

解析极限编程

——拥抱变化

Extreme
Programming
explained

EMBRACE CHANGE

Kent Beck 著

唐东铭 译

北京 SPIN 审校



Foreword by Erich Gamma

Addison
Wesley

Pearson Education
出版集团

人民邮电出版社

POSTS & TELECOMMUNICATIONS PRESS

解析极限编程——拥抱变化 E~~X~~treme Programming *explained*

软件开发项目可以是充满乐趣、成效卓著的,甚至是惊险的。它们可以在掌控之下稳定地为企业创造价值。

构思和开发极限编程(XP)是为了满足由小型团队实施的软件开发项目的特殊要求,它们面对的是不明确和多变的需求。这一新的轻量级方法论对许多传统的原则提出了挑战。这些原则包括长期为大家所接受的假设:对软件进行更改的成本将必然随时间发展而急剧增加。XP认为:必须努力降低项目的成本并且把节省下来的资金利用好。

XP的基本原理包括:

- 区分根据商业利益做出的决策和由项目负责人做出的决策。
- 总是在编程之前编写单元测试并将所有测试保持在运行状态。
- 集成并测试整个系统(一天几次)。
- 结对生产所有的软件,两个程序员合用一台监视器。
- 从简单的设计开始项目,不断改进系统,增加必要的灵活性并去除不必要的复杂性。
- 将最小的系统迅速投入生产,并朝证明其最有价值的方向发展它。

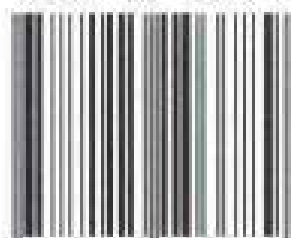
为什么XP会引起这么多争议?一些“权威人士”(sacred cow)不赞同XP的下列主张:

- 不要强迫团队成员专攻某个方向从而成为分析员、结构设计师、程序员、测试员和集成工程师——每一个XP程序员每天都要参与所有这些重要活动。
- 不要进行完全的前置分析和设计——XP项目是从迅速分析整个系统开始的,而XP程序员则在整个开发过程中不断做出分析和设计决策。
- 在开发应用程序的同时开发结构和框架,而不是提前——把提供商业价值放在第一位是驱动XP项目的原动力。
- 不要编写和维护实现文档——XP项目中的沟通是面对面的,或者是通过有效的测试和认真编写的代码来进行的。

你可以喜欢或者厌恶XP,但《解析极限编程——拥抱变化》将会使你用全新的眼光来审视开发软件的方式。

Kent Beck拥有并经营着First Class Software, Inc.,在他的公司里他把注意力集中在他的两大兴趣上:模式和极限编程。他对软件开发的一些先驱模式、CRC卡、HotDraw绘画编辑器框架、xUnit单元测试框架和重新发现“测试先行”的编程做出了贡献。

ISBN 7-115-10378-X



9 787115 103789 >

ISBN7-115-10378-X/TP·2929

定价:29.00元



Pearson Education
出版集团

人民邮电出版社
<http://www.ptpress.com>

471

XP 系列丛书

解析极限编程 ——拥抱变化

Extreme Programming Explained

Kent Beck 著

唐东铭 译

北京 SPIN 审校



A1013242

人民邮电出版社

Addison
Wesley

Pearson Education 出版集团

图书在版编目 (CIP) 数据

解析极限编程: 拥抱变化/ (美) 贝克 (Beck, K.), 著; 唐东铭译.
—北京: 人民邮电出版社, 2002.6
(XP 系列丛书)

ISBN 7-115-10378-X

I. 解... II. ①贝...②唐... III. 程序设计 方法 IV. TP311.11
中国版本图书馆 CIP 数据核字 (2002) 第 043085 号

版 权 声 明

Simplified Chinese edition Copyright © 2000 by PEARSON EDUCATION NORTH ASIA LIMITED and Posts & Telecommunications Press.

Extreme Programming Explained: embrace change

By Kent Beck

Copyright © 2000

All Rights Reserved.

Published by arrangement with Addison-Wesley, Pearson Education, Inc.

This edition is authorized for sale only in People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macau).

本书封面贴有 **Pearson Education** 出版集团激光防伪标签, 无标签者不得销售。

XP 系列丛书

解析极限编程 —— 拥抱变化

◆ 著 Kent Beck

译 唐东铭

审 校 北 京 SPIN

责任编辑 俞 彬

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号

邮编 100061 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

读者热线 010-67180876

北京汉魂图文设计有限公司制作

北京顺义振华印刷厂印刷

新华书店总店北京发行所经销

◆ 开本: 800×1000 1/16

印张: 13.25

字数: 212 千字

2002 年 6 月第 1 版

印数: 1-5 000 册

2002 年 6 月北京第 1 次印刷

著作权合同登记 图字: 01-2001-5026 号

ISBN 7-115-10378-X/TP · 2929

定价: 29.00 元

本书如有印装质量问题, 请与本社联系 电话: (010) 67129223



内容提要

极限编程 (XP) 是一种经历过实践考验的轻量级软件开发方法学, 本书是 XP 宣言, 也是第一本有关 XP 的图书。

本书共分三部分, 第 1 部分包括第 1 章至第 9 章, 通过讨论创建新的软件开发规范中要解决的问题的不同层面来设定极限编程的前提。第 2 部分包括第 10 章至第 18 章, 内容着重于如何将第一部分中的抽象概念转化为具体方法论的实践, 这部分不会确切地说明如何执行这些实践, 而是要讨论它们的大体结构, 同时提供了一套指导性的准则和策略。第 3 部分包括第 19 章至第 27 章, 该部分讨论了如何将上一部分中的策略确切地付诸实践。

本书语言轻松活泼, 实用性与可读性较强, 适合于软件开发人员、软件项目管理人员、软件工程研究人员, 以及所有想要了解 XP 背后思想的各界人士参考。

XP 系列丛书

Kent Beck, 丛书顾问

极限编程, 通常称为 XP, 是一种针对业务和软件开发的规则, 它的作用在于将两者的力量集中在共同的、可以达到的目标上。XP 团队以可持续的步调生产优质软件。选择 XP “书” 中的实践时考虑的是它们对人类创造力的依赖和对人类弱点的包容。

虽然 XP 经常作为一系列实践出现, 但它可不是终点线。只有最终达到了期望的目标以后, 在做 XP 时你才能越做越好。XP 是一个起跑线。它提出这样的问题: “如何能少干活而生产出优秀的软件来?”

最初的答案是如果我们希望软件开发条理分明, 必须准备好完全接受那些要采用的实践。行事中庸就是把现在不解决的问题留到更三心二意的时候去处理。最终你会被无数不彻底的措施所包围, 以至于再也看不出来程序员所创造的价值核心来自编程。

我说“最初的答案……”是因为并不存在任何最终的答案。XP 丛书的作者都是过来人, 他们回过头来讲述自己的故事。这套丛书正是他们沿途立下的路标: “当心暴龙”、“前方 15 公里为风景区”、“雨天路滑”。

不好意思, 我得回去编程了。

丛书书目:

《解析极限编程——拥抱变化》, Kent Beck

《规划极限编程》, Kent Beck 和 Martin Fowler

《极限编程实施》, Ron Jeffries, Ann Anderson 和 Chet Hendrickson

《极限编程研究》, Giancarlo Succi 和 Michele Marchesi

《极限编程实践》, James Newkirk 和 Robert C. Martin

《探索极限编程》, William C. Wake

《应用极限编程——积极求胜》, Ken Auer 和 Roy Miller



中文版丛书序言

发展我国软件产业的重要性已成共识。软件工作者和许多有识之士已决心投入软件产业领域。多年努力，已取得不少进展。但也出现无法回避的进一步发展瓶颈。

对软件开发的认识已有很大进步，从只要简单地照搬规范行事，到关注软件工程、软件开发方法、软件过程改进等方面。人民邮电出版社这次组织翻译的这套 XP 系列图书，全面介绍了当前软件开发方法中一种有影响的流派：XP。其主要特征是要适应环境变化和需求变化，充分发挥开发人员的主动精神。

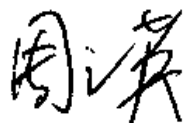
XP 属于轻量开发方法中较有影响的一种方法。轻量开发方法是相对于传统的重量开发方法而言。简单地理解，“量”的轻重是指用于软件过程管理和控制的、除程序量以外的“文档量”的多少。文档从不同角度提供软件开发的可见性，作为测量、预见、管理、决策和控制的客观依据。XP 等轻量开发方法认识到，在当前很多情况下，按传统观念建立的大量文档，一方面需要消耗大量开发资源，同时却已失去帮助“预见、管理、决策和控制的依据”的作用。因此必须重新审视开发环节，去除臃肿累赘，轻装上阵。

当然，轻量带来一些好处的同时，也应看到它的局限性。就是开发中常缺乏一定的预见性，容易造成结构性的质量问题，必须花大力气补救。在学习 XP 过程中，应特别注意和认识 XP 的对策。

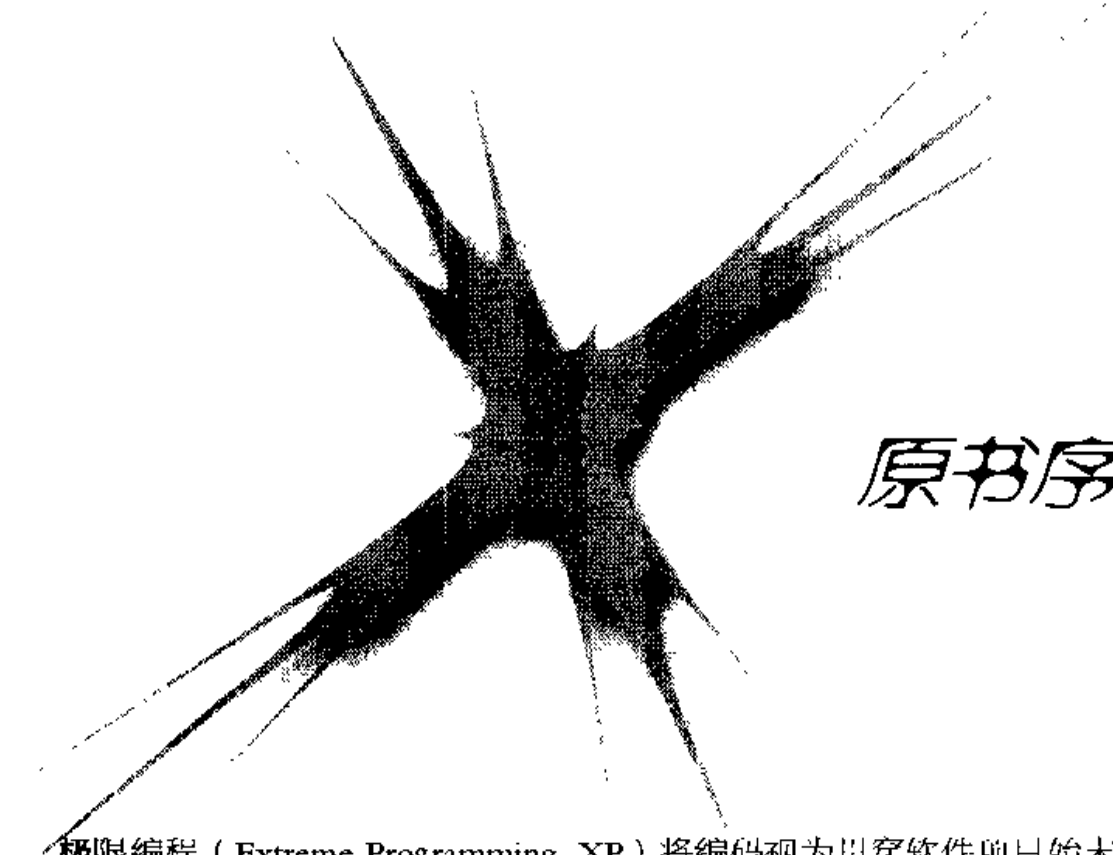
XP 用“沟通、简单、反馈和勇气”来减轻开发压力和包袱；无论是术语命名、专著叙述内容和方式、过程要求，都可以从中感受到轻松愉快和主动奋发的态度和气氛。这是一种帮助理解和更容易激发人的潜力的手段。XP 用自己的实践，在一定范围内成功地打破了软件工程“必须重量”才能成功的传统观念。

网络技术使世界的距离缩短，促进了经济全球化的进程，快速变化和多样性（包括软件开发技术的多样性）成为当今时代的特点，也加剧了学习难度和多样性所导致的无所适从的问题。XP 精神可以启发我们如何学习和对待快速变化、多样的开发技术。成功学习 XP（及其他软件开发方法）的关键，是用“沟通、简单、反馈和勇气”来对待 XP：轻松地来感受 XP 的许多实践思想和我们的开发常识多么接近；根据自己认真实践后，对真实反馈的分析，来决定 XP 对自己的价值；有勇气接受它，或改进它，或打破它。

我很高兴了解到，人民邮电出版社正在引进其他有关软件开发方法学方面的系列著作，其中不乏荣获《Software Development》杂志年度图书大奖的好书，相信这些图书在国内的陆续出版，会对软件开发方法学的研究及教学产生积极的推动作用。



2002 年 5 月于清华园



原书序

极限编程 (Extreme Programming, XP) 将编码视为贯穿软件项目始末的关键活动，这绝对行不通。

是时候花一秒钟来思考一下我自己的开发工作了。我在一种“即时”的软件文化中工作，其特征是紧凑的发行周期和很高的技术风险。不得不让变化成为我的朋友是一种生存的技巧。在地理上身处不同位置的团队内部和团队之间的沟通是用代码来完成的。我们通过阅读代码来了解新的或正在发展中的子系统 API。复杂对象的生命周期和行为是在测试用例中定义的——仍然是用代码。问题报告是用说明问题的测试用例生成的，同样是用代码。最后，我们不断通过重构来优化现有的代码。很显然，我们的开发是以代码为中心的，而且我们能成功地及时交付软件，所以这应该还是行得通的。

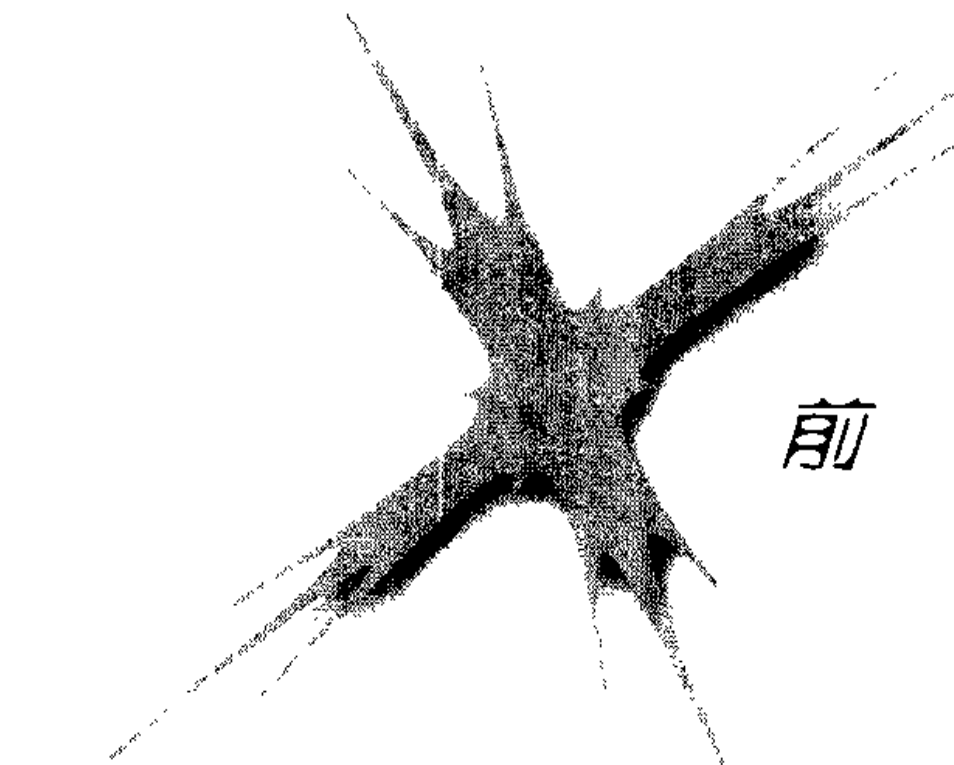
要是得出交付软件需要的就只是大胆编程的结论，可就大错特错了。交付软件是困难的，按时交付优质软件就更加困难。要做到这一点，需要有章法地使用其他最优实践。Kent 在他这本关于 XP 的引人深思的书中正是从这里开始的。

作为 Tektronix 中的一个领导者，Kent 在用 Smalltalk 对复杂工程应用

程序进行循环结对编程（loop pair programming）时认识到了人的潜力。他与 Ward Cunningham 一起推动了这场模式运动，对我的职业生涯产生了巨大影响。XP 所描述的开发方法结合了许多成功的开发人员使用过的实践，而它们原本早已淹没在有关软件方法和过程的书山文海之中。与模式一样，XP 建立在像单元测试、结对编程和重构这样的最优实践之上。在 XP 中，这些实践是结合使用的，这样它们就能优势互补，而且经常是彼此制衡的。问题的核心在于不同实践之间的相互作用，这正是本书的成就所在。唯一的目标是在期限内交付具有正确功能的软件。虽然 OTI 很成功的 Just In Time 软件过程不是纯粹的 XP，但它有许多共同的思路。

我很高兴能与 Kent 进行交流，在一个叫 Junit 的小程序上练习 XP 方法也着实是乐事一桩，他的观点和方法总能引发我对软件开发方法的反思。毫无疑问，XP 对一些传统的主流方法发出了挑战，本书将帮您决定是否接受 XP。

Erich Gamma



前言

这是一本有关极限编程 (Extreme Programming, XP) 的书。XP 是适用于中小型团队在需求不明确或者迅速变化的情况下进行软件开发的轻量级方法学。本书旨在帮助你判断 XP 是否适合你。

对某些人来说, XP 看来就像是普通的常识。那么为何在名称中有“极限”二字呢? 因为 XP 将常识性的原理和实践用到了极致。

- ✧ 如果代码评审是好的, 那么我们会始终评审代码 (结对编程)。
- ✧ 如果测试是好的, 那么所有人都应该始终进行测试 (单元测试), 甚至包括客户 (功能测试)。
- ✧ 如果设计是好的, 那么我们将把它当作每个人的日常事务的一部分 (重构)。
- ✧ 如果简单是好的, 那么我们将始终把系统保持为支持其当前功能的最简单的设计 (可能有效的最简单的东西)。
- ✧ 如果体系结构重要, 那么所有人都将不断进行定义和完善体系结构的工作 (隐喻)。
- ✧ 如果集成测试重要, 那么我们将在一天内多次集成并测试 (持续集成)。

- ◇ 如果迭代周期短些好,那么我们将使迭代时间非常非常短——秒、分或小时,而不是周、月或年(“计划游戏”)。

当我第一次构建出 XP 时,我想到了控制板上的旋钮。

每个旋钮都是一种(经验告诉我的)有效的实践。我将所有的旋钮都调到 10,然后等着看会出现什么情况。我有点惊讶地发现整套实践是稳定、可预测和柔性的。

XP 做出了两套承诺:

- ◇ 对程序员,XP 做出的承诺是他们每天能够处理真正重要的工作。他们不必单独面对令人担忧的状况。他们将能够集中全力来使他们的系统获得成功。他们将做出最适合由他们来做的决策,而且他们将不必做自己做不了的决策。
- ◇ 对于客户和管理人员,XP 的承诺是他们将从每个编程周中获得最多的利益。每隔几周他们都会看到在他们所关心的目标上有具体的进步。他们将能够在开发的中途更改项目的方向而不用承担太高的成本。

总之,XP 承诺降低项目风险,改善对业务变化的反应能力,在系统的整个生命周期内提高生产力,并且为团队的软件开发过程增加乐趣——所有这一切都在同时发生。我说的是真的,不要笑了。现在你必须读本书的其余部分来弄清楚我是不是疯了。

关于本书

这本书讲的是 XP 背后的思想——它的根源、哲学、情节以及神话。它旨在帮助你选择是否在项目中使用 XP 时做出明智的决策。如果你读了本书后正确地决定不把 XP 用于你的项目,或者你正确地决定使用它,我同

样达到了我的目的。本书的第二个目的是帮助那些已经在使用 XP 的读者更好地理解它。

这不是一本关于如何精确地进行“极限编程”的书。你不会在本书中见到大量检查清单或看到许多示例，也不会看到大量编程理论。要获得那些内容，你必须上网，与这里提到的一些教练进行谈话，参阅那些主题性和指导书性质的书籍，或者你也可以做出自己的版本来。

现在，接受 XP 的下一个场合掌握在一群对软件开发的现状不满意的人手中（你可能就是其中一个）。你需要更好的软件开发方法，你需要更好地处理与客户的关系，你需要更快乐、更可靠、更多产的程序员。简而言之，你在寻求巨大的回报，而且为了获得它们你不惧怕尝试新的理念。而且，如果你准备冒险的话，你希望能确信不是因为愚蠢才这么做的。

XP 让你以不同的方式工作，有时 XP 的建议与公认的明智做法完全相反。这里我希望那些选择使用 XP 的人在得到令人信服的理由以后再以不同方式工作，不过要是已经有了理由的话，就不要再犹豫了。我写这本书就是为了向你说明这些理由。

什么是 XP

什么是 XP？XP 是一种轻量、高效、低风险、柔性、可预测、科学而且充满乐趣的软件开发方式。它与其他方法论的不同之处在于：

- ✧ 它的短周期内的早期、具体和持续的反馈。
- ✧ 它递增地进行计划编制，这种方法迅速提供一个总体计划，然后在项目的整个生命周期内不断发展它。
- ✧ 它针对不断变化的业务需求灵活地对功能的实现进行计划的能力。

- ✧ 它依赖于由程序员或客户编写的自动测试来监控开发进度，使得系统得以发展并及早捕获缺陷。
- ✧ 它依赖于口头交流、测试和源代码来沟通系统的结构和意图。
- ✧ 它依赖于在整个系统存在期间一直持续的进化式设计过程。
- ✧ 它依赖于技术水平一般的程序员之间的紧密协作。
- ✧ 它依赖于能同时满足程序员的短期本能和项目的长期利益的实践。

XP 是一种软件开发规则，说它是一种规则是因为有些东西是 XP 中必须做的。你不需要选择是否编写测试——如果你不这样做，那么你就不是极限的：不讨论了。

XP 旨在用于由 2 至 10 名程序员组成的团队开发的项目，这样的项目不能为现有的计算环境所束缚，而且要能够用一天中的少量时间完成合理的测试执行任务。

某些人在第一次接触 XP 时，会感到吃惊或愤怒。不管怎么说，XP 中没有一样概念是新的。大多数概念和编程一样老。某种程度上 XP 是保守的——它的所有技术都经过数十年（对于实现策略）或数百年（对于管理策略）的验证。

XP 的创新之处在于：

- ✧ 把所有这些实践结合在一起。
- ✧ 确保尽可能彻底地执行它们。
- ✧ 确保这些实践能在最大可能程度上互相支持。

够了

在《The Forest People and The Mounted People》中，人类学家 Colin

Turnbull 描绘了两个社会的明显反差。在山区，资源稀缺，人们总是处于饥饿的边缘。由此形成的文明是非常可怕的。一旦小孩有了任何存活下去的机会，母亲们就丢弃他们，让他们与野外的孩子一起流浪。暴力、残忍和背叛是社会的公理。

相反，森林中有丰富的资源。一个人每天只需花费半个小时就能满足自己的基本需要。森林文化是山区文化的反面。成年人共同养育孩子，在孩子们能完全照顾自己之前，他们一直受到养护。如果一个人不小心杀死了另外一个人（蓄谋犯罪的概念还没有），他会被放逐，但也只是走到附近的森林中，并且只是几个月的时间，即使在这种情况下，其他部落的人也会给他食物。

XP 可以作为一个回答“如果你有足够的时间，你将如何编程？”这个问题的试验。但是，你不可能有额外的时间。因为这毕竟是商业，而无疑我们都想成功。但是如果你有足够的时间，你会编写测试，你会在学到一些东西后重新构建系统的结构，你会与程序员同事或客户进行大量讨论。

与那种强加了不可能的期限的、无情而让人难以忍受的苦差事不同，这样的“富足心理”是符合人性的。富足心理是一件好事，它创造出了它特有的效率，正如匮乏心理会造成特有的浪费。

摘要

这本书的写作方式就像是你和我在一起创造一种新的软件开发方法。我们从审视我们关于软件开发的基本假设开始。然后我们创造方法本身。最后我们审视我们所创造的方法的影响——如何采用它、何时不应采用它以及它能为业务创造什么样的机会。

本书分为三个部分。

- ✧ “问题”——从“风险：基本问题”到“回到基本问题”。这些章节提出了“极限编程”试图解决的问题，并提出用来评估解决方案的标准。这部分将给你一个“极限编程”的总体概念。
- ✧ “解决方案”——从“简短概述”到“测试策略”。这些章节将第一部分中的抽象概念转化为具体方法论的实践。这部分不会确切地说明如何执行这些实践，而是要讨论它们的大体结构。每种实践的讨论都将实践与在第一部分中介绍的问题和法则联系起来。
- ✧ 实现 XP——从“采用 XP”到“工作中的 XP”的章节描述了围绕如何实现 XP 的各种主题——如何采用它、极限项目中对各种各样的人的期望、业务方面的人士是如何看待 XP 的。

鸣谢

我是写这本书的第一个人，不是因为这是我的观点，而是因为这是我对这些观点的看法。XP 中的大多数实践和编程一样老。

Ward Cunningham 是你在本书中读到的许多内容的直接来源。从很多意义上看来，过去的 15 年我做的唯一一件事情就是试图向其他人解释他是如何顺应自然地工作的。感谢 Ron Jeffries 的尝试，他使它更加完美。感谢 Martin Fowler 以平易近懂的方式解释它。感谢 Erich Gamma 在观看 Limmat 的天鹅时与我长谈，使我免于思维混乱。而如果没有从我的父亲 Doug Beck 那里学习，并在这些年里一直使用他的编程技术，那么所有这一切将不会发生。

感谢 Chrysler 的 C3 团队坚持跟随我，然后超越我而走向成功。特别感谢我们的经理 Sue Unger 和 Ron Savage 的勇气，给予我们试验的机会。

感谢 Daedalos Consulting 支持本书的写作。

审阅冠军的荣誉属于 Paul Chisholm，他是靠着大量见解独到而通常又

率直尖刻的评论获得桂冠的。如果没有他的反馈，本书不会取得它今天一半的成绩。

我从与所有审阅者的交流中得到了很多乐趣。至少我从他们那儿得到了极大的帮助。我无法完全表达我对他们认真审阅我文章的 1.0 版的谢意，其中一些是用另外一种语言写的。感谢（按我读到他们的评论的随机顺序排列）Greg Hutchinson、Massimo Arnoldi、Dave Cleal、Sames Schuster、Don Wells、Joshua Kerievsky、Thorsten Dittmar、Moritz Reeker、Daniel Gubler、Christoph Henrici、Thomas Zang、Dierk Koenig、Miroslav Novak、Rodney Ryan、Frank Westphal、Paul Trunz、Steve Hayes、Kevin Bradtke、Jeanine De Guzman、Tom cubit、Falk Bruegmann、Hasko Heinecke、Peter Merel、Rob Mee、Pete McBreen、Thomas Ernst、Guido Haechler、Dieter Holz、Martin Knecht、Dierk Konig、Dirk Krampe、Patrick Lisser、Elisabeth Maier、Thomas Mancini、Alexio Moreno、Rolf Pfenninger 和 Matthias Ressel。

Kent Beck



目 录

第 1 部分 问题 1

第 1 章 风险：基本的问题 3

软件开发不能按时交付，也就不能创造价值。这不仅会造成经济损失，而且对当事人本身也有很大的影响。我们需要找到一种新的方法来开发软件。

第 2 章 开发情节 7

日复一日的编程从与客户要求的特性明确相关的任务开始，然后测试、实现、设计，最后到集成。软件开发中每个活动的细节都包含在各个情节中。

第 3 章 软件开发的经济学 11

从经济上考虑，为了使软件开发更有价值，我们需要减少开销，加快收益并增加项目的可能的高效生产期。但最重要的是我们需要增加用于业务决策的选项。

第 4 章 四个变量 15

我们将控制项目中的四个变量——成本、时间、质量和范围。其中，范围的控制最有价值。

第 5 章 变化的成本 21

在某些情况下，更改软件引起的成本以指数方式上升的趋势会随时间的推移而趋于缓和。如果可以使曲线变得平滑，那么以前对有关开发软件的最佳方式的假定将不再成立。

第6章 学会开车 27

我们需要通过做许多小的调整（而不是几次大的调整）来控制软件的开发，这有点像开车。也就是说我们需要反馈来知道我们何时出现了错误，我们需要很多机会来纠正这些错误，而且，我们必须能够以比较合理的成本完成这样的纠正。

第7章 四个准则 29

当我们形成一种风格，能够体现出“沟通”、“简单”、“反馈”和“勇气”这样一整套协调的既能为人们所用、又能满足商业需要的准则的时候，我们就成功了。

第8章 基本原则 37

我们可以从这四个准则衍生出许多基本原则来规范我们的新风格。我们将提出用开发实践来查看它们是否符合这些原则。

第9章 回到基本问题 45

我们希望能够竭尽全力做到稳定、可预测的软件开发。但是我们没有时间去做任何额外的事情。开发软件的四项基本工作是：编码、测试、倾听和设计。

第2部分 解决方案 53

第10章 简短概述 55

我们将依靠简单实践（就是那些几十年前常常被视为不切实际或天真而遭摒弃的实践）之间的协作。

第11章 这如何奏效 67

实践互相支持。一种实践的弱点可以由其他实践的优点来弥补。

第12章 管理策略 77

我们使用商业基本要素全面地管理项目，这些要素包括：分阶段交付，进行迅速和具体的反馈，清晰地阐述系统的业务要求和为特殊任务配备专家。

第13章 设备策略 83

我们将为团队创建一个开放的工作空间，外围是小的私人空间，中间是公共编程区。

第 14 章 拆分业务责任和技术责任..... 87

我们的策略的一个关键点是让技术人员把精力集中在技术问题上，让业务人员把精力集中在业务问题上。项目必须由业务决策来驱动，而技术决策则要给业务决策提供有关成本和风险的信息。

第 15 章 计划策略..... 93

我们制订计划的方法是：迅速制订一个总体计划，然后在越来越短的时间范围内——年、月、周和日——逐步深入地将其完善。我们将迅速并低成本地制订计划，这样当我们必须改动它的时候也不会受到惰性影响。

第 16 章 开发策略..... 105

与管理策略不同，开发策略从根本上背离了传统的观念——我们会认真制订今天的问题的解决方案，并相信我们能够在明天解决明天的问题。

第 17 章 设计策略..... 111

从一个非常简单的起点出发，我们将继续完善系统的设计。我们将去掉任何不能证明是有用的灵活性。

第 18 章 测试策略..... 123

我们总是在编码前编写测试。我们将一直保留这些测试，并频繁地运行全部测试。我们还会根据客户的观点生成测试。

第 3 部分 实现 XP..... 129

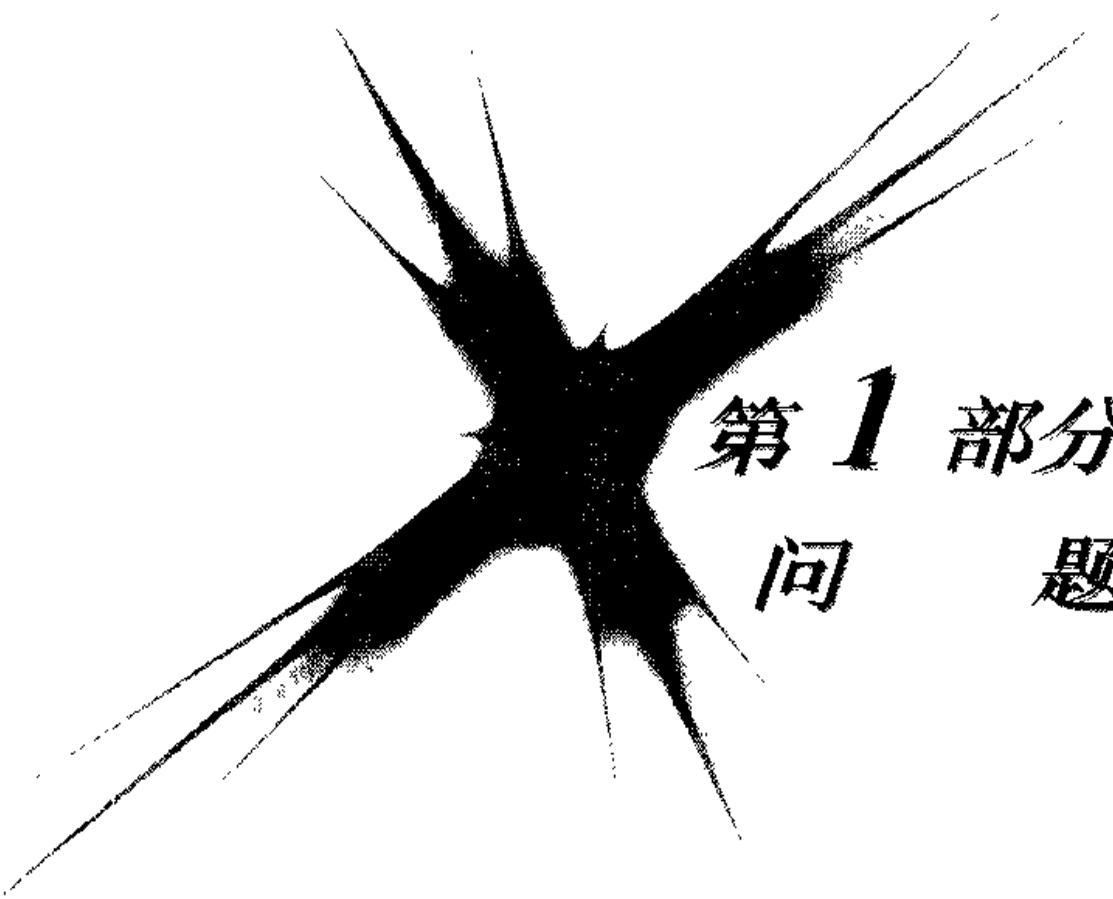
第 19 章 采用 XP..... 131

一次一种实践地采用 XP，始终处理对团队最紧要的问题。一旦这个问题不再是最紧要的，就接着转向下一个问题。

第 20 章 改进 XP..... 133

希望改变其现有文化的项目远比能从头创造新文化的项目更常见。从测试或计划开始，在现有项目上每次进行一点点来逐渐采用 XP。

第 21 章 理想的 XP 项目的生命期	139
理想的 XP 项目要经历一个短暂的初期开发阶段,随后是多年同时进行的生产和优化,最后,当项目失去意义时体面地退休。	
第 22 章 人员的角色	147
我要使极限团队运转起来,就必须有人充当特定的角色——程序员、客户、教练、跟踪者。	
第 23 章 20-80 原则	157
XP 的最大功效只有在采用了所有实践时才能发挥出来。许多实践可以逐个采用,但如果能同时采用它们,它们的效果就会倍增。	
第 24 章 使 XP 难以执行的原因	159
即使蓝领程序员也能够执行单个的实践,但是要把所有的实践组合在一起并保持它们的统一很不容易。使 XP 难以执行的原因主要是人的情感——尤其是恐惧心理。	
第 25 章 什么时候不应使用 XP	163
XP 的确切局限尚不清楚,但是有一些因素会让 XP 难以奏效——团队太大,客户多疑以及技术不能很好地支持更改。	
第 26 章 工作中的 XP	167
XP 可以适应常见形式的合同(虽然稍微有些改动)。特别地,利用计划游戏,固定价格/固定范围的合同会成为固定价格/固定日期/大致固定范围的合同。	
第 27 章 结 论	173
附录 A 参考书目与注释	177
附录 B 词汇表	191



第 1 部分 问 题

本部分通过讨论创建新的软件开发规范中要解决的问题的不同层面来设定极限编程的前提。本部分讨论在选择针对软件开发的不同方面的实践时使用的基本假设——驱动隐喻、四个准则、从这些准则派生出来的原则以及要根据我们的新开发规范组织的活动。



第 1 章

风险：基本的问题

软件开发不能按时交付，也就不能创造价值。这不仅会造成经济损失，而且对当事人本身也有很大的影响。我们需要找到一种新的方法来开发软件。

软件开发的基本问题是其风险。以下是一些关于风险的例子：

- ✧ 进度延迟——交付的日期到了，您却只能通知客户，软件还需再
过 6 个月才能完成。
- ✧ 项目取消——经过多次延迟后，项目尚未投入生产就被取消了。
- ✧ 系统恶化——软件成功地投入了生产，但几年之后，对其进行更
改的成本以及缺陷率大量增加，以至于必须更换系统。
- ✧ 缺陷率——软件投入了生产，但缺陷率太高，以至于没有人使用
它。
- ✧ 业务误解——软件投入了生产，但是没有解决原先提出的业务问
题。
- ✧ 业务变更——软件投入了生产，但所设计的软件要解决的业务问
题在 6 个月前已经被其他更紧迫的业务问题取代。
- ✧ 错误特性多——软件有许多潜在的、非常有趣的特性，所有这些

特性使编程变得非常有趣,但却不能为客户创造任何实际的价值。

- ◆ 人员调整——两年后,这个项目的优秀程序员开始厌恶而离开了。

在本书中,您会读到关于极限编程(Extreme Programming, XP)的知识,极限编程是一种处理软件开发过程中各个级别上风险的软件开发方法。XP的生产效率很高,它可以产生高质量的软件,而且在执行过程中有很多乐趣。

XP如何处理上述列出的风险呢?

- ◆ 进度延迟——XP要求发行周期较短,最多为几个月,因此任何延迟的范围就受到限制。在一个发行周期内,XP对客户所要求的特性进行一周到四周的迭代操作,以获得对进度的详细反馈。在一次迭代中,XP用一到三天的任务期进行计划,因此甚至在一次迭代中团队也能解决问题。最后,XP要求首先实现最高优先级的特性,因此错过该发行周期的任何特性将是不太重要的。
- ◆ 项目取消——XP让客户选择具有最大业务意义的最小版本,从而在投入生产前减少发生错误的机率,同时软件的价值也得到最大化。
- ◆ 系统恶化——XP创建并维护一整套测试程序,每次变化(一天几次)发生后都要运行或者重新运行这些程序,以确保质量基准。XP总是使系统保持在最佳状态,不允许累积错误。
- ◆ 缺陷率——XP进行测试时,既遵从程序员按逐条功能编写测试程序的观点,又遵从客户按逐个程序特性编写测试程序的观点。
- ◆ 业务误解——XP要求客户成为整个团队中的一部分。在开发过程中,项目的说明书不断得到改进,因此客户和团队所学到的东西

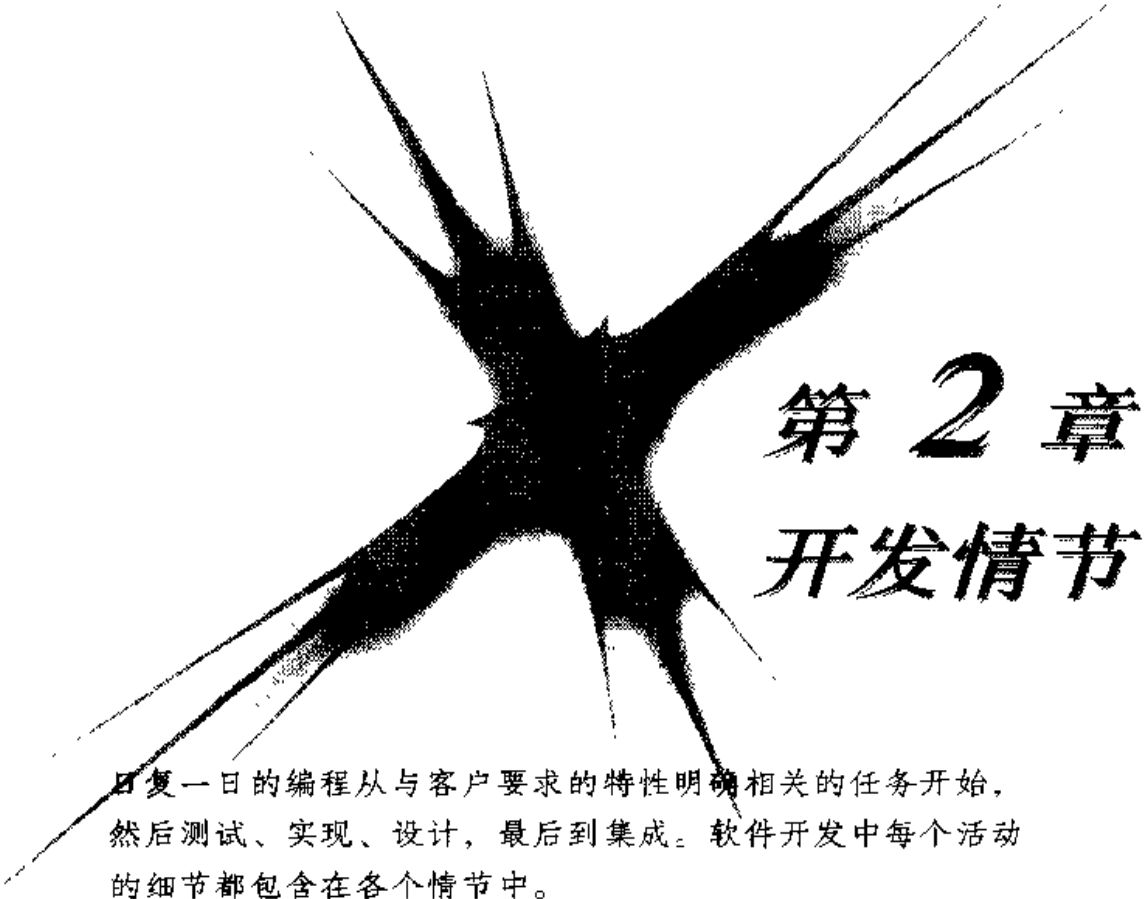
会在软件中反映出来。

- ◇ 业务变更——XP 缩短了版本周期，因此某个版本的开发过程中的变化就更少。在一个发行周期中，欢迎客户用新功能取代仍未完成的功能。团队甚至不会注意到正在处理的是新发现的功能还是几年前就定义的功能。
- ◇ 错误特性多——XP 强调只专注于具有最高优先级的任务。
- ◇ 人员调整——XP 要求程序员承担估算和完成自己工作的责任，并将实际所花费时间反馈给他们，以改进他们的估算，并且尊重他们的估算。谁能够做出和改变估算的规则是非常清楚的。这样，就可能更少地因为要求程序员做明显不可能完成的工作而使之感到沮丧。XP 还鼓励团队成员间的相互接触，以减少常常由于对工作不满意而产生的孤独感。最后，XP 整合了一个明确的人员调整模型。鼓励新的团队成员逐步承担越来越多的责任。在这个过程中，其他新成员和现有程序员都会帮助他们。

我们的使命

如果我们将项目风险作为要解决的问题，那么我们从什么地方着手寻找解决方案呢？我们需要做的是发明一种处理这些风险的软件开发方式，尽可能地使程序员、管理人员和客户清楚这种方法。并且提出使它适用于实际情况的指导方针（即，就哪些是固定的而哪些是可变的进行沟通）。

这就是本书第一和第二部分所包含的内容。我们会逐步探究创建一种新的开发方式或方法的各种问题，然后再解决这些问题。基于一套基本的假设，我们可以得出方案来指导软件开发过程中出现的各种活动——计划、测试、开发、设计和部署。



第 2 章 开发情节

日复一日的编程从与客户要求的特性明确相关的任务开始，然后测试、实现、设计，最后到集成。软件开发中每个活动的细节都包含在各个情节中。

但在开始之前先粗略看一下我们前进的方向。本章讲述有关 XP 核心的故事——开发情节。程序员在这个情节中实现一个工程任务（进度中的最小单元），并把它集成到系统的其他部分中。

我看着我的那堆任务卡。最上面的一张写着：“Export Quarter-to-date Withholding.” 在今天上午的碰头会上，我记得你说过你已经完成了“quarter-to-date”的计算。我问你（我的假定伙伴）是否有时间来帮助输出。你说：“当然。”这个规则就是：如果有人需要你帮忙，你必须说：“好的。”我们刚刚结成了结对编程伙伴。

我们花了几分钟来讨论你昨天所做的工作。你讲到你昨天所添加的程序，测试结果如何，或许还稍稍提到你昨天注意到当你把监视器移退一英尺后，结对编程会工作得更好。

你问：“这项任务的测试用例是什么？”

我说：“当我们运行输出工作站的时候，输出记录中的值应该和程序中

的值相匹配。”

“需要具有哪些字段？”你问。

“我不知道。我们去问埃迪吧。”

我们打断了埃迪 30 秒。他解释了他所知道的有关 “quarter-to-date” 的五个字段。

我们去看了一些现有的输出测试用例的结构，发现了一个差不多正是我们所需要的实例。通过抽象一个超类，我们可以轻易地实现我们的测试用例。我们对它进行了重构，运行了现有的测试。它们都可以运行。

我们发现其他几个输出测试用例可以利用我们刚刚创建的超类。我们想要看看这项任务的一些结果，所以我们就在计划卡上写下 “Retrofit AbstractExportTest”。

现在我们编写测试用例。因为我们刚刚做了测试用例的超类，编写新的测试用例很容易。花了几分钟我们就写完了。在进行到大约一半的时候，我说：“我甚至能够想到我们怎样来实现它。我们能……”

你打断了我：“让我们先完成测试用例吧。” 在我们编写测试用例的时候，想到了三种方法。你把它们写在计划卡中。

我们完成了测试用例并且运行它。它失败了。当然，我们还没有实现任何东西。“等会儿，”你说。“昨天上午，拉尔夫和我在编写一个计算器程序。我们写了五个认为会出错的测试用例。除了一个用例外都运行出错了。”

我们对这个测试用例进行了调试，查看了必须进行计算的对象。

我编写了代码（或者你来写，谁的思路最清晰，谁来写）。在实现的时候，我们意识到应该再多写几个测试用例。 我们把它们放在了计划卡中。测试用例可以运行了。

我们逐个地转到各个测试用例。我实现它们。你发现代码可以写得更简单些，试图向我解释怎样简化代码。我一边听着你说话一边实现失去了信心，所以我把键盘推给了你。你重构了这些代码。运行测试用例，它们

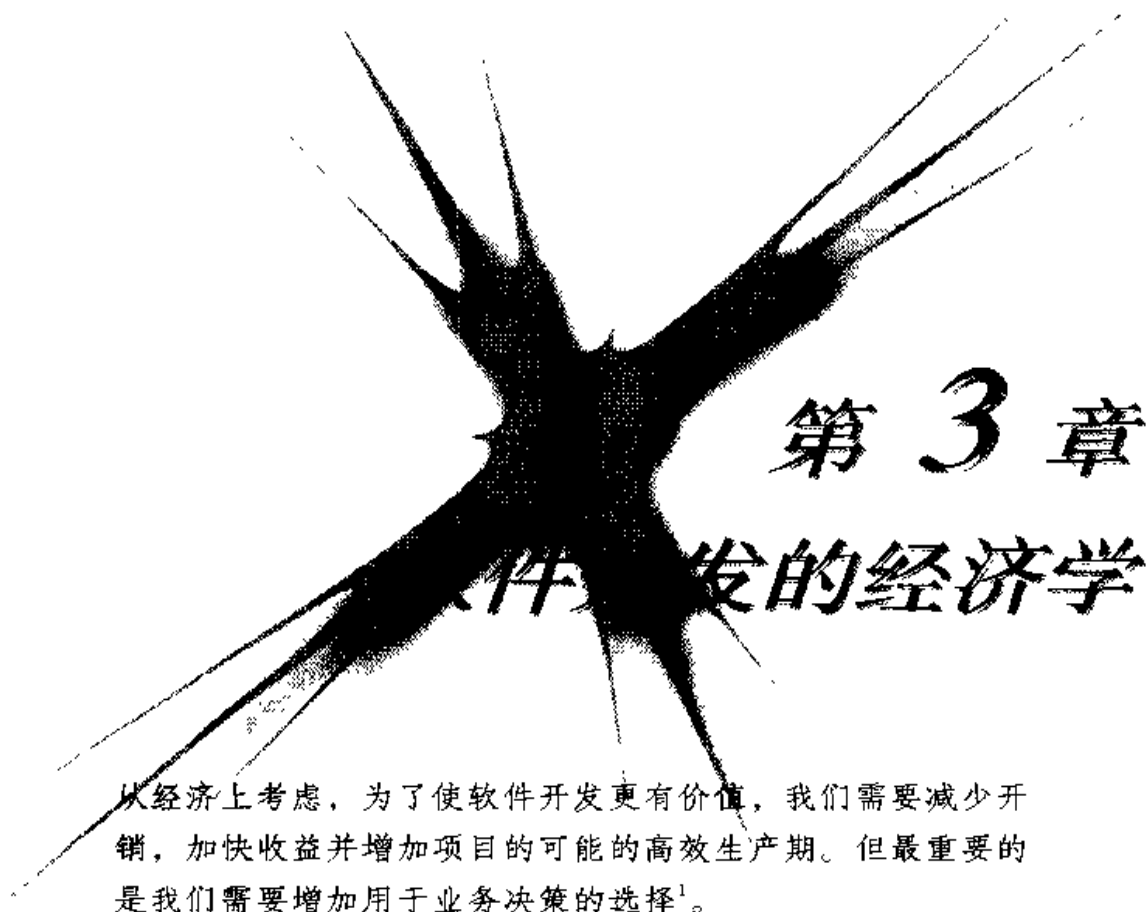
通过了。你实现了剩下的几个测试用例。

过了一会儿，我们看了看计划卡，上面只有一项：重构其他测试用例。事情进行得很顺利，于是我们继续工作，把它们进行重构，以确认当我们完成时它们可以运行。

现在计划列表已经空了。我们注意到用于集成的机器正空着。我们调用了最新版本，加载我们所做的变动。然后我们运行了所有的测试用例，包括我们新写的以及其他人以前写的。其中有一个失败了。“真奇怪。我已经差不多有一个月没有在集成的过程中碰到失败的测试用例了。”你说。没有问题。我们调试测试用例并改正了代码。然后又运行了整个系统。这一次它通过了。我们发布了我们的代码。

这就是 XP 的完整开发过程。请注意：

- ✧ 两个程序员一起编程。
- ✧ 测试驱动开发。首先测试，然后编码。除非所有的测试都能运行，否则你就没有完成。当所有的测试都能够运行，而且你想不出其他的可能会失败的测试的时候，你就完成了增加功能的工作。
- ✧ 结对程序员不仅让测试用例运行，还改进系统的设计。任何特定区域都不限制变化。结对程序员提高了系统的分析、设计、实现和测试的作用。他们在系统需要的任何地方提供这种作用。
- ✧ 开发之后紧接着就是集成，包括集成测试。



第 3 章

软件开发的经济学

从经济上考虑，为了使软件开发更有价值，我们需要减少开销，加快收益并增加项目的可能的高效生产期。但最重要的是我们需要增加用于业务决策的选择¹。

通过合计出入项目中的现金流，我们可以简单地分析一下是什么使软件项目有价值。将利率的影响考虑在内，我们能够计算出现金流中的净现值。然后，用扣除利息的现金流乘以项目最后能够成功支付或赚得这些现金流的概率，我们就可以进一步改善这种分析。

使用下面三个因子：

- ◇ 出入的现金流。
- ◇ 利率。
- ◇ 项目失败率。

我们可以创建一个能够使项目的经济价值得到最大化的策略。可以通

1. 感谢 John Favaro 用选择定价对 XP 进行的分析。

过如下方式做到这一点：

- ✧ 减少开销。这一点很难，因为每个人开始的时候具有的工具和技能差不多都相同。
- ✧ 增加收益。只要具有极其优秀的市场营销机构，这才是可能的。谢天谢地，本书不会涉及这个话题。
- ✧ 推迟开销并提早收益。这样我们就可以为开销支付更少的利息，而利用收入的钱赚得更多的利息。
- ✧ 提高项目存活的几率。这样会提高在项目的后期获较大盈利的可能性。

3.1 选择

另外有一种看待软件项目经济学的方式——把它当作一系列选择。软件项目管理可以看作具有四种选择：

- ✧ 放弃的选择——可以把某些东西从项目中去掉（甚至取消整个项目）。即使不能按照项目最初设想的形式实际交付，你由于放弃而获得的利益越多就越好。
- ✧ 转换的选择——可以改变项目的方向。如果在项目完成的中途客户可以改变需求，项目管理策略就更有价值。需求的变化能够越大越频繁，就越好。
- ✧ 延期的选择——可以等到形势明了后再投资。如果您可以等待时机，但又不会完全失去投资机会，这时候项目管理策略就更有价值。延期越长，能够延期的钱越多，就越好。
- ✧ 增长的选择——如果某一市场看来马上要腾飞，你可以快速增长以从中获利，在项目有更多投资的时候能够适度成比例地不断扩

大生产规模，项目管理策略就更有价值。项目增长得越快，增长的时间越长，就越好。

选择价值的计算两分是艺术，五分是数学，一分是有效的老式 Kentucky 偏差修正。

其中涉及到五个因子：

- ◇ 获得选择所需的投资量。
- ◇ 如果要实行选择，那么最终赢得收获所需的代价。
- ◇ 收获的现值。
- ◇ 实行这些选择的时间量。
- ◇ 收获的最终价值的不确定性。

其中，选择的价值通常是由最后一个因子（即不确定性）来支配。由此，我们可以做一个具体的预测。假定我们创建了一个项目管理策略，它通过提供以下内容来最大化分析为选择的项目的价值：

- ◇ 关于进度的准确和频繁的反馈。
- ◇ 许多对需求做出显著更改的机会。
- ◇ 更小的初期投资。
- ◇ 进展更快的机会。

不确定性越大，策略的价值越高。无论不确定性是来自技术风险、变化的业务环境还是来自飞速变化的需求，结论都一样。（这为“什么时候应该使用 XP？”的问题提供了一个理论上的答案：在需求比较模糊或需求不断变化的时候使用 XP。）

3.2 示例

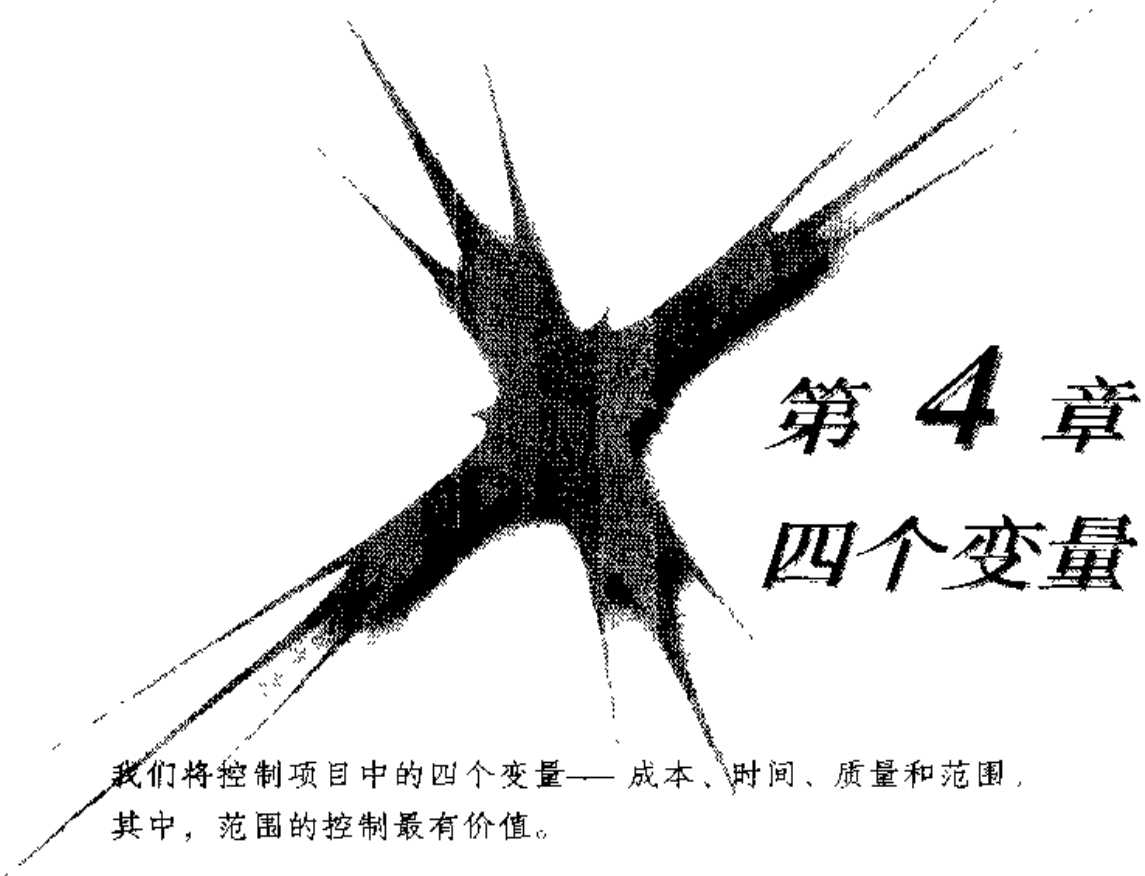
假定您正在愉快地编程,这时候您发现可以花费 10 美元的成本添加一个功能。估计该功能的回报(它的净现值)大约在 15 美元左右。因此,添加该功能的净现值就是 5 美元。

假定您打心眼里明白这个新功能究竟值多少钱根本说不清楚——15 美元只是您的猜测,而不是确切地知道对于客户而言它就值那么多。实际上,您估摸着它对客户的价值与您的估计相差可能会高达 100%。进一步假定(请参见第五章“变化的成本”)距今一年后添加该功能仍然需要 10 美元的成本。

那么,只是等待而不马上实现该功能的策略的价值有多少呢?以大约 5% 的正常利率计算,选择理论计算器给出了 7.87 美元的价值。

选择等待比现在投资添加该功能($NPV = 5$ 美元)更划算,为什么呢?由于不确定性很大,该功能对客户价值无疑可能会高得多,因此在这种情况下,等待要比现在实现它更好。或者,它可能一文不值,那么在这种情况下,您就避免了无谓的劳作。

按贸易的行话来讲就是选择“消除了不利的风险。”



第 4 章

四个变量

我们将控制项目中的四个变量——成本、时间、质量和范围，其中，范围的控制最有价值。

这里有一个从控制变量系统的角度建立的软件开发模型。这个模型中，软件开发过程中有四个变量：

- ◇ 成本。
- ◇ 时间。
- ◇ 质量。
- ◇ 范围。

在这个模型中，软件开发的游戏规则是让外力（客户、管理人员）确定任意三个变量的值，而开发团队确定第四个变量的结果值。

有些管理人员和客户认为他们可以确定所有四个变量的值。“你要用这个小组在下个月一号之前达到所有这些需求。质量是这儿的首要工作，所以它应该达到通常的标准。”这种情况下，质量总是被抛诸脑后（虽然一般说来这确实是达到了通常的标准），这是因为在太大的压力下，没有人能做好工作。时间上也可能会失控，最后，你得到的是蹩脚的软件。

解决办法是让这四个变量变得可见。如果每个人——程序员、客户和管理人员——都能看到所有这四个变量，他们就能有意识地选择控制哪些变量。如果不喜欢第四个变量得出的结果，他们可以改变输入，或者选择控制另外三个变量。

4.1 变量之间的相互作用

成本——钱多一点可以促进工作顺利进行，但是太短时间内投入太多的钱则会产生更多的无法解决的问题。另一方面，拨给项目的钱太少，将不能解决客户的业务问题。

时间——更长的交付时间能够帮助提高质量和扩大范围。由于来自生产系统的反馈的质量要比任何其他种类的反馈高得多，因此，给一个项目太多时间有损质量。划给项目的时间太少也会使质量受损，而且会有类似程度的对范围、时间和成本的损害。

质量——作为一个控制变量，质量极其重要。如果有意识地牺牲质量，你可能在短期（几天或几周）内获利，但是耗费的成本——人力、商务和技术是巨大的。

范围——更小的范围可能会有助于提高质量（只要能够解决客户的业务问题）。它也使你能够更快或更经济地交付产品。

在这四个变量之间没有一个简单的关系。例如，你不能仅仅通过花更多的钱来更快地完成软件。就像谚语所说：“九个女人在一起也不能在一个月里生出一个孩子。”（我从一些管理人员那里听到的是，十八个女人在一起也不能在一个月里生出一个孩子。）

在许多方面，成本是受约束最多的变量。你不能只靠花钱就得到质量、范围或者短的发行周期。实际上，在项目开始的时候，你根本花不了多少钱。投资只能从小开始，随着时间的推进而增长。过些时间，你就可以有

成效地花费越来越多的钱。

我曾经听一个客户说：“我们已经承诺完成所有这些功能。为了做到这一点，我们必须有 40 个程序员。”

我说：“你不能在第一天就用 40 个程序员。你们必须从一个人的团队开始，然后增加到两个，然后是 4 个。两年后你就可以拥有 40 个程序员，但不是现在。”

他们说：“你不懂。我们必须有 40 个程序员。”我说：“你不需要 40 个程序员。”他们说：“我们必须这样。”

他们不需要。我的意思是说，他们确实这么去做了。他们雇用了 40 个程序员。事情进展得并不顺利。原来的程序员离开了，他们接着又雇用了 40 个。4 年后，他们才刚刚开始能够提供商务价值，每次做一个小的子项目，起先他们差点被取消了订单。

所有这些对成本的约束简直要把管理人员逼疯。尤其是在他们死盯住年度预算过程时，他们是如此习惯于通过成本来驱动一切，以至于他们会因为无视成本控制的约束而犯大错。

与成本有关的其他问题是更高的成本经常被用来满足不相关的目的，比如说地位或声誉。“当然，我拥有一个 150 人的项目（哼哼）。”这会由于管理人员试图让人觉得他很了不起而导致项目失败。毕竟，为同一个项目配备 10 个程序员而以一半的时间完成，这个地位又是多高呢？

另一方面，成本同其他变量密切相关。在有意义的投资范围之内，你可以通过花更多的钱来扩大范围，或者可以更加周密地行动以提高质量，或者你可以（在一定程度上）缩减投入市场的时间。

花钱也可以减小资源冲突——更快的计算机、更多的技术专家和更好的办公室。

通过控制时间来控制项目所受的约束通常来自外部——2000 年问题就是最近的例子。年末；季度开始前；旧系统预定关闭的时间；一场大的

商业展览——这些都是外部时间约束的例子。因此，时间变量经常是在项目管理人员的控制之外，而掌握在客户手中。

质量是另一个奇特的变量。经常出现这样的情况，坚持更好的质量使你能够更快地完成项目，或者在给定的时间内完成更多的工作。当我开始编写单元测试的时候遇到了这种情况（在第二章，开发情节中有描述）。一旦完成了测试，我就对自己的代码变得非常自信，没有了压力，写得就更快了。我能轻松地清理系统，使进一步的开发更容易。我见过在团队中出现相同的状况。当他们开始测试，或者当在编码标准上达成了一致时，他们就能够更快地进行工作。

在内部和外部质量之间存在一种奇特的关系。外部质量是由客户衡量的质量，而内部质量是由程序员衡量。暂时牺牲内部质量以缩减投入市场的时间，然后还希望外部质量不会损害太多，这真是一个诱人的短期投机。在几周或几个月里，你可能没有把事情搞得一团糟。然而，最终内部质量会惩罚你，使你的软件维护费用昂贵，或者外部质量不能达到有竞争力的水平。

另一方面，你可以不时地放松对质量的约束以使工作完成得更快。有一次，我在开发一个系统，以替换一个传统的 COBOL 系统。我们的质量约束是要精确地产生旧系统所产生的答案。当我们越来越接近发行日期的时候，有一点变得明显了，那就是我们可以复制出旧系统的所有错误，但那只能通过将交货日期推迟许多才可以做到。我们到了客户那儿，向他们说明我们的答案更加正确，并且向他们提供了按时交货的选择——如果他们要改为相信我们的答案的话。

存在一个由质量而来的人为影响。每个人想干好工作，而且，如果感觉自己做得不错，他们就会干得更好。如果你有意识地降低质量，你的团队开始时或许会进行得更快，但是很快由制造废品而产生的消沉情绪将压倒性地抵消你通过不测试、不评审或者不坚持标准而获得的暂时收益。

4.2 专注于范围

许多人知道把成本、质量和时间作为控制变量，却不了解第四个变量。对于软件开发来说，范围是应该注意的最重要的变量。无论是程序员还是业务人员，都仅对开发中的软件的价值所在有一个模糊概念。在项目管理中，最有力的决定之一是消除范围。如果积极主动地管理范围，你就可以向管理人员和客户提供对成本、质量和时间的控制。

范围的一个重要特点就是它是一个变化很大的变量。几十年来，程序员不断在抱怨：“客户不能告诉我们他们想要什么。而当我们给出他们所说的需要的东西时，他们又不喜欢它。”这在软件开发中是一个不争的事实。在开始的时候，需求从来都是不清楚的。客户从来不能确切地告诉你他们需要什么。

软件的开发会改变软件本身的需求。一旦客户见到第一个版本，他们就明白了在第二个版本中他们需要的东西……或者他们在第一个版本中实际上想要什么。这是宝贵的知识，因为它不可能根据猜想而得到。它是只能从经验获得的知识。但是客户无法独自做到这一点。他们需要会编程的人来共事而不只是做向导而已。

要是我们把需求的这种“柔性”看作一个机会，而不是一个问题，会怎么样呢？接下来，我们就可以选择将范围当作四个变量中最容易控制的。由于它是如此具有柔性，我们可以对它进行塑造——这样来一下，那样再来一下。如果到发行日期的时间开始变得紧张时，那么总会有一些东西可以延期到下一个版本。通过不去试图完成太多的功能，我们就可以保持能力按时实现满足所需质量的产品。

如果我们根据这一模型创建一种开发的规则，就可以先使软件的日期、质量和成本固定下来，然后看一下这三个变量所揭示的范围。然后，在开发进行的时候，我们可以不断地调整范围以适应遇到的情况。

这必须是一个能够很容易地包容变化的过程，因为项目会经常改变方向。你不想在最后没人使用的软件上花费太多。你不想建造一条因为你改变方向了而从来不会在上面行驶的路。而且，你必须有一个过程，这个过程在系统的生存期内将变化的成本保持在合理范围内。

如果在每个发行周期结束时，你都丢弃了重要的功能，那么客户很快会变得烦躁。为了避免这种情况，XP 使用了两个策略：

1. 进行大量估算和反馈实际结果的工作。更好的估算能减小你不得不放弃功能的几率。
2. 首先实现客户最重要的需求，这样如果不得不放弃别的功能，那么这个功能也不会比已经在系统中运行的功能更重要。

第 5 章

变化的成本

在某些情况下，更改软件引起的成本以指数方式上升的趋势会随时间的推移而趋于缓和。如果可以使曲线变得平滑，那么以前对有关开发软件的最佳方式的假定将不再成立。

软件工程中一个普遍的假定是更改程序的成本会随时间的推移而以指数方式上升。我还记得在大学三年级时，坐在油毡地板的大教室里看教授在黑板上绘制图 1 中的曲线。

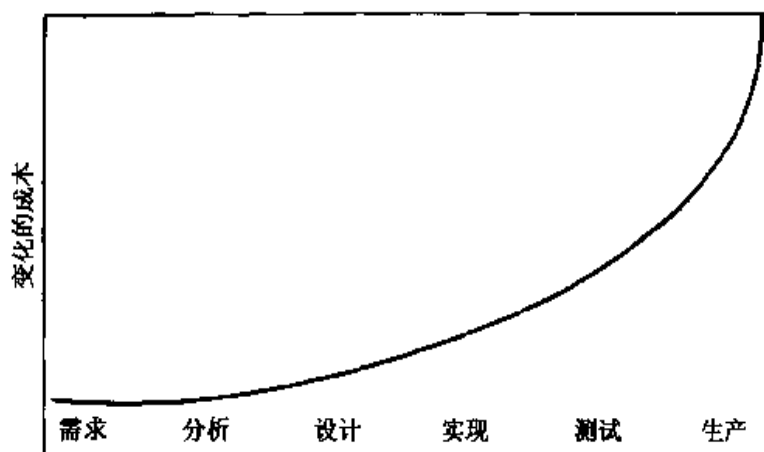


图 1 变化的成本随时间的推移而以指数方式上升

“解决一个软件中的问题的成本随时间的推移而以指数方式上升。如果在需求分析阶段发现问题，可能只花一美元就可以解决，但如果是在生产

中发现的，就可能要花数千美元。”

那时我就决定，我决不让问题留到生产阶段。决不，先生，我要尽早地找到问题。我要事先弄清每一个可能的问题。我会反复地评审和查对代码。我决不会浪费老板 10 万美元。

问题是，这条曲线不再有效了。或者，随着技术和编程实践的结合，有可能会遇到与此完全相反的曲线。可能会遇到像下面的问题，这个问题是我最近在一个人寿保险合同管理系统中遇到的：

1700——我发现，到现在为止，我们的系统的一个杰出功能，即——单笔交易可以从若干帐户借记和对若干帐户贷记——竟然没有使用。每笔交易都是从一个帐户转到另一个帐户。是否有可能如图 2 中所示那样简化系统呢？

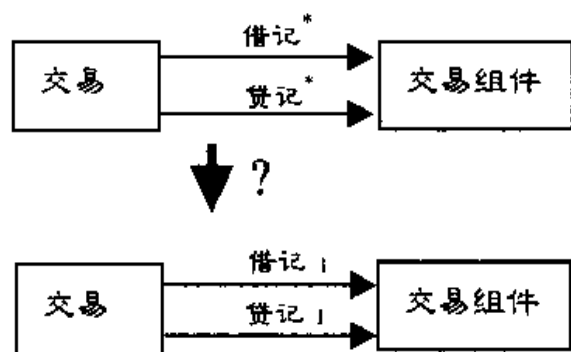


图 2 是否可为借记和贷记分别提供一个组件

1702——我叫 Massimo 与我一起分析这种情况。我们编写了一个查询。系统的 30 万笔交易中，每笔业务都只有一个借记帐户和一个贷记帐户。

1705——如果我们要改正这个错误，应该怎么做呢？我们要改变交易的接口并改变实现。我们编写了所需的四个方法然后开始测试。

1715——测试(超过 1000 个单元测试和 6 个功能测试)仍能 100%运行。我们想不出更改不起作用的原因。我们开始处理数据库迁移代码。

1720——完成了每天晚上的批处理工作并备份了数据库。我们安装了新版本的代码并运行迁移。

1730——我们正常地进行了几次测试。任何我们能够想起来的东西都可以正常工作。直到我们想不出还有什么别的可做，就各自回家了。

第二天——错误日志是很清楚的。用户没有任何抱怨。所做的更改已经能够正常工作了。

在之后的几个星期里，我们发现了一系列由于采用了新的结构而能进行的进一步简化，这使我们能够将全新的功能应用到系统的记帐部分，同时使系统变得更简单、更清楚，并降低了冗余度。

近几十年，软件开发群体花费了大量资源试图减少变化的成本，这些变化包括更好的语言、更好的数据库技术、更好的编程实践、更好的环境与工具和新的表示法。

如果所有的投资都得到了回报,我们该怎么办？如果所有对语言、数据库和任何其他内容的工作事实上都取得了一定成果,会怎么样？如果变化的成本不是随着时间的推移以指数方式上升,而是以慢得多的速度上升,最终成为渐近线,将会怎么样？如果未来的软件工程教授在黑板上绘制出的是图 3，又会如何？

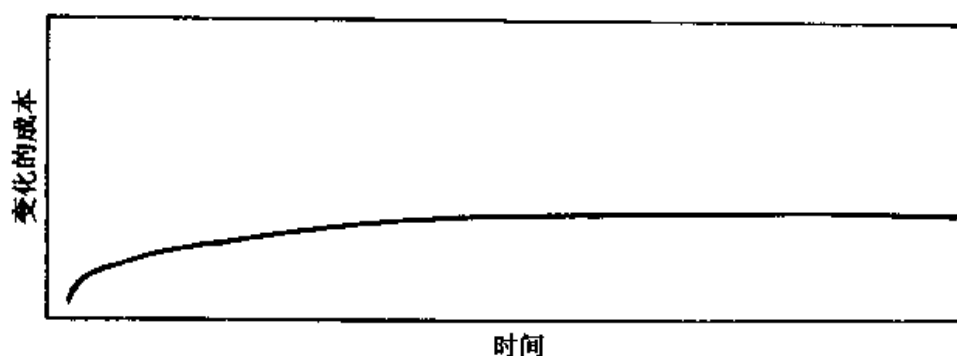


图 3 变化的成本可能不会随时间的推移急剧上升

这是 XP 的前提之一。它是 XP 的技术前提。如果变化的成本随时间推移缓慢上升¹，那么你工作的方式将会与假定成本以指数方式上升时完全不同。你会在过程中尽可能迟地做出重大决定，从而推迟支付做决定的成本并使得做出正确决定的可能性最大。你将仅实现你必须实现的部分，

而寄希望于预期的未来需求不会变成现实。你仅当元素简化了现有代码或者使编写后面的代码更简单时，才将它们引入设计。

如果平滑的变化成本曲线使 XP 成为可能，那么急剧上升的变化成本曲线将使 XP 变得不可能。如果变化极其昂贵，你就会因为没有经过缜密分析就提前支付了费用而气得发疯。但是，如果变化的花费一直很少，那么附加值和通过早期的具体反馈而减少的风险将在价值上超过早期变化所引起的附加成本。

保持变化成本低廉不是变魔术。有很多技术和实践可以保持软件的柔性。

在技术方面，对象是一项关键的技术。发送消息是低成本地创造许多变化的机会的一种有效方法。每个消息都成为一个将来无需更改现有代码就可以进行修改的可能点。

对象数据库将这种柔性传递到了永久存储的领域中。使用对象数据库，就有可能方便地将一种格式的对象迁移为另一种格式的对象，这是因为代码是附加在数据上的，而不是像早期数据库技术中那样是分开的。即使你无法找到迁移对象的方法，也可以使两种实现并存。

这并不是说，你必须使用对象才可以得到柔性。通过观察我父亲如何用汇编语言编写实时进程控制代码，我学到了 XP 的基本原理。他开发了一种样式，使他可以不断改善他程序的设计。但是，我的经验是，不使用对象时变化成本的上升比使用对象时更急剧。

这并不是说有了对象就足够了。我曾经见到过（说实话，可能也编写过）大量没人愿意碰的面向对象的代码。

有几个因素能决定代码是否易于修改（即使是在生产数年之后）：

- ◆ 简单的设计，没有多余的设计元素——没有现在还用不到但预计将来会使用的构想。
- ◆ 自动测试，这样我们就有把握能知道是否意外地更改了系统的现

有行为。

- ✧ 修改设计方面的大量实践，这样，当要更改系统时，我们就不会太害怕去尝试。

有了这些元素——简单设计、测试和不断改良设计的态度——我们就得到了图 3 中的平滑曲线。在进行大量编码前只花了几分钟的更改，在投产两年后将使用 30 分钟。我还看见一些项目花费数天乃至数星期来作出同类的决定，而没有去做他们今天该做的事并相信在以后需要的时候能够修改它。

随着有关变化成本的假定出现了这样的转变，现在是时候来对软件开发采用完全不同的方法了。它与其他方法一样严谨，只是在严谨的尺度上不一样了。与在早期做重大决定而在后期做较不重要的决定相反，我们可以创建一种软件开发的方法，它能够快速做出每个决定，并且用自动测试支持每个决定，这样就使你能够在了解到设计软件的更好方法时改进软件的设计。

但是，创建这样的方法并不容易。我们必须重复审查最根本的假定以确定什么是对软件开发有好处的。我们可分阶段来做这项工作。我们将以一个故事开始，这个故事将使我们明确如何做其他事情。



第 6 章

学会开车

我们需要通过做许多小的调整（而不是几次大的调整）来控制软件的开发，这有点像开车。也就是说我们需要反馈来知道我们何时出现了错误，我们需要很多机会来纠正这些错误，而且，我们必须能够以比较合理的成本完成这样的纠正。

现在我们知道了问题的大致情况——巨大的风险成本和通过选项来管理风险的机会——以及得出解决方案所需的资源：在生命期后期中做出更改而不会显著增加成本的自由度。现在我们需要开始把注意力集中到解决方案上。我们首先做一个比喻，一个大家都熟悉的故事，这样在压力大或者要做出决定的时候我们就可以参考它，以帮助我们正确工作。

我还清楚地记得我开始学开车的第一天。我和母亲正在靠近 Chico（加利福尼亚州）的 5 号州际公路上行驶，高速公路笔直而平坦，一直延伸到天边。妈妈要我从副驾驶座位上探身过来抓住方向盘。她先让我感觉方向盘的运动如何影响汽车的方向。然后告诉我，“这样来开：让汽车沿着车道的中线跑，一直向前。”

我非常认真地侧脸盯着前面的公路。我把车正好保持在车道中间，笔直地指向公路中间。我干得很棒。然后我开始有点心不在焉了……

当汽车压到路边的石头上时我才回过神来。母亲（现在想起来她的镇

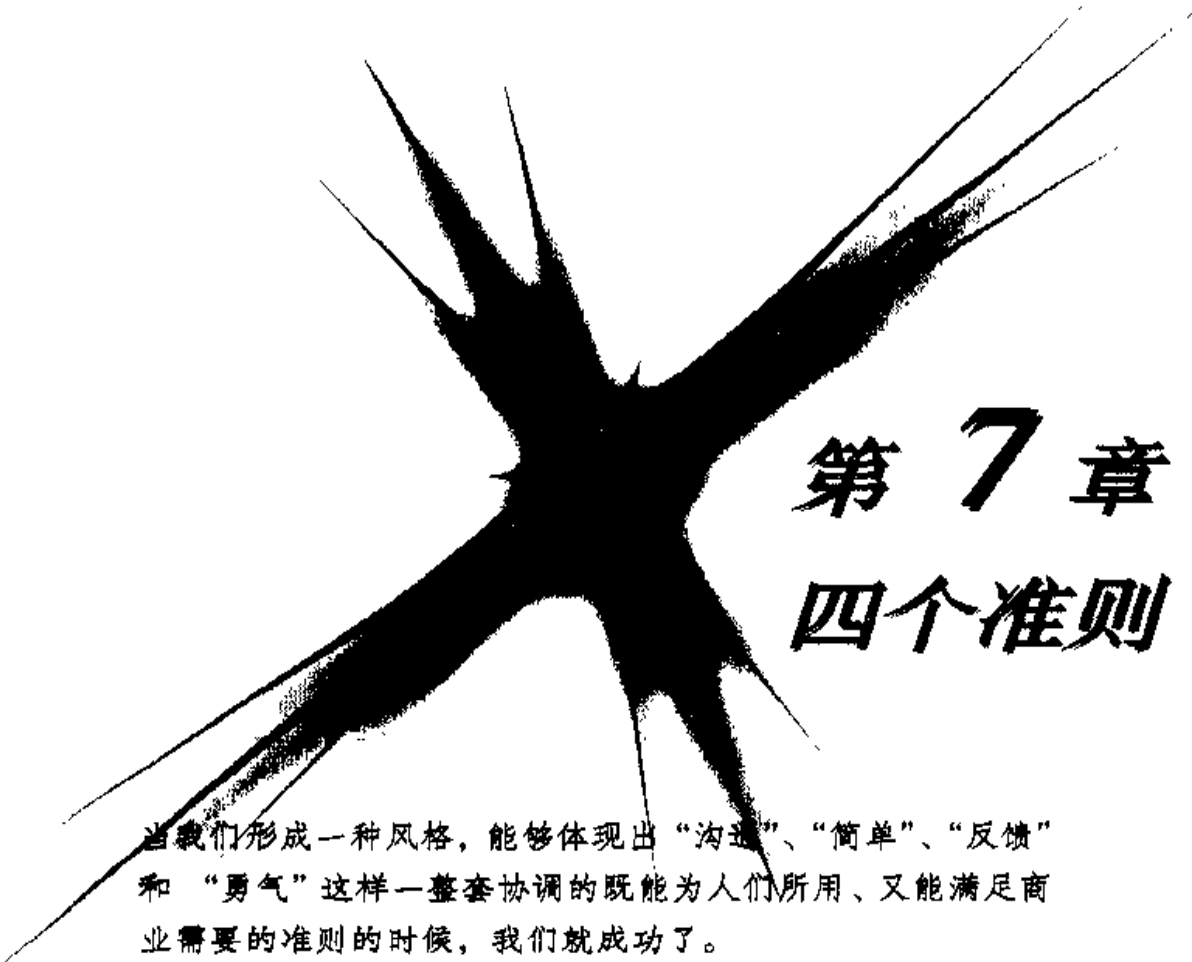
定真令我吃惊)逐渐将汽车开回到了正路上。然后她才真正教我开车。“开车并不是简单地把车开对方向。开车就要保持注意力集中,这样稍微调整一下,再那样稍微调整一下。”

这就是 XP 的范例。再没有像开车这样直白和简单的事了。即使工作看上去进行得很顺利,你也不要吧视线从公路上移开。只有变化是不变的。时刻准备着这样调整一点,那样调整一点。有时你可能不得不朝完全不同的方向前进。这就是作为一个程序员的生活。

软件中的所有东西都在变。需求变化、设计变化、业务变化、技术变化、团队变化、团队成员变化。问题不在于变化,因为变化总是要发生;问题实际上是在发生变化时没有能力应对。

客户就是软件项目的司机。如果软件不能完成客户要它完成的功能,你就失败了。当然,他们并不确切知道软件应该做什么。这就是软件开发像转动方向盘而不是把车笔直地沿着公路驾驶的原因。作为程序员,我们的职责是向客户提供方向盘并且提供有关我们在公路上的确切位置的反馈信息。

开车故事还对 XP 过程自身具有一定的寓意。下一章中描述四个准则——沟通、简单、反馈和勇气——将告诉我们软件开发应该是什么样的。但是,实践起来要做到这一点会因时间、地点和人不同而有所不同。在操控开发过程时,你可以采用能提供给你感觉的一套简单实践。随着开发的进行,你会经常注意到哪些实践可帮助你实现目标而哪些实践则有损于目标的实现。每个实践都是试验,在被证明不适用之前应该一直遵守。



第 7 章 四个准则

当我们形成一种风格，能够体现出“沟通”、“简单”、“反馈”和“勇气”这样一整套协调的既能为人们所用、又能满足商业需要的准则的时候，我们就成功了。

在能够将学习开车的故事演化成一套软件开发实践之前，我们需要用一些标准来判断我们的方向是否正确。如果在采用了一种开发风格之后却发现我们不喜欢它，或它无法工作，那会是件很麻烦的事情。

短期的个人目标经常与长期的社会目标相冲突。社会已经学会通过所形成的、为大众所接受的准则来解决这个问题，并利用神话、宗教仪式、惩罚和奖励来支持这些准则。如果没有这些准则，人们就容易转而追求自己最大的短期利益。

XP 的四个准则是：

- ✧ 沟通。
- ✧ 简单。
- ✧ 反馈。
- ✧ 勇气。

7.1 沟通

XP 的第一个准则是沟通。项目中出现的问题无一例外总是出自那些不愿与别人探讨重要问题的家伙身上。有时程序员不把设计中的重要变化告诉别人。有时程序员不向客户询问该问的问题，而导致在关键性领域的决策中出现严重失误。有时管理人员不向程序员询问该问的问题，而导致项目进度被误报。

沟通不良并不是偶然发生的。有很多情况会导致不良沟通。程序员向管理人员报告了一个坏消息，而管理人员却迁怒于他。客户告诉了程序员一些重要的事情，而程序员却把它当作耳旁风。

XP 旨在采用许多只能通过沟通完成的实践来保持良好的沟通。这是一些在短期内有意义的实践，如单元测试、结对编程及任务估算。测试、配对及估算的作用就是让程序员、客户和管理人员必须进行沟通。

这并不意味着在 XP 项目中沟通永远不会受到阻塞。人们会担心，会犯错误，也会分心。XP 雇用一位教练（coach），他的工作就是观察大家什么时候没有进行沟通，然后提醒大家。

7.2 简单

XP 的第二项准则是简单。XP 教练会问整个开发小组：“什么是能成功的最简单的东西？”

做到简单并非易事。世界上最难的事情就是不去考虑自己明天、下星期乃至下个月的任务。但是强制性地提前考虑正是屈服于对变化成本指数曲线的担心。有时教练要温和地提醒开发团队，告诉他们正在杞人忧天：“你可能比我聪明得多，你能弄明白这些复杂的动态平衡树问题。但我简单地使用线性查找也能工作。”

Greg Hutchinson 写道：

有个在我这里咨询的人认为我们需要一个通用对话框来显示文本（哎，好像我们的对话框还不够多似的，扯远了）。我们讨论了这个对话框的界面，以及它将如何工作。那个程序员认为这个对话框应该高度智能化，可以根据字体大小和其他变量来修改自身的大小和字符串中换行符的数目。我问他现在多少程序员需要它，只有他自己。我建议他暂时不要把对话框设计得如此智能化，先让它能够满足当前的要求（20 分钟的活），让大家了解它的类和接口，那么在下次需要这些需求时，就可以将它修改得更加智能化。我没能说服他，他们花了两天的时间来设计这些代码。到了第三天，连他们自己的需求都完全变了，也不再需要这部分代码了。本来已经很紧张的项目就这样浪费了两个人日。不管怎样，告诉我你究竟是否真的需要这些代码。（来源：电子邮件。）

XP 在打赌。它打赌今天做得简单一些，然后在明天需要时再多花些功夫进行改进，要比今天做得很复杂，但以后再也用不到要好得多。

简单和沟通之间有一种奇妙的互相支持的关系。沟通得越多，就越清楚哪些工作需要做，并能更加确信哪些工作不需要做。系统越简单，需要的沟通就越少，这将使沟通更加全面，尤其是在你能够将系统简化到需要更少程序员的时候。

7.3 反馈

XP 的第三个准则是反馈。比较教导化的口头禅是“不要问我，去问系统”和“你编写测试用例了吗？”对系统当前状态的具体反馈是极为宝贵的。乐观是编程这一行业的大忌，而反馈则是良药。

反馈可以以不同的时间规模来进行。首先，在分钟和天的级别里进行反馈。程序员为系统中所有可能出错的逻辑编写单元测试。他们每分钟都

得到有关系统状态的具体反馈。当客户编写新的“故事”（功能说明）时，程序员马上对它们进行估算，这样客户就会得到关于他们的故事质量的具体反馈。负责跟踪进度的人观察任务的完成情况，向整个团队发出有关他们能否在一段时间内完成已经开始的所有任务的反馈。

还可以在周和月的级别进行反馈。客户和测试者为需要系统实现的所有故事（考虑“简化的用例”）编写功能测试。他们会得到关于系统当前状态的具体反馈。客户隔两到三周检查一次日程，查看开发团队的整体速度是否与计划相符，并随之调整计划。随着系统的成形，它会马上被投入生产，因此公司会开始“感觉”系统运行的实际情况，并研究如何将它发挥至最佳效果。

提前生产需要一点解释。计划过程中的策略之一就是开发小组将最有价值的故事以最快的速度投入生产。这将向程序员提供有关他们决策和开发过程的质量的具体反馈，而这些只能在生产正式开始之后才可能得到。有的程序员从来没有接触过生产中的系统。他们怎么可能学着做得更好呢？

大多数项目似乎有着截然相反的策略。他们的想法是：“一旦系统投入生产，就不能再进行‘有意思的’的更改，因此要尽可能长时间地让系统停留在开发过程中。”

这实在是颠倒黑白。“在开发过程中”是一个临时状态，它在系统的寿命中只占很小的一个比例。应该让系统独立运作。你会被要求同时支持生产并开发新功能。你最好能尽快习惯于在生产和开发之间像变戏法一样快速转换。

具体的反馈与沟通和简单可以搭配工作。反馈越多，沟通就越容易。如果某人对你编写的代码有异议，并能向你提供一个能够证明其错误的测试用例，这就比花上千个小时来讨论设计美学要更有价值得多。如果你的沟通很清晰，你就知道应该测试和测量哪些内容来了解系统。简单的系统更容

易测试。编写测试使你能够管中窥豹地了解系统究竟能有多简单——在测试还不能运行的时候，你的工作还没有结束；而当所有的测试都能运行时，你的工作就完成了。

7.4 勇气

通过了前面三个准则——沟通、简单和反馈之后，该到了加速的时候了。如果你还没有拿出最快的速度，而别人做到了，他们就会抢走你的饭碗。

这里有一个有关行动中的勇气的故事。在第一个大型 XP 项目的第一个版本中，有一个 10 次迭代的工程计划，当进行到第 8 次迭代的时候（30 个星期中的第 25 个星期），开发小组发现了体系结构中的一个重大缺陷。功能测试的分数本来一直在令人满意地增长，但这下却变得平缓，远远低于原来的期望值。修复一个错误就会引起另一个错误。问题在于：这是体系结构上的缺陷。

（我们可以告诉那些好奇的人，这个系统是用来计算工资单的。拨款额代表公司应付给雇员的部分。扣除额代表雇员应付给别人的部分。有人在应该使用正的扣除额时一直误用了负的拨款额。）

开发小组做出了正确的决定。当他们发现没法再前进的时候，他们就修复了这个缺陷。这马上使原来运行通过的测试中的一半出现了错误。但是经过几天的集中攻关，测试的分数再一次向着胜利的方向前进。这就是勇气发挥的作用。

另一种勇敢的行为就是放弃原有的代码。有时候你整天处理一个程序，但它的状况并不理想，计算机也在这时候崩溃！如果第二天你花半小时的时间把昨天一整天的工作重新做完，并且这次既简单又清晰，你的感觉会怎样呢？

采用这种方法吧。如果一天已经接近尾声，代码却依然失控，那就把它扔掉。如果你很喜欢自己设计的界面，可以保存那些测试用例，但也可以不保存。或者完全从头开始。

或者，你可能有三种可选的设计方案。于是，花一天时间对每种方案进行编码，看一下它们的效果如何。然后扔掉代码，从最有希望成功的设计重新开始。

XP 的设计策略就像一个爬山算法。你做出了一个简单的设计，然后把它稍稍改得复杂一点，然后再简单些，然后再复杂些。爬山算法的问题在于达到了局部最优的时候，小的更改无法改进状况，只有大的更改才可以。

怎样才能保证自己不钻牛角尖呢？是勇气。每隔一会儿，开发小组中就可能会有人想出一个能够大大降低系统复杂性的古怪念头。如果他们有勇气，他们就会进行尝试。（有时候）那些办法确实可行。如果他们有勇气，他们还会把它应用到生产中。这样，你就相当于在攀登一座全新的山峰。

如果你不能满足前面三个准则，只有勇气，就是有勇无谋。然而，当把沟通、简单设计和具体反馈结合在一起时，勇气就十分可贵了。

沟通支持勇气，因为它带来了高风险、高回报的试验的可能性。“你不喜欢吗？我也讨厌那些代码。让我们看看一个下午我们可以替换多少。”简单支持勇气，因为有了简单的系统，你可以比以前勇敢得多。你无意中将其破坏的可能性也大大减小。勇气支持简单，因为只要有可能简化系统，你就会去尝试。具体的反馈支持勇气，因为如果你按下按钮就能够看到测试结果通过（或不通过，这时你就可以将代码扔掉），那么即使对代码进行根本的改动你也会感觉安全得多。

7.5 实践中的准则

我曾问过 C3 开发小组（我前面提到的第一个人型 XP 项目）他们在那

个项目中最自豪的时刻。我以为会听到关于大的重构、测试拯救或某个满意客户的故事。但事实上，我听到的是：

我最自豪的时刻是 Eddie 找到了一份离家更近的工作，不用每天花两小时乘车，这样他就能有更多的时间和家人在一起了。开发小组十分尊敬他。没人对他的离去表示异议。大家只是询问有什么能帮得上忙的地方。

这件事指出了更深刻的准则——尊重，它隐含在上面四个准则的表面之下。如果一个开发小组的成员不互相关心，也不在乎别人都在做什么，那么 XP 就完了。也许有很多其他方法来编写软件（或完成任何任务），但 XP 对这一点非常敏感。只要有一些基本的同情和兴趣，XP 就能够为所有的矛盾起到调解作用。如果开发小组的成员对项目漠不关心，那就无药可救了。只要有最低限度的热情，XP 就能给出正反馈。这并不是耍手腕；关键在于乐于成为一个有成效的团体中的一员，而不是一个瘫痪的团体。

这番高调的讲话说得没错，但如果不能将它付诸实践、巩固强化并将这些准则都形成习惯，那么我们只是大步跨入了另一个善意的方法论沼泽之中而已。接下来是一个更为详细的指导，教我们在实践中如何满足和体现这四个准则（沟通、简单、反馈和勇气）。



第 8 章

基本原则

我们可以从这四个准则衍生出许多基本原则来规范我们的新风格。我们将提出用开发实践来查看它们是否符合这些原则。

“学开车”提醒我们进行大量的小改动，并且要目不转睛地盯着道路。这四个准则——沟通、简单、反馈和勇气——向我们提供了成功解决方案的标准。但是，对于确定使用何种实践来说，这些准则太模糊了，无法向我们提供太多帮助。我们需要提炼这些准则，得出可以使用的具体原则。

这些原则将帮助我们在几个可选方法之间进行选择。我们将选择更加符合原则的方法。每一原则都体现着这些准则。准则可能会有些模糊。一个人觉得简单，而另一个人可能会觉得很复杂。原则就更加具体一些。无论你是否拥有快速的反馈。下面是一些基本原则：

- ◇ 快速反馈。
- ◇ 假设简单性。
- ◇ 递增更改。
- ◇ 提倡更改。
- ◇ 优质工作。

快速反馈——学习心理学让我们知道行动及其反馈之间的间隔是学习的关键。动物试验显示反馈的时间上即使有一丝差别都将导致学习上的巨大不同。刺激和反应之间有几秒钟延迟，老鼠就不知道红色按钮意味着食物。因此，原则之一是获取反馈，解释它并将所学到的知识尽快投入到系统中。业务员学习如何使系统能够发挥最佳作用，他们以天或周而不是以月或年为周期反馈所学知识。程序员学习如何最佳设计、实现并测试系统，他们以秒或分钟而不是以天、周或月为周期反馈所学知识。

假设简单性——就是把每个问题都看作可以用近乎荒谬的简单设计来解决。在可以这样解决的 98% 的问题上节省下来的时间向你提供了大量资源，可以应用到余下 2% 的问题当中。在许多方面，这是程序员最难接受的原则。我们以前一直被教导要为将来而计划，要为重用而设计。而 XP 则号召我们出色地完成今天的工作（测试、重构、沟通），并且相信自己具有将来必要时增加系统复杂性的能力。像选项等软件经济学就支持这种方法。

递增更改——一次性进行巨大改动不会起作用。甚至在瑞士（以谨慎计划而闻名，我的现居地），他们都尽量不做大的改动。任何问题都是通过一系列能带来差异的微小改动来解决的。

在 XP 中，你将会发现递增更改应用在许多方面。设计每次只改动一小点。计划每次只改动一小点。团队每次只改动一小点。甚至采用 XP 的过程都必须一小步一小步进行。

提倡更改——最佳策略是在实际解决最重要的问题的情况下保留最多选项的那一个。

优质工作——没有人喜欢做事拖泥带水。每个人都希望出色地完成工作。在四个项目开发变量（范围、成本、时间和质量）中，而质量实际上不是一个自由变量。因为它仅有的可能值只有“出色”和“极其出色”，这取决于是否会危及生命。否则，你将不会以工作为乐趣，不能出色完成工作，并且项目变得毫无价值。

下面有一些不太重要的原则。这些原则仍能帮助我们在特定情况下决定该做些什么。

- ◇ 教授知识。
- ◇ 小的初始投资。
- ◇ 积极求胜。
- ◇ 具体试验。
- ◇ 公开坦诚的沟通。
- ◇ 顺着人类的本能（而不是逆着它们）工作。
- ◇ 接受的责任。
- ◇ 本地调整。
- ◇ 轻装上阵。
- ◇ 诚实的度量。

教授知识——不使用教条主义者的论调（如“你必须像 XYZ 一样进行测试”），我们将关注传授用于确定应该做多少测试、做多少设计、多少重构和所有其他事情的策略。我们将以确定无疑的口吻申明这些观点，会有一些与我们不同的信心观点，我们将把它们作为策略来陈述，读者可以用来寻找自己的答案。

小的初始投资——在项目中太早和太多的资源是灾难的配料。严格的预算会迫使程序员和客户精简需求和步骤。严格的预算所产生的专注作用将激励你出色地完成所做的每件事。但是，资源有时也会变得太紧张了。如果你竟然没有足够的资源来解决一个重要的问题，那么你创建的系统也肯定不会是多重要的。如果有人指定范围、日期、质量和成本，那么你最后就不太可能成功地实现目的。然而，通常所有人都能适应资源不够充裕的情况。

积极求胜——看 John Wooden 的 UCLA 篮球队总是令人心情愉快。他们总是以绝对优势战胜对手。然而，即使到了比赛结束的最后几分钟，UCLA 也能完全肯定他们将要胜出。毕竟他们以前胜利过太多次了。因此，他们会很放松。他们做了该做的事情。他们又赢了。

我记得俄勒冈州篮球队就截然相反。俄勒冈队曾经与拥有四名 NBA 球员的国家级水平的亚利桑那州队比赛。到中场时，出乎意料的是，俄勒冈队高出 12 分。亚利桑那队不知所措，俄勒冈队的进攻使他们晕头转向。但是，中场过后，俄勒冈队开始尽量放慢节奏，希望减少对方得分而将优势保留到比赛结束。当然，这种策略没有发挥作用。亚利桑那队找到了机会，用他们技术上的巨大优势取得了这场比赛的最终胜利。

两者的区别在于为了胜利而比赛和为了不输而比赛之间。我所见到的大多数软件开发是为了不输而比赛。写了大量文章、开了很多会，每个人都尽量“按规章”开发，不是因为它特别有道理，而是因为他们想在最后能够说这不是他们的过错——他们是遵照过程进行工作的。

积极求胜的软件开发完成任何有助于成功的工作，而不做任何于取胜无补的事情。

具体试验——每次做了决定而不对它进行测试，都可能会做出错误的决定。所做的决定越多，风险越大。因此，设计会议的结果应该是一系列针对会议中提出的问题的试验，而不是已完成的设计。需求讨论的结果也应该是一系列试验。每一个抽象的决定都应该进行测试。

公开坦诚的沟通——这真是条婆婆妈妈的教诲，我几乎把它漏掉了。谁不想与人公开并坦诚地进行沟通呢？我所见到的每个人都如此。程序员必须能够解释其他人所做决定的结果，“你在这里违反了封装，让我很难做。”他们必须能够互相告知代码中的问题所在。他们必须自由地表达他们的担心并获得支持。他们必须自由地向客户和管理人员传达坏消息，尽早传达，而不应被惩罚。

如果我看到有人回答问题之前在环顾四周看看谁在听着，我认为这是项目有大问题的一个信号。如果有个人问题需要讨论，我能够理解尊重隐私的需要。但是决定使用两个对象模型中的哪一个就不应该是那种需要盖“最高机密”戳记的事情。

顺着人类的本能（而不是逆着它们）工作——人们喜欢胜利。他们喜欢学习，喜欢与别人交流，喜欢成为团队的一部分，喜欢将事情置于掌控之中，喜欢受到信任，喜欢出色地完成工作，喜欢使他们的软件发挥作用。

Paul Chisolm 写道：

在开会时，想成为经理的质量保证人员建议添加半打字段（添加到一个在线窗体，这个窗体已经填满了从来没有人使用的数据），不是因为这些信息以后会有用，而据说是因为填充这些额外的字段将会“节省时间”。我的反应：我用我的脑袋重重地撞击会议室的桌子（就像华纳兄弟的卡通人物刚听到一些难以置信的事情时那样），并告诉他不要再对我说谎。（直到今天，我也不知道那是我所做过的最不专业的事情之一，抑或是最专业的东西之一。一个眼科医生告诉我不要再用头去撞东西，因为那会使视网膜脱落。）（来源：电子邮件。）

根据短期的个人利益设计进程而同时也符合团队的长远利益，这需要一定技巧。你可以详细解释你对某个实践或其他事物的计划如何在长远上符合所有人的最佳利益，但是当压力变大时，如果这种实践不能“解决”最紧要的问题，那么它最终会被放弃。如果 XP 不能满足人们的短期利益，那么它必然会被遗弃到无用的方法论的黑暗中。

有些人很喜欢 XP 的这个特点：提倡让程序员按照他们自己的构想完成工作，而仅对使整个过程保持在正轨上进行控制。我记得有这样一句话：“XP 完全符合程序员的理念。”

接受的责任——再没有什么比被要求去做什么更能剥夺团队或个人的

活力了，尤其是当工作显然是不可能完成的时候。权威仅在让人们像他们往常那样工作的时候能够发挥作用。于是，被要求去做什么的人将找出无数种表达他们的挫折感的方法，其中大多数会对团队造成损害，也有一些会对他自己不利。

另一个选择是责任应该被接受，而不是被赋予。这并不意味着你总是做自己喜欢做的事情。你是团队的一部分，如果团队得出需要完成某个任务的结论，就应该有人去完成它，无论这个任务是如何讨厌。

本地调整——你必须调整在本书中所读到的内容以适应你的具体情况。对于你的开发过程来说，这将是接受的责任的一个应用。采用 XP 并不意味着让我来决定你如何开发。它意味着你自己决定如何开发。我可以告诉你我发现的有成效的东西。我可以指出我看出来的偏离的后果。不管怎么说，这是你自己的开发过程。你需要在今天决定一些事情；你必须了解它明天是否还有效；你必须改变并适应。请不要读出这种思想：“现在我终于知道如何开发了。”你应该在结束时说：“难道我不得不做出所有这些决定才能编程吗？”是的，你需要这么做。但是这是值得的。

轻装上阵——不要希望在携带许多行李时还能快速移动。我们使用的制品应该：

- ◇ 少。
- ◇ 简单。
- ◇ 有价值。

XP 团队应该是智慧的游牧人，随时准备迅速收起帐篷，跟随牧群到处流浪。这里的牧群可能是与预期不同方向的设计、与预期不同方向的客户、离开的团队成员、突然升温的技术或者是不断变换的商业环境。

就像游牧人一样，XP 小组习惯于轻装上阵。除了不断向客户提供价

值所需的東西——測試和代碼外，他們不帶很多行李。

誠實的度量——對控制軟件開發的追求使我們去度量，這沒有問題，但是我們已經在精確度量的路上走到了我們的儀器支持不了的地步了。如果你沒有以這種精確等級估算的能力的話，寧可說“這將花費兩周時間，或者長些或者短些”比說成“14.176 小時”更好。我們還將努力選擇與我們希望的工作方式相關的度量方式。例如，當我們學會更好的編程方式後，代碼行數將因為減少的代碼而成為沒用的度量單位。



第 9 章 回到基本问题

我们希望能够竭尽全力做到稳定、可预测的软件开发。但是我们没有时间去做任何额外的事情。开发软件的四项基本工作是：编码、测试、倾听和设计。

“学习开车。”四条准则：沟通，简单，反馈和勇气。还有更多一些的原则。现在我们已经能够开始制定软件开发的规范了。第一步是决定范围。我们将要规范的是什麼？我们应该重视哪些问题和忽略哪些问题？

我还记得第一次学习用 BASIC 编程时的情形。我有几本讲述编程基本理论的练习册。在很快地把它们练习了一遍之后，我想解决一个比书中的练习更难的问题，我决定编写一个“星际追踪”游戏，它有点像我在伯克利的劳伦斯科技馆玩过的一个游戏，但是更酷些。

在为解答练习册上的练习编写程序时，我先花了几分钟盯着某个问题考虑一下，然后键入解决该问题的代码，接着再处理出现的任何问题。于是按照这个顺序，我很自信地坐下来开始编写这个游戏，但是写不出来。我不知道如何编写多于二十行的应用程序，所以我试着在键入程序之前把整个程序写在纸上，可是只写了三行就再也写不下去了。

我需要做一些编程以外的事情。可我不知道该去做些什么。

那么，如果我们回到以前的状况，但是却多了经验，会是什么样子？我们会怎样做？我们清楚不能只是“编码直到完成为止”，我们还需要做哪些工作？当重新做每一项工作时，我们该试着从中获得什么？

9.1 编码

在一天结束的时候，必须编出来一个程序。所以，我认为编码是一项必不可少的工作。无论是绘制生成代码的图表或者是在浏览器中键入内容，都是在编码。

那么，我们想从代码中得到什么呢？最重要的事情是学习。我学习的方法是先有一个构思，然后对它进行测试，看看这种构思好不好。编写代码是据我所知最好的实现方式。代码不会受修辞的魅力和逻辑所摆布。它不关心你有没有人学文凭或高薪。它只是坐在那里，心甘情愿地做你应做的工作。如果情况不是这样的，问题肯定出在你自己那里。

编码时，还有机会去理解代码的最佳结构。通过代码中的某些标志，你可以知道是否理解了它的基本结构。

代码还给了你进行简洁明了的交流的机会。如果你有一种想法并把它解释给我听，我可能很容易发生误解。但是，如果我们一起对这一想法进行编码，我能在你编写的逻辑中精确地读到你的想法。当然，我了解的并不是你大脑里的想法，而是它们在外部世界中得到体现后的样子。

这种交流很容易转换成学习：我了解了你的想法，同时也有了我自己的想法；因为不能准确无误地把它表达出来，所以我也求助于代码；由于我的想法与你的想法是相关的，因此我们使用了相关的代码；你看到我的想法后，又有了新的想法。

最终，代码成了一种开发所不可或缺的制品。我曾经听说过有些系统的源代码丢了，却仍然在生产，不过这样的怪兽系统变得越来越少了，因

此一个系统要生存就必须保留源代码。

既然源代码必不可少，我们就应该尽可能多的将它用于软件工程。事实证明代码可以用来沟通——表达策略性意图，描述算法，指出将来可能进行扩展和收缩的地方。代码还可以用来表示测试，这些测试客观地对系统的运行进行测试，同时为系统在各个级别提供宝贵的运行规范

9.2 测试

英国实证哲学家洛克、贝克莱和休姆曾经表示，任何不能度量的事物都是不存在的。讲到代码，我完全同意他们的观点。不能使用自动化测试证实的软件功能是不存在的。我很有办法让我自己相信我所编写的就是我想要表达的，我想要表达的就是我应该要表达的。所以在测试之前我不相信任何自己编写的东西。测试提供了一个机会，使我可以不考虑如何实现，只考虑我想要什么。然后测试会告诉我是否实现了我认为我实现了的东西。

当多数人想到自动测试时，他们想到的是测试功能，即计算哪些数字。随着有了更多编写测试的经验后，我发现可以为非功能性需求（如性能或遵守代码标准）编写测试。

Erich Gamma 造了一个新词“测试症”（Test Infected），用来形容那些若没有测试就不编码的人。测试告诉你何时完成了编码——当测试能够运行时，编码工作就暂时完成了。当你想不出其他可能出错的测试时，编码就全部完成了。

测试既是一种资源又是一种职责。你不能只编写一个测试，运行它，然后宣布收工。你有责任编写你能想到的每一个不能立即运行的测试。不久后，你就会成为分析测试的高手——如果两个测试都成功了，那么完全可以推断第三个测试不必编写也能运行。当然，也正是这种分析导致了程序中的错误，所以一定要小心。如果日后出现了问题，而且这些问题如果

编写了第三个测试的话原本能够发现，你就要准备好接受教训，下一次记着编写第三个测试。

多数软件没有经过全面的自动化测试就交付了。自动化测试显然并不是特别重要。那么我为什么还要把测试当成基本工作呢？有两个答案：一个短期的和一个长期的。

长期的答案是测试可以使程序的生存时间更长（如果运行并维护测试的话）。有了测试，你可以在更长时间内进行更多更改。如果坚持编写测试，你对系统的信心会随着时间的推移不断增强。

我们的原则之一是顺应人的天性而不是违反它。如果你能做的只是长期争论测试的必要性，你完全可以忘掉它。有些人做测试是出于责任感或者是有人要求他这么做。一旦那种要求放松或者压力增加，不但不会有新测试被编写出来，已经写出来的测试也不会被运行，整个项目会变得一团糟。所以，如果希望顺应人的天性并想要进行测试，我们必须为测试找到一个短期的和自私的理由。

幸运的是，确实有一个编写测试的短期理由。有了测试，编程的工作比没有测试时有趣得多，而且编码时你会更有自信。永远不用这样来烦自己：“嗯，这就是现在应该做的事情，但我不知道有没有犯什么错误。”按下按钮，运行所有测试，如果绿灯亮了，你就可以充满自信地进行下一步。

我曾经在一次公开编程演示活动中这样做过。每次面向观众转向编程时，我都要按下测试按钮。我没有更改任何代码，环境也没有任何改变，我只是想要一点自信。看到测试仍能运行就能给我这种自信。

编程和测试相结合也比只编程快。开始时，我并没有想到这一点，但是后来我注意到了，也听许多其他人讲过。半小时内不进行测试，你可能会赢得更高的生产率，但是一旦你习惯了做测试，就会很快注意到生产率上的差异。做测试能够节省调试的时间——你不再需要花费一小时来查找错误，你可以在几分钟之内找到它——这就是生产率提高的来源。有时候

你实在没办法使测试运行，你可能是遇到了更大的问题，必须退回去，确保测试是否正确，或者是否需要改进设计。

然而，有一种情况很危险。做得很差的测试是一种让人觉得万事无忧的有色眼镜。你会由于测试都能运行而得到系统一切正常的错觉。继续做下去的时候几乎不会意识到已经留下了一处可怕的陷阱，就等着你下一次再经过时掉进去。

测试的一个窍门是找出所能容忍的缺陷的级别。如果你每月只能忍受一个客户的抱怨，那么就在测试上投资，改进测试过程，直到能达到这个级别。然后，使用该测试标准继续工作，此时可以认为如果测试能运行，系统就一切正常。

接下来，我们将有两套测试。一套是程序员编写的单元测试。程序员使用这些测试来确保程序以他们所期望的方式运行。另一套是客户编写的（至少是由客户指定的）功能测试。客户使用此测试来确保系统整体上以他们所期望的方式运行。

测试有两类观众。程序员需要以测试的方式巩固他们的自信，这样别人也能够分享这种自信。客户需要准备一套测试来表达他们的自信：“好的，我想如果能计算所有这些事例，系统肯定能正常工作。”

9.3 倾听

程序员什么都不懂。更准确的说，他们对业务员感兴趣的事情一无所知。嘿，如果那些业务员用不着程序员了，他们会立刻把我们赶出大门去。

我有点离题了。如果你决心要测试，就必须想办法得到答案。既然（作为一个程序员）你什么都不懂，就必须去问别人，他们会告诉你想要的答案，以及从业务的角度来讲哪些是特例。

如果想问问题，最好先做好倾听答案的准备。因此倾听是软件开发的

第三项工作。

一般来说程序员也必须倾听。他们倾听客户讲述业务上的问题，帮助客户理解什么容易做，什么难做，所以这是一种积极的倾听。程序员提供的反馈能够帮助客户更好地理解他们的业务问题。

只说“你们应该互相倾听，并且要倾听客户的话”没什么用。人们试过这样做，但是没用。我们必须找到一种构建沟通的方法，使得应该沟通的事情得到沟通，并且以适当的详细程度得到沟通。同样地，我们开发的规则也必须阻止一些无用的沟通的进行，比如在真正理解要交流的内容之前所做的沟通，以及由于过于详细而掩盖了重要部分的沟通。

9.4 设计

为什么不能只是倾听、编写一个测试用例、运行它、倾听、编写一个测试用例、运行它，这样一直进行下去呢？因为我们知道正确的方法不是这样的。一段时间内你可以这样做，如果使用的是一种更灵活的语言，你甚至可以在很长一段时间内这么做。但是，最终你会遇到困难。让下一个测试用例运行的唯一方法是破坏另一个，或者使测试用例运行的唯一方法根本不值得使用。熵运动又有了一个新的牺牲品。

避免这些问题的唯一方法是设计。设计就是创建组织系统中的逻辑的结构。好的设计能够这样组织逻辑：对系统中某一部分的更改不会总是需要对其他部分也进行更改。好的设计可以确保系统中的每一部分逻辑有且只有一个“家”。好的设计将逻辑放在它所操作的数据附近。好的设计使得能够只对一个部分进行更改而扩展系统。

糟糕的设计正好相反：一个概念性的更改需要对系统中的很多部分进行更改；逻辑必须被复制。最后，糟糕设计的代价让人难以忍受。你再也记不起来所有暗含的相关更改所发生的地方；如果不破坏现有功能就无法

添加新功能。

复杂性是糟糕设计的另一个来源。如果一个设计需要间接的四个层次才能弄明白要做什么，而且这些层次没有任何特别的功能性或解释性的作用，那么这种设计就是糟糕的。


所以，在新规范里我们必须做的最后一项工作是设计。我们必须提供这样一种环境：能够产生好的设计，改正糟糕的设计，而且任何需要学习的人都可以学习当前的设计。

在下面的章节中，你会了解到 XP 进行设计的方法与许多软件开发过程所采用的方法有很大的区别。设计是所有 XP 程序员在编码过程中要做的日常工作的一部分。但是不管采取的是什么样的策略，设计工作是必不可少的。要进行高效的软件开发就必须认真地考虑设计。

9.5 结论

因此，要编码是因为如果你不编码，就什么也没做；要测试是因为如果你不测试，就不知道编码何时完成；要倾听是因为如果你不倾听，就不知道为什么编码或要测试什么；要设计是为了能够可以不断进行编码、测试和倾听。就是这样。这些就是我们要努力构建的活动：

- ◇ 编码。
- ◇ 测试。
- ◇ 倾听。
- ◇ 设计。



第2部分 解决方案

现在我们已经完成了初期工作。我们知道要解决什么问题，那就是决定如何进行软件开发的基本活动——编码、测试、倾听和设计。我们有一套指导性的准则和原则，可以在每次做出基本活动的决策时为我们提供指导。而且，我们拥有平滑成本曲线这张秘密王牌，它能够简化所选择的策略。



第 10 章

简短概述

我们将依靠简单实践（就是那些几十年前常常被视为不切实际或天真而遭摒弃的实践）之间的协作。

我们的新软件开发规范的原材料有：

- ◇ 学开车的故事。
- ◇ 四条准则——沟通、简单、反馈和勇气。
- ◇ 原则。
- ◇ 四个基本活动——编码、测试、倾听和设计。

我们的工作就是构造这四个活动。我们不但必须构造这些活动，而且不得不根据一长列、有时是相互矛盾的原则来进行构造。同时我们必须尽量改进软件开发在经济方面的表现，这样才会有人倾听。

没问题。

这个……

由于本书的目的在于解释这怎样才能奏效，所以我将简短地说明一下 XP 中的实践的主要领域。在下一章里，我们将了解到这些简单得可笑的解决方案如何能够大显奇功。在某种实践较弱的地方，就会有其他实践的

长处来弥补这个弱点。后面的章节将更加详细地讨论一些这方面的主题。

首先，下面是所有的实践：

- ✧ 计划游戏——通过结合使用业务优先级和技术评估来快速确定下一个版本的范围。当计划赶不上实际变化的时就应更新计划。
- ✧ 小版本——将一个简单系统迅速投产，然后以很短的周期发布新版本。
- ✧ 隐喻——用有关整个系统如何运行的简单、众所周知的故事来指导所有开发。
- ✧ 简单设计——任何时候都应当将系统设计得尽可能简单。不必要的复杂性一旦被发现就马上去掉。
- ✧ 测试——程序员不断地编写单元测试，在这些测试能够无误地运行的情况下，开发才可以继续。客户编写测试来证明各功能已经完成。
- ✧ 重构——程序员重新构造系统（而不更改其行为）以去除重复、改善沟通、简化或提高柔性。
- ✧ 结对编程——所有的生产代码都是由两个程序员在同一台计算机上编写的。
- ✧ 集体所有权——任何人在任何时候都可以在系统中的任何位置更改任何代码。
- ✧ 持续集成——每天多次集成和生成系统，每次都完成一项任务。
- ✧ 每周工作 40 小时——一般情况下，一周工作不超过 40 小时。不要连续两个星期都加班。
- ✧ 现场客户——在团队中加入一位真正的、起作用的用户，他将全职负责回答问题。
- ✧ 编码标准——程序员依照强调通过代码沟通的规则来编写所有代

码。

在本章中，我们将简短概述执行每种实践所涉及的内容。在下一章（这如何奏效）中，我们将探讨这些实践之间的联系，使得一种实践的弱点可以由其他实践的长处来弥补。

10.1 计划游戏

业务上和技术上的考虑都不应是压倒一切的。软件开发始终是一种可能的和想要的之间不断讨价还价的对话。对话的本质是：它会更改认为什么是可能的和认为什么是想要的这两者

业务人员需要决定的内容有：

- ◇ 范围——对系统而言，将问题解决到什么程度在生产中才算是有价值的？业务人员知道多少为不够，而知道多少又太多。
- ◇ 优先级——如果你最初只能拥有 A 或 B，那你想要哪一个呢？业务人员能够确定这一点，在这方面他们要比程序员强得多
- ◇ 版本的组成——软件需要完成多少才能使业务能够因为使用了该软件而改善状况？程序员对于这个问题的直觉可能会与正确答案人相径庭。
- ◇ 发布的日期——软件（或软件的一部分）的发布会造成很大影响的重要的日期是什么时候？

在业务上，不可能凭空做出这些决策。软件的开发需要做出为业务决策提供原始材料的技术决策。

技术人员决定的内容有：

- ◇ 估算——实现一种功能需要花费多长时间？
- ◇ 后果——有一些战略性的业务决策只有在了解到技术后果后才能做出。数据库的选择就是一个很好的例子。业务人员可能宁愿与一家大公司合作，也不与一个刚起步的企业合作，但是两倍的生产力可能会使它值得承受额外风险和不便之处，或者不是这样。开发人员需要解释这些后果。
- ◇ 过程——如何组织工作和团队？团队需要适应所处的文化。但是你首先应当将软件做好，而不是保持一种封闭的文化的非合理性。
- ◇ 详细的日程计划——在一个版本内，先完成哪些任务呢？程序员需要有可以首先安排风险最大的开发阶段的自由，以减少项目的总体风险。在这种约束下，他们仍倾向于在过程中提高业务优先级，从而减少在临近某个版本开发的结尾时不得不将重要的功能舍弃掉的可能性。

10.2 小版本

每个版本都应尽可能地小，包含最有价值的业务需求。作为一个整体，版本必须是有意义的——就是说，你不能仅为了缩短发布周期而只实现了一半功能就发行。

每次计划一个或两个月远好过每次计划六个月或一年。向客户交付大型软件的公司可能不能这样经常发布软件。他们还是应该尽可能地缩短周期。

10.3 隐喻

每个 XP 软件项目都是由一个全面的隐喻指导的。有时隐喻是“幼稚

的”，例如使用合同、客户和背书（endorsement）等术语描述的合同管理系统。有时隐喻需要一些解释，例如说计算机屏幕显示是桌面，或者退休金计算就像电子表格一样。这些都是隐喻，因为我们不会照字面意义理解为“系统是电子表格”。隐喻只是帮助项目中的每个人理解基本元素及其关系。

用于标识技术实体的词语应该从选择的隐喻中前后一致地提取。随着开发的继续进行和隐喻的逐渐成熟，整个团队将从对隐喻的探讨中找到新的灵感。

XP 中的隐喻在很大程度上取代了别人所说的“体系结构”。将那个 10000 米长的系统视图称作体系结构的问题是体系结构并不一定能够对系统进行任何意义上的浓缩。体系结构就是大方框和连接。

你可以说，“做得很差的体系结构当然很糟糕。”我们需要强调体系结构的目的，那就是向每个人提供一个进行工作的连贯的情节，一个为业务和技术人员所周知的情节。通过引入隐喻，我们可以得到易于沟通和阐述的体系结构。

10.4 简单设计

在任何时候，正确的软件设计都具有下面的特征：

1. 能够运行所有测试。
2. 没有重复的逻辑。请注意隐蔽的重复（如平行类层次结构）。
3. 陈述每个对程序员重要的意图。
4. 有尽可能少的类别和方法。

系统中的每一部分设计都必须是符合现有约定的这些条款。Edward Tufte¹ 为图形设计者提供了一个练习——随心所欲地设计图形。然后，擦掉所有不减少任何信息的部分。不能再擦掉而保留下来的就是正确的图形设计。简单设计就像这样——去掉任何可以拿掉而不违反规则 1、规则 2 和规则 3 的设计元素。

这是与你通常听到的相反的建议：“为今天实现，为明天设计。”如果你认为未来不可确知，并且认为可以不费力地改变你的想法，那么根据推测就增加功能是不理智的，在需要时再添加你需要的东西。

10.5 测试

没有经过自动化测试的程序功能根本不存在。程序员编写单元测试，以便使他们对程序运行的信心可以成为程序本身的一部分。客户编写功能测试，这样他们对程序运行的信心也能够成为程序的一部分。结果是随着时间推进程序变得越来越可靠——它接受变化的能力变得更强，而不是更差。

你不必为所编写的每个方法都编写测试，而只对有可能会出错的方法编写。有时你只是想知道某个想法是否可能实现。你就仔细地研究了半个小时。的确，它是可能的。于是你就扔掉代码，重新从测试开始。

10.6 重构

在实现程序功能时，程序员一直想知道是否有一种更改现有程序从而

1. Edward Tufte. The Visual Display of Quantitative Information. Graphics Press, 1992.

使添加功能变得简单的方法。在添加了功能后,程序员在继续运行所有的测试的同时,会考虑现在是否可以看出如何使程序更加简单。这称为重构。

请注意,这意味着有时你要完成的工作量比仅仅使一项功能能够运行所需的工作更多。但是以这种方式工作,你就可以确保用合理的工作量添加一个又一个功能。不要根据猜测进行重构;应在系统要求你重构时再那么做。当系统要求你复制代码时,它也就是在要求重构。

如果程序员有两种方法:一种只需用一分钟就可以完成测试但设计很累赘,另一种需要用十分钟才能完成测试但设计更加简单,那么正确的选择是付出那十分钟。幸运的是,你可以使用几个小的、低风险的步骤对系统的设计做出重大更改。

10.7 结对编程

所有的生产代码都是由两个人使用同一台计算机,用同一个键盘、同一个鼠标编写的。

每一对成员中都有两种角色。其中一个使用键盘和鼠标的成员应思考实现此方法的最佳途径。另一个成员应更加偏重于战略性的角度进行思考:

- ◇ 整个方法会奏效吗?
- ◇ 还有哪些其他测试用例可能会失败?
- ◇ 有没有什么方法可以简化整个系统,以便使当前的问题干脆消失掉呢?

配对是动态的。如果有两个人上午配了对,他们可以在下午很容易地与其他人配对。如果你负责的任务是你所不熟悉的领域,那么你可以要求近期内对该领域有工作经验的人与你配对。更常见的是,团队中的任何人

都能充当成员。

10.8 集体所有权

如果有任何人发现改进任何部分代码的机会，他应该立即执行改进。

将这个代码所有权模型与其他两个代码所有权模型（无人所有权和个人所有权）进行比较。在过去，没有人拥有任何特定的代码。如果有人想更改某些代码，他们那么做肯定只是为了满足个人目的，无论是否符合当时的实际情况。结果是造成混乱，特别是对于那些很难静态地确定其中一行代码与另一行代码行之间的关系的对象来说。代码增长得很快，但是它也迅速地变得不稳定。

为了控制这种情形，引出了个人代码所有权。只有代码的正式所有者才可以更改代码。其他所有人在发现代码需要更改时，都必须向代码的所有者提交请求。实行严格所有权的结果是代码变得不为团队所理解，原因是人们不愿意打扰代码的所有者。但是，代码需要现在马上更改，而不是以后。因此虽然代码保持稳定，但它们发展的速度却不如应有的速度那样快。然后所有者离开……

在 XP 中，每个人都负责整个系统。并不是每个人都对各部分有同等的了解，但是每个人都对各部分有所了解。如果某对程序员在工作的时候发现一个改进代码的机会，那么他们就改进它（如果这样做能让他们的生活更轻松的话）。

10.9 持续集成

几小时（最多一天）的开发后对代码进行集成和测试。要这样做，有一个简单的方法，那就是用一台计算机专门做集成工作。当计算机空闲时，

有要集成的代码的程序员组合就坐下来，调用当前的版本，加载他们的更改(检查并解决任何冲突)，并一直运行测试到它们通过为止(100% 正确)。

同时集成一组更改效果很好，这是因为谁应该修复失败的测试是很明显的——是我们，因为前一对离开时测试是 100% 正确的，那么一定是我们使它出错了。如果我们无法使测试 100% 正确地运行，就应该放弃所做的更改，重新开始，原因是我们显然对编写该功能没有足够的了解(尽管现在我们可能确实了解得足够清楚了)。

10.10 每周工作 40 小时

我希望每天早晨都是精力充沛的，每天晚上都是疲惫又心满意足的。我希望星期五那天能够足够地疲惫和心满意足，这样我就可以舒坦地用两天时间考虑一些工作之外的事情。然后我希望星期一能够充满热情和创意地投入工作。

这是否意味着我们每周要在工作场所工作整整 40 小时并不是特别重要。不同的人对工作有不同的耐力。一个人也许能够精力集中地投入 35 小时，而另一个人可以达到 45 小时。但是没有人能够连续好几周每周工作 60 小时仍能保持精力充沛、富有创造力、仔细和自信。别那么做。

加班是项目存在严重问题的征兆。XP 规则很简单——你不能连续两周加班。如果只在一个星期内加班，那么还好，有点变得暴躁，但还是可以投入一些额外的时间。如果在星期一开始工作时说，“如果要达到我们的目标，我们必须再加班，”这时你就已经遇到一个问题，这个问题无法通过工作更长时间来解决。

相关的问题是假期。欧洲人经常连续度假，两周、三周或四周。美国人很少一次度假超过几天。如果是我的公司，我将坚持让大伙每年度假两周，同时还有至少另外一周或两周供短期休假用。

10.11 现场客户

真实的客户必须与团队坐在一起，以便回答问题、解决争端和确定小规模者优先级。我用“真实的客户”来指那些在系统投产后真正使用系统的人。如果你正在建立一个客户服务系统，那么这个客户应该是一个客户服务代表。如果你正在生成一个债券交易系统，那么这个客户应该是债券交易人。

这条规则的一个大缺陷是对于处于开发状态的系统向团队提供一个真实用户成本太高。管理人员必须决定下面这两个哪个价值更高——让软件能更快和更好地投入生产，还是支出一个或两个人的费用。如果使用该系统给业务带来的价值并不比多雇用一个人工作更高，或许就不应该建立这个系统。

况且，团队中的客户并不是不能完成任何工作。即使是程序员也不可能每周都产生 40 小时的问题。现场客户会有与其他客户分隔开的弱点，但是他们可能会有时间进行他们的正常工作。

现场客户的不利情况是在他们花费了数百个小时帮助程序员后，项目被取消了。那么你就失去了客户所做的工作，也失去了他们如果不把精力放在一个失败的项目上时，原本能够做的工作。XP 尽各种可能确保项目不会失败。

我曾经从事过一个项目，那时我们很勉强地得到了一位真实的客户，不过“只能用一小会儿”。系统成功发行（并且显然能继续发展下去）后，客户方面的管理人员向我们提供了三位真实客户。那家公司本可以通过更多的业务配合从该系统中获得更多好处。

10.12 编码标准

如果你要让所有这些程序员能从系统的一个部分换到另一个部分，一天数次交换成员，并且经常重构对方的代码，那么你肯定承受不了几套不同的编码实践。经过了一些实践后，将不可能判断团队中谁编写了什么代码。

标准应提倡可能的最小工作量，这与“一次且仅一次”规则（没有任何重复代码）一致。标准应强调沟通。最后，标准必须被整个团队自愿地采纳。



第 11 章

这如何奏效

实践互相支持。一种实践的弱点可以由其他实践的优点来弥补。

等一下。前面描述的实践中没有一个是独特或首创的。它们从编写程序开始的时候就被使用了。这些实践中的大多数已经由于它们的弱点日渐明显而被放弃，人们使用了更复杂、成本更高的实践。但 XP 为什么不是使软件变得易如反掌的方法呢？在继续之前，我们最好先说服自己：这些简单的实践不会像几十年前扼杀软件项目一样置我们于死地。

更改成本的指数曲线的崩溃使得所有这些实践得以重拾当年风光。每种实践仍旧有以前的弱点，但是如果现在这些弱点可以用其他实践的优点来弥补，情况会如何呢？或许我们可以轻松地完成工作。

本章向你展现对这些实践的另一种看法。但是，这一次我们将把注意力放在实践的弱点，并说明其他实践怎样使每种实践的不利影响不会搞砸整个项目。本章也说明整个 XP 理论是如何发挥功效的。

11.1 计划游戏

你不可能只从一个粗略的计划开始软件开发。你也不能经常更新计划——那会花费太长时间，使客户感到厌烦。除非：

- ◇ 客户基于程序员所提供的估算自己更新计划。
- ◇ 开始的时候你就有足够的计划给客户一个接下来几年中可能的进展的粗略概念。
- ◇ 采用短发行周期，以便计划中的任何错误最多只会有几周或几个月的影响。
- ◇ 客户和团队坐在一块，这样他们就能很快发现可能的变化和改进的机会。

如果是这样，你或许就可以从一个简单的计划开始开发软件，并在工作进行过程中不断改善它。

11.2 短发行周期

你不可能几个月后就投入生产。你当然无法以一天或几个月的周期发布系统的新版本。除非：

- ◇ “计划游戏”能够帮助你处理最有价值的部分，因此即使一个小系统也有商业价值。
- ◇ 你在持续集成，因此打包一个小版本的成本很小。
- ◇ 测试已经使缺陷率非常低，这样在软件出手之前你不需要经过一个冗长的测试周期。
- ◇ 可以做一个简单的设计，对这个版本够用就行了，而不需要照顾

一辈子。

如果情况是这样，你或许就可以在开始开发后很快发布小的版本。

11.3 隐喻

你不可能仅用一个隐喻就开始开发软件。隐喻里没有足够的详细信息。况且，如果错了怎么办？除非：

- ✧ 很快会有来自实际代码和关于在实践中隐喻是否有效的测试的具体反馈。
- ✧ 客户能够习惯用隐喻的形式谈论系统。
- ✧ 用重构来不断改善对实践中隐喻的含义的理解。

如果情况是这样，你或许可以仅从一个隐喻开始开发软件。

11.4 简单设计

你不能只是做足够今天的代码使用的设计。这样会把自己一路设计到死角里然后卡壳，没办法继续改进系统。除非：

- ✧ 你习惯于重构，因此做更改不是一件烦恼的事。
- ✧ 你有一个清楚的总体隐喻，因此确信未来的设计更改会趋向于不断收敛。
- ✧ 你在同一个伙伴编程，因此能确信所做的是简单的设计，而不是愚蠢的设计。

如果情况是这样，你或许就很有可能出色地完成今天而设计那样的

工作。

11.5 测试

你不可能编写出所有的测试。那要花费太多时间。程序员不愿意编写测试。除非：

- ✧ 设计已经尽可能的简单，因此编写测试根本不难。
- ✧ 你正和一个伙伴编程。如果你想不出另一个测试，你的伙伴能；如果你的伙伴想要放弃某个测试，你可以轻轻地夺过键盘。
- ✧ 当看到所有的测试都能运行时，感觉很棒。
- ✧ 你的客户看到他们的所有测试能够运行时，对系统相当满意。

如果情况是这样，那么程序员和客户或许就会编写测试。而且，如果你不编写自动化测试，XP 的其余部分也几乎不能正常运转。

11.6 重构

你不可能不停地重构系统的设计。那会花费太多时间，太难控制，并很可能会破坏系统。除非：

- ✧ 你习惯了集体所有制，因此不介意在任何需要的地方做改动。
- ✧ 你有编码标准，因此在重构前你无需重定格式。
- ✧ 你结对编程，因此更有勇气解决困难的重构，同时也更不容易造成损害。
- ✧ 你有简单的设计，因此重构更容易。
- ✧ 你有测试，因此不容易出现造成了破坏还不知道的情况。

- ✧ 你在持续集成，因此如果你偶然在某处破坏了什么东西，或者你的一个重构与别人的工作相冲突，你能够在几个小时内了解到这一点。
- ✧ 你休息得很好，因此更有勇气，并且出错的可能性更小。

如果情况是这样，那么你无论什么时候发现能简化系统、减少重复、或者更清晰地进行沟通的机会，你可能就会进行重构。

11.7 结对编程

你不可能在结对的情况下写出所有生产代码。那太慢了。如果两个人相处不来怎么办？除非：

- ✧ 编码标准能够消除无聊的争吵。
- ✧ 每个人都休息得很好，容光焕发，从而进一步减少了无用的——怎么说呢——“讨论”的可能性。
- ✧ 结对的伙伴一起编写测试，这样他们就有机会在解决实现的实际内容以前先沟通双方的构思。
- ✧ 结对伙伴可以用隐喻来作为命名和基本设计决策的基础。
- ✧ 结对伙伴进行的是简单的设计，因此他们都了解所从事的工作。

如果情况是这样，你或许就能以结对的方式编写所有的生产代码。而且，如果人们单独编程，特别是在压力之下，更有可能出错、做出超度设计和可能放弃其他实践。

11.8 集体所有制

你不可能让每个人在任何地方更改任何的内容。人们可能会犯这样那样的错误，而集成的成本会急剧上升。除非：

- ✧ 在足够短的时间内集成，这样就减小了冲突的可能性。
- ✧ 编写并且运行测试，从而减小了偶然造成破坏的可能性。
- ✧ 结对编程，这样破坏代码的可能性就更低，同时程序员能够更快地了解他们能对什么进行有益的更改。
- ✧ 坚持编码标准，这样你就不会卷入可怕的、一地鸡毛的混战中。

如果情况是这样，你就可能让任何人在发现可以改进系统的机会时，随时随地更改代码。而且，如果没有集体所有制，改进设计的速度会大大降低。

11.9 持续集成

你不可能在工作几小时后就集成。集成会花费太长时间，有太多冲突和偶然破坏代码的可能性。除非：

- ✧ 能很快地运行测试，从而知道没有破坏任何东西。
- ✧ 结对编程，这样要集成的更改量就少了一半。
- ✧ 进行重构，这样会有更多更小的组件，从而减小了冲突的可能性。

如果情况是这样，你或许就能够在几个小时后集成。而且，如果你不很快集成，冲突的可能性就会上升，集成的成本会陡然增高。

11.10 每周工作 40 小时

你也许能在每周 40 小时的情况下工作，但你在 40 小时内且无法创造足够的商业价值。除非：

- ✧ 计划游戏安排你做更有价值的工作。
- ✧ 计划游戏和测试的结合减少了经常出现的令人讨厌的意外情况。所以在意外情况下，要处理的问题会比你想象的多得多。
- ✧ 各种实践作为一个整体能够帮助你以最快的速度编程，没有比这样更快的编程速度了。

如果情况是这样，你或许就有可能在一周的 40 小时内创造足够的商业价值。而且，如果团队不能保持精力充沛和充满活力，你就不能执行其他实践。

11.11 现场客户

你不可能让真正的客户全职待在团队中。他们在别处可以为业务创造高得多的价值。除非：

- ✧ 他们能通过写功能测试为项目创造价值。
- ✧ 他们能够为程序员做出小规模的优先级和范围的决策，从而为项目创造价值。

如果情况是这样，他们或许就有可能对项目做出贡献，从而为公司创造出更多价值。而且，如果团队中不包括一名客户，他们将必须过早地安排计划，在没有确切地了解到必须满足什么测试和可以忽略什么测试之前进行编码，这就增加了项目的风险。

11.12 编码标准

你不可能让团队在一个共同的标准下编码。程序员都非常有个性，他们宁愿退出，也不愿意把大括号放在其他位置。除非：

◇ 整个 XP 使他们更有可能成为一个成功在即的团队的成员。

如果情况是这样，他们或许会愿意在自己的风格上稍稍灵活一点。而且，如果没有编码标准，不必要的磨擦会显著降低结对编程和重构的效率。

11.13 结论

任何一种实践都难以独当重任（测试可能是个例外）。它们需要其他实践来帮助保持平衡。图 4 是一个总结这些实践的图表。两个实践之间的连线表示这两个练习互相加强。我不想在一开始的时候就给出这幅图，因为它会使 XP 看起来很复杂。每个单独的部分都很简单。而各部分之间的相互作用则使得整个系统变得丰富起来。

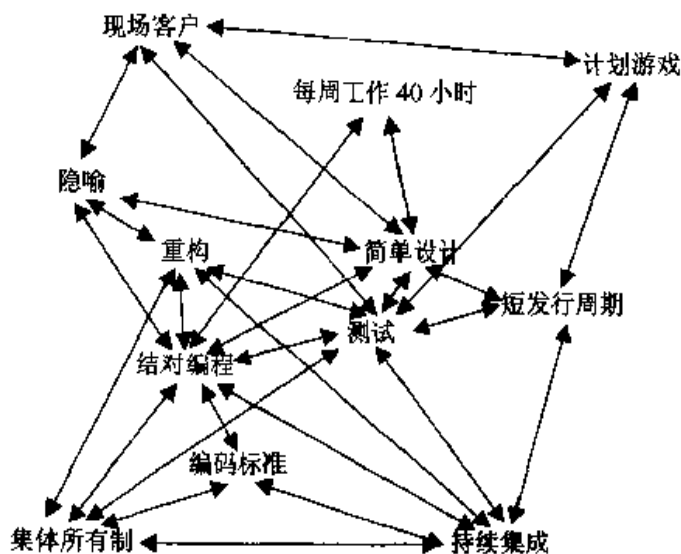
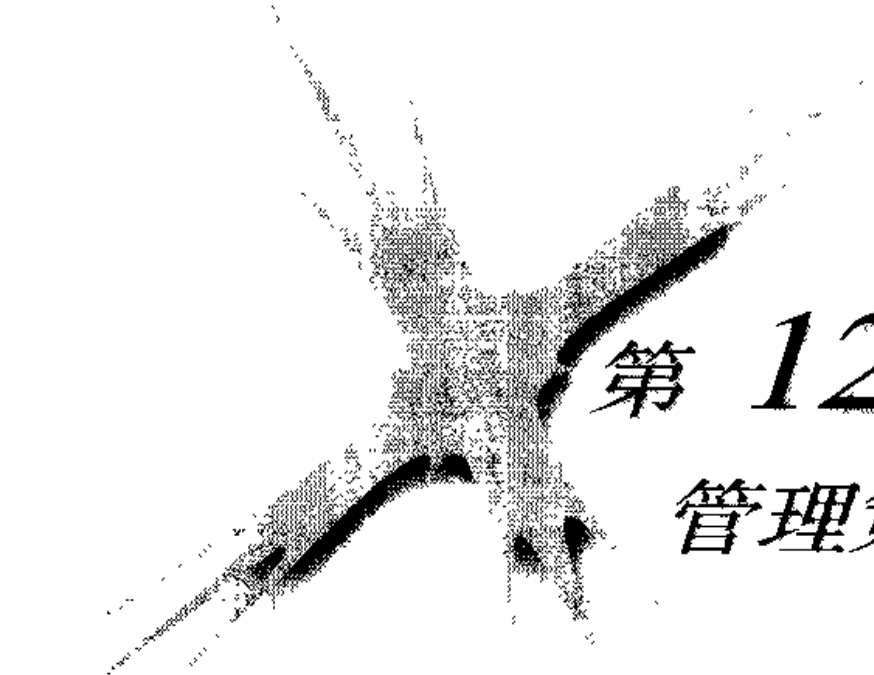


图 4 实践之间互相支持



第 12 章

管理策略

我们使用商业基本要素全面地管理项目，这些要素包括：分阶段交付，进行迅速和具体的反馈，清晰地阐述系统的业务要求和为特殊任务配备专家。

管理上的两难之处：一方面，你希望管理人员能做出所有决策。因为只有一个人，这样就消除了沟通的成本。只有一个人负责高层管理。只有一个人制定远景。不需要别人了解它，因为所有决定都来自一个人。

我们知道这种策略不可行，因为没有任何个人能有足够的知识来出色地完成所有的决策。倾向于集中控制的管理策略执行起来也很难，原因是被管理的人会造成大量系统开销。

另一方面，相反的策略也不行。你不能让每个人随心所欲，在没有任何监督的情况下干自己想干的事情。无一例外，人们会偏离正轨。需要有一个人来把握项目的全局，这个人能在项目要偏离轨道时施加他的影响。

于是，我们又回到了帮助我们在这两个极端之间权衡的原则上。

- ◇ 接受的职责——这意味着管理人员的任务是强调需要做的事情，而不是布置工作。
- ◇ 优质工作——这意味着管理人员和程序员之间的关系要建立在信

任的基础上，因为程序员希望能做出优秀的工作。另一方面，这并不意味着管理人员什么也不干。不管怎么说，在“我在想办法让这些人干的活说得过去”和“我要帮助这些人把工作干得更好”之间有很大的差别。

- ◆ 递增更改——这意味着管理人员要自始至终提供指导，而不是在开始时拿出一本厚厚的指导手册。
- ◆ 本地适应——这意味着管理人员要带头将 XP 应用到具体情况中，了解 XP 文化与公司文化的冲突，并找到解决它的方法。
- ◆ 轻装上阵——这意味着管理人员不应强加太多系统开销，比如冗长的全体人员会议和进度报告。无论管理人员要求程序员做什么事情，完成起来都不应该需要花费很多时间。
- ◆ 诚实的度量——这意味着无论管理人员使用的度量标准是什么，都应该在实际精确度的水平上。如果你的手表只有分针，那就不要试图用秒来计算。

从这种准则体系得出的策略更像是分散式的决策而不是集中控制。管理人员的任务是：运行计划游戏，进行度量，确保工作被度量的人员了解所做的度量，以及偶尔在无法以分布式的方式解决的情况下实施干预。

12.1 度量

XP 的基本管理工具是度量 (metric)。例如，估计的开发时间同日程表时间的比率是运行计划游戏的基本度量。它使得团队能够设定项目速度。如果该比率上升（即对于给定的预期开发量，花费了更少日程表时间），则可能意味着团队的工作进行得很好。否则，可能意味着团队除了完成要求（比如重构和结对编程）外工作完成得不够好，这样就要付出长远代价。

测量的方法是“大的可视图表”：管理人员周期性地（不少于每周一次）更新一个显眼的图表，而不是向每个人发送大家已经学会不理睬的电子邮件。通常这就是所需的全部干涉。你认为所编写的测试不够吗？那么贴一张测试数量的图表，每天更新它。

不要使用太多度量，准备好撤掉那些已经达到目的的度量。正常情况下，一个团队同时最多能忍受三到四种度量。

随着时间的推进，度量会变得失去意义，特别是，任何接近 100% 的度量可能很快就要失去意义了。对于必须为 100% 的单元测试来说，这条建议并不适用。但是，单元测试更像是一个假设，而不是一个度量。然而，你不能指望 97% 的功能测试成绩意味着你还有 3% 的工作可做。如果一个度量接近 100%，那么用另一个一位数的度量轻松地取代它。

这并不是建议你“用数字”管理 XP 项目。相反，数字是一种交流变化的必要性的温和和非强制性的方法。对于 XP 管理人员来说，衡量变化所需的最敏感的晴雨表就是他（或她）自己的感觉——身体上和情绪上的感觉。如果在早上钻进车里的时候，你胃痛得厉害，那么你的项目一定是出了问题，需要你做出一些改动。

12.2 指导

在 XP 中，大多数人心目中的管理者分为两个角色：教练和跟踪者（这两个角色可能由，也可能不由同一个人扮演）。教练主要关心的是过程中的技术执行（和改进）部分。理想的教练是一个好的沟通者，不易恐慌，技术过硬（虽然这一点不是一个绝对要求），很自信。通常，你会希望对教练这个位置使用那些在其他团队中是首席程序员或者系统构架设计师的人。不过，XP 中教练的角色有很大不同。

“首席程序员”和“系统构架设计师”这样的词语会让人想起这样的画

面：某些独立工作的天才做出项目中重要决策。教练则恰恰相反。衡量一个教练的标准是他或她做出了多么少的技术性决策。他的任务是所有其他的人做出正确的决策。

教练不会负责许多开发任务。相反，他工作的职责是这样的：

- ✧ 充当开发伙伴，特别是对于那些刚开始承担责任的新程序员或者困难的技术任务来说。
- ✧ 明白长期的重构目标，鼓励小规模的重构来实现这些目标的一部分。
- ✧ 用个人技术技巧帮助程序员，如测试、格式和重构。
- ✧ 向上层管理人员解释过程。

但是，教练最重要的工作可能是采购玩具和食物。XP 项目好像对玩具很有吸引力。其中许多是那些横向思维专家到处向人推荐的普通益智玩具。但是，教练会不时通过买了合适的玩具而对开发产生深远的影响。保持这种可能性是教练的最大责任之一。例如，在 Chrysler C3 项目中，设计会议进行了几个小时也没有结果。因此我买了一个普通的厨用计时器，然后规定所有设计会议都不能超过十分钟。我不相信有谁用过这个计时器，但是可以确信，当讨论失去意义而成为了一个逃避通过编码来得到确凿答案的过程时，它这种可视的存在提醒了大家。

食物也是 XP 项目的一个特色。通过和人一起吃东西，你能做到一些很了不起的事情。如果你在咀嚼的同时和别人讨论，那么效果一定全然不同。因此，在 XP 项目里总是到处堆满了吃的。（如果你能找到 Frigor Noir 巧克力条，我要特别推荐它们。但是一些项目好像要靠 Twizzler 甘草棒来维持。也非常欢迎你开发自己的本地菜单。）

Rob Mee 写道：

你知道，这些测试是很阴险的。在我的团队里，我们用食物和饮料犒劳自己。2点45分：“如果我们在3点整之前又做到100%，我们会喝茶，来一次快餐。”当然，无论如何我们都会去吃快餐，即使测试要进行到3点15分。但是，我们在测试确实能运行前几乎从来不吃快餐——有了成就感做铺垫，短暂的休息就象一个小派对一样（来源：电子邮件）

12.3 跟踪

跟踪是XP中管理的另外一个重要组成部分。你可以做出所有想要的估算，但是如果不度量实际状况与预期的差异，你永远不会明白。

跟踪者的任务是搜集某个时刻所有正在跟踪的测量数据，确保团队知道实际度量的是什么（以及提醒所预期的或所希望的是什么）。

运行计划游戏是跟踪的一部分。跟踪者需要冷静地了解游戏规则，并要做好准备，即使是在很情绪化的环境（计划总是情绪化的）中，也要实施它们。

跟踪不应造成大量系统开销。如果搜集实际开发时间的人一天两次向程序员询问他们的状况，程序员会宁愿马上离开，也不愿面对打扰。相反，跟踪者应该尝试用尽量少的时间来高效完成工作。两周一次搜集实际的开发数据就足够了。更多的测量不会给出更好的结果。

12.4 干预

管理XP团队可不单是取油炸圈和扔飞盘那么简单。有时候，在慈爱和细心的管理人员照料下的团队所产生的急智实在是解决不了手头的问题。这时候，XP管理人员必须适时地介入，做出决定——即使是不受欢迎的决定——并将决定的后果分析清楚。

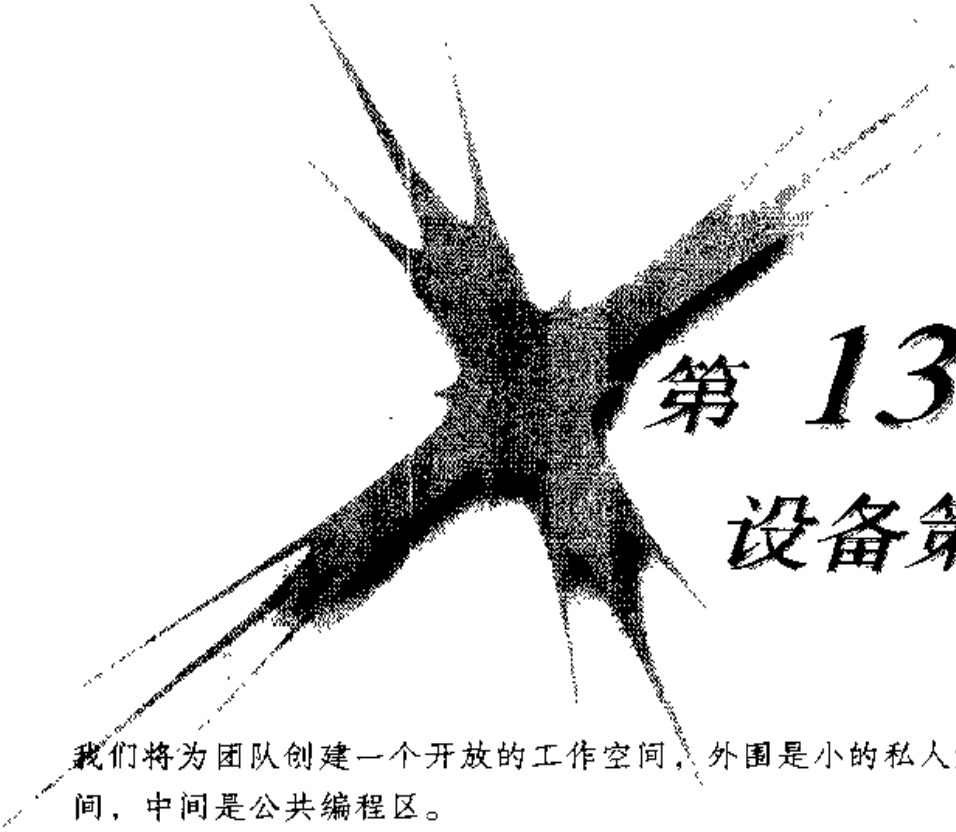
当然，管理人员必须首先仔细研究，查看是否原本应该注意或者处理

某些事情，从而完全避免问题。干预不是穿上盔甲跃马扬鞭。相反，干预是走到团队面前说：“我不知道我怎么把项目弄成现在这个样子的，但是现在我必须怎样怎样……”谦卑是干预时要遵循的规则。

人员变动是干预中一个非常重大的课题。如果团队成员总是解决不了问题，管理人员需要让他们离开。这样的决定赶早不赶晚。一旦你发现那个出错的家伙除了碍事以外起不了任何作用，就应该做出决定。等待只会让问题变得更糟。

一个稍微愉快些的职责是在团队的过程需要改变的时候进行干预。一般说来，指示要改变什么和怎样改变不是管理人员的职责，管理人员应该指出的是改变的必要性。团队应该提出一个或更多要运行的试验。然后，管理人员就回去报告试验所引起的度量出的变化。

管理人员最后的干预性职责是终止项目。团队可能从来不会自动退出。但是，当对目前的系统进行进一步工程投资比其他选项（比如启动一个替换项目）缺乏吸引力时，这一天就会到来。管理人员负责掌握什么时候已经达到了这种极限并通知管理上层改变的必要性。



第 13 章

设备策略

我们将为团队创建一个开放的工作空间，外围是小的私人空间，中间是公共编程区。

关于图 5，Ron Jeffries 写道：

这个图片显示了 DaimlerChrysler C3 “薪金册”小组的工作区。有两张大型工作桌，每张桌上放着六台开发计算机。程序员结对坐在任何能用来完成他们的工作的计算机前。（这张照片不是故作姿态拍出来的：实际上他们确实是这样工作的。摄影师和在后面那张桌前、背对着照相机的 Chet 一起工作。）

可以看到的两面墙上铺有白板，后面的那个板上写着需要注意的功能测试、计划的 CRC 会议以及迭代计划。左边的书写板顶部的纸张上有一些小符号，包含着工作组的 XP 规则。照相机的右边下方沿墙面并排着一些小隔间，大小刚好够一部电话和写字的地方。

房间的最后面，在计算机桌和白色书写板之间，有一张标准桌，供团队举行 CRC 会议时使用。这张桌上通常放着 CRC（Class/Responsibility/Collaborator，类/责任/合作者）卡片和一些食品，这是因为团队有这么一条规矩：“一定要有吃的”。

该房间是由团队自己设计的：实际上是我们“选择了”这里，

房间里静悄悄的，如果有人说话，声音都特别低，但是如果你需要帮助，只要稍微把声音提高一点就行了。你会立刻得到帮助：注意，地板上没有地毯，这意味着椅子真的能移动！



图 5 DaimlerChrysler C3 工作区

如果你没有合理的工作空间，你的项目不会成功。团队工作空间的好坏所造成的影响是直接和显著的。

有一次，我被邀请去审查一个项目的面向对象设计，那是我作为顾问的开发生涯中的一个转折点。我检查了那个系统，确认它糟糕透了。然后我注意到每个人的座位布置情况。那个团队有四个高级程序员。在一个中等大小的房间的四个角落，他们各有一间办公室。我告诉他们应该把办公室挪到一起。我是因为有 Smalltalk 和对象的知识被请过去的，而我给他们提供的最有价值的建议却是他们应该重新布置办公室。

在任何情况下，设备都是一个很难解决的工作。有很多冲突的约束条件。设备的筹划者想的是如何花钱最少，而又能维持最大的灵活性。使用

设备的人想的是能和团队的其他成员在紧密联系的情况下一起工作。同时，他们还需要有私人空间，以便能够做一些像打电话约医生之类的私事。

XP 宁可选择有太多的公共空间。它是一种团队软件开发准则。团队成员需要能够看见对方，能够听见别人突然喊出且只说一次的问题，能够“偶然”听到与他们自己息息相关的谈话。

XP 可使设备得到充分应用。一般的办公室布置对 XP 不合适。例如，把计算机放在角落里不行，因为这样就不可能让两个人并肩坐着编程。正常的小隔间隔板高度也不合适——隔板高度应为正常高度的一半或者干脆不要隔板。同时，不同的团队之间应该分开。

最好的布置是一个开放的大房间，其中外围有一些小隔间。团队成员可以在小隔间里放私人物品，可以在里面打电话，当不想被打扰时也可以到里面去坐。如果某位队员坐在小隔间里，团队的其他成员应尊重他的“虚拟”隐私权。把最大的、速度最快的开发计算机放在空间中央的桌子上（小隔间里可以有也可以没有计算机）。这样，如果有人要编程，他们自然而然地就走到开放的公共空间去了。在那里大家都可以看到工作的进展，程序员可以轻松地组对，而同时进行开发的各个组对之间可以互相促进。

如果可以的话，把最好的空间留一小部分出来作为公共空间，里面可以放咖啡壶、沙发、玩具、画具等。通常，在开发时如果想放松一下，最有效的方法是走开一会儿。如果有一个舒适的空间，需要时你会很愿意到那儿去放松一下。

在 XP 对设备的态度中很好地体现了勇气准则。如果老板和团队对设备的态度意见不一致，那么，团队说了算。如果摆放计算机的位置不对，移开。如果中间有碍事的东西，拿走。如果灯光太亮，换掉。如果电话铃声太响，那么有一天你会发现，神不知鬼不觉地，所有电话铃都被用棉花塞住了。

有一次，在我第一天到一家银行时发现，给我们使用的办公桌又旧又

难看，而且还是单人桌。办公桌的两边各有一组铁抽屉，中间剩下的地方刚够塞进一个人的双腿。很显然，那样不成。我们四处寻找，终于找到了一把工业用加力螺丝刀，然后卸掉了办公桌的一组抽屉。这样两个人就可以并肩坐在一张办公桌前了。

这样在办公设备上拧来拧去会惹来麻烦。如果设备管理人员发现有人未经他们允许或参与擅自挪动了办公桌，会很生气（不要去想做改动申请，那可能要等上数周或数月才能批准）。我说，“太糟糕了。”我要编软件，如果把障碍物拿掉能让我把软件编得更好，我就要那么做。如果组织不能容忍这种主动的行为，那么无论如何我也不想在这工作。

能够控制物理环境给了团队一个强有力的信息：他们不会让组织中无意义的权益纷争阻碍他们的成功。控制物理环境是他们在总体上控制如何工作的第一步。

应该对设备进行经常性的试验（这里运用了反馈准则）。毕竟，组织花了那么多钱购买所有这些可拆装的办公室设备。如果不发挥设备的灵活性，岂不是浪费了这些钱。让两个人的小隔间靠得近一些怎么样？远一点呢？把集成用的计算机放在中间怎么样？放在角落里呢？也可以试试。怎么合适就怎么来。如果不合适，就重新再来。



第 14 章

拆分业务责任和技术责任

我们的策略的一个关键点是让技术人员把精力集中在技术问题上，让业务人员把精力集中在业务问题上。项目必须由业务决策来驱动，而技术决策则要给业务决策提供有关成本和风险的信息。

在业务和开发之间的关系中，有两种常见的模式是失败的。业务方和开发方任何一方得到的权力过大，项目都会受到损害。

14.1 业务方

如果业务方有权力，他们会觉得可以向开发方指定所有四个变量。“这是你要完成的工作。这是完成工作的时间。不，你不能使用更多新工作站。而且，最好达到最高质量，否则你就有麻烦了，小子。”

在这种情况下，业务方总会指定太多东西。需求列表中的某些项绝对是重要的。但有些需求则不是那么重要。而且，如果开发方没有任何权力，那么他们不能反对；他们不能强迫业务方选择什么不选择什么。因此开发方只好低着头，顺从地去干分配给他们的不可能完成的任务。

不太重要的需求引起的风险最大，这似乎是个自然规律。通常他们是

最不为大家所理解的那部分需求，因此在开发过程中就有需求完全改变的巨大风险。出于某些原因，在技术上他们也会招致更大风险。

“业务方负责”方案的结果是，项目付出太多的努力，承担太大的风险，而得到的回报却太少。

14.2 开发方

现在，轮到开发方负责，你可能认为日子会好过些。但事实并非如此，最终结果和业务方掌权时完全一样。

当开发方负责时，他们会把所有从来没时间去做的过程和技术加进来——他们觉得需要“这些套件”。他们会安装新工具、新语言和新技术。而他们之所以选择那些工具、语言和技术，是因为它们有趣并且是前沿的。前沿意味着风险。（如果我们现在还没明白这一点，那么什么时候才能明白？）

因此，“开发方负责”方案的结果也是付出太多的努力，承担太多的风险，而得到的回报却太少。

14.3 怎么办

解决方案是，以某种方式在业务方和开发方之间拆分责任和权力。业务人员应该做适合他们来做的决策。程序员应该做适合程序员来做的决策。任何一方的决定都应通知另一方。任一方都不能单方面作任何决定。

要维持这种政治制衡方案看起来似乎不可能。既然联合国都无法做到，我们又怎么能做到呢？当然，如果你所想到的只是“平衡政治力量”这样的模糊目标，那你肯定做不到。一旦出现了强硬人物，平衡就会被打乱。幸运的是，我们的目标比这要具体得多。

首先我们要讲一个故事。如果有人问我，是想要法拉利还是小型货车，

我几乎肯定会选择法拉利。法拉利肯定会更有意思。但是，一旦有人问：“你想要 200000 法郎的法拉利还是 40000 法郎的小型货车？”这时我会开始做一个基于具体信息的决定：添加新的要求，如“要能够带五个孩子”或“时速要达到 200 公里/小时”，这样就能够理清楚地知道需要什么车了。有时候这两种决定都合情理，但是不可能只根据漂亮的照片来做出正确的决定。你还要知道你具有的什么样的资源、要受什么约束以及各自的成本。

按照此模型，业务人员应选择：

- ✧ 发布的范围或时间。
- ✧ 提出的功能的相对优先权。
- ✧ 提出的功能的确切范围。

对于这些决定开发组织必须确定：

- ✧ 实现各种功能所需的时间估算。
- ✧ 各种可选的技术方案的后果估算。
- ✧ 适合他们的个性、业务环境和公司文化的开发过程。没有一个“你要这样编写软件”的列表可能适用于所有情况。实际上，不可能有适合任何情况的一个列表，原因是情况总是在变化。
- ✧ 使用哪组实践来开始，以及使用什么样的进程来评审实践的效果和对变化进行试验。这有点像美国宪法，它给出基本的价值体系，一组基本的规则（权利和自由法案，前十条修正案），和用于更改规则的规则（添加新的修正案）。

由于业务决策在整个项目周期中不断发生，因此，让业务人员负责业务决策意味着客户与程序员一样是 XP 团队的一部分。特别是，他们最好

能够与团队的其他成员坐在一起并专职负责回答问题。

14.4 技术的选择

虽然技术的选择初看上去像是一个纯粹的技术决策，但实际上它是一个业务决策，而且是一个开发方必须参与的业务决策。客户将必须与数据库或语言供应商打多年交道，并且必须能同样处理好业务级别和技术级别的关系。

如果客户对我说：“我们想要这个系统，你必须使用这个关系数据库和那个 Java IDE”，那么我的工作就是指出这个决定的后果。如果我认为对象数据库和 C++ 更适合，我会从两方面对项目进行评估。然后业务人员就可以作出业务决策。

但是，技术决定还有另一面，即纯粹属于开发阵营的技术决策。一旦公司引进了某种技术，就必须在该技术的使用期间保证它起作用。最新和最好的技术的成本不只在初期的生产开发阶段中，甚至不只在整个开发阶段中。这些成本必须包括使技术发挥作用的创建和维护的成本。

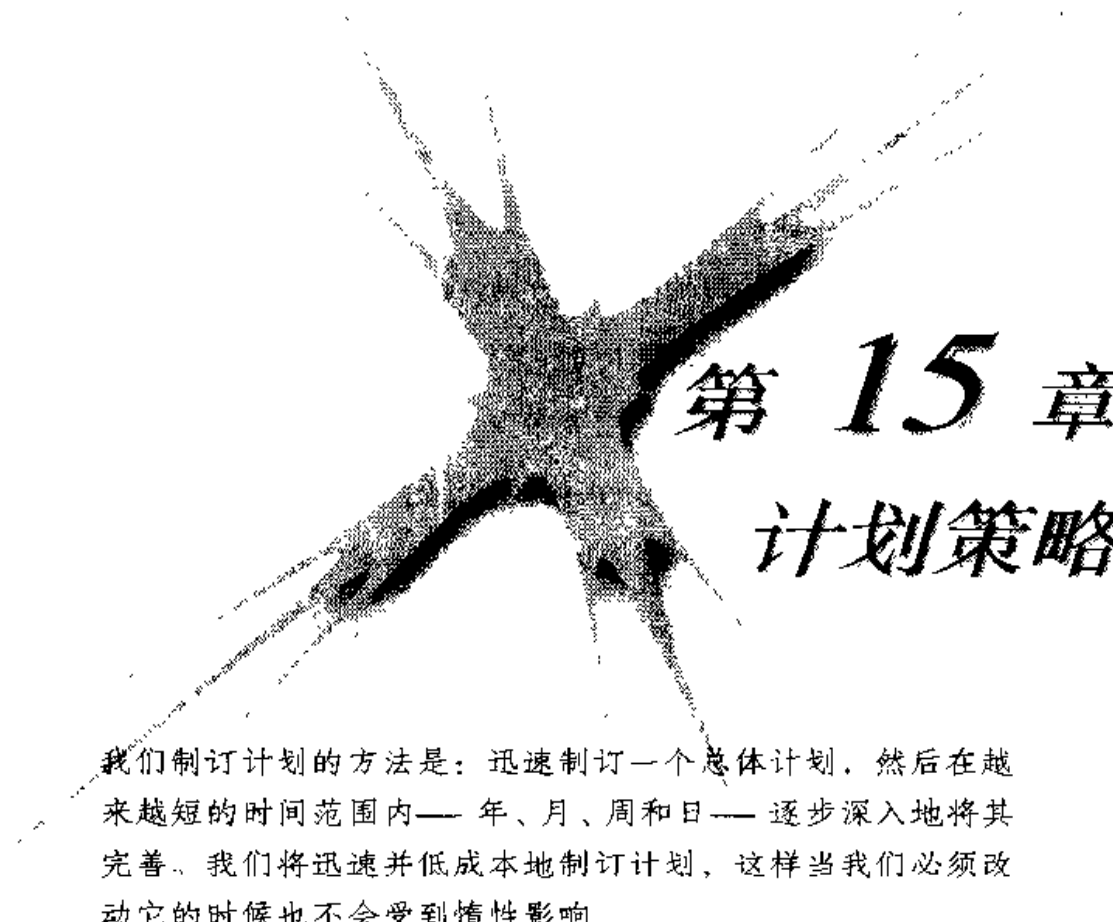
14.5 如果这很难做到又该怎么办

大多数时候，用此过程做出的决定实现起来出乎意料地简单。程序员很善于发现每个故事中暗藏的危险。业务人员会说：“想不到会这么贵。只做这三分之一就行了。现阶段这么做就够了。”

但是，有时事情并非如此。有时，程序员看来最小和最有价值的开发部分实际上却是庞大且最有风险的部分。发生这种情况时，你不能无视它的存在。你必须要认真对待。你可以承受几个错误。你可能必须投入更多资源。但是，过了这道坎后，你就能真正赚钱了。要想方设法缩小范围。要

尽可能降低风险。然后你就可以轻松地赚钱了。

换一种说法，业务方和开发方之间的权力拆分不是回避艰苦工作的借口。事实正相反。它是一种方法，用于从你还不知如何使之变简单的工作中区别出真正棘手的工作。大多数时候，工作比你开始想象的要简单。即使工作不是那么简单，你还是得做，因为那正是出钱雇你的原因。



第 15 章

计划策略

我们制订计划的方法是：迅速制订一个总体计划，然后在越来越短的时间范围内——年、月、周和日——逐步深入地将其完善。我们将迅速并低成本地制订计划，这样当我们必须改动它的时候也不会受到惰性影响。

制订计划就是猜测为客户开发一套软件的情况会是什么样的过程。制订计划的几个目的是：

- ◇ 团结和组织开发团队。
- ◇ 决定范围和优先级。
- ◇ 估算成本和日程。
- ◇ 让大家对系统的成功信心百倍。
- ◇ 为反馈提供一个基准。

我们先来审视一下影响制订计划的一些原则。（其中一些是第 8 章“基本原则”中提到的一般原则。其他的是专门针对于制订计划的。）

- ◇ 只制订下一个阶段所需的计划——在任何详细级别，只把计划制

订到下一阶段——如下一个版本、下一次迭代的结束。这不是说不能制订长期计划。你可以这么做，只是不要过于详细，你可以将一个版本的计划制订得十分具体，而用一组项目符号来制订后面五个（建议）版本的计划。较之于将六个版本全都制订得非常详细，这样的简略不会损失多少

- ✧ 接受的责任——责任只能被接受，而不能被强加。这意味着 XP 中没有所谓的自上而下的规划。管理人员不会跑到开发团队那里去说：“现在我们有一大堆活，必须在这么长时间内干完。”项目经理必须要求开发团队承担起工作的责任，然后等待答复。
- ✧ 负责实现的人进行估算——如果开发团队承担完成某件事情的责任，他们就要确定需要多长时间。如果开发团队中的某个人要承担完成某项工作的责任，他们也要说出需要多长时间。
- ✧ 忽略各部分之间的依赖关系——制订计划的时候就假设开发的各部分可以随意互换。只要你足够细心，首先去实现最高优先级的业务，这个简单的规则可以保证你高枕无忧。“咖啡多少钱？”“25 美分一杯，续杯免费。”“那么只给我一个续杯。”这种情况好像不太可能发生。
- ✧ 为优先级作计划与为开发作计划的比较——谨记计划的目的是使客户能够建立优先级的计划可以比为实现而制订的计划简略得多，后者还需要有具体的测试用例。

15.1 计划游戏

XP 的计划制订有意将计划过程抽象为两个参与者——业务方和开发方。这有助于在讨论计划过程中消除一些无益的情绪躁动。计划游戏中没有“Joe，你这个白痴，你说过会在星期五前交给我”这样的字眼，相反，

计划游戏会说“开发方学到了些东西，需要从业务方那里得到最佳的反应。”一套简单的规则不可能消除情绪，当然，它也没想这样做。这些规则是用来提醒大家该如何工作的，而在事情进展得不顺利的时候，它们能够提供一个共同的参考。

业务方往往不喜欢开发方。需要系统的人和生成系统的人之间的关系总是很紧张，就像是几个世纪的宿敌一样。猜疑、谴责以及微妙和间接的勾心斗角随处可见。在这种环境中不可能开发出像样的软件。

如果你所处的环境不是这样，恭喜你。最好的环境就是互相信任，双方互相尊重。双方都相信对方会把自己的最高利益和整个团体的利益放在心中。双方都愿意让对方来做他们的工作，切磋彼此的技巧、经验及看法。

这种关系是无法通过强制手段来实现的。你不能只在那里说：“我们知道自己把事情搞糟了。我们非常的抱歉。这种事不会再发生了。我们分开工作吧，就从午饭后开始。”地球不是这么转的。人们在压力之下往往会返回到以前的行为，不管那种行为过去曾带来多坏的后果。

建立互相尊重的关系需要一套规则来支配如何引导这些关系——由谁来做哪些决定，什么时候做出决定，以及如何记录这些决定。

但是，别忘了游戏的规则只是辅助性的，只是迈向与客户之间的理想的关系的一个步骤。这套规则永远无法战胜真正的人与人的关系的那种微妙、灵活性和热情。但是，如果没有一套规则，你可能都无法开始改善境况。一旦规则发挥了作用，关系得到了改善，那么你就可以进一步修改规则，让开发进行得更顺利。最后，当规则成为习惯后，就可以把它们完全置之脑后了。

但是，你首先要学会按章行事。请往下读。

15.1.1 目标

计划游戏的目标是让开发团队生产的软件的价值得到最大化。软件的

价值中必须扣除开发的成本，以及开发过程中所带来的风险。

15.1.2 策略

开发团队的策略是尽可能少地投入，并尽可能快地将最有价值的功能投入生产，但这只能在结合了为降低风险而设计的编程和设计策略的情况下实现。得益于第一个系统的技术和业务教训，业务方会很清楚地知道什么是最有价值的功能，而开发团队将迅速将其投入生产。诸如此类。

15.1.3 牌

计划游戏中的牌是故事卡。图 6 给出了一个例子。

15.1.4 玩家

计划游戏中的两个玩家是开发方和业务方。开发方由所有负责实现系统的人共同组成。业务方由所有负责决定系统功能的人共同组成。

Customer Story and Task Card		B/W Development / COLA	
DATE: 3/19/91	TYPE OF ACTIVITY: NEW: <input checked="" type="checkbox"/> FIX: <input type="checkbox"/> ENHANCE: <input type="checkbox"/> FUNC. TEST: <input type="checkbox"/>		
STORY NUMBER: 1275	PRIORITY: USER: <input type="checkbox"/> TECH: <input type="checkbox"/>		
PRIOR REFERENCE: <input type="text"/>	RISK: <input type="text"/> TECH ESTIMATE: <input type="text"/>		
TASK DESCRIPTION: SPLIT COLA: When the COLA rate changes in the middle of the B/W Pay Period, we will want to pay the 1 st week of the pay period at the OLD COLA rate and the 2 nd week of the pay period at the NEW COLA rate. Should occur automatically based on system design.			
NOTES: For the OT, we will run a m/frame program that will pay or calc the COLA on the 2 nd week of OT. The plant currently retransmits the hours data for the 2 nd week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA.			
TASK TRACKING: Gross Pay Adjustment, Create RM Boundary and Place in DE Ent Gross COLA			
Date	Status	To Do	Comments

图 6 故事卡

有时很容易就能看出谁在计划游戏中扮演业务方的角色。如果一个证券经纪人花钱开发一款定制的软件，那么他就是业务方。他来决定什么最重要，应该最先完成。如果你在开发一个面向大范围市场的套装软件产品呢？谁又是业务方？业务方要为范围、优先级和版本内容作出决定。这些决定通常是由市场部门来作的。如果他们足够聪明，他们就会通过指出以下内容来支持自己的决定：

- ◆ 产品的真正用户。
- ◆ 目标群体。
- ◆ 销售人员。

我所见过的一些最好的业务方玩家是专家级用户。例如，我曾为一个共有基金的客户服务系统工作过。业务方是由一位在前一个系统上工作了多年的客户服务主管担任的，她知道那个系统的每一个细节。她常常搞不明白新系统和旧系统的功能该有什么不同，但当她熟悉了那些故事之后，她就明白了。

15.1.5 步骤

游戏分三个阶段。

- ◆ 探索——找出系统有哪些新的功能。
- ◆ 承诺——确定以后要完成所有可能的需求中的哪些部分。
- ◆ 指导——像现实塑造计划那样指导开发过程。

每个阶段的步骤通常都是在本阶段内完成的，但这并不是绝对的。例如，可以在指导阶段编写新故事。这些阶段也是循环的；当你经过了一段

时间的指导后，你可能会需要回到探索阶段。

1. 探索阶段

探索阶段的目的是让两个玩家都对系统最终将能完成什么功能有一个了解。探索共分三步。

编写一个故事——业务方编写故事来描述系统要做的事。故事写在索引卡上，有一个名称和一小段说明故事的目的的文字。

估算时间——开发方要估算实现故事需要多长时间。如果开发方无法对故事进行估算，可以要求业务方将其解释清楚或分解。评估故事最简单的方法就是问自己：“如果这个故事由我来实现，在没有任何干扰或会议的情况下，我会需要多长时间呢？”在 XP 中，我们把它叫做“理想工程时间”。你将会在下面看到（在“设置速度”中），在对日程做出承诺之前，你要先测量一下理想时间和日历之间的比率。

分割时间——如果开发方不能评估整个故事，或者如果业务方认识到故事中的某个部分比其他部分更重要，业务方可以让开发方把故事分割为两个或多个故事。

2. 承诺阶段

承诺阶段的目的是为了让业务方选择下一个版本的范围和日期，并让开发方有信心地做出交付的承诺。承诺阶段分为四步。

按价值进行分类——业务方将故事分为三部分：（1）没有它们系统就无法运行的部分，（2）稍为次要些但有很高的商业价值的部分，（3）如果有则更好的部分。

按风险分类——开发方将故事分为三部分：（1）可以精确估算的，（2）可以进行合理估算的，（3）完全无法估算的。

设置速度——开发方告诉业务方使用“理想工程时间”计算的话，开发团队每个月能干得多快。

选择范围——业务方选择一个版本中的卡片集，有两种方式：设置工程结束日期并根据估算和项目速度来选择卡片，或者选择卡片然后计算工期。

3. 指导阶段

指导阶段旨在根据开发方和业务方掌握的情况来更新计划。指导阶段分为四步。

迭代——每次迭代的开始时（每一到三周），业务方选择一个要实现的最有价值的故事。第一次迭代的故事必须能产生一个可以端到端地运行的系统，无论它多么不成熟。

恢复——如果开发方意识到高估了自己的速度，他可以向业务方询问哪组故事最有价值，以便根据新的速度和估算在当前版本中把它们保留下来。

新故事——如果业务方在开发一个版本的过程中意识到自己需要新的故事，可以编写一个新故事。开发方将对故事进行评估，然后业务方从剩余计划中删除具有同等估算时间的故事，并插入新故事。

重新估算——开发方觉得计划已经不能准确地描述开发的状况，它可以重新估算其余故事并重新设置速度。

15.2 迭代计划

以上的计划游戏使客户能够每隔三周对开发进行指导。在每次迭代中，开发团队用差不多相同的规则来计划他们的活动。

迭代计划游戏与计划游戏很类似，因为它们都用卡片来当作牌。但这次牌是任务卡，而不是故事卡。玩家是所有的单个程序员。时间上规模更小一些——整个游戏在一次迭代中就完成了（一至四周）。阶段和步骤都类似。

1. 探索阶段

编写任务——把故事安排在迭代期中并转变为任务。通常，任务是小于整个故事的，原因是你无法在短短几天内实现整个故事。有时一个任务能够支持数个故事。有时一个任务不会直接关联到任何特定的故事——例如，迁移到系统软件的一个新版本。图 7 就是一个真实任务卡的例子。

Engineering Task Card

DATE: 3/17/98 BIW Smalltalk/Future
Based on Conversation w/REB:AMA **NEW**

STORY NUMBER: X923 SOFTWARE ENGINEER: _____ TASK ESTIMATE: _____

TASK DESCRIPTION:
Composite Bin - Regular Base Needs to Be Displayed on GUI. We have the hidden bin for Regular Base (Last Time) to display NOT the auto gen bin but the BIW that composites the Auto Pay the Last Time. There is

SOFTWARE ENGINEER'S NOTES: a separate composite bin started that needs to be completed??

TASK TRACKING:

Date	Done	To Do	Comments

图 7 任务卡

分割任务/组合任务——如果估算某个任务时发现无法在几天之内完成它，就把它分割为几个小任务。如果有几个任务分别只需要一小时即可完成，就将它们组合成一个较大的任务。

2. 承诺阶段

接受任务——程序员承担起完成一个任务的责任。

估算任务——负责的程序员估算实现每个任务所需的理想工程天数。通常这要倚仗从另一个对要修改的代码更为熟悉的程序员那里得到的帮助。要花好多天才能完成的任务必须分割（必须通过比较按时完成与未能

按时完成的任务来找到自己的准确阈值)。

你可能认为自己必须在估算中明确地把成对编程的效果计算在内。不要管它。用来帮助其他程序员、与客户讨论和参加会议的时间已经包含在负载因子 (load factor) 中了。

设置负载因子——每个程序员都要为迭代选择负载因子——他们实际花在开发上的时间的百分比。这是一个经测定后才能得到的数字——理想编程天数与日历的关系。如果你在最近的三次迭代中分别完成了相当于 6、8 和 7.5 个理想天数的任务，那么这就是在这个迭代中你应该承担的大约数目。每次迭代中任务的理想天数对于新团队成员或教练来说可能很少——在一个三周的迭代中只有两三天。对任何人来说，这个数目都不应该高于七八天，否则他们将没有足够的时间来帮助别人。

平衡——每个程序员将他们的任务估算进行合计，再乘以他们的负载因子。负担过重的程序员必须放弃一些任务。如果整个开发团队都负担太重，他们就必须找出一种恢复平衡的办法（参见下面的“恢复”）。

3. 指导阶段

实现任务——程序员领取任务卡，找到伙伴，编写任务的测试用例，使它们全部能运行，然后，当全部测试套件都能运行后，就集成并发布新的代码。

记录进度——每隔两三天团队中的一个成员就会询问每个程序员他们在自己的每个任务上花费了多少时间，还剩下多少时间。

恢复——证明是承担了过多任务的程序员可以通过以下几种方式来求助：（1）缩小一些任务的范围，（2）请求客户缩小一些故事的范围，（3）放弃不重要的任务，（4）得到更多或更有效的帮助，（5）最后一招，请求客户将一些故事延期到稍后的迭代中。

确认故事——一旦功能测试准备就绪且故事的任务已经完成，就会运

行功能测试来确认故事是否能成功运行。实现过程中出现的一些有用的情况也可以添加到功能测试集中。

迭代计划和发布计划之间的差别主要在于：迭代的日程中比承诺的日程中允许更多摇摆。如果一个三周的迭代进行了一周时进度很慢，那么，完全可以停下一天来进行协作的重构，而这是每个人的前进都需要的。没有一个程序员会觉得整个项目会在那时崩溃掉（有了经验后他就不会这么觉得了）。但是如果客户看到一天内发生如此重大的更改，他们一定会紧张起来。

从某种意义上讲，这样做看起来像撒谎，因为你在向客户隐瞒开发过程中的一部分。一定要记住，不能让客户这么想。你并不是故意要隐瞒什么。如果客户想旁听这一整天的重构，可以，虽然他们可能有更要紧的事要做，但只要他们想参加，我们欢迎。迭代间和迭代内计划的区别在于扩展了分割业务和技术决策的原则。某个细化级别上的更改将不再是业务方该考虑的事情——程序员比任何业务人员都更清楚该如何微观管理自己的时间。

计划游戏和迭代计划游戏之间的一个区别是程序员在估算之前就承担任务。开发团队无形中承担了所有故事的总体责任，因此估算应该由开发团队来集体完成。单个程序员接受的是任务的责任，因此他们必须自己对任务进行估算。

与迭代计划之间的另一个区别是：有些任务不是直接与客户的需求相关的。如果有人想要加强集成工具，而这项任务又大到无法在开发过程的间隙内完成，那么它就变成了一个独立的任务，要与所有其他任务一起排在日程内并划分优先级别。

我们来回顾一下迭代计划过程要受到的一些约束，以及上面的策略如何能满足它们。

- ◇ 你并不想花太多的时间来制订计划，因为计划总是没有变化快。从十五天中取出半天时间不算花销太大。当然，如果你能花更少一点的时间，那就更好了，不过半天也不算太多。
- ◇ 你想很快得到关于自己的工作的反馈，这样就能在三分之一的日程内知道是否出了错误。负责跟踪进度的人员提出的问题答案会让你在迭代进行了一半的时候对你是否能按日程完成进度做到心中有数。通常这样就会有足够的时间来对问题进行内部响应，而无需请求客户做出更改。
- ◇ 你希望负责交付的人也负责估算。只要程序员在估算前承担责任，这就有效。
- ◇ 你想把开发的范围限制到实际所需要的内容。如果有人说他在三周的时间里只能做 7.5 天的工作（15 天除以测得的负载因子 2），大家一定都会觉得奇怪。但是，当你的估算变得越来越好，你会发现他说的完全正确。如果你觉得自己并没有真的忙得不可开交，你就会想多做一些任务。但是你知道与此同时你必须保证任务的标准和质量（而且你的伙伴正在盯着同一个屏幕，确保你能保持质量）。因此，你会希望做一些简单的工作，并能诚实地说自己完成了任务。
- ◇ 你不想要一个这样的工作进程：它带来如此大的压力，以致于人们只为了满足短期计划的要求而做一些傻事。于是，又回到了能完成 7.5 天工作的说法。你就是不能承担太多的任务。如果你去做一个迭代，会有大量的公开反馈告诉你不该尝试去做那么多。下次你就不会再这样做了。这将让你量力而行，并能保证质量。

对于较小的项目，就不需要迭代计划了。当然，它对协调十个程序员的工作来说是很必要的。对两个人的工作来说就不是那么必要了。根据项

目的不同，你将会发现如果有协调工作的需要，那么正规迭代计划带来的额外工作还是值得的。

15.3 在一周内做计划

如果只有一周时间，你会如何计划项目呢？这种情况常常出现在团队被要求作固定价格投标的时候。你得到一份标书，而做出答复的时间只有一周，你没有足够的时间来编写一整套故事，每一个故事都去估算和测试。也没有时间编写原型，以便你能够根据经验对各种情况进行估算。

XP 的解决方案是通过编写更大的故事，从而在计划中承担更多风险。以理想编程月份为单位（而不是理想编程星期）来编写你能够估算的故事。你可以给客户通过缩减或推迟一些故事来进行权衡的机会，就像你在通常的计划游戏中做的那样。

估算应该根据经验来进行。与计划游戏不同，在一周之内对一个建议做出响应，你没有足够的时间来得到很多经验。你应该根据以往编写类似系统的经验来进行估算。如果没有编写类似系统的经验，就不要对固定价格标书做业务响应。

拿到合同后，要做的第一件事就是回去开始启动计划游戏，这样你就能够马上对自己按合同交货的能力进行反复查对。



第 16 章

开发策略

与管理策略不同，开发策略从根本上背离了传统的观念——我们会认真制订今天的问题的解决方案，并相信我们能够在明天解决明天的问题。

XP 对它的各种任务使用了编程的隐喻——即，某种意义上说，你所做的所有事情看起来都像是编程：XP 中的编程与编程相似，只是增添了一些小东西（如自动化测试）。但是，与 XP 的其他部分一样，XP 开发只是看起来简单。所有部分解释起来都很简单，但是执行起来却很困难。这时会出现焦虑。而在压力下，人们会返回到以前的习惯。

开发策略从迭代计划开始。持续集成减少了开发冲突并为开发过程创造了一个自然而然的结尾。集体所有权鼓励整个团队改善整个系统。最后，结对编程将整个过程联系在一起。

16.1 持续集成

任何代码都要在几个小时之内进行集成。在每个开发过程的结尾，代码将与最新的版本进行集成，而所有测试都必须能 100% 运行。

在持续集成的外部限制下，每次更改方法时，所做的更改将立即被反

映到其余所有人的代码中。没有支持这种风格所需的体系结构和带宽，它就不能很好地工作。开发时，你希望假想你是该项目中的唯一程序员。你希望能全速向前行进，而不用理睬你所做的更改与其他人恰好正在做的更改之间的关系。在你的直接控制外发生的更改将使这种幻觉破灭。

几个小时（一定不超过一天）就集成一次能提供这两种风格（即单个程序员和即时集成）的许多好处。开发时，你可以假想你和你的伙伴是该项目中仅有的一对。你可以在任何地方做出更改。然后你要转换角色。作为集成者，你将注意到（工具会告诉你）类或方法的定义当中何处存在冲突。通过运行测试，你将会了解到语义的冲突。

如果集成花了若干小时，就不能用此方式进行工作。拥有支持快速集成/生成/测试循环的工具是很重要的。你还需要一个能在几分钟内就能运行的相当完整的测试套件。解决冲突所需的工作不能太多。

这不是问题。经常性的重构会将系统分割为许多小对象和小方法。这降低了两对程序员同时更改同一个类或方法的可能性。即使他们这样做了，对所做的更改进行折衷所需的工作量也会很少，这是因为每一种更改仅需要几个小时的开发。

接受持续集成的成本的另一个重要原因是它会极大地降低项目的风险。如果两个人对一段代码的外观或操作有不同意见，你将会在几个小时内知道。你永远不需要花费好几天来找一个在几周前的什么时间出现的错误。到了创建最终项目的时候，集成时进行的所有实践就会派上用场。“生产期生成”不是什么人不了的事情。团队中的每个人都能在那时候边睡觉边完成它，这是因为几个月以来他们一直在做这个。

持续集成还在开发期间为开发者提供了一个颇有价值的优点。在完成一项任务时，会有数以百计的事情需要你操心。通过一直工作到一个自然而然的间断——待办事项卡上不再有小项目了——然后集成，你就赋予了开发过程一个节奏。学习/测试/编码/发布。这几乎与呼吸一样。形成一个想

法，表达它，将它添加到系统中。现在你的头脑很清楚，开始接着构想下一个主意。

一次又一次，持续集成迫使你任务的实现分为两个过程。我们接受这种分割所造成的开销，并牢记已经做了什么和接下来需要做什么。在过渡时期中，你可能会了解到是什么原因导致第一个过程进行得如此缓慢。进行了一些重构后，你开始下一个过程，第二个过程的剩余部分将进行得顺利得多。

16.2 集体所有权

集体所有权（即任何人可以随时更改系统中的任何代码）表面上看来是很疯狂的想法。如果没有测试，要这么做绝对会累死人。有了测试，并且在几个月内通过编写大量测试改进了测试的质量后，你可以成功地做到这一点。如果你每次只集成几个小时的更改工作，那么就可以做到。当然，这正是你将要做的。

集体所有权的效果之一是：复杂代码不会生存很长时间。由于每个人都习惯于到处查看系统，因此这样的代码很快就会被发现。并且当它被发现时，就会有人去尽量简化它。如果简化不起作用（如测试失败所证明的），那么将放弃这些代码。即使出现了这种情况，仍然会有代码的原作者（程序员对）以外的其他人能理解这些代码为什么不得不复杂。但是，通常简化是有效的，或者至少部分是有效的。

集体所有权倾向于在最初就防止复杂代码进入系统。如果知道其他人不久（在几小时之后）会看你此刻编写的代码，你会在增加无法立刻证明其合理性的复杂性之前仔细斟酌。

集体所有权让你感觉对项目有更大影响力。在 XP 项目中，你永远不会因为别人的愚蠢行为而进退两难。发现有什么妨碍时，只需直接把障碍

移除。如果由于某些东西暂时有用而选择保留它们，那是你自己的事。但你永远不会被卡住。所以你永远不会有这种感觉：“如果我不是必须对付这帮蠢材的话，我原本能够完成我的工作。”少了挫折感，思路更清晰。

集体所有权还能够帮助在团队内传播系统的知识。不太可能出现这种情况：系统的某个部分只有两个人了解（起码这两个人肯定是一对，这已经比常见的状况好多了，不会让一个聪明的程序员掌握所有其他人的命运）。这进一步减少了项目的风险。

16.3 结对编程

结对编程实在值得单独写一本书来讲。它是一门精微的技巧，你可以用整个后半生来修习它并不断得益。本章的目的仅在于指出它为什么能在XP中发挥作用。

首先要讲结对编程不是什么。它不是一个人编程时另一个人在旁边看着。只是观看别人编程就好像在沙漠中看着青草如何死去。结对编程是两个同时努力编程（分析、设计和测试）并一起研究如何才能更好地编程的人之间的对话。它是一种多层次的会话，由计算机协助并以计算机为会话的中心。

结对编程也不是辅导会话。有时一对中的一个伙伴比另一个的经验丰富得多。这时候，开头的几次会话看来很像是辅导性的。缺乏经验的伙伴将会提出很多问题，而几乎不进行输入。可是，缺乏经验的伙伴很快就开始能发现一些愚蠢的小错误，如括号不对称。经验丰富的伙伴会注意到他的伙伴提供的帮助。再过几周之后，缺乏经验的伙伴开始能理解经验丰富的伙伴所使用的较大型的模式，并注意到背离这些模式的地方。

通常，几个月后，伙伴之间的差距就不再像最初时那么大。经验较少的伙伴更加频繁地进行输入。他们注意到两人彼此都互有优点和弱点。生

生产率、质量和满意度都提高了。

结对编程不是把两个人强绑在一起。如果你看了第二章“开发情节”，你将会注意到我所做的第一件事是寻求帮助。有时在任务开始时你要找的是某个特定伙伴。然而，更常见的是，你只能找到有空的人。并且，如果两个人都有任务要做，那么你们将协商在上午完成一个任务，下午做另一个。

如果两个人不能融洽相处怎么办？他们不必非要配对。两个人无法配对使得安排其余的配对更加难办。如果人际关系实在糟糕，那么花点时间重新组合总要好过剑拔弩张。

如果有人拒绝配对怎么办？他们可以选择像团队的其他人一样工作，也可以选择在团队外找工作。XP 不能用于所有人，也不是每个人都适合 XP。但是，你不必在极限工作的第一天就开始全职配对。就像所有其他事情一样，你需要朝着既定方向逐步前进。先尝试一小时。如果不行，试着找出问题所在，然后再试一小时。


那么，为什么结对编程能在 XP 中发挥作用呢？第一个准则是沟通，几乎没有什么形式的沟通比面对面的交流更加有效的了。因此，结对编程能在 XP 中发挥作用是因为它鼓励沟通。我喜欢用一池水来作类比。当团队中的某个人获得了一点新信息时，这就像在水中放入了一滴染料。由于各个组对总在不停地变换，因此这些信息就迅速在团队中传播开来，正如染料在水池中扩散开一样。可是，与染料不同，这些信息在传播过程中将变得更 valuable、更有效，团队中每个人的经验和智慧不断丰富着这些信息。

以我个人的经验看来，结对编程的生产率比将工作分割开给两个程序员，然后再集成结果要高。结对编程通常是采用 XP 的人们最难通过的一关。我所能讲的就是：你应该掌握结对编程，然后在结对编写生产代码的情况下进行一次迭代，在独自编程的情况下也进行一次迭代。接下来你就可以自己做决定了。

即使你的生产率并没有提高，你仍然希望配对，这是因为产生的代码的质量要高出很多。当一个伙伴忙于输入时，另一个人可以在更具战略性的层面上思考。这条开发路线通往何方？它会走到死胡同吗？有更好的总体策略吗？有机会重构吗？

结对编程的另一个强大功能是：许多实践缺了它将无法进行。在重压之下，人们会返回以前的习惯。他们会不再编写测试。他们会推迟重构。他们会避免集成。可是如果有你的伙伴在边上看着，有这样的可能性：即使你很想放过其中的一个实践，可你的伙伴不会同意。这并不是说程序员对就不会犯过程错误。他们当然会犯错误，否则就用不着教练了。但是，结对工作时忽视对团队的义务的机会要比单独工作时小得多。

结对编程的对话天性也改善了软件的开发过程。你很快就学会在各种不同层面讨论问题——这里的代码、系统中其他地方处的类似代码、过去的类似开发情节、多年前的类似系统、正在使用的实践和如何改进它们。



第 17 章

设计策略

从一个非常简单的起点出发，我们将继续完善系统的设计。
我们将去掉任何不能证明是有用的灵活性。

从许多角度看来，这是最难写的一章。XP 的设计策略是永远使用能运行当前测试套件的最简单的设计。

这倒不是什么坏事。简单有什么错？测试套件有什么错？

17.1 可能有效的最简单的东西

让我们回过头来一步步地看一下这个答案。所有四个准则都包含在此策略中：

- ✧ 沟通——复杂的设计比简单的设计更难沟通，因此应该创建一个能够产生尽可能简单的设计的设计策略，并且使它与其余的目标相一致。另一方面，应该创建一种能够产生沟通性设计的设计策略，在沟通性的设计中，设计元素能够将系统的要点传达给读者。
- ✧ 简单——我们的设计策略应该能产生简单的设计，但是策略本身

也应该是简单的。这并不意味着它必须容易执行。好设计从来都不容易。但是策略的表达应该简单明了。

- ✧ 反馈—— 在使用 XP 编程方法之前，在设计中我总会遇到的一个问题是不知道设计对了还是错了。进行设计的时间越长，这个问题就越严重。简单的设计耗时短，因而可以解决这个问题。下一步是对设计进行编码，看看代码怎么样。
- ✧ 勇气—— 设计一段时间后停下来，相信自己在需要的时候能够添加更多的设计，有什么比这更需要勇气的呢？

要遵循这些准则，必须：

- ✧ 创建一个可以产生简单的设计的设计策略。
- ✧ 迅速找到一种能够验证其质量的方法。
- ✧ 将我们学到的东西反馈到设计中。
- ✧ 尽量缩短整个过程的周期。

这些原则也适用于设计策略。

- ✧ 少量的初始投资—— 应该在得到回报之前在设计上投尽可能少的钱。
- ✧ 假定简单性—— 应该假定所能想到的最简单的设计实际上能行。这样当最简单的设计无效时，我们就有时间对其进行改善，而且用不着承担额外的复杂性的成本。
- ✧ 递增变化—— 随着逐步的更改，设计策略会起作用。我们一次只做少量设计，永远不会有系统“已经设计完毕”的时候。系统永远需要更改，尽管其中某些部分在一段时间内会保持不变。

- ✧ 轻装上路——设计策略不应生成任何“额外”设计。应该有足够的设计来满足当前的目的（完成高质量工作的需要），但是不能画蛇添足。如果我们愿意做更改，就会乐意从简单的设计开始，然后不断完善它。

XP 的工作方式与许多程序员的本能相背离。程序员习惯于预见问题。当问题晚些时候出现时，我们会很高兴；如果这些问题不出现，我们就注意不到。所以设计策略必须绕开这种“推测未来”的行为。幸运的是，大多数人可以去掉“自找麻烦”（用我祖母的话来说）的习惯。不幸的是，越聪明的人，就越难去掉这个习惯。

了解这一点的另一种方法是提出这样的问题：“什么时候添加更多的设计？”。通常的答案是应该为明天进行设计，如图 8 所示。

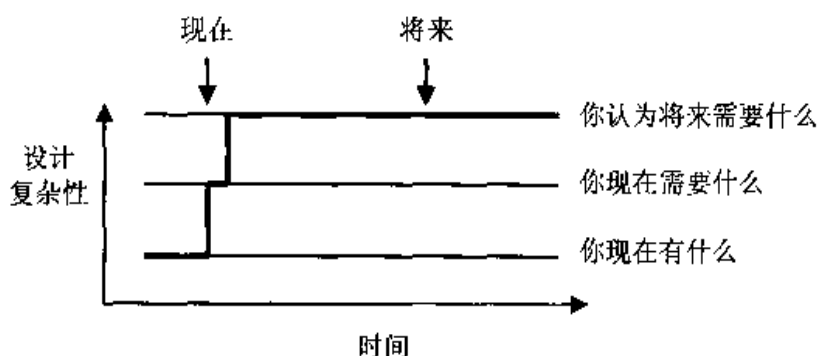


图 8 如果更改的成本随着时间推移而大幅增加

如果在当前和将来这段时间内不会有任何变化发生，这种策略是正确的。如果你完全了解将要发生什么样的问题，并且完全知道怎样解决它，一般来讲，最好是把现在和将来需要的都添加进去。

此策略的问题是不确定性。特别是：

- ✧ 有时明天永远不会来临（也就是说，客户把你提前设计的功能取消了）。

◇ 有时你会在“明天”来临前学到一种更好的工作方法。

在每种情况下，都必须做出选择：要么支付用于除去额外设计的费用，要么负担使用一个更复杂而不会立即带来效益的设计的成本。

我绝对不会打赌不会发生变化，当然，我也不会打赌我学不到东西。这样，我们就需要改变：今天为今天的问题进行设计，明天为明天的问题进行设计，如图 9 所示。

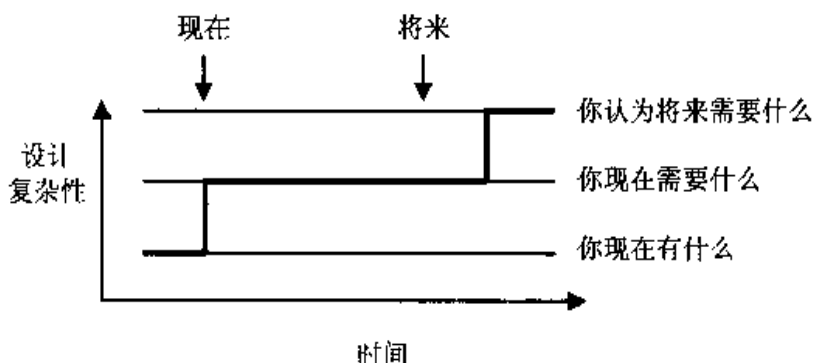


图 9 如果随着时间的推移,更改的成本一直很低

这样就产生了下面的设计策略：

1. 从一个测试开始，这样就能知道何时完成。我们必须专门为编写测试来做一些设计：对象和它们的可见方法是什么？
2. 设计并实现，只要能使该测试运行即可。要使这个测试和所有以前的测试都能运行，你将不得不设计足够的实现。
3. 重复。
4. 如果有机会可以使设计更简单，就抓住机会去做。请参见 17.3 节“什么是最简单的”以获得指导这种策略的原则的定义。

这个策略可能看起来简单得可笑。它确实简单，但并不可笑。它能创建大型的复杂系统，当然，这并不容易。在紧张的工期内工作而且还要抽

出时间来进行清理，没有比这更难的事情了。

以这种方式进行设计，在第一次遇到问题时，你将以一种十分简单的方法实现它。第二次使用这种方式时，它会变得更通用。第一次使用是付学费。第二次使用则是为了得到灵活性。这样，永远不用为你用不着的灵活性付出什么，而对于第三次、第四次、第五次变化，则可以根据需要可以使系统的相应部分更灵活。

17.2 “通过重构进行设计”的工作原理

如果说设计策略有什么奇怪的地方，那么在执行过程中你会感觉到一点。以第一个测试用例为例。我们会说：“假如需要做的只是实现这个测试用例，那么只需要一个对象、两种方法就行了。”我们实现了这个对象和两种方法，可以收工了。整个设计只是一个对象，需要一分钟。

然后就开始实现下一个测试用例。我们既可以抄袭一个解决方案，也可以将现有对象重新构造为两个对象。接下来，实现测试用例需要替换这两个对象之一。因此，我们先进行重新构造，运行第一个测试用例以确保它能运行，然后再实现下一个测试用例。

这样一两天后，系统就足够大了，使得我们可以想象有两个团队对系统进行工作而不用担心总是互相冲突。这样我们用了两对程序员，同时实现测试用例，周期性地（一次几个小时）集成它们的更改。再过一两天，系统就能够支持整个团队以这种方式进行开发了。

有时，队员会有一种感觉，好像有种隐患已经悄然逼近。也许他们已经觉察到了一种与他们的估算不同的持续偏差，或者也许当他们得知必须对系统的某一部分进行更改时，胃部有些不舒服。在任何一种情况下，都会有人喊一句“到时候了”。于是，成员聚集在一起，结合使用 CRC 卡、简图和重构，从整体上重建系统的结构。

不是所有重构都能在几分钟内完成。如果发现生成了一个乱糟糟的继承层次结构，可能需要一个月的集中工作来整理它。但是你不可能用一个月的时间来做一件事情。你需要为这个迭代增加故事。

当面对的是大规模的重构时，必须把它分成较小的步骤来进行（又是递增变化）。你将需要不断编写测试用例，而这时你会发现有机会向人目标更进一步。抓住那个机会，这边改变一个方法，那边改动一个变量。最终这个大规模的重构工作就只是小事一桩了。这样几分钟内就可以完成它。

我曾经在一个保险合同管理系统上以一次一步的方式进行大规模重构。层次结构如图 10 所示。

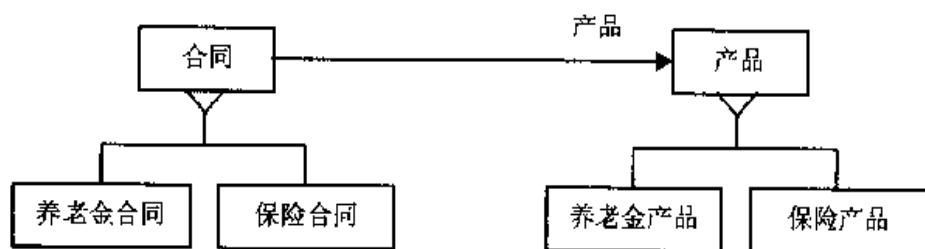


图 10 合同和产品有并行子类

此设计违反了“一次且只有一次”的规则，因此我们转为使用如图 11 所示的设计。

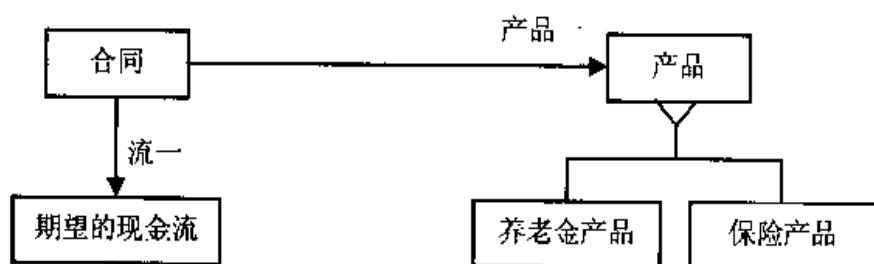


图 11 Contract 引用 Function 但是没有子类

在我参加整个系统的工作的一年里，我们完成了大量小步骤，一步步走向想要的设计。我们把 Contract 子类中的职责放到了 Function 子类或 Product 子类里。当我的工作合同到期时，Contract 子类还没有清除完毕，

但是它比最初时简洁多了，而且正在渐渐消失。同时，我们不断把新功能投入到生产中。

就是这样。就这样设计“极限”。从 XP 的角度来看，设计不是画几幅图形，然后按图实现系统。应该像开车一样。“学习开车”指出了一种不同的设计方式——启动汽车，朝这边开一点，朝那边开一点，然后再开回这边。

17.3 什么是最简单的

最佳设计的定义：能运行所有测试用例的最简单的设计。如果采用这一定义，那么“最简单的”是什么意思？

类最少的设计是最简单的吗？如果是，那么这将导致太大而无效的对象。最简单的设计是方法最少的吗？这会导致太大的方法和重复。代码行最少的设计是最简单的吗？这会导致为了压缩而压缩，从而损害沟通。

下面是“最简单”的意思——四种约束，按优先级排序：

1. 系统（代码和测试一起）必须能够沟通任何你希望沟通的内容。
2. 系统不能包含重复的代码（1 和 2 一起构成了“一次且只有一次”规则）。
3. 系统拥有尽可能少的类。
4. 系统拥有尽可能少的方法。

系统设计的目的首先是沟通程序员的意图，其次是为系统的逻辑提供生存的地方。以上所述的约束提供了可用于满足这两项需求的框架。

如果把设计看作是一种沟通媒介，那么你就会发现每个重要概念的对象或方法。你将选择一起合作的类和方法的名称。

如果沟通时受到了限制，就必须找到一种方法，消除系统中所有重复的逻辑。对我来说，这是设计中最难的部分，因为首先必须找出重复，然后还要找到消除它的方法。消除重复自然而然地使你创建出许多小对象和小方法，因为如果不这样，就一定会有重复。

但是创建新的对象和方法可不是为了好玩。如果发现有些类或方法不做任何事也不进行沟通，那么就删除它。

还可以把这个过程看做是擦除。现在有一个能运行测试用例的系统，删除没有任何沟通或计算上的用途的东西，剩下的就是可能有效的最简单的设计。

17.4 这是如何工作的

降低软件超时成本的传统策略是减少返工的可能性和费用。XP 的策略恰好相反。XP 不会降低返工的频度，而是欢迎返工。没有重构的一天就像生活在阴霾里一样。这怎么会降低成本呢？

关键是风险和时间一样都是金钱。如果今天添加一个设计功能而明天使用它，你赢了，因为今天添加更省钱。但是，第三章“软件开发的经济学”中认为这种评估并不全面。如果不确定性很高，选择等待的价值就很高，所以这时最好等待。

设计不是免费的。此情况的另一方面是你今天添加的设计越多，系统的开销就越大。有更多内容需要测试、理解和解释。所以每天不仅要支付费用的利息，还要支付少量设计税。考虑到这一点，今天和明天的投资之间的差别可能要大得多，因此明天的问题明天设计仍然是个好主意。

要是这还不够，还有风险这位杀手。正如“软件开发的经济学”一章中所指出的，你无法对明天发生的事情的成本进行估算，必须估算发生这种事情的可能性。现在，我和其他人一样喜欢猜测并猜对。可是细想起来，

却发现猜对的几率并不象我想象的那么高。通常我在一年前创建的绝妙设计中几乎没有任何正确的推测。在完成设计前，我不得不对每一部分返工，有时甚至返工两三次。

因此可以看出，今天做出设计决策的成本等于决策费用加上成本利息，再加上它添加到系统中的惯性。今天做出设计决策的好处是将来能够使用此决策的预期价值而得到实益。

如果今天的决策成本很高，决策正确的可能性又很低，而明天能够找到更好的方法的可能性很高，并且将其添加到设计中的费用很低，那么我们可以得出如下结论：永远不要在今天做出今天不需要的决策。实际上，这就是 XP 的结论：“今天就够麻烦的了。”

现在，有几个因素能使上面的评估一文不值。如果明天做出决策的成本非常高，那么我们应该今天就做出决策（虽然正确的几率很小）。如果设计的惯性足够小（例如，你有非常非常聪明的人员），那么实时设计的好处就很小。如果你非常非常善于推测，那就可以在今天完成所有设计。对于所有其他人，我看不到还有比这更好的结论：今天的设计应当今天做，明天的设计留到明天做。

17.5 设计中图形的角色

所有这些漂亮的设计和分析图形怎么样？比起代码来，有些人确实更善于用图形进行设计。这些更具形象思维能力的人能为设计做些什么呢？

首选，使用明晰的图形（而不是纯想象的和文字式的系统模型）来设计软件没有任何错误，关于图形方式还有很多东西要讲。绘制图形可以提供一些有关设计的状况的细微线索。是否觉得无法将图形中元素的数量减少到可以管理的程度？是否有明显的不对称？是否连线的数量比实体还多？这些都是因为有了设计的图形表示形式而使糟糕的设计变得明显的症

状。

用图形进行设计的另一个好处是速度快。在为一个设计进行编码的时间内，可以使用图形比较和对比三个设计。

然而，使用图形的麻烦在于它们不能提供具体的反馈。它们能够提供有关设计的某种反馈，但是却使你无法得到其他反馈。不幸的是，这些无法得到的反馈正是最重要的——能不能运行测试用例？支持简单代码吗？这些反馈只能从编码中获得。

因此，一方面，使用图形可以加快设计速度；另一方面，使用图形进行设计是在冒险。我们需要的是既能利用图形设计的强项，又能弥补其弱点的策略。

我们不是孤军作战的，我们有原则来指引方向。看看它们是什么：

- ✧ 少量的初期投资——意味着我们一次只画几幅图形。
- ✧ 积极求胜——意味着我们不应出于恐惧（例如，可能是因为我们想推迟承认对设计没概念的时间）而使用图形。
- ✧ 快速反馈——意味着我们要迅速弄清图形是否准确。
- ✧ 顺应人的本能——意味着要鼓励那些善于使用图形的人使用图形。
- ✧ 拥抱变化和轻装上路——意味着一旦图形对代码起到了作用后就不再保留，因为它们所代表的决策明天就可能会改变。

XP 的策略是任何人都可以他们想要的图形进行设计，但是一旦出现了问题，而该问题可以由代码来回答，设计人员就必须从代码中寻找答案。不要保留图形，例如，可以在书写板上画图。如果你希望把白板保存起来，那就意味着还没有对团队或系统传达设计。

如果有一种源代码，以图形表示最为合适，那么就完全应该用图形来

表示、编辑和维护它。使你能指定系统的整体行为的 CASE 工具很不错。它们称它们所做的是“代码生成”，但是对我来说，它看起来确实像一种编程语言。我不反对使用图形，而是想同化同一信息的多种形式。

如果你正在使用文字式的编程语言，那么如果遵循这条建议的话，画图的时间绝对不会超过十到十五分钟。然后就会明白想要问系统什么问题。

得到答案后，可以再画几幅图形，直到遇到另一个需要具体答案的问题。

同样的建议也适用于其他非代码设计表示方法，如 CRC 卡。花几分钟画图来发现问题，然后转到系统本身去解决困惑你自己的风险问题

17.6 系统的体系结构

在前面，我没有用过“体系结构”这个字眼。

在 XP 项目中，体系结构和在其他软件项目中一样重要。体系结构一部分是通过系统隐喻来概括的。如果使用了好的隐喻，团队中的每个人都能说出系统作为一个整体是如何工作的。

下一步是弄清楚故事怎样转变成对象。“计划游戏”的规则规定第一次迭代必须产生一个系统的整体功能框架，但是你仍然必须做可能有效的最简单的事情。怎样把两者协调起来呢？

对于第一次迭代，选择一套简单的、基本的、你认为能够创建整个体系结构的故事。接下来，缩小范围，用可能有效的最简单的方法实现故事。这个过程结束时，你就拥有了一个体系结构。可能它并不是你所期望的，但是你会了解一些东西。

如果找不到一套可以使你创建体系结构的故事，而且你很清楚将非常需要这个体系结构，该怎么办？可以根据推测设计出整个体系结构，或者设计出能满足当前需求的那部分体系结构，而相信将来能添加体系结构的

更多部分。我提出现在需要的体系结构，并相信自己有能力在以后对它进行更改。



第 18 章

测试策略

我们总是在编码前编写测试。我们将一直保留这些测试，并频繁地运行全部测试。我们还会根据客户的观点生成测试。

讨厌！没有人想讨论测试。测试是软件开发中令人讨厌、不被注意的部分。问题是，每个人都知道测试很重要。每个人都知道他们做的测试还不够。我们注意到项目没能按理想状况很好地进行，因此我们希望通过做更多的测试能够帮助解决这个问题。但是当我们接着去读测试书籍时，马上就会在无数种测试的种类和方法面前头脑发懵。我们不可能在进行所有测试的情况下仍能完成任何开发。

下面谈一下 XP 测试是什么样的。每次程序员编写了一些代码，他们都认为这些代码能行。于是，每次他们认为某些代码能行时，他们就把这种凭空而来的信心变为程序中的一种附属品。这种信心可以供他们自己使用。而由于它就在程序中，所以其他人都会使用这种信心。

同样的问题也会出现在客户那里。每次他们想到程序具体应该做什么时，他们就会把它变为另一份信心投入程序中。然后他们的信心与程序员的信心汇集到一起，程序就变得越来越自信。

现在，测试人员会看着 XP 测试傻笑。这不是喜欢测试的人干的活。

恰恰相反。这是希望程序能运行的人的工作。因此，你应该编写有助于使程序能够运行并使程序一直能够运行的测试。这就够了。

记住这条原则：“顺应人的本性，而不是忤逆它”。我所读过的测试书籍中的基本错误是，他们是以“测试是开发的中心”为前提的。你必须做这样的测试，那样的测试，对，另外还有这一个。如果想要程序员和客户编写测试，我们最好尽可能使这个过程不那么痛苦，要知道测试就像是工具，每个人关心的是使用工具测试的系统的行为，而不是测试本身。如果有可能不用测试就进行开发，我们会立刻扔掉所有的测试。

Massimo Arnoldi 写道：

不幸的是，至少就我（并且不止我一个）而言，测试是违背人的本性的。如果你在自己身上放任这种懒惰思想，那么你会发现自己没有进行测试就开始编程了。然后过了一会儿，当理性的部分占了上风时，你就会停下来，又开始编写测试。你也提到过，结对编程减小了两个成员同时放任懒惰思想的可能性。（来源：电子邮件。）

在 XP 中，你必须编写的测试是独立的和自动化的。

首先，每个测试都不与你编写的其他测试交互。这样就避免因为一个测试的失败而导致成百的其他测试失败。没有什么比错误的反面信息更能浇测试的冷水了。当早上到了办公室发现有一大堆缺陷时，你会非常冲动。当最后证明它不是什么大不了的事时，就会非常沮丧。这种情况发生了五次或十次后你还能在测试上集中注意力吗？不可能。

测试还是自动化的。当压力增加时，当人们工作得太累时，当人为判断开始失效时，最能体现测试的价值。因此测试必须是自动的——返回不受限制的成功/失败，来指示系统是否能够运转。

在测试变得和代码一样复杂和易出错之前，测试所有内容是不可能的。

什么都不测试是自杀（从测试是独立、自动化的意义上来说）。那么，在所有可以想到的要进行测试的东西中，应该测试哪些呢？

应该测试可能出错的部分。如果代码非常简单，不可能出错，并且估计可疑代码实际上不会在实践中出错，那么就不应为这些代码编写测试。如果让你测试所有内容，那么很快你就会意识到你编写的大多数测试都是没有价值的，并且如果你和我完全一样，你就会不再编写它们。“这些测试真是一点用也没有。”

测试就是一种赌博。当结果与预期相反时，测试就胜出了。测试胜出的一种可能是在没指望测试能运行的时候，它却能行。这时候最好去找出它能行的原因，因为代码比你更聪明。测试胜出的另一种可能是在希望测试能运行的时候，它却出错了。无论哪种情况，你都能学到一些东西。而软件开发也就是学习。学得越多，开发得就越好。

这样看来，如果能做到，你会只编写那些能胜出的测试。由于无法知道哪个测试可以胜出（如果你能够知道，那么你已经都清楚了，用不着学任何东西），所以要编写可能会胜出的测试。测试时，思考哪些种类的测试更可能胜出以及哪些种类的测试不会，然后多编写一些确实能胜出的测试，少编写些不会胜出的测试。

18.1 谁编写测试

如我在本章开头所说的那样，测试有两个来源：

- ◆ 程序员。
- ◆ 客户。

程序员逐个方法地编写测试。程序员在下列情况下编写测试。

- ✧ 如果某个方法的接口还不清楚,那么在编写该方法前要编写测试。
- ✧ 如果接口很清楚,但是你认为实现会稍微有一点复杂,那么在编写该方法前要编写测试。
- ✧ 如果你想到了代码不像被编写的那样工作的一种异常情况,那么应当编写一个测试以针对这种情况进行沟通。
- ✧ 如果你后来发现了一个问题,那么应该编写一个测试来隔离这个问题。
- ✧ 如果你打算重构某些代码,但是不确定它应该有什么样的行为,而且还没有有疑问的行为方面的测试,那么应该先编写一个测试。

程序员编写的单元测试必须 100% 运行。如果单元测试中有一个出错了,那么对于团队中的所有人来说,最重要的工作莫过于修复该测试。因为,如果测试出错了,就需要做数量未知的工作来修复它。它可能只需要花费一分钟,但也许需要花费一个月,你不知道。并且由于程序员控制单元测试的编写和执行,因此他们可以保持测试完全同步。

客户逐个故事地编写测试。他们需要问自己的问题是,“在我可以确定这个故事完成了之前必须检查什么呢?”他们提出的每个方案都变成了测试(在这种情况下是功能测试)。

功能测试没必要一直都 100% 运行。由于它们来自一个不同于代码本身的来源,我还没找到以同步代码和单元测试的相同方式同步这些测试和代码的方法。因此,单元测试的测量是二进制的(100% 正确或失败),功能测试的测量则必须是基于百分比的。随着时间的推移,你会希望功能测试的得分能够增加到接近 100%。在你接近发布的时候,客户需要将失败的功能测试分类。某些功能测试相对于其他功能测试更为重要,需要优先修复。

客户通常不会自己编写功能测试。他们需要有人先将他们的测试数据


翻译成测试，随后逐渐创建出可以让客户编写、运行和维护他们自己的测试的工具。这也是 XP 团队不论大小都至少配有一名专职测试员的原因。测试员的工作是将客户有时比较模糊的测试概念转换为真实、自动化和独立的测试。测试员还将客户授意的测试用作确定可能会使软件出错的变量的起始点。

即使你有专职的测试员（他以攻破被认为已完成的软件为乐事），他们的工作也要受到与编写测试的程序员相同的经济框架的束缚。测试员下赌注，希望本应失败的测试成功或者本应成功的测试失败。因此测试员随着时间的推移也在学会编写越来越好的测试（更可能胜出的测试）。测试员的用途当然不只是费尽心机做出尽可能多的测试。

18.2 其他测试

尽管单元测试和功能测试是 XP 测试策略的核心，但还有些其他测试可能会很有意义。一个 XP 团队想发现他们何时走错了路，这时就用得上新的测试。他们可以编写下列测试种类中的任何一种（或者你可以在测试书籍中找到的任何其他测试中的一种）。

- ✧ 并行测试——一种旨在验证新系统是否完全像旧系统一样工作的测试。更确切地说，该测试显示新系统与旧系统之间的差异，这样当差异小到业务人员可以将新系统投入生产时，业务人员就可以做出业务决策。
- ✧ 压力测试（stress test）——一种旨在模拟可能的最高负载的测试。压力测试对于性能特性不容易预知的复杂系统很有用处。
- ✧ 灵活测试（monkey test）——一种旨在确保系统遇到无意义的输入时能做出有意义的反应的测试。



第 3 部分

实现 XP

在本部分中，我们将讨论把上一部分中的策略付诸实践。一旦选择了一组显著简化了的策略，你马上就会拥有更人的灵活性。你可以将这种灵活性用于多种用途，但是需要意识到它的存在并了解它能提供什么样的机会。



第 19 章 采用 XP

一次一种实践地采用 XP，始终处理对团队最紧要的问题：一旦这个问题不再是最紧要的，就接着转向下一个问题。

非常感谢 Don Wells 为关于如何采用 XP 的问题提供了简单而显然正确的答案。

1. 挑出最棘手的问题。
2. 以 XP 方式解决它。
3. 当它不再是最棘手的问题时，重复上述步骤。

测试和“计划游戏”是两个明显的起始位置。许多项目都为质量问题或业务方与开发方之间权力的不平衡所困扰。另一本 XP 书籍《应用极限编程——积极求胜》将讨论这两个主题，因为它们是很常见的切入点。

这种方法有许多优点。它是如此的简单，连我都能懂（Don 一教我就明白了）。由于每次只学习一种实践，因此你可以详细地学习每种实践。由于你始终在处理最紧要的问题，因此改动的动机很强，而你付出的努力马上就会得到积极的回馈。

解决最紧要的问题还能消除一种对 XP 的误解，即认为它可以“放之四海皆准”。在采用每种实践时，要具体情况具体分析。如果没有问题，那么也不用考虑以 XP 方式解决它。

在采用 XP 时不要轻视物质环境的重要性，即使你没有意识到它是个问题。我经常用一把螺丝刀和一个六方孔螺钉扳手开始。下面两个步骤也要添加到进程中。

-1. 重新布置一下家具，以便你可以结对编程，并且客户可以与你坐在一起。

0. 买一些快餐食品。



第 20 章

改进 XP

希望改变其现有文化的项目远比能从头创造新文化的项目更常见。从测试或计划开始，在现有项目上每次进行一点点来逐渐采用 XP。

将 XP 用于新的团队是一种挑战，将它用于现有团队和现存的代码库则更难。你得面对所有现存挑战——学习技能、训练、完成自己的进度。你还得承受使生产软件保持在运行状态的直接压力。这些软件不太可能是按照新标准编写的。它可能比实际需要的更复杂。对它进行的测试可能没有达到你希望的程度。在新的团队中，你可以只选择那些愿意尝试 XP 的人。现有团队中则可能有某些持怀疑态度的人。最明显的问题是所有的办公桌都已布置好了，你甚至没办法组对编程。

在一个项目中改进 XP 所花费的时间比在一个同等的新团队中采用 XP 花费的时间更长。这是坏消息。好消息是你不必面对“从头开始”的 XP 开发所必须面对的一些风险。你永远不会处于这样一个危险的位置：认为自己对软件有个好主意，但又不是很明确。你永远不会处于这样一个危险的位置：在没有从真实客户那里得到直接而残酷的反馈的情况下做出许多决策。

我和许多团队交谈过，他们说，“是的，我们已经在做 XP 了。所有的东西，除了测试。我们有一份 200 页的需求文档。但是其他方面我们是完全按 XP 做的。”这就是为什么本章是一个实践接着一个实践地安排的原因。如果你已经在做与 XP 所倡导的相同的实践，那么你就可以忽略那个小节。如果你想采用某种新的实践，请查阅专门讲述该实践的小节。

如何才能使现有的团队将 XP 用于已经在生产中的软件呢？你将必须在下列方面修改采用的策略：

- ◇ 测试。
- ◇ 设计。
- ◇ 计划。
- ◇ 管理。
- ◇ 开发。

20.1 测试

在将现有代码转换到 XP 时，测试或许是最令人厌烦的部分。在测试之前编写的代码实在令人担心。你从来不会明白自己的明确处境。这种更改安全吗？你不能确定。

一旦开始编写测试，马上就柳暗花明。你对新代码充满了信心。你不介意进行改动。事实上，它挺有趣的。

旧代码与新代码之间的转变就象夜晚和白昼。你会发现自己总是避免处理旧代码。必须抵制这种倾向。在这种情况下获取控制权的唯一方法是把所有代码向前推进。否则，不利因素就会暗暗滋长，你将面对不可预知的风险。

在这种情况下，会诱使人返回头为全部现有代码编写测试。别这么做。相反，根据需要来编写测试。

- ◇ 当你需要向未测试的代码添加功能时，首先为代码的现有功能编写测试。
- ◇ 当你需要修复错误时，首先编写测试。
- ◇ 当你需要重构时，首先编写测试。

你将发现开发刚开始时感觉很慢。你在编写测试上花费的时间将远超过在正常的 XP 中编写测试的时间，并且觉得在新功能上取得进展要比以前慢许多。但是，系统中你经常关注的那些部分（引入注意的和新功能的部分）将迅速被彻底地测试。很快，你会觉得系统中最常使用的部分像是用 XP 编写的。

20.2 设计

到 XP 设计的过渡很像到 XP 测试的过渡。你将注意到新代码在感觉上完全不同于旧代码。你会希望立刻改正一切。不要那样，每次只改正一点。添加新功能时，请首先准备重构。在 XP 开发中，在实现之前始终要准备先进行重构，但是在向 XP 过渡时，将必须更频繁地这样做。

在过程的早期阶段，让团队确定一些大规模的重构目标。可能会有某个继承层次特别混乱，或者你想统一某个分散在系统中的功能。设定好这些目标，把它们写在卡片上，让它们看上去很显眼。到了可以宣布大的重构已经完成了的时候（可能会有几个月甚至一年的艰辛工作），办一个大聚会，隆重地将卡片烧掉，好好地享受一番美味佳肴。

这种策略在效果上非常像需求驱动测试策略。在开发活动中经常关注的那部分系统很快就会感觉像是你现在编写的代码一样，额外重构的费用很快会降低。

20.3 计划

必须将现有的需求信息转变成故事卡片，必须教会客户游戏的规则，客户必须决定下一个版本的内容。

转换到 XP 计划的最大挑战（和机会）是让客户明白他们可以从团队中获得更多利益。他们大概还没遇到过欢迎需求变化的开发团队。习惯于客户从团队中获得更多利益得花点时间。

20.4 管理

最难的过渡之一是习惯 XP 的管理。XP 管理是一个间接的和影响的游戏。如果你是一位管理人员，你或许会自己做出一些本来应该由程序员或客户来做出的决定。如果你遇到这种情况，不要惊慌，只需提醒你自己和其他在场的人你是在学习，然后请合适的人做这个决定并了解他们做了什么决定。

突然面对新责任的程序员不太可能马上能胜任工作。作为管理人员，你必须注意在过渡期间提醒每个人他们所选择的规则。在压力之下，所有人都会回到以前的行为模式，无论这些模式是否有效。

这种感觉有点像转换设计或测试。最初，会感觉有点困难。你明白自己不能全速前进。不断关注着天天出现的情况，你（和程序员以及客户）将学会如何毫不费力地处理它们。你很快就会开始感觉新的过程很舒服。不过，有时会出现你以前没有“极限完成”的情况。当出现这种情况时，向后退一步。提醒团队规则、准则和原则，然后决定做什么。

急速转变到 XP 的管理的最困难的一个方面是确定某个团队成员工作没有成效。在这种情况下，你最好停止使用他们。你一旦确定情况继续下去不会有什么改善时就应该做出更改。

20.5 开发

你需要做的第一件事情是将那些办公桌布置妥当。这不是说笑，请重新阅读一下有关结对编程的材料（第 16 章“开发策略”）。布置好办公桌，以便使两个人可以并排坐下，并且无需移动椅子就可以来回移动键盘。

一方面，在过渡到 XP 时，对于结对编程应该比通常所需的更加严格。结对编程一开始可能让人感觉不太舒服。即使你不喜欢这样，也强迫自己去这样做。另一方面，找个时间休息一下。独自离开，一个人编上几个小时的代码。当然，最后把结果扔掉。但是不要为了能在一周里进行 30 小时的结对编程而毁了编程的乐趣。

一点一点地将测试和设计问题解决掉。使所有处理过的代码遵循大家一致同意的编码标准。即使是这么简单的活动，你也会为自己从中学到的东西而感到惊讶。

20.6 有麻烦了吗

有些读者可能有一个现有团队，但是软件尚未进入生产阶段。你的项目中可能问题很多。XP 看上去可能是一种解救的办法。

别这么指望。如果你从开始就使用了 XP，你也许能（也许不能）避免当前的状况。但是，如果在河中央换马很难的话，那么从一匹快要溺死的马上换下来更是难上十倍。情绪会很高。士气会很低。

如果面临的选择是转变到 XP 或被解雇，首先要明白能够一致采用新实践的可能性不是很大。在压力之下，你会返回到旧有的习惯。你已经有许多压力了。成功进行转变的可能性正在急剧减小。请为自己制定一个比挽救整个项目更合时宜的目标。每次花费一天的时间，庆祝你可以学到这么多有关测试或间接管理的知识；或者是你可以做出如此漂亮的设计；或

者是删除了如此多的代码，或许你能从可能在第二天出现的混乱中理出些头绪来。

但是，如果要将一个有问题的项目转变到 XP，请果断行动。行事中庸会使每个人或多或少地处于原来的状态。仔细地评估当前的代码库，不用它是否更好？如果是这样，扔掉它，全部都扔掉。生一大堆火，把旧磁带都烧掉，放一周假，然后精力充沛地从头开始。



第 21 章

理想的 XP 项目的生命期

理想的 XP 项目要经历一个短暂的初期开发阶段，随后是多年同时进行的生产和优化，最后，当项目失去意义时体面地退休。

本章将使你对 XP 项目有一个总体的认识。它是理想化的——到现在为止你或许已经了解，任意两个 XP 项目都不可能（或不应该）是完全相同的。我希望你能从本章中了解到项目的总体流程。

21.1 探索

预生产对系统来说是一种不正常的状态，应当尽可能快地结束它。我最近听说的谚语是什么呢？“投产就是死亡”。XP 的说法恰恰相反。不投产就是只花不赚。这可能只是我的钱包的问题，但是我觉得老是支出而没有进账让人很不舒服。

当然，在投产之前，你必须相信可以投产了。你必须对自己的工具有足够的信心，确信可以完成程序。你必须相信一旦代码完成，就可以成天运行它了。你必须相信自己有（或者能学会）所需的技能。团队成员需要学会互相信任。

探索阶段是所有这些汇聚在一起的阶段。当客户认为故事卡片上的材料丰富到了能够保证一个不错的第一版，程序员认为不开始实际地实现系统就无法再更好地进行估算的时候，探索阶段就完成了。

在探索期间，程序员使用他们要在生产系统中使用的每一项技术。他们积极地探索系统体系结构的各种可能性。他们使用的方法是：花一周或两周的时间生成一个与最终要生成的系统相似的系统，但是要使用三种或四种方法来生成。不同的程序员对可以尝试用不同的方法生成系统并且进行比较，或者你可以让两个程序员对用相同的方法试着生成系统，然后看看产生的差异。

如果一周时间都不能开始起用和运行某项技术，那么我只能认为该技术太冒险了。这并不意味着你不该使用它，这只说明你应该更加仔细地探索它，并考虑替换的方法。

你可能会考虑在探索期间请来一位技术专家，这样你的尝试就不会受到那些原本可以由这些熟悉情况的人轻松解决的蠢毛病的牵制。不过一定要当心，不要盲目接受有关最终所使用的技术的建议。专家有时会形成一些基本价值体系与极限编程不完全一致的习惯。团队必须对他们所选择的实践感到满意。当项目变得逐渐失去控制时，说“是专家这么说的”可不会怎么令人满意。

程序员还应对他们要使用的技术的性能限度进行实验。如果可能的话，应该用生产硬件和网络模拟实际负载。并非要完成整个系统后才能编写负载模拟器。只是通过计算就能得到很多进展，例如，知道了网络必须支持每秒多少字节，就可以做实验检查网络是否能够提供所需的带宽。

程序员还应对有关体系结构的想法（如何生成一个可用于多级撤消操作的系统？）进行实验。在一天之内用三种不同的方法实现它，然后看哪一种感觉最好。当用户提出一些你还不知道如何实现的需求时，这些小的体系结构探索就变得非常重要。

程序员应该对他们在探索期间所从事的每一项编程任务进行估算。当完成了一项任务时，他们应该根据该任务实际要求的日历时间进行报告。当到了进行公开承诺的时候，估算实践将有助于提高团队对自己的估算的信心。

在团队对技术进行实践的同时，客户在实践编写故事。别指望这会完全一帆风顺。起初故事不会是你需要的那样。关键是使客户尽快得到对前几个故事的大量反馈，以便他们可以快速学会指定程序员所需要的，而不去指定程序员不需要的。关键问题是“程序员是否可以肯定地估算出完成故事所需的工作量？”有时需要编写不同的故事，有时程序员需要停下来一会儿做实验。

如果你的团队已经了解他们的技术并且彼此已经熟悉，那么探索阶段就可以在几周之内完成。如果团队对于技术或领域是完全陌生的，那么可能必须用上几个月的时间。如果探索用去的时间比这还长，那么也许应该找一个他们可以轻松完成的小型但是真实的项目来加快进程。

21.2 计划

计划阶段的目的是使客户和程序员能够就完成最小、最有价值的一套故事的期限有信心地达成一致。请参阅“计划游戏”以了解这样做的方法。如果在探索期间做过准备，那么计划（产生承诺的进度）应该会用一两天的时间。

第一版的计划应该在两到六个月之间。时间再短，你可能就无法解决任何重要的业务问题。（但是如果能做到，好极了！有关如何缩短第一版的方法，请参阅 Tom Gilb 的《Principles of Software Engineering Management》。）如果时间长过几个月，那就太冒险了。

21.3 第一版的迭代

承诺的进度分为多个一到四周的迭代。对于每个迭代计划内的每个故事，该迭代都会为其产生一套功能测试用例。

第一次迭代将产生体系结构，为第一次迭代挑选能迫使你创建“整个系统”（即使只是个骨架）的故事。

为后面的迭代挑选故事则完全由客户掌握，要问的问题是，“在本次迭代中，什么最有价值，而该去做呢？”

当你顺利地进行迭代时，你要寻找的是与计划的偏差。有什么比预期中多用了一倍的时间吗？还是只用了一半时间？测试用例能按时完成吗？做得开心吗？

检测到与计划的偏差时，就需要做出一些改动。可能需要更改计划——添加或删除故事，或者更改它们的范围。可能需要更改过程——你找到了利用技术的更好方法，或者使 XP 运转得更好的方法。

理想状况下，在每次迭代结束时，客户都已经完成了功能测试并且所有功能测试都能运行成功。每次迭代结束后，小小庆祝一次——买些比萨饼，放些烟火，让客户在已完成的故事卡片上签字。嗨，你刚刚按时交付了优质软件。也许它只用了你三周时间，可它毕竟是一项成果，值得庆祝。

在最后一次迭代结束时，就可以投产了。

21.4 生产

在进行一个版本的收场戏（“生产”）时，反馈的周期要缩短。不是进行三周的迭代，你可能在这时进行一周的迭代。你可能会举行每天一次的碰头会，以便每个人都知道其他人在做什么。

通常会有某个用于验证软件是否能够投产的过程，准备好实现新的测

试以证明软件适合进行生产。这个阶段经常用到并行测试。

在这个阶段，你可能还需要调节系统的性能。在本书中，关于性能调节，我讲得不多。我坚信这句格言，“能运行，能正确运行，能快速运行。”游戏后期是进行调节的理想时期，那时你会将尽可能多的知识嵌入在系统的设计中，你能够对系统中的生产负载进行最现实的估算，并且你可能会得到生产硬件。

在生产期间，你会放慢发展软件的步伐。不是说软件停止发展，而是说在你评估某个更改是否值得放入该版本中时风险变得更加重要。不过，要知道，你对系统的经验越多，对于应该如何设计系统就越有洞察力。如果你开始发现有大量想法，而你无法决定是否应该把这些想法加入这个版本，请做一个显眼的列表，这样所有人都能看到在此版本投产后你们行进的方向。

当软件实际投产后，办一个大型聚会。许多项目从未活到投产的那一天。你的项目能够充满活力地进行，这本身就是庆祝的理由。如果与此同时你一点都不觉得担心，你肯定是疯了，不过聚会可以帮助缓解已经过量而还要不断增加的压力。

21.5 维护

维护是 XP 项目真正的正常状态。你必须同时开发新功能，使现有系统保持运行，在团队中加入新人，并且向离开的成员说再见。

每个版本都从探索阶段开始。你可以尝试在上一个版本的后期制作中没敢进行的大重构。你可以尝试打算在下一个版本中添加的新技术，或者迁移到已经在使用的技术的更新版本。你可以对新的体系结构想法进行实验。客户可以尝试编写古怪的新故事以图成为业务上的大成功者。

开发一个已经在生产中的系统与开发一个还没有进行生产的系统一点

也不一样。你要对所做的改动更加小心。你必须准备好中断开发来应对生产问题。已经有了既有数据，在更改设计时，必须小心地迁移它们。要是预生产不是这么危险的话，你可能永远都不会投产。

进入生产状态可能会改变开发速度。应该保守地进行新的估算。在探索阶段，测量一下生产支持对开发活动的影响。我曾经见到过在投产后，日历时间与理想工程时间的比率有 50% 的增长（从每个工程日两个日历日到每个工程日三个日历日）。当然，不要猜测，去测量。

准备好改变团队的结构以应对生产。可以轮流担任“咨询台”的咨询人员，这样大多数程序员大多数时候都不必处理生产中断。请一定让所有程序员轮流出任这个职位——有些知识只能从生产支持中学到。当然，它可不像开发那样有趣。

在工作不断进行的过程中将新开发的软件投入生产。你可能知道软件的某些部分不能执行。无论如何，将它投产。我曾经参与过周期是天或周的项目，但是在任何情况下，都不应把代码闲置超过一次迭代的时间。时间的掌握依赖于验证和迁移的成本。在完成一个版本时最后需要做的是集成一块“不可能”破坏任何东西的代码。如果将生产代码库与开发代码库基本保持同步，就会更早地得到有关集成问题的警告。

当新成员加入团队中时，给他们两个或三个迭代的时间，而他们的工作就是大量提问，充当结对编程伙伴和阅读大量测试和代码。当他们感觉准备好了的时候，可以让他们负责一些编程任务，但是负载因子要小些。当他们证明自己能够完成任务时，就可以提高他们负载因子。

如果团队的组成在逐渐发生变化，那么在不到一年之内，你可以用全新的人员替换原有的开发团队，而不会干扰生产支持，也不会干扰正在进行的开发。与典型的“拿去，这堆文件里有你需要的全部信息”比较，这是一种风险小得多的移交方式。事实上，在项目中传播我们的文化与沟通设计和实现的详细信息同等重要，而传播文化只能通过人际交往来完成。

21.6 死亡

死得其所与活得舒适一样重要。人对 XP，这都没错。

如果客户无法想出任何新的故事，就是把系统封存起来的时候了。现在可以写一个五到十页的系统简介（就是那种在五年内发现需要改动些什么时，希望能找得到的文档）了。

这是很好的死亡理由——客户对系统很满意，想不出来在可预知的未来要添加什么。（我还没遇到这种事，但我听说过，所以我在这里提到它。）

还有一个不是那么好的死亡理由——系统就是无法交付。客户需要一些功能，而你怎么也做不到多快好省地添加它们。缺陷率爬升到了让人无法忍受的地步。

这是你长久以来一直与之斗争不已的熵运动死亡，XP 可没什么魔力，熵（entropy）最终也会抓住 XP 项目，你只能希望它晚点发生。

在任何情况下，我们都假定了不可能的事——系统需要死亡。每个人都应该清醒地认识到这一天会到来。团队应该明白事情的变迁兴衰。客户和管理人员应当能够统一思想，承认团队和系统实在无法完成所需要的软件。

然后就到了惜别的时候了。办一个聚会，邀请曾经为这个系统工作过的每个人，回来一起回忆。抓住这个机会努力寻找一下系统失败的根源，这样就能更加清楚以后该怎么做，与团队一起想象下次换种什么方式进行工作。



第 22 章

人员的角色

我要使极限团队运转起来，就必须有人充当特定的角色——程序员、客户、教练、跟踪者。

只有当团队成员各司其职时，一个运动团队才能做到最好。在足球队里，需要有守门员、前锋、后卫等。篮球队中，你需要有盯人后卫、中锋、前卫等。

出任其中一个位置的队员要承担起一些特定的责任——组织队友得分、阻止对方球队得分、或许还要负责球场上某块特定的区域。有些角色几乎是独立的，而另一些则需要纠正队友犯的错误，或者组织他们之间的互动。

这些角色已成为惯例，有时甚至成为比赛规则的一部分，而这完全是因为它们有效。也许所有职责的其他组合方式以前都被尝试过了。现在能看到这种球场上的分工是因为它们是有效的，而别的则被淘汰。

优秀的教练可以有效地使球员在他们的位置上出色地发挥作用。他们能在平时的训练中发现某个位置的球员的不足，并帮助球员改正偏差，或者，好的教练能够理解为什么可以允许那个球员的小偏差。

然而，伟大的教练知道这些位置的安排只是出于习惯而非自然法则。

有时，当比赛规则和球员的变化足够大时，就会出现新的位置，而某个老位置则会被废除。伟大的教练总是试图通过创建新的布阵和放弃现有位置安排来取得优势。

伟大的运动教练的另一种能力就是能够根据球员来塑造体系，而不是相反。如果你有一个在球员速度快的情况下非常犀利的体系，而在第一次集训时你发现自己面对的是一帮又高又壮的家伙，那么最好创造一个新的体系，尽可能让球队发挥自己的优势。许多教练都无法做到这一点。他们太在乎“体系”的完美性了，以至于他们看不到那个体系并不实用。

所有这些都对下面将要讲到的一些情况的一个大的警告。有一些角色在以前的项目中证明是很有效的。如果你的人不适合这些角色，那么请你改变这些角色。不要试图改变你的人员（或者，不要改变太多）。不要装得像这么做不会出问题一样。如果一个角色说：“这个人必须乐于冒险”，而你却只有一个谨小慎微的人，此时你只能去找另外一种责任划分方案，可以不需要愿意冒险的角色就能实现目标。

例如，我曾经和一个经理谈论过他的团队，其中一个程序员就是客户。我说那可行不通，因为程序员必须执行过程、做出技术决策而一定要把业务决策留给客户来做（请参阅“计划游戏”）。

那个经理与我争辩。“这家伙是一个真正的证券交易员，”他说，“他只是碰巧也懂得如何编程。其他的证券交易员都喜欢并尊敬他，他们都愿意信任他。他非常清楚系统该是什么样的。其他程序员能区分他什么时候是作为客户讲话，什么时候是做技术决策。”

好。这里的规则讲程序员不能是客户。在此情况下，这个规则不适用，仍然适用的是技术和业务决策的分离。整个团队（程序员/客户，特别是教练）必须清楚在特定时刻这个程序员/客户到底是在扮演哪个角色。教练还需要知道不管这种安排在过去进行得如何顺利，一旦陷入了麻烦当中，双重角色很可能就是问题的原因所在。

22.1 程序员

程序员是 XP 的核心。实际上，如果程序员总是能够在做出决策时仔细权衡短期利益和长期利益，那么项目中就不再需要程序员以外的任何其他技术人员了。当然，如果客户的业务运作并不是一定需要软件，那么就不需要程序员，这样也就用不着因为需要重要程序员而头大了。

表面看来，XP 程序员与其他软件开发方法中的程序员很相似。你花费时间来处理程序，使它们更庞大、更简单、更快。然而，实质上，两者的重点完全不同。你的工作并不是在计算机理解该做什么的时候就结束了。你的第一条准则是和别人进行沟通。如果程序能够运行，但还有某些重要的沟通没有进行，那么你的工作还没有完成。你要编写测试来证实软件的重要部分。你要把程序分成更多更小的块，或将太小的块合并为更大更连贯的块。你要找到一个能更准确地反映意图的命名系统。

可能这听起来像是不切实际地追求完美。但是事实绝非如此。你尽力为客户开发最有价值的软件，而不开发任何没有价值的东西。如果你能将问题的规模减到足够小的程度，那么你就有能力仔细地完成其余的工作。这样，你就会养成认真的习惯。

还有一些 XP 程序员必备的技能，而在其他开发风格中则并不要求或者至少不强调这些技能。结对编程是可以学会的技能，但经常会与那种从事编程的人的倾向相左。也许我应该讲得更清楚一些——书呆子通常不善于讲话。当然会有一些例外情况，学会与别人进行交谈是可能的，但事实是若要成功，只能与其他程序员进行紧密的沟通和配合。

极限程序员需要的另一种技能是习惯于简单。当客户说“你必须做这个、这个和这个”时，你必须做好准备讨论这些条目中哪些是必需的，每个条目需要完成多少。简单还要扩展到编写的代码。随时将上次的设计和模式准备在手边的程序员不大可能会成为成功的 XP 程序员。当然，

工具箱中的工具多些更有助于工作，但是有一些你知道何时不应使用的工具，远比了解所有工具的所有知识并冒险使用太多解决方案重要。

你还需要更偏向于技术性的技能。你必须能够编好程序。你必须会重构，这是一项起码与编程一样复杂难学的技能。你必须能分单元测试你的代码，而这就像重构一样，需要体验和判断才可以应用好。

你需要愿意放弃那种对系统的某个部分的个人所有权的想法，而接纳整个系统的共享所有权。如果有人改动了你编写的代码，无论改动是在系统的哪一部分进行的，你都必须信任这种改动并从中学习。当然，如果改动是错误的，你有责任改正它们。

最重要的是，你必须愿意承认你的恐惧。每个人都害怕——

- ◇ 怕自己看上去很蠢。
- ◇ 怕被认为是废物。
- ◇ 怕跟不上时代。
- ◇ 怕不能胜任。

如果少了勇气，XP 干脆就行不通。你将花费你的全部时间竭尽全力避免失败。相反，如果你愿意，你可以在团队的帮助下，承认你的恐惧。然后你就可以成为一个轻松惬意地编写优秀软件的团队的一分子了。

22.2 客户

客户是极限编程的两个重要组成部分的另一半。程序员知道如何编程。客户知道要编些什么。当然，最开始可不知道，但是客户和程序员一样愿意学习。

做一个 XP 客户并不是件容易的事。需要学习一些技能，比如编写好

的故事，以及学习一种能够使你成功的态度。当然，最重要的是，你要学会适应在没有项目的控制权的情况下对项目施加影响。事实上，你的控制范围以外的力量与你做出的决策同样能决定产生什么样的软件。商业行情的变化、技术、团队的组成和能力，所有的这些都对最后交付什么样的软件有很大影响。

你不得不做出决策。这是让与我一起工作过的一些客户最为难的技能。他们习惯于 IT 总是兑现不到一半的承诺，而且交付的产品还有一半是错误的。他们学会了对 IT 寸步不让，因为他们认为无论如何都是注定要失望的。有了这样的客户，XP 无法顺利运转。如果你是一个 XP 客户，团队需要你确信地说，“这个比那个更重要”、“这个故事这么多就够了”、“这些故事合起来刚刚够”。并且当时间变得紧张时（时间总会变得紧张），团队需要你能够改变主意。“我想在下个季度前我们不是非要完成这个不可。”有能力做出这样的决策有时可以拯救你的团队，还可以减轻他们的压力，这样他们就能为你尽力工作。

最好的客户是那些将会实际使用正在开发的系统，而且对要解决的问题有一定认识的人。如果你是这样的客户之一，你必须能够意识到自己在持有一种观点时，不是因为问题中的一些基本因素，而是因为问题一直是那样处理的。如果实际上你并不是系统的直接使用者，那么你能更加努力地工作，以确保你能够准确地代表实际用户的需求。

你需要学习如何编写故事。初看来这似乎是不可能的任务，但是团队将向你提供大量对你所编写的前几个故事的反馈，这样你就可以迅速了解到每个故事要占用多少资源，需要在故事里包含和排除什么信息。

你还需要学会编写功能测试。如果你是一个有数学基础的程序客户，那么你的工作很容易——利用电子表格，几分钟或几小时就已足够用来创建测试用例的数据了。或者你的团队将会为你生成一个工具，便于你输入新的测试用例。有公式基础（例如工作流）的程序也需要功能测试。你需