THE MYTHICAL MAN-MONTH

人月神话

FREDERICK P. BROOKS, JR.

翻译：Adams Wang

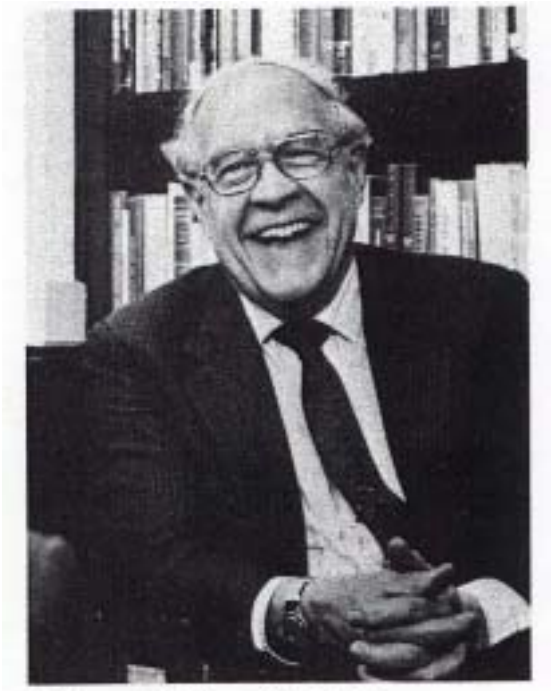


Photo credit: © Jerry Markatos

关于作者

Frederick P. Brooks，Jr.是北卡罗来纳大学 Kenan-Flagler 商学院的计算机科学教授，北卡来罗来纳大学位于美国北卡来罗来纳州的查布尔希尔。Brooks 被认为是“ IBM 360 系统之父”，他担任了 360 系统的项目经理，以及 360 操作系统项目设计阶段的经理。凭借在上述项目的杰出贡献，他、Bob Evans 和 Erich Bloch 在 1985 年荣获了美国国家技术奖（National Medal of Techology)。早期，Brooks 曾担任 IBM Stretch 和 Harvest 计算机的体系结构师。

在查布尔希尔，Brooks 博士创立了计算机科学系，并在 1964 至 1984 年期间担任主席。他曾任职于美国国家科技局和国防科学技术委员会。Brooks 目前的教学和研究方向是计算机体系结构、分子模型绘图和虚拟环境。

1975 年 版 献 辞

致两位特别丰富了我 IBM 岁月的人：

Thomas J. Watson, Jr.,

他对人们的关怀在他的公司依然无所不在

和

Bob O. Evans,

他大胆的领导使工作成为了探险。

1995 年 版 献 辞

致 Nancy,

上帝赐给我的礼物。

二十周年纪念版序言（*Preface to the 20th Anniversary Edition*）

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。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，哪些是已经改变观点的，是怎样改变的？尽管我在一些讲座上也分析过这个问题，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（除了一些细小的修正），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》。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 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*》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，它预言十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。十年只剩下一年了，我的预言看来安全了。《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作了说明，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，一直进行的软件工程研究和经验已经批评、证实和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，对我来说很有帮助。我在第 18 章列出这些观点的概要，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，新观点并不象原来的书一样来自我的亲身经历。我在大学里工作，不是在工业界，做的是小规模的项目，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，我找了一些正工作在软件工程领域的朋友，征求他们的当

前观点。他们很乐意和我共享他们的想法，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 Eard 出色地对新的章节进行了技术加工。

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 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-Wesleys 出版社按规定不允许我向它的一些扮演了关键角色的员工致谢。有两个人的贡献必须被特别提到：执行编辑 Norman Stanton 和美术指导 Herbert Boes。Boes 设计了优雅的风格，他在评注时特别提到：“ 页边的空白要宽，字体和版面要有想像力 ”。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。

查珀尔希尔，北卡罗来纳　　F.P.B., Jr.

1995 年 3 月

第一版序言（*Preface to the First Edition*）

在很多方面，管理一个大型的计算机编程项目和其它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；在很多另外的方面，它又有差别——比大多数职业经理所认为的差别还要大。

这个领域的知识在累积。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，但是还没有成型的方法来系统地进行阐述。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 年间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。在 1964 年，我成为操作系统 OS/360 的经理，发现前些年的进展使编程世界改变了很多。

管理 OS/360 的开发是很有帮助的经历，虽然是失败的。那个团队，包括我的继任经理 F. M. Trapnell，有很多值得自豪的东西。那个系统包括了很多优秀的设计和实施，成功地应用在很多领域，特别是设备无关的输入输出和外部库管理，被很多技术革新广泛复制。它现在是十分可靠的，相当有效，和非常通用的。

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做得更好。设计和实现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。大多数这些缺陷发生在 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。

就象当初接受 OS/360 的任务时协商好的，在 1965 年离开 IBM 后，我来到查珀尔希尔。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。特别地，我要说明 System/360 硬件开发和 OS/360 软件开发中的管理经验是非常不同的。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 1964-65 年的经理助理 R.P.Case，还有 1965-68 年的经理 F.M.Trapnell，进行了长谈，从中受益良多。我对比了其他大型编程项目的经理的结论，包括 M.I.T.的 F.J.Corbato，Bell 电话实验室的 V.Vyssotsky，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，送给职业程序员、职业经理、特别是程序员的职业经理。

虽然写出来的是分离的章节，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区别很大；我相信关键需要是维持产品自身的概念完整性。这些章节探讨了其中的困难和解决的方法。后续的章节探讨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。因此我尝试给出参考资料，说明某个特定知识点和指引感兴趣的读者去看其他有用的工作。很多朋友读过了本书的手稿，其中一些朋友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书中。

因为这本书是随笔不是课本，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。

深切感谢 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。

查珀尔希尔，北卡罗来纳　　F.P.B., Jr

1974 年 10 月

目录（*Contents*）

二十周年纪念版序言（*PREFACE TO THE 20TH ANNIVERSARY EDITION*） ..... **I**

第一版序言（*PREFACE TO THE FIRST EDITION*） ..... **III**

目录（*CONTENTS*） ..... **V**

焦油坑（*THE TAR PIT*） **1**

编程系统产品 1
职业的乐趣 3
职业的苦恼 4

人月神话（*THE MYTHICAL MAN-MONTH*） **6**

乐观主义 7
人月 8
系统测试 10
空泛的估算 11
重复产生的进度灾难 12

外科手术队伍（*THE SURGICAL TEAM*） **16**

问题 16
MILLS 的建议 17
如何运作 20
团队的扩建 21

贵族专制、民主政治和系统设计（*ARISTOCRACY, DEMOCRACY, AND SYSTEM DESIGN*） **22**

概念一致性 22
获得概念的完整性 23
贵族专制统治和民主政治 24
在等待时，实现人员应该做什么？ 26

画蛇添足（*THE SECOND-SYSTEM EFFECT*） **29**

结构师的交互准则和机制 29
自律——开发第二个系统所带来的后果 30

贯彻执行（*PASSING THE WORD*） **33**

文档化的规格说明——手册 33
形式化定义 34
直接整合 36
会议和大会 36
多重实现 38

电话日志.....38
产品测试.....38

为什么巴比伦塔会失败？（*WHY DID THE TOWER OF BABEL FAIL?*）.....40

巴比伦塔的管理教训.....41
大型编程项目中的交流.....41
项目工作手册.....42
大型编程项目的组织架构.....44

胸有成竹（*CALLING THE SHOT*）.....49

PORTMAN 的数据.....50
ARON 的数据.....51
HARR 的数据.....51
OS/360 的数据.....53
CORBATO 的数据.....53

削足适履（*TEN POUNDS IN A FIVE-POUND SACK*）.....55

作为成本的程序空间.....55
规模控制.....56
空间技能.....57
数据的表现形式是编程的根本.....58

提纲挈领（*THE DOCUMENTARY HYPOTHESIS*）.....60

计算机产品的文档.....60
大学科系的文档.....62
软件项目的文档.....62
为什么要有正式的文档？.....63

未雨绸缪（*PLAN TO THROW ONE AWAY*）.....64

试验性工厂和增大规模.....64
唯一不变的就是变化本身.....65
为变更计划系统.....66
为变更计划组织架构.....66
前进两步，后退一步.....68
前进一步，后退一步.....69

干将莫邪（*SHARP TOOLS*）.....71

目标机器.....72
辅助机器和数据服务.....73
高级语言和交互式编程.....76

整体部分（*THE WHOLE AND THE PARTS*）.....78

剔除 BUG 的设计.....78
构件单元调试.....80

系统集成调试 82

祸起萧墙（*HATCHING A CATASTROPHE*） 85

里程碑还是沉重的负担？ 85
“其他的部分反正会落后” 86
地毯的下面 87

另外一面（*THE OTHER FACE*） 92

需要什么样的文档 93
流程图 95
自文档化（SELF-DOCUMENTING）的程序 96

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*） 102

摘要 [1] 102
介绍 103
是否一定那么困难呢？——根本困难 103
以往解决次要困难的一些突破 106
银弹的希望 108
针对概念上根本问题的颇具前途的方法 113
NO *118*

再论《没有银弹》 （*“NO SILVER BULLET”REFIRED*） 120

人狼和其他恐怖传说 120
存在着银弹－就在这里！ 121
含糊的表达将会导致误解 121
HAREL 的分析 124
JONE 的观点——质量带来生产率 127
那么，生产率的情形如何？ 128
面向对象编程——这颗铜质子弹可以吗？ 129
重用的情况怎样？ 130
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 132
子弹的本质——形势没有发生改变 133

《人月神话》的观点：是或非？（*PROPOSITIONS OF* THE MYTHICAL MAN-MONTH: *TRUE OR FALSE*？） 134

第 1 章 焦油坑 134
第 2 章 人月神话 135
第 3 章 外科手术队伍 136
第 4 章 贵族专制、民主政治和系统设计 137
第 5 章 画蛇添足 137
第 6 章 贯彻执行 138
第 7 章 为什么巴比伦塔会失败？ 139
第 8 章 胸有成竹 141

第 9 章 削足适履 141
第 10 章 提纲挈领 143
第 11 章 未雨绸缪 143
第 12 章 干将莫邪 146
第 13 章 整体部分 148
第 14 章 祸起萧墙 149
第 15 章 另外一面 150
原著结束语 152

20 年后的人月神话（THE MYTHICAL MAN-MONTH *AFTER* 20 YEARS） 153

为什么会出现二十周年纪念版本？ 153
核心观点：概念完整性和结构师 154
开发第二个系统所引起的后果：盲目的功能和频率猜测 156
图形（WIMP）界面的成功 157
没有构建舍弃原型——瀑布模型是错误的！ 160
增量开发模型更佳——渐进地精化 162
关于信息隐藏，PARNAS 是正确的，我是错误的 165
人月到底有多少神话色彩？BOEHM 的模型和数据 167
人就是一切（或者说，几乎是一切） 168
放弃权力的力量 169
最令人惊讶的新事物是什么？数百万的计算机 171
全新的软件产业——塑料薄膜包装的成品软件 173
买来开发——使用塑料包装的成品软件包作为构件 174
软件工程的状态和未来 176

结束语 :令人向往、激动人心和充满乐趣的五十年（*EPILOGUE FIFTY YEARS OF WONDER, EXCITEMENT, AND JOY*） 178

注解和参考文献（*NOTES AND REFERENCES*） 180

第 1 章 180
第 2 章 180
第 3 章 180
第 4 章 181
第 5 章 181
第 6 章 182
第 7 章 182
第 8 章 182
第 9 章 183
第 10 章 183
第 11 章 184
第 12 章 184
第 13 章 185
第 14 章 186
第 15 章 187
第 16 章 187

第 17 章.....188
第 18 章.....190
第 19 章.....190

索引（*INDEX*）.....193

焦油坑（*The Tar Pit*）

岸上的船儿，如同海上的灯塔，无法移动。

- *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。它们挣扎得越是猛烈，焦油纠缠得越紧，没有任何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，其中只有非常少数的项目满足了目标、时间进度和预算的要求。各种团队，大型的和小型的，庞杂的和精干的，一个接一个淹没在了焦油坑中。表面上看起来好像没有任何一个单独的问题会导致困难，每个都能被解决，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，就必须试图先去理解它。

因此，首先让我们来认识一下软件开发这个职业，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改造的简陋车库中，编出了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 1000 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*。

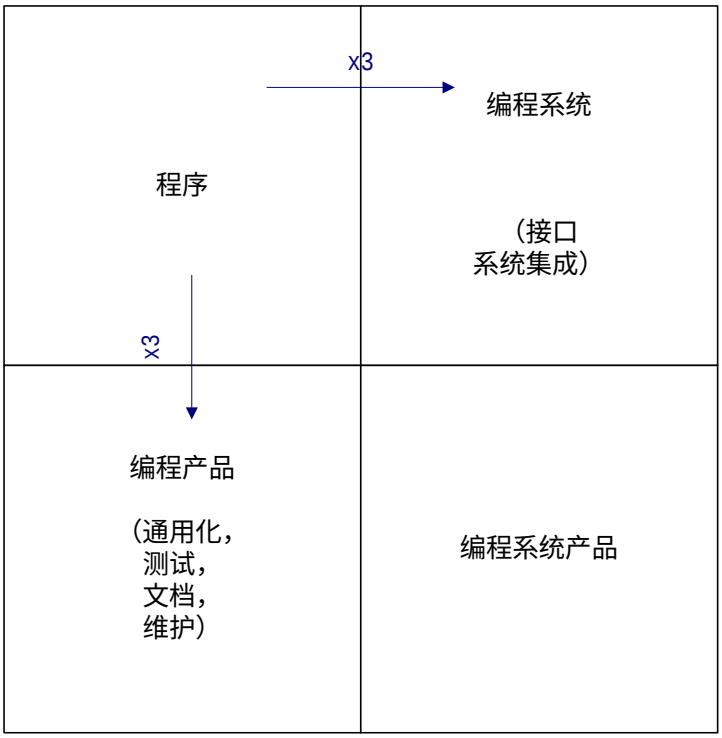


图 1.1：编程系统产品的演进

在图 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。

有两种途径可以使程序转变成更有用的，但是成本更高的东西，它们表现为图中的边界。

水平边界以下，程序变成*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。它可以运行在多种操作系统平台上，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扩展，以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，至少是已经过测试的程序的三倍。

回到图中，垂直边界的右边，程序变成*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的程序集合，具有规范的格式，可以进行交互，并可以用来组装和搭建整个系统。要成为系统构件，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，程序必须同其它系统构件单元一道，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，所以测试的范围非常广。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三倍。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）。和以上的所有的情况都不同的是，它的成本高达九倍。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？

首先是一种创建事物的纯粹快乐。如同小孩在玩泥巴时感到愉快一样，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、崭新的树叶和雪花上的喜悦[1]。

其次，快乐来自于开发对其他人有用的东西。内心深处，我们期望其他人使用我们的劳动成果，并能对他们有所帮助。从这个方面，这同小孩用粘土为“爸爸办公室”捏制铅笔盒没有本质的区别。

第三是整个过程体现出魔术般的力量——将相互啮合的零部件组装在一起，看到它们精妙地运行，得到预先所希望的结果。比起弹珠游戏或点唱机所具有的迷人魅力，程序化的计算机毫不逊色。

第四是学习的乐趣，来自于这项工作的非重复特性。人们所面临的问题，在某个或其它方面总有些不同。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，乐趣还来自于工作在如此易于驾驭的介质上。程序员，就像诗人一样，几乎仅

仅工作在单纯的思考中。程序员凭空地运用自己的想象，来建造自己的“城堡”。很少有这样的介质——创造的方式如此得灵活，如此得易于精炼和重建，如此得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；可以移动和运行，能独立产生可见的输出；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了现实。在键盘上键入正确的咒语，屏幕会活动、变幻，显示出前所未有的或是已经存在的事物。

编程非常有趣，在于它不仅满足了我们内心深处进行创造的渴望，而且还愉悦了每个人内在的情感。

职业的苦恼

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

首先，必须追求完美。因为计算机也是以这样的方式来变戏法：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，我认为学习编程的最困难部分，是将做事的方式往追求完美的方向调整。

其次，是由他人来设定目标，供给资源，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（相对于正式）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，而往往这些程序设计得并不合理，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完整的。

下一个烦恼——概念性设计是有趣的，但寻找琐碎的 bug 却只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，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它已有的系统，而不是未实现的概念。因此，我们所面临的挑战和任务是在现有的时间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。对于许多人而言，其中的乐趣远大于苦恼。而本书的剩余部分将试图搭建一些桥梁，为通过这样的焦油坑提供一些指导。

人月神话（*The Mythical Man-Month*）

美酒的酿造需要年头，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 Antoine 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，它比其他所有因素加起来的影响还大。导致这种普遍性灾难的原因是什么呢？

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进行估算这项工作。

第四，对进度缺少跟踪和监督。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无谓的举动。

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，结果是勿庸置疑的：“这次它肯定会运行。”或者“我刚刚找出了最后一个错误。”

所以系统编程的进度安排背后的第一个假设是：*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“*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、或者程序的出现，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，当某人阅读书本、使用计算机和运行程序的时候，他与作者的思想相互沟通，从而创作过程得以结束。

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、电器安装等。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，实际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，因此总会有 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”的假设在时间进度上具有可实现性。因为所遇

的延迟是一个概率分布曲线，“ 不会延迟 ”仅具有有限的概率，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，甚至接近于无。

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。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（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。

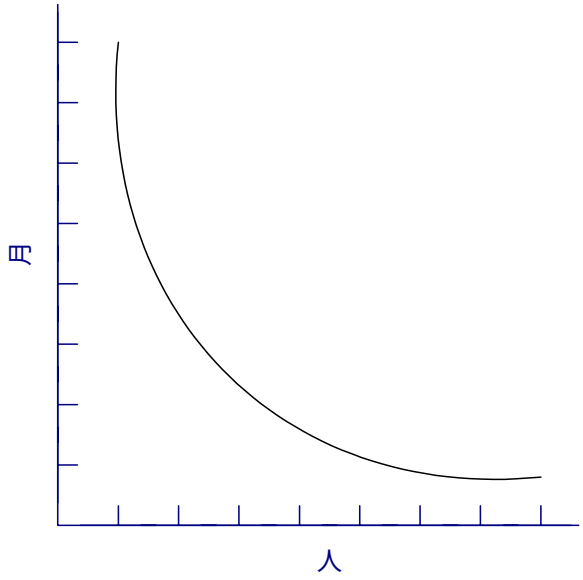


图 2.1：人员和时间之间的关系——完全可以分解的任务

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（图 2.2）。无论多少个母亲，孕育一个生命都需要十个月。由于调试、测试的次序特性，许多软件都具有这种特征，

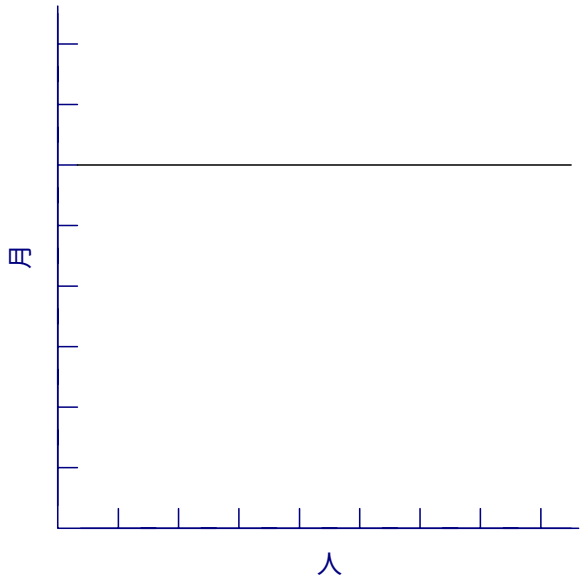


图 2.2：人员和时间之间的关系——无法分解的任务

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，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，也比未调整前要差一些（图 2.3)。

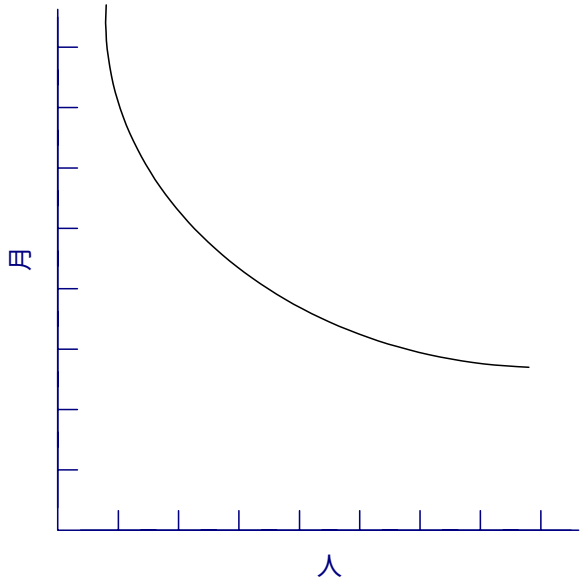


图 2.3：人员和时间之间的关系——需要沟通的可分解任务

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、项目目标以及总体策略上的培训。这种培训不能分解，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和其他部分单独协作，则工作量按照 n(n-1)/2 递增。一对一交流的情况下，三个人的工作量是两个人的三倍，四个人则是两个人的六倍。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，此时我们会被带到图 2.4 的境地。

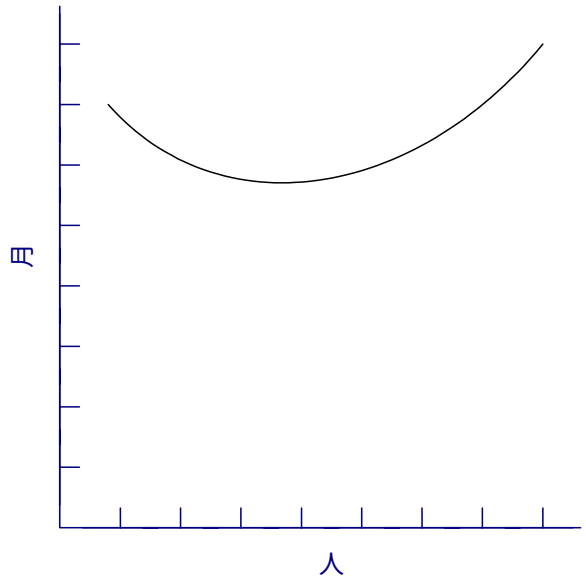


图 2.4：人员和时间之间的关系——关系错综复杂的任务

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

在时间进度中，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，要求的时间依赖于所遇到的错误、缺陷数量以及捕捉它们的程度。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寻常的多。即便如此，仍不足以产生详细和稳定的计划规格说明，也不足以容纳对全新技术的研究和摸索。

2. 对所完成代码的调试和测试，投入近一半的时间，比平常的安排多很多。

3. 容易估计的部分，即编码，仅仅分配了六分之一的时间。

通过对传统项目进度安排的研究，我发现很少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。更重要的是，当软件用来支持其他的商业活动（计算机硬件到货，新设备、服务上线等等）时，这些活动延误出现即将发布前，那么将付出相当高的商业代价。

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持不如厨师，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的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健壮可靠和规避风险的估计。

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、估算规则等等，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 至 2.4 所示，这可能有所帮助，也可能无法解决问题。

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 4 个月时间，在每个月的末尾安排了可测量的里程碑 A、B、C、D（图 2.5）。

现在假定两个月之后，第一个里程碑没有达到（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，即如图 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。

2. 假设任务必须按时完成。假设整个任务的估计偏低，即如图 2.7 所示，那么还有 18 个人月的工作量以及 2 个月的时间，需要将原来的 3 个人增至 9 个人。

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，一旦开发团队观察到进度的偏差，总是倾向于对任务进

行削减。当项目延期所导致的后续成本非常高时，这常常是唯一可行的方法。项目经理的相应措施是仔细、正式地调整项目，重新安排进度；或者是默默地注视着任务项由于轻率的设计和不完整的测试而被剪除。

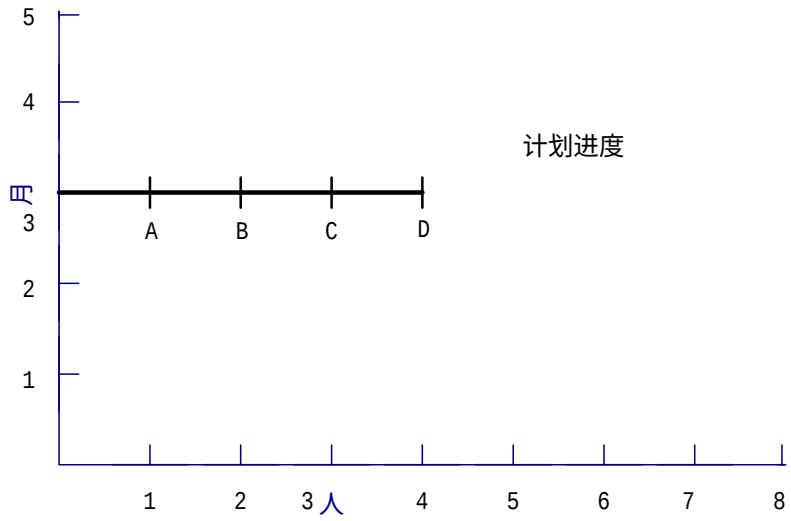


图 2.5

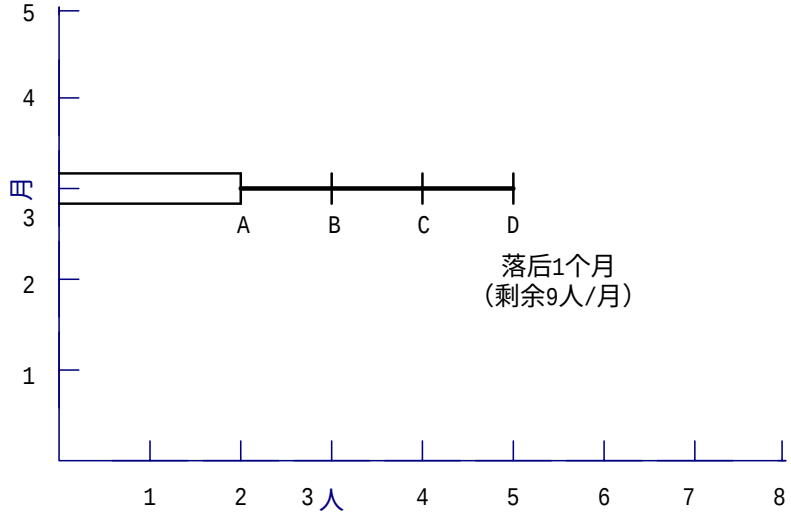


图 2.6

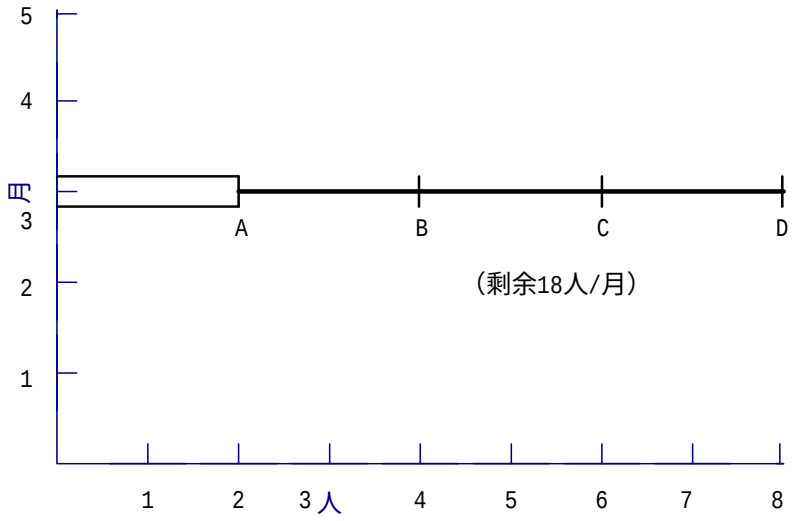


图 2.7

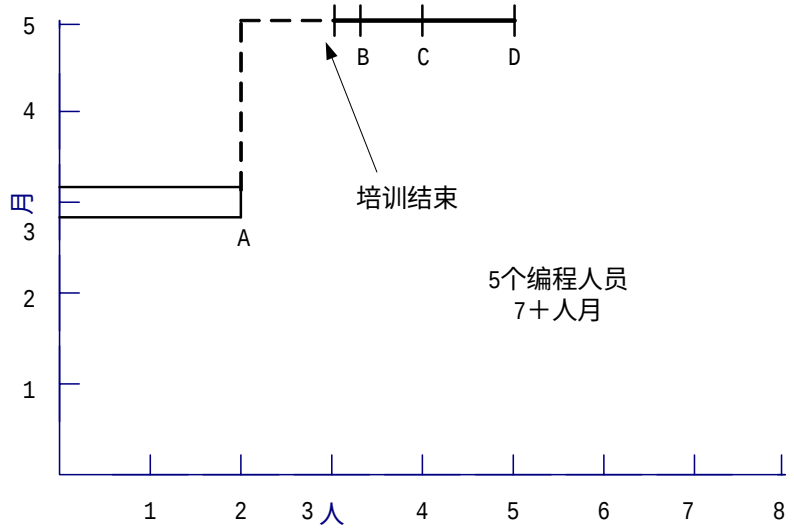


图 2.8

前两种情况中，坚持把不经调整的任务在四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（图 2.8）——不论在多短的时间内，聘请到多么能干的两位新员工，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*三个人月将会投入到原有进度安排以外的工作中。*另外，原先划分为三个部分的工作，会重新分解成五个部分；某些已经完成的工作必定会丢失，系统测试必须被延长。因此，在第三个月的月末，仍然残留着 7 个人月的工作，但此时只有 5 个有效的人月。如同图 2.8 所示，产品还是会延期，如同没有增加任何人手（图 2.6）。

期望四个月内完成项目，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，在第二个月末需要增添 4 个，而不是 2 个人员。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。

注意在第三个月的结尾时，情况看上去还是很糟。除去管理的工作不谈，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，结果往往会是上述循环的重复。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。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，重现“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。

简单、武断地重复一下 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，人员的数量依赖于单个子任务的数量。从这两个数值可以推算出进度时间表，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表。总之，在众多软件项目中，缺乏合理的时间进度是造成项目滞后的最主要原因，它比其他所有因素加起来的影响还要大。

外科手术队伍（*The Surgical Team*）

这些研究表明，效率高和效率低的实施者之间具体差别非常大，经常达到了数量级的水平。

- *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头等人才组成的小型、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——如何在有意义的时间进度内创建*大型*的系统？那么就让我们现在来仔细讨论一下这个问题的每一个方面。

问题

软件经理很早就认识到优秀程序员和较差的程序员之间生产率的差异，但实际测量出的差异还是令我们所有的人吃惊。在他们的一个研究中，Sackman、Erikson 和 Grand 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；在运行速度和空间上具有 5:1 的惊人差异！简言之，$20,000/年的程序员的生产率可能是$10,000/年程序员的 10 倍。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的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、不充分的，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、SAFE 等等——这个列表可以不断地继续下去。

得出的结论很简单：如果一个 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，原有的开发队伍不是理想的*小型*强有力的团队，因为通常的共识是不超过 10 个人，而该团队规模如此之大，以至于至少需要两层的管理，或者说大约 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 200 人的队伍仍然不足以开发真正的大型系统。例如，考虑 OS/360 项目。在顶峰时，有超过 1000 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、支持小组等等。从 1963 年到 1966 年，设计、编码和文档工作花费了大约 5000 人年。如果人月可以等量置换的话，我们所假设的 200 人队伍需要 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 10 人队伍。作为一个尺度，假设他们都非常厉害，比一般的编程人员在编程和文档方面的生产率高 7 倍。假定 OS/360 原有开发人员是一些平庸的编程人员（这与实际的情况*相差很远*）。同样，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较少的沟通和交流。那么，5000/（10×7×7）= 10，他们需要 10 年来完成 5000 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，由一个人来进行问题的分解，其他人给予他所需要的支持，以提高效率和生产力。

简单考虑一下，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；该计算机系统不仅仅能进行测试，还存储程序的各种版本，以允许简单的文件更新，并对他的文档提供文本编辑能力。首席程序员需要极高的天分、十年的经验和应用数学、业务数据处理或其他方面的大量系统和应用知识。

副手。他是外科医生的后备，能完成任何一部分工作，但是相对具有较少的经验。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的功能和接口讨论中代表自己的小组。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，但针对代码的任何部分，不承担具体的开发职责。

管理员。外科医生是老板，他必须在人员、加薪等方面具有决定权，但他决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、工作地点安排和机器的专业管理人员，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。外科医生负责产生文档——出于最大清晰度的考虑，他必须书写文档。对内部描述和外部描述都是如此。而编辑根据外科医生的草稿或者口述的手稿，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以及监督文档生成的机制。

两个秘书。管理员和编辑每个人需要一个秘书。管理员的秘书负责项目的协作一致和非产品文件。

程序职员。他负责维护编程产品库中所有团队的技术记录。该职员接受秘书性质的培训，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*计算机的运作和产物，并将所有的程序和数据看作是团队的所有物，而非私人财产。

程序职员的专业化分工，使程序员从书记的杂事中解放出来，同时还可以对那些杂事进行系统整理，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令人满意，而且需要具备较高的可靠性。外科医生则是这些工具、服务可用性的唯一评判人员。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人员。因为他的工作是检查*他*的外科医生所需要的工具。工具维护人员常常要开发一些实用程序、编制具有目录的过程库以及宏库。

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头，同时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。这些专家非常有帮助，很快大家会向他咨询。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（两到三天）。通常一个语言专家可以为两个到三个外科医生服务。

以上就是如何参照外科手术队伍，以及如何对 10 人的编程队伍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。十个人，其中七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——副手队伍架构之间的区别。首先，传统的团队将工作进行划分，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作缓冲区，然而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。

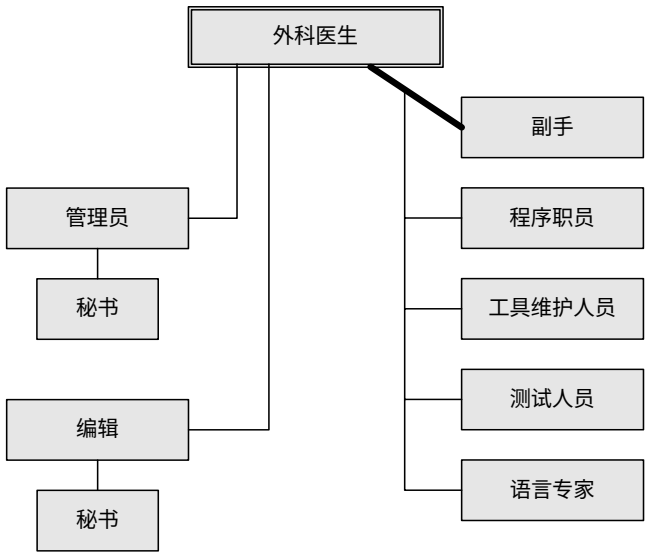


图 3.1：10 人程序开发队伍的沟通模式

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。

团队的扩建

就目前情况而言，还不错。然而，现在所面临的问题是如何完成 5000 人年的项目，而不是 20 或 30 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，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七分之一或更少。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，这在后续的章节中会继续进行讨论。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。

贵族专制、民主政治和系统设计（*Aristocracy, Democracy, and System Design*）

大教堂是艺术史上无与伦比的成就。它的原则既不乏味也不混乱……真正达到了风格上的极致，完成这件作品的艺术家们，完全领会和吸收了以往的成功经验，也完全掌握了他们那个时代的技术，而且在应用的时候做到了恰如其分，绝不轻率，也绝不花哨。

正是 Jean d‘ Orbais 构思了建筑的整体设计，这个设计得到了后继者的认同，至少在本质上如此。这也是这个建筑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]

概念一致性

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和个人品味上的改变。所以，在雄伟的哥特式的教堂上，依附着祥和的诺曼第风格十字架，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和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，同时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。

- 如何得到概念的完整性？
- 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，和一群创造性天分和构思被压制的平民编程实现人员？
- 如何避免结构设计师产出无法实现、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十倍。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，比用在学习、记忆和搜索手册上的时间要少时，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，随着越来越多的功能添加，收益和成本的比率正逐渐地减少。而 IBM 650 使用的容易程度总萦绕

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，功能与理解上复杂程度的比值才是系统设计的最终测试标准。单是功能本身或者易于使用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。但是，一旦以易用性作为衡量标准，单独的功能和易于使用都是不均衡的，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（simplicity）*是不够的，Mooers 的 TRAC 语言和 Algol 68 用很多独特的基本概念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，还需要学习晦涩的用法，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、原则和一致的折衷机制。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。第一种是仔细地区分设计方法和具体实现。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 360 计算机产品线上的巨大成功。但同时我也看到了这种方法在 360 操作系统的开发中，由于缺乏广泛应用所遭受的失败。

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[2]。

因此，系统的结构师，如同建筑的结构师一样，是用户的代理人。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所鼓吹的利益。

体系结构同实现必须仔细地区分开来。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。它的结构包括表面、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、以编译器为核心、快速编译和优化以及侧重语法的实现。相类似的，任何汇编语言和任务控制语言都允许有多种编译器或调度程序的实现。

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被那些精英单独占有，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者或者用户。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 ”。就必须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。如果要得到系统概念上的完整性，那么必须控制这些概念。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是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。在给定体系结构下的设计实现，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，产品的成本性能比很大程度上依靠实现人员，就如同易用性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，但这似乎并没有被压制他的创造性。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。

类似的，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 语言的编译器。他说：“ 最后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，但同时也是让人记忆深刻的教训。当年我们计划和组织编写 OS/360 外部技术说明的那个夜晚，常常重现在我的脑海。我和体系结构经理、程序实现经理一起制订计划进度，并确认责任分工。

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

程序实现经理有 150 人。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，如果光是由体系结构的团队

承担该工作，他的 150 人只能坐在那儿干等 10 个月，无所事事。

对此，体系结构的经理的反应是，如果让程序实现队伍来负责该工作，结果*不会*按时完成，仍将推迟 3 个月，而且质量更加低劣。我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 名编程人员进行工作的愿望。而它也正是我想强调的致命危险。

当建议由体系结构的团队来编写计算机和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

这些问题中的第一个确实是一项危险，在下一章中我们将讨论这个问题，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别的创造性活动。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？

如同 Blaauw 所指出的，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）、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定义良好的成本和目标时，工作就可以开始了。他可以开始设计数据流、控制序列、大体的系统划分等等。同时，还需要选用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，在物理实现的级别，电路、板卡、线缆、机箱、电源和内存必须分别设计、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。。

在编程系统的开发中，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的事情可以做。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、算法以及所有的工具。另外，还需要花费一些时间和体系结构师沟通。

同时，在物理实现的级别，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。同工作的水平分割相比，垂直划分从根本上大大减少了劳动量，结果是使交流彻底地简化，概念完整性得到大幅提高。

画蛇添足（*The Second-System Effect*）

聚沙成塔，集腋成裘。

- *奥维德*

Adde parvum parvo magnus acervus erit.

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

- *OVID*

如果将制订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、仔细和谐的交流。另外，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预算或修订设计，调整到下一期工程。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。

类似的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：削减设计或者建议成本更低的实现方法——挑战估算的结果。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平静；
- 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——当正在实现产品时，某些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，所以他会谨慎仔细地工作。

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，熟练掌握了相应的知识，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推迟了。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，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。709 的操作集合被设计得如此丰富和充沛，以至于只有一半操作被常规使用。

让我们来看看更严重的例子——Stretch 计算机的结构（architecture）、设计实现（implementation）、甚至物理实现（realization），它是很多人被压抑创造力的宣泄出口。如果 Strachey 在评审时所述：

对于 Stretch 系统，我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[1]*。*

操作系统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*两个*以上早期操作系统的经验[2]。因此,OS/360是典型的第二次开发(second-system effect)的例子，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，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（second-system effect）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。除了这些基本的功能，它还支持程序的覆盖（overlay)。这是所有实现的覆盖服务程序中最好的一种。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。它还允许在运行时刻改变覆盖，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，它同时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多么好啊！

另外，链接编辑器需要如此大的空间，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还要慢。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]中魔术般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。使用语言解释器和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，那么系统将是多么的优秀啊！

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、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。

结构师如何避免画蛇添足——开发第二个系统所引起的后果（second-system effect）？是的，他无法跳过二次系统。但他可以有意识关注那些系统的特殊危险，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。

项目经理如何避免画蛇添足（second-system effect）？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。

贯彻执行（*Passing the Word*）

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，关于总统的权力*[1]

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

- *HARRY S. TRUMAN, ON PRESIDENTIAL POWER*[1]

假设一个项目经理已经拥有行事规范的结构师和许多编程实现人员，那么他如何确保每个人听从、理解并实现结构师的决策？对于一个由 1000 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，尽管光有文档是不够的。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；同样的，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。

手册不但要描述包括所有界面在内的用户可见的一切，它同时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，而实现人员的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，所以每条说明都必须重复所有的基本要素，所以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* 的一致完整性来自仅有两名作者的事实：Gerry Blaauw 和 Andris Padegs。思路是大约十个人的想法，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，必须对很多原先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，*决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 *System360 Principles of Operation* 的附录。它精确仔细地规定了 System/360 兼容性的*限制*。它定义了兼容性，描述了将达到的目标，列举了很多外部显示的各个部分：源于某个模型与其他模型差异，带来变化的部分和保持不变的部分；或者是某个给定模型的拷贝不同于其他拷贝的地方；甚至是工程上的变更引起拷贝自身上的差异。而这正是一个规格说明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

英语或者其他任何的人类语言，从根本上说，都不是一种能精确表达上述定义的手段。因此，手册的作者必须注意自己的思路和语言，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。如文中所示，形式化定义是精确的，它们倾向于更加完整；差异得更加明显，可以更快地完成。但是形式化定义的缺点是不易理解。记叙性文字则可以显示结构性的原则，描述阶段上或层次上的结构，以及提供例子。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*。在表达的精确和简明性上，目前所提出的形式化定义，具有了令人惊异的效果，增强了我们进行准确表达的信心。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：“决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主要方式，形式化定义用作辅助表述。System/360 也将记叙性文字用作标准，以及形式化定义用作派生的论述。

很多工具可以用于形式化定义，例如巴科斯范式在语言定义中很常用，它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 建议了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[8]、System/360。

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表达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。

如同前面所示，形式化定义可以是一种设计实现。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。如果手册有一些模糊的地方？“问一问机器！”——设计一段程序来得到其行为，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，是一系列可怕的缺点。实现可能更加过度地规定了外部功能。例如，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“古玩”——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，

同时还声明了自己到底做了些什么。

因此，当尖锐的问题被提及时，实现有时会给出未在计划中的意外答案；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，往往是低效或者代价高昂的。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，成为了真正定义的一部分。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分发和强制定义。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和共享存储器的声明，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，可以通过增加或插入新变量，或者重新编译受影响的程序。这种方法不需要修改程序内容。

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，加上硬件和软件实现人员代表和市场计划人员参与，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。新问题通常会被讨论一些时间。重点是创新，而不仅仅是结论。该小组试图发现解决问题的新方法，然后少数解决方案会被传递给一个和几个结构师，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，允许工作继续进行。如果任何人对结果*过于*不高兴，可以立刻诉诸于项目经理，但是这种情况非常少见。

这种会议的卓有成效是由于：

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了注意力，强制了决策的制订，避免了会议草稿纪要方式的不一致。

5. 清晰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，有时周例会没有重新考虑，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。出席人员不仅仅包括体系结构小组和编程人员、实现人员的结构代表，同时包括编程经理、市场和实现人员，由 System/360 的项目经理主持。议程典型地包括大约 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，会制订出决策，加上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更新了的手册说明，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。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。显然有很多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，并发布给用户和实现人员。这种机制很不正式，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敌人——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。

为什么巴比伦塔会失败？（*Why Did the Tower of Babel Fail?*）

……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，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，这个塔将高达云宵，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，使用一种的语言，如果他们一开始就能建造城市和高塔，那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. (Book of 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，但巴比伦塔同时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，来看看有什么值得学习的教训。这个项目到底有多好的先决条件？他们是否有：

1. *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. *人力？*非常充足。

3. *材料？*在美索不达米亚有着丰富的泥土和柏油沥青。

4. *足够的时间？*没有任何时间限制的迹象。

5. *足够的技术？*是的，金字塔、锥形的结构本身就是稳定的，可以很好分散压力负载。对砖石建筑技术，人们有过深刻的研究。同样，项目远在达到技术限制之间，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？他们还缺乏些什么？两个方面——*交流，*以及交流的结果——*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。通过史书的字里行间，我们推测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

现在，其实也是这样的情况。因为左手不知道右手在做什么，所以进度灾难、功能的不合理和系统缺陷纷纷出现。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）功能的实现者遇到了问题，并且统计报告显示了应用程序很少使用该功能。基于这些考虑，他降低了覆盖功能的速度。与此同时，整个开发队伍中，其他同事正在设计监控程序。监控程序在很大程度上依赖于覆盖功能，它在速度上的变化成为了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？通过所有可能的途径。

- *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，能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。项目工作手册不是独立的一篇文档，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。如果某人就硬件和软件的某部分，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，文章的剪裁粘贴与钢笔一样有用。

基于上述理由，再加上“未来产品”的质量手册将诞生于“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。同许多其它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。1000 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*上的要求都紧迫了许多。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 强烈要求制订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。

我们很快决定了*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，实际上这很难做到。不过，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编辑系统，它对实时维护有不可思议的帮助。编辑、排版、打印的工作直接在计算机和打印机上完成，周转时间少于一天。但即便如此，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？ ”。

理解的问题可以通过持续的文档维护来解决。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，稳定运行了六个月。工作手册大约厚达 1.5 米！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 5 厘米，归入档案的页数大概有 150 页。日常工作手册的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

缩胶片大量地减轻了归档问题。

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，微缩胶片不容易被读者强调、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，文档更加容易使用。

总之，我觉得微缩胶片是非常好的一种方法。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”的方式保存，在一个固定的地方提供访问。编程人员可能会每天阅读，但如果错过了一天，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，根据某种经过细致考虑的规则组织在一起。唯一发生改变的地方是分发机制和查询方法。斯坦福研究机构的 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。

卡内基－梅隆大学的 D.L.Parnas 提出了更彻底的解决方法[1]。他认为，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和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 2^n 个必须合作的潜在团队。团队组织的目的是减少不必要交流和合作的数量，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，树状管理结构所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出现。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. 技术主管和结构师（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。我们先分析一下两个角色，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，划分工作及制订进度表。他要求，并一直要求必要的资源。这意味着他主要的工作是与团队外部，向上和水平地沟通。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。这种方式非常容易应用在很小型的队伍中，可能是三个或六个开发人员。在大型的项目中则不容易得到应用。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，思考者－实干家太少了。

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观点；他们必须在主要的技术问题出现之前，私下讨论它们；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来（如，办公室的大小、地毯、装修、复印机等等）体现技术主管的威信，尽管决策权力的源泉来自管理。

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，总有些该死的笨蛋要我做一些关于卡车、或者电话、以及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 非常温和的说：“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，为你搭建一个免于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，而不是卡车的合同。”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。

过了四个小时，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。他是你的新奴隶，你仍是总工程师，毫无疑问的老板。Jock 是其他一切的主管。从现在起，你不需要担心其他的任何问题，除了建造登月飞船的一些细节问题。”

他们握了一下手。“Coster 先生，我只想问一件事，”Berkeley 认真的说，“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。只需按个按钮知会一声，它们就会被完成！”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？”

“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。

胸有成竹（*Calling the Shot*）

实践是最好的老师。

- *PUBILIUS*

实践是最好的老师，但是，如果不能从中学习，再多的实践也没有用。

- *《可怜的理查年鉴》*

Practice is the best of all instructors.

- *PUBILIUS*

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的时间？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六分之一左右，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 3200 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和调试时间为 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四分之一，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 码短跑记录外推，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

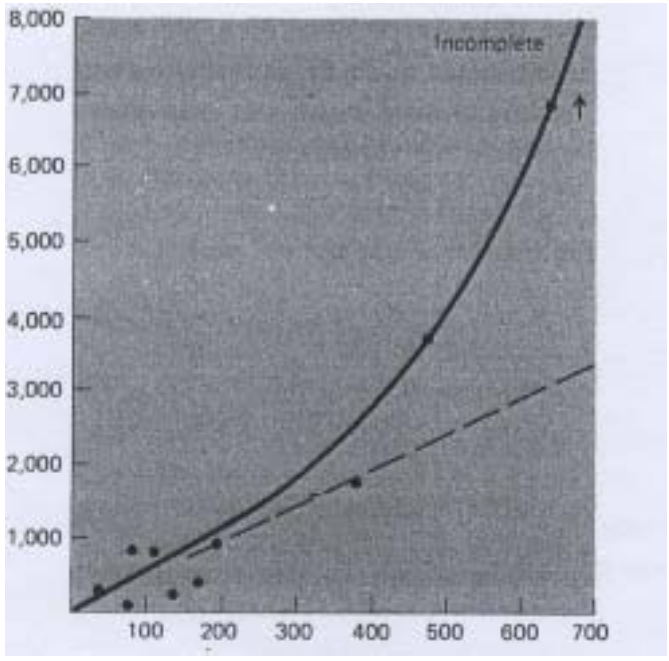
情。即使在不考虑相互交流沟通，开发人员仅仅回顾自己以前工作的情况下，这些数字仍然显示出工作量是规模的幂函数。

图 8.1 讲述了这个悲惨的故事。它阐述了 Nanus 和 Farr[2] 在 System Development Corporation 公司所做研究，结果表明该指数为 1.5，即，

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

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，提出了很多估计的技术。 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。



注：

- incomplete－未终结的

图 8.1：编程工作量是程序规模的函数

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大约 1/2，每项工作花费的时间大约是估计的两倍。这些

估计通常是非常仔细的，由很多富有经验的团队完成。他们对 PERT 图上数百个子任务估算过（用人小时作单位）。当偏移出现时，他要求他们仔细地保存所使用时间的日志。日志显示事实上他的团队仅用了百分之五十的工作周，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，在他所工作过的 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）[7]的基础上，对程序员的生产率进行了研究。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。当这些数据采用除以 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，Bell 电话实验室电子交换系统领域的编程经理，在 1969 年春季联合计算机会议[8]的论文中，汇报了他和其他人的经验。这些数据如图 8.2、8.3 和 8.4 所示。

这些图中，图 8.2 是最数据详细和最有用的。头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。没有包括计划、硬件机器支持、文书工作等类似活动的工作量。

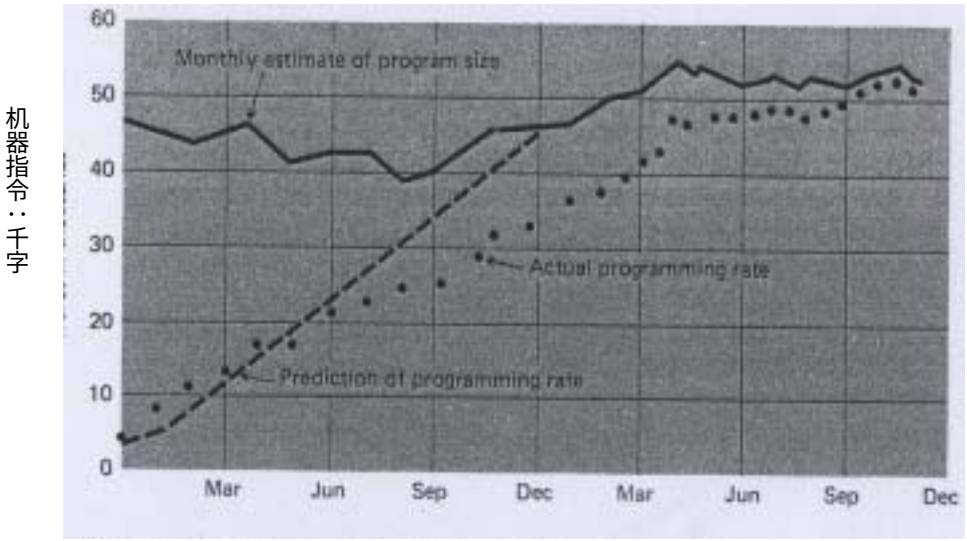
生产率同样地被划分为两个类别，控制程序的生产率大约是 600 指令每人年，语言翻译大约是 2200 指令每人年。注意所有的四个程序都具有类似的规模——差异在于工作组的大小、时间的长短和模块的个数。那么，哪一个是原因，哪一个是结果呢？是否因为控制程

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？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 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。

	程序单元	程序员人数	年	人年	程序字数	字/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	$2^1/_4$	17	38,000	2230
语言解释器（汇编）	15	13	$2^1/_2$	11	25,000	2270

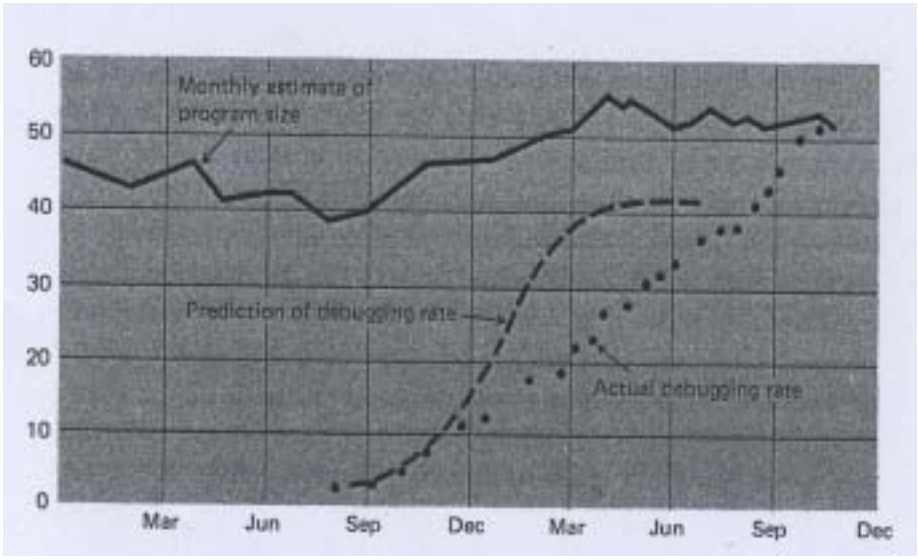
图 8.2：4 个 NO.1 的 ESS 编程工作总结



注：

- Monthly estimate of program size－程序规模月估计
- Actual Programmize rate－实际编程速度
- Prediction of programming rate－预计编程速度

图 8.3：ESS 预计和实际的编程速度



注：

- Monthly estimate of program size－程序规模月估计
- Actual Programmize rate－实际调试速度
- Prediction of programming rate－预计调试速度

图 8.4：ESS 预计和实际的调试速度

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 2000～3000（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。在复杂程度估计这片“ 沼泽 ”上的指导原则是：编译器的复杂度是批处理程序的三倍，操作系统复杂度是编译器的三倍[8]。

Corbato 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数据公布得很少。Corbato 的 MIT 项目 MAC 报告表示在 MULTICS 系统上，平均生产率是 1200 行经调试的 PL/I 语句（大约在 1 和 2 百万指令之间）/人年。

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良好平均值。

但 Corbato 的数字是*行/*人年，不是*指令！*系统中的每个语句对应于手写代码的 3 至 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.
- 使用适当的高级语言，编程的生产率可以提高 5 倍。

削足适履（*Ten Pounds in a Five-Pound Sack*）

他应该瞪大眼睛盯着诺亚，……好好学习，看他们是怎样把那么多东西装到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.

- *SYDENY 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 美金，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 美金/每月每千字节。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 1920 美金的内存租用费。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，仅仅因为它耗资两千七百万美元一样无知。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是什么？投资在内存上的每月 4800 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。相反的，其他的做法是非常不负责任的。所以，应

该从整体上来进行评价。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。他必须研究用户和他们的应用，以设置将开发系统的规模。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，我们依然能从中得到一些痛苦的教训。

首先，仅对核心程序设定规模目标是不够的，必须把所有的方面都编入预算。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目标。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，他会把它分解成链接库。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。每个人都汇报了核心的大小，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 性能仿真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 Modal 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：和制订驻留空间预算一样，应该制订总体规模的预算；和制订规模预算一样，应该制订后台存储访问的预算。

下一个教训十分类似。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了用户空间的一部分。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。

在内存大小一定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小的功能模块，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（overlaid），系统的主干占用的空间，会被用作其他部分的交换页面。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 倍临时性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引起空间和速度上的损失。这样后果其实是——很难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－时间的折衷。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－时间折衷。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掌握的知识和先前的经验。特别是使用新语言或者新机器时，培训显得尤其重要。熟练使用往往需要快速的学习和经验的广泛共享，也许它应该伴随特别的新技术奖励或者表扬。

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

创造出自精湛的技艺，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。这种战略上突破有时是一种新的算法，如快速傅立叶变换，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。

很容易就能找到重新表达所带来好处的例子。我记得有一个年轻人承担了为 IBM650 开发精细的控制台解释器的任务。他发现用户交互得很慢，并且空间很昂贵。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，反而得到了十倍的补偿。（Brooks 和 Iverson 第六章结尾的练习以及 Knuth 的练习[2]--*自动数据处理*[1]包含了许多类似的例子。）

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。

提纲挈领（*The Documentary Hypothesis*）

前提：

在一片文件的汪洋中，少数文档形成了关键的枢纽，每件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。

The hypothesis:

Amid a wash of paper, a small number of documents become the critical pivots around which every project's management revolves. 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，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。

慢慢的，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。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：计算机手册和性能规格说明。它是在计划新产品时第一个产生，并且最后

完成的文档。

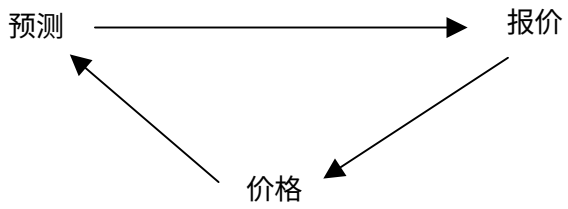
进度、时间表

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。预算的存在会迫使技术决策的制订，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。

组织机构图

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，首先需要制订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。固定成本又决定了价格。

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，以产出更低的报价。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六个月变化一次。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，成本降低是主要的焦点，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，学校的决策机构完成了这项任务。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、人物、做什么、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。不论项目的规模如何小，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。接着，他会和其他管理人员一样要求各种文档。

做什么：目标。定义了待完成的目标、迫切需要的资源、约束和优先级。

做什么：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：进度表

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，一开始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。项目经理常常会不断发现，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“完备信息管理系统”——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、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，并根据计划实现。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、人物、做什么、资金。这些少量的关键文档封装了一些项目经理的工作。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。

未雨绸缪（*Plan to Throw One Away*）

不变只是愿望，变化才是永恒。

- *SWIFT*

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。不管怎么样，重要的是先去尝试。

- *富兰克林 D. 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*[1]

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“*实验性工厂（pilot planet）*”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/每天的试验场所测试，然后再用于 2,000,000 加仑/每天的净化系统。

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，接着根据时间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，所以开发的系统必须被抛弃，但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不可避免，从而直面整个变化现象是非常有用的。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，而不仅仅是实际的产品。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，例如新型汽车或者计算机。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。

> 我从不建议顾客目标和需求的所有变更必须、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]。

为变更计划系统

如何为上述变化设计系统，是个非常著名的问题，在书本上被普遍讨论——可能讨论得比实践还要多得多。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、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、日程安排都当作是尝试性的，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，他必须能够为他的每个结果进行辩护。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，整个团队可以灵活地安排。在大型的项目中，项目经理需要有两个和三个顶级程序员作为技术轻骑兵，当工作繁忙最密集的时候，他们能急驰飞奔，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，如 Bell Labs 的一些实验室，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 IBM 的另外一些实验室，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。

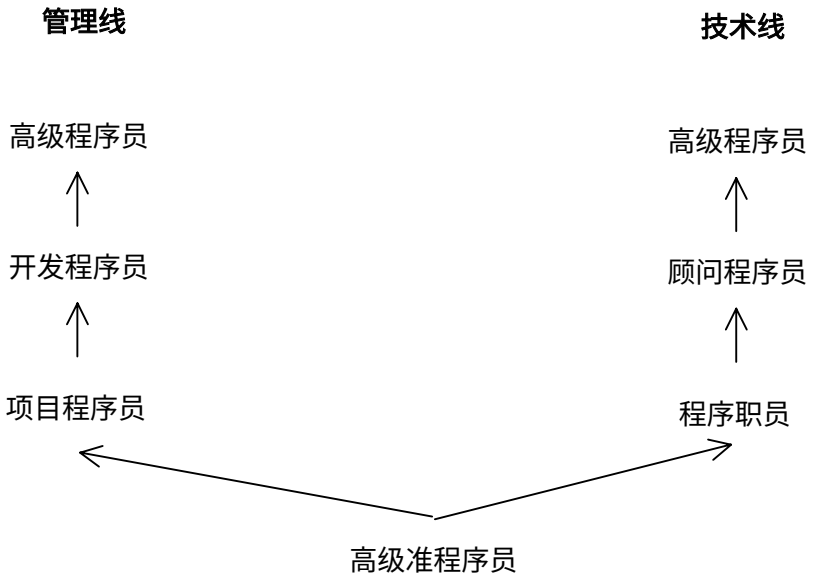


图 11.1：IBM 的两条职位晋升线

很容易为上述层次建立相互一致的薪水级别。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。秘书和其他支持也必须相同。从技术线向管理同级调动时，不能伴随着待遇的提升，而且应该以“ 调动 ”，而不是“ 晋升 ”的名义。相反的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、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，以管理团队或者亲自参与开发工作。这是件工作量很大的任务，但显然很值得！

组建外科手术队伍式的软件开发团队，这整个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创造性乐趣的社会障碍。

另外，上述组织架构的设计是为了最小化成员间的接口。同样的，它使系统在最大程度上易于修改。当组织构架必须变化时，为整个“ 外科手术队伍 ”重新安排不同的软件开发任务，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，它不会停止变化。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、润滑和对损坏器件的修复。它主要包含对设计缺陷的修复。和硬件维护相比，这些软件变更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严重影响。用户越多，所发现的错误也越多。

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。解决了这些问题之后，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到达了新的熟练水平，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]

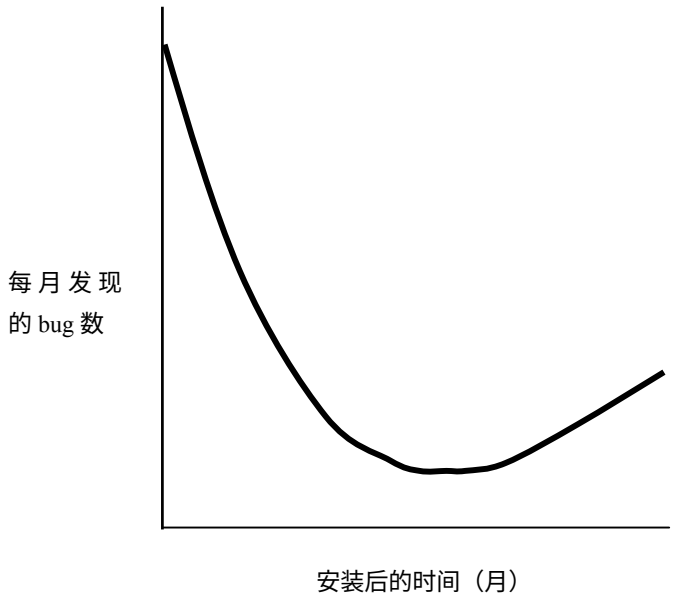


图 11.2：出现的 bug 数量是发布时间的函数

程序维护中的一个基本问题是——缺陷修复总会以（20－50）%的机率引入新的 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，看上去很轻微的错误，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。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、至少是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]。他们发现模块数量随版本号的增加呈线性增长，但是受到影响的模块以版本号指数的级别增长。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。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的漏洞所引起修复工作越来越多。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。尽管理论上系统一直可用，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，所以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。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。一些致命的缺陷会将自私和残酷的人带到塔尖，接着一切开始滑落，回到到痛苦和堕落。实际上，机器失灵了。看上去，就好像是机器正常启动，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。

干将莫邪（*Sharp Tools*）

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、磁盘实用程序等工具。

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，个性化的工具妨碍——而不是促进沟通。其次，当机器和语言发生变化时，技术也会随之变化，所有工具的生命周期是很短的。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－老板使用工具。同时，他还能编制老板需要的专业工具。

因此，项目经理应该制订一套策略，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，对这类工具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，将所有分散的人员集结起来，形成一个公共的工具小组，会有更高的效率。但实际上却不是这样。

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*。接下面我们逐一讨论它们[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操作系统，则该机器不仅充当目标机器的角色，同时也作为辅助机器。

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即时更新和实时可用的。

如果还需要其他的机器，那么将是一件很古怪的东西——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/每分的打字机速度要快。大容量内存可以进行进程覆盖（overlay）和功能测试之后的剪裁工作，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，能查出程序中不可思议的行为或者性能意外下降的原因。

计划安排。当目标机器刚刚被研制，或者当它的第一个操作系统被开发时，机器时间是非常匮乏的，时间的调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，这时资源配给成了严重问题。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。

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运算任务。我们尝试了每天运行四次（周转时间为两个半小时），而实际要求的周转时间为四小时。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，监督时间周期。

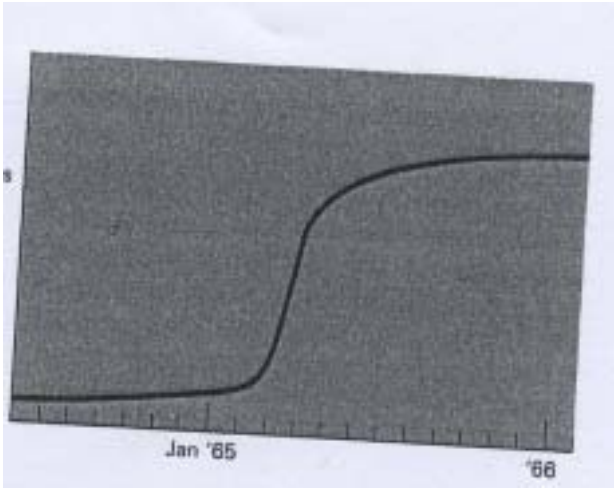


图 12.1：目标机器使用的增长曲线

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、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 至 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，小组通常需要一到两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。二十年前，当所有机房负责人在家中安睡时，我正工作在 701 上。三代机器过去了，技术完全改变了，操作系统出现了，然而大家喜好的工作方式没有改变。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，我们已经习惯于计算机硬件自始至终能正常工作。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，他都会被建议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和早期硬件*不会*像定义的那样运行，*不会*稳定工作，甚至每天都*不会*一样。当一些缺陷被发现时，所有的机器拷贝，包括软件编程小组所使用的，都会发生修改。这种飘忽不定的开发基础实在是够糟的。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——可能根本就没有问题。所以，一套运行在稳定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，可以在仿真器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，一个非常成功的重要辅助机器应用是维护程序库。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 同时还提供 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例程和数据。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，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。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。

这有两个重要的理念。首先是*受控，*即程序的拷贝属于经理，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（playpen）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，很多大型的项目都独立地发展了这些技术[2]，包括 Bell 试验室、ICL、剑桥大学等。它同样适用于文档，是一种不可缺少的技术。

编程工具。随着调试技术的出现，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。

与之类似，一整套实用程序同样是必要的，用来实现磁带走带、拷贝磁盘、打印文件、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 英尺的 OS/360 手册，很多人认为它表达的是一大堆口头垃圾，巨大容量带来了新的不理解问题——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 的文档规模是不可避免的，需要制订仔细的阅读计划。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，它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（*“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。正如我们将在下章讨论到的，彻底地开发一个。使用相同的自顶向下设计方法，来实现性能仿真器、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

在十年前的 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。他们是（1）高级语言和（2）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，技术上的困难很快就不再成为借口。

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（第 8 章）。其中，并没有提到大量的数字论据，但是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类似的错误，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。我无法想象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：它无法完成我想做的事情；目标代码过于庞大；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，可以将其中的百分之一至五替换成手写的代码，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，将它们翻译成某个系统环境下的 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，以及协助终端用户共同开发的设施。我确信在某些应用上，批处理系统决不会被交互式系统所取代。但是，我认为 Multics 小组是交互式系统开发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了开发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，来自 Bell 实验室 John Harr 的论文。它们如图 12.2 所示。这些数字分别反映了代码编写、汇编装配和程序调试的情况。第一个大部分是控制程序；其他三个则是语言解释、编辑等程序。Harr 的数据表明了系统软件开发中，交互式编程的生产率至少是原来的两倍[8]。

程序	规模	批处理（B）或交互式（C）	指令/人年
ESS 代码	800,000	B	500-1000
7094 ESS 支持	120,000	B	2100-3400
360 ESS 支持	32,000	C	8000
360 ESS 支持	8,300	B	4000

图 12.2：批处理和交互式编程生产率的对比

由于远程键盘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进行开发。有了高级语言，可以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。

整体部分（*The Whole and the Parts*）

我能召唤遥远的精灵。

那又怎么样，我也可以，谁都可以，问题是你真的召唤的时候，它们会来吗？

- *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*

和古老的神话里一样，现代神话里也总有一些爱吹嘘的人：“我可以编写控制航空货运、拦截弹道导弹、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，但是其他人成功了吗？”

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。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以及 bug 更不容易产生。

上述方法所意味的详尽体系结构设计正是出于这个目的。Bell 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源于那些产品未精确定义的地方。[1]”细致的功能定义、详细的规格说明、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测试小组，以详细地检查说明的完整性和明确性。如同 Vyssotsky 所述，开发人员自己不会完成这项工作：“ 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。”

自顶向下的设计。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所使用的设计流程[2]形式化。尽管他的理念是为了程序设计，同样也完全适用于复杂系统的软件开发设计。他将程序开发划分成体系结构设计、设计实现和物理编码实现，每个步骤可以使用自顶向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*。开始是勾画出能得到主要结果的，但比较粗略的任务定义和大概的解决方案。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，后者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和其他的工作独立，而模块的大小程度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，尽可能使用级别较高的表达方法来表现概念和隐藏细节，除非有必要进行进一步的细化。

好的自顶向下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的描述。其次，模块分割和模块独立性避免了系统级的 bug。另外，细节的隐藏使结构上的缺陷更加容易识别。第四，设计在每个精化步骤的层次上是可以测试的，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反过来，直到推翻顶层设计，重新开始整个过程。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某个臃肿的设计，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，而对它添加了很多表面装饰般的补丁。自顶向下的方法减少了这样的企图。

我确信在十年内，自顶向下进行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]。Bohm 和

Jacopini 的为其提供了理论证明[4]。

基本上，该方法所设计程序的控制结构，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。Bohm 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。

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有效，例如，在多个条件下的多路分支（CASE、SWITCH 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，而不是独立的跳转语句。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二十年中有过很多反复，甚至在某些方面，它们又回到了出发的起点。整个调试过程有四个步骤，跟随这个过程来检验每个步骤各自的动机是一件很有趣的事情。

本机调试。早期的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。这使得磁带输入/输出对于调试是不可忍受的。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。

在那种情况下，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。花费在编写调试程序上的时间，可能是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，并对执行终止位置进行计划的前提下，粗暴地按下“ 开始（START）”。

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，但是计算机的资源非常匮乏，成本很高。想象一下计算机时间的浪费，那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，但它的方式比以前更为含混，并且发生在测试执行之后。特定用户调试用的时间更长，因为测试依赖于批处理的周期。总之，整个过程的设计是为了减少计算机的使用时间，从而尽可能满足更多的用户。

快照。采用内存转储技术的机器往往配有 2000～4000 个字（word 双字节），或者 8K～16K 字节的内存。但是，随着内存的规模不断增长，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]都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，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，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。Strachy 的想法不断得到改进，终于在 1963 年由 MIT 的 Corbato 和他的同事在 7090 的实验性系统上实现[7]。这个开发结果导致了 MULTICS、TSS 和现在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工具性软件、调度监控程序和其它相关语言解释编译器的出现。而现在，已经可以用高级语言来编程和调试，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，但前者并没有象后者要求的那样，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三倍[8]。这强烈地暗示着，由于缺乏对调试会话的计划，我们没有发掘交互式调试的潜力，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。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，研究和解释调试中出现的奇怪现象。剩余一半时间用于准备：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，Grunberger 提出了特别好的对策[9]，在其他的文章中，也有较为简便的方法[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，需要一种完备系统化和可计划的方法来降低它的困难程度。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），越早将各个部分合拢，系统 bug 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——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。

更微妙的一种方法是“文档化的 bug”。它申明构件单元所有的缺陷已经被*发现*，还没有被*修复*，但已经做好了系统调试的准备。在系统测试期间，依照该理论，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）*，实际上并不常见。不过 OS/360 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）*。用来测试数据发生器、特殊的打印输出、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。对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；以及一个安全版本，其他人员可以在该拷贝上工作，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，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，看上去就像伸着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！这正是系统测试所关注的方面。我们必须假设系统中存在着许多错误，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，在添加了新构件之后，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不时出现在我们面前，带着他们工作的最新版本——更快、更卓越、更完整，或者公认 bug 更少的版本。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 出示了证据，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，我的经验也同样证实了这一点——因此我决不会在实践中冒险采用后一种策略。

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。

祸起萧墙（*Hatching a Catastrophe*）

带来坏消息的人不受欢迎。

- *索福克里斯*

项目是怎样延迟了整整一年的时间？…一次一天。

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是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，由于雷击打坏了公司的供电变压器，所有机器无法启动。明天，因为工厂磁盘供货延迟了一周，磁盘例程的测试无法进行。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项目？第一个步骤是*制订*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，在代码编写时间达到一半的时候就

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、调试。

里程碑有明显边界和没有歧义，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，以致于无法自欺欺人时，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。研究结果显示：

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的糟糕，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下降。

3. *过低估计*在活动中不会有太大的变化，一直到计划的结束日期之前大约三周左右。

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，而不确切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，它会彻底碾碎小组的士气。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*这种心理素质，是很多优秀队员和团队不可缺少的。它表现为“ 要求跑得更快 ”，“ 要求移动得更加迅速 ”，“ 更加努力尝试 ”。对软件开发队伍，*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异常事件，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，会对进取超前会造成一些消极的影

响——这时，人们往往会比较乐观地放缓工作节奏。就这一点来说，它们是令人扫兴的事情。不过，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。它显示谁需要什么样的东西，谁位于关键路径上，他的工作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时，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三次，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产生额外的工作量，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、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，运用才智制订下一份 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：需要采取行动的计划方面的问题，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，但得到状态的真相是很困难的。

一线经理的利益和老板的利益是内在冲突的。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，搞乱了其他计划。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。这样的反应肯定压制信息的完全公开。

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

如果老板把会见、评审、会议明显标记为*状态检查（status-meeting）*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：手册（SLR）的批准时间有所冲突，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 号会议的议程，使得每个人都知道问题的所在，而产品构件经理应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
DS/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	10/28/4	C	10/13/4	C	11/13/4	C	01/15/5	C				09/01/5
		12/31/5				01/11/5		11/18/4	A	02/22/5					11/30/5
FORTRAN	POK	04/--/4	C	10/28/4	C	10/21/4	C	12/17/4	C	01/15/5	C				09/01/5
		12/31/5				01/22/5		12/19/4	A	02/22/5					11/30/5
COBOL	ENDICOTT	04/--/4	C	10/28/4	C	10/15/4	C	11/17/4	C	01/15/5	C				09/01/5
		12/31/5				01/20/5	A	12/08/4	A	02/22/5					11/30/5
RPG	SAN JOSE	04/--/4	C	10/28/4	C	09/30/4	C	12/02/4	C	01/15/5	C				09/01/5
		12/31/5				01/05/5	A	01/18/5	A	02/22/5					11/30/5
UTILITIES	TIME/LIFE	04/--/4	C	06/24/4	C			11/20/4	C						09/01/5
		12/31/5						11/30/4	A						11/30/5
SORT 1	POK	04/--/4	C	10/28/4	C	10/19/4	C	11/12/4	C	01/15/5	C				09/01/5
		12/31/5				01/11/5		11/30/4	A	03/22/5					11/30/5
SORT 2	POK	04/--/4	C	10/28/4	C	10/19/4	C	11/12/4	C	01/15/5	C				03/01/6
		06/30/6				01/11/5		11/30/4	A	03/22/5					05/30/6
44K DESIGN LEVEL (F)															
ASSEMBLY	SAN JOSE	04/--/4	C	10/28/4	C	10/13/4	C	11/13/4	C	02/15/5					02/01/5
		12/31/5				01/11/5		11/18/4	A	03/22/5					11/30/5
COBOL	TIME/LIFE	04/--/4	C	10/28/4	C	10/15/4	C	11/17/4	C	02/15/5					03/01/6
		06/30/6				01/20/5	A	12/08/4	A	03/22/5					05/30/6
MPL	HURSLEY	04/--/4	C	10/28/4	C										
		03/31/6													
2250	KINGSTON	03/30/4	C	11/05/4	C	12/08/4	C	01/12/5	C	01/04/5	C				01/03/6
		03/31/6				01/04/5		01/29/5		01/29/5					NE
2280	KINGSTON	06/30/4	C	11/05/4	C					04/01/5					01/28/6
		09/30/6								04/30/5					NE
200K DESIGN LEVEL (H)															
ASSEMBLY	TIME/LIFE			10/28/4	C										
FORTRAN	POK	04/--/4	C	10/28/4	C	10/16/4	C	11/11/4	C	02/15/5					03/01/6
		06/30/6				01/11/5		12/16/4	A	03/22/5					05/30/6
MPL	HURSLEY	04/--/4	C	10/28/4	C					07/--/5					01/--/7
		03/31/7													
MPL H	POK	04/--/4	C	03/30/4	C					02/01/5					10/15/5
										04/01/5					12/15/5

注：

- SYSTEM/360 SUMMARY STATUS REPORT －SYSTEM/360 总结状态报告

- OS/360 LANGUAGE　PROCESSORS + SERVICE PROGAMS－OS/360 语言处理器 + 服务程序
- AS OF FEBRAURAY 01.1965－1965 年 2 月 1 号
- APPOVEL－批准
- COMPLETED－完成
- PROJECT－项目
- LOCATION－地点
- COMMITMNT ANNOUNCE RELEASE－计划 发布
- OBJECTIVE AVAILABLE APPROVED－目标制订 批准
- SPECS AVAILABLE APPROVED－规格说明提交 批准
- SRL AVAILABLE APPROVED－SRL 提交 批准
- ALPHA TEST ENTRY EXIT－ALPHA 测试 进入 退出
- COMP TEST START COMPLETE－单元测试 开始 结束
- SYS TEST START COMPLETE－系统测试 开始 结束
- BULLETIN AVAILABLE APPROVED－公告发布 批准
- BETA TEST ENTRY EXIT－BETA 测试 进入 退出

图 14.1

Bell 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，而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哪些地方如果他不采取任何措施，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（一至三个人）来关注它的更新、修订和报告，这个小组可以看作是老板的延伸。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，以及里程碑是否被达到。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——仅仅需要作出决策。

我们拥有一个富有热情的、有经验的、熟练的计划和控制小组。这个小组由 A. M. Pietrasanta 负责，他投入了大量创造天分来设计有效的、谦逊的控制方法。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而言，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。它对项目的贡献方式和直接开发软件产品有很大的不同。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。他们是早期预警系统，防止项目以一次一天的方式落后一年。

另外一面（*The other face*）

不了解，就无法真正拥有。

- *歌德*

- *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[jypan1]

- *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呈现面貌：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面貌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，但是连一台收银机也没有卖出去。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，经理身体力行地*演示了如何*出售收银机。事实证明，这个方法是可行的。

我曾经非常勤奋地给我的软件工程师们举办了多年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劝诫以上的观点。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。结果显示，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，形容了树叶，但却没有一副森林的图案。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和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更多的信息。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。和一般用户一样，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，或者是对文档中类似描述的引用。

3. 对所有文件规划的解释。

4. 数据流的概要描述——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、功能回调的位置以及出口；原作者对可能会扩充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，整体结构的概观就严重地被破坏了。

因此，一页纸的流程图，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。

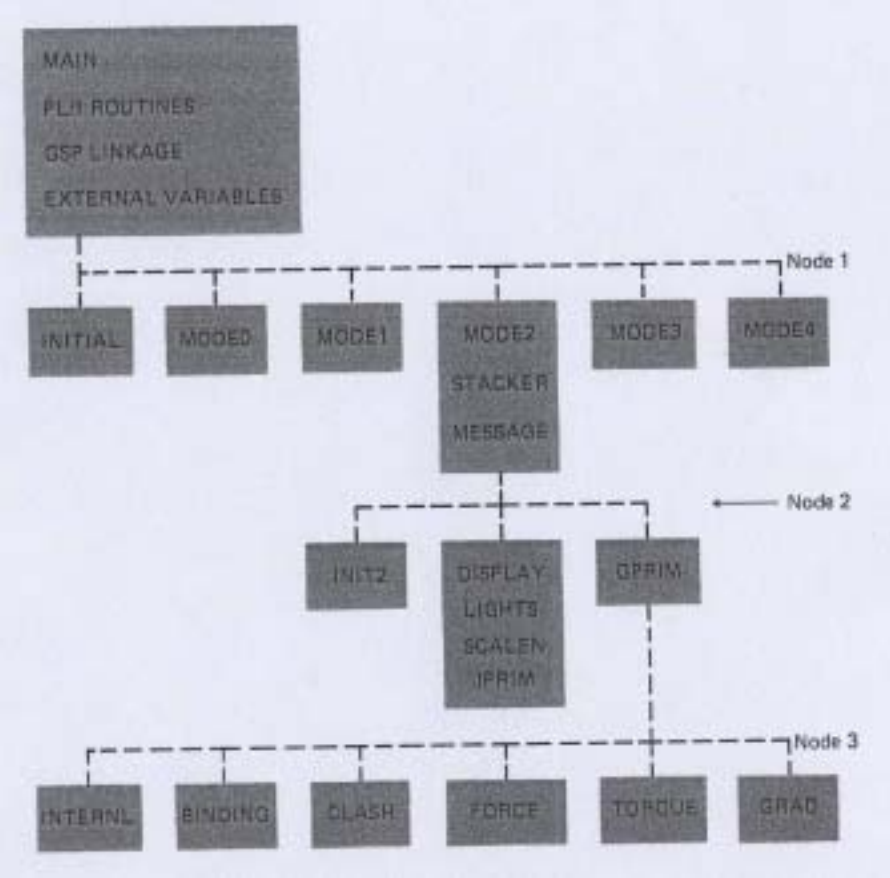


图 15.1：程序结构图（Courtesy of W. V. Wright）

当然，上述图纸既没有，也不需要遵循精心制订的 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 Neumann[1]引入这种方法时，框图和框图中的内容作为一种高级别语言，将难以理解的机器语言组合成一连串可理解的步骤。如同早期 Iverson 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（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，剩下的只有箭头。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，那么所有的箭头都消失了，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。

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。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道，“ 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，我持有相同的观点。

自文档化（self-documenting）的程序

数据处理的基本原理告诉我们，试图把信息放在不同的文件中，并努力维持它们之间的同步，是一种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我们自己的原则。典型的，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 ”，即把文档整合到源代码。这对正确维护是直接有力的推动，保证编程用户能方便、即时地得到文档资料。这种程序被称为*自文档化*

(self-documenting)。

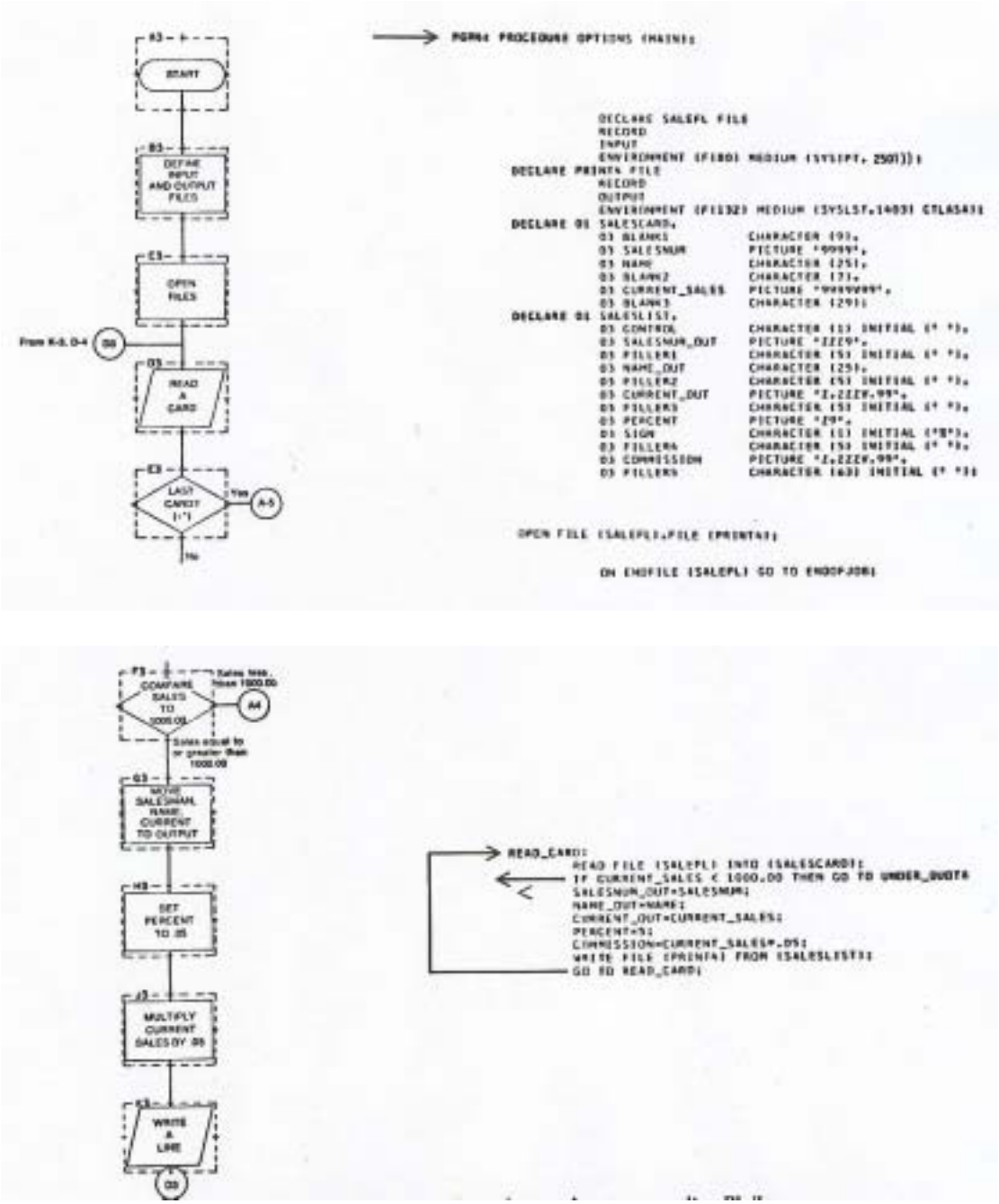


图 15.2：流程图和对应程序的对比[节选自 Thomas J. Cashman 和 Willian J. Keys（Harper & Row，1971）所著的“ *Data Processing and Computer Programming: A Modular Approach* ”中的图 15－41、15－44]

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。

文档是我们以及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、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。大多数文档一般都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，通常就包含了很多注释。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。

```
①//QLT4 JOB ...
②QLTSRT7: PROCEDURE (V);
    /**************************************************************************/
③   /*A SORT SUBROUTINE FOR 2500 6-BYTE FIELDS, PASSED AS THE VECTOR V.  A    */
    /*SEPARATELY COMPILED, NOT-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.                                                       */

⑦INIT: 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;
                                                                     ⑩
⑧K0:  K = -1;
 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      ⑫          /*                                        |||  */
         ⑨DO;                 /*                                        |||  */
 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 15.3：一段子文档化程序

1. 为每次运算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（4）组成，那么后缀可以作为运算编号，把列表和日志联系在一起。这种技术要求为每次运算准备新的任务卡，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，同时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，以显示与设计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。它们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. 使用行注释来解释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，表示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，这成为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，与此同时文本编辑的访问和修改，也在朝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，每个字符需要的击键次数也更少，并且电子草稿不需要重复打印。

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，那么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文档中。这显然是一种聪明的作法。

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。

自文档化方法激发了高级语言的使用，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，而不是人为机器服务。因此从各个方面而言，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。

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*）

没有任何技术或管理上的进展，能够独立地许诺十年内使生产率、可靠性或简洁性获得数量级上的进步。

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——打造由抽象软件实体构成的复杂概念结构，次要任务——使用编程语言表达这些抽象实体，在空间和时间限制内将它们映射成机器语言。软件生产率在近年内取得的巨大进步来自对后天障碍的突破，例如硬件的限制、笨拙的编程语言、机器时间的缺乏等等。这些障碍使次要任务实施起来异常艰难，相对必要任务而言，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，也不会给生产率带来数量级上的提高。

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。
- 在获取和制订软件需求时，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降低的尚方宝剑。

但是，我们看看近十年来的情况，没有银弹的踪迹。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。本章中，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，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上的提高。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，伴随着辛勤的劳动，对规范化过程应进行持续不懈的努力。由此，诞生了现在的软件工程。

是否一定那么困难呢？——根本困难

不仅仅是在目力所及的范围内，没有发现银弹，而且软件的特性本身也导致了不大可能有任何的发明创新——能够像计算机硬件工业中的微电子器件、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一倍的增长。

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降低成本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，让我们通过观察预期的软件技术产业发展速度，来了解中间的困难。效仿亚里士多德，我将它们分成*根本的*——软件特性中固有的困难，*次要的*——出现在目前生产上的，但并非那些与生俱来的困难。

我们在下一章中讨论次要问题。首先，来关注内在、必要的问题。

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、功能调用等等。这些要素本身是抽象的，体现在相同的概念构架中，可以存在不同的表现形式。尽管如此，它仍然是内容丰富和高度精确的。

*我认为软件开发中困难的部分是规格化、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，我们还是会犯一些语法错误，但是和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，那么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要复杂，因为没有任何两个软件部分是相同的（至少是在语句的级别）。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。计算机拥有大量的状态，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系统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，因此整个软件的复杂度以更大的非线性级数增长。

软件的复杂度是必要属性，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建立模型以简化复杂的现象，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必要属性。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问题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，列举和理解所有可能的状态十分困难，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，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上的困难，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问题。物理学家甚至在非常“基础”的级别上，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，他必须控制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。

某些情况下，因为是开发最新的软件，所以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，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，它们会被后续模型所取代，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。汽车的更改十分罕见，计算机的现场调整时有发生。然而，它们和软件的现场修改比起来，都要少很多。

其中部分的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人的念头。

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的计算机硬件平台要长。即使不是更换计算机，则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、计算机硬件等等。后者持续不断地变化着，这些变化无情地强迫着软件随之变化。

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，忽略的地方变得明显。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，我们发现它不仅仅包含一个，而是很多相互关联、重叠在一起的图形。这些图形可能描绘控制流程、数据流、依赖关系、时间序列、名字空间的相互关系等等。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思路。这种缺憾不仅限制了个人的设计过程，也严重地阻碍了相互之间的交流。

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。勿庸置疑，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五倍，同时可靠性、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？首先，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、磁盘等等。高级语言所达到的抽象程度包含了（抽象）程序所需要的要素，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提供所有编程人员在抽象程序中能想到的要素。可以肯定的是，我们思考数据结构、数据类型和操作的速度稳固提高，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上的负担。

分时。大多数观察者相信分时提高了程序员的生产率和产品的质量，尽管它带来的进步不如高级语言。

分时解决了完全不同的困难。分时保证了及时性，从而使我们能维持对复杂程度的一个总体把握。批处理编程的较长周转时间意味着不可避免会遗忘一些细枝末节，如果我们停下编程，调用编译程序或者执行程序，思维上的中断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。

较长的周转时间和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。分时所起作用也非常有限。主要效果是缩短了系统的响应时间。随着它接近于零，到达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无了。

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，并且使生产率提高了 5 倍。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。我们将在下章中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？它们是属于必要问题，或者依然是解决我们剩下的次要困难？它们是提供了创新，还是仅仅是增量改进？

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 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，所有 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。60 年代，曾在内存和速度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，这类语言出现最大的回报来自出现时的冲击，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，解决剩余部分的获益必然也要少一些。

我预言在以后的十年中，当 Ada 的效率被大家评估认可时，它会带来相当大的变化，这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，即对开发人员需要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，是允许定义可以被后续子类型精化的通用接口。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什么新的信息。对于抽象数据类型和层次化类型，它们都是解决了高级别的次要困难和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 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。追究其原因，我们必须剖析“人工智能”意味着什么，以及它如何应用。

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：使用启发式和基于规则的特定编程技术。在这种方法中，对人类专家进行研究，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……不幸的是，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。语音识别技术与图象识别技术的共同点非常少，它们与专

家系统中应用的技术不同。例如，我觉得很难去发现图象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、被最大范围使用的部分，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[5]。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。

- 推导引擎技术的开发独立于应用程序，因此可以用于多个用户。在该引擎上花费较大的工作是很合理的。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，所以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性与程序本身相分离。

如何把它应用在软件开发工作中？可以通过很多途径：建议接口规则、制订测试策略、记录各种 bug 产生的频率、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，对可能难以寻找的原因提供基本的建议。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，该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时，它产生的推测和推荐的测试也越来越准确。该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，因此当产品模块化修改时，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，那么事实上它就可以减少测试用例设计的总体工作量，以及帮助整个软件生命周期的维护、修改和测试。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、深刻理解为什么的分析专家；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，用最优秀开发者的经验和知识积累为他们提供了指导。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。近四十年中，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，声称它本身是语义不完整的。

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，并经常地用于排序程序中。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出特征。
- 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，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和可视化编程，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，并为绘制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和激动的进步。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软件结构表达方法[10]。实际上，它最好被视为是冯．诺依曼、戈尔德斯廷和勃克斯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，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。流程图已经成为完全不必要的设计工具——程序员在开发之后，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，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 ”。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，如同我们上面所争论的，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、层次化数据结构等等，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，它的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，能够在系统设计级别、源代码级别消除 bug 呢？是否可以在大量工作被投入到实现和测试之前，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？

我并不认为在这里能找到魔法。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，以致于只有少量的程序能够得到验证。

程序验证不意味着零缺陷的程序。这里并没有什么魔术，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，但却不能省略程序测试。

更严肃地说，完美的程序验证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对技术规格说明进行测试。

环境和工具。向更好的编程开发环境开发中投入，我们可以期待得到多少回报呢？人们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，以及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上的一些提高。但是，由于它自身的特性，目前它的回报很有限。

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？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。编译还需要一些提高，不过一旦机器运算速度提高十倍，那么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。

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：

$$任务时间 = \Sigma\ （频率）_i \times （时间）_i$$

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，那么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，其中的一些非常有希望。

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。

情况每一天都有些好转，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学时，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，我已在其他的地方提出一些建议。

以上提到的任何软件，购买都要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成本，而不是复制的成本。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 1950～1960 年期间，一个接一个的研究显示，用户不会在工资系统、物流控制、帐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 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。什么导致了这样的变化？

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客户化一些，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 年以前更加不同和复杂。

重大的变化在于计算机硬件/软件成本比率。在 1960 年，2 百万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。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，除了电子表格和简单的数据库系统。这些强大的工具，出现得如此之晚和如此醒目，导致无数应用中的一些并不十分规范。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所取代。

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，他们中间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，并且把这些人员散布到各个岗位。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，同样地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。需求工作对系统的影响比其他任何一个部分的失误都大，当然纠正需求的困难也比其他任何一个部分要大。

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，客户不知道他们自己需要什么。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。原型不必受到相同硬件速度、规模或者成本约束的限制。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，从而客户可以测试一致性和可用性。

现在的软件开发流程基于如下的假设——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往复地开发和规格化。

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，我所感受到的震动。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和我考虑的一样，那些概念性的结构非常复杂，以致于难以事先精确地说明和零缺陷地开发，那么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。光是大脑本身，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 Mill 建议所有软件系统都应该

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，其效果不可思议。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顶向下设计，因为它本身是一种自顶向下增长的软件。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情况的新模块，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了出来。当一个新图形软件系统的第一副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。在开发过程中的每个阶段，总有可运行的系统。我发现开发团队可以在四个月内，*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。象软件工程研究所 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。其正确性是勿庸置疑的。

不过，我不认为我们可以用相同的方式取得下一次进步。低劣设计和良好设计之间的区别可能在于设计方法中的完善性，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——就象萨列里和莫扎特。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、更优雅，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，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（图 6.1）。

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，但我认为我们可以着手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，它们中间没有任何的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、安排、个人的设备、差旅费用、人员支持等——必须完全一致。

如何培养杰出的设计人员？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。最好的通常不是那些最有经验的人员。
- ❑ 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。
- ❑ 为每个方面制订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、正式的高级教育和以及短期的课程——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学习的机会。

再论《没有银弹》 （*“No Silver Bullet ” Refired*）

生死有命，富贵在天

- *威廉三世，奥伦治王子*

那些想看到完美方案的人，其实在心底里就认为它们以前不存在，以后也不可能出现。

- *亚历山大·波普，批判散文*

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 e’er shall be.

- *ALEXANDER POPE, AN ESSAY ON CRITISIM*

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》（第 16 章）最初是在 IFIP ’86 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》*杂志上翻印了该文章，封面是一副类似于《*伦敦人狼》*[2]影片的恐怖剧照。同时，还有一栏补充报道《杀死人狼》，描述了银弹将要完成的（现代）神话。在出版以前，我并未注意到补充报道和文字，也没有料到一篇严肃的技术性文字会被这样润色。

Computer 杂志的编辑们是取得他们想要的效果的专家，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。我希望这副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》中声称和断定，在近十年内，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，因此也该看看是否这些预言得到了应验。

*《人月神话》*一文被大量地引用，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，以及将来也不会有。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，我不禁发现在 1986 年～1987 年期间，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。类似的，我很乐意接受银弹已经出现的观点，例如，某个名副其实的中立客户走到面前，并声称，“ 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十倍。”

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。本章中，我将同大家分享这些改进，以及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，我已经尽我所能地清晰表达了《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，这些术语来自亚里斯多德的古老用法[4]。术语“ accidental ”，我不是指“ 偶然发生 ”，也不是指“ 不幸的 ”，而是更接近于“ 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桃乐丝·赛尔丝看待创造性活动的观点，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）”的部分指它的实现过程。

现实问题。对我而言（尽管不是所有人），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来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，那么即使不占用任何时间（除非出现*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[7] 等人在 1959 年的研究。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以及无法忍受的内存限制。

***因为是根本*困难所以*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），有说服力地指出重用和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没有任何处理软件开发中根本困难的希望——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，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。来自一家国际管理咨询公司，MYSIGMA Lars Sodahl 的 Lars Sodahl 和合作伙伴曾写道：

就我的经验而言，在系统工作中所遇到的大多数困难是组织结构上的一些失误征兆。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，软件没有揭示类似的规律性原理，但是解释为什么没有的重担在你的身上。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表达（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。作为一个学科，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，是不大可能产生足够的一致性，能用通用规律进行精确描述。

不过，很多复杂性并不完全是因为和外部世界保持一致，而是因为实现的本身，例如数据结构、算法、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，从而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，对已出版的《没有银弹》进行了很多最仔细的分析。

悲观主义 vs．乐观主义 vs．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 Halel 一样认为《没有银弹》一文过于悲观，从而他提出 “ 但是，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 ”。他们的论调都有些问题。

首先，我的妻子、同事和我的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 “ 我们看看近十年来的情况，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。”它预言了如果 1986 年的很多创新能持续开拓和发展，那么实际上它们的*共同*作用能使生产率获得数量级的提高。随着 1986～1996 十年过去了，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？是否爱因斯坦关于任何物体运动的速度无法超过光速的论断过于 “ 黯淡 ” 或者 “ 令人沮丧 ” 呢？那么哥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 ”。Tuski 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，人们花费了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，那种寻找出路和希望能一劳永逸的愿望，依然十分的强烈。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，因此它们总是得以延续。所以，将圆形变方的论文被发表，恢复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。

我们太过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[11]。

我和 Turski 都坚持认为这个白日梦*限制了向前的发展，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
- 仅仅预言了十年，而不是足够长的时间 “ 出现任何重大的进步。”

第一个主题，它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，依次独立的评估它们是非常公平的。我持反对态度的并不是这些技术，而是那种它们能起到魔术般作用的观点。Glass、Vessey 和 Conger 在他们 1992 年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，选择较短的时间是承认我们并没有足够强的能力可以预见到十年以后的事情。我们中间有谁可以在 1975 年预见到 80 年代的微型计算机革命吗？

对于十年的期限，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，在十年后你将获得成功 ”。另外，硬件的性能/价格比可能每十年就会有成百倍的增长，尽管这种比较不很合适，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 年代价取得数量级的进展，很难被认为是不可思议的进步。

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，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产生了提高若干数量级的效果。

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 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、故障和落后的完成期限。

接着，他阐述了在传统的小型个人程序中，那些假设的错误、故障和落后的最终期限如何在接下来的 25 年中，得到数量级的改进和提高。

但是，50 年代该领域的实际情况并不是小型个人程序。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、导弹性能计算等等 [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车间使用着超过 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 Vanilla 框架有可能是革命性的。我也希望如此。Ken Brooks 在报告中提到，在实际工作中应用时，它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/图形方式来自然地表达：

适当使用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在可视化图形表达方式的周围。首先，我们会使用“合适的”实体和关系来形成概念，然后表达成一系列逐步完善的模型，不断地系统化阐明和精化设计概念。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，每方面象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是会采用文字性描述。

我和 Harel 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简单两维或三维上的映射。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，“ 下个十一月在哪？ ”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。

Jone 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。Jone 提出，“ 不。关注质量，生产率自然会随着提高。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中、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，如果一味追求完美质量，生产率会像 IBM 的航天软件一样再次下降。

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：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担心拥有无法控制的“ 艺术家们 ”而可能导致的巨大灾难，他们往往对异常复杂系统开发承担责任 20。

那么，生产率的情形如何？

生产率数据。生产率的数据非常难以定义、测量和寻找。Capers Jones 相信两个相隔十年、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，我看到了人们的工作获得了 5 倍的提高。”Tom DeMarco 认为“你的期望——十年内，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，不管和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。当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、产品性能和支撑成本，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。比起复杂的大型产品工具，它们更加像电锯、电转和砂磨机。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，象供人们使用的组合工具箱，其中的某些工具会经常被使用，以致于熟能生巧。这种工具必须注重常人使用的方便，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更加容易定制一些，但并不太多。”即使在表面上接受了这种论述，我相信用户察觉到软件包更加通用和易于本地化，这种感觉使用户更容易接受软件包。在我公司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？

本章一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。

使用更大的零件来构建。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*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它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》后的九年中，对面向对象技术的期望稳步增长。为什么增长如此缓慢？理论过多。James Coggins，已经在 *The C++ Report* 做了四年 “ The best of comp.lang.c++ ”专栏的作者，他提出了这样的解释：

*问题是 0-0 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，而不是高层次的抽象。例如，他们开发了很多象链表或集合这样的类，而不是*用户接口、射线束模型*或者*有限元素模型。*不幸的是，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。他赞同自顶向下的设计：

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[0-0]和各种复杂语言的联系已经很紧密。人们并没有被告诉 0-0 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 0-0 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。结果是人们使用这种语言做出不好的设计，没有从中获得什么价值。而一旦获得的价值少，它就不会流行。

资金的先行投入，收益的后期获得。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——主要是培训程序员用很新的方法思考，同时还要把函数打造成通用的类。我认为它的好处是客观实在的，并非仅仅是推测。面向对象应用在整个开发周期中，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。Coggin 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才会异乎寻常的迅速。[22]"

为了预期中的，但是有些不确定的收益，冒着风险投入金钱是投资人每天在做的事情。不过，在很多软件公司中，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 O-O 技术应用迟缓的最大原因。即使如此，在很多机构中，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。软件包只是达到上述目标的方法之一，另外的方法是程序重用。实际上，类的容易重用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，新模式就不再象一开始那么简单。

当然，程序员经常重用他们自己的手头工作。Jone 提到：

大多数有丰富经验的程序员拥有自己的私人开发库，可以使他们使用大约 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，从而培养重用自身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，专业期刊和算法搜集，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很多领域要容易。其他领域中，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地发展起来的。

现在公司级别的重用情况如何？存在着大量的研究。美国国内的实践相对较少，有报道声称国外重用较多[25]。

Jones 报告，在他公司的客户中，所有拥有 5000 名以上程序员的机构都进行正式的重用研究，而 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*机构上*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文档。即使我们

看到了并不十分常见的优秀设计，但如果没有好的文档，我们也不会看到能重用的构件。

Ken Brooks 关于预测产品通用化的一些困难的评论：“ 我不断地进行修改，即使在第五次使用我自己的个人用户界面库的时候。”

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%[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。时间证明了使模块能够重用的成本非常高。

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] ”在第一章的讨论中，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函数库的方式，我们必须面对我们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 3000 个的类库。很多对象需要 10 到 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 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，所以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。语法和语义是在上下文中，通过使用逐渐地学习。
- 人们根据语义上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 年的文字精确地总结了我在 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刻，而不是等待或希望它的出现。

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，以致于可以认为是现实主义者的人——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]。

《人月神话》的观点 :是或非？（*Propositions of* the Mythical Man-Month: *True or False*？）

我们理解的也好，不理解的也好，描述都应该简短精练。

塞缪尔·巴特勒，讽刺诗

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。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——它们是当年我认为将会是正确的 :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 177 页的篇幅来论述？ ”) 方括号中的评论是新增内容。

所有这些观点都是可操作验证的，我将它们表达成刻板的形式是希望能引起读者的思考、判断和讨论。

第 1 章 焦油坑

1.1 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九倍。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。

1.2 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 ”，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，不间断学习的乐趣
- 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

1.3 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一点好一些：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，（bug、工作时间）能收敛得快一些，然而软件项目的情况却是越接近完成，收敛得越慢
- 产品在即将完成时总面临着陈旧过时的威胁

第 2 章 人月神话

2.1 缺乏合理的时间进度是造成项目滞后的最主要原因，它比其他所有因素加起来影响还大。

2.2 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。

2.3 所有的编程人员都是乐观主义者：“ 一切都将运作良好 ”。

2.4 由于编程人员通过纯粹的思维活动来开发，所以我们期待在实现过程中不会碰到困

难。

2.5 但是，我们的构思是有缺陷的，因此总会有 bug。

2.6 我们围绕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。*

2.7 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。

2.8 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。

2.9 作为一个学科，我们缺乏数据估计。

2.10 因为我们对自己的估计技术不确定，所以在管理和客户的压力下，我们常常缺乏坚持的勇气。

2.11 Brook 法则：向进度落后的项目中增加人手，只会使进度更加落后。

2.12 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

3.1 同样有两年经验而且在受到同样的培训的情况下，优秀的专业程序员的工作效率是较差程序员的十倍。（Sackman、Erikson 和 Grand）

3.2 Sackman、Erikson 和 Grand 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。

3.3 小型、精干队伍是最好的——尽可能的少。

3.4 两个人的团队，其中一个项目经理，常常是最佳的人员使用方法。[留意一下上帝对婚姻的设计。]

3.5 对于真正意义上的大型系统，小型精干的队伍太慢了。

3.6 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、速度缓慢、不充分的，开发出的产品无法进行概念上的集成。

3.7 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

4.1 “概念完整性是系统设计中最重要的考虑因素”。

4.2 “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。]

4.3 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。

4.4 “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。”[同样适用于小型项目。]

4.5 “如果要得到系统概念上的完整性，那么必须控制这些概念。这实际上是一种无需任何歉意的贵族专制统治。”

4.6 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。

4.7 概念上统一的系统能更快地开发和测试。

4.8 体系结构 (architecture)、设计实现 (implementation)、物理实现 (realization) 的许多工作可以并发进行。[软件和硬件设计同样可以并行。]

第 5 章 画蛇添足

5.1 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。

5.2 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。

- 对上述的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。

5.3 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。

5.4 OS/360 是典型的画蛇添足（second-system effect）的例子。[Windows NT 似乎是 90 年代的例子。]

5.5 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

6.1 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。

6.2 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。

6.3 出于精确性的考虑，我们需要形式化的设计定义，同样，我们需要记叙性定义来加深理解。

6.4 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。

6.5 设计实现，包括模拟仿真，可以充当一种形式化定义的方法；这种方法有一些严重的缺点。

6.6 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。]

6.7 “ 如果起初至少有两种以上的实现，那么（体系结构）定义会更加整洁，会更加规范。”

6.8 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。[电子邮件是一种可选的介质。]

6.9 “项目经理最好的朋友就是他每天要面对的敌人——独立的产品测试机构/小组。”

第 7 章 为什么巴比伦塔会失败？

7.1 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

7.2 “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。”由于对其他人的各种假设，团队成员之间的理解开始出现偏差。

7.3 团队应该以尽可能多的方式进行相互之间的交流：非正式、常规项目会议，会上进行简要的技术陈述、共享的正式项目工作手册。[以及电子邮件。]

项目工作手册

7.4 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。”

7.5 “项目*所有*的文档都必须是该（工作手册）结构的一部分。”

7.6 需要*尽早*和*仔细*地设计工作手册结构。

7.7 事先制订了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。

7.8 “*每一个*团队成员应该了解*所有*的材料（工作手册）。”[我想说的是，每个团队成员*应该能够*看到所有材料，网页即可满足要求。]

7.9 实时更新是至关重要的。

7.10 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。

7.11　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。

7.12　今天[即使在 1975 年]，共享的电子手册是能更好达到所有这些目标、更加低廉、更加简单的机制。

7.13　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。

7.14　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允许看到其他部分的内部结构，只需要了解接口。

7.15　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

7.16　团队组织的目标是为了减少必要的交流和协作量。

7.17　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）*。

7.18　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。

7.19　组织中的*交流*是网状，而不是树状结构，因而所有的特殊组织机制（往往体现成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。

7.20　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着很大的区别，需要不同的技能。

7.21　两种角色中的任意组合可以是非常有效的：

- ❑　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

8.1 仅仅通过对编码部分的估计，然后乘以任务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。

8.2 *构建独立小型程序的数据不适用于编程系统项目。*

8.3 程序开发呈程序规模的指数增长。

8.4 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]]

8.5 Portman 的 ICL 数据显示相对于其他活动开销，全职程序员仅将 50%的时间用于编程和调试。

8.6 IBM 的 Aron 数据显示，生产率是系统各个部分交互的函数，在 1.5K 千代码行/人年至 10K 千代码行/人年的范围内变化。

8.7 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 0.6KLOC/人年，编译类工作的生产率大约为 2.2KLOC/人年。

8.8 Brooks 的 OS/360S 数据与 Harr 的数据一致：操作系统 0.6～0.8KLOC/人年，编译器 2～3 KLOC/人年。

8.9 Corbato 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 1.2KLOC/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。

8.10 在基本语句级别，生产率看上去是个常数。

8.11 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

9.1 除了运行时间以外，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。

9.2 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要好。规模本身不是坏事，但不必要的规模是不可取的。

9.3 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一样。

9.4 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。

9.5 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。

9.6 在大型的团队中，各个小组倾向于不断地局部优化，以满足自己的目标，而较少考虑队用户的整体影响。这种方向性的问题是大型项目的主要危险。

9.7 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。

9.8 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。

9.9 在早期应该制订策略，以决定用户可选项目的粗细程度，因为将它们作为整体大包能够节省内存空间。[常常还可以节约市场成本。]

9.10 临时空间的尺寸，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。]

9.11 为了取得良好的空间－时间折衷，开发队伍需要得到特定与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。

9.12 编程需要技术积累，每个项目需要自己的标准组件库。

9.13 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。[现在看来有些过时。]

9.14 精炼、充分和快速的程序。往往是*战略性*突破的结果，而不仅仅技巧上的提高。

9.15 这种突破常常是一种新型算法。

9.16 更普遍的是，战略上突破常来自于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

10.1 “前提：在一片文件的汪洋中，少数文档形成了关键的枢纽，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。”

10.2 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。

10.3 对于大学科系，关键文档类似：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。

10.4 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。

10.5 因此，即使是小型项目，项目经理也应该在项目早期规范化上述的一系列文档。

10.6 以上集合中每一个文档的准备工作都将注意力集中在对讨论的思索和提炼，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。

10.7 对每个关键文档的维护提供了状态监督和预警机制。

10.8 每个文档本身就可以作为检查列表或者数据库。

10.9 项目经理的基本职责是使每个人都向着相同的方向前进。

10.10 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。

10.11 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。

10.12 出于这个原因，广受吹捧的市场概念——支持管理人员的‘完备信息管理系统”并不基于反映管理人员行为的有效模型。

第 11 章 未雨绸缪

11.1 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*实验性工厂（pilot planet）*来为提高产量和在缺乏保护的环境下运作提供宝贵经验。

11.2　对于编程产品而言，这样的中间步骤是同样必要的，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。它不同于有限功能的原型，alpha 版本，后者同样是我所倡导的实践。]

11.3　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。

11.4　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。这是个*必须完成*的步骤。

11.5　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。

11.6　因此，*为舍弃而计划，无论如何，你一定要这样做。*

11.7　“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）

11.8　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*。

11.9　软件产品易于掌握的特性和不可见性，导致了它的构建人员（特别容易）面临着永恒的需求变更。

11.10　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。

11.11　为变更计划软件产品的技术，特别是细致的模块接口文档——非常地广为人知，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。]

11.12　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。

11.13　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

11.14　程序员不愿意为设计书写文档的原因，不仅仅是由于惰性。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）

11.15　为变更组建团队比为变更进行设计更加困难。

11.16　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；特别是希望能在技术和管理角色之间自由地分配人手的时候。

11.17　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，人们必须警惕和积极地同它做持续的斗争。

11.18　很容易为不同的晋升线建立相互一致的薪水级别，但要同等威信的建立需要一些强烈的心理措施：相同的办公室、一样的支持和技术调动的优先补偿。

11.19　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

11.20　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、或者是使用环境或者配置变换引起的调整。

11.21　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。

11.22　维护成本受用户数目的严重影响。用户越多，所发现的错误也越多。

11.23　Campbell指出了一个显示产品生命期中每月bug数的有趣曲线，它先是下降，然后攀升。

11.24　缺陷修复总会以（20－50）%的机率引入新的bug。

11.25　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。

11.26　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。

11.27　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——系统熵随时间增加

11.28　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，但是受到影响的模块以版本号指数的级别增长。

11.29　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。即使是最熟练的软件维护工作，也只是放缓了系统退化到不可修复混乱的进程，从中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

12.1　项目经理应该制订一套策略，以及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。

12.2　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即时更新和实时可用的。

12.3　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。

12.4　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。

12.5　同天文工作者一样，系统调试总是大部分在夜间完成。

12.6　抛开理论不谈，一次分配给某个小组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。

12.7　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得以延续 20 年[在 1975 年]，是因为它的生产率最高。[在 1995 年依然如此]

12.8　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，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。

12.9　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。

12.10　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。

12.11　系统文档中的巨大容量带来了新的不理解问题[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。

12.12　自顶向下、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

12.13　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。]

12.14　高级语言不仅仅提升了生产率，而且还改进了调试：bug 更少，以及更容易寻找。

12.15　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。

12.16　现在可供合理选择的语言是 PL/I。[不再正确。]

交互式编程

12.17　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。]

12.18　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。

12.19　有限的数据表明了系统软件开发中，交互式编程的生产率至少是原来的两倍。

第 13 章 整体部分

13.1 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以及 bug 更不容易产生。

13.2 V.A.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。”

13.3 在编写任何代码之前，规格说明必须提交给测试小组，以详细地检查说明的完整性和明确性。开发人员自己不会完成这项工作。（Vyssotsky）

13.4 “ 十年内[1965～1975]，Wirth 的自顶向下进行设计[逐步细化]将会是最重要的新型形式化软件开发方法。”

13.5 Wirth 主张在每个步骤中，尽可能使用级别较高的表达方法。

13.6 好的自顶向下设计从四个方面避免了 bug。

13.7 有时必须回退，推翻顶层设计，重新开始。

13.8 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。这种方法出色地避免了 bug，是一种正确的思考方式。

13.9 Gold 结果显示了，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三倍。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*如此。]

13.10 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：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。

13.11 系统调试（相对于单元测试）花费的时间会比预料的更长。

13.12 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。

13.13 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 “ 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。]

13.14 开发大量的辅助调试平台（scaffolding 脚手架）和测试代码是很值得的，代

码量甚至可能会有测试对象的一半。

13.15　必须有人对变更进行控制和文档化，团队成员应使用开发库的各种受控拷贝来工作。

13.16　系统测试期间，一次只添加一个构件。

13.17　Lehman 和 Belady 出示了证据，变更的阶段（量子）要么很大，间隔很宽；要么小和频繁。后者很容易变得不稳定。[Microsoft 的一个团队使用了非常小的阶段（量子）。结果是每天晚上需要重新编译生成增长中的系统。]

第 14 章　祸起萧墙

14.1　“项目是怎样延迟了整整一年的时间？…一次一天。”

14.2　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。

14.3　根据一个严格的进度表来控制项目的第一个步骤是*制订*进度表，进度表由里程碑和日期组成。

14.4　里程碑必须是具体的、特定的、可度量的事件，能进行清晰能定义。

14.5　如果里程碑定义得非常明确，以致于无法自欺欺人时，程序员很少会就里程碑的进展弄虚作假。

14.6　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，才有所变化。

14.7　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），确保制订下一条 deadline。[2]”]

14.8　*进取*对于杰出的软件开发团队，同优秀的棒球队伍一样，是不可缺少的必要品德。

14.9　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。

14.10　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、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。

14.11　第一份 PERT 图总是很恐怖的，不过人们总是不断进行努力，运用才智制订下一份 PERT 图。

14.12　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。

14.13　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。

14.14　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。

14.15　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。

14.16　必须有评审的机制，从而所有成员可以通过它了解真正的状态。出于这个目的，里程碑的计划和完成文档是关键。

14.17　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这些日期的怀疑。”

14.18　对于大型项目，一个对里程碑报告进行维护的*计划和控制（Plan and Control）*小组是非常可贵的。

第 15 章　另外一面

15.1　对于软件编程产品来说，程序向用户所呈现的面貌与提供给机器识别的内容同样重要。

15.2　即使对于完全开发给自己使用的程序，描述性文字也是必须的，因为它们会被用户－作者所遗忘。

15.3　培训和管理人员基本上没有能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激励作用。

15.4　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。

15.5　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。

15.6　由于关键的用户文档包含了跟软件相关的基本决策，所以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）。

15.7　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。

15.8　对于必须修改程序的人而言，他们所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）。

15.9　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。）

15.10　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 军用标准实在错得很厉害。]

15.11　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。

15.12　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。

15.13　最小化文档负担的 3 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。

15.14　程序修改人员所使用的文档中，除了描述事情如何以外，还应阐述它为什么那样。对于加深理解，*目的*是非常关键的，但即使是高级语言的语法，也不能表达目的。

15.15　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。

原著结束语

E.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)。

E.2 软件工程的焦油坑在将来很长一段时间内会继续地使人们举步维艰，无法自拔。

20 年后的人月神话（The Mythical Man-Month *after* 20 Years）

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*

为什么会出现二十周年纪念版本？

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映。最后，当我们向舱门移动时，我无法再等下去了：

“这本书如何？你有什么评论吗？”

“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。

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 1000 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，我们发现二十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？ ”这些独特的问题是完全平等的，我将尽我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，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个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，像书籍、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。竞争带来了压力，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，但与此同时，还需要在概念上保持与单个使用人员的一致。如何组织设计队伍来获得上述的概念一致性？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，大型编程项目的管理与小型项目在性质上都不同。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 到第 7 章，我一直不断地在表达一个观点——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，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。

体系结构的递归。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的体系结构工作。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的最重要一步。这个原理决不仅限于软件系统，它适用于所有的复杂事物，如计算机、飞机、防御系统、全球定位系统等。在软件工程试验室进行 20 次以上的讲授之后，我开始坚持每 4 个学生左右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是有点极端，但我仍然发现这种方法即使对小型团队也运作良好，并且促使了设计的成功。

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（Featuritis）。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可用性的代价，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添加，手册慢慢地增厚，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，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、明确和共有的用户群描述，结构师应该*猜测（guess），*或者*假设（postulate）*完整的一系列属性和频率值。

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，这能起到解释的作用，以及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。*

“开发第二系统所引起的后果（second-system effect）”情况怎样？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），第二系统在第 5 章中被认为是最危险的系统。

这实际上是语言引起的的差异，现实情况并不是如此。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（WIMP）界面的成功

在过去 20 年内，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，以致于不需要任何解释。这个概念首先在 1968 年西部联合计算机大会（Western Joint Computer Converence）上，由斯坦福研究机构（Stanford Research Institute）的 Doug Englebart 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，以致于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，以及一致、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；图标和嵌套文件夹源于桌面的文档，回收站也是如此；剪切、复制和粘贴则完全反映了我们使用文档的一些习惯；我们甚至可以通过向回收站拖拽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，以致于新用户常常会被它所体现出的理念打动。

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，常常是新产生，而不是选择一个口头或者书面祈使句：“请将这个归档。”“请找出前面一致的地方。”“请把这个交给 Mary 去处理。”

无论是手写还是口头的命令，现有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去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水平菜单和垂直的下拉子菜单中。

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、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）名词的，接着指出一个动词菜单项。这确实牺牲了很多易用性。当我看到用户、或者用户录像、或者光标移动的计算机跟踪情况。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“ 上次在这个空间的什么地方 ”——总而言之，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？这时，光标停留在数据区，为后续点击拾取提供了充分的空间活动能力。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，紧记购买的目的，遵照菜单上不同的动词略为试验一下，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简单操作，还是为专业用户设计强大的功能？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（♂[jypan2]）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快速学习的需要，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，因此拿不准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。一个新手从他最常使用的命令中学习快捷键。由于♂z 可以撤消任何单一操

作，所以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，以确定命令是否有效。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。

强制体系结构的实施，作为设备的直接整合。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程序。从而，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，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，从而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足够广泛的应用，以致于可以形成实际的标准。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。如同我们支配我们机器一样，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。” 现在我觉得这是错误的，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。该模型源自于类似甘特图布局的阶段化流程，常常绘制成图 19.1 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，从而容纳它引起的成本增加和进度延迟

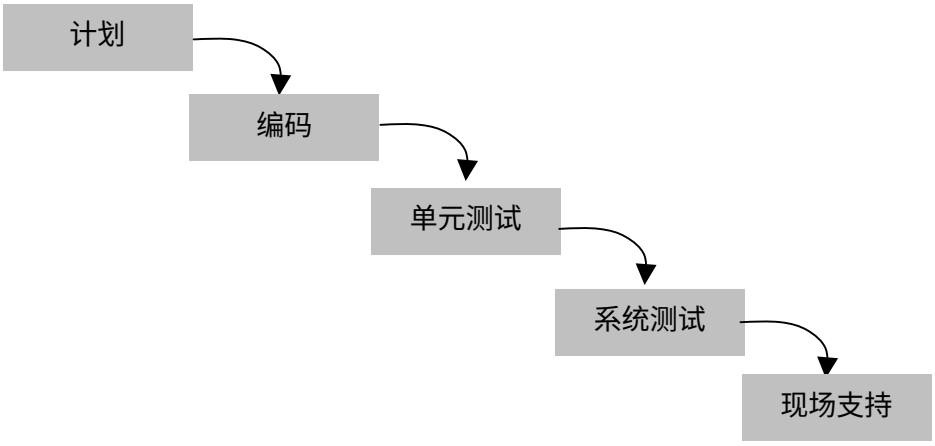


图 19.1：软件开发的瀑布模型

他给开发者提出的建议“构建两次”比《人月神话》的好[8]。受到瀑布模型不良影响并不只是第 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。

实际上，《未雨绸缪》并没有迎面痛击这个错误。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。

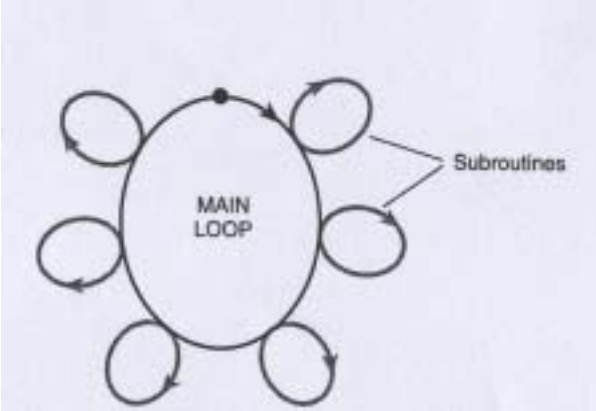
例如，设计实现会发觉有些体系结构的功能定义会削弱性能，从而体系结构必须重新调整。编码实现会发现一些功能会使空间剧增，超过要求，因此必须更改体系结构定义和设计实现。

所以，在把任何东西实现成代码之前，可能要往复迭代两个或更多的体系结构－设计－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，为每个功能都提供子函数调用（*占位符*），但仅仅是空的子函数（图 19.2）。对它进行编译、测试，使它可以不断运行。它不直接完成任何事情，但至少是正常运行的[10]。



注：

- MAIN LOOP－主循环
- Subroutines－子函数

图 19.2

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，那么每个阶段都会有一个经过

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。

因为我们在所有时刻都拥有一个可运行的系统，所以

- 我们可以很早开始用户测试，以及
- 我们可以采用按预算开发的策略，来彻底保证不会出现进度或者预算的超支（以允许的功能牺牲作为代价）。

我曾在北卡罗来纳大学教授软件工程实验课程 22 年，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。大约是一半的时候，我转而教授增量开发的课程。我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。

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，从而搭建一棵相关产品的家族树（图 19.3）。

设计类似一棵树的技巧是将那些变化可能性较小的设计决策放置在树的根部。

这样的设计使得模块的重用最大化。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，而且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。

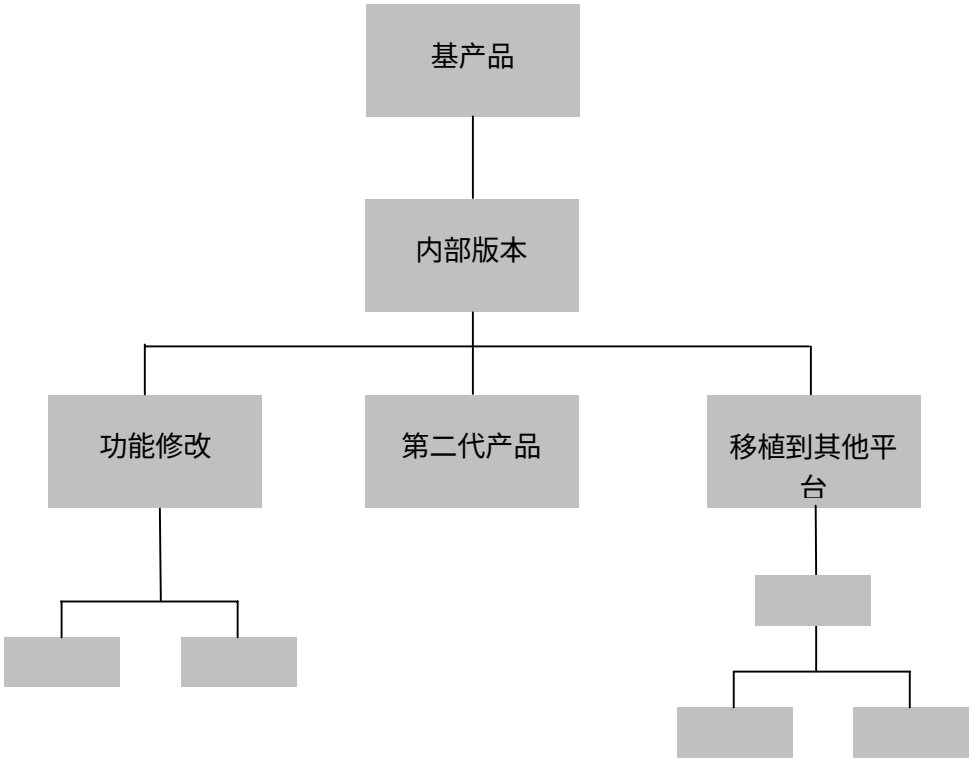


图 19.3

Microsoft 的“每晚重建”方法

Jams 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，

在我们第一次产品发布之后，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，一个和多个程序员-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，我们将停下整个过程，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 千 2 百万行的系统。

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？我认为它们既是互相关联，又是相互独立的。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。

类似的，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，从而可以尽早发现性能上的潜在问题。那么，“ 从第一个里程碑开始构建 ” 的微软流程和快速原型之间的差别是什么？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，而可发布产品和定义中的一样，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，我对比了两种方法。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——每个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内部结构面前。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。Mill 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接口的详细语义。这些接口的隐藏会导致系统的 bug。Parnas 的技术在面对变更时是很健壮的，更加适合为变更设计的理念。

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，那么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的错误可能性。

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它自己的操作来访问。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，以及容易保证实施的类型规范化访问方法。

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）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，模块隐藏，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*）不但包括了很多结果，而且还有一系列逐步推广的成本模型。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，这本书将作为一代经典。

他的结果充分地吻合了《人月神话》的结论，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。特别的，他发现：[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，花的时间也越长。
- 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*》[16]（*Software Project Dynamics：An Integrated Approach*）中。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，指出了在各种假设下的情况，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，但并不一定会使项目更加落后。” 特别的，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加入更

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他的模型，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，他还试验了的其他方法，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，作为真理的最佳近似，以及一项经验法则——警告经理们避免对进度落后的项目采取的盲目、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，较少涉及到技术问题。这种倾向部分因为我在 IBM 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 模型发现团队质量目前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、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 年出版的数据，《*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，而是是创造工作的可能。”它涉及了如空间、布置、团队的餐饮等主题。DeMarco 和 Lister 从他们的 Coding War Games 项目中提供的数据，显示了相同组织中开发人员的表现之间，和工作空间和生产率以及缺陷水平之间令人吃惊的关联。

顶尖人员的空间更加安静、更加私人、保护得更好以免受打断，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？

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，但是非常关键的特性。很多地点分散的公司，项目从一个实验室转移到另一个。从中，我认为团队融合正是管理上被忽视的因素。

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，导致了产品雏形的夭折，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，那么项目管理面对的中心问题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，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理论。他的第一个原理是引自 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的帮助，

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，越强和越有效的社会权威将会是国家更加融洽和繁荣的条件[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大量的自由，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帐目和资产负债表*[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和保留小规模的创造性尚待分晓。

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。

微软的 Jim AcCarthy 向我描述了他在解放团队上的经验：

每个队伍（30 至 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 个专家组成，包括开发、测试和书写文档等。由团队而不是老板对争论进行仲裁。我无法形容授权和由团队自行负责项目的成功与否的重要性。

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。这就像是魔术！改进的质量、提高的生产率、高涨的士气。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，因此感受到市场的压力。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，同时也反映出真正的下放显得多少有些保守。不过，授权是朝着正确方向迈出的一大步，它产生了像 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；从整体而言，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，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》后二十年中最重要的改变。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 20 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
- 适合小型的应用，并且
- 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 20 年前只有国王的薪水才能买得起的软件。Schumacher 的目标值得仔细思考，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 到 6 个月就需要重新输入一遍，这中间，人们往往还不断地产生新文稿。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、制订计划、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。在这些领域中，手工操作都需要重

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，无须失去思维的连贯性，就象分时带给软件开发的好处一样。

同样，新的、灵活的辅助工具增进了创造力。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。

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多种彻底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展现了更多的选择。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日的工作在几分钟内就可以完成。电子表格可以对大量“ what if ”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。至于现在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。70 年代的软件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被消除。快速的个人计算机现在是软件开发者的常规工具，从而周转时间的概念几乎成为了历史。如今的个人计算机不仅仅比 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，从而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所作更改的更多上下文。

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少于开发成本的价格获得。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，如今他们依然存在：

- 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、银行、保险、政府机构等，他们为自己使用的软件开发应用程序包。
- 定制程序开发者：为用户承包开发私用软件包，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，而不仅仅是通用系统分析方法、通用语言、通用测试技术的使用。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 代语言作出了巨大的贡献。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、数据库－通讯软件包的应用生成程序。

操作系统世界已经统一了。在 1975 年，存在着很多操作系统：每个硬件提供商每条产品线最少有一种操作系统，很多提供商甚至有两个。如今是多么的不同啊！开放式系统是基本原则。目前，人们主要在 5 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；而在竞争激烈的开发市场面前，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的开发。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专业化的产品。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 Stack、Excel 模板、Minicard 的特殊 Pascal 函数以及 AutoCad 的 AutoLisp 函数。

元编程。Hypercard Stack、Excel 模板、Minicard 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 60 年代早期，很多计算机提供商和信息管理系统（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。类似的，IBM 的 OS/360 队列远

程通讯访问方法中（Queued Telecommunications Access Method），在遇到机器级别指令之前，人们可以读到若干页汇编语言的通讯程序。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。因为包开发现象并没有影响到一般的MIS编程人员，所以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，而且它们作为广为人知、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是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用成品户：

- 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，向应用程序中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，使用一个和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

个软件集的控制之下，而每个软件通常假设是受自己的控制。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

我曾问过北卡罗来纳州大学化学系的系主任 Jim Ferrell 关于化学工程的历史以及和化学的区别的问题，于是他作了一个 1 小时的出色即兴演说，从很多产品（从钢铁到面包，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化学工程系，来发现、发展和讲授所有过程的共有技术基础。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之间的很多相似之处所震动。Parans 对我写的关于*软件工程（software engineering）*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”十分冒昧。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工程化领域，拥有精确的数学基础。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地使人们举步维艰，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；经论证的管理方法的最佳应用；良好判断的自由发挥；以及能够使我们认识到自己不足和容易犯错的——上帝所赐予的谦卑。

结束语：令人向往、激动人心和充满乐趣的五十年（*Epilogue Fifty Years of Wonder, Excitement, and Joy*）

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 是它的构架设计师，而 IBM 的工程师 Clair Lake，Benjamin Durfee 和 Francis Hamilton 是它的实施设计师。同样令人向往的是读到 Vannevar Bush 1945 年 4 月发表在亚特兰大月刊上的论文“ That We May Think ”（我们的期望？）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，并为用户提供机器从已有的链接以及指明其他的相关链接。

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，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，让我成为了为数不多的那些开开心心做着自己喜欢的工作的人之一。

我实在无法想像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 IBM7030 Stretch 超级计算机，Stretch 在 1961 到 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 1000 倍（如果按定值美元来算，便宜了 5000 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，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。当我在五十年代刚从学校毕业的时候，我能看完当时*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。

注解和参考文献（*Notes and References*）

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. 这导致了巨大的压力, 甚至是限制了在第 7 章中, 所讨论的根本、非正式结构和沟通的演化. 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, 这必须进行技术上进行培训以及集成到原有的结构.
2. International Computers Limited 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.” 在 Wolverton, R. W., “The cost of developing lage-scale software,” *IEEE Trans. on Computers*, C-23, 6(June, 1974) pp.615-636 提出了若干其他的进度划分.
3. 图 2.5 至 2. 8 出自 Jerry Ogdin, 他引用了这章的早期版本, 必须改进相应的描述. Ogdin, J. L., “The Mongolian hordes versus superprogrammer,” *Infosystems* (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,” CAM, 11, 1(Jan., 1968), pp. 3-11.
2. Mills, H., “Chief programmer teams, principles, and procedures,” IBM Federal Systems Division Report FSC 715108, Gaitherburg, 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, 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, 而不是帮助, 但实际情况中完全不是这样. 规范的思维实际上是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. Stuttgard, 1970.
7. 关于编程技术必要性的讨论, 参见 C. H. Reynolds, “What's wrong with computer programming management?” in G. F. Weinwurm (ed.). *On the Management of Computer Programming.* Philadelphia: Auerbach, 197.1 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 systems: 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., 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., 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. 一份未发表的 E. F. Bardain1964 研究指出程序员实际的生产时间占 27%. (为 D. B. Mayer and A. W. Stalnaker 所引用, “Selection and evaluation of computer personnel, ” *Proc. 23d ACM Conf.,* 1968, p. 661.)
7. Aron, J. , Private communication.
8. 材料在小组会议中给出, 没有包括于 the AFIPS Proceedings.
9. Wolverton, R. W. “The cost of developing large-scale software,” *IEEE Trans. On Computers*. C-23, 6, (June,1974), pp. 615-636. 这篇重要新近发表的文章包含的数据核实了生产率方面的结论, 同时还有许多所讨论问题的数据.
10. Corbató, F. J. “Sensitive issues in the design of multi-use systems,” 在好莱坞 EDP 技术中心 1968 年的公开演讲.
11. W. M. Taliaffero 同时指出了在 Fortran 和 Cobol 编译器方面的生产率为 2400 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 Estimation of Computer Programming Costs,* 尽管标注有较大的背离, 仍然显示了高级语言带来了 1 至 3 倍生产率的提高(pp. 66-67).

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．在 Oglethorpe 大学 1932 年 5 月 22 号的演讲.

2. 描述了 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, 这里我过于简化了. 参见 J. H. Saltzer “Evolutionary design of complex systems,” in D. Eckman (ed.), *Systems: Research and Design.* New York: Wiley, 1961. 当所有的事被提出和完成, 我依然提倡构建一个被抛弃的实验性系统.

5. Campbell, E., “Report to the AEC Computer Information Meeting,” December, 1970. 该现象同时有 J. L. Ordin 在“Designing reliable software,” *Datamation*. 18, 7(. July. 1972), pp. 71-78 中讨论. 至于曲线是否会再次下降, 我的具有丰富检验的朋友们各执己见.

6. Lehman, M., and L. Belady, “Programming systems dynamics,” given at the ACM SIGOPS Third Symposium on Operating Systems Principles ,October, 1971.

7. Lewis, C. S., *Mere Christianity.* New York: Macmillan, 1960, p. 54.

第 12 章

1. 参见 J. W. Pomeroy, “A guide to programming tools and techniques,” IBM *Sys. J.*, 11,3(1972), pp. 234-254.

2. Landy B., 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. *Corbato 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. Corbato F. J., J. H. Saltzer, and C. T. Clingen, “MULTICS - the first seven years”, *AFIPS Proc SJCC*, 40(1972) pp. 571-582. *“出于达到最优性能的原因, 仅有半打使用 PL/L 编程的领域重新用汇编进行了改写. 许多最初使用机器语言编写的程序都用 PL/L 重新编写,译提高它们的可维护性.”*

6. 引用 Corbato 论文中的参考资料 3: *"PL/I is here now and the alternatives are still untested"*. 同时,书写良好的提出反面意见的文章, 参见 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 说: "*我的经验开始告诉我, 在产品开发中, 将秘书安排到终端面前. 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. Yarr J., "Programming Experience for the Number 1 Electronic Switching System," paper given at the 1969 SJCC.

第 13 章

1. Vyssotsky V. A., 在 Chapel Hill, N. C1972 年举办的计算机程序测试方法讨论会"Common sense in designing testable software". Vyssitsky 的大多数演讲收录在 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. 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处理. 参见 Dijkstra 的书信 "GOTO statement considered harmful," CACM., 11,3(March, 1968), pp. 147-148.
4. Bohm 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 p.341 上的评论, 他汇报了类似于 Strachey 论文中所建议工作的进展.

7. Corbato F. J., M. Merwin-Daggett, and 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. Carngie-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. 一种良好的规格说明开发和系统构建及测试处理方法由 F. M. Trapnell 提出，“A systematic approach to the development of system programs,” *AFIPS Proc SJCC*, 34, (1969), pp. 41-48.
13. 实时系统需要环境仿真器. 例子参见 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 systems dynamics,” 提出于 ACM SIGOPS Third Symposium on Operating Systems Priciples, October, 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. 更详细的讨论，参见 Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition.* New York: Wiley, 1969. P. 428-230.
4. Private communication.

第 15 章

1. Goldsteine H. H., and J. von. Neumann, 在为 U.S. Army Ordinance Department, 1947; 所提交的报告中“Planning and coding problems for en electronic computing instrument,” Part II, Vol. 1.; 并在 J. von. Neumann, “Collected Works,”中重新发表 A. H. Taub (ed.). Vol. v., New York: Macmillan. P. 80-151.
2. Private communication, 1957. 该观点在 Iverson, K. E., “The use of APL in Teaching,” Yorktown, N.Y.: IBM Corp., 1969 中提出.
3. PL/I 的另外一个例子由 B. Walter and M. Bohl, 在“From better to best - tips for good programming,” Software *Age,* 3, 11(Nov., 1969), pp. 46-50 中提出. 相同的技术可以使用在 Algol 中, 甚至还有一个 Fortran 格式的程序”STYLE”来达到上述效果. 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. 提名为"No Silver Bullet"的论文源自于 Information Processing 1986, 由 H. –J. Kugler (1986)所编辑的 the Proceedings of the IFIP Tenth World Computing Conference, pp. 1069-76. 在 IFIP 和 Elsevier Science B. V., Amsterdam, The Netherlands 的获准后重印.
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,” Software Engineering 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, 2 (Dec., 1985), pp. 1326-1335. Also in *American Scientist*, 73,5 (Sept.-Oct., 1985), pp. 432-440.
6. Balze , R., “A 15-year perspective on automatic programming,” 在 Mostow, 引文中.
7. Mostow, 引文.
8. Parnas, 1985, 引文.

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. 该题目在本书的第 15 章有所讨论.
11. Mills, H., “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

未被引用的材料源自于私下交流.

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. 许多信件和一些回复，出现在 the July, 1987 issue of *Computer*.

   非常高兴地看到《没有银弹》没有接受任何大奖, Bruce M. Skwiersky’s 的评论作为 Computer Reviews 在 1988 年选出的最佳评论. E. A. Weiss, “Editorial,” Computer Reviews (June, 1988), pp. 283-284, 均宣布了上述评论的获奖情况和重新提出了 Skwiersky 的观点. 该评论有一个重大的错误: “sixfold”应该为“10^6”.
4. “根据经院哲学中亚里士多德提出，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, D. L., *The Mind of the Maker.* New York: Harcourt, Brace, 194.
6. Glass, R. L., and S. A. Conger, “Research software talks: Intellectual or clerical?” *Information or Management*, 23, 4 (1992). 作者提出关于软件需求的度量结果是 80%的智力和 20%的书记工作. Fjelstadt and Hamlen, 1979, 对应用软件维护得到了相同的结果. 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,” *Communication 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 AIEEIRE Computer Conference* (Philadelphia, Dec. 10-12, 1951). New York: American Institute of Electrical Engineers. pp. 13-20.
14. *Ibid.,* pp. 36, 68, 71, 97.
15. *Proceedings of the Eastern Joint Computer Conference,* (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. Bennington, “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., Lachover H., Haamad A., Pnueli A., Politi M., Sherman R., Shtul-Traurig A.* “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.* Engltwood Cliffs, N. J.: Prentice-Hall, 1994. p. 619.
20. Coqui, H., “Corporate survival: The software dimension,” *Focus '89*, Cannes, 1989.
21. Coggins, J. 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，没有类似关于第 15 次应用的报告.

23. Jones, 引文, p. 604.

24. Huang, Weigiao, “Industrializing software production,” *Proceedings ACM 1988 Computer Science Conference.* 1988. Atlanta. 我觉得在类似的安排中，缺乏个人工作机会的增长.

25. 关于重用的整个 IEEE Software1994 年 9 月期刊.

26. Jones, 引文, p. 323.

27. Jones, 引文, p. 329.

28. Yourdon, E., *Decline and Fall of the American Programmer*. Englewood Cliffs, N. J.: Yourdon Press, 1992. p. 22.

29. Glass, R. L., “Glass” (专栏), *System Development.* (Jan., 1988), pp. 4-5.

第 18 章

1. Boehm, B. W., *Software Engineering Economics.* Englewood Cliffs, N. J.: Prentice-Hall, 1981. p. 81-84.

2. McCarthy, J., “21 Rules for Delivering Great Software on Time,” Software World USA Conference, Washington (Sept. 1994).

第 19 章

未被引用的材料源自于私下交流.

1. 关于这个痛苦的话题，参见 Niklaus Wirth “A plea for lean software,” *Computer*, 28, 2 (Feb., 1995), pp. 64-68.

2. Coleman, D., “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，一些机器语言和编程语言命令的概率数据被发表. 例子可参见 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，*事先*估计和*实际*情况的比较就更少. Ken Brooks 建议即使只有少数人会作出答复，现在 Internet 上的公告牌为提供成本更低廉的方法，从新产品的预期用户获取数据.

4. Conklin, J., and M. Begeman, “gIBIS: A hypertext Tool for Exploratory Policy Descussion,” *ACM Transactions on Office Information Systems,* Oct. 1988. p. 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). p. 395-410.
6. Apple Computer, Inc., *Macintosh Human Interface Guidelines*, Reading, Mass.: Addison-Wesley, 1992.
7. Apple Desk Top Bus 在电气上可以控制两个鼠标, 但操作系统并未提供类似功能.
8. Royce, W. W., 1970. “Managing the development of large software system, s: Concepts and techniques,” *Proceedings, WESCON* (Aug., 1970). 在 *ICSE 9 Proceedings* 上重新发表. Royce 和其他人均认为软件过程从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, (Feb., 1986), p. 251-257.
9. DOD-STD-2167 重新制订的工作产生了 DOD-STD-2167A (1988), 它允许但并为制订新的模型如螺旋模型等. Boehm 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, 因此依然继续使用瀑布模型. Larry Druffel 和 George Heilmeyer 所领导的国防科学委员会(Defence Science Board Task Force), 在他们 1994 年的报告“Report of the DSB task force on acquiring defense software commercially”中曾提倡大规模的使用更现代的模型.
10. Mills, H., “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), p. 1-9; Parnas, D. L., “Designing software for ease of extension and construction,” *IEEE Trans. on Software Engineering,* SE-5, 2 (March, 1979), p. 128-138.
12. D. Harel, “Biting the silver bullet,” *Computer*, (Jan., 1992), p. 8-20.
13. 信心隐藏方面的开创性文章是: Parnas, D. L., “Information distribution aspects of design methodology,” *Carnegie-Mellon Univ., 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), p. 330-336; Parnas, D. L. (1972). “On the criteria to be used in decomprosing systems into modules,” *Comm. ACM,* 5, 12 (Dec., 1972), p. 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*. Perennian Library Edition. New York: Harper and Row, 1973. P. 244.
21. Schumacher, 引文, p. 34.
22. 一则发人深醒的海报声称: "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), p. 101-108.
24. Unix 的发明人 Ken Thompson of Bell Labs 很早就认识到大屏幕对编程的重要性. 他在他原始的 Tektronix 电子显象管上发明了在两列中显示 120 行代码的方法. 他在整个高速显象管和小型窗口的时代中坚持使用该终端.

索引（*Index*）

英文	*中文*
Abdel-Hamid, T.	N/A
abstract data type	抽象数据类型
accident	次要
accounting	管理
Ada	Ada 语言
administrator	管理员
Adobe Photoshop	N/A
advancement, dual ladder of	两条职位晋升线
advisor, testing	测试顾问系统
Aiken, H. H.	N/A
aiplane-seat metaphor	“飞机坐舱座椅”比喻
Algol	Algol 语言
algorithm	算法
allocation, dynamic memory	动态内存分配
alpha test	alpha 测试
alpha version	alpha 版本
Alto personal workstation	Alto 个人工作站
ANSI	美国国家标准化组织
APL	APL 语言
Apple Computer, inc.	美国 Apple 计算机公司
Apple Desk Top Bus	Apple 桌面总线
Apple Lisa	Apple Lisa 型计算机
Apple Macintosh	Apple Macintosh 型计算机
AppleScript	AppleScript 语言
architect	体系结构师
architecture	体系结构
archive, chronological	根据时间顺序归档
aristocracy	贵族专政
Aristotle	亚里士多德
Aron, J.	N/A
ARPA network	ARPA 网络
artifical intelligence	人工智能
assembler	汇编
authority	权威
AutoCad	AutoCad 软件
AutoLisp	AutoLisp 语言
automatic programming	自动编程
Bach, J.S.	N/A
Backus, J. W.	N/A
Backus-Naur Form	巴科斯范式
Backer, F. T.	N/A
Balzer, R.	N/A
Bardain, E. F.	N/A
barrier, sociological	社会性障碍
Begeman, M.	N/A
Belady, L.	N/A
Bell Northern Research	Bell 北方研究所
Bell Telephone Laboratories	Bell 电话实验室
Bell, C. G.	N/A
Bengough, W.	N/A
Bennington, H. D.	N/A
beta version	beta 版本
Bible	圣经
Bierly, R.	N/A
Blaauw, G. A.	N/A
Bloch, E.	N/A
Blum, B.	N/A
Boehm, B. W.	N/A
Boehm, E. M.	N/A
Boes, H.	N/A
Bohl, M.	N/A
Bohm, C.	N/A
Booch, G.	N/A
Boudot-Lamotte, E.	N/A
brass bullet	铜质子弹
breakthrough	突破

英文	
Breughel, P. , the Elder	N/A
Brooks's Law	Brooks 法则
Brooks, F. P. Jr.	N/A
Brooks, K. P.	N/A
Brooks, N. G.	N/A
Buchanan, B.	N/A
Buchholz, W.	N/A
budget access size	预算 访问 规模
bug documented	N/A 文档化
Build-every-night approach	"每晚重建"方法
build, incremental system	增量式开发 系统
build-to-budget strategy	按预算开发的策略
build-up, manpower	内建，人力
building a program	构建程序
bullet, brass silver	铜质子弹 银弹
Burke, E.	N/A
Burke, A. W.	N/A
Bush, V.	N/A
Butler, S.	N/A
buy versus build	购买和自行开发
C++	C++语言
Cambridge Multiple-Access System	剑桥多重访问系统
Cambridge University	剑桥大学
Campbell, E.	N/A
Canova, A.	N/A
Capp, A.	N/A
Carnegie-Mellon University	卡内基－梅隆大学
CASE statement	CASE 语句
Case, R. P.	N/A
Cashman, T. J.	N/A

英文	中文
cathedral	大教堂
change summany	变更小结
change control of design organization	变更 变更控制 设计 组织机构
changeability	可变性
channel	通道
chemical engineering	化学工业
chief programmer	首席程序员
ClarisWorks	N/A
class	类
Clements, P. C.	N/A
clerk, program	程序职员
client-server system	客户机－服务器系统
Clingen, C. T.	N/A
COBOL	COBOL 语言
Codd, E. F.	N/A
Coding War Games	Coding War Gamesx 项目
coding	编码
Coggins, J. M.	N/A
Coleman, D.	N/A
command key	命令键
command	命令
comment	评论
committee	委员会
communication	交流、沟通
compatibility	兼容性
compile-time operation	编译操作
compiler	编译器
complexity arbitrary conceptual	复杂度 任意的、随意的 概念复杂度
component debugging	单元测试
component dummy	构件、组件 伪构（组）件
comprehensibility	理解程度
computer facility	计算机设施
conceptual construct	概念性结构要素
conceptual integrity	概念完整性

英文	
conceptual structure	概念结构
conference	大会
conformity	一致性
Conger, S. A.	N/A
Conklin, J.	N/A
control program	控制程序
convergence of debugging	调试的收敛性
Conway, M. E.	N/A
Conway, R. W.	N/A
Cooley, J. W.	N/A
copilot	副手
Coqui, H.	N/A
Corbato, F. J.	N/A
Cornell University	康奈尔大学
Cosgrove, J.	N/A
cost	成本
cost, development front-loaded	开发成本 先行投入
courge, managerial	管理勇气
court, for design disputes	仲裁设计分歧的会议
Cox, B. J.	N/A
Crabbe, G.	N/A
creation, component stages	构件阶段的创造
creative joy	创造的乐趣
creative style	创造性
creative work	创造性工作
creativity	创造力
critical-path schedule	关键路径进度
Crockwell, D.	N/A
Crowley, W. R.	N/A
cursor	光标
customizability	客户化
customization	定制
d'Orbais, J.	N/A
Dahl, O. J.	N/A
Daley, R. C.	N/A
data base	数据基础
data service	数据服务

英文	中文
database	数据库
datatype, abstract	抽象数据类型
date, estimated scheduled	估计日期 计划日期
debugging aid	调试辅助程序
debugging, component, high-level language, interactive, on-machine, sequential nature of, system	构件单元测试 高级语言 交互式 本机调试 次序特性 系统集成调试
DEC PDP-8	DEC PDP-8 型计算机
DEC VMS operating system	DEC VMS 操作系统
DECLARE	DECLARE 语句
Defense Science Board Task Force on Military Software	国防科学委员会军事软件工作组
Defense Science Board	国防科学委员会
DeMarco, T.	N/A
democracy	民主政治
Department of Defense	国防部
dependability of debugging vehicle	可靠的调试平台
description; *See* specification	描述，*参见*规格说明
design change	设计变更
design-for-change	为变更设计
designer, great	卓越的设计人员
desktop metaphor	桌面的类比
development, incremental	增量式开发
diagram	图
difference in judgement	观点差异
Digitek Corporation	Digitek 公司

英文	
Dijkstra, E. W.	N/A
director, technical, role of	技术主管的角色
discipline	学科、领域、规范
Disk Operation System, IBM 1410-7010	IBM 1410-7010 磁盘操作系统
display terminal	显示终端
division of labor	人力划分
DO...WHILE	DO……WHILE 语句
document	文档
documentation system	文档系统
documentation	文档
DOD-STD-2167	军标 DOD-STD-2167
DOD-STD-2167A	军标 DOD-STD-2167A
Dragon voice recognition system	Dragon 语音识别系统
Druffel, L.	N/A
dual ladder of advancement	两条职位晋升线
dummy component	伪构件
dump, memory	内存转储
Durfee, B.	N/A
ease of use	易用性
Eastman Kodak Company	Eastman Kodak 公司
Eckman, D.	N/A
editor, job description for text	编辑，职责描述文字
Einstein, A.	N/A
electronic mainl	电子邮件
electronic notebook	电子手册
Electronic Switching System	电子交换系统
encapsulation	封装
Engelbart, D. C.	N/A
English, W.	N/A
entropy	熵
environment	环境

英文	中文
Erikson, W. J.	N/A
Ershov, A. P.	N/A
Eschapasse, M.	N/A
essence	根本（困难）
estimating	估计
Evans, B. O.	N/A
Everett, R. R.	N/A
Excel	Excel 软件
expert system	专家系统
extension	扩展
Fagg, P.	N/A
Falkoff, A. D.	N/A
family, software product	软件产品族
Farr, L.	N/A
Fast Fourier Transform	快速傅立叶变换
featuritis	盲目的功能
Ferrell, J.	N/A
file, dummy miniature	伪文件缩影
filters	过滤器
Fjelstadt	N/A
floorspace	空间
flow arrow	箭头
flow chart	数据流图
forecast	预测
formal definition	形式化定义
formal progression of release	正式发布的进展
formality, of written proposals	书面建议的正式性
Fortran	Fortran 语言
Fortran, H.	N/A
FoxPro database	FoxPro 数据库
Franklin, B. (Poor Richard)	N/A
Franklin, J. W.	N/A
frequency data	频率数据
frequency guessing	频率猜测
fusion	融合

英文	
Galloping Gertie, Tacoma Narrows Bridge	塔科马大桥
Gantt Chart	甘特图
General Electric Company	通用电气公司
generator	发生器
gIBIS	gIBIS 系统
Ginzberg, M. G.	N/A
Glass, R. L.	N/A
Glegg, G. L.	N/A
Global Positioning System	全球定位系统
GO TO	GO TO 语句
God	上帝
Godel	N/A
Goethe, J. W. won	N/A
Gold, M. M.	N/A
Goldstine, H. H.	N/A
Gordon, P.	N/A
GOTO	GOTO 语句
Grafton, R. B.	N/A
Grant, E. E.	N/A
graph structure	图 结构图
graphical programming	图形化编程
great designer	卓越的设计人员
Green wald, I. D.	N/A
growing software	培育软件
Gruemberger, F.	N/A
Hamilton, F.	N/A
Hamlen	N/A
hardware, computer	计算机硬件
Hardy, H.	N/A
Harel, D. L.	N/A
Harr, J.	N/A
Hayes-Roth, R.	N/A
Heilmeyer, G.	N/A
Heinlein, R. A.	N/A

英文	中文
Hennessy, J.	N/A
Henricksen, J. O.	N/A
Henry, P.	N/A
Herzberg, F.	N/A
Hetzel, W. C.	N/A
Hezel, K. D.	N/A
hierarchical structure	层次化结构
high-level language, *See* language, high-level	high-level language, *参见* language, high-level
Hoare, C. A. R.	N/A
Homer	N/A
Hopkins, M.	N/A
Huang, W.	N/A
hustle	进取
Hypercard	Hypercard
hypertext	超文本
IBM 1401	IBM 1401 型计算机
IBM 650	IBM 650 型计算机
IBM 7030 Stretch computer	IBM 7030 Stretch 型计算机
IBM 704	IBM 704 型计算机
IBM 709	IBM 709 型计算机
IBM 7090	IBM 7090 型计算机
IBM Corporation	IBM 公司
IBM Harvest computer	IBM Harvest 型计算机
IBM MVS/370 operating system	IBM MVS/370 操作系统
IBM Operating System/360, *See* Operating System/360	IBM Operating System/360, *参见* Operating System/360
IBM OS-2 operating system	IBM OS-2 操作系统
IBM PC computer	IBM PC 计算机
IBM SAGE ANFSQ/7 data processing system	IBM SAGE ANFSQ/7 数据处理系统
IBM System/360 Model 165	IBM System/360 模型 165

英文	
IBM System/360 Model 30	IBM System/360 模型 30
IBM System/360 Model 65	IBM System/360 模型 65
IBM System/360 Model 75	IBM System/360 模型 75
IBM System/360 Principles of Operation	*IBM System/360 Principles of Operation*
IBM VM/360 operating system	IBM VM/360 操作系统
IBSYS operating system for the 7090	7090 的 IBSYS 操作系统
Ichikawa, T.	N/A
icon	图标
ideas, as stage of creation	构思，作为创造的阶段
IEEE *Computer* magazine	IEEE *计算机*杂志
IF...THEN...ELSE	IF...THEN...ELSE 语句
Ihm, C. C.	N/A
implementation	设计实现
implementations, multiple	多重实现
implementer	实现者
incorporation, direct	直接整合
incremental development	增量式开发
incremental-build model	增量开发模型
indenting	缩进
information hiding	信息隐藏
information theory	信息理论
inheritance	继承
initialization	初始化
input range	输入范围
input-output format	输入－输出格式
instrumentation	配备
integrity, conceptual	概念完整性

英文	***中文***
interaction, as part of creation first of session	交互，作为创造的一部分 第一次会话
interactive debugging	交互式调试
interactive programming	交互式编程
interface metaprogramming module WIMP	接口、界面 元编程 模块 WIMP
Interlisp	Interlisp 语言
International Computers Limited	国际计算机有限公司
Internet	Internet
interpreter, for space-saving	解释程序，用于节省空间
invisibility	不可见性
iteration	迭代
Iverson, K. E.	N/A
Jacopini, A.	N/A
Jobs, S.	N/A
Jones, C.	N/A
joys of the craft	行业的乐趣
Kane, M.	N/A
keyboard	键盘
Keys, W. J.	N/A
King, W. R.	N/A
Knight, C. R.	N/A
Knuth, D. E.	N/A
Kugler, H. J.	N/A
label	标签
Lachover, H.	N/A
Lake, C.	N/A
Landy, D. E.	N/A
language description, formal	形式化语言描述
language translator	语言翻译程序

英文	
language, fourth-generation high-level machine programming scripting	第四代语言 高级 机器 编程 脚本
late project	进度落后的项目
lawyer, language	语言专家
Lehman, M.	N/A
Lewis, C. S.	N/A
library class macro program	库 类库 宏库 程序库
linkage editor	链接编辑程序
Lister, T.	N/A
Little, A. D.	N/A
Locken, O. S.	N/A
Lowry, E. S.	N/A
Lucas, P.	N/A
Lukasik, S.	N/A
Macintosh WIMP interface	Macintosh WIMP 界面
Madnick, S.	N/A
magic	魔术
maintenance	维护
man-month	人月
management information system (MIS)	管理信息系统
manual System/360	手册 360 系统手册
market, mass	大众市场
matrix management	矩阵管理
matrix-type organization	矩阵类型的组织结构
Mausner, B.	N/A
Mayer, D. B.	N/A
McCarthy, J.	N/A
McCracken, D. D.	N/A
McDonough, E.	N/A

英文	中文
Mealy, G.	N/A
measurement	测量、度量
medium of creation, tractable	容易驾驭的创造媒介
meeting, problem action status review	问题－行动会议 状态检查会议
memory use pattern	内存使用方式
mentor	导师
menu	菜单
Merwin, R. E.	N/A
Merwin-Dagget, M.	N/A
metaphor	类比、象征
metaprogramming	元编程
microcomputer revolution	微型计算机革命
microfiche	微缩胶皮
Microsoft Corporation	微软公司
Microsoft Windows	Microsoft Windows
Microsoft Word 6.0	Microsoft Word 6.0
Microsoft Works	Microsoft Works
milestone	里程碑
Mills, H. D.	N/A
MILSPEC documentation	MILSPEC 军用标准
mini-decision	细小（迷你）决定
MiniCad design program	MiniCad 设计软件
MIT	麻省理工学院
mnemonic name	助记名
model COCOMO incremental-build spiral waterfall	模型 COCOMO 增量开发 螺旋 瀑布
Modula	Modula
modularity	模块化
module	模块
modules, number of	模块数量
Mooers, C. N.	N/A

英文	
Moore, S. E.	N/A
Morin, L. H.	N/A
Mostow, J.	N/A
mouse	鼠标
moving projects	项目迁移
Mozart, W. A.	N/A
MS-DOS	MS-DOS
Multics	Multics 系统
multiple implementations	多重实现
MVS/370	MVS/370
Nammed, A.	N/A
Nanus, B.	N/A
Naur, P.	N/A
Needham, R. M.	N/A
Nelson, E. A.	N/A
nesting, as documentation aid	嵌套，作为文档辅助
network nature of communication	交流的网络特性
Neustadt, R. E.	N/A
Newell, A.	N/A
Noah	N/A
North Carolina State University	北卡罗来纳大学
notebook, status system	状态记录 系统
Nygaard	N/A
object	对象
object-oriented design	面向对象设计
object-oriented programming	面向对象编程
objective cost and performance space and time	目标 成本和性能 空间和时间
obsolescence	陈旧过时
off-the-shelf package	现货成品包
office space	办公室空间
Ogdin, J. L.	N/A

英文	*中文*
open system	开放系统
operating system	操作系统
Operating System/360	操作系统/360
optimism	乐观主义
option	选项
Orbais, J. d'	N/A
order-of-magnitude improvement	数量级的提高
organization chart	组织结构图
organization	组织结构
OS/360 Concepts and Facilities	*OS/360 概念和设施*
OS/360 Queued Telecommunications Access Method	OS/360 队列远程通讯访问方法
OS/360, *See* Operating System/360	OS/360, *参见* Operating System/360
overlay	覆盖
overview	总览
Ovid	N/A
Padegs, A.	N/A
paperwork	文书工作
Parnas families	N/A
Parnas, D. L.	N/A
partitioning	分区
Pascal programming language	Pascal 语言
Pascal, B.	N/A
pass structure	数据流
Patrick, R. L.	N/A
Patterson, D.	N/A
people	人
Peopleware: Productive Projects and Teams	*人件：高生产率的项目和团队*
perfection, requirement for	要求完美
performance simulator	性能仿真装置
performance	性能
PERT chart	PERT 图

英文	
pessimism	悲观注意
Peter and Apostle	N/A
philosopher's stone	点金石
Piestrasanta, A. M.	N/A
pilot plant	试验工厂
pilot system	试验系统
pipes	管道
Pisano, A.	N/A
Pius XI	N/A
PL/C language	PL/C 语言
PL/I	PL/I 语言
planning	策划、计划
Plans and Controls organization	计划和控制机构
playpen	开发库
Pnueli, A.	N/A
pointing	指针选取
policed system	具有错误控制的系统
Politi, M.	N/A
Pomeroy, J. W.	N/A
Poor Richard (Benjamin Franklin)	N/A
Pope, Alexander	N/A
Portman, C.	N/A
power tools for the mind	创造性活动的强大工具
power, giving up	放弃权力
practice, good software engineering	软件工程的良好实践
price	价格
PROCEDURE	PROCEDURE 语句
procedure, catalogued	具有目录的过程库
producer, role of	制片人的角色
product test	产品测试
product, exciting programming system programming	激动人心的产品 编程系统产品 编程产品
productivity equation	生产率公式
productivity, programming	软件开发的生产率
program clerk	程序职员

英文	*中文*
program library	程序库
program maintenance	程序维护
program name	程序名称
program products	编程产品
program structure graph	程序结构图
program auxiliary self-documenting	程序 辅助程序 自文档化程序
programmer retraining	程序员的再培训
programming environment	编程环境
programming language	编程语言
programming product	编程产品
programming system	编程系统
programming systems product	编程系统产品
programming system project	编程系统项目
programming, automatic graphical visual	编程自动化 图形化 可视化
progressive refinement	渐进地精化
Project Mercury Real-Time System	Mercury 实时系统项目
project workbook	项目工作手册
promotion, in rank	按级晋升，
prototyping, rapid	快速原型
Publilius	N/A
purple-wire technique	紫色线束的手法
purpose, of a program of a variable	程序的目标 变量的目的
Quadragesimo Anno, Encyclical	*教皇通谕 Quadragesimo Anno*
quality	质量

英文	
quantization, of change of demand for change	变更量子化(阶段化) 变更要求的量子化（阶段化）
Raeder, G.	N/A
raise in salary	涨薪
Ralston, A.	N/A
rapid prototyping	快速原型化
real-time system	实时系统
realism	现实主义
realization, step in creation	物理实现，创造过程的一个步骤
refinement, progressive requirements	渐进地精化 需求
regenerative schedule disaster	再现的进度灾难
Reims Cathedral	兰斯大教堂
release, program	程序发布
reliability	可靠性
remote job entry	远程任务项
repartitioning	重新分配
representation, of information	信息的表现形式
requirements refinement	需求精化
rescheduling	重新计划进度
responsibility, versus authority	责任和权威
Restaurant Antoine	Antoine 餐厅
reusable component	可重用的构件
reuse	重用
Reynolds, C. H.	N/A
role conflict, reducing	减少角色冲突
ROM, read-only memory	ROM，只读存储器
Roosevelt, F. D.	N/A
Rosen, S.	N/A
Royce, W. W.	N/A
Rustin, R.	N/A
Ruth, G. H.(Babe)	N/A

英文	中文
Sackman, H.	N/A
Salieri, A.	N/A
Saltzer, J. H.	N/A
Sayderman, B. B.	N/A
Sayers, D. L.	N/A
scaffolding	脚手架（测试平台）
scaling up	扩建、增大规模
Scalzi, C. A.	N/A
schedule, *See* Late project cost-optimum	进度，参见 Late project 最优进度
scheduler	调度程序
scheduling	计划安排、进度安排
Schumacher, E. F.	N/A
screen	屏幕
second-system effect	开发第二个系统的后果（画蛇添足）
secretary	秘书
security	安全性
self-documenting program	自文档化程序
Selin, I.	N/A
semantics	语义
Shakespeare, W.	N/A
Shannon, E. C.	N/A
Share 709 Operating System (SOS)	Share 709 操作系统（SOS）
Share Operating System for IBM	IBM Share 操作系统
Shell, D. L.	N/A
Sherman, M.	N/A
Sherman, R.	N/A
short cuts	快捷键
shrink-wrapped software	塑料薄膜包装的成品软件
Shtul-Trauring, A.	N/A
side effect	副作用
silver bullet	银弹
simplicity	简洁
Simula-67	Simula-67 语言

英文	
simulator environment logic performance	仿真装置 环境 逻辑 性能
size, program	程序规模
Skwiersky, B. M.	N/A
slippage, schedule, *See* Late project	slippage, schedule, *参见* Late project
Sloane, J. C.	N/A
Small is Beautiful	*小就是美*
Smalltalk	Smalltalk 语言
Smith, S.	N/A
snapshot	快照
Snyder, Van	N/A
sociological barrier	社会障碍
Sodahl, L.	N/A
Software Engineering Economics	软件工程经济学
Software Engineering Institute	软件工程研究所
software industry	软件工业
Software Project Dynamics	*软件项目动力学*
Sophocles	N/A
space allocation	空间分配
space, memory office program, *See* Size, program	内存空间 办公室空间 程序空间，*参见* 程序规模
specialization of function	限定职责范围
specification architectural functional interface internal performance testing the	规格说明 体系结构 功能 接口 内部 性能 测试规格说明
speed, program	程序运行速度
spiral, pricing-forecasting	价格预测螺旋
spreadsheet	电子表格

英文	中文
staff group	团队
staffing, project	项目人员配备
Stalnaker, A. W.	N/A
standard	标准
standard, de facto	事实标准
Stanford Research Institute	斯坦福研究机构
Stanton, N.	N/A
start-up firm	新兴公司
Statemate design tool	状态机设计工具
status control	状态控制
status report	状态报告
status review meeting	状态检查会议
status symbol	状态特征
Steel, T. B., Jr.	N/A
Strachey, C.	N/A
straightforwardness	直白
Strategic Defense Initiative	防御系统
Stretch Operating System	Stretch 操作系统
stub	占位符
Stutzke, R. D.	N/A
subroutine	子程序
Subsidiary Function, Principle of	附属职能行使原理
superior-subordinate relationship	上下级的关系
support cost	支持成本
surgical team	外科手术队伍
Sussenguth, E. H.	N/A
Swift, J..	N/A
synchronism in file	文件同步
syntax abstract	语法 抽象
system build	系统构建
system debugging	系统集成调试
System Development Corporation	System Development Corporation（公司）
system integration sublibrary	系统集成子库
system test	系统测试

英文	
system, large programming	大型系统 编程
System/360 computer family	System/360 计算机家族
systems product, programming	系统编程产品
Tacoma Narrows Bridge	塔科马大桥
Taliaffero, W. M.	N/A
target machine	目标机器
Taub, A. H.	N/A
Taylor, B.	N/A
team, small, sharp	小型、精干的队伍
technical director, role of	技术主管的角色
technology, programming	编程技能
telephone log	电话日志
test case	测试用例
test, component system	构件单元测试 系统集成测试
tester	测试人员
testing advisor	测试顾问系统
testing regression specification	测试 回归测试 规格说明的测试
TESTRAN debugging facility	TESTRAN 调试软件
text-editing system	文本编辑系统
Thompson, K.	N/A
throw-away	抛弃
time, calendar machine	日期时间 机器时间
Time-Sharing System, PDP-10	PDP－10 分时系统
Time-Sharing System/360	分时 System/360
time-sharing	分时
tool power, for the mind	工具 创造性活动的强大工具

英文	中文
toolsmith	工具维护人员
top-down design	从上至下设计
top-down programming	从上至下编程
Tower of Babel	巴比伦塔
TRAC language	TRAC 语言
tracing, program	程序跟踪
trade-off, size-function size-speed	规模－功能折衷 规模－速度折衷
training, time for	培训时间
transient area	临时性空间
Trapnell, F. M.	N/A
tree organization	树型组织机构
Truman, H. S.	N/A
TRW, Inc.	TRW 公司
Tukey, J. W.	N/A
turnaround time	周转时间
turnover, personnel	演变
Turski, W. M.	N/A
two-cursor problem	双光标问题
two-handed operation	双手操作
type, abstract data	抽象数据类型
type-checking	类型检查
Univac computer	Univac 计算机
Unix workstation	Unix 工作站
Unix	Unix
University of North Carolina at Chapel Hill	位于查布尔希尔的北卡来罗来纳大学
user novice power	用户 新手 专业用户
USSR Academy of Sciences	苏联科学院
utility program	实用程序
Vanilla Framework design methodology	香草框架设计方法学
vehicle machine	辅助机器
verification	验证

英文	中文
version alpha beta	版本 alpha 版本 beta 版本
Vessey	N/A
virtual environment	虚拟环境
virtual memory	虚拟内存
visual programming	虚拟编程
visual representation	虚拟表达
vocabularies, large	大量的词汇
Voice Navigator voice recognition system	Voice Navigator 语音识别系统
von Neumann, J.	N/A
Vyssotsky, V. A.	N/A
Walk, K.	N/A
Walter, A. B.	N/A
Ward, F.	N/A
waterfall model	瀑布模型
Watson, T. J., Jr.	N/A
Watson, T. J., Sr.	N/A
Weinberg, G. M.	N/A
Weinwurm, G. F.	N/A
Weiss	N/A
Wells Apocalypse，The	*韦尔斯启示录*
werewolf	人狼
Wheller, E.	N/A

英文	
William III of England, Prince of Orange	N/A
Wilson, T. A.	N/A
WIMP interface	WIMP 界面
window	窗口
windows NT operating system	windows NT 操作系统
Windows operating system	Windows 操作系统
Wirth, N.	N/A
Witterrongel, D. M.	N/A
Wizard-of-Oz, technique	向导技术
Wolverton, R. W.	N/A
workbook	工作手册
workstation	工作站
World-Wide Web	万维网
wormhole	后门接口
Wright, W. V.	N/A
Xerox Palo Alto research Center	Xerox Palo Alto 研究中心
Yourdon, E.	N/A
zoom	推进
Zraket, C. A.	N/A

封底文字

Frederick P. Brooks，Jr.荣获了计算机领域最具声望的图灵奖（A.M.Turing Award）桂冠。美国计算机协会（ACM）称赞他“对计算机体系结构、操作系统和软件工程作出了里程碑式的贡献。”（*landmark contributions to computer architecture, operating system, and software engineering.*）

内容简介

软件项目管理领域很少能有著作能像《*人月神话*》一样具有影响力和畅销不衰。Brooks为任何人管理复杂项目提供了颇具洞察力的见解，既有很多发人深醒的观点，也有大量的软件工程现实。这些论文出自 Brooks 的 IBM System/360 家族和 OS/360 项目管理经验。在第一次出版 20 年后的今天，Brooks 重新审视了他原先的观点，增加了一些新的想法和建议。这既是为了熟悉原有内容的人们，也为了第一次阅读它的读者。

增加的章节包括（1）原著中的所提出观点的一些精华，包括 Brooks《*人月神话*》的核心论点：由于人力的划分，大型项目遭遇的管理问题与小型项目的不同；因此，产品的概念完整性很关键；取得这种统一性是很困难，但并不是不可能的。（2）一个时代以后，Brooks对这些观点的看法。（3）重新收录了 1986 年的经典文章《*没有银弹*》。（4）以及对 1986 年所下论断——“在十年内不会出现银弹。”——现在的认识。