软件开发的 201 个原则

（中译本）

201 Principles of Software Development

Alan M. Davis

University of Colorado

at Colorado Springs

百度《201 个原则》翻译小组

仅供内部学习使用

目录

译者序

其实我不是译者，而仅仅是一名“校对者”。大家让我来写这篇译者序，盛情难却，无法推脱。

《软件开发的 201 个原则》是我于 2017 年至 2020 年在百度举办“代码的艺术训练营”时使用的指定教材。这本书的内容深受训练营学员的好评。由于之前没有中文版，对于部分英文基础不太好的同学来说阅读有些困难。终于在 2019 年底，有十多名“代码的艺术训练营”的毕业生自发组织起来，开始了此书的翻译。我从 2020 年 5 月初加入校对工作，完成全部的校对，我大约花费了 80-100 个小时。由此推断，负责翻译的同学花费了数倍于此的时间。非常感谢这些同学的无私付出！

初识《201 个原则》是在 20 年前。当时我还在清华大学读书，在老师的指导下做一个有一定规模的软件研发项目。在项目的研发过程中，遇到了不少软件工程方面的问题。于是在那一年，我阅读了大约 10 本软件工程方面的书籍，包括《Code Complete》（代码大全）、《Rapid Development》（快速开发）、《Programming Pearls》（编程珠玑），等等。《201 个原则》是我当时在清华图书馆中发现的一个“宝贝”。我必须说，这本书对我的影响非常深，很多我现在经常提起的软件工程原则，其实都源于对这本书的阅读。

2006 年我离开清华，到目前已经在工业界工作十多年，经历了多家公司。我发现，虽然我们的软件研发规模已经和 20 年前有了很大的发展，但是在软件研发的理念方面的进步还是太慢了。有太多的软件从业者，即使已经工作多年，但对于软件研发的基本理念和原则还是了解不多。以我多次的调查，阅读超过 2 本“真正的”软件工程书籍的人是非常少数的。很多

软件工程师，仍然在使用非常低效的、甚至是错误的方法在工作！

于是在 2015 年，我在百度开办了“代码的艺术”面授课程，其中就重点推荐了《201 个原则》。而在 2017 年做“代码的艺术训练营”的时候，这本书就成了指定教材。为什么要选择这本书？因为它对软件工程的内容覆盖全面，且篇幅短小。对于一个短期培训班来说，如果选择类似《Code Complete》这样的书籍，阅读所需要的时间有些太多了。在这个场合，《201 个原则》是一个性价比更高的选择。另外，我常常感觉，对于一个软件工程师，掌握正确的意识是比掌握具体知识更重要的。如果有正确的意识，即使不记得具体的知识点，还可以在需要的时候进行查阅。而反过来就不是这样了。

必须要说，《201 个原则》写于 1995 年，距今已经有 25 年时间。这也成为很多人担心的来源 — 计算机技术的发展如此之快，这本书是不是已经过时了？但是，正如我在“代码的艺术”课程中所述的“知识、方法、精神”三者的对比，方法的变化速度远远慢于知识。尤其是在本次校对过程中，我惊奇的发现，本书中真的可以说是“过时”的原则还不到 5 个！是软件研发的方法变化太慢，还是本书的内容太深刻？我想两者兼而有之。在此，我必须要对本书的原作者 Alan M. Davis 致敬，并对《201 个原则》中所有原则的贡献者和历史上所有软件工程领域的大师们致敬！

最后，要隆重的介绍本次负责翻译的百度同学。他们是：叶王，马学翔，吴斌，曾浩浩，甘璐，李殿斌，王冰清，杨光，李子昂，肖远昊。另外，经过大家的商定，本书翻译出版的所有稿酬，都将捐赠给公益事业。

对所有的读者，所有的阅读此书的软件工程师，所有准备从事软件研发的同学们，希望本书能够对你们有所帮助！

章淼 博士

百度 BFE 团队技术负责人，百度代码规范委员会主席

2020 年 6 月 14 日写于百度

前言

如果软件工程真的是一门工程学科，那么它是对经过验证的原则、技术、语言和工具的智慧的运用，用于有成本效益的创造和维护能够满足用户需求的软件。本书是有史以来第一本成册的软件工程原则集*。*原则*是一种关于软件工程的基本真理、规则或假设，不管所选的技术、工具或语言是什么都仍然有效。除了少数例外，在这里发表的原则都不是原创的。有许多是从很多软件工程从业者和研究者的著作中摘录的。这些人已经足够无私的和我们分享他们的经验、想法和智慧。我并不认为这 201 条原则是相互独立的。不像 Boehm 的 7 条"基本"软件工程原则，其中一些原则的组合可能蕴涵了另一个原则。我也不认为这 201 条原则是 100%兼容的。俗话说，"别离使心更近"和"眼不见，心不烦"都是真理，每个都可以应用在我们的生活中，但是它们却不能同时用来证明同一个决定是正确的。本书中包含的原则都是有效的，它们都能够被用来提升软件工程，但也许并不能将某些组合应用到任何一个项目中。

Manny Lehman [LEH80] 已经雄辩的说明，为什么软件工程的基本原则和人类探索的其它领域的基本原则存在根本性的不同。他说，没有理由期待，软件工程的原则具备和（例如）物理学原理一样的精确性和可预测性。原因是，不像物理学或生物学，软件开发的过程是由人来管理和实现的；这样，从长远看软件开发的行为是不可预测的，依赖于人的判断、奇想和行动。另一方面，软件似乎展现出很多有规律和可预测的特征 [LEH80]。这使得很多基本原理可以被列出，并可以被没经验的、或有经验的软件工程师和管理者使用，以增强软件工程过程和软件产品的质量。

本书的目的是在一本书中展现软件工程的原则，以作为参考指南。本

书针对以下 3 类读者：

1. *软件工程师和管理者*。在本书中，你可以弄清什么是好的，什么是不好的。如果在软件工程或软件管理方面你是个新人，本书是明确你需要了解哪些东西的一个好地方。
2. *软件工程方向的学生*。对学生来说，本书有两个主要用途。首先，这里有基本的、而非教条的原则，这些原则是每个软件工程师都应该知道的。其次，本书各页的参考文献指向在软件工程方面最好的一些论文和书籍。即使你只是阅读了参考文献的内容，本书也是非常成功的，你将接触到非常丰富的知识。
3. *软件研究人员*。研究人员也许经常发现找寻一个想法的最初来源是困难的。我已经提供了参考文献，以反映最初的来源，或引用了最初来源的替代且出色的工作。

我真诚的希望，每个购买本书的读者，能够努力去阅读尽可能多的参考资料。我关于原则的简要描述，希望做到友好、易于阅读和见解深刻。但是要真正理解，你需要去阅读参考文献。这些参考文献，并不一定是这些想法的最初来源（虽然很多时候它们确实是最初来源）。这里给出的原则，也不一定是这些参考文献的重点内容。然而，在每个案例中，参考文献都包含了和原则有关的大量有帮助的背景、洞察、理由、备查数据或信息。

总之，本书应该成为你查阅任何软件工程想法的第一个地方。然而，这是一本关于原则的书，而不是关于技术、语言或工具的。在这里，你无法找到如何使用本书原则中提到的技术、语言、工具。而且，本书尽力避免各种流行趋势，不管是好的还是坏的！在大多数情况下，流行趋势会持续 3-10 年，然后就失宠了。在本书中能够看到可能在某个流行趋势背后的基本原理，而不是流行趋势本身。例如，你在这里看不到面向对象自身的

参考文献，但是你可以看到面向对象之下的基本原则，如封装。

所有的原则被组织为大的类别，以便于查找，也便于发现类似的原则。这些类别对应于软件开发的主要阶段（即，需求分析，设计，等等），也对应于其它关键的“支持”活动，如管理、产品保证等等。如图 P-1 所示。

Alan M. Davis

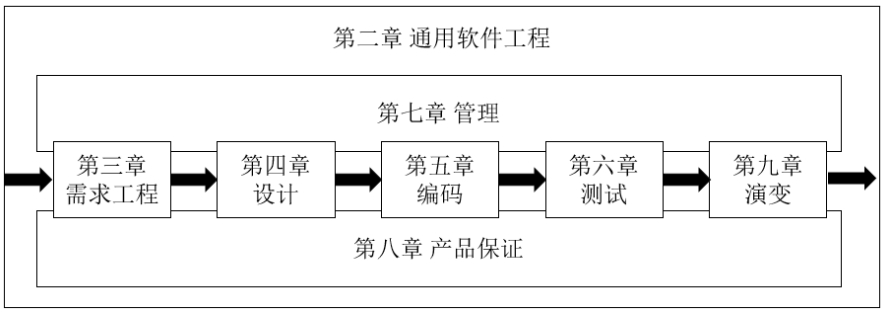


图 P-1 本书的组织

* Winston Royce 和 Barry Boehm 发表了软件工程原则方面最早的两篇论文，分别包含 5 个原则和 7 个原则。

参考文献

[BOE83] Boehm, B., "Seven Basic Principles of Software Engineering," Journal of Systems and Software, 3, 1 (March 1983), pp. 3-24.

[LEH80] Lehman, M., "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle," Journal of Systems and Software, 1, 3 (July 1980), pp. 213-221.

[ROY70] Royce, W., "Managing the Development of Large Software Systems," WESCON '70, 1970; reprinted in 9th International Conference on Software Engineering, Washington, D.C.: IEEE Computer Society Press, 1987, pp.328-338.

致谢

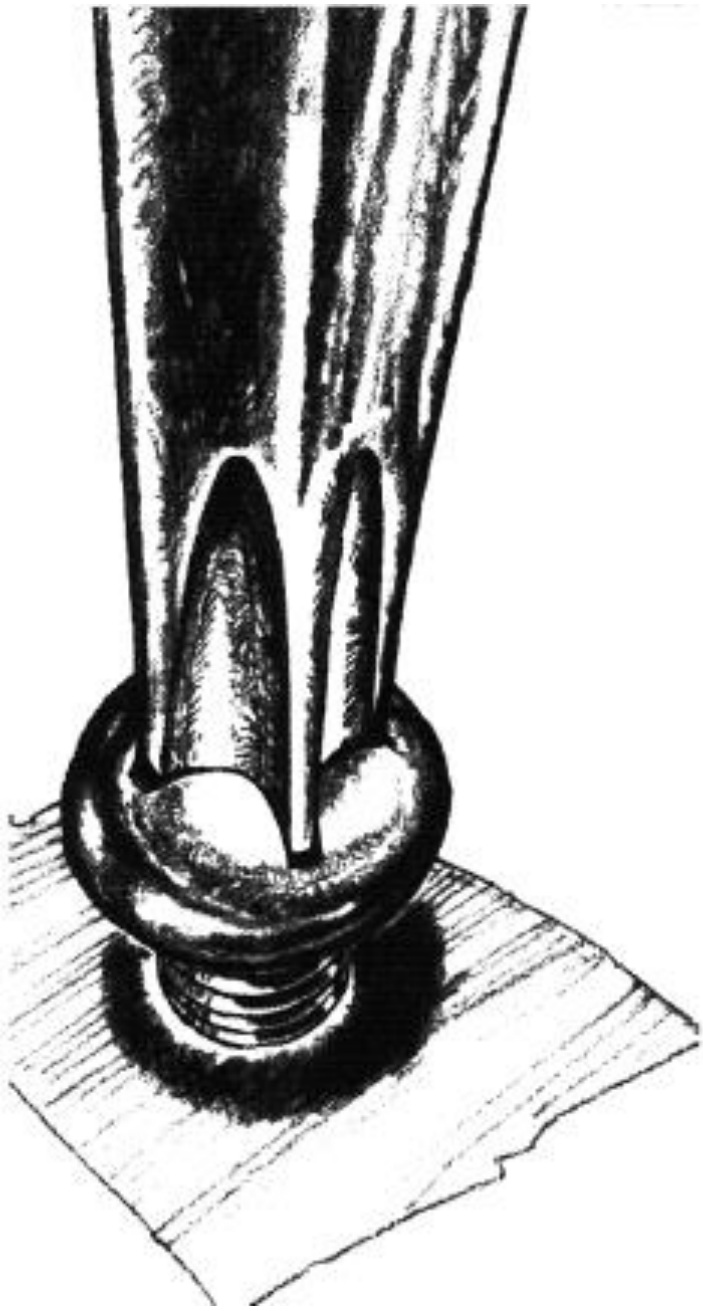
ACKNOWLEDGMENTS

在我们一起教一门课的时候，Drexel 大学的 Stephen Andriole 在不知不觉中启发了我来写这本书。我刚才提到，软件工程和其它工程学科一样是由一组底层原则来驱动的。我的说法似乎很合乎逻辑。然而，Steve 挑战我：”AI，说出一个。就说出一个！”。很幸运我脑子很快，想出了一个。他说，”好，只要再说一个，我就会相信确实存在软件工程的原则”。我想出了一个又一个。

Kerry Baught 负责建立和维护手稿的质量。Steve andriole，Manny Lehman 和 Jawed Siddiqi 评审了手稿的早期版本。Siddiqi 博士在如何组织这些原则方面提供了重要的建议。

最后，但同样重要的，我想感谢我的妻子 Ginny，我们的孩子 Marsha 和 Michael，我的父母 Barney 和 Hannah Davis。感谢他们所有的爱、支持和心甘情愿，在我投入写作的时候，生活在缺少丈夫、父亲和儿子的情况下。

软件开发的 201 个原则



第一章 引言

INTRODUCTION

本书包含一系列软件工程的原则。这些原则代表了我们所认为的软件开发过程中的最新理念。其它工程领域都有基于物理学、生物学、化学或数学定律的原则。然而由于软件工程的产物是非实体的（nonphysical），所以实体的定律（laws of the physical）并不能轻易的成为坚实的基础。

软件行业已经有大量讨论技术、语言和工具的书籍，但没有书籍试图去编制基本原则的列表。如图 1-1 所示，*原则*（Principle）是工作的准则；原则代表了许多人从经验中总结出来的集体智慧。它们往往被描述为绝对真理（这总是正确的）或用作推论（当 X 发生时，Y 将会发生）。

技术（Technique）是一种按部就班的流程，它帮助软件开发者执行一部分软件工程过程。技术倾向于强制遵循基本原则的一个子集。大部分技术会创建文档和（或）程序。许多技术也会分析现有的文档和（或）程序，或将其转变为产品。

语言（Language）由一组基本元素（如单词或图形符号）、规则和语义组成。规则可以让人们用基本元素构造出更复杂的实体（如句子、图表、模型），语义则赋予每个实体组合以意义。语言用于表达所有软件工程的产出，无论是中间的还是最后的。那些通过技术创建或分析的文档和程序通常也会用某种语言来表达。

工具（Tool）是软件程序，可帮助软件工程师执行软件工程某些步骤。它们可以：

- 作为工程师的顾问（例如，基于知识的需求助理）；

- 分析某些东西是否符合某种技术（例如，数据流图检查器）或原则的子集；
- 使软件工程中的一些方面自动化（例如，编译器）；
- 辅助工程师完成一些工作（例如，编辑器）。

一个学科的原则集合，会随着学科的发展而发展。现存的原则会发生改变，新的原则会加进来，旧的原则将不再适用。实践和从实践中获得的经验，促使我们发展了那些原则。在如今，当我们去审视一些来自于 1964 年的软件工程原则时，会觉得它们看起来很傻（例如，总是使用简短的变量名，或者尽可能让程序体积更小）。三十年后，如今的一些原则也会如此。

现在，请看*现代*的软件工程原则。

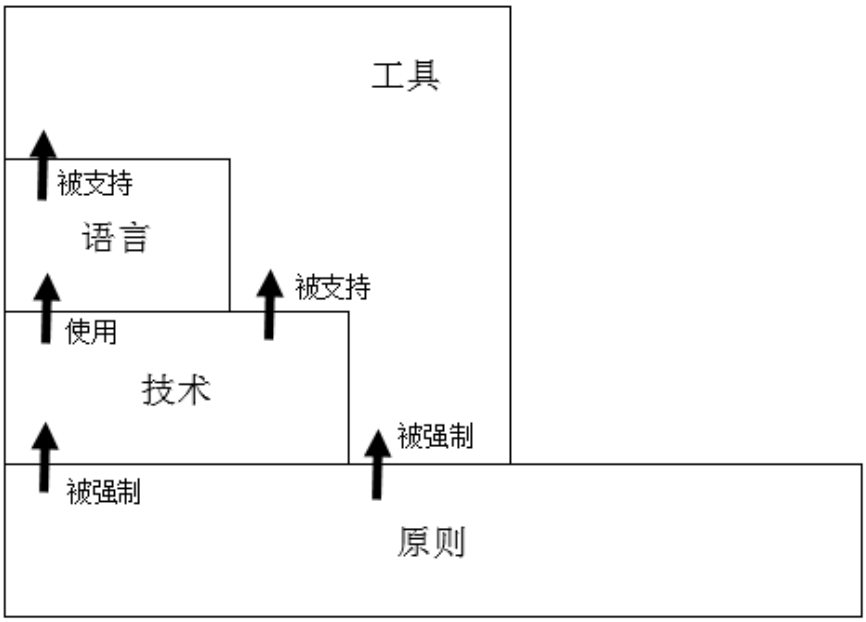
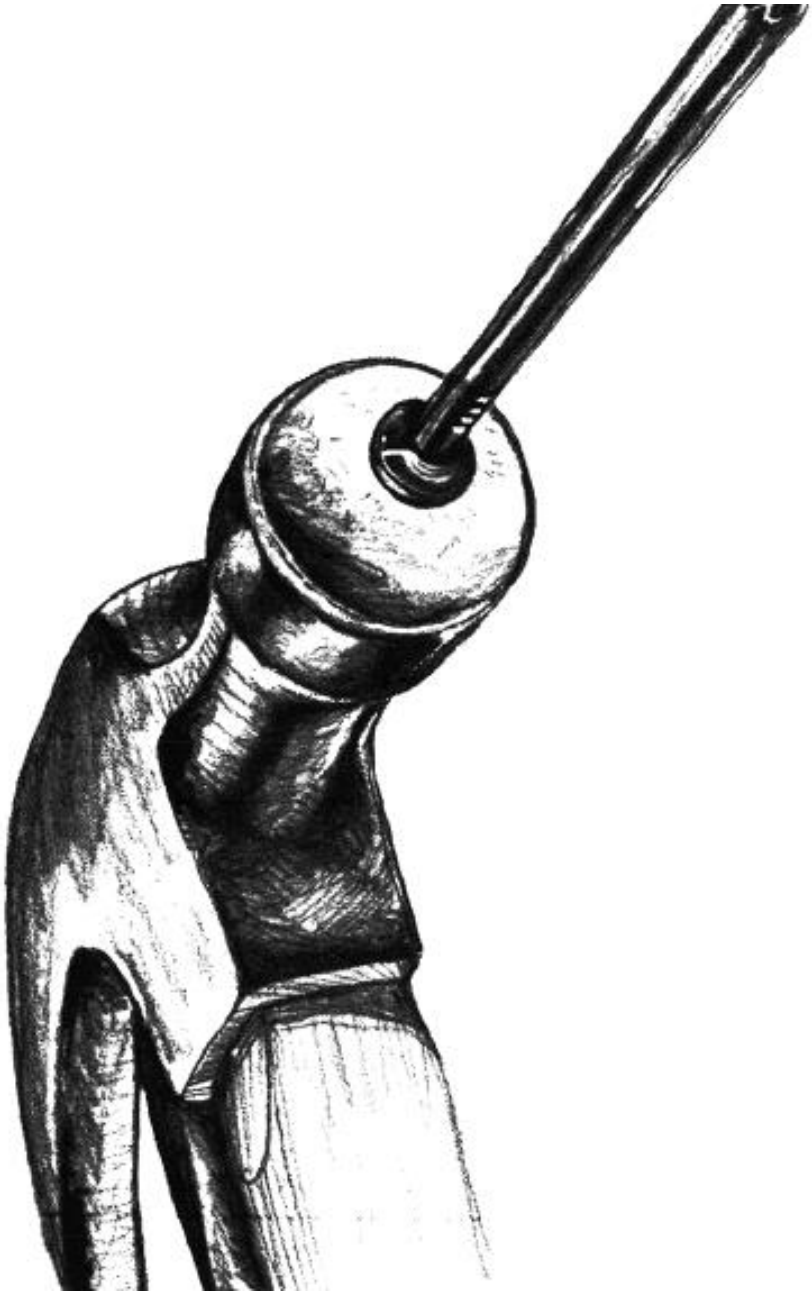


图 1-1 原则，技术，语言，工具

译者注

虽然作者说，“今天的原则在三十年后会看起来同样的荒谬”。但是非常令人吃

惊的是，在原书出版 25 年后，我们看到其中超过 95%的原则都没有过时！



第二章 一般原则

GENERAL PRINCIPLES

原则 1 质量第一

QUALITY IS #1

无论如何定义质量，客户都不会容忍低质量的产品。质量必须量化，并建立可实施落地的机制，以促进和激励质量目标的达成。即使质量差、也按时交付产品，这似乎是政治正确的行为，但这是短视的。从中长期来看，这样做是自杀。质量必须放在首位，没有可权衡的余地。Edward Yourdon 建议，当你被要求加快测试、忽视剩余的少量 bug、在设计或需求达成一致前就开始编码时，直接说"不"。

参考文献

YourDon, E., Decline and Fall of the American Programmer, Englewood Cliffs, N.J.: Prentice Hall, 1992 (Chapter 8).

原则 2 质量在每个人眼中不同

QUALITY IS IN THE EYES OF THE BEHOLDER

软件质量没有唯一的定义。对开发者来说，质量可能是优雅的设计或优雅的代码。对在紧张环境中工作的用户来说，质量可能是响应时间或大容量。对成本敏感的项目来说，质量可能是低开发成本。对一些客户来说，质量可能是满足他们所有已知和未知的需求。这里的难题是，以上要求可能无法完全兼顾。当优化某人关注的质量时，可能会危害其他人关注的质量（这就是温伯格的“政治困境”原则）。项目必须确定各因素的优先级，并清晰地传达给所有相关方。

参考文献

Weinberg, G., Quality Software Management, Vol. 1: Systems Thinking, New York: Dorset House, 1992, Section 1.2.

原则 3 开发效率和质量密不可分

PRODUCTIVITY AND QUALITY ARE INSEPARABLE

开发效率与质量之间存在明显的关系（开发效率可以用每人月完成的代码行数或功能点数来度量）。对质量要求越高，开发效率就越低。对质量要求越低，开发效率就越高。越是强调提高开发效率，最终的质量就越低。贝尔实验室发现，在要求每千行代码有 1-2 个 bug 时，每人月的效率通常为 150-300 行代码 [参见 Fleckenstein, W. "Challenges in Software Development", IEEE Computer, 16, 3 (March 1983), pp. 60-64]。当试图提高开发效率时，bug 的密度就会增加。

参考文献

Lehman, M., "Programming Productivity - A life Cycle Concept," COMPCON 81, Washington, D.C.: IEEE Computer Society Press, 1981, Session 1.1.

原则 4 高质量软件是可以实现的

HIGH-QUALITY SOFTWARE IS POSSIBLE

尽管我们的行业充斥着表现不佳、充满 bug、或者根本无法满足用户需求的软件系统的例子，但仍然有一些成功的例子。大型软件系统可以以非常高的质量构建，但价格昂贵：每行代码高达 1000 美元。例如：IBM 为美国宇航局的航天飞机而开发的机载飞行软件。总共约 300 万行代码，源于严谨的软件开发过程，产品发布后每万行代码发现的错误少于一个。

作为软件开发人员，应该学习和了解已被验证、可以极大提高软件质量的方法。这些方法包括：让客户参与（原则 8）、原型设计（在全面开发之前验证需求；原则 11 至 13）、保持设计简单（原则 67）、代码评审（原则 98）和雇用最优秀的人（原则 130 和 131）。作为客户，追求卓越的同时，要意识到随之而来的高额成本。

参考文献

Joyce, E., "Is Error-Free Software Achievable?" Datamation (February 15, 1989).

原则 5 不要试图改进质量

DON’T TRY TO RETROFIT QUALITY

质量无法通过软件的改进来获得。这适用于质量的任何定义：可维护性、可靠性、适应性、可测试性、安全性等等。即使我们在开发过程中努力，使软件具备高质量也是十分不易的。如果我们不努力，又怎么可能期望获得高质量呢？这就是绝不能将“一次性原型”转换成产品的主要原因（原则 11）。

参考文献

Floyd C., "A Systematic Look at Prototyping," in Approaches to Prototyping, R. Budde, et al., Berlin, Germany: Springer Verlag, 1983, pp. 1-18, Session 3.1.

原则 6 低可靠性比低效率更糟糕

POOR RELIABILITY IS WORSE THAN POOR EFFICIENCY

如果软件执行效率不高，通常可以分离消耗大部分执行时间的程序单元，重新设计或编码以提高效率（原则 194）。低可靠性问题不仅难以发现，也更难以修复。系统的低可靠性问题可能会在系统上线多年后才暴露出来 — 甚至可造成人员伤害。一旦低可靠性问题显现，通常难以隔离其影响。

参考文献

Sommerville, I., Software Eginnering, Reading, Mass.: Addision Wesley, 1992, Session 20.0.

原则 7 尽早把产品交给客户

GIVE PRODUCTS TO CUSTOMERS EARLY

在需求阶段，无论你多么努力地试图去了解用户的需求，其实确定他们真实需求的最有效方法就是给他们一个产品，让他们使用它。如果遵循传统的瀑布模型，那么在 99%的开发资源已经耗尽之后，才会第一次向客户交付产品。如此一来，大部分的客户需求反馈将发生在资源耗尽之后。

和以上方法相反，可在开发过程的早期构建一个快速而粗糙的原型。将这个原型交付给客户，收集反馈，然后编写需求规格说明、并进行正规的开发。使用这种方法，当客户体验到第一个产品版本时，只消耗了 5-20%的开发资源。如果原型包含合适的功能，就可以更好的理解和把握最有风险的用户需求，最终产品也就更有可能让用户满意。这有助于确保将剩余的资源用于开发正确的系统。

参考文献

Gomaa, H., and D. Scott, "Prototyping as a Tool in the Specification of User Requirements," Fifth International Conference on Software Engineering, Washington, D.C.: IEEE Computer Society Press, 1981, pp. 333-342.

原则 8 与客户/用户沟通

COMMUNICATE WITH CUSTOMERS/USERS

永远不要忽视软件开发的原因：满足真正的需求，解决真正的问题。解决真正需求的唯一方法，是去跟有真正需求的人沟通。客户或用户是你的项目的最重要参与者。

如果你是一个商业开发人员，经常和客户交谈。让他们参与进来。当然，闭门造车式的开发更容易，但是客户会喜欢这样的结果吗？如果你是软件外包的生产商，在开发过程中很难找到“客户”，那就进行角色扮演。在你的组织中指定 3-4 个人作为潜在的客户，征求他们的意见：何以能让他们持续成为客户，并使他们满意。如果你是政府项目的承包商，要经常与签约官员、技术代表以及（如果可能的话）产品的用户交谈。政府里的人和事经常会变化，跟上变化的唯一方法就是沟通。忽视上述变化可能在短期内会让生活看起来更容易，但最终的系统将无法使用。

参考文献

Farbman, D., "Myths That Miss," Datamation (November 1980), pp. 109-112.

原则 9 激励开发者与客户对齐

ALIGN INCENTIVES FOR DEVELOPER AND CUSTOMER

项目经常会因为客户和开发人员有不同（或不兼容的）目标而失败。一个简单的案例是，客户希望在特定日期前获得特性 1、2、3，而开发人员希望最大化营收或利润。为了最大化营收，开发人员可能会尝试完整地开发这三个特性，即使会导致延期。

为对齐双方的目标，有如下方法：

（1） 按优先级对需求排序（原则 50），以便开发人员了解它们的相对重要性

（2） 根据需求的优先级奖励开发人员（例如：所有高优先级的需求必须完成；每完成一个中优先级的需求，开发人员可获得一些额外的小奖励；每完成一个低优先级的需求，可获得的奖励非常小）

（3） 对逾期交付实行严厉的处罚

原则 10 做好抛弃的准备

PLAN TO THROW ONE AWAY

对一个项目来说，最关键的成功因素之一是，它是否是全新的。在全新领域(可能涉及：应用程序、体系结构、接口、算法等)研发的程序很少第一次就成功。弗雷德·布鲁克斯（Fred Brooks）在《人月神话》中明确建议："无论如何，你一定要做好抛弃的准备"。 这个建议最初由温斯顿·罗伊斯（Winston Royce）在 1970 年提出，他说一个人应该做好准备：第一个完整部署的系统，往往是第二个被创建的系统。第一个系统至少可用于验证关键的设计问题和操作概念。此外，罗伊斯建议，应该使用大约 25%的资源开发这样的预发布版本。

作为一个全新定制产品的开发人员，在开始全面的开发之前，要规划开发一系列"一次性原型"(原则 11、12 和 13)。作为商用大规模系统的开发人员，可以预期，第一个产品版本在一定年限内将能够被修改，之后它将被完全替换（相关原则 185，186，188 和 201）。作为产品的维护者，请注意，在程序变得不稳定以至于必须被替换之前，你对程序可以调整的地方还有很多 (请参阅相关原则 186，191，195 和 197)。

参考文献

Royce, W., "Managing the Development of Large Software System," WESCON' 70, 1970; reprinted in 9th International Conference on Software Engineering, Washington D.C.: IEEE Conputer Society Press, 1987, pp. 328-338.

原则 11 开发正确的原型

BUILD THE RIGHT KIND OF PROTOTYPE

有两种原型：一次性（throwaway）原型和演进式（evolutionary）原型。*一次性*原型用快速而粗糙的方式构建，交给客户以获得反馈，在得到期待的信息后即被废弃。获得的信息被整理进需求规格说明，用于正规的产品开发。*演进式*原型用高质量的方式构建，交给客户以获得反馈，获得期待的信息便进行修改，以更加贴近用户的需求。重复此过程，直到产品收敛到所期望的样子。

一次性原型应该在关键需求特性没有很好理解时使用。演进式原型应该在关键特性已被充分理解，但很多其他需求特性没被充分理解时使用。如果对大多数功能都不了解，则首先构建一个一次性原型，然后从零开始构建一个演进式原型。

参考文献

Davis, A., "Operational Prototyping: A New Development Approach," IEEE Software, 9, 5 (September 1992), PP. 70-78

原则 12 构建合适功能的原型

BUILD THE RIGHT FEATURES INTO A PROTOTYPE

当建立一次性原型时，只需要开发那些没有被充分理解的特性。如果你开发已充分理解的特性，最终除了浪费资源外，将一无所获。当建立演进式原型（原则 13）时，要优先开发那些已经被充分理解的特性。（注意，它们可能已经被充分理解，因为之前已使用一次性原型进行验证）你的希望是，通过体验这些特性，用户能更好地确定其它需求。如果你基于模糊的需求(高质量的)开发了一个演进式原型，一旦需求搞错了，你将不得不抛弃这个高质量的软件，并且浪费了资源。

参考文献

Davis, A., "Operational Prototyping: A New Development Approach," IEEE Software, 9, 5 (September 1992), PP. 70-78.

原则 13 要快速的开发一次性原型

BUILD THROWAWAY PROTOTYPES QUICKLY

如果你已经决定开发一次性原型，就要用最快的方式。不用担心质量。可使用“一页纸”的需求规格说明。不用担心设计或编码中的文档。可以使用任何工具。可以使用任何编程语言，只要能够便利程序的快速开发。不用担心编程语言的可维护性。

参考文献

Andriole, S., Rapid Application Prototyping, Wellesley, Mass.:QED, 1992.

原则 14 渐进地扩展系统

GROW SYSTEMS INCREMENTALLY

渐进地扩展系统，是降低软件开发风险的最有效方法之一。从一个小的可用系统开始，只实现少数功能。然后逐步扩展，覆盖越来越多的最终功能子集。

这样做的好处是：（1）降低每次开发的风险；（2）看到一个产品版本，通常可以帮助用户想象出他们想要的其他功能。

这样做的缺点是：如果过早选择了一个不合适的系统架构，则可能需要全面的重新设计、才能适应后续的变更。在开始增量开发之前，开发一次性原型（原则 11，12 和 13），可以降低这种风险。

参考文献

Mills, H., "Top-Down Programming in Large Systems," in Debugging Techniques in Large Systems, R. Ruskin, ed., Englewood Cliffs, N.J.: Prentice Hall, 1971.

原则 15 看到越多，需要越多

THE MORE SEEN, THE MORE NEEDED

在软件行业，一次次见证了：提供给用户越多的功能（或性能），用户想要的功能（或性能）就越多。当然，这支持了原则 7（尽早把产品交给客户），原则 14（渐进的开发系统），原则 185（软件将会持续改变）以及原则 201（系统的存在促进演变）。但更重要的是，你必须为不可避免的情况做好准备。管理和工程流程的每个方面都应该意识到，一旦客户看到产品，他们就会想要更多。

这意味着，所产生的每个文档都应该以有利于更改的方式进行存储和组织。这意味着，配置管理流程（原则 174）必须在距离交付很长时间之前就位。这也意味着，在软件部署后不久，你就应该准备好，以应对用户口头或书面请求的冲击。这还意味着，你选择的设计方案应使容量、输入速率和功能都很容易变更。

参考文献

Curtis, B., H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," Communications of the ACM, 31, 11 (November 1988), pp. 1268-1287.

原则 16 开发过程中的变化是不可避免的

CHANGE DURING DEVELOPMENT IS INEVITABLE

爱德华·伯索夫（Edward Bersoff）等人将系统工程的第一定律定义为：“无论你在系统[开发]生命周期中的何处，系统都将发生变化，并且对其进行改变的愿望将在整个生命周期中持续存在”。 与原则 185 和 201（强调软件部署后，需求可能发生巨大变化）不同，本原则想表达，在开发过程中，软件也可能发生巨大变化。这些变化可能体现在编写新的代码、新的测试计划或新的需求规格说明。这些变化可能意味着，要去修复某个被发现是不正确的中间产品。可能它们反映了完善、或改进产品的自然过程。

为变化做好准备，要确保：软件开发涉及的所有产品之间的相互引用都是适当的（原则 43，62 和 107）；变更管理流程已就位（原则 174，178 至 183）；预算和进度表有足够的余地，不会为了满足预算和进度表而倾向于忽略必要的变化（原则 147，148 和 160）。

参考文献

Bersoff, E., V. Henderson, and S. Siegel, Software Configuration Management, Englewood Cliffs, N.J.: Prentice Hall, 1980, Section 2.2.

原则 17 只要可能，购买而非开发

IF POSSIBLE, BUY INSTEAD OF BUILD

要降低不断上涨的软件开发成本和风险，最有效的方法就是，购买现成的软件，而不是自己从头开发。确实，现成的软件也许只能解决 75％的问题。但考虑一下从头开发的选择吧：支付至少 10 倍于购买软件的费用，冒着超出预算 100％且延期的风险（如果最后能够完成！），并且最终发现，它只能满足 75%的预期。

对一个客户来说，新的软件开发项目似乎最初总是令人兴奋。开发团队也是“乐观的”，对“最终”解决方案充满了希望。但几乎很少有软件开发项目能够顺利运行。不断增加的成本通常会导致需求被缩减，最终研发出的软件，它可以满足的需求，也许跟现成的软件差不多。作为一个开发者，应该复用尽可能多的软件。复用是“购买而非开发”原则在较小范围内的体现。参考相关的原则 84。

参考文献

Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer, 20, 4 (April 1987), pp. 10-19.

原则 18 让软件只需简短的用户手册

BUILD SOFTWARE SO THAT IT NEEDS A SHORT USERS' MANUAL

衡量软件系统质量的一种方法是查看其用户手册的大小。手册越短，软件质量越好。设计良好的软件，用法应该不言而喻。不幸的是，太多的软件设计师也将自己塑造成人机界面设计专家。而大量的用户手册充分证明，大多数界面设计师并不像他们宣称的那样出色。（顺便说一句，当我说“用户手册”时，也包括在线帮助文档。因此，把用户手册发布到网上，软件并不会在一夜之间突然变得更好。）

应使用标准的界面。让行业专家设计浅显易懂的图标、命令、协议和用户场景。要记住：仅仅因为软件开发人员“喜欢”某种用户界面，并不意味着你的客户就会知道怎么使用它。许多软件开发人员喜欢带有内置技巧（可作为快捷方式）的用户界面。通常，客户需要简单、干净、清晰的用户界面，而不是那些技巧。

参考文献

Hoare, C.A.R., "Programming: Sorcery or Science?" IEEE Software, 1, 2 (April 1984), pp. 14-15.

原则 19 每个复杂问题都有一个解决方案

EVERY COMPLEX PROBLEM HAS A SOLUTION

Wlad Turski 说，“每一个复杂的问题，都有一个简单的解决方案…但这是错误的！”。无论任何人向你提出“只要遵循这 10 个简单步骤，软件质量问题就会消失”，或是其他类似建议，都要保持高度的怀疑。

参考文献

Turski, W., oral comments made at a conference in the late 1970s.

原则 20 记录你的假设

RECORD YOUR ASSUMPTIONS

系统运行的环境在本质上是无限的，不可能被完全理解。当我们开发一个系统，宣称要解决某个环境中的一个问题时，我们会对该环境进行假设。Manny Lehman 提出：“我们大约每 10 行代码就会做出一个假设，即使偏差了两三倍，每 20-30 行代码也会做出一个假设”。这些关于无限世界的有限假设会使你陷入麻烦。Lehman 描述了一种表现不如预期的直线加速器。一位物理学家提出，也许月球的相位会产生影响，对此每个人都说：“你一定是在开玩笑吧！”。然而，在考虑了月球的因素后，得到的方程式解释了大多数看似“不正确”的行为。这是一个假设（没有月球效应）无效的例子。

对需求工程、设计、编码和测试期间所做的所有假设，始终保持觉察是不可能的。尽管如此，我还是建议，对你有意识做出的假设做个记录。即使这个假设是显而易见的、或其它选项很荒谬，也要这样做。还要记录它们的影响，也就是说在产品中，这些假设是如何体现的？理想情况下，你应该会通过封装每个假设来隔离这些影响（原则 65）。

参考文献

Lehman, M., "Software Engineering, the Software Process and Their Support," Software Engineering Journal, 6, 5 (September 1991), pp. 243-258, Section 3.6.

原则 21 不同的阶段，使用不同的语言

DIFFERENT LANGUAGES FOR DIFFERENT PHASES

业界对“简单方法解决复杂问题”的永恒渴望（原则 19），促使许多人宣称：最佳的软件开发方法，是在整个开发生命周期中、使用相同的符号表达方法。既然在任何其他工程领域都并非如此，为什么在软件工程领域会是这样？在不同的设计活动中，电力工程师会使用不同的表达方法：方框图、电路图、逻辑图、时序图、状态转换表、柱状图等。这些表达方法为我们提供了在思维中可操纵的模型。使用越多的符号、越丰富多样的表达方式，我们就越能更好地对开发中的产品进行可视化。除非对所有阶段都是最优选择，否则为什么软件工程师想要将 Ada 用于需求、设计和代码？除非对所有阶段都是最优选择，否则为什么要在所有阶段都使用“面向对象”的方法？

对于需求工程，应该选择一组最优的技术和语言（原则 47 和 48）。对于设计工作，应该选择一组最优的技术和语言（原则 63 和 81）。对于编码，应该选择一个最适合的语言（原则 102 和 103）。在不同阶段之间转换是困难的。使用同一种语言并没有帮助。另一方面，如果一个语言从某方面在两个阶段都是最优选择，就务必使用它。

参考文献

Matsubara, T., "Bringing up Software Designers," American Programmer, 3, 7 (July-August 1990), pp, 15-18.

译者注

Ada，是一种程序设计语言。详见 https://zh.wikipedia.org/wiki/Ada

原则 22 技术优先于工具

TECHNIQUE BEFORE TOOLS

一个没规矩的木匠使用了强大的工具，会变成一个危险的没规矩的木匠。一个没规矩的软件工程师使用了工具，会变成一个危险的没规矩的软件工程师。在使用工具前，你应该“有规矩”（即，理解并遵循适当的软件开发方法）。然，你也要了解如何使用工具，但这和“有规矩”相比是第二位的。

我强烈建议，在投资于工具、以对某个技术“自动化”*之前*，先*手工*验证这个技术，并说服自己和管理层、这个技术是可行的。在大多数情况下，如果一项技术在手工时不灵，那在自动时它也不灵。

参考文献

Kemerer, C., "How the Learning Curve Affects Tool Adoption," IEEE Software, 9,3(May, 1992), pp. 23-28.

原则 23 使用工具，但要务实

USE TOOLS, BUT BE REALISTIC

一些软件工具（如 CASE）会让他们的用户更加高效。务必要使用它们。就像文字处理软件对作家而言是必须的助手，CASE 工具对软件工程师也是重要的助手。它们各自将使用者的开发效率提高了 10%到 20%。它们各自使用户修改和发展其产品的能力提高 25％到 50％，但是在两种情况下，艰难的工作（思考）都不是由工具完成。使用 CASE 工具，但要切实考虑其对开发效率的影响。请注意，70%的 CASE 工具在购买后从未被使用过。我认为，造成这种情况的主要原因是过度乐观和由此带来的失望，而不是工具的无效性。

参考文献

Kemerer, C., "How the Learning Curve Affects Tool Adoption," IEEE Software, 9,3(May, 1992), pp. 23-28.

译者注

CASE，是“电脑辅助软件工程”（Computer-Aided Software Engineering）的缩写。详见 https://en.wikipedia.org/wiki/Computer-aided_software_engineering

原则 24 把工具交给优秀的工程师

GIVE SOFTWARE TOOLS TO GOOD ENGINEERS

软件工程师使用工具(例如 CASE)变得更多产，就像作家使用文字处理软件变得更多产一样(原则 23)。然而，就像文字处理软件不能让一个平庸的小说家（写小说，但卖不出去）变得出色，CASE 工具也不能让一个平庸的软件工程师（写软件，但不可靠、不满足用户需求等）变的出色。因此，你想把 CASE 工具只提供给优秀的工程师。你最不想做的一件事，就是把 CASE 工具提供给平庸的工程师：你希望他们尽量少（而非多）的开发出质量低劣的软件。

原则 25 CASE 工具是昂贵的

CASE TOOLS ARE EXPENSIVE

工作站或者高端个人电脑去配置一套 CASE 工具环境，花销在 5000 至 15000 美元。CASE 工具本身，每份花费 500 到 50000 美元。工具每年需要的授权和维护费用一般相当于它们售价的 10%至 15%。而且，还需要为每一位接受培训的员工支付两到三天的工资。因此，每套软件的预期总安装成本可能超过 17000 美元（对于价格适中的 CASE 工具），而每套软件的经常性年度成本可能超过 3000 美元。

CASE 工具对软件开发是必需的。它们应该被视为业务成本的一部分。在做投资回报分析时，不仅需要考虑购买工具的高额费用，还需要考虑没有购买工具带来的更高代价（更低开发效率、更高的客户失望率、延迟的产品发布、增加的重复工作、更差的产品质量、增加的员工流动）。

参考文献

Huff,C., "Elements of a Realistic CASE Tool Adoption Budget" Communications of the ACM, 35, 4 (April 1992), pp. 45-54.

译者注

目前大量的软件工具已经可以免费获得。即使是收费软件，一般来说，其购买费用和软件工程师的人工成本相比也是很低的。对于一般的软件开发场景，这个原则可能已经不合时宜。但，关于软件工具的成本和收益的分析思路，仍然是可以借鉴的。

原则 26 “知道何时”和“知道如何”同样重要

“KNOW-WHEN” IS AS IMPORTANT AS KNOW-HOW

在行业中经常发生，一个工程师学习一项新技术后，判断这是“放之四海而皆准”的技术。同时，同组另一个人在学习另外一项新技术，一场情绪化的争辩随之而来。事实上，没有一方是正确的。知道如何很好地使用技术，既不会让技术本身成为好技术，也不会让你成为一名优秀的工程师。知道如何用好木工车床，并不能使你成为你一名好木匠。一名优秀的工程师了解很多不同种类的技术，并且知道每种技术何时适合项目或项目的一部分。一个好木匠知道多种工具的用法，知道很多不同的技巧，而且，最重要的是，知道什么时候该用哪一种。

在进行需求工程时，要了解哪种技术对问题的哪些方面最有用（原则 47）。当进行设计时，要理解哪些技术对系统的哪些方面最有用（原则 63）。当进行编码时，要选择最合适的编程语言（原则 102）。

原则 27 实现目标就停止

STOP WHEN YOU ACHIEVE YOUR GOAL

软件工程师遵循许多方法（也称为技术或流程）。每个方法都有各自的用途，通常对应软件开发的一个子目标。例如，结构化（或者面向对象）分析的目标是理解要解决的问题，DARTS 的目标是处理架构，结构化设计的目标是理清调用层次结构。这些例子中的方法都包含一系列的步骤。不要太过陷于具体的方法，而忘记了目标本身。不要为更换目标而感到内疚。例如，如果只执行了方法的一半步骤，你就理解了问题，那就停下来。另一方面，你需要对整个软件过程有很好的认识，因为基于本原则所抛弃的某个方法的后续步骤可能会对未来软件的使用产生重要影响。

译者注

有资料显示，DARTS 是 Design Approach for Real-Time Systems（实时系统设计方法）的缩写，参考：http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.6735&rep=rep1&type=pdf

原则 28 了解形式化方法

KNOW FORMAL METHODS

没有过硬的离散数学技能，使用形式化方法是不容易的。但是，它们的使用（即便是很简单的使用），可以极大地帮助发现软件开发中许多方面的问题。每个项目中，至少应该有一个人熟练使用形式化方法，以确保不会错过提升产品质量的机会。

很多人以为，使用形式化方法的唯一途径，就是完全使用它们来定义系统。其实并非如此。实际上，最有效的方法之一，是先用自然语言描述。然后再尝试用形式化方法去写其中某些部分。尝试用更形式化的方式书写，会帮助你发现在自然语言中存在的问题。修正自然语言表达中的问题，你会得到一个更好的文档。在完成之后，如果有需要，可以再把形式化的描述去掉。

参考文献

Hall, A. "Seven Myths of Formal Methods," IEEE Software, 7, 5 (September 1990), pp. 11-19.

原则 29 和组织荣辱与共

ALIGN REPUTATION WITH ORGANIZATION

普遍认为：日本软件工程师对待 bug 的态度，和美国不同。尽管有许多影响因素，有个日本的观念与此密切相关：产品中的缺陷是公司的耻辱；软件工程师引起的公司耻辱，是工程师的耻辱。这种观念在日本比美国更有效，因为日本劳动者倾向于一辈子只服务一家公司。不管在一家公司工作的时间长还是短，这种心态是很重要的。

一般而言，当任何人发现你在产品中的错误时，你应该心存感激，而不是试图辩解。人非圣贤，孰能无过。过而能改，善莫大焉！当发现一个错误，导致错误的人应该使其被周知，而不是藏着掖着。将错误广而告之有两个好处：(1) 帮助其他工程师，避免同样的错误 (2) 对后续的错误修正，也可以不那么抵触。

参考文献

Mizuno, Y., “Software Quality Improvement”, IEEE Computer, 16, 3 (March 1983), pp. 66-72.

原则 30 跟风要小心

FOLLOW THE LEMMINGS WITH CARE

即使有五千万人说傻话，那仍然是傻话。

安那托尔 •佛朗士（Anatole France）

大家都做的事情，不一定对你也是正确的。也许它是正确的，但你也应该评估它对你所处环境的适用性。这样的例子包括：面向对象，软件度量（原则 142、143、149、150 和 151)，软件复用（原则 84），过程成熟度（原则 163），计算机辅助软件工程（CASE，原则 22 至 25），原型设计（原则 11、12、13、42）。在所有案例中，这些方法都提供了非常积极的帮助，体现在提高质量、降低成本、提高用户满意度等方面。然而，这些好处只在它们有作用的组织中才会显现出来。尽管回报显著，它们的作用常常被过度宣传，其实它们并不是那么必然或通用。

当你学习“新”技术，不要轻易接受与之相关的不可避免的炒作（原则 129）。要仔细阅读，理性考虑它的收益和风险。在大规模应用之前要进行试验。但同时也绝对不要忽略“新”技术（参见原则 31）。

参考文献

Davis, a., "Software Lemmingineering," IEEE Software, 10, 6 (September 1993), pp. 79-81, 84

原则 31 不要忽视技术

DON'T IGONRE TECHNOLOGY

软件工程技术日新月异。在几年内对新的发展视而不见，是你无法承受的。软件工程的发展像波浪一样。每一波都会带来大量的“潮流元素”和流行语。尽管每一波只持续 5-7 年，但它们并不是简单消失。恰恰相反，其后每一波都是基于前一波的最好特征。（理想情况，“最好”应该指“最有效”，但遗憾的是，它往往指“最流行”)

有两种方式可以让你紧跟技术潮流：阅读正确的杂志，和正确的人交谈。《IEEE Software》期刊就是一个很好的渠道，可以了解未来 5 年内可能有用的技术。《PC Week》、《MacWorld》等是学习硬件平台、常见商用工具和语言的好地方。要通过和人交谈来学习，就要找到正确的人。虽然和同事交流很必要，但还不够。每年都应该努力参加 1-2 个关键会议。和参会者的交流，很可能比会议报告更重要。

原则 32 使用文档标准

USE DOCUMENTATION STANDARDS

如果你的项目、组织或客户要求遵循一套文档标准，就要遵循它。无论如何，永远不要抱怨标准，认为这是不需要的。所有我熟悉的标准，无论是政府标准还是商业标准，都提供了组织和内容方面的指导。

创新！即遵循标准，同时聪明的执行。无论标准怎么规定，把你知道应有的内容都包含进去。这意味着用清晰的语言来编写，意味着添加额外的有意义的组织层级。如果你的文档没有被要求遵循某个标准，至少使用检查清单来检查是否有重大的遗漏。IEEE 发布的文档标准，是我所知道的、最广泛的可用软件文档标准之一。

参考文献

IEEE Computer Society, Software Engineering Standards Collection, Washington, D,C.: IEEE Computer Society Press, 1993.

原则 33 文档要有术语表

EVERY DOCUMENT NEEDS A GLOSSARY

当阅读文档遇到不懂的术语时，我们都会感到沮丧。但当我们在术语表中查到说明时，沮丧的情绪顷刻就烟消云散了。

所有术语的定义都应该以这样的方式编写：定义中使用的任何单词，都应该尽量避免再去术语表中查找含义。一种技巧是首先用日常用语解释，然后再使用术语解释。在术语的说明文字中，在其它地方定义的术语要用*斜体*标识。例如：

> **数据流图**： 是图形化符号，用于展示系统的功能、数据库、和系统有关的环境之间的信息流动。通常用于：*结构分析*，*转换*（气泡表示），*数据流*（箭头表示）和*数据存储*（两条平行线表示），以及*外部实体*（三角形表示）。

原则 34 软件文档都要有索引

EVERY SOFTWARE DOCUMENT NEEDS AN INDEX

这条原则，对于所有的软件文档的*读者*来说都是不言自明的。令人惊讶的是，很多*作者*并没有意识到这一点（想想，每个作者有时也是读者）。*索引*通常是文档所使用的所有术语和概念的列表，包括一个或多个页码用于标记术语或概念在哪里被定义、使用或引用。对于需求、设计、编码、测试、用户和维护文档来说都是如此。索引可以帮助读者快速查找信息，对于文档后续的维护和优化也很重要。

现代文字处理软件提供将索引引用嵌入到正文中的指令，这使创建索引变得简单。然后文字处理软件会对索引进行编译，然后按字母顺序排列，并将结果输出。大多数 CASE 工具也能生成可用的索引。

原则 35 对相同的概念，用相同的名字

USE THE SAME NAME FOR THE SAME CONCEPT

写小说时，保持读者的兴趣是第一目标；而在技术文档中，必须使用相同的术语来表示相同的概念，使用相同的语句结构来表述相似的信息。否则会令读者感到困惑，导致读者需要花费时间确认，在重述中是否有新的技术信息。应该把这条原则应用到所有技术文档的写作中，包括需求规格说明、用户手册、设计文档、代码中的注释等等。

举个例子：

有三类特殊命令。常规命令有四种类型。

不如写为：

有三类特殊命令。有四类常规命令。

参考文献

Meyer, B., "On Formalism in Specifications," IEEE Software, 2, 1 (January 1985), pp. 6-26.

原则 36 研究再转化，不可行

RESEARCH-THEN-TRANSFER DOESN'T WORK

关于软件工程研究所中令人难以置信的技术成就，有大量报道。但它们很少能应用于软件开发实践。原因是：

1. 一般来说，软件研究者很少有开发实际系统的经验。
2. 软件研究者可能会发现，在解决一些技术问题的时候没有必要花费过多时间去"适配"真实场景，这样可以使得解决问题变得更快更容易。
3. 研究者和实践者在用语上存在巨大的分歧，导致他们很难相互沟通。于是研究者更愿意在越来越多的无实际意义的问题上演示他们的想法。

要实现从研究所到开发机构的最成功的成果转化，从一开始双方就要紧密合作。需要使用工业界的环境作为萌发想法并验证效果的实验室，而不是在想法成形后再做技术转化。

参考文献

Basili, V., and J. Musa, "The future Engineering of software: A Management Perspective", IEEE Computer, 24, 9(September 1991), pp.90-96.

原则 37 要承担责任

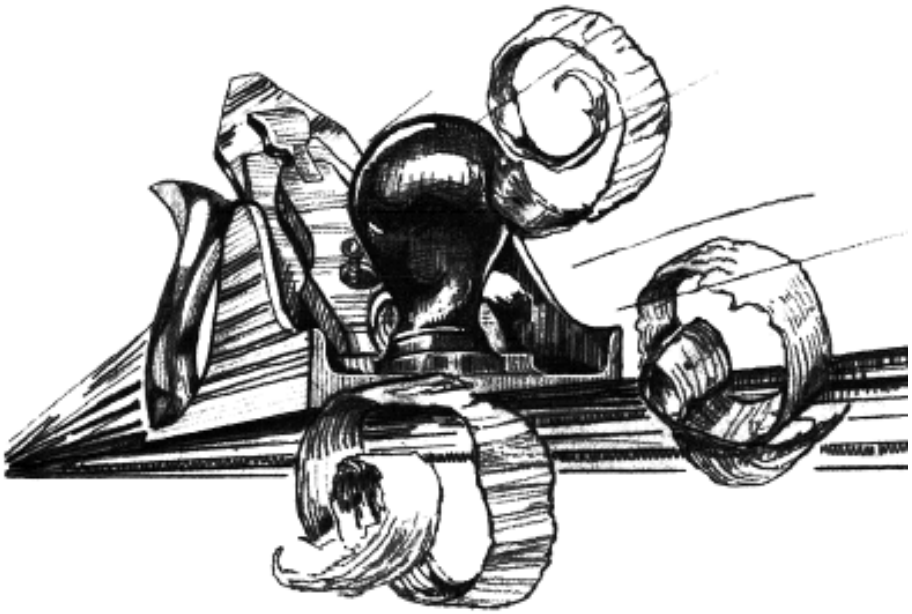
TAKE RESPONSIBILITY

在所有工程学科中，如果一个设计失败，工程师会受到责备。因此，当一座大桥倒塌，我们会问“工程师哪里做错了？”。当一个软件失败了，工程师很少受到责备。如果他们被责备了，他们会回答，“肯定是编译器出错了”，或“我只是按照指定方法的 15 个步骤做的”，或“我的经理让我这么干的”，或“计划剩余的时间不够”。 事实是，在任何工程学科中，用最好的方法也可能产出糟糕的设计，用最过时的方法也可能做出精致的设计。

不要有任何借口。如果你是一个系统的开发者，把它做好是你的责任。要承担这个责任。要么做好，要么就压根不做。

参考文献

Hoare, C.A.R., "software Engineering: A keynote Address," IEEE 3rd International Conference on Software Engineering, 1978, pp. 1-4.



第三章 需求工程原则

REQUIREMENTS ENGINEERING PRINCIPLES

*需求工程*包括以下活动：(1) 提出或研究需要解决的问题；(2) 具体说明一个能解决该问题的系统外部（黑盒）行为。需求工程的最终产出是需求规格说明（Requirement Specification）。

原则 38 低质量的需求分析，导致低质量的成本估算

POOR REQUIREMENTS YIELD POOR COST ESTIMATES

造成低质量成本估算的前五个原因，都与需求分析流程有关：

1. 频繁的需求变更
2. 不完整的需求列表
3. 不充足的用户沟通
4. 低质量的需求规格说明
5. 不充分的需求分析

可以使用原型，来降低需求不准确的风险。可以使用配置管理，来控制需求变更。应该为将来的发布，规划好新的需求。应该使用更正式的方法，进行需求分析和需求规格说明。

参考文献

Lederer, A., and J. Prasad, "Nine Management Guidelines for Better Cost Estimating," Communications of the ACM, 35, 2(February 1992), pp. 51-59.

原则 39 先确定问题，再写需求

DETERMINE THE PROBLEM BEFORE WRITING REQUIREMENTS

当面对他们认定的问题时，大多数工程师都会匆忙提供解决方案。如果工程师对这个问题的看法是正确的，那么解决方案可能奏效。然而，问题往往是难以捉摸的。例如，唐纳德·高斯（Donald Gause）和杰拉尔德·温伯格（Gerald Weinberg）描述了高层办公楼中的一个“问题”，里面的住户抱怨电梯等待时间太长。这真的是个问题吗？这是谁的问题？从居住者的角度来看，问题可能是他们浪费了太多时间。从房主的角度来看，问题可能是入住率（及租金）可能会下降。

显而易见的解决办法是提高电梯的速度。但其他想法可能包括（1）增加新电梯（2）错开工作时间（3）给快递保留一些电梯（4）提高租金（以便业主能够容忍入住率降低）（5）改进电梯使用的“归位算法”（homing algorithm），以便在闲置时移动到高需求楼层。这些解决方案的成本、风险和时间延迟差别巨大。而任何一个方案生效，取决于特定的场景。在试图解决问题前，针对面临问题的人及问题的本质，要确保探索所有的可能选择。在解决问题时，不要被最初方案带来的潜在兴奋所蒙蔽。方案的变化总是比构建系统的成本低。

参考文献

Gause, D., and G. Weinberg, Are Your Lights On? New York: Dorset House, 1990.

原则 40 立即确定需求

DETERMINE THE REQUIREMENTS NOW

需求难以理解，更难以说明。对此，错误的解决方法是草率地完成需求规格说明，匆忙地进行设计和编码，然后徒劳地希望：

1. 任何系统都比没有系统要好。
2. 需求迟早会解决的。
3. 或者，设计师们在开发的过程中会明确可以开发什么。

正确的解决方法是，*立刻*不计代价、尽可能多地获取需求信息。要使用原型的方法。要和更多的客户交谈。可以与客户一起工作一个月，以获得客户使用情况的第一手信息。要收集数据。要使用所有可能的手段。现在就把你所理解的需求记录下来，并规划构建一个满足这些需求的系统。如果你预期需求会发生很大变化，那也没关系。可以用增量的方式开发（原则 14），但这并不是在任何一个增量开发上做不好需求规格说明的借口。

参考文献

Boehm, B., "Verifying and Validating Software Requirements and Design Specifications," IEEE Software, 1, 1(January 1984), pp. 75-88.

原则 41 立即修复需求规格说明中的错误

FIX REQUIREMENTS SPECIFICATION ERRORS NOW

如果在需求规格说明中有错误，你将付出以下代价：

- 如果错误保持到系统设计阶段，定位和修复要多花 5 倍的代价。
- 如果保持到编码阶段，要多花 10 倍。
- 如果保持到单元测试阶段，要多花 20 倍。
- 如果保持到交付阶段，要多花 200 倍。

这是“要在需求分析阶段修复错误”的最令人信服的证据。

参考文献

Boehm, B., "Software Engineering," IEEE Transactions on Computers, 25, 12(Decemember 1976), pp. 1226-1241.

原则 42 原型可降低选择用户界面的风险

PROTOTYPES REDUCE RISK IN SELECTING USER INTERFACES

在全面开发之前，以低风险、高回报的方式关于用户界面达成一致，没有什么方法比原型更有效。有许多工具可以帮助快速创建屏幕演示。这些“故事板”可以给用户提供一个真实系统的印象。它们不仅有助于确认需求，还能赢得客户和用户的心。

参考文献

Andriole, S., "Storyboard Prototyping for Requirements Verification," Large Scale Systems, 12(1987), pp. 231-247.

原则 43 记录需求为什么被引入

RECORD WHY REQUIREMENTS WERE INCLUDED

在创建需求规格说明时，要完成很多工作：访谈、辩论、讨论、架构调研、工作机制描述、问卷、JAD/RAD 环节、其他系统的需求规格说明、早期的系统层面的需求分析。需求规格说明描述了从以上这些工作获得的需求分析结果。假设后续用户要求做一个需求变更。我们需要知道原始需求的动机，以便确认是否可以安全地变更。同样，当系统无法满足某个需求时，我们需要知道需求的背景，才能决定是修改系统设计以满足需求、还是修改需求以匹配系统。

当做出需求决策时（例如响应时间应该是两秒种），记录一个指向其来源的标识。例如，如果决策是在客户访谈时作出的，需要记录日期、时间、及访谈的参与者。理想情况下，应明确所参考的文字、录音或录像记录。只有基于这样的档案记录，才能 (1) 随后扩展需求，或 (2) 在已完成的系统不能满足需求时作出响应。

参考文献

Gilb, T., Principles of Software Engineering Management, Reading, Mass.: Addision Wesley, 1988, Section 9.11.

译者注

JAD，即“联合应用开发”（Joint Application Development）

RAD，即“快速应用开发”（Rapid Application Development）

原则 44 确定子集

IDENTIFY SUBSETS

在编写需求规格说明时，要清晰识别有用的需求的最小子集。同时，还要识别使最小子集越来越实用的最小增量。这种识别为软件设计者提供了洞察最佳软件设计的视角。例如，它将使设计师能够：

1. 更容易地使每个组件只包含一个功能。
2. 选择更具内聚性和可扩展性的架构。
3. 了解如何在日程或预算紧缩的情况下减少功能。

记录子集的一种非常有效的技巧，是在软件需求规格说明中的每个需求旁边加上几列。每列对应不同的版本。这些版本可以代表一个产品的多种功效，每种功效对应一个不同的客户或场景，它们也可以代表产品随时间日益提高的层级。在上述两种情况下，在适当的列中放置一个“X”，以指示哪些版本将具有哪些功能。

参考文献

Parnas, D., "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, 5, 2(March 1979), pp. 128-138.

原则 45 评审需求

REVIEW THE REQUIREMENTS

许多相关方都对产品的成功有影响：用户，客户，销售，开发，测试，质量保证等等。所有这些人也对需求规格说明的正确性和完整性有影响。在进行设计或编码之前，应该对需求规格说明进行正式的评审。

由于需求规格说明是用自然语言编写的，对其进行评审没有简单的方法；然而，Barry Boehm 给出的指引可以使评审变得容易一些。当然，如果需求规格说明的某些部分是用更正式的语言编写的（原则 28、54 和 55），则可以对这些部分进行手工的评审（因为它们没有歧义），并在某些情况下“执行”。可执行的需求 [如 Pamela Zave 的 PAISLey（"An Insider's Evaluation of PAISLey," IEEE Transactions on Software Engineering, 17, 3 (March 1991), pp. 212-225)] 可以用适当的工具进行演示，相关人员可以“看到”系统功能，而不只通过“阅读”了解系统如何运行。

参考文献

Boehm, B., "Verifying and Validating Software Requirements and Design Specifications," IEEE Software, 1, 1 (January 1984), pp. 75-88.

原则 46 避免在需求分析时进行系统设计

AVOID DESIGN IN REQUREMENTS

需求阶段的目标是明确系统的外部行为。这些外部行为需要足够明确，以保证当使用需求规格说明作为指引时、所有设计人员都能对系统的目标行为做出同样的理解。但在需求阶段不应该去明确软件架构或者算法，因为这是设计人员的领域。后续设计人员会选择能够最好地满足需求的架构和算法。

如果撰写需求的人发现，很难在没有系统设计（例如，通过有限状态机描述系统行为）的情况下、毫无歧义的定义外部行为，应该留下这样的信息：

警告：这里包含的设计，仅用于辅助理解产品的外部行为。在系统外部行为相同的情况下，设计人员可以选择任何设计方案。

参考文献

Davis, A., Software Requirements: Objects, Fuctions and States, Englewood Cliffs, N.J.: Prentice Hall, 1993, Section 3.1.

原则 47 使用正确的方法

USE THE RIGHT TECHNIQUES

没有任何一种需求分析方法适用于所有软件。复杂软件的需求，只有使用多种方法才能被充分理解。要使用对于你的软件来说最合适的一种或一组方法。

例如，对于数据密集型的软件，应使用实体关系图（Entity-Relation Diagram）；对于反应式（实时）系统，应使用有限状态机或者状态图；对于有同步难题的软件，应使用 Petri 网；对于决策密集型的软件，应使用决策表；诸如此类。

参考文献

Davis, A., "A Comparison of Techniques for the Specification of External System Behavior," Communications of the ACM, 31, 9 (September 1988), pp. 1098-1115.

原则 48 使用多角度的需求视图

USE MULTIPLE VIEWS OF REQUIREMENTS

任何单一的需求视角，都不足以理解或描述一个复杂系统的预期外部行为。比起使用结构化分析、面向对象分析或状态图，要选择并使用一个有效的组合。

例如，对于一个复杂的系统，你可能需要使用面向对象分析来评估，那些与软件相关的重要实体。面向对象分析（OOA）可以帮助确定实体，并且理解它们之间的关系和相关属性。你可能需要使用有限状态机来描述，用户操作界面的预期行为。你可能需要使用决策树来描述，在响应外部条件的复杂组合时，系统的预期行为。诸如此类。

参考文献

Yeh, R., P. Zave, A. Conn, and G. Cole, Jr, "Software Requirements: New Directions and Perspectives," in Handbook of Software Engineering, C.Vick and C. Ramamoorthy, eds., New York: Van Nostrand Reinhold, 1984, pp. 519-543.

原则 49 合理地组织需求

ORGANIZE REQUIREMENTS SENSIBLY

我们通常层次的组织需求。这有助于读者理解系统功能，也有助于需求编写者在变更时定位章节。组织需求有很多方式，最适合方式的选择取决于具体产品。

要以一种对客户、用户或者市场人员最自然的方式来组织需求。这里有一些例子：从用户（类别）的角度，从激励（类别）的角度，从反馈（类别）的角度，从对象（类别）的角度，从功能（类别）的角度，从系统模式的角度等。举例来说，对于电话交换系统，可依次按照功能类别、功能、用户来组织需求：

1. 单方通话
    1.1 呼叫转移
    1.2 呼叫驻留
2. 双方通话
    2.1 本地通话
        2.1.1 主叫方视角
        2.1.2 被叫方视角
    2.2 长途通话
        2.2.1 主叫方视角
        2.2.2 被叫方视角
3. 多方通话
    3.1 电话会议
    3.2 接线员协助呼叫

参考文献

Davis, A., Software Requirements: Objects, Fuctions and States, Englewood Cliffs, N.J.: Prentice Hall, 1993, Section 3.4.11.

原则 50 给需求排优先级

PRIORITIZE REQUIREMENTS

并非所有需求都是同样重要的。对于载人航天飞行器来说，需求可能同时包括速溶橙汁和全功能生命支持系统。但显然前者没有后者重要。如果没有果汁，你大概不会中断发射，但如果生命支持系统不能工作，你肯定要终止发射。

一种设定需求优先级的方法，是给需求规格说明中的每个需求加上后缀 M、D 或者 O 来表示必须（Mandatory）、期望（Desirable）、可选（Optional）。尽管这里创造了一个可选需求的矛盾概念，但是它清楚准确地说明了相对优先级。另一个更好的方式是给每个需求按照重要性打 0-10 分。

参考文献

Davis, A., Software Requirements: Objects, Fuctions and States, Englewood Cliffs, N.J.: Prentice Hall, 1993, Section 3.4.11.

原则 51 书写要简洁

WRITE CONCISELY

我经常看到需求规格说明中包含类似如下的描述：

目标跟踪功能应提供显示所有活动目标的当前跟踪坐标的能力。

和下面的描述对比一下：

在跟踪时，系统应显示所有活动目标的当前位置。

原则 52 给每个需求单独编号

SEPARATELY NUMBER EVERY REQUIREMENT

需求规格说明中的每条需求能很容易被引用，这很重要。这对后续在设计中追踪需求（原则 62）和在测试中追踪需求（原则 107）是必要的。

最简单的方法是给每个需求打上唯一标识符（例如“【需求 R27】”）。另外一种方法是给每个段落编号，然后对“段落 i.j”中的“句子 k” 中的需求，编号为“需求 i.j-sk”。 第三种方法是遵循以下规则：每条要求都包含词语“应该”（或除保留词外的任何其他合适词语），例如：“系统应该在 0.5 秒内发出拨号音，...”。 然后，使用简单的文本匹配程序提取、编号，并在附录中列出所有需求。

参考文献

Gilb, T., Principles of Software Engineering Management, Reading, Mass.: Addson Wesley, 1988, Section 8.10.

译者注

这里作者所推荐的方法可能已经过时，但是“对需求进行编号以便追踪”的思路仍然是非常正确的。

原则 53 减少需求中的歧义

REDUCE AMBIGUITY IN REQUIREMENTS

大多数需求规格说明用自然语言编写。由于词、短语和句子的语义不严密，自然语言存在固有的歧义问题。尽管消除需求中所有歧义的唯一方法是使用形式语言，但是通过仔细的评审和重写有明显、或微妙歧义的文字，可以在一定程度上减少歧义。关于歧义及其后果，阿尔-戴维斯（Al Davis）提供了非常多的例子。

减少歧义的三个有效方法是：

1. 对软件需求规格说明使用范根检查法（Fagan Inspection）。
2. 尝试对需求构建更形式化的模型，并在发现问题后重写自然语言的描述（原则 28）。
3. 组织好软件需求规格说明，使对开页分别包含自然语言描述和形式模型描述。

参考文献

Davis, A., Software Requirements: Objects, Fuctions and States, Englewood Cliffs, N.J.: Prentice Hall, 1993, Section 3.4.2

译者注

范根检查法（Fagan Inspection），是在软件开发过程中、尝试从文档中发现缺陷的流程。详见以下地址中的说明：https://en.wikipedia.org/wiki/Fagan_inspection

原则 54 对自然语言辅助增强，而非替换

AUGMENT, NEVER REPLACE, NATURAL LANGUAGE

在试图减少需求中的歧义时，软件开发人员经常决定要使用比自然语言更精准的符号。这当然应该鼓励，通过使用有限状态机、谓词逻辑、Petri 网络、状态图等来减少歧义（原则 53）。然而，这么一来，对于那些计算机科学或数学背景不如需求编写者的人来说，需求规格说明变得更不好理解（原则 56）。

为了缓解使用形式化符号带来的这个问题，应该保留自然语言描述。事实上，一个好主意是，在对开的页面上并行保留自然语言和更加形式化的描述。要对两者做人工核对，以保证一致性。这样就可以让所有读者都理解，而且无数学知识的读者也能获得有用的信息。

参考文献

Meyer, B., "On Formalism in Specifications," IEEE Software, 2, 1(January 1985), pp. 6-26.

原则 55 在更形式化的模型前，先写自然语言

WRITE NATURAL LANGUAGE BEFORE A MORE FORMAL MODEL

原则 54 说，要创建同时包含自然语言和形式化模型的需求规格说明。一定要先写自然语言的描述。如果先基于形式化模型描述，会倾向于用自然语言描述模型，而非描述解决方案系统。

比较下面两段，以理解我的意思：

> *为拨打长途电话，用户应该拿起电话。系统应该在 10 秒内返回一个拨号音。用户应该拨"9"。 系统应该在 10 秒内返回一个不同的拨号音。*

> *系统包含四种状态：空闲，拨号音，不同的拨号音和接通。要从"空闲"状态转换到"拨号音"状态，应拿起电话。要从"拨号音"状态转换到"不同的拨号音"状态，应拨"9"。*

请注意，在后一个例子中，文字描述完全没给读者提供帮助。最好的方法是：(1) 写自然语言，（2） 写形式化模型，（3）根据形式化模型中发现的问题去修改自然语言、以减少歧义。

原则 56 保持需求规格说明的可读性

KEEP THE REQUIREMENTS SPECIFICATION READABLE

需求规格说明必须可被大范围的个人或组织阅读和理解：用户、客户、市场营销人员、需求作者、设计师、测试人员、管理人员等等。文档必须使所有人充分领会所需要的、并被开发中的系统，从而不会出现意外。

仅当你可以保证各个版本之间的一致性时，创建多个需求规格说明（每个对应一个相关方的子集）才是可行的。一种更有效的方法是，保留自然语言（原则 54），同时结合更形式化的多视角（原则 48 和 53）。

参考文献

Davis, A., Software Requirements: Objects, Functions, and States, Englewood Cliffs, N.J.: Prentice Hall, 1993, Section 3.4.5.

原则 57 明确规定可靠性

SPECIFY RELIABILITY SPECIFICALLY

软件可靠性很难准确说明。不要因为含糊其辞，让问题变得更困难。比如，“这个系统会 99.999% 可靠”没有任何意义。这是意味着，这个系统在一年内宕机不会超过 5 分钟，但可以偶尔出现一个错误（如一个电话系统可能偶尔转错电话）？还是意味着，每处理十万次事务，最多只能犯一个错误（例如，一个病人监护系统不会“致死”超过十万分之一的病人）？

当编写可靠性需求时，要区分以下概念：

1. *需求失效（Failure on Demand）*。系统无法正确响应的可能性（百分比）是多少？例如，“*系统应正确报告 99.999% 的 病人生命体征异常*”。
2. *失败率（Rate of Failure）*。这个概念和“需求失效”类似，但它以单位时间来衡量。例如，“*系统无法正确报告病人生命体征异常的次数，每年不超过 1 次*”。
3. *可用性（Availability）*。系统不可用的时间百分比是多少？例如，“*在任何自然年内，电话系统在（至少） 99.999% 时间可用*”。

参考文献

Sommerville, I., Software Engineering, Reading, Mass.: Addison-Wesley, 1992, Section 20.1.

原则 58 应明确环境超出“可接受”时的系统行为

SPECIFY WHEN ENVIRONMENT VIOLATES "ACCEPTABLE" BEHAVIOR

需求规格说明通常会定义系统环境的特征。这些信息被用于做理智的设计决策。这通常意味着，开发人员有义务来容纳这些特性。但当系统环境超出这些限制时，系统部署后将会发生什么？

假设空中交通管制系统的需求规定，在一个区域中应能同时处理最多 100 架飞机。系统被开发出来并正确的满足这些需求。3 年后，偶然有 101 架飞机进入一个区域。软件应如何处理？

可能的选项有：

1. 打印错误信息：“系统环境超出需求规格”。
2. 崩溃（系统停止运行）。
3. 忽略第 101 架飞机。
4. 处理第 101 架飞机，但可能无法满足部分时间约束（比如屏幕的刷新间隔）。

很显然，第 1、2、3 选项都是不可接受的。然而基于（没有）在需求规格说明中的描述，它们都是正确的系统响应。正确的解决方案是：当环境超出为其定义的任何约束时，在软件需求规格说明中明确声明预期的系统响应。

参考文献

Davis, A., Software Requirements: Objects, Functions and States, Englewood Cliffs, N.J.: Prentice Hall, 1993, Section 5.3.2.

原则 59 自毁的待定项

SELF-DESTRUCT TBD'S

通常来讲，需求规格说明中不应包含待定项(TBD：To Be Determined)。显然，包含待定项的需求规格说明是未完成的，但可能有很好的理由接受、并将包含待定项的文档作为基线。当某些需求的精确性对重要设计决策无关键影响时，更是如此。

当创建一个待定项时，一定要为它加上“自毁”的注释，即要明确：到何时为止，由谁处理这个待定项。例如一个注释可能写为：“开发经理将在 1995 年 12 月之前解决这个待定项”。这确保这个待定项不会一直保留。

参考文献

IEEE, ANSI/IEEE Guide to Software Requirements Specifications, Standard 830-1994, Washington, D.C.: IEEE Computer Society Press, 1994.

原则 60 将需求保存到数据库

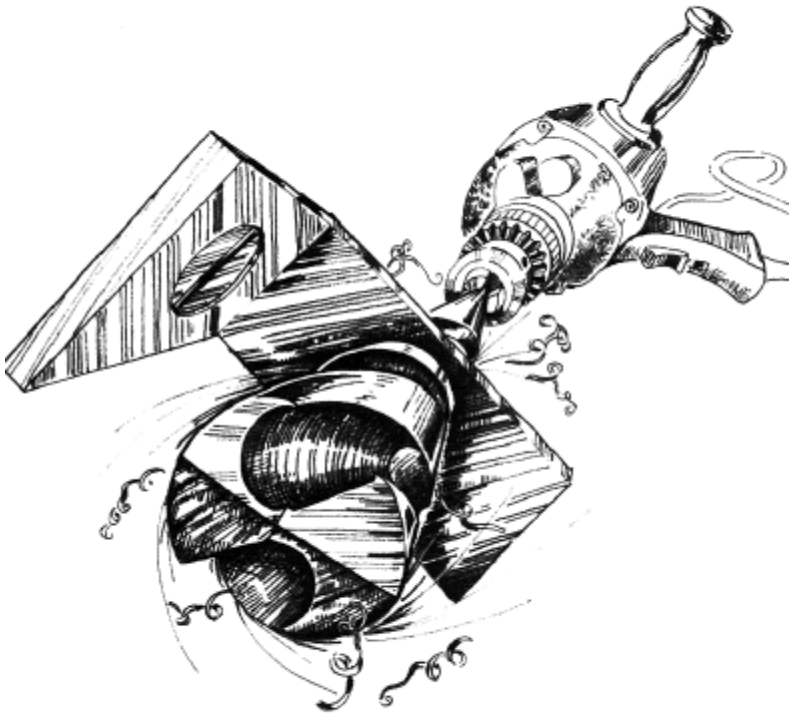
STORE REQUIREMENTS IN A DATABASE

需求是复杂和非常不稳定的。出于这些原因，应将他们保存到电子设备、最好是数据库中。这将方便进行修改、排查修改带来的影响、记录特定需求的细节属性等等。

在数据库中存储的内容可能包括：需求的唯一标识（原则 52），需求的文本描述，与其它需求的关系（例如对需求的更抽象或更详细的描述），需求的重要性（原则 50），预期的需求易变性，指向需求来源的标识（原则 43），需求应用的产品版本（原则 44 和 178），等等。理想情况下，需求规格说明本身就是整个数据库有组织的导出。

译者注

现在已经可以使用一些现成的工具来管理需求，如百度的 iCafe、腾讯的 TAPD、Atlassian 的 Jira 等。



第四章 设计原则

DESIGN PRINCIPLES

设计包括以下活动：(1) 定义满足需求的软件架构（architecture）；(2) 具体说明架构中的各个软件组件的算法。架构包括：软件中所有模块的定义；它们之间如何提供接口；它们之间如何组装；组件的拷贝如何实例化（即：在内存中创建并执行的组件拷贝）和销毁。设计的最终产出是设计规格说明（Design Specification）。

原则 61 从需求到设计的转换并不容易

TRANSITION FROM REQUIREMENTS TO DESIGN IS NOT EASY

需求工程最终会形成需求规格说明，是一个系统外部行为的详细描述。设计的第一步，是综合形成一个理想的软件架构。在软件工程领域，从需求到设计的转换，没有理由比在其他任何工程学科中更容易。设计很难。从外部视角到内部最优设计的转换，从根本上说是一个难题。

一些方法声称，将需求规格说明中的"架构"作为软件架构，这样转换就是容易的。因为设计是困难的，有以下几种可能性：

1. 在需求分析阶段，完全没有考虑选择最优设计。在这种情况下，不能接受将需求阶段的设计作为最终设计。
2. 在需求分析阶段，列出各种可选设计，并进行分析及选优。在确定需求基线、做出创作或购买的决策和进行开发成本估算之前，组织无法负担的起去做彻底的设计（通常占开发总成本的 30% 到 40%）。
3. 该方法假设某种软件架构对所有软件都是最理想的。这显然是不可能的。

参考文献

Cherry, G., Software Construction by Object-Oriented Pictures, Canadaigua, New York: Thought Tools, 1990, p.39.

原则 62 将设计追溯至需求

TRACE DESIGN TO REQUIREMENTS

设计软件时，设计者必须知道，哪些需求能被每个组件满足。当选择软件架构时，重要的是所有需求都能被覆盖。软件部署后，当检测到故障时，维护人员需要快速分离出那些最有可能包含故障原因的软件组件。在维护期间，当一个软件组件被修复时，维护人员需要知道哪些需求可能会受到不利的影响。

所有这些要求可以通过创建一个大的二维表格来满足，它的行对应所有的软件组件，它的列对应需求规格说明中的每个需求。任何位置的 1 表示此设计组件有助于满足此需求。注意，没有 1 的行表示该组件没有用处，没有 1 的列表示一个未被满足的需求。有人认为这张表格很难维护。但我认为你需要这张表格去设计或者维护软件。没有这张表格，你可能设计出一个不正确的软件组件，而且在维护期间会花费过多的时间。这张表格的成功创建，依赖于你唯一引用每个需求的能力 （原则 52）。

参考文献

Glass, R., Building Quality Software, Englewood Cliffs, N.J.: Prentice Hall, 1992, Section 2.2.2.5.

原则 63 评估备选方案

EVALUATE ALTERNATIVES

在所有工程学科中，一个重要思想是：详细列出多种方法，在这些方法之间权衡分析，并最终采用一种。在需求达成一致后，你必须充分考虑各种架构和算法。你当然不会想直接使用那种在需求规格说明中提到的架构（原则 46）。毕竟，在需求规格说明中选择这种架构是为了优化系统外部行为的可理解性。你需要的架构是与需求规格说明中包含的需求保持最优一致的那种架构。

例如，通常架构的选择是为了优化吞吐、响应时间、可变更性、可移植性、互操作性、安全性或者可用性，同时满足功能需求。实现目的的最好方法是列举各种软件架构，根据目标分析（或模拟）每种架构，并选择最佳方案。一些设计方法会导致特定的软件架构。因此，要有多种架构，就要使用多种设计方法。

参考文献

Weinberg, G., Rethinking Systems Analysis and Design, New York: Dorset House, 1988, Part V.

原则 64 没有文档的设计不是设计

DESIGN WITHOUT DOCUMENTATION IS NOT DESIGN

我经常听到软件工程师说，“我已经完成了设计，剩下的就是写文档”。这种想法毫无道理。你能想象一个建筑设计师说, “我已经完成了你新家的设计，剩下的就是把它画出来”，或者一个小说家说，“我已经完成了这部小说，剩下的就是把它写下来”？ 设计，是在纸或其他媒介上，对恰当的体系结构和算法的选择、抽象和记录。

参考文献

Royce, W., "Managing the Development of Large Software Systems," WESCON '70, 1970; reprinted in 9th International Conference on Software Engineering, Washington, D.C.: IEEE Computer Society Press, 1987, pp.328-338.

原则 65 封装

ENCAPSULATE

信息隐藏，是一个简单且经过验证的概念，它使软件更容易测试和维护。大多数软件模块应该对所有其他软件隐藏一些信息。这些信息可能是：数据的结构，数据内容，算法，设计决策，硬件接口，用户接口或给其它软件提供的接口。信息隐藏有助于隔离错误，因为当隐藏的信息在某种方式下变得不可接受时（如：失败，或必须变更以适应新的需求），只有隐藏该信息的软件需要被检查或更改。封装，指的是一组统一规则集，关于哪些类型的信息应该被隐藏。例如，在面向对象设计中，封装通常是指在每个对象中隐藏属性（数据）和方法（算法）。除了通过调用方法，其他对象无法影响属性的值。

参考文献

Parnas, D., "On the Criteria to Be Used in Decomposing Systems into Modules," Communications of the ACM, 15, 12 (December 1972), pp.1053-1058.

原则 66 不要重复造轮子

DON'T REINVENT THE WHEEL

当电子工程师设计新的印刷电路板时，他们会查阅可用集成电路的目录，以选择最合适的组件。当电子工程师设计新的集成电路时，他们会查阅标准单元的目录。当建筑师设计新房屋时，他们会查阅预制门窗、饰条和其他组件的目录。所有这些都被称为“工程”。 软件工程师经常一次又一次地重新发明组件。他们很少修补已有的软件组件。有趣的是，软件业称这种罕见的实践为“复用”，而不是“工程”。

参考文献

Ramamoorthy, C. V., V. Garg, and A. Prakash, "Programming in the Large," IEEE Transactions on Software Engineering, 12, 7 (July 1986), pp.769-783.

原则 67 保持简单

KEEP IT SIMPLE

一个简单的架构或者一个简单的算法，在实现高可维护性方面，有很长的路要走。记住 KISS 原则。另外，当你将软件分解成子组件时，记住一个人很难同时理解超过 7（加或减 2）个事物。托尼·霍尔（Tony Hoare）说过：

> *构建软件设计有两种方法。一种方法是使它简单到明显没有缺陷，另一种方法是使它复杂到没有明显的缺陷。*

参考文献

Miller, G., "The Magical Number Seven, Plus or Minus Two," The Psychological Review, 63, 2 (March 1956), pp.81-97.

译者注

[1] KISS，即 Keep It Simple and Stupid。详见 https://baike.baidu.com/item/KISS原则

[2] 托尼·霍尔（Tony Hoare），是英国计算机科学家，图灵奖得主。他发明了快速排序算法（Quick Sort）。

原则 68 避免大量的特殊案例

AVOID NUMEROUS SPECIAL CASES

在你设计算法时，无疑会发现存在许多例外情况。例外情况会使得特殊案例加入到算法中。每一个特殊案例都会使你更难调试，并使其他人更难修改、维护和增加功能。

如果你发现太多的特殊案例，你可能设计了一个不合适的算法。应重新思考并重新设计算法。见相关原则 67。

参考文献

Zerouni, C., as reported by Bentley, J., More Programming Pearls, Reading, Mass.: Addison-Wesley, 1988, Section 6.1.

原则 69 缩小智力距离

MINIMIZE INTELLECTUAL DISTANCE

艾兹格·迪科斯彻（Edsger Dijkstra）将智力距离（Intellectual Distance）定义为，现实问题和对此的计算机解决方案之间的距离。理查德·费莱（Richard Fairley） 认为，智力距离越小，维护软件就越容易。

为了做到这一点，软件的结构应该尽可能接近的模仿现实世界的结构。面向对象设计和杰克逊系统方法（Jackson System）等设计方法，将最小的智力距离作为主要的设计驱动。但是你可以使用任何设计方法去缩小智力距离。当然，要意识到"现实世界的结构"并不是唯一的。正如杰威德·西迪奇（Jawed Siddiqi）在 1994 年 3 月发表在《IEEE Software》的文章（标题为"Challenging Universal Truths of Requirements Engineering"，挑战需求工程的普遍真理）中所指出，不同的人在审视同一个现实世界时，往往会感知到不同的结构，这样构造出相当多样化的"构造的现实"。

参考文献

Fairley, R., Software Engineering Concepts, New York: McGraw-Hill, 1985.

译者注

[1] 艾兹格·迪科斯彻（Edsger Dijkstra），著名的计算机科学家，图灵奖得主。详见 https://baike.baidu.com/item/艾兹格·迪科斯彻/5029407

[2] 杰克逊系统方法（Jackson System），是一种软件开发方法。详见 https://en.wikipedia.org/wiki/Jackson_system_development

原则 70 将设计置于知识控制之下

KEEP DESIGN UNDER INTELLECTUAL CONTROL

如果设计是以能使其创建者和维护者完全理解的方式创建和记录的，那么这个设计就是在知识可控范围内的。

这种设计的一个基本属性是，它是分层构建的和多视角的。层次结构使读者能够抽象的理解整个系统，并在向更深层次移动时，理解越来越多的细节。在每个层次上，组件应该仅从外部视角描述（原则 80）。此外，（在层次结构中的任何级别）任何单个组件都应该展现出简单和优雅。

参考文献

Witt, B., F. Baker, and E. Merritt, Software Architecture and Design, New York: Van Nostrand Reinhold, 1994, Section 2.5.

原则 71 保持概念一致

MAINTAIN CONCEPTUAL INTEGRITY

概念一致是高质量设计的一个特点。它意味着，使用有限数量的设计“形式”，且使用方式要统一。设计形式包括：模块如何向调用方通知错误，软件如何向用户通知错误，数据结构如何组织，模块通信机制，文档标准，等等。

当设计完成后，它应该看起来都是一个人做的，尽管它其实是很多参与者的产出。在设计过程中，经常会有偏离既定形式的诱惑。对这样的诱惑，有些是可以让步的，比如理由是提升系统的一致性、优雅性、简单性或性能。有些则不能让步，比如仅仅为了确保某个设计者在设计中留下自己的印记。概念一致比自我满足更重要。

参考文献

Witt, B., F. Baker, and E. Merritt, *Software Architecture and Design*, New York: Van Nostrand Reinhold, 1994, Section 2.6.

原则 72 概念错误比语法错误更严重

CONCEPTUAL ERRORS ARE MORE SIGNIFICANT THAN SYNTACTIC ERRORS

在软件开发中，不论是写需求规格说明、设计文档、代码还是测试，我们都花费大量精力来排除语法错误。这是值得赞许的。 然而，构建软件真正的困难来自概念性错误。大多数开发者会花很多时间寻找并修改语法错误，因为一旦发现，这些看起来愚蠢的错误，在某种程度上会使开发者感到愉悦。相反，当开发者发现概念性错误，通常会感觉自己在某些方面不足或无能。 不管你有多优秀，都会犯概念性错误。去寻找它们吧。

在开发的各个阶段，问自己一些关键的问题。在需求阶段，问自己，“这是客户想要的吗？”。在设计阶段，问自己，“这个架构在压力下可以正常工作吗？”，或者，“这个算法真的适用于各种场景吗？”。在编码阶段，问自己，“这段代码的执行和我想的一样吗？”，或者，“这段代码是否正确实现了这个算法？”。在测试阶段，问自己，“执行这段测试能让我确信什么吗？”。

参考文献

Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer, 20, 4 (April 1987), pp. 10-19.

原则 73 使用耦合和内聚

USE COUPLING AND COHESION

耦合和内聚是由 Larry Constantine 和 Edward Yourdon 在 20 世纪 70 年代定义的。它们依然是目前所知用来度量软件系统自身可维护性和适应性的最好方法。简单来说，*耦合*，是两个软件组件间相互关联程度的度量。*内聚*，是一个软件组件内功能间相关程度的度量。我们要追求的是低耦合和高内聚。*高耦合*意味着，当我们修改一个组件时，很可能需要修改其他组件。*低内聚*意味着，难以分离出错误原因、或者为满足新需求而要修改的位置。Constantine 和 Yourdon 甚至为我们提供了一个简单易用的方法来度量这两个概念。自 1979 年后，大部分关于软件设计的书都会讲到这些度量方法。学习并使用它们来指导你的设计决策吧。

参考文献

Constantine, L., and E. Yourdon, *Structured Design*, Englewood Cliffs, N.J.: Prentice Hall, 1979.

原则 74 为变化而设计

DESIGN FOR CHANGE

在软件开发中，我们经常会遇到错误、新需求或早期错误沟通导致的问题。所有这些都会导致设计的变化，甚至在设置基线之前（查看相关原则 16）。而且，在对设计设置基线和交付产品后，会出现更多的新需求（查看相关原则 185）。这些都意味着，你必须选择架构、组件和规范技术，以适应重大和不断的变化。

为了适应变化，设计需要做到：

- *模块化*，即应该由独立的部分组成，每一部分可以很容易地升级或替换，而对其他部分造成最小的影响（查看相关原则 65，70，73，80）。
- *可移植性*，即应该很容易修改以适应新的硬件和操作系统。
- *可塑性*，即可以灵活地适应预期外的新需求。
- *最小智力距离*（原则 69）。
- *在智力可控范围内*（原则 70）。
- 这样它就表现出*概念一致*（原则 71）。

参考文献

Witt, B., F. Baker, and E. Merritt, *Software Architecture and Design*, New York: Van Nostrand Reinhold, 1994, Section 1.3.

原则 75 为维护而设计

DESIGN FOR MAINTENANCE

对于非软件产品，设计后的最大成本风险是制造。对于软件产品，设计后的最大成本风险是维护。对于前者，为制造而设计是主要的设计驱动力。不幸的是，为维护而设计并不是软件的标准。它本应该是。

设计者有责任选择最优的软件架构以满足需求。很明显，这个架构是否得体将对系统性能产生深远的影响。不仅如此，架构的选择也对最终产品的可维护性有深远的影响。特别是，从对可维护性的影响来说，架构选择比算法或代码更加重要。

参考文献

Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, 7, 2 (March 1990), pp. 17-25.

原则 76 为防备错误而设计

DESIGN FOR ERRORS

不管你为软件付出多少努力，它都会有错误。你的设计决策应该尽可能做到以下优化：

1. 不引入错误。
2. 引入的错误容易被检测。
3. 部署后软件中遗留的错误要么是不危险的，要么在执行时有补偿措施，这样错误不会造成灾难。

将这种健壮性融入到设计中并不容易。下面是一些有帮助的想法：

1. 不要“省略 case 语句”。比如，如果某个变量有四个可能的值，不要只检查三种情况就假定第四个是剩下的唯一可能值。相反，要设想不可能情况的发生。要检查第四个可能值，并尽早处理错误情况。
2. 要尽可能多地预想“不可能”的情况，并制定恢复策略。
3. 为了减少可能造成灾难的情况，要对可预测的不安全情况进行故障树分析（Fault Tree Analysis，具体可查看 Leveson, N., "Software Safety: What, Why, and How," ACM *Computing Surverys*, 18, 2 (June 1986), pp. 125-163）。

参考文献

Witt, B., F. Baker, and E. Merritt, *Software Architecture and Design*, New York: Van Nostrand Reinhold, 1994, Section 6.4.2.6.

原则 77 在软件中植入通用性

BUILD GENERALITY INTO SOFTWARE

一个软件组件的通用性体现在，它在不同的场景下都可以不做任何修改就能执行预期功能。通用的软件组件要比不太通用的组件更难设计。此外，它们通常执行更慢。不过，这样的组件有以下优点：

1. 在复杂系统中是比较理想的，因为在复杂系统中，一个相似的功能需要在不同的地方被执行。
2. 更可能不经修改就在其他系统中复用。
3. 可以减少组织的维护成本，因为独特或相似的组件数量会减少。

当把一个系统拆分成子组件时，要注意其潜在的通用性。很明显，当多个地方都需要一个相似的功能时，只需构建一个通用功能组件，而非多个相似功能组件。同样，在开发只在一个地方需要的功能时，要尽可能植入通用性，以便日后扩展。

参考文献

Parnas, D., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, 5, 2 (March 1979), pp. 128-138.

原则 78 在软件中植入灵活性

BUILD FLEXIBILITY INTO SOFTWARE

一个软件组件的灵活性体现在，它很容易被修改，以在不同的场景下执行其功能（或者相似功能）。灵活的软件组件比不太灵活的组件更难设计。不过，这样的组件有以下优点：(1) 比通用组件（原则 77）运行时更高效；(2) 相比于不太灵活的组件，在不同的应用场景中更加容易复用。

参考文献

Parnas, D., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, 5, 2 (March 1979), pp. 128-138.

原则 79 使用高效的算法

USE EFFICIENT ALGORITHMS

了解算法复杂度理论是成为一名优秀设计者的绝对前提。给定任何问题，你都可以给出无限多种可选算法来解决它。“算法分析”理论让我们知道，如何区分本来速度就慢的算法（不管编码如何优秀）和速度快几个数量级的算法。关于这个主题有很多优秀的书籍。每一个拥有计算机科学专业的优秀本科院校都会开设这方面的课程。

参考文献

Horowitz, E., and S. Sahni, *Fundamentals of Computer Algorithms*, Potomac, Md.: Computer Science Press, 1978.

原则 80 模块规格说明只提供用户需要的所有信息

MODULE SPECIFICATIONS PROVIDE ALL THE INFORMATION THE USER NEEDS AND NOTHING MORE

设计过程中一个关键部分，是系统中每个软件组件的精确定义。这个规格说明将成为组件“可见”或“公开”的部分。它必须包含用户（这里的“用户”是指，另一个软件组件，或另一个组件的开发者）需要的全部内容，如：用途，名字，调用方法，如何同所在环境通信的细节。任何用户不需要的内容，都要明确排除在外。在大部分情况下，应该排除使用的算法和内部数据结构。因为如果它们是“可见”的，用户可能就会利用这些信息。那么后续的扩展或修改将变得非常困难，因为组件的任何修改都会对所有使用它的组件造成级联效应。可查看关于封装的相关原则 65。

参考文献

Parnas, D., "A Technique for Software Module Specification with Examples," *Communications of the ACM*, 15, 5 (May 1972), pp. 330-336.

原则 81 设计是多维的

DESIGN IS MULTIDIMENSIONAL

在设计一个房子时，建筑设计师需要以多种方式来描述，以方便建筑工人、建筑原材料购买者，以及房屋购买者来充分了解房屋的本质。这些描述方式包括：立面图、平面图、框架、电气图、管道图、门窗细节，以及其它。对于软件设计，也是一样的道理。一份完整的软件设计至少需要包括：

1. *打包方案（Packaging）*。通常用层次图的形式给出，用于说明“什么是什么的一部分？”。它通常隐含说明了数据可见性。它还能体现封装性，如对象内包含的数据和方法。
2. *依赖层次（Needs Hierarchy）*。用于说明“谁需要谁”。以组件网状图的形式表达，其中箭头的指向表明组件间的依赖关系。依赖可能是数据、逻辑或者其它信息。
3. *调用关系（Invocation）*。用于说明“谁调用谁”。以组件网状图的形式表达，其中箭头的指向表明组件间的调用、中断、消息传递关系。
4. *进程组织（Processes）*。一批组件被组织在一起，成为异步处理的进程。这是与其它进程同时运行的组件副本。零个、一个或多个副本可能同时存在。另外，还需要说明进程创建、执行、停止或销毁的条件。

参考文献

Witt, B., F. Baker, and E. Merritt, Software Architecture and Design, New York: Van Nostrand Teinhold, 1994, Section 1.1.

原则 82 优秀的设计出自优秀的设计师

GREAT DESIGNS COME FROM GREAT DESIGNERS

坏设计与好设计的差异，可能源于完善的设计方法、出众的训练、更好的教育或其它因素。无论如何，真正优秀的设计，是真正优秀设计者的智慧结晶。优秀设计的特征是：简洁（Clean）、简单（Simple）、优雅（Elegant）、快速（Fast）、可维护（Maintainable）、易于实现（Easy to Implement）。优秀的设计源于灵感和洞察力，而不仅是努力工作或按部就班的设计方法。对于最好的设计者要重点支持。他们才是未来。

参考文献

Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE computer, 20, 4 (April 1987), pp. 10-19.

原则 83 理解你的应用场景

KNOW YOUR APPLICATION

无论需求文档写得多好，架构和算法的最优选择，主要应基于对应用场景特质的理解。压力下的预期行为，预期的输入频率，响应时间的极限，新硬件的可行性，天气对预期系统性能的影响，等等，这些都是和应用场景相关的。在做架构和算法选型时，需要对此特别考虑。

参考文献

Curtis, B.,H. Krasner, and N. Iscoe, “A Field Study of the Software Design Process for Large Systems”, Communications of the ACM, 31, 11 (November 1988), pp. 1268-1287.

原则 84 无需太多投资，即可实现复用

YOU CAN REUSE WITHOUT A BIG INVESTMENT

要复用软件组件，最有效的方法是：从一个包含精心制作和挑选的组件的代码库开始，这些组件是专门为重用而定制的。然而，这需要大量时间和金钱的投入。通过“废物利用”（Salvaging）技术，短期内实现复用是可能的。简而言之，“废物利用”就是在团队中询问，“你是否曾经实现过具有 X 功能的软件组件？”。如果找到了，就对它进行适配，然后使用。这个方法在长期可能并不有效，但是当前它确实奏效。这样你就没有理由不进行复用了。

参考文献

Incorvaia, A.J. A. Davis, and R. Fairley, “Case Studies in Software Reuse, ”Fourteenth IEEE International Conference on Computer Software and Applications, Washington, D.C.: IEEE Computer Society Pres, 1990, pp. 301-306.

原则 85 “错进错出”是不正确的

“GARBAGE IN, GARBAGE OUT” IS INCORRECT

很多人引用“错进错出”，好像软件这样运行是可以接受的。它是不可以接受的。如果一个用户提供了非法的输入数据，程序应该返回一个好理解的提示，解释为什么这个输入是非法的。如果一个软件组件收到非法的输入数据，不应继续处理，而应给发出错误数据的组件返回一个错误码。这样的思维方式，可以帮助减少软件错误带来的多米诺效应，并且更容易定位错误。因为这样可以尽早捕获错误，并阻止进一步的数据污染。

参考文献

McConnell，S， Code Complete, Redmond, Wash.: Microsoft Press， 1993， Section 5.6.

原则 86 软件可靠性可以通过冗余来实现

SOFTWARE RELIABILITY CAN BE ACHIEVED THROUGH REDUNDANCY

在硬件系统中，高可靠性和可用性（原则 57）经常通过冗余来实现。如果期望一个系统组件的平均故障间隔时间（Mean-Time-Between-Failures）为 x，我们可以生产 2-3 个类似的组件，然后以下面两种方式之一运行：

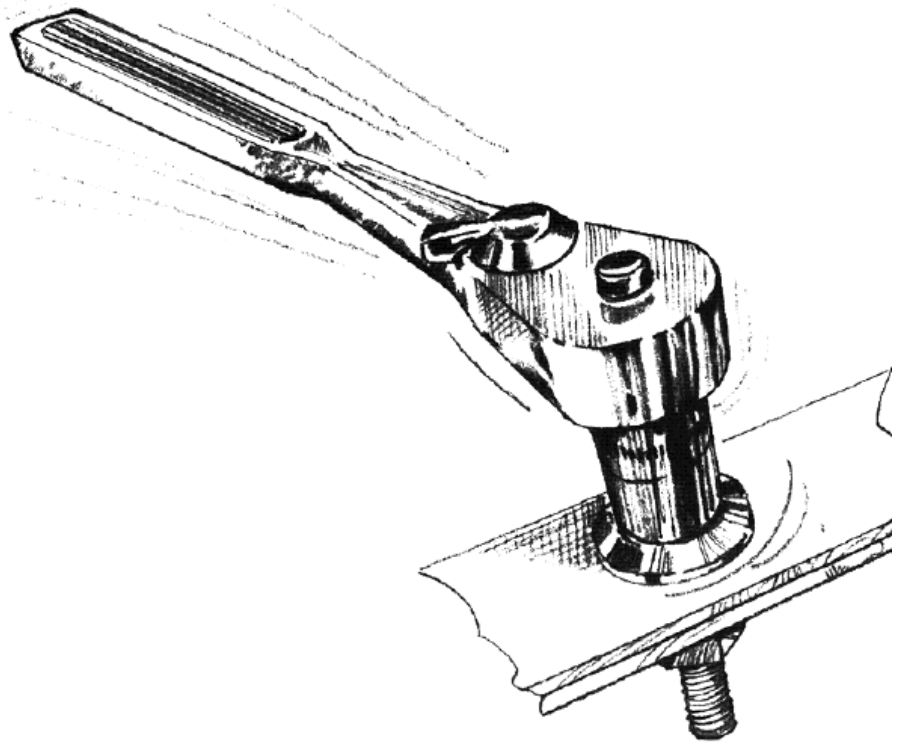
1. *并行方式（Parallel）*。例如，多个组件都执行相同功能，当它们返回结果不同时，关闭其中一个，不会影响到整个系统的功能。
2. *冷备方式（Code Standby）*。仅当在用的计算机硬件出错时，才启用备份的计算机。

用以上方式，设备的制造成本比加倍稍微多一些，设计的成本会少量增长，可靠性则是指数级提升。

在软件系统中，我们不能使用同样的方法。如果我们对相同的软件做两份拷贝，并不会增加软件的可靠性。如果其中一个失败，另外一个也会失败。而可行的方案是：根据相同的需求规格说明，（让两个不同的设计团队）设计出两套软件系统，然后并行部署。用以上方案，开发成本会翻倍，可靠性会指数级提升。需要留意的是，在硬件的例子中，设计成本只有轻微的增长；而在软件的例子中，设计成本（这是软件的主要成本）会翻倍。软件的超高可靠性是非常昂贵的。（原则 4）

参考文献

Musa，J.，A. Iannino, and K. Okumoto，Software Reliability， New York： McGraw-Hill，1987，Section 4.2.2.



第五章 编码原则

CODING PRINCIPLES

*编码*是包含以下行为的集合：

1. 将设计阶段确定的算法转换为用计算机语言编写的程序。
2. 将程序（通常是自动化的）转换为可被计算机直接执行的语言。

编码的主要输出结果就是一组形成文件的程序清单。

原则 87 避免使用特殊技巧

AVOID TRICKS

很多程序员喜欢写带有特殊技巧的程序。这些程序虽然可以执行正确的功能，但是使用了非常晦涩难懂的方式。典型的表现是，他们利用一个函数的副作用来实现一个主要功能。程序员将这些视为“聪明”，但正如艾伦·马克罗（Allen Macro）所指出，他们“通常只是愚蠢地使用了高智商”。

特殊技巧被频繁使用的理由有很多：

1. 程序员都非常聪明，他们想展示这种聪明。
2. 维护人员在最终搞清这些特殊技巧如何生效时，不仅会认识到原来的程序员有多聪明，也会意识到自己有多么聪明。
3. 职业安全感。

底线：避免编写使用特殊技巧的代码，以向世界展示你有多聪明！

参考文献

Macro，A., *Software Engineering: Concepts and Management*, Englewood Cliffs, N.J.: Prentice-Hall International, 1990, p. 247.

原则 88 避免使用全局变量

AVOID GLOBAL VARIABLES

全局变量会使程序编写很方便。毕竟，如果你想访问或者修改变量 *x*，直接操作就可以。不幸的是，如果访问变量 *x* 时发现取值不正常（如，-16.3 艘船），很难确定是哪个模块出了问题。“全局”意味着，任何人都可能错误的修改它的值。

作为替代方案，可以将重要数据封装在对应模块中（原则 65），这样任何人都必须通过指定方式来访问或者修改它。此外，可以显式传递参数给需要特定数据的程序。如果发现参数过多，那么可能你的设计需要改造。

参考文献

Ledgard, H., *Programming Practice*, Vol. II, Reading, Mass.: Addison-Wesley, 1987, Chap. 4.

原则 89 编写可自上而下阅读的程序

WRITE TO READ TOP-DOWN

人们通常阅读程序代码都是从上（即，第一行）到下（即，最后一行）。要编写有助于读者理解的程序。

本原则的含义包括：

1. 顶部要包含详细的对外说明，用以明确定义程序的目的与用途。
2. 顶部要说明外部可访问的方式、局部变量和算法。
3. 使用被称为“结构化”的编程结构，这从本质上更易于遵循。

参考文献

Kernighan, B., and P. Plauger, *The Elements of Programming Style*, New York: McGraw-Hill, 1978, pp. 20-37.

原则 90 避免副作用

AVOID SIDE-EFFECTS

程序的*副作用*，是指程序的某些操作不是其主要目的，并且这些操作对程序外部可见（或其结果能被外部感知）。副作用是软件中许多细微错误的来源。即，这些错误是潜伏最深的，一旦它们的症状表现出来是最难排查的。

参考文献

Ledgard, H., *Programming Proverbs*, Rochelle Park, N.J.: Hayden Book Company, 1975, Proverb 8.

原则 91 使用有意义的命名

USE MEANINGFUL NAMES

一些程序员坚持使用诸如 N_FLT 或更糟的名称（如 F）进行变量命名。通常他们的说法是：这样可以使程序员更高效，因为更少的键盘操作。优秀的程序员应该只花很小比例的时间敲代码（或许 10%-15%），大部分时间应该花在思考上。所以，实际上真的能节省多少时间呢？不过，还有一种更好的论点：过短的命名实际上会降低效率。原因有两个：（1）测试和维护成本将提高，因为要花更多时间去尝试理解这些命名；（2）当使用短命名时，有可能要花更多的时间敲代码。第二个原因成立，因为短命名需要更多的注释。例如：

```
        N_FLT = N_FLT + 1
```

需要加一行注释“LOOK AT NEXT FLIGHT”（按键 32 次），但是

```
        NEXT_FLIGHT = PREVIOUS_FLIGHT + 1
```

就不需要增加注释（按键 29 次）。

参考文献

Ledgard, H., *Programming Proverbs*, Rochelle Park, N.J.: Hayden Book Company, 1975, pp. 94-98.

译者注

“LOOK AT NEXT FLIGHT”，中文意思是“获得下一个航班号”。为了便于读者理解原文的意思（即按键次数的差异），在正文部分保持了英文原文。

原则 92 程序首先是写给人看的

WRITE PROGRAMS FOR PEOPLE FIRST

在计算机时代早期，计算机处理速度相对较慢。任何能够减少一些指令操作的事情都值得去做。在非常昂贵的计算机系统上，最有效地利用资源是主要目标。如今形势发生了变化。现在最有价值的资源是人力：开发软件的人力，维护软件的人力和提高软件能力的人力。除了非常少数的例外场景，程序员应该首先考虑的是，后续需要尝试理解和适配软件的人员。任何能够帮助他们的事情都应该去做（原则 87 到 91 会提供一些帮助）。执行效率也很重要（参考原则 63、79、94），但它们并不是互斥的。如果你需要执行效率，这没问题，但要*提升*程序的可读性，以免在提升执行效率的过程中浪费人力。

参考文献

McConnell, S., *Code Complete*, Redmond, Wash.: Microsoft Press, 1993, Section 32.3.

原则 93 使用最优的数据结构

USE OPTIMAL DATA STRUCTURES

数据的结构，与处理该数据的程序的结构，是密切相关的。如果你选择了正确的数据结构，算法（以及代码）将变得易于编写、阅读以及维护。要去阅读任何关于算法或者数据结构的书（它们是一致和相同的！）。

当你准备编写程序时，应该将算法和数据结构一起考虑。在选择最佳组合之前，请尝试两个或三个或更多不同的组合。应确保将数据结构封装在一个组件内（原则 65），这样当发现更好的数据结构时，可以轻松地修改。

参考文献

Kernighan, B., and P. Plauger, *The Elements of Programming Style*, New York: McGraw-Hill, 1988, pp. 52, 67.

原则 94 先确保正确，再提升性能

GET IT RIGHT BEFORE YOU MAKE IT FASTER

提升正常运行程序的性能，比“让高性能程序正常运行”容易很多。当你进行初始编码时，不要担心优化问题。【另一方面，请勿使用效率低下的算法或数据结构（原则 79 和 93）。】

每个软件项目都有很大的进度压力。有些项目可能在早期阶段压力不大，但后期会加快步伐。在这种情况下，在任何时候一个组件要是能够按时（或者提前）完成并且可靠运行，应值得庆祝。要努力让软件项目成为庆祝的理由，而不是失望的原因。如果你让程序能够正常运行（即使运行缓慢一点），团队中的每个人都会赞赏。参考相关的原则 34。

参考文献

Kernighan, B., and P. Plauger, *The Elements of Programming Style*, New York: McGraw-Hill, 1978, pp. 124-134.

原则 95 在写完代码之前写注释

COMMENT BEFORE YOU FINALIZE YOUR CODE

我经常听程序员说，“为什么我现在要找麻烦为我的代码写注释？代码是会改变的！”。我们写代码注释是为了让软件更易于调试、测试以及维护。在写代码的同时写注释（或者提前写注释，参见原则 96），这会让你更容易调试软件。

当你调试软件时，无疑会发现一些错误。如果从算法到代码的转换过程存在错误，那么你只需要修改代码，而不需要修改注释。如果算法存在错误，那么你对注释和代码都需要修改。但如果不写代码注释，你怎么能发现算法的错误呢？

参考文献

Kernighan, B., and P. Plauger, *The Elements of Programming Style*, New York: McGraw-Hill, 1978, pp. 141-144.

原则 96 先写文档后写代码

DOCUMENT BEFORE YOU START CODING

这个建议对一些读者而言或许有些奇怪，但当实践一段时间之后，它会被视为理所当然。第 95 条原则解释了为什么你该在写完代码前加以注释。第 96 条原则更进一步：在开始写代码之前，你就该这么做！

在为一个组件完成详细设计（即，将它的外部接口和算法写为文档）之后，在代码中编写行间注释。这些注释大部分与前面完成的接口与算法的文档没什么不同。让这些注释通过编译，确保没有低级错误的产生（比如漏掉了注释分隔符）。之后将每行注释转化为与之对应的代码片段。（注意：如果最后发现每条注释只对应一行代码，你很可能对算法描述的过于细致了。）你会发现调试过程变得顺畅许多。

参考文献

McConnell, S., Code Complete, Redmond, Wash.: Microsoft Press, 1993, Sections 4.2-4.4.

原则 97 手动运行每个组件

HAND-EXECUTE EVERY COMPONENT

手工执行一些简单的测试用例，一个软件组件或许会花 30 分钟时间。一定要做这件事！我这样建议，是补充、而不是代替现存的那些更深入和完整的、程序化的单元测试。有多大成本？ 就 30 分钟而已。如果不这么做？现在节省 30 分钟，直接去做单元测试、集成测试和系统测试。一旦系统挂了，将花费 3-4 人天的成本去定位失败的原因。假设有 6 个组件被筛选出来作为嫌疑对象。每个组件都要由它的开发者做深入的检查。然后对每个组件花 30 分钟，手工执行一些简单的测试用例。总之，30 分钟比 3-4 人天加上 6×30 分钟的成本要少。

参考文献

Ledgard, H., Programming Proverbs, Rochelle Park, N.J.: Hayden Book Company, 1975, Proverb 21.

原则 98 代码审查

INSPECT CODE

软件的详细设计评审和代码审查，由 Michael Fagan 首次提出，论文标题为“用设计和代码审查减少程序中的错误”（"Design and Code Inspections to Reduce Errors in Program Development"，IBM Systems Journal, 15, 3 (July 1976), pp. 182-211）。由此发现的错误，能占到所有被发现的软件错误的 82%。对于发现错误，代码审查比测试要好得多。定义完成审查的标准。记录追踪在代码审查中发现的各类问题。Fagan 提出的代码审查方法，大约会消耗 15% 的研发资源，可以带来总开发成本净减少 25%-30%。

你最初的项目排期就应该考虑到评审（及修正）每个组件的时间。你或许认为这对你的项目来说过于“奢侈”。 然而，你不该将评审视为一种奢侈。数据显示，你甚至可以减少 50% 至 90% 的测试时间。如果这都不够激励，我不知道还能做什么。顺便说一下，关于如何做好代码审查，在参考书籍中有大量的数据支撑和建议。

参考文献

Grady, R., and T. VanSlack, "Key lessons in Achieving Widespread Inspection Use" IEEE Software, 11, 4 (July 1994), pp. 46-57.

译者注

Code Inspect，即目前常说的 Code Review（代码审查）。所以在中文翻译中，提法都统一改为“代码审查”。

原则 99 你可以使用非结构化的语言

YOU CAN USE UNSTRUCTURED LANGUAGES

非结构化的代码打破了 Edsger Dijkstra 的建议，其要求对控制结构限制在 IF-THEN-ELSE，DO-WHILE，DO-UNTIL 和 CASE 几类。注意，使用一种没有这些控制结构的语言（如，汇编语言），也可以写出结构化的代码。可以在代码中增加结构化控制的注释，并限制 GOTO 只能用来实现这些控制结构。

为此，要首先使用前面的控制结构编写算法。然后，将它们转换为行内注释。接下来，将注释转换为等效的编程语言语句。GOTO 语句会被使用到，但它们将实现更好的控制结构，并且将促进而不是妨碍可读性、可维护性和可证明性。对于某些读者来说，这种建议似乎有些奇怪，但是经过一段时间的实践，它会变得很自然。

译者注

目前似乎只有在使用汇编语言的场景，才有可能用到这条原则。目前使用的绝大多数编程语言，都已经是结构化编程语言。

原则 100 结构化的代码，未必是好的代码

STRUCTURED CODE IS NOT NECESSARILY GOOD CODE

由 Edsger Dijkstra 提出的*结构化编程*的最初定义是为了便于程序证明。他推荐的结构（IF-THEN-ELSE，DO-WHILE，等）现在已经非常普遍（尽管还没有程序证明），以至于它们的使用现在被称为“编程”，而不是“结构化编程”。 但需要注意的是，并非所有的“结构化”程序都是好的。一个人可以写出异常晦涩的程序，虽然它是结构化的。对高质量的程序，结构几乎是必要条件，但远不是充分条件。

参考文献

Yourdon, E., How to Manage Structured Programming, New York: Yourdon, inc., 1976, Section 5.2.2.

原则 101 不要嵌套太深

DON'T NEST TOO DEEP

嵌套 IF-THEN-ELSE 语句大大简化了编程逻辑。但另一方面，嵌套超过三层会严重降低可理解性。人类的头脑在变得混乱之前只能记住一定数量的逻辑。有很多简单的技巧可以用来减少嵌套。有关示例和技术，请参阅以下参考资料。

参考文献

McConnell, S., Code Complete, Redmond, Wash.: Microsoft Press, 1993, Section 17.4.

原则 102 使用合适的语言

USE APPROPRIATE LANGUAGES

在帮助你完成工作的能力方面，编程语言之间差异很大。特定项目或产品目标通常会指定合适的语言。下面这些只是指南，而不是永恒真理。

如果首要目标是可移植性，那么使用已被证明具有高度可移植性的语言（如 C、FORTRAN 或 COBOL）。如果首要目标是快速开发，那么就使用有助于快速开发的语言（4GL，Basic，APL，C，C++，或 SNOBOL）。如果首要目标是低维护成本，那么使用具有许多内置、高质量特性的语言（如 Ada 或 Eiffel）。如果程序需要大量使用字符串或复杂的数据结构，请选择支持它们的语言。如果产品必须由已有的一组懂得 X 语言的维护人员来维护，那么就使用 X 语言。最后，如果客户说“你应该用 Y 语言”，那么就用 Y 语言，否则你就做不成生意了。

参考文献

McConnell, S., Code Complete, Redmond, Wash.: Microsoft Press, 1993, Section 3.5.

译者注

这里提到了很多现在可能已经不常用的编程语言（FORTRAN，COBOL，4GL，Basic，APL，SNOBOL，Ada，Eiffel）。编程语言确实已经有了很大的发展。读者只需要理解和运用作者的思想就好，可以根据情况选择现在合适的编程语言。

原则 103 编程语言不是借口

PROGRAMMING LANGUAGES IS NOT AN EXCUSE

有些项目被迫使用不太理想的编程语言。这可能是由于希望降低维护成本（“我们所有的维护人员都懂 COBOL”）、快速编程（“我们用 C 的开发效率最高”）、确保高可靠性（“Ada 程序最能减少崩溃”）或实现高执行速度（“我们的程序对实时性要求很高，需要使用汇编语言”）。使用任何语言都能写出高质量程序。事实上，如果你是一个好的程序员，对任何一种编程语言你都应该是个好程序员（原则 104）；尽管不太理想的编程语言可能会让你的工作困难一些。

参考文献

Yourdon, E., How to Manage Structured Programming, New York: Yourdon, Inc., 1976, Section 5.2.5.

原则 104 编程语言的知识没那么重要

LANGUAGE KNOWLEDGE IS NOT SO IMPORTANT

不管使用哪种语言，优秀的程序员都是优秀的。不管使用哪种语言，糟糕的程序员仍然是糟糕的。不可能有一个人是"优秀的 C 程序员"，同时是"糟糕的 Ada 程序员"。如果他确实在 Ada 语言上表现的很糟糕，那大概率也不会在 C 语言上表现很好！除此之外，一个真正优秀的程序员应该可以很容易的学会一种新语言。这是因为一个真正优秀的程序员理解和赞赏高质量编程的概念，而不只是某些编程语言的语法和语义特性。

所以，为一个项目选择语言的首要驱动力应该是什么语言更合适（原则 102），而不是程序员都在抱怨“我们只知道 C 语言”。如果由于项目选择了其他语言而导致一些人退出，这个项目很可能会更好！

参考文献

Boehm, B., Software Engineering Economics, Englewood Cliffs, N.J.: Prentice Hall, 1981, Section 26.5.

原则 105 格式化你的代码

FORMAT YOUR PROGRAMS

使用标准的缩进规则，可大大提高程序的可读性。选择遵循哪种规则无关紧要，但一旦选择了，就要维持一致。

我遵循的规则是：让 THEN 和 ELSE 正位于对应的 IF 的下方，END 正位于对应的 BEGIN 或者 DO 下方，等等。举例来说：

```
IF___
THEN BEGIN
     ___________
     ___________
     END
ELSE IF _______
     THEN _______
     ELSE _______
DO WHILE ()
     ______
     ______
END DO;
```

更多的例子见参考信息。顺便说一下，唯一比不一致的缩进更糟糕的，是不正确的缩进（例如把 ELSE 和错误的 IF 或者 THEN 对齐）！为了避免意外的错误对齐，要使用市面上能找到的好打印机。

参考文献

McConnell, S., Code Complete, Redmond, Wash.: Microsoft Press, 1993, Chapter18.

译者注

具体代码的缩进方式，请遵循团队内的编程规范，或遵循行业内的主流规范。

原则 106 不要太早编码

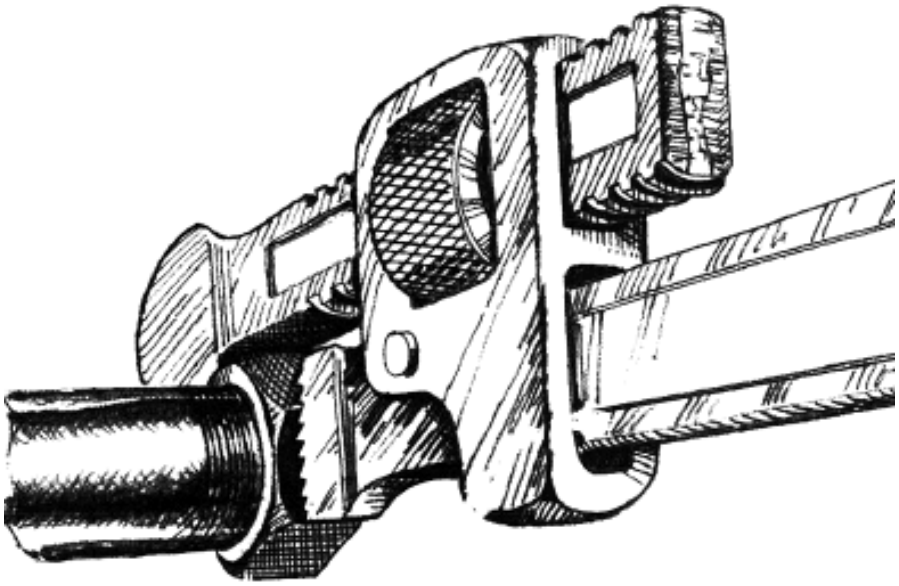
DON'T CODE TOO SOON

编写软件和盖房子类似。这两者都需要很多准备工作。没有坚固稳定的混凝土地基，盖房子不会成功。没有坚固稳定的需求和设计作为基础，编码也不会成功。想一想当地基已经浇筑完成之后，对房子做修改有多么困难！

不要因为管理层想看到“进展”，就被迫过早编写代码。在设立基线前，要确认需求和设计是正确且合适的，在对最终产品编码前更要确认。附带说一下，不要从这个原则推断出原型试验的方法有问题（原则 5，10，11，12，13）。在需求基线完成很早之前，试验性编码没有错。只是不要认为这是最终的产品。针对本条原则，Manny Lehman 提出了一个相反的观点：不要太晚编码！

参考文献

Berzins, V., and Luqi, Software Engineering with Abstractions, Reading, Mass.: Addison-Wesley, 1991, Section 1.5.



第六章 测试原则

TESTING PRINCIPLES

测试是包含以下行为的集合：

1. 对独立的软件组件执行测试（即：单元测试，Unit Testing），以确保其行为与组件设计规格说明中的定义足够的接近。
2. 对执行过单元测试的组件集合执行测试（即：集成测试，Integration Testing），以确保这些组件一起工作时的行为足够接近设计中的说明。
3. 对集成测试过的所有组件进行测试（即：软件系统级测试，Software Systems-level Testing），以确保它们可以作为一个系统来运行，且行为足够接近软件需求规格说明中的定义。
4. 制定软件系统级测试的测试计划。
5. 制定软件集成测试的测试计划。
6. 制定单元测试的测试计划。
7. 建立测试装置（test harness）和测试环境（test environment）。

原则 107 依据需求跟踪测试

TRACE TESTS TO REQUIREMENTS

理解哪些测试可以验证哪些需求是很重要的。有如下两个原因:（1）在生成测试时，你会发现，了解是否所有需求都在被测试是很有用的。（2）在执行测试时，你会发现，了解正在验证哪些需求是很有用的。此外，如果你的需求已经排了优先级（原则 50），可以很容易得出测试的相对优先级；也就是说，一个测试的优先级是其所有对应需求的优先级的最大值。

维护一个大二进制表，其中行对应于所有软件测试，列对应于软件需求规格说明中的每个需求。任何位置的 1 表示此测试有助于验证此需求。注意，一整行都没有被置 1 表示此测试没有目的，一整列都没有被置 1 表示该需求漏测。能够成功地创建这样一个表，取决于你唯一地引用每个需求的能力（原则 52）。

参考文献

Lindstrom, D., "Five Ways to Destroy a Development Project," IEEE Software, 10, 5 (September 1992), pp. 55-58.

原则 108 在测试之前早做测试计划

PLAN TEST LONG BEFORE IT IS TIME TO TEST

通常，软件开发人员会先创建他们的软件产品，然后挠头说，“现在我们要如何测试这个玩意呢?”。 测试计划是一项重要的任务，必须与产品开发同时进行，以便同步完成测试计划和初始（即，预测试）开发活动。

对于软件系统测试，测试计划人员应在设定需求基线之前，从可测试性的角度对软件需求规格说明进行评审，并向需求编写者提供反馈。在设定需求基线后不久，应开始认真的测试开发。对于集成测试，测试计划人员应在对初步设计确定基线之前对其进行评审。他们还应该向项目经理和设计人员提供以下反馈:（1）合理的资源分配，以确保“正确的”组件（从测试的角度来看）以正确的顺序生产；（2）对设计的修改，以使设计本质上更容易测试。对初步设计确定基线后不久，应开始认真的集成测试开发。对于单元测试，可以在详细设计完成后立即开始制定单元测试计划。

参考文献

Goodenough, J., and S. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on software Engineering. 1, 2 Oune 1975), pp. 156-173, Section IIIC.

原则 109 不要测试自己开发的软件

DON'T TEST YOUR OWN SOFTWARE

软件开发人员永远不应成为自己软件的主要测试者。开发人员比较适合进行初始调试（译者注：自测）和单元测试。 [相反的观点，可参见 Mills, H., et al., "Cleanroom Software Engineering"， in IEEE Software, 4, 5 (September 1987), pp. 19-25.] 在以下场景，独立的测试人员是必要的：

1. 在开始集成测试之前，检查单元测试是否足够
2. 所有集成测试
3. 所有软件系统测试

在测试期间，正确的态度是希望暴露 BUG 。开发人员怎么可能接受这种态度呢？如果测试人员带着不想发现 BUG 的偏见，测试将变得更加困难。

参考文献

Myers, G., The Art of Software Testing, New York: John Wiley & Sons, 1979, p. 14.

译者注

目前确实有新的倾向，由程序员来测试自己的代码。但本原则提到的因素依然值得考虑。一个能够充分对自己代码进行测试的程序员，需要能够把自己的视角切换到一个测试人员，并且有发现 bug 的足够欲望。

原则 110 不要为自己的软件做测试计划

DON'T WRITE YOUR OWN TEST PLANS

你不仅不应该测试自己的软件（原则 109），而且也不应该负责为软件生成测试数据、测试方案或测试计划。如果你负责了，那么你可能会在测试生成中犯与软件创建中相同的错误。例如，如果你在设计软件时对合法输入的范围做了一个错误的假设，那么在生成测试计划时，你很可能会做出同样的假设。

如果你是一名程序员或者设计人员，而你的经理要求你编写测试计划，我建议你将测试计划生成的职责交给其他程序员或设计人员。如果你是需求工程团队的成员，同时还负责系统测试的生成，那么我建议将你的团队成员职责细分，以免任何人对他自己编写的需求生成测试。

参考文献

Lehman, M., private communication, Colorado Springs, Col.: Oanuary 24, 1994).

译者注

请参见原则 109 的译者注。

原则 111 测试只能揭示缺陷的存在

TESTING EXPOSES PRESENCE OF FLAWS

无论多么彻底和深入，测试只能揭示程序中缺陷的存在，而并不能确保程序没有缺陷。它可以增加你对程序正确性的信心，但它不能证明程序的正确性。为了获得真正的正确性，必须使用完全不同的方法，即正确性证明。

参考文献

Dijkstra, E., "Notes on Structured Programming," in Structured Programming, Dahl, 0., et al., eds., New York: Academic Press, 1972.

原则 112 虽然大量的错误可证明毫无价值，但是零错误并不能说明软件的价值

THOUGH COPIOUS ERRORS GUARANTEE WORTHLESSNESS, ZERO ERROR SAYS NOTHING ABOUT THE VALUE OF SOFTWARE

这是杰拉尔德·温伯格（Gerald Weinberg）的“无差错谬论”（Absence of Errors Fallacy）。它真正地将测试纳入了视野。它还将所有的软件工程和管理纳入视野。本原则的第一部分显然是正确的，有很多错误的软件是没用的。第二部分则发人深省。它表达的是：无论你多么努力地消除错误，除非你在开发正确的系统，否则你都是在浪费时间。Akao 的《质量功能部署》（Quality Function Deployment, Cambridge, Mass.: Productivity Press, 1990）详细介绍了一种方法，用于确保你在整个软件生命周期中开发正确的系统。本原则的一个推论是，如果你在开发错误的系统，那么世界上所有的形式化方法、所有的测试和所有的产品保证都将于事无补。

参考文献

Weinberg, G., Quality Software Management, Vol. 1: Systems Thinking, New York: Dorset House, 1992, Section 12.1.2.

译者注

[1] 杰拉尔德·温伯格（Gerald Weinberg），美国杰出的专业作家和思想家，其主题主要集中在两个方面：人与技术的结合；人的思维模式、思维习惯以及解决问题的方法（引自百度百科）。

[2] Akao，即 Yoji Akao （赤尾 洋二），是一名日本规划专家。详细介绍，见 https://en.wikipedia.org/wiki/Yoji_Akao

原则 113 成功的测试应发现错误

A SUCCESSFUL TEST FINDS AN ERROR

我经常听到测试人员兴高采烈地宣布，“好消息!我的测试成功了，程序运行正常”。 这是运行测试时的错误态度。[它也支持程序员永远不要测试他们自己的软件的观点（原则 109）] 一种更有建设性的态度是，通过测试来发现错误。因此，成功的测试是能够发现错误的测试。以医学测试为例，看看类似的情况。假设你感觉生病了，医生把你的血样送到实验室。几天后，医生打电话告诉你，“好消息！你的血液正常”。 这不是什么好消息。你病了，否则你不会去看医生的。一次成功的血液检测应该报告你到底有什么问题。软件是有缺陷存在的（不然你不会测试它）。一个成功的测试会报告这些错误是如何表现出来的。

在生成测试计划时，你应该根据发现错误的可能性来选择测试（译者注：优先执行容易发现错误的用例）。在测试软件时，衡量测试小组的工作，应该根据多么善于发现错误，而不是根据多么拙于发现错误。

参考文献

Goodenough, J., and S. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, 1, 2 (June 1975), pp. 156-173.

原则 114 半数的错误出现在 15% 的模块中

HALF THE ERRORS FOUND IN 15 PERCENT OF MODULES

保守估算，在大型系统中，大约所有软件错误的半数出现在 15% 的模块中，80% 的软件错误出现在 50% 的模块中。Gary Okimoto 和 Gerald Weinberg 的结论更引人注目，所有错误的 80% 是在仅仅 2% 的模块中发现的（参见 Weinberg 的《质量软件管理》： Quality Software Managetnent, Vol. 1: Systems Thinking, New York: Dorset House, 1992）。因此，在测试软件时，你可以这样认为，在发现错误的地方，很可能会发现更多错误。

要维护日志（译者注：指测试日志），不仅记录在项目的每个时间段内发现了多少错误，还要记录每个模块发现了多少错误。当历史表明一个模块非常容易出错时，你最好从头开始重写它，强调简单性（原则 67），而不是聪明。

参考文献

Endres, A., "An Analysis of Errors and Their Causes in System Programs," IEEE Transactions on Software Engineering, 1, 2 (June 1975), pp. 140-149.

原则 115 使用黑盒测试和白盒测试

USE BLACK-BOX AND WHITE-BOX TESTING

黑盒测试使用组件外部行为的定义作为其唯一输入。必须要确定，是否软件做了应做的事情，以及没有做不应该做的事情。白盒测试使用代码本身来生成测试用例。这样白盒测试可能会要求，例如，长度为 50 条指令或更少的程序的所有路径都必须被覆盖（原则 122）。但是请注意，即使同时使用黑盒测试和白盒测试，测试也只能利用输入域中的很小一部分可能数据值（原则 111）。

为了说明黑盒测试和白盒测试是如何相辅相成的，让我们看一个示例。假设程序的定义规定，应该打印输入列表中所有数字的总和。程序完成后，它会等待一个 213 的输入，如果出现了，则把总和设为 0。由于这不在程序的定义中，除偶然情况（即，碰巧选择一个包含 213 的随机测试用例）之外，无法通过黑盒测试找到这个错误。白盒测试将要求对执行路径进行更充分的测试，因此很可能会检测到“213”的情况。通过组合使用黑盒和白盒，你可以最大化测试的效果。两者各自都不能做到全面的测试。

参考文献

Dunn, R., Software Defect Removal, New York: McGraw-hill, 1984, Section 7.4.

原则 116 测试用例应包含期望的结果

A TEST CASE INCLUDES EXPECTED RESULTS

一个测试用例的文档必须包含期望的正确结果的详细描述。如果这点被忽略，测试者无法判断软件成功还是失败。而且，测试者可能会把一个错误的结果评估为正确的，因为潜意识里总是希望看到正确结果。更坏的情况是，测试者可能会把正确的结果评估为错误的，导致设计者和开发者一阵忙乱就为了“修复”正确的代码。

要为测试计划设定组织标准，其中应要求对测试用例期望的中间结果和最终结果进行文档说明。质量保证团队应该确认所有测试计划遵从这个标准。

参考文献

Myers, G., The Art of Software Testing, New York: John Wiley & Sons, 1979, p. 12.

原则 117 测试不正确的输入

TEST INVALID INPUTS

为尽可能多的可接受的输入情况生成测试用例，是自然和常见的做法。同样重要但是不太常见的是，为所有不正确或者非期望的输入生成大量的测试用例。

举个简单的例子，假设我们要写一个程序来为 0 到 100 范围内的整型列表排序。测试的列表应该包含：一些负数、全部相等的数字、一些非整型的数字、一些字符数据、一些空的记录，等等。

参考文献

Myers, G., The Art of Software Testing, New York: John Wiley & Sons, 1979, p. 14.

原则 118 压力测试必不可少

ALWAYS STRESS TEST

当面对“正常”负载的输入或刺激时，软件设计通常表现得很好。对软件的真正测试是，在面对剧烈的负载时，它是否可以保持正常运行。这些剧烈的负载通常在需求文档中被说明为“最多 x 个同时运行的组件（widget）”或者“每个小时最多 x 个新的组件产生”。

如果需求文档规定了软件每小时能够处理最多 x 个组件，那么你必须验证软件能做到。事实上，你不仅应该测试让它处理 x 个组件，你还应该让它处理 x+1 或者 x+2（或者更多）个组件，看看会发生什么（原则 58）。毕竟系统不能控制它所处的环境，而你不会希望，在环境以意想不到的方式“失灵”的时候软件会崩溃。

参考文献

Myers, G., The Art of Software Testing, New York: John Wiley & Sons, 1979, pp. 113-114.

原则 119 大爆炸理论不适用

THE BIG BANG THEORY DOES NOT APPLY

在一个项目接近交付期限，而软件还没有准备好的时候，往往充满绝望情绪。假设排期要求两个月的单元测试时间，两个月的集成测试时间，以及两个月的系统测试时间。现在距离计划的交付日期还有一个月。假设百分之五十的组件已经完成了单元测试。通过简单的计算可以得知，你落后进度五个月。你有两个选择：

1. 向你的客户承认五个月的延误：请求推迟交付。
2. 把所有组件集成到一起（包括 50% 尚未进行单元测试的组件），期盼有好的结果。

在第一种情况中，你可能是在过早地承认失败。在你的经理眼中，你可能在竭尽全力解决问题之前就放弃了。在第二种情况中，可能存在 0.001% 的机会，当你把所有组件集成在一起时，它能够正常运行并如期交付。项目经理往往屈服于后者，因为这看起来似乎他们在承认失败之前竭尽全力。不幸的是，这很可能会让你的排期再延长六个月。你不能通过忽略单元测试和集成测试来节省时间。

参考文献

Weinberg, G. Quality Software Management, Vol. 1: System Thinking, New York: Dorset House, 1992, Section 13.2.3.

原则 120 使用 McCabe 复杂度指标

USE MCCABE COMPLEXITY MEASURE

虽然有很多度量方法可以用来报告软件的内在复杂度，但是没有一个像 Tom McCabe 用于衡量测试复杂度的圈数法那样直观和易用。虽然不是绝对可靠，但是它可以相对一致地预测测试难度。只要为你的程序画一个图，其中节点对应连续的指令序列，边对应非连续的控制流。McCabe 指标通过简单计算 e-n+2p 获得 。其中 e 是边的数量，n 是节点的数量，而 p 是你要检查的独立图个数（一般都是 1）。这个指标还可以用曲奇刀的类比来计算：想象下把形状像你程序图的曲奇刀按压到铺开的面团上。产生的曲奇数量（图中区域的数量）就是 e-n+2p 。这么简单的方法，没有理由不使用它吧。

对每一个模块使用 McCabe 方法来帮助评估单元测试的复杂度。另外，把它用在集成测试这个级别上，每个过程是一个节点，每个调用路径是一条边，这样可以帮助评估集成测试的复杂度。

参考文献

McCabe, T., "A Complexity Measure", IEEE Transactions on Software Engineering, 2, 12 (December 1976), pp. 308-320.

译者注

[1] 本原则中提到的方法，常被称为“圈复杂度”。在一些代码检查工具中，提供了圈复杂度的检查能力。可以进一步查看百度百科中的说明。

[2] McCabe 先生是始创于 1977 年的美国 McCabe 公司的创始人，他同时也是一位数学家。他在业界第一个提出了软件度量方法，在如何持续改进软件质量方面提出了处于领导地位的方法论，从而享誉世界。(来自百度百科)

原则 121 使用有效的测试完成度标准

USE EFFECTIVE TEST COMPLETION MEASURES

很多项目都在时间耗尽的时候宣布测试结束。这么做从政治角度来说是有意义的，但却是不负责任的。在做测试计划的时候，应该定义一个标准，以决定测试什么时候完成。如果在时间耗尽时没有达到目标，你仍然可以选择是交付产品还是错过里程碑，但至少你要清楚自己是否交付了一个高质量的产品。

有效度量测试进度的两个想法是：

1. 每周发现新错误的比率。
2. 暗中在软件中埋下已知的 bug （ Tom Gilb 管这个叫 bebugging ）后，这些 bug 到目前为止被发现的百分比。

对于测试进度的一个无效指标是测试用例通过的百分比（当然除非你确定测试用例很好地覆盖了需求）。

参考文献

Dunn, R., Software Defect Removal, New York: McGraw-Hill, 1094, Section 10.3.

原则 122 达成有效的测试覆盖

ACHIEVE EFFECTIVE TEST COVERAGE

尽管事实是测试不能证明正确性，但是做一个全面的测试还是很重要的。在测试计划生成或测试执行阶段，有一些指标可以用来确定代码执行测试的全面程度。这些指标易于使用，并且有工具用来监控测试覆盖水平。一些例子包括：

1. 行覆盖率，用于衡量至少执行一次的语句的百分比。
2. 分支覆盖率，用于衡量程序中被执行的分支的百分比。
3. 路径覆盖率，用于衡量所有可能路径（通常是无限的）覆盖程度。

需要记住的是，虽然“有效”覆盖比零覆盖要好，但别自欺欺人地认为程序在任何定义下都是“正确的”（原则 111）。

参考文献

Dunn, R., Software Defect Removal, New York: McGraw-Hill, 1094, Section 10.3.

原则 123 不要在单元测试之前集成

DON'T INTEGRATE BEFORE UNIT TESTING

正常情况下，各个组件是分别进行单元测试的。当它们通过各自的单元测试之后，一个单独的团队将它们集成到有意义的集合中，用以测试它们的接口。没有单独完成单测的组件常常会被集成到子系统中，以徒劳地追赶落后的进度。这些尝试实际上会造成更多的进度延后。这是因为，子系统无法满足集成测试计划，可能是由于接口的错误，也可能是由于一个事先没经过测试的组件的错误。而很多时间要被花费在确定哪一个是真正的原因。

如果你在管理一个项目，你可以做很多事来避免这种情况。首先也是最重要的是，尽早制定一个集成测试的计划（比如，在高层设计完成之后不久）。这个计划应该明确哪些组件是最重要的，应该最先被集成，以及组件应该以什么顺序被集成。一旦你把这些写下来，要分配适合的资源给高优先级的组件进行编码和单元测试，以确保集成测试者不用花费过多的时间空闲等待。其次，当发现集成测试的重要组件无法在需要的时候达到可用时，就让集成测试人员开始开发临时脚手架软件来模拟缺失的组件。

参考文献

Dunn, R., Software Defect Removal, New York: McGraw-Hill, 1094, Section 10.3.

原则 124 测量你的软件

INSTRUMENT YOUR SOFTWARE

测试软件的时候，往往很难确定为何软件会失败。一个发现原因的方法是测量你的软件，也就是，嵌入特殊的指令到软件中，来报告执行轨迹、异常状况、过程调用等等。当然，如果你的调试系统提供了这类能力，就不要手动测量了。

参考文献

Huang, J., "Program Instrumentation and Software Testing," IEEE Computer, 11, 4 (April 1978), pp. 25-32.

原则 125 分析错误的原因

ANALYZE CAUSES FOR ERRORS

错误在软件中是很常见的。我们会花费大量的资源来发现和修复它们。从一开始就防止它们的发生，从而降低它们的影响，是更划算的。为此的一个方法是，当检测到错误的时候，分析它们的原因。错误的原因应该被告知给所有开发者，这么做是基于一个理念：如果一类错误的原因我们已经全面地分析和研究过了，我们就不那么容易犯下同类的错误了。

当一个错误被发现时，有两件事要做：（1）分析它的原因 （2）修复它。对于错误的原因，要尽可能全面记录。这并不只是包括技术问题，类似“在使用传入参数之前，我应该检查它们的正确性”或者“在交给集成测试之前，我应该确认，应该执行循环 n 次还是 n-1 次”。 同时也包括管理问题，类似 “我应该在单元测试之前手工检查一下”或者“如果当 Ellen 想检查我的设计是否满足全部需求的时候，我允许了，那么……”。在收集所有这些信息之后，通知所有人，让每个人都知道什么引起了错误，这样，这类知识就可以更广泛的传播，这类错误就会更少发生。

参考文献

Kajihara, J., G.Amamiya, and T. Saya, "Learning from Bugs", IEEE Software, 10, 5 (September 1993), pp.46-54.

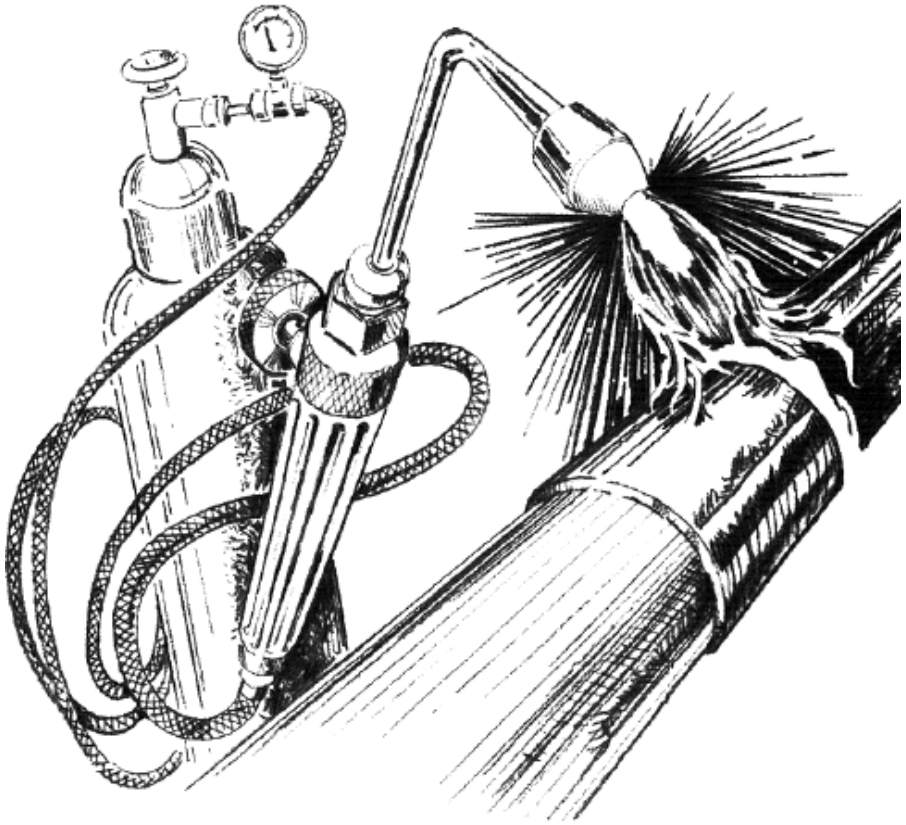
原则 126 对“错”不对人

DON’T TAKE ERRORS PERSONALLY

编写软件需要的细节和完善程度，是任何人都无法达到的。我们应该致力于不断的进步，而不是尽善尽美。当你或他人在你的代码中发现错误时，公开坦诚地讨论它。与其责骂自己，不如将它当作自己和他人的学习经历（更多信息见原则 125）。

参考文献

Gerhart, S., and L. Yelowitz, “Observations of Fallibility in Applications of Modern Programming Methodologies”, IEEE Transactions on Software Engineering, 2, 3 (September 1976), pp. 195-207, Section Ⅰ.



第七章 管理原则

MANAGEMENT PRINCIPLES

*管理*是围绕软件开发的所有工程活动，进行计划（plan）、控制（control）、监视（monitor）和报告（report）的一组活动。

原则 127 好的管理比好的技术更重要

GOOD MANAGEMENT IS MORE IMPORTANT THAN GOOD TECHNOLOGY

好的管理能够激励人们做到最好。糟糕的管理会打击人们的积极性。所有伟大的技术（CASE 工具、技术、计算机、文字处理器等）都弥补不了拙劣的管理。好的管理，即使是在资源匮乏的情况下，也能产生巨大的效果。成功的软件初创公司，不是因为他们有强大的流程或者强大的工具（或与此相关的优秀产品）而成功。大多数的成功都是源于成功的管理和出色的市场营销。

作为一个管理者，你有责任做到最好。对于管理，没有一个普遍的“正确”的风格。管理风格必须适应于场景。在某个情形下是一个独裁者，仅仅几分钟后在另外一个场景又变为基于共识的领导者，这对一个成功的领导者并不是罕见的事情。有些管理风格是与生俱来的。有些是可以靠后天学习培养的。如果必要，可以阅读书籍，并参加关于管理风格的短期培训。

参考文献

Fenton, N., "How Effective Are Software Engineering Methods?" *Journal of Systems and Software*, 22, 2 (August 1993), pp. 141-146.

译者注

CASE，为计算机辅助软件工程(Computer Aided Software Engineering)。

原则 128 使用恰当的方法

USE APPROPRIATE SOLUTIONS

技术问题需要使用技术的方法。管理问题需要使用管理的方法。政治问题需要用政治的方法。切忌用不恰当的方法来解决问题。

原则 129 不要相信你读到的一切

DON'T BELIVE EVERYTHING YOU READ

一般来讲，相信特定想法的人，会搜索支持这个想法的数据，而抛弃不支持的数据。一个人要说服处于某个位置的某人，显然会使用支持的数据，而不是不支持的数据。当你读到，“使用 X 方法，你也可以实现高达93%的开发效率（或质量）的增长”，这个方法可能真的取得了这样的结果。但是这很可能是个例外。在大多数情况下，大多数项目很少会经历戏剧性结果。而有些项目甚至可能会因使用方法 X 而减产。

参考文献

Fenton, N., "How Effective Are Software Engineering Methods?" *Journal of Systems and Software*, 22, 2 (August 1993), pp. 141-146.

原则 130 理解客户的优先级

UNDERSTAND THE CUSTOMERS' PRIORITIES

很有可能的是，如果客户能按时获得 10%的系统功能，那么他们宁愿 90%的功能延迟交付。原则 8 的推论更令人震惊，但很有可能就是这样的。一定要弄明白！

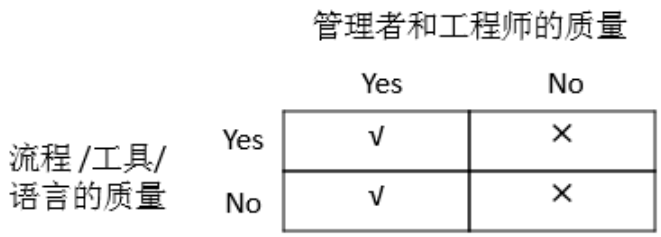
当你和客户沟通时，一定要确认你知道客户的优先级。这些可以很容易地记录在需求规格说明中（原则 50），但真正的挑战是理解可能不断变化的优先级。此外，你必须理解客户对于“必要”（essential）、“期望”（desirable）和“可选”（optional）的说明。他们真的会对一个不满足任何期望和可选需求的系统感到满意吗？

参考文献

Gilb, T., "Deadline Pressure: How to Cope with Short Deadlines, Low Budgets and Insufficient Staffing Levels," in *Information Processing*, H.J. Kugler, ed., Amsterdam: Elsevier Publishers, 1986.

原则 131 人是成功的关键

PEOPLE ARE THE KEY TO SUCCESS



具备合适经验、才能、培训的高技能人才，是在预算内按时完成满足用户需求软件的关键。合适的人，即使没有足够的工具、语言和流程，也会成功。不合适的人（或者合适的人，但没有足够的培训或者经验），即使有合适的工具、语言和流程也很可能会失败。根据构造性成本模型（COCOMO）（Boehm, B., *Software Engineering Economics*, Englewood Cliffs, N.J.: Prentice Hall, 1984）估算，最优秀的人效率是其他人的四倍。如果最优秀的人花费四倍的薪水，你可以做到收支平衡，而且最终你很可能会得到一个更好的产品（原则 82）。如果他们的花费没有这么多，你降低了成本，还得到更好的产品。这是双赢。

在面试候选人时，记住人才质量是无法替代的。在面试完两个人后，公司经常说，"面试者 *x* 比 *y* 更好，但是 *y* 已经足够好了，并且了成本更低"。你不可能有个全明星阵容的组织，但是，除非你现在拥有的超级明星过多，否则雇佣他们吧！

参考文献

Weinberg, G., *The psychology of Computer Programming*, New York: Van Nostrand Reinhold, 1971, Chapters 6-7.

原则 132 几个好手要强过很多生手

A FEW GOOD PEOPLE ARE BETTER THAN MANY LESS SKILLED PEOPLE

本原则与原则 131 是一致的。原则 131 说你应该总是雇佣最好的工程师。本原则想说：对一个关键任务，你最好只安排少数有足够经验的工程师，而不是安排许多没有经验的工程师。这就是 Don Reifer 的"管理原则第 6 条"。另一方面，Manny Lehman 警告说，你不能完全依赖"少数优秀的人"。如果他们离职了呢？最好的建议是，在一个项目中建立合适的人员配比，并且切忌不要向两个极端发展。

参考文献

Reifer, D., "The Nature of Software Management: A Primer", Tutorial: Software Management, D. Reifer, ed., Washington, D.C: IEEE Computer Society Press, 1986, pp. 42-45.

译者注

[1] Don Reifer，是一个软件工程方面的专家，拥有 40 年以上的经验，已出版 9 本专著。详见 https://reifer.com/bio/

[2] Manny Lehman，是一个计算机科学家，1989 年获得英国皇家工程学院院士。详见 https://en.wikipedia.org/wiki/Manny_Lehman_(computer_scientist)

原则 133 倾听你的员工

LISTEN TO YOUR PEOPLE

你必须信任那些为你工作的人。如果他们不值得信赖（或者你不信任他们），你的项目将会失败。如果他们不信任你，你的项目也将会失败。你的员工很快就能看出你不信任他们，就跟你能很快发现你的老板不信任你一样。

信任的第一个原则就是倾听。有很多机会去倾听你的员工：当他们来你的办公室说他们遇到的问题时，当你需要从他们那获取软件开发的预估时，当你在做“走动管理”时。不论你的员工何时向你说，倾听（listen）并且听取（hear）。他们认为跟你汇报的都是重要的事情，否则他们也不会告诉你。有很多方法可以让他们知道你在倾听：眼神交流、恰当的身体语言、“复述”你自认为听到的内容、提出合适的问题来获取更多信息，等等。

参考文献

Francis, P., *Principles of R&D Management*, New York: AMACOM, 1977, pp. 114-116.

译者注

走动管理，Managing By Walking Around（MBWA），指管理人员通过随机非正规的走动方式，来了解工作状态的管理方式。详见 https://en.wikipedia.org/wiki/Management_by_wandering_around

原则 134 信任你的员工

TRUST YOUR PEOPLE

一般来讲，如果你信任你的员工，他们就是可信赖的。如果你对待员工好像你不信任他们，那么他们会给你不要去信任他们的原因。当你信任别人，而且也没有给他们理由不信任你时，他们也会信任你。相互信任是成功管理的要素。

当你的某个员工说，“我今天下午 2 点可以离开吗？之后我将在周末多工作几个小时”，你应该回答，“当然可以”。 你没什么损失，而且还得到了员工对你的忠诚与尊敬。成为一个坏人的机会远远比成为一个好人要多。抓住每个能让你成为好人的机会。或许几周后，你就会需要员工多工作几个小时，来完成一个你需要完成的工作。

参考文献

McGregor, D., *The Human Side of Enterprise*, New York: McGraw-Hill, 1960.

原则 135 期望优秀

EXPECT EXCELLENCE

如果你对员工有更高的期待，他们将表现的更好。沃伦·本尼斯（Warren Bennis）的研究最后证明：你期望的越多，获得的成就会越多（显然有一定限制）。在许多试验中，来自不同背景的组员被划分为两个有着一致目标的小组。一个小组被期望是优秀的。另一个小组被期望是平庸的。在所有试验中，被期望优秀的组都表现的比另一组好。

有很多方法可以展现你对优秀的期望：成为表率（努力工作、为你的努力成就感到骄傲、工作期间不要玩电脑游戏）。为你的员工提供教育培训福利，以帮助他们达到最佳状态。奖励出色的行为（但请参见原则 138）。指导、辅导、劝勉以及尝试激励表现比较差的人向更好的工作产出和习惯转变。如果你(或者他们)失败了，在你的组织或者公司中，为他们寻找更多其他合适的机会。如果任何尝试都失败了，那就帮他们在外面再找个工作吧。你不能让他们待在一个不合适的工作岗位，但是你也必须要表现出同情。如果你让他们放任自流，你的产品将会是低质量的，并且你的其他员工将会觉得不佳的表现是可以接受的。

参考文献

Bennis, W., *The Unconscious Conspiracy: Why Leaders Can't Lead*, New York: AMACOM, 1976.

译者注

沃伦·本尼斯（Warren Bennis），是美国当代杰出的组织理论、领导理论大师。详见 https://wiki.mbalib.com/wiki/沃伦·本尼斯

原则 136 沟通技巧是必要的

COMMUNICATION SKILLS ARE ESSENTIAL

在为你的项目招募成员时，不要低估团队合作和沟通的重要性。最好的设计师可能会变成差劲的资产，如果他/她不能沟通、说服、倾听和妥协。

参考文献

Curtis, B., H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems", *Communications of the ACM,* 31, 11 (November 1988).

原则 137 端茶送水

CARRY THE WATER

当你的员工要工作很长时间来完成软件工程的工作时，你应该工作相同的时间。这样树立了正确的榜样。如果你的员工知道你会和他们同面困境，那他们会更愿意付出努力并且出色地完成工作。我的第一位工业界经理，汤姆林森·劳舍尔，正是这么做的。这对我们的态度影响重大。在多次危机中，汤姆扮演“为他的员工效劳”的角色。这确实有效。

如果你不能帮助解决工作中的问题，可以让员工感受到你能够跑腿、订披萨、拿苏打水、端茶到送水及任何他们需要的事情。给他们惊喜！在午夜给他们带披萨。

参考文献

Rauscher, T., private communication, 1977.

原则 138 人们的动机是不同的

PEOPLE ARE MOTIVATED BY DEFFERENT THINGS

这可能是我学习成为一名管理者的过程中最大的教训。我曾经错误的认为，能够打动员工的东西和打动我的东西是一样的。我记得有一年，为了能够公平的分配加薪，我努力思考如何处理加薪池。我特别希望对干的好的特定员工大幅加薪，以激励他们更加努力的工作。当我向某个员工第一次提出加薪时，他说："谢谢，但是我真正需要的是一台更快的电脑"。

有时要搞清激励个人时哪里用萝卜、哪里用大棒并不容易。众所周知，人各不同，负面和正面的激励都起作用，但是正面的激励经常被管理层忽视。开始找出激励个人的因素，一个比较好的方法就是倾听（原则 133）。剩下的可能就是反复试验。不过不论你做什么，不要因为害怕选错而克制奖励。

参考文献

Herzberg, F., "One More Time: How Do You Motivate Employees" *Harvard Business Review* (September-October 1987).

原则 139 让办公室保持安静

KEEP THE OFFICE QUIET

最有效率的员工和公司都拥有安静和私密的办公区。他们把电话静音或者设置呼叫转移。他们隔绝于日常的、非工作事宜的干扰。与此相反，通常工业界朝着开放、美观的办公室风格发展，这降低了设施的花销，但显著地降低了开发效率和质量。当然，通常的管理路线说，这样的安排可以“便利沟通”。这不是真的。这样的安排“便利了干扰和噪音”。

参考文献

DeMarco, T., and T. Lister, Peopleware, New York: Dorset House, 1987, Chapter 12.

译者注

虽然越来越多的公司使用开放的办公区布局，以增强研发人员之间的沟通，但保持编码或设计时的安静环境仍然是一种很强的需求。在某些资料中，建议使用降噪耳机达到类似的目的。

原则 140 人和时间是不可互换的

PEOPLE AND TIME ARE NOT INTERCHANGEABLE

只用“人-月”来衡量一个项目几乎没有任何意义。如果一个项目能够由六个人在一年内完成，是不是意味着 72 个人能在一个月内完成呢？当然不是！

假设你有 10 个人在做一个预期三个月完工的项目。现在你认为你比计划晚了三个月，也就是说，你预估还需要 60 人月（6 个月 * 10 个人）。你不能再加 10 个人并期望项目回到计划上来。实际上，很可能因为额外的培训和沟通成本，再增加 10 个人会使项目更进一步延期。这个原则通常叫做布鲁克斯定律（Brooks' Law）。

参考文献

Brooks, F., *The Mythical Man-Month*, Reading, Mass.: Addison-Wesley, 1975, Chapter 2.

译者注

本原则由 Frederick P. Brooks 在《人月神话》（The Mythical Man-Month）中提出。Brooks 以领导开发 IBM 的大型计算机 System/360 和 OS/360 操作系统而著名。1999 年，Brooks 因其在计算机体系结构、操作系统和软件工程领域划时代的贡献而获得图灵奖。

原则 141 软件工程师之间存在巨大的差异

THERE ARE HUGE DIFFERENCES AMONG SOFTWARE ENGINEERS

从最好的软件工程师到最差的软件工程师，研发效率（按每人月的代码行来衡量）可能相差 25 倍之多，质量（按每千行代码中发现的错误来衡量）可能相差 10 倍之多。

参考文献

Sackman, H., et al., “Exploratory Experimental Studies Comparing Online and Offline Programming Performance”, Communications of the ACM, 11, 1 (January 1968), pp. 3-11.

原则 142 你可以优化任何你想要优化的

YOU CAN OPTIMIZE WHATEVER YOU WANT

任何项目都可以优化任何其想要优化的"质量"因素。在优化任何一个因素时，通常会贬低其他"质量"因素。在 G. Weinberg 和 E. Schulman 主导的具有里程碑意义的试验中，五个软件开发团队被赋予相同的需求，但每个团队都被告知要优化一些不同的东西：开发时间、程序大小、使用的数据空间、程序清晰度和用户友好性。除了一个团队之外，就其要优化的属性而言，所有其他团队开发的程序都被评为了最佳。

如果你告诉你的员工一切（例如工期、大小、可维护性、性能和用户友好性）都同等重要，那么任何地方都不会被优化。如果你告诉他们只有一两个很重要，而其余的都不重要，那么只有重要的地方会得到改善。如果你给他们一个先验的相对排名，该排名可能不适用于项目中的所有情况。事实是，在产品开发过程中，有很多权衡 — 不同的权衡 — 要不断进行取舍。与你的员工一起工作，并帮助他们了解你和你的客户的优先级。

参考文献

Weinberg, G., and E. Schulman, "Goals and Performance in Computer Programming," Human Factors, 16 (1974), pp. 70-77.

原则 143 不显眼地收集数据

COLLECT DATA UNOBTRUSIVELY

数据收集在以下方面极为重要：帮助未来成本预测，评估项目或组织的当前状态，评估管理、过程或技术变更的影响等。另一方面，以突兀的方式收集数据是没有意义的（比如说，如果它需要软件开发人员做大量额外的工作），因为这种收集方式会影响数据本身。此外，从不想提供此类数据的开发人员那获取数据可能会毫无用处，因为不合作的开发人员不太可能提供有意义的数据。

收集数据的最佳方法是自动进行，而没有开发人员可感知到的干扰。显然，你不能在所有时间对所有数据都这么做，但是应尽可能自动地执行数据收集。

参考文献

Pfleeger, S., “Lessons Learned in Building a Corporate Metrics Program”, IEEE Software 10, 3 (May 1993), pp. 67-74.

原则 144 每行代码的成本是没用的

COST PER LINE OF CODE IS NOT USEFUL

给定一组特定的要求，我们可以选择以多种语言中的任何一种来实现程序。与选择非常低级的语言相比，选择非常高级的语言将花费我们更少的时间（原则 152）。因此，使用高级语言时，总开发成本将大大降低。但是，由于软件开发的固定成本（例如用户文档、需求和设计），如果我们选择高级语言，每行代码的成本实际上会增加！Capers Jones 通过一个与制造业的类比很好地解释了这一点：随着生产元件的数量降低，元件的单位成本会上涨，因为现在固定成本必须要由更少的元件分摊。

参考文献

Jones, C., Programming Productivity, New York: McGraw-Hill, 1986, Chapter 1.

原则 145 衡量开发效率没有完美的方法

THERE IS NO PERFECT WAY TO MEASURE PRODUCTIVITY

定义开发效率，两种最常用的方法是每人每月的源代码行数（SLOC: Source Lines Of Code）和功能点数（FP: Function Points）。两者都有问题。在大多数工程或制造领域，乍看用源代码行数（SLOC）来衡量看起来不错，因为产出越多越好。然而，如果你有两个程序实现相同的功能，一个程序的大小是另一个的两倍，并且两个程序都具有相同的质量（当然，除了大小），那么较小的程序会更好。功能点数（FP） 似乎能解决这个问题，因为它们不是衡量解决方案的复杂性，而是（通过对需求规格说明的分析来）衡量问题的复杂性。但是这里也存在一个问题。假设两个需求规格说明在各个方面都是相同的，除了一个说，“如果系统崩溃，全人类将被摧毁”，另一个说，“如果系统崩溃，两个五岁的孩子将轻微感觉不便”。显然，前者是一个更加困难的问题，因此其开发效率应该比后者低得多。有一些公开的技术可以将代码行数估计值转换为功能点数估计值，反之亦然。显然，任何一个都不能比另一个拥有持续的优势。

接受这个事实：完美是不可能的。使用开发效率度量和成本估算模型来确认你的直觉和你的亲身经验。永远不要依靠它们作为你唯一的衡量。

参考文献

Fairley, R., "Recent Advances in Software Estimation Techniques,"14th IEEE International Conference on Software Engineering. Washington, D.C.: IEEE Computer Society Press, 1992.

原则 146 剪裁成本估算方法

TAILOR COST ESTIMATION METHODS

许多成本估算方法可商购获得。每种方法都是基于从大量已完成项目中收集的数据。这些方法中的任何一种都可以用于为软件开发生成大致的估计。要使用它们生成更准确的估计，必须针对工作环境进行剪裁。这种剪裁使模型适应你的人员和应用类型，消除在你的环境中不变的变量，增加在你的环境中影响开发效率的变量。

在 Barry Boehm 的《软件工程经济学》（《Software Engineering Economics》）中，第 29 章详细说明了如何根据你的环境剪裁构造性成本模型（COCOMO）。其他成本估算方法也提供了类似的剪裁指南。你必须完全接受这种剪裁的精神，否则最终会得出不准确的结果。

参考文献

Boehm, B., Software Engineering Economics, Englewood Cliffs, N.J.: Prentice Hall, 1981, Section 29.9.

原则 147 不要设定不切实际的截止时间

DON'T SET UNREALISTIC DEADLINES

不可避免的结局是，一个不切实际的截止日期将无法得到满足。这样截止日期的设立削弱士气，使你的员工不信任你，造成高离职率，并产生其他不良影响。这些因素使不切实际的截止日期更加无法实现。绝大部分软件项目的完成都远远超出预算，并且大大超出了计划的完成日期。为了满足日程表的约束，质量通常会被降低。这损害了整个软件行业的信誉。问题通常不在于软件工程师的生产力低下或经理的管理不善。问题在于预先做出的估算很差。

参考文献

DeMarco, T., "Why Does Software Cost So Much?" IEEE Software, 10, 2 (March 1993), pp.89-90.

原则 148 避免不可能

AVOID THE IMPOSSIBLE

这似乎是显而易见的建议。另一方面，许多项目承诺按时交付产品，这是 100% 不可能的。Barry Boehm 将“不可能的区域”定义为：预期的产品开发时间与需要消耗的人月数之间的关系。具体来说，从编写软件需求规格说明到交付产品所花费的时间不会少于 2.15 乘以人月的立方根，即：

$$\mathrm{T} > 2.15\ \sqrt[3]{PM}$$

所有已完成项目中有 99% 遵守了该规则。是什么让你认为自己可以做得更好？ 如果你仍然认为可以做更好，请参阅原则 3、19、158 和 159。

参考文献

Boehm, B., Software Engineering Economics, Englewood Cliffs, N.J.: Prentice Hall, 1981, Section 27.3.

原则 149 评估之前先要了解

KNOW BEFORE YOU COUNT

Gerald Weinberg（Rethinking Systems Analysis and Design, New York: Dorset House, 1988, p.32）很好地阐明了这一原则："在你可以评估任何事情之前，你必须了解一些东西"。他谈论的是许多人在评估软件中的东西，但不知道他们在评估什么。他提供了一个很好的例子。我们有一些数据，是关于软件界有多大比例是做维护而非开发。但是我们可以识别什么是维护工作吗？是否可以将完全替代已有系统的"新"开发视为维护或开发？对现有系统的"修改"将现有功能加倍并删除了 95% 的旧功能，是否被视为维护或开发？

当为你的项目选择指标时，请确保你在测量的与你要实现的目标有关。（请参阅 1993 年 9 月 《IEEE Software》的 Manager Column 中的开头段落）。这通常需要使用多个指标。记住：即使每个人都以同一种方式进行衡量某事，这种方式也并非自动适合你。要考虑你的指标。由于所有东西都可以被观察（并且在大多数情况下可以被测量），因此请仔细选择什么是你想要观察（和测量）的。下面引用的参考文章，是我所见过的对组织定制化指标计划的最佳描述。

参考文献

Stark, G., R. Durst, and C. Vowell, "Using Metrics in Management Decision-Making." IEEE Computer, 27, 9 (September 1994).

原则 150 收集生产力数据

COLLECT PRODUCTIVITY DATA

所有成本评估模型的准确性都取决于这些模型为你的工作场景进行的剪裁。但是，如果你尚未从过去的项目中收集详细的数据，那么今天你就无法剪裁你的成本估算模型。因此，现在你有个很好的借口不进行准确的成本估算。但是明天呢？如果你今天不开始收集详细数据，那么你将来也无法剪裁成本估算模型。那你还在等什么呢？还请记住曼尼·雷曼（Manny Lehman）的建议：少量经过充分理解、认真收集、模型化及演绎的数据，要好于大量没有这些特性的数据。

参考文献

Boehm, B., Software Engineering Economics, Englewood Cliffs, N.J.: Prentice Hall, 1981, Section 32.7.

原则 151 不要忘记团队效率

DON’T FORGET TEAM PRODUCTIVITY

确定一套针对个人的生产力衡量标准相对容易（当然，这些标准可能无法提供准确的结果。如原则 142、144 和 145 所强调）。但是，请注意，优化所有个体的生产力并不一定会产出最佳的团队生产力。把这和一支篮球队相比较。每个球员都能通过总是在控球时自己投篮得分来优化自己的表现。然而这个球队肯定会输。曼尼·雷曼（Manny Lehman）报告了一项软件开发工作，其中个人生产力增加了两倍，而企业生产力却下降了！

这里有两个教训要总结：首先，不同的措施适用于不同的人。其次，要衡量团队的整体效率，可通过跟踪一些数据（如，按照时间周期和问题难度总结的、反映团队解决突出问题能力的报告）。

参考文献

Lehman. M., private communication, Colorado Springs, Col.: (January 25, 1994).

原则 152 LOC/PM 与语言无关

LOC/PM INDEPENDENT OF LANGUAGE

通常认为，不管使用哪种语言，程序员平均每人每月可以生成 x 行高质量代码。因此，如果一个程序员每月可以用 Ada 语言写出 500 行优质代码，那么这个人用汇编语言也可以每月写出 500 行优质代码。C.Jones 在《Programming Productivity》（New York: McGraw-Hill, 1986, 第一章）中提出相反的观点。使用高级语言时，实际的生产力当然会大大提高，因为 500 行 Ada 代码可以做的事比 500 行汇编代码多得多。此外，语言选择会极大地影响可维护性（原则 193）。

在启动项目时，你需要对程序员将使用的语言有所了解。这是必需的，以便你可以估算代码行数。代码行数又可以用来计算项目的工作量和工期。

参考文献

Boehm, B., Software Engineering Economics, Englewood Cliffs, N. J.: Prentice Hall, 1981, Section 33.4.

译者注

[1] LOC，Line of Code，代码行数。

[2] PM，Person month，人月。

原则 153 相信排期

BELIEVE THE SCHEDULE

		排期的现实性		
		现实的	勉强的	非常勉强的
团队相信排期	Yes	高	中	低
	No	低	低	低

一旦建立了可行的排期（原则 146、147 和 148）并分配了适当的资源（原则 157），所有各方都必须相信排期。如果工程师不认为排期切合实际，他们将不会成功的按照排期执行。排期成功的概率，与现实相比，更多是一个关于排期信心的函数。

最好的建议是，让工程师制定排期。不幸的是，这并不总是可能的。第二个最好的建议是，让工程师参与在功能、进度和项目放弃之间进行的艰难权衡。很少有工程师会宁愿因取消项目而失去工作，也不愿努力以满足苛刻的排期。

参考文献

Lederer, A., and J. Prasad, “Nine Management Guidelines for Better Cost Estimating· "Communications of the ACM, 35, 2 (February 1992), pp. 51-59, Guideline 1.

原则 154 精确的成本估算并不是万无一失的

A PRECISION-CRAFTED COST ESTIMATE IS NOT FOOLPROOF

假设你的团队已经收集了有关过去表现的大量数据。假设你根据此数据，从众多成本估算模型中选择了一个并进行了量身定制，以适应团队的能力。假设你是一个项目经理。你有一个新项目并使用了这个定制的模型。该模型报告该软件将花费 100 万美元。这意味着什么？这并不意味着你的软件将花费 100 万美元。

原因有三个：（1）你，（2）假设（3）概率。首先，是你。你的领导能力将对实际结果产生重大影响。例如，你可以在五秒钟内破坏团队花了一年时间建立的士气。其次，你为生成初始估计所做的所有假设可能不都证明是准确的。例如，如果你只有数量更少的合格人才怎么办？如果需求改变怎么办？如果你的关键人物生病了怎么办？如果一半的工作站在你最需要的时候出现故障怎么办？第三，估计值只是概率分布中的峰值。如果我告诉你我要抛硬币 100 次，并要求你预测出现硬币正面的次数，你很可能会选择 50 次。这是否意味着真的会出现 50 次正面？当然不是。实际上，如果真的刚好出现了 50 次正面，你将感到很惊奇！

参考文献

Gilb, T., Principles of software Engineering Management, Reading, Mass.: Addison-Wesley, 1988, Section 16.7.

原则 155 定期重新评估排期

REASSESS SCHEDULES REGULARLY

排期通常在项目启动时设定。其中包括中间期限和产品交付期限。每个阶段完成后，排期必须被重新评估。一个进度落后的项目很少能在后续阶段恢复到原计划。因此，设计完成延迟一个月的项目，将至少延迟一个月交付。在大多数情况下，增加或减少人员只会进一步延迟该项目（原则 140）。最常见的方法是不更改产品的交付日期（毕竟，我们还不想让客户失望，对吗？）。随着每个中间里程碑错过越来越多的时间，分配给测试的时间越来越少（原则 119）。最后，以下两种情况之一必然会发生：（1）产品出厂时没有足够高的质量，或者（2）在项目后期很晚才通知客户有严重延期。这两种情况都不可接受。作为一名管理者，你的责任是预防灾难。

相反，应与客户和/或上级建立工作关系。要报告每个可能的日期变更（通常是延期），并讨论克服这些困难的可选策略。只有各方的早期干预和参与才能防止延期升级。

参考文献

Gilb, T., Principles of Software Engineering Management, Reading, Mass: Addison-Wesley, 1988, Section 7.14.

原则 156 轻微的低估不总是坏事

MINOR UNDERESTIMATES ARE NOT ALWAYS BAD

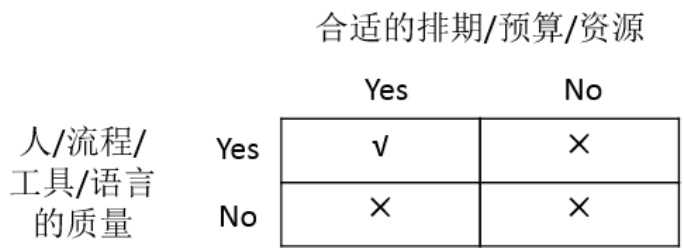
假设士气没有削弱，在被认为稍稍落后进度的项目中，其成员会努力工作赶上进度，从而提高生产力。类似地，在被认为稍稍提前进度的项目中，其成员经常会休假、工作更少时间、花更长的时间读邮件、以其他方式放松，从而降低生产力。因此，成本预估本身就会影响项目产出。对任何一个特定的项目，被轻微低估成本对比被轻微高估成本，都会花费更少的资源。然而要注意，如果项目成员认为排期被严重低估了，士气和生产力都会下降。

参考文献

Abdel-Hamid, T., and S. Madnick, "Impact on Schedule Estimation on Software Project Behavior," IEEE Software, 3, 4 (July 1986), pp. 70-75.

原则 157 分配合适的资源

ALLOCATE APPROPRIATE RESOURCES



不管人员的质量如何，工具、语言或流程的可用性如何，人为强加的进度和不恰当的预算将会毁了一个项目。

如果你试图压缩排期或预算，参与项目的工程师将不会高效地工作，当不可避免的延期发生时，没有人会采取行动，士气将受到影响，并且最重要的是，项目的花费很可能比合理的成本要高。

参考文献

DeMarco, T., "Why Does Software Cost So Much?" IEEE Software, 10, 2 (March 1993), pp.89-90.

原则 158 制定详细的项目计划

PLAN A PROJECT IN DETAIL

每个软件项目都需要一个计划。详细程度应该适合于项目的大小和复杂性。你需要计划的最小集合如下：

- 显示任务之间相互依赖关系的 PERT 表。
- 显示每个任务的活动何时进行的甘特图。
- 实际里程碑的列表(基于早期的项目，见原则 150)。
- 编写文档和代码的一套标准。
- 各种不同的任务中的人员分配。

随着项目的复杂性增加，以上的每个要求都会变得越来越详细和复杂，其他类型的文档也会变得非常需要。一个没有计划的项目，在它开始之前就已经失控了。正如《爱丽丝梦游仙境》中柴郡猫对爱丽丝所说：“如果你不知道要去哪里，那你也就无法达到那里！”。

参考文献

Glaser, G., "Managing Projects in the Computer Industry," IEEE Computer, 17, 10 (October 1984), pp. 45-53.

译者注

PERT，Program Evaluation and Review Technique，计划评审技术

原则 159 及时更新你的计划

KEEP YOUR PLAN UP-TO-DATE

这是 Don Reifer 的"管理原则#3"。原则 158 说对于一个软件项目，你必须做计划。然而，有一个过时的计划比完全没有计划更糟糕。当你没有计划时，你应该知道你已经失控了。当你有一个过时的计划，你可能天真地以为在你控制之中。所以无论任何时候情况发生变化，要更新的你的计划。这些情况包含需求变更、进度延迟、方向变更、发现过多错误或者与其他任何与原始条件的偏差。

一份写得好的计划应该列举风险、潜在风险正成为威胁的警告信号、为减少威胁而制定的应急计划（原则 162 ）。随着项目的进行，如果预期的风险成为威胁，要实施应急计划并更新项目计划。真正的挑战是那些不可预见的变化。在这种时候，人们常常需要重新做计划。在这种时候，人们常常需要全面地重新规划整个项目的其余部分，包括新的假设、新的风险、新的应急计划、新的排期、新的里程碑、新的人力资源分配等等。

参考文献

Reifer, D., "The Nature of Software Management: A Primer," Tutorial: Software Management, D. Reifer, ed., Washington, D.C.: IEEE Computer Society Press, 1986, pp. 42-45.

原则 160 避免驻波

AVOID STANDING WAVES

遵循原则 159 (保持你的计划是最新的)的一个奇怪的副作用是驻波。在这种情况下，你总是计划“在未来几周内”可以“康复”的策略。由于落后于计划的项目往往会进一步落后于计划，因此这种“康复”的策略“在未来几周内”将会需要越来越多的资源(或奇迹！)。如果不采取纠正措施，波动会变得越来越大。一般来说，重新安排时间和重新计划需要采取行动，而不仅仅是保证很快就能解决问题。不要因为你只是落后了几天，就认为问题会消失。所有的项目都是"一天一天地落后"。

参考文献

Brooks, F., The Mythical Man-Month, Reading, Mass.: Addison-Wesley, 1975, Chapter4.

译者注

驻波（英语：standing wave 或 stationary wave）为两个波长、周期、频率和波速皆相同的正弦波相向行进干涉而成的合成波。参考：https://zh.wikipedia.org/wiki/驻波

原则 161 知晓十大风险

KNOW THE TOP 10 RISKS

作为项目经理，当你开始一个项目时，你需要熟悉最经常导致软件灾难的情况。这些是你最可能遇到的风险，但很可能不是全部。根据 Boehm 的说法，它们是：

- 人员短缺（原则 131）。
- 不切实际的排期（原则 148）。
- 不理解需求（原则 40）。
- 开发糟糕的用户界面（原则 42）。
- 当客户并不需要时尝试镀金（原则 67）。
- 不控制需求变更（原则 179 和 189）。
- 缺乏可重用的或者接口化的组件。
- 外部执行任务不足。
- 糟糕的响应时间。
- 试图超越当前计算机技术的能力。

既然现在你了解了最常见的风险，可以在这个基础上添加你的环境和项目中的特有风险，并制定降低这些风险的计划（原则 162 ）。

参考文献

Boehm, B., "Software Risk Management: Principles and Practices," *IEEE software, 9,* 1 (January 1991), pp. 32-39.

译者注

第 8 条，“外部执行任务不足”，英文原文为 Shortfalls in externally performed tasks。意思是，由外部承包商完成的任务不满足要求。

原则 162 预先了解风险

UNDERSTAND RISKS UP FRONT

在任何软件项目中，都无法准确预测会出现什么问题。然而，总有地方会出现问题。在项目计划的早期阶段，要梳理与你项目相关的最大风险列表。对于每个风险，要量化其真正发生会带来的破坏程度，并量化这种损失发生的可能性。这两个数字的乘积，是你对特定风险的风险敞口。

在项目开始时，构建一个决策树，梳理所有可能降低风险敞口的方法。然后要么立刻对可能造成的后果采取行动；要么制定计划，在风险敞口超过可接受范围时，采取各种措施。（当然，需要预先说明如何识别这些风险，以便趁早采取纠正措施。）

参考文献

Charette, R., *Software Engineering Risk Analysis and Management,* New York: McGraw-Hill, 1989, Section 2.2, Chapter 6.

译者注

风险敞口: 在对风险未采取任何防范措施而可能导致出现损失的部分，即实际所承担的风险。

原则 163 使用适当的流程模型

USE AN APPROPRIATE PROCESS MODEL

软件项目中可以使用很多流程模型：瀑布模型、一次性原型、增量开发、螺旋模型、操作原型等等。没有任何一种流程模型适用于公司中的所有项目。每个项目都必须选择一个最适合它的流程。选择应该基于企业文化、风险意愿、应用领域、需求的易变性以及对需求的理解程度。

要研究你项目的特点，然后选择一个最适合的流程模型。例如，在构建原型时，你应该遵循最小化规约、促进快速开发和不需要担心分权制衡的流程。而在构建性命攸关的产品时，情况恰恰相反。

参考文献

Alexander, L., and A. Davis, "Criteria for the Selection of a Software Process Model," *IEEE COMPSAC '91,* Washington, D.C.: IEEE Computer Society Press, pp. 521-528.

原则 164 方法无法挽救你

THE METHOD WON'T SAVE YOU

大家都听过“方法狂热者”的布道，他们说：“如果你正好采用我的方法，你的大多数问题都会消失”。 虽然很多方法都曾受到这类狂言的影响，在 1970 年代和 1980 年代早期大多数名字里都包含“结构化”（structured），在 1980 年代后期和 1990 年代，大多数名字中包含“对象”（object）。虽然这两次浪潮都带来了深刻的见解，并带来提升质量的软件开发概念和步骤，但它们并不是万能药。那些在开发高质量软件方面真正优秀的组织，在采用“结构化”方法之前就很优秀，在采用“面向对象”方法后依然出色。那些以往表现不佳的组织，在采用最新流行的方法后仍然会表现不佳。

作为一名管理者，要提防那些声称基于新的方法将大大提高质量或生产力的虚假的预言家。采用新的方法并没有错，但一个公司如果过去“失败”过（不论是生产力还是质量），在寻找解决方案之前，请尝试找出失败的根源。你现在使用的方法，很可能不是问题的根源！

参考文献

Loy, P., "The Method Won't Save You (But It Can Help)," *ACM Software Engineering Notes, 18,* 1 (January 1993), pp. 30-34.

原则 165 没有奇迹般提升效率的秘密

NO SECRETS FOR MIRACULOUS PRODUCTIVITY INCREASE

这个行业充满了推销员，他们鼓吹通过使用这种工具或那种技术能够降低开发成本。我们都在商务会议上听到有关软件经理的说法，他们声称通过使用工具 x 或语言 y 或方法 z，生产力提高了 50%，75%，甚至 100%。不要相信！这是炒作。软件行业的生产力正在适度提高（每年 3% 至 5%）。事实是，我们有一种简单的方法来降低需求工程的成本：就是不做！对所有其它阶段也是如此。实际上，只通过不开发软件，我们就可以节省很多钱！

你应该会对可削减几个百分点成本或提高几个百分点质量的工具、语言和方法感到满意。然而，如果不了解对客户满意度的影响，降低成本毫无意义。

参考文献

DeMarco, T., and T. Lister, Peopleware, New York: Dorset House, 1987, Chapter 6.

原则 166 了解进度的含义

KNOW WHAT PROGRESS MEANS

我经常听到项目经理报告，“我们比预算低 25%”，或者“比排期提前 25%”。 两者都未必是好消息。“低于预算”通常意味着比预期花费更少的钱。这可能是好事，但是也不一定，除非工作也按期或比排期提前。相似地，“比排期提前”通常表示你比预期完成地多。这可能是好消息，但是也不一定，除非费用还符合或低于预算。下面是一些衡量进度的有意义的标准：

BCWP	“已完成工作预算费用”（Budgeted cost of work performed）衡量你预期目前已完成的工作会花费多少。
ACWP	“已完成工作实际费用”（Actual cost of work performed）衡量你在项目中实际花费了多少。
BCWE	“预期工作预算费用”（Budgeted cost of work expected）衡量你预期花费多少。
$\frac{\text{BCWP} - \text{BCWE}}{\text{BCWE}}$	它体现了真实的技术状态。值大于零表示你比排期提前的百分比。值小于零表示落后排期的百分比。
$\frac{\text{BCWP} - \text{ACWP}}{\text{BCWP}}$	它体现真实的预算状态。值大于零表示低于预算的百分比。值小于零表示超出预算的百分比。

参考文献

U.S Air Force, *Cost/Schedule Management of Non-Major Contracts*, Air Force Systems Command Publication #178-3, Andrews AFB, Md.: (November 1978).

原则 167 按差异管理

MANAGE BY VARIANCE

首先，没有详细的计划是不可能管理一个项目的（原则 158）。一旦你有了计划，就要在必要时更新它（原则 159）。既然你有了最新的计划，你的责任就是根据这个计划来管理项目。当你汇报进度时（无论是书面、口头、正式还是非正式），只需汇报计划和实际之间的差异。项目经理通常会花费大部分时间来报告他们做得如何好。在项目完成时将会有大量的时间用于颁奖。但当项目正在进行时，进度报告应该是“各项都按计划进行，除了…”。通过这种方式，可以将注意力和资源放在有问题的地方。

原则 168 不要过度使用你的硬件

DON'T OVERSTRAIN YOUR HARDWARE

要注意硬件限制对软件开发成本的巨大影响。尤其是，有数据显示，当内存或 CPU 使用率接近 90% 时，软件开发成本将*翻倍*！当接近 95% 时，成本将会增加*两倍*！随着每条指令每秒成本和每字内存成本的极大下降，这个问题似乎不像 15 年前那么严重了。另一方面，在许多应用中仍然有强烈的动机来控制硬件成本（例如大量销售的低成本产品）。

如果内存很容易添加、更快的处理器很容易集成到你的环境中，那就不用担心这个原则；需要时添加即可。如果你的环境使你必须压缩每个字内存和 CPU 周期，那么一定要相应地增加排期。

参考文献

Boehm, B., "The High Cost of Software," in *Practical Strategies for Developing Large Software Systems*, E. Horowitz, ed., Reading, Mass.: Addison-Wesley, 1975.

原则 169 对硬件的演化要乐观

BE OPTIMISTIC ABOUT HARDWARE EVOLUTION

1984 年，13 家主要的航空公司预测：到 1988 年，50% 的软件开发将依然在哑终端上进行。而到 1988 年，大部分软件开发已经从哑终端迁移到 PC 和工作站了。在同一调查中，它们预测只有 15% 的软件开发会使用以太网，而且软件环境中基于 UNIX 的机器将只有 27% 普及率。很明显，这些预测都错了。硬件的速度、容量、标准化以及价格/性能都超出了预期。

参考文献

Davis, A., and E. Comer, "No Crystal Ball in the Software Industry," *IEEE Software*, 10, 4 (July 1993), pp. 91-94, 97.

译者注

狭义的终端分两种：一种是字符终端，或称哑终端（Dumb Terminal），其只有输入输出字符的功能，没有处理器或硬盘，通过串行接口联接主机，一切工作都要交给主机来做；一种是图形终端或工作站，有独立的处理器、内存和硬盘处理图形界面功能，一般通过以太网与主机联接。

原则 170 对软件的进化要悲观

BE PESSIMISTIC ABOUT SOFTWARE EVOLUTION

1984 年，13 家主要的航空公司预测：到 1988 年，它们 46% 的软件开发会使用 Ada（且少于 4% 依然使用 C），且它们 54% 的软件将会从之前的应用中复用。而且，到 1994 年，70% 的软件开发将由基于知识的系统辅助。这些预测都没有实现。在上述情况中，技术要么太不成熟，要么由于一些事件而被取代。

参考文献

Davis, A., and E. Comer, "No Crystal Ball in the Software Industry," *IEEE Software*, 10, 4 (July 1993), pp. 91-94, 97.

原则 171 认为灾难是不可能的想法往往导致灾难

THE THOUGHT THAT DISASTER IS IMPOSSIBLE OFTEN LEADS TO DISASTER

这是杰拉尔德·温伯格（Gerald Weinberg）的“泰坦尼克效应”原则。你决不能沾沾自喜地以为一切都在控制之中，并且会一直保持这样。过度自信是许多灾难的主要原因。陷入麻烦的往往是那些说“这只是一次小小的攀登，我不需要绳索”的登山者，或者那些说“这只是一次短途徒步，我不需要水”的徒步者，亦或是那些说“这把牌我肯定能赢”的扑克玩家。原则 162 强调需要预先分析所有潜在的灾难，提前制定应急计划，并不断重新评估新的风险。本原则强调需要预想这些风险成为现实。最大的管理灾难会在你认为不会发生的时候出现。

参考文献

Weinberg, G., *Quality Software Management,* Vol. 1: Systems Thinking, New York: Dorset House, 1992, Section 15.3.5.

原则 172 做项目总结

DO A PROJECT POSTMORTEM

忘记过去的人注定会重蹈覆辙。

—乔治·桑塔亚纳（George Santayana）, 1908

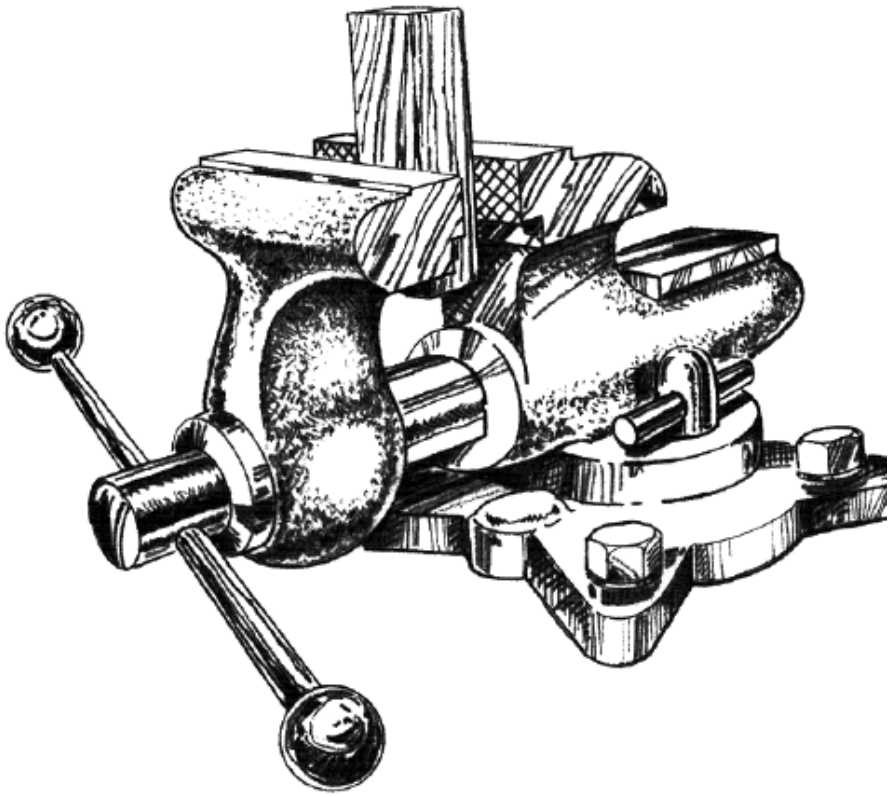
每个项目都会有问题。原则 125 涉及记录、分析技术错误并从中学习。本原则用于对管理错误或者整体的技术错误进行同样的操作。在每个项目结束时，给所有的项目关键参与者一个 3-4 天的任务来分析项目中出现的每一个问题。例如，“我们延迟了 10 天开始集成测试；我们应该告诉客户”。或者，“我们早在知道最基本的需求之前就开始了设计”。或者，"大老板在错误的时间发布了一个‘不加薪’的公告，影响了大家的积极性”。总的来说，主要思路是记录、分析所有不符合预期的事情并从中学习。同时，记录下你认为将来可以采取的不同措施以预防问题发生。未来的项目将会极大受益。

参考文献

Chikofsky, E., "Changing Your Endgame Strategy," *IEEE Software, 7, 6*(November 1990), pp. 87, 112.

译者注

乔治·桑塔亚纳（George Santayana），西班牙著名自然主义哲学家、美学家，美国美学的开创者，同时还是著名的诗人与文学批评家。（引自百度百科）



第八章 产品保证原则

PRODUCT ASSURANCE PRINCIPLES

产品保证是通过使用分权制衡（checks and balances）来确保软件质量的一系列工作。产品保证通常包括：

1. *软件配置管理（Software configuration management）*：是管理软件变更的过程。
2. *软件质量保证（Software quality assurance）*：是检查所有做法和产品是否符合既定流程和标准的过程。
3. *软件验证和确认（Software verification and validation）*：这个过程用于验证（verify）每个中间产品是否正确地建立在以前的中间产品的基础上，以及确认（validate）每个中间产品是否适当地满足客户的要求。
4. 测试（*Testing*）。在前面的章节已介绍过。

原则 173 产品保证并不是奢侈品

PRODUCT ASSURANCE IS NOT A LUXURY

产品保证包含软件配置管理（software configuration management），软件质量保证（software quality assurance），验证和确认（verification and validation），以及测试（testing）。在以上四点中，测试和评估的必要性是最常被承认的，哪怕预算不足。另外三点则经常作为奢侈品而被摒弃掉，如同它们只是大型项目或者昂贵项目的一部分。这些准则的分权制衡（checks and balances），提供了明显更高的可能性，以生产出满足客户期望的产品，并在更接近排期和成本目标的情况下完成。关键是根据项目的规模、形式和内容去定制产品保证的准则。

参考文献

Siegel, S., "Why We Need Checks and Balances to Assure Quality," Qulity Time Column, IEEE Software, 9, 1, (January 1992), pp. 102-103.

原则 174 尽早建立软件配置管理过程

ESTABLISH SCM PROCEDURES EARLY

有效的软件配置管理（SCM: Software Configuration Management）不仅仅是一个记录谁在什么时候对代码和文档进行了怎样修改的工具。它也包括深思熟虑的创建命名约定、策略和过程，以确保所有相关方都参与软件的更改。它必须根据每个项目进行定制。它的存在意味着：

- 我们知道怎样去报告一个软件问题。
- 我们知道怎样去提出一个新的需求。
- 所有利益相关方对于建议的改动都能知晓，且他们的意见都被考虑了。
- 有一块看板用于展示变更请求的优先级和排期。
- 所有基线化的中间产品或最终产品都在掌控之中（即，它们不可能不遵循合适的流程而被修改）。

以上所有内容最好记录在一个文档中，这个文档通常被称为*软件配置管理计划*（SCMP：Software Configuration Management Plan）。这个文档应当在项目早期编写，典型的是在软件需求规格说明评审通过的同时也评审通过。

参考文献

Bersoff, E., V. Henderson, and S. Siegel, Software Configuration Management, Englewood Cliffs, N.J.: Prentice Hall, 1980, Section 5.4.

原则 175 使软件配置管理适应软件过程

ADAPT SCM TO SOFTWARE PROCESS

软件配置管理（SCM）并不是一套对所有项目一律适用的标准实践。SCM 必须根据每个项目的特点去定制：项目规模、易变性、开发过程、客户参与度等等。不是所有情况都适用同一模式。

比如，美国联邦航空管理局 (FAA) 的国家空管系统 (NAS) 有一个七层配置控制看板；显然这对小型项目并不适用。一次性原型的开发，很可能没有在配置管理下的软件需求规格说明也能存活；显然一个大规模的开发项目是做不到的。

参考文献

Bersoff, E., and A. Davis, "Impacts of Life Cycle Models on Software Configuration Management," Communications of the ACM, 34, 8(August 1991), pp.104-117.

原则 176 组织 SCM 独立于项目管理

ORGANIZE SCM TO BE INDEPENDENT OF PROJECT MANAGEMENT

软件配置管理（Software configuration management，SCM）只有在独立于项目管理的情况下才能做好本职工作。经常出于排期压力，项目经理可能试图绕过那些使项目能够长期发展的控制措施。例如，在出现这样的排期问题时，会有这样的诱惑：尽管没有记录它满足了哪些变更需求，但仍有可能接受一个新版本的软件作为基线。当 SCM 向项目经理报告时，除了接受之外，他们几乎什么也做不了。如果它们之间是独立的，SCM 可以实施最适合所有相关人员的规则。

参考文献

Bersoff, E., “Elements of Software Configuration Management”, IEEE Transactions on Software Engineering, 10, 1 (January 1984), pp. 79-87.

原则 177 轮换人员到产品保证

ROTATE PEOPLE THROUGH PRODUCT ASSURANCE

在很多组织中，在以下情况下人员会被转到产品保证组织： （1）作为他们分配的第一个工作（2）当他们在工程软件方面表现不佳时。然而，产品保证工作对于工程的质量和专业水平，与设计和编码工作有同等的要求。另一种选择是，轮换最好的工程人才到产品保证组织工作。一个好的指导方针可能是，每一个优秀的工程师每隔两到三年，要投入六个月的时间到产品保证上。所有这些工程师的期望是，他们可以在"访问"期间对产品保证做出重大改进。这样的政策必须明确说明，工作轮换是对表现优异的一种奖励。

参考文献

Mendis, K., "*Personnel Requirements to Make Software Quality Assurance Work,*" in *Handbook of Software Quality Assurance*, C.G. Schulmeyer, and J. McManus, eds., New York: Van Nostrand Reinhold, 1987, pp. 104-118.

原则 178 给所有中间产品一个名称和版本

GIVE ALL INTERMEDIATE PRODUCTS A NAME AND VERSION

软件开发过程中会有许多中间产出：需求规格说明、设计规格说明、代码、测试计划、管理计划、用户手册等等。每个这样的产出都应该有唯一的名称、版本/修订号和创建日期。如果其中任何一个包含可以相对独立发展的部件(例如，程序中的软件组件，或整个测试计划文档中的单个测试计划)，这些部件也应该被赋予唯一的名称、版本/修订号和日期。“部件列表”应列举中间产出中包含的所有部件及其版本或修订信息，以便你知道每个特定版本和修订的中间产出是由哪个部件的哪些版本和修订组成。

此外，当最终产品发布给客户时，必须给它分配一个唯一的（产品）版本/修订号和日期。然后发布一个“部件列表”，列举所有组成产品的中间产出（以及它们各自的版本和发布号）。

只有通过这样的命名，你才能控制对产品不可避免的更改(原则 16 和 185)。

参考文献

Bersoff, E., V. Henderson, and S. Siegel, *Software Configuration Management*, Englewood Cliffs, N.J.: Prentice Hall, 1980, Chapter 4.

原则 179 控制基准

CONTROL BASELINES

软件配置管理（SCM）的职责，是保持商定的规格并控制对其的变更。

在修复或增强软件组件时，软件工程师偶尔会发现可以更改的其他内容，也许是修复尚未报告的 bug 或添加一些快速的新特性。这种不受控制的变化是不能容忍的。见相关原则 187 条。SCM 应该避免将这些变更合并到新的基线中。正确的过程是由软件工程师提出变更请求（CR: Change Request）。然后，这个变更请求要与来自开发、营销、测试和客户的其它变更请求一起由配置控制委员会（Configuration Control Board）处理。这个委员会负责确定变更请求的优先级和排期。只有这样才能允许工程师进行变更，只有这样 SCM 才能接受变更。

参考文献

Bersoff, E., V. Henderson, and S. Siegel, *Software Configuration Management*, Englewood Cliffs, N.J.: Prentice Hall, 1980, Section 4.1

原则 180 保存所有内容

SAVE EVERYTHING

Paul Erlich 说过，“明智修补的首要原则是保存所有的零件”。 软件从本质上是不断被修补的。由于修补会导致很多错误（原则 195），任何软件更改都很可能需要被回滚。做到这一点的唯一方法是确保在进行更改之前保存所有内容。软件配置管理组织的工作，即在对基线进行批准的更改之前，保存所有内容的所有副本。

参考文献

Erlich, P., as reported by Render, H., private communication, Colorado Springs, Col: 1993.

原则 181 跟踪每一个变更

KEEP TRACK OF EVERY CHANGE

每次变更都有可能引发问题。三个常见问题是：

1. 变更未解决预期要解决的问题。
2. 变更解决了问题，但导致了其他问题。
3. 在将来的某天变更被注意到时，没有人能弄清楚更改的原因（或由谁更改）。

在这三种情况下，预防措施都是跟踪所有变更。

跟踪意味着记录：

- 最初的变更请求（这可能是客户对新功能的请求，客户对故障的投诉，开发人员发现了一个问题，或者开发人员想要添加一个新功能）。
- 用于批准变更的审批流程（谁，何时，为什么，在哪个发布版本中）。
- 所有中间产出的变更（谁，什么，何时）。
- 在变更请求、变更审批和变更本身之间适当的交叉引用。

这样的审计追踪使你可以轻松撤消、重做、并且/或者理解变更。

参考文献

Bersoff, E., V. Henderson, and S. Siegel, Software Configuration Management, Englewood Cliffs, N.J.: Prentice Hall, 1980, Section 7.1.

原则 182 不要绕过变更控制

DON'T BYPASS CHANGE CONTROL

变更得到控制，每个人都会收益。（“控制”并不意味着“阻止”。）能够直接接触到开发人员的客户，通常会直接要求开发人员为他们进行特定修改，来绕过变更控制。这是灾难性的。 它会使项目管理陷入困境。它会导致成本上升。 它使需求规范不准确。这得有多糟糕？

参考文献

Curtis, B., H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," Communications of the ACM, 31, 11(November 1988), pp. 1268-1287.

原则 183 对变更请求进行分级和排期

RANK AND SCHEDULE CHANGE REQUESTS

对于任何投入使用的产品，来自用户、开发人员和市场人员的变更请求都会集中到开发团队。这些变更请求可能反映了对新功能的需求、性能下降的报告、或对系统错误的投诉。应该成立一个委员会，通常称为*配置控制委员会*（CCB：Configuration Control Board），以定期评查所有变更请求。他们的责任是将所有这些变更进行优先级排序，并安排何时（或至少确定将在哪个即将来临的发布版本）它们将会被解决。

参考文献

Whitgift, D., Methods and Tools for Software Configuration Management, New York: John Wiley & Sons, 1991, Chapter 9.

原则 184 在大型开发项目使用确认和验证（V&V）

USE VALIDATION AND VERIFICATION (V&V) ON LARGE DEVELOPMENTS

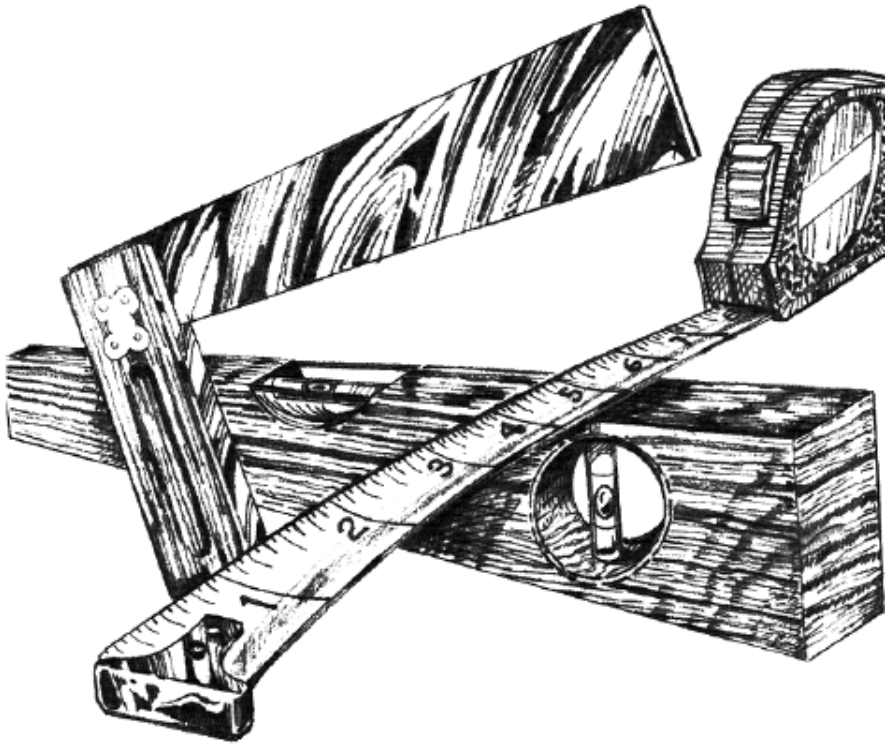
大型软件系统开发需要尽可能多的制衡，以确保优质的产品。一种行之有效的技术是让独立于开发团队的组织来进行确认和验证（V&V）。*确认（Validation）*是检查每个中间产品的过程。例如，确认（Validation）可确保：软件需求满足系统要求，高阶的软件设计可满足所有软件需求（且不满足其它需求），算法可满足组件的外部规格说明，代码可实现算法，等等。*验证（Verification）*是检查软件开发的每个中间产品以确保其满足需求的过程。

你可以将 V&V 视为儿童电话游戏的一种解决方案。在这个游戏中，让一群孩子连成一组，通过耳语传递一条特定的口头信息。在最终，最后一个孩子说出的他/她听到的内容，很少能与最初的消息相同。确认会使每个孩子问前一个孩子，“你说的是 x 吗？”。验证会使每个孩子问第一个孩子，“你说的是 x 吗？”。

在项目中，应尽早计划 V&V。它可以记录在质量保证计划中，也可以存在于单独的 V&V 计划中。在这两种情况下，其过程、参与者、操作和结果都应在软件需求规格说明被批准的大约同一时间批准。

参考文献

Wallace, D., and R. Fujii, "Software Verification and Validation: An Overview," IEEE Software, 6, 3(May 1989), pp. 10-17.



第九章 演变原则

EVOLUTION PRINCIPLES

*演变*是与修改软件产品相关的一系列工作，用于：

1. 满足新功能。
2. 更有效地运行。
3. 正常运行（当检测到原始产品中的错误时）。

原则 185 软件会持续变化

SOFTWARE WILL CONTINUE TO CHANGE

任何正在使用的大型软件系统都将经历不断的变化，因为系统的使用会使人想出新的功能。它会一直变化，直到从头开始重写变得更划算。这就是曼尼·雷曼（Manny Lehman）的“持续变化定律”（Law of Continuing Change）。

参考文献

Lehman, M., "Programs, Cities, and Students-Limits to Growth?" Inaugural Lecture, Imperial College of Science and Technology, London (May 14, 1974);
另见 Belady, L., and M. Lehman, "A Model of Large Program Development, "IBM Systems Journal, 15, 3 (March 1976), pp. 225-252.

原则 186 软件的熵增加

SOFTWARE'S ENTROPY INCREASES

任何经历持续变化的软件系统都会变得越来越复杂，并且变得越来越杂乱无章。由于所使用的所有软件系统都会发生变化（原则 185），并且变化会导致不稳定，因此所有有用的软件系统都将朝着较低的可靠性和可维护性迁移。这就是曼尼·雷曼（Manny Lehman）的“熵增加定律”。

参考文献

Lehman, M., "Programs, Cities, and Students-Limits to Growth?" Inaugural Lecture, Imperial College of Science and Technology, London (May 14, 1974);

另见 Lehman, M., "Laws of Program Evolution-Rules and Tools for Programming Management," InfoTech State of the Art Conference on Why Software Projects Fail (April 1978), paper #11.

原则 187 如果没有坏就不要修理它

IF IT AIN'T BROKE, DON'T FIX IT

当然，这个建议适用于生活的许多方面，但它特别适用于软件。就像其名字那样，软件被认为是可塑的、易于修改的。不要误以为软件中的“失灵”很容易发现或修复。

假设你在维护一个系统。你正在检查组件的源代码。你可能是想增强它，或者是想找到错误的原因。在检查时，你觉得自己发现了另外一个错误。不要试图“修复”它。很有可能你会引入而不是修复一个错误（原则 190）。相反，应记录并提交变更请求。期望通过配置控制和相关的技术评审来确定它是否是一个错误，以及应该以什么样的优先级进行修复。（原则 175，177，178 和 179）

参考文献

Reagan, R., as reported by Bentley, J., More Programming Pearls, Reading, Mass.: Addison-Wesley, 1988, Section 6.3.

原则 188 解决问题，而不是症状

FIX PROBLEMS, NOT SYMPTOMS

当软件出错时，你的责任是彻底理解错误的原因，而不只是草草分析一下，并对你认为的原因进行一个快速的修复。

假设你在试图定位软件故障的原因。你已经发现每次一个特定组件传输一个值时，它正好是期望值的 2 倍。一个快速而肮脏的解决方案是，在传输生成的值之前将其除以 2。这个解决方案是不合适的，因为（1）它可能不适用于所有情况，并且（2）它留给程序本质上两个相互补偿的错误，这会使得程序在未来实际上无法维护（原则 92）。更糟糕的快速和肮脏的解决方案是，接收者在使用之前，将它收到的值除以 2。这个解决方案有所有与第一个解决方案相同的问题，而且它会导致调用错误组件的所有未来组件接收错误的值。正确的解决方案是检查程序并确定为什么值总是加倍，然后修复它。

参考文献

McConnell, S., Code complete, Redmond, Wash.: Microsoft Press, 1993, p. 638.

原则 189 先变更需求

CHANGE REQUIREMENTS FIRST

如果各方都同意对软件进行增强，那么第一件事就是更新软件需求规格说明（SRS: Software Requirements Specification），并获得批准。只有这样，才比较有可能让客户、市场营销人员和开发人员对变更内容达成一致。有时由于时间限制无法做到这一点（这不应该一直存在，否则管理层需要阅读本书中的管理原则！）。在这种情况下，请先对 SRS 进行更改，然后再开始对设计和代码进行更改，并且在完成设计和代码的修改之前，批准对 SRS 的变更。

参考文献

Arthan, J., Software Evolution, New York: John Wiley & Sons, 1988, Chapter 6.

原则 190 发布之前的错误也会在发布后出现

PRERELEASE ERRORS YIELD POSTRELEASE ERRORS

发布之前错误就比较多的组件，发布之后也会发现比较多的错误。这对开发者来说是个令人失望的消息，但确实是被经验数据所充分支持的（而且由原则 114 可知，你在一个组件中发现的错误越多，将来也会发现更多）。最好的建议是废弃、替换、从头创建任何具有不良历史记录的组件。不要花钱填坑。

参考文献

Dunn, R., Software Defect Removal, New York: McGraw-Hill, 1984, Section 10.2.

原则 191 一个程序越老，维护起来就越困难

THE OLDER A PROGRAM, THE MORE DIFFICULT IT IS TO MAINTAIN

在对软件系统进行更改（无论是维修还是增强）时，系统中必定有一些组件要被修改。随着程序变“老”，每次改动时，整个系统中需要修改的组件的比例也会随之增加。每次更改都会使所有后续的更改更加困难，因为程序的结构必然会恶化。

参考文献

Belady, L., and B. Leavenworth, "Program Modifiability," in Software Engineering, Freeman, H., and P. Lewis, eds., New York: Academic Press, 1980, pp.26-27.

原则 192 语言影响可维护性

LANGUAGE AFFECTS MAINTAINABILITY

开发所使用的编程语言，会极大地影响维护期间的开发效率。某些语言（例如 APL，Basic 和 LISP）促进了功能的快速开发，但是它们本质上难以维护。其他语言（例如 Ada 或 Pascal）在开发过程中更具挑战，但本质上更易于维护。倾向于强制高内聚和低耦合（原则 73）的语言，例如 Eiffel，通常有助于开发和后续维护。级别很低的语言（如汇编语言）通常会在开发和维护期间抑制开发效率。可对照查看原则 99。

参考文献

Boehm, B., Software Engineering Economics, Englewood Cliffs, N. J.: Prentice Hall, 1981, Section 30.4.

原则 193 有时重新开始会更好

SOMETIMES IT IS BETTER TO START OVER

如今关于重建（reengineering）、翻新（renovation）和逆向工程（reverse engineering）的讨论太多了，我们可能都开始相信这样做很容易。这很难做。有时这很有意义，值得投资。其它时候这是对珍稀资源的浪费，从头开始设计和编码可能是更好的选择。举例来说，扪心自问，如果你制作了设计文档，维护者们真的会使用它们吗？

参考文献

Agresti, W., “Low-Tech Tips for High Quality Software”, Quality Time Column, IEEE Software, 9, 6 (November 1991), pp. 86, 87-89.

原则 194 首先翻新最差的

RENOVATE THE WORST FIRST

原则 193 建议，重新开始有时可能是最好的主意。另一个不那么痛苦的方法是，完全重新设计和重新编码“最差”的组件。这里“最差”的组件是指那些消耗了最多改正性维护费用的组件。Gerald Weinberg 报告说，在一个系统中重写一个 800 行的模块（占全部改正性维护成本的 30%），就可以为整体维护工作节省大量的资源。

参考文献

Weinberg, G., ”Software Maintenance”, Datalink (May 14, 1979), as reported by Arthur, J., Software Evolution, New York: John Wiley & Sons. 1988, Chapter 12.

原则 195 维护阶段比开发阶段产生的错误更多

MAINTENANCE CAUSES MORE ERRORS THAN DEVELOPMENT

维护期间对程序的修改（无论是改进功能还是修正缺陷）引入的错误远远超过最初的开发阶段。维护团队报告说，维护期间有 20% 到 50% 的改动会引入更多错误。

出于这个原因，遵守“规则”是如此重要：制定 SCM 计划（原则 174），控制基准（原则 179），并且不要绕过变更控制（原则 182）。

参考文献

Humphrey, W., “Quality From Both Developer and User Viewpoints,” Quality Time Column, IEEE Software, 6, 5 (September 1989), pp. 84, 100.

译者注

SCM，即 Software Configuration Management，软件配置管理。

原则 196 每次变更后都要进行回归测试

REGRESSION TEST AFTER EVERY CHANGE

回归测试，是在变更发生后，对所有先前已测试过的功能进行的测试。大多数人绕过回归测试，因为他们认为自己的变更是没有影响的。

你对一个模块的变更，可能是为了修正一个错误（改正性维护，corrective maintenance）、添加一个新的特性（适应性维护，adaptive maintenance），或是提升它的性能（完善性维护，perfective maintenance）。你必须验证你的改动是否能正确的运行。也就是说，你必须测试之前不能正确运行的部分，检查新的特性是否生效，或者是确认性能确实得到了提升。如果这些测试都通过了，就没问题了吗？当然不是！不幸的是，软件总会出现一些奇怪的问题。你还必须要做回归测试，来验证之前那些运行正确的功能，现在是否还能正常工作。

参考文献

McConnell, S., *Code Complete*, Redmond, Wash.: Microsoft Press, 1993, Section 25.6.

原则 197 “变更很容易”的想法，会使变更更容易出错

BELIEF THAT A CHANGE IS EASY MAKES IT LIKELY IT WILL BE MADE INCORRECTLY

这是 Gerald Weinberg 的“自我失效模型”（Self-Invalidating Model）原则，并且与原则 171 所述的更一般的情况密切相关。因为软件是很复杂的，要正确运行必须处于“完美”的状态，所以必须认真考虑每一个改动可能带来的影响。只要开发人员认为变更是简单的、容易的或不证自明的，他们就会放松警惕，忽视那些能帮助保证质量的手段，在大部分情况下就会执行不正确的变更。这会表现为一个错误的变更，或者一个意想不到的副作用。

为了避免这种情况，要确保你正在做的变更是经过核准的（原则 182 及 183 ），对每项变更进行核查（原则 97），并在每组变更后进行回归测试（原则 196）。

参考文献

Weinberg, G., *Quality Software Management, Vol. 1: Systems Thinking*, New York: Dorset House, 1992, Section 15.2.3.

原则 198 对非结构化代码进行结构化改造，并不一定会使它更好

STRUCTURING UNSTRUCTURED CODE DOES NOT NECESSARILY IMPROVE IT

假如说你必须维护一个以非结构化方式编写的程序。你可以机械地将其转换为等效的结构化代码，还具有相同的功能。这样的程序不一定更好！通常这种机械化的重构会导致同样糟糕的代码。相反的，你应该采用合理的软件工程原则，重新考虑模块，并从头开始重新设计。

参考文献

Arthur, J., *Software Evolution*, New York: John Wiley & Sons, 1988, Sections 7.2 and 9.1.

原则 199 在优化前先进行性能分析

USE PROFILER BEFORE OPTIMIZING

当需要优化程序以使其更快时，请记住 80% 的 CPU 周期将被 20% 的代码消耗（Pareto 定律）。因此，先去找到那些能够带来优化效果的 20% 的代码。最好的方法是使用任何市场上可买到的可用的性能分析工具。*性能分析工具*在你的程序运行过程中监控它，并识别出“热点”，也就是消耗最多 CPU 周期的部分。优化这部分。

参考文献

Morton, M., as reported by Bentley, J., *More Programming Pearls*, Reading, Mass.: Addison-Wesley, 1988, Section 6.4.

译者注

Pareto 定律，即常说的二八定律。

原则 200 保持熟悉

CONSERVE FAMILIARITY

这就是 Manny Lehman 的“熟悉守恒定律”（Law of Conservation of Familiarity）。软件产品的维护阶段，通常会采用增量版本方式来发布。每个新版本都包含一定数量的变化（即与早期版本中所熟悉的功能不同）。新发布的版本如果要包含大于平均水准的变化，往往会出现“性能差，可靠性差，故障率高，成本和时间都超支”的问题。此外，变化量高于平均水准越多，风险就越大。出现这种现象的原因，似乎是向用户发布软件的稳定效应（stabilization effect）。由于软件的变更往往会导致不稳定（原则 184 和 190），版本之间的大量变更可能会引起一定程度的不稳定，并且这种不稳定无法通过发版来弥补。此外，开发人员对产品的心理熟悉程度在多个版本之间会逐渐降低；也就是说，软件修改的时间越长，开发人员对它的“感觉”就越陌生。当产品发布后，开发者要重新学习才会再次感到熟悉。如果两个版本之间做太多的改动，就会对“不熟悉的”代码做太多的修改，进而影响质量。

总结：保持产品发布版本之间的改动量相对稳定。

参考文献

Lehman, M., "*On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle,*" Journal of Systems and Software, 1, 3 (September 1980), pp. 213-221.

原则 201 系统的存在促进了演变

THE SYSTEM'S EXISTENCE PROMOTES EVOLUTION

让我们假设一下，我们可以提前“完美”地完成需求规格说明。更进一步假设，在开发过程中需求没有变更，因此当系统被创建后，实际上满足了现有的需求。即使这些假设成立，演变仍然会发生，因为将这个系统引入到它要解决的问题的环境中，本身就改变了这个环境，也就会引发新的问题。

这意味着，无论你认为自己多完美地实现了需求，都必须为部署之后必要的变更做好计划。

参考文献

Lehman, M., "Software Engineering, the Software Process, and Their Support," *Software Engineering Journal, 6, 5*(September 1991), pp. 243-258.

主题索引

SUBJECT INDEX