PEARSON

40周年中文纪念版

人月神话

THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

(Frederick P. Brooks, Jr.)
[美]小弗雷德里克·布鲁克斯/著
UMLChina翻译组 汪颖/译

PEARSON
清华大学出版社

THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

PEARSON

40周年中文纪念版

人月神话

有些书对读者和作者像是年金，它们年复一年地分红……

《人月神话》仍然是计算机书籍中被引用次数最多的经典著作，而且即便本书最初出版于20世纪，其内容至今仍未过时。在阅读的时候，每隔几页不说一句“对极了！”是很难受的。

——Steve McConnell，Construx公司首席软件工程师
名著《代码大全》、《快速软件开发》作者

这是一本经典著作，与软件开发有关的每一个人都应该不止一遍地读这本书。

——Philippe Kruchten，Rational统一过程首席架构师

我唯一一本读过很多遍的书，是Fred Brooks的《人月神话》，实际上我每过一两年都重读一遍。因为这本书文笔很好，而且书中的忠告很有价值，即使是在40年以后。当然，其在很多细节上和我们现在做事情的方法有所不同。我们的工作更自动化，计算机的“马力”更强劲，但书中依然有许多好的忠告，我非常推崇这本书。这是我唯一能想起来的你能从中体会到乐趣和思想的计算机科学书籍。

——Brian Kernighan，名著《C程序设计语言》的合著者之一
（与Dennis M. Ritchie合作）

上架建议：软件开

ISBN 978-7-302-39264-4

9 787302 392644 >

定价：68.00 元

清华大学出版社数字出版网站

WQBook 书文局泉

www.wqbook.com

人月神话

(40 周年中文纪念版)

[美] 小弗雷德里克·布鲁克斯 著
UMLChina 翻译组 汪颖 译

清华大学出版社
北 京

图书在版编目(CIP)数据

人月神话(40 周年中文纪念版) / (美) 布鲁克斯(Brooks, F. P.) 著；UML China 翻译组，汪颖 译. —北京：清华大学出版社，2015(2015.6 重印)

书名原文：The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition

ISBN 978-7-302-39264-4

Ⅰ. ①人… Ⅱ. ①布… ②U… ③汪… Ⅲ. ①软件工程 Ⅳ. ①TP311.5

中国版本图书馆 CIP 数据核字(2015)第 024400 号

责任编辑：陈　莉　高　岫
封面设计：周晓亮
版式设计：牛静敏
责任校对：曹　阳
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：180mm×250mm　　**印　张**：24.5　　**字　数**：316 千字
版　　次：2015 年 4 月第 1 版　　**印　次**：2015 年 6 月第3 次印刷
印　　数：10001～13000
定　　价：68.00 元

产品编号：052237-01

神品——代序*

这是本书中唯一的一节废话。

我是一个书狂，积习甚深，费尽心机在软件工程、系统工程方面积累了一些书。书，在我看来当分为神品、精品和普通三等，其中神品、精品又分别有一、二和三品之分。我所收集的书中，软件工程书大都属于精品，神品只有两本，Frederick P. Brooks 的这本书就属于神品之列。

软件作为一个行业，逐步背起了“solving the wrong problem”(解决错误的问题)的名声。问题决定解决方案，这也就是说，我们一直在制造错误解决方案！这方面有大量的证据，其中最著名的是美国政府统计署(GAO)的数据：全球最大的软件消费商(美国军方)每年要花费数十亿美元购买软件，而在其所购软件中，可直接使用的只占 2%，另外 3%需要做一些修改，其余 95%都成了垃圾。一句话，不管这些软件是否符合需求规格，它们都显然没有满足客户的需求。面向对象技术并没有给我们带来“神奇的效应”，不管开发商如何吹嘘面向对象 OO(Object-Oriented)工具多么万能，也不管那些 OO 狂热者多么毅然地前赴后继，这方面的数据从 20 世纪 80 年代以来并没有发生大的改观。

这实在令我们的软件工程专家和从业人员们羞愧，因为它揭示了我

---

* 编注：该序的作者是王计斌(Dave Wang)，清华大学博士，研究方向包括软件工程和集成产品开发(IPD)，长期从事 IPD/CMM 推行，创办软件工程研究与实践论坛(http://www.seforum.net)，现在华为技术有限公司工作。

们可能一开始就从根本上做错了什么！20 世纪 90 年代中期，当软件工程一代宗师 Michael Jackson(非歌坛巨星 Michael Jackson)宣布他们的研究结果时，立刻在软件工程界激起了阵阵涟漪。Jackson 指出，软件从业人员和方法学大师们只是简单地模仿和照搬其他学科的方法，却将最重要的方面(问题域)忽略了。他指出，面向对象方法和结构化方法对问题域的处理没有什么大的区别，却被人们过分地用美好的词汇美化了：

“...You can see the results clearly in many object-oriented modeling descriptions. Often they are accompanied by fine words about modeling the real world. But when you look closely you can see that they are really descriptions of programming objects, pure and simple. Any similarity to real-world objects, living or dead, is purely coincidental...”

(……从众多面向对象建模的描述中，你可以很清楚地看到这些恶果。而且它们还经常伴随着有关现实世界建模的非常美好的词汇。然而，仔细看看，你就会发现它们其实是彻头彻尾的编程对象！如果说有任何和现实世界对象相似的地方，不管是死是活，纯属巧合……)

回首软件工程近 40 年的发展，Jackson 哀叹软件行业普遍缺乏专业性，充满了业余人员，“手中有一个锤子，看到什么都是钉子”，谁都可以开发性命攸关的软件。

这就是我们面临的严峻而复杂的现实，也许您会感到震惊！然而在大师 Frederick P. Brooks 眼里，是那么的平静。因为早在 28 年前，他就在《人月神话》(*The Mythical Man-Month*)这本不朽著作中对这些内容做了深入论述。

这本小册子行文优美，思想博大精深，字字真言，精读之有不尽的趣味，藏之又是极珍贵的文献，名眼高人，自能鉴之。

前些年，一位朋友从印度归来，说此书在印度极为普及，我也动起笔来，但惭愧终未成正果。汪颖兄素来勤恳，明知此翻译为“success without applause, diligence without reward”(没有掌声的成功，没有回报的勤勉)，却兢兢业业，反复琢磨，历经单调、繁琐、艰辛的劳动，终于付梓。钦佩之余随即作序共勉。

Dave Wang
SE Forum China
2002 年 3 月于深圳

人月神话20周年纪念版序言

令我惊奇和高兴的是，《人月神话》在20年后仍然继续流行，印数超过了250 000册。人们经常问，我在1975年提出的观点和建议，哪些是我仍然坚持的，哪些是已经改变了的，又是怎样改变的？尽管我在一些讲座上也分析过这个问题，但我还是一直想把它写成文章。

Peter Gordon现在是Addison-Wesley的出版伙伴，他从1980年开始和我共事。他非常有耐心，对我帮助很大。他建议我们准备一个纪念版本。我们决定不对原版本做任何修订，只是原封不动地重印(除了一些无足轻重的修正)，并用更新的思想来扩充它。

第16章重印了一篇在1986年IFIPS会议上的论文“没有银弹：软件工程的根本和次要问题”(*No Silver Bullet: Essence and Accidents of Software Engineering*)。这篇文章来自我在国防科学委员会主持军用软件方面研究时的经验。我当时的研究合作者，也是我的执行秘书，Robert L. Patrick，他帮助我回想和感受那些做过的软件大项目。1987年，IEEE的《计算机》(*Computer*)杂志重印了这篇论文，使它传播得更广了。

“没有银弹”被证明是富有煽动性的，它预言10年内没有任何编程技巧能够给软件的生产率带来数量级上的提高。10年只剩下一年了，我的预言看来是安全了。“没有银弹”激起了越来越多文字上的剧烈争论，比《人月神话》还要多。因此在第17章，我对一些公开的批评做了说明，并更新了在1986年提出的观点。

在准备《人月神话》的回顾和更新时，一直在进行的软件工程研究和经验已经批评、证实或否定了少数书中断言的观点，也影响了我。剥去辅助的争论和数据后，把那些观点粗略地分类，对我来说是很有帮助的。我在第 18 章列出了这些观点的概要，希望这些单调的陈述能够引来争论和证据，然后得到证实、否定、更新或精炼。

第 19 章是一篇更新的短文。读者应该注意的是，新观点并不像原来的书一样来自我的亲身经历。我在大学里工作，而不是在工业界，做的是小规模的项目，而不是大项目。自 1986 年以来，我就只是教授软件工程，不再做这方面的研究。我现在的研究领域是虚拟环境及其应用。

在这次回顾的准备过程中，我找了一些正在软件工程领域工作的朋友，征求他们现在的观点。他们很乐意与我分享他们的想法，并仔细地对草稿提出了意见，这些都使我重新受到启发。感谢 Barry Boehm、Ken Brooks、Dick Case、James Coggins、Tom DeMarco、Jim McCarthy、David Parnas、Earl Wheeler 和 Edward Yourdon。感谢 Fay Ward 对新的章节进行了出色的技术加工。

感谢我在国防科学委员会军事软件工作组的同事 Gordon Bell、Bruce Buchanan、Rick Hayes-Roth，特别是 David Parnas，感谢他们的洞察力和生动的想法。感谢 Rebekah Bierly 对第 16 章的论文进行了技术加工。我把软件问题分成“根本的”和“次要的”，这是受 Nancy Greenwood Brooks 的启发，她在一篇“Suzuki 小提琴教育”的论文中应用了这样的分析方法。

在 1975 年版本的序言中，Addison-Wesley 出版社按规定不允许我在书中向该社的一些扮演了关键角色的员工致谢。可是，有两个人的贡献必须特别指出：执行编辑 Norman Stanton 和美术指导 Herbert Boes。Boes 设计了优雅风格的版式和封面，他在评注时特别提到：“页边的空白要宽，字体和版面要有想象力。”更重要的是，他提出了至关重要的建议：

为每一章的开头配一幅图片(当时我只有“焦油坑”和“兰斯大教堂”的图片)。寻找这些图片使我多花了一年的时间，但我永远感激这个忠告。

Soli Deo Gloria——愿神独得荣耀。

Frederick P.Brooks, Jr.
Chapel Hill, N.C.
1995 年 3 月

第 1 版序言

在很多方面，管理一个大型的计算机编程项目与管理其他行业的大型工程很相似——比大多数程序员所认为的还要相似；在另外一些方面，它又有差别——比大多数职业经理人所认为的差别还要大。

这个领域的知识在于累积。现在，AFIPS(美国信息处理学会联合会)已经有了一些讨论和会议，也出版了一些书籍和论文，但是还没有成形的方法对这一领域来进行系统地阐述。提供这样一本主要反映个人观点的小书看来是合适的。

虽然我原来从事计算机科学的编程方面的工作，但是在 1956—1963 年，自动控制程序和高级语言编译器开发出来的时候，我主要参加的是硬件构架方面的工作。1964 年，我成为操作系统 OS/360 的经理，我发现前些年的进展使编程世界改变了很多。

虽然是失败的，但管理 OS/360 的开发仍是一次很有帮助的经历。负责这次开发项目的团队，包括我的继任经理 F. M. Trapnell，有很多值得自豪的东西。该系统在设计和执行方面都很出色，并被成功地应用到很多领域，特别是设备独立的输入输出和外部库管理，在很多技术革新中被广泛复制。现在，这一系统是十分可靠的，相当有效且非常通用。

但是，并不是所有的努力都是成功的。所有 OS/360 的用户很快就能发现它应该能够做得更好。设计和执行上的缺陷在控制程序中特别普

遍，相比之下，语言编译器就好得多。大多数缺陷发生在1964—1965年的设计阶段，所以这肯定是我的责任。此外，这个产品发布推迟了，需要的内存比计划中的要多，成本也是估计的好几倍，而且第一次发布时并不能很好地运行，直到发布了几次以后，问题才得以解决。

按照当初接受OS/360任务时的协议，在1965年离开IBM后，我来到Chapel Hill。我开始分析OS/360的经验，看能不能从中学到什么管理和技术上的教训。特别要说明的是，System/360硬件开发和OS/360软件开发中的管理经验是大相径庭的。对Tom Watson关于为什么编程难以管理的探索性问题，这本书是一份迟来的答案。

在这次探索中，我和1964—1965年的经理助理R. P. Case，还有1965—1968年的经理F. M. Trapnell进行了长谈，从中受益很多。我还对比了其他大型编程项目经理的结论，这些项目经理包括M. I. T.的F. J. Corbato，贝尔电话实验室的V. Vyssotsky和John Harr，International Computers Limited的Charles Portman，苏联科学院西伯利亚分部计算实验室的A. P. Ershov和IBM的A. M. Pietrasanta。

我自己的结论体现在下面的文字中，送给专业程序员、职业经理，特别是程序员的职业经理。

虽然写出来的是各自独立的章节，但本书还是有一个中心的论点，特别包含在第2～7章。简言之，我相信由于人员的分工，大型编程项目碰到的管理问题和小项目碰到的管理问题区别很大；我相信关键需要的是维持产品自身的概念完整性。这几章探讨了其中的困难和解决的方法。而后续的章节则探讨了软件工程管理的其他方面。

这个领域的文献并不多，但散布很广。因此我尝试在书后给出了参考文献，说明某个特定知识点并指导感兴趣的读者去参阅其他有用的文献。很多朋友读过了本书的手稿，其中一些朋友还给出了很有帮助的意

见。这些意见很有价值，但为了不打乱文字的通顺，我把它们作为注解包含在本书中。

因为这本书是一部文集而不是一部教材，所有的参考文献和注解都被放到书的末尾，建议读者在读第一遍时略去不看。

深深地感谢 Sara Elizabeth Moore 小姐、David Wagner 先生和 Rebecca Burris 夫人，他们帮助我准备了手稿。感谢 Joseph C. Sloane 教授在图解方面的建议。

Frederick P. Brooks, Jr.
Chapel Hill, N.C.
1974 年 10 月

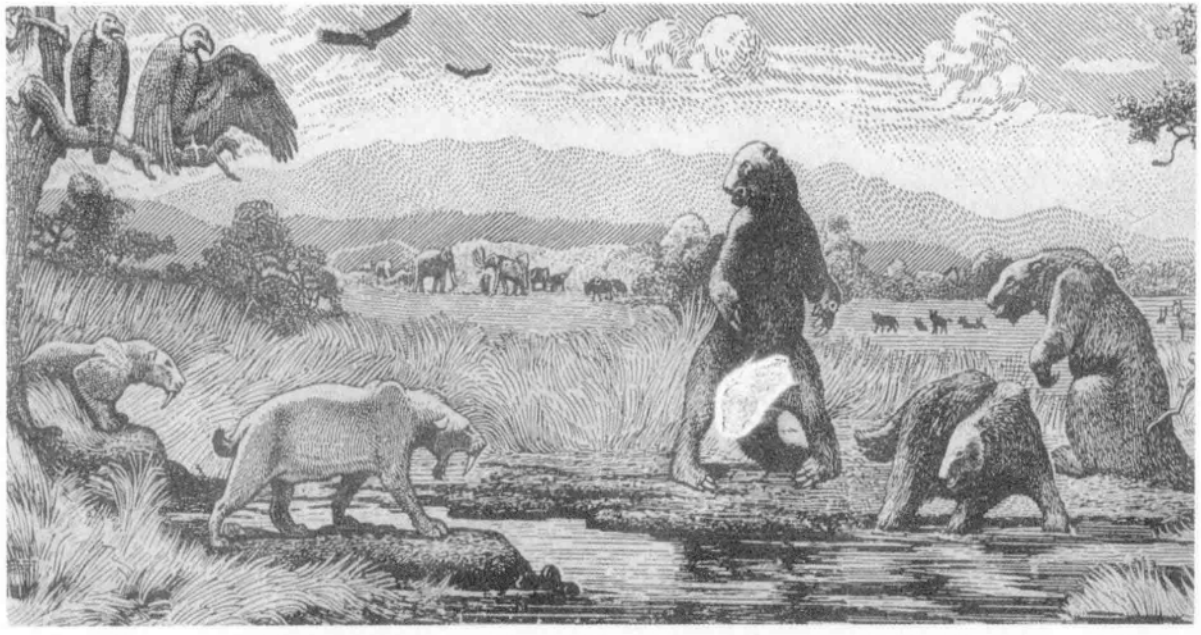
人月神话

目 录

第 1 章

焦 油 坑

The Tar Pit



C. R. 奈特的《拉布雷阿的焦油坑壁画》(*Mural of La Brea Tar Pits*)

注：拉布雷阿焦油坑是著名的旅游景点，位于美国洛杉矶化石博物馆。

资料来源：The George C. Page Museum of La Brea Discoveries, The Natural History Museum of Los Angeles County

前车之覆，后车之鉴。

——荷兰谚语

Een schip op het strand is een baken in zee.

[A ship on the beach is a lighthouse to the sea.]

DUTCH PROVERB

史前史中，没有别的场景比巨兽们在焦油坑中垂死挣扎的场面更令人震撼。上帝见证着恐龙、猛犸象、剑齿虎在焦油中挣扎。它们挣扎得越猛烈，焦油纠缠得就越紧，没有哪种猛兽足够强壮或具有足够的技巧，能够挣脱束缚，它们最后都沉到了坑底。

过去几十年的大型系统开发就犹如这样一个焦油坑，很多大型和强壮的动物在其中剧烈地挣扎。他们中大多数开发出了可运行的系统——不过只有极少数的项目满足了目标、进度和预算的要求。各种团队，大型的或小型的，庞杂的或精干的，一个接一个地淹没在了焦油坑中。表面上看起来好像没有任何一个单独的问题会导致困难，每个问题都能获得解决，但是当它们相互纠缠和累积在一起的时候，团队的行动就会变得越来越慢。对于问题的麻烦程度，每个人似乎都会感到惊讶，并且很难看清问题的本质。不过，如果我们想解决问题，就必须试图先去了解问题。

因此，首先让我们来认识一下系统开发这个职业，以及充满在这个职业中的乐趣和苦恼吧！

编程系统产品

报纸上经常会出现这样的新闻，讲述两个程序员如何在经过改造的简陋车库中，编出超过大型团队工作量的重要程序。接着，每个编程人员准备相信这样的神话，因为他知道自己能以超过产业化团队的 1 000 代码行/年的生产率来开发任何程序。

为什么不是所有的产业化队伍都会被这种专注的二人组合所替代？我们必须看一下产出的是什么。

图 1-1 的左上部分是**程序**(Program)。它本身是完整的，可以由作者在所开发的系统平台上运行。它通常是车库中产出的产品，以及作为单个程序员生产率的评估标准。

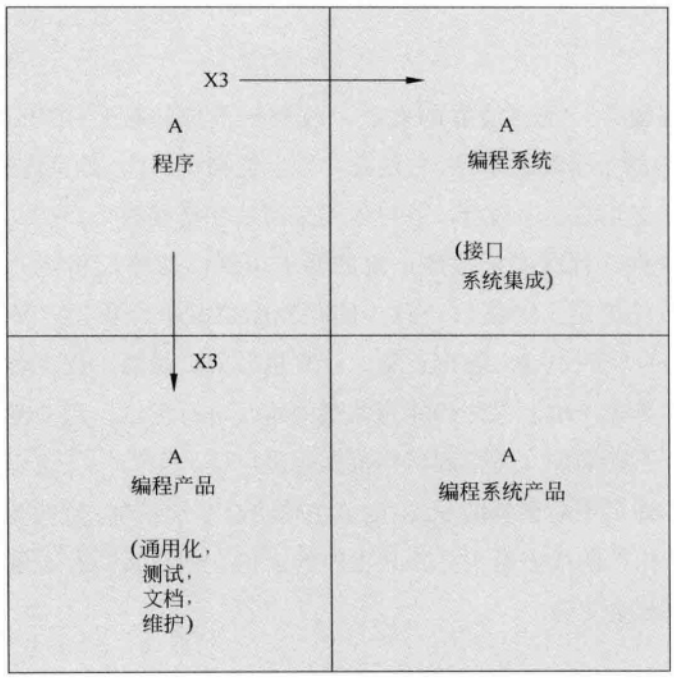


图 1-1 编程系统产品的演进

有两种途径可以使程序转变成更有用但是成本更高的产物，这两种途径表现为图中的边界。

水平边界以下，程序转变成**编程产品**(Programming Product)。这是可以被任何人运行、测试、修复和扩展的程序。它可以在多种操作系统平台上运行，供多套数据使用。要成为通用的编程产品，程序必须按照普遍认可的风格来编写，特别是输入的范围和形式必须广泛地适用于所有可以合理使用的基本算法。接着，对程序进行彻底测试，确保它的稳

定性和可靠性，使其值得信赖。这就意味着必须准备、运行和记录详尽的测试用例库，用来检查输入的边界和范围。此外，要将程序提升为程序产品，还需要有完备的文档，每个人都可以加以使用、修复和扩展。经验数据表明，相同功能的编程产品的成本，至少是已调试的程序的成本的 3 倍。

回到图中，垂直边界的右边，程序转变成编程系统(Programming System)中的一个构件单元。它是在功能上能相互协作、具有规范的格式、可以进行交互的程序集合，并可以用来组装和搭建整个系统。要成为编程系统构件，程序必须按照一定的要求编制，使输入和输出在语法和语义上与精确定义的接口一致。同时程序还要符合预先定义的资源限制——内存空间、输入输出设备、计算机时间。最后，程序必须同其他系统构件单元一道，以任何能想象到的组合进行测试。由于测试用例会随着组合不断增加，所以测试的范围必须广泛。因为一些意想不到的交互会产生许多不易察觉的 bug，测试工作将会非常耗时，因此相同功能的编程系统构件的成本至少是独立程序的 3 倍。如果系统有大量的组成单元，成本还会更高。

图 1-1 的右下部分代表**编程系统产品**(Programming Systems Product)。与以上的所有的简单的程序都不同的是，它的成本高达 9 倍。然而，只有它才是真正有用的产品，是大多数系统开发的目标。

职业的乐趣

编程为什么有趣？作为回报，它的从业者期望得到什么样的快乐？

首先，这种快乐是一种创建事物的纯粹快乐。如同小孩在玩泥巴时感到快乐一样，成年人喜欢创建事物，特别是自己进行设计。我想这种快乐是上帝创造世界的折射，一种呈现在每片独特的、崭新的树叶和雪花上的喜悦。

其次，这种快乐来自于开发对他人有用的东西。内心深处，我们期望我们的劳动成果能够被他人使用，并能对他们有所帮助。从这一角度而言，这同小孩用粘土为“爸爸的办公室”捏制铅笔盒没有任何本质的区别。

第三，快乐来自于整个过程体现出的一股强大的魅力——将相互啮合的零部件组装在一起，看到它们以精妙的方式运行着，并收到了预期的效果。比起弹球游戏机或自动电唱机所具有的迷人魅力，程序化的计算机毫不逊色。

第四，这种快乐是持续学习的快乐，它来自于这项工作的非重复特性。人们所面临的问题总有这样那样的不同，因而解决问题的人可以从中学习新的事物，有时是实践上的，有时是理论上的，或者兼而有之。

最后，这种快乐还来自于在易于驾驭的介质上工作。程序员，就像诗人一样，几乎仅仅在单纯的思考中工作。程序员凭空地运用自己的想象，来建造自己的“城堡”。很少有创造介质如此灵活，如此易于精炼和重建，如此容易实现概念上的设想(不过我们将会看到，容易驾驭的特性也有它自己的问题)。

然而程序毕竟同诗歌不同，它是实实在在的东西；它可以移动和运行，能独立产生可见的输出；它能打印结果，绘制图形，发出声音，移动支架。神话和传说中的魔术在我们的时代已变成现实。在键盘上键入正确的咒语，屏幕会活动、变幻，显示出前所未有的也不可能存在的事物。

编程的快乐在于它不仅满足了我们内心深处进行创造的渴望，而且还唤醒了每个人内心的情感。

职业的苦恼

然而这个过程并不全都是快乐的。我们只有事先了解一些编程固有的苦恼，这样，当它们真的出现时，才能更加坦然地面对。

首先，苦恼来自追求完美。因为计算机是以这样的方式来变戏法的：如果咒语中的一个字符、一个停顿，没有与正确的形式一致，魔术就不会出现(现实中，很少有人类活动会要求如此完美，所以人类对它本来就不习惯)。实际上，我认为，学习编程最困难的部分，是将做事的方式向追求完美的方向调整[1]。

其次，苦恼来自由他人来设定目标、供给资源和提供信息。编程人员很少能控制工作环境和工作目标。用管理的术语来说，个人的权威和他所承担的责任是不相配的。不过，似乎在所有的领域中，对要完成的工作，很少能提供与责任相一致的正式权威。而现实情况中，实际(相对于形式)的权威来自于每次任务的完成。

对于系统编程人员而言，对其他人的依赖是一件非常痛苦的事情。他依靠其他人的程序，而这些程序往往设计得并不合理、实现拙劣、发布不完整(没有源代码或测试用例)或者文档记录得很糟。所以，系统编程人员不得不花费时间去研究和修改，而它们在理想情况下本应该是可靠的、完整的。

下一个苦恼——概念性设计是有趣的，但寻找琐碎的 bug 却是一项重复性的活动。伴随着创造性活动的，往往是枯燥沉闷的时间和艰苦的劳动。程序编制工作也不例外。

另外，人们发现调试和查错往往是线性收敛的，或者更糟糕的是，具有二次方的复杂度。结果，测试一拖再拖，寻找最后一个错误比第一个错误将花费更多的时间。

最后一个苦恼，有时也是一种无奈——当投入了大量辛苦的劳动，产品在即将完成或者终于完成的时候，却已显得陈旧过时。可能是同事和竞争对手已在追逐新的、更好的构思；也许替代方案不仅仅是在构思，而且已经在安排了。

现实情况比上面所说的通常要好一些。当产品开发完成时，更优秀的新产品通常还不能投入使用，而仅仅是为大家谈论而已。另外，它同样需要数月的开发时间。事实上，只有实际需要时，才会用到最新的设想，因为所实现的系统已经能满足要求，并体现了回报。

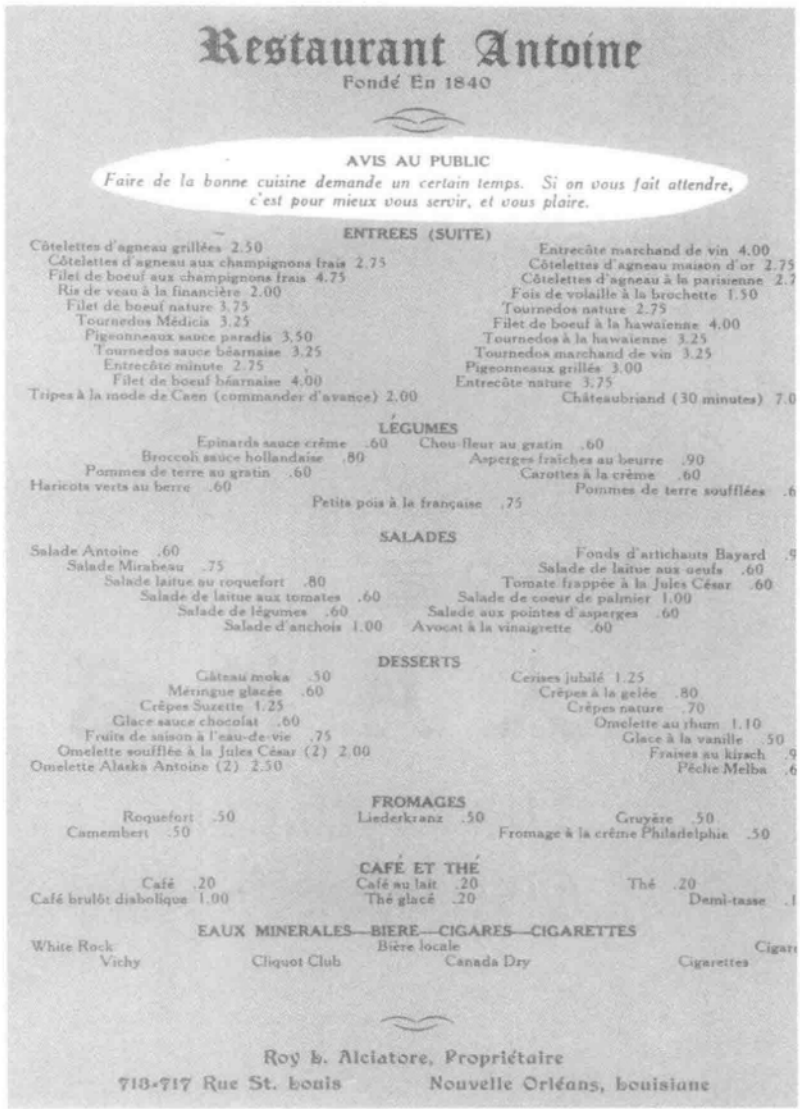
诚然，产品开发所基于的技术在不断地进步。一旦设计被冻结，在概念上就已经开始陈旧了。不过，实际产品需要一步一步按阶段实现。实现落后与否的判断应根据其他已有的系统，而不是未实现的概念。因此，我们所面临的挑战和任务是在实际的进度和有效的资源范围内，寻找解决实际问题的切实可行方案。

这，就是编程，一个许多人痛苦挣扎的焦油坑以及一种乐趣和苦恼共存的创造性活动。对许多人而言，其中的快乐远远大于苦恼。本书的以下章节将试图搭建一些桥梁，为通过这样的焦油坑提供一些指导。

第2章

人月神话

The Mythical Man-Month



Restaurant Antoine

Fondé En 1840

AVIS AU PUBLIC

Faire de la bonne cuisine demande un certain temps. Si on vous fait attendre, c'est pour mieux vous servir, et vous plaire.

ENTREES (SUITE)

Côtelettes d'agneau grillées 2.50
Côtelettes d'agneau aux champignons frais 2.75
Filet de boeuf aux champignons frais 4.75
Ris de veau à la financière 2.00
Filet de boeuf nature 3.75
Tournedos Médicis 3.25
Pigeonneaux sauce paradis 3.50
Tournedos sauce béarnaise 3.25
Entrecôte minute 2.75
Filet de boeuf béarnaise 4.00
Tripes à la mode de Caen (commander d'avance) 2.00
Entrecôte marchand de vin 4.00
Côtelettes d'agneau maison d'or 2.75
Côtelettes d'agneau à la parisienne 2.7
Fois de volaille à la brochette 1.50
Tournedos nature 2.75
Filet de boeuf à la hawaienne 4.00
Tournedos à la hawaienne 3.25
Tournedos marchand de vin 3.25
Pigeonneaux grillés 3.00
Entrecôte nature 3.75
Châteaubriand (30 minutes) 7.0

LÉGUMES

Epinards sauce crême .60
Broccoli sauce hollandaise .80
Pommes de terre au gratin .60
Haricots verts au beurre .60
Chou-fleur au gratin .60
Asperges fraiches au beurre .90
Carottes à la crème .60
Pommes de terre soufflées .6
Petits pois à la française .75

SALADES

Salade Antoine .60
Salade Mirabeau .75
Salade laitue au roquefort .80
Salade de laitue aux tomates .60
Salade de légumes .60
Salade d'anchois 1.00
Fonds d'artichauts Bayard .9
Salade de laitue aux oeufs .60
Tomate frappée à la Jules César .60
Salade de coeur de palmier 1.00
Salade aux pointes d'asperges .60
Avocat à la vinaigrette .60

DESSERTS

Gâteau moka .50
Méringue glacée .60
Crêpes Suzette 1.25
Glace sauce chocolat .60
Fruits de saison à l'eau-de-vie .75
Omelette soufflée à la Jules César (2) 2.00
Omelette Alaska Antoine (2) 2.50
Cerises jubilé 1.25
Crêpes à la gelée .80
Crêpes nature .70
Omelette au rhum 1.10
Glace à la vanille .50
Fraises au kirsch .9
Pêche Melba .6

FROMAGES

Roquefort .50
Camembert .50
Liederkranz .50
Gruyère .50
Fromage à la crême Philadelphie .50

CAFÉ ET THÉ

Café .20
Café brulôt diabolique 1.00
Café au lait .20
Thé glacé .20
Thé .20
Demi-tasse .1

EAUX MINERALES—BIERE—CIGARES—CIGARETTES

White Rock
Vichy
Bière locale
Cliquot Club
Canada Dry
Cigar
Cigarettes

Roy b. Alciatore, Propriétaire

718-717 Rue St. louis Nouvelle Orléans, louisiane

新奥尔良市安托万餐厅的菜单

注：安托万餐厅成立于 1840 年。本章的序言即来自这张菜单。

美食的烹调需要时间；片刻等待，更多美味，更多享受。

——新奥尔良市安托万餐厅的菜单

Good cooking takes time. If you are made to wait, it is to serve you better, and to please you.

MENU OF RESTAURANT ANTOINE, NEW ORLEANS

在众多软件项目中，缺乏合理的进度安排是造成项目滞后的最主要原因，它比其他所有因素加起来的影响还要大。导致这种灾难如此普遍的原因是什么呢？

首先，我们对估算技术缺乏有效的研究，更加严肃地说，它反映了一种悄无声息但并不真实的假设——一切都将运作良好。

第二，我们采用的估算技术隐含地假设人和月可以互换，错误地将进度与工作量相互混淆。

第三，由于对自己的估算缺乏信心，软件经理通常不会有耐心持续地估算这项工作。

第四，对进度缺少跟踪和监督。在其他工程领域中，经过验证的跟踪技术和常规监督程序，在软件工程中常常被认为是大胆的革新。

第五，当意识到进度的偏移时，下意识(以及传统)的反应是增加人力。这就像使用汽油灭火一样，只会使事情更糟。越来越大的火势需要更多的汽油，从而进入了一场注定会导致灾难的循环。

进度监督是另一篇论文的主题，而本文中我们将对这一问题的其他方面进行更详细的讨论。

乐观主义

所有的编程人员都是乐观主义者。可能是这种现代魔术特别吸引那些相信美满结局和幻想中的圣母的人；也可能是成百上千琐碎的挫折赶走了大多数人，只剩下了那些习惯上只关注结果的人；还可能仅仅因为计算机还很年轻，程序员更加年轻，而年轻人总是些乐观主义者——无

论是什么样的程序，结果是毋庸置疑的：“这次它肯定会运行”，或者“我刚刚找出最后一个错误”。

所以系统编程的进度安排背后的第一个错误的假设是：**一切都将运作良好，每一项任务仅花费它所“应该”花费的时间。**

对这种弥漫在编程人员中的乐观主义，理应受到慎重的分析。Dorothy Sayers 在她的《创造者的思想》(*The Mind of the Maker*)一书中，将创造性活动分为三个阶段：构思、实现和交流。书籍、计算机或者程序的存在，首先是作为一个构思或模型出现在作者的脑海中，它与时间和空间无关。接着，借助钢笔、墨水和纸，或者电线、.硅片和铁氧体，在现实的时间和空间中实现它们。然后，当某人阅读书籍、使用计算机和运行程序的时候，他与作者的思想相互沟通，创造过程从而得以结束。

以上 Sayers 的阐述不仅仅可以描绘人类的创造性活动，而且类似于“基督三位一体的教义”，能指导我们的日常工作。对于创造者，只有在实现的过程中，才能发现我们构思的不完整性和不一致性。因此，对于理论家而言，书写、试验以及“工作实现”是非常基本和必要的。

在许多创造性活动中，往往很难掌握活动实施的介质，例如木头切割、油漆、电气接线等。这些介质的物理约束限制了思路的表达，它们同样对实现造成了许多预料之外的困难。

由于物理介质和思路中隐含的不完善性，实现起来需要花费大量的时间和汗水。对遇到的大部分实现上的困难，我们总是倾向于去责怪那些物理介质，因为它们不顺应“我们”设定的思路。其实，这只不过是我们的骄傲使判断带上了主观主义色彩。

然而，计算机编程基于十分容易掌握的介质，编程人员通过非常纯粹的思维活动——概念以及灵活的表现形式来开发程序。正是由于介质

的易于驾驭，我们期待在实现过程中不会碰到困难，因此造成了乐观主义的弥漫。而我们的构思是有缺陷的，因此总会发现 bug。也就是说，我们的乐观主义并不应该是理所应当的。

在单个的任务中，“一切都将运转正常”的假设在进度上具有可实现性。因为所遇到的延迟是一个概率分布曲线，“不会延迟”具有限定的概率，所以现实情况可能会像计划安排的那样顺利。然而大型的编程工作，或多或少包含了很多任务，某些任务间还具有前后的次序，从而一切正常的概率变得非常小，甚至接近于零。

人月

第二个谬误的思考方式是在估计和进度安排中使用的工作量单位：人月。成本的确随开发产品的人数和时间的不同，有着很大的变化，进度却不是如此。人员和时间的关系如图 2-1 所示。因此我认为**用人月作为衡量一项工作的规模是一个危险和带有欺骗性的神话**。它暗示着人员数量和时间是可以相互替换的。

人数和时间的互换仅仅适用于以下情况：某个任务可以分解给参与人员，并且**他们之间不需要相互的交流**(见图 2-1)。这在割小麦或收获棉花的工作中是可行的；而在系统编程中近乎不可能。

当任务由于次序上的限制不能分解时，人手的添加对进度没有帮助(见图 2-2)。无论哪位母亲，孕育一个生命都需要 10 个月。由于除错的次序特性，许多软件开发工作都具有这种特征。

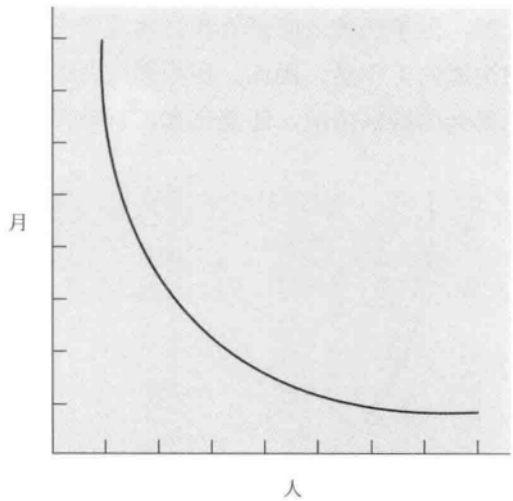


图 2-1 人员和时间之间的关系——完全可以分解的任务

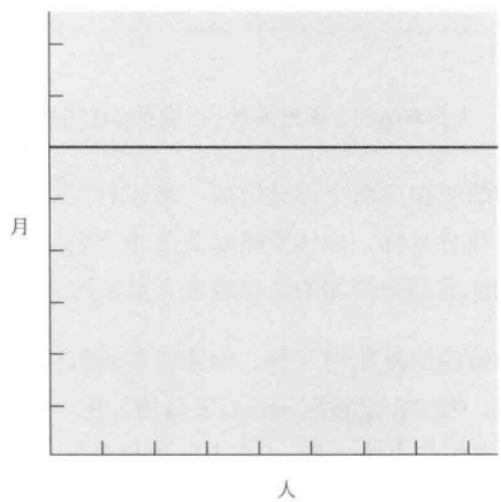


图 2-2 人员和时间之间的关系——无法分解的任务

对于可以分解，但子任务之间需要相互沟通和交流的任务，必须在计划工作中考虑沟通的工作量。因此，在相同人月的前提下，采用增加人手来减少时间得到的最好情况，还是比未调整前差一些(见图 2-3)。

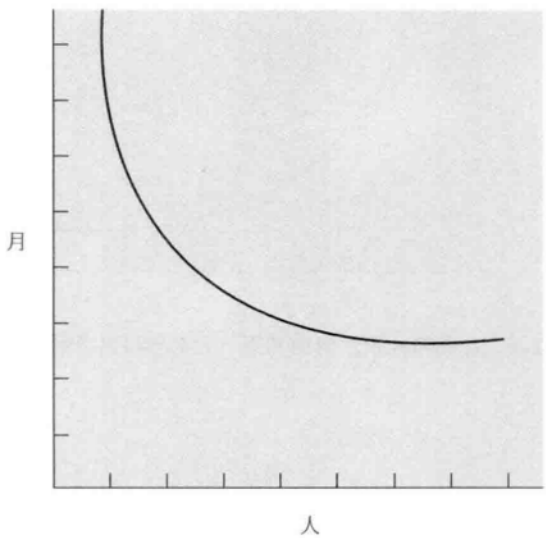


图 2-3　人员和时间之间的关系——需要沟通的可分解任务

沟通所增加的负担由两个部分组成：培训和相互的交流。每个成员需要进行技术、项目目标、总体策略以及工作计划的培训。这种培训是不能分解的，因此这部分增加的工作量随人员的数量呈线性变化[1]。

相互之间交流的情况更糟一些。如果任务的每个部分必须分别与其他部分单独协作，则工作量按照 $n(n\text{-}1)/2$ 递增。在一对一交流的情况下，3 个人的工作量是 2 个人的 3 倍，4 个人的工作量则是 2 个人的 6 倍。而对于需要在三、四个人之间召开会议、进行协商、一同解决的问题，情况会更加恶劣。所增加的用于沟通的工作量可能会完全抵消对原有任务分解所产生的作用，此时我们会被带到如图 2-4 所示的境地。

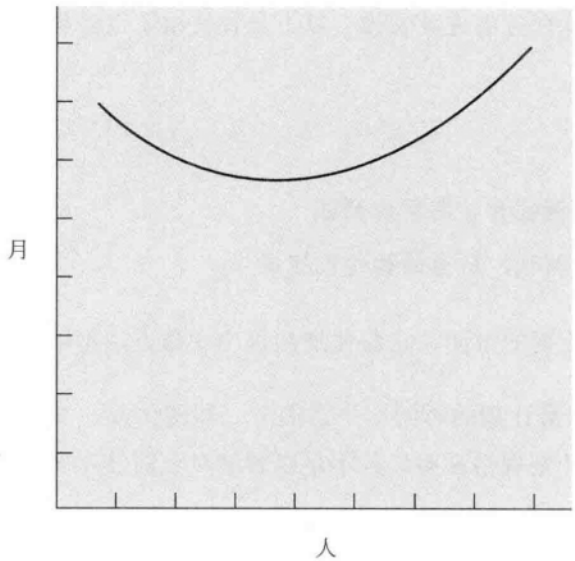


图 2-4 人员和时间之间的关系——关系错综复杂的任务

因为软件开发本质上是一项系统工作——错综复杂关系下的一种实践，沟通、交流的工作量非常大，它很快会消耗任务分解所节省下来的个人时间。从而，添加更多的人手，实际上是延长了而不是缩短了时间进度。

系统测试

在进度安排中，由于顺序限制所造成的影响，没有哪个部分比单元调试和系统测试所受到的牵涉更彻底。而且，需要的时间依赖于所遇到的错误、缺陷的数量及其难以捕捉的程度。理论上，缺陷的数量应该为零。但是，由于我们的乐观主义，通常实际出现的缺陷数量比预料的要多得多。因此，系统测试进度的安排常常是编程中最不合理的部分。

对于软件任务的进度安排，以下是我使用了很多年的经验法则：

1/3 计划

1/6 编码

1/4 构件测试和早期系统测试

1/4 系统测试，所有的构件已完成

在许多重要的方面，它与传统的进度安排方法不同：

(1) 分配给计划的时间比平常的多。即便如此，其只是勉强产生详细和稳定的计划规格说明，并不足以容纳对全新技术的研究和探索；

(2) 对所完成代码的调试和测试投入近一半的时间，这比平常的安排多很多；

(3) 容易估计的部分，如编码，仅仅分配了 1/6 的时间。

通过对传统项目进度安排的研究，我发现很少有项目允许为测试分配一半的时间，但大多数项目的测试实际上是花费了进度中一半的时间。它们中的许多项目，在系统测试之前还能保持进度。或者说，除了系统测试，进度基本能够保证[2]。

特别需要指出的是，不为系统测试安排足够的时间简直就是一场灾难。因为延迟发生在项目快完成的时候。直到接近项目的发布日期，才有人发现进度上的问题。因此，坏消息没有任何预兆，很晚才出现在客户和项目经理面前。

另外，此时此刻的延迟具有不寻常的、严重的财务和心理上的反应。在此之前，项目已经配置了充足的人员，每天的人力成本也已经达到了最大的限度。更为严重的是，在用软件支持其他的商业活动(计算机硬件运

送、新设备操作等)的情况下，若在即将发布时出现延误，所付出的二次商业代价是非常高昂的。实际上，上述的二次成本远远高于其他开销。因此，在早期进度策划时，允许充分的系统测试时间是非常重要的。

空泛的估算

观察一下编程人员，你可能会发现，同厨师一样，某项任务的计划进度，可能受限于顾客要求的紧迫程度，但紧迫程度无法控制实际的完成情况。就像约好在两分钟内完成一个煎蛋，看上去可能进行得非常好，但当它无法在两分钟内完成时，顾客只能选择等待或者生吃煎蛋。软件顾客的情况类似。

厨师还有其他的选择：他可以把火开大，不过结果常常是无法“挽救”的煎蛋——一面已经焦了，而另一面还是生的。

现在，我并不认为软件经理内在的勇气和坚定不如厨师，或者不如其他工程经理。但为了满足顾客期望的日期而造成的不合理进度安排，在软件领域中却比其他任何工程领域要普遍得多。而且，非阶段化方法的采用，少得可怜的数据支持，加上完全借助软件经理的直觉，这样的方式很难生产出有力的、看似可靠的和规避风险的估计。

显然我们需要两种解决方案。开发并推行生产率图表、缺陷率图表、估算规则等，而整个组织最终会从这些数据的共享上获益。

或者，在基于可靠基础的估算出现之前，项目经理需要挺直腰杆，坚持他们的估计，确信自己的经验和直觉总比从期望派生出的结果要强得多。

重复产生的进度灾难

当一个软件项目落后于进度时，通常的做法是什么呢？自然是加派人手。如图 2-1、图 2-2、图 2-3、图 2-4 所示，这可能有所帮助，却可能无法解决问题。

我们来考虑一个例子[3]。设想一个估计需要 12 个人月的任务，分派给 3 个成员在 4 个月的时间内完成，并在每个月的末尾安排了可测量的里程碑 A、B、C、D(见图 2-5)。

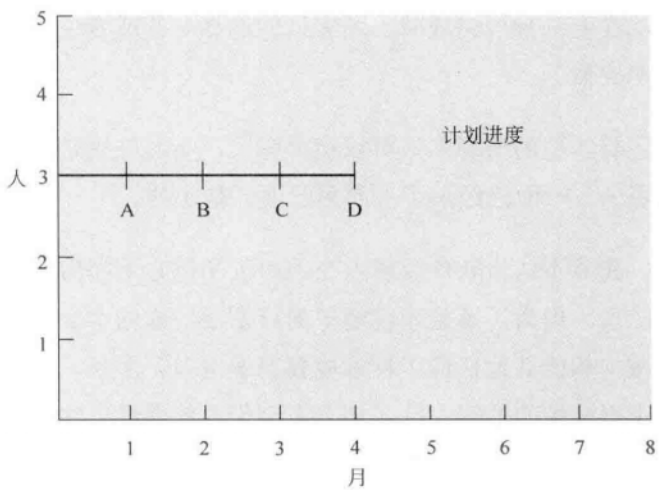


图 2-5　4 个里程碑

现在假定两个月之后，第一个里程碑没有达到(见图 2-6)。项目经理面对的选择方案有哪些呢？

(1) 假设任务必须按时完成。假设仅仅是任务的第一个部分估计不得当，如图 2-6 所示，则剩余了 9 个人月的工作量，时间还有两个月，即需要 4.5 个开发人员，所以需要在原来 3 个人的基础上增加 2 个人。

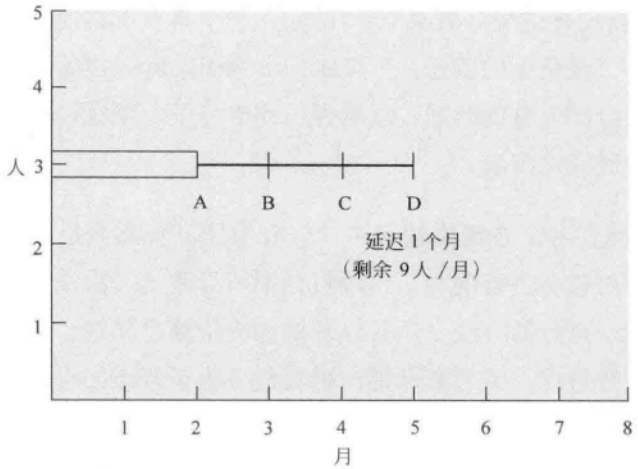


图 2-6 2个月后，第一个里程碑没有达到

(2) 假设任务必须按时完成。假设整个任务的估计偏少，如图 2-7 所示，那么还有 18 个人月的工作量，这样就需要 2 个月的时间和 9 个人，比原来计划的 3 人多了 6 人。

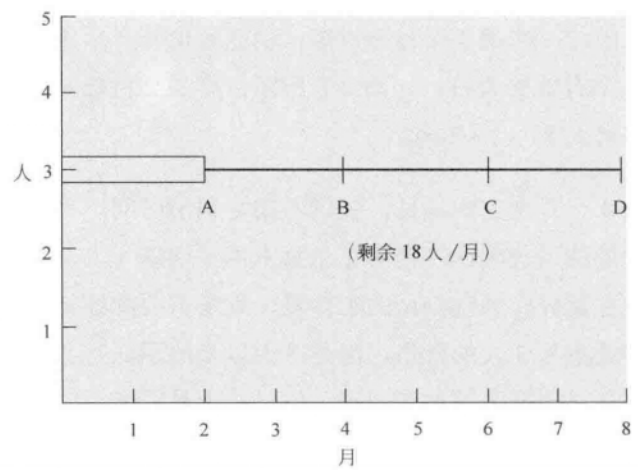


图 2-7 估计偏少的情况

(3) 重新安排进度。我喜欢 P. Fagg，一个具有丰富经验的硬件工程师的忠告："避免小的偏差。"(Take no small slips.)也就是说，在新的进度安排中分配充分的时间，以确保工作能仔细、彻底地完成，从而无需重新确定时间进度表。

(4) 削减任务。在现实情况中，当开发团队观察到进度的偏差时，总是倾向于对任务进行削减。当项目延期所导致的二次成本非常高时，这常常是唯一可行的方法。项目经理的相应措施是仔细、认真地调整项目，重新安排进度；或者默默地注视着任务由于轻率的设计和不完整的测试而被剪除。

在前两种情况下，坚持把不经调整的任务在 4 个月内完成将是灾难性的。考虑到重复生成的工作量，以第一种为例(见图 2-8)——不论在多么短的时间内，聘请到两名多么能干的新员工，他们都需要接受一位有经验的职员的培训。如果培训需要一个月的时间，那么 **3 个人月将会投入到原有进度安排以外的工作中**。另外，原先划分为 3 个部分的工作，会重新分解成 5 个部分；某些已经完成的工作必定会丢失，系统测试必然被延长。因此，在第 3 个月的月末，仍然残留着 7 个人月的工作，但此时只有 5 个有效的人月。如图 2-8 所示，产品交付还是会延期，如同没有增加任何人手(见图 2-6)。

期望在 4 个月内完成项目，仅仅考虑培训的时间，不考虑任务的重新划分和额外的系统测试，在第 2 个月末需要增添 4 人，而不是 2 人。如果考虑任务划分和系统测试的工作量，则还需要继续增加人手。到那时所拥有的就不是 3 人的队伍，而是 7 人以上的团队；并且小组的组织和任务的划分在类型上都不尽相同，这已经不是程度上的差异问题。

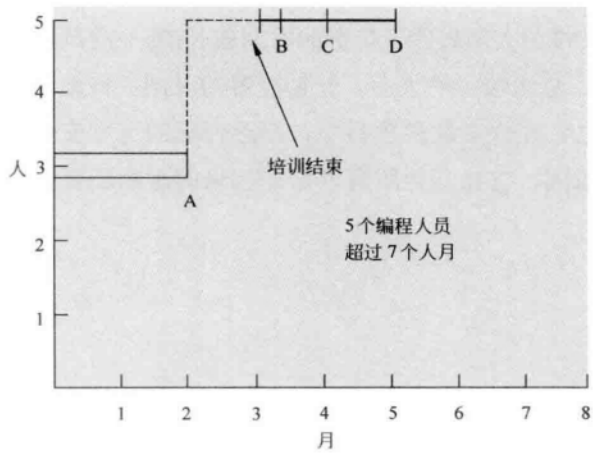


图 2-8 重复的工作量

注意在第 3 个月的结尾时，情况看上去非常糟。尽管已经做出了所有的管理方面的努力，3 月 1 日的里程碑仍未达到。此时，对项目经理而言，仍然存在着很强的诱惑——添加更多人力，结果往往会是上述情况的循环重复。这简直就是一种疯狂、愚蠢的做法。

前面的讨论仅仅是第一个里程碑估计不当的情况。如果在 3 月 1 日，项目经理做出了比较保守的假设，即整个估计过于乐观了，如图 2-7 所示，6 个人手需要添加到原先的任务中。这里，我们把培训、任务的重新分配、系统测试工作量的计算作为练习留给读者。但是毫无疑问，重复“灾难”所开发出的产品，比没有增加人手，而是重新安排开发进度所产生的产品更差。

冒昧地简化一下 Brooks 法则：

向进度落后的项目中增加人手，只会使进度更加落后。(Adding manpower to a late software project makes it later.)

这就是除去了神话色彩的人月。项目的时间依赖于顺序上的限制，人员的最大数量依赖于独立子任务的数量。从这两个数值可以推算出进

度表，该表安排的人员较少，花费的时间较长(唯一的风险是产品可能会过时)。相反，分派较多的人手，计划较短的时间，将无法得到可行的进度安排。总之，在众多软件项目中，缺乏合理的进度安排是造成项目滞后的最主要原因，它比其他所有因素加起来的影响还要大。

CHAPTER 3

THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 3 章

外科手术队伍

The Surgical Team

外科手术现场

资料来源：UPI[United Press International] Photo/The Bettman Archive

这些研究表明，效率高和效率低的实施者之间个体差异非常大，经常能够达到数量级的水平。

—— *Sackman, Erikson* 和 *Grant*[1]

These studies revealed large individual differences between high and low performers, often by an order of magnitude.

SACKMAN, ERIKSON, AND GRANT[1]

在计算机领域的会议中，常常听到年轻的软件经理声称，他们喜欢由一流人才组成的小型、精干的队伍，而不是那些几百人的大型团队，这里的“人”当然暗指平庸的程序员。其实我们也经常有相同的看法。

但这种幼稚的观点回避了一个很困难的问题——如何在有意义的进度安排内创建大型的系统？那么就让我们来仔细讨论一下这个问题。

问题

软件经理很早就认识到优秀程序员和较差程序员之间生产率的差异，但实际测量出的差异令我们所有人吃惊。在他们的一个研究中，Sackman、Erikson 和 Grant 曾对一组具有经验的程序人员进行测量。在该小组中，最好的和最差的表现在生产率上平均为 10∶1；在编程速度和空间上具有 5∶1 的惊人差异！简言之，20 000 美元/年的程序员的生产率可能是 10 000 美元/年的程序员的 10 倍。反之亦然。数据显示，经验和实际的表现没有相互联系(我怀疑这种现象是否普遍成立)。

我常常重复这样的一个观点，需要协作沟通的人员数量影响着开发成本，因为成本的主要组成部分是相互的沟通和交流，以及更正沟通不当所引起的不良结果(系统调试)。这一点，也暗示系统应该由尽可能少的人员来开发。实际上，绝大多数大型编程系统的经验显示出，一拥而上的开发方法是高成本的、速度缓慢的、低效的，开发出的是无法在概念上进行集成的产品。OS/360、Exec 8、Scope 6600、Multics、TSS、SAGE 等——这个列表可以不断地继续下去。

得出的结论很简单：如果在一个 200 人的项目中，有 25 个最能干和最有开发经验的项目经理，那么开除剩下的 175 名程序员，让项目经理来编程开发。

现在我们来验证一下这个解决方案。一方面，这个开发队伍不是通常所说的不超过 10 个人的、理想的小型精干的队伍，该团队的规模如此之大，以至于至少需要两个层级的管理，或者大约 5 名管理人员。另外，它需要额外的财务、人员、空间、文秘和机器操作方面的支持。

另一方面，如果采用一拥而上的开发方法，那么原有的 200 人的队伍仍然不足以开发真正的大型系统。例如，在 OS/360 项中，当项目进行到顶峰时，有超过 1 000 人在为它工作——程序员、文档编制人员、操作人员、职员、秘书、管理人员、支持小组等。从 1963 年到 1966 年，设计、编码和文档工作花费了大约 5 000 个人年。如果人月可以等量置换的话，我们所假设的 200 人队伍则需要 25 年的时间，才能使产品达到现有的水平。

这就是小型、精干队伍概念上的问题：**对于真正意义上的大型系统，它太慢了**。设想 OS/360 的工作由一个小型、精干的团队来解决，譬如一个 10 人团队。作为一个尺度，假设他们都非常能干，比一般的编程人员在编程和文档方面的生产率高 7 倍。同时假设 OS/360 原有开发人员是一些平庸的编程人员(这与实际情况相差很远)。同样作为一个尺度，假设另一个生产率的改进因子提高了 7 倍，因为较小的队伍所需的沟通和交流较少。同时假设同样的队伍完成的是同样的工作。那么，5 000/(10×7×7)= 10，他们需要 10 年来完成 5 000 人年的工作。一个产品在最初设计的 10 年后才出现，还有人会对它感兴趣吗？或者它是否会随着软件开发技术的快速进步，而显得过时呢？

这种进退两难的境地是非常残酷的。对于效率和概念的完整性来说，最好由少数干练的人员来设计和开发，而对于大型系统，则需要大量的人手，以使产品能在时间上满足要求。如何调和这两方面的矛盾呢？

Mills 的建议

Harlan Mills 的提议提供了一个崭新的、创造性的解决方案[2, 3]。Mills 建议大型项目的每一个部分由一个团队解决，但是该队伍以类似外科手术的方式组建，而并非一拥而上。也就是说，同每个成员截取问题某个部分的做法相反，由一个人来完成问题的分解，其他人给予他所需要的支持，以提高效率和生产力。

请简单地考虑一下，如果上述概念能够实施，似乎它可以满足迫切性的需要。很少的人员被包含在设计和开发中，其他许多人来进行工作的支持。它是否可行呢？谁是编程队伍中的麻醉医生和护士，工作如何划分？让我们继续使用医生的比喻：如果考虑所有可能想到的工作，这样的队伍应该如何运作？

外科医生。Mills 称之为**首席程序员**。他亲自定义功能和性能技术说明书，设计程序，编制源代码，测试以及书写技术文档。他使用例如 PL/I 的结构化编程语言，拥有对计算机系统的访问能力；该计算机系统不仅能够进行测试，还能够存储程序的各种版本，以允许简单的文件更新，并对其文档提供文本编辑能力。首席程序员需要极高的天分、丰富的经验和应用数学、业务数据处理或其他方面的大量系统知识和应用知识。

副手。他是外科医生的后备，能完成任何一部分工作，但是相对具有的经验较少。他的主要作用是作为设计的思考者、讨论者和评估人员。外科医生试图和他沟通设计，但不受到他建议的限制。副手经常在与其他团队讨论有关功能和接口问题时代表自己的小组。他需要详细了解所有的代码，研究设计策略的备选方案。显然，他充当外科医生的保险机制。他甚至可能编制代码，但对代码的任何部分，不承担具体的开发职责。

管理员。他是外科医生的老板，必须在人员、薪酬、办公空间等方面具有决定权，但他绝对不能在这些事务上浪费任何时间。因而，他需要一个控制财务、人员、工作地点和办公设备的专业管理人员，该管理员充当与组织中其他管理机构的接口。Baker 建议，仅在项目具有法律、合同、报表和财务方面的需求时，管理员才具有全职责任。否则，一个管理员可以为两个团队服务。

编辑。外科医生负责文档的生成——出于最大透明度的考虑，他必须创建各种文档，无论是对内部描述还是外部描述。而编辑根据外科医生的草稿或者口述，进行分析和重新组织，提供各种参考信息和书目，对多个版本进行维护，并监督文档生成的机制。

两个文秘。管理员和编辑每个人需要一个文秘。管理员的文秘负责非产品文件和使项目协作一致。

程序职员。他负责维护编程产品库中所有团队的技术记录。该职员接受文秘性质的培训，承担机器码文件和可读文件的相关管理责任。

所有的计算机输入汇集到这个职员处。如果需要，他会对它们进行记录或者标识。输出列表会提交给程序职员，由他进行归档和编制索引。另外，他负责将任何模型的最新运行情况记录在状态日志中，而所有以前的结果则按时间顺序进行归档保存。

Mills 概念的真正关键是“从个人艺术到公共实践”的编程观念转换。它向所有的团队成员展现了所有计算机的运行和产物，并将所有的程序和数据看做是团队的所有物，而非私人财产。

程序职员的专业化分工，使程序员从文书等杂事中解放出来，同时还可以对那些经常被忽视的杂事进行系统整理，确保了它们的质量，并强化了团队最有价值的财富——工作产品。上述概念显然考虑的是批处

理程序。当使用交互式终端，特别是在没有纸张输出的情况下，程序职员的职责并未消失，只是有所更改。他会记录小组程序和私有工作拷贝之间的更新，依然控制所有程序的运行，并使用自己的交互式工具来控制产品逐步增长的完整性和有效性。

工具维护人员。现在已经有很多文件编辑、文本编辑和交互式调试等工具，因此团队很少再需要自己的机器和机器操作人员。但是这些工具使用起来必须毫无疑问地具备令人满意的反应和可靠性。外科医生则是对这些工具的服务是否充分可用的唯一评判人员。他需要一个工具维护人员，保证所有基本服务的可靠性，以及承担团队成员所需要的特殊工具(特别是交互式计算机服务)的构建、维护和升级责任。即使已经拥有非常卓越的、可靠的集中式服务，每个团队仍然要有自己的工具维护人员。因为他的工作是检查**他的**外科医生所需要的工具，而不是其他团队的需要。工具维护人员常常要开发一些实用程序，编制具有目录的函数库以及宏库。

测试人员。外科医生需要大量合适的测试用例，用来对他所编写的工作片段，以及对整个工作进行测试。因此，测试人员既是为他的各个功能设计系统测试用例的对手，也是为他的日常调试设计测试数据的助手。他还负责计划测试的步骤和为单元测试搭建测试平台。

语言专家。随着 **Algol** 语言的出现，人们开始认识到，在大多数计算机项目中，总有一两个乐于掌握复杂编程语言的人。这些专家是非常有帮助的，大家会向他咨询。这些天才不同于外科医生，外科医生主要是系统设计者以及考虑系统的整体表现。而语言专家则寻找一种简洁、有效的使用语言的方法来解决复杂、晦涩或者棘手的问题。他通常需要对技术进行一些研究(2～3 天)。通常一个语言专家可以为 2～3 个外科医生服务。

以上就是如何参照外科手术队伍对 10 人的编程团队进行专业化的角色分工。

如何运作

文中定义的开发团队在很多方面满足了迫切性的需要。10 个人，其中 7 个专业人士在解决问题，而系统是一个人或者最多两个人思考的产物，因此其在客观上达到了概念的一致性。

要特别注意传统的两人队伍与“外科医生—副手”团队架构之间的区别。首先，传统的队伍将工作进行划分，每人负责一部分工作的设计和实现。在外科手术团队中，外科医生和副手都了解所有的设计和全部的代码。这节省了空间分配、磁盘访问等的劳动量，同时也确保了工作概念上的完整性。

第二，在传统的队伍中大家是平等的，出现观点的差异时，不可避免地需要讨论和进行相互的妥协和让步。由于工作和资源的分解，不同的意见会造成策略和接口上的不一致，例如谁的空间会被用做缓冲区，而事实上最终它们必须整合在一起。而在外科手术团队中，不存在利益的差别，观点的不一致之处可以由外科医生单方面来统一。这两种团队组建上的差异——对问题不进行分解和上下级的关系——使外科手术队伍可以达到客观的一致性。

另外，团队中剩余人员职能的专业化分工是高效的关键，它使成员之间采用非常简单的交流模式成为可能，如图 3-1 所示。

Baker 的文章[3]提出了专一的、小规模的测试队伍。在那种情况下，它能按照所预期的进行运作，并具有良好的效果。

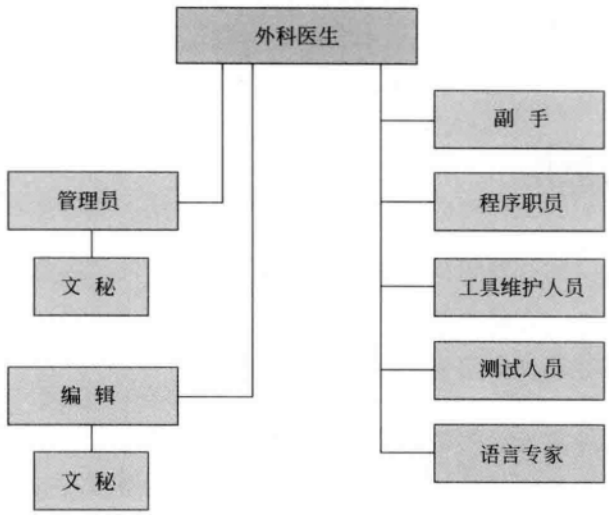


图 3-1　10 人程序开发队伍的沟通模式

团队的扩建

现在所面临的问题是如何完成 5 000 个人年的项目，而不是 20 或 30 个人年规模的系统。如果整个工作能控制在范围之内，10 人的团队无论如何组织，总是比较高效的。但是，当我们需要面对几百人参与的大型任务时，应该如何应用外科手术团队的概念呢？

扩建过程的成功依赖于这样一个事实，即每个部分的概念完整性得到了彻底的提高——决定设计的人员是原来的 1/7 或更少。所以，可以让 200 人去解决问题，而仅仅需要协调 20 个人，即那些“外科医生”的思路。

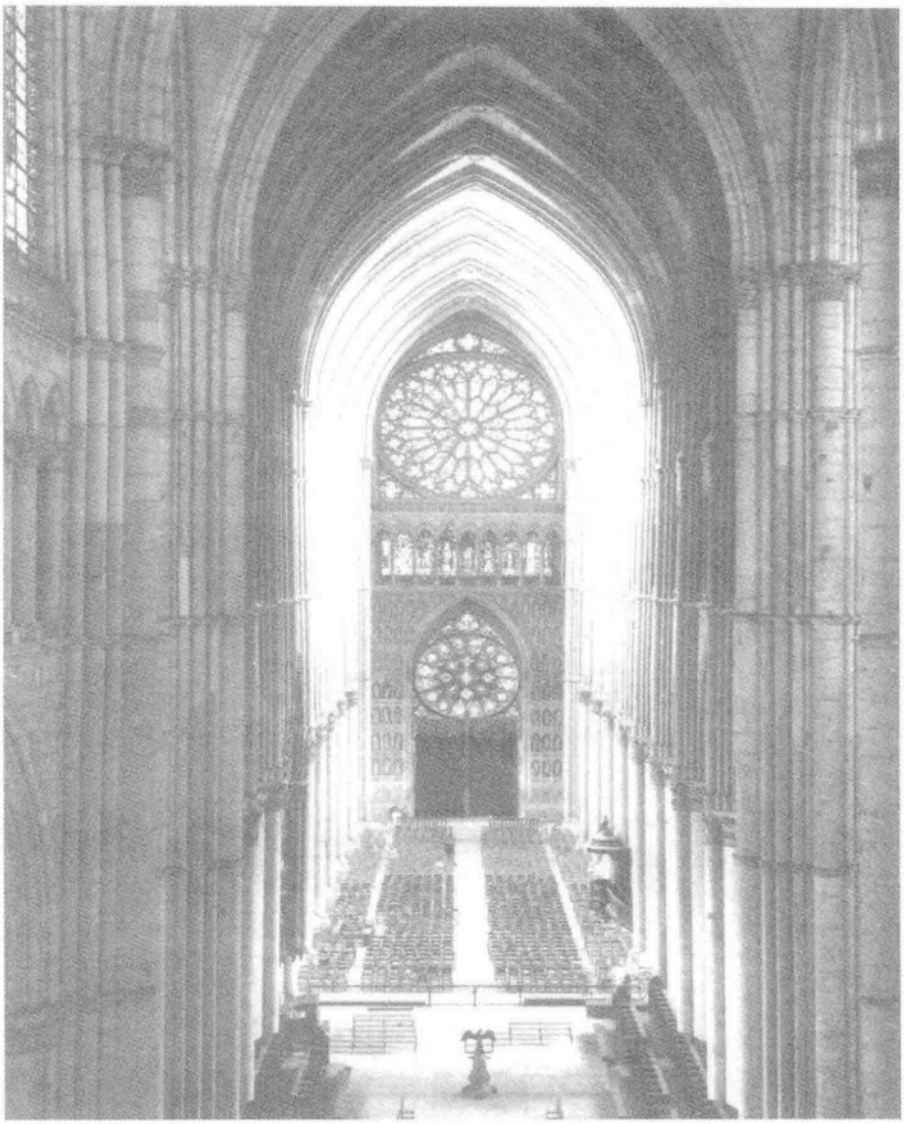
对于协调的问题，还是需要使用分解的技术，我们会在后续的章节中继续进行讨论。在这里，可以认为整个系统必须具备概念上的完整性，

要有一个系统结构师从上至下地进行所有的设计。要使工作易于管理，必须清晰地划分体系结构设计和实现之间的界线，系统结构师必须一丝不苟地专注于体系结构。总的说来，上述的角色分工和技术是可行的，在实际工作中，具有非常高的效率。

第 4 章

贵族专制、民主政治和系统设计

Aristocracy, Democracy, and System Design



E. 布多-拉莫特拍摄的兰斯大教堂内景

大教堂是艺术史上无与伦比的成就。它所宣扬的理念既不乏味也不混乱……它是一种风格上的极致，要完成这样一件艺术品，建筑大师要首尾融会贯通其前辈建筑师的成果，同时也完全掌握他们那个时代的建筑技术，并在运用这些技术时做到恰如其分，避免轻浮的炫耀，也绝不花哨。

无疑当初正是Jean d'Orbais构思出了这个建筑的整体设计，这个设计得到了其后继建筑师的认同，至少在建筑的基本要素方面是如此。这就是这个宏伟的建筑能够如此和谐统一的原因之一。

——《兰斯大教堂指南》[1]

This great church is an incomparable work of art. There is neither aridity nor confusion in the tenets it sets forth……It is the zenith of a style, the work of artists who had understood and assimilated all their predecessors' successes, in complete possession of the techniques of their times, but using them without indiscreet display nor gratuitous feats of skill.

It was Jean d'Orbais who undoubtedly conceived the general plan of the building, a plan which was respected, at least in its essential elements, by his successors. This is one of the reasons for the extreme coherence and unity of the edifice.

REIMS CATHEDRAL GUIDEBOOK[1]

概念的完整性

绝大多数欧洲的大教堂中，由不同时代、不同建筑师所建造的各个部分之间，在设计或结构风格上都存在着许多差异。后来的建筑师总是试图在原有建筑师的基础上有所“提高”，以反映他们在设计风格上的改变和个人品味上的不同。所以，在雄伟的哥特式的教堂上，依附着祥和的诺曼底风格十字架，它在显示上帝荣耀的同时，展示了同样属于建筑师的骄傲。

与之对应的是，法国城市兰斯(Reims)在建筑风格上的一致性与上面所说的大教堂形成了鲜明的对比。设计的一致性和那些独到之处一样，同样让人们赞叹和喜悦。如同旅游指南所述，风格的一致和完整性来自8代拥有自我约束和牺牲精神的建筑师们，他们每一个人牺牲了自己的一些创意，以获得纯粹的设计。同样，这不仅显示了上帝的荣耀，也体现了他拯救那些沉醉在自我骄傲中的人们的力量。

对于计算机系统而言，尽管它们通常没有花费几个世纪的时间来构建，但绝大多数系统体现出的概念差异和不一致性远远超过欧洲的大教堂。这通常并不是因为它由不同时代的设计师们开发，而是由于设计被分成了由若干人完成的若干任务。

我主张，在系统设计中，概念完整性应该是最重要的考虑因素。也就是说，为了反映一系列连贯的设计思路，宁可省略一些不规则的特性和改进，也不提倡独立和无法整合的系统，哪怕它们其实包含着许多很好的设计。在本章和以下的两章里，我们将解释在编程系统设计中，这个主题的重要性。

- 如何获得概念的完整性？

- 这样的观点是否要有一位杰出的精英，或者说是结构设计师的贵族专制，和一群创造性天赋和构思被压制的平民编程实现人员？
- 如何避免结构设计师制订出无法实现，或者是代价高昂的技术规格说明，使大家陷入困境？
- 如何才能与实现人员就技术说明的琐碎细节充分沟通，以确保设计被正确地理解，并精确地整合到产品中？

获得概念的完整性

编程系统(软件)的目的是使计算机更加容易使用。为了做到这一点，计算机装备了语言和各种工具，这些工具实际上也是被调用的程序，受到编程语言的控制。使用这些工具是有代价的：软件外部描述的规模大小是计算机系统本身说明的 10～20 倍。用户会发现寻找一个特定功能是很容易的，但相应却有太多的选择，要记住太多的选项和格式。

只有当这些功能说明节约下来的时间，比花费在学习、记忆和搜索手册上的时间要多时，易用性才会得到提高。现代编程系统节省的时间的确超过了花费的时间，但是近年来，随着越来越多复杂功能的添加，节省和花费的比率正在逐渐地减小。IBM 650 的易用性总萦绕在我的脑际，即使该系统没有使用汇编和任何其他的软件。

由于目标是易用性(simplicity)，功能与概念的复杂程度的比值才是系统设计的最终测试标准。单是功能本身或者简洁都无法成为一个好的设计评判标准。

然而这一点被广泛地误解了。操作系统 OS/360 由于其复杂且强大的功能被它的开发者广为推崇。功能，而非简洁，总是被用来衡量设计人员工作的出色程度。而另一方面，PDP-10 的时分系统由于它的简洁和概念的精干被建造它的人员所称道。当然，无论使用任何测量标准，后者的功能与 OS/360 都不在一个数量级上。但是，一旦以易用性作为衡量标准，单独的功能和简洁都是不均衡的，都只达到了真正目标的一半。

对于给定级别的功能，能用最简洁和直接的方式来指明事情的系统是最好的。只有简洁是不够的。Mooers 的 TRAC 语言和 Algol 68 用很多独特的基本概念的方法进行衡量，达到了所需的简洁特性，但它们并不**直白(straightforward)**。要表达一件待完成的事情，常常需要对基本元素进行意料不到的复杂组合。而且，仅仅了解基本要素和组合规则还不够，还需要学习惯用的用法，以及在实际工作中如何进行组合。简洁和直白来自概念的完整性。每个部分必须反映相同的原理需求的一致平衡。在语法上，每个部分应使用相同的技巧；在语义上，应具有同样的相似性。因此，易用性实际上需要设计的一致性和概念上的完整性。

贵族专制统治和民主政治

概念的完整性要求设计必须由一个人，或者非常少数互有默契的人员来实现。

而进度压力却要求很多人员来开发系统。有两种方法可以解决这种矛盾。第一种是仔细地对设计方法和具体实现进行分工。第二种是前一章节中所讨论的、一种崭新的组建编程开发团队的方法。

对于非常大型的项目，将设计方法、体系结构方面的工作与具体实现相分离是获得概念完整性的强有力方法。我亲眼目睹了它在 IBM 的 Stretch 计算机和 System/360 计算机产品线上的巨大成功。但我也看到，

这种方法在 OS/360 操作系统的开发中，由于缺乏广泛应用而遭受了失败。

系统的**体系结构**(architecture)*指的是完整和详细的用户接口说明。对于计算机，它是编程手册；对于编译器，它是语言手册；对于控制程序，它是语言和函数调用手册；对于整个系统，它是用户要完成自己全部工作所需参考的手册的集合。

因此，系统的结构师，如同建筑的结构师一样，是用户的代言人。结构师的工作，是运用专业技术知识来支持用户的真正利益，而不是维护销售人员、制作者所鼓吹的利益[2]。

必须仔细地将体系结构同实现区分开来。如同 Blaauw 所说的，“体系结构陈述的是发生了**什么**，而实现描述的是**如何**实现[3]。”他举了一个简单的例子——时钟。它的结构包括表盘、指针和上发条的旋钮。当一个小孩知道了时钟的外表结构，他很容易从手表或者教堂上的时钟辨认时间。而时钟的实现，描述了表壳中的事物——很多种动力提供装置中的一种，以及众多控制精度方案的一种。

例如，在 System/360 中，一个计算机的结构可以用 9 种不同的模型来实现；而单个实现——Model 30 的数据流、内存和微代码实现——可以用于 4 种不同的体系结构：System/360 计算机、拥有 224 个独立逻辑子通道的复杂通道、选择通道以及 1401 计算机[4]。

同样的划分方法也适用于编程系统。例如，美国的 Fortran IV 标准，是多种编译器所遵循的体系结构标准。该体系结构下有多种可能的实现：以文本为核心或以编译器为核心，快速编译或优化，以及侧重语法的实现。相类似地，任何汇编语言或任务控制语言都允许由多种编译器或调

* 译者注：本书中使用的体系结构(architecture)与现有广为业界所接受的体系结构略有不同。后者往往指系统的框架，属于设计层次，而前者更加偏重于现今的需求概念。

度程序来实现。

现在让我们来处理具有浓厚感情色彩的问题——贵族统治和民主政治。结构师难道不是新贵？他们是一些智力精英，专门来告诉可怜的实现人员如何工作？是否所有的创造性活动都被这些精英单独占有，实现人员仅仅是机器中的齿轮？难道不能遵循民主的理论，从所有的员工中搜集好的创意，以得到更好的产品，而不是将技术说明的开发工作仅限定于少数人？

最后一个问题是最简单的。我当然不认为只有结构师才有好的创意。新的概念经常来自实现人员或者用户。然而，我一直试图表达，并且我所有的经验使我确信，系统的概念完整性决定了其使用的容易程度。不能与系统基本概念进行整合的良好想法和特色，最好放到一边，不予考虑。如果出现了很多非常重要但不兼容的构想，就应该抛弃原来的设计，对不同基本概念进行合并，在合并后的系统上重新开始。

至于贵族专制统治的问题，必须回答“是”或者“否”。就只能存在少数的结构师而言，答案是肯定的，他们的工作产物的生命周期比那些实现人员的产物要长，并且结构师一直处在解决用户问题，实现用户利益的核心地位。若要得到系统概念上的完整性，必须有人控制这些概念。这实际上是一种无需任何歉意的贵族专制统治。

第二个问题的答案是否定的，因为外部技术说明的编制工作并不比具体设计实现更富有创造性，它只是一项性质不同的创造工作而已。在给定体系结构下实现其设计，同样需要同编制技术说明一样的创造性、同样新的思路和卓越的才华。实际上，产品的成本性能比在很大程度上依靠实现人员，就如同易用性在很大程度上依赖结构师一样。

有很多行业和领域中的案例让人相信纪律和规则对行业是有益的。实际上，如同某艺术家的格言所述，“没有规矩，不成方圆。”最差的建

筑往往是那些预算远远超过起始目标的项目。巴赫曾被要求每周创作一篇形式严格的歌剧，但这似乎并没有压制他的创造性。并且，我确信，如果 Stretch 计算机有更严格的限制，那么该计算机会拥有更好的体系结构。就我个人意见而言，System/360 Model 30 预算上的限制，完全获益于 Model 75 的体系结构。

类似地，我观察到外部的体系结构规定实际上是增强，而不是限制实现小组的创造性。一旦他们将注意力集中在没有人解决过的问题上，创意就开始奔涌而出。在毫无限制的实现小组中，在进行结构上的决策时，会出现大量的想法和争议，对具体实现的关注反而会比较少[5]。

我曾见过很多次这样的结果，R. W. Conway 也证实了这一点。他在 Cornell 的小组曾编制 PL/I 语言的 PL/C 编译器。他说："最终，我们的编译器决定支持不经过改进和增强的语言，因为关于语言的争议已经耗费了我们所有的时间和精力。"[6]

在等待时，实现人员应该做什么

几百万元的失误是非常令人惭愧的经验，但也是让人记忆深刻的教训。当年我们计划如何组织编写 OS/360 外部技术说明的那个夜晚，常常栩栩如生地重现在我的脑海。我和体系结构经理、程序实现经理一起制定进度计划，并确认责任分工。

体系结构经理拥有 10 位很好的员工，他声称他们可以书写规格说明，并能够出色地完成任务。该任务需要 10 个月，比所允许的进度多了 3 个月。

程序实现经理拥有 150 名员工。他认为在体系结构队伍的协助下，他们可以准备技术说明，并且能按照时间进度，完成高质量的、切合实

际的说明。此外，如果仅由体系结构的团队承担该工作，他的150人只能坐在那里干等10个月，无所事事。

对此，体系结构经理的反应是，如果让程序实现队伍来负责该工作，工作将不会按时完成，仍将推迟3个月，而且质量更加低劣。遗憾的是，我还是将工作分派给了程序实现队伍，其结果也确实如此。体系结构经理的两个结论都得到了证实。另外，概念完整性的缺乏导致系统开发和修改上要付出更昂贵的代价，我估计至少增加了一年的调试时间。

当然，很多因素造成了那个错误的决策，但决定性因素是时间进度和让150名编程人员参与工作的愿望。而它也正是我想强调的致命危险。

当建议由小型体系结构团队来编写计算机或编程系统的所有外部技术说明时，编程人员提出了三个反对意见：

- 该说明中的功能过于繁多，而对实际情况中的成本考虑比较少；
- 结构师获得了所有创造发明的快乐，剥夺了实现人员的创造力；
- 当体系结构的队伍缓慢工作时，很多实现人员只能空闲地坐着等待。

第一个问题确实是危险的，我们将在下一章中讨论这个问题，但其他的两个问题都是一些简单而纯粹的误解。正如我们前面所看到的，实现同样是一项高级的创造性活动。具体实现中创造和发明的机会，并不会因为指定了外部技术说明而大为减少，相反创造性活动会因为规范化而得到增强，整个产品也一样。

最后一个反对意见是时间顺序和阶段性上的问题。问题的简要回答是，在说明完成的时候，才雇用编程实现人员。这也正是在搭建一座建筑时所采用的方法。

在计算机这个行业中，节奏非常快，而且人们常常想尽可能地压缩时间进度，那么技术说明和开发实现能有多少重叠呢？

Blaauw 指出，整个创造性活动包括了三个独立的阶段：体系结构(architecture)、设计实现(implementation)和物理实现(realization)。在实际情况中，它们往往可以同时开始和并发地进行。

例如，在计算机的设计中，一旦设计实现人员有了对手册的模糊设想，对技术有了相对清晰的构思以及拥有了适当的成本和目标时，工作就可以开始了。他可以开始设计数据流、控制序列、进行总体概念包装等。同时，还需要设计所需的工具以及进行相应的调整，特别是记录存档系统和设计自动化系统。

同时，在物理实现的阶段，对于电路、板卡、线缆、机箱、电源和内存，我们必须分别设计、细化和编制文档。这项工作与体系结构及设计实现并行进行。

在编程系统的设计中，这个原理同样适用。在外部说明完成之前，设计实现人员有很多事情可以做。只要有一些最终将并入外部说明的系统功能雏形，他就可以开始了。首先，必须设定良好定义的时间和空间目标，了解产品运行的平台配置。接着，他可以开始设计模块的边界、表结构、路径和阶段分解、算法以及所有的工具。另外，其还需要花费一些时间与体系结构师进行沟通。

同时，在物理实现的阶段，也有很多可以着手的工作。编程也是一项技术，如果是新型的机器，则在库的调整、系统管理以及搜索和排序算法上，有许多事情需要处理[7]。

概念的完整性的确要求系统只反映唯一的设计理念，用户所见的技术说明来自少数人的思想。实际工作被划分成体系结构、设计实现和物理实现，但这并不意味着该开发模式下的系统需要更长的时间来创建。

经验显示恰恰相反，整个系统将会开发得更快，所需要的测试时间将更少。同工作广泛的水平分割相比，垂直划分从根本上大大减少了劳动量，使交流彻底地被简化，概念完整性得到了大幅提高。

CHAPTER 5

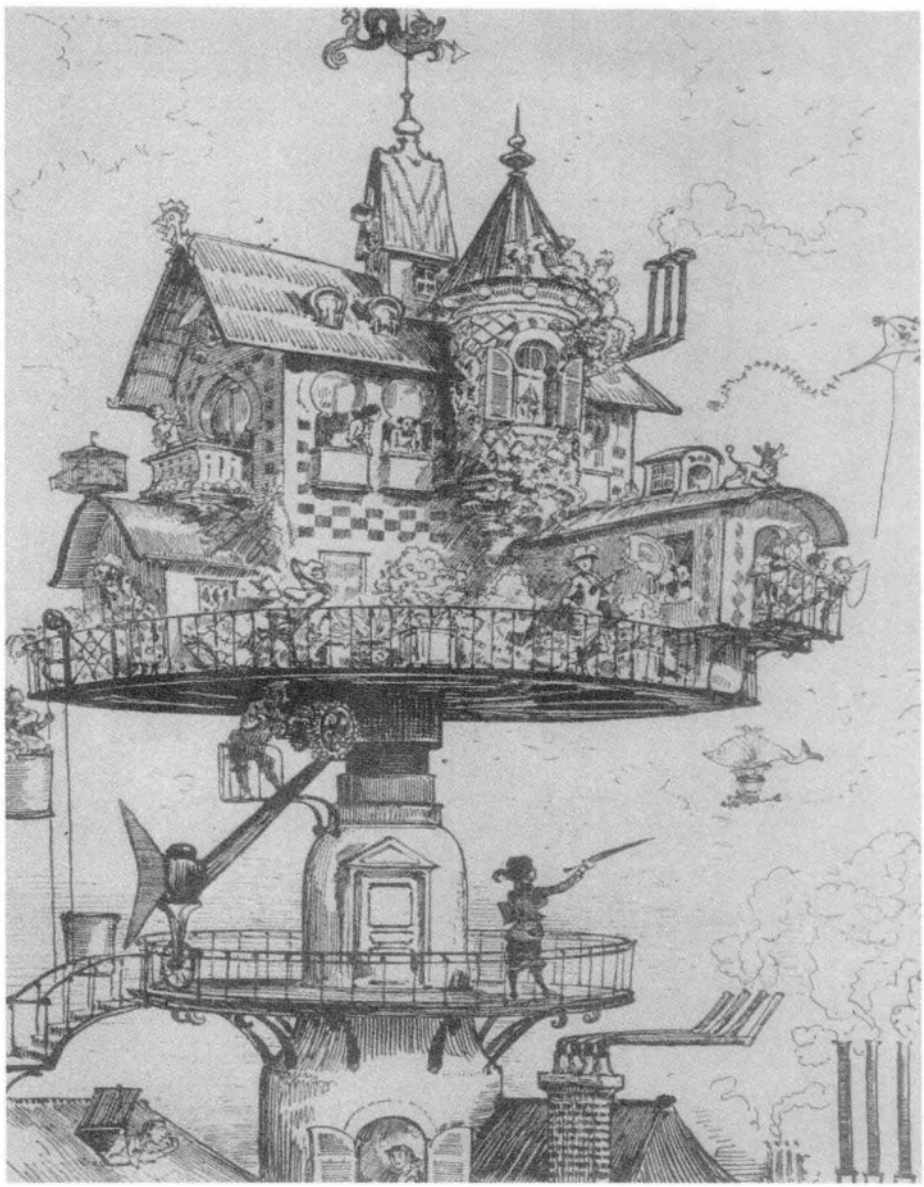
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第5章

画蛇添足

The Second-System Effect



A. 罗比达 **1882** 年在《二十世纪》(***Le Vingtième Siècle***)一书中画的插图平版画《适合空中交通的旋转屋》

聚沙成塔，集腋成裘。

——奥维德

Adde parvum parvo magnus acervus erit.

［Add little to lillte and there will be a big pile.］

OVID

如果将制定功能规格说明的责任从开发快速、成本低廉的产品的责任中分离出来，那么有什么样的准则和机制来约束结构师的创造性热情呢？

基本回答是结构师和建筑人员之间彻底、谨慎、和谐的交流。尽管还有很多值得关注的、更细致的答案。

结构师的交互准则和机制

建筑行业的结构设计师使用估算技术来编制预算，该估算技术会由后续的承包商报价来验证和修正。承包商的报价总会超过预算。接下来，设计师会重新改进他的估算技术或修订设计，调整到下一期工程中。他也可能会向承包商建议，使用更加便宜的方法来实现设计。

类似过程也支配着计算机系统和计算机编程系统的结构师。相比之下，他具有能在设计早期从承包商处得到报价的优势，几乎是只要他询问，就能得到答案。他的不利之处常常是只有一个承包商，后者可以增高或降低前者的估计，来反映对设计的好恶。实际情况中，尽早交流和持续沟通能使结构师有较好的成本意识，以及使开发人员获得对设计的信心，并且不会混淆各自的责任分工。

面对估算过高的难题，结构师有两个选择：削减设计或者采用成本更低的实现方法。后者是固有的主观感性反应。此时，结构师是在向开发人员的做事方式提出挑战。想要成功，结构师必须：

- 牢记是开发人员承担创造性和发明性的实现责任，所以结构师只能建议，而不能支配；

- 时刻准备着为所指定的说明建议一种实现的方法，同样准备接受其他任何能达到目标的方法；
- 对上述的建议保持低调和不公开；
- 准备放弃坚持所作的改进建议。

一般开发人员会反对体系结构上的修改建议。通常他是对的——当实现产品时，某些次要特性的修改会造成意料不到的成本开销。

自律——开发第二个系统所带来的后果

在开发第一个系统时，结构师倾向于精炼和简洁。他知道自己对正在进行的任务不够了解，所以会谨慎、仔细地工作。

在设计第一个项目时，他会面对不断产生的装饰和润色功能。这些功能都被搁置在一边，作为“下一个”项目的内容。第一个项目迟早会结束，而此时的结构师，对这类系统充满了十足的信心，其精通这一级别系统，并且时刻准备开发第二个系统。

第二个系统是设计师们所设计的最危险的系统。而当他着手第三个或第四个系统时，先前的经验会相互验证，得到对此类系统通用特性的判断，而且系统之间的差异会帮助他识别出经验中不够通用的部分。

一种普遍倾向是过分地设计第二个系统，向系统添加很多修饰功能和想法，它们曾在第一个系统中被小心谨慎地放在次要位置。结果如同Ovid所述，是一个“大馅饼”。例如，后来被嵌入到7090的IBM 709系统的体系结构，709是对非常成功和简洁的704系统进行升级的二次开发项目。设计师将709的操作集合设计得如此丰富和充沛，以至于只有一半操作被客户经常使用。

让我们来看一个更严重的例子——Stretch 计算机的结构(architecture)、设计实现(implementation)、物理实现(realization)，它是很多人被压抑创造力的宣泄出口，也是他们中大部分人的第二个系统。正如 Strachey 在评审时所述：

我对 Stretch 系统的印象是，从某种角度而言，它是一个产品线的终结。如同早期的计算机程序一样，它非常富有创造性，设计非常复杂，却非常高效。但不知为什么，我同时感觉到，它粗糙、浪费、缺乏优雅，并让人觉得必定存在某种更好的方法可以代替它。[1]

OS/360 对于大多数设计者来说是第二个系统。它的设计小组成员来自 1410-7010 磁盘操作系统、Stretch 操作系统、Mercury 实时系统项目和 7090 的 IBSYS。几乎没有人有**两次**以上的操作系统经验。[2]因此，OS/360 是典型的开发第二次所引起的后果(second-system effect)，是软件行业的 Stretch 系统。Strachey 的赞誉和批评可以毫无更改地应用在其中。

例如，OS/360 开发了 26 字节的常驻日期翻转例程来正确地处理闰年的 12 月 31 日的问题(闰年的第 366 天)，其实它完全可以留给操作员来完成。

开发第二个系统所引起的后果的另一个表现与纯粹的功能修饰和增强明显不同，也就是说存在对某些技术进行细化、精炼的趋势。由于基本系统设想发生了变化，这些技术已经显得落后。OS/360 中有很多这样的例子。

例如，链接编辑器的设计，它用来对分别编译后的程序进行装载，解决它们之间的交叉引用。除了这些基本功能，它还支持程序的覆盖(overlay)。这是所有对覆盖服务程序的实现中最好的一种。它允许链接

时在外部完成覆盖结构，而无需在源代码中进行设计。它还允许在运行时改变覆盖，而不必重新编译。它配备了丰富的实用选项和各种功能。某种意义上，它是若干年静态覆盖技术开发的顶峰。

然而，它也是最后和最优秀的恐龙，因为它属于一个基本运行方式为多道程序，以动态内核分配为基础的系统，这直接与静态覆盖的概念相冲突。如果我们把投入在覆盖管理上的工作量，用在提高动态内核分配和动态交叉引用的性能上，那么系统将会运行得很好。

另外，链接编辑器需要很大的空间，而且它本身就包含了很多链接库，以至于即使在不使用覆盖管理功能，仅仅使用链接功能的时候，它也比绝大多数系统的编译程序慢。具有讽刺意味的是，链接程序的目的是为了避免重新编译。这种情况就像一个挺着大肚子的节食者一样，直到系统的思想已经十分优越时，才开始对原有技术进行细化和精炼。

TESTRAN 调试程序是这个趋势的另一个例子。它在批调试程序中是出类拔萃的，配备了真正优雅的快照和内存信息转储功能。它使用了控制段的概念和卓越的生成技术，从而不需要重新编译或解释，就能实现选择性跟踪和快照。这种 709 共享操作系统[3]中富于想象的概念得到了广泛的使用。

但同时，整个无需重编译的批调试概念变得落伍了。使用语言解释器或增量编译器的交互式计算系统向它提出了最根本的挑战。即使是在批处理系统中，快速编译/慢速执行编译器的出现也使源代码级别调试和快照技术成为优先选择的技术。如果在构建和优化交互式和快速编译程序之前，就已经着手 TESTRAN 的开发，那么系统将非常优秀。

还有另外一个例子是调度程序。OS/360 的调度程序是非常杰出的，它提供了管理固定批作业的杰出功能。从真正意义上讲，该调度程序是作为 1410-7010 磁盘操作系统后续的二次系统，经过了精炼、改进和增

强。它是除了“输入—输出”以外的非多道程序批处理系统，是一种主要用于商业应用的系统。但是，它对 OS/360 的远程任务项、多道程序和永久驻留交互式子系统，几乎完全没有影响和帮助。实际上，OS/360 调度程序的设计使它们变得更加困难。

结构师如何避免开发第二个系统所引起的后果，从而避免画蛇添足？是的，虽然他无法跳过二次系统，但他可以有意识地关注这个系统的特殊危险，运用特别的自我约束准则，来避免那些功能上的过于修饰；根据系统基本理念及目的变更，舍弃一些功能。

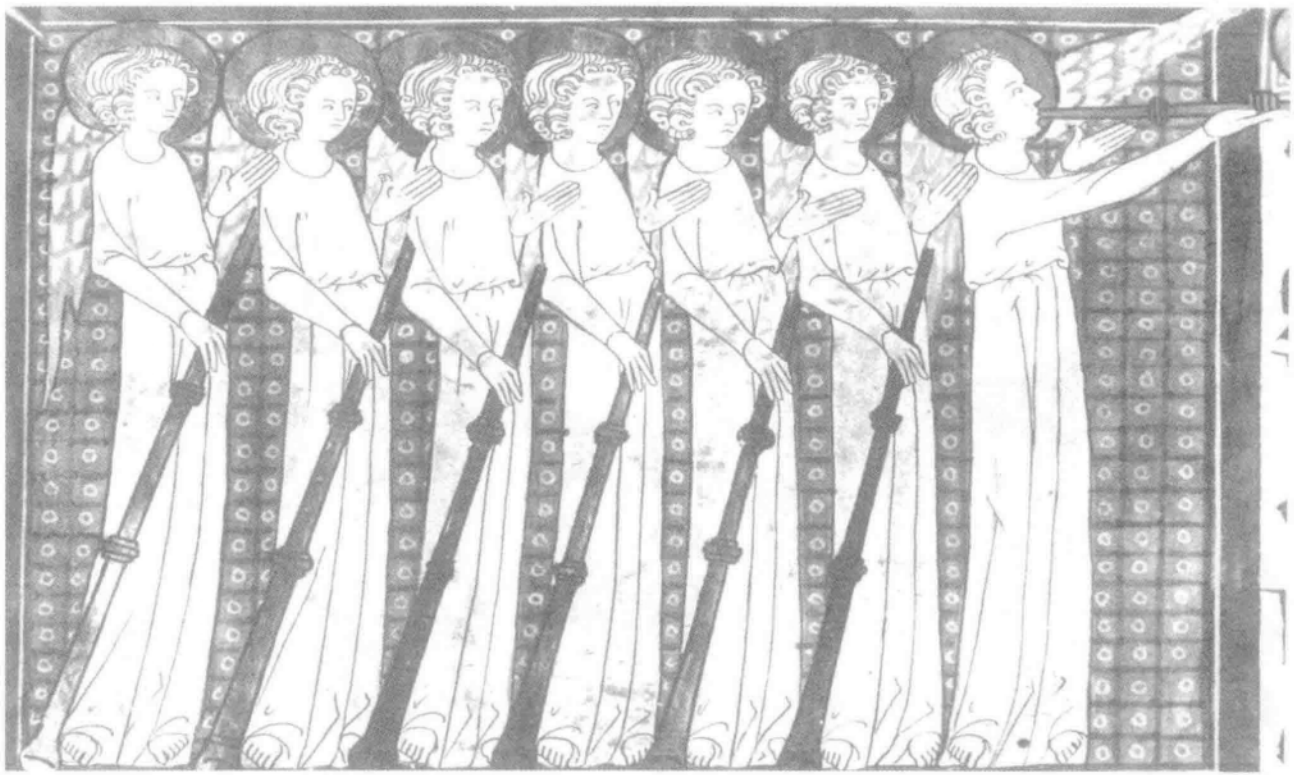
一个可以开阔结构师眼界的准则是为每个小功能分配一个值：每次改进，功能 x 不超过 m 字节的内存和 n 微秒。这些值会在一开始作为决策的向导，在物理实现期间充当指南和对所有人的警示。

项目经理如何避免开发第二个系统所引起的后果，从而避免画蛇添足？他必须坚持至少拥有两个系统以上开发经验结构师的决定。同时，保持对特殊诱惑的警觉，他可以不断提出正确的问题，确保原则上的概念和目标在详细设计中得到完整的体现。

第6章

贯彻执行

Passing the Word



《七个吹号角天使》（*The Seven Trumpets*）

注：出自 14 世纪出版的 *The Wells Apocalypse*。在《新约 • 启示录》中记载，每个天使一吹号，天地间就会发生一些变故。

资料来源：The Bettman Archive

他只是坐在那里，嘴里说：“做这个！做那个！”当然，什么都不会发生，光说不做是没有用的。

——哈里·杜鲁门，“论《总统的权力》”[1]

He'll sit here and he'll say, "Do this! Do that!" And nothing will happen.

HARRY S. TRUMAN, ON *PRESIDENIAL POWER*[1]

假设一个项目经理已经拥有行事规范、富有经验的结构师和许多编程实现人员，那么，他如何确保每个人听到、理解并实现结构师的决策？对于一个由1 000人开发的系统，一个10个结构师的小组如何保持系统概念上的完整性？在System/360硬件设计工作中，我们摸索出来一套实现上述目标的方法，它们对于软件项目同样适用。

文档化的规格说明——手册

手册或者书面规格说明，是一个非常必要的工具，仅有文档是不够的。手册是产品的**外部**规格说明，它描述和规定了用户所见的每一个细节；同样地，它也是结构师主要的工作产物。

随着用户和实现人员反馈的增加，规格说明中难以使用和难以构建实现的地方不断被指出，规格说明也不断地被重复准备和修改。然而对实现人员而言，修改的阶段化是很重要的——在进度表上应该有带日期的版本信息。

手册不仅要描述包括所有界面在内的用户可见的一切，还要避免描述用户看不见的事物。后者是编程实现人员的工作范畴，而其设计和创造是不应该被限制的。体系结构设计人员必须为自己描述的任何特性准备一**种**实现方法，但是他不应该试图支配具体的实现过程。

规格说明的风格必须清晰、完整和准确。用户常常会单独提到某个定义，所以每条说明必须重复所有的基本要素，所有文字都要相互一致。这往往使手册读起来枯燥乏味，但是精确比生动更加重要。

System/360 Principles of Operation 的一致完整性仅来自两名作者：Gerry Blaauw 和 Andris Padegs。思路是大约 10 个人的想法，但如果想保持文字和产品之间的一致性，则必须由一个或两个人来完成将其结论转换成书面规格说明的工作。而且，将定义书写成文字，必须对很多最初并不是非常重要的问题进行判断，并得出结论。例如，System/360 需要决定在每次操作后，如何设置返回的条件码。其实，对于在整个设计中，保证这些看似琐碎的问题处理原则上的一致性，**绝对不是一件无关紧要的事情**。

我想我所见过的最好的一份手册是 Blaauw 的 System/360 Principles of Operation 的附录。它精确仔细地规定了对 System/360 兼容性的**限制**。它定义了兼容性，描述了将要达到的目标，列举了外观部分，这些部分可能是结构师有意省略的，或者源于某个模型与其他模型的差异，某个给定模型的拷贝和其他拷贝的差异，甚至是工程上的变更引起的拷贝自身上的差异。而这说明了作者所应该追求的精确程度，他必须在仔细定义规定什么的同时，定义未规定什么。

形式化定义

英语或者其他任何一种人类语言，从根本上说，都不是一种能精确表达上述定义的方式。因此，手册的作者必须注意到自己的思路和语言，达到所需要的精确程度。一种颇具吸引力的作法是对上述定义使用形式化标记方法。毕竟，精确度是我们需要的东西，这也正是形式化标记方法存在的理由和原因。

让我们来看一看形式化定义的优点和缺点。如文中所述，形式化定义是精确的，倾向完整。差异越明显，填补得越快。但是形式化定义的缺点是不易理解的。记叙性文字可以表达结构性的原则，描述阶段上或

层次上的结构，并提供实例。它可以很容易地表达异常和强调对比的关系，最重要的是，它可以解释原因。在表达的精确度上，目前所提出的形式化定义，可以收到令人惊异的效果，并增强了我们在这方面的信心。但是，它还需要记叙性文字的辅助，才能使内容易于领会和讲授。出于这些原因，我想将来的规格说明应同时包括形式化和记叙性定义两种方式。

一句古老的格言警告说："不要携带两个时钟出海，带一个或三个。"同样的原则也适用于形式化和记叙性定义。如果同时具有两种方式，则必须以一种作为标准，另一种作为辅助描述，并照此明确地进行划分。它们都可以作为表达的标准，例如，Algol 68 采用形式化定义作为标准，记叙性文字作为辅助。PL/I 使用记叙性定义作为标准，形式化定义用做辅助表述。System/360 也将记叙性文字用做标准，以形式化定义用做派生的论述。

很多工具可以用于形式化定义，例如巴科斯范式(Backus-Naur Form)在语言定义中很常用，其在书本中有详细的描述。[2] PL/I 的形式化定义使用了抽象语法的新概念，该概念有很确切的解释。[3] Iverson 的 APL 曾用来描述机器，突出的应用是 IBM 7090[4]和 System/360[5]。

Bell 和 Newell 曾建议采用能同时描述配置和机器结构的新标注方法，并且在许多机型的应用上得以体现，如 DEC PDP-8[6]、7090[6]和 System/360[7]。

在规定系统外部功能的同时，几乎所有的形式化定义均会用来描述和体现硬件系统或软件系统的某个设计实现。语法和规则的表达可以不需要具体的设计实现，但是特定的语义和意义通常会通过一段实现该功能的程序来定义。理所当然，这是一种实现，不过它过多地限定了体系结构。所以必须特别指出，形式化定义仅仅用于外部功能，说明它们是什么。

如前面所述，形式化定义是一种设计实现。反之，设计实现也可以作为一种形式化定义的方法。当制造第一批兼容性的计算机时，我们使用的正是上述技术：新的机器同原有的机器一致。如果手册中有一些表述模糊的地方，“问一问机器！”——设计一段程序来确定其行为，新机器必须按照上述结果运行。

硬件或软件系统的仿真装置，可以按照相同的方式完整运用。它是一种实现，可以运行。因此，所有定义的问题可以通过测试来解决。

使用实现来作为一种定义的方式有一些优点。首先，所有问题可以通过试验清晰地得到答案，从来不需要争辩和商讨，回答是快捷迅速的。通过定义得出的答案，总是同所要求的一样精确和正确。但是，相对于这些优点的，是一系列难以克服的缺点。实现可能更加过度地规定了外部功能。无效的语法通常会产生某些结果。在拥有错误控制的系统中，它通常**仅仅**导致某种“无效”的指示，**而不会产生其他的东西**。在无错误控制的系统中，会产生各种副作用，它们可能被程序员所使用。例如，当我们着手在 System/360 上模拟 IBM 1401 时，有 30 个不同的“古董”——被认为是无效操作的副作用——得到广泛的应用，并被认为是定义的一部分。作为一种定义，实现体现了过多的内容：它不但描述了系统必须做什么，还声明了自己到底做了些什么。

因此，当提及尖锐的问题时，实现有时会给出未在计划中的意外答案；在这些答案中，**真正的**定义常常是粗糙的，因为它们从来没有被仔细考虑过。这些粗糙的功能在其他的设计实现中，往往是效率低下或者代价高昂的。例如，一些机器在乘法运算之后，将某些运算的垃圾遗留在被乘数寄存器中。该功能确切的特性，即保存运算垃圾，成为**真正**定义的一部分。然而，重复该细节可能会阻止某些快速乘法算法的使用。

最后，关于实际使用标准是形式化描述还是叙述性文字这一点而言，使用实现作为形式化定义特别容易引起混淆，特别是在程序化的仿真中。

另外，当实现充当标准时，还必须防止对实现的任何修改。

直接整合

对软件系统的体系结构师而言，存在一种更加可爱的方法来传播和推行定义。对于建立模块间接口语法，而非语义时，它特别有用。这项技术是设计被传递参数或共享存储器的声明，并要求编程实现在编译时的一些操作(PL/I 的宏或%INCLUDE)来包含这些声明。另外，如果整个接口仅仅通过符号名称进行引用，那么需要修改声明的时候，可以增加或插入新变量，或者重新编译而不需修改使用的程序。

会议和大会

无需多说，会议是必要的。然而，数百人在场的大型磋商会议往往需要大规模和非常正式地召集。因此，我们把会议分成两个级别：周例会和年度大会——这实际上是一种非常有效的方式。

周例会是每周半天的会议，由所有的结构师，硬件和软件实现人员代表以及市场计划人员参与，由首席系统结构师主持。

会议中，任何人可以提出问题和修改意见，但是建议书通常是以书面形式在会议之前分发。新问题通常会需要一些讨论时间。重点是创新，而不仅仅是结论。该小组试图发现解决问题的各种方法，然后少数解决方案会被传递给一个或多个结构师，详细地记录到书面的变更建议说明书中。

接着会对详细的变更建议做出决策。这会经历几个反复过程，实现人员和用户会仔细地进行考虑，正面和负面的意见都会被很好地描述。

如果达成了共识，非常好；如果没有，则由首席结构师来决定。这需要花费时间，最终所发布的结论是正式和果断的。

周例会的决策会给出迅捷的结论，使工作继续开展下去。如果任何人对结果过于不满，可以立刻诉诸项目经理，但是这种情况非常少见。

这种会议的卓有成效是由于以下几种原因。

(1) 数月内，相同小组——结构师、用户和实现人员——每周交流一次。因此，大家对项目相关的内容比较了解，不需要安排额外时间对人员进行培训。

(2) 上述小组十分睿智和敏锐，深刻理解所面对的问题，并且其与产品密切相关。没有人是“顾问”的角色，每个人都要承担义务。

(3) 当问题出现时，在界线的内部和外部同时寻求解决方案。

(4) 正式的书面建议集中注意力，强制了决策的制定，避免会议草稿纪要方式的不一致。

(5) 明确地授予首席结构师决策的权力，避免了妥协和拖延。

随着时间的推移，一些决定没有很好地贯彻，一些小事情并没有被某个参与者真正地接受，其他决定造成了未曾遇到的问题。对于这些问题，有时周例会并不赞同重新考虑。慢慢地，很多小要求、公开问题或者不愉快会堆积起来。为解决这些堆积起来的问题，我们会举行年度大会，典型的年度大会会持续两周。(如果由我重新安排，我会每六个月举行一次。)

这些会议在手册冻结的前夕召开。出席人员不仅包括体系结构小组和编程人员、实现人员的结构代表，还包括编程经理、市场和实现人员，由 System/360 的项目经理主持。典型的议程包括大约 200 个条目，大多

数条目的规模很小，它们列举在会议室周围的图表上，每个不同的声音都有机会得到表达，然后，会制定出决策。通过出色的计算机化文本编辑工作(许多优秀员工的卓越的工作成果)，每天早晨，会议参与人员会在座位上发现已更新的手册说明，其记录了前一天的各项决定。

这些“收获的节日”不仅可以解决决策上的问题，而且使决策更容易被接受。每个人都在倾听，每个人都在参与，每个人对复杂约束和决策之间的相互关系有了更透彻的理解。

多重实现

System/360 的结构师具有两个空前有利的条件：充足的工作时间，拥有与实现人员相同的策略影响力。充足时间来自新技术的开发日程；而多重实现的同时开发带来了策略上的平等性。不同实现之间严格要求相互兼容，这种必要性是强制规格说明的最佳代言人。

在大多数计算机项目中，机器和手册之间往往会在某一天出现不一致，人们通常会忽略手册。因为与机器相比，手册更容易改动，并且成本更低。然而，当存在多重实现时，情况就不是这样了。这时，如实地遵从手册更新机器所造成的延迟和成本的消耗，比根据机器调整手册要低。

在定义某编程语言的时候，上述概念可以卓有成效地得到应用。可以肯定的是，迟早会有很多编译器或解释器被推出，以满足各种各样的目标。如果起初至少有两种以上的实现，那么定义会更加整洁和规范。

电话日志

随着实现的推进，无论规格说明已经多么精确，还是会出现无数结

构理解和解释方面的问题。显然，很多此类问题需要文字澄清和解释，还有一些仅仅是因为理解不当。

显然，对于存有疑问的实现人员，应鼓励他们打电话询问相应的结构师，而不是一边自行猜测一边工作，这是一项很基本的措施。他们还需要认识到的是，上述问题的答案必须是可以告知每个人的权威性结论。

一种有用的机制是由结构师保存**电话日志**。日志中，他记录了每一个问题和相应的回答。每周，对若干结构师的日志进行合并，重新整理，并分发给用户和实现人员。这种机制不是很正式，但非常快捷和易于理解。

产品测试

项目经理最好的朋友就是他每天要面对的对手——独立的产品测试机构/小组。该小组根据规格说明检查机器和程序，充当麻烦的代言人，查明每一个可能的缺陷和相互矛盾的地方。每个开发机构都需要这样一个独立的技术监督部门，来保证其公正性。

在最后的分析中，用户是独立的监督人员。在残酷的现实使用环境中，每个细微缺陷都将无从遁形。产品测试小组则是顾客的代理人，专门寻找缺陷。不时地，细心的产品测试人员总会发现一些没有贯彻执行、设计决策没有正确理解或准确实现的地方。出于这方面的原因，设立测试小组是使设计决策得以贯彻执行的必要手段，同样也是需要尽早着手，与设计同时实施的重要环节。

CHAPTER 7

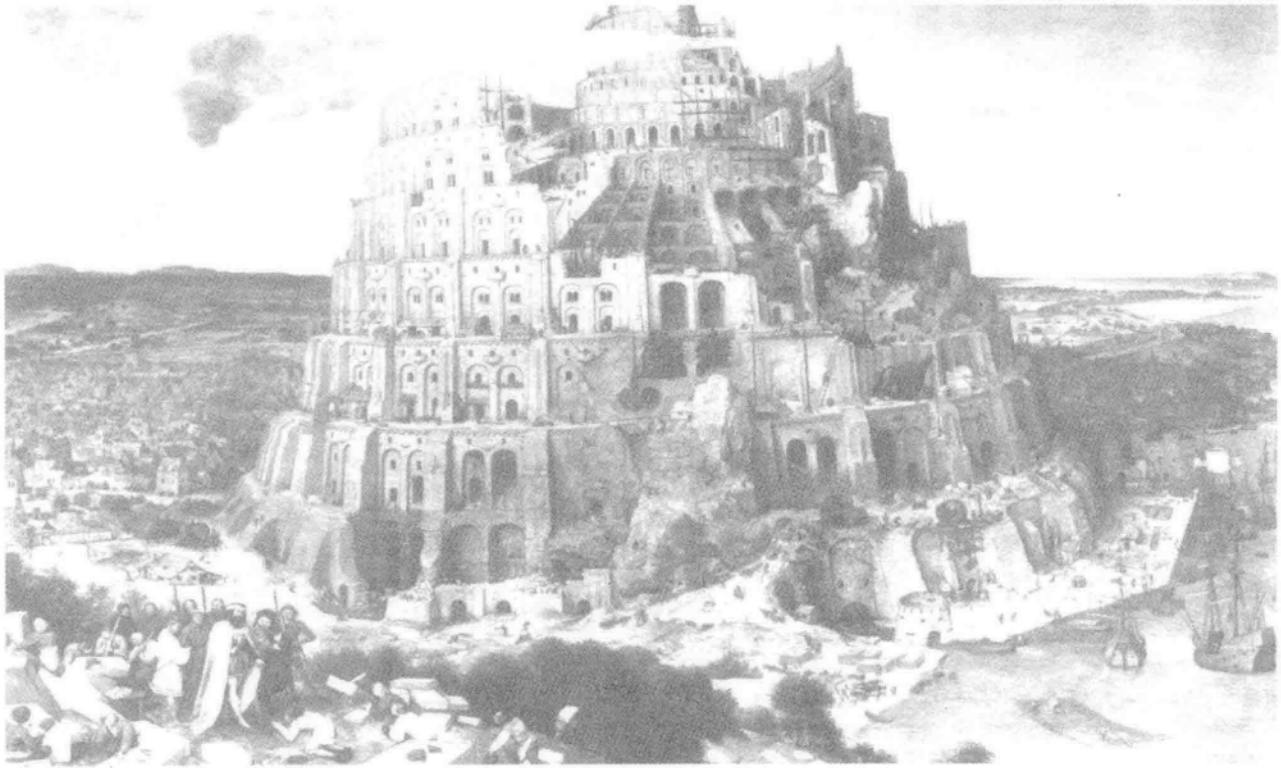
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 7 章

为什么巴比伦塔会失败

Why Did the Tower of Babel Fail?



P. 勃鲁盖尔 1563 年的油画作品《巴比伦塔》(*Turmbau zu Babel*)
(资料来源：Kunsthistorisches Museum, Vienna)

现在整个大地都采用一种语言，只包括为数不多的单词。在一次从东方往西方迁徙的过程中，人们发现了苏美尔地区的一处平原，并在那里定居下来。接着他们奔走相告说："来，让我们制造砖块，并把它们烧好。"于是，他们用砖块代替石头，用沥青代替灰泥(建造房屋)。然后，他们又说："来，让我们建造一座带有高塔的城市，这个塔将高达云霄，也将让我们声名远扬；同时，有了这个城市，我们就可以聚居在这里，再也不会分散在广阔的大地上了。"于是上帝决定下来看看人们建造的城市和高塔。看了以后，他说："他们只是一个种族，使用一种语言，如果他们一开始就能建造城市和高塔，那么以后就没有什么难得倒他们了。来，让我们下去，在他们的语言里制造一些混淆，让他们相互之间不能听懂。"这样，上帝把人们分散到世界各地，于是他们不得不停止建造那座城市。

——《创世纪》，11：1—8

Now the whole earth used only one language, with few words. On the occasion of a migration from the east, men discovered a plain in the land of Shinar, and settled there. Then they said to one another, "Come, let us make bricks, burning them well." So they used bricks for stone, and bitumen for mortar. Then they said, "Come, let us build ourselves a city with a tower whose top shall reach the heavens (thus making a name for ourselves), so that we may not be scattered all over the earth." Then the Lord came down to look at the city and tower which human beings had built. The Lord said, "They are just one people, and they all have the same language. If this is what they can do as a beginning, then nothing that they resolve to do will be impossible for them. Come, let us go down, and there make such a babble of their language that they will not understand one another's speech." Thus the Lord dispersed them from there all over the earth, so that they had to stop building the city.

GENESIS 11: 1-8

巴比伦塔的管理教训

据《创世纪》记载，巴比伦塔是人类继诺亚方舟之后的第二大工程壮举，同时，其也是第一个彻底失败的工程。

这个故事在很多方面和不同层次都是非常深刻和富有教育意义的。让我们将它仅仅作为纯粹的工程项目，来看看在管理上有什么值得学习的教训。这个项目到底有多好的先决条件？

(1) 他们是否有：**清晰的目标**？是的，尽管幼稚得近乎不可能。而且，项目早在遇到这个基本的限制之前，就已经失败了。

(2) 他们是否有：**人力**？非常充足。

(3) 他们是否有：**材料**？在美索不达米亚有着丰富的泥土和柏油沥青。

(4) 他们是否有：**足够的时间**？是的，没有任何时间限制的迹象。

(5) 他们是否有：**足够的技术**？是的，金字塔或锥形的结构本身就是稳定的，可以很好地分散压力负载。对砖石建筑技术，人们有过深入的研究。同样，项目远在达到技术限制之前，就已经失败了。

那么，既然他们具备了所有的这些条件，为什么项目还会失败呢？他们还缺乏什么？其缺乏两个方面，其一是**交流**；其二是交流的结果——**组织**。他们无法相互交谈，从而无法合作。当合作无法进行时，工作陷入了停顿。通过史书，我们推测出来，交流的缺乏导致了争辩、沮丧和群体猜忌。很快，部落开始分裂——大家选择了孤立，而不是互相争吵。

大型编程项目中的交流

现实也是如此。因为左手不知道右手在做什么，所以进度缓慢、功能不合理和系统缺陷等问题纷纷出现。随着工作的进行，许多小组慢慢地修改自己程序的功能、规模和速度，他们明确或者隐含地更改了一些有效输入和输出结果用法上的约定。

例如，程序覆盖(program-overlay)功能的实现者可能遇到了问题。根据统计报告显示，应用程序很少使用该功能。于是，实现者降低了覆盖功能的速度。与此同时，整个开发队伍中，其他同事可能正在设计监控程序，而监控程序在很大程度上依赖于覆盖功能的速度，它在速度上的变化成为主要的规格说明变更。因此需要从系统角度来考虑和衡量该变化，以及公开、广泛地发布变更结果。

那么，团队如何进行相互之间的交流沟通呢？

- **非正式途径**。清晰定义小组内部的相互关系和充分利用电话，鼓励大量的电话沟通，从而达到对所书写文档的共同理解。
- **会议**。常规项目会议。会议中，团队一个接一个地进行简要的技术陈述。这种方式非常有用，能澄清成百上千的细小误解。
- **工作手册**。在项目的开始阶段，应该准备正式的项目工作手册。理所应当，我们专门用一节来讨论它。

项目工作手册

是什么。项目工作手册不是一篇独立的文档，它是对项目必须产出的一系列文档进行组织的一种结构。

项目**所有**的文档都必须是该结构的一部分。这包括目的、外部规格说明、接口说明、技术标准、内部说明和管理备忘录。

为什么。技术说明几乎是必不可少的。如果某人就硬件或软件的某一部分，去查看一系列相关的用户手册。他发现的不仅仅是思路，而且还有能追溯到最早备忘录的许多文字和章节，这些备忘录对产品提出建议或者解释设计。对于技术作者而言，文章的剪切粘贴与钢笔一样有用。

基于上述理由，再加上“未来产品”的高质量手册，将诞生于“今天产品”的备忘录，所以正确的文档结构非常重要。事先将项目工作手册设计好，能保证文档的结构本身是规范的，而不是杂乱无章的。另外，有了文档结构，后来书写的文字就可以放置在合适的章节中。

使用项目手册的第二个原因是控制信息发布。控制信息发布并不是为了限制信息，而是确保信息能到达所有需要它的人的手中。

项目手册的第一步是对所有的备忘录编号，从而每个工作人员可以通过标题列表来检索是否有他所需要的信息。还有一种更好的组织方法，就是使用树状的索引结构。而且如果需要的话，可以使用树结构中的子树来维护发布列表。

处理机制。同许多其他的软件管理问题一样，随着项目规模的扩大，技术备忘录的问题以非线性趋势增长。10 人的项目，文档仅仅通过简单的编号就可以了。100 人的项目，若干个线性索引常常可以满足要求。1 000 人的项目，人员无可避免地散布在多个地点，对结构化工作手册的**需要**和手册**规模**上的要求都增加了许多。那么，用什么样的机制来处理呢？

我认为 OS/360 项目做得非常好。O. S. Locken 强烈要求制定结构良好的工作手册，他本人在他的前一个项目 1410-7010 操作系统中，看到

了工作手册的效果。

我们很快决定每一个编程人员应该了解所有的材料，即在每间办公室中应保留一份工作手册的拷贝。

工作手册的实时更新是非常关键的。工作手册必须是最新的，如果每次变更都要重新打印所有的文档，那将很难做到。但是，如果采用活页夹的方式，则仅需更换变更页。我们当时拥有计算机文本编辑系统，它对实时维护有着不可思议的帮助。编辑、排版、打印的工作直接在计算机和打印机上完成，周转时间不到一天。即便如此，所有接收的人员还是会面临消化理解的问题。当他第一次收到更改页时，他需要知道，“修改了什么？”迟些时候，当他就问题进行咨询时，他需要知道，“现在的定义是什么？”

后一种需要可以通过不间断的文档维护来解决。文档变更的强调有若干个步骤。首先，必须在页面上标记发生改变的文本，例如，使用页边上的竖线标记每行变化的文字；第二，分发的变更页附带简短、独立的总结性文字，对变更的重要性以及批注进行记录。

这种机制在我们项目中碰到别的问题之前，稳定运行了 6 个月。工作手册大约厚达 5 英尺。如果将我们在曼哈顿 Time-Life 大厦办公室里所使用的 100 份手册叠在一起，它们比这座大厦还要高。另外，每天分发的变更页大约有 2 英寸厚，归入档案的页数大概有 150 页。工作手册的日常维护工作占据了每个工作日的大量时间。

这个时候，我们换用了微缩胶片，在为每个办公室配备了微缩胶片阅读机之后，节约了大量金钱，工作手册的体积减少了 18 倍。更重要的是，对数百页更新工作的帮助——微缩胶片大大地减轻了归档问题。

微缩胶片有它的缺点。从管理的角度而言，笨拙的文字归档工作确保了所有变更都会被阅读，这正是工作手册要达到的目的。微缩胶片使工作手册的维护工作变得过于简单，除非列举变化的文字说明和变更胶片一起分发。

另外，读者不能在微缩胶片上做强调、标记和批注。对作者来说，采用文档方式与读者沟通更加有效；对读者来说，文档也更加方便使用。

总之，我觉得微缩胶片是一种非常好的方法。对于大型项目，我建议把它作为文字工作手册的补充。

现在如何入手？在当今很多可以应用的技术中，我认为一种选择是采用可以直接访问的文件。在文件中，记录修订日期记录和标记变更标识条。每个用户可以从一个显示终端(打印机太慢了)来查阅。每日维护的变更小结以“后进先出”(LIFO)的方式保存，在一个固定的地方提供访问。编程人员可以每天阅读，如果错过了一天，他只需在第二天多花一些时间。在他查看小结的同时，他可以停下来，去查询变更的文字。

注意：工作手册本身没有发生变化。它还是所有项目文档的集合，根据某种经过细致设计的规则组织在一起。唯一发生改变的地方是分发机制和查询方法。斯坦福研究院的 D. C. Engelbart 和同事开发了一套系统，并用它在 ARPA 网络项目中建立和维护文档。

Carnegie-Mellon 大学的 D. L. Parnas 提出了更彻底的解决方法[1]。他认为，当编程人员仅了解自己负责的部分，而不是整个系统的开发细节时，工作效率最高。这种方法的先决条件是精确、完整地定义所有接口。这的确是一个彻底的解决方法。如果能处理得好，的确是能解决很多“灾难”。一个好的信息系统不但能暴露接口错误，还能有助于改正错误。

大型编程项目的组织架构

如果项目有 n 个工作人员，则有$(n^2- n)/ 2$个相互交流的接口，有将近$2n$个必须合作的潜在团队。团队组织的目的是减少所需的交流和合作的数量，因此良好的团队组织是解决上述交流问题的关键措施。

减少交流的方法是人力划分(division of labor)和限定职责范围(specialization of function)。当使用人力划分和职责限定时，树状管理结构反映出对详细交流的需要会相应减少。

事实上，树状组织架构是作为权力和责任的结构而出现的。其基本原理——管理角色的非重复性——导致了管理结构是树状的。但是交流的结构并未限制得如此严格，树状结构几乎不能用来描述交流沟通，因为交流是通过网状结构进行的。在很多工程活动领域，树状模拟结构不能很精确地用于描述一般团队、特别工作组、委员会，甚至是矩阵结构组织。

让我们考虑一下树状编程队伍，以及要使它行之有效，每棵子树所必须具备的基本要素。它们是：

(1) 任务(a mission)；

(2) 产品负责人(a producer)；

(3) 技术主管或结构师(a technical director or architect)；

(4) 进度(a schedule)；

(5) 人力的划分(a division of labor)；

(6) 各部分之间的接口定义(interface definitions among the parts)。

所有这些是非常明显和约定俗成的，除了产品负责人和技术主管之间有一些区别。我们先分析以下两个角色，然后再考虑它们之间的关系。

产品负责人的角色是什么？他组建团队，划分工作及制定进度表。他争取，并一直保证必要的资源。这意味着他主要的工作是与团队外部进行向上的沟通和水平的沟通。他建立团队内部的沟通和报告方式。最后，他确保进度目标的实现，根据环境的变化调整资源和团队的构架。

那么技术主管的角色是什么？他对设计进行构思，识别系统的子部分，指明从外部看上去的样子，勾画它的内部结构。他提供整个设计的一致性和概念完整性；他控制系统的复杂程度。当某个技术问题出现时，他提供问题的解决方案，或者根据需要调整系统设计。用 Al Capp 所喜欢的一句谚语，他是“攻坚小组中的独行侠(inside-man at the skunk works)”。他的沟通交流在团队中是首要的。他的工作几乎完全是技术性的。

现在可以看到，这两种角色所需要的技能是非常不同的。这些技能可以按不同的方式进行组合。产品负责人和技术主管所拥有的特殊技能可以用不同方式组合，组合结果控制和支配了他们之间的关系。团队的搭建必须根据参与的人员来组织，而不是将人员纯粹地按照理论进行安排。

存在三种可能的关系，它们都在实践中得到了成功的应用。

产品负责人和技术主管是同一个人。这种方式可以非常容易地应用在小型的队伍中，这样的队伍可能有 3～6 个开发人员，而在大型的项目中则不容易获得应用。原因有两个：第一，同时具有管理技能和技术技能的人很难找到。思考者很少，实干家更少，既是思考者又是实干家的太少了。第二，大型项目中，每个角色都必须全职工作，甚至还要加班。对产品负责人来说，很难在承担全部管理责任的同时，还能抽出时间进

行技术工作。对技术主管来说，很难在保证设计的概念完整性，没有任何妥协的前提下，担任管理工作。

产品负责人作为总指挥，技术主管充当其左右手。这种方法有一些困难。很难在技术主管不参与任何管理工作的同时，建立其在技术决策上的**权威**。

显然，产品负责人必须预先声明技术主管的技术权威，在即将出现的绝大部分测试用例中，他必须支持后者的技术决定。要达到这一点，产品责任人和技术主管必须在基本的技术理论上具有相似的观点；他们必须在主要的技术问题出现之前，先私下讨论这些问题；产品责任人必须对技术主管的技术才能表现出尊重。

另外，还有一些技巧。例如，产品责任人可以通过一些微妙状态特征暗示(如，办公室的大小、地毯、装修、复印机等)来体现技术主管的威信，尽管他身在管理团队之外，但他是决策的根源。

这种组合可以使工作很有效。不幸的是，它很少被应用。不过，它至少有一个好处，即项目经理可以使用并不很擅长管理的技术天才来完成工作。

技术主管作为总指挥，产品负责人充当其左右手。Robert Heinlein在《出售月球的人》(*The Man Who Sold the Moon*)中，用一幅场景描述了这样的安排：

Coster低下头，双手捂着脸，接着，抬起头。“我知道。我了解需要做什么——但每次我试图解决技术问题时，总有一些该死的笨蛋要我做一些关于卡车合同，或者客户电话，或者其他一些讨厌的事情。我很抱歉。Harriman先生，我原以为我可以处理好。”

Harriman非常温和地说：“Bob，别让这些事烦你。近来好像睡眠不

大好，是吗？告诉你吧，我将在你的位子上干几天，为你搭建一个免受这些事情干扰的环境。我需要你的大脑工作在反向量、燃油效率和压力设计上，而不是在卡车的合同上。”Harriman 走到门边，扫了一圈，点了一个可能是、也可能不是办公室主要职员的工作人员。“嘿，你！过来一下。”

那个人看上去有些惊慌，站了起来，走到门边说道，“什么事？”

“把角落上的那个桌子和上面所有的东西搬到本层楼的一个空的办公室去，马上。”

他监督着 Coster 和他的桌子移到另一个办公室，看了看，发现新办公室的电话没有接上。接着，想了一下，搬了一个长沙发过来。“今晚我们将安装一个投影仪、绘图仪、书架和其他一些东西，”他告诉 Coster。“把你工程所需要的东西列一个表。”他回到了原来的总工程师办公室，愉快地想了想如何进行工作组织，以及是否有什么不妥。

过了 4 个小时，他带 Berkeley 进来，与 Coster 会面。这位总工程师正在他的桌子上睡觉，头枕在臂弯里。Harriman 慢慢地退出去，但 Coster 醒了过来。“喔，对不起，”他有点不好意思地说，“我肯定是打了个瞌睡。”

“这就是我给你带来长沙发的原因，”Harriman 说道，“它更加舒适。Bob，来见一下 Jock Berkeley。他是你的新下属。你仍是总工程师，毫无疑问的老板。Jock 负责打理其他事情。从现在起，除了建造登月飞船这样的微妙细节外，你不需要担心任何问题。”

他们握了一下手。“Coster 先生，我只想问一件事，” Berkeley 严肃地说，“所有你需要做的事，我都无权过问——你即将进行一个技术演示——但是看在上帝的份上，能否记录一下，从而让我了解一下。我将会把一个开关放在你的桌上，它会开启我桌上的一个密封的录像机。”

“好的!” Coster 正看着他，Harriman 想，够年轻的。

“如果要做任何非技术的事情，不需要自己动手。只需按一下按钮知会一声，就会有人把这些事情完成!”Berkeley 扫了 Harriman 一眼。“老板说他想同你谈一谈实际的工作。我得先走，去忙去了。”他离开了。

Harriman 坐了下来，Coster 整了整衣服，说道：“喔!”

“感觉好一些了吗？”

“我喜欢 Berkeley 这小伙子的样子。”

“太好了！不用担心，他现在就是你的孪生兄弟。我以前用过他。你可以认为你正住在一个头等的疗养院里。”[2]

这个故事几乎不需要任何的分析解释，这种安排同样能使工作非常有效。

我猜测最后一种安排对小型的团队是最好的选择，如同在第 3 章《外科手术队伍》一文中所述。对于真正大型项目中的一些开发队伍，我认为产品负责人作为管理者是更合适的安排。

巴比伦塔可能是第一个工程上的彻底失败，但它不是最后一个。交流和交流的结果——组织，是成功的关键。交流和组织的技能需要管理者仔细考虑，相关经验的积累和能力的提高同软件技术本身一样重要。

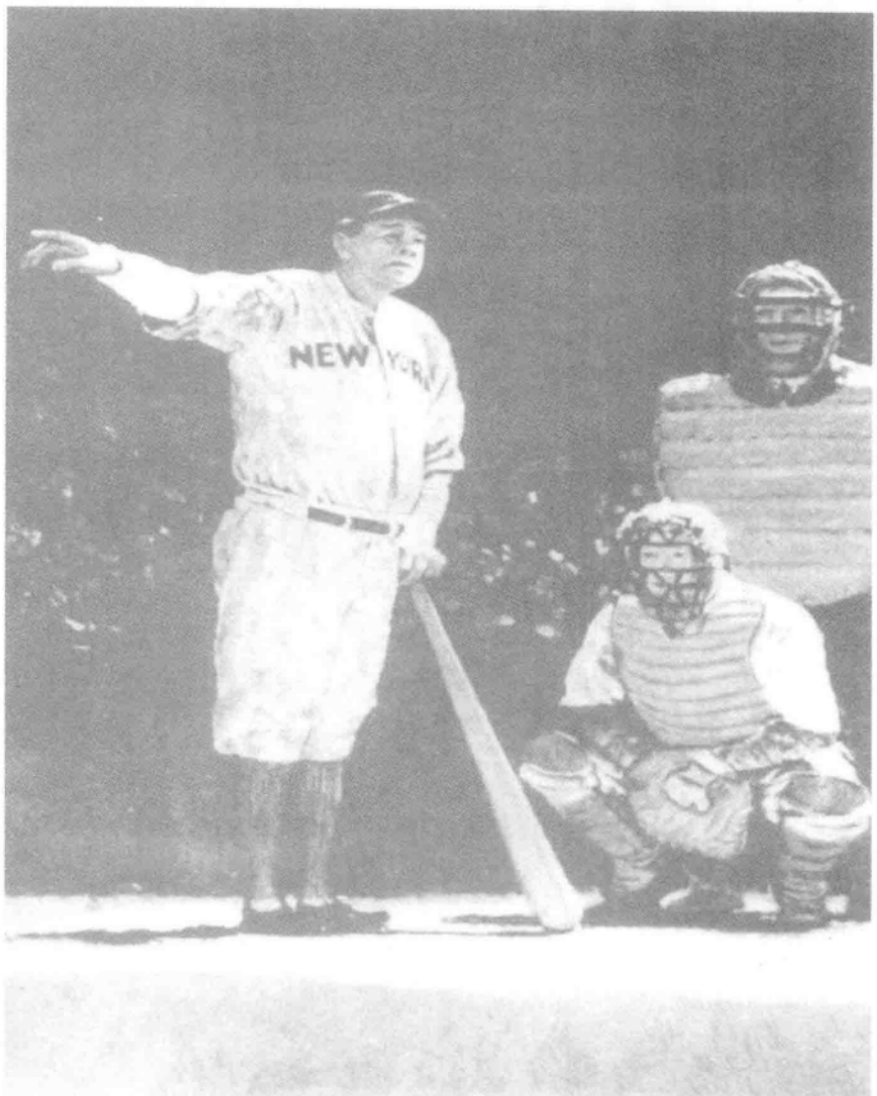
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第8章

胸有成竹

Calling the Shot



“鲁思的本垒打预告”

注：照片是 Douglass Crockwell 在 1932 年世界职业棒球锦标赛上拍摄的。

资料来源：Reproduced by permission of Esquire Magazine and Douglass Crockwell，© 1945(renewed 1973)by Esquire, Inc., and courtesy of the National Baseball Museum

实践是最好的老师。

——普布里乌斯

实践是最好的老师，但智者还能从其他的地方有所收获。

——《穷理查年鉴》

Practice is the best of all instructors.

PUBLILIUS

Experience is a dear teacher，but fools will learn at no other.

POOR RICHARD'S ALMANAC

系统编程需要花费多长时间？需要多少的工作量？如何进行估计？

先前，我推荐了用于计划进度、编码、构件测试和系统测试的比率。首先，需要指出的是，仅仅通过对编码部分的估计，然后应用上述比率，是无法得到对整个任务的估计的。编码大约只占了问题的 1/6 左右，编码估计或者比率的错误可能会导致不合理的荒谬结果。

第二，必须声明的是，构建独立小型程序的数据不适用于编程系统产品。对规模平均为 3 200 指令的程序，如 Sackman、Erikson 和 Grant 的报告中所述，大约单个的程序员所需要的编码加调试时间为 178 个小时，由此可以外推得到每年 35 800 语句的生产率。而规模只有一半的程序花费时间大约仅为前者的 1/4，相应推断出的生产率几乎是每年 80 000 代码行[1]。计划、编制文档、测试、系统集成和培训的时间必须被考虑在内。因此，上述小型项目数据的外推是没有意义的。就好像把 100 米短跑记录外推，得出人类可以在 3 分钟之内跑完 1 英里的结论一样。

在将上述观点抛开之前，尽管不是为了进行严格的比较，我们仍然可以留意到一些事情。即使在不考虑相互交流沟通，开发人员仅仅回顾自己以前工作的情况下，这些数字仍然显示出工作量是规模的幂函数。

图 8-1 讲述了这个悲惨的故事。它阐述了 Nanus 和 Farr[2]在 System Development 公司所做研究，结果表明该指数为 1.5，即，

$$\text{工作量} = \text{常数} \times \text{指令的数量}^{1.5}$$

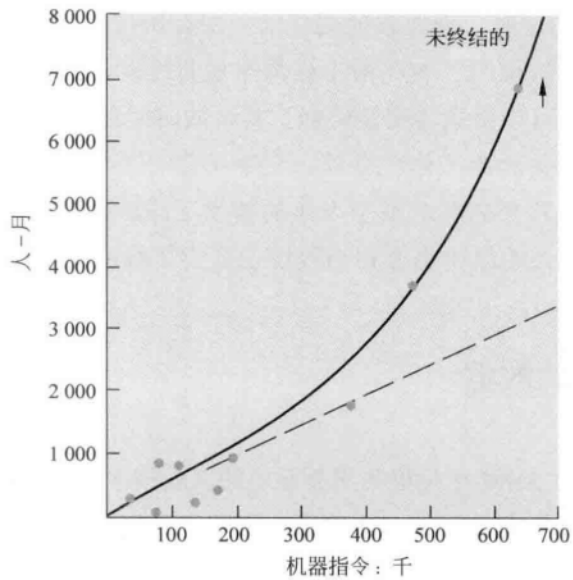


图 8-1 编程工作量是程序规模的函数

Weinwurm[3]的 SDC 研究报告同样显示出指数接近于 1.5。

现在已经有了一些关于编程人员生产率的研究，提出了几种估计的技术。Morin 对所发布的数据进行了一些调查研究[4]。这里仅仅给出了若干特别突出的条目。

Portman 的数据

曼彻斯特 Computer Equipment Organization(Northwest)的 ICL 软件部门的经理 Charles Portman，提出了另一种有用的个人观点。[5]

他发现他的编程队伍落后进度的大约 1/2，每项工作花费的时间大约是估计的 2 倍。这些估计通常是非常仔细的，由很多富有经验的团队完成。他们估算过波特图(PERT chart)上数百个子任务(用人小时作为单

位)。当偏移出现时，他要求他们仔细地保存所使用时间的日志。日志显示，他的团队仅用了50%的工作周来进行实际的编程和调试，估算上的失误完全可以由该情况来解释。其余的时间包括机器的当机时间、高优先级的无关琐碎工作、会议、文字工作、公司业务、疾病、事假等。简言之，项目估算对每个人年的技术工作时间数量做出了不现实的假设。我个人的经验也在相当程度上证实了他的结论。[6]

Aron 的数据

Joel Aron，IBM 在马里兰州盖兹堡的系统技术主管，在他所参与过的 9 个大型项目(简要地说，大型意味着程序员的数目超过 25 人，将近 30 000 行的指令)的基础上，对程序员的生产率进行了研究。[7]他根据程序员(和系统部分)之间的交互划分这些系统，得到了如下的生产率：

非常少的交互	10 000 指令每人年
少量的交互	5 000
较多的交互	1 500

该人年数据未包括支持和系统测试活动，仅仅是设计和编程。当这些数据被除以 2，以包括系统测试的活动时，它们与 Harr 的数据非常的接近。

Harr 的数据

John Harr，贝尔电话实验室电子交换系统的编程经理，在 1969 年春季联合计算机会议(Spring Joint Computer Conference)的论文中汇报了他和其他人的经验。[8]这些数据如图 8-2、图 8-3 和图 8-4 所示。

这些图中，图 8-2 是数据最详细和最有用的。前两个任务是基本的控制程序，后两个是基本的语言翻译。生产率以经调试的指令/人年来表达，它包括了编程、构件测试和系统测试。但其不清楚包括了多少计划、硬件机器支持、文书工作等类似活动的工作量。

	程序单元	程序员人数	年	人年	程序指令	指令/人年
操作性	50	83	4	101	52 000	515
维护	36	60	4	81	51 000	630
编译器	13	9	$2\frac{1}{4}$	17	38 000	2 230
语言解释器(汇编)	15	13	$2\frac{1}{2}$	11	25 000	2 270

图 8-2　4 个 NO.1 的 ESS 编程工作总结

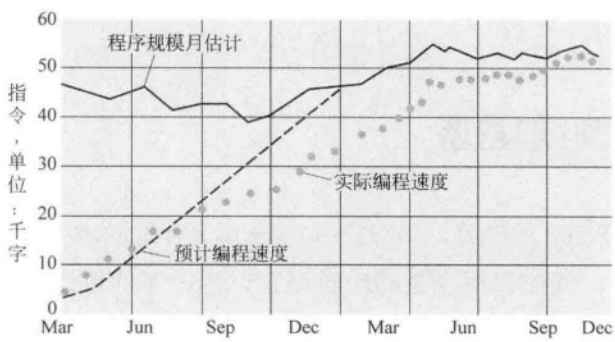


图 8-3　ESS 预计和实际的编程程度

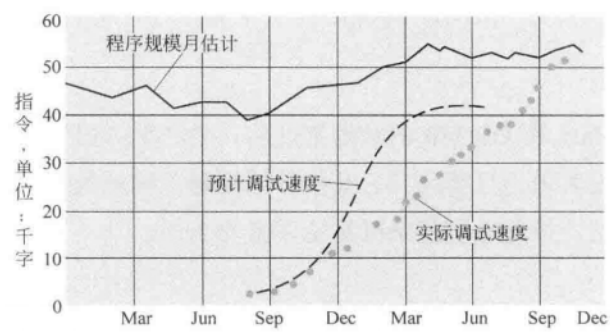


图 8-4　ESS 预计和实际的调试速度

生产率同样地被划分为两个类别：控制程序的生产率大约是 600 指令每人年，语言翻译大约是 2 200 指令每人年。注意所有的 4 个程序都具有类似的规模——差异在于工作组的大小、时间的长短和模块的个数。那么，哪一个是原因，哪一个是结果呢？是否因为控制程序更加复杂，所以需要更多的人员？或者因为它们被分派了过多的人员，所以要求有更多的模块和更多的人月？是因为复杂程度非常高，还是分配较多的人员，导致花费了更长的时间？没有人可以确定。控制程序确实更加复杂。除开这些不确定性，数据反映了实际的生产率——描述了在现在的编程技术下，大型系统开发的状况。因此，Harr 数据的确是真正的贡献。

图 8-3 和图 8-4 显示了一些有趣的数据，将实际的编程速度、调试速度与预期做了对比。

OS/360 的数据

IBM OS/360 的经验，尽管没有 Harr 那么详细的数据，但还是证实了那些结论。就控制程序组的经验而言，生产率的范围大约是 600～800(经过调试的指令)/人年。语言翻译小组所达到的生产率是 2 000～3 000(经过调试的指令)/人年。这包括了小组的计划、代码构件测试、系统测试和一些支持性活动。就我的观点来说，它们同 Harr 的数据是可比的。

Aron、Harr 和 OS/360 的数据都证实，生产率会根据任务本身复杂度和困难程度表现出显著差异。其指导原则是：编译器的复杂度是批处理程序的 3 倍，操作系统复杂度是编译器的 3 倍。[9]

Corbató的数据

Harr 和 OS/360 的数据都是关于汇编语言编程的，好像使用高级语言系统编程的生产率数据公布得很少。Corbató的 MIT 项目 MAC 报告表示在 MULTICS 系统(大约在 100 万～200 万指令之间)上，平均生产率是 1 200 行经调试的 PL/I 语句/人年。[10]

该数字非常令人兴奋。如同其他的项目，MULTICS 包括了控制程序和语言翻译程序。和其他项目一样，它产出的是经过测试和文档化的系统编程产品。在所包括的工作类型方面，数据看上去是可以比较的。该数字是其他项目中控制程序和翻译器程序生产率的平均值。

但 Corbató的数字是行/人年，而不是**字指令**。系统中的每个语句对应于手写代码的 3～5 个指令。这意味着两个重要的结论：

- 对常用编程语句而言，生产率似乎是固定的。这个固定的生产率包括了编程中需要注释并可能存在错误的情况；[11]
- 使用适当的高级语言，编程的生产率可以提高 5 倍。[12]

CHAPTER 9

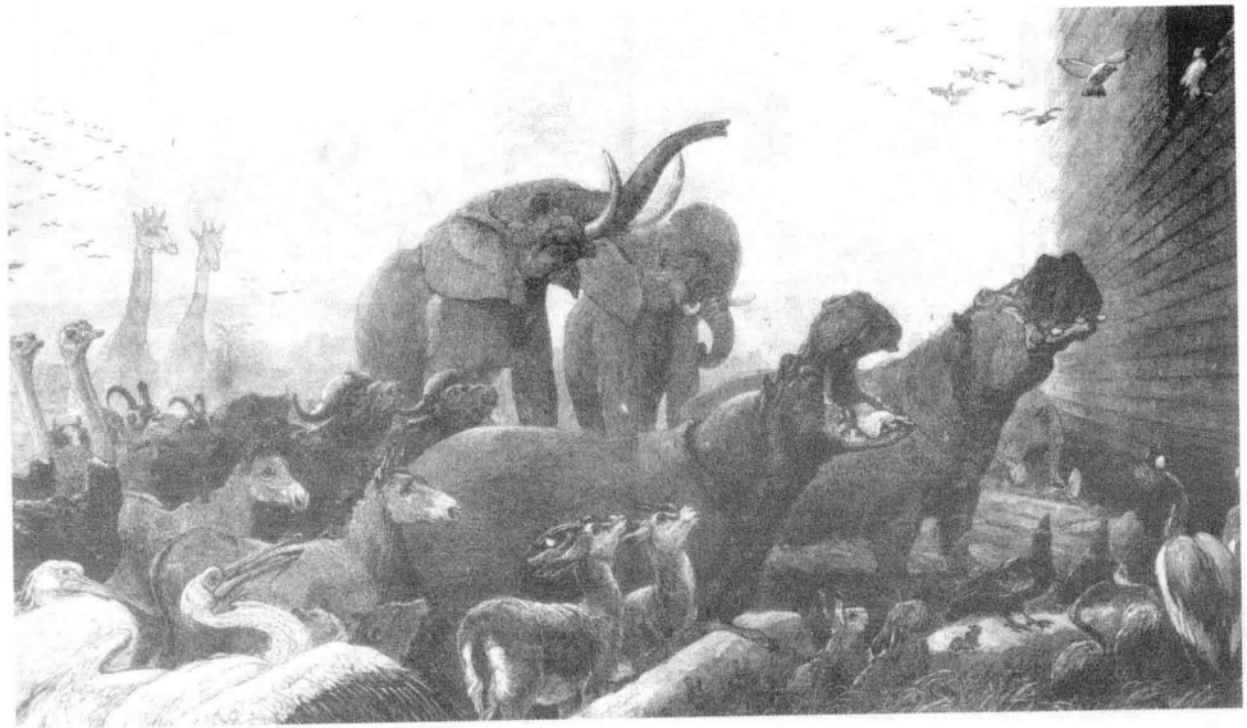
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第9章

削足适履

Ten Pounds in a Five-Pound Sack



H. 哈迪画作的雕版画《它们瞪大眼睛紧盯着诺亚方舟》

资料来源：The Bettman Archive

他应该瞪大眼睛紧盯着诺亚，然后……好好学习一下，看他们当时是怎样把那么多东西塞进一个小小的方舟上的。

——西德尼·史密斯，《爱丁堡评论》

The author should gaze at Noah, and ...learn, as they did in the Ark, to crowd a great deal of matter into a very small compass.

SYDNEY SMITH, *EDINBURGH REVIEW*

作为成本的程序空间

程序有多大？除了运行时间以外，它所占据的空间也是主要开销。这同样适用于专用开发的程序，用户支付给开发者一笔费用，作为必要分担的开发成本。考虑一下 IBM APL 交互式软件系统，它的租金为每月 400 美元，在使用时，它至少占用 160K 字节的内存。在 Model 165 上，内存租金大约是 12 美元/每月·千字节。如果程序在全部时间内都可用，他需要支付 400 美元的软件使用费和 1 920 美元的内存租用费。如果某个人每天使用 APL 系统 4 小时，他每月需要支出 400 美元的软件租金和 320 美元的内存租用费。

常常听到的一个“可怕的”谈论是在 2M 内存的机器上，操作系统就需要占用 400K 内存。这种言论就好像批评波音 747 飞机，仅仅因为它耗资 2 700 万美元一样无知。我们首先必须问的是“它能干什么？”对于所耗费的资金，获得的易用性和性能(高效的系统利用)是什么？投资在内存上的每月 4 800 美元的租金能否比用在其他硬件、编程人员、应用程序上更加有效？

当系统设计者认为对用户而言，常驻程序内存的形式比加法器、磁盘等更加有用时，他会将硬件实现中的一部分移到内存上。相反，其他的做法是非常不负责任的。所以，应该从整体上进行评价。没有人可以在自始至终提倡更紧密的软硬件设计集成的同时，又仅仅就规模本身对软件系统提出批评。

由于规模是软件系统产品用户成本中如此大的一个组成部分，开发人员必须设置规模的目标，控制规模，考虑减小规模的方法，就像硬件开发人员会设立元器件数量目标，控制元器件的数量，想出一些减少零件的方法。同任何开销一样，规模本身不是坏事，但不必要的规模是不可取的。

规模控制

对项目经理而言，规模控制既是技术工作的一部分，也是管理工作的一部分。他必须研究用户和用户需求，以设置待开发系统的规模。接着，把这些系统划分成若干部分，并设定每个部分的规模目标。由于“规模—速度”权衡方案的结果在很大的范围内变化，规模目标的设置是一件颇具技巧的事情，需要对每个可用方案有深刻的了解。聪明的项目经理还会给自己预留一些空间，在工作推行时分配。

在 OS/360 项目中，即使所有的工作都完成得相当仔细，我们依然从中得到了一些痛苦的教训。

首先，仅对核心程序设定规模目标是不够的，必须把所有方面的规模都编入预算。在先前的大多数操作系统中，系统驻留在磁带上，长时间的磁带搜索意味着它无法自如地运用在程序片段上。OS/360 和它的前任产品 Stretch 操作系统和 1410-7010 磁盘操作系统一样，是驻留在磁盘上的。它的开发者对自由、廉价的磁盘访问感到欣喜。而如果使用磁带，会给性能带来灾难性的后果。

在为每个单元设立核心规模的同时，我们没有同时设置访问的预算。正如大家能想到的一样，当程序员发现自己的单元核心未能达到要求时，他会把它分解成覆盖模块。这个过程本身增加了程序整体的规模，并降低了运行速度。最重要的是，我们的管理控制系统既没有度量，也没有捕获这些问题。每个人都汇报了内核的大小，由于都在目标范围之内，所以没有人发现规模上的问题。

幸运的是，OS/360 性能模拟程序投入使用的时间较早。第一次运行的结果反映出很大的麻烦。Fortran H 在带磁鼓的 Model 65 上，每分钟模拟编译 5 条语句。嵌入的例程显示控制程序模块进行了很多次磁盘访问。甚至使用频繁的监控模块也犯了很多同样的错误，结果很类似于页

面的切换。

第一个道理很清楚：和制定驻留空间预算一样，应该制定**总体**规模的预算；和制定规模预算一样，应该制定后台存储访问的预算。

下一个教训十分相似。在每个模块分配功能之前，已编制了空间的预算。其结果是，任何在规模上碰到问题的程序员，会检查自己的代码，看是否能将其中一部分扔给其他人。因此，控制程序所管理的缓冲区成为用户空间的一部分。更严重的是，所有的控制模块都有相同的问题，彻底影响了系统的稳定和安全性。

所以，第二个道理也很清晰：在指明模块有多大的同时，确切定义模块的功能。

第三个更深刻的教训体现在以上的经验中。项目规模本身很大，缺乏管理和沟通，以至于每个团队成员认为自己是争取小红花的学生，而不是构建系统软件产品的人员。为了满足目标，每个人都在局部优化自己的程序，很少会有人停下来，考虑一下对客户的整体影响。对大型项目而言，这种导向和缺乏沟通是最大的危险。在整个实现的过程期间，系统结构师必须保持持续的警觉，确保连贯的系统完整性。在这种监督机制之外，是实现人员自身的态度问题。培养开发人员从系统整体出发、面向用户的态度是软件编程管理人员最重要的职能。

空间技能

空间预算的多少和控制并不能使程序规模减小，为实现这一目标，它还需要一些创造性和技能。

显然，在速度保持不变的情况下，更多的功能意味着需要更多的空间。所以，其中的一个技巧是用功能交换尺寸。这是一个较早的、影响较深远的策略问题：为用户保留多少这种选择？程序可以有很多的选择功能，每个功能仅占用少量的空间。也可以设计成拥有若干选项分组，根据选项组来剪裁程序。任何一系列特殊选项被合并在一起进行分组时，程序需要的空间较少。这很像小汽车。如果把照明灯、点烟器和时钟作为整个配件来标明价格，则成本会比单独提供这些选择所需要的成本低。所以，设计人员必须决定用户可选项目的粗细程度。

当内存大小处于一定范围内的情况下进行系统设计时，会出现另外一个基本问题。内存受限的后果是即使最细密的功能模块，它的适用范围也难以得到推广。在最小规模的系统中，大多数模块被覆盖，系统的大部分常驻空间，会被用做其他部分的交换页面或暂存区。它的尺寸决定了所有模块的尺寸。而且将功能分解到很小的模块会耗费空间和降低性能。所以，当可以提供20倍暂存区空间的大型系统使用这些模块时，节省的也仅仅是访问次数，仍然会因为模块的规模过小引起空间和速度上的损失。这样一来，其限制了用小型系统的模块构造出非常高效的系统。

第二个技能是考虑“空间—时间”的折中。对于给定的功能，空间越多，速度越快。这一点在很大的范围内都适用。也正是这一点使空间预算成为可能。

项目经理可以做两件事来帮助他的团队取得良好的“空间—时间”折中。一是确保他们在编程技能上得到培训，而不仅仅是依赖他们自己的才能和先前的经验。特别是使用新语言或者新机器时，培训显得尤其重要。若要熟练使用，其往往需要快速的学习和经验的广泛共享，也许它应该伴随新技术特别的奖励或者表扬。

另外一种方法是认识到编程需要技术积累，需要开发很多公共单元构件。每个项目要有能用于队列、搜索、散列和排序的例程或者宏库。对于每项功能，库至少应该有两个程序实现：运行速度较快和短小精炼的。上述的公共库开发是一件重要的实现工作，它可以与系统设计工作并行进行。

数据的表现形式是编程的根本

精湛的技艺出自创造，精炼、充分和快速的程序也是如此。技艺改进的结果往往是战略上的突破，而不仅仅是技巧上的提高。这种战略上的突破有时是一种新的算法，如 Cooley-Tukey 快速傅立叶变换，或者是将比较算法的复杂度从 n^2 降低到 $n \log n$。

更普遍的是，战略上突破常来自数据或表的重新表达——这是程序的核心所在。如果提供了程序流程图，而没有表数据，我仍然会很迷惑。而给我看表数据，往往就不再需要流程图，程序结构是非常清晰的。

重新表达所带来好处的例子比容易被找到。我记得有一位年轻人承担了为 IBM 650 开发精密控制台解释器的任务。他发现用户交互的频率低、速度慢，但所占用的空间却很昂贵。于是，他编写了一个解释器的解释器，使得最后程序所占的空间减少到不可思议的程度。Digitek 小而优雅的 Fortran 编译器使用了非常密集的、专业化的代码来表达自己，以至于不再需要外部存储。对这种表达方式解码会损失一些时间，但由于避免了“输入—输出”，反而得到了 10 倍的补偿。(相关示例，请参见 Brooks 和 Iverson 在 *Automatic Data Processing*[1]一书在第 6 章末尾处的练习，以及 Knuth 在 *The Art of Computer Programming*[2]一书中的练习。)

由于缺乏空间而绞尽脑汁的编程人员，常常能通过从自己的代码中挣脱出来，回顾、分析实际情况，仔细思考程序的数据，最终获得非常好的结果。实际上，数据的表现形式是编程的根本。

CHAPTER 10

THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 10 章

提 纲 挈 领

The Documentary Hypothesis

W. 本戈 1897 年的绘画《旧国会图书馆堆积如山的文档》(*Scene in the old Congressional Library*)

资料来源：The Bettman Archive

前提：

在堆积如山的文件资料中，少数文档是关键枢纽，每一件项目管理的工作都围绕着它们运转。这些文档是项目经理最重要的个人工具。

The hypothesis:

Amid a wash of paper, a small number of documents become the critical pivots around which every project's management re-volves. These are the manager's chief personal tools.

技术、周边组织机构、行业传统等若干因素凑在一起，定义了项目必须准备的一些文书工作。对于一个刚从技术人员中任命的项目经理来说，这简直是一件彻头彻尾、令人生厌的事情，而且是毫无必要和令人分心的，让他充满了被吞没的威胁。但是，在实际工作中，大多数情况都是这样的。

慢慢地，他逐渐认识到这些文档的某些部分包含和表达了一些管理方面的工作。每份文档的准备工作是集中考虑，并使各种讨论意见明朗化的主要时刻。如果不这样，项目往往会处于无休止的混乱状态中。文档的跟踪维护是项目监督和预警的机制。文档本身可以作为检查列表、状态控制，也可以作为汇报的数据基础。

为了阐明软件项目如何开展这项工作，我们首先借鉴一下其他行业一些有用的文档资料，看是否能进行归纳，得出结论。

计算机产品的文档

如果要制造一台机器，哪些是关键的文档呢？

目标：定义待满足的目标和需要，定义迫切需要的资源、约束和优先级。

技术说明：计算机手册加性能规格说明。它是在计划新产品时第一个产生，并且最后完成的文档。

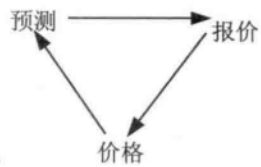
进度

预算：预算不仅仅是约束。对管理人员来说，它还是最有用的文档之一。预算的存在会迫使其制定技术决策，否则，技术决策很容易被忽略。更重要的是，它促使和澄清了策略上的一些决定。

组织结构图

工作空间的分配

报价、预测、价格：这三个因素互相牵制，决定了项目的成败。



为了进行市场预测，首先需要制定产品性能说明和确定假设的价格。从市场预测得出的数值，连同从设计得出的组件单元的数量，决定了生产的估计成本，进而可以得到每个单元的开发工作量和固定的成本。这些成本又决定了价格。

如果价格低于假设值，令人欣慰的循环开始了。预测值较高，单元成本较低，因此价格能够继续降低。

如果价格高于预测值，灾难性的循环开始了，所有的人必须努力奋斗来打破这个循环。新应用程序必须提高性能和支持更高的市场预测。成本必须降低，以获得更低的报价。这个循环的压力常常是激励市场人员和工程师工作的最佳动力。

同时，它也会带来可笑的踌躇和摇摆。我记得曾经有一个项目，在三年的开发周期中，机器指令计数器的设计每 6 个月变化一次。在某个阶段，需要好一点的性能时，指令计数器采用触发器来实现；下一个阶段，降低成本是主要的焦点，指令计数器采用内存来实现。在另一个项目中，我所见过的最好的一个项目经理常常充当一个大型调速轮的角色，他的惯性降低了来自市场和管理人员的起伏波动。

大学科系的文档

除了目的和活动上的巨大差异，数量类似、内容相近的各类文档形成了大学系主任的主要资料集合。校长、教师会议或系主任的每一个决定几乎都是一个技术说明，或者是对这些文档的变更。

- 目标
- 课程描述
- 学位要求
- 研究报告(申请基金时，还要求计划)
- 课程表和课程的安排
- 预算
- 教室分配
- 教师和研究生助手的分配

注意这些文档的组成与计算机项目非常相似：目标、产品说明、时间安排、资金分配、空间分派和人员的划分。只有价格文档是不需要的，校董事会完成这项任务。这种相似性不是偶然的——任何管理任务的关注焦点都是时间、地点、人员、项目内容和资金。

软件项目的文档

在许多软件项目中，开发人员从商讨结构的会议开始，然后开始书写代码。不论项目的规模有多小，项目经理聪明的做法都是：首先立刻

正式生成若干正式的小文档，以作为自己的数据基础，然后再要求提供和其他经理类似的文档。

- **内容：目标。**定义待完成的目标、迫切需要的资源、约束和优先级。
- **内容：产品技术说明。**以建议书开始，以用户手册和内部文档结束。速度和空间说明是关键的部分。
- **时间：进度**
- **资金：预算**
- **地点：工作空间分配**
- **人员：组织图。**它与接口说明是相互依存的，如同 Conway 的规律所述："设计系统的组织架构受到产品的约束限制，生产出的系统是这些组织机构沟通结构的映射。"[1]Conway 接着指出，最初反映系统设计的组织架构图，肯定不会是正确的。如果系统设计能自由地变化，则项目组织架构必须为变化做好准备。

为什么要有正式的文档

首先，书面记录决策是必要的。只有记录下来，分歧才会明朗，矛盾才会突出。书写这项活动需要上百次的细小决定，正是由于它们的存在，人们才能从令人迷惑的现象中得到清晰、确定的策略。

第二，文档能够作为同其他人的沟通渠道。项目经理会不断发现，许多理应被普遍认同的策略，完全不为团队的一些成员所知。正因为项目经理的基本职责是使每个人都向着相同的方向前进，所以他的主

要工作是沟通，而不是做出决定。这些文档能极大地减轻他的负担。

最后，项目经理的文档可以作为数据基础和检查列表。通过周期性的回顾，他能清楚项目所处的状态，以及哪些需要重点进行更改和调整。

我并不是很同意销售人员所吹捧的“完全信息管理系统”——管理人员只需在计算机上输入查询，显示屏上就会显示出结果。有许多基本原因决定了上述系统是行不通的。一个原因是只有一小部分管理人员的时间——可能只有 20%——用来从自己头脑外部获取信息。其他的工作是沟通：倾听、报告、讲授、规劝、讨论和鼓励。不过，对于基于数据的部分，少数关键的文档是至关重要的，它们可以满足绝大多数需要。

项目经理的任务是制订计划，并实现计划。但是只有书面计划是精确和可以沟通的。计划中包括了时间、地点、人员、项目内容和资金。这些少量的关键文档封装了项目经理大量的工作。如果一开始就认识到它们的普遍性和重要性，那么就可以将文档作为工具友好地利用起来，而不会让它成为令人厌烦的繁重任务。通过遵循文档开展工作，项目经理能更清晰和快速地设定自己的方向。

CHAPTER 11

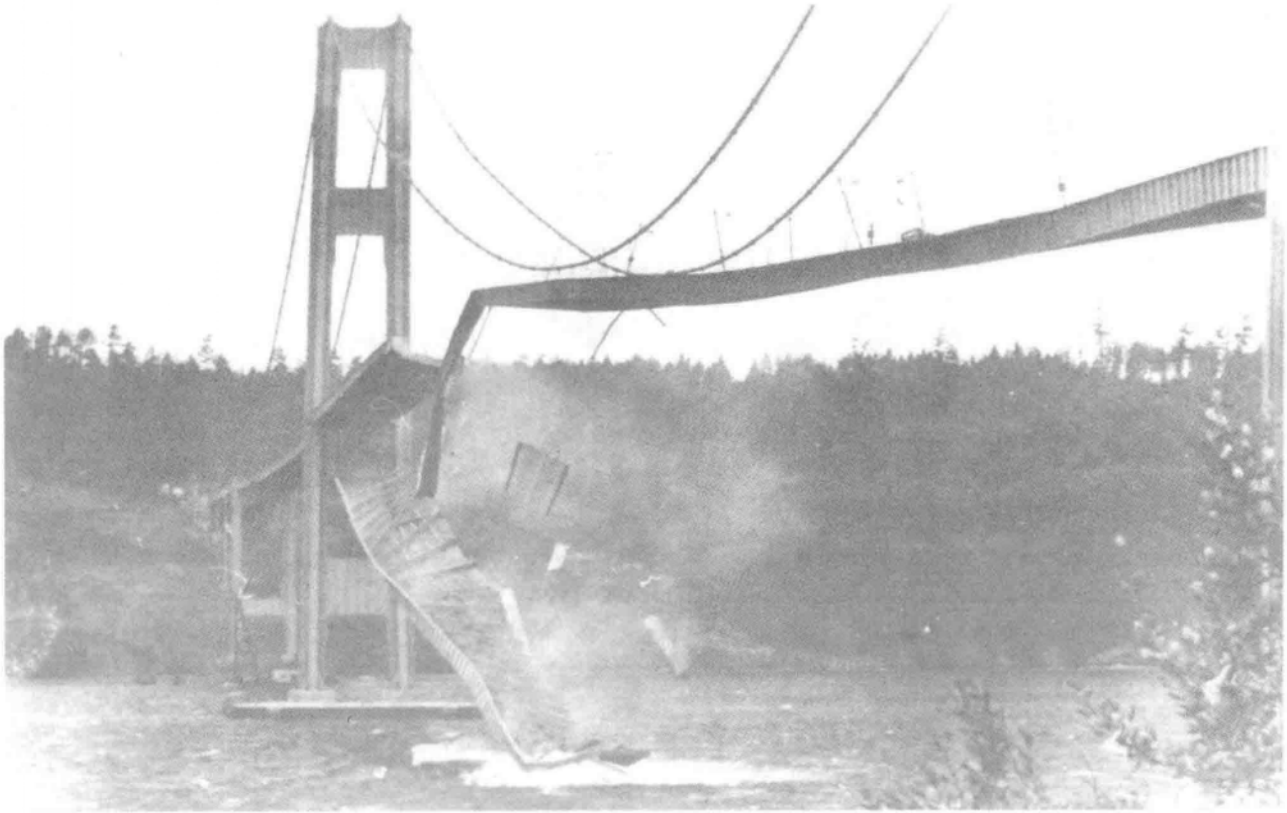
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 11 章

未雨绸缪

Plan to Throw One Away



1940 年塔科马海峡大桥(Tacoma Narrows Bridge)的倒塌现场，倒塌原因是由于空气动力学上的设计失误。

资料来源：UPI Photo/The Bettman Archive

不变只是愿望，变化才是永恒。

——斯威夫特

普遍的做法是，选择一种方法，试试看；如果失败了，没关系，再试试别的方法。不管怎么样，重要的是先去尝试。

——富兰克林·罗斯福[1]

There is nothing in this world constant but inconstancy.

SWIFT

It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.

FRANKLIN D. ROOSEVELT

试验性工厂和增大规模

化学工程师很早就认识到，在实验室可以进行的反应过程，并不能在工厂中一步实现。一个被称为“**试验性工厂**”(pilot plant)的中间步骤是非常必要的，它会为提高产量和在缺乏保护的环境下运作提供宝贵经验。例如，海水淡化的实验室过程会先在产量为 10 000 加仑/天的试验场所测试，然后再用于 2 000 000 加仑/天的净化系统。

软件系统的构建人员也面临类似的问题，但似乎并没有吸取教训。一个接一个的软件项目都是一开始设计算法，然后将算法应用到待发布的软件中，接着根据进度把第一次开发的产品发布给顾客。

对于大多数项目，第一个开发的系统并不合用。它可能太慢、太大，而且难以使用，或者三者兼而有之。要解决所有的问题，除了重新开始以外，没有其他的办法——即开发一个更灵巧或者更好的系统。系统的丢弃和重新设计可以一步完成，也可以一块块地实现。所有大型系统的经验都显示，这是必须完成的步骤。[2]而且，新的系统概念或新技术会不断出现，必须构建一个用来抛弃的系统，因为即使是最优秀的项目经理，也不能无所不知地在最开始解决这些问题。

因此，管理上的问题不再是“**是否**构建一个试验性的系统，然后抛弃它？”你**必须**这样做。现在的问题是：“是否预先计划抛弃原型的开发，或者是否将该原型发布给用户？”从这个角度看待问题，答案更加清晰。将原型发布给用户，可以获得时间，但是它的代价高昂——对于用户，使用起来极度痛苦；对于重新开发的人员，分散了精力；对于产品，影响了声誉，即使最好的再设计也难以挽回名声。

因此，**为舍弃而计划，无论如何，你一定要这样做。**

唯一不变的就是变化本身

一旦认识到试验性的系统必须被构建和丢弃，具有变更思想的重新设计将不可避免，那么，面对整个变化现象就是非常有用的。第一步是接受这样的事实：变化是与生俱来的，不是不合时宜和令人生厌的异常情况。Cosgrove 很有洞察力地指出，开发人员交付的是用户满意程度，而不仅仅是有形的产品。用户的实际需要和用户感觉会随着程序的构建、测试和使用而变化。[3]

当然对于硬件产品而言，同样需要满足要求，无论是新型汽车或是新型计算机。但物体的客观存在容纳和阶段化(量子化)了用户对变更的要求。软件产品易于掌握的特性和不可见性，导致它的构建人员面临永恒的需求变更。

我从不建议顾客所有的目标和需求的变更必须、能够或者应该整合到设计中。项目开始时建立的基准，肯定会随着开发的进行越来越高，甚至开发不出任何产品。

然而，目标上的一些变化无可避免，事先为它们做准备总比假设它们不会出现要好得多。不但目标上的变化不可避免，而且设计策略和技术上的变化也不可避免。抛弃原型概念本身就是对事实的接受——随着学习的过程更改设计。[4]

为变更设计系统

如何为上述变化设计系统，是一个众所周知的问题，在书本上被普遍讨论。它们包括细致的模块化、可扩展的函数、精确完整的模块间接口设计和完备的文档。另外，还可能会采用包括调用队列和表驱动的

一些技术。

最重要的措施是使用高级语言和自文档技术，以减少变更引起的错误。采用编译时的操作来整合标准声明，在很大程度上帮助了变化的调整。

变更的阶段化是一种必要的技术。每个产品都应该有数字版本号，每个版本都应该有自己的日程表和冻结日期，在此之后的变更属于下一个版本的范畴。

为变更计划组织架构

Cosgrove 主张把所有计划、里程碑和日程安排都当做是尝试性的，以方便进行变化。这似乎有些走极端——现在软件编程小组失败的主要原因是管理控制得太少，而不是太多。

不过，他提出了一种卓越的见解。他观察到不愿意为设计书写文档的原因，不仅仅是由于惰性或者时间压力。相反，设计人员通常不愿意提交尝试性的设计决策，再为它们进行辩解。“通过设计文档化，设计人员将自己暴露在每个人的批评之下，他必须能够为他书写的一切进行辩护。如果团队架构因此受到任何形式的威胁，则没有任何东西会被文档化，除非架构是完全受到保护的。”

为变更组建团队比为变更进行设计更加困难。每个人被分派的工作必须是多样的、富有拓展性的工作。从技术角度而言，整个团队可以被灵活地安排。在大型的项目中，项目经理需要有两个或三个顶级程序员作为技术轻骑兵。当工作最密集的时候，他们能高效地解决各种问题。

当系统发生变化时，管理结构也需要进行调整。这意味着，只要管理人员和技术人才的天赋允许，老板必须对他们的能力培养给予极大的关注，使管理人员和技术人才具有互换性。

这其中的障碍是社会性的，人们必须同顽固的戒心做斗争。首先，管理人员自己常常认为高级人员太“有价值”，而舍不得让他们从事实际的编程工作；其次，管理人员拥有更高的威信。为了克服这个问题，一些实验室，如贝尔实验室，废除了所有的职位头衔。每个专业人士都是“技术人员中的一员”。而一些实验室，如 IBM，保持了两条职位晋升线，如图 11-1 所示。相应的级别在概念上是相同的。

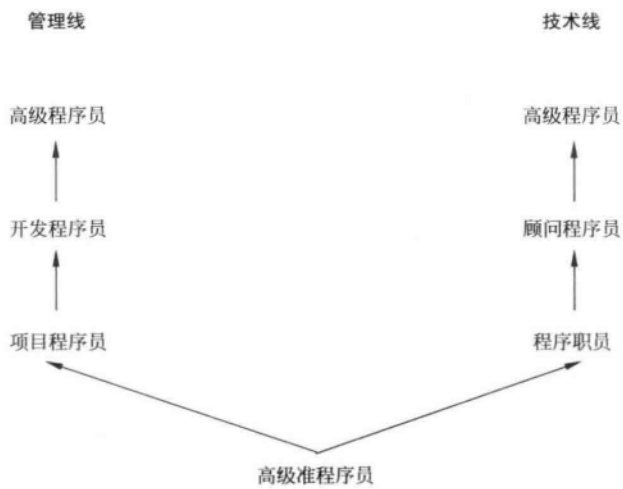


图 11-1　IBM 的两条职位晋升线

为上述层次建立相互一致的薪水级别很容易。但要建立一致的威信，会困难一些。比如，办公室的大小和布局应该相同。秘书和其他支持性人员也必须相同。从技术线向管理线同级调动时，不能伴随着待遇的提升，其应该以“调动”而不是“晋升”的名义。相反，调整则应该伴随着待遇的提高，对于传统意识进行补偿是必要的。

管理人员需要参与技术课程，高级技术人才需要进行管理培训。项目目标、进展和管理问题必须在高级人员整体中得到共享。

只要能力允许，高层人员必须时刻做好技术和情感上的准备，或者管理团队亲自参与开发工作。其工作量很大，但很值得。

组建外科手术队伍式的软件开发团队，其整体观念是对上述问题的彻底冲击。其结果是当高级人才编程和开发时，不会感到自降身份。这种方法试图清除那些会剥夺他创造性工作的乐趣的社会障碍。

另外，上述组织架构的设计是为了最小化成员间的接口。同样地，它使系统在最大程度上易于修改。当组织构架必须变化时，为不同的软件开发任务重新安排整个“外科手术队伍”，会变得相对容易一些。这的确是一个长期、有效的灵活组织构架解决方案。

前进两步，后退一步

在程序发布给顾客使用之后，并不会停止变化。发布后的变更被称为“**程序维护**”，但是软件的维护过程不同于硬件维护。

计算机系统的硬件维护包括了三项活动：替换损坏的器件，清洁和润滑，修改设计上的缺陷。(大多数情况下——但不是全部——变更修复的是实现上，而不是结构上的一些缺陷。对于用户而言，这常常是不可见的。)

软件维护不包括清洁、润滑或对损坏器件的修复。它主要包含对设计缺陷的修复。与硬件维护相比，这些软件变更通常包含了更多的新增功能，它通常是用户能察觉的。

对于一个广泛使用的程序，其维护总成本通常是开发成本的 40%或更多。令人吃惊的是，该成本受用户数目的影响很大。用户越多，所发现的错误也就越多。

麻省理工学院核科学实验室的 Betty Campbell 指出，特定版本的软件发布生命期中一个有趣的循环。如图 11-2 所示。起初，上一个版本中被发现和修复的 bug，在新的版本中仍会出现。新版本中的新功能会产生新的 bug。解决了这些问题以后，程序会正常运行几个月。接着，错误率会重新攀升。Campbell 认为，这是因为用户的使用达到了新的熟练水平，他们开始运用新的功能。这种高强度的考验查出了新功能中很多不易察觉的问题。[5]

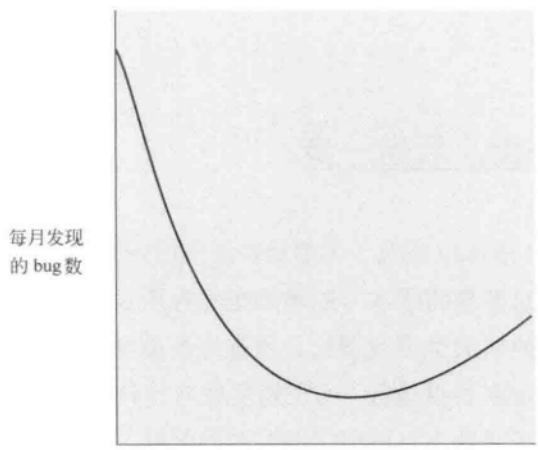


图 11-2　出现的 bug 数量是发布时间的函数

程序维护中的一个基本问题是——缺陷修复总会以固定(20%～50%)的几率引入新的 bug。所以，整个过程是前进两步，后退一步。

为什么缺陷不能更彻底地被修复？首先，看上去很微小的错误，似乎仅仅是局部操作上的失败，实际上却是系统级别的问题，通常这不是

很明显的。修复局部问题的工作量很清晰，并且往往不大。但是，更大范围的修复工作常常会被忽视，除非软件结构很简单，或者文档书写得非常详细。其次，维护人员常常不是编写代码的开发人员，而是一些初级程序员或者新手。

作为引入新 bug 的一个后果，程序每条语句的维护需要的系统测试比其他编程要多。理论上，在每次修复之后，必须重新运行先前所有的测试用例，从而确保系统不会以更隐蔽的方式被破坏。实际情况中，**回归测试**必须接近上述理想状况，所以它的成本非常高。

显然，使用能消除或至少能指明副作用的程序设计方法，会在维护成本上有很大的回报。同样，设计实现的人员越少、接口越少，产生的错误也就越少。

前进一步，后退一步

Lehman 和 Belady 研究了大型操作系统的一系列发布版本的历史。[6] 他们发现模块总数量随版本号的增加呈线性增长，但是受到影响的模块数量随版本号的增加呈指数增长。所有修改都倾向于破坏系统的架构，增加了系统的混乱程度(熵)。用在修复原有设计上瑕疵的工作量越来越少，而早期维护活动本身所引起的漏洞的修复工作越来越多。随着时间的推移，系统变得越来越无序，修复工作迟早会失去根基。每一步前进都伴随着一步后退。尽管系统在理论上一直可用，但实际上，整个系统已经面目全非，无法再成为下一步进展的基础。而且，机器在变化，配置在变化，用户的需求在变化，现实系统不可能永远可用。崭新的、基于原有系统的重新设计是完全必要的。

通过对统计模型的研究，关于软件系统，Belady 和 Lehman 得到了更具普遍意义、为所有经验支持的结论。“事物在最初总是最好的，”

正如 Pascal. C. S. Lewis 所敏锐指出的：

这正是历史的关键。使用卓越的能源，构建文明，成立杰出的机构，但是每次总会出现问题。自私和残酷的人类升到塔尖后，总有一些致命的缺陷，使得一切开始滑落，回到痛苦和废墟之中。实际上，机器失灵了。看上去，就好像是机器启动时很正常，跑了几步，然后垮掉了。[7]

系统软件开发是减少混乱度(减少熵)的过程，所以它本身是处于亚稳态的。软件维护是提高混乱度(增加熵)的过程，即使是最熟练的软件维护工作，也只是放缓了系统退化到非稳态的进程。

CHAPTER 12

THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 12 章

干 将 莫 邪

Sharp Tools

A. 皮萨诺的《雕刻师》(*Lo Scultore*)(约 1335 年刻于佛罗伦萨圣母百花大教堂钟塔)

资料来源：Scala/Art Resource, NY

巧匠因为他的工具而出名。

——谚语

A good workman is known by his tools.

PROVERB

就工具而言，即使是现在，很多软件项目仍然像经营一家五金店。每个骨干人员都仔细地保管自己工作生涯中搜集的一套工具集，这些工具成为个人技能的直观证明。正是如此，每个编程人员也保留着编辑器、排序、内存信息转储和磁盘空间实用程序等工具。

这种方法对软件项目来说是愚蠢的。首先，项目的关键问题是沟通，个性化的工具会妨碍而非促进沟通。其次，当机器和工作语言发生变化时，技术也会随之变化，所有工具的生命周期都是很短的。最后，毫无疑问，开发和维护公共的通用编程工具的效率更高。

不过，仅有通用工具是不够的。专业需要和个人偏好同样需要很多专业工具。所以在前面关于软件开发队伍的讨论中，我建议为每个团队配备一名工具管理人员。这个角色管理所有通用工具，能指导他的客户和老板如何使用工具。同时，他还能编制老板需要的专业工具。

因此，项目经理应该制定一套策略，并为通用工具的开发分配资源。与此同时，他还必须意识到对专业工具的需求，对这类工具的开发不能吝啬人力和物力——这种企图的危害非常隐蔽。可能有人会觉得，将所有分散的工具管理人员集中起来，形成一个公共的工具小组，会有更高的效率，实际上却不是这样。

项目经理必须考虑、计划、组织的工具到底有哪些呢？首先是**计算机设施**。它需要硬件和使用安排策略；它需要**操作系统**，提供服务的方式必须明了；它需要语言，语言的使用方针必须明确；然后是**实用程序**、**调试辅助程序**、**测试用例生成工具**和处理文档的**字处理系统**。接下来我们逐一讨论它们。[1]

目标机器

机器支持可以有效地划分成**目标机器**和**辅助机器**。目标机器是软件所服务的对象，程序必须在该机器上进行最后测试。辅助机器是那些在开发系统中提供服务的机器。如果是在为原有的机型开发新的操作系统，则该机器不仅充当目标机器的角色，也可作为辅助机器。

目标机器的类型有哪些？团队开发的监督程序或其他系统核心软件当然需要它们自己的机器。目标机器系统会需要若干操作员和一两个系统编程人员，以保证机器上的标准支持是及时更新和实时可用的。

如果还需要其他的机器，其将是一件很古怪的事情——运行速度不必非常快，但至少要若干兆字节的主存，百兆字节的在线硬盘和终端。字符型终端即可满足要求，但是它必须比 15 字符/秒的打字机速度要快。大容量内存可以进行功能测试之后的进程覆盖和剪裁工作，从而极大地提高生产率。

另外，还需要配备调试机器或者软件。这样，在调试过程中，所有类型的程序参数可以被自动计数和测量。例如，内存使用模式是非常强大的诊断措施，能查出程序中不合逻辑的行为或者性能意外下降的原因。

进度安排。当目标机器刚刚被研制，且它的第一个操作系统被开发时，机器时间是非常匮乏的，进度安排成了主要问题。目标机器时间需求具有特别的增长曲线。在 OS/360 开发中，我们有很好的 System/360 仿真器和其他的辅助设施，并根据以前的经验，我们计划出 System/360 的使用时间(小时数)，向制造商提前预定了机器。不过，起初它们日复一日地处于空闲状态。突然有一天，所有 16 个系统全部上线，这时资源配给出现了问题。实际使用情况如图 12-1 所示。每个人在同一时间，开始调试自己的第一个组件，然后团队大多数成员持续地进行某些调试工作。

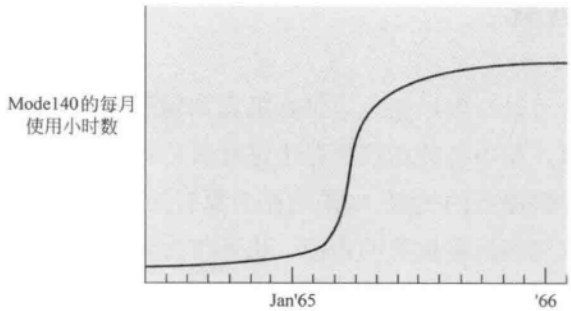


图 12-1　目标机器使用的增长曲线

我们集中了所有的机器和磁带库，并组建了一个富有经验的专业团队来操作它们。为了最大限度地利用 S/360 的时间，我们在任何系统空闲和可能的时间里，以批处理方式运行所有调试任务。我们尝试了每天运行 4 次(周转时间为 2.5 小时)，而实际要求的周转时间为 4 小时。我们使用了一台带有终端的 1401 辅助机器来进行调度，跟踪成千上万的任务，监督周转时间。

但是整个开发队伍实在是过度运转了。在经过了几个月的缓慢周转、相互指责和极度痛苦之后，我们开始把机器时间分配成连续的块。例如，整个从事排序工作的 15 人小组，会得到系统 4～6 小时的使用时间块，由他们自己决定如何使用。即使没有安排，其他人也不能使用机器资源。

这种方式，是一种更好的分配和安排方法。尽管机器的利用程度可能会有些降低(常常不是这样)，生产率却提高了。上述小组中的每个人，6 小时中连续 10 次操作的生产率，比间隔 3 小时的 10 次操作要高许多，因为持续的精力集中能减少思考时间。在这样的冲刺之后，提出下一个时间块要求之前，小组通常需要 1～2 天的时间来从事书面文档工作。并且，通常 3 人左右的小组能卓有成效地安排和共享时间块。在调试新操

作系统时，这似乎是一种使用目标机器的最好方法。

上述方法尽管没有在任何理论中被提及，在实际情况中却一直如此。另外，同天文工作者一样，系统调试总是夜班性质的工作。20年前，当所有机房负责人在家中安睡时，我却不愿意严格遵守作息时间，在黎明开始之前，仍在701上辛勤地工作。三代机器过去了，技术完全改变了，操作系统出现了，大家喜好的工作方式并没有改变。这种工作方式得以延续，是因为它的生产率最高。现在，人们已开始认识到它的生产力，并且敞开地接受这种富有成效的实践。

辅助机器和数据服务

仿真装置。如果目标机器是新产品，则需要一个目标机器的逻辑仿真装置。这样，在生产出新机器之前，就有辅助的调试平台可供使用。同样重要的是——即使在新机器出现之后，仿真装置仍然可以提供**可靠**的调试平台。

可靠并不等于**精确**。在某些方面，仿真机器肯定无法精确地达到与新型机器一致的实现。但是至少在一段时间内，它的实现是**稳定**的，新硬件就不会。

现在，计算机硬件能够自始至终正常工作，对此我们已经习以为常。除非程序开发人员发现相同运算在运行时会产生不一致的结果，否则出错时，他都会在别人的建议下去检查自己代码中的错误，而不是去怀疑他的运行平台。

这样的经验，对于支持新型机器的编程工作来说，是不好的。实验室研制和试制的模型产品或早期硬件**不会**像定义的那样运行，**不会**稳定工作，甚至**不会**保持不变。当发现 bug 时，所有的机器拷贝，包括软件编程小组所使用的，都会被修改。这种飘忽不定的开发基础实在糟糕。而硬件失败，通常是间歇性的，导致情况更加恶劣。不确定性是所有情况中最糟糕的，因为它剥夺了开发人员查找 bug 的动力——也许 bug 根本就不存在。所以，一套运行在稳定辅助平台上的可靠仿真装置，提供了远大于我们所期望的功用。

编译器和汇编平台。出于同样的原因，编译器和汇编软件需要运行在可靠的辅助平台上，为目标机器编译目标代码。接着，可以在仿真装置上立刻开始后续的调试。

高级语言的编程开发中，在目标机器上开始全面测试目标代码之前，编译器可以在辅助机器上完成很多目标代码的调试和测试工作。这为直接运行提供了支持，而不仅仅是稳定机器上的仿真结果。

程序库和管理。在 OS/360 开发中，一种非常成功和重要的辅助机器应用是对程序库的维护。该系统由 W. R. Crowley 带领开发，连接两台 7010 机器，共享一个很大的磁盘数据库。7010 还提供了 System/360 汇编程序。所有经过测试或者正在测试的代码都保存在该库中，包括源代码和汇编装载模块。这个库实际上划分成不同访问规则下的子库。

首先，每个组或者编程人员分配了一个区域，用来存放他的程序拷贝、测试用例以及单元测试需要的测试辅助**平台**。在这个**开发库**(playpen)中，不存在任何限制开发人员的规定。他可以自由处置自己的程序，因为他是它们的拥有者。

当开发人员准备将软件单元集成到更大的部分时，他向集成经理提交一份拷贝，后者将拷贝放置在**系统集成子库**中。此时，原作者不可以再改变代码，除非得到了集成经理的批准。当系统合并在一起时，集成经理开始进行所有的系统测试工作，识别和修补 bug。

有时，系统的一个版本可能会被广泛应用，它被提升到**当前版本子库**。此时，这个拷贝是不可更改的，除非有重大缺陷要修复。该版本可以用于所有新模块的集成和测试。7010 上的一个程序目录对每个模块的每个版本进行跟踪，包括它的状态、用途和变更。

这里有两个重要的理念。首先是**受控**，即程序的拷贝由经理负责，他可以独立地授权程序的变更。其次是使发布的进展变得正式，以及开发库与集成、发布的**正式分离**。

在我看来，这是 OS/360 工作中最优秀的成果之一。它实际上是管理技术的一部分，几个大型的项目都独立地发展了这种技术，包括贝尔实验室、ICL、剑桥大学等。[2]它同样适用于文档，是一种不可缺少的技术。

编程工具。随着新调试技术的出现，旧方法的使用减少了，但并没有消失。因此，还是需要内存转储、源文件编辑、快照转储，甚至跟踪等工具。

与此类似，一整套实用程序同样是必要的，用来实现磁带走带、拷贝磁带、打印文件和更改目录等工作。如果一开始就任命了项目的工具操作和维护人员，那么这些工作可以一次完成，并且随时处在待命状态。

文档系统。在所有的工具中，最能节省劳动力的，可能是运行在可靠辅助平台上的、计算机化的文本编辑系统。我们有一套使用非常方

便的系统，由 J. W. Franklin 发明。没有它，OS/360 手册的进度可能会远远落后，而且更加晦涩难懂。另外，对于 6 英尺厚的 OS/360 手册，很多人认为它是无用的，它的庞大带来了新的疑惑。这种观点有一些道理。

对此，我通过两种途径作出了反应。首先，OS/360 的文档规模是极其庞大的，但阅读计划是被仔细安排的。如果选择性地阅读，则可以忽略大部分内容和省下大量时间。人们必须把 OS/360 的文档看成是图书馆或者百科全书，而不是一系列强制阅读的文章。

第二，OS/360 的文档比那些刻画了大多数编程系统特性的短篇文档更加可取。不过，我也承认，手册仍有某些需要大量改进的地方，经改进后文档篇幅会大大减少。事实上，某些部分(例如概念和设施)已经被很好地改写了。

性能仿真装置。最好有一个性能仿真装置。正如我们将在下章讨论到的，彻底地开发一个性能仿真装置。使用相同的自上向下的设计方法，来实现性能仿真装置、逻辑仿真装置和产品。尽可能早地开始这项工作，仔细地听取“它们表达的意见”。

高级语言和交互式编程

在 10 年前的 OS/360 开发中，并没有使用现在最重要的两种系统编程工具。目前，它们也没有得到广泛应用，但是所有证据都证明它们的功效和适用性。他们是高级语言和交互式编程。我确信只有懒散和惰性会妨碍它们的广泛应用，技术上的困难不再成为借口。

高级语言。使用高级语言的主要原因是生产率和调试速度。我们在前面已讨论过生产率的问题(见第 8 章)。其中，并没有提到大量的数字论据，但是其所体现出来的是整体提升，而不仅仅是部分增加。

调试上的改进来自下列事实——存在更少的 bug，而且更容易查找。bug 更少的原因，是因为它避免在错误面前暴露所有级别的工作，这样不但会造成语法上的错误，还会产生语义上的问题，如不当使用寄存器等。编译器的诊断机制可以帮助找出这些错误，更重要的是，它非常容易插入调试的快照。

就我而言，这些生产率和调试方面的优势是势不可挡的。我无法想像使用汇编语言能方便地开发出系统软件。

那么，上述工具的传统反对意见有哪些呢？这里有三点：(1)它无法完成我想做的事情；(2)目标代码过于庞大；(3)目标代码运行速度过慢。

就功能而言，我相信反对不再存在。所有证据都显示了人们可以完成想做的事情，只是需要花费时间和精力找出如何做而已，这可能需要一些讨人嫌的技巧。[3, 4]

就空间而言，新的优化编译器已非常令人满意，并且将持续地改进。

就速度而言，经优化编译器生成的代码，比绝大多数程序员手写代码的效率要高。而且，在前者被全面测试之后，可以将其中的 1%～5%替换成手写的代码，这往往能解决速度方面的问题。[5]

系统编程需要什么样的高级语言呢？现在唯一可供合理选择的语言是 PL/I。[6]它提供完整的功能集；它与操作系统环境相吻合；它有各种各样的编译器，一些是交互式的，一些速度很快，一些诊断性很好，另

一些能产生优化程度很高的代码。我自己觉得使用 APL 来解决算法更快一些，然后，将它们翻译成与系统环境相吻合的 PL/I 语言。

交互式编程。MIT 的 Multics 项目的成果之一是它对软件编程系统开发的贡献。在那些系统编程所关注的方面，Multics(以及后续系统，IBM 的 TSS)和其他交互式计算机系统在概念上有很大的不同：多个级别上数据和程序的共享和保护，可延伸的库管理，以及用于终端用户之间协作的设施。我确信在许多应用上，批处理系统绝对不会被交互式系统所取代。但是，我认为 Multics 小组是系统编程应用上最具有说服力的成功案例。

然而，目前还没有非常明显的证据来证明这些功能强大的工具的效力。正如人们所普遍认识的那样，调试是系统编程中很慢和较困难的部分，而漫长的调试周转时间是调试的祸根。就这一点而言，交互式编程的逻辑合理性是勿庸置疑的。[7]

另外，从很多采用这种方式开发了小型系统和系统某个部分的人那里，我们听到了很多好的证据。我唯一见到的关于大型编程系统开发方面的数字，是出自贝尔实验室 John Harr 的论文。这些数字如图 12-2 所示，它们分别反映了代码编写、汇编和程序调试的情况。第一个程序主要是控制程序；其他三个则是语言翻译、编辑等程序。Harr 的数据表明，在系统软件开发中，交互式编程的生产率至少是原来的 2 倍。[8]

程序	规模	批处理(B)或交互式(C)	指令/人年
ESS 代码	800 000	B	500～1 000
7094 ESS 支持	120 000	B	2 100～3 400
360 ESS 支持	32 000	C	8 000
360 ESS 支持	8 300	B	4 000

图 12-2　批处理和交互式编程生产率的对比

由于电传打字机和打印机终端无法用于内存转储的调试，大多数交互式工具的有效使用需要采用高级语言来开发。有了高级语言，就能很容易地修改代码和选择性地打印结果。实际上，它们组成了一对强大的工具。

CHAPTER 13

THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 13 章

整 体 部 分

The Whole and the Parts

沃特·迪斯尼公司的米老鼠

注：©The Walt Disney Company

我可以召唤地下的幽魂。

这我也会，什么人都会，可是当您召唤它们的时候，它们会应召而来吗？

——莎士比亚，《亨利四世》，第一部分

I can call spirits from the vasty deep.

Why so can I, or so can any man; but will they come when you do call for them?

SHAKESPEARE, *KING HENRY IV*, PART I

和古老的神话一样，现代神话里也总有一些爱吹嘘的人："我可以编写控制航空运输、拦截导弹、管理银行账户、控制生产线的系统。"对这些人，回答很简单，"我也可以，任何人都可以，但是当你真的写了，你就会成功吗？"

如何开发一个可以运行的系统？如何测试系统？如何将经过测试的一系列构件集成到已测试过、可以依赖的系统？对这些问题，我们以前或多或少地提到了一些解决方法，现在就来更加系统地考虑一下。

剔除 bug 的设计

防范 bug 的定义。系统各个组成部分的开发者都会做出一些假设，而这些假设之间的不匹配是大多数致命和难以察觉的 bug 的主要来源。第 4、5、6 章所讨论的获取概念完整性的途径，就是直接面对这些问题的。简言之，产品的概念完整性在使它易于使用的同时，也使开发更容易进行，而且 bug 更不容易产生。

上述方法意味着详尽、艰苦的体系结构设计正是出于这种目的。贝尔实验室安全监控系统项目的 V. A. Vyssotsky 提出："关键的工作是产品定义。许许多多的失败完全是因为那些产品未精确定义的地方而导致的。"[1]细致的功能定义、仔细的规格说明、规范化的功能描述说明以及这些方法的实施，大大减少了系统中必须查找的 bug 数量。

测试规格说明。在编写任何代码之前，规格说明必须提交给外部测试小组，以详细地检查说明的完整性和明确性。如同 Vyssotsky 所说的，开发人员自己无法完成这项工作："他们不会告诉你他们不懂。相反，他们乐于自己摸索出解决问题和澄清疑惑的办法。"

自上而下的设计。在1971年的一篇论文中，Niklaus Wirth把一种被很多最优秀的编程人员多年使用的设计流程形式化。[2]尽管他的理念是为了程序设计，也完全适用于复杂系统的软件开发设计。他将系统开发划分为体系结构设计、设计实现和物理编码实现，每个步骤都可以使用自上而下的方法很好地实现。

简言之，Wirth的流程将设计看成一系列**精化步骤**。最初，他通过比较粗略的任务定义和大概的解决方案得到主要结果。然后，对该定义和方案进行细致的检查，以判断结果与期望之间的差距。同时，将上述步骤的解决方案在更细的步骤中进行分解，每一项任务定义的精化变成了解决方案中算法的精化，还可能伴随着数据表达方式的精化。

在这个过程中，当识别出解决方案或者数据的**模块**时，对这些模块的进一步细化可以独立于其他的工作，而模块的大小决定了程序的适用性和可变化的程度。

Wirth主张在每个步骤中，尽可能地使用级别较高的表达方法来表现概念和隐藏细节，直到有必要进行进一步的细化。

好的自上而下的设计从几个方面避免了bug。首先，清晰的结构和表达方式更容易对需求和模块功能进行精确地描述。其次，模块分割和模块独立性避免了系统级的bug。第三，细节的抑制使结构上的缺陷更加容易识别。第四，设计在每个精化步骤上都是可以测试的，所以测试可以尽早开始，并且每个步骤的重点可以放在合适的级别上。

当遇到一些意想不到的问题时，按部就班的流程并不意味着步骤不能逆转。实际上，这种情况经常发生。至少，它让我们更加清楚在什么时候和为什么抛弃了整个设计并重新开始。一些糟糕的系统往往就是试图挽救一个基础很差的设计，而对它添加了各种表面装饰般的补丁。自

上而下的方法减少了这样的企图。

我确信在十年内，自上而下的设计将会是最重要的新型形式化软件开发方法。

结构化编程。另外一系列减少 bug 数量的新方法很大程度上来自 Dijkstra。[3]Böhm 和 Jacopini 为其提供了理论证明。[4]

该方法所设计程序的控制结构基本上仅包含语句形式的循环结构，例如 DO WHILE，以及 IF...THEN...ELSE 的条件判断结构，而具体的条件部分在 IF...THEN...ELSE 后的花括号中描述。Böhm 和 Jacopini 展示了这些结构在理论上是可以证明的。而 Dijkstra 认为另外一种方法，即通过 GO TO 不加限制的分支跳转，会产生导致自身逻辑错误的结构。

虽然这种方法的基本理念非常优秀，但仍有人提出了一些反面的意见。一些附加的控制结构非常实用，例如，在多个条件下的多路分支(CASE 语句)，异常跳转等(Go To Abnormal End)。此外，关于完全避免 GO TO 语句的说法显得有些教条主义，而且似乎有些吹毛求疵。

关键的地方和构建无 bug 程序的核心，是把系统的结构作为控制结构来考虑，而不是独立的分支语句。这种思考方法是我们在程序设计发展史上向前迈出的一大步。

构件单元调试

程序调试过程在过去的 20 年中经过了一个大循环，甚至在某些方面，它们又回到了起点。整个循环有四步，跟随这个过程并检验每个步骤各自的动机是一件很有趣的事情。

本机调试。早期机器的输入和输出设备很差，延迟也很长。典型的情况是，机器采用纸带或者磁带的方式来读写，采用离线设备来完成磁带的准备和打印工作。这使得调试无法忍受磁带的输入/输出。因此，在一次机器交互会话中会尽可能多地包含试验性操作。

在这种情况下，程序员仔细地设计他的调试过程—— 计划停止的地点，检验内存的位置，需要检查的东西以及如果没有预期结果时的对策。花费在编写过度繁琐的调试程序上的时间，可能是被调试程序编制时间的一半。

这个步骤的“重大罪过”是在没有把程序划分成测试段和对执行终止位置进行计划的前提下，就粗暴地按了“开始”(START)键。

内存转储。本机调试非常有效。在两小时的交互过程中可能会发现一打问题，由于计算机的资源非常匮乏，因此成本很高。计算机时间的浪费实在是一件可怕的事情。

因此，当使用在线高速打印机时，测试技术发生了变化。某人持续地运行程序，直到某个检测失败，这时所有的内存都被转储。接着，他将开始艰苦的桌面工作，考虑每个内存位置的内容。桌面工作的时间和本机调试并没有太大的不同，但它的方式与以前相比更为含混，不易理解，并且发生在测试执行之后。由于测试依赖于批处理的周期，特定用户调试用的时间更长。总之，整个过程的设计是为了减少计算机的使用时间，尽可能满足更多的编程人员。

快照。采用内存转储技术的机器往往配有 2 000～4 000 个字(word 双字节)，或者 8K～16K 字节的内存。但是，随着内存规模的不断增长，对整个内存都进行转储变得不大可能。因此，人们开发了有选择的转储、选择性跟踪和将快照插入程序的技术。OS/360 TESTRAN 允许将快照插

入程序，无需重新汇编和编译，它是快照技术方向的终极产品。

交互式调试。1959 年，Codd 和他的同事[5]以及 Strachey[6]分别发表了关于协助分时调试工作的论文，提出了一种兼有本机调试方式实时性和批处理调试高效使用率的方法。计算机将多个程序载入到内存中准备运行，被调试的程序和一个只能由程序控制的终端相关联，由监督调度程序控制调试过程。当终端前的编程人员停止程序以检查进展情况或者进行修改时，监督程序可以运行其他程序，从而保证了机器的使用率。

Codd 的多道程序系统已经开发出来了，但是它的重点是通过有效地利用输入/输出来提高吞吐量，并没有实现交互式的调试。Strachey 的想法不断得到改进，终于在 1963 年由 MIT 的 Corbató和他的同事在 7090 的实验性系统上实现了。[7]这个开发结果导致了如今的 MULTICS、TSS 和其他分时系统的出现。

在最初使用的本机调试方法和现在的交互式调试方法之间，用户可以感觉到的主要差异是由于调度监控程序和相关语言解释编译器的出现而带来的便利。而现在，已经可以用高级语言来编程和调试了，高效的编辑工具使修改和快照更为容易。

交互式调试拥有和本机调试一样的操作实时性，但前者并没有像后者要求的那样，在调试过程中要预先进行计划。在某种程度上，像本机调试那样的预先计划显得并不是很有必要，因为在调试人员停顿和思考时，计算机的时间并没有被浪费。

不过，Gold 实验得到一个有趣的结果。这个结果显示，在每次调试会话中，第一次交互取得的工作进展是后续交互的 3 倍。[8]这强烈地暗示着，由于缺乏对调试会话的计划，我们没有充分利用交互式调试的潜力，原有本机调试技术中那段高效率的时间消失了。

我发现对良好终端系统的正确使用，往往要求每两小时的终端会话对应于两小时的桌面工作。这其中一半时间用于上次会话的清理工作：更新调试日志，把更新后的程序列表加入到项目文件夹中，解释调试中出现的奇怪现象。剩余的一半时间用于准备：为下一次操作设计详细的测试，进行计划的变更和改进。如果没有这样的计划，则很难保持两个小时的高生产率；而没有事后的清理工作，则很难保证后续终端会话的系统化和持续推进。

测试用例。关于实际调试过程和测试用例的设计，Gruenberger 提出了特别好的对策，[9]在其他文章中，也有更为简便的方法。[10,11]

系统集成调试

软件系统开发过程中出乎意料的困难部分是系统集成测试。前面我已经讨论了一些困难产生和困难不确定的原因。其中需要再次确认的两件事是：系统调试花费的时间会比预料的更长，它的困难证明了需要一种完备系统化和可计划的方法。下面来看看这样的方法所包括的内容。[12]

使用经过调试的构件单元。尽管其并不是普遍的实际情况——不过通常的看法是——系统集成调试要求只能在每个部分都能正常运行之后开始。

实际工作中，存在着与上面看法不同的两种情况。一种是“合在一起尝试”的方法，这种方法似乎是基于这样的观点：除了构件单元上的 bug 之外，还存在系统 bug(如接口)，将各个部分合拢得越早，系统 bug 就出现得越早。另一种观念则没有这么复杂：使用系统的各个部分进行相互测试，避免了大量测试辅助平台的搭建工作。这两种情况显然都是

合理的，但经验显示，它们并不完全正确——在系统测试中使用完好的、经过调试的构件，能比搭建测试平台和进行全面的构件单元测试节省更多的时间。

更精妙的一种方法是“文档化的 bug”。它声明当构件单元所有的缺陷已经被**发现**，但在还没有完全被**修复**时，就已经做好了系统调试的准备。在系统测试期间，根据该理论，测试人员知道这些缺陷造成的后果，从而可以忽略它们，将注意力集中在新出现的问题上。

但是所有这些只是良好的愿望，只是试图为结果的偏离寻找一些合理理由。实际上，调试人员**并不**了解 bug 引起的所有后果；不过，如果系统比较简单，系统测试倒不会太困难。另外，对文档记录 bug 的修复工作本身会注入未知的问题，接下来的系统测试会令人困惑。

搭建充分的测试平台。这里所说的辅助测试平台，指的是供调试使用的所有程序和数据，它们不会整合到最终产品中。测试平台可能会有相当于测试对象一半的代码量，但这是合乎情理的。

一种测试辅助的形式是**伪构件**(dummy component)，它仅仅由接口和可能的伪数据或者一些小的测试用例组成。例如，系统包含某种排序程序，但该程序还未完成，这时其他部分的测试可以通过伪构件来实现，该构件只是读入输入数据，对数据格式进行校验，输出格式良好、但没有实际意义的有序数据以供使用。

另一种形式是**微缩文件**(miniature file)。很常见的一类 bug 来自对磁带和磁盘文件格式的错误理解。所以，创建一个仅包含典型记录，但涵盖全部描述的小型文件是非常值得的。

微缩文件的特例是**伪文件**(dummy file)，它实际上并不常见。不过 OS/360 的任务控制语言提供了这种功能，对于构件单元调试非常有用。

还有一种方式是**辅助程序**(auxiliary program)。用来测试数据的发生器、特殊的打印输出和交叉引用表分析等，这些都是需要另外开发的专用辅助工具的例子。[13]

控制变更。在测试期间进行严密控制是硬件调试中一项令人印象深刻的技术，它同样适用于软件系统。

首先，必须有人负责。他必须控制和负责各个构件单元的变更或者版本之间的替换。

接着，就像前面所讨论的，必须存在系统的受控拷贝：一个是供构件单元测试使用的最终锁定版本；一个是测试版本的拷贝，用来进行缺陷的修复；以及一个开发库，其他人员可以在该拷贝上进行各自的程序开发工作，例如修复和扩展自己的模块和子系统等。

在 System/360 工程模型中，在一大堆常规的黄颜色电线中，常常可以不经意地看到紫色的电线束。在发现 bug 以后，我们会做两件事情：设计快速修复电路，并安装到系统中，从而不会妨碍测试的继续进行。这些更改过的接线使用紫色电线，看上去就像一个受了伤的大拇指。我们需要把更改记录到日志中，同时，还要准备一份正式的变更文档，并启动设计自动化流程。最后，在电路图或者黄色线路中会实现该设计的调整——更新相应的电路图和接线表以及开发一个新的电路板。现在，物理模型和电路图重新吻合了，紫色的线束也就不再需要了。

软件开发也需要用到“紫色线束”的手法。对于最后成为产品的程序代码，它更迫切地需要进行严密控制和深层次的关注。上述技巧的关键因素是对变更和差异的记载，即在一个日志中记录所有的变更，而在

源代码中显著标记快速补丁和正式修改之间的区别，正式修改是完备并经过测试的，而且需要文档化。

一次添加一个构件。这样做的好处同样是显而易见的，但是乐观主义和惰性常常诱使我们破坏这个规则。因为离散构件的添加需要调试伪程序和其他测试平台，有很多工作要做，毕竟，可能我们不需要这些额外工作？可能不会出现什么 bug?

不！拒绝诱惑！这正是系统化系统测试所关注的地方。我们必须假设系统中存在许多错误，并需要计划一个有序的过程把它们找出来。

注意，必须拥有完整的测试用例，在添加了每一个新构件之后，都要用它们来测试子系统。因为那些原来可以在子系统上成功运行的用例必须在现有系统上重新运行，对系统进行回归测试。

阶段(量子)化、定期变更。随着项目的推进，系统构件的开发者会带着他们工作的最新版本——更快、更小、更完整，或者公认的、bug更少的版本不时地出现在我们面前。将使用中的构件替换成新版本仍然需要进行和构件添加一样的系统化测试流程。这个时候通常已经具备了更完整、有效的测试用例，因此测试时间往往会减少很多。

项目中，其他开发团队会使用经过测试的最新集成系统作为调试自己程序的平台。测试平台的修改会阻碍他们的工作。当然，这是必须的。但是，变更必须被阶段化，并且定期发布。这样，每个用户拥有稳定的生产周期，其中穿插着测试平台的改变。这种方法比持续波动所造成的混乱无序要好一些。

Lehman 和 Belady 提供了证据，阶段(量子)要么很大，间隔很宽；要么小而频繁。[14]根据他们的模型，小而频繁的阶段很容易变得不稳定，我的经验同样证实了这一点——我决不会在实践中冒险采用后一种

策略。

阶段(量子)化变更方法非常优美地容纳了紫色线束技术：直到下一次系统构件的定期发布之前都一直使用快速补丁；而在当前的发布中，其把已经通过测试并进行了文档化的修补措施整合到系统平台中。

CHAPTER 14

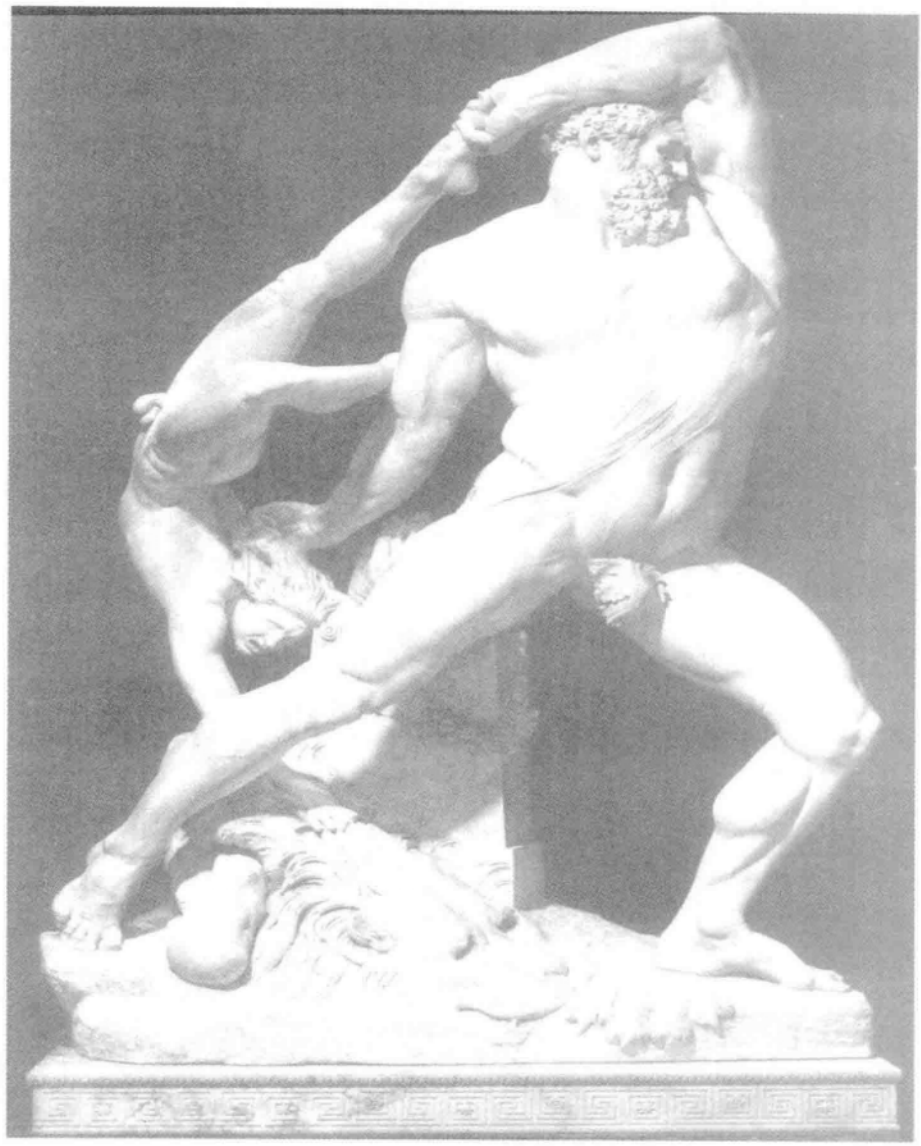
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 14 章

祸起萧墙

Hatching a Catastrophe



A. 卡诺瓦 1802 年的雕塑作品《赫拉克勒斯与利喀斯》(*Ercole e Lica*)

注：雕塑讲述的是希腊神话中大力神赫拉克勒斯把不知情带来抹上毒血的衣服的侍从利喀斯扔进了大海。

资料来源：Scala/Art Resource, NY

带来坏消息的人不受欢迎。

——索福克勒斯

项目怎么会被延迟了整整一年的时间
……延迟的时间是一天天积累下来的。

None love the bearer of bad news.

SOPHOCLES

How does a project get to be a year late?
...One day at a time.

当人们听到某个项目的进度发生了灾难性偏离时，可能会认为项目一定遭受了一系列重大灾难。然而，通常灾祸来自白蚁的肆虐，而不是龙卷风的侵袭。同样，项目进度经常以一种难以察觉，但是残酷无情的方式慢慢落后。实际上，重大灾害是比较容易处理的，它往往和重大的压力、彻底的重组、新技术的出现有关，整个项目组通常可以应付自如。

但是一天一天的进度落后是难以识别、不容易防范和难以弥补的。昨天，某个关键人员生病了，无法召开某个会议。今天，由于雷击打坏了大厦的供电变压器，所有机器无法启动。明天，因为工厂磁盘供货延迟了一周，磁盘例行的测试无法进行。下雪、应急任务、私人问题、同顾客的紧急会议、管理人员检查——这个列表可以不断地延长。每件事情都只会将某项活动延迟半天或者一天，但是整个进度开始落后了，尽管每次只有一点点。

里程碑还是沉重的负担

如何根据一个严格的进度表来控制大型项目？第一步是**制定**进度表。进度表上的每一件事被称为“里程碑”，它们都有一个日期。选择日期是一个估计技术上的问题，在前面已经讨论过，它在很大程度上依赖以往的经验。

里程碑的选择只有一个原则，那就是里程碑必须是具体的、特定的、可度量的事件，能够进行清晰定义。以下是一些反面的例子，例如编码，在代码编写时间刚到一半的时候就已经“90%完成”了；调试在大多时候都是“99%完成”的；“计划完毕”是任何人只要愿意，就可以声明的事件。[1]

然而，具体的里程碑是百分之百的事件。“结构师和实现人员签字认可的规格说明”，“100%源代码编制完成，纸带打孔完成并输入到磁盘库”，“测试版通过了所有的测试用例”。这些切实的里程碑澄清了那些划分得比较模糊的阶段——计划、编码和调试。

里程碑边界明显和没有歧义，比它容易被老板核实更为重要。**如果**里程碑定义得非常明确，无法自欺欺人时，很少有人会就里程碑的进展弄虚作假。但是如果里程碑很模糊，老板就常常会得到一份与实际情况不符的报告。毕竟，没有人愿意承受坏消息。这种做法只是为了起到缓和的作用，并没有任何蓄意的欺骗。

对于大型开发项目中的估计行为，政府的承包商做了两项有趣的研究，以下是研究结果。

(1) 如果在某项活动开始之前就着手估计，并且每两周进行一次仔细的修订。这样，随着开始时间的临近，无论最后情况会变得如何糟糕，它都不会有太大的变化。

(2) 活动**期间**，对时间长短的**过高**估计会随着活动的进行持续而下降。

(3) **过低估计**在活动中不会有太大变化，一直到计划的结束日期之前大约三周左右。[2]

好的里程碑对团队来说实际上是一项服务，可以用来向项目经理提出合理要求的一项服务，而模糊的里程碑是难以处理的负担。当里程碑没有正确反映损失的时间，并对人们形成误导，以致事态无法挽回的时候，它会彻底打击小组的士气。慢性进度偏离同样也是士气杀手。

“其他的部分反正会落后”

进度落后了一天，那又怎么样呢？谁会关心一天的滞后？我们可以跟上进度。何况，和我们有关的其他部分已经落后了。

棒球队队长知道，**进取**是很多优秀队员和团队不可缺少的心理素质。它表现为“比要求的跑得更快”，“比要求的移动得更加迅速”，“更加努力尝试”。对软件开发队伍来说，**进取**同样是非常必要的。**进取**提供了缓冲和储备，使开发队伍能够处理常规的灾祸，可以预计和防止小的灾祸。而对任务进行计算和对工作量进行度量，会对进取超前造成一些消极的影响。如同我们看到的，**必须**关心每一天的滞后，它们是大灾祸的基本组成元素。

并不是每一天的滞后都等于灾难。尽管会如上文所述，事先估计会给工作进度的超前带来影响，但对活动的一些计算和考虑还是必要的。那么，如何判断哪些偏离是关键的呢？只有采用 PERT 或者关键路径技术才能判断。它能显示谁需要什么样的东西，谁位于关键路径上，在哪里发生滞后会影响最终的完成日期。另外，它还指出一个任务在成为关键路径以前，可以落后的时间。

严格地说，PERT 技术是关键路径计划的细化。如果使用 PERT 图，它需要对每个事件估计 3 次，每次对应于满足估计日期的不同可能性。我觉得不值得为这样的精化发生额外的工作量，但为了方便，我把任何关键路径法都称为 PERT 图。

PERT 的准备工作是 PERT 图使用中最有价值的部分。它包括整个网状结构的展开、任务之间依赖关系的识别和各个任务链的估计，这些都要求在项目早期进行非常专业的计划。第一份 PERT 图总是很恐

怖的，不过人们总是不断地进行努力，运用才智制定下一份 PERT 图。

随着项目的推进，PERT 图为前面那个泄气的借口，“其他的部分反正会落后”提供了答案。它展示某人为了使自己的工作远离关键路径，需要超前多少，也建议了补偿其他部分失去的时间的方法。

地毯的下面

当一线经理发现自己的队伍出现了计划偏离时，他肯定不会马上赶到老板那里去汇报这个令人沮丧的消息。团队可以弥补进度偏差，他应该可以想出应对方法或者重新安排进度以解决问题，为什么要去麻烦老板呢？从这个角度来看，好像还不错。解决这类问题的确是一线经理的职责。老板已经有很多需要处理的真正的烦心事了，他不想被更多的问题打搅。因此，所有的污垢都被隐藏在地毯之下。

但是每个老板都需要两种信息：需要采取行动计划方面的问题，用来进行分析的状态数据。[3]出于这个目的，他需要了解所有开发队伍的情况，但得到真相是很困难的。

一线经理的利益和老板的利益在这里是存在内在冲突的。一线经理担心如果汇报了问题，老板会采取行动，这些行动会取代经理的作用，降低自己的威信，搞乱其他计划。所以，只要项目经理认为自己可以独立解决问题，他就不会告诉老板。

有两种掀开毯子把污垢展现在老板面前的方法，它们必须都被采用。一种是减少角色冲突和鼓励状态共享，另一种是猛地拉开地毯。

减少角色的冲突。首先老板必须区别行动信息和状态信息。他必须规范自己，不对项目经理可以解决的问题做出反应，并且决不在检查状态报告的时候做安排。我曾经认识一个老板，他总是在状态报告的第一个段落结束之前，拿起电话发号施令。这样的做法肯定会压制信息的完全公开。

不过，当项目经理了解到老板收到状态报告之后不会惊慌，或者不会越俎代庖时，他就逐渐会提交真实的结果。

如果老板把会见、评审、会议明显标记为**状态检查**(status-review)和**“问题—行动”**(problem-action)会议，并且相应控制自己的行为，这对整个过程会很有帮助。当然，事态发展到无法控制时，状态检查会议会演变成“问题－行动”会议。不过，至少每个人知道“当时游戏的分数是多少”，老板在接过“皮球”之前也会三思。

猛地拉开地毯。不论协作与否，拥有能了解状态真相的评审机制是必要的。PERT 图以及频繁、明确的里程碑是这种评审的基础。大型项目中，可能需要每周对某些部分进行评审，大约一个月左右进行整体评审。

有报告显示，关键的文档是里程碑和实际的完成情况。图 14-1 是上述报告中的一段摘录。它显示了一些问题：某些部分的规格核准延迟了。手册(SLR)的批准时间有所延迟，其中一个的时间比独立产品测试(Alpha)的开始时间还要迟。这样一份报告将作为 2 月 1 日的会议议程，使得每个人都知道问题的所在，而产品构件经理应该准备解释延迟的原因，什么时候结束，采取的步骤和需要的任何帮助——老板提供的，或者是其他小组间接提供的。

SYSTEM/360 SUMMARY STATUS REPORT
OS/360 LANGUAGE PROCESSORS + SERVICE PROGRAMS
AS OF FEBRUARY 01,1965

A=APPROVAL
C=COMPLETED

*=REVISED PLANNED DATE
NE=NOT ESTABLISHED

PROJECT	LOCATION	COMMITMNT ANNOUNCE RELEASE	OBJECTIVE AVAILABLE APPROVED	SPECS AVAILABLE APPROVED	SRL AVAILABLE APPROVED	ALPHA TEST ENTRY EXIT	COMP TEST START COMPLETE	SYS TEST START COMPLETE	BULLETIN AVAILABLE APPROVED	BETA TEST ENTRY EXIT
OPERATING SYSTEM										
12K DESIGN LEVEL (E)										
ASSEMBLY	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	10/13/4 C 01/11/5	11/13/4 C 11/18/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
FORTRAN	POK	04/--/4 C 12/31/5	10/28/4 C	10/21/4 C 01/22/5	12/17/4 C 12/19/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
COBOL	ENDICOTT	04/--/4 C 12/31/5	10/28/4 C	10/15/4 C 01/20/5 A	11/17/4 C 12/08/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
RPG	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	09/30/4 C 01/05/5 A	12/02/4 C 01/18/5 A	01/15/5 C 02/22/5				09/01/5 11/30/5
UTILITIES	TIME/LIFE	04/--/4 C 12/31/5	06/24/4 C		11/20/4 C 11/30/4 A					09/01/5 11/30/5
SORT 1	POK	04/--/4 C 12/31/5	10/28/4 C	10/19/4 C 01/11/5	11/12/4 C 11/30/4 A	01/15/5 C 03/22/5				09/01/5 11/30/5
SORT 2	POK	04/--/4 C 06/30/6	10/28/4 C	10/19/4 C 01/11/5	11/12/4 C 11/30/4 A	01/15/5 C 03/22/5				03/01/6 05/30/6
44K DESIGN LEVEL (F)										
ASSEMBLY	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	10/13/4 C 01/11/5	11/13/4 C 11/18/4 A	02/15/5 03/22/5				09/01/5 11/30/5
COBOL	TIME/LIFE	04/--/4 C 06/30/6	10/28/4 C	10/15/4 C 01/20/5 A	11/17/4 C 12/08/4 A	02/15/5 03/22/5				03/01/6 05/30/6
NPL	HURSLEY	04/--/4 C 03/31/6	10/28/4 C							
2250	KINGSTON	03/30/4 C 03/31/6	11/05/4 C	12/08/4 C 01/04/5	01/12/5 C 01/29/5	01/04/5 C 01/29/5				01/03/6 NE
2280	KINGSTON	06/30/4 C 09/30/6	11/05/4 C			04/01/5 04/30/5				01/28/6 NE
200K DESIGN LEVEL (H)										
ASSEMBLY	TIME/LIFE		10/28/4 C							
FORTRAN	POK	04/--/4 C 06/30/6	10/28/4 C	10/15/4 C 01/11/5	11/11/4 C 12/10/4 A	02/15/5 03/22/5				03/01/6 05/30/6
NPL	HURSLEY	04/--/4 C 03/31/7	10/28/4 C			07/--/5				01/--/7
NPL H	POK	04/--/4 C	03/30/4 C			02/01/5 04/01/5				10/15/5 12/15/5

图 14-1　报告摘录

贝尔实验室的 V. Vyssotsky 添加了以下的观察意见：

我发现在里程碑报告中很容易记录“计划”和“估计”的日期。计划日期是项目经理的工作产物，代表了经协调后的项目整体工作计划，它是合理计划**之前**的判断。估计日期是最基层经理的工作产物，基层经理对所讨论的工作有着深刻的了解，估计日期代表了在现有资源和已得到了作为先决条件的必要输入(或得到了相应的承诺)的情况下，基层经理对实际实现日期的最佳判断。项目经理必须停止对这些日期的怀疑，将重点放在使其更加精确上，以便得到没有偏见的估计，而不是那些合乎心意的乐观估计或者自我保护的保守估计。一旦它们在每个人的脑海中形成了清晰的印象，项目经理就可以预见到将来如果他在哪些地方不采取任何措施，就会出现问题。[4]

PERT 图的准备工作是老板和要向他进行汇报的经理们的职责。需要一个小组(1～3 人)来关注它的更新、修订和报告，可以将这个小组看做是老板的延伸。对大型项目，这种**计划和控制**(Plan and Control)小组的价值是非常可贵的。小组的职权仅限于向产品线经理询问他们什么时候设定或更改里程碑，以及是否达到了里程碑。计划和控制小组处理所有的文字工作，因此产品线经理的负担将会减到最少——仅仅需要做出决策。

我们拥有一个富有热情的、老练的、熟练的计划和控制小组。这个小组由 A. M. Pietrasanta 负责，他在设计有效的、谦逊的控制方法方面有丰富的创造天赋。结果，我发现他的小组被广为尊重，而不仅仅是被容忍。对于这样一个本来就十分敏感的角色，这的确是一个成功。

对计划和控制职能进行适度的技术人力投资是非常值得赞赏的。它在项目的贡献方式和直接开发软件产品方面差异较大。计划和控制小组作为监督人员，明白地指出了不易察觉的延迟，并强调关键的因素。他们是早期的预警系统，防止项目以一次一天的方式落后一年。

CHAPTER 15

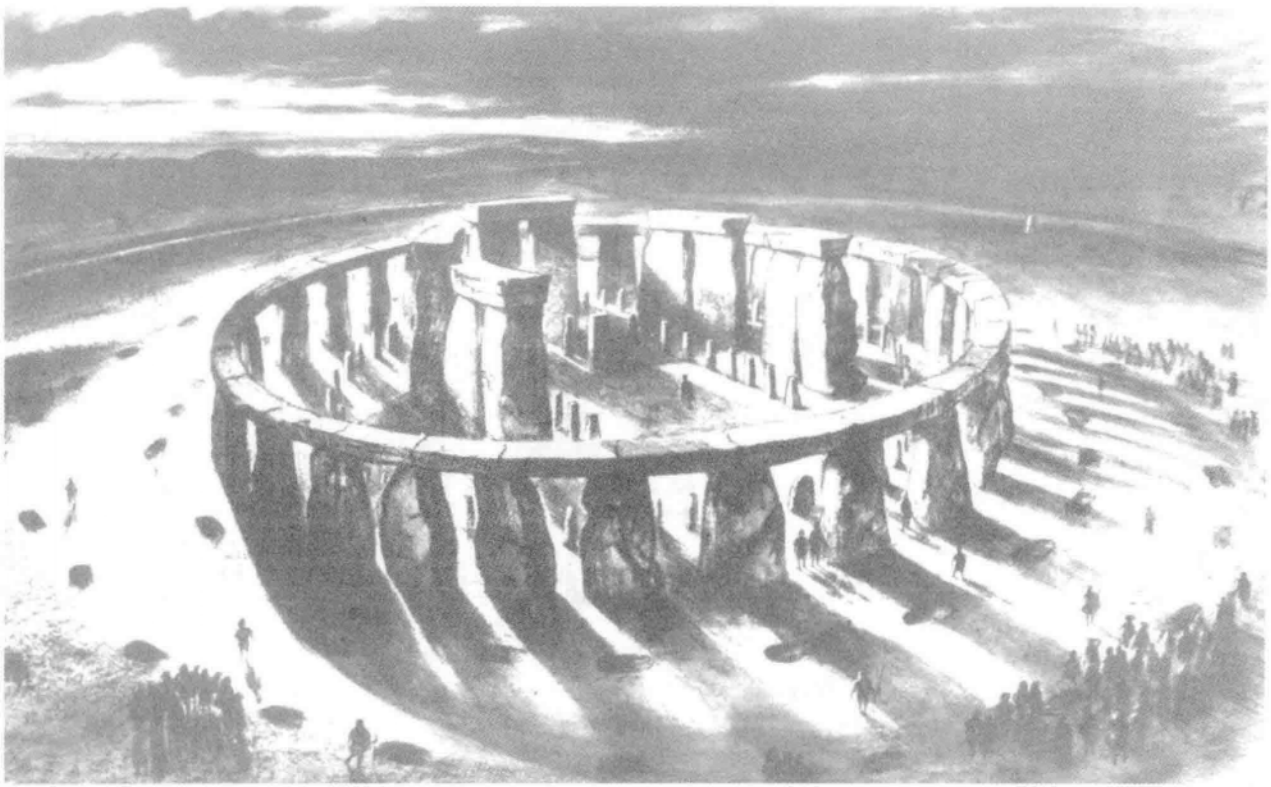
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 15 章

另 外 一 面

The Other Face



世界上最大的计算机——史前巨石阵的复原景象

资料来源：The Bettman Archive

不了解，就无法真正拥有。

——歌德

噢，赐予我朴素的评论者吧，他们不会因过于深奥而让人困惑不解。

——克雷布

What we do not understand we do not possess.

GOETHE

O give me commentators plain,
Who with no deep researches vex the brain.

CRABBE

计算机程序是从人传递到机器的一些信息。为了将人的意图清晰地传达给不会说话的机器，程序采用了严格的语法和严谨的定义。

但是书面的计算机程序还有其他的方式向用户诉说自己的“故事”。即使是完全开发给自己使用的程序，这种沟通仍然是必要的。因为记忆衰退的规律会使用户—作者失去对程序的了解，于是他不得不重拾自己劳动的各个细节。

公共应用程序的用户在时间和空间上都远离它们的作者，因此对这类程序，文档的重要性更是不言而喻。对软件编程产品来说，程序向用户所呈现的和提供给机器识别的内容同样重要。

面对那些文档“简约”的程序，我们中的大多数人都不免曾经暗骂那些远在他方的匿名作者。因此，一些人试图向新人慢慢地灌输文档的重要性，旨在延长软件的生命期，克服惰性和进度的压力。但是，很多次尝试都失败了，我想很可能是由于我们使用了错误的方法。

Thomas J. Watson 讲述了他年轻时在纽约州的北部刚开始做收银机推销员的经历。他带着一马车的收银机，满怀热情地动身了。他工作得非常勤奋，但是没有卖出去一台收银机。他很沮丧地向经理汇报了情况，销售经理听了一会儿，说道：“帮我抬一些机器到马车上，收紧缰绳，出发！”他们成功了。在接下来的客户拜访过程中，经理身体力行，向他**演示了如何**出售收银机。事实证明，这个方法是可行的。

多年来，我曾经非常勤奋地给我的软件工程师们举办了关于文档必要性以及优秀文档所应具备特点方面的讲座，向他们讲述——甚至是热诚地向他们游说以上的观点。不过，这些都行不通。我想他们知道如何正确地编写文档，却缺乏工作的热情。后来，我尝试了向马车上搬一些收银机，以此演示如何完成这项工作。结果发现，这种方法的效果要好得多。所以，文章剩余部分将对那些说教之辞一笔带过，而把重点放在

“如何做”(才能产生一篇优秀的文档)上。

需要什么样的文档

不同用户需要不同级别的文档。某些用户仅仅偶尔使用程序，有些用户必须依赖程序，还有一些用户必须根据环境和目的的变动对程序进行修改。

使用程序。每个用户都需要一段对程序进行描述的文字。可是大多数文档只提供了很少的总结性内容，无法达到用户要求，就像是描绘了树木，形容了树皮和树叶，但却没有一幅森林的图案。为了得到一份有用的文字描述，就必须放慢脚步，稳妥地进行。

(1) **目的。**主要的功能是什么？开发程序的原因是什么？

(2) **环境。**程序运行在什么样的机器、硬件配置和操作系统上？

(3) **范围。**输入的有效范围是什么？允许显示的合法输出范围是什么？

(4) **实现功能和使用的算法。**精确地阐述它做了什么。

(5) **“输入—输出”格式。**必须是确切和完整的。

(6) **操作指令。**包括控制台及输出内容中正常和异常结束的行为。

(7) **选项。**用户的功能选项有哪些？如何在选项之间进行挑选？

(8) **运行时间。**在指定的配置下，解决特定规模问题所需要的时间？

(9) **精度和校验。**期望结果的精确程度？如何进行精度的检测？

一般来说，三四页纸常常就可以容纳以上所有的信息。不过往往需要特别注意的是表达的简洁和精确。由于它包含了和软件相关的基本决策，所以这份文档的绝大部分需要在程序编制之前书写。

验证程序。除了程序的使用方法，还必须附带一些程序正确运行的证明，即测试用例。

每一份发布的程序拷贝应该包括一些可以例行运行的小测试用例，为用户提供信心——他拥有了一份可信赖的拷贝，并且正确地安装到了机器上。

然后，需要得到更加全面的测试用例，在程序修改之后，进行常规运行。这些用例可以根据输入数据的范围划分成三个部分。

(1) 针对遇到的大多数常规数据对程序主要功能进行测试的用例。它们是测试用例的主要组成部分。

(2) 数量相对较少的合法数据测试用例，对输入数据范围边界进行检查，确保最大可能值、最小可能值和其他有效特殊数据可以正常工作。

(3) 数量相对较少的非法数据测试用例，在边界外检查数据范围边界，确保无效的输入能有正确的数据诊断提示。

修改程序。调整程序或者修复程序需要相当多的信息。显然，这要求了解全部的细节，并且这些细节已经记录在注释良好的列表中。与一般用户一样，修改者迫切需要一份清晰明了的概述，不过这一次是关于系统的内部结构。那么这份概述的组成部分是什么呢？

(1) 流程图或子系统的结构图，对此以下有更详细的论述。

(2) 对所用算法的完整描述，或者是类似算法的参考资料。

(3) 对所有文件规划的解释。

(4) 数据流处理的概要描述——从磁盘或者磁带中，获取数据或程序处理的序列——以及在每个处理过程完成的操作。

(5) 初始设计中，对已预见修改的讨论；特性、功能回调以及出口的位置；原作者对可能会修改的地方以及可能处理方案的一些意见。另外，对隐藏缺陷的观察也同样很有价值。

流程图

流程图是被吹捧得最过分的一种程序文档。事实上，很多程序甚至不需要流程图，很少有程序需要一页纸以上的流程图。

流程图显示了程序的流程判断结构，它仅仅是程序结构的一个方面。当流程图绘制在一张图上时，它能非常优雅地显示程序的判断流向，但当它被分成几张时，也就是说需要采用经过编号的出口和连接符来进行拼装时，整体结构的概观就被严重地破坏了。

因此，一页纸的流程图就成为表达程序结构、阶段或步骤的一种非常基本的图示。同样，它也非常容易绘制。图 15-1 展示了一个子程序流程图的图样。

当然，上述图纸既没有，也不需要遵循精心制定的 ANSI 流程图标准。所有图形元素如方框、连线、编号等，只需要能使这张详细的流程图可以理解就行了。

因此，逐一记录的详细流程图过时而且令人生厌，它只适合启蒙初学者的算法思维。当 Goldstine 和 Von Neumann[1]引入这种方法时，框图和框图中的内容作为一种高级语言，将难以理解的机器语言组合成一连串可理解的步骤。如同 Iverson 早期所认识到的[2]，在系统化的高级语言

中，分组已经完成，每一个方框相应地包含了一条语句(见图 15-2)。从而，方框本身变成了一件单调乏味的重复练习，可以去掉它们。这时，剩下的就只有箭头了。而连接相邻后续语句的箭头也是冗余的，可以擦掉它们。现在，留下的只有 GO TO 跳转。如果大家遵守良好的规则，使用块结构来消除 GO TO 语句，所有的箭头就都消失了，尽管这些箭头能在很大程度上帮助理解。大家完全可以丢掉流程图，使用文字列表来表达这些内容。

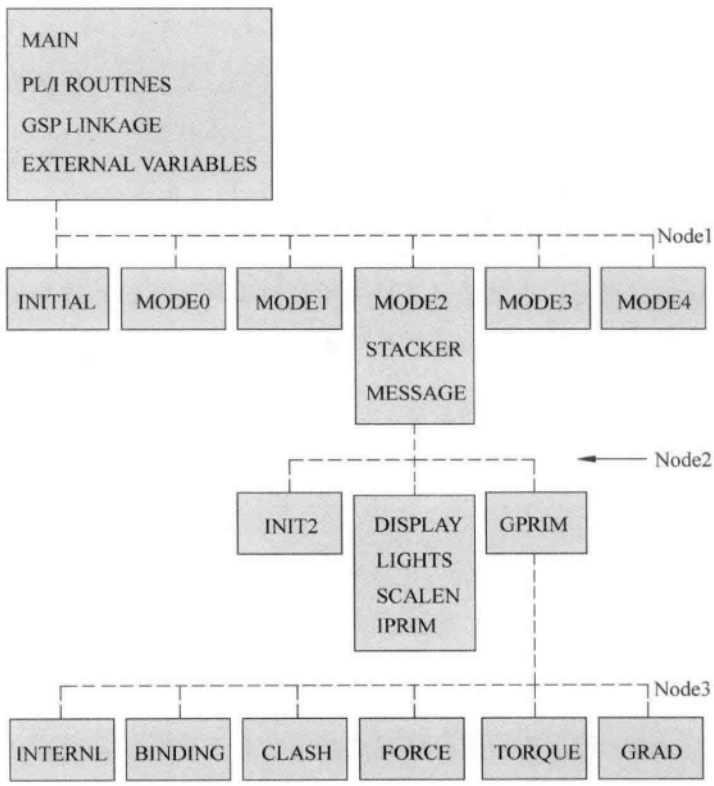
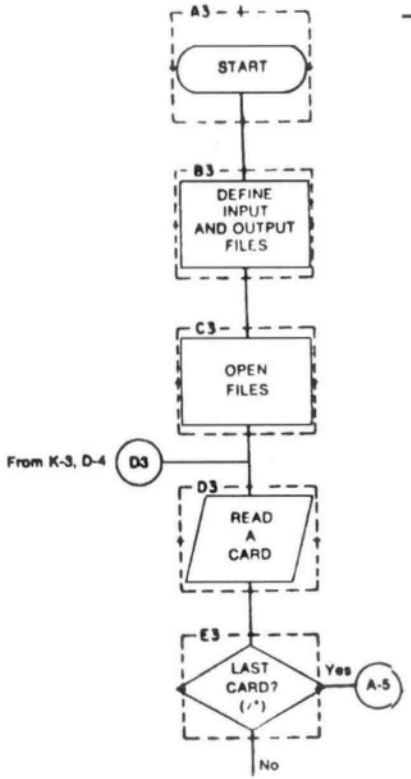


图 15-1　程序结构图(Courtesy of W.V. Wright)



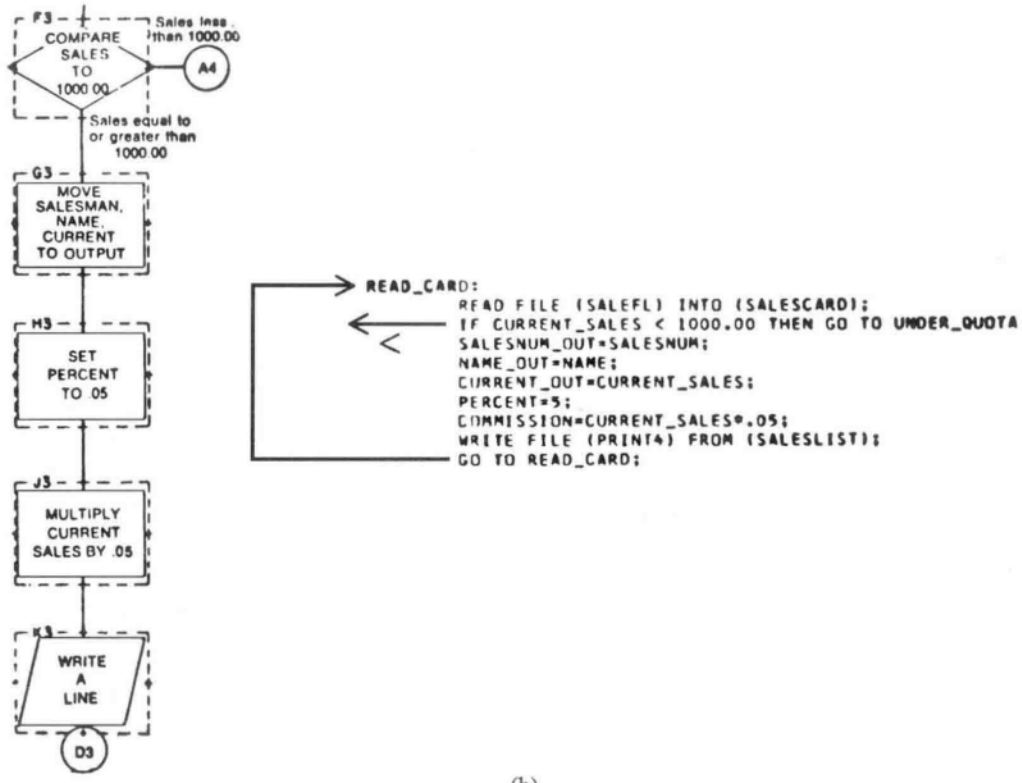

```
PGM4: PROCEDURE OPTIONS (MAIN);


          DECLARE SALEFL FILE
          RECORD
          INPUT
          ENVIRONMENT (F(80) MEDIUM (SYSIPT, 2501));
DECLARE PRINT4 FILE
          RECORD
          OUTPUT
          ENVIRONMENT (F(132) MEDIUM (SYSLST,1403) CTLASA);
DECLARE 01 SALESCARD,
          03 BLANK1          CHARACTER (9),
          03 SALESNUM        PICTURE '9999',
          03 NAME            CHARACTER (25),
          03 BLANK2          CHARACTER (7),
          03 CURRENT_SALES   PICTURE '9999V99',
          03 BLANK3          CHARACTER (29);
DECLARE 01 SALESLIST,
          03 CONTROL         CHARACTER (1) INITIAL (' '),
          03 SALESNUM_OUT    PICTURE 'ZZZ9',
          03 FILLER1         CHARACTER (5) INITIAL (' '),
          03 NAME_OUT        CHARACTER (25),
          03 FILLER2         CHARACTER (5) INITIAL (' '),
          03 CURRENT_OUT     PICTURE 'Z,ZZZV.99',
          03 FILLER3         CHARACTER (5) INITIAL (' '),
          03 PERCENT         PICTURE 'Z9',
          03 SIGN            CHARACTER (1) INITIAL ('%'),
          03 FILLER4         CHARACTER (5) INITIAL (' '),
          03 COMMISSION      PICTURE 'Z,ZZZV.99',
          03 FILLER5         CHARACTER (63) INITIAL (' ');


 OPEN FILE (SALEFL),FILE (PRINT4);

          ON ENDFILE (SALEFL) GO TO ENDOFJOB;
```

(a)



(b)

图 15-2　流程图和对应程序的对比

资料来源：Abridged and adapted from Figs. 15—41, 15—44, in *Data Processing and Computer Programming: A Modular Approach* by Thomas J. Cashman and William J. Keys (Harper & Row, 1971)

现实中，流程图被鼓吹的程度远大于它们的实际作用。我从来没有看到过一个有经验的编程人员，在开始编写程序之前，会例行公事地绘制详尽的流程图。在一些要求流程图的组织中，流程图总是事后才补上的。一些公司则很自豪地使用工具软件，从代码中生成这个“不可缺少的设计工具”。我认为这种普遍经验并不是令人尴尬和惋惜的、对良好实践的偏离(似乎大家只能对它露出窘迫的微笑)；相反，它是对技术的良好评判，向我们传授了一些流程图用途方面的知识。

耶稣门徒彼得谈到新的异教皈依者和犹太戒律时说：“为什么让他们背负我们的祖先和我们自己都不能承担的重负呢？”(《使徒行传》 15：10 现代英文版本)。对于新的编程人员和陈旧的流程图方法，我持有的观点相同。

自文档化的程序

数据处理的基本原理告诉我们，试图努力维持不同文件之间的同步关系，是一件非常费力不讨好的事情。更合理的方法是：每个数据项包含两个文件都需要的所有信息，采用指定的键值来区别，并把它们组合到一个文件中。

不过，我们在程序文档编制的实践中却违反了自己的原则。典型地，我们试图维护一份机器可读的程序以及一系列包含记叙性文字和流程图的人类可读性文档。

结果和我们自己的认识相吻合，不同文件的数据保存带来了不良的后果。程序文档质量声名狼藉，文档的维护更是低劣：程序变动总是不能及时精确地反映在文档中。

我认为相应的解决方案是“合并文件”，即把文档整合到源程序。这对正确维护是直接有力的推动，保证编程用户能方便、及时地得到文

档资料。这种程序被称为**自文档化**(Self-Documenting)。

现在看来，在程序中包括流程图显然是一种笨拙(但不是不可以)的做法。考虑到流程图方法的落后和高级语言的使用占统治地位，把程序和文档放在一起显然是很合理的。

把源程序作为文档介质，强制推行了一些约束。另一方面，对于文档读者而言，一行一行的源程序本身就可以再次利用，使新技术的使用成为可能。现在，已经到了为程序文档设计一套彻底的新方法的时候了。

文档是我们和前人都不曾成功背负的重担。作为基本目标，我们必须试图把它的负担降到最小。

方法。第一个想法是借助那些出于语言的要求而必须存在的语句，来附加尽可能多的“文档”信息。因此，标签、声明语句和符号名称均可以作为工具，用来向读者表达尽可能多的意思。

第二个方法是尽可能地使用空格和一致的格式提高程序的可读性，表现从属和嵌套关系。

第三，以段落注释的形式，向程序中插入必要的记叙性文字。大多数文档一般包括足够多的逐行注释，特别是那些满足公司呆板的“良好文档”规范的程序，通常就包含了过多的注释。即使是这些程序，在段落注释方面也常常是不够的，而段落注释能提供总体把握和真正加深读者对整件事情的理解。

因为文档是通过程序结构、命名和格式来实现的，所有这些**必须**在第一次书写代码时完成。不过，这也只是**应该**完成的时间。另外，由于自文档化的方法减少了很多附加工作，使这件工作遇到的障碍会更少。

一些技巧。图 15-3 是一段自文档化的 PL/I 程序。[3]圆圈中的数字不是程序的组成部分，而是用来帮助我们进行讨论的。

```
①//QLT4 JOB ...
②QLTSRT7: PROCEDURE (V);
   /*********************************************************************/
③  /*A SORT SUBROUTINE FOR 2500 6-BITE FIELDS, PASSED AS THE VECTOR V.  A  */
   /*SEPARATELY COMPILED, NOF-MAIN PROCEDURE, WHICH MUST USE AUTOMATIC CORE  */
   /*ALLOCATION.                                                          */
   /*                                                                     */
④  /*THE SORT ALGORITHM FOLLOWS BROOKS AND IVERSON, AUTOMATIC DATA PROCESSING,*/
   /*PROGRAM 7.23, P. 350.  THAT ALGORITHM IS REVISED AS FOLLOWS:         */
⑤  /*  STEPS 2-12 ARE SIMPLIFIED FOR M=2.                                 */
   /*  STEP 18 IS EXPANDED TO HANDLE EXPLICIT INDEXING OF THE OUTPUT VECTOR.  */
   /*  THE WHOLE FIELD IS USED AS THE SORT KEY.                           */
   /*  MINUS INFINITY IS REPRESENTED BY ZEROS.                            */
   /*  PLUS INFINITY IS REPRESENTED BY ONES.                              */
   /*  THE STATEMENT NUMBERS IN PROG. 7.23 ARE REFLECTED IN THE STATEMENT     */
   /*    LABELS OF THIS PROGRAM.                                          */
   /*  AN IF-THEN-ELSE CONSTRUCTION REQUIRES REPETITION OF A FEW LINES.   */
   /*                                                                     */
   /*TO CHANGE THE DIMENSION OF THE VECTOR TO BE SORTED, ALWAYS CHANGE THE  */
   /*INITIALIZATION OF T.  IF THE SIZE EXCEEDS 4096, CHANGE THE SIZE OF T,TOO.*/
   /*A MORE GENERAL VERSION WOULD PARAMETERIZE THE DIMENSION OF V.        */
   /*                                                                     */
   /*THE PASSED INPUT VECTOR IS REPLACED BY THE REORDERED OUTPUT VECTOR.    */
   /*********************************************************************/

⑥/* LEGEND  (ZERO-ORIGIN INDEXING)                                       */

  DECLARE
    (H,                     /*INDEX FOR INITIALIZING T                   */
     I,                     /*INDEX OF ITEM TO BE REPLACED               */
     J,                     /*INITIAL INDEX OF BRANCHES FROM NODE I      */
     K) BINARY FIXED,       /*INDEX IN OUTPUT VECTOR                     */

    (MINF,                  /*MINUS INFINITY                             */
     PINF) BIT (48),        /*PLUS INFINITY                              */

     V (*)   BIT (*),       /*PASSED VECTOR TO BE SORTED AND RETURNED    */

     T (0:8190) BIT (48); /*WORKSPACE CONSISTING OF VECTOR TO BE SORTED, FILLED*/
                          /*OUT WITH INFINITIES, PRECEDED BY LOWER LEVELS  */
                          /*FILLED UP WITH MINUS INFINITIES                */

  /* NOW INITIALIZATION TO FILL DUMMY LEVELS, TOP LEVEL, AND UNUSED PART OF TOP*/
  /* LEVEL AS REQUIRED.                                                    */

⑦ INIT: MINF= (48) '0'B;
        PINF= (48) '1'B;

        DO L=   0 TO 4094;  T(L) = MINF;       END;
        DO L=   0 TO 2499;  T(L+4095) = V(L);  END;
        DO L=6595 TO 8190;  T(L) = PINF;       END;

⑧ K0:  K = -1;                                                          ⑩
   K1:  I = 0;                  /*          ⑪                          <-------| */
   K3:  J = 2*I+1;              /*SET J TO SCAN BRANCHES FROM NODE I.    <------|| */
   K7:  IF T(J) <= T(J+1)       /*PICK SMALLER BRANCH                    ------>|| */
          THEN                  /*                                          ||| */
          ⑨DO;      ⑫          /*                                          ||| */
   K11:       T(I) = T(J); /*REPLACE                                       ||| */
   K13:       IF T(I) = PINF THEN GO TO K16; /*IF INFINITY, REPLACEMENT_*'__|||*/
                           /* IS FINISHED                                 |||| */
   K12:       I = J;       /*SET INDEX FOR HIGHER LEVEL                    |||| */
            END;           /*                                             |||| */
          ELSE             /*                                          <---+|| */
            DO;            /*                                              | || */
   K11A:      T(I) = T(J+1); /*                                            | || */
   K13A:      IF T(I) = PINF THEN GO TO K16;      /*                   __*_| || */
   K12A:      I = J+1;     /*                                              | || */
            END;           /*                                           _< | || */
   K14:  IF 2*I < 8191 THEN GO TO K3;   /*GO BACK IF NOT ON TOP LEVEL  ----+-|| */
   K15:  T(I) = PINF;           /*IF TOP LEVEL, FILL WITH INFINITY        |  | */
   K16:  IF T(0) = PINF THEN RETURN;    /*TEST END OF SORT           <---|  | */
   K17:  IF T(0) = MINF THEN GO TO K1;  /*FLUSH OUT INITIAL DUMMIES   ___*__| */
   K18:  K = K+1;               /*STEP STORAGE INDEX                      | */
         V(K) = T(0);  GO TO K1;⑫       /*STORE OUTPUT ITEM           -------| */
  END QLTSRT7;
```

图 15-3　一段自文档化程序

(1) 为每次运行使用单独的任务名称。维护一份日志，记录程序运行的目的、时间和结果。如果名称由一个助记符(这里是 QLT)和数字后缀(这里是 4)组成，那么后缀可以作为运行编号，把列表和日志联系在一起。这种技术要求为每次运行准备新的任务卡，不过这项工作可以采用

“重复进行公共信息的批处理”来完成。

(2) 使用包含版本号和能帮助记忆的程序名称。即，假设程序将会有很多版本。例子中使用的是1967年的最低一位数字。

(3) 在过程(PROCEDURE)的注释中，包含记叙性的描述文字。

(4) 尽可能地为基本算法提供参考引用，通常它会指向更完备的处理方法。这样，既节省了空间，还允许那些有经验的读者能非常自信地略过这一段内容。

(5) 显示和算法书籍中的传统算法的关系。

a) 更改　　b) 定制细化　　c) 重新表达

(6) 声明所有的变量。采用助记符，并使用注释把 DECLARE 转化成完整的说明。注意，声明已经包含了名称和结构性描述，需要增加的仅仅是对目的的解释。通过这种方式，可以避免在不同的处理中重复名称和结构性的描述。

(7) 用标签标记出初始化的位置。

(8) 对程序语句进行分组和标记，以显示与算法描述文档中语句单元的一致性。

(9) 利用缩进表现结构和分组。

(10) 在程序列表中，手工添加逻辑箭头。它们对调试和变更非常有帮助，还可以补充在页面右边的空白处(注释区域)，成为机器可读文字的一部分。

(11) 使用行注释或标记任何不很清楚的事情。如果采用了上述技术，那么注释的长度和数量都将小于传统惯例。

(12) 把多条语句放置在一行，或者把一条语句拆放在若干行，以吻合逻辑思维，体现和其他算法描述一致。

为什么不？这种方法的缺点在什么地方？很多曾经遇到的困难，已经随着技术的进步逐渐解决了。

最强烈的反对来自必须存储的源代码规模的增加。随着编程技术越来越向在线源代码存储的方向发展，这成了一个主要的考虑因素。我发现自己编写的 APL 程序注释比 PL/I 程序要少，这是因为 APL 程序保存在磁盘上，而 PL/I 则以卡片的形式存储。

然而，文本编辑的访问和更新也在向在线存储的方向前进。就像前面讨论过的，程序和文字的混合使用**减少**了需要存储的字符总数。

对于自文档化程序需要更多输入击键的争论，也有类似的答案。采用打字方式，每份草稿、每个字符需要至少一次击键。而自文档化程序的字符总数更少，且由于电子草稿不需要重复打印，每个字符需要的击键次数也更少。

那么流程图和结构图的情况又如何呢？如果仅仅使用最高级别的结构图，另外使用一份文档的方法可能更安全一些，因为结构通常不会频繁变化。它理所当然也可以作为注释合并到源程序中。这显然是一种聪明的做法。

以上讨论的用于文档和软件汇编的方法到底有多大的应用范围呢？我认为“自文档化”方法的基本思想可以得到大规模的应用。“自文档化”方法对空间和格式的要求更为严格，这一点的应用可能会受限；而命名和结构化声明显然可以利用起来，在这方面，宏可以起到很大的帮助；另外，段落注释的广泛使用在任何语言中都是一个很棒的实践。

高级语言的使用激发了自文档化方法，特别是用于在线系统的高级语言——无论是对批处理还是交互式，它都表现出最强的功效和应用的理由。如同我曾经提到的，上述语言和系统强有力地帮助了编程人员。因为是机器为人服务的，而不是人为机器服务的。因此从各个方面来说，无论是从经济上还是从以人为本的角度来说，它们的应用都是非常合情合理的。

CHAPTER 16

THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 16 章

没有银弹

——软件工程中的根本和次要问题

No Silver Bullet——Essence and Accident in Software Engineering

1685 年的德国线雕画《艾森巴赫的狼人》(*The Werewolf of Eschenbach*)

资料来源：Courtesy of The Grainger Collection, New York

在未来的十年内，无论是在技术还是管理方法上，都看不出有任何突破性的进步，能够保证在十年内大幅度地提高软件的生产率、可靠性和简洁性。

There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

摘要[1]

所有软件活动包括：根本任务，即打造构成抽象软件实体的复杂概念结构；次要任务，即使用编程语言表达这些抽象实体，在空间和时间限制下将它们映射成机器语言。软件生产率在近年来取得的巨大进步来自对人为障碍的突破，例如硬件的限制、笨拙的编程语言和机器时间的缺乏等，这些障碍使次要任务实施起来异常艰难。相对根本任务而言，软件工程师在次要任务上花费了多少时间和精力？除非它占了所有工作的 9/10，否则即使全部次要任务的时间缩减到零，也不会带来生产率数量级上的提高。

因此，现在是关注软件任务中的必要活动的时候了，也就是那些和构造异常复杂的抽象概念结构有关的部分。我建议：

- 仔细地进行市场调研，避免开发已上市的产品；
- 在获取和制定软件需求时，将快速原型开发作为迭代计划的一部分；
- 有机地更新软件，随着系统的运行、使用和测试，逐渐添加越来越多的功能；
- 不断挑选和培养杰出新生代的概念设计人员。

介绍

在所有恐怖民间传说的妖怪中，最可怕的是人狼，因为它们可以完全出乎意料地从熟悉的面孔变成可怕的怪物。为了对付人狼，我们在寻找可以消灭它们的银弹。

大家熟悉的软件项目具有一些人狼的特性(至少在非技术经理看来),常常看似简单明了的东西,却有可能变成一个落后进度、超出预算、存在大量缺陷的怪物。因此,我们听到了近乎绝望的寻求银弹的呼唤,寻求一种可以使软件成本像计算机硬件成本一样迅速降低的尚方宝剑。

但是,我们看看近十年来的情况,没有发现银弹的踪迹。没有任何技术或管理上的进展,能够独立地许诺在生产率、可靠性或简洁性上取得数量级的提高。本章我们试图通过分析软件问题的本质和很多候选银弹的特征,来探索其原因。

不过,怀疑论者并不是悲观主义者。尽管我们没有看见令人惊异的突破,并认为这种银弹实际上是与软件的内在特性相悖的,不过还是出现了一些令人振奋的革新。这些方法的规范化、持续地开拓、发展和传播确实是可以在将来使生产率产生数量级的提高。虽然没有通天大道,但是路就在脚下。

解决管理灾难的第一步是将大块的“巨无霸理论”替换成“微生物理论”,它的每一步——希望的诞生,本身就是对一蹴而就型解决方案的冲击。它告诉工作者进步是逐步取得的,要伴随着辛勤的劳动,对规范化过程进行持续不懈的努力。由此,诞生了现在的软件工程。

根本困难

不仅仅是在目力所及的范围内没有发现银弹,软件的特性本身也导致不大可能有任何的发明创新——能够像计算机硬件工业中的电子器件、晶体管、大规模集成一样——提高软件的生产率、可靠性和简洁程度。我们甚至不能期望每两年有两倍的增长。

首先，我们必须看到这样的畸形并不是由于软件发展得太慢，而是因为计算机硬件发展得太快。从人类文明开始，没有任何其他产业技术的性价比，能在30年之内取得6个数量级的提高，也没有任何一个产业可以在性能提高或者成本降低方面取得如此的进步。这些进步来自计算机制造产业的转变，从装配工业转变成流水线工业。

其次，让我们了解中间的困难，来看看我们能期待什么样的软件技术产业发展速度。效仿亚里士多德，我将它们分成**根本的**(essence)——软件特性中固有的困难，次要的(accident)——出现在目前生产中，但并非那些与生俱来的困难。

我们在下一节中讨论次要问题。首先，我们来关注根本问题。

一个相互牵制关联的概念结构是软件实体必不可少的部分，它包括：数据集合、数据条目之间的关系、算法和功能调用等。这些要素本身是抽象的，体现在不同的表现形式下的概念构造是相同的。尽管如此，它仍然是内容丰富的和高度精确的。

我认为软件开发中困难的部分是规格说明、设计和测试这些概念上的结构，而不是对概念进行表达和对实现逼真程度进行验证。当然，我们还会犯一些语法错误，但是与绝大多数系统中的概念错误相比，它们是微不足道的。

如果这是事实，软件开发总是非常困难的，天生就没有银弹。

让我们来考虑现代软件系统中这些无法规避的内在特性：复杂度、一致性、可变性和不可见性。

复杂度。规模上，软件实体可能比任何由人类创造的其他实体更复杂，因为没有任何两个软件部分是相同的(至少在语句的级别上)。如果有相同的情况，我们会把它们合并成供调用的子函数。在这个方面，软

件系统与计算机、建筑或者汽车大不相同，后者往往存在着大量重复的部分。

数字计算机本身就比人类建造的大多数东西复杂。计算机存在很多种状态，这使得构思、描述和测试都非常困难。软件系统的状态又比计算机的状态多若干个数量级。

同样，软件实体的扩展也不仅仅是相同元素重复添加，而必须是不同元素实体的添加。大多数情况下，这些元素以非线性递增的方式交互，因此整个软件复杂度要比非线性增长多得多。

软件的复杂度是根本属性，不是次要因素。因此，抽掉复杂度的软件实体描述常常也去掉了一些本质属性。数学和物理学在过去三个世纪取得了巨大的进步，数学家和物理学家们为复杂的现象建立了简化的模型，从模型中抽取出各种特性，并通过试验来验证这些特性。这些方法之所以可行，是因为模型中忽略的复杂度不是被研究现象的根本属性。当复杂度是本质特性时，这些方法就行不通了。

上述软件特有的复杂度造成了很多经典的软件产品开发问题。由于复杂度，团队成员之间的沟通非常困难，导致了产品瑕疵、成本超支和进度延迟；由于复杂度，使列举(还远远不是理解)所有可能的状态变得十分困难，影响了产品的可靠性；由于函数的复杂度，函数调用变得困难，导致程序难以使用；由于结构性复杂度，程序难以在不产生副作用的情况下用新函数扩充；由于结构性复杂度，还造成很多安全机制状态上的不可见性。

复杂度不仅仅导致技术产生困难，还引发了很多管理上的问题。它使全面理解问题变得困难，从而妨碍了概念上的完整性；它使所有离散出口难以寻找和控制；它引起了大量学习和理解上的负担，使开发慢慢演变成了一场灾难。

一致性。并不是只有软件工程师才面对复杂度问题。物理学家甚至在非常“基础”的级别上，也会面对异常复杂的事物。不过，物理学家坚信，必定存在着某种通用原理，或者在夸克中，或者在统一场论中。爱因斯坦曾不断地重申自然界一定存在着简化的解释，因为上帝不是专横武断或反复无常的。

软件工程师却无法从类似的信念中获得安慰，他必须掌握的很多复杂度是随心所欲、毫无规则可言的，来自若干必须遵循的人为惯例和系统。它们随接口的不同而改变，随时间的推移而变化，而且，这些变化不是必需的，仅仅由于它们是不同的人——而非上帝——设计的结果。

许多情况下，因为是开发最新的软件，它必须遵循各种接口。另一些情况下，软件的开发目标就是兼容性。在上述的所有情况中，很多复杂性来自保持与其他接口的一致性，对软件的任何再设计，都无法简化这些复杂特性。

可变性。软件实体经常会遭受到持续的变更压力。当然，建筑、汽车、计算机也是如此。不过，工业制造的产品在出厂之后不会经常地发生修改，它们会被后续模型取代，或者必要更改会被整合到具有相同基本设计的后续产品系列中。汽车的更改十分罕见，计算机的现场调整有时发生。然而，与软件的现场修改比起来，它们都要少很多。

其中的部分原因是因为系统中的软件包含了很多功能，而功能是最容易感受变更压力的部分。另外的原因是因为软件可以很容易地进行修改——它是纯粹思维活动的产物，可以无限扩展。日常生活中，建筑有可能发生变化，但众所周知，建筑修改的成本很高，从而打消了那些想提出修改的念头。

所有成功的软件都会发生变更。现实工作中，经常发生两种情况。当人们发现软件很有用时，会在原有应用范围的边界，或者在超越边界的情况下使用软件。功能扩展的压力主要来自那些喜欢基本功能，又对软件提出了很多新用法的用户们。

其次，软件一定是在某种计算机硬件平台上开发，成功软件的生命期通常比当初开发软件所用的计算机硬件平台的生命期要长。即使不是更换计算机，也有可能是换新型号的磁盘、显示器或者打印机。软件必须与各种新生事物保持一致。

简言之，软件产品扎根于文化的母体中，如各种应用、用户、自然及社会规律、计算机硬件等。后者持续不断地变化着，这些变化无情地强迫着软件也随之变化。

不可见性。软件是不可见的和无法可视化的。例如，几何抽象是强大的工具。建筑平面图能帮助建筑师和客户一起评估空间布局、进出的运输流量和各个角度的视觉效果。这样，矛盾变得突出，遗漏的地方可以捕捉到。同样，机械制图、化学分子模型尽管是抽象模型，但都起了相同的作用。总之，都可以通过几何抽象来捕获物理存在的几何特性。

软件的客观存在不具有空间的形体特征。因此，没有已有的几何表达方式，就像陆地海洋有地图，硅片有膜片图，计算机有电路图一样。当我们试图用图形来描述软件结构时，发现它是很多相互关联、重叠在一起的图形。这些图形可能代表控制流程、数据流、依赖关系、时间序列和名字空间的相互关系等。它们通常不是有较少层次的扁平结构。实际上，在上述结构上建立概念控制的一种方法是强制将关联分割，直到可以层次化一个或多个图形。[2]

除去软件结构上的限制和简化方面的进展，软件仍然保持着无法可视化的固有特性，从而剥夺了一些具有强大功能的概念工具的构造创意。这种缺憾不仅限制了个人的设计过程，也严重地阻碍了思路相互之间的交流。

以往解决次要困难的一些突破

如果回顾一下软件领域中取得的最富有成效的三次进步，我们会发现每一次都是解决了软件构建上的巨大困难，但是这些是次要困难，不是本质属性，也不是主要困难。同样，我们可以对每一次进步进行外推，来了解它们的固有限制。

高级语言。毋庸置疑，软件生产率、可靠性和简洁性上最有力的突破是使用高级语言编程。大多数观察者相信开发生产率至少提高了 5 倍，可靠性、简洁性和理解程度也大为提高。

那么，高级语言取得了哪些进展呢？它减轻了一些次要的软件复杂度。抽象程序包含了很多概念上的要素：操作、数据类型、流程和相互通讯，而具体的机器语言程序则关心位、寄存器、条件、分支、通道、磁盘等。高级语言所达到的抽象程度体现了抽象程序所需的要素，避免了更低级的元素，它消除了并不是程序所固有的整个级别的复杂度。

高级语言最可能实现的是为所有编程人员在抽象程序中提供想到的要素。可以肯定的是，我们正在逐渐思考数据结构、数据类型和操作的复杂程度，只不过是以非常缓慢的速度。另外，程序开发方法越来越接近用户的复杂度。

然而，对于较少使用那些复杂深奥语言要素的用户，高级语言在某种程度上增加而不是减少了脑力劳动的负担。

分时。大多数观察者相信，分时在提高程序员的生产率和产品质量方面起到了很大作用，尽管它产生的进步不如高级语言。

分时解决了完全不同的困难，保证了及时性，从而使我们能维持对复杂度的一个总体把握。批处理编程的较长周转时间意味着会不可避免地遗忘一些细枝末节。如果我们停止编程，调用编译程序或者执行程序，思维上的中断将使我们不得不重新进行思考，它在时间上的代价非常高昂。最严重的结果可能是失去对复杂系统的掌握。

较长的周转时间与机器语言的复杂度一样，是软件开发过程的次要困难，而不是本质困难。这直接导致分时作用非常有限。主要效果缩短了系统的响应时间。随着它接近于零，超过人类可以辨识的基本能力——大概 100 毫秒时，所获得的好处就接近于零了。

统一编程环境。第一个集成开发环境——Unix 和 Interlisp，现在已经得到了广泛应用，并且由于整合的因素使生产率提高。为什么？

它们主要通过提供集成库、统一文件格式、管道和过滤器，解决了共同使用程序的次要困难。这样，概念性结构理论上的相互调用、提供输入和互相使用，在现实中可以非常容易地实现。

因为每个新工具可以通过标准格式在任何一个程序中应用，这种突破接着又激发整个工具库的开发。

由于这些成功，环境开发是当今软件工程研究的主要题目。我们将在下节讨论期望达到的目标和限制。

银弹的希望

现在，让我们来讨论一下当今可能作为潜在银弹的最先进的技术进步。它们各自针对什么样的问题？它们属于必要问题，或者依然是我们接下来要解决的次要困难？它们是提供了创新，还是仅仅做出了一些改进？

Ada 和其他高级编程语言。近来，最被吹捧的开发进展之一是编程语言 Ada，一种 20 世纪 80 年代的高级语言。Ada 实际上不仅仅反映了语言概念上的突破性进展，而且蕴涵了鼓励现代设计和模块化概念运用的重要特性。由于 Ada 采用的是抽象数据类型、层次结构的模块化理念，所以 Ada 理念可能比语言本身更加先进。Ada 使用设计来承载需求，作为这一过程的自然产物，它可能过于丰富了。不过，这并不是致命的，因为它的词汇子集可以解决学习问题，硬件的进展能提供更高的 MIPS(每秒百万指令集)，减少编译的成本。软件系统结构化的先进理念实际上非常好地利用了 MIPS 上的进展。20 世纪 60 年代，曾在内存和循环成本上广受谴责的操作系统，如今已被证明是一种能使用某些 MIPS 和廉价内存的非常优秀的系统。

然而，Ada 仍然不是消灭软件生产率怪兽的银弹。毕竟，它只是另一种高级语言，这类语言最大的回报来自第一次切换，它通过使用更加抽象的语句来开发，降低了机器的次要复杂度。一旦这些难题被解决，就只剩下非常少的问题了，解决剩余部分的获益必然也要少一些。

我预言，在以后的 10 年中，当 Ada 的效率被大家评估认可时，它会产生相当大的变化，但并不是因为任何特别的语言特性，不是由于这些语言特性被合并在一起，也不是因为 Ada 开发环境会不断发展进步。Ada 的最大贡献在于编程人员培训方式的转变，即需要对开发人员进行现代软件设计技术培训。

面向对象编程。软件专业的一些学生坚持面向对象编程是当今若干新潮技术中最富有希望的。[3]我也是其中之一。达特茅斯的 Mark Sherman 提出，必须仔细地区别两个不同的概念：抽象数据类型和层次化类型，后者也被称为**类**(class)。抽象数据类型的概念是指对象类型应该通过一个名称、一系列合适的值和操作来定义，而不是理应被隐藏的存储结构。抽象数据类型的例子是 Ada 包(以及私有类型)和 Modula 的模块。

层次化类型，如 Simula-67 的类，是允许通用接口的定义被后续子类型精化的。这两个概念是互不相干的——可以只有层次化，没有数据隐藏；也可能是只有数据隐藏，而没有层次化。两种概念都体现了软件开发领域的进步。

它们的出现都消除了开发过程中的非本质困难，允许设计人员表达自己设计的内在特性，而不需要表达大量句法上的内容，这些内容并没有添加新的信息。对于抽象数据类型和层次化类型，它们都解决了高级别的次要困难并允许采用较高层次的表现形式来表达设计。

不过，这些提高仅仅能消除所有设计表达上的次要困难。软件的内在问题是设计的复杂度，上述方法并没有对它有任何的促进。除非在我们现在的编程语言中，不必要的低层次类型说明占据了软件产品设计的90%，面向对象编程才能带来数量级上的提高。对面向对象编程这颗“银弹”，我深表怀疑。

人工智能。很多人期望人工智能上的进展可以给软件生产率和质量带来数量级上的增长，[4]但我不这样认为。究其原因，我们必须剖析“人工智能”意味着什么，以及它是如何应用的。

Parnas 澄清了术语上的混乱：

现在有两种差异非常大的 AI 定义被广泛使用。AI-1：使用计算机来解决以前只能通过人类智慧解决的问题。AI-2：使用启发式或基于规则

的特定编程技术。在这种方法中，对人类专家进行了研究，判断他们解决方法的启发性思维或者经验法则……程序被设计成以人类解决问题的方式来运作。

第一种定义的意义容易发生变化……今天可能适合 AI-1 定义的程序，一旦我们了解了它的运行方式，理解了问题，就不再认为它是人工智能……遗憾的是，我无法识别这个领域的特定知识体系……绝大多数工作是针对问题域的，我们需要一些抽象或者创造性来解决上述问题。[5]

我完全同意这种批评意见。语音识别技术与图像识别技术的共同点非常少，它们又都与专家系统中应用的技术不同。例如，我觉得很难去发现图像识别技术能给编程开发实践带来什么样的差异。同样，语音识别也差不多——软件开发上的困难是决定说什么，而不是如何说。表达的简化仅仅能提供少量的促进作用。

至于 AI-2 专家系统技术，应该用专门的章节来讨论。

专家系统。人工智能领域最先进的、被最广泛使用的部分，是开发专家系统的技术。很多软件科学家正非常努力地工作着，想把这种技术应用在软件的开发环境中。[6]那么它的概念是什么，前景如何？

专家系统是包含归纳推论引擎和规则基础的程序，它接收输入数据和假设条件，通过从基础规则推导逻辑结果，提出结论和建议，向用户展示前因后果，并解释最终的结果。推论引擎除了处理推理逻辑以外，通常还包括复杂逻辑或者概率数据和规则。

对于解决相同的问题，这种系统明显比传统的程序算法要先进很多。

- 推论引擎技术的开发独立于应用程序，因此可以有多种用途。在该引擎上付出较大的力气是很合理的。实际上，这种引擎技术非常先进。

- 基于应用的、可变更的部分，在基础规则中以一种统一的风格编码，并且为规则基础的开发、更改、测试和文档化提供了若干工具。这实际上对一些应用程序本身的复杂度进行了系统化。

Edward Feigenbaum 指出，这种系统的能力不是来自某种前所未有的推导机制，而是来自非常丰富的知识积累基础，这一基础更加精确地反映了现实世界。我认为这种技术提供的最重要进步是具体应用的复杂度与程序本身相分离。

如何把它应用在软件开发工作中呢？可以通过很多途径：建议接口规则、制定测试策略、记录各种 bug 产生的频率和提供优化建议，等等。

例如，考虑一个虚构的测试顾问系统。在最根本的级别，诊断专家系统和飞行员的检查列表很相似，对困难可能的成因提供基本建议。建立基础规则时，可以依据更多的复杂问题征兆报告，从而使这些建议更加精确。可以想象，一种调试辅助程序起初提供的是一般化建议，随着基础规则包括越来越多的系统结构信息，它产生的推测和推荐的测试也越来越准确。这种类型的专家系统可能与传统系统彻底分离，系统中的规则基础可能与相应的软件产品具有相同的层次模块化结构，因此当对产品进行模块化修改时，诊断规则也能相应地进行模块化修改。

产生诊断规则也是在为模块和系统编制测试用例集时必须完成的任务。如果它以一种适当通用的方式来完成，对规则采用一致的结构，拥有一个良好可用的推测引擎，事实上它就可以减少测试用例设计的总体工作量，并帮助整个软件生命周期的维护和修改测试。同样，我们可以推测其他的顾问专家系统——可能是它们中的某一些，或者是较简单的

系统——能够用在软件开发的其他部分。

在较早实现的用于软件开发的专家顾问系统中存在着很多困难。在我们假设的例子中，一个关键的问题是寻找一种简单的方法，能从软件结构的技术说明中，自动或者半自动地产生诊断规则。另外，更加重要也是更加困难的任务是：寻觅能够清晰表达，深刻理解来龙去脉，前因后果事情的自我分析专家；开发有效的技术——抽取专家们所了解的知识，把它们精炼成基础规则。这项工作的工作量是知识获取工作量的两倍。构建专家系统的必要前提条件是拥有专家。

专家系统最强有力的贡献是给缺乏经验的开发人员提供服务，用最优秀的开发者的经验和知识积累为他们提供指导，这是非常大的贡献。最优秀和一般的软件工程实践之间的差距是非常大的，可能比其他工程领域中的差距都要大，一种传播优秀实践的工具特别重要。

“自动”编程。近40年，人们一直在预言和编写有关“自动编程”的文字，从问题的一段陈述说明自动产生解决问题的程序。现在，仍有一些人期望这样的技术能够成为下一个突破点。[7]

Parnas暗示这是一个用于魔咒的术语，本身的语义是不完整的，并断言：

一句话，自动编程总是成为一种热情，使用现在并不可用的更高级语言编程的热情。[8]

他指出，大多数情况下所给出的技术说明本质上是问题的解决方法，而不是问题自身。

可以找到一些例外情况。例如，数据发生器的开发技术就非常实用，并经常用于排序程序中。一些微积分方程系统也允许直接描述问题。系统评估若干参数，从问题解决方案库中进行选择，生成合适的程序。

这样的应用具有非常良好的特性：

- 问题通过相对较少的参数迅速地描述特征；
- 存在很多已知的解决方案，提供了可供选择的库；
- 在给定问题参数的前提下，大量广泛的分析为选择具体的解决技术提供了清晰的规则。

具有上述简洁属性的系统是一个例外，除此之外很难看到该方法能普及到更广泛的寻常软件系统，甚至难以想象这种突破如何能够进行推广。

图形化编程。在软件工程的博士论文中，一个很受欢迎的主题是图形化或可视化编程，计算机图形在软件设计上的应用。[9]这种方法的推测部分来自 VLSI 芯片设计的类比，计算机图形化在该设计中扮演了高生产力的角色。部分源于——人们将流程图作为一种理想的设计介质，并为构建它们提供了很多功能强大的实用程序——这证实了图形化的可行性。

不过，上述方法中至今还没有出现任何令人信服，更不用说令人激动的进步。我确信将来也不会出现。

首先，如同我先前所提出的，流程图是一种非常差劲的软件结构表达方法。[10]实际上，它最好被视为是 Burks、Von Neumann 和 GoldStine 试图为他们所设计的计算机提供的一种当时迫切需要的高级控制语言。如今的流程图已经变得复杂了，一张图有若干页，有很多连接结点，这种表现形式实在令人同情。流程图已经被证明是完全不必要的设计工具——程序员是在开发之后，而不是之前绘制描述程序的流程图。

其次，现在的屏幕非常小，就像素级别，无法同时表现软件图形的所有正式、详细的范围和细节。现在所谓“类似桌面”的工作站实际上像是“飞机坐舱座椅”。在飞机上任何坐在两个肥胖乘客之间，反复挪动一大兜文件的人都会意识到这中间的差别——每次只能看到很少的内容。真正的桌面提供了很多文件的总览，让大家可以随意地使用它们。此外，当人们的创造力一阵阵地涌现时，开发人员大多数都会舍弃工作台，使用空间更为广阔的地板。要使我们面对的工作空间满足软件开发工作的需要，硬件技术必须进一步发展。

更加基本的是，如同我们上面所讨论的，软件非常难以可视化。即使用图形表达出了流程图、变量范围嵌套情况、变量交叉引用、数据流和层次化数据结构等，也只是表达了某个方面，就像盲人摸象一样。如果我们把很多相关的视图叠加在所产生的图形上，那么很难再抽取出全局的总体视图。对 VLSI 芯片设计方法的类推是一种误导——芯片设计是对两维对象的层次设计，几何特性反映了它的本质特性，而软件系统不是这样。

程序验证。现代编程的许多工作是测试和修复 bug。是否有可能出现银弹，能够在系统设计阶段、源代码阶段就消除 bug？是否可以在大量工作被投入实现和测试之前，通过采用证实设计正确性的“深奥”策略，彻底提高软件的生产率和产品的可靠性？

我并不认为这里能找到魔法。程序验证的确是很先进的概念，它对安全操作系统内核等这类应用是非常重要的。不过，这项技术并不能保证节约劳动力。验证要求如此多的工作量，最终却只有少量的程序能够真正得到验证。

程序验证并不意味着零缺陷的程序。这里也没有什么魔术，数学验证仍然可能是有错误的。因此，尽管验证可能减少程序测试的工作量，

却不能省略程序测试。

更严肃地说，即使完美的程序验证也只能建立满足技术说明的程序。这时，软件工作中最困难的部分已经接近完成，形成了完整和一致的说明。开发程序的一些必要工作实际上已经变成了对技术规格说明进行的测试。

环境和工具。向更好的编程开发环境研究中投入，我们可以期待得到多少回报呢？人的本能反应是首先着手解决高回报的问题：层次化文件系统，统一文件格式以获得一致的编程接口和通用工具等。特定语言的智能化编辑器在现实中还没有得到广泛应用，不过它们最有希望实现的是消除语法错误和简单的语义错误。

开发环境上，现在已经实现的最大成果可能是集成数据库的使用，用来跟踪大量的开发细节，供每个程序员精确地查阅信息，并在整个团队协作开发中保持最新的状态。

显然，这样的工作是非常有价值的，它能带来软件生产率和可靠性方面的一些提高。但是，由于自身的特性，目前它的回报应该很有限。

工作站。随着工作站的处理能力和内存容量的稳固和快速提高，我们能期望在软件领域取得多大的收获呢？有多少 MIPS 可供自由使用呢？现在的运算速度已经可以完全满足程序编制和文档书写的需要了。编译还需要一些提高，不过一旦机器运算速度提高 10 倍，程序开发人员的思考活动将成为日常工作的主要活动。实际上，这已经是现在的情况了。

我们当然欢迎更加强大的工作站，但是不能期望有魔术般的提高。

针对概念上根本问题的颇具前途的方法

虽然现在软件上没有技术上的突破能够预示我们可以取得像在硬件领域上一样的进展，但在现实的软件领域中，既有大量优秀的工作，也有不引人注意的平稳进步。

所有针对软件开发过程中次要困难的技术工作基本上能表达成以下的生产率公式：

$$\text{任务时间}=\sum_i (\text{频率})_i \times (\text{时间})_i$$

如果和我所认为的一样，工作的创造性部分占据了大部分时间，那些仅仅是表达概念的活动并不能在很大程度上影响生产率。

因此，我们必须考虑那些解决软件上必要困难的活动——即，准确地表达复杂概念结构。幸运的是，其中的一些问题非常有希望被解决。

购买和自行开发。构建软件最可能的彻底解决方案是不开发任何软件。

情况每一天都在好转，越来越多的软件提供商为各种眼花缭乱的应用程序提供了质量更好、数量更多的软件产品。当我们的软件工程师正忙于生产方法论时，个人计算机的惊天动地的变化为软件创造了广阔的市场。每个报摊上都有大量的月刊，根据机器的类型，刊登着从几美元到几百美元的各种产品的广告和评论。更多专业厂商为工作站和 Unix 市场提供了很多非常有竞争力的产品，甚至很多工具软件和开发环境软件都可以随时购买使用。对于独立的软件模块市场，我已在其他地方提出了一些建议。

以上提到的任何软件，购买都比重新开发要低廉一些。即使支付100 000 美元，购买的软件也仅仅是一个人年的成本。而且软件是立即可用的。至少对于现有的产品，对于那些专注于该领域开发者的成果而言，它们是可以立刻投入使用的。并且，它们往往配备了书写良好的文档，在某种程度上比自行开发的软件维护得更加完备。

我相信，这个大众市场将是软件工程领域意义最深远的开发方向。软件成本一直是开发的而非复制的成本，所以，即使只在少数使用者之间实现共享，也能在很大程度上减少每个用户的成本。另一种看法是使用软件系统的 n 个拷贝，将会使开发人员的生产率有效地提高 n 倍。这是一个领域和行业范围的提高。

当然，关键的问题还是可用性。是否可以在自己的开发工作中使用商用的软件包？这里，有一个令人吃惊的问题。在 20 世纪 50 到 60 年代期间，一个接一个的研究显示，用户不会在工资系统、物流控制、账务处理等系统中使用商用软件包。需求往往过于专业，不同情况之间的差别太大。在 20 世纪 80 年代，我们发现这些软件包的需求大为增加，并得到了大规模的使用。是什么导致了这样的变化呢？

并不是软件包发生了变化。它们可能比以前更加通用和更加定制化一些，但并不太多。同样，也不是应用发生了变化。即使有，今天的商业和学术上的需要也比 20 年前更加不同和复杂。

重大的变化在于计算机硬件/软件成本比率。在 1960 年，200 万美元机器的购买者觉得他可以为定制的薪资系统支付 250 000 美元，而这样的系统很容易慢慢地变得不适用。现在，对 50 000 美元的办公室机器购买者而言，很难想象能为定制薪资系统再支付费用。因此，他们把上述系统的模块进行调整，添加到可用的软件包中。计算机现在如此的普遍，上述的改编和调整是发展的必然结果。

我的上述观点也存在一些戏剧性的例外——软件包的通用化方面并没有发生什么变化，如电子表格和简单的数据库系统。这些强大的工具出现得如此之晚并如此醒目，产生无数的应用，而其中的一些是非正统的。大量的文章甚至书籍讲述了如何使用电子表格应付很多意想不到的问题。原先作为客户程序，使用 Cobol 或者报表生成程序编写的大量应用，如今已经被这些工具取代。

现在很多用户天天操作计算机，使用着各种各样的应用程序，但并不编写代码。事实上，他们中很多人无法为自己的计算机编写任何程序，不过他们非常熟练地使用计算机来解决新问题。

我认为，对于现在的很多组织机构来说，最有效的软件生产率策略是在生产一线配备很多个人计算机，安装好通用的书写、作图、文件管理和电子表格程序以及配备能熟练使用它们的人员，并且把这些人员分配到各个岗位。类似的策略——通用的数学和统计软件包以及一些简单的编程能力，同样适用于很多实验室的科学工作者。

需求精炼和快速原型。开发软件系统的过程中，最困难的部分是确切地决定搭建什么样的系统。概念性工作中，没有其他任何一个部分比确定详细的技术需求更加困难。详细的需求包括了所有的人机界面，与机器和其他软件系统的接口。如果失误了，需求工作对系统的影响比其他任何一个部分都大，当然纠正需求的困难也比其他任何一个部分都要大。

因此，软件开发人员为客户所承担的最重要的职能是不断重复地抽取和细化产品的需求。事实上，客户不知道自己需要什么。他们通常不知道哪些问题是必须回答的。并且，连必须确定的问题细节常常根本不予考虑，甚至只是简单地回答——“开发一个类似于我们已有的手工信息处理过程的新软件系统”——实际上都过于简单。客户决不会仅仅只

要求这些。复杂的软件系统往往是活动的、变化的系统。活动的动态部分是很难想象的。所以，在计划任何软件活动时，要让客户和设计人员之间进行多次广泛的交流沟通，并将其作为系统定义的一部分，这是非常必要的。

这里，我将向前多走一步，下一个定论。在尝试和开发一些客户定制的系统之前，即使他们和软件工程师一起工作，想要完整、精确、正确地抽取现代软件产品的需求——这，实际上也是不可能的。

因此，现在的技术中最有希望的，并且解决了软件的根本而非次要问题的技术，是开发作为迭代需求过程的一部分——快速原型化系统的方法和工具。

软件系统的快速原型对重要的系统界面进行模拟，并演示待开发系统的主要功能。同时，原型不必受到相同硬件速度、规模或者成本的限制。原型通常展示了应用程序的功能主线，但不处理任何如无效输入、退出清除等异常情况。原型的目的是明确实际的概念结构，使客户可以测试一致性和可用性。

现在的软件开发流程是基于如下假设的——事先明确地阐述系统，为系统开发竞标，实际进行开发，最后安装。我认为这种假设根本上就是不正确的，很多软件问题就来自这种谬误。因此，如果不进行彻底地调整，就无法消除那些软件问题。其中，一种改进是对产品和原型不断反复地进行开发和规格化。

增量开发——增长，而非搭建系统。我现在还记得，在1958年，当听到一个朋友提及**搭建**(building)，而不是**编写**(writing)系统时，我所受到的震动。一瞬间，我的整个软件开发流程的视野开阔了。这种暗喻是非常有力和精确的。现在，我们已经理解软件开发是如何类似于其他的

建造过程，并开始随意地使用其他的暗喻，如**规格说明、构件装备、脚手架(测试平台)**(specifications, assembly of components, and scaffolding)。

暗喻“搭建系统”的使用已经有些超出了它的有效期限，是重新换一种表达方式的时候了。如果现在的开发情况与我所考虑的一样，那些概念性的结构非常复杂，以至于难以事先精确地说明和零缺陷地开发，我们必须采用彻底不同的方法。

让我们转向自然界，研究一下生物的复杂性，而不是人们的僵硬工作。我们会发现它们的复杂程度令我们敬畏。仅是大脑本身，就比任何对它的描述都要复杂，比任何的模拟仿真都要强大，它的多样性、自我保护和自我更新能力异常丰富和有力。其中的秘密就是逐步发育成长，而不是一次性搭建。

所以，我们的软件系统也必须如此。很多年前，Harlan Mills 建议所有的软件系统都应该以增量的方式开发。[11]即，首先系统应该能够运行，即使未完成任何有用功能，只能正确调用一系列伪子系统。接着，系统一点一点被充实，子系统轮流被开发，或者是在更低的层次调用程序、模块、子系统的占位符(伪程序)等。

从我在软件工程试验班上开始推动这种方法起，其效果就不可思议了。在过去几十年中，没有任何方法和技术能如此彻底地改变我自己的实践。这种方法迫切地要求自上而下设计，因为它本身是一种自上而下增长的软件。增量化开发使逆向跟踪很方便，并非常容易进行原型开发。每一项新增功能，以及针对更加复杂数据或环境的新模块，从已经规划的系统中有机地增长。

这种开发模式对士气的推动是令人震惊的。当一个可运行系统——即使是非常简单的系统出现时，开发人员的热情就迸发出来了。当一个新图形软件系统的第一幅图案出现在屏幕上时，即使是一个简单的长方形，

工作的动力也会成倍地增长。在开发过程的每个阶段，总有可运行的系统。我发现开发团队可以在 4 个月内，**培育**(grow)出比**搭建**(building)**复杂得多**的系统。

大型项目同样可以得到与我所参与的小型项目相同的好处。[12]

卓越的设计人员。关键的问题是如何提高软件行业的核心，一如既往的是——人员。

我们可以通过遵循优秀而不是拙劣的实践，来得到良好的设计。优秀的设计是可以传授的。程序员的周围往往是最出色的人员，因此他们可以学习到良好的实践。因此，美国的重大策略是颁布各种优秀的现代实践。新课程、新文献，像软件工程研究所(SEI)等新机构的出现都是为了把我们的实践从不足提升到更高的水平。其正确性是毋庸置疑的。

不过，我不认为我们可以用相同的方式取得下一次的进步。低劣设计和良好设计之间的区别可能在于设计方法的完善性，而良好设计和卓越设计之间的区别肯定不是如此。卓越设计来自卓越的设计人员。软件开发是一个**创造性**的过程。完备的方法学可以培养和释放创造性的思维，但它无法孕育或激发创造性的过程。

其中的差异并不小——就像萨列里和莫扎特。一个接一个的研究显示，非常卓越的设计者产生的成果更快、更小、更简单、更干净，实现的代价更少。卓越和一般之间的差异接近于一个数量级。

简单地回顾一下，尽管很多杰出、实用的软件系统是由很多人共同设计开发的，但是那些激动人心、拥有广大热情爱好者的产品往往是一个或者少数伟大设计师们的思想。考虑一下 Unix、APL、Pascal、Modula、Smalltalk 的界面，甚至 Fortran，与之对应的产品是 Cobol、PL/I、Algol、MVS/370 和 MS-DOS(见图 16-1)。

YES	NO
Unix	Cobol
APL	PL/I
Pascal	Algol
Modula	MVS/370
Smalltalk	MS-DOS
Fortran	

图 16-1　激动人心的产品

因此，尽管我强烈地支持现在的技术转移和开发技能的传授，但我认为我们可以做的最重要工作是寻求培养卓越设计人员的途径。

没有任何软件机构可以忽视这项挑战。尽管公司可能缺少良好的管理人员，但决不会比良好设计人员的需求更加迫切，而卓越的管理人员和设计人员都是非常缺乏的。大多数机构花费了大量的时间和精力来寻找和培养管理人员，但据我所知，它们中间没有任何一家在寻求和培育杰出的设计人员上投入相同的资源，而产品的技术特色最终依赖于这些设计人员。

我的第一项建议是每个软件机构必须决定和表明，杰出的设计人员和卓越的管理人员一样重要，他们应该得到相同的培养和回报。不仅仅是薪资，还包括各个方面的认可——办公室规模、家具、个人的设备、差旅费用和人员支持等——必须完全一致。

如何培养杰出的设计人员呢？限于篇幅，不允许进行较长的介绍，但有些步骤是显而易见的。

- 尽可能早地、有系统地识别顶级的设计人员，最好的通常不是最有经验的人员。

- 为设计人员指派一位职业导师，负责他们技术方面的成长，仔细地为他们规划职业生涯。
- 为每个方面制定和维护一份职业计划，包括与设计大师的、经过仔细挑选的学习过程，正式的高级教育和短期的课程——所有这些都穿插在设计和技术领导能力的培养安排中。
- 为成长中的设计人员提供相互交流和激励的机会。

CHAPTER 17

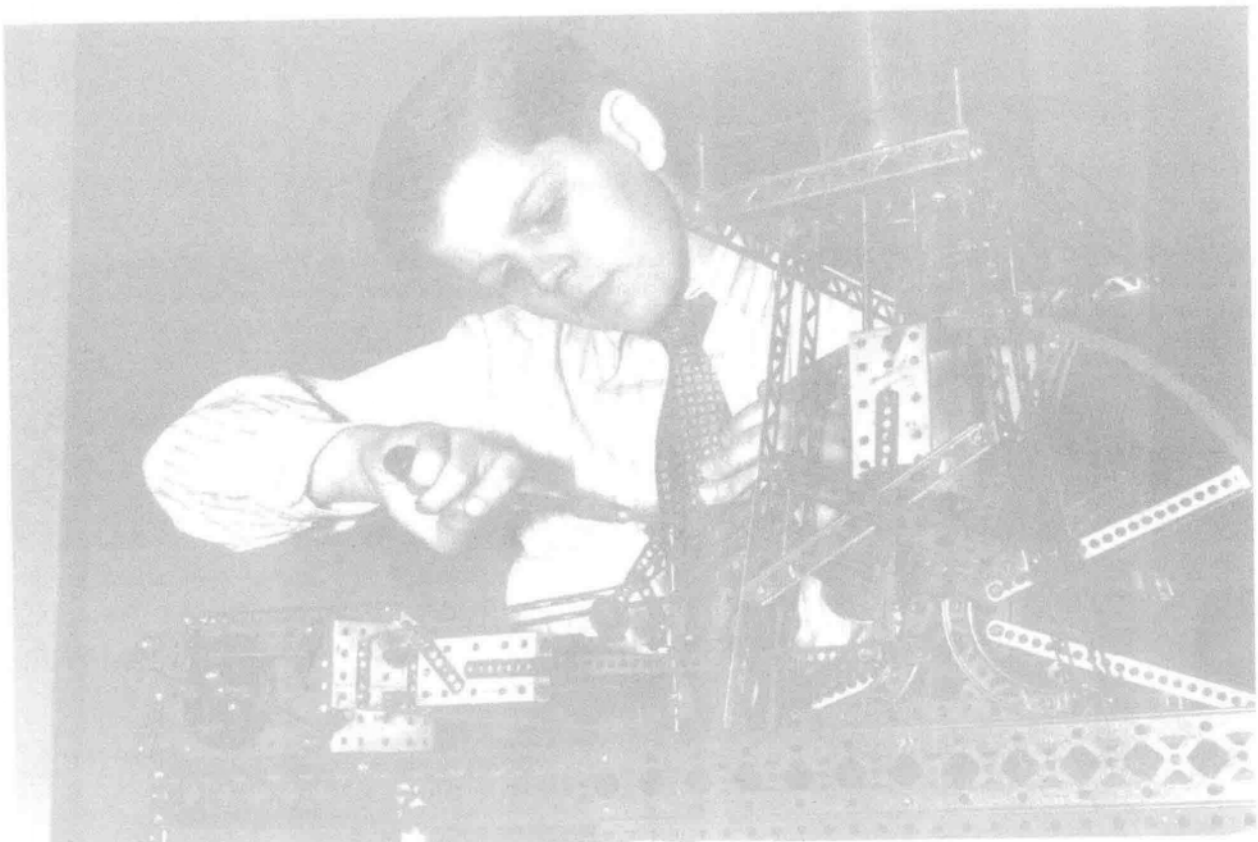
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 17 章

再论“没有银弹”

“No Silver Bullet” Refired



1945 年，人们用现成的零件组装成具有复杂结构的装置

资料来源：The Bettman Archive

生死有命，富贵在天。

——威廉三世，奥兰治亲王

任何人若想看到一件完美无瑕的作品，他所想的那种作品过去不存在，现在和将来也不会出现。

——亚历山大·蒲柏，《批判论文》

Every bullet has its billet.

WILLIAM III OF ENGLAND, PRINCE OF ORANGE

Whoever thinks a faultless piece to see,
Thinks what ne'er was, nor is, nor e'er shall be.

ALEXANDER POPE, *AN ESSAY ON CRITICISM*

人狼和其他恐怖传说

“没有银弹——软件工程中的根本和次要问题”(参见第 16 章)最初是 IFIP 1986 年都柏林大会的约稿，接着在一系列的刊物上发表。[1]《计算机》(*Computer*)杂志上翻印了该文章，封面是一幅类似于《伦敦人狼》(*The Werewolf of London*)影片的恐怖剧照。[2]同时，还有一栏补充报道《杀死人狼》，描述了只有银弹才能完成的(现代)神话。在出版之前，我并未注意到补充报道和插图，也没有料到一篇严肃的技术性文字会被这样润色。

《计算机》杂志的编辑们精于此道，不过，似乎有很多人阅读了那篇文章。因此，我为那一章选择了另一幅人狼插图，一幅对这种近乎滑稽物种的古老素描。我希望这幅并不刺眼的图案有相同的正面效果。

存在着银弹——就在这里

“没有银弹”声称和断定，在近 10 年内，没有任何单独的软件工程进展可以使软件生产率有数量级的提高(引自 1986 年的版本)。现在已经是第九个年头了，因此也该看看这些预言是否得到了应验。

《人月神话》一书被大量地引用，很少存在异议；相比之下，“没有银弹”却引发了众多的辩论，编辑收到了很多文章和信件，至今还在延续。[3]他们中的大多数攻击其核心论点和我的观点——没有神话般的解决方案，将来也不会有。他们大都同意“没有银弹”一文中的多数观点，但接着断定实际存在着杀死软件怪兽的银弹——由他们所发明的银弹。今天，当我重新阅读一些早期的反馈时，我不禁发现在 1986—1987 年间，曾被强烈推崇的秘方并没有出现所声称的戏剧性效果。

在购买计算机软件和硬件时，我喜欢听取那些真正使用过产品并感到满意的用户的推荐。同样，当某个名副其实的中立客户走到面前声称，“我使用了这种方法、工具或者产品，它使我的软件生产率提高了 10 倍。”这时，我很乐意接受银弹已经出现的观点。

很多书信作者进行了若干正确的修订和澄清，其中一些还提供了很有针对性的分析和辩驳，对此我非常感激。本章我将同大家分享这些改进，并对反面意见进行讨论。

含糊的表达将会导致误解

某些作者指出我没有将一些观点表达清楚。

次要(accident)。在第 16 章的摘要中，我已经尽我所能清晰地表达了“没有银弹”一文的主要观点。然而，仍有些观点由于术语“accident”(**偶然**)和“accidental”(**次要**)而被混淆，这些术语来自亚里士多德的古老用法。[4]术语“accidental”，不是指“偶然发生”，也不是指“不幸的”，其意思更接近于“附带的”或者“从属的”。

我并不是贬低软件构建中的次要部分，相反，我认同英国剧作家、侦探小说作者和神学家 D. 塞耶斯看待创造性活动的观点，她认为所有创造性活动包括：(1)概念性结构的形式规格化；(2)使用现实的介质来实现；(3)在实际的使用中，与用户交互。[5]在软件开发中，我称为“必要”(essence)的部分是构思这些概念上的结构；我称为“次要”(accident)的部分是指它的实现过程。

现实问题。对我来说(尽管不是对所有人)，关键论点的正确与否归结为一个现实问题：整个软件开发工作中的哪些部分与概念性结构的精确和有序表达相关，哪些部分是创造那些结构的思维活动？根据缺陷是

概念性的(例如未能识别某些异常)，或者是表达上的问题(例如指针错误或者内存分配错误)等，可以将这些缺陷的寻找和修复工作进行相应的划分。

在我看来，开发的次要或者表达部分现在已经下降到整个工作的一半或一半以下。由于这部分是现实的问题，所以其价值原则上可以应用测量技术来研究。[6]这样，我的观点也可以通过更科学和更新的估计来纠正。值得注意的是，还没有人公开发表或者私下写信告诉我，次要部分的任务占据了工作的 9/10。

“没有银弹”无可争辩地指出，如果开发的次要部分少于整个工作的 9/10，即使不占用任何时间(需要出现**奇迹**)，也不会给生产率带来数量级的提高。因此，必须着手解决开发的根本问题。

由于“没有银弹”，Bruce Blum 把我的注意力引向 Herzberg、Mausner 和 Sayderman 等人在 1959 年的研究。[7]他们发现动机因素可以提高生产率。另一方面，环境和次要因素，无论起到多么积极的作用，仍无法提高生产率。但是在产生负面影响时，它们会使生产率降低。“没有银弹”认为，很多软件开发过程已经消除了以下负面因素：十分笨拙的机器语言，漫长的批处理周转时间、拙劣的工具以及无法忍受的内存限制。

因为是根本困难所以就没有希望？ 1990 年 Brad Cox 的一篇非常出色的论文“这就是银弹”(There Is a Silver Bullet)，有说服力地指出可重用和可交互的构件开发是解决软件根本困难的一种方法。[8]我由衷地表示赞同。

不过，Cox 在两点上误解了“没有银弹”。首先，他理解该文并断定软件困难来自“编程人员缺乏构建当今软件的技术”。而我认为，根本困难是固有的概念复杂性，无论是任何时间，使用任何方法设计和实现软件的功能，它都存在。其次，他(以及其他人)阅读“没有银弹”，

并认定文中的观点是处理软件开发中的根本困难是没有希望的——这不是我的本意。作为本质上的困难，构思软件概念性的结构本身就有复杂性、一致性、可变性及不可见性的特点。不过实际上，每一种困难产生的麻烦都是可以改善的。

复杂性是层次化的。例如，复杂性是最严重的内在困难，但并不是所有的复杂性都是不可避免的。我们的很多软件，但不是全部，来自应用本身随意的复杂特性。MYSIGMA Sødahl 的 Lars Sødahl 和合作伙伴——一家国际管理咨询公司，曾写道：

> 就我的经验而言，在系统工作中所遇到的大多数复杂性是组织结构上一些失误的征兆。试图为这些现实建模，建立同等复杂的程序，实际上是隐藏，而不是解决这些杂乱无章的情况。

Northrop 的 Steve Lukasik 认为，即使是组织机构上的复杂性也不是任意的，可能容易受到策略调整的影响。

> 我曾作为物理学家接受过培训，因此倾向于用更简单的概念来描述“复杂”事物。现在你可能是正确的，我无法断定所有的复杂事物都容易用有序的规律表达……同样的道理，你不能断定它们不能。
>
> ……昨天的复杂性是今天的规律。分子的无序性启迪了气体动力学理论和热力学的三大定律。现在，软件也许并没有揭示类似的规律性原理，但是解释为什么没有的重担却落在了你的身上。我不是迟钝和好辩的。我相信有一天软件的“复杂性”将以某种更高级的规律性概念来理解(就像物理学家的不变式)。

我并没有着手于 Lukasik 提倡的更深层次的分析。作为一门学科，我们需要更广泛的信息理论，它能够量化静态结构的信息内容，就像针对交互流的香农信息论一样。这已经超越了我的能力。作为对 Lukasik 的简单回应，我认为系统复杂性是无数细节的函数，这些细节必须精确

而且详细地说明——或者是借助某种通用规则，或者是逐一阐述，但决不仅仅是统计说明。仅靠若干人不相干的工作，是不大可能产生能用通用规律进行精确描述的足够的一致性的。

不过，很多软件结构的复杂性并不完全是因为与外部世界保持一致，而是因为实现的本身，例如数据结构、算法和互联性等。而在更高的级别开发(发展)软件，使用其他人的成果，或者重用自己的程序——都能避免面对整个层次的复杂性。“没有银弹”提出了全力解决复杂性问题的方法，这种方法可以在现实中取得十分乐观的进展。它倡导向软件系统增加必要的复杂性：

- 层次化，通过分层的模块或者对象；
- 增量化，使系统可以持续地运行。

Harel 的分析

David Harel，在 1992 年的论文“批评银弹”(Biting the Silver Bullet)中，对已发表的“没有银弹”进行了很多最仔细的分析。[9]

悲观主义、乐观主义与现实主义。Harel 同时阅读了“没有银弹”和 1984 年 Parnas 的文章“战略防卫系统的软件问题”(Software Aspects of Strategic Defense Systems[10])，认为它们“太过黯淡”。因此，他试图在论文“走向系统开发的光明未来”(Toward a Brighter Future for System Development)中展现其明亮的一面。Cox 同 Harel 一样认为“没有银弹”一文过于悲观，他提出，“如果从一个新视点去观察相同的事情，你会得到一个更加乐观的结论。”他们的论调都有一些问题。

首先，我的妻子、同事和编辑发现我犯乐观主义错误的几率远远大于悲观主义。毕竟，我的从业背景是程序员，乐观主义是这个行业的职业病。

“没有银弹”一文明确地指出：“我们看看近 10 年来的情况，没有银弹的踪迹……怀疑论者并不是悲观主义者……虽然没有通天大道，但是路就在脚下。”它预言，如果 1986 年的很多创新能持续开拓和发展，实际上它们的共同作用就能使生产率获得数量级的提高。从 1986 年到 1996 年，已经 10 年过去了，这个预言即使说明了什么，那也是过于乐观，而不是过于悲观。

就算“没有银弹”总体看来有些悲观，那么到底存在什么问题呢？爱因斯坦关于任何物体运动的速度无法超过光速的论断是否过于“黯淡”或者“令人沮丧”？那么戈德尔关于某些事物无法计算的结论，又如何呢？“没有银弹”一文认为，“软件的特性本身导致了不大可能有任何的银弹。”Turski 在 IFIP 大会上发表了一篇论文作为出色的回应，文中指出：

在所有被误导的科学探索中，最悲惨的莫过于对一种能够将一般金属变成金子的物质，即点金石的研究。这个由统治者不断地投入金钱，被一代代的研究者不懈追求的、炼金术中至高无上的法宝，是一种从理想化想象和普遍假设中——以为事情会像我们所认为的那样——提取出的精华。它是人类纯粹信仰的体现，人们花费大量的时间和精力来认可和接受这个无法解决的问题。即使被证明是不存在的，那种寻找出路和希望能一劳永逸的愿望，依然十分强烈。而我们中的绝大多数总是很同情这些明知不可为而为之的人的勇气，因此它们总是能得以延续。所以，将圆形变方的论文被发表，抑制脱发的洗液被研制和出售，提高软件生产率的方法被提出并成功地推销。

我们太倾向于遵循我们自己的乐观主义(或者是发掘我们出资人的

乐观主义)。我们太喜欢忽视真理的声音，而去听从万灵药贩卖者的诱惑了。[11]

我和 Turski 都坚持认为这个白日梦限制了发展，浪费了精力。

“消极”主题。Harel 认识到“没有银弹”中的消极来自三个主题：

- 根本和次要问题的清晰划分；
- 独立地评价每个候选银弹；
- 仅仅预言了 10 年，而不是期望在足够长的时间内出现任何重大的进步。

第一个主题是整篇文章的主要观点。我仍然认为上述划分对于理解为什么软件难以开发的原因是绝对关键的。对于应该做出哪些方面的改进，它也是十分明确的指南。

至于独立地考虑不同的候选银弹，“没有银弹”并非如此。各种各样的技术一个接一个地被提出，每一种都过分宣扬自身的效果。因此，依次独立地评估它们是非常公平的。我并不是对这些技术持反对态度，而是反对那种期望它们能起到魔术般作用的观点。Glass、Vessey 和 Conger 1992 年在他们的论文中提供了充足的证据，指出对银弹的无谓研究仍未结束。[12]

关于选择 10 年还是 40 年作为预言的期限，选择较短的时间就等于承认我们并没有足够强的能力可以预见到 10 年以后的事情。我们中间有谁可以在 1975 年预见到 20 世纪 80 年代的微型计算机革命呢？

对于 10 年的期限，还有其他的一些原因：各种银弹都宣称它们能够立刻取得效果。我回顾了一下，发现没有任何一种银弹声称：“向我的秘方投资，在 10 年后你将获得成功”。另外，硬件的性价比可能每 10

年就会有成百倍的增长，尽管这种比较不是很合适，但是直觉上的确如此。我们确信会在下一个 40 年中取得稳步的发展。不过，以 40 年为代价取得数量级的进展，很难被认为是不可思议的进步。

Harel 想象的试验。Harel 建议了一种想象的试验，他假设“没有银弹”是发表在 1952 年，而不是 1986 年，不过表达的论断相同。他使用**反证法**来证明将根本和次要问题分开是不恰当的。

这种观点站不住脚。首先，“没有银弹”一开始就声称，20 世纪 50 年代的程序开发中曾占支配地位的次要困难如今已经不存在了，并且消除这些困难已经得到了数量级的改善。将辩论推回到 40 年前是不合理的，在 1952 年，甚至很难想象开发的次要问题不会占据开发工作的主要部分。

其次，Harel 所设想的 20 世纪 50 年代行业所处状态是不准确的：

> 当时已经不是构建大型复杂系统的时代，程序员的工作模式已经成为只需执行有限算法工作的常规个人程序的开发(在现代的编程语言中，大概是 100～200 行代码)。在已有技术和方法学的前提下，这些任务是令人恐怖的，处处都是错误、故障以及延误完成时间。

接着，他阐述了在传统的小型个人程序中，那些假设的错误、故障以及延误完成时间如何在接下来的 25 年中得到数量级的改进和提高。

事实上，20 世纪 50 年代该领域并不是小型个人程序的天下。在 1952 年，Univac 还在使用大约 8 人开发的复杂程序处理 1950 年的人口普查。[13]其他机器则用于化学动力学、中子漫射计算和导弹性能计算等。[14]汇编语言、重定位的链接和装载程序、浮点解释系统等，还经常被使用。[15]1955 年，人们开发 50～100 人-年的商用程序。[16]1956 年，通用电气在路易斯维尔的设备工厂使用着超过 80 000 指令的薪资系统。1957 年，SAGE ANFSQ/7 防空计算机系统已经运转了两年，这个系统分布在 30

个不同的地点，是拥有 75 000 条语句的基于通讯、自消除故障的热备实时系统。[17]因此，几乎无法坚持说个人程序的技术革命，能够用来描述 1952 年以来的软件工程上的努力。

银弹就在这里。Harel 接着提出了他自己的银弹，一种称为“香草(Vanilla)框架”的建模技术。文中并没有对该方法提供足够评估的详细描述，不过给出了一些论文和参考资料。[18]建模所针对的确实是软件开发的根本困难，即概念性要素的设计和调试，因此香草框架有可能是革命性的。我也希望如此。Ken Brooks 在报告中提到，其在实际工作中应用时的确是一种颇有帮助的方法学。

不可见性。Harel 强烈地主张软件的许多概念性要素本质上是拓扑的，这些关系可以用空间/图形方式来表达：

> 使用适当的可视化图形可以给工程师和程序员带来可观的成效。而且，这种效果并不仅仅局限于次要问题，开发人员思考和探索的质量也得到了改进。未来的成功系统的开发将围绕着可视化图形的表达方式。首先，我们会使用“合适的”实体和关系来形成概念，然后将它表达成一系列逐步完善的模型，不断系统化地阐明和再阐明概念。模型用若干可视化语言的适当组合来描述，它必须是多种语言的组合，因为系统模型具有若干方面的内容，每方面像变戏法般产生不同类型的思维图像。
>
> ……就使自己成为良好可视化的表达方式而言，建模过程的某些方面并不会立刻出现改观。例如，变量和数据结构上的算法操作可能还会采用文字性描述。

我和 Harel 的观点颇为一致。我认为软件要素并不存在于三维空间中，因此并不存在概念性设计到图形的简单映射，无论是两维还是更多维。他承认，我也同意——这需要多种图形，每种图形覆盖某个特定的方面，而且有些方面无法用图形来表达。

Harel 采用图形来辅助思考和设计的热情彻底地感染了我。我一直喜欢向准程序员提问：“下个 11 月在哪儿？”如果觉得问题过于模糊，接着我会问：“告诉我，你自己关于时间历法的模型。”优秀程序员具有很强的空间想象能力，他们常常有时间的几何模型，而且通常无需考虑，就能理解第一个问题。他们往往拥有高度个性化的模型。

Jones 的观点——质量带来生产率

Capers Jones 最开始在一系列备忘录里，而后在一本书里，提出了颇有洞察力的观点。很多和我有书信往来的人向我提到了他的观点，“没有银弹”如同当时的很多文章，关注于**生产率**——单位输入对应的软件产出。Jones 提出：“不。关注**质量**，生产率自然会随之提高。”[19] 他认为，很多代价高昂的后续项目投入了大量的时间和精力来寻找和修复规格说明、设计和实现上的错误。他提供的数据显示了缺乏系统化质量控制和进度灾难之间的密切关系。我认同这些数据。不过，Boehm 指出，如果一味地追求完美质量，生产率就会像 IBM 的航天飞机软件一样再次下降。

Coqui 也提出相似的主张：系统化软件开发方法的发展是为了解决质量问题(特别是避免大型的灾难)，而不是出于生产率方面的考虑。

但是注意：20 世纪 70 年代，在软件生产上应用工程原理的目标是提高软件产品的质量、可测试性、稳定性以及可预见性——而不是软件产品的开发效率。

在软件生产上应用工程原理的驱动力是其担心因拥有无法控制的“艺术家们”而可能导致巨大的灾难，他们往往对异常复杂系统的开发承担责任。[20]

那么，生产率的情形如何

生产率数据。生产率数据非常难以定义、测量和寻找。Capers Jones相信两个相隔10年、完全等同的COBOL程序，一个采用结构化方法开发，另一个不使用结构化方法，它们之间的差距是3倍。

Ed Yourdon说："由于工作站和软件工具，我看到人们的工作获得了5倍的提高。"Tom DeMarco认为："你的期望——10年内，由于所有的技术而使生产率得到数量级的提高——太乐观了。我没有看到任何机构取得数量级的进步。"

塑料薄膜包装的成品软件——购买，而非开发。我认为，1986年"没有银弹"中的一个估计被证实是正确的："我相信，这个大众市场是……软件工程领域意义最深远的开发方向。"从学科的角度说，不管与内部还是外部客户软件的开发相比，大众市场软件都几乎是一个崭新的领域。软件包的销量一旦达到百万或者即使只是几千，这时关键的支配性问题就变成了质量、时机、产品性能和支持成本，而不再是对于客户系统异常关键的开发成本。

创造性活动的强大工具。提高信息管理系统(MIS)编程人员生产率最戏剧化的方法是到一家计算机商店去，购买理应由他们开发的商业成品。这并不荒唐可笑。价格低廉、功能强大的薄膜包装软件已经能满足要求，而以前这会要求进行定制软件包的开发。与复杂的大型产品工具相比，它们更加像电锯、电钻和砂磨机。把它们组合成兼容互补的集合，像Microsoft Works和集成更好的Claris Works一样，能够带来巨大的灵活性。另外，像供人们使用的组合工具箱，其中的某些工具会经常被使用。这种工具必须注重使用时的方便，而不是专业。

Ivan Selin，美国管理系统公司主席，在1987年曾写信给我：

我有些怀疑你的关于软件包没有真正地改变很多……的观点。我觉得你太过轻易地抛开了你的观察所蕴涵的事实；你观察到——[软件包]“可能比以前更加通用和容易定制一些，但并不太多。”即使我表面上接受了这种论述，但是我仍然相信用户会察觉到软件包更加通用和易于定制化，这种感觉也使用户更容易接受软件包。在我公司所发现的大多数情况中，是[最终]用户，而不是软件人员不愿意使用软件包，因为他们认为会失去必要的特性或功能。因此，对他们而言，易于定制是一个非常大的卖点。

我认为 Selin 是十分正确的——我低估了软件包客户化的程度和它的重要性。

面向对象编程——这颗铜质子弹可以吗

使用更大的零件来构建。本章开始的描述提醒我们，当很多零件需要装配，而且每个零件可能很复杂时，如果它们的接口设计得很流畅，大量丰富的结构就能快速地组合在一起。

面向对象编程的第一个特征是，它强制的**模块化**和清晰的接口。其次，它强调了封装，即外界无法看到组件的内部结构；它还强调了**继承**和伴随着的**层次化**类结构以及虚函数。面向对象还**强调了强抽象数据类型化**，它确保某种特定的数据类型只能由自身的相应函数来操作。

现在，无需使用整个 Smalltalk 或者 C++的软件包，就可以获得这些特点中的任意一个——其中一些甚至出现在面向对象技术之前。面向对象方法吸引人的地方类似于复合维他命药丸：一次性(即编程人员的再培训)得到所有的好处。面向对象是一种非常有前途的概念。

面向对象技术为什么发展缓慢？“没有银弹”发表后的 9 年中，对

面向对象技术的期望稳步增长。为什么增长得如此缓慢呢？是因为理论。James Coggins 已经在《C++的报告》(*The C++ Report*)做了 4 年“The Best of comp.lang.C++”专栏的作者，他进行了这样的解释：

问题是 OO 的程序员经历了很多错综复杂混乱的应用，他们所关注的是低层次的抽象，而不是高层次的抽象。例如，他们开发了很多像链表或集合这样的类，而不是**用户接口、射线束**或者**有限元素模型**。遗憾的是，C++中帮助程序员避免错误的强类型检查，使得从小型事物中构建大型物体变得非常困难。[21]

他回归到基本的软件问题，主张一种解决软件不能满足要求的方法，即通过客户的参与和协作来提高脑力劳动的规模。他赞同自上而下的设计：

如果我们设计大粒度的类，关注用户已经接触的概念，则在进行设计的时候，他们能够理解设计并提出问题，并且可以帮助设计测试用例。我的眼科客户并不关心堆栈，他们关心描述眼角膜形状的勒让德多项式。在这方面，小规模的封装带来的好处比较少。

David Parnas 的论文是面向对象概念的起源之一，他用不同的观点看这个问题。他写信给我：

答案很简单。因为 OO 和各种复杂语言的联系已经很紧密。人们并没有被告诉 OO 是一种设计的方法，并向他们讲授设计方法和原理，大家只是被告知 OO 是一种特殊工具。而我们可以用任何工具写出优质或低劣的代码。除非我们给人们讲解如何设计，否则语言所起的作用非常小。结果人们使用这种语言做出不好的设计，没有从中获得多少价值。而一旦获得的价值太少，它就不会流行。

先行投入资金，后期获得收益。面向对象技术包含了很多方法学上的进步。面向对象技术的前期投入很多——主要是重新培训程序员用很

新的方法进行思考，同时还要把函数打造成通用的类。我认为它的优势是客观存在的，并非仅仅是推测。面向对象在整个开发周期中得到了应用，但是真正的获益只有在后续开发、扩展和维护活动中才能体现出来。Coggins 说：“面向对象技术不会加快首次或第二次的开发，产品族中第五个项目的开发将会异乎寻常的迅速。”[22]

为了预期中的、但有些不确定的收益，冒着风险投入资金是投资人每天在做的事情。不过，在很多软件公司，这需要真正的管理勇气，一种比技术竞争力或者优秀管理能力更少有的精神。我认为极度的前期投入和收益的推后是使 OO 技术应用迟缓的最大原因。即使如此，在很多机构，C++仍毫无疑问地取代了 C。

重用的情况怎样

解决软件构建根本困难的最佳方法是不进行任何开发。软件包只是实现上述目标的方法之一，另外的方法是程序重用。实际上，易于重用类和通过继承方便地定制是面向对象技术最吸引人的地方。

事情常常就是这样。当某人在新的做事方法上取得了一些经验，新模式就不再像一开始那么简单了。

当然，程序员经常重用他们自己的手头工作。Jones 提到：

大多数有丰富经验的程序员都拥有自己的私人开发库，使用大约30%的重用代码来开发软件。公司级别的重用能提供 70%的重用代码量，它需要特殊的开发库和管理支持。公司级别的重用代码也意味着需要对项目中的变更进行统计和度量，从而提高重用的可信程度。[23]

W. Huang 建议用责任专家的矩阵管理来组织软件工厂，培养每个人重用自己的代码的日常工作习惯。[24]

JPL 的 Van Snyder 向我指出，数学软件领域有着软件重用的长期传统：

我们推测重用的障碍不在生产者一边，而在消费者一边。如果一个软件工程师，潜在的标准化软件构件消费者，觉得寻找能满足他需要的构件进行验证比自行编写的代价更加昂贵时，重复的构件就会产生。注意我们上面提到的“觉得”。它和重新开发的真正投入无关。

数学软件上重用成功的原因有两个：(1)它很晦涩难懂，每行代码需要大量高智商的输入；(2)存在丰富的标准术语，也就是用数学来描述每个构件的功能。因此，重新开发数学软件构件的成本很高，而查找现有构件功能的成本很低。数学软件界存在一些长期的传统——例如，专业期刊出版和搜集算法，用适度成本提供算法，出于商业考虑开发的高质量算法(尽管成本有些高，但依旧适度)等——使查找和发现满足某人需要的构件比其他的很多领域都要容易。其他领域有时甚至不可能简洁地提出明确的要求。这些因素合在一起，使数学软件的重用比重新开发更有吸引力。

同样的原因，在很多其他领域中也可以发现相同的重用现象，如那些为核反应、天气模型、海洋模型开发软件的代码编制工作。这些领域都是在相同的课本和标准概念下逐步发展起来的。

现在公司级别的重用情况如何？在这方面有大量的研究。美国国内的实践相对较少，有报道声称在国外重用较多。[25]

Jones 报告，在他公司的客户中，所有拥有 5 000 名以上程序员的机构都进行正式的重用研究，而拥有 500 名以下程序员的组织，只有不到 10%着手重用研究。[26] 报告指出，最具有重用潜质的企业中，重用性

研究(而非部署)“是活跃和积极的，即使没有完全成功”。Ed Yourdon 报告，有一家马尼拉的软件公司，200 名程序员中有 50 名从事供其他人使用的重用模块的开发，“我所见到的个案非常少——是由于诸如奖励结构等机构上的因素而进行重用研究，而不是技术上的原因。”

DeMarco 告诉我，大众市场软件包提供了数据库系统等通用功能，充分地减轻了压力，减少了处在重用模块边缘的开发。“不管怎样，重用的模块一般是一些通用功能。”

Parnas 写道：

重用是一件说起来容易，做起来难的事情。它同时需要良好的设计和卓越的文档。即使我们看到了非常罕见的优秀设计，但如果没有好的文档，我们也不会看到能重用的构件。

关于通用化的可行性，Ken Brooks 也指出了一些当中存在的困难：“即使在第五次使用我自己的个人用户界面库的时候，我还是在不断地进行修改。”

真正的重用似乎才刚刚开始。Jones 报告，在开放市场上仅有少量的重用代码模块，它们的价格在常规开发成本的 1%～20%之间。[27] DeMacro 说：

对整个重用现象，我变得有些气馁。对于重用，现有理论几乎是整体缺乏。时间证明了要使模块能够重用，其成本非常高。

Yourdon 估计了这个高昂的费用：“一个良好的经验法则是，可重用的构件的工作量是‘一次性’构件的两倍。”[28]在第 1 章的讨论中，我观察到了真正产品化构件所需的成本。因此，我对工作量比率的估计是三倍。

显然，我们正在看到很多重用的形式和变化，但离我们所期望的还较远，还有很多需要学习的地方。

学习大量的词汇——对软件重用的一个可预见但还没有被预言的问题

思索的层次越高，所需要处理的基本思考要素也就越多。因此，编程语言比机器语言更加复杂，而自然语言的复杂程度更高。高级语言有更广泛的词汇量、更复杂的语法以及更加丰富的语义。

作为一个科目，我们并没有就程序重用的实际情况，仔细考虑它蕴涵的意义。为了提高质量和生产率，我们需要通过经过调试的大型要素来构建系统。在编程语言中，这些函数的级别远远高于语句。所以，无论采用对象类库还是程序库的方式，我们必须面对编程词汇规模彻底扩大的事实。对于重用，词汇学习并不是思维障碍中的一小部分。

现在人们拥有成员超过 3 000 个的类库。很多对象需要 10～20 个参数和可选变量的说明。如果想获得所有潜在的重用，任何使用类库编程的人员必须学习其成员的语法(外部接口)和语义(详细的功能行为)。

这项工作并不是没有希望的。一般人日常使用的词汇超过了 10 000 个，受过教育的人远多于这个数目。另外，我们在自然而然地学习着语法和非常微妙的语义。我们可以正确地区分**巨大**、**大**、**辽阔**、**大量**和**庞大**。人们不会说：庞大的沙漠或者辽阔的大象。

对软件重用问题，我们需要研究适当的学问，了解人们如何拥有语言。一些经验教训是显而易见的：

- 人们在上下文中学习，因此我们需要出版一些复合产品的例子，而不仅仅是零部件的库；
- 人们只会记忆背诵单词，而语法和语义是在上下文中通过使用逐渐地学习的；
- 人们根据语义的分类对词汇组合规则进行分组，而不是通过比较对象子集。

子弹的本质——形势没有发生改变

现在，我们回到基本问题。复杂性是我们这个行业的属性，而且复杂性是我们主要的限制。R. L. Glass 在 1988 年精确地总结了我在 1995 年的看法：

又怎么样呢？Parnas 和 Brooks 不是已经告诉我们了吗？软件开发是一件棘手的事情，前方并不会有魔术般的解决方案。现在是从业者研究和分析革命性进展的时候，而不是等待或希望它出现。

软件领域中的一些人发现这是一幅使人泄气的图画。他们是那些依然认为突破近在眼前的人们。

但是我们中的一些——那些非常固执，以至于认为我们是现实主义者的人——把它看成是清新的空气。我们终于可以将焦点集中在更加可行的事情上，而不是空中的馅饼。现在，有可能，我们可以在软件生产率上取得逐步的进展，而不是等待不大可能到来的突破。[29]

CHAPTER 18

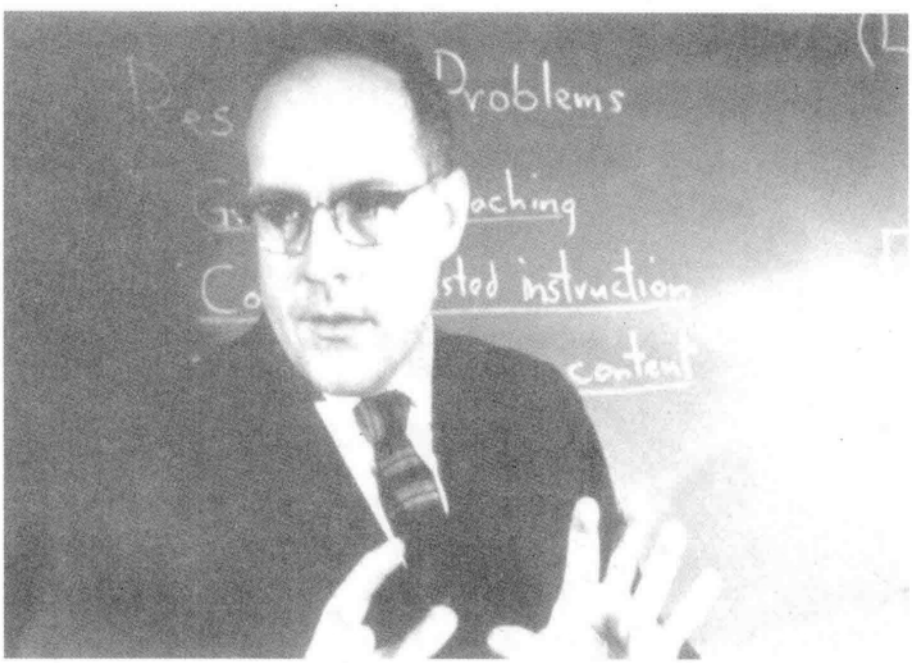
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 18 章

《人月神话》的观点：是与非

Propositions of The Mythical Man-Month: True or False



1967 年布鲁克斯正在提出预言

资料来源：Photo by J. Alex Langley for *Fortune Magazine*

我们理解也好，不理解也好，描述都应该简短精练。

——塞缪尔·勃特勒，《休迪布拉斯》

For brevity is very good,

Where we are, or are not understood.

SAMUEL BUTLER, *Hudibras*

现在我们对软件工程的了解比1975年要多得多了。那么在1975年版本的人月神话中，哪些观点得到了数据和经验的支持？哪些观点被证明是不正确的？又有哪些观点随着世界的变化，显得陈旧过时了呢？为了帮助判断，我将1975年书籍中的论断毫无更改地抽取出来，以摘要的形式列举在下面——我认为是正确的：是客观事实和经验中推广的法则。你也许会问："如果这些就是那本书讲的所有东西，为什么要花这么长的篇幅来论述呢？"方括号中的评论是新版的新增内容。

所有这些观点都是可操作验证的，我以刻板的形式表达是希望能引起读者的思考、判断和讨论。

第1章　焦油坑

1. 编程系统产品(Programming Systems Product)开发的工作量是供个人使用的、独立开发的构件程序的9倍。我估计软件构件产品化引起了3倍工作量，将软件构件整合成完整系统所需要的设计、集成和测试又强加了3倍的工作量，这些高成本的构件在根本上是相互独立的。

2. 编程行业"满足我们内心深处的创造渴望和愉悦所有人的共有情感"，其提供了五种乐趣：

- 创建事物的快乐；
- 开发对其他人有用的东西的乐趣；
- 将可以活动、相互啮合的零部件组装成类似迷宫的东西，这个过程所体现出令人神魂颠倒的魅力；
- 面对不重复的任务，不断学习的乐趣；

- 工作在如此易于驾驭的介质上的乐趣——纯粹的思维活动——其存在、移动和运转方式完全不同于实际物体。

3. 同样，这个行业具有一些内在固有的苦恼：

- 将做事方式调整到追求完美是学习编程的最困难部分；
- 由其他人来设定目标，并且必须依靠自己无法控制的事物(特别是程序)；权威不等同于责任；
- 实际情况看起来要比这点好一些：真正的权威来自于每次任务的完成；
- 任何创造性活动都伴随着枯燥艰苦的劳动，编程也不例外；
- 人们通常期望项目在接近结束时，软件项目能收敛得快一些，然而，情况却是越接近完成，收敛得越慢；
- 产品在完成前总面临着陈旧过时的威胁；只有实际需要时，才会用到最新的设想。

第 2 章　人月神话

1. 缺乏合理的时间进度是造成项目滞后的最主要原因，它比其他所有因素的总和影响还大。

2. 良好的烹饪需要时间，某些任务无法在不损害结果的情况下加快速度。

3. 所有的编程人员都是乐观主义者：“一切都将运作良好”。

4. 由于编程人员通过纯粹的思维活动来开发，我们期待在实现过程

中不会碰到困难。

5. 但是，我们的构思本身是有缺陷的，因此总会有 bug。

6. 围绕着成本核算的估计技术，混淆了工作量和项目进展。**人月是危险和带有欺骗性的神话，因为它暗示人员数量和时间是可以相互替换的。**

7. 在若干人员中分解任务会引发额外的沟通工作量——培训和相互沟通。

8. 关于进度安排，我的经验是为 1/3 计划、1/6 编码、1/4 构件测试以及 1/4 系统测试。

9. 作为一门学科，我们缺乏数据估计。

10. 我们对自己的估计技术不确定，因此在管理和客户的压力下，我们常常缺乏坚持的勇气。

11. Brooks 法则：为进度落后的项目增加人手，只会使进度更加落后。

12. 向软件项目中增派人手从三个方面增加了项目必要的总体工作量：任务重新分配本身和所造成的工作中断；培训新人员；额外的相互沟通。

第 3 章　外科手术队伍

1. 同样有两年经验而且在受到同样培训的情况下，优秀的专业程序员的生产率是较差的程序员的 10 倍(Sackman、Erickson 和 Grant)。

2. Sackman、Erickson 和 Grant 的数据显示，经验和实际表现之间没有相互联系，我怀疑这种现象是否普遍成立。

3. 小型、精干队伍是最好的——思绪尽可能少。

4. 两个人的团队，其中一个是领导者，常常是最佳的人员使用方法(留意一下上帝对婚姻的设计)。

5. 对于真正意义上的大型系统，小型精干的队伍太慢了。

6. 实际上，绝大多数大型编程系统的经验显示，一拥而上的开发方法是高成本、速度缓慢、低效的，开发出的产品无法进行概念上的集成。

7. 一位首席程序员、类似于外科手术队伍的团队架构提供了一种方法——既能获得由少数头脑产生的产品完整性，又能得到多位协助人员的总体生产率，还彻底地减少了沟通的工作量。

第 4 章　贵族专制、民主政治和系统设计

1. “概念完整性是系统设计中最重要的考虑因素。”

2. “功能与理解上的复杂程度的比值才是系统设计的最终测试标准”，而不仅仅是丰富的功能。(该比值是对易用性的一种测量，由简单和复杂应用共同验证。)

3. 为了获得概念完整性，设计必须由一个人或者具有共识的小型团队来完成。

4. “对于非常大型的项目，将体系结构方面的工作与具体实现相分离是获得概念完整性的强有力方法。”(其同样适用于小型项目。)

5. “如果要得到系统概念上的完整性，就必须有人控制这些概念。这实际上是一种无需任何歉意的贵族专制统治。”

6. 纪律、规则对行业是有益的。外部的体系结构规定实际上是增强，而不是限制实现小组的创造性。

7. 概念上统一的系统能更快地开发和测试。

8. 体系结构(architecture)、设计实现(implementation)、物理实现(realization)的许多工作可以并行。(软件和硬件设计同样可以并行。)

第5章　画蛇添足

1. 尽早交流和持续沟通能使结构师有较好的成本意识，使开发人员获得对设计的信心，并且不会混淆各自的责任分工。

2. 结构师如何成功地影响实现：

- 牢记是开发人员承担创造性的实现责任；结构师只能提出建议。
- 时刻准备着为所指定的说明建议一种实现的方法，准备接受任何其他可行的方法。
- 对上述建议保持低调和平静。
- 准备对所建议的改进放弃坚持。
- 听取开发人员在体系结构上改进的建议。

3. 第二个系统是人们所设计的最危险的系统，通常的倾向是过分地进行设计。

4. OS/360 是典型的画蛇添足的例子。(Windows NT 似乎是 20 世纪 90 年代的例子。)

5. 为功能分配一个字节和微秒的优先权值是一个很有价值的规范化方法。

第 6 章　贯彻执行

1. 即使是大型的设计团队，设计结果也必须由一个或两个人来完成，以确保这些决定是一致的。

2. 必须明确定义体系结构中与先前定义不同的地方，重新定义的详细程度应该与原先的说明一致。

3. 出于精确性的考虑，我们需要形式化地设计定义；同样，我们需要记叙性定义来加深理解。

4. 必须采用形式化定义和记叙性定义中的一种作为标准，另一种作为辅助措施；它们都可以作为表达的标准。

5. 设计实现，包括模拟仿真，可以充当一种体系结构的定义；这种方法有一些严重的缺点。

6. 直接整合是一种强制推行软件的结构性标准的方法。(硬件上也是如此——考虑内建在 ROM 中的 Mac WIMP 接口。)

7. “如果起初至少有两种以上的实现，(体系结构)定义会更加整洁和规范。”

8. 允许体系结构师对实现人员的询问做出电话应答解释是非常重要的，并且必须进行日志记录和整理发布。(电子邮件是现在可选

的介质。)

9. “项目经理最好的朋友就是他每天要面对的对手——独立的产品测试机构/小组。”

第7章　为什么巴比伦塔会失败

1. 巴比伦塔项目的失败是因为缺乏交流以及交流的结果——**组织**。

交流

2. “因为左手不知道右手在做什么，从而进度灾难、功能的不合理和系统缺陷纷纷出现。”由于存在对其他人的各种假设，团队成员之间的理解开始出现偏差。

3. 团队应该以尽可能多的方式进行相互之间的交流：非正式地进行简要技术陈述的常规项目会议，共享的正式项目工作手册[以及通过电子邮件]。

项目工作手册

4. 项目工作手册“不是独立的一篇文档，它是对项目必须产生的一系列文档进行组织的一种结构”。

5. “项目**所有**的文档都必须是该[工作手册]结构的一部分。”

6. 需要**尽早**和**仔细**地设计工作手册结构。

7. 事先制定良好结构的工作手册“可以将后来书写的文字放置在合适的章节中”，并且可以提高产品手册的质量。

8. “**每**一个团队成员应该了解**所有**的材料(工作手册)。”(我现在想说的是，每个团队成员应该能够看到所有材料，网页即可满足要求。)

9. 实时更新是至关重要的。

10. 工作手册的使用者应该将注意力集中在上次阅读后的变更以及关于这些变更重要性的评述上。

11. OS/360 项目工作手册开始采用的是纸介质，后来换成了微缩胶片。

12. 今天(即使在 1975 年)，共享的电子手册是能达到所有这些目标的、更好的、更加低廉的、更加简单的机制。

13. 仍然需要用变更条和修订日期[或具备同等功能的方法]来标记文字；仍然需要后进先出(LIFO)的电子化变更小结。

14. Parnas 强烈地认为使每个人看到每件事的目标是**完全错误**的；各个部分应该被封装，从而没有人需要或者被允许看到其他部分的内部结构，只需要了解接口。

15. Parnas 的建议的确是灾难的处方。(**Parnas 让我认可了该观点，使我彻底地改变了想法。**)

组织架构

16. 团队组织的目标是为了减少必要的交流和协作量。

17. 为了减少交流，组织结构包括了**人力划分**(division of labor)和**限定职责范围**(specialization of function)。

18. 传统的树状组织结构反映了**权力**的结构原理——不允许双重领导。

19. 组织中的**交流**是网状，而不是**树状结构**，因此所有的特殊组织机制(往往体现为组织结构图中的虚线部分)都是为了进行调整，以克服树状组织结构中交流缺乏的困难。

20. 每个子项目具有两个领导角色——**产品负责人，技术主管**或**结构师**。这两个角色的职能有很大的区别，需要不同的技能。

21. 两种角色的任意组合都可以是非常有效的：

- 产品负责人和技术主管是同一个人；
- 产品负责人作为总指挥，技术主管充当其左右手；
- 技术主管作为总指挥，产品负责人充当其左右手。

第 8 章　胸有成竹

1. 仅仅通过对编码部分时间的估计，然后乘以其他部分的相对系数，是无法得出对整项工作的精确估计的。

2. 构建独立小型程序的数据不适用于编程系统项目。

3. 程序开发随程序规模的大量增长而增长。

4. 一些发表的研究报告显示，指数约为 1.5。(**Boehm 的数据并不完全一致，在 1.05 和 1.2 之间变化**[1]。)

5. Portman 的 ICL 数据显示，相对于其他活动，全职程序员仅将 50% 的时间用于编程和调试。

6. Aron 的 IBM 的数据显示，生产率是系统各个部分交互的函数，在 1.5 千代码行/人-年至 10 千代码行/人-年的范围内变化。

7. Harr 的 Bell 实验室数据显示，对于已完成的产品，操作系统类的生产率大约是 0.6 千 LOC/人-年，编译类工作的生产率大约为 2.2 千 LOC/人-年。

8. Brooks 的 OS/360 数据与 Harr 的数据一致：操作系统 0.6～0.8 千 LOC/人-年，编译器 2～3 千 LOC/人-年。

9. Corbató的 MIT 项目 MULTICS 数据显示，在操作系统和编译器混合类型上的生产率是 1.2 千 LOC/人-年，但这些是 PL/I 的代码行，而其他所有的数据是汇编代码行。

10. 在基本语句级别，生产率看上去是一个常数。

11. 当使用适当的高级语言时，程序编制的生产率可以提高 5 倍。

第 9 章　削足适履

1. 除了运行时间以外，程序所占据的**内存空间**也是主要开销。特别是对于操作系统，它的很多程序是永久驻留在内存中的。

2. 即便如此，花费在驻留程序所占据内存上的金钱仍是物有所值的，比其他任何在配置上投资的效果都要好。规模本身不是坏事，但不必要的规模是不可取的。

3. 软件开发人员必须设立规模目标，控制规模，发明一些减少规模的方法——就如同硬件开发人员为减少元器件所做的事情一样。

4. 规模预算不仅仅在占据内存方面是明确的，同时还应该指明程序对磁盘的访问次数。

5. 规模预算必须与分配的功能相关联；在指明模块大小的同时，确切定义模块的功能。

6. 在大型团队中，各个小组倾向于不断地局部优化，以满足自己的目标，而较少考虑对用户的整体影响。这种方向性的问题是大型项目的

主要危险。

7. 在整个实现的过程期间，系统结构师必须保持持续的警觉，确保连贯的系统完整性。

8. 从系统整体出发以及面向用户的态度是软件编程管理人员最重要的职能。

9. 在早期应该制定策略，以决定用户可选项目的粗细程度，因为将它们作为整体打包能够节省内存空间(常常还可以节约市场成本)。

10. 暂存区空间的尺寸以及每次磁盘访问的程序数量是很关键的决策，因为性能是规模的非线性函数。(这个整体决策已显得过时——起初是由于虚拟内存，后来则是成本低廉的内存。现在的用户通常会购买能容纳主要应用程序所有代码的内存。)

11. 为了取得良好的空间—时间折衷，开发队伍需要得到特定的某种语言或者机型的编程技能培训，特别是在使用新语言或者新机器时。

12. 编程需要技术积累，每个项目需要自己的标准组件库。

13. 库中的每个组件需要有两个版本，运行速度较快和短小精炼的。(现在看来，这有些过时了。)

14. 精炼、充分和快速的程序往往是**战略性**突破的结果，而不仅仅是技巧上的提高。

15. 这种突破常常是一种新型算法。

16. 更普遍的是，战略上的突破常来自于对数据或表的重新表达。**数据的表现形式是编程的根本。**

第 10 章　提纲挈领

1. “前提：在一片文件的汪洋中，少数文档成为了关键的枢纽，每个项目管理的工作都围绕着它们运转。它们是经理们的主要个人工具。”

2. 对于计算机硬件开发项目，关键文档是目标、手册、进度、预算、组织机构图、空间分配以及机器本身的报价、预测和价格。

3. 对于大学科系，关键文档类似于目标、课程描述、学位要求、研究报告、课程表和课程的安排、预算、教室分配、教师和研究生助手的分配。

4. 对于软件项目，要求是相同的：目标、用户手册、内部文档、进度、预算、组织机构图和工作空间分配。

5. 因此，即使是小型项目，项目经理也应该在项目早期对上述的一系列文档进行规范化。

6. 以上集合中每一个文档的准备工作都将注意力集中在思索和对讨论的提炼上，而书写这项活动需要上百次的细小决定。正是由于它们的存在，人们才能从令人迷惑的现象中得到清晰、确定的策略。

7. 对每个关键文档的维护提供了状态监督和预警机制。

8. 每个文档本身就可以作为检查列表或者数据库。

9. 项目经理的基本职责是使每个人都向着相同的方向前进。

10. 项目经理的主要日常工作是沟通，而不是做出决定；文档使各项计划和决策在整个团队范围内得到交流。

11. 只有一小部分管理人员的时间——可能只有 20%——用来从自己头脑外部获取信息。

12. 出于这个原因，广受吹捧的市场概念——支持管理人员的“完全信息管理系统”并不基于反映管理人员行为的有效模型。

第 11 章　未雨绸缪

1. 化学工程师已经认识到无法一步将实验室工作台上的反应过程移到工厂中，需要一个试验性工厂(pilot plant)来为提高产量和在缺乏保护的环境下运作提供宝贵经验。

2. 对于编程产品而言，这样的中间步骤同样是必要的，但是软件工程师在着手发布产品之前，却并不会常规地进行试验性系统的现场测试。(现在，这已经成为了一项普遍的实践，beta 版本不同于有限功能的原型，alpha 版本同样是我所倡导的实践。)

3. 第一个开发的系统对于大多数项目并不合用。它可能太慢、太大，而且难以使用，或者三者兼而有之。

4. 系统的丢弃和重新设计可以一步完成，也可以一块块地实现，但这是**必须完成**的步骤。

5. 将开发的第一个系统——丢弃原型——发布给用户，可以获得时间，但是它的代价高昂——对于用户，使用极度痛苦；对于重新开发的人员，分散了精力；对于产品，影响了声誉，即使最好的再设计也难以挽回名声。

6. 因此，**为舍弃而计划，无论如何，你一定要这样做。**

7. “开发人员交付的是用户满意程度，而不仅仅是实际的产品。”(Cosgrove)

8. 用户的实际需要和用户感觉会随着程序的构建、测试和使用而变化。

9. 软件产品易于掌握的特性和不可见性，导致它的构建人员(特别容易)面临着永恒的需求变更。

10. 目标上(和开发策略上)的一些正常变化无可避免，事先为它们做准备总比假设它们不会出现要好得多。

11. 为变更而计划软件产品的技术，特别是拥有细致的模块接口文档的结构化编程广为人知，但并没有相同规模的实践。尽可能地使用表驱动技术同样是有所帮助的。(现在内存的成本和规模使这项技术越来越出众。)

12. 高级语言的使用、编译时操作、通过引用的声明整合和自文档技术能减少变更引起的错误。

13. 采用定义良好的数字化版本将变更量子(阶段)化(当今的标准实践。)

为变更计划组织架构

14. 程序员不愿意为设计书写文档，不仅仅是因为惰性，更多的是源于设计人员的踌躇—— 要为自己尝试性的设计决策进行辩解。(Cosgrove)

15. 为变更组建团队比为变更进行设计更加困难。

16. 只要管理人员和技术人才的天赋允许，老板必须对他们的能力培养给予极大的关注，使管理人员和技术人才具有互换性；特别是希望

在技术和管理角色之间自由地分配人手的时候。

17. 具有两条晋升线的高效组织机构存在着一些社会性的障碍，人们必须警惕并积极地同它做持续的斗争。

18. 很容易为不同的晋升线建立相互一致的薪水级别，但同等威信的建立需要一些强烈的心理措施：相同的办公室、一样的支持以及技术调动的优先补偿。

19. 组建外科手术队伍式的软件开发团队是对上述问题所有方面的彻底冲击。对于灵活组织架构问题，这的确是一个长期行之有效的解决方案。

前进两步，后退一步——程序维护

20. 程序维护基本上不同于硬件的维护；它主要由各种变更组成，如修复设计缺陷，新增功能，或者是使用环境或配置变换引起的调整。

21. 对于一个广泛使用的程序，其维护总成本通常是开发成本的40%或更多。

22. 维护成本受用户数目的影响。用户越多，所发现的错误也越多。

23. Campbell 指出了一个显示产品生命期中每月 bug 数的有趣曲线，其先是下降，然后上升。

24. 缺陷修复总会以 20%～50%的几率引入新的 bug。

25. 每次修复之后，必须重新运行先前所有的测试用例，确保系统不会以更隐蔽的方式被破坏。

26. 能消除、至少是能指明副作用的程序设计方法，对维护成本有很大的影响。

27. 同样，实现设计的人员越少、接口越少，产生的错误也就越少。

前进一步，后退一步——系统熵随时间增加

28. Lehman 和 Belady 发现，模块数量随大型操作系统(OS/360)版本号的增加呈线性增长，但是模块随版本号指数的增长而受到影响。

29. 所有修改都倾向于破坏系统的架构，增加了系统的混乱程度(熵)。即使是最熟练的软件维护工作，也只是延缓了系统退化到不可修复的混乱状态的进程，以致必须要重新进行设计。(许多程序升级的真正需要，如性能等，尤其会冲击它的内部结构边界。原有边界引发的不足常常在日后才会出现。)

第 12 章　干将莫邪

1. 项目经理应该制定一套策略，并为通用工具的开发分配资源；与此同时，他还必须意识到专业工具的需求。

2. 开发操作系统的队伍需要自己的目标机器，进行调试开发工作。相对于最快的速度而言，它更需要最大限度的内存，还需要安排一名系统程序员，以保证机器上的标准软件是及时更新和实时可用的。

3. 同时还需要配备调试机器或者软件，以便在调试过程中，所有类型的程序参数可以被自动计数和测量。

4. 目标机器的使用需求量是一种特殊曲线：刚开始使用率非常低，突然出现爆发性的增长，接着趋于平缓。

5. 同天文工作者一样，大部分系统调试工作总是在夜间完成。

6. 抛开理论不谈，一次分配给某个小组的连续的目标时间块被证明是最好的安排方法，比不同小组的穿插使用更为有效。

7. 尽管技术不断变化，这种采用时间块来安排匮乏计算机资源的方式仍能够延续 20 年(在 1975 年)，这是因为它的生产率最高。(在 1995 年依然如此。)

8. 如果目标机器是新产品，就需要一个目标机器的逻辑仿真装置。这样，可以**更快**地得到辅助调试平台。即使在真正机器出现之后，仿真装置仍可提供**可靠**的调试平台。

9. 主程序库应该被划分成：(1)一系列独立的私有开发库；(2)正处在系统测试下的系统集成子库；(3)发布版本。正式的分离和进度提供了控制。

10. 在编制程序的项目中，节省最大工作量的工具可能是文本编辑系统。

11. 系统文档中的巨大容量产生了新的不易理解问题[例如，看看 Unix]，但是它比大多数未能详细描述编程系统特性的短小文章更加可取。

12. 自上而下、彻底地开发一个性能仿真装置。尽可能早地开始这项工作，仔细地听取 “它们表达的意见”。

高级语言

13. 只有懒散和惰性会妨碍高级语言和交互式编程的广泛应用。(如今，它们已经在全世界使用。)

14. 高级语言不仅提高了生产率，还改进了调试：bug 更少，而且更容易寻找。

15. 传统的反对意见——功能、目标代码的尺寸、目标代码的速度，随着语言和编译器技术的进步已不再成为问题。

16. 现在可供系统编程合理选择的语言是 PL/I(不再正确)。

交互式编程

17. 某些应用上，批处理系统决不会被交互式系统所替代(依然成立)。

18. 调试是系统编程中较慢和较困难的部分，而漫长的调试周转时间是调试的祸根。

19. 有限的数据表明，系统软件开发中，交互式编程的生产率至少是原来的两倍。

第 13 章　整体部分

1. 第 4、5、6 章所意味的煞费苦心、详尽体系结构工作不但使产品更加易于使用，而且使开发更容易进行且 bug 更不容易产生。

2. Vyssotsky 提出，“许许多多的失败完全源于那些产品未精确定义的地方。”

3. 在编写任何代码之前，规格说明必须提交给外部测试小组，以详细地检查说明的完整性和明确性。开发人员自己无法完成这项工作。(Vyssotsky)

4. “10 年内(1965—1975 年)，Wirth 自上而下地进行设计(逐步细化)将会是最重要的新型形式化软件开发方法。”

5. Wirth 主张，在每个步骤中，都尽可能地使用级别较高的表达

方法。

6. 好的自上而下的设计从四个方面避免了 bug。

7. 有时必须回退，推翻顶层设计，重新开始。

8. 结构化编程中，程序的控制结构仅由支配代码块(相对于任意的跳转)的给定集合所组成。这种方法很好地避免了 bug，是一种正确的思考方式。

9. Gold 的试验结果显示，在交互式调试过程中，第一次交互取得的工作进展是后续交互的 3 倍。这实际上获益于在调试开始之前仔细地调试计划。(我认为，在 1995 年**依然**会如此。)

10. 我发现对良好(对交互式调试做出快速反应)系统的正确使用，往往要求每两小时的终端会话对应于两小时的桌面工作：1 小时会话后的清理和文档工作；1 小时为下一次计划变更和测试。

11. 系统调试(相对于单元测试)所花费的时间会比预料的更长。

12. 系统调试的困难程度证明了需要一种完备系统化和可计划的方法。

13. 系统调试仅仅应该在所有部件能够运作之后开始。(这既不同于为了查出接口 bug 所采取的“合在一起尝试”的方法，也不同于在所有构件单元的 bug 已知但未修复的情况下，即开始系统调试的做法。对于多个团队尤其如此。)

14. 开发大量的辅助调试平台和测试代码是很值得的，代码量甚至可能有测试对象的一半。

15. 必须有人对变更和版本进行控制和文档化，团队成员应使用开发库的各种受控拷贝来工作。

16. 系统测试期间，一次只添加一个构件。

17. Lehman 和 Belady 出示了证据，变更的阶段(量子)要么很大，间隔很宽；要么小且频繁。后者很容易变得不稳定。[Microsoft(微软)的一个团队使用了非常小而频繁的阶段(量子)。结果每天晚上都需要重新编译生成增长中的系统。]

第 14 章　祸起萧墙

1. “项目是怎样被延迟了整整一年时间的……一次一天。”

2. 一天一天的进度落后比起重大灾难更难以识别，更不容易防范和更加难以弥补。

3. 根据一个严格的进度表来控制大型项目的第一个步骤是**制定**进度表，进度表由里程碑和日期组成。

4. 里程碑必须是具体的、特定的和可度量的事件，能进行清晰的定义。

5. 如果里程碑定义得非常明确，以至于无法自欺欺人时，程序员很少会就里程碑的进展弄虚作假。

6. 对于大型开发项目中的估计行为，政府的承包商所做的研究显示：每两周进行仔细修订的活动时间估计，随着开始时间的临近不会有太大的变化；期间内对时间长短的**过高**估计，会随着活动的进行持续下降；过低估计直到计划的结束日期之前大约三周左右，才会有所变化。

7. 慢性进度偏离是士气杀手。[Microsoft 的 Jim McCarthy 说：“如果你错过了一个最终期限(deadline)，确保完成下一条最终期限。”[2]]

8. 同优秀的棒球队伍一样，**进取**对于杰出的软件开发团队是不可缺

少的必要品德。

9. 不存在关键路径进度的替代品，使人们能够辨别计划偏移的情况。

10. PERT 的准备工作是 PERT 图使用中最有价值的部分。它包括了整个网状结构的展开、任务之间依赖关系的识别和各个任务链的估计。这些都要求在项目早期进行非常专业的计划。

11. 第一份 PERT 图总是很恐怖，不过人们总是不断努力，运用才智来制定下一份 PERT 图。

12. PERT 图为那个使人泄气的借口——“其他的部分反正会落后”提供了答案。

13. 每个老板同时需要采取行动的异常信息以及用来进行分析和早期预警的状态数据。

14. 状态的获取是困难的，因为下属经理有充分的理由不提供信息共享。

15. 老板的不良反应肯定会对信息的完全公开造成压制；相反，仔细区分状态报告、毫无惊慌地接收报告、决不越俎代庖，将能鼓励诚实的汇报。

16. 必须有评审机制，使所有成员可以通过它了解真正的状态。出于这个目的，里程碑的进度和完成文档是关键。

17. Vyssotsky：“我发现在里程碑报告中很容易记录‘计划(老板的日期)’和‘估计(最基层经理的日期)’的日期。项目经理必须停止对估计日期的怀疑。”

18. 对于大型项目，一个对里程碑报告进行维护的**计划和控制**小组是非常可贵的。

第 15 章　另外一面

1. 对于软件编程产品来说，程序向用户所呈现的面貌——文档，与提供给机器识别的内容同样重要。

2. 即使是完全开发给自己使用的程序，描述性文字也是必需的，因为它们会被用户-作者所遗忘。

3. 培训和管理人员基本上没有向编程人员成功地灌输对待文档的积极态度——文档能在整个生命周期对克服懒惰和进度的压力起促进和激励作用。

4. 这样的失败并不都是因为缺乏热情或者说服力，而是没能正确地展示**如何**有效和经济地编制文档。

5. 大多数文档只提供了很少的总结性内容。必须放慢脚步，稳妥地进行。

6. 由于关键的用户文档包含了与软件相关的基本决策，因此它的绝大部分需要在程序编制之前书写，它包括了 9 项内容(参见相应章节)。

7. 每一份发布的程序拷贝应该包括一些测试用例，其中一部分用于校验输入数据，一部分用于边界输入数据，另一部分用于无效的输入数据。

8. 对于必须修改程序的人而言，他们需要程序内部结构文档，同样要求一份清晰明了的概述，它包括了 5 项内容(参见相应章节)。

9. 流程图是被吹捧得最过分的一种程序文档。详细逐一记录的流程图是一件令人生厌的事情，而且高级语言的出现使它显得陈旧过时。(流程图是**图形化**的高级语言。)

10. 如果这样，很少有程序需要一页纸以上的流程图。(在这一点上，

MILSPEC 军用标准文档需求实在错得很厉害。)

11. 即使的确需要一张程序结构图,也并不需要遵照ANSI的流程图标准。

12. 为了使文档易于维护，将它们合并至源程序是至关重要的，而不是作为独立文档进行保存。

13. 最小化文档担负的三个关键思路：

- 借助那些必须存在的语句，如名称和声明等，来附加尽可能多的“文档”信息；
- 使用空格和格式来表现从属和嵌套关系，提高程序的可读性；
- 以段落注释，特别是模块标题的形式，向程序中插入必要的记叙性文字。

14. 程序修改人员所使用的文档中，除了描述事情如何，还应阐述它**为什么**那样。对于加深理解，**目的**是非常关键的，即使是高级语言的语法，也不能表达目的。

15. 在线系统的高级语言(应该使用的工具)中，自文档化技术发现了它的绝佳应用和强大功能。

第 1 版结束语

1. 软件系统可能是人类创造中最错综复杂的事物(从不同类型组成部分数量的角度出发)。

2. 软件工程的焦油坑在将来很长一段时间内仍然会使人们举步维艰，无法自拔。

CHAPTER 19

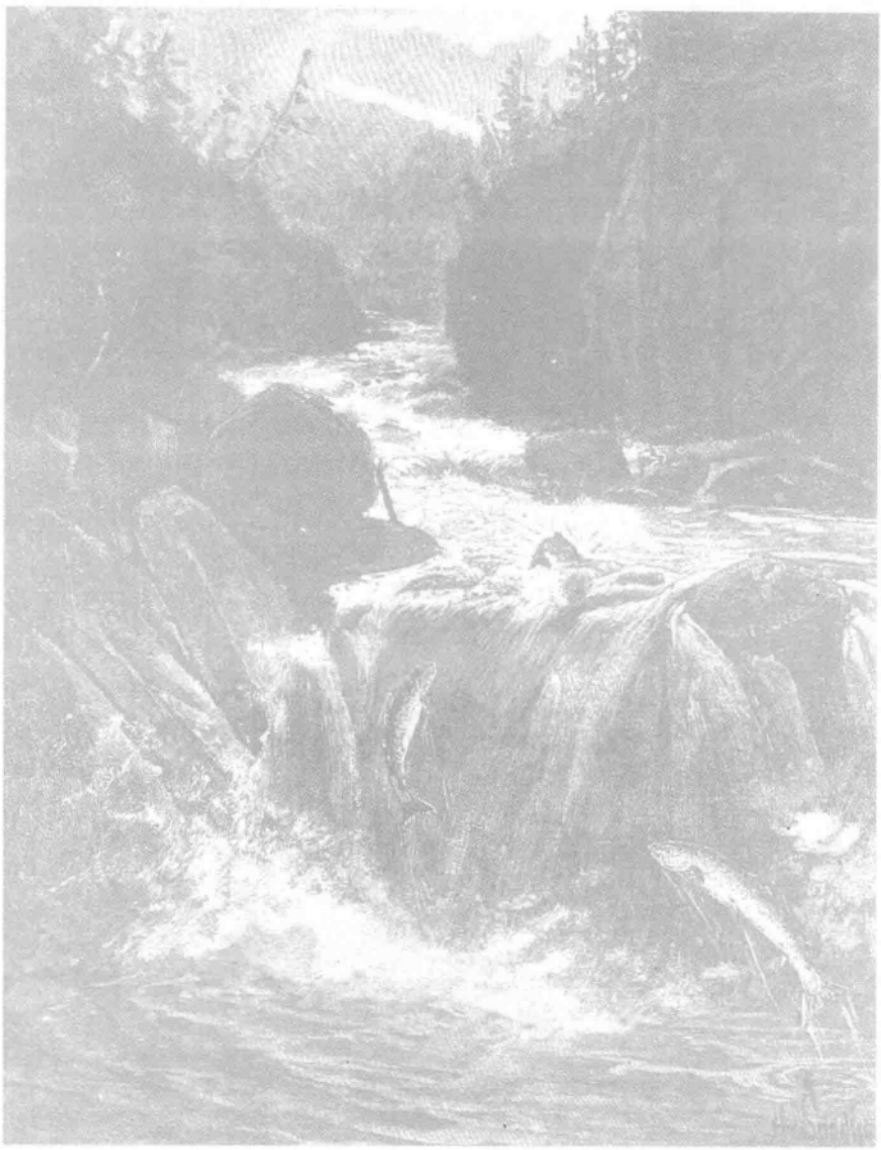
THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

第 19 章

20 年后的《人月神话》

"The Mythical Man-Month" after 20 Years



《急流勇进》(*Shooting the Rapids*)

资料来源：The Bettman Archive

只能根据过去判断将来。

——帕特里克·亨利

然而永远无法根据过去规划将来。

——埃德蒙·伯克

I know no way of judging the future but by the past.

PATRICK HENRY

You can never plan the future by the past.

EDMUND BURKE

为什么要出版20周年纪念版本

飞机划破夜空，嗡嗡地飞向纽约的拉瓜迪亚机场。所有的景色都隐藏在云层和黑暗之中。我正在看一篇平淡无奇的文档，不过并没有感到厌烦。紧挨着我的一位陌生人正在阅读《人月神话》。我在旁边一直等待着，看他是否会通过文字或者手势做出反应。最后，当我们向舱门移动时，我无法再等下去了：

“这本书如何？你有什么想法吗？”

“噢！这里面的东西我早就知道。”

此刻，我决定不介绍自己。

为什么《人月神话》得以持续？为什么看上去它仍然和现在的软件实践相关？为什么它还拥有软件工程领域以外的读者群，律师、医生、社会学家、心理学家和软件人员一样，不断地对这本书提出评论意见，引用它，并和我保持通信？20年前的一本关于30年前软件开发经验的书，如何能够依然和现实情况相关？更不用说有所帮助了。

我常听到的一个解释是软件开发学科没有正确地发展。人们经常通过比较计算机软件开发生产率和硬件制造生产率来支持这个观点，后者在20年内至少翻了1 000倍。正像第16章所解释的，反常的并不是软件发展得太慢，而是计算机硬件技术以一种与人类历史不相配的方式爆发出来。大体上这源于计算机制造从装配工业向流水线工业、从劳动密集型向资金密集型的逐渐过渡。与生产制造相比，硬件和软件开发保持着固有的劳动密集型特性。

第二个经常提及的解释——《人月神话》仅仅是顺便提及了软件，而主要针对团队中的成员如何创建事物。这种说法的确有些道理。1975年版本的前言中提到，软件项目管理并不像大多数程序员起初所认为的

那样，而更加类似于其他类型的管理。现在，我依然认为这是正确的。人类历史是一个舞台，总是上演着相同的故事。随着文化的发展，这些故事的剧本变化非常缓慢，而舞台的布局却在随时改变。正是如此，我们发现 20 世纪本身会反映在莎士比亚、荷马的作品和《圣经》中。因此，某种程度上，《人月神话》是关于人与团队的书，所以它的淘汰过程会是缓慢的。

不管出于什么原因，读者仍然在购买这本书，并且常给我发一些致谢的评论。现在，我常常被问到："你现在认为哪些在当时就是错误的？哪些是现在才过时的？哪些是软件工程领域中新近出现的？"这些独特的问题都很好，我将尽最大的能力来回答它们。不过，不以上述顺序，而是按照一系列主题来答复。首先，让我们考虑那些在写作时就正确，且现在依然成立的部分。

核心观点——概念完整性和结构师

概念完整性。一个整洁、优雅的编程产品必须向它的每位用户提供一个条理分明的概念模型，这个模型描述了应用、实现应用的方法以及用来指明操作和各种参数的用户界面使用策略。用户所感受到的产品概念完整性是易用性中最重要的因素。(当然还有其他因素。Macintosh 上所有应用程序界面的统一就是一个重要的例子。此外，有可能建立统一的接口，尽管它可能很粗糙，就像 MS-DOS。)

有很多由一个或者两个人设计的优秀软件产品例子。大多数纯智力作品，如书籍、音乐等都是采用这种方式创作出来的。不过，很多产业的产品开发过程无法负担这种获取概念完整性的直接方法。竞争压力产生了紧迫性，很多现代工艺的最终产品是非常复杂的，它们的设计需要很多人月的工作量。软件产品十分复杂，在进度上的竞争也异常激烈。

任何规模很大或者非常紧急，并需要很多人力的项目，都会碰到一个特别的困难：必须由很多人来设计，与此同时，还需要在概念上与单个使用人员保持一致。如何组织设计队伍来获得上述的概念完整性？这是《人月神话》关注的主要问题。其中一点：由于参与人数的不同，大型编程项目与小型项目的管理在性质上都不同。为了获得一致性，经过深思熟虑的，有时甚至是英勇的管理活动是完全必要的。

结构师。从第 4 章到第 7 章，我一直在不断地表达一个观点——委派一名产品**结构师**是最重要的行动。**结构师**负责产品所有方面的概念完整性，这些是用户能实际感受到的。结构师开发用于向用户解释使用的产品概念模型，概念模型包括所有功能的详细说明以及调用和控制的方法。结构师是这些模型的所有者，同时也是用户的代理。在不可避免地对功能、性能、规模、成本和进度进行平衡时，其卓有成效地体现用户的利益。这个角色是全职工作，只有在最小的团队中，才能和团队经理的角色合并。结构师就像电影的导演，而经理类似于制片人。

将体系结构和设计实现、物理实现相分离。为了使结构师的关键任务更加可行，有必要将用户所感知的产品定义——体系结构，与它的实现相分离。体系结构和实现的划分在各个设计任务中形成了清晰的边界，边界两边都有大量的工作。

结构师方案的重用。对于大型系统，即使所有实现方面的内容都被分离出去，一个人也无法完成所有体系结构的工作。所以，有必要由一位主结构师把系统分解成子系统，系统边界应该划分在使子系统间接口最小化和最容易严格定义的地方。每个部分拥有自己的结构师，他必须就体系结构向主结构师汇报。显然，这个过程可以根据需要重复递归地进行。

今天，我比以往更加确信：概念完整性是产品质量的核心。拥有一位结构师是迈向概念完整性最重要的一步。这个原理决不仅限于软件系

统，它适用于任何复杂事物的设计，如计算机、飞机、战略防御系统和全球定位系统等。在软件工程实验室进行 20 次以上的讲授之后，我开始坚持每 4 名学生组成的小组就选择不同的经理和结构师。在如此小的队伍中定义截然不同的角色可能有点极端，但我仍然发现这种方法即使对小型团队也运作良好，并且促进了设计的成功。

开发第二个系统所引起的后果——盲目的功能和频率猜测

为大型用户群设计。个人计算机革命的一个结果是，至少在商业数据处理领域中，客户应用程序越来越多地被商用软件包所代替。而且，标准软件包以成百上千，甚至是数百万拷贝的规模出售。源厂商支持性软件的系统结构师必须不断地为大型的不确定用户群，而不是为某个公司的单一、可定义的应用进行设计。许许多多的系统结构师现在面临着这项任务。

但自相矛盾的是，设计通用工具比设计专用工具更加困难，这是因为必须为不同用户的各种需要分配权重。

盲目的功能。对于如电子表格或字处理等通用工具的结构师，一个不断困扰他们的诱惑是以性能甚至是易用性为代价，过多地向产品添加边界实用功能。

功能建议的吸引力在初期阶段是很明显的，性能代价在系统测试时才会出现。而随着功能一点一点地增加，手册慢慢地变厚，易用性损失以不易察觉的方式蔓延。[1]

对幸存和发展了若干代的大众软件产品，这种诱惑特别强烈。数百万的用户需要成百上千的功能特色，任何需求本身就是一种“市场需要

它”的证明。而常见的情况是，原有的系统结构师得到了嘉奖，正工作在其他岗位或项目上，而现在负责体系结构的结构师，在均衡表达用户的整体利益方面往往缺乏经验。一个对 Microsoft Word 6.0 的近期评价声称：“Word 6.0 对功能特性进行了打包，通过包缓慢地更新……Word 6.0 同样是大型和慢速的。”有点令人沮丧的是——Word 6.0 需要 4MB 内存，丰富的新增功能意味着“甚至 Macintosh IIfx 都不能胜任 Word 6 的任务”。[2]

定义用户群。用户群越大和越不确定，就越有必要明确地定义用户群，以获得概念完整性。设计队伍中的每个成员对用户都有一幅假想的图像，并且每个设计者的图像都是不同的。结构师的用户图像会有意或者无意地影响每个结构决策，因此有必要使设计队伍共享一幅相同的用户图像。这需要把用户群的属性记录下来，包括：

- 他们是谁；
- 他们需要(need)什么；
- 他们认为自己需要(need)什么；
- 他们想要(want)的是什么。

频率。对于任何软件产品，任何用户群属性实际上都是一种概率分布，每个属性具有若干可能的值，每个值有自己的发生频率。结构师如何成功地得到这些发生频率？对并未清晰定义的对象进行调查是一种不确定和成本高昂的做法。[3]经过很多年，我现在确信，为了得到完整、明确和共享的用户群描述，结构师应该**猜测**(guess)或者假设(postulate)一系列完整的属性和频率值。

这种不是很可靠的过程有很多好处。首先，仔细猜测频率的过程会使结构师非常细致地考虑对象用户群。其次，把它们写下来一般会引发讨论，这能够起到解释的作用，并澄清不同设计人员对用户图像认识上

的差异。另外，明确地列举频率能帮助大家认识到哪些决策依赖于哪些用户群属性。这种非正式的敏感性分析也是颇有价值的。当某些非常重要的决策需要取决于一些特殊的猜测时，很值得为那些数值花费精力来取得更好的估计。(Jeff Conklin 开发的 gIBIS 提供了一种工具，能精确和正式地跟踪设计决策和文档化每个决策的原因。[4]我还没有机会使用它，但是我认为它应该非常有帮助。)

总结：为用户群的属性明确地记载各种猜测。**清晰和错误都比模糊不清好得多。**

“开发第二个系统所引起的后果(second-system effect)**”是什么？**一位敏锐的学生说，《人月神话》推荐了一剂对付灾难的处方：计划发布任何新系统的第二个版本(第 11 章)，第二个系统在第 5 章中被认为是最危险的系统。我不得不说，他上当了。

这实际上是语言引起的差异，现实情况并非如此。第 5 章中提到的“第二个”系统是第二个实际系统，它是引入了很多新增功能和修饰的后续系统。第 11 章中的“第二个”系统指开发第一个实际系统所进行的第二次尝试。它在所有的进度、人员和范围约束下开发，这些约束刻画了项目的特征，形成了开发准则的一部分。

图形界面的成功

在过去的几十年间，软件开发领域中令人印象最深刻的进步是窗口(Windows)、图标(Icons)、菜单(Menus)、指针选取(Pointing)界面的成功——或者简称为 WIMP。这些在今天是如此的熟悉，不需要做任何解释。这个概念首先在 1968 年西部联合计算机大会(Western Joint Computer Conference)上，由斯坦福研究机构(Stanford Research Institute)的 Doug Englebart 与他的团队公开提出。[5]接着，这种思想被 Xerox Palo Alto

Research Center 所采纳，用在了由 Bob Taylor 和他的团队所开发的 Alto 个人工作站中。Steve Jobs 在 Apple Lisa 型计算机中应用了该理念，不过 Apple Lisa 运行速度太慢，无法承载这个令人激动的易用性概念。后来，在 1985 年，Jobs 在取得商业成功的 Apple Macintosh 机器上体现了这些想法。接下来，它们被 IBM PC 及其兼容机的 Microsoft Windows 所采用。我自己的例子则是 Mac 版本。[6]

通过类比获得的概念完整性。WIMP 是一个充分体现了概念完整性的用户界面的例子，完整性的获得是通过采用大家非常熟悉的概念模型——对桌面的比喻和一致、细致的扩展，后者充分发挥了计算机的图形化实现能力。例如，窗口采用覆盖，而不是排列的方式，这直接来自类比。尽管这种方法成本很高，但却是正确的决定。计算机图形介质提供了对窗口尺寸和形状的调整，这是一种保持一致概念的延伸，给用户提供了新的处理能力，桌面上的文件是无法轻易地调整大小和改变形状的；拖放功能则直接出自模仿，使用指针来选择图标是对人用手拾起东西的直接模拟；图标和嵌套文件来源于桌面的文档，回收站也是如此；剪切、复制和粘贴则完全反映了我们使用桌面文档的一些习惯；我们甚至可以通过向回收站拖放磁盘的图标来弹出磁盘——象征手法是如此的贴切，扩展是如此的连贯一致，新用户常常会被它所体现出的理念打动。如果界面不是如此一致，甚至自相矛盾，效果就没有那么明显了。

哪些地方使 WIMP 界面远远超越了桌面的比喻？主要是在两个方面：菜单和单手操作。在真正的桌面上工作时，人们实际上是操作文档，而不是叫某人来完成这些动作。当要求他人进行某个活动时，常常是引发新的指令，而不是选择一个口头或者书面祈使句：“请将这个归档”，“请找出前面的信件”，“请把这个交给 Mary 去处理”。

无论是手写还是口头的命令，现有的处理能力还无法对自由产生的命令形式进行可靠的翻译和解释。所以，界面设计人员从用户对文档的

直接动作中删除了上面提到的两个步骤。他们非常聪明地从桌面文档操作中选取了一些常用命令，形成了类似于公文的“便条”，用户只需从一些语义标准的强制命令菜单中进行选择。这个概念接着被延伸到有垂直下拉子菜单的水平菜单中。

命令表达和双光标问题。命令是祈使句，它们通常都有一个动词和直接宾语。对于任何动作，必须指定一个动词和一个名词。对事物选取的直接模仿要求：使用屏幕上不同的两个光标，同时指定两件事物。每个光标分别由左右手中的鼠标来控制。毕竟，在实际的桌面上，我们通常使用两只手来操作。(不过，一只手常常是将东西固定在某处，这一点在计算机桌面是默认的情况。)而且，我们当然具备双手操作的能力，我们习惯上使用双手来打字、驾驶和烹饪。但是，提供一个鼠标已经是个人计算机制造商向前迈进的一大步，没有任何商业系统可以容纳由双手分别控制的两只鼠标同时进行的动作。[7]

界面设计人员接受了现实，为一只鼠标设计。设计人员采用的句法习惯是首先指出(选择)名词，接着指出动词——一个菜单项。这确实牺牲了很多易用性。当我看到用户、用户录像或者计算机跟踪光标移动时，我立刻对一个光标必须完成两件事而感到惊讶：选择窗口上桌面部分的一个对象，再选择菜单部分的一个动词；寻找或者重新寻找桌面上的一个对象，接着，拉下菜单(常常是同一个)选择一个动词。光标来来往往、周而复始地从数据区移到菜单区，每一次都丢弃了一些有用的位置信息，如“上次在这个空间的什么地方”。总而言之，这是一个低效的过程。

一个卓越的解决方案。即使软件和器材可以很容易地实现两个同时活动的光标，也仍然存在一些空间布局上的困难。WIMP 象征手法中的桌面实际上包括了一个打字机，它必须在实际桌面的物理空间中容纳一个物理键盘。键盘加上两个鼠标垫会占据大量双手所及的空间。不过，键盘问题实际上是一个机会——为什么不用一只手在键盘上指定动词，

另一只手使用鼠标来指定名词，从而使高效的双手操作成为可能呢？这时，光标停留在数据区，为后续名词点击拾取提供了充分的空间活动能力。这是真正的高效，真正强大的用户功能。

用户功能和易用性。不过，这个解决方案舍弃了一些易用性——菜单提供了任何特定状态下的一些可选的有效动词。例如，我们可以购买某个商品，将它带回家，谨记购买的目的，遵照菜单上不同的动词略微试验一下，就可以开始使用，并不需要去查看手册。

软件结构师所面临的最困难的问题是如何确切地平衡用户功能和易用性。是为初学者或偶尔使用的用户设计能简单操作的功能，还是为专业用户设计强大的功能呢？理想的答案是通过概念一致的方式把两者都提供给用户——这正是 WIMP 界面所达到的目标。每个频繁使用的菜单动词(命令)都有一个快捷键，因此可以作为组合通过左手一次性地输入。例如，在 Mac 机器上，命令键(⌘)正好在 Z 和 X 键的下方，因此使用最频繁的操作被编码成⌘z、⌘x、⌘c、⌘v、⌘s。

从新手向熟练用户的逐渐过渡。双重指定命令动词的系统不但满足了新手较低的需要，也满足了熟练用户对效率的需求，而且它在不同的使用模式之间提供了平滑的过渡。被称为**快捷键**的字符编码，显示在菜单上的动词旁边，因此没有把握的用户可以激活下拉菜单，检查对应的快捷键，而不是直接在菜单上选取。每个新手从他最常使用的命令中学习快捷键。⌘z 可以撤销任何单一操作，因此他可以尝试任何感到不确定的快捷键。另外，他可以检查菜单，以确定什么命令是有效的。新手会大量地使用菜单，而熟练用户几乎不使用，中间用户仅偶尔需要访问菜单，因为每个人都了解组成自己大多数操作的少数快捷键。我们中大多数的开发设计人员对这样的界面非常熟悉，对其优雅而强大的功能感到非常欣慰。

由于实施强制性的体系结构，可成功地实现设备的直接整合。Mac 界面在另一个方面很值得注意。没有任何强迫，它的设计人员在所有的应用程序中使用标准界面，包括了大量的第三方所写的程序。这样用户在界面上获得的概念一致性不仅仅局限在机器所配备的软件方面，而且遍及所有的应用程序。

Mac 设计人员把界面固化到只读内存中，使开发者使用这些界面比开发自己的特殊界面更容易和快速。这些获取一致性的措施得到了非常广泛的应用，可以形成实际的标准。苹果(Apple)的管理投入和大量说服工作协助了这些措施。产品杂志中很多独立评论家，认识到了跨应用概念完整性的巨大价值，通过批评不遵从产品的反面例子，对上述方法进行了补充。

这是第 6 章中所推荐技术的一个非常杰出的例子，该技术通过鼓励其他人直接将某人的代码合并到自己的产品中来获得一致性，而不是试图根据某人的技术说明来开发自己的软件。

WIMP 的命运：过时被淘汰。尽管 WIMP 有很多优点，我仍期望 WIMP 界面在下一代技术中成为历史。如同我们支配自己的机器一样，指针选取仍将是表达名词的方式，语音则无疑成为表达动词的方法。Mac 上的 Voice Navigator 和 PC 上的 Dragon 已经提供了这种能力。

没有构建舍弃原型—— 瀑布模型是错误的

一幅让人无法忘怀的图画，倒塌的塔科马大桥，开启了第 11 章。文中强烈地建议：“为舍弃而计划。无论如何，你一定要这样做。”现在我认为这是错误的，并不是因为它太过极端，而是因为它太过简单。

在“未雨绸缪”一章(第 11 章)所体现的概念中，最大的错误是它隐含地假设了使用传统的顺序或者瀑布开发模型。该模型源自类似甘特图布局的阶段化流程，常常绘制成如图 19-1 所示的形状。Winton Royce 在 1970 年的一篇经典论文中改进了顺序模型，他提出：

- 存在一些从一个阶段到前一个阶段的反馈；
- 将反馈限定在直接相邻的先前阶段，从而遏制它引起的成本增加和进度延迟。

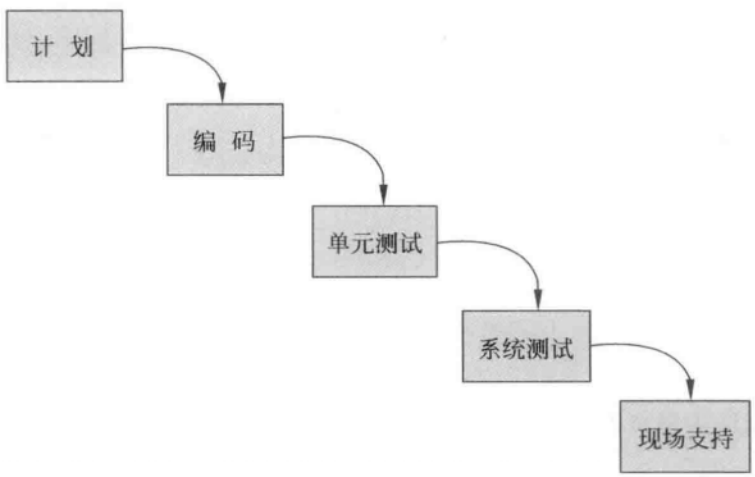


图 19-1　软件开发的瀑布模型

他给开发者提出了“构建两次”的建议。[8]受到瀑布模型不良影响的并不只是第 11 章，而是从第 2 章的进度计划规则开始，贯穿了整本书。第 2 章中的经验法则分配了 1/3 的时间用于计划，1/6 用于编码，1/4 用于单元测试以及 1/4 用于系统测试。

瀑布模型的基本谬误是它假设项目只经历一次过程，而且体系结构出色并易于使用，设计是合理可靠的，随着测试的进行，编码实现是

可以修改和调整的。换句话说，瀑布模型假设所有错误发生在编码实现阶段，因此它们的修复可以很顺畅地穿插在单元和系统测试中。

“未雨绸缪”明确地迎面痛击了这个错误。它不是对错误的诊断，而是补救措施。现在，我建议应该一块块地丢弃和重新设计系统，而不是一次性地完成替换。就目前的情况而论，这没有问题，但它并没有触及问题的根本。瀑布模型把系统测试以及潜在地把用户测试放在构件过程的末尾。因此，只有在投入了全部开发投资之后，才能发现无法接受的性能问题、笨拙功能以及察觉用户的错误或不当企图。不错，Alpha 测试对规格说明的详细检查是为了尽早地发现这些缺陷，但是对于实际参与的用户却没有对应的措施。

瀑布模型的第二个谬误是它假设整个系统一次性地被构建，在所有的设计、大部分编码、部分单元测试完成之后，才为闭环的系统测试合并各个部分。

瀑布模型，这个大多数人在 1975 年考虑的软件项目开发方法，不幸地被奉为军用标准 DOD-STD-2167，作为所有国防部(DoD)军用软件的规范。所以，在大多数有见地的从业者认识到瀑布模型的不完备并放弃之后，它仍然得以幸存。幸运的是，DoD 也已经慢慢察觉到这一点。[9]

必须存在逆向移动。就像本章开始图片中精力充沛的大马哈鱼一样，在开发过程“下游”的经验和想法必须跃行而上，有时会超过一个阶段，来影响“上游”的活动。

例如，设计实现会发觉有些体系结构的功能定义会削弱性能，因此体系结构必须重新调整。编码实现会发现一些功能会使空间剧增，超过要求，因此必须更改体系结构和设计实现。

所以，在把任何东西变成代码之前，可能要往复迭代两个或更多的“体系结构—设计”以实现循环。

增量开发模型更佳——渐进地精化

构建闭环的框架系统

从事实时系统环境开发的 Harlan Mills，早期曾提倡，我们首先应该构建实时系统的基本轮询回路，为每个功能提供子函数调用(**占位符**)，但仅仅是空的子函数(见图 19-2)。对它进行编译、测试，可以使它不断运行。它不直接完成任何事情，但至少是正常运行的。[10]

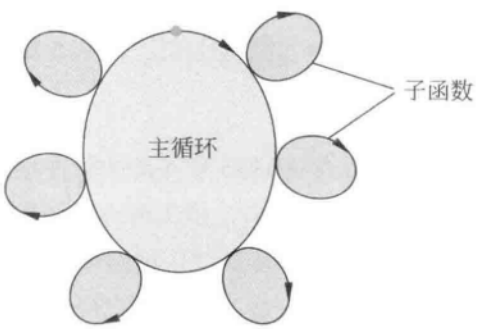


图 19-2　实时系统的基本轮询回路

接着，我们添加(可能是基本的)输入模块和输出模块。瞧，一个可运行的系统出现了，尽管只是一个框架。然后，一个功能接一个功能，我们逐渐开发和增加相应的模块。**在每个阶段，我们都拥有一个可运行的系统。**如果我们非常勤勉，每个阶段就都会有一个经过调试和测试的系统。(随着系统的增长，使用所有先前的测试用例对每个新模块进行的回归测试也采用这种方式进行。)

在每个功能基本可以运行之后，我们一个接一个地精化或者重写每个模块——**增量地开发**(growing)整个系统。不过，我们有时的确需要修改原有的驱动回路，或者甚至是回路的模块接口。

我们在所有时刻都拥有一个可运行的系统，因此：

- 我们可以很早就开始用户测试；
- 我们可以采用按预算开发的策略，彻底保证不会出现进度或者预算超支的情况(以允许的功能牺牲作为代价)。

我曾在北卡罗来纳大学教授了 22 年的软件工程实验课，有时与 David Parnas 一起。在这门课程中，通常 4 名学生的团队会在一个学期内开发某个真正的实时软件应用系统。大约过了一半时间，我转而教授增量开发的课程。我常常因为屏幕上第一幅图案、第一个可运行的系统对团队士气产生的鼓舞效果而感到震惊。

Parnas 产品族

在这整个 20 年的时间里，David Parnas 曾是软件工程思潮的带头人。每个人对他的信息隐藏概念都很熟悉，但对他的另一个非常重要的概念——将软件作为一系列相关的产品族来设计[11]——相对了解得较少。Parnas 力劝设计人员对产品的后期扩展和后续版本进行预测，定义它们在功能或者平台上的差异，搭建一棵相关产品的家族树(见图 19-3)。

设计类似一棵树的技巧是将那些不易于变化的设计决策放置在树的根部。

这样的设计策略使得模块的作用最大化。更重要的是，可以延伸相同的策略，使它不但可以包括发布产品，还包括以增量开发策略创建的后续中间版本。这样，产品可以通过它的中间阶段，以最低限度的回溯代价增长。

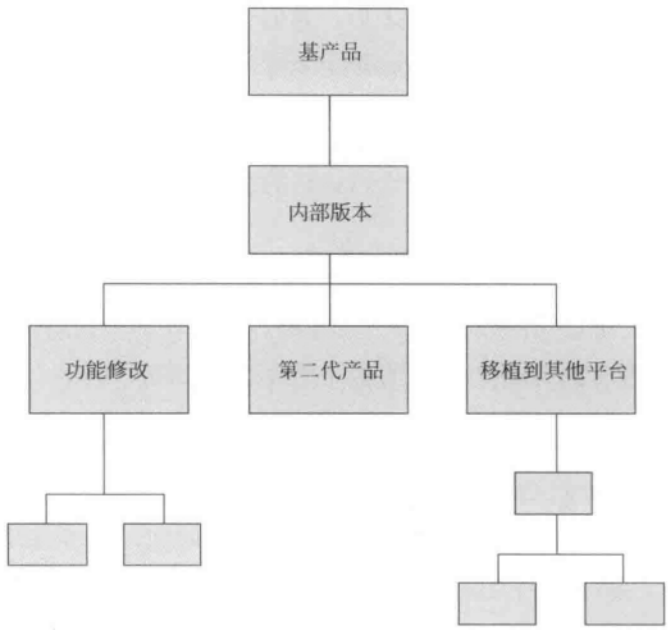


图 19-3　产品家族树

Microsoft 的“每晚重建”方法

James McCarthy 向我描述了他的队伍和微软其他团队所使用的产品开发流程，这实际上是一种逻辑上的增量式开发。他说：

在我们第一次发布产品之后，我们会继续发布后续版本，向已有的可运行系统添加更多的功能。为什么最初的构建过程要不一样呢？因此，从我们第一个里程碑开始(第一次发布有三个里程碑)，我们每晚重建开发中的系统(以及运行测试用例)。该构建周期成了项目的“心跳”。每天，一个或多个程序员—测试员队伍提交若干具有新功能的模块。在每次重建之后，我们会获得一个可运行的系统。如果重建失败，我们将停止整个过程，直到找到问题所在并进行修复。在任何时间，团队中的每个人都了解项目的状态。

这是非常困难的。你必须投入大量的资源，而且它是一个规范化、可跟踪、开诚布公的流程。它向团队提供了自身的可信度，而可信度决定了你的士气和情绪状态。

其他组织的软件开发人员对这个过程感到惊讶，甚至震惊。其中一个人说："我们可以实现每周一次的重建，但是如果每晚一次的话，我想不大可能，工作量太大了。"这可能是对的。例如，Bell 北方研究所就是每周重建 1 200 万行的系统。

增量式开发和快速原型

增量开发过程能使真正的用户较早地参与测试，那么它与快速原型之间的区别是什么呢？我认为它们既互相关联，又相互独立。各自可以不依赖对方而存在。

Harel 将原型精彩地定义成：

仅仅反映了概念模型准备过程中所做的设计决策的一个程序版本，它并未反映受实现考虑所驱使的设计决策。[12]

构建一个完全不属于发布产品的原型是完全可能的。例如，可以开发一个界面原型，但是并不包含任何的实际功能，而仅仅是一个看上去履行了各个步骤的有限状态机。甚至可以通过模拟系统响应的向导技术来原型化和测试界面。这种原型化对获取早期的用户反馈非常有用，但是它和产品发布前的测试区别很大。

类似地，实现人员可能会着手开发产品的某一块，并完整地实现该部分的有限功能集合，从而尽早地发现性能上的潜在问题。那么，"从第一个里程碑开始构建"的 Microsoft 流程和快速原型之间的差别是什么呢？是功能。第一个里程碑产品可能不包含足够的功能使任何人对它产生兴趣，而可发布的产品与定义中的一样，其在完整性上配备了一系列实用的功能集，在质量上它可以健壮地运行。

关于信息隐藏，Parnas是正确的，我是错误的

在第7章中，关于每个团队成员应该在多大程度上被允许和鼓励相互了解设计和代码的问题，我对两种方法进行了对比。在操作系统OS/360项目中，我们决定**所有**的程序员应该了解**所有**的材料——每名项目成员都拥有一份大约10 000页的项目工作手册拷贝。Harlan Mills颇有说服力地指出“编程是一个开放性的公共过程”。把所有工作都暴露在每个人的凝视之下，能够帮助质量控制，这既源于其他人优秀工作的压力，也由于同伴能直接发现缺陷和bug。

这个观点和David Parnas的观点形成了鲜明的对比。David Parnas认为，代码模块应该采用定义良好的接口来封装，这些模块的内部结构应该是程序员的私有财产，外部是不可见的。编程人员被屏蔽而不是暴露在他人模块的内部结构面前。在这种情况下，工作效率最高。[13]

我在第7章中并不认同Parnas的概念是“灾难的处方”。但是，Parnas是正确的，我是错误的。现在，我确信信息隐藏——现在常常内建于面向对象的编程中——是唯一提高软件设计水平的途径。

实际上，任何技术的使用都可能演变成灾难。Mills的技术是通过了解接口另一侧的情况，使编程人员能理解他们所工作的接口的详细语义。这些接口语义的隐藏曲解会导致系统的bug。另一方面，Parnas的技术在面对变更时是很健壮的，更加适合作为变更设计的理念。

第16章指出了下列情况：

- 过去在软件生产率上取得的进展大多数来自消除非内在的困难，如笨拙的编程语言、漫长的批处理周转时间等；

- 像这些比较容易解决的困难已经不多了；
- 彻底的进展将来自对根本困难的处理——打造和组装复杂概念性结构要素。

最明显的实现这些的方法是，认为程序由比独立的高级语言语句——函数、模块或类等更大的概念结构要素组成。如果能对设计和开发进行限制，我们仅仅需要从已建成的集合中参数化这些结构要素，并把它们组装在一起，这样我们就能大幅度提高概念的级别，消除很多无谓的工作和大量语句级别错误的可能性。

Parnas 的模块信息隐藏定义是研究项目中的第一步，它是面向对象编程的鼻祖。Parnas 把模块定义成拥有自身数据模型和自身操作集的软件实体。它的数据仅仅能通过自己的操作来访问。第二步是若干思想家的贡献：把 Parnas 模块提升到**抽象数据类型**，从中可以派生出很多对象。抽象数据类型提供了一种思考和指明模块接口的统一方式，以及容易实施的规范化的访问方法。

第三步，面向对象编程引入了一个强有力的概念——**继承**，即类(数据类型)默认获得类继承层次中祖先的属性。[14]我们希望从面向对象编程中得到的最大收获实际上来自第一步，模块封装，以及预先建成的、**为了重用而设计和测试**的模块或者类库。很多人忽视了这样一个事实，即上述模块不仅仅是程序，某种意义上是我们在第 1 章中曾讨论过的编程产品。许多人希望大规模重用，但不付出构建产品级质量(通用、健壮、经过测试和文档化的)模块所需要的初始代价——这种期望是徒劳的。面向对象编程和重用在第 16 和 17 章中有所讨论。

人月到底有多少神话色彩？Boehm 的模型和数据

很多年来，人们对软件生产率和影响它的因素进行了大量的量化研究，特别是在项目人员配备和进度之间的平衡方面。

最充分的一项研究是 Barry Boehm 对 63 个软件项目的调查，其中大多数是航空项目和 25 个 TRW 公司的项目。他的《软件工程经济学》(*Software Engineering Economics*)不仅包括很多结果，还有一系列逐步推广的有价值的成本模型。尽管一般商业软件的成本模型和根据政府标准开发的航空软件成本模型中的系数肯定不同，不过他的模型使用了大量的数据来支撑。我想从现在起，这本书将会成为一代经典。

他的结果与《人月神话》的结论充分地吻合，即人力(人)和时间(月)之间的平衡远不是线性关系，使用人月作为生产率的衡量标准实际是一个神话。特别是他发现以下几个问题。[15]

- 第一次发布的成本最优进度时间，$T = 2.5(MM)^{1/3}$。即，月单位的最优时间是估计工作量(人月)的立方根，估计工作量则由规模估计和模型中的其他因子导出。最优人员配备曲线是由推导得出的。
- 当计划进度比最优进度长时，成本曲线会缓慢攀升。时间越充裕，所花费的时间就越长。
- 当计划进度比最优进度短时，成本曲线急剧升高。
- **无论安排多少人手，几乎没有任何项目能够在少于 3/4 的计算出的最优时间内获得成功！**当高级经理向项目经理要求不可能的进度担保时，这段结论可以充分地作为项目经理的理论依据。

Brooks 准则有多准确？曾有很多细致的研究来评估 Brooks 法则的

正确性。简言之，向进度落后的软件项目中添加人手只会使进度更加落后。最棒的研究发表在 Abdel-Hamid 和 Madnick 在 1991 年出版的一本颇有价值的书《软件项目动力学：一条完整的路》(*Software Project Dynamics: An Integrated Approach*)中。[16]书中提出了项目动态特性的量化模型。关于 Brooks 准则的章节提供了更详细的分析，指出了在各种假设的情况下，即何时添加多少人员将会产生什么样的结果。为了进行研究，作者扩展了他们自己一个中型规模项目的模型，假设新成员有学习曲线和需要额外的沟通和培训工作。他们得出结论："向进度落后的项目中添加人手总会增加项目的成本，但并不一定总会使项目更加落后。"特别地，由于新成员总会立刻带来需要数周来弥补的负面效应，所以在项目早期添加额外的人力比在后期添加更加安全一些。

Stutzke 为了进行相似的研究，开发了一个更简单的模型，得出了类似的结果。[17]他对引入的新成员进行了详细的过程和成本分析，其中包括把他们的指导人员调离原有的项目任务。他在一个真正的项目上测试了自己的模型，在项目中期的一些偏移之后，他成功地添加了一倍人手，并且保证了原先的进度。相对于增加更多程序员，他还试验了其他的方法，特别是加班工作。在他的很多条实践建议中，最有价值的部分是如何添加新成员，进行培训，用工具来支持等。特别值得注意的是，他建议，开发项目后期增加的开发人员必须作为团队成员，愿意在过程中努力投入和工作，而不是企图改变或者改进过程本身！

Stutzke 认为，在更大型的项目中，增加的沟通负担是次要作用，没有对它建模。至于 Abdel-Hamid 和 Madnick 是否或者如何考虑这个问题，则不是很清楚。上面提到的两个模型都没有考虑开发人员必须重新安排的事实，而在实际情况中，我发现这常常是一个非常重要的步骤。

这些细致的研究使"异常简化"的 Brooks 准则更加实用。作为平衡，我还是坚持这个简单的陈述作为最接近真理的经验法则——警告经

理们避免对进度落后的项目盲目地采取本能的修补措施。

人就是一切(或者说，几乎是一切)

很多读者发现很有趣的是，《人月神话》的大部分文章在讲述软件工程管理方面的事情，较少涉及技术问题。造成这种倾向的部分原因是我在 IBM OS/360 操作系统(现在是 MVS/370)项目中角色的性质。更基本的是，这来自一个信念，即对于项目的成功而言，项目人员的素质、人员的组织和管理是比使用的工具或采用的技术方法更重要的因素。

随后的研究支持了上述观点。Boehm 的 COCOMO 模型发现，目前，团队质量是项目成功最大的决定因素，实际上是下一个次重要因素的 4 倍。现在，软件工程的大多数学术研究集中在工具上。我很欣赏和期盼强大的工具，同样我也非常鼓励对软件管理动态特征——对人的关注、激励和培养——的持续研究。

人件。近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的书，《人件：高生产率的项目和团队》*(Peopleware: Productive Projects and Teams)*。它所表达的观点是："我们行业的主要问题实质上更侧重于**社会学**(sociological)而不是**科学技术**(technological)。"它充满了很多精华，如"管理人员的职责不是要人们去工作，而是创造工作的可能。"它涉及了如空间、布置、团队的餐饮等世俗的主题。DeMarco 和 Lister 在 Coding War Games 项目中提供的数据显示，相同组织中开发人员的表现之间以及工作空间同生产率、缺陷水平之间存在令人吃惊的关联。

顶尖人员的空间更加安静、更加私人、保护得更好以免受打扰，还有很多……这对你真的很要紧吗……是否安静、空间和免受打搅能够帮助你的人员更好地完成工作，或者(换个角度)能帮助你吸引和留住更好

的人员吗？[18]

我衷心地向我的读者推荐这本书。

项目转移。DeMarco 和 Lister 对团队**融合**给予了相当大的关注。团队融合是一个无形的，却非常关键的特性。我观察到，很多地点分散的公司，把项目从一个实验室转移到另一个。我认为，其忽视了团队融合这个管理中非常重要的因素。

我的观察和经验大约局限在六七个项目转移中，其中没有一个是成功的。**任务**可以成功地转移，但是对于项目的转移，即使拥有良好的文档、先进的设计以及保留部分原有人员，新队伍实际上依然是重新开始。我认为正是由于破坏了原有团队的整体性，导致产品雏形的夭折，项目重新开始。

放弃权力的力量

如果人们认同我在文中多处提到的观点——创造力来自于个人，而不是组织架构或者开发过程，项目经理面临的中心问题就是如何设计架构和流程，来提高而不是压制主动性和创造力。幸运的是，这个问题并不是软件组织所特有的，一些杰出的思想家正努力地致力于这项工作。E. F. Schumacher 在他的经典《小就是美：人们关心的经济学》(*Small is Beautiful: Economics as if People Mattered*)中，提出了最大化员工创造力和工作乐趣的企业组织理论。他的第一个原理引自 Pope Pius XI 教皇通谕(Quadragesimo Anno)中的“附属职能行使原理”：

向大型组织指派小型或者附属机构能够完成的职责是不公平的，同时也是正常次序的不幸和对它的干扰。对于每项社会活动，就其本质而言，应该配备对社会个体成员提供帮助，而不是去破坏和吸收它们……

那些当权者应该确信遵守“附属职能行使”原理，能在各种各样的组织中维持更加完美的次序，从而使社会权威和社会效率获得进一步的提高，国家更加融洽和繁荣。[19]

Schumacher 继续解释到：

附属职能行使原理告诉我们——如果较低级别组织的自由和责任得以保留，中心权威实际上是得到了加强；其结果是，从整体而言，组织机构实际上将“更加融洽和繁荣”。

如何才能获得上述的架构……大型组织机构由很多准自治单元构成，我们称之为准公司。它们中的每一个都拥有足够的自由，来为创造性和企业家职能提供最大的可能机会……每个准公司同时具备盈亏账目和资产负债表。[20]

软件工程中最激动人心的进展是将上述组织理念付诸实践的早期阶段。首先，微型计算机革命创造了新型的软件工业，出现了成百上千的新兴公司。所有这些小规模的公司最显著的是热情、自由和富有创造性。随着很多小型公司被大公司收购，这个产业正在发生着变化，而那些大公司是否理解保留小规模公司创造性的重要性尚待分晓。

更不寻常的是，一些大型公司的高层管理已经开始着手将一些权力下放到软件项目团队，使它们在结构和责任上接近于 Schumacher 的准公司。其运作的结果是令人欣喜和吃惊的。

Microsoft 的 Jim McCarthy 向我描述了他在解放团队方面的经验：

每个队伍(30～40 人)拥有自己的任务、进度，甚至如何定义、构建、发布的过程。团队由 4 或 5 位专家组成，包括开发、测试和书写文档等。对争论进行仲裁的不是老板，而是团队。我简直无法形容授权和由团队对项目自行负责成功与否的重要性。

Earl Wheeler，IBM 软件业务的退休主管，告诉我他着手下放 IBM 部门长期集权管理权力的经验：

(近年来)关键的措施是将权力向下委派。改进的质量、提高的生产率和高涨的士气，这就像是魔术！我们的小型团队，没有中心控制。团队是流程的所有者，并且必须拥有一个流程。他们有不同的流程。他们是进度计划的所有者，但能感受到市场的压力。这种压力导致他们使用和利用自己的工具。

和团队成员个人的谈话，显示了他们对被委派的权力和自由的赞同，同时反映出真正的下放显得多少有些保守。不过，授权是朝着正确的方向迈出的一大步，它产生了如 Pius XI 所预言的好处：通过权力委派，中心的权威实际上是得到了加强；就整体而言，组织机构实际上更加融洽和繁荣。

最令人惊讶的新事物是什么？数百万的计算机

每位我曾交谈过的计算机带头人都承认，他们对微型计算机革命和它引发的塑料薄膜包装软件产业感到惊讶。毫无疑问，这是继《人月神话》后 30 多年中最重要的改变。它对软件工程意味着很多。

微型计算机革命改变了每个人使用计算机的方式。Schumacher 在 30 多年前，陈述了面对的挑战：

我们真正想从科学家和技术专家那里得到什么？我会回答，我们需要这样的方法和设备：

- 价格足够低廉，使几乎所有人都能够使用；
- 适用于小规模的应用；

- 满足人们对创造的渴望。[21]

这些正是微型计算机革命带给计算机产业和它的用户(现在已覆盖到普通公众)的杰出特性。一般人现在不但可以买得起自己的计算机，而且还可以负担30多年前只有国王的薪水才能买得起的软件。Schumacher的每个目标都值得仔细思考，每个目标达到的程度值得品评，尤其是最后一个。在一个一个的领域中，普通人同专家一样可以应用新的自我表达方法。

其他领域中进步的部分原因和软件创造相近——消除了次要的困难。例如，文书处理方式曾经是很僵化的，合并更改内容需要重新打字，成本和时间都比较高昂。一份300页的手稿，常常每3～6个月就需要重新输入一遍。这中间，人们往往要不断地在文稿上做标记。另外，逻辑流程和语句韵律的修订很难进行。而现在，文书处理已经非常方便和流畅了。[22]

计算机同样给其他一些领域带来了相似的处理能力，绘画、制定计划、机械制图、音乐创作、摄影、摄像、幻灯、多媒体甚至是电子表格等。在这些领域，手工操作需要重新拷贝大量的未改变的部分，以便在上下文中区别修改情况。现在我们能享受这样的好处，即立刻对结果进行修订和评估，无需失去思维的连贯性，就像分时带给软件开发的好处一样。

同样，新的、灵活的辅助工具提高了创造力。以写作为例，我们现在拥有拼写检查、语法检查、风格顾问、目录生成系统以及对最终排版预览的能力。我们还没有意识到，唾手可得的百科全书或万维网上的无穷资源，对一个作家即兴搜索材料意味着什么。

最重要的是，当一件创造性工作刚刚成形时，工作介质的灵活性使得对不同的可选方案的探索变得容易。这实际上是一个量变引起质变的

例子，即时间变化引起工作方式上的巨大变化。

绘图工具使建筑设计人员为每小时的创造性投资探究了更多的选择。计算机与合成器的互联，加上自动生成或者演奏乐谱的软件，使得人们更容易捕获创作的灵感。数字式相机和 Adobe Photoshop 一起，使原先在暗室中需要数小时的工作在几分钟内就可以完成。电子表格可以对大量“如果……那么”的各种情况进行实验、比较。

最后，个人计算机的普遍存在导致了全新创造性活动介质的出现。Vannevar Bush 在 1945 年提出的超文本，仅能在计算机上实现。[23]多媒体表现形式和体验更是如此——在个人计算机和大量价格低廉的软件出现以前，实现起来有太多的困难。并不便宜或普遍的虚拟环境系统，将成为另一个创造性活动的媒介。

微型计算机革命改变了每个人开发软件的方式。20 世纪 70 年代的软件开发过程本身被微处理器革命和它所带来的科学技术进步所改变。很多软件开发过程的次要困难已被消除。快速的个人计算机现在是软件开发者的常规工具，周转时间的概念几乎成为了历史。如今的个人计算机不仅比 1960 年的超级计算机要快，而且它比 1985 年的 Unix 工作站还要快。所有这些意味着即使在最差的计算机上，编译也是快速的，而且大内存消除了基于磁盘链接所需要的等待时间。另外，符号表和目标代码可以在内存中保存，使高级别的调试无需重新编译。

在过去的 20 年里，我们几乎全部采用了分时作为构建软件的方法学。在 1975 年，分时才刚刚作为最常用的技术替换了批处理计算。网络使软件构建人员不仅可以访问共享文件，还可以访问强大的编译、链接和测试引擎。今天，个人工作站提供了计算引擎，网络主要提供了对文件的共享访问，这些文件将作为团队开发的工作产品。客户—服务器系统则使测试用例检入、开发和应用的共享访问更加简单。

同样，用户界面也取得了类似的进步。和一般的文本一样，WIMP界面对程序文本提供了更加方便迅捷的编辑方式。24行、72列的屏幕已经被整页甚至是双页的屏幕所取代，因此程序员可以看到更多的其所做更改的上下文信息。

全新的软件产业——塑料薄膜包装的成品软件

在传统软件产业的旁边，爆发了另一个全新的产业。产品以成千上万，甚至是数百万的规模销售。整套内容丰富的软件包可以以低于1个支持程序员1个人天的成本获得。这两个产业在很多方面都不同，它们共同存在着。

传统软件产业。在1975年，软件产业拥有若干可识别的但多少有些差异的组成部分，如今它们依然存在：

- 计算机提供商：提供操作系统、编译器和一些实用程序；
- 应用程序用户：如公共事业单位、银行、保险公司和政府机构等，它们为自己使用的软件开发应用程序包；
- 定制程序开发者：为用户开发私用软件包，这类承包商大多数工作在国防项目上，这些项目的需求、标准和行销步骤都是与众不同的；
- 商业包开发者：那个时候是为专业市场开发大型应用，如统计分析软件包和CAD系统等。

Tom DeMarco注意到了传统软件产业的分裂，特别是应用程序用户。

我没有料到的是：整个行业被分解成各个特殊的领域。你完成某事的方式更像是专业领域的职责，而不仅仅是使用通用系统分析方法、通用语言和通用测试技术的。Ada 是最后一个通用语言，并且它已经慢慢变成了一门专业语言。

在日常的商业应用领域中，第 4 代语言做出了巨大的贡献。Boehm 说："大多数成功的第 4 代语言是以选项和参数方式系统化某个应用领域的结果。"这些第 4 代语言最普遍的情况是带有查询语言的数据库-通讯软件包和应用生成器。

操作系统世界已经统一了。在 1975 年，存在着很多操作系统：每个硬件提供商在每条产品线上最少有一种操作系统，很多提供商甚至有两个。如今是多么不同啊！开放式系统是基本原则。目前，人们主要在五大操作系统环境上行销自己的应用程序包(按照时间顺序)：

- IBM MVS 和 VM 环境
- DEC VMS 环境
- Unix 环境，某个版本
- IBM PC 环境，DOS、OS-2 或者 Windows
- Apple Macintosh 环境

塑料薄膜包装的成品软件产业。对于这个产业的开发者，面对的是与传统产业完全不同的经济学：软件成本是开发成本与数量的比值，包装和市场成本非常高。在传统内部的应用开发产业，进度和功能细节是可以协商的，开发成本则可能不行；而在竞争激烈的开放市场面前，进度和功能支配了开发成本。

正如人们所预期的，完全不同的经济学引发了非常不同的编程文化。

传统产业倾向于被大型公司以已指定的管理风格和企业文化所支配。另一方面，始于数百家创业公司的成品软件产业，行事自由，更加关注结果，而不是流程。在这种趋势下，那些天才的个人程序员更容易获得认可，这隐含了“卓越的设计来自于杰出的设计人员”的观点。创业文化能够对那些杰出人员，根据他们的贡献进行奖励。而在传统软件产业中，公司的社会化因素和薪资管理计划总会使上述做法难以实施。因此，很多新一代的明星人物被吸引到薄膜包装的软件产业，这一点并不奇怪。

买来开发——使用塑料包装的成品软件包作为构件

彻底提高软件健壮性和生产率的唯一途径是，提升一个级别，使用模块或者对象组合来进行程序开发。一个特别有希望的趋势是使用大众市场的软件包作为平台，在上面开发更丰富和更定制化的产品。如使用塑料包装的数据库和通讯软件包来开发货运跟踪系统，或者学生的信息系统等。而计算机杂志上的征文栏目提供了许许多多的 Hypercard stacks、Excel 定制化模板、MiniCad 的 Pascal 特殊函数以及 AutoCad 的 AutoLisp 函数。

元编程。Hypercard stacks、Excel 模板和 MiniCad 函数的开发有时被称为**元编程**(metaprograming)，为部分软件包用户进行功能定制的过程。元编程并不是新概念，仅仅是重新被提出和重新命名。在 20 世纪 60 年代早期，很多计算机提供商和大型信息管理系统(MIS)厂商都拥有小型专家小组，他们使用汇编语言的宏来装备应用编程语言。Eastman Kodak 的 MIS 开发车间使用一种用 IBM 7080 宏汇编定义的自有应用语言。类似地，IBM 的 OS/360 队列远程通信访问方法中(Queued Telecommunications Access Method)，在遇到机器级别指令之前，人们可以读到若干页表面上像汇编语言的远程通信程序。现在元编

程人员提供要素的规模是宏的若干倍。这种二级市场的开发是非常鼓舞人心的——当我们在期待 C++类开发的高效市场时，可重用元程序的市场正在悄无声息地崛起。

它处理的确实是根本问题。基于软件包开发现象并没有影响到一般的 MIS 编程人员，因此对于软件工程领域并不是很明显。不过，它将快速地发展，因为它针对的正是概念结构要素打造的根本问题。成品软件包提供了大型的功能模块和精心定制的接口，它内部的概念结构根本无需再设计。功能强大的软件产品，如 Excel 或者 4th Dimension 实际上是大型的模块，但它们作为广为人知、文档化、测试过的模块，可以用来搭建用户化系统。下一级应用程序的开发者可以获得丰富的功能、更短的开发时间、经过测试的组件、良好的文档和彻底降低的成本。

当然，存在的困难是成品软件作为独立实体来设计，元程序员无法改变它的功能和接口。另外，更严肃地说，对于成品软件的开发者而言，把产品变成更大型系统中的模块似乎没有什么吸引力。我认为这种感觉是错误的，在为方便元程序员开发提供软件包方面，有一个未开拓的市场。

那么需要什么呢？我们可以识别出四个层次的软件成品用户。

- 直接使用用户。他们以简便直接的方式来操作，对设计者提供的功能和接口感到满意。
- 元程序员。在单个应用程序的基础上，使用已提供的接口来开发模板或者函数，主要为最终用户节省工作量。
- 外部功能作者，向应用程序添加自行编制的功能。这些功能本质上是新应用语言原语，调用通用语言编写的独立模块。这往往需要命令中断、回调或者重载函数技术，向原接口添加新功能。

- 元程序员，使用一个或多个特殊的应用程序，作为更大型系统的构件。他们是需求并没有得到满足的用户群。同时，这也是能在构建新应用程序方面获得较大收获的用法。

对于成品软件，最后一种类型的用户还需要额外的文档化接口，即元编程接口(metaprogramming interface，MPI)。这在很多方面提出了要求。首先，元程序需要在整个应用程序集的控制之下，而每个软件通常假设是受自己控制的。软件集必须控制用户界面，而应用程序一般认为这是自己的职责。软件整体必须能够调用任何应用程序的功能，就好像是用户使用命令行传递参数那样。它还应该像屏幕一样接受应用程序的输出，只不过屏幕是显示一系列字符串，而它需要将输出解析成适当数据类型的逻辑单元实体。某些应用程序，如 FoxPro，提供了一些接收命令的后门接口(wormhole)，不过它返回的信息是不够充分和未被解析的。这些接口是对通用解决方案需要的一个特殊补充。

拥有能控制应用程序集合之间交互的脚本语言是非常强有力的。Unix 首先使用管道和标准的 ASCII 字符串文件格式提供了这种功能。今天，AppleScript 是一个非常优秀的例子。

软件工程的状态和未来

我曾向北卡罗来纳州立大学化学工程系的系主任 Jim Ferrell 问过关于化学工程的历史以及和化学的区别的问题，于是他做了一个 1 个小时的出色即兴演说，从很多产品(从钢铁到面包，再到香水)的不同生产过程开始。他讲述了 Arthur D. Little 博士如何在 1918 年在麻省理工学院建立了第一个工业化学系来发现、发展和讲授所有过程共享的共有技术基础。首先是经验法则，接着是经验图表，后来是设计特殊零件的公式，再后来是单个导管中热传导、质量转移和动量转移的数学模型。

如同 Ferrell 故事所展现的，在几乎 50 年后，我仍被化学工程和软件工程发展之间的很多相似之处所震动。Parnas 对我写的关于**软件工程**的文章提出了批评。他对比了电气工程和软件领域，觉得把我们所做的称为“工程”只是一厢情愿。他可能是正确的，这个领域可能永远不会发展成像电气工程那样的拥有精确的数学基础的工程化领域。毕竟，软件工程就像化学工程一样，与如何扩展到工业级别处理过程的非线性问题有关。而且，和工业工程类似，它总是被人类行为的复杂性所困扰。

不过，化学工程的发展过程让我觉得“27 岁的”软件工程并不是没有希望的，而仅仅是不够成熟的，就好像 1945 年的化学工程。毕竟，在二次世界大战之后，化学工程师才真正提出闭环互联的连续流系统。

今天，软件工程的一些特殊问题正如第 1 章中所提出的：

- 如何把一系列程序设计和构建成**系统**；
- 如何把程序或者系统设计构建成健壮的、经过测试的、文档化的、可支持的**产品**；
- 如何维持对大量的**复杂性**的控制。

软件工程的焦油坑在将来很长一段时间内会继续使人们举步维艰，无法自拔。软件系统可能是人类创造中最错综复杂的事物，只能期待人们在力所能及的或者刚刚超越力所能及的范围内进行探索和尝试。这个复杂的行业需要：进行持续的发展；学习使用更大的要素来开发；新工具的最佳使用；经论证的工程管理方法的最佳应用；良好的自我判断以及能够使我们认识到自己的不足——上帝所赐予的谦卑。

人月神话结束语：令人向往、激动人心和充满乐趣的50年

我依然记得那种向往和开心的感觉——当我在1944年8月7日读到哈佛大学 Mark Ⅰ型计算机研制成功的报道时——那时候我才 13 岁。Mark Ⅰ是电子机械学上的奇迹，哈佛大学的 Aiken 是它的结构设计师，而 IBM 的工程师 Clair Lake、Benjamin Durfee 和 Francis Hamilton 是它的实现设计师。同样令人向往的是读到 Vannevar Bush1945 年 4 月发表在亚特兰大月刊上的论文“*That We May Think*”的时候。在这篇论文中，他建议将大量的知识组织成超文本的网络方式，从用户的计算机上，可以跟随已有的链接，也可以跳到新的相关链接，从而实现链接之间的漫游。

我对计算机的热情在 1952 年进一步高涨，因为得到了 IBM 在纽约恩迪科特的一份暑期工作。正是那次，我有了在 IBM 604 上编程的实际经验，也了解了如何编制 IBM 701(它的第一个存储程序计算机)程序的正式指令；从哈佛大学 Aiken 和 Iverson 名下毕业终于让我的职业梦想变成了现实，并且，就这样沉迷了一辈子。感谢上帝，让我成为为数不多的那些开开心心地做着自己喜欢的工作的人之一。

我实在无法想象还有哪种生活会比热爱计算机更加激动人心，自从从真空管发展到晶体管，再到集成电路以来，计算机技术已经飞速发展。我用来工作的第一台计算机，是从哈佛刚刚出炉的 IBM 7030 Stretch 超

级计算机，Stretch 在 1961—1964 年间都是世界上运算速度最快的计算机，一共卖出了 9 台。而我现在用的计算机，Macintosh Powerbook，不但快，还有大容量内存和大容量硬盘，而且便宜了 1 000 倍(如果按定值美元来算，便宜了 5 000 倍)。我们依次看到了计算机革命、电子计算机革命、小型计算机革命和微型计算机革命，这些技术上的革命每一次都带来了计算机数量上的剧增。

在计算机技术进步的同时，计算机相关学科知识也在飞速发展。当我在 20 世纪 50 年代中期刚从学校毕业的时候，能看完当时所有的期刊和会议报告，掌握所有的潮流动向。而我现在只能对层出不穷的学科分支遗憾地说“再见”，对我所关注的东西也越来越难以全部掌握。兴趣太多，令人兴奋的学习、研究和思考的机会也太多——多么不可思议的矛盾啊！这个神奇的时代远远没有结束，它依然在飞速发展。更多的乐趣，尽在将来。

注解与参考文献

第 1 章

1. Ershov 认为，编程是一种乐趣和苦恼共存的活动。A. P. Ershov, "Aesthetics and the human factor in programming," *CACM*, 15,7(July,1972), pp. 501—505.

第 2 章

1. Bell 电话实验室的 V. A. Vyssotsky 估计一个大项目必须维持每年 30%的人员投入。这导致巨大的压力，甚至限制了根本非正式结构和沟通的改进。第 7 章已对此进行了讨论。麻省理工学院的 F. J. Corbató指出，一个长期的项目必须预见到每年有 20%的人员更替，这些人必须进行技术上的培训，并整合到原有结构中。

2. 在 International Computers Limited 任职的 C. Portman 提出："当所有的一切看上去可以工作，已经被集成时，你至少还有 4 个多月的工作需要完成。"有关其他的进度划分，请参见：Wolverton, R. W., "The cost of developing large-scale software," *IEEE Trans. on Computers*, C-23, 6(June, 1974) pp.615—636。

3. 图 2-5、图 2-6、图 2-7、图 2-8 出自 Jerry Ogdin，他引用了本章的早期版本，并对相应的描述进行了改进。Ogdin, J. L., "The Mongolian

hordes versus superprogrammer," *Info-systems*(Dec., 1972), pp.20—23.

第 3 章

1. Sackman, H., W. J. Erikson, and E. E. Grant, "Exploratory experimental studies comparing online and offline programming performance," *CACM*, 11, 1(Jan., 1968), pp. 3—11.

2. Mills, H., "Chief programmer teams, principles, and procedures," IBM Federal Systems Division Report FSC 715108, Gaithersburg, Md., 1971.

3. Baker，F. T.，"Chief programmer team management of production programming," *IBM Sys*. J. 11, 1 (1972).

第 4 章

1. Eschapasse, M., *Reims Cathedral*, Caisse Nationale des Monuments Historiques, Paris, 1967.

2. Brooks, F. P., "Architectural philosophy," in W. Buchholz(ed.), *Planning A Computer System*. New York: McGraw-Hill, 1962.

3. Blaauw, G. A., "Hardware requirements for the fourth generation," in F. Gruenberger (ed.), Fourth Generation Computers. Englewood Cliffs, N. J.: Prentice-Hall, 1970.

4. Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition*. New York: Wiley, 1969, Chapter 5.

5. Glegg, G. L., *The Design of Design*. Cambridge: Cambridge Univ. Press, 1969，其在书中提出："乍一看，用任何规则或者原理来约束创

造性思维的想法是一种阻碍，而不是一种帮助，但在实际情况下，却完全不是这样。规范的思维实际上促进而不是阻碍了灵感的产生。”

6. Conway, R. W., “The PL/C Compiler,” *Proceedings of a Conf. on Definition and Implementation of Universal Programming Languages.* Stuttgart, 1970.

7. 有关编程技术必要性的讨论，请参见：C. H. Reynolds, “What's wrong with computer programming manage-ment?” in G. F. Weinwurm (ed.). *On the Management of Computer Programming*. Philadelphia: Auerbach, 1971，pp. 35—42。

第 5 章

1. Strachey,C., “Review of *Planning a Computer System,*” *Comp. J.*, 5, 2 (July, 1962), pp. 152—153.

2. 这仅仅适用于控制程序。OS/360 项目中的一些编译器开发团队正构建他们的第三个或第四个系统，他们卓越的产品展示了这一点。

3. Shell, D. L., “The Share 709 system: a cooperative effort”; Greenwald, I. D., and M. Kane, “The Share 709 system: programming and modification”; Boehm，E. M., and T. B. Steel, Jr. “The Share 709 system: machine implementation of symbolic programming”; all in *JACM*, 6, 2(April, 1959), pp. 123—140.

第 6 章

1. Neustadt，R. E., *Presidential Power*. New York: Wiley, 1960, Chapter 2.

2. Backus, J. W., "The syntax and semantics of the proposed international algebraic language."*Proc. Intl. Conf. Inf. Proc. UNESCO*, Paris, 1959, published by R. Oldenbourg, Munich, and Butterworth, London. Besides this, a whole collection of papers on the subject is contained in T. B. Steel, Jr. (ed.). *Formal Language Description Languages for Computer Programming.Amsterdam*: North Holland, (1966).

3. Lucas, P., and K. Walk, "On the formal description of PL/I, "*Annual Review in Automatic Programming Language*. New York: Wiley, 1962, Chapter 2, p. 2.

4. Iverson, K. E., *A Programming Language*. New York: Wiley, 1962, Chapter 2.

5. Falkoff, A. D., K. E. Iverson, E. H. Sussenguth, "A formal description of System/360,"*IBM Systems Journal*, 3, 3(1964), pp. 198—261.

6. Bell, C. G., and A. Newell, *Computer Structures*. New York: McGraw-Hill, 1970, pp. 120—136, 517—541.

7. Bell, C. G., private communication.

第7章

1. Parnas, D. L., "Information distribution aspects of design methodology," Carnegie-Mellon Univ., Dept. of Computer Science Technical Report, February, 1971.

2. Copyright 1939, 1940 Street & Smith Publications, Copyright 1950, 1967 by Robert A. Heinlein. Published by arrangement with Spectrum Literary Agency.

第 8 章

1. Sackman, H., W. J. Erikson, and E. E. Grant, "Exploratory experimentation studies comparing online and offline programming performance," *CACM*, 11, 1(Jan. 1968), 11, pp. 3—11.

2. Nanus, B., and L. Farr, "Some cost contributors to large-scale programs," *AFIPS Proc.* SJCC, 25(Spring, 1964), pp. 239—248.

3. Weinwurm, G. F., "Research in the management of computer programming," Report SP-2059, System Development Corp., Santa Monica, 1965.

4. Morin, L. H., "Estimation of resources for computer programming projects," M. S. thesis. Univ. of North Carolina, Chapel Hill, 1974.

5. Portman, C., private communication.

6. E. F. Bardain 在 1964 年的一份未发表研究中指出，程序员实际的生产时间占 27%。参见：D. B. Mayer and A. W. Stalnaker, "Selection and evaluation of computer personnel," *Proc. 23rd ACM Conf.*, 1968, p. 661。

7. Aron, J. , private communication.

8. 该论文由小组分会提出，*AFIPS Proceedings* 并没有收录该论文。

9. Wolverton, R. W., "The cost of developing large-scale software," *IEEE Trans. on Computers*, C-23, 6(June,1974), pp. 615—636. 这篇新近发表的重要文章包含的数据涉及了本章所讨论的许多问题，同时也核实了生产率方面的结论。

10. Corbató, F. J., "Sensitive issues in the design of multi-use systems." 这是其于 1968 年在 Honeywell EDP 技术中心落成典礼上的公开演讲。

11. W. M. Taliaffero 在 Fortran 和 Cobol 编译器方面的生产率同样也达到了 2 400 语句/年。参见："Modularity. The key to system growth potential," *Software*, 1, 3 (July, 1971), pp. 245—257。

12. E. A. Nelson's System Development Corp. Report TM-3225, *Management Handbook for the Estimation of Computer Programming Costs*, pp.66—67.该报告指出，尽管标准有较大的背离，仍然显示高级语言使生产率提高了 1～3 倍。

第 9 章

1. Brooks, F. P. and K. E .*Iverson, Automatic Data Processing, System/360 Edition*. New York: Wiley, 1969, Chapter 6.

2. Knuth, D. E., *The Art of Computer Programming*.Vols. 1—3. Reading, Mass.: Addison-Wesley, 1968,ff.

第 10 章

1. Conway, M. E., "How do committees invent? " Datamation，14,4(April，1968), pp. 28—31.

第 11 章

1. 1932 年 5 月 22 日在 Oglethorpe 大学的演讲。

2. 一本描述 Multics 在两个成功系统上所获得经验的书籍是：F. J. Corbató, J. H. Saltzer, and C. T. Clingen, "Multics—the first seven years," *AFIPS Proc SJCC*，40(1972), pp. 571—583。

3. Cosgrove, J., "Needed: a new planning framework,"*Datamation*, 17, 23(Dec.，1971), pp. 37—39.

4. 设计变更的问题是很复杂的，本文讨论的则比较简单，具体内容请参见：J. H. Saltzer，“Evolutionary design of complex systems,” in D. Eckman (ed.), *Systems: Research and Design*. New York: Wiley, 1961. 即使所有的事都已经提出和完成，我依然提倡构建一个计划好抛弃的先导系统。

5. 该现象在以下两篇文章中进行了讨论：Campbell, E.，“Report to the AEC Computer Information Meeting,” December, 1970.J. L. Ogdin，“Designing reliable software,” *Datamation*，18, 7(July，1972), pp. 71—78. 至于曲线是否会再次下降，我的具有丰富经验的朋友们各执己见。

6. Lehman, M., and L. Belady,“Programming system dynamics,”given at the ACM SIGOPS Third Symposium on Operating System Principles ,October, 1971.

7. Lewis, C. S., *Mere Christianity*. New York: Macmillan, 1960, p. 54.

第 12 章

1. 同时请参见：J. W. Pomeroy, “A guide to programming tools and techniques,” *IBM Sys*. J., 11,3(1972), pp. 234—254.

2. Landy B., and R. M. Needham, “Software engineering techniques used in the development of the Cambridge Multiple-Access System，” *Software*, 1,2 (April, 1971), pp. 167—173.

3. Corbató F. J. , “PL/I as a tool for system programming,” *Datamation*, 15, 5(May, 1969), pp. 68—76.

4. Hopkins, M., “Problems of PL/I for system programming” IBM Research Report RC 3489，Yorktown Heights, N. Y., August 5, 1971.

5. Corbató，F. J., J. H. Saltzer, and C. T. Clingen, "Multic—the first seven years，" *AFIPS Proc SJCC*, 40(1972)，pp. 571—582. "为了达到最优性能，仅有少数几个领域的 PL/I 程序重新用机器语言进行了改写。而大多最初使用机器语言编写的程序都用 PL/I 重新进行编写，以提高它们的可维护性。"

6. 引用了 Corbató论文中的参考资料第 3 条: "PL/I is here now and the alternatives are still untested." 同时请参见一些持有反面意见的文章：Henricksen，J. O. and R. E. Merwin, "Programming language efficiency in real-time software systems", *AFIPS Proc SJCC*，40(1972) pp. 155—161。

7. 并不是所有人都同意。在一次私下的交流中, Harlan Mills 指出："我的经验开始告诉我，在产品开发中，被安排到终端面前的是秘书。其思想是使编程成为在众多团队成员监督下更加大众化的实践，而不是一项专有的技术。"

8. Harr, J., "Programming Experience for the Number 1 Electronic Switching System," paper given at the 1969 SJCC.

第 13 章

1. Vyssotsky, V. A., 1972年在美国 Chapel Hill, N. C.举办的计算机程序测试方法讨论会(The Computer Program Test Methods Symposium)的演讲"Common sense in designing testable software"。Vyssotsky 的大多数演讲都收录在：Hetzel, W. C. (ed.), *Program Test Methods*.Englewood Cliffs, N. J.: Prentice-Hall, 1972，pp. 41—47。

2. Wirth, N., "Program development by stepwise refinement," *CACM* 14, 4(April, 1971)，pp. 221—227. 同时请参见：Mills, H. "Top-down programming in large systems," in R. Rustin (ed.). *Debugging Techniques in Large Systems*. Englewood Cliffs, N. J.: Prentice-Hall, 1971, pp. 41—55

and Baker F. T., "System quality through structured programming," *AFIPS Proc FJCC*, 41-I(1972), pp. 339—343.

3. Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. London and New York: Academic Press, 1972. 该专栏包括了最完整的讨论。同时请参见 Dijkstra 早期的文章："GOTO statement considered harmful," *CACM*, 11,3(March, 1968), pp. 147—148。

4. Böhm,C.,and A. Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules," *CACM*, 9, 5(May, 1966), pp. 366—371.

5. Codd, E. F., E. S. Lowry, E. McDonough, and C. A. Scalzi, "Multiprogramming STRETCH: Feasibility considerations," CACM, 2, 11(Nov., 1959), pp. 13—17.

6. Strachey, C., "Time sharing in large fast computers,"*Proc. Int. Conf. on Info. Processing*, UNESCO (June, 1959), pp. 336—341. 参见 Codd 在第 341 页上的评论，他汇报了类似 Strachey 论文中所建议工作的进展状况。

7. Corbató,F. J., M. Merwin-Daggett,R. C. Daley, "An experimental time-sharing system,"*AFIPS Proc*. SJCC, 2, (1962), pp. 335—344. 重印于：S. Rosen, *Programming Systems and Languages*. New York: McGraw-Hill, 1967, pp. 683—698.

8. Gold, M. M., "A methodology for evaluating time-shared computer system usage," Ph. D. dissertation, Carnegie-Mellon University, 1967, p. 100.

9. Gruenberger, F., "Program testing and validating," *Datamation*, 14,7(July, 1968), pp. 39—47.

10. Ralston, A., *Introduction to Programming and Computer Science*. New York: McGraw-Hill, 1971,pp. 237—244.

11. Brooks，F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition*. New York: Wiley, 1969, pp. 296—299.

12. F. M. Trapnell 提出了一种开发良好的规格说明、系统构建及测试处理的方法，请参见：F.M.Trapnell，“A systematic approach to the development of system programs,” *AFIPS Proc SJCC*, 34 (1969) pp. 411—418.

13. 实时系统需要环境仿真器。有关举例请参见：M. G. Ginzberg, “Notes on testing real-time system programs,” *IBM Sys*. J., 4, 1(1965), pp. 58—72.

14. Lehman, M., and L. Belady, “Programming system dynamics,” given at the ACM SIGOPS Third Sympo-sium on Operating System Principles, Octo-ber, 1971.

第 14 章

1. See C. H. Reynolds, “What's wrong with computer programming management?” in G. F. Weinwurm (ed.), *On the Management of Computer Programming*. Philadelphia: Auerbach, 1971, pp. 35—42.

2. King, W. R., and T. A.Wilson, “Subjective time estimates in critical path planning—a preliminary analysis,” *Mgt. Sci*., 13, 5(Jan., 1967), pp. 307—320, and sequel, W.R. King, D. M. Witterrongel, K. D. Hezel, “On the analysis of critical path time estimating behavior,” Mgt. Sci., 14,1(Sept., 1967), pp. 79—84.

3. 有关更详细的讨论请参见：Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition*, New York: Wiley, 1969, pp. 428—430。

4. Private communication.

第 15 章

1. Goldstine,H. H., and J. von Neumann, "Planning and coding problems for an electronic computing instrument," Part Ⅱ, Vol. 1.,report prepared for U.S.Army Ordinance Department,1947;reprinted in J. von Neumann, *Collected Works*, A. H. Taub (ed.), Vol. v., New York: McMillan, pp. 80—151.

2. Private communication, 1957. 该观点发表于：Iverson, K. E., "The use of APL in Teaching," Yorktown, N.Y.: IBM Corp., 1969。

3. 有关 PL/I 的另外一个例子，请参见：A.B. Walter and M. Bohl, "From better to best—tips for good programming," Software Age, 3, 11(Nov., 1969), pp. 46—50. 相同的技术可以使用在 Algol 中, Colorado 大学的 D. E. Lang 有一个称为"STYLE"的 Fortran 格式的程序，可以达到上述效果。同时请参见：D. D. McCracken and G. M. Weinberg, "How to write a readable FORTRAN program," *Datamation*, 18, 10(Oct., 1972), pp. 73—77。

第 16 章

1. 标题为"No Silver Bullet"的论文出自：Information Processing 1986, the Proceedings of the IFIP Tenth World Computing Conference, edited by H. －J. Kugler (1986),pp. 1069—1076. 在获得 IFIP 和 Elsevier Science B. V., Amsterdam, The Netherlands 的许可后重印。

2. Parnas, D. L., "Designing software for ease of extension and contraction," *IEEE Trans. on SE*, 5, 2 (March, 1979), pp. 128—138.

3. Booch, G., "Object-oriented design," in Software Engi-neering with Ada. Menlo Park, Calif.: Benjamin/Cummings, 1983.

4. Mostow, J., ed., Special Issue on Artificial Intelligence and Software Engineering, *IEEE Trans. on SE*, 11, 11 (Nov., 1985).

5. Parnas, D. L., "Software aspects of strategic defense systems," *Communications of the ACM*, 28, 12 (Dec., 1985), pp. 1326—1335. Also in American Scientist, 73,5 (Sept.—Oct., 1985), pp. 432—440.

6. Balzer, R., "A 15-year perspective on automatic programming," in Mostow, op.cit.

7. Mostow, op.cit.

8. Parnas, 1985, op.cit.

9. Raeder, G.,"A survey of current graphical programming techniques," in R. B. Grafton and T. Ichikawa, eds., Special Issue on Visual Programming, *Computer*, 18, 8 (Aug., 1985), pp. 11—25.

10. 该主题在本书第 15 章讨论。

11. Mills, H.D.,"Top-down programming in large systems," *Debugging Techniques in Large Systems*, R. Rustin, ed., Englewood Cliffs, N. J., Prentice-Hall, 1971.

12. Boehm, B. W., "A spiral model of software development and enhancement," *Computer*, 20, 5 (May, 1985), pp. 43—57.

第 17 章

引用但未被引证的材料来自私下的交流。

1. Brooks, F. P., "No silver bullet—essence and accidents of software engineering," in *Information Processing 86*, H. J. Kugler, ed., Amsterdam: Elsevier Science (North Holland), 1986, pp. 1069—1076.

2. Brooks, F. P., "No silver bullet—essence and accidents of software engineering," *Computer* 20, 4 (Apr., 1987), pp. 10—19.

3. 许多来往信件，刊登在 *Computer* 杂志，1987 年 7 月。

"非常高兴地看到《没有银弹》没有得到任何大奖"——Bruce M. Skwiersky 的评论成为 Computer Reviews 1988 年选出的最佳评论。同时宣布上述奖项并重提了 Skwiersky 观点的还有：E.A. Weiss, "Editorial," Computing Reviews (June, 1989), pp. 283—284。不过该评论有一个重大的失误："sixfold"应该为"106"。

4. "根据亚里士多德的经院哲学中的观点，次要(accident)是不属于事物必要或者根本的属性，而是作为其他原因引起的后果。" *Webster's New International Dictionary of the English Language*, 2d ed., Springfield, Mass.: G. C. Merriam, 1960.

5. Sayers, Dorothy L., *The Mind of the Maker*. New York: Harcourt, Brace, 1941.

6. Glass, R. L., and S. A. Conger, "Research software tasks: Intellectual or clerical?" *Information and Management*, 23, 4 (1992). 作者提出了关于软件需求的度量结果是 80%的智力和 20%的文书工作。关于应用软件维护，Fjelstadt 和 Hamlen 在 1979 年得到了相同的结果。对于完整的任务而言，据我所知还没有类似的测量。

7. Herzberg, F., B. Mausner, and B. B. Sayderman. *The Motivation to Work*, 2nd ed. London: Wiley, 1959.

8. Cox, B. J., "There is a silver bullet," Byte (Oct., 1990), pp. 209—218.

9. Harel, D., "Biting the silver bullet: Toward a brighter future for system development," *Computer* (Jan., 1992), pp. 8—20.

10. Parnas, D. L., "Software aspects of strategic defense systems," *Communications of the ACM*, 28, 12 (Dec., 1985), pp. 1326—1335.

11. Turski, W. M., "And no philosophers' stone, either," in *Information Processing 86*, H. J. Kugler, ed., Amsterdam: Elsevier Science(North Holland),1986, pp. 1077—1080.

12. Glass, R. L., and S. A. Conger, "Research Software Tasks: Intellectual or Clerical?" *Information and Management*, 23, 4 (1992), pp. 183—192.

13. *Review of Electronic Digital Computers, Proceedings of a Joint AIEE-IRE Computer Conference*(Philadelphia, Dec. 10—12, 1951). New York: Ameri-can Institute of Electrical Engineers, pp. 13—20.

14. *Ibid.*, pp. 36, 68, 71, 97.

15. *Proceedings of the Eastern Joint Computer Confe-rence*, (Washington, Dec. 8—10, 1953). New York: Institute of Electrical Engineers, pp. 45—47.

16. *Proceedings of the 1955 Western Joint Computer Conference*, (Los Angeles, March 1—3, 1955). New York: Institute of Electrical Engineers.

17. Everett, R. R., C. A. Zraket, and H. D. Benning-ton, "SAGE—A data processing system for air defense," *Proceedings of the Eastern Joint Computer Conference*, (Washington, Dec. 11—13, 1957). New York: Institute of Electrical Engineers.

18. Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A.Shtul-Trauring, "Statemate: A working environment for the development of complex reactive systems,"*IEEE Trans. on SE*, 16, 4 (1990), pp. 403—444.

19. Jones, C., *Assessment and Control of Software Risks*. Englewood Cliffs, N. J.: Prentice-Hall, 1994. p. 619.

20. Coqui, H., "Corporate survival: The software dimension," *Focus*'89, Cannes, 1989.

21. Coggins, James M., "Designing C++ libraries," C++Journal, 1, 1 (June, 1990), pp. 25—32.

22. 时态是将来时，我所了解到的是，没有类似关于应用了 15 次的报告。

23. Jones,op.cit., p. 604.

24. Huang, Weigiao, "Industrializing software produc-tion," *Proceedings ACM 1988 Computer Science Conference*, Atlanta, 1988. 我觉得在这样的安排中，缺乏个人工作机会的增长。

25. 1994 年 9 月的 *IEEE Software* 全篇讨论的都是有关这些的。

26. Jones, op.cit., p. 323.

27. Jones, op.cit., p. 329.

28. Yourdon, E., *Decline and Fall of the American Programmer*. Englewood Cliffs, N. J.: Yourdon Press, 1992, p. 221.

29. Glass, R. L., "Glass" (column), System Development, (Jan., 1988), pp. 4—5.

第 18 章

1. Boehm, B. W., Software Engineering Economics, Englewood Cliffs, N. J.: Prentice-Hall, 1981, pp. 81—84.

2. McCarthy, J., "21 Rules for Delivering Great Software on Time," Software World USA Conference, Washington (Sept., 1994).

第 19 章

引用但未被引证的材料来自私下的交流。

1. 关于这个痛苦的话题，同时请参见：Niklaus Wirth "A plea for lean software," *Computer*, 28, 2 (Feb., 1995), pp. 64—68。

2. Coleman, D., 1994, "Word 6.0 packs in features; update slowed by baggage," *MacWeek*, 8, 38 (Sept. 26, 1994) , p. 1.

3. 在发布安装之后，发表了一些机器语言和编程语言命令的概率数据。有关举例可参见：J. Hennessy and D. Patterson, *Computer Architecture*. 尽管这些概率数据从不会精确匹配，但对构建后续的产品非常有用。据我所知，在产品设计之前没有任何书面的概率估计，将事先估计和实际情况进行比较就更少了。Ken Brooks 建议，Internet 上的公告牌现在提供了一种成本更为低廉的从新产品的预期用户那里获取数据的方法，即使

只有少数人会对此做出答复。

4. Conklin, J., and M. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Dtscussion," *ACM Transactions on Office Information Systems*, Oct. 1988, pp. 303—331.

5. Englebart, D., and W. English, "A research center for augmenting human intellect," *AFIPS Conference Proceedings, Fall Joint Computer Conference*, San Francisco (Dec. 9—11, 1968), pp. 395—410.

6. Apple Computer, Inc., *Macintosh Human Interface Guidelines*, Reading, Mass.: Addison-Wesley, 1992.

7. 尽管 Apple Desk Top Bus 可以从电气原理上控制两个鼠标，但操作系统并未提供类似功能。

8. Royce, W. W., 1970. "Managing the development of large software systems: Concepts and techniques," Proceedings, WESCON(Aug., 1970), reprinted in the *ICSE 9 Proceedings*. Royce 和其他人都认为软件过程自始至终不修订前期文档是不可能的；模型是作为理想情况和概念提出的。D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Transactions on Software Engineering*, SE-12,2(Feb., 1986), pp. 251—257.

9. DOD-STD-2167 重新制订的工作产生了 DOD-STD-2167A (1988)，它允许但并未制订新的模型如螺旋模型等。Boehm 的报告指出：不幸的是，2167A 所参考的军标 MILSPECS 和说明性的例子依然是面向瀑布模型的，因此瀑布模型依然继续被采用。Larry Druffel 和 George Heilmeyer 所领导的国防科学委员会(Defense Science Board Task Force)，在他们 1994 年的报告"Report of the DSB task force on acquiring defense software commercially"中曾提倡大规模使用更现代的模型。

10. Mills, H.D., "Top-down programming in large systems," in *Debugging Techniques in Large Systems, R.Rustin*, ed. Englewood Cliffs, N. J.: Prentice-Hall, 1971.

11. Parnas, D. L., "On the design and development of program families," *IEEE Trans. on Software Engineering*, SE-2, 1 (March, 1976), pp. 1—9; Parnas, D. L., "Designing software for ease of extension and contraction," *IEEE Trans. on Software Engineering*, SE-5, 2 (March, 1979), pp. 128—138.

12. D. Harel, "Biting the silver bullet," *Computer* (Jan., 1992), pp. 8—20.

13. 在信息隐藏方面，开创性论文包括：Parnas, D. L., "Information distribution aspects of design method-ology," Carnegie-Mellon, Dept. of Computer Science，Technical Report (Feb., 1971); Parnas，D. L., "A technique for software module specification with examples," *Comm. ACM*, 5, 5 (May, 1972), pp. 330—336; Parnas, D. L. (1972). "On the criteria to be used in decomposing systems into modules," *Comm. ACM*, 5, 12 (Dec., 1972), pp. 1053—1058.

14. "对象"的思想首先由 Hoare and Dijkstra 提出，但是第一个和最有影响力的案例是 Dahl and Nygaard 发明的 Simula-67 语言。

15. Boehm, B. W., *Software Engineering Economics*, Englewood Cliffs, N. J.: Prentice-Hall, 1981，pp. 83—94; 470—472.

16. Abdel-Hamid, T., and S. Madnick, *Software Project Dynamics: An Integrated Approach,* ch. 19, "Model enhancement and Brooks's law." Englewood Cliffs, N. J.: Prentice-Hall, 1991.

17. Stutzke, R. D., "A Mathematical Expression of Brooks' s Law," In *Ninth International Forum on COCOMO and Cost Modeling*. Los Angeles: 1994.

18. DeMarco, T., and T. Lister, *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1987.

19. Pius XI, Encyclical *Quadragesimo Anno*, [Ihm, Claudia Carlen, ed., *The Papal Encyclicals* 1903—1939. Raleigh, N. C.: McGrath, p. 428.]

20. Schumacher, E. F., *Small Is Beautiful: Economics as if People Mattered,* Perennial Library Edition. New York: Harper and Row, 1973, p. 244.

21. Schumacher, op.cit., p. 34.

22. 一则发人深省的海报声称："言论自由属于拥有它们的人。"

23. Bush, V., "That we may think," Atlantic Monthly, 176, 1 (Apr., 1945), pp. 101—108.

24. Unix 的发明人，Bell Labs 的 Ken Thompson 很早就认识到大屏幕对编程的重要性。他在原始的 Tektronix 电子显像管上发明了 120 行代码的显示方法。他在整个高速显像管和小型窗口的时代中坚持使用该终端。

人月神话

THE Mythical Man-Month

Essays on Software Engineering, Anniversary Edition

附录：人月落地实战体验

一、名家谈人月

1. 年金

有些书对于读者和作者像是年金，它们年复一年地分红。《人月神话》就是这么一本书。我直到在 1994 年接到 Brooks 教授的电话时，才真正完全赏识它。

原因来自他的电话。电话中他说，出版商请他修订他在 1975 年第一次出版的这本书。我立刻表示了我的一点妒忌和羡慕；因为我的出版商从未叫我修订一本出版接近 20 周年的书。确实，我甚至表示了这样的意见：当代的软件工程师不会考虑阅读这本如此古老的书，因此，可能无法卖出一本。“哦，不”，Brooks 教授回答道：“《人月神话》到目前为止，每年稳定地卖出 10 000 本。”

自此书出版至今，我已读过 1975 版至少 4 遍；在接到 Brooks 电话后，我再次将它从书架上拿下来，重读了一遍。但这次，是有特殊目的的：实际的原因来自于他的电话，Brooks 教授告诉我，目的是发现自这本书 1975 年出版以来，计算机领域是否有有意义的事件发生。

我必须申明，这样的问题是相当难以回答的。Brooks 继续告诉我，他现在已基本离开软件工程社团，将他的大部分专业精力投身于虚拟现实领域的研究。所以，在准备再版这本书时，他想知道：什么已经改变了，什么还没有？他的原书中哪些是正确的？哪些是错误的？哪些是不切题的？

当然，我不是向他提供同样信息的唯一的人：我的一些同僚、大量的业界领袖、作者、顾问和摇旗呐喊者都被邀请回答这个问题……我们所有人很高兴地做了。并且，像读者所期望的一样，我们的回答被 Brooks

教授处理、分析、过滤、综合进那个非凡的新版本——它是真正的国际财富。

原书的内容仍在那儿，现在新增了 4 章。它们是 Brooks 教授对他的思想的反思和对我们反馈的反应。新增的第 1 章由恰到好处地浓缩的原书主题组成，包括可以被称为书的中心论点的东西：大型编程项目忍受管理问题，这些问题因为劳力的分配而与小项目有本质上的差别；软件产品的概念完整性是如此关键；概念完整性是困难的但可达到的。第 2 章小结了 20 年后 Brooks 对这些主题的观点。第 3 章是他 1986 年首发于 IEEE Software 的经典报告：《没有银弹》的重印。最后一章是对 Brooks1986 年的断言“在未来十年，依然没有银弹”的思考。

年轻的软件工程师、吝啬的研究生以及懒惰的软件老手常请我推荐当前为止最好的软件图书。他们问：“如果我带着仅有的一本计算机书在沙漠荒岛上，应该是哪本书？”这是一个荒谬的问题，但人们坚持要一个答案。假如你真的被放逐到这样的小岛上(或者你决定躲藏到这样的地方去避免未来软件崩溃的恐惧)，《人月神话》应该紧随着你。

作者介绍

Ed Yourdon 是方法学大师。他的《结构化设计》、《面向对象分析》、《面向对象设计》等书两次引领了结构化分析设计和面向对象分析设计方法学的潮流。

2. 《人月神话》与实践

Adams Wang

关于《人月神话》一书，已经有了许多评论和讨论。可能，作者本身的经历——“被认为是‘IBM 360 系统之父’，担任了 360 系统的项目经理，以及 360 操作系统项目设计阶段的经理。”——已经是最好的评论。

许多朋友认为现在的软件工程数据比较理论化，可操作性不高，往往只能了解一些理念。在面对具体项目的时候，还是有些迷茫。而在整个翻译的过程中，Brooks 的观点以及治学态度经常令人叹为观止。这里，我就自己的一些体会和实践同大家探讨。

“焦油坑”(The Tar Pit)一章中，对编程系统产品的观点：“编程系统产品(Programming Systems Product)和以上的所有的情况都不同的是，它的成本高达 9 倍。然而，只有它才是真正有用的产品，是大多数系统开发的目标。”这种观点很有意义：“现实世界中的管理就是在更大程度上以人员的生命为代价，让他们更努力、更长时间地工作。经理们总是不停地吹嘘他们的人员的加班时数和能从这些人身上榨取更多时间的小把戏。”(《人件》，清华大学出版社出版，译者为 UMLChina 翻译组方春旭、叶向群)这种情况下，开发产品的质量一定会下降，甚至惨不忍睹，因为开发人员唯一能控制的是质量，当他们不得不牺牲质量，痛苦地面对自己的工作，践踏工作的乐趣时，可以想象项目成本会大大增加，并且项目的发布往往伴随着一大批程序员的倒下。

团队组建

“人月神话”(*The Mythical Man-Month*)提出了这样的论断：“向进度落后的项目中增加人手，只会使进度更加落后。”这中间还涉及如何组建你的开发团队，或者面向一个软件开发任务时，如何规划开发计划、划分任务项和分配资源。紧接着，在“外科手术队伍”(The Surgical Team)中，Brooks 提出了用外科医生加副手来组织团队，保证设计思路的完整性。其中，还提到了采用“语言专家”来帮助疑难问题的解决；安排工具维护人员，也就是通过现在意义上的系统管理员来保证系统开发、管理环境的有效运行；而其他人来解决一些文件管理等工作。

在一个小型的 C++项目操作中，我对上述方法进行了实践。

- 由 PM 和结构师一用一备来分析需求、进行框架设计，确保整个项目的概念完整性。分析设计的产出物达到框架示意代码的级别，

这部分框架代码主要是帮助团队对项目开发的理解，不存在于正式的代码中。

- 安排开发人员负责配置管理。这里的配置管理不仅仅局限于文档、软件产物的管理，而是在使用框架驱动的迭代开发时，需要对框架和各个组件不断地进行编译、整合和检查记录 Bug。这位开发人员往往是团队中的主程序员(Chief Programmer)，对各种开发方法、方法学有着一定的经验。
- 安排人员对工具进行预研，如了解 STL 类库等。该角色具体的人员在不同的阶段会进行调整。因为，他/她需要对语言、类库、开发技巧进行学习研究，往往会占用大量的工作时间。在实际情况中，前期由主程序员承担，后期由 PM 承担。

这样的安排的确能解决产品思路的一致性，在很大程度上提高生产力和产品质量。不过，它要求以下几点。

1. PM 和结构师有非常良好的沟通，包括分析问题的风格、对开发的理解等，他们之间的不一致会直接影响项目的开发方向。

2. 对分析的结果进行良好的贯彻。软件行业具有年轻的、有朝气的特点，同时也有些浮躁。有的开发人员好高骛远，对代码质量重视不够，影响框架的稳定性。

这样安排的风险在于“外科医生”的工作负荷可能过大，尤其在国内的公司中，他们往往同时兼任 PM 和系统分析的工作。同样，相应的绩效考核机制也不容易建立。

另外一个风险在于“一用一备”的安排，在“为什么巴比伦塔会失败？(Why Did the Tower of Babel Fail?)”的“大型编程项目的组织架构”中进行了详尽地论述。不过，在国内的开发环境下，似乎很难实行。比较折中的方案是对某种类型的项目做一些系统性工作，包括业务类型、开发方法和语言类型，建立开发和管理指南，从而保证概念完整性，在实践中遇到的问题在于如何贯彻实施。

上述方法潜在的一个声音是“软件重用性”。提到重用性，可能大

家脑海里马上想到的是对象、类库等。不错，面向对象的方法、商业组件(类)库的确极大地提高了软件重用性。但对于中小型企业，甚至于一个成熟的开发团队，积累自己大粒度的框架是提高软件生产力的一个重要措施。设计模式中一些模式，如 Visitor、Observe 本身就可以作为软件框架来使用。以设计模式为基础，根据开发类型积累特定业务领域的框架是完全可行的最佳实践。而 Brooks 一再强调“概念完整性”(Concept Integrity)，以及在“外科手术队伍”中推荐由外科医生来负责系统的开发，保证了产生的系统是少数人思维的结果，它们往往简捷、纯粹，没有众多人的影响，经过了项目的洗礼之后，往往能成为重用的框架。相反，在分析设计阶段，安排许多开发人员来共同开发，尽管同样完成了项目，但会相应导致“画蛇添足(The Second-System Effect)”——**“它极富有创造性，极端复杂，非常高效。但不知为什么，同时也感觉到粗糙、浪费、不优雅，以及让人觉得必定存在某种更好的方法”**，而重用的框架往往是捕捉到了问题的根本，用简单优美的方案，解决 80%的问题。

贯彻实施

Brooks 在“贵族专制、民主政治和系统设计(Aristocracy, Democracy, and System Design)”一章中，一针见血地指出了成功软件具有高度的概念一致性：**“结构师难道不是新贵？……至于贵族专制统治的问题，必须回答 “是”或者“否”。就少数的结构师而言，答案是肯定的……”**《人月神话》中的体系结构更加类似于现在的需求概念，而非现在意义的体系结构。不过，在具体的实践中，需求和体系结构依然应该由少数人来承担。这样的观点可能会引起争议，但就个人观点而言，软件行业实际上是“精英行业”。高素质、具有丰富经验的需求分析人员、结构师往往是软件企业的核心骨干人员，相应的一般编程人员流动性会大一些。

从个人发展的角度，安安静静地作为一个螺丝钉仿佛不是这个时代的精神。这个问题同企业与人之间的关系一样，很难一言以蔽之。就具

体的项目操作来说，理想情况是程序员较少地参加前期分析工作，其由有经验的同仁来负责。在较早的项目中，有的一般编程人员不安于编码实现，过早地接触一些 PM 性质工作，以及项目压力等其他因素，造成了代码质量不高，使得框架的重用程度降低。而在其他项目中，由于企业的文化，同事们严格按照分工进行开发，提高了质量。后者看似少了机会，但从长远的角度看，大家对设计的了解更加透彻，对流程的理解更加深刻，为以后的发展打下了基础。

同样在这一章节中，Brooks 提出了“在等待时，实现人员应该做什么”的问题。他认为：“首先，必须设定良好定义的时间和空间目标，……同时，在物理实现的级别，也有很多可以着手的工作。”实际项目中，早期的投入往往人员较少，一般 15 人月左右的项目，初期就 2～3 人，这其中包括了需求分析、设计人员以及主程序员。主程序员会对系统有初步的了解，对系统开发采用的技术、工具和开发技巧进行研究，同时与 PM 一起负责搭建软件工程环境和软件开发的环境。这样的安排会导致一个问题，即编程人员与分析设计人员的沟通。它需要大家通畅、自由地交流，尽可能地对开发达成共识，减少信息的丢失。

关于项目交流的方法，Brooks 就 OS360 开发经验，在“贯彻执行”(Passing the Word)一章中进行了具体的讲解，如会议与大会、多重实现、电话日志等。就小型项目而言，如果不考虑异地开发的情况，较正式的组内会议可以安排一周一次，即周例会的形式。另外，在里程碑之处安排评审会议，以对开发的进展、需求的实现以及项目计划进行调整和修正。而对于异地开发，或者用户不在本地的情形，则建议采用电话会议等形式，效果虽然不如本地开发好，但总是聊胜于无。“为什么巴比伦塔会失败？”(Why Did the Tower of Babel Fail?)又对软件开发中的交流问题进行了强调，指出了文档是沟通的一种重要方式。后续的“提纲挈领”(The Documentary Hypothesis)则用类比的方式阐述了如何定义软件项目的文档集合，“另外一面”(The Other Face)讨论了程序文档的一些形式。这些观点对实际工作也具有指导意义，相应的 Rational Suite 中

Requisite Pro 工具使用参考中，推荐了软件项目的参考文档集合。在 C++ 和 Domino 若干项目的实践中，发现最小的文档集合包括需求文档(Software Requirement Specification)、项目计划(Software Development Plan)、框架设计和设计元素说明。其中，项目计划不仅仅是进度，而是从软件的配置管理、生命周期、资源分配、风险分析、培训等方面进行描述，这部分的内容往往不局限于单个项目，而是在同一种类型的项目中具有共性。因此，可以在不同项目中共享。

总而言之，Brooks 对文档的建议是：**“项目经理聪明的做法都是立刻正式生成若干文档作为自己的数据基础，哪怕这些迷你文档非常简单……如果一开始就认识到它们的普遍性和重要性，那么就可以将文档作为工具友好地利用起来，而不会让它成为令人厌烦的繁重任务。通过遵循文档开展工作，项目经理能更清晰和快速地设定自己的方向。”**

面向变更的开发

“稳定状态的生产思想特别不适合项目工作。我们倾向于忘记这一点：项目的全部目的就是让自己死亡。项目生命中的唯一稳定状态是死后僵硬……”《人件》(*Peopeware*)；

“软件开发是减少混乱度(减少熵)的过程，所以它本身是处于亚稳态的。软件维护是提高混乱度(增加熵)的过程，即使是最熟练的软件维护工作，也只是放缓了系统退化到非稳态的进程。”——“未雨绸缪(Plan to Throw One Away)”

软件开发本身就是一个具有无序趋势的活动。当人们四处游走，寻找一个类似于计算机硬件那样流水线化的方式方法时，可能应该静静思考一下，这本身就与开发内在发展规律相悖。并且，软件开发是人类的思维创造活动，同诗歌、乐曲一样，好像人类历史上还没有为上述创造性工作进行流水线化的尝试。

从软件开发产物的角度而言，CMM 规范 2 级中的配置管理一般包括四个活动：配置标识、版本控制、变更控制和度量。配置标识和版本

控制都是为了给变更提供一个稳定的承载基础。而 Brooks 对 360 机器的开发进行了描述："在 System/360 工程模型中，在一大堆常规的黄颜色电线中，常常可以不经意地看到紫色的电线束。……这些更改过的接线使用紫色电线，看上去就像伸着一个受了伤的大拇指。"这一段体现了配置管理的雏形，作者进一步提出了在软件项目中"……软件开发也需要用到'紫色线束'的手法。对于最后成为产品的程序代码，它更迫切地需要进行严密控制和深层次的关注……而且需要文档化。"

在 CMM 试点项目以及后来的项目实践中，为项目的文档产出物进行标识是比较容易实现的。互联网上有大量的资料和模板可供参考。版本控制可以使用 VSS 或者 Open Source 的 CVS，而 CMM 规范中反复出现的"基线"(baseline)概念，在配置管理实践的初期或者小规模的项目中并不是强制性的。当配置标识和版本控制实践到一定的程度之后，再推广基线和变更控制，就相对容易得多。

所以对于某种类型的项目，可以采用制定配置标识规范、识别文档类型、确定该类项目的目录结构、建立版本控制机制来进行初期的实践，具有一定的成熟度之后，再实施变更管理。变更使用"阶段(量子)化、定期变更……量子(阶段)化变更方法非常优美地容纳了紫色线束技术：直到下一次系统构件的定期发布之前，都一直使用快速补丁；而在当前的发布中，把已经通过测试并进行了文档化的修补措施整合到系统平台。"这种方法能在一定程度上缓解项目压力、变更和质量之间的矛盾。

迭代开发

Brooks 在"20 年后的人月神话"(The Mythical Man-Month after 20 Years)明确提出了迭代开发的概念，即**"增量开发模型更佳——渐进地精化"。这种开发方法对项目产生的激励作用是不可估量的，作者在北卡罗来纳大学时，"常常会被屏幕上第一幅图案、第一个可运行的系统对团队士气产生的鼓舞效果而感到震惊"**。而在 C++的现实项目中，在极大的压力下，当第一个可执行的原型出现在开发人员面前时，长期的

疲惫、沮丧一扫而空。从而，为继续开发提供了动力。这里，长时间高负荷工作并不是被提倡的，但增量开发是软件开发动力学的一种重要手段。

迭代开发可以用“外科手术队伍”、OO 及设计模式的实现来实践。面向对象框架的分析设计思想，并不一定非要用面向对象的语言实现。它的一个重要特点是，固化用户要求或者是待开发系统中相对稳定的部分，以牺牲模型某个维度的代价来得到较稳定的模型，然后在该框架的基础上不断地迭代。“外科手术队伍”则由具有丰富经验的“外科医生”操刀，来主持需求分析和建立迭代框架。少数的人能确保框架不带有过多不必要的非结构性的功能特色，从而保证框架的简洁和具有良好的扩充性。

迭代开发本身是对变更这个“怪物”的一剂良药，不同迭代周期能较好地容纳阶段(量子)化的变更。在变更的同时，始终有可运行的系统供调试、测试，从而确保项目的质量——质量决定成本，**“远远超过最终用户需求的质量是一种取得更高生产力的手段(《人件》)。”**

同样，Brooks 在“未雨绸缪(Plan to Throw One Away)”中提出了抛弃型原型的概念，并在“《人月神话》的观点：是或非？(Propositions of the Mythical Man-Month：True or False？)”中强调：**“因此，为舍弃而计划，无论如何，你一定要这样做。”**迭代开发是容纳原型，包括抛弃型和非抛弃型。而且，它是一个很好的说服管理层和用户的途径——因为，不同的迭代周期和任务本来就在计划之中。在 C++的项目中，对于没有坚持在框架设计完成之后抛弃框架代码原型，至今还有些遗憾，相应所付出的代价就是产品质量和开发人员对该项目信心的丧失。

另外一个提高项目产品质量的方法，Brooks 在“整体部分(The Whole and the Parts)”的“系统集成调试”中给出了专家意见，**“使用经过调试的构件单元”、“搭建充分的测试平台”、“一次添加一个构件”和“阶段(量子)化、定期变更”**。这在目前的实践中依然有指导意义。现在的软件测试水平相信业界人士都非常清楚：很少有规范化的测试；白

盒测试基本不做；项目压力过大，不断压缩测试时间……因此，仔细的系统集成测试非常有必要，能够做到《人月神话》中 OS360 系统测试的一半已经非常不错了。

项目计划

项目计划的重要性相信每个人都了然于胸。Brooks 在“祸起萧墙”(Hatching a Catastrophe)一文中提及了项目计划/跟踪。另外，在其他的许多章节中，阐述了计划文档化的重要性。这里，项目计划不仅仅指的是项目进度，采用 Microsoft Project 所画出的仅仅是项目的进度。在具体的实践中，项目计划可以从以下方面考虑。

- 项目描述：包括项目定义、名称、背景、目标与范围、交付物和验收标准等。
- 项目组织结构：定义项目人员组成(这部分内容会根据项目的实施不断调整)。
- 软件生命周期：依照项目特点决定合适的周期。并不是所有的项目都适用于迭代周期，也不是说瀑布模型就一无是处，甚至有的项目需要瀑布模型作为主干，在某个阶段加入迭代特性。
- 项目管理：包括客户管理、进度管理、成本管理、风险管理和培训管理等。
- 配置管理：定义项目目录结构、产物标识方法和版本控制等。

项目计划必须仔细地考虑用户、项目特点，它可能是寥寥数语，但是它的作用不在需求规格说明之下。项目计划和需求规格说明是项目文档集合中最基本的强制性文档。

软件开发中，理论与实践结合是一个亘古不变的话题。我们有幸诞生在一个信息共享的时代，能够接触到前人睿智的思想。**“这个神奇的时代远远没有结束，它依然在飞速发展。更多的乐趣，尽在将来。”** Brooks 如是说。

3. Frank Chance 评人月

出版于 1975 年的《人月神话》是软件开发方面的经典作品。1995 年版包括了令人感兴趣的新的几章，但原来的随笔依然是这本书的心脏与灵魂。在这本书中，Brooks 解决了如何组织和管理大规模编程项目的问题。这些项目要求成百上千的程序员产生几百万行代码(想想 SAP、Oracle 数据库引擎、Windows2000)。这部书由一系列简明的随笔组成。在这篇评论中我将讨论开篇随笔——我的最爱之一。

焦油坑

Brooks 将大系统编程比做史前的焦油坑，来开始他的第一篇随笔：**“记忆中，我们看到恐龙、猛犸象、剑齿虎正在挣脱沥青的魔爪。挣扎得越剧烈，陷入得越深，没有哪只野兽足够强壮或熟练，它们最终都沉没了。大系统编程在过去的十年间就像焦油坑，许多大而强有力的野兽在其中已经惨烈地失败了。大部分已实现并在运行的系统，很少有达到目标、时间表和预算的。大和小、厚重和细实，一个接一个的团队卷入了沥青(陷阱)。没有什么事情似乎会导致这个困难——任何特殊的手掌都能被拉出来。但同时并相互作用的因数的相互聚集导致运动越来越慢。每个人似乎都惊讶于问题的难缠，难于面对它的本质。”**

记住，这些话写于 1975 年。今天它们仍然可用吗？考虑一下 WindowsNT5.0。第一次计划于 1997 年发布，随后延迟到 1998 年早期，再者是 1998 年末，然后是 1999 年(为此它被重新命名为 Windows2000)。这里是一些公开的估计：

- 5 000 个程序员
- 35 000 000 行代码

显然，NT5.0 是个大系统编程项目。同样显而易见，Brooks 的焦油坑在今天同样普遍！

让我们继续 NT5.0 的例子。假设最糟糕的情况，全部 35 000 000 行

代码都是新编的。有理由假设开发工作大致在 1994 年开始。所以我们有：

- 5 000 程序员×5 年= 25 000 程序员年
- 35 000 000 行代码/ 25 000 程序员年= 1 400 行/程序员年

如果你是一个程序员，或者你只接受过编程课程的教育，似乎感到这个数字(1 400 行每年)令人惊异地低。我们当中的大部分人都能在一两天内堆积出接近 1 000 行的代码。什么使得 Microsoft 的程序员一整年才产出 1 400 行代码？

两种可能性跃入我们的脑海：

- Microsoft 雇用了 5 000 名不合格的程序员去开发 NT 5.0；
- 写一个大规模的程序系统产品远难于堆砌出单一的程序。

Brooks 讨论认为后一个答案是正确的。他由定义术语开始。

(a) 程序

一个独立的程序是我们两天编程狂欢的结果。它是准备自己运行于我们编程的那台机器上的。如果我们加上文档、通用化代码和编写测试用例，使得代码可以由其他无关的编程人员来维护，我们就有了程序产品。

(b) 程序产品

如果我们接受我们的程序，并且完整地定义了它的接口，使得它达到预定义的规范，并且测试了它和大量的其他组件的交互作用，我们就有了程序系统组件。

(c) 程序系统组件

如果我们都做了(加上文档、通用化代码、编写测试用例、使得代码可维护、定义了接口和测试了交互作用)，我们就有程序系统产品组件。

Brooks 用手边的三倍规则说明在上述每个步骤中的工作要求：

b＝3 倍 a 的人力

c＝3 倍 b 的人力

c＝9 倍 a 的人力

换句话说，开发一个独立的程序仅仅要求开发一个程序系统组件的 1/9 的人力。

回到 Microsoft 的例子，如果我们将这个 9 倍的因子乘以 1 400 行/程序员年的生产力测量，我们得到 12 600 行/程序员年(举例来说，假设我们掌握每一程序员，并且使得他们独立工作，堆砌在单一的程序上)。在一篇独立的随笔中，Brooks 引用一个发现这点的经理的话，平均他的每个程序员仅能将他的一半时间用于开发——其他时间由文书工作、会议和各种其他任务占据。把这些因素考虑到 Microsoft 的例子中，我们达到了 25 200 行/程序员年。那么，Microsoft 的程序员看来非常可敬。另一个测量自 1975 年来有了很小的改变，Brooks 引用的估计是 1 000 行/程序员年。如果上面引用的 1 400 行/程序员年是精确的，那么，它表现了在 1975 年到 1995 年 20 年间，生产力仅仅提升了 1.75%每年。这个结果证实了 Brooks 的另一个假定——程序员的生产力相对是一个常量，它不受开发所用的语言的影响。因此，实际的生产力收获来自于迁移到高级语言编程，这些语言每行表达了更多的实际工作。尽管目标是大系统项目，Brooks 的解释常常被广泛地应用。例如，这个第一篇随笔用标有“手艺的快乐”和“手艺的悲哀”的小节来结束。在悲哀中，他讨论了荒废的问题：“……**这个人们已经工作了很长时间的产品，显然在完成前将被废弃。同事和竞争者已经在热烈地用新的和更好的主意反击。人们的孩童般想法的取代已经不仅仅在构思，而且付诸时间表。这一切总是似乎比它的实际更糟糕。新的和更好的想法通常在完成之前不被应用；它仅仅被谈论。真老虎永远不能和纸老虎相比。**”

小结

Brooks 的随笔涉及了大系统编程所固有的多种挑战，但对任何投身于软件开发的人来说读这本书都是有用的。“人月神话”这一篇讨论了许多编程任务的不可分割性，和为什么增加人力到软件项目中无法产生效用。我的另一篇最爱是“贵族、民主和系统设计”(概念完整性的讨论)和“计划和投放之路”(在付运前多次交付的明确计划的益处)。一些问题已经因为技术的进步而废弃，例如关于如何在一个大型团队中分发写

好的文档。然而，你可能惊讶 Brooks 面对的许多问题今天如何阻止我们。另外的益处是 Brooks 简洁、清晰的作品读起来令人愉快。如果你是一个程序员，如果你和程序员一起工作，如果你管理程序员，你应该阅读这本书。

4. 软件尚方宝剑(Silver Bullet)何在

——Fred Brooks 和 Brad Cox 的不同观点

欧阳进(中国台湾)

前言

二十年来，人们一直寻找解决软件危机的方法，包括结构化、人工智能和物件导向等方法；但软件大师 Fred Brooks 在 1986 年发表的文章里预言在 10 年内找不到解决软件危机的尚方宝剑，历经了十年，果然不幸被他言中。1995 年，Brooks 与 Cox 两位大师分别深思软件尚方宝剑，Brooks 仍持怀疑态度，而 Cox 则相当乐观。

本文以时间的前后，依序交叉介绍两位大师的见解，期能引起读者对软件未来的些许兴趣。由于笔者的学识和功力远不如两位大师，只能介绍他们文章里的精华，并未添加笔者的阐释或见解，以免您受到笔者有限能力的误导或局限您的思想和视野。希望您有空细读两位大师的原文著作，培养您自己的思维和创意。

介绍

二十年前(1975)，IBM 大型电脑之父——Fred Brooks 出版一本书叫 *The Mythical Man-Month*，收集了他在 20 世纪 60 年代领导 1 000 多人共同发展 OS/360 大型软件系统的心得和经验。从实际经验中，他体会到开发大型软件过程中，难以汇集参与人员的设计理念，然后提供给使用者一致的设计概念(conceptual integrity)，因而导致软件的高度复杂性，

最终使得大型软件系统往往会进度落后、成本暴涨及错误百出，就是所谓的软件危机(software crisis)。

经过了 10 年(1986)，Brooks 发表了一篇著名的论文——*No Silver Bullet：Essence and Accidents of Software Engineering*。

他断言："在 10 年内无法找到解决软件危机的尚方宝剑(银弹)"(There will be no silver bullet within ten years)。

这文章激起许多软件专家的讨论与争辩，而"No Silver Bullet"也成为脍炙人口的名词。Brooks 认为软件专家所找到的各种方法皆舍本逐末，解决不了软件的根本困难——即概念性结构(conceptual structure)的复杂无法达到概念的一致性，软件自然不亲切不好用！

到了 1990 年，曾首先提出"Software IC"名词的 OO 大师 Brad Cox 针对 Brooks 的观点发表了一篇重要文章——*There Is a Silver Bullet*，说明他找到了尚方宝剑——即有些经济上的有利诱因会促使人类社会中的文化改变(culture change)，人们会乐于去制造类似硬体晶片(IC)般的软件组件(software component)，将组件内的复杂结构包装得完美，使得组件简单易用。由这些组件整合而成的大型软件，自然简单易用；软件危机就此化解了。

在 1995 年初，Brooks 的上述名著第二版出炉了，书中含有一篇关于尚方宝剑的新文章——*No Silver Bullet Refired*。文章里，Brooks 赞扬 Cox 的文章，但他认为 Cox 误解了他的本意。

尽管历经了十年，软件开发方法也有所进展，但 Brooks 仍然认为尚方宝剑仍未出现；大型软件系统的开发工作仍然困难重重，只能渐进地改善，不要奢望短期内会出现尚方宝剑，一举解决软件的困境。

在 1995 年底，Brad Cox 发表了新文章——No Silver Bullet Reconsidered。这篇文章里，Cox 更深刻地探讨文化和思维变迁(paradigm shift)的话因与效果，从人类文化的观点阐释软件危机的原因，而得到乐观的结论——软件的尚方宝剑是可能的。

过去 10 年，Brooks 的预测是千真万确的，果真软件的困境不但未

解决，反而更加严重。那么下个十年又会如何呢？有趣的是，大家来欣赏一下两位大师的深思灼见，然后再看看您个人的见解，也许会激发出许多新创见也说不定，不是吗？

1975 年

Brooks 在 *The Mythical Man-Month* 书中的看法如下。

该书是论文集，其中有一篇文章叫 *The Mythical Man-Month*，他就以此作为书名。1956—1965 年，Brooks 领导 IBM 360 大型电脑的开发计划，包括硬体结构及庞大的 OS/360 作业系统在内，为此他有“IBM 大型电脑之父”之尊称。由于 OS/360 是多达 1000 位程序员共同合作的大型软件开发工作，让他深刻了解到大型软件开发的技术和管理上所面临的种种困难和挑战。于是，他就将其领导开发 OS/360 软件系统的经验心得收集在这本书里。

人们常拿 Man-Month(多少人，做多少个月)来计算软件的工作量，但是 Brooks 发现软件的开发工作是需要人与人之间密切沟通的，使得设计工作不易分割，所以 Man-Month 为单位的计算方法是有问题的(mythical)。也就得出著名的 Brooks 法则——“对于进度已落后的软件开发计划而言，若再增加人力，只会让其更加落后。”(Adding manpower to a late software project makes it later.)

这是该书名称的涵义。

Brooks 从经验中深深感觉到参与人员之间的概念一致性(conceptual integrity)是软件设计中最重要的一环(conceptual integrity is the most important consideration in system design)。一旦软件设计师们有一致的设计概念，其所共同创造出来的软件才有可能达到简单(simplicity)及好用(straightforward)的效果。

为了达到这个效果，Brooks 主张应注重软件的主架构(architecture)，并将它与实施细节(implementation)分开来。所谓主架构，就是对使用者界面之完整而详细之叙述(the complete and detailed specification of the

user interface)。主架构汇集各软件设计师的不同思绪，融合成为一致的概念，然后呈现给使用者，让使用者感觉软件是源自于单一的设计理念(single philosophy)，而不是基于庞杂无序的众多思绪。将主架构与施行细节分开来，是让大型软件计划获得一致性概念的有力途径(separation of architectureal effort from implementation is a very powerful way of getting conceptual integration on very large projects)。

1987 年

Brooks 在 *No Silver Bullet* 文章中的看法如下。

自从 *The Mythical Man-Month* 一书上市以来，历经了十年，尽管软件专家们努力研究新方法来解决软件的困难，但是 Brooks 认为这些方法，包括高阶语言、OOP、AI 等皆舍本逐末，只解决了一些概念的表达(representation)技巧，并无法解决根本性的概念结构(conceptual structure)问题。在 *No Silver Bullet* 一文中，他借亚里斯多德(Aristotle)的用词而将软件之困难分为两种：

- Essence——此为概念上(conceptual)的根本困难；
- Accidents——此为将概念转换为电脑表示法时，在表达(representation)上的困难。

例如，OOP 上的抽象资料型态(即类别)观念，只是改进了设计的表示方式，所以抽象时只去除掉表达性(accidents)的复杂和困难，但并未去除掉任何根本性的复杂。

由于没办法解决这种根本性的困难，使得原本单纯可爱的软件逐渐演变为进度落后、成本暴涨、错误丛生等，像恶梦中的狼群般蜂拥而至，于是哀号而希望有种“银弹”(silver bullet 又译为尚方宝剑)能即刻平息它们。然而 Brooks 认为：“不仅眼前找不到尚方宝剑，由于软件的本质使然，未来也不太可能找得到(Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any)。”

Brooks 列出的根本性困难包括以下几点。

- 复杂性(complexity)——“复杂”是软件的根本特性，可能来自于程序员之间的沟通不良，而产生结构错误或时间延误；也可能因为人们无法完全掌握程式的各种可能状态；也可能来自新增功能时而引发的副作用，等等。
- 一致性(conformity)——大型软件开发中，各小系统的界面常会不一致，而且易于因时间和环境的演变而更加不一致。
- 易变性(changability)——软件的所处环境常是由人群、法律、硬件设备及应用领域等各因素融合而成的文化环境，这些因素皆会快速变化。
- 不可见性(invisibility)——软件是看不见的，即使利用图示方法，也无法充分表现其结构，使得人们心智上的沟通面临极大的困难。

这些是软件的本性，Brooks 认为没有捷径可以解决(there is no royal road)，但可以渐进式地进行改善。他认为可行的方案有：

- 买来装配(buy and build)——软件的建造尽量使用现有零组件，不要一切都从头做起。
- 快速雏型(rapid prototyping)——使用重复式的开发方法(iterative development) 来渐进地改善软件的雏型，以求逐步厘清使用者的需求。
- 有机成长(growing organically)——生命皆是由小慢慢长大的，建造大型软件的方法应该是逐渐成长的，而不是一次建造完成的。
- 优秀设计者(great designer)——人是软件设计的核心，良好的方法可以改善人的创造过程，但无法激励人的原本创造力，其需倚重有创意的人。

1990 年

Cox 在 *There Is a Silver Bullet* 文章中的看法如下。

Cox 在文章开头就提到 Brooks 在 *No Silver Bullet* 一文中的悲观看法，Cox 写道：

And in "No Silver Bullet: Essence and Accidents of Software Engineering" he(Brooks) argues that the difficulties are inevitable, arising from software's inescapable essencenot from accident, but from some deficiency in how programmers build software today.

(在"NSB"文中，Brooks 声称这些软件的困难是无法避免的，是源自软件上无法逃避的根本——不是来自一些旁技细节，而是今日程序员设计软件时的缺点。)

接着，Cox 针对 Brooks 所提出的软件困境所在，从新的观点来进行阐释，从而得到乐观的结论。

Cox 认为软件危机是必然会面临(irresistible)的困境，但并非是无法克服(immovable)的。当全球经济进入资讯时代，市场对资讯殷切需求而产生的经济诱因(economic incentive)，将促成人们社会中的文化改变(cultural change)。加上目前遭遇的软件困境：成本过高、品质不良，以及人们无力去改善它们，使得人们对软件的看法和思维产生巨大的变迁(paradigm shift)，从过去一行一行撰写指令的土法炼钢方法转而尝试去建立可重复使用的标准组件(standard component)以及组件交易市场(market)。在这种新环境中，软件的价值体系(value system)、权力结构(power structure)，以及软件设计师与消费者关系的重新定位，彻底从过去重视程序的编写过程(process) 转而重视组件价值、拥有及买卖系统等基本文化要素，而导致软件产业的革命(software industry revolution)。

软件组件类似于硬件的晶片(IC)，一般把复杂的组件封装起来。对使用者而言，调组件再也不复杂了。因此，建立软件晶片市场，让人们不再受困于组件内程序编写的复杂，而专注在组件本身的使用、价值和买卖上，软件的复杂和困难就自然烟消云散了。

因此，文化和思维的变迁将是一把尚方宝剑，能克服软件的困境；而且这种变迁是即将来临的，并非海市蜃楼！

1995 年

Brooks 在 *No Silver Bullet* 文章里的看法如下。

自从 1987 年 Brooks 发表 *No Silver Bullet* 一文之后，激发了许多人对到底有没有尚方宝剑的讨论与争辩，例如前述的 Cox 之 *There Is a Silver Bullet* 文章，以及 Harel 的 *Biting the silver bullet* 文章等。

Brooks 在理解各方的反应，以及检验当今各种软件设计方法之后，他仍认为尚方宝剑尚未出现(the magic solutions are not just around the corner)。

Brooks 赞扬 Cox 的 *There Is a Silver Bullet* 文章是一篇极佳的文章，可是他认为 Cox 误会了他的本意，误解之处有二：

(1) 如上节所述，Cox 在文章中写道："...but from some deficiency in how programmers build software today."

Cox 把根本性困难解释为软件建造方法上的缺失。事实上 Brooks 所谓的根本性困难是指软件上概念性的复杂所导致的，无论使用什么方法，无论在任何时刻，皆必然呈现，这是软件的固有特性。

(2) 如上节所述，Cox 在文章中写道："...Brooks argues that the difficulties are inevitable..."

事实上，Brooks 认为他自己只是持怀疑之态度，并非认为尚方宝剑是完全没有希望的。例如，Brooks 认为软件系统的复杂是可分层次(level)的，若软件下层的小组件(object)组成上层的大组件，再由大组件组成更大之组件，层次分明地组织起来，则复杂度就可以降低。还有，当软件由小而大渐进地有机成长时，虽然复杂有系统地增加，但可确保软件的正确性。导致这种困难的原因仍可渐进式地改善或去除，只是没有捷径罢了。

Brooks 认为 OO 是一个有希望的途径，但由于牵涉人们思维方式的转换，使得企业组织必须先投下资金(front-loaded cost)，才能逐渐回收其利益(down-stream benefits)，因而 OO 的进展会是慢如蜗牛。即使它可解决软件危机，也不会是在眼前。

1995 年底

Cox 在 *No Silver Bullet* 文章中的看法如下。

Cox 认为 Brooks 所谓的根本(essential)困难——complexity、conformity、changability 及 invisibility——并非是最根本的，而是由一个隐藏在深处的病因(cause)所造成的症状(symptoms)罢了。这个病因就是：目前文化中缺乏有利的诱因来鼓励企业公司生产、买卖软件的组件(component)以及更小的子子孙孙组件(subcomponent)。

如果买卖各层组件者皆有利可图，自然会想尽办法将组件封装得简单易用，则 Brooks 所指的软件困难自然消失。

Cox 觉得 Brooks 的观点是以技术为核心(technocentric)，而他的观点则是以人为核心(human-centric)。由于观点的差异，对软件危机的阐释自然会获得不同的结论。他的结论是：尚方宝剑是可能的，只是这把剑是软件思维的变迁(paradigm shift)，而不是技术(technology)。

想一想，造铅笔的公司卖掉了一支铅笔，就少了一支铅笔，亦即少了这铅笔的各小组件，如铅笔擦、笔心等。为了要再造另一支铅笔，就得付费去购买铅笔擦和笔心，使得制造笔心等小组件的公司也有利可图。因此，从铅笔到铅笔擦、笔心等制造者皆会用心去把组件包装好，使其简单易用！

但是，在软件中，制造试算表软件的公司，卖出一套试算表软件，只是一个拷贝动作，并不会少掉一个位元；不必再向小组件的制造者购买，使得小组件制造者无利可分，自然无法进行生产、包装和买卖了；这是目前软件业的困境。

在未来新文化之中，使用者(end user)每使用一次某物件(object)，就必须付一次费用给这物件的生产者，而这生产者则必须付一次费用给其内部各次物件(subobject) 的生产者，依此类推到最小的组件为止。在此环境中，各层次物件之复杂结构皆封装得完义无缺，使得各层物件皆简单好用，软件的复杂与困难就迎刃而解了。

结论

在前言中已提到，本文只是介绍 Brooks 与 Cox 的文章精华，笔者因能力所限而不做推论，当然不会有结论。盼您在细读过这两位大师的原文后，做出您自己的结论。如果您愿意，也盼您有所结论时，能与笔者分享，自当感激不尽！

二、名著评人月

(一) 中译本引用

我唯一一本读过一遍以上的书，是 Fred Brooks 的《人月神话》，实际上我每过一两年都会重读一遍。部分原因是这本书文笔很好，另外就是书中的忠告很有价值，即使是在 40 年以后。当然，很多细节上的地方与我们做事情的方法有所不同。我们的工作更自动化，计算机的“马力”更强劲，但书中依然有许多好的忠告，因此，我非常推崇这本书。这是我唯一能想起来的能从中体会到乐趣和思想的计算机科学书籍。

“给我看你的流程图而藏起你的表，我将仍然感到莫名其妙。如果给我看你的表，那么我将不再要你的流程图，因为它们太明显了。”

我想说的是，数据结构设计是程序构造过程的中心环节。一旦数据结构安排好了，算法就会瓜熟蒂落，编码也比较容易。(《程序设计实践》中译本，P.48)

—— Brian Kernighan，名著《C 程序设计语言》的合著者之一(与 Dennis M. Ritchie 合作)

一颗濒于坍塌的行将死亡的恒星，一个正在识字中的小孩，一群对病毒发起攻击的白细胞：这些只不过是物质世界中少数涉及巨大复杂性的对象。软件也可能会牵扯很复杂的元素，但这里所看到的复杂性却是根本不同的类型。正如 Brooks 指出的：“爱因斯坦认为肯定有对自然的更简单的解释，因为上帝并不是反复无常的和独断专行的，但这样的信念并不能安慰软件工程师。他所要对付的复杂性绝大多数都是任意性的。”(《面向对象分析与设计》第 2 版中译本，P.1)

——Grady Booch，IBM Fellow，UML 创始人之一

它(《人月神话》)仍然是计算机书籍中被引用次数最多的书籍，而且即便本书最初出版于 1975 年，其内容至今仍未过时。在阅读的时候，每隔几页不说一句“对极了”是很难受的。(《快速软件开发》中译本，电子工业出版社，P.271)

——Steve McConnell，Construx 公司首席软件工程师，名著《代码大全》、《快速软件开发》的作者

Frederick P. Brooks 在 1987 年发表了一篇《没有银弹——软件工程的本质与故障》的文章，后来这篇文章成为了软件工程领域中最具影响力和最负盛名的文章……Fred Brooks 指出，我们的软件行业正卷入一项长期的探索之中，即要找到消除低生产率这个恶魔的神奇银弹。亚力山大大帝流泪了，原因是他已找不到没有被他征服过的世界。(《代码大全》第 2 版中译本，电子工业出版社，P.687)

——Steve McConnell，Construx 公司首席软件工程师，名著《代码大全》、《快速软件开发》作者

在 20 世纪 80 年代，著名的软件工程思想领袖 Frederick Brooks 博士(《人月神话》的作者)阐述了瀑布方法的缺陷，以及使用 IID(迭代增量开发)方法的必要性。(《UML 和模式应用》中译本第 3 版，第 2 章，“迭代、进化和敏捷”，P.29)

——Craig Larman

Fred Brooks 说，“要计划丢弃某些东西”。1988 年，我把 Ruby 卖给比尔·盖茨，当时只是一个原型……接下来要把它和比尔的 QuickBasic 结合，成为 Visual Basic。我做的第一件事就是抛弃 Ruby 的原型，除智慧和经验外，一切从零开始。(《软件创新之路》中译本，第 3 章)

——Alan Cooper，Cooper 公司创始人，交互设计大师

让你思考 4 个变量(成本、时间、质量、范围)的一些故事。周年版中还有一段关于著名的文章 *No Silver Bullet* 的有趣对话。

—— **Kent Beck，极限编程(XP)创始人**(以上文字出自他的名著 *Extreme Programming Explained：Embrace Change, Second Edition*)

在 1987 年发表的那篇经典文章《没有银弹：软件过程的本与末》中，Frederick Brooks 雄辩地指出了需求在软件项目中的重要地位。(《软件需求》第 2 版中译本，第 1 章，P.11)

——**Karl Wiegers，Process Impact 公司首席咨询顾问，软件工程大师**

这是一本经典著作，与软件开发有关的每一个人都应该不止一遍地读这本书。(《Rational 统一过程引论》中译本，P.226，作者在书中把《人月神话》列在推荐书籍的首位)

——**Philippe Kruchten，Rational 统一过程首席架构师**

敏捷实践并不是真正的新事物，而是一种新的态度和更好的包装。在这些实践中，许多都有悠久的历史(Fred Brooks 在 20 世纪 50 年代中期就使用了结对编程)。

敏捷方法和计划驱动方法都不是能够杀死 Fred Brook 提出的“软件工程狼人”的银弹。它们都有一些要素可以看做是“铅弹”。(《平衡敏捷和规范》中译本，P.12、P.104)

——**Barry Boehm，美国国家工程院院士，AIAA、IEEE、ACM 会员**

(二) 国外原版书引用

可能这样说有点轻描淡写：设计 Closure Tools 是为了解决许多恰好处在设计核心的问题。其中许多工具来源于我们在 Gmail 的朋友。一开始，Gmail 是一个比较朴实的 JavaScript 应用，然后，他们添加了越来越多的特性，看着它长大到无法再控制或维护。Frederick Brooks 有一段描

述大型系统编程的话，“就像焦油坑，许多大而强有力的野兽在其中已经惨烈地失败了。”类似 JavaScript 这样的语言，是一个高度动态的环境，里面几乎所有的东西都可以修改，如果没有标准方式来阐述契约(类型检查或其他)，即使只有一小队开发人员，焦油也会快速杀来并淹没他们。

Michael Bolin, Closure: The Definitive Guide, O'Reilly Media, 2010

在著名论文《没有银弹——软件工程中的根本和次要问题》中，《人月神话》的作者 Fred Brooks 做出了如下声明：在未来的十年内，无论是在技术还是管理方法上，都看不出有任何突破性的进步，能够独立保证在十年内大幅度地提高软件的生产率、可靠性和简洁性。

在软件开发行业，“没有银弹”的概念已经深入人心。人们随意用它来表示没有任何新技术会成为软件设计和生产率问题的最终解决方案。然而Rails经常被某些人错认为是银弹，因为它的生产率肯定比.NET或Java高得多。正如Brooks在文章中说的，有两种复杂性，即偶发的和本质的。前者由开发人员和方法引起，后者是解决问题所固有的。

Antonio Cangiano, Ruby on Rails for Microsoft Developers, John Wiley & Sons, 2009

在《人月神话》中，Frederick Brooks建议，当你面对新概念或新技术时，你应该计划“建造一个要丢弃的系统”，这样，你的第二个系统就是你真正要用的系统，就可以吸取做第一遍时的教训。

显然，这不总是可行的，我们没有必要鼓动你折腾你当前项目的代码。我们的意思是有时你发现自己面对这样的项目，各种不同的思想和技术混杂在一起，以至于打补丁、添加测试或消除冗余就像给断腿贴创可贴。

G. Ann Campbell, Patroklos P. Papapetrou, SonarQube in Action, Manning Publications, 2014

顾客可能只需要500个特性中的5个，但顾客出于业务需要还需要另外5个目前没有的特性，这个欲望却被当做不合理的要求而被拒绝。

从闭源工业的这个视角，他们的态度是可以理解的。在《人月神话》中，Fred Brooks做出判断“当工作量线性增加时，项目的复杂性以及所耗费的沟通成本却随着开发人数的增加而成倍上升。”没有基于社群的开发方法，很难交付比上一代产品有少许增量改进的产品，最坏的情况就是补丁的集合。

Russell Bryant, Leif Madsen, and Jim Van Meggelen, Asterisk: The Definitive Guide, O'Reilly Media, 2013

正如你将在第 3 章和第 4 章看到的，当你实际去实现 UML 状态机时，你一直能完全控制执行的次序，因此被 UML 语义强加的严格限制是不必要的。类似地，状态图需要大量的符号(也就是伪状态，例如结合、分叉、联合、选择点等)来图形化地表示控制流。这些元素就是经过伪装的老式流程图，它很久以前就被结构化编程技术证明没什么用。换句话说，在用于比较简单的结构化代码时，图形符号在表示控制流方面没有增加更多的价值。你可以在《人月神话》中找到 Brooks 对流程图的评论。

Miro Samek, Practical UML Statecharts in C/C+, Second Edition, Elsevier, 2008

对于软件行业来说，将他们与其他工程专业人士、涉众和顾客讨论专业实践的方式稳定化，是本质的。在试图阐明技术挑战或设计技巧的价值时，说外来方言是不慎重的。

Fred Brooks教授在讨论巴比伦塔的时候谈到了不同软件词汇集导致的混乱。他的书《人月神话》指出了不断扩张的软件方言导致的泥潭。该书第一次出版以来，在最短的时间跨度里，软件文化中已经散播了更多的

的趋势还在继续，使得传统工程职业和软件工匠以及工程印象派之间的沟通障碍变得更宽。

Richard Schmidt., Software Engineering: Architecture-driven Software Development, Elsevier, 2013

不是所有 IT 项目都是软件开发项目——很多是现场系统、半定制、全定制的混合——但他们确实代表了某种特有的困难。

直到 20 世纪 90 年代，项目管理才在软件项目里有用武之地。曾经的 IBM 人 Frederick Brooks 在他的《人月神话》中写了软件项目管理的挑战，第一次指出了估算的困难(包括“乐观偏见”)，还有“在项目往错误的方向发展时增派资源只会适得其反。”

Peter W.G. Morris, Reconstructing Project Management, John Wiley & Sons, 2013

在著名的《人月神话》中，Fred Brooks 谈到了“第二系统效应”，即在系统的第二个版本中糟糕设计决策带来的过度工程和污染。他指出，这个问题是因为遗留系统的维护者通常是非常年轻的工程师，害怕挑战“高手”的设计决策。他建议的解决方案是，让年轻工程师参与到新的系统设计中来(和有经验的工程师一起)，让有经验的工程师负责遗留系统。但从我们非正式的观察结果来看，这很少发生。因此，再教育和调整态度是两个主要的重构。

Phillip A. Laplante, Colin J. Neill, Antipatterns: Identification, Refactoring, and Management, Auerbach Publications, 2005

用例、用户故事或积压条目提供了项目范围的宽泛定义。很多团队把这样的工件考虑为业务用户、PO 或顾客的责任。询问业务用户提供范围的效果，依赖于那个没有软件设计经验的人能否给我们一个高层次

的解决方案。设计一个解决方案是最有挑战性和最关键的步骤之一。现在，我们务必要回顾一下 Fred Brooks 在《人月神话》中的话：“搭建一个软件系统最困难的部分是决定究竟要做什么。正如爱因斯坦所说的，问题本身常常比解决它的方法更重要。”

Gojko Adzic, Specification by Example: How Successful Teams Deliver the Right Software, Manning Publications, 2011

正如 Fred Brooks 在 1986 年的精彩描述——没有银弹，适当的设计依然是开发人员的责任。例如，使用 OSGi 框架来达到模块化，然后使得每件事情公开。我们将在后续章节讨论模块化，因此不要担心目前细节还不清楚。要理解的重点是 OSGi 框架提升了模块化，从而减少管理大型项目和提升复用的复杂性。

Alexandre de Castro Alves, OSGi in Depth, Manning Publications, 2011

TradeProcessor 是一个领域对象。作为一个设计人员，你应该只考虑它如何与其他领域对象和服务的行为协作，以完成对客户的承诺。实例化和管理服务的生命周期不是这个领域抽象的核心责任。

这个例子展示一个领域对象如何被设计用于处理非本质复杂性。这些事情由更低层的架构栈来处理更好。Fred Brooks 把它称为“accidental complexity”，有些人更喜欢用术语“incidental complexity”。

Debasish Ghosh, DSLs in Action, Manning Publications, 2010

这些是哲学问题或美学问题。我会坚持，拼图块的结合看起来是最美丽的，可能也是最正确的。把这些小块整编起来的术语我勉强称为“测试架构”。看起来，《人月神话》作者 Fred Brooks 和我的意见一致。在 2007 年的 OOPSLA (一个面向对象的软件会议)上，他把架构师的角色定

义为两部分：

(1) 首先，代表顾客的利益；

(2) 为了做到这一点，需要从头到尾了解整个产品。

Tim Riley and Adam Goucher, Beautiful Testing: Leading Professionals Reveal How They Improve Software, O'Reilly Media, 2009

三十多年以前，Brooks 认识到开发大型复杂软件的困难，并把大型软件系统的开发比喻成一只巨大的野兽在焦油坑中挣扎。把握软件的特定部分相对容易，但要把整个软件拉出焦油坑几乎不可能。过了十年，Brooks 又写到，事情不仅没有变化，甚至连出现“银弹”的机会都没有。银弹指可以大幅度提高生产率的一种方法、一款软件工具、技术进展或管理技能。

V.S. Alagar, K. Periyasamy, Specification of Software Systems, **2nd edition,** Springer, 2011

在其标志性著作《人月神话》中，Fred Brooks 强烈而有说服力地说明，概念完整性是可靠系统设计的关键。只有由少数的头脑聚集到一起设计出系统的架构才能带来概念完整性。本书第 5 章和第 7 章展示如何创建一个架构来达到其行为和质量需求。

在编程中，术语“架构”一开始用来指可以应用于多个系统的，关于计算机系统的描述。今天这层意义还在使用。1969 年，Fred Brooks 和 Ken Iverson 把架构称为“程序员所看到的计算机概念结构”。几年后，Brooks 在《人月神话》中把架构定义为“用户接口的完整和详尽的规格”，仔细划分了架构和实现。他引用 Blaauw 的话，“架构告诉我们发生什么，实现告诉我们如何使它发生。”这个区分在今天的面向对象编程时代依然存在，而且茁壮成长。

Len Bass, Paul Clements, and Rick Kazman, Software Architecture in Practice, Second Edition, Addison-Wesley Professional, 2003

如果用尽处理能力，应用能跑多快？和《人月神话》中的 Brooks 定律类似，应用只能被分割这么多次。虽然添加更多的处理器确实不会像向项目添加人手一样让作业慢下来，但对速度也没有帮助。1967 年 Gene Amdahl 提出的 Amdahl 定律说明，程序中不能被并行化的部分的比例将限制并行化的总体速度。换句话说，程序的非顺序部分受益于并行化，其他部分却没有。

我们要讨论的下一个和最后一个因素是推动团队规模扩张的因素。通常来说，这个因素是出于业务需求的需要。业主和项目经理要建造更多和更大的面向顾客的项目，以便持续抛开竞争对手，培育新的收入流，提高顾客基数。大项目需要保持团队小规模。首先，依赖于使用的产品开发生命周期方法学，更多迭代和更多开发时间。净效应是一样的，项目需要更长时间来交付给顾客。

其次，在职工程师数量的增加需要更多的行政支持人员，包括经理。工程师经理被称为行政支持人员听起来有点冒犯，但实际上这就是管理的实质——支持团队完成项目。团队越大，需要的经理越少。上面说的这些，和 Frederick Brooks 在《人月神话》中说过的类似。

Martin L. Abbott, Michael T. Fisher, The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Addison-Wesley Professional, 2009

增量开发是一个允许早期原型、早期集成和测试的开发范型。在项目的早期允许增量开发，路上的每一步系统都有能正常工作的完整功能，尽管仅占所有功能的一小部分。Fred Brooks 在 1995 年写了关于系统第

一次成功做某些事情时对团队士气产生的“兴奋效应”。没有增量开发，要么所有东西都能运转，要么没有东西能运转，相当于退化到了开发的瀑布模型。整个系统的子集的用处不只在于开发，它们为每一个经历的事件提供了安全的反馈：对于项目经理来说，在交付时提供给顾客系统的一个工作子集，比起道歉和承诺来说要好太多了。

Paul Clements, Felix Bachmann, and Len Bass, Documenting Software Architectures: Views and Beyond, Second Edition, Addison-Wesley Professional, 2010

有时候，突击就是没效果。有些任务无法在更短的时间内完成，不管给它分配了多少资源(就像妇女生小孩)。很多情况下，增加更多的资源反而使工期延长。新人需要时间来跟上速度，所以在他们的生产率低于原来的团队时，成本增加了。问题还会更复杂，因为原来的资源会被牺牲，因为其不得不帮助新人。添加更多的人也意味着更多的管理和沟通成本。Frederick Brooks 在《人月神话》中有更详细的描述。

Bonnie Biafore, Microsoft Project 2010: The Missing Manual, O'Reilly Media, 2010

如何将缺陷修复和更广泛的软件开发过程结合？如何估算需要花多长时间来修复一个缺陷或软件当前的所有缺陷？如何确保项目不会像 Brooks 在《人月神话》中生动描述的那样，陷入无尽缺陷修复的焦油坑？

Paul Butcher, Debug It! Find, Repair, and Prevent Bugs in Your Code, Pragmatic Bookshelf, 2009

应该无情地重构代码。还可以再拔高一个级别，就像 TPM 运动在 50 多年前的创新一样：一开始就打造更好维护的机器。使代码可读和使代码可执行一样重要。1960 年前后，TPM 圈子引入了终极实践，聚焦

于引进新机器或取代旧机器。正如 Fred Brooks 所劝告的，每隔七年我们可能应该重做一遍软件的主要代码块，或者清除慢慢腐烂的代码。也许我们应该更新 Brooks 的时间常数，把每七年改成每七周、七天或七小时。

Robert C. Martin, Clean Code A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008

三、读者感言

——关于《人月神话》20周年纪念版

1. 读书有感——人月神话

May 29，2004

看到小女子介绍《人月神话》这本书，请不要以为是罗曼史小说或者传奇故事。

它是一本二十多年前出的、讲三十年前软件专案管理问题与经验的书，原名是 *The Mythical Man-Month*。其中的很多事例，现在仍存在；其中很多经验，现在拿来还很有指导意义。

我们曾经接过急迫且大型的网站建置项目，但是进展上还算顺利，这是由于系统工程师十分有经验，在分工、次序与掌控上都做了很好的安排。

但是最近的一位朋友遇到了网站机制改版的问题，在时程与资源运用上不断延迟。还好这是内部机制，不用向客户交代，否则蒙受的将不仅仅是金钱上的损失。

我们知道时程延迟，但原因在哪里？为何会发生？

《人月神话》的第二章，应该是这本书的精华，也是书名的由来。看完这个章节，我发现作者道出很多软件专案上的管理关键，而软件专案是不可以完全套用传统产业的管理方式的。

第一个关键 乐观

首先作者提出了一件常常发生的事情，这也是一个在我朋友身上发生的故事："程式设计师都是乐观的家伙。"而且我发现，越年轻越乐观，但是无可避免的这是个年轻的产业，很少会遇到很有经验的程序设计人员。这些年轻人，即使我们告诉他这样是会有问题的，他也不会有任何改变。

所以"他们所犯的第一个错误是假设一切都会进行得很顺利"。可是没有想到一个 bug，就可能花上一天时间也无法解决。然后，就延迟了后面本来假设可以顺利进行的部分。

第二个关键 人月

人月，是我们预估和排定时程用的，但是作者提出一个前提："使用人月必须要在人力与工时可以互换的状况下，而且要当工作可以切割，投入工作的人不用沟通时，人力与工时才能互换。"就是说要可以互换，才能使一个人做 30 天，与 30 个人做一天的结果一样，不然单纯用人月去估算时程，一定会有误差。

为什么呢？首先软件项目有其连续性。作者举了个例子，他说，生小孩就是要九个月，你叫多少个妈来生都是一样。第二点是因为即使工作可切割，但是需要沟通，越多人投入，就需要越多的沟通与教育的时间，所以一个人做 30 天的工作不可能用 30 个人做一天来完成。

所以，Brooks 说，在一个时程已经落后的软件项目中增加人手，只会让它更加落后。

因此，要让一个项目顺利进行，首先要有良好的时程规划，考虑所有的因素；其次，程序人员要有勇气，不要妥协，一定要坚持自己预估的时程。就像是厨师一样，即使外面的客人在等着上菜，一只鸡应该烤多久就要烤多久，不能因为妥协就用大火，这样只会将鸡烤焦。

外科手术团队

这个篇章看完突然想到研究所的项目管理课程，三个学分，却是我们一个学期的研究重心；四个实际运作的项目，大家分组进行。教授不断地制造项目危机，比如公司被并购、人员流失、项目形式改变和客户不断施以压力等，但最后大家还是顺利地结案。

这是个难以忘怀的经验，组员们的不够默契在起初的确造成危机，但是，由于大家的素质还算整齐，很快就建立沟通的语言。加上大家熟悉协同科技概念与网络沟通，可以做到分时分地作业，也使项目进行得比较顺利。

但当我看完这篇外科手术团队后，我发现其实还有个成功的关键，就是同学们各有专长，所以分工很容易；又因为有选出领导者进行掌控，所以不会乱。大家不会做重复的事情。只要每个人按照时程交差，工作就可以顺利完成。

作者认为，在一个外科手术中，有操刀的大夫，有护士、麻醉师等，各司其职，而不是像屠夫团队，每个人都得拿刀。一个团队中只有一个人操刀，其他人扮演支援的角色。这样整件事情就会出自一个脑袋，也不需要不断地沟通协调。

读这些篇章我比较有感觉，写出来与大家分享。

最后，作者还有一个“人月”的概念很重要，这会影响到组织的公平性：一个好手，花一天可以做完一件事，但同样一件事，庸才却需要花三四天以上，甚至加班赶工也不能完成。如果照这样去计算人月，一定会差之千里。

2. 我这几天很烦(产品概念完整性)

2004-06-18

我这几天很烦！

一是因为现在做的项目处于测试阶段，由于一些原因，导致现在发现了很多关于模块交互方面的问题。要将这些模块“组装”成一个像样的系统，这些问题必须解决，目前只能靠自己与其他开发人员商量解决！

这个项目在设计的时候有 7 个人参与，应该是开发部所有的开发人员。当时先是讨论整个框架的设计，然后将各个模块分下来，一人完成一两个，再接着开会讨论。讨论来讨论去，讨论的差不多了，也就是每个人将自己设计的模块接口写成电子表格(模块内部的设计有些有文档，有些没文档，因为每个模块的设计人员基本上就是这个模块的编码人员，所有其他人不会太过关心)，然后每个人就开始实现自己的模块。后来因为人手不够，又招了几个写代码的人员，分给各个模块的负责人员协助编码。事实上在真正进入编码阶段时模块的负责人只剩下三个，那么其他人呢？两个离职，一个是公司领导，不参与实际编码，还有一个因为考博，没时间亲自参与开发。而剩下的三个人中有一个人是这个开发小组的组长，协调开发小组的开发工作，当然也不参与实际的开发。

可以说在整个开发过程中没有一个真正的产品负责人(或者称之为总设计师)，用于负责产品的整体结构设计、各模块的交互关系、功能设计和实现方案的取舍。这个人必须十分了解整个产品及设计方案，他可以不写代码，因为维护整个产品设计方案的工作贯穿于开发全过程，工作量是很巨大的。他也就是维护产品概念完整性的那个人！

其实，除了缺乏真正的产品负责人外，存在的问题还有很多！近期我会抽时间把它整理下来，寻找问题的根源到底在哪里，以避免再犯同样的错误！

还有，我去年买的《人月神话》，当时没看懂。前段时间拿出来，我终于把它完整地看完了，而且感触颇多。

为什么《人月神话》在 20 多年后还能再出新版，而且又一次引起业界巨大的轰动，就是因为我们没有吸取前人在 20 多年前总结的经验教训，还在一次次地犯同样的错误！

……

这个项目是一个二次开发包。完了之后紧接着有很多项目都要依赖它。二次开发包做成这样，不难预测以后依赖此开发包的项目又将使人进入新的“焦油坑”。我现在也不知道什么时候开发第二个版本，至少今年不太可能。

加班！加班！

3. 关于我们的思考——“项目开发”及读《人月神话》有感

项目开发终于结束了。

按项目流程，我应该写一份《项目开发总结报告》。拿来“GB856T—88”标准文档，有框架指导我该写些什么，但是怎么能让这些框架束缚了自由的思想呢？于是我决定换一种形式。

项目开发接近尾声的时候，也是最关键的时候，有人送来一本《人月神话》。我很幸运能在这个时候读这本书，因为它带给我很多思考，关于项目，关于团队，关于我们……

……

四、完美与放弃

在《人月神话》中有一段被截取，称为“程序员的苦与乐”，在网上广为流传。Brooks 大师用简短的篇幅，描绘出整个程序世界的苦与乐。而在这次项目开发中，我有两个苦恼体现的比较明显。

一是追求完美。“因为计算机是以这样的方式来变戏法的：如果口语中的一个字符、一个停顿没有与正确的形式一致，魔术就不会出现(现实中，很少的人类活动要求完美，所以人类对它本来就不习惯)。实际上，我认为，学习编程最困难的部分是将做事的方式向追求完美的方向调整。”

看来我再次做出了正确的选择。我是个追求完美的人，而且我也一直认为程序员就应该有这种本性。在写程序的时候，甚至对一个变量名我都要反复斟酌，直到选择一个我认为最合适的。我并不认为这是在浪费时间：一是有助于对程序的理解和维护，好的程序本身就是注释；二是可以减少错误发生的可能。这次开发因为时间短，我尝试采用设计→编码→编译的方式来写程序，经常把一个几千行代码的模块写完之后才开始调试。这种方式效果不错，因为对设计考虑得比较充分，设计上没出现什么问题，基本上都是一些拼写上的错误。不过有一个错误却令我苦恼了很久。因为一个结构的成员变量名与函数参数的变量名一样，而这个参数又在多处使用。在写的时候，也可能在拷贝代码的时候，很容易忘记结构名或多加了一个结构名，而这时又不会有语法上的错误。吸取了这次教训，我把整个程序检查了一遍，并做了一些修改。这段程序我参考了一位大师的原型，令我欣慰的是，他的下一个版本和我的程序一样做出了同样的一些修改。

另一个苦恼来自“设定目标、供给资源和信息提供”。编程人员很少能控制工作环境和工作目标。《人月神话》后面的章节还提到：“结构师获得了所有创造发明的快乐，剥夺了实现人员的创造力”；“实现同样是一项高级的创造性活动。在具体实现中增加创造和发明的机会，产品会因为指定了外部技术说明而大为减少，相反创造性活动会因为规范化而得到增强，整个产品也一样。”程序员要学会放弃一部分乐趣，整个项目也一样。这是我读《人月神话》感触最深的一点。

设计网络认证的时候，我本打算用一种简单的认证方式实现，因为这不是重点。Y 提出用证书加自己写的 Socket 类实现认证的通信过程。这是一个很好的创意，但 VC 写出的类无法在 Delphi 里使用，BCB 也不行。封装，回调……大家都搞晕了，只剩下两个字：“放弃”。

由于时间比较紧，在产品的功能上我们也放弃了一些我们认为是很有创意的，但需求中没有提到的部分。后面的开发中也有类似的问题，每个人都按自己的喜好调用和提供接口，可以说是八仙过海，各显神通。

问题主要在于交流。

五、善于交流

人月之所以不能成为神话，正是因为增加人手的同时也增加了人与人之间的沟通成本。

我们在项目一开始就通过 QQ 群组进行沟通，后来又搭建了企业内部协作平台，而且还有不定期的会议。这与大师所说的三种交流途径(非正式途径、会议、工作手册)不谋而合，但是执行的效果不很理想。是大师说的不对？不是。是我们没有利用好这几种途径。我们很好地利用会议，解决很多细小方面的误解；过多地使用 QQ 群组，一是在登录 QQ 的时候消息太多而没看仔细，二是这种方式针对小的误解是很有效的，但对于一些系统的全面理解，则必须通过文档；程序员们很少使用协作平台，没有仔细阅读文档的习惯，也没有写出一个完整文档的习惯。

即使在 QQ 群组的交流中，我们也没有把问题表述清楚。经常看到："某某，你那里有问题？""哪里？什么问题？"接着有人回答："不可能啊？"什么事情都有可能发生，只要能满足条件。域名都会不翼而飞，还有什么是不可能的呢？

……

九、结束语

本文只是我在项目开发和读了《人月神话》之后的一点感想，对于本文一些观点的正确与否欢迎与您一起讨论。本文只是按照项目流程，挑主要的部分叙述了我在开发中体会到的一些感想。这次开发让我学到太多太多的东西，将使我终身受益。还有很多的想法，如团队建设、整体设计、开发模式……希望以后能与大家多多交流，相互学习。

《人月神话》这本书我推荐大家都去读一下。它不是良药，不能为你解决实际工作中的问题，但它可以给你提供很多思考的空间，让你变得更加成熟。软件工程是为开发软件服务的，标准不是目的，只是手段。《人月神话》没有讲标准，但它为怎样做好一个项目提供了一些启发。

它会增强你的自信心，当有人反驳你的观点的时候，你可以告诉他："Brooks 大师如是说！"

4. 我的"人月神话"

……

"人月神话"这个题目来自于《人月神话》一书，内容也来自于《人月神话》一书，只不过换成了我所经历的。

我编写代码已经数年了。说到编写代码的能力，我还是比较自信的。但是，领导终究是领导，始终会有没有学过管理或者尝试以流水线理论来主导软件开发的领导。在国内，这样的领导并不少见，其共同之处在于：目标产品是定下期限要完成的，所有的东西是可以以人手定量的，而人员配备"按照正常情况足够满足开发进度"。在这一前提下，形形色色的变种会出现。

有些是不能明确说出功能列表；有些则是没有能力知道开发中可能遇到的障碍点在哪里；有些则不知道团队人员的真正能力；更有些是集以上之大成。在有些时候，我也是其中的一员。

说到这里，我要简单阐释一下：我们所有的进度都是以"人月"代码产量来衡量的，而实际情况是，增加"人"并不能缩短"月"的量。

一个目标产品本身能有多大的代码量大致不会和预算的相差很多。这是我的经验，当然也有一些连代码量也估算不准的 LEADER。我们通常最终会将代码量分解到每个模块，并且根据程序员的工作能力来分配进度要求。在很多情况下，遇到进度失衡的时候，第一反应是增加人手。但是事实上增加人手的项目只有不到 10%能准时完成。很多情况下，增加进去的人手并不能真正进入工作，因为模块已经无法细分一小块出来给新加入的人手。

而人月代码产量本身就不是一个固定的值。我的最高编写记录可以达到 1 600 行/天。那么，真的就是 32 000 行/月了吗？不！代码产量一

般在 200～300 行/天。很多时候，一个算法就会花费 1 天，这样就会出现 100 行/天的情况。根据最近 3 年的状况，5 000 行/3 月是比较客观的量。这是 C/C++的速度，也是我的速度，其他程序员有这样的效率吗？真正能超过的并不多见。即使是这样的代码效率，也不适合将计算纳入商业产品的进度进行考虑，因为很多难点并不是因为降低人月代码产量就能够攻克的。

我本人目前比较倾向的时间分配，即比较真实的、没有难点的时间分配具体如下：

20% 代码编辑

30% DEBUG

30% 文档

20% 保留时间

这就说明即使在没有已知难点的状况下，有 20%的保留时间仍然有必要。因为很有可能 1 个小小的数学逻辑就能让你忙上一天。这并不是不专心，是疏忽导致的，而从来就没有人能避免疏忽。

了解了这个神话，我们就可以采取主动行动。

1. 首先，不要低估任何一个产品的难度，难度估计得高点总是没有错的。(我曾经犯过多次这样的错误)这样，在确定任务进度前要争取更多的时间。

2. 很显然，既然有可能在任意时刻发生问题，为什么不提前多干点儿呢？很少有人愿意这样。但是我的经验是一定要提前多干。在最近的 2 个项目中，我都是提前很多时候完成了大部分的工作。90%的东西完成了，而产品交付时间则剩下 1 个月。眼看可以轻松了，却仍然忙着攻克最后的难点，到了最后一天才真正完成任务。险得很！按照时刻表完成进度的程序员一定会翻船。

《人月神话》中写到，好的程序员可能效率比糟糕的程序员高 10 倍。在我的人月神话中确实有这样的例证。当时团队中就有人一天无法完成一个极度简单功能的 PROGRAMMER，但是在人月理论中，这样的

人也照样要占着进度表的一条。

5. 《人月神话》软玉生香

刘天北　文

从尺寸说起

憨豆先生(Mr. Bean)(编注：著名喜剧片的主人公)赞许惠斯勒的《画家母亲的肖像》，首先谈的一点就是："这幅画比较大(quite big)，所以了不起(excellent)"。这种批评套路，中国古已有之：品茶时赞美"热得好"就是。今天我们谈谈 Brooks 的 *The Mythical Man-Month*(中文译为《人月神话》，以下简称 MMM)，也效法英国来的专家，从尺寸说起：这本书的一个突出优点，在于它比较小。

小有两点，一是开本，二是页数。尼采说，我的野心是用一句话说完别人十本书才能说清的事(他还加了一句——甚或是，十本书都说不清的事)。可是对于出版业，这样的野心家不要也罢。多数书长，还是因为所涉及的领域本身头绪众多，不过总可以用得上法国人的俏皮话：他没有时间往短里写。把书写短不是人人都可以办到的，这需要良好的分寸感和结构感，而我们很难在匆忙的 IT 从业者中找到这样的素质。所以一般技术书籍，多为人高马大，卷帙浩繁(当然还有国内影印版的惯常做法，把原来的大开本缩印成"小兄弟"，但考虑到增加的阅读难度，结果并没有想象的那么经济)。"大"，对于书——尤其是技术书——来说，不见得就"了不起"。以我个人的阅读习惯，多于 500 页、开本又超过"大 32 开"的技术专著，只能当字典用，本来一卷在手，有如软玉生香在抱，总以"轻"、"薄"为妙；若是庞然巨物，体态狼伉，未免让人产生心理障碍(借此一角向通读了 *C++ Primer* 的各位致意：你们都可以去做《大内密探零零发》里艳福齐天的皇上了)。眼下说的这本 MMM，正是技术书中的"赵飞燕"，可作字面意义上的掌上舞：大 32 的开本，算上注释、

索引共 332 页，要紧的是，每章前的大量篇幅留给题图和题记，19 章下来，实在要读的内容不到 200 页。

如果决定读这本书，你也许可以参照我个人的经验做出时间预算：并非出于固定的读书计划，而是为了“谋杀”早晨巴士上的无聊时间，我选中了这本相貌可人、插图众多的 MMM。在前一周的周日下午我兴味盎然地翻阅了书末的三章：后记(“The MMM after 20 Years”)、作者著名的单篇论文《没有银弹》(*No Silver Bullet*)和《再论没有银弹》(*No Silver Bullet Refired*)；这之后，第 1 章以下的若干章大约以每次 1 章半到 2 章的速度平均分布在 5 次车中颠簸的早高峰时间(每次约 35 至 40 分钟，并有一次坐过站)；最后，一个睡眼惺忪的周六上午让我完成了其余部分——我没法相信，这种“休闲”的阅读形式适用于任何一本其他的技术“名著”(可能的例外是 *Peopleware* 中文名为《人件》，由清华大学出版社出版，译者为 UMLChina 翻译组方春旭、叶向群，我们一会儿还会谈到它)或许你明早会带上一本 *Planning XP* 试试。

美味的分析

那么，除了小，这本书还有什么好处？对，再就是：它比较好读。我看书也纯粹是“以貌取人”，外表即是一切(法国人说：皮肤是人最深刻的部分)。就像饭菜，“好吃”就好，营养等在我辈老饕看来，还是第二义，而出于“精神食粮”这话的本意，“好读”也该是对书的最高赞许。

好读有两层意思：读起来容易，读起来可口。小书不都易读，易读的书也不都可口，但对于一本技术书而言，把自己的主题讲清楚了，而且还易读，也是了不起的成就。好些人写书，一味在“内在美”上下工夫，殊不知，“可接近性”才是首要的一环：我们不也说“可口”、“可人儿”吗？“可”(也是 accessibility 里的 ability)最要紧。

回到正题，谈谈全书的结构：我手上这本 MMM 的 20 年纪念版(Anniversary Edition)，共 19 章(另有简短的前言——初版和再版各一篇——和尾声)。其中，第 1 章到第 15 章是初版内容，最长的一章 17 页，最短的

第5章刨去题图和题记，不过薄薄两张半纸而已。作者称这些章节为“随笔(essays)”，每章各自处理相对独立的主题(“人月”问题、软件项目的人员构成、开发中的交流、文档、排错等)，因而你基本可以不顾及章节间的联系，单独从你感兴趣的任何段落入手。书末的四章里，第18章“Proposition of MMM：True or False?”是对之前各章所有观点的一个列表——一个非常便于翻检和温习的设置；其他的三章(对于“银弹”的两篇讨论和后记)篇幅在20～40页，值得专门对待。对于一本初版于多年前的技术书籍，作者并没有逐条修订原有的内容，而选择了重印这些章节，同时在后记中根据二十年来的技术上和认识上的变化检讨得失。

现在，你大概能模糊地看到这本书的样子：很薄，随手翻上去每章很短，并包含大量整页的插图(从侧面看，书页的颜色黑白参差)。换言之，可读性首先来自于外表，你拿起它，马上就感到美味当前。

然后你会注意到书的风格。关于原版技术书的语言风格，我曾经在别处做过一个区分：可读的和可翻译的。“可读的”风格主要出现在用母语写作的作家笔下。他们能够自如地使用语言中的元素，几乎总能找到对特定概念最合适的表达。这些表达，往往植根于该语言自身，因此很难转换为另一种语言。比如本书作者给讨论“系统空间限制”的一章的标题是“ten pounds in five-pound sack”。对任何具备高考英语水平的读者，这里的意思都非常直观，但是谁能轻易地给出它的中文等价物？“削足适履”，还是“五磅的麻袋装十磅”？另一方面，“可翻译的”文本则多为移民作家的产物。对于一个概念，他们缺乏“自然的”表达方式。他们在用别人的语言，一串串的词像临时借用的家具一样摆放那里。你可以简单地逐字翻译这样的文本，甚至可以采用机器翻译而不会犯大错误，但是无论译或不译，都很难马上明白他说的是什么。当然，这只能是对实际情况的最简化的0阶模拟(一些喜欢在行文中加入俚语的本土作家几乎不能归入任何一边)。但是，当我说出“可读的”时，我首先想到的就是Brooks和另一位大师Martin Fowler。二者的风格具有不少共同之处：亲切、不卖弄，能用最普通的英语说明(有时是令我们的汉

语绝望的)复杂问题。

本书的作者 Brooks(也许应该说“小”布鲁克斯才对，因为他的全名是“Fredrick P. Brooks，Jr.”)，曾担任 IBM 的大型机 System/360 及其操作系统 OS/360 的开发项目经理，被称为“S/360 之父”，之后转入大学，执教多年，因而难能一见地在实际项目管理和教学上都具备世界级水平。在离开 IBM 近十年之后(1975 年)，他根据大量切身实践和教学经验写出了本书，在风格上，也兼具教师的醇厚耐心和工程师的技术素质。我个人最受益于书中随处可见的精彩比喻。仅仅举几个例子：“焦油坑”、“大教堂”、“外科手术队伍”、“银弹”。就我记忆而言，几乎每一个都是对所指对象最贴切的形容。而这些隐喻的光彩，又通过众多题图、题记(有时不失幽默)的反射，获得了熠熠生辉的效果。

在很多情况下，过分追求“幽默感”也会产生止咳糖浆式的效果。典型的例子是 Peopleware。与 MMM 相比，那本书总是给我一种推销员教材的印象。作者技术背景(我不认为 DeMarco 们曾参加过严肃的软件项目)、宗教背景的不同(Brooks 是一位虔诚的基督徒)也许可以解释卖弄和朴实、空洞和厚重之间的差距。

来自法式餐馆、手术台和狼人的启示

让我们回顾豆豆本人的批评实践。在我看来，它具备难以错认的形式主义特征：主要考虑“外在”的、形式的(比如画的大小)要素，而忽略作品实际再现的内容——我们记得豆豆本人唯一一次提及画中的那位老太太时的措辞：停在仙人掌上的、丑恶的老蝙蝠(“a hideous old bat who looks like she's got a cactus lodged up her backside”)。我也要采用这样的策略吗？读者们(我设想，看到这篇书评的各位多数是技术专家)恐怕会讥之为买椟还珠；另外，MMM 的内容也远比“老蝙蝠”更有可谈之处。即便如此，逐一考察书中的观点也是不现实的：首先，没有人比作者已经做过的更好，如果你确实需要，不如去看原书第 18 章；其次，全书并没有一个一以贯之的中心思想可以提取，对于沟通、文

档、排错、工具等领域，每一个都值得专文仔细讨论。经过考虑，我决定谈谈那些无需我重读原书就可以回想起来的内容——经过我的巴士阅读实践幸存下来的精华成分。

人月

首先是“人月”(man-month)。熟悉软件项目管理的各位肯定清楚，人们常常根据人月来估计工作量(并相应收费)，比如一个项目 5 个人 2 个月完成，那么总工作量就是 10 人月。本书以此命名，套用唱片工业的术语，可以说“人月神话”是本“专辑”中的主打篇目，而除了以之为标题的第二章之外，并没有太多内容与此有关。

称之为“神话”，作者的用意也并非完全否定作为计量方法的人月，而是要理清这个概念中隐含的种种错觉。根据我勤勉而不可靠的记忆，这里的主要论断包括：

1. 人、月之间不能换算，换言之，2 个人做 5 个月完成，不等于说 5 个人做 2 个月就能完成；

2. (也是最有争议的一点)在项目后期增加人手，只能使工期进一步推迟；

3. 项目越大，单位工作需要的人月越多。

或者说，归根结底，作者要粉碎的是“人月”概念可以线性把握的神话：无论是在开发人员的人数上，还是在工作量本身上的变化上，都可能导致最终完成时间的非线性变化。作者的戏剧性表述是：“Adding manpower to a late software project makes it even later。”这个观点事实上并非像它看上去那么有挑衅性，而作者在初版以及后记中，都表示这包含了合理的夸张，后记里甚至援引进一步的研究，增加人手不一定会使工期进一步推迟，不过肯定会使工程效率进一步降低——而如果一定要增加人手，越早越好。

得出这样的结论，主要的考虑是增加人手带来的隐性花费(人员培训、多人工作时的通信成本等)以及项目进程中存在的无法逾越的关键路

径(crucial path)。对于项目管理人员来说，这应该已经是一个不言而喻的真理。但由于最终决策者的压力，实际项目中却往往会发生与此违背的情况。作者把一份法式餐厅菜单塞进了书中作为题图，我也建议大家学会菜单上的那个句子：Faire de la bonne cuisine demande un certain temps。Si on vous fait attendre，c'est pour mieux vous servir，et vous plaire.(做好菜需要一定的时间。如果人家让您多等会儿，那是为了让您最后吃得高兴。) 对于说法语的客户或老板，这会是一个好的搪塞。

概念完整性

如果一定要找到贯穿全书的概念的话，“概念完整性(conceptual integrity)”可能算是一个。宁可少添加一些七七八八的功能(anomalous features)，也应该保证整个系统体现的是一套完整的设计理念：这是引入概念完整性的含义。概念完整性的受益者包括最终用户、系统开发者、培训员和服务人员。概念完整性保持得好的系统更易用，需要较短的培训时间和学习时间，同时也易于开发。但是在软件生产的实践中，各级决策者都很容易产生强调“功能”，而忽视概念完整性的倾向。因此我们看到了太多包含大量“功能”，而就整体而言却不知所云的系统，也出现过太多次由于一两个附加功能的实现难度而导致整个系统开发推迟、甚至难产的事例。

所以作者提出，概念完整性是系统设计中最重要的考虑。在保证概念完整性的各方面中，我认为作者提出的两点最为有趣。第一，要克服系统的“第二版效应(the second system effect)”。一个系统设计师在设计第一版系统时，往往出于较弱的自信心以及量力而行的考虑，会尽量剪裁要实现的功能数量。而当第一版系统成功发布，开始第二版的设计时，随着自信心的增强，大量以前被压制的提议都会重现，设计师在塞入新的功能时也会不再那么保守。这很可能导致一个臃肿而缺乏概念完整性的第二版系统(作者的 1995 年举的例子是 Windows 3.1 之后的 Windows NT)。第二，整个项目团队的有效组织有助于保证概念完整性。

首先，要区分产品人员和开发人员，即 architect 和 implementer。这里所说的 architect 相当于一般而言的产品经理。只有他以及少数的几个人能确定整个产品的需求，而不应把即使是最细微的需求留给开发人员确定。在需求确定的过程中，多次反复，包括来自开发人员的反馈都是必要的，但为确保概念完整性，做出决定的必须是少数，甚至一个人。另外，在开发中可以考虑“外科手术式的”开发团队，即由唯一的“手术师”(技术大拿)以及多个助手、测试员、联络人员、文档/工具/配置管理员、秘书等组成的队伍。显然无论是产品/开发人员的区分，还是外科手术式队伍的引入，核心原则都是：职能细分，并将高要求的工作集中在少数人手中。事实上，作者在其他的章节里说过，提高软件项目的质量的最好办法之一，就是找到合适的人。“伟大”的设计师和普通设计师之间的差别，比采用任何先进工具/方法论/管理模式前后产生的差别都要大。

银弹和其他

另外，增补部分中的“银弹”是作者独创的又一个概念。在古代的狼人传说中，只有用银质子弹才能制服这些举止无常的怪兽。因此作者也用“银弹”一词来命名人们渴望找到的，用于制服软件项目这头难缠的怪兽的法宝。“没有银弹”意味着，没有任何一种方法(无论技术上的或是管理上的)，单单采取它就能将现有的软件开发生产率(可靠度/简洁度)提高一个数量级。

作者的论证是简洁而有效的(在我看来)，软件开发的困难来自两个方面，即本质的和偶然的(类似于经院哲学中的本质/偶性的对立)。本质的困难是软件开发本身所固有的，无法用任何方式取消，而偶然的困难是其中的非本质因素，可以通过引入新工具、方法论或管理模式来消除。关键在于，只要本质的困难在软件开发中消耗 10%以上的工作量，则即使全部消除偶然困难也不可能使生产率提高 10 倍。

就像作者其他的论证一样，读完关于银弹的部分我们或者会说“本来就是这样，这还用说”，或者会说“这么简单的事，我以前怎么没有

想到”。正是这种介乎漠视和震惊之间的摇摆体现了这些论断的价值：它们也许在理论上已经被过分强调、被超越、被唾弃。即便如此，在实践中我们往往连这些基本的原则都会完全不顾。对我们而言，MMM 或许首先是一本关于态度的书。它指出的问题首先是，对待软件开发工作时应采取怎样的态度。正是因为某种态度，我们才会热衷于寻找解决一切问题的银弹(RUP 也好，CMM 也好，XP/Agile 也好)，从未诚心诚意地贯彻其中任何一个原则；我们才会急于在系统中塞入一个又一个的功能，无视用户使用时的实际效果；我们才会无法相信自己能够组成“外科手术式”的团队，无法找到不做产品设计的开发人员；我们才会一次次接受形形色色的人月调整，最后把一个个项目带进了“焦油坑”。据我看来，软件工程首先是一种缓慢的艺术。在这中间，筹划、交流、冥想以及迭代占有很大比重，同样(如果不是更重要的)，错位、反复甚至离题也应占有一席之地。即便强调工程的可控性，也难以排除这些必要的环节(是否记得法式餐馆的例子)。我们所能做的，是给它们足够的时间和容忍(你知道我不是指纵容延期)。

Brooks 在书中提出的两组时间比例也许值得留意：一个“能转(ready to run)”的程序(program)要变成程序产品(programming product)需要 3 倍于编程的时间；同样，“能转”的程序产品变成程序系统(programming system)也需要 3 倍的时间。因此，要获得程序系统产品(programming systems product)，需要 9 倍于编写“能转”的程序的时间。另外，他在提到工期规划时说，整个工期的 1/3 给计划(planning)，1/6 给编码(coding)，1/4 给组件测试和早期系统测试(component test and early system test)，最后 1/4 给系统测试。

类似的划分，远比我们常见的各种方法论中的“里程碑”粗糙得多，但是，有多少人在实践中相信编码应只占整个工期的 1/6 甚至 1/9？是否仍是“态度”的原因使我们失去了缓慢工作的能力？我们能把这种态度命名为“浮躁”吗？当所有竞争者都许诺只用那 1/6 就能完成整个项目的时候，你能坚持那个“一”吗？我们能像提高气功境界那样，从一种

态度跃迁到另一种吗？软件开发是一种气功吗？外科手术、银弹和法国菜都是气功吗？狼人能用气功治好吗？气功也能用人月估计吗？豆豆和老蝙蝠，谁更像狼人？狼人也会浮躁吗？浮躁的狼人究竟更需要气功还是银弹？还是书评？一篇豆豆式的书评？

尾声

写到这里，我愿意介绍我手头这本 MMM 的由来。打开书皮，我能看到封底处“Low Price Edition (LPE) authorized for sales only in India，Bangladesh，Pakistan，Nepal，Sri Lanka and Maldives”的字样。原有的书店标签(我印象中是“Graham，250rs”)不知什么时候脱落了。但即使缺乏这个标签，仅仅“LPE”还是能让我回想起在班加罗尔 Graham 书店度过的那个匆忙的黄昏。书店的第三层或第四层是技术专区，我开始只找到了 Jacobson 们的 *OOSE*(有点困惑于复杂的书店布局)，看到另外一人手中的 *Software Project Survival Guide* 后，我求助于一个店员，他立刻拿给我那本书和一本 MMM，另一个店员看到它们，又给我递来一本 *Peopleware*。

本文的完成，主要与我一次偶然的阅读有关，与一些论坛中朋友们对 MMM 的热衷和猜想有关，而与该书中文版本的推出可能只有间接的关系。我无意以此给这个译本做任何广告(我在别处谈过我对中文翻译的态度)，当然更无意做反广告。不过，如果现在手头没有这本书的话(一个不幸的假设)，我会到书店翻翻那个译本，看看插图复制的质量(尤其是那些铜版画——LPE 能让我数清狼人脚爪上的毛)，再看看书后还有没有注释和索引。仍然是那些外在的、肤浅的因素会决定我是否购买，就像在任何一个豆豆式书评的作者那里一样。

[参考资料]

The Mythical Man-Month:Essays on Software Engineering, Anniversary Edition(2nd Edition)by Frederick P. Brooks, 1995 Addison-Wesley, 322 pp

Peopleware: Productive Projects and Teams, 2nd Ed., by Tom Demarco,

Timothy R. Lister, 1999 Dorset House, 264 pp

Object-Oriented Software Engineering: A Use Case Driven Approach by Ivar Jacobson et al., 1992 Addison-Wesley, 524 pp

Bean: The Movie, by Mel Smith (Director), 1997, 90mins