第 2 章
Chaper 2

技术杰出

快　照

我们邀请你回顾软件工程的历史，从它被命名的那一年开始，即1968年。我们在寻找一些概念，时间已经证明这些概念是很重要的，另外还有其他一些思想，最终只是短暂流行。我们知道，构成敏捷软件开发基础的所有关键技术实践在软件工程的头5年里都出现了前身。我们发现，项目管理实践在这些年来一直被吹捧为“软件危机”的答案，但却得到了令人失望的结果。我们得出结论，在寻找经得起时间考验的系统开发实践时，应该主要关注技术参考取景框。

我们探讨了4个重要的技术概念，它们经受了时间的考验，并从处理软件基本复杂性的经典方法开始，即低依赖性的架构。我们转向软件开发的圣杯：发现一个过程，能够更早发现错误，而不是更晚。这让我们追溯到Edsger Dijkstra[⊖]和Harlan Mills，他们第一次提出了建构者（constructionist）的概念，这成为测试驱动开发和持续集成[⊜]的基础。我们将回顾演进式开发，它支持了因特网、个人计算机和开源软件的隐式革命。我们还会讨论专业知识，这可能是技术杰出的最重要的部分。在本章的最后，我们将介绍竞争力领导者的角色。

⊖ Dijkstra, “Structured Programming,” 1969。

⊜ Mills, “Top Down Programming in Large Systems,” 1971。

事实、短暂的流行和谬误

在19世纪40年代，170万移民来到了美国，这几乎是前十年移民数的3倍。许多人冲向了中西部的农场，大多数走的都是车辙路，直通到他们的家。在19世纪30年代刚经历了一场经济崩溃后，州政府没什么钱来修路和养路。然后在1844年，传言多伦多的某个人发明了一种新技术，一种更好、更便宜的修路方法。路面使用木板条（这在中西部地区很容易得到），成本只有砾石路的三分之一。根据广告宣传，木板条路将提供更平滑的骑乘，所以负载可以重一倍，旅行时间可以缩短一半。维护也是最小限度的，并且木板条可以用8到10年。投资回报率达到每年20%。最大的优点是，木板条路对社交有好处。邻居们可以互相拜访，家里人在坏天气时也能去教堂。

在接下来的几年里，木板条路修筑风行一时，每一条道路的完成都伴随着市民的欢呼。这个故事在中西部地区引发了极大的兴趣，新的公司成立了，成千上万的本地投资者拥有股份，在短短几年里，就修筑了1万英里的木板条路，并且每个人都对这种道路很满意。

但是没过多久木板条路就开始腐烂了 —— 也许是3至4年。腐烂的木板条陷住了马脚，让马车倾倒。但是通行费还不足以支付维修的费用，所以坏了的木板条路就被忽略了，而且变得越来越危险。人们开始逃避对坏道路的通行费，收入暴跌。到了1853年，木板条路的修筑完全停止了，这些公司开始放弃使用他们的道路。州政府接手了原有的道路系统，并转而使用碎石进行

维修。

“木板条路热是信息瀑布群的一个好例子，”[⊖]James Surowiecki在《The Wisdom of Crowds》中写道。开始的一些木板条路是极大的成功。人们寻找道路问题的解决方案，并找到了一个现成的方法。随着越来越多的人修筑木板条路，他们的合法性越来越受到保护，因而考虑其他解决方案的愿望缩小了。过了几年木板条路的基本弱点就变得明显了：它们不能坚持足够长的时间。

这些年来，我们在软件开发上已经看到了我们的木板条路。我们有好的想法、过度的宣传、最初的成功、风行一时，然后过了几年，得到的是破败的木板条。考虑这一段历史，我们猜想可能有许多现代的木板条路，那么我们怎么才能识别它们？为了回答这个问题，我们决定研究软件的历史，寻找一些模式，看看人们用怎样的取景框来看待关键问题，这样我们就可以看到哪些取景框只是一时的风尚。不必走得太远，我们的书架上就已经满是软件书籍，日期可以追溯到20世纪60年代。起初，我挑出了8本书，它们的标题中包含“结构化”这个词。我们现在不大能听到结构化编程了。“结构化”会是早期的软件行业的木板条路吗？

结构化编程

结构化编程是从1968年在德国Garmisch召开的NATO软件工程会议上开始的（这一点存在争议），一些软件开发界最了不起的人聚在一起，讨论“软件危机”。为什么创建可靠的大型软件系统有这么困难？Edsger W. Dijkstra是荷兰Eindhoven技术大学的教授，提交了一篇论文，名为《Complexity Controlled by Hierarchical Ordering of Function and Variability》[⊜]。在这篇论文中，他提出复杂的系统应该构建为一系列的抽象层次，或者是叠加的层次结构，其中每一层作为上面一层的虚拟机。

大约在与这个会议差不多的时间，Dijkstra发表了《A Constructive

⊖ Surowiecki, 《The Wisdom of Crowds》, 2004, 第3章。这段摘自第25页。当某个被认为是专家的人做出了一个决定，其他人不经过自己的分析就遵循这个决定时，就称为信息瀑布群。随着更多的人复制这个决定，它的可信度增加了。

⊜ Dijkstra, “Complexity Controlled by Hierarchical Ordering of Function and Variability,” 1968, 可以在http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html访问到。

Approach to the Problem of Program Correctness》[一]，这也许是他对软件业的贡献中最被误解的一项。在1975年，Peter Freeman总结了这个概念[二]：

> Dijkstra的第二个过程方面的建议更为精妙，似乎被许多想追随他的建议的人忽略了……Dijkstra建议采用下面的过程来构建正确的程序：分析如何证明一类计算可以满足需求，也就是说，如果我们想证明算法执行正确的话，就要明确说明必须保持的条件。然后编写一个程序，让这些条件得以实现。
>
> 因此结构化编程是一种思考方式，导致了建构式（constructive）编程，它不是一个可以精确指定的过程……要得到结构化编程的更明确的定义可能是有害的。我们不希望妨碍发现结构和技术的过程，这一过程帮助我们建构式地编程。

今天的层次化分层结构与Dijkstra的描述很类似，存在于许多系统的核心之中，从因特网协议到多层架构。但Freeman是正确的，层次化分层结构不是唯一的建构式编程的方法。Dijkstra的建构者（constructionist）方法是今天测试驱动开发的前身，在这种方法中，我们从描述或以示例说明代码必须做什么才算正确执行开始（也称为测试），然后编写代码，从而使每个示例都产生预期的行为。

Dijkstra这样总结他的哲学：“那些想得到真正可靠的软件的人，会发现他们必须从一开始就找到一些方法来避免主要的错误，这样会导致编程过程的成本变得更低。如果你想要更有效率的程序员，你会发现他们不会浪费时间来调试——他们应该从一开始就不会引入缺陷。”[三]这是软件开发的圣杯，所以结构化编程的目标得到了热情地拥抱。但编写无错代码的技术通常被曲解了，正如我们稍后看到的那样。

自顶向下编程

Harlan Mills是IBM院士，他怀着极大的兴趣阅读了Dijkstra的著作，并产生了深刻理解[四]。他试验了几种“从可证明性的观点”来编程的方法，像

[一] Dijkstra, “A Constructive Approach to the Problem of Program Correctness,” 1968。

[二] 参见Freeman, 《Software System Principles: A Survey》, 1975, 第489页。

[三] Dijkstra, “The Humble Programmer,” 1972。

[四] 参见Mills, 《Software Productivity》, 1988, 第3页。

Dijkstra建议的那样。Mills发现，为了一直知道程序是正确的，声明应该以一定的次序进行开发。声明的正确性不应该依赖于将来的声明。这意味着程序员应该采用一种递归的方式：从能工作的声明开始，添加一条新的声明，确保它们都能工作，然后添加另一条声明并确保3条都能工作，如此进行下去。他总结了这个思想：如果它适用于声明，就应该也适用于程序。所以，举例来说，打开文件的代码应该在访问文件的代码之前编写。具体来说，系统的任务控制代码应该先写，然后再增量式地添加程序，这与当时的实践刚好是相反的。Mills在一个为纽约时报开发的主要项目上测试了他的想法，他从编写系统的一个可行走的骨架开始，然后每次添加一个模块，每次添加后都检查整个系统仍然能工作。结果得到广泛报道——项目的品质和效率令人印象深刻，在集成、验收测试和第一年的运营中，几乎没有发现什么缺陷㈠。

Mills意识到，试图在集成时解决发现的差异是行不通的，所以他建议应该编写自顶向下的模块，并且一边写一边集成进一个系统，而不是最后再来集成。他的方法可能被称为“分步集成”，就是今天“持续集成”的前身。Mills注意到，使用这种技术的人体验到了集成过程的极大改进。“在软件开发的后期，根本就没有集成的困难。”㈡

但对于那时的大多数人来说，“自顶向下编程”是一个容易记住的短语，可以有许多种解释，很少能与Mills脑子里想的分步集成扯上关系。他写道，“我很高兴人们在使用这个术语。如果人们开始使用这个思想，我会更高兴。”Mills说他可以区分人们是否在使用这个思想：“判断是否实际上在进行自顶向下编程的首要依据就是有没有消除集成困难。对布丁的证明就在吃的过程中！”㈢

结构化编程后来怎么了

从1970到1980年，所有东西都必须是结构化的，就像结构化编程和它的变种是对软件开发中所有问题的答案那样。1982年，Gerald Weinberg抱怨说，“结构化编程”可以与“我们最新的计算机”互换使用，他是这样说的㈣：

㈠ Baker, “System Quality Through Structured Programming,” 1972, 第339页。

㈡ Mills, 《Software Productivity》, 1988, 第5页。

㈢ 同上, 第4页。

㈣ Weinberg, 《Rethinking System Analysis and Design》, 1982, 第21页。原书使用了斜体。

如果你在数据处理方面遇到问题，安装“我们最新的计算机”就可以解决这些问题。“我们最新的计算机”有更好的性价比，更容易使用。你的员工会爱上“我们最新的计算机”，但是当“我们最新的计算机”安装好以后，你就不需要那么多人了。转换费用？不是问题！有了“我们最新的计算机”，你最多只要几个星期就能意识到节省的费用。

正如我们看到的，结构化编程受欢迎有一个很好的理由：它以核心技术概念作为取景框，这些技术概念经过了时间的考验，包括层次化分层和分步集成。建构式编程走过了一条更为曲折的道路。人们设计了许多方法来构建正确性的证明，包括Hoare逻辑和Bertrand Meyer的契约式编程，但这些方法太难以广泛采用了。直到xUnit框架在20世纪90年代后期出现时，构建式编程才找到了一种实用的方法。

Dijkstra和Mills的理论（从可证明性的观点来编写代码和避免大爆炸式的集成）在当年是很难付诸实践的，特别是对当时的计算机和工具来说。没有对这些核心技术基础的好的实现，结构化编程就丧失了它的潜力，它没有活过那些过度的宣传，只是曾经红极一时。

面向对象语言

当Dijkstra和Mills从对代码的正确性建立信心的取景框来看软件问题时，David Parnas创建了一个新的取景框，关注正在出现和可能出现的更大的问题：软件维护。如图2-1所示，维护成本在1970年是所有计算机成本的一半，并在迅速增长之中。

Dijkstra和Parnas都理解软件的基本问题在于它的复杂性，以及人类的思维无法详细理解大型系统的复杂性。但当Dijkstra信赖层次化抽象，将复杂性组织成能够理解的层时，Parnas意识到这不是对复杂性问题的唯一解决方案。他同意Dijkstra的“分而治之”的理念，但它提出了一种新的判据，将系统分解为模块，从而使软件更易于维护。这个原则称之为“信息隐藏”[⊖]。

Parnas指出，用于系统分解的最常见的判据是程序的流程，但这种方法导致模块具有复杂的接口。更糟糕的是，典型的更改，如输入格式或存储位

⊖ Parnas, “On the Criteria to Be Used in Decomposing Systems into Modules,” 1972。

置的更改，都需要修改所有的模块。他提出了另一种分解系统的判据，即每个模块具备一些知识（或职责），能够完成单个设计决定，这些知识对其他模块都是隐藏的。而且，选择暴露出来的接口应该尽可能少地暴露模块内部的工作。后来的一些作者重新表述了这些判据：模块本身应该具有高内聚的特点，模块之间应该低耦合。

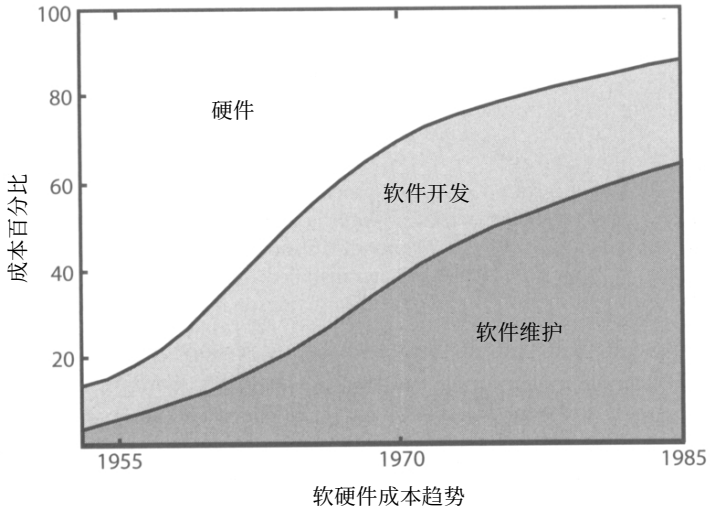

图2-1 不断增长的维护冲击[⊖]

1960年，Ole-Johan Dahl和Kristen Nygaard在奥斯陆的挪威计算机中心设计了一种编程语言Simula。Simula是第一种面向对象的语言，是专门为进行模拟而设计的。它引入了许多现在大家熟悉的概念，如对象、类、子类、继承和动态对象创建。在1972年，Dahl、Dijkstra和Hoare出版了一本书，名为《Structured Programming》[⊜]。当时，结构化编程被认为包括了面向对象，但随着时间推移，“结构化”这个术语缩小了范围，只指层次结构，而面向对象编程被排除在绝大多数结构化开发之外。

Smalltalk是在施乐的PARC实验室创建的，它以Simula为模型，是20世纪80年代流行的通用面向对象编程语言。除了它的忠实追随者，现在支持Smalltalk的人不多，先是被C++取代，在20世纪90年代后期又被Java取代。

⊖ 摘自Boehm，“Software Engineering,”1976, 第1226～1241页，使用得到许可。

⊜ Dahl、Dijkstra和Hoare，《Structured Programming: A.P.I.C Studies in Data Processing No. 8》, 1972。

1994年，在Parnas发表他的第一篇关于信息隐藏的论文20多年之后，他写道，“控制软件老化的第一步就是应用古老的谚语，‘为变化而设计’。从70年代早期以来，我们就已经知道如何为变化而设计软件。我们需要应用的原则有着不同的名称，如‘信息隐藏’、‘抽象’、‘关注点分离’、‘数据隐藏’，或最新的名称，‘面向对象’”㊀。

在那20年里，IT部门发现维护和集成的成本超过了所有其他成本。当他们发现层次化的结构并不一定能带来容易维护的代码时，就转向了面向对象开发。因为模块化的基础和要素是将来可能的变化，至少在理论上，面向对象的软件应该从设计一开始，就以易于维护的方式进行。

但是许多开发者发现面向对象的思考方式很难，需要花时间来学习如何编写构造良好的面向对象代码。而且，利用对象进行编程的好模式当时还没有形成，没有传播，也没有有效地使用。因此，虽然在理论上面向对象开发得到的代码应该易于修改，但在实践中，面向对象系统可能和其他系统一样难以修改，特别是当没有深刻理解信息隐藏并有效运用的时候。

高级语言

想一想在20世纪70年代，绝大多数系统都是用汇编语言写成的，使用宏来帮助实现一般的控制结构，这是很有趣的。这并不是因为那时候还不存在高级语言。在20世纪50年代中期，Grace Hooper设计了FLOW-MATIC，它是COBOL的前身，而John Backus则领导设计了FORTRAN。但是，那时候计算机的内存、存储和处理能力都相当有限，所以人们认为汇编语言是优化使用硬件所必需的。实际上，当Barry Boehm在1955年第一天做程序员时，他的主管告诉他，“现在听着。我们每小时要为这台计算机付600美元，而我们每小时才付给你2美元，我希望你根据这一事实来工作。”㊁在那种经济条件下，汇编语言有很大的意义。

但是随着计算机变得越来越大、越来越快、越来越便宜，软件的成本先是达到硬件的水平，然后快速超过了硬件的成本（参见图2-1）。目标从优化硬件转向了优化人员，高级语言开始得到广泛使用。大多数人预期这会导致

㊀ Parnas, “Software Aging,” 1994。

㊁ Boehm, “An Early Application Generator and Other Recollections,” 1997。

软件效率和品质的改进，因为高级语言让技术经验较少的人都能为计算机编程。开始的时候，事情大致就是这样发展的。

Dijkstra总结了高级语言带来的影响：“COBOL设计时的公开意图是这样的，它应该让专业程序员显得多余，因为它允许‘用户’用‘普通的英语’写下他想要的功能，每个人都能够阅读和理解。我们都知道，那个美好的梦想没有成真……COBOL不但没有消除专业的程序员，反而成为不断增长的程序员的主要编程工具。”㊀他解释说，高级语言消除了编程中的艰苦工作，使得这个工作更为复杂，要求才能更高的人，他们“像以前一样，仍旧愿意编写大段的、无法理解的代码，唯一的区别就是现在他们作品的规模更宏伟，高级的缺陷代替了低级的缺陷。”

后来，更高级的语言流行起来，如SQL、MAPPER和PowerBuilder等，你只要用这些名称替换掉Dijkstra说的COBOL，就能明白我们走了多远。就像木板条路一样，绝大多数高级语言会立即产生令人印象深刻的改进，成本很低，但要不了多久光彩就会褪去。虽然创建最初的应用相对比较容易，但通常对设计可维护的应用没有太大帮助。在很短的时间内（比预期的要短得多），最初的应用就开始老化了，维护成本比预期的要高得多。不像木板条路，高级语言可以提供持久的好处，但最长期的好处只能来自于稳固的技术方法，它们能管理我们所面对的复杂性。

生命周期概念

在20世纪70年代初期，编程和软件开发被认为是同一件事情。但到了70年代末，编程变成只是一个更大过程中的一步，这个过程名为“软件生命周期”。这发生在计算成本经历天翻地覆的转移时，正如我们在图2-1中所看到的。随着软件比硬件的相对成本快速增长，软件维护成本占据了成本增长的主要部分，人们开始寻找控制这种成本的方案。

在20世纪70年代中期，IBM的Michael Fagan发表了《Design and Code Inspection on Reduce Errors in Program Development》。㊁他报告说编程中有一半

㊀ Dijkstra, “Two Views of Programming,” 1975。

㊁ 参见Fagan, “Design and Code Inspection on Reduce Errors in Program Development,” 1996。Fagan发表了同一篇论文的不同版本，这里是最后的版本。

的时间都花在不增加价值的测试工作上，并声称在开发工作的前一半进行返工比在后一半进行返工要节省10倍到100倍的费用。和Dijkstra和Mills一样，Fagan发现解决办法就是尽可能早地发现并修复错误。但是，Fagan的方法不一样。他将开发视为3个过程：设计、编码和测试，每个过程都有输入和输出。在每个过程的末尾，他添加了一个正式审查的步骤，确保输出结果满足预先确定的退出判据。Fagan的工作帮助人们建立起了这样的思考方式，即将软件开发看成是一系列的“输入-处理-输出”的步骤，在每一步的末尾都有正式的验证。

Boehm扩展了Fagan的方法，强调了另外4个过程。在他1976年的文章㊀《Software Engineering》中，他画了一个经典的瀑布图，如图2-2所示㊁。Bohem称这个处理序列为“软件生命周期”。他引用了Fagan的工作和他自己的研究，说明减少成本的最大机会在于尽快找到错误，方法是在每个过程的末尾插入正式的审查。

分离设计与实现

生命周期概念的问题不在于它善于尽早发现缺陷，这一点当然是好的。问题有两方面，第一，大批量的工作会在每个过程步骤上排队，所以缺陷不会在进入时就被检测出来，而必须等到批处理的验证时才能发现。Dijkstra的想法是编程应该这样进行，在程序构建的每一步，人们都对程序的正确性有信心。Dijkstra消除软件开发中的缺陷的方法与生命周期过程是完全相反的：声明如果要证明代码正确就必须保证的条件，然后编写代码让这些条件为真。

生命周期概念第二个问题是分离了设计和实现。Dijkstra说，“老实讲，我看不出这些活动能够严格的分离，如果我们想做一份像样的工作的话……我相信产品的品质不能够事后再建立。一个软件的正确性是否能够保证，很大程度上取决于它的组成结构。这意味着让客户或你自己相信产品不错，这种能力交织在设计过程之中。”㊂

㊀ Boehm, “Software Engineering,” 1976。

㊁ 第一张“瀑布”图是Winston Royce发表的（Royce, “Manageing the Development of Large Software System,” 1970），在Poppendieck,《Lean Software Development: An Agile Toolkit》, 2003, 第24页中有讨论。Boehm1976年的论文再次强调了生命周期的概念，对它后来的流行产生了重要的影响。

㊂ Naur和Randell,《Software Engineering: Report on a Conference Sponsored by the NATO Science Committee》, 1968。

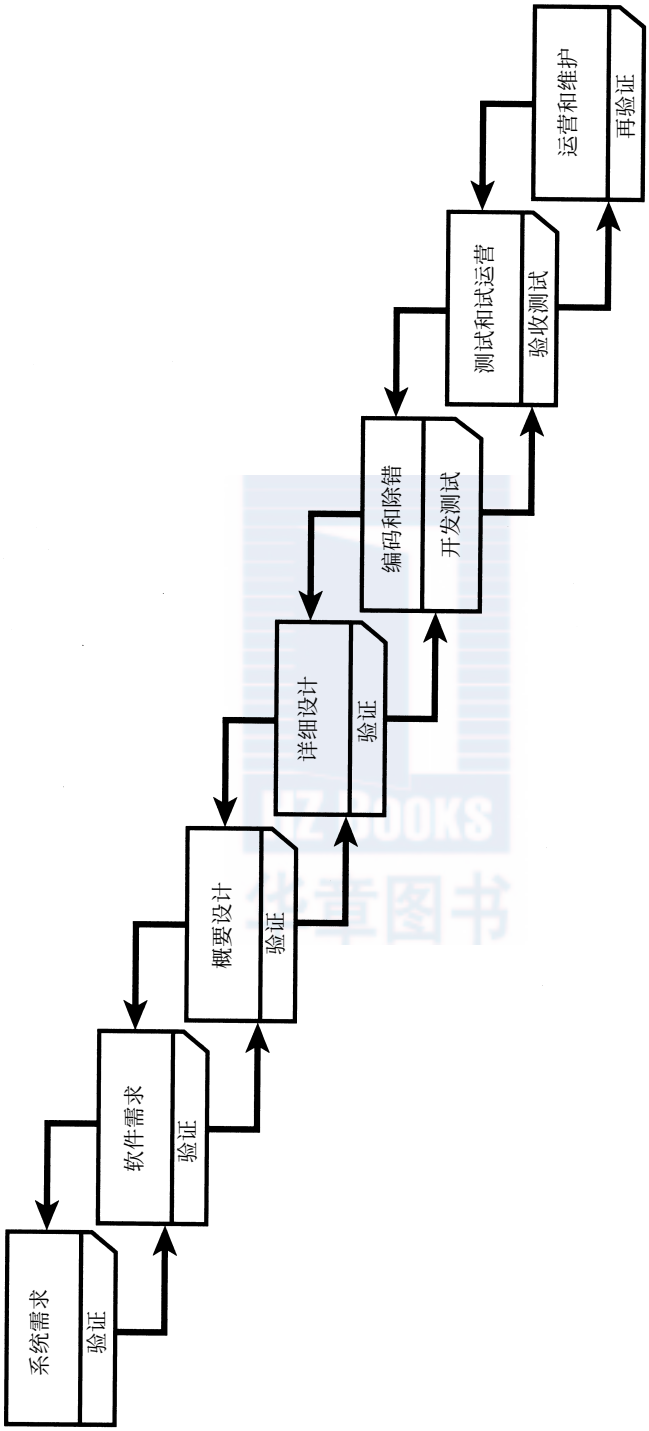

图2-2 Boehm的软件生命周期的最初版本⊖

⊖ 摘自Boehm, “Software Engineering,”使用得到了许可。

Dijkstra不是唯一认为不应该分离设计和实现的人。1982年，William Swartout和Robert Balzer（他们一度认为瀑布式开发是个好主意）在《On the Inevitable Interwinding of Specification and Implementation》[⊖]中写道：

> 一些年以来，我们和其他人一直在小心地指出，分离规格说明和实现有多么重要。按照这个观点，你先用正式的语言完全说明一个系统的规格，这种说明的抽象层较高，与实现方式无关。然后，在一个分离的阶段中考虑实现问题，编写出实现规格说明书的程序……
>
> 所有当前的软件开发方法学都采用了一个共同的模型，即分离规格说明和实现。不幸的是，这个模式过于幼稚，不符合现实。规格说明和实现实际上是紧密交织在一起的。

生命周期概念是有害的

没过多久，就听到了对生命周期概念的严肃而强烈的抗议。抗议来自Dan MacCracken（计算机语言教科书的作者、ACM的前主席）和Michael Jackson（以对象框架来进行系统开发方法的发明人）。1982年，他们在*ACM SIGSOFT*发表了《Life Cycle Concept Considered Harmful》。他们说[⊜]：

1）任何形式的生命周期都是强加给系统开发的一种项目管理结构。主张生命周期进度计划或其变种可以适应于所有系统开发，要么停留在事实的表面，要么假定生命周期只是初步而空洞的概念。

详细描述生命周期（类似于Bohem的）似乎是管理大型项目，而开发工具又不足时的唯一可行的方法。（这似乎是唯一选择，虽然不能防止许多这样的项目的失败。）

2）生命周期概念到目前为止保持了整个行业的失败，不能在最终用户和系统分析师之间建立起跨越沟通鸿沟的有效桥梁。在很多方面它限制我们进一步思考，而只是将过去的失败套进一个模子里。它忽略了一些因素，诸如：

人们逐渐认识到系统需求不能事先说明，甚至在原则上也不行，因为用户事先不知道它们——甚至在原则上也不知道。开发过程本身改变了用户对

⊖ Swartout和Balzer, “On the Inevitable Interwinding of Specification and Implementation,” 1982。

⊜ McCracken和Jackson, "Life Cycle Concept Considered Harmful," 1982。在这里使用得到了许可。

可能性的看法，深化了他们对应用环境的看法，而实际上往往改变了环境本身。我们要么承认这个事实，要么忽略这个事实。

3）生命周期概念是僵硬的思考方式，因此在系统要响应变化时，效果就很不好。我们都知道系统及其需求不可避免地会随时间变化。过去，受到不足的设计和实现工具的严格限制，我们的选择很少，只有尽早冻结设计，比用户期望的要早得多，随之而来的是不得不面对的大把的需求变更……将这个（生命周期的）概念强加于正在涌现的新方法，在我们看来是令人悲哀的短视行为，这些新方法对可能的变更有着很好的响应。

演进式开发

1981年，Tom Gilb发表了《Evolutionary Development》㊀一文。他指出人们极为渴望一种理论，指导演进式系统的设计和开发，但关于这一主题的文献不多，只有来自于IBM Federal Systems division的Harlan Mills的一些文章。Gilb提出不要开发详细的需求，人们应该从简短的、可测量的系统业务目标开始（他称为按目标设计）㊁，并进行增量式的交付。他指出增量式的交付不是原型，而是为人们带来真正价值的真正的软件。

同时，在没有人引起注意的时候

正如Gilb指出的那样，在20世纪80年代，绝大多数计算机文献都关注于语言和生命周期，而软件开发所取得的一些最有趣的进展虽然正在发生，却几乎没有得到公司IT部门的注意 —— 这些进展在ARPANET、因特网和个人计算机方面（参见表2-1）。

表2-1 因特网和PC软件演进的一些里程碑

年份	里程碑
1977	Apple II
1978	WordStar
1979	VisiCalc
1980	MTP(SMTP的前身)
1981	IBM PC
1982	Lotus 1-2-3

㊀ Gilb, “Evolutionary Development,” 1981。

㊁ Gilb, “Using Design by Objectives Tools(DBO),” 1984。

（续）

年份	里程碑
1983	TCP/IP标准化
1984	POP（邮局协议）
1985	PageMaker
1986	LISTSERV
1987	HyperCard
1988	第一个因特网病毒
1989	http（互联网）
1990	因特网商业化
1991	首次发布Linux

在20世纪80年代，有用的PC软件大量涌现，受到了低成本PC和创业氛围的支持。创业公司找到了一些方法来定期升级和维护软件包，从而获取更多的收入。因为软件包基本上彼此互不依赖，PC的拥有者就可以从大量程序中挑选，它们可以装在同一台计算机上，绝大部分不会影响其他程序。广泛成功的领域特定的语言（电子表格）成为了杀手应用，让许多公司都买了PC。

同时，因特网的开发是由非商业的力量推动的，在20年间，政府和学术机构合力开发了ARPANET，作为学术和军方通信的平台。只是在1990年允许商业使用之后，商业力量才接管，那时电子邮件已经发明，ARPANET的低依赖架构已被证明是极其健壮的。

“因特网？那不是软件”我们在20世纪90年代末还常常听到人们这样说。像许多颠覆性的技术一样，因特网在很长一段时间内被许多主流软件开发者忽略了，因为它打破了流行的软件模式。正如Wiki的发明者Ward Cunningham所说的，“因特网的确喜欢文本。它不想到处发送对象，它想发送文本。”[⊖]传送文本页面的东西怎么会是软件？

电子表格和电子邮件也没有被认为是软件，开源软件在许多年时也被绝大多数人忽略。但当第四代语言基本上无法做到“让专业程序员的编程成为多余”时，因特网、个人计算机，以及20世纪90年代的开源软件，成功地将大量的编程转移到了用户身上。

这些都是颠覆性技术的典型例子，这些技术不是去解决普遍存在的问

⊖ Cunningham, “Wiki Inventor Ward Cunningham with John Gage,” 2006。你可以在www.youtube.com/watch?v=bx6nNqSASGO找到这段信息丰富的视频。引用的这段话在这段视频的1:02至1:05处。

题，而是发现没有得到满足的低端需求环境，在这些环境中，现有的技术太昂贵或太麻烦，根本不能考虑。在这种小环境中建立了立足点之后，颠覆性技术就成长得越来越强大，最终能够完成原有技术完成的工作，而且成本更低、尺寸更小、用电更省、更为简单，总之是更好。这就是颠覆性技术随时间发展的本质。因此，因特网、PC软件和开源软件都是成功的演进式开发的案例。

为什么它有效

是什么让PC、因特网和开源软件在主流软件开源发的雷达之下，创造了大量成功的软件呢？

因特网

1）技术愿景：也许没有人具有J. C. R. Licklider这样的愿景，他认为计算机是让人们进行更有效沟通的一种方式。他提出构想，筹集资金，设法进行了ARPANET的早期开发。还有许多其他的愿景，伴随在Licklider左右或追随在他身后。

2）健壮性：ARPANET的设计目标是提供不同类型计算机之间的可靠通信，即使该网络的某一部分失效。对这个问题的创新解决方案就是从一台计算机发起一个数据包，让它根据数据包的地址，找到通往另一台计算机的路，不需要任何集中式管理或控制。

3）自然出现的标准：电子邮件最早是作为ARPANET上各种计算机的管理员之间的一种通信机制。随着新的管理员和不同类型的计算机加入这个网络，邮件协议就进行修改，从而容纳新来的用户。几年之后，电子邮件头的基本元素自然就出现了。

PC

1）最小化的依赖关系：早期的PC软件由专注于某问题的应用构成，应用与应用之间没有依赖关系，所以程序必须共存，而且对彼此都不会产生损害。人们很快形成了这种共同预期，即人们在为PC添加一个新程序时，不应该产生任何不希望的后果。

2）分阶段开发：PC当时是相对比较便宜的，第一批提供PC程序的是资金有限的企业家，因此，早期的程序只是初步的，目标是产生收入，并从市

场得到反馈。如果软件包成功，在大约一年内就会跟进推出新版本。随着时间的推移，人们预期有用的软件会定期推出新版本。

3）试验：早期的PC软件市场是一个巨大的试验。例如，Intuit的流行软件Quicken在1984年发布时，是PC上发布的第47个财务软件包。10年以后，因特网经历了同样高水平的商业试验。James Surowiecki解释说，所有正在自然形成的市场都是以这种方式进行的，他写道，“似乎没有系统擅长事先选择优胜者……系统成功之道在于能够产生许多失败者，然后意识到这些失败，并打败这些失败。”⊖

开源软件

1）个人投入：编写开源软件的人写出软件，解决他们自己遇到的问题，并贡献给社区。开发者对他们的问题理解得很好，对解决方案的完整性也有切身的关注，因为他们会用它。此外，他们的工作会开放给公众检查。

2）共同目标：开源的贡献者关注他们的工作，因为他们共同承诺实现一个共同目标。他们不断协作，在全球的范围内，通过简单的文本消息工具、简单的规则，以及代码集本身。

今天，这8项因素的每一项仍然是创造有价值的软件密集型系统的核心因素。

插曲

到了20世纪90年代中期，PC软件已经成长起来了，因特网带着各种可能性爆发了，开源软件开始有用了。但回到公司IT部门的世界，一个严重的威胁开始吸引了大量的关注和金钱，它就是Y2K。人们非常担心所有的软件已构成了这个世界的依赖，而它们在日期从1999年变到2000年时，会错误百出。所以IT部门没有去关心那些正在自然形成的技术，而是投入大量的工作来检查在已有应用中是否存在问题，或者在世纪之交时替换掉关键软件。等到Y2K过去之后，这些IT部门有时间上来透口气了，这时软件开发的世界已经永远改变了。

⊖ Surowiecki,《The Wisdom of Crowds》, 2004, 第29页。

Y2K

我在20世纪70年代和80年代写了大量的代码，大部分是在控制硬件的微型计算机上。在那些日子里，微型计算机没有一种可靠的方式来确定日期，当微型计算机启动时，人们必须为实时时钟输入日期，因为它在断电后不会保存日期。所以，没有思维正常的程序员会在代码中添加日期依赖关系，我们只会使用时钟来建立相对时间，而非绝对时间。所以我们相当肯定，绝大多数这种老式的微型计算机会没有损失地通过这个神奇的日期线。我比大多数人都要安心。

Mary Poppendieck

敏捷的未来

当我们回顾软件开发的历史时，我们看到了一个一致的目标：避免在开发过程的后期才发现缺陷的高成本，要在开发过程中尽可能早地发现它们。但是关于如何实现这一目标，我们看到了极为不同的观点。一方面，我们有Dijkstra和Mills这样的人，他们从建立正确性的取景框来看问题，认为这是系统开发的一项不间断的活动。另一方面，我们有生命周期的概念，认为建立正确性是一项项目管理活动。我们回想起Daniel McCracken和Michael Jackson在1982年说过，“任何形式的生命周期都是强加给系统开发的一种项目管理结构。”从历史上来看，我们看到这两者常常混淆起来（参见表2-2）。

表2-2 历史经验

我们不要混淆	
系统开发	项目管理
成功 = 完成系统的总体目标	成功 = 实现计划的成本、进度和范围
学习/反馈循环	有退出判据的阶段
通过构建保证品质	通过审查保证品质
封装复杂性	管理范围
有技能的技术领导者	资源管理
技术原则	成熟度水平
从历史上看很健壮	从历史上看很脆弱

软件系统开发的历史为我们提供了一些技术概念，如层次结构、尽早集成（持续集成的前身）、信息隐藏（面向对象编程的前身），甚至是构建式编程

（测试驱动开发的前身），这些概念在今天看来都是十分重要的。我们看到，简单、低依赖性的架构和有经验的技术实现是成功软件开发中不变的因素。对于开发好的软件来说，这些专注于技术的实践的重要性从未降低——虽然它们偶尔会被人们遗忘。当它们被重新发现时，它们被证明和从前一样有效，甚至在有了现代工具后变得更有效。这些技术概念随着时间的推移得到了加强。

另一方面，当我们回顾软件开发中项目管理的历史时，我们看到了那些经不起时间考验的实践。确保需求正确之后再进行下一步，结果是创建了大批的未测试的代码。分离设计和实现的阶段式开发体系是无法工作的，因为它们不允许反馈。如果系统开发时间超过了环境中可能发生变化的时间，或系统本身将会改变环境，阶段式的开发体系就不能工作了。在每一个阶段末尾处等待测试违反了一项要求，即品质是在一开始构建到软件中的。成熟度级别不一定能证明是软件开发能力的指标。

那么敏捷软件开发在这幅图画中处于什么位置？我们猜想敏捷软件开发的这些方面存在于项目管理的取景框之中，有可能随时间而改变，因此是一个选择问题。另一方面，敏捷开发在系统开发的取景框下，扩展了许多实践。这些技术实践经受了时间的考验，它们在未来将继续长期有效。所以如果你在应用敏捷软件开发（或者你还没有用它），请从系统开发取景框开始。这是基础，而不是一时的流行。

取景框5：基本复杂性

软件的复杂性是一项基本属性，不是偶然的属性。开发软件产品的经典问题来自于这一基本复杂性，以及复杂性随规模增长而出现的非线性增长。

——Fred Brooks[⊖]

软件开发的历史就是尝试征服这种复杂性的历史。但在Fred Brooks的地标性文章《没有银弹》中，他解释了不存在能够终结“软件危机”的银弹，因为软件具有与生俱来的复杂性。他这样写道[⊜]：

爱因斯坦反复争辩说自然一定存在简单的解释，因为上帝不会反复

⊖ Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering,” 1986。

⊜ 同上。

无常。没有这样的信仰可以安慰软件工程师，他要掌握的许多复杂性是任意而复杂的，没有节奏和原因，必须满足许多人的惯例，必须符合许多系统的接口。这些接口彼此不同，由不同的人设计，不是上帝设计的。

分而治之

在历史上，关于如何处理软件与生俱来的复杂性不存在太多争议，每个人都同意答案就是分而治之。真正的问题是“按什么线来分割软件，以便征服它的复杂性”？

Edsger Dijkstra相信，控制复杂性的最好办法就是创建层次化的结构，每一层将下一层作为一个虚拟机。我们可以称之为“按结构划分”。Dave Parnas指出这样的划分可能在一开始是可行的，但对于可维护的软件来说不是正确划分判据，所以他主张“按职责划分”。最后，按职责划分被证明是设计可维护代码时最有价值的技术，尽管许多情况下人们使用分层的架构作为重要的简化技术。

Barry Boehm推荐我们“按过程划分”，在每个过程之后都有验证步骤。但是，这种方法在处理软件与生俱来的复杂性时没有效果。Tom Gilb推荐我们“按价值划分”，这种方法经受住了时间的考验。

“分而治之”是基本策略，但在实践中，确定最好的划分坐标不是那么简单。重要的是设计一种架构，能够有效地处理“领域上下文环境”中的复杂性，并非是对所有领域都一样。因此，要想长期征服复杂性，就要从一个架构开始，它适合这个领域。例如，请考虑因特网的架构。

因特网架构自然形成

1983年1月1日，有12年历史的ARPANET架构声明作废。希望留在该网络中的200个站点必须切换到新TCP/IP协议上。最初的ARPANET架构依赖于接口消息处理器（IMP，那时候已经是老化的计算机）和网络控制协议（NCP，已经达到了它的极限）。为了克服最初协议的限制，TCP/IP的目标是创建一个网络组成的网络，因此才有了Internet这个名称。新的协议有4个主要设计目标：[⊖]

⊖ Leiner等，“The Past and Future History of the Internet,”1997。

- 网络连接：每个独立的网络必须自我维持，这样的网络在连接到因特网时，不要求任何内部的改动。
- 错误恢复：通信会尽最大努力实现。如果一个包不能达到它的目的地，就会很快从出发源重发。
- 黑盒设计：应该用黑盒（后来被称为网关或路由器）来连接这些网络。网关不会保存任何有关通过它们的单个包流的信息，保持简单，避免从不同的失败模式中进行复杂的调整和恢复。
- 分布式：没有运营层面上的全球控制。

这些设计目标在宽度和简单性方面极为出色，基本上，因特网的底层架构表达了这4项设计原则。设计者Robert Kahn和Vinton Cert最初认为TCP/IP协议最终会被改造。但当它开发并部署时，“它就一直扩张个不停。”[⊖]结果，正如他们所说的，就是历史。

低依赖性架构

随着软件行业的成长，情况变得很明显，大量的金钱都花在了软件维护上，也就是说，修改已有的系统。据此可以得出一个符合逻辑的结论，好的软件架构[⊜]必须支持长时间的变化。这意味着，就像Dave Parnas在1972年所说的，好的架构有一种划分策略，将可能一起变化的东西放在同一个子系统中（高内聚），使子系统不会受到别的子系统变更的影响（低耦合）。随着时间的推移，这一点变得非常清楚，如果遵守这些规则，修改就会很容易，成本很低。好的架构必须是低依赖性的。

不存在低依赖性架构的替代品，无论多少计划或预见都不能取代它。使依赖关系最小化（或打破）应该具有最高技术优先级，在希望使用敏捷软件开发的组织机构中尤其是这样。我们曾看到过一些公司试图使用敏捷实践来维护一些包含大量依赖关系的整体式代码，但很少看到这种方式获得成功。

⊖ Stewart,《The Living Internet》, www.livinginternet.com/i/ii_cerf.htm, n.d。

⊜ “‘架构’是个许多人都试图定义的术语，少有一致意见。存在两个通用元素：一是系统的最高层次的分解，成为组成部分；另一点就是一些很难改变的决定。Fowler,《Patterns of Enterprise Application Architecture》, 2003, 第1页。”架构表达了形成系统的重要设计决定，其中重要性是由改变它的成本来测量的。Grady Booch, www.handbookofsoftwarearchitecture.com/index.jsp?page=blog&part=2006。

为什么？在敏捷开发中，变更是受鼓励的，而不是要避免的。但对于紧密耦合的架构来说，变更既有风险，成本又高。因此，在紧密耦合架构的大型系统中，尝试敏捷开发会产生一种不协调，很难解决。

紧密耦合的经典理由就是为了效率：更少的接口应该意味着更少的执行指令和更少的代码跳转。David Parnas在他1972年的论文中指出，信息隐藏会需要更多的处理能力，因为程序执行流经常在模块之间转移。㊀但当你比较猛涨的维护费用、处理能力的快速增长和不断下降的价格时，紧密耦合的理由就没有任何意义了。今天，很少有借口放弃易于修改的便利，而选择执行的速度，只有某些嵌入式系统或网络密集型的软件例外。

如果你还在与紧密耦合的架构搏斗，请从因特网的故事中学习一下，努力工作去还这些技术债。㊁但是，不要花大量的时间去详细定义一个新的架构，毕竟，因特网的底层架构可以由4条深思熟虑的规则来表达。如果你花大量的时间来开发一个架构框架，世界在你实现这个框架之前早就变化了。确定牢固的、低依赖的技术目标，然后朝着这个方向去演进。

如果你们需要达成一致，就失败了

Amazon.com㊂就是增长。回到20世纪90年代后期，当公司刚刚起步的时候，它没太担心伸缩性问题，所以Amazon的系统变成了一个大的前端和一个大的后端，这个架构在当时被认为是最佳实践。但每到假期，我们就几乎被赶下悬崖。到了2000年，公司意识到它目前的架构严重地阻碍了增长，必须做些不同的事情。

所以Amazon改变了它的架构。实际上，再也没有像数据库这样的东西了，只有封装了数据和业务逻辑的服务。没有从服务以外直接访问数据库的情况，服务之间也没有数据共享。有的只是成百上千个服务，以及许

㊀ Parnas, “On the Criteria to Be Used in Decomposing Systems into Modules,” 1972。

㊁ 关于设计的经典参考是Evans,《Domain-Driven Design: Tackling Complexity in the Heart of Software》, 2003。关于处理遗留代码的经典参考是Feathers,《Working Effectively with Legacy Code》, 2004。

㊂ 这段故事的信息来自以下来源：Vogels, “A Conversation with Werner Vogels,” 2006; Roseman, “Working Backwards & 2-Pizza Teams,” 2006; Vogels, “The Amazon.com Technology Platform: Building Blocks for Innovation,” 2007; Vogels, “Availability & Consistency,” 2007; Vogels, Keynote, 2008。

多应用服务器，它们负责集成来自服务的数据。

为了让这个架构能工作（能伸缩），服务必须分解为小的、自治的构建块。很像因特网，没有可能失败的中央控制，服务都在局部运行。它们只根据局部信息来做决定，所以不论整个系统发生了什么情况，它们都能保持运行。这种分布式架构的理由很简单：CTO Werner Vogels说，“如果你需要在高负载的情况下做一些可能会出错的事，而你们又需要达成一致，就失败了。”

猜猜发生了什么情况？因为架构是分布式的，所以组织也可以是分布式的。Amazon发现每个服务都可以有各自的自治团队，负责做所有的工作——客户交互、决定开发什么、选择工具、编程、测试、部署、运营、支持等所有事情。没有工作需要传递，服务与其他服务之间通过文档清楚的接口进行交互，在请求的级别上达成一致。

CEO Jeff Bezos发现团队的规模应该有所限制，以2个比萨饼能喂饱为宜——可能是8到10人。Amazon.com有许许多多的2个比萨饼团队，每个团队完全拥有一个服务，从服务的诞生直到死亡。如果架构的特征对2个比萨饼团队太大，Amazon倾向于将这个特征分成更小的部分，因为团队的效果和架构一致性的效果同样重要。团队长时间呆在一起（每个成员需要承诺至少2年的工作时间），对他们所服务的所有方面负有长期的责任。

Amazon花了几年时间演进到新的架构，并学会了如何让它工作。当然，依赖关系仍然存在，公司开发了一些精致的依赖性管理工具。配置管理是一项挑战，因为正在测试，但这些挑战为创新解决方案提供了机会。综合各方面来考虑（包括技术方面和组织机构方面），低依赖关系的架构效果好得令人吃惊。

Marry Poppendieck

Conway法则

“设计系统的组织机构会受到一种限制，即产生的设计会复制这些组织机构的沟通结构”，Mel Conway在1968年写道[⊖]。他接着说，设计者第一次想

⊖ Conway, “How Do Committees Invent?”, 1968。

到的架构几乎不可能是最好的设计，所以公司应该同时准备好改变它们的产品架构和组织结构。

高依赖性架构为什么如此难处理，一个原因是它们通常反映了组织机构的结构，而改变组织机构的结构通常不是可以考虑的选择，但是应该改变。我们经常发现一些公司努力在一些职能团队中实现敏捷实践，也就是说，按技术路线组织的团队。例如，可能是一个数据库团队，大主机团队，一个Web服务器团队，甚至是一个测试团队。这些技术（或职能）团队总是发现，他们的工作依赖于其他几个团队，所以交付最小的特征集都需要大量的沟通和协调。沟通的复杂性增加了时间，也增加了出错的可能性，增加的开销成本使小的增量版本成为不切实际的事情。⊖

“我们的架构太复杂，不适合跨职能团队”

“我是一个Scrum团队的产品拥有者，”她说。“我们有几十个团队，每一个都有自己的产品拥有者。每次冲刺我决定我的团队应该做什么，但我总是要其他3、4个团队完成一些工作，我才能完成自己的工作。我去找那些产品拥有者，他们也有他们自己的承诺，可能要花3、4个冲刺才能有空实现我的要求。我必须对3、4个团队做同样的事，所以基本上，我在一个冲刺中不能完成任何事情。实际上，做任何有用的事情可能都要花上几个月的时间。”

“听起来你的团队是按技术分层来组织的。”我回答到。

“是的，”她说。“每个团队负责一个技术层的一部分。所以如果我需要另一层的某些东西，这是常有的事，我就必须让另一个团队去做。而按照我们的结构方式，其他团队没有什么动机来帮助我完成工作。”

“你们考虑过跨职能团队吗？你可以想象一个团队，具备实现一个完整特征的所有技能吗？”我问道。

“不，这不可能。”她的老板回答了我的问题。“我们的架构太复杂，不适合跨职能团队。代码的许多部分只有几个人能懂。我们不能够将这些人散布到所有团队中。”

⊖ 参见Larman和Vodde,《精益和敏捷开发大型应用指南》, 2009, 第3章（该书已由机械工业出版社于2009年出版，ISBN: 978-7-111-28449-9）, 深度探讨了这个主题。

“你们是不是能够让这些专家提供咨询，帮助团队中的人，这样可以在他们的仔细检查下修改他们的代码？他们也会传播他们的知识，同时加速开发。”我建议道。

“不，这是不可能的。我们的系统有一段时间了，现在它是一些相互锁定的应用结成的网。”他说。显然，这也是他的组织机构的结构。

在这样的组织机构中实施Scrum不会有太大帮助。经理们曾经希望Scrum是加速开发的一种简单方法，但他们忽略了这背后需要合理地调整他们的复杂技术架构和组织架构。我猜他们只有解决了高依赖关系的架构问题，才能够建立起特征团队，快速并独立地交付客户关心的特征。

Marry Poppendieck

敏捷开发很难在高度依赖性的环境中发挥作用，因为各团队不能够独立地做出并实现承诺。敏捷开发最适合跨职能的团队，单个团队或几个团队拥有向客户交付有用特征集所需的技能和授权，不依赖于其他团队。这意味着只要可能，团队就应该按特征或服务的路线来建立。当然，如果这些特征或服务相对比较小，而且有清晰的接口，效果就最好。Conway法则少有例外。

我们在组织结构上看到的另一个大错误，就是分离软件开发和该软件支持的更大的系统。例如，在涉及硬件和软件时，我们常常看到分离硬件和软件（有时候是固件）团队，这类似于按照技术分层路线来建立团队。Conway法则说，最好是组织起团队，让他们既负责硬件又负责软件，作为产品的一个子系统。

例如数码相机，它带有一个画面捕捉子系统，负责设置曝光、捕捉画面和处理画面。它还有一个存储子系统，负责从存储卡中存取照片。还有一个电子接口子系统，负责将照片传送给计算机。可能还有一个智能透镜子系统。每个子系统都是软件密集型的，但照相机不应该划分成硬件子系统和软件子系统。为什么？假定对照相机的聚焦子系统进行变更。如果按子系统划分团队，这项变更只涉及画面捕捉子系统。如果按软件硬件划分团队，这项变更会涉及两个团队，显然不会是少数人的工作。当变更限制在较小的子系统中时，变更的成本和风险就会显著减小。

类似的，如果业务过程需要更改，最好是让软件开发者加入业务过程再

造团队，而不是分离出软件开发团队。这并不意味着软件开发人员不再有技术大本营和技术同事、技术管理层。我们后面会看到，保持技术竞争力和好的技术领导是很重要的。组织机构上的挑战是要提供有技术背景的人，即使他们需要在跨职能的团队中工作。我们将在本章结束时再来讨论这个挑战。

取景框6：构建品质

如果你想要更有效率的程序员，就会发现他们不应该浪费时间来调试——他们从一开始就不会引入缺陷。⊖

—— Edsger W. Dijkstra

实际上所有的软件开发方法学都有同样的目标：通过尽早发现错误和立即改正错误来降低改正错误的成本。但是，正如我们前面所讨论的，不同的方法学看待这个目标的取景框是截然不同的。串行式生命周期方法打算创建一个完整、详细的设计，这样就能在进行编码之前，通过审查来发现缺陷——因为他们认为在编写代码之前修改设计缺陷，比在后来修改设计缺陷的成本更低。这个理论有一个根本问题：它分离了设计和实现。仔细读一下1968年NATO会议的文章，就会发现分离设计和实现不是好主意，当时甚至还没出现“软件工程”这个术语。⊜

但事情变得更糟糕。串行式开发将测试放到了开发过程的末尾——完全是寻找缺陷的错误时间。毫无疑问，在开发循环的后期来发现和修复缺陷是代价高昂的，这是串行式开发最开始的前提。然而，串行式方法刚好引入了他们想要防止的结果：将测试延迟到缺陷进入代码之后很久才进行。实际上，在串行式过程中工作的人们预期会在最后的测试中发现缺陷，三分之一或更多的软件发布周期留给了发现和修复缺陷，并在“理论上”完成编码之后。讽刺之处在于，串行式过程的设计目的是尽早修复缺陷，但在大多数情况下，最终的结果却是在很晚才发现和修复缺陷。

我们怎么会觉得在开发过程的最后再来发现和修复缺陷是合适的做法

⊖ Dijkstra, “The Humble Programmer,” 1972。

⊜ Naur和Randell,《Software Engineering: Report on a Conference Sponsored by the NATO Science Committee》, 1968。

呢？我们知道的每一种教授的品质保证方法都是刚好相反的，也应该知道最好是在开始就避免缺陷，或者至少在缺陷进入代码后就立即发现。这和丢失钥匙的情况是一样的：如果你在几秒钟前丢失了钥匙，向下看就行了。如果你在几分钟前丢了钥匙，原路返回寻找。如果你在几周前丢失了钥匙，就要换锁。[㊀]但我们仍然等到开发过程的末尾才来测试我们的代码，那时候我们需要很多的锁匠。为什么我们这么做？

推迟测试的部分原因是因为测试在历史上是劳动密集型的，而且难以对不完整的系统进行。有了精益的思考方式，我们不再能够接受这种历史的思维方式，而必须改变它。实际上，改变历史上对测试的思维定式，是我们看到的成功敏捷开发项目最重要的特征。你可以将所有敏捷项目管理实践放在一起，但如果忽略了测试驱动开发和持续集成的原则，那么你成功的机会就大大减小了。这些所谓的工程纪律不是可选项，没有不用的时候。

测试驱动开发

那么如何防止在代码中引入缺陷呢？Dijkstra说得很清楚，你在编写程序时应该“采用可证明性的观点”。这就是说，你在开始时要声明程序正确执行所必须保持的条件，然后再编写代码，让这些条件成真。[㊁]实际上，构建正确代码的一些正式方法已经存在了很长的时间，但它们一直难以使用，没有找到实际的、广泛使用的应用。那么，现在情况发生了什么变化？

xUnit框架

1994年，Kent Beck在Smalltalk Report[㊂]上发表了《Simple Smalltalk Testing》。在三页的篇幅中，他简单介绍了一个测试Smalltalk代码的框架，并建议如何使用该框架。1997年，Kent Beck和Erich Gamma编写了同一个测试框架的Java版本，称为JUnit。人们为其他语言写了许多xUnit框架，这些框架成为第一批广泛使用的、实用的工具，支持构建正确的程序。

测试怎么会被看成是支持构建正确程序的工具呢？测试难道不是事后进

㊀ 感谢Tomo Lennox的这段类比。

㊁ Freeman,《Software Systems Principles: A Survey》, 1975, 第489页。

㊂ Beck, “Simple Smalltalk Testing,” 1994, 第16~18页。

行的吗？将xUnit框架称为测试框架是有一点错误的，因为这些框架的推荐使用方法是为代码编写设计规模说明。xUnit“测试”代码正确执行所必须达到的状态，然后再写代码，让这些测试通过。虽然用xUnit框架编写规格说明可能不如契约式设计这样的正规方法严格，但由于它们简单灵活，所以得到了广泛采用。如果有一套深思熟虑的xUnit测试，聪明地发展并执行，就会使构建品质成为可以实现的目标。

xUnit“测试”，或用其他工具写下的测试，就是软件规格说明，它们从技术的角度说明了最小的代码单元是如何工作的，也说明了这些代码单元与其他代码单元如何交互，并构成一个组件。这些规格说明是由开发者编写的，并作为软件设计的一个工具。它们从来不是以大批量的方式，在代码编写之前（或之后）完成的，而是以一次编写一个测试的方式，在代码编写之前，由开发者编写完成的。

有一些工具可以测量代码集的xUnit单元测试覆盖率，但100%的代码覆盖率并不是要点，好的设计才是要点。你应该尝试在你的环境中去发现，怎样的代码覆盖率会产生好的设计。另一方面，测试集中100%的xUnit单元测试应该总是通过的。如果它们不能通过，策略一定是：停下，找到并修复问题，或者将导致问题的代码倒退出去。单元测试套件中的测试总是通过的，这意味着当有错误的时候，错误就是刚刚进入系统的代码所引起的。这消除了大量调试浪费的时间，Dijkstra的艰巨任务完成了。

单元测试不是到处都适用。例如，由代码生成器生成的代码（也许是第4代语言或领域专用语言）通常不是用单元测试可以修复的，你可能需要某种形式的验收测试。而且，对已有的代码集编写单元测试通常也不现实，除了你打算修改的代码。

对于新的代码，或者是对已有代码的改动，xUnit测试或其他等价的测试应该被认为是强制的。为什么？因为先编写测试，然后再编写代码通过测试，就意味着代码是可测试的代码。如果你想在将来修改这些代码，你就需要可测试的代码，从而可以自信地修改。当然，这意味着这些测试必须长时间维护，像其他代码一样。

验收测试

在许多组织机构中，在代码编写之前会编写一份相对比较详细的规格说明书，从客户的角度说明软件应该做什么。这份规格说明书应该由真正懂得系统需求的人来编写，由实现这些特征的开发团队成员提供帮助。在代码之前，不要过早编写规格说明书的细节，毫无疑问，这些细节会改变，相应的工作就会浪费。详细的规格说明书应是迭代式地编写，以一种即时（just-in-time）的方式完成。

软件规格说明书有两项传统用途：开发者编写代码来满足这份规格说明，测试人员创建不同类型的测试，并编写测试脚本来确保代码满足规格说明。但问题在于，这两个小组通常是独立工作的，更糟糕的是，测试人员会在代码完成之后再编写测试脚本。在这种情况下，开发者与测试人员对规格说明的解读会存在不可避免的分歧，所以代码和测试不能匹配。在代码写好之后，测试会执行，然后会发现这些差异。这不是一个错误 - 证明过程，而是一个缺陷-注入过程（参见图2-3）。

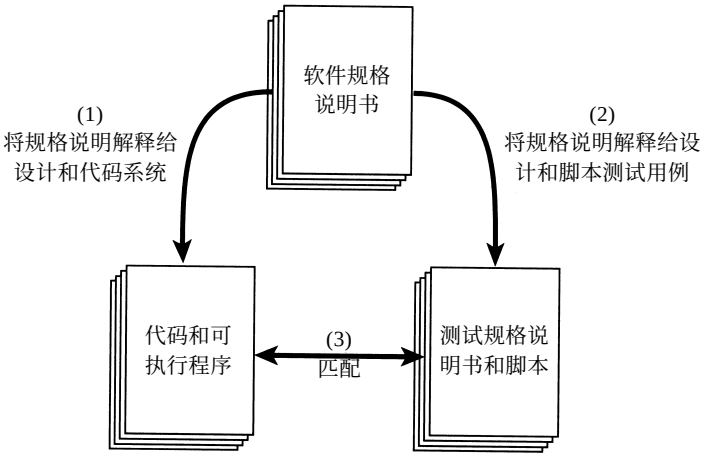

图2-3　缺陷 - 注入过程

要使这个过程变成缺陷-证明的过程，开发的次序就需要改变。测试应该先创建 —— 在编写代码之前。让测试由规格说明导出，或者让测试本身就是规格说明更好。不论哪种情况，代码都要等这部分代码的测试完成之后再编写，因为这样开发者才知道他们需要编码实现什么。这不是欺骗，这是错误-证明过程（参见图2-4）。

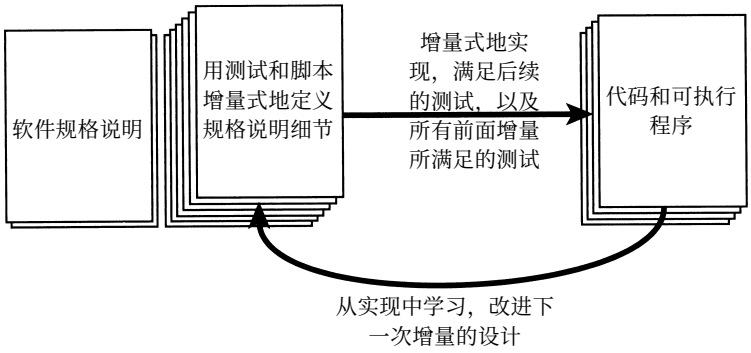

图2-4 错误 - 证明过程

对于指定软件产品或应用的细节来说，我们倾向的方法是编写验收测试。这些测试是用户场景（end-to-end）测试，规定了应用从业务的角度来看需要做什么，所以它们与单元测试有很大的不同。如果规格说明是以可运行的测试的形式写成的，那么缺陷就可以在进入到系统时检测出来，并进行修改。

当然，并非所有的规格说明都能写成自动化的测试，而且自动化测试的创建也需要认真思考，否则它们很快就会变成复杂的负担，像其他软件一样。但是，据我们所知，创建一套智能的测试套件来检测代码应该做什么，是产生满足规格说明的代码的最佳方式。

测试自动化

可执行的测试（实际上是规格说明），不论它们是验收测试、单元测试或其他测试，都是对软件开发过程进行错误-证明的工具。这些测试构成测试套件，然后在每次编写新代码时，代码就添加到系统中，同时执行测试套件。如果测试通过，就说明不仅是新代码能工作，过去能工作的所有代码仍然能工作。如果测试不能通过，那么刚刚添加的代码就有问题，或者至少暴露了一个问题，所以新代码回滚，问题马上得到关注和解决。

如果这听起来很容易，接下来的内容肯定就不容易了。决定什么要自动化、编写好的测试、让测试自动化、在正确的时间执行测试，以及维护这些测试，每一项工作都是挑战，就像编写好的产品代码一样。这要求我们有好的测试架构，明智地选择不同测试套件包含哪些测试，并不断更新这些测试，让它们一直能匹配当前的代码。

需要开发自动化脚本来支持测试早执行和常执行，也需要纪律保证在发

现缺陷时，就停下来进行修复。虽然做这些事情的方法都已存在，但它们是新方法，有较大坡度的学习曲线。如果你打算花钱进行敏捷开发的培训，购买一些工具（你应该这样），那么可以首先在这方面进行投资。⊖

展现层　在大多数情况下，自动化测试不应该通过用户界面来执行，因为用户界面通常执行得很慢，而且它不可避免地会经常改动，使得这种测试的维护变得几乎不可能。分离用户界面层和业务功能层是一个好主意，同样，分离展现层的测试和其他代码的测试也是一个好主意。利用一个很薄的展现层，展现层测试可以简单地验证屏幕动作将正确的结果发送给了下一层代码。这样，其他的测试就可以在用户界面下自动执行，而不需要通过用户界面。

失效测试

采用建构式方法来开发软件，确保了软件会完成人们希望它做的事情。但是，你不能确保软件不会做人们不希望它做的事情。实际上，你甚至不能列出究竟不希望软件做哪些事情——不论多么努力，软件总会找到另一种方法让你吃惊。确保软件健壮的经典方法，就是将它推到所有能想到要面对的失效面前，使其在这些失效面前都工作得很好。

探索式测试　要确保你的组织机构中有一些测试，它们的思维很特殊，喜欢破坏系统。当它们成功时，大家应该感到高兴，因为它们发现了弱点。因此应该设计一个测试，让正常的测试在将来能够检测到这种失效。而且，要能查看失效的原因，看看是否可以使用更好的设计方法，在将来避免这种失效，或者是否存在一个边界条件需要清楚地控制。在开发过程中，探索式测试应该早进行、常进行，因为它常常会揭示设计缺陷，而设计缺陷发现得越早，就越容易修复。

可用性测试　即使系统的行为与规格说明完全一致，但用户弄不清楚如何有效地使用它，系统也不会太有用。高额培训成本、低生产效率，以及高用户出错率都是糟糕的交互设计的标志。人机交互的能力应该通过可用性测试来验证。这种验证通常最初是在简单的纸面模型上完成的，然后可能是一些原型，最后是实际的系统——同样是越早越好。

⊖ 我们强烈推荐Crispin和Gregory的《Agile Testing: A Practical Guide for Testers and Agile Teams》, 2009, 该书非常详细地介绍了测试自动化和测试的一般概念。

压力测试 性能、负载、伸缩性、安全、弹性、稳定性、可靠性、兼容性、互操作性、可维护性，这些是压力测试的不同方面，应该根据你的具体情况来进行。通常你希望在压力测试中将系统推下悬崖，这样就知道悬崖在哪里，然后在正常运行时远离悬崖。同样，尽可能早、尽可能频繁地执行这些测试。

图2-5总结了我们在这一节中讨论的各种类型的应用测试。

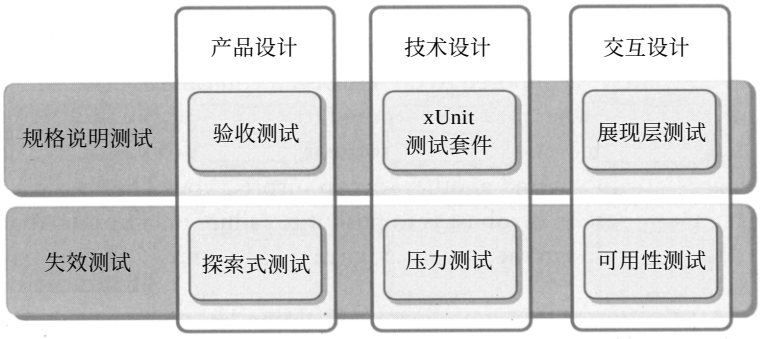

图2-5 应用测试

持续集成

代码不是存在于真空之中，如果是这样，软件开发就容易了。代码是存在于一个社会环境中，可能我们在软件开发中遇到的最大问题就是代码并非总是很好相处。因此，集成通常是软件最大的问题，而且如果它是问题，那么将集成放到开发的最后阶段又成为了缺陷 - 注入过程的一个例子。Harlan Mills在1970年就想明白了这个问题，而且他发现了解决方案。先开发一个系统骨架，然后每次添加一些小程序，确保它们能工作，再添加后面的程序。他将这种方法称为“ 自顶向下编程 ”，但背后的思想在传播的过程中丢失了，很少有人真正理解Mills的术语的含义。

今天，与Mills的“ 自顶向下编程 ”的术语等价的是“ 分步集成 ”。而且，有了测试套件框架的帮助，我们可以更进一步，持续地进行集成。我们可以使用Harlan Mills使用过的同样的依据来检验它的有效性：如果你有大爆炸式的集成问题（或者是小爆炸式的集成问题），你就没有很有效地进行持续集成。

我们是什么时候开始认为，应该等到开发的末尾才将系统的各部分组合在一起？这只是另一个例子，将所有隐藏的缺陷都保存起来，直到开发的末

尾才去寻找它们。这违反了本书中人尽皆知的良好测试实践。我们怎么会认为这样做有意义？为什么我们要这样做？

解释可能存在于串行式参考取景框的扭曲逻辑中，但主要是因为在每一小步集成后都进行测试是非常困难的。在过去，测试绝大多数都是手工执行的，因此很辛苦。而且，测试部分完成的代码很容易误报错误。同时，当我们将集成测试的水平提升到更大的系统上时，测试就变得更复杂，因为我们必须测试更大规模的程序。最后，确实是不可能发现所有会出错的地方，测试所有可能性的子集也会是一个很长的过程，因此人们不能够经常进行测试。

所以我们不奇怪，持续集成是不容易的。随着集成升级到更高的系统水平，这就变成了巨大的挑战。但请你考虑一下好处：通过一系列的集成测试，并在测试发现缺陷时就立即修复缺陷，许多公司将产品发布周期缩短了30%。另一方面，最好的公司留给最后验证的预算不会超过整个发布周期预算的10%，而且他们的发布周期越来越短。他们能做到这一点是因为缺陷已经在开发过程的早期被发现了，如果他们在最后的验证中发现了缺陷，就会感到吃惊，并努力找到方法，防止将来这些缺陷潜伏到最后验证中才被发现。

通常，在产品交付后，还需要额外的集成测试。这是正常的，只要集成过程非常顺利，不会发现缺陷就行。但是，如果客户在集成你的软件时发现问题，你就需要再做一些工作，使你的产品更容易消费。在价值链的任何地方发生的集成问题都是你的挑战，你需要找到并修复问题的真正根源，让这样的问题不会再发生。

为复杂系统实现持续集成将对架构选择提出一些约束。人们会偏向于选择容易测试的、低依赖性的架构。

多频繁算“持续”

在过去的10年里，人们开发了许多工具和技术，让频繁集成可以在单元级、应用级和系统级上进行。但是，随时集成所有代码并非总是可行的。而且，有了低依赖性的架构，可能也不需要对真正独立的代码部分进行集成测试。以怎样的频率进行集成和测试，这取决于需要怎样的方式来尽快发现缺陷，包括那些在集成时会影响系统其他部分的缺陷。所以判断集成频率够不够的依据就在于你随时快速集成的能力，而且在集成时没有发现缺陷。话虽

这样说，但还是有一些常见的集成频率，你可以考虑采用。

每分钟集成　敏捷开发者会非常频繁地向配置管理系统签入他们的代码，这很常见。在签入代码之前，他们会先签出最新的版本，在本地执行测试套件。然后他们签入代码，这会触发自动构建和测试循环，预期会通过。在极少的情况下，构建和测试会失败，签入代码的开发者会马上回滚代码，在本地修复问题。有了这条纪律，缺陷的原因就很容易定位，因为在测试套件执行之前，只有少量的代码改动过，所以如果发生问题，很清楚就是最新的代码导致的。这几乎和调试一样容易，但需要绝对的纪律，一直保持测试套件100%通过。

这样频繁使用的测试套件必须能够快速执行，这一点很重要，否则其他开发者就会延迟签入（这太花时间了），这意味着还没发现任何缺陷就花费了许多时间 —— 这造成了恶性循环。因此，明智地决定这个测试套件的结构和内容是很重要的，因为快速执行的测试套件不可能覆盖整个应用。

每天集成　每天结束时，就是执行那些太花时间且不能持续执行的测试的时候。这可能意味着执行验收测试套件。验收测试工具与xUnit测试工具有一些不同。首先，工具必须能够区分应该通过的测试和预期不会通过的测试，因为代码还没有完成。其次，验收测试常常需要在数据库中建立一种场景，或者在真正的数据库上执行。最后，这些测试会花费更长时间（通常它们在晚上执行），当开发者第二天回来时，第一件事就是停下来修复晚上发现的错误。如果在这个晚上的所有东西都是正常的，那么只有昨天的代码可能有问题。失败的测试可能存在于一些没想到的代码之中，但失败最有可能是因为昨天代码的一些问题所引起的。

验收测试通常是在一个“干净”的数据库上执行的。在实践中，生产数据库可能有些乱，所以在干净数据库上执行验收测试不足以发现所有的问题。但是，每天执行验收测试套件并修复所有的错误（只要可行）将带来巨大的好处。然而系统迟早都应该在更合理的生产数据库的副本上进行测试。

每次迭代集成　当软件准备发布时，通常会有一些阶段性的活动。例如，系统测试（包括用户场景测试，在生产数据库副本上进行测试，在用户环境中执行）会进行，从而确保代码已经准备好发布。一般来说，将所有这些阶段性的活动留到系统准备发布时进行是一个错误。更好的做法是尽早执行系统测试，在引入问题时就发现它，而不是等到以后再发现。准备系统测试可

能非常复杂，系统测试的执行时间也可能很长，不完整的特征还可能会误报错误，浪费许多时间。但尝试改变这种现状，对部分完成的应用进行有效的系统测试还是有价值的。

在每次迭代结束时[⊖]（或者更早的时候），开发团队都应该得到一个可以进行系统测试的应用。在用户的眼里，这个应用可能还没有完成，但特征应该已经足够完整，可以进入系统测试并成功通过测试。当然，只有大部分系统测试自动化了，才可能做到这一点，尤其是当测试时间很长时。因此，测试自动化是很重要的，包括测试准备、测试执行和缺陷报告。最后，必须有一种清楚的机制，判断哪些特征能工作，哪些特征还没有完成。这种机制是在应用完成之前进行的系统测试的一部分。

系统级的压力测试也很重要。毫无疑问需要执行一些压力测试。在每次迭代末尾执行这些压力测试是个好主意，这样如果有什么不对，你就可以尽早发现，对于压力测试来说，这一点特别重要，因为在开发过程的晚期再来处理压力测试的失败通常会代价高昂。

用户验收测试（UAT）也可以作为早执行、常执行的测试，只是与用户合作可能有一些问题。然而，如果更早的UAT意味着更快的交付和更少的缺陷，好处可能非常大。即使用户不允许你在他们的环境中执行UAT，创建一个模拟的用户环境也能取得明显效果，帮助你寻找缺陷。否则，这些缺陷会在开发过程晚期才暴露出来。

部署之后

虽然你尽了最大的努力，缺陷偶尔还是会溜进生产系统中。如果你进行了我们前面提到的所有测试，这种情况出现的几率就应该比以前小很多。但如果缺陷溜进了生产系统，开发团队可以将它看成是一个好机会，发现设计方式和开发过程中有什么问题，导致了缺陷溜进生产系统。我们建议开发者，至少是开发团队的代表，跟着他们的系统进入生产环境，与负责支持系统的人合作，寻找问题的根源。这有助于开发者明白系统的使用方式，并帮助他们改进设计，这样同类的问题将来就不会再溜进生产系统。[⊜]这是Amazon.com的

⊖ 一次迭代可能是2周或4周。在使用看板调度时，像第3章中介绍的那样，系统测试至少会在发布之前进行。即使发布版之间的间隔时间不短，系统测试仍然可以更频繁地进行。

⊜ 参见Zeller的《Why Programs Fail》, 2006, Zeller, “Predicting Bugs from History,” 2008, 以及Nygard的《Release It! Design and Deploy Production-Ready Software》, 2007。

方法中包含的智慧（参见小故事《如果你们需要达成一致，就失败了》），他们的每个服务团队负责运营和开发。

代码清晰性

构建品质的最后一个方面就是简单性。简而言之，错误隐藏在复杂性之中，而在简单、构造良好的代码中，错误会暴露出来。交付、维护和扩展软件所需的时间和工作量直接与代码的清晰性有关。缺少清晰性是技术债，最终会连本带利要求偿还，这种技术债会使你无法开发新特征。还不起技术债常常会导致破产：系统必须抛弃，因为它不再值得维护。从一开始就不要欠技术债，情况会好得多。要保持简单、清晰、干净。

在《Clean Code》一书中，Robert Martin采访了几位世界级的编程专家，询问他们如何识别干净的代码。⊖

Bjarne Stroustrup，C++的发明者：

> 我喜欢代码优雅而有效。逻辑应该很简单，使缺陷难以隐藏。依赖关系应该最少，从而便于维护。根据表达清晰的策略，有完备的错误处理。性能接近最优，这样就不会诱使人们进行无原则的优化，把代码弄得一团糟。干净的代码能做好一件事情。

Grady Booch,《Object-Oriented Analysis and Design with Applications》的作者：

> 干净的代码是简单而直接的，就像写得很好的诗。干净的代码从不隐藏设计者的意图，而是充满了鲜活的抽象和简单的控制结构。

“Big”Dave Thomas，OTI的创始人和Eclipse战略的教父：

> 干净的代码可以让除作者以外的人阅读和增强。它有单元测试和验收测试，包含有意义的名称。它只提供一种方式做一件事，而不是许多方式。它有最少的依赖关系，而且依赖关系是明确定义的。它提供了清晰的、最小化的API。代码应该是有文化的，因为根据采用的语言，并非所有需要的信息都能够在代码中清楚地表现出来。

⊖ 这些内容引自Martin的《Clean Code: A Handbook of Agile Software Craftsmanship》，2009，使用得到了许可。如果你想看关于创建简单可读代码的最佳技术的深入讨论，我们强烈建议你读这本书。

Michael Feathers，《Working Effectively with Legacy Code》的作者：

> 我可以列出所有在干净代码中注意到的品质，但有一种品质是最重要的。干净代码总是让人一看就知道，它是某人用心写的，而且你找不到明显的方法使它变得更好。作者考虑了所有方面，如果你试图改进，就会回到原来的代码，坐在那里欣赏某人留给你的代码——特别用心关注这项技艺的人写的代码。

Ward Cunningham，Wiki和Fit的发明者，极限编程的发明者之一。设计模式背后的力量。Smalltalk和OO思想的领导者。所有关心代码的人的教父：

> 如果每个程序读起来都在你的意料之中，你就知道面对的是干净的代码。你可以称之为漂亮的代码，因为这种代码就像是专为这个问题而设计的语言。

信息隐藏秘史

在1969年的一个安息日，David Parnas惊奇地发现学术界推崇的清晰、简单的设计在工业界被认为没有用。他讲了数据库专家Johan和当时在编写一个编译器的Jan的故事。在一次午餐的时候，Parnas专注地看着Johan在一张餐巾纸上画出“打开文件”命令的细节，以便让Jan能够编写编译器中打开文件的代码。Dave不知道究竟出了什么问题，但直觉告诉他，这个代码集令人痛苦。

这张餐巾纸被编译器团队中的其他成员复制并使用，打开文件代码的不同实例进入到编译器的许多地方。几个月后，Johan忘记了那张餐巾纸，改动了文件的结构。回到编译器团队，没有人意识到打开文件的细节已经更改，所以编译器不能工作了。人们花了很长的时间来弄清楚编译器哪里出了错，并花了更多的时间来修复。

Parnas意识到，如果文件处理模块的结构向外界隐藏了它的内部工作细节，这种问题就永远不会发生。这就是为什么他将这种方法称为“信息隐藏”。最后他明白了如何解释好的设计在商业上也很重要：在发生改变时，清晰、简单的设计将使系统维护容易得多。

Mary Poppendieck，利用了Parnas提供的信息，
《The Secret History of Information Hiding》，2002

重构

没人会想到一本书的第一稿就很完美，你也不应该期望代码的第一稿会很完美。更重要的是，实际上所有有用的代码集最终都需要变更（这是好事，不是坏事）。代码必须保持简单和清晰，即使是随着时间推移，开发者对问题了解得更清楚，或问题本身发生了变化。因此，当我们通过更深入的了解而有所发现，或引入最新的变更时，代码必须持续重构，处理任何形式的重复或意义不明之处。辅助重构的工具越来越多，但工具通常不是问题，重要的是开发者必须认为重构是他们工作的标准组成部分。和持续测试一样，持续重构不是可选的，而是必需的。选择不重构就是选择不还技术债，最终会导致越来越多的失效要求。

取景框7：演进式开发

Peter Denning从事软件很长时间了。他对操作系统原理的理解做出了贡献，几乎担任过ACM所有的领导职位。[⊖]在2008年，他和两个同事写了一篇命题论文，讲的是为什么非常大的系统开发项目（作者们工作的Navy Postgraduate School of Computer Science所承担的项目）常常不能成功。他们说：[⊜]

> 如果开发时间比环境变化的时间短，那么交付的系统可能会满足客户的要求。但是，如果开发时间比环境变化的时间长，那么交付的系统就过时了，在它完成之前可能就是无用的。在政府和大型组织机构中，对大型系统官僚主义式的采购过程通常会花上10年以上的时间，而在这期间环境通常早已发生了巨大变化，因为18个月就能发生巨大变化（摩尔定律）……
>
> 传统的采购过程试图通过小心的事先计划、预期和分析来避免风险和控制成本。对于复杂的系统来说，这个过程通常要花10年以上的时间。是否有别的方法，使花的时间更少，而且仍然能适用？是的。演进式

⊖ ACM全称为Association for Computing Machinery。www.acm.org。

⊜ Denning、Gunderson和Hayes-Roth, “Evolutionary System Development,” 2008。摘录得到了许可。本节中的许多思想来自这封信。

> 系统开发在动态的社会网络中产生大型的系统。因特网、World Wide Web、Linux都是突出的例子。这些成功没有核心的、事先计划的过程，只有系统架构的一般概念，这为协作创新提供了一个取景框……
>
> 演进式开发是一个成熟的思想……它让我们能够成功地创建大型关键系统。演进式的方法以增量的方式交付价值。它们不断优化较早的成功系统，交付更多的价值。不断增加的价值维持着系统的成功，形成一系列短期的升级换代。

Denning和他的同事谈到了两种类型的演进。第一种是系统通过一系列的发行版实现演进。他们说，这样做是可行的，条件是发行版要足够接近。当发行版的时间间隔超过了环境变化的时间，这个策略就不行了。第二种演进策略是多个系统相互竞争时发生的，只有最合适的能够生存下来。[㊀]为了测试多种可选方式，World Wide Consortium for the Grid（W2COG）启动了两个项目，开发一个安全的面向服务的架构系统。一个项目采用标准的采购和开发过程，另一个项目采用“有限技术试验（limited technology experiment，LTE）”的方式，这是一种演进式开发的过程。

> 两个项目都收到了政府提供的同样的软件，作为开始的基线。18个月以后，LET的过程交付了一个开放架构原型，实现了80%的政府需求，成本是10万美元，所有嵌入的软件都是当前最新的，并且计划在6个月内转变成为完整的商业上架销售（COST）软件。相比之下，在18个月之后，标准过程只交付了一份概念文档，没有提供一个功能架构，没有能工作的原型和部署计划，没有时间表，花费了150万美元。[㊁]

演进式开发并不是新方法。正如我们所看到的，它是Tom Gilb在1981年提出来的，是开发PC软件、因特网和开源软件的主要取景框。对于减少风险来说，演进式开发是一种很好的方式，因为它直接面对风险，而不是避免风险。演进式开发试图发现最能够处理风险的系统，当出现风险时，它们采用当前的系统来处理风险，或者实地调查多个系统，并选择最好的方案。

演进式开发方法的核心是一系列短期的发现循环，每个循环交付一个或

㊀ 这被称为set-based开发。

㊁ Denning、Gunderson和Hayes-Roth, “Evolutionary System Development,” 2008。摘录得到了许可。

多个“试验品”。这些试验品构造的方式是让它能够快速失败，这样就能够快速发现开发团队的弱点，从而使开发团队能够基于这种反馈来改进系统。每个发现循环包含三个阶段：族群研究、协作建模和快速试验（见图2-6）。

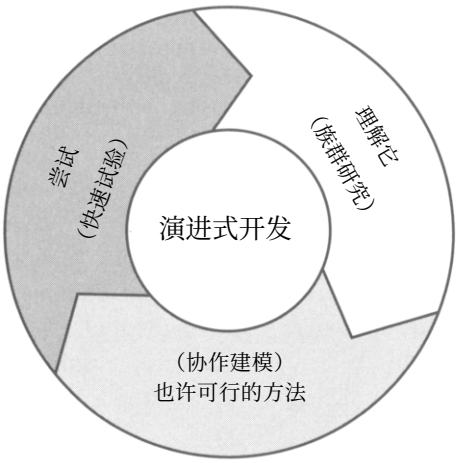

图2-6 发现循环

族群研究

Tom Kelley是IDEO的总经理，他说，族群研究“通过观察人的行为，对人们在物理上和情感上如何与产品、服务和空间实现互动形成深入的理解，为组织机构带来了新的知识和深刻见解，”[⊖]。他坚持认为现场研究是设计中最重要的步骤。开发团队中的各种成员（具有不同背景和职位的人）花时间来观察人们如何与问题较量，而他们的新系统的目标是解决这些问题。他们以一种新手的心态来观察，看哪些事情让人生气，哪些事情妨碍人们完成他们的工作。他们创建一份缺陷清单，指出当前体验中不对的地方。

族群研究不是收集需求或聚集关注人群。它不是让一个产品拥有者或产品经理来告诉开发团队系统应该做什么，而是让具有不同背景的几个开发团队成员走到需要使用系统的现场，与需要使用系统的人进行交谈。它是获得这些人现在如何工作的第一手观察资料，了解他们有什么烦恼（甚至那些小的、没说出来的烦恼），以及他们的解决办法。族群研究专注于什么有效和什么无效，它是不断地问为什么。为什么人们采取这样的行为方式？为什么它

⊖ Kelley和Littman,《The Ten Faces of Innovation》, 2005。这一节中的许多思想来自于该书的第1章。

们对事情工作的方式感到烦恼？为什么产品和服务按现在的方式设计？为什么它们难以（或容易）使用？观察局外人是有回报的：他们不会遵守规则，不会像通常的方式那样使用产品，他们要么喜欢一个产品，要么恨一个产品，而且不会拒绝说出理由。

跟我回家

公平地说，软件开发行业的族群研究是由Intuit率先尝试的，它是Quicken、QuickBook和TurboTax的供应商，这些产品是美国流行的家庭财务管理软件。在1989年，Intuit开始了它的“跟我回家”计划，目的是帮助保持不断增长的公司与客户之间的联系。《Inside Intuit》一书介绍了这个计划的工作方式：⊖

市场部员工和工程部员工跟随自愿第一次购买其产品的客户从商店回家，观察他们安装和使用Quicker。Intuit提供了反馈请求和调查问题。观察客户在他们家里的情况，这种体验让工程师和市场人员理解了一句格言：“我们不是我们的客户”，也理解了关键的必然推论：“我们必须让Quicken更易于使用。”

“我做过一次‘跟我回家’”，一位员工在我们的一次研讨会上说。“我曾在Intuit工作”。这有点让人吃惊，因为我们离Intuit的总部很远。

“情况如何？”我问道。

“我当时是一名开发者。我走到客户家里观察她使用我们的软件。几个人一起去的，背景都不同，所以我们看到了不同的东西，问了不同的问题。结果让人吃惊，”他说，“我们学到了很多从来没有想到过的东西。这确实改变了我对软件的看法。”

Marry Poppendieck

软件密集型系统失败的最大原因不是技术上的失败，而是构建了错误的产品。我们知道这一点，一次次的研究证明了这一点。但是我们仍然继续在开发团队和客户之间设置中介。我们可能将这些中介系统称为分析师或产品所有者，但通常他们代表的只是信息传递。关于要解决问题的二手信息阻碍

⊖ Taylor和Schroeder,《Inside Intuit》, 2003。

了人们对系统的行为做出详细的决定。产品经理和产品所有者的关键角色是作为客户和开发团队之间的桥梁，这样他们就能够彼此交谈。

协作建模

在族群研究中获得的信息应该由善于思考的团队进行消化，然后他们决定该怎么做。这是发挥想象力、进行头脑风暴和讲故事的时候。团队需要形成一个模型，解释改进用户体验的下一步应该怎么做。他们可能编写一些用例，然后将用例分解成带有测试的故事。它们可能更新产品的路线图或创建一个思维导图。

快速原型在建模时是有用的。团队成员可能会勾画一张草图，或创建一个纸质原型，每个人都可以看到它、移动它、修改它、改进它。开发者可能尝试开发一些长钉测试来研究一些有风险的技术问题。某些团队成员还可能制作新业务过程的原型，或通过模拟器执行数据库负载设想，看看系统的反应如何。

在《Elegant Solution》[⊖]一书中，Matthew May提到了伟大发明家的12项实践，其中有一项就是“图示思考”。这在软件开发中是非常有用的，因为对于不读原代码的人来说，软件是不可视的。通过用图像、示意图和图表等手段来展示思考模型，开发团队成员可能更容易彼此交流、与客户交流，或者与对他们的工作感兴趣的人交流。例如，使用纸质原型，而不是计算机生成的原型来对用户界面建模。唯一要注意的是不要在画图上投入太多精力。如果模型太完美，或需要花太多精力来制作，它可能会难以修改，而模型的目的是鼓励讨论和修改。

快速试验

试验比原型建模要更进一步，在这个阶段，你实际上已经实现了真实系统的一些部分。换言之，试验是在实现时意识到实现的东西可能会改变，或者其他解决方案可能被事实证明工作得更好。例如，TCP/IP曾被认为是一项试验，虽然它的结果非常好，以至于我们在30年以后仍在使用它。即使是TCP/IP，也在这些年里更新了好几个版本。

在软件开发中，有两种一般的试验方法。较常见的一种方法是迭代式开

⊖ May,《The Elegant Solution: Toyota’s Formula for Mastering Innovation》, 2007。

发，先实现风险最高、优先级最高的软件。在使用迭代式开发时，重要的是将早期的迭代看成是试验，让这些代码容易随着系统增长而修改，从而包含越来越多的特征。软件设计应该关注于让变更在任何情况下都容易进行，所以低依赖性的架构和测试套件应该是迭代过程的一部分。

但是，重要的是不要在迭代开发过程的早期做出一些关键的、改动代价高昂的设计决定，难以改变的决定应该以基于集合（set-base）的方法来处理。基于集合的设计常在硬件开发中使用，因为改动通常比软件开发中的改动更困难。在基于集合的设计中，决定通常放到最后时刻才做出（当决定必须做出或者采取某个缺省决定的时间点），然后同时开发几个可选方案。通常这些开发的可选方案覆盖了尽可能多的设计空间，以便在几种截然不同的方法之间进行试验。在决定的时刻到来时，人们根据系统的观点来做出选择。

迭代式开发和基于集合的开发都是低风险的方法，使开发过程可预测，虽然它们听起来可能有些违反直觉。在迭代式开发时，你先开发高风险和高价值的特征，在这个过程中了解开发团队的技能和产能，同时也得到客户的反馈。因此就更容易预测系统何时会完成，或者在任意时刻停止开发，但仍有一个非常有用的系统。在基于集合开发时，决定时刻是有计划的，当决定时刻到来时，总有一个能工作的可选方案，同样，这也减少了风险，保证了进度。我们建议你综合这两种方法来进行演进式开发。

发现循环

试验完成之后，就重复发现循环：在现场进行试验，发现其工作的好坏情况，然后再次利用族群研究、协作建模和实验。我们认识到完全部署每次实验并非总是可行，也不一定需要，但要以尽可能短的间隔进行部署。我们也认识到族群研究并非总是可行，特别是在合同开发的情况下。我们理解有时候需要中介，而不是让开发团队直接与客户联系。如果你的环境不允许每一个阶段都按照我们所描述的方式进行，那么就要尽可能遵循这些原则：

1）在跨职能团队中工作。

2）确保团队深刻理解了客户的状况。

3）通过团队协作得到解决方案。

4）以某种方式对解决方案建模（文字、示意图、草图、原型，或任何管

用的东西），产生一个公认的方向。

5）快速在尽可能真实的环境中尝试解决方案。

6）将解决方案视为试验，并预期会进行修改。

7）以小步骤的方式开展工作，最好是采用有规则的节拍。

8）反复进行。

演进式开发的底线是：你要开发的系统不能在规模或时间上超出环境变化允许的时间。如果你正使用长时间的计划过程来增加可预测性，那么可以确信这一点：如果系统开发的时间超过环境变化的时间，系统就会失败。

取景框8：精湛的专业知识

程序员是作者是还是译者？

代码是为机器编写还是为人编写？

某些人将编程工作看成是翻译：拿来一份基于文本的规格说明书，将它翻译成机器能懂的语言。但这是正确的思维模型吗？两个独立工作的程序员会对同一个问题创造出截然不同的解决方案，通常在解读上有很大的不同。类似地，两个作者将对同一主题给出极为不同的阐释。但是，两个译者会对同一部作品给出类似的译文。如果你认为编程类似于将规格说明书翻译成软件，那么你可能不是一个程序员。很少有“写代码”的人认为自己是译者，而会认为自己是代码的作者。

有些人，特别是那些从未写过代码的人，认为程序员写代码是给机器读的。没有什么比这更离谱的了。程序员写代码是给人读的。程序员首先是为他们自己写代码，但更重要的是，代码是为必须与这段代码打交道的人和将来要修改代码的人而写的。好的程序员在写代码时会努力澄清他们的意图。他们希望读者，即那些读他们代码的人，知道他们想让代码干什么。

当你阅读母语写成的文章时，你可以很快说出它在讲什么，但当你看一种不懂的语言写成的文章时，你就不知道它在讲什么。程序员以一种奇怪的语言来写作，他们为理解这种语言的人而写，不能阅读这种语言的人不是读者。有时候我们看到人们要求为不能读代码的人提供文档，但我们知道真想修改代码的人必须去读代码。我们相信在大多数情况下，代码能清楚地说明

自己的意图远比文档翻译它的意思重要。

写作很像编程

现在我写书，但我曾做了很多年的程序员，并写了许多代码。我发现写书很像编程，我使用相同的思维过程，甚至相同的方法来写作。

要编写软件，我必须进入人们的头脑，他们会使用这个系统。我必须想象他们怎样使用它，需要清楚地知道系统中其他人的工作期望我的代码做些什么。我编写小段的、有逻辑的代码，当一段代码写成后，我就在一个模拟器中检验它。在代码写成后，我会花一些时间来修改它，让它工作得很好，将它整理得清晰可读。我知道如果没有可读的代码，过两天我自己也弄不清楚代码的意图是什么，更不必说将来那些要修改它的人了。

要写一本书，我也需要进入人们的头脑，他们会读这本书。我需要想象人们在阅读时会对什么内容感兴趣，需要清楚地知道其他人对我在写的这些主题都说过一些什么。我知道目前的第一稿只是一个开始，文字将经过大量的复查和修改，然后才会成为一本书。我非常感激复查者提供的无价的深刻见解。我花许多时间对一本书重构，确保我的意图是清楚的，也会花许多时间来消除重复，确保这本书读起来流畅。

Mary Poppendieck

专业很重要

“构建软件是一个创造性的过程，”⊖Fred Brooks写道。“了不起的设计来自于了不起的设计者……一次次的研究表明，最好的设计师设计的结构更快、更小、更清楚，而且工作量也很少。好的方法和一般的方法具有数量级的差别”。

如果编写软件是一个翻译过程，你大可请几个杰出的设计师，让他们设计一个杰出的系统，将设计交给一个译者团队，并期望得到极好的结果。但这样做的结果通常会很凄惨。现在我们应该知道，在软件开发中，我们不能够分离设计和实现。为什么？因为编写代码是一个创造性的过程。写代码的

⊖ Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering,” 1986。

人是作者，不是译者，他们会不断做出设计决定。“我如何处理一个缓冲区以防止溢出？这10件东西最好的分组方式是什么？我如何发现微妙的重复并消除它？啊！这是几年前导致客户系统崩溃的原因，最好不要用在这里。”诸如此类。大约每10分钟，都会做出一个设计决定。

专业知识在开发组织中的每一个层面上都需要。当然，你们应该有一些了不起的设计师，但他们必须与实现系统的人分享他们的观点和专业知识。否则，他们的杰出设计就永远不会实现，他们的专业知识也不能融入到软件之中。如果你希望系统“更快、更小、更清楚，而且工作量也很少”，希望系统让用户喜欢，希望系统易于维护和扩展，那么你必须在实际做出设计决定时具备设计专业知识。技术杰出来自于有意识、有计划的努力，在组织机构的每一个层面上培养并保持专业知识。

培养专业知识[⊖]

美国航空公司1549航班在2009年1月15日从LaGuardia机场起飞后两个引擎都丧失了动力。如果你是飞机上的一名乘客，那就要庆幸遇上了最好的飞行员。机长Chesley B. Sullenberger III，58岁，在14岁时就取得了飞行执照。他为空军开了7年的F-4幻影战机，并成为飞行队长和培训教官。他从1980年开始驾驶商业飞机，到2009年，已经积累了19000小时的飞行经验。他曾与联邦航空官员一起调查过飞机失事，设计改进的方法来处理紧急情况。他甚至飞过滑翔机。在那个寒冷的一月的一天，他将没有动力的空中客车滑翔到Hudson河上，完美着陆。

水面着陆的最大风险在于，如果水面与飞机的接触不平稳，飞机就会翻筋斗或折断，所以着陆必须完美。在这次着陆过程中，一个引擎（在机翼下）脱离了，但两个机翼都还完好。飞机漂浮在河面上，150名乘客聚集在机翼上或挤在救生筏里。附近的船只冲过来救援，几分钟之内，所有乘客都安全了。当大家祝贺飞行员的神奇着陆时，他回答道：“我们接受训练就是为了这样的时刻。”

⊖ 这一节的信息来自Ericsson等的《The Cambridge Handbook of Expertise and Expert Performance》, 2006，Colvin的《Talent is Overated》, 2008，以及Gladwell的《Outliers: The Story of Success》, 2008。

这个故事里有一个有趣的数字：19000。学习专业知识的人知道，不论我们谈论的是飞行员、音乐家、律师、高尔夫选手或软件设计师，证据都很明显：至少需要10000小时认真而专注的实践，才能成为专家。这意味着大约需要10年时间来达到顶峰。世界级的大师在达到顶峰之后不会停止实践，他们继续认真的实践，不断投入精力，让自己比领域中的其他人做得更好。1549航班上的乘客确实幸运，他们幸运地遇上了驾驶舱里的Sullenberger机长，副驾驶是Jeffery Skiles，他是一位49岁的飞行员，有26年的商业飞行经验，从15岁起就是一名飞行员。另外还有3名机组成员，他们加起来也有超过92年的经验。正是这些专家的技能让飞机安全地降落，人员安全地疏散。

你不希望凭运气来使你的软件系统可信。技术杰出不是运气问题，它是技能，而技能需要花时间来培养。如果你正在开发系统，如果你希望有杰出的系统，那就要注意培养一些有技能的开发骨干。不一定每个人都必须是专家，但应该有一些专家，也有一些人正在积累专业知识，还有一些新手。你需要确保他们都在不断增加经验，尤其是像软件这样快速变化的技术领域。

认真的实践

专业表现的研究者们达成了广泛的共识，天才不能解释为一道门槛，你必须至少花一些时间来开始某项运动或职业。在那之后，胜出的人是工作最努力的人。他们必须经过10年的认真实践，才能成为专家。认真的实践不是堆积时间，而是认真工作以改进表现。认真的实践工作是这样的：

1）确定需要改进的具体技能。

2）设计（或从老师那里学习）一项练习来改进这项技能。

3）反复实践。

4）每次努力后立即得到反馈，并根据反馈进行调整。

5）专注于突破极限，预期会反复失败。

6）实践通常很密集，也许每天3小时。

认真的实践并不是意味着你在做擅长的事情，而是意味着挑战自己，做自己还不擅长的事情，所以它不一定有趣。日复一日地持续进行这类练习要求参与者有足够的热情，也需要来自重要的人的鼓励（例如同事或同伴），还需要一个老师，他能指出需要改进的技能，并知道最新的实践技术（至少在

开始时是这样的)。

不幸的是，我们的组织机构不是为培养专家而建立的。首先，工作指派很少提供挑战，让开发者能够突破能力极限。相反，我们倾向于安排人们做他们擅长的事情。其次，我们很少提供深入了解技术的经理，让他们理解管理的工作，提供定期的、专注的反馈，以便让开发者改进。我们也没有提供其他的途径来产生这样的反馈，例如，提供技术现场指导或有经验的团队成员，以及深入思考的复查过程。最后，我们倾向于让开发者与他们的客户保持距离，而不是允许他们立即得到反馈，并且深深地专注于工作的整体成功。

十年法则

你不会希望让没有经验的开发者在没有指导的情况下开发关键的软件，就像你不希望没有经验的机长来驾驶飞机一样。你可以雇佣有经验的开发者，但如果他们的经验不在你的领域内，可能就不充分了。学位和证书证明了此人达到了成为专家的门槛，但培养真正的专业技能需要花上多年的时间。

人们发现十年法则几乎在所有复杂领域的研究中都成立，在任何领域，都需要大量的认真实践才能造就高水平的表演者。如果只用一半时间，你会具备竞争力，但不会有辉煌的表演。专业知识具有领域特殊性，转移到类似领域的人可能具有一定基础，但不会马上成为顶级的表演者。花十年做同样的事情并不一定代表具有专业知识，如果经验中没有包含认真的努力，没有改进弱点和尝试新的东西，可能对培养专业知识没有效果。

所以，一个刚从大学毕业的学生不太可能成为了不起的设计者或了不起的程序员，除非他们在上大学之前就写过多年的程序。为了成为杰出的人才，刚从学校毕业的雇员需要时间、现场指导和持续专注的实践。光写代码是不够的。通过编写以有效方式解决问题的代码，编写风格优雅和清晰的代码，编写比上一次更好的代码，他们才会学到专业知识。为了做到这一点，开发新手需要老师或现场指导者，需要有挑战的工作安排，需要迅速的反馈信息，最重要的是要有一项可以全身心投入的工作。你需要一些专家来指导这些正在培养编程技术的人。你必须有一种办法，让这些正在实践的人知道他们什么时候干得漂亮，哪里需要改进。

留住人才

“好的，”你会说。“难道你从没听说过人员流动吗？我怎么将一个人留住10年？”人员流动有两种原因：一种是系统性的人员流动，你也许不能控制。另一种人员流动是可以减少的，如果你的组织机构是比现在更有吸引力的工作场所。让我们从系统性的人员流动开始。军方组织有着非常高的人员流动率，大多数新兵参军几年后就转向了其他职业。某些国家（我想起了印度）的软件行业新手中也存在着类似的高人员流动率。如果你的情况是这样的，请研究一下顶级军方组织如何处理人员流动率高的问题。首先，他们有能力快速让新兵进入状态，通过专门的训练和仔细设计的工作分派。其次，他们保持了一个核心领导团队，这些人在组织机构中服务许多年，他们的专业培养计划集中在这些中长期的人员上。

除了系统性的人员流动之外，我们发现留住人才通常是一个自实现的预言：如果你不认为留住人才重要而且可能，那么它就不可能实现。但是，如果你努力提供一个场所，让人们接受挑战，帮助他们发挥所有的潜能，那么这个场所就是有激情的人愿意留下来工作的地方。

为什么那些开发开源项目的人免费贡献出这么多工作？部分原因是因为开源工作刚好提供了这种认真的实践，而大多数公司却没有提供。开发者取得一些有挑战的任务，提交的代码由专家复查，然后专家迅速地给出反馈意见。如果代码成功提交，开源社区会提供更多（有时是更严厉的）反馈意见，这正是努力想成为专家的人真正需要的。开源开发者很在意他们的工作，专业作家、飞行员和小号演奏家也是如此。我们相信，要留住正在积累专业知识的人才，最好的方式就是让他们有机会改进他们的技能，让他们对工作产生热情。我们还将在第5章进一步讨论这个主题。

标准

标准是目前组织机构中好的实现的基线，每个人都要遵守。它们有点像音乐中的音阶学习。专家的一项职责就是确保这些标准可用。那些通过艰苦努力学到经验，知道在环境中最好的做事方式的人，应该确保其他人不需要重新发明轮子。但是标准永远都不会完美，需要不断地挑战和改进。所以，当某人掌握了标准实践之后，可以进行一项好的练习，即在一名指导者的帮

助下挑战标准中需要改进的一些方面，找到做事情更好的方式。我们将在第4章中深入讨论这个主题。

代码复查

代码复查是将编程转化成认真实践的一种好方式。让专家（相当于代码提交者）或同事（例如通过结对编程）来复查代码，可以对代码风格、清晰性、健壮性和在你的环境中的重要问题提供迅速的反馈。这种反馈通常伴随着改进的思想。所有的人，如果将编程视为一项需要持续改进的技能，就会感激这种反馈。因为这些编写代码的人如果知道同事会阅读和修改代码，就会注意代码的风格和清晰性。新手不仅可以看到代码的例子，还可以和有经验的人共同编写这些代码。

在你的组织机构中培养专业知识，就是要提供一个鼓励专业知识的环境，帮助人们充分发挥他们的潜能：

1）为智慧的专家提供时间和场所，让他们作为指导者，指导正在学习的人。

2）通过有趣的挑战和有效的反馈，为认真的实践提供许多机会。

3）让人们有可能对工作产生激情。

肖像：竞争力领导者

2006年，Jeff Immelt说他为GE寻找的领导者要具备五个特点：外部关注、清晰思维、想象力、包容力和领域专业知识。[⊖]“为什么要领域专业知识？”有人问他。“GE最成功的部分就是那些领导者长期工作的部门，而那些我们不断换人的地方……就是我们失败的部门。”Immelt解释说，具有领域专业知识的领导者能做出正确的重要决定，因为他们靠的是深入的业务知识。我们认为同样的原则也适用于竞争力领导：最好的决定和最好的指导来自于这样一些领导，他们在负责的技术领域有着深厚的专业知识。

竞争力领导实际上在做什么？首先最重要的一点，他们全身心投入在组

⊖ Immelt和Stewart, “Growth as a Process”, 2008。

织机构中培养杰出的技术。他们从制定好的软件开发取景框开始，包括支持架构、错误-证明过程、演进式开发和技术专业知识。他们确保实现低依赖性的架构，确保测试驱动开发和持续集成得到有效使用。他们通过迭代式开发和基于集合的设计来提供学习循环，以及设置标准，坚持代码的清晰性，确保代码复查的重点放在强化学习上。

“啊，你的意思是说指挥家！”

有一次，我们和巴西的一个成长型公司进行电话会议。这个由4个人创办的公司已经发展到了20个人，创始人意识到，他们需要一种方式来提供一致的品质，即教会每一个人好的软件开发实践。他们正在培养技术（竞争力）领导者，并试图定义一个角色，让团队领导者知道他们负责什么，也让技术领导者有清楚的职责。

我说，“好吧，请想想音乐。你有一名指挥负责每一场演出，但每一名乐手都有自己的老师，老师和他们一起改进在某种乐器上的技能。技术领导者有点像音乐老师，而产品捍卫者有点像指挥。”

“啊，你的意思是说指挥家！”我通过Skype电话会议也可以感受到他们眼睛一亮。“产品捍卫者就像指挥家。技术经理就像老师。”

“我的小舅子是一名指挥家，”Samuel说。“观察他的工作，我意识到他就像一个乐团的指挥。而且，他对乐团的所有乐手进行安排，一般他会确定一份曲谱，包含每个乐手演奏各自乐器的细节和节奏。当他们决定演奏一首新曲子时，指挥家会制定这些曲谱，从而确定乐团如何演奏这首曲子。然后，每个乐手会在家中按照曲谱独自练习。当他们再次聚在一起的时候，一般来说，就可以开始流畅地演奏新曲子了。有趣的是，他们在这之前从来没有一起演奏过，但他们却能够和谐地演奏。”

Mary Poppedieck，摘录来自

《Samuel Crescencio, OnCast Technologies》

培养技术专业知识

竞争力领导者最重要的角色可能就是老师，他指导有目的性的练习，以

培养专业知识。老师的角色在培养音乐和体育专业知识时是很重要的，在教育中也是这样：研究生在该领域的一名导师的指导下工作。老师提供目标，有挑战性的任务，以及迅速的反馈。他们帮助学生改进，提供机会让他们测试自己正在培养的专业知识。老师不一定是在这个领域的顶级表演者，但他们清楚哪些技术和培训方法是最好的，并使用这些技术和方法来指导他们的学生。

如果你希望组织机构中的人员培养他们的专业知识，充分发挥他们的潜能，你就必须提供老师。不仅仅是提供培训，而且提供老师：这些老师将工作任务分配给个人；他们寻找有挑战性的任务，每个人都能改进；他们密切关注正在做的工作并提供反馈。你可能称这些人是指导者或经理，但在这本书中我们称之为竞争力领导者。

竞争力领导者常常是产品线经理，但产品线经理并不一定是竞争力领导者。有时候我们发现，当50或60个人向一名产品线经理报告，很难想象这样的经理如何能够扮演老师的角色。谁会关注所有这些人的技术进步，关心他们的职业生涯，确保他们得到培养，以便充分发挥他们的潜能呢？谁会关注核心技术，为公司的比较优势做出贡献呢？

关键技术需要领导者。必须有人培训技术新手、设定改进目标、确保技术增强，从而保持在竞争中的领先位置。假定你的市场部门有最快的Web响应时间，或非常有能力开发特别安全的交易，或者具备某些类似的竞争优势。如何能够长期保持在竞争中的领先位置？一些人掌握着你们公司现在的专业知识，他们需要看到他们的未来在哪里，而你们将来的专业知识会掌握在新人手里，需要去招聘并进行指导，这就是竞争力领导者的职责。这是一项非常重要、非常有挑战性的工作。遗憾的是，在一些组织机构中这样的人通常不存在。

练习

1. 在你们的发布周期中，有多大比例的时间用于“强化”，即系统测试、用户验收测试和类似的活动？当缺陷第一次发现时，平均的年龄是多大？你们打算怎样让这些数字趋近于零？

2. Conway法则（设计系统的组织机构对产品的设计有约束，产品设计复制了组织机构的沟通结构）在你们的组织机构中明显吗？
3. 你认为你们的基本系统架构是低依赖性的还是紧密耦合的？如果它不是低依赖性的，请回答下列问题：
   1）开发一个中等规模的特征，需要涉及多少人？
   2）需要涉及多少团队？他们是怎样彼此沟通的？
   3）使用普通的过程，实现一个中等规模的特征有多快？
   4）有没有一些竞争对手，他们实现新特征的速度总是要快很多？
4. 观看Martin Fowler和Dan North的视频“The Yawning Crevasse of Doom”（www.infoq.com/news/2008/08/Fowler-North-Crevasse-of-Doom）。这段视频讨论了分离开发者和客户的常见错误。在你的组织机构中，有多少次工作传递分离了开发者和客户？
5. 查看下面的基本实践列表，针对每项实践为你自己或你的团队评分，分值从0至5。（0表示你从未听说过，5表示你可以在这方面讲课。）看看分值低于3的实践，想想如何改进这项实践。
   1）编码标准（为了代码的清晰性）
   2）设计/代码复查
   3）配置/版本管理
   4）一键式构建（私人和公用）
   5）持续集成
   6）自动化单元测试
   7）自动化验收测试
   8）如果测试未通过就停下来
   9）每次迭代的系统测试
   10）压力测试（应用级和系统级）
   11）自化的发布/安装打包
   12）漏掉的缺陷分析和反馈
6. 每年SANS研究所都会发布25个最危险的编程错误。下面的列表是2009年1月发布的（参见www.sans.org/top25errors/#s4）。你们的开发者意识到哪些错误？哪些错误是每个人都知道如何预防的？谁来确保你的组织机构不会

犯这样的错误？已有哪些类型的培训或指导计划？已经准备好哪些编码标准来预防这些错误和类似错误？

分类：组件间的安全交互

1）不正确的输入验证

2）不正确的输出编码或退出

3）不能保护SQL查询结构（或“SQL注入”）

4）不能保护Web页面的结构（或“跨站点的脚本”）

5）不能保护操作系统命令结构（或“OS命令注入”）

6）敏感信息的明文传输

7）跨站点的请求伪造（CSRF）

8）竞争条件

9）错误消息信息泄漏

分类：有风险的资源管理

1）不能将操作约束在内存缓冲区的边界之内

2）外部控制关键状态数据

3）外部控制文件名或路径

4）不可信的搜索路径

5）不能控制代码的年代（或“代码注入”）

6）下载代码时没有完整性检查

7）不正确的资源关闭或释放

8）不正确的初始化

9）不正确的计算

分类：漏洞预防

1）不正确的访问控制（授权）

2）使用脆弱的、有风险的加密算法

3）硬编码的口令

4）不安全的关键资源权限分配

5）使用不充分的随机数

6）优先级不够也能执行

7）客户端来保证服务器端的安全

7. 在你的组织机构中，3项最重要的技术竞争力是什么？也就是说，为你的公司提供竞争优势的竞争力是什么？谁负责确保每项竞争力都得到很好的使用、保持和增强？在这些方面，谁负责招聘新人并提供指导？

HZ BOOKS
华章图书