谨以本书献给为软件工程带来火花的研究者

和使火继续燃烧的实践者

译者序

本书抽取了软件行业中经典的55个基本事实和10个谬误进行分析，对于每个观点，作者都提供了观点的来源和证据以及相关的参考文献。除此之外，本书还介绍了业界对这个观点的其他不同认识和理解。

本书的内容和组织方式非常有意思、有特点，市面上绝无类似的书籍，因为一般只有专业论文才采取这样的方式陈述观点。作者的行业经验和性格决定了本书的价值。它可能会让你拍案惊奇，也可能会让你认为有点狂妄，从而进一步促使你思考这些观点，跟同事和同仁进行辩论和分析，而这恰恰就是作者希望达到的目的。

阅读本书总会有一些让你感慨万千的地方，我觉得市面上能够真正做到这点的书并不多。对于这些行业观点，我们很少收集和总结，这恰恰就是我们的缺陷。你可以在本书基础上进一步扩展和完善你认为有价值的事实和谬误，进而形成自己的经验库。经常翻阅它，提醒自己不要忘记那些最基本的行业规律，规避那些前人已经明确指出来的“陷阱”。

作者列举的事实和谬误，有的令我很惊诧，确实是这样吗？说实话，本书的一些观点确实对我触动很大，有助于我定位一些更新的研究目标。总之，我喜欢这本书，喜欢作者的坦率。

原书的句子比较晦涩，涉及的领域知识比较广泛，我们竭力保证译文的正确、通顺和优雅，但错误之处在所难免，恳请广大读者多加指正。

本书主要由严亚军和龚波翻译，由龚波统稿。李红玲、马丽、李志、邓波等在本书的翻译过程中都提供了很多指导性意见和建议，在此表示感谢！同时还要感谢高军老师的信任，以及编辑老师的辛勤工作！

龚波

2005年

致　谢

感谢 Paul Becker。

他目前在 Addison-Wesley 工作，几乎编辑过我参与编写的所有公开出版物，感谢他多年来对我的信任。

感谢 Karl Wiegers。

感谢他经常帮助我回忆那些容易被遗忘的基本事实，同时感谢他花费大量精力来审阅和修订本书。

感谢 James Bach、Vic Basili、Dave Card、Al Davis、Tom DeMarco、Yaacov Fenster、Shari Lawrence Pfleeger、Dennis Taylor 和 Scott Woodfield。

感谢他们帮助我确定本书很多事实和谬误的来源和引用之处。

序

初次听到 Bob Glass 准备参照我的《201 Principles of Software Development》来写这本书时，我有一点担心。毕竟，Bob 是该行业中最优秀的作者之一，他的书将对我的书形成激烈竞争。当 Bob 邀请我写本书的序时，我更加担心了，我怎么可以为一本与自己的书直接竞争的书作序呢？等读完这本书后，我对于有机会为本书作序感到欣喜和荣幸（而不再担心）。

目前的软件行业与 19 世纪晚期的制药行业处于相同的阶段。似乎在我们当中，蛇油销售员和预言者比精明的参与者和讲道理者多。我们每天都会听说有人新发现了治疗某种不治之症的药物。同样，我们常常听到某些快速方法可以解决低效率、低质量、客户不满意、沟通不畅、需求变更、无效测试以及糟糕的管理等问题。不负责任的“博学者”实在是太多了，以至于我们有时候会怀疑那些所谓的万能药。我们该去问谁？在这个行业中我们该信任谁？我们在哪里能找到真理？答案是 Bob Glass。

多年来，Bob 曾经为我们提供了许多有关软件灾难的短篇论文。我一直期待着他讨论这些灾难的共性，以便我们可以从他的经验中受益。在现在这本精彩的书中，Bob Glass 讨论的 55 个事实不仅仅是个人的推想和认识。这些事实正是我所期待的：作者通过详细查阅自己过去发表的文章中的数百个案例，从中精炼出这些精妙的结论。

可能并非所有的读者都喜欢后面的这 55 个事实，其中有些事实与某些所谓的现代方法截然相反。对于忽略这些建议的那些人，我只能祝你旅途愉快，但是我真替你们担忧。你们正走向一条前途险恶的路，其中荆棘满布，许多人会身陷其中，不得不结束自己的职业生涯，我建议你最好去读 Bob 有关软件灾难的早期书籍。对于那些接受这些建议的人，你们正走向另一条前途光明的路，在这条路上充满了成功的实例，这是一条富有知性和知识的道路。应该信任 Bob Glass，因为这是他曾经的经历。他有权在分析他人的数百个成功或失败案例的同时，分析自己的成功和失败。站在他的肩上，你在该行业的成功机率就更大。忽视他的建议，

那你就等着 Bob 在多年之后再打电话询问你的项目——他会将此收入下一版的软件灾难故事之中。

Alan M. Davis
2002 年春

作者附言：

我尽力劝 Al Davis 低调写这个序，毕竟这有点夸张。但是他拒绝了我所有的劝告（我确实努力劝过他）。实际上，在一次交流中，他说："你值得受尊重，我来帮助你！"我的经验告诉我：捧得越高，摔得越狠。

无论如何，我都想像不出比 Al 所送给我的更宽厚、更精彩的评价。多谢！

Robert L. Glass
2002 年夏

目　录

第 2 部分　5+5 谬误

第 1 部分

55 个事实

简　介

本书包含很多有关软件工程的事实与谬误。

听起来很无聊，难道不是吗？一个有关软件构建方面事实和谬误的列表听起来可不像足球那样讨人喜欢，不值得花费几个小时。但是，这些事实和谬误有一些特别之处，那就是它们是基础性的，而更重要的是它们经常被人遗忘。实际上，这就是本书的中心主题。关于构建软件，我们理应知道许多东西，但是实际上却因为这样或那样的原因不知道，甚至我们一些想当然的看法实际上是显而易见的错误。

上一段中的我们指的是谁？当然是指构建软件的人。我们似乎需要反复多次接受同样的教训，但如果我们能牢记这些事实，就可以避免那些教训。同时，这里的我们还指那些从事软件研究的人。有些研究人员太局限于理论方面，以至于忽视了一些重要的基本事实，而这些事实甚至可以推倒他们自己创建的理论。

因此，本书读者包括对构建软件感兴趣的所有人，具体指软件专业人士（包括技术人员和管理人员）、学生、教师和研究人员。我认为，不谦虚地说，本书适合所有从事计算机行业的人。

本书最初的书名很冗长，含 13 个单词：Fifty-Five Frequently Forgotten Fundament Facts（and a Few Fallacies）about Software Engineering［有关软件工程的 55 个容易忘记的基本事实（和部分谬误）］。这个名字相当繁琐，至少本书的策划者认为繁琐。当今，很多书籍都使用非常“酷”的书名来增加卖点。出版商和我最后商定采用 Facts and Fallacies of Software Engineering（软件工程的事实与谬误）。这个名字简洁、清晰，又非常朴素。

我曾尝试过将最初的长书名昵称为 F-Book，仅仅是为了与书名中所有以 F 开头的单词压韵。但是出版商反对这样做，我想他是对的。F 是字母表中的“下等”字母（有人倡导使用 H 和 D，但是很少有人推崇使用 F），所以就没有叫 F-Book（我早期曾经写过一本有关构造编译器的书，仅仅因为自己随意在封面放了一个龙的图片，就称之为 Dragon Book。这次，我没有沿袭那种习惯）。

但是我想做如下辩解：这些以 F 开头的单词都是有意义的，对于理解书名的含义很重要。当然，55 只是一个数字，我提供 55 个事实，目的是使书名更押韵（我打赌，在 Alan Davis 那本优秀著作中，也是先确定了数字 201，然后才寻找软件工程中相应的 201 个原则的）。但是其他以 F 开头的单词都是精挑细选的。

frequently forgotten（经常被忘记）？因为其中许多事实确实经常被忘记。读本书时有读者可能会说："哦，我想起来了。"你应该反思为什么这么多年都忘记了它。

fundamental（基本的）？选择所介绍事实的首要原则是它们在软件领域中至关重要。我们可能忘记了其中许多事实，但这不是说它们不重要。如果你仍拿不准是否继续读本书，那么我给你一个理由，我坚信：在这组事实中，你会发现软件工程领域中最基本的重要知识。

facts（事实）？奇怪，这可能是书名中最具争议的单词。对于我所选的事实，你可能并非完全同意。你甚至会强烈反对其中的一些。我个人看来它们都是事实，但是你也可以不这样看。

a few fallacies（一些谬误）？在软件领域，有许多"神圣"的歪论，我忍不住要揪出来。我必须承认，自己所谓的谬误在别人看来可能是事实。但是在读这本书的过程中，你应该对于我所谓的事实和谬误形成自己的观点。

这些事实和谬误是否已经过时？本书的一位审校者认为有些内容过时了。他的指责让我羞愧。但是，那些经常被忘记的事实和谬误应当持续一段时间。在这个组合中有许多闪闪发光的经典之物。但是，我相信你在其中一定会找到一些惊奇的新观点，因为你并不熟悉它们。这些事实和谬误的关键不在于它们是否陈旧，而是对于你来说是否是新的。

在本书的这一部分，我将简要介绍这些事实。关于谬误将在本书后一部分中介绍。在这一部分，我们将浏览这 55 个经常被忘记的基本事实，看其中有多少与这些以 F 开头的单词有关。客观地说，其中有一些事实并没有被忘记。

- 其中的 12 个事实鲜为人知。许多人从未听说过它们，自然无所谓忘记，但是我敢断言，它们都绝对重要。
- 其中的 11 个事实被广泛接受，但是似乎没有人按其行事。
- 其中的 8 个事实也被接受，但是我们不知道如何（或者是否）确定它们所反映的问题。
- 其中的 6 个事实可能被绝大多数人安全接受，没有争议，也很少被忘记。

- 其中的 5 个事实被许多人旗帜鲜明地反对。
- 其中的 5 个事实被许多人接受，但是有一些人强烈反对，所以相当有争议。

这些加起来不等于 55，因为（a）有些事实适合多个种类；（b）还有一些事实属于其他的类别，例如“只有供应商会反对”。我不具体告诉你哪些事实归于哪一类，我想你会自己做出判断。

你将看到本书中有大量的争议。为了帮助你理解，我在每个事实的讨论之后列举了相关的争议。我希望无论你我的观点是否吻合，这样做都能包含你的观点，你可以找出相似之处。

即使有这么多的争议，我还是明智地告诉你：我自己对于选择这些事实和谬误非常自信。（在本书的后面有一个幽默的自传，因此这里会讲快一点。）我作为技术实践者和研究者，已经在软件工程领域奋斗了 45 年。在这一领域，我已经写了 25 本书和超过 75 篇专业论文。我在 3 个最主要的期刊上有定期的专栏文章，分别是：在《Communications of the ACM》的 Practical Programmer、在《IEEE Software》的 Loyal Opposition 和在 ACM SIGMIS 的《DATABASE》的 Through a Glass, Darkly。我知道自己是一个批判者，很荣幸地被称为“最爱发脾气的老家伙”就证明了这一点。你能期待我质疑那些无可挑剔的东西，正如我前面所说的，要揪出“神圣”的歪论。

对于这些事实，我还有一点要说。我已经说过这些事实都经过仔细筛选，以确保它们都是本领域中的基本问题。但是，它们当中到底多少事实被忘记了，从这一点可以反映出我们没有利用的知识量。项目管理者会说其中的许多事实已经被忘记或者从来没听说过。软件开发者因缺乏对这些事实与谬误知识的了解而工作于受限的领域，研究者们认为如果把这些事实与谬误考虑进来，所实现的东西将是荒诞可笑的。我深信那些决定继续读下去的人必须有丰富的学习经验或者极好的记忆力。

现在，在开始讨论这些事实之前，我想提出几点重要的希望。我在阐述这些事实的同时，还提出了该领域的一些问题。我并不想在此给出这些问题的答案。这本书只讨论是什么，而不讨论怎么办。我认为有一点很重要：我只想将这些事实公之于众，这样大家就可以自由讨论并付诸行动。我认为这个目标已经相当重要，没有必要讨论解决方法，否则会使主题模糊。这些事实所带来的问题的解决方法通常见于已出版的专业性书籍和论文中：软件工程教科书、软件工程专题书、

主要软件工程学术期刊和受欢迎的学术杂志（虽然其中许多内容良莠不齐、鱼龙混杂）。

为了便于你阅读，我按照下面的结构来陈述这些事实：

- 首先讨论一个事实。
- 然后提出围绕这一事实的争议。
- 最后提出有关这一事实的信息来源，以及有关背景和前景信息的参考文献。根据软件工程的标准（它们也是经常被忘记的事实），有些文献年代较为久远，有些非常新颖，有些则兼而有之。

我将这 55 个事实分为几类，分别是：

- 管理
- 生命周期
- 质量
- 研究

我将谬误也进行类似的分类，分别是：

- 管理
- 生命周期
- 教育

好了，我们已经做好足够的准备，希望你会喜欢我所提供的这些事实和谬误。更重要的是，我希望它们会对你有用。

Robert L. Glass

2002 年夏

第1章

管　理

说实话，我一直认为管理是很烦人的话题。在我读过的有关管理的书籍中，95%都是常识，其余5%是重温陈词滥调。那么在本书中，我为何要首先讨论管理这一话题？

这是因为平心而论，在软件领域中的许多高层次、显而易见的问题与管理有关。例如，我们将很多失败都归因于管理，而许多成功也可以归因于管理。1995年 Al Davis 在有关软件原则的一本精彩著作《201 Principles of Software Development》的第127条原则中说到："好的管理比好的技术更重要。"虽然我不愿意承认，但是 Al 是对的。

为什么我不愿意承认？在我早期的职业生涯中，面临着不可回避的选择：我可以继续当一个技术人员，继续做我喜欢做的软件开发，我还可以选择另一条道路——成为一个管理者。我认为这个选择非常难。美国的软件业成功之路需要改善产业结构，如果没有优秀的管理方法会很难。但是后来有两个原因使我意识到自己不会把技术抛在脑后。

1. 我愿意实干，而不愿意指挥别人去做。
2. 我喜欢自己做出决定，而不愿意成为夹在中间的管理人员，传达上司的决定。

你也许觉得后一个原因很奇怪。技术人员怎么可能比自己的管理者拥有更大的自由，我的经验告诉我确实如此，但是这对别人很难解释得通。关于这个问题我后来写了一本书，即《The Power of Peonage》（即《雇佣者的力量》，1979）。那本书的主旨——也是让我继续做技术人员的信念——是处于管理最底层且专长于干活的人通常比其他任何人有更大的威力。他们不会降级，通常不会有比他们更低的职位。也许有别的方式可以惩罚一个优秀的技术人员，但是绝不会是降级处理。在我的技术生涯中，我不止一次使用这种威力。

对不起，我跑题了。我们的话题是为什么我这样一个执着的技术人员选择管理作为本书的开头。我在此想说的是技术人员比管理人员更快活。我并没有说技术人员比管理人员更重要。实际上，在经常被忘记的软件事实中，至关重要的通常是管理方面的问题。不幸的是，管理者通常深陷于各种常识和陈腐的建议中，以至于他们忽略了一些非常独特、至关重要、应该时刻牢记的事实。

诸如关于人员。人员有多么重要。一些人如何比另一些人优秀许多。为什么说项目成败的关键是由谁来做，而不是如何做。

诸如关于工具和技术（这些通常由管理者选定）。对于这些东西的鼓吹如何会弊大于利。采用新方法，生产效率为何会先下降，后上升。为什么新技术、新工具很少真正投入使用。

诸如关于估算。我们的估算通常是何等的差。获得估算数据的过程是何等的糟糕。我们如何将这些糟糕的估算与许多重要的项目失败关联起来。管理者和技术人员之间在估算问题上为何会有“隔阂”。

诸如关于复用。我们采用复用的时间有多久。近几年来，复用的进展程度有多大。人们对于复用的（可能是错误的）期望有多大。

诸如关于复杂性。构建软件的复杂性说明了该领域中的许多问题。复杂性提高得有多快。聪明人如何克服这种复杂性。

以上就是本章的内容概要。下面，我们将深入讨论这些在管理方面应该熟记却经常遗忘的话题。

参考文献

- Davis, Alan M. 1995. *201 Principles of Software Development*. New York: McGraw-Hill.
- Glass, Robert L. 1979. *The Power of Peonage*. Computing Trends.

1.1 人员

事实 1	在软件开发中，最重要的因素不是程序员采用的工具和技术，而是程序员自身的质量。

讨论

在构建软件的过程中，人员因素起直观作用。这是事实 1 的中心思想。工具很重要，技术很重要，过程当然也很重要，但是人员的作用远远超过其他因素。

这一中心思想与软件行业一样古老。多年来，这一思想显现在许多软件研究论文和报告中，以至于成为软件领域中最重要的“永恒真理”之一。然而，我们这些软件人员却一直将它抛之脑后。我们提倡：在软件开发过程中，过程管理应该始终贯穿其中。我们提倡将工具作为构建软件的突破点。我们积累了多种技术，并且让成千上万的程序员阅读技术资料、学习技术课程，还通过训练和实践将他们的鼻子牵到技术上，然后在高层面的项目实施中采用这些技术。这简直就是凌驾于人员之上的工具/技术/过程。

有时候，我们甚至反过来采取反对人员的方法。我们将人员看作装配线上可以随意更换的小螺丝。我们认为如果时间表紧张、限制严格，那么人员就可以工作得更出色。我们甚至否定了程序员最基本的信念，却要他们干出成绩来证明自己。

在这一方面，了解 SEI（Software Engineering Institute，软件工程研究所）及其 CMM（Capability Maturity Model，能力成熟度模型）非常有意义。CMM 假设良好的过程会得出良好的软件。CMM 制定了许多关键过程域（key process area），并设计了一套改进步骤，各软件机构依照该步骤改进软件生产过程，这一切都基于上述假设。关于 CMM 非常有意思的是：CMM 已经存在多年，美国国防部老早就设立了专门机构来研究改进软件组织的方法，大家也纷纷效仿；然而，多年以后 SEI 开始考虑人员以及人员在软件构建过程中的重要性。现在，SEI 又提出个人能力成熟度模型（People Capability Maturity Model）。但是相对于过程 CMM 而言，P-CMM 声名却很小。许多软件人员反复认为过程重于人员，甚至在有些情况下重要得多。看来，我们永远学不到真谛。

争议

有关人员重要性的争议非常微妙。每个人都口头上承认人很重要。几乎所有人在表面上都承认人员胜过工具、技术和过程。然而，我们的行为却一直否认这一点。也许是因为与工具、技术和过程的因素相比，人员更难引起注意。也许这有点象“小傻子”笑话（有一个有 60 多年历史的笑话，一个小傻子在一个灯柱

下找东西。有人问他干什么，他回答说在找自己的钥匙。问他“钥匙丢在哪儿”，他回答到“在那边”，别人又问“那你为什么不去那边找”，他回答说“因为这边亮”）。

我们都是软件人员，我们都知道，在更深的层次上人的因素更重要。但是所有的技术人员都真心希望能发明新的技术来简化工作。

来源

有关人员重要性的最明显的表述是在经典的《Software Engineering Economics》（即《软件工程经济学》，Barry Boehm 著，1981）一书的封面上。在此，作者画了一个柱状图，用来描述对软件开发有贡献的各种因素。你会注意到，最长的柱子就是人员的质量。这个柱状图告诉我们：人员比工具、技术、语言都重要，当然也比人员所采用的过程重要。

关于这一点最重要的表述来源于同样经典的《Peopleware》（即《人件》，DeMarco 和 Lister 著，1999）。你可以根据书名猜到全书阐述了软件领域中人力资源的重要性。其中说到：我们工作中最重要的问题与其说是技术问题，还不如说它本质上是社会问题。并进一步说到：一开始就着眼技术问题是“高技术幻想”。看这本书，你必然会明白：在软件领域，人员的作用远远大于其他任何因素。

对此最简洁的表述见于 Davis 的书中：“人员是成功的关键”（1995）。最近的表述来自于敏捷制造运动（Highsmith 2002）：“揭开各种工程方法的表面，寻找成功的原因，答案是人员”。关于人员重要性的最早表述有：Bucher（1975）的“影响软件可靠性的首要因素是寻找、激励和管理软件的设计和维护人员”；Rubey（1978）的“说到底，决定软件生产效率的最终因素是每个参与者的能力”。

但是，我最欣赏的竟然是出自某重要期刊上的一篇晦涩文章。文中讲到：“假如你的生命依靠某一软件，那么，关于该软件你想知道什么？”Bollinger（2001）回答到：“与其他事情相比，我最想知道写该软件的人，这人一定才华横溢，严谨认真，狂热追求软件完美并按照需求运作。其他对我来说都是次要的。”

参考文献

- Boehm, Barry. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall.
- Bollinger, Terry. 2001. “On Inspection vs. Testing.” *Software Practitioner*,

Sept.

- Bucher, D.E.W. 1975. "Maintenance of the Computer Sciences Teleprocessing System." Proceedings of the International Conference on Reliable Software, Seattle, WA, April.
- Davis, Alan M. 1995. *201 Principles of Software Development*. New York: McGraw-Hill.
- DeMarco, Tom, and Timothy Lister.1999. *Peopleware*. 2d ed. New York: Dorset House.
- Highsmith, James A. 2002. *Agile Software Development Ecosystems*. Boston: Addison-Wesley.
- Rubey, Raymond L. 1978. "Higher Order Languages for Avionics Software—A Survey, Summary, and Critique." Proceedings of NAECON.

事实 2	对"个体差异"的研究表明，最好的程序员要比最差的程序员强 28 倍之多。即使他们的报酬不同，优秀程序员也是软件业中最廉价的劳动力。

讨论

上一个事实的中心思想是在构建软件过程中人员非常重要，这一事实的中心思想是人员之间的差异！

这一中心思想同样与软件行业一样古老。实际上我引用的事实可以追溯到 1968 年～1978 年。似乎我们很早就已经深刻领悟到这一基本点，以至于轻易地将它抛在脑后。

这一事实具有重要的意义。假若程序员之间的差异确实大到 5～28 倍，那么软件经理最重要的任务是爱护和关心现有的优秀程序员。实际上比别人强 28 倍的人所得的报酬还不及别人的 2 倍，他们才是软件开发中最廉价的劳动力（强 5 倍的人也没有拿到应有的报酬）。

因为软件领域没有根据该事实来运作，所以其中必然存在一个问题，即我们不知道如何去寻找"最优秀"的人员。多年来，我们尝试采用程序员能力测试、数据处理认证考试、ACM 自评估程序和底线等方法，我们在这些方法上注入了心血、汗水甚至泪水，结果发现测试成绩和工作表现没有关联性（你认为这很令人

失望吗？当时我们还知道计算机科学系学生的课堂成绩和工作表现也没有关联性[Sackman 1968]）。

争议

有关这一事实的争议主要在于我们忽略了其重要性。从没有人怀疑其正确性，我们只是不记得该事实比技术重要得多。

来源

关于该事实有很多“过时的”参考文献，例如：

- “我们在研究中发现的最重要事实是不同程序员的工作效率差别极大”（Sackman 1968）。该学者在研究批处理和分时系统的效率差别时发现，不同程序员之间的差别竟达到 28:1（个体差异如此之大，以至于无法做出使用方法的有效比较）。
- “在一群程序员中，不同人的能力处于不同的数量级上”（Schwartz 1968）。Schwartz 当时在研究大型软件开发中的问题。
- “不同程序员之间工作效率的差异通常会达到 5:1”（Boehm 1975）。Boehm 当时正在研究软件高成本问题。
- “不同程序员劳动成果的差异非常巨大。例如，两个人只找出一个错误，而五个人找出七个错误。虽然大家都明白编程新手之间的编程能力差异很大，但资深程序员之间的差异也大得惊人”（Myers 1978）。Myers 最早开展了有关软件可靠性方法的权威研究。

这些引用以及其中的数字非常有份量，以至于我觉得没有理由怀疑前人的发现，除非说这个事实会随时间而改变。那么我再加上 Al Davis 书（1995）中的一些原则，“原则 132——少数优秀的人员胜过多个一般程序员”和“原则 141——软件工程师之间的差异非常巨大”。

下面是最近的说法：McBreen（2002）提议给“伟大的程序员”相当的报酬（15 万～25 万美元），而给较差的程序员较低的报酬。

参考文献

➥ Boehm, Barry. 1975. “The High Cost of Software.” *Practical Strategies for Developing Large Software Systems*, edited by Ellis Horowitz. Reading, MA:

Addison-Wesley.

- Glass, Robert L. 1995. *Software Creativity*. Englewood Cliffs, NJ: Prentice Hall.
- McBreen, Pete. 2002. *Software Craftsmanship*. Boston: Addison-Wesley.
- Myers, Glenford. 1978. "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections." *Communications of the ACM*, Sept.
- Sackman, H., W. I. Erikson, and E. E. Grant. 1968. "Exploratory Experimental Studies Comparing Online and Offline Programming Performances." *Communication of the ACM*, Jan.
- Schwartz, Jules. 1968. "Analyzing Large-Scale System Development." In *Software Engineering Concepts and Techniques*, Proceedings of the 1968 NATO Conference, edited by Thomas Smith and Karen Billings. New York: Petrocelli/Charter.

事实 3	给延期的项目增加人手会使项目进一步延期。

讨论

这是软件工程中的一个经典的事实。实际上，这不仅是事实，而且还是一个法则，即“Brooks 法则”（1995）。

直觉告诉我们，如果一个项目落后于计划，就应该增加人手来赶超工期。这个事实告诉我们：直觉是错误的。其原因是向项目中增加人手，就必须花时间来进行培训。项目新增人员必须学很多东西，才能提高效率，而且必须由项目中现有的程序员向他们传授这些东西，这一点至关重要。结果通常是：新增人员做贡献非常慢，即使他们的效率确实很高时，那也只是耗尽了原有程序员的时间和精力。

而且，项目中的程序员越多，程序员之间的相互交流就越复杂。因此，给延期的项目增加人手会导致更进一步的延期。

争议

大多数人都承认该事实的正确性。但同时会在一些细节方面有争议。例如：

如果新加入的人员熟悉当前的应用领域，甚至熟悉当前的项目，结果会怎么样？如果是这样，学习过程就缩减了，新生力量将会很快做出贡献。如果项目刚刚开始，加入人员会怎么样？如果是这样，新人提速更容易一些。

McConnell（1999）曾旗帜鲜明地反对这一事实，他认为：在实践中应当忽略这些陈腐之言，而且该事实只是在易于确定和避免的特定环境中才有效。

此外，还有少数人反对这个基本事实。但向项目中增加人手应慎重（因此，无论是在项目的早期或者后期增加人手都应该慎重。为了提速而增加人手，这种做法相当诱人，但也相当危险）。

来源

该事实诠释了一部软件工程经典著作的书名，这就是《The Mythical Man-Month》*（即《人月神话》，Brooks 著，1995）。该书指出：虽然我们采用人月（people per month）为单位来安排人手，但是每个人对软件的贡献各不相同，所以这些人月也不尽相同。在项目后期加入的人员更是如此，这些人的贡献几乎可以忽略不计。

- Brooks, Frederick P., Jr. 1995. *The Mythical Man-Month*. Anniversary ed. Reading MA: Addison-Wesley.
- McConnell, Steve.1999, "Brooks' Law Repealed." From the Editor. *IEEE Software*, Nov.

事实 4	工作环境对工作效率和产品质量具有深刻影响。

讨论

当前的软件项目基本都倾向于召集优秀的人才、采用适当的支持方法论、根据 SEI CMM 建立改进过程，以及加强内部竞争等。但是，在这一堆东西中遗漏了一些重要的东西。分析员作分析，设计员作设计，程序员作编码，测试员作测试，然而在这套机制中环境起着很大作用，甚至是决定性作用。

说到底，软件是智力密集型的，构建软件的环境应该有利于思考。拥挤以及

* 该书影印版已由中国电力出版社出版。——编者注

拥挤带来的有意无意的干扰对于工作进展有致命的影响。

致命到什么程度？对此，有一本非常经典的著作，即《Peopleware》（DeMarco和 Lister 合著，1999），该书使用较多篇幅说明环境的影响程度和影响方式。在该书中，作者介绍了他们有关工作环境对于工作效率影响方面的研究。作者选择出几个项目团队，并区分出表现最好的四分之一的团队和最差的四分之一的团队（最好的四分之一的表现是最差的四分之一的 2.6 倍）。他们随后查看了最好的和最差的团队的工作环境，结果发现前者的工作空间是后者的 1.7 倍（测量可用的地板空间）。前者的工作环境“认为安静”的比例通常是后者的 2 倍。他们还发现前者“认为尊重隐私”的比例要高出后者 3 倍多。前者电话呼叫转移或者静音的比例要高出后者 4～5 倍。前者受不必要干扰的频率大约为后者的一半。

正如我们在事实 2 中看到的，人与人之间的个体差异对软件的生产效率确实有很大的影响。但是该事实告诉我们，还有其他必需的东西。你必须找到优秀的人员并善待他们，特别是给他们提供舒适的环境。

争议

有关的争议不太明显。几乎没有人公开反对该事实。但是到了提供工作空间的时候，许多人又回到“让他们尽量紧凑”的老说法。增加额外工作空间所需的资金很容易测算，然而人员拥挤对于生产效率和产品质量的影响却很难衡量。

人们会说：“你永远不能处理身边的事情”（这是《Peopleware》中一章的题目）。经理负责工作环境有关的事务，他们被称为“家具警察”（这是那本书中另一章的题目）。然而，一直以来这种现状很少得到改观。即使在学术界（思考的时间比许多装备都更有价值），许多人也受到狭小空间的影响，工作在拥挤的或纷杂的办公室中。

有句老话说“有形胜无形”，意思是说那些可以严格度量的东西（有形的东西）可以从不能度量的东西（无形的东西）那里转移人的注意力。该事实不仅仅对软件有效，而且用在这里似乎也非常恰当。对于工作空间的度量是有形的，而生产效率的提高却是无形的，你说谁会赢？

关于该事实，有一个处于萌芽状态的争议。极限编程倡导结对编程。在结对编程中，两个软件开发者在工作中保持紧密联系，甚至共用一个键盘。在此我们看到，人为的拥挤却得到较高的生产效率和产品质量。在软件学术界还没有关于这两种观点的争议；但是随着更多人知道极限编程，该争议会变得越发激烈。

来源

关于极限编程有很多信息来源，下面是最早也是最知名的：

- Beck, Kent. 2000. *Extreme Programming Explained*. Boston: Addison-Wesley.

Laurie Williams 多次提到了结对编程，下面列出的文献中就有：

- Williams, Laurie, Robert Kessler, Ward Cunningham, and Ron Jeffries. 2000. "Strengthening the Case for Pair Programming." *IEEE Software* 17, no.4

参考文献

- DeMarco, Tom, and Timothy Lister. 1999. *Peopleware*. 2d ed. New York: Dorset House.

1.2 工具和技术

事实 5	夸大宣传是软件的瘟疫。多数软件工具对于效率和质量的提高幅度仅为5%～35%。但是总有人反复说提高幅度是“数量级”的。

讨论

在很久以前，新的软件工程思想真正有所突破，如：高级语言、类似调试器的自动工具、通用操作系统等。然而，此一时（20 世纪 50 年代）彼一时也。能够形成突破性的技术，产生 Fred Brooks（1987）所谓的银弹，这些都是很久以前的事了。

我们似乎还有第四代语言(“非程序员也能编程”)、CASE 工具(“自动编程”)、面向对象（“构建软件的最好方法”）和极限编程（“未来的方向”），以及其他类似突破。但是，这些东西的宣传中充斥着废话，它们对于我们构建软件的能力并没有显著的帮助。况且，在 Brooks 自己看来，对于突破性技术而言，最合理的观点是：不管是过去、现在还是将来，所谓的“银弹”都不可能出现。

实际上，有很确切的数据可以证明工具和技术的提高程度。从 20 世纪 70 年代至今，那些所谓的“突破”带给软件工程师的收益几乎都是有限的（不超过 35%）。而那些所谓的“突破”都声称提高了“数量级”（即数十倍），然而他

们的说法与事实真是大相径庭。

有关此话题的证据非常有力。我曾做过深入研究，考察了一些客观的学者对于这些改进的评价（Glass，1999），我发现以下几种情况：

- 因为没有相关的研究，所以评估方面的研究奇缺。
- 足够的研究可以形成一些有意义的结论。
- 没有什么证据能够证明这些东西能带来突破性的益处。
- 比较有力的证据表明确实有益处，但是幅度有限，仅为 5%～35%。

（从本文的参考文献可以查到这些客观的评估数据的原始研究资料。）

这些发现充分展示在一张表格中，这张表格包含在 Grady（1997）有关软件过程改进的一本优秀的著作中。在该表格中，作者列出了各种过程变更以及所带来的收益。你可能会问哪种过程变更的收益最大？答案是复用。Grady 认为是 10%～35%。这个数字和那些狂热者所鼓吹的“数量级”之间的差异有多大！

我们为什么会反复陷入这个宣传-期望破灭的怪圈？这个怪圈的维持需要两种人：鼓吹者和甘心上当者。事实证明鼓吹者几乎都在编造假话，他们的目的是增加销售量、提高培训课程的价格或者骗取项目研究经费。从贩卖万金油的时代开始，就总有人期望能够真正带来飞跃。

如果说总有人期望飞跃，那么我很担心那些真正迷信者。为什么总有人反复相信鼓吹者的谎言？为什么我们总是在那些被夸大的新概念上花费大量的开销和培训？回答这个问题，是当前我们软件业最重要的任务之一。如果很容易作答，那么我们就不会提出这个问题。不过在我看来，答案就在下面的材料中。

争议

所有人都承认软件业中不乏谎言。但是，人们的行为往往与自己的信念不一致。如果从理性角度看，那么几乎所有的人都同意 Brooks 提出的不存在“银弹”的说法。但是，总有一些人脱离实际，在软件工程中热心追求有突破意义的新时尚。

有时候，我认为这一局面是“羡慕硬件”的结果。近几十年来，硬件确实屡屡有突破。在硬件制造行业中，不断出现更便宜、更好、更快的替代品。研究科学技术史的朋友告诉我说，计算机硬件可能是发展最快的领域。也许我们软件人员羡慕硬件的进展，所以就违心地说软件的突破正在发生或者即将发生。

还有一件事。整个软件和硬件体系都在飞速地大踏步前进，业内人士都怕落后，所以都想接触新生事物。我们制定了产品和过程更新的生命周期，我们对“先

行者”欢呼，对“落后者”致以嘘声。计算机领域的核心文化之一就是新事物优于旧事物。如果这样，还会有谁不喜新厌旧？从个人情感出发，买入谎言是好事，站在谎言大潮前面螳臂挡车是坏事。

这一切是何等耻辱。软件行业反复受到撒谎者和追随者的戕害。更糟糕的是在你读本书的时候，又有一些新的东西粉墨登场，其狂热的追随者不仅就范，而且声称自己大受裨益，你的同事也尾随其中。我敢保证，绝对如此。

来源

对于虚假事实的鼓吹远远大于对事实的揭露。我无法追溯到第一个公开的鼓吹者，同时，如果可能我也不会再给予他们以“信任”。但是，这些都存在并继续了很长时间。例如，我的第一本书是《The Universal Elixir, and Other Computing Projects Which Failed》(1976)，其中讲了一些故事来讽刺这些兜售万能药的鼓吹者。这些故事在收入本书之前的十几年就（以匿名 Miles Benson）发表于《Computerworld》杂志上了。

还有更多的文献承诺万能药，紧随其后便有声音质疑其误导。在这里和随后的参考文献一节中，有如下记载：

- Davis, Alan M. 1995. *201 Principles of Software Development*. New York: McGraw-Hill. Principle 129 is “Don’t believe everything you read.”
- Weinberg, Gerald. 1992. *Quality Software Development: Systems Thinking*. Vol. 1, p.291. New York: Dorset House .

参考文献

- Brooks, Frederick P., Jr. 1987. “No Silver Bullet—Essence and Accidents of Software Engineering.” *IEEE Computer*, Apr. This paper has been published in several other places, most motably in the Anniversary Edition of Brooks’s best-known book, *The Mythical Man-Month* (Reading, MA:Addison-Wesley, 1995), where it is not only included in its original form but updated.
- Glass, Robert L. 1976. “The Universal Elixir, and Other Computing Projects Which Failed.” *Computerworld*. Republished by Computing Trends, 1977, 1979, 1981 and 1992.
- Glass, Robert L. 1999. “The Realities of Software Technology Payoffs.”

Communications of the ACM, Feb.

➥ Grady, Robert B. 1997. *Successful Software Process Improvement*. Table 4-1, p.69. Englewood Cliffs, NJ: Prentice Hall.

事实 6	在学习新工具或者新技术的初期，程序员的工作效率和产品质量都会下降。只有克服了学习曲线以后，才可能得到实质性的收益。只有满足下面两个条件，采用新工具或新技术才有意义：（a）新东西确实有用；（b）要想获得真正的收益，必须耐心等待。

讨论

如果新技术或者新工具确实有价值，那么学习它是件好事。但是其价值可能不如以前的尝试者所说的那么大。学习新思想需要付出代价。我们必须领悟新思想，考虑新东西是否适用于我们的工作，确定采用的方法和时机。因为我们不得不认真思考原本很自然的东西，所以就慢了。

无论是首次使用测试覆盖分析器并判定该方法在测试中的意义，还是尝试极限编程中的新技术，采用这些新思想的人的效率都很差。这并不是说要回避这些新思想，这只说明采用新思想的第一个项目会比以往的项目更慢，而不是更快。

近二十多年来，软件过程不懈追求的目标是提高生产效率和产品质量。我们采用新技术的目的同样也是提高生产效率和产品质量。但是技术转变的过程非常有意思，当我们改变工艺尝试新东西时，生产效率和产品质量先是下降，而不是上升。

不用紧张。如果新东西确实有效，那么其效益迟早会呈现出来。但是这就引出了一个问题：需要多久？这就是我们讨论的学习曲线问题。在学习曲线中，刚开始时效率和效力都急剧下降，然后超过原有水平，最后稳定在该技术应有的水平。如果这样，“需要多久”这个问题转变为多个“需要多久”。收益递减需要持续多久？回归到收益持平需要多久？获得最大收益需要多久？

在此，我们每个人都希望采用三个月或六个月这种说法。但是，我们对于这个问题，不能这么说。学习曲线的长度因形势和环境而异。通常，最终的收益越大，曲线就越长。简单地学习面向对象编程可能只需要三个月，但是要想熟练掌握可能需要整整两年。对于其他的东西，曲线长度又截然不同。惟一可以预见的是必然会有个曲线。你可以在坐标上画一个曲线，但是却无法确定坐标轴的有效尺度。

还有一个问题是“影响有多大”。新思想能带来多大收益？该问题如同“需要多久”一样不可知。但是有一点很明确，根据我们在前面事实中得到的结论，新思想的效益很可能会比以前高5%～35%。

还有一个因素，虽然我们对于“需要多久”和“有多大”不能给出统一的答案，但还是有答案的。对于任何一个新思想，比如测试覆盖分析器或者极限编程，你可以从有经验的人那里得到参考。找一些已经掌握了新思想的人，通过结对编程、用户组、职业活动等，询问他们“需要多久”（他们通常知道该问题的答案）和“有多大”（这个问题比较难回答，只有做软件度量的人才知道答案）。还有，通过询问这些问题，发现他们在采用新思想的过程中得到哪些正反两面的教训。

当然，询问时应避开狂热者，他们有时候给出的答案出自信念，而不是出自经验。

争议

关于这个问题原本不应该有争议。很显然，学习新东西必然要付出代价，但是实际上却经常有争议。狂热者通常将巨额收益和快速的学习曲线上升为管理者的意志（见事实5）。经理采用新方法，并要求新方法一旦引入就步入正轨。在这种条件下，他们在估算成本和制定计划时，总是设想一开始就能得到收益。

下面是关于这种效应的一个经典的故事。一个医药企业安装了一个SAP，设想可以很快就从SAP收益，所以在一些项目中以低价竞标。该公司后来懂得了这个曲线，却因为破产和诉讼而关门了。我们由此得到以下教训：一定不要依靠设想，必须要对短期内无法实现的项目做好打算。

来源

有许多文献都涉及学习曲线及其对改进的影响，包括：

- Weinberg, Gerald. 1997. *Quality Software Management: Anticipating Change.* Vol. 4, pp.13, 20. New York: Dorset House.

事实7	软件开发者对于工具说的多，评估的少，买的多，用的少。

讨论

工具是软件开发者手中的玩具，开发者热衷于了解、尝试使用和获得新工具。

然后很有意思的是，他们很少使用这些工具。

近十几年来，出现了一个闪亮的术语“CASE”。在 CASE 工具的运动高潮中，人人似乎都相信 CASE 工具是未来软件的发展方向，是一种高度自动化的软件开发方法，这时 CASE 工具的销售量巨大。但是其中多数工具都被束之高阁，从未用过，我们用一个新术语阁件（shelfware）来描述这一现象。

我是这场运动的受害者之一。当时，我正在教授有关软件质量的课程。在课上我表述了自己的观点，即这些 CASE 工具不可能达到别人所说的那种突破程度。课后有些学生找到我，说我对 CASE 的认识已过时了，他们坚信这些工具确实可以使软件开发过程自动化。

我非常严肃地看待这几个学生的问题，立即集中精力学习 CASE 知识，以避免有所疏漏，保证自己不过时。随着时间的流逝，我原来的观点再一次被证实。我很快得知 CASE 工具确实有用，但是肯定不能达到不可思议的突破，这些希望破灭的同时也就产生了 shelfware。

我跑题了。该事实不是讨论工具是否突破（我们在事实 5 中已讨论过了），而是说多数工具确实能够有效地提高工作效率，那么这些工具为什么也会被束之高阁？

回想我们在事实 6 中所说：尝试新工具时，一开始的工作效率不是上升，而是下降。尝试新工具的冲动一旦退去，开发者不得不制定真正的计划来开发真正的软件。这样，开发者通常又回到自己熟知的领域，采用惯用的工具。他们通常会采用自己熟悉的语言以及相应的编译器、调试器、文本编辑器、链接器、加载器，以及去年的（甚至是过去近十年的）配置管理工具。难道你的工具箱中缺少工具？更不用说覆盖分析器、条件编译器、标准兼容检查器或者其他工具。开发者会说这些工具很有趣，但是在生效之前它们确实是个负担。

除此之外，还有一个有关软件领域中工具的问题。不存在“最小标准工具集”，也就是说无法定义程序员的工具箱中应该用哪些工具。假设有了这个工具集，那么程序员可能会使用其中的工具。目前似乎没有人致力于研究这个问题。（IBM 曾经做出一个工具集 AD/Cycle，但是它不仅昂贵而且设计得很不好，所以市场表现一般。此后就再没有人尝试做工具集了。）

争议

软件参与者通常存在“不合作”（not-invented-here，NIH）现象，他们因喜欢

构建自己的东西，不愿意在别人的工作基础上做开发而受到指责。

当然一些业内人士确实如此，但是不像人们想像的那么多。在我所认识的许多程序员中，如果让他们在新旧事物之间做出选择，他们通常会在可以保证更快完成手头工作的前提下选择新事物。因为这个前提总不能满足（还是因为前面提到的学习曲线问题），所以他们又回到了旧的、可靠的、真实的事物。

我敢肯定这个问题不是 NIH，而是属于文化范畴。我们采用不切实际的时间表，而且按其行事，这是过于重视时间表而无暇学习新观念的文化观念。有些商品目录（例如 ACR）介绍各种商业工具的特征、来源和用途。程序员很少知道市面上的工具，而知道工具商品目录的就更少了。我们将在以后的事实中讨论这些思想。

因为人们很少考虑最小标准工具集的问题，所以没有争议。试想，如果大家开始考虑这个问题，也将会有非常有意思的争议。

来源

- ACR. The ACR Library of Programmer's and Developer's Tools, Applied Computer Research, Inc., P.O.Box 82266, Phoenix AZ 85071-2266.这是每年都更新的软件工具目录，但是近期已停止发行了。
- Glass, Robert L. 1991. "Recommended: A Minimum Standard Software Toolset." In *Software Conflict*, Englewood Cliffs, NJ: Yourdon Press.
- Wiegers, Karl. 2001. Personal communication. Wiegers 说："我已经反复说过这个事实。它曾经发表于 David Rubinstein 对我的一个访谈中，发表在 2000 年 10 月 15 日的《Software Development Times》上"。

1.3 估算

事实 8	项目失控的两个最主要的原因之一是糟糕的估算（另一个原因见事实 23）。

讨论

失控项目指难以控制的项目。这种项目通常不会产生任何成果，即使有成果，也会滞后于时间表、超出预算。回想起来，失败项目太多了。有些项目是"向死

亡前进”，有些是以“危机模式”运作。不论怎么称呼，不论结果如何，失控项目总不是一个好现象。

在软件工程领域，人们经常询问项目失控的原因。对该问题的回答通常与回答者的个人偏见有关。有些人说因为缺少正确的方法论，这些人都是兜售方法论的。还有人说缺少优秀的工具（猜一下这些人靠什么维生？）。还有人说程序员缺少纪律和自律（这些人倡导的方法通常主要基于严格的纪律）。只要有一个鼓吹的理念，就会有人说因为其缺乏而导致项目失控。

幸运的是，在这些喧嚣和吵闹中，存在真正客观的答案，这个答案出自无所求的人。这些人的答案惊人地一致：使项目失控的最主要的两个原因是糟糕（通常是太乐观）的估算和不稳定的需求。对这两点都有人做过研究。

在这里我们主要讨论估算（后面将讨论需求）。正如你所知，估算是确定项目成本和开发时间表的过程。在软件领域，估算通常都非常糟糕，我们的许多估算不是客观目标，而是主观愿望。更糟糕的是，我们不知如何改进这种糟糕的做法。最终的结果是因为谁都不会得到可行的估算值，所以就忽略了正确的方法，而走捷径。因此，不可避免的项目失控而引发了方法失控。

我们曾经尝试过各种看似合理的方法来提高自己的估算能力。首先我们依靠专家，所谓的专家就是有软件开发经验的人。这种做法的问题在于它非常主观化。不同的人因为不同的经验形成不同的估计。实际上，无论专家曾经做过什么项目，无论这些项目与当前项目何等相似，专家都不可能做出合理的推断（软件项目的重要特征之一是各个项目所解决的问题差异很大。后面我们将讨论这个话题）。

接着我们尝试了估算算法。计算机科学家变成了数学家。他们简单地设计出含有参数的公式（通常根据以往项目得到）来解决估算问题。输入一些与项目有关的数据，动用算法的威力，便可以得出看起来高度可信的估计。但是，这是不可行的。多项研究（例如 1981 年 Mohanty 的研究）表明：采用一个假想的项目，按照各种建议算法输入相关数据，得到的结果差异很大（2～8 倍）。算法和专家估计一样，都存在不一致的问题。随后的研究印证了这一不幸的发现。

如果复杂的算法不能用，那么简单的算法是否可行呢？业内许多人提倡根据一个或少数几个关键数据，例如代码行数（line of code，LOC）来估算。有人说：如果我们可以估算一个系统的代码行数，那么就可以根据代码行数确定项目的时间表和成本（这种想法有点可笑，因为估算代码行数似乎比估算时间表

和成本更难。可是许多明智的计算机科学家仍然倡导这种做法）。此外，还有“功能点（function point，FP）”的方法。有人说我们应该着眼于系统的输入输出数量等关键参数，在此基础上做估算。实际上，功能点方法的问题很多。首先，专家对于什么是功能点和如何计算功能点有争议。其次，功能点对于有些应用有效，对于另一些项目则无效，例如程序中的输入输出比业务逻辑简单得多（有些专家针对“功能”不重要的应用，将功能点修正为“特征点”。但是这逃避了原来的问题，针对多种应用，我们到底需要多少种“点”的计数方法？对此无人知晓）。

问题的现状是：在21世纪的头十年，我们并不知道如何进行有效的估算，如何成功地预言项目的时间和成本。这个现状让人失望。这个现状说明：在避免“危机模式”的过程中，仍旧把有问题的时间表和成本估计作为软件项目的关键控制因素，我们就不要指望能够有所改善。

应该注意的是，失控的项目，至少是因为糟糕的管理而失控的项目，发生的原因并不是程序员工作得不好。这些项目失控的原因是：在刚开始时，对于管理的内容就估算错了。我们将在下面的几个事实中讨论这些因素。

争议

软件估算很糟糕，这一点几乎无可争议。然而有很多争议涉及到如何改进估算。例如，提倡算法的人倾向于支持自己的算法，而抵毁他人的方法。提倡功能点方法的人总是说LOC方法如何不好。Jones（1994）将LOC列为软件职业中最糟糕的“两种弊病之一”，甚至将其称作“玩忽职守的管理”。

幸运的是，如果不考虑估算的准确性，那么有些争议可以解决。大多数学习估算方法的学生开始认为“带子和吊带裤（belt and suspenders）”方式是这个大难题最好的折衷处理方法。他们认为估算应包含两方面内容：一是专家的观点，二是算法对历史问题和当前问题的计算结果。这样才能得到相对合理和准确的答案。来自两方面的估算结果可以帮助确定当前项目的大致估算范围。二者的结果可能不一致，但是对于估算范围多少有一点了解比什么都没有好。

近期的一些研究表明“人为调节（human-mediated）估算方法可以得到非常准确的结果”，比“简单算法模型”好得多（Kitchenham等，2002）。这种观点强烈倾向于使用专家估算方法。它是否真的经得起验证？我们将拭目以待。

来源

很多研究结果表明，估算失误是引起软件失控的两个重要原因之一。下面是两个例子，下一节还有 3 份参考文献。

- Cole, Amdy, 1995. “Runaway Projects—Causes and Effects.” *Software World (UK)* 26, no. 3. 本文是有关失控项目及其原因、结果和人的作用等方面最好的客观研究。本文得出的结论是“糟糕的计划和估算”是 48%项目失控的主要原因。
- Van Genuchten, Michiel.1991. “Why Is Software Late？” *IEEE Transactions on Software Engineering*, June. 该研究的结论是：“乐观的估算”是 51%项目失控的主要原因。

还有一些书指出项目因为错误的时间表而陷入困境。

- Boddie, John. 1987. *Crunch Mode*. Englewood Cliffs, NJ: Yourdon Press.
- Yourdon, Ed. 1997. *Death March*. Englewood Cliffs, NJ: Prentice Hall.

参考文献

- Jones, Caper. 1994. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ:Yourdon Press. This strongly opinioned book cites “inaccurate metrics” using LOC as “the most serious software risk” and includes four other estimation-related risks in its top five, including “excessive schedule pressure” and “inaccurate cost estimating.”
- Kitchenham, Barbara, Shari Lawrence Pfleeger, Beth McCall, and Suzanne Eagan. 2002. “An Empirical Study of Maintenance and Development Estimation Accuracy.” *Journal of Systems and Software*, Sept.
- Mohanty, S.N. 1981. “Software Cost Estimation: Present and Future.” In *Software Practice and Experience*, Vol. 11, pp. 103-21.

事实 9	许多估算是在软件生命周期开始时完成的。后来，我们才认识到在需求定义之前，即理解问题之前进行项目估算是不正确的；也就是说，估算时机是错误的。

讨论

为什么软件估算很糟糕？我们将通过一些事实来解释该问题。

该事实涉及到估算的时机。我们通常是在项目刚刚开始时做估算。听起来没错，对吧？难道你还想在其他时候估算？但是，应该注意，要想得到有意义的估算，必须对当前的项目有一定的了解，至少应该明白所要解决的问题。但是，软件生命周期的第一步是确定需求。这就是说，在开始时我们确定了项目应该解决的问题。说白了，确定需求就是确定需要解决的问题。如果你还不明白将面临的问题，你怎么能估算解决问题所需的时间。

这非常可笑。我在软件业的各个论坛上提出这一事实，并询问谁对此有异议。这毕竟是一个显而易见的问题，到目前为止，没有人提出异议，大家都点头表示理解和同意。

争议

很奇怪，对于这个事实似乎没有争议。正如我前面所说的，似乎大家一致同意这个事实。但该事实所反映的实际情况却很荒唐，有些人的做法急需改进，他们没有按照正确的方法进行操作。

来源

我想该事实如同都市传说和老妇人的桌子一样，很难追溯其来源。然而，有很多地方明确定义了“错误时机”的问题。

- 在一篇问答型的文章中，Roger Pressman 引用了一个问题：“我的问题是提交日期和预算是在项目开始之前确定的。我的管理者所问的惟一问题是‘我们是否可以在 6 月 1 日之前交付？’如果最终期限和预算都已事先确定了，那么再做详细的项目估计还有什么意义？”（1992）。
- 一个基于预测算法工具的广告词（SPC 1998）描述了一段很常见的对话：“市场部经理：‘那么你认为这个项目需要多长时间？’你（项目主管）回答说：‘大约 9 个月。’市场部经理：‘我们计划将该产品调整到最多 6 个月。’你：‘6 个月？’市场部经理：‘你可能还不了解，我们已经对外宣布了发布日期。’”

注意，在这些引用中，不仅估算的时机不对，而且估算的人员也不对。后面

的事实中将讨论估算人员的问题。

参考文献

- Pressman, Roger S. 1992. "Software Project Management: Q and A." *American Programmer* (now *Cutter IT Journal*), Dec.
- SPC. 1998. Flyer for the tool Estimate Professional. Software Productivity Centre (Canada).

事实 10	许多软件项目都是由高层管理人员或者营销人员来估算，而不是由真正构建软件的人或者他们的主管来进行估算。因此，估算软件的人员是错误的。

讨论

该事实说明了软件估算糟糕的第二个原因。该事实涉及到应该由谁来进行软件估算。

常识告诉我们，估算软件的人应该是了解该软件的人，比如软件工程师、他们的上司和项目经理。但是，政治在这里战胜了常识。现在通常估算软件项目的人是需要软件产品的人，包括高层管理者、营销人员、客户和用户。

换句话说，当前的软件项目估算数据不是客观真实的，而是主观期望的。高层管理者或营销人员希望在下一年第一季度之前完成某软件，所以开发人员就必须按该时间表完成。请注意，在这种情况下，基本上没有做任何“估算”。源于其他无形过程的时间表和成本目标被强加于该项目。

下面，我来讲一个有关软件估算的故事。我以前的经理富有航天领域的工作经验，他非常聪明。他想把一些软件外包给另一家航天公司。在谈判中，他告诉对方预期的提交日期，对方的回答是不能如期完成。猜一下合同中的最终提交日期是什么？随着时间的流逝，预期的日期过了，软件最终按照承包商所说的日期完成了。但是这超过了合同日期（我想你明白合同中提交日期是什么了），承包方自然要按照合同交罚款。

这个故事中有两个要点。一是即使非常聪明的高级管理人员，在受到政治压力时，也无能为力。二是如果项目的最终期限不合实际，那么总会付出代价。这种代价通常是与人有关的，例如名誉、道德、健康等，而在这个过程中我们还看

到了经济方面的代价。

该事实告诉我们，经常由错误的人来估算软件，这对整个软件行业造成了恶劣的影响。

争议

这又是一个无可争议的事实，几乎所有的人都承认实际情况确实如此。这一事实确实指出了在明白软件的人和不明白软件的人之间的重大隔阂。也许该事实是已有的隔阂所引起的结果。

解释一下，该事实所导致的隔阂是：在估计项目时，软件人员可能并不知道什么是可能的，但是他们总知道什么是不可能的。如果高级管理人员或者营销人员不听取这些软件人员有价值的警告，软件人员会不再信任那些决策者，他们的工作激情也会消失殆尽。

另一方面，在软件人员和高级管理人员之间也存在长期隔阂。高级管理人员因为软件人员经常不能如期完工，所以对他们失去了信心。即使软件人员说自己不能如期完工，高级管理人员也不会在意。毕竟软件人员的言行总不一致，谁还有理由信任他们？

来源

有人曾仔细研究过这个问题，并证实该事实。Lederer（1990）研究了所谓的“政治”预测与“理性”预测问题。（用词很精彩，你可能会猜到高级人员和营销人员搞“政治”预测[我尤其喜欢这个用词]。相反，软件人员做“理性”预测。他的研究表明政治预测是主导者。）

另一份研究报告（CASE 1991）指出许多（70%）估算是由与“人事部门”有关的人员完成的，很少（4%）由项目团队进行估算。不论“人事部门”是不是高级管理人员或者营销人员，都可以肯定这些参与估算的人是错误的。

在其他应用领域也有类似的例子（请注意这些研究是针对信息系统）。例如，事实9中的两个引用不仅估算的时机是错误的，而且估算的人也是错误的。

参考文献

- CASE. 1991. “CASE/CASM Industry Survey Report.” HCS, Inc., P.O. Box 40770, Portland, OR 97240.

➥ Lederer, Albert. 1990. "Information System Cost Estimating." *MIS Quarterly*, June.

事实 11	软件估算很少根据项目进度进行调整。因此，这些估算通常是错误的人在错误的时间得出的错误结果。

讨论

我们再次考虑有关软件项目估算的常识。无论最初的估算怎么糟糕，随着项目的进展，人们眼前的形势会日益明朗。我们是否可以调整原来的估算，使之切近现实？但是实际情况确实有悖于常识。软件人员对于原本错误的估算逆来顺受。高级管理人员通常不乐意调整原来的估算。如果说这些估算代表的是主观愿望而不是客观情况，那么高级管理人员的愿望岂能让人随意篡改？

我们可以跟踪多个项目的进展，确定里程碑的偏离以及最后里程碑的延迟情况。但是当我们统计项目的结果时，总是用原始的估算数据来评判项目的成败。请注意，这些数据是错误的人在错误的时机编造的。

争议

显而易见，不去修定早期的项目估算是一种错误的做法。但是还有极少的人反对这一点（有一项研究[NASA 1990]建议在生命周期中重新估算，我想没有人会听取该建议）。因此虽然对该事实应该有很多争议，但是我一点都没见到。软件人员只能接受一个规定，即不得调整对自己工作的估算。

如果一个项目出现了将要超出原始估算的趋势，自然会有人大声询问何时才能完成。我又想起丹佛国际机场行李处理系统。微软新发布的每一款新产品也如此。因此，在政策性估计和真实性结果之间存在冲突，但是这种冲突的主要后果几乎总是指责软件工作人员。“对无辜者的指责”一次又一次得逞，而常识却总是失败。

来源

对于这个问题，我知道没有具体的研究报告。但是在流行的计算机出版物和软件管理书籍中，有很多关于不及时调整估算导致项目失败的故事。我依靠这些东西来证明该事实：

1. 前面提到的例子（丹佛国际机场和微软产品）。
2. 我本人在软件领域 40 多年的工作经验。
3. 我的有关软件项目失败的大量书籍。
4. 我在公共论坛上提出该事实并请读者批评指正时，没有人提出异议。

参考文献

➥ NASA. 1990. *Manager's Handbook for Software Development*. NASA-Goddard.

事实 12	因为估算的数据如此糟糕，所以在软件项目不能达到估算目标时，不应该再考虑估算。但是无论如何，每个人都在考虑它。

讨论

这个事实又是一个常识。假如软件的估算结果非常糟糕——错误的时间、错误的估算者、不可改变的现象——那么你就可以忽视估算，对吗？不对！实际上，软件项目总是按照时间表来管理。因此，至少在高层管理人员看来，时间表是软件中最重要的因素。

从专业角度看，根据时间表管理意味着建立一组短期的和长期的里程碑，通过项目在里程碑的情况确定进展是否正常。如果你在里程碑 26 处落后于时间表，那么，你的项目正陷入困境。

我们还有其他管理软件项目的方法吗？下面我给出一些例子来说明时间表并非惟一的管理方法。

- 我们可以根据产品来管理。可以根据有多少可行的或者正常工作的最终产品来判定成功或失败。
- 我们可以根据问题来管理。可以根据项目中问题的解决程度和问题处理速度来判定项目成功或失败。
- 我们可以根据风险来管理。可以根据在项目之初确定的一系列风险是否被克服来判定成功或失败。
- 我们可以根据商业目标来管理。可以根据软件在多大程度上提高了商业性能来判定成功或失败。
- 我们可以根据质量来管理。可以根据在产品中成功地实现多少质量指标

来判定成功或失败。

我可以听到你在考虑我这些话时，低声地叨唠“这小子真天真”。“在这个快节奏的时代，时间表确实比什么都重要。”也许是吧。但是，难道不是因为基于估算的管理，出现了可控制性最低、正确性最低等许多亟待解决的管理问题吗？

争议

每个人（包括软件人员）都简单地接受根据时间表进行管理的工作方式。虽然根据时间表管理会带来许多烦人的不良效果，但是好像没有人站出来提出改进方法。

然而似乎有些新观念要动摇这一现状。极限编程（Beck 2000）建议客户或者用户先确定成本、时间表、功能和质量这四个因素中的前三个，软件开发者再确定第四个。极限编程很好地指出了软件项目中的关键问题，以及由哪种参与者来决定具体的估算值。极限编程还建议：因为在估算中错误的权力结构是导致糟糕估算的主要原因，所以必须改变。

来源

对于这个事实，我没听说过有什么研究。但是在软件工作室，我做过很少几个试验来解释这个问题。下面，我来描述其中一个试验。

我要求参与者完成一个小任务。我有意增加他的工作量，使工作时间不足。我希望参与者尽力正确地完成整体工作，这样他们会因为时间不够而得出一个未完成的产品。事实并非如此，这些参与者按照不可能的时间表勉强完成了工作。他们的产品粗糙且虚假，看起来完整但根本不可用。

这说明了什么？说明在我们今天的文化中，人们都尽力满足了不可能的时间表，以至于他们愿意为此牺牲产品的完整性和质量。说明存在一个盛行的因果流程，即：人为了错误的原因来做错误的事情。最后，最令人不安的是：这种境况很难改变。

在随后的“参考文献”一节中的材料很好地描述了极限编程。

参考文献

➥ Beck, Kent, 2000. *eXtreme Programming Explained.* Boston: Addison-Wesley.

事实 13	在管理者和程序员之间存在隔阂。对于一个未满足估算目标的项目的调查表明：从管理者看来这是一个失败的项目，而在技术人员看来却是最成功的项目。

讨论

下面会解释这个看起来有点自相矛盾的事实。虽然前面讨论了许多有关估算的问题，但惊人的是，许多技术人员努力忽视估算目标和最终期限。正如我在事实 12 中所讲述的软件工作室实验，技术人员通常不会成功。但是，许多人都明白估算都很不真实，他们都希望根据估算以外的因素来决定项目的成败。

有一项研究虽然不能掀起风浪，却很有说明力、很重要。我不得不将它列出来，这项研究中的情况以后可能会经常发生。

这一节讨论的重点是 Linberg（1999）做的一个调查性项目。Linberg 研究了一个真实的软件项目，管理者认为该项目是一个失败的项目，该项目严重超出了估算目标。Linberg 询问该项目中的技术人员曾经参与过的最成功的项目是什么。情况变得很复杂！那些技术人员，或者至少 8 个人中的 5 个，认为最近这个项目是最成功的。在参与者看来，管理者认为失败的项目是巨大的成功。这是关于我们所说的隔阂的一个多么奇怪的例证！

这个项目到底怎么样？有关事实如下：它达到了经费预算的 419%；超过了时间表计划，达到其 193%（估算 14 个月，实际用了 27 个月）；软硬件的规模分别达到估算的 130%和 800%。但是项目成功完成了。它实现了预期的目标（控制一台医疗设备），也满足需求中的“提交后没有软件缺陷”。

因此，这种隔阂是有根源的。估算的目标几乎无法完成。但是一旦可用，软件产品将很好地满足预期要求。

还有，将一个可用的、有用的产品投入使用，这难道不是项目“最成功”的原因吗？这难道不是一个你以前多次期望这些技术人员拥有的经历吗？根据这项研究，这些问题的答案都是“对的”。实际上，这些技术人员认为这是一个成功的项目，还因为其他一些原因：

- 该产品按照预期的方式来运转（这并不奇怪）。
- 开发这一产品曾经是一个技术挑战（许多数据表明，几乎所有的技术人员都喜欢解决有难度的问题）。
- 团队小而效率高。

- 技术人员对项目管理的认识是“我遇到的最好的”。为什么？“因为团队可以自由地完成一个优秀的设计”，因为没有“边界变迁”，因为“技术人员从未感到时间表的压力。

当 Linberg 询问这些参与者如何理解该项目为何延期的问题时，他们的回答又增加了以下几点：

- 这个时间估算不现实（是的，谢谢！）。
- 缺少资源，特别是专家建议。
- 在开始时对于涉及的领域知识理解得不好。
- 项目起步晚了。

在这些见解中，有一点非常有意思。它们都是关于在项目开始时看似正确的事项。不是在项目进行过程中，而仅仅是在开始时。换句话说，在这个项目的第一天已经投下了赌注。无论技术人员如何努力、如何认真地工作，他们都不可能满足管理者的期望。如果这样，他们会从自己的观点出发做出最好的东西，开发一个有用的产品，他们这样很得意。

有关管理和技术理解方面的其他研究，也表明存在一个较大的隔阂。例如，在一项研究中，Colter 和 Cougar（1983）询问管理者和技术人员关于软件维护的认识。管理者认为改动通常比较大，涉及到大约 200 行的代码；技术人员认为实际只需要改动 50～100 行代码。管理者还认为需要改动的代码行数与完成这项任务的时间相关；而技术人员认为二者没有关联性。

有证据表明，对于失控软件，技术人员比他们的管理人员察觉的时间要早得多（72%的时间）（Cole 1995）。这同时还意味着：技术人员意识到了，但是没有报告给管理者——这是最根本的隔阂。

也许对于该主题最吸引人的评论是两篇有关项目管理的文章。Jeffery 和 Lawrence（1985）发现“根本就没有做过估计的项目的进展速度最快”（其次是由技术人员做估计的项目，最糟糕的是，由管理人员做估计）。Landsbaum 和 Glass（1992）发现“在工作效率和驾驭感之间有非常强的关联性”（也就是说，如果程序员感觉到能驾驭自己的结局，那么他们的工作效率会高得多）。换句话说，高度控制的管理并不一定会得到最好的或者效率最高的项目。

争议

该事实包含两方面的本质问题：一是项目成功的要素是什么；二是管理人员

和技术人员间的隔阂问题。

关于成功的问题，Linberg 的研究发现还没有见诸于他的论文中，所以还没有时间因此而形成争议。我猜想，管理人员在阅读和回想该事实时，会因为技术人员将一个明显失败的项目看作成功的项目而感到恐慌。我还猜想，技术人员在阅读和回想该事实时，会发现这都非常合理。如果我的猜想是正确的，那么围绕该事实所提出的问题会出现沉默的争议。该争议是：什么是项目成功的组成要素。如果我们在成功项目的定义上不能达成一致，那么这个领域就存在更大的问题。我猜想，你根本没有听说过这个事实和问题。

关于隔阂问题，在学术界我没有看到任何有关的评论。来源于 Jerrery 和 Landsbaum 的引用是从管理的底层向上对于这些问题传统的看法，而不是真正有价值的信息。

来源

除了参考文献之外，还有一个相关的来源：

- Procaccino, J. Drew, and J.M.Verner. 2001. "Practitioner's Perceptions of Project Success: A Pilot Study." *IEEE International Journal of Computer and Engineering Management*.

参考文献

- Cole, Andy.1995. "Runaway Projects—Causes and Effects." *Software World* (*UK*) 26, no.3.
- Colter, Mel, and Dan Coufer. 1983. From a study reported in *Software Maintenance* Workshop *Record*. Dec.6.
- Jeffery, D. R., and M. J. Lawrence. 1985. "Managing Programmer Productivity." *Journal of Systems and Software*, Jan.
- Landsbaum, Jerome B., and Robert L. Glass. 1992. *Measuring and Motivating Maintenance Programmers*. Englewood Cliffs, NJ: Prentice Hall.
- Linberg, K. R. *1999*. "Software Developer Perceptions about Software Project Failure: A Case Study." *Journal of Systems and Software* 49, nos. 2/3, Dec.30.

事实 14	对于可行性调研的回答几乎总是“可行”。

讨论

错误的可行性调研影响了软件领域的许多方面。其中一个方面是我们“不会得不到尊重”。我们帮助那些传统人士解决了几十年都没有软件的问题，他们对我们非常感激，从现在起，他们已经不能没有我们了。

一位工程管理人员针对一个无人驾驶飞机项目发表上述观点时，这只是“荒谬剧场”的演出。问题不在于无人驾驶飞机离开了计算机和软件就不能工作。这个家伙想丢下所有的技术问题，继续做他的项目。

错误的可行性调研另一个影响我们的方面是我们似乎经常有一种不可救药的乐观。似乎我们可以解决别人都不能解决的问题，也相信没有难得无法解决的新问题。而且，惊人的是事实通常确实如此。但是，有些时候却并非如此，在这时候，乐观主义使我们陷入困境。例如，我们相信自己可以在明天完成该项目，或者最迟两天以内完成；我们相信在自己的产品中总是没有错误，却发现错误消除的时间比系统分析、设计和编码的时间总和还长。

第三个方面是可行性研究。如果技术可行性确实有问题，那么乐观主义一定会给我们带来麻烦。我们很少在开始之前真正做可行性研究，而且研究的结果几乎总是“是的，我们可以”。但是，我们在几个月之后才会发现，在百分之几的情况下，这个研究结果是错误的。

争议

在可行性研究得出错误答案与发现研究结果错误之间的时间间隔很长，以至于我们几乎不会将两件事联系在一起。因此，关于该事实的争议比预期的要少。可行性研究是否有必要做（很少有人做）可能比可行性研究经常得到错误结论更有争议。

来源

该事实的来源非常有意思。我 1987 年在东京参加软件工程国际会议（International Conference on Software Engineering，ICSE），著名的 Jerry Weingerg 是一位主要发言人。他在演讲期间问到有多少人曾经参加过可行性研究，回答都是“没有”。在观众中出现了少有的安静，然后是笑声。没有一个人举手。我认为：

我们在座的1500人同时认识到Jerry的问题触及该领域的重要现象，一个我们以前从未考虑过的现象。

1.4 复用

事实15	小规模的复用（子程序库）开始于50多年以前，这个问题已经得到很好的解决。

讨论

计算机界通常认为任何优秀的观念都是新观念。对于复用也是如此。

实际上，复用的概念与软件行业一样悠久。在20世纪50年代中期，成立了一个有关IBM大型机（近来已不再使用这个术语）在科学应用方面的用户组织，该组织最重要的功能之一是作为交换软件子程序的场所。这个组织叫做Share，会员所捐献的子程序可能就是世界上第一个可复用的软件库。在计算机发展的初期，只要向该软件库中捐献高品质的子程序，就可以在业界获得良好的声誉（然而，这样做不能获得金钱。在那个时代，软件只是随硬件免费赠送，没有经济价值。注意，又有一个优秀观念不是新观念，即开放源代码或者免费软件）。

在这些早期的软件子程序库中，包括今天我们所说的小规模复用子程序，例如，数学函数、排序和组合、局部调试器、字符串处理程序等。所有这些子程序库满足了许多程序员的需求。其实，我就是因为向Share库中捐献了一个调试补丁而首次成名（名气相当有限）。

从那时起，软件开发中就采用了复用的方法。如果你的软件中需要一些常用的功能，那么你应该首先到Share库查看是否已经有现成的东西（其他用户组织，如Guide和Common等，可能也有针对自己应用领域的软件库。当时，我不是商业性软件的程序员，所以不清楚Guide和Common是否也和Share一样运作）。我记得自己曾写过一个随机数生成器，随后在Share库中也找到了一个（在Share库中有很多随机数生成器，从简单随机数生成器到正态分布随机数生成器一应俱全）。

当时的复用是用尽一切方法能抓到什么就算什么，对于加入库的东西没有质量控制。然而，能将自己的名字写入Share库中就是最高的待遇。捐献者在提交

之前尽力消除自己作品中的错误。在复用 Share 库中的程序时，我没有发现任何质量问题。

我们为什么要回顾历史？这是因为历史对于理解复用现象、复用的现状非常重要，而且历史还能帮助我们认识到复用是一个很古老、很成功的观念。继小规模复用成功之后，人们在大规模复用方面投入了大量的精力，但是多年来并没有获得明显改观。有关原因分析见事实 16。

争议

有关的争议主要是业内的许多人都认为复用是一个全新的观念。所以，复用有许多狂热的追随者（通常是受骗的）。如果能理解复用的历史以及多年来进步甚微的事实，那么这些狂热者可能会变得更现实些。

来源

我对于早期的复用概念记忆犹新。实际上（很不谦虚地说），对于该事实最好的解释是我的个人职业回顾（Glass 1998）。你也可以在用户组织 Share（至今还存在）中找出一些早期的文档（实际上，该组织已经建立了我们所谓的工具和分类目录，用户可以根据问题在目录中找出相应的子程序）。

参考文献

- Glass, Robert L.1998. “Software Reflections—A Pioneer’s View of the History of the Field.” In *In the Beginning: Personal Recollections of Software Pioneers*. Los Alamitos, CA: IEEE Computer Society Press.

事实 16	虽然每个人都认为大规模复用（组件）非常重要、非常急需，但是这个问题至今还没有基本解决。

讨论

构建小规模的可复用组件是一回事，构建大规模的可复用组件是完全不同的另一回事。在事实 15 中，我们追溯到 40 多年以前，就解决了小规模复用问题。但是，在此后这么长的时间内，却未解决大规模复用的问题。

为什么？是因为对此问题有许多不同的观点。我将在接下来的“争议”一节中讨论“为什么”这个问题。

但是，理解这个问题的关键词是有用性（useful）。构建通用的、可复用的子程序并不很困难。构建相应的专用子程序却难得多——有人说要难 3 倍（在事实 18 中，将讨论这个问题）——但是这也不是做不到。问题是，构建的这些子程序应该满足不同程序员的各种需求。

这就是症结所在。从有关的争议中（至少从多个视角），我们看到不同的问题需要不同的组件集，这些需要千变万化，结果是（至少在目前）大规模复用不可行。

争议

有关大规模复用的问题有很多的争议。首先，有人提倡大规模复用是软件的发展方向，将来的软件就是对已有组件的组合（称之为基于组件的软件工程）。其他人，特别是更好理解软件业的人（这样说并没有偏向），蔑视这个观点，他们认为几乎不可能抽象出足够的功能，并在此基础上开发专用的、适合当前问题的组件。

为了解决这个争议，我们又转到软件多样性的问题。如果在不同的项目，甚至应用领域中有许多共同的问题，那么基于组件的编程最终将盛行起来；正如许多人猜测的那样，如果软件的应用和领域的多样性意味着所有的问题都不相同，那么只能对一般应用场合的函数和任务进行归纳，在典型的程序代码中这些东西仅占很小的比例。

有一组数据能说明该问题。NASA-Goddard 在其软件工程实验室（Software Engineering Laboratory，SEL）对软件现象开展了多年的研究，该实验室致力于一个特定的领域——航空动力学软件。研究发现，程序中的 70%多可以通过复用模块来构建。然而，SEL 认为复用可行的原因是严格限定了应用领域，他们并不奢望在更大领域中也同样成功。

其次，在业界对于大规模复用未能普及的原因也有争议。许多人，特别是研究人员，认为软件实践者很固执，采用 NIH 方式使他们忽视了别人的工作成果。许多人认为 NIH 的问题往往在于管理，解决之道也在于管理。根据这个观点，大规模复用的问题在于主观意愿，而不在于客观技术。这是管理者的任务，这些人认为应当建立制度和规程来鼓励复用意愿。

实际上，极少有人认为复用存在技术问题。虽然大家普遍承认构建一个通用

的复用库非常难，但是肯定有人能够完成这项工作。

我的观点不同于 NIH 和“意愿而非技术”。我认为这个问题非常难。正如前面所说的，因为问题之间差异很大，所以建立一个适用于多个应用的组件非常难，更不用说适用于多领域的通用组件。我持上述观点的理由是：多年来我致力于将小规模复用发展为大规模复用，我曾经尝试构建与 Share 库中子程序同样广泛使用的大规模复用组件。我逐渐意识到这件事非常难，而当前很少有人意识到这一点。例如，我知道通用报表生成器是信息系统领域的一个重要工具，于是就尝试着在科学技术领域构建相似的东西。虽然花费了好几个月的工夫，却没有找到在科学技术报表生成的共性，所以无法确定该组件的定义，更不用说构建该组件。

在我看来，大规模复用的失败还将继续，其原因不在于 NIH，也不在于主观意愿，更不是技术问题。这个问题的根源是软件的多样性，这个根本问题实在难以解决。

当然，大家都期望我的观点是错误的。显然，我没错。装配组件是一种很有趣的软件构建方法，根据需求规格说明自动生成代码也同样是一种很有趣的软件构建方法。但是，在我看来这些都是不实际的。

来源

虽然有关大规模复用的材料很多，但是多数人都认为这个问题可以解决。

正如前面所说的，有些盲目乐观的人认为这一问题是管理问题，通过改善管理方法可以激发复用的意愿。近期的两个来源如下：

- IEEE Standard 1517. “Standard for Information Technology—Software Life Cycle Processes—Reuse Processes; 1999. ”一个由 IEEE 技术人员制定的、鼓励构建可复用组件的标准。
- MaClure, Carma. 2001. *Software Reuse—A Standards-Based Guide*. Los Alamitos, CA: IEEE Computer Society Press. 有关如何使用 IEEE 的指导材料。

近年来，有些作者对于复用的看法比较接近实际，其中 Ted Biggerstaff、 Will Tracz 和 Don Reifer 所写的东西值得一读。

- Reifer, Donald J.1997. *Practical Software Reuse*. New York: John Wiley and Sons.
- Tracz, Will. 1995. *Confessions of a Used Program Salesman: Institutionalizing Reuse*. Reading, MA: Addison-Wesley.

事实 17	大规模复用最好适用于相关的系统，也就是依赖于具体应用领域，这样就限制了它的应用范围。

讨论

大规模复用即使是可能的，也很难。是否存在提高其可行性的方法？

确实存在。虽然我们不可能建立适用于多个应用领域的组件，但是建立某个领域内的组件却容易得多。SEL 构建航空动力学软件的经验相当鼓舞人心。

软件界同仁常说应用“类”、“产品线”和“特定系列体系结构”。大规模复用对于这些人来说更现实。如果要想成功，则必须应用于解决同类问题的程序，例如：工资管理程序、人力资源管理程序、雷达数据处理程序、仓储管理程序、空间轨道计算程序。请注意，这些可以大规模复用的应用领域是很受限的。

在一个更小的特定应用领域中采用大规模复用的方法，成功的概率就比较大。而在跨项目和跨应用领域中采用大规模复用方法的成功概率很小（McBreen 2002）。

争议

有关这个特定事实的争议主要存在于那些不愿意放弃通用大规模复用幻想的人。其中的一些人是销售大规模复用支持产品的供应商，另一些人是对于应用领域不甚了解的学者，他们认为单一领域内的复用没有必要。后者的哲学与通用工具和方法联系密切，认为无论致力于什么领域，软件的组件都是一样的。他们错了。

来源

近期出版了很多有关各软件产品类和产品结构的书籍。这说明许多人开始理解该事实，并逐步形成了支持该事实的群体。最近有两本讨论特定领域内的大规模复用问题的书籍：

- Bosch, Jan. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Boston: Addison-Wesley.
- Jazayeri, Mehdi, Alexander Ran, and Frank van der Linden. 2000. *Software Architecture for Product Families: Principles and Practice*. Boston: Addison-Wesley.

参考文献

➥ McBreen, Pete. 2002. *Software Craftsmanship*. Boston: Addison-Wesley. Says "cross-project reuse is very hard to achieve."

事实 18	有关复用问题，有两个“三倍法则”：（a）构建可复用的组件比使用组件难三倍；（b）在将组件收录到复用库并成为通用组件之前，应该在三个不同的应用中尝试应用该组件。

讨论

数字 3 在复用问题上很常见。在这两个三倍原则中，“三”仅仅来源于经验，但是它们很恰当、便于记忆、很真实。

第一条原则涉及到构建可复用组件的工作量。正如我们在前面看到的，构建可复用组件是一项非常复杂的工作。通常，构建者考虑将要解决的特定问题，确定是否存在与这个问题相似的通用问题。当然，可复用组件应当能解决通用问题，这样也就解决了原来的特定问题。

不仅组件本身是通用的，而且还应当采用通用的方法来测试组件。因此，构建一个可复用组件的复杂性存在于生命周期的各环节：需求——“什么是通用问题？”、设计——“如何解决通用问题?”、编码、测试。也就是说，从头到尾都很难。

难怪，明白复用的专家说复用需要三倍的实现时间。还有必要指出，虽然大多数人能够采用通用的方法来思考问题，但是解决当前的问题却需要另一种思路。许多人提倡采用特定的专家知识系统来帮助实现。

第二条缘于经验的法则涉及到如何确保可复用组件是通用组件。仅仅解决了当前的问题还不够，还应该能解决一些相关的其他问题，包括在开发组件时没有想到的应用问题。再重复一遍，数字“三”（在三种不同的问题中试用该组件）只是随口说的。我猜，至少三个。也就是说，我建议至少在三种不同的应用中试用你的组件，才可以确切地说该组件是通用的。

争议

该事实提出两条缘于经验的“三倍法则”。很少有人对这些法则提出质疑。大

家都承认：开发可复用组件确实比单任务组件更困难，所需的验证也更多。可能有人对数字三有所异议，但是这仅仅是一个基于经验的数字，谁都很难确定准确的数字。

来源

多年来，大家都熟知该事实，称之为“Biggerstaff 三倍法则（Biggerstaff's Rules of Three）”。Ted Biggerstaff 在二十世纪六七十年代发表的一篇论文中首先提出了复用的三倍法则。不幸的是，时间间隔太久远了，我在因特网上多次查找也没找到。但是在参考文献中，我提到了有关 Biggerstaff 的研究。

因为有很特殊的原因，我确实牢记这些源于经验的法则和 Biggerstaff。当 Biggerstaff 的那篇论文发表时，我正致力于开发一个通用的商业报表生成器（前面已提到）。我当时的任务是开发三个报表生成器。因为我此前从未做过报表生成程序，所以我非常认真地考虑这个问题。

第一个生成器的开发工作进展非常缓慢，当时我一直认为所有的问题都很特殊。对一列数据求和、对和求和、对和的和求和，这其中有很多有趣的问题，它们与我所习惯的科学领域中的问题截然不同。

第二个程序进展也不快，原因是我开始认识到这三个程序在很大程度上相同，这时我想到可能会存在一个通用的解决方案。

第三个程序的进展非常顺利，我在第二个问题中完成的（同时回顾第一个问题）通用方法非常有效。第三次编程不仅按照需求完成了第三个报表生成器，而且也形成了一个通用的报表生成器（我称之为 JARGON。这个缩写的由来比较复杂，我解释一下。我当时工作的公司叫 Aerojet，我们采用的操作系统是 Nimble。JARGON 代表 Jeneralized Aerojet Report Generator on Nimble）。

现在，我认为：要想得出的通用方案就必须全面考虑三个特定的问题。实际上，我形成了一个观点，即：形成通用解决方案的惟一合理方法是针对同一个问题的三个版本形成三种方案。如果你阅读过 Biggerstaff 的文章就会明白，为什么这么多年来我时刻牢记着它。

不幸的是，我无法验证第一条法则，即需要花费三倍的时间。但是我可以肯定，在完成 JARGO 时，我所花的精力远远大于一个特定的报表生成器。我认为本文中的数字三很可信。

➥ Biggerstaff, Ted, and Alan J. Perlis, eds.1989. *Software Reusability*. New

York: ACM Press.

➥ Tracz,Will.1995. *Confessions of a Used Program Salesman: Institutionalizing Reuse.* Reading, MA: Addison-Wesley.

事实 19	修改复用的代码特别容易引起错误。如果一个组件中超过 20%～25%的代码需要修改，那么重新实现的效率会更高。

讨论

除了系列应用以外的大规模复用，即使有可能也非常难，这主要因为所面临问题的多样性。如果这样，为什么不能稍微调整一下大规模复用的理念？除了原封不动地复用组件的方法之外，难道我们就不能修改组件来适应当前的问题？通过适当修改组件，我们可以让它们适合于各种问题，甚至不相干的系列应用。

事实证明，这种想法同样不可行。因为构建和维护大型的软件系统非常复杂（在后面的事实中将讨论），修改现有的软件也非常困难。通常，特定软件系统被限定于特定的设计架构（即确定和限制解决方案的框架）和设计思想（实现同一个软件解决方案，不同的人可能会采用截然不同的方法）。除非修改者理解软件体系结构、接受设计思想，否则很难成功地修改原有的软件。

况且，设计架构通常与对应的原始问题非常合适，但是却会限制那些不适合这种架构的应用问题，比如要求跨领域复用组件的应用问题（注意，这是极限编程方法的基本问题之一，即：尽早建立简单的解决方案，随后再修改原来的解决方案，这种做法的难度其实非常大）。

修改已有软件除有难度外还有一个根本问题。研究软件维护的人发现：在维护过程中，有一个任务的难度超过了修改软件时的其他所有任务。该任务就是“理解现有的解决方案”。软件中有一个众所周知的现象，那就是：经过一段时间以后，即使开发者自己也难以修改。

为了解决这些问题，软件人员发明了维护文档的思想，维护文档就是描述程序工作方式和工作原理的文档。通常，这些文档开始于原始的软件设计文档，并在此基础上建立。但是，这样我们又涉及另一种软件现象。虽然大家都认为维护文档很重要，但是一旦软件项目出现成本或时间进度的问题，就会将它当作垃圾而忽略。因此，保留完整维护文档的软件系统几乎不存在。

更糟糕的是，在维护过程中，虽然修改了软件（正如事实 42 中所说的，这种修改是软件业的主要活动），但是很少修改维护文档，即使有修改，也明显滞后，所以它不可信。因为这些原因，绝大多数软件维护通常从阅读代码开始。

我们回到一个基本问题——修改软件非常难。因为时间表、成本等老问题，我们没有适用软件维护的有效措施，即使采取了一些方法，也是不当的方法。这是一个两难的问题。除非我们找到其他的管理方法，否则难以改变这种两难的局面。

对于修改软件组件，下面是与本事实相关的一个必然结论：

修改经过打包的商业性软件系统通常是错误的。

理由是修改非常难，我们已经谈过这一点了。但是，还有一点原因。**商业性**软件通常有多个发行版本。供应商根据客户的需要，在新版本中解决旧版本的问题、增加新的功能或兼而有之，客户通常也希望获得这样的新版本（实际上，经过一段时间之后，供应商就停止对旧版本的维护，这样客户只能升级到新版本）。

自行修改软件包的问题在于每个发行版都需要修改。如果供应商显著修改了软件的实现方法，那么就需要重新设计修改方案以适应新的版本需求。因此，修改已有的软件包是一项永无止境的工作，每一次新版本的使用都需要耗费开发成本，而且经费开支也非常大。对同一个软件进行反复修改，软件人员对此最憎恶了。精神和经济的重复付出使人们普遍接受了上述的必然结论。

其实，这个结论并不新奇。早在 20 世纪 60 年代，人们就认识到修改商业性软件是一项长期的、痛苦的工作，因此不采用修改的做法。不幸的是，与本书中讨论的其他事实一样，我们仍然在反复地接受教训。

我做过一些有关维护企业资源规划（Enterprise Resource Planning，ERP）（例如 SAP）的研究，发现有些用户自行修改了 ERP 软件。他们都是在半途而废时，才意识到这项工作的难度。

注意，在开源软件运动中，这个问题有了变数。拿到开放性源代码很容易，但是修改却很难，除非你修改后的一个版本成为系统的一个新的分支，与原来的标准系统并行发展。我从未听到开源的倡导者讨论这个问题（当然，还有一种解决方法，那就是让该软件的主要负责人将你的修改纳入标准软件之中。但是，这一点很难保证）。

争议

要接受这一事实，应该首先接受另一个事实，即：软件产品很难构建和维护。软件实践者通常会接受这一事实。不幸的是，有些人（特别是从未构建过实用软件的人）认为构建和维护软件解决方案都很容易。有些人从未见识过大型软件解决方案，他们可能只解决了一些琐碎的问题（许多学者和学生通常如此），或者只是在一些软件课程中接触过难度相当于“hello world”的问题，所以会产生这种看法。

由于这种看法异常顽固，许多人都不愿意接受“修改现有软件非常困难”这一事实。因此，这些人仍然坚信修改是解决大规模软件复用多样性问题的方法（对于商业性软件包的修改也一样）。这些人已经无可救药了，尽量不要考虑他们了。

来源

有关软件错误和软件成本的研究都证实了这一基本事实。本书中反复提到了NASA-Goddard 所属的 SEL。SEL 曾详细比较了修改旧代码和完全重写所需的成本问题（MaGarry 等，1984；Thomas 1997）。他们的结果清晰明确，给人深刻的印象。如果现有软件系统的 20%～25%以上需要修改，那么从头构建新产品更便宜、更容易。这个百分比低的惊人。

我们回想 SEL 曾致力于一个非常特定的领域——航空动力学。你可能想起 SEL 成功地采用的大规模软件复用解决了其领域中的问题。有人会因为这个特定的领域而怀疑他们的研究结果。但是我认为应该接受他们的结果，理由是：（a）SEL 在研究这个（和其他）问题时非常客观；（b）SEL 具有采用各种方法实现大规模复用的强烈动机；（c）我的经验表明，成功地修改别人的软件非常困难。更不要说 Fred Brooks[1995]的著名论断“软件工作是人类所从事的最复杂的工作。”

参考文献

- Brooks, Frederick P., Jr.1995. *The Mythical Man-Mouth.* Anniversary ed. Reading, MA: Addison-Wesley.
- McGarry, F., G. Page, D. Card, et al. 1984. “An Approach to Software Cost Estimation. ”NASA Software Engineering Laboratory, SEL-83-001（Feb.）.This study found the figure to be 20 percent.

➥ Thomas, William, Alex Delis, and Victor R. Basili. 1997. "An Analysis of Errors in a Reuse-Oriented Development Environment." *Journal of Systems and Software*, 38, no. 3. This study reports the 25 percent figure.

事实 20	设计模式复用是解决代码复用中固有问题的一种方法。

讨论

到目前为止，有关复用的讨论都相当让人失望。小规模复用已经有超过45 年的历史了，其中的问题已经得到很好的解决。大规模复用问题似乎难以解决，只不过对于相似的问题有一点希望。修改可复用组件通常非常困难，不是一种非常好的方法。那么，当程序员面对一个新问题时，如何能避免彻底从头再来？

软件实践者常用的方法之一是根据以往问题的处理方法来解决当前的问题。在软件有价值之前（20 世纪 70 年代），他们经常将一项工作的代码带入另一项工作。从那时起，不同公司之间签订软件供货合同，有关法律开始禁止随意复用代码。

当然，无论采用什么方法，程序员还会这样做。他们可以采用记忆中的原有解决方案，实际上也有人用纸或磁盘记录原有解决方案。但是，我们实在是太需要采用过去的方案来解决当前的问题，所以无法停止这种做法。我身为一个法律顾问，偶然会有人向我询问这种做法的后果。

移植通常不是原封不动地照搬照抄原来的代码。我们在新问题中体现了原来代码中的设计思想。在二十多年前的一次学术会议上，Visser（1987）的报告提出许多实践者熟知的一个观点，即“设计很少从头开始”。

我们在这里讨论的是软件复用的另一个层次。我们已经讨论了代码复用，现在我们讨论设计模式复用。设计复用在 20 世纪 90 年代蓬勃发展。其实它与软件本身一样久远，但是被包装成“设计模式”之后，其适用性似乎激增，并受到了极大关注。1995 年 Gamma 在一本书中很精彩地定义和描述了设计模式，从此，设计模式受到了实践者和学者的一致认可。

什么是设计模式？设计模式是对反复出现的问题以及该问题的解决方案的一种描述。模式包含 4 项基本内容：模式名称、解决方案的适用条件、解决方案、

使用后果。

为什么业界这么快就接受了模式？实践者认为模式就是他们过去通常采用的方案，只不过披上了新的外衣，摇身变成尊贵之物。学者认为设计复用中涉及到了设计，比代码复用更抽象、更有概念性，所以是一个更有意思的概念。

虽然模式如此可人，但是并没有显著地改变软件实践。这大概有以下两方面的原因：

1. 正如我前面所说的，实践者早已采用了这种做法。
2. 在一开始时，那些公开出版的模式只是所谓的常规（基本的、与领域无关的）模式。人们逐渐认识到领域相关的模式的重要性，并正在逐渐完善。

这个事实引出一个非常有意思的结论：

设计模式源于实践，而不是源于理论。

Gamma 和同事（1995）承认实践的重要作用，他说到：“本书中的所有模式都不是新的、未经证明的……[它们]已经在不同的系统中多次用到”，“聪明的设计者复用那些已经用过的方案”。这一有趣的例子证明理论源于实践。在实践中形成了模式的概念，并神话般成功。理论界看到了这些，便很快建立了有关模式的理论框架，并采用了新的、更有效的方式来建立各种模式文档。

争议

设计模式的概念被广泛接受。有一群狂热的学者正在研究模式这一不断拓展的领域。因为这些人使模式更组织化、条理化，同时还提出了一些实践者不熟悉的新模式，所以实践者也重视他们的工作。

然而，衡量这些工作对于实践的影响却很难。有多少典型的应用软件构建在熟悉的模式之上？我还没有听说有关这个问题的研究。有人说到滥用模式（将模式强行引入并不适合的应用中）可能导致“莫名其妙的……代码，生产者在商品表面加上无意义的装饰”。

还有，因为没有人怀疑这些工作的价值，所以可以很有把握地说设计模式是软件业中深入人心的、被遗忘最少的真理之一。

来源

近些年来，出版了很多有关模式的书籍。这类书基本上都不错。你所读的有

关模式的任何东西似乎都有用。大多数有关模式的书实际上只是将一些常用事项归类成册。有关模式的最重要的、同时也是先驱性的经典书籍是 Gamma 等人的著作，该书的作者被称为著名的“Gang of Four”。详见随后所列的参考文献。

参考文献

- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns*. Reading, MA: Addison-Wesley.
- Visser, Willemien. 1987. “Strategies in Programming Programmable Controllers: A Field Study of a Professional Programmer.” Proceedings of the Empirical Studies of Programmers: Second Workshop. Ablex Publishing Corp.

1.5 复杂性

事实 21	问题的复杂性每增加 25%，解决方案的复杂性就增加 100%。这不是一个可改变的条件（即使人们都努力降低复杂性），而是客观存在的。

讨论

这是我喜欢的事实之一，理由是它鲜为人知，所以在此非常重要，必须解释清楚。我们已经知道构建和维护软件都很难，下面来分析其缘由。

该事实同时还解释了本书中的其他事实。

- 为什么人员如此重要？（因为需要采用相当的智力和技巧来克服复杂性。）
- 为什么难以估计？（因为我们的解决方案比问题复杂得多。）
- 为什么大规模复用不成功？（因为复杂性加大了多样性。）
- 为什么会出现需求爆炸问题（当我们从需求阶段转到设计阶段时，为了做出可行的设计，必须由显式需求激增出许多隐含需求）？（因为我们从 25%的工作区转入 100%的工作区。）
- 为什么同一个问题有多个正确的设计方法？（因为问题的解决比问题本身复杂。）
- 为什么优秀的设计员会采用迭代的、启发式的方法？（因为几乎没有简单的、显而易见的设计方法。）

- 为什么很少对设计进行优化？（因为设计非常复杂，几乎无法优化。）
- 为什么100%的测试覆盖几乎不可能，而且，无论如何测试总是不够？（因为许多程序中有无数的路径，此外，测试覆盖不能追踪到因为软件复杂性所导致的所有问题。）
- 为什么审查是效率最高、效果最显著的消除错误的方法？（因为需要一个人来过滤这些复杂性，以便找出错误。）
- 为什么软件维护非常费时？（因为在开始时，几乎不可能确定一个解决方案的所有输出结果。）
- 为什么“理解现有的产品”是软件维护过程中最主要，也是最难的任务？（因为解决任何问题，可能存在的正确解决方案都太多了。）
- 为什么软件中有许多错误？（因为在第一次就做好太难了。）
- 为什么软件学者采用鼓吹的方法？（也许是因为在软件领域，开展那些急需的评估性研究非常困难，所以未经研究就宣传。）

哇！直到开始列这个清单时，我才意识到该事实何等重要。如果忘了本书的其他内容，也请你牢记：问题的复杂性每增加 25%，解决方案的复杂性就增加100%。而且，不存在解决这个问题的“银弹”。软件解决方案复杂性的根源是问题本身固有的多样性。

争议

知道这一事实的人并不多。我想，假如知道的人多了，便会有人怀疑其真实性，有些人（特别是那些认为软件的解决方案很容易的人）会说解决方案的复杂性并不是固有的，而是由不称职的程序员造成的。

来源

➥ Woodfield, Scott N. 1979. “An Experiment on Unit Increase in Problem Complexity.”*IEEE Transactions on Software Engineering*. 我找这本书所用的时间比其他书都多。我查阅了自己以前的课程笔记和书籍（我确信在其他地方也引用过它），使用搜索引擎，给许多同事发邮件。我想自己可能惹恼了其中的一些人（他们都没有听说过这本书，但是都说但愿自己听说过）。最后是 IEEE 的 Dennis Taylor 找到正确的引用，Maryland 大学的 Vic Basili 给我一份复印件。在此深表感谢！

事实 22	80%的软件工作是智力活动。相当大的比例是创造性的活动。很少是文书性的工作。

讨论

经过多年，有一个争议已经白热化，即，是否软件工作无足轻重，可以自动完成，还是说大多数复杂任务是由人完成的。

在无足轻重/自动生成的阵营中，有《Programming without Programmer》、《CASE——The Automation of Software》等名著的作者和学者，他们试图或者倡导根据需求规格说明书自动生成代码。在“最复杂”的阵营中，有 Fred Brooks 和 David Parnas 等著名的软件工程师，虽然这些观点之间差异非常大，但是对于这一关键的问题却很少有人客观分析。似乎每个人在很早以前已经选定了阵营，认为没有必要再通过研究来验证自己的信仰。而该事实恰好是与此争议有关的研究（顺便提一下，这也是业内另一个激烈争议的很好例证，该争议是：在计算机研究方面，精确和适度哪个更重要？讨论完第一个争议后，我们再谈第二个）。

你如何确定计算工作是无足轻重/自动化，还是异常复杂？至少在本文中，该问题的答案是研究工作中的程序员。在系统分析员执行系统分析（需求定义）任务时，我对他们进行录像。他们正坐在一张桌子旁分析问题描述。我是主管该项目的研究者，看这些录像非常有意思，也很乏味。系统分析员在很多时候几乎不干任何事情（很乏味），然后他们经常会写些东西（这也很乏味，但这说明有趣的东西即将开始。）

我反复观察这种工作方式，很明显，分析员在无所事事地静坐时，他们正在思考；他们写下什么东西时，是记录思考的结果。稍微多想一点，很明显，考虑时间代表工作中的智力活动的时间，而写东西的时间代表文书性的工作。

现在事情变得有意思了。将多个分析员的录像进行分析，很快发现如下模式：分析员 80%的时间在思考，20%的时间作记录。换句话说，80%的分析任务是智力活动（至少录像中的情景如此），其余 20%是文书活动。这一发现在多个项目中也同样正确。

下面我们花点时间来讨论精确/适度的问题。你可以想像这个试验不是一个复杂的研究过程。从一个学者的角度看，这不够精确。只是大略地说！关于这个问题，我不能想出一个更合适的研究。然而，在这项研究中的一个同事告诉我说最

好再精确一点。我们决定在这项研究上再增加一项内容。原有研究的一项不足是只涉及系统分析阶段，而不是软件开发的全过程。其次，这只是一个经验化的东西，研究结果依赖于临时找来的几个观察对象。

第二阶段的工作解决了这些问题。我们决定观察不同的开发阶段。我们选择各项任务，并确定参与者进行智力活动和文书活动的时间。

现在，问题变得非常奇怪。分类结果表明，80%的软件开发工作是智力活动，其余 20%是文书活动——这个 80/20 的比例与分析阶段相同。

这两个 80/20 的比例未必准确。这两项研究采用不同的方法研究了不同的问题。80/20 很可能是偶然结果，未必确切。即使这不够精确，至少也是一个适度的结果。可以说在软件开发过程中，通常有大约 80%的工作是智力活动，20%是文书性工作。我可以肯定，这说明了在无足轻重/自动生成和异常复杂的争议中一些非常重要的东西。也就是说，文书性的工作无足轻重，可以自动完成，但是智力性工作却不能。

有关这一故事，还有一些补充。这一研究最终延伸到分析软件开发过程的创新性（不仅仅是智力性）测试中。在上一个研究的第二方面，我们从智力活动中分出创新性的活动。继第一阶段的 80/20 之后，我们希望有关软件开发过程创新性研究的活动，也获得类似喜人的结果。

我们有点失望。我们的第一个问题是给创新性下一个有用的、可行的定义。我们找到一个有关创新性的文献，这样我们就可以工作了。欣慰的是，我们研究发现，大约 16%的工作归类于创新性工作；不幸的是，不同的分类者之间的差异很大，有一个人认为仅 6%的工作是创新性的工作，而另一个人则认为是 29%。无论如何，可以比较肯定的说，在软件开发过程中智力性活动比文书性活动占的比例要大得多，其中至少有相当的比例（但是比例不大）是创新性的。至少在我看来，软件开发工作相当复杂，根本不是无足轻重，或者是可以自动生成的。

争议

因为我们在“讨论”一节中，谈到了两个争议，所以剩下的争议就不多了。至少我认为上面的研究已经解决了第一个争议——很明显，构建软件的过程非常复杂，不是无足轻重的。我想这同时也是一个很好的例子，说明为什么精确的研究还不够。如果一定要我在精确的但是不适度的研究，和适度的但是不精确的研究中做出选择，我通常会选择适度作为主要目标。当然，这是实践者的观点。真

正学者的观点与此差异很大。

虽然，我从这些研究中得到深刻的结论，但是我不得不承认这两个争议依然会很激烈。很可能任何一个都不会（或者也不应该）解决。

实际上，在第一个争议中，持无足轻重/自动生成观点的一些人的思想已经有所变化。面向对象编程“三人帮”（这三个人组建了 Rational Software 公司，创造了 UML 面向对象的建模方法）之一的 Jacobson（2002）认为许多软件开发工作是“例程”（他在一篇论文中分析敏捷软件过程和 UML 方法之间的关系时，提出这一结论）。他说，80%是例程，20%的创造性活动来自于“与同事讨论和我个人的经验。”显然，他所说的 20%与本事实吻合，但是 80%的例程与本事实不吻合。注意，Jacobson 没有考虑中间类型，“智力性活动”是介于创造性活动和例程之间的重要活动。

来源

有关智力/文书和创造性/智力/文书的研究很多，但是下面一本书同时讨论了这两个问题：

- Glass, Robert L. 1995. *Software Creativity*. Section 2.6, “Intellectual vs. Clerical Tasks.” Englewood Cliffs, NJ: Prentice Hall.

参考文献

- Jacobson, Ivar. 2002. “A Resounding ‘yes’ to Agile Processes, but Also to More.” *Cutter IT Journal*, Jan.

第2章

生 命 周 期

软件生命周期（software life cycle）指讨论软件构建过程的组织方法。但事实通常不是这样。虽然这一定义本身非常直观，但是人们对生命周期的表述很快体现出个人的信念。对于那些有倾向的人而言，生命周期变成对软件开发过程和工作顺序的严格定义。人们将这种严格的生命周期称为瀑布型生命周期。瀑布模型是一个单向下行的过程。人们还构建了相关的方法。瀑布模型在一些软件社区越流行，其他人就越意识到其错误所在。

有关瀑布模型的错误在于它将非常复杂的过程描述得相当简单。真正明白软件构建的人都知道，优秀的软件人员不会按照精确的顺序执行这些精确的步骤。这些步骤都很好，但是其顺序有问题。

我们退一步来定义什么是生命周期。生命周期开始于需求的定义和开发，在这一阶段，定义和分析“什么”问题。接着是设计，在这一阶段确定如何解决问题。然后是编码，将设计转化为计算机上可运行的代码。随后，因为整个过程中极易出现错误，所以进行错误消除。最终，完成了全部测试之后，软件产品交付使用，便开始了维护。

现在，我们回过头来讨论瀑布模型生命周期中的问题。倡导者认为应该按照严格的顺序来构建软件。你先彻底完成需求定义，然后才可以着手做设计。只有彻底完成了设计才可以编码。编码完成之后才测试。当然，测试完成之后才会发布产品，才可以开始做维护。

这些听起来都很合理，但是实际上，软件开发高手很早以前就知道采用“构建一小块，测试一小块”的方法。在软件开发过程中，需求不断地变化。设计者需要用一小段代码来验证概念性设计是否可以真正转化为可行的程序。当然，需要测试这一小段代码来确保试验性的代码和方案是否真正可行。高手深知瀑布模型永远是一个不可及的理想状态。即使高级管理人员按照瀑布模型来管理，他们

也一如既往地工作(因为瀑布模型比真正软件开发过程中的混乱局面更易于管理，所以高级管理人员偏好于瀑布模型)。

这种愚蠢的行为持续了大约十年，在这期间，高手们忽视管理命令，采用不得以为之的方法来构建软件，同时也有人逐渐清醒。学术界开始提到所谓螺旋形的生命周期，它准确而生动地描述了高手们的做法。很快，在软件界的很多领域，瀑布模型消失，螺旋模型取而代之。

但是，在整个过程中，有一件事一直有效，即这些步骤本身。我们现在会说自己在需求、设计、编码、错误消除和维护等活动中螺旋式、迭代前进，但实际上，我们仍然做了全部工作，只不过是将原来的严格顺序重新调整。

本章将按照上述顺序展开，逐一讨论生命周期中的各步骤。对于每一步骤，我们将提出一些容易被忘记的基本事实。

- 有关需求的事实。还记得我们曾说过需求经常变化吗？有关这一现象有一些非常重要的事实，同时还有如何解决有关需求的问题。
- 有关设计的事实。设计可能是软件生命周期中最需要智力、最有创造性的部分，这部分工作体现了软件开发过程真正的复杂性。有关设计的事实也同样复杂。
- 有关编码的事实。编码是软件开发过程中的重点。因为人们对此已有深入理解，在此我们只讨论了较少的事实。
- 有关错误消除的事实。在生命周期的前几个阶段，智力难题和复杂性使我们的产品中存在错误。错误消除的有关质疑——例如几乎不可能构建没有错误的软件程序经由这些事实予以反映。
- 有关测试的事实。测试指通过在软件产品中执行测试数据，来确定软件是否成功运行。虽然测试对于消除错误非常重要，但是几乎不可能开展完全彻底的测试，并找到所有的错误。
- 有关评审和检查的事实。因为测试不可能消除所有的问题，所以必须采用评审和检查等静态方法来补充。但是经过“测试+评审和检查”的软件中仍然会有错误。有关这两项活动的事实很好地说明构建完好的、高品质的软件何等困难。
- 最后是有关维护的事实。在软件周期各环节中，人们对维护的理解最少，但是在某些方面它最重要。这里所列的事实可能是本书中经常被遗忘的事实中最惊人的代表。

下面我们将浏览一遍生命周期，不用担心会陷入瀑布模型。

2.1 需求

事实 23	导致项目失控的两个最常见原因之一是不稳定的需求（另一个见事实 8 所说的项目估算失误）。

讨论

如事实 8 所说的，失控项目是指无法控制的项目。在软件业中有许多的失控项目。我想，虽然没有那些相信“软件危机”的人所说的那么多，但是确实很多。

通常，失控的项目在一开始就已经失控了。在事实 8 中已经讲过这些。糟糕的或者说理想化的估算认为，构建软件所需的时间和金钱比实际所需的要少得多，这是导致软件失控的一个重要原因。使得这些项目被定义了不可能完成的目标，所以在一开始就已经失控了，它们的失败是估算的失败，而不是软件开发的失败。

另一方面，不稳定的需求（理想化估算邪恶的孪生兄弟）似乎是导致项目失控的更复杂的原因。导致这个问题的原因是软件的客户和用户对于所要解决的问题并不真正清楚。开始时他们可能认为自己知道，但是在项目的进展过程中，他们发现原来对问题的认识过于简单，或者不现实，或者根本就不是自己所希望的。也许他们在一开始时就一无所知，只是探寻解决一个模糊的问题。

你会想像，无论哪种情形都会使软件开发团队陷入困境。解决一个确定的问题都非常困难，那么解决一个需求不断变化的问题几乎不可能。难怪这类需求经常会导致软件失控。同时，对需求的糟糕的理解也很常见。毕竟，采用软件的方法来解决问题也不过 50 年的历史，我们必须面对各种各样非常复杂的问题，其中许多问题在一二十年以前想都不敢想。

看一下软件业都曾经采取过哪些方法来应对不稳定的需求。最早，许多业内人士认为问题产生的原因在于温和的软件管理，解决方法是在原始的需求下面画一个终止符，软件团队只解决这些问题，客户和用户也只能接受相应的产品。在这一时期计算机学者提出格式化的需求规格说明的概念，用一个非常精确的方法来代表用户需求。

当然，这一方法根本无效。最终的产品并没有解决客户和用户真正希望解决的问题，所以只能被忽略和遗弃。花费了时间和金钱来构建的软件被拒绝。那些华丽的、死板的、精确的需求规格说明也同样被拒绝。客户和开发者的关系通常如此。

开发者一旦开始认识到自己不得不允许需求变化，他们就尝试采用各种完全不同的解决方法。如果该问题需要采用探索的方法来确定需求，那么通常会采用原型的方法。根据这一思想，我们通常会构建一个简单的产品，让用户试用并确定真正的需求（关于采用原型的开发方法，有许多教训，但是我们在此只讨论在需求定义方面的应用）。随着用户越来越强烈地参与，我们发明了联合应用开发（Joint [with the users] Application Development，JAD）的方法，这种方法配合原型方法使用。（我始终不理解这里的 Development 的含义，我觉得应该称为 JARR，即 Joint Application Requirements Resolution）原型法和 JAD 在今天仍然被采用，特别在需求难以理解的时候更有用。

同时，在管理中应该解决如何应对需求不稳定和“变化的目标”的问题。起初，当需求变化时不采取任何管理手段。但是，管理者最终认识到：虽然自己不能冻结需求，但是他们可以强调改变需求意味着改变项目的条件。“你需要新的或者变更后的功能吗？那么我们讨论一下这会使时间和成本在原始估算上增加多少。”不幸的是，这使我们陷入了困境，即在项目进展中间改变估算，我们在事实 11 中已经讨论过这个问题，但是修改估算是应对需求变化的惟一方法，看来只能推翻事实 11。

我们继续探讨软件领域中不稳定的、不断发展的需求的问题。经过前面的讨论，我们已经知道如何处理它，但是采用这些方法的成功经验不多。而且，随着软件领域面对更富有挑战性和多变性的项目，需求变化更为普遍。在许多著名的失控项目中，用户和开发者使得需求变化无法控制，以至于无法开发出解决任何问题的产品（有关于此的一个典型例子是丹佛国际机场行李自动处理系统[Glass 1998]，在该系统中需求发生了彻底的变化，以至于所有的工作都受到影响。时至今日，原来预想的庞大的行李处理系统仍未建成，在丹佛机场的所有航线，除美联航以外都采用手工处理系统）。

有关不稳定需求的问题，有了一个新的转折性的进展。极限编程的轻量级方法建议在开发期间将用户代表吸纳到软件开发团队中来。客户代表的持续存在显然有助于尽早确定和解决无效的、变化的需求。然而，还有一些问题，有多少用户组织（a）能提供这样的专职人员；（b）一个人能否代表所有其他潜在客户和用户的不同观点。

争议

原来有很多人认为应该严格冻结需求，现在这个争议已经很少了。几乎所有的人都接受需求必须允许变化这个事实，惟一有争议的是在当前形势下如何管理。

有关格式化需求规格说明也有一些争议。在实践中很少使用那种精确的方法，但是学术界仍然倡导和教授。这又是一个实践和理论缺少沟通的地带。理论者认为实践者不妥协，而实践者认为这个理论白费时间。这个问题根源在于过去理论者和实践者的沟通问题，这个争议很可能在短期内不能解决。

来源

该事实的主要来源与事实 8 相同。

- Cole, Andy.1995. "Runaway Projects —Causes and Effects." *Software World(UK)* 26, no.3. 这个研究报告发现"项目目标没有完全明确"是大约51%项目失控的罪魁祸首（还有 48%失控项目的主要原因是"项目计划和项目估计很差"）。
- Van Genuchten, Michiel.1991. "Why Is software Late?" *IEEE Transactions on Software Engineering*, June. 该研究结果发现，项目延迟交付的第 2 个原因是"设计/实现经常变化"（原因是需求经常变化），比例大约是 50%（"理想化估计"是首要原因，比例大约是 51%）（有些项目失控的原因可能不止一个）。

参考文献

- Glass Robert L. 1998. *Software Runaways*. Englewood Cliffs, NJ: Prentice Hall. This book tells the story of many runaway projects that suffered from requirements instability, including the Denver Airport Automated Baggage Handling System.

事实 24	在产品完成时修订需求错误的代价最大，在开发早期修订需求错误的代价最小。

讨论

该事实确实是个常识。错误在软件（或者在其他的产品中）的存留时间越长，修订的代价越大（还有什么会比从需求到产品发布更长？）。Boehm和Basili（2001）指出，修订产品中错误的代价是修订开发早期错误成本的100倍。

对于软件而言，该事实要说明的问题确实比较严重。在早期阶段，软件是无形的，除了一些文档性的规格说明以外没有什么具体有形的东西，而且比较容易修改。但随着时间的推移，软件产品越来越详细（在设计阶段），越来越具体（在编码阶段），越来越固定（当代码逐渐接近最终的方案时）。在早期可以轻易改变的东西，越到后期就越难得多。

该事实的内容非常清楚，非常易于理解——在软件生命周期中，尽量早地找出产品中的错误。那么，为什么该事实经常被遗忘？因为，虽然我们都接受这个事实，但是人们并不根据它来工作。尽早找出需求中的错误意味着大量采用需求净化技术，而我们为了追上时间表（这通常不可能）经常顾不得采用这些技术。有哪些需求净化技术？需求一致性检查、需求复审（客户和用户分析和更正误解及遗漏的错误）、需求驱动的早期测试用例设计、坚持采用可测试的需求，以及采用建模、仿真、原型的方法检查早期需求的有效性。如果你信得过的话，还可以采用格式化的需求规格说明书（因为格式化使规范定义更严格）。虽然有多种方法都可以找出需求中的错误，但是一般的软件项目中很少使用上述方法。

争议

对于该事实的实质其实没有争议。正如我们前面所说的，该事实仅仅是一般常识。

惟一的争议是我们该怎么做。几乎每个人都有一种做法。计算机学者会坚持采用格式化的规格说明书技术；开发者则将复审放在首位；测试和质量保证人员要求有可测试的需求，并建立早期测试用例；系统分析员可能会要求采用建模的方法；极限编程者提倡在开发团队中吸纳一个客户代表。

在上一段中，不同参与者的各个观点向我们提示了一些线索。无法统一确定采用哪一种方法来解决这个问题最好。但是我们在几段之前看到一个最重要的提示，这就是我们的老对手——不合理的时间表。谁都不愿意冒险陷入需求“分析瘫痪”（这确实会有问题）。因此，我们疯狂地冲出生命周期的早期阶段，赶上

甚至超过了时间表，一直冲到项目的后期，所得的软件经常不合格，还要无休止地测试它。瞧！我们又错了，花了许多的时间来搜寻从一开始就隐藏在产品中的错误。

来源

有关该事实有很多来源，下面是其中之一，其他请参见“参考文献”一节。

- Davis, Alan M. 1993. *Software Requirements: Objects, Functions, and States*, pp. 25-31. Englewood Cliffs, NJ: Prentice Hall。这个事实是 Davis 这本书的主要观点。

参考文献

- Boehm, Barry, and Victor R.Basili. 2001. “Software Defect Reduction Top 10 List.” *IEEE Computer*, Jan. “Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design stage.”

事实 25	遗漏需求是最难修订的需求错误。

讨论

怎么会遗漏需求呢？因为在需求收集阶段出现了错误。

需求收集要确定有待解决的问题。但是，这提出一个问题：如何收集需求？收集需求通常是（但不总是）人与人的交流。软件开发团队的代表拜访那些有问题需要解决的人——客户、用户或者他们的“业务分析员”。

执行这些拜访任务的人称为系统分析员。系统分析可能是由一个全能程序员或者以前做过系统分析的专家来承担的任务。无论如何，明显可以看出系统分析涉及到许多人与人的交流，是一项很容易出错的活动。不能搜集到重要需求的可能性时刻存在。

系统分析是搜集业务应用领域需求的主要方法。但是还有其他领域和其他的需求搜集方法。在有些应用中，软件是其他更大系统的一部分，需求搜集的过程定位于整个系统，在任何局部软件的需求定义之前完成。这些需求一经形成就写

成规范的文档来描述整个系统。

从事这种需求搜集的人通常被称为系统工程师。因为系统工程师要面对多种问题，所以没有成功的系统工程（规程化）方法，虽然有这方面的课程，但是这仅仅是一个学科名称，没有内容。

现在我们考虑系统工程师所生成的文档，定义一个系统需求的方法之一是认真仔细地阅读系统的文档，从中挑出那些与软件有关的需求。这也是一个极易出错的过程。例如，软件工程师怎么能知道哪些系统需求真正影响到软件？根据我的经验，软件需求收集过程应该（a）迭代（看第一遍时很难确定哪些需求与软件相关），（b）互动（软件需求收集者应当与其他领域的需求收集者交流，适当地分配需求）。

在此，我们看到系统分析也是一项人与人之间的交流活动，容易出现错误。我们还看到系统工程是一个多学科的过程，同样容易出现错误。在这么多易于出错的过程中，出现错误也就不足为奇了。这其中最大的错误是完全遗漏一项需求。

为什么遗漏需求对于解决问题会造成灾难性的影响？因为每项需求都对解决问题的难度水平有所贡献，这些需求之间的关系显著提高了解决方案的复杂性。缺少一项需求可能导致在设计方案时遗漏许多问题。

为什么遗漏需求后很难发现和改正？因为在软件错误消除过程中，许多工作都是需求驱动的。例如，我们定义测试用例来验证解决方案是否满足各项需求。（后面我们将看到需求驱动的测试非常必要，但是远远不是完备的测试方法）。如果缺失了一项需求，那么在需求规格说明书中不会有它，在任何说明书驱动的评审和检查中也不会审查它，况且也没有建立测试用例来验证该需求是否满足，因此各种基本错误消除方法都不能发觉到需求遗漏。

该事实的一个必然结论是：

> 最持久的软件错误是遗漏逻辑错误，它可以逃过软件测试过程，进入发布的产品中。遗漏需求会导致遗漏逻辑。

多年以前，我深刻感受到软件错误消除过程的复杂性，决心研究在软件项目中哪些错误的后果最严重。我认为在这个痛苦的复杂过程中，查找严重错误比查找一般错误的价值更大。

当然，不同的软件项目采用不同的方法来确定错误的严重性顺序。这些严重等级的范围包括从演示停止（show stopper）——可以导致整个系统无法工作——

到微不足道的错误，例如，用户乐意接受系统，错误可以延期修订（这是一个很重要的区别。虽然有人呼吁无错误软件，但是几乎所有的复杂软件都带着一些已知的错误非常成功地运行。例如，NASA 的宇航员可以在一些成功的宇航软件中找出一些已知的错误。近期 Microsoft 发布了系统中的一些错误，其反对者对于如此大量的错误兴奋不已，没有人针对这种局面指出这个区别及其意义）。

但是优先级是旁观者的看法，还是无法说明高级错误的基本共性特征，因此就无法制定有效的方法来克服这些错误。我需要一种更客观的研究方法。

考虑这个问题时，我逐渐认识到无论错误的本质和等级，最严重的错误是进入软件发行版中的错误。我们已经看到，即使产品中的一些错误比其他错误严重得多，这些错误也比发行之前就找出的错误影响更大。因此，我考虑这些持久性的错误有哪些特性？然后，考虑如何找出这些错误。

事实上，找出这些错误很容易。当时，我在一家知名航空公司的软件开发和研究机构工作，能接触到大量软件错误的数据。我根据发现错误的日期找出产品中的错误，然后对错误数据进行归类。

分析数据时，形势逐渐明晰。我发现在所有的持久性错误中，（我所谓的）缺少编码逻辑的错误远远大于其他错误。这是最主要的持久性错误，约占 30%。其次是重复性错误，即在维护过程中修订旧错误时引入新的错误，约占 8.5%，与30%相差很大。很有意思的是，第三类错误根本就不是软件错误，而是文档错误（8%），有人认为对应的软件有问题，但是实际上软件居然没有错。

那么，缺少编码逻辑包括哪些东西？它包括：--个数据项使用后没有正确初始化就再次使用、在条件语句中遗漏了一个或几个条件等等——这类错误的产生原因都是编程者或设计者未能完整、深入地考虑当前的问题。

为什么这些错误能持续到产品的发行版中？因为很难测试不存在的东西。覆盖分析之类的测试方法能帮助我们测试所有的代码片断，确定其是否正常工作。但是，如果一个代码片断不存在，那么覆盖方法也无法检测出其缺失。同样，评审者可以找出眼前代码中的错误，但是也未必能发现代码片断的缺失。

那么，这与遗漏需求有什么关系？显然，遗漏需求会导致遗漏逻辑。同理，发现遗漏需求与发现遗漏逻辑一样难。

争议

该事实及其必然结论讨论有关遗漏的问题。对这些话题的争议也基本上被遗

漏了。没有争议的原因是许多人根本就不曾意识到这些遗漏。这也许是软件界中的一个严重问题。我们因为不能区分自己所犯的各种错误而吃尽了苦头。(注意前面提到的 Microsoft 的例子。设想 NASA 因为航天软件中存在的已知错误所受到的非议。)

来源

该事实的来源之一是:

- Wiegers, Karl E. 2002. *Peer Reviews in Software: A Practical Guide*. Boston: Addison-Wesley.

必然结论的来源是:

- Glass, Robert L. 1981. "Persistent Software Errors." *IEEE Transactions on Software Engineering*, Mary.

2.2 设计

事实 26	从需求转入设计时,因为制定方案过程的复杂性,会激增出大量的衍生需求(针对一种特定设计方案的需求)。设计需求是原始需求的 50 倍之多。

讨论

在软件开发过程中,形势很快变得杂乱。

随着从需求阶段("什么"问题,描述了软件将要解决的问题)上升到设计阶段("如何"问题,描述了软件怎样解决这些问题),很快就发生了一些戏剧性的变化。当设计者努力寻找解决方案时,有限的问题需求开始转变为针对特定方案的需求。原来从问题角度看上去非常简单的东西,从解决方案角度看却突然变得非常复杂。这些新的设计需求从显式问题需求中得出,所以有时称之为衍生需求或隐式需求。有人发现从显式需求到隐式需求的激增数量至少达到 50 倍!

因为在软件生命周期的早期增加复杂性会影响形成简单方案的能力,所以软件设计者希望限制这种激增。但是这是一种"自行其事"的事实。无论软件理论家和实践者如何努力、持什么观点,似乎都不可能抑制需求扩充的问题。人们一直在寻找简单的设计方案,但是从未找到。

有句谚语“使用最简单的方法，但是不要比这更简单”太正确了！

该事实引出一个必然结论，这一结论与当前的讨论有相当大的关系：

虽然大家都认为需求追溯很有必要，但是需求扩充在一定程度上影响了需求追溯。需求追溯是指在产品的各个阶段的制品中追溯原始需求。

几十年来软件人士一直在寻找一种魔法，该魔法能从问题需求追溯到对应的解决方法。这种可追溯性将会使软件人员根据一项需求来确定对应的设计、编码、测试用例、文档以及其他软件制品。

这种可追溯性还有其他潜在的用途。也许最重要的用途是在改变需求时，确定所有需要改变的软件部分。对于软件维护者而言，这种追溯能力用途极大。软件维护者最大的问题是充分理解现有的软件产品，在此基础上做出修改。

可追溯性的另一用途是逆追溯。就是从详细的制品追溯到更基础的单元，可以用来查找那些存在于程序中但是不针对任何需求的代码，即悬置代码。悬置代码问题比软件新手所想像的更常见。悬置代码存在于程序中，通常没有什么坏处，所以问题并不严重。但是悬置代码会占据计算机内存，降低执行速度，搅乱人们对软件的理解过程。

虽然我们如此渴求可追溯性，但是事实证明这是一个难以获得的魔法。有一些商业性的工具和方法具有追溯功能，但是这些不能满足我们对于追溯的预期。

为什么实现可追溯性这么难？简单想来，似乎需求与对应的设计、编码和其他制品之间的关系是简单的链表问题。软件人员都有链表方面的基本知识，使用起来也非常容易。

但不幸的是，这些想法过于简单。你可能已经认识到了，它复杂的原因在于需求激增。将一个需求与由它衍生的 5～6 个设计需求链接比较容易。但是，如果每个需求通常会链接到 50 多个设计需求，每个设计需求又链接到更多的编码单元，而编码单元又可能对应于多个需求。如此一来，问题的复杂性不断增加，根本无法手工实现，采用自动化方法也非常难以实现。

争议

这又是一个很少有人知道的事实。因为这个事实也鲜为人知，所以很少有争议。该事实经常被遗忘，其原因是该事实已经存在了几十年了。我记得在 20

世纪 80 年代早期，自己在西雅图大学向研究生讲授软件工程课程时提到了该问题。

注意该事实和事实 21 的联系。事实 21 说增加问题的复杂性会显著地、呈指数关系地增加解决方案的复杂性。我们在事实 21 的争议中的说法在此依然有效——那些认为构建软件很容易的人通常不会接受这个事实。

来源

Mike Dyer 在二十多年以前首先定义了需求激增问题，他当时在 IBM 从事系统集成工作（Glass 1992）。可追溯性的历史更久远（经过这么多年，人们似乎有足够的机会找到解决可追溯问题的可行方法）。有关该事实的其他来源如下：

- Ebner, Gerald 和 Hermann Kaindl. 2002. "Tracing All Around in Reengineering." *IEEE Software*, May. 这是关于可追溯性的最新文章。
- Glass, Robert L. 1982. "Requirements Tracing." In *Modern Programming Practices*, pp. 59-62. Englewood Cliffs, NJ: Prentice Hall. 摘录了 TRW 和 Computer Science Crop.在 20 世纪 70 年代的官方报告，描述了他们在软件产品开发中的可追溯性需求实践。

参考文献

- Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall. An informal reference to the Dyer study is found on page 55. I have tried contacting Mike Dyer to get a more specific reference, but as of this writing, I have been unsuccessful.

事实 27	对于一个软件问题，通常不存在惟一的最佳设计方案。

讨论

在该事实中有两个关键词：惟一和最佳。

许多软件问题有不止一种解决方案；这讨论的是惟一性。而且即使你有了一个方案，你也很难确定是否找到了一个“最佳”的方案；这讨论的是最佳性。

该事实令人失望，但是不会使人吃惊。对于同一个问题定义，多位优秀的设

计者通常也不可能提出完全相同的最优设计方案（在软件业中，我最喜欢引用的一句话是“在一个房间中坐满了顶级的软件设计人员，如果其中任意两个人达成一致，那就可以通过”）。回想上一个事实中所提到的设计阶段的复杂性和需求激增问题。复杂性和激增表明软件设计是一个非常困难、非常复杂的过程，该过程无法简化，显然也就没有最佳的解决方案。

该事实让人失望，理由是：如果可以让设计人员在碰到一个最佳设计方案之前随意寻找，这很容易（但是，如果他们根本碰不到这样一个方案，那么就不能正确地解决问题）。

考虑该事实与一项极限编程原则之间的关系。极限编程建议尽量简化设计方案。虽然该事实并不否定这一原则，但是该事实表明许多问题没有这种简单方案（在事实 28 中讨论设计的复杂性，也强烈说明了这一点）。该事实同时从另一方面支持极限编程的这项原则。如果没有最好的设计方案，那么极限编程的简单方案可能与其他方案同样成功（只要不是过于简单）。

争议

有关该事实的争议相当含蓄。许多群体都认为存在惟一的、最佳的方案。例如，在复用社团中，有些学者提议通过特定问题的方案来建立一种组件查找机制。假如大多数问题有且仅有一个方案，那么这种机制非常有效。在相关的学术报告中，也有人提出用这种方法来解决某一问题。但是，报告者采用的解决方案通常和我想到的不同。注意，如果设计者要提出解决问题的一个方案（通常采用复用方法），而且他的设计思路与组件设计者不同，那么他就无法采用方案驱动的方法来找到特定的组件。

尽管如此，大多数设计经验丰富的实践者都认同这一事实。

来源

Bill Curtis 曾经在一次软件工程学术会议上指出“在一个房间中坐满了顶级的软件设计人员，如果其中任意两个人达成一致，那就可以通过”。

关于一般的软件设计以及该事实最好的材料来自 Curtis 和他在 MCC（Microeletronics and Computing Consortium）的同事以及 Elliott Soloway 和他在耶鲁大学的同事，这都是十几年以前的事了。当时这两个研究小组都致力于最佳设计方面的经验性研究。Curtis 等人还打算建立一个工具集来支持他们得到

的设计过程，并使设计过程自动化。不幸的是，两个小组的研究结果都表明，设计过程是随机的，这意味着例如设计过程没有特定的顺序、不可预测，几乎不可能建立任何类似的工具集。继 Curtis 和 Soloway 之后，我再没有看到类似的研究。

- Curtis, B., R. Guindon, H. Krasner, D. Walz, J. Elam, and N. Iscoe. 1987. "Empirical Studies of the Design Process: Papers for the Second Workshop on Empirical Studies of Programmers." MCC Technical Report Number STP-260-87.
- Soloway, E., J. Spohrer, and D. Littman. 1987. "E Unum Pluribus: Generating Alternative Designs." Dept. of Computer Science, Yale University.

事实 28	设计是一个复杂的、迭代的过程。最初的设计方案可能是错误的，当然也不是最优的。

讨论

事实 27 介绍了 Bill Curtis 和 Elliott Soloway 所代表的两个团队在分析软件设计的本质方面所做的研究。在事实 27 中，我提到了随机性设计的概念。随机性设计没有固定的顺序、不可预测、不采用结构化的方法，是一种完全不同的设计方法。

这种设计方法有许多特征，在此我们讨论“从难点开始”这一特征。顶级设计者在设计中并不是按部就班地采用自顶向下（或者自底向上）的方法，而是着眼于权重更大的目标。这些目标通常是难点问题，设计者不能轻易地看出这些问题的解决方案。为了得到整个问题的设计方案，设计者必须先致力于难点的设计并消除其中的疑惑。

毕竟，设计阶段的一项重要工作是可行性研究——是否有可能解决这个问题？（正如我们在事实 14 中所介绍的，即使许多软件工程师事后发现有些项目根本不可行，绝大多数软件项目的可行性研究结果也是“切实可行”。）

下面的例子是关于随机的、从难点开始设计。你是否记得在事实 18 中介绍开发通用报表生成器的故事？此前，我从未做过任何的报表生成器。我分析构建报表生成器的任务，认为其中最难的概念是各列数据求和、对和求和、对和的和求

和。我称之为“反复求和”。在开始设计报表生成器时，我分析整个问题来考虑大致方案，然后定位到反复求和问题。直到对该问题有了满意的答案，我才回过头来考虑如何解决整个问题。当然，我的反复求和问题的设计方案对于整个问题的方案起到了关键性作用。

除了“随机性”之外，Curtis 和 Soloway 的工作还有一个有意义的发现。顶级设计者在开始一个项目的实质性设计时，他们的设计方案可能是启发式的、试验性的、错误的。他们会想出一个设计方案（可能是基于以前对相关问题的设计），在心里向方案中输入一些有代表性的数据，模拟这些数据的处理过程，接受该方案的输出，确定输出结果是否正确（因为设计者等不及使用草纸或者仿真等物理方法，所以这是一个心理活动）。

最初的输出很少是正确的。如果真是正确的，那么就认为这个备选方案对于这些测试数据是可行的。这样，就可以再输入更多的数据，并重复该过程。当测试数据的覆盖足够大的时候，就可以将备选方案作为真正的方案。在多数时候，输出是错误的。这时，备选方案是错误的，应修正备选方案并消除错误。再次使用前面的测试数据，反复重复，直到正确为止。然后重复该过程。

根据 Curtis 和 Soloway（1987）的发现，设计根本不是可预测的、有结构的、规范化的过程，而是凌乱的、琐碎的、易出错误的。别忘了，这是对顶级设计者实际工作情况的研究结果。可以设想，普通设计者的工作会更凌乱。最糟糕的设计方法，也是许多新手最想采用的方法是“从容易的地方下手”。采用这种方法，虽然可以很容易地启动设计工作，但是这样所得到的“小范围方案”不能整合形成“大范围方案”。因此，这些小范围方案通常没有实际价值而被摒弃。

从本节中，我们可以明显看出设计是复杂的、迭代的（Wiegers 在 1996 年曾明确表述了该观点）。实际上，设计是软件开发中最具智力性的活动。我们还可以明显看出最初的设计方案通常是错误的。什么是最佳方案？很显然，最初的设计方案并不是最好的。但是这个词引出了另一个有趣的问题，即是否存在最佳设计方案？

从某种意义上说，事实 27 已经提到该问题并给出了答案，我们知道几乎不存在一个最佳的设计方案。但是又有一些新进展。Simon（1981）认为：复杂的设计过程通常不能得出最佳的结果，但是我们必须尽力寻找一个“令人满意的”方案。找到最佳设计方案不可能或者代价太高，而“令人满意的”方案（而不是最佳方案）可以满足优秀设计标准，是值得（冒险）选择的解决问题

的方法。

争议

对于很多人（特别是研究开发方法的学者）来说，很难放弃顺序的、可预知的设计，接受随机设计的观点。随机设计的方法不仅理性不足，而且如果建立设计过程文档，我们会发现设计文档很难理解（理由是“从难点开始”，不同人的难点不同）。实际上，Parnas 和 Clements 等著名的计算机科学家曾建议（1986）“伪造”设计过程。即使设计过程并非结构化，但是也要描述出一个看似结构化的设计过程，这样的设计文档才有用。

很难让别人放弃软件设计简单化的观念。例如，极限编程（Extreme Programming , XP）、敏捷开发（Agile Development, AD）等新的开发方法支持这种观念，并建议寻找最简单的设计方案。当然，对于一些非常简单的问题，设计过程可能同样简单。但是对于复杂的问题，XP/AD 方法可能隐藏重大的危险。

我们继续讨论最佳设计问题。确实有些人相信最佳设计是存在的，并且完全可以实现，尤其他们使用科学方法来解决商业应用时（比如所谓的管理科学家）。但实际情况是这种寻求最佳设计的科学方法只适用于极大简化后的实际问题。这时，解决方案可能是最佳的，但是同时在真实问题环境下也是无用的。

来源

有关 Curtis 和 Soloway 的材料见事实 27 的来源一节。Glass（1995）对有关问题做了详细的总结和论述。Simon（1981）的材料非常经典，我强烈建议对此有兴趣的读者阅读这本书。

参考文献

- Glass, Robert L. 1995. *Software Creativity*. Englewood Cliffs, NJ:Prentice Hall.
- Parnas, David L., and Paul C. Clements 1986. “A Rational Design Process: How and Why to ‘Fake It.’” *IEEE Transactions on Software Engineering*, Feb.
- Simon, Herbert. 1981. *The Sciences of the Artificial.* Cambridge, MA:MIT Press.

➥ Wiegers, Karl E. 1996. *Creating a Software Engineering Culture*. p. 231. New York: Dorset House.

2.3 编码

事实 29	从设计转到编码阶段时，设计者按照自己掌握的水平，已经将问题分解为“原语”。如果编程者和设计者不是同一个人，二者的“原语”不吻合，就会出问题。

讨论

通常认为将设计转化为编码是平滑的过程。只要编程者和设计者是同一个人，那么一般不会有问题。但是，有些软件公司严格区分不同的工种。系统分析员或系统工程师做需求，设计者做设计，程序员做代码（在这些公司，专门的测试员做测试）。有时，是由公司内的各工作组完成这些工作，有时也会把部分工作外包给其他公司。

如果存在工种差异，那么就有必要讨论如何才能最好地从设计转到编码。通常，设计者会将问题分解为一定层次的原语——易于理解和编码的基本软件单元。实际上，这种说法过于简单化了。简单化问题的原因在于不同的人有不同的原语集。同一个东西，对于一个人来说是原语，对另一个人却不是。

是否还记得在事实 18 中我第一次构建报表生成器？对我而言，最难的是如何处理我所谓的“反复求和”问题。在开始编码之前，我在这个问题上花了很大精力。但是对于那些做过无数个报表生成器的商业系统程序员来说，这不过是一个无足轻重的问题。这些人眼中的“原语”和我所看到的原语完全不同，其抽象层次比我高得多。与我相比，有经验的商业系统程序员会很早就停止设计，转入编码。

问题的症结在于：如果设计者的原语层次比编码者高，编码者无法将此设计作为起点。因此，编码者在真正编码之前需要花费时间完成额外的设计，填补中间的层次。因为编码者无法完成设计者期望的设计方案，所以在编码者看来这些设计很笨拙，甚至有问题。

相反的情景仍然有问题。如果设计者和故事中的我一样经验欠缺，那么就会

开发一个非常详细的设计方案。如果编码者的经验更丰富，那么他可能会拒绝这个详细的方案；并用自己的观点来弥补。其问题并不在于设计者必须比编码者聪明或训练有素，而是这应该有一致的背景或者原语集。

区分不同的工种并使用易于理解的原语以填补差距，这种看似简单的做法突然变得异常复杂。除非设计者和编码者有共同的原语集，否则从设计到编码的转变不会非常平滑。考虑到“不存在惟一的、最佳的方案”这一事实，共同的原语集是不可能的。

在我看来，因为该事实，所以不要轻易将设计工作和编码工作分开。McBreen在2000年指出：“原始的分工方法并不适合于软件开发”。当然，对于大型项目，我们不得不采用分工的方法。

争议

与本书中的许多事实一样，因为该事实鲜为人知，所以争议很少。我从来没有听说哪个有人员分工的软件公司讨论过这个问题。也许是因为这些公司的设计者和编码者有大致相同的背景。也可能是因为设计者熟知编码者所期望的原语层次，设计结果正好满足编码者的需求。

公司分设编码者和设计者也可能是为了“提高成功系数”。在一个大的项目中，通常只有很少的一部分人做早期需求和设计工作。只有完成设计后，才由编程团队接手（注意，这样做的目的不是为了分工，而是为了控制劳动力成本）。因此，这种现象在大型项目中更常见。采用这种方式时，经常会发生设计/编码不匹配的问题。在编码者开始接手之前从来没有接触过项目，这是一种错误的做法，也会影响士气。

Web软件编程创造了一种全新的软件文化。在这种文化中，项目通常很小（人们所说的3*3项目，即3人3月完成），时间很紧张。在这种条件下，通常根本就没有设计——小系统没有必要设计——所以就没有这个问题。因此，在敏捷开发或极限编程中没有提及到这个问题，其中的一些做法甚至对设计的必要性提出质疑。

来源

正如前面所提到的，在文献中很少讨论这个事实。我只能引用自己的一份材料：

➥ Glass, Robert L. 1995. "Ending of Design." In *Software Creativity*, pp.182-83. Englewood Cliffs, NJ: Prentice Hall.

参考文献

➥ McBreen, Pete. 2002. *Software Craftsmanship*. Boston: Addison-Wesley.

事实 30	COBOL 是一种非常糟糕的语言，但是其他的（用于商业数据处理的）语言也同样糟糕。

讨论

当时，编程语言是基于特定应用领域的。Fortran 用于科学计算。还有许多其他语言，比如 Ada 用于实时应用。后来，又有更多的语言，包括系统编程语言（system programming language，SYMPL）。还有 RPG 和 SQL 用于生成报表。万一其他语言都不奏效，我们还有汇编语言。

COBOL 是种古老的、备受指责的语言。COBOL 是商业应用编程语言。该语言不仅容易写，而且容易读。在 COBOL 发明时的说法是“小店职员可以写，经理可以读”，意思是非技术人员可以写 COBOL 程序，任何人都可以读该程序。

这个模糊的目标在给 COBOL 带来益处的同时，也带来了坏处。也许最能说明问题的例子是该语言的 MOVE CORRESPONDING。这个运算符要写 16 个字母，这在一定程度上了说明了 COBOL 是最罗嗦的语言。但是它同时也执行了多项任务——它可以将一个数据结构中的数据移入另一个数据结构的对应位置。这相当于多个 MOVE 语句的功能。

有关 COBOL 最重要的是其中含有商业程序员所需的许多语言特征：固定格式的数据结构适用于许多商业应用，易于生成报表；文件操作支持不同类型的文件和十进制算术，非常便于记账；数据和文件特征极大简化了文件操作，当时（甚至现在）的许多语言都没有这些操作；十进制算术特征，计算精度准确到分，满足了记账需要，而浮点数等各种常见的数据类型却无法实现这一点（有关该语言特征的更多说明，见随后的“参考文献”一节中的 *Software Practitioner*）。

在 20 世纪 50 年代，随着多种与应用领域相关的语言的问世，在编程语言方

面出现了一件怪事：不再流行定义语言的使用领域。从在20世纪60年代IBM的PL/1开始（试图让它满足所有领域的需要），语言设计者开始尝试一项与应用领域无关的全能语言。不要在意PL/1所受到的嘲笑——有人因为其中包含了其他语言的许多特征，所以指责PL/1是极度现实主义者。PL/1成为后来的Pascal、Modulas、C类和Java等语言的榜样（Ada最初的设计目标是实时语言，后来也转向通用语言，当它放弃了最初目标时，很快就消亡了）。几乎没有人再去考虑不同应用领域的不同特征(在一些学术会议和出版物中曾经倡导与应用有关的语言，Hunt和Thomas[2000]提倡“领域语言”，但是它们与计算机界的主流相距甚远，似乎从那时起没有再出现过)。

这段离奇的故事所导致的结果是COBOL依然作为商业应用的可选语言孤独存在。因为有人嘲笑COBOL的笨重、晦涩以及其他的缺点，所以选择COBOL需要很大的勇气。每年对实践者的调查均表明COBOL的应用都会显著降低，但实际上COBOL却每年都是用量增幅最大的语言。

争议

因为COBOL实在不受欢迎，所以站出来对COBOL作出正面的评价需要很大的勇气。COBOL在很多方面成为计算机领域的笑柄，但是COBOL仍然年复一年地被应用，这一事实愚弄了那些依然预言其消亡的人(据估计明年又会增加500亿行COBOL代码!)。

我这样描述COBOL，是为了解释Winston Churchill的观点：“COBOL是一种糟糕的语言，但是其他的商业数据处理语言也同样糟糕。”

来源

虽然COBOL受到很多人的诋毁，但也拥有大量的支持者。也许最大的支持者是那些发行通讯稿、开通网站和举办学术会议的人。该出版物和网站的名字是*COBOL Report*，其中有很多丰富的、新鲜的COBOL知识。在COBOL多年的发展中（有一个组织提出、研究并认可该语言的更新，更新内容包括面向对象、Web支持等“现代”特征），*COBOL Report*一直报导COBOL社团的最新动态。

几年以前，我对COBOL不受欢迎但依然使用这一现象很感兴趣，于是分析COBOL与同期的其他语言相比有什么优点。我的研究结果收录在一个名为*Software Practitioner*的系列通讯稿中：

- 1996 年 9～10 月，文章包括："How Does COBOL Compare with the 'Visual' programming Languages?" 和 "Business Applications: What Should a Programming Language Offer?"
- 1996 年 11～12 月，文章包括："How Does COBOL Compare with C++ and Java?"和 "How Best to Provide the Services IS Programmers Need"。
- 1997 年 1～2 月，文章包括："COBOL: The Language of the Future?"和 "Business Applications Languages: No 'Best' Answer"。

另一个（间接）支持 COBOL 的资料见下面一本反对软件工程的书中，其中说道："多年来，软件工程一直努力破坏 COBOL。"

➥ McBreen, Pete. 2002. *Software Craftsmanship*. Boston: Addison-Wesley.

参考文献

➥ Hunt, Andrew, and David Thomas. 2000. *The Pragmatic Programmer*. Boston: Addison-Wesley.

2.4 错误消除

事实 31	错误消除是软件生命周期中最耗时的阶段。

讨论

这是软件生命周期中惟一名称严重不统一的阶段。在本书的草稿中，我称之为"检出（checkout）"（因为我在有关软件质量的书中采用了这样的叫法）。我查看了一些软件工程方面的书籍，发现许多人称之为"测试（testing）"，但同时也有人称之为"验证和确认（verification and validation）"。幸运的是怎么叫都没关系，有关系的是在此要说明什么。我选用"错误消除"，仅仅因为它最好地描述了这一阶段的工作。这是软件开发过程中努力消除错误的工作环节。

但是真正重要的是：对于许多软件产品而言，错误消除所用的时间比汇集需求、进行设计或编码都长，通常要长一倍。

这一点通常让软件开发者吃惊。回想起来，需求、设计和编码阶段的任务都是有形的、看得见的。这些阶段的工作都是很直观的，至少在一定程度上是可以

预测的。但是在错误消除阶段会怎么样？这个阶段在很大程度上依赖于以前各阶段的结果，这些结果通常不尽人意。

在软件发展的早期阶段，有经验的前人们（他们可能在本领域有几十年的历史）通常会在错误消除开始时与新手打赌说自己的代码中没有错误。这些新手对于软件没有经验，通常会接受打赌。但是新手们总是失败。软件人员，甚至这些前人，总是极力相信自己心爱的产品中没有错误，这是软件精神的一种体现。而那些新手们根本想像不到这些人通过努力工作竟然未曾得到完美无瑕的产品。

多年来，错误消除所占的时间比例有所变化，但基本上是 20-20-20-40，即需求、分析和编码各占 20%的时间（从直觉看，许多程序员都认为编码的时间最长，但是这种直觉通常是错误的），而错误消除占了 40%的时间（Glass 1992）（近来，有人建议在需求和设计阶段投入更多的时间，将比例变为 25-25-20-30）。

有关错误消除，还有一点需要说明。在我们的讨论中，把错误消除说成编码和维护中间的一个夹层，似乎这只是一次性的工作。但是，你肯定会想起来在螺旋形的软件生命周期中，软件开发在不同的阶段反复迭代。这对于错误消除非常重要。通常而言，错误消除是介于编码和维护之间的独立活动（当然，单元测试通常分散在编码过程之中），但是其他的错误消除（例如，评审和检查等）通常存在于整个生命周期中——对各个阶段所生产的产品都要评审。

争议

有很多人不相信这个事实。即使我在这个行业浪迹多年，也难以相信错误消除需要这么长的时间。但是，战胜这种不信任很有意义。该事实是我们在本书中最重要的基本真理之一。此前，我们已经积累了以下重要真理：构建软件的过程非常难，而且容易出错；主观愿望、“突破性进展”和“银弹”都不能改变这一点。

来源

“参考文献”一节所说的那本书提到许多资料，这些资料都能很好地证明该事实。

参考文献

➥ Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall.

2.5 测试

事实 32	普通程序员认为已经彻底测试过的软件其实只执行了 55%～60%的逻辑路径。采用覆盖分析器等自动工具，可以将上述比例提高到 85%～90%。几乎不可能测试软件中 100%的逻辑路径。

讨论

在软件测试文献中有很多都是有关测试方法的（在此，我将测试定义为执行软件产品，并找出其中的错误）。但是很难统一地描述、分类，甚至讲述这些方法。

为了便于讨论，尝试以下测试方法：

- 需求驱动测试（测试是否满足了所有需求）
- 结构驱动测试（测试已构建的软件的所有组成部分是否正确运行）
- 统计驱动测试（随机测试确定软件执行的时间和结果）
- 风险驱动测试（测试确定是否已经消除了最主要的风险）

该事实与结构驱动测试有关。但是，在详细讨论该事实之前，我们先说一下如何使用上述各种测试方法。

需求驱动测试是必需的，但是并非完备。所有的软件都应该进行彻底的需求驱动测试。但是，因为前面提到的需求激增和其他原因，这种测试并不够。当激增的设计需求转变为代码时，除非能够满足一些明确的需求，否则软件产品的很多部分不会执行。

因此，我们采用结构驱动测试。为了测试这些与原始需求联系不密切的衍生功能点，我们有必要尽力测试程序中的所有结构单元。我们说“尽力”是因为不可能测试全部结构。我们说“结构单元”是因为虽然最成功的结构测试方法基于逻辑片断，但是也有人提倡采用数据流（来代替逻辑）的方法。

注意，我在此很少讨论统计驱动测试和风险驱动测试。前者使预期用户相信软件可以交付使用，后者对于高风险项目非常重要，在这些项目中，应确保找到并处理所有的风险。但是，假如彻底完成需求驱动测试或结构驱动测试的复杂度是即定的，那么后面的这两种方法只是在关键或者不常见的可靠性系统中有用。有关统计驱动测试和风险驱动测试的更多信息，请参考 Glass（1992）。

在本书中，我们只按照逻辑片断进行结构驱动测试，而不是根据数据流（数

据流显著增加了测试过程的复杂性，所以在实践中很少采用）来测试。虽然前面已经说过测试单元是逻辑片断，但是还没有回答“什么是逻辑片断？”。我们可以在语句、逻辑路径、模块、组件等多个不同层次上逐个测试。在结构驱动测试中，通常需要进行模块/组件测试，但是这还不够（因为在一个模块或者组件内部，仍然有许多结构点需要测试）。事实证明，即使做到了语句级的测试也不够（解释这一违反直观的发现需要较长的篇幅，在此就不细说了。要理解为什么语句级的测试不如逻辑路径测试有效，请参考 Glass[1992]的例子）。

下面就剩下逻辑路径测试了。在本事实的后面，我将讨论尽力测试所有逻辑路径的问题（这要测试所有的逻辑分支所对应路径的代码，所以通常称作分支测试[branch testing]，我宁愿选择逻辑测试，因为我们没有进行分支测试，而是在做分支间的代码测试）。

下面我们讨论结构测试覆盖。研究发现，程序员说某段代码已经彻底测试过了，这意味着他实际上只执行了55%～60%的逻辑路径（Glass 1992）。这一点与本节中的其他材料一起证明了软件产品的复杂性，也说明即使软件片段缔造者本人对自己的作品也不甚了解。逻辑路径测试方面的专家指出：使用工具可以将逻辑路径覆盖率提高到85%～90%（Glass 1992）。但是不可能将该比例提高到100%（因为有很多非常隐晦的路径，例如不可达的路径、异常处理等）。这些专家还建议采用审查的方法来覆盖这些未执行的路径（从事实37开始，我们将讨论评审和检查）。

在此要说的是：虽然我们有了很多种测试方法，但是由于软件产品固有的复杂性，任何测试都不会是彻底的测试。因此，（a）测试工作实际上是一种折衷的活动，关键是作出适当的折衷选择；（b）许多重要软件的发行版中存在错误，这不足为奇（追求无瑕疵软件的想法是天真的）。

争议

虽然人们对软件测试领域有了一定的了解，但是实际中的测试却异常简单。许多软件都经过需求驱动测试，但是仅很少的软件经过了各种系统化的结构驱动测试（许多程序员认识到需求驱动测试的不足，试图采取结构化测试方法，但是他们几乎不采用任何结构分析工具来确定测试的覆盖率）。在实践中很少使用统计驱动测试和风险驱动测试的方法。

在此，关键不是对于该事实及其隐含内容的争议，而是人们通常对软件生命周期中测试阶段重视不够。不幸的是，其原因显而易见。我们已经看到，因为时间表安排不合理，开发团队工作时间异常紧张。在生命周期的后期，特别是在测试阶段时间很紧。因此，很少缩减生命周期的前期时间，但是缩减了后期测试的时间。软件实践者及其经理不仅很少使用测试覆盖分析器(我们回头讨论这一点)，而且他们根本不愿意在测试方面花费必要的时间和金钱。实际上，许多人根本不知道分析器的存在。

从某种意义上讲，针对许多项目结构覆盖测试不足的问题本来应该有激烈的争议。也许在21世纪的早期，还没有人讨论软件业这一现状。

还有一些关于构建无错误软件方面的争议。虽然许多学者都认为有可能构建无瑕疵软件，并指责那些没有实现这一目标的软件公司，但是从本节中可以明显看出只有最小的软件项目才能实现这一高尚目标。我认为最重要的是应该放弃构建无瑕疵软件这一不现实的目标，而专注于更现实可行的目标，例如构建无严重错误的软件产品。

来源

在“参考文献”一节所提到的材料中，我详细描述了多种软件测试方法。虽然时过多年，但我依然相信这本书非常详细地讨论了该事实(和以后的几个事实)。

参考文献

➥ Glass, Robert L.1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall.

事实 33	即使测试覆盖有可能达到 100%，这种测试也不够。大约 35%的错误是源于逻辑路径的缺失，还有 40%的错误源于执行特定的路径组合。不可能实现 100%的覆盖。

讨论

我们暂且不考虑事实 32，假设有可能实现 100%逻辑路径测试。这样，我们是否就能达到软件无错误的目标？我们还假设在执行彻底的逻辑路径测试过程

中，我们做了所有必要的工作——构造相应的测试用例、认真执行这些用例、检验执行结果并确定这些测试都通过。这样，是否能使我们的软件中没有错误？

几十年前，我在第一次接触到逻辑路径结构测试思想和覆盖测试等工具时，就自问过上述问题。无论是以前还是现在，这都是一个非常重要的问题。

在当时，我找到解决该问题的一个研究方法。当时我从事航天软件开发工作，可以接触到许多航天软件的错误数据。我决定针对每一个错误，研究一个问题，即“是否完全逻辑路径覆盖可以找到该错误”。假如所有的答案是“可以”，那就意味着结构驱动测试似乎可以消除全部错误，以得到无错误软件。

从前面的论述你一定会猜到：对于非常少的错误而言，答案是“可以”。有许多错误即使完全覆盖也检测不到。我认真察看了许多错误报告，发现它们主要是以下两类：

1. 缺失类错误，即程序中没有执行某种任务的逻辑。
2. 组合类错误，即只有执行特定的逻辑路径组合时才显示出的错误。

第一类的问题很明显。如果逻辑根本就不存在，那么逻辑路径也就不会存在。第二类的问题比较微妙。从这些错误报告中，我发现：独立的测试每一个逻辑路径，却不能发现这些只有在特定的路径组合中才出现的错误。有一个非常通俗、非常简单的例子，如果逻辑路径中的一个变量在前面执行的某一个逻辑路径中不能正确地初始化，但是在其他路径中可以。如果连续执行了这两个路径，就会出问题。

这自然会引出一个问题，即这两种错误出现的频率如何？遗憾的是，在我看到的数据中这些错误非常多。在我研究的错误中，缺失类错误占35%（实际上，我在随后的一项研究中发现许多持久性错误也是这种错误，持久性错误是能逃过所有的测试并进入产品中的错误[Glass 1981]），而组合类错误竟然高达40%。100%的测试覆盖远远没有实现消除全部错误的目标，这种做法很诱人，但是并不够，不够的程度是75%（完全的结构覆盖只能消除软件产品中25%的错误）。

当时我提倡结构测试和结构覆盖分析的方法。虽然这一发现对我来说是一个打击，但是我还坚信这是该领域中的一项重要发现。

争议

我公布这一研究结果的方法是错误的。我没有像许多学者那样写论文并公开发表，而只是将它收录到自己当时写的一本书中（Glass 1979）。因此，这个事实

并没有像书名所说的那样成为经常被遗忘的事实，相反却成了鲜为人知的事实。

因为有了这段历史，所以人们对这个事实不是反对，而是怀疑。我记得自己曾在德国的一个学术会议上提到该事实，软件工程界的一位著名委员竟公开抱怨说她从未听说过该发现。她的说法也许是对的，她可能从未听说过该发现。

还应该注意的是，几乎不可能实现没有错误的软件。如果严格的需求驱动测试和结构驱动测试还不能消除软件中的错误，那么我们还有什么办法呢？有人提出采用形式化验证、自测试软件等方法，但是还没有看到有严格的、实在的研究能证明这些替代方法的有效性。

该问题的实质是：为了构建成功的、可靠的软件，需要综合采用多种错误消除方法，通常是越多越好。对于这个问题，没有神奇魔法。

来源

该事实源于我的一项研究，但不幸被我埋在书中多年。我也通过一些途径反复宣传该发现，例如下面的材料，但是仍鲜为人知。

➥ Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall.

参考文献

➥ Glass, Robert L. 1979. *Software Reliability Guidebook*. Englewood Cliffs, NJ: Prentice Hall.

➥ Glass, Robert L.1981. "Persistent Software Errors." *IEEE Transactions on Software Engineering*, Mar. Note that by this point I had learned that it was important to publish software engineering research findings in stand-alone papers.

事实 34	没有工具就无法做好错误消除工作。人们常用调试器，很少使用覆盖分析器等其他工具。

讨论

我引用喜剧演员 Rodney Dangerfield 的一句话“勿以等闲视之”来说明软件

生命周期的后期阶段。

人们非常重视软件周期的需求分析和设计等前期阶段，并采用 CASE 工具支持这些阶段。还有许多针对它们设置的大学课程（系统分析和设计的课程随处可见）。有很多文献都讨论如何做好这些工作。

对于后期的测试和维护就不是这样了。存在支持这些阶段的工具（这是该事实要讨论的），但是却很少使用。几乎没有关于这些阶段的大学课程（在许多大学，生命周期似乎在编码之后就停止了）。文献似乎都瞧不起这些阶段，特别是维护（有一些关于测试的材料，但是相对较少）。

在测试阶段，使用工具不足的问题最明显。有许多测试工具，例如：调试器、覆盖分析器、测试管理器、环境仿真器、捕获/重放“自动”测试器、标准测试（在有限的环境中）、测试数据生成器。问题是这些工具很少应用。实际上，很少买[工具因为阁件（shelfware）而知名，阁件指那些买来之后束之高阁的软件。很少有测试工具是阁件。自动化测试工具（参事实 35）是个例外。它们把整个架子都填满了]。

问题出在哪里？是不是这些工具不管用？事实证明，许多前端工具过于夸张，仅仅对于一些次要方面有较少的帮助作用。但是，如果我们使用生命周期的后期工具，会受益匪浅。调试器对于追踪错误非常有意义；覆盖分析器是结构驱动测试的基础工具；测试管理器显著降低重复测试的数据工作量；如果没有环境仿真器，几乎无法测试嵌入式系统；虽然捕获/重放不能实现“自动测试”的说法，但是也能显著降低重复测试中的文书工作；如果有必要（例如检查语言与编译器的符合程度），也应该采用标准化测试；测试数据生成器在统计测试等方面非常重要，可以生成和再生成测试数据。

问题不是工具缺少有效性，而是我们缺少重视。测试工具使用不足，有三方面的原因：

1. 软件领域重视生命周期前期工作的可视化，管理者和学术界尤为严重。当然，使前期工作可视化无可厚非。我们已经看到前期出现的错误会造成严重的损失，而且需求/设计错误非常难克服。但是，没有必要在强调生命周期前期的同时忽视后期，而实际情况正是如此。
2. 软件生命周期后期工作的技术是“低劣的”。即使不能真正做分析或设计的人，也可以想像其中的工作。而测试和维护却不是这样。做测试和维护需要有非常神秘的技术（我称之为“低劣”），这些东西非常神秘，以至于

一般肤浅的管理者没有兴趣了解。

3. 在生命周期的后期，时间表压力非常大。特别是，通常没有时间做足够的测试。经常需要急匆匆地完成测试，以便赶上时间表。拥有和使用测试工具会占据前端的注意力和本阶段的精力——注意力和精力通常无法得到。

最根本的测试工具是调试器。根据研究和实践经验，多数软件开发者确实使用调试器，但是却忽视了多数其他工具，甚至许多开发者和更多的经理都不知道它们的存在。许多测试员也是这样。

有趣的是，针对许多人对于工具无知这个问题，一些供应商提供了所谓的工具商品目录，在该目录中列出了各种工具的来源、价格和适用范围（ACR 2001）。但是，即使这些商品目录也很少有人买（我所访问过的机构都很少听说过它。我的一位朋友是一种目录的开发者，所以我知道他们的营销何等不易）。

在应用测试工具方面，也许最中肯的是 Zhao 和 Elhaum（2000）近来对于开源软件可靠性方面的研究。对于开源软件开发者的调查表明，测试通常是斯巴达式的。仅 39.6%的人使用过测试工具，其中的大多数仅仅使用调试器，几乎都不使用覆盖分析工具，几乎没有人制定任何测试计划。

这并不是想给开源软件的开发者找麻烦，对于传统开发者的研究也得出类似的结果（测试计划可能更常用）。然而，开源软件的开发者使用工具不足的原因却很有意思，这些开发者显然是相信“假如有了足够多的眼球，所有的 bug 都显而易见”，希望这样能彻底清除他们产品中的错误（开放源代码者希望他们的用户能阅读和分析代码，这样就可以找出其中的错误）。当然，如果软件用户想读代码，而且确实读过代码，那么这种观念可能会非常成功。但是，如果该产品没有吸引到足够的眼球，那么应该说这些开源软件没有达到预期的可靠性。当然，开发者无法知道是否有足够的眼球。

争议

正如我们前面所提到的，极少有人注意软件生命周期后期。因此，对于该事实几乎没有争议。问题并不是某人对于该事实持赞成还是反对的态度，也不是人们对于该事实的无知，而是对于后期应该投入同样的精力，特别是在测试工具方面，人们对于该问题的兴趣很少，所以也不被重视。我猜想，如果由软件工程师来判定，他们中的许多人也会赞同该事实——然后耸起双肩，继续按照原来的做法行事。

来源

上述对于开源软件可靠性的研究表明了工具使用的欠缺，见“参考文献”一节。

参考文献

- ACR. 2001. The ACR Library of Programmer's and Developer's Tools. Applied Computer Research Inc., P.O. Box 82266, Phoenix AZ 85071-2266. This was an annually updated software tools catalog. It is recently discontinued.
- Zhao, Luyin, and Sebastian Elbaum. 2000. “A Survey on Quality Related Activities in Open Source.” *Software Engineering Notes*, May.

事实 35	自动测试很少，也就是说有些测试可以也应该自动化，但是有许多测试任务不能自动完成。

讨论

在软件业的历史中，人们一直梦想着测试过程自动化。结果一个又一个梦想都破灭了。

一方面，由规格说明自动生成代码，许多研究人员认为这有可能，直到 Rich 和 Waters（1988）的一篇研究论文中提出“鸡尾酒会的秘密（cocktail party myth）”，研究人员才开始放弃这种想法。接着，人们又认为 CASE 工具自动完成生命周期前期的许多工作。尽管还有人宣扬“不用程序员也能编程”和“编程自动化”，但是整个思想最终消亡了。虽然 CASE 工具有用，但是其中的许多已经搁置不用了。出现这种现象的原因是 CASE 并不像宣传的那么神奇。

那些鼓吹自动化的人已经在软件周期的前期失败了，他们一定会转移到后期。在软件界中即将上演的是自动测试工具软件。如同前面的鼓吹一样，软件测试自动化的说法有一定的合理性。已经有了一些优秀的测试工具，确实可以帮助程序员和测试员找出一些错误，而且使部分测试工作自动化。

但是应该强调“部分”一词。不同测试工具使不同的工作自动化。但是这些自动化都加起来也与完全测试自动化相去甚远。

- 例如，捕获/回放工具可以记录测试输入，以便在需要时多次执行。
- 例如，测试管理工具可以重复运行一组测试用例，并比较它们与预期（*oracle*，即正确结果）是否一致（test oracle 是表示一组已知正确答案的计算机术语）。
- 例如，如果插入了新代码，生成和处理回归测试集合可以很好分离受影响的旧代码和正确代码。

但是还有一些关键的测试任务没有实现自动化，它们是：

- 选择测试什么，如何测试。
- 生成测试用例，保证每个测试所代表的等价类最大化。
- 收集预期正确测试结果，生成测试预期结果集。
- 规划测试过程的基本架构。
- 确定测试是否完成。
- 协调用户和客户，做好测试过程中的组织、管理和结果检查。
- 选择有效的折衷方案，使得测试技术和测试行为的收益最大化。

如此等等。测试和前面提到的编程都异常复杂，不能实现完全自动化。但是，这不妨碍我们使用工具自动完成尽量多的工作，我们也不应该再相信测试过程完全自动化的说法。

争议

到目前为止，只有测试工具的供应商还会坚持完全自动测试的说法。我们应当忽略所有的供应商，不应该再去理会那些坚持完全自动化的人。

注意，本书中的很多事实都基于一点，即：构建软件的过程非常复杂，依赖于智力活动，几乎很难简化。自动化使这种繁琐的活动简单化，而那些声称已解决了该问题的人严重影响软件界寻找更好、更现实的工具。

来源

有很多人站出来勇敢反对软件业中的鼓吹者，但是鼓吹者仍然络绎不绝。也许 Brooks 反对鼓吹的方法最明显，他反对寻找“银弹”这种能消除软件复杂性的魔法。

➥ Brooks, Frederick P., Jr. 1995. “No Silver Bullet—Essence and Accident in Software Engineering.” In *The Mythical Man-Month*, Anniversary ed. Reading MA: Addison-Wesley。它还作为论文发表过。该书的第一版在几十年以前

就发行了。

在下面一篇文章中，对于测试自动化持现实的观点。

- Sweeney, Mary Romero. 2001. "Test Automation Snake Oil." In *Visual Basic for Testers*, by James Bach. Berkeley, CA:Apress.

在该书中，提出了许多测试方面的教训。第5章打破了许多有关自动化测试的神话。

- Kaner, Cem, James Bach and Bret Pettichord. 2002. *Lessons Learned in Software Testing*. New York: John Wiley & Sons.

参考文献

- Rich, Charles, and Richard Waters. 1998. "Automatic Programming: Myths and Prospects." *IEEE Computer* (Aug.): 40-51.

事实 36	程序员在程序中嵌入测试代码、目标代码中的编译参数等方法，都是测试工具的重要补充。

讨论

在这个高度自动化的年代，似乎采用软件测试工具和技术可以轻易完成各项工作。但是这种做法是错误的。

我们手头就有一些简单的测试方法，它们通常是最基本的测试方法。放弃闭门检查自己代码的习惯，争取让同事帮助你检查代码。如果有些神秘 bug 的难度超过书面检查的能力，那么你应该在程序中加入简单的测试代码。调试是软件编程中的探寻过程，你扮演福尔摩斯来探寻难以捉摸的 bug。与福尔摩斯一样，你必须动脑子，必须采用自己能想到的各种方法。

向程序中增加代码来隔离和探寻错误的做法似乎违反直觉，但是这对于发现错误非常有意义。只知道了一个变量在特定时间的值，就将极大地帮助你了解程序运行细节。可以插入一些代码来显示变量在特定时刻的值。

如果是持久性问题或者采用探索工具也许奏效的问题，你可以考虑半永久地注销程序中的测试代码。在编译时，还可以采用适当的文本编辑器或者条件编译选项来决定是否包含这些代码。因此一旦加入了调试代码，就可以一直保留以备

后用（别忘了在调试代码开和关两种方式下都要调试你的程序）。

争议

许多程序员几乎都会自觉地采用这些基本技巧。但是在许多入门课程中都不会明确讲述该事实，所以有必要在此讨论。这也证明软件实践者懂得一些方法的重要性，但是计算机理论家却不懂。

来源

有关书面检查和源代码调试，请参阅：

- Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall.

2.6 评审和检查

事实 37	在运行第一个测试用例之前进行严格审查可以消除软件产品中多达 90%的错误。

讨论

在前面的多个事实中，我们一直在倡导：在构建软件过程中，不存在任何突破性的工具和方法。在此，我们又看到不存在“银弹”。

有希望了：在消除错误的环节，确实存在一种比其他技术更有突破性的技术。通过严格审查来查找软件中的错误，这种方法有了很大的突破性。研究表明：在运行测试用例之前进行严格的审查可以消除软件产品中多达 90%的错误。这说明该过程非常有效。

而且，研究还表明：要找到同一个错误，审查的成本低于测试的成本。因此，审查不仅效果显著，而且效率很高。这真是相当好的组合。

那么为什么检查（有时候也称为复审，不过有些人区分这两种叫法）的方法还不如 CASE 工具等“突破性”更小的方法更受欢迎？原因如下：

1. 几乎没有供应商能从审查中获利。
2. 审查没有任何新鲜内容（因此没有市场）。

3．通常认为审查处于软件生命周期中遥不可见的后期。

4．虽然审查的效率很高，但是需要大量的、艰苦的脑力劳动。

我们逐一讨论这些理由。

前两个理由都讨论动机问题，是一回事。还有谁会愿意宣讲审查的重要性？当然叫卖审查课程的人会，但是他们所叫卖的内容与二三十年以前没有太大差异。因为极少的公司愿意学习如何做好审查，所以这些鼓吹者失去了经济动机，就保持沉默。

我们已经讨论了第三个理由，即人们普遍忽视生命周期的后期，对于审查也如此。别忘了，我们对于生命周期各阶段的产品都可以审查。但是人们普遍认为这只是生命周期后期的事。

对于第四个理由需要做详细解释。执行审查似乎很简单，但事实并非如此。虽然叫卖审查课程的人非常强调形式化的步骤、标准和角色，但是决定检查成败的关键不是采用形式化的过程，而是团队成员在审查过程中的严格程度和注意力的集中程度。这种严格需要付出很高的代价。我曾参与过审查，一个小时的高度集中已经让我和其他人筋疲力尽。而且审查一百行代码大约需要一小时，审查一个几千行代码的小软件也需要相当的时间投入。

结果是，即使审查比其他的测试方法成本更低，但是很少有公司会对自己构建的软件逐行审查。我们又看到在错误消除中有许多重要的妥协——我们应该审查哪些代码片段的哪些部分？答案通常是“关键代码的关键部分”。当然这个答案是与领域有关的。

或许还没有注意该事实的一个特殊点。一项技术可以找到软件产品中90%的错误，这当然令人印象深刻。但是这引出一个问题：因为只有使用软件一段时间之后，我们才能知道软件产品中有多少错误。我们怎么可能知道在测试阶段已经找到了90%的错误？当然不可能。虽然我从未见过有关这个问题的讨论，但是我想正确的表述应该是“在运行第一个测试用例之前进行严格的审查，可以消除软件产品中多达90%的已知错误（*known errors*）”。

争议

知道该事实的人通常不会怀疑上述有关审查的说法，而不知道的人可能几十年来一直都不知道，因此目前也不会产生新的观点。所以有关审查的真正争议在于如何做审查。许多人鼓吹采用形式化的步骤、标准和角色的方法，还有一些人

建议最好采用会议的形式。

但是已经有许多研究成果怀疑上述两种观点。还有一些（Rifkin 和 Deimel 1995）审查方法不强调形式化，而强调技术方法——研究如何读软件或应该从中找什么。在有些时候，个人审查和会议审查同样高效。甚至还有人研究会议审查的最佳参加人数，结果是 2～4 个人。

然而，也有人恳切地反对审查，理由是这会影响团结和士气。显然，我们应当认真看待审查所带来的社会影响。

来源

据我所知，审查最少可找出 60%的错误（Boehm 和 Basilli 2001）。Bush（1989）和其他研究结果宣称达到 90%。

上面的讨论涵盖了许多研究成果。我不是逐一引用这些研究，而是给出有关该话题的一项更详细的研究，在这里你会找到这些材料的详细出处。

- Glass, Robert L.1999. "Inspections—Some Surprising Findings." Practical Programmer. *Communications of the ACM*, Apr.

在“参考文献”一节中，Rifkin 的文章是非正规的、高技术的审查方法与传统的审查方法很好的例证。

即使有关审查的很多书都强调形式化的方法，但其中不乏优秀者。在一本最近的好书中，作者很好地描述了各种形式化理论，即：

- Wiegers, Karl E. 2002. *Peer Reviews in Software—A Practical Guide*. Boston: Addison-Wesley.

参考文献

- Boehm, Barry, and Victor R. Basili. 2001. "Software Defect Reduction Top 10 List." *IEEE Computer*, Jan.
- Bush, Marilyn. 1989. Report at the 14th Annual Software Engineering Workshop. NASA-Goddard, Nov.
- Rifkin, Stan, and Lionel Deimel. 1995. "Applying Program Comprehension Techniques to Improve Software Inspections." *Software Practioner*, May.

事实 38	虽然严格审查有很多优点，但是不能也不应该代替测试。

讨论

前面看到，审查是软件中最具“突破性”的技术。现在我们来澄清“没有实质性突破”的观点。

审查是一种有效的技术，但是并不足以完成全部的错误消除工作。在本书前面我已经说到，在后面还会说（事实50）：错误消除是一项复杂的工作，需要综合采用测试者所掌握的一切方法和工具。

寻找错误消除阶段中的银弹与寻找软件开发许多阶段的银弹一样，是一项持久的工作。鼓吹者多次吹嘘找到了许多错误消除方法的银弹。计算机学者在很早以前就说严格执行正规审查就足够了。缺陷耐受者认为软件自查可以找出并更正错误，这已经足够了；测试人员有时也会说 100%的测试覆盖已经足够了。只要说出你对错误消除的立场，我就知道你会怎么说。

不！软件构建是一项非常复杂的、容易出错的活动，会出现各种难以发觉的不同错误。软件开发者和其他人一样有同样的人性弱点。不存在错误消除的银弹。即使审查有很强大的功能，但是仅仅审查还是不够。

争议

尽管偶然会有人说审查可以 100%有效（Radice 2002），但是多数人都知道无论审查如何重要，也不能当作惟一的错误消除方法。

有关这一事实，反对比争议更多。软件人员极力相信有可能消除软件中的全部错误，他们需要正确的错误消除方法实现这一目标。正如我们从来没有停止在其他开发领域寻找银弹，在该领域的寻找也从来没有停止过。因此总有人会说他们找到了神奇的东西，使不可能变得可能。

不要相信他们！

来源

多年来，有很多研究比较测试和审查的效果（Basili 和 Selby 1987; Collofello 和 Woodfield 1989; Glass 1991; Myers 1978）。所有这些研究结果都证明（各种）审查比测试更高效、成本更低，但是他们都不建议取消测试。

另外，在我们的其他一些事实中有许多相关的来源。事实 37 说审查可以发现90%的软件错误；事实 32 和事实 33 测试分析法覆盖率可达软件产品的 90%，但是即使达到了100%的覆盖也不够；事实32讨论因为存在需求扩充问题，所以100%的需求驱动测试也不够；事实 35 说测试的自动化程度太低，不足以满足需要。形式化验证（又称为正确的证据）与软件构建一样也容易出错，此外还存在其他原因，所以也不能称之为银弹（Glass 2002）。

有很多事实可以证明软件错误消除是一个涉及多层面的话题。

参考文献

- Basili, Victor, and Richard Selby. 1987. "Comparting the Effectiveness of Software Testing Strategies." *IEEE Transactions on Software Engineering*, Dec.
- Collofello, Jim, and Scott Woodfield. 1989. "Evaluating the Effectiveness of Reliability Assurance Techniques." *Journal of Systems and Software*, Mar.
- Glass, Robert L. 2002. "The Proof of Correctness Wars." Practical Programmer. *Communications of the ACM*, July.
- Glass, Robert L. 1991. *Software Conflict*. Englewood Cliffs, NJ: Yourdon Press.
- Myers, Glenford. 1978. "A Controlled Experiment in Program Testing and Code Walkthroughts/Inspections." *Communications of the ACM*, Sept.
- Radice, Ronald A. 2002. *High Quality, Low Cost Software Inspections*. p. 402. Andover, MA: Paradoxicon.

事实 39	通常认为，事后评审对于了解客户的满意度和改进过程都很重要。但是很多软件公司不开展事后评审。

讨论

根据本书前面的事实，在软件发行版中存在问题，我们不必感到奇怪。在软件生命周期过程中，有些东西一直不能达到团队成员的预期要求，我们同样不必感到奇怪。

真正惊人的是我们对此无动于衷。我们已经采用 alpha 测试、beta 测试、回归测试等方法，但是我们还没有回顾历史、分析过去、提出改进方法。

最终结果是：我们抛弃了在典型软件项目中得到的所有教训，甚至在项目结束时就随风而去。为什么？因为软件业必须疯狂地追赶时间表，程序员昨天刚完成一个项目，今天又被塞进另一个项目中。在新的项目中，他们一直忙于追赶新的时间表，没有时间再回过头去考虑 X 个月以前的事了。

X 个月以前？常识告诉我们，在项目刚刚结束时就总结教训为时过早（因为还没收集到使用反馈方面的教训）。多数人建议最好在交付后 3～12 个月再进行事后评审（Kerth[2001]称之为回顾）。但是 3～12 个月是软件的几个轮回，因此 Kerth 建议在软件结束后 1～3 周评审，在那时可以评审项目的所有技术内容（可能还没有应用反馈）。

事后评审包括哪些内容？有人建议评审两方面内容（实际上是两种不同的评审）：一是以用户为中心的评审，从用户角度讨论产品；二是以开发者为中心的评审，使开发者有机会反思错误，提出改进内容和改进方法。

不管怎么说，评审内容包含了许多想法，但是当前都不现实（可笑的是，我们似乎一直在寻找时机研究历史教训，并更正反复出现的错误）。多么惭愧。因为该事实，软件业原地停滞，在一个又一个项目中犯同样的错误。Brossler（1999）说“开发团队不能从已有的经验中受益，他们反复犯同样的错误”。例如，我们口口声声说要找到最好的解决方法，但是却草率检查最好的实践文档，这说明我们又回到了几十年以前软件工程教课书中所描述的做法。在此，我们丢弃了回顾历史、提出改进方案的生动素材。

我们先讨论别的东西，即有关软件发展迟缓的其他更深入的说法。一位澳大利亚同事 Steve Jenkin 向我说出他对软件行业进步程度的观点。他说，随着时间的推移，软件开发者的平均经验水平会停滞不前。这听起来不可思议，难道不是吗？但是他的意思是随着不断有新人涌入这个快速增长的职业，老手不断增长的经验水平与新手较低的经验水平相抵消。我考虑 Steve 说的话，想到应该引入一个基本原则：

> 软件业的智慧一直没有增长。

如果经验水平不增长，那么智慧也不会增长。实际上，我们在疯狂追求新东西时往往抛弃了许多旧东西（例如极限编程和敏捷开发等最新、最流行的软件方

法倾向于拒绝老方法中积累的智慧)。

怎样才能增加智慧？怎样捕获那些“经验教训”(IEEE 1993)？怎样实施事后评审？马里兰大学的 Vic Basili 及其同事（Basili 1992，IEEE 2002）提出的 Experience Factories（经验工厂）怎么样？基于最优秀的实践文档而不是基于教课书可以吗？Brooks（1995）和 Glass（1998）等人的个人回顾可以吗？在行业中高度重视增加集体智慧可以吗？在知识智能管理时代，我们急需获得、管理和应用这些知识（Brossler 1999）。

争议

考虑过这一基本事实的人都知道这是个问题，但是似乎没有人提出任何可行的改进方法。

对于前面的必然结论有许多潜在的争议。在本书出版前，Karl Wiegers 评论本书说：“我不同意……加入的新手不如离开的老手。如果你承认加入的新手至少带来一些智慧，那么我认为积累的智慧会不断增加。”然后他讲了一个有趣的故事，说自己作为一个有经验的软件工程师如何与一个年轻的新手合作，将两者的智慧有机结合。

无论如何，我相信很多人会承认：软件业一直忙于加速工作，以致于没有时间来考虑如何做得更好，而不只是更快。我们讨论更明智的工作，而不是更繁重的工作。但是谁有时间来实现更明智的工作？

而且我们缺少从经验吸取教训的机制。学术界似乎对那些通过经验性研究、来源于实践经验的新理论并不感兴趣。学术界似乎只追捧自己的东西，对于实践中形成的理论问题不感兴趣。

来源

该事实的来源如下：

- Basili, Victor. 1992. “The Software Engineering Laboratory: An Operational Software Experience Factory.” Proceedings of the International Conference on Software Engineering. Los Alamitos, CA: IEEE Computer Society Press. 在近些年来，Basili 写过许多有关该话题的材料，组织过有关的讲座。他的联系地址是：Computer Science Dept., University of Maryland。
- Brooks, Frederick P., Jr. 1995. *The Mythical Man-Month*. Anniversary ed.

Reading, MA: Addison-Wesley. 这是项目的最后回顾。在本书中，Brooks讨论了他自己的经验，以及自己在开发空前的大型软件项目 IBM Operation System 360（OS/360）中所汲取的教训。

- Brossler, P. 1999. "Knowledge Management at a Software Engineering Company — An Experience Report." Proceedings of the Workshop on Learning Software Organisations (LSO'99).
- Glass, Robert L. 1998. *In the Beginning: Personal Recollections of Software Pioneers*. Los Alamitos, CA: IEEE Computer Society Press.
- IEEE. 2002. "Knowledge Management in Software Engineering. Special issue." *IEEE Software*, May。其中包含有关事后评审和经验工厂等多个论文。
- IEEE. 1993. "Lessons Learned."Special issue. *IEEE Software*, Sept.
- Kerth, Norman L. 2001. *Project Retrospectives: A Handbook for Team Reviews*. New York: Dorest House.

事实 40	同行评审涉及技术和社会两方面问题，忽视任何一方面都会产生严重的灾难。

讨论

有一个词与评审相关，有必要在此强调其重要性，这个词是*严格*（rigor）的。参加评审的人必须全力投入、全神贯注。因为评审的本质和软件产品的复杂性，评审者应该比其他软件开发者更高度集中注意力。高度集中是保证绝对严格的惟一方法。

在评审过程中，参与者应该尽力熟知当前软件中的每一个决定和细节。因为评审者必须从评审对象作者的角度来处理评审对象，而不是按照自己的方式，所以评审非常难。我们说过，对于一个问题几乎不存在惟一的最佳设计方案。在评审过程中，假设能按照评审者自己的观点来评审，这样会比较容易。但是实际上需要按照开发者所选择的方法，所以就很难。许多人穿上别人的鞋都寸步难行。

当然，这些都是技术问题。构建软件方案的技术因素要求开发者必须严格，

但是，真正实现严格又非常难。而所有这些技术问题却引出另一个问题。

因为我们高度重视良好的评审需要高度集中这一技术问题，所以忽视了评审的社会问题。在许多社会活动中，只有一小部分人致力于活动的内容，其他人则致力于社会关系方面的工作。在软件评审中，这变得困难。对评审严格性的专注淡化了对社会问题的关注。而评审中的社会问题非常重要。虽然我们提倡“忘我编程”，但是多数人在软件产品中投入大量的情感和智力，因此如果被别人评头论足将会非常敏感。评审者经过严格认真的工作才能得到评审结果，这些结果在评审组的其他人面前也体现出评审者自我的因素。因为存在这些重要的“自我意识”以及缺少社会保护机制，所以会爆发社会问题。

现在有很多的正规评审方法，其中大许多正规方法考虑到如何处理社会问题。禁止经理参加评审（他们倾向于评审开发者，而不是评审产品）、禁止没有准备的人参加评审（他们会使有准备者失望，还会偏离主题）。开发者不能作为评审主管（缩小可能涉及到开发者的自我因素）。在许多正规评审中体现了许多宝贵的社会经验；在非正规评审中，也同样应该注意这些问题。

我清楚地记得自己第一次参加代码评审的情景。我们几个人想在一种友好的氛围下评审产品（其中的一些代码是我写的）。（我们在评审代码的同时，也想体验评审代码这一新观念。）为了得到良好的社会氛围，我们在一个评审者的接待室集合，首先规定在评审中采用完全公开、自由的政策。这些都有效，但是还不够。在第一个小时结束时，因为上述的原因，大家都感到不安。实际上，我们在一小时之后（仅 60 行代码）就结束了评审。我们都非常希望能够完成这项工作，但是却没有。

争议

有关评审的争议主要在于评审是否有必要，对于社会方面的争议很少。我想，正是因为许多人深刻认识到了评审的社会化问题的难度，所以才会质疑评审的必要性。

来源

多年来，有许多有关评审方面的好书，但是我喜欢的是（因为这该书比较新，而且我参与了审定）：

➥ Wiegers, Karl E. 2002. *Peer Reviews in Software: A Practical Guide.* Boston: Addison-Wesley.

2.7 维护

事实 41	维护开支通常占软件成本的 40%～80%（平均为 60%）。因此，维护可能是软件生命周期中最重要的阶段。

讨论

在讨论软件维护的话题中，有许多问题会使不甚了解软件领域的人感觉吃惊。首先是维护在软件业的含义。在许多其他行业中，维护指的是修理破坏或者磨损的部件。但是软件的本质决定它永远不会破坏或者磨损。软件是无形的事物，没有任何特定的物理形态，因些不存在破坏或者磨损的部件。

但是在软件中可能存在错误，而且可以修改软件来实现新的功能（实际上这就是软件 中软字的由来。因为软件是无形的，所以也是高度可塑的）。软件中的错误并不是因为材料的脆弱，而是在软件构建或者修改过程中引入的错误，因此软件维护就是在发现错误时进行修订，并在必要时做修改。对于其他行业的人来说并非如此，但是对软件人员确实如此。

第二点惊人的是，软件维护消耗的时间和金钱。在构建软件阶段所花费的时间和金钱占 20%～60%，维护阶段占其余的 40%～80%（因为后面要详细说明这一点，我们现在只说软件维护约消耗了软件生命周期 60%的时间和金钱）。从金钱和时间的角度看，维护是软件产品中的主要阶段。错误更正和修改消耗了大量的金钱和时间，因此维护可能是软件生命周期中最重要的阶段。

即使在软件人员看来，这也有悖于直觉。正是因为软件人员坚信他们的产品中没有错误，所以他们相信自己的产品投入使用后会安稳地运行几年甚至几十年。实际上，Y2K 问题（千年虫问题）就是软件维护的一个很好的例子（用两位数表示年度，当从 1999[99]年进入 2000[00]年会有问题，在有些程序中会意味着时间倒转。Y2K 就要修订这个问题）。Y2K 有两点使人吃惊，一是该问题影响面非常大；二是程序经过很长时间之后还需要修订（许多程序可以追溯到 20 世纪 60 或 70 年代）。

在软件界中有一句名言，这也是我要引出的一个必然结论：

古老的硬件会被废弃，古老的软件每天都在使用。

争议

该事实证明了介绍那些经常被忘记的事实的必要性。软件人员在工作中，似乎认为原始的软件产品不再变动。学院中的软件课程也是这么讲。无论是在实践中还是理论界，人们对于所谓的软件生命周期的前期阶段都非常重视，但是对于维护这一重要环节却很少提及。实际上，许多公司没有收集足够的数据，不知道维护的工作量。计算机科学或软件工程课程中涉及到软件维护的资料确实很少。

忘记该事实的结果是一些大愚若智的人疯言疯语。在下一个事实的讨论中，我们将看到有哪些疯言疯语。我们将讨论一个明确的问题："软件改进和错误更正分别占软件维护多大的比例？"如果你认为错误更正的比例大，那么又有一些东西让你感到吃惊。

来源

人们很早就知道了该事实，在软件界中第一次公开该事实大约在三十年以前，此后一直有人重复这种说法，因此没有理由忘记该事实。但是人们向来如此。

- Boehm, Barry W. 1975. "The High Cost of Software." In *Practical Strategies for Developing Large Software Systems*, edited by Ellis Horowitz. Reading, MA: Addison-Wesley.
- Lientz, Bennet P. E., Burton Swanson, and G.E. Tompkins. 1976. "Characteristics of Applications Software Maintenance." UCLA Graduate School of Management. 这是随后在《Communications of the ACM》（1978 年 6 月）发表的一篇论文的基础，其中的资料发展成软件维护方面最知名的、最重要的早期著作（这本书非常重要，可以说，这么多年来它仍然是软件维护方面的经典著作）。

为避免你认为该事实会随时间而消逝，我引用了一些近期的资料：

- Landsbaum, Jerome B.,和 Robert L. Glass. 1992. *Measuring and Motivating Maintenance Programmers*. Englewood Cliffs, NJ: Prentice Hall.

事实 42	增强功能大约占软件维护成本的 60%，错误更正仅占 17%。因此，软件维护的主体是在旧软件中加入新功能，而不是更正错误。

讨论

大约 60%的软件开支都用于维护。这些支出获得了什么？

事实证明，这又使人震惊，这也是整个软件业中让人极为震惊的一点。60%的维护成本全部用于改进——修改使软件更有用。在软件领域，这种改进称为增强（enhancements），通常这些活动源自于原始开发中未曾考虑的新需求（有些是因为成本和时间表的原因而推迟实现的需求）。

回想事实 23 中所说，需求不稳定是导致软件失控的两个重要原因之一。好，不稳定需求又来影响我们了。难点在于最初开发软件时，客户和用户仅从较为片面的角度考虑了产品应实现的功能。只有在产品投入应用而且用户使用一段时间之后，他们才开始意识到改进软件产品还可以实现哪些功能。并且，他们经常会要求进行改进。

这是否是软件业中的一个问题？当然是。无论软件产品有多“软”，改变现有产品一般都非常困难。但是，无论有多少难题，这都是提供给软件用户的一项重要服务。我们将在事实 43 中讨论这一现象是否是个问题。

在改进占 60%的前提下，其余 40%的维护成本怎么花。最惊人的是，我们在错误修订方面花的钱非常少。多项研究表明，错误更正在软件成本中占很低的比例——仅为维护成本的 17%。虽然我们讨论了有错误的软件，但是从维护成本的比例看，软件产品不应该是易于出错的。

好，60%+17%=77%。其他维护成本怎么花？18%用于所谓的适应性维护，即在改变工作环境时保证软件正常运行，例如在另一台计算机上运行、在另一种操作系统上运行、与新的软件包交互、引入新设备等。注意到用于适应性维护 18%的维护成本仅略多于用于错误更正的 17%。实际上修订错误的成本非常低。

那么其他 5%的维护成本怎么花？用于“其他方面”。很有意思的是为了使软件更容易维护所做出的维护工作（描述这一活动的老术语是预防性维护[*preventive maintenance*]，新术语是重构（*refactoring*）[Fowler 1999]）。

现在，我总结这里讨论的两个 60%的事实，我称之为软件维护的 60/60 规则：

60/60 规则：60%的软件成本用于软件维护，维护成本的 60%用于功能增强。因此，增强旧软件是个大问题。

争议

人们经常会忘记 60/60 规则中的第一个 60%，对第二个 60%则忘记得更多。即使学识丰富的软件专家说到软件维护成本过多时，似乎主要指的也是修改原始产品中错误的成本。一位业内领袖在解释一条商业管理口号时，甚至说要“取消”软件维护，似乎维护是一件坏事。我们将在下一个事实中看到，增强是一件好事。60%的资金份额用于向老产品中增加特征，这只有在软件界中才有可能。

来源

同事实 41 的来源。

参考文献

➥ Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code.* Boston: Addison-Wesley. *

事实 43	维护是解决方案，而不是问题。

1. 讨论

我非常喜欢这个事实！它充分指出了有关软件维护的重要特性（说实话，我认为这是一个观点，而不是一个事实，但是我希望前面的事实能让你认可这一观点就是事实）。

很多人认为软件维护是一个问题，应该减少或“取消”。这种说法表现了他们的无知。假设软件维护只是修正错误，那么软件维护才是个问题。我们已经看到事实远非如此。

实际上，只有维护才能解决在软件中独有的一个问题，即“我们已经构建了一个东西，但是现在需要一个稍微不同的东西。”对于有形的产品，做出这个选择很难。如果你曾经翻修过一座房子，你会明白改变一个有形的产品有多少不可预知的

* 该书中文版《重构——改善既有代码的设计》已由中国电力出版社出版。——编者注

困难（一个翻修工曾告诉我，在争吵激烈的翻修过程中，他的许多客户都离婚了，我当时决定自己决不翻修房子）。相反，修改软件实现新功能相对容易（注意关键词相对［comparatively］，修改软件也并不简单，只是比修改有形的产品容易些）。

争议

对于该事实的争议是对前两个事实的争议积累的结果。等到了这种问题普遍出现时，对于该事实（及隐含问题）的争议可能会比其他问题多。但是争议通常表现了同一个主题两种相对不同的观点，两种观点都有可能是正确的。在此，认为软件维护是个问题这种观点确实不对。

来源

同样，该事实的来源与事实 41 和 42 相同。在下面的材料中有一章非常详细地讨论了该事实。

➥ Glass, Robert L. 1991. *Software Conflict*. Englewood Cliffs, NJ: Yourdon Press.

事实 44	比较软件开发和软件维护中的工作，除了维护中“理解现有的产品”这项工作之外，其他工作都一样。这项工作占据了大约 30%的维护时间，是主要的维护活动。因此可以说维护比开发更难。

讨论

所有上述 60/60 的材料建议我们应该高度重视软件的增强/维护这一重要阶段。并非所有的实践者或者学者都重视，但是那些重视的人已经获得许多重要的成果，最重要的成果是将软件生命周期中的维护工作分解为几个阶段。毕竟，从工作的角度详细考虑软件维护花销的去向非常重要。

研究证明，软件维护的生命周期几乎与软件开发的生命周期完全相同。分析问题的需求是更正还是增强；设计一个基于现有产品设计的解决方案；编码实现该方案并将其插入现有产品中；测试已完成编码的方案，不仅要注意确保新增部分的工作正常，而且要确保其不影响原有部分的正常工作；然后将修订的产品投入使用，并开始新的维护。

浏览上述任务名字可以看出维护与开发仅有细微差别，但是我们疏漏了一个重要的问题，该问题存在于看似正常的阶段，即：在现有产品设计的基础上设计一个方案。这项任务比你想像的匆匆读一遍代码段难得多。实际上，研究数据表明，理解现有产品是软件维护中最难的任务。

为什么？在某种意义上，答案基于前面所述的几个事实：从需求转入设计时的需求激增问题（事实 26）；对于许多问题不存在惟一的设计方案（事实 27）；设计是复杂的、迭代的过程（事实 28）；问题的复杂性每增加 25%，解决方案的复杂性就增加 100%（事实 21）。这些事实的组合告诉我们设计是困难的、智力性的，甚至是创造性的（事实 22）过程。在此，我们讨论的是所谓的逆设计（undesign）——从已有的产品中得到设计方案的逆向工程，其难度至少与原始的设计任务相当。

现有的产品难以理解或逆设计还有一个原因。原始设计者构造了一个设计架构，即解决问题的框架，架构在开发阶段是已知的。但是更正错误或者增强功能时会生成一组新的需求。这些新需求未必适合原来的设计架构。如果适合，修改工作会比较容易。如果不适合，修改工作就比较难，甚至不可能。

别忘了软件开发生命周期的 20-20-20-40 现象——20%需求，20%设计，20%编码，40%错误消除。虽然维护生命周期与此相似，但是有一个重大区别。下面是 Fjelsted 和 Hamlen 对维护生命周期的研究结果（1979）：

- 定义和理解修改——15%
- 评审产品的文档——5%
- 追踪逻辑——25%
- 实现修改——20%
- 测试和调试——30%
- 更新文档——5%

下面我们把这些工作和开发生命周期关联起来进行比较：

- 定义和理解修改（15%）类似于需求定义（20%）
- 评审和追踪（30%）是逆设计，类似于设计（20%）
- 实现（20%）类似于编码（20%）
- 测试和调试（30%）类似于错误消除（40%）
- 在开发过程中，更新文档（5%）通常不作为一项独立任务

注意，虽然测试和调试在维护生命周期中（在开发中也一样）占据了较大比

例，但是这里引入一个新的阶段，即逆设计。逆设计占维护生命周期的30%，因此是维护中最重要的阶段。当然这与初始设计活动截然不同。

是否其他人也赞同逆设计很难这一观点？美国空军网站在1983年做了一项调查，发现“软件维护的最大问题”是“频繁的[人员]流动”，占8.7（幅度为10）。随后处于第二和第三位的是“理解和文档缺乏”和“确定修改位置”，分别占7.5和6.9。维护先驱者Ned Chapin也在1983年说过“理解是维护中最重要的因素。”虽然这些发现都很古老，但是这么多年来，这些事实一直没有改变。

在软件界通常认为软件维护贬低了软件开发者的智商，不值得做。我希望这一数据能使你明白事实并非如此。实际上，软件维护是一项非常复杂的工作，认为软件维护不值得做是非常错误的观点。因为软件维护需要研究他人开发活动的技术核心，所以是既苦又累的工作。因为这个原因，并非所有的开发者都喜欢这项工作。但是它绝非一项不值得做的工作。

注意，在维护生命周期中，文档活动占较小的百分比。维护者用5%的时间评审文档，又用5%的时间更新文档。如果考虑这些数字，你可能会认为它们都小得惊人。如果说逆设计是维护中的主要工作，那么读产品的设计文档或维护文档是不是一项最重要的工作？如果你认为是，那么你可能是正确的。但是……

在此有必要说：维护是软件工作中不受重视的工作之一。你可以确定不受重视的原因是不值得，还是因为其复杂性。但是，不受重视所导致的一个后果是缺少维护文档。常识告诉我们设计文档在构建产品时生成，是逆设计的重要基础。但是在这里常识可能是错误的。软件产品完成之后，程序不断偏离原始的设计规范。正在进行中的维护使设计规范和产品之间的偏离更大。实际情况是到了软件产品的维护阶段，设计文档几乎完全不可信。因此几乎所有的逆设计都要阅读代码（代码总在更新），而忽视文档（文档总不更新）。

那么“更新文档”是怎么回事？有一个类似的问题。在许多人看来，如果文档已经不可信，那么为什么还要维护它？不用管那些了解此事的人，Wiegers（2001）反驳了这种说法：“不要把你所在的坑挖得更深。”这些问题的关键在于我们的老对手——时间表压力。对于产品修改的要求太多，而使得维护者（将下一次维护中更深入的修改留待以后完成）不能安排时间来修订文档。

这样的结果是维护文档成为软件产品中支持最少的部分。实际上，如果你查看一个典型软件项目的发行列表，维护文档甚至可能不在其中，可能也不包括早

期软件生命周期中的修正文档，例如设计文档。

也许维护只是短期工作，但是因为上述原因，软件维护比软件开发更难。人们都不愿意听到这一点，所以我只能小声说。

但是有一个人说出了类似的观点，而且值得一听。这个人在回想他如何更好地编程时说："成为程序员最好的方法是写程序，并研究别人所写的优秀的程序……我到了计算机科学中心的垃圾筐，找到了他们操作系统的列表。"这个重视理解他人程序的人是谁？你可能也听说过，他的名字是 Bill Gates（比尔·盖茨）。因为本书介绍了许多先驱如何理解自己的工作，所以深受读者喜爱。

争议

没有做过大量软件维护的人可能认为该事实有悖于直觉。但是人们对软件生命周期中的维护阶段很少感兴趣，以至让我怀疑很少有人听说过这一概念，更不要说对维护生命周期形成观点了。因此，虽然许多人不相信该事实，但是对于它的争议很少。

来源

有关软件维护生命周期的数据来自 IBM 一个实验室的早期研究（20 多年以前），发表于 Guide（IBM 的一个用户组学术组织）学术会议的学报中。我不知道从那以后是否还有学者再研究过这一问题。

从 Wiegers 等人那里，可以看到一些有关软件维护文档的思想，见下面的"参考文献"一节。

参考文献

- Fjelsted, Robert K.,and William T. Hamlen. 1979. "Application Program Maintenance Study Report to Our Respondents." Proceedings of Guide 48, The Guide Corporation, Philadelphia.
- Glass, Rebert L. 1981. "Documenting for Software Maintenance: We're Doing It Wrong." In *Software Soliloquies*, Computing Trends.
- Lammers, Susan. 1986. *Programmers at Work*. Redmond, WA: Microsoft Press.
- Wiegers, Karl E. 2001. "Requirements When the Field Isn't Green." *Software Test and Quality Engineering*, May.

事实 45	更好的软件工程开发导致更多而不是更少的维护。

讨论

在一系列惊人的事实之后，我们将讨论可能是最惊人的事实，以此来结束软件维护这一话题。当我初次听到这一事实时也感到惊奇。实际上，发现该事实的人也感到惊奇！Dekleva 在 1992 年从维护的角度研究了采用“现代开发方法”对于软件项目的效果。

什么是所谓的“现代开发方法”？包括面向结构或过程的软件工程、面向数据的信息工程、原型化方法和 CASE 工具等，换句话说，是一系列的方法、方法论、工具和技术。实际上，有些发现是可以猜测到。采用这些方法构建的系统比采用老办法构建的系统更可靠，需要的更正更少。但是这些系统需要的维护时间更长。这是怎么回事？

Dekleva 仔细考虑该现象，最后得出上述明确答案。这些系统需要更长的维护时间，原因是人们对它们的修改更多。修改更多的原因是增强构建良好的系统的功能更加容易。多数组织对于其软件产品而言，都有很多有待增强的预留功能（*Software* 1988）。与构建糟糕的系统相比，构建良好的系统可更快地实现这些预留功能的增强工作。

这个有趣的例子证明了“维护是一种方案，不是一个问题”（事实 43）。如果我们把维护活动视为一种方案，那么维护越多越好。假如我们执意认为维护是一个问题，那么就无法将维护活动的增加视为一件好事。

争议

几乎没有人知道这一事实。假设他们知道，就会有强烈的争议。不说别的，该事实有悖于直觉的本质也会引起争议。因为这些争议会迫使我们研究其中的重要真相，所以争议对于软件业非常有益。

来源

有关来源见下面的“参考文献”一节。

参考文献

- Dekleva, Sasa M. 1992. "The Influence of the Information System Development Approach on Maintenance." *Management Information Systems Quarterly*, Sept.
- *Software.* 1988 (Jan.). This issue of *Software* magazine (now defunct) listed the backlog of maintenance activities as 19 months for PC programs, 26 months for minicomputers, and 46 months for mainframes. Although this is clearly dated, I know of no more recent data.

第3章

质　量

质量（quality）是一个难以捉摸的术语，软件的质量更难以捉摸。Pirsig（1974）在《Zen and the Art of Motorcycle Maintenance》一书中，着重讨论了质量这个难以捉摸的问题。本章从学术的角度讨论这一术语的真实含义，疯狂地寻找一个可行的定义。

无论我们如何得意地看待那种疯狂（当然我们没有人为此疯狂），实际情况是我们对于软件的质量问题知之甚少。对于软件我们认为“一看见就应明白”，但实际上我们对于质量的定义以及谁对产品的质量负责并没有一致的意见。假设我们有了一个一致认可的定义，还需要确定如何衡量软件产品的质量。我们将逐一讨论这些问题。

质量有没有可行的定义？在软件界对于质量的定义有很大的争议，更糟糕的是有些业内人士认为对于质量没有一致的看法，还有人支持错误的定义。在本书的事实 46 中，我给出自己推崇的定义（质量是七项属性的集合），然后在事实 47 中，还列出我认为错误的一些定义。声明：你可以不同意我的这些观点。

质量是谁的责任？许多有关软件质量的书籍和课程都认为质量是一项管理任务。但是，如果你愿意看看我将质量定义为一组属性，那么你会发现许多高深的技术问题。这些属性之一是可修改性，涉及到如何构建可修改的软件。可靠性涉及到如何降低软件中含有错误的概率，以及选用各种有意义的错误消除方法。可移植性指构建易于在不同平台之间移植的软件。质量的这些属性都是深入的技术问题，要想得到较高的质量必须有深厚的技术知识。基于这些原因，我敢断言质量是软件业中技术性最强的要素，管理者的工作是为技术人员提供便利、消除障碍，这与承担保证质量的责任相距甚远。

我们为什么不能度量质量？因为不仅质量本身不可捉摸，而且事实 46 中构成软件的各个属性也同样不可捉摸。对于诸如可理解性、可修改性、可测试性等属

性几乎不可能打分。虽然我们可以给可靠性打分，在一定程度上给效率打分，但是这不能改变这一事实，即这些衡量质量的属性难以捉摸。多年以前，美国国防部资助了一项有关衡量软件质量属性的研究（Bowen、Wigle 以及 Tsai，1985）。研究得出含有许多报表和选项的三卷报告。不幸的是，等这些工作彻底结束后，你依然不能准确地对软件质量进行定量分析。

那么，下面有关软件质量的材料将讨论哪些内容？

- 努力给出质量的正确定义和不正确的定义。
- 概述可靠性的有关方面，以及错误和错误制造者的特征。特别是我回顾前面一些有关错误消除事实的衍生结论，并充分论述了一些衍生结论。
- 概述效率的有关方面。我们将从随后的事实中看到效率的重要性。几十年以前一些有关效率的老教训值得列为事实。
- 敏锐的读者可能注意到我只列出七个属性中的两个，并重点强调。这并不是忽视其他的属性，只是因为对应于这两个属性存在着经常被忘记的基本事实（毕竟，这是本书的主题）。

来源

除了参考文献中所列的资料外，还有：

- Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall. 该书第 3.9.1 节“State of the Theory”也分析了本书前面提到的 DoD 报告。

参考文献

- Bowen, Thomas P., Gary B. Wigle, and Jay T. Tsai. 1995. “Specifications of Software Quality Metrics.” RADC-TR-85-73, Feb.
- Pirsig, Robert M. 1974. *Zen and the Art of Motorcycle Maintenance*. New York: Morrow.

3.1 质量

事实 46	质量是一组属性的集合。

讨论

对于软件质量有许多不同的定义。我在此给出最经得起时间考验的定义。

软件质量是软件产品七项属性的集合：可移植性、可靠性、效率、可用性（人类工程学）、可测试性、易理解性以及可修改性。虽然不同的人提出不同的属性集合，但是近三十多年来大家普遍接受上述说法。

这些属性具体是什么？

1. 可移植性是指生成易于在不同平台之间移植的软件产品。
2. 可靠性是指软件产品满足预期要求，值得信赖。
3. 效率是指软件产品在运行时间和空间消耗上的经济性。
4. 人类工程学（又称为可用性）是指软件产品用起来既容易又舒服。
5. 可测试性是指软件产品易于测试。
6. 易理解性是指维护者易于理解软件产品。
7. 可修改性是指维护者易于修改软件产品。

我从来没有按照任何优先顺序来排列这些质量属性。实际上这没有任何意义，也就是说，人们不应该按照通用的、惟一正确的顺序来获得各项软件质量属性。然而，这并不是说不应该对这些属性排序。在项目开始时，确定这些属性的优先顺序非常重要，例如，如果产品将运用于许多种平台，那么应该将可移植性放在列表的首位或者重要的位置。如果软件所控制的产品生死攸关，那么应该将可靠性放在列表的首位。如果产品的使用时间将很长，那么很可能需要将易理解性和可修改性等维护属性放在列表的首位或者重要的位置（应该注意，在上述七个属性中有两个与维护明显相关）。如果产品运行在资源缺乏的环境中，那么应首先考虑效率。

例如，对于一般项目通常采取以下的优先顺序：

1. 可靠性（如果产品不能正确工作，那么其他属性都没有意义）。
2. 人类运行工程学（近来人们非常重视 GUI，这说明可用性的重要性）。
3. 易理解性和可修改性（任何有价值的软件产品都有可能需要较长时间的维护）。
4. 效率（对于效率的排位我有些犹豫。在有些应用中效率位于首位）。
5. 可测试性（位居倒数第二位并非降低其重要性，可测试性直接影响居于首位的可靠性）。

6. 可移植性（对于许多产品根本不用考虑可移植性，但是对于某些产品应首先考虑可移植性）。

如果上述顺序与你的顺序不一致，不必大惊小怪。在我写第一本有关软件质量的书（Glass 1992）时，该书的一些复审者坚持调换我（随意）采用的顺序；他希望按照他自己的顺序排列，而他的顺序恰好与我的顺序大相径庭。我认为排列质量属性的通用顺序就像生成一个优秀的设计一样：如果有两个人同意，那么就算他们说对了。

争议

下面是几项关于该事实的争议：

1. 质量的正确定义是什么？
2. 正确的属性列表是什么？
3. 是否有正确的属性列表顺序？

有关争议 1，许多软件人员（包括一些学者）认为“该定义不正确”。其中许多人采用我在事实 47 中所列的定义。在事实 47 中，我将分析为什么这些人是错误的。

有关争议 2，有些软件人员不同意我的属性列表。例如，一位软件专家强烈反对包含可移植性，理由是其他产品的质量属性中不包括可移植性。我敢说这是一个非常有意思的错误观点。例如，汽车的一个重要属性是“舒适性”。该属性对于包括软件在内的许多产品都没有意义。还有一些人对我的列表中的属性有不同的叫法。我认为这争议不大，只要它们所表达的概念包含在我的定义之中，我不在乎你怎么称呼它们。

有关争议 3，我们已经讨论过不存在通用的顺序。

来源

将质量定义为属性集合的最早、最知名的人是 Barry Boehm。

➥ Boehm, Barry W., et al, 1978. *Characteristics of Software Quality*. Amsterdam: North-Holland.

对于 Barry 的研究有许多深入的论述，我最喜欢的是“参考文献”一节中的材料。将质量定义为属性的更早的例子是：

➥ McCall, J., P. Richards, and G. Walters. 1977. “Factors in Software Quality.”

NTIS AD-A049-015, 015, 055, Nov.

参考文献

➥ Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall.

事实 47	软件质量不是用户满意、满足需求、满足成本和时间表目标，或者可靠性。

讨论

对于软件质量中质量一词的定义非常多，以至于让我感到绝望。我绝望的理由是错误的定义太多了，而这些定义的倡导者都相信其正确性。

在这一事实中列出 4 种替代定义，每一个都非常吸引人，都涉及到软件业的重要方面。但是，我肯定它们都不是正确的质量定义。

长期以来，我一直相信其他的定义有问题，但是我不能明确指出问题之所在。直到听到 Computer Sciences Corp. 的一位讲演者讨论这些术语的关系，我顿悟其他定义之错误所在。该讲演者给出的关系如下：

用户满意 = 满足需求 + 按时提交 + 适当的成本 + 产品质量

这当然是对于用户满意的直观定义。如果产品满足需求、如期交货、成本低并且质量较好，那么用户就满意。但是，如果你仔细分析就会发现，这些实体是相互独立、可区分的。注意，质量即是这些实体之一。也就是说（我敢断言），质量与其他实体有着显著的区别。

注意，这些都是非常重要的实体。我们说质量不同于其他实体，并不是贬低其他实体的重要性，只是说质量是完全不同的实体。满足需求、按时提交、适当的成本都非常重要，但是都不是关于质量的。用户满意与质量有关，但是还与其他重要的因素有关。

实际上，这涉及到的许多东西并非质量本身。然而，有一个例外，即可靠性。许多软件专家认为质量就是产品中是否存在错误。我们从事实 46 得知质量当然与错误有关，其中的可靠性与错误有关，但是质量还与其他的许多问题有关。

这些认为质量就是可靠性的人都很清楚明白。在讨论中，他们还承认质量的

其他属性，但是讨论一结束，他们又会继续讨论错误，似乎错误是质量的惟一内容。因为可靠性是软件质量的重点内容（在事实 46 中，我将它排在第一位），所以将质量和可靠性混同的做法很能迷惑人。

争议

这个事实自身就有争议。你可能还会听到有人说质量是用户满意、满足需求或达到估算（怎么会有这东西？我认为它和质量完全无关。我不明白它为什么会出现在这个序列中）或者可靠性。这些讨论都有坚定的信念支持。即使这样，也不能改变它们错误的本质。

来源

我不愿意提供这些对于软件质量的错误观点的出处，理由如下：（a）这样做会激发那些错误观念；（b）我必须提供一些人的名字，而你可能认识他们。重要的是，我们都应该接受质量的正确定义，忽视其错误定义。只有这样，对于软件质量的讨论才有意义。在此之前，很难寻找到有意义的方式对软件质量这一主题进行讨论。

3.2 可靠性

事实 48	绝大多数程序员都会犯某些错误。

讨论

有些软件错误比其他错误更常见，这一点也不奇怪。曾经参加过代码审查的人可能这样说，“噢！又是一个那种（类型）的错误。”实际上，人们总是容易犯某几种错误，比如定义/引用不一致、忽略了详细的设计细节、不能初始化一个常用的变量、忽略一组条件中的一个条件。

奇怪的是，几乎没有研究者深入探讨这一现象。该事实除了来源于我个人的经验以外，还来自一位德国研究者在 Bremerhaven 召开的一个鲜为人知的学术会议（Gramms 1987）论文中。该学者将它们称之为“偏好性错误”，他还认为这些错误源于“思维陷阱”。这很奇怪，因为你认为这些偏好性错误可能是错误消除的

重点对象，也会被列入审查的重点。人们还会开发出专用工具来发现和分离它们，并对它们进行重点测试。学者们还会研究出其他的方法来避免或者发现它们。

有关偏好性或者常见错误，还有一点很有意思。在容错编程（fault-tolerant programming）（能够编写捕获软件本身的错误）中，有一种叫 N-Version 编程的概念。N-Version 的基本思想是：对于同一个问题，N 个独立的编程团队的 N 种不同方案不会重犯同样的错误；这些方案相互联系，如果一个方案的结果与其他不一致，就可以发现其中的错误。但是错误偏好性这一事实说明，在 N 个方案中确实不止一个包含了同样的错误，因此 N-Version 方法的效能并不很强。在软件开发的可靠性极度重要的软件（例如，航空、铁路系统）中，也是同样一个问题。

争议

该事实引出的现象不会使多少软件人士吃惊。因为该事实并不为人所知，所以不会引起任何争议。

来源

因为我曾在 German Computing Society 学术会议（见随后的“参考文献”一节）中发言，所以我了解该会议。我综合了一些关于 Gramm 的发现的文章，如下：

- Glass, Robert L. 1991. “Some Thoughts on Software Errors.” In *Software Conflict*. Englewood Cliffs, NJ: Yourdon Press. 前面提到的 German Computing Society 的演示文档就包含这部分内容。

参考文献

- Gramms, Timm. 1987. Paper presented on “biased errors” and “thinking traps.” Notices of the German Computing Society Technical Interest Group on Fault-Tolerent Computing Systems, Bremerhaven, West Germany, Dec.

事实 49	错误通常聚集在一起。

讨论

看下面有关错误位置的陈述：

- “半数的错误发现在 15%的模块中”（Davis 1995，引用 Endres 1975）。
- 80%的错误发现于仅仅 2%（sic）的模块中（Davis 1995，引用 Weinberg 1992）（你可能认为 2%是一个打印错误，后面将给出此引用）。
- “大约 80%的错误发现于 20%的模块中，大约半数的模块中没有错误”（Boehm 和 Basili，2001）。

无论真实数字是多少，错误明显集中出现在软件产品中。注意，在几十年以前，人们就知道了该事实——1975 年 Endres 的研究。

为什么错误会如此集中？是不是因为程序中的有些部分比其他部分复杂很多，而这种复杂性导致了错误（我认为如此）？是不是因为将编码任务分配给多个程序员，而有些程序员犯的错误比他人多，或者发现的错误比他人少？（这很有可能，想想我们在事实 2 中提到的个体差异。）

该事实的中心思想是什么？如果你在程序的某些模块中发现错误异常多，那么就应该继续在此找错误。这里的错误可能更多。

争议

这里的数据非常清楚，并经受了时间的考验。据我所知，对于该事实没有争议。

来源

支持该事实的来源见随后的“参考文献”一节。

参考文献

- Boehm, Barry, and Victor R. Basili. 2001. “Software Defect Reduction Top 10 List.” *IEEE Computer*, Jan.
- Davis, Alan M. 1995. *201 Principles of Software Development*. New York: McGraw-Hill. Principle 114.
- Endres, A. 1975. “An Analysis of Errors and Their Causes in System Programs.” *IEEE Transactions on Software Engineering*, June.
- Weinberg, Gerald. 1992. *Quality Software Management: Systems Thinking*. Vol. 1, section 13.2.3. New York: Dorset House.

事实 50	没有惟一最好的消除软件错误的方法。

讨论

同一句话翻来覆去地说！我在前面几个事实中反复论述了这一观点。我在此重复是因为它本身就是一个事实，而不是其他事实的一部分。

该事实的基本思想是什么？是说没有消除错误的银弹，永远也不会有。无论哪种测试都不够；无论谁定义的评审和检查方法也不够；不论你是否喜欢正确性验证，它也不够；虽然容错性对于关键性软件意义重大，但是还不够。不论你喜欢哪种错误消除方法，该方法都不够。

争议

有关争议主要来自于鼓吹者。鼓吹银弹的人还会夸大他们叫卖的错误消除技术。实际上，这些鼓吹者永远是错误的，但是这不能阻止他们在将来重蹈覆辙。

来源

该事实是下面这本书的中心思想：

➥ Glass, Robert L.1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall.

事实 51	总会有残存的错误。我们的目标应该是消除严重错误，或者使之最少。

讨论

又是重复！但是，我还是要说，重述该事实是因为它本身就是一个事实，而不依附于其他相关事实。

即使经过了最严格的错误消除过程，在软件中通常还会有残存的错误。我们的目标是使错误数量最少，特别是使残存错误的严重性降至最低。

讨论软件错误这一话题时，很有必要引出错误严重性的观点。我们应该消除软件中的严重错误，最好也消除其他错误（例如文档错误、冗余代码错误、不可达路径错误、算法中次要数据的错误等等），但是这通常不是必需的。

争议

对于软件产品中是否总有残存错误没有争议。但是有关该现象是否持久却有

极大争议。现实主义者（有人称之为悲观主义者，甚至辩证者）相信这种情况不会改变（因为我们从讨论过的各种复杂性因素就可以了解）。理想主义者（有人称之为幻想主义者，或者鼓吹者）相信如果按照非常严格的过程，无缺陷软件近在咫尺。

最近一项研究（Smidts、Huang 和 Widmaier，2002）很好地说明了该问题。两个采用完全不同的软件开发方法（一个团队使用传统方法，处于 CMM4；另一个团队采用前卫的形式化方法）的团队都不能构建一个可靠性达到 98%的简单产品（虽然都能满足时间表安排和“慷慨的”费用）。

至此，你可能已经决定自己倾向于争议的哪一方，我不再细说了。

来源

该事实是下面一本书的另一个中心思想：

- Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall.

下面是涉及该事实的一些有趣的引用。

关于残存错误：

- “有经验的个人实践可以使错误的发生减少 75%”（Boehm 和 Basili，2001）。
- 大约 40%～50%的用户程序包含着较显著的缺陷”（Boehm 和 Basili，2001）。（注意，该引用也涉及到缺陷的危害性。）
- “你不会发现所有的 bug”（Kaner、Bach 和 Pettichord，2002）。

关于残存的严重错误：

- “小于 10%的错误导致 90%故障的发生”（Boehm 和 Basili，2002）。

参考文献

- Boehm, Barry, and Victor R. Basili. 2001. “Software Defect Reduction Top10 List.” *IEEE Computer*, Jan.
- Kaner, Cam, James Bach, and Bret Pettichord. 2002. *Lessons Learned in Software Testing*. Lessons 9,10. New York: John Wiley & Sons.
- Smidts, Carol, Xin Huang, and James C. Widmaier. 2002. “Producing Reliable Software: An Experiment.” *Journal of Systems and Software*, Apr.

3.3 效率

事实 52	效率主要来自于优秀的设计，而不是优秀的编码。

讨论

多年来，永远乐观的程序员通常认为自己的编码方式有助于获得高效的产品。这也是汇编语言长盛不衰的原因之一。我们将在事实 53 中讨论汇编语言。在此，我们将讨论编码和设计的关系。

该事实的意义在于讨论导致了软件产品低效率的原因是什么。有关原因包括：来自外部的输入/输出（I/O）（例如，低速数据访问）、笨拙的接口（例如，不必要的或远程过程调用）和软件内部时间开销（例如，无谓的逻辑循环）。

我们首先来解决 I/O 引起的低效率问题。计算机在获取和替换外部设备中的数据时速度非常慢。因此在 I/O 操作设计上细微的改进会显著提高应用程序的速度。你可以选择多种数据格式，而你的选择在很大程度上决定程序的效率。我过去经常劝说计算机学者：数据和文件结构课程的重点是针对不同的应用选择效率最高的方法。毕竟，线性或哈希数据结构很简单，为什么人们还要发明链表、树、索引？为什么人们要使用数据缓存，缓存在逻辑上的低效方法事实上却能显著提高效率？因为（我告诉他们）数据结构是增加逻辑复杂性和改善数据访问效率之间的折衷（有些计算机学者认为效率是过去应考虑的事情，他们仅仅将数据结构看作有意思的数据管理方法）。

因此在设计阶段，我们应认真选择正确的数据结构、文件结构或者数据库访问方法。这样，就避免了浪费大量精力去编码实现不适合的数据访问方法。

与低效率的 I/O 相比，接口和内部低效率的问题并不严重。然而，糟糕的循环策略仍然有可能使循环长时间混乱，甚至陷入死循环。也许最糟糕的是数学迭代算法。很慢的、甚至不存在收敛的算法可能浪费许多 CPU 时间。其次是低效率的数据访问（即使访问内部的数据也可能会有问题）。再重复一遍，在设计阶段对效率的细微考虑比漂亮高效的编码更有意义。

这里的底线是什么？如果一个项目需要效率，那么应该在生命周期的早期，即开始编码之前考虑。

争议

对于一些急切的程序员而言，编码是软件构建中最重要的任务。他们都想尽快开始该工作。他们还认为设计只是推迟问题最终实现的活动。

如果这些人一直致力于相对简单的问题，那么他们不会相信自己是错误的，而且他们的做法也没有什么严重的后果。但是对于相对复杂的问题，这种小型化设计、快速编码的方法会失败（回想在事实 21 中有关问题的复杂性可以显著影响解决方案的复杂性）。

在那些认为设计几乎没有意义的人和那些认为设计是编码基础的人之间，对于该事实的争议最多。极限编程运动倡导简单化的设计方法和快速进入编码阶段，这无疑又加剧了争议。极限编程强调：在编码之后通过持续地“重构（refactoring）”来修订设计中的低效率和错误。

来源

这又是一个人们很早以前就知道的事实。因为时间太久远，所以几乎无法追溯到原始出处。近三十多年以来的所有软件工程教课书都阐明了这一点。

想了解有关快速设计并开始编码、随后再通过重构来修订错误的内容，请参考有关极限编程方面的文献：

- Beck, Kent. 2000. *eXtreme Programming Explained*. Boston: AddisonWesley. 请参考关于“简单设计”和“重构”的内容。

对于重构最详细的讨论见：

- Fowler, Martin.1999. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.

事实 53	高级语言（High-order language，HOL）代码配合适当的编译器优化，大约可以达到汇编语言 90%的效率。对于一些复杂的现代体系结构，效率更高。

讨论

在计算机界有关高级语言（HOL）和汇编语言之争由来已久。每一方都尽力想在项目中采纳自己的观点，在很早以前就有了充分的数据可以终结该争议。这方面的来源可以追溯到 20 世纪 70 年代的研究结果。因此，虽然高级语言与汇编

语言之争同应用领域之争同样新鲜，但是当今许多顽固的汇编语言支持者仍旧坚持他们的观点，软件时代的智慧可能足以消除该争议。

权威的数据来源于20世纪70年代开展的一系列研究中。开展这些研究是因为该争议对于应用领域至关重要，即航空电子软件（控制飞行器和航天器电子原件的软件）。Rubey（1978）很好地总结了这些研究的结果，他说："几乎所有的报告表明，在航空电子软件中采用高级语言，效率会降低10%～20%"（后续研究表明，经过编译器优化可以使代码效率至少增加10%。编译器可以很好地优化高级语言程序，与之相比，汇编语言经过优化可以提高2%～5%的效率）。

那么为什么该争议这些年来还盛行？虽然有了上述数据，而且在许多编码中高级语言有绝对的优势，但是在有些情况下选用汇编语言确实更好。换句话说，高级语言的优势与具体任务密切相关，在有些任务中采用高级语言很难得到较高的效率。

高级语言有什么优势？因为这些优势已经广为接受，所以几乎没有必要再说，但是我还是将其列出来：

- 解决同一个问题所需的高级语言代码比汇编语言代码行数少得多，所以高级语言可以显著提高工作效率。
- 高级语言代码智能地解决了容易出错的难点，例如寄存器操作。
- 高级语言代码可以移植，而汇编语言代码通常不能。
- 高级语言代码易于维护（理解和修改）。
- 必要时，高级语言代码可以通过优化的方法来提高效率。
- 缺少经验的程序员也可以采用高级语言代码。
- 高级语言编译器可以根据流水线、缓存等现代体系结构来优化。

汇编语言有什么优势？

- 高级语言语句不涉及硬件特征，而有些汇编代码可以充分利用硬件，比对应的高级语言代码更简单。
- 同样，在汇编代码密集的操作系统方面，高级语言的交互比较麻烦。
- 严格的时间和空间要求以效率为根本出发点。

Prentiss（1977）研究了在严格限制的领域采用汇编语言的优点。这个项目最初完全采用高级语言编码实现，后来发现存在效率问题，就采用汇编语言改写了20%的代码。这种80/20的比例很好地指示了任何系统中汇编语言的最大含量，即使现代系统也是如此。

争议

似乎有关的争议永远不会消逝！汇编是一种非常诱人的技术，哪位真正的程序员不愿意真正亲密接触自己的计算机和操作系统。但是，该争议已经在 20 世纪 70 年代很好地解决了。问题是，当今的许多程序员不懂得历史上的航空电子学研究。

来源

有关该事实的来源见随后的“参考文献”一节。

参考文献

- Prentiss, Nelson H., Jr. 1977. “Viking Software Data.” RADC-TR-77-168. Portions of this study, including those dealing with HOL versus assembler considerations, were reprinted in Robert L. Glass, *Modern Programming Practices* (Englewood Cliffs, NJ: Prentice Hall, 1982).
- Rubey, Raymond. 1978. “Higher Order Languages for Avionics Software—A Survey, Summary, and Critique.” Proceedings of NAECON Reprinted in Robert L. Glass, ed., *Real-Time Software* (Englewood Cliffs, NJ: Prentice Hall, 1983). This study references the other studies about the efficiency of HOL versus assembler.

事实 54	在空间和时间之间存在折衷。通常，改进一方面会降低另一方面。

讨论

似乎有什么东西既可以提高程序的时间效率，又可以提高其空间效率。但是，这不可能。

例如，考虑简单的三角函数。大多数函数都根据算法编码实现，它们的代码占用较小的空间，但是算法的迭代特征决定了必须有较长的执行时间。一种替代的实现方法是在程序中建立一个函数值列表，只需要简单插值就能得到函数值。插值算法当然比迭代算法简单得多，但是列表所占的空间远远大于迭代代码（我

曾经在一个空间项目中使用该方案，在该项目中对于空间要求不严格，但是对时间要求非常严格）。

另一个更现代的例子，考虑 Java 编程系统。Java 代码并不编译成机器代码，而是字节码（byte code）。字节码比对应的机器代码紧凑得多，因此 Java 程序的空间效率很高。但是时间效率如何？很差！因为那些字节码不是机器代码，因此必须先解释再执行，而解释有时会使执行时间增加 100 倍。

因此，有关效率的研究也是一种折衷。很少有时间效率和空间效率同步提高的例子（有意思的是，Rubey[1978]曾在研究高级语言效率时表明：空间效率比时间效率更容易度量，前者可以静态地度量，而后者必须动态地度量。因此，无论如何，描述空间效率比时间效率容易）。

这里的底线是什么？如果讨论质量的效率属性，那么你应该牢记自己关注哪种效率？

争议

这是一个少有的不言自明的事实，对此争议很少。许多关注属性的人都清楚地知道自己究竟在意什么属性；而那些不知道的人不会考虑到这一点，还可能会犯一些致命的错误。

来源

有关该事实的来源见随后的“参考文献”一节。

参考文献

➥ Rubey, Raymond. 1978. “Higher Order Languages for Avionics Software—A Survey, Summary, and Critique.” Proceedings of NAECON. Reprinted in Robert L.Glass, ed., *Real-Time Software* (Englewood Cliffs, NJ: Prentice Hall, 1983).

第4章

研　究

你可能对于本书讨论研究这一话题感到意外。本书要讨论的问题与软件工程实践有关，包括管理、生命周期和质量。为什么我要用珍贵的55个事实中的一个来讨论研究？因为我谦虚地认为21世纪（回到20世纪也一样）的软件工程研究有问题。研究成果对软件实践的帮助太少，远远少于应有的帮助，而且实践中有许多东西非常需要研究。

现在我认识到研究的存在并不仅仅是为了帮助实践。我承认单纯理论性的、不考虑实践的研究很重要，但是实践中的有些东西非常需要理论的帮助，却从未得到（这使我想起Reilly和Maloney的二重唱："I Don't Know What I Want from You, but I Ain't Gettin'It'Blues。"但是在此，软件实践确实知道自己需要什么，不需要什么！）。

实践需要理论的帮助来理解哪些新技术真正对实践者有益，以及益处有多大。当前的情况是，本书多处提到的鼓吹者左右我们对于这些新技术所带来益处的认识。更糟的是，有时候这些鼓吹者中包含有学者，他们一起鼓吹自己喜欢的新的（或者旧的）观念。

很早以前，我就说过在软件的理论界和实践界存在沟通隔阂。因为研究者对于所鼓吹概念的实际情况没有把握，所以加深了这种隔阂。

幸运的是，研究者全部致力于长期的、纯理论性研究，在一些实际问题上投入的时间和经费较少。

关于这一事实的讨论已经够多了。让我们了解一下。

来源

➥ Glass, Robert L. 1977. "Philosophic Failures of the Field in General." In *The Universal Elixir and Other Computing Projects Which Failed.* Computing

Trends. Reprinted in 1979, 1981 and 1992.

事实 55	许多软件研究者不是做调查，而是鼓吹。因此，（a）有些概念比鼓吹的糟糕、更少；（b）缺少有助于确定这些概念真实价值的评估性研究。

讨论

有许多研究方法。研究可以是非正式的、现象观察或者报导已有的文献；研究也可以是建议性的，提出改进方法；研究还可以是分析性的，可以分析例案研究或理论；研究还可以是评估性的，评价方法、模型或理论（Glass 1995）；当然，研究还可以混和采用多种方法（从机构组织角度考虑，学术和产业界的研究机构都会开展研究。在此，我主要指的是学术研究）。

在软件研究中有一个问题，即评估性研究很少。在各种研究中仅14%的软件工程研究基本上是评估性的。在计算机科学研究中仅11%是评估性的。相比而言，其他主要的计算机领域，例如信息系统，主要采用评估的研究方法，占67%（Glass、Vessey 和 Venkatraman 2003）。

缺少评估所导致的后果是：研究不能帮助这个行业理解各种构建软件方法的优劣。再加上行业中的大肆鼓吹者，这使一些问题变得非常难。那些应该帮助我们区分好与坏的人都没有尽到应尽的责任。

我曾经强调过该问题（Glass 1999），我说到："少数优秀的评估性研究点亮了黑暗的宇宙。"在那个研究中，我随意察看以前对于新技术的评估性研究情况，结果发现没有材料能够证明哪些技术具有鼓吹的那种突破性的益处；但有意思的是，我发现一般益处都不大（这些技术包括 CASE 工具、第四代语言、面向对象和形式化方法，等等）。

近几年来，许多人努力呼吁重视该问题。Potts（1993）称这种错误的研究方法为"研究，随后转移"，意思是：在建议性研究中提出一种方法，就推荐将该方法用于实践，在二者之间缺少评估性研究。我曾经采用鼓吹性研究来描述同一现象（Glass 1994）。Fenton 及其同事（1994）说到"科学和实质"，意思是软件研究缺少实质性，因此很不科学。Tichy 和他的同事（1995）检查了 400 份计算机研究论文，发现 40%～50%的软件工程和计算机科学论文根本没有评估性成份（相反，他指出在光学工程和神经计算领域中仅 12%～14%的发表论文

没有评估）。实际上，对于改善软件研究的呼吁至少可以追溯到 Vessey 和 Weber（1984），他们当时分析是否有足够的证据证明结构化方法具有所声称的益处（没有找到足够证据）。

鼓吹的问题不仅限于供应商（他们也会做一些虚伪的研究来支持自己的宣传）和实践者。学院的软件研究者经常会提出一些新观念，将这些观念描述地很有效，并指责不尽快在实践中采用该技术的人。形式化的方法就是一个很好的例子。他们在开始做一些评估性研究之前，会夸大自己观念的益处，或者被误导。

争议

关于该事实有很多争议。许多研究者都否认该事实的正确性。他们认为自己的调查比鼓吹多。也许他们不面对业内鼓吹，但是他们不相信这对于自己工作的必要性。在软件研究者的听众中提及该事实的观点，可能会激起更激烈的讨论。这可能是本书中最有争议的问题。

来源

有关该事实的来源见随后的“参考文献”一节。

参考文献

- Fenton, Norman, Shari Lawrence Pfleeger, and Robert L. Glass. 1994. “Science and Substance: A Challenge to Software Engineers.” *IEEE Software*, July.
- Glass, Robert L. 1999. “The Realities of Software Technology Payoffs.” *Communications of the ACM*, Feb.
- Glass, Robert L. 1995. “A Structure-Based Critique of Contemporary Computing Research.” *Journal of Systems and Software*, Jan.
- Glass, Robert L. 1994. “The Software-Research Crisis.” *IEEE Software*, Nov.
- Glass, Robert L., Iris Vessey, and Ramesh Venkatraman, 2003. “A Comparative Analysis of the Research of Computer Science, Software Engineering, and Information Systems.” In review.

- Potts, Colin. 1993. "Software Engineering Research Revisited." *IEEE Software*, Sept.
- Tichy, Walter F., Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. 1995. "Experimental Evaluation in Computer Science: A Quantitative Study." *Journal of Systems and Software*, Jan.
- Vessey, Iris, and Ron Weber. 1984 "Research on Structured Programming: An Empirical Evaluation." *IEEE Transactions on Software Engineering*, July.

第 2 部分

5+5 谬误

简　介

在本书中增加一些谬误的想法是逐渐形成的。开始时，我认为在业内有许多事实被遗忘，或者说应该知道而不知道。我想采用简洁的、直观的方式将这些事实展现给业界。

但是，在查找这些事实、争议和它们的来源时，我经常会接触到自己所谓的谬误。我不谦虚地认为：许多软件人员深信的东西其实是错误的。这些东西不仅包括著名软件人员在很早以前就说过的话，还包括追随者和无主见者，他们根本不去考虑这些话的内容，只知道重复。

起初，我发现了一两个谬误，后来发现更多了。最终，我决定将谬误数量锁定在 10 个。实际上，我将 10 写为“5+5”只是为了美观，这和前面的 55 个事实正好押韵。我相信只要我们用心，谁都可以找到更多的谬误。

有时候，我想自己应该在考虑引出谬误时结束本书。陈述事实是一回事，有些人可能不同意我提出的一些事实，这些人可以忽略它们；而谬误是另一回事，我的谬误可能就是别人的事实。谁眼里的事实一旦被认定为谬误，这个人一定不会善罢甘休！

因为我把一些人眼中的事实认定为谬误，所以在此先向以下人士致歉：

- Tom DeMarco，曾经写过许多软件领域的重要文章或专著，他说过“你不能控制自己无法度量的东西”。这种说法被追随者篡改为“你不能管理自己无法度量的东西”。
- Jerry Weinberg，也是软件业的伟人之一。他有关“忘我编程”的观念是事实和谬误的完美结合，我一直在考虑是否应该将它列为谬误。
- Harlan Mills，也是软件界中的一位重要人物。我将他的“随机测试”列入谬误。不过，我在另一个地方也赞扬了他“先读后写”的思想。
- 开放源代码者，他们称赞“足够多的眼睛可以发现所有的 bug”，但是其中含有许多的问题，我不得不在此称之为谬误。
- 极少的研究软件维护方面的研究者（呀！），他们应该了解实际情况，改

变对问题的理解，否则他们的发现没有意义。

- 计算机学术界，他们教学生在读代码之前就写程序，他们知道这不对，但是似乎不知道如何改正。

所以，本书从现在开始就小心翼翼的。我们将进入别人的心灵圣地，甚至你自己的心灵圣地。注意自己血压的变化。如果你同意我的这些谬误，别人也许不会感谢你。如果不同意，那么你将会觉得非常不舒服。

我已经警告你了！继续吧。

第5章

管　理

谬误 1	你不能管理自己无法度量的东西。

讨论

这种说法旨在指出度量对于管理者至关重要。显然，管理者需要回答诸如费用、时间、程度等问题。整个软件工程界都在研究软件的度量问题，不断还有人强烈建议一些新东西也应该度量，并提出了度量方法。

有意思的是软件度量很少应用于实践。软件管理者采用的各种工具和技术的调查表明，度量工具和方法都是最少采用的。当然，也有例外，有些企业（例如IBM、Motorola、Hewlett-Packard 等）都非常重视度量。但是，大多数企业都忽视了度量方法。这是为什么？也许管理者不相信度量的结果，也许有些度量所必需的数据很难收集。

有许多关于度量价值和成本的研究，其中多数都得到了肯定的结果。例如NASA-Goddard 研究发现：持续收集必需的度量数据的成本小于 3%（数据采集和分析）+ 4%～6%（数据处理和分析）= 7%～9%的项目总成本（Rombach 1990）。NASA-Goddard 认为不论结果的价值如何，这都是一笔不错的买卖。

然而，一些旧的度量方法也受到了影响。起初，管理者收集的数据意义经常不大，或者成本过高。这种被动的度量不仅成本高，而且收效不大。直到建立了GQM 方法（最初由 Vic Basili 提出）之后，度量才有了一定的理性。GQM 是指：建立度量需要解决的目标（Goal）；确定为了满足这些目标应该提出哪些问题（Question）；然后收集必需的度量元（Metrics）结果来回答那些问题。

度量元也是软件科学中的问题。计算机科学的伟大先驱 Murray Halstead 努力建立了软件工程的基础学科（Halstead 1977）。他定义了应该度量的因子以及相应的度

量方法。在当时，这似乎是一项很有价值、很重要的目标。但是，不断有研究得到与计算机科学的数据不一致或者相反的结果。有些研究甚至认为计算机科学等同于占星术。收集软件项目的数据的“科学”方法的口碑很不好，在大多数地方都放弃了该方法。那些还记得软件科学崩溃的人会以同样的方式贬低所有的软件度量活动。

然而，人们经常要收集软件度量元的数据，甚至排出了实践中最常用的“前十名”度量元。为了回答“什么是软件度量元”的问题，我们列出这“前十名”。

软件度量元	使用率（%）
发布后发现的缺陷数	61
修改或者修改请求的次数	55
用户或者客户的满意程度	52
开发过程中发现的缺陷数	50
文档的完整性和精确性	42
确定和改正缺陷所用的时间	40
不同类型/种类的缺陷分布情况	37
主要功能/特征的错误	32
需求规格说明的测试覆盖	31
代码的测试覆盖	31

看一下列在最后五位的度量元也同样有意思：

软件度量元	使用率（%）
模块/设计复杂性	24
提交的源代码行数	22
文档规模/复杂性	20
复用源代码的行数	16
功能点数	10

（这些数据来源于 Hetzel[1993]。虽然近期有人说采用功能点的比例有所上升，但是我们相信这么多年来这个列表变化不大。）

争议

“你不能管理自己无法度量的东西”，这种说法的问题是我们一直在管理自己

无法度量的东西。我们管理肿瘤研究，管理软件设计。我们没有任何数值的帮助，也能管理各种非常智能的甚至创造性的活动。优秀的知识管理者趋向于定性度量，而非定量度量。

然而，我们不应该因为该说法的错误，而拒绝它带来的正确观念。有了度量数据，管理就会容易，也会管理得更好。实际上，采用数据来辅助理解事物是管理者或者说是普通人的本性。我们喜欢在球类、田径等比赛中使用分值来表示结果，我们喜欢篮球和壁球，并发明了“三双（triple double）”等来衡量这些运动。我们甚至在滑冰、跳水等本来没有数据的项目中也采用数据进行衡量（裁判根据表现打分）。

实际情况是度量对于软件管理非常重要，谬误存在于具体进行度量的手段和过程中。

来源

“你不能管理自己无法度量的东西”这种说法在软件管理、软件风险方面以及（特别是）软件度量元方面的文章和书籍中很常见。当我试图查找这一说法的出处时，发生了一件很有趣的事情。几位度量专家说这种说法的出处是 DeMarco（1998）的《Controlling Software Projects》，因此，我也与 DeMarco 本人联系过，他说：“不错，这是我在《Controlling Software Projects》中的原话，但是”，他又说道，“这种说法实际上是‘你不能控制自己无法度量的东西’”。因此，这种错误的说法实际上是对于 DeMarco 真实含义的篡改！

参考文献

- DeMarco, Tom. 1998. *Controlling Software Projects: Management, Measurement, and Estimation*. Englewood Cliffs, NJ: Yourdon Press.
- Halstead, M.H. 1977. *Elements of Software Science*. New York: Elsevier Science.
- Hetzel, Bill. 1993. *Making Software Measurement Work*. Boston: QED.
- Rombach, H. Dieter. 1990. “Design Measurement: Some Lessons Learned.” *IEEE Software*, Mar.

谬误 2	可以管理软件产品的质量。

讨论

这里重申了本书前面提到的一个观点。在关于质量的一章，我提出“谁对质量负责？”，你可能记得我的回答是：无论有多少人相信管理者对于软件质量负责，但由于在管理当中技术问题太多，所以不能将它留给管理者。然后，我进一步说：所有的质量属性都有很深刻的技术内容，只有技术人员才能处理这些技术内容。

获得质量是技术任务，认为它是管理任务的人通常会误入歧途。多年来，管理者似乎认为普通的技术人员只在意自己工作的数量，尽量有意识地向他们灌输质量观点。“质量第一”的标语和“全面质量管理”的方法似乎是管理者获取软件产品质量的主要方法。技术人员不是接受这些方法，而是与之背道而驰。大家都知道对大多数软件项目而言产品质量最大的敌人是时间表压力，但是于事无补。管理在一方面是提高质量的动机和方法，在另一方面是影响质量的时间表压力。这些管理者当然不能同时满足这两方面。多数聪明的技术人员已认识到了这一点。

那么，这里的谬误是什么？谬误是把质量当作一项管理工作。当然，管理对于获得质量至关重要。他们可以建立高度重视质量的文化。他们可以消除障碍，促进技术人员提高质量。他们可以雇用高品质的员工，这是目前提高产品质量的最佳途径。他们可以在消除了困难和培养了企业文化以后，给这些高品质的人自由，让他们做自己一直想做的、引以为豪的东西。

争议

对此有很多的争议。我记得自己在 Seattle 大学教授研究生软件工程课程时，需要一本软件质量方面的教材。我拿到的每本书都非常重视管理。为了按照自己的想法来讲授软件质量问题，我最终决定为该课程写一本教材。在今天，绝大多数有关质量的书籍还是非常重视管理。

来源

➥ Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall. 该书的第 246 页有个寓言，形象地说明管理软件质量存在的问题。最后两行写道：

但是谁负责管理产品质量呢？

没有人回答。

➥ DeMarco, Tom, and Timothy Lister. 1999. *Peopleware*. 2d ed. New York: Dorest House. 该书作者发现质量的管理方法令人非常不愉快。他们把管理活动描述为“墙上的海报和图表”。他们把采用该方法对团队造成的影响比喻为“团队杀手（teamicide）”，把负责悬挂海报和图表的人称作假冒的项目成员，并说他们这样做会使大多数项目组成员不必要地紧张。由此可以看出，作者并不喜欢使用通常的管理方法来实现质量监控的目的。

5.1 人员

谬误 3	可以，也应该“忘我”地编程。

讨论

Weinberg（1971）的《The Psychology of Computer Programming》曾是软件工程方面的第一本畅销书。在该书中有许多优秀的思想，但同时也提到应该忘我地编程。作者认为程序员不应该在产品中投入自我的因素。当然，这种提法有非常好的理由。作者说：许多程序员在程序中纳入太多的自我因素，从而使产品偏离了设计目标。他们将错误报告视为人身攻击，将评审会议看作威胁，将对自己劳动成果的质疑看作是对立的。

与自我的程序员相对的是什么？以团队为中心的程序员。以团队为中心的程序员认为软件产品是团队努力的成果。错误报告、评审意见和问题是团队改进产品的驱动力，而不是使程序偏离正轨的威胁。

乍一看，很难说“忘我的程序员”这种观念是否正确。当然，自我的程序员往往太看重自己的成果，不欢迎修改，而通过修改来改善产品的质量又是不可避免的。再深入考虑，就可以清楚地认识这种观念了。倡导忘我的编程本身是正确的，但实际情况是人的自我因素是自然存在的，很难找到一个能将自我因素和自己工作分离开的人。例如，设想一位忘我的管理者。当然，这种想法很荒谬！在典型的管理者身上，自我因素恰好是高效工作的动力。我认为我们不应该将自我因素与普通的管理者分开，同样也不应该将自我因素与普通的程序员分开。如何

妥善地调整自我因素是我们始终需要关注的。

争议

即使 Jerry Weinberg——《The Psychology of Computer Programming》（1971）一书的作者——在近些年也认识到忘我编程的争议，修改了该观点。但他依然相信自己在 1971 年的说法；他认为反对该观点的人过于强调字面意思。将自我和程序员分开带来的优点无可非议。许多人认为团队方法（包括评审、检查和一度受到称赞的首席程序员团队，甚至极限编程中的结对编程）比其他方法更好，而且程序员确实应该乐于接受批评。本书中反复提到我们不能写出无缺陷的程序，这意味着程序员必须经常面对自己技术上的缺点和脆弱性。

但是，这里应该有一些折衷。我们不可能为了满足别人的需要来抑制自己的需求，我们也同样不太可能为了团队的利益而抑制自我。一个有效运行的系统必须承认人的个性，也必须在这些个性的范围内运作。自我就是这些特性之一。

来源

在随后的“参考文献”一节列出了 Weinberg 的这本经典著作。不要因为我不同意 Weinberg“忘我”的观点而妨碍你读这本书和 Weinberg 的许多优秀书籍（Dorset House 在 1998 年出版了这本书的 25 周年纪念版）。

参考文献

➥ Weinberg, Gerald. 1971. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

5.2 工具和技术

谬误 4	工具和技术是通用的。

讨论

在软件界确实有很多人相信“通用（one site fits all）”这一说法。这些人包括：兜售方法论的人、定义过程方法的人、推行工具和技术的人、期望建立基于组件

的软件的人、制定标准的人、研究下一个软件魔法的人、研究理论的所有学者。这些人都在寻找“软件中通用的灵丹妙药”，许多人甚至认为自己已经找到了。他们都想将这东西卖给你！

但是，这里有一个问题。因为软件要解决各种各样的问题，所以显然几乎不存在通用的解决方法。适合于商业应用程序的东西决不适用于关键的实时软件项目。适用于系统编程的东西通常与科学计算无关。适用于小项目的方法，包括当今所说的敏捷开发方法，不能很好地应用于需要几百个程序员的大型项目。适合于简单项目的东西，极不适用于关键项目。

在这个行业中，我们刚刚开始感受到所要解决的各种问题之间的差异有多大。我曾经试图描述这些问题范围（Glass 和 Oskarsson 1996）：

- 大小因素。小型的比特大型的容易得多。
- 应用领域的因素。例如，科学应用非常需要强大的数学基础函数，而其对于商业性程序和系统程序而言并非必需的。
- 关键性因素。如果在一个项目中涉及到生命或者巨额资金，那么该项目的处理方法将非常不同，特别是需要高度可靠性的时候。
- 创新性因素。如果你所面临的问题与以前解决的问题不同，那么你需要更多的探索，解决问题的方法会更少。

当然别人也有他们自己喜欢的分类方法。例如，Jones（1994）将应用领域划分为管理信息系统、系统软件、商业性市场产品、军用软件、合同/外包软件和最终用户软件（他随后很好地描述了每一个领域的特征，还讨论了各自最常见的风险因素）。

当然你还有自己喜欢的分类方法。许多实践者深知“我的项目与众不同”。然而，太多理论家蔑视这种说法，认为实践者根本不愿意尝试新的（通常是“通用的”）事物（但是请参考文献 Glass 2002a）。

争议

有关该谬误的讨论很激烈，那些一直相信存在通用方法的人却不断找到更多的通用方法。Plauger（1994）说：“相信存在通用工具的人都是一丘之貉。”Yourdon（1995）说“正在发生典型的转变”，在这个行业中，“人们正在改变‘所有软件本质都是一样的’这种观点。”Sanden（1989）提到了“折衷的设计方法”。Vessey 和 Glass（1998）指出：在解决问题的方法中，通常认为与问题相关的方法（类似

于固定型号的扳手）比较“强”，而通用方法（类似于型号可调的扳手）比较“弱”。项目间的差异非常显著（Glass 2002a）。即使通常受蔑视的“ad hoc（特别的）”观念也逐渐被重新认识（Glass 2002b）（词典中说“ad hoc”的含义是“适合当前的问题”）。如果你认为通用是坏事，那么该观念对立面的爆发对于这个行业非常有益。

来源

通用观念的对立面越来越强，例如，最近流行的敏捷开发方法认为“不同的方法适用于不同的项目”，还认为“sweet spot”项目很适合使用敏捷方法（Cockburn 2002）：2～8 个人在同一个房间内，一个现场专家，一个月的项目增量，有经验的开发者。在他的论文后面还提到传统的严格方法适用的项目（Highsmith 2002）：大型团队、关键项目和需要常规思维的项目。

- McBreen, Pete. 2002. *Software Craftsmanship*. Boston: Addison-Wesley.其中一节的标题就是“One Size Does Not Fit All”。

参考文献

- Cockburn, Alistair. 2002. *Agile Software Development*. Boston: Addison-Wesley.
- Glass, Robert L. 2002a. “Listen to Programmers Who Say ‘But Our Project is Different.’” The Practical Programmer. *Communications of the ACM*.
- Glass, Robert L. 2002b. “In Search of Meaning (A Tale of Two Words).” The Loyal Opposition. *IEEE Software*.
- Glass, Robert L., and Östen Oskarsson. 1996. *An ISO Approach to Building Quality Software*. Upper Saddle River, NJ: Prentice Hall.
- Highsmith, Jim. 2002. *Agile Software Development Ecosystems*. Boston: Addison-Wesley.
- Jones, Capers. 1994. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Yourdon Press.
- Plauger, P.J. 1994. *Programming on Purpose*. Englewood Cliffs, NJ: Prentice Hall.
- Sanden, Bo. 1989. “The Case for Eclectic Design of Real-Time Software.” *IEEE Transactions on Software Engineering* SE-15(3).（The journal, not

understanding the word *eclectic*, replaced it with *electric* when they published the article!）

➥ Vessey, Iris, and Robert L. Glass. 1998. "Strong vs. Weak Approaches to Systems Development." *Communications of the ACM*, Apr.

➥ Yourdon, Ed. 1995. "Pastists and Futurists: Taking Stock at Mid-Decade." *Guerrilla Programmer*, Jan.

谬误 5	软件需要更多的方法论。

讨论

奇怪的是我不记得谁真正说过这一谬误。在进入下一个谬误之前，你会自问“这家伙在干什么？”下面我将阐明这一问题。没有人在讨论发明更多的方法论，但是似乎每个人都在忙着做发明。

导师在做，研究生在做，严格方法论的对立者在做，甚至教授或者学者偶尔也在做。“人们想发明一种新方法或者模型，以便在软件业名垂青史”（Wiegers 1998）。似乎方法论机制运行得非常快，永不停息。

关于该谬误，还有一点非常有趣。方法工程研究者（Hardy、Thompson 和 Edwards 1995; Vlasbom、Rijsenbrij 和 Glastra 1995）的多项研究表明几乎没有实践者使用新鲜出炉的方法，相反，许多人在使用之前会修改方法来适应当前的形势。

起初，方法工程研究者对于一个事实感到为难，即“实践者怎么敢修改方法论专家的杰作？”他们似乎这么认为。但是随着时间的流逝，方法工程人员逐渐适应了现实——方法论不得不被调整，因为必须调整它们。实践者表现出必要的智慧，而不是异常的固执。

现在，该谬误关系到：我们是否需要方法论专家形成的所有方法。这个问题的实际情况是：许多人会说我们不需要。Karl Wiegers（在本书的前面，你已经听说过他了）极力反对发明新方法论，他甚至明确表示该问题的基调，他发言的题目是“Read My Lips，No New Models”（这里“模型”的含义是“技术、方法和方法论”）。

Wiegers 为什么要这么说？他说，是因为没有人使用已有的方法论。如果许多软件工具当前都处于阁件（Shelfware）状态，那么许多方法论都处于“哑炮（dull thud）”状态。他说到，方法论展现在舞台上，而应该采用它们的人却完全忽视了它们。

他们是否应该忽视方法论？这是一个公平的问题，这可能又是一个谬误或者事实。然而我个人认为：许多方法论都是（a）无知的产品（研究生或者老师怎么会知道软件实践中痛苦的实际情况？）；（b）一种智力警察（许多权威希望别人采用自己的方法论，理由是“这是正确的方法”，而不是证明自己的方法确实有助于构建软件产品）。DeMarco 和 Lister（1999）指出存在有 M（Methodologies）和 m（methodologies），M 来源于所谓的“方法论警察”。DeGrace 和 Stahl（1993）也做出同样的区分，但是使用不同的术语——他们称方法论警察为“罗马”，而更灵活的方法论为“希腊”。如果真是这样，我个人认为 m 是好事，而 M 非常糟糕，使用时一定要非常小心。

争议

我相信除了 Karl Wiegers 之外没有人会细分该谬误。也许应该有人这么做。关于方法论，我们有很多东西需要学习。

- 是否所有的方法论都有经验性事实的支持？（对于大多数方法论而言，答案是“没有”。）
- 方法论很少使用的原因是什么，是方法论本身无效，还是使用者无知？
- 实践者是否应该屈服于方法论？如果是这样，为什么？
- 是否应该在全公司使用统一的方法论？这样是否过于统一？
- 应该整体接受方法论，还是选用其组成要素中最精华的部分？
- 方法论的各组成要素除了仅仅“配合良好”之外，是否还有基本原则？
- 我们在什么时候采用什么方法论？或者其中的哪些组成要素？

我个人认为我们应该认真对待这些问题，并找到答案。当然，很可笑的是，我们已经有滋有味地使用这些方法论（结构化方法、信息工程、面向对象、极限编程、敏捷方法，等等）达数十年。

如果这样，是否还有争议？当然有。我们拥有的方法比我们会使用的方法多。对于方法论有许多问题没有解答。还有许多人不断发明新的方法，似乎这是他们的职业。我们应该叫停所有这些行为，应该设法解答那些问题，应该接受 Karl Wiegers 的呼吁，“不再需要新模型”。

来源

除了“参考文献”一节列出的来源之外，下面的材料对于方法论的观点进行

了总结和解释：

- Glass, Robert L. 1995. *Software Creativity*. Englewood Cliffs, NJ: Prentice Hall.

参考文献

- DeGrace, Peter, and Leslie Stahl. 1993. *The Olduvai Imperative: CASE and the State of Software Engineering Practice*. Englewood Cliffs, NJ: Prentice Hall.
- DeMarco, Tom, and Timothy Lister. 1999. *Peopleware*. 2d ed. New York: Dorset House.
- Hardy, Colin J., J. Barrie Thompson, and Helen M. Edwards. 1995. "The Use, Limitations, and Customization of Structured Systems Development Methods in the United Kingdom." *Information and Software Technology*, Sept.
- Vlasbom, Gerjan, Daan Rijsenbrij, and Matthijs Glastra. 1995. "Flexibilization of the Methodology of System Development." *Information and Software Technology*, Nov.
- Wiegers, Karl. 1998. "Read My Lips: No New Models!" *IEEE Software*, Sept.

5.3 估算

谬误 6	要估算成本和时间表，应首先估算代码行数。

讨论

我们在本书的前几个事实中多次提到估算，估算是软件开发中的重要活动。从这些事实中看出，我们尽力采用各种方法来做好估算。

多年来，我们已经提出了最受欢迎的估算方法，即首先估算产品的代码行数（lines of code, LOC）的观点。根据该观点，我们随后就可以将 LOC 转变为成本和计划时间表（大概根据 LOC 和构建这些 LOC 所需的成本和时间表的历史数据）。这种观念的基本思想是：我们可以通过类似的历史产品来估计 LOC，并根据已知的 LOC 推断当前的其他问题。

那么，为什么这种在计算机界最流行的方法是错误的？因为没有特别的原因表明估算 LOC 比估算成本和时间表更容易或者更可靠；因为没有通用的方法能将 LOC 转化为成本和时间表（我们在前面已经批评过通用的观念）；因为一个程序员的 LOC 可能与另一个程序员的 LOC 差异很大：一行 COBOL 代码怎么可能与一行 C++代码同样复杂？一行数学应用的代码怎么能和一行商业系统的代码相比？新手的一行代码怎么能与最优秀程序员的一行代码相比？（参考事实 2 中对该问题的回答：个体差异可达 28:1。）有许多注释程序的 LOC 怎么能与没有注释的程序的 LOC 相比？实际上的问题是，LOC 到底包含了什么？

争议

让我们开始争论吧！

在事实 8 中，我已经深入讨论过该谬误，我说到“这种想法有点可笑，因为估算 LOC 似乎比估算时间表和成本更难。可是许多‘明智’的计算机学者仍然倡导这种做法”。

你认为这很无情？我们还没有开始感受到对该谬误最强烈的抨击。Capers Jones 严厉抨击 LOC 方法。他在谈到软件界最大威胁时，将不精确的度量元列在首位，而且不失时机地说：是因为 LOC 度量元的原因才将度量列在首位。“在 1978 年已经证明，LOC 不能安全地应用于计算工作效率和产品质量”（Jones 1994）。他还列出了“LOC 度量元引出的 6 个严重问题”，以防你万一不是将“不精确的度量元”与 LOC 联系在一起，他又说道：“采用 LOC 度量元是最严重的问题”。

为避免不足以阻止你相信 LOC 的方法，Jones（1994）又列出“十大”风险，其中很多风险与使用 LOC 有关（Jones 将排名放在括号中）：

- 不充分的度量（2）
- 脱离现实的管理（4）
- 不精确的成本估算（5）

我们还可依次列出其他参与 LOC 激烈争议的人，但是与 Jones 尖刻的反对相比，他们都微不足道。

来源

Jones（1994）强烈地反对 LOC，我们在接下来的“参考文献”一节中列出他

的一本非常有意思、非常独到的好书。

参考文献

- Jones, Capers. 1994. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Yourdon Press.

第6章

生命周期

6.1 测试

谬误 7	随机测试输入是优化测试的好方法。

讨论

在事实 32 中（其中讨论了测试覆盖，提出几乎不可能达到 100%的覆盖），我首先提出了随机测试的观念。在那里，我将称它为四种基本的测试方法之一。这四种方法是需求驱动测试、结构驱动测试、统计驱动测试和风险驱动测试。该谬误就是针对我所说的统计驱动测试而言的。

“统计驱动测试”只是随机测试的一种悦耳称呼，其基本思想是：随机产生测试用例，不必察看需求、结构或者风险，仅仅通过随机检查来尽力覆盖所有的缺陷和漏洞。说得更复杂一些，随机测试用例的方法是从操作的层面进行测试。也就是说，虽然测试用例是随机选择的，但是它们应当符合软件典型应用的输入。

随机测试有一个重要的优点。假设所有的测试用例都来自用户将来运行时的输入，那么随机测试的结果可用来模拟真实应用。实际上，经过了统计驱动测试，软件人员可以说“该产品在 97.6%的情况下运行正常”。这是对用户非常好的说法。对用户这样说，当然比“该产品满足了 99.2%的需求”（这听起来很感人，但是我们已经知道 100%功能驱动测试仍然不够），或者“该产品经过了 93.4%的结构测试”（典型用户通常没有“结构”的概念），或者“在测试中，该产品成功的覆盖了可能面临风险的 91%”（是过程风险还是产品风险？是用户提供的风险还是开发者带来的风险？不达到 100%的风险测试怎么能接受？）都有意义。

但是，随机测试有很多缺点。一方面，它代表了风险。如果这些测试真正是

随机的，那么程序员或者测试员不知道软件中的哪些部分经过了彻底的测试，哪些部分还没有。特别是，异常处理对于许多软件系统的成功至关重要（许多最糟糕的软件灾难都是由于异常处理的失误引起的），即使随机测试是操作的验证，也无法证明能够命中异常处理代码。

另一方面，它忽视了程序员和测试员的智慧和直觉。别忘了，许多程序员趋向于犯“偏好”（一般）性错误（事实 48），而错误趋向于集中出现（事实 49）。许多程序员和测试员直觉中意识到这些东西，因此可以专注于测试这些问题。而且，许多程序员知道哪些部分的问题最难，因此会专注于这些部分。但是随机测试不“知道”这些问题，因此也不能专注于这些位置（如果能这样，就不是真正的随机了）。

再者，存在重复测试的问题。这是一种回归测试方法，如果要用一组特定的测试来验证修订的版本，就需要同一组测试反复运行。使用测试管理器也一样，测试管理器可以比较当前用例的测试结果与以前成功或者正确的测试结果是否一致。如果测试是真正随机的，就无法保证重复同一组测试。当然，也可以生成一组随机测试用例，然后“冻结”它们，以便重复测试。但是，这样我们又引入随机测试另一个层面的问题，即“动态”随机测试。

动态随机测试的观念是指根据一个成功的标准，在测试进行过程中动态地生成测试用例。例如，那些热衷于“遗传算法”的计算机科学家就将测试用例生成（“遗传测试”）看作该理论的一项应用。测试用例随机生成，以满足某些成功标准。在测试进展中，测试用例动态调整以提高与标准的符合程度（更详细的解释可以参见 Michael 和 McGraw[2001]）。当然，除非修正动态测试的基本思想，否则动态测试用例不可能重复。

争议

随机测试包括什么，还有人声称其用法可以优化，而争议最多的是许多知名的计算机科学家认为随机测试是自己错误消除理论的关键。例如，近来 Harlan Mills 将随机测试列为“净室（Cleanroom）”方法中的错误消除方法（Mills、Dyer 和 Linger 1987）。这种自身含有争议的方法要求：

- 形式化的验证（正确性证明）所有代码
- 程序员不做任何测试
- 一个独立的测试组做所有测试
- 所有的测试都基于操作层面，都是随机的

在实践中偶然会用到净室方法来消除错误，但是通常要调整部分思想来适应当前的形式。例如，经常用严格的审查（我们已经在事实 37 中看到审查方法非常有效）来代替形式化验证。我猜测，但是还没有数据来证明，排他性地采用随机测试的这种做法也需要调整（否则应用无效）。

近来出现了另一个争议。我们已经提到了遗传测试（Michael 和 McGraw[2001]）。这项研究还对随机测试进行评估，结果表明随着所测试的软件规模变大，其效力变差：

> 在我们的试验中，只要增加必须覆盖的条件，随机测试生成器的表现就变差，而且变差的速度比预期的更快。这表明：要满足各自的测试要求，在大程序中比在小程序中更难。而且这隐含着：当程序的复杂性增加时，急需采用非随机的测试生成技术。

随机测试用例生成变成软件复杂性的又一个受害者，似乎就要失败了。这使它成为本书中的谬误之一。虽然它可能还是一种测试方法，但是它很可能不再是一种最优的方法。

来源

有关该事实的来源见随后的“参考文献”一节。

参考文献

- Michael, Christopher C., and Gary McGraw. 2001. “Generating Software Test Data by Evolution.” *IEEE Transactions on Software Engineering*, Dec.
- Mills, Harlan D., Michael Dyer, and Richard Linger. 1987. “Cleanroom Software Development: An Empirical Evaluation.” *IEEE Transactions on Software Engineering*, Sept.

6.2 评审

谬误 8	“假如有了足够多的关注，所有的 bug 都显而易见。”

讨论

事实上，该谬误是引自开源社区的格言，含义是“如果有足够多的人看你的代码，就可以发现所有的错误”。这似乎非常正确，那么为什么我将它列为谬误？原因有多种，包括吹毛求疵的原因：

- 错误的深浅与查找错误的人数没有关系。

相关的原因：

- 有关审查的研究表明：增加审查者人数，发现错误数量的增加幅度会迅速减少。

关键的原因：

- 没有数据表明这句话的正确性。

下面我们逐一讨论这些原因。

吹毛求疵的原因。这可能仅仅是文字游戏，但是有一点很明确：有些 bug 比其他 bug 浅显，无论有多少人找 bug，其深度并不会改变。提到这个原因是因为：在很多人看来所有的 bug 似乎有相似的后果，而我们在本书的前面看到应当根据错误的严重性区别对待。“有许多错误消除者就可以减轻错误的影响”这种说法不过是误导。

相关的原因。这很重要。有关软件审查的研究表明：存在有效审查者的人数上限，超过该限度审查效果会迅速下降（例如事实 37）。这个人数非常有限，通常是 2～4 个。因此，如果该发现是正确的，那么我们必须质疑“假如有了足够多的关注”。当然，参加错误消除的人越多，发现的错误就越多。但是，类似于蒙古游牧部落的错误消除者，无论动机有多好，也不会生成无缺陷软件产品，还不如采用其他的错误消除方法。

关键的原因。没有证据能证明该谬误所蕴含的思想是正确的。我曾经听到开源的狂热拥护者引用各种研究文献来证明开源软件比其他软件更可靠（理由是有这么多关注）。我曾追踪每个来源，结果发现事实并非如此。例如，人们引用所谓的 Fuzz Papers 作为研究结果来说明开源软件更可靠（Miller）。实际上，Fuzz Papers 并没有切中开源的实质，甚至其作者（通过个人联系）也不认为开源更可靠（他认为应该如此，但是研究却没有证明确实如此）。实际上，Zhao 和 Elbaum（2000）表明开源程序员采用的错误消除方法并不比非开源程序员多，也许是因为他们期望有许多人能为他们做这个工作（这个期望不能保证，因为他们不知道，也无法

控制有多少人会真正为他们工作)。

争议

小心对待这些狂热拥护者。我不希望开源倡导者在这一谬误上取胜。

是否存在争议?当要有,或者很快就有。开源社区最重要的原则是该方法能生成更好的软件,而该谬误的实质是无法确定这种说法是否正确,这正中该原则的要害。

那么,为什么我要在开源面前螳臂挡车?因为非常有必要在此澄清事实。因为无论那种软件运动有多少狂热拥护者,我们都不能容忍他们随意欺骗这个行业。公正的说,与非开源的狂热鼓吹从规格说明书自动生成代码、不需要程序员就能编程的第四代语言(4GLs)和 CASE 工具一样,这些缺乏证据的言论也同样是鼓吹。

注意,我并没有说开源软件的可靠性比其他软件差。我的意思是(a)其中的一句格言是错误的;(b)少有或没有证据能证明该原则是否正确。

来源

除了“参考文献”一节中的文献外,请看下面的分析:

- Glass, Robert L. 2001. “The Fuzz Papers.” *Software Practitioner*, Nov. 该文分析 Fuzz Papers 的内容,以及开源可靠性。
- Glass, Robert L. 2002. “Open Source Reliability—It's a Crap Shoot.” *Software Practitioner*, Jan. 该文分析 Zhao 和 Elbaum 的研究结果。

参考文献

- Miller, Barton P. “Fuzz Papers.” There are several Fuzz Papers, published at various times and in various places, all written by Prof. Barton P. Miller of the University of Wisconsin Computer Science Department. They examine the reliability of utility programs on various operating systems, primarily UNIX and Windows (with only passing attention to Linux, for example). The most recent Fuzz Paper ad of this writing appeared in the Proceedings of the USENIX Windows Systems Symposium, Aug. 2000.

➥ Zhao, Luyin, and Sebastian Elbaum. 2002. "A Survey on Quality Related Activities in Open Source." *Software Engineering Notes*, May.

6.3 维护

谬误 9	估计将来的维护成本和做出产品更新的决策需要参考过去的成本数据。

1. 讨论

我们人类通常根据过去的数据来预测未来。毕竟，你不能仅仅依靠将来来预测将来。因此，我们假设将来的情况和过去的情况类似。这种方式有时有效，有时根本不奏效。

在软件维护过程中，反复出现的两个有趣的问题是：

- 维护本产品的预期成本是多少？
- 是否到了替换该产品的时候了？

这两个问题既有意思，又很重要。因此，难怪我们的一些老朋友会说采用前面的说法："我们应该根据过去来估计"。问题是，这种预测方法是否可以应用于软件维护？

要回答这个问题，先考虑一下维护方式。我们在事实 42 中看到，维护的主要内容是功能增强，因此，考虑软件产品修复频率的意义不大。为了保证预测方法有效，我们必须考虑产品功能增强的频度。

那么，功能增强的频度是否有规律？没有足够的数据能回答该问题，但是可以参考一些事实。做软件维护的人很早以前就说维护成本呈"浴缸"形（Sullivan 1989）。在产品刚刚投入使用的时候，需要许多维护，这是因为（a）用户从一个全新的视角来看待所要解决的问题，他们会发现许多相关的新问题需要解决；（b）起初是高强度应用，以后逐步过渡到稳定应用，所以起初暴露出的问题较多。随后，我们进入稳定期，即维护成本较低的中间阶段。因为用户对功能增强的兴趣有所降低，错误得到有效控制，所以用户可以从软件产品中获得到较高的价值。然后（当产品投入使用一段时间之后），不断的改变将产品推到原始设计架构的边缘，在这时候再做任何简单的修改或者功能增强都需要付出巨大的代价，我们又到了浴缸的另一个向上的斜边。原来低廉的修改现在变得非常昂贵。

在维护代价再次上升时，预测就变得非常困难。产品修改代价的提高和需要修改队列的增长妨碍了用户提出更多的修改要求，甚至用户因为产品不再满足需求而放弃该产品。不管怎么样，维护成本会再次下降，甚至迅速下降。浴缸形成一个光滑的、向右的尾巴。

现在，我们又回到根据过去来预测将来这个问题上。当然，我们刚讨论的现状是一个可预测的形状。但是，预测形状变化的时机很重要，实际上却几乎不可能。我们是否到了浴缸的底部，维护成本非常稳定？我们是否已经到了浴缸的边缘，维护成本在哪里持续增加？我们是否已经走出了边界，开始走平滑的下坡路，维护成本会下降？最坏的情况是，我们是否正处于一个转折点，曲线的形状在此迅速改变？或者当前的软件是否和刚才我们讨论的或多或少不一致，甚至其中没有浴缸或者平滑曲线？这些问题会使数学曲线拟合者发疯。

基于这些，我可以肯定：根据过去来预测将来的维护成本，这种方法收效甚微。理由是：（a）预测将来的成本很难；（b）预测产品替换的有效决策点几乎不可能。

关于将来的成本，最好是询问客户或者用户将来修改的预期成本，而不要根据过去的维护成本折算将来的成本。对于产品替换的问题，这种说法就更糟了。许多公司发现退役一个现有的产品几乎不可能。构建一个替代产品需要与当前版本的需求相吻合，而这些需求可能已经无处可找。因为文档没有及时更新，所以其中没有这些需求；因为原始客户、用户、开发者都已离开（一般软件产品都会使用相当长的时间），所以也不可能从他们身上找到需求；虽然有可能对现有的产品采用逆工程得到需求，但是这个过程容易出错，而且几乎所有人都尽力回避此过程。这验证了一句老说法“旧软件只会衰退，永远不会死亡”。

根据过去的软件维护成本来预测将来的成本，这种说法是谬误吗？你自己决定吧！也许其荒谬程度比你想像的还要大。

争议

将这件事纳入谬误的惟一原因是：研究者看待维护问题时（非常少的），总是倾向于采用优美、简洁的数学方法来回答我们上面提到的那类问题。所有的这些研究者几乎根本不懂得维护曲线的形状，因此他们假设产品的维护成本会持续上升，越来越高，越来越快，直到无法忍受。

在此，他们又犯了一个错误：他们认为需要研究的是修改成本，而不是功能

增强成本。增长越来越快的曲线可能不受欢迎，但这毕竟是一个可以预测的曲线，他们可以对此建立数学模型，无法忍受的那一个点代表应该做产品替换的时机。这种理论真神奇，是吧？它甚至采用公式来回答了丰富多彩的现实世界中的难题。

我最初在一次学术会议上听到一位学者发表有关该方法的论文。随后，我找到他并悄悄告诉他，为什么他的观点不切实际。我给了他我的名片，并告诉他我可以向他提供一些信息来源，以证明他的观点是错误的。他从来没有与我联系。相反，该学者的研究竟然成了许多后继研究的基础。在维护的预测方面，其他研究者把他的论文当作火种一样来参考。

我将这个问题说成一个谬误，希望那些还在探索曲线拟合数学法的研究者会看到本文，并认识到自己方法的错误所在。

来源

在此，我不想引用那篇被人们当作火种的论文，只要其作者及其追随者认识到我的说法，没有人会追问该论文。但是，下面是一些有关维护决策，包括产品替换方面的资源：

- Glass, Robert L. 1991. "The ReEngineering Decision: Lessons from the Best of Practice." *Software Practitioner*, May. 本文基于 Patricia L. Seymour 的部分研究成果，他是一位独立的维护问题顾问。
- Glass, Robert. 1991b. "DOING Re-Engineering." *Software Practitioner*, Sept.

参考文献

- Sullivan, Daniel. 1989. Presentation by Daniel J. Sullivan of Language Technology, Inc., at the Conference on Improving Productivity in EDP System Development.

第7章

教　育

谬误 10	教别人编程的方法是教别人写程序。

讨论

你是怎么学编程的？我打赌在许多大学课堂上，教师或者教授向你讲述一些编程语言的规则，然后就让你写一些代码。如果你学过多门编程语言，我打赌都是这样学习的。你读一门新语言或者学习一门新语言时，也是从写代码开始。

这种学习编程的方法很自然，似乎没有任何错误，但是在很多情况下确实有错误。

问题是：在学习其他的任何语言时，你首先学着如何*读*。你找到一本《Dick and June》或《War and Peace》之类的东西，然后就读。在读优秀作者的许多作品之前，你不会指望写出自己的《Dick and June》或《War and Peace》（相信我，写这个需要技巧！你必须采用适合读者年代和技能的词汇）。

那么，我们软件业如何陷入这种错误的轨道，即教人家先写后读？我也不知道，但是我们确实陷入很深。我没听说过任何院校和教课书采用先读后写的方法。实际上，计算机科学、软件工程、信息系统等计算领域的标准课程表都包含了写程序的课程，但没有读程序的课程。

我很早就开始思索，为什么我们会陷入这个境界。对此，我逐渐有一些想法。

1. 要教授读代码，我们必须选择所读的范例。也许这是顶级的代码，但是确实很难找到。也许是有瑕疵的代码，告诫别人不应该怎么做，但是这也很难找到。问题是：在我们软件界，对于优秀的代码和糟糕的代码通常没有一致的意见。

   此外，许多程序并非只包含优秀的代码或糟糕的代码，而是二者皆有。多

年以前，SEI 为了同样的目的，试图寻找一些示范代码，但最终放弃了。最后，他们找到了一些典型的优秀代码和糟糕代码的例子。（虽然，许多程序员认为自己是全世界编程水平最高的人，但是我们必须承认，谁都没有写出软件界的《War and Peace》!）

2. 要教授读代码，需要指导性的教课书，但是却没有。我所知的编程方面的每一本教课书都是有关写代码的，而不是读代码的。出现这种情况的原因之一是，谁也不知道如何写一本关于读代码的书。最近，我为一家知名的出版社审校一本这方面的书。乍看，这本书似乎要解决我们的问题。但是，这位有成就的作者的书稿中有一个严重的问题，因此使我建议拒绝该书。这本书是写给那些已经知道如何编程的人，而不是给那些真正需要先读后写的人。真正需要这样一本书的人是初学者（我说过，写《Dick and June》需要技巧）。
3. 我前面提到，在多年以前，我们制定了软件相关学科的标准课程表，这是一件好事。但是，在这些标准课程中忽略了代码阅读。这说明已经形成了先写后读的制度。你也知道要改变制度化的东西有多难！
4. 我们在软件中需要读代码的惟一时机是维护。维护很不受欢迎，理由之一是读代码是一项非常难的活动。发挥你自己的创造力写新代码比读别人的创造的老代码有意思得多。

争议

几乎所有认真思考关于学习编程问题的人都认识到我们的错误。但是几乎没有人愿意站出来努力改变现实（虽然前面所说的那本书被我被拒绝，但是其作者就是极少的例外之一）。我们将看到有人提倡先读后写，但是这似乎没有收效。结果是对该问题没有争议。实际上，激烈的争议非常有助于行业的健康发展。

来源

如果你从来没有深刻思考该问题——我认为这是对软件人士的标准——你会对下面这些持有不同观点的代表人物感到吃惊。

- Corbi, T.A. 1989. “Program Understanding: Challenge for the 1990s” *IBM Systems Journal* 28, no. 2. 注意该论文的发表日期——这是一个长期存在的问题。作者说：“对于其他‘语言’学科，经典的学习包括说、读、写……

许多计算机系统绝对信任他们的教育使学生适应于职业需要……在职业教育中，忽略了对程序理解技能的获取。”

- Deimel, Lionel. 1985.“The Uses of Program Reading.” *ACM SIGCSE Bulletin* 17, no. 2 (June). 该论文指出程序阅读非常重要，应该教给学生，他还建议了一些可能的教学方法。
- Glass, Robert L. 1998. Software 2020, Computing Trends. 这本论文集中有一篇的题目是“The ‘Maintenance First’ Software Era,”认为维护活动比软件开发更重要，因为在维护过程中必须阅读，所以建议采用先读后写的教学方法。
- Mills, Harlan. 1990. 在一次软件维护会议上的演讲，被 Glass（1998）引用。这位知名的计算机科学家说，我们应该“先教别人怎样阅读……因为在许多自然科学中（而不是在计算机科学中）我们先教读，后教写。”

这里还有一个引用，也许它非常有意思。我在事实 44 中讨论软件维护时提到该引用，不过它与这里的话题也密切相关。

- Lammers, Susan. 1986. *Programmers at Work*. Redmond, WA: Microsoft Press. 其中引用了 Bill Gates（年轻时）的一句话，“成为程序员最好的方法是写程序，研究别人所写的优秀的程序……我到计算机科学中心的垃圾筐找到了他们操作系统的列表。”

结　论

在这里，我们讨论了软件工程方面最基本的 55 个事实和一些谬误。你可能会同意其中的一些事实和谬误，而不同意另一些。但是我希望这能激发你创造性的灵感，帮助你解放思想，更好地构建和维护软件。

下面我列出这些事实和谬误中的一些重要的思想。

- 软件过程和软件产品的复杂性决定了我们在该领域的许多认识和行为。复杂性不可避免，我们不应该与之对立，而应该学会适应它。其中有 15 个事实涉及到复杂性，还有其他一些事实是复杂性引起的。
- 在软件领域，糟糕的估算和由此带来的时间表压力一直在迫害我们。例如，许多失控的项目并不是因为糟糕的软件构件实践，而是因为不现实的既定目标。有 10 个事实涉及到该主题。
- 在软件管理者和技术人员之间有隔阂。这解释了许多事实，例如，技术人员在项目刚开始时就已经知道了因为急急匆匆而导致项目失控的问题，但是没有报告。有 5 个事实集中讨论了隔阂的问题。
- 鼓吹和通用的观念影响了我们形成专注于有力的、明智的项目方案的能力。虽然明智的人反复告诉我们不要再去寻找魔法，但是我们一直在寻找。关于该错觉有 4 个事实。

这些事实和谬误的*来源*中，也说明了许多重要的问题。为了统计该数据，我找到了这些事实和谬误的所有来源，确定每个来源的关键作者是研究院所的研究者、实践者、兼而有之的人（在两领域都工作的人），还是“宗教领袖”（因为杰出的观点而知名的人物）。在统计数据的过程中，我逐渐对自己的发现感到吃惊。

这些事实和谬误的主要来源是那些同时具有实践者和研究者双重身份的人，即类似于 Barry Boehm、Fred Brook、Al Davis 和我自己这样的人。大约 60 个来源（占 35%）属于这一类。第二大来源是实践者，即软件业中的全职工作者，这一类来源大约有 50 个（占 29%）。科研院所研究者的数量比我预期得少得多，大

约 40 个（占 23%）。让我非常吃惊的是，领袖类的最少，大约 20 个（占 12%）。

我想，这一结果仅仅表明了我在选择这些事实及其来源时的个人偏好。虽然我喜欢优秀的评估性技术研究，但是我更欣赏基于实践的优秀评估性研究。而且，我特别注意查找领袖的声音。

我还想特别表扬两个以研究为主的组织。我认为，Software Engineering Laboratory（SEL）作为一个学术/实践/政府的联合体，多年来成为软件业中许多重要的、基于实践的、高质量研究的源泉。同时还应该称赞 NASA-Goddard（政府）、Computer Sciences Corp.（企业）和（特别是）Maryland 的计算机科学系（科研院所）所形成的联合体，他们开展了许多重要工作。软件界需要更多这种的组织。

还想表扬 Carnegie Mellon University 的 Software Engineering Institute（SEI）在软件工程技术转化方面所做的先驱性工作。一旦类似于 SEI 的研究者有了重要的研究发现，有人应该帮助做后继研究。SEI 可能不会如我所期望的那样专注于某些研究技术，但是他们一旦选定了某项技术，就会做出杰出的成效。

我还要列出一些对于本书中的事实做出重要贡献的人，他们的研究基于实际，研究结果非常可信。非常感谢 Vic Basili、Barry Boehm、Fred Brooks、Al Davis、Tom DeMarco、Michael Jackson、Caper Jones、Steve McConnell、P. J. Plauger、Jerry Weinberg、Karl Wiegers 和 Ed Yourdon。我所提到的软件工程界的任何一项重要发现，都很有可能与他们中的一个或几个人相关。

现在，我该总结了。在软件工程领域，我喜欢的说法之一是：

> 一组丑陋事实中的真实性是漂亮理论的终结者。

在本书中，我不准备列出“一组丑陋的事实”，我也不相信所有的理论都很漂亮。但是，我确信，在软件工程领域，任何有价值的理论，无论是否丑陋，都应该与我所提到的这些事实一致。我建议与这些事实不一致的理论家应该认真反思自己提出和倡导的理论。我还建议实践者在考虑与这些事实不一致的工具、技术、方法和方法论的时候，意识到其中潜在的严重缺陷。

多年来，我们软件领域犯了许多错误。这并不是指失控的项目和失败的产品，它们比人们设想的要少得多；也不是指“NIH（not-invented-here）”和“it-won't-work”的人，我认为这些人也很少。我所说的错误出自那些大愚若智的业内人士，他们提出、倡导、建立了许多明显的谬误。

我希望对这些事实和谬误的整理有助于消除那些错误。

关 于 作 者

Robert L. Glass，迄今已在计算领域工作有近 50 年，1954 年～1957 年期间在 North American Aviation 致力于航空航天事业，从那时起他就已步入计算机领域。这使他成为软件业中一位真正的先驱。

在 North American 之后，他还到其他的航空航天公司工作过（Aerojet-General Corp., 1957～1965，Boeing Co., 1965～1970 和 1972～1982）。他的主要工作是开发工具软件，供应用领域的专业人士使用。当时宇航事业正在飞速发展，在这一时期从事航空航天工作也振奋人心，这同时也是计算机领域迅速崛起的时期。两个行业的发展都非常迅速！

在航空航天领域工作期间，他得到最主要的结论是：自己喜欢做一个软件技术人员，而不是管理者。他积极向技术专家方面发展，这对于他的职业产生了两点影响：（a）他的技术知识一直是不断更新的、有用的；（b）他的管理知识和工作收入却相应地减少。

当 Glass 只能在技术方面发展时，他随后转到学术界。1982 年～1987 年期间，他在 Seattle 大学教授软件工程研究生课程，1987 年～1988 年在 Software Engineering Institute 从事研究工作（他早期[1970-1972]曾受华盛顿大学资助，从事以工具为中心的项目研究）。

在这些学术生涯中，他得到的主要教训是：喜欢将自己的头脑置于软件工程的学术方面，但是自己的心灵仍然处于实践中。显然，你可以将一个人带出某个行业，但是你不能从一个人心中抹去这个行业。凭借这些新获得的智慧，他开始寻找一个桥梁，来连接自认为在学术界和产业界之间存在的“隔阂沟通”。

他找到了许多方法。他的许多书籍（20 多本）和专业论文（超过 75 篇）都重点集中在评估学术界的发现，以及将这些发现应用于产业界，从而实现其价值（这肯定是一项非常重要的任务，在很大程度上说明了他的观念和文章中的批判性）。他有关软件工程的演讲和论文也将重点集中在对于实践有意义的理论和最佳

的实践发现。他的通讯稿“The Software Practitioner”是同样的风格。多年来，他曾经是《Journal of Systems and Software》这一（比较学究的）期刊的编辑（现在是名誉编辑），该期刊也是同样的风格。他定期为《Communications of the ACM》、《IEEE Software》、《ACM SIGMIS's DATABASE》等出版物写专栏文章，也是同样的风格。虽然他的许多作品都是严格的、批判性的，但是其中的许多作品也不乏计算机幽默。

他在计算机领域有了这么多成就，其中最自豪的是什么？在 1995 年，瑞典的 Linkoping University 授予他荣誉博士学位。他在 1999 年被提名为 ACM 的专业会员。