



.NET最佳实践

◎作者：Stephen Ritchie
◎译者：黄灯桥 / 黄浩宇 / 李永



华章科技

InfoQ

第1章

冷静待之

对于有些人而言，“最佳实践”中的“最佳”二字让他们难以接受。尽管“最佳实践”这个概念在软件开发领域已经存在多年，但是直到现在依然有许多人对于这个概念存在很多误解。将某一实践称之为“最佳实践”并不表示该实践是可以解决任何问题的灵丹妙药。适用于某一环境的实践极有可能完全不适用于另一种环境。在这种情况下，如果坚持采用该实践往往适得其反。因此，为了避免误解，本书将尽量避免使用“最佳实践”这四个字，而更倾向于使用“需冷静待之的有益实践”来描述那些适合自己而不一定适合其他人的优秀实践。该词更能表达出对最佳实践应持有的正确态度：开发人员应该对任何称之为“最佳实践”的实践保持一种怀疑态度。开发人员应该根据他所在的具体环境来选择采用何种“最佳实践”。本章会进一步讲述该词的真正含义，还将进一步阐述如何去选择适合团队的实践，还会讲述为一些需要改进的开发领域选择实践时，如何坚持这种态度。

“冷静”这个词在此是“热情”的反义词，我们用这个词来表达我们对于“最佳实践”的态度：淡然处之。“最佳”是最高级，没有什么会比“最佳”更好了。“最佳”这个词本身会传达出一种隐含的意思：这个实践是最优秀的，无需考虑其他实践了。有些人往往使用这个词来结束与其他实践相关的讨论。而现实中，应该仔细考虑那些各式各样的新颖的实践。既不应该对“最佳实践”抱有成见，也不应该盲目相信“最佳实践”。应该去选择那些适合自身和团队的实践。

“有益”这个词则提醒我们将注意力集中在应用实践之后的结果和积极产出上。尝试不同的实践意味着团队需要做出一些相应的改变。一段时间之后团队则能够获得相应的成果。问题可能会解决得更迅速，或者测试出的问题更少。整个项目的交付周期、产品质量和人际关系得以改善，客户可能对你们的工作更加满意。因此，从这一角度而言，应该选择适合你和你的团队的实践。

本书从头到尾都在讲述如何从众多新颖的各式各样的实践中选择适合你、你所在的团队以及你所在的组织的更好的实践。不管你将这些实践称之为“最佳实践”或者“需冷静待之的有益实践”，你都应该从内心深处将它们看做最适合你的实践。

按语

从众多良好的实践中挑选出适合你的最佳实践并不容易。对于许多项目而言，决定使用何种实践来帮助项目走出困境非常具有挑战性。我觉得应该避免尝试列出各种实践的优缺点来进行实践的选择，虽然很多人在用，但我认为这是不正确的。事实上我发现通过这种途径我们最终很少能够做出正确的决定。列出各种实践的优缺点并不能让我们很轻易地识别出哪种实践更优秀。对我而言，当需要选择一个新的不同于以往的实践时，我通常会问自己下面两个问题：

- 该实践应用于当前情况是否还有不足？
- 该实践是否能够轻而易举地解决当前面临的问题？

通过第一个问题，我们希望可以将那些可能不适用当前环境、风险太大或者收益充满争议的实践排除掉。通过第二个问题，我们则希望将那些能够解决现有问题、风险很小或者可能的收益很大的实践包括进来。通过这种途径，我们更容易地选择出能够帮助项目走出困境的实践。通过这种途径，那些适合我们的实践从众多实践中慢慢浮现出来，而更好的实践也可认得到采纳。

作为一名曾经在一个庞大的、持续多年的项目上工作过的开发人员，我观察到一个奇怪的现象。每过几周，就会有一个顾问花上一下午的时间给我们开发人员描述那些我们经历过的问题。渐渐的，顾问就成了众所周知的“教授”，而开发人员参加会议就变成了应付差事。“教授”提出的大部分建议往往让整个团队无所适从。我们根本不知道如何在工作中应用这些建议。这些充满理论性的建议其实并不适合我们当时的情形。我们尝试着应用了其中一些建议，最终发现它们或者完全不奏效，或者让事情变得更糟。我们花费大量时间去听取和思考这些不切实际的建议，最后发现状况毫无改善。今天我知道优秀的实践都是切实可行且都可以通过应用和成果来证明自身的价值的。我将继续坚持通过这两方面来验证一个实践是否是好的实践。

在一个早期的项目里我得到了一个非常有挑战性的编程任务。有人给了我一份非常详细的需求清单，要求我去实现它。然而这份需求清单文档写得非常晦涩难懂。里面的信息也极其复杂。于是，我决定去找编写这份清单的业务分析人员，看看能不能让他给我解释一下。后来发生的一切证明了这个决定从一开始就是一个巨大的错误。业务分析人员非常自大傲慢。他对他的工作充满了莫名的优越感，同时对我满怀敌意。他居然对我说他写的材料简单到连5岁小孩都可以看懂。于是我回答道，“那么，你就把我当成一个五岁小孩来解释你写的一切吧”。

业务分析人员愣住了，停下来思考了1分钟。然后，他的态度就改变了。他走到一块白板前，开始耐心地从头开始向我解释所写的每一个内容。他综述了整个需求的前因后果。我则一边记笔记一边提出相关的问题。当他讲得太深入或者太专业的时候，我就不断提醒他，要像给一个5岁小孩讲述一样讲给我听。我觉得冷静待之的实践直至今天仍然非常有用，尤其是在我需要厘清需求的时候，即便在面对最简单的术语的时候。

1.1 实践选择

本书即将呈现的都是和软件开发相关的一些标准、技术和专业上的习惯用法。这些都是得到广大 .NET 开发人员认可的优秀实践。不同背景或不同专业的人可能会认为还有更好的实践存在。在此需要说明的是一个“需冷静待之的有益实践”仅仅意味着如果你亲身实行该实践的话，它会给你带来一定程度的改善。相比花费大量时间探寻那些最佳实践，从众多可靠、有益的实践中选择出可以带来有效改善的实践对我们而言更为便捷高效。因此，非常重要的一点就是，我们需要掌握一条行之有效的方法来帮助我们选择合适的实践。

在《快速软件开发》(Rapid Development) 这本书中，Steve McConnell 讲述了 27 个最佳实践。除此之外，该书还介绍了许多备选的实践，并对这些实践给出了大致评估。该书详细描述了有关最佳实践相关的话题。对于那些打算应用最佳实践的商业软件开发组织而言，这本书在关于如何尝试实践方面提出了许多切实可行的建议。在该书中，如何评估最佳实践包括了 5 个方面的标准：

- 是否有助于缩短周期
- 是否有助于改善流程
- 是否有助于降低计划风险
- 一次性成功的机会
- 长期成功的可能

关于以上这 5 条，Steve McConnell 提供了完整而又详细的分析。然而正如这本书所述，你需要能够判断出哪些实践适合你和你所在的团队，哪些又不适合你和你所在的团队。这并不容易。

本章节基于分类原则提出了一个有助于个人、团队和组织发现适合他们的实践的方法。当你需要评估一个实践的时候，本书鼓励你基于下面 4 个评估条件去寻找答案：

- 可行性：在当前情况下，该实践是否现实且是否可行？
- 认可度：该实践是否已经得到广泛使用？是否已经得到清楚理解？有无相关文档？
- 价值：该实践能否有助于我们解决问题、改善交付、提高质量或者修复关系？
- 原型：有没有一个清晰的模型来描述该实践？有没有相关的示范样例？

这 4 个标准对于评估实践是否合适而言非常有帮助。它们分别从各自的角度准确地评估你所考虑的实践。如果我们不能将一个实践应用在某一个项目，这就说明对于这个项目而言该实践并不是一个更好的实践。当你需要决定是否应用某一个新的不同的实践的时候，你可以尝试上述 4 个评估条件。相比较列出各种实践的优缺点，该方法更有助于你剔除掉那些并不适用于你当前环境的最佳实践。同时该方法还能帮助你将注意力集中在那些值得你尝试的实践上。

1.1.1 可行性

毫无疑问，任何一条“需冷静待之的有益实践”首先必须是可行的。如果发现新的实践不可行，一个原因可能是你的团队还做好相关准备。比如，持续部署的前提是项目本身已

经支持自动部署。如果一个团队还没有支持自动部署，那么对于他们而言持续部署这个实践就是不可行、不现实的。我们应该选择那些在团队能力范围之内的、接近团队近期目标的实践。我们需要识别出那些阻碍我们部署这些实践的障碍，然后集中精力去解决这些问题。在该例子中，一旦我们已经支持了自动部署，那么我们就可以讨论下一步的持续部署。因此，你应该选择对于团队而言切实可行的实践。

许多时候很多实践因为缺少可行性而遭到反对。因此，当我们发现一条可行的实践的时候，我们应该马上开始尝试进行下一步并应用该实践。通过着手应用这些实践，我们可以了解这些实践如何改善我们的项目开发，同时我们也可以亲身体会这些实践。比如，很多时候我们大力提倡或命令开发人员去编写单元测试。事实上这些并不够。我们应该给开发人员一些具体的单元测试示范来告诉他们如何去编写单元测试。因此，在应用实践时我们需要关注5个方面：用户期望、开发原则（标准）、开发资源、验收标准和最终结果[⊖]。表1-1列出了对于这5个方面的详细描述。此外，该表还通过单元测试这个例子对每一个话题给出了进一步的说明。

表1-1 应用实践的相关主题

主 题	描 述	针对单元测试的示例
用户期望	指定目标、时限、数量和质量这4个方面的预期结果	通过编写单元测试来确保被测试代码可以如预期那样工作。为每一个类方法编写测试方法。为每一个逻辑分支编写测试用例。测试所有的边界和异常情况
标准	包括开发策略、流程、原则和约定。讨论那些“坚决禁止的做法”（no-no's）和可能导致错误的用法	所有的单元测试应该使用一个专用的测试框架和一个专用的模拟（mocking）框架。单元测试应遵守命名约定和符合测试方法结构。每个测试方法只能有一个主断言
相关资源	告诉开发人员如何去获得帮助，比如参考书籍、相关网站和内部标准文档	如果你需要的话，团队领导将会帮助你编写单元测试。你可以参考一些推荐的单元测试书籍，或者参考一些已有的单元测试范例
验收标准	为可接受的性能提供指南。说明监测及其度量指标。解释如何进行审核	单元测试编写完成才能说明代码是完整的。测试代码覆盖率必须符合一定的标准。测试代码必须经过审核并通过。团队可以在反思会上讨论测试代码。评审性能的时候会提供一些对单元测试的建议
最终成果	描述那些可预期的收益。将这些成果解释给整个团队。告诉大家这些改变将会带来哪些潜在的影响	通过单元测试，开发人员的调试将会更加简单。那些潜在的问题将会在更早的时候暴露出来。其他开发人员也会更容易理解那些被测试覆盖的代码。其他开发人员将会看到这些改变是如何影响系统的其他部分的

上述5个话题有助于消除对实践的模糊理解，同时也能够帮助我们更好地管理实践。通过提供上述5个方面的信息，那些新颖的实践可以更好地帮助开发人员去实行这些实践。很多时候实践没有能够得到有效应用就是因为其中某一个话题没有提供足够的令人信服的信息。实践会因为缺少验收标准而变得不一致。如果一条实践缺少开发资源这一话题，开发人员就无从知道从何开始或者遇到难处停滞不前。

[⊖] 节选自 Stephen R. Covey 的《Principle Centered Leadership》(New York: Submit, 1991)。

通过分析以上 5 个主题，我们可以对于那些感兴趣的实践进一步分析，弄清楚那些潜在的局限性。我们能够公开的讨论那些基于暗含的假设的模糊的障碍、难点。比如，以前一个曾经非常流行的观点认为花在单元测试上的时间很难反映出开发人员的开发效率。那么现在我们就可以很容易打消他们的疑虑。我们可以告诉他们衡量开发人员效率的方式已经改变，而单元测试覆盖率和测试代码审核就是该新方式的一部分。

1.1.2 认可度

每一个“需冷静待之的有益实践”都源于不止一个好的点子。因此，我们推荐采用那些在某一团队或组织已经得到广泛应用的实践。某一实践只适用于某一特定场合，如果你在另外一个场合里采用了该技巧导致你的效率下降的话，那么该实践就不是一个得到广泛认可的实践。实践的通用性非常重要。那些得到广泛应用的实践可以告诉你它适用和不适用的场合。例如，在一个相互信任的环境里，毫无疑问沟通和合作都非常顺畅，那么在这样的一个场合下，如果我们试图尝试敏捷实践就很容易成功。相反，在一个缺乏信任的环境中，大家都对彼此持有保留态度或者充满争议，那么如果我们依然尝试敏捷实践往往会使情况变得更糟，最终我们可能还不如以前。敏捷实践只有在那些认可敏捷宣言（Agile Manifesto[⊖]）的团队和组织里才有机会获得成功。因此，只有那些已经做好准备的团队和组织才能受益于这些得到广泛认可和普通使用的实践。你的团队应该能够理解应用这些实践所需要的氛围和必备前提，同时你们还需要对它们进行充分的讨论。只有这样你们才能够发现该技巧是否适合应用于你们当前的环境。

我们应用某一实践越多，就越能够获得更多支持采用该实践的经验证据。所以，你需要了解某项实践是否已经得到广泛应用。通过广泛应用，人们对于那些限制实践奏效的障碍和难点就会有更深入的理解并且形成文档。而你就可以通过文档获得该实践的足够信息和经验。先行者们会总结他们的思考，并且通过写书或者发表博客来描述应该如何应用这些实践。这些都是对实践非常重要的回顾总结。很多时候许多实践都起源于某一次尝试，然后基于进一步的理解和更多的支持慢慢发展形成了一项更为通用的实践。

在此我们可以举个持续集成（Continuous Integration, CI）的例子。早期人们聚在一起讨论自动生成以及相关的主生成脚本[⊖]。自动生成服务器先获取最新的代码，然后执行生成脚本。这样方便人们在早期发现集成相关问题。如今已经有许多广泛使用的持续集成服务器产品面世，用于执行主生成脚本，执行自动测试，进行代码分析并且自动部署软件。很显然，那些早期就开始尝试持续集成的项目已经受益良多。然而对于其他项目而言，如果也试图在早些时候尝试那些尚未成熟的持续集成技巧，那么很有可能在团队内部会产生很大的分歧，那样就带来一定程度的风险。如今持续集成已经得到广泛应用，它已经是一个普遍认可的实践，人们对此已经没有太多的异议，它的优点也显而易见。

采用那些得到广泛认可的实践还有一些其他的优势，比如你可以招聘已经熟练掌握该实

[⊖] 敏捷宣言：<http://agilemanifestor.org>。

[⊖] 最早期的有关持续集成的文章：<http://martinfowler.com/articles/originalContinuousIntegration.html>。

践的开发人员。另外一个优势就是你可以很容易得到管理者的大力支持并获得团队的认可。总而言之，采用那些已经得到广泛认可的.NET 实践和原则，使你可以从他人身上学到相关的知识和经验。

1.1.3 价值

每一条“需冷静待之的有益实践”都应该告诉人们在应用该技巧之后所能获得的价值。价值本身是一个很主观的东西。一名开发人员在其环境中应用实践之后所获得的价值肯定和另一名开发人员在另外一个完全不同的环境中获得的价值不同。如果开发人员对需求已经非常清楚，了解得非常详细完整，而我们仍然召开一个需求评审会议，那么毫无疑问这个会议就是毫无价值的。而当程序员对于一份非常复杂的需求毫无头绪的时候，增加一次需求评审会议则非常有价值。对于第二个案例而言，所预期的成果就是帮助开发人员澄清需求。而通过应用需求评审这个实践就可以帮助我们达到预期的目标。因此，我们应该重新审视一下单个开发人员、团队或者组织当前所能获得的预期成果。如果你希望能够获得更好的成果，那么你应该尝试一些实践。对于你而言，那些有助于你获得更好成果的实践都是更有价值的实践。

整个团队可以一起思考一下：如果我们想要达到所预期的目标，我们还欠缺什么。比如，一名开发人员打算解决一个缺陷。他可能需要一些重现缺陷的步骤或者一些其他的欠缺信息。如果对于项目而言该缺陷是一个系统级别的问题，但是该问题又无法重现，那么尝试一个有价值的实践就会改善当前状况。该实践可能会定期提供一些更有效的重现步骤，或者帮助改善开发人员的调试环境。总而言之，你必须首先知道你或你团队目前所欠缺的东西，然后你才能够发现那些对于你或你团队而言有价值的实践。

你也可以通过分析当前成果和预期成果之间的差距来帮助你发现那些有价值的实践。比如，团队领导们可能会对编程标准（coding standard）感兴趣。团队领导们认为编程守则对项目大有裨益，因此他们希望能够发现一条推广编程守则的途径。我们在第11章中讨论的代码分析技巧会在这方面起到作用。比如，StyleCop 工具会帮助开发人员坚持编程守则。同时它提供了代码自动监控功能。因此，你可以通过了解你所想达到的目标来寻找那些对于你而言有价值的实践。

1.1.4 原型

每一个“需冷静待之的有益实践”都应该提供清晰的示范样例。开发人员可以把该示例作为实践的模型来遵循。对于开发人员和团队而言，仅仅有概念并不足以让他们去实际操作。绝大部分开发人员都希望能够提供具体的示例来帮助他们理解概念。这些示例能够帮助开发人员将概念和现实联系起来。一些新的不同的实践可以通过示例来告诉开发人员如何在他们的项目中应用该实践。作为团队带头人，非常重要的一点就是你需要发现或自己开发出某实践的实现原型。这样会更好有助团队成员遵循实践。此外，通过实现实践的原型，可以进一步提前发现可能存在的障碍难点，从而可以帮助我们更快、更好地实施该实践。

我们可以重新回头去看看持续集成实践这个例子。创建持续集成原型的第一步就是选择一台持续集成服务器。我们在第 10 章讲述了许多值得你尝试的持续集成服务器。下一步我们需要开始安装持续集成服务器相关的软件、建立版本控制系统设置、编写构建脚本和创建通知机制。通过这么一个实验性的项目，我们可以增进对持续集成这一实践的了解以及各个组件如何一起协同工作。此外这个实验性项目也告诉我们如何沿着“教程”一步一步往下实践。

原型的一大优点就是它会帮助我们及早识别出应用实践中可能会碰到的障碍难点，并将其移除。通过原型，实践就变得看得见摸得着。这一点谁都无法否认。当我们打算尝试一些新的实践来改善项目状况的时候，我们需要一些关于该实践的实实在在的示范样例。而在一点上原型可以切切实实地帮助我们。回到持续集成这个例子上，当有人问起关于通知机制、安全或者成本等一般问题的时候，我们就可以举出一些具体的例子来回答这些问题。人们可以通过这些具体的例子来实实在在地体会到这些实践如何改善我们的状况。

因此，在你完全赞同或打算实施某一新实践之前，你可以花一些时间按照该实践提供的相关信息实现一个小例子。如果该实践确实是一条“需冷静待之的有益实践”，那么该实践就会提供一个非常完整的示例原型。

1.2 关注需要改善的目标领域

我们可以从很多方面着手来改善软件开发。一些经理认为可以去公海上开发而降低成本。这个建议非常不明智。它会给产品交付、软件质量和团队关系这三个方面带来负面影响。事实上，任何恶化软件开发的变化都源自于这三个方面。我们经常听到的抱怨就是某些团队无法按时交付满足商业需求的产品，还有一些抱怨则涉及产品质量和延时交付。

每一个“需冷静待之的有益实践”都会关注于如何改善下面三个重要领域中的某一方面：

- 产品交付
- 软件质量
- 团队关系

一般而言，我们有两种帮助改善以上三个领域的方法：一种方式是帮助处理相关问题或者减少问题的发生；另外一种就是通过创新改变工作方式。某一问题得以解决往往包括发现问题、分析问题、处理问题和减少问题这四个方面。在绝大多数情况下，人们很容易发现问题的存在，人们非常清楚问题的存在也愿意承认这是个问题。创新常常包括主动性、新颖的方式、点子和创造力。创新通常意味着改变。然而改变并不总是很容易发生或受到欢迎。优秀的实践或者有助于解决问题，或者带来创新。人们发现尝试有助于解决问题的实践时风险很小，而伴随创新的实践则会带来长期的利益。

每一个“需冷静待之的有益实践”都会有助于你或团队的能力得到提升，从而你或团队花费更少的时间和精力来解决问题。比如，某一个实践可能会帮助开发人员更方便地调试程序、更快地定位问题根源和更有效地分析问题。图 1-1 通过一个概念图展示了优秀的实践给团队花费在解决问题的时间上的变化。通过图 1-1，我们看到在项目早期，大量的时间和

精力花费在处理各种各样的问题和争论上。尝试了一些实践之后，花在这方面的时间得以减少。再之后，团队又尝试了一些其他的良好实践。整个状况得以明显改善。团队花费在处理问题、调试程序上的时间明显减少。更多的时间用在了那些更重要的事情上。整个团队的能力也得到了明显的提升。

每一个“需冷静待之的有益实践”都有可能通过改变原有工作习惯来帮助你或者你的团队的能力得以提升。例如，开发人员通过尝试一种新的实践来提高编写代码的效率。这就意味着开发人员可以更快地实现一个功能。图 1-2（示意图）描述了更好的实践在实现预期结果的同时给开发人员的生产力所带来的变化。在项目早期，项目通过每一次冲刺（sprint）来衡量团队的产出。后来团队通过尝试一些良好的实践来提高自己的生产力。在应用了一些实践之后，团队的生产力得到明显提高。他们的每一次冲刺都可以完成更多的任务。尝到甜头之后，团队更加愿意尝试一些新的实践。他们的生产力因此可以继续得以提高。团队通过创新提高了工作能力。

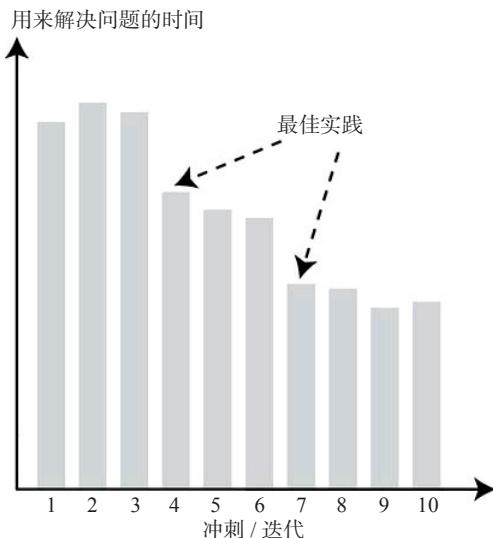


图 1-1 优秀的实践给团队花费在解决问题的时间上所带来的变化

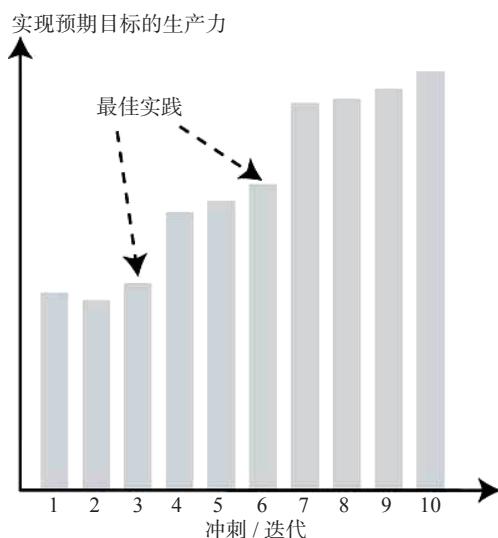


图 1-2 实现预期目标的更好实践在生产力方面的变化

1.2.1 产品交付

需冷静待之的有益实践众多目标之一就是帮助你改善项目交付状况。优秀的实践可以帮助改进团队的工作效率和工作能力。团队按期交付的能力得以提高，而且团队有可能完成得比预期承诺的更多。优秀的实践通过多种方式来改善项目交付状况，比如帮助更好的解决问题或者引进创新。

导致项目交付延迟的一个原因是团队花费了大量的时间去定位和解决软件问题。如果团队能够更快地定位问题、解决问题，团队就能够节约大量时间去完成更重要的任务。团队整

体的产出就能得以提高。例如，编写单元测试的一个重要测试点就是去测试那些边界条件。因此，当给某一方法编写单元测试的时候，团队会谨慎分析那些边界条件并仔细考虑方法的参数。开发人员会据此编写一系列相关的单元测试来确保被测试的方法已经考虑了这些边界情况。此外，测试代码在方法被调用之前还给该类预设了许多期望的或者不期望的前提条件。边界测试的目标就是通过测试被测试方法的一些边界条件来更早地发现问题。开发人员在编写生产代码的时候就可以提前考虑这些问题。这样他们就可以确保这些发现的问题已经在代码中得到妥善处理。因此，单元测试实践可以提前帮助开发人员发现问题。开发人员就可以更主动地解决问题。此外还有许多其他的实践可以有效地帮助开发人员在早期开发过程中发现软件相关的问题。

另外一种可以改善项目交付的做法就是尝试创新。例如，一些软件开发工具可以极大地提高开发人员的工作效率。如今许多集成开发环境（Integrated Development Environment, IDE）都内置了代码编辑器、编译器、连接器和调试器。这种集成开发环境为开发人员提供了从编写代码到执行代码这种完全不同于以往的崭新开发方式。然而在此之前，从编写代码到最终执行代码需要花费很长时间。显然，这种延迟制约了开发人员的工作效率。微软的 Visual Studio IDE 就是一款得到广泛使用的集成开发环境。通过使用该工具，众多开发人员可以更快地交付更好的软件产品。除此之外还有很多通过引入创新方法来提高工作效率的工具，比如 ReSharper 和 Visual Studio 2010 Productivity Power Tools，这些工具都能够帮助开发人员改善生产效率。

1.2.2 软件质量

每一个“需冷静待之的有益实践”都会有效帮助你改进软件质量。更好的实践可以帮助改进系统功能和产品质量。同样和帮助改进其他领域一样，实践也是从两个方面改进产品质量：帮助解决问题和创新。

人们经常提及的一个质量问题就是那些在回归测试中暴露出来的缺陷。这些缺陷被解决后，过段时间又暴露出来。这种类型的缺陷极大地打击了人们对这款软件的信心。开发人员和测试人员之间也渐渐失去对彼此的信任。这时候我们就需要尝试一些可以帮助开发人员避免引入这些回归缺陷的实践。例如，我们可以引入一些自动化测试来验证那些已经解决的缺陷在新的版本中是否重现。你需要为每一个已经记录的缺陷编写测试代码。开发人员首先要编写可以稳定重现缺陷的测试代码。为了确认该缺陷已经被解决，我们需要保证和该缺陷相关的测试用例全部通过，并且其他的自动化测试用例也全部通过。接着我们新编写的测试用例也成为了整个项目的自动化测试用例集的一部分。我们需要保证所有的自动化测试用例在持续集成的时候要全部通过。如果某一测试用例突然失败，持续集成服务器将会停止构建。开发人员必须要在测试人员开始测试之前解决该问题。

此外团队也可以引入新的工作模式来提高产品质量。新的工作模式不一定要完全不同于以往。它只需要给你的团队和组织的工作模式带来新鲜的空气。例如，团队已经开始尝试应用工程分析这个实践。但是仍然有一些团队未在开始编写代码之前完成对需求的基本分析和软件设计。在工程分析这个实践中不可省略的一步就是开发人员必须在编写代码之前对需求

有着非常清楚的、完整的理解。另外非常重要的一步就是将解决方案通过图表的方式表示出来，这样团队就可以对将要开展的工作制订合适的计划。下一步就是团队一起对软件设计进行审核，这样团队就可以保证该设计完全吻合需求。工程分析主要用来帮助你找出需求理解偏差和设计上的不足之处。此外，因为你在编码之前已经有了合适的解决方案，因此它还可以有助于你加快开发速度。当团队已经做出得到一致认可的计划之后，开发人员就可以通过自己最擅长的方式来高质量地完成分配的任务。

1.2.3 团队关系

每一个需冷静待之的有益实践都会有助于改善团队内和团队之间的相互关系。如果我们将不同人员之间的互动看做软件开发的核心，那么我们就可以通过团队内部的关系状况来衡量不同人员之间的合作互动。如果团队内部关系比较差，那么团队内部的沟通也就比较少，团队的工作能力也有可能会因此下降。而如果团队内部非常和谐，那么团队内部的交流也会非常顺畅，生产效率也会因此而得到显著提升。团队领导将会负责维护团队内部的关系以及和团队之间的友谊。通过维护这种关系，团队内部和团队之间将会彼此信任，从而一起为项目的成果而努力。相反，如果一个团队内部互相缺乏信任，彼此勾心斗角，毫无疑问，在该项目上将会有很多问题产生。

改善团队内部或团队之间的紧张关系非常不易。很少有人愿意去蹚这趟浑水。然而当团队内部和团队之间交流得不是那么理想的话，我们就应该进行必要的干预。我们需要尝试一些新的实践来解决这个问题。例如，业务分析师、开发人员和测试人员相互之间的关系对于创建出一款伟大的软件而言非常关键。达成对软件需求的共识的实践可以帮助团队改善彼此的关系：不管有什么其他差异，团队应该认识到软件的特色和功能是组成该团队的基础。整个团队必须认识到，无论他们之间有何偏见，软件的功能需要整个团队一起通过交流来实现。团队之间再也不依靠文档来互相传递信息，而是通过交流的方式共享信息，比如团队可以一起站在白板前面讨论问题，这种更柔的方式打破了以前那种僵硬的工作方式。团队内部更加活跃地一起讨论相关问题。通过这样的改进，团队加深了对于问题的理解，各方面的合作更深一步。

我们也可以适当改变工作方式来改善团队之间的关系。当团队内部已经很和谐的时候，适当的改变将会让团队内部更和谐。这其实也是敏捷背后的关键思想之一。例如，很和谐的团队每天的站立会都会很短。每个人都能从站立会上得到关于项目的关键信息，同时妨碍项目继续进行的一些障碍、难点也会得到充分讨论。问题就有可能在早期得到解决。敏捷一直提倡的这种信息的迅速交流和尽早解决问题的做法将会提升团队的工作效率。你很容易就能看到这种改变在团队内部带来的变化。每一个问题都有人在跟踪，并且能够迅速地得以解决。需要注意的是这些敏捷实践并不能够每次都奏效。如果在错误的环境中尝试这种实践，那么它对于团队而言毫无帮助。只有整个团队充满融洽的氛围，彼此之间亲密无间，同时又有着相同的追求，那么站立会将会改进团队整体的工作效率。

1.3 整体改善

除了能够给以上几个领域带来改善之外，需冷静待之的有益实践还会给整个团队带来整体性的改善。一个良好的实践能够同时改善多个开发领域。例如，添加单元测试的同时也减轻了测试人员的压力，测试人员发现、记录和验证的缺陷也减少了。另外一些实践则帮助人们在关于某一开发领域究竟需要花费多少时间上做出正确的决定。这些技巧帮助团队减少了无用的争吵，将宝贵的时间花费在正确的事情上。

优秀的实践还可以帮助团队变得面目一新，并可改进项目的可持续性。整个团队和组织会因这些积极的改进成果而雀跃。人们会用更乐观的眼光看待项目的未来。项目将会变得更具有生气。团队会因为工作效率的提高而更积极地参与到各项工作中。每一个团队成员都通过自身的努力将更多的创新和活力带入到团队中来。良好的实践就是通过这样一种方式帮助团队可持续地向前演进。

1.3.1 均衡

每一个需冷静待之的有益实践帮助团队在多个开发领域均衡改善。如果在开发过程中忽视了某一方面，那么团队就无法从那一方面受益。如果所有的精力都花费在某一领域，那么团队就无法得到相应的回报。通过这种均衡工作方式，团队意识到需要在各个开发领域分配对应的时间和精力。团队还能够意识到分配给某一领域的时间过少和分配给某一领域的时间过多所带来的风险。总而言之，重要的一点就是你需要能够认清某一开发领域的目的和基本原理，这样就可以确保你在这一领域分配了合适的时间和精力。此外，通过多个开发领域之间的协调互补团队可以获得更多的收益，从而整个项目获得整体性的改善。

表 1-2 列出了五个软件开发领域以及如果某领域给予的关注太少或太多所带来的后果。该表还列出了处在平衡点所带来的优点。优秀的开发实践能够同时改进多个开发领域。例如，单元测试实践就会为需求、设计、开发、质量和管理这五个方面同时带来改进。当开发人员开始编写单元测试的时候，他们就会对需求进行审核。这样有助于解决需求的不完整或者不清晰的问题。这时业务人员就会参与进来继续澄清需求直到开发人员满意为止。单元测试还可以在增加新功能的时候帮助改进设计。自动化测试用例将会保证所测试的代码正确地实现了需求。另外，通过编写测试用例，开发人员可以在早期发现潜在的缺陷，然后及时修复它们。软件质量因此得以提升。最后，单元测试可促使开发人员思考代码如何与外界进行交互。这样开发人员就可以有效地安排他们的开发计划，从而可以和其他团队进行高效的沟通以及更好地管理他们的工作。从上述例子我们可以看到单元测试是如何同时在多个领域产生影响并改善团队的工作的。

表 1-2 均衡关注多个开发领域

领 域	关注太少	平衡点	关注太多
需求	需求不完整，不清晰，不一致，充满错误的决定	团队之间对于需求的理解、目标和优先级达成一致	需求清单过于详细，要求过多

(续)

领域	关注太少	平衡点	关注太多
设计	设计不充分，整体不一致，充满困惑	有基础架构，完整，清晰，一致	缺乏弹性，规则太多，无谓的一致性
开发	效率偏低，缺少产出	高效，有目的，有较好的正确性	过于追求完美，项目延期
质量	存在潜在缺陷，忽略了系统级的问题	缺陷、问题得到妥善处理，系统可靠性得到提高	过于吹毛求疵，关注不重要的细节，太过谨慎
管理	失去控制，争斗，负担过重	计划妥当，合作，协调，以目标为导向	墨守成规，过于担心风险，官僚主义

1.3.2 面貌一新

良好的实践可以给团队和组织带来一种面貌一新的感觉。如果一名开发人员待在一个死气沉沉的团队里，他也会变得毫无生气，感觉不到一丝希望。同样的问题一遍又一遍地发生，而整个团队却无动于衷。对于每一个开发人员而言，他们每天筋疲力尽地应付着这些问题。整个团队气势低沉，人们很难去参与那些会给他们带来积极变化的改变中来。此时我们就需要在这个团队里推广一些新的实践。这些实践会给整个团队带来新的局面。即使是小范围的改变都会给整个团队带来希望，比如改善开发环境。团队因此会变得乐观起来。在意识到了某一种实践所带来的积极的变化之后，人们就会开始关心下一个值得采纳的实践。然后就是下一个，下一个。随着越来越多的有益于恢复信心的实践得到应用，整个团队的士气开始恢复，生产力得以提高。

很多开发人员都对新事物非常感兴趣。尽管有一些人不愿意接受变化，仍然有许多开发人员愿意去学习新的技能、工具或技术。所以在采纳一个更好的实践很关键的一点就是你需要找到正确的人来起带头作用。他将负责首先尝试这些新的实践。例如，自动化部署给开发人员提供了一次学习部署脚本和与之相关的自动化工具的机会。如果有一名开发人员在手动部署中遇到许多问题并且他不得不解决这些问题，那么他就会主动去学习这些自动化部署技术以此来避免这些问题反复发生。而对于另一名开发人员，他没有遇到类似手动部署的问题，那么这个时候就主要依靠他自身的主动性来尝试这些新的自动化部署技巧。在这两种情况下，随着他们尝试新的实践之后并且改善了部署流程，他们对项目的信心因此得以提升。最终整个团队的部署就会更加稳定，开发人员也会更有成就感。

尝试良好的实践会给团队带来更多的欢乐。开发人员享受每一个帮助他们减少麻烦、提高生产力和改变工作方式的实践。只要团队充满欢乐，团队就获得了新生，从而团队就会更加有责任感，也会做出更多的承诺。

1.3.3 可持续性

每一条需冷静待之的有益实践都可以帮助团队保持良性发展的势头。在一个不可持续的环境里，团队会乱成一锅粥，各种各样的问题层出不穷。例如在一个缺少计划的团队里，很有可能会给开发人员分配过多的任务，同时还要求他们在短期内完成。在这种情况下，整个

团队都面临着巨大的压力。很多事情都一带而过。例如，项目的架构在短期内就被草率地制定。而这样的架构并没有经过仔细的通盘考虑，四处充斥着漏洞。而处于压力下的开发人员只能不经思考地编写代码。最终的结果就是层出不穷的缺陷和开发人员四处救火。这样的项目最终无法持续下去。此时该团队就需要尝试一些新的实践来帮助他们更好地避免问题解决问题。更好的实践可以帮助团队在计划安排、不同部门之间的协调以及团队的生产力方面得以改善。将这些实践一起应用在团队里面可以建立起一个可持续的开发环境。

很多着急追赶最后期限的团队都很容易将与代码集成相关的问题推迟到后期再解决。一些开发人员并没有经常提交代码的习惯。另外一种情况就是开发人员很少基于最新的代码来测试新写的代码。这样做的一种原因是他们担心会遇到代码集成问题。他们宁愿在自己的代码全部完成后再去解决代码集成问题。还有一些程序员经常提交一些会终止构建的代码。因此团队带头人需要经常解决这种一团乱麻的状况。我们将该团队遇到的这种困境称之为集成陷阱[⊖]。随着代码行数越来越多，代码集成遇到的问题就越来越多，整个项目面临一种无法继续下去的局面。那么这个时候团队该怎么办呢？一种办法就是改变开发的方式。团队带头人必须要求所有开发人员在每一天结束之前将他们当天编写的代码提交出去。第二天团队带头人将最新的代码复制到一个新的文件夹内，然后开始重新构建。如果构建过程中出现问题，那么开发人员需要尽快找出构建失败的原因然后快速修复它。这种坚持每天集成的习惯将会避免团队在错误的道路上一直走下去。更好的做法就是配置一台持续集成服务器。它会基于最新的代码进行自动构建。持续集成非常符合许多项目经理的想法：自动化、监测、控制、后果和责任心。这种做法更吸引人的一点就是它可以在代码提交的五分钟之内发现所提交的代码是否会终止构建。

通过尝试这些优秀的实践，开发人员将会对职业生涯变得更有信心。学习更好的实践毫无疑问是助益开发人员职业生涯、保持开发人员技术水平得以继续提升的一个重要因素。尝试提高效率的实践可以得到更好的结果。例如，通过购买一些可以提高编写代码速度和质量的工具，开发人员的效率得以提高，新的软件功能就会得以更快地完成。开发人员也会因为自身效率的提高而觉得自己的价值得以提升。持续对团队做出积极贡献可以极大扩大开发人员的声望。开发人员的职业生涯将会长期受益于此。而对于团队领导而言，每一名团队成员的提升同样有助于你的职业生涯更进一步。

可持续性是与每一名开发人员、每一个团队和组织的长期命运紧密相连的。通过尝试优秀的实践可以有效地帮助团队走出困境，并且树立每一位团队成员对明天的信心！

1.4 小结

本章告诉每一位读者，当你打算尝试新的不同于以往的实践的时候，第一步你应该去了解项目的当前状况。你应该尝试那些适用于你当前状况的切实可行的实践。你应该首选那些已经得到广泛应用同时文档详细的实践。你所选择的实践要能够有针对性地解决你当前所面

[⊖] 一篇关于集成陷阱的非常不错的文章：<http://c2.com/xp/IntegrationHell.html>。

临的问题，或者通过改变工作方式改善你所在团队的项目交付、产品质量和团队关系。你所选择的更好的实践应该有一个清晰的原型，这样其他人都可以通过示例来学习如何在他们的日常工作中应用该实践。

本书讲述了许多需冷静待之的有益实践。这些实践广泛适用于单个开发人员、团队和公司组织。这些实践既着眼改进整体开发环境，又关注重点开发领域。你应该理智地仔细思考这些实践。你必须基于当前的环境来选择适合的实践。如果要对某一实践做出一些调整，或者确实需要放弃应用某一实践，千万不要犹豫。如果需要你或你的团队做出改变来适应某一新的实践，那么请勇敢地迈出第一步。你应该坚信，这些更好的实践将会给你和你的团队带来更好的成果！

第 2 章

.NET 实践领域

本章旨在找出那些可以尝试更好的 .NET 实践的领域。本章主要关注 .NET 开发领域和通用软件开发领域。在这些领域中，我们可以有机会去发现或者学习优秀的实践。本章试图为你提供一个大而全的领域列表。一般来说，我们可以在以下方面来尝试良好实践：

- 了解所发生的问题、争议、缺陷或者严重故障。
- 着眼于开发过程中发生的相关联的事件和开发行为。
- 寻找那些我们可以适用的设计模式、编程惯例和开发守则。
- 应用新的工具和技术。
- 尝试新的点子、新的工作方式和研究。

在尝试那些广泛使用的工具和技术之前，还有一些 .NET 实践需要我们关注：自动测试、持续集成和代码分析。这些实践非常重要。本书在后面几章专门讨论这几个相关话题：

- 第 8 章中将会讨论自动测试。
- 第 10 章将会讨论持续集成。
- 第 12 章将会讨论代码分析。

本章即将谈及的一些 .NET 实践同样也很重要，非常值得你进行进一步探索。希望你能够在本章中获得与这些 .NET 实践相关的足够信息，这样你就能够更好地准备下一步探索。下一步你应该探索那些已经超出本书范畴的领域。附录 A 中为你提供了相关的资源和参考。

关于实践，我们所关注的并不局限于如何选择实践、如何应用实践以及如何评估实践的实施效果。在后面几章我们还将讨论关注结果、价值量化和实施策略的重要性：

- 第 3 章将会讨论有关交付、完成和展示积极产出。
- 第 4 章将会讨论实施良好实践之后如何量化所带来的价值。
- 第 5 章将会讨论如何鉴别良好实践所带来的战略意义。

本章有关实践相关领域的讨论将会列出一些方法来帮助我们选择合适的良好实践。在相关的案例中我们将会重点强调那些强力推荐的实践。表 2-1 列出了本章中推荐的相关实践。希望这些需冷静待之的有益实践对于你、你所在的团队和公司组织而言都是切实可行的。

表 2-1 能够改善团队的需冷静待之的有益实践

	策 略
2-1	定期监测和跟踪技术债以此发现那些疏忽或潜在的缺陷
2-2	评审和分类跟踪系统中记录的缺陷以发现更好的实践
2-3	通过反思分析来发现新的不同的实践
2-4	通过前瞻性分析会议来预见潜在的问题并解决
2-5	查找系统级问题和解决方案以改进应用程序生命周期
2-6	领悟《.NET设计规范》中关于.NET框架设计背后的深意
2-7	利用微软模式和实践小组提供的资源
2-8	紧随微软研究院的创新成果
2-9	关注.NET实践领域新兴的自动化测试生成工具
2-10	尝试通过契约式编码来提高程序的可测试性和软件的可验证性
2-11	重视安全应用开发和微软安全开发生命周期(Microsoft Security Development Lifecycle)

按语

很多软件项目都难以做到按时交付。当我加入一个已经错过交付期限或者被其他严重问题所困扰的项目团队的时候，我所做的第一件事情就是从头到尾浏览一下缺陷跟踪系统里的所有记录。我会大致阅读一下缺陷报告里所记录的内容，然后将它们分类并按照优先级排序。此后我的脑海中就能够清晰地浮现出相关问题的产生模式。通常一个项目中经常存在许多开发人员不能理解的功能模块。此外还有许多特别容易出错、需要重新设计的软件模块。通过分析已经暴露出来的缺陷、争议和问题，我们能够选择出那些适合我们的实践。通过实施那些更好的实践，我们的项目就能够形成良性发展。

在我早期参与的一些.NET项目中，我经常需要编写一些类库模块。我经常对如何选择一个合适的实践而感到迷茫。我主要寻找那些与命名惯例、正确地使用.NET框架以及其他开发模式相关的实践。其间偶然发生了对我日后工作产生重大影响的两件事物。一个就是一本名叫《.NET设计规范》[⊖](Framework Design Guidelines)的书，我们将在本章的稍后章节详细谈论这本书。另一个就是FxCop软件工具。我们将在第11章讨论该工具。我仔细阅读了《.NET设计规范》的每一章，从中我了解到.NET框架是如何构建的。这本书同时还记录了设计构建.NET框架团队的思考结晶。通过这本书我们可以学习到许多非常重要的.NET实践，它是一项非常重要的资源。FxCop工具可以看成是对《.NET设计规范》的补充。它会帮助你对.NET程序集按照事先指定的规则进行一一检查。我从FxCop中学到了很多实践，并深信将来还会从中学习到更多。总而言之，《.NET设计规范》和FxCop互为补充，它们给我们的开发提供了许多非常实用的.NET实践。我经常将这两者作为重要的学习材料和实践资源推荐给开发人员。

时不时会有人让我推荐一些重要的.NET实践领域。许多开发人员让我推荐一些他们应该学习的工具和技术。遇到这些情况，我主要推荐那些与以下四个方面有关的工具和技术：

[⊖] Krzysztof Cwalina 和 Brad Abrams 编写，《Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition》(Upper Saddle River, NJ: Addison-Wesley Professional, 2008)。

自动测试、持续集成、代码分析和自动部署。近些年来在以上领域，业界已经取得了非常大的进展，新的产品陆续问世，一些令人激动的产品在迅速地得以改进。常用的工具集已经能够提供那些新的不同的实践技术。例如，如果你卷起袖子打算在 Team Foundation Server 上大干一场的时候，你就有机会在上述四个领域学习到许多实践经验。因此，我对那些蠢蠢欲动的人们的建议就是抓住机会对那些产品做一次深入的研究。你应该评估相关的产品，然后对它们一一比较，互为参照。通过这种方式，那些新颖的实践领域就会自然而然地出现在你的面前。

2.1 从内部挖掘

其实，在你的团队、项目或者组织内部有一座富含各种实践的金矿。你可以通过挖掘团队、项目或者组织内部存在的问题、争议、缺陷或严重故障来发现那些有价值的实践。这座金矿远远比那些外部的实践来源更有价值。因为你从内部发掘出来的实践：

- 具有不可否认的重要作用：这些实践对你的团队而言毫无疑问是非常重要的。你的团队对此也容易产生兴趣；
- 非常及时：你的团队非常需要这些实践来解决最近发生的问题；
- 容易接受：这些实践对你团队而言可以说是量身打造的。它们非常吻合团队目前的状况。

从团队自身挖掘出来的更好的实践在实施的时候不会引起太多的反对意见。这主要是因为团队自身就迫切需要这些实践。例如，如果我们根据对缺陷跟踪系统里记录的缺陷分析发现了一些实践，那么这些实践就会显得非常重要，因为它们会帮助项目减少缺陷。这种发现实践的方式显著地降低了实施实践的难度，人们会转而关注这些实践的实施效果如何。因为这些实践来自于团队内部某一处，随着该处的改善我们就很容易看到这些实践所带来的效果。例如，某一实践随着分析缺陷跟踪系统而产生，在实施该实践之后，我们可以通过观察缺陷跟踪系统所汇报的缺陷数目来衡量实践是否奏效。我们将在第 4 章详细讨论如何量化实施实践所带来的价值。

2.1.1 技术债

“技术债”是一种隐喻的说法[⊖]。它由 Ward Cunningham 创造，用来帮助理解和跟踪软件中那些有意或无意的架构设计和代码缺陷。例如，我们可以将一个过于简单的设计视为技术债，我们也可以将一个轻率设计的充满缺陷的架构视为技术债。如果有一段代码并没有如所期待的那样工作，那么这段代码可以认为是技术债。债务本来是与公司财务相关的。像处理财务债务一样，你同样需要对技术债做到收支平衡。不妥善的设计和代码所造成的技术债需要花费额外的精力来处理，而这些往往会令人沮丧。在“债务”这种隐喻的用法中，我们将那些需要回滚、修复和改善的设计和代码所花费的时间精力称为“本金付款”。

[⊖] 更多信息可参见 <http://c2.com/cgi/wiki?TechnicalDebt>。

有一些项目会很明智和慎重地处理这些技术债。他们会正确地看待这些技术债，并且能够理解和看重技术债所带来的后果。然而另外一些项目则完全不同。技术债被扔在一旁，无人理会。因此，团队可以通过仔细分析、研讨所存在的技术债来发现一些新颖的可以帮助改善现状的实践。事实上，“技术债”本身这个词就是一个有助于帮助团队关注以下更好的实践的有效方式。

并不是所有的技术债都同等重要。软件设计中的小瑕疵所带来的影响可以忽略不计，而重大的设计缺陷则需要大量的后续工作。例如，如果在前期忽略了本地化这个需求，那么后期为了支持本地化，成千上万行的代码需要重新修改。这些都是非常昂贵的代价。此外，修改本身又是一项容易出错的行为。

在此我们需要了解技术债分两种：

- **设计债务：**过于谨慎的抉择、忽略基本的需求、没有充分考虑的架构和匆忙的开发都会造成设计债务。大部分时候团队需要解决所有的设计局限。
- **代码债务：**工作在一个充满压力的环境里可能会造成代码债务。采取简单短期的解决方案、缺乏周详考虑的代码也会引入代码债务。通常这些代码都需要重新修改和改进。

因此，你的团队应该花一些时间来评审你们所制定的设计。设计中有没有需要及时解决的不足之处？哪些优秀的实践可以帮助团队避免或者修复这些设计中的缺陷？你应该尝试那些可以保证设计债务不再累积的实践，你也可以选择那些在问题爆发之前就可以帮助你解决设计债务的实践。

你的团队还应该定期评审所编写的代码。哪段代码需要重写还是改进？哪种实践可以帮助你的团队来编写更好的代码？团队应该尝试那些可以帮助团队编写更好代码的实践。

实践 2-1 定期监测和跟踪技术债以此发现那些疏忽或潜在的缺陷

当技术债逐渐累积的时候，你应该把它们一一记录下来[⊖]。这可以适当提高团队对技术债的认识。此后，团队需要花费大量的精力来处理这些技术债。团队应当根据所记录下来的信息来列出逐步解决技术债的计划。即使处于巨大压力之下，团队也应该根据之前达成一致的计划按部就班地解决技术债。总之，为了更好地交付软件产品，我们可以通过监测、分析技术债这一途径来发现和实施那些更好的实践。

2.1.2 缺陷跟踪系统

一般来说，缺陷跟踪系统主要用来收集来自于测试人员和客户的反馈意见。这些反馈意见通常反映出某些地方存在缺陷。如果你打算改进整个开发流程，你会从缺陷跟踪系统中得到大量有用的信息。通过分析缺陷跟踪系统里汇报的缺陷，你能发现许多提醒你某些领域需要改进的共同问题：

- 需求不明确或者不清晰；
- 设计不完整或不充分；

[⊖] 根据技术债的特性进行分类。请参见 <http://martinfowler.com/bliki/TechnicalDebtQuadrant.html>。

- 大量容易出错的模块，这些模块都需要重新设计或重写；
- 难以理解和维护的源代码；
- 缺陷报告不完整或不清晰；
- 由于缺少细节，客户汇报的问题难以重现。

实践 2-2 评审和分类跟踪系统中记录的缺陷以发现更好的实践

你很容易在缺陷跟踪系统里观察到与同一功能、同一模块或者同一代码文件相关的大量缺陷。这些缺陷都是某些事情没有妥善完成的产物。对这些缺陷放在一起进行汇总分析会给你带来一些新颖的实践的灵感。例如，如果有相当多的缺陷都是由于不清晰或者不完整的需求造成的，那么这些缺陷就在提醒你的团队需要改进需求分析和软件设计。当然，改善对于需求的理解是不够的，要想彻底根除这种类型的缺陷依然任重道远。

有些情况下记录在缺陷跟踪系统里的缺陷本身反而成了新的问题。这些缺陷记录可能缺少重现问题的步骤。还有一些情况下，客户无法提供有关缺陷的足够信息，开发人员不得不去猜测引起这个问题的原因。有一些更好的实践会帮助改善这一类问题，比如规定测试人员需要填写完整清楚的缺陷报告。还有一些其他的办法也可以改善这一类问题，比如在软件中添加系统记录和调试信息来帮助开发人员更容易地定位问题。

2.1.3 反思分析

在每一个敏捷冲刺（sprint）或者项目迭代的末尾，团队都需要召开一次反思会（retrospective meeting）。在每一个项目收尾的时候，团队同样需要召开一次反思会。在会上团队需要讨论过去的一段时间内做得不错的地方和需要改进的地方。这些会议的主要目标就是找出那些需要改进的地方以及在将来的迭代或项目中可以采取的改进措施。这里有一条称为 4L's Technique[⊖]的技巧。该技巧主要关注下面四个主题和相关问题：

- 喜欢：在过去这一迭代周期内我们喜欢什么？
- 学习：在过去这一迭代周期内我们学到了什么？
- 缺少：在过去这一迭代周期内我们缺少什么？
- 渴望：在过去这一迭代周期内，如果可以的话，我们期望拥有什么？

实践 2-3 通过反思分析来发现新的不同的实践

通过了解我们喜欢什么和我们学习到了什么帮助我们发现那些在过去迭代周期内已经奏效的实践。那些团队喜欢的实践值得团队延续下去。那些团队新学到的知识通常意味着团队发现了一些更好的实践。有时候团队会认识到一些不奏效的实践，这也是一种学习，因此团队应该停止实施该实践。了解到团队缺少什么和团队渴望什么同样也暗示着团队需要某种实践。所以，在反思会上团队应该一起讨论、分析出那些需要在接下来的迭代周期内实施的实践。

[⊖] 请参见 <http://agilemaniac.com/tag/4ls/>。

2.1.4 前瞻性分析

前瞻性分析帮助团队去猜测和讨论一些潜在的问题、错误或者故障。团队应该在实施任何重大的需求、开展设计和大规模测试之前完成该项活动。前瞻性分析的目标就是希望能够对可能的故障模式及其后果的严重性进行深思熟虑。前瞻性分析包括以下列出来的一些活动：

- 仔细评审需求来降低发生错误的可能性；
- 识别出可能会导致缺陷的设计特征；
- 设计用来检测和隔离故障的系统测试；
- 在整个开发过程中定位、跟踪和管理潜在的风险。

实践 2-4 通过前瞻性分析来预见潜在的问题并解决

风险识别和风险管理是项目经理经常使用的两种前瞻性分析实践。这两种分析技巧同样适用于与诸如需求、设计和测试相关的技术开发话题。参与该活动的人们需要通过他们的想象力和创新技巧来进行前瞻性分析。通过预测问题和思考如何避免问题发生，团队常常会得出一些更好的实践。

2.2 应用程序生命周期管理

在软件开发模型中，传统瀑布模型由几个完全不同且各自独立的阶段组成。如今来看这种开发模型显得太正式、缺少弹性。相比较于定制的硬件设计方案，软件应用本身的特质就决定了它会非常灵活、非常强大。如果团队工程师使用瀑布模型去建造一架飞机，那么结果是可以得到保证的。对于硬件开发而言，因为后期的改动将会带来巨大的代价，因此他们非常需要这种各个周期非常明确、正式、独立的瀑布开发模型。软件本身非常灵活、适应性强，因此，使用瀑布模型就无法发挥软件支持改变的响应能力，同时使软件的首要本质变得逊色。



注意

你需要根据应用程序领域的成熟度和关键度来决定选择灵活的还是严格的生命周期。如果正在开发用于控制核电厂的软件，那么你们应该严格遵循瀑布开发模型。原因在于物理原理并不会发生变化，另外控制软件是一个非常关键的部分。该应用领域有着一套非常严格的物理环境设置和规章制度，因此非常适合应用瀑布开发模型。而社交多媒体网站的需求则总会发生变化，它们随着市场需求的改变而改变，因此它们应该采用非常灵活的开发流程，比如敏捷开发流程。这样我们就可以确保软件开发可以随着社交多媒体局面的变化而变化。

在此我们来列举一个租用个人喷气式飞机从伦敦去往迈阿密的例子。乘客们希望能够拥有一个安全、高效、舒适的旅程。承租公司就会与旅客代表一起讨论租用开销、飞行安排和旅客期望，然后双方就会签订租用合约。之后，飞行员和机组人员就会一起做一些飞行准备，诸如给飞机加油、填写飞行计划、装运旅程中使用的毛毯、饮料和饭食。起飞之后，飞

行员和机组人员会根据空中状况做出一些调整。如果气候比较差，飞机就会往高处飞一些。如果旅客们开始犯困打算睡觉，那么机组人员会把客舱的灯调暗并且关闭广播保持安静。机组人员会尽力提供非常舒适的环境。但有时候仍会遇到个别非常挑剔的乘客。如果遇到某一个有着古怪要求的乘客，那么机组人员就会陷入麻烦之中。有一些非常傲慢的乘客会要求飞机返航，仅仅因为他们想回去取一些忘在豪华轿车上的东西。这还不算最糟糕的。有一些乘客会突然要求飞机改变目的地，飞往里约热内卢。很少租赁公司会去满足这些来自于摇滚巨星、世界巨头或者百万富翁的变化无常的要求。类似，软件程序应该能够快速支持那些合理的、相关的需求变更，而不是那些杂乱无章、异想天开的需求变更。

应用程序生命周期是由一系列贯穿软件终生的互相关联的事件组成。出于下面讨论的目的，我们认为软件程序生命周期包括以下几个阶段：

- 立项阶段：项目愿景、目标和核心需求在这一阶段形成。
- 分析阶段：建立核心功能集和系统的期望行为。
- 架构阶段：创建系统架构和概要设计。
- 开发阶段：分析功能细节、展开详细设计、开始编写代码、部署项目、开始系统测试。
- 运营阶段：软件开始运作、监控系统运行情况、解决暴露的缺陷、继续改进系统。

图 2-1 描绘了以上五个阶段以及之间的先后关系[⊖]。不要认为这五个阶段是相互独立存在的。该图仅仅是一个关于这个五个阶段的概念模型。它告诉我们应用程序是如何一步一步演进的。最初大家有了关于该项目的一些初始想法和愿景，渐渐地大家对该项目的目标形成了共同的理解。主要的功能列表也列了出来。然后团队开始进行设计并开始实施。最终项目成型并投入使用。在每一个阶段团队都需要对前一阶段达成的结论和决定进行调整和修改。但它们不会变得面目全非。立项阶段会为分析阶段打下一个良好的基础。分析阶段又会为架构阶段打下良好基础。开发阶段会交付一个有效的系统给运营阶段。

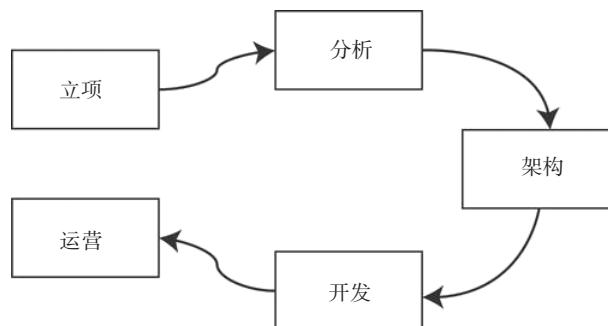


图 2-1 应用程序软件生命周期的各个阶段

那些陷入泥潭的许多项目都在立项阶段、分析阶段和架构阶段就犯下了许多基本错误。他们将许多误解带入到了这三个阶段。因此对于这些项目而言，这三个阶段是非常的重要。

[⊖] 摘自 Grady Booch 的《Object Solutions: Managing the Object-Oriented Project》(Menlo Park, CA: Addison-Wesley, 1996)。

他们应该采取一些新颖的、行之有效的实践来改善整个程序开发过程。我们确实可以在开发阶段改进软件的整体架构。但是如果我们发现整体架构都是错误的或者不充分的，那么后期的改进措施就会显得无能为力。同样，我们也可以在开发阶段支持之前没有预料到的需求变更。但是如果在分析阶段有一个非常关键的核心需求被遗漏掉或者未被重视，那么后期开发阶段的改进也会帮不上忙。

实践 2-5 查找系统及问题和解决方案以改进应用程序生命周期

应用程序开发周期中的各种事件和定义的开发规范涵盖了从变更管理到配置管理，再到发布管理等众多话题。对于众多.NET开发人员而言，最重要的事件发生在日常的开发阶段。因此对于开发人员而言，当他们寻找优秀的.NET实践的时候，他们需要关注那些和开发相关的基本原理和关键原则。应用程序开发周期中的各种事件和定义的开发规范涵盖都和以下七个方面联系在一起：

- 需求
- 设计
- 编码
- 测试
- 部署
- 维护
- 项目管理

如果你考虑一下敏捷开发流程，那么你就会发现敏捷提供的开发方式同样注重改善上述七个领域。它会帮助你改善开发阶段的许多方面，诸如成员合作、适应能力和团队协作，并且贯穿整个应用程序的生命周期。

Visual Studio产品团队意识到了应用程序生命周期管理的重要性。微软正在开发能够支持将团队成员、开发活动和产品运营集成在一起的产品。事实上对应用程序生命周期管理的开发阶段和运营阶段的支持已经放在了微软Visual Studio长期路线图的最重要的位置上。微软正在分配大量的资源来将更好的实践引入到微软的应用程序生命周期Visual Studio vNext。对于.NET开发而言，应用程序生命周期管理同样非常重要。团队和公司组织需要持续地寻找更好的实践来改进它。

2.3 设计模式和开发指南

如今业内有许多拥有卓越经验、明智的辨别力和直觉决断力的.NET开发人员。非常幸运的是，这些开发人员通过写书或者发表博客来和我们分享他们的心得体会。微软公司就有许多这样的人。他们非常乐于将他们的经验记录下来和大家分享。微软开发者网络（MSDN）就拥有很多这样的资源。微软内部有一个称之为模式和实践小组（patterns and practices group）的组织。他们负责向外界宣传一些开发指南、可重用的软件组件和供参考的程序范例。在这其中就有许多讲述.NET实践的详细资料。

这一节的主要目的就是让你知道有这么一个资源的存在。你可以在其中去寻找和调研适用于你的.NET 实践领域。本节会从那些提供开发指南信息的源头开始讲起。然后我们会谈及一些重要的模式。最后我们用一张表格来总结一些特定的工具。

2.3.1 .NET 设计规范

有这么一本你必须要读的书。该书提供了关于各种优秀的和糟糕的.NET 开发实践的大量信息。这本书就是 Krzysztof Cwalina 和 Brad Abrams[⊖]合著的《.NET 设计规范：NET 约定、惯用法和模式（第2版）》[⊖] (Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (2nd Edition))。这本书花费了大量的篇幅来告诉开发人员如何基于微软.NET 框架去设计可重用的类库。其实，每一个想和.NET 框架最佳实践保持一致的开发人员都应该阅读这本书。它对于那些尚不熟悉.NET 框架的开发人员非常有帮助。同样，这本书对于那些不了解.NET 框架设计和构建背后深意的开发人员也值得一读。

实践 2-6 领悟《.NET 设计规范》中关于.NET 框架设计背后的深意

《.NET 设计规范》里面的许多开发指南针对.NET 开发提供了许多建议。这些建议都带有非常明确简单的指示，例如：

- 遵循：绝大部分时候你应该经常遵循这些指南（极少数情况下可以不遵循）。
- 考虑：大部分时候你应该遵循该指南（少部分时候可以不遵循）。
- 绝不：你绝对不应该这么做。
- 避免：你应该避免这么做。

此外，《.NET 设计规范》里面的许多内容都已经在 MSDN 网站上可以得到。你可以在 MSDN 的“适用于开发类库的设计指南”(Design Guidelines for Developing Class Libraries) 主题下找到相关资源。你还可以从 Krzysztof Cwalina 和 Brad Abrams 在 MSDN 上的博客找到其他的信息。

2.3.2 微软的模式和实践小组

微软模式和实践小组通过多种形式给开发社区提供了各种开发指南、开发工具、示范代码和其他众多信息，例如：

- 开发指南：模式和实践小组出版了许多技术书籍。这些书籍提供了许多关于如何通过使用微软工具和技术来更好地设计和开发应用程序的建议。
- 企业知识库：模式和实践小组给开发社区提供了许多可重用的组件和核心功能库。开发人员可以基于这些通用类库来处理那些项目中重要的基础性问题，例如缓存、加密、异常处理、程序日志和验证。

[⊖] Brad Abrams 已离开微软，但他的 MSDN 的博客仍十分有用。

[⊖] 《.NET 设计规范：约定、惯用法和模式（第2版）》的英文版已由人民邮电出版社于 2010 年 1 月 1 日出版。——译者注

- 讲解视频、专题讨论和动手实验室：微软提供了多种途径帮助开发人员去学习各种技术、开发指南和企业知识库。
- 示例代码：企业知识库提供了许多用来演示该类库的正确用法的源代码和示例程序下载。企业知识库（EntLib）包括了非常详细的文档和示范源代码。对于学习如何更好地架构、设计和编码而言，企业知识库毫无疑问是一个非常具有价值的资源。

实践 2-7 利用微软模式和实践小组提供的资源

2.3.3 显示界面层设计模式

不管是开发网站还是开发 Windows 桌面程序，和应用程序显示界面相关的需求通常会更改得很频繁。新的功能也通常和显示界面有关。之所以这样，原因在于应用程序的显示界面是和最终用户产生交互的地方。显示界面就是人机之间唯一的交流接口。

要设计出既支持频繁更改的需求又不对系统其他地方产生不良影响的用户界面，开发人员需要颇费一番工夫。显示界面层的设计目的就是要降低系统不同部分之间的耦合度，例如用户界面的更改不应该对领域模型和业务逻辑产生影响。目前已经有许多非常优秀的 .NET 用户界面设计模式。开发人员花费了大量的时间和精力来打磨这些开发模式。因此对于你而言，你应该开始研究和尝试这些设计模式。你应该在你所在项目的开发过程中尝试这些优秀的 .NET 实践。表 2-2 列出了一些值得一看的显示界面层设计模式。

表 2-2 .NET 显示界面层设计模式

模 式	描 述	更多信息
模型 – 视图 – 表示器 (Model-View-Presenter, MVP)	MVP 模式非常适合用来开发 SharePoint 程序。界面显示的众多职责被分成多份。视图主要负责用户界面。表示器主要负责视图和模型之间的交互。模型主要负责业务逻辑和数据持久化	msdn.microsoft.com/en-us/library/ff649571.aspx
模 型 – 视 图 – 视 图 模 型 (Model-View-ViewModel, MVVM)	在开发 Windows Presentation Foundation 和 Silverlight 程序的时候可以考虑采用 MVVM 模式。视图主要负责用户界面。视图模型主要负责视图和模型之间的数据绑定。模型主要负责业务逻辑和数据持久化	msdn.microsoft.com/en-us/magazine/dd419663.aspx
ASP.NET 模 型 – 视 图 – 控 制 器 (Model-View-Controller, MVC)	MVC 模式将界面显示层的职责分离开来。视图主要负责用户界面，控制器主要负责响应视图动作（用户的操作）。模型则负责业务逻辑和数据持久化的正确	www.asp.net/mvc
Knockout.js	Knockout.js 是一个 JavaScript 库。它使用观察者模式及 MVVM 模式来同步客户端的用户界面和底层数据模型。通过这种方式你可以更容易地使用 JavaScript 和 HTML 来创建在浏览器运行的富客户端界面	knockoutjs.com

2.3.4 对象 – 对象映射

对象 – 对象映射框架主要负责将一个数据对象的数据映射到另外一个数据对象上。比如，为了能够从数据库中获取数据，某一个基于 Windows Communication Service 的服务需要将数据库实体对象映射到数据协议对象上。对象 – 对象映射的一种传统做法就是创建许多数据转换对象。这些对象负责在众多数据对象之间复制数据。对于拥有大量数据对象的程序而言，开发人员需要花费大量的时间精力编写大量的数据转换对象来支持数据对象映射。这一过程非常无聊沉闷。现在业内已经有了许多支持 .NET 开发的数据对象映射框架。你可以采用这些框架从而不需要自己编写那些数据转换对象。这些数据对象映射框架支持以下一些特性：

- 简单属性映射
- 复杂类型映射
- 双向映射
- 隐式和显示映射
- 递归和集合映射

此外还有许多支持对象 – 对象映射的工具。表 2-3 列出了一些值得尝试的对象 – 对象映射工具。

表 2-3 对象 – 对象映射工具

工 具	描 述	参 考链接
AutoMapper	该工具是一种明确直接、基于惯例的对象 – 对象映射框架。你可以定制一些平面映射规则	automapper.org
EmitMapper	该对象 – 对象映射框架将性能放在首要位置。该工具使用了动态代码生成的方式	emitmapper.codeplex.com
ValueInjecter	这是一款非常简单但是很灵活的对象 – 对象映射框架。同时它还支持平面和非平面映射	valueinjecter.codeplex.com

2.3.5 依赖注入

如果你需要将某一对象创建的逻辑挪到使用该对象的类的外面，你应该使用工厂方法模式。类厂封装了创建该对象的复杂逻辑。你的类仅仅需要调用类厂的方法就能获得你需要的对象的引用。有没有可能将工厂模式实现得更简单？有。你需要从你的类所依赖的外部对象类型中提取出新的接口。然后在你的类的构造函数中定义属于这些接口类型的参数。你的类会假设其他类将会将外部依赖对象的实例通过构造函数传递给该类。从这一角度而言，你的类并没有创建任何外部依赖对象的实例。调用你的类的某一其他类将会负责调用创建那些外部依赖对象的实例，然后再传给你的类。这就是依赖注入（Dependency Injection，DI）的概念。依赖注入对于单元测试而言非常有用，因为这种方式将会使你更容易地将某一外部依赖对象的伪造（fake）实现传递给被测试的类。

依赖注入所带来的好处有以下几点：

- 提高了可测试性：我们可以更容易地使用伪造（fake）、存根（stub）和模拟（mock）来

编写单元测试。同时我们还可以验证被测试类和外部依赖对象之间的交互。

- 配置灵活性：我们可以通过配置来给某一类提供外部依赖的不同实现。
- 生命周期管理：依赖对象的生命周期可以集中管理。
- 暴露外部依赖关系：通过构造函数注入，我们可以很清楚了解某一类对外部对象的依赖关系。



注意

抽象接口会在使用依赖注入的时候引入进来。这样可能会造成一些多余的接口定义。对象之间的关系也不如以前那么直接。开发人员可能对于如何初始化对象感到困惑。你应该花一些时间来清晰、妥当地实现依赖注入。

依赖注入有不同类型的实现方式：构造注入、设值注入和接口注入。通常我们会通过控制反转（Inversion of Control, IoC）这一模式来构建那些提供依赖注入功能的工具。控制反转这一概念和依赖注入有着紧密的联系。使用控制反转模式通常会涉及一些常用框架，或者“容器”。提到依赖注入人们则会容易想到这是一种策略。目前已经有了许多可利用于.NET开发的 IoC 容器[⊖]。表 2-4 列出了一小部分值得评估的 IoC 容器。

表 2-4 .NET 中的依赖注入 / 控制反转容器

容 器	描 述	链 接
Autofac	这是一款开源的依赖注入容器。该容器得到广泛使用。它的逻辑非常简洁明了。该容器支持构造注入	code.google.com/p/autofac/
StructureMap	这同样是一款开源的并且得到广泛应用的依赖注入容器。它支持设值注入和构造注入	structuremap.net/structuremap/
Unity Application Block	Unity 是一款轻量级的具有良好扩展性的依赖注入容器。它支持设值注入、构造注入和方法 – 调用注入。Unity 是 EntLib 的一部分。技术支持由微软模式和实践小组负责	unity.codeplex.com

2.4 研究和开发

从 1991 年开始微软研究院就开始了包括软件开发和硬件开发在内的计算机科学基础性研究[⊖]。Kinect 就是出自微软研究院。他们同时还进行着许多其他项目，诸如 CHESS、Code Contracts、Cuzz、Doloto、Gadgeteer、Gargoyle、Pex 和 Moles。你可以从 Microsoft Research 和 DevLabs 网站获得它们的更多信息[⊖]。他们的有些研究主要针对某一特性问题寻找解决方案。他们同时也有一些适用性非常广的工作成果。这些工作成果有些还停留在实验室里。有

[⊖] 详细信息请参见 <http://elegantcode.com/2009/01/07/ioc-libraries-compared/>。

[⊖] 详细信息请参见 <http://research.microsoft.com>。

[⊖] 详细信息请参见 <http://msdn.microsoft.com/en-us/devlabs/>。

一些则已经成为 .NET 框架的一部分。开发人员可以从这些工作成果中得到许多优秀的 .NET 开发实践。

实践 2-8 紧随微软研究院的创新成果

在本节中，我们将注意力集中在两个特定的 .NET 实践领域。这两个领域皆形成于微软研究院长期的研究和开发过程之中。当然除了这两个实践领域之外还有许多其他的 .NET 实践领域。在不久的将来还会有更多的 .NET 实践领域逐渐形成。本节大致讲述了以上提及的两个 .NET 实践领域。此外本节还希望能够展示给你如何通过关注微软研究院研究和开发的成果来发现那些新的 .NET 实践领域。

2.4.1 自动化测试生成工具

可能开发人员需要花费大量时间来编写测试代码。但是这却值得开发人员去这么做，因为测试代码十分重要。本书的第 8 章将会集中讨论单元测试以及如何编写有效的测试代码。开发人员在编写代码的时候面临许多挑战。这些挑战绝大部分来自于许多项目共同面临的令人头疼的一堆问题：

- 项目中存在着大量没有单元测试覆盖的代码。同时开放人员几乎没有时间去编写测试代码。
- 项目中缺少许多测试案例和测试场景。开发人员缺少有效的途径去发现这些测试案例和测试场景。
- 开发人员两极化。部分开发人员热衷编写完善的测试代码。而另外一部分开发人员对此则完全没有兴趣。
- 开发人员没有分配足够的时间去编写那些能够提前捕获问题的测试代码。这些问题在后期被暴露出来。开发人员不得不花费更多的时间去修复它们。

自动化测试生成工具的一个目的就是试图去解决上述这些问题。自动化测试生成工具通过创造一套工具集来完全自动化发现测试场景、制订计划和编写测试代码这一流程。一些测试自动生成工具通过制定测试模板来达到自动化目的。这些工具首先会对源代码或者程序集进行分析检查。然后它会根据事先定义好的模板来生成相对应的测试代码[⊖]。还有一些工具则通过其他途径来生成测试代码。它们的注意力会集中在对已有代码的“探索”上，然后在探索过程中生成相对应的测试代码。

实践 2-9 关注 .NET 实践领域新兴的自动化测试生成工具

微软研究院有一个名为 Pex 的项目。Pex 是参数化探索（Parameterized Exploration）的简写。该项目就是通过对源代码的探索这一途径来生成相对应的测试代码[⊖]。在测试代码生

[⊖] 我曾经使用过由 Kellerman Software 提供的工具 NUnit Test Generator。该工具帮助我生成了上百个有用的测试方法存根。这些存根帮助我编写了许多用来测试遗留代码的单元测试。你可以参考下面链接来获得更多信息：<http://www.kellermansoftware.com/p-30-nunit-test-generator.aspx>。

[⊖] 你可以通过下面链接来获得更多信息：<http://research.microsoft.com/en-us/projects/pex/downloads.aspx>。

成过程中 Pex 会和另外一个叫做 Moles 的工具进行相互操作。出于篇幅的原因，本书不会对 Pex 和 Moles 的细节进行讨论。我们在附录 A 给出了与这两个工具相关的资源链接。在这里我们需要提及的重要一点就是 Pex 和 Moles 通过相互操作提供了以下两点重要的功能：

- 生成自动化测试代码。
- 通过执行探索性测试以发现一些重要的测试用例。通过这些测试用例我们能够发现系统的一些不适当或者异常的行为。

Visual Studio Power Tools 已经包括了 Pex 和 Moles 这两个工具。MSDN 订阅者可以下载 Pex 用于商业项目。Visual Studio 2008 或者 2010 的专业版本或者更高版本开始支持 Pex 工具。你还可以下载相关的示范代码。通过这些示范代码你能够对 Pex 和 Moles 产生的测试代码有一个大致的了解。同时你也能够发现这些工具值不值得你去研究[⊖]。

示范代码：Pex 和 Moles

为了了解需要准备哪些工具才能够运行本书提供的示范代码，你可能需要阅读一下本书的前言部分。

接下来我们会假設本章相关的示范代码存放在 Samples\Ch02 文件夹下面。我们将用 \$ 符号来表示该文件夹。

- 在 Visual Studio 中打开 \$\\2_PexAndMoles 文件夹下面的 Lender.Slos.sln 解决方案文件。
- 在 Lender.Slos.Financial 项目中，类 Calculator 具有两个分别名为 ComputeRatePerPeriod 和 ComputePaymentPerPeriod 的方法。这两个方法主要用来计算贷款利率和贷款支付。
- 测试项目 Tests.Unit.Lender.Slos.Financial 中包括了已经达到 100% 代码覆盖率的测试方法。

我们将在第 8 章中详细讨论单元测试。这里的示范代码主要用来展示 Pex 和 Moles 如何在已有的测试代码中发现新的测试场景。对于许多开发人员而言，100% 代码覆盖率意味着单元测试非常完整。所有的测试情况都已经得以覆盖。然而通过使用 Pex 和 Moles 你会发现上述观点竟然是不对的。

使用 Pex 来生成参数化单元测试是一个非常简单直接的过程。本书不会对此进行介绍。如果你希望学习如何使用 Pex 来生成参数化单元测试，那么可以参考 \$\\1-Start 文件夹下面的文件和 Pex 相关文档[⊖]。有一篇名为《Parameterized Unit Testing with Microsoft Pex》的教程详细地讲述了参数化单元测试和如何使用 Pex 来进行探索性测试。

通过运行 Pex，我们创建了一套参数化单元测试用例。Pex 会创建一个名为 Tests.Pex.Lender.Slos.Financial 的新工程项目。该项目包括了所有 Pex 生成的测试代码。接着，我们在项目的右键菜单中点击 Run Pex Exploration。

结果令我们感到惊讶。Pex Exploration 运行结果包括了三个失败的单元测试用例。你可

[⊖] 第 8 章关注现有的自动化测试实践。本节主要关注那些正在逐渐成熟的自动化测试生成工具。你可能会认为我们应该在第 8 章而不是这里去讨论 Pex 和 Moles。你可以将本节看成对即将到来的工具和技术的一个预览。

[⊖] 你可以在 <http://research.microsoft.com/en-us/projects/pex/documentation.aspx> 发现 Pex 的文档和教程。

以从 Pex Explorer 窗口中看到运行结果。图 2-2 是运行 Pex Exploration 之后的 Pex Explorer 窗口截图。其中一个单元测试用例失败的原因是 ComputePaymentPerPeriod 方法抛出了 DivideByZero 异常。另外两个失败的单元测试用例则是因为该方法抛出了溢出异常。

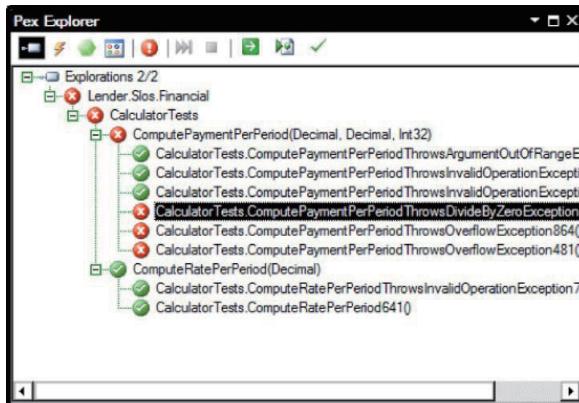


图 2-2 Pex Explorer 的测试结果

这三个单元测试用例之所以没有通过是因为 ComputePaymentPerPeriod 方法抛出了事先没有预料的异常。ComputePaymentPerPeriod 方法代码没有如所期待的那样工作。该方法的代码逻辑其实并不完善。Pex 帮助我们发现了该问题：我们没有验证方法参数 ratePerPeriod 的有效性。

在对 Pex 生成的测试用例分析之后，我们在 Tests.Unit.Lender.Slos.Financial 项目中新加了五个测试用例。具体代码如代码清单 2-1 所示。Pex 测试代码就这样帮助我们发现了已有单元测试中的遗漏测试用例。你可以根据你所在的开发环境来决定需不需要将 Pex 生成的代码作为一个新项目添加到解决方案里。如果你是一名个人开发人员，你可以使用 Pex 和 Moles 来增强你的单元测试。如果你是一名团队领导，你可以通过 Pex 和 Moles 来审核别人编写的测试代码。我们可以将 Pex 生成的测试项目加到解决方案里。然后我们将这些测试用例代码作为永久的自动化测试集提交到代码仓库中。

最终，我们在被测试的方法内部加上了新的 if-then-throw 语句。这样新添加的五个测试用例都可以成功通过。

代码清单 2-1 Pex 和 Moles：发现测试用例

```
[TestCase(7499, 0.0, 113, 72.16)]
[TestCase(7499, 0.0173, 113, 72.16)]
[TestCase(7499, 0.7919, 113, 72.16)]
[TestCase(7499, -0.0173, 113, 72.16)]
[TestCase(7499, -0.7919, 113, 72.16)]
public void ComputePayment_WithInvalidRatePerPeriod_ExpectArgumentOutOfRangeException(
    decimal principal,
    decimal ratePerPeriod,
    int termInPeriods,
    decimal expectedPaymentPerPeriod)
```

```

{
    // Arrange

    // Act
    TestDelegate act = () => Calculator.ComputePaymentPerPeriod(
        principal,
        ratePerPeriod,
        termInPeriods);

    // Assert
    Assert.Throws<ArgumentOutOfRangeException>(act);
}

```

从上述的例子我们确信 Pex 和 Moles 可以帮助我们有效地改善软件质量。通过运行 Pex 和 Moles，我们发现了五个新的测试用例。这几个测试用例都是在参数 ratePerPeriod 被赋予一个无效值的时候发现的。因此，自动化测试生成工具将会非常有效地帮助开发人员开发更完善的测试代码。除了 Pex 和 Moles，微软研究院还有其他几个正在进行的自动化测试生成工具项目[⊖]。

2.4.2 契约式编码

契约式编码是微软研究院的 Spec# 项目的一个副产品[⊖]。现在你可以在 DevLaps 里发现契约式编码。此外 .NET Framework 4.0 也已经包括契约式编码。我们将在第 7 章讨论 .NET Framework 4.0。契约式编码主要和验证代码的正确性有关[⊖]。契约式编码和另外一个称之为契约式设计的概念有关。它们主要关注于下面几点：

- **前提条件**：每一个方法调用者必须保证在该调用该方法之前要能够满足前提条件。
- **后置条件**：方法的具体实现要能够确保当方法调用完成之后能够满足后置条件。
- **对象不变量**：当这些对象不变量都为“真”时，我们认为该对象处于一个有效的状态。System.Diagnostics.Contracts 类提供了些许方法来帮助你在代码中声明前提条件、后置条件和不变量。目前已经有两个工具支持契约式编码：Static Checker 和 Rewriter。这两个工具将 Contracts 类的方法调用转换成编译期和运行时检查。

实践 2-10 尝试通过契约式编码来提高程序的可测试性和软件的可验证性

使用契约式编码最主要的优势就是使软件的可测试性得以改善。一些工具分别在编译期和运行时对代码按照事先定义好的契约进行验证检查。由此可见，契约式编码是一项对开发人员非常有帮助的技术。你应该在你的项目中尽快尝试这项已经存在的 .NET 4.0 新技术。

2.5 微软安全开发生命周期

微软安全开发生命周期（Security Development Lifecycle, SDL）是一项微软正在开发和推

[⊖] 你可以在 <http://research.microsoft.com/en-us/projects/atg/> 网站上发现更多信息。

[⊖] 你可以在 <http://research.microsoft.com/en-us/projects/specsharp> 网站上发现更多信息。

[⊖] 你可以在下面的博客里得到关于契约式编码的详细描述：<http://devjourney.com/blog/code-contracts-part-1-introduction/>。

广的安全程序开发流程项目。他们希望所有的开发人员能够通过该途径来开发安全程序。微软免费提供了和安全开发生命周期相关的开发工具和信息。这里面包括安全开发实践、开发指南和相关技术等有用信息。微软将所有信息分类到以下七个阶段：

- 培训
- 需求
- 设计
- 实现
- 验证
- 发布
- 反馈

微软内部的软件开发经验已经证明了安全开发生命周期的各个技巧可以有效地提高软件的安全性。当他们开始按照安全开发生命周期去开发产品的时候，他们发现和安全相关的缺陷降低了大概 50% ~ 60%[⊖]。

实践 2-11 重视安全应用开发和微软安全开发生命周期

微软安全开发生命周期流程列出了许多可以尝试的各种实践领域。例如，在设计阶段我们需要着重应用威胁模型（Threat Modeling）实践。通过进行威胁模型分析，我们可以及时识别、评估和缓解威胁系统安全的缺陷。我们可以更容易地发现那些潜在的攻击和安全漏洞。威胁模型分析技巧能够更好的帮助你在开发的过程中解决安全问题。

在验证阶段我们则可以尝试称之为模糊测试（Fuzz Testing，简写为 Fuzzing）的实践。模糊测试背后的想法其实非常简单明确：

- 创建畸形的或者无效的输入数据。
- 强迫被测试的程序使用该数据。
- 观察程序是否崩溃、中止、内存泄露或者存在其他奇怪的行为。

Fuzzing 通过试图摸索程序中存在的问题来发现缺陷和安全漏洞[⊖]。如果程序中存在一些潜在的安全问题，攻击者们就会发现这些问题，然后通过类似拒绝访问攻击或者更坏的程序攻击来给你或者最终用户带来影响。因此，你应该尝试 Fuzzing 这一类 .NET 实践来及早发现和修复这些安全问题。

对于开发人员而言，非常重要的一点是你应该知道如何保护你的程序避免受到跨站脚本攻击（Cross Site Scripting，XSS）、SQL 注入（SQL Injection）攻击和其他攻击。表 2-5 列出了微软安全开发生命周期已经提供或正在开发的重要工具。这些工具可以帮助你通过尝试 .NET 实践来开发出更加安全的应用程序。

[⊖] 数据来自于 Michael Howard 编写的报告《A Look Inside the Security Development Lifecycle at Microsoft》。
你可以通过 <http://msdn.microsoft.com/en-us/magazine/cc163705.aspx> 获得该报告。

[⊖] 你可以在 <http://blogs.msdn.com/b/sdl/archive/2010/07/07/writing-fuzzable-code.aspx> 发现更多的细节。

表 2-5 微软安全开发生命周期工具

工 具	描 述	好 处	链 接
威胁模型工具 (Threat Modeling Tool)	该工具帮助开发人员及早地发现、识别、解决和验证那些和安全相关的缺陷	该工具可以帮助开发人员在项目早期设计阶段发现和解决那些安全问题。这是一种“通过设计保障系统安全”的开发态度。该工具基于组件安全和可信任边界来解决安全问题	www.microsoft.com/download/en/details.aspx?displaylang=en&id=2955
MiniFuzz	MiniFuzz 是一款在开发阶段使用的简单文件模糊工具	Fuzzing 能够非常有效地发现代码中的缺陷。它可以帮助发现拒绝服务缺陷和安全漏洞。即便没有接触过安全领域的开发人员也可以便捷地使用该工具	www.microsoft.com/download/en/details.aspx?displaylang=en&id=21769
.NET 代码分析工具 (Code Analysis Tool .NET)	CAT.NET 2.0 定义了一组代码规则。如今 CAT.NET v2.0 已经集成在 VS2010 和 FxCop 中	该工具可以有助于发现软件中潜在的问题和漏洞，例如动态 SQL、跨站脚本攻击、cookies 以及 ViewState 加密	blogs.msdn.com/b/securitytools
微软保护库 (Windows Protection Library, WPL)	WPL 对诸如反跨站脚本攻击 (Anti-XSS) 和 SQL 注入在内的恶意攻击提供了综合保护技术	该库可以帮助 ASP.NET 应用程序避免遭到 XSS 攻击。它定义了一个“众所周知的良好行为”的白名单。这样该库可以在不同语言环境中基于该白名单提供有效保护	wpl.codeplex.com
网站应用程序配置分析器 (Web Application Configuration Analyzer, WACA)	WACA 基于推荐的 .NET 安全实践对网站运行环境进行扫描，以发现潜在问题	该工具可以作为测试的一部分来确认和验证网站应用程序的配置。它会对诸如网站配置、IIS 设置、SQL 服务器和 Windows 权限管理进行检查	www.microsoft.com/download/en/details.aspx?id=573

2.6 小结

在本章你可以学习到从哪里你可以发现更好的 .NET 实践。每一天都会有各式各样的新的实践开发出来或者正在被采用。它们被应用在许多领域，诸如应用在安全开发和微软研究院的项目中。微软 PnP 小组将他们长期实践所形成的专业知识和开发经验整理成了一些开发指南书籍、企业知识库和其他海量的文档和参考程序。

更重要的一点就是，你应该意识到你的团队和公司组织内部本身就藏有一座包含各种 .NET 实践的金矿。你们应该通过举行反思会、分析技术债和建立缺陷跟踪系统来发现那些适合于你们团队和公司的 .NET 实践。

在第 2 章你将会学到对于想方设法收获预期成果的颂歌。我们尝试那些实践的最终目的就是能够获得积极的产出。我们将通过各式各样的方法来衡量这一点。

第3章 实现预期目标

对于每一个项目来说，最终结果毫无疑问是非常重要的。好的结果能促进互信，反之，则会导致互相猜疑。结果是评价自己和自己的团队的依据。考察一下典型项目的情况：一切都以结果为准。在项目中，人员之间的交流方式、使用的硬件和软件、人员编制、规划是否周详，等等诸多因素都会影响最终结果。改善以上因素，会获得更好的结果。不过，改善组织架构是非常具有挑战性的，而且有可能受到团队及其成员之外的影响。本章的主要目的是加深对可控制或可影响的改变的了解，并通过切实可行的技巧来实现更好的结果。

本章主要讲述结果导向的有关问题。作为一名开发人员或团队领导，只有采取切实可行的方法并时刻关注既定目标，才能获得积极成果。首先，要制订计划，这是项目拥有正确起点的保证。多倾听和沟通所期望的结果是什么或应该是什么，这是缔造成功的条件。要实现预期目标，就要努力厘清实际所需，摒弃不切实际的东西，并提供、完成和展示实际成果。同时，要实现预期目标，要最大程度地防止被那些不太重要的事情或可避免的问题转移或分散注意力。

几乎每个项目都不可避免地需要员工协同工作，因而，人际交往能力是能否实现预期目标的重要组成部分。“敏捷宣言”（Agile Manifesto）认识到个体互动的重要性，并把它列为宣言价值观的第一条[⊖]。本章为实现团队成员之间的相互协作并为同一目标而奋斗提出了切实可行的建议。

表3-1总结了本章所涵盖的实践方法。

表3-1 实现预期目标：需冷静待之的有益.NET实践

	实现预期目标
3-1	确保具备成功的要素
3-2	逐项列出所有超出范围的事项
3-3	坚持达成共识
3-4	使用线框图将需求、结构和实现统一为可视化图像

⊖ 敏捷宣言：<http://agilemanifesto.org>。

(续)

	实现预期目标
3-5	图表化、文档化以及可交流的系统架构
3-6	为每一个报表创建模型
3-7	坚决要求细节化示例
3-8	创建一个供其他人学习的原型
3-9	重点在交付，而不是活动
3-10	将结果与目的、衡量标准和关注区域对应起来
3-11	可视化趋势以了解和评估随着时间推移的进度

按语

许多年以前，我参与了一个大型项目，为雇主开发下一代的商业财务软件。项目刚开始就出现了两个重大的意见分歧：第一个分歧是新的应用程序是作为现有系统的一部分，还是完全面向新市场和新客户，作为具有新特性和新功能的产品；第二个分歧是新的应用程序是独立的Windows桌面应用程序，还是基于浏览器的多层应用程序。这两个巨大分歧造成了大量的冲突。最初的决定是创建一个拥有全部新功能的Web应用程序。通过规划和原型设计，发现存在预算不足、开发周期太短以及技术难度太大等问题。经过新一轮的规划和原型设计，发现要交付的产品并不能达到预期效果。在项目中，这些分歧始终无法调解。行政管理人员、副总裁、产品经理、开发人员、分析人员和测试人员对这两个分歧都有自己的看法。任何项目的成功都依赖于团队在基本目标和方向上达成一致，可惜，该项目始终无法达成一致，最终，只能终止项目。不具备成功要素的项目，是不可能实现既定目标的。

在另一个项目中，有一次，我参加了一个需求评估会议。编写文档的分析人员将文档分成几节，并引导与会人员每次只讨论其中一节。这样，文档的每一个细节都会讨论到。在会议的过程中，我发现开发人员和测试人员这两个需要了解需求的人并没有参与讨论。这些关键个体居然与分析人员没有任何交流。于是，我打断会议，问了测试人员一个问题“你认为这些是用户需要的东西吗？”。测试人员斩钉截铁地回答说：“不是”。然后，她开始讲解当前系统是如何工作的，用户是如何使用系统的，系统是如何测试的。不久，开发人员开始在白板上画出流程图，而分析人员则从旁协助，指出新增的功能及如何修改当前流程。很快，这些人就开始真正的交流起来。他们互相攀谈起来，并最终就预期目标和如何实现它们达成共识。

3.1 成功要素

项目如何开始往往决定着最后的成败。如果项目伊始就具备正确要素，那么项目通常会保持这些有利要素。当遇到问题和挑战时，团队也能够聚集在一起并做出调整。当一开始就已经存在错误要素时，就会坏事不断，诸如造成冲突和混乱。不久，已产生的误解会导致更多的误解。冲突仍在继续，预期被遗忘，各种打击接踵而至。

有些项目似乎从一开始就注定了要失败，而项目中每一个人从一开始就意识到了这个问题并坦白地承认这一点。另一种情形是项目在开始的时候貌似不错，但在反思的时候才意识

到该项目迟早也会失败。有时候，当项目不能交付或取消之后，总会有人说该项目本来就毫无希望，而其他人也会附和并认同这一点。这是因为如果项目中每个人都意识到项目从一开始错就错了，失败是迟早的事。如果访问该团队并进行深入剖析，会发现它自始至终缺乏两样东西：意识和认同。他们要么意识不到项目不具备成功的要素，要么意识到了，但是不能认同问题的严重性。团队成员在一言不发地看着项目从错误开始，并在沉默中让它逐步滑入深渊。直到一切都为时已晚或反思时，才意识到失败是唯一可能的结果。

所有项目的团队成员都必须积极参与到项目中。每一名成员都必须努力去发现潜在的问题并在成功要素不足的情况下发出预警。这样做的目的不是去消除问题，而是提高意识，并让大家认同问题、风险、争议和顾虑。在任何给定的项目中，每一位成员都可以协助大家发现隐藏的缺陷，指出过于乐观的地方，并检查合理的规划方法。

实践 3-1 确保具备成功的要素

具备成功要素的项目都很少被打断、干扰和干预，但在项目的开发中顾及打扰问题相当重要。相反，那些工作断断续续，团队成员进进出出，受到外部干预和窥探的项目，都不可避免地会出现交付延误、超出预算或失败等情况。要求个别团队成员去处理所有的任务切换是很难的。由于开发工作的特殊性，开发人员会觉得这样做尤其困难。编程和调试工作需要专心致志以及良好的短期记忆，而大量的任务切换会极大地降低开发人员的效率。开发人员要通过沟通和协调来协同工作，这意味着他们需要大量可用于交流与协同的时间，并逐步加深了解以融洽关系。项目中的每一个人都必须提高认识并努力去避免干扰、分心和中断等破坏项目成功的事情。



警告

有很多项目，都具备雄心勃勃的预期和抽象的成功要素，这也是好事，如 NASA 的阿波罗登月计划。不过，这通常会让项目显得不切实际。将目标设置得小点，可以在一个有限的时间内去实现，这更重要。这其实是敏捷冲刺的概念——先制定一些短期成功要素，并确保它们与整体成功要素保持一致。不断地在微观尺度内进行比较和评估，进而冲刺到宏观尺度上的项目。

在这一点上，成功要素的定义取决于其所在的环境以及先决条件，要想改变甚至去影响它，很难。在这个阶段，出现各种各样的问题不可避免，也无法阻止它们不再发生。这些问题包括很多东西，如团队成员的资质、长期存在的不满和没有培训经费等。我们的目标就是找出应对的策略与方法，以尽量减少影响。即使项目已经开始一段时间了，依然可以在一个领域内进行改进，那就是提高团队成员的交付能力。而这常常通过在职培训和指导等方式来实现。在更广的层面来说，可以在三个领域内加以改进：知识、技能和态度。图 3-1 详细描述了这三个领域。

知识是指对“做什么”的认识与理解。对于团队成员，知识涉及很多方面，如项目目标、技术、合理的计划方法、功能、编码规范、与其他团队合作和许多其他知识。明确、连贯和

周详的交流方式是向团队或在团队内部进行知识交流的有效途径。

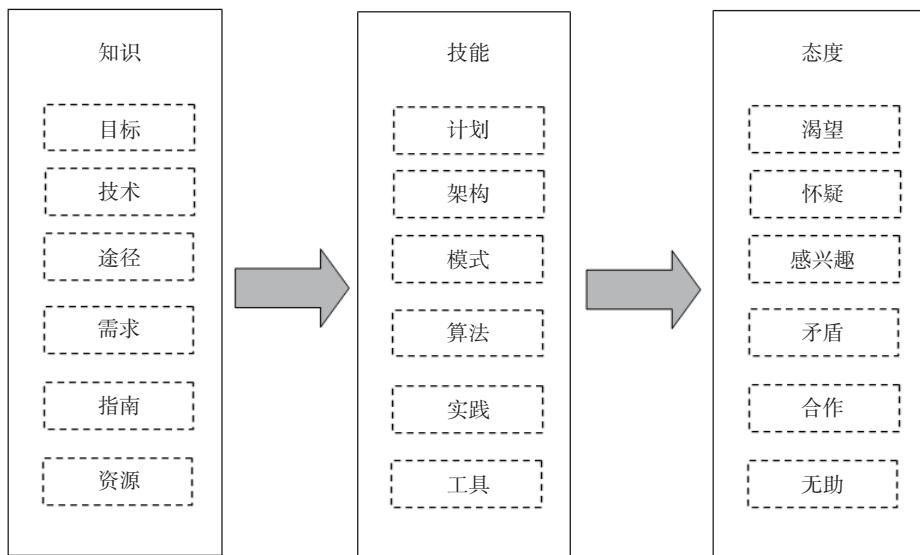


图 3-1 成功要素：知识、技能和态度

技能就是知道怎么做。这与经验、培训和学习能力有关。没有适合的技能，知识也无法转换为生产力。缺乏关键技能的开发人员往往会裹足不前以及缺乏动力。很少有人愿意让人知道自己能力不足，这样也就无法接受别人的帮助从而获得改善。要认清如果没有技能改进，效果就会大打折扣这个事实，然后在团队内部通过培训、自学教程、示范或指导等方式来改进技能。

态度就是要了解为什么要这样做，并愿意这样做。它涉及动机与兴趣。没有正确的态度，即使拥有知识和技能，其结果也是效率低下以及缺乏创造力。开发人员没有正确的态度就会公然不合作或暗中消极工作。图 3-1 中的箭头着重指出了处理的先后顺序。首先，花时间弄清楚是否有遗漏的知识并对其进行讨论。然后，花点时间了解团队技能及其熟练程度，如果存在问题就去解决它。最后，如果确保问题不是源自知识或技能的缺乏，那么就将重点放在态度上，花点时间倾听和诊断问题。一旦完全了解问题所在，问题也就迎刃而解了。

任何项目要取得成功，必然存在成功要素。作为团队的一员，可以自我了解自己的知识、技能和态度。团队领导则需要定期对团队成员开放对话窗口。无论是一对一形式还是小组形式，都可以要求团队成员考虑以下问题：

- 知道做什么吗？
- 知道怎么去做吗？
- 有足够的动力去做吗？如果没有，为什么呢？

这三个问题可应用到项目的各个方面，从需求到技术，再到特定的实践，也可应用于所有目标和优先事项。表 3-2 列出了一些可能的优先事项以及与之对应的知识、技能和态度的

优先事项示例。例如，如果正确性拥有较高的项目优先级，那么就需要团队决定是否采用 NUnit 框架进行自动单元测试来解决这个问题。在示例中，一个团队成员不知道自动化测试的优先级较高，另一个则对单元测试没有任何经验，还有一个则对此是否有利于项目持怀疑态度。

表 3-2 优先事项以及它们与知识、技能和态度的关系

优先事项	知 识	技 能	态 度
正确性	自动化测试	NUnit	怀疑
功能	贷款额度	算法	疑惑
计划表	制订计划	估算	举棋不定
可测试性	MVC 模式	ASP.NET MVC	兴奋
可伸缩性	分析	收益分析	高兴得不知所措

知识可通过彼此之间交流和了解获得提高。例如，在这种情况下，要反复强调单元测试的重要性；要让团队成员了解他们正在做的单元测试的情况；要测量单元测试覆盖率并共享成果。



在此，本书建议大家将自己认为有价值的东西拿出来与大家分享。在这个例子中，重点是让别人知道你很重视自动化测试。这么做的最终目标是让每一个人实现自我监督，避免扼杀过程的“监工”（supervisor）。要提升团队成员的主动性、归属感和成就感。要注重激励而不是过程。

通过参加培训和积累经验可以提高技能。在当前情况下，应该亲自动手编写如何正确编写单元测试的示例。然后对测试代码进行讨论。目的是学习和提高技能。讨论项目需要先决条件，如正规的培训，因而，要在日程表上为指导和在职培训预留时间。

态度比知识和技能复杂得多，但是它可能更重要。在这方面没有什么简单的解决之道。一些对负面因素的忧虑反映了实际项目中的困境，应该加以解决。首要任务是了解团队成员的真正态度，是好还是坏。构建一个了解他们想法的平台很重要。即使没有真正的解决方案，对话也是较好的做法。可能的结果是不愿意但支持的声明：“我不相信自动化测试是正确的做法，但我会支持它，因为我希望看到项目成功。”一个公开支持的怀疑态度比暗中不支持的怀疑态度好多了。

3.1.1 项目启动阶段

在事情刚刚开始的时候，每个项目开始后都有一个启动阶段。当团队成员开始一起工作时，在某些项目，这只是一个非正式的启动阶段。而在一些大项目或有正式流程的项目，则会有项目启动阶段。在这个阶段，团队正在形成，对于项目，每个人都有不同的想法。正如预期的那样，会有差异。有些团队成员对使用新工具和技术感到兴奋。有些则期待带有完整用例和场景的详细需求。有人会对他不是团队领导感到失望和愤怒。有的则担心目标不太

现实，无法在最后期限实现，而且觉得期望值定得实在太高。兴奋、期待、愤怒和担心等，在项目开始之前就充满了各式各样的情绪。

首先，项目启动阶段首要的任务是知识交流。项目负责人之间必须进行磋商和沟通，以确保他们都清楚该项目是怎么一回事。然后这些负责人必须通过沟通为团队达成一致的规定和期望。整个团队必须有一个公开的对话窗口来包含项目的所有议题以及包含以下问题：

- 项目的商业理由是什么？
- 哪些是项目范围之内的？哪些一定不在项目范围之内？
- 决定项目架构的关键功能和目标有哪些？
- 有哪些候选技术和实施方案？
- 已确定的风险有哪些？如何降低这些风险？
- 初步的项目日程和预算如何？

项目启动的时候往往带有大量悬而未决的问题。团队成员不应该想当然。碰到棘手问题就要提出来。每个人都必须对这些问题做出判断，并检验判断是否正确。

在项目初始阶段，让每个人了解项目的整体规划非常重要。每个人都应该知道项目最终交付的是什么。在这阶段里既要共享远景也要达成一致。作为团队成员，必须对自己所发现的任何分歧保持警惕，并积极指出来。

3.1.2 超出范围

或许，确定哪些东西肯定超出了项目范围比确定系统需求更重要。人通常都不愿意去提出不要做什么，想当然地认为这不需要提出，最终导致争执与期望落空。例如，开发人员不应当想当然地认为“全球化”已经超出项目范围。如果认为全球化超出了项目范围，那么就应该公开提出来并坚持让项目经理认可这不是目标之一。

实践 3-2 逐项列出所有超出范围的事项

要花时间列出所有超出项目范围的特性和功能。要确保列表清晰明了和详细全面。这样做的目的是消除想当然的思想，避免不必要的争执。

3.1.3 干扰和分心

时间是一种资源，不管你愿意不愿意，都需要花费。任何项目都可以不在服务器、软件和培训上花钱以节省开支，但时间依然流逝。这也就是为什么转向和分心会导致开发失败，不能实现预期目标的原因。过多的打断和干扰项目会吞食宝贵的开发时间。

限制干扰的方式之一是在自己既不能影响也不能控制的事上设置一个时间上限。根据该标准来衡量是否值得在某个议题上花费时间。例如，如果不能影响决定，但决定很重要，那就把时间和精力花在为决定设置一个最后期限上。要避免在那些自己不能施加影响的事情上花费时间和精力。

能限制干扰和分心的另一种方式是在不占用开发人员时间的前提下促进有效的合作和交

流。可以定期召开非常简短的会议，如敏捷开发青睐的每日站立会。要防止分心、延误和进度停滞，可使用诸如讨论板之类的协作工具，而不是临时打断或不可靠的人传人等手段来发布信息。

广而言之，分心的事对你或你的团队都不是重要的事。对其他人来说，它可能很重要，但他们要优先处理的事并不是你要优先处理的事。花点时间思考一下以下问题：

- 这个问题或打岔的事对我或项目来说重要吗？
- 这个问题或打岔的事，其紧迫性表现在哪里？

当问题出现时，它可能仅仅对某个人来说很重要。产品问题很重要，因为用户会受影响，而解决技术支持问题则没那么重要，客服人员就可以处理。公司主管的担忧很重要，因为他的持续支持对项目来说非常重要，而分析一个受欢迎的但不在项目范围的功能则没有那么重要。别人占用你的时间那是因为对他们来说这事很重要，而你有责任确定这事对你或你的团队来说是否重要。当重要的人提出非重要的问题，有效的策略是将话题从具体问题转移到时间和资源应该花在哪些优先考虑的事情上。用不了多久大家就会了解和认可你优先考虑的事情。

3.1.4 学习与工作之间的平衡

发展新技能可以提高效率。发现一个较好方法可以显著改善工作情况。更好的实践可以带来更好的结果。毫无疑问，必须花时间进行学习以提高效率。为了更有效率，就必须提高生产能力。不过，花太多时间去学习，工作就完成不了了。在学习与工作之间取得平衡意味着学习、工作两不误。考虑以下问题：

- 获得新知识或新思维比运用现有知识更有帮助吗？
- 是希望开发人员提高工作能力、才能和效率，还是希望他们立即投入工作？
- 运用当前技术和实践是否比发展新思维或更好的构思更有用？
- 是否有人认为低效的短期目标比不断改进的长期目标更重要？

实现预期目标就是在提高工作能力与实际工作进展之间取得平衡。没有基本知识，有许多任务都做不了，但有些东西不去做，就学不到。学校和培训课程所教的都是基本知识。像骑自行车、驾驶汽车或组织会议等，都需要体验过才能获得知识。要知道什么时候学习，以及什么时候去应用学到的知识。



进度压力是学习新工具和技术的敌人。如果时间有限就不要考虑那些不熟悉的工具和技术。在工作中学习会让时间压力变得非常难以管理。即使还在学习技术的过程中，且对其只有基本或粗浅的理解，人们都会迫不及待地开始编写生产代码。

3.2 共识

没有共识就不能达成一致。任何理解上的差异都会引起问题。有时可能是实现上的差异：

有人认为控件要放在屏幕左侧，而有人认为要放在右侧。也可能是观点上的差异：有人认为放在左侧更好，而有人则认为放在右侧更好。又可能是优先次序的差异：有人认为放在左侧，用户更容易找到它，而有人则认为放在右侧更明显。在实现、观点和优先次序上达成共识是非常重要的。这样做是为了在工作上达成共识。在较高层面上，误解往往集中在项目目标、成功标准和竞争优势上。而在细节层面上，误解往往与需求、工具、技术、模式、方法、规范和指导有关。

实践 3-3 坚持达成共识

要留意由误解产生的危险信号：目标不明确、成功标准模糊以及竞争优势不确定。每个成员都要不断沟通找出差异所在。使用清晰的图表或图把想象中的实现表示出来，让大家都可以看到它们。至于观点与竞争优势，则由他们当中的代表或相关负责人决定。讨论是一种达成共识的好方式，但最终还是需要一个权威的决定。

3.2.1 线框图

图形这种表达方式往往比文字表现得更好。许多用户都无法描述他们想要什么，但当他们看到或者喜欢上它的时候，就知道这是他们想要的。为什么要花费时间去讨论或描述那些不能形象化的东西？通过在白板上绘制或创建图表等方式协助大家把脑海中的影像勾画出来。

用户界面（UI）原型设计通常是一种用来描述结果是什么样子的有效手段，不过，它往往存在以下两个显著缺点[⊖]：

- 从讨论构想到原型展示之间存在时间差。就算使用最后的 UI 原型工具还是会存在时间滞后，从而制约了交流。
- 原型会让人对完成版本产生错误的印象。在大多数情况下，只有开发人员才知道原型究竟完成到了什么程度或者存在哪些瑕疵。这主要是因为原型的架构设计是匆忙的、捷径式的，并带有错误的理解。这对于进一步的开发并不是好的基础。与概念车或建筑模型不同，大多数用户分辨不出软件原型与预期结果之间的区别。

因此，要避免使用 UI 原型，多使用线框模型。

实践 3-4 使用线框图将需求、结构和实现统一为可视化图像

线框图提供了一个有效的方法来描述模型和进行可视化设计。其可在合作与不留下关于完整版的错误印象之间寻求一种平衡。使用线框图建模工具，大家就可以一起弥补缺陷和调和分歧。图 3-2 显示了一个使用线框图工具 Balsamiq 创建的线框图模型示例。图中的线框图不单展示了要在屏幕上显示的信息，还展示了用户可以在屏幕上做什么。注解说明了具体的

[⊖] 对于概念验证（Proof of Concept, PoC）原型来说，这种说法不太准确。PoC 原型是用来建立和验证技术的可行性和设计的。该原型可以显示出哪里存在缺陷，或者在初期阶段存在哪些困难，从而降低风险。PoC 原型是一种粗糙的、用来快速验证解决方案的策略，因此，必须明确禁止开发人员使用 PoC 原型代码作为生产代码的基础。

规则和限制。这样做就可让大家将注意力集中到达成共识上，还有利于生动地描画屏幕以及描述以下主题：

- 信息显示。
- 提供的功能和操作。
- 主要目的和重点区域。
- 规则、限制和选项。
- 用例和情景。

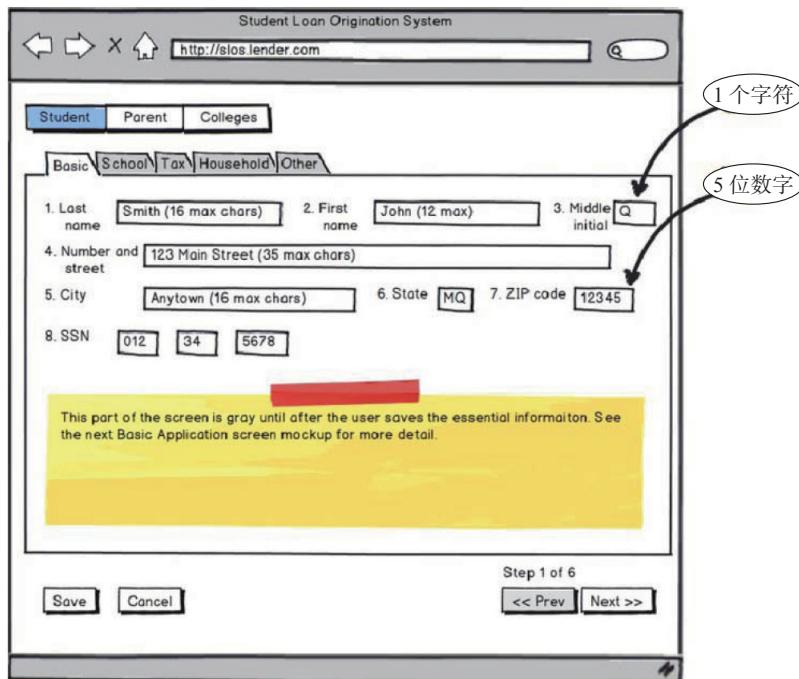


图 3-2 使用 Balsamiq 创建的线框图模型示例

线框图模型所花的时间主要集中在三个方面：信息显示、屏幕内与屏幕之间的导航以及屏幕的布局和可用区域。

- 信息显示是指哪些信息需要显示在屏幕上，以及如何使这些信息易于阅读和理解。
- 导航是指屏幕之间的关系。哪些可以互相切换？哪些是工作流程所需的切换？
- 布局和可用区域是指用户界面设计，这包括图片、链接、按钮、选项区域和其他界面元素。

3.2.2 文档化架构

高层次的系统设计为开发引入了秩序。所谓的良好架构，就是详细设计和开发要与项目利益相关者、开发团队就决定以及方向一致保持良好的沟通。有了良好的架构，详细设计和开发就可以以股东和开发团队达成的一致目标和方向为基准。该基准对于创建框架、

硬件规划、软件组件规划、接口规划和交互规划很重要。架构是就项目解决方案达成共识的一种策略。

实践 3-5 图表化、文档化以及可交流的系统架构

架构必须对系统高层次设计文档化、图表化并可用于交流，它包含以下几个方面：

- ❑ 软件组件以及它们之间的交互。
- ❑ 核心技术和工具集。
- ❑ 驱动需求和优先事项。
- ❑ 技术与非技术上的限制。
- ❑ 重要的设计原则、开发指南以及编码规范。

3.2.3 报表模型

虽然不是每一个系统都有单独的报表，但其中的大多数还是有的。这些报表反映了大部分的重要工作，它们隐含着数据库需求和架构设计。例如，有些需求决定了报表如何显示数据，有些需求要求数据可查询。遗憾的是，在项目生命周期的早期阶段对报表没有给予足够的重视。该实践的目的是就项目的报表需求取得共识，并给开发人员提供足够的细节来交付正确的报表。

报表要让每个人都能理解其直观表示。电子表格是显示报表的极佳方式。它可以显示实数以及它们的总和，还可以显示布局、字体和其他可视元素。更重要的是，分析师可以使用电子表格与客户沟通，并持续更新需求。

实践 3-6 为每一个报表创建模型

使用电子表格，如 Excel，为每一个报表创建一个报表模型。报表要包括显示数据以及正确显示每一个合计和计算。这样做的目的是让报表展示的内容清晰明了。尽量不要简化过程，否则可能会忽略掉一些重要的细节。作为一名开发人员，要在数据库中追溯数据出处，并处理不一致的地方或丢失的数据。

3.2.4 细节化示例

大多数时候，提供给开发人员的示例或过于简单或不完整。对于一些罕见场景或异常情况尤其如此。起初，简单的示例用来协助说明概念，并聚焦主用例上。然而，软件系统必须妥善处理更复杂的使用场景。随着需求从高层次设计推进到细节设计，坚持细节化示例就很重要了。对细节化的努力付出要成为深层次的共识。

复杂的算法和业务规则需要细节化的示例。要编写代码来实现这些算法和商业规则，开发人员需要理解细节并验证代码是否满足所有需求。简单的示例的代码也简单。大量完整和细节化的示例有助于高效的开发人员进行单元测试并正确实现代码。细节化的示例还有助于调试和故障诊断。充分体现意图的代码利于更好地理解示例。

实践 3-7 坚决要求细节化示例

在整个项目的开发周期内，从需求阶段到测试阶段要做到：要不断要求分析人员和测试人员提供细节化的示例来演示系统是如何工作的；要求用示例数据来填充线框图；要求报表模型显示正确的数据；要求使用示例和插图来阐述业务规则；要求数据能再现故障和错误。要求这些信息对正确开发、诊断和解决许多重大项目问题来说非常重要。

3.2.5 创建原型

创建原型是指创建一个示例以便其他开发人员模仿和学习。原型就是一个使用专门方法编写的样品，它应该能被其他人模仿。这样，就可以让开发人员认可这种学习方式。如果有一个清晰明了的原型，那么开发人员的工作就可以通过模仿提供的良好的系统模型开始。这样做的目的就是创建让其他开发人员学习而且必须学习的模型。原型还包含以下好处：

- 验证架构。
- 使抽象概念具体化。
- 通过其他开发人员的回应以改进设计。

除了作为一种标准之外，原型还有助于确认和验证架构。高层次的设计是从白板或设计师的脑海中开始的。这些想法和方法都需要验证。构建原型时要使其功能涵盖架构的所有层面。

大部分开发人员是通过示例来学习的。学习新的架构意味着既要学习用它创建点什么，还要学习如何引用现有的模型。一个好的原型就是一个教材，通过一个中等大小且全面的应用程序来指导开发人员。当 ASP.NET MVC 开始引入使用的时候，学习其概念的最好方式就是创建 NerdDinner 应用程序[⊖]。NerdDinner 应用程序就是一个原型，用来学习“做什么”和“怎么做”的。

一个好的原型要比许多解释所表达的信息清晰得多。开发人员都可做出回应以改进原型。他们的建议往往是非常有益的。许多开发人员都习惯“有本事就把你的代码给我看看”(show me the code)，并对解释和辩论感到不耐烦。他们希望看到架构的一个工作示例和开发风格。事实上，原型可有效地解决进退两难的知识、技能和态度的优先次序问题。对于态度端正的开发人员，原型正是他们所需要的、用来转化为生产力的东西。对于对项目有疑虑的开发人员，原型提供了一种讨论具体问题的途径。

实践 3-8 创建一个供其他人学习的原型

创建一个有足够细节的原型就可作为其他开发人员学习的模型。有时候它也被称为“垂直切片”[⊖](vertical slice)，原型就是一个从头到尾演示系统是如何工作的示例。它没必要作为一个独立的应用程序，实际上，它只是在重要的功能方面运行得很好。它的重点在于让其他开发人员模仿代码风格、设计模式和方法。总的来说，原型就是希望其他人学习的模型。

[⊖] 请参阅：<http://www.asp.net/mvc/tutorials/introducing-the-nerddinner-tutorial>。

[⊖] 详情请参阅：http://en.wikipedia.org/wiki/Vertical_slice。——译者注

3.3 预期目标

项目预期目标的重点在于交付软件。尽管已经取得许多实现软件的实际成果，但是软件不能工作，许多项目还是会判为失败。总体而言，能工作的软件就是首要的交付成果。次要交付的，如文档，也很重要，但不应以牺牲首要交付为代价。

在软件项目中，各项活动都不是目标。所有活动都必须以获得成果为目标。日复一日，程序员都在讨论如何编写代码，可惜，最终评估是否合格是由交付的特性和功能的质量决定的。项目利益相关者开发软件是为了获得实际成果，期望所有的活动能带来实际成果。

3.3.1 交付

软件要交付的产品可能是有形的，也可能是无形的。它可能是一个安装包，也可能是一个保存贷款申请的（应用程序）。对于安装包来说，就是有形的文件，可以在网络上下载，或通过其他方式获取。对于贷款申请，用户可坐在计算机前进行填写，如果输入没有错误，系统就可以让用户保存申请，因而，实现“保存申请”就是能让用户无形中完成工作的应用程序。

归根结底，预期目标与交付的好坏有着紧密的联系。使用阶段性目标来衡量进度是很重要的。实际评估用的交付成果既可以是已完成的也可以是未完成的。

实践 3-9 重点在交付，而不是活动

重点在于交付的成果应该是以结果为导向的，可在目标条款中进行说明，如完整性、正确性和一致性的标准是什么。用户的重点是看到并体验产品，而开发人员或团队领导，重点在交付而不是活动。当然，活动也很重要，但仅限于生产优质的交付成果。

3.3.2 实际成果

评判预期目标的一种方式就是将目光聚焦于满意度和质量上。当交付期待和预期的特性和功能的时候，客户和利益相关者会感到满意。完美质量是指能满足所有需求，而且系统能为每一种实际用途提供服务。想象一下一个新的购车者情形：在购物初期，他会着重于汽车的造型、装饰以及各种功能。尽管购车者要关心的东西可以列出一个长长的愿望单，但汽车不可或缺的用途是提供代步功能。这是理所当然的事。在同等结构质量的前提下，毫无疑问，客户会期望汽车有好的动力且可靠性高。在购物时，许多购车者重点都在他们想要什么而不是他们期望什么。然而，在购买汽车一年后，客户就能切身体会到汽车是好还是不好。经过几年的保养花费以及认识到可靠性相关的问题后，结构质量就显得很重要了。

要实现长期成功就必须长期维持较高的满意度。对于多数项目来说，其重点永远是特性和功能。在项目的初期阶段，用户和分析人员会专注于愿望单和当前系统不能做什么。最低限度的需求和必要的功能是理所当然的。随着项目的进行，质量保证团队会专注于系统特性和功能的测试，即“适用性”测试。

系统最终用户和客户可以根据软件已实现的功能和没有实现的功能来评估满意度。测试

团队则可根据需求来衡量系统质量。只要与最小需求、必要功能基本一致，并提供了愿望单中的功能，就会获得好评。不幸的是，出来的成果多半是一个结构不健壮的系统。非功能性需求得不到满足，很快就让初期的愉快变成不满。性能缓慢、错误多、可扩展性问题和其他非功能性缺陷会破坏信任关系。

图 3-3 将满意度和质量划分成了四个象限。象限 1 表示预期内特性和性能，这确定了最基本的质量。象限 2 的结果是使用可靠和高性能代码建立的健壮系统。象限 3 的结果是出乎意料地令人满意。象限 4 的系统技术优雅，代码很棒，设计非凡。

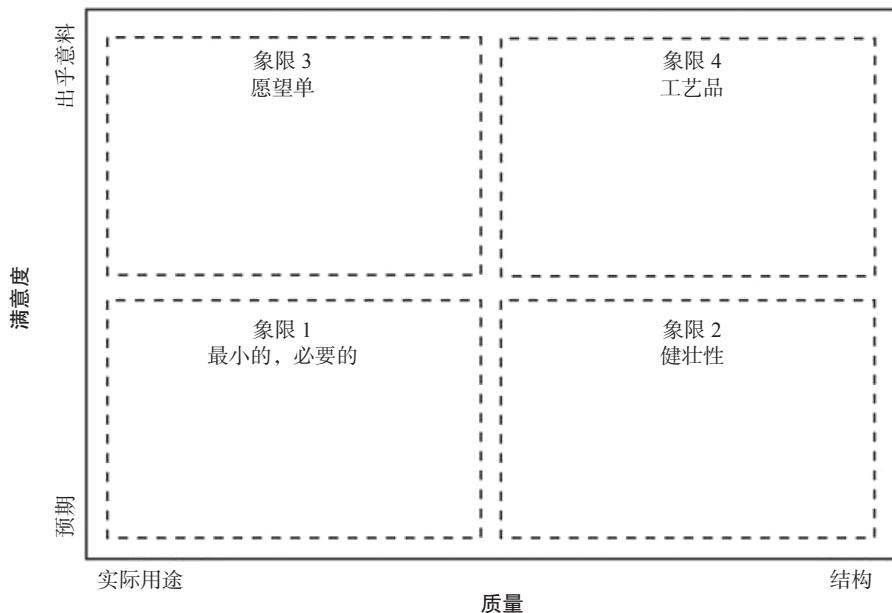


图 3-3 满意度和质量的象限

大多数项目的需求过于集中在象限 3 的结果——愿望单。随着项目的推进，通过测试评估后得出的结论则集中在象限 1 的结果。对于大多数项目来说只需关注这两个区域。在验收测试或部署之后，将经受象限 2 的结果考验。如果结果不理想，有关性能、可靠性和可扩展性的投诉就会如洪水般涌来。一度看起来不错的系统开始显得难堪。对于象限 4，往往会因系统内部的问题而无法实现结果，而且面条式代码[⊖] (spaghetti code) 和设计限制的影响确实存在，并反映在维护和升级成本上。

从长远来看，要保持能实现实际的成果，就需要对项目的关注区域的逻辑顺序进行划

[⊖] 面条式代码 (spaghetti code) 是软件工程中反面模式的一种，是指一个代码的控制结构复杂、混乱而难以理解，尤其是用了很多 GOTO、例外、线程或其他无组织的分歧架构。其命名的原因是因为程序的流向就像一盘面条一样扭曲、纠结。面条式代码的产生原因有许多，例如没有经验的程序员及一个经过长期频繁修改的复杂程序。结构化程序设计可避免面条式代码的出现。具体请参阅 <http://zh.wikipedia.org/wiki/%E9%9D%A2%E6%9D%A1%E5%BC%8F%E4%BB%A3%E7%A0%81>。——译者注

分。象限1和2是项目的基础，在这两个区域做大量的工作是很有必要的，尤其是在需求注重结构质量以及实施结构测试的时候。不过，整个系统都不能忽略4个象限中任何一个象限。虽然提供象限3或象限4的产品可以获得高满意度，但绝对不能放弃象限1和象限2。以下是每一象限的详细说明：

- 最小的，必要的：基本的特性和功能是项目首要目标。该象限的工作结果就是发布“正确的产品”(the right things)。该区域的发布目标就是既满足预期需求又要确保系统做它需要做的一切事情。这是意料之中的事。系统的功能就是为实际用途提供服务。
- 健壮性：系统的健壮性是项目的次要目标。该象限的工作结果是发布“以正确方式工作”的产品。该区域的发布目标是结构稳健、性能优异、可靠和值得信赖。这些都是预期的非功能性需求。
- 愿望单：渴望或期盼得到的东西就是项目的第3个目标。该象限的工作结果是发布符合“理想条款”(the desirable items)的产品。该区域的发布目标是添加用户或客户希望增加的功能。这些都是有用的特性和功能。
- 工艺品：优质是项目的第4个目标。该象限的工作结果是发布具有“丰富经验和专业知识”(superior experience and expertise)的产品。该区域的交付目标是技术、远见、领先和长久的考验。这将提供令人惊讶的品质。

要将这4个象限牢记于心，才能整理出目标、指标和预期值。由于上述各个方面都重要，因此要逐一确立它们的目标所在。将所有目标和它们与目标区域的关系都列出来，想办法监控或评估它们，或者以其他方式去衡量它们。要确保把注意力从过多关注的象限转移到对其他象限的关注上。短期的压力会推动长期价值。要记住，客户或用户今天关注的价值所在并不是他们的长远价值所在。

实践 3-10 将结果与目标、衡量指标和关注区域对应起来

该做法既适用于个人，也适用于团队。要寻找一种方式来对应结果和目标，例如，首要目标是确保功能支持用户完成他们的工作。要寻找一种方式将度量指标或测量指标与结果对应起来，例如，项目的最小需求和建议的《.NET设计规范》都要学习。要寻找一种方式将结果与关注区域对应起来，例如，现今已经完成的结果需要进行耐用测试以确保系统扩展正确。积极成果不是即时成果所能比的，它们可以在短期和长期内实现优质和超越期望。

3.3.3 趋势

预期目标不能仅在单一时间点内预览。随着时间推移的趋势也很重要。趋势图可以显示出改善率。人们很难观察到不同时间的事件之间的联系。趋势图可以显示出好的实践如何一步一步地去实现预期目标。相反，它也可以显示在缺少重要实践的情况下，状况是如何逐步恶化的。因此，随着时间的推移，检查跟踪目标很重要。了解当前状态与之前的有什么关联也很重要。

趋势有助于鉴别哪些实践行得通，哪些实践行不通。趋势可鉴别出系统性问题、转折点

和主要指标。例如，如果自动化测试的趋势图意外呈现下降趋势，说明有些重要指标已经发生改变。问题可能并不严重，但也很重要并值得去做调查。如果危及了项目，这种趋势改变就是一个警报。敏捷开发中的燃尽图（burn down chart）就是重要趋势技术的一个示例^Θ。燃尽图会显示出整个开发工作是如何沿着冲刺过程完成的，并不断将冲刺计划和期望与实际成果做比较。

实践 3-11 可视化趋势以了解和评估随着时间推移的进度

要花时间确保定期收集结果数据。使用有效的图表和图形来可视化趋势。目的是了解情况是如何随着时间推移一步一步发生改变的。这对在自己或别人的脑中树立进度评估、突发事件和改进的必要性的观念很重要。

3.4 小结

本章学习了如何实现预期目标。3.1 节讨论了成功要素以及如何让项目参与者积极地参与到项目中来。本章还讲述了在知识、技能和态度三方面提高认识和承认差距的重要性。

本章还学习了达成共识的重要性，还讲述了几种达成共识的技巧，并描述了几种可能存在理解差异的情况。这些技术包括线框图、文档化架构、创建报表模型、坚持细节化示例和建立原型。

本章还学习了重要的是交付，而不是活动。需要特别注意满意度和质量在短期和长期这两方面的实际成果。在最后的分析中，将预期目标实现的测量与目标、关键度量指标和衡量指标以及整体价值对应。

^Θ 更详细信息请查阅 http://en.wikipedia.org/wiki/Burn_down_chart。

第 4 章

量化价值

采用更好的实践可以为开发过程增值。众所周知，遵循合理的原则、好的实践、既定方针和公认的成功之道，就可提高开发效率。同样，避免错误观念、坏的实践、错误的技术和失败之道，也可以提高开发效率。可惜的是，常识并不是常用的实践。要采用新的不同的实践，就要改变当前的习惯和做法。而提供改变的动力很重要，通常需要我们将好的实践进行量化。

本章将介绍不同利益相关者的观点以及这些观点会如何影响改变实践的预期收益。每个人、每个群体对价值的理解都不同，本章的主要目的就是去学习和了解这些价值观。之后，将讨论量化价值的数据的来源，这包括定量与定性的数据、业界证据和证明。量化价值的重点是将数据与相关人员及其价值观联系起来。

对于开发人员自身而言，将采用的新的或不同的实践的价值进行量化的重点在于证明或说明改变是必需的。例如，如果要采用单元测试，那么就较低的缺陷数的价值进行量化就可加强对单元测试所带来的收益的认识。对于团队领导来说，决定是否采用更好的实践，要看实现改变所花费的时间能否带来显著的价值。

尽管本章是从一个局外人的角度来写的，但其目的并不止于提供参考。当然了，要局外人与你感同身受，那很难。在实践中，来自于项目内部或组织内部的改变更易于接受，也更持久。局内人有助于打破情感障碍，进而从内部实现和维持改变。因而，希望作为局内人的你或你的团队成员，能把本章内容与项目联系起来，并给项目带来积极影响。本章还将讲述如何激发起大家的热情以便获得支持。本章还介绍了量化采用更好开发实践价值的具体做法。表 4-1 总结了本章所涵盖的实践。

表 4-1 实现预期目标：零冷静待之的有益 .NET 实践

	量化价值
4-1	学习和了解不同利益相关者的价值观
4-2	确定财务措施并开始跟踪它们
4-3	将改进后的实践与提高后的可控性联系起来
4-4	借助清晰的说明实现更好的实践，交付更好的产品

(续)

	量化价值
4-5	使用常识来调整可改善的常用实践
4-6	收集和跟踪指标及数据
4-7	收集和跟踪描述信息和观察结果
4-8	收集和分享成功案例
4-9	从客户和专家那里获取推荐做法

按语

在我倡导最佳实践之初，常把话说过头了，如单元测试将带来更高质量。我没有意识到，我的本意是希望减少 20% 的缺陷，但项目经理听来却是可以实现零缺陷。随着时间的推移，大家都认为我有点言过其实了。期望相同，但为减少 20% 的缺陷却要增加了 30% 的开发时间是不是有点得不偿失呢？现在，我的主张温和多了：“让我们尝试一下单元测试，或许有所改善。”我现在尽量避免把话说过头，如“更高的质量”、“更加合理”、“更易于维护”以及“快速生成”，等等。我希望做到“轻承诺，高回报”。在整个改变过程中要采用增量和迭代方式来实现。即使改善有限，但结果是具体的，实实在在的。过头话不仅抽象、不切实际，还会让每个人都期待不切实际的结果。

实行新的实践，尽管是更好的做法，也需要从当前状况转换到新的状况。要引入变化是不容易的。每一个人都需要时间考虑新的实践带来的影响。在改变之前，人们需要通过学习来认同改变是好的。在他们承诺改变之前，团队需要讨论变化以及进行切身感受。在试行期间要寻求共识，不能强迫他们接受改变。最重要的一点是，要求量化价值往往成为实际的关注点。在许下承诺之前，许多人需要的只是参与感。

最近，我问一个团队领导，他们团队是否正在采用某个最佳实践。基本上，他们的回答都是：“我们不采用那个实践。我们认为花时间去落实这个，不会有显著的回报。”从某种意义上来说，有这样的感觉很正常。如果他不相信花费时间去采用最佳实践可以获得显著回报，那么如何去说服他呢？一方面要说服他实际获得的价值比他所预计得要显著得多。另一方面要说服他实际投入的时间会比他所预计得要少。如果本身就是团队领导或项目经理，那么还可以通过修改团队的预期目标来实现。某些目标，如每天的 NCover 报表，只有通过新的实践才能取得。

大多数人所期待的是数字和事实。可惜的是，很难通过数据来说明采用最佳实践所获得的回报。如果能够根据确切信息来说明诸如采用特定的实践，将获得 23% 的投资回报率这类问题，那就太好了。很遗憾，这些数字难以找到，且往往缺乏说服力。笔者认为这与实现新的不同的实践需要采用增量和迭代的方式有关。持续的改进将是一条不断上升的学习曲线，并可获得长期效益。对于不情愿、持怀疑态度或者悲观的人，往往缺乏有力的数据来转变他们的态度。所谓“事实胜于雄辩”，因而，最重要的是在自己能控制和能影响的范围内开始积极地做出改变。接下来，量化这些变化的价值，这样，就可以从中发现这些改变是如何结出硕果的，并展示给其他人。

4.1 价值

无论是管理人员、开发人员、最终用户，还是客户，每个人对“价值”的理解都不同。如果一个管理人员想对最佳实践进行价值量化，那么价值就可能是管理人员预估计的财务回报。管理人员正在寻找一种以货币形式表现的成本–效益分析。他们也在寻找更好的方法来管理和控制开发项目，而且不会危及他们已经取得的成果。

对于大多数开发人员来说，花在落实实践上的时间和精力就是开发成本。开发人员想知道这样做有什么好处，是否会让他的工作更快、更容易或者更好。客户和最终用户则通常认为更多的功能和更好的质量才是好的。他们担心这样做会增加成本或延迟交付时间。他们需要你明白，如果不能在当前预算和进度限制内交付产品，所谓新的、不同的实践就不是最佳实践。你需要花时间去学习和了解不同利益相关者所持有的观点以及他们所关心的财务风险。

实践 4-1 学习和了解不同利益相关者的价值观

许多开发项目都会有不同的利益相关者，每个人所看重的东西都不同。要重点关注以下几个方面：

- 财务回报
- 提高可控性
- 提高质量品质
- 更加高效

4.1.1 财务回报

通常来说，执行董事和高级管理人员感兴趣的是财务回报。持续的、不断完善的过程中所潜在的益处是讨论的重点。这些都来自于长期的投资。以下内容是这些管理人员希望听到的财务回报的下限和上限：

- 更低的开发成本
- 维护、支持和升级更便宜
- 额外的产品和服务
- 吸引和留住客户
- 新市场和新机遇

事物都有两面性，这里的另一面就是财务风险。这也是管理人员常常用来反对改变的根源。盲目臆测的做法会让生产力和效率遭遇风险。不要忽视财务风险，要正视它。要辨别出潜在的风险并解释如何去减少。在可能的情况下，尽量提倡一些无成本或者低成本，带有少量潜在风险或者能带来丰厚回报的新的不同的实践。

每个组织早已有跟踪财务的措施了。这些信息不会提供给你，但可通过间接措施获得这些信息。项目经理或会计部门会跟踪每一个人在项目中的总时数。测量每人时的工作成果就是一种间接测量成本的方法。例如，尝试去带来发现、报告、修复、跟踪和验证一个典型

错误的总人时。一个典型缺陷所带来的工作总量会让你大吃一惊的。作为一个独立的开发人员，更少的缺陷意味着可为公司节省更多的钱。作为团队领导，还要把数字乘上团队的成员数。减少 20% 的错误是否值得增加 30% 的开发时间呢？这就是我们可以接触到的一个引人注目的财务参数。

实践 4-2 确定财务措施并开始跟踪它们

对于一个试图增长的小型商业软件开发公司，会有许多经济利益方面的东西让高级经理人员感兴趣。表 4-2 列出了一些潜在的经济利益以及相关的直接或间接措施。从表格最右边那列开始，选择与你的组织相关的间接措施并开始测量它们。新的和更好的实践应该能提高这些指标。采用更好的实践一段时间后，直接措施也可能得到提高。可通过询问你的组织正在跟踪的直接措施来验证这些假设。如果有结果显示出来，就可以把相关点连接起来，虽然不一定要这样做。

表 4-2 潜在经济利益以及相关的直接或间接措施

潜在利益	直接措施	间接措施
更低的开发成本	工资、保留率、承包商发票	努力的结果、缺陷数量、初期支持电话、士气
更低成本的部署系统	硬件成本、软件授权	性能、可扩展性
总体拥有成本 (TCO)	整个开发周期的成本	维护工作、支持工作、升级工作
收益	销售增长、更新率、竞争优势	最终用户的满意度、正面评价、新功能
新客户	销售给新客户、市场反应统计	更好的功能、易于使用、提高精确度
新市场	新产品的供应、销售给新的国家和个人消费者	交付产品、更好的设计、全球化

4.1.2 提高可控性

管理人员也是在压力下工作的。这意味着他需要通过别人的努力成果来完成自己的责任。如果改变降低了团队的表现，那么管理人员要对此负责。这种压力会让管理和控制的重点集中在维持现状上。具有讽刺意味的是，许多新的和改进的实践只是可以帮助管理人员更好地管理和控制项目。

项目经理往往是与开发人员的活动联系得最紧密的管理人员。部门经理，如开发总监，也要关注开发人员的日常工作。对于这些管理人员，重要的价值都来自于一般的管理原则，他们需要在以下方面寻求改善：

- 可见度和报告
- 控制和校正
- 效率和速度
- 计划和预测
- 客户满意度

实践 4-3 将改进后的实践与提高后的可控性联系起来

表 4-3 列出了新的开发方法与期望改进的项目可控性之间的联系。该表列出的实践以及所提供的可控性并不完整，但作为开始已经足够了。最重要一点是量化价值，对许多管理人员来说就是将开发方法与他们所看重的原则联系起来。如果实践方法能带给他们不曾拥有的东西时，尤其是带来管理和控制项目的能力时候，这样做就十分正确了。

表 4-3 开发方法与提高可控性

开发方法	提高可控性
持续整合	可视度和报告、控制和校正
自动化测试	控制和校正、效率和速度
自动化部署	关注度和报告、控制和校正、效率和速度
建立模型	计划和预测、客户满意度
代码分析	控制和校正、可视度和报告

4.1.3 提高质量品质

最终用户和客户感兴趣的是交付时间。当量化价值的时候，他们想知道更好的实践如何为他们生产出更好的产品。为了向最终用户和客户明确描述所能获得的好处，开始就要注意以下几项他们看重的内容：

- 更多功能
- 易于使用
- 更少缺陷
- 性能更快
- 更好的支持能力

要使团队生产出更好的产品，采用信息沟通就是更好的实践。把常识性的信息做得足够详细，能明确表述出客户的满意度和质量要求，这是团队的目标。

实践 4-4 借助清晰的说明实现更好的实践，交付更好的产品

表 4-4 列出了基本的质量品质类别。通常会把最左边这列称为 FURPS：功能、可用性、可靠性、性能和可支持性。当然，还有很多，如本地化功能（可以根据需要扩展该分类列表）。第二列是与每个 FURPS 分类相关的具体组成部分的例子。最右边那列是与每一个 FURPS 分类相关的开发方法。只有在表格中列出具体细节（见表 4-4），才能让最终用户和客户清楚知道更好的开发实践是怎样支付更好的产品的。

表 4-4 与基本质量品质相关的开发方法

FURPS	具体组成部分	开发方法
功能	当前功能集、新特性、安全性、新性能	线框模型、报表模型、持续集成、单元测试、安全测试

(续)

FURPS	具体组成部分	开发方法
可用性	外观、一致性、易用性、工具提示	线框模型、报表模型
可靠性	正确性、出错频率、故障时间	代码分析、单元测试、稳定性测试、自动部署
性能	速度、响应时间、吞吐量	性能测试
可支持性	较少的扩展功能工作 / 时间、更短的错误修复时间	代码分析、单元测试、稳定性测试

4.1.4 更加高效

开发人员和团队领导一般对个人或团队的效率比较感兴趣。对于开发人员来说，量化更好实践的价值的重点是提高效率，那会容易很多。常见的论点是：团队只要采用更好的实践，就会变得更高效。同样，只要能避免不好的实践，也可以变得更高效。开发人员一直在寻找使运作更顺畅的方法。以下列表是开发人员希望听到的好处：

- 个人生产力
- 减压
- 更大的信任
- 更少会议和干扰
- 更少的冲突和混乱

实践 4-5 使用常识来调整可改善的常用实践

常识当然不是全部的理由，还有其他理由。转而采用新的或不同的实践，会涉及习惯的改变。而这点又体现在知识、技能和态度上。例如，若编写单元测试并不是已有习惯，那么要养成这种习惯就需要一定的时间和耐心，而耐心是尤其重要的。此外，开发人员还需要花时间去练习和扩展技能。开发人员在他们感受到好处之前，需要经历改变。要量化新的、不同的实践的价值，就要将常识与学习经历结合在一起。

技术债（参阅第 2 章）是一个很形象的比喻，非常有道理。^Θ技术债背后的原则是，任何开发决定都隐含或隐藏有超出短期利益的长期消极影响。短期的便利会“挤掉”长期的优势。捷径会导致长期的损害。在未来总有一天要一次性付清的。积累下来的技术债所造成的后果可能会非常严重。技术债往往会在不经意间出现。它通常是由于缺乏认识、不愿意承认缺陷或低估了潜在的影响而造成的。为了做到更高效，开发团队必须明智的积累技术债。表 4-5 展示了图表化积累的技术债的方式。每种开发实践都是经过逐个审慎考虑的。表 4-5 列出了 5 种开发方法。

表 4-5 技术债务积累图表

开发方法	短期回报	长期风险	必然性
持续集成	没有持续集成服务器、无须费时集成	更长的集成期、要费时解决集成问题	这是一定要偿还的债务

Θ 参阅：<http://www.martinfowler.com/bliki/TechnicalDebt.html>。

(续)

开发方法	短期回报	长期风险	必然性
线框模型	更少的对话、简单的会议	预期落空、指责	这是一定要偿还的债务
单元测试	无须费时编写测试代码、更明显的生产力	粗心大意的错误、集成问题、还原功能	这是一定要偿还的债务
代码分析	无须费时进行分析	违反标准、可维护性差、没有问题预防机制	长期债务
自动部署	无须在自动化方面费时	配置困难、错误不断、部署不可预测	这是一定要偿还的债务

先以第一种方法持续集成为例。开发团队为了短期利益而没有考虑持续集成。既然没有持续集成服务器，也就不需要浪费时间去安装和配置持续集成服务器了。开发人员由于延迟或忽略任何潜在的集成问题而节省了时间，甚至没有花时间去考虑集成问题。然而，由于没有采用持续集成的实践，从而留下了长期风险。之后可能需要更长的整合期。团队可能要花时间来解决问题或重新设计组件。花大量的时间和精力既是不可避免的，也是无法预测的。最后，整合问题必须处理，所有的集成问题也必须去解决。显然，持续整合在预防出现问题、更好的设计和更佳的可预测性等方面有着显著的价值。

4.2 数据来源

进行量化需要数据。本节将讲述怎样去识别和跟踪数据。现在先来关注一下以下3种数据源：

- 确切的数据
- 观察到的
- 业界说法和推荐做法

数据采集后，就需要进行分析和了解。最终目的是找出隐藏在数据背后的含义。还要考虑以下问题：

- 相关性：数据是与一个还是多个开发实践有关？
- 关联度：如果实践改变了，数据是否也会以某种相关联的方式改变？
- 突出：实践改变是否一定会改变数据？

不仅要对数据进行分析，从而得出的一个或多个结论，还要把它们展示出来。这些就是用来量化改善开发实践价值的数据，也是最终用以获取支持的数据。

4.2.1 定量数据

广义上来说，定量数据包括所有可被测量的数据。很多人会认为，定量数据就是一些确切的数据，原因是它是数字且可被测量。由于数字可以被收集和量化，似乎数据很客观和合理。定量数据可以通过测量进行收集，但还需要对数据进行解读以了解它的含义。例如，一个方法具有非常高的圈复杂度（cyclomatic complexity，一种代码复杂度的衡量标准，具体请

看第 11 章), 通常表明该方法是很难去维护的, 但情况并非总是这样的。如果该方法包含一个非常大的 switch 语句, 它的圈复杂度也会具有较高的数值, 但是, 经过代码审查后也许会发现该方法是合理的且是可维护的。通过圈复杂度来提醒要进行代码审查是一件好事, 但不一定总是需要进行重新设计或重新编码。

实践 4-6 收集和跟踪指标及数据

定量数据的收集和跟踪是很有价值的——事实胜于雄辩。表 4-6[⊖]列出了要收集的定量数据的类型和一些措施示例。要记住, 这些数据意味着事实。收集和跟踪这些数据目的是为了分析并了解事实。这些数据还可以用于决策、回溯和学习。在第 11 章讲述静态代码分析的时候将介绍 NDepend, 一个用来计算许多定量措施的产品[⊖]。跟踪这些指标还能支持可管理性。例如, 测量非空白行、非注释行代码 (NCLOC) 对开发来说可能微不足道, 但是, NCLOC 与软件系统的整体大小密切相关。在这种情况下, 跟踪 NCLOC 有助于测量整体进度。使用 NCLOC, 可以通过计算不同的相对大小比率来规范数据。跟踪单元测试的总数除以被测试代码的 NCLOC 数量也许很有帮助。对于编写了大量单元测试, 但只专注于覆盖范围百分比的项目来说, 如果测试十分全面, 那么 NCLOC 比率就会有一个较高的初始测试数。如果单元测试面 (thoroughness) 急剧下降, 那么该比率也会下降, 这是因为在这种情况下, 单元测试会忽略边界条件和极端情况测试。

表 4-6 定量数据的类型以及措施示例

量化	测试数量、开发人员数量、每天的工作量、每天的缺陷报告数量、每天缺陷修复数量
大小	NCLOC、可交付文件大小、数据库大小
时段	创建时间、部署时间、加载时间、工期、编码时间特征 (time-to-code feature)
工作量	开发人员的人时、测试人员的人时、分析人员的人时、技术支持人员的人时、客户支持人员的人时、销售人员的人时
成本	持续集成服务器的硬件、持续集成服务器的软件许可证、生成工具、代码分析工具、代码覆盖率工具、设计分析工具
代码质量指标	圈复杂度、每 NCLOC 的缺陷数、覆盖率、传入耦合和传出耦合、继承树的深度

同样重要的是如何展示和说明数据。下结论要谨慎, 要确保这些结论与所了解的真实情况相一致。表 4-7 是在 6 个命名空间下类的数量的计数。记数又根据最大圈复杂度 (CCMax) 的范围进行了细分。在示例中, 原则就是, 如果该类的方法的 CCMax 值高于 15, 通常认为该类难于理解和维护; 而方法的 CCMax 值高于 30 的时候, 该类会相当复杂, 应当拆分成更小的方法或者重新设计。示例数据是从 NCover 输出的报表中收集的。可惜的是, 表格中的数据对于收集和说明来说都太枯燥了。图 4-1 要说明的数据与表 4-7 是一样的, 不过它是以仪表盘报表形式显示的。

⊖ 原书是表 4-4, 有误, 这里修改为正确的表 4-6。——译者注

⊖ Scott Hanselman 的一篇博客文章 www.hanselman.com/blog/EducatingProgrammersWithPlacematsNDepend-StaticAnalysisPoster.aspx, 对 NDepend 指标讲解得很好。

表 4-7 示例数据：类的数量和最大圈复杂度

命名空间	类的数量	CCMax<=15	CCMax<=30	CCMax>30
财务	29	15	11	3
数据访问对象	27	27	0	0
模型	15	13	2	0
Web MVC 控制器	11	11	0	0
数据访问层	9	9	0	0
公用	10	0	5	5

图 4-1 中的仪表盘报表是在持续集成服务器上生成的显示结果。MSBuild 脚本会读取 NCover 生成的 XML 文件，然后使用 XPath 来处理数据，最后使用 XSLT 来生成 HTML 格式的报表。使用这种方法量化数据并生成一目了然的图片，此类图片可反映出代码的复杂程度。有了这些信息，团队领导就会知道在 Financial 和 Common 命名空间内的类需要做进一步的代码审查。项目经理就可了解到这些方面的复杂程度并意识到需要审查这些代码。

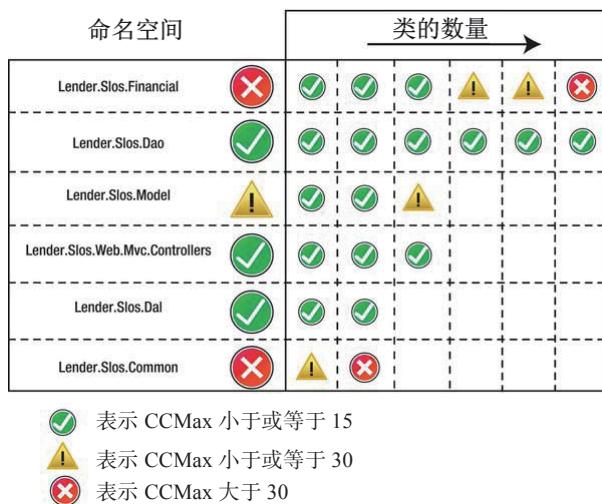


图 4-1 仪表盘示例：类的数量和最大圈复杂度

图 4-2 展示了如何通过图表化量化数据来协调开发目标和管理目标。实线表示 Lender SLOS Upgrade 项目的单元测试（用于测试已编写的代码）的实际覆盖率。虚线表示趋势。开发人员的代码覆盖目标必须满足以下要求：

- 30 个工作日后覆盖率大于 60%
- 60 个工作日后覆盖率大于 75%
- 90 个工作日后覆盖率大于 85%

根据图表中的数据，可以清晰地回溯任何相关的工作。能实现前两个目标是因为飙升的工作成果。跟踪覆盖率可以鼓励开发人员以让他更积极。项目经理则可通过结构质量指标查看结果和进度。

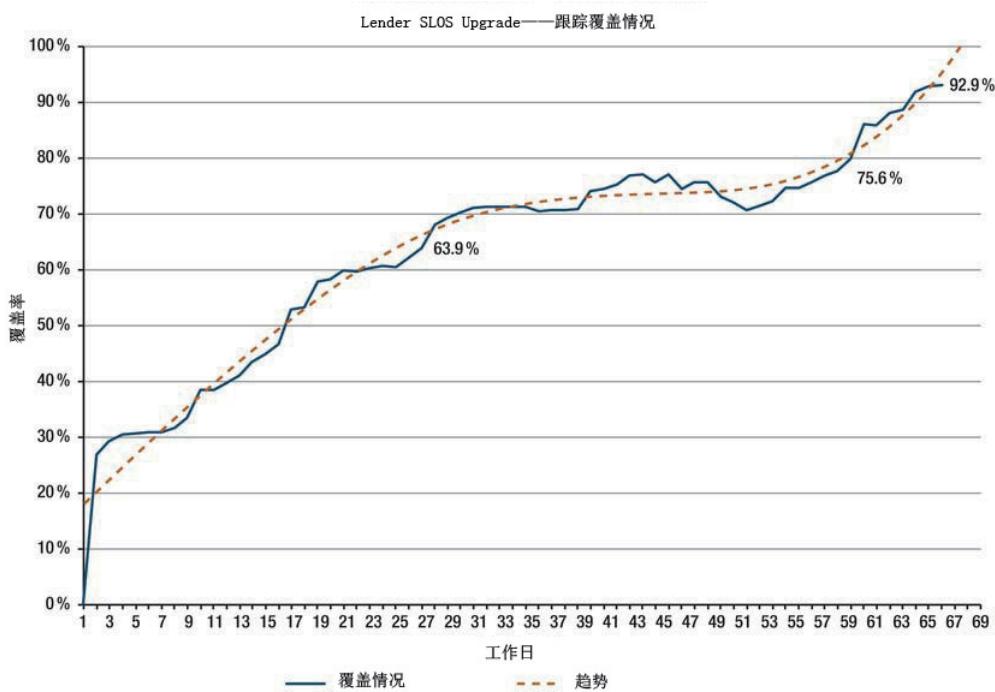


图 4-2 跟踪随着时间变化的覆盖率

趋势线显示了三个阶段的情况。首先是开发的早期阶段，开发人员正在第一个 30 天内学习编写高效的单元测试技巧。第二个阶段在 30 到 60 天期间，处于稳定状态。随着实践和技能的不断深入发展，新代码和新单元测试也不断添加进来。第三阶段显示在采用了新实践和深入了解了如何高效编写单元测试代码后，飙升的工作成果。70 个工作日后，覆盖率为 92.9%，远高于原计划。

在这种情况下，使用代码分析的实践方法来测量覆盖率，是非常有用和值得的。这很清晰地说明了量化数据有助于解释实践的价值。

4.2.2 定性数据

定性数据对应的是那些可被观察但不能测量的描述性信息。例如，当最终用户看到一个新的页面设计并描述它的外观时，这些就是定性数据。这些观察结果都要记录下来，并整理成一份定性评估报告。定性数据是主观的。调查是一种用来获取描述信息和观察结果，并相应整理成图表或统计报告的有效方式。不幸的是，要设计、实施和整理出一个好的调查报告却很难。大量的调查对量化更好的实践的价值不会有太大帮助。

由于定性数据还是有用的，因而一个较好的策略就是开始收集描述信息和观察结果。遵循的原则就是问一些能引起广泛回应的问题。例如，可以问开发人员：“这源代码是优雅的还是丑陋的？”要认真聆听回应中的描述。可能会听到很多抱怨，这时要深挖其中的细节。要收集并跟踪描述信息和观察结果。回应和讨论应集中在基本的定性话题上，例如：

- 外观
- 可用性
- 满意度
- 不确定性
- 信心
- 期望

实践 4-7 收集和跟踪描述信息和观察结果

表 4-8 列出了各种能引起大家回应，与定性描述信息和观察结果相关的示例问题。这些问题的目的是引起回应。要捕捉这些回应作为反馈和定性数据。随着时间的推移，如果更好的实践能发挥作用，那么这些回应就会相应发生变化。这是一个协调关键个人意见或一致性意见的非正式调查。它不用太科学，其主要目的是了解主观思想上的趋势。随着时间的推移，当大家感觉到有人聆听他们的意见时，获得的描述信息和观察结果也会越具体、越实用。最重要的一点是，要在团队中持续跟踪回应所涉及的问题的改善情况。

表 4-8 能引起回应的问题和随着时间推移的定性回应

能引起回应的问题	目前情况	30 天后	60 天后	90 天后
该软件是优雅还是丑陋？	丑陋	好了些	大为改进	优雅
该软件简单易用还是复杂烦琐？	复杂烦琐	易用了些	易用了不少	简单易用
该软件让人感到欣喜还是恼人？	恼人	略有失望	满意	欣喜
该软件的开发是可预测的还是杂乱无章的？	杂乱无章	有点规律了	大部分可预测了	可预测的
对部署是有信心的还是心虚的？	明显心虚	担忧	些许保留	有信心
该软件是能被接受的还是存在不足的？	存在不足	基本功能	更多新功能	能接受

4.2.3 业界证据

在没有足够的其他可用信息来量化实践价值的时候，可以使用业界证据来量化。业界说法是指在特定情况下发生的事情的描述。它只是一面之词，要成为证据，业界说法必须是真实且可验证的。要让业界说法发挥作用，就必须提供有助于审阅者得出有意义结论的信息。

实践 4-8 收集和分享成功案例

有两种形式的业界证据颇具说服力：成功案例和推荐做法。成功案例的侧重点是经验描述，推荐做法则侧重于可信性或相关来源。对许多管理人员来说，以下三个成功案例都颇具说服力：

- 自动测试：因为使用自动测试，所以在开发过程的早期就发现了一个非常棘手的错误，而问题在很短的时间内由少数的几个人就解决了。错误在列入 QA 列表之前就被解决了。由开发人员提出问题到解决问题只花了 1.5 小时。大多数人认为，如果错误被列入 QA 列表，那就需要测试人员和开发人员等好几个人花上好些日子去重现、诊断和解决。

- 自动部署：新的自动部署速度比之前手动部署快很多。部署时间从 8 ~ 18 个小时下降到约 2 ~ 4 个小时。部署也更可靠，更可预测。由于维护窗口更短，部署给最终用户带来的不便也就越少。现在，参与部署的人都觉得压力少了很多。
- 持续集成：现在，在持续集成服务器上运行单元测试，只要有失败的测试，就说明有问题，需要立即去解决。在推送更改过的代码之前，开发人员都会主动运行所有单元测试。当发现异常问题时，他们会修复代码。他们会针对集成问题提出适合长期需要的解决方案，并向团队领导建议变更设计。不单是 QA 测试人员，整个项目团队都在积极参与到提高质量上。

为了在项目中采用新的、不同的实践，要花时间去收集和整理成功案例。还要通过谈话、博客、电子邮件等方式去分享成功案例。自己组织内的经验自然是可信的和有实际价值的。很少有人反对成功的东西。

实践 4-9 从客户和专家那里获取推荐做法

一个好的推荐做法需要可靠的消息来源和相关环境。如果推荐做法来自于团队外的专家，那么这个人的专业知识和经验应该是被认可的。如果组织使用了敏捷开发，那么《敏捷宣言》的作者应该是可信的^Θ。附录 A 列举了许多备受尊敬的作家和最佳实践倡导者所写的书籍和文章。

相对于团队外部的专家，来自于最终用户或客户的推荐做法往往更可信和更被认可。因为他们是利益相关者。如果最终用户满意结果，他们会大声说出来。客户满意度非常重要，因为客户会用支票来给项目投票，管理人员绝对不能忘记这个事实。

4.3 小结

量化价值就是去了解这些价值是什么，并让它们更合理。大多数情况下，确切的数据是至关重要的。由于具体情况在不断变化之中，因而唯一可信的数据就是组织中收集和跟踪到的数据。开始收集和跟踪这些数据，并将它们与开发实践联系起来以反映改进效果，是一种好的做法。除了数据，还有许多观察结果和描述信息来支持新的、不同的实践。分享成功案例和收集推荐做法是一种有效的方法。使用多种积极方式去展示每一项改进是如何影响结果的。

在本章，学到了量化新的不同的实践的益处。还了解了高级管理人员、一线管理人员、最终用户和开发人员这些人对价值的定义的不同。要牢记，对某个小组来说有价值的东西，并不意味着对另一小组也有价值。在本章，还学到了许多不同的支持数据的来源。还了解了如何将间接措施、定性数据和业界证据综合到一起。在本章，还学到如何在开发方法、基于数据的结论和其他价值之间建立联系。

在第 5 章，将学习到许多战略方面的重要性。开发人员往往只注重战术问题和实现的解决方案。战略是认识问题、高层次的规划、设定目标和获得重要的见解。当真正且充分认识到战略状况后，那么更好的实践就会自然而然地浮现出来。

^Θ 参见 <http://agilemanifesto.org/authors.html>。

第 13 章

反感和偏见

本章将讲述很多人的倾向性以保证能采用的更好的实践。作为人类，我们所看到的世界，并不是它是什么样的，而是我们认为它是什么样的。如果我们的眼睛和大脑不会欺骗我们，那就不会有错觉了。[⊖]对此有以下两个基本理论：

- 生理错觉：人们基于对事物的感觉影响或改变了看法。
- 认知错觉：人们基于对事物的判断、思考或回忆影响或改变了看法。

虽然生理错觉很有意思，但它不是本章讲述的重点。没人会因为字面上看到的小问题就去反对一个更好的实践。本章更多关注采用更好的实践对心理所造成的影响和冲击。具体来说，往往有以下两种认知错觉会阻挠新的、不同的实践：

- 反感：个人或团队不愿意去接受、认可或采用一个新的和不同的实践，哪怕是一个可能的更好实践。
- 偏见：个人或团队趋向于或倾向于秉持一个新的和不同的实践存在潜在的危害的观念。

如果存在“反感”，开发人员或团队对实践是不会感兴趣的；如果存在偏见，开发人员或团队不会公平或完整地去了解实践。有些人认为“反感”只是更强烈的“偏见”。然而，区分它们很重要。人们通常更愿意去谈论他们存在偏见的实践。如果不考虑根据实践做出改变，那就是“反感”；如果只是矛盾、怀疑、不安或担心，这很有可能是需要克服的“偏见”。知道如何处理反感和偏见很重要。

没有人愿意把难以下咽的食物吞下，哪怕知道这是“健康”食物。人们反感新的、不同的实践的理由往往与此类似，甚至最终理由证明这是一个不太好的理由，但这仍然是一个理由。除了诸如生命或死亡等紧急情况下，处理“反感”最糟糕的方法就是强制个人或团体实施更好的实践。在任何情况下，最好的方式是先倾听。假设还有更好的但还不了解的理由，关键是去找出这些理由。这一点不奇怪，许多好的实践实现得好就在于找到了反对在实践中做出变化的理由。由于无法清楚表达他们的忧虑，因而，他们会反感。了解他们知道什么，或了解他们不愿意告诉你的想法，这是有好处的。不要指望能说服他们，而是要使他们相

[⊖] 该网站介绍了许多常见的错觉：<http://www.123opticalillusions.com/>。

信：你了解和赞同他们所关心的问题。

对于“偏见”这种情况，处理方式取决于偏见的程度和性质。如果轻微地混淆了，那么做一个简单的解释就可以消除偏见。有时候，做一点儿培训也完全有必要。其他情况可能要在小组演示时给出一个强有力案例才行。我们的目标就是致力于知识、技能和态度方面的变化来加强说服力。

广义上来说，本章讲述了认知偏见。这些是认知科学和社会心理学研究的可观察行为。实验表明，一个人实际得到的信息取决于该人的假设、反感、偏见和成见。[⊖]也就是说，人们不可能总是看到面前事实正确的那一面。人们更倾向于如他们所期望的那样感觉。这就是能影响人们评估事实的偏见。研究人员发现在制定决策、思考、互动和记忆中都存在偏见。认知偏见实在太多了，本章不可能为每一个这样的偏见都提供应对方法。因而，本章介绍了表13-1中列出的12种反感和偏见，并提供了应对策略。希望在了解了这些认知偏见对改变实践方法的影响后，你能掌握识别、应对和管理这些偏见的方法。

表13-1 应对反感和偏见：冷静待之的.NET实践

	策 略
13-1	使用事实来支持更好的实践，以减少不良后果
13-2	记录下出现的问题和事情，并在回顾期间提供建议
13-3	确定具体的个人和方法，并鼓励他改进自己的个人实践
13-4	通过增量的积极变化来改变现状
13-5	全面描述当前实践的优良品质和不良品质
13-6	不要想当然地认为每个人都有同样的技能和能力
13-7	对于被忽略的不利情况要提高认识
13-8	不能依赖过去的好运来防止未来的问题
13-9	尝试新的、不同的实践，以弥补缺失的信息
13-10	指出问题的多个方面来拓宽决策期间的关注点
13-11	要突出展示回报，以便促进渐进式改进
13-12	收集与其他人经历过的问题有关的业界证据来为“灾难”做准备

13.1 团体利益偏见

对人来说，自然而然地会倾向于信任成功者而疏远失败者。对于个人来说，这叫做自利性偏见（self-serving bias），而对于团队来说，就是团体利益偏见。[⊖]开发团队更愿意相信做什么会令团队成功，而不是那些会给团队带来无法控制或影响的状况或环境的困难和问题。这种偏见会阻止团体去采用更好的实践，因为他们不会一开始就承认他们对这些问题负有一定的责任。他们会抵制这样的说法：“我们本可以做得更好”。

⊖ 这本引人入胜的书涵盖了许多认知偏见：Richards J Heuer的《情报分析心理学》（华盛顿特区：中央情报局智能研究中心，1999年）

⊖ 详情请参阅 http://en.wikipedia.org/wiki/Self-serving_bias。

例如，开发团队在严重的进度压力下编写大量的代码，在系统发布前，只让质量保证（QA）测试人员做了一点单元测试或完全没做单元测试。开发人员对自己的生产力和非凡的工作能力都自我感觉良好。然而，QA 测试人员很快发现系统不稳定，重要的功能不完整或不正确。系统并未如预期的那样运行。在这个示例中，事实很清楚地表明软件为了保证部署质量，必须在准备部署之前进行 QA 测试。关于这一点，开发人员本应该承认：他需要在 QA 测试开始前对自己那部分进行更多的单元或集成测试，这样就可发现这些问题。

如果整个团队都明白并接受开发人员已交付了一个不稳定和不完整的系统，那么整个项目团队就应该共同承担起责任。而另一方面，如果开发人员忽视或歪曲这种情况，那么开发人员必须承担在系统准备好之前将其发给 QA 的责任。现实情况通常是介于这两个极端之间。更好的实践是：除非开发人员确信软件经过测试已经足够稳定、没有显著“搅局”的问题，否则不会发布软件给 QA。

实践 13-1 使用事实来支持更好的实践，以减少不良后果

经常有明确的事实可证明：已经发生的错误或正在发生的错误，而后来证明事情本可以做得更好。更好的实践是以“错误是可以避免且会有更好的方法来进行”这样的想法为前提的。要用事实来点出更好的实践。在本节的示例中，开发人员本可以坚持要求他们需要时间来确认系统是否已准备好进行 QA 测试。他们应该建立一套单元测试和集成测试系统。他们可以和 QA 一起努力来建立确定软件是否具备测试条件的最低标准和基本标准。这里最重要的实践是避免自利性偏见，并寻找证明一些更好的实践可以获得更好的结果的事实。

13.2 玫瑰色回顾

回顾问题、错误或事件，并认为它并不像看起来那么糟糕，甚至感觉不错。人们偏好于美好的回忆，并认为相对于实际情况，这不算消极或这更积极了。[⊖]问题来了，实际情况可能很可怕，但现在回想起来，问题已被最小化了。对于不可预测的和无法控制的事，这是可以理解的，因为它可以帮助人们克服事情造成的创伤。很少有人愿意去回忆事情是多么糟糕。但是，如果事情是可以预防的，这种偏见就有害无益了。潜台词就是“现在回想起来，这个问题并没有那么糟糕，因此，如果它再次发生，我们一定能看穿它”。

许多更好的实践因为这种倾向而没被采用。持续集成是一种有助于避免许多开发项目需要面对的长期而艰巨的后期整合周期的实践。大多数的开发人员都忘了后期整合是多么痛苦。开发人员会在回顾会议期间对实际情况进行辩解，而不是明确地去回忆所涉及的困难。后期整合会被作为一个不可避免的且需要克服的障碍来看待，并放在下次迭代或未来项目中克服。

实践 13-2 记录下出现的问题和事情，并在回顾期间提供建议

[⊖] 详情请参阅 http://en.wikipedia.org/wiki/Rosy_retrospection。

用简单的说明记录下出现的难点和问题。这些说明在回顾会议期间会很有用，因为它们可以回忆起哪些是需要认真和严肃对待的问题。他们有助于明确问题，并让人们把重点放在采用更好的实践来避免本可以预防的问题。

13.3 团体与个人的评价

在一些组织中，管理人员会因某一两个人的作为或不作为责备整个部门。典型例子是一封斥责组织中每一个人的电子邮件，原因是没有按时提交他们的时间表。那些按时提交了时间表的人会忽略电子邮件，因为电子邮件的斥责对象不是他们。具有讽刺意味的是，许多没有上交时间表的人也会忽略电子邮件。有人会认为该部门没有人会按时上交时间表，并认为自己的行为是正常的。最终，部门中的人会认为那个差评是给整个部门的，而不是他们自己的。

在很多软件开发领域，团队领导可以集合团队并告诉他们缺陷率高得令人无法接受。团队领导希望代码错误很多的一两个程序员接收到这件事的消息并做出改进。但这种情况很少发生，原因是这些开发人员并不认为自己应该对团队的高缺陷率负责。

实践 13-3 确定具体的个人和方法，并鼓励他改进自己的个人实践

在示例中，更好的做法是团队领导应该直接告诉那个开发人员，他的代码的缺陷率高得无法令人接受。要让开发人员意识到，他的个人表现对于团队来说是一个问题。让开发人员意识到期望改进他的个人实践这事，以便让他学习更好的实践提供了一个机会。

13.4 维持现状和辩解机制

人们习惯了的东西，就希望维持现状不变。在做事方式要二选一的时候就会去贬低其中一种方式。[⊖]现有的情况都是已熟悉且知根知底的。人们已经形成了习惯并知道做什么和如何去做。出于偏见，他们会为维持现状而进行辩解和辩护，并支持维持现状。对于任何打破现状的做法都会反对，有时甚至是阻止。有些人不能理性地探讨一个新的、不同的实践，只是因为这意味着要改变一直以来的做事方式。潜台词就是“不需要改变，因为任何的改变都是破坏性的和糟糕的”。

一些针对开发环境的做法和流程已经变得根深蒂固了。其中一个例子是手动部署。可能是开发人员将文件复制到目标服务器并修改配置设置。所有的事实都表明手动部署容易出错，原因是手动步骤经常被误用。有些开发人员会很小心，而有些则不是，这造成了部署在质量和可靠性上参差不齐。然而，部署一直都是这样做的。没有人愿意采用自动部署的解决方案，这在很大程度是因为人们喜欢维持现状。

[⊖] 详情请参阅 http://en.wikipedia.org/wiki/System_justification。

实践 13-4 通过增量的积极变化来改变现状

人们很害怕改变现状会让他失去什么。一个更好的方式是着重于带来一个稍微好一点儿的现状。新的现状应该维持旧现状的所有好处，同时采用一些增量改进。对于手动部署这种情况，可以使用命令文件将文件复制到目标服务器。开发人员仍然可以在对应的文件夹看到这些文件。命令文件还可以运行 MSBuild 脚本来设置所有配置设置。同样，开发人员要审查脚本以确保脚本能正常工作。该方式只是改进了任务较为烦琐和容易出错的部分，并没有减少开发人员的参与程度。根据这个增量的更好实践的结果来看，结果是相一致，部署也更可靠了。

13.5 优势错觉

许多人都习惯性认为自己目前的做事方式就算不比其他方式更好，也还算不错。换句话说，就是高估了优良品质和低估了不良品质，也就是所谓的优势错觉。[⊖]潜台词就是“我们正在使用的实践优于其他新的和不同的实践”。

例如，对待自动代码分析的态度。有些团队领导认为小组式的代码审查是唯一适合代码分析的形式。可以肯定地说，小组式审查源代码有许多好处。这些好处是不可否认的。然而，团队的代码审查小组也会有不良品质：一方面，他们占用了大量的时间；另一方面，他们不可能频繁地接触到足够多的源代码或覆盖足够多的源代码。最明显的是，他们使用的目的仅限于希望自动代码分析工具能轻易识别代码标准和规范。换句话说，小组所涉及的只是已经解决的事项和最终需要浪费时间来补救的可预防问题。

实践 13-5 全面描述当前实践的优良品质和不良品质

这样做的目的是为了能永远正确地去认识事物。不要忽略了当前实践中的不良品质而去做评判。对于手动代码审查这种实践，可通过自动代码分析来加强。例如，在团队一起进行代码审查之前，配置自动代码分析工具来检查所有的代码标准和规范是否能满足开发人员的预期。是的，异常是必要的，这能引起小组的注意。这样，代码审查就不再集中在微不足道的和固有的问题上，而仍然集中在解决方案的质量和设计的实现上。更好的实践就是承认当前实践的不良品质，并通过其他新的和不同的实践获得改进。

13.6 达克效应

有着丰富经验和专业知识的人，会臆测别人也会有与他一样的知识和技能。这种错误度量别人的方式就是著名的“达克效应”的其中一部分。[⊖]专家都习惯性地认为，如果他们认

[⊖] 详情请参阅 http://en.wikipedia.org/wiki/Illusory_superiority。

[⊖] 详情请参阅 http://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect。

为某些东西是好的，那么其他人也会认为该东西是好的，这是由于把专业知识当成了常识。在面对他人的反对和质疑的时候，这会削弱其主张的更好的实践。他们的偏见让批评者找到反对的理由，因为他们认为批评者的学识与他们是一样的。在大多数情况下，事实并非如此。

例如，一个编写单元测试代码的开发人员在10年内已经获得了足够多的经验。如果他还博览群书并保持进行单元测试，那么他就可为其他人带来经验和专业知识。由于资质、技能和知识的关系，他会坚定地认为单元测试是有益的实践。另一个不太有经验的开发人员，单元测试的经验有限，会认为该方法是没有生产力和无益的。有经验的倡导者错误地认为他或她的同事的技术水平与他一样，因此，他不会去促进单元测试这种实践。这样，一个从更高级开发人员那里获取经验和专业知识的机会就这样被错过了。

实践 13-6 不要想当然地认为每个人都有同样的技能和能力

在开始的时候就要假定每个人都有不同的经验、技术和能力。试想一下，那些了解你想想法的人，很可能会支持你主张的更好的实践。这类做法的建议有：建立概念性验证测试、设置试用期、运行一个试点项目或者设计一个训练。这样做的目的是让每一个人都紧密联系起来，共同去验证该实践是好还是不好。最重要的是要着重通过经验和秘技进行知识传授和倡导更好的实践方法。

13.7 鸵鸟效应

很多时候，人们会忽略一个明显的不利局面，而不是迎难而上。就如众所周知的鸵鸟一样，把头埋进沙子里，以避免严峻形势的包围。[⊖]处理面临的困境所造成的影响可能过于痛苦，因而有时人们希望通过忽略困境而让它自己去改善。还有人可能会认为，只要不承认不利局面，那么就不用为没有解决它而被追究责任。潜台词就是“如果我看不到这个问题，我就不用去处理它”。

在软件开发中，鸵鸟效应的一个例子是安全性和威胁评估。软件漏洞是日益严重的问题，然而，许多开发人员都未充分地投入到处理安全性和最大限度减少漏洞上来。开发系统都会有“设计安全”要求，设计和创建软件的开发人员要积极地去评估该软件的安全性。忽略安全性以及不关心漏洞和威胁都会造成不安全的设计。没有任何软件方面的改进是通过忽略潜在的或隐含在软件开发方面的现有问题来实现的。

实践 13-7 对于被忽略的不利情况要提高认识

安全开发生命周期就是每一个开发步骤都是出于考虑了安全的影响而做出的决策。其中一个例子就是威胁建模实践。通过该实践，设计将从威胁脆弱性方面入手，以便发现潜在的负面情况。通过威胁建模返回的信息，就可以改进设计。一般来说，直接面对当前的和潜在的负面情况是一种防止问题的最有效方法，也是采用了的更好的实践。

[⊖] 详情请参阅 http://en.wikipedia.org/wiki/Ostrich_effect。

13.8 赌徒谬误

对于玩抽奖的人来说，往往有一个错误假设，以为每天出现相同的号码（numbers），只要不选择该数字作为抽奖数字，这样，每抽一次就会提高一次中奖几率。偏信过去的事情会影响未来的概率就是赌徒谬误。[⊖]胜出的几率其实都是一样的，并不会因为过去的事情而产生影响。同样，过去的运气不会影响未来的运气。我们当中的一些“赌徒”会用它来抵制能防止问题发生的新的和不同的实践。潜台词就是“迄今为止，我们已经避免了任何问题，因此，可以推断我们可以继续避免问题出现”。

在软件开发中可以看到重要的实践会被忽略，只是因为问题尚未出现。例如，外行的程序员可能不喜欢反向控制。他们更喜欢在自己的开发环境运行不受约束的版本控制系统。他们不会定期地检查自己的代码或定期从其他开发人员那里获取最新的代码。他们所依仗的事实是，他们一直都能以这种方式处理事情，因而可以继续以这种方式运作。当他们运气用完的时候，问题就在后期整合中、丢失的更新中或更严重的问题中出现了，不遵循更好的实践，是不明智的，也是难辞其咎的。

实践 13-8 不能依赖过去的好运来防止未来的问题

对于采用更好的实践来说，直接和明确的沟通是非常重要的。必须明确指出，不遵循实践来防止问题出现是不明智的，也是不能接受的。任何过去有运气而没有遵循实践的开发人员都必须及早避免这事。项目中的“赌徒”必须停止赌博心理以确保一直遵循更好的实践。

13.9 歧义效应

当信息存在歧义或缺失的时候，人们就会倾向于避开指示、实施或选择等。这可能是一个更好的实践，然而只是因为没有足够的信息，这个原因也是针对潜在的更好的实践的偏见。[⊖]人们对歧义感到不安，但这并不意味着要避开新的和不同的实践，它意味着需要去了解、研究和重视实践。一旦理清不明确的地方，那么就可以确定这到底是一个有用的实践还是一个无益的实践。

关于“单元测试”出现了很多歧义的地方。一些开发人员仅仅因为存在的疑问比答复多而避开单元测试。他们避开实践是因为他们不知道从哪里开始或如何进行。这同样适用于其他新的和不同的实践，虽然该实践提供了很多的承诺，但也存在很多不确定性。

实践 13-9 尝试新的不同的实践，以弥补缺失的信息

通常，在缺失信息的情况下而犹豫不决是否采用更好的实践的时候，一点小小的经验或简短的教程就可以弥补信息的不足。有些时候，团队需要一小段时间来试用实践，以便理清

[⊖] 详情请参阅 http://en.wikipedia.org/wiki/Gambler%27s_fallacy。

[⊖] 详情请参阅 http://en.wikipedia.org/wiki/Ambiguity_effect。

一些模糊和疑惑之处。在不愿意确认可行性和实用性的时候，更是如此。对于单元测试，许多开发人员已经发现单元测试是有用的，并能提高他们的整体生产力，只要他们能体会到问题预防和解决早期缺陷的好处。试用潜在的更好的实践只是一个简单的举动，能弥补缺失的信息，并能说服团队采用该实践。

13.10 集中效应

有一种称为集中效应的倾向，即人们会在决策过程中过分强调问题或事情的某一个方面。^①常常有很多因素能影响到事情，然而，许多人只是探究或集中在一个特定因素上。由于这一奇特的集中，这样的偏见导致了其他方面的重要影响被忽略了。

在研究一个显著问题的时候，集中效应偏见会导致人们只探究一个根本原因。而实际上，该问题可能存在很多影响因素。每一个影响因素都可能为问题出现创造了条件或产生一些行为导致问题出现。解决问题就综合考虑多方面的因素，而不是，只认真考虑了一个方面。在实际开发中，我们应该选择一组更好的实践，而不是只选择其中一种。有时，建议的实践采用了但没有变化，原因是研究的方向只是一个不得不研究的方面。审查问题的所有的影响因素是一个更好的实践，因为更好的实践可以改变条件并避免问题。

实践 13-10 指出问题的多个方面来拓宽决策期间的关注点

把问题或事件发生的原因假定为多方面的，往往更好。发生问题时通常都存在多个条件。能影响到问题的，通常都是有多个动作发生了或没有发生。通过多种原因来拓宽关注点这种方式可以表明需要更多的新的和不同的实践才能改善这种情况。

13.11 双曲贴现

思考风险和回报不会总是以理性方式进行。加入时间延迟的元素后，即使延迟的回报会更大，人们还是会偏向于即时的回报。这种认知偏见就是双曲贴现的根本所在。^②鉴于两个选项具有相同的风险，相对于等待以后更大的回报，很多人更喜欢虽然小但今天就可以兑现的回报。今天的回报可以在今天享受到，但未来的回报需要等待才能享受到。

了解这个偏见有助于以渐进式的思路去改进更好的实践。在软件开发中，更好的实践不被采用，很多时候是因为会有几个选择，但没有哪一个是明确为喜欢的。有些实践需要很长的时间才能改进，而且需要在开始的时候耗费大量精力，但其潜在的好处是巨大的。其他的实践虽然有立竿见影的效果，但其潜在的好处却不大。在这种情况下，要着重于有立竿见影效果的实践。快速回报总会令人非常高兴，并有助于进一步地增量改进。决策者喜欢能即时回报的改变。

^① 详情请参阅 http://en.wikipedia.org/wiki/Focusing_effect。

^② 详情请参阅 http://en.wikipedia.org/wiki/Hyperbolic_discounting。

实践 13-11 要突出展示回报，以便促进渐进式改进

要花时间去考虑更多的、能被实现的更好的实践，有些实践可能会立即或在短期内得到改进。这些都是有早期回报的实践，应当予以推广。采用更好的实践是一种有助于渐进式改进的倡议，而不是激进式的变革。把之前选择的每一种实践的回报落实到位有助于保持所倡议的活动。

13.12 常态偏见

通常，人们只是因为之前从来没有发生过这样的问题或事情，就不为该问题或事情做规划，仅仅是因为之前从来都没有发生过。因为没有经历过，就有了偏见，从而抵制为问题制定规划。这就是常态偏见[⊖]。大多数人都会为预计中可能在正常情况下发生的不测事情做规划。而另一方面，会有意想不到且灾难性的不寻常的事情发生，而很多人对此都没有做任何准备。有些人仅仅因为他们之前没有经历过这样灾难，就拒绝为这样的灾难做规划。

如今，许多企业正经历着网络攻击和黑客入侵之类的问题。然而，一些开发团队会拒绝为这些攻击和其他威胁做规划或准备。与鸵鸟效应不同，危险还没显露出来，因而很难指向一个特定的或已经发生的问题。现在，一切看起来很正常、很平静，但等待着我们的会是一场灾难。更好的实践仅仅因为问题以前从来没有发生过就遭到了抵制。

实践 13-12 收集与其他人经历过的问题有关的业界证据来为“灾难”做准备

你是否读过介绍其他组织不得不应对拒绝服务攻击的文章？这些故事从令人烦恼到令人讨厌不一而足[⊖]。在最近的新闻中，一个大公司发送电子邮件告知他们所有的客户，他们的个人信息（包括信用卡信息）已经泄露了。考虑一下，如果不让客户知道他们的个人数据已经泄露的话，那么这将给自己的组织带来多大的影响。其他人的可怕经历就是纠正常态偏见最有力的证据。要收集那些不遵循更好的实践的业界证据和其他人的经历，并利用这些信息来帮助改进实践。

13.13 小结

本章介绍了认知偏见对遵循或改进更好的实践的影响。由于软件开发明显受到个体互动的影响，要考虑个人或团体对不同的实践的感受和想法是相当重要的。现在，已经了解了仅靠合理的论据来消除赌徒谬误、达克效应和团体利益偏见的影响是不足以推进实践的改变的，还需要了解反感，还需要去抵制偏见。

[⊖] 详情请参阅 http://en.wikipedia.org/wiki/Normalcy_bias。

[⊖] Phil Haack 在 <http://haacked.com/archive/2008/08/22/dealing-with-denial-of-service-attacks.aspx> 描述他的经历。



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

1kg 多背一公斤

.org

爱自然 | 更爱孩子

