没有银弹
——本质与意外

弗雷德里克·*P*·布鲁克斯
北卡罗来纳大学教堂山分校

> 在技术或管理技术中，没有任何单一的发展能够在十年内承诺生产力、可靠性、简单性方面实现一个数量级的改进。

摘要[1]

所有软件构建都涉及基本任务，即构建组成抽象软件实体的复杂概念结构，以及偶然任务，即在编程语言中表示这些抽象实体并将其映射到空间和速度约束内的机器语言。 软件生产力中过去大部分的增益都来自于消除人为障碍，使偶然任务变得异常困难，比如严格的硬件限制、笨拙的编程语言、缺乏机器时间。 现在软件工程师所做的工作中有多少是专注于偶然的，而不是本质的？ 除非超过所有工作量的9/10都是偶然的，将所有偶然活动缩减到零时间将不会带来数量级的改进。

因此，看起来是时候着手解决软件任务的基本部分了，那些涉及构建极其复杂的抽象概念结构的部分。我建议：

- 利用大众市场，避免构建可以购买的东西。
- 将快速原型制作作为建立软件需求的计划迭代的一部分。
- 有机地发展软件，随着系统的运行、使用和测试，逐渐增加更多功能。
- 识别和培养新一代伟大的概念设计师。

介绍

在我们的民间传说中，所有的怪物中，没有比狼人更可怕的了，因为它们会意想不到地从熟悉变成恐怖。 为了对付它们，我们寻找能够神奇地让它们安息的银弹。

---

[1]转载自：Frederick P. Brooks，《神话般的程序员，带有*4*个新章节的周年纪念版》，*A*ddiso*n*-Wes*l*e*y*（1995年），本文再版自*IFIP*第十届世界计算大会论文集，H.-J. Kug*l*er主编，E*l*s*e*v*i*er Scie*n*ce B.V.，阿姆斯特丹，荷兰（1986年）第1069-76页。

熟悉的软件项目具有这种特点（至少在非技术经理看来），通常是无辜和直接的，但有可能变成错过时间表、超支和有缺陷产品的怪物。 因此，我们听到了对银弹的绝望呼声，希望有一种东西能够使软件成本像计算机硬件成本一样迅速下降。

但是，当我们展望十年后的未来时，我们看到没有银弹。 无论是技术还是管理技术方面的任何单一发展，都不能单独承诺生产率、可靠性和简单性方面的一个数量级的改进。 在本章中，我们将尝试看看为什么，同时检查软件问题的性质和提出的解决方案的属性。

怀疑并不意味着悲观。 尽管我们没有看到令人瞩目的突破，而且实际上，我们认为这与软件的性质不一致，但许多令人鼓舞的创新正在进行中。有纪律性、一致性的努力去开发、传播和利用它们确实会产生一个数量级的改进。

没有捷径，但有一条路。

对疾病管理的第一步是用细菌理论取代恶魔理论和体液理论。 这一步，希望的开始，却打破了所有对于神奇解决方案的希望。 它告诉工人们，进展将逐步取得，需要付出巨大努力，并且必须对清洁纪律付出持续不懈的关注。 所以今天的软件工程也是如此。

必须艰难吗？-基本困难

现在看来，软件中没有银弹，而且软件的本质使得不太可能会有任何发明能够像电子、晶体管和大规模集成电路对计算机硬件所做的那样，提高软件的生产力、可靠性和简单性。 我们不能指望每两年看到两倍的增长。

首先，我们必须观察到的异常不是软件进展如此缓慢，而是计算机硬件进展如此迅速。 自文明开始以来，没有其他技术在30年内看到了六个数量级的性价比提升。 在没有其他技术可以选择在性能提升或成本降低中获益。 这些收益来自于计算机制造业从装配业转变为过程业的转变。

其次，为了看看我们可以期望软件技术取得什么样的进展，让我们来看看它的困难。 继亚里士多德之后，我将它们分为本质–软件本质上固有的困难–和意外–今天伴随其生产但并非固有的那些困难。

我将在下一节讨论这些意外。 首先让我们考虑本质。

软件实体的本质是一种相互关联的概念构建：数据集、数据项之间的关系、算法和函数调用。 这种本质是抽象的，因为概念构建在许多不同的表现下是相同的。 尽管如此，它仍然非常精确和丰富详细。

我认为构建软件的难点在于这种概念构建的规范、设计和测试，而不是代表它和测试表现的劳动。 当然，我们仍然会出现语法错误；但与大多数系统中的概念错误相比，它们微不足道。

如果这是真的，构建软件将永远是困难的。 本质上没有银弹。

让我们考虑现代软件系统的本质属性：复杂性、一致性、可变性和不可见性。

复杂性。 软件实体比其他任何人造构造物都更复杂，因为没有两个部分是相同的（至少在语句级别以上）。.如果它们是相似的，我们将这两个相似的部分合并成一个子程序，开放或封闭。 在这方面，软件系统与计算机、建筑物或汽车有着根本的不同，这些领域中重复的元素随处可见。

数字计算机本身比大多数人造物品更复杂；它们具有非常多的状态。 这使得构思、描述和测试它们变得困难。 软件系统的状态数量比计算机多几个数量级。同样，将软件实体扩展不仅仅是将相同元素重复放大；它必然是不同元素数
量的增加。

在大多数情况下，元素以某种非线性方式相互作用，整体复杂性增加的远远超过线性增长。
软件的复杂性是其本质属性，而不是偶然属性。因此，对软件实体的描述如果忽略了其复杂性，往往也就忽略了其本质。 数学和物理科学通过构建复杂现象的简化模型、从模型中推导属性，并通过实验证实这些属性，连续三个世纪取得了巨大进展。 这种方法之所以奏效，是因为模型中忽略的复杂性并非现象的本质属性。 当复杂性就是本质时，这种方法就不再奏效。

许多开发软件产品的经典问题源于这种本质复杂性，随着规模的增大，这种非线性复杂性也在增加。 复杂性导致团队成员之间沟通困难，从而导致产品缺陷、成本超支和进度延误。 复杂性导致难以枚举，更不用说理解程序的所有可能状态，而由此而来的是不可靠性。 函数复杂性导致调用这些函数的困难，使得程序难以使用。结构复杂性导致难以扩展程序到新功能而不产生副作用。 结构复杂性导致未可视化的状态，构成安全陷阱。

问题不仅仅是技术问题，管理问题也源于复杂性。 这种复杂性使得概览变得困难，从而阻碍了概念完整性。
这使得找到并控制所有的松散端变得困难。 这造成了巨大的学习和理解负担，使得人员流动成为一场灾难。

一致性。 软件人员并不是唯一面对复杂性的人。.物理学处理着极其复杂的对象，甚至在“基本”粒子水平上。 然而，物理学家坚定地相信可以找到统一的原则，无论是在夸克还是统一场理论中。 爱因斯坦反复主张必须有关于自然的简化解释，因为上帝并不是反复无常或武断的。

没有这样的信念能安慰软件工程师。 他必须掌握的许多复杂性是任意的复杂性，由许多人为而无端的强加。

机构和系统，他的接口必须遵守。 这些因接口而异，不是因为必要性，而仅仅是因为它们是由不同的人设计的，而不是由上帝设计的。

在许多情况下，软件必须确认，因为它最近才出现在舞台上。 在其他情况下，软件必须遵守，因为它被认为是最符合要求的。 但在所有情况下，许多复杂性来自对其他接口的遵守；这不能仅通过对软件本身的任何重新设计来简化。

可变性。 软件实体不断受到变化的压力。.当然，建筑物、汽车和计算机也是如此。 但制造出来的东西在制造后很少被改变；它们被后来的型号取代，或者基本设计的后续序列号副本中包含了基本的变化。 汽车的召回确实相当少见；计算机的现场更改稍微少一些。两者都比现场软件的修改频率低得多。

部分原因是系统中的软件体现了其功能，而功能是最容易感受到变化压力的部分。 部分原因是软件可以更容易地进行更改–它是纯粹的思想材料，可以无限改变。 建筑实际上会发生变化，但所有人都了解的高昂的变更成本会抑制变更者的冲动。

所有成功的软件都会发生变化。 两个过程在起作用。 当一个软件产品被发现有用时，人们会在原始领域的边缘或之外尝试它的新用例。 对扩展功能的压力主要来自喜欢基本功能并为其发明新用途的用户。

其次，成功的软件也会超越最初编写的机器载具的正常寿命而存活下来。 如果没有新的计算机，至少会出现新的磁盘、新的显示器、新的打印机；软件必须符合其新的机会载具。

简而言之，软件产品嵌入在应用程序、用户、法律和机器载具的文化矩阵中。 所有这些都在不断变化，它们的变化无情地迫使软件产品发生变化。

隐形。 软件是看不见的，无法可视化的。.几何抽象是强大的工具。 建筑物的平面图帮助建筑师和客户评估空间、交通流量和视野。 矛盾变得明显，遗漏可以被发现。 机械零件的比例图和分子的棒图模型，虽然是抽象的，但具有相同的目的。 几何现实被捕捉在几何抽象中。

软件的现实并非固有地嵌入空间中。 因此，它在几何上没有现成的表示方式，就像土地有地图，硅片有图表，计算机有连接图。 一旦我们尝试绘制软件结构，我们发现它构成的不是一个，而是几个通用的有向图，相互叠加在一起。 这几个图可能代表控制流，数据流，依赖模式，时间顺序，命名空间关系。 这些通常甚至不是平面的，更不用说是分层的。事实上，其中一种方式是

建立对这种结构的概念控制是通过强制切割链接直到一个或多个图变得分层。[2]

尽管在限制和简化软件结构方面取得了进展，但它们仍然本质上无法可视化，因此剥夺了大脑一些最强大的概念工具。 这种缺乏不仅妨碍了一个人内部设计的过程，而且严重阻碍了不同思维之间的沟通。

过去的突破解决了偶然困难

如果我们审视过去在软件技术中最有成效的三个步骤，我们会发现每个步骤都攻击了构建软件中的不同主要困难，但它们是偶然的，而不是本质的困难。 我们还可以看到对每次攻击的外推的自然限制。

高级语言。 毫无疑问，对软件生产力、可靠性和简单性最有影响的一招是逐步使用高级语言进行编程。.大多数观察者认为，这种发展至少使生产力提高了五倍，并伴随着可靠性、简单性和可理解性的增加。

高级语言能实现什么？ 它使程序摆脱了许多偶然复杂性。 一个抽象程序由概念构造组成：操作、数据类型、序列和通信。. 具体的机器程序涉及位、寄存器、条件、分支、通道、磁盘等。 高级语言体现了抽象程序中所需的构造，并避免了所有较低级别的构造，从而消除了一个从未存在于程序中的复杂性层次。

高级语言能做的最多就是提供程序员在抽象程序中想象的所有构造。 确实，我们在思考数据结构、数据类型和操作方面的复杂程度不断提高，但增长速度却在逐渐减少。 语言发展越来越接近用户的复杂程度。

此外，高级语言的详细阐述在某一点上会成为增加用户智力任务的负担，而不是减少，因为用户很少使用神秘的构造。

分时共享。 大多数观察者认为，分时共享显着提高了程序员的生产力和产品质量，尽管没有高级语言带来的那么大改进。

分时共享攻击了一个明显不同的困难。 分时共享保持即时性，因此使我们能够保持对复杂性的概览。 批处理编程的缓慢反馈意味着我们不可避免地会忘记细枝末节，如果不是在停止编程并调用编译和执行时所思考的主要内容。 这种意识中断在时间上是昂贵的，因为我们必须重新刷新。 最严重的影响很可能是对复杂系统中正在发生的一切的理解力的衰减。

[2] Parnas, D.L., “为了便于扩展和收缩而设计软件,” *IEEE Trans. on SE*, 5, 2 (1979年3月), pp. 12-138.

慢的反馈，像机器语言的复杂性一样，是软件过程中偶然而非本质的困难。 分时共享的贡献限制直接来源于此。 其主要效果是缩短系统响应时间。 当它趋近于零时，在某一点它超过了人类可察觉的阈值，大约100毫秒。

除此之外，不应期望任何好处。

统一的编程环境。 Unix和Interlisp，第一个广泛使用的集成编程环境，被认为通过整体因素提高了生产力。为什么?

它们通过提供集成库、统一文件格式以及堆栈和过滤器来解决一起使用程序时的意外困难。 因此，原则上始终可以相互调用、传递和使用的概念结构在实践中确实可以轻松实现。

这一突破反过来刺激了整个工具库的发展，因为每个新工具都可以通过使用标准格式应用于任何程序。

由于这些成功，环境成为当今软件工程研究的重点。 我们将在下一节中看看它们的承诺和局限性。

对银的希望

现在让我们考虑最经常被提出作为潜在银弹的技术发展。 它们解决了什么问题? 它们是本质问题，还是我们意外困难的剩余部分? 它们提供革命性的进步，还是渐进的进步?

Ada和其他高级语言的进步。 最近最被吹捧的发展之一是编程语言**Ada**，这是上世纪80年代的通用高级语言。**. Ada**确实不仅反映了语言概念的演进改进，而且体现了鼓励现代设计和模块化概念的特点。 也许**Ada**的哲学比**Ada**语言更先进，因为它是模块化的哲学，抽象数据类型，分层结构化的哲学。

Ada可能过于丰富，这是对其设计提出要求的过程的自然产物。 这并不致命，因为子集工作词汇可以解决学习问题，硬件进步将为我们提供廉价的MIPS来支付编译成本。 推进软件系统的结构化确实是我们的美元购买的增加MIPS的非常好的用途。操作系统，在1960年代因其内存和周期成本而大声抨击，已被证明是利用过去硬件激增的MIPS和廉价内存字节的极好形式。

然而，Ada并不会证明是杀死软件生产力怪兽的银弹。 毕竟，这只是另一种高级语言，这些语言带来的最大回报来自第一个过渡，从机器的意外复杂性转向更抽象的逐步解决方案陈述。

一旦那些意外被消除，剩下的意外就会更小，它们的消除带来的回报肯定会更少。

我预测十年后，当评估Ada的有效性时，人们会看到它产生了实质性的差异，但并不是因为任何特定的语言特性，也不是因为所有这些特性的结合。新的Ada也不会

环境证明是改进的原因。 Ada最大的贡献将是切换到它促使程序员接受现代软件设计技术的培训。

面向对象编程。 许多艺术的学生对面向对象编程抱有比当今任何其他技术潮流更大的希望。[3] 我就是其中之一。. 达特茅斯的马克·谢尔曼指出，我们必须小心区分两个概念，即抽象数据类型和分层类型，也称为类。 抽象数据类型的概念是对象的类型应该由名称、一组适当的值和一组适当的操作来定义，而不是它的存储结构，存储结构应该是隐藏的。 例如**Ada**包（带有私有类型）或M**od**u**la**的模块。

层次类型，如Simula-67的类，允许定义可以通过提供下级类型进一步完善的通用接口。 这两个概念是正交的–可能存在没有隐藏的层次结构和没有层次结构的隐藏。
这两个概念代表了构建软件艺术上的真正进步。

每个概念都从过程中消除了一个意外困难，使设计者能够表达设计的本质，而无需表达大量不添加新信息内容的语法材料。 对于抽象类型和层次类型，结果是消除更高级别的意外困难，并允许更高级别的设计表达。

然而，这种进步只能消除设计表达中的所有意外困难。 设计本身的复杂性是必不可少的；这样的进展对此毫无影响。 只有在我们编程语言中今天仍然存在的类型规范的不必要废物本身占据了设计程序产品所涉及工作的九成时，面向对象编程才能实现一个数量级的增益。我对此表示怀疑。

人工智能。 许多人期望人工智能的进步能够提供革命性的突破，从而在软件生产力和质量方面实现数量级的增长。[4] 我不这么认为。要明白原因，我们必须剖析“人工智能”指的是什么，然后看它如何应用。

帕纳斯澄清了术语混乱：

> 今天普遍使用两种截然不同的人工智能定义。 *AI-1*：使用计算机解决以前只能通过应用人类智能才能解决的问题。 *AI-2*：使用一组特定的编程技术，称为启发式或基于规则的编程。 在这种方法中，研究人员研究专家
>
> 确定他们在解决问题时使用的启发式或经验法则。 . . . . . .这个程序被设计为以人类似乎解决问题的方式来解决问题。

---

[3]布奇，G.，“面向对象设计”，在使用*Ada*进行软件工程。 加利福尼亚州门洛帕克：本杰明·卡明斯，1983年。

[4]莫斯托，J.，主编，人工智能和软件工程专题，IEEE软件工程杂志，**11**, 11 (1985年11月).

> 第一个定义有一个不断变化的含义. . . . . .某些东西可以符合*AI-1*的定义今天，但是，一旦我们看到程序如何工作并理解问题，我们将不再将其视为*AI*. . . . . 不幸的是，我无法确定一套技术体系这是这个领域独有的. . . . . 大部分工作是特定于问题的，还有一些需要抽象或创造力来看如何转移它.[5]

我完全同意这一批评. 用于语音识别的技术似乎与用于图像识别的技术没有什么共同之处，两者都与专家系统中使用的技术不同. 例如，我很难看出图像识别如何在编程实践中产生任何可感知的差异.

语音识别也是如此。 构建软件的困难之处在于决定要说什么，而不是说出来。没有表达的便利性能带来的只是边际收益。

专家系统技术，AI-2，值得拥有自己的章节。

专家系统。 人工智能艺术中最先进的部分，也是最广泛应用的部分，是用于构建专家系统的技术。.许多软件科学家正在努力将这项技术应用于软件构建环境。[6]概念是什么，前景如何?

专家系统是一个包含通用推理引擎和规则库的程序，旨在接受输入数据和假设，并通过从规则库中推导出的推理来探索逻辑结果，得出结论和建议，并提供通过追溯其推理为用户解释其结果的方式。 推理引擎通常可以处理模糊或概率数据和规则，除了纯粹的确定性逻辑。

这些系统在达到相同问题的相同解决方案方面比编程算法提供了一些明显的优势：

- 推理引擎技术是以应用无关的方式开发的，然后应用于许多用途。 人们可以在推理引擎上投入更多的努力。
  事实上，该技术已经非常先进。
- 应用特定材料的可变部分以统一的方式编码在规则库中，并提供了用于开发、更改、测试和文档化规则库的工具。 这样可以使应用程序本身的复杂性得到规范化。

爱德华·费根鲍姆表示，这类系统的力量并不来自越来越花哨的推理机制，而是来自反映现实世界更准确的知识库。 我认为技术提供的最重要进步是将应用程序复杂性与程序本身分离。

这如何应用于软件任务？ 有很多方式：建议界面规则，提供建议测试策略，记住按钮类型频率，提供优化提示等。

---

[5] Parnas, D.L., “战略防御系统的软件方面,”*ACM*通讯, **28**, 12 (1985年12月, 1985), 页码 1326-1335. 同时发表在美国科学家, **73**, 5 (1985年9-10月, 1985), 页码 432-440.

[6] Balzer, R., “自动编程的15年展望,” 在Mostow着作中.

考虑一个虚构的测试顾问，例如。 在其最基本的形式中，诊断专家系统非常类似于飞行员的清单，基本上提供可能导致困难的原因建议。 随着规则库的发展，建议变得更具体，更复杂地考虑所报告的故障症状。 人们可以想象一个调试助手，起初提供非常概括的建议，但随着越来越多的系统结构体现在规则库中，提出的假设和推荐的测试变得越来越具体。 这样一个专家系统可能在最根本上最大程度地偏离传统系统，因为它的规则库应该像相应的软件产品一样进行分层模块化，这样当产品进行模块化修改时，诊断规则库也可以进行模块化修改。

生成诊断规则所需的工作是必须完成的工作，因为在为模块和系统生成测试用例集时，这些工作必须完成。 如果以一种适当通用的方式进行，具有规则的统一结构和良好的推理引擎可用，实际上可能会减少生成调试测试用例的总体工作量，同时有助于终身维护和修改测试。 同样，我们可以假设其他顾问，可能有许多，可能是简单的，用于软件构建任务的其他部分。

许多困难阻碍了早期实现对程序开发人员有用的专家顾问。 我们想象情景的一个关键部分是开发从程序结构规范到自动或半自动生成诊断规则的简便方法。 更加困难和重要的是知识获取的双重任务：找到能够解释清楚、自我分析的专家，他们知道为什么要做某些事情；并开发高效的技术，提取他们所知道的内容并将其提炼成规则库。 构建专家系统的基本前提是拥有专家。

专家系统最强大的贡献肯定是将最好程序员的经验和积累的智慧服务于经验不足的程序员。 这不是小小的贡献。 最佳软件工程实践与平均实践之间的差距非常大–也许比其他任何工程学科都要大。 传播良好实践的工具将是重要的。

“自动”编程。 近40年来，人们一直在期待和写作“自动编程”，即根据问题规范生成解决问题的程序。. 今天有些人写作时似乎期待这项技术能够带来下一个突破。[7]

帕纳斯暗示该术语被用于迷人而非语义内容，并断言，

> 简而言之，自动编程一直是比程序员当前可用的更高级语言编程的委婉说法。[8]

他认为，本质上，在大多数情况下，需要给出解决方法的规范，而不是问题本身。

[7] 莫斯托，引用文献
[8] 帕纳斯，1985年，引用文献

可以找到异常。 构建生成器的技术非常强大，并且在用于排序的程序中通常被很好地利用。 一些用于积分微分方程的系统还允许直接指定问题。

系统评估参数，从解决方法库中选择，并生成程序。

- 这些应用具有非常有利的特性：
- 问题可以通过相对较少的参数轻松表征。
- 已知许多解决方法，以提供备选库。
- 广泛的分析已经导致了明确的规则，用于选择解决方案技术，给定问题参数。

很难看到这些技术如何推广到普通软件系统的更广泛世界，那里具有如此整洁特性的情况是例外。 甚至很难想象这种泛化突破如何可能发生。

图形编程。 在软件工程中，图形或视觉编程是博士论文的一个热门主题，即将计算机图形应用于软件设计。[9] 有时，这种方法的承诺是根据与VLSI芯片设计的类比假设的，计算机图形在那里发挥着如此丰硕的作用。

有时候，通过将流程图视为理想的程序设计媒介来证明这种方法是合理的，并为构建它们提供强大的工具。

从这些努力中，没有什么令人信服的，更不用说令人兴奋的东西出现了。 我相信没有什么会发生。

首先，正如我在其他地方所论证的那样，流程图是对软件结构的一个非常糟糕的抽象。[10] 事实上，最好将其视为伯克斯、冯·诺伊曼和戈德斯坦为他们提出的计算机提供迫切需要的高级控制语言的尝试。 在今天流程图已经被详细阐述为可怜的、多页的、连接框形式时，它已被证明在设计工具方面基本无用程序员在编写描述程序之前绘制流程图。

其次，今天的屏幕在像素方面太小，无法显示任何严肃详细软件图的范围和分辨率。 今天工作站所谓的“桌面隐喻”实际上是一个“飞机座位隐喻”。 任何在两个胖乘客之间的教练座位上整理一大堆文件的人都会意识到这种差异–一个人一次只能看到很少的东西。 真正的桌面提供了对多页的概览和随机访问。 此外，当创造力爆发时，不止一个程序员或作家已经被知道放弃桌面，转而选择更宽敞的地板。 硬件技术必须有相当大的进步，才能使我们的范围足以满足软件设计任务的需求。

更根本地，正如我在上面所说的，软件是非常难以可视化的。
无论我们是绘制控制流程，变量作用域嵌套，变量交叉引用，数据膨胀，分层数据结构，还是其他任何东西，我们只感觉到软件大象错综复杂地相互交织的一个维度。 如果我们将许多相关视图生成的所有图表叠加在一起，很难提取出任何全局概览。VLSI

---

[9] Raeder, G., “当前图形编程技术概述”，见于 R. B. Grafton 和 T. Ichikawa 编，视觉编程专题，计算机，**18**，**8**（**198**5 年 **8** 月），第 **11**-25 页 **1**0Brooks **1**995，前作，第 **1**5 章。

类比基本上是误导的–芯片设计是一个反映其本质的分层二维对象。 软件系统不是。

程序验证。 现代编程中的大部分工作都投入到测试和修复错误中。. 也许在系统设计阶段消除错误可以找到一种银弹吗？ 通过在实施和测试之前证明设计的正确性，是否可以通过遵循完全不同的策略来根本上提高生产力和产品可靠性？

我不相信我们会在这里找到魔法。 程序验证是一个非常强大的概念，对于安全操作系统内核等事物非常重要。
然而，技术并不承诺节省劳动力。 验证工作量如此之大，以至于只有少数实质性的程序曾经被验证过。
程序验证并不意味着无错误的程序。 这里也没有魔法，。 数学证明也可能存在错误。 因此，虽然验证可能减少程序测试的负担，但无法消除它。

更严重的是，即使完美的程序验证只能证明程序符合其规范。 软件任务中最困难的部分是得出完整一致的规范，构建程序的本质实际上在于调试规范。

环境和工具。 对于更好的编程环境的爆炸性研究，还能期待多大收益？？一个人的直觉反应是，首先攻克的是最有回报的问题，并且已经解决了：分层文件系统，统一文件格式以便具有统一的程序接口，以及通用工具。 特定语言的智能编辑器尚未被广泛应用于实践，但它们所承诺的最多只是摆脱语法错误和简单语义错误。

也许在编程环境中尚未实现的最大收益是使用集成数据库系统来跟踪程序员必须准确记住并在单个系统的协作者组中保持最新的无数细节。

毫无疑问，这项工作是值得的，而且它肯定会在生产力和可靠性方面取得一些成果。 但从本质上讲，从现在开始的回报必须是边际的。

工作站。 从个人工作站的功率和内存容量的确定和快速增加中，软件艺术可以期待什么收益？？嗯，一个人能够有效利用多少MIPS？ 今天的速度完全支持程序和文档的构成和编辑。 编译可能需要提升，但是机器速度提高10倍肯定会使思考时间成为程序员一天中的主要活动。事实上，现在似乎是这样。

我们当然欢迎更强大的工作站。 我们不能期待从中获得神奇的增强。

对概念本质的有希望的攻击

尽管没有技术突破承诺在硬件领域中我们如此熟悉的神奇结果，但现在正在进行大量良好的工作，并且有稳定的，虽不起眼但持续的进展。

对软件过程的技术攻击在根本上受到生产力方程的限制：

任务时间 = $\sum$(频率)$_i$ x (时间)$_i$

如果，正如我所相信的，任务的概念组件现在占据了大部分时间，那么在仅仅是概念表达的任务组件上进行任何活动都不会带来大幅提高生产力。

因此，我们必须考虑那些解决软件问题本质的攻击，即这些复杂概念结构的制定。 幸运的是，其中一些非常有希望。

购买还是构建。 构建软件的最激进可能解决方案是根本不构建。

随着越来越多的供应商为各种应用程序提供更多更好的软件产品，每天都变得更容易。 虽然我们软件工程师一直在努力生产方法论，但个人电脑革命已经为软件创造了不止一个，而是许多大众市场。 每个报摊都有按机器类型分类的月刊，广告和评论数十种产品，价格从几美元到几百美元不等。 更专业的来源为工作站和其他Unix市场提供非常强大的产品。 甚至软件工具和环境也可以直接购买。 我曾在其他地方提出过一个个体模块的市场。

任何这样的产品购买起来比重新构建要便宜。 即使以10万美元的成本，购买的软件成品的成本也仅相当于一个程序员一年的工资。
而且交付是立即的！至少对于那些真实存在的产品来说是立即的，开发者可以向潜在用户推荐一个满意的用户。 此外，这些产品往往比自制软件有更好的文档记录，并且在一定程度上更易于维护。
我相信，大众市场的发展是软件工程中最深远的长期趋势。 软件的成本一直是开发成本，而不是复制成本。 即使在少数用户中分享这些成本，也会大幅降低每个用户的成本。 另一种看待这个问题的方式是，使用软件系统的n个副本实际上将开发人员的生产力乘以n。这是学科和国家生产力的提升。

当然，关键问题是适用性。 我能使用现成的软件包来完成我的任务吗？ 这里发生了一件令人惊讶的事情。 在1950年代和1960年代，一项又一项研究表明，用户不会使用现成的软件包来处理工资单、库存控制、应收账款等。要求太过专业化，案例间的变化太大。 在1980年代，我们发现这些软件包需求量大，使用广泛。发生了什么变化？

并不是真正的软件包。 它们可能比以前更通用，更可定制，但并不多。 也不完全是应用程序。如果

有什么变化，那就是今天的商业和科学需求比20年前更加多样化，更加复杂。

硬件/软件成本比发生了巨大变化。 1960年购买200万美元机器的买家觉得他可以再支付25万美元定制的工资单程序，这样的程序可以轻松地、不会造成中断地融入计算机不友好的社会环境。 今天购买5万美元办公设备的买家不可能负担得起定制的工资单程序；因此他们会调整工资单程序以适应现有的软件包。 计算机现在如此普遍，尽管还不那么受人喜爱，这些调整被视为理所当然。

有一些引人注目的例外情况，即软件包的泛化在多年来几乎没有改变：电子表格和简单的数据库系统。 这些强大的工具，事后看来显而易见，但出现得很晚，适用于各种用途，有些用途相当非正统。 现在有大量文章甚至书籍介绍如何使用电子表格处理意外任务。 大量以前可能用Cobol或报表程序生成器编写的应用程序现在通常使用这些工具完成。

许多用户现在整天在各种应用程序上操作自己的计算机，而从未编写过程序。事实上，许多这些用户不能为他们的机器编写新程序，但他们仍然擅长用它们解决新问题。

我相信当今对于许多组织来说，最强大的软件生产力策略是为那些对计算机一窍不通的智力工作者配备个人计算机和优秀的通用写作、绘图、文件和电子表格程序，然后让他们自由发挥。 同样的策略，配备简单的编程能力，也适用于数百名实验室科学家。

需求细化和快速原型制作。 构建软件系统中最困难的部分是准确决定要构建什么。.概念工作中没有其他部分像确立详细的技术需求那样困难，包括与人员、机器和其他软件系统的所有接口。 如果做错了，没有其他工作部分会像这样严重损害最终的系统。 没有其他部分比以后更难纠正。

因此，软件构建者为客户做的最重要的功能是对产品需求进行迭代提取和细化。 事实上，客户并不知道他们想要什么。 他们通常不知道必须回答哪些问题，几乎从未考虑过必须明确规定的问题细节。 即使简单的答案–“使新软件系统像我们的旧手动信息处理系统一样工作”–实际上也太简单了。 客户从来不想要完全那样。 复杂的软件系统是行动、移动、工作的东西。 这种行动的动态很难想象。 因此，在规划任何软件活动时，必须允许客户和设计师之间进行广泛的迭代，作为系统定义的一部分。

我想进一步提出，即使是与软件工程师合作的客户，也很难在构建和尝试产品的某些版本之前，完全、准确地指定现代软件产品的确切要求。

因此，当前技术努力中最有前途的之一，也是攻击软件问题本质而非意外的努力之一，是开发系统快速原型设计方法和工具，作为需求迭代规范的一部分。

原型软件系统是模拟重要接口并执行预期系统主要功能的系统，虽然不一定受相同硬件速度、大小或成本约束。 原型通常执行应用程序的主要任务，但不尝试处理异常情况，正确响应无效输入，干净地中止等。原型的目的是使指定的概念结构变为现实，以便客户可以测试其一致性和可用性。

当今许多软件获取程序都建立在这样一个假设之上，即可以事先指定一个令人满意的系统，为其建造招标，建造并安装。我认为这一假设基本上是错误的，许多软件获取问题源于这种谬误。 因此，不能在没有根本修订的情况下解决这些问题，根本修订应提供原型和产品的迭代开发和规范。

增量开发 - 成长，而不是构建，软件。 我仍然记得1958年我第一次听到朋友谈论构建程序而不是编写程序时所感受到的震撼。一下子，他拓宽了我对软件过程的整体看法。. 这种隐喻转变是强大而准确的。 今天，我们明白软件构建与其他建筑过程相似，我们自由地使用隐喻的其他元素，比如规格说明、组件的组装和脚手架。

建筑隐喻已经过时了。 是时候再次改变了。 如果，正如我所相信的，我们今天构建的概念结构太复杂，无法事先准确指定，而且太复杂，无法无误地构建，那么我们必须采取一种根本不同的方法。

让我们转向自然，研究生物体中的复杂性，而不仅仅是人类的作品。在这里，我们发现构造物的复杂性让我们感到敬畏。单单大脑就是一个超越映射的错综复杂结构，超越模仿的强大实体，丰富多样，自我保护和自我更新。 秘密在于它是生长的，而不是建造的。

因此，我们的软件系统也必须如此。 几年前，哈兰·米尔斯提出，任何软件系统都应通过增量开发来发展。[11] 也就是说，系统应该首先运行起来，即使它除了调用一组虚拟子程序之外没有任何有用的功能。 然后，逐步完善，逐步将子程序开发为下一级别中的操作或调用空存根。

自从我开始在我的软件工程实验室课程中向项目建设者推荐这种技术以来，我看到了最引人注目的结果。 在过去的十年中，没有任何事物如此根本地改变了我的实践，或者它的有效性。 这种方法需要自顶向下的设计，因为它是软件的自顶向下的发展。 它允许轻松地回溯。 它适合早期原型。 每个添加的功能和为更复杂的数据或情况提供新的规定都是有机地从已有的内容中发展出来的。

---

[11] 米尔斯，H. D.，“大型系统中的自顶向下编程”，大型系统调试技术，R. R*u*stin主编，新泽西州恩格尔伍德克利夫斯，Prentic*e*-Hall，1971年。

士气效果令人震惊。 当有一个运行系统时，甚至是一个简单的系统，热情会上升。当新的图形软件系统在屏幕上出现第一幅图片时，即使只是一个矩形，努力会加倍。 在整个过程的每个阶段，总是有一个可运行的系统。 我发现团队在四个月内可以培养出比他们能够建造的更复杂的实体。

在大型项目上可以实现与我的小项目一样的好处。[12]

优秀的设计师。 如何改进软件艺术的核心问题始终是关于人的。

通过遵循良好的实践而不是糟糕的实践，我们可以得到良好的设计。 良好的设计实践是可以教授的。 程序员是人口中最聪明的一部分，因此他们可以学习良好的实践。 因此，美国的一个主要推动力是推广良好的现代实践。 为了将我们的实践水平从糟糕提升到良好，新的课程、新的文献、新的组织如软件工程研究所等都已经出现。这是完全正确的。

然而，我不相信我们可以以同样的方式向上迈出下一步。虽然贫乏的概念设计和良好设计之间的差异可能在于设计方法的合理性，但良好设计和优秀设计之间的差异显然不在于此。优秀的设计来自优秀的设计师。 软件构建是一个创造性的过程。 良好的方法论可以赋予和解放创造性思维；它不能点燃或激发苦工。

差异并不是微小的–这更像是萨列里和莫扎特。研究表明，最优秀的设计师制作的结构更快、更小、更简单、更干净，并且付出的努力更少。 伟大和普通之间的差异接近一个数量级。

稍作反思就会发现，虽然许多优秀、有用的软件系统是由委员会设计并由多方项目构建的，但那些激发热情粉丝的软件系统是由一个或几个设计思维、伟大的设计师的产品。 考虑Unix、APL、Pascal、Modula、Smalltalk界面，甚至Fortran；与Cobol、PL/I、Algol、MVS/370和MS-DOS进行对比（图1）

是不	
Unix	Cobol
APL	PL/1
Pascal	Algol
Modula	MVS/370
Smalltalk	MS-DOS
Fortran	

图**1**令人兴奋的产品

因此，尽管我强烈支持目前正在进行的技术转移和课程开发工作，但我认为我们可以展开的最重要的单一努力是开发培养优秀设计师的方法。

---

[12]Boehm, B. W., “软件开发和增强的螺旋模型”，计算机，**20**，5（1985年5月），第43-57页。

没有软件组织可以忽视这一挑战。 尽管优秀的经理人虽然稀缺，但优秀的设计师并不比他们更稀缺。 优秀的设计师和优秀的经理人都非常罕见。 大多数组织在寻找和培养管理潜力方面花费了大量精力；我不知道有哪个组织花同样的精力来寻找和培养最终决定产品技术卓越性的优秀设计师。

我的第一个建议是，每个软件组织都必须确定并宣布，优秀的设计师对其成功同样重要，就像优秀的经理一样重要，并且可以期望他们得到类似的培养和奖励。 不仅工资，还有认可的各种福利–办公室大小、家具、个人技术设备、差旅基金、员工支持–必须完全相等。

如何培养优秀的设计师？ 空间不允许进行长篇讨论，但一些步骤是显而易见的：

- 系统地尽早确定顶尖设计师。 最好的通常不是最有经验的。
- 指定一位职业导师负责发展潜力，并保持一份仔细的职业档案。
- 为每个潜力制定并维护职业发展计划，包括与顶尖设计师的精心选择的学徒期、高级正规教育的片段以及短期课程，所有这些都与独立设计和技术领导任务交替进行。
- 为成长中的设计师提供与彼此互动和激励的机会。

■