

敏捷开发修炼之道

高效程序员的10个习惯

精选版



Venkat Subramania 等 著

钱安川、郑柯 译

InfoQ企业软件开发丛书

免费在线版本

(非印刷免费在线版)

[登录China-Pub网站购买此书完整版](#)



了解本书更多信息请登录[本书的官方网站](#)

InfoQ 中文站出品

InfoQ中文站
www.infoq.com/cn

本迷你书精选了书中的 10 个习惯，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://infoq.com/cn/minibooks/practice-agile-developer>

译者序

“武功者，包括内功、外功、武术技击术之总和。有形的动作，如支撑格拒，姿势回环，变化万千，外部可见，授受较易，晨操夕练，不难熟练。而无形的内功指内部之灵惠素质，即识、胆、气、劲、神是也，此乃与学练者整个内在世界的学识水平密切相关，是先天之慧根悟性与后天智能的总成，必需寻得秘籍方可炼成。”

——摘自《武林秘籍大全》

公元21世纪，软件业江湖动荡，人才辈出，各大门派林立，白道黑帮，都欲靠各自门派的武功称霸武林。

在那些外家功门派（传统的瀑布开发方法、CMM、ISO和RUP等）和非正统教（中国式太极敏捷UDD等）当道之际，一股新势力正在崛起——以敏捷方法为总称的一批内家功门派。

下面的歌诀是对内家武功招数的概述：

迭代开发，价值优先
分解任务，真实进度

站立会议，交流畅通
用户参与，调整方向

结对编程，代码质量

测试驱动，安全可靠

持续集成，尽早反馈

自动部署，一键安装

定期回顾，持续改进

不断学习，提高能力

上面的每种招式，都可寻得一本手册，介绍其动作要领和攻防章法。几乎每个内家功门派都有自己的拳法和套路。


但，正所谓“练拳不练功，到老一场空”。学习招数和套路不难，难的是如何练就一身真功夫。内家功，以练内为主，内外结合，以动作引领内气，以内气催领动作，通过后天的修炼来弥补先天的不足。

本书是一本内功手册。它注重于培养软件开发者的态度、原则、操守、价值观，即识、胆、气、劲、神是也。

敏捷的实践者Venkat Subramaniam和Andy Hunt携手著下此书。望有志之士有缘得到此书，依法修习，得其精要；由心知到身知，入筋、入骨、入髓，修炼得道。而后，匡扶正义，交付高质量的软件，为人类造福。

安 川

2008年4月于北京



目 录

译者序.....	I
习惯一：对事不对人.....	1
习惯二：跟踪变化.....	6
习惯三：让设计指导而不是操纵开发.....	9
习惯四：提早实现自动化部署.....	13
习惯五：度量真实的进度.....	16
习惯六：用代码沟通.....	19
习惯七：编写内聚的代码.....	24
习惯八：根据契约进行替换.....	28
习惯九：报告所有的异常.....	32
习惯十：做代码复查.....	34

架构师

www.infoq.com/cn/architect

每月8号出版



1

对事不对人



你在这个设计上投入了很多精力，为它付出很多心血。你坚信它比其他任何人的设计都棒。别听他们的，他们只会把问题变得更糟糕。”

你很可能见过，对方案设计的讨论失控变成了情绪化的指责——做决定是基于谁提出了这个观点，而不是权衡观点本身的利弊。我们曾经参与过那样的会议，最后闹得大家都很不愉快。

但是，这也很正常。当Lee先生在做一个新方案介绍的时候，下面有人会说：“那样很蠢！”（这也就暗示着Lee先生也很蠢。）如果把这句话推敲一下，也许会好一点：“那样很蠢，你忘记考虑它要线程安全。”事实上最适合并且最有效的表达方式应该是：“谢谢，Lee先生。但是我想知道，如果两个用户同时登录会发生什么情况？”

看出其中的不同了吧！下面我们来看看对一个明显的错误有哪些常见的反应。

- 否定个人能力。
- 指出明显的缺点，并否定其观点。
- 询问你的队友，并提出你的顾虑。

第一种方法是不可能成功的。即使Lee是一个十足的笨蛋，很小的问题也搞不定，但你那样指出问题根本不会对他的水平有任何提高，反而会导致他以后也不会提出自己的任何想法了。第二种方法至少观点明确，但也不能给Lee太多的帮助，甚至可能会让你自己惹火上身。也许Lee能巧妙地回复你对非线性安全的指责：“哦，不过它不需要多线程。因为它只在Frozb什么模块的环境中使用，它已经运行在自己的线程中了。”哎哟！忘记了Frozb这一茬了。现在该是你觉得自己蠢了，Lee也会因为你骂他笨蛋而生气。

现在看看第三种方法。没有谴责，没有评判，只是简单地表达自己的观点。让Lee自己意识到这个问题，而不是扫他的面子^①。由此可以开始一次交谈，而不是争辩。

① 通常，这是一个很好的技巧：引导性地提出一个疑问，让他们自己意识到问题。



Venkat如是说……

要专业而不是自我

多年以前，在我担任系统管理员的第一天，一位资深的管理员和我一起安装一些软件，我突然按错了一个按钮，把服务器给关掉了。没过几分钟，几位不爽的用户就在敲门了。

这时，我的导师赢得了我的信任和尊重，他并没有指责我，而是对他们说：“对不起，我们正在查找是什么地方出错了。系统会在几分钟之内启动起来。”这让我学到了难忘的重要一课。

在一个需要紧密合作的开发团队中，如果能稍加注意礼貌对待他人，将会有益于整个团队关注真正有价值的问题，而不是勾心斗角，误入歧途。我们每个人都有一些极好的创新想法，同样也会萌生一些很愚蠢的想法。

如果你准备提出一个想法，却担心有可能被嘲笑，或者你要提出一个建议，却担心自己丢面子，那么你就不会主动提出自己的建议了。然而，好的软件开发作品和好的软件设计，都需要大量的创造力和洞察力。分享并融合各种不同的想法和观点，远远胜于单个想法为项目带来的价值。

负面的评论和态度扼杀了创新。现在，我们并不提倡在设计方案的会议上手拉手唱《学习雷锋好榜样》，这样也会降低会议的效率。但是，你必须把重点放在解决问题上，而不是去极力证明谁的主意更好。在团队中，一个人只是智商高是没有用的，如果他还很顽固并且拒绝合作，那就更糟糕。在这样的团队中，生产率和创新都会濒临灭亡的边缘。

消极扼杀创新

Negativity kills

我们每个人都会有好的想法，也会有不对的想法，团队中的每个人都需要自由地表达观点。即使你的建议不被全盘接受，也能对最终解决问题有所帮助。不要害怕受到批评。记住，任何一个专家都是从这里开始的。用Les Brown^①的一句话说就是：“你不需要很出色才能起步，但是你必须起步才能变得很出色。”

① 莱斯·布朗，全球领军励志演讲家和作家。——编者注

团体决策的骆驼

集体决策确实非常有效，但也有一些最好的创新源于很有见地的个人的独立思考。如果你是一个有远见的人，就一定要特别尊重别人的意见。你是一个掌舵者，一定要把握方向，深思熟虑，吸取各方的意见。

另一个极端是缺乏生气的委员会，每个设计方案都需要全票通过。这样的委员会总是小题大作，如果让他们造一匹木马，很可能最后造出的是骆驼。

我们并不是建议你限制会议决策，只是你不应该成为一意孤行的首席架构师的傀儡。这里建议你牢记亚里士多德的一句格言：“能容纳自己并不接受的想法，表明你的头脑足够有学识。”

下面是一些有效的特殊技术。

设定最终期限。如果你正在参加设计方案讨论会，或者是寻找解决方案时遇到问题，请设定一个明确的最终期限，例如午饭时间或者一天的结束。这样的时间限制可以防止人们陷入无休止的理论争辩之中，保证团队工作的顺利进行。同时（我们觉得）应现实一些：没有最好的答案，只有更合适的方案。设定期限能够帮你在为难的时候果断做出决策，让工作可以继续前进。

逆向思维。团队中的每个成员都应该意识到权衡的必要性。一种客观对待问题的办法是：先是积极地看到它的正面，然后再努力地从反面去认识它^①。目的是要找出优点最多缺点最少的那个方案，而这个好办法可以尽可能地发现其优缺点。这也有助于少带个人感情。

设立仲裁人。在会议的开始，选择一个仲裁人作为本次会议的决策者。每个人都要有机会针对问题畅所欲言。仲裁人的责任就是确保每个人都有发言的机会，并维持会议的正常进行。仲裁人可以防止明星员工操纵会议，并及时打断假大空式发言。

如果你自己没有积极参与这次讨论活动，那么你最好退一步做会议的监督者。仲裁人应该专注于调停，而不是发表自己的观点（理想情况下不应在整个项目中有

^① 参见“Debating with Knives”，在<http://blogs.pragprog.com/cgi-bin/pragdave.cgi/Random/FishBow1.rdoc>。

既得利益)。当然, 这项任务不需要严格的技术技能, 需要的是和他人打交道的能力。

支持已经做出的决定。一旦方案被确定了(不管是什么样的方案), 每个团队成员都必须通力合作, 努力实现这个方案。每个人都要时刻记住, 我们的目标是让项目成功满足用户需求。客户并不关心这是谁的主意——他们关心的是, 这个软件是否可以工作, 并且是否符合他们的期望。结果最重要。

设计充满了妥协(生活本身也是如此), 成功属于意识到这一点的团队。工作中不感情用事是需要克制力的, 而你若能展现出成熟大度来, 大家一定不会视而不见。这需要有人带头, 身体力行, 去感染另一部分人。



对事不对人。让我们骄傲的应该是解决了问题, 而不是比较出谁的主意更好。

切身感受

一个团队能够很公正地讨论一些方案的优点和缺点, 你不会因为拒绝了有太多缺陷的方案而伤害别人, 也不会因为采纳了某个不甚完美(但是更好的)解决方案而被人忌恨。

平衡的艺术

- 尽力贡献自己的好想法, 如果你的想法没有被采纳也无需生气。不要因为只是想体现自己的想法而对拟定的好思路画蛇添足。
- 脱离实际的反方观点会使争论变味。若对一个想法有成见, 你很容易提出一堆不太可能发生或不太实际的情形去批驳它。这时, 请先扪心自问: 类似问题以前发生过吗? 是否经常发生?
- 也就是说, 像这样说是不够的: 我们不能采用这个方案, 因为数据库厂商可能会倒闭。或者: 用户绝对不会接受那个方案。你必须评判那些场景发生的可能性有多大。想要支持或者反驳一个观点, 有时候你必须先做一个原型或者调

查出它有多少的同意者或者反对者。

- 在开始寻找最好的解决方案之前，大家对“最好”的含义要先达成共识。在开发者眼中的最好，不一定就是用户认为最好的，反之亦然。
- 只有更好，没有最好。尽管“最佳实践”这个术语到处在用，但实际上不存在“最佳”，只有在某个特定条件下更好的实践。
- 不带个人情绪并不是要盲目地接受所有的观点。用合适的词和理由去解释为什么你不赞同这个观点或方案，并提出明确的问题。

跟踪变化



“软件技术的变化如此之快，势不可挡，这是它的本性。继续用你熟悉的语言做你的老本行吧，你不可能跟上技术变化的脚步。”

赫拉克利特说过：“唯有变化是永恒的。”历史已经证明了这句真理，在当今快速发展的IT时代尤其如此。你从事的是一项充满激情且不停变化的工作。如果你毕业于计算机相关的专业，并觉得自己已经学完了所有知识，那你就大错特错了。

假设你是10多年前的1995年毕业的，那时，你掌握了哪些技术呢？可能你的C++还学得不错，你了解有一门新的语言叫Java，一种被称作是设计模式的思想开始引起大家的关注。一些人会谈论被称作因特网的东东。如果那个时候你就不再学习，而在2005年的时候重出江湖。再看看周围，就会发现变化巨大。就算是在一个相当狭小的技术领域，要学习那些新技术并达到熟练的程度，一年的时间也不够。

技术发展的步伐如此快速，简直让人们难以置信。就以Java为例，你掌握了Java语言及其一系列的最新特性。接着，你要掌握Swing、Servlet、JSP、Struts、Tapestry、JSF、JDBC、JDO、Hibernate、JMS、EJB、Lucene、Spring……还可以列举很多。如果你使用的是微软的技术，要掌握VB、Visual C++、MFC、COM、ATL、.NET、C#、VB.NET、ASP.NET、ADO.NET、WinForm、Enterprise Service、Biztalk……并且，不要忘记还有UML、Ruby、XML、DOM、SAX、JAXP、JDOM、XSL、Schema、SOAP、Web Service、SOA，同样还可以继续列举下去（我们将会用光所有的缩写字母）。

不幸的是，如果只是掌握了工作中需要的技术并不够。那样的工作也许几年之后就不再有——它会被外包或者会过时，那么你也将会出局^①。

假设你是Visual C++或者VB程序员，看到COM技术出现了。你花时间去学习它

① 参考My Job Went to India: 52 Ways to Save Your Job[Fow05]一书。新版改名为Passionate Programmer。

(虽然很痛苦), 并且随时了解分布式对象计算的一切。当XML出现的时候, 你花时间学习它。你深入研究ASP, 熟知如何用它来开发Web应用。你虽然不是这些技术的专家, 但也不是对它们一无所知。好奇心促使你去了解MVC是什么, 设计模式是什么。你会使用一点Java, 去试试那些让人兴奋的功能。

如果你跟上了这些新技术, 接下来学习.NET技术就不再是大问题。你不需要一口气爬上10楼, 而需要一直在攀登, 所以最后看起来就像只要再上一二层。如果你对所有这些技术都一无所知, 想要马上登上这10楼, 肯定会让你喘不过气来。而且, 这也会花很长时间, 期间还会有更新的技术出现。

如何才能跟上技术变化的步伐呢? 幸好, 现今有很多方法和工具可以帮助我们继续充电。下面是一些建议。

迭代和增量式的学习。每天计划用一段时间来学习新技术, 它不需要很长时间, 但需要经常进行。记下那些你想学习的东西——当你听到一些不熟悉的术语或者短语时, 简要地把它记录下来。然后在计划的时间中深入研究它。

了解最新行情。互联网上有大量关于学习新技术的资源。阅读社区讨论和邮件列表, 可以了解其他人遇到的问题, 以及他们发现的很酷的解决方案。选择一些公认的优秀技术博客, 经常去读一读, 以了解那些顶尖的博客作者们正在关注什么(最新的博客列表请参考pragmaticprogrammer.com)。

参加本地的用户组活动。Java、Ruby、Delphi、.NET、过程改进、面向对象设计、Linux、Mac, 以及其他的各种技术在很多地区都会有用户组。听讲座, 然后积极加入到问答环节中。

参加研讨会议。计算机大会在世界各地举行, 许多知名的顾问或作者主持研讨会或者课程。这些聚会是向专家学习的最直接的好机会。

如饥似渴地阅读。找一些关于软件开发和非技术主题的好书(我们很乐意为你推荐), 也可以是一些专业的期刊和商业杂志, 甚至是一些大众媒体新闻(有趣的是在那里常常能看到老技术被吹捧为最新潮流)。



跟踪技术变化。你不需要精通所有技术，但需要清楚知道行业的动向，从而规划你的项目和职业生涯。

切身感受

你能嗅到将要流行的新技术，知道它们已经发布或投入使用。如果必须要把工作切换到一种新的技术领域，你能做到。

平衡的艺术

- 许多新想法从未变得羽翼丰满，成为有用的技术。即使是大型、热门和资金充裕的项目也会有同样的下场。你要正确把握自己投入的精力。
- 你不可能精通每一项技术，没有必要去做这样的尝试。只要你在某些方面成为专家，就能使用同样的方法，很容易地成为新领域的专家。
- 你要明白为什么需要这项新技术——它试图解决什么样的问题？它可以被用在什么地方？
- 避免在一时冲动的情况下，只是因为想学习而将应用切换到新的技术、框架或开发语言。在做决策之前，你必须评估新技术的优势。开发一个小的原型系统，是对付技术狂热者的一剂良药。

让设计指导而不是操纵开发



“设计文档应该尽可能详细，这样，低级的代码工人只要敲入代码就可以了。在高层方面，详细描述对象的关联关系；在低层方面，详细描述对象之间的交互。其中一定要包括方法的实现信息和参数的注释。也不要忘记给出类里面的所有字段。编写代码的时候，无论你发现了什么，绝不能偏离了设计文档。”

“设计”是软件开发过程不可缺少的步骤。它帮助你理解系统的细节，理解部件和子系统之间的关系，并且指导你的实现。一些成熟的方法论很强调设计，例如，统一过程（Unified Process，UP）十分重视和产品相关的文档。项目管理者和企业主常常为开发细节困扰，他们希望在开始编码之前，先有完整的设计和文档。毕竟，那也是你如何管理桥梁或建筑项目的，难道不是吗？

另一方面，敏捷方法建议你早在开发初期就开始编码。是否那就意味着没有设计呢？^①不，绝对不是，好的设计仍然十分重要。画关键工作图（例如，用UML）是必不可少的，因为要使用类及其交互关系来描绘系统是如何组织的。在做设计的时候，你需要花时间去思考（讨论）各种不同选择的缺陷和益处，以及如何做权衡。

然后，下一步才考虑是否需要开始编码。如果你在前期没有考虑清楚这些问题，就草草地开始编码，很可能被很多意料之外的问题搞晕。甚至在建筑工程方面也有类似的情况。在锯一根木头的时候，通常的做法就是先锯一块比需要稍微长一点的木块，最后细致地修整，直到它正好符合需求。

但是，即使之前已经提交了设计文档，也还会有一些意料之外的情况出现。时刻谨记，此阶段提出的设计只是基于你目前对需求的理解而已。一旦开始了编码，一切都会改变。设计及其代码实现会不停地发展和变化。

一些项目领导和经理认为设计应该尽可能地详细，这样就可以简单地交付给“代

^① 查阅Martin Fowler的文章*Is Design Dead?* (<http://www.martinfowler.com/articles/designDead.html>)，它是对本主题深入讨论的一篇好文章。

码工人们”。他们认为代码工人不需要做任何决定，只要简单地把设计转化成代码就可以了。就作者本人而言，没有一个愿意在这样的团队中做纯粹的打字员。我们猜想你也不愿意。

如果设计师们把自己的想法绘制成精美的文档，然后把它们扔给程序员去编码，那会发生什么（查阅习惯39，在第152页）？程序员会在压力下，完全按照设计

或者图画的样子编码。如果系统和已有代码的现状表明接收到的设计不够理想，那该怎么办？太糟糕了！时间已经花费在设计上，没有工夫回头重新设计了。团队会死撑下去，用代码实现了明明知道是错误的设计。这听起来是不是很愚蠢？是够愚蠢的，但是有一些公司真的就是这样做的。

设计满足实现即可，不必过于详细
Design should be only as
detailed as needed to

严格的需求-设计-代码-测试开发流程源于理想化的瀑布式^①开发方法，它导致在前面进行了过度的设计。这样在项目的生命周期中，更新和维护这些详细的设计文档变成了主要工作，需要时间和资源方面的巨大投资，却只有很少的回报。我们本可以做得更好。

设计可以分为两层：战略和战术。前期的设计属于战略，通常只有在没有深入理解需求的时候需要这样的设计。更确切地说，它应该只描述总体战略，不应深入到具体的细节。

做到精确

如果你自己都不清楚所谈论的东西，就根本不可能精确地描述它。

——约翰·冯·诺依曼

前面刚说过，战略级别的设计不应该具体说明程序方法、参数、字段和对象交互精确顺序的细节。那应该留到战术设计阶段，它应该在项目开发的时候再具体展开。

① 瀑布式开发方法意味着要遵循一系列有序的开发步骤，前面是定义详细的需求，然后是详细的设计，接着是实现，再接着是集成，最后是测试（此时你需要向天祈祷）。那不是作者首先推荐的做法。更多详情可以查阅[Roy70]。

良好的战略设计应该扮演地图的角色，指引你向正确的方向前进。任何设计仅是一个起跑点：它就像你的代码一样，在项目的生命周期中，会不停地进一步发展和提炼。

战略设计与战术设计

Strategic versus tactical design

下面的故事会给我们一些启发。在1804年，Lewis与Clark^①进行了横穿美国的壮举，他们的“设计”就是穿越蛮荒。但是，他们不知道在穿越殖民地时会遇到什么样的问题。他们只知道自己的目标和制约条件，但是不知道旅途的细节。

软件项目中的设计也与此类似。在没有穿越殖民地的时候，你不可能知道会出现什么情况。所以，不要事先浪费时间规划如何徒步穿越河流，只有当你走到河岸边的时候，才能真正评估和规划如何穿越。只有到那时，你才开始真正的战术设计。

不要一开始就进行战术设计，它的重点是集中在单个的方法或数据类型上。这时，更适合讨论如何设计类的职责。因为这仍然是一个高层次、面向目标的设计。事实上，CRC（类-职责-协作）卡片的设计方法就是用来做这个事情的。每个类按照下面的术语描述。

- 类名。
- 职责：它应该做什么？
- 协作者：要完成工作它要与其他什么对象一起工作？

如何知道一个设计是好的设计，或者正合适？代码很自然地设计的好坏提供了最好的反馈。如果需求有了小的变化，它仍然容易去实现，那么它就是好的设计。而如果小的需求变化就带来一大批基础代码的破坏，那么设计就需要改进。



好设计是一张地图，它也会进化。设计指引你向正确的方向前进，它不是殖民地，它不应该标识具体的路线。你不要被设计（或者设计师）操纵。

^① 这世界真小，Andy还是William Clark的远亲呢。

切身感受

好的设计应该是正确的，而不是精确的。也就是说，它描述的一切必须是正确的，不应该涉及不确定或者可能会发生变化的细节。它是目标，不是具体的处方。

平衡的艺术

- “不要在前期做大量的设计”并不是说不要设计。只是说在没有经过真正的代码验证之前，不要陷入太多的设计任务。当对设计一无所知的时候，投入编码也是一件危险的事。如果深入编码只是为了学习或创造原型，只要你随后能把这些代码扔掉，那也是一个不错的办法。
- 即使初始的设计到后面不再管用，你仍需设计：设计行为是无价的。正如美国总统艾森豪威尔所说：“计划是没有价值的，但计划的过程是必不可少的^①。”在设计过程中学习是有价值的，但设计本身也许没有太大的用处。
- 白板、草图、便利贴都是非常好的设计工具。复杂的建模工具只会让你分散精力，而不是启发你的工作。

^① 1957年的演讲稿。

提早实现自动化部署



“没问题，可以手工安装产品，尤其是给质量保证人员安装。而且你不需要经常自己动手，他们都很擅长复制需要的所有文件。”

系统能在你的机器上运行，或者能在开发者和测试人员的机器上运行，当然很好。但是，它同时也需要能够部署在用户的机器上。如果系统能运行在开发服务器上，那很好，但是它同时也要运行在生产环境中。

这就意味着，你要能用一种可重复和可靠的方式，在目标机器上部署你的应用。不幸的是，大部分开发者只会在项目的尾期才开始考虑部署问题。结果经常出现部署失败，要么是少了依赖的组件，要么是少了一些图片，要么就是目录结构有误。

如果开发者改变了应用的目录结构，或者是在不同的应用之间创建和共享图片目录，很可能会导致安装过程失败。当这些变化在人们印象中还很深的时候，你可以快速地找到各种问题。但是几周或者几个月之后查找它们，特别是在给客户演示的时候，可就不是一件闹着玩的事情了。

如果现在你还是手工帮助质量保证人员安装应用，花一些时间，考虑如何将安装过程自动化。这样，只要用户需要，你就可以随时为他们安装系统。要提早实现它，这样让质量保证团队既可以测试应用，又可以测试安装过程^①。如果还是手工安装应用，那么最后把应用部署到生产环境时会发生什么呢？就算公司给你加班费，你也不愿意为不同用户的机器或不同地点的服务器上一遍又一遍地安装应用。

质量保证人员应该测试部署过程

QA should test

有了自动化部署系统后，在项目开发的整个过程中，会更容易适应互相依赖的变化。很可能你在安装系统的时候，会忘记添加需要的库或组件——在任意一台机器上运行自动化安装程序，你很快就会知道什么丢失了。如果因为缺少了一些组

^① 确保他们能提前告诉你运行的软件版本，避免出现混乱。

件或者库不兼容而导致安装失败，这些问题会很快浮现出来。



Andy如是说……

从第一天起就开始交付

一开始就进行全面部署，而不是等到项目的后期，这会有很多好处。事实上，有些项目在正式开发之前，就设置好了所有的安装环境。

在我们公司，要求大家为预期客户实现一个简单的功能演示——验证一个概念的可行性。即使项目还没有正式开始，我们就有了单元测试、持续集成和基于窗口的安装程序。这样，我们就可以更容易更简单地给用户交付这个演示系统：用户所要做的工作，就是从我们的网站上点击一个链接，然后就可以自己在各种不同的机器上安装这个演示系统了。

在签约之前，就能提供出如此强大的演示，这无疑证明了我们非常专业，具有强大的开发能力。



一开始就实现自动化部署应用。使用部署系统安装你的应用，在不同的机器上用不同的配置文件测试依赖的问题。质量保证人员要像测试应用一样测试部署。

切身感受

这些工作都应该是无形的。系统的安装或者部署应该简单、可靠及可重复。一切都很自然。

平衡的艺术

- 一般产品在安装的时候，都需要有相应的软、硬件环境。比如，Java或Ruby的某个版本、外部数据库或者操作系统。这些环境的不同很可能会导致很多技术支持的电话。所以检查这些依赖关系，也是安装过程的一部分。
- 在没有询问并征得用户的同意之前，安装程序绝对不能删除用户的数据。

- 部署一个紧急修复的bug应该很简单，特别是在生产服务器的环境中。你知道这会发生，而且你不想在压力之下，在凌晨3点半，你还在手工部署系统。
- 用户应该可以安全并且完整地卸载安装程序，特别是在质量保证人员的机器环境中。
- 如果维护安装脚本变得很困难，那很可能是一个早期警告，预示着——很高的维护成本（或者不好的设计决策）。
- 如果你打算把持续部署系统和产品CD或者DVD刻录机连接到一起，你就可以自动地为每个构建制作出一个完整且有标签的光盘。任何人想要最新的构建，只要从架子上拿最上面的一张光盘安装即可。

度量真实的进度

“用自己的时间表报告工作进度。我们会用它做项目计划。不用管那些实际的工作时间，每周填满40小时就可以了。”



时间的消逝（通常很快）可以证明：判断工作进度最好是看实际花费的时间而不是估计的时间。

哦，你说早已经用时间表进行了追踪。不幸的是，几乎所有公司的时间表都是为工资会计准备的，不是用来度量软件项目的开发进度的。例如，如果你工作了60个小时，也许你的老板会让你在时间表上只填写40个小时，这是公司会计想看到的。所以，时间表很难真实地反映工作完成状况，因此它不可以用来进行项目计划、评估或表现评估。

即使没有时间表，一些开发人员还是很难面对现实了解自己的真实进度。你曾经听到开发人员报告一个任务完成了80%吗？然而过了一天又一天，一周又一周，那个任务仍然是完成了80%？随意用一个比率进

专注于你的方向

Focus on where
you're going

行度量是没有意义的，这就好比是说80%是**对的**（除非你是政客，否则**对**和**错**应该是布尔条件）。所以，我们不应该去计算工作量完成的百分比，而应该测定还剩下多少工作量没有完成。如果你最初估计这个任务需要40个小时，在开发了35个小时之后，你认为还需要另外30个小时的工作。那就得到了很重要的度量结果（这里诚实非常重要，隐瞒真相毫无意义）。

在你最后真正完成一项任务时，要清楚知道完成这个任务真正花费的时间。奇怪的是，它花费的时间很可能要比最初估计时间长。没有关系，我们希望这能作为下一次的参考。在为下一个任务估计工作量时，可以根据这次经验调整评估。如果你低估了一个任务，评估是2天，它最后花费了6天，那么系数就是3。除非是异常情况，否则你应该对下次估计乘以系数3。你的评估会波动一段时间，有时候过低估计，有时候过高估计。但随着时间的推移，你的评估会与事实接近，你也会对任务所花费的时间有更清楚的认识。



Andy如是说……

登记时间

我的小姨子曾经在某个大型国际咨询公司中工作。每天每隔6分钟她们就得登记她们的时间。

她们甚至有代码来专门记录填表登记时间所花费的时间。这个代码不是0、9999或者一些容易记的代码，而是类似948247401299-44b这么一个临时的代码。

这就是为什么你不愿意把会计部门的规则和约束掺合到项目中的原因。

如果能一直让下一步工作是可见的，会有助于进度度量。最好的做法就是使用待办事项（backlog）。

待办事项就是等待完成的任务列表。当一个任务被完成了，它就会从列表中移除（逻辑上的，而物理上就是把它从列表中划掉，或者标识它是完成的状态）。当添加新任务的时候，先排列它们的优先级，然后加入到待办事项中。你也可以有个人的待办事项、当前迭代的待办事项或者整个项目的待办事项。^①

通过代办事项，就可以随时知道下一步最重要的任务是什么。同时，你的评估技巧也在不停地改进，你也会越来越清楚完成一项任务要花费的时间。

清楚项目的真实进度，是一项强大的技术。



度量剩下的工作量。不要用不恰当的度量来欺骗自己或者团队。要评估那些需要完成的待办事项。

^① 使用待办事项及个人与项目管理工具的列表的更多信息，参考Ship It![RG03]。

Scrum方法中的sprint

在Scrum开发方法中（Sch04），每个迭代被称作sprint，通常为30天时间。sprint的待办事项列表是当前迭代任务列表，它会评估剩下的工作量，显示每个任务还需要多少小时可以完成。

每个工作日，每个团队成员会重新评估完成一个任务还需要多少小时。不管怎么样，只要所有任务的评估总和超过了一个迭代剩余的时间，那么任务就必须移到下一个迭代中开发。

如果每月还有一些剩余的时间，你还可以添加新的任务。这样做，客户一定会非常喜欢。

切身感受

你会觉得很舒服，因为你很清楚哪些任务已经完成，哪些是没有完成的，以及它们的优先级。

平衡的艺术

- 6分钟作为一个时间单位，它的粒度实在太细了，这不是敏捷的做法。
- 一周或者一个月的时间单元，它的粒度太粗了，这也不是敏捷的做法。
- 关注功能，而不是日程表。
- 如果你在一个项目中花费了很多时间来了解你所花费的时间，而没有足够的时间进行工作，那么你在了解你所花费的时间上花费的时间就太多了。听懂了吗？
- 一周工作40个小时，不是说你就有40个小时的编码时间。你需要减去会议、电话、电子邮件以及其他相关活动的时间。

用代码沟通



“如果代码太杂乱以至于无法阅读，就应该使用注释来说明。精确地解释代码做了什么，每行代码都要加注释。不用管为什么要这样编码，只要告诉我们到底是怎么做的就好了。”

通常程序员都很讨厌写文档。这是因为大部分文档都与代码没有关系，并且越来越难以保证其符合目前的最新状况。这不只违反了DRY原则（不要重复你自己，Don't Repeat Yourself，见[HT00]），还会产生使人误解的文档，这还不如没有文档。

建立代码文档无外乎两种方式：利用代码本身；利用注释来沟通代码之外的问题。

如果必须通读一个方法的代码才能了解它做了什么，那么开发人员先要投入大量的时间和精力才能使用它。反过来讲，只需短短几行注释说明方法行为，就可以让生活变得轻松许多。开发人员可以很快了解到它的意图、它的期待结果，以及应该注意之处——这可省了你不少劲儿。

不要用注释来包裹你的代码

Don't comment to cover

应该文档化你所有的代码吗？在某种程度上说，是的。但这并不意味着要注释绝大部分代码，特别是在方法体内部。源代码可以被读懂，不是因为其中的注释，而应该是由于它本身优雅而清晰——变量名运用正确、空格使用得当、逻辑分离清晰，以及表达式非常简洁。

如何命名很重要。程序元素的命名是代码读者必读的部分。^① 通过使用细心挑选的名称，可以向阅读者传递大量的意图和信息。反过来讲，使用人造的命名范式（比如现在已经无人问津的匈牙利表示法）会让代码难以阅读和理解。这些范式中包括的底层数据类型信息，会硬编码在变量名和方法名中，形成脆弱、僵化的

① 在《地海巫师》（The Wizard of Earthsea）系列书籍中，知道一件事物的真实名称可以让一个人对它实施完全的控制。通过名称来进行魔法控制，是文学和神话中一种常用的主题，在软件开发中也是如此。

代码，并会在将来造成麻烦。

使用细心挑选的名称和清晰的执行路径，代码几乎不需要注释。实际上，当Andy和Dave Thomas联手写作第一本关于Ruby编程语言的书籍时（即参考文献[TH01]），他们只要阅读将会在Ruby解释器中执行的代码，几乎就可以把整个Ruby语言的相关细节记录下来了。代码能够自解释，而不用依赖注释，是一件很好的事情。Ruby的创建者松本行弘是日本人，而Andy和Dave除了“sukiyaki”（一种日式火锅）和“sake”（日本清酒）之外，一句日语也不会。

如何界定一个良好的命名？良好的命名可以向读者传递大量的正确信息。不好的命名不会传递任何信息，糟糕的命名则会传递错误的信息。

例如，一个名为readAccount()的方法实际所做的却是向硬盘写入地址信息，这样的命名就被认为是很糟糕的（是的，这确实发生过，参见[HT00]）。

foo是一个具有历史意义、很棒的临时变量名称，但是它没有传递作者的任何意图。要尽量避免使用神秘的变量名。不是说命名短小就等同于神秘：在许多编程语言中通常使用i来表示循环索引变量，s常被用来表示一个字符串。这在许多语言中都是惯用法，虽然都很短，但并不神秘。在这些环境中使用s作为循环索引变量，可真的不是什么好主意，名为indexvar的变量也同样不好。不必费尽心机去用繁复冗长的名字替换大家已习惯的名称。

对于显而易见的代码增加注释，也会有同样的问题，比如在一个类的构造方法后面添加注释//Constructor就是多此一举。但很不幸，这种注释很常见——通常是由过于热心的IDE插入的。最好的状况下，它不过是为代码添加了“噪音”。最坏的状况下，随着时间推进，这些注释则会过时，变得不再正确。

许多注释没有传递任何有意义的信息。例如，对于passthrough()方法，它的注释是“这个方法允许你传递”，但读者能从中得到什么帮助呢？这种注释只会分散注意力，而且很容易失去时效性[假使方法最后又被改名为sendToHost()]。

注释可用来为读者指定一条正确的代码访问路线图。为代码中的每个类或模块添加一个短小的描述，说明其目的以及是否有任何特别需求。对于类中的每个方法，

可能要说明下列信息。

- 目的：为什么需要这个方法？
- 需求（前置条件）：方法需要什么样的输入，对象必须处于何种状态，才能让这个方法工作？
- 承诺（后置条件）：方法成功执行后，对象现在处于什么状态，有哪些返回值？
- 异常：可能会发生什么样的问题？会抛出什么样的异常？

要感谢如RDoc、javadoc和ndoc这样的工具，使用它们可以很方便地直接从代码注释创建有用的、格式优美的文档。这些工具抽取注释，并生成样式漂亮且带有超链接的HTML输出。

下面是一段C#文档化代码的摘录。通常的注释用//开头，要生成文档的注释用///开头（当然这仍然是合法的注释）。

```
using System;
namespace Bank
{
    /// <summary>
    /// A BankAccount represents a customer' s non-secured deposit
    /// account in the domain (see Reg 47.5, section 3).
    /// </summary>
    public class BankAccount
    {
        ...
        /// <summary>
        /// Increases balance by the given amount.
        /// Requirements: can only deposit a positive amount.
        /// </summary>
        ///
        /// <param name="depositAmount">The amount to deposit, already
        /// validated and converted to a Money object
        /// </param>
        ///
        /// <param name="depositSource">Origination of the monies
        /// (see FundSource for details)
        /// </param>
        ///
        /// <returns>Resulting balance as a convenience
        /// </returns>
        ///
        /// <exception cref="InvalidTransactionException">
        /// If depositAmount is less than or equal to zero, or FundSource
        /// is invalid (see Reg 47.5 section 7)
        /// or does not have a sufficient balance.
```

```

    /// </exception>

    public Money Deposit(Money depositAmount, FundSource depositSource)
    {
        ...
    }
}

```

图6-1展示了从C#代码示例中抽取出来的注释生成的文档。用于Java的Javadoc、用于Ruby的Rdoc等工具也都以类似的方式工作。

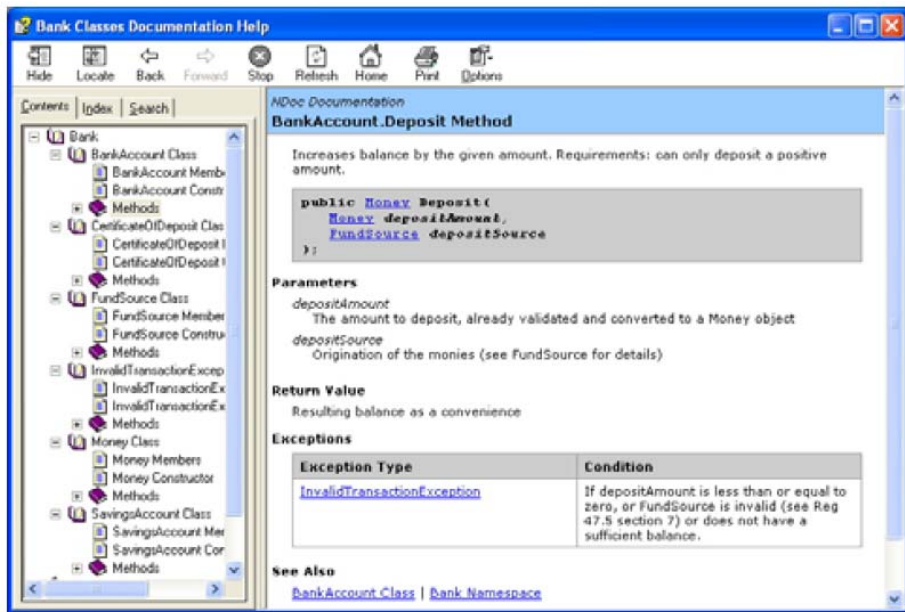


图6-1 使用ndoc从代码中抽取出来的文档

这种文档不只是为团队或组织之外的人准备的。假定你要修改几个月之前所写的代码，如果只要看一下方法头上的注释，就知道需要了解的重要细节，那么修改起来是不是会方便很多？不管怎么说，如果一个方法只有在发生日全食的时候才能正常工作，那么先了解到这个情况（而不必管代码细节）是有好处的，否则岂不是要白白等上10年才有这个机会？

代码被阅读的次数要远超过被编写的次数，所以在编程时多付出一点努力来做好文档，会让你在将来受益匪浅。



用注释沟通。使用细心选择的、有意义的命名。用注释描述代码意图和约束。注释不能替代优秀的代码。

切身感受

注释就像是可以帮助你的好朋友，可以先阅读注释，然后快速浏览代码，从而完全理解它做了什么，以及为什么这样做。

平衡的艺术

- Pascal定理的创始人Blaise Pascal^①曾说，他总是没有时间写短信，所以只好写长信。请花时间去写简明扼要的注释吧。
- 在代码可以传递意图的地方不要使用注释。
- 解释代码做了什么注释用处不那么大。相反，注释要说明为什么会这样写代码。
- 当重写方法时，保留描述原有方法意图和约束的注释。

① 布莱兹·帕斯卡尔（Blaise Pascal，1623—1662），生于法国奥弗涅，卒于巴黎。他是早慧的神童，早夭的天才。主要的数学成就是射影几何方面的Pascal定理，他与Fermat是概率论的奠基者。不过对后世影响最大的，是他的宗教性著作《沉思录》。——译者注

编写内聚的代码



“你要编写一些新的代码，首先要决定的就是把这些代码放在什么地方。其实放在什么地方问题不大，你就赶紧开始吧，看看IDE中现在打开的是哪个类，直接加进去就是了。如果所有的代码都在一个类或组件里面，要找起来是很方便的。”

内聚性用来评估一个组件（包、模块或配件）中成员的功能相关性。内聚程度高，表明各个成员共同完成了一个功能特性或是一组功能特性。内聚程度低的话，表明各个成员提供的功能是互不相干的。

假定把所有的衣服都扔到一个抽屉里面。当需要找一双袜子的时候，要翻遍里面所有的衣服——裤子、内衣、T恤等——才能找到。这很麻烦，特别是在赶时间的时候。现在，假定把所有的袜子都放在一个抽屉里面（而且是成双放置的），全部的T恤放在另外一个抽屉中，其他衣服也分门别类。要找到一双袜子，只要打开正确的抽屉就可以了。

与此类似，如何组织一个组件中的代码，会对开发人员的生产力和全部代码的可维护性产生重要影响。在决定创建一个类的时候，问问自己，这个类的功能是不是与组件中其他某个类的功能类似，而且功能紧密相关。这就是组件级的内聚性。

类也要遵循内聚性。如果一个类的方法和属性共同完成了一个功能（或是一系列紧密相关的功能），这个类就是内聚的。

看看Charles Hess先生于1866年申请的专利，“可变换的钢琴、睡椅和五斗柜”（见图7-1）。根据他的专利说明，它提供了“……附加的睡椅和五斗柜……以填满钢琴下未被使用的空间……”。接下来他说明了为什么要发明这个可变换的钢琴。读者可能已经见过类似这种发明的项目代码结构了，而且也许其中有你的份。这个发明不具备任何内聚性，任何一个人都可以想象得到，要维护这个怪物（比如换垫子、调钢琴等）会是多么困难。

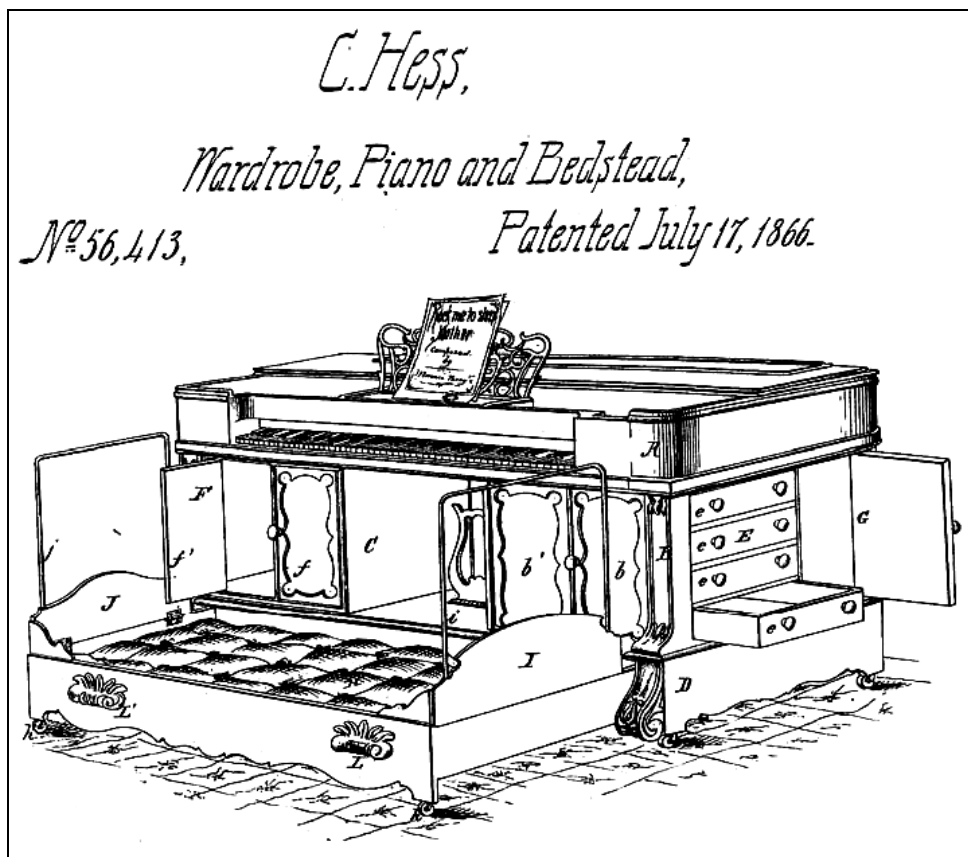


图7-1 美国专利56 413：可变换的钢琴、睡椅和五斗柜

看看最近的例子。Venkat曾经见过一个用ASP开发的、有20个页面的Web应用。每个页面都以HTML开头，并包含大量VBScript脚本，其中还内嵌了访问数据库的SQL语句。客户当然会认为这个应用的开发已经失去了控制，并且无法维护。如果每个页面都包括展示逻辑、业务逻辑和访问数据的代码，就有太多的东西都堆在一个地方了。

假定要对数据库的表结构进行一次微调。这个微小的变化会导致应用中所有的页面发生变化，而且每个页面中都会有多处改变——这个应用很快就变成了一场灾难。

如果应用使用了中间层对象（比如一个COM组件）来访问数据库，数据库表结构变更所造成的影响就可以控制在一定的范围之内，代码也更容易维护。

低内聚性的代码会造成很严重的后果。假设有这样一个类，实现了五种完全不相干的功能。如果这5个功能的需求或细节发生了变化，这个类也必须跟着改变。如果一个类（或者一个组件）变化得过于频繁，这样的改变会对整个系统形成“涟漪效应”，并导致更多的维护和成本的发生。考虑另一个只实现了一种功能的类，这个类变化的频度就没有那么高。类似地，一个更具内聚性的组件不会有太多导致其变化的原因，也因此而更加稳定。根据单一职责原则（查看《敏捷软件开发：原则、模式与实践》[Mar02]），一个模块应该只有一个发生变化的原因。

一些设计技巧可以起到帮助作用。举例来说，我们常常使用模型-视图-控制器（MVC）模式来分离展示层逻辑、控制器和模型。这个模式非常有效，因为它可以让开发人员获得更高的内聚性。模型中的类包含一种功能，在控制器中的类包含另外的功能，而视图中的类则只关心UI。

内聚性会影响一个组件的可重用性。组件粒度是在设计时要考虑的一个重要因素。根据重用发布等价原则（[Mar02]）：重用的粒度与发布的粒度相同。这就是说，程序库用户所需要的，是完整的程序库，而不只是其中的一部分。如果不能遵循这个原则，组件用户就会被强迫只能使用所发布组件的一部分。很不幸的是，他们仍然会被不关心的那一部分的更新所影响。软件包越大，可重用性就越差。



让类的功能尽量集中，让组件尽量小。要避免创建很大的类或组件，也不要创建无所不包的大杂烩类。

切身感受

感觉类和组件的功能都很集中：每个类或组件只做一件事，而且做得很好。bug很容易跟踪，代码也易于修改，因为类和组件的责任都很清晰。

平衡的艺术

- 有可能会把一些东西拆分成很多微小的部分，而使其失去了实用价值。当你需要一只袜子的时候，一盒棉线不能带给你任何帮助。^①
- 具有良好内聚性的代码，可能会根据需求的变化，而成比例地进行变更。考虑一下，实现一个简单的功能变化需要变更多少代码。^②

① 你可以把这个叫作“意大利面OO”系统。

② 本书的一位检阅者告诉我们这样一个系统，向一个表单中添加一个字段，需要16名团队成员和6名经理的同意。这是一个很清晰的警告信号，说明系统的内聚性很差。

根据契约进行替换



“深层次的继承是很棒的。如果你需要其他类的函数，直接继承它们就好了！不要担心你创建的新类会造成破坏，你的调用者可以改变他们的代码。这是他们的问题，而不是你的问题。”

保持系统灵活性的关键方式，是当新代码取代原有代码之后，其他已有的代码不会意识到任何差别。例如，某个开发人员可能想为通信的底层架构添加一种新的加密方式，或者使用同样的接口实现更好的搜索算法。只要接口保持不变，开发人员就可以随意修改实现代码，而不影响其他任何现有代码。然而，说起来容易，做起来难。所以需要一点指导来帮助我们正确地实现。因此，去看看Barbara Liskov的说法。

Liskov替换原则[Lis88]告诉我们：任何继承后得到的派生类对象，必须可以替换任何被使用的基类对象，而且使用者不必知道任何差异。换句话说，某段代码如果使用了基类中的方法，就必须能够使用派生类的对象，并且自己不必进行任何修改。

这到底意味着什么？假定某个类中有一个简单的方法，用来对一个字符串列表进行排序，然后返回一个新的列表。并用如下的方式进行调用：

```
utils = new BasicUtils();  
...  
sortedList = utils.sort(aList);
```

现在假定开发人员派生了一个BasicUtils的子类，并写了一个新的sort()方法，使用了更快、更好的排序算法：

```
utils = new FasterUtils();  
...  
sortedList = utils.sort(aList);
```

注意对sort()的调用是完全一样的，一个FasterUtils对象完美地替换了一个BasicUtils对象。调用utils.sort()的代码可以处理任何类型的utils对象，而且可以正常工作。

但如果开发人员派生了一个`BasicUtils`的子类，并改变了排序的意义——也许返回的列表以相反的顺序进行排列——那就严重违反了Liskov替换原则。

要遵守Liskov替换原则，相对基类的对应方法，派生类服务（方法）应该不要求更多，不承诺更少；要可以进行自由的替换。在设计类的继承层次时，这是一个非常重要的考虑因素。

继承是OO建模和编程中被滥用最多的概念之一。如果违反了Liskov替换原则，继承层次可能仍然可以提供代码的可重用性，但是将会失去可扩展性。类继承关系的使用者现在必须要检查给定对象的类型，以确定如何针对其进行处理。当引入了新的类之后，调用代码必须经常重新评估并修正。这不是敏捷的方式。

但是可以借用一些帮助。编译器可以帮助开发人员强制执行Liskov替换原则，至少在某种程度上是可以达到的。例如，针对方法的访问修饰符。在Java中，重写方法的访问修饰符必须与被重写方法的修饰符相同，或者可访问范围更加宽大。也就是说，如果基类方法是受保护的，那么派生重写方法的修饰符必须是受保护的或者公共的。在C#和VB.NET中，被重写方法与重写方法的访问保护范围必须完全相同。

考虑一个带有`findLargest()`方法的类`Base`，方法中抛出一个`IndexOutOfRangeException`异常。基于文档，类的使用者会准备抓住可能被抛出的异常。现在，假定你从`Base`类继承得到类`Derived`，并重写了`findLargest()`方法，在新的方法中抛出了一个不同的异常。现在，如果某段代码期待使用`Base`类对象，并调用了`Derived`类的实例，这段代码就有可能接收到一个意想不到的异常。你的`Derived`类就不能替换使用到`Base`类的地方。在Java中，通过不允许重写方法抛出任何新的检查异常避免了一个问题，除非异常本身派生自被重写方法抛出的异常类（当然，对于像`RuntimeException`这样的未检查异常，编译器就不能帮你了）。

不幸的是，Java也违背了Liskov替换原则。`java.util.Stack`类派生自`java.util.Vector`类。如果开发人员（不小心）将`Stack`对象发送给一个期待`Vector`实例的方法，`Stack`中的元素就可能被以与期望的行为不符的顺序被插入或删除。

当使用继承时，要想想派生类是否可以替换基类。如果答案是不能，就要问问自己为什么要使用继承。如果答案是希望在编写新类的时候，还要重用基类中的代码，也许要考虑转而使用聚合。聚合是指在类中包含一个对象，并且该对象是其他类的实例，开发人员将责任委托给所包含的对象来完成（该技术同样被称为委托）。

针对is-a关系使用继承；针对has-a或uses-a关系使用委托

Use inheritance for *is-a*;
use delegation for

图8-1中展示了委托与继承之间的差异。在图中，一个调用者调用了Called Class中的methodA()，而它将会通过继承直接调用Base Class中的方法。在委托的模型中，Called Class必须要显式地将方法调用转向包含的委托方法。

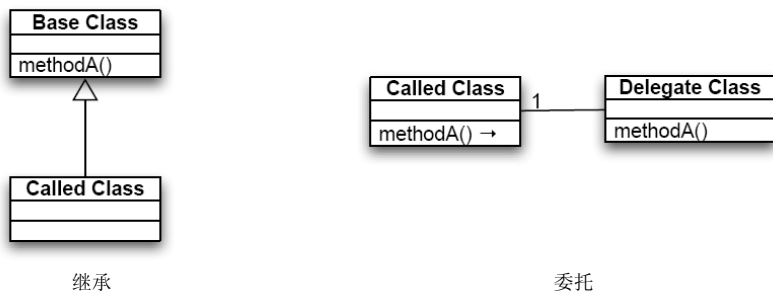


图8-1 委托与继承

那么继承和委托分别在什么时候使用呢？

- 如果新类可以替换已有的类，并且它们之间的关系可以通过is-a来描述，就要使用继承。
- 如果新类只是使用已有的类，并且两者之间的关系可以描述为has-a或是uses-a，就使用委托吧。

开发人员可能会争辩说，在使用委托时，必须要写很多小方法，来将方法调用指向所包含的对象。在继承中，不需要这样做，因为基类中的公共方法在派生类中就已经是可用的了。仅凭这一点，并不能构成使用继承足够好的理由。

你可以开发一个好的脚本或是好的IDE宏，来帮助编写这几行代码，或者使用一种更好的编程语言/环境，以支持更自动化形式的委托（比如Ruby这一点就做得不错）。



通过替换代码来扩展系统。通过替换遵循接口契约的类，来添加并改进功能特性。要多使用委托而不是继承。

切身感受

这会让人觉得有点鬼鬼祟祟的，你可以偷偷地替换组件代码到代码库中，而且其他代码对此一无所知，它们还获得了新的或改进后的功能。

平衡的艺术

- 相对继承来说，委托更加灵活，适应力也更强。
- 继承不是魔鬼，只是长久以来被大家误解了。
- 如果你不确定一个接口做出了什么样的承诺，或是有什么样的需求，那就很难提供一个对其有意义的实现了。

报告所有的异常



“不要让程序的调用者看到那些奇怪的异常。处理它们是你的责任。把你调用的一切都包起来，然后发送自己定义的异常——或者干脆自己解决掉。”

从事任何编程工作，都要考虑事物正常状况下是如何运作的。不过更应该想一想，当出现问题——也就是事情没有按计划进行时，会发生什么。

在调用别人的代码时，它也许会抛异常，这时我们可以试着对其处理，并从失败中恢复。当然，要是在用户没有意识到的情况下，可以恢复并继续正常处理流程，这就最好不过了。要是不能恢复，应该让调用代码的用户知道，到底是哪里出现了问题。

不过也不尽然。**Venkat**曾经在使用一个非常流行的开源程序库（这里就不提它的名字了）时倍受打击。他调用的一个方法本来应该创建一个对象，可是得到的却是`null`引用。涉及的代码量非常少，而且没有其他代码发生联系，也很简单。所以从他自己写的这块代码的角度来看，不太可能出问题，他摸不到一点头绪。

幸好这个库是开源的，所以他下载了源代码，然后带着问题检查了相关的方法。这个方法调用了另外的方法，那个方法认为他的系统中缺少了某些必要的组件。这个底层方法抛出了带有相关信息的异常。但是，上层方法却偷偷地用没有异常处理代码的空`catch`代码块，把异常给忽略掉了，然后就抛出一个`null`。**Venkat**所写的代码根本不知道到底发生了什么，只有通过阅读程序库的代码，他才能明白这个问题，并最后安装了缺失的组件。

像Java中那样的检查异常会强迫你捕捉异常，或是把异常传播出去。可是有些开发人员会采取临时的做法：捕捉到异常后，为了不看到编译器的提示，就把异常忽略掉。这样做很危险——临时的补救方式很容易被遗忘，并且会进入到生产系统的代码中。必须要处理所有的异常，倘若可以，从失败中恢复再好不过。如果不能处理，就要把异常传播到方法的调用者，这样调用者就可以尝试对其进行

处理了（或者以优雅的方式将问题的信息告诉给用户，见习惯37）。

听起来很明白，是吧？其实不像想象得那么容易。不久前有一条新闻，提到一套大型航空订票系统中发生了严重的问题。系统崩溃，飞机停飞，上千名旅客滞留机场，整个航空运输系统数天之内都乱作一团。原因是什么？在一台应用服务器上发生了一个未检查异常。

也许你很享受CNN新闻上提到你名字的感觉，但是你不太可能希望发生这样的情形。



处理或是向上传播所有的异常。不要将它们压制不管，就算是临时这样做也不行。在写代码时要估计到会发生的问题。

切身感受

当出现问题时，心里知道能够得到抛出的异常。而且没有空的异常处理方法。

平衡的艺术

- 决定由谁来负责处理异常是设计工作的一部分。
- 不是所有的问题都应该抛出异常。
- 报告的异常应该在代码的上下文中有实际意义。在前述的例子中，抛出一个 `NullPointerException` 看起来也许不错，不过这就像抛出一个 `null` 对象一样，起不到任何帮助作用。
- 如果代码中会记录运行时调试日志，当捕获或是抛出异常时，都要记录日志信息；这样做对以后的跟踪工作很有帮助。
- 检查异常处理起来很麻烦。没人愿意调用抛出31种不同检查异常的方法。这是设计上的问题：要把它解决掉，而不是随便打个补丁就算了。
- 要传播不能处理的异常。

做代码复查



“用户是最好的测试人员。别担心——如果哪里出错了，他们会告诉我们的。”

代码刚刚完成时，是寻找问题的最佳时机。如果放任不管，它也不会变得更好。

代码复查和缺陷移除

要寻找深藏不露的程序bug，正式地进行代码检查，其效果是任何已知形式测试的两倍，而且是移除80%缺陷的唯一已知方法。

——Capers Jones的《估算软件成本》[Jon98]

正如Capers Jones指出的，代码复查或许是找到并解决问题的最佳方式。然而，有时很难说服管理层和开发人员使用它来完成开发工作。

管理层担心进行代码复查所耗费的时间。他们不希望团队停止编码，而去参加长时间的代码复查会议。开发人员对代码复查感到担心，允许别人看他们的代码，会让他们有受威胁的感觉。这影响了他们的自尊心。他们担心在情感上受到打击。

作者参与过的项目中，只要实施了代码复查，其成果都是非常显著的。

Venkat最近参与了一个日程安排非常紧凑的项目，团队不少成员都是没有多少经验的开发者。通过严格的代码复查过程，他们可以提交质量极高而且稳定的代码。当开发人员完成某项任务的编码和测试后，在签入源代码控制系统之前，会有另一名开发人员对代码做彻底的复查。

这个过程修复了很多问题。噢，代码复查不只针对初级开发者编写的代码——团队中每个开发人员的代码都应该进行复查，无论其经验丰富与否。

那该如何进行代码复查呢？可以从下面这些不同的基本方式中进行选择。

□ **通宵复查。**可以将整个团队召集在一起，预定好美食，每个月进行一次“恐

怖的代码复查之夜”。但这可能不是进行代码复查最有效的方式（而且听起来也不太敏捷）。大规模团队的复查会议很容易陷入无休止的讨论之中。大范围的复查不仅没有必要，而且有可能对整个流程造成损害。我们不建议这种方式。

- **捡拾游戏**。当某些代码编写完成、通过编译、完成测试，并已经准备签入时，其他开发人员就可以“捡拾”起这些代码开始复查。类似的“提交复查”是一种快速而非正式的方式，保证代码在提交之前是可以被接受的。为了消除行为上的惯性，要在开发人员之间进行轮换。比如，如果Joey的代码上次是由Jane复查的，这次不妨让Mark来复查。这是一种很有效的技术。^①
- **结对编程**。在极限编程中，不存在一个人独立进行编码的情况。编程总是成对进行的：一个人在键盘旁边（担任司机的角色），另一个人坐在后面担任导航员。他们会不时变换角色。有第二双眼睛在旁边盯着，就像是在进行持续的代码复查活动，也就不必安排单独的特定复查时间了。

在代码复查中要看什么呢？你可能会制订出要检查的一些特定问题列表（所有的异常处理程序不允许空，所有的数据库调用都要在包的事务中进行，等等），不过这里是一个可供启动的最基本的检查列表。

- 代码能否被读懂和理解？
- 是否有任何明显的错误？
- 代码是否会对应用的其他部分产生不良影响？
- 是否存在重复的代码（在复查的这部分代码中，或是在系统的其他部分代码）？
- 是否存在可以改进或重构的部分？

此外，还可以考虑使用诸如Similarity Analyzer或Jester这样的代码分析工具。如果这些工具产生的静态分析结果对项目有帮助，就把它集成到持续构建中去吧。



复查所有的代码。对于提升代码质量和降低错误率来说，代码复查是无价之宝。如果以正确的方式进行，复查可以产生非常实用而高效的成果。要让不同的开发人员在每个任务完成后复查代码。

^① 要了解这种方式的更多细节，查看*Ship It!* [RG05]一书。

切身感受

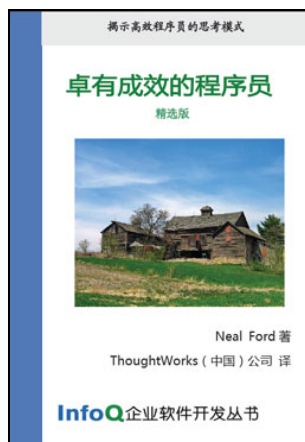
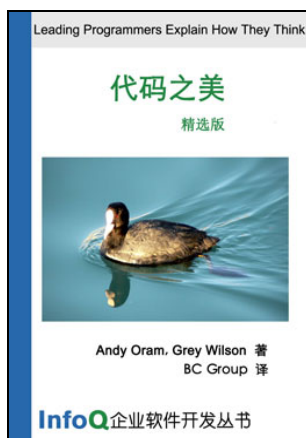
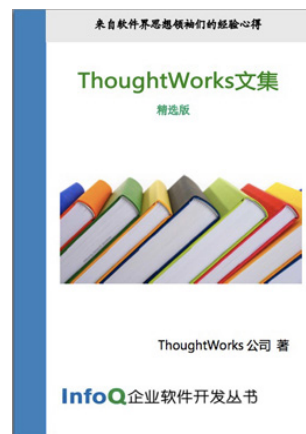
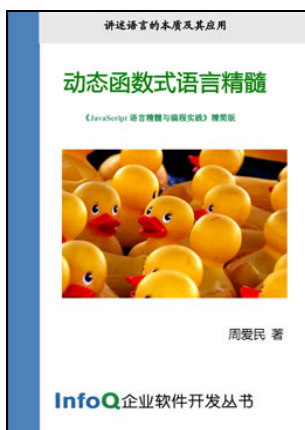
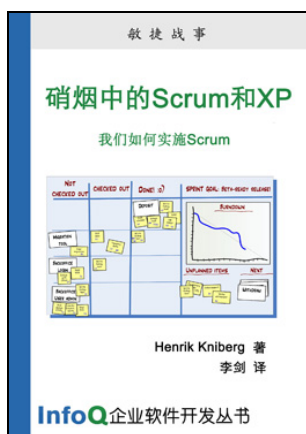
代码复查随着开发活动持续进行，而且每次针对的代码量相对较少。感觉复查活动就像是项目正在进行的一部分，而不是一种令人畏惧的事情。

平衡的艺术

- 不进行思考、类似于橡皮图章一样的代码复查没有任何价值。
- 代码复查需要积极评估代码的设计和清晰程度，而不只是考量变量名和代码格式是否符合组织的标准。
- 同样的功能，不同开发人员的代码实现可能不同。差异并不意味着不好。除非你可以让某段代码明确变得更好，否则不要随意批评别人的代码。
- 如果不及时跟进讨论中给出的建议，代码复查是没有实际价值的。可以安排跟进会议，或者使用代码标记系统，来标识需要完成的工作，跟踪已经处理完的部分。
- 要确保代码复查参与人员得到每次复查活动的反馈。作为结果，要让每个人知道复查完成后所采取的行动。

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