

免费试读章节

(非印刷免费在线版)

如果你喜欢本书, 请去

[China-pub](#)、[卓越网](#)、[当当网](#)

购买印刷版以支持作者和[InfoQ中文站](#)

向电子工业出版社以及译者韩磊致谢

InfoQ中文站
www.infoq.com/cn

本节选由[InfoQ中文站](#)免费发放, 如果你从其它渠道获取此摘选,
请注册[InfoQ中文站](#)以支持作者和出版商

本摘选主页为

<http://infoq.com/cn/articles/37signals-small-beauty>

《梦断代码》官方网站为

<http://www.dreamingincode.cn/>

第 9 章 方法

还记得质量三角吗——时间、金钱和特性（或质量）？直到 2004 年秋天，也就是 Chandler 公开宣告后两年，OSAF 在其中任何一方面都做得不太好。快吗？不。便宜？花了好几百万美金，而且到现在还没什么能拿得出手的东西。好吗？还得走着瞧。

Chandler 0.4 版于 2004 年 10 月 26 日发布时，一周内大约只有一千人下载，而 Chandler 1.0 在发布 24 小时内就有一万五千人下载。新版本在启动时看来有点更像是真正的软件应用，如果你愿意深入挖掘的话，还能发现一些内建的新功能。第一次出现了部分可工作的细节视图，可以给条目打戳，使之变为另一种类型。

但对于基础的日常任务——发送或接受电子邮件、保存日历信息、组织便条——Chandler 还是不能让试用者们满意。即便对于它的程序员们来说也不是可以吃的狗食。启动超慢，崩溃频繁。

下载数量的下降不足为奇。世界，至少到目前为止，一直在进步。许多以前为 Chandler 发布高唱赞歌的外部人员抛弃了它，有些正式参与者认为它迷失了道路。“对我来说，那简直像是一次火车事故，”当我在

0.4 版发布后不久和安迪·赫兹菲尔德聊天时，他悲哀地摇着头这么说。

然而，OSAF 内部第一次充满了宁静的感觉。发布之前的几个星期一点也不狂躁。经理们和 QA 人员系统检查了 Bugzilla 中的开放缺陷列表，并且做了筛选：这个本周内修正，那个是微不足道的小问题；这个放到下一版，那个的确是只“拦路虎”，必须得修正。应用组——负责 Chandler 的外观、用户界面，这部分缺陷最多——有系统地减少着列表上的缺陷数量。设计组在 0.4 版完成之前就开始计划做 0.5 版计划，而那些没陷入缺陷修正工作的开发者则开始做与 0.5 版开发相关的事情。

OSAF 也许还没有足可交付给公众的产品。但却另有收获：开始有了工作流程，还有一套可能让它朝目标行进的可行的方法论。

嗯，等一下，你可能会问——难道他们这两年的工作都**无流程可依**吗？不完全是这样。OSAF 有很多流程，也许流程太多了，而且这些流程有时看来像 Chandler 代码一般不固定、常被修改。但在开发 0.4 版的 8 个月里发生了一些变化：项目终止了拍脑瓜，建造了某种结构——某种组织机制，可以依靠它向前冲，而不必每几个月就重建一次，或者为每个新版本重新启动其引擎。

从 Chandler 项目的最早期开始，卡普尔就坚持要做诚实、现实的计划和进度安排。但项目在满足进度方面却拥有不佳的记录：平均 6 个月能发布一个版本，但计划却总假设应在 3、4 个月内完成一个版本。部分原因或许是在软件开发和其他领域中计划总是超出了能预见的范围。如丽萨·杜索特所言，“要留心，如果当前计划涉及一年以后，有可能这个计划会失败。”

Chandler 和绝大多数其他软件项目一样遭遇了这令人沮丧的现实。软件开发者很少成组地共同开发一系列项目；他们不太像运动队或是军队单位，也不太像合唱团，而更像是专才们为制作一部电影而临时组合

①艾德加·狄杰斯特拉，1930年出生于荷兰阿姆斯特丹，2002年逝世于荷兰纽南。以发现了图论中的最短路径算法（狄杰斯特拉算法）而闻名于世，1972年因为ALGOL第二代编程语言而获得图灵奖。

②移去一部分胶片，使得动作产生跳格移动的剪辑手法。



艾德加·狄杰斯特拉

我开始研究 20 世纪 60 年代，那是个软件创造者们初次系统地审视自己和自己工作领域的年代。1968 年，艾德加·狄杰斯特拉（Edsger Dijkstra）^①在《美国计算机学会通讯（Communications of the ACM）》上发表了一篇短文，标题有点儿逗乐：《Go To 语句被认为有害（Go To Statement Considered Harmful）》。在许多当年占优势地位的编程语言中，例如 Fortran 和 Basic，计算机逐个运行一系列编号指令语句，程序员可以使用 GOTO 语句在执行顺序中各个点之间跳转。如同电影剪接师使用跳接^②手法，GOTO 语句也是弃程序一行而不顾，另择一行执行之。它无条件地将控制从程序的一点移转到另一点，对其他因素——不同变量的值或是程序的状态和数据——不管不顾。

“Go to 语句过于原始，”狄杰斯特拉写道，“它是一份请柬，把混乱邀进了程序。”

在 20 世纪 60 年代和 70 年代，许多程序员接受了这份请柬，制造出混乱——枝蔓缠结的软件，后来被称为“意大利面式代码”。“结构化编程”的倡导者们从狄杰斯特拉和其他设想一种更模块化编程风格的理论家那里得到灵感，提出一系列编程实践，试图消除过程式意大利面带来的磨难。结构化编程有几个对立派别；在狄杰斯特拉看来，核心观念是编写由一组子单元组成的程序，每个子单元只有一个进入点和一个退出点。狄杰斯特拉写道，这样的程序不再是一堆面条，而是“一串珍珠项链”，其简洁和明晰的特点不但降低了出错可能性，还让程序员更能理解自己的手工作品。

结构化编程以防御式下蹲的姿态提出建议，尽力让可能犯错的程序员自避其短。“我脑子不好使，头很小，我学会了与之共处，正确看待自己的不足之处，并完全依赖它们。”狄杰斯特拉写道。他在 1972 年获得图灵奖——计算机界的诺贝尔奖，获奖演说的标题是《小程序员（The Humble Programmer）》。他的著作中也充斥着对“可怜程序员”的同情，

和对人类这种动物“力有不逮”的耶利米哀歌（Lamentations）^①。

结构化编程的具体规条很快就体现到新一代编程语言的设计和语法中。其主要规则——“程序的每一层都该自成一体”——在此后几十年里不断推动创新的产生，程序员们（怀着以可复用乐高积木式组件为最终圣杯的梦想）设计出各种新方法孤立、“模块化”和“封装”软件中可移动的部分。然而，和这些将出现的创新一样，结构化编程并没有帮助那个时代的软件变得完美；反之，它为将来更具野心的软件埋下了坚实的基石，而那些软件也将以结构化编程想都意想不到的方式遭到失败。

改进组织代码的方式，最终是与改进组织人员及其工作方式之间的步态竞赛^②。当产业在整个 20 世纪 60 年代和 70 年代表现平平，新软件方法论的拥护者们的注意力也转向了企业中“人”的一方面。具体而言，在与让大规模项目搁浅的趋势做斗争的时候，他们开始聚焦于改进制定计划的过程。而且，随着时间推移，他们分裂为两个意见相左的阵营。就像开源项目的贡献者们反目成仇一般，他们也分道扬镳了。

其中一派看着计划说：“好计划太有价值，但是太难制定，你得更努力地做计划，花更长时间，更细节化，然后再做一次计划！直至无所计划为止——然后再做一点计划。”另一派说，本质上，“做计划是不可能的，而且结果也没什么用，你应该摒弃所谓的计划。船到桥头自然直！写代码、听反馈、问客户、不断修改。这是唯一值得依赖的计划。”坦白地说，我有点儿夸张。但这种分裂是真有其事，而且多数软件开发者和经理，因其天生倾向或痛苦经验，都会倒向其中一派。

项目管理大师瓦茨·汉弗里（Watts Humphrey）提出了“*Must. Plan. More!*（必须制定更多计划）”的主张：“除非开发者为个人工作制定计划并遵循之，否则工作将不可预料。而且，假使开发者的个人工作成本及进度不可预料，则团队工作的成本及进度也不可预料。当然，假使项目团队的工作不可预料，则整个项目也不可预料。简言之，只要个体开发

^①《圣经·旧约》中的一章，乃先知耶利米回顾他在《耶利米书》中预言圣城耶路撒冷被毁之事。

^②旧时美国南部黑人比赛步态优美，优胜者奖以蛋糕。



瓦茨·汉弗里

者不为个人工作制定计划并遵循之，项目就会无法控制、不可管理。”

另一方面，当代管理学之父彼得·德鲁克的研究结果却是：“关于知识工作者任务的多数讨论都建议他们制定自己的工作计划。听起来很好。可惜错在它很难实现。计划总是停留于纸上，想法总是好的。付诸实施者寥寥。”

德鲁克于 1966 年发表上述言论。与此同时，年轻的瓦茨·汉弗里即将执掌 IBM 的软件管理大权。

10101010101010101011010110110

当时，IBM 正努力要推出 OS/360——IBM 新一代大型计算机 System/360 的操作系统——其惨败正是弗里德里克·布鲁克斯写出《人月神话》的肇因。在接受负责 OS/360 团队的工作之前，布鲁克斯已经一只脚踏出了 IBM 的大门；他正打算到北卡罗莱纳大学创办计算机科学系。依汉弗里所言，尽管布鲁克斯尽其所能带领 360 编程团队走过了概念和设计阶段，他还是在想法实现之前离开了，而项目也在他离开之后遭遇了“大麻烦”。

所谓**大麻烦**：缓慢。延误。再次赶不上重定的进度。更多延误。最终 IBM 交付了没配上完整操作系统的 System/360 硬件。光这就多花了整一年时间。

汉弗里于 1966 年 1 月接管 IBM 软件组织后，震惊于情况之严重，写下一份简报。“我首先要知道工作进行到什么位置……每到一个实验室，我就要求管理人员提供计划和日程安排。他们只拿得出非正式的笔记或备忘录。当我询问经理们如何看待管理软件项目的最佳方式，他们说应该在开发开始前做计划。当我问及为什么不这么做时，他们的回答是没时间。这根本毫无道理！做这事的正确方法无疑就是最快和最廉价

的方法。显然这些经理没在做管理，只是随机应变罢了。压力太大，工作太多，他们只能做那些发布代码所必需的事情。除了火烧眉毛的事，别的都往后推。我认为责任在我，与他人无涉。只要我允许他们在没有计划的情况下宣告和发布产品，他们就会继续这么干下去。”

汉弗里还发现，在软件团队还没有发布产品的计划或资源时，IBM 的市场部门就急于宣告。他跑去找老板。“我告诉他，既然所有的发布日期毫无价值，我有意全部取消。然后，我要叫所有软件经理为每个项目做计划。往后，在我拿到带有说明文档和开发经理签名的计划之前，我们将不会宣告、交付或投入资源到任何编程项目……那些实验室花了大概 60 天做第一份计划。这个从未做过交付进度安排的团队，在之后两年半的时间里再也没有延误过。”

汉弗里在 IBM 执行强制进度纪律的成功基于两条原则。计划是强制性的。计划必须符合现实情况——“从底至上”，依据那些负责按计划执行的程序员的经验和知识而来，而不是“从顶至下”，靠管理者拍脑袋或对市场的期望而来。

20 世纪 80 年代，汉弗里从 IBM 退休，考虑下一步应该做些什么。如他后来告诉《商业周刊》所言：“我女儿拽我去参加一个关于誓愿的训练班。在训练班上，一直在讨论‘发宏愿’，比如消灭世界饥饿。所以我也发个宏愿，就是离开 IBM 后，我要改变这世界的软件开发方式。安坐沙滩上，实在太无聊。”

汉弗里加入了卡耐基·梅隆大学软件工程学院（Software Engineering Institute, SEI）。SEI 是五角大楼于 1984 年成立的机构，旨在改进用纳税人的钱购买的不断增加的大量软件系统的质量。在 SEI，汉弗里和同事们创建了软件成熟度模型（Capability Maturity Model, CMM），作为一种衡量软件开发组织品质的准绳。

CMM 给出了可供编程团队攀登的五级台阶。要读完 CMM 原则的全部文档可能得耗尽你的余生,但汉弗里在 1997 年发表的非正式描述给出了简单的概述:“位于第一级的组织基本上什么都没做。第二级组织做一些计划、跟踪、配置管理工作,也讨论质量保证之类的话题。第三级组织开始定义过程——如何工作、如何完成任务、可训练的事项等。在第四级,他们采用衡量准绳。他们有一套真正跟踪和管理自己所做工作的框架,一种可统计跟踪的系统。第五级组织拥有持续改进的过程。”

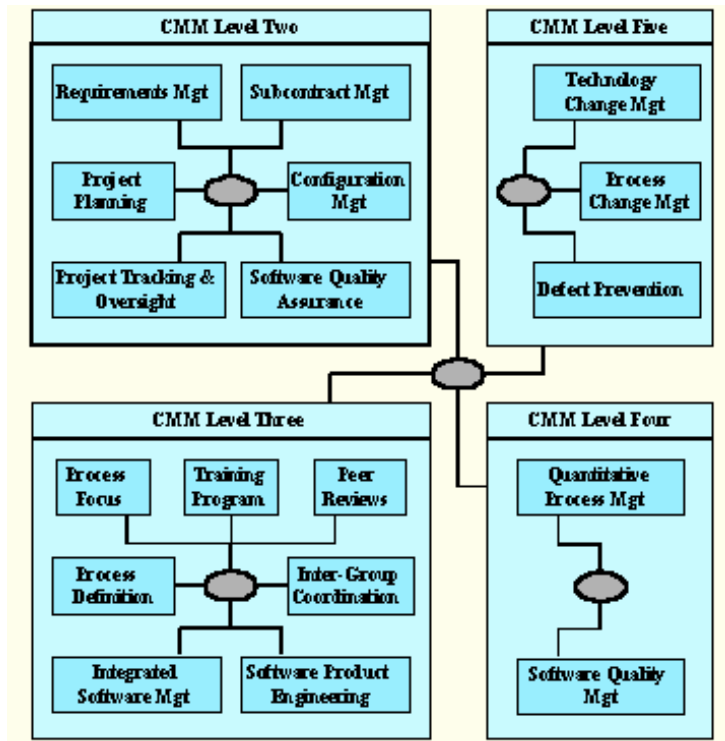


图 9-1 CMM 体系结构

美国国防部用 CMM 测量承包商的组织力量，它在这方面表现得最好。而在商业软件世界和急速增长的个人计算机软件业中则问津者无几。直至今日，如果在多数小软件公司中提到 CMM，都会招来茫然的眼光。在关注过 CMM 的程序员和经理当中，最普遍的批评集中于其非人性和官僚化，带来沉重、缺乏创新的负担。在 1994 年一篇题为《CMM 不成熟之处（The Immaturity of CMM）》的文章中，软件质量专家詹姆斯·巴赫（James Bach）试图揭穿 CMM：“CMM 重过程而轻人力……创新本身在 CMM 中没有得到体现，只在第五级上有所提倡……因为 CMM 怀疑个人贡献、无视需要非线性思维的情况、满足于将它们埋葬于约束的上层建筑之中，达到 CMM 2 级标准可能会扑灭公司赖以发展的星星之火。”

CMM 主要目的是帮助规模庞大的组织改进软件进度和质量。实际上，继 CMM 之后，汉弗里和 SEI 还提出一对相关计划——小组软件过程（Team Software Process, TSP）和个体软件过程（Personal Software Process, PSP）——某种程度上就是为了回应这些抱怨。TSP 和 PSP 批评“专制的管理风格”，鼓励个人开发者和小团队主动做计划和质量控制、分享信息、按需“动态平衡”工作量，以此夺回自己命运的控制权。汉弗里有一张幻灯片上列出了以下很有说服力的要点：

- 我们都为组织工作。
- 组织需要计划。
- 除非你的工作足够独立，否则必须按进度工作。
- 如果你不自己做进度安排，别人就会给你做安排。
- 这样别人就会控制你的工作。

CMM、TSP 和 PSP 的灵感都来自于制造质量专家 W·艾德沃茨·戴明（W. Edwards Deming）。戴明提出，质量不该是一种后悔药，而应体

现到生产过程的每一阶段上。对于软件而言，它意味着 CMM 反对“编码然后修正”（程序员先写出充满缺陷的产品、测试员找到缺陷、然后程序员再回头修正）的传统。“在所有的现代技术中，”汉弗里说，“软件是唯一一种在产品测试之前不考虑质量的领域。然而这不是工程师的错，因为他们就是这样被训练和管理的。”

就像戴明的理念在战后日本的汽车工业得到应用，远远早于在他的祖国——美国受到关注，CMM 也主要是在国外大受欢迎——特别是在印度快速增长的软件业中，软件公司发现，获得某个 CMM 等级会让海外客户认为他们值得信赖、足堪承担外包工作。但在美国的商业软件或桌面计算世界，CMM 及其相关方法论还没产生重要影响。

2000 年出版的一本名为《软件阴谋：为什么软件公司扔出错误重重的产品，它们如何能伤害你，而你又该怎么做（The Software Conspiracy: Why Software Companies Put Out Faulty Products, How They Can Hurt You, And What You Can Do About It）》的著作宣称这是一桩丑闻：既然这些方法被证明可以产生出更好的结果，为什么不是每个软件团队都采用、每个客户都要求呢？作者马克·米纳西（Mark Minasi）将指责范围扩大到不受约束的程序员、鼠目寸光的公司以及乐意忍受次品和缺陷的公众。

米纳西和其他 CMM 的拥护者认为，软件业知道如何按时、在成本预算之内生产出高质量的软件——但它又肥又笨又懒，不愿意接受改进的方法。他们也许说对了。另一方面，他们的言论假设软件制造者面临的最大挑战是降低产品中的缺陷数量。据汉弗里所言，在一位典型的有经验程序员的工作成果中，每 7 到 10 行代码就能找到一个缺陷；SEI 的研究显示，CMM 和相关的方法明显地减少了缺陷的数量。

CMM、TSP 和 PSP 借用了工厂管理的术语，将缺陷（bug）称作“缺点（defect）”，而缺点是被工人“注入（injected）”的——就像是说产品一开始处于某种柏拉图式的理想完美境界，后来被疯眼工人将充满缺陷

的针剂注入血管，遭到了破坏。这些方法强调一见到缺陷就杀灭，因为根据所谓的贝姆定律（Boehm's Law），在开发过程中越晚修正缺陷，代价就会越高。

“先修正缺陷”是个好原则，多数程序员都会同意该走这条路——原则上同意。但不管程序员抱着何种意图开始做，代码总是生而有缺陷，因为编写代码的过程，借用软件开发词汇表中最热门的说法，是“迭代”的。先写个能开始的东西；不断重复修正——“迭代”——改进。如果你非要坚持完美，就连第一行代码都永远写不完。实践上，在等着最终产品出来的人们失去耐心之前，“先修正缺陷”原则工作良好。然后，该原则就在忙于交付即便不完美也勉强可工作的代码的混乱之中被弃置道旁。

另外，缺陷总是存在。如果你瞅瞅任何一个活跃软件项目维护的缺陷列表，就会找到各种大问题和小麻烦。根据对用户使用的影 响、寻找和修复缺陷的努力程度不同，缺陷也大有不同。而每份缺陷列表都是风马牛不相及的具体的表现。

软件缺陷编年史上记录了数量不多但足以令人警醒的惊人灾难。1962年6月，水手一号探测飞船在发射后5分钟偏离轨道，为避免坠入居民区，飞行控制人员只好将其引爆。问题在哪儿？导航控制程序中少了一个连字符。1996年6月，欧洲航天局投入5亿美元建造的阿丽亚娜5号无人运载火箭发射后40分钟爆炸，原因是在控制其导航系统的软件中有一个缺陷。（代码要将64位变量转换为16位变量，但数字太大，出现缓冲区溢出，系统停滞了。）从1985年到1987年，由于软件缺陷，一台名为Therac-25的放射治疗仪向6名病患放射了过量的X射线。在1991年的海湾战争中，在飞毛腿导弹袭来之时，爱国者导弹的一个电池未能成功点火；敌方导弹击中美军营地，导致28人死亡。调查发现，软件中存在一个累积计算错误，经过数百小时连续使用之后，爱国者导弹的计

数就会变得很大，从而无法点火。

上述事例及类似的故事数量太少，不足以引起媒体的愤怒和政治上的骚动，但它们却警示着我们对程序员工作质量的依赖。相对于每个致命缺陷的例子，都有上千个导致财产损失或经济损失的缺陷存在。人们指责软件问题导致了停电、航班延误和机场关闭、银行账户混淆、自动回拨，以及几乎所有你能想到的混乱和不便。

但引起灾难的瑕疵却并非“每十行代码一个”的缺陷；如果真是这样，那么任何软件系统都不能工作了。尽管计算缺陷数量明显是一种有价值的措施——总比不计算好——但也会倾向于把严重的系统故障和不值一提的小问题扔到同一个篮子里。而且还会鼓励程序员修葺既有产品，而不去打造新事物。这也不错——除非你就是想做点新东西出来。

10101010101010101011010110110

在 20 世纪 80 年代和 90 年代，美国军方接纳 CMM 和软件工程学院字母形花片汤的其他部分，当作改进其所需软件的工具。但在千禧年之际，当五角大楼的软件规划师们考虑大规模未来新武器计划、同时面临 9/11 军事承诺的压力时，他们开始担心，即便有这些方法论的帮助，程序员还是跟不上新系统所需软件的增长速度。

在 2004 年于盐湖城举办的五角大楼软件开发年会上，美国空军器材司令部工程和技术管理主任扬·奥格（Jon Ogg）指出，军方正面临他所谓的“软件发散困境”。在过去 50 年里，一个典型军用系统中的代码量增长了成百倍：例如，20 世纪 60 年代喷射战斗机可能会有 5 万行代码，而新的联合攻击战斗机则会有五百万行代码。但在同一时期，程序员的平均生产力只翻了一番。奥格说，这意味着我们发现“当今开发一种功能所需的人月数量”增长了 50 倍。

南加州大学软件工程教授、成本估算领域先驱和贝姆定律之父巴瑞·贝姆（Barry Boehm）在该次会议上以更简洁的语言指出：“国防部需要软件的数量以指数级增加，而在有限时间内根本不可能完成。”

面临这样的困境，军用软件供应商们受到五角大楼这个最重要客户的鼓励，开始试验不太传统的措施。他们并不孤独。在 CMM 及其相关方法论在这些防务承包商中扎根的同一年，外界开始演化出另一种传统，它直接来自于那些知道过程已经破毁、渴求找到更好路径的程序员的经验。

运动的一条主根埋藏于“模式社区（Patterns community）”中，组成该社区的软件开发者从建筑哲学家克里斯托弗·亚历山大处得到启发。20 世纪 80 年代末 90 年代初，面向对象编程的崛起及其令人生畏的复杂抽象把程序员们推向理解力的极限。软件模式运动——其领导者包括 wiki 发明人沃德·坎宁汉和一位名为肯特·贝克（Kent Beck）的程序员——为他们设想出一条新生命线，一种较少条条框框的软件方法论。他们没规定最佳实践的定规，而是以扼要叙述记录下他们的经验。“对于此类问题，”他们会说，“我们发现这种编程模式很有用。”模式运动将软件开发比作飞行器；如资深计算机专栏作家布莱恩·赫斯（Brian Hayes）所写，“程序员就是空乘人员，和木匠或石工一样，他们的知识自经验和师徒相承而来。”

坎宁汉和贝克等模式拥护者将上手实践总结带到迫切需要的领域。例如，对于降低某套面向对象代码的复杂性，坎宁汉和贝克建议程序员通过在桌上排布索引卡——每个软件对象一张卡片——来设计新程序。“卡片的好处是便宜、便于携带、容易获取、而且很常见，”他们写道，“我们惊奇于将卡片移来移去带来的价值。当学习者捡起一个对象，看起来更乐意和它打交道，而且也准备好从对象的角度来应付剩下的设计。正是这种物理性互动的价值让我们始终抵制卡片的计算机化。”

模式运动让个体程序员有了一条解决问题的新路子，并且提炼经验、供同事所用。但它并不太有助于找到在更大项目中协调工作的方法。

几十年来，典型的项目组织方式遵循“瀑布模型（waterfall model）”。瀑布方式——最初出现于1970年——将项目切分为依序进行的多个独立阶段，比如需求定义、设计、实现、集成、测试和发布。每个阶段需在下一阶段开始前完成。这套做法在纸上看似合乎逻辑，但实践起来却总是导致延误、混乱和灾难。每个阶段都耗时无算，但没一个工作正常的。程序员要么坐等需求，要么干脆在拿到需求之前开始做设计。“大设计优先”引发大延误，“大爆炸式集成”——单独编写代码的各个主要部分，在项目将近结束时拼到一起——导致系统崩溃。到产品终于完成那天，已经过了很久，该程序要解决的问题已经无关紧要了，新问题又在呼吁解决方案。

瀑布模型逐渐得到了它该得的坏名声。到了20世纪80年代中期，巴瑞·贝姆定义了一种叫做“螺旋模型（spiral model）”的替代方案，将开发切割为六个月到两年的“迭代周期”——更快生产出可工作代码的小瀑布，从部分完成的产品使用中得到反馈、指导下一步迭代。螺旋模型在大规模政府承包软件开发中成为标准，在那个领域，典型项目的“采办周期”可能会长达十年之久。

然而，在商业软件飞速发展的世界中，这还是太慢。20世纪90年代，软件业方法学的信徒们竖起了快速应用开发（Rapid Application Development）的大旗，RAD承诺通过快速原型设计和更紧迫的迭代周期，依靠新工具让计算机处理一些繁重的编程工作，加速完成软件的交付。RAD帮助软件公司更敏捷地工作——但在它开始掌舵后不久，就和Web的到来以及软件产业对速度的渴求再次凸显一起，迈出了互联网时间的疯狂步伐。

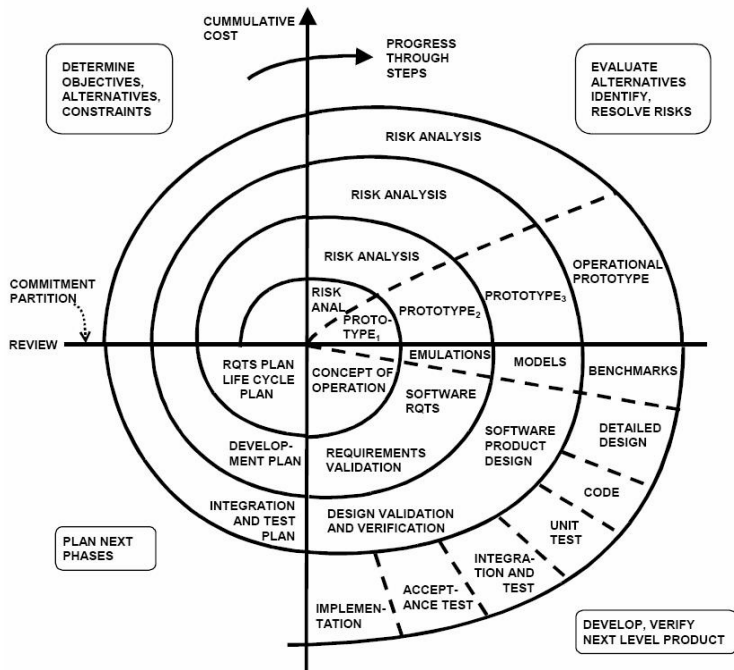


图 9-2 螺旋模型

20 世纪 90 年代快速出现的新方法论群体，钟情于适应项目的持续变化而不设法预测及控制其产出，被称作“轻量级方法论”。这让它们区别于 CMM 世界缓慢而笨拙的“重量级”方法论，却无助于将它们兜售给商业领袖们；它们听起来无所用处。2001 年，该领域中的 17 位领军人物^①，包括坎宁汉和贝克，聚集于犹他州的滑雪胜地，想要找出他们互相关联但有不同的方法之间的共同立场。参加了那次聚会的软件测试方法论专家布莱恩·马瑞克（Brian Marick）写道：“那次会议的部分目的……是找到术语‘轻量级过程’的替代说法。人们会觉得‘我是轻量级的’这种说法没有传递准确的信息。”

^① 他们是 Kent Beck、Mike Beedle、Arie van Bennekum、Alistair Cockburn、Ward Cunningham、Martin Fowler、James Grenning、Jim Highsmith、Andrew Hunt、Ron Jeffries、Jon Kern、Brian Marick、Robert C. Martin、Steve Mellor、Ken Schwaber、Jeff Sutherland、Dave Thomas。

会议为运动取了个更有男子气概的名字——敏捷软件开发（Agile Software Development）——还发布了一份宣言，全文如下：

我们正通过实践和帮助他人来揭示开发软件的更好方法。

经由这项工作，我们估量：

个体和交互 胜于 过程和工具

可工作的软件 胜于 面面俱到的文档

客户协作 胜于 合同谈判

响应需求 胜于 遵循计划

即，尽管右栏条目有其价值，但我们更看重左栏条目。

在一个其理论文档倾向于过度使用缩略语和充斥八股文体的领域，《敏捷宣言（Agile Manifesto）》因其简明扼要而引人注目。但“敏捷开发”更像是共有价值观的保护伞，而不是具体过程的路线图。自从宣言发表后，各种相关但互异的敏捷方法论层出不穷。其中一种叫做争球式开发（Scrum），将项目分解为 30 天一轮的“竞跑”，强调每天开例会、维持项目在正轨上运转。不过，到现在为止，敏捷方法论最流行的变种还是极限编程（Extreme Programming，或称 XP——不要跟 Windows XP 操作系统搞混）。

肯特·贝克在 20 世纪 90 年代提出了极限编程背后的支持理念，当时他正为克莱斯勒公司（Chrysler）做咨询工作，试图挽救一套偏离正轨的薪资册自动化项目。“极限编程”这哗众取宠的名字听似跟批量购买高价运动用品有关，而且实际上贝克也是将“极限运动”对运动员专注和勇气的要求与极限编程对编码者的要求相提并论。不过极限编程的名称主要还是指它采用了一系列广为接受的方法，并将这些方法的使用推至极限。

“我们撷取最佳程序员已经采用的那些实践方法，并把所有的拨号盘都拨到 10，” XP 的创立者之一罗恩·杰弗里斯（Ron Jeffries）如是说。测试重要？那就让程序员在写代码之前写测试。开发组和客户交谈有好处？那就让客户在旁边随时回答开发者的问题。不做周期性代码走查，而是让开发者一直结对工作，这样每行代码从被写出来那一刻起都有不止一双眼睛看过。至为重要的是接受客户的需求，软件的目标就会保持变化，并且让项目“拥抱变化”。

XP 往软件开发世界中引入了一套全新并且有点陌生的词汇表。它要求将项目切分为“故事（stories）”：每个故事代表客户向开发者讲述的一种特性需求，解释了程序应该做什么。然后程序员为“故事”编码；如果结果正是客户需要的，万事大吉，准备做下一个故事。在 XP 激进的渐进机制下，不再有“大设计优先”；编码几乎就立刻开始。这对于许多急于开始编码但又受冗长设计过程所阻挠的程序员来说颇有吸引力。XP 认可的忘掉详细规约和代码文档的方式也很流行。

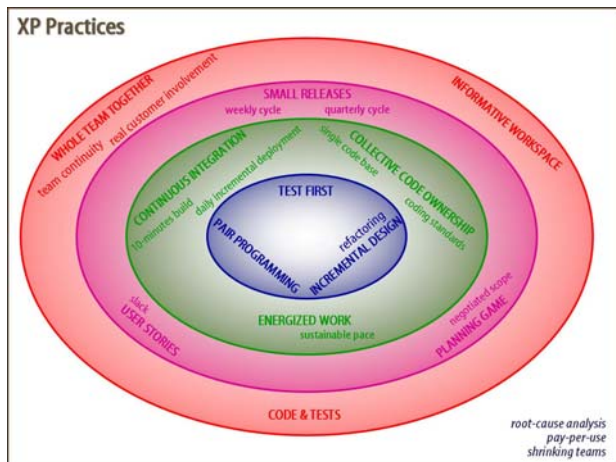


图 9-3 极限编程实践

任何接触过这个领域的开发者都会承认，软件史过去 30 年中的方法论队伍丰富了这个领域，而且给了程序员以新工具和思考其工作的方法。但要想找到一个真正实施了某种特定方法论的开发人员或团队可不容易。有两位宾夕法尼亚州的软件工程教授在 2004 年做了一项研究，询问在多个行业中两百个软件团队的开发实践。研究报告写道，“我们震惊和失望地发现，最常用的实践就是什么实践也不用——有整整三分之一的受访单位采用了这种实践（如果可以叫做实践的话）。”

震惊和失望！许多软件开发者还在反抗，哭喊着“别把我关进去”。但其他许多程序员，尽管抵制那些标签和口号，还是静静地拥抱和推进了这些年来出现的各种方法论所建议的技术。“如果没有某种系统实践，这东西很难做，”他们会这样说。

祖尔·索伯斯基 (Joel Spolsky) 就是其中一员。他曾正式担任微软项目经理，负责 Excel 开发。Excel 是在 20 世纪 90 年代早期从米奇·卡普尔的 Lotus 1-2-3 手中夺得头魁并一直雄踞榜首的 Microsoft Office 电子表格产品。当索伯斯基离开微软、创办自己的小软件公司时，还开设了一个名为“祖尔谈软件 (Joel on Software)”的网站，发表文笔尖刻但充满实用建议的趣文。以下是其中之一：

如果你还有哪怕一点点常识，就该问：“数据在哪儿？如果要我转用激烈编程 (Intense Programming)，我想要看到那些多花在狗窝和鸟笼上的钱这回要花在提升程序员自尊心上的证明。给我数据看看！”

当然，我们没数据……

你无法公正地比较两支软件团队的生产力，除非他们在完全一致的情形下做完全一致的事情，两边的人手也完全一致，他们以某种方式克隆出来，确保先前是白纸一张……

①网络泡沫时代的创业公司很多都购置这种按摩椅，极尽奢侈之能事。

我们没有数据。你可以举出关于方法论X可行或不可行的各种奇闻轶事，但你却无法证明其可行不是因为团队里面有一位真正、真正优秀的程序员，而且你也无法证明其失败不是因为公司就快破产、人人自危、手足无措，即便是坐上安乐按摩椅（Aeron chairs）^①也无所助益。



图 9-4 祖尔·索伯斯基的演讲和文章一样火爆热辣

索伯斯基非常怀疑他所谓的“巨无霸方法论（Big-M methodologies）”。在另一篇文章中，他写道：“小心那些方法论。它们是通往阴森但尚可通行的效能水平的康庄大道，但同时它们让那些痛恨被约束的才智人群极为烦恼。我很明白，好厨子不会乐意在麦当劳做汉堡，正是因为麦当劳那些规矩所致。”

索伯斯基写下上述文字几年之后，我采访他时，他仍旧心存蔑视。“方法论的真正目的，”他说，“是卖书，而不是真正解决问题。”索伯斯基还说：“方法论的关键问题在于，那类发明方法论的聪明人实施方法论时，就会有用。但如果是让那些只知道听令行事的笨蛋来实施，就不管用了。无论如何，大多数开发者不看关于软件开发的书，也不上关于软件开发的网站，甚至不看 Slashdot。所以，不管我们写出多少文字，他

们都领会不到。”

“即便到了今天，我环顾四周，虽然什么都上了网，还可以买到一百万本关于如何管理的图书，有一百万种关于如何做好软件项目的阅读材料，你仍然看得到同样的那些硅谷旧人物，他们本该知道得更多，因为这已经是他们开的第三个公司了，但他们还在犯同样的错误，做同样的错事，提同样愚蠢的假设。”

索伯斯基也有自己的一种方法论：他称之为祖尔测试（Joel Test），基于他自己的经验和“集体智慧”，给出一套快餐式原则，判断开发组织是否符合这条原则——“不会叫人头疼的 CMM”。祖尔测试询问以下十二个问题：

你们使用源代码控制吗？

你们每步都做构建吗？

你们做每日构建吗？

你们有缺陷数据库吗？

你们会在写新代码之前修复缺陷吗？

你们有与当前工作吻合的进度安排吗？

你们有规约吗？

程序员工作环境安静吗？

你们采用了市面上最好工具吗？

你们有测试人员吗？

你们会要求应聘者在面试时写代码吗？

你们做走廊可用性测试^①吗？

^①到走廊上随便拉一个人试用程序。

“得 12 分为最佳，”索伯斯基写道，“11 分还可以接受，得 10 分或更低就说明你们问题大了。其实多数软件组织只能得 2、3 分，他们迫切需要帮助，因为微软等公司一直都得 12 分。”

诚然，微软是历史上最成功的软件公司，年复一年地赚取几十亿美元利润。它的行为值得留意。另一方面，与其商业竞争对手相比、或与那些编写公司内部软件的程序员相比，微软也不乏进度延误和忽视缺陷的情况。假使微软拔得祖尔测试的头筹，为什么还是有问题呢？“我想微软正处于在规程上要求万事齐备的阵痛之中，不再以一致水平制造人们愿意购买的产品，”索伯斯基说，“太过沉迷于过程……到了任何微软产品都要花 6 个月才能发布新版本的程度。他们真的开不了快船。在 SP2 (Windows XP Service Pack 2) 上花了差不多一年的时间。出于安全考虑，他们做了件好事，不过它的作用基本上也就是清理、维护和打补丁。”

“这就是军队叫做内务的东西。在军队里，内务就是保持装备在最佳工作状态的一切事情：擦鞋、刷牙、时刻准备着、子弹保持清洁、确保枪膛里没沙子。所有这些都叫做内务，步兵每天要花 2 个钟头做这些事。但它却不是你真正要干的事情。微软现在到了大概百分之八、九十的时间都在做内务的阶段。”

索伯斯基用步兵来做类比，这来自于他的第一手经验——他在以色列长大，曾经在那儿当过兵。不过还可以做更多比较。软件开发者常被描绘为激烈的公司战役中“死亡征途”上的步兵。他们接受任务，拼命在急剧变化的周遭环境和战争迷雾一般的可见度条件下完成任务。他们身上也常带有历史上老兵们那种宿命在天和坚定不移的组合特征。

我参加了那次五角大楼软件大会，聆听个体软件过程培训师戴维·库克 (David Cook) 半是鼓舞士气半是存在主义训诫地解释方法论规程的目的。

“你的软件有多少可能会改动呢？”库克激昂地问道，“噢，百分之

和总裁杰森·弗瑞德（Jason Fried）解释说，在他们自己意识到之前，已经做出了一套基于网页的应用。又做了4个月，他们把软件转换为称作 Basecamp 的服务。Basecamp 发布于2004年2月，很快在类似 Flickr 和 Google 的 Gmail 等新 Web 富应用天堂中名列前茅。

Basecamp 只是这家公司花一年多时间投入少量程序员做出来的一系列值得注意的小而精的产品之一。Basecamp 之后是 Ta-da List，用于保存和共享待办事项（及类似事项）列表。几个月后推出了 Backpack，它允许用户保存和共享便签及文件。每种产品都可靠并易于使用，而且都是精心设计的。每种产品通常也都只包括少量新特性。例如，Basecamp 就有一些精巧的电子邮件功能：和其他服务和程序一样，也可以设置邮件到达提醒——还可以从另外的计算机或手机等移动设备向 Backpack 网页发送邮件，邮件文本就会在页面上显示出来。

我刚开始使用 Backpack 时，是用来保存本书的零散调研笔记。2004年秋天在一个技术大会上偶遇弗瑞德，我问他 37 Signals 怎么能在如此之短的时间内做出这么有用的软件。他大力鼓吹自己的方法——他公司开了个名为“制作 Basecamp”的训练班，将所用原则做成了一套 PowerPoint 幻灯片——而且逼着我在酒店大堂里听了45分钟关于其方法论的概要介绍。

首先，37 Signals 只有一位开发者，所以就避开了布鲁克斯法则的泥沼——就像米奇·卡普尔最初做 Lotus 1-2-3 那样，当时也只有乔纳森·萨赫斯（Jonathan Sachs）一位程序员。开发者之间的协调不成问题。37Signals 唯一的开发者戴维·海因梅尔·汉森（David Heinemeyer Hansson）住在丹麦，就连这似乎也不成问题。弗瑞德说，在大多数公司里，地理上的分隔会被看做是严重问题，不过时差却让他们真的只有区区几个小时可以讨论，所以他们会高效利用这点时间，跟着开发者们就能平心静气地写代码，不受干扰。

照 37 Signals 的做法，约束是朋友。“约束是打造伟大产品的关键，”弗瑞德说，“约束产生创意。如果有人能给你全世界的财富，让你做任何想做的东西，那这东西多半永远发布不了。给我一个月就好！”

37 Signals 生产优秀软件的另一关键要素是紧抓 Web 应用不放。所有东西都通过网页浏览器运行，所以程序可以在任何能运行浏览器的计算机和操作系统上工作。版本更新可以很容易地在运行服务的服务器上做到，用户无须下载和安装更新。汉森还热衷于 Ruby，一种面向对象动态编程语言。Ruby 近似于 Python，不过较少为人知，汉森发现它简化了自己的工作。最后，37 Signals 的方式还避开了编写规约的环节；相反，一开始就做用户将看到的详细网页。这些页面设计成了规约。弗瑞德说，他的团队很少会长时间争辩页面上的每个词、按钮和方块。

37 Signals 只做小程序，不做野心勃勃的新平台或应用程序框架。但在打造 Basecamp 的过程中，汉森还写了一些有用的创新代码，改善和简化了所有 Web 应用在保存和获取数据时都要执行的细节基础操作。Basecamp 发布后，他和 37 Signals 决定把这部分工作拿出来，作为一套开源平台发布，名字是 Ruby on Rails。这套将被命名为 Rails 的框架在某种程度上通过约束程序员的可选手段使得编写 Web 应用更为简单。“灵活性被过分高估——约束才是解放，”汉森说。Rails 也具备实现 AJAX 风格增强界面的能力，这种新界面风格让基于 Web 的程序足以与桌面应用抗衡。

37 Signals 从 Basecamp 中抽出 Rails 的同时，还从 Basecamp 的经验中归纳出一套设计哲学，体现为一系列小警句：“精简代码。”“拒绝在先。”“找对人。”“与其做半成品，不如做功能减半的优质品。”这些短句是为了通过幻灯片快速演示，不过合起来却是一整套软件开发方法——姑且称之为实用最小主义。它也许不能满足鼓舞了如此多程序员的改变世界之瘾。你也可以批评它是锋芒尽失的表现。它看似不适用于那些别无选择只能做大的软件。用程序员们的话说，就是“配不上”。

出了解决困扰当今软件业的方案，但在下一波问题蜂拥而至时却无计可施，这也是为什么每过十年就会出现一大把方法论的原因。

然而在一种情况下方法论的确管用。那就是商业思考者尼古拉斯·凯尔(Nicholas Carr)于2003年5月发表在《哈佛商业评论(Harvard Business Review)》上那篇声名狼藉的以《IT无关紧要(IT Doesn't Matter)》为题的文章中展示的场景。凯尔激怒了硅谷的幻想家和技术管理者，他提出，他们的产品——信息技术，或称IT的全部内容——已经变得无关紧要。当柏林墙倒塌、苏联解体时，黑格尔学派哲学家弗朗西斯·福山(Francis Fukuyama)^①曾喊出了名噪一时的“历史终结(The End of History)”，现在凯尔也号称软件历史上已经**完结**——给它个痛快吧！我们知道软件是什么，知道如何在商业世界中部署软件。除了用重量级方法论使之完善外，别无他法。



图 9-5 尼古拉斯·凯尔：IT 无关紧要

^①政治经济学家。日裔美国人，出生于芝加哥，分别在康乃尔大学和哈佛大学拿到了古典文学学士和政治学博士学位。著作品重点关注于民主化和国际政治经济体系等问题，主要有：《历史的终结及最后之人(The end of history and the last man 1992)》，《大分裂：人类本性与社会秩序的重建(The great disruption: human nature and the reconstitution of social order 1999)》等。

凯尔比较了过去几代的“破坏性技术”，例如铁路和电力，他认为一开始计算机和软件让具有远见卓识的早期采用者们拥有了获得相对优势的机会。他说，不过现在它们全都是商品了。它们很重要——忽视它们就会落后于竞争；但它们不能“让某个公司明显地与竞争对手区分开来”。在公司战略和商业竞争的大游戏中，它们无足轻重。它们变成了烦人的基础设施——只是些水管罢了。

凯尔文章颇具煽动性的标题引起了意料中的反驳，但如果你把 IT 看做一种功能和能力的固定集合，那么他的论点也不无道理。IT “无关紧要”，只要你一早就知道你期望用软件实现的业务——会计、 workflow 自动化、仓储管理等等——并不会改变，而且竞争对手也在用软件完成同样的事情的话。

但在企业重金投入的所有资本货物中，软件是唯一一种多变的。那些占据了大公司 CTO 和 CIO 所有时间的用于“客户关系管理”和“企业资源规划”的庞大软件包，可能会相当笨重而昂贵。然而它们也还是——就像丹·布瑞克林和米奇·卡普尔在四分之一世纪之前引入到这个世界的小小电子表格——（用弗里德里克·布鲁克斯的话来说）“思想的产物”。所以，当人们决定将软件的某个部分用于新目的时，用到的部分就要进行改动。当你引入一套新的雇员调查系统，某人头上的灯泡就亮了——嘿，何不改造一下这套系统，用来安排公司晚会节目？只要几个星期就够！既然开始做了，开发组也许可以加一个讨巧的功能，管理我的音乐收藏——嗯，这样我们就能投票决定晚会上放什么音乐了！

我们是否已经发掘了软件提供的全部有用之处——当然，还有所有无用之处呢？相信这套说法，就不仅是傲慢自大，而且还是对计算机历史的贸然无视。例如，20 世纪 80 年代中期，商业分析师们断言整个个人计算机产业已经成熟并到达巅峰，可麦金塔和 Windows 风格的图形界

面系统即将出现。到了 90 年代早期，微软开始称雄于 PC 软件业，大家都以为整个业界趋于稳定——但互联网又将引领新方向。2000 年和 2001 年，互联网泡沫破灭，许多观察家都准备宣称整个网络工业可以盖棺定论了；然而就在街角，Google 成长起来，并且推动了在线商业的整个新一波浪潮。

一次又一次，那些打赌在技术工业的太阳底下不再有新鲜事物的人输了个精光。相信我们已经知道软件的所有可能用途，等于是假设我们拥有的程序能满足所有需求，人们也不会再去寻找更好的东西。

《软件阴谋》的作者马克·米纳西就是这类软件缺陷的愤怒批评者，与尼古拉斯·凯尔之类软件商业怀疑论分析者共有一副“历史终结”的眼罩。如果你相信我们已经知道需要软件做的所有事情，那么很自然也该相信，只要工作足够努力、计划足够详细，我们就能让软件变得完美——而且那也正是该全力投入的地方。不要再想新特性和新点子；所有人都集中力量于缩减缺陷列表，直至有史以来第一次可以说大多数软件已经备其全形。微软确乎尝试过；在一连串令人难堪的病毒爆发事件后，微软从 2003 年开始停止所有新产品开发，将大量编程人力投入到修正 Windows 安全漏洞的工作上。结果推出 XP Service Pack 2，改善了 Windows 安全机制，但这并没能挡住人们发明新病毒来折磨它的步伐。

这种孤注一掷、不顾其他的做法偶尔也有意义，当普遍使用的有缺陷软件亟待修正时尤为如此。不过在多数时候却无关紧要：它没明白为什么我们对自己依赖的软件还不满意。用户可能会为缺陷所烦扰，软件开发者可能会因没能做到最好而失望，经理们可能会因他们计划之不可靠而感到挫败。但是到最后，什么也不及软件不以我们期望的方式工作来得严重，而且只要它继续这样，就不值得去打磨。

101010101010101011010110110

一次在 OSAF 吃午餐时，约翰·安德森承认，有时他会觉得，如果能挥动魔杖、按计算机内存大小计将时钟拨回十年前，那么软件可能会更易于编写、质量也会更好。

“用今天的工具和过程，加上昨天的内存限制，我们真的能做得更好，”他说。

“你只是说说而已，还得设定软件目标的上限，”我反驳说，“应该做功能更少的软件。”

“嗯，对，”他答道，“功能应该更少——但缺陷也要更少。”

101010101010101011010110110

只要我们持续要求软件完成新任务、解决新问题，修正今天的缺陷就不能让我们免于明天的崩溃。在打算重复成功时，虽然软件方法论非常有帮助，你还是只能照葫芦画瓢。如果是在探寻未知的领域，最佳实践可能会帮助你稍微加快一点进度，但却无法指点前进方向。

软件开发常与建筑工业相提并论，但这种类比在某种意义上站不住脚。尽管我们学会如何建造解决特定问题的建筑——供一家人居住，或者为治疗病患提供合适的空间——我们还是得不停建造更多建筑。不管我多喜欢你的房子，我也不能据为己有——你已经住在里面了，如果我也想要，就得再建一幢。软件可不是这样；只要我们知道如何写一套程序来计算支票簿余额或显示一张网页，那这个程序的额外副本完全没有生产成本，你付出的价钱，如果你给钱的话，是为了奖励设计出软件的人，而不是为了制造这份副本。

必为了驾驶而勉力成为汽修爱好者；可我们却一直在等待，期待按照同样的原则，软件领域中也能出现类似的情形。

此论有理。也许有一天我们能到达软件真的变成像水管或汽车一样的地步——能循常例生产、不费心思也能用、而且永不改动。

然而，若以过往为鉴，看来软件还是会做全新的不同工作。这个十年的汽车会在下个十年变成飞车——然后，潜水艇！而且我们会欢迎、甚至要求这样的变化。