西，他们能迅速指出在谈论测度软件产品、软件过程和(特别地)软件人员的任何人们的论据中的“缺陷”。另一方面，同样是这些人员，他们似乎在标识哪些程序设计语言是最好的、管理者所做的“破坏”项目的最愚蠢的事情、以及谁的方法学适用于什么环境等方面又没有什么困难。

Berard 提到的“爱—恨关系”是真实的，然而，随着 00 系统变得更普遍深入，软件工程师具有量化的机制来评估设计的质量和 00 程序的效果是非常重要的。

23.1 面向对象度量的目的

面向对象度量的基本目标和那些针对传统软件的度量的目标是一样的：

- 更好地理解产品的质量。
- 评估过程的效果。
- 改善在项目级别完成的工作的质量。

每一个目标都很重要，对软件工程师而言，产品质量是最重要的。但是，我们如何测度 00 系统的质量？可以评估设计模型的哪些特征以确定系统是否易于

实现、便于测试、修改简单、最重要、为终端用户所接受？本章的下面部分将讨论这些问题。

23.2 区别性的特征

对任何工程产品的度量是由产品的独特特征所决定的。例如，对电动汽车计算每加仑多少英里是没有意义的，该度量对传统的(即，汽油为动力的)汽车是有效的，但是当推动模式发生巨大改变后，它是不适用的。面向对象的软件和用传统方法开发的软件有本质性不同，为此，对 OO 系统的技术度量必须调整以适应那些区别 OO 和传统软件的特征。

Berard [BER95] 定义了 5 个导致特殊度量的特征：局部化、封装、信息隐蔽、继承和对象抽象技术。在下面几节中，将简略地讨论每个特征。

23.2.1 局部化

局部化是软件的一个特征，它指明信息在程序中被集中的方式，例如，针对功能分解的传统方法围绕功能局部化信息，它们典型地以过程模块来实现。数据驱动方法围绕特定的数据结构局部化信息。在 OO 语境中，信息是通过封装数据和处理在类或对象的边界内而集中的。

因为传统软件强调函数为局部化机制，软件度量着重于函数的内部结构或复杂性(如，模块长度、内聚性或 cyclomatic 复杂性)或函数间相互连接的方式(如，模块耦合)。

因为类是OO系统的基本单位，所以，局部化是基于对象的，因此，度量应该应用于作为一个完全实体的类(对象)。此外，在操作(函数)和类间的关系不必要是一对一的。因此，反应类协作方式的度量必须能够适应一对多和多对一的关系。[①]

23.2.2 封装

Berard [BER95] 定义封装为“一组项的包装(或捆绑在一起)，(对传统软件的)低层封装例子包括记录和数组，而子程序(如，过程、函数、子例程和段落)是封装的中层机制。”

对 OO 系统，封装包含了类的责任，包括其属性(和针对聚合对象的其他类)和操作，以及由特定的属性值定义的类的状态。

封装通过将测度的焦点从单个模块改变到数据(属性)和处理模块(操作)包而影响度量。此外，封装鼓励在高抽象层的测度，例如，在本章后面，将介绍和

每个类的操作数量关联的度量。将此层次的抽象同传统的度量相比较，传统的着重于布尔条件的计数(cyclomatic 复杂性)或代码行数。

23.2.3 信息隐蔽

信息隐蔽隐瞒(或隐藏)程序构件的操作细节，只将对访问该构件必需的信息提供给那些希望访问它的其他构件。

良好设计的 OO 系统应该鼓励信息隐蔽，因此，指明隐蔽所达到程度的度量应该提供了对 OO 设计质量的一个指标。

23.2.4 继承

继承是使得某对象的责任能够传播到其他对象的机制，继承出现在类层次的所有层面上，通常，传统的软件不支持该特征。

因为继承是很多 OO 系统的关键特征，所以很多 OO 度量是关注于它的。例子(在本章后面讨论)包括子女数(类的直接子类的数量)、父辈数(类的立即一般化类的数量)。以及类层次嵌套级别(在继承层次中类的深度)。

23.2.5 抽象

抽象是使得设计者能够关注程序构件(数据或过程)的本质性细节而不需考虑低层细节的机制。如 Berard 所说：“抽象是一个相对概念，当我们移向更高的抽象级别时，我们忽略了越来越多的细节，即，我们提供了对概念或项的更一般化的视图；当我们移向抽象的低层时，我们引入了更多的细节，即，我们提供了概念或项的更特定的视图。”

因为类是一种抽象，它可以在很多不同的细节级别上并以一系列不同的方式(如，作为一个操作列表、作为一个状态序列、作为一系列协作)来观察，所以 OO 度量用类的测度(如，每个应用的每个类的实例数、每个应用的参数化类数以及参数化类和非参数化类的比率)来表示抽象。

23.3 对 OO 设计模型的度量

关于面向对象设计的很多东西是主观性的——一个有经验的设计者“知道”如何刻划 OO 系统使得它将有效地实现客户需求。但是，当 OO 设计模型在规模和复杂性上增长时，对设计特征的更客观的观察对有经验的设计者(他可获得更深的洞察)和新手(他可以得到关于质量的指标，否则是得不到这些指标的)都是有益的。

对设计的客观观察应该有量化的成分——从而导向 OO 度量。在现实中，OO 系统的技术度量不仅应用于设计模型，也可应用于分析模型。在下面几节中，我们探讨在 OO 类级别和操作级别提供质量指标的度量以及可应用于项目管理和测试的度量。

23.4 面向类的度量

类是 OO 系统的基本单位，因此，对个体类、类层次和类协作的测度和度量对必须评估设计质量的软件工程师将是无价的。在前几章，我们已经看到类封装操作(处理)和属性(数据)，类经常是子类(有时称为子女)的“父辈”，子类继承父类的属性和操作。类经常和其他类协作。这些特征的每一种均可作为测度的基础。

23.4.1 CK 度量套件

最广为引用的OO软件度量体系之一是由Chidamber和Kemerer [CHI94] 提出的，作者建议了U欠設O系统的 6 种基于类的设计度量(经常被称为CK度量套件(CK metrics suite))。[①]

每个类的加权方法(Weighted Methods for per Class， WMC)

假定对类C定义了复杂度为c_1，c_2，…c_n的n个方法，所选择的特定的复杂性度量(如cyclomatic复杂性)应该规范化，使得对某方法的名义上的复杂性取值 1.0。

$$WMC=\Sigma ci$$

对 i=1 或 n。

方法的数量和它们的复杂性是对实现和测试类所需的工作量的合理指标，此外，方法数量越多，继承树也越复杂(所有子类继承它们父类的方法)，最后，随着一个给定类的方法的数量的增多，它有可能变得越来越针对某一应用，因此限制了潜在的复用。由于这些理由，WMC 应该保持合适的低值。

虽然对类的方法数量的计数似乎是相当直接的，但此问题实际上比看上去更复杂。Churcher 和 Shepperd [CHU95] 对此问题有如下论述：

为了统计方法的数量，我们必须回答这样一个基本问题：“方法仅仅属于定义它的类呢，或者它也属于直接或间接继承它的每个类？”这样的问题似乎是微不足道的，因为运行时系统将最终解决此问题，然而，对度量的意义却可能是重大的。

一种可能性是限制对当前类计数，忽略继承来的方法。这样做的动机是：继承来的方法已经在定义它们的类中被计数，所以类增量是其功能性的最好测度——这反应了它存在的理由。为了理解一个类做什么，最重要的信息源是它自己的操作。如果一个类不能对某消息作出回应(即，它缺少自己的对应方法)，则它将消息传送到其父类。

在另一个极端，计数可能包括定义在当前类中的所有方法及所有继承来的方法。该方法强调在理解类时状态空间的重要性，而不是类增量的重要性。

在这两个极端之间，也存在其他可能性，例如，可能限制计数到当前类的和直接从父类继承来的方法。该方法基于这样一个论据：父类的特例化是同子类的行为最直接相关的。

和软件度量中的大多数计数惯例一样，上面概述的任何一种方法均是可接受的，只需当收集度量数据时，一致地应用计数方法即可。

继承树的深度(Depth of the Inheritance Tree，DIT)

该度量被定义为“从结点到树根的最大长度”[CHI94]，在图 23－1 中，所显示的类层次的 DIT 值是 4。

当 DIT 增大时，有可能低层的类将继承很多方法，这导致了企图预测类的行为时的潜在困难。一个深的类层次(DIT 值大)也导致更大的设计复杂性。从正面来说，大的 DIT 值表明很多方法可以被复用。

子女的数量(Number of Children，NOC)

在类层次中直接从属于某类的子类称为子女，在图 23－1 中，类C_2有三个子女——子类C_{21}、C_{22}和C_{23}。

当子女的数量增加时，复用也增加。但是，当 NOC 增大时，父类表示的抽象可能在减弱，即，有可能某些子女实际上不是父类的合适成员。当 NOC 增大时，测试的工作量(需要时对每个子女在其运行语境内进行测试)也将增大。

对象类之间的耦合(Coupling Between Object Classes，CBO)

CRC 模型(第 20 章)可被用于确定 CBO 的值，本质上，CBO 是对某个类在 CRC 索引卡片上列出的协作的数量。

当 CBO 增大时，有可能类的可复用性将减弱。CBO 的高值也使得修改和为确定何时进行修改的测试更为复杂。通常，每个类的 CBO 值应该保持适当的低值，这和传统软件中减少耦合的一般性指南是一致的。

对类的响应(RePonse for a Class，RFC)

类的响应集是“一组方法，它们可能会作为该类的某对象接收到的消息的响应而被执行”［CHI94］，RFC 被定义为响应集中方法的数量。该类的某对象接收到的消息的响应而被执行”

当 RFC 增大时，因为测试序列(第 22 章)增加，所以，测试所需的工作量也增大。同样，当 RFC 增大时，类的整体设计复杂性增加。

方法中内聚性的缺乏(Lack of Cohesion in Methods，LCOM)

类中的每个方法访问一个或多个属性(也称为实例变量)，LCOM是访问一个或多个相同属性的方法的数量，如果没有方法访问相同的属性，[①]则LCOM为 0。

为了说明 LCOM≠0 的情形，考虑具有 6 个方法的类，其中 4 个方法有一个或多个属性是共同的(即，它们访问共同属性)，因此，LCOM=4。

如果 LCOM 是高的，方法可以通过属性相互耦合，这增加了类设计的复杂性。通常，LCOM 的高值表明最好将该类分解为两个或更多的独立类。虽然存在 LCOM 的高值是合理的情形，但是，总希望保持高内聚性，即，保持 LCOM 值低。

23.4.2 Lorenz 和 Kidd 建议的度量

在他们关于OO度量的书中，Lorenz和Kidd［LOR94］将基于类的度量分为四大类：大小、继承、内部和外部。OO类的面向大小的度量着重于个体类的属性和操作的计数以及OO系统整体的平均值。基于继承的度量着重于在整个类层次中操作被复用的方式。类内部的度量考察内聚性(23.4.1 节)和面向代码的问题；外部度量检查耦合和复用。Lorenz和Kidd建议的度量体系的抽样如下：[②]

类大小(Class Size，CS)

可用下面测度来确定类的整体大小：

- 被封装在类中的操作的总数(包括继承来的和私有的事例操作)
- 被封装在类中的属性的数量(包括继承来的和私有的事例属性)

Chidamber 和 Kemerer(23.4.1 节)提出的 WMC 度量也是类大小的加权测度。

如前面提到，大的 CS 值指明类可能有太多的责任，它将降低类的复用性和使实现和测试复杂化。通常，在确定类的大小时，继承的和公共的操作和属性应该被更重地加权［LOR94］，私有的操作和属性造成特例化并在设计中是更局部化的。

也可以计算出类属性和操作数量的平均值。大小的平均值越低，系统中的类越可能被广泛地复用。

由子类重载的操作数量(Number of Operations Overridden by a Subclass，NOO)

存在这样一种情形，子类用它自己使用的特殊版本替换从其超类继承的某操作，这称为重载。大的 NOO 值通常指明了，如 Lorenz 和 Kidd 所指出的设计问题：

因为子类应该是其超类的特例化，它应该主要扩展超类的服务(操作)，这将导致独特的新的方法名。

如果 NOO 是大的，则设计者破坏了超类所蕴含的抽象，这导致了弱的类层次并可能使 OO 软件难于测试和修改。

由子类增加的操作的数量(Number of Operations Added by a Subclass，NOA)

子类通过加入私有的操作和属性而特例化。当 NOA 的值增大时，则子类将漂离超类所蕴含的抽象。通常，当类层次的深度增加(DIT 变大)时，在层次中低层的 NOA 值将下降。

特例化索引(Specialization Index，SI)

特例化索引提供了在 OO 系统中每个子类的特例化程度的粗略指标。可通过加入或删除操作或通过重载而达到特例化。

$$SI=[NOO\times level]/Mtotal$$

这里 level 是在类层次中类驻留的层数，Mtotal 是类的方法的总数。SI 的值越高，类层次中越有可能包含了更多的不遵从超类的抽象的类。

23.5 面向操作的度量

因为类是 OO 系统中的支配性单元，很少有针对类操作提出的度量。Churcher 和 Shepperd [CHU95] 对此有如下论述： 苌儆姓攵岳嗖僮魈岢龅亩攘俊 hurcher 和当前研究的结果表明：方法在语句数量和逻辑复杂性 [WIL92] 方面趋向减小，这意味着系统的连接性结构可能比个体模块的内容更为重要。

然而，通过检查类操作的一般特征可以获得某些深入了解，Lorenz 和 Kidd [LOR94] 提出了统如下的三个简单度量：

平均操作大小(Average operation size，OSavg)虽然代码行可被用作操作大小的指标，但是，LOC 测度受第 4 章中讨论的所有问题的影响，为此，操作发送的消息数量提供对操作大小的另一种度量。当某单个操作发送的消息数量增加时，有可能责任没有在类中很好地分配。

操作复杂度(Operation complexity，OC)可使用对传统软件提出的任何复杂度度量(第 18 章)来计算操作的复杂度［ZUS90］。因为操作应该被限制到某一特定的责任，设计者应该努力保持 OC18 章)来尽可能低。

每个操作的平均参数的数量(AveragenumberofParametersPeroperation，NPavg)操作参数的数量越大，对象间的协作越复杂，通常，NPavg 应该保持尽可能低。

23.6 对面向对象测试的度量

在 23.4 和 23.5 节给出的设计度量提供了设计质量的指标，它们也提供了测试 OO 系统所需的测试工作量的一般性指标。

Binder［BIN94］建议了一组设计度量，它对 OO 系统的“易测试性”有直接的影响，该度量谋蛔橹　　从 x 匾　杓铺卣鞴睦啾穑*

封装

在方法中内聚性的缺乏(LCOM)。[①]LCOM的值越高，则必须测试越多的状态以保证方法不会产生副作用。

公共和受保护属性的百分比(Percent public and protected，PAP)。公共属性是从其他类继承来的，因此对那些类是可见的。受保护属性是特例化的，为特定的类所私有。该度量指明公共属性的百分比，PAP 的高值增加了类间副作用的可能性，必须设计测试以保证发现这样的副作用。

对数据成员的公共访问(Public access to data members，PAD)。该度量指明可以访问另一个类的属性的类(或方法)的数量，是对封装的违背。PAD 的高值导致了类间副作用的潜在可能，必须设计测试以保证发现这样的副作用。

继承

根类的数量(Number of root classes，NOR)。该度量是在设计模型中描述的不同类层次的数量，必须为每个根类和对应的类层次开发测试。随着 NOR 增加，测试工作量也增加。

扇入(Fan in，FIN)。当用于 OO 语境时，扇入是多继承的指标，FIN＞1 指明类从多于一个的根类继承属性和操作。FIN＞1 应该尽可能避免。

子女数(NOC)和继承树的深度(DIT)。[①]如我们在第 22 章中所讨论，超类的方法将必须针对每个子类被重新测试。

除了上面的度量，Binder［BIN94］还定义了对类复杂度和多态性的度量。对类复杂度的度攵粤堪　匸　鲧 K 度量(23.4.1 节)：每个类的加权方法(WMC)、

对象类间的耦合(CBO)、以及对类的回应(RFC)。此外，也定义了和方法计数关联的度量。和多态性关联的度量是非常特殊的，对它的讨论参见 Binder。

23.7 对面向对象项目的度量

如我们在本书第二部分所发现那样，项目管理者的工作是计划、协调、跟踪和控制软件项目。在第 19 章，我们讨论了某些和 OO 项目的项目管理关联的特殊问题。但是测度又怎样呢？有特殊的 OO 度量可被项目管理者使用以提供对进展的更深入洞察吗？当然，回答是“是”。

项目管理者完成的第一个活动是计划，而早期的计划任务之一是估算。回忆演化过程模型，[②]在软件的每次递进后被重新考察计划，因此，在OOA、OOD、以及甚至OOP的每次递进后被重新考察计划及其项目估算。

项目管理者在计划过程中面临的关键问题之一是对软件的实现大小的估算，大小是直接正比于工作量和时间的。下面的 OO 度量 [LOR94] 可以提供对软件大小的考察：

场景脚本的数量(Number of scenario scripts，NSS)。场景脚本或使用实例(第 20 章)的数量直接正比于满足需求所需的类的数量、每个类的状态数量、以及方法、属性和协作的数量。NSS 是对程序大小的有力的指标。

关键类的数量(Number of key classes，NKC)。关键类直接聚焦于问题的业务领域并且通过复用[③]而实现它们的可能性将很低。为此，HKC的高值表明了仍要进行实质性的开发工作量。Lorenz和Kidd [LOR94] 指出在典型的OO系统中所有类的 20％到 40％是关键类，其余的支持基础设施(GUI、通信、数据库、等等)。

子系统的数量(Number of subsystems，NSUB)。子系统的数量提供了对资源分配、进度安排(特别强调并行开发)以及整体集成工作量的考察。

度量 NSS、NKC 和 NSUB 可针对过去的 OO 项目来收集，并且和花费在整体项目上的工作量以及花费在单个过程活动(如，OOA、OOD、OOP 和 OOT)上的工作量相关联，这些数据也可结合本章前面讨论的设计度量一起用于计算“生产率度量”，如每个开发者的类的平均数量或每个人月的方法的平均数量，这些度量可结合起来用于针对当前项目估算工作量、工期、人员配备、以及其他项目信息。

23.8 小结

面向对象软件和用传统方法开发的软件有根本性不同，因此，OO 系统的度量着重于以下度量：可以应用于类和用于那些使得类独特的设计特征——局部化、封装、信息隐蔽、继承和对象抽象技术的度量。

CK 度量套件定义了 6 个面向类的软件度量，它们着重于类和类层次。该度量套件也给出了评估类间协作和驻留在类中的方法的内聚性的度量。在面向类的级别上，CK 度量套件可以用 Lorenz 和 Kidd 提出的度量来增强，这些包括类“大小”的测度和对子类的特例化程度提供考察的度量。

面向操作的度量着重于个体操作的大小和复杂性，然而，重要的是要注意，对 OO 设计度量的主要切入点在类级别。

已经提出了大量的 OO 度量以评估 OO 系统的易测试性，这些度量着重于封装、继承、类复杂度和多态性。

分析和设计模型的可测度的特征可以帮助 OO 系统的项目管理者的计划和跟踪活动，场景脚本(使用实例)、关键类和子系统的数量一起提供了关于实现系统所需的工作量的信息。

思考题

23.1 复审在本章和第 18 章提出的度量，你如何刻划传统软件和 OO 软件的度量间在语法和语义上的不同？

23.2 局部化如何影响对传统软件和 OO 软件所开发的度量？

23.3 为什么不更多地强调那些涉及类中操作的特定特征的 OO 度量？

23.4 复审在本章讨论的度量并建议一些直接或间接地测度信息隐蔽的度量。

23.5 复审在本章讨论的度量并建议一些直接或间接地测度抽象的度量。

23.6 类 X 有 12 个操作，对在 OO 系统中的所有操作已经计算出 Cyclomatic 复杂度，并且模块复杂度的平均值是 4。对类 X，操作 1 到 12 的复杂度分别是 5、4、3、3、6、8、2、2、5、5、4、4，计算 WMC。

23.7 根据图 19－8a 和 19－8b，计算每个继承树的 DIT 值，类 X2 对两棵树的 NOC 值是多少？

23.8 根据 [CHI94] 给出一页的关于 LOCM 度量的正式定义的讨论。棵树的 NOC 值是多少？

23.9 在图 19-8b 中，类 X3 和 X4 的 NOA 值是多少？

23.10 根据图 19-8b，假定在继承树(类层次)中已经重载了 4 个操作，该类层次的 SI 值是多少？

23.11 某软件项目组到目前已经完成了 5 个项目，已经对所有项目收集了下面的数据：

项目号	NSS	NKC	NSUB	工作量(天)
1	34	60	3	900
2	55	75	6	1575
3	122	260	8	4420
4	45	66	2	990
5	80	124	6	2480

一个新项目正处于 OOA 的早期阶段，估计将会为该项目开发 95 个使用实例，估算：

a.实现系统所需要的类的总数。

b.实现系统所需的总的工作量。

23.12 由老师选出本章的一组 OO 度量，对下面的一个或多个问题计算这些度量的值：a.SafeHome 系统的设计模型。

b.思考题 12.13 中描述的 PHTRS 系统的设计模型。

c.思考题 21.14 中考虑的视频游戏的设计模型。

d.思考题 21.15 中考虑的 e-mail 系统的设计模型。

e.思考题 21.16 中考虑的 ATC 系统的设计模型。

推荐阅读文献及其他信息源

一系列关于 OOA、OOD 和 OOT 的书(见第 20、21、22 章的“推荐阅读文献及其他信息源”)对 OO 度量均有一定涉及，但是，很少详细地讨论该主题。Jacobson(Object－Oriented SoftwareEngineering， Addison-Wesley，1994)和 Graham(Object－Oriented Methods，Addison－Wesley，2ndedition，1993)的书比其他书进行了更多的讨论。

Lorenz 和 Kidd [LOR94] 和 Henderson－Sellers(Object－Oriented Metrics：Measures of Complexity，Prentice－Hall，1996)是当前出版的关于 OO 度量的唯一书。Whitmire(Encyclopedia of SoftwareEngineering，J.Marciniak(ed.)，Wiley，1994)其中一章是关于 OO 度量的。其他针对传统软件度量的书籍(见第 4、18 章的“推荐阅读文献及其他信息源”)包含有限的对 OO 度量的讨论。

关于 OO 度量的主要信息可从出版于技术文献中的论文和文章中得到，Whitty(Object－oriented Metrics：People and Publications，1995)收集了当前已出版的关于 OO 度量的最全面的参考书目，其电子版本可在如下网址得到：

http：//www.sbu.ac.uk/～csse/publications/OOMetrics.html

本书第四部分提到的所有 Internet 信息源对 OO 度量的讨论是值得探查的，此外，关于 OO 度量的偶尔的讨论可在 Internet 新闻组 comp.object、comp.software－eng、comp.specification、和 comp.testing 找到。

其他的关于 OO 度量的信息可在下面网址找到：

http：//www.sbu.ac.uk/～csse/metkit.html

http：//louis.ecs.soton.sc.uk/dsse/moops.html

ftp：//ftp.sbu.ac.uk/pub/Metrics

关于面向对象度量的WWW最新文献列表可在http：//www.rspa.com找到。

① 在此的讨论引自［ER95］。

① Chidember和Kemerer用的是术语“方法”，而不是“操作”本节中使用了这一术语。

① 正式定义更为复杂。详情可参阅［CHI94］

② 更全面的讨论请参阅［LOR94］。

① LCOM的描述参见第 23.4.1 节。

① NOC和DIT的描述参见第 23.4.1 节。

② 演化过程模型最适于OO系统（参见第 2 章和第 19 章）。

③ 直到一个特殊领域中的复用构件的强健复用类库被开发出来时，复用才真正得于实现。

第五部分　软件工程高级课题

这部分我们考虑一些高级课题，这将扩展你对软件工程的理解。在下面章节中，我们将讨论下列问题：

- 什么是“形式化方法”，如何用于刻划软件？
- 对形式化描述软件需要什么符号和数学预备知识？
- 净室软件工程方法和传统方法有什么不同？
- 什么是在净室过程中进行的关键技术活动？
- 领域工程如何被用作建立可复用构件库的先导？
- 当开始进行复用过程时，必须考虑哪些技术问题？

• 什么是支持复用的经济论据？

• 什么是业务过程再工程？它如何设定软件再工程的阶段？

• 什么是软件再工程所需要的关键技术活动？

• 客户/服务器体系结构如何影响软件开发的方式？

• 什么是建立 CASE 工具环境的体系结构选项？

• 什么软件工程的发展方向？

一旦回答了这些问题，你将理解了在下个十年中对软件工程有深远影响的课题。

第 24 章　形式化方法

可以根据形式化程度对软件工程方法进行分类。本书前面讨论的分析和设计方法将位于从非形式化到适度严格之间的区段。运用图、文本、表格和简单的符号的结合来创建分析和设计模型。

我们现在考虑形式化程度的另一端，这里以刻划系统功能和行为的形式化的语法和语义来描述规约和设计一个形式化规约在形式上是数学的(例，谓词演算可用作形式化规约语言的基础)。

Anthony Hall 在其关于形式化方法的引导性讨论［HAL90］中陈述：餍问交嬖加镅缘幕*

形式化方法是有争议的。它们的支持者宣称：它们可以引发软件开发的革命。而批评者认为：这是极端困难的。同时，对大多数人来说，他们对形式化方法是如此的不熟悉，以至难于判断这些争论。

在本章，我们探索形式化方法并检查它们对未来几年中软件工程的潜在影响。

24.1 基本概念

The Encyclopedia of Software Engineering［MAR94］中对形式化方法的定义如下：于

用于开发计算机系统的形式化方法是描述系统性质的基于数学的技术。这样的形式化方法提供了一个框架，人们可以在框架中以系统的而不是特别的方式刻划、开发和验证系统。

如果一个方法有良好的数学基础，那么它是形式化的，典型地以形式化规约语言给出的。这个基础提供一系列精确定义的概念，如：一致性和完整性，以及更进一步，定义规约、实现和正确性。

形式化规约的希望性质——无二义性、一致性和完整性——是所有规约方法的目标。然而，形式化方法的使用导致了达到这些理想的更高的可能性。规约语言的形式化语法(24.4 节)使得需求或设计以唯一的方式被解释，从而排除了当自然语言(例如英语)或图形符号被读者解释时经常产生的二义性。集合论和逻辑符号的描述机制(24.2 节)使得可以清晰地陈述事实(需求)。为了一致，在规约中某地方陈述的事实不能与其他地方有矛盾，一致性是通过对初始事实可以形式化地映射(使用推理规则)到规约中后面的陈述的数学证明来保证的。

即使使用形式化方法，完整性也是难于达到的。当创建规约时，系统的某些方面可能未被定义；某些特征可能被有意识地省略以允许设计者在选择实现方法时具有一定自由度；而且，在一个大型复杂系统中不可能考虑以每一个操作场景。可能仅仅是由于错误而忽略了某些事。

虽然数学提供的形式化对某些软件工程师有吸引力，其他人(某些人说是大部分)却对软件开发的数学视图怀有疑问。为了了解为什么形式化方法有益于软件开发，我们必须首先考虑和欠形式化方法相关的不足。

24.1.1 少量的形式化方法的不足

本书第三、第四部分中讨论的分析、设计方法大量使用自然语言和多种图形符号。虽然对分析、设计方法小心地运用，并结合彻底的复审可以也确实导致了高质量软件的开发，但是，这些方法应用的偏差可能产生各种问题。系统规约中可能包含矛盾、二义性、含糊性、不完整陈述以及抽象层次的混杂等。

矛盾指一组相互有分歧的陈述的集合。例如，系统规约的某部分可能规定系统必须监控化学反应堆中的所有温度，而另一部分规约，可能由另一个成员撰写，可能规定只监控在一定范围内的温度。正常地，很容易查出出现在系统规约同一页的矛盾，然而，矛盾通常被很多页分开。

二义性指可以以不同方式解释的陈述。例如，下面陈述具有二义性：

操作员标识由操作员姓名和口令组成，口令由 6 位数字构成。它应该在安全 VDU 上显示，并且当操作员登录进系统时被存放在注册文件中。

在这段摘录中，“它”到底代表“口令”还是“操作员标识”？

因为系统规约是非常庞大的文档，所以含糊性经常发生。要一致地达到高层次的精度是几乎不可能完成的任务。它可能导致这样的陈述：“由雷达操作员使用的系统界面应该是用户友好的”或“虚拟界面将基于简单的整体概念，这些概

念是同理解和使用直接相关的，是在数量上很少的”。不经意的阅读这些陈述将不可能发现这样一个事实：这些陈述实际上缺少任何有用的信息。

不完整性可能是系统规约中最经常发生的问题之一。例如，考虑如下功能需求：

系统应该从位于水库中的深度传感器每小时获取一次水库深度，这些值应保留 6 个月。

这描述了系统的主要数据存储部分。假设系统的某命令是：

AVERAGE 命令的功能是在 PC 上对某特定传感器显示在两个日子间的平均水深。

如果在这个命令中没有更多的细节，那么，命令的细节将是严重不完整的。例如，命令的描述中没有包括：如果系统的用户给定的时间是在当前时间的 6 个月以前，那么，将发生什么事件？

抽象层次的混杂指非常抽象的陈述中随机地混杂了一些关于细节的低层次的陈述。例如，如下陈述：

系统的目的是跟踪仓库中的库存。

可能混杂了下面的陈述：

当店员输入命令 withdraw 后，他将输入订单号、被移走的物项的标识以及移走的数量。系统将回答允许移走的确认。

虽然这样的陈述在系统规约中是重要的，但是，规约撰写者常常将它们混杂到如此程度以致非常难于看清系统的整体功能结构。

以上提到的每个问题都是很常见的，每个问题都展示了传统的和面向对象的规约方法的潜在不足。

24.1.2 软件开发中的数学

数学对大型系统的开发者来说有许多有用的性质。其中最有用的性质之一是它能够简洁而准确地描述物理现象、对象或某动作的结果。当一个应用数学家称某特定方程的解由如下积分给出时：

$$\text{ftan}\left[x^2+\exp(\cos(x))\right]$$

对其语义不存在争议。虽然如何求解积分可能需要大量努力，但求解积分的人准确地知道需要什么。例如，可能采用分析解法，或者如果已经知道积分的上、下限，则可能需要仿真或其他数值方法。理想情况下，软件开发者应该和应用数

学家处于相同的地位，他应该得到系统的数学规约，并应该开发出实现该规约的以软件体系结构形式开发的解答[①]。

在软件开发过程中使用数学的另一个优点是：它提供了在软件工程活动间的平滑过渡。不仅功能规约、而且系统设计也可以用数学表达，当然，程序代码也是数学符号——虽然相当的冗长。

数学的主要性质是它支持抽象，而且是优秀的建模介质。因为它是准确的几乎没有二义性的介质，规约可以被数学地验证以揭示矛盾性和不完整性，而且含糊性完全消失。此外，数学可用于以有组织的方式表示系统规约中的抽象层次。

数学是理想的建模工具。它使得规约的本质可以被展示出来，帮助分析员和系统规约撰写者确认功能规约，而不需涉及反应时间、设计指示、实现指示以及项目约束等问题。它也可以帮助设计者，因为系统设计规约展示了模型的性质，提供最合适的使任务被执行的细节。

数学作为软件开发工具的最后一个优点是：它提供了高层确认的手段，可以使用数学证明来展示设计和规约匹配、以及某些程序代码是对设计的正确反映。

24.1.3 形式化方法概念

本节的目的是给出软件系统的数学化规约中涉及的主要概念，而不是给读者枚举太多的数学细节。为此，我们使用一些简单例子：

例 1：符号表 程序被用于维护一个符号表，经常在许多不同类型的应用中使用此符号表，它由一组没有重复的项构成。图 24－1 给出了一个典型的符号表例子，它表示了操作系统用于保持系统用户名的表。其他表的例子包括：工资管理系统中的职员名表、网络通讯系统中的计算机名表、以及铁路时刻表生成系统中的目的地名表等。

假设在此例所示的表中包括不多于 Max2D 的用户名。这一陈述为表设定了一个限制，这是称为数据不变式(一个重要的贯穿本章的概念)的条件的一个构成成分。

数据不变式是一个条件，它在包含一组数据的系统的执行过程中总保持为真。上面讨论的符号表的数据不变式有两个构成成分：(1)表中包含的名字数不超过 MaxIds 和(2)在表中没有重复的名字。在上面描述的符号表程序的情形下，这意味着，在系统执行过程中无论什么时候检查符号表，它将总是包含不超过 MaxIds 的职员标识符，并且没有重复。

另一个重要的概念是状态，在形式化方法[①]的语境中，状态是系统访问和修改的存储数据。在符号表程序的例子中，状态是符号表。

最后一个概念是操作，这是在系统中发生的读或写状态数据的动作。如果对符号表程序考虑从符号表加入或移出职员名，则它将关联两个操作：一个加一个指定名到符号表中的操作，一个从表中移出一个现存名的操作。如果程序提供检查是否某指定名包含在表中的机制，则有一个返回某种表示名字是否在表中的指示值的操作。

一个操作和两个条件相关联：前置条件和后置条件。前置条件定义某特定操作在其中合法的环境，例如，对加一个名字到职员标识符符号表中的操作，仅当表中不含有将被加入的名字，而且在表中只有少于 MaxIds 的职员标识符其前置条件才是有效的。操作的后置条件定义当操作完成后所产生的现象，是通过其对状态的影响来定义的。在加标识符到职员标识符符号表的操作的例子中，后置条件将数学地刻划表已经增加了新标识符。

例 2：块处理器 在操作系统中一个更重要的部分是维护由用户创建的文件的子系统，块处理器是文件子系统中的一部分。文件存储中的文件由存储设备上的存储块构成，在计算机的操作中，文件被创建和删除，需要存储块的获取和释放。为了处理这些，文件子系统维持一个未用块池，并将保持对当前使用块的跟踪。当块从被删除文件释放时，它们通常被加入到等待进入未用块池的块队列中。如图 24—2 所示，图中显示了一些部件：未用块池、被操作系统管理的文件使用的块、以及那些等待被加入到未用块池中的块。等待块被组织为队列，队列中每个元素包含来自某被删除文件的一组块。

对这个子系统而言，状态是自由块的集合、已用块的集合、以及返回块的队列，数据不变式用自然语言表达如下：

- 没有块同时被标记为未用和已用。
- 所有在队列中的块集合将是当前已用块集合的子集。
- 没有队列元素包含相同的块号。
- 已用块和未用块的集合将是组成文件的块的总集。
- 在未用块集合中没有重复的块号。
- 在已用块集合中没有重复的块号。

和子系统关联的某些操作如下：

- 将一个块集合加到队列尾。
- 从队列前面移走一个已用块集合并将其放到未用块集合中。
- 检查是否块队列为空。

第一个操作的前置条件是:将被加入的块必须在已用块集合中,后置条件是:块集合必须被加入到队列尾部。

第二个操作的前置条件是：队列中必须至少有一项，后置条件是：块必须被加到未用块集合中。

最后一个操作——检查是否返回块的队列为空——没有前置条件,这意味着不管状态具有什么值，操作总是有定义的。后置条件是：如果队列为空，返回 true，否则，返回 false。

例 3：脱机打印管理器 在多任务操作系统中，一组任务请求打印文件。通常，没有足够的打印设备同时满足所有的打印请求，任何不能被立即满足的打印请求被放置在等待打印队列中。操作系统处理这样的队列管理的部分称为脱机打印管理器(Print spooler)。

在这个例子中，我们假定操作系统拥有不超过 MaxDevs 个输出设备，每个设备有一个关联队列。我们还将假定每个设备关联一个对其将打印的文件行数的限制，例如，某打印行数限制为 1000 的输出设备将关联一个包含不超过 1000 行文本的队列的文件。脱机打印管理器有时强加这个限制以禁止大的打印任务，这些任务可能占用低速的打印设备非常长的时间。图 24－3 是脱机打印管理器的示意性表示。

如图所示，管理器状态包含四个部分：等待打印的文件队列，每个队列被关联到某特定的输出设备；管理器控制的输出设备集合；输出设备和可以打印的最大文件行数的关系；以及等待打印的文件和它们行数的关系。例如，图 24－3 显示输出设备 LP1，其打印行限是 750 行，有两个文件 ftax 和 persons 等待打印，行数分别是 650 行和 700 行。

脱机打印管理器的状态被表示为四个部分：Queues，OutputDevices，Limits 和 Sizes。其时间不变式包括五个部分：

- 每个输出设备关联一个打印行数的上限。
- 每个输出设备关联一个可能非空的等待打印的文件队列。
- 每个文件关联行数大小。
- 和某输出设备关联的每个队列包含行数大小小于输出设备上限的文件。
- 管理器将管理不超过 MaxDevs 个输出设备。

一系列操作和脱机打印管理器相关联，例如：

- 将新的输出设备及其关联的打印限制加入到脱机打印管理器中。
- 从和特定输出设备关联的队列中移走一个文件。

• 将一个文件加入到和特定输出设备关联的队列中。

• 修改某特定输出设备的打印行数上限。

• 从和某输出设备关联的队列中移一个文件到和另一个输出设备关联的队列中。

每个操作对应于打印管理器的一项功能，如，第一个操作对应于管理器被通报一个新设备。

如前，每个操作同前置和后置条件关联。例如，第一个操作的前置条件是：输出设备名尚不存在且当前打印管理器知道的输出设备数少于 MaxDevs。后置条件是：新设备的名字被加入到现存设备名集合中，形成一个新的设备项且没有文件和其队列关联，以及设备和它的到打印限制关联。

第二个操作(从和特定输出设备关联的队列中移走一个文件)的前置条件是：设备是管理器已知的且在同设备关联的队列中至少有一项。后置条件是：和输出设备关联的队列的头被移走且在管理器中跟踪文件大小的项被删去。

上面描述的第五个操作(从和某输出设备关联的队列中移一个文件到和另一个输出设备关联的另一队列中)的前置条件是：

• 第一个输出设备对管理器已知。

• 第二个输出设备对管理器已知。

• 和第一个设备关联的队列中包含了将被移动的文件。

• 文件大小小于或等于和第二个设备关联的打印限制。

后置条件是：文件被从一个队列中移走并被加入到另一个队列中。

在本节提到的每个例子中，我们已介绍了形式化规约的关键概念。在没有强调使规约形式化所需的数学下，完成了这些介绍。在下一节中我们将考虑这些数学。

24.2 数学预备知识

为了有效地应用形式化方法，软件工程师必须具有对与集合和序列相关的数学符号以及用于谓词演算中的逻辑符号的使用知识。本节的目的是做一介绍，更详细的讨论，读者可以参考相关专著(例如参考文献［WIL87］、［GRI93］以及［ROS95］)。

24.2.1 集合和构造性规约

集合是对象或元素的聚集。集合论是形式化方法的基础。集合中包含的元素是唯一的(即，不允许重复)。具有少量元素的集合用花括号({，})括起来，元素间用逗号分开。例如，这是一个包含 4 个元素的自然数集合：

$$\{7，14，3，12\}$$

下面的集合包含五种程序设计语言的名字：

$$\{C＋＋，\ Pascal，\ Ada，\ COBOL，\ C\}$$

下面的数的聚集不是集合，因为它包含了重复的元素 2：

$$\{13，2，9，11，12，88，2\}$$

集合中元素出现的顺序是不重要的，集合中元素的数量称为集合的基数(cardinality)，操作符#返回集合的基数，例如，下面的表达式说明基数操作符被用于已知集合，其结果指出集合中项的数量：

$$\#\{A，\ B，\ C，\ D\} = 4$$

有两种定义集合的方式，一是通过枚举出集合的元素来定义(如上面的集合定义)，二是创建一个构造性集合规约，用布尔表达式来刻划集合成员的一般形式。因为集合性规约可以为大集合提供简洁的定义，所以它比枚举方式更受到青睐，它也显式地定义了用于构造集合的规则。

下面是一个构造性规约的例子：

$$\{n：N \mid n< 3 \cdot n\}$$

规约中有三个部分，一个基调 n：N，一个谓词 n＜3，以及一个项 n。基调刻划在形成集合时考虑的值的范围，谓词(布尔表达式)定义集合如何被构造，项则给出集合中项的一般形式。在上面例子中，N 表示自然数，这样自然数将被考虑；谓词指出只有小于 3 的自然数被包含在集合中；项规定集合中每个元素的形式为 n。这样，上面的规约定义集合为：

$$\{0，1，2\}$$

当集合元素的形式是明显的时候，项可以省略，例如，上面集合可表示为：

$$\{n：N \mid n < 3\}$$

上述的集合均只含单项元素。集合元素可以是对，三元组等等，例如，集合定义为：

$$\{x，y：N \mid x＋y＝10×(x，y^2)\}$$

描述了形为(x，y■2)的自然数对的集合，这里 x 和 y 之和是 10，下面是此集合：

{(1，81)，(2，64)，(3，49)，…}

另一个构造性集合规约的例子是自然数对的集合，这里第二个元素比第一个元素大 3 并且第一个元素大于 120，规约如下：

{n：N | n＞120×(n，n＋3)}

在上面例子中，试图使用两个变量，然而因为对的第二个元素可以唯一地用第一个元素来表达，所以这是不需要的。

明显地，表示某些计算机软件成分所需的构造性集合规约将比上面的例子复杂得多，然而，其基本形式和结构是相同的。

24.2.2 集合运算符

使用一个特定的符号集来表示集合和逻辑操作，想应用形式化方法的软件工程师必须理解这些符号。

运算符∈用于表示集合中的成员关系，例如，如果 x 是集合 X 中的成员，表达式

$$x \in X$$

的值为真，否则其值为假。例如，谓词

$$12 \in \{6，1，12，22\}$$

值为真，因为 12 是集合中成员。

运算符∈ 的反是运算符 ∉，如果x不是集合X的成员，表达式

$$x \notin X$$

的值为真，否则为假。例如，谓词

$$13 \notin \{13，1，124，22\}$$

值为假。

运算符 ⊂ 和 ⊆ 将集合作为操作对象，如果集合A的成员均包含在集合

B 中，谓词

$$A \subset B$$

的值为真，否则值为假。这样，谓词

$$\{1, 2\} \subset \{4, 3, 1, 2\}$$

值为真。然而，谓词

$$\{HD1, LP4, RC5\} \subset \{HD1, RC2, HD3, LP1, LP4, LP6\}$$

值为假，因为元素 RC5 没有包含在运算符右边的集合中。

运算符 $\subset$ 类似于 $\subset$，然而，如果它的操作对象相等，则它的值为假，

这样，谓词

$$\{HD1, LP4, RC5\} \subseteq \{HD1, RC5, HD3, LP1, LP4, LP6\}$$

值为真，而谓词

$$\{HD1, LP4, RC5\} \subseteq \{HD1, LP4, RC5\}$$

值为假。

一个特殊的集合是空集 ϕ，对应普通数学中的 0，空集具有它是所有集合的子集的性质，两个涉及空集的有用的等式是，对任何集合 A：

$$\phi \cup A=A \text{ 和 } \phi \cap A=\phi$$

这些等式直接来自下面给出的∪和∩的定义。

有一组二元运算符将集合作为它们的操作对象，并且以集合为结果。三个最常见的运算符是:并运算符∪,有时称作“cup”;交运算符∩,有时称作“cap”;以及集合差运算符＼。

并运算符以两个集合为操作对象，结果为一个集合，它包含在两个集合中的所有元素(排除重复)。这样，表达式

{File1, File2, Tax, Compiler} ∪ {NewTax, D2, D3, File2}

的结果为集合

{File1， File2， Tax， Compiler，NewTax，D2，D3}

交运算符以两个集合为操作对象，结果为一个集合，它包含每个集合中的公共元素。这样，表达式

{12，4，99，1} ∩ {1，13，12，77}

结果为集合

{12,1}

集合差运算符，观名知意，结果为从其第一个操作对象中移走第二个操作对象中的包含元素。这样，表达式

{New，Old，TaxFile，Sysparam} \ {Old，Sysparam}

的结果为集合

{New，TaxFile}

下面表达式的值为空集 Φ：

{a，b，c，d} ∩ {x，y}

此运算符总是产生一个集合，然而，在此情形下，两个操作对象间没有公共元素，因此，结果集合中也没有元素。

最后一个运算符是交积(cross product)×，有时也称为笛卡尔积。它以两个集合操作对象，其结果是对的集合，这里每个对由来自第一个操作对象的元素和来自第二个操作对象的元素构成。下面表达式是交积的例子：

{1，2} × {4，5，6}

该表达式的结果为：

{(1，4)，(1，5)，(1，6)，(2，4)，(2，5)，(2，6)}

注意，第一操作数的每个元素均和第二操作数的每个元素组合。

对形式化方法很重要的一个概念是幂集(powerset)，一个集合的幂集是一个，元素是该集合的子集的集合。本章中用来表示幂集运算符的符号是 P，这是一个单元运算符，当用于某集合时，返回其操作对象的子集的集合，例如：

P {1，2，3}

的结果为：

{φ，{1}，{2}，{3}，{1，2}，{1，3}，{2，3}，{1，2，3}}

其中所有元素均是集合{1，2，3}的子集。

24.2.3 逻辑运算符

形式化方法的另一个重要成分是逻辑：关于真、假表达式的代数。常见逻辑运算符的意义是每个软件工程师都了解的，然而，和常见的程序设计语言关联的逻辑运算符是用键盘上已有的符号表示的，和这些符号等价的数学运算符显示在表24－1中。唯一在程序设计语言符号中经常被忘记的逻辑运

算符是蕴含(implies)运算符 ⇒，它有两个操作数，读作“第一操作数蕴含

第二操作数”，例如：

$$x>10 \Rightarrow x>8$$

读作“x大于10蕴含x大于8”。

表24－1　常见逻辑运算符

and	∧
or	∨
not	¬
implies	⇒

全称量化是对集合中元素的一种陈述方法，该陈述对集合中每个元素均

为真。全称量化使用符号∀，一个例子如下：

$$\forall i,j:\mathbb{N} \bullet i > j \Rightarrow i^2 > j^2$$

该表达式陈述的是：对在自然数集合中的每一个值对，如果i大于j，则i的平方大于j的平方。

24.2.4 序列

序列是一种数学结构，对元素是有序的这一事实建模。一个序列 s 是对的集合，它的元素从 1 到最高的数排序，例如：

{(1, Jones), (2, Wilson), (3, Shapiro), (4, Estavez)}

是一个序列。形成对的第一个元素的项总称为序列的定义域，第二个元素总称为序列的值域。本书中序列用尖括号指明。例如，上面的序列写作：

＜Jones，Wilson，Shapiro，Estavez＞

和集合不同，序列中允许元素的重复，且序列的顺序是重要的。这样有，

〈Jones，Wilson，Shapiro〉≠〈Jones，Shapiro，Wilson〉

和

〈Jones， Wilson，Wilson〉≠〈Jones， Wilson〉

空序列表示为〈〉。

在形式化规约中使用一组序列运算符。连接(Catenation)⌢是一个二元运算符，它通过将第二个操作对象加到第一个操作对象的尾部而构造成新的序列。例如：

〈2，3，34，1〉⌢〈12，33，34，200〉

的结果为序列：

〈2，3，34，1，12，33，34，200〉

其他的序列运算符有 head， tail， front 和 last。运算符 head 抽取出序列的第一个元素； tail 对一个长度为 n 的序列返回其后面的 n-1 个元素所形成的序列； front 对一个长度为 n 的序列返回其前面的 n-1 个元素形成的序列；last 抽取出序列的最后一个元素。例如：

head＜2，3，34，1，99，101＞=2

tail＜2，3，34，1， 99， 101＞=＜3， 34，1，99， 101＞

last＜2， 3， 34，1， 99，101＞=101

front＜2，3，34，1，99，101＞=＜2，3，34，1，99＞

因为序列是对的集合，所有在 24.2.2 节中描述的集合运算符均可用于序列。当一个序列被用在状态中，它应当通过关键字 seq 指明。例如：

FileList：seq FILES

NoUsers：N

描述了一个具有两个成分的状态：一个文件序列和一个自然数。

24.3 应用数学符号描述形式规约

为了说明教学符号在软件部件的形式规约中的使用，我们再一次考虑24.1.3 节中提出的块处理器的例子。回顾一下，计算机操作系统中的一个重要部件维护用户创建的文件，这个部件的一部分就是块处理器。块处理器维护一个未用块池，并同时保持对当前被使用块的跟踪，当块从被删除文件释放时，它们通常被加入到等待进入未用块池的块队列中。这如图 24-2 所示，图中显示了一些部件：自由(未用)块池、被操作系统管理的文件使用的块、以及那些等待被加入到未用块池中的块。等待块被保持在队列中，队列中每个元素包含来自某被删除文件的一组块。描述块处理器的不变式有如下一组条件：

- 没有块同时被标记为未用和已用。
- 所有在队列中的块集合将是当前已用块集合的子集。
- 在未用块集合中没有重复的块号。
- 没有队列元素包含相同的块号。
- 已用块和未用块的集合将是组成文件的块的总集。
- 在已用块集合中没有重复的块号。

假定有集合 BLOCKS 是所有块号的集合,集合 AllBlocks 是在 1 和 MaxBlocks 间的块的集合。状态由两个集合和一个序列来模拟，两个集合分别是 used 和 free，均包含块，used 集合包含当前被文件使用的块，free 包含对新文件可用的块，序列将包含准备从已删除文件释放的块的集合。状态可以描述为：

used，free：P BLOCKS

BlockQueue：seqPBLOCKS

这和程序变量的声明非常相似,它说明 used 和 free 是块的集合,BlockQueue 将是一个序列，序列中的元素是块的集合。数据不变式可以描述为：

$$used \frown free = \varnothing /\backslash$$

$$used \cup free = AllBlocks \wedge$$

$$\forall i：\ dom\ BlockQusue.BlockQueusi \subseteq used \wedge$$

$$\forall i，\ j：\ dom\ BlockQueue.i \neq j \Rightarrow BlockQueue\ i \cap BlockQueue\ j = \varnothing$$

这个数据不变式的数学成分和以前描述的自然语言成分中的 4 条相匹配。数据不变式的第一行说明在块的 used 集合和 free 集合间不存在公共块；第二行说明 used 集合和 free 集合的和总是等于整个系统中全部的块的集合；第三行指出在块队列中的第 i 个元素总是 used 块的一个子集；最后一行说明如果块队列中的任意两个元素不相同，则在这两个元素间没有公共块。数据不变式的最后两条自然语言描述是通过 used 和 free 是集合、因此没有重复元素这一事实表现出来的。

我们将定义的第一个操作是从块队列头移走一个元素，前置条件是队列中必须有至少一个项：

$$\#BlockQueue > 0$$

后置条件是队列头必须被移走并放入自由块集合中，队列被调整以显示这一变化：

$$used' = used \setminus head\ BlockQueue\ \wedge$$

$$free' = free \blacksquare head\ BlockQueue \wedge$$

$$BlockQueue' = tail\ BlockQueue$$

在许多形式化方法中的一种习惯用法是以操作后的变量值为主，这样，上面给出的表达式的第一行指出：新的已用块(used')将等于旧的已用块减去已被移走的块；第二行指出：新的自由块(free')将等于旧的自由块加入块队列的头；第三行指出：新的块队列将等于旧的块队列的尾，即除了第一个元素外的所有元素的队列。我们描述的第二个操作是将一个块的集合 Ablocks 加入到块队列中，前置条件是 Ablocks 当前是已用块的集合：

$$Ablocks \blacksquare used$$

后置条件是块的集合被加入到块队列的尾部，同时已用块和自由块的集合均保持不变：

$$BlockQueue' = BlockQueue \frown \langle Ablocks \rangle \wedge$$

$$used' = used \wedge$$

free'=free

毫无疑问，块队列的数学规约比自然语言叙述或图形的模型要严格得多，这种严格需要更多的工作，但从一致性和完整性方面得到的好处可在很多类型的应用中得到证明。

24.4 形式化规约语言

形式化规约语言通常由三个主要的成分构成：(1)语法，定义用于表示规约的特定符号；(2)语义，帮助定义用于描述系统的“对象的全域(universe of objects[WIN90])”；(3)一组关系，定义确定出哪个对象真正满足规约的规则。

形式化规约语言的语法域通常基于从标准集合论符号和谓词演算导出的语法，例如，变量如 x，y，z 描述一组和问题(有时称为论域)相关的对象，并和在 24.2 中描述的运算符一起使用。虽然语法通常是符号化的，但如果图符(如方框、箭头和圆圈等图形符号)没有二义性，则也可以使用。

规约语言的语义域指出语言如何表示系统需求，例如，程序设计语言有一组形式化语义使得软件开发者可以刻划转换输入到输出的算法。形式化文法(如 BNF)可以用于描述程序设计语言的语法，然而，程序设计语言只能表示计算功能，所以并不是好的规约语言。规约语言必须有更广的语义域，即，规约语言的语义域必须能够表达这样的概念：“对在无限集 A 中的所有 x，在无限集 B 中有某 y，使得性质 P 对 x 和 y 成立”［WIN90］。其他规约语言应用可以描述系统行为，规约的语义，例如，可以设计一种语法和语义以刻划状态和状态变迁、事件和它们对状态变迁的影响、或者同步和定时。

用不同的语义抽象以不同的方式来描述同一个系统是可能的，我们在第 12 章和第 20 章已以欠形式的方式做到了这一点。数据流和对应的加工用数据流图描述，而系统行为用状态—变迁图来描述。类似的符号被用来描述面向对象的系统。不同的建模符号可以用来表示相同的系统，每种表示方法的语义提供了对系统的补充视角。为了说明当使用当形式化方法时的情形，假定一种形式规约语言被用于描述在系统中使得特定状态发生的事件的集合，另一种形式化关系描述出现在某给定状态中的所有功能，这两种关系的交叉则提供了对引起特定功能发生的事件的标识。

当前已有许多形式规约语言在使用——CSP[HIN95，HOR851，LARCH[GUT93]，膪 DM［JON91］和 Z[SPI88，SPI92]是充分展示以上特征的代表性形式规约语言。在本章中，Z 规约语言被用演示用途。Z 语言配备有一个称为“证明助手”的自动工具，它存储的引致对规约正确性的数学证明的公理、推导规则和面向应用的定理［WOO89］。

24.5 用 Z 表示一个软件构件的例子

Z 规约被构造为一组 schema——盒子式的结构，用于引入变量并刻划这些变量之间的关系。一个 schema 是实质上类似于程序设计语言中的子例程或过程的形式化规约，采用和过程或子例程被用于构造系统的相同方式，schema 被用于构造形式化规约。

在本节中，我们使用 Z 规约语言来建模在 24.1.3 节中引入并在 24.3 节进一步讨论的块处理器的例子。Z 语言符号的小结在表 24-2 中给出。下面 schema 的例子描述了块处理器的状态和数据不变式：

```
———————BlockHandler——————————

used， free：P BLOCKS

BlockQueue：seqPBLOCKS

used⌢free=■∧

used■free=AllBlocks∧

■i：dom BlockQueue.BlockQueue i■used∧

■i，j：dom BlockQueue.i≠j■

BlockQueue i ⌢BlockQueue j=■
```

该 schema 包含两个部分，中线上面的部分表示状态的变量，而中线下面的部分描述数据不变式。只要当表示不变式和状态的 schema 被用在另一个 schema 中，则以符号△为前缀，这样，如果上面的 schema 被用在描述某个操作的 schema 中，则它将被写作△BlockHandler。如上面语句所蕴含的那样，schema 可用于描述操作，下面 schema 的例子描述了从块队列中移走一个元素的操作：

```
———————RemoveBlock——————————

△ BlockHandler

#BlockQueue＞0

used'=used\head BlockQueue∧

free'=free■head BlockQueue∧

BlockQueue'=tail BlockQueue
```

上面例子中对△BlockHandler 的包含使得所有构成状态的变量对 RemoveBlock schema 是可用的，并且保证在操作的执行前后将保持数据不变式。

第二个操作，加一个块集合到队列的尾部，表示如下：

－－－－－－AddBlock ————————

△BlockHandler

Ablocks? ： P BLOCKS

Ablocks? ■used

BlockQueue'=BlockQueue⌢〈Ablocks? 〉

used'=used∧

free'=free

根据 Z 中的约定，读入的输入变量如果不形式状态的一部分，则以问号结尾，这样 Ablocks? 是作为一个输入参数，以问号结尾。

表 24-2 Z 符号体系小结

Z 的符号体系基于附类型集合论和一阶逻辑。Z 提供了一个结构，称为 schema，用来描述规约的状态空间和操作。一个 schema 将变量声明和限制变量的可能取值的一组谓词组合在一起。在 Z 中，schema X 的定义形式如下：

－x————————————

declarations

predicates

全局函数和常量的定义形式如下：

declarations

predicates

声明(dedaration)给出函数或常量的类型，而谓词(piedicates)给出它的值。本表中仅列出 Z 的符号的缩略集合：

sets：

S：P X　　S 被声明为 X 的集合

X ■S　　X 是 S 中的成员 x■S　x 不是 S 中的成员

S■T S 是 T 的子集：S 中每个元素均在 T 中

S■T S 和 T 的并：包含在 S 或 T 或二者中的每个元素

S⌒T S 和 T 的交：包含同时在 S 和 T 中的元素

S＼T S 和 T 的差：包含在 S 中的而不在 T 中的每个元素

■空集：不包含任何成员

{X} 单元素集：仅仅包含 X 的而不在 T 中的每个元素

N 自然数集合： 0，1， 2，…

S：F X S 被声明为 X 的有限集

max(S) 数的非空集合 S 中的最大者中的每个元素

Functions：

f：X■Y f 被声明为从 X 到 Y 的部分内射

dom f f 的定义域：f(x)有定义的值 x 的集合

ran f f 的值域： x 在 f 的定义域上变化而形成的 f(x)值集合 f■{x■y} 一个和 f 相同的函数，除了 x 被映射到 y(x)值集合 {X} ■f 一个和 f 相似的函数，除了 x 被从定义域中去除┳导 鋻 ogic：

P∧Q： P 与 Q 同时为真时才为真

P■Q P 蕴含 Q：或者 Q 为真或者 P 为假时为真

θS'=θS 在操作中 schema S 中没有成分改变

24.6 形式化方法的十条戒律

在现实世界中作出使用形式化方法的决策并不是一件简单的事，Bowen和Hinchley［BOW94］发明了“形式化方法的十条戒律”作为对那些希望应用这个重要的软件工程方法的人们的指南[①]。

1.你应该选择适当的符号体系。为了从众多的形式规约语言中进行有效的选择，软件工程师应该考虑语言词汇、被规约的应用类型以及语言的使用广度等。

2.你应该形式化，但不要过分形式化。通常没有必要对主要系统的每个方面应用形式化方法，那些安全关键的部件是首选对象，然后才是那些不允许出错(出于商业理由)的部件。

3.你应该估计成本。形式化方法的启动成本很高，人员的培训、支持工具的获取、以及合同顾问的雇用将产生很高的第一次成本，在检查和形式化方法相关联的投资回报时必须考虑这些成本。

4.你应该有随时可以请教的形式化方法顾问。当第一次使用形式化方法时，专家培训和进行中的咨询是成功的关键。

5.你不应该放弃传统的开发方法。集成形式化方法和传统的和/或面向对象的方法(第 12 章和第 20 章)是可能的，在很多情况下是所希望的。每个方法有各自的长处和弱点，如果合适地应用组合可能产生最好的结果[②]。

6.你应该建立详尽的文档。形式化方法对建立系统需求文档提供了简洁的、无二义的和一致的方法。然而，推荐使用自然语言注释配合形式规约作为增强读者对系统的理解的机制。

7.你不应该对你的质量标准作任何折衷。“对形式化方法没有任何不可思议的东西”［BOW94］，为此，在系统开发过程中其他的 SQA 活动(第 8 章)必须一如既往地贯彻实施。

8.你不应该教条化。软件工程师必须认识到形式化方法并不是正确性的保证，即使在开发中使用形式化方法，最终系统仍可能(有人说很可能)有小的遗漏、小的 bug、以及其他不满足期望的属性。

9.你应该测试、测试、再测试。软件测试的重要性已在第 16、17、和 22 章中讨论过。形式化方法并不能将软件工程师从对管理良好计划的、彻底的测试的需求中解脱出来。

10.你应该复用。长期来说，减少软件成本和增加软件质量的唯一合理的方法是复用(第 26 章)。形式化方法并没有改变这个现实，事实上，形式化方法是一种合适的创建可复用构件库的途径。

24.7 形式化方法——未来之路

虽然基于形式的、数学的规约技术还没有在产业界广泛应用，但它确实比欠形式化方法有实质性的优点。Liskov 和 Bersins［LIS86］有如下总结：泛应用，但它确实比欠形式化方法有形式化规约可以用数学方法研究，而非形式化方法则不能。例如，一个正确的程序可以被证明满足其规约，或两个规约集可以被证明是等价的，…。某些型式的不完整性和不一致性可以被自动地检测。

此外，形式规约消除了二义性并在软件工程过程的早期阶段鼓励使用更严格的方法。

但是，问题仍然存在，形式化规约主要关注于功能和数据，而问题的时序、控制和行为等方面却更难于表示。此外，有些问题元素(如，人机界面)最好用图形技术来刻划。最后，使用形式化方法来建立规约比本书中提出的其他分析方法更难于学习，并且对某些软件实践者来说它代表了一种重要的“文化冲击”。为此，很有可能形式化的数学规约技术将形成未来一代CASE工具的基础，到那时，基于数学的规约可能会被软件工程界更广泛的采用。[①]

24.8 小结

形式化方法提供了规约环境的基础，它使得所生成的分析模型比用传统的或面向对象的方法生成的模型更完整、一致和无二义。集合论和逻辑符号的描述设施使得软件工程师能创建清晰的关于事实(需求)的陈述。

支配形式化方法的基本概念是：(1)数据不变式——一个条件表达式，它在包含一组数据的系统的执行过程中总保持为真；(2)状态——系统访问和修改的存储数据；(3)操作——系统中发生的动作，以及对状态数据的读或写。每一个操作是和两个条件相关联的：前置条件和后置条件。

离散数学——与集合和构造性规约、集合运算符、逻辑运算符、以及序列相关联的符号体系——形成了形式化方法的基础。这些数学在形式化规约语言，如Z 语言中被实现。

Z 和其他所有的形式规约语言一样，有语法和语义域。语法域使用与集合和谓词演算的符号体系紧密联系的符号体系，语义域使得语言能够以简洁的方式表达需求。Z 的结构使用 schema——盒子式的结构，来引入变量并刻划变量间的关系。

使用形式化方法的决策必须考虑启动成本以及与根本上不同的技术相关的文化变更。在大多数情况下，形式化方法对安全关键和业务关键的系统具有最高的回报。

思考题

24.1 回顾一下 24.1.1 节中欠形式化方法对软件工程的不足点，根据你自己的经验，对每点不足给出三个例子。

24.2 数学作为规约机制的益处已在本章进行了详细讨论，那么，有坏处吗？

24.3 你参加了一个开发 fax modem 软件的项目组，你的任务是开发应用的“phone book”部分。phone book 功能使得可以存储多达 MaxNames 个地址名，

并连带相关的公司名、fax 号码、以及其他相关信息。使用自然语言定义：a.数据不变式；b.状态；c.可能的操作。

24.4 你参加了一个开发 MemoryDoubler 软件的项目组，该软件为 PC 机提供比物理内存更大的形式内存。这个功能是通过标识、收集和重分配已经被分配给某现存应用但未被使用的内存块而完成的。未用块被重分配给需要附加内存的应用。做适当的假设并用自然语言定义：a.数据不变式；b.状态；c.可能的操作。

24.5 给出某集合的构造性规约，该集合包含形为(X, y, Z^2)的自然数三元组，其中x和y之和等于z。

24.6 基于 PC 的应用的安装程序首先确定是否存在可接受的硬件和系统资源集合，它检查硬件配置以确定是否存在各种设备(很多可能设备中的)，并且确定是否已安装了系统软件和驱动程序的某特定版本。什么集合运算符可被用于完成该功能？给出一个例子。

24.7 为下面的陈述用逻辑和集合运算符写出一个表达式：“对所有 x 和 y，如果 x 是 y 的父辈且 y 是 z 的父辈，则 x 是 z 的祖父辈。每个人都有父辈”。[提示：用函数 P(x，y)和 G(x，z)分别来表示父辈和祖父辈函数。]

24.8 为一个对的集合给出构造性集合规约，这里每个对的第一个元素是两个非零自然数的和，而第二个元素是相同的两个数的差。两个数应该在 100 和 200 之间。

24.9 对问题 24.3 给出状态和数据不变式的数学描述。在 Z 规约语言中精化该描述。

24.10 对问题 24.4 给出状态和数据不变式的数学描述。在 Z 规约语言中精化该描述。

24.11 使用表 24.2 中的 Z 符号，选择本书中描述过的 SafeHome 保安系统的某部件，尝试用 Z 去刻划它。

24.12 使用本章参考文献或“推荐阅读文献及其他信息源”中的一个或多个信息源，开发一个半小时的关于形式规约语言(非 Z 语言)的基本语法和语义的演讲稿。

推荐阅读文献及其他信息源

除了本章中作为参考文献使用的书目外，过去几年中已出版了大量的关于形式化方法的书籍。下面是一些有价值的书籍列表：

Barden，R.，S.Stepney，and D.Cooper，Z in Practice，Prentice-Hall，1994.

Bowen，J.，Formal Specification and Documentation and Documentation using Z：A Case Study Approach，International Thomson Computer Press，1996.

Cooper，D.，and R.Barden， Z in Practice， Prentice-Hall，1995.

Craigen，D.，S.Gerhart，and T.Ralston，Industrial Application of Formal Methods to Model，Design and Analyze Computer Aystems，Noyes Data Corp.，Park Ridge，NJ，1995.

Diller，A.，Z：An introduction to Fromal Methods，2nd edition，Wiley，1994.

Hinchey，M.，and J.Bowen，Applications of Fromal Methods，Prentice-Hall，1996.

Lano，J.， and H.Haughton(eds.)，Object-Oriented Specification Case Studies，Prentice-Hall，1993.

Rann，D.，J.Turner，and J.Whitworth，Z：A Beginner's Guide，Chapman &Hall，1994.

Ratcliff， B.，Introducing SpecificationUsing Z：A Practical Case Study Approach，McGraw-Hill，1994.

D.Sheppard，An Introduction to Formal Specification With Z and VDM，McGraw-Hill，1995.

IEEE Transaction on Software Engineering，IEEE Software，以及 IEEE Computer 的 1990 年 9 月期均是关于形式化方法的优秀的有价值的信息源。

Schuman 编辑了一部关于形式化方法和对象技术书(Formal Object-Oriented Development)，该书为形式化方法的选择使用以及如何结合 OO 方法进行使用提供了指南。

大量有价值的形式化方法信息，包括指向更多的书目、技术报告、规约语言、工具和其他有用信息的指针可在 NASA 的形式化方法网址和形式化方法虚拟库中找到：

http：//atb-www.larc.nasa.gov/cgi-bin/fm.cgi

http：//www.comlab.ox.ac.uk/archive/formal-methods/

更多的讨论可在新闻组 comp.specification.misc 和 comp.specification，z 中找到。

关于 VDM(另一种形式化语言)的综合信息可在下面地址得到：

gopher：//nisp.ncl.ac.uk/11/lists-u-z/vdm-forum/files

关于形式化方法的一般信息，以及研究文章和 web 指针可在以下地址得到：

http：//www.research.att.com/orgs/ssr/areas.html

形式化方法的最新的WWW文献列表可在http：//www.rspa.com找到。

① 这一节和本章第一部分的其他内容引自DarrelIncel编写的、欧洲版的、第三版的《软件工程》——实践者的研究途径一书。

① 在这点上请千万小心，本章中出现的数字系统说明不是与积分一样的简单，软件系统没那么复杂的，希望用一行数学公式来说明它是概本不现实的。

① 请回顾一下第 12 章和第 20 章中提到的术语“状态”，它代表的是系统或对象的行为。

① 这的处理是一个［OW94］的超级缩略版。关于形式化方法的详细文字或其他有用信息可通过以下网址获得http：//www.cl.cam.ac.uk/users/mgh

② 净室软件工程（见第 25 章）是将形式化方法和更贯用的开发方法相结合使用的一个实例。

① 其他不同观点的看法也是非常值得注意的，参见［YOU94］。

第 25 章　净室软件工程

传统的软件工程建模、形式化方法、程序验证(正确性证明)、以及统计 SQA 的集成使用已经组合成一种可以导致极高质量软件的技术。净室软件工程(Cleanroom software engineering)是一种在软件开发过程中强调在软件中建立正确性的需要的方法。代替传统的分析、设计、编码、测试和调试周期，净室方法建议一种不同的观点［LIN94］：

在净室软件工程后面的哲学是：通过在第一次正确地书写代码增量并在测试前验证它们的正确性来避免对成本很高的错误消除过程的依赖。它的过程模型是在代码增量积聚到系统的过程的同时进行代码增量的统计质量验证。

净室方法在很多方面将软件工程提升到另一个层次。象第 24 章中讨论的形式化方法技术一样，净室过程强调在规约和设计上的严格性，以及使用基于数学的正确性证明来对结果设计模型的每个元素进行形式化验证。作为对形式化方法中采用的方法的扩展，净室方法还强调统计质量控制技术，包括基于客户对软件的预期的使用的测试。

当现实世界中软件失败时，则充满了立即的和长期的危险。这些危险可能和人的安全、经济损失、或业务和社会基础设施的有效运作相关。净室软件工程是一个过程模型，它在可能产生严重的危险前消除错误。

25.1 净室方法

在硬件构造技术方面“净室”的哲学是相当简单的：建立一种排除产品缺陷引入的价格有效和时间有效的构造方法。不是先制作一个产品，然后再去消除缺陷，净室方法要求在规约和设计中消除错误，然后以“净”的方式制作。

对软件工程的净室哲学首先由 Mill 和其同事于 1980 年代提出［MIL87］，虽然对这个严格的软件开发方法的早期经验显示了很大的希望［HAU94］，但它并没有得到广泛的使用，个严格的 Henderson 为此总结了三个理由：

1.一种信念认为净室方法学太理论、太数学、以及太激进，以至难于在真实的软件开发中使用。

2.它提倡开发者不需要进行单元测试，而是进行正确性验证和统计质量控制，这些概念代表了和当前大多数软件开发方式的很大的背离。

3.软件开发产业的成熟度。净室过程的使用需要在整个生命周期阶段定义的过程的严格的应用，因为大多数软件企业的运作还处于 ad hoc 级别(由 SEI 的 CMM 定义)，因此，还没有准备好应用那些技术。

虽然在上面提到的每个顾虑中都有真实的成分，但是，净室软件工程的潜在益处远远超出要克服这些顾虑的核心——文化阻力所需要的投入。

25.1.1 净室策略

净室方法使用增量软件模型(第 2 章)的一个专门版本。一个“软件增量的流水线”［LIN94］被若干小的、独立的软件工程小组开发，一旦每个增量被认证通过，它将被集成为一个整体。因此，系统的功能随时间增加。

对每个增量的净室任务序列在图 25-1 中给出。用在第 10 章讨论的系统工程方法开发整个系统或产品需求，一旦功能已被分配给系统的软件元素，净室增量的流水线被初始化，发生下列任务：

增量计划。开发一个采用增量策略的项目计划，建立每个增量的功能、它的项目大小、以及净室开发进度表。必须特别小心以保证通过认证的增量将被定时集成。

需求收集。使用类似于在第 11 章引入的技术，为每个增量开发一个客户级需求的更详细的描述。

盒结构规约。使用一个运用盒结构的规约方法［HEV93］来描述功能规约。遵从在第 11 章讨论的操作分析原则，盒结构“在每一个精化级别上分离和分开行为、数据及过程的创造性定义”。

形式化设计。使用盒结构方法，净室设计是规约的自然的无缝的扩展。虽然，在两个活动间可进行清楚的区分，但是，规约(称为“黑盒”)是被递进地求精(在

一个增量内)以成为类似于体系结构的和过程的设计(分别称为“状态盒”和“清晰盒”)。

正确性验证。净室小组对设计及代码进行一系列严格的正确性验证活动。验证(25.3 节和 25.4 节)从最高层次的盒结构(规约)开始，然后移向设计细节和代码。正确性验证的第一层次通过应用一组“正确性问题”［LIN88］来进行，如果这没有证明规约是正确的，则使用更形式化的(数过学的)验证方法。

代码生成、检查和验证。以某种专门语言表示的盒结构规约被转换为合适的程序设计语言。然后，使用标准的走查或检查技术(第 8 章)来保证代码和盒结构的语义相符性，以及代码的语法正确性。然后，对源代码进行正确性验证。

统计性测试计划。分析软件的项目级使用情况，计划和设计一组执行用途的“概率分布”的测试用例(25.4 节)。如图 25－1 所示，这个净室活动是和规约、验证及代码生成并行进行的。

统计性使用测试。记住，对计算机软件进行彻底测试是不可能的(第 16 章)，因此，总需要设计有限数量的测试用例。统计性使用技术［P0088］执行一系列由特定对象的所有用户的所有可能的程序执行的统计样本(上面提到的概率分布)所导出的测试。

认证。一旦完成验证、检查和使用测试(并且所有错误被修正)，则开始进行增量集成前的认证工作。

和本书中讨论的其他软件过程模型一样，净室过程很强地依赖于对生产高质量的分析和设计模型的需求。就象我们在本章后面将看到的那样，盒结构符号体系仅仅是软件工程师表示需求和设计的另一种方式。净室方法的真正特性是对工程模型：运用形式化验证。

25.1.2 什么使得净室独特？

Dyer［DYE92］在他定义过程时，间接提到了净室方法的不同之处：

净室代表了将软件开发过程置于统计质量控制之下，并结合良好定义的持续过程改善策略的第一次实际的尝试。为了达到这个目标，设计了一个独特的净室生命周期，它着重于为了正确地进行软件设计的基于数学的软件工程，以及为了软件可靠性认证的基于统计的软件。

净室软件工程和本书第三和第四部分讨论的传统的和面向对象的观点不同，因为：

1.它显式地使用统计质量控制。

2.它使用基于数学的正确性证明来验证设计规约。

3.它很强地依赖于统计性用法测试来揭示高影响的错误。

显然，净室方法应用了大多数，即使不是全部，在本书中提到的基本的软件工程原理和概念。要产生高质量的结果，好的分析和设计过程是很重要的。但是，净室软件工程和传统软件实践的差别在于：它不再强调(有人称，删除)单元测试和调试的作用，大量地减少(或删除)由软件开发者所做的测试工作量。[①]

在传统软件开发中，错误是作为事实而接受的，因为错误被认为是不可避免的，每个程序模块必须进行单元测试(以揭示错误)和调试(以消除错误)。当软件最终发布时，实际使用还会揭示更多的缺陷，因此，开始又一个测试和调试的周期。和这些活动相关的再工作既耗钱又耗时，更糟糕的是，它可能是退化的。——错误修正可能(不经意地)导致更多的引入错误。

在净室软件工程中，单元测试和调试被正确性验证和基于统计的测试所替代。这些活动，配合保持对持续改善所必需的信息的纪录，使得净室方法与众不同。

25.2 功能规约

不管选择哪种分析方法，在第 11 章提出的操作原理总是适用的。数据、功能和行为被建模，划分(求精)结果模型以提供增加的、更多的细节。整体目标是从捕获问题实质的规约转移到提供实质实现细节的规约。

净室软件工程通过使用称为盒结构规约的方法来遵从操作分析原则。一个“盒”在某个细节层次封装系统(或系统的某些方面)。通过逐步求精的过程，盒被精化为层次，其中每个盒具有引用透明性——“每个盒规约的信息内容对定义其精化是足够的，不需要依赖于任何其他盒的实现”［LIN94］。这使得分析员能够层次地划分一个系统，从在顶层的本质表示转移向在底层的实现特定的细节。有三种盒类型：

黑盒。这种盒刻划系统或系统的某部分的行为。通过运用由激发得到反应的一组变迁规则，系统(或部分)对特定的激发(事件)作出反应。

状态盒。这种盒以类似于对象的方式封装状态数据和服务(操作)。[①]在这个规约视图中，表示出状态盒的输入(激发)和输出(反应)。状态盒也表示黑盒的“激发历史”，即，封装在状态盒中的、必须在蕴含的变迁间保留的数据。

清晰盒。在清晰盒中定义状态盒所蕴含的变迁功能，简单地说，清晰盒包含了对状态盒的过程设计。

图 25-2 给出了使用盒结构规约的求精方法。黑盒(BB_1)定义了对一个完整激发集合的反应，BB_1可以被精化成一组黑盒。BB1.1 到BB1.n的每一个都关注一类行为。精化过程继续到直至标识出行为的一个内聚类(例，$BB_{1.}1.1$)。然后对黑盒BB1.1.1 定义状态盒(SB1.1.1)，SB1.1.1 包含了实现BB1.1.1 定义的行为所需

的所有数据和服务。最后，$SB_1.1.1$ 被精化为一组清晰盒(CB1.1.1 到CB1.1.1n)，并且刻划出过程的设计细节。

当进行这些精化的每一步时，也同时进行正确性验证。验证状态盒规约以保证每个规约均同其父辈黑盒规约定义的行为相一致，类似地，针对其父辈状态盒进行清晰盒规约的验证。

应该注意，基于形式化方法的规约方法(第 24 章)可以用来代替盒结构规约方法，唯一的要求是每一层次的规约可以被形式化地验证。

25.2.1 黑盒规约

黑盒规约是一个抽象，用图 25-3 所示的符号 [MIL88] 描述了系统如何对激发作出反应。函　齠被应用到输入(激发)S 的序列 S*，并变换它们为输出(反应)R。对简单的软件部件而言，f 可以是一个数学函数，但一般情况下，f 被用自然语言(或形式规约语言)描述。

为面向对象系统引入的很多概念(第 19 章)也适用于黑盒。黑盒封装数据抽象和操纵那些抽象的操作，和类层次一样，黑盒规约可以展示使用层次，其中，低层盒从树结构中的那些高层盒继承属性。

25.2.2 状态盒规约

状态盒是“状态机制的一个简单通用化”［MIL88］。一个状态是系统行为的某些可观测的模式(回忆 12 章对行为建模和状态变迁图的讨论)。当进行处理时，一个系统对事件(激发)作出反应从当前状态转变到某新状态。当转变进行时，可能发生某个动作。状态盒使用数据抽象来确定到下一个状态的转变及作为转变的后果而发生的动作(反应)。

如图 25-4 所示，状态盒同黑盒协作。作为黑盒的输入的激发 S，来自某外部源及一组内部系统状态 T。Mills 给出了包含在状态盒内的黑盒的函数 f 的数学描述：

$$g：S^* \times T^* \rightarrow R \times T$$

这里 g 是和特定状态 t 连接的子函数。当整体地考虑时，状态—子函数对(t，g)定义了黑盒函数 f。

25.2.3 清晰盒规约

清晰盒规约是和过程设计及结构化编程紧密关联的。在本质上，状态盒中的子函数 g 被实现 g 的结构化编程结构所替代。

作为一个例子，以图 25-5 所示的清晰盒为例。在图 25-4 中的黑盒 g 被一个带有条件的顺序结构所替代，随着逐步求精的进程，这些又可以进一步精化为低层的清晰盒。

重要的是要注意：在清晰盒层次中所描述的过程性规约可被证明是正确的，这一主题将在下一节中讨论。

25.3 设计求精和验证的优点

在净室软件工程中使用的设计方法大量使用结构化编程哲学(第 14 章)，但是，在此情形下，结构化编程要以严格的多地方式来使用。

基本的处理函数(描述在规约的早期求精中)被精化，其方法是“逐步扩充数学函数为逻辑连接词(如，if-then-else)和子函数构成的结构，扩充不断进行直至所有标识出来的子函数可以被在用于实现的程序设计语言中直接陈述”［DYE92］。

结构化程序设计方法可以被有效地用于函数的精化，但是，对数据设计又怎样呢？这里，可以使用一组基本的设计概念(第 13 章)，程序数据被封装为一组，由子函数提供服务的抽象。数据封装、信息隐蔽和数据类型等概念被用于创建数据设计。

25.3.1 设计求精和验证

每个清晰盒规约代表了一个完成某状态盒变迁所需的过程(子函数)的设计。对清晰盒规约，如图 25-6 所示使用结构化编程构造和逐步求精。一个程序函数f被精化为一系列子函数g和h[①]的顺序结构，这些又被进一步精化为条件结构(if-then-else和do-while)。进一步的求精例举出连续的逻辑细节。在每个求精层次，净室小组[②]执行一次形式化正确性验证。为此，一组类属的正确性条件被附加到结构化编程构造上，如果函数f被扩充为g和h(如图 25-6 所示)，则对f的所有输入的正确性条件是：

• g 结合 h 能完成 f 的功能吗？

如果一个函数 p 被精化为形为 if＜c＞then q else r 的条件形式，则对 p 的所有输入的正确性条件是：

• 一旦当条件＜c＞为真，q 能完成 p 的功能吗？一旦当条件＜c＞为假，r 能完成 p 的功能吗？

如果一个函数 m 被精化为循环，则对 m 的所有输入的正确性条件是：

• 终止能够保证吗？

• 一旦当＜C＞为真，n 结合 m 能完成 m 的功能吗？一旦当＜c＞为假，退出循环仍能完成 m 的功能吗？

每次当一个清晰盒被精化为下一个细节层次时，都应用上面给出的正确性条件。

值得注意的是：结构化编程构造的使用限制了必须进行的正确性测试的数量。对顺序结构只检查单个条件，对 if-then-else 结构只测试两个条件，对循环则只验证三个条件。

为了说明对过程设计的正确性验证，我们使用一个简单的由 Linger 和其同事给出的例子［LIN79］，意图是设计并验证一个小的程序，该程序对某给定的整数 x，找出其平方根的整数部分 y。图 25-7 用流程图给出了其过程设计。

为了验证该设计的正确性，我们必须定义如图 25-8 中所示的入口和出口条件，入口条件说明 x 必须大于或等于 0，出口条件需要 x 保持不变而 y 在图中指定的值域内取值。为了证明设计是正确的，必须证明图 25-8 中显示的条件 init、loop、cont、yes 和 exit 在所有情形下都是正确的，这有时被称为子证明。

1.条件init要求［x≥0 and y=0］。基于问题的需求，假定入口条件是正确的，[①]因此，init 条件的第一部分x≥0 是满足的。在流程图中，在init条件前的语句设置y=0，因此， init条件的第二部分也是满足的，因此，init为真。

2.条件 loop 可能以下为两种方式之一遇到：(1)直接从 init(在此情形下，loop 条件被直接满足)，或(2)通过穿过条件 cont 的控制流。因为 cont 条件和 loop 条件相同，因此，不管导向它的流的路径如何，loop 为真。

3.条件cont仅仅在y的值被递增 1 后遇到，此外，导向cont的控制流路径只在yes条件也为真时被调用，因此，如果$(y+1)^2 \leqslant x$，一定有$y^2 \leqslant x$，cont条件被满足。

4.条件 yes 在如图所示的条件逻辑中被测试，因此，当控制流沿着所示的路径移动时，yes 条件一定为真。

5.条件exit首先要求x保持不变，对设计进行检查可发现x没有出现在赋值操作的左边，没有函数那样调用x，因此，x将不变。因为条件测试$(y+1)^2 \leqslant x$必须失败时才可能达到exit条件，这意味着$(y+1)^2 > x$。此外，loop条件还必须保持为真(即，$y^2 \leqslant x$)，因此，$(y+1)^2 > x$和$y^2+ \leqslant x$可以组合在一起满足exit条件。

我们必须进一步保证循环可终止，对循环条件的检查指明，因为 y 是递增的，而 x≥0，因此，最终循环必定终止。

上面的 5 步是时在图 25-7 所示的算法的设计的正确性证明，我们现在可以肯定，这个设计将计算出平方根的整数部分。

更严格的针对设计验证的数学方法是可能的，然而，这个话题的讨论超出了本书范围，有兴趣的读者可参考［LIN79］。

25.3.2 设计验证的优点[①]

对清晰盒设计的每一步精化的严格的正确性验证有一系列显著的优点，Linger［LIN94］描述这些优点如下：

• 它将验证缩成为一个有限的过程。在清晰盒中，以嵌套的、顺序的方式组织控制结构，这就自然地定义了一个层次，该层次揭示了必须被验证的正确性条件。替代公理［LIN79］使得我们可以用在子证明层次的函数的控制结构精化来替换指定函数。例如，在图 25-9 中的指定函数 f1 的子证明需要证明：操作 g1 和 g2 与指定函数 f2 的组合对数据具有和 f1 相同的效果。注意，f2 替代了在证明中它的精化的所有细节。这一替代使得证明论据局部于当前的控制结构，事实上，它使得软件工程师可以以任意顺序执行证明。过分强调将验证缩成到有限的过程而对质量产生的正面效果是不可能的，即使所有程序，除了那些最微不足道的程序，展示了本质上无限的执行路径，它们也可以在有限步骤内完成验证。

	Subproofs：
[f1]	f1=[DO g1; g2; [f2]END]?
DO	
g1	
g2	
[f2]	f2=[WHILE p1 DO [f3] END]?
WHILE	
p1	
DO [f3]	f3=[DO g3; [f4]; g8 END]? 3]END]?
g3	
[f4]	f4=[IF p2; THEN[f5] ELSE[f6] END]?
IF	
p2	
THEN [f5]	f5=[DO g4; g5 END]?

```
    g4

    g5

    ELSE [f6]              f6=[DO g6; g7 END]?
THEN[f5]    ELSE[f6]   END]?

    g6

    g7

    END

    g8

    END

    END
```

图 25-9 带有子证明的设计 [LIN94]

• 它使得净室小组验证设计和代码的每一行。小组可以在正确性定理的基础上通过小组分析和讨论来执行验证，并且当在关键使命系统中需要额外的信心时他们可以生成书面的证明。

• 它达到几乎零缺陷的水平。在小组的复审过程中，每个控制结构的每个正确性条件被顺序验证。每个小组成员必须就每个条件都是正确的达成共识，这样只有小组的每个成员均未能正确地验证某条件时，才有可能出现错误。基于个体验证的无异议的全体认同的需求使得生产的软件在其第一次执行前几乎没有或根本没有任何缺陷。

• 它具有可伸缩性。每个软件系统，不管有多大，均具有顶层的由顺序、选择和迭代结构构成的清晰盒过程，典型每个过程均调用一个有数千行代码的大的子系统，并且每个子系统也具有自己的顶层的指定的函数和过程。所以，这些高层控制结构的正确性条件的验证采用与那些低层结构验证相同的方法。高层的验证可能需要更多的时间(这是值得的)，但它不需要更多的理论。

• 它生产出比采用单元测试方法更好的代码。单元测试仅仅检查从很多可能的路径中选出的测试路径的执行结果。基于函数验证的理论，净室方法可以验证所有数据的每个可能的结果，因为虽然一个程序可能有很多执行路径，但它只有一个函数。验证也比单元测试有效的多，大多数验证条件可以在几分钟内被检查，但单元测试要花费大量时间去准备、执行和检查。

应该注意，设计验证最终必须应用到源代码本身，此时，它通常称为正确性验证。

25.4 净室测试

净室测试的策略和战术在本质上不同于传统测试方法。传统测试方法导出一组测试用例来揭示设计和编码错误，而净室测试的目的是：通过证明使用实例(第19章)的统计性样本已经被成功地执行来验证软件需求。

25.4.1 统计的使用测试

计算机程序的用户很少需要去了解设计的技术细节，程序的用户可见的行为是被通常由用户产生的输入和事件所驱动的。但是，在复杂系统中，输入和事件的可能的范围(即 use cases)可能是非常宽的。什么是可以充分地验证程序行为的使用实例的子集？这是统计的使用测试所关注的第一个问题。

统计的使用测试“等同于以用户试图使用软件的方式来测试软件”［LIN94］。为了完成测试工作，净室测试小组[①]必须确定被测软件的使用概率分布。通过分析软件的每个增量的规约(黑盒)，定义一组导致软件改变其行为的激励(输入或事件)，再通过和潜在用户的交流、使用场景的建立以及对应用领域的总体性了解，为每个激发赋上一个使用概率。

按照使用概率分布为每个激励[②]生成测试用例。为了举例说明，考虑本书中前面讨论过的SafeHome安全系统。正在使用净室软件工程开发一个软件增量，它管理和安全系统键区的用户交互。对这个增量，已经标识出如表 25-1 的 5 个激发，通过分析，给出了每个激发的百分概率。为了更简单地选择测试用例，这些概率被映射到 1 至 99 的数字区间［LIN94］。

表 25-1 程序的激励及其概率分布

程序激励	分布	区间
Arm	disarm(AD)	50％
Zoneset(ZS)	15％	50-63
Query(Q)	15％	64-78
Test(T)	15％	79-94
Panicalarm(PA)	5％	95-99

为了生成符合使用概率分布的使用测试用例序列，在 1 至 99 间生成一系列的随机数，随机数对应于概率分布的区间(表 25-1)。因此，使用用例序列被随机定义，但又对应于适当的激励出现的概率。例如，假定生成了下面的随机数序列：

13-94-22-24-45-56

81-19-31-69-45-9

38-21-52-84-86-97

基于表 25.1 中显示的分布区间选择适当的激励，从而导出下面的 use cases：

AD-T-AD-AD-AD-ZS

T-AD-AD-Q-AD-AD

AD-AD-ZS-T-T-PA

测试小组执行上面的(以及其他的)use cases，针对系统规约来验证软件行为。记录下来测试的时间，使得可以确定间隔时间。利用间隔时间，认证小组可以计算出平均失效时间 MTTF。如果一个长的测试序列被无失败地完成，则 MTTF 低，且软件的可靠性较高。

25.4.2 认证

本章前面讨论的验证和测试技术导致了可以被认证的软件构件(和完整的增量)，在净室软件工程方法中，认证意味着可以刻划每个构件的可靠性(用平均失效时间 MTTF 来测度)。

可认证的软件构件的潜在影响远远超出了单个净室项目的范围，可复用的软件构件可以和它们的使用场景、程序激励、以及概率分布一起存储，每个构件在描述的使用场景和测试体系下有认证过的可靠性，这些信息对那些希望使用这些构件的人来说是极其极贵的。

认证方法涉及以下 5 个步骤［WOH94］：使用这些构件的人来说是极其极贵的。

1.必须创建使用场景。

2.刻划使用轮廓。

3.从轮廓中生成测试用例。

4.执行测试，记录并分析失败数据。

5.计算并认证可靠性。

步骤 1 到步骤 4 已在前面几节中讨论过，本节中，我们集中于可靠性认证。

净室软件工程的认证需要创建三个模型［P0093］：

取样模型。软件测试执行 m 个随机测试用例，如果没有错误发生或只有少于指定数量的错误发生，则被认证通过。值 m 被用数学的方式导出以保证能够达到需要的可靠性。

构件模型。对一个由 n 个构件组成的系统进行认证，构件模型使得分析员能够确定构件 i 在完成前将失败的概率。

认证模型。系统的整体可靠性被规划和认证。

在统计的使用测试完成后，认证小组已得到了交付软件所需要的信息，包括认证后的 MTTF，这是使用上面的每个模型计算出来的。

关于取样的计算、构件和认证模型的详细讨论已超出本书讨论的内容，有兴趣的读者可参考文献［MUS87］、[CUR86]和[P0093]。

25.5 小结

净室软件工程是软件开发的一种形式化方法，它可以生成有非常高的质量的软件。它使用盒结构规约(或形式化方法)进行分析和设计建模，并且强调将正确性验证，而不是测试，作为发现和消除错误的主要机制。使用统计的使用测试来获取认证被交付的软件的可靠性所必需的出错率信息。

净室方法从使用盒结构表示的分析和设计模型入手，一个“盒”在某特定的抽象层次上封装系统(或系统的某些方面)。黑盒用于表达系统的对外可观测行为，状态盒封装状态数据和操作，清晰盒用于对某状态盒中的数据和操作所蕴含的过程设计进行建模。

一旦完成了盒结构设计，则运用正确性验证。软件构件的过程设计被划分为一系列子函数，为了证明每个子函数的正确性，要为每个子函数定义出口条件并实施一组子证明。如果每个出口条件均被满足，则设计一定是正确的。

一旦完成了正确性验证，便开始统计的使用测试。和传统测试不同，净室软件工程并不强调单元或集成测试，而是通过定义一组使用场景、确定对每个场景的使用概率及定义符合概率的随机测试来进行软件测试。将产生的错误记录和取样、构件和认证模型相结合使得可以数学地计算软件构件的可靠性。

净室哲学是一种严格的软件工程方法，它是一种强调正确性的数学验证和软件可靠性的认证的软件过程模型，其目标和结果是非常低的出错率，这是使用欠形式化方法难于或不可能达到的。

参考文献

[CUR86] Curritt, P.A., M.Dyer, and H.D.Mills, “Certifying the Reliability of Software, ” IEEE Trans, Software Engineering, vol.SE-12, no.1, January 1994.

[DYE92] Dyer, M., The Cleanroom Approach to Quality Software Development, Wiley, 1992

[HAU94] Hausler,P.A.,R.Linger,and C.Trammel, “Adopting Cleanroom SoftwareEngineering with a Phased Approach,” IBM Systems Journal,vol.33, no.1, January1994, pp.89-109.

[HEN95] Henderson, J., “Why Isn’t Cleanroom the Universal Software developmentMethodology? ” Crosstalk, vol.8, no.5, May 1995, pp.11-14.

[HEV93] Hevner, A.R., and H.D. Mills, “Box Structure Methods for System DevelopmentWith Objects, ” IBM Systems Journal, vol.31, no.2, February 1993, pp.232-251.

[LIN79] Linger, R.M., H.D.Mills, and B.I.Witt, Structured Programming: Theory andPractice, Addison-Wesley, 1979.

[LIN88] Linger, R.M., and H.D.Mills, “A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility, ” Proc.COMPSAC’88, Chicago, October1988.

[LIN94] Linger, R., “Cleanroom Process Model, ” IEEE Software, March 1994, pp.50-58.

[MIL87] Mills, H.D., M.Dyer, and R.Linger, “Cleanroom Software Engineering, ” IEEESoftware, September 1987, pp.19-24.

[MIL88] Mills, H.D., “Stepwise Refinement and Verification in Box Structured Systems, ” IEEE Computer, June 1988, pp.23-35

[MUS87] Musa, J.D., A.Iannino, and K.Okumoto, Engineering and Managing Software withReliability Measures, McGraw-Hill, 1987.

[POO88] Poors, J.H., and H.D.Mills, “Bringing Software Under Statistical Quality Control,” Quality Progress,November 1988,pp.52-55.

[POO93] Poors, J.H., H.D.Mills, and D.Mutchler, “Planning and Certifying SoftwareSystem Reliability, ” IEEE Software, vol.10, no.1, January1993, pp.88-99.

[WOH94] Wohlin, C., and P.Runeson, “Certification of Software Components,”IEEE Trans, Software Engineering, vol.20, no.6, June 1994, pp.494-499.

思考题

25.1 如果你必须选出净室软件工程和传统的或面向对象的方法有着根本不同的一个方面，那么这个方面是什么？

25.2 增量过程模型和认证工作如何一起生产出高质量软件？

25.3 使用盒结构规约，开发 SafeHome 系统的“first-pass”分析和设计模型？

25.4 对思考题 12.13 中引入的 PHTRS 系统的一部分开发其盒结构规约。

25.5 对思考题 21.15 中给出的 e-mail 系统，开发其盒结构规约。

25.6 一个冒泡排序算法定义如下：

```
procedure bubblesort;

vari, t: integer;

begin

repeat until t=a[1]

t: =a[1];

for j: =2 to n do

if a[j-1]} a[j]then begin

t: =a[j-1];

a[j-1]: =a[j];

a[j]: =t;

end

endrep

end
```

划分该设计为子函数，并定义一组条件，使得你能够证明该算法是正确的。

25.7 就思考题 25.6 中讨论的冒泡排序的正确性验证证明做一份记录。

25.8 选择你在其他情形下设计(或导师布置的)的某程序构件并给出其完整的正确性证明。

25.9 选择某你经常使用的程序(例如，e-mail 处理器、字处理器、电子表格程序)，为这些程序创建一组使用场景，定义对每个场景的使用概率，然后开发出类似表 25-1 所示的程序的激励和概率分布表。

25.10 对思考题 25.9 中开发的程序激励和概率分布表，使用一个随机数生成器来开发一组用于统计的使用测试中的测试用例。

25.11 用你自己的话，描述净室软件工程中认证的意图。

25.12 写一篇短文，描述用于定义在 25.4.2 节中简略介绍的认证模型的数学，以参考文献[MUS87]、[CUR86]和[POO93]作为起点。

推荐阅读文献及其他信息源

关于净室软件工程唯一可用的书是 Dyer[DYE92]。Cleanroom Pamphlet(SoftwareTechnology Support Center，Hill AF Base， UT，April 1995)包含了一系列重要文章。Linger[LIN94]是对此主题的较好的导论之一。软件工程技术资产源 ASSET(美国国防部)中提供了 6 卷优秀的 Cleanroom Engineering Handbooks，ASSET 可以通过 info@source.asset.com 联系。

净室软件工程公司的 Michael Deck 准备了一组关于净室话题的参考书目，罗列如下：

一般的和导论性的：

Cobb，R.H.，and H.D.Mills，“Engineering Software under Statistical Quality Control，”IEEESoftware，November 1990，pp.44-54.

Coffee，P.，“Can Mass Testing Fix Windows’Quality Woes？”PC Week，September 19，1994，p.28.Responses in PC Week，October 10，1994，pp.73，78.

Deck，M.D.，“Cleanroom Software Engineering：Quality Improvement and Cost Reduction，”Proc.Pacific Northwest Software Quality Conference，October 1994，pp.243-258.

Hevner，A.R.，S.A.Becker，and L.B.Pedowitz，“Integrated CASE for Cleanroom Development，”IEEE Software，March 1992，pp.69-76

Keuffel,W.,“Clean Your Room:Formal Methods for the’90s,”Computer Language, July1992, pp.39-46.

Linger, Richard C., “Cleanroom Software Engineering for Zero-Defect Software, ”Proc.15thIntl.Conf.on Software Engineering, May 1993.

Lokan, C.J., “The Cleanroom Process for Software Devlopment, ”The Australian ComputerJournal, vol.25, no.4, November 1993.

管理实践：

Deck,M.D.,and P.A.Hausler,“Cleanroom Software Engineering:Theory and Practice, ”Proc.Software Engineering and Knowledge Engineering90, June 1990, pp.71-77.

Linger, R.C., and Spangler, R.A., “The IBM Cleanroom Software Engineering TechnologyTransfer Program, ”Sixth SEI Conf.on Software Engineering Education, San Diego, CA, October1992.

规约、设计和复审：

Deck, M.D.,“Using Box Structures to Link Cleanroom and Object-Oriented SoftwareEngineering, ”Technical Report 94.01b, Cleanroom Software Engineering, Inc., Boulder, CO, 1994.

Dyer,M.,“Designing Software for Provable Correctness:The Direction for Quality Software, ”Information and Software Technology, vol.30, no.6, July/August 1988, pp.331-340.

测试和认证：

Dyer, M., “An Approach to Software Reliability Measurement,”Information and SoftwareTechnology,vol.29,no.8,October 1987, pp.415-420.

Whittaker, J.A., and Thomason, M.G., “A Markov Chain Model for Statistical SoftwareTesting, ”IEEE Trans.on Software Engineering, October 1994, pp.812-824.

实例研究和经验报告：

Green, S.E., A.Kouchakdjian, and V.R.Basili, “Evaluation of the Cleanroom Methodology inthe Software Engineering Laboratory,” Proc.14th Annual Software Engineering Workshop, NASAGoddard Space Flight Center, November 1989.

Hausler, P.A., “A Recent Cleanroom Success Story: The Redwing Project, ”Proc.17th AnnualSoftware Engineering Workshop, NASA Goddard Space Flight Center, December 1992.

Head,G.E., “Six-Sigma Software Using Cleanroom Software Engineering Techniques, ”Hewlett-Packard Journal, June 1994, pp.40-50.

Tann,L-G., “OS32 and Cleanroom”Proc.1st Annual European Industrial Symposium onCleanroom Software Engineering, Copenhagen, Denmark, 1993, pp.1-40.

Trammel, C.J., Binder, L.H., and Snyder, C.E., “The Automated Production ControlDocumentation System: A Case Study in Cleanroom Software Engineering, ”ACM Trans.onSoftware Engineering and Methodology, vol.1, no.1, January 1992, pp.81-94.

Wohlin, C., “Engineering Reliable Software, ”Proc.4th Intl.Symposium on Software ReliabilityEngineering, November 1993, pp.36-44.

在Internet新闻组comp.software-eng上偶尔也有对净室软件工程的讨论，详细的净室技术讨论,包括可提供的课程描述,可在IBM净室软件技术中心找到：

http：//www.clearlake.ibm.com/MFG/solutions/cstc.html

净室软件工程公司准备了参考文献的FTP站点：

http：//www.csn.net/～deckm/

对净室方法已建立了一个电子邮件表和讨论组，信息可在如下站点得到：

http：//www.quality.org/qc/lists/sw-cleanroom.info.txt

最新的关于净室软件工程的WWW文献源可在http：//www.rspa.com找到。

① 导出了测试，但是是通过某个独立的测试流导出的。

① 参见本书的第4部分。

① 基本的结构化编程的结构参见第14章。

② 由于验证过程中与整个系统有关，所以在导出自身的验证时很少会出错。

① 注意在此平方根取负值是毫无意义。

① 该节和图25-7到25-9引自［LIN94］，已获引用权。

① 也称为“校验小组”。

② 可用自动工具完成之。详情参阅［DYE92］。

第26章　软件复用

作为人类，当我们开始以理性的方式认识我们的世界时候，我们就知道了复用的益处。我们复用几乎每样东西——概念、对象、论据、抽象、过程——它们构成了日常的生活。一句老话“重新发明轮子(reinventing the wheel)”概括了这一切。重新发明已经发明了的东西在经济上、智力上或实用上均无意义。重新发明轮子已经折磨了软件界几乎半个世纪。

Peter Freeman[FRE87]给出了如下关于复用的讨论：

复用是活动，而不是对象。它是这样一种常见的活动，以至通常大多数字典都没有收录它，一个假设前提是：聪明的读者将理解“复用”意味着“再次使用某东西”。我们已熟悉“再生(recycling)”等词语以及常见的格言：使某东西“完成双重任务(do double duty)”。

在创建软件相关的系统的语境中，复用仅仅是非常简单的任何过程，该过程通过复用来自以前开发工作的某些东西来生产(或帮助生产)一个系统。那么，唯一的问题是：复用什么、什么是导致成功复用的过程。

在软件工程的范围内，复用既是旧概念，也是新概念。程序员从最早的计算时代已开始复用概念、对象、论据、抽象和过程，但是我们复用的途径是特定的。今天，必须在非常短的时间内建立复杂的、高质量的基于计算机的系统，这要求一种更有组织的复用方法。

但是，一系列问题随之出现，有可能通过组装一组可复用的构件而构造复杂系统吗？能否以成本及时间合算的方式完成这项工作？能否建立适当的激励机制鼓励软件工程师复用，而不是重发明？管理将招致对可复用软件构件的创建增加费用吗？实现复用所必需的构件库可以以并使得那些需要它的人可以访问它的方式来创建吗？现存的构件能否被那些需要它的人找到？

这些和很多其他问题还在继续困扰研究者和产业专业人员，他们正在努力使软件构件复用成为主流的软件工程方法。我们在本章中可以找到对某些问题的回答。

26.1 管理问题[①]

对软件复用的可能益处的快速审查揭示：在软件产品的开发中复用带来的不仅仅是简单的费用节省，而是更多。例如，对已知是可靠的软件构件的复用比在每个新的应用中重设计和重编码同一个构件有更少的风险。如果可以将注意力集中于优化一组可复用构件，而不是必须不断地重优化已存在的模块的新版本，效率问题也可以被更有效地处理。

然而，复用的明显益处经常由于组织的障碍、对软件复用缺乏本质的理解以及较少或没有鼓励和实现复用技术的策略等因素而被抵消。在下面几节中，我们将考虑这些话题。

26.1.1 复用的障碍

对软件可复用性不断增加的关注意味着什么这一问题仍然存在混乱，虽然很多产业观察员承认某些现存的软件开发实践将必须改变，但是，他们似乎并不知道哪些特定的方面需要改变。对软件复用还存在一系列障碍，为了开发一个有效的复用策略，管理者(以及技术人员)必须知道这

些障碍是什么。

1.很少有公司和组织具有和全面的软件复用计划有一点点相似之处的任何东西。虽然很多公司和组织有某些人在某些地方正在“研究”该概念,但很少有谁试图在不远的未来去实现这样一个计划。软件复用对很多软件公司来说均不是“最高优先级的”。

2.虽然越来越多的软件商正在销售对软件复用提供直接辅助的工具或构件,但大多数软件开发者并不使用它们。帮助进行软件复用的软件产品包括：提供对基础设施支持的(例如，可复用构件库，浏览器)，帮助创建可复用构件的工具，以及完整的软件复用系统——专门为辅助和鼓励软件复用而设计的工具和软件构件。

3.几乎没有相应的培训以帮助软件工程师和管理者理解和应用复用。不仅只有很少针对复用的培训，而且这方面的话题在大多数软件工程培训中仅被概略地提到。

4.很多软件实践者仍然相信复用“相对于其价值来说带来更多麻烦”。经常我们会听到技术人员罗列一个长长的关于“复用软件的缺点”，管理者似乎也相对地愿意维持现状。即使当管理者购买了对软件复用必需的工具和培训，反对该概念的职员仍然很多。

5.很多公司继续鼓励对复用无促进的软件开发方法学(例如，功能分解)，而不鼓励那些可能对复用有促进的方法学(例如，面向对象方法)。

6.很少有公司对生产可复用的软件构件有激励措施。事实上，还存在阻碍。当前项目的客户对用于开发可复用构件有额外投资费用是犹豫的，作为结果，项目管理者推动用尽可能特定的程序构件来完成指定工作。

除了必须强调的技术问题，很多其他相关问题对复用也有影响。政治的、管理的、法律的、文化的、财政的、市场的、以及产品化的等方面的问题也必须考虑。

26.1.2 硬件类比

在计算机硬件产业，复用是规范的。例如，有标准的 CPU 芯片，标准的 RAM、数学协处理器和 ROM 芯片,这些芯片和其他集成电路被用于广范的应用中。很长时间以前，电子工程师便发现：最重要的可复用性公理之一是通用性。

和软件工程师不一样,电子工程师没有必要去验证某特殊 CPU 支持的每一个“操作码”在某特定应用中的执行，他们也没有必要去证明 RAM 的每个最后字节均被使用。电子工程师通过数据得到芯片可靠性的量化指示。

如何将上面的类比应用到可复用软件？考虑对一个可复用软件构件的设计，为了真正可复用，软件构件必须在某些不是其当前特定应用时(或应用的某部分)的其他地方可用。为了增加此事发生的机会，软件工程师必须将构件做得更通用化。事实上，存在众所周知将软件构件的通性提高到非常高的层次的方法。

当然，存在着权衡。增加通用性可能减少效率、或甚至增加复杂性，这些和其他考虑必须同

有利因素进行权衡，有利因素有：由于使用已验证过的构件而带来的潜在增加的可靠性，由于复用而带来的成本和时间上的节省等。

在过去几年中，关于软件复用已有很多成果。虽然关于特定问题的争论仍在继续，但是以下观点已被普遍接受：

• 不仅仅是源代码可以被复用。事实上，设计、计划、测试数据和文档(在软件配置中的其他项中)可以被复用。代码复用只带来最小的生产率和可靠性收益，更大的收益是通过复用与“可复用源代码”相关联的设计和文档而达到的。因此，可复用性并不是仅仅限于编码的概念。

• 研究似乎指出、而且实践似乎也显示很多软件产品的大部分可以从可复用构件组装得到。

• 存在大量和软件复用相关联的度量。它们的范围从测度在某应用中复用的源代码和文档的纯粹大小的度量到测度软件可复用性的影响的度量(例如，软件生产率度量)。

软件复用技术实际上是一组事物的集合，这些事物包括：可复用构件、分类学、复用支持系统、组装概念、方法学及其他。该技术和软件产品的大小的结合多于和复用发生规模的结合。

软件设计方法(第14和21章)确实在软件复用中扮演了重要的角色。某些方法可以促进复用，而有些方法阻碍复用。我们很早就已知道，软件设计应该构造化以便于修改。我们仅仅现在才开始认识到：在软件过程的第一阶段就引入软件可复用化论题，我们可以大大地增加软件复用的作用效果。最后，通过研究在某特定应用领域中的很多系统的设计，可以分离并发展候选的复用构件。

26.1.3 建立复用途径的一些建议

下面步骤将帮助组织进行必要的变革以充分地使用软件复用的概念：

1. 建立内部的软件复用计划。这样一个计划可以帮助组织控制软件的质量和成本。

2. 要求将软件复用作为任何技术和管理培训的一个不可缺的部分。对面向对象的培训尤其应该如此。

3. 按照内部的软件复用计划，寻求对软件复用有最积极贡献的工具和库。

4. 鼓励采用已被证明为可以促进软件复用的方法和工具。

5. 跟踪并测度软件复用以及软件复用的影响。政策决策应该基于“实际数据”，而不是臆测。

6. 管理上必须显示它积极地鼓励软件的复用。

7. 认识到不仅仅是“模块”可以被复用，工具、测试数据、设计、计划、环境、以及其他软件项均可被复用。

8. 最重要的是，认识到软件复用不是“平常的业务”，对软件复用的采用将需要某些变革，

应该以有效的方式引入这些变革。记住，软件复用的概念是被大多数技术和管理人员所不容的。

26.2 复用过程

复用应该是每个软件过程的一个不可缺少的部分，在第 2 章中，“构件组装模型”(图 26-1)被用于举例说明如何将一个可复用构件库集成到典型的演化过程模型中。在本节后部，我们将

探讨某过程模型，该模型强调那些为了使复用发生而必需出现的活动。但是首先，必须理解当软件开发进行时被复用的“软件制品”。

26.2.1 可复用的软件制品

我们已经知道到软件复用不仅仅涉及源代码，但是，还涉及多少东西呢？CaperJones[JON94]定义了可作为复用候选的十种软件制品：

项目计划。软件项目计划(见本书第二部分)的基本结构和许多内容(例如，SQA 计划)均是可以跨项目复用的。这样减少了用于制定计划的时间，也减低了和建立进度表、风险分析和其他特征相关的不确定性。

成本估计。因为经常不同项目中含有类似的功能，所以有可能在极少修改或不修改的情况下，复用对该功能的成本估计(第 5 章)。

体系结构。即使当考虑不同的应用领域时，也很少有截然不同的程序和数据体系结构。因此，有可能创建一组类属的体系结构模板(例如，事务处理体系结构)，并将那些模板作为可复用的设计框架。

需求模型和规约。类和对象的模型和规约(第 20 章)是明显的复用的候选者，此外，用传统软件工程方法(第 12 章)开发的分析模型(例如，数据流图)也是可复用的。

设计。用传统方法(第 14 章)开发的体系结构、数据、接口和过程化设计是复用的候选者，更常见的是，系统和对象设计(第 21 章)是可复用的。

源代码。验证过的程序构件(用兼容的程序设计语言书写的)是复用的候选者。

用户和技术文档。即使特定的应用是不同的，也经常有可能复用用户和技术文档的大部分。

用户界面。可能是最广泛被复用的软件制品，GUI 软件经常被复用。因为它可占到一个应用的 60%的代码量，因此，复用的效果非常显著。

数据。在大多数经常被复用的软件制品中，数据包括：内部表、列表和记录结构，以及文件和完整的数据库。

测试用例。一旦设计或代码构件将被复用，相关的测试用例应该“附属于”它们。

表 26-1[JON94]给出了一些数据(来自军事及系统项目)，这些数据显示了 1 美元投资在 4 年后的回报(针对上面列出的每个软件制品)。Jones 描述了整体影响[JON94]：“复用所有 10 种软件制品的总计值可以产生可能是任意已知的软件技术中的最好回报”。

表 26-1 软件复用在 4 年后的回报价值[JON94]

可复用软件制品	4 年回报	可复用软件制品	4 年回报
项目计划	$2.00	源代码	$6.00
成本估计	$3.00	用户和技术文档	$1.50
体系结构	$1.50	用户界面	$1.00
需求模型和规约	$3.00	数据	$3.50
设计	$5.00	测试用例	$3.50

应该注意，复用可以扩展到上面所讨论的可交付的软件制品之外，它也包含了软件工程过程中的元素。特定的分析建模方法、检查技术、测试用例设计技术、质量保证过程、以及很多其他软件工程实践可以被“复用”，例如，如果某软件项目组有效地应用净室软件工程方法(第 25 章)，该方法可能适用于另一个项目。为了作出决定，有必要定义一组描述功能(26.3.2 节)，它们使得潜在的净室用户能够对其应用作出适当的决策。

26.2.2 一个过程模型

已经提出了一系列针对复用的过程模型，每个均强调并行的轨迹，在轨迹中领域工程(26.3 节)和软件工程同时进行。领域工程执行一系列工作，以建立一组可以被软件工程师复用的软件制品，然后，这些软件制品被传越过分隔领域工程和软件工程的“边界”。

图 26-2 给出了一个典型的显然适用于复用过程模型[CHR95]。领域工程创建应用领域的模型，该模型被用作在软件工程流中分析用户需求的基础。软件体系结构(相应的结构点，见 26.3.3 节)为应用的设计提供了输入。最后，在可复用构件已被构造好后(作为领域工程的一部分)，它们可以在软件构造活动中被软件工程师所使用。

26.3 领域工程

领域工程的目的是标识、构造、分类和传播一组软件制品，它们对某特定应用领域中对现存的和未来的软件系统具有很好适用性。其整体目标是建立相应的机制，以使得软件工程师在工作于新的或现存的系统时可以分享这些软件制品——复用它们。

领域工程包括三个主要的活动——分析、构造和传播。在第 20 章中已给出领域分析的概述，然而，本节中将再讨论这个话题。领域构造和传播将在本章以后几节中讨论。

有人争辩说“复用将消失，不是被消除，而是被集成”进软件工程实践之中[TRA95]，随着

复用被更多的强调，人们相信在下个十年领域工程将变得和软件工程一样重要。

26.3.1 领域分析过程

在第 20 章，我们讨论了在面向对象软件工程范围内领域分析的方法，过程中的步骤定义如下：

1.定义将被研究的领域。

2.分类从领域中抽出的物项。

3.收集领域中有代表性的应用样本。

4.分析样本中的每个应用。

5.开发对象的分析模型。

必须注意，领域分析适用于任意软件工程范型，并且可以用于传统的以及面向对象的软件开发。

Prieto－Diaz[PRI87]扩展了上面给出的领域分析的第二个步骤，建议了一个 8 步骤的标识和分类可复用软件制品的方法：

1.选择特定的功能/对象。

2.抽象功能/对象。

3.定义分类法。

4.标识公共特征。

5.标识特定的关系。

6.抽象关系。

7.导出功能模型。

8.定义领域语言。

领域语言使得在领域中进行应用的规约及构造成为可能。

虽然上面的步骤提供了一个有用的领域分析模型,但是它没有提供帮助决定哪些软件制品是复用候选的有用的指南。Hutchinson 和 Hindley[HUT88]提出了下面一组实际的问题，它们可以用作标识可复用软件构件的指南：

●构件功能对未来的实现工作是需要的吗？

●在领域中构件功能的公共性怎样？

●在领域中存在构件功能的重复吗？

●构件是否依赖于硬件？

●在不同实现之间硬件是否保持不变？

●硬件细节可被移动到另一个构件吗？

●设计为下面的实现进行过足够的优化吗？

●我们能够将一个不可复用的构件参数化以使其变成可复用的吗？

●构件是否可以仅仅经过少许修改就能够在很多实现中复用吗？

●通过修改进行复用是可行的吗？

●某不可复用的构件能够通过被分解以产生一组可复用构件吗？

●针对复用的构件分解是有效的吗？

关于领域分析的深入讨论不在本书范围之内，更多的信息可见[PRI93]。

26.3.2 领域特征

要确定一个潜在可复用的软件制品在某特殊情况下是否真地可被使用，有时是一件困难的事。为了能够确定这样的事情，有必要定义一组领域特征，它们被领域中所有的软件所分享。领域特征定义了存在于领域中的所有产品的某种类属属性，例如，类属特征可能包括：安全/可靠性的重要性、程序设计语言、处理中的并发性、以及很多其他内容。

某可复用软件制品的领域特征的集合可以表示为 $\{D_p\}$，其中集合中每一项 D_{pi} 表示某特定的领域特征。赋给 D_{pi} 的值表示一个顺序的等级，它指明了该特征对软件制品p的相关性，一组典型的等级[BAS94]可能是：

1.与复用是否合适没有相关性。

2.仅仅在不寻常的情况下相关。

3.相关，软件制品可以被修改以使得其可以被复用，而不管其间的差别。

4.明显地相关，并且如果新软件不具有这个特征，复用将是低效率的，但复用仍然是可能的。

5.明显地相关，并且如果新软件不具有这个特征，复用将是无效的，此时不推荐复用。

当新软件w将在某应用领域内被构造，可为它导出一组领域特征，然后，在D_{pi}和D_{wi}间进行比较，以决定是否现存的软件制品p可以被有效地在应用w中复用。

表 26-2[BAS94]列出了典型的可能对软件复用有影响的领域特征，为了有效的复用软件制品，必须考虑这些领域特征。

表 26－2 影响复用的领域特征[BAS94]

产品	过程	人员
需求稳定性	过程模型	动机
并发软件	过程符合性	教育
内存限制	项目环境	经验/培训
应用大小	进度限制	▪应用领域
用户界面复杂性	预算限制	▪过程
程序设计语言	生产率	▪平台
安全/可靠性		▪语言
寿命需求		开发队伍
产品质量		生产率
产品可靠性		

即使将被开发的软件显然属于某应用领域,在领域中的可复用软件制品也必须被分析以确定它们的可用性。在某些情况下(希望不要太多)，“重新发明轮子”可能仍然是最为成本合算的选择。

26.3.3 结构建模和结构点

当使用领域分析时，分析员寻找在某领域中应用间的重复模式。结构化建模(Structuralmodeling)是一种基于模式的领域工程方法，应用该方法的前提假设是：每个应用领域有重复的模式(功能的、数据的和行为的)，它们具有可复用的潜在可能。

Pollak 和 Rissman[POL94]描述结构建模如下：

结构模型由少量的用于表明清晰的交互模式的结构元素组成。使用结构模型的系统的体系结构通过多个由这些模型元素组成的东西来刻划，这样，在系统的体系结构单元间的复杂交互可以用在这些少量元素间的简单交互模式来描述。

每个应用领域可用一个结构模型来刻划(例如，飞行器电子设备在细节上差异很大，但是在该领域的所有现代软件具有相同的结构模型)，因此，结构模型是一种体系结构制品，它可以也应该在领域内所有应用中被复用。

McMahon[MCM95]描述结构点(structure Point)为：“在结构模型中的一个独特构成物”，结构点有三个显著的特征：

1.一个结构点是一个抽象，它应该有有限数量的实例。用面向对象的行话(第 19 章)来陈述，类层次的规模应是小的。此外，该抽象应该在领域中所有应用中不断重现，否则，用于验证、文档化和传播结构点所需的努力不可能是成本合算的。

2.管理结构点的使用的规则应该是容易理解的。此外，结构点的接口应该相当简单。

3.结构点应该通过隐藏所有包含在结构点内部的复杂性而实现信息隐蔽,这会减少整个系统可被感知的复杂性。

作为把结构点当作系统的体系结构模式的一个例子，考虑警报系统的软件领域，该领域可能包含如SafeHome[①]这样简单的系统，或如工业过程警报系统这样复杂的系统，然而，在每种情形，均可以遇到一组可以预测的结构模式：

●界面，使用户能够和系统交互。

●范围设置机制，允许用户设置将被测度的参数的范围。

●传感器管理机制，和所有的监控传感器通讯。

●反应机制，对传感器管理系统提供的输入作出反应。

●控制机制，使用户能够可以控制监控执行的方式。

这些结构点中的每一个被集成到一个领域体系结构中。

定义跨越一组不同的应用领域的类属的结构点[STA94]是可能的：

应用前端。GUI，包括所有菜单、面板、输入和命令编辑设施。

数据库。所有和应用领域相关的对象的仓库。

计算引擎。操作数据的数值和非数值模型。

报告设施。产生所有种类的输出的功能。

应用编辑器。根据用户特定需要定制应用的机制。

结构点已被建议作为在软件成本估计中代码行和功能点的替代物[MCM95],在 26.6.2 节中给出使用结构点进行成本估计的概要讨论。

26.4 建造可复用构件

关于创建可复用的软件并没有任何神奇之处。抽象、隐蔽、功能独立性、求精、以及结构化程序设计等设计概念，连同面向对象方法、测试、SQA、以及正确性验证方法——所有均对可复用[②]软件构件的创建有贡献。本节中，我们将不重复讨论这些话题，而是考虑特定于复用的问题，它们是对完整的软件工程实践的补充。

26.4.1 为了复用的分析和设计

本书第三、第四部分对分析模型的构件已有详细讨论。数据、功能和行为模型(用一系列不同符号表示)可以被创建以描述特定应用必须完成的任务，书面的规约被用于描述这些模型并生成完整的需求描述。

理想地，分析分析模型以确定模型中的那些指向现存的可复用软件制品的元素。问题是以能够导致“规约匹配”的形式从需求模型中抽取信息。Bellinzoni 及其同事[BEL95]描述了一种针对面向对象系统的方法：

构件被在不同的抽象层次定义和存储为规约、设计和实现类——每个类是来自以前应用的某产品的工程化描述。规约知识——开发知识——被以复用建议(reuse-suggestion)类的形式存储，它们包括对以构件的描述为基础检索可复用构件及检索后组装和剪裁构件的指导。

使用自动化工具浏览构件库，以试图匹配当前规约中所标记的需求和那些为现存可复用构件(类)描述的需求。领域特征(26.3.2 节)和关键词被用于发现潜在的可复用构件。

如果规约匹配生成符合当前应用需要的构件，设计者可从可复用构件库中提取这些构件并将它们用于新系统的设计中。如果没能找到设计构件，软件工程师必须应用传统的或面向对象的设计方法去创建它们。正是在这点——当设计者开始创建新的构件时——应该考虑为了复用的设计(DFR)。

我们已经提到过，为了复用的设计需要软件工程师应用已有的设计概念和原则(第 13 章)，但是，也必须考虑应用领域的特征。Binder[BIN93]建议了作为为了复用的设计的基础而应该考虑[①]的一系列关键问题：

标准数据。应该研究应用领域，并标识出标准的全局数据结构(例如，文件结构或完整的数据库)，那么所有设计构件可以使用这些标准数据结构来刻划。

标准接口协议。应该建立三个层次的接口协议：模块内接口的本质、外部的技术(非人)接口的设计、以及人机界面。

程序模板。结构模型(26.3.3 节)可以作为新程序的体系结构设计的模板。

一旦已经建立了标准数据、接口和程序模板，则设计者有了一个可在其中创建设计的框架。符合这个框架的新的构件对以后的复用有更高的概率。

26.4.2 构造方法

和设计一样，可复用软件制品的构造依赖于本书中其他地方已经讨论过的软件工程方法，构造可以使用传统的第三代语言、第四代语言和代码生成器、可视化程序设计技术、或更高级的工具来完成。

更高级的构造技术的一个有代表性的例子是Netron公司[BAS94b]开发的Frame技术，Netron方法定义了一组自适应的、称作 Frame 的类属构件。和对象一样，Frame 封装数据和操作，但是，它扩展了对象的定义，它结合进了使软件工程师能够通过选择、删除、修改或重复那些构成该 Frame 的任意子构件而对 Frame 进行适应性修改的机制。

可以通过组装来自 Frame 层次的构件而构造应用。在该层次底部的 Frame 类似于在因子化体系结构(第 13 章)中的工作者(worker)模块，即，它们包含执行低层系统功能(例如，操作系统交互、接口构造、数据库交互)的操作和数据结构；在 Frame 层次中层的 Frame 着重于和特定信息系统领域相关的功能(例如，事务处理、银行业务、客户服务)；在层次的顶部是“规约 fream”，它的作用是作为“系统的主要蓝图和开发者创建来定义系统的唯一的 Frame”[YOU94]。

26.4.3 基于构件的开发

当复用占据了一个应用开发的主导地位时，则构造方法有时被称为基于构件的开发或构件软件。如我们已经看到的，领域工程提供了基于构件的开发所需要的可复用构件库。这些可复用构件中的某些是内部开发的，其他的可以从现存应用中抽取的，还有一些可以从第三方获取。

但是，我们如何创建一个具有一致结构的构件库——它可以被一系列不同的内部和外部源查询并且还可以被集成进在某应用领域内的任意系统？答案是对这样的构件的标准的采用。4 个“体系结构成分”[ADL95]将被用于实现基于构件的开发：

数据交换模型。应该对所有的可复用构件定义使得用户和应用间能够交互和传递数据的机制(例如，拖和放、剪切和粘贴)。数据交换机制不仅应允许人和软件、构件和构件间的数据传递，而且也应使得能够在系统资源间进行数据传递(例如，将一个文件拖到某打印机图符上以实现输出)。

自动化。应实现一系列工具、宏和脚本为可复用构件间的交互服务。

结构化存储。包含在“复合文档”中的异质数据(例如，图形数据、声音、文本和数值数据)应该被作为单独的数据结构来组织和访问，而不是作为一组分开的文件。“结构化数据维持了嵌套结构的一个描述性索引，使得应用可以自由地进行导航浏览以定位、创建或编辑个体数据内容，就象终端用户直接操作一样”[ADL95]。

底层对象模型。对象模型保证在不同平台上用不同程序设计语言开发的构件可以互操作，即，对象必须能够跨网络进行通信。为了达到这个目标，对象模型定义了构件互操作标准，该标准是语言独立的，并使用接口定义语言(IDL)来定义。因为复用对软件产业的潜在影响非常巨大，一些主要的公司和产业联盟[①]已经提出了对构件软件的建议标准：

OpenDoc。主要技术公司(包括 IBM、Apple、和 Novell)的一个联盟已经提出了复合文档和构件软件的标准 OPenDoc。该标准定义了为使得由某开发者提供的构件能够和另一个开发者提供的

构件相互操作而必须实现的服务、控制基础设施、和体系结构。

OMG/CORBA。对象管理组织发布了公共对象请求代理体系结构(OMG/CORBA)。一个对象请求代理(ORB)提供了一系列服务，它们使得可复用构件(对象)可以和其他构件通信，而不管它们在系统中的位置。当用OMG/CORBA标准建立构件时，那些构件在某系统内的集成(没有修改)可以得到保证。[②]

使用客户/服务器的比喻，在客户端应用中的对象向 ORB 服务器请求一个或多个服务，请求是通过 IDL 或在运行时动态地进行的。一个接口池包含了所有关于服务请求和回答格式的信息。CORBA 将在第 28 章进一步讨论。

OLE2.0。微软开发了一个构件对象模型(COM)，它提供了对在单个应用中使用不同厂商生产的对象的规约。对象连接和嵌入(OLE)是 COM 的一部分，定义了可复用构件的标准结构。OLE 已成为微软操作系统(Windows 95、Windows NT)的一部分。

这些标准中哪一个将在产业中占据支配地位？这在当前是不容易回答的问题。当前 OLE 被用得更广一些(由于基于 Windows 应用的广泛使用)，但是，很多 OMG/CORBA 工具和开发环境正在被引入。对进一步的关于构件标准和支持工具的讨论见参考文献[ADL95]和[LIN95]。

26.5 分类和检索构件

考虑一个大的大学图书馆，成千上万的书籍、期刊和其他信息资源是可用的。但是为了访问这些资源，必须有合适的分类模式。为了在这些大量的信息中导航浏览，图书馆管理者定义了一种分类模式，它包括分类码、关键词、作者名、以及其他索引条目，所有这些使得用户可以快速和方便地找到所需资源。

现在考虑一个大的构件仓库，其中存放了成千上万的可复用构件。但是，软件工程师如何找到他所需要的构件呢？为了回答这个问题，又出现了另一个问题：我们如何以无二义的、可分类的术语来描述软件构件？这些是困难的问题，至今还没有确定性的答案。在本节中，我们探讨当前的研究方向，这将使得未来的软件工程师可以导航浏览复用库。

26.5.1 描述可复用构件

可以用很多方式来描述可复用软件构件，但是理想的描述包括 TracZ[TRA90]提出的 3C 模型——概念(concePt)、内容(content)和语境(context)。

软件构件的概念是“对构件做什么的描述”[WHI95]。构件的接口被完整地描述，而且语义——表示在前置条件后置条件的语境中——被标识。概念将传达构件的意图。

构件的内容描述概念如何被实现。在本质上，内容是对一般用户隐蔽的信息，只有那些企图修改该构件的人才需要了解的信息。

语境将可复用软件构件放置到其应用的领域中。即，通过刻划概念的、操作的和实现的特征，

语境使得软件工程师能够发现适当的构件以满足应用需求。

为了可用在实际环境中，概念、内容和语境必须被转换为具体的规约模式。已有很多的文章涉及可复用构件的分类模式。[①]所提出的方法可以分为三大类：图书馆和信息科学方法、人工智能方法、以及超文本系统。目前为止，绝大部分研究工作建议使用图书馆科学方法进行构件分类。

图 26—3 给出了源于图书馆科学索引方法的一个分类法，“受限的索引词汇(Controlledindexing vocabularies)”限制了可以用来分类对象(构件)的术语和语法，“不受限的索引词汇(Uncontrolled indexing Vocabularies)”则对描述的性质没有限制。大多数软件构件分类模式可归为如下的三类：

枚举分类(Enumerated Classification)。通过定义一个层次结构来描述构件，在该层次中定义软件构件的类以及不同层次的子类。实际的构件被罗列在枚举层次的任何路径的最低层，例如，对窗口操作[①]的枚举层次可能是：

window operations

display

open

menu-based

OpenWindow

system—based

sysWindow

close

via pointer

…

resize

viacommand

setWindowSize, stdResize, shrinkwindow

viadrag

pullwindow, stretchWindow

up/downshuffle

…

move

…

close

…

枚举分类模式的层次结构使得它易于理解和使用，然而，在建立层次之前，必须进行领域工程以获得在层次中适当的项的足够的信息。

刻面分类(Faceted Classification)。分析领域，并标识出一组基本的描述特征，这些特征，称为刻面，被根据其重要性区分优先次序并被联系到构件。刻面可以描述构件执行的功能、被操作的数据、构件应用的语境或任意其他特征。描述构件的刻面的集合称为刻面描述子，通常，刻面描述被限定不超过 7 或 8 个刻面。

作为一个简单的在构件分类中使用刻面的例子，考虑使用下列构件描述子的模式[LIA93]：

{function，object type，system tyPe}

刻面描述子中的每个刻面可含有一个或多个值，这些值一般是描述性关键词，例如，如果功能(function)是某构件的刻面，赋给此刻面的典型值可能是：

function=(copy，from)or(copy，replace，all)

多个刻面值的使用使得原函数 copy 能够被更完全地精化。

关键词(值)被赋给复用库中的每个构件的刻面集，当软件工程师在设计中希望查询构件库以发现可能的构件时，规定一列值，然后到库中寻找匹配项。可使用自动工具以完成同义词词典功能，这使得查找不仅包括软件工程师给出的关键词，还包括这些关键词的技术同义词。

刻面分类模式给领域工程师在刻划构件的复杂描述子时更大的灵活性[FRA94]，因为可以很容易地加入新的刻面值，因此，刻面分类模式比枚举分类方法易于扩展和进行适应性修改。

属性—值分类(Attribute－Value Classification)。为领域中的所有构件定义一组属性，然后值被以和刻面分类方法非常相似的方式赋给这些属性，事实上，属性-值分类方法和刻面分类方法是类似的，除了下面几点不同：(1)对可使用的属性数量没有限制，(2)属性没有优先级，(3)不使用同义词词典功能。

基于对上面分类技术的实验研究，Frank 和 Pole[FRA94]指出没有明显“最好”的技术和“没有某种方法比别的方法在查找效果上更适度…”。对复用库有效的分类模式的开发仍有许多

工作要做。

26.5.2 复用环境

软件构件复用必须有环境的支撑，环境应包含如下元素：

●用于存储构件和对检索构件必需的分类信息的构件库。

●提供对构件库访问的库管理系统。

●允许客户应用从构件库服务器中检索构件和服务的软件构件检索系统(如，对象请求代理)。

●将复用的构件集成到新设计或实现中的 CASE 工具。

每一个功能在复用库的范围内交互或者被包含在复用库中。

复用库是一个更大型 CASE 仓库(第 29 章)的一个元素，并且为一系列可复用软件制品(例如，规约、设计、代码、测试用例、用户指南)的存储提供设施。复用库包含一个数据库以及查询数据库和构件检索所必需的工具，构件分类模式(26.5.1 节)是构件库查询的基础。

查询通常用前面描述的 3C 模型中的语境来刻划，如果某初始查询产生大量的候选构件，则查询被求精以减少候选对象。然后，概念和内容信息被抽取出来(在找到候选构件集后)以辅助开发者选择合适的构件。

关于复用库结构及管理它们的工具的详细讨论已超出本书范围，有兴趣的读者可参阅文献[HOO91]和[LIN95]。

26.6 软件复用经济学

在直觉上，软件复用是有吸引力的。从理论上讲，它将为软件组织提供在质量和时间上的优势，而这些将直接导致成本的节省。但是，有实际的数据支持我们的直觉吗？

为了回答这个问题，我们必须首先了解什么是在软件工程过程中可以复用的东西以及和复用相关联的成本究竟是多少。作为结果，有可能形成对复用的成本—收益分析。

26.6.1 对质量、生产率和成本的影响

过去几年，大量来自产业实例研究(例如参考文献[HEN95]、[MCM95]和[LIM94])的证据表明从积极的软件复用可获得实质性的商业收益，产品质量、开发生产率、以及整体成本都将得到改善。

质量。理想情况下，为了复用而开发的软件构件已被验证是正确的(见第 25 章)且不含有错

误。在现实中，形式化验证并不能定期地执行，错误可能、也确实存在。然而，随着每一次复用，错误被发现和消除，构件的质量也随之改善。随时间的推移，构件变成实质上无错误的。

在 HP 公司的研究中，Lim[LIM94]报告说：被复用代码的错误率是每 KLOC 有 0.9 个错误，而新开发代码的错误率是每 KLOC 有 4.1 个错误。对一个包含 60％复用代码的应用来说，错误率是每 KLOC 有 2.0 个错误——对期望的错误率有 51％的改善，相对于不使用复用的开发。Henry 和 Faller[HEN95]的研究指出在质量上有 35％的改善。虽然不同的报告得到不同的改善率(位于合理的范围内)，但是，公正地说，复用对交付的软件在质量和可靠性方面确实带来了实质性的收益。

生产率。当可复用软件制品被应用于软件开发全过程中，对开发一个可交付系统所必需的计划、模型、文档、代码和数据的创建工作将花费较少的时间，导致的结果是用较少的投入给客户提供了相同级别的功能，因此生产率得到了改善。虽然，对生产率改善百分率的报告是众所周知难于解释的，[①]但是，似乎 30％到 50％的复用可以产生 25％到 40％的生产率改善。

成本(COSt)。对复用带来的净成本节省估算如下：估算出项目如果是从头开始开发所需的成本C，然后减去和复用关联的成本C与被交付的软件的实际成本C_d之和。

C_s可以用第 5 章中讨论的估算技术的一个或多个来确定，和复用关联的成本C_r包括[CHR95]：

●领域分析和建模。

●领域体系结构开发。

●为促进复用所增加的文档量。

●可复用软件制品的维护和增强。

●对从外部获取的构件的版税和许可证(licenses)费。

●可复用构件库的创建或者获得以及操作。

●对设计和构造复用的人员的培训。

虽然和领域分析(26.4 节)与复用库操作相关联的成本可能是实质性的，但是它们由很多项目分担。上面提到的很多其他成本所强调的问题实际上是一个好的软件工程习惯的一部分，不管是否优先考虑复用。

26.6.2 使用结构点的成本分析

在 26.3.3 节中，我们将结构点定义为在特定应用领域中重复出现的体系结构模式。软件工程师(或系统工程师)可以通过定义领域体系结构然后以结构点来实现它为某新的应用、系统或产品开发体系结构。这些结构点或者是个体可复用的构件，或者是可复用构件的包。

虽然结构点是可复用的，但它们的适应性修改、集成和维护成本并不是微不足道的。在进行

复用前，项目管理者必须了解和使用结构点相关的成本。

因为所有结构点(和一般可复用构件)具有过去的历史，所以可为每一个收集成本数据。在理想情况下，和复用库中每个构件关联的适应性修改、集成和维护成本是复用的每一次实例所分担的，这些数据可以被分析以确定复用的下一次实例的成本。

作为一个例子，考虑一个新应用X，它需要60%的新代码，并且复用三个结构点SP_1，SP_2和SP_3。这些可复用构件的每一个均已经在一系列其他的应用中使用过，它们的适应性修改、集成和维护的平均成本是已知的。

为了估计出交付X所需要的成本，必须确定下面公式：

$$\text{Overall effort}=E_{new}+E_{adapt}+E_{int}$$

这里，E_{new}是开发和构造新软件构件需要的成本(用第5章中技术确定)

E_{adapt}是对SP_1，SP_2和SP_3进行适应性修改所需的成本

E_{int}是集成SP_1，SP_2和SP_3所需的成本

对SP_1，SP_2和SP_3进行适应性修改和集成所需的成本通过取历史数据的平均值而得到，这些历史数据是从可复用构件在其他应用中进行适应性修改和集成时收集的。

26.6.3 复用度量

为了测度在基于计算机的系统中复用的收益，一系列的软件度量体系提出了。在系统S中和复用关联的收益可以表示为：

$$R_b(S)=[C_{noreuse}-C_{reuse}]/C_{noreuse}$$

这里C是不使用复用开发S的成本

C_{reuse}是使用复用开发S的成本

比率$R_b(S)$可以表示为在如下范围内的无单位值：

$$0\leqslant R_b(S)\leqslant 1 \quad (26.2)$$

Devanbu及其同事[DEV95]建议：(1)Rb将被系统的设计所影响，(2)因为R_b被系统的设计影响，所以应将R_b作为设计选择评估的一部分，(3)和复用关联的收益和每个个体构件的成本收益紧密相关。$R_b(S)$的值越高，复用就越有吸引力。

在面向对象系统中对复用的一种一般性测度，称为复用影响(reuse leverage)[BAS94a]，被定义如下：

$R_{lev}=OBJ_{reused}/OB_{Jbuilt}$ (26.3)

这里OBJ_{reused}是系统中复用的对象数

OBJ_{built}是构建系统的对象数

显然，当R_{lev}增加时R_b也增加。

26.7 小结

构件复用为软件质量、开发者生产率、以及整个系统成本带来了固有的收益，然而，在复用过程模型被广泛地用于软件产业前，必须克服很多障碍。

软件工程师可以获得一系列可复用的软件制品，这些包括软件的技术表示(例如，规约、体系结构模型、设计和代码)、文档、测试数据，甚至包括过程相关的任务(如，检查技术)。

复用过程包括两个并发的子过程：领域工程和软件工程。领域工程的目的是在特定应用领域中标识、构造、分类和传播一组软件制品。然后，软件工程可在新系统开发中选取这些软件制品作为复用。

对可复用构件的分析、设计技术采用和在良好的软件工程实践中使用的相同原则和概念。可复用构件应该在一个环境中设计，该环境为每个应用领域建立标准数据结构、接口协议和程序体系结构。

基于构件的开发使用数据交换模型、工具、结构化存储、以及底层对象模型利用已经存在的构件来构造应用。对象模型通常遵从一个或多个构件标准(例如，OMG/CORBA)，它们定义了应用访问可复用对象的方式。分类模式使得开发者能够发现和检索可复用构件并遵从明确标识概念、内容和语境的模型。枚举分类、刻面分类和属性—值分类是很多构件分类模型中的典型代表。

软件复用经济学可用一个问题来表达：少构建多复用是成本合算的吗？通常，答案是“yes”，但是，一个软件项目计划者必须考虑和可复用构件的适应性修改及集成相关联的、可观的成本。

参考文献

[ADL95] Adler，R.M.，“Emerging Standards for Component Software，”IEEE Computer，vol 28，no.3， March 1995，pp.68-77.

[BAS94a] Basili,V.R.，L.C.Briand，and W.M.Thomas，“Domain Analysis for the Reuse ofSoftware Development Experiences，”Proc.19th Annual Software EngineeringWorkshop，NASA/GSFC，Greenbelt，MD，December 1994.

[BAS94b] Bassett，P.G.，What Is Frame Technology？Netron，Inc.，Toronto，Canada，

October1994.

[BEL95] Bellinzona R., M.G.Gugini, and B.Pernici, “Reusing Specifications in OOApplications, ”IEE E Software, March 1995, pp.65-75.

[BIN93] Binder, R., “Design for Reuse is for Real, ”American Programmer, vol.6, no.8, August 1993, pp.30-37.

[CHR95] Christensen, S.R., “Software Reuse Initiatives at Lockheed, ”CrossTalk, vol.8, no.5, May 1995, pp.26-31.

[DEV95] Devanbu, P.et al., “Analytical and Empirical Evaluation of Software ReuseMetrics, ” Technical Report, Computer Science Department, University ofMaryland, August 1995.

[FRA94] Frakes, W.B., and T.P.Pole, “An Empirical Study of Representation Methods forReusable Software Components, ” IEEE Trans.Software Engineering, vol.20, no.8, August 1994, pp.617-630.

[FRE87] Freeman, P., “A Perspective on Reusability, ”in Software Reusability, P.Freeman(ed.), IEEE Computer Society Press, 1987, pp.2-3.

[HEN95] Henry, E., and B.Faller, “Large Scale Industrial Reuse to Reduce Cost and CycleTime, ”IEEE Software, September 1995, pp.47-53.

[HOO91] Hooper, J.W., and R.O.Chester, Software Reuse:Guidelines and Methods, PlenumPress, 1991.

[HUT88] Hutchinson, J.W., and P.G.Hindley, “A Preliminary Study of Large Scale SoftwareReuse, ”Software Engineering Journal, vol.3, no.5, 1988, pp.208-212.

[JON94] Jones, C., “The Economics of Object-Oriented Software, ” American Programmer, vol.7, no.10, October 1994, pp.28-35.

[LIA93] Liao, H., and Wang, F., “Software Reuse Based on a Large Object-Oriented Library, ”Software Engineering Notes, vol.18, no.1, January 1993, pp.74-80.

[LIM94] Lim, W.C., “Effects of Reuse on Quality, Productivity, and Economics, ”IEEESoftware, September 1994, pp.23-30.

[LIN95] Linthicum, D.S., “Component Development(a special feature), ”ApplicationDevelopment Trends, June 1995, pp.57-78.

[MCM95] McMahon, P.E., “Pattern-Based Architecture: Bridging Software Reuse and CostManagement, ”Crosstalk, vol.8, no.3, March 1995, pp.10-16.

[ORF96] Orfali, R.D., Harkey, and J.Edwards, The Essential Distributed Objects SurvivalGuide, Wiley, 1996.

[POL94] Pollak, W., and M.Rissman, “Structural Models and Patterned Architectures, ”IEEE Computer, vol.27, no.8, August 1994, pp.67-68.

[PRI87] Prieto-Diaz, R., “Domain Analysis for Reusability, ”Proc.COMPSAC′87, Tokyo, October 1987, pp.23-29.

[PRI93] Prieto-Diaz, R., “Issues and Experiences in Software Reuse, ”AmericanProgrammer, vol.6, no.8, August 1993, pp.10-18.

[STA94] Staringer, W., “Constructing Applications from Reusable Components, ”IEEESoftware, September 1994, pp.61-68.

[TRA90] Tracz, W., “Where Does Reuse Start? ”Proc.Realities of Reuse Workshop, Syracuse University CASE Center, January 1990.

[TRA95] Tracz, W., “Third Internatlonal Conference on Software Reuse-Summary, ”Software Engineering Notes, vol.20, no.2, April 1995, pp.21-22.

[WHI95] Whittle, B., “Models and Languages for Component Description and Reuse, ”Software Engineering Notes, vol.20, no.2, April 1995, pp.76-89.

[YOU94] Yourdon, E., “Software Reuse, ”Application Developmnet Strategies, CutterInformation Corp., vol, VI, no.12, December 1994, p.11.

思考题

26.1 复用的关键障碍之一是使软件开发者考虑复用现存的构件，而不是重新发明新的构件。(毕竟,构建东西是有趣的!)请建议 3 到 4 种软件组织可以用来激励软件工程师进行复用的方式。为了支持复用效果，应该采用什么技术？

26.2 在本章讨论的“复用软件制品”中有项目计划和成本估计。这些可被如何复用？这样做带来什么收益？

26.3 研究一下领域工程并丰富在图 26.2 中概述的过程模型。标识对领域分析和软件体系结构开发所需的任务。

26.4 应用领域的领域特征和构件分类模式有什么相同点和不同点？

26.5 为和大学的学生数据处理相关的信息系统开发一组领域特征。

26.6 开发一组和字处理/桌面出版软件相关的领域特征。

26.7 对你的导师指定的或你熟悉的某应用领域开发一个简单的结构模型。

26.8 什么是结构点？

26.9 获取一份最近的 OMG/CORBA 标准的拷贝，准备 3 到 4 页的讨论其主要特点的论文。获取关于对象请求代理工具的信息，并举例说明工具如何符合标准。

26.10 针对你导师指定或你熟悉的某应用领域，设计一种枚举分类模式。

26.11 针对你导师指定或你熟悉的某应用领域，设计一种刻面分类模式。

26.12 研究文献以获取最近的支持使用复用技术的质量和生产率数据。

26.13 某面向对象系统被估计共需要 320 个对象构成。进一步估计表明 190 个对象可以从现存的库中获得，那么，复用影响是多少？假定新对象成本为$1000，且适应性修改一个对象的成本为$600，集成一个对象的成本为$400，那么，系统的估计成本是多少？$R_b$的值是多少？

推荐阅读文献及其他信息源

关于复用的文献正快速增多。Lim(Managing Software Reuse，Prentice-Hall，1996)讨论了一些管理问题，诸如在公司环境中实现和部署复用的成本理由和策略；Bassett(Framing SoftwareReuse：Lessons from the Real World，Yourdon Press，1996)描述了一些针对大型信息系统项目的复用的自动化工具的产业实现；Karlsson(Software Reuse: A Holistic Approach，Wiley，1995)给出了一个针对复用更技术的方法。

由 Freeman(Software Reusability IEEE Computer Society Press，1987)和 Tracz(Software Reuse:Emerging Technology，IEEE Computer Society Press，1988)编辑的教程是对早期文章的优秀汇集；Hooper 和 Chester[HOO91]覆盖了所有重要的子课题并提出了一个完整的书目；Tracz(Confessions of a Used Program Salesman:Institutionalizing Software Reuse，Addison-wesley，1995)以一种轻松的、有意义的方式讨论了和创建复用文化关联的一些问题；Nierstrasz 和 Tsichritzis(Object-Oriented Software Composition，Prentice-Hall，1996)编辑了一本书，书中给出了一系列关于面向对象软件组装和复用的研究项目的结果。

Biggerstaff 和 Perlis(Software Reusability，Volumes.1 and 2，ACM Press，1989)，以及 Schaefer，Prieto-Diaz 和 Matsumoto(Software Reusability Ellis Horwood，New York，1994)编辑的文集包含了对正在进行的研究和观点的有用的分析；IEEE Software 的 1994 年 9 月期是一个关于复用的专集，包含了一组有用的文章。

关于对象复用的流行标准的书籍有：Orfali(The Essential Distributed Objects SurvivalGuide，Wiley，1995)，Mowbray 和 Zahavi(The Essential CORBA，Wiley，1994)，以及 Denning(OLE Controls: Inside and Out，Microsoft Press，1995)，每本书均提供了对象模型的详细的技术描述。

电子快讯“Reuse News”定期出版，可以在一系列关于软件的新闻组中找到，包括

comp.object 和 comp.software-eng。可以在 WWW 上找到大量关于复用的信息源，很多包含了指向其他地址的指针。下面的 Web 地址提供了大量关于复用的有趣论文的 FTP，包括对复用度量和其他相关话题的讨论：

http：//rbse.jsc.nasa.gov/eichmann/wisr/wisr.html

http：//www.cs.umd.edu/projects/SoftEng/tame/

下面 WWW 地址包含了关于复用的一般信息、技术文章、和/或指向关于复用和相关话题的信息源的指针：

ARPS STARS Reuse Paper

http：//www.stars.ballston.paramax.com/Papers/ReusePapers.html

ASSET

http：//source.asset.com/

Courseware in Software Reuse

http：//ricis.cl.uh.edu/virt-lib/reuse_courses.htmlEuropean SER Consortium

http：//www.sema.es/projects/SER

Loral(Unisys)Domain Engineering

http：
//www.stars.reston.unisy.gsg.com/process/domain_engineering/DE_guidebook.htmlObject Management Group

http：//www.omg.org/

Pacific Software Research Center

http：//www.cse.ogi.edu/PacSoft

SEI SE Information

http：//www.sei.cmu.edu/

Software Design by Reuse

http：//www.ksl.stanford.edu/KSL_Abstracts/KSL-92-38.html

SPC Reuse Adoption Guidebook

http：//software.software.org/vcoe/products/reuse_adoption_gb.html

U.Leipzig-Reuse/Reengineering

http：//rzaix340.rz.unileipzig.de/～siebert/reuse.html

关于使用 WWW 来维护复用库的信息可以在下面地址找到：

http：//www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/werkman/www94.html 一系列不同应用领域的可复用软件库可在下面 WWW 地址找到：

Defense Software Repository

http：//ssedl.ims.disa.mil/srp/dsrs/page.htmlElectronic Library Services(ELSA)

http：//rbse.mountain.net/ELSA/

Guide to Mathematical Software

http：//gams.nist.gov/

HPCC National Software Exchange

http：//www.netlib.org/nse/

Public Software Archives

http：//wuarchive.wustl.edu/

Public Ada Library

http：//wuarchive.wustl.edu/languages/ada/Reusable DoD Software(CARDS)

http：//dealer.cards.com/

Reuse Based on OO Technology

http：//www.sema.es/projects/SER/reboot.html关于软件复用的WWW参考文献最新列表可在http：//www.rspa.com找到。

① 该节重组和编辑了Edwarderard1994 年发布在Internet新闻组Comp.software-eng中的一系统文章，这些资料已获引用权。

① 在本书前的章节中已用了saftHome安全系统作用例子。

② 为了对这些主题有更多的了解，请参阅本书第 13，14，16，17，21 和 25 章。

① 通常，对复用的设计的准备应该在区域工程中考虑（见第 27.3 节）。

① 对“分布式对象”标准的出色讨论在[ORF94]中，有重要的讨论。

② 从http：//WWW.omg.org/中可获得OMG文档。

① [WHI95]中含有大量文献。

① 所有可能的操作仅有一个部分被提到了。

① 有许多其他的因素（如应用领域，问题的复杂性，开发队伍的结构及大小，方案的时效性，可应用的技术）对方案的产生都有影响。

第 27 章　再工程

在一篇为“Harvard Business Review”撰写的文章中，Michael Hammer[HAM90]为考虑业务过程和计算的管理的革命奠定了基础：

该是停止铺设牛道的时候了。代替将过时的过程嵌入到芯片和软件中，我们应该删除它们并重新开始。我们应该“再工程(reengineer)”我们的业务：使用现代信息技术的力量去从根本上重新设计我们的业务过程，以达到它们性能的极大改善。

每个公司按照很多无关联的规则运作…，工程力图摆脱我们用于组织和管理业务的旧的规则。

和所有革命一样，Hammer的鼓吹①产生了正面和负面的改变。某些公司已经进行了合理努力的再工程，其结果导致了竞争力的改善；而另一些公司仅仅依赖于减小规模和外购(代替再工程)去改善它们的底线，这很容易地产生了一些对未来发展几乎没有潜力的“平庸”组织[DEM95]。

但是，业务再工程和软件再工程间的关系是什么？答案位于“系统视图”中。

软件经常是 Hammer 所讨论的业务规则的实现，当这些规则改变时，软件也必须改变。今天，大多数的公司有成千上万的支持“旧的业务规则”的计算机程序，当管理者试图修改规则以达到更高的效率和竞争力时，软件也必须保持同步。在某些情况下，创建这意味着大量新的基于计算机的系统，但是，在多数情况下，这意味着现有应用修改和/或重建，以使得它们将能够胜任满足 21 世纪的业务需要。

在本章中，我们以自顶向下的方式考察再工程，从业务过程再工程的概述开始，进而到更详细地讨论软件再工程中发生的技术活动。

27.1 业务过程再工程

业务过程再工程(BPR)远远超出了信息技术和软件工程的范畴。在很多对 BPR 给出的定义(大多数有点抽象)中，有一个定义出现于 Fortune 杂志[STE93]：“搜寻并实现业务过程中的根本性改变以达到突破性结果”。但是，如何执行搜

寻？如何完成实现？更重要的是，我们如何能够保证建议的“根本性改变”将事实上导致“突破性结果”而不是组织的混乱？

在 1990 年代早期，有大量和 BPR 关联的过分宣传和广告。今大，形势已趋于成熟。事实上，很多 BPR 早期的鼓吹者已经大大地缓和了他们的观点。然而，没有人对需要使业务更具竞争力存有疑问。信息技术是达到此目标的工具，而是否 BPR 对此具有作用则仍然是争论中的话题。

本节给出了业务过程再工程的概述，我们的意图是建立一个实施软件再工程的语境。

27.1.1 业务过程

一个业务过程是“一组逻辑相关的任务，执行它们以达到预定义的业务结果”[DAV90]。在业务过程中，人、设备、材料资源、以及业务规程被组合在一起以生成特定结果。业务过程的例子包括：

- 设计新的产品。
- 购买服务和供给。
- 雇佣新雇员。
- 向供给商付费。

每一个都需要一组任务，而且在业务中每一个利用不同的资源。

每个业务过程有一个指定的客户——即接收过程结果(例如，概念、报告、产品)的人或小组。此外，业务过程跨越组织边界，它们需要不同的组织小组参与构成过程的“逻辑相关的任务”。

在第 10 章，我们注意到每个系统实际上是子系统构成的层次。业务也不例外，图 27-1 例举了 BPR 在其中运作的层次。整个业务被划分为一组业务系统(也称为业务功能)，每个业务系统由一个或多个业务过程组成，而每个业务过程由一组子过程定义。

BPR 可以被用于图 27-1 显示的层次的任意层，但是，随着 BPR 的范围扩大(即，我们在层次中向上移动)，和 BPR 关联的风险也急剧增长。为此，大多数 BPR 工作着重于个体过程或子过程。

信息技术(IT)的能力及在业务中发生的过程再工程之间存在很强的循环关系，随着 IT 能力增长，它们可以促使业务过程中的改变，图 27-2 说明了这一关系。

作为一个例子，考虑笔记本计算机。随着此技术变得日益广泛，它刺激了在很多业务过程中的变化。销售人员可以为甚至最复杂的工作或产品提供即时的报价；产品目录册消失了，代之以 CD-ROM 或磁盘，它们可以在现场被访问。当销售过程改变后，在此领域的人们看到了增多的机会，他们说，和总部办公室的通信是必须的。IT 对此的回应是新的笔记本计算机(和合适的软件)，它们提供了蜂窝通信能力。该循环一直继续着。

27.1.2 业务过程再建工程的原则

在很多方面，BPR 的关注点和范围与第 10 章讨论的信息工程过程是相同的。在理想情况下，BPR 应该以自顶向下的方式进行，从标识主要的业务目的和目标开始，以对规定特定业务过程的任务的非常详细的规约为结束。

Hammer[HAM90]给出了一组原则，用于指导从顶(业务)开始的 BPR 活动：

围绕结果、而不是任务进行组织。很多公司划分业务活动，使得没有单个的人(或组织)对业务的结果具有责任(或控制)。在这样的情况下，很难确定工作的状况，更难调试过程问题(如果它们确实发生)。BPR 应该设计过程以避免这一问题。

让那些使用过程结果的人来执行过程。该建议的意图是允许那些需要过程结果的人去控制那些允许他们以及时方式得到结果的所有变量。涉及到过程中的独立客户越少，快速通往结果的路就越平滑。

将信息处理工作结合到生产原始信息的现实工作中。随着 IT 技术变得更分布化，则有可能将大多数信息处理工作局部于生产原始数据的组织中。这使得控制局部化，减少通信时间，并将计算能力交给那些对所生产的信息有很大兴趣的人。

将地理分布的资源视为好象它们是集中的。基于计算机的通信已经变得如此高级，使得地理分布的小组可以被放置于同一个“虚拟办公室”。例如，替代在单个位置运行三个工程轮班，一个全球公司可以在欧洲运行第一轮班，在北美运行第二轮班，而在亚洲运行第三轮班。在每种情形，工程师将在白天时间工作并通过高带宽网络进行通信。

连接并行活动以代替集成它们的结果。当不同的客户在并行地执行工作时，有必要设计一个过程，该过程需要不断的通信和协调，否则，集成问题肯定会出现。

在执行工作的地方设置决策点，并将控制加入过程中。使用软件设计的行话来说，该原则建议一个更平坦的具有简单因子化的组织结构。

在其源头一次性获取数据。数据应该被联机存储，使得一旦收集到数据，则再也不需要重新输入。

上面的每条原则代表了 BPR 的一个“大图”视图，在这些原则的指导下，业务计划者和过程设计者必须开始过程再设计。在下一节，我们将更详细地考察 BPR 过程。

27.1.3 BPR 模型

和大多数工程活动一样，业务过程再工程是迭代的。业务目标及达到目标的过程必须适应变化中的业务环境，为此，对 BPR 而言，没有开始和结束——它是一个演化的过程。图 27-3 描述了一个业务过程再工程模型，该模型定义了 6 个活动：

业务定义。在 4 个关键驱动因素(减少成本、减少时间、质量改善，以及人力开发和授权)的环境内标识业务目标，这些目标可以在业务级或针对某特定业务成分来定义。

过程标识。标识对达到在业务定义中定义的目标关键的过程，然后它们可以根据重要性及改变的需要区与优选先级别、或以任何其他适于再工程活动的方式区分优先顺序。

过程评价。彻底分析和测度现存过程，包括：标识过程任务，注释过程任务花费的成本和时间，以及隔离质量/性能问题。

过程规约和设计。基于在上面三个 BPR 活动中获得的信息，为每个将被重新设计的过程准备 USE cases(第 20 章)。在 BPR 的范围内，use cases 标识了交付某些结果给客户的场景。使用 usecases 作为过程的规约，为过程设计一组新任务(遵从在 27.2.1 节中提到的原则)。

原型实现。一个重设计的业务过程在被完全地集成进业务中前，必须被原型化。该活动“测试”过程，使得求精工作可以进行。

求精和实例化。基于来自原型的反馈，精化业务过程，然后在业务系统中实例化。

上述的 BPR 活动有时是和工作流分析工具配合使用的，那些工具的目的是建立现存工作流的模型以便更好地分析现存过程。此外，通常和信息策略计划及业务范围分析(第 10 章)等信息工程活动关联的建模技术可以用于实现上面过程模型中描述的前 4 个活动。

27.1.4 几句警告

一个就业务方法(在此情况是 BPR)首先被宣传为万能药，然后又被严重地批评，变得一钱不值，是司空见惯的。在过去几年中，关于 BPR 的功效的争论大为

风行(例如参考文献[BLE93]和[DIC95])。Weisz[WEI95]在其对支持和反对 BPR 的情况的精彩总结中，对争论作出了如下总结：

人们试图将 BPR 视为另一个银弹(silver-bullet)时尚。从几个观点——系统思想、人件(Peopleware)、简单历史——来看，你可能必须对该概念预测高的失败率，该失败率似乎已为经验证据所证实。对很多公司来说，银弹显然是失去了。

虽然，对其他公司而言，再工程努力显然是卓有成效的…

如果 BPR 的应用是由有动机的、受过训练的、认识到过程再工程是一个持续活动的人员来执行，它确实可以具有成效。如果 BPR 被有效地执行，信息系统将被更好地集成进业务过程中。对旧的应用的再工程可以在更广阔的业务策略范围内考察，并且明智地建立软件再工程的优先级。

但是，即使业务再工程被公司所拒绝，软件再工程有时也是必须进行的。成千上万的遗产系统——对业务(无论大小)的成功至关重要的应用——是急需重更新或重构建的。

27.2 软件再工程

这种场景是太常见的：某应用已经为某公司的业务需要服务了 10 或 15 年。在那段时间中，它曾被多次纠错、适应性修改和增强。完成这些工作的人们具有最好的愿望，但是，好的软件工程习惯总是被抛到一边(由于其他事情的压力)。现在，应用是不稳定的，它仍然工作，但每次修改后的产生未预料到的和严重的副作用。然而，应用仍必须继续演化，我们应做些什么？

不可维护的软件并不是一个新问题。事实上，对软件再工程的强调是源于 30 多年不断形成的软件维护“冰山(iceberg)”。

27.2.1 软件维护

几乎 30 年以前，软件维护被刻划[CAN72]为“冰山”。我们希望那些一眼可见的就是所有实际存在的，但是我们知道，在表面之下存在大量潜在的问题和成本。在 1970 年代早期，维护冰山大到足以使一艘航空母舰沉没。而今天，它可以容易地沉没整个海军。

对现存软件的维护可能占据开发组织所花费的所有努力的 60％以上，而且随着更多的软件被生产出来，这个百分比在继续上升[HAN93]。在当前视野内，我们可以预见，“维护束缚(maintenance-bound)”的软件开发组织不可能再生产新的软件，因为所有可用的资源被花费在旧软件的维护上。

有的读者可能会问，为什么需要这么多维护？为什么要花费这么多的努力来进行维护？Osborrne 和 Chikofsky[OSB90]提供了部分答案：

很多我们现在使用的软件已有 10 到 15 年的使用史。即使这些程序是采用当时最好的设计和编码技术创建的(大多数并不是如此)，当时主要关注点是程序大小和存储空间。然后它们被移植到新的平台上，根据机器和操作系统技术方面的变化进行调整并增强以满足新的用户需要——所有这些并没有对整体体系结构给予足够关注。

结果是：我们现在仍保持运行的软件系统具有设计很差的结构、糟糕的编码、不完全的逻辑、以及贫乏的文档…

变化是所有软件工作中普遍存在的性质，当构建基于计算机的系统时变化是不可避免的，因此，我们必须开发评价、控制和进行修改的机制。

读了上面的一段话，读者可能抗议说：“但是我并没有花费我的 60%的时间去修正我所开发的程序中的错误”。当然，软件维护并不仅仅是“修正错误”，我们可以通过描述程序被发布使用后发生的 4 类活动[SWA76]来定义维护：

- 纠错性维护。
- 适应性维护。
- 完善性维护或增强。
- 预防性维护或再工程。

所有维护工作中仅仅大约 20%花费于“修正错误”。其余的 80%是花费在：修正现有系统已适应外部环境的改变，根据用户要求进行增强，以及为了未来的使用对应用进行再工程。当维护包括所有这些活动时，我们可以相当容易地看出为什么它花费那么多的劳动。

27.2.2 软件再工程过程模型

再工程花费时间；它花费大量的钱财，并占用资源(否则可被当前的关注点所占用)。由于所有这些理由，再工程不是在几个月或甚至几年内可以完成的。信息系统的再工程是一个将占据信息技术资源达几年的活动，这也是为什么每个组织都需要软件再工程的实际策略。

一个可操作的策略被包含在再工程过程模型中，我们将在本节后部讨论该模型，但是首先，讨论某些基本原则。

再工程是一个重新构建活动，如果我们考虑一个类比活动：重建一所房子，我们可能能够更好的理解信息系统的再工程。考虑如下情况：

让我们假定你在另一个州购买了一所房子。你从来没有实际地看到该房产，但是你以一个令人惊异的低价格获得了它，给你的警告是它可能需要彻底地重建。你将如何进行该项工作？

• 在你开始重建前，检查一下房子似乎是合理的。为了确定它是否确实需要重建，你(或职业的检查员)将列出一组标准，以使得你的检查可系统地进行。

• 在你拆掉并重建整个房子前，确认其结构是不牢固的。如果该房子结构良好，则可能是“改造(remodel)”而不是重建(以低得多的价格和少得多的时间)。

• 在你开始重建前，确认你已经了解了原房是如何建造的。到墙内部看一看，了解布线、管道、以及内部结构。即使你废弃掉所有这些，对原房的洞悉对你开始建造是一定有帮助的。

• 如果你开始重建，只使用最现代的、可耐久的材料。这可能会贵一些，但是，它将帮助你避免以后的昂贵的和耗时的维护。

• 如果你决定重建，一定要采用严格的方式。使用在今天和未来均将导致高质量的惯例。

虽然上面的原则着重于房子的重建，它们也同样很好地适用于基于计算机的系统和应用的再工程。

为了实现这些原则，我们应用如图 27-4 所示的软件再工程过程模型，它定义了 6 类活动。在某些情况下，这些活动以线性顺序发生，但并不总是这样，例如，有可能在文档重构开始前，必须先进行逆向工程(理解某程序的内部工作原理)。

在图中显示的再工程范型是一个循环模型。这意味着作为该范型的一部分的每个活动均可能被重复，对任意特定的循环，过程可以在任意一个活动之后终止。

库存目录分析(Inventory Analysis)。每个软件组织应该保存所有应用的库存目录。

该目录可能仅仅是包含如下信息的一个电子表格模型：

• 应用的名字。

• 最初构建的年份。

• 已对它进行过的实质性修改的次数。

• 完成这些修改所花费的总劳动。

• 最好一次实质性修改的日期。

- 最好一次实质性修改所花费的劳动。
- 它驻留的系统。
- 和它有接口的应用。
- 它访问的数据库。
- 过去 18 个月所报告的错误。
- 用户数量。
- 它被安装的机器数量。
- 程序结构的复杂性、代码的复杂性和文档的复杂性。
- 文档的质量。
- 整体可维护性(等级值(scale value))。
- 预期寿命(以年算)。
- 预期在未来 36 个月内的修改次数。
- 年度维护成本。
- 年度运作成本。
- 年度业务值。
- 业务重要程度。

应该对每一个现存的应用收集上面列出的信息，通过按照业务重要程度、寿命、当前可维护性、以及其他局部重要标准对这些信息排序，可以选出再工程的候选者，然后可以明智地分配资源。

必须注意：上面描述的目录表应该定期整理修订，应用的状况(如，业务重要程度)可能随时间发生变化，其结果是：再工程的优先级将发生变化。

文档重构。贫乏的文档是很多遗产系统的注册商标。但是，我们对此可做些什么呢？

选项 1：建立文档是非常耗费时间的。系统正常运作，我们将保持现状。在某些情况下，这是一个正确的方法，不可能为数百个计算机程序重新建立文档。如果一个程序是相对静止的，正在走向其有用生命的末端，并且可能不会再经历什么变化，那么，让它保持现状。

选项 2：文档必须更新，但是，我们只有有限的资源。我们将使用“使用时建文档”的方法。可能不需要对某应用全部重构文档，而是对系统中当前正在进行改变的那些部分建立完整文档。随时间流逝，将得到一组有用的和相关的文档。

选项 3：系统是业务关键的，而且必须完全地重构文档。即使在此情形，明智的方法是设法将文档工作减少到必需的最小量。

每个选项均是可行的，软件组织必须选择最适合于每种情形的方法。

逆向工程。术语“逆向工程”源于硬件领域，一个公司分解某竞争者的硬件产品去了解竞争者的设计和制造“秘密”。如果得到了竞争者的设计和制造规约，则这些秘密可以被很容易地理解。但是，这些文档是别人专有的，对做逆向工程的公司来说是不可得到的。本质上，成功的逆向工程通过检查产品的实际样本导出一个或多个关于产品的设计和制造规约。

软件的逆向工程是相当类似的。然而，在大多数情况下，被逆向工程的程序不是来自于竞争者，而是公司自己的软件(经常是很多年以前的)。将被理解的“秘密”是未开发过相关规约所带来的模糊不明。因此，软件的逆向工程是分析程序以在比源代码更高的抽象层次上创建程序的某种表示的过程。逆向工程是一个设计恢复过程，逆向工程工具从现存的程序中抽取数据、体系结构和过程的设计信息。

代码重构。最常见的再工程类型(实际上，在这里使用术语“再工程”是有疑问的)是代码重构。某些遗产系统具有相对完整的程序体系结构，但是，个体模块被以难于理解、测试和维护的方式编码。在这样的情形下，在可疑模块内的代码可被重构。

为了完成该活动，用重构工具去分析源代码。标注出和结构化程序设计概念相背的部分，然后重构代码(此工作可以自动进行)。复审和测试生成的重构代码以保证没有引入异常和不规则。更新内部的代码文档。

数据重构。数据体系结构差的程序将难于进行适应性修改和增强。事实上，对很多应用来说，数据体系结构比源代码本身对程序的长期生存力有更大影响。

和代码重构不同，数据重构发生在相当低的抽象层次上，它是一种全范围的再工程活动。在大多数情况下，数据重构以逆向工程活动为开始，当前的数据体系结构被分解。必要时，定义数据模型(第 12 章)，标识数据对象和属性，并从质量的角度复审现存的数据结构。

当数据结构较差时(例如，平坦文件正被实现，而关系型方法将大大地简化处理)，数据被再工程。

因为数据体系结构对程序体系结构及其中的算法有很强的影响，对数据的改变将总是会导致体系结构或代码层的改变。

正向工程。在理想的情况，可以使用自动的“再工程引擎”来重建应用，旧的程序被输入引擎，分析、重构、然后以展示软件质量最好的方面的形式重新生成。在短期内，这样的“引擎”还不可能出现，但是，CASE厂商已经开发了一些工具，它们提供了针对特定应用领域的(例如，用特定数据库系统实现的应用)这些能力的有限子集。更重要的是，这些再工程工具正不断地变得越来越高级。

正向工程也称为革新或改造[CHI90]，不仅从现存软件恢复设计信息，而且使用该信息去改变或重构现存系统，以改善其整体质量。在大多数情况，被再工程的软件重新实现现存系统的功能，并且加入新功能和/或改善整体性能。

27.3 逆向工程

逆向工程用魔法召唤出“魔术槽”的映象，我们将无结构的、无文档的源程序列喂到槽中，在另一端出来的是计算机程序的完整文档。不幸的是，该魔术槽并不存在。逆向工程可以从源程序抽取出设计信息，但是，抽象的层次、文档的完整性、工具和分析员一起工作的程度、以及过程的方向性却是高度可变的。

逆向工程过程及用于实现该过程的工具的抽象层次是指可从源代码中抽取出来的设计信息的精密程度。理想地，抽象层次应该尽可能高，即，逆向工程过程应该能够导出过程的设计表示(一种低层的抽象)；程序和数据结构信息(稍高一点层次的抽象)；数据和控制流模型(一种相对高层的抽象)；以及实体—关系模型(一种高层抽象)。随着抽象层次增高，软件工程师获得更有助于理解程序的信息。

逆向工程过程的完整性是指在某抽象层次提供的细节程度。在大多数情况，随着抽象层次增高，完整性就降低。例如，给定源代码列表，得到一个完整的过程设计表示是相对容易的，简单的数据流表示也可被导出，但是，要得到数据流图或状态—变迁图的完整集合却困难得多事。

完整性改善直接正比于做逆向工程的人员所完成的分析量。交互性是指为了建立一个有效的逆向工程过程，人和自动工具“集成”的程度。在大多数情况，随着抽象层次增高，交互必须增加，而完整性将遭受损害。

如果逆向工程过程的方向是单向的，所有从源程序中抽取的信息被提供给软件工程师，他们然后可以在任何维护活动中使用这些信息。如果方向是双向的，则信息被输入到再工程工具，以试图重构或重生成旧程序。

逆向工程过程如图27-5所示，在逆向工程活动可以开始前，无结构的(“脏的(dirty)”)源代码被重构(27.4.1节)使得它仅包含结构化程序设计结构，[①]这使得源代码容易阅读并为所有后续的逆向工程活动提供基础。

逆向工程的核心是被称为抽取抽象的活动，工程师必须评价旧程序并从源代码(经常是无文档的)中抽取出被执行的处理、被应用的用户界面、以及被使用的程序数据结构或数据库的有意义的规约。

27.3.1 理解处理的逆向工程

第一个真正的逆向工程活动从理解然后抽取源代码所表示的过程抽象的启图开始。为了理解过程抽象，要在不同的抽象层次分析代码：系统、程序、模块、模式和语句。

在进行更细节的逆向工程前必须理解整个系统的整体功能。这为进一步的分析建立语境，并提供对系统中应用间的互操作问题的洞悉。构成应用系统的每一个程序代表了在高详细层次的功能抽象，建立表示这些功能抽象间的交互的块图；每个模块执行某种子功能并表示某种定义的过程抽象，为每个模块建立处理叙述。在某些情况，系统、程序和模块规约已经存在，在此情况下，复审这些规约以保持和现存代码的相符性。[②]

当考虑模块中的代码时，事情变得更复杂。工程师需寻找表示类属过程模式的代码段。几乎在每个模块中，一个代码段为处理(在模块内)准备数据，一个不同的代码段完成处理工作，而另一个代码段准备处理的结果以从模块输出。在每个代码段中，我们可能遇到更小的模式(例如，数据确认和范围检查经常出现在为处理准备数据的代码段中)。

一种称为程序分割[NIN94]的技术被提出，作为用模型标识过程模式并包装这些模式为有意义的函数的一种方法。使用自动浏览工具，软件工程师隔离聚集操作(focusing operations)——功能(语义)相关的不相邻代码段。一旦聚焦操作被隔离出来，则应用因子化操作。被聚焦的代码段被从现存代码中抽取出来，并重新包装成新模块。

对大型系统，通常用半自动方法来完成逆向工程。CASE 工具(如参考文献[MAR94])被用于“解析(parse)”现存代码的语义，然后该过程的输出被传送给重构和正向工程工具以完成再工程过程。

27.3.2 理解数据的逆向工程

数据的逆向工程发生在不同的抽象层次。在程序层，作为整体再工程努力的一部分内部的程序数据结构必须被逆向工程。在系统层，全局数据结构(如，文件、数据库)经常被再工程以便符合新的数据库管理范型(如，从平坦文件移向关系型或面向对象数据库系统)，当前全局数据结构的逆向工程设置了引入新的系统范围数据库的场所。

内部数据结构。针对内部程序数据的逆向工程技术着重于对象的类的定义。[③]这是通过意图组合相关程序变量的对程序代码的检查来完成的。在很多情况下，在代码中的数据组织标识了抽象数据类型，例如，记录结构、文件、列表和其他数据结构经常提供类的初始指示器。

Breuer 和 Lano[BRE91]建议了逆向工程类的下列方法：

1.标识程序中记录关于全局数据结构(如，文件或数据库)的重要信息的标记和局部数据结构。

2.定义在标记和局部数据结构与全局数据结构间的关系。例如，在文件空时某标记可能被设置，或某局部数据结构可能作为一个缓冲区，它包含从某中心数据库获取的最后 100 个记录。

3.对每个表示一个数组或文件的变量(在程序中)，列出所有与其有逻辑联系的其他变量。

这些步骤使得软件工程师能够在和全局数据结构交互的程序中标识类。

数据库结构。不管其逻辑组织和物理结构，一个数据库允许数据对象的定义并支持在对象间建立关系的方法。因此，将一个数据库模式再工程到另一个模式需要对现存对象和它们的关系的理解。

可运用下面步骤[PRE94]将现存数据模型定义为再工程一个新的数据库模型的先驱：

1.通过复审在平坦文件数据库中的记录或在关系模式中的表建立一个初始的对象模型。可以获得被定义为模型的一部分的类。包含在记录或表中的项成为类的属性。

2.确定候选键。检查属性，确定它们是否被用于指向另一个记录或表，那些作为指针的属性成为候选键。

3.精化实验性的类。确定是否相似的类可被组合到某单个类中。

4.定义一般化。检查具有很多相似属性的类，确定是否应该构造一个，将一般化类放在其头部的类层次。

5.发现关联。使用类似于 CRC 方法(第 20 章)的技术来建立类间的关联。

一旦知道了定义在上面步骤中的信息，则可以应用一系列变换[PRE94]将旧的数据库结构映射为新的数据库结构。

27.3.3 用户界面的逆向工程

随着高级的 GUI 对每种类型的基于计算机的产品和系统成为必须的，用户界面的重新开发已经成为最常见的再工程活动类型之一。但是，在用户界面可以被重建之前，应该进行一个逆向工程活动。

为了我们能完全地理解某现存的用户界面(UI)，必须刻划界面的结构和行为。Merlo 及其同事[MER93]提出了三个基本问题，它们必须在 UI 的逆向工程开始前被回答：

• 什么是界面必须处理的基本动作——例如，击键和按鼠标？

• 什么是系统对这些动作的行为反应的简洁描述？

• “替代者”意味着什么？或更精确地说，什么界面的等价的概念是同这里相关的？

行为建模符号体系(第 12 章)可以提供一种表示上面提出的头两个问题的答案的工具，对创建行为模型必需的多数信息可以通过观察现存界面的外部表现而得到，但是，对创建行为模型必需的附加信息必须从代码中抽取。

进程代数可用于以形式化的方式表示界面的行为。Merlo 及其同事[MER93]有如下陈述：

在描述发生在界面中的进程时，一个关键的不寻常点是某些对象必须准备对用户输入作出反应，这些输入不能被控制，只能被拒绝。这样，人们必须抓住这个概念，至少有两种并发的活动实体：系统和用户。第二，人们必须能够表达一定的“外部”选择，这些选择不在用户的控制之下。这些因素、反应、并发性和外部选择是进程代数的基础。

对界面的描述使用代理和动作。一个代理是完成系统某方面功能的某种东西，动作允许代理间相互通信。在本质上，进程代数是一种，用于表示代理和动作如何完成某 UI 的功能的速记符号，基本语法如图 27-6 所示。

名字	符号	含义
非活动代理	0	
常量	A	代理 A
前缀	α.E	执行动作 a 并且象代理 E 那样执行
和	$E_1 + E_2$	代理 E1 复合代理 E2
或	$E_1 \backslash E_2$	象代理 E1 或代理 E2 那样执行
限制	E\L	代理 E 被限制到集合 L 中的动作
重标记	E[f]	按照函数 f 重命名代理 E
替代	E	X
递归	Fax(X=E)	代理 X ，使得 X=E

图 27-6 进程代数符号[MER93]

作为一个例子，考虑在工具条上有一个打印机图符的某现代界面，打印机图符 P 是一个代理。打印工作也可以通过键盘传送给打印驱动代理 D 的打印命令(动作 c)来完成。另一种方式，可以通过下拉菜单代理 M 来启动一个鼠标点击(动作 m)，该点击直接和打印驱动接口。使用为进程代数符号定义的“和”运算符(如图 27-6)，我们可以将打印行为表示为：

P=c.D+m.M

这表示代理P的行为方式是和动作c及代理D的结果行为或动作m及代理M的结果行为相同的。

关于进程代数及其在界面再工程中的使用的详细讨论已超出本书范围，有兴趣的读者可以参见文献[MER93]或[MIL89]。

27.4 重构

软件重构修改源代码和/或数据以使得它适应未来的变化。通常，重构并不修改整体的程序体系结构，它趋向于关注个体模块的设计细节以及定义在模块中的局部数据结构。如果重构扩展到模块边界之外并涉及软件体系结构，则重构变成了正向工程(27.5节)。

Arnold[ARN89]总结了软件重构可能得到的一系列的收益：

• 导致高质量的程序——更好的文档和低复杂性；同现代软件工程惯例和标准相一致。

• 减少那些必须操作程序的软件工程师的挫折感，因而改善生产率并使得学习更容易。

• 减少执行维护活动所需的工作量。

• 使得软件易于测试和调试。

当某应用的基本体系结构是坚固的时候发生重构，即使技术的内部细节需要修改。当软件的大部分是有用的，仅仅部分模块和数据需要扩展性修改时，启动重构活动。[①]

27.4.1 代码重构

代码重构的目标是生成可提供相同功能的设计，但是该设计比原程序有更高的质量。通常代码重构技术(例如，Warnier的逻辑简化技术[WAR74])用布尔代数对程序逻辑建模，然后应用一系列变换规则来重构逻辑。目标是采用“面条碗”式的代码并导出遵从结构化程序设计哲学(第14章)的过程设计。

27.4.2 数据重构

在开始数据重构前，必须先进行分析源代码的逆向工程活动。评估所有包含数据定义、文件描述、I/O、以及接口描述的程序设计语言语句，目的是抽取数

据项和对象，获取关于数据流的信息，以及理解现存的已实现的数据结构。该活动有时被称为数据分析[RIC89]。

一旦数据分析已完成，则开始数据重设计。作为其最简单的形式，数据记录标准化澄清数据定义以达到在现存数据结构或文件格式中的数据项名或物理记录格式间的一致性。另一种重设计形式称为数据名合理化，保证所有数据命名约定遵从局部标准并且当数据在程序中流动时别名被删除。

当重构超出标准化和合理化的范畴时，对现存数据结构进行物理修改以使得数据设计更为有效。这可能意味着从一种文件格式到另一种文件格式的转换，或在某些情况下，从一种类型的数据库到另一种类型数据库的转换。

27.5 正向工程

一个程序的控制流图形化等价于一碗面条，“模块”包含的语句有 2000 行，在 290.000 行源代码中几乎没有有意义的注释行，并且没有其他文档，如果此程序必须被修改以适应变化的用户需求，我们有下列选择：

1.我们可以努力地去不断修改，与隐含的设计和源代码“(fighting)”以实现必要的修改。

2.我们可以试图去理解程序的更多的内部工作，努力使修改更有效。

3.我们可以重设计、重编码并测试该软件需要修改的部分，对所有被修订的片断应用软件工程方法。

4.我们可以完全地重设计、重编码并测试该程序，使用 CASE 工具(再工程工具)来辅助我们理解现有的设计。

这里没有单个的“正确”选择。即使其他选择是更希望的，形势可能指定了第一种选择。

不是坐等维护请求的到来，开发或维护组织挑选一个程序，(1)它将在一定的年数内保持使用，(2)它当前是被成功地使用的，(3)它很可能会在不远的将来被较大程度地修改或增强。那么，可应用[①]上面的选择 2、3 或 4。

这种预防性维护方法是 Miller[MIL81]在“结构化翻新”这个标题下首先探讨的，他将这个概念定义为“将今天的方法学应用到昨天的系统以支持明天的需求”。

乍一看，这个在某大型程序的可用版本已经存在的情况下我们就重新开发它的建议似乎是相当浪费的，在下判断之前，考虑如下几点：

1.维护一行源代码的成本可能是该行代码初始开发成本的 20 到 30 倍。

2.采用现代设计概念重新设计软件体系结构(程序和/或数据结构)可以大大地有利于未来的维护。

3.因为软件的原型已经存在，开发生产率将远高于平均水平。

4.现在用户已对该软件有经验，因此，可以很容易地确定新的需求和变化的方向。

5.再工程的 CASE 工具自动执行部分工作。

6.完整的软件配置(文档、程序和数据)存在于预防性维护的完成…。

当软件开发组织将软件作为产品销售时，预防性维护体现在程序的“新发布”中。一个大的内部软件开发组织者(如，一个大型消费者产品公司的业务系统软件开发小组)可能在其责任范围内有 500 到 2000 个产品程序，可以根据其重要程度对这些程序进行优先排序，然后作为预防性维护的候选者进行复审。

正向工程过程应用软件工程的原则、概念和方法来重建造某现存应用。在大多数情况，正向工程并不仅仅是创建某旧程序的一个现代等价物，而是将新的用户和技术需求集成到再工程中，重新开发的程序扩展了旧应用的能力。

27.5.1 向客户/服务器体系结构的正向工程

在过去十年，很多主机型应用被再工程以适应客户/服务器(C/S)体系结构(第 28 章)。本质上，中心式的计算资源(包括软件)被分布到很多客户平台。虽然可以设计出一系列不同的分布式环境，被再工程为客户/服务器体系结构的典型的主机型应用具有下列特征：

- 应用功能迁移到每个客户计算机。
- 在客户端实现新的 GUI 界面。
- 数据库功能被分配给服务器。
- 特殊的功能(如，计算相关的分析)可以保留在服务器端。
- 必须在客户端和服务器端同时建立新的通信、安全、归档和控制需求。

必须注意，从主机型到 C/S 计算的迁移需要同时进行业务和软件的再工程，此外，必须建立“企业网络基础设施”[JAY94]。

针对 C/S 应用的再工程从对现存主机型应用所在的业务环境的彻底分析开始。可以标识出三层抽象(图 27-7)。数据库处于一个客户/服务器体系结构的基础位置，并且管理事务和查询，而这些事务和查询必须被控制在一组业务规则(由

现存的或再工程后的业务过程所定义)的范围内。客户应用提供面向用户的目标功能。

作为数据库基础层重设计的先驱，现存数据库管理系统的功能和现存数据库的数据体系结构必须被逆向工程。在某些情形，一个新的数据模型(第 12 章)被创建。在每种情况，C/S 数据库被再工程以保证：事务以一致的方式被执行，所有更新只能被授权的用户执行，核心业务规则被强制执行(例如，在某厂商记录被删除前，服务器保证该厂商没有相关的可付帐号、合同、或通信存在)，查询可被高效地完成，以及完善的归档能力被创建。

业务规则层表示同时驻留在客户和服务器端的软件，该软件执行控制和协调任务以保证在客户应用和数据库间的事务和查询遵从已建立的业务过程。

客户应用层实现特定终端用户群所需要的业务功能，在很多场合，主机型应用被分割为一组小的、再工程后的桌面应用，在桌面应用间的通信(当必要时)由业务规则层控制。

27.5.2 向面向对象体系结构的正向工程

面向对象软件工程正在迅速地成为很多软件组织选择的开发范型。但是，现存的用传统方法开发的应用怎么办？在某些情形，答案是保持这些应用的现状，而在另一些情形，旧的应用必须被再工程，使得它们能够被容易地集成进大型的面向对象的系统。

将传统的软件再工程为面向对象的实现采用很多在本书第四部分讨论的相同的技术。首先，现存的软件被逆向工程，以便建立适当的数据、功能和行为模型。如果被再工程的系统扩展原应用的功能或行为，则创建相应的 use cases(第 20 章)。然后，在逆向工程中创建的数据模型被结合 CRC 建模技术(第 20 章)一起使用，以建立类定义的基础。再定义类层次、对象—关系模型、对象—行为模型、以及子系统，并开始面向对象的设计。

随着面向对象的正向工程从分析进展到设计，可启用复用过程模型(第 26 章)。如果现存应用有一个已经存在很多面向对象应用的领域，则很可能已存在一个强壮的复用库并可在正向工程中被使用。

对那些必须从头开始开发的类，有可能复用来自现存传统应用的算法和数据结构，然而，这些必须被重设计以符合面向对象的体系结构。

27.5.3 用户界面的正向工程

随着应用从主机迁移到桌面，用户不再愿意忍受不可思议的、基于字符的用户界面。事实上，从主机型到客户/服务器计算的变迁中花费的所有工作量中的很大一部分是花费于客户应用的用户界面的再工程。

Merlo 及其同事[MER95]提出了下面的用户界面再工程的模型：

1.理解原界面以及在它和应用的其余部分间移动的数据。目的是理解其他的程序元素如何和实现界面的现存代码交互。如果开发新的 GUI，则在 GUI 和其他程序间流动的数据必须和当前在基于字符的界面和程序间流动的数据是一致的。

2.将现存界面蕴含的行为重新建模为一系列在 GUI 语境内有意义的抽象。虽然交互模式可能有本质的不同，但旧的和新的界面的用户展示的业务行为(当从使用场景的角度来考虑)必须保持相同。一个重设计的界面必须仍允许用户展示合适的业务行为，例如，当进行数据库查询时，旧的界面可能需要一长串的基于文本的命令来刻划查询，而再工程后的 GUI 可能简化为一个短的鼠标动作序列，但是，查询的意图和内容保持不变。

3.引入使交互模式更有效的改进。研究现存界面在工效方面的弱点，并在新的 GUI 的设计中进行修正。

4.建造并集成新的 GUI。类库和第四代工具的存在可以大大地减少建造 GUI 所需的工作量，然而，和现存应用软件的集成可能是更费时的。必须小心以保证 GUI 不会传播不利的副作用到应用的其余部分中。

27.6 再工程经济学

在一个完美的世界中，每个不可维护的程序将立即退出使用，被用现代软件工程惯例开发的高质量的、再工程后的应用所替代。但是，我们生活在一个资源有限的世界，再工程耗费可能用于其他业务目的的资源，因此，在一个组织试图再工程某现存应用前，应该进行成本—收益分析。

Sneed[SNE95]提出了再工程的成本—收益分析模型，定义了 9 个参数：

P_1=当前对某应用的年度维护成本

P_2=当前某应用的年度运作成本

P_3=当前某应用的年度业务价值

P_4=在再工程后的预期年度维护成本

P_5=在再工程后的预期年度运作成本

P_6=在再工程后的预期年度业务价值

P_7=估计的再工程成本

P_8=估计的再工程日程

P_9=再工程风险因子(名义上P9=1.0)

L=期望的系统生命期(以年为单位)

和某候选应用(即，未执行再工程)的持续维护相关联的成本可以定义如下：

$$C_{maint}=[P_3-(P_1+P_2)]\times L \qquad (27.1)$$

和再工程相关联的成本用下面关系定义：

$$C_{reeng}=[P_6-(P_4+P_5)\times(L-P_8)-(P_7\times P_9)] \qquad (27.2)$$

使用表示在方程(27.1)和(27.2)中的成本，再工程的整体收益计算如下：

$$Costbenefit=C_{reeng}-C_{maint} \qquad (27.3)$$

可以对所有在库存目录分析(27.2.2 节)中标识的高优先级应用进行上面方程中表示的成本—收益分析，那些显示最高成本—收益的应用可以作为再工程对象，而其他应用的再工程可以推迟到有足够资源时。

27.7 小结

再工程发生在两个不同的抽象层次。在业务层次，再工程着重于业务过程，企图改变业务过程以改善在某业务领域的竞争力；在软件层次，再工程检查信息系统和应用，企图重构它们以使其展示更高的质量。

业务过程再工程(BPR)定义业务目标，标识并评估现存业务过程(在定义的目标范围内)，刻划并设计修订的过程，并将它们在业务中原型化、精化和实例化。和信息工程一样，BPR 的关注扩展到软件之外，BPR 的结果经常是使信息技术能够更好地支持业务的方法的定义。

软件再工程包括一系列的活动：库存目录分析、文档重构、逆向工程、程序和数据重构以及正向工程。这些活动的目的是创建现存程序的可以展示更高质量和更好维护性的版本——将在 21 世纪中良好动作的程序。

库存目录分析使得组织能够系统地访问每个应用，目标是确定再工程的候选者；文档重构创建一个文档框架，它是对应用的长期支持所必需的；逆向工程是分析程序试图从中抽取出数据的、体系结构的、和过程的设计信息的过程；最后，正向工程使用现代软件工程惯例和在逆向工程中获得的信息重构程序。

再工程的成本和收益可以量化地确定，现状的成本(即，和某现存应用的不断的维护相关联的成本)与预期的再工程成本和在维护成本上产生的减少进行比较，在程序具有长的生命期且当前展示出弱的可维护性的几乎每一个情形，再工程均代表了一种成本合算的业务策略。

参考文献

[ARN89] Aruold,R.S,“Software Restructuring,”Proc.IEEE, vol.77, no.4, April 1989, pp.607-617.

[BLE93] BleaKLey, F.R, “The Best Laid Plans: Many CoMPanies Try Management Fads, Only to See Them FloP, ”The Wall Street Journal, July 6, 1993.

[BRE91] Breuer, P.T,and KLano,“Creating specifincation From Code: Reverse—EngineeringTechniques, ”Journal of Software Maintenance.Research and Practice, Volume 3, Wiley, 1991, pp.145-162,

[CAN72] Canning, R, “The Maintenance ‘Iceberg’, ” EDP Analyzer, vol.10, no.10, October 1972.

[CAS88] “Case Tools for Reverse Engineering, ”CA SE Outlook, CASE Consulting Group, Lake Oswego, OR, vol.2, no.2, 1988, pp.1—15.

[CHI90] Chikofsky, E.J, and J.H.Cross, Ⅱ, “Reverse Engineeriny and Design Recovery: ATaxonomy, ”IEEE Software, January 1990, pp.13—17.

思考题

27.1 考虑你在过去五年中从事过的任何工作，描述你在其中工作的业务过程。使用在 27.1.3 节中描述的 BPR 模型来建议对该过程的改变以使其更为高效。

27.2 对业务过程再工程的功效进行研究，给出对该方法的正面的和负面的论据。

27.3 你的老师将从在本课程中班级中每个人都已开发的程序中选择一个，随机的将你的程序和其他人的程序交换，不解释或走查该程序。现在，对你所接收的程序实现某些增强(由老师指定)。

a.完成包括粗略的走查(但不能和程序的作者交流)的所有软件工程任务。

b.对测试中遇到的所有错误保持仔细的跟踪。

c.在班上讨论你的经验。

27.4 使用在 27.2.2 节描述的库存目录分析的某些或所有的标准，试图开发一个量化的软件分级系统，它将被应用于现存程序并试图从中挑出再工程的候选者。

27.5 提出一种对纸和墨水或传统的电子文档的替代物，它可作为文档重构的基础。[提示：考虑新的能够用于传达软件的目的的描述技术。]

27.6 某些人相信人工智能技术将增加逆向工程过程的抽象层次，对此专题(即，AI 在逆向工程中的使用)进行研究并撰写一篇支持此论点的论文。

27.7 为什么当抽象层次增加时，完整性更难于达到？

27.8 为什么如果完整性增强时，交互性必须增加？

27.9 获取三个逆向工程工具的产品文献，并给出它们的特征。

27.10 对进程代数进行研究并使用在图 27-6 中描述的符号开发一个简单的过程规约。

27.11 在重构和正向工程之间存在细微的不同，是什么？

27.12 研究文献，发现一或二篇讨论从主机型到客户/服务器型再工程的实例研究，做一小结。

27.13 如何在 27.6 节给出的成本—收益模型中通过P_7确定P_4？

推荐阅读文献及其他信息源

再工程在软件工程领域是一个“热点”话题。因为构成再工程的技术继续在演化，技术期刊是最好的信息源。American Programmer 的 1995 年 6 月期，IEEE Computer 的 1995 年 1 月期，以及 Communications of the ACM 的 1994 年 5 月期是很多包含再工程话题特殊问题的期刊的代表。

涉及再工程的书籍尚不多，一个很好的开始点(涉及业务过程再工程)是 Hammer 和 Champy 的畅销书 [HAM93] .Arnold(Software Reengineering，IEEE Computer Society Press，1993)出版了一本关注软件再工程技术的重要论文的优秀文选；Berztiss(Software Methods for BusinessReengineering，Springer-Verlag，1996)以及 Spurr 及其同事(Software Assistance for BusinessReengineering，Wiley，1994)讨论了支持 BPR 的工具和技术；Aiken(Data Reverse Engineering，McGraw-Hill，1996)讨论了如何回收、重组织和复用组织化的数据。

更多的关于再工程的信息和全面的参考书目可在如下软件再工程网页找到：

http：//www.erg.abdn.ac.uk/users/brant/sre/

下面网址提供了附加的再工程信息和指针：

The Asset Repository

http：//source.asset.com

ESEG at UMCP

http：//www.cs.umd.edu/projects/SoftEng/tame

REDO Project Archive(Esprit)

http：//www.comlab.ox.ac.uk//archive/redo.html

Reverse Engineering-Geogia Tech

http：//www.cc.gatech.edu/reverse/

UCSD Software Evolution Group

http：//www-cse.ucsd.edu/users/wgg/swevolution.html

WWW Virtual Library-SE

http：//ricis.cl.uh.edu/virt-lib/software-eng.html

Reasoning Systems，Inc 建立了一个“再工程论文联机书目”：

http：//www.reasoning.com/papers.html

关于软件再工程的WWW参考文献最新列表可在http：//www.rspa.com找到。

① 当他的合作者所著的《再建工程的合作》[HAM93]一书卖的非常火的时候；Hammer的提议和影响已发生了戏剧化的变化。

① 同“重构引擎”——重构源代码的CASE工具—可以自动重构代码。

② 通常，在软件生命周期书写的说明永远不变。当做修改时，代码不再与说明相符合。

③ 面向对象概念的详细讨论参见第 19 章。

① 有时候过大的重构与重新开发是很难区分开来的。两者均为再建工程。

① 在继续之前请复习第 1 章，当“时代软件工厂”的责任是正确管理时，在许多公司中再建工程已成为一件常见工作。

第 28 章　客户/服务器软件工程

当开发新的基于计算机的系统时，工程师受到现存技术局限的限制，也在新技术提供了早期工程师不可及的能力时得到能力上的增强。在本世纪初，新一代能够保持紧密公差的机床的开发使得工程师能够设计称为大规模生产的新的工厂过程。在新的机床技术出现前，机床不能保持紧密公差。没有紧密公差，容易组装的可互换的零件——大规模生产的基石——不可能被建造。

分布式计算机体系结构的进展，使得系统和软件工程师能够为在组织中如何组织工作及如何处理信息开发新的途径。新的组织结构和新的信息处理方法(例如，决策支持系统、组件和成像)代表了对早期的主机型和基于微机的技术的根本背离，新的计算体系结构提供了使组织能够再工程它们的业务过程(第 27 章)的技术。

在本章，[①]我们考察一种占主导地位的对信息处理的新体系结构——客户/服务器(C/S)系统。客户/服务器系统的演化是同在桌面计算、新的存储技术、改善的网络通信、以及增强的数据库技术等方面的进展紧密联系的。本章的目标是给出客户/服务器系统的一个概述，重点强调当这样的C/S系统被分析、设计、测试和支持时，必须考虑的特殊的软件工程问题。

28.1 客户/服务器系统的结构

硬件、软件、数据库和网络通信技术一起为分布的和协作的计算机体系结构作出了贡献。以其最一般的形式，图 28-1 给出了一个分布的和协作的计算机体系结构。根系统，典型地是大型机，作为公司数据的中心库。根系统被连接到服务器(典型地是强大的工作站或 PC)，服务器具有双重角色。服务器更新和请求由根系统维护的公司数据，它们也维护局部的部门系统并在通过局域网(LAN)互连用户层 PC 方面扮演了关键角色。

在 C/S 结构中，位于另一个计算机上层的计算机称为服务器，而在下层的计算机称为客户机。客户请求服务，而服务器提供服务。然而，在图 28-2 所示的体系结构范围内，可以完成一系列不同的实现[ORF94]：

文件服务器。客户请求在某文件中的特定的记录，服务器通过网络传输这些记录到客户。

数据库服务器。客户发送结构化查询语言(SQL)请求到服务器，这些是作为消息在网络上传输。服务器处理 SQL 请求并找到请求的信息，仅将结果传送回客户。

事务服务器。客户发送在服务器端调用远程过程的请求，远程过程可以是一组 SQL 语句。当请求导致远程过程的执行并将结果传输回客户时，发生一个事务。

组件服务器。当服务器提供一组应用，它们使得通信能够在客户(以及使用它们的人员)间使用文本、图像、公告板、视频、以及其他表示进行，则存在一个组件体系结构。

28.1.1 C/S 系统的软件构件

不同于将软件视为在一台机器上实现的单个应用，适合于 C/S 体系结构的软件有几种不同的构件，它们可以被分配到客户或服务器，或在两者间分布：

用户交互/表示构件。该构件实现通常和图形用户界面(GUI)关联的所有功能。

应用构件。该构件在应用运作的领域的范围内实现被应用定义的需求。例如，某业务应用可能基于数值输入、计算、数据库信息和其他考虑生成一系列报表。某组件应用可能提供使得公告板通信或电子邮件得以进行的设施。在上述两种情形中，应用软件可以被划分，以使得某些构件驻留在客户机上，而另一些构件驻留在服务器上。

数据库管理。该构件执行应用请求的数据操纵和管理。数据操纵和管理可以简单到记录的传递，或复杂到高级 SQL 事务的处理。

除了这些构件外，另一种软件建造块，经常称为中间件，存在于所有 C/S 系统中。中间件由同时存在于客户和服务器的软件元素构成，并且包括网络操作系统的元素和用以支持数据库特定的应用、对象请求代理标准(28.1.5 节)、组件技术、通信管理、以及其他方便客户/服务器连接的特征等特殊应用软件的元素。Orfali [ORF94] 及其同事称中间件为“客户/服务器系统的神经系统”。

28.1.2 软件构件的分布

一旦已经确定客户/服务器应用的基本需求，软件工程师必须决定如何在客户和服务器间分布软件构件(在第 28.1.1 中讨论)。当和三类构件的每一种关联的大多数功能被分配给服务器，则得到一个“胖”服务器设计；相反，当客户实现了用户交互/表示、应用和数据库构件的大多数功能，则产生“胖”客户设计。

当实现文件服务器和数据库服务器体系结构时，经常遇到胖客户设计。在这种情形，服务器提供数据管理支持，但所有应用和 GUI 软件驻留在客户端。当实现事务和组件系统时，经常采用胖服务器设计，服务器提供对来自客户的事务和通信进行反应所需的应用支持，客户软件着重于 GUI 和通信管理。

胖客户和胖服务器可以用于举例说明一般的客户/服务器软件构件的分配方法，然而，一种更细粒度的软件构件分配方法定义了 5 种不同的配置：

分布式表示。在这种初步的客户/服务器方法中，数据库逻辑和应用逻辑保留在服务器端，典型地是主机(大型机)。服务器也包含准备屏幕信息的逻辑，使用如 CICS 之类的软件。使用特殊的基于 PC 的软件将来自服务器的基于字符的屏幕信息转换为 PC 上的 GUI 表示。

远程表示。在分布式表示方法的这种扩展中，主要的数据库和应用逻辑保留在服务器端，客户端使用服务器传送的数据准备用户表示。

分布式逻辑。客户端被赋予所有的用户表示任务及和数据输入(如，域级的确认、服务器查询陈述、服务器更新信息和请求)相关联的处理。服务器端被赋予数据库管理任务，和对客户查询、服务器文件更新、客户端版本控制和企业范围应用的处理。

远程数据管理。在服务器端的应用通过格式化从其他某处(例如，从某公司级来源)抽取到的数据创建新的数据源。分配给客户端的应用被用于开发利用由服务器格式化后的新数据。决策支持系统被包括在这种策略中。

分布式数据库。构成数据库的数据被散布在多个服务器和客户上，因此，客户端必须支持数据管理软件构件和应用以及 GUI 构件。

28.1.3 分布应用构件的指南

虽然没有绝对的规则来指导应用构件在客户端和服务器间的分布，下面指南是通常应该遵循的：

表示/交互构件通常放置在客户端。基于窗口的环境的可用性及对图形用户界面所必需的计算能力使得这样的分配是成本合算的。

如果数据库将被多个用 LAN 连接的用户所分享，则典型地数据库被放在服务器端。数据库管理系统和数据库访问能力也和物理数据库一起被放置在服务器端。

用于引用的静态数据应该分配到客户端。这样将数据放在和需要它们的用户相近的地方，从而减小不必要的网络通信及服务器的负荷。

对应用构件在客户端和服务器间分布的平衡是基于这样一个原则：该分布可以优化服务器和客户端的配置以及连接它们的网络。例如，相互排斥的关系的实现通常涉及查找数据库以确定是否存在一个可以匹配查找模式的参数的记录，如果没有找到匹配记录，则选用另一个查找模式。如果控制该查找模式的应用被完全地包含在服务器中，则网络交通被最小化。从客户端到服务器的第一次网络传输将同时包含主要的和次要的查找模式的参数，在服务器端的应用逻辑将确定该次要的查找模式是否是需要的。对客户端的回应消息将包含主要的或次要的查找所找到的记录。另一种可选的方法(客户端逻辑确定是否需要次要查找)将包括：第一次记录检索的消息，没有找到该记录的回应(通过网络)，包含第二次查找参数的第二个消息，带有检索结果的最终回应。如果第二次查找在 50％的情况下是需要的，那么，将评价第一次查找并在必要时初始化第二次查找的逻辑放在服务器上，将大大地减少网络交通。

关于构件分布的最终决策不仅仅应基于个体应用，还应该基于在系统上操作的应用的混合。例如，一个安装可能包含某一组需要广泛的 GUI 处理和很少的中心数据库处理的应用，这将导致在客户端使用强大的工作站和使用一个简单的服

务器。基于这样的配置，其他应用亦将支持胖客户方式，从而不需要提升服务器的能力。

应该注意，随着客户/服务器体系结构的使用趋于成熟，其趋势是将不稳定的应用逻辑放置在服务器端，这样当对应用逻辑进行修改时，可以简化对软件更新的部署［PAU95］。

28.1.4 连接 C/S 软件构件

一系列不同的机制被用于连接客户/服务器体系结构的各种构件。这些机制被结合进网络及操作系统结构中，对客户端的终端用户是透明的。最常见的连接机制有：

• 管道(pipe)——广泛用于基于 UNIX 的系统中，管道允许在运行不同操作系统的不同机器间传递消息。

• 远程过程调用(RPC)——允许一个进程去激活另一个驻留于不同机器上的进程或模块的执行。

• 客户/服务器 SQL 交互——用于从一个构件(典型地在客户端)传送 SQL 请求和相关的数据到另一个构件(典型地是在服务器上的 DBMS)，该机制仅限于 RDBMS 应用。

此外，客户/服务器软件构件的面向对象的实现导致“连接”使用对象请求代理，这将在下一节中讨论。

28.1.5 中间件和对象请求代理体系结构

在前面节中讨论的 C/S 软件构件被能够在单个机器内(或者是客户或者是服务器)或跨越网络相互交互的对象所实现。对象请求代理(ORB)是一个中间件构件，它使得一个驻留在客户端的对象可以发送消息到封装在驻留在服务器上的另一个对象中的方法。在本质上，ORB 截获消息并处理所有的通信和协调活动，这些活动是发现消息传送的目标对象、激活它的方法、传递合适的数据给该对象、以及传送结果数据给发送消息的源对象所必需的。

一种广泛使用的对象请求代理标准称为 CORBA，由对象管理组织 OMG 开发。CORBA 标准［MOW95］已在第 26 章简要讨论过，CORBA 的基本体系结构如图 28-3 所示。

当于客户/服务器系统中实现 CORBA 时，在客户和服务器两端的对象和对象类(第 19 章)均用接口描述语言(IDL)来定义，这是一种允许软件工程师定义对象、属性、方法和消息的说明型语言。为了适应客户端对象对服务器端方法的请

求，需创建客户和服务器的 IDL Stub，stubs 提供了一个通路，通过它们实现了对跨越客户和服务器系统的对象的请求。

因为对跨越网络的对象的请求随时发生，所以必须建立存储对象描述的机制，以使得关于对象及其位置的有关信息在需要时可以获得，接口池完成这项工作。

当某客户应用必须激活在系统中另一个地方的某对象所包含的方法时，CORBA 使用动态调用来(1)从接口池中获取关于希望的方法的有关信息；(2)用将传递给接收对象的参数来创建数据结构；(3)创建对接收对象的请求；(4)激活该请求。然后请求被传送给 ORB 核心(Core)——网络操作系统的管理请求的实现特定的部分——并完成请求。

请求被通过核心传送并被服务器处理。在服务器端，一个对象适配器(object adapter)存储类和对象信息到服务器端接口池中，接收和管理来自客户端的输入请求，并完成一系列的其他对象管理功能［ORF94］。在服务器端，IDL Stubs(类似于那些定义在客户机上的)被用作与驻留在服务器端的实际对象实现的接口。

对现代客户/服务器系统的软件开发是面向对象的。使用在本节简要描述的 CORBA 体系结构，软件开发者可以创建一个环境，对象可以在此环境中在一个大型网络范围内被复用。关于 CORBA 的进一步信息及其对 C/S 系统软件工程的整体影响，有兴趣的读者可以参阅文献［ORF96］和[MOW95］。

28.2 对 C/S 系统的软件工程

在第 2 章中引入了一系列不同的软件过程模型。虽然它们中任意一个均可修改以适用于 C/S 系统软件的开发，一个使用基于事件的和/或面向对象的软件工程方法的演化范型似乎是最为有效的。

客户/服务器系统被使用传统的软件工程活动——分析、设计、构造和测试——来开发，系统从一组一般的业务需求演化到一组确认过的、已经在客户和服务器机器上实现的软件构件。

28.3 分析建模问题

对 C/S 系统的需求活动和针对更传统的计算机体系结构的分析建模方法几乎没有不同，因此，在第 11 章讨论的基本分析原则以及在第 12 章和第 20 章讨论的分析建模方法同样适用于 C/S 系统。

因为分析建模避免了实现细节的规约，所以它仅被作为过渡用于设计同在客户机和服务器间分配软件构件的问题[①]。然而，因为演化的软件工程方法被应用到C/S系统，所以可能在早期的分析和设计迭代中确定关于整体C/S系统的实现决策(例如，胖客户或胖服务器)。

28.4 对 C/S 系统的设计

当软件的实现是采用特定的计算机体系结构开发的时，设计方法必须考虑特定的构造环境。在本质上，设计应该被定制以适应硬件体系结构。

当软件被设计用于采用客户/服务器体系结构的实现时，设计方法必须被“定制”以适应下列问题：

• 数据设计(第 14 章)是设计过程中最重要的工作。为了有效地使用关系型数据库管理系统(RDBMS)[②]或面向对象数据库管理系统(OODBMS)的能力，数据的设计变得比在传统应用中更加重要。

• 当选择使用事件驱动的范型时，应该进行行为建模(一种分析活动，第 12 章)，并且将行为模型中蕴含的面向控制的方面转化到设计模型中。

• C/S 系统的用户交互/表示构件实现了典型的和图形用户界面(GUI)关联的所有功能，因此，界面设计(第 15 章)是重要的。

• 设计的面向对象视图(第 21 章)经常被选用。替代了过程型语言提供的顺序结构，通过在事件(在 GUI 处启动)和事件处理函数(在基于客户的软件内)间的链接，提供对象结构。

虽然关于C/S系统的最好的分析和设计方法的争论仍在继续，面向对象方法(第 20、21 章)似乎是最好的选择。然而，传统方法(第 12、14 章)也可以采用，对分析和设计的传统符号包括数据流图(DFD)、实体关系图(ERD)和结构图。[③]

28.4.1 传统设计方法

在客户/服务器系统中，DFD 可用于建立系统的范围，标识高层的功能和主题数据区域(数据存储)，并允许高层功能的分解。然而，和传统的 DFD 方法不同的是，分解停留在基本业务加工层次，而不是继续到原子加工层次。

在 C/S 语境内，基本业务加工(EBP)可以定义为可被客户端的一个用户不中断地完成的一组任务，这些任务或者被完整地完成，或者一点也不做。

ERD 也承担了扩张的角色，它连续地被用于分解 DFD 中的主题数据区域(数据存储)，从而建立高层的可用 RDBMS 实现的数据库视图。它的新作用是提供定义高层业务对象的结构(28.4.2)。

不是作为功能分解的工具，结构图现在被用作组装图，以显示涉及到某基本业务加工的解决方案中构件，这些构件，包括界面对象、应用对象和数据库对象，确立了数据的处理方式。

28.4.2 数据库设计

数据库设计被用于定义、然后刻划在客户/服务器系统中的业务对象的结构。标识业务对象所需的分析是使用在第 10 章讨论的信息工程方法来完成的，传统的分析建模符号(第 12 章)，如 ERD，可用于定义业务对象，但是，应该建立一个数据中心库，以捕获那些不能用 ERD 这类的图形符号完全表述的附加信息。

在这个中心库中，业务对象被定义为对系统的购买者和用户(而不是它的实现者)可见的信息。每个在 ERD 中标识的对象(实体)在设计过程中被扩展为：一个数据结构(如，一个文件及其相关域)；管理文件的所有定义；文件记录中的数据项间的关系；这些关系的确认规则；以及刻划针对数据的处理的外部视图的业务规则。

在数据库设计中必须开发大量的设计信息，这些信息用关系数据库实现，可以在如图 28.4 [POR94] 中的设计中心库中保存，个体表(图的左边)被用于定义下面的客户/服务器数据库的设计信息：

- 实体——在新系统的 ERD 中被标识。
- 文件——实现在 ERD 中标识的实体。
- 文件与域关系——通过标识哪个域被包含在哪个文件中而建立文件的布局。
- 域——定义在设计(数据字典)中的域。
- 文件与文件关系——标识可以被联结以创建逻辑视图或查询的相关文件。
- 关系确认——标识被用于确认的文件—文件或文件—域关系的类型。
- 域类型——用于允许从域超类(如，日期、正文、数字、值、价格)中继承域特征。
- 数据类型——包含在域中的数据的特征。
- 文件类型——用于标识文件的位置。
- 域功能——键、外键、属性、虚域、导出域等。
- 允许值——为 status 类型的域标识允许值。
- 业务规则——编辑、计算导出域规则等。

当 C/S 体系结构变得更为普及时，分布数据管理的趋势也开始加速。在运用了该方法的 C/S 系统中，数据管理构件同时驻留在客户端和服务器端。在数据库

设计的语境内，一个关键的问题是数据分布，即，如何在客户和服务器间分布数据，及如何跨网络节点分散数据？

关系型数据库系统(RDBMS)通过结构化查询语言(SQL)使得可容易地访问分布数据。在C/S体系结构中SQL的优点是：它是“非导航的”[BER92]。在RDBMS中，用SQL刻划数据的类型，但是，不需要任何导航性的信息。当然，这意味着RDBMS必须足够高级，以维护所有数据的位置并能够定义通往它的最好路径。在欠高级的数据库系统中，对数据的请求必须指出将访问什么以及它在什么地方。如果应用软件必须保持导航性信息，则对C/S系统来说，数据管理变得有过复杂。

应该注意，设计者可采用其他数据分布和管理技术[BER92]：

手工抽取。允许用户手工地从服务器拷贝适当的数据到客户端。当用户需要静态数据及抽取的控制权可以交给用户时可采用该方法。

快照(snapshot)。该技术通过刻划将按预定义的间隔从服务器向客户端传递的数据的“快照”自动完成手工抽取过程。这种方法可用于分布相对静态的、仅需要不频繁的更新的数据。

复制(replication)。当必须在不同位置(如，不同的服务器或客户和服务器)维护数据的多个拷贝时可使用该技术。这里，复杂性的层次逐步上升，因为必须在多个位置协调数据一致性、更新、安全和处理等问题。

分割(fragmentation)。在这个方法中，系统数据库被在多个机器间分割。虽然在理论上是诱人的，但是，分割的实现是极其困难的，并不会经常遇到。

数据库设计，或者更特定的，C/S系统的数据库设计已超出了本书范围，有兴趣的读者可参见文献[BRO91]、[BER92]、[VAS93]和[ORF94]，以获取进一步的讨论。

28.4.3 某设计方法的概述

Poter[POR95]提出了一组设计基本业务加工的步骤，它组合了传统设计方法和面向对象设计方法中的某些元素。它假定定义业务对象的需求模型已经在基本业务加工的设计开始前被开发并精化。然后，运用下面步骤去导出设计：

1.对每个基本业务加工，标识被创建、更新、引用或删除的文件。

2.使用在第一步中标识的文件作为定义构件或对象的基础。

3.对每个构件，检索业务规则和其他对有关文件已经建立的业务对象信息。

4.确定哪些规则是和加工相关的，分解这些规则到方法层次。

5.当需要时，定义任意对实现方法所需的附加的构件。

Porter[POR95]提出了一种专门的结构图符号体系(图 28－5)来表示基本业务加工的构件结构，然而，使用了不同的象征符号，以使该图符合 C/S 软件的面向对象性质。在图中，有 5 种不同的符号：

界面对象。这种类型的构件也称为用户交互/表示构件，典型地是建立在单个文件之上或单个文件及其相关文件(它们通过查询而被联结)之上。它包括格式化 GUI 界面和与界面上的控制相关联的客户端应用逻辑的方法，它也包括内嵌的 SQL 语句，这些语句刻划了在主要文件(界面建立在之上)上完成的数据库处理。如果在正常情况下和界面对象关联的应用逻辑被代之以在服务器端实现，典型地，通过中间件工具的使用，则在服务器上运行的应用逻辑将被标识为单独的应用对象。

数据库对象。这种类型的构件被用于标识如基于某文件(该文件不是界面对象被建立在其之上的主要文件)上的记录创建或选择之类的数据库处理。应该注意，如果主要文件(用户界面被建立在其上)被以不同的方式处理，那么，第二组 SQL 语句可用于在另一个序列中检索某文件。这第二种文件处理技术应该被在结构图上作为单独的数据库对象而被单独地标识出来。

应用对象。被界面对象或者数据库对象使用，该类构件被数据库触发器或者远程过程调用所激活。它也被用于标识为了运行已被移到服务器端的业务逻辑(在正常情况下是和界面处理相关联的)。

数据耦合。当某对象激活另一个独立的对象时，一个消息(第 19 章)在两个对象间传递，数据耦合符号(图 28-5)被用于表示这种情形。

控制耦合。当某对象激活另一个独立的对象，并且二者间没有数据传送时，则使用控制耦合符号。

28.4.4 加工设计的迭代

用于表示业务对象的设计中心库(28.4.2 节)也可以用于表示界面、应用和数据库对象(见图 28－4 右手边)。查阅图 28－4，注意到存在以下实体：

• 方法——描述某业务规则将被如何实现。

• 基本加工——定义在分析模型中标识的基本业务加工。

• 加工/构件链——类似于制造业中的材料单，该表标识了构成基本业务加工的解决方案的构件。必须注意，这个链接技术允许某给定的构件被包括在多个基本业务加工的解决方案中。

• 构件——描述显示在结构图上的构件。

• 业务规则/构件链——标识对某给定的业务规则的实现有重要意义的构件。

如果已用 RDBMS 实现了在图 28－4 中描述的那种类型的中心库，设计者将拥有了一种有用的设计工具，它可提供报告以帮助 C/S 系统的构造及其未来维护。

28.5 测试问题[①]

客户/服务器系统的分布性质对软件测试者带来了一些独特的问题，Binder[BIN92]提出了如下问题：

• 客户端 GUI 的考虑。

• 目标环境及平台多样性的考虑。

• 分布数据库的考虑(包括复制的数据)。

• 分布处理的考虑(包括复制的处理)。

• 非鲁棒的目标环境。

• 非线性的性能关系。

必须以允许强调上面的每个问题的方式设计和 C/S 测试关联的策略和战术。

28.5.1 整体 C/S 测试策略

通常，客户/服务器软件的测试发生在三个不同的层次：(1)个体的客户端应用以“分离的”模式被测试——不考虑服务器和底层网络的运行；(2)客户端软件和关联的服务器端应用被一起测试，但网络运行不被明显的考虑；(3)完整的 C/S 体系结构，包括网络运行和性能，被测试。

虽然在上面的每个层次有很多不同类型的测试被进行，下面的测试方法是 C/S 应用中经常遇到的：

应用功能测试。用本书中前面讨论的方法测试客户端应用的功能。在本质上，应用被独立的测试，以揭示在其运行中的错误。

服务器测试。测试服务器的协调和数据管理功能，也考虑服务器性能(整体反应时间和数据吞吐量)。

数据库测试。测试服务器存储的数据的精确性和完整性，检查客户端应用提交的事务，以保证数据被适当地存储、更新和检索。也测试归档功能。

事务测试。创建一系列的测试以保证每类事务被按照需求处理。测试着重于处理的正确性，也关注性能问题(如，事务处理时间和事务量测试)。

网络通信测试。这些测试验证网络节点间的通信正确地发生，并且消息传递、事务和相关的网络交通无错地发生。网络安全性测试也可能作为此测试的一部分。

为了完成这些测试，Musa[MUS93]提出了从客户/服务器用户场景导出的运行轮廓(operational profile)的开发，运行轮廓指出了不同类型的用户如何和 C/S 系统交互操作，即，轮廓提供一种“使用的模式(pattern of usage)”，它可被用于在测试的设计和执行中。例如，对某特定类型的用户，各类事务(查询、更新和命令)的百分比是多少?

为了开发运行轮廓，必须导出一组用户场景[BIN95]，每个场景强调谁、哪里、什么和为什么，即，用户是谁、系统交互在哪里发生(在物理的 C/S 体系结构中)、事务是什么、以及它为什么发生。场景可以在 FAST 会议(第 11 章)中或通过和终端用户的非正式讨论而导出，然而，结果应该是一样的。每个场景应该指出：为特定用户提供服务所需的系统功能，这些功能被需要的顺序，所期望的定时和反应，以及每个功能被使用的频率。这些数据然后被组合(对所有用户)，以创建运行轮廓。

对 C/S 体系结构的测试策略类似于在第 17 章描述的对基于软件的系统的测试策略。测试从小型测试开始，即，测试单个客户端应用，然后，客户端、服务器和网络的集成被逐步测试，最后，完整的系统被作为一个运行实体测试。

传统的测试将模块/子系统/系统集成和测试视为自顶向下、自底向上或二者的某种变体。在 C/S 开发中的模块集成可能具有某些自顶向下或自底向上的成分，但是，在 C/S 项目中的集成更趋向于的跨所有设计层次模块的并行开发和集成，这样，在 C/S 项目中的集成测试有时最好使用非增量式或“大爆炸”式的方法来完成。

系统不是被建造去使用预先指定的硬件和软件这一事实影响系统测试。C/S 系统的网络的跨平台性质需要我们的对配置测试和兼容性测试给以更多的关注。

配置测试强调在所有已知的、可能运行于其中的硬件和软件环境中进行系统测试。兼容性测试保证跨硬件和软件平台上功能一致的界面。例如，窗口类型的界面可能根据实现环境而呈现视觉上的不同，但是，同样的基本用户行为应产生同样的结果，而不管客户端界面是 IBM 的 OS/2 表示管理器、微软的 Windows、苹果公司的 Macintosh 或 OSF 的 Motif。GartnerGroup[GAR93]建议了一种客户/服务器测试计划，其大纲如表 28-1 所示。

表 28－1 基于 GartnerGroup 的建议的修订的客户/服务器测试计划

1.0 窗口(GUI)测试	4.4 代码质量
1.1 业务场景标识	4.5 测试工具
1.2 测试用例创建	5.0 功能测试
1.3 验证	5.1 定义
1.4 测试工具	5.2 测试数据创建
2.0 服务器	5.3 验证
2.1 测试数据创建	5.4 测试工具
2.2 大量	压力测试
2.3 验证	6.1 定义
2.4 测试工具	6.2 可用性测试
3.0 连通性	6.3 用户满意程度调查
3.1 性能	6.4 验证
3.2 大量	压力测试
3.3 验证	7.0 测试管理
3.4 测试工具	7.1 测试队伍
4.0 技术质量	7.2 测试进度
4.1 定义	7.3 所需资源
4.2 缺陷标识	7.4 测试分析、报告和跟踪管理
4.3 度量	

28.5.2 C/S 测试策略

即使 C/S 系统没有采用面向对象技术实现，面向对象测试技术(第 22 章)还是有意义的，因为复制的数据和处理可以被组织到共享同一组性质的对象类中。一旦为某对象类(或它们在用传统方法开发的系统中的等价体)已经导出测试用例，那些测试用例应该可广泛地用于该类的所有实例。

当考虑现代 C/S 系统的图形用户界面时，OO 观点是特别有价值的。GUI 是天生面向对象的，并且不同于传统的界面，因为它必须运行于多个平台上。此外，测试必须探索大量的逻辑路径，因为 GUI 创建、操纵和修改大量的图形对象。因为对象可能存在或不存在，它们可能存在一个较长的时间段，已及它们可能出现在桌面的任何地方，这使得测试更加复杂。

这意味着测试传统的基于字符的界面的传统的捕获/回放技术必须被修改，以便于处理 GUI 环境的复杂性。捕获/回放范型的一种功能变体称为结构化捕获/回放[FAR93]，是针对 GUI 测试的演化。

传统的捕获/回放将输入记录为击键、输出记录为屏幕图像，它们被存放以和后续测试的输入和输出图象进行比较。结构化捕获/回放是基于对外部活动的内部(逻辑)视图，应用程序和 GUI 的交互被记录为内部事件，它们可以存放为用微软的 Visual Basic、某种 C 变体、或厂商自己的语言书写的“脚本”。一系

列有用的工具(如参考文献[HAY93]、[QUI93]和[FAR93])已经被开发出来以支持这种测试方法。

测试 GUI 的工具没有强调传统的数据确认或路径测试需求，在第 16 章讨论的黑盒和白盒测试方法可用于很多情形，在第 22 章讨论的特殊的面向对象策略对客户端和服务器端都是适用的。

28.6 小结

虽然客户/服务器系统可以采用一个或多个软件过程模型以及很多在本书前面部分讨论的分析、设计和测试技术，但 C/S 的特殊的体系结构特征需要对这些软件工程方法进行定制调态。通常，应用于 C/S 系统的软件过程模型在本质上是演化型的，并且技术方法经常倾向面向对象的方法。开发者必须描述对象，以得到用户交互/表示、数据库和应用构件的实现。为这些构件定义的对象必须被分配在客户端或服务器端，并且可以通过对象请求代理来连接。

对象请求代理体系结构支持 C/S 设计，其中客户端对象向服务器端对象发送消息。CORBA 标准使用接口定义语言，接口池管理对象的请求而不管它们在网络上的位置。

对客户/服务器系统的分析和设计使用数据流图和实体关系图、修改的结构图、以及其他在传统应用开发中遇到的符号体系。测试策略必须被修改以适应对网络通信及对驻留在客户和服务器端的软件间的相互作用的测试。

参考文献

[BER92] Berson，A.，Client/Server Architecture，McGraw-Hill，1992.

[BIN92] Binder，R.，“A CASE-Based Systems Engineering Approach to Client-ServerDevelopment，”CASE Trends， 1992.

[BIN95] Binder，R.，“Scenario-BasedTesting for Client Server Systems，”SoftwareDevelopment，vol.3，no.8，August 1995，pp.43-49.

[BRO91] Brown，A.W.，Object-Oriented Databases，McGraw-Hill，1991.

[FAR93] Farley，K.J.，“Software Testing For Windows Developers，”Data Based Advisor，November 1993， pp.45-46， 50-52.

[GAR93] The GartnerGroup，conference presentation，1993.

[HAY93] Hayes，L.G.，“Automated Testing For Everyone，”OS/2 Professional，November1993，p.51.

[MOS96] Mosley,D.,“Managing Software Testing in the Client/Server Milieu,”(in press)[MOW95] Mowbray and Zahavi, The Essential CORBA, Wiley, 1995.

[MUS93] Musa, J., “Operational Profiles in Software Reliability Engineering,”IEEESoftware, March 1993, pp.14-32.

[ORF94] Orfali,R.,D.Harkey,and J.Edwards,Essential Client/Server Survival Guide, Wiley, 1994.

[ORF96] Orfali, R., D.Harkey, and J. Edwards, The Essential Distributed Objects SurvivalGuide, Wiley, 1996.

[PAU95] Paul, L.G., “Client/Server Deployment,”Computerworld, December 18, 1995.

[POR94] Porter,J.,O-DES Design Manual,Fairfield University,1994.

[POR95] Porter, J., Synon Developer's Guide, McGraw-Hill, 1995.

[QUI93] Quinn, S.R., J.C.Ware, and J.Spragens, “Tireless Testers; Automated Tools CanHelp Iron Out the Kinks in Your Custom GUI Applications,”Infoworld, September 1993, pp.78-79, 82-83, 85.

[VAS93] Vaskevitch, D., Client/Server Strategies, IDG Books, 1993.

思考题

28.1 使用出版物或 Internet 资源作为背景信息，定义一组评价 C/S 软件工程工具的标准。28.2 举出 5 个应用例子，其中胖服务器似乎是合适的设计策略。

28.3 举出 5 个应用例子，其中胖客户端似乎是合适的设计策略。

28.4 对 CORBA 标准作进一步研究，确定标准的最新版本如何强调在不同厂商提供的不同 ORB 间的互操作性。

28.5 研究结构化查询语言(SQL)，给出一个简略的如何用该语言刻划事务的例子。

28.6 研究在组件(groupware)方面的最新进展，并在班上作一个简略的报告。老师可以将不同的功能分配给不同的报告者。

28.7 某公司正在建立一个新的编目销售部门，以销售休闲服装和野外用品。产品编目将 WWW 上发布，可以通过 email、Web、电话或传真订货。将建立一个客户/服务器系统以支持在公司网址的订货处理。定义一组对订货处理系统所必

需的高层对象，并将这些对象组织为三个构件范畴：用户交互/表示、数据库和应用。

28.8 针对思考题 28.7 中描述的系统，建立关于如果通过信用卡付帐时何时可以供货的业务规则。再考虑通过支票付帐时的业务规则。

28.9 开发一个状态变迁图(第 12 章)，它定义对在编目销售部门内客户端 PC 上工作订单输入职员可见的事件和状态。(思考题 28.7)

28.10 给出三或四个消息例子，它们可能导致从客户端对服务器端的某方法的请求。

推荐阅读文献及其他信息源

虽然对客户/服务器体系结构的基本分析和设计方法和更传统的系统是很相似的，但必须引入 C/S 特定的知识。Orfali 及其同事[ORF94]撰写了一部针对该技术的更易读的导论，Watterson(Client/Server Technology for Managers，Addison－Wesly，1995)给出了对涉及的技术和早期系统的应用焦点的导论性概述，在更详细的层次， Nance(Introduction to Networking，QueCorporation，1994)和 Dewire(Client/Server Computing，McGraw－Hill，1993)对技术问题进行了全面讨论。

Tech－Ease(Client－Server Computing，Prentice－Hall，1996)给出了对 C/S 系统和体系结构的一般性导论。下列书籍也是值得参阅的：

Berson，A.，Client/Server Architecture，2nd edition，McGraw-Hill，1996.

Goglia， R.A.，Testing Client/Server Applications，Wiley，1993.

Inmon，W.H.，Developing Client/Server Applications，Wiley，1994.

Koelmel， R.L.，Implementing Application Solutions in a Client/Server Environment，Wiley， 1995.

Spohn，D.L.，Data Network Design，2nd edition，McGraw-Hill，1996.

因为客户/服务器技术演化得如此之快，产业期刊和电子信息资源可能是当前最好的信息源。对新闻组 comp.client－server 的 FAQ 可在下面网址找到：

http：//www.abs.net/～lloyd/csfaq.txt

或

ftp：//rtfm.mit.edu/pub/usenet/news.answers/client-server-faq

最新 CORBA 标准的信息位于：

http：//www.omg.org

对 C/S 系统的测试的讨论可在下面网址找到：

http ：//www.icon－stl.net/～djmosley

客户/服务器咖啡屋(Coffeehouse)提供了关于在试图实现 C/S 技术的专业人员间的 Q&A 的论坛：

http：//www.onr.com/oz/house.html

关于客户/服务器软件工程的最新WWW文献列表可在http：//www.rspa.com找到。

① 本章的部分内容引自JohnPorter在FairfieldUniversity的EI工程学院提出的客户/服务器环境的教课材料，已获引用权。

① 例如，顺应CORA的CS结构（见 28.1.5）对设计和决策的确定有较大的影响。

② 在C/S体系结构中已大量使用了RDMS和面向对象数据库管理系统。

③ 结构图的详细讨论在第 12 章。

① 该节是DanielMosley[MAS96]所著相关内容的缩略版，引用已获作者的许可。

第 29 章 计算机辅助软件工程

每个人都听到过鞋匠孩子的故事：鞋匠总是忙于给其他人做鞋，他自己的孩子却没有鞋穿。在过去 20 年，很多软件工程师便是“鞋匠的孩子”。虽然这些技术专业人员建造了使其他人的工作自动化的复杂系统，但他们自己却几乎很少使用自动化。事实上，直到最近，软件工程师还基本上是手工在开展工作，其中仅仅在过程的后期阶段才有工具可使用[①](大多是编译器和编辑器)。

今天，软件工程师终于得到了他们的第一双鞋——计算机辅助软件工程(CASE)。这些鞋并没有呈现它们应有的多样性，经常有些僵硬并且有时不舒适，不能为那些时髦者提供足够的满意度，以及和软件开发者使用的其他“衣服”不相匹配。但是，它们为软件工程师的行头提供了一个绝对基本的“服饰”，并且，随着时间的推移，将变得更为舒适、更为可用、更适应单个实践者的需要。

在本书前面章节，我们试图让大家对软件工程技术的基础有合理的理解。在本章，关注点被移向帮助实现软件过程自动化的工具和环境。

29.1 什么是 CASE?

对任何工匠——技工、木匠或软件工程师——而言，最好的工作间具有三个基本的特征：(1)一组有用的工具，可给建造产品的每个步骤提供帮助；(2)一个

组织的很好的布局，使得能够快速地找到工具，并高效地使用它；(3)一个熟练的工匠，他知道如何以有效的方式去使用这些工具。软件工程师现在认识到他们需要更多的、各式各样的工具(仅仅是手边的工具将不能满足现代基于计算机的系统的需要)，并且他们需要一个组织的很好的和高效的工作车间来放置工具。

软件过程的车间称为集成的项目支撑环境(Integrated project support environment)(本章后面将讨论)，放置在车间中的工具称为计算机辅助软件工程(CASE)工具。

CASE 工具加入到软件工程师的工具箱中，CASE 为软件工程师提供了将手工活动自动化的能力，并改善工程师的洞察力。和在其他行业中的工程师使用的计算机辅助工程和设计工具一样，CASE 工具帮助保证完成产品建造前及建造中的质量的设计。

29.2 构造 CASE 的积木块

计算机辅助软件工程可能简单到一个单个工具，它支持某特定的软件工程活动，或复杂到一个完整的环境，它包含工具、数据库、人员、硬件、网络、操作系统、标准以及无数的其他部件。CASE 的构造积木块如图 29－1 所示，每个构造积木块形成下一个的基础，而工具位于构造积木块堆的顶部。有趣的是，我们可看到，对有效的 CASE 环境的基础几乎很少涉及软件工程工具本身，相反，成功的软件工程环境是建造在包含适当的硬件和系统软件的环境体系结构之上。此外，环境体系结构必须考虑在软件工程过程中应用的人员的工作模式。

环境体系结构，由硬件平台和操作系统支持(包括网络和数据库管理软件)构成，铺设了 CASE 的基石，但是，CASE 环境本身要求其他的构造积木块。一组可移植服务提供了 CASE 工具及其集成框架与环境体系结构间的连接桥梁。集成框架是一组专用程序，它们使得单个的 CASE 工具可以和其他工具相互通信，能够创建项目数据库，以及使终端用户(软件工程师)看到同样的软件界面。可移植服务允许 CASE 工具及其集成框架能够跨越不同的硬件平台和操作系统使用，而不需要大量的相应修改。

图 29－1 中描述的构造积木块表示了 CASE 工具集成的整个基础，然而，今天使用的大多数 CASE 工具并没有使用上面讨论的所有构造积木块来构造。事实上，某些 CASE 工具保持“点解决方案”，即工具被用于辅助某个特定的软件工程活动(如，分析建模)，但并不直接和其他工具通信，也不关联到一个项目数据库，而且不是某集成 CASE 环境(I－CASE)的一部分。虽然这种方案不很理想，但 CASE 工具仍可以被有效地使用，即使它只提供点解决方案。

CASE 集成的相对层次如图 29－2 所示，在集成体系的低端是单个的(点解决方案)工具；当单个工具提供了数据交换(必须这样做)的设施，则集成层次被少许改善，这样的工具以标准格式产生输出，可以和其他能够读该格式的工具相兼容；在某些情况下，需要使用一些互补的 CASE 工具，以形成工具间的连接(例如，分析和设计工具被结合成一个代码生成器)。使用这种方法，在工具间的协作可

以生产出单独使用其中某个工具难于得到的终端产品；单源集成发生在当某单个CASE 工具厂商集成一系列不同的工具并将它们作为工具包销售的情况下，虽然这种方法是相当有效的，但是，大多数单源环境的封闭体系结构使得来自其他厂商的工具难于加入。

在集成体系的高端是集成项目支撑环境(IPSE)，在此环境中会建立上面所描述的每个构造积木块的标准，CASE 工具厂商使用这些 IPSE 标准来建造工具，它们将和 IPSE 兼容，并因此和其他工具兼容。

29.3 CASE工具分类 ①

当我们试图对 CASE 工具分类时，总存在一系列固有的风险。例如有一个潜在的意思是：为了创建一个有效的 CASE 环境，必须所有类型的工具均存在——但这并不是真实情况。当我们将某工具划入某类，而其他人到可能认为它应该属于另一类时，彼此就可能产生混乱(或对抗)。某些读者可能感到没有一个完全的工具分类——因此，一组完全的工具被包含在整个 CASE 环境中的可能被排除了。此外，简单的分类又难于说明问题——即我们没有显示工具的层次交互或工具间的关系。即使有这些风险存在，也有必要给出一个 CASE 工具的分类——以便更好地理解 CASE 的广度，以及更好地认识到这些工具可以应用到软件过程中的何处。

对 CASE 工具的分类可以根据功能、根据它们被作为管理工具还是技术工具的角色、根据它们在软件工程过程各个步骤中的使用、根据支撑它们的环境体系结构(硬件和软件)、或甚至根据它们的起源或价格[QED89]来划分，下面主要按使用功能的标准来进行分类：

信息工程工具。通过对某组织的战略性信息需求的建模，信息工程工具提供了一个可从中导出特定信息系统的“元模型”。这些 CASE 工具不是关注于特定应用的需求，而是对业务信息在公司内各个组织实体间的流动进行建模。这类工具的主要目标是表示业务数据对象、它们的关系、以及这些数据对象如何在公司内部的不同业务区域间流动。

过程建模和管理工具。如果某组织试图改善其业务(或软件)过程，它必须首先理解它。过程建模工具(也称“过程技术”工具)被用于表示过程中的关键元素，使得过程能够被更好地理解。这样的工具也可以提供过程描述的链接，它可以帮助那些需要知道过程的人们去理解要完成过程所必需的任务。此外，过程管理工具可以提供其他支持过程活动定义的工具的链接。

项目计划工具。这类工具关注两个主要区域：软件项目工作量和成本估算、以及项目进度安排。估算工具计算估计工作量、项目工期、以及推荐的人员数量(使用第 5 章引入的一个或多个技术)。项目进度安排工具使得管理者能够定义所有项目任务(工作分解结构)，创建任务网(通常使用图形输入)，表示任务间依赖性，以及模拟对项目可能的并行度(第 7 章)。

风险分析工具。标识潜在的风险，并开发一个计划去减轻、监控和管理它们对大型项目是极为重要的。风险分析工具通过提供对风险标识和分析的详细指南，而使得项目管理者能够建立风险表(第 6 章)。

项目管理工具。项目进度和项目计划必须在连续的基础上进行跟踪和监控，此外，管理者应该使用工具来收集度量数据，从而最终为软件产品的质量提供指示。这类工具通常是项目计划工具的扩展。

需求跟踪工具。当开发大型系统时，被交付的系统经常不能满足客户指定的需求。需求跟踪工具的目标是从客户的建议请求(RFP)开始，提供系统化的孤立需求的方法。典型的需求跟踪工具将用户交互的文本评估和数据库管理系统结合起来，数据库中存储并分类每个从初始的 RFP 或规约中“分析”得到的系统需求。

度量和管理工具。软件度量改善管理者的控制和协调软件过程的能力，以及开发者提高所生产的软件的质量的能力。当前的度量和测度工具着重于过程、项目和产品特征。面向管理的工具捕获项目特定的度量(如，每人月的 LOC、每个功能点的缺陷数等)，它们对生产率或质量提供整体的指示。面向技术的工具确定技术度量(第 18 和 23 章)，它们为设计或代码的质量提供更多的洞察。很多更高级的度量工具维护一个“产业平均”测度的数据库，基于用户提供的项目和产品特征，这样的工具根据产业平均(和过去的本地性能)“评估”本地数值，并提出改善策略。

文档工具。文档生成及桌面出版工具涉及了几乎软件工程的每个方面，并为有软件开发者提供了实质性的“效力”机会。大多数软件开发组织花费大量时间来开发文档，在很多情况下，文档开发过程本身是相当低效的。软件开发组织花费占总开发工作量的 20%到 30%的工作量来完成文档是常见的事。为此，文档工具提供了重要的改善生产率的机会。

系统软件工具。CASE 是一种工作站技术，因此，CASE 环境必须适应高质量的网络系统软件、电子邮件、公告牌和其他的通信能力。

质量保证工具。大多数宣称关注质量保证的 CASE 工具实际上是审计源代码以确定语言标准符合度的度量工具。其他工具抽取技术度量(第 18 章)，以努力规划被建造的软件的质量。

数据库管理工具。数据库管理软件是建立 CASE 数据库(中心库)的基础，我们也将中心库称为项目数据库(第 9 章)。出于对配置对象的强调，对 CASE 的数据库管理工具可能从关系型数据库管理系统(RDBMS)演化到面向对象数据库管理系统(OODBMS)。

软件配置管理工具。软件配置管理(SCM)位于每个 CASE 环境的核心，在所有 5 个主要的 SCM 任务——标识、版本控制、变化控制、审计和状况说明和报告——等方面均可以使用工具来辅助完成。CASE 数据库提供了标识每个配置项并将其和其他项关联的机制；在第 9 章讨论的控制过程可在特殊工具的辅助下实现；对

单个配置项的轻易访问方便了审计过程；CASE 通信工具可以大大改善状况说明和报告(报告关于变化的信息给所有需要知道的用户)。

分析和设计工具。分析和设计工具使得软件工程师能够创建将要建造的系统的模型，模型包含数据、功能和行为的表示(在分析层次)，以及数据的、体系结构的、过程的和界面的设计特征。通过进行模型的一致性和合法性检查，分析和设计工具为软件工程师提供对分析表示的某种程度的深入洞察，并帮助在错误传播到设计之前，或更糟糕地，传播进已实现的软件体之前，删除错误。

PRO/SIM 工具。PRO/SIM(原型和仿真)工具[NIC90]为软件工程师提供了在实时系统被建造前预测系统行为的能力，此外，它们使得软件工程师能够开发实时系统的模仿版本，允许客户在实际的实现完成前对其功能、操作和响应有深入理解。

界面设计和开发工具。界面设计和开发工具实际上是如菜单、按钮、窗口结构、图符、滚动机制、设备驱动器等程序构件的一个工具箱，然而，这些工具箱正在被界面原型工具所替代，这种原型工具能够帮助在屏幕上快速地创建遵从当前软件采用的界面标准的现代用户界面。

原型工具。可以使用一系列不同的原型工具。屏幕画笔使得软件工程师能够为交互式应用快速地定义屏幕布局。更高级的 CASE 原型工具使得能够结合屏幕和报表的布局进行数据设计的创建。很多分析和设计工具具有提供原型选项的扩展。PRO/SIM 工具为工程(实时)应用生成 Ada 和 C 源代码骨架。最后，一系列第四代语言工具具有原型工具的特征。

编程工具。编程类工具包括编译器、编辑器和调试器，它们对大多数传统程序设计语言是可用的，此外，面向对象程序设计环境、第四代语言、图形程序设计环境、应用生成器、以及数据库查询语言也属于本类。

集成和测试工具。在他们的软件测试工具的目录内，《软件质量工程》[SQE95]一书定义了下面的测试工具范畴：

- 数据获取——用于获取在测试中将被使用到的数据的工具
- 静态测度——分析源代码，但不执行测试用例的工具
- 动态测度——在执行中分析源代码的工具
- 仿真——仿真硬件或其他外部环境功能的工具
- 测试管理——辅助测试的计划、开发和控制的工具
- 交叉功能工具——跨越上面类别的边界工具

应该注意的是，很多测试工具具有跨越上面类别的两个或更多的特性。

静态分析工具。静态分析工具辅助软件工程师导出测试用例。在产业中有三种不同类型的静态测试工具被使用：基于代码的测试工具、专门的测试语言和基于需求的测试工具。基于代码的测试工具接收源代码(或 PDL)作为输入，完成一系列的分析，导出测试用例的生成。专门的测试语言(如 ATLAS)使得软件工程师可以书写详细的测试规约，描述每个测试用例和它们的执行逻辑。基于需求的测试工具孤立特定的用户需求，并建议针对需求的测试用例(或测试的类)。

动态分析工具。动态测试工具与执行中的程序之间的交互，检查路径覆盖率、测试关于特定变量的值、以及插装程序的执行流。动态工具可以是或者插装的，或者非插装的。插装的工具通过插入完成上面提到的活动的探针(额外的指令)来改变将被测试的软件。非插装的工具使用和包含被测试程序的处理器并行运行的单独的硬件处理器。

测试管理工具。测试管理工具用于控制和协调每个主要测试步骤(第 17 章)的软件测试。这类工具管理和协调回归测试，进行在实际值和期望输出间的差异的比较，以及管理具有交互式人机界面的程序的成批测试。除了上面提到的功能外，很多测试管理工具也作为类属的测试驱动器，测试驱动器从测试文件中读取一个或多个测试用例、格式化测试数据以符合被测试软件的需要、然后激活将被测试的软件。

客户/服务器测试工具。C/S 环境要求专门的测试工具，它可以测试图形用户界面和在客户和服务器间的网络通信。

再工程工具。再工程工具类可以被进一步按功能划分：

• 逆向工程工具——以源代码为输入，生成图形的结构化分析和设计模型、何处要使用的列表、以及其他设计信息。

• 代码重构和分析工具——分析程序语法，生成控制流图，然后自动地生成结构化程序。

• 联机系统再工程工具——用于修改联机的数据库系统(例如，转换 IDMS 或 DB2 为实体关系格式)。

上面的很多工具被局限用于特定的程序设计语言(虽然大多数主流语言被涉及)，并且需要和软件工程师间进行某种程度的交互。

下一代的逆向和正向工程工具将更多地使用人工智能技术，采用应用领域特定的知识库(即一组分解规则，它可以应用到在某特定应用领域——如制造控制或飞行器电子设备——中的所有应用软件中)。AI 成分将辅助系统分解和重构，但是，它的应用仍将需要在整个再工程生命期内和软件工程师间的交互才能完成。

29.4 集成化 CASE 环境

虽然强调单独的软件工程活动的单个 CASE 工具可以带来很多收益，但是，CASE 的真正力量只能通过集成达到。集成化 CASE(I—CASE)的优势包括：(1)信息(模型、程序、文档、数据)从一个工具到另一个工具的平滑传递，以及从一个软件工程步骤到下一个步骤的平滑过渡；(2)减少需要用于完成软件配置管理、质量保证和文档生成等活动的工作量；(3)通过更好的计划、监控和通信增加对项目的控制；(4)改善大型项目的开发人员间的协调。

但是 I—CASE 也带来大量的挑战。集成需要软件工程信息的一致表示、工具间标准的接口、在软件工程师和每个工具间相似的通信机制、以及有效的使得 I—CASE 能够在不同硬件平台和操作系统间移动的方法。虽然提出了对这些挑战所蕴含的问题的解决方案的研究，但完全的 I—CASE 环境还仅仅是刚开始出现。

术语“集成”意味着“组合”和“闭包”。I—CASE 组合一系列不同的工具和不同的信息，使得在工具间、人员间能跨越软件过程实现通信的闭包。工具的集成，使得软件工程信息对每个需要它的工具均是可用的；使用法的集成，使得所有工具都提供相同的操作界面；开发哲学的集成，蕴含着采用标准的、应用现代惯例和已证明的方法的软件工程途径。

为了在软件过程的语境内定义集成，有必要为 I—CASE 建立一组需求，一个集成化 CASE

- 提供一种机制，使得包含在环境中的所有工具间可以共享软件工程信息。
- 使得对一个信息项的改变将可以跟踪到其他的相关信息项。
- 对所有的软件工程信息提供版本控制和整体的配置管理。
- 允许对包含在环境中的任意工具进行直接的、非顺序的访问。
- 为集成工具和数据到标准的工作分解结构(第 7 章)中提供自动支持。
- 使得每个工具的用户能够在人机界面方面享用相同的产物。
- 支持软件工程师间的通信。
- 收集能够用于改善过程和产品的管理和技术度量。

为了达到这些需求，CASE 体系结构(图 29-1)中的每个构造积木块必须以一种无缝的方式结合在一起。如图 29-3 所示，基础构造积木块(环境体系结构，即硬件平台和操作系统)必须通过一组可移植服务被“联结”到实现上面需求的集成框架

29.5 集成体系结构

软件工程项目组使用CASE工具、对应的方法、以及过程框架来创建软件工程信息池，集成框架方便了信息进出信息池的传递。为了达到这个目标，下列体系结构构件必须存在：一个存储信息的数据库、一个管理信息变化的对象管理系统、一个协调CASE工具使用的工具控制机制、以及一个在用户动作和包含于环境中的工具之间提供一致路径的用户界面。大多数集成框架的模型(如参考文献[FOR90]和[SHA95])将这些构件表达为层次结构，一个仅仅描述上面提到构件的简单框架模型如图29-4所示。

用户界面层(图29-4)包括标准的界面工具箱和公共的表示协议。界面工具箱包含人机界面管理软件和显示对象库，二者提供了一致的在界面和单个CASE工具间的通信机制。表示协议是一组指南，它们给予所有CASE工具相同的观感，屏幕布局约定、菜单名和组织、图符、对象名、键盘和鼠标的使用，此外工具访问机制也被定义为表示协议的一部分。

工具层包括一组工具管理服务和CASE工具本身。工具管理服务(TMS)控制在环境中工具的行为，如果在一个或多个工具的执行中使用了多任务机制，TMS完成多任务同步和通信、协调从中心库和对象管理系统到工具的信息流、完成安全和审计功能、以及收集关于工具使用的度量。

对象管理层(OML)完成在第8章描述的配置管理功能。在本质上，在框架体系结构这层的软件提供了工具集成的机制，每个CASE工具被“插入”到对象管理层。和CASE中心库一起，OML提供集成服务——一组将工具和中心库耦合在一起的标准模块。此外，OML提供配置管理服务，使得能够标识所有的配置对象，完成版本控制，并提供对变化控制、审计、以及状况说明和报告的支持。

共享中心库层使得对象管理层能够与CASE数据库交互并完成对CASE数据库的访问控制。通过对象管理层和共享中心库层达到数据集成。

29.6 CASE中心库

Webster's字典中将单词“repository”定义为“any thing or person thought of as a center ofaccumulation or storage”，即，“任何被认为是积聚或存储中心的东西或人”。在软件开发的早期历史中，中心库确实是人——程序员，他必须记住所有和软件项目相关的信息的位置，必须回忆起那些从未写下来的信息，并重构已经失去的信息。很悲哀的是，使用人作为“积聚和存储的中心”(虽然这符合字典的定义),并不能够很好地工作。今天,中心库是一种“东西”——一个作为软件工程信息积聚与存储的中心的数据库。人(软件工程师)的角色是使用与中心库集成在一起的CASE工具去和中心库打交道。

在本书中，一系列不同的术语被用于指示软件工程信息的存储地方：CASE数据库、项目数据库、集成化项目支撑环境(IPSE)数据库、数据字典(限制的数据库)、以及中心库。虽然这些术语中的某些有微妙的不同，但是，所有均指“东西”——积聚和存储的中心。

29.6.1 在 I-CASE 中中心库的角色

在 I-CASE 中中心库是一组实现“数据—工具”以及“数据—数据”集成的机制和数据结构，它提供了明显的数据库管理系统的功能，但是，此外，中心库还完成下面功能[FOR89b]：

• 数据完整性——包括确认中心库的数字项，保证相关对象间的一致性，以及当对一个对象的修改需要对其相关对象进行某些修改时自动完成“层叠式”修改等功能；

• 信息共享——提供在多个开发者和多个工具间共享信息的机制，管理和控制对数据及加锁/未锁对象的多用户访问，以使得修改不会被相互间不经意地覆盖；

• 数据—工具集成——建立可以被 I—CASE 环境中所有工具访问的数据模型，控制对数据的访问，以及完成合适的配置管理功能；

• 数据—数据集成——数据库管理系统建立数据对象间的关系，使得可以完成其他功能；

• 方法学实施——存储在中心库中的数据的 ER 模型可能蕴含了特定的软件工程范型——至少，关系和对象定义了一系列为了建立中心库的内容而必须进行的步骤；

• 文档标准化——在数据库中对象的定义直接导致了创建软件工程文档的标准方法。

为了实现这些功能，中心库用元模型来定义。元模型确定了信息如何存储于中心库、数据如何被工具访问及被软件工程师使用、数据安全性和完整性被维护的方便程度、以及现存模型被扩展以适应新的需要的容易程度[WEL89]。

元模型是一个模板，软件工程信息被放置其中。可以创建一个实体—关系的元模型，但是，其他更高级的模型也可以考虑。关于这些模型的详细讨论已超出本书范围，有兴趣的读者可参考文献[WEL89]、[SHA95]和[GRI95]。

29.6.2 特征和内容

可以从两个方面来很好地理解中心库的特征和内容：在中心库中存放什么？以及中心库提供什么特定的服务？通常，存储在中心库中的内容的类型包括：

• 将被求解的问题。

• 关于问题域的信息。

• 系统解决方案。 • 关于被遵从的软件过程(方法学)的规则和指令。

• 项目计划、资源和历史。

• 关于组织的信息。

存储在 CASE 中心库中的表示、文档和可交付产品的类型的详细列表包含在表 29-1 中。

表 29-1 CASE 中心库内容[FOR89b]

企业信息	构造
组织的结构	源代码；目标码
业务域分析	系统建造指示
业务功能	二进制映象
业务规则	配置依赖
过程模型(场景)	变化信息
信息体系结构	
	确认和验证
应用设计	测试计划；测试数据用例
方法学规则	回归测试脚本
图形表示	测试结果
系统图	统计分析
命名标准	软件质量度量
参考的完整性规则	
数据结构	项目管理信息
过程定义	项目计划
类定义	工作分解结构
菜单树	估算；进度
性能标准	资源装载；问题报告
时序限制	变化请求；状况报告
屏幕定义	审计信息报表定义
逻辑定义	系统文档
行为逻辑	需求文档
算法	外部/内部设计
变换规则	用户手册

一个强健的 CASE 中心库提供两种不同类型的服务：(1)可以从任何高级的数据库管理系统期望得到的服务类型；(2)特定于 CASE 环境的服务类型。

很多中心库需求和那些建立在商用数据库管理系统(DBMS)之上的典型应用软件的需求是相同的，事实上，当前大多数 CASE 中心库使用 DBMS(通常为关系

型或面向对象)作为基本的数据库管理技术。支持软件开发信息管理的 CASE 中心库的标准 DBMS 特征包括：

非冗余数据存储。CASE 中心库为软件系统开发过程中有关的所有信息提供单一的存储地方，从而去除了浪费的、潜在易错的信息复制。

高层的范围。中心库提供了公共的数据访问机制，使得数据处理设施不需要在每个工具中重复存在。

数据独立性。CASE 工具及目标应用与物理存储是隔离的，这使得当配置发生变化时，它们不会受影响。

事务控制。中心库管理多部件交互，通过在有并发用户及在系统失败时数据的维护来达到完整性。这通常蕴含了记录锁、两阶段提交、事务日志和恢复过程。

安全性。中心库提供了控制谁可以查看和修改包含在其中的信息的机制。至少，中心库应该采用多层口令和由单个用户赋予的许可级别。中心库也应该提供对自动备份和恢复、以及被选择信息组(例如，根据项目或应用)的归档的帮助。

特别的数据查询和报告。中心库允许通过方便的用户界面(如 SQL 或面向表的“浏览器”)直接访问其内容，使得用户能定义的分析超越了 CASE 工具集提供的标准报告。

开放性。中心库通常提供简单的移入/移出机制，以使得能够进行大量数据的装载和传输。接口通常是简单的 ASCII 文件传输或标准的 SQL 接口。某些中心库具有反应它们的元模型结构的高级接口。

多用户支持。一个强健的中心库必须允许多个开发者同时开发某应用系统，这必须管理多个工具和用户对数据库的并发访问，该功能使用的技术是访问仲裁和在文件或记录层次上加锁。对基于网络的环境，多用户可以支持中心库和常见的网络协议和设施的交互。

CASE 环境也对中心库提出了特殊的要求，这些要求超出了商用 DBMS 直接可用的功能。CASE 中心库的特殊特征包括：

复杂数据结构的存储。中心库必须适应如图、文档和文件等复杂数据类型，以及简单的数据元素。中心库也包含了描述存储在其中的数据的结构、关系和语义的信息模型(或元模型)，元模型必须可扩展，以使得可以适应新的表示和独特的组织信息。中心库不仅存储开发中的系统的模型和描述，而且也存储关联的元数据(即描述软件工程信息本身的附加信息，如，当某特定的设计构件被创建时，它当前的状况是什么、以及它依赖于其他的什么构件)。

完整性实施。中心库信息模型也包含规则或方针，描述合法的业务规则和其他关于存入中心库(直接或通过 CASE 工具)的信息的限制和需求。一个称为触发

器的设施可被用来在对象被修改时激活对象关联的规则，使得有可能实时地检查设计模型的合法性。

语义丰富的工具接口。中心库信息模型(元模型)包含了使得一系列工具能够解释存储在中心库中的数据含义的语义信息，例如，由某 CASE 工具创建的数据流图以基于信息模型的形式被存放在中心库中，独立于使用它的工具本身的任何内部表示，另一个 CASE 工具然后可以解释中心库的内容并在需要时使用这些信息。这样，存储在中心库中的语义信息允许在一系列工具间实现数据共享，这完全不同于特定的“工具—工具”约定或“桥梁式连接”。

过程/项目管理。中心库中不仅包含关于软件应用本身的信息，而且包含关于每个特定项目的特征，以及开发组织的一般软件工程过程(阶段、任务和交付产品)的信息，这样提供了自动协调技术开发活动和项目管理活动的可能性。例如，对项目任务状况的更新可以自动地完成或作为使用 CASE 工具的副产品，对开发者来说，状况更新可以容易地完成，不必要离开正常的开发环境。还可以通过电子邮件处理任务分配和查询。问题报告、维护任务、修改授权和维修状况可以通过访问中心库的工具协调和监控。

下面的中心库特征全部包含在软件配置管理(第 9 章)中，这里将再次进行考察，以强调它们和 I-CASE 环境的关系。

版本控制。随着项目的进展，将创建单个工作产品的很多版本。中心库必须能够存储所有这些版本，使得能够有效地管理产品发布并允许开发者在测试和调试过程中回到以前的版本。版本控制可以使用压缩算法使得存储花销最少，并允许通过某些处理重新生成任意的以前版本。

依赖跟踪和变化管理。中心库管理存储在其中的数据间的大量关系，这些关系包括：企业实体和加工间关系、某应用设计中部件间关系、设计构件和企业信息体系结构间关系、设计元素和可交付产品间关系、等等。其中某些关系仅仅是关联，而某些关系是依赖或托管关系。在开发对象间维护这些关系称为链接管理。

跟踪所有这些关系的能力对存储在中心库中的信息的完整性和基于这些信息生成可交付产品是至关重要的，并且，这是中心库概念对软件开发过程的改善最重要的贡献之一。在链接管理支持的很多功能中，标识和访问变化影响的能力是重要的。当设计演化以满足新的需求时，标识所有可能被影响的对象的能力使得能够更精确地估价成本、停工期和困难程度。它也可以帮助预防不期望的、可能导致缺陷和系统失败的副作用。

需求跟踪。依赖于链接管理的一个特殊功能是需求跟踪，这是指跟踪产生自某特定需求规约的所有设计构件和可交付产品的能力(正向跟踪)，以及标识哪个需求生成任何给定的可交付产品的能力(反向跟踪)。

配置管理。另一个依赖于链接管理的功能是配置管理。配置管理设施与链接管理和版本控制设施一起紧密协作，以跟踪一系列表示特定的项目里程碑或生产发布的配置。版本控制提供需要的版本，链接管理跟踪相互依赖性。例如，配置

管理经常提供建造设施，以自动化将设计构件变换为可执行的可交付产品的过程。

审计跟踪。和变化管理相关的是对审计跟踪的需要，审计跟踪将建立关于修改是什么时间、为什么、以及被谁完成的附加信息。实际上，这对具有强健的信息模型的中心库来说不是一个困难的需求，关于变化源的信息可以作为特定对象的属性被存放到中心库中。中心库触发机制将在设计元素被修改时，帮助提示开发者或他(她)正使用的工具去启动审计信息(如修改的原因)的输入。

29.7 小结

计算机辅助软件工程工具跨越软件过程的每个步骤和那些贯穿整个过程的全程性活动。CASE 组合一组构造积木块，从硬件和操作系统软件层次开始，到单个工具结束。

在本章中，我们考虑了 CASE 工具的分类，类别包括管理的和技术的活动，并跨越大多数软件应用域。每个工具类被考虑为一个“点解决方案”。

I-CASE 环境组合了针对数据、工具和人机交互的集成机制。数据集成可以通过直接信息交换、通过公共文件结构、通过数据共享或互操作、或通过使用完全的 I-CASE 中心库来实现；工具集成可以由一起工作的厂商定制设计或通过作为中心库的一部分的管理软件来实现；人机集成通过正在产业界日益变得流行的界面标准来实现。设计一个集成体系结构，可方便用户和工具、工具和工具、工具和数据、以及数据和数据间的集成。

CASE 中心库也被称为“软件总线”，当软件过程发展时，信息通过它移动，从工具传递给工具。但是，中心库并不仅仅是一个“总线”，它也是一个存放地，结合了高级的集成 CASE 工具的机制，因此，改善了软件开发的过程。中心库是关系型的或面向对象的数据库，该库是软件工程信息的“积聚和存储中心”。

参考文献

[FOR89a] Forte，G.，“In Search of the Integrated Environment，”CASE Outlook，March/April1989，pp.5-12.

[FOR89b] Forte，G.，“Rally Round the Repository，”CASE Outlook，ecember，1989，pp.5-27.

[FOR90] Forte,G.,“Integrated CASE:A Definition,”Proc.3rd Annual TEAMWORKERSIntl.User's Group Conference,Cadre Technologies,Providence,RI，March 1990.

[GRI95] Griffen, J., “Repositories: Data Dictionary Descendant Can Extend Legacy CodeInvestment, ” Application Development Trends, April 1995, pp.65-71.

[NIC90] Nichols, K.M., “Performance Tools, ”IEEE Software, May 1990, pp.21-23.

[QED89] CASE: The Potential and the Pitfalls, QED Information Sciences, Inc., Wellsley, MA, 1989.

[SHA95] Sharon,D.,and R.Bell,“Tools that Bind:Creating Integrated Environments, ”IEEE Software, March 1995, pp.76-85.

[SQE95] Testing Tools Reference Guide, Software Quality Engineering, Jacksonville, FL, 1995.

[WEL89] Welke, R.J., “Meta Systems on Meta Models, ”CASE Outlook, December 1989, pp.35-45.

思考题

29.1 列出所有你使用的软件开发工具。按照本章提出的分类方法组织它们。

29.2 使用主机和终端的“旧式”软件开发环境体系结构的长处是什么？其缺点是什么？

29.3 使用在第 14 和/或 19 章引入的思想，你将如何建议可移植服务的建造？

29.4 为包含在 29.3 节中提到的范畴的项目管理工具建造一个纸上原型，使用本书第 2 部分作为附加指南。

29.5 研究面向对象数据库管理系统，讨论为什么 OODBMS 是 SCM 工具理想的选择。

29.6 收集关于至少三个分析和设计工具的产品信息，给出它们的特征比较矩阵。

29.7 收集两个 PRO/SIM 工具的产品信息，给出它们的特征比较矩阵。

29.8 收集关于至少三个第四代编码工具的产品信息，给出它们的特征比较矩阵。

29.9 是否存在动态测试工具是“唯一的可行方式”的情况？如果存在，是什么？

29.10 讨论其他的人类活动，在其中一组工具的集成比每个工具的单独使用可以提供更多的实质性收益。不要使用计算领域的例子。

29.11 用你自己的话描述“数据—工具”集成的含义。

29.12 在本章的不少地方使用了术语“元模型”和“元数据”，用你自己的话描述这些术语的含义。

29.13 你认为还有其他的配置项可以包含在表 29－1 显示的中心库内容中吗？给出列表。

推荐阅读文献及其他信息源

在 1980 年代出版了一系列关于 CASE 的书籍，其目的是利用当时产业界对此的高度兴趣，结果，几乎很少有符合主题的书籍出现。在这些已出版的书籍中，很多具有下面所列的一个或多个失误：(1)书籍仅仅着重于非常窄的工具域(如，分析和设计)，虽然宣称覆盖了更广的范围；(2)书籍花费非常少的时间在 CASE 上，而更多的时间是概述(经常是贫乏的)工具支持的基本方法；(3)书籍花费很少的时间讨论集成问题；(4)由于强调特定 CASE 工具，表示法已过时。参考下面的书籍可以避免某些失误：

Braithwaite，K.S.，Application Development Using CASE Tools，Academic Press，1990.

Fisher，C.，CASE：Using Software Develpoment Tools，Wiley，1988.

Gane,C.，Computer-Aided Software Engineering：The Methodologies，the Products andtheFuture，Prentice-Hall，1990.

Lewis，T.G.，Computer-Aided Software Engineering，Van Nostrand Reinhold，1990.

McClure，C.，CASE is Software Automation，Prentice-Hall，1988.

Schindler，M.，Computer-Aided Software Design，Wiley，1990.

Towner，L.E.，CASE：Concepts and Implementation，McGraw-Hill，1989.

由 Chikofsky 编辑的文集(Computer-Aided Software Engineering，IEEE Computer SocietyPress，1988)包含了一些早期的关于 CASE 和软件开发环境的论文。当前最好的关于 CASE 工具的信息源是技术期刊和产业快讯。

虽然人们对 I—CASE 环境和 CASE 中心库具有很大兴趣，但是，非常少的书籍详细地涉及这些主题。Garg 和 Jazayeri(Process Centered Software Engineering Environments，IEEE ComputerSociety Press，1995)

考虑了过程模型驱动的软件开发环境，Brereton(Software EngineeringEnvironment，Wiley，1988)，Charette(Software Engineering Environments， McGraw-Hill，1986)和Barstow、Shrobe和Sandewall(Interactive Programming Environments， McGraw-Hill，1984)提出了不同的“理想的”CASE环境，并对本章讨论的内容提供了历史的补充。

IEEE标准1209(Evaluation and Selection of CASE Tools)提出了一组评价针对“项目管理过程、开发前过程、开发过程、开发后过程和完整过程”的CASE工具的指南。Automated Software Engineering(Kluwer Academic Publishers，URL：http：//www.wkap.nl)是一个讨论软件工具新进展的杂志。

大量的CASE工具信息可从WWW上得到，每个主要的CASE厂商都有自己的网站，并且有很多工具的清单。在下面网址可以找到CASE工具的索引：

http：//www.cern.ch/PTTOOL/SoftKnow.html

很多CASE厂商提供网站，一个完全的、按工具类别组织的指针集可在下面网址上得到：

http：//www.qucis.queensu.ca./Software-Engineering/toolcat.html

对自由的和共享的CASE工具的直接访问以及CASE厂商网址目录可在如下网址上得到：

http：//osiris.sunderland.ac.uK/sst/casehome.html

各种工具集成标准(包括PCTE)的电子拷贝可在MIT Microsystems Lab上得到：

http：//www-mtl.mit.edu/CFI/Whole server.html

很多包含在前面章节中的电子参考文献包含了指向特定CASE工具的指针。

关于CASE环境的最新WWW文献列表可在http：//www.rspa.com上得到。

① 在许多情况下，对软件工程来说可用的工具仅有编译器和文章编辑器。这些工具只能专用于代码编写——整个软件过程中占不到20%的工作。

① 本章未“推荐阅读文献及其他信息源”中列出了本节中各类有代表的工具的销售商名单

第30章　未来之路

在本章前面的29章中，我们探讨了软件工程过程。我们讨论了管理规程和技术方法、基本原则和专门技术、可以自动化的面向人的活动和任务、纸和笔的符号以及CASE工具。我们论证了对质量的测度、限制和重点，关注了如何产生

满足客户需要的软件、可靠的软件、可维护的软件、更好的软件。但是，我们并未允诺软件工程是万能药。

当我们走向新世纪的黎明之季，软件和系统技术对从事基于计算机的系统的建造的每个软件专业人员和每个公司来说仍是一种挑战。Max Hopper [HOP90] 对当前状态有如下陈述：

因为信息技术的变化正变得如此快速和不可遏阻，并且落后的后果是如此的不可挽回，因此，公司将或者掌握技术，或者死去。…将它想象为技术踏车，公司将必须越来越艰苦地转动它，仅仅为了保持在原有位置上。

在软件工程技术方面的变化确实是“快速和不可遏阻”的，但同时进展经常又是相当的慢。在决策采用新的方法(或新工具)的时候，需要进行理解其用途的培训，而且要技术引入到软件开发文化中，此时会伴随出现某些新东西(以及甚至更好的东西)，而且过程也是重新开始。

在本章中，我们谈未来之路。我们的目的不是去探讨每个有希望的研究领域，也不是去凝视“水晶球”并预言未来。我们将探讨变化的范围，以及变化本身将如何在未来的几年影响软件过程。

30.1 软件的重要性——再论

计算机软件的重要性可以以很多方式来描述。在第 1 章，软件被表示为区分器，软件交付的功能是产品、系统和服务来区分的，它提供了在市场上的竞争能力。但是，软件并不仅仅是区分器，作为软件的程序、文档和数据帮助生成任何单个、公司或政府可以获取的最重要的日用品——信息。Pressman 和 Herron [PRE91] 以下面的方式描述软件：

计算机软件是 90 年代对现代社会的几乎每个方面均有重要影响的仅有的少数关键技术之一。它是使商业、产业和政府自动化的机制，是传递新技术的媒介，是捕获有价值的专家意见供其他人使用的方法，是区分某公司的产品和其竞争产品的手段，以及是为进入企业而学习共同知识的窗口。软件对商业的几乎每个方面均是关键的，但是，软件也以很多方式呈现为隐藏技术。当我们去工作、购买商品、进银行、打电话、看医生、或从事日常的反映现代生活的成百的活动中的任意一个活动时，我们都会遇到软件(经常我们没有意识到)。

软件是普遍深入人心的，但是，很多居于负责位置的人几乎没有或没有真正理解它到底是什么、如何建造它、或它对他们(和它)控制的机构意味着什么。更重要的是，他们几乎没有意识到软件带来的危险和机会。

软件对人们生活的普遍渗透给了我们一个简单的结论：一旦当某技术具有广泛的影响——可以拯救生命或威胁生命、建造业务或破坏业务、协助政府领导或误导他们，那么，它必须被“小心处理”。

30.2 变化的范围

对计算有影响的技术的变化似乎接受了称为 5—5—5 规则[①]的级数——一个基础的新概念从初始的思想到大量的市场产品似乎要经历 15 年[②]。在第一个 5 年中，新思想被形成并进化为用于展示基本概念的原型；实验的原型被科学家和工程师在第二个 5 年内精化，并且第一个产品(反应新思想)也在这时间内出现；最后的 5 年用于将产品(及其后继)投放市场。在 15 年(5－5－5)结束时，一个具有技术价值的新思想可能增长成包含数十亿美元的市场(图 30－1)[③]。虽然 5－5－5 规则仅仅是一个近似，但是，从初始思想到相当大的市场份额的 15 年时间跨度似乎是合理的衡量，我们可以使用它来测度在计算机业务中的演化变化。

过去 40 年在计算领域的变化的主要驱动因素是“硬科学”的进展——物理、化学、材料科学和工程方面的进展。5—5—5 规则似乎对从硬科学中的基础导出新技术是非常合适的，然而，在下面的几十年中，在计算领域的革命性进展的驱动力可能是“软科学”——人的心理学、神经生理学、社会学、哲学和其他。从这些学科导出技术的酝酿阶段是非常难于预测的。

例如，对人类智能的研究已经进行了几个世纪，仅仅得到了对思想的心理学和大脑的神经生理学的断断续续的了解，然而，在过去的 30 年却得到了很大的进展。从软科学导出的信息正被用于创建新的软件方法——人工神经网络[WAS89] ——它可能导致机器学习和对用传统的基于计算机的系统不可能解决的“模糊”问题的解决。

软科学的影响可能帮助形成硬科学中的计算研究方向，例如，“未来计算机”的设计可能受对大脑生理学的了解的指导，比受对传统的微电子学的了解的指导要多。

在未来十年影响软件工程的变化将同时受到来自四个方向的影响：(1)做工作的人；(2)他们使用的过程；(3)信息的性质；(4)基本的计算技术。在下面几节中，将对每个因素——人、过程、信息和技术——详细论述。

30.3 人及他们建造系统的方式

每个必须建造现代的基于计算机的系统的公司都面临着困难的抉择。高技术系统需要的软件变得越来越复杂，并且产生的程序大小成比例地增长，曾经有一个阶段，当一个程序需要 100 000 行代码时，就被认为是大型应用软件，今天，对个人计算机应用(如，字处理器、电子表格、图形程序)的一般程序长度经常都是该大小的很多倍，而用于工业控制、计算机辅助设计、信息系统、电子仪器、工厂自动化、以及几乎每个其他的“产业相关”的应用的程序经常超过 1000 000 行代码[①]。

程序平均大小的快速增长将并不会给我们带来什么问题，如果不是由于这样一个简单的事实：当程序大小增加，必须为此程序投入的人力也会增加。经验表

明，当一个软件项目组的人数增加，则项目组的整体生产率可能受到损害。围绕这个问题的一个解决方法是增加软件工程项目组的数量(第 3 章)，因而将人员划分为个别的工作小组。然而，当软件工程项目组的数量增加，他们之间的通信也会变得困难和费时，就和单个间的通信情况一样，更糟糕的是，通信(在个体间或项目组间)趋向于低效——即，太多的时间只能传递太少的信息内容，而且，经常的情况是，重要的信息“分裂为破碎的片断”。

如果软件工程界要有效地解决好通信困难，那么，软件工程的未来之路必须包括在单个和项目组间相互通信方式的根本性变化。电子邮件、公告牌和中心的视频会议现在是常见的连接大量人员为一个信息网的机制，当然，这些工具在软件工程范围内的重要性不能被过分强调。通过有效的电子邮件或公告牌系统，在纽约的软件工程师遇到的问题可以通过在东京的同事的帮助而解决。在现实中，公告牌和 WWW 网址已成为知识源，允许大量技术人员的集体智慧能够被组合起来解决技术问题或管理问题。

视频使通信个性化。在大多数情况下，它使得在不同地方(或不同大陆)的同事可以定期地“开会”。但是，视频也提供了另一个好处，它能被用作关于软件的知识中心库，并用于培训项目中的新来者。

当硬件和软件技术不断进展，工作地点的性质将发生改变。下面的场景摘录自 Pressman 和 Herron [PRE91] ，描绘了在 21 世纪的第一个十年中软件工程师的工作环境：

“早上好”，当你进入办公室时说。

你的工作站屏幕亮了，一个窗口出现在屏幕上，一个具有两性特点的面孔出现，并且一个非常具有人性特点的声音说：“早上好，你有 6 个语音邮件消息，两份传真和一些日常工作项，有 5 个开发任务”。该面孔和声音属于你的“代理”，一个完成一系列高级书记职责的界面程序。它已经被定制以符合你的需要，识别你的声音，并能同时做很多事情——如定时回应你的电话，查找消息，直接和你通信，以及完成其他的数据处理功能。和代理的通信可以通过口头或手写来完成，但大多数人喜欢直接对代理说。

“给我显示日常工作项和开发任务”，你说。

立即，工作项列表出现在屏幕上，并且代理开始大声地读该列表，并在读时加亮每个项。

“请安静，并保留列表”，你打断代理。“当你保留列表时，按关键词检查语音邮件和传真”。

你刚刚要求代理对每个到达消息进行分析，以确定是否它包含一组关键词中的任意内容(这些关键词将是人的名字、地址、电话号码，或你认为是特别重要的话题)。当你浏览屏幕上的工作项列表时，你看见了两个约会、一些将要打的电话、以及需购买的周年纪念礼品。

此时你已经浏览完工作项列表，代理的面孔也从屏幕上消失。

此时尚早，你还没有进入工作状态。你困惑(但是你为什么这样呢，你是在和机器通信！)地问：“我刚才要你做什么？”

“你要我按关键词检查语音或传真，你需要列出来吗？”

“好，但是只列那些有变化的部分”，一个消息表出现在屏幕上。你看着列表中的一项并说“打开”，在不到一秒钟的时间内，一个内置于工作站中的视频摄像机对你说“打开”的时候跟踪你的眼睛移动，系统计算出你看着的是哪一项。你开始阅读一会消息，然后停止。

“请查找出在工厂自动化系统中所有在上个月被修改过的模块，存储模块名、修改源和日期，并生成我复审它们的工作项”。

“你将使用系统的什么版本？”当一个滚动窗口出现时，代理问。

“所有”，你回答。

“OK”，代理说。

当你在阅读邮件时，代理将让一个“学徒”去完成你请求的任务，即，代理“孵化”一个任务来完成配置管理功能。在几秒钟内，代理就返回来听取你的指示，同时，第一个学徒正在搜索 CASE 中心库以查找模块名。

“可以为我启动一个字处理器吗？”你问代理。一个字处理程序，和你今天看见的完全不一样，出现在屏幕上，你开始口述一封信(也可使用键盘或手写板)。当你说出每个单词，文字出现在屏幕上。当你口述时，你想起某些事要代理去做，可以使用击点设备在代理的窗口上点击一下。

“我需要模块 find.inventory.item 的源代码清单，把它插在我正工作的文档的文本中，加上我将注意到的标志。另外，给系统工程部的 Emily Harrison 去电话，告诉她我将在今天稍后把文档传送给她”。

“OK”，代理回答到。然后孵化学徒去生成表格并在你回去口述时打电话。

上面的“对话”涉及的环境将改变软件工程师的工作环境，替代工作站被用作工具，硬件和软件变成了助手，完成仆人的任务、协调人和人的通信、以及在某些情况下应用领域中的特定知识去增强工程师的能力。

知识获取的方式发生了很大变化。在Internet上，软件工程师可以参加与他当前关心的技术领域相关的新闻组，在新闻组中张贴的问题可得到全球的其他感兴趣者的回答。WWW为软件工程师提供了世界上最大的关于软件工程的研究论文和报告、培训指南、以及参考文献库。[①]

如果以过去的历史为标准，应该说人们本身并没有改变，然而，他们通信的方式、他们工作的环境、他们获取知识的方式、他们使用的方法、他们应用的限制——进而，因此，软件开发的整体文化——将有较大的甚至深层的改变。将影响从事软件工程工作的人们的某些变化在图 30－2 中给出。

30.4 “新”的软件过程

将软件工程实践的前 20 年说成为“线性思维”的时代是合理的，在传统的生命期模型的养育下，软件工程被处理为线性的活动，其中应用一系列顺序的步骤，以解决复杂的问题。然而，软件开发的线性方法违反了大多数系统实际被建造的方式。在现实中，复杂系统迭代地、甚至增量地演化，正是为此，软件工程界的大部分正移向软件开发的演化模型。

演化过程模型认识到：不确定性支配了大多数项目，即时间经常是不可能短的，迭代提供了增量式解决问题的能力，甚至当一个完整的产品不可能在分配的时间内完成时。演化模型强调对增量式工作产品、风险分析、计划及计划修订、以及客户反馈的需要。

但是，什么活动必须放置于演化过程中？在过去十年，SEI 提出的能力成熟度模型 CMM [PAU93] 对改善软件工程实践的努力产生了实质性的影响。CMM 引发了大量争论(如参考文献 [BOL91] 和 [GIL96])，然而，它很好地给出了当坚实的软件工程实践被进行时，必须存在的属性。

对象技术的使用(本书第四部分)是朝着演化过程模型变化趋势的自然结果，基于构件的组装(如果它能被广泛实现)将对软件开发生产率和产品质量有深远的影响，可复用程序构件的导出是面向对象范型的自然结果。

构件复用提供了立即的和引人注目的收益(第 26 章)，当复用和支持应用原型的 CASE 工具结合时，程序增量的建造可以比使用传统方法更快。原型将客户吸引入过程，因此，有可能客户和用户将更多地参于软件开发，反过来，这将导致更高的终端用户满意度和更好的软件质量。

30.5 表示信息的新模式

在过去 20 年，用于描述商业界的软件开发工作的术语发生了一些微妙的变迁。20 年前，术语“数据处理”是描述计算机在商业中应用的习惯用语，今天，数据处理已经让位给另一个短语——“信息技术”——它蕴含的事相同，但是在关注点上有微妙的偏移，重点不仅仅是大量数据的处理，而是从这些数据中抽取有意义的信息。显然，这是总的目的，但是，术语上的改变反应了在管理哲学上更重要的变化。

当我们今天讨论软件应用时，“数据”和“信息”这两个词再三地出现，我们也在某些智能应用中遇到词“知识”，但是它的使用相对较少。事实上，没有人在计算机软件应用的范围内讨论智慧。

数据是未加工的信息——必须被处理以使其有意义的事实的集合，信息是通过将事实和给定的语境相关联而导出的，知识将某个语境中得到的信息和在不同语境中得到的信息相关联，最后，当从完全不同的知识导出一般性原理时，智慧出现了。图 30－3 以图表形式表示了这四个“信息”的视图。

当前，绝大多数软件是用于处理数据或信息的，21 世纪的软件工程师将同样关心系统的知识处理，知识是二维的。针对一系列相关和不相关的话题而收集的信息被联结在一起形成我们称为知识的实体，其中的关键是我们如何关联来自一系列不同源(它们在相互间可能没有明显的联系)的信息，并将它们组合在一起为我们提供某些独特收益的能力。

为了说明从数据到知识的进展，以人口普查数据为例，它指明在 1991 年美国的出生数是 410 万，该数值表示一个数据值。将该数据和前 40 年的出生数关联，我们可以导出一种有用的信息——正在变老的 1950 年代的“潮婴儿”正在他们的分娩年龄结束前作最后的生育努力。然后，该信息可以和其他似乎无关的信息相关联——例如，将在下个十年退休的初中中学教师的当前数量；具有初中和中级教育程度毕业的学生数；或政治家承受的降低税率，并因此限制教师薪金增长的压力。

每个这些信息可以被组合而形成知识的表示——在 90 年代后期将对美国的教育系统有较大的压力，该压力将持续十年。使用该知识，可能形成一个商业机会，将存在重要的机会去开发新的比当前的方法更有效和更价廉的学习模式。

我们已经处理数据几乎 40 年，抽取信息 20 多年，现在，软件工程界面临的最重要的挑战之一是建造在“信息”谱中向下一步的系统——从数据和信息中以实用的和有益的方式抽取知识系统。

30.6 技术作为推动力

建造和使用软件的人员、应用的软件工程过程、以及生产的信息均受到硬件和软件技术方面发展的影响。历史上，硬件是计算领域的技术推动力，新的硬件技术提供了潜能，然后软件建造者根据用户需求力图去发挥该潜能，图 30－4 应用 5－5－5 规则试图将各种硬件技术放置到整个演化周期中。将一个特定的技术放置到 5－5－5 曲线上可能是困难的，例如，移动计算技术当前已发展到产品阶段，但是它还不是一个成熟的市场，因此，它被放置在技术成熟度的原型阶段。

硬件技术的未来之路可能会沿着两条并行的路径发展，在一条路径上，硬件技术将继续快速地演化，由于传统硬件技术提供的更大的能力，对软件工程师的要求将继续增长。

但是，硬件技术方面的真正变化可能发生在另一条路径上，非传统的硬件体系结构(如，大规模并行机、光处理器、神经网络机)的发展可能会导致在我们建造的软件种类上的根本性变化以及我们的软件工程方法的基本改变。因为这些非传统的方法还停留在 15 年周期的第一阶段，要确定哪个将存活并成熟是困难的，而要预测软件界将如何变化以适应它们则是更困难的。

我们已经注意到，软件技术往往反应出硬件技术的变化。应用 5－5-5 规则到软件技术(图 30－5)，我们发现今天的软件产品将可能由处于成熟度的第一和第二阶段的软件技术进行结合，且可能被替代。在 21 世纪的曙光到来时，显示在图 30－5 中原型阶段的技术将无容置疑地变得极其重要。事实上，复用和基于构件的软件工程可能提供最好的在系统质量和走上市场时间方面的按量级的改善。

软件工程的未来之路将被软件技术驱动。当软件更激烈地移进模糊问题领域(AI、人工神经网、专家系统)时，有可能演化成的软件开发方法将成为主导范型。当面向对象方法开始主宰软件界时，软件工程的演化范型将被修改以适应构件复用，建造“软件 IC”的“铸造车间”可能会成为一种主要的新软件业务。事实上，随时间流逝，软件商业可能开始看起来非常象今天的硬件商业，可能有建造离散设备(可复用软件构件)的厂商、其他建造系统构件(如，一组人机交互工具)的厂商、以及为终端用户提供解决方案的系统集成者。

软件工程将发生变化——对此我们可以肯定。但是不管发生如何根本性的变化，我们可以保证质量将不会失去其重要性，有效的分析和设计以及有效的测试将总是在基于计算机的系统的开发中占据自己的位置。

30.7 结束语

一个有趣的现象，并不完全是巧合，是：本书的第 4 版完成了一次 5-5-5 规则。基本的“概念”(第 1 版)出现于 1982 年，一个更精化的“原型”(第 2 版)在 1987 年发布，非最后的“产品”(第 3 版)于 1992 年完成，现在，第 4 版终结了最后一个 5 年周期。在过去的 15 年中，本书发生了引人注目的变化——在范围、规模和内容等方面。象软件工程一样，它已经长大并(有希望)在几年内成熟。

计算机软件开发的工程方法是已经该出现的哲学体系，虽然在合适的范型、自动化程度以及最有效的方法等方面还存在争论，软件工程的基本原则现在已在产业界得到普遍接受。然而，为什么我们仅仅是在近年来才看见它们的广泛采用呢？

我认为答案是由于技术转化和伴随而来的文化变化的原因，即使我们的大多数人认识到了软件的工程法律的需要，我们仍需要与过去的习惯惯性作斗争。

为了使转化更容易，我们需要很多东西——一个更成熟的软件过程、有效的方法、强大的工具、实践者的接受和管理者的支持、以及大量的教育和“广告”。软件工程还没有收到大量广告带来的收益，但是，随时间流逝，软件工程概念正在兜售它自己。在某种形式上，本书是对该技术的一次“广告”。

你不可能同意本书中描述的每一个方法，书中某些技术和观点也是相互矛盾的，其他的调整必须的也是以使其在不同的软件开发环境中可用。然而，我真诚地希望本书已经描述了我们面临的问题、展示了软件工程概念的优势、也提供了方法和工具的框架。

当我们开始我们向新的千年进军时，软件已经变成了世界上最重要的产品和最重要的产业，它的影响和重要性已经走过了一段长长的路。然而，新一代的软件开发者必须迎接很多和前一代人面临过的同样的挑战。让我们希望迎接挑战的人们——软件工程师——将具有更多的智慧去开发改善人类条件的系统。

参考文献

[BOL91] Bollinger，T.，and McGowen，C.，“A Critical Look at Software CapabilityEvaluations，”IEEE Software，July 1991，pp.25-41.

[GIL96] Gilb，T.，“What is Level Six？”IEEE Software，January 1996，pp.97－98，103.

[HOP90] Hopper，M.D.，“Rattling SABRE，New Ways to Compete on Information，”Harvard Business Review， May/June 1990.

[HOR90] Homer， M.，“Future Directions in CASE，”Le Rendez-vous du Génie Logicial，Centre International de Communication Avancée(CICA)，Sophia Antipolis，France， May 29，1990.

[PAU93] Paulk，M.et al.，Capability Maturity Model for Software，Software EngineeringInstitute， Carnegie Mellon University， Pittsburgh，PA，1993.

[PRE91] Pressman，R.S.， and S.R.Herron，Software Shock， Dorset House， 1991.

[WAS89] Wasserman，P.D.，Ne ural Computing：Theory and Practice，Van Nostrand Reinhold，

思考题

30.1 找一本本周的主要商业和新闻杂志，列出可用于说明软件的重要性的每一篇文章或每一项新闻。

30.2 选出三个已经形成主要产品的“基础性新思想”，画出时间表，并确定 5-5－5 规则是太保守的、太乐观的或是好的。

30.3 为描述在 30.3 节的软件工程师的环境加入附加的特征，画出注释草图，它表现了 2005 年软件工程师的办公室的工作模式。

30.4 复审在第 2 章中讨论的演化过程，研究之并收集当前该主题的论文，基于论文中描述的经验总结演化范型的优势和缺点。

30.5 给出一个例子，它从未加工数据的收集开始，然后是信息获取，再后是知识，最后是智慧。

30.6 选择在图 30－4 中表示的任何一个硬件技术，并撰写一篇 2～3 页的论文，对该技术进行概述。在班上提交论文。

30.7 选择在图 30－5 中表示的任何一个软件技术，并撰写一篇 2～3 页的论文，对该技术进行概述。在班上提交论文。

推荐阅读文献及其他信息源

讨论软件和计算的未来之路的书籍跨越了技术、科学、经济、政治和社会等诸多方面的问题。Negroponte 的畅销书(Being Digital， Alfred A. Knopf，Inc.，1995)讨论了计算及其在 21 世纪的整体影响。 Naisbitt 和 Aburdene(Megatrends 2000， William Morrow & Co.，1990)提供了已经在进行中的变化的迷人图画。 Rich 和 Waters(The Programmer’s Apprentice，Addison-Wesley，1990)对“在未来的软件开发中期望什么”这一问题提出了一种观点。 Bowyer(Ethics and Computing，IEEE Computer Society Press， 1995)考虑了对软件“专业人员以对社会负责的方式工作”的需要。

在更窄的计算和软件范围内，Allman(Apprentices of Wonder，Bantam Books， 1989)描述了人工神经网络的潜在影响——该书建议在我们谈论“软件”一词时的含义应有根本性变化。Stoll(The Cuckoo’s Egg,Doubleday,1989)提出了对计算机网络、黑客和计算机安全世界的一种迷人的观点——这些话题在当我们变成一个集成的“电子社会”时是非常重要的。Yourdon(The Rise and Resurrection of the American Programmer，Prentice-Hall，1996)预测了软件产业在美国的持续主导地位。

为了历史的兴趣， Alvin Toffler(Powershift， Bantam Publishers，1990)完成了从 Future Shock 开始的三部曲，通过讨论已经建立起的权力结构在世界范围内的瓦解——权力的变迁，他将其直接归因于软件及其生产的信息。他的很多预言已成为现实。

GartnerGroup 出版了详细的报告，预测了很多计算技术领域的未来趋势，某些资料可在下面网址上得到：

http：//www.gartner.com/default.html

Byte Magazine 永远是关于软件和计算的当前和未来趋势的很好的信息源，它的地址是：

http：//www.byte.com

某些更根本性的观点可访问 Hotwired 和 Trajectories 而得到：

http：//www.hotwired.com/login/

http：//www.nets.com/site/raw/trajectories.html

对 21 世纪技术趋势的深入讨论可在下面网址上得到：

http：
//www.nml.org/resources/misc/nasa_technology_directions/tablecontents.html

The Institute for Learning Sciences 汇编了计算和软件对教育的未来影响的讨论，可在下面网址上得到：

http：//www.ils.nwu.edu/～e_for_e/nodes/NODE-269-pg.html

其他对计算的当前和未来趋势的讨论可使用WWW上的标准搜索引擎得到。

① 该规则首先是由数字设备公司的MichaclHorner在[HOR90]中提出的。

② 当然，这假定该规则很好，且有足够多的资源可被使用。

③ 图 30－1 已同时考虑新型计算机硬件有相应的快速发展。目前新过程的生命期——从初始的概念期直到产品的作废时——是滞后 4 年的时间。

① 随着面向对象技术的利用和可编程构件复用的不断增加，编程代码量已大大减少，从而节约了不少时间，但是，整个程序的代码量（包括可复用构件在内）却仍然在不断地增长。

① 参见本书的“推荐阅读文献及其他信息源”