人月神话

001

焦油坑

过去几十年的大型系统开发就犹如一个焦油坑，很多大型动物在其中剧烈挣扎，他们中大多数开发出了可运行的系统--不过，其中只有非常少数的项目满足了目标、时间进度和预算的要求。

各种团队，大型的和小型的，庞杂的和精干的，一个接一个淹没在了焦油坑中。表面上看起来好像没有任何一个单独的问题会导致困难，每个都能被解决，但是当它们相互纠缠和累积在一起的时候，团队的行动就会变得越来越慢且很难看清问题的本质。

水平边界以下，程序变成*编程产品*（

Programming Product）。这是可以被任何人运行、

测试、修复和扩展的程序。它可以运行在多种操作系统平台上，供多套数据使用。要成为通

用的编程产品，程序必须按照普遍认可的风格来编写，特别是输入的范围和形式必须扩展，

以适用于所有可以合理使用的基本算法。接着，对程序进行彻底测试，确保它的稳定性和可

靠性，使其值得信赖。这就意味着必须准备、运行和记录详尽的测试用例库，用来检查输入

的边界和范围。此外，要将程序提升为程序产品，还需要有完备的文档，每个人都可以加以

使用、修复和扩展。经验数据表明，相同功能的编程产品的成本，至少是已经过测试的程序

的三倍

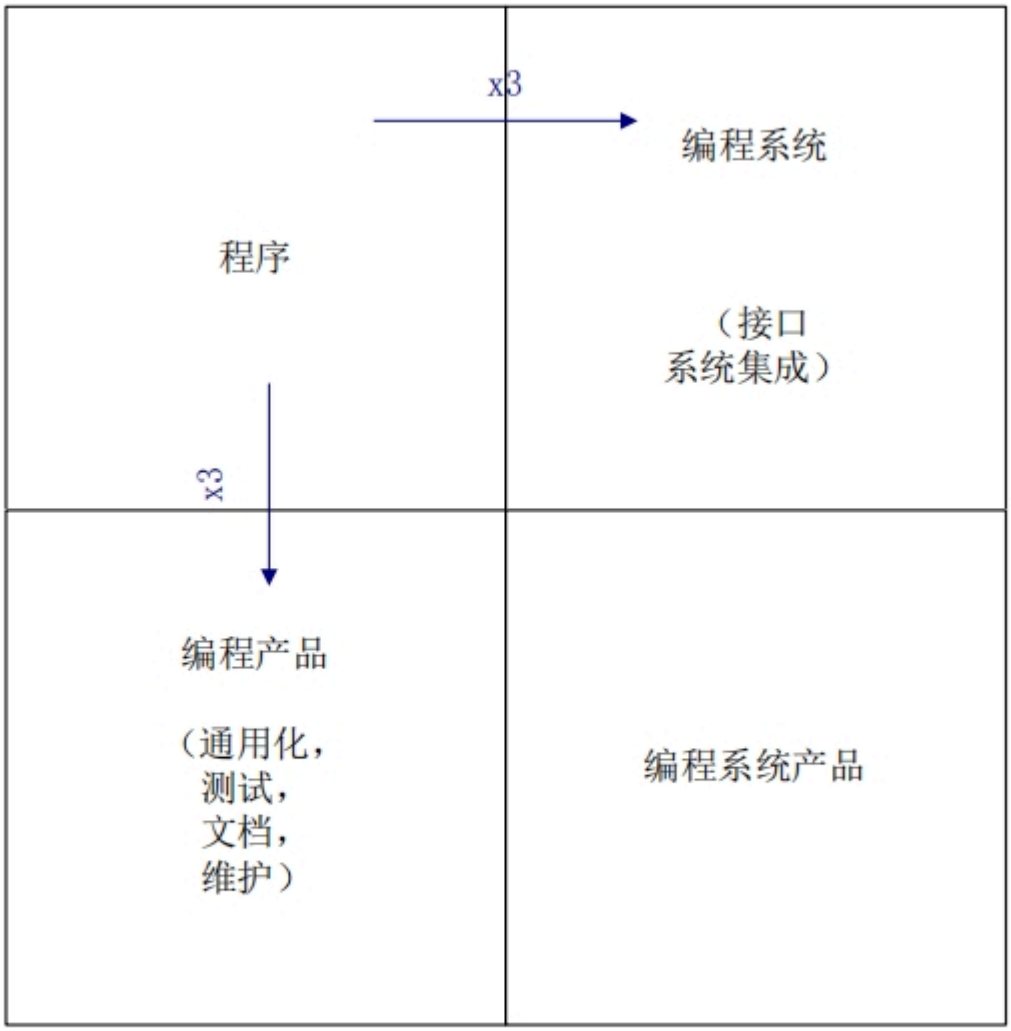


图 1.1：编程系统产品的演进

回到图中，垂直边界的右边，程序变成*编程系统*（Programming System）中的一个构件单元。它是在功能上能相互协作的程序集合，具有规范的格式，可以进行交互，并可以用来组装和搭建整个系统。要成为系统构件，程序必须按照一定的要求编制，使输入和输出在语法和语义上与精确定义的接口一致。同时程序还要符合预先定义的资源限制——内存空间、输入输出设备、计算机时间。最后，程序必须同其它系统构件单元一道，以任何能想象到的组合进行测试。由于测试用例会随着组合不断增加，所以测试的范围非常广。因为一些意想不到的交互会产生许多不易察觉的 bug，测试工作将会非常耗时，因此相同功能的编程系统构件的成本至少是独立程序的三倍。如果系统有大量的组成单元，成本还会更高。

图 1.1 的右下部分代表*编程系统产品*（

Programming Systems Product）。和以上的所

有的情况都不同的是，它的成本高达九倍。然而，只有它才是真正有用的产品，是大多数系统开发的目标

002人月神话

在众多软件项目中，缺乏合理的时间进度是造成项目滞后的最主要原因，它比其他所有因素加起来的影响还大。导致这种普遍性灾难的原因是什么呢?

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

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

第三，由于对自己的估算缺乏信心，软件经理通常不会有耐心持续地进行估算这项工作。

第四，对进度缺少跟踪和监督。其他工程领域中，经过验证的跟踪技术和常规监督程序，在软件工程中常常被认为是无谓的举动。

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

所有的编程人员都是乐观主义者。可能是这种现代魔术特别吸引那些相信美满结局的人；也可能是成百上千琐碎的挫折赶走了大多数人，只剩下了那些习惯上只关注结果的人；还可能仅仅因为计算机还很年轻，程序员更加年轻，而年轻人总是些乐观主义者——无论是什么样的程序，结果是勿庸置疑的：“这次它肯定会运行。”或者“我刚刚找出了最后一个错误。”

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

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

人数和时间的互换仅仅适用于以下情况：某个任务可以分解给参与人员，并且*他们之间不需要相互的交流（图 2.1）*。这在割小麦或收获棉花的工作中是可行的；而在系统编程中近乎不可能。

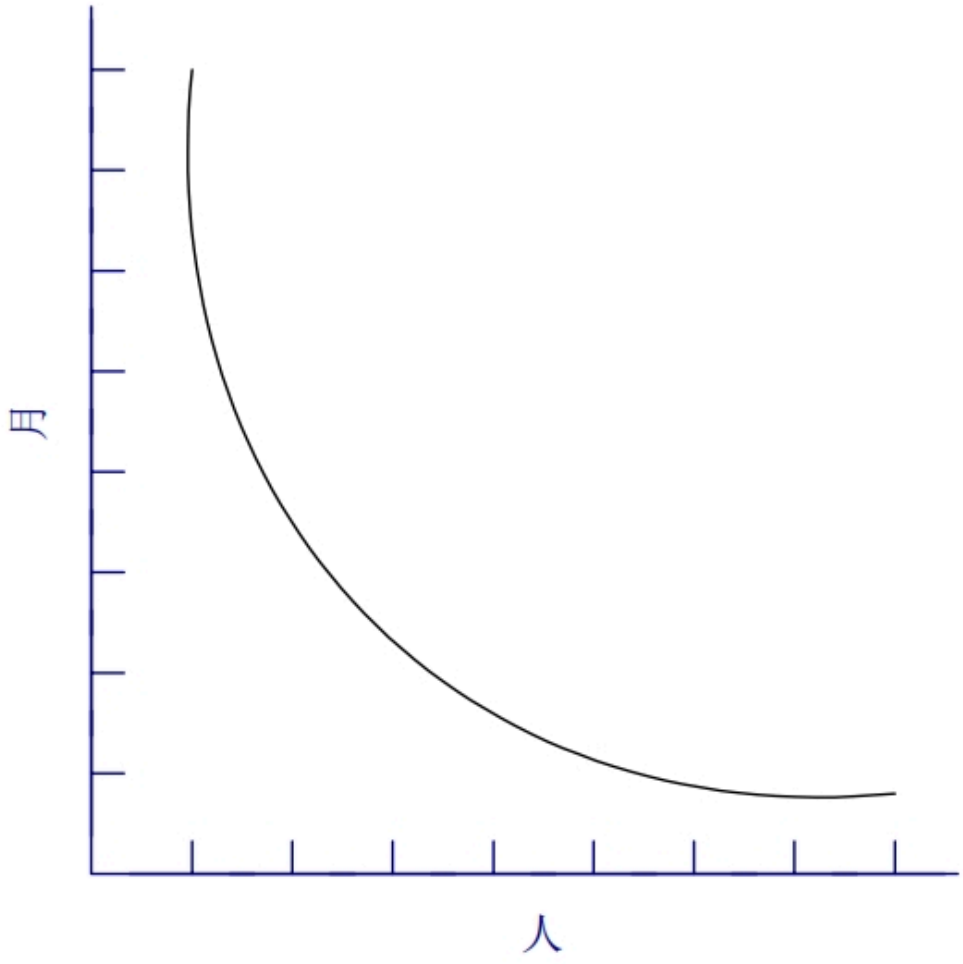


图 2.1：人员和时间之间的关系——完全可以分解的任务

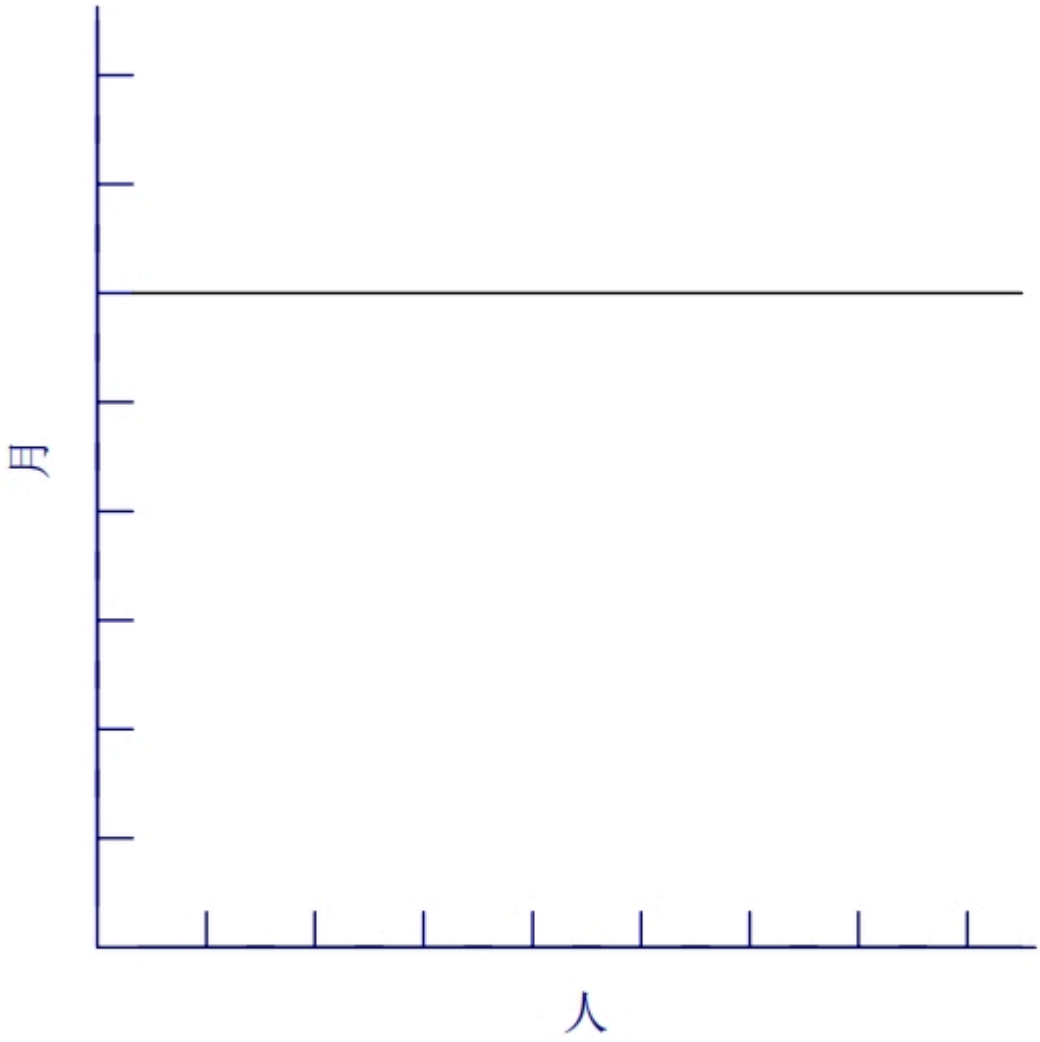


图 2.2：人员和时间之间的关系——无法分解的任务

。

对于可以分解，但子任务之间需要相互沟通和交流的任务，必须在计划工作中考虑沟通的工作量。因此，相同人月的前提下，采用增加人手来减少时间得到的最好情况，也比未调整前要差一些（图 2.3）。

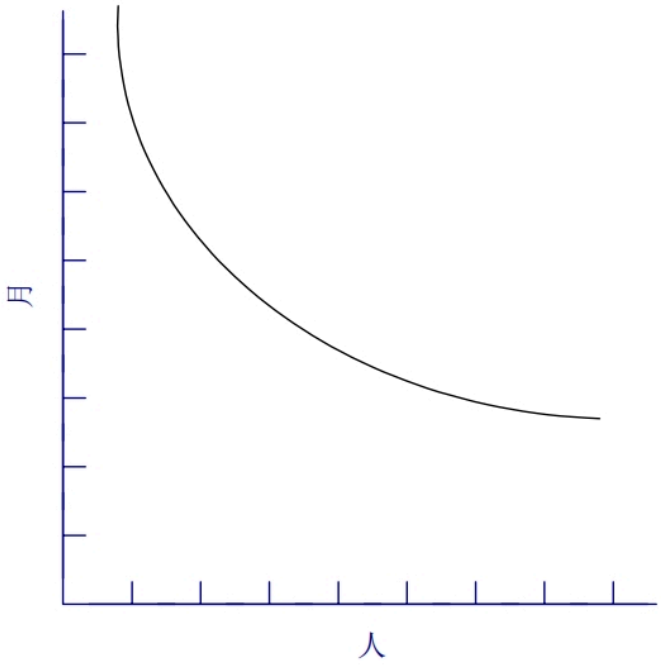


图 2.3：人员和时间之间的关系——需要沟通的可分解任务

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

相互之间交流的情况更糟一些。如果任务的每个部分必须分别和其他部分单独协作，则工作量按照 n(n-1)/2 递增。一对一交流的情况下，三个人的工作量是两个人的三倍，四个人则是两个人的六倍。而对于需要在三四个人之间召开会议、进行协商、一同解决的问题，情况会更加恶劣。所增加的用于沟通的工作量可能会完全抵消对原有任务分解所产生的作用，此时我们会被带到图 2.4 的境地。

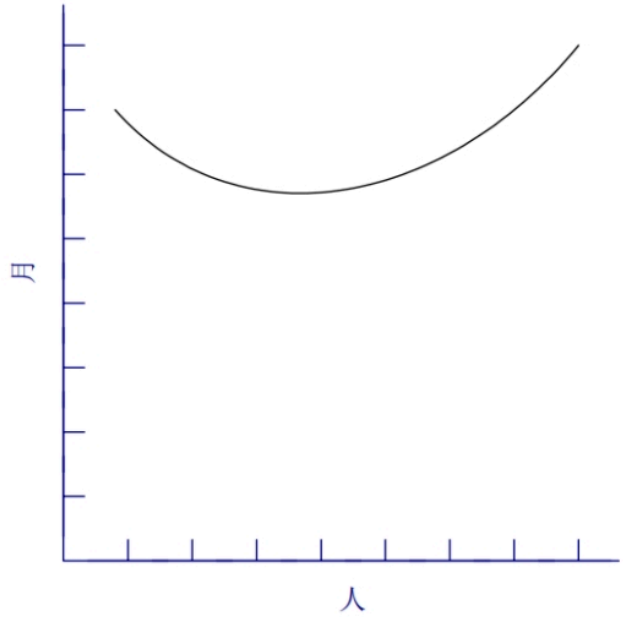


图 2.4：人员和时间之间的关系——关系错综复杂的任务

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

对于软件任务的进度安排，以下是我使用了很多年的经验法则：

1/3 计划

1/6 编码

- 10 -

月1/4 构件测试和早期系统测试

1/4 系统测试，所有的构件已完成

在许多重要的方面，它与传统的进度安排方法不同：

1. 分配给计划的时间比寻常的多。即便如此，仍不足以产生详细和稳定的计划规格说明，也不足以容纳对全新技术的研究和摸索。
2. 对所完成代码的调试和测试，投入近一半的时间，比平常的安排多很多。
3. 容易估计的部分，即编码，仅仅分配了六分之一的时间。

前面的讨论仅仅是第一个里程碑估计不当的情况。如果在 3 月 1 日，项目经理做出了比较保守的假设，即整个估计过于乐观了，如图 2.7 所示。6 个人手需要添加到原先的任务中。培训、任务的重新分配、系统测试工作量的计算作为练习留给读者。但是毫无疑问，重现“灾难”所开发出的产品，比没有增加人手，而是重新安排开发进度所产生的产品更差。

简单、武断地重复一下 Brooks 法则：

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

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。

003外科手术队伍（The Surgical Team）

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

得出的结论很简单：如果一个 200 人的项目中，有 25 个最能干和最有开发经验的项目经理，那么开除剩下的 175 名程序员，让项目经理来编程开发。

现在我们来验证一下这个解决方案。一方面，原有的开发队伍不是理想的*小型*强有力的团队，因为通常的共识是不超过 10 个人，而该团队规模如此之大，以至于至少需要两层的管理，或者说大约 5 名管理人员。另外，它需要额外的财务、人员、空间、文秘和机器操作方面的支持。

外科医生。Mills 称之为*首席程序员*。他亲自定义功能和性能技术说明书，设计程序，编制源代码，测试以及书写技术文档。他使用例如 PL/I 的结构化编程语言，拥有对计算机系统的访问能力；该计算机系统不仅仅能进行测试，还存储程序的各种版本，以允许简单的文件更新，并对他的文档提供文本编辑能力。首席程序员需要极高的天分、十年的经验和应用数学、业务数据处理或其他方面的大量系统和应用知识。

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

管理员。外科医生是老板，他必须在人员、加薪等方面具有决定权，但他决不能在这些事务上浪费任何时间。因而，他需要一个控制财务、人员、工作地点安排和机器的专业管理人员，该管理员充当与组织中其他管理机构的接口。Baker 建议仅在项目具有法律、合同、报表和财务方面的需求时，管理员才具有全职责任。否则，一个管理员可以为两个团队服务。

编辑。外科医生负责产生文档——出于最大清晰度的考虑，他必须书写文档。对内部描述和外部描述都是如此。而编辑根据外科医生的草稿或者口述的手稿，进行分析和重新组织，提供各种参考信息和书目，对多个版本进行维护以及监督文档生成的机制。

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

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

工具维护人员。现在已经有很多文件编辑、文本编辑和交互式调试等工具，因此团队很少再需要自己的机器和机器操作人员。但是这些工具使用起来必须毫无疑问地令人满意，而且需要具备较高的可靠性。外科医生则是这些工具、服务可用性的唯一评判人员。他需要一个工具维护人员，保证所有基本服务的可靠性，以及承担团队成员所需要的特殊工具（特别是交互式计算机服务）的构建、维护和升级责任。即使已经拥有非常卓越的、可靠的集中式服务，每个团队仍然要有自己的工具人员。因为他的工作是检查*他*的外科医生所需要的工具。工具维护人员常常要开发一些实用程序、编制具有目录的过程库以及宏库。

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

语言专家。随着 Algol 语言的出现，人们开始认识到大多数计算机项目中，总有一两个乐于掌握复杂编程语言的人。这些专家非常有帮助，很快大家会向他咨询。这些天才不同于外科医生，外科医生主要是系统设计者以及考虑系统的整体表现。而语言专家则寻找一种简洁、有效的使用语言的方法来解决复杂、晦涩或者棘手的问题。他通常需要对技术进行一些研究（两到三天）。通常一个语言专家可以为两个到三个外科医生服务。

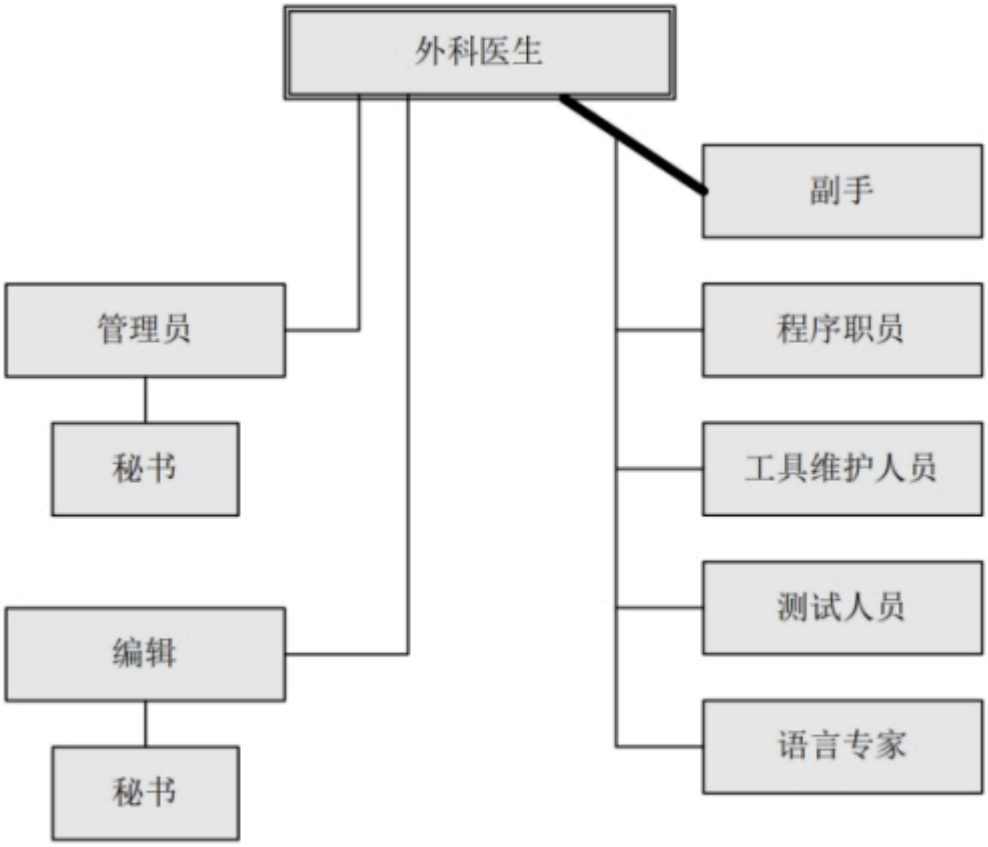


图 3.1：10 人程序开发队伍的沟通模式

004贵族专制、民主政治和系统设计（Aristocracy,

Democracy, and System Design）

我主张在系统设计中，概念完整性应该是最重要的考虑因素。也就是说为了反映一系

列连贯的设计思路，宁可省略一些不规则的特性和改进，也不提倡独立和无法整合的系统，

哪怕它们其实包含着许多很好的设计。在本章和以下的两章里，我们将解释在编程系统设计

中，这个主题的重要性。

对于非常大型的项目，将设计方法、体系结构方面的工作与具体实现相分离是获得概

念完整性的强有力方法。我亲眼目睹了它在 IBM 的 Stretch 计算机和 360 计算机产品线上的巨大成功。但同时我也看到了这种方法在 360 操作系统的开发中，由于缺乏广泛应用所遭受的失败。

现在让我们来处理具有浓厚感情色彩的问题——贵族统治和民主政治。结构师难道不

是新贵？他们一些智力精英，专门来告诉可怜的实现人员如何工作？是否所有的创造性活动被那些精英单独占有，实现人员仅仅是机器中的齿轮？难道不能遵循民主的理论，从所有的员工中搜集好的创意，以得到更好的产品，而不是将技术说明工作仅限定于少数人？

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

005管理

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

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

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

手册不但要描述包括所有界面在内的用户可见的一切，它同时还要避免描述用户看不见的事物。后者是编程实现人员的工作范畴，而实现人员的设计和创造是不应该被限制的。

体系结构设计人员必须为自己描述的任何特性准备*一种*实现方法，但是他不应该试图支配具体的实现过程。

一句古老的格言警告说：“决不要携带两个时钟出海，带一个或三个。”同样的原则也适用于形式化和记叙性定义。如果同时具有两种方式，则必须以一种作为标准，另一种作为

辅助描述，并照此明确地进行划分。它们都可以作为表达的标准，例如，Algol 68 采用形式化定义作为标准，记叙性文字作为辅助。PL/I 使用记叙性定义作为主要方式，形式化定义用作辅助表述。System/360 也将记叙性文字用作标准，以及形式化定义用作派生的论述。

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

周例会的决策会给出迅捷的结论，允许工作继续进行。如果任何人对结果*过于*不高兴，可以立刻诉诸于项目经理，但是这种情况非常少见。

这种会议的卓有成效是由于：

1. 数月内，相同小组——结构师、用户和实现人员——每周交流一次。因此，大家对项目相关的内容比较了解，不需要安排额外时间对人员进行培训。
2. 上述小组十分睿智和敏锐，深刻理解所面对的问题，并且与产品密切相关。没有人是“顾问”的角色，每个人都要承担义务。
3. 当问题出现时，在界线的内部和外部同时寻求解决方案。
4. 正式的书面建议集中了注意力，强制了决策的制订，避免了会议草稿纪要方式的不一致。
5. 清晰地授予首席结构师决策的权力，避免了妥协和拖延。

巴比伦塔的管理教训

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

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

1. *清晰的目标？* 是的，尽管幼稚得近乎不可能。而且，项目早在遇到这个基本的限制之前，就已经失败了。
2. *人力？* 非常充足。
3. *材料？* 在美索不达米亚有着丰富的泥土和柏油沥青。
4. *足够的时间？* 没有任何时间限制的迹象。
5. *足够的技术？* 是的，金字塔、锥形的结构本身就是稳定的，可以很好分散压力负载。

对砖石建筑技术，人们有过深刻的研究。同样，项目远在达到技术限制之间，就已经失败了。

那么，既然他们具备了所有的这些条件，为什么项目还会失败呢？他们还缺乏些什么？

两个方面——*交流，*以及交流的结果——*组织*。他们无法相互交谈，从而无法合作。当合作无法进行时，工作陷入了停顿。通过史书的字里行间，我们推测交流的缺乏导致了争辩、沮丧和群体猜忌。很快，部落开始分裂——大家选择了孤立，而不是互相争吵。

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

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

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

1. 任务（a mission）
2. 产品负责人（a producer）
3. 技术主管和结构师（a technical director or architect）
4. 进度（a schedule）
5. 人力的划分（a division of labor）
6. 各部分之间的接口定义（interface definitions among the parts）

产品负责人的角色是什么？他组建团队，划分工作及制订进度表。他要求，并一直要求必要的资源。这意味着他主要的工作是与团队外部，向上和水平地沟通。他建立团队内部

的沟通和报告方式。最后，他确保进度目标的实现，根据环境的变化调整资源和团队的构架。

那么技术主管的角色是什么？他对设计进行构思，识别系统的子部分，指明从外部看

上去的样子，勾画它的内部结构。他提供整个设计的一致性和概念完整性；他控制系统的复

杂程度。当某个技术问题出现时，他提供问题的解决方案，或者根据需要调整系统设计。用

Al Capp 所喜欢的一句谚语，他是“攻坚小组中的独行侠”（

inside-man at the skunk works.）。

他的沟通交流在团队中是首要的。他的工作几乎完全是技术性的。

技术主管作为总指挥，产品负责人充当其左右手。

我猜测最后一种安排对小型的团队是最好的选择，如同在第 3 章《外科手术队伍》一

文中所述。对于真正大型项目中的一些开发队伍，我认为产品负责人作为管理者是更合适的

安排

006胸有成竹（Calling the Shot)

必须声明的是，构建独立小型程序的数据不适用于编程系统产品。对规模平均 为 3200 指令的程序，如 Sackman、Erikson 和 Grant 的报告中所述，大约单个的程序员所需 要的编码和调试时间为 178 个小时，由此可以外推得到每年 35,800 语句的生产率。而规模

只有一半的程序花费时间大约仅为前者的四分之一，相应推断出的生产率几乎是每年

80,000 代码行 1 。计划、编制文档、测试、系统集成和培训的时间必须被考虑在内。因此，

上述小型项目数据的外推是没有意义的。就好像把 100 码短跑记录外推，得出人类可以在 3

分钟之内跑完 1 英里的结论一样。

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

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

Weinwurm[3] 的 SDC 研究报告同样显示出指数接近于 1.5。

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

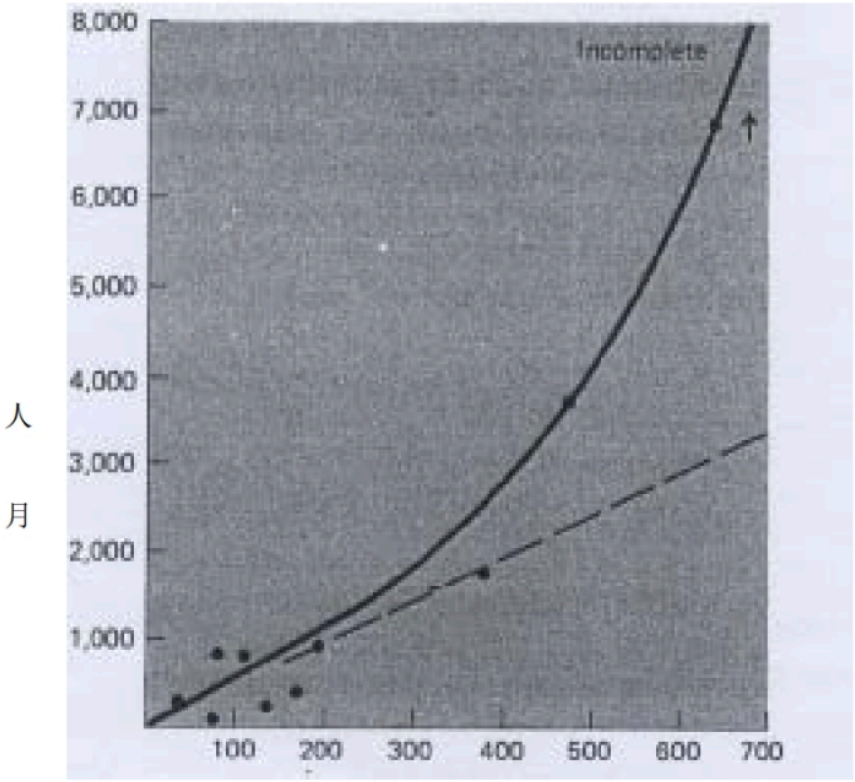
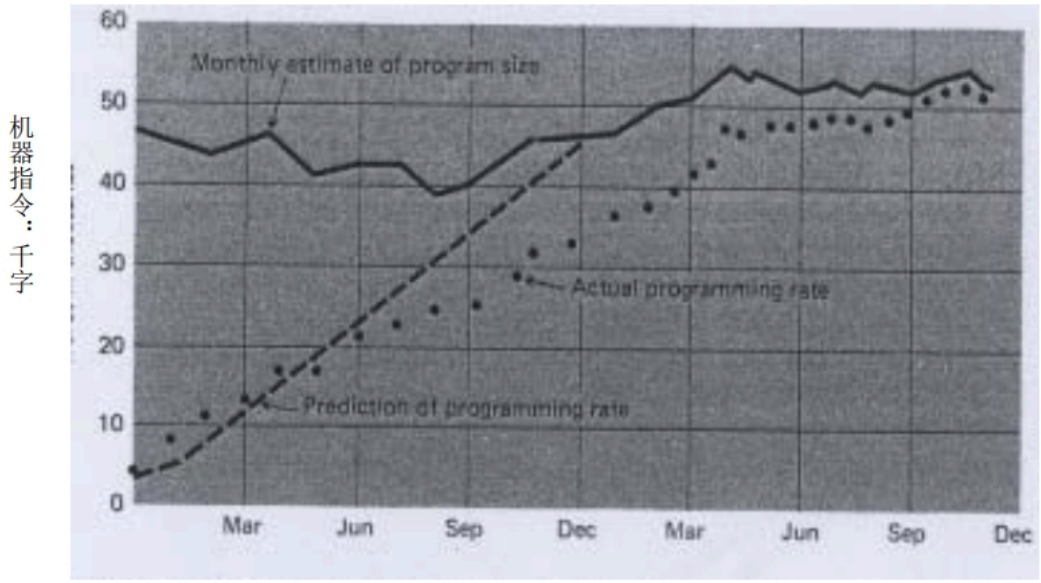


图 8.3 和 8.4 显示了一些有趣的数据，将实际的编程速度、调试速度与预期做了对比。

	程序单元	程序员人数	年	人年	程序字数	字/人年
操作性	50	83	4	101	52,000	515
维护	36	60	4	81	51,000	630
编译器	13	9	$2^1/_4$	17	38,000	2230
语言解释器（汇编）	15	13	$2^1/_2$	11	25,000	2270

图 8.2：4 个 NO.1 的 ESS 编程工作总结



注：

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

图 8.3：ESS 预计和实际的编程速度

Aron、Harr 和 OS/360 的数据都证实，生产率会根据任务本身复杂度和困难程度表现出显著差异。在复杂程度估计这片“沼泽”上的指导原则是：编译器的复杂度是批处理程序的三倍，操作系统复杂度是编译器的三倍 8 。

对常用编程语句而言。生产率似乎是固定的。这个固定的生产率包括了编程中需要注释，并可能存在错误的情况.

‰ 使用适当的高级语言，编程的生产率可以提高 5 倍

削足适履（Ten Pounds in a Five-Pound

Sack）

第三个更深刻的教训体现在以上的经验中。项目规模本身很大，缺乏管理和沟通，以

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

提纲挈领（The Documentary Hypothesis）

计算机产品的文档

如果要制造一台机器，哪些是关键的文档呢？

目标：定义待满足的目标和需要，定义迫切需要的资源、约束和优先级。

技术说明：计算机手册和性能规格说明。它是在计划新产品时第一个产生，并且最后

完成的文档。

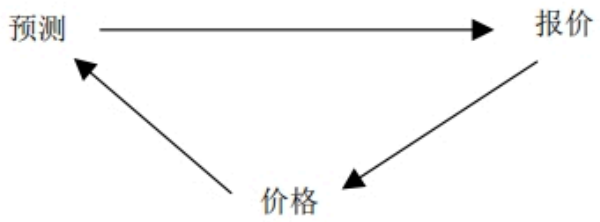
进度、时间表

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

组织机构图

工作空间的分配

报价、预测、价格：这三个因素互相牵制，决定了项目的成败。



大学科系的文档

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

目标

课程描述

学位要求

研究报告（申请基金时，还要求计划）

课程表和课程的安排

预算

教室分配

教师和研究生助手的分配

注意这些文档的组成与计算机项目非常相似：目标、产品说明、时间安排、资金分配、空间分派和人员的划分。只有价格文档是不需要的，学校的决策机构完成了这项任务。这种相似性不是偶然的——任何管理任务的关注焦点都是时间、地点、人物、做什么、资金。

在许多软件项目中，开发人员从商讨结构的会议开始，然后开始书写代码。不论项目的规模如何小，项目经理聪明的做法都是：立刻正式生成若干文档作为自己的数据基础，哪怕这些迷你文档非常简单。接着，他会和其他管理人员一样要求各种文档。

做什么：目标。定义了待完成的目标、迫切需要的资源、约束和优先级。

做什么：产品技术说明。以建议书开始，以用户手册和内部文档结束。速度和空间说明是关键的部分。

时间：进度表

资金：预算

地点：工作空间分配

人员：组织图。它与接口说明是相互依存的，如同 Conway 的规律所述：“设计系统的组织架构受到产品的约束限制，生产出的系统是这些组织机构沟通结构的映射。1 ”Conway 接着指出，一开始反映系统设计的组织架构图，肯定不会是正确的。如果系统设计能自由地变化，则项目组织架构必须为变化做准备。

未雨绸缪（Plan to Throw One Away）

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

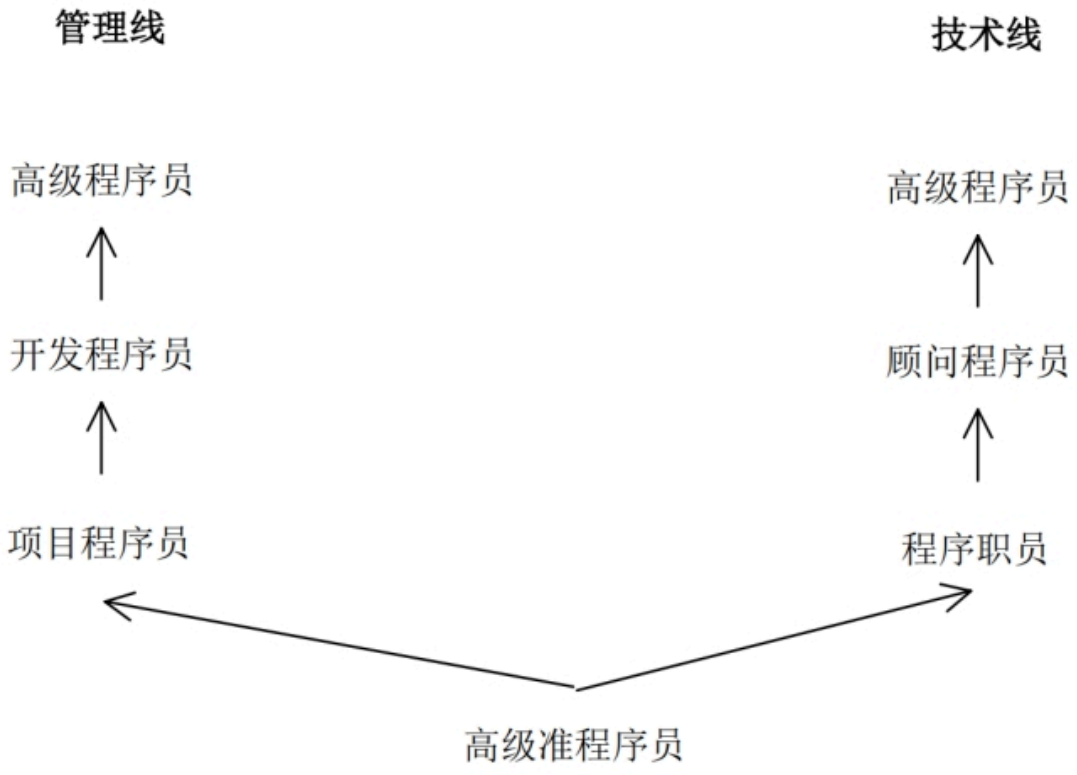


图 11.1：IBM 的两条职位晋升线

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

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

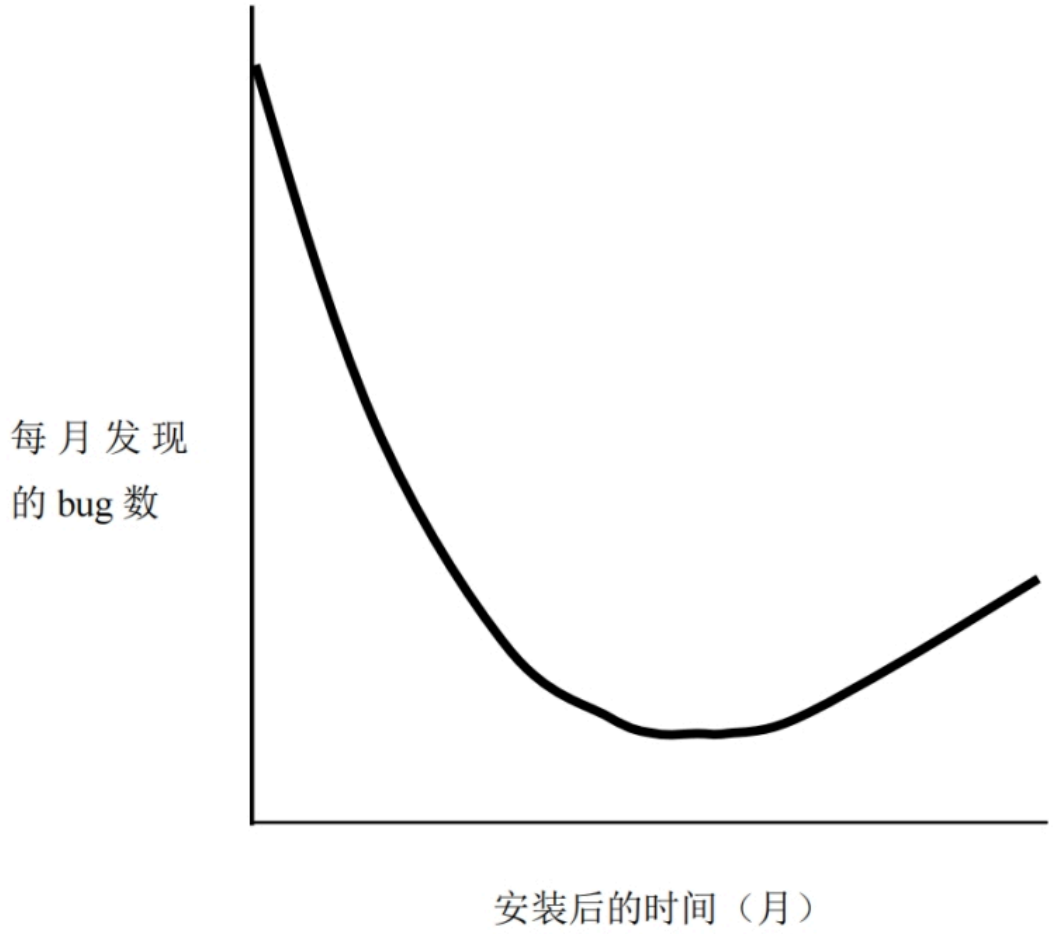


图 11.2：出现的 bug 数量是发布时间的函数

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

为什么缺陷不能更彻底地被修复？首先，看上去很轻微的错误，似乎仅仅是局部操作上的失败，实际上却是系统级别的问题，通常这不是很明显。修复局部问题的工作量很清晰，并且往往不大。但是，更大范围的修复工作常常会被忽视，除非软件结构很简单，或者文档

书写得非常详细。其次，维护人员常常不是编写代码的开发人员，而是一些初级程序员或者新手。

干将莫邪（Sharp Tools）

辅助机器和数据服务

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

- 73 - 后，仿真装置仍然可以提供*可靠*的调试平台。

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

现在，我们已经习惯于计算机硬件自始至终能正常工作。除非程序开发人员发现相同运算在运行时会产生不一致的结果，否则出错时，他都会被建议去检查自己代码中的错误，而不是去怀疑他的运行平台。

这样的经验，对于支持新型机器的编程工作来说，是不好的。实验室研制和试制的模型产品和早期硬件*不会*像定义的那样运行，*不会*稳定工作，甚至每天都*不会*一样。当一些缺陷被发现时，所有的机器拷贝，包括软件编程小组所使用的，都会发生修改。这种飘忽不定的开发基础实在是够糟的。而硬件失败，通常是间歇性的，导致情况更加恶劣。不确定性是所有情况中最糟糕的，因为它剥夺了开发人员查找 bug 的动力——可能根本就没有问题。所以，一套运行在稳定平台上的可靠仿真装置，提供了远大于我们所期望的功用。

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

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

程序库和管理。在 OS/360 开发中，一个非常成功的重要辅助机器应用是维护程序库。

该系统由 W. R. Crowley 带领开发，连接两台 7010 机器，共享一个很大的磁盘数据库。7010 同时还提供 System/360 汇编程序。所有经过测试或者正在测试的代码都保存在该库中，包括源代码和汇编装载模块。这个库实际上划分成不同访问规则下的子库。

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

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

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

这有两个重要的理念。首先是*受控*，即程序的拷贝属于经理，他可以独立地授权程序的变更。其次是使发布的进展变得正式，以及开发库（playpen）与集成、发布的*正式分离*。

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

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

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

文档系统。在所有的工具中，最能节省劳动力的，可能是运行在可靠平台上的、计算机化的文本编辑系统。我们有一套使用非常方便的系统，由 J. W. Franklin 发明。没有它，

OS/360 手册的进度可能会远远落后，而且更加晦涩难懂。另外，对于 6 英尺的 OS/360 手册，很多人认为它表达的是一大堆口头垃圾，巨大容量带来了新的不理解问题——这种观点有一些道理。

对此，我通过两种途径作出了反应。首先，OS/360 的文档规模是不可避免的，需要制订仔细的阅读计划。如果选择性地阅读，则可以忽略大部分内容和省下大量时间。人们必须把 OS/360 的文档看成是图书馆或者百科全书，而不是一系列强制阅读的文章。

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

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

整体部分（The Whole and the Parts）

我能召唤遥远的精灵。

那又怎么样，我也可以，谁都可以，问题是你真的召唤的时候，它们会来吗?

- 莎士比亚，《亨利四世》，第一部分

自顶向下的设计。在 1971 年的一篇论文中，Niklaus Wirth 把一种被很多最优秀的编程人员所使用的设计流程 2 形式化。尽管他的理念是为了程序设计，同样也完全适用于复杂系统的软件开发设计。**他将程序开发划分成体系结构设计、设计实现和物理编码实现，每个步骤可以使用自顶向下的方法很好地实现。**

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

好的自顶向下设计从几个方面避免了 bug。首先，清晰的结构和表达方式更容易对需求

和模块功能进行精确的描述。其次，模块分割和模块独立性避免了系统级的 bug。另外，细节的隐藏使结构上的缺陷更加容易识别。第四，设计在每个精化步骤的层次上是可以测试的，所以测试可以尽早开始，并且每个步骤的重点可以放在合适的级别上。

当遇到一些意想不到的问题时，按部就班的流程并不意味着步骤不能反过来，直到推翻顶层设计，重新开始整个过程。实际上，这种情况经常发生。至少，它让我们更加清楚在什么时候和为什么抛弃了某个臃肿的设计，并重新开始。一些糟糕的系统往往就是试图挽救一个基础很差的设计，而对它添加了很多表面装饰般的补丁。自顶向下的方法减少了这样的企图。

我确信在十年内，自顶向下进行设计将会是最重要的新型形式化软件开发方法。

结构化编程。另外一系列减少 bug 数量的新方法很大程度上来自 Dijkstra[3]。Bohm 和

Jacopini 的为其提供了理论证明[4]。

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

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

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

个日志中记录所有的变更，而在源代码中显著标记快速补丁和正式修改之间的区别，正式修改是完备并经过测试的，而且需要文档化。

祸起萧墙（Hatching a Catastrophe）

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

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

里程碑还是沉重的负担?

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

里程碑的选择只有一个原则，那就是，里程碑必须是具体的、特定的、可度量的事件，能够进行清晰定义。以下是一些反面的例子，例如编码，在代码编写时间达到一半的时候就

- 85 - 已经“90％完成”了；调试在大多时候都是“99％完成”的；“计划完毕”是任何人只要愿意，就可以声明的事件 1 。

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

里程碑有明显边界和没有歧义，比它容易被老板核实更为重要。*如果*里程碑定义得非

常明确，以致于无法自欺欺人时，很少有人会就里程碑的进展弄虚作假。但是如果里程碑很模糊，老板就常常会得到一份与实际情况不符的报告。毕竟，没有人愿意承受坏消息。这种做法只是为了起到缓和的作用，并没有任何蓄意的欺骗。

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

1. 如果在某项活动开始之前就着手估计，并且每两周进行一次仔细的修订。这样，随着开始时间的临近，无论最后情况会变得如何的糟糕，它都不会有太大的变化。
2. 活动*期间*，对时间长短的*过高*估计，会随着活动的进行持续下降。
3. *过低估计*在活动中不会有太大的变化，一直到计划的结束日期之前大约三周左右。

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

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

另外一面（The other face）

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

需要什么样的文档

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

使用程序。每个用户都需要一段对程序进行描述的文字。可是大多数文档只提供了很

少的总结性内容，无法达到用户要求，就像是描绘了树木，形容了树叶，但却没有一副森林的图案。为了得到一份有用的文字描述，就必须放慢脚步，稳妥地进行。

1. *目的。*主要的功能是什么？开发程序的原因是什么？
2. *环境。*程序运行在什么样的机器、硬件配置和操作系统上？
3. *范围。*输入的有效范围是什么？允许显示的合法范围是什么？
4. *实现功能和使用的算法。*精确地阐述它做了什么。
5. *输入 - 输出格式。*必须是确切和完整的。
6. *操作指令。*包括控制台及输出内容中正常和异常结束的行为。
7. *选项。*用户的功能选项有哪些？如何在选项之间进行挑选？
8. *运行时间。*在指定的配置下，解决特定规模问题所需要的时间？

- 93 - 9. *精度和校验。*期望结果的精确程度？如何进行精度的检测？

一般来说，三、四页纸常常就可以容纳以上所有的信息。不过往往需要特别注意的是表达的简洁和精确。由于它包含了和软件相关的基本决策，所以这份文档的绝大部分需要在程序编制之前书写。

验证程序。

除了程序的使用方法，还必须附带一些程序正确运行的证明，即测试用例。

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

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

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

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

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

修改程序。

调整程序或者修复程序需要更多的信息。显然，这要求了解全部的细节，

并且这些细节已经记录在注释良好的列表中。和一般用户一样，修改者迫切需要一份清晰明了的概述，不过这一次是关于系统的内部结构。那么这份概述的组成部分是什么呢?

1. 流程图或子系统的结构图，对此以下有更详细的论述。
2. 对所用算法的完整描述，或者是对文档中类似描述的引用。
3. 对所有文件规划的解释。
4. 数据流的概要描述——从磁盘或者磁带中，获取数据或程序处理的序列——以及在每个处理过程完成的操作。
5. 初始设计中，对已预见修改的讨论；特性、功能回调的位置以及出口；原作者对可能会扩充的地方以及可能处理方案的一些意见。另外，对隐藏缺陷的观察也同样很有价值。

- 94 - **流程图 （真的吗？？）**

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

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

因此，一页纸的流程图，成为表达程序结构、阶段或步骤的一种非常基本的图示。同样，它也非常容易绘制。图 15.1 展示了一个子程序流程图的图样

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

没有银弹 - 软件工程中的根本和次要问题

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

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

如果回顾一下软件领域中取得的最富有成效的三次进步，我们会发现每一次都是解决了软件构建上的巨大困难，但是这些困难不是本质属性，也不是主要困难。同样，我们可以

- 106 - 对每一次进步进行外推，来了解它们的固有限制。

高级语言

勿庸置疑，软件生产率、可靠性和简洁性上最有力的突破是使用高级语言编程。大多数观察者相信开发生产率至少提高了五倍，同时可靠性、简洁性和理解程度也大为提高。

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

统一编程环境。第一个集成开发环境——Unix 和 Interlisp 现在已经得到了广泛应用，并且使生产率提高了 5 倍。为什么？

现在，让我们来讨论一下当今可能作为潜在银弹的最先进的技术进步。它们各自针对什么样的问题？它们是属于必要问题，或者依然是解决我们剩下的次要困难？它们是提供了创新，还是仅仅是增量改进？

Ada 和其他高级编程语言。

面向对象编程。

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

“自动”编程。近四十年中，人们一直在预言和编写有关“自动编程”的文字，从问题的一段陈述说明自动产生解决问题的程序。现在，仍有一些人期望这样的技术能够成为下一个突破点 7 。

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

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

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

增量开发——增长，而非搭建系统。我现在还记得在 1958 年，当听到一个朋友提及*搭建（building），*而不是*编写（writing）*系统时，我所感受到的震动。一瞬间，我的整个软件开发流程的视野开阔了。这种暗喻是非常有力和精确的。现在，我们已经理解软件开发是如何类似于其他的建造过程，并开始随意地使用其他的暗喻，如*规格说明*、*构件装备*、*脚手架（测试平台）（specifications, assembly of components,* and *scaffolding）*。

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

‰ 尽可能早地、有系统地识别顶级的设计人员。最好的通常不是那些最有经验的人员。

‰ 为设计人员指派一位职业导师，负责他们技术方面的成长，仔细地为他们规划职业生涯。

‰ 为每个方面制订和维护一份职业计划，包括与设计大师的、经过仔细挑选的学习过程、正式的高级教育和以及短期的课程——所有这些都穿插在设计和技术领导能力的培养安排中。

为成长中的设计人员提供相互交流和学习的机会。

……昨天的复杂性是今天的规律。分子的无序性启迪了气体动力学理论和热力学的三大定律。现在，软件没有揭示类似的规律性原理，但是解释为什么没有的重担在你的身上。我不是迟钝和好辩的。我相信有一天软件的"复杂性"将以某种更高级的规律性概念来表达（就像物理学家的不变式）

《没有银弹》提出了全力解决复杂性问题的方法，这种方法可以在现实中取得十分乐观的进展。它倡导向软件系统增加必要的复杂性：

‰ 层次化，通过分层的模块或者对象。

‰ 增量化，从而系统可以持续地运行。

数学软件领域有着软件重用的长期传统：

我们推测重用的障碍不在生产者一边，而在消费者一边。如果一个软件工程师，潜在的标准化软件构件消费者，觉得寻找能满足他需要的构件，进行验证，比自行编写的代价更加昂贵时，重复的构件就会产生。注意我们上面提到的"觉得"。它和重新开发的真正投入无关。

数学软件上重用成功的原因有两个：（1）它很晦涩难懂，每行代码需要大量高智商的输入；（

2）存在丰富的标准术语，也就是用数学来描述每个构件的功能。因此，重新开发数学软件构件的成本很高，而查找现有构件功能的成本很低。数学软件界存在一些长期的传统——例如，专业期刊和算法搜集，用适度成本提供算法，出于商业考虑开发的高质量算法（尽管成本有些高，但依旧适度）等——使查找和发现满足某人需要的构件比其他很多领域要容

易。其他领域中，有时甚至不可能简洁地提出明确的要求。这些因素合在一起，使数学软件的重用比重新开发更有吸引力。

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

- 134 - 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

- 135 - 难。

2.5 但是，我们的构思是有缺陷的，因此总会有 bug。

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

2.7 在若干人员中分解任务会引发额外的沟通工作量——培训和相互沟通。

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

2.9 作为一个学科，我们缺乏数据估计。

2.10

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

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

2.12

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

第 3 章 外科手术队伍

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

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

3.3 小型、精干队伍是最好的——尽可能的少。

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

3.5 对于真正意义上的大型系统，小型精干的队伍太慢了。

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

- 136 - 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

- 137 - 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

- 138 - 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。

- 139 - 7.11 OS/360 项目工作手册开始采用的是纸介质，后来换成了微缩胶片。

  7.12

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

7.13

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

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

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

组织架构

7.16

团队组织的目标是为了减少必要的交流和协作量。

7.17

为了减少交流，组织结构包括了*人力划分（division of labor）*和*限定职责范围（specialization of function）*。

7.18

传统的树状组织结构反映了*权力*的结构原理——不允许双重领导。

7.19

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

7.20

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

7.21

两种角色中的任意组合可以是非常有效的：

‰ 产品负责人和技术主管是同一个人。

‰ 产品负责人作为总指挥，技术主管充当其左右手。

‰ 技术主管作为总指挥，产品负责人充当其左右手。

第 8 章 胸有成竹

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

8.2 *构建独立小型程序的数据不适用于编程系统项目。*

8.3 程序开发呈程序规模的指数增长。

8.4 一些发表的研究报告显示指数约为 1.5。[*Boehm 的数据并不完全一致，在 1.05 和 1.2 之间变化。__1*]

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

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

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

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

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

8.10

在基本语句级别，生产率看上去是个常数。

8.11

当使用适当的高级语言时，程序编制的生产率可以提高 5 倍。

第 9 章 削足适履

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

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

- 141 - 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

- 143 - 要一个*实验性工厂（pilot planet）*来为提高产量和在缺乏保护的环境下运作提供宝贵经验。

11.2

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

11.3

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

11.4

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

11.5

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

11.6

因此，*为舍弃而计划，无论如何，你一定要这样做。*

11.7

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

11.8

用户的实际需要和用户感觉会随着程序的构建、测试和使用而*变化*。

11.9

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

11.10

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

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

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

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

- 144 - **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。

- 145 - 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]

- 146 - 12.8

如果目标机器是新产品，则需要一个目标机器的逻辑仿真装置。这样，可以*更*

*快*地得到辅助调试平台。即使在真正机器出现之后，仿真装置仍可提供*可靠*的调试平台。

12.9

主程序库应该被划分成（

1）一系列独立的私有开发库；（

2）正处在系统测试

下的系统集成子库；（

3）发布版本。正式的分离和进度提供了控制。

12.10 在编制程序的项目中，节省最大工作量的工具可能是文本编辑系统。

12.11 系统文档中的巨大容量带来了新的不理解问题[例如，看看 Unix]，但是它比

大多数未能详细描述编程系统特性的短小文章更加可取。

12.12

自顶向下、彻底地开发一个性能仿真装置。尽可能早地开始这项工作，仔细地

听取 “它们表达的意见”。

高级语言

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

全世界使用。]

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

找。

12.15 传统的反对意见——功能、目标代码的尺寸、目标代码的速度，随着语言和编

译器技术的进步已不再成为问题。

12.16 现在可供合理选择的语言是 PL/I。[不再正确。]

交互式编程

12.17 某些应用上，批处理系统决不会被交互式系统所替代。[依然成立。]

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

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

第 13 章 整体部分

13.1

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

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

13.3

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

13.4

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

13.5 Wirth 主张在每个步骤中，尽可能使用级别较高的表达方法。

13.6

好的自顶向下设计从四个方面避免了 bug。

13.7

有时必须回退，推翻顶层设计，重新开始。

13.8

结构化编程中，程序的控制结构仅由支配代码块（相对于任意的跳转）的给定

集合所组成。这种方法出色地避免了 bug，是一种正确的思考方式。

13.9 Gold 结果显示了，在交互式调试过程中，第一次交互取得的工作进展是后续

交互的三倍。这实际上获益于在调试开始之前仔细地调试计划。[我认为在 1995 年*依然*如此。]

13.10 我发现对良好终端系统的正确使用，往往要求每两小时的终端会话对应于两小

时的桌面工作：1 小时会话后的清理和文档工作；1 小时为下一次计划变更和测试。

13.11 系统调试（相对于单元测试）花费的时间会比预料的更长。

13.12 系统调试的困难程度证明了需要一种完备系统化和可计划的方法。

13.13 系统调试仅仅应该在所有部件能够运作之后开始。（这既不同于为了查出接口

bug 所采取 “合在一起尝试” 的方法；也不同于在所有构件单元的 bug 已知，但未修复的

情况下，即开始系统调试的做法。）[对于多个团队尤其如此。]

13.14 开发大量的辅助调试平台（

scaffolding 脚手架）和测试代码是很值得的，代

- 148 - 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。

- 149 - 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—

- 150 - —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。

- 151 - **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。

核心观点：概念完整性和结构师

概念完整性。一个整洁、优雅的编程产品必须向它的每个用户提供一个条理分明的概念模型，这个模型描述了应用、实现应用的方法以及用来指明操作和各种参数的用户界面使用策略，。用户所感受到的产品概念完整性是易用性中最重要的因素。（当然还有其他因素。Macintosh 上所有应用程序界面的统一就是一个重要的例子。此外，有可能建立统一的接口，尽管它可能很粗糙，就像 MS-DOS。）

有很多由一个或者两个人设计的优秀软件产品例子。大多数纯智力作品，像书籍、音

- 154 - 乐等都是采用这种方式创作出来的。不过，很多产业的产品开发过程无法负担这种获取概念

完整性的直接方法。竞争带来了压力，很多现代工艺的最终产品是非常复杂的，它们的设计需要很多人月的工作量。软件产品十分复杂，在进度上的竞争也异常激烈。

结构师。从第 4 到第 7 章，我一直不断地在表达一个观点——委派一名产品*结构师*是最重要的行动。结构师负责产品所有方面的概念完整性，这些是用户能实际感受到的。结构师开发用于向用户解释使用的产品概念模型，概念模型包括所有功能的详细说明以及调用和控制的方法。结构师是这些模型的所有者，同时也是用户的代理。在不可避免地对功能、性能、规模、成本和进度进行平衡时，卓有成效地体现用户的利益。这个角色是全职工作，只有在最小的团队中，才能和团队经理的角色合并。结构师就像电影的导演，而经理类似于制片人。

特别的，他发现：15

‰ 第一次发布的成本最优进度时间，T = 2.5（MM）1/3。即，月单位的最优时间是估计工作量（人月）的立方根，估计工作量则由规模估计和模型中的其他因子导出。最优人员配备曲线是由推导得出的。

‰ 当计划进度比最优进度长时，成本曲线会缓慢攀升。时间越充裕，花的时间也越长。

‰ 当计划进度比最优进度短时，成本曲线急剧升高。

‰ *无论安排多少人手，几乎没有任何项目能够在少于 3/4 的最优时间内获得成功！*当高级经理向项目经理要求不可能的进度担保时，这段结论可以充分地作为项目经理的理论依据。

微型计算机革命改变了每个人使用计算机的方式。Schumacher 在 20 年前，陈述了面对的挑战：

我们真正想从科学家和技术专家那里得到什么？我会回答：我们需要这样的方法和设备：

‰ 价格足够低廉，使几乎所有人都能够使用

‰ 适合小型的应用，并且

‰ 满足人们对创造的渴望

传统软件产业。在 1975 年，软件产业拥有若干可识别的、但多少有些差异的组成部分，

如今他们依然存在：

‰ 计算机提供商：提供操作系统、编译器和一些实用程序

‰ 应用程序用户：如公共事业、银行、保险、政府机构等，他们为自己使用的软件开发应用程序包。

‰ 定制程序开发者：为用户承包开发私用软件包，需求、标准和行销步骤都是与众不同的。

‰ 商业包开发者：那个时候是为专业市场开发大型应用，如统计分析和 CAD 系统等

操作系统世界已经统一了。在 1975 年，存在着很多操作系统：每个硬件提供商每条产品线最少有一种操作系统，很多提供商甚至有两个。如今是多么的不同啊！开放式系统是基本原则。目前，人们主要在 5 大操作系统环境上行销自己的应用程序包（按照时间顺序）：

‰ IBM MVS 和 VM 环境

- 173 - ‰ DEC VMS 环境

‰ Unix 环境，某个版本

‰ IBM PC 环境，DOS、OS-2 或者 Windows

‰ Apple Macintosh 环境

塑料薄膜包装的成品软件产业。对于这个产业的开发者，面对的是与传统产业完全不同的经济学：软件成本是开发成本与数量的比值，包装和市场成本非常高。在传统内部的应用开发产业，进度和功能细节是可以协商的，开发成本则可能不行；而在竞争激烈的开发市

场面前，进度和功能支配了开发成本。

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

我依然记得那种向往和开心的感觉－当我在 1944 年 8 月 7 日读到哈佛大学 Mark I 型计算机研制成功的报道时－那时候我才 13 岁。Mark I 是电子机械学上的奇迹，哈佛大学的 Aiken 是它的构架设计师，而 IBM 的工程师 Clair Lake，Benjamin Durfee 和 Francis Hamilton 是它的实施设计师。同样令人向往的是读到 Vannevar Bush 1945 年 4 月发表在亚特兰大月刊上的论文“That We May Think”（我们的期望？）的时候，在这篇论文中，他建议将大量的知识组织成超文本的网络方式，并为用户提供机器从已有的链接以及指明其他的相关链接。

我对计算机的热情在 1952 年进一步高涨，因为得到了 IBM 在纽约恩迪科特的一份暑期工，正是那次，我有了在 IBM 604 上编程的实际经验，也了解了如何编制 IBM 701（它的第一个存储程序计算机）程序的正式指令；从哈佛大学 Aiken 和 Iverson 名下毕业终于让我的职业梦想变成了现实，并且，就这样沉迷了一辈子。感谢上帝，让我成为了为数不多的那些开开心心做着自己喜欢的工作的人之一。

我实在无法想像还有哪种生活会比热爱计算机更加激动人心，自从从真空管发展到集成

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

我依然记得那种向往和开心的感觉－当我在 1944 年 8 月 7 日读到哈佛大学 Mark I 型计算机研制成功的报道时－那时候我才 13 岁。Mark I 是电子机械学上的奇迹，哈佛大学的 Aiken 是它的构架设计师，而 IBM 的工程师 Clair Lake，Benjamin Durfee 和 Francis Hamilton 是它的实施设计师。同样令人向往的是读到 Vannevar Bush 1945 年 4 月发表在亚特兰大月刊上的论文“That We May Think”（我们的期望？）的时候，在这篇论文中，他建议将大量的知识组织成超文本的网络方式，并为用户提供机器从已有的链接以及指明其他的相关链接。

我对计算机的热情在 1952 年进一步高涨，因为得到了 IBM 在纽约恩迪科特的一份暑期工，正是那次，我有了在 IBM 604 上编程的实际经验，也了解了如何编制 IBM 701（它的第一个存储程序计算机）程序的正式指令；从哈佛大学 Aiken 和 Iverson 名下毕业终于让我的职业梦想变成了现实，并且，就这样沉迷了一辈子。感谢上帝，让我成为了为数不多的那些开开心心做着自己喜欢的工作的人之一。

我实在无法想像还有哪种生活会比热爱计算机更加激动人心，自从从真空管发展到集成