第 I 部分　敏捷开发

“人与人之间的交互是复杂难懂的，效果总是难以预期，但其重要性却远远高于工作中的其他任何一个方面。”

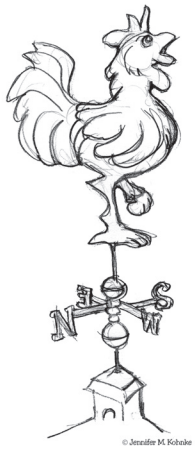
—— 迪马可 & 李斯特，《人件》

原则、模式和实践都很重要，但让它们真正起作用的是人。正如科博恩（Alistair Cockburn）所说：“过程和技术是项目成效的次要因素，首要的因素是人。”

如果把程序员团队看成一个由过程驱动的组件化系统，就没有办法对他们进行管理了。人不是“插件式的编程单元”。要想项目取得成功，就必须构建成高度协作的自组织团队。

鼓励这种特性团队的公司比那些把软件开发团队看作是团伙或是一群乌合之众的公司具备更大的竞争优势。凝聚力强的软件团队才能发挥出最强大的软件开发能力。

第 1 章　敏捷实践



“教堂屋顶上的风标，即使是由钢铁制成，如果不懂得顺势而为，也很快会被暴风摧毁。”

——海涅[①]

我们许多人都经历过因为缺乏实践指导而导致的项目梦魇。有效实践的缺失会带来不确定性、重复的错误以及徒劳无功。客户失望于延期的进度、增长的预算和糟糕的质量；开发者也感到沮丧，因为他们用的时间更长却写出了质量更低劣的软件。

一旦经历这样的惨败，人们就会害怕重蹈覆辙。这种恐惧感促使我们创造出一种过程，以此来约束自己的活动并要求特定的输出或产出物（artifact）。我们根据过去的经验制定出这些约束和产出，挑选出之前项目中看似工作得还不错的方法。我们希望这些方法能再次奏效，从而消除自己内心的恐惧。

不过，项目没有简单到只用一些约束和产出物就可以避免错误发生。随着错误的持续产生，我们在诊断之后又增加了更多的约束和产出物要求，目的是防止将来犯同样的错误。在经历过许多项目后，这些压得我们喘不过气来的厚重过程反过来严重影响了我们完成工作的能力。

① 中文版编注：海涅（Christian Johann Heinrich Heine，1797 —　　），歌德之后德国最重要的诗人。早期的创作主要是抒情诗《歌集》，有不少作曲家为海涅的诗谱曲，差不多有三千多首，其中包括门德尔松谱曲的《乘着歌声的翅膀》。

厚重的过程会产生事与愿违的结果，它会在一定程度上拖慢团队的进度，拖垮项目的预算，也会降低团队的响应力，让软件质量变得不堪。不幸的是，这又会加剧很多团队确信他们需要更多过程的认知。因此，在失控的恶性膨胀下，团队过程的厚重程度愈演愈烈。

失控的恶性膨胀很好地描述了 2000 年前后很多软件公司的状况。尽管还有不少团队在工作中并没有使用过程方法，但是很多公司都逐渐采纳了大而厚重的过程方法，大公司更是如此。（参见附录 C）

敏捷联盟

2001 年初，由于观察到很多公司的软件团队被困在不断增长的过程方法的泥泞里，一群行业专家挺身而出，拟定了一系列可以让团队快速响应变化的价值观和原则。他们成立了敏捷联盟。在随后的几个月里，共同创造出一份价值陈述。这便是《敏捷宣言》（agilealliance.org）。

敏捷宣言

敏捷软件开发宣言

我们正在这样做以及帮助其他人这样做来揭示更好的开发软件方法。借由这项工作，我们开始重视

个体交互　优先于　过程和工具

可以工作的软件　优先于　面面俱到的文档

客户合作　优先于　合同谈判

响应变化　优先于　遵循计划

尽管右项有其价值，但我们更加重视左项的价值。

Kent Beck	Mike Beedle	Arie van Bennekum	Alistair Cockburn
Ward Cunningham	Martin Fowler	James Grenning	Jim Highsmith
Andrew Hunt	Ron Jeffries	Jon Kern	Brian Marick
Robert C. Martin	Steve Mellor	Ken Schwaber	Jeff Sutherland
Dave Thomas			

个体交互优先于过程和工具

人是项目成功的最关键因素。如果缺少优秀的人，即使是很好的过程方法，也无法避免项目失败，反之，不好的过程方法倒是可以让最优秀的人变得低效。一群优秀的人如果无法作为一个团队运作，也会导致项目遭受巨大的失败。

优秀的人未必非得是顶尖的程序员，他们可以是处于平均水平的程序员，但可以和其他人合作无间。默契的合作、沟通和交流比单纯的编程才能重要得多。一群沟通顺畅的、处于平均水平的程序员比一群无法合作的明星程序员更有可能获得成功。

合适的工具对工作大有裨益。编译器、IDE 和源代码控制这些工具，对团队而言非常重要。不过，工具的重要性也可能被过分强调了。笨重的工具和缺乏工具，两者的结果一样坏。

我的建议是从最简单的开始。不要预先假设手头上的工具无法支撑需求，除非在尝试后发现确实如此。别去购买先进又昂贵的源代码版本控制工具，在你能证明无法支撑需求之前，先找个免费的用着。别去购买最佳 CASE 工具的团队证书（license），在你能给出充足的需要理由之前，先用白板和图纸。在使用顶级的数据库之前，先用没有相对结构的代替。也不要假设更大更好的工具就能自动帮你做好事情。通常情况下，用它们，弊大于利。

记住，团队建设比环境搭建更重要。很多团队和管理者都搞错了，以为搭建好的环境，团队就能自动凝聚到一起。应该首先建设团队，然后让他们自己从基本需求出发，去配置环境。

可以工作的软件优先于面面俱到的文档

没有文档的软件就是一场灾难。代码不是用来沟通系统基本原理和结构的理想介质。相反，团队需要提供可读的文档，这些文档描述了系统和设计决策的基本原理。

不过，太多的文档比没有文档更可怕。庞大的软件文档需要花很多时间去写，还需要更多的时间保持和代码的同步。如果无法保持同步，它们就会变成庞大且费解的谎言，成为误解的罪魁祸首。

让团队编写和维护基本原理与结构的文档通常都是有益的，但是这些文档需要短小精悍，言简意赅。所谓短，说的是最多只有 12 到 24 页；所谓精，则是说它应该描述整体设计的基本原理，只包含系统中高层次的结构。

如果团队只有简短的基本原理和结构的文档，那么如何在工作中培训新人呢？答案是和他们一起工作。通过坐在旁边亲自辅导来传递知识，通过紧密的培训和交互来使他们成为团队的一份子。

代码和团队是给新人传递信息的最佳文档。代码不会说谎，虽然从代码中提取基本原理和意图比较困难，但是代码却是无二义性信息的来源。团队成员的大脑里关于系统的路线图无时无刻不在改变。传递这样的路线图，除了人与人之间的交互，没有其他更快、更高效的方式了。

很多团队迷失在追求文档而非软件上。这通常是个巨大的错误。有条简单的规则，称为“文档的马丁第一原则”，意在防止此类错误发生：

除非文档紧急且重要，否则不要写。

客户合作优先于合同谈判

软件无法像商品一样订购。写一份关于软件的描述，然后让其他人在固定的时间内以固定的价格开发完成，这种做法是不可行的。尝试这种做法的软件项目频繁失败，有时候，这种失败是惊人的。

对于公司的管理者而言，告诉开发同事自己的需求，期待他们离开一会儿，回来的时候就带来一个满足需求的系统，这一做法是很吸引人。然而，这种操作只会带来低劣的质量，进而导致项目失败。

成功的项目需要客户规律和频繁的反馈。不同于依赖一纸合同和一份工作量陈述，软件的客户应该和开发团队在一起工作，给团队的工作提供频繁的反馈。

指定需求、时间计划和经费的项目从根本上就是错误的。在大多数情况下，这些条约远在项目完成（有时候远远在合同被签订之前）之前就变得毫无意义。保证开发团队和客户能一起工作才是最好的合同。

举一个合同成功的例子。1994 年，我谈了一个 50 万行代码的大型遗留项目。每月，我们开发团队只拿相对较低的报酬，在交付大的功能模块时，才会获得全款。那些功能模块并没有被合同详细指定。相反，合同只是声明当功能模块通过客户的验收测试后，才会支付相应的酬劳。验收测试的细节也没有在合同中详细指定。

在这个项目中，我们和客户密切合作。基本上，每周五我们就发布一个软件版本。在下周的周一或者周二，客户提交一系列的修改。我们一起对这些修改进行优先级排序，然后放入接下来的计划中。客户和我们一起工作让验收测试变得很顺畅，他也知道哪些功能模块何时能满足需要，因为他亲眼见证了软件的演进过程。

这个项目的需求处于不断变化的状态。重大的变化并不少见。有整个功能模块被移除，也有新的功能模块被插入。不过，合同和项目都顺利生存下来了。成功的关键是和客户的紧密合作，而且合同保证了合作关系，而不是尝试划定范围的细节以及固定经费下的时间计划。

响应变化优先于遵循计划

能否响应变化往往决定着软件项目的成功和失败。制定计划时，需要保持灵活，时刻准备着来自业务和技术的改变。

软件项目的过程无法规划得太远。首先，商业环境很可能变化，从而导致需求发生更改。其次，客户看到系统开始工作之后很可能改变需求。最后，即便知道并且确信需求不会改变，也很难估算出需要多久才能开发完成。

对于新晋管理者，做一个好看的项目全局 PERT[①] (Program Evaluation and Review Technique) 图或者甘特图贴到墙上，是很有吸引力的。他们会觉得这幅图表意味着掌握全局，他们可以追踪个人的任务项，打个叉表示任务完成。他们也可以比较计划完成时间和实际完成时间的差异，然后做出反应。

这类结构的图表有什么真正的缺陷呢？一旦团队深入了解系统，客户明白了自己的需要，图表上的某些任务就变得没有必要了，其他的任务会被发掘出来。简单地说，计划会变化，而不仅仅是时间会变。

更好的计划策略是为接下来的两周安排详细计划，接下来三个月安排比较粗糙的计划，再远就是非常粗略的计划。对于接下来的三个月时间，只要粗略地知道需求就好。对于一年以后的系统，只要有个模糊的想法就行。

这种逐渐模糊的计划方式意味着只有紧急的任务才值得制定详细计划。一旦详细计划被制定出来，就很难改变，因为团队需要大量时间和精力来实施。然而，由于这个计划只管理几周时间，剩下的时间可以保持灵活。

原则

上述价值启发了以下 12 条原则，即敏捷实践区别于厚重过程方法的关键特点。

---

① 全称为 Program Evaluation and Review Technique，计划评审技术，主要针对不确定性较高的工作项目，以网络图来规划整个项目，排定期望的项目日程。

1. **我们最看重的是，通过及早、持续交付有价值的软件，来满足客户的需求。**

   MIT《斯隆管理评论》杂志刊登过一篇文章，分析了对公司构建高质量产品有帮助的软件开发过程。这篇文章发现了很多对最终系统质量有重大影响的实践。其中一项实践表明，尽早交付部分功能的系统和系统质量之间有很强的相关性。文章指出，初期交付的功能越少，最终交付的系统质量越高。文中另一项发现是，增量频繁交付和最终质量也有强相关性。交付越频繁，最终的质量也越高。敏捷实践会尽早地、频繁地交付。我们努力在项目刚开始的几周内就交付一个具有基本功能的系统。然后，以每两周增量迭代的方式，持续地交付系统。如果客户认为目前的功能已经足够了，他们就可以把系统部署到生产环境。或者，简单评审一下当前已有的功能，然后指出想要的变更。

2. **欢迎需求有变化，即使是在软件开发后期。轻量级的敏捷流程可以驾驭任何有利于提升客户竞争优势的变化。**

   这是一份关于态度的声明。敏捷过程的参与者不惧怕变化。他们认为改变需求是好事，因为这意味着团队学会了很多可以满足市场需要的知识。敏捷团队会非常努力地保证软件的灵活性，这样，当需求变化时，对系统造成的影响是很小的。在本书的后续部分，我们会学习一些面向对象设计的原则和模式，这些内容会帮助我们维持这种灵活性。

3. **频繁交付能用起来的软件，频率从两周到两个月，倾向于更短的时限。**

   尽早（项目刚开始之后的几周）、频繁（此后每隔几周）地交付可工作的软件。我们不赞成交付大量的文档或者计划，因为它们不是真正的交付物，我们关注的是交付满足客户需要的软件。

4. **业务人员和开发人员必须合作，这样的合作贯穿于整个项目中的每一天。**

   为了保证项目能以敏捷的方式开展，客户、开发人员以及相关利益者就必须进行有意义的、频繁的交互。软件项目不像发射之后就能自动导航的武器，它必须不断地引导。

5. **围绕着主动性强的个人来立项。为他们提供必要的环境和支持，同时信任他们能够干成事情。**

   给他们提供所需要的环境和支持，并且相信他们能够完成工作。在敏捷项目中，人被认为是取得成功最关键的因素。所有其他的因素，比如过程、环境和管理等，都是次要的，并且，当这些因素对人有负面影响时，就要改变它们。例如，如果办公环境对团队造成阻碍，就必须改造办公环境。如果某些过程形成阻碍，这些过程就得整改。

6. **开发团队内部以及跨团队之间，最有效和最高效的信息传递方式是，面对面进行对话。**

   在敏捷项目中，成员面对面交谈。面对面交谈是最主要的沟通方式，也会有文档，但是文档不会包含项目的所有信息。敏捷团队不需要书面的规格、计划或者设计文档，

除非这些文档是紧急且重要的，团队成员才会去写，但这不是默认的沟通方式，面对面交谈才是。

7. **能用起来的软件，就是衡量进度的基本依据。**

敏捷项目是通过统计当前软件满足多少客户的需求来度量项目进度的。他们不会根据所处的开发阶段、已经写好的文档的数量或者已经创建的基础设施代码行数来度量进度。只有当 30% 的必需功能可以工作时，才可以确定 30% 的完成度。

8. **敏捷流程倡导可持续的开发。发起人、开发人员和用户都能够长期保持一种稳定、可持续的节拍。**

责任人、开发者和用户应该能保持长期的、恒定的开发速度敏捷项目不是 50 米短跑，而是马拉松长跑。团队不是一开始马力全开并试图在项目开发期间维持那个速度。相反，他们会以快速但可持续的速度前进。跑得太快会导致团队精疲力尽，短期冲刺，直至崩溃。敏捷团队会测量自己的速度。他们不允许自己过劳，不会借用明天的精力多完成一点儿今天的工作。他们工作在一个可以让整个项目开发始终保持最高质量标准的速度上。

9. **持续保持对技术卓越和设计优良的关注，这是强化敏捷能力的前提。**

高质量是高开发速度的关键。保持软件尽可能的整洁健壮是开发软件的快车道。因而，所有的敏捷团队都致力于编写最高质量的代码。他们不会弄乱代码后告诉自己，有时间了再去清理。如果今天弄乱了代码，他们就会在当天下班前清理干净。

10. **简洁为本，极简是消除浪费的艺术。**

敏捷团队不会试图去构建那些华而不实的系统，他们总是使用和目标一致的最简单的方法。他们并不过多关注预测未来会出现的问题，也不会在今天就做出防卫。相反，他们会在今天以最高的质量完成最简单的工作，深信即便未来出现了问题，也可以从容处理。

11. **最好的架构、需求和设计，是从自组织团队中涌现出来的。**

敏捷团队是自组织的团队。任务不是从外部分配给单个团队成员，而是分配给整个团队，然后再由团队来确定完成任务最佳方式。敏捷团队的成员共同解决项目中的所有问题。每位成员都有权参与项目所有的方面参与权力。不存在单个成员对系统架构、需求或者测试负责的情况。整个团队共同承担那些责任，每位成员都能影响它们。

12. **按固定的时间间隔，团队反思提效的方式，进而从行为上做出相应的优化和调整。**

每隔一定时间，团队会对如何更有效地工作进行反省，然后做出相应的调整敏捷团队会不断地对团队的组织方式、规则、惯例和关系等进行调整。敏捷团队知道团队所处的环境在不断变化，并且知道为了保持团队的敏捷性，就必须适应环境变化。

小结

每位软件开发人员、每个开发团队的职业目标，都是尽可能给他们的雇主和客户交付价值。可是，我们的项目以令人沮丧地速度失败或者未能交付价值。虽然在项目中采用过程方法是出于好意，但是膨胀的过程方法对于这些项目的失败至少是需要负一点责任的。敏捷软件开发的原则和价值观形成一套帮助团队打破过程膨胀恶性循环的方法。

在写本书的时候，已经有很多敏捷过程可供大家选择。包括SCRUM（www.controlchaos.com），Crystalcrystal（methodologies.org），特征驱动软件开发（Feature Driven Development，FDD），Java Modeling In Color With UML: Enterprise Components and Process, Peter Coad, Eric Lefebvre, and Jeff De Luca, Prentice Hall, 1999），自适应软件开发（Adaptive Software Development，ADP）[Highsmith2000]以及最重要的极限编程（eXtreme Programming，XP）[Beck 1999], [NewKirk 2001]。

参考文献

1. Beck, Kent.*Extreme Programming Explained: Embracing Change.* Reading, MA: Addison-Wesley, 1999. 中文版《极限编程详解：拥抱变化》
2. Newkirk, James, and Robert C . Martin. *Extreme Programming in Practice*. Upper Saddle River, NJ: Addison-Wesley, 2001. 中文版《极限编程实战》
3. Highsmith, James A. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* New York, NY: Dorset House, 2000. 中文版《自适应软件开发：复杂系统管理的协作方法》

第 2 章　极限编程实践

“作为开发人员，我们要记住一点，极限编程并非唯一的选择。”
—— 皮特 • 麦克布雷恩（Pete McBreen），《软件工艺》作者

前面一章中，我们简要地概括了敏捷软件开发的内容。但是，它并没有明确地告诉我们去做些什么，除了一些说教性的陈词滥调和目标，它并没有给出实际的指导方法。本章会补充这部分内容。

极限编程实践

极限编程（eXtreme Programming，XP）是最著名的敏捷方法。它由一组简单且互相依赖的实践组成。这些实践结合在一起形成一个整体大于部分的集合。本章中，我们简要探讨一下这套实践集，在后续的章节中，我们会对其中一些实践进行单独研究。

客户团队成员

我们希望客户和开发者在一起紧密地工作，以便彼此知晓对方所面临的问题，并一起解决这些问题。

谁是客户呢？ XP 团队中的客户是指定义产品特性并给这些特性排列优先级的人或者团体。有时候，客户是和开发人员同属一家公司的一组业务分析师或者市场专家。有时候，客户是用户团体委派的用户代表。有时候，客户是实际支付开发费用的人。不过，在 XP 项目中，无论谁是客户，他们都是能够和团队一起工作的团队成员。

最好的情况是客户和开发人员在同一个房间中工作，再次一点的情况是客户和开发人员之间的距离在几十米以内。距离越大，客户就越难成为真正的团队成员。如果客户工作在另外一幢建筑或另外一个省（州），那么他就更难融入团队了。

如果确实无法和客户一起工作，该怎么办呢？我的建议是找一个愿意并能代替真正客户的人来共同工作。

用户故事

为了进行项目计划，必须要知道需求的有关内容，但无需知道得太多。若目的是做计划，只要对需求了解到足够估算的程度就够了。你可能认为，为了对需求进行估算，就必须要了解该需求的所有细节，其实并非如此，你需要知道存在很多细节，也需要知道需求的大致类别，但是不必指明特定的细节。

需求的特定细节很可能随着时间而改变，尤其是当客户看到了集成好的系统时，更会如此。看到新系统上线是关注需求的最佳时刻。因此，在离真正实现需求还很遥远的时候关注该需求的特定细节，很可能会产生浪费。

在 XP 中，我们会和客户反复讨论，获得对需求细节的理解，但是不去捕获那些细节。我们更愿意客户在索引卡片上写下一些我们达成共识的词语，这些只言片语可以帮助我们回忆这次交谈的内容。基本上，在客户写的同时，开发人员会在该卡片上写下对应需求的估算。估算基于我们和客户交谈过程中对细节的理解。

用户故事就是需求澄清过程中的助记词。它是一个计划工具，客户根据它的优先级和估算来安排计划。

短交付周期

XP 项目每两周交付一次可工作的软件。每两周一轮迭代产出一个可以满足干系人部分需求的可工作软件。为了获得干系人的反馈，每轮迭代结束后，系统都要演示给他们看。

迭代计划

一轮迭代一般时长两周。这段期间会产生一个较小的产出物，可能会发布到生产环境。这个产出物是客户在开发者给出的预算范围内挑选出来的一组用户故事。

开发者通过测量他们在前一轮迭代中完成的用户故事给出当前迭代的预算。客户可以挑选任何数量的用户故事放入当前迭代，只要它们的估算不超出预算的范围。

一旦迭代启动，客户就承诺不会在当前迭代中改变用户故事的定义和优先级。在这段时间里，开发者可以自由地把用户故事拆分成任务[①]，并且依据最符合技术和业务意义的优先级开发这些任务。

发布计划

XP 团队通常会创建一次计划来规划随后大约 6 轮迭代的内容，这就是所谓的发布计划。一次发布通常需要 3 个月的工作量。它代表一次较大的交付，通常这次交付会被发布到产品环境中。发布计划由一组排好优先级的用户故事组成，这些用户故事由客户在开发者给出的预算范围内挑选而来。

开发者通过测量他们在前一个发布中完成的用户故事给出当前发布计划的预算。客户可以挑选任意数量的用户故事加入当前发布计划中，只要它们的估算不超过预算。客户也可以决定这次发布计划中需要完成的用户故事的优先级。如果团队成员强烈要求的话，客户可以指明哪些迭代应该完成哪些用户故事，据此规划出发布计划中的前几轮迭代的内容。

发布计划不是一成不变的，客户可以随时改变其中的内容。他们可以取消用户故事、编写新的用户故事或者改变用户故事的优先级。

① 译注：任务拆分方法要符合正交且穷尽，每一个任务完成也必须是独立可验收的。

验收测试

客户通过验收测试捕获用户故事的细节。验收测试的编写要先于或者和用户故事的实现同步进行。它们用一些脚本语言编写，这样就可以自动并重复地运行。与此同时，它们负责验证系统的行为是否符合客户的期望。

编写验收测试所使用的语言和系统的增长和演进保持同步。客户可能会招募新人开发一个简单的脚本系统，或者他们有一个独立的质量保证部门（QA）来负责开发。很多客户会借助 QA 来开发验收测试的工具，并且自己编写验收测试。

一旦验收测试通过，它就会被加入到通过的验收测试的集合里，并且不允许再次失败。这个逐渐增长的验收测试的集合在每次系统构建时都会运行。如果验收测试失败了，那么这次构建也会宣告失败。所以，一旦需求实现，它就永远不会被破坏。系统从一种可工作状态迁移到另一种可工作状态，绝对不允许出现超过几个小时不可工作的状态。

结对编程

所有的产品代码都应该由结对的程序员在一台开发机器上共同完成。[①]结对的两人一个掌控键盘，写代码，另一个人看着对方写，寻找错误和可以提高的地方。两个人交互频繁，全神贯注地投入编写软件的过程中。

两人频繁切换角色。掌控键盘的人可能感到疲劳或遇到困难，此时，他的同伴会接过键盘继续写。在一个小时内，键盘可能在他们之间来回传递好几次。最终的代码是由他们俩人共同设计和实现的，两人功劳均等。

结对组合至少每天要改变一次，以便每个程序员在一天内可以在两个不同的结对组合中工作。在一轮迭代过程中，每个团队成员都应该和其他团队成员结对工作过，并且所有人都应该参与本轮迭代中所涉及的每项工作。

这种做法将极大地促进知识在团队内的传播。当然，专业知识还是必不可少的，那些需要一定专业知识的任务通常需要合适的专家去完成，不过那些专家也几乎会和团队中的所有人结对。这将加快专业知识在团队内的传播。在紧要关头，团队中的其他人就能够代替专家的角色。

[Williams2000], [Cockburn2001] 和 [Nosek] 研究成果表明，结对非但不会降低开发团队的效率，反过来还会大大降低缺陷率。

---

① 我曾经见过这样结对编程的情景，一人掌控键盘，另一人控制鼠标。

测试驱动开发

本书第4章是有关测试的内容，其中详细地讲解了测试驱动开发的方法。下面仅仅提供一个快速的预览。

所有的产品代码都是为让失败的单元测试通过而写的。首先，我们写一个失败的单元测试，因为此时它测试的功能还不存在，然后我们实现功能代码让其通过。

编写测试用例和实现代码之间的更迭速度是很快的，基本上几分钟左右。测试用例和代码共同演进，其中测试用例循序渐进地对代码的实现进行引导。

最终，一个非常完整的测试用例集就和实现代码一起发展起来了。程序员可以使用这些测试用例来检查程序的正确性。如果结对的程序员对代码做了微小改动，那么他们就可以运行测试确保没有破坏任何逻辑。这会非常有利于重构（后续章节会讨论）。

当写出的代码是想要让测试通过时，这样的代码就会被定义为可测试的代码。另外，这样做会大大激发你去解耦每个模块，以便对它们单独进行测试。因此，这样写出来的代码，设计往往是松耦合的。面向对象设计的原则在解耦方面具有巨大的促进作用。

集体所有权

结对编程中每一对成员都有权拉取（check out）和改进任何模块中的代码。没有哪个程序员单独对哪个特定的模块或技术负责。每个人都会参与GUI、中间件和数据库方面的工作。也没有人比其他人在某个模块或技术上更权威。

这并不意味着XP不需要专业知识。如果你专精于GUI领域，那么你最有可能从事GUI方面的任务，但也可能要求你去和别人结对，从事中间件和数据库方面的任务。如果你决定学习另一项专业知识，那么你可以承接相关任务，并和能够传授这方面知识的专家一起工作。你不会被限制在自己的专业领域内。

持续集成

程序员每天会多次提交（check in）代码并进行集成。规则很简单：率先提交的人成功提交到代码库，其他人得合并（merge）本地代码后才能提交。[①]

① 译注：可以参考ThoughtWorks提倡的7步提交法：1. 更新代码；2. 本地编码；3. 本地构建；4. 再次更新代码；5. 本地构建；6. 提交到代码仓库；7. 持续集成服务器上构建。

XP 团队使用非阻塞的源代码控制工具[①]。这意味着程序员可以在任意时间拉取任何模块，而不管其他人是否拉取过这个模块。当程序员完成该模块的修改并提交时，必须把自己的改动和别人先于他提交的改动进行合并。为了避免合并时间过长，团队的成员会非常频繁地提交他们的模块。

结对人员会在一项任务上工作 1~2 个小时。他们编写测试用例和产品代码。在某个适当的间歇点，也许远远在任务完成之前，他们决定把代码提交回去。最重要的是要确保所有的测试都能通过。他们把新代码集成进代码库中。如果需要，他们会对代码进行合并。如有必要，他们还会和先于自己提交的程序员协商。一旦集成进代码仓库，他们就开始从新代码中构建出新系统（详情参见《重构》）。他们运行系统中的每一个测试，包括当前所有运行着的验收测试。如果破坏了原先可以工作的部分，他们就得进行修复。一旦所有的测试都通过，他们就算是完成了此次提交工作。

因而，XP 团队每天都会进行多次系统构建，他们会重新创建整个系统。如果系统的最终结果是一个可以访问的网站，他们就部署该网站，很可能部署到一台测试服务器上。

可持续的开发速度

软件项目不是短跑比赛，而是马拉松长跑比赛。那些跃过起跑线就拼命狂奔的团队在距离终点线很远的地方就会筋疲力尽。为了快速完成开发，团队必须以一种可持续的速度前进。团队必须保持旺盛的精力和高度的警觉，必须有意识地保持稳定、适中的速度。

XP 的规则是不允许团队加班的。不过，在版本发布前一周是该规则唯一的例外，如果发布目标近在眼前并且能够一蹴而就，则允许加班。

开放的工作空间

团队在一个开放的办公空间里一起工作，房间中有一些桌子，每张桌子上摆放了两三台工作机，每台工作机前有两把椅子预备给结对编程的人员，墙壁上挂满了状态图表、任务分解表和 UML 图等。

房间里充满了嗡嗡的交谈声。每对结对人员都坐得近，相互间可以听得到，彼此都能得知对方是否陷入麻烦，也都能了解对方的工作状态。所有人都能够随时随地参与热烈的沟通中。

① 译注：事实上，现在常用的源代码控制工具都是非阻塞的，如 Git、SVN 和 Mercurial 等。

可能有人觉得这种环境会分散人的注意力，很容易担心外界不断的干扰会让人什么事也做不成。但是事实并非如此。而且，密歇根大学的一项研究表明，在“作战室（war room）”里工作，生产率非但不会降低，反而会成倍提升。（http://www.sciencedaily.com/releases/2000/12/001206144705.htm。）

规划游戏

在第 3 章中，我会详细介绍 XP 中规划游戏（planning game）的内容。在这里，先简要描述一下。

计划游戏的本质是划分业务人员和开发人员之间的职责。业务人员（也就是客户）决定特性的重要性（feature 指的是面向最终用户的软件所具备的功能），开发人员决定实现一个特性所花费的代价。

在每次发布和迭代的开始，开发人员会基于最近一次迭代或发布的工作量估算出当前的预算。客户挑选出的用户故事其总花销不超过预算上限。

采用这些简单的原则，经过短周期迭代和频繁的发布，客户和开发人员很快就会适应项目开发的节奏，客户在了解开发人员的速度后，就可以确定项目会持续多长时间以及会花费多少成本。

简单设计

XP 团队总是尽可能把设计做得简单和富有表现力（expressive）。此外，他们仅仅关注本轮迭代中计划完成的用户故事，不会担心将来的事情。相反，他们在一次次迭代中演进系统设计，让当前系统实现的用户故事保持在最优的设计上。

这意味着 XP 团队不大可能从基础设施开始工作，他们不会优先选择数据库或者中间件，而是选择以尽可能简单的方式实现第一批用户故事。只有当某个用户故事迫切依赖基础设施时，才会考虑引入。

下面有三条 XP 咒语（mantra）可以指导开发人员。

考虑可行的最简单的事情

XP 团队总是尽可能寻找针对当前用户故事的最简单的设计。在实现当前用户故事时，如果可以用平面文件[①]，就不去用数据库或者 EJB（企业级 Java Bean）；如果能用简单的套接字连接，就不去用 ORB（对象请求代理）[②]或者 RMI（远程方法调用）。多线程能不用就不用。我们尽量考虑用最简单的方法来实现当前的用户故事。然后，挑选一种我们能实际得到且尽可能简单的解决方案。

你并不需要它

你说得都对，但是我们知道总有一天需要数据库，总有一天需要 ORB，也总有一天得去支持多用户。所以，我们现在就得为这些东西预留位置，是吧？

如果在确切需要基础设施之前拒绝引入会怎么样呢？ XP 团队会对此认真考虑。他们开始时假设不需要那些基础设施。只有当有证据或者至少有十分明显的迹象表明现在引入这些基础设施比继续等待更加划算时，团队才会引入基础设施。

一次且仅有一次

极限编程人员者不能容忍重复代码。无论在哪里发现重复代码，他们都会消除掉。

导致代码重复的因素有很多，最明显的是用鼠标选中一段代码后四处粘贴。当发现那些重复代码时，我们会定义一个函数或基类，用这种方法去消除。有时两个或多个算法非常相似，但是它们之间又有些微妙的差别，我们会把它们变成函数，或者运用模板方法（参见第 14 章的）。无论导致重复的是何种因素，只要发现，必定消除。

消除重复最好的方法就是抽象。毕竟，如果两种事物相似的话，必定可以通过某种抽象统一它们。消除重复的行为会迫使团队提炼出许多的抽象，并进一步减少代码中的耦合。

重构

第 5 章会对重构做详细讨论，下面只是一个简单的介绍。

---

① 译注：平面文件有别于关系型数据库，它指的是没有包含结构化索引和关系的记录文件。它可以是文本也可以是二进制文件，典型的平面文件有 *nix 中的 /etc/passwd 等。

② 译注：对象请求代理是对象之间建立客户端 / 服务端关系的中间件。

代码总是会腐化。随着我们逐渐添加特性，不断处理 bug，代码的结构会慢慢退化。如果置之不理，代码就会变得纠结不清，无法维护。

XP 团队通过频繁地运用重构手法扭转这种局面。重构就是在不改变代码行为的前提下，进行小步改造（transformation）从而改进系统结构的实践方法。每一步改造都是微不足道的，几乎不值一提。但是所有的改造叠加到一起，就会显著地改进系统的设计和架构。

在每次小步改造后，我们运行单元测试来保证没有破坏任何功能。然后继续做下一步改造，如此往复，周而复始，每一步都要运行测试。这样，我们在改善系统设计的同时，始终保持系统可以正常运行。

重构是持续进行的，而不是在项目结束后、版本发布后、迭代结束后，甚至是每天快下班时才去做的。重构是我们每隔一个小时或者半个小时就要去做的事情。重构可以持续地让我们的代码保持尽可能干净、简单和富有表现力。

隐喻

隐喻（metaphore）是所有 XP 实践中最难理解的。极限编程的拥趸本质上都是务实主义者，隐喻这个缺乏具体定义的概念让我们很不舒服。的确，一些 XP 的倡导者经常讨论如何把隐喻从 XP 的实践中移除。然而，在某种意义上，隐喻却是 XP 中最重要的实践之一。

想象一下智力拼图玩具。你怎么知道如何把各个小块拼到一起呢？显然，每一块都和其它块相邻，并且它的形状必须与相邻的块完全吻合。假如你眼神不好但是触觉灵敏，你可以锲而不舍地筛选每个小块，不断调整位置，最终也能拼出整张图来。

不过，还有一种比摸索形状去拼图更为强大的力量，这就是整张拼图的图案。图案是真正的向导。它的力量巨大到如果图案中相邻的两块无法吻合，你就可以断定拼图玩具的制作者把玩具做错了。

这就是隐喻，它是整个系统联结在一起的全景图，它是系统的愿景，它让所有独立模块的位置和形状一目了然。如果模块的形状和整个系统的隐喻不符，那么你就可以断定这个模块是错误的。

通常，隐喻是一个名称系统，名称提供了系统元素的词汇表，它有助于定义元素之间的关系。

举个例子，我曾经做过一个系统，要求以每秒 60 个字符的速率把文本显示到屏幕

上。在这个速率下，铺满屏幕需要花一些时间。所以，我们写了一个程序让它生成文本并填充到一个缓冲区，当缓冲区满了后，我们把程序从内存交换到磁盘上。当缓冲区见底，我们又把程序交换回内存继续运行。

我们把这个系统说成自卸卡车托运垃圾。前面的缓冲区是小型卡车，显示屏是垃圾场，我们的程序是垃圾生产者。这些名称恰如其分，也有助于我们将这个系统当成一个整体来理解。

另一个例子，我做过一个分析网络流量的系统。每隔 30 分钟，它就会轮询数十个网卡，从中抓取监控数据。每个网卡给我们提供一小块由几个独立变量构成的数据，我们把这些小块称为“切片”，这些切片都是原始数据需要进一步分析。分析程序需要“烹饪”这些切片，所以我们把分析程序称为“烤面包机”，把切片中的独立变量称为“面包屑”。总的来说，这个隐喻有用，也有趣。

小结

极限编程是一组构成敏捷开发流程的简单、具体实践的集合。这个流程已经运用到很多团队，也取得了不错的效果。

XP 是一套优良的、通用的软件开发方法论。项目团队可以直接采用，也可以增加一些实践，或者对其中的一些实践进行修改后再采用。

参考文献

1. Dahl, Dijkstra. *Structured Programming*. New York: Hoare, Academic Press, 1972.
2. Conner, Dary1 R. *Leading at the Edge of Chaos*. Wiley, 1998.
3. Cockburn, Alistair. The Methodology Space. Humans and Technology technical report HaT TR.97.03 (dated97.10.03),http://members.aol.com/acockburn/papers/methyspace/methyspace.htm.
4. Beck, Kent. *Extreme Programming Explained: Embracing Change.* Reading, MA: Addison-Wesley, 1999.
5. Newkirk, James, and Robert C. Martin. *Extreme Programming in Practice.* Upper Saddle River,

NJ: Addison-Wesley, 2001.

6. Williams, Laurie, Robert R. Kessler, Ward Cunningham, Ron Jeffries. *Strengthening the Case for Pair Programming.* IEEE Software, July-Aug. 2000.
7. Cockburn, Alistair, and Laurie Williams. *The Costs and Benefits of Pair Programming.* XP2000 Conference in Sardinia, reproduced in *Extreme Programming Examined*, Giancarlo 8. Succi, Michele Marchesi. Addison-Wesley, 2001.
8. Nosek, J. T. *The Case for Collaborative Programming.* Communications of the ACM(1998): 105-108.
9. Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Reading, MA: Addison-Wesley, 1999.