第1章 焦油坑

1.1 编程系统产品（Programming Systems Product）开发的工作量是供个人使用的、独立开发的构件程序的九倍。我估计软件构件产品化引起了3倍工作量，将软件构件整合成完整系统所需要的设计、集成和测试又强加了3倍的工作量，这些高成本的构件在根本上是相互独立的。

1.2 编程行业“满足我们内心深处的创造渴望和愉悦所有人的共有情感”，提供了五种乐趣：

创建事物的快乐；

开发对其他人有用的东西的乐趣；

将可以活动、相互啮合的零部件组装成类似迷宫的东西，这个过程所体现出令人神魂颠倒的魅力；

面对不重复的任务，不间断学习的乐趣；

工作在如此易于驾驭的介质上的乐趣——纯粹的思维活动，其存在、移动和运转方式完全不同于实际物体。

1.3 同样，这个行业具有一些内在固有的苦恼：

将做事方式调整到追求完美，是学习编程的最困难部分；

由其他人来设定目标，并且必须依靠自己无法控制的事物（特别是程序）；

权威不等同于责任；

实际情况看起来要比这一点好一些：真正的权威来自于每次任务的完成任何创造性活动都伴随着枯燥艰苦的劳动，编程也不例外；

人们通常期望项目在接近结束时，（bug、工作时间）能收敛得快一些，然而软件项目的情况却是越接近完成，收敛得越慢；

产品在即将完成时总面临着陈旧过时的威胁。

第2章 人月神话

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

2.2 良好的烹饪需要时间，某些任务无法在不损害结果的情况下加快速度。

2.3 所有的编程人员都是乐观主义者：“一切都将运作良好”。

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

2.5 但是，我们的构思是有缺陷的，因此总会有bug。

2.6 我们围绕成本核算的估计技术，混淆了工作量和项目进展。人月是危险和带有欺骗性的神话，因为它暗示人员数量和时间是可以相互替换的。

2.7 在若干人员中分解任务会引发额外的沟通工作量——培训和相互沟通。

2.8 关于进度安排，我的经验是为1/3计划、1/6编码、1/4构件测试以及1/4系统测试。

2.9 作为一个学科，我们缺乏数据估计。

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

2.11Brook法则：向进度落后的项目中增加人手，只会使进度更加落后。

2.12 向软件项目中增派人手从三个方面增加了项目必要的总体工作量：任务重新分配本身和所造成的工作中断；培训新人员；额外的相互沟通。

第3章 外科手术队伍

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

3.2 Sackman、Erikson和Grand的数据显示经验和实际表现之间没有相互联系。我怀疑这种现象是否普遍成立。

3.3 小型、精干队伍是最好的——尽可能的少。

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

3.5 对于真正意义上的大型系统，小型精干的队伍太慢了。

3.6 实际上，绝大多数大型编程系统的经验显示出，一拥而上的开发方法是高成本、速度缓慢、不充分的，开发出的产品无法进行概念上的集成。

3.7 一位首席程序员、类似于外科手术队伍的团队架构提供了一种方法——既能获得由少数头脑产生的产品完整性，又能得到多位协助人员的总体生产率，还彻底地减少了沟通的工作量。

第4章 贵族专制、民主政治和系统设计

4.1 “概念完整性是系统设计中最重要的考虑因素”。

4.2 “功能与理解上的复杂程度的比值才是系统设计的最终测试标准”，而不仅仅是丰富的功能。[该比值是对易用性的一种测量，由简单和复杂应用共同验证。]

4.3 为了获得概念完整性，设计必须由一个人或者具有共识的小型团队来完成。

4.4 “对于非常大型的项目，将设计方法、体系结构方面的工作与具体实现相分离是获得概念完整性的强有力方法。”[同样适用于小型项目。]

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

4.6 纪律、规则对行业是有益的。外部的体系结构规定实际上是增强，而不是限制实现小组的创造性。

4.7 概念上统一的系统能更快地开发和测试。

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

第5章 画蛇添足

5.1 尽早交流和持续沟通能使结构师有较好的成本意识，以及使开发人员获得对设计的信心，并且不会混淆各自的责任分工。

5.2 结构师如何成功地影响实现：

牢记是开发人员承担创造性的实现责任；结构师只能提出建议。

时刻准备着为所指定的说明建议一种实现的方法，准备接受任何其他可行的方法。

对上述的建议保持低调和平静。

准备对所建议的改进放弃坚持。

听取开发人员在体系结构上改进的建议。

5.3 第二个系统是人们所设计的最危险的系统，通常的倾向是过分地进行设计。

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

5.5 为功能分配一个字节和微秒的优先权值是一个很有价值的规范化方法。

第6章 贯彻执行

6.1 即使是大型的设计团队，设计结果也必须由一个或两个人来完成，以确保这些决定是一致的。

6.2 必须明确定义体系结构中与先前定义不同的地方，重新定义的详细程度应该与原先的说明一致。

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

6.4 必须采用形式化定义和记叙性定义中的一种作为标准，另一种作为辅助措施；它们都可以作为表达的标准。

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

6.6 直接整合是一种强制推行软件的结构性标准的方法。[硬件上也是如此——考虑内建在ROM中的Mac WIMP接口。]

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

6.8 允许体系结构师对实现人员的询问做出电话应答解释是非常重要的，并且必须进行日志记录和整理发布。[电子邮件是一种可选的介质。]

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

第7章 为什么巴比伦塔会失败?

7.1 巴比伦塔项目的失败是因为缺乏交流，以及交流的结果——组织。

交流

7.2 “因为左手不知道右手在做什么，从而进度灾难、功能的不合理和系统缺陷纷纷出现。”由于对其他人的各种假设，团队成员之间的理解开始出现偏差。

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

项目工作手册

7.4 项目工作手册“不是独立的一篇文档，它是对项目必须产生的一系列文档进行组织的一种结构。”

7.5 “项目所有的文档都必须是该（工作手册）结构的一部分。”

7.6 需要尽早和仔细地设计工作手册结构。

7.7 事先制订了良好结构的工作手册“可以将后来书写的文字放置在合适的章节中”，并且可以提高产品手册的质量。

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

7.9 实时更新是至关重要的。

7.10 工作手册的使用者应该将注意力集中在上次阅读后的变更，以及关于这些变更重要性的评述。

7.11OS/360项目工作手册开始采用的是纸介质，后来换成了微缩胶片。

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

7.13 仍然需要用变更条和修订日期[或具备同等功能的方法]来标记文字；仍然需要后进先出（LIFO）的电子化变更小结。

7.14Parnas 强烈地认为使每个人看到每件事的目标是完全错误的；各个部分应该被封装，从而没有人需要或者允许看到其他部分的内部结构，只需要了解接口。

7.15Parnas的建议的确是灾难的处方。[Parnas让我认可了该观点，使我彻底地改变了想法。]

组织架构

7.16 团队组织的目标是为了减少必要的交流和协作量。

7.17 为了减少交流，组织结构包括了人力划分（divisionoflabor）和限定职责范围（specialization of function）。

7.18 传统的树状组织结构反映了权力的结构原理——不允许双重领导。

7.19 组织中的交流是网状，而不是树状结构，因而所有的特殊组织机制（往往体现成组织结构图中的虚线部分）都是为了进行调整，以克服树状组织结构中交流缺乏的困难。

7.20 每个子项目具有两个领导角色——产品负责人、技术主管或结构师。这两个角色的职能有着很大的区别，需要不同的技能。

7.21 两种角色中的任意组合可以是非常有效的：

产品负责人和技术主管是同一个人。

产品负责人作为总指挥，技术主管充当其左右手。

技术主管作为总指挥，产品负责人充当其左右手。

第8章 胸有成竹

8.1 仅仅通过对编码部分的估计，然后乘以任务其他部分的相对系数，是无法得出对整项工作的精确估计的。

8.2 构建独立小型程序的数据不适用于编程系统项目。

8.3 程序开发呈程序规模的指数增长。

8.4 一些发表的研究报告显示指数约为1.5。[Boehm的数据并不完全一致，在1.05和1.2之间变化。1]

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

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

8.7 Harr 的Bell 实验室数据显示对于已完成的产品，操作系统类的生产率大约是0.6KLOC/人年，编译类工作的生产率大约为2.2KLOC/人年。

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

8.9 Corbato的MIT项目MULTICS数据显示，在操作系统和编译器混合类型上的生产率是1.2KLOC/人年，但这些是PL/I的代码行，而其他所有的数据是汇编代码行。

8.10 在基本语句级别，生产率看上去是个常数。

8.11 当使用适当的高级语言时，程序编制的生产率可以提高5倍。

第9章 削足适履

9.1 除了运行时间以外，所占据的内存空间也是主要开销。特别是对于操作系统，它的很多程序是永久驻留在内存中。

9.2 即便如此，花费在驻留程序所占据内存上的金钱仍是物有所值的，比其他任何在配置上投资的效果要好。规模本身不是坏事，但不必要的规模是不可取的。

9.3 软件开发人员必须设立规模目标，控制规模，发明一些减少规模的方法——就如同硬件开发人员为减少元器件所做的一样。

9.4 规模预算不仅仅在占据内存方面是明确的，同时还应该指明程序对磁盘的访问次数。

9.5 规模预算必须与分配的功能相关联；在指明模块大小的同时，确切定义模块的功能。

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

9.7 在整个实现的过程期间，系统结构师必须保持持续的警觉，确保连贯的系统完整性。

9.8 培养开发人员从系统整体出发、面向用户的态度是软件编程管理人员最重要的职能。

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

9.10 临时空间的尺寸，以及每次磁盘访问的程序数量是很关键的决策，因为性能是规模的非线性函数。[这个整体决策已显得过时——起初是由于虚拟内存，后来则是成本低廉的内存。现在的用户通常会购买能容纳主要应用程序所有代码的内存。]

9.11 为了取得良好的空间－时间折衷，开发队伍需要得到特定与某种语言或者机型的编程技能培训，特别是在使用新语言或者新机器时。

9.12 编程需要技术积累，每个项目需要自己的标准组件库。

9.13 库中的每个组件需要有两个版本，运行速度较快和短小精炼的。[现在看来有些过时。]

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

9.15 这种突破常常是一种新型算法。

9.16 更普遍的是，战略上突破常来自于数据或表的重新表达。数据的表现形式是编程的根本。

第10章 提纲挈领

10.1 “前提：在一片文件的汪洋中，少数文档形成了关键的枢纽，每个项目管理的工作都围绕着它们运转。它们是经理们的主要个人工具。”

10.2 对于计算机硬件开发项目，关键文档是目标、手册、进度、预算、组织机构图、空间分配、以及机器本身的报价、预测和价格。

10.3 对于大学科系，关键文档类似：目标、课程描述、学位要求、研究报告、课程表和课程的安排、预算、教室分配、教师和研究生助手的分配。

10.4 对于软件项目，要求是相同的：目标、用户手册、内部文档、进度、预算、组织机构图和工作空间分配。

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

10.6 以上集合中每一个文档的准备工作都将注意力集中在对讨论的思索和提炼，而书写这项活动需要上百次的细小决定，正是由于它们的存在，人们才能从令人迷惑的现象中得到清晰、确定的策略。

10.7 对每个关键文档的维护提供了状态监督和预警机制。

10.8 每个文档本身就可以作为检查列表或者数据库。

10.9 项目经理的基本职责是使每个人都向着相同的方向前进。

10.10 项目经理的主要日常工作是沟通，而不是做出决定；文档使各项计划和决策在整个团队范围内得到交流。

10.11 只有一小部分管理人员的时间——可能只有20%——用来从自己头脑外部获取信息。

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

第11章 未雨绸缪

11.1 化学工程师已经认识到无法一步将实验室工作台上的反应过程移到工厂中，需要一个实验性工厂（pilotplanet）来为提高产量和在缺乏保护的环境下运作提供宝贵经验。

11.2 对于编程产品而言，这样的中间步骤是同样必要的，但是软件工程师在着手发布产品之前，却并不会常规地进行试验性系统的现场测试。[现在，这已经成为了一项普遍的实践，beta版本。它不同于有限功能的原型，alpha版本，后者同样是我所倡导的实践。]

11.3 对于大多数项目，第一个开发的系统并不合用。它可能太慢、太大，而且难以使用，或者三者兼而有之。

11.4 系统的丢弃和重新设计可以一步完成，也可以一块块地实现。这是个必须完成的步骤。

11.5 将开发的第一个系统——丢弃原型——发布给用户，可以获得时间，但是它的代价高昂——对于用户，使用极度痛苦；对于重新开发的人员，分散了精力；对于产品，影响了声誉，即使最好的再设计也难以挽回名声。

11.6 因此，为舍弃而计划，无论如何，你一定要这样做。

11.7 “开发人员交付的是用户满意程度，而不仅仅是实际的产品。”（Cosgrove）

11.8 用户的实际需要和用户感觉会随着程序的构建、测试和使用而变化。

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

11.10 目标上（和开发策略上）的一些正常变化无可避免，事先为它们做准备总比假设它们不会出现要好得多。

11.11 为变更计划软件产品的技术，特别是细致的模块接口文档——非常地广为人知，但并没有相同规模的实践。尽可能地使用表驱动技术同样是有所帮助的。[现在内存的成本和规模使这项技术越来越出众。]

11.12 高级语言的使用、编译时操作、通过引用的声明整合和自文档技术能减少变更引起的错误。

11.13 采用定义良好的数字化版本将变更量子（阶段）化。[当今的标准实践。]

为变更计划组织架构

11.14 程序员不愿意为设计书写文档的原因，不仅仅是由于惰性。更多的是源于设计人员的踌躇——要为自己尝试性的设计决策进行辩解。（Cosgrove）

11.15 为变更组建团队比为变更进行设计更加困难。

11.16 只要管理人员和技术人才的天赋允许，老板必须对他们的能力培养给予极大的关注，使管理人员和技术人才具有互换性；特别是希望能在技术和管理角色之间自由地分配人手的时候。

11.17 具有两条晋升线的高效组织机构，存在着一些社会性的障碍，人们必须警惕和积极地同它做持续的斗争。

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

11.19 组建外科手术队伍式的软件开发团队是对上述问题所有方面的彻底冲击。对于灵活组织架构问题，这的确是一个长期行之有效的解决方案。

前进两步，后退一步——程序维护

11.20 程序维护基本上不同于硬件的维护；它主要由各种变更组成，如修复设计缺陷、新增功能、或者是使用环境或者配置变换引起的调整。

11.21 对于一个广泛使用的程序，其维护总成本通常是开发成本的40%或更多。

11.22 维护成本受用户数目的严重影响。用户越多，所发现的错误也越多。

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

11.24 缺陷修复总会以（20－50）%的机率引入新的bug。

11.25 在每次修复之后，必须重新运行先前所有的测试用例，从而确保系统不会以更隐蔽的方式被破坏。

11.26 能消除、至少是能指明副作用的程序设计方法，对维护成本有很大的影响。

11.27 同样，设计实现的人员越少、接口越少，产生的错误也就越少。前进一步，后退一步——系统熵随时间增加

11.28 Lehman和Belady发现模块数量随大型操作系统（OS/360）版本号的增加呈线性增长，但是受到影响的模块以版本号指数的级别增长。

11.29 所有修改都倾向于破坏系统的架构，增加了系统的混乱程度。即使是最熟练的软件维护工作，也只是放缓了系统退化到不可修复混乱的进程，从中必须要重新进行设计。[许多程序升级的真正需要，如性能等，尤其会冲击它的内部结构边界。原有边界引发的不足常常在日后才会出现。]

第12章 干将莫邪

12.1 项目经理应该制订一套策略，以及为通用工具的开发分配资源，与此同时，他还必须意识到专业工具的需求。

12.2 开发操作系统的队伍需要自己的目标机器，进行调试开发工作。相对于最快的速度而言，它更需要最大限度的内存，还需要安排一名系统程序员，以保证机器上的标准软件是即时更新和实时可用的。

12.3 同时还需要配备调试机器或者软件，以便在调试过程中，所有类型的程序参数可以被自动计数和测量。

12.4 目标机器的使用需求量是一种特殊曲线：刚开始使用率非常低，突然出现爆发性的增长，接着趋于平缓。

12.5 同天文工作者一样，系统调试总是大部分在夜间完成。

12.6 抛开理论不谈，一次分配给某个小组连续的目标时间块被证明是最好的安排方法，比不同小组的穿插使用更为有效。

12.7 尽管技术不断变化，这种采用时间块来安排匮乏计算机资源的方式仍得以延续20年[在1975年]，是因为它的生产率最高。[在1995年依然如此]

12.8 如果目标机器是新产品，则需要一个目标机器的逻辑仿真装置。这样，可以更快地得到辅助调试平台。即使在真正机器出现之后，仿真装置仍可提供可靠的调试平台。

12.9 主程序库应该被划分成（1）一系列独立的私有开发库；（2）正处在系统测试下的系统集成子库；（3）发布版本。正式的分离和进度提供了控制。

12.10 在编制程序的项目中，节省最大工作量的工具可能是文本编辑系统。

12.11 系统文档中的巨大容量带来了新的不理解问题[例如，看看Unix]，但是它比
大多数未能详细描述编程系统特性的短小文章更加可取。

12.12 自顶向下、彻底地开发一个性能仿真装置。尽可能早地开始这项工作，仔细地听取 “它们表达的意见”。

高级语言

12.13 只有懒散和惰性会妨碍高级语言和交互式编程的广泛应用。[如今它们已经在全世界使用。]

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

12.15 传统的反对意见——功能、目标代码的尺寸、目标代码的速度，随着语言和编译器技术的进步已不再成为问题。

12.16 现在可供合理选择的语言是PL/I。[不再正确。]

交互式编程

12.17 某些应用上，批处理系统决不会被交互式系统所替代。[依然成立。]

12.18 调试是系统编程中很慢和较困难的部分，而漫长的调试周转时间是调试的祸根。

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

第13章 整体部分

13.1 第4、5、6章所意味的煞费苦心、详尽体系结构工作不但使产品更加易于使用，而且使开发更容易进行以及bug更不容易产生。

13.2V.A.Vyssotsky 提出，“许许多多的失败完全源于那些产品未精确定义的地方。”

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

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

13.5Wirth主张在每个步骤中，尽可能使用级别较高的表达方法。

13.6 好的自顶向下设计从四个方面避免了bug。

13.7 有时必须回退，推翻顶层设计，重新开始。

13.8 结构化编程中，程序的控制结构仅由支配代码块（相对于任意的跳转）的给定集合所组成。这种方法出色地避免了bug，是一种正确的思考方式。

13.9Gold 结果显示了，在交互式调试过程中，第一次交互取得的工作进展是后续交互的三倍。这实际上获益于在调试开始之前仔细地调试计划。[我认为在1995年依然如此。]

13.10 我发现对良好终端系统的正确使用，往往要求每两小时的终端会话对应于两小时的桌面工作：1小时会话后的清理和文档工作；1小时为下一次计划变更和测试。

13.11 系统调试（相对于单元测试）花费的时间会比预料的更长。

13.12 系统调试的困难程度证明了需要一种完备系统化和可计划的方法。

13.13 系统调试仅仅应该在所有部件能够运作之后开始。（这既不同于为了查出接口bug所采取 “合在一起尝试” 的方法；也不同于在所有构件单元的bug已知，但未修复的情况下，即开始系统调试的做法。）[对于多个团队尤其如此。]

13.14 开发大量的辅助调试平台（scaffolding 脚手架）和测试代码是很值得的，代码量甚至可能会有测试对象的一半。

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

13.16 系统测试期间，一次只添加一个构件。

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

第14章 祸起萧墙

14.1 “项目是怎样延迟了整整一年的时间？…一次一天。”

14.2 一天一天的进度落后比起重大灾难，更难以识别、更不容易防范和更加难以弥补。

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

14.4 里程碑必须是具体的、特定的、可度量的事件，能进行清晰能定义。

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

14.6 对于大型开发项目中的估计行为，政府的承包商所做的研究显示：每两周进行仔细修订的活动时间估计，随着开始时间的临近不会有太大的变化；期间内对时间长短的过高估计，会随着活动的进行持续下降；过低估计直到计划的结束日期之前大约三周左右，才有所变化。

14.7 慢性进度偏离是士气杀手。[Microsoft的Jim McCarthy说：“如果你错过了一个最终期限（deadline），确保制订下一条deadline。2”]

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

14.9 不存在关键路径进度的替代品，使人们能够辨别计划偏移的情况。

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

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

14.12 PERT图为前面那个泄气的借口，“其他的部分反正会落后”，提供了答案。

14.13 每个老板同时需要采取行动的异常信息以及用来进行分析和早期预警的状态数据。

14.14 状态的获取是困难的，因为下属经理有充分的理由不提供信息共享。

14.15 老板的不良反应肯定会对信息的完全公开造成压制；相反，仔细区分状态报告、毫无惊慌地接收报告、决不越俎代庖，将能鼓励诚实的汇报。

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

14.17 Vyssotsky：我发现在里程碑报告中很容易记录“计划（老板的日期）”和“估计（最基层经理的日期）”的日期。项目经理必须停止对这些日期的怀疑。”

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

第15章 另外一面

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

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

15.3 培训和管理人员基本上没有能向编程人员成功地灌输对待文档的积极态度——文档能在整个生命周期对克服懒惰和进度的压力起促进激励作用。

15.4 这样的失败并不都是因为缺乏热情或者说服力，而是没能正确地展示如何有效和经济地编制文档。

15.5 大多数文档只提供了很少的总结性内容。必须放慢脚步，稳妥地进行。

15.6 由于关键的用户文档包含了跟软件相关的基本决策，所以它的绝大部分需要在程序编制之前书写，它包括了9项内容（参见相应章节）。

15.7 每一份发布的程序拷贝应该包括一些测试用例，其中一部分用于校验输入数据，一部分用于边界输入数据，另一部分用于无效的输入数据。

15.8 对于必须修改程序的人而言，他们所需要程序内部结构文档，同样要求一份清晰明了的概述，它包括了5项内容（参见相应章节）。

15.9 流程图是被吹捧得最过分的一种程序文档。详细逐一记录的流程图是一件令人生厌的事情，而且高级语言的出现使它显得陈旧过时。（流程图是图形化的高级语言。）

15.10 如果这样，很少有程序需要一页纸以上的流程图。[在这一点上，MILSPEC 军用标准实在错得很厉害。]

15.11 即使的确需要一张程序结构图，也并不需要遵照ANSI的流程图标准。

15.12 为了使文档易于维护，将它们合并至源程序是至关重要的，而不是作为独立文档进行保存。

15.13 最小化文档负担的3个关键思路：

借助那些必须存在的语句，如名称和声明等，来附加尽可能多的“文档”信息。

使用空格和格式来表现从属和嵌套关系，提高程序的可读性。

以段落注释，特别是模块标题的形式，向程序中插入必要的记叙性文字。

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

15.15 在线系统的高级语言（应该使用的工具）中，自文档化技术发现了它的绝佳应用和强大功能。

原著结束语

E.1 软件系统可能是人类创造中最错综复杂的事物（从不同类型组成部分数量的角度出发）。

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