制定敏捷实施策略

敏捷实践实施模式

技术实践组合

Amr Elssamadisy 著

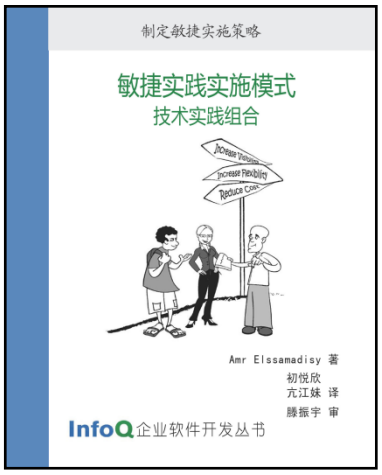
初悦欣
亢江妹 译

滕振宇 审

InfoQ企业软件开发丛书

免费在线版本

（非印刷免费在线版）



了解本书更多信息请登录本书的官方网站

InfoQ 中文站出品

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

本迷你书主页为

http://www.infoq.com/cn/minibooks/agile-patterns-cn

致谢

有很多人与我一起分享他们的想法和见解，为本书出谋划策，在此我想对他们表达最诚挚的感谢。

第一个要感谢的人是我的妻子 Maha。她一直鼓励我，督促我，帮助我，尽一切努力让我好好写下去。同时，她还花了相当多时间编辑这本书，编辑今年我写的与本书相关的论文和文章。

接下来要感谢Ashley Johnson、Dave West和Ahmed Elshamy。2006 年春亚利桑那 ChiliPLoP大会 [1]期间，我与他们花了两天半时间讨论模式、敏捷实践以及实施。我们四人共同分享了过去多年的经验，并以模式的形式进行了分析整理。在那之后，Ahemd帮我在 XP2006 组织了一个研讨会，与会者有 40 多位来自世界各地的敏捷实践者。会议期间我们向大家介绍了这些想法，同时也从与会者那里收集到了更多的信息。Ashley花了无数个小时思考、琢磨这些想法，使之更为精炼准确，再汇总到本书中。

Dave 和我接过了 ChiliPLoP 的工作，做了进一步的改进，然后在 PLoP2006 上做了介绍。期间，另一组人又再次针对它进行了讨论。特别要感谢 Ademar Aguiar 在 PLoP 对我们无微不至的关心和照顾。Linda Rising 和 Mary Lynn Manns 也读了这本书的早期版本。Richard Gabriel 组织了审核这本书的讨论会 ,参与者包括 Donald Little、Rebecca Rikner、James F. Kile、Till Schümmer、Lise B. Hvatum、Joseph Bergin 和 Guy Steele。

今年早些时候 Jean Whitmore 和我合写了一篇文章，它是功能测试模式和测试驱动需求组合的基础。Jason Yip 曾对这个模式提出指导，并帮我们修改、提炼了在 PLoP 上的演讲。Ralph Johnson、Jason Yip、Hesham Saadawi、Dirk Riehle 和 Paddy Fagan 组成一个小组，对该模式进行了复审。本书中所收集的模式，来自于我自己的实践，也汲取了很多其他人的经验。在 ChiliPLoP，XP2006 的研讨会以及 XPDay 蒙特利尔 2006 的自由讨论会上，很多人都曾参与讨论这些模式，分享各自见解，还竭尽全力去去找出其中的共同之处。没有他们的参与，不可能有这本书。这些人是（按名字先后排序）：

Soile Aho, Görge Albrecht, Walter Ambu, Giovanni Asproni, Emine G. Aydal, Meir Ben-Ami, Gilad Bornstein, Filippo Borselli, Ole Dalgaard, Ian Davies, Vasco Duarte, Emmanuel Gaillot, Gabor Gunyho, Janne Hietamäki, Mina Hillebrand, Ashley Johnson, Kan Karkkainen, Tuomas Karkkainen, Maaret Koskenkorva, Krisztina

[1] ChiliPLoP 是在美国举行的 Pattern Languages of Programs 的会议，其网址是：http://hillside.net/chiliplop/

Kovacs, Juha Laitinen, Andreas Larsson, Mikko Levonmaa, Youri Metchev, Aivar Naaber, Paul Nagy, Keijo Niinimaa, Loua Nordgvist, Virva Nurmua, Marko Oikarinen, Jukka Ollakka, Paolo Perrotta, Dimitri Petchatnikov, Ron Pijpers, Aussi Piirainen, Ilja Preus, Timo Pulkkinen, Niko Ryytty, Abdel Aziz Saleh, Aki Salmi, Meelis Salvvee, Timo Taskinen, Olavi Tiimus, Ingmar van Dijk, Jussi Vesala 和 Daniel Wellner. Filippo Borselli, John Mufarrige, Ron Jeffries, Floyd Marinescu, Deborah Hartmann 与 Kurt Christenson 花时间阅读了本书的草稿，并提供了他们的反馈，让这本书变得比以前更臻完美。

最后，感谢来自 InfoQ 的 Floyd Marinescu 和 Deborah Hartmann，他们让我有机会撰写本书，并使其面对广大读者。

Amr Elssamadisy

阿默斯特，马萨诸塞

2006 年 12 月 9 日

目录

序言一

Amr 为我们绘制了一副敏捷地图，并展示了它的用法。对于软件项目来说，这好比一本旅游指南。几年以前，我和妻子 Ricia 一起在意大利旅行，跟 Martin Flower 和他的夫人 Cindy 一起呆了几天。他们并不比我们更加熟悉周围环境，但他们很有旅行经验，Martin 非常善于阅读地图，即使身处意大利最忙乱的交通状况中，认清方向对他来说也不在话下。比起我们以前单独旅游，和他们一起时，我们游览了更多的风景名胜、扑捉到更多的趣闻异事，迷路的次数却少了很多。从 Martin 和他妻子 Cindy 身上，我们学到了很多。因此在他们离开之后，我们单独旅游也大有进步。

无论何时，当我们在一个未知的地方旅行时，有一个熟悉当地的向导相伴，无疑会非常棒。如果找不到向导，最好有一个会读地图、善于识别道路标记并熟知各种地理标识的人。如果只能靠自己，学会如何做到这些也是很有帮助的。

软件项目往往是在未知的领域进行探索。对于我们比较熟悉的方面，我们会做得很好。其他的部分我们可能不太熟悉，需要得到帮助。在敏捷项目中，特别是当我们刚开始应用敏捷时，有些情形看似熟悉，但敏捷思维往往需要我们以新的方法来应对。

敏捷项目以交付业务价值为中心，这是 Amr 的出发点。他描述了不同类型的业务价值，帮助我们选择自己组织需要的业务价值。

接下来，Amr 开始帮助我们识别业务和流程中的“异味”，这些迹象表明有些事情不对劲。这部分读起来感觉特别象我以前在软件开发中遇到的故事。我见到过所有类似的现象，估计您也是。好消息是：接下来，Amr 将帮助我们改善这些方面！

首先，Amr 帮我们识别哪些敏捷实践能够尽早达到上面列出的商业目标，哪些实践能够解决领域中遇到的问题。他告诉我们如何达到目标，如何应付这个过程中所遇到的困难。在这部分结尾，他告诉我们如何根据最需要完成的目标，确定在项目一开始应该采纳哪些敏捷实践。

一旦项目开始进行，是不是就只能靠我们自食其力了？完全不是。迄今为止我说到的只是本书的 20%的内容。接下来的第二部分中，Amr 描绘了敏捷开发方式的基本元素——技术模式，包括测试、重构等。对每一个模式，Amr 都描述了其对应的业务价值，还用一个小故事说明如何让这个实践融入开发流程。然后 Amr 会给出使用这个模式的上下文，在付诸于行动时将感受到的效力。他帮我们发现我们需要做什么，怎样着手。

最后，在第三部分，Amr 谈了敏捷实践组合。他给出一些例子，说明如何把单个的实践组

合在一起进行应用，使用这种更强大、更可靠的方法去交付价值。

在一个新的地方旅行时，没有什么能比得上一个富有经验的向导或旅行者的帮助。类似地，如果您刚开始尝试敏捷软件开发，没有什么能比得上一个经验丰富的教练。如果有可能得到一个教练的帮助，那就想尽一切办法去做到这一点。

在旅行时，无论有没有一个向导，您还得需要地图或书籍来参考。Amr 写了这本敏捷旅游指南，如果您还得需要找到自己的敏捷之路，我建议带上这本书踏上敏捷之旅。

敏捷对于开发软件来说是一个好方法，希望能在这条旅途中的某个地方看到您。好好品味这本书吧！

Ron Jeffries

序言二

敏捷方法及实践的实施，如 Scrum、敏捷建模和测试驱动开发（TDD）等，正在业界蓬勃发展。但是“快速实施”和“很好地实施”则完全不是一回事！很多人混淆了敏捷方法的核心，认为实践比价值观更重要。然而，敏捷方法的本质是敏捷宣言所描述的四条价值观（“人和交互重于过程和工具……”）。结果就是，当他们试图实施一个具体的实践时（比如 TDD 或 Scrum 日常站立会议），出现了很多问题，因为他们只是重视实践的形式而不是背后的原则。实施的形式会因环境的不同而变化，背后的原则才能真正指导敏捷方法的实施，让其充满活力。采纳敏捷方法的实质是价值观和原则的改变——思维的改变——而不仅仅是具体的实践本身。

另外一个相关的关键点是，敏捷方法按道理应该由自组织团队去实施。在这样的团队中，“开发人员掌控开发流程”（引自 Jim Coplien 的话）——即团队自己决定该采纳什么，怎样实施。然而，我们却看到越来越多的“由上而下”的强制性的实施（如“你们必须要使用 Scrum”）。

这些现象意味着人们并不理解敏捷宣言的核心价值观和原则。相反，大家致力于无数“表面”实践以变得敏捷。这是一个严重的错误。

Amr 懂得这种错误，而且他知道应该怎样在敏捷价值观的深层内涵的指引下，去帮助别人成功采纳具体的实践。在过去多年作为教练、与其他教练协作的过程中，他积累了实施的智慧和技巧。Amr 在这些模式中分享了他的技巧和经验。遵循这些建议，您会节省很多时间和投资，避免经历无谓的痛苦和困难。

Craig Larman

Valtech 首席科学家

丹佛，科罗拉多

译者序

来旧金山工作几个月了，很喜欢这里，其中很重要的一个原因是它的城市布局。旧金山的市中心很繁华，基本上全是密集的高楼大厦和商业中心，但城市其余近 80%的面积都被住宅区环绕，另有两座大桥（其中之一就是著名的金门大桥）通往加州北部和中部，这样的布局使得人们的工作生活非常方便，白天人们在市中心工作，晚上只要通过公共交通或者开车很快就可以到家了。象笔者已经住在靠海的地方了，坐快车也就十五分钟的时间。这就是一个好的城市布局模式可以让人生活地非常舒适。

这个道理同样适用于软件开发，一系列好的开发模式可以让团队在开发出卓越软件的同时，提高团队成员自身的素质，让人们乐于其中。软件的开发伴随着学习，如果一个团队的成员已经具有主人翁精神，那么只需要一系列好的模式来加快他们的沟通和学习进程，成功的软件产品也就指日可待了。本书有许多好的软件开发模式，就像好的城市布局一样，需要你亲身来体验一下并融入其中。希望本书在国内许多公司实施敏捷的过程中能贡献出自己的一份力量。如果喜欢本书，请参阅《软件开发成功路线图-敏捷模式》一书，该书是本书的源本，除了本书描述的内容外，还增加了许多更完备的敏捷开发模式。

借此机会感谢我的父母和支持我的朋友们。

初悦欣

这本书适合您吗？

您在采纳一些敏捷实践吗？或者您正在慎重地考虑是否要在团队中尝试一些敏捷实践？您读过一本有关敏捷方法如极限编程、Scrum 或测试驱动开发的书吗？或者至少感觉在理论上很令人信服，正想做一些尝试？

或许您刚成功完成第一个敏捷项目，另外一个团队希望您能加入去帮助他们。当然，每个项目都是不同的。那么，您上次使用的实践会在这个项目中同样有效吗？不一定！这本书将帮您了解为什么“不一定”，然后决定什么实践应该采纳，同时对于哪些实践需要进行调整、如何做出调整也会给出一些提示。

也许您很不走运，曾参加过失败的敏捷项目（或许您目前正在一个失败项目中）。读读这本书，了解一下为什么您正在用的那些实践可能不适用。敏捷地实施敏捷。

如果您属于上述任一情形，那么这本书对您就是合适之选。它将帮助您了解各个实践、其间的关系，传授您一种实施策略。很多公司都采用过这个策略，并在很多项目中获得了成功。同时，这本书还将提醒您实践会出什么样的问题，如何识别并解决那些问题。这不只是我一个人的意见，也不是从未试过的新方法。您将要用的这些模式都来自于若干真实的项目案例。

最后，这本书不适合于：

- 已经获得敏捷实践经验，寻找新理论或方法的高级实践者。所有的信息都来自于多个项目的经验，或许这些您已经都知道了。
- 从零开始的初学者。这本书没有从零开始描绘每一个实践。如果想要阅读深入全面探讨敏捷实践的书，这本小书是个不错的备选。
- 只对敏捷开发非技术实践感兴趣的人。那些实践也很重要但本书没有涵盖。

简介

这本书中我们将一起关注敏捷实践的实施。我将帮您解答一些基本问题，您可能正想知道：

- 我该从哪里下手？
- 对我们特定的环境来说，哪些实践最适合？
- 怎样逐步实施这些实践？
- 有哪些陷阱我需要注意？

计划

对于上述问题的答案，本书提供了一些指导。除此之外，我会给您更多问题，它们都是需要在敏捷的旅程上考虑和回答的。是不是这些听起来太过完美，以至于不太现实？的确没那么完美。我们中的很多人在敏捷社区里面摸爬滚打了好多年，发现了一条艰难的路径——通过试验和错误来学习。这本书分享了大家的经验。读完这本书，您将会做到：

1. 以客户的业务价值为中心。 列出对许多客户重要的业务价值，例如“减少成本”。
2. 识别当没有交付业务价值时，所发生的种种症状。我把这些症状称为“异味”。与“减少成本”相关的一个“异味”是：“客户什么都想要（甚至包括厨房的洗涤槽[2]）”。您还能把这些业务价值、异味和某个敏捷实践关联起来。
3. 利用 1、2 中所得到的信息，决定采纳哪些实践，以提高您组织的业务价值，并消除呈现的异味。此时您将能够拿出一个粗略的敏捷实施策略。
4. 以模式的方式对每个实践给出详细阐述，涵盖实施相关的种种细节。
5. 找出能够彼此配合、产生卓越效果的的实践，形成组合。把这些组合与业务价值、异味也关联起来。描述这些组合及其实施策略。

[2] "Everything including the kitchen sink"是一个比喻，意思是“所有的东西，无论是否必要”。——译者

范围

这本书的第一部分阐述了一种对所有开发实践都适用的实施策略。第二部分和第三部分阐述了技术实践以及相关的组合。为了使这本书尽量短小紧凑，所涵盖的实践仅限于：

- 自动化测试
- 测试后行开发
- 测试先行开发
- 重构
- 持续集成
- 简单设计
- 功能测试
- 集体代码所有权

以及组合：

- 演进式设计
- 测试驱动开发
- 测试驱动需求

本书没有包含其他实践如迭代、站立会议、客户作为团队成员等。其中很多在附录《引用到但未定义的敏捷实践模式》中有简要介绍。

如何阅读这本书

关于您需要做什么，已经谈得足够多了。那么您怎么做呢？第一件必须做的事，是为您的团队找出一组敏捷开发实践。您可以阅读第一部分（不到 20 页），完成每一章后面的练习，这样您就能够得到答案了。花一定的时间，认真完成那些练习，这非常重要。完成这些章节之后，您将会得到一组排好优先级的实践。

此时您可以开始阅读第二部分，它包括了敏捷实践模式及其组合。针对您那一组排好优先级的实践，阅读对应的每一个模式，深入挖掘，看它是否的确适用于您的环境。发现一个合适

的，就可以参考书中的指导，着手逐步实施。同样，您也需要利用每个模式中所提到“异味”，注意发现走错时的症状，避免落入陷阱。

最后，持续评估已采纳实践的有效性，及时调整，以争取为您的组织获得更大的业务价值。翻到下一章，马上开始吧！

QCon 全球企业开发大会（北京站）

QCon北京2011大会讲师陆续确认

——1月31日7折优惠报名结束

Spring首席架构师
Juergen Hoeller

视觉中国网站技术总监
潘凡

领域驱动设计之父
Eric Evans

前金山技术总监
许式伟

Twitter服务团队的
总工程师 Evan Weaver

腾讯公司互联网测试部
助理总经理 吴凯华

《软件开发成功路线图—
敏捷模式》作者
Amr Elssamadisy

台湾软件技术的教父级
人物代表 高焕堂

Oracle TopLink的首席
产品经理 Shaun Smith

上海贝尔产品研发集团
研发能力总监 田立新

全球最大视频网站Netflix
架构师 Adrian Cockroft

淘宝网负责主站性能
评测团队负责人吴毓雄

- 1月31日前单人报名可以享受课程通票价格7折优惠，单张通票可以节省1080元！团购（5人以上）报名可享受更多优惠

- 2月1日至2月28日门票价将调整为8折，更多信息请关注QCon北京2011大会官方网站。

时间：2011.4.8-2011.4.10
地点：北京-京仪大酒店
网站：www.qconbeijing.com
主办方：InfoQ
电话：13911797020
010-89880682
010-64738142
E-mail：qcon@cn.infoq.com

第一部分：业务价值、异味和实施策略

看来，您对敏捷开发感兴趣，为什么？您还想改进软件开发过程，这又是为什么？很多人会回答："为了开发更好的软件"。再多问一句，为什么要开发更好的软件？在敏捷社区里，关注的是客户，我们希望软件能给客户带来更多的价值。

这一部分将重点介绍如何为客户带来更多的价值。不过，同样的东西在不同客户眼中的价值各不相同，那么您的客户最看重什么？"业务价值"这一章，将介绍客户认为重要的常见业务价值。 在阅读并完成这一章的练习后，您会对客户所关注的东西有一个深刻的认识。这会帮助您选择实施合适的实践，从而为客户创造更多的价值。

本书关注点在于如何实施。不是每个人都会采纳新的开发实践，用以改进他们当前的工作状态。如果像我一样，只在发现问题的时候，才去寻找解决方案，那么"异味"这一章就是为您准备的。读读这一章，可以了解到当软件开发流程出现问题的时候，会散发出什么样的异味。完成章后的练习，做好创建敏捷实施策略的准备，这可以帮助团队摆脱困境。

第一部分的最后一章名为"实施敏捷实践"，本章将告诉您如何利用业务价值和异味，有针对性地采纳一系列敏捷实践，来解决客户真正关心的问题，交付他们珍视的价值。该章最后的练习会指导您针对自己的环境量体裁衣，创建一个初步的、按照优先级排序的实践列表

第一章　业务价值

第二章　异味

第三章　敏捷实施策略

1

业务价值

给客户带来价值，是所有敏捷实践的主要动力来源。我们之中有多少人明确知道：哪些价值对于我们客户的业务来说最为重要？又有多少人知道 :我们使用的软件开发实践能为客户交付什么样的业务价值？接下来的章节不但会逐步揭晓答案 ,而且会告诉您如何利用这些答案来选择合适的实践方法。

在这一章中，首先，我们会整体了解各种不同领域的业务价值；然后，本章列出了 7 种最常见的业务价值，并对这些业务价值一一进行描述。通过阅读本章，您可以了解客户真正关注的价值是什么。

章节最后的练习十分必要。如果想要选择恰当的敏捷实践，就请认真做这些练习，它们会帮助您发现客户最为珍视的业务价值。

缩短上市时间

缩短软件的上市时间，客户可以尽早使用产品，从而为他们带来了更多的价值。如果公司生产的软件是商业产品，那么显而易见的是，软件上市越早，公司也就可以早点赚钱。

让我们进一步思考一个问题 :客户能否从部分交付的软件功能中获得价值？好比说完成了 5 个预定用例中的两个，他们会觉得有价值么？与其从最终版本中一下子得到整个软件，客户通常更愿意早一些试用软件功能集合的一部分子集。因此，频繁、递增式的版本发布不仅能从总体上减少上市时间，而且也增强了软件的业务价值和实用性。

一些实践方法可以帮助您和您的团队更早、更频繁地发布软件，由此可以打消客户对上市时间方面的担忧，从而为他们带来了业务价值。

提升产品的市场价值

软件开发需要提取出抽象的需求，并开发出相应的系统以满足这些需求。开发团队提出解决方案来满足商业需求，这种从抽象理念到可运行软件的实现是一种创造的过程。然而，满足商业需求的解决方案众多，哪个才是最合适的？许多实践方法可以帮助您进行正确的抉择，

从而创造业务价值。

那么，如何判断哪种解决方案更好？最终，它是对客户来说最有用的软件。它能为客户的工作带来帮助么？只有明确了这些，最终才可以开发出对客户最有价值的软件。那些可以帮助客户选择更好的解决方案，并且能够及时传达给开发团队的实践方法，同样会带来业务价值。

最后，提升产品的市场价值与减少上市时间这二者也有联系。产品上市越早，就有可能越早从市场收集反馈信息。由于有了频繁和具体的反馈，团队就有可能增强软件产品对于客户的实用性。一些实践方法可以帮助您和团队从这些信息中发现价值，这也就提升了产品的市场价值。

提高产品质量

缺陷、可用性和可扩展性等许多方面都与产品的质量相关，它们也是开发团队所能遇到的最明显的问题。有助于解决这些问题的实践方法，能增加产品的业务价值。

提高灵活性

当业务方向发生变化时，是否能够及时做出响应？这才是“敏捷”这个时髦词背后所隐藏的业务价值。例如所在地的税收规章制度发生了改变，那么就需要所使用的财务软件能及时地适应这样的变化。

提高灵活性的价值对于客户而言并不明显，但是缺乏灵活性却会影响到其他的业务价值，例如产品上市速度过慢、质量不高。那么，为什么我把它作为一种价值单独列出来呢？越来越多的公司直接意识到灵活性的这个概念（也可以说变得敏捷）。客户希望能够了解您快速响应他们需求变更的能力。

在业务发生的变化时，一些实践方法可以帮助软件开发进行相应调整，因此也就帮助您和您的团队提升了业务价值

增强可见度

客户有权了解项目的进展，并获悉项目的真实状态。这一点很重要，客户可以因此掌控项目，从而可以控制风险和期望值。

如果缺乏可见度，当项目最终没能按时完成时，客户会感到吃惊与失望。因此，将会导致信

任危机、责怪与CYA[3]文化。

反之，增强可见度的软件实践，能够使客户从整个开发过程得到好处，并且最终会带来更多的信任与合作。

降低成本

为了生存下去，我们必须做得更快，更好，更便宜。之前的描述已经涵盖了更快（快速投放市场）和更好（上市的质量和价值）。降低成本这个业务价值是关于如何以更少的成本（更便宜）开发系统。

开发、维护和软硬件平台的开销，都与软件开发的成本相关。有些实践方法可以在保证质量的基础上，缩减这些费用，从而促进了整体成本的减少。

少写代码是节约成本的另一条途径。80/20 原则说明：在 80%的时间里，人们只使用了产品 20%的功能。有些实践方法，能够帮助团队针对需求进行优先排序，并开发出最需要的功能。这些方法也就降低了产品成本，从而为客户带来了业务价值。

延长产品寿命

更长的产品寿命会直接影响到投资回报率（ROI）。不幸的是，软件维护需要连续性，时间越长维护起来越困难。许多公司花费了巨大的精力维护多个版本的产品，但是最终也不得不因为成本的原因而停止了维护。

对于许多产品型的公司来说，延长产品的生命周期具有很重要的业务价值。许多敏捷开发实践能够增强代码的可维护性和灵活性，开发团队也因此能够保持产品的生命力。这些实践方法直接或间接地延长了产品的寿命，从而给客户带来价值。

理论到实践：确定您所在组织的业务价值

通过回答下面这些问题，就能够实际了解什么样的业务价值对于您的客户和组织机构最重要。一旦收集到足够的信息，立刻与别人分享，很可能他们根本不知道这样的信息。

1. 什么样的业务价值对您的客户最重要？请加以排序。
2. 邀请客户按重要程度对业务价值进行排序。与您的列表相比，他们是怎么排序的？根据

[3]可能有些读者不知道这个缩写所包含的意思，它是“盖住你的屁股 cover your ass”的缩写，意味着推卸责任，是指一种状态，人们只希望逃避任何针对自身的指责，而不关心能否为团队创造价值。

业务价值优先级的不同，您以后会如何调整自己的工作方式？

3. 在您的业务中，还有什么业务价值非常重要？自己思考之后，再就这个问题问一下您的客户。（例如是否包括 “个人成长”或“支持开源发展”等价值）
4. 考察一下自己对业务价值的感知度，您是否真的关注于增强业务价值的活动？团队中的成员是否都能意识到哪些活动中真正蕴含着业务价值？如果答案是否定的，那就要想尽办法灌输给他们！
5. 针对之前在组织机构里发现的业务价值因素，您应该如何调整实践方法才能为客户交付更多的价值？
6. 对于每一种业务价值，至少找到一种方法，来验证是否取得了进展。也就是说，如果采用一种实践方法去增强某种业务价值，您需要定期来验证这个实践是否奏效。不一定非得采用定量方式，有时定性方式反而更为自然，并尽可能做到简单易行。举个例子，如果您想衡量是否降低了成本，那么可以查看这样一个简单（且粗略）的数据指标：一次重大版本发布所包含的工作小时数。

2

异味

敏捷社区使用“异味”这个词作为一种指示信号，说明有一些事情出了问题。存在异味说明没有交付应有的业务价值。如果想知道需要解决什么问题，按照什么样的顺序进行，这个概念非常有用。对大多数组织而言，更自然的方式是去识别并解决当前存在的棘手问题，而不是直接投入精力去改进工作流程。

异味与业务价值之间的关系并不是一一对应的。一种异味常常是一种或多种业务价值有待改进的症状。反过来，从每一个有待改进的主要业务价值中也都会发现一种或多种异味。

在本章中，我先介绍两种不同类型的异味。一是业务异味，指可被客户察觉的异味。另外一种是流程异味，只对开发团队是可见，客户不易察觉。既便如此，流程异味也直接影响着业务价值的交付。

接下来，这一章会列出几个业务异味与流程异味，并一一描述。它们是开发过程有问题存在的指示器。在决定应该采纳哪些敏捷实践时，异味是很好的出发点——也就是说，只要找出能有效去除异味的实践就可以了。认真研读本章里面的这些异味，并且看看能否从您的组织中发现它们。

与之前一样，请花时间做章后的练习，把本章中的理论与组织的环境联系起来。

业务异味

如果把业务价值比作一枚硬币的正面，那么业务异味就是其反面。业务异味是客户由于软件不能达到需要而感受到的种种痛苦。接下来的部分，将列出五种常见的业务异味，并从开发组组织的角度，对其进行详细的描述。

交付质量令无法客户接受

我们的客户对产品质量不满意。事实上，我们也是费尽周折才让他们升级到最新版本。不幸的是，他们吸取了教训，知道升级到最新的版本就意味着不得不应付若干我们没发现的 bug。我们正在失去客户，在市场上名誉扫地。我们必须提供质量更好的代码。它已开始影响到我

们的生存底线。

交付新功能需要太长时间

我们根据客户要求添加新功能十分困难。加入新功能、全面测试、然后再部署到用户环境，整个过程耗时太长。竞争对手增加新功能的速度很快，我们跟不上，无法与对手抗衡。因为很多问题得不到改善，导致了我们发布周期过长，其中包括：

- 有些功能依赖专家资源的帮助，而专家资源又是瓶颈。
- 测试周期过长。
- 过去没有预测到现在需要的功能，由于现有架构的限制，难以添加这些新功能。

有些功能客户没有使用

我们的研究表明：我们添加的很多新功能，用户都没有用到甚至从来就没注意过。这是由多种复杂的原因造成的：

- 在需求阶段客户并不知道他们真正的需求，因此我们都是基于错误的假设来开发系统。
- 我们组织中的市场部门只是客户的传声筒。从市场上来的需求只是预测而已，而且这种预测并不是每次都准确。
- 一些功能的使用频率不像我们想象的那样高。我们认为，这表明我们设想的优先次序与客户头脑中的优先次序不一致。
- 开发者想当然地认为他们所添加的功能有用，事实并非如此。
- 需求发生改变。

软件对于客户用处不大

我们的软件没能帮助客户提高工作效率。事实上，客户对可用性的抱怨如潮水般涌来。软件的关键功能部分也不完善。这不是我们的错——需求规范是怎么说的，我们就是怎么做的。公平一点儿，这也不是我们客户的错。她告诉我们问题是怎样怎样。但在开始，当我们和客户商定需求时，双方都不知道该如何完全解决这个问题。等我们后来知道的时候，已经太晚了，因为我们已经对之前确定的需求做了不少工作，付出了时间和精力。

客户投入了资金，我们投入了时间和精力，系统开发出来了，可最终客户感到很沮丧——我们的软件对他们而言是一种负担，而不是一个有用的工具。

构建软件太过昂贵

软件开发的过程会花很多钱。要开发一个成功的项目，需要投入大量高薪的专业人员，耗费数月（有时甚至是数年）的时间，这些都是项目的成本。每一个软件开发项目所得的回报，并不一定总能验证我们对其的投入是值得的。很多商业机会正从我们这里流失，涌向成本极其低廉的海外开发（但这又会带来一系列该类型项目特有的严重问题）。

流程异味

流程异味是软件开发流程内部出现问题的症状，这些对客户是不可见的。但这类流程问题还是会间接影响业务价值，因为它们同样会对业务价值的交付带来负面影响.

流程异味通常比业务异味更容易发觉。但是，因为它们对业务价值不会产生直接影响，所以它们不应成为推动实施敏捷实践的主要原因。如果发现存在某些流程异味，请找到与其有对应的业务价值，以确认它们与组织最重要的业务价值之间的关系。

“我们”与“他们”的对立

“客户不知道他们想要什么！”“当我们需要时，这些程序员们从来没有给我们想要的东西！”“测试人员没有团队精神——他们不知道按时交付是多么重要！”“市场部门总是承诺我们做不到的东西！”上面这些闹情绪的话是不是听上去很熟悉？

软件开发涉及人员的多样性令人难以置信。如果出现问题时，大家相互指责，问题会更加恶化。每一个子团队，不管是开发人员、客户还是测试人员，会按团队自身的需要进行优化，而不是从整个组织所需要的业务价值角度来考虑问题。这会对组织的整体目标和成果带来灾难性影响。在任何层次上存在“我们”与“他们”的对决，都意味着沟通障碍的存在，而且没有考虑到组织的业务价值。无论实施的是敏捷还是传统方法，成功的团队都具有“团队整体”意识。

客户不管三七二十一，什么都要

客户与开发团队有时存在不信任关系。其实在现如今，典型的情况是，需求收集在项目前期

就完成了，并且将其放在正式签字的合同之中。任何新的需求变化，必须经过严密的变更管理流程，想要变更需求就必须面对非常高的门槛。最终产品也许满足了“字面上的”需求 ，但可能无法满足客户的真正需要。

客户知道这一点。由于他们清楚只可能有一次机会提出要求，所以他们会要求自己能想到的一切。这个异味表明我们没能交给客户他们真正需要的功能，而且我们没有给他们机会去学习、调整并且找到自己的真实需求。最后，所有的参与者都付出了巨大代价。由于对要开发的系统缺乏足够的反馈，诸如就会出现“构建软件太过昂贵”、“有些功能客户没有使用”这样的业务异味。

“客户？什么客户？” 无法实现直接、经常性的客户参与

场景 1：我们是一个产品公司，没有真正的客户在我们身边。我们的市场人员就是准客户。他们没跟我们坐在一起，而且也有自己的工作。他们不能(也不会)抽出时间参与到项目中，成为项目组一员。因此他们只跟经理们一起工作，而这些经理的下属们则和我们——开发团队——一起，开发正确的功能。

场景 2：我们的客户是企业中的业务人员；我们是他们的支持人员。他们没有时间跟我们一起工作。即使偶尔能挤出一点时间和我们在一起，我们还需要把他们说的记录下来。我们有他们的联系方式，可以自由地发邮件联系并定期举行会议。这已经足够好了。客户都是大忙人，而开发软件是我们的职责。

在这两种情况中，客户参与都很少。这种过程“异味”与下列业务“异味”有高度关联：“有些功能客户没有使用”、“软件对于客户用处不大”，或者“使用软件已经成为负担”。为了解决这些问题，经常性的客户参与和反馈是必要的。

感到意外的管理层——缺乏可见度

没有足够的可见度，管理层不能够了解项目开发的真实进展状况。开发团队总是很乐观，尽管有些功能已经出现问题，他们还是确信：他们在最后一分钟能够通过英雄式的努力能让一切就绪。不幸的是，哪些地方是潜在隐患，这样的具体细节不但管理层不知道，连开发团队自己都不是十分确定。再过几个月就要集成了，开发团队都清楚这将是痛苦的过程，但没有意识到到底会有多痛苦。

当真正的最后期限即将来临，团队不能回避无法满足最后期限这个事实，管理层再想做出有效针对措施，已经为时已晚。如果这种情况经常发生，那么管理层就会为项目组做出的任何

承诺留出缓冲量。缺乏信任的状况开始蔓延。

瓶颈资源 -- 项目成员同时属于多个团队

为了获得最好的软件质量，我们鼓励开发人员尽可能专攻某一方向，成为某一方面的专家。这种做法带来的副作用是：几乎总是同时有多个项目需要同一专长。在多个项目并进的过程中，一些关键的参与者成为瓶颈。想要调动一些成员到其他项目也十分困难。这就导致了组织成员同时被分配到多个开发团队的状况。这就是大型组织中司空见惯的劣根性，总是有很多项目需要完成——是吧？

大量研究表明，多任务的效率远远低于单任务。同时在多个项目工作，生产效率要低很多。如果上市时间和快速响应很重要的话，您必须设法解决这些瓶颈。

项目反复拖延

项目一拖再拖没能按时完成。一旦错过了第一个截止日期，就会有第二次、第三次。主要的设计决策没能预见到后来涌现的问题。因此进行多次尝试以期交付对客户有用、高质量的软件。有些时候尽管投入了大笔资金，项目还是会被迫中止。也有些时候项目会历经波折，直至最终才有一个可工作的系统建成。

Bug 跟踪系统中有数百（或上千）个 bug

当发现一个 bug 之后，会录入 bug 跟踪系统并且进行排序。在发布之前一般会修正影响发布功能和高优先级的 bug。任何低优先级的 bug 会一直停留在 bug 系统中。有的时候，在后续新版本开发过程中会修复一些中等优先级的 bug，但多数情况下到那时这些 bug 可能已经失效了。

在缺陷 bug 跟踪系统中存在大量的 bug 意味着工作的浪费。我们花费了精力去发现、定位并且识别这些 bug，但是在它们被解决、集成和发布之前，业务价值就没有得到交付。这种现象是一个直观的指示器，意味着我们投入了大量工作开发的功能，却没真正发布给用户，没能产生任何价值。

发布前需要一个“固化”阶段

在发布之前，需要有一阶段，在此期间不允许提交任何新功能代码；代码必须被冻结、分支

与严密测试；只能修复高优先级的 bug，而且必须经过预先批准。这一般要经过一个充足的时间，通常是 1 至 3 个迭代左右，才最终发布。

这是一个好的实践，对吧？为什么它在“异味”这一章？如果迭代很完美，也就是说在每一个迭代的后期，会展示一个可工作的、集成的、经过测试的系统——因此“固化”阶段没必要存在。这个“固化”阶段的存在表明我们的迭代不是真正意义上的迭代，仅仅是工作的时间段。迭代需要稳定期意味着之前的迭代出了问题，应该捕捉到 bug 没发现，也没有得到解决。

没能经常集成（通常是因为集成一次太痛苦了）

因为集成工作困难且费时，所以一般在发布之前的几个迭代完成。若干小组各司其职，分别完成应用程序的不同部分。为了保证发布时各部分能顺利集成，所以提前创建了文档和设计资料。然而事实是，他们很少能顺利集成。

对于大多数的开发团队来说，这是一种很自然的开发应用程序的方式。集成和测试整个软件系统看起来很美，但却不现实。很多团队埋头苦干手头工作，直到最后关头才去集成。然后发现实际上构建和链接竟然如此费劲，经常集成因为太花费时间，不可能经常进行。更频繁地集成会带来哪些好处呢？为什么这是一种异味？

缺乏集成会导致大批未经测试的代码。集成是反馈周期中关键的一环——如果没有完全的集成，就会掩盖大量的错误、沟通误会以及概念误解，直到发布的最后阶段才能发现。这阻碍了团队自我完善改进和判断项目进度的能力。

理论到实践：您能找到什么异味？

回答下列问题，发现、理解并排序组织中的不同异味：

1. 在组织中找到尽可能多的业务异味。一个好的起点是，和您的用户、客户支持人员或市场人员一起来找。他们知道哪里有问题。根据异味的重要性以及它们导致的问题将其排序。
2. 找到与这些异味关联的业务价值。对比在前一章中识别出的对客户来说很重要的业务价值，两者是否一致？
3. 查找尽可能多的流程异味，并将它们排序。把它们与业务价值相关联。
4. “异味”列表中的排列是否与业务价值的排列有出入？举例来说，最痛苦的“异味” 是

否与最重要的业务价值相关联？这又说明了什么问题？

5. 对您的组织来说，关注价值或者解决“异味”哪个更有效？为什么？
6. 根据前面得到的信息，对找出的“异味”重新排序。与之前的排序是否有区别？如果有的话，哪些发生了改变？

3

敏捷实施策略

至此您已阅读了“业务价值”和异味部分。相信您也已经参考每章节末的练习，对您的企业做了分析，得出两份列表：业务价值优先级列表、待修复异味优先级列表。如果还没有，建议您先停下来回去做做看。如果已经对客户的期望、公司的症结了如指掌，那您就可以回答这个问题了：如果想解决面临的问题，让我们的努力产生最大价值，我们应该采纳哪些敏捷实践呢？

在这一章中，首先我会给出一些指导方法，告诉您如何着手选择合适的敏捷实践。我会要求您评估当前现状，制定一个目标——即便它只是主观臆断——帮助您“敏捷”地实施敏捷。不过，这些都仅仅只是建议，您需要根据自身的需要，找出一些实践来进行尝试。请不要期望这里能有什么“敏捷攻略”——告诉您先实践 A，接着 B，不要做 C，……。（如果您在其他什么地方找到了所谓的秘籍，建议您不信为妙。）

敏捷模式到业务价值的映射

让我们从本章的核心说起。下面两个表格分别给出了提升业务价值和消除异味的敏捷实践及其组合。参考这两个表，来决定实施哪些实践。

表 1. 提升业务价值的敏捷实践

业务价值	敏捷实践组合	敏捷实践模式
缩短上市时间	测试驱动开发、演进式设计、测试驱动需求	简单设计、重构、测试先行开发、测试后行开发、持续集成、功能测试
增强产品的市场价值	测试驱动需求	功能测试
提高产品质量	测试驱动开发、测试驱动需求、演进式设计	测试先行开发、测试后行开发、重构、简单设计、持续集成

提高灵活性	演进式设计、测试驱动开发、测试驱动需求	自动化测试、重构、集体代码所有权、功能测试
增强可见度	测试驱动需求	功能测试、持续集成
降低成本	演进式设计、测试驱动开发、测试驱动需求	简单设计、重构、集体代码所有权、测试先行开发、测试后行开发、功能测试
延长产品寿命	测试驱动开发、演进式设计、测试驱动需求	重构、自动化测试、功能测试、简单设计

上表要按行来读。每一行代表着一种业务价值。“敏捷实践组合”一列包含了能提升该业务价值的实践，并且按照有效程度排序。因此，如果要提高产品质量，您应该首先考虑测试驱动开发实践组合。如果在您的环境中，采纳整个实践组合不太合适或者觉得这一步迈得太大，可先考虑从单个实践开始。

“敏捷实践模式”一列包含了有助于提高对应业务价值的敏捷实践，它们也按照有效性排序。举例来说，如果您想要“缩短上市时间”，您应该优先考虑采纳简单设计。

敏捷实践模式与异味的映射

表 2 消除异味的敏捷实践及其组合

异味	敏捷实践组合	敏捷实践模式
交付质量无法令客户接受	测试驱动开发、测试驱动需求、演进式设计	测试先行开发、测试后行开发、重构、简单设计、持续集成
交付新功能需要太长时间	测试驱动开发、演进式设计、测试驱动需求	简单设计、重构、测试先行开发、测试后行开发、持续集成、功能测试
用户不使用有些功能	测试驱动需求	功能测试
软件对用户来说用处不大	测试驱动需求	功能测试
软件开发成本太高	演进式设计、测试驱动开发、测试驱动需求	简单设计、重构、集体代码所有权、测试先行开发、测试后行开发、功能测试

“我们”与“他们”的对立	测试驱动需求	功能测试
客户不管三七二十一，什么都要	测试驱动需求	功能测试
客户？什么客户？！	测试驱动需求	无
管理层感到吃惊	测试驱动需求	功能测试
资源瓶颈		集体代码所有权
项目反复拖延	测试驱动开发、测试驱动需求	自动化测试、功能测试、持续集成
Bug 跟踪系统中有成百上千的 bug	测试驱动开发、测试驱动需求	自动化测试、功能测试、持续集成
需要“固化”阶段		持续集成
没有经常集成		持续集成

使用这个异味表格的方法和前述业务价值表完全一样。有些异味的实践组合为空，这表明实践组合可能不会带来非常显著的效果。需要记住一点，还有很多其他的实践，本书并没有涉及。这两个表只包含了本书提及的实践。

我已经替你准备好了拼图游戏的所有拼板，您要做的就是根据特定的环境，设计一个敏捷实施策略，接下来我们将讨论如何把它们有效地拼在一起。

以业务价值为中心

软件开发的目标是为客户提供价值。我们要始终记住这一点，并把它放在心上。如果没有办法接触到客户，那么您应该尽力去争取。要跟客户一起工作以期真正理解他们的需要。参考前面章节中的练习，与客户进行深入沟通。一旦得到了排好优先级的业务价值列表，要在团队里传播这些认识。让整个团队都知道这些信息。把它放在信息辐射器上，让整个开发团队随时都能看到并牢记在心。

目标驱动

我并不是无端地让您去花很大功夫去找出对您所在组织重要的业务价值和异味，并把它们做

优先级排序。现在您可以依据这些列表来确定采纳敏捷实践的目标。

在采纳一个实践的时候，思考您的目标，搞清楚为什么要这么做。是为了缩短上市时间么？还是为了提高产品质量？亦或是想去除“固化”迭代？

不要独自做决定。让客户参与进来，这样他会理解您为什么需要引入这些变化。做决策的时候要时刻想着其业务价值。市场部门并不关心是否要测试先行开发，他们只关心产品的缺陷是否会少一点。您需要让他们知道，采纳测试先行开发是为了减少缺陷，告诉他们可以在发布的下一个版本中看到成果。

让客户的需要来驱动敏捷的实施过程，并跟踪记录所取得的进步（即使是很主观的），您就会得到客户的支持。在回顾时，让他们参与进来，不去刻意掩盖自己的错误，他们或许会由此建立起对您的信任，与您一起齐心协作。当您交付改善后的成果后，您就会拥有一些狂热的追随者。

以迭代方式逐步实施

小步前进。从一个小团队开始，先积累些经验。大多数得到适当指导的小团队，都能很好地完成向敏捷的转型。边实践边学习。这本书中学到的会有些帮助，但只是一个开始。您需要去亲身体验，然后构建出自己的经验体系。

在一个敏捷实施项目成功之后，可以逐渐扩展到更多人，更多项目。交流本书中的知识，分享您的经验。周期性地检查业务价值和异味——它们会随着一些实践的成功实施而发生些变化。使用本书中的知识，这可以帮您发现新的异味并采取相应措施，从而不会掉入陷阱。

敏捷地实施

周期性地对实施过程进行回顾总结。根据这些回顾得到的反馈，调整一些实践，改进开发过程。放弃那些没起到作用的实践。引入一些新的实践，使对应的实践组合更加完整。

使用测试驱动方式制定敏捷实施策略

使用迄今为止您得到的关于业务价值和异味的信息，来决定哪些实践您应该考虑采纳：

1. 完全依据业务价值来选择实践

   这种情况下，您没有面临什么严重的困难，只是想通过提升业务价值的方式来改进软件开发过程。使用表 1 中所描述的业务价值和敏捷实践组合的映射关系，去选择那些最能

提升企业业务价值的实践。

2. 依据消除异味的效果来选取实践

   按其业务价值排定优先级。这种方式让您在不忽视业务价值的前提下，关注于减轻痛苦。根据客户对业务价值的在乎程度将异味进行优先级排序。再依据这个异味优先级列表，参考表 2 提供的异味和敏捷实践的映射关系，选择适用的实践。

针对表现最明显的异味，选择实践。这是常见做法，不过我不推荐使用。这种“救火”的做法十分痛苦——只是想着去除自身的最大的痛苦，根本不考虑能给客户带来多大的业务价值。在那些根本不考虑客户而独自决定优先级的技术团队中，这种情况太普遍了（我就经常为此心怀愧疚）。

在这一部分开头的表格中，各实践是按照其有效性排列的。因此，无论是提升某个业务价值或消除某种异味，每个图表中的第一个实践往往是最有效的一个，建议您先试试第一个实践，实践成功后，可回头再来看与您的业务价值、异味相关联的其它实践或组合。

不管您怎样去排列采纳实践的优先级，您都要尽可能以迭代的方式引入实践。准备好待引入实践列表后，采用如下方式能够保证您成功地引入列表中的敏捷实践：

1. 评估现状。仔细思量、评估您想要提升的业务价值、您想要消除的异味，哪怕是主观判断也没关系。
2. 设定一个目标。您想在多大程度上提升您的业务价值？想在多大程度上减少异味？需要花多长时间？可以初步做个预测，随着经验的增多，再逐步调整。
3. 努力实施您所创建的列表中的第一个实践或其组合。
4. 阅读该实践或其组合相关的敏捷模式。您分析您的实际情况与面临的问题，确定该实践是否真正适用（关于什么是敏捷模式以及之间有什么不同，请参考第二部分：敏捷模式，第 47 页）。如果该（组）实践对您的特定环境不适用，那就再重新从业务价值和异味列表，选择下一个实践。
5. 一旦确定了哪些模式在您的环境中适用，就去找出对应的章节，从头到尾仔细阅读。然后，参考敏捷模式中“采纳方法”部分的建议，开始实施。
6. 定期进行检查：目标业务价值得到改进了吗，打算消除的异味是否正在消除？如果不是，参考敏捷模式的“变化”与“但是”两个部分中的窍门，调整您的实践。
7. 返回步骤 1，重新评估您的业务价值和异味列表。如果还需要改进（也就是说，如果没达到步骤 2 中设定的目标），考虑引入另外一个、甚至一组实践来进一步解决您的问题。

如果您已达到目标，就可以设定新的目标。

那么这种方法中，哪些地方算是 “测试驱动” ？答案就是步骤 2 中设定的目标。在步骤 6 中，我们对实践实施的结果进行评估，判定已采纳的实践是否达到预定目标、效果如何。这样的循环过程——设定目标，实施某个或某组实践，然后对比预期目标进行验证——即构成了一个测试驱动的实施策略。[4]

从理论到实践：制定适用于您自己的敏捷实践实施策略

请回答下面问题以制定您的实施策略（使用“业务价值”和“异味”部分练习的答案）。也可参照附录“实施策略案例研究”部分遴选的案例，了解在真实环境中如何完成这项任务。

1. 实施敏捷实践的目标是什么？您是想清除一些异味还是增加业务价值吗？务必具体。如果您的目标不只一个，那就需要设定优先级。
2. 评估您期望改进的业务价值或要消除的异味。不必担心他们是否过于主观或模糊。尽可能地去了解您的组织今天的业务价值以及存在的异味。
3. 选择一个实施策略。利用该策略选择要采纳的实践。
4. 阅读下一章的模式介绍。然后按照本章中列出的步骤，实施您的第一个实践。不要忘记进行周期性地评估，确保您的实践的确有效。
5. 恭喜您已经踏上了敏捷实施之旅！祝您好运！

[4] 在管理实践中，这种方法通常被称之为 PDCA 循环（计划，执行，检查，调整），最初由贝尔实验室的 Walter Shewhart 在二十世纪三十年代提出，之后在五十年代由质量管理大师 W.Edward Deming 进一步推广而变得广为人知。

第二部分：模式

4

简介

什么是模式？

通常来讲，每个模式会描述一个特定的问题以及对应的解决方案。在本书中，模式用来专门描述软件开发团队中存在的一系列问题，及应用敏捷实践来解决这些问题的方法。因为每个模式都被开发团队在实际项目中反复使用，所以它们是值得人们信赖的，并不是那种靠碰运气的“一次性”解决方案或所谓的“好主意”。这些模式不是我们“创造”出来的，只是被我们发现了而已。

本书中对模式的阐述都遵循如下格式：

> 名称
>
> 描述：对模式或者模式组合的概述。
>
> {依赖关系图}：一幅展示对应实践依赖关系（针对实践）和分类（针对组合）的图。
>
> 业务价值：针对实践方法或实践组合的业务价值，分类进行描述。
>
> 小故事：利用一个虚构的故事，描述该模式如何应用于具体的软件开发项目。
>
> 适用情况：说明在什么样的前提和环境下，该模式才能发挥功效。适用情况包括一组约束条件，这些约束条件本身不会因为应用了模式而发生改变。
>
> 动因：进一步地描述适用情况，并给出该模式（部分)解决的具体问题。实际上，如果成功的应用了该模式，应当可以消除很多这样的问题。
>
> 因此：该模式的描述。
>
> 采纳方法：采纳该模式的步骤、顺序和指导。
>
> 但是：应用该模式可能带来的负面效果。
>
> {变化}：虽不同于因此部分中的做法，但能成功实施该实践的一些不同的做法。
>
> {参考资料}：更多的阅读材料。

名称、描述和依赖关系图这三个部分展示了该模式的概貌。您可以浏览模式描述以及关系依赖图部分，了解不同的模式及其相互关系。您通过阅读小故事部分，可以了解到在实际中如何应用该模式的一幅全景。

如果您正在考虑，对于自己的情况，是否能合适去应用某个模式，那么适用情况这一部分就是为您准备的。它包括了该模式必须的所有前提条件和保证该模式有效的环境因素。如果环境因素不匹配，那么实施该模式可能不会太有效。

动因部分描述了本模式代表的方案所要解决的问题。与适用环境相似，这一部分可以帮您判断是否采纳这种实践方法。如果在项目中找到了对应的阻力，那这就说明该模式很可能会产生积极的影响，能够帮您解决相应的问题。

接下来的一部分是因此，也就是解决方案——实际上它是该实践的描述。这些信息可以帮您理解该实践方法及其细节。但要记住，本书不是指导手册。如果不知道这个实践方法是什么，您可能需要找一些别的资源来进一步地研讨该实践方法。

接下来的三个章节对于真正采纳该实践方法很有帮助。采纳方法部分教给您如何利用一种渐进策略去成功地应用该模式。但是部分会告诉您在应用该模式的时候，可能会发生哪些问题。接着变化部分描述一些不合常规但曾被成功应用过的实施方法。您可利用这三个部分的指导和帮助，循序渐进地前进，然后做到因此小节中所描述的实施方式。

您可以通过参考资料部分了解到有哪些资料也阐述了该实践。我没有把所有参考资料一起放在书中的文献目录中，而是对于每个模式，指出从哪里可以获得更多的阅读材料。

有效地利用模式

有很多种阅读模式的方法。根据具体情况，下面列出了一些模式的应用方法：

- 我已经在实践这个模式了，没什么大问题。我只是想了解下别人是怎么使用这一模式的。
  - 按名字查找这个模式。
  - 阅读适用情况，了解下您是否和别人处于相同的境况。
  - 阅读因此和变化部分，看看您所使用的方式是不是和其中的一些相类似。
- 我在实践一个模式，但看起来不是很有效。是不是我使用该模式的方法不正确？或者这个模式在我的环境不适用？
  - 按名字查找这个模式。

- 阅读适用情况部分。如果您的环境与其不符，或许您该考虑调整一下所用的实践或者干脆不用它。
- 阅读动因部分。您在尝试解决同样类型的问题吗？如果不是，需要斟酌下。或许这个实践有用但您还需要其他实践来解决您真正的问题。
- 看看但是部分。您会了解到别人是在这儿如何犯错，如何获取建议纠正错误，并最终从中充分获益的。

- 在我的团队中有些问题，我想要采纳敏捷实践来解决这些问题。
  - 回到第四章异味部分，找着看您的问题与哪些异味相符。
  - 阅读能够解决这些异味的实践。
  - 对于每一个实践：
    - 阅读适用情况，确保它适用于您的情况。
    - 阅读其余部分。
    - 如果您决定采纳该实践，按照采纳方法部分的建议去做。
    - 做周期性检查，看看是否有但是部分所描述的问题发生。
- 在异味一章中，找不到我想要解决的问题。那是否意味着所有的实践都帮不上忙？
  - 不是。阅读每一个模式中的动因部分，看看其中是否有相似的问题。您很有可能会找到一个。
- 我们在采纳一个特定的实践。我们成功了吗？我们的采纳方法是否达到极致，完全发挥了它的作用？
  - 按名字查到这个模式。
  - 查看动因。是否还有哪个问题在您的团队中依旧存在？
  - 查看但是部分。是否存在其中所描述的某个异味？如果是，解决它们。
  - 如果以上任一情况都已经不适用，那您可能已经不用读这本书了。您或许已经有足够的经验及直觉，可以灵活地根据自己的需要，选择适当的模式。可喜可贺！

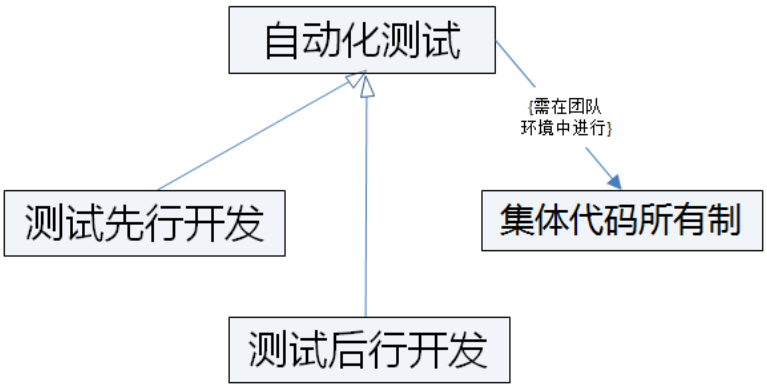
最后，毫无保留的相信这些模式。使用模式这种方式，你可以很方便地打造出自己的方案。所有的这些模式，都来自于许多项目的实践。在某些领域里，它们的确经过了反复的验证。但，银弹是不存在的。在某些情况下，这些模式很可能就不合适。把这些模式当作指导，当

实际情况与理论背道而驰时——实事求是地解决问题。

5

自动化测试[5]（抽象模式）

为了降低 Bug 识别与修复的成本，让设计能够随需求的变化逐步演进，开发者编写并维护了一组测试，代码质量也因此得以提升。严格地编写测试也促使设计不断趋向松耦合。



业务价值

自动化测试使得我们能够在软件开发过程中尽早捕获 Bug，从而提高了产品质量。由这些测试构成的“安全网”再加上重构，增强了灵活性，延长了产品寿命。上述价值显而易见，但不那么明显的是：自动化测试实际上缩短了开发时间，从而也缩短了推向市场的时间，节约了开发成本。

小故事

第三次发布刚开始的时候，WaterfallWill[6]和Uthman UpfrontDesgin加入了Scott

[5] 原文为 Automated Developer Tests，即由开发人员编写的自动化测试，中文翻译版本为描述方便，简略为“自动化测试”。——译注

[6] 文中的人名都显示了这些人的一些背景，例如 Uthman UpfrontDesign 就表示 Uthman 曾经习惯于预先设计，Will Waterfall 就表示 Will 曾经习惯于瀑布模型的工作方式，Scott ScrumMaster 就说明 Scott 是一个 Scrum Master，下文的 Aparna Analyst 也表明她的角色是一个分析。——译注

ScrumMaster的敏捷开发团队。Scott团队已经完成了两次非常成功的发布，团队中的一些开发者还跑去别的团队“散播敏捷病毒”。在此之前Will和Uthman所在的团队使用传统的开发模式，都是由QA完成大部分测试，而开发者只根据临时需要才会编写特定的测试。

当时，测试是团队严格遵守的一项纪律。Waterfall 和 Uthman 加入团队后，和其他成员一起结对编程。有些成员像 Cindy Coder 那样，使用测试先行开发，先写测试。其他一些人，比如 Dave Developer，常常在完成一些功能代码之后才编写单元测试。但是在提交代码之前，他们都会确保完成了相关的测试代码。

Scott 团队是一个自组织的团队，所以不会强迫大家编写测试，而是为了保证消除“子团队”，保证每个开发者都有权修改代码库的任一部分，号召和鼓励大家为所有代码写测试。

当 Uthman 和 Will 开始开发票据子系统的时候，他们一时冲动犯了一个错误。他们一起结对，增量式地开发了这个子系统（很不幸，它没有任何测试），他们非常得意于自己的设计和功能的灵活性。Aparna Analyst 在进行了（手动）测试后，也对他们新增的功能非常满意，并验收了这些功能。在那之后，Dave 和 Cindy 一起修改了“收费”对象的行为（票据子系统严重依赖于该对象）。他们成功地提交了这些代码，系统看似也没什么问题。但当 Aparna 在准备下一个迭代时，她发现票据子系统不工作了。Will 和 Uthman 发现了问题，立刻找到 Cindy 和 Dave，质问他们“你们做了这么大的更改，为什么不告诉我们？！”Dave 和 Cindy 反问道，“你们的测试呢？我们依赖测试来告诉自己是否有功能遭到了破坏！”

这是自动化测试实施过程中，令人恼火的教训之一。接下来的迭代中，Will 和 Uthman 发挥了重要作用：为使新添加的收费代码能够正常工作，他们花了很大工夫重写了原来的票据代码，并添加了测试。

适用情况

这个模式在很多情况下都能发挥作用。无论碰到下列哪一种情形，甚至所有情形，您都将从该实践中获益：

- 您的开发项目需要显著提高产品质量——例如，减少 Bug 数量。
- 您的开发团队已决定采纳迭代和简单设计，并且需要考虑如何根据新的需求来演化设计。
- 您的开发团队是一个分布式团队，成员不在同一个地方同时开发。由于缺少面对面的交流和持续的反馈，导致 Bug 的数量不断增加。
- 您的开发团队正在实施集体代码所有权，并不是每个人都能理解所有代码，但每个人都有可能要修改系统中的各个部分。这时您需要其它实践配合来解决这个问题。

动因

- 仅仅依赖 QA 测试来发现和修复提交的代码里面的 bug 会极大地增加项目成本。
  - 最明显的一点是，QA 要和开发者一起去发现 bug；并且需要通过一个工具沟通来记录完成的工作。
  - 在很多情况下，在 bug 未能清晰重现之前，bug 会像皮球那样在 QA 和开发者之间踢来踢去。
  - 发现、确认和修复 bug 通常非常耗时，如果开发者能在提交代码之前就发现并解决缺陷，花费的时间会减少至少一个数量级。到那时候其他开发者已经签出了这部分错误代码，并在其基础上进行了开发。
- 对一个 bug 的修复经常会导致新的缺陷。有时候，甚至会产生连锁反应，没完没了。
- 系统中较复杂的部分往往存在更多的 bug。正因为代码不容易看懂，所以 bug 很容易反复出现。
- 为了让系统能够灵活地适应需求变更，系统设计往往追求通用性。不幸的是，这种额外的灵活性并不是免费的——过度的系统复杂性也会导致一定的成本。每次开发人员面对一段复杂的代码，都需要花费很多的时间去读懂，并学会正确使用。这就是“设计的累计开销”。
- 有时对一部分的更改会破坏所有依赖于它的各个部分，但由于对系统设计的修改太过昂贵，因此就会做一些“创可贴”式的修补。久而久之，会导致代码重复、设计变得难以维护，代码维护性也变得很差。在发布前，要花很长时间才能通过 QA 测试，而且即使这样还是会有问题遗漏。由于害怕出错，我们尽量将修改产生的影响降到最小。

因此

为了减少查找和修复 bug 的工作量，应该让开发者更严格地测试自己的代码以期及早发现 bug。尽量让测试自动化，并且让开发者能够访问到所有的测试代码，这样在完成一些代码改动后，如果导致了其他部分的测试失败，他们就可以及时发现。引入这样一种实践：在提交代码到代码库之前，保证所有测试都先运行通过。为了帮助大家理解，对复杂的设计撰写文档；同时确保在系统更新的时候，该文档也得到同步更新。最好的方法就是编写“可执行的文档”（比如：编写良好的测试）。最后，每次发现 bug 后，编写一个测试来重现它，并将其加入测试套件，然后修复，并提交实现和测试代码。如此以来，您就可以确保该 bug

再也不会重复出现。因为每当有人改动代码再次导致该 bug 时，测试就会失败，大家只得先修正错误然后才能提交。

在一个团队开发环境中，只要您写了代码，就有可能会导致既有测试的失败。说到底，测试最大的好处之一就是作为一个安全网：当您的改动不符合系统其他部分的设想时，它就会给出警告。必须记住，只有在所有测试都通过之后，才能提交代码——这样您就不得不修改系统中被影响的部分来保证测试全部通过。可以借助集体代码所有权和结对编程解决该问题。

引入自动化测试，把测试组织成不同的测试包，快速有效运行，这样您就可以解决动因部分提到的问题。一旦编写测试成为一种习惯，开发者在如何对待问题和解决问题上，就会发生一些变化——由于测试能及时帮助大家发现自己的错误，大家在对设计进行改动时，变得十分有信心。充满勇气。因此，对于所有编写的代码，必须及时编写良好的测试，否则就会像前面小故事中的 Will 与 Uthman，遭遇同样的尴尬。

什么是良好的测试？这是一个模糊不清的问题，跟“什么是好的代码”一样很难回答。最好像对待其他代码一样对待测试——而不是作为二等公民——所以，任何您所知道的关于代码的设计方法，也同样适用于测试。最好的学习方法就是边做边学：阅读其他人编写的测试，密切关注所发生的问题，然后调整编写测试的技巧，以避免那些问题再次发生。

采纳方法

我们不会具体介绍如何编写开发测试——有很多其它的书籍已经涵盖这部分的细节。相反，我们会介绍您应该采取的实施步骤，从而最大可能地保证让您和团队成功实施该实践：

1. 作为一个团队承诺编写测试，把它作为一项团队纪律。
    a. 要意识到这主要是人的问题而不是工具的问题。
    b. 测试和产品代码同等重要，团队要对此达成共识。
    c. 团队要认同这一点：即使少了一个功能特性，也比只完成功能却没有测试要好。
    d. 所有人都要具有耐心。视不同项目而定，想要让自动化测试成为习惯并从中真正获益，有可能要花 2 到 6 个月的时间。
2. 找一个易于使用的工具。测试工具应该保证大家能够不用花太多功夫就可以完成一个测试，而不是让您觉得每次写测试都是不得已而为之。
    a. Java 平台可以使用 JUnit 和 TestNG。.NET 平台可使用 NUnit 和 Visual Studio 的内嵌测试工具。CXXTest 和 CPPUnit 适用于 C 或 C++平台。

b. 不要完全依赖于自动化测试工具，而应仅将其当作辅助工具。如果你过分依赖自动生成测试，您就没有机会去思考：怎样能让代码更“可测”？怎样才能使代码耦合度更低，设计更加优雅。

3. 要象对待产品代码一样对待测试代码。测试也应该有好的设计。

   尽可能获得更多的帮助：

   a. 如果可以，找一两个有经验的咨询师加入团队。
   b. 从公司里找到那些曾经成功在其他项目上使用测试先行开发或者测试后行开发的同事，争取他们的帮助。
   c. 多买几本关于测试驱动开发和 xUnit 测试的书籍。（在本章的结尾处有推荐列表）鼓励团队成员抽些时间进行阅读。
   d. 加入专注于测试驱动开发、敏捷开发等内容的在线社区，本地用户组等。
   e. 起初不要担心 Mock 对象和纯单元测试。(即使还没有了解这些概念也没有必要担心，现阶段它们并不重要。)最初，先为每个类的重要方法编写测试。没有必要为简单的 getter 和 setter 方法编写测试。

4. 在团队中行使集体代码所有权。这样您就可以在测试失败的时候立即作出修正。

5. 为了缩短团队的学习周期，考虑利用结对编程作为辅助实践。当与另外一个人一起工作时，更容易保证编写测试的团队纪律。

6. 从当前迭代开始编写测试。如果是一个新的项目，那么预计速度会下降 50%。如果项目已经有许多未经测试的代码，那么速度可能会下降更多。坚持一段时间以后您花费在测试上的时间可能会降到仅占整个开发投入的 20%到 30%。最终您会越过一个临界点，到那时已有测试会对新代码的编写提供很大帮助，会加快您的开发速度。信不信由你，即使有测试开销，开发速度还是会变得越来越快。

7. 在几次迭代之后，团队将开始为准备测试数据发愁。因为编写的对象依赖于其他对象，这些对象又依赖于更多对象，需要写越来越多的代码来为测试准备。解决这种问题有两种方法：

   a. 把一些通用的数据准备代码抽取出来，放到通用的类中。这些类是一种特殊的对象工厂，它们负责创建类和准备测试数据。他们能够创建特定状态的业务对象。Martin Fowler 在 XPUniverse 2001 大会上提出了最初的 ObjectMother 概念 http://www.martinfowler.com/bliki/ObjectMother.html。ObjectMother 是复杂测试准备代码的一种常见演化。您的测试总是用来验证真的业务对象（这是

一件好事）。不过，由于可能要支持的特殊情况太多，ObjectMother 很快就会变成一个维护负担，它很容易变得不够灵活。由于每个测试对许多业务对象有依赖，基于这种方案的测试会变得非常难以维护。

b. 使用 Mocks 和 Stubs 来避免复杂的 ObjectMother。Mocks 和 Stubs 在被测业务对象的作为占位符。它们可以用来切断一个对象对其他多个对象的联系，从来达到单独测试这个对象的目的。jMock 测试框架小组曾经写过一篇很好的关于 Mocks 和 Stubs 的文章《模拟角色 ,而不是对象"Mock Roles ,Not Objects" 》。Mocks 可以让您的测试变得易于阅读也更加健壮。不好的一面是，Mocks 会导致逻辑重复——一个像样的 Mock 对象是对应业务对象的镜像。因此就会带来所有由于代码重复导致的问题：如果业务对象改变了，那么一个 Mock 也必须改变；如果没有对它进行修改，即使应当失败的测试还是会成功。

Mocks 和 ObjectMother，两种方法其实都不错。关键在于一致性：要达成团队共识使用同一种方法。这会让团队成员能够很容易地接手其他人的代码。

使用 Mock 对象和 Stubs 来测试那些需要与外界系统交互的类。

如果您对这种开发形式不熟悉，那么请从 ObjectMother 开始，避免一次尝试太多的新工具。

在团队完全适应了自动化测试之后，就可以开始尝试使用 Mock 对象。

但是

刚开始尝试该实践时很容易犯的一些错误：

- 该实践很容易失败：它需要组内的所有成员共同努力。
  - 如果有有人导致测试不通过，但不去修复，那么其他人也很容易犯同样的错误。通常类似于“这块不是我写的”或“我晚点再修复它”这样的借口是自动化测试实施失败的开始。
  - 开发者必须习惯于及时修复失败的测试，哪怕这些测试不是自己写的。这也就意味着，他们将需要维护系统中那些从来没有接触过的部分，而不是制造一个 bug，然后继续干自己的。
- 即使编写测试也绝对说明不了产品代码的质量没有问题。无论使用任何语言和技术都可能出现设计很差劲的代码。测试会促使大家写出松耦合的代码，而训练有素的开发者也

会利用这种技术来写出好的代码。如果在这种情况下，还是会写出糟糕的代码，如果你有这样的问题，应该考虑使用结对编程或定期对测试代码进行代码评审。

- 如果有时候把测试作为二等公民来对待：我们打破了优秀设计的准则。不可避免的是测试逐渐变得越来越难于维护，越来越难以编写测试。把产品代码和测试代码同等看待。当设计不能满足需要时，应该对其进行重构。要留心耦合和内聚，以及其他您所了解和实践的准则。
- 编写测试——尤其是对那些没有测试的已有系统来编写测试，是很困难的。不要放弃。找出如何逐步增加测试的方法。做好准备在开发速度大幅提升之前，开发速度会有显著的降低。买本 Michael Feathers 的《修改代码的艺术》，从中寻找如何实施的建议。
- 所有的测试应该随时运行并通过——这没得商量。有时候，开发团队提交的代码会导致一个测试失败，这个测试没有被立即修复（我们过一段时间再回来修复它）。那在接下来仅仅几个迭代中，像滚雪球一样，这个失败的测试会变成 10 个，100 个，甚至 400 个。这是无法接受的。您已经丧失了这类测试的一个主要优势——及早地捕获 bug，避免在错误的代码基础上引入更多 bug。您的团队也会对此越来越麻木。必须立刻解决：将失败的测试移到单独的测试套件中；强制/说服/请求您的团队不要再让更多的测试失败。只要有测试失败，都必须立即强制将代码回滚。然后逐步地修复失败的测试，并将它们移回到正常工作的测试套件中；最后再将它们放入持续构建的测试套件中。
- 测试覆盖率变成了一种极其重要的衡量工具。管理人员利用测试覆盖率来进行决策驱动。尽管利用测试覆盖率来表明哪些区域的代码需要更加注意，这是合理的，如果用其来驱动开发就有问题了。这会（而且经常） 变成“数字游戏” ：
  - 事实上，是否在测试中调用某一方法与测试本身的质量丝毫没有关系。测试覆盖率的数据通常被误用为“测试质量”的指标——事实上不是的。
  - 如果我们实际编写的测试是用来验证代码是否吻合需求，测试和类里面的方法之间就不是一对一的，而是多对多的。而测试覆盖率能够保证一对一形式的测试，确保您有测试来运行该方法。

变形

有两种形式的自动化测试：测试先行开发和测试后行开发。它们方式不尽相同，但都能充分解决动因部分的问题。这种抽象的模式的适用条件和动因对两种方式来说都适用。这里会有相当一部分内容重复。

作为使用集体代码所有权来共享代码的替代，有些团队会使用结对编程。有些成员有特殊的专长，让每个人都去学习并具备修改所有代码并不现实。他们的解决方案是更多依赖结对编程，并创造鼓励结对编程的团队文化。但这也会导致一个明显的问题：为团队中的瓶颈资源额外添加了更多的任务。要解决这种问题，要么是增加人手，要么是放弃对结对的过分依赖，充分利用集体代码所有权。

该实践在敏捷社区里也被称为自动化单元测试。我没用“单元”而改用“开发人员”一词的原因在于，对于这些测试是否是真正的单元测试（一次验证一个类）还有争论。但这对于是否采纳这个实践并不重要。实际上，在还没掌握这种实践之前，不写单元测试来得更容易些。等到获取了足够的信息，您就可以独立判断是否编写真正的单元测试。

参考资料

自动化测试在测试驱动开发以及其他的一些专门为 JUnit（Java 中领先的测试工具）写的书都有相关的讨论。下面列出的 Michael Feather 的书，介绍了对已有系统的测试和测试驱动开发。

- 《测试驱动开发：实用指南》（美）艾斯特尔斯（Astels,D.） 著 中国电力出版社出版。
- 《测试驱动开发》[美]Kent Beck 著 中国电力出版社出版。
- 《修改代码的艺术》[美]Michael C.Feathers 人民邮电出版社出版
- Jeffries, Ron. 2004. Extreme Programming Adventures in C#. Redmond, WA: Microsoft Press.
- 《Junit in Action 中文版》 Massol, V.，电子工业出版社，鲍志云译，2005
- Rainsberger, J.B.《Junit Recipes 中文版》，电子工业出版社，李笑，陈昊，王耀伟译，2006

6

测试后行开发（实现自动化测试）

为了完成某个特定任务，根据需求，在开发完工作代码后再编写测试，这就是测试后行开发。系统构建完成之后，可以运行这些测试以对系统进行验证。

业务价值

测试后行开发与自动化测试所强调的业务价值相同，体现在产品质量、灵活性、产品生命周期、上市时间和成本这几个方面。

小故事

当 Uthman UpfrontDesign 和 Will Waterfall，一起加入了 Scott ScrumMaster[译]的小组，他们都乐意与小组中的其他成员进行结对编程，并尽最大的努力去学习小组的已经采用的开发实践。其中之一就是为每一段代码编写自动化测试。

与小组其他成员结对编程几个迭代之后，Uthman 和 Will 开始结对实现帐单子系统。由于初次接触自动化测试，所以他们决定在写一些代码之后再写测试（测试后行开发）。他们先设计，后编码，最后再将代码以增量方式提交到帐单子系统中。但是提交的代码不包含任何测试，因为此时此刻已经完成了编码（而且时间已接近迭代周期的尾声），，他们省略了测试，直接宣称任务已经完成了，尽管实际上他们并没有完成。在下一个迭代周期，他们接受了更多帐单子系统的任务，他们一直开发到最后一刻，测试再次被省略了。（无论怎样他们开发的是一个独立的系统，因此即使没有测试的也不会产生不良影响。只要作为子系统仍旧正常工作，就没有问题。）

后来，开发者 Dave 和 Cindy 对帐单子系统所依赖的系统的一部分做了修改，为了验证代码的正确性，他们运行了所有的测试，看起来都通过了。但实际上，帐单子系统却悄悄地不工作了。Aparna Analyst 在为下一个迭代周期准备订单子系统的需求时，发现了这个问题，并告诉了 Will 和 Uthman。毫无疑问，他俩很不高兴，并质问 Dave 和 Cindy 为什么不小心一点儿。Dave 和 Cindy 回答他们：“我们运行了所有的测试，而且都通过了，我们怎么

可能知道你们的帐单系统会停止工作？”

为了让系统恢复工作，Will 和 Uthman 在下一个迭代中，花费了很多的时间对代码进行了必要的修改，补上早就应该增加的测试。好的一面是，他们终于开始学会了编写测试的习惯（或规矩）。为了避免同样的错误再次发生，他们以增量的方式编写测试，也就是在每一步的开发结束后，都会给刚写的代码编写测试。

适用情况

如果您所在的的团队决定实施自动化测试，那么该模式就适用。再者，如果大部分（或所有）团队成员都没有测试先行开发的经验，并希望采用一种与以前所习惯的开发模式相近的开发实践，那么测试后行开发模式就更加合适。

或者，您所在的公司购买了一种可以生成测试代码的工具，这种情况也是适用的。不过这样的工具只能对已编写的代码生成测试。

动因

所有在“自动化测试”模式中罗列的动因，都适用于这种开发方式，另外还包括：

- 与在编写产品代码之前编写测试相比，在已有代码的基础上编写测试，学习曲线要小。

因此

编写产品代码的时候要小步前进。在每个小步后，使用选择的工具编写测试，然后验证编写的代码。将自己编写的和其他人编写的测试整理成测试套件，这样便于分组运行所有的测试。绝不要将任何没有经过全面测试的代码提交到代码库中。在提交到代码库之前，运行所有的测试，以保证改动的部分不会让任何他人编写的测试无法通过。

这种开发方式不仅限于测试，更与所实现的产品代码息息相关。它使得基于设计而诞生的产品代码更容易测试。相比以前不要求写测试而编写出的代码，这种方式会迫使开发人员时刻考虑系统可测性，从而大大降低代码中的耦合性。以该方式编写的代码和测试，会减少变更所产生的开销。设计也更易于修改，而不像某些系统的代码，因为害怕引入更多 bug，不敢大刀阔斧地改，只能采取权宜之计，到处贴补丁。

采纳方法

针对父模式自动化测试的实施策略，足够用来应对测试后行开发。但要注意的是，它没有测试先行开发高效，但由于它没有颠覆之前的开发行为，所以比较容易被接受。因此很多团队用它来作为走向测试先行开发的铺垫，从而可以感受测试的纪律性。

但是

这种开发方法并不是刚刚诞生的。实际上，在极限编程技术流行起来之前，这种编写测试的开发方法就一直被使用。除了自动化测试中所有的“但是”部分，如下是测试后行开发所独有的：

- 在项目关键时刻放弃测试。这是测试先行开发和测试后行开发所具有的共同问题。这种问题对后者更普遍，因为在测试被编写之前，代码总是貌似“完成”了。小故事中描述的 Will 和 Athman 的做法是司空见惯的。
- 测试被认为是无谓开销，有时这个实践被全部弃用。
- 测试过分针对具体实现方案。开发人员所编写的代码仅仅是针对需求中提出问题的一种解决方案。测试应该从逻辑上保证代码是符合需求，但通常它们只符合了代码自己的逻辑要求，只对代码自身的逻辑进行了验证。

参考资料

- Dave Astels 的《测试驱动开发》一书极力倡导测试后行开发。
- Astels, D., 2003. Test-Driven Development: A Practical Guide, Upper Saddle River, New Jersey: Prentice Hall, 2004.
- Feathers, M.《修改代码的艺术》，人民邮电出版社，刘未鹏译，2007
- Massol, V.，《Junit in Action 中文版》，电子工业出版社，鲍志云译，2005
- Rainsberger, J.B.《Junit Recipes 中文版》，电子工业出版社，李笑，陈昊，王耀伟译，2006

7

测试先行开发（实现自动化测试）

测试先行开发是在先编写测试，然后开发产品代码，并最终运行通过预先编写的测试。由于该实践中没有早期设计，所以测试就反映了开发者对需求的理解。

业务价值

测试先行开发与自动化测试所强调的业务价值是相同的，体现在质量、灵活性、产品生命周期、上市时间和成本这几个方面。一般而言，尽管测试先行开发和测试后行开发强调了相同的业务价值，不过测试先行开发更有效，但也更难于实施。

小故事

Uthman UpfrontDesign 和 Will Waterfall 今年早些时候加入了 Scrum ScrumMaster 的开发团队。他们与小组其他队员一起结对编程，找到了自动化测试开发的感觉，而且切身体会到在敏捷团队中，如果不写测试会发生什么样的事情。（请查看关于自动化测试和测试后行开发的小故事。）

Will 和 Uthman 在修改了帐单子系统的代码，补上了应有的测试之后，决定在下一个迭代中一起继续实现该系统的几个任务。不过他们决定先写测试。虽然曾与其他队员在其他任务上尝试过这样做，但是他们却从来没有单靠自己完成过，因此遇到了不少困难。在没写代码的情况下，就编写测试，是很棘手的。后来，他们利用纸和白板详细讨论了一个设计方案，评估了静态和动态的结构，接着针对脑海中的虚拟方案，编写了一个测试。在迭代结束时，他们提交了一小段经过测试的产品代码，可如此过程让他们觉得很不开心。

在后续的几个迭代中，Will 重新使用测试后行开发，并且严格对照产品代码来逐步增加测试。另一方面，Uthman 有点感觉到了测试后行开发的效力，他决定开始与具有该实践经验的人尽可能多结对。他阅读了 Kent Beck 关于 TDD 的书，并完成了里面的练习。因为看到许多值得崇拜的人都只使用这种开发方法，他也打消了自己先前的怀疑。同时，他也学到了利用 Mock Roles、Not Objects 来模拟对象，这使得他可以利用 JMock 高效地编写测试。

经过了许多个缓慢的迭代过程，Uthman 不仅开窍了，而且被深深吸引。

接下来，Uthman 和 Will 要结对完成一系列任务，当 Uthman 处于主导地位时，他用测试先行的方式来引导开发，并向 Will 说明了自己的做法。而当 Will 处于结对的主导地位时，他采取测试后行开发方式。Uthman 感觉到：对比两种不同方式产生的代码，使用测试后行开发的测试代码质量明显高于测试后行开发的，不过他保留了自己的想法。

适用情况

你的开发团队具备使用自动化测试模式的环境，而且：

- 你希望通过编写测试来获得更多价值
  - 你希望提升开发速度
  - 你希望通过编写测试来获得并提倡松耦合设计
  - 你希望使用测试覆盖所有需求，而不是用测试覆盖设计
- 可能你已经实施了测试后行开发，而且发现人们不是总能完成测试代码——在项目时间紧张时尤其严重。不幸的是，这通常是你最需要测试的时候。
- 实践该方法，团队需要经历一段非常艰难的阶段（通常 1-3 个月），才能做到习惯成自然，对此大家有足够的心理准备。

动因

测试先行开发能解决所有在自动化测试开发中所遇到的问题。如下的一些问题也能由测试先行开发解决：

- 在编码完成之后再编写测试，那么很有可能，测试永远被遗忘了。在时间紧迫的情况下，开发者很容易而且经常跳过为当前任务编写测试的环节，而直接开始开发下一个任务。
- 在编码完成后再编写测试，这样的测试对于代码的驱动是间接的。当开发者在写产品代码时，总是惦记着这些代码要能够测试。在这样的情况下，设计就会受制于要编写的测试。
- 测试应当用于验证针对实际问题的解决方案，但是在编码之后编写的测试，通常会偏于去验证已完成的代码。也就是说，测试应当验证产品代码是否满足了需求，而不是产品代码本身。当先写产品代码时，开发者不一定解决了对应的问题。基于这种情况再编写

的测试，只是验证了已写的产品代码，也就仅仅验证了产品代码是否象开发者所想的那样工作而已，而真正的需求却可能没有被测试到。

- 编码后编写的测试，不一定测试了所有相关的需求。

因此

在编写产品代码之前编写测试，可以更好地实现需求。在这种情况下，因为还没有为解决问题而编写任何代码，所以你能获得的唯一信息就是需求。如果强迫自己来编写测试，就需要决定哪些类能够用于实现功能，什么样的设计能完成当前的需求。因此，测试就会反映这些需求，并在形式上成为了一种可执行的需求。当然，这些测试可能会运行失败，而且很有可能无法通过编译，但这恰恰是我们所期待的。

先写完测试之后，接下来你需要编写产品代码来通过刚才的测试。由于已经决定了赋予哪些类什么样的职责，所以现在就只需要写些代码来让测试通过。之前编写的测试会驱动你创建类、方法以及与系统中其他类的关联（其实这就是面向对象的设计）。

一旦测试通过了，那么你现在不但拥有了通过的测试，还开发了满足需求的代码，而且没有编写任何一行跟需求无关的代码。如果严格遵循“没有失败的测试做驱动，就永远不要写产品代码”的原则，那么代码就会完全地覆盖需求。

最后一个步骤就是重构——就是在不改变系统行为的前提下，改进代码的结构。由于有测试作保障，所以任何修改，只要能保证测试通过，系统功能就不会受影响。不要增加新的功能，但请大胆地改进代码，清除马马虎虎的代码，让产品代码变得更高效、更易于维护。在清理完代码后，要运行现有测试以保证系统的行为依然正确。

刚才所讲的是测试驱动开发模式[7]中所描述的“红—绿—重构”循环。它也是测试先行开发的必要组成部分。

采纳方法

许多人靠自己成功实践了测试先行开发，不少人还在网上、会议和博客上撰写了成功的经验。但这种情况只是少数。目前为止，许多人还是依靠外界的帮助才得以成功。下面描述如何去实施。

1. 【必须】以团队的方式来学习、实践，并耐心等待队员们逐步深入体会、领悟该实践。

[7] 自动化测试、尤其是测试先行开发是测试驱动开发的主干，尽管它们不是团队测试驱动开发过程的全部。——作者注

在前两个月先不要急于质疑。开发速度可能会降低 50%(如果工作在一个具有遗留代码，且没有对应测试的项目上，那么速度可能会降低 75%)，对此要做好准备。

2. 【推荐】把开发者送去学习TDD入门培训 [8]。这种类型的课程通常有 80%到 90%的时间用于测试先行开发的实战；还会在经验丰富的实践者/指导者的帮助下，体会其它的实践方法。这类课程同时也为学员创造了一个良好的环境，学习遵守原则，让他们能起个好头。
3. 【极力推荐】寻求帮助直至成功。(最好能找到一个外部的咨询师，但是如果内部人员中有成功测试先行开发经验的 ,也可以寻求他们的帮助)。你真正需要的是那些经历过、并且相信这些实践方法的教练。他们要能与别的开发者结对编程，这样才能手把手地将这项技能传授给他们。一般来说，四到五个开发者就至少要配备一个导师，时间要长达几个月，而且该导师每个月至少拿出一个礼拜进行辅导。这样的专家要待在现场，这可以保持士气高涨，同时可以展示写测试的多种方式，还能够帮助团队调整实践方法，适应团队特殊的环境。
4. 【极力推荐】它的父模式，即自动化测试开发，鼓励采用结对编程，它可以大大降低学习曲线。

但是

这是所有常见的敏捷实践中最难采纳的方法之一。在自动化测试开发模式中所罗列的问题，这里都适用。另外，这种方式彻底颠覆了许多开发者的习惯，他们需要花很大的力气才能认识到，这的确是一种非常有效的开发方法。开发者必须先把怀疑放到一边，花足够长的时间学习体会，最终才能掌握如何有效地进行测试先行开发。该实践的失败率很高。

变化

许多分布式团队利用测试先行开发作为产生需求文档的一种方法。与客户面对面谈论需求的人负责编写测试，然后再将它们交给开发者来实现产品代码。

[8] 1. 许多课程是由咨询公司举办的，通常会以注册培训或者指导项目的方式来进行。可以通过 www.valtech.com, www.objectmentor.com 及其他网站获取更多的信息，或者找社区里曾经参加过这种类型课程的人帮你推荐，尽量参加具备实战经验的教练的课程。——作者注

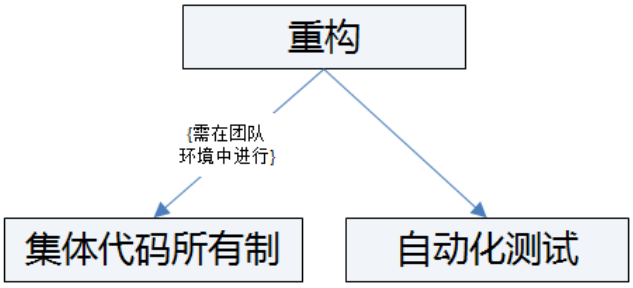
参考资料

- 在许多出版物中，测试先行开发是测试驱动开发的同义词。
- Kent Beck，《测试驱动开发》，中国电力出版社，孙平平等译，2004
- Feathers, M.《修改代码的艺术》，人民邮电出版社，刘未鹏译，2007
- Jeffries, R., Extreme Programming Adventures in C#. Redmond, Washington:Microsoft Press, 2004.
- Massol, V., 《Junit in Action 中文版》，电子工业出版社，鲍志云译，2005
- Rainsberger, J.B.《Junit Recipes 中文版》，电子工业出版社，李笑，陈昊，王耀伟译，2006

8

重构

重构是指保持代码原有行为的同时，改变代码的结构（例如设计）。



业务价值

重构允许并鼓励开发者根据需要来改进系统的设计，藉此增加系统的灵活性，延长产品的寿命。由于持续的重构会改进系统的设计，保证代码简单易懂、易于维护和变更，所以产品推向市场的时间和开销都会减少。

小故事

Uthman UpfrontDesign 在 Scott ScrumMaster 的团队里工作了几个月后，开玩笑地说他想把自己的名字改为 Rashid Refactoring。由于学习了其他的一些敏捷实践，例如结对编程和自动化测试开发，他开始关注自己从未涉及到的领域。当需求改变了或者增加了，他能够重新设计代码。通过阅读 Martin Fowler 的《重构：改善既有代码的设计》一书，在具备自动化测试的前提下，他学到了如何渐进地改进设计所对应的代码以满足新的需求。

在开发新功能时，他再也不会在一开始就试图完成大量的精心设计。不过这并不像他之前想得那样糟糕，他发现自己实际上每天都在做一些设计。重构，这一实践方法很快就代替了他曾经偏好的前期设计，成为他最喜欢的实践。

适用情况

适用于使用自动化测试开发的团队。你正在开发一个当前系统设计难以支持的需求；或者你刚完成了一个任务（当然，包括测试），但是你希望在提交到代码库前改进一下设计，使得解决方案更加清晰。

动因

软件开发会很自然地产生一些问题：

- 传统的经验告诉人们，随着时间的增长，软件复杂度会越来越高，越来越难于修改。需求变化了，但由于代码库过于陈旧和脆弱，完美的解决方案开销越来越大，因此只好对软件进行了创可贴式的治疗。
- 权宜之计会导致项目在设计上欠债，这就会让每天修改和阅读代码的工作变得更加艰辛。
- 代码重复必然妨碍代码的变更，而且很可能产生 bug。
- 出现了新需求或者需求变更了，当前的设计不再是解决问题的最佳方案。
- 软件开发是一个学习的过程，今天觉得还不错的设计，通过第二天的学习，可能就会觉得它很糟糕了。

因此

以渐进的方式来改进代码设计，这要好过快速却丑陋的修复。不要在增加新功能的同时改进设计，因为这会增加工作的复杂度。相反，可以在功能维护的时候进行代码和设计的改进。同时你得确保你所进行的功能维护是依赖于自动化测试和功能测试的。也就是说，起先所有的测试都能顺利通过，随后着手修改设计，接着修改程序代码，从而让失败的测试能重新顺利通过（不是改变测试代码）。基于一点——从全部测试通过开始，到全部测试通过结束——改进了设计的同时，系统的行为始终如一。这个过程就是重构。

重构是一个简单而优雅的活动。定期进行重构能够产生最好的效果，而且最好是在任务开始和结束的时候。也就是说，在新任务开始时，阅读代码，看看当前代码能否很好地支持你手头上的工作需求。如果答案是否定的，那么在真正动手实现功能前，对设计做一些必要的改进，然后再继续前进，完成任务及其相关测试。任务完成后，重新评估设计，如果还有改进空间，继续重构，这样才不会给继任者留下麻烦。

采纳方法

重构属于那些“马上动手”的实践之一（嗯，差不多是这样）。要牢记：重构是一种实践，而不是工具，虽然工具的支持确实挺有用。下面是实践重构的要领：

1. 利用自动化测试，直到你已经习惯于为所有的任务编写测试。对于要修改的某段代码，必须有足够高的测试覆盖率，再尝试去重构它们。。
2. 在团队中，使用集体代码所有权。由于重构而导致失败的测试，要及时的修复。
3. 阅读 Martin Fowler 的《重构：改善既有代码的设计》和有关测试驱动开发的书籍，并通过书中的练习进行演练。
4. 按部就班，动手实践。对每一个任务，查看它们的设计，看看面对新的任务，设计是否有值得改进的地方。完成任务之后，查看自己的代码，如有需要，立即清理。要严格地、毫不留情地进行重构，也就是说：只要可能，在任务开始和结束时都要重构。
5. 两周举行一次小组讨论，让大家一起分享重构的经验。
6. 熟悉了《重构》一书中的所有经典模式之后，就可放开手脚、大刀阔斧地进行重构了。当你和团队都认为需要对设计做出重大改进的时候，向这个方向努力。买本 Joshua Kerievsky 的《重构与模式》，并找人形成小组，系统学习书中的实例以应对大型重构。

但是

重构是开发者最强大的手段之一。尽管如此，还是有下面的一些地方需要小心：

- 重构不会带来任何直接的业务价值。重构的定义告诉我们，它会维持系统的行为，所以对用户来说这个过程是完全透明的。因此，从客户的角度来看，没有需求驱动的代码重构，实际上是一种浪费。
- 如果错过了许多小规模的重构机会，日积月累，就会导致大规模重构，而这种重构通常很难实施。因此，平时需要更勤奋地去做重构，培养对代码、设计异味的敏感程度。
- 在团队中，重构可能会导致别人写的测试失败。有些才接触敏捷的人，在提交代码之后，可能希望编写了测试代码的人来修正失败的测试。如果这样的事情发生了，就破坏了自动化测试开发。要记住：提交的代码是决不应该导致已有测试失败的。确保团队实行集体代码所有权，这样才有可能在重构后通过所有测试。

变化

我们都知道：在软件开发中，最大的开销不是从无到有开发出软件系统，而是维护费用。所以，如何让软件系统更易于维护，这非常值得关注。

传统的做法是超前设计——也就是说我们要设计足够灵活的系统，当新需求到来时，，不用对已构建的系统进行设计上的任何改动就能开发新模块。但是这种方法暗含了一种开销：具备通用性的设计更为复杂。在每次开发的过程中，开发者都需要费神费力去弄懂、并使用这些复杂的模块和代码，这其中就耗费了大量成本。设计模式是最常用来进行灵活设计的技巧。Erich Gamma 在他 1995 年版的《设计模式：可复用面向对象软件基础》中介绍了这一系列的解决方法。

在 2005 年夏天的一次采访中，Gamma 说他的想法已经有所改变，他已经开始利用简单设计来完成手上的需求。当新的需求出现时，他会向着某个设计模式的方向来重构解决方案。这样一来，除非到了真正需要的时刻，他不会进行任何复杂的设计。

当然，这会带给我们另一种想法：利用重构向模式靠拢，而不是在先期大量使用设计模式。这样的步骤可以让我们同时拥有两种技术的优点。Joshua Kerevsky 的《重构与模式》一书，包含了许多实例，专门针对目前的开发环境，讲解如何有效利用重构，解决常见问题。

参考资料

对于软件开发者，Fowler 的书应该人手一册。Kerievsky 的书适合有了一定重构经验的读者，它可以帮助你的重构匹配已知的模式。

- Martin Fowler《重构：改善既有代码的设计》，中国电力出版社，侯捷、熊节译，2003
- Kerievsky, J《重构与模式》，杨光、刘基诚译，2006

9

持续集成

每一次向代码库中提交变更代码时，都会执行一次完整的编译构建，进行全面集成，并运行所有的测试，这就是持续集成。它常随着每个开发人员的代码集成入库而进行。

业务价值

由于可以及早并且经常地发现集成相关的缺陷，持续集成能够缩短上市时间，同时也可以保证拥有较高的质量，从而去除了迭代“稳定化”阶段以及随之而来的重复工作。同时，通过让开发团队和相关负责人明确了解项目的进展情况，持续集成也增加了项目的可见度。

小故事

BuildMaster Bob 阅读了许多关于持续集成的资料，而且注意到许多项目中出现的问题，听说都可以由这个实践来解决。在接下来的几周里，他花了一些时间，彻底自动化了构建的过程。此时，他会运行每夜构建，并将构建结果展现在一个 Web 页面上，以让整个团队都能看到。

Bob 向 Scott ScrumMaster，Cindy Coder 和 Dave Developer 展示了他的杰作，并希望得到他们的支持。大家达成一致意见，决定一旦发现构建问题就会尽早解决。Cindy 和 Dave 也同意与 Bob 一起把他们所写的自动化测试作为每夜构建的一部分运行。

过了几个迭代后，开发团队开始依赖于这种运行所有开发人员测试的每夜构建。只要一发现问题，他们就会马上解决，这让整个团队更加清楚地了解到集成方面的问题和原因。Bob 向所有开发者共享了构建脚本，利用该脚本，开发人员可以在向代码库中提交代码之前先在本地集成。

基于前期的成功，团队决定实施彻底的持续集成。Bob 利用了接下来三周的大部分时间，安装、配置了一个持续集成工具，并且将构建时间缩短为 10 分钟以内。

适用情况

适用于处于下列情况的开发团队：希望降低“稳定化”迭代带来的风险；希望在实践自动化测试的同时，保证所有测试依然通过；正在引入功能测试实践，希望保证团队逐步增加新功能的同时，不会破坏原有的功能。

动因

持续集成能直接地解决当今典型的软件开发生命周期中存在的诸多问题：

- 人们一直认为集成充满风险、困难重重。因此一些实践相应而生，以降低不确定性带来的风险。（但它们都不是最佳解决方案）：
  a) 各个子团队独立工作，并用通过 Stub 方式模拟其他子团队的代码，以避免集成的需要。
  b) 每个子团队开始工作时，都会拿到关于自己所负责的子系统边界接口的详细设计，以保证各个子系统之间无缝整合（此类的情况几乎从未发生过）。
  c) 在一个开发周期快结束时利用一段“稳定化”迭代，找出团队做出的错误假设，并进行解决。
- 集成的风险随着时间呈指数级增长。
- 由于缺乏集成，导致大量 bug 在系统中，很多很严重，这也可作为重大的设计不符合系统的症状。
- 每个 bug 都隐含着一个错误。如果这些错误未被发现和纠正，对这部分有依赖的代码，就会基于不正确的假设。
- 成功的集成是功能测试成功的先决条件。
- 在敏捷社区里，集成指的是构建一个完全可工作的系统——其中的过程包括编译、部署和测试——而不仅仅只是一次成功编译。

因此

如果您因为集成太过痛苦就逃避集成，而且将其推迟到软件快发布时才开始，这是不对的，您应该拥抱痛苦！在软件开发中，痛苦意味着存在某些问题；我们应该利用这样的反馈修复

问题，而不是视而不见。

我们通常会认为我们做集成是出于义务。。我们进行集成的唯一原因是向客户交付系统，这是(这是非常重要的原因)，所以我们只会在发布软件之前才去集成。

通过反思集成带来的痛苦，可以知道做好集成能够带来哪些价值：

- 缺乏集成会带来风险，并掩饰缺陷。因此，利用频繁集成可以尽早，经常地发现缺陷，从而有可能避免某些深层次的错误。
- 集成的困难之处，往往都与需要许多手动完成的步骤、不同版本的代码库间的同步、所依赖程序库、及其它所需资源的同步等有关。不存在什么步骤完全无法自动化。因此，应该努力将集成过程中所有的步骤都自动化。
- 集成是关于系统整体状况的一种有效反馈信息。信息反馈越频繁，我们才会变得越敏捷。这种信息不要藏着只让负责构建的人知道，它需要作为一种反馈，让整个团队都能看到。

采纳方法

要实现持续集成，首先要完成自动化构建。也就是说要取消所有的手动步骤[9]。针对不同的开发环境，可以利用对应的工具来完成自动化构建[10]。如果正在使用的工具不支持自动化，那么很可能需要更换或者增加一些工具（例如，有些工具需要有人定期处理对话框操作，像这样的就不适合自动化构建了）。不过这样的工作不需要整个团队参与，分配给某些人单独完成即可。

[9] 一些通常的步骤包括：签出某个构建版本的当前代码，标记构建号，生成数据库结构，编译，运行自动化测试等等。——作者注

[10] 如果使用 Java，可以利用开源工具比如 Ant 和 Maven；如果使用.Net，可以利用 NAnt 和 MSBuild；针对 C/C++也有许多构建工具（而不仅仅是 make）。——作者注

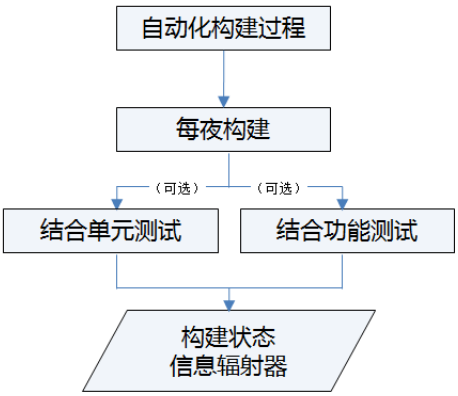


图 1 达成持续集成的步骤

自动化构建完成之后，就要让它定期运行——最好是在晚上，并为团队产生一个构建报告。展示构建报告最好的方式之一是建立信息辐射器 [11]。此时，必须要让开发团队参与进来，告诉大家每天都有构建报告，让他们知道解决导致构建失败的问题是这个团队的高优先级任务。

要保证让大家知道，看着构建失败坐视不管是不可接受的。如果出现任何错误，必须在新的功能提交到代码库之前做出修正。要实现这样的目标，需要每个开发者都可以利用一个脚本在本地构建系统，这样就可以在提交代码前进行验证，避免破坏构建。下面是达成该目标所需的两个步骤：

1) 保证每个开发者的机器上都有构建版本和外部部署配置说明；

2) 尽可能加快构建速度，这样开发人员在提交之前，就能比较现实地在本地完成测试。

根据不同的开发环境，上述步骤可能难度很大，复杂的环境很可能让人退缩。如果这对团队确实是大问题，那么就使用一台或更多的部署机器，供开发者在正式提交前先验证修改过的代码。切记人的实践总是比工具重要。作为团队，必须要重视提交代码之前的集成——特别是在关键时刻——要避免版本构建失败。

一旦开发人员能够在提交代码前，完成一次完整的本地集成，这就是时候向他们展示持续集成的核心理念了，即——频繁获得反馈。本地构建使得开发者的代码融合进系统，从集成的结果可以得到迅速的反馈。但是，什么情况下使用？使用的频度如何？答案是：每一次向代码库提交代码前，都应在开发者的机器上使用这个集成工具。只有在本地集成成功后，开发

[11] 信息辐射器放在团队的工作区域，方便让人们看到，理解它们。比如每天打印出来的构建状态报告、海报，或是用于展示构建状态的显示器。——作者注

人员才能再提交代码。

依照经验来看，开发者应当尽可能频繁地集成，用于验证自己的代码能否与整个系统一起工作。到底能够多频繁？这要根据开发方法的不同确定。测试先行开发等实践鼓励采纳小步，可测试的工作方式；在这种情况下，一个开发者每天大概会提交 4 到 5 次代码。即使不能保证这样的速度，每天也应该尽量提交一次代码，这样可以迫使他们每天至少可以成功地集成一次。[12]

这种情况下，开发者每天至少会从集成状态里获得一次反馈。对于大多数的提交，本地集成都会顺利地通过，少数会导致本地构建失败，这时候，开发者就需要在提交代码前将问题修正。本地通过，却在正式的构建机器上失败的情况更少发生，一旦出现这样的情况，需要找到导致失败的那次提交，把回滚代码让构建通过，然后在本地将错误修正。接下来需要在本地机器上修改构建脚本，从而可以在将来捕获类似的错误。

最后就是关于测试了。前面我们故意避开有关测试的话题，就是为了让大家知道，即使没有测试，持续集成依然拥有极高的价值。不过，如果团队拥有自动化开发人员测试或功能测试，就应当把它们集成到持续集成中。应当尽早地把测试集成进来作为构建的一部分，或者在构建成功之后立即运行。此时，导致构建失败的含义，就从编译和部署的失败转变为没能通过所有自动化测试的失败。

但是

持续集成有时候会变得很缓慢或者很脆弱。构建的持续失败，是这种现象发生的一个主要因素。虽然很糟糕，但并不是一无是处，它让我们知道有些东西发生了错误，而且没有人来修复。开发团队开始忽视构建流程，并且退化到以前的不良状态。尽一切可能想些办法来加快测试的速度。

频繁、长期地构建失败是持续集成失败的前兆。通常，如果可以很容易地知道是谁破坏了构建，那么就不会有什么问题。怎么才能让谁破坏了构建更明显地表现出来？如果只有一个人，比如 Cindy Coder，提交了代码，然后导致构建失败，那么找到犯错的人很容易，当然，应该是 Cindy 负责。

但是，如果是 Cindy、Dave 和许多人都提交了代码。谁才是导致失败的罪魁祸首？通常，人人都信誓旦旦地保证自己的提交不可能导致构建失败，而且身上还肩负着一个紧急的任务

[12] 许多人都习惯于利用几周的时间开发一个功能块，然后再提交代码（之后很可能在迭代“稳定化”期要等上几个月去集成）。这通常跟问题的任何限制条件无关，完全是我们自己的工作方式所导致的恶果。因此努力争取每天提交一次。——作者注

停不下来。总之，他们之前都进行了本地构建，没有任何错误，所以肯定是别人犯了错。请提前警告所有人：这种情况是一个恶性循环，如果不修正问题，其他四个人会提交代码，可能让问题更加错综复杂。他们也都运行了本地构建，但提交之后却都失败了，但这是因为之前的失败，而不是他们新提交的代码所致，对吧？容忍一个失败的测试，就会在一两天之内产生十个失败的测试。

那么，问题的根源是什么？答案是构建太慢了。如果在下一个构建周期前，许多人（通常超过三个人）都可以持续的提交代码，这就意味着构建需要加速。但如果是每夜构建，那么就无法避免这种情况，解决的方法是可以让一个人专门作为“构建警察”监控构建，如果失败了，通知并帮助开发者修正错误。但是要想拥有真正的持续集成，就必须让构建更快一些。不同的开发环境所产生的构建速度也不同，所以这里也无法给出通用的解决方案。尽量去创造，记住构建的地位不再低下，需要像关注开发流程的其它任一部分一样，去关注它。

持续集成不是一个轻而易举的实践方法，在采纳阶段尤为困难。对于许多开发环境，需要花费相当多的时间来完成自动化测试和加快构建速度。而且，也必需整个开发团队投入一定时间和精力来维护构建。

变化

由于持续集成使得软件开发环境更有效，管理更透明，所以它也是非敏捷社区采纳的实践之一。以下是一些持续集成实践常见的变化：

- 持续集成经常在企业级应用场景下使用。每个项目都有自己的持续集成工具，工具之间通过层级关系链接在一起，一个构建会从别的构建库中下载成功的版本作为自己的组成部分。
- 使用单一的源代码库。当持续集成包含了自动化测试和功能测试时（更多详细内容可参见测试驱动开发和测试驱动需求组合两章），团队就会对代码合并的质量充满信心，所以就不需要拆分源代码库，使用单一的可用源代码库就足够了。
- 因为功能测试是对完整的系统进行验证，所以要比自动化测试更花时间。因此有些团队就将功能测试从持续集成中分离开来，作为补充部分。这是一个很常见的模式，所以值得去分离出来。但同时要强调的是，这一点经常被过早地使用。如果把这些测试从持续集成中分离了，这些测试就会过时，变得没用，进而抵消了集成带来的大多数好处。如果可能，我个人希望尽量不要使用这种解决方案，把精力用在如何使测试和构建更快上。

参考资料

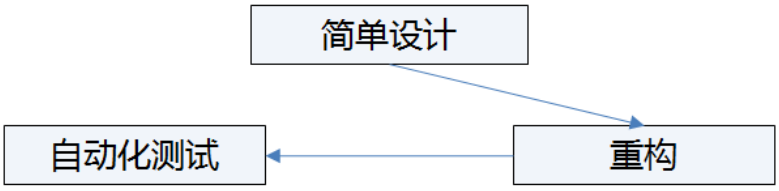
许多关于持续集成的资料都可以从网上获得，利用任一搜索引擎都可以找到一些工具和文章。最早关于持续集成的文章是 Martin Fowler 在 2006 年 5 月更新的。

- Martin Fowler，"Continuous Integration"，www.martinfowler.com/articles/continuousIntegration.html。

10

简单设计

设计的复杂程度应当只满足当前的需求，多则画蛇添足。保持设计简单，外加上自动化测试和重构作为保障，会使开发变得快速有效，维护变得更加轻松，并能增量修改设计。



业务价值

简单设计是一个非常有效的实践方法。因为团队只开发了需要的部分，产品就得以上市，从而获得商业价值回馈的周期缩短了，成本也降低了。同时，因为简单设计易于理解，易于改变，产品寿命也相应地延长了。

小故事

一年前，当 Waterfall Will 加入 Scott ScrumMaster 的敏捷开发小组的时候，他很担心团队成员的编码技术和团队本身。最重要的，他还有一个非常急切的问题，“设计在哪里？！”。他并不很清楚怎么样来实施简单设计才能奏效。根据以往的经验，团队必须预先进行架构和前端设计，否则频繁的变更就会不停地导致设计的变化，从而让开销以指数级的增长。

Will 决定暂时不去担心这些问题，在未来的几个月内，先试一试这种新的开发方法。起初，Will 有点不情愿，但又不得不承认，简单设计确实更加优雅。事实胜于雄辩， Will 也一次又一次地发现，有了自动化测试这张安全网，简单设计在处理需求变化时非常灵活，他也开始喜欢上了这种开发方法。他发现自己的设计变得更加简洁，而实际上也没有抛弃设计，而是将设计融入每天的工作中。

适用情况

如果你的团队有以下需求之一，就适用该实践：

- 由于需求经常发生变化，所以团队必须很有弹性，灵活应对变化。
- 客户可能并不知道到底需要什么。为了帮助客户判断什么是真正需要的，应当尽快的开发出一些东西让他们使用。
- 团队希望大大缩短上市时间。
- 由于团队在使用复杂或不熟悉的技术，您希望等到团队熟悉了这些问题或技术后，再开始进行重要的设计。

动因

该实践方法尝试解决存在于软件开发方法中的一些典型问题：

- 开发者和客户的交流反馈很少，因为从需求到可工作的软件需要很长的时间。
- 需要实现的功能非常复杂，开发人员有“跑题”的趋势，并且给客户带来了一些错误的解决方案。
- 因为过度设计，需要花很大一部分开发时间，去理解和使用软件中的花哨设计，即使客户从来就不会使用到这些。
- 设计的复杂性会带来“设计的累计开销”，也就是说每一位开发者在试图理解、使用和测试这些复杂代码的时候，都会带来不必要的时间花费。假设这些复杂性是为“明天”设计的，我们却在“今天”进行了投入。不幸的是我们开发者费尽心思设计要应对的复杂情况，却从未真正出现过。

因此

敏捷社区相信 那种为了缩减成本 ,而在软件中预先加入复杂性的做法是错误的。在设计中，通用性设计往往不是仅仅满足当前需求，而过分注重提供更多灵活性。敏捷社区把这种方式贬称为先行过度设计，缩写 BDUF（Big Design Up Front）。

我们不是预言家，也无法预知未来的变化。先行设计不是免费的午餐，追求这种通用性会导致软件更加的复杂且难于理解，维护成本也会远远高于简单设计。因此先行过度设计所带来

的累计开销远远高于它所带来的收益。

设计的复杂度应当仅限于当前迭代的需求。设计应当是简单的设计，并且不包含任何通用的扩展，原因有两点：首先，未来两年里，没人知道需求会变成什么样子，然而将设计的通用性搬进软件中，会导致两年内都要承担设计的累计开销；其次，由于有了重构和自动化测试，当需求要求设计发生变化时，花费的成本也不会很大。

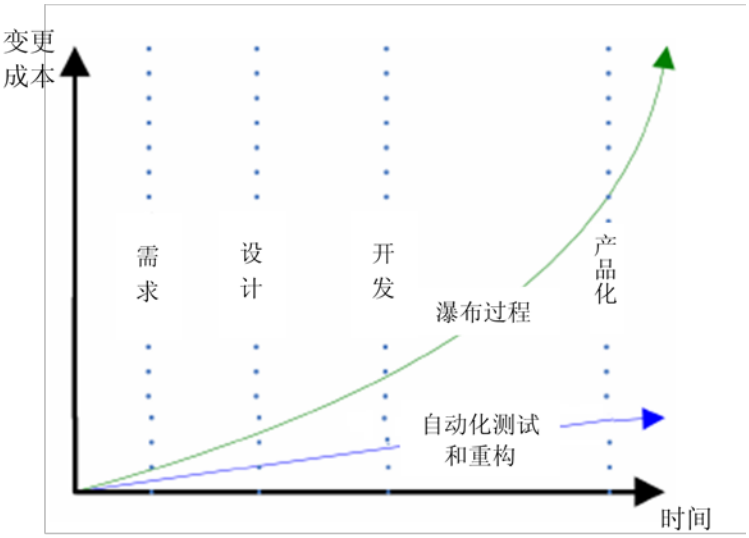


图 2 瀑布模型和 TDD 中变更所花费的成本

采纳方法

那么，如何进行简单设计？

1. 充分理解当前的需求和任务，不要做任何含糊的判断。
2. 确定解决方案。可以通过先写测试，然后让测试推动解决方案的诞生，或者利用更传统的方法，编码前进行设计。
3. 如果建立在当前代码基础上的解决方案，不足以支持新的需求，那就先重构代码，让它们具有扩展新功能的能力。利用现有的测试来保证改动仅仅更改了设计，而不是软件的原有行为。
4. 增加一个新的功能，与之对应的解决方案的复杂度，应当仅仅满足新的需求。

如果没有重构和逐步演进设计作为保证的前提下，就不应该采用简单设计。另一方面，重构依赖于一系列的自动化测试。这些实践是简单设计的必要前提。

对于开发者，要想有效的使用简单设计开发方法，必须暂时放下自己的怀疑 [13]，尝试在开始的几个迭代中使用该方法，先观察它是如何起到实效的。

但是

小心以下的陷阱：

- 有人会让团队回归于先行过度设计，从而可能放弃了简单设计。他们认为重构是一种浪费。他们并没有意识到先行过度设计是一种超前的浪费行为；与期望相反，大多数的简单设计带来的是复杂度较低的设计。
- 团队认为简单设计就是需要耗时最短的设计，经常性地复制、粘贴解决方案。这不是简单设计。这种代码非常糟糕。
- 因为不相信简单设计的有效性，团队会拒绝采纳这种实践。人们不采纳它，通常是因为简单设计这种思维方式相悖于人们过去的经验。我们强烈推荐大家像 Waterfall Will 一样，放下顾虑，尝试一下这种开发方法。经过二、三个月的实践，团队将会毫不犹豫的采纳这种开发方法。

变化

以下是与简单设计开发方法一起使用的其他开发方法的组合。所有的例子都属于演进式设计组合。

- 测试先行开发，简单设计，重构。先写测试，在简单的设计上编码让测试通过，然后利用重构将简单设计变得更加合理。
- 简单设计，测试后行开发，重构。用最小化的设计来满足当前的需求，在设计的基础上进行编码，编写测试来验证代码，然后重构代码让设计变得更加合理，代码变得更加整洁。

参考资料

简单设计是从极限编程中诞生出来的：

- Kent Beck 解析极限编程--拥抱变化，人民邮电出版社，唐东铭译，2002

[13] 针对当前的需求，许多富有经验的开发者往往很难实施简单设计。多年来，对未来的特性进行通用性设计和超前设计，让他们很难相信当前的设计是可变更的，也很难相信变更的开销会很小。——作者注

- Kent Beck， Andres, C. 解析极限编程--拥抱变化（第二版），电子工业出版社，雷剑文、 陈振冲、李明树译，2006

11

（自动化）功能测试

业务价值

功能测试主要通过大大增强团队沟通以及彻底验证需求，增强了产品的市场价值和开发团队进展的可见度。作为一种测试和反馈，本实践能帮助延长产品的生命周期，缩短上市时间和降低系统开发成本。

小故事

Mustapha Mentor 作为兼职顾问加入 ScrumMaster Scott 的敏捷团队，来帮助团队改进软件开发过程。两周之后 Mustapha 发现有很大一部分完成的用户故事没有通过 Chris Customer 和 Aparna Analyst 的验证。通过进一步的调查，Mustapha 发现由于从需求到代码的过程中存在误解、失误和疏忽，因此即使开发者完成开发之后还是需要一两天作为“稳定期”。

Mustapha 之前遇到过这种情况。尽管团队在实践测试驱动开发，这能保证“正确地解决问题”，但如何去“解决正确的问题”，还存在一些弊病。虽然利用迭代方式开发，能够定期捕获并修正一些缺陷，但的确还可以做得更好。

在接下来的几个迭代，Mustapha 让团队成员阅读 Rick Mugridge 和 Ward Cunningham 所写的《用 FIT 进行敏捷软件测试》一书，并帮助他们使用 FIT 工具构建一系列自动化测试，它们不再使用用户故事，转而用这些自动化测试来管理需求。接受这种方式的过程很缓慢，但经过了五、六个迭代的艰苦工作和积极尝试，团队完成了大量的测试，积累了丰富的经验。彻底消除了系统稳定阶段，开发速度得到了提升，同时设计质量也得到了改善！

适用情况

您所在的软件开发团队希望显著的改善软件产品质量。也就是说，想写出缺陷更少的软件代码，不只是正确地解决问题，更是解决正确的问题！您和团队成员都愿意为这种改善付出最大的努力，愿意重新审视当前的设计和架构，并且加以完善。为了最终能提高工作效率，大家愿意暂时牺牲项目进度。

动因

在软件开发过程中，该实践能够解决如下常见的问题：

- 随着模块间依赖的增长，Bug 也不断出现。单元测试能够保障单个类不出现臭虫，但不能控制模块之间的 Bug。此外，随着代码量的增加，模块之间潜在的 Bug 数会增长得更快。
- 不知道任务什么时候完成。几乎每个人都经历过项目声称已经结束，但之后还需要继续工作几周或几个月。
- 对需求的误解。尤其在分布式、国际化的团队里，因为文化的差异，经常错误理解需求。这一边看似很明白的东西，但是另外一边却常常会觉得不知所云。
- 不明确的需求。 在项目宣布“完成”后却拖延下去的原因之一是最初需求不够明确。口头和文字上提供的需求无法为代码开发提供足够的细节。开发者根据对客户的意图的猜测去实现功能，然后认为项目完成了。但如果开发者猜错了的话，代码就不得不重新来过了。
- 自相矛盾的需求。很多“结束”的项目由于陷入修复 bug 的循环，停滞在测试阶段。一个简单循环的例子是当臭虫 A 被修复，又发现臭虫 B；把 B 修复后 A 又再次出现。但是该循环通常不那么明显，尤其当 A 与 B 属于系统的不同部分，或者必须经过较多的手动测试步骤才能重现时更是如此。
- 过时的需求。项目持续时间越长，部分需求滞后于代码实现的可能性越高。坦白地讲，我们中谁能保证经过一年的开发，需求依然百分之百符合最新的要求？过时的需求要比没有需求更糟糕。如果没有需求的话，开发者会从客户、代码或者单元测试等所有可能项中获取相对较新的需求。但是过时的需求是错误信息。它们会使开发者走向错误的方向，从而会浪费可观的时间。
- 发布延期。随着应用的成长和产品逐步成熟，测试部门需要花更长的时间完成测试周期，

这就造成了发布不停地推迟。

- 缓慢的手动测试。相对于小型项目而言，大型项目花往往由专门的测试部门进行手动测试，花在这上面的时间会显著增长。由于手动测试很慢，代码修改导致的问题要很久之后才会被发现，从而导致 bug 的反馈周期加长。由于反馈不及时，因而要花费更长的时间来分析是由哪次修改导致了 bug，最终修正由测试部门发现的缺陷的周期也变得更很长。
- 缓慢的补丁。低效手动测试的副作用是，针对某部分报告的 bug，其补丁的发布也变慢。在许多开发环境中，开发人员必须部署一个完整的数据库并进行多步手动操作来重现一个 bug。而且他们必须通过重现来找出问题并确保 bug 已经被修复。

因此

引入一种能用编写成代码的方式来描述业务流程的自动化测试，并把它们当做“可执行的需求”。这些可执行的需求是由客户在每个迭代之初编写，并提供给开发者的。不幸的是，该实践依赖与一些工具，需要一个能被非开发者使用的工具，比如 FIT（http://fit.c2.com）或者 FITNesse（http://fitnesse.org/）。编写可执行测试对需求的精确度要求很高，因此要求客户需求明确（无二义性）而且十分具体，这对客户来说是很大的挑战，因此他们需要测试人员或者开发者的帮助。这种类型的测试就定义为功能测试，它们是客户和开发者共有的自动化业务流程测试。

在功能测试中包含着开发人员写完代码后，客户一般用于验收测试的信息。一个功能测试可以被看做是有一些具体输入值的用例。例如，设想我们需要为一个在线商店制作应用程序，当前迭代的需求包括保证存货清单管理正确工作。下面是可用作需求的 FIT 测试样例。（别被表格的格式吓到了，花一些时间阅读这些表格并把它们当作需求。）

物件清单管理测试

为测试加载所使用的基本数据

首先，让我们从外部数据源加载一批标准的物件存入到持久化库中。这样的话，我们的测试就有了重要的基础数据。

fit.ActionFixture		
Start	Com.valtech.post.service.tests.fit.	

	ItemInventoryFixture	
Enter	Inventory	./src/conf/valtech/post/service/tests/fit/inventory.txt
check	Total items	10

好，我们有 10 件物品。现在需要确保细节的正确。

com.valtech.post.service.tests.fit.ItemInventoryDisplayFixture		
Upc	Description	Price
2458	Chocolate	0.75
1234	Cola	0.99
3214	Milk	2.34
8743	Eggs	2.35
0987	Olives	2.43
1233	Apples	1.12
8745	Paper Towels	3.45
9457	Canned Soup	1.24
2345	Cheese	5.65

现在，成功读取了一批物品后，需要维护一下类别信息了。

fit.ActionFixture		
start	com.valtech.post.service.tests.fit.ItemInventoryFixture	
enter	select	2458
check	description	Chocolate
enter	description	Dark Chocolate
check	Description	Dark Chocolate
enter	Add item	1111

enter	Description	Honey
enter	Price	5.60
check	total items	11
enter	remove item	0987
enter	Select	0987
check	upc	0987
check	description	Not Found
check	price	0.01

从开发的角度来看，这些测试应该对系统中图形界面以下的那层进行验证。这一层通常被称作服务层或系统外观层(System Facade)。在这层写测试会验证整个系统和所有的业务逻辑（图形界面中不应有业务逻辑）。因此，功能测试可以保证需求得以满足。因为开发者可以用这些先前所写的这些测试，来判定开发是否“完成”——当开发者编写了足够的代码使测试通过时，就满足了需求。

所有可执行的需求都应该详细记录到能够经常运行的测试套件中——最好作为持续集成的一部分。这样测试套件会逐渐增长，并不间断地运行，只要保证测试都通过，系统就永远不会破坏任何已满足的需求。也就是说，如果任何需求之间发生了冲突，起码会导致一个测试失败，然后开发者就会马上与客户澄清。

基于功能测试实施的成功，您可以期望得到如下好处：

开发团队更有信心：当开发基于一个稳定的测试框架时，开发者会展现出绝对的自信心。如何让开发者对自己的代码更有信心？自动化测试和测试驱动开发两种方法对此大有帮助。这并不仅仅是一种舒适的感觉(总是对士气有益)，更由于开发者可以通过重构来改进代码，从而使开发更快。功能测试对信心的提升要比自动化测试高一到两个等级。由于客户、分析人员和测试人员与需求和回归测试的直接关系，功能测试对他们信心的提升也大有裨益。只要测试是绿色的[14]，就意味着相关的功能可以正常可用。

强健的测试：不关注业务逻辑，功能测试驱动着服务层。业务逻辑趋向于稳固不变，所以对应的测试也不需要做太多改动。相反，当图形界面元素进行了重组，那么原先基于图形界面元素的测试就会失败。

[14] 敏捷测试工具常使用绿色标识测试通过，红色标识测试失败。——译者注

错误和 bug 能被快速重现：当发现一个 bug 的时候，就会添加一个功能测试，这样该 bug 就不会再烦扰我们。当然，对于 bug 较多的代码，也应该编写自动化测试，不过当开发人员开始调查某个 bug 的时候，是很难发现哪个单元发生了问题。不过（希望）他们知道是哪个用例发生了问题，这样就能够立即编写功能测试。通过在发现 bug 时及时地添加测试，可以杜绝由于 bug 反复出现，导致系统变得不稳定。

当系统从开始开发到投入生产，开发新功能上使用的时间逐步递减。通过功能测试框架，建立了“业务语言”，就可以直接根据 bug 的报告编写一个功能测试来重现错误这比自动化测试要好很多。这允许开发者用一种可执行的 bug 重现方式来反复研究代码，而不必花费精力去不停地准备环境。

测试人员有时间变得更为积极主动。如果缓慢的手动测试是实施功能测试的动因，那么快速的自动化测试会带来很大好处。这最终会使得测试人员从日复一日对主要业务规则的手动测试中解脱出来。如此以来，测试人员有了更多的时间，工作也会更为积极主动，帮助开发者一起设计更多可测试的代码，而不是等待到每个迭代末尾去“打扫战场”。

什么时候任务完成对大家一目了然。使用功能测试确实能帮我们了解一个任务什么时候算是完成了。不仅如此，功能测试还使整个项目的实际进展对整个开发团队——客户、分析人员、开发者、测试人员和管理者——完全透明。任何时候，测试通过或失败都清清楚楚。在功能测试层上，对产生的业务价值花一点精力，就可以将它们用于需求管理分析。

更好的设计，更好的架构：功能测试能促使更好的分层和子系统的分离。设想针对一个多层级的架构：因为功能测试针对服务层，任何在表现层出现的业务逻辑，要么必须在测试准备时重复这些逻辑，要么需要把这些逻辑移到服务层。

同样的，想想系统中的子系统——某些功能模块，例如计税模块。任何从这个模块中漏掉的税务内容， 除非将它们放入计税模块中，否则都会在数据准备代码里重复出现。功能测试从而促进子系统形成稳定的职责。

分析人员需要思考更多的需求细节：由于分析人员撰写测试时需要很多描述信息，所以他们要思考更多的需求细节。例如，一位分析师可能会要求文本框在不需要的时候，要设成禁用状态。但当他为这个需求写功能测试的时候，就需要详细的描述在哪些情况下哪些文本框——或者其实是在下面服务层中的相应属性——需要被禁用。

改善了客户和开发者之间的沟通：经过一段时间，对功能测试的讨论会帮助团队产生通用词汇和对系统的共同理解（正如 Jim Shore 在 www.jamesshore.com/Blog/A-Vision-For-Fit.html 上分享了他的想法）。从 Mugridge 和 Cunningham 所写的《集成测试框架:用 Fit 进行敏捷软件测试》一书中可以找到对于这

种合作方式的例子。

采纳方法

那么如何才能成功地实施功能测试实践方法呢？

1. 在开始的几次迭代中，让测试人员、开发者与客户、分析人员一起编写测试。
2. 【极力推荐】获得外界的帮助：找几个真正在敏捷环境中成功实施功能测试的人。因为这是一种整体团队的开发实践，它并不仅仅是系统级的自动化测试。
3. 【推荐】与客户、分析人员、开发者、测试人员一起成立一个学习小组阅读《集成测试框架:用 Fit 进行敏捷软件测试》。
4. 选一个工具，不要尝试自己开发。FIT 和 Fitnesse 在该领域应用得最广泛。
5. 为了支持团队将要发展的“领域语言”，需要让开发者计划实现一些“fixture”。不要尝试让客户和分析人员了解已经编写好的对象和方法，这将会妨碍创建出项目对应的领域语言。
6. 开始时，安排一个分析人员或客户与一个开发者共同负责一个迭代的一个用户故事，一同编写包含具体值的用例情景作为测试。
7. 逐渐让更多的团队成员了解功能测试。
8. 如果您正在维护一个既有系统，那么很可能需要进行一次不小的重构来适应测试。这时候，您需要实践自动化测试。如果团队还没有引入这些实践，那么为了能够继续系统重构，您就需要采纳这些实践。
9. 不要停止功能测试，在对测试所需要的相关部分重构完成之前，可以先编写测试并手动运行。即使测试无法自动化，这种层次的细节依然可以用于测试驱动——可以写下这种测试，然后开发者可用其判断功能是否“完成”。
10. 在向功能测试迁移的过程中，让一个开发者承担“功能测试警察”的职责会很有帮助。警察的工作是查出谁导致功能测试不通过，并协助找出代码破坏测试的原因，然后帮助他们修正问题。
11. 【可选】在成功编写功能测试之后建立学习小组，阅读《领域驱动设计》。把团队创建的语言与书中所写的内容直接联系起来。
12. 计划 3 到 6 个月的适应期，直到团队开始有规律地编写功能测试。

13. 如果是一个存在了很长时间并且没有测试的项目，计划一段 3 到 12 个月的时间，因为很可能需要一段非常重要的重构工作来使应用该实践成为可能。

与敏捷开发中的任一方法相同，采纳功能测试也需要循序渐进。要记住“人”比“过程”重要。也就是说，逐步地使开发者和客户得到培训，并让他们编写一些集成中的功能测试。然后，在团队有了一些可工作的功能测试，并把这些功能测试作为构建的一部分之后，收集开发人员和客户对于工具与过程的反馈。不停地改进工具和流程，直到开发者和客户都对功能测试满意为止。然后再逐步地在整个团队内推广该实践。

但是

有许多有根据的原因造成功能测试没有在敏捷社区中推广开来。功能测试很容易出错。一般有两种误区容易导致错误。一种是与其他但是章节一样，在于采纳实践本身。另一种非常重要的误区是关于系统的基础架构。如果系统架构对功能测试支持不够不友好，那么就需要先改变系统架构。

实施异味

发现测试失败时，缺乏甚至没有责任感：如果对失败的测试没有一点责任感，那么测试就不会得到修复。一般说来，如果很难确定是谁的代码导致测试失败，那么就很容易造成责任感的丧失。当测试运行周期速度大大落后于开发者提交代码周期的速度，上述现象就会经常发生；例如，在上次运行测试之后若干开发者提交了代码，这就很难确定是谁的改动破坏了测试。那么怎么解决这个问题？简单，让测试运行得快一些，以下是该怎么做：

首先，团队中的所有人都应该认同功能测试是开发过程中的一个主要实践方法，而不是什么可有可无的东西。当不可能放弃测试时，团队就会找到创造性的解决方法。主要是如何能够提高功能测试运行的速度，使之能够在提交之前运行在开发者本地的机器上。我们发现的一些实用策略如下：

- 在不同的机器上运行功能测试：通过把测试分组打包成相关的套件，每一个套件可以很方便的在单独的机器上运行。这使得测试套件可以同步运行，并且可以通过添加相对数量的机器来提高测试运行速度。
- 功能测试回滚数据库事务：一个简单但有效的做法：如果在从头到尾的运行测试，就不要提交数据库事务。我们注意到许多不同的项目中都使用过该方法，它常常会显著提升测试运行速度。

- 把功能测试重构到更小的粒度：对于一个用例，在每个测试中只覆盖小的场景，而不是多个场景（或全部场景），这样我们就有了较好的粒度从而方便分割测试。我们发现较大的测试倾向于拥有更多的冗余信息——拆分这些测试可以得到更快的单个测试。
- 通过业务领域聚合功能测试：按照业务领域聚合功能测试，可以使开发者在本地运行一组相关的测试，而不是全部的测试。这使红–绿–红测试循环变得更快，并且可以避免某个测试套件减缓开发速度的情况。

注意，上述的先决条件是每个功能测试都要使用独立的数据库沙箱进行测试。如果两个功能测试使用同一个数据库的话，其中一个有可能因为另一个测试所插入的数据错误的报告“运行失败”。

- 丧失对功能测试的信心：任凭测试失败而不去修正，就从一开始丧失了功能测试这张用来阻止 bug 进入构建的安全网。这些测试就不会捕获 bug 并帮助我们保证代码是在期望的运行状态。失去这张安全网，对测试的信心就会丧失。编写测试的数量会逐渐减少，更严重的，测试会被删除然后最后全部被抛弃。
- 小改动引起许多测试失败：当很多测试失败的时候，一般假定是提交了大规模修改的代码。然而如果仅仅是小改动引起大面积的失败，那么肯定有大量的重复测试。

  为了解决这个问题，可以让每个测试关注功能的一个薄切片。当每个测试只关心某个功能的薄切片并且不与其他测试有重叠时，如果引入一个 bug，就只有一两个测试失败。诊断一个小测试为什么失败是非常容易的。因此，针对一个大型系统中的某个功能切片编写的测试，能为该业务流程的实例提供最好的反馈。
- 功能测试尝试–和失败–来捕捉单元层面的测试：如果功能测试没有缩减测试团队和客户所发现的 bug 数，那么问题可能是 bug 所在的层面与功能测试不同。
- 即使测试覆盖率看起来很高，功能测试也不能替代自动化测试。自动化测试对容易出现问题的代码进行测试，即使这些代码埋藏很深，用别的方式根本无法测试，因此自动化测试是功能测试的有力补充。自动化测试用来捕获单元层上的 bug，而功能测试关注交互层的 bug。
- 在没有适当重构的前提下编写了功能测试：业务逻辑被直接复制到测试准备数据中，并被工具使用。代码重复导致了维护的噩梦。不要这样做，测试代码和产品代码同样重要，使用重构消除代码的重复。
- 盲目追加功能特性：功能特性列表，与功能测试绑定，变成了预先编写的需求。丧失了反馈，您现在又回到了 Waterfall Will 的世界中了！

- 把功能测试作为衡量进度的工具：这是假设每一个功能都含有对等价值，但事实并非如此。您可以很简单的欺骗自己，认为刚完成 40 个正常运行的功能测试，就是交付了业务价值。但如果那些测试并不真地包含客户期待的业务价值呢？请关注业务价值。

架构异味

如果您在使用很好的工具和技术，但编写功能测试还是比较困难，那么问题的根源可能来自系统的架构。特别的，如果准备数据包含业务逻辑，而不仅仅是把测试详情翻译成方法的调用 ,那么您可能需要考虑下以下的异味。当一个功能测试很难验证一个单一 ,完整的用例时，我们也会认为这是一种异味。

功能测试能帮助业务逻辑进入正确的层级（在分层架构中）和正确的功能模块中。当业务逻辑放在错误的位置时，功能测试会暴露这个错位信息。

- 准备数据中包含了映射图形界面工作的业务逻辑：如果您发现在所编写的测试准备数据中，必须执行某段映射到图形界面工作的业务逻辑，这可能就有了架构异味。一般导致这种重复业务逻辑的原因是使用了经典包括展现层、领域层、持久层的三层架构。这种架构有时候不会把业务逻辑从展现层中剥离。实际上，图形界面在这样的创建过程中包含“控制”逻辑是很普遍的。

  例如，从一个账户到另一个账户（account1, account2）转账的简单图形界面通常会这样做：

```
Account1.withdraw($100)
Account2.deposit($100)
```

  这是一个简单的逻辑 ,但这是业务逻辑并非展示逻辑。如果在转账(account1, account2 ）的测试准备数据中有这部分逻辑，那么与用户界面相同的重复代码也就生成了（这很糟糕），同时也说明有未覆盖的业务逻辑在展现层中（这就更糟糕了）。

  当您遇到这种问题，解决办法是把这些重复代码放到一个公共的地方。这个位置就是在展现层和领域层之间包含控制逻辑的服务层。这样一来，功能测试能够帮助合理分割业务和展现逻辑，并鼓励一个新的逻辑层来持有控制逻辑。

- 针对某个模块的测试准备数据包含属于该模块的业务逻辑：当功能模块无法包含所有属于它的业务逻辑时，业务逻辑也可能在测试准备数据中出现。下面这个实例可以很好地说明这个问题。

假设我们子系统中有一个计税的模块，负责所有相关税务的计算。在引入功能测试之前，我们在编写完这个模块并认为功能分割比较不错。不幸的是，在项目的开发过程中，并不是所

有使用者都熟悉这个计税模块，导致有些依赖免税日期的“预算”逻辑编写于此模块之外。这个功能本应该放置在计税模块中；在某种程度上来说，计税模块的边界被破坏了。

当为计税模块编写功能测试的时候，我们会发现测试准备代码需要做一些依赖免税日期的“预算”工作。在这个时候一个有责任心的开发者会注意到代码的重复，并把测试准备数据和非计税模块中的计算部分重构到计税模块中。

功能测试会频繁地强化子系统的边界与职责。功能测试也会使您的系统模块更加完整。

很难对一个单一，完整的用例进行功能测试：对于遗留系统——也就是那些没有考虑到功能测试的系统——是极其难于测试的。有些时候即使是一个简单的业务实例，测试都很难实现。这是一个非常难于消灭的异味，只能依赖架构来解决这一问题。

在某些情况下，问题的根源来自于该模块假设多个用例同时运行。当您尝试分离某个测试用例时，您会发现依然需要为其他用例做准备，否则系统就直接崩溃了。

变化

以下是有效利用功能测试的多个不同的情况。

仅仅覆盖领域模型：采纳方法的章节关注验证服务层逻辑的功能测试，它们穿过领域层并一直向下到达持久层。并不是所有的功能测试都必须验证这些层次；事实上在《 集成测试框架:用 Fit 进行敏捷软件测试》一书中 Mugridge 和 Cunningham 对于编写功能测试只用于验证领域逻辑也存在争论。这样的测试确实有用，但并不能覆盖容易产生 bug 的子系统的边界情况。如果在一次开发者提交的周期内无法运行一次端到端测试，才应该考虑使用只针对领域逻辑的方式。

功能测试由全体人员负责：由于客户或分析人员处于撰写需求的最佳地位，所以他们应该编写功能测试。然而，测试人员、开发者也可以加入客户和分析人员，一起合作编写测试。

测试人员是编写测试用例的专家，他们可以帮助编写覆盖必要细节的需求。这可能需要开发者的帮助，利用工具让需求可执行。例如，在测试执行之前，集成测试框架（ FIT ）要求程序员编写测试准备数据。全体人员编写测试通常发生在实施功能测试的初始阶段，这是因为分析人员在学习像测试人员和开发者那样思考来建立领域语言。在以后的阶段，由全体人员负责编写的测试数量会逐步减少，取而代之的是担子落到了偶尔得到开发团队支持的分析人员肩上。

使用单元测试工具编写功能测试：某些团队使用单元测试工具比如 NUnit 或 JUnit 来编写功能测试。使用 xUnit 工具可以充分的覆盖代码，但是由于测试是使用客户和分析人员既不

会写也不会读的语言编写的，这样做会缺乏他们的参与，而且把需求转化成测试的翻译成了开发者的任务。不管测试的状态是通过还是失败，对于客户或测试团队也不是很明显。

完全依赖某一方来写测试，可能是一种好的变化，也可能是一种异味。如果客户有技术并且可以编写测试，这就是好的变种。否则，如果客户只是简单地陈述需求，再由开发者把这些描述转化成代码，那么基于 xUnit 的功能测试可能会逐渐变成项目的障碍，因为这样会过于关注覆盖率。虽然有这些测试比没测试强，但可以认为是一种异味。

在传统开发环境中的功能测试：到现在为止，有记录的功能测试经验都是基于敏捷开发环境的，但也并不是不能应用于非敏捷项目。关键点是功能测试必须频繁运行，并且和开发人员提交代码的周期相一致。这样的话，测试失败的原因很容易被发现。这样也会获得敏捷功能测试的好处，只是如果没有持续集成的环境，反馈的周期会比较慢。当在传统环境实施时，由于提交代码周期一般会更长一些，所以测试运行速度的重要性也随之降低。

参考资料

正如这里所定义的，功能测试通常与 FIT（集成测试框架）一起讨论：

- Gandhi, P., N. Haugen, M. Hill, and R. Watt. 2005.《用 FIT 创建活的详情说明文档》 http://www.Agile2005.org/XR22.pdf .
- Marick, Brian. 2002. Bypassing the GUI. Software Testing and Quality Engineering September / October 41–47.
- Mugridge, R., and W. Cunningham. 2005. 集成测试框架:用 Fit 进行敏捷软件测试. 电子工业出版社.
- Shore, Jim. 《Fit 的眼光》 http://www.jamesshore.com/Blog/A-Vision-For-Fit.html

12

集体代码所有权

开发团队中的任何人都有权利和责任修改任何一部分代码。

业务价值

虽然集体代码所有权是其他许多敏捷实践的辅助实践，但它通过提升团队中开发者的知识水平和责任心，会直接为项目带来更高的灵活性，并最终摆脱像“创可贴”那样不停打补丁的方式，形成完备的解决方案。

小故事

Scott ScrumMaster 的团队阅读过 Kent Beck 所撰写的《解析极限编程——拥抱变化》之后，决定不实施集体代码所有权。其实，他们感觉让每个人都写 GUI 代码会是一种浪费，效率也会变低，比如不专业的人会写出很糟糕的 UI 界面。或许还有其它什么顾虑吧，总之他们就这样决定了。

Scott 的团队首先引入了迭代和自动化测试。很快，他们发现采用原先分头进行的开发方法，在每个迭代的后期总会导致不同子系统代码的很难集成。

这样，两个迭代过去了，每次都有若干预期目标没有完成。于是，团队决定尝试集体代码所有权。这直接促使团队成员在迭代过程中，自发使用结对编程来分享知识。而且，迭代的预期目标也变得很容易完成，很大程度上消除了项目中的瓶颈资源。

适用情况

你加入了一个对开发者按照传统专业化方式化分的开发团队。比如，一些开发者负责 GUI，一些负责中间层，还有一些负责数据库开发。团队中的开发者各自拥有自己的代码，视作宝贝一样保护起来：“谁也别动我的代码”。

团队正在一个接一个地尝试敏捷实践，逐步从预先的静态设计过度到不断演化的设计。为了

保证系统正常运行，团队成员需要修改多个部分的代码，而不是仅仅修改自己负责的那一部分。

或许你想要减少团队中的瓶颈资源，不想让团队因为某一个人员的问题而导致整个团队能力的缺失。随着时间的推移，你也希望能够轮换团队中的人员。

动因

集体代码所有权可以解决在很多其他的敏捷方法实施过程中所出现的问题。

对系统进行任何形式的修改都可能引起其它部分发生变化。很多敏捷开发实践允许而且鼓励变化。实际上，《解析极限编程》一书的副标题就是“拥抱变化”，因此这句话也成为早期敏捷社区的一个口号。

- 为了保证针对整个系统的自动化测试都正常通过，开发人员需要经常性地修改那些不是自己编写的代码。
- 频繁的重构会导致同样的问题：你写的代码会依赖其它部分的代码，因此就需要修改系统中一些不是你写的代码。
- 如果不能够修改系统中那些非你所写的部分，那么演进式设计就会受到很大的限制。
- 持续集成保证系统不停地集成并且能够运行。这样，对影响系统各部分的修改必须彻底完成后才能够提交至代码库中。

因此

为了激活许多实践的“敏捷性”，开发者必须有权改动系统中任一部分。这种行为不应该脱离测试这层安全网——自动化测试——可以在开发者不熟悉的环境下提供支持。大家可以相互修改对方的代码，代码逐渐变为公有。在完成一个任务时，对系统中某个部分的变动，如果引起了另一部分的修改，应该鼓励开发者去修改所有相关的部分，必要时也可以找到相关的人帮忙完成这些修改。团队中每个人都拥有项目代码，允许和鼓励大家修改所有需要修改的代码，这样的实践即是集体代码所有权。

采纳方法

你将会发现其他许多实践都需要集体代码所有权的支持。所以在其它实践有需要的时候再考虑采纳该方法。一旦需要集体代码所有权时，那么：

1. 定义共有代码的范围。一般来说是整个系统，但有的时候如果你们属于同一个软件产品下同时存在的众多团队之一，那么范围就可能是该子系统。
2. 【极力推荐】考虑采用自动化测试作为安全网，来缓解开发者熟悉系统中新增部分的痛苦。
3. 设置一条原则，如果开发者 B 需要修改某部分代码，作为该部分的专家，开发者 A 不能拒绝。这样，开发者 B 就可以对这些还未熟悉的代码进行安全地修改。
4. 【极力推荐】采用结对编程来传播系统中不同部分的知识，并且经常轮换结对。
5. 鼓励开发者承担系统不同的部分的开发工作，哪怕他们并不熟悉该部分。这同样有助于在团队中传播经验。

但是

这也是一种效果不是特别直观的实践 ,经验丰富的开发者需要经历一段时间才能认同它的价值。对一个总想保住系统某部分秘密的“专家”来说，确实有点“威胁”。

- 开发者不能从心态上放开“自己的代码”，当发现别人修改他们最初编写的代码时，容易变得防护心理很强，总是防备别人，或是情绪失控。这些都是成长的烦恼，应该根据每个人的不同反应，分别解决。
- 开发者对其他人的建议缺少沟通或尊重，设计就会被改来改去，一塌糊涂。开发者只应该在需求导致了变化时，才进行设计调整。你将经常发现某个开发者出于自己的偏好，将设计改回了所谓的“正确的道路”，而不是为了满足特定的需求。

变化

在大规模团队中，因为有太多的技术需要学习，所以出现通才的可能性不高。解决这种问题的方法是，逐渐由培养通才变为培养多面手。除了自己主要关注的部分之外，开发者也会去学习相关子系统的技术和代码。

参考资料

集体代码所有权是极限编程原书中 12 个实践中的一个。

- Kent Beck 解析极限编程--拥抱变化，人民邮电出版社，唐东铭译，2002

- Kent Beck， Andres, C. 解析极限编程--拥抱变化（第二版），电子工业出版社，雷剑文、 陈振冲、李明树译，2006

第三部分：模式组合

第十三章 实践的组合

第十四章 演进式设计

第十五章 测试驱动开发

第十六章 测试驱动需求

13

实践的组合

极限编程(XP)是一个众所周知的敏捷开发过程。1999 年 Kent Beck 首先提出了极限编程，同时给出了十二个需要一起实施、缺一不可的实践。这些实践相互支撑，不能丢弃或修改其中任一实践。这些实践都是“生成式实践”——即作为一个整体实施所带来的价值，比单独实施各个实践的价值总和还要大得多。当一起使用这些实践时，他们会相互作用，产生合力，带来奇效。

Kent Beck 从没有说过“十二个实践缺一不可”。但是，敏捷社区在早期一直把这当作实施“真经”，直至后来 Kent 提出了测试驱动开发——它是仅仅面向单个程序员的一组编程实践，是十二个实践的一组子集，

事实是，有些实践之间会相互作用，当您把其当成一组进行实施时，会产生额外的效力。我们称这些实践为“生成式”实践。还有一些仅是依赖于其它某个实践，比如没有自动化测试就无法做到真正的重构。不要把这种依赖关系和“生成性” 这两个不同的概念混淆。

由此引出了第三部分：敏捷实践组合。它由一组 “生成式”实践组成，但不仅仅是这些单个实践的集合。在这个组合中，各个实践都有着统一的着眼点。因此可以根据这些组合，来确定下一步应该选择哪一个新的敏捷实践。这并不是说，您该使用这些组合来驱动整个实施过程——业务价值和异味还应该是实施策略的驱动力，但组合会让您的实施更上一个台阶。

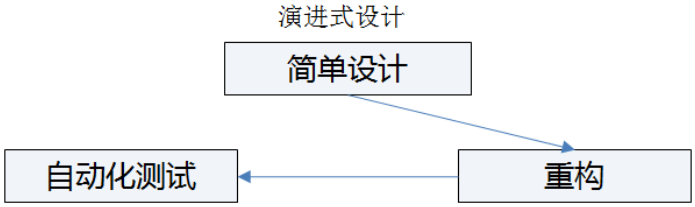
本部分中对组合的描述如同之前的实践一样，依然使用“模式”方式。因此，您会看到他们各自的实施部分，指导您迭代地采纳它包含的各个实践。

迄今为止我已完成了一些技术组合模式，这最后一部分包括了其中三个最常用的技术组合。除此之外，还有很多敏捷实践以及“生成式”实践组合。如果您感兴趣，请定期访问敏捷实施模式 Wiki（www.Agilepracticepatterns.org），那里会有还在挖掘之中的敏捷模式、组合。

14

演进式设计

为了拥有真正意义的迭代开发过程 [15]，系统的设计必须跟随着新需求的实现而逐步演化。先进行简单设计，保持设计简单化，仅当设计无法再支持新需求时，才进行设计更改，这样就做到了演进式设计。这种设计更改机制称为重构，其实施基础是自动化测试。



业务价值

与主要的实践简单设计类似，演进式设计会缩短上市时间和降低成本。这组实践之间相互配合，但又都遵循一个核心原则——仅在必要情况下更改设计。这三种实践一起实施，会产生合力，极大地提升这些价值，并为产品带来灵活性。此外，演进式设计还会延长产品寿命。

小故事

ScottMaster 最初组建的敏捷项目团队中，Amy Architect 是成员之一。她是一个动手能力很强的架构师，经常与团队中的其他开发人员一同编程。她知道在 ScottMaster 的团队中不会有架构师这一角色，也准备好了迎接这种挑战。团队的其他人员都知道她软件开发经验很丰富并且非常能干，所以都非常高兴她的加入。

因为团队开发速度需要提高，他们决定采用迭代开发和简单设计。而且，他们也明白，如果日后想要有效地实施简单设计，肯定会需要自动化测试，所以他们也采纳了测试后行开发——测试先行开发对于他们还太过陌生。

[15] 而不是仅仅划分了时间段的瀑布过程。——作者注

这是团队的第一个敏捷项目，出于习惯他们会跑去问 Amy，依赖于 Amy 做出设计决策。Amy 非常乐于帮助大家，但却习惯于给出一个通用性的设计以保证灵活性（与我们中的很多人一样）。最终结果是，设计的确很优雅，但对于目前的需求来说过于复杂。若干迭代后，一次 Amy 与 Jim Jr Developer 结对编程时，她尝试向 Jim 解释，相关部分是如何使用“Template Method 设计模式”来实现一组算法的。Jim 没有听明白，Amy 只好去掉抽象代码，直接把实现插入进来。“哦，我明白了。但是，这模版方法是为将来考虑的吧？我们现在应该使用简单设计！”。于是他们删除了复杂的代码，改成更为简单和直接的方案。Amy 和 Jim 修改了设计，但保留了所有的行为，所以测试依旧可以通过（这其实是在重构代码）。

这件事情让 Amy 重新反思她花在复杂设计上的时间。有两点让她深有感触：由复杂到简单的转变十分容易；测试给了她自己极大的自信心，设计改动之后不用老是担心系统是否依旧能稳定运行。于是，在结对编程的过程中 Amy 开始删除复杂代码，并且移除了结对伙伴和她都认为是过度抽象的代码。又过了若干迭代，设计变得越来越精益。而且，出乎她意料的是，系统中复杂的那些部分竟然与她一开始的设想不同。

软件开发团队依旧向 Amy 寻求设计上的建议，但是她给出的建议已经和之前的建议有所区别了。她仅考虑当前需求，做出一定的通用性设计，总是给出显而易见的解决方案。几个月后，她看到了在自动化测试、简单设计和重构之间的密切配合下，诞生了一个如此精益又优雅的系统，而且可维护性比其他任何在开始阶段想到的方案都要好。Amy 的经验对团队来说依然很宝贵，只不过她现在更多的是引导大家而不是直接决策。

适用情况

有些实践可以应用在所有类型的开发项目中，该实践就是其中之一。因此只要您在一个软件开发项目中，就可以使用该实践。下面这些情况下，该实践会显得“恰好匹配”，但并不是必须的：

缩短产品上市时间对您的团队来说非常重要。

项目组的大部分人对项目所要使用的技术比较陌生。（比如，ATG-Dynamo 团队的第一个 JEE 项目，虽然大家对于构建 Web 应用程序很有经验但是对于 Java 或者 JEE 却很陌生。）

动因

因为主要源自预先设计，所以这可能是所有传统软件开发项目中都会存在的问题：

- 在传统过程中，“预先设计”基于这样一个假设：变更成本会随着时间成指数级增长。严格的自动化测试与重构降低了在开发过程中改变设计的成本。

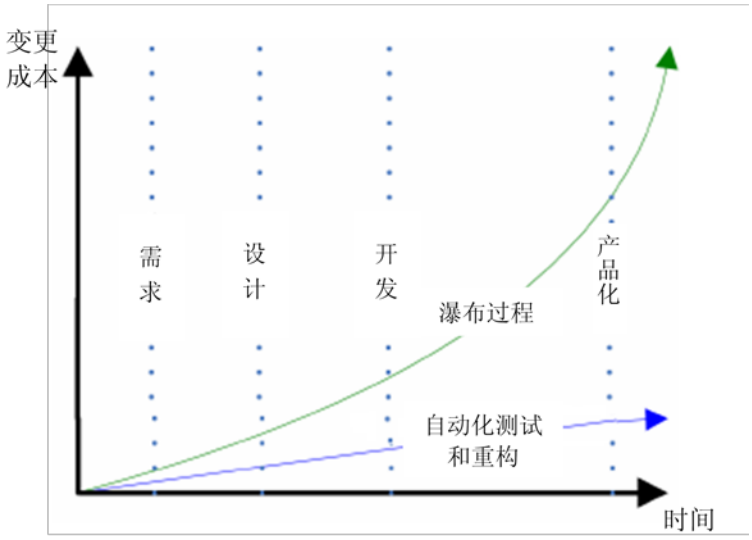


图 3 瀑布过程和演进式设计中的变更成本曲线

- 仅仅是画几张设计图，远不能展现出预先设计的成本：
  - 需求和规范必须详细到足以以支持设计决策。
  - 必须做出设计，而且还要让实施开发的团队充分理解。
  - 尽管软件系统非常复杂，难以预先设计，但系统还必须得开发出来。
  - 过度的通用性设计会带来“设计的累计开销”，而且开发人员也并不一定会经常用到它。由于开发人员必须理解设计框架，在符合其设计要求下扩展功能，因此每一项开发任务都变得很复杂。
  - 基于预先设计，软件在实现时需要考虑到种种复杂性因而很容易出错。
- 软件问题没那么直接，通常都会比较复杂。知识是团队开发软件的有力武器。但这种知识是在开发系统过程中逐步获得的。之所以能够在以后获得更好的设计方案，是因为您和团队在当前项目中尝试了各种方法来开发系统，在此过程中您们学到了宝贵的知识。

因此

不要做任何预先设计。不管您经验有多丰富，不要过多考虑未来。不停地再创造。使用自动

化测试，以使您的团队能够做到由需求驱动设计更改。先进行简单设计，仅在设计无法满足需求时，才对设计进行重构。重构设计时，依赖于现有的测试：如果您破坏了什么功能，测试会发出警告。不要在系统上打补丁，如果 设计不再适应新的需求，进行重构。

通过同时使用自动化测试、简单设计和重构三种实践，您会获得：

- 对于给定的需求，因为总是用最简单的设计，所以交付速度会更快。
- 在整个开发过程中持续在学习上投入，会获得更好的设计决策。相对于预先设计，这样产生的设计会更为精益。同时，由于设计更易于理解（简单设计），又有自动化测试作保障，系统更改起来也更容易，因而系统的维护成本也会降低。
- 对于复杂问题，因为不用一次性就全部解决，这使得我们能够成功处理更多更复杂的问题。

采纳方法

可直接从简单设计入手实施演进式设计。但简单设计需要重构作基础，重构又依赖于自动化测试，所以：

1. 通过阅读自动化测试模式部分，决定您要采用哪种测试方式：测试先行开发还是测试后行开发。
2. 实施简单设计，同时实施自动化测试。
3. （强烈建议）考虑采用结对编程来辅助上述实践。当实施这些纪律性很强的实践时，结对伙伴会时刻防止您松懈下来。
4. 阅读“重构”部分并准备好实施重构。当设计无法再满足需求的时候，尝试修改设计。

在这个时候，您已经成功实施了上面三个实践。现在您需要关注他们在实施过程中各自的质量。团队成员是否真正在使用简单设计，还是像小故事中 Amy Architect 那样在过度设计？是否在每个任务前后都考虑了重构？如果答案是否定的，那么即使单个实践都在应用，但因为缺乏相互配合，所有无法使设计得到演化。您可以尝试以下步骤，以做到真正的演进式设计：

1. 每周进行一次内部代码复核，让某个开发人员展示他自己完成的代码：
    a. 从简单设计的视角审视这段代码的设计。
    b. 如果这段代码设计过于复杂，给出如何修改的建议。

c. 尝试记录导致设计复杂的原因录。是否因为预先设计？是否因为重构不够及时？

2. 注意是否有大量缺陷从自动化测试“逃脱”至 QA 测试。是否大部分缺陷来源于重构？是否是自动化测试不够？

但是

演进式设计不是碰运气。它要求很强的自律性，而且三个实践协同实施。三个实践相互密切依赖，一个实践的松懈常常会导致另外两个实践也跟着出问题，下面是相关的细节：

- 低劣的自动化测试会影响团队重构的能力。为了新需求改变设计，完成之后所有测试也都通过，提交以后却导致系统崩溃。没有一张测试的安全网作保障，开发成本会急剧上升，因为令人头痛的事情又都出现了：拼命找缺陷，不停地修复缺陷，有人在错误代码之上添加了新功能等等。
- 没有经常重构 :设计已经不能支持新需求了 ,但开发人员还是勉强把需求塞进去。这时，您不得不处处打补丁，代码变得难以理解，也不知道怎么用才是对的，修改代码的成本又重新走高。 虽然从简单设计开始，但却无法做到演化设计，只能停滞不前。这个时候，您将不得不进行若干更为困难的大型重构。

演进式设计可能会导致架构上的不一致 ,这是因为每一组团队都针对相似的问题演化着自己的解决方案。可以通过以下方式解决这一特殊问题：

- 在团队中找一个架构师作为“代码论”的维护者。这个角色通过阅读代码、与其他开发人员结对，以及举行专门的设计审查来保证设计的演化。就像“异花授粉” ，她会引导所有的设计不断向应用程序的“代码论”靠拢。
- 团队规模小一点，会比较容易形成一致的核心架构设计。可以先构建一个小点的团队，针对最难的问题，实现一小部分功能。待设计基本成型后，扩大团队成员，进一步演进这个架构。这在 Elshamy 和 Elssamadisy 的文章《Divide after You Conquer》有详细论述。
- 发挥架构师的核心作用，让她了解到所有主要设计决策，必要时协调。

变化

正如“但是”部分中提到的，在大型项目中演进式设计的问题之一是在设计不统一。有一种常用的技巧是“Divide after You Conquer”，即在大型项目中，先由一个核心团队为整个

系统构建出一个基本的支持框架。这使得架构既能围绕真正的需求进行演化，同时因为是小团队，一致性也得到了保证。

参考资料

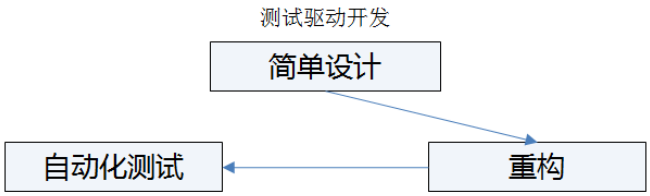
虽然主要的软件过程书籍中，都没有很明显地提出演进式设计，但常常会在说到简单设计时讨论它。因而，参考资料与简单设计一样：

- Kent Beck ，《解析极限编程--拥抱变化》人民邮电出版社，唐东铭译，2002
- Kent Beck， Andres, C. 《 解析极限编程--拥抱变化（第二版 》，电子工业出版社，雷剑文、 陈振冲、李明树译，2006

15

测试驱动开发

测试驱动开发是一组非常有效的实践，把自动化测试放到了开发的最前线，用可测试性来主导设计。这种形式的开发过程会产生极为松耦合的设计，因而在需求发生变化的时候，（相对来说）要易于演进。



业务价值

测试驱动开发几乎涉及本书中的所有实践。同时，因为组合自身的特性，会增强这些实践的业务价值。最值得一提的是，测试驱动开发能够提高产品质量，缩短上市时间，显著地延长产品寿命。如同演进式设计一样，它也会提高灵活性，减少开发成本。

小故事

在 Scott ScrumMaster 团队中，有开发人员 Cindy Coder，Dave Developer，Waterfall Will, Uthman Upfront Design, Amy Architect 和 Jim Jr. Developer。 他们正在使用测试驱动开发，尽管他们还没打算从一开始就实施全套实践，他们一个接一个地实施。

因为自动化测试的好处很明显，团队最先采纳了自动化测试。一些开发人员做到了测试先行开发，而另外一些更喜欢测试后行开发。因为要确保所有的测试时时刻刻都能运行，这种开发方式很快就促使大家采纳了集体代码所有权。对于大多数更有经验的开发人员来说，简单设计的价值并不是那么明显，毕竟他们很难一下子彻底放弃预先设计，因而它的引入要迟一些。但是，随着测试的快速增长和重构习惯的养成，简单设计就变得越来越有必要。采纳简单设计后，过了几个迭代，整个系统的设计逐渐开始演变，越来越“精益”了。他们采纳的

这一组实践，相互配合，互相促进，放大了各自的价值——不但写出了更好的代码，而且让整个系统也变得更精益了。在以前的非敏捷团队中，从来没看到过这样的事——系统从来都是因为不断增加的信息量而变得臃肿不堪。

团队还没有一个可以长期稳定运行的持续集成构建，因为这还需要 Bob BuildMaster 花上一段时间，确保构建跑得够快才行。但是，在提交代码之前，他们的确做到了总是在本地先运行所有的测试。当 Bob BuildMaster 找他们商量构建工具计划时，他们给予了全力支持。添加了这个工具后，他们就能够在本机上运行完整的集成测试，这样整个开发过程的速度和质量又上了一个台阶。

适用情况

您的团队想要大幅提高工作效率。想要开发速度能更快一点，缺陷更少一点；当需求改变的时候，设计也能随着变；软件生命周期中的整体成本也能得到降低。

您知道想要得到这样一次“大变身”，必须对软件开发方式进行一次重大改造。您和团队都已经想好了，愿意花上三个月到一年的时间来学习这些技能。您也愿意付出学费，在团队掌握这些技能之前，您的开发团队没有办法交付跟过去一样多的东西。

动因

测试驱动开发是若干实践的组合，因而也具备所有相关实践——演进式设计，持续集成，集体代码所有权——的动因。也就是说，测试驱动开发将能最终解决所有相关的软件开发问题。但是，究竟哪些问题会迫使开发团队采纳测试驱动开发一整套实践，而不是其中的单个实践呢？

- 演进式设计的核心是在开发软件的时候，尽可能地把在设计上的投资最小化。这种技巧借助测试作为工具，让设计通过重构不断演化。测试并不是设计背后的驱动力。
- 如果测试不是软件开发过程的核心，那么测试就不一定能充分验证整个系统。
- 测试是一个采样的过程。对程序的整个状态空间都进行测试是不现实的，但是我们可以测试到程序的大部分细节。因此，必须确保所写的每个测试都物有所值。
- 理想状况下，测试应该是一种可执行的需求。
- 有些重构会牵扯到整个系统，导致很多部分都要跟着改动。要在做这种大规模改动的同时，保证测试 100%通过，必须确保其对所有子系统的影响都得到妥善处理。

- 一次“成功的构建”应运行系统中所有可能存在的测试，并全部通过。
- 为了能够持续改进，敏捷实践非常注重频繁反馈，有一些成功的实践可以采纳，持续集成就是其中一种。只要采用了持续集成，就很难再放下了。
- 在一个团队协作环境中，当有成员需要对设计做出大规模改动时，因为没有“预先”设计的规范可供遵循，如何实施演进式设计会很有挑战。仅仅依靠演进式设计，无法填补取消预先设计所带来的沟通“空洞”。设计随时都在发生着变化，团队必须及时交流，充分理解当前的设计。

因此

如果想要大大缩短上市时间，提高软件的质量和灵活性，降低软件开发成本，请考虑采用演进式设计，持续集成和集体代码所有制。本书的其他部分中已对这三个实践模式分别做了描述。

把编写测试当作开发过程的首要任务：

- 将所有测试作为持续集成的一部分。把成功构建的定义修改为：包括所有的自动化测试，并都能成功通过。
- 使用集体代码所有制提高重构的能力和速度。构建需要通过所有测试，鼓励开发人员为此进行必要的改动，即使是改动其他子系统中的代码。
- 演进式设计过程中以测试为中心，而不是设计。让测试来驱动开发。设计要保持简单，它可以不断演化但只能由测试来驱动。通过这种方法，就能确保每行代码都有对应的测试。这种方式会使设计更为松耦合，从而延长了系统生命周期和灵活性。

在以测试为中心，成功采纳了这些实践组合之后，团队的交付质量会好很多，灵活性逐步上升，成本相应也会降低。但请记住，必须投入时间去学习这种新的开发方式，而且在实践中不断磨练，才能有此收获。

实施

根据自动化测试编写方式的不同，有两种广为流行的测试驱动开发形式：测试先行开发和测试后行开发。前者要优于后者，但其实施难度也更大一些。

1. 在实施计划中预留一大段时间，以使不同的实践活动能相互高效配合，这种组合的强大效力得以发挥。对于小团队，一个新项目，这可能要花费 3 个月的时间；对于大团队，

且依赖于遗留代码又没有任何测试的项目，可能要花费长达一年的时间才能充分看到该模式的优势。

2. 信任团队成员。鼓励团队成员改动代码，相信这些改动是必要且有价值的，给他们足够的学习空间。创建一个激励大家去采纳实践的工作氛围。[16][1]
3. 从演进式设计着手，整个团队都要参与。同时，试探着开始实施持续集成。
4. 【强烈推荐】引入结对编程作为演进式设计及其他相关实践活动的辅助实践。结对编程能够促使大家总是遵循原则去编写测试，一个人写代码很容易因为懒惰而违背原则。另外，结对编程也会促进集体代码所有制的实施，在整个团队中传播知识。可利用结对编程作为一种实施工具，实施成功之后您可以选择继续结对或是放弃结对。
5. 当演进式设计导致大规模重构，由此导致失败的测试比所写的功能代码还要多时，使用集体代码所有制。
6. 在团队还未完全习惯于写自动化测试之前，可以使用持续集成，但其实施往往依赖于高度自律。关于这一点，持续集成部分阐明了如何向整个团队介绍这种观念、相关工具和实践活动等。
7. 当您感到团体已经能够熟练应用演进式设计、持续集成和集体代码所有制时，回过头检查整个开发过程。重新审视各个实践，尤其是以测试为中心的这些实践：
8. 如果您正在使用测试后行开发，可以考虑用测试先行开发取而代之。否则，就要对测试代码进行周期性的审查。

确保是测试在驱动设计，而不是设计决定测试。这是松耦合设计的主要驱动力之一。

已有代码的测试是否可当作文档来读？好的测试驱动开发的标志是，测试可作为产品文档。

但是

作为一组实践，测试驱动开发可以向客户创造很多价值。这就要求所有组员都要坚持不懈地写测试，不断地演化设计，维护持续构建，和其他组员一起修复由于重构所破坏的测试。这实际上意味着：如果放弃任何一个实践，那么该实践组合的强大效力就不复存在了。如果其中某一个实践没能坚持下去，会发生的各种情形有：

- 如果放弃了集体代码所有制，那么由设计改变所造成的失败测试就不能及时得到修复，

[16] 这虽然是一种略带感情色彩的方式，但无论什么实践，它的确是最成功的实施策略之一。如何才能创造一个团队成员都乐意使用这些实践的工作氛围呢？——作者注

从而严重阻碍了演进式设计的实施：

- 这可能导致提交的代码中有失败的测试，而这是测试驱动开发的“大忌”（至少是很大的问题）。这也会造成测试不能 100%全部通过，从而破坏了持续集成。
- 这还可能导致代码不能正常提交，或者必须把失败的测试移交给其他人来修复。这拖延了开发速度，让有能力修复测试的人变成了瓶颈资源。
- 如果演进式设计或任何依赖实践遭到破坏，那测试驱动开发就丧失了一半效力。参照演进式设计组合中的细节，保证其得到正确应用。
- 如果持续集成遭到破坏，那么整个系统在若干改动之后是否完好就无从知晓了。演进式设计可能还能继续，但进展速度将会很慢。
- 由于重构导致了测试失败，而对于相关代码又不太熟悉，这种情况下要修复这些测试仅仅依靠集体代码所有制可能还不够。即使我们鼓励所有开发人员积极改动代码，但并不是所有开发人员都具备相应能力。引入结对编程让知识得到传播和共享。
- 采用测试驱动开发的目的是开发出更好的软件。软件实际上是否真正满足客户需要，并不是这个组合所能解决的问题。不要误以为，实施这组实践就是在开发对客户更有价值的软件。看看其他实践如功能测试、测试驱动需求以及客户作为团队成员，它们将帮助您开发出对客户更有价值的软件。

变化

测试驱动开发（TDD）使得大家趋向于统一的源代码库——取消所有的发布版本、补丁等对应的分支，使用单一的源代码库。在使用该模式时，大家经常基于同一个代码库进行集成，编写测试，保证测试通过，这无形中使单一代码库成为可能。通过使用单一源代码库，您会省出大量的时间去完成别的任务。

参考资料

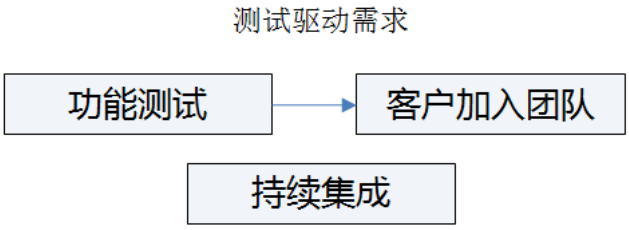
测试驱动开发的资料有很多，这里列出其中一些：

- Astels, David. 《测试驱动开发：实用指南》，中国电力出版社，崔凯译，2004
- Beck, Kent. 《测试驱动开发（中文版）》，中国电力出版社，孙平平等译，2004
- Feathers, Michael. 《修改代码的艺术》，人民邮电出版社，刘未鹏译，2007

- Jeffries, Ron. 2004. Extreme Programming Adventures in C#.
- Redmond, WA: Microsoft Press.
- Martin, Robert C. 《敏捷软件开发：原则、模式与实践》，清华大学出版社，邓辉译，2003。

16

测试驱动需求



业务价值

测试驱动需求能够增强产品的市场价值,并且通过在客户和开发团队之间创建一个紧凑的沟通和反馈循环 ,从而极大地提高了项目进展的可见度。该组合把功能测试与持续集成相结合，很好地促进了反馈。同时 ,作为一种系统测试的方式 ,测试驱动需求也促进了其他业务价值：产品上市时间得以缩短；产品生命周期得到延长；产品质量有了提高；整个应用的灵活性也得到增强；软件系统总体成本也得以降低。虽然测试驱动需求的价值常被低估，但它是一组真正有价值的实践。

小故事

Aparna Analyst ,Tina Tester, 和 Cindy Coder 三人已经在进行着敏捷开发——更具体点说，他们在过去的六个月中实施了迭代、测试驱动开发、简单设计和持续集成, 并已驾轻就熟。他们极大地提高了开发效率，与此同时还显著地降低了 bug 的数量。但还是存在一些 bug，依然有改进的空间。在上一次的回顾会议上，他们达成共识，认为这部分仍需进一步改进。

他们想，既然 TDD 适用于开发人员，那也会适用于整个团队。如果 TDD 让开发人员受益，那么在迭代开始时为需求编写可执行的自动化测试，也会对整个团队带来好处。他们意识到这如同 TDD 一样，不仅仅是开发人员的责任，需要整个团队的参与。Aparna, Tina, 和 Cindy 自告奋勇，准备在 Caleb 指导下试试看。

团队在第一个迭代的末尾阶段开始尝试这一组实践，Aparna 感到非常头疼因为她不得不编

写非常详细的需求文档。Tina 惊奇地发现，预先写的测试竟与她本在实现完成后才要写的验收测试非常相似。Cindy 意识到植入支持代码，让 FIT（Framework for Integrated Tests）工作起来并不是小菜一碟，甚至可能需要巨大的工作量。要完成这些支持代码，困难之处在于必须重构 UI 中的业务代码。虽然不太情愿，但她不得不承认这一点。Caleb 早就想过重构这部分代码——一方面这可使 FIT 运行起来，另一方面会改善当前的系统架构，对适应未来变化中不断演进也铺平了道路，因此他很高兴看到这个结果——团队自己发现了这个问题，还找到了解决方案！他们开始体会到了敏捷开发就是持续改进。

适用情况

您所在的项目中有这样的情况：有一个客户很乐意作为团队的一员，更多地参与到团队中，而且客观条件也允许他这样做；您的团队愿意去改动已有代码，哪怕是改动很困难；虽然有一定的学习曲线，您觉得值得为此付出努力。另外，下面列出了其他一些问题。虽然它们不是必要条件，但如果在你的项目中存在，则更能说明这个模式适用于您：

- 您的团队是一个分布式团队，需求来源于一个地方，而开发工作在另外一个地方进行。
- 您需要大幅减少缺陷数量。
- 您需要显著地缩短产品上市时间。
- 您要保证开发完成的软件系统，能解决客户真正的问题。想要它在市场上更有价值（第一部分中的业务价值之一）。

动因

测试驱动需求所能解决的问题恰恰是对应单个实践所能移除的问题的集合。比起单个实践来，这个组合效力更强的地方体现在：

- 功能测试不是持续集成构建的一部分，常常悄无声息地就失败了。当发现测试失败时，已经不知道是由哪次提交（通过持续集成在后台运行了多次构建）导致失败的。这种情况下，缺乏及时的反馈导致了部分功能测试的失败。从功能测试开始就无法满足质量改进的要求，导致功能测试沦为“二等”测试。
- 即使有客户作为团队成员，但如果没有功能测试，需求和代码之间的“翻译”过程可能还是会出问题，客户说的是一件事情，开发人员却理解成另外一件事情。
- 如果是分布式团队，客户和开发人员不在同一个地方，需求和代码之间的“翻译”问题，

会更为突出。如果存在文化差异，情况则会更为严重。

- 没有持续集成，功能测试会悄无声息地失败，并且无人过问。

因此

保证有一个客户作为团队成员，与开发人员紧密配合编写功能测试。让客户以功能测试的方式编写需求，而不是象之前那样编写详细的需求文档。这样以来，即使是在分布式或者多文化的团队，客户和开发人员间就有了一个具体明白、无二义性的沟通方法。同时，保证持续集成不仅包含自动化测试，而且包含所有功能测试。要使其可行，建议您参照功能测试模式中的提到的技巧，确保测试运行得足够快。

开发者的任务是开发系统的一小部分功能，让功能测试通过，并且建立适当的基础框架以保证测试能够正确运行。当新的功能测试通过时，开发者会在本地运行所有的自动化测试和整个系统的功能测试。成功之后，把新代码提交至版本控制代码库。由于持续集成中会运行所有功能测试，因此整个团队在过去所有迭代中实现的需求，都将会被覆盖到。

如果把这些实践当成一个组合，并按照上述方式使用，即成为测试驱动需求。需求会以测试的形式编写，测试驱动开发中的快速反馈循环也会扩展到整个团队。

实施

当然，对该组合的实施需要依赖于每个实践的实施情况。客户作为团队成员应该在功能测试之前实施。持续集成可在任何时候开始实施。测试驱动需求不仅仅是简单地全部采纳这三个实践。您必须将它们联系起来，让它们协调一致地工作：

1. 真正把功能测试当作需求来使用，客户必须学着编写测试。这通常需要很长一段时间。这常常需要从技术人员那里寻求帮助，或由测试人员或开发人员在开始的若干迭代中与客户进行结对编程，直到大家都熟练掌握了为止。（更多细节参考功能测试模式）
2. 第二个要点是，在客户与开发人员之间使用测试作为一种沟通“语言”。这是非常聪明的做法。随着这些实践的采纳，做好演进这种语言的准备。
3. 尽您所能保证功能测试在持续集成的每一次构建中运行。相对于自动化测试，这些测试运行速度会比较慢。需要多关注功能测试的运行速度，避免因其导致持续集成构建失败。

但是

与其他技术类组合相似，测试驱动需求要依赖于所有相关实践，才能良好运作。任一个实践出了问题，就会影响这一组实践。因此，请查看功能测试、持续集成和客户作为团队成员的“但是”部分。[17]

最常见的问题是功能测试运行缓慢。这会引起两个问题：

- 开发人员在提交代码之前不再运行所有测试，以致于提交经常导致持续集成失败。
- 持续集成需要很长时间，如果集成持续失败，很难找到修复失败测试的负责人。

为确保功能测试能融入到持续集成中，您需要保证测试运行速度足够快。首先，整个团队必须达成共识，功能测试是首要的开发实践，不能可有可无。这样当不可能放弃测试的时候，团队就会找出创造性的解决办法。主要问题还是提高功能测试的运行速度，这样开发人员在提交代码之前就可以很快地运行这些测试。如何提高测试运行效率？下面是一些很有实效的策略：

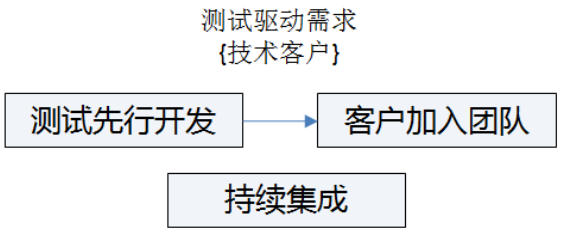
- 在独立的机器上运行功能测试。把测试组织成不同类型的测试包，在不同的机器上分别运行这些测试包，这将有效地并行运行不同的测试。依赖于测试机器的数目，测试机器越多，执行效率也会越高。
- 功能测试使用数据库事务回滚。这是一个简单却有效的想法：当测试完成的时候，不要提交数据库事务。我们看到过在几个不同的项目组中都用到了这个办法，而且效率得到了极大的提升。
- 重构功能测试，使其粒度更细。不是在同一测试中覆盖某一用例的若干场景（甚至所有场景），而是把测试分解，在每一个测试中只测试单个较小的场景，这样测试就有了更小的粒度。而且，较大的测试往往会有更多的冗余。测试分解后，会让每个测试运行得更快。
- 根据业务领域对功能测试进行分组。这样，开发者在本机就不用运行所有测试，只是运行相关的某一组测试。如此一来，我们能够通过红-绿-红测试循环更快地得到反馈，并且避免了因测试包而阻碍开发效率。

另外，使用独立的测试数据库运行每一个功能测试。如果两个功能测试使用相同的测试数据库，其中一个测试可能会因为另一个测试插入了数据而误报失败。

[17] 很遗憾，本书未包括客户作为团队成员，你需要查阅其他的资料，了解与该实践可能遇到的问题。——作者注

变化

如果客户有技术背景的话，测试驱动需求可以使用 XUnit 框架进行编写。对于有技术背景的客户来说，比起 Fit 表格，使用代码编写测试更为直观。但如果客户没有技术背景不能动手写测试，只是告诉开发者要什么，然后让他们编写，那么这种方式就不再有效，而是一种异味。



参考资料

Ron Jeffries 使用 "Running Tested Features" 作为跟踪项目进度的一项重要指标，这是功能测试和持续集成结合使用：

- Jeffries, Ron. "Running Tested Features."
- http://www.xprogramming.com/xpmag/jatRtsMetric.htm .
- Joshua Kerievsky 描述了一个几乎与测试驱动需求完全一致的一个实践，名字叫用户故事驱动开发：
- Kerievsky, Joshua. "Don't Just Break Software, Make Software."
- http://www.industriallogic.com/papers/storytest.pdf .

结论

您已读完这本书了——我们得举手相庆！希望您已经根据自己的组织、环境及开发团队的情况，初步创建了一个敏捷实施策略。记住循序渐进地实施，目的是为了提升某些业务价值或去除某种异味，确保所采纳的实践能够产生预期的效果。做得越好，就越会开发出更好的软件。

如果您还没有创建好一个实施策略，我希望着这些模式和组合对您当前的敏捷实施有所帮助。在附录“充分利用敏捷模式”和“快速有效地阅读模式”中可以找到如何使用这些模式。

我知道还有很多敏捷实践没能在本书中详细讨论。附录“其他引用到的敏捷实践模式”给出了对这些实践的简短描述。还有一些敏捷实践及组合正在计划中。期望在 2007 年下半年能完成后续部分，详述和解释这些实践及组合。

记住稍微质疑一下这些模式。只有在您是为自己量身打造解决方案的时候，使用模式才能帮得到您。所有的这些模式，都来自于许多项目的实践。在某些领域里，它们的确经过了反复的验证。但，银弹是不存在的。在某些情况下，这些模式很可能就不合适。把这些模式当作指导，当实际情况与理论背道而驰时——实事求是地解决问题。

最后，如果您希望能了解更多正在完成的敏捷实践模式，您可以随时到如下网站查看最新的进展：http://www.elssamadisy.com。

附　录

17

敏捷实践模式与商业价值的映射

表中“敏捷实践组合”和“敏捷实践模式”列中的实践，是按照它们对于提升相应商业价值的有效性进行排列的。举例来说，如果您想要“缩短产品上市时间”，可以优先考虑采纳简单设计，然后才是功能测试。

表：提升商业价值的敏捷实践及组合

商业价值	敏捷实践组合	敏捷实践模式
缩短产品上市时间	测试驱动开发，演进式设计，测试驱动需求	简单设计，重构，测试先行开发，测试后行开发，持续集成，功能测试
提高产品市场价值	测试驱动需求	功能测试
提高产品质量	测试驱动开发，测试驱动需求，演进式设计	测试先行开发，测试后行开发，重构，简单设计，持续集成
提高灵活性	演进式设计，测试驱动开发，测试驱动需求	自动化测试，重构，集体代码所有权，功能测试
提高可见度	测试驱动需求	功能测试，持续集成
降低成本	演进式设计，测试驱动开发，测试驱动需求	简单设计，重构，集体代码所有权，测试先行开发，测试后行开发，功能测试
延长产品寿命	测试驱动开发，演进式设计，测试驱动需求	重构，自动化测试，功能测试，简单设计

18

敏捷实践模式与异味的映射

表中“敏捷实践组合”和“敏捷实践模式”列中的实践，是按照它们对于去除相应异味的有效性进行排列的。假设您想要解决“交付质量无法令客户接受”的异味，您会优先考虑采纳测试先行开发，然后才是持续集成。

表：去除异味的敏捷实践及组合

异味	敏捷实践组合	敏捷实践模式
交付质量无法令客户接受	测试驱动开发，测试驱动需求，演进式设计	测试先行开发，测试后行开发，重构，简单设计，持续集成
交付新功能需要太长时间	测试驱动开发，演进式设计，测试驱动需求	简单设计，重构，测试先行开发，测试后行开发，持续集成，功能测试
有些功能并不被用户使用	测试驱动需求	功能测试
软件对用户用处不大	测试驱动需求	功能测试
软件开发过于昂贵	演进式设计，测试驱动开发，测试驱动需求	简单设计，重构，集体代码所有权，测试先行开发，测试后行开发，功能测试
“我们”与“他们”的对立	测试驱动需求	功能测试
客户不管三七二十一，什么都要	测试驱动需求	功能测试
客户？什么客户？！	测试驱动需求	无
令管理层惊讶	测试驱动需求	功能测试
瓶颈资源		集体代码所有权

项目反复拖延	测试驱动开发，测试驱动需求	自动化测试，功能测试，持续集成
缺陷跟踪系统中有成百上千的缺陷	测试驱动开发，测试驱动需求	自动化测试，功能测试，持续集成
需要“稳定化”阶段		持续集成
没有经常集成		持续集成

19

敏捷实施策略案例研究[18]

简介

人们太容易沉迷于尝试新的敏捷实践如结对编程，迭代开发和测试驱动需求，而忽略了先去思考为什么要实施这些实践，背后的动机是什么。或许是因为有一个笼统的想法——“任何新东西肯定会比我们惯用的旧方法（通常比较痛苦）好”，因而 仅仅凭借“我们正在尝试新的东西”这个理由，往往就去投入时间和精力去实施新实践。但又存在另外一个问题。在敏捷这把大伞下有那么多敏捷实践，可能您也发现自己正在努力想办法一次就尽可能采纳所有实践，因为或许恰恰被您忽略的那个实践才会带来最好的效果。带着这种疯狂想法，人们常采用的一个办法是捡起某一种方法论，或一组实践，在软件开发组织内部推行，结果使它变成了一个“XP 商店”或“Scrum 商店”或“UP 商店”。团队中的每个成员，加上市场团队，以及一些更有经验的高级管理人员，购买一份您提供的敏捷方法实施的复制品，再对诸如什么时间在哪里站立会议、使用哪一个持续集成工具达成一致意见后，您的组织就变成了一个完完全全的敏捷开发商店了！

虽然这是一个常见且有用的方法，但没有特定目标，很容易造成仅仅为了改变而改变。再说，实际上是有一种更有针对性的敏捷实施方法，它不会有意偏重于某一种特定的方法论，而是帮您找到最有利于实现组织目标的敏捷实践。这种方法可归纳为下面三点：

- 真是太容易忘记谁是真正的客户了！
- 为了改变而改变，很容易偏离敏捷而不是走向真正的敏捷。
- 不必为了特意看到一些效果去采纳每一个流行的敏捷实践，而是采纳一个有针对性、对症下药的方法来助您在真正需要的方面做得更快更好。

这篇文章里我们将探讨正在进行的BC2.0 团队[19]的工作情况。这个开发团队目前正在重写一个曾很成功的网站，每天有着百万点击率。如何识别出最能帮助我们的敏捷实践？这里将与大家分享我们的经验。我们所采取的方法是这样的：先找出任何有可能获得的业务价值，形

[18] 注：这篇文章已在 InfoQ 发表过，这儿的使用得到了 InfoQ 及其作者的许可。——作者注

[19] 指当时发表这篇文章的时间，2007 年 1 月。——译者注

成一个全面的业务价值列表，排定优先级；然后团队在其中找出最能准确表达他们想通过BC2.0 所达到的几个最重要的业务价值；接着，我们开始讨论哪些敏捷实践最有可能帮助他们获取最重要的三个业务价值，我们发现一些关键实践不是受另一些实践的影响，就是需要依赖于其它的实践；最后我们讨论了团队目前正在实施的敏捷实践，并基于此创建了实施计划，逐步采纳其他更有影响力的各个实践。

制定敏捷实践实施策略

确定业务价值

无论团队当前情况如何，第一步是要关注于要带给客户的业务价值。实际中，这需要我们稍退一步，重新去识别客户究竟是为谁服务的？要知道众多的公众服务网站，其收入主要来源于广告，而很少来自于网站的最终用户。带着这些想法，我们进行了一个练习，让开发团队根据自己的理解对业务价值进行排序。我们必须再次让 BC2.0 团队的客户——包括广告团队，出版团队和管理团队——回答这一问题。此时我们得出了最初版本的业务价值列表，按优先顺序排列：

1. 针对市场的价值/产品实用性
2. 针对市场的产品质量
3. （对客户的）可见度

还有一些其他可能对公司很重要的业务价值，比如：

- 降低成本
- 灵活性（变化太多）
- 上市时间
- 产品生命周期

在这首要的三项业务价值中，基本上产品实用性最为重要。这意味着他们的重点应该是交付一个对最终用户真正有用的网站。交付一个高质量的网站，让客户随时知道正在发生的变化，这两者也很有价值。对这个特定的公司来说，这份列表甚至还告诉了他们什么是低优先级的业务价值。在很多公司，减少成本、快速交付以及开发一个“长命”的产品是其关键目标，可想而知这会地影响到他们对敏捷实践的选择。然而，如果决定将精力集中于交付高质量、实用的站点，意味着他们只能偏重于某几个方面，具体地说就是处理客户的参与和反馈。

业务价值驱动的开发过程和技术决策

由业务价值驱动开发过程和技术决策——这个建议看起来很明白，实际上我们软件开发社区却从未做好过。这意味着，如果一个实践或技术不能对组织或客户看重的业务价值有所促进，我们就不应该采纳。

这里有一系列 有待考虑的软件开发实践（对所有的业务价值）：

1. 测试先行开发
2. 测试后行开发
3. 演进式设计 （组合）
4. 预先设计
5. 预先构架
6. 预先需求分析
7. 重构
8. 持续集成
9. 简单设计
10. 集体代码所有权
11. 测试驱动开发（组合）
12. 功能测试
13. 测试驱动需求（组合）
14. 迭代
15. 站立会议
16. 回顾
17. 结对编程
18. 启动会议
19. 用户故事
20. 用例
21. 信息辐射器

22. 客户作为团队成员

23. 唤醒式文档

24. 设定了优先级的待办工作列表

25. 演示

在上述实践之中，以下是团队正在应用的:

1. 预先构架
2. 预先需求分析
3. 持续集成
4. 功能测试（刚开始的阶段）
5. 迭代
6. 回顾
7. 启动
8. 用户故事
9. 集体代码所有权

需要引入哪些实践，需要重点关注哪些实践？对于我们最关注的三项业务价值，对应地有三幅敏捷实践依赖图，我们借助它们来做出决策。每幅图显示了能其相应业务价值施加影响的各个实践，及其之间的相互依赖关系。

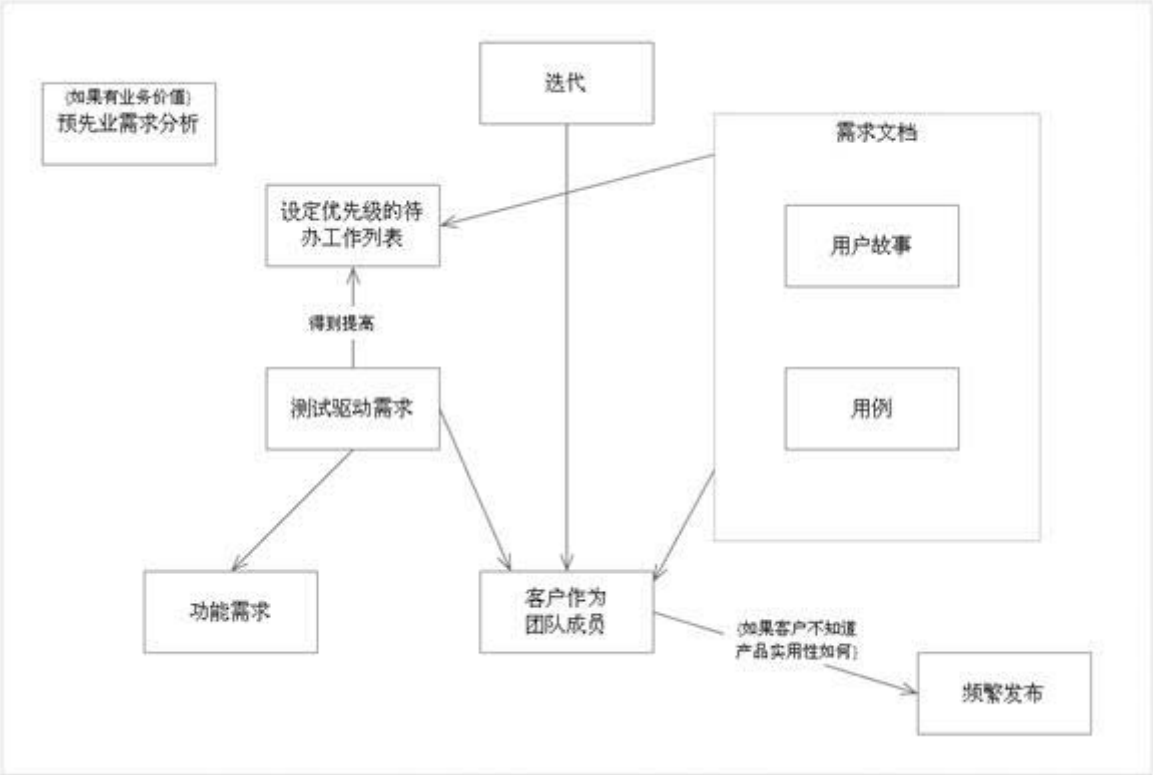


图 4：对应于产品实用性的实践

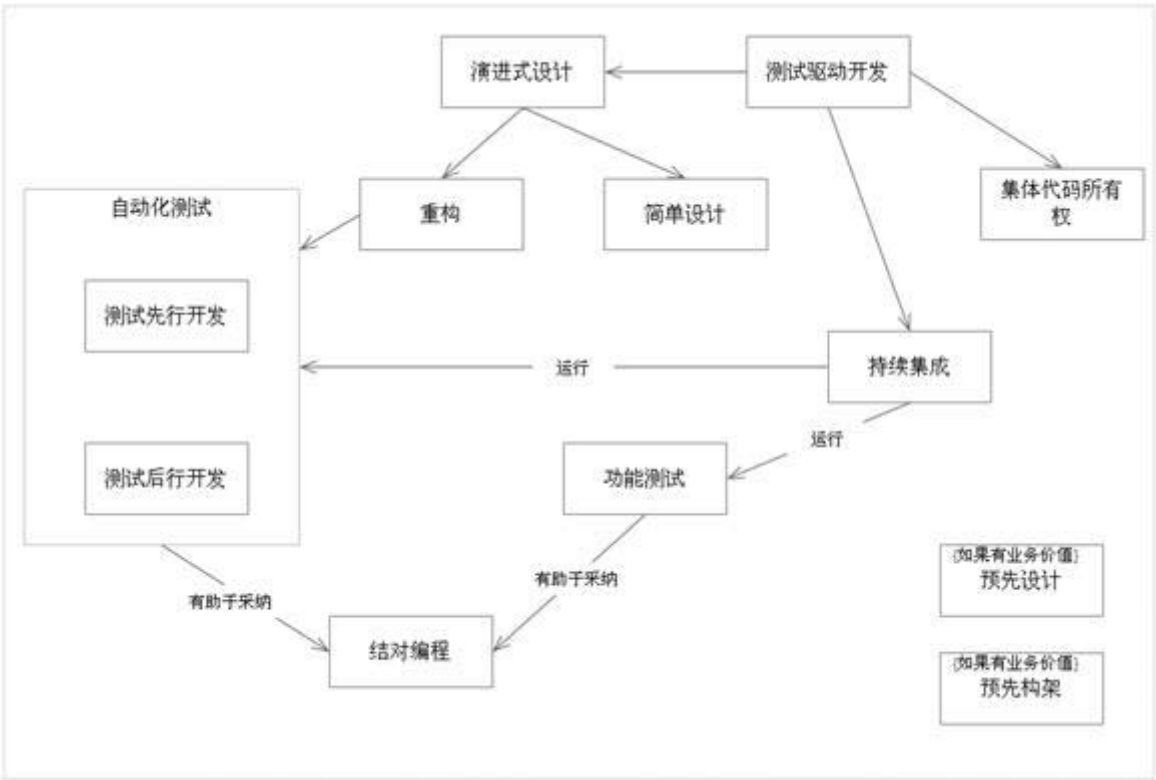


图 5：对应于产品质量的实践

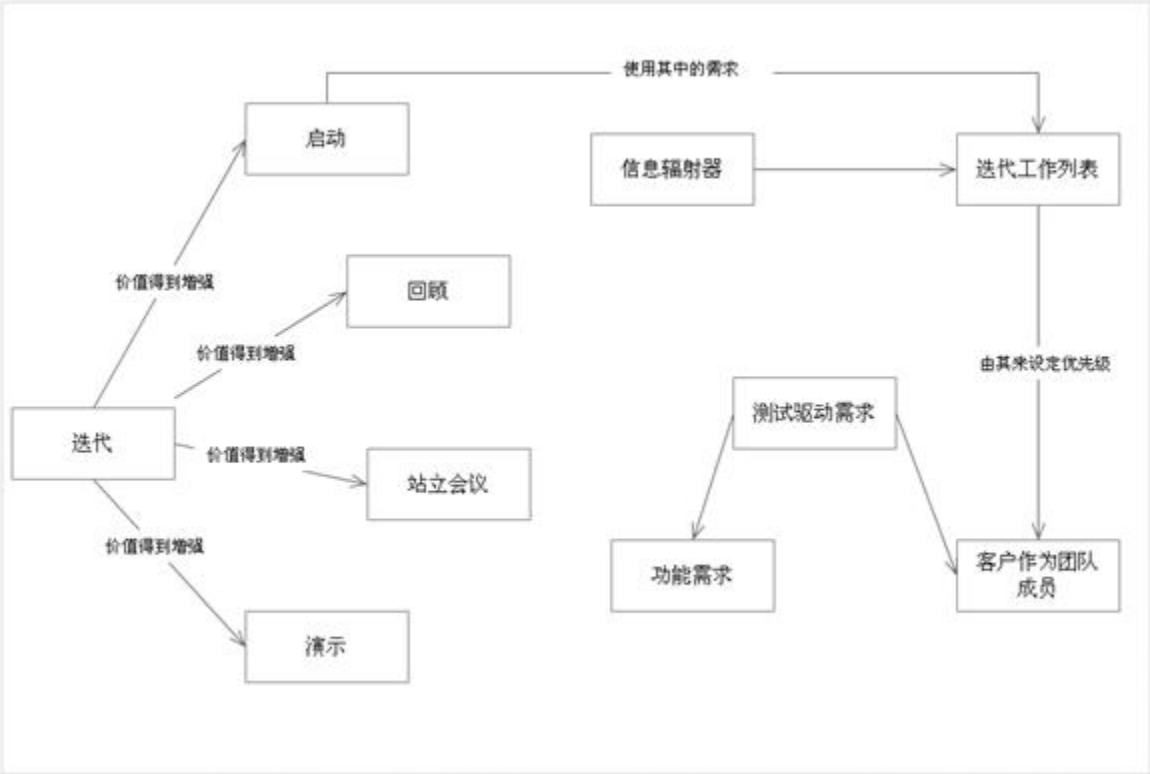


图 6：对应于提高可见度的实践

以上图中，你会发现有一些关键实践，它们要么影响着许多其它实践，要么是其它大多数实践的基础。根据业务价值优先级， 可以从其中任一关键实践入手，然后逐步地采纳其它各个实践。

为直接提升高优先级商业价值，在当前已采纳的所有实践（除了预先架构）前面已描述过了。因为现实点讲，很难放弃那些实践，所以会保留它们，但考虑减少预先架构和预先需求分析。此时我们有大串实践想要采纳，还有一些想要逐步减少。一次就实施一大堆实践从来就不是什么好主意，增量式实施策略会更好。

那么，哪些实践应该采纳呢？我们从最重要的业务价值——产品实用性——入手。在图 4 中，我们选择了输入依赖最多的实践，因为它们最能促使其他实践的实施。由此我们得到了：

- 客户作为团队成员
- 频繁发布
- 自动化功能测试

接下来看对应于产品质量的实践。因为有很多其他实践的实施依赖于自动化测试和结对编程，所以我们选择了这两者：

- 结对编程
- 自动化测试

对于第三位的业务价值，我们选择一个独立的实践去实施：

- 信息辐射器

当团队成功地实施了这些实践后，他们会引入更多的实践，来增加交付给客户的业务价值。一个可行的实施策略会以迭代地方式逐步吸收新实践，换句话说：小步实施。BC2.0 团队将从这些实践开始，边实施边学习。他们需要亲自体验这些实践，积累自己的经验。在一两个周期后，他们应该重新评估业务价值列表，看看有什么变化或者有什么已经明显得到解决。出了每个迭代末的回顾，他们也应该周期性地回顾已经取得的进步和得到的反馈，以其作为持续改进的方向盘。

结论

这个开发团队全力以赴地交付可用性和质量，在如今这种不懈追求低成本以及强调上市时间的年头，这看起来有点不寻常。但是，从实施角度讲，这是一个相当典型的案例，采用一种混合式的方法，整体上采纳 Scrum，而同时吸纳个人极限编程实践如持续集成和用户故事。这是一种“逐个击破”的方式，而不是一下子实施所有极限编程实践。但这种方式也没什么错，我们只是想强调某些实践或其组装在一起，能够有针对地提升某些业务价值。因此追求“有效利用每一分钱”的团队应该选择与其背后驱动力相契合的实践。

深入阅读

这是一个实际案例，针对特定开发团队和项目，量体裁衣地制定敏捷实施策略。迷您书《敏捷模式：技术实践》从一个更为详细的视角描述了如何制定敏捷实施策略，以及如何增量地实施多个敏捷实践。

- Amr Elssamadisy and John Mufarrige January 14, 2007
- amr.elssamadisy@valtech.com and john.mufarrige@valtech.com

20

其他引用到的敏捷实践模式

为了让这本书短小精悍，早点出版，书中只完整描述了一小部分敏捷实践。还有很多敏捷实践及有用的组合，需要以模式的形式描述出来——期待在 2007 年晚些时候完成。就是说，对于那些在本书中提到，但暂没有以完整的模式方式定义出来的实践，这里会一一列出。如果您想了解更多的正在完成的敏捷实践模式，您可以随时在 http://www.elssamadisy.com 找到最新的更新。

敏捷实践模式	描述
客户作为团队成员	客户（或客户代理，如业务分析师）作为开发团队的一部分。需要有一个客户与开发人员、测试人员经常交流，共同努力去达成项目目标。理想情况下，这个客户会与开发人员坐在一起工作。
信息辐射器	一份文档，一张海报，一个网页或是其他设备，位于一个显眼的位置，所有团队成员都能经常看见它，用来传递重要信息，帮助团队成员做出合理的决策。它们也会用来创建一个敏捷的工作环境。
迭代	在一个给定的时间段，团队设定目标并作出承诺，然后排除干扰，完成既定目标。这个目标应该总是系统的一部分，可演示，可工作。
结对编程	两个人在同一台机器上，一起工作，开发软件。通常情况下这会产生更高质量的代码，在实践中具有更强的自律性，增强信息和知识的传播。
回顾	在每个循环周期（迭代，发布等）末，团队去回顾总结那些地方做得好，那些地方需要改进。从回顾中得到的反馈会促使软件开发过程和实践的改进，从而在将来交付更多的业务价值。

站立会议	开发团队每天花 10 到 15 分钟的时间 ,在一起回顾前一天的进展情况，提出遇到的障碍。

21

充分利用敏捷实践模式

这本书的第一部分阐述了怎么样着手设立目标，选择合适的实践，最终达到目的。一旦您已经有了目标，知道那些是合适的实践，那您的团队应该能够清楚地回答下列问题。如果团队在这一刻无法给出清楚的答案，那在后面也会是糊里糊涂，步履蹒跚。假定下面这些问题是关于采纳实践 A：

1. 它应该适用哪一种实施策略？应该第一个实施它吗？或是先使用其他实践热热身，过几个月之后再引入它？
2. 哪一个开发实践与实践 A 相关？为了让实践 A 能起到实效，有哪些实践是必须先进行的？实践 A 是其他一些实践的先决条件吗？一组相关的开发实践紧密相关，一起采用时的价值会远大于一个一个实施，实践 A 是否隶属于某个实践组合？
3. 我们应该立刻全面实施实践 A，还是分阶段实施？是否有一些特殊的技巧来帮我们实施它？
4. 在采纳实践 A 的时候，有什么陷阱需要警惕的？会有一些事很容易做错吗？都是些什么事？会产生出什么异味？做错了会有什么样的症状？
5. 是不是在有些情况下我们不应该采纳实践 A？
6. 我们可以在不改变其实质的情况下，调整实践 A 的形式吗？到底什么是它的本质？
7. 要使实践 A 具有实效，对它所能带来的价值做出的假设，是不是团队成员都真正理解了？
8. 最后依照敏捷原则——实践 A 到底能为开发团队带来什么业务价值？
9. 所有这些都很重要。当团队决定要采纳一个开发实践的时候，应该要问这些问题。有些问题的答案不是那么显而易见的。但大多数问题，通过参考这本书，您都能给出简明扼要的答案。

22

有效地阅读这些模式

您会发现书中所描述的模式内容上有一个自然的叠合。这不是巧合。如果把这些叠合的部分移走，就没办法单独摘出某一个模式来阅读。

在每一个模式中，各部分之间也有些自然的重复。动因部分列出了模式所能解决的问题。因此部分讲述解决这些问题的具体方法。但是部分常常讨论一些 例外，又把我们带回到动因中所描述的内容。最后，采纳方法部分实际上重复了因此部分的内容，虽然它们是从两个不同的角度来描述的。

阅读本书中的模式，可以有很多种方式。比如您可以在一个模式的各个部分中跳来跳去。虽然有时这些重复让人烦，但比起需要翻前翻后对照阅读要好得多，希望您也有同感。

本书中的模式阅读起来可以很灵活。下面给出了一些不同情况下的阅读方式：

- 我已经在实践这个模式了，没什么大问题。我只是想了解下别人是怎么使用这一模式的。
  - 按名字查找这个模式。
  - 阅读适用情况，了解下您是否和别人处于相同的境况。
  - 阅读因此和变化部分，看看您所使用的方式是不是和其中的一些相类似。
- 我在实践一个模式，但看起来不是很有效。是不是我使用该模式的方法不正确？或者这个模式在我的环境不适用？
  - 按名字查找这个模式。
  - 阅读适用情况部分。如果您的环境与其不符，或许您该考虑调整一下所用的实践或者干脆不用它。
  - 阅读动因部分。您在尝试解决同样类型的问题吗？如果不是，需要斟酌下。或许这个实践能帮些忙但您还需要另外一个实践来解决您真正的问题。
  - 看看但是部分。您会了解到别人是在这儿如何犯错，如何获取建议纠正错误，并最终从中充分获益的。

- 在我的团队中有些问题，我想要采纳敏捷实践来解决这些问题。
  - 回到第四章异味部分，找着看您的问题反映出了哪些异味。
  - 阅读能够解决这些异味的实践。
  - 对于每一个实践
    - 阅读适用情况，确保它适用于您的情况。
    - 阅读其余部分。
    - 如果您决定采纳该实践，按照采纳方法部分的建议去做。
    - 做周期性检查，看看是否有但是部分所描述的问题发生。
- 在异味一章中，找不到我想要解决的问题。那是否意味着所有的实践都帮不上忙？
  - 不是。阅读每一个模式中的动因部分，看看其中是否有相似的问题。您很有可能会找到一个。
- 我们在采纳一个特定的实践。我们成功了吗？我们的采纳方法是否达到极致，完全发挥了它的作用？
  - 按名字查到这个模式。
  - 对照动因进行检查。是否还有哪个问题在您的团队中存在？
  - 对照但是进行检查。是否出现了其中所描述的某个异味？如果是，解决它们。

如果以上任一情况都已经不适用，那您可能已经不用读这本书了。您或许已经有足够的经验及直觉，可以灵活地根据自己的需要，选择适当的模式。可喜可贺！

参考书目

- Astels, David. 《测试驱动开发：实用指南》，中国电力出版社，崔凯译，2004
- Beck, Kent. 《解析极限编程--拥抱变化》，人民邮电出版社，唐东铭译，2002
- Beck, Kent and Andres Cynthia. 《解析极限编程--拥抱变化（第二版）》，电子工业出版社，雷剑文、 陈振冲、李明树译，2006
- Beck, Kent. 《测试驱动开发（中文版）》，中国电力出版社，孙平平等译，2004
- Feathers, Michael.《修改代码的艺术》，人民邮电出版社，刘未鹏译，2007
- Fowler, Martin.《重构：改善既有代码的设计》，中国电力出版社，侯捷、熊节译，2003
- Fowler, Martin. "Continuous Integration," www.martinfowler.com/articles/continuousIntegration.html .
- Gandhi, P., Haugen, N., Hill, M., and Watt, R., 2005, "Creating a Living Specification Document with FIT," Agile 2005 Conference.
  Jeffries, Ron. Extreme Programming Adventures in C#. Redmond, Washington: Microsoft Press, 2004.
- Jeffries, Ron., "Running Tested Features,"
- www.xprogramming.com/xpmag/jatRtsMetric.htm
- Kerievsky, Joshua.《重构与模式》，人民邮电出版社，杨光、刘基诚译，2006。
- Kerievsky, Joshua. Don’ t Just Break Software, Make Software.
- http://www.industriallogic.com/papers/storytest.pdf .
- Marick, Brian., 2002, "Bypassing the GUI," Software Testing and Quality Engineering, September/October, 41–47.
- Martin, Robert C.《敏捷软件开发 :原则、模式与实践》,清华大学出版社 ,邓辉译，2003。
- Massol, Vincent. 《JUnit In Action中文版》，电子工业出版社，鲍志云译，2004
- Mugridge, R., and Cunningham W., 《集成测试框架--用 Fit 进行敏捷软件测试》，电子工业出版社， 吴兰陟译，2007

- Rainsberger, J. B.《JUnit Recipes 中文版——程序员实用测试技巧》,电子工业出版社，李笑 、陈浩、王耀伟译，2006
- Shore, J., "A Vision for Fit," www.jamesshore.com/Blog/A-Vision-For-Fit.html

关于作者

Amr Elssamadisy 是一个软件开发实践者——他在多个开发团队中担任过各种不同的角色，包括教练、指导人员、开发人员、架构师、技术负责人、Scrum Master，项目经理等等。他热衷于开发出好的软件。这需要创造性，很有挑战，有时会让人垂头丧气，但最终又会觉得一切付出都非常值得。这也是为什么他痴迷于敏捷开发实践的原因——因为如果能正确地应用这些敏捷实践，将会为软件开发团队带来意想不到的奇迹。

Amr 自从 1999 年下半年在 ThoughtWorks 接触到极限编程后，一直都在从事软件开发工作（作为一个咨询顾问，为不同的客户工作）。到本书出版之日止，Amr 已经花了 7 年的时间致力于敏捷实践，帮助各种团队采纳敏捷实践，并根据自身特点调整改进，最终开发出更好的软件。

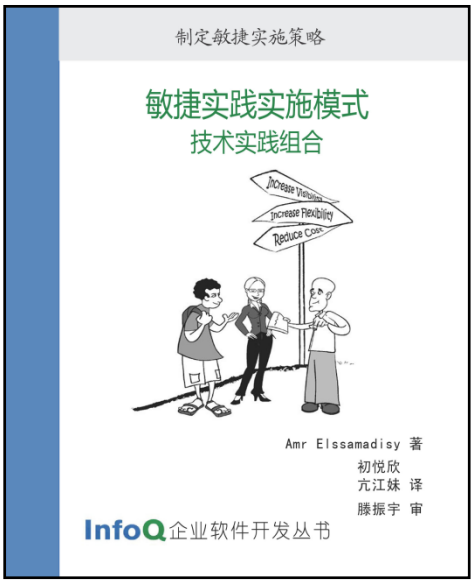
目前 Amr 是 Valtech 的一名首席咨询顾问，致力于利用最新的技术，采纳和适应敏捷实践，帮助 Valtech 的客户开发出更好的软件。

关于 Amr 本人，他非常享受简单的生活：家庭，美食，音乐和视频游戏。您可以通过 http://www.elssamadisy.com 联系到他。

关于译者

初悦欣，现就职于ThoughtWorks的Mingle团队，热衷于编程，擅长Ruby on Rails，J2EE，Javascript 等开发语言，敏捷咨询师，Web 应用开发者和设计师，Blogger。详细信息可访问 http://www.phoenixchu.com 。

亢江妹，高级业务分析师，目前在 ThoughtWorks 工作。她对技术充满热情，梦想用技术改变世界。目前专注于如何在软件开发组织中培养、发挥个人和团队的软技能，建立快乐团队，使团队能真正走向敏捷。联系方式：Jessica.kjm@gmail.com



敏捷实践实施模式

----技术实践组合

责任编辑：郑柯

美术编辑：胡伟红

审校编辑：滕振宇

本迷你书主页为

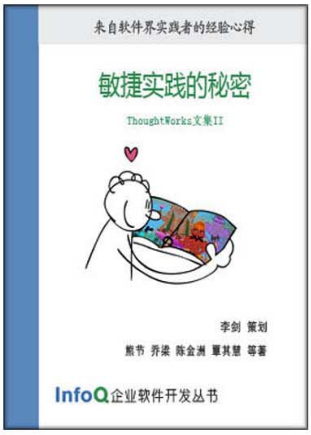
http://www.infoq.com/cn/minibooks/agile-patterns-cn

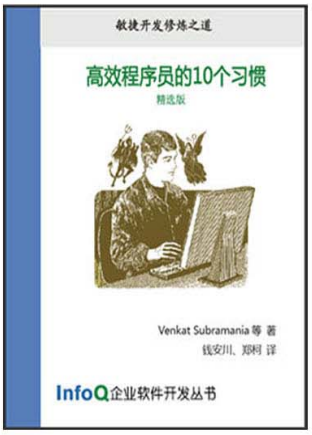
本书属于 InfoQ 企业软件开发丛书。

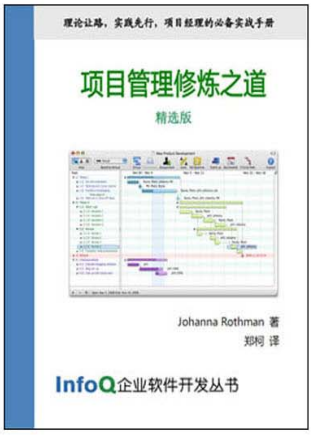
如果您打算订购 InfoQ 的图书，请联系 books@c4media.com

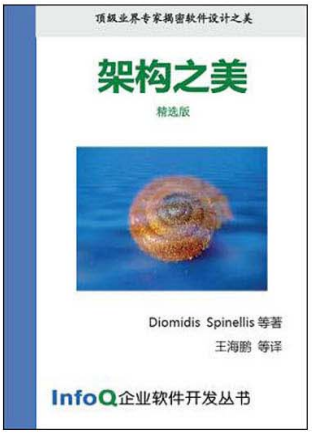
InfoQ企业软件开发丛书

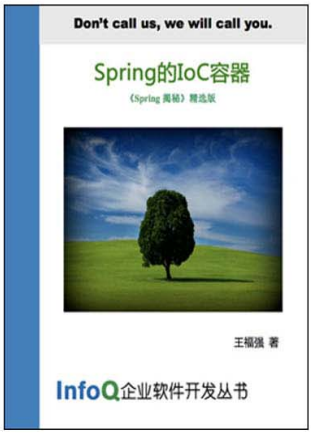
欢迎免费下载











商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com