第 50 期（2005 年 6 月）

观念

X 非程序员
-Programmer
软 件 以 用 为 本

联系：think@umlchina.com

http://www.umlchina.com/

敏捷开发者发现新西兰人干得更好

[2005/6/16]

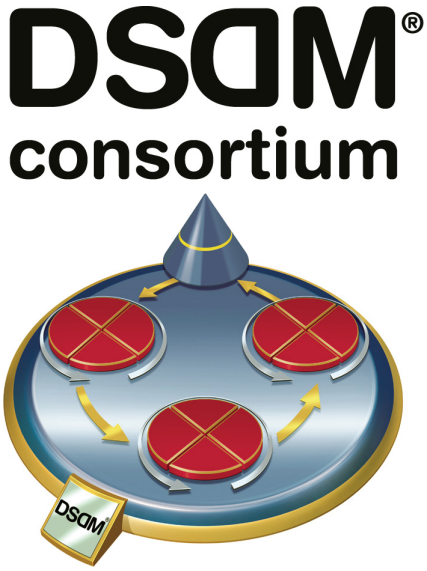
在伦敦工作的英国软件开发者 Stephen Hilson 发现他公司的 20 个雇员中有 18 个是来自新西兰。他们工作都很出色，这使得 Hilson 产生了去新西兰看看的念头，是什么导致了这些优秀开发者的产生。

Hilson 说他的另外一个动机是不再想当“大池子里面的小鱼一条”了，在英国情况就是这样的。首先他去奥克兰(译者注：新西兰北岛西北岸港市)看了看，发现在这一点上和伦敦没什么两样。第二站他去了威灵顿（译者注：新西兰的首都，位于新西兰北岛最南端的库克海峡的小港内），这里他从 2003 年 2 月呆起，开始转移他的公司 Visual Exchange 和一些新西兰职员（Kiwi staff）。

Hilson 和 Visual Exchange 专攻使用 DSDM (Dynamic Systems Development Method)方法的敏捷编程。这个方法的一个要点就是要在开发队伍和业务人员之间建立良好的交互。Hilson 指出，新西兰人平等的工作方式非常适合 DSDM。打个比方，如果你要去见某个高级主管，在英国要提前预约，在新西兰就并不总是如此。

Hilson 指出，这样的延误（译者注：预约等做法）会给使用 DSDM 的项目带来很大混乱，因为这些项目通常时间都很紧张。事实上，Hilson 建议至少要有一个业务代表和开发人员坐在同一间屋子里，即使是走过一个走廊来回答某个不是很清楚的某个问题，都会浪费太多时间。

上个月，Hilson 在一次报告中遇到了听众的这样一个问题：对于一个大合同，往往客户会要求非常严格的规约（specifications），这是否会和敏捷技术产生冲突？。Hilson 的回答是：“你必须和客户建立协作关系”，他并没有遇到这样的困难，即使对于严格定义的规约，DSDM 方法的前 15%的时间用来做计划，如果需要的话，在这个阶段结束时对于一个充分细致说明的项目，可以得到一个确定价格的合同，之后，为实现这个目标，采取什么方法就没有那么多关系了。



他承认，因为项目必须在指定时间内完成，必须要牺牲一些“nice-to-have”（有则更好）的功能。DSDM 方法建议在最开始对功能的分类进行协商，分为“must have”，“should have”，“could have” 和“want to have”，“musts” 和“shoulds”是“coulds”中那些可以在指定时间内较容易完成的功能，当时间和需求的安排必须考虑项目中不可预测的困难的出现。为满足进度要求，一些“wants”功能将被牺牲。

另外，在对问题的责任上，英国人和新西兰人的态度也不一样。Hilson 指出，在英国，业务人员只负责业务分析，项目主管只关心项目的管理，而在他的新西兰团队中，则是“我们出现了这个问题，谁想解决它？”

过去通常认为敏捷方法只能适用于 6 个月以下的较小的项目，但 Visual Exchange 已经成功地完成了较大的项目。Hilson 指出，一个例子是为 BBC 电视台做的脚本书写工具的项目，项目时间超过 18 个月，花费二千五百万新西兰$。这个项目非常成功，后来又被 Granada 公司购买了。

（自 computerworld，袁峰 摘译，不得转载用于商业用途）

Visual Studio 2005 要等到 2006 年了

[2005/6/6]

Visual Studio 2005 看来又要迟到了。

根据来自 Borland 公司（Borland 是微软的 ISV partner）产品市场部主管 Marc Brown 的消息，代号为 Whidbey 的微软产品又要延期了。根据 Borland 从微软得到的消息，Visual Studio 2005 将在今年底或明年初上市。

在本周微软 Florida 州 Orlando 举办的 TechEd 2005 大会上，Borland 将演示其 CalibreRM 需求管理工具包的一个版本，CalibreRM 可集成到 Visual Studio 2005 Team System。

开会时 Brown 说，“当然，VSTS 要发售时，微软会给消息的”。

上周，一个微软代表在回复问询的一封 email 中说到，“微软希望在 2005 年下半年发布 Visual Studio 2005”。但他也指出，“发布时间表最终会根据顾客的反馈来决定”。

去年下半年 Whidbey 实际上就已经到预定的发布日期了，但这个日期先被延后到 2005 年上半年，现在又延后到 2005 年下。但关注微软策略的高级分析员 Greg DeMichillie 指出，这个套件中包含了对.NET 框架和 ASP.Net Web 环境的更新，变动很大。

DeMichillie 说，“和上个版本比，变动大得多。而且 Team System 总是比 Visual Studio 的其他版本发布得晚的”。

“如果 Team System 2005 年发布还有问题的话，我不会诧异”，DeMichillie 指出，Team System 在建模设计工具以及基于团队开发方面都提供了先进的功能。

DeMichillie 说，微软必须依托 Team System 自己深入到应用生命周期管理这个领域中，因为微软过去在这个领域的主要伙伴 Rational 已经被竞争对手 IBM 收购了。

“这次收购把微软逼上了梁山，因为再也无法把 Rational 作为战略上的伙伴了”，另外，DeMichillie 指出，按照微软现在的 non-UML (Unified Modeling Language)方案，Team System 对一些顾客来说是并不受欢迎的。

“这难以吸引那些已经接受了 UML 的顾客”，Team System 使用微软的软件工厂技术来支持模型驱动的开发。

但 Rational 通过一位代表表示，它将继续对.NET 的支持。Rational 在.NET 方面的变更管理产品是 RequisitePro。

Borland 的 for Team System 的 CalibreRM 版本预计将在 2006 年初，在 Team System 之后推出。Brown 说，“微软在 Visual Studio Team System 中没有提供需求管理这部分功能，实际上将由 Borland 来提供”。

CalibreRM 将管理包括功能和用户需求在内的需求。“将会有一个和 Visual Studio Team System 数据仓库无缝集成的特别版本”，Brown 透露，CalibreRM for Team System 价格大概会在 2000 美元每用户。

根据微软的一份 email，他们在 TechEd 上计划对 Visual Studio 2005 进行加强，包括对编辑以及在 Visual C#调试时对变更代码的持续支持。

微软声称，有超过 11000 个顾客、伙伴和分析人员注册了此次会议。

除了 Borland 之外，诸如 Compuware 和 Infragistics 等合作伙伴也展示了他们的产品。

Compuware 将展示他们的 DevPartner、应用生命周期管理产品 QACenter 和 Visual Studio 2005 Team System 的集成。计划中的 DevPartner Studio8.0 专业版将提供在 Web Services 上的加强功能以及性能分析的功能。计划的 TestPartner5.4 测试组件 QACenter 将为 Team System 提供功能测试和回归测试功能。

Infragistics 将宣布其 NetAdvantage 2005 Volume 2 的发售，这是为 windows Forms, ASP.Net, Tablet PC, 和 COM 应用构建表示层的工具集。其中增加了 Windows Forms 树、grid 和图表功能，另外，也增加了 ASP.Net 的 grid 和图表功能。

（自 arnnet，袁峰 摘译，不得转载用于商业用途）

Telelogic 发布基于角色的生命周期集成解决方案

[2005/5/31]

生命周期管理现在是业内的热门词汇，但是要把软件开发过程中不同角色－从业务分析人员到系统架构师、设计人员、开发人员和测试人员－都整合在一起的说法过去还没有过。

Ovum 分析师 Bola Rotibi 说，“为了增强整个开发生命周期中的协作，企业正在寻找拉近他们的 IT 策略、产品开发路线和业务目标、顾客需求的距离的方法。他们需要这样的生命周期解决方案，以帮助全面理解业务及顾客的当前及未来的需求”。

Rotibi 介绍了以 Telelogic 最新版本为例的提供从集成初始到部署的产品类型。该公司宣布即将发布的 Telelogic 生命周期解决方案下的几款新产品。这个包中将包括 DOORS，Synergy，TAU 和 DocExpress。

Telelogic 负责建模和测试产品的 VP 产品主管 Matt Graney 指出，“并不是说我们要在同一天同一套光盘中推出所有这些产品。而是我们已经加强了这些产品之间的集成以利于团队间的交流，并为贯穿整个开发生命周期的协作提供更高的支持”。

伴随着这些产品的，是 Telelogic 将赌注压在自动化生命周期管理产品上的雄心，是它的老字号 DOORS 需求管理工具、TAU UML 工具和 Synergy 变更和配置管理工具以及 DocExpress 自动化报告和文档化工具的整合。Graney 解释说，这些 产品将继续保持各自的独立，也不会使用公共的仓库或文件系统，但一个工具的用户可以在该工具的本地环境中调用其他工具的功能。

Graney 告诉 AppTrends，这个版本的核心在于基于角色的集成。“我们已经关注这种集成有一阵子了。举例来说，如果某个建模人员想看看他要建模的到底是什么东西，想了解更多需求，他不需要从建模工具 TAU 中折腾到 DOORS 里面去，而是可以直接在 TAU 中去看放在 DOORS 数据库中的一个需求快照。同样，您也可以直接在 DOORS 中去看一个在 Synergy 中进行过的需求变更”。

这个版本不仅是在同一天发布 Telelogic 的这些产品，另外还提供统一的安装和 license 控制，以及统一的用户管理。这个版本中，Telelogic 公司还将推出基于 token 的版本控制机制。顾客公司可以购买整个产品套件的 token 系列，并根据需要分配到各个产品上。

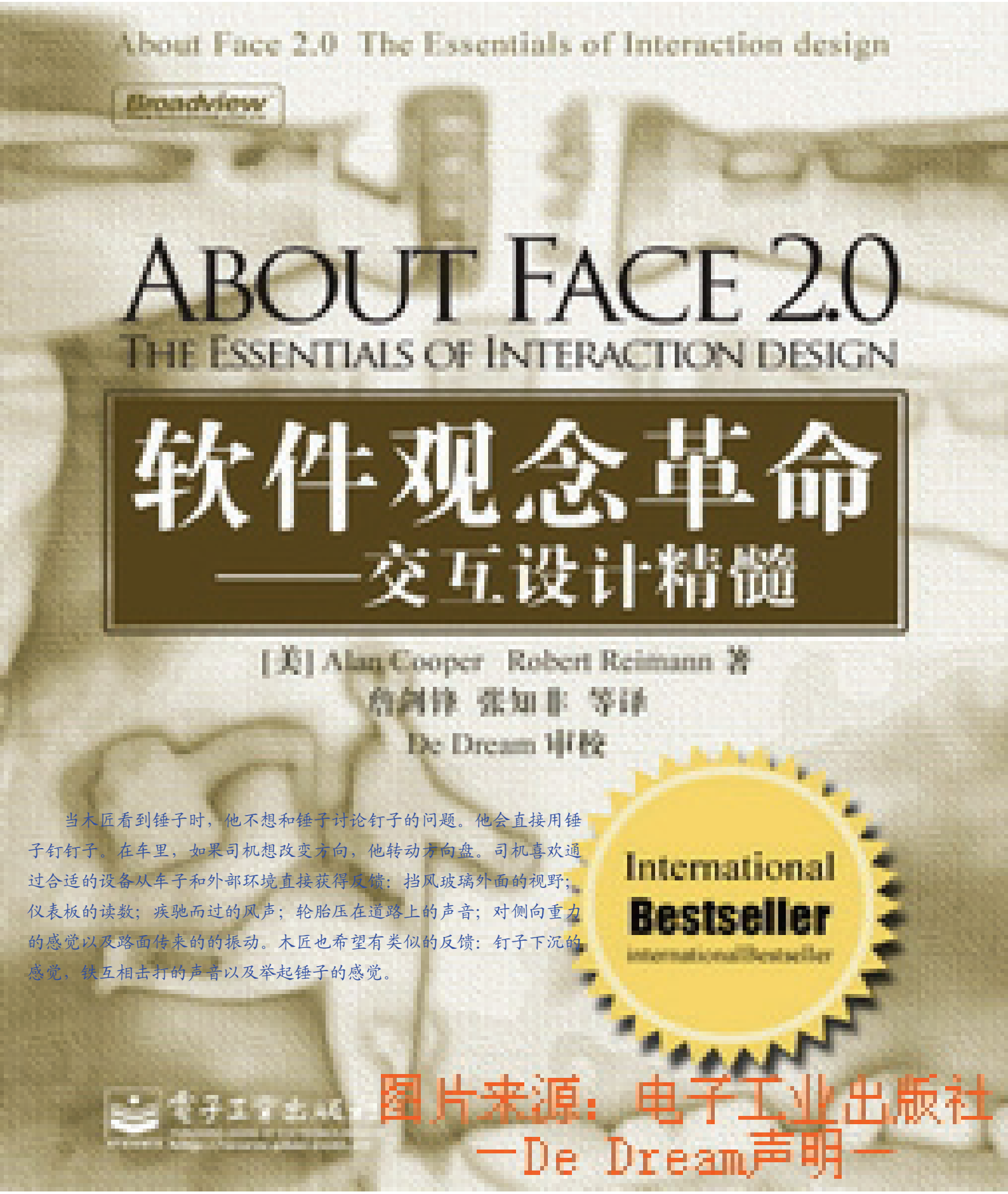
4 月，Telelogic 公司收购了 Popkin 以及后者的企业架构和过程建模工具。这些工具目前还不会包括在新版本中。

Telelogic 希望在 6 月 17 日发布这些集成的产品。

（自 adtmag，袁峰 摘译，不得转载用于商业用途）

征稿

http://www.umlchina.com/xprogrammer/xprogrammer.htm



UMLChina 辅助教材

国内首个交互设计网站 De Dream'

敏捷软件开发调查－－改进项目的方法

Gocept 著，李静 译

8 改进什么，如何改进

使用什么方法可以改进我们的开发过程？鉴于我们是由2-5位互相合作的开发者组成的团队，我们不需要过多考虑流程和仪式。这也正是我在这项调查中强调真实的开发实践和与开发相关的管理实践的原因。

8.1 软件质量

这项调查的目标就是提高软件开发的质量。为什么目标是这个？因为高质量意味着高生产力。象价格，范围，交货期一样，质量通常也是一种可以交付的属性。从市场/客户的角度看这种观点是正确的，但是从开发者的角度来看[4，p.19-23]，这就是不正确的。每个人都有自己的质量标准，我们都更愿意把自尊和我们交付的产品质量联系起来，而不是和我们交付的产品数量联系起来。从这一点可以推断：故意降低质量标准并且强迫开发者把低质量的软件交付给客户，会直接伤害他们的自尊心。交付低质量的软件不是一份令人愉快的工作。因此提高质量可以改善开发者的满意度。

如果你的开发者是满意的，那么他们会开发出更多的产品。一项研究报告显示高满意度的团队可以提高50%的生产力[39]。这在某种程度上导致允许开发者定义待交付产品的质量[4]。

8.1.1 质量是什么？

直到现在我还是以一种非常抽象的方式来使用质量这个术语，因为每个人对于质量的含义都有一个模糊的感觉。ISO9126 定义了软件质量方面的内容。它把质量划分为六类：功能性，可靠性，可用性，效率，可维护性和可移植性。

为了评估特定的开发实践，我会密切的关注这一分类：

功能性（适宜性，精确性，安全性）：

质量的标志之一就是产品能够满足客户的需要。这点格外重要，因为我们要针对客户需求开发特定的解决方案。如果客户需要钻孔机，而我们交付的是锤子，那么客户可以狠狠的责备我们。毕竟软件的价值不在于它是什么，而在于它能做什么［29］。

不幸的是，客观地估量一个软件产品是否满足了客户的需求是很困难的，至少对于我们来说很困难。Alan MacCormack[24]在一项研究中指出早点发布低功能性 Beta 版本给客户可以提高质量。即使发布的 Beta 版本的数量对于质量没有关系，我还是建议多与客户联系（通过让他们测试系统）以此来提高质量（就适宜性而言）。

可靠性（成熟度，容错性，可复原性）：

软件越成熟就越不容易出错。成熟度和软件因本身的缺陷（如缺点，错误…）造成软件失败的频率有关。错误存在于软件开发过程。如果你做一些事情就不可避免地会出错。很明显地，一个系统的缺点越少越好。当系统投入使用时，缺陷率就会变得非常重要。软件必须能够操纵错误的或者被误用的数据，能够从故障中恢复过来。

可用性（易懂，易学，易操作，友好）：

现在，可用性成为非常重要的问题。同时可用性包括容易获得的标准和指导（对于我们而言相关的指导如下：1. Section 503 认证：美国国家标准要求非专业人员可以获得联邦机构的电子和信息技术。2.WAI-AA/WCAG 认证，W3C“无障碍网页内容可及性规范”。尽管到目前为止，可及性还没有成为我们任何一个项目中的需求。）。可用性重要是因为用户必须能够使用软件。可用性不容易实现，有很多关于可用性的书籍。Joel Spolsky（Joel Spolsky 是纽约的一个软件开发人员，他在 Microsoft, Viacom,和 Juno Online Services 供职，2000 年他创立了自己的公司。他也写了一本关于可用性的书。）指出程序严格按照用户需求执行，用户界面也应该被精心的设计[43]。所以我认为多和用户沟通会取得改善，结果可能不是很显著的,但是会是合乎情理的。

效率（时间，资源利用，可测量性）：

我们有三百个开发人员，我们可以做吗？我们需要输入 30000 页…这种事情是我们真正要问的。Zope 在这方面对于我们有很大的帮助，它有很杰出的聚簇技术和一个可升级的数据库后端。在实现的部分，它没有给我们提供明显的帮助。我记得一个事件，当增加越来越多的目录对象的时候，目录对象的显示变得非常慢，

大约每 20 条需要 60 秒。但是我们当中没有一个人试着增加多于三条或四条。所以我们并不知道它很慢。结果遭到客户抱怨。经过一些调查，我们发现解决这个问题非常简单。那些情况不应该发生。这就是效率的含义。（注：可测量性不是 ISO9162 中对于效率定义的一部分。）

可维护性（分析性，可变性，稳定性，可测性）：

在某些情况下客户会改变他的想法，如果系统也容易改变那么将会有很大的益处。可维护性使得开发出的软件可以通过修改适应其他的项目。这对于降低成本很重要，换言之可以提高开发速度。

可维护性重要还在于它影响“变更的成本”。随着时间增加，“变更的成本”会增加，但是增加的幅度取决于我们。传统的开发方法和敏捷软件开发方法的“变更的成本”有很大的差距。使用正确的敏捷开发方法是关键，这一点我会在 8.1.2 部分中进行解释。

可移植性（系统适应性，可安装性，共存性和可换性）：

我们正在使用 Python 和 Zope 开发软件。因为 Python，和 JAVA 类似，开发出的软件独立于位元码，位元码在执行时需要编译。我们开发的软件可以在所有的主要的平台上运行。大部分应用软件在 Zope 应用程序服务器上运行，Zope 应用程序服务器可以进一步降低对操作系统和硬件的依赖性。你可能还需要花费一些精力关注其他的方面（例如路径和文件操作），但是这些都是小问题。

8.1.2 变更的成本

在传统的瀑布模型中，“变更的成本”随着时间显著增长（图 8.1 最左图）。在需求分析阶段“变更的成本”是 e1，在分析设计阶段“变更的成本”是 e10，编码阶段“变更的成本”是 e100，测试阶段 e1000，投入生产后 e10000+。了解了这点，为什么分析师，设计师和程序员都尽力猜测未来可能发生的变化而且都尽力把所有可能需要注意的事情都控制在设计阶段就非常显而易见了。

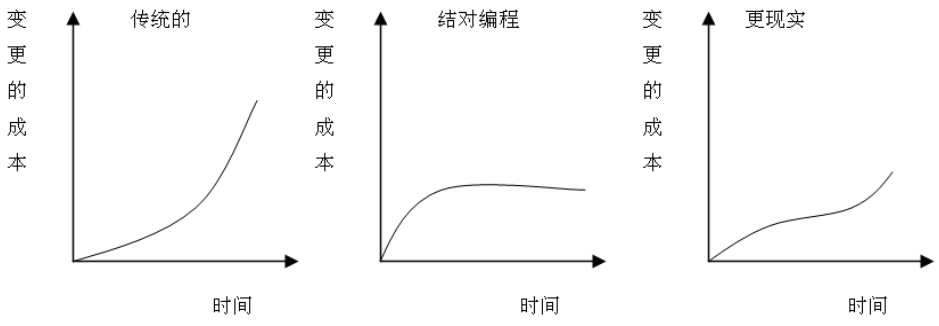


图 8.1 传统方法、极限编程、敏捷方法预计的变更成本[33]

相反，Kent Beck 认为“变更的成本”不会随时间而增加，在极限编程[2]的前提下，它是一条平的“变更的成本”曲线。他这么认为是因为 XP 使得反馈缩短到了数分钟，而使用传统的开发方法需要数月或数年。他坚持认为 XP 把短的反馈周期，简单的设计和多样的实践有机结合从而使“变更的成本”曲线几乎是平的。尽管没有以论文的形式或其他研究的形式提供证据支持这个观点—仅仅只是基于 Kent Beck 的经验。

Scott W. Ambler 在他的文章“调查变更的成本”[33]中提出一条更现实的“变更的成本”

曲线（见上页图 8.1，右图）。短的反馈周期，简单的设计，测试优先的开发模式，关注高质量的代码使得“变更的成本”确实降低了，但是并不是平的。成本还是会逐渐上升的。随着时间推进，系统成长，必须不断的改变。而且改变非代码的产品，比如用户手册，需要更多的努力，因为它们不像代码产品那样灵活。诸如创建 CD 等配置问题也会增加后期的成本。

总结，如果采用了恰当的实践，在很多情况下使用敏捷软件开发方法产生的“变更的成本”会比传统的瀑布模型上升的慢，但是这种变化不会是平的。把更多的努力放在设计方面（创建一个简单的设计）然后不断的重构。

8.2 评估开发和管理实践

在第 9 章和第 10 章，我将讨论一些开发实践和与开发相关的管理实践，这些有助于提高软件开发质量。我不会比较那些实践，这就像拿苹果和桔子比较一样。比较本身暗示着通过一个和另一个对比来得出结论，在这里我的意图不是通过一个实践和另一个实践比较来评估实践活动。

我们旨在就上文提到的六个变量：功能性，可靠性，可用性，效率，可维护性，可移植性，弄清是否特定的实践有助于提高产品的质量，为了得到一个更好的总体看法，我会用表 8.1 中列举的符号，评估每一个方面。

++	非常有助于提高质量
+	有助于提高质量
o	对于质量提高没有帮助也没有不良影响
-	不利于维持合理的质量水平
--	不可能维持一个合理的质量水平

表8.1 实践等级

8.3 客户满意度

这项调查的另一个目标就是提高客户满意度。通过上文对于质量的定义，我认为单独的看待客户满意度是没有意义的。原因如下：

质量重在最大程度地满足客户需求。至少从开发者的角度来看，对于我来说客户满意度是关键。开发者所能达到的最好程度就是满足客户需求。剩下的就是传统的管理，如经常联系和建立信任关系。但是那些传统的管理任务不在本调查考察范围之内。

8.4 内部实践评估

在我对各种各样的实践进行评估的过程中，我所做的另一件事情就是给我们的开发者分发一份调查问卷。对于每一个需要考虑的实践，我问了四个关系密切的问题（加引号）：

有用性：“这项实践多么有用？”

适用性：“你可以多好地使用它？”

实际使用：“你让它发挥了多大的作用？”

接受：“你喜欢它吗？”

评估的范围限定在 1 （最低，最差）到 5（最高，最好）。不幸的是样本很少：只有四个参与者。在接下来的各种各样的实践的评估部分，调查的结果分别列出。第 76 页的 11.2 部分有一个总结。附录中有本调查的原始数据（从 81 页开始）。

9 开发实践

在这一章我会引入一些敏捷开发实践，然后看看它们是否对产品质量有影响以及它们如何对产品质量产生影响。

9.1 结对编程

两个人的智慧比一个人大，四只眼睛比两只眼睛更明亮。– Laurie Williams[25]

结对编程是一种技术，两个程序员共同开发一个产品[25]。两个人用一台电脑工作。包括设计，算法，编码等。结对编程不是一个人编程另一个人观看。Kent Beck 指出[2]，这就像观察小草在沙漠中死亡一样有趣。例如一个程序员输入程序，提出一个方法--他从战术上考虑。同时另一个程序员考虑如何使类适合整个环境—他从战略上考虑[32]。

两个人多多交流是很重要的。特别是输入程序的人必须解释他所使用的概念。否则另一个人无法了解他的思路。他们需要一起编程---结对编程是一个对话。

结对编程是极限编程的核心技术之一。

9.1.1 调查

尽管结对编程已经显示对有质量的软件交付有影响，缩短软件投放市场时间，但是从未使用结对编程的人仍然不愿意使用它。多年来程序员已经习惯于单独工作。管理者经常认为结对编程浪费时间。“为什么一个人就可以做的工作还要安排两个人？我支付不起。”平均来看，结对编程确实比单独编程要多用 15%的时间。

但是 Laurie Williams 的一项研究[25]显示结对编程在以下方面有很多优点：

- 合作提高了问题解决的流程；
- 产品的质量显著提高，由于持续的代码回顾，减少了 15%的缺点，缩短了 20%的代码；
- 开发者更关注任务，例如有更好的纪律，减少了分心因素（结对编程时不可能读邮件或者做类似的事情）；
- 程序员对他们对代码更自信；

- 程序员更满意。

功能性	+	代码会有更少的安全问题。
可靠性	++	研究显示使用结对编程减少了 15%的错误。同时两个程序员比一个程序员可以看到更多的潜在失败点（就容错性和可修复性而言）。
可用性	。	
效率	+	当代码非常低效时，第二个程序员很可能干涉。
可维护性	++	减少了 20%代码
可移植性	+	两个程序员可能更多了解不同的系统。

表 9.1 实践等级：结对编程

所有这些的效果就是，你的团队更可能变成胶冻的。Tom DeMarco and Tim Lister 在《人件》[4，123-128]中指出一个胶冻团队是由一组坚固的联结在一起的人组成的团队，整个团队大于各个部分之和。一旦团队成为胶冻的，这个团队生产力更高，而且生产出的软件质量也更高。

Laurie Williams 在她的研究中使用了一些没有经验的开发者（高年级的研究生）。批评了香港大学的一项由有多年工业经验的开发者执行的研究。Laurie Williams 的这项研究显示结对编程只有在开发者没有很深的领域知识的情况下才会更有效。如果开发人员有很深的领域知识，那么最好是让两个开发者单独工作[31]。

9.1.2 gocept 的经验

结对编程	Min	Avg	Max	Var	Med
这项实践多么有用？	4.00	4.75	5.00	0.25	5.00
你可以多好地使用它？	3.00	4.00	5.00	0.67	4.00
你让它发挥了多大的作用？	1.00	2.50	3.00	1.00	3.00
你喜欢它吗？	3.00	3.75	4.00	0.25	4.00

表 9.2: 实践调查: 结对编程

在过去的半年中，我们对于一些项目在一定程度上使用了结对编程。所有这些项目都在满足客户需求方面获得了成功。

在 L 项目中，我们从项目初期就使用结对编程。程序员双方能够就所有细小的问题展开讨论,这（对于项目进展）非常有益。我们可能会遇到这种情况，就是我们无法明确地知道需求是什么样子。只有我们不知道。我们讨论，开发，进一步更深的讨论，我们会发现同伴对一些问题有不同的意见。事实上客户持有另一种意见。结对编程使我们关注与彼此的交流。

另一个优点是结对编程的压力。Laurie Williams[25] 认为“结对编程压力”以一种来自同伴的积极的压力的形式存在。对于任务我们会持续的维持高的集中力，例如，我们会不断互相提醒请首先写下测试。

不可否认，两个人一起测试程序非常困难，特别是当程序缺陷是由于误解了应用程序接口或者程序缺陷是来自应用程序服务器的错误的时候。在上述情况下，你不得不测试系统，在某种程度上依靠感觉判断程序在哪里出错。这种模糊的感觉很难交流。

考虑结对编程在简单任务上的应用，研究的结果也支持了我们关注的问题。如果结对的两个成员都了解某领域，分开进行会提高效率。特别当建立简单的用户界面的时候，这在我们的例子中，意味着较多的采用 HTML，CSS 或相似的技术，这种情况下结对不是非常敏捷。虽然不使用结对编程，质量不会下降，但是对于结对编程还是要予以关注。

如果两个人中任何一个对于目标系统的了解不够，那么进行结对编程几乎是不可能的。在我们的例子中，我们的结对组成员中，一个人对于 Plone 没有任何了解，另一个有非常深的了解。这对开发人员无法开展工作，它更是一种老师—学生的关系。这不是结对编程显而易见的本来面目。特别是在两个结对成员都没有很多结对编程经验的情况下。现在还不是很明确是否需要建筑沟通知识鸿沟的桥梁。

综上，结对编程是一种提高软件开发质量的方法。正如你在上页表 9.2 中所看到的一样，它成为一种被普遍接受的实践方法，并且被认为是有用的。但是我们还没有在所有的项目中都应用它，我们原本应该那样做。

9.2 自动化测试

测试是软件开发过程中没有人愿意谈论的部分，尽管每个人都知道测试的重要性。为什么测试如此重要？为什么自动化测试这么重要？

当你写了一些代码之后，通过测试你才可以唯一的确定它是否发挥作用，换言之对代码进行测试。你可以通过操作用户界面手工测试代码（可能你现在并没有这么做），或者可以通过解释程序来手工测试代码（可能没有基于你所使用的语言的）。但是这样做需要时间。当测试需要很多时间的时候，你不可能对所有代码的全部功能都一一测试。为了能够始终确保代码发挥作用，你需要自动化测试为你测试。

既然我们已经知道我们需要自动化测试，当我们写测试的时候问题出现了。传统上首先你写代码，然后一些测试员写测试代码。一个非常有吸引力的方法是测试驱动开发—即先写测试后写代码。

9.2.1 测试驱动开发（TDD）

在没有测试用例的时候，不要写功能代码。--kent beck

先写测试？也许会有别的重要的东西，但是先写测试是最重要的一点。

测试驱动开发遵循四个步骤：

1．写一个测试

你要了解需求并且需要知道你如何实现需求。你为一个方法或者一个功能写一个测试。这个测试要能模拟方法的功能。

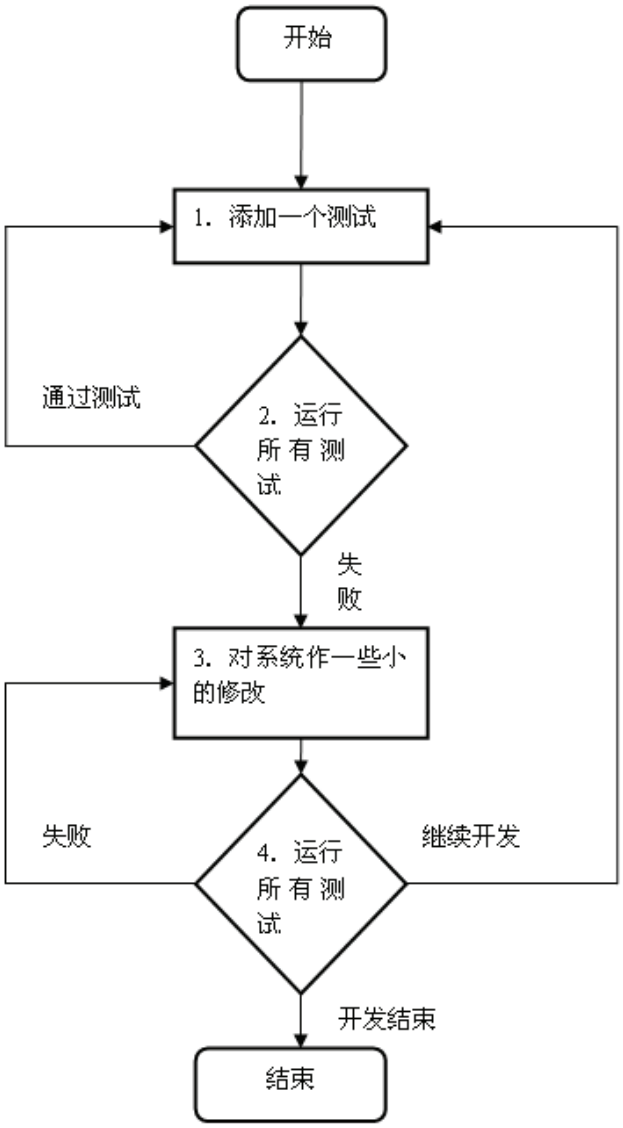


图 9.1 测试驱动开发的步骤[36]

2．运行测试

接下来，你运行所有的测试。当然这个新增加的测试会失败—你没有实现任何事情。如果新增加的测试没有失败，那么这个测试用例就是错误的。修改测试或者增加新的测试，也就是回到第一步。

3．实现需求

现在，你实现你为之编写测试的方法，仅仅实现这个方法。

4．运行测试

最后你再次运行测试。如果这些测试没有通过，你需要继续实现需求（回到第三步）。如果它们通过了测试，你就可以结束，否则，如果还有更多需要实现的功能，继续进行第一步。

这种方法在分析和设计方面有很丰富的含义[34]。当你为特定的方法编写一个测试时，你要考虑这个方法在什么情况下会被使用，否则你无法写出这个测试用例。很清楚地，对于这个方法并没有打算实现的功能，不需要编写测试。你还需要考虑方法是如何被使用的，因为事实上在这个方法存在之前就在测试中使用了该方法。在这个阶段，改变方法的预定行为是容易的。

有趣的是测试驱动开发并不仅仅被视为一种测试技术，更被视为是一种开发技术。它在很大程度上有助于提高代码的质量，但是你可能仍然需要验收测试，系统测试和集成测试。但是如果你需要测试驱动开发，你可以在项目初期就进行这个工作[36]。

9.2.2 单元测试

调查

单元测试意味着测试最小的测试单元：一个方法或一个功能。当待测试方法存在已经定义的接口时，单元测试才有意义。接口描述了方法是如何工作的。单元测试核实该方法产生了作用还是没有产生作用。

一个单元测试应该只测试一个单独的方法。如果有可能的话，这个方法调用的其他组件要被模拟，因为这些组件可能会隐藏一些错误，或者仅仅由于它们的副作用使得测试通过。模拟一个组件意味着创建这个组件的最小版本，这个模拟的组件只需要做测试需要它做的事情即可。

当你采用测试驱动开发的时候，大部分情况你要写单元测试，但是单元测试并不仅限于测试驱动开发，你也可以在以后写测试。事实上，每一次当你遇到错误，或者更糟糕的是用户遇到错误的时候，你应该写一个单元测试。原因在于：1.可以证明错误确实存在。2. 当测试通过以后你可以确定错误已经修复。

与功能测试或者验收测试不同，所有的单元测试都必须始终通过测试。大部分的项目开发组中，如果没有通过单元测试，那么任何东西都不允许登入系统。

功能性	+	代码更准确，至少在测试过的边界处。
可靠性	++	在系统作为产品前可以发现很多错误。
可用性	。	
效率	。	
可维护性	++	如果你要写测试，那么系统本身需要是可以测试的。单元测试为开发背景的系统有更多联结的组件从而使系统更易变更。
可移植性	+	在写测试的同时，很可能这个测试就是具有可移植性的。

表 9.3 实践等级：单元测试

Gocept 的经验

到目前为止，我们已经使用单元测试很长时间了。大约半年前我们开始有几分强迫的对新的产品或多或少的进行全部的测试。

单元测试	Min	Avg	Max	Var	Med
这项实践多么有用?	5.00	5.00	5.00	0.00	5.00
你可以多好地使用它?	4.00	4.25	5.00	0.25	4.00
你让它发挥了多大的作用?	3.00	3.75	5.00	0.92	3.50
你喜欢它吗?	4.00	4.50	5.00	0.33	4.50

表 9.4 实践调查：单元测试

表 9.4 表明我们都认为单元测试非常有用。这点非常有趣，因为我们都非常不愿意写单元测试（我没有任何有力的事实来证明这点，我只是记得对于我来说让任何一个人写测试是多么的困难）。这和我们所使用的 Zope2 环境有很大关系。如果对 Zope 不是足够了解那么写单元测试确实非常困难。Zope 2 的问题在于它不可能模拟组件，因为它没有一个组件机构。这导致单元测试更像一个功能测试。但是我们已经恢复过来，而且现在我们不仅仅认为单元测试有用而且乐于做测试。

9.2.3 验收测试

调查

验收测试（或者功能测试）用来检验是否一个特定的用例，用户素材或者其他高层的需求被恰当的实现。和单元测试不同，验收测试测试整个系统，包括所有相关组件［37］

与验收测试相关的一件重要的事情就是客户需要明确表达出测试内容，他们自己才是确定软件是否可以接受的唯一的人选。进而程序员通过编码将验收测试纳入自动化测试。不幸的是这项工作非常困难，特别是当涉及到用户界面的时候，而用户界面往往总是涉及到这项工作。

功能性	+	如果用户明确阐明测试，开发者能够更好的知道应该做什么
可靠性	。	
可用性	+	用户明确表明测试应当包括他们的界面的部分.
效率	。	
可维护性	–	非常小的变更，而不是明显的原因，也会带来很多的测试。这种情况也可能是由于测试不够抽象造成的。
可移植性	。	

表 9.5 实践等级：验收测试

Gocept 的经验

在我们的工作中测试用户界面意味着测试网络应用。测试员的任务就是分析已经形成的 HTML 网页的内涵，而不是它们的形式。

用户界面的自动化功能测试	Min	Avg	Max	Var	Med
这项实践多么有用?	4.00	4.75	5.00	0.25	5.00
你可以多好地使用它?	2.00	3.00	4.00	0.67	3.00
你让它发挥了多大的作用?	1.00	2.00	3.00	0.67	2.00
你喜欢它吗?	3.00	4.00	5.00	1.33	4.00

表 9.6 实践调查：用户界面的自动化功能测试

对于我们来说完成这个任务几乎是不可能的。很多次，我们试图写用户界面测试，但是这么做不值得----即使我们认为这样做很有用。

9.3 简单设计

往往简单的设计是最好的设计。设想布莱尔(Marcel Breuer)的“温索里椅”(Wassily chair)：几乎是 80 年前的设计，但是现在还在被广泛的使用——它很干净也很整洁。这和软件是如何相关的呢？

9.3.1 调查

简单设计意味着对于系统设计尽可能始终保持简单。做可以完成工作的最简单事情，设计只需要完成现在的需要，而不需要完成明天的需要。

当然这样说还是很模糊。为了更清楚一些，下面是 Kent Beck 对于简单的理解[2，pg109]：

1．系统（代码和测试）必须对你所要交流的每一件事情进行交流。

2．系统必须没有冗余代码。

3．系统应该包含最少可能的类。

4．系统应该只包含最少可能的方法。

前面的项目存在优先顺序。这是很重要的。最好的设计绝对不是类最少的设计。这会导致非常庞大的类。Beck 把交流视为最重要的问题，因为要尽一切可能使系统易懂。1 和 2 共同表述了极限编程中的“确保信息传输,并且是一次且仅一次的传递”规则（once-and-only-once rule）。

功能性	+	直接关注需求
可靠性	++	简单的事情更可靠。即使可能有人会说简单的软件容错性差或者可恢复性差---如果需要高容错性或高恢复性的话，简单设计可以要求实现这样的功能。
可用性	。	
效率	。	
可维护性	++	易读和易理解的代码易于分析和修改。无代码冗余的要求也有助于可维护性的实现。
可移植性	+	简单设计意味着很多小的组件。如果需要的话，组件可以被移植。

表 9.7 实践等级：验收测试

然而简单设计不意味着少的工作。建立复杂的设计很容易，建立简单设计需要更多的思考。你需要考虑设计的本质是什么，需要去除所有的冗余。为什么要进行这些额外的努力？简单设计是减低变更的成本（参见 51 页 8.1.2 部分）的方法。现在多思考一点可以使以后系统的变更减少，在很多情况下，这是值得投资的事情。

简单设计	Min	Avg	Max	Var	Med
这项实践多么有用?	3.00	4.00	5.00	0.67	4.00
你可以多好地使用它 ?	2.00	3.50	4.00	1.00	4.00
你让它发挥了多大的作用?	3.00	4.00	5.00	0.67	4.00
你喜欢它吗?	2.00	3.50	5.00	3.00	3.50

表9.8 实践调查：简单设计

9.3.2 gocept 经验

评估结果显示简单设计被认为非常有用，但是实际应用中并不像通常认为的那样。

我们做的很多的事情是“为现在的需要设计，而不是为明天的需要进行设计”。我和 Christian Theune 对这一点展开了多次讨论。他认为我们所做的看起来更像是“如果情况变复杂，那么就停下来思考”。他认为这样是低质量的。下面我们就这个问题进行深入探讨。

Christian 对于简单设计的讨厌情绪很大程度是源于他低质量的感受。危及产品的质量会引发很强的情感，因为对于产品质量的攻击常常被认为是对于人自尊心的攻击。意识到简单设计并不意味着低质量是很重要的。质量应该是很高的。如果你现在不需要 X 特征，则不用现在就建立 X。当然这么做建立在如下基础上：就是以后增加 X 的成本不会比现在增加 X 的成本高很多，而且现在不能确定以后是否需要增加 X。

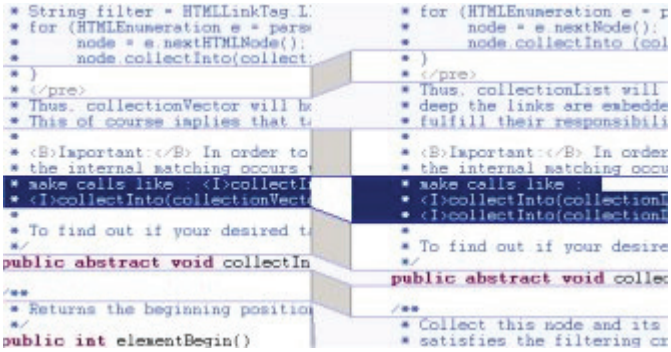
为今天设计意味着有意不去考虑一些因素。不考虑一些因素意味着不在细节上对这些因素作思考。也可能根本就不思考。通常，这样做违背了程序员的本能。程序员通常会提前作计划：预计问题是他们的习惯。如果出现一个问题，看到自己的程序可以解决这个问题，程序员会很开心。当然这个问题可能根本不会发生。虽然这种“猜测未来”的战略会是不成熟的。不幸的是看起来对于敏捷的人[2，p104-105]来说这很困难。

为了阐明在实践中“为今天的需要设计”是如何发挥作用，下面是来自我们的项目的一个小的报告：

对于一个项目，除了其他一些特征以外，我们需要一些高级的团队管理功能。我们建立一个粗略的团队管理模式和一些其他的特征。首先讨论设计和这些特征的优先级，不考虑团队管理。随后我们会考虑细节，考虑如何整合它。当需求有细小变化。如果我们需要团队管理，我们去除一些大的方面，重塑团队。当考虑了一些其他特征以后我们可以找到更好的团队管理的方法。--CZ 关于 MP 项目。

很难说，简单设计对于我们的项目有多大影响。确定现在的设计是否是最简洁的几乎不可能。但是拥有最简洁的设计并不是最重要的事情。比较简洁就好，直到有一天你会发现你已经拥有了最简洁的设计。保持简洁设计的信心是很重要的。

9.4 持续集成



9.4.1 研究

持续集成是极限编程中的另一种技术。它在非极限编程领域也被广泛应用（尽管并不以“持续集成”这个方式称呼）[44]。它意味着另一个开发特征的主要转变：编写/测试代码之后，代码就立刻成为全部系统的一部分，没有一行代码是非集成的。

功能性	。	
可靠性	++	更容易发现集成错误.
可用性	。	
效率	。	
可维护性	+	代码更易于清晰：你每天都要显示代码。
可移植性	。	

表9.9 实践等级 :持续集成

持续集成是如何发挥作用的？

当你完成了一个特定的开发任务之后，你进行转换使代码集成。你更新你的备份，记录源代码库（代码库是源码的集中存放处的。这样的库通常和代码版本控制器连接在一起，版本控制器允许标记，分支，合并，有助于开发员找到冲突。我们现在使用的是 CVS。它的增强版（svn）也是一个可备选则的代码版本控制软件）的变化。库的变更将与你的文件合并——大多数情况下是自动的。当自动合并失败的时候，你需要人工解决冲突。更新之后，你需要确保所有的地方都如所预期的一样运转（如运行测试）。

持续集成的好处是什么？

当引入了一个错误的时候，如果其他人马上使用了你的代码，那么你（或者别人）可能当天就可以发现这个错误。可能你一整夜的努力都白费了。不仅这样，而且因为系统的变更非常小，所以搜索错误发生的范围也被显著缩小了。这降低了搜索和消灭错误的时间。

更重要的是，当你开发一个新的特征，你往往误以为你是唯一一个接触代码的人。这样你会完全集中到开发本身。如果你考虑到同时其他人也在改变代码，那么会分散你对现行任务的关注。进而，当你将你做的变更进行集成时，你才可以完全集中于此。

首先持续集成使我们更容易关注且仅关注一个任务。当你完成这个任务之后，你集成你的代码发散你的思维。在大脑中集中一个任务比集中很多任务要容易很多。当你达到“自然迸发”的思维境界[2，p97-99]时你再去集成。

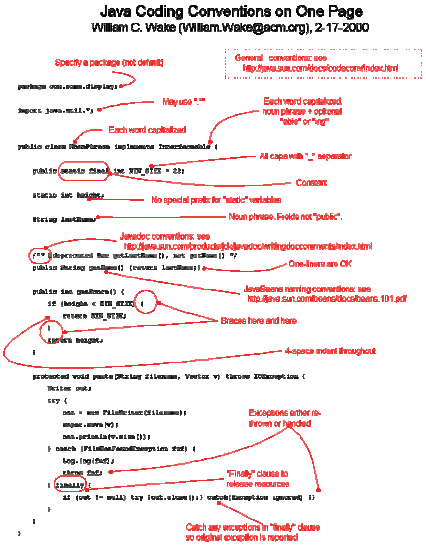
9.4.2 gocept 的经验

一些人建议应该有一个单独的电脑用来集成。原因在于只有唯一的集成点，才可以避免集成的“空中相撞”。我们没有这样的电脑。

我们鼓励每个人在开发的代码足够稳定后就核查。这个核查在某种程度上可能会打破其他开发人员的代码。但是每一个人都在第一时间使用，然后测试最终的代码。创建一个发布版就成为一件单纯标记所有文件然后根据这些文件创建档案文件的事情。

对于持续集成我没有分发调查问卷。

9.5 编码标准



9.5.1 调查

编码标准很重要。所有的程序员都被允许改变系统的每一个部分，这时你没有能力让每个人都使用他自己的标准。尽管每一个程序员自己的标准都是最好的也都是唯一的。必须有一套团队中每一个人都承认的风格上的指南，你不能强迫任何一个程序员使用一个对他毫无意义的风格指南。程序员必须使用对他们有益的编码标准。

引用 Guido van Rossum 对于 Python 代码[55]的风格指南：

风格指南是关于一致性的指南。风格一致很重要。项目一致性更重要。模型或者功能一致性最为重要。

一致性使得代码更易读也更易懂。如果你有一个一致的代码风格，实现类似功能的代码看起来就是相似的。获得可以理解的代码是重要的。通常情况不存在独立于代码的文件（即使有，通常也不是最新版的）。程序员更喜欢把代码作为原始文件，但是只有代码可以理解才可以发挥出文件的作用。一致性是易懂的一个步骤。

一致性也使得代码易重构易维护。当你对于一致的部分感到很安适很自在的时候，你就不会被不熟悉的结构打扰，就能集中于现行的任务。

再次引用 Guido van Rossum[55]的观点：

但是最重要的是：知道何时要不一致——有些时候风格指南不能够应用。当产生怀疑的时候，使用你最好的判断。看看其他的例子然后决定哪个风格最好。不要对提问产生犹豫。

在某些情况下可以违反编码标准，这点很重要。如果一个特定的标准总是被违反，那么就是该改变标准的时候了。

功能性	。	
可靠性	+	编码标准应该暗示主要的可以做的和不可以做的从而第一时间阻止错误
可用性	。	
效率	。	
可维护性	+	代码的感官上的一致性可以提高可读性和可理解性。如极限编程中“once and only once ”规则有助于提高可维护性
可移植性	。	

表9.10 实践等级:编码标准

9.5.2 gocept 的经验

在 gocept 我们至今还没有自己的编码标准。我们多多少少的使用一些基于 Python 代码[55]的和 Zope 3 代码[56]的标准。

拥有以上两个标准，是否有自己的标准就显得不那么重要。然而我们需要有一个共同的基点，因为在不同的代码风格下工作是很令人烦恼的。

10 管理实践

在这一章，我将分析一些与软件开发密切相关的管理实践，这些实践可以很容易应用于每天的工作。

10.1 即兴表演

“即兴表演”（Jam Sessions）背后的含义很简单：给开发者提供他可以不受干扰的时间框架。这种“即兴表演”（Jam Sessions）的想法的来源很难说清楚--对于我们而言，它来源于Bert Schulski[46]的一篇论文。

10.1.1 调查

著名的著作《人件》已经出版很多年了。从 1987 年，《人件》第一次出版开始，Tom DeMarco 和 Tim Lister 就在倡导无干扰的工作。相关的章节在《人件》中被称为“脑力时间和体力时间”。这是非常中肯的。每一次你被打扰，你都丧失了有价值的时间——不仅仅是被打扰的这五分钟，还有你要重新回到工作状态的时间。通常五分钟的打扰损失了 20 分钟的工作时间。你的人在工作，但是你并没有真正的工作。

为什么会这样？当你集中精力关注你的工作任务时，你处在一种英文中称为“顺流（Flow）”的状态。Flow 是一种思维完全的活跃的关注一个活动的状态，通常伴随着高水平的工作乐趣和责任感[47]。达到这种状态需要时间——15 分钟或更多的时间。每一次打扰都会使你远离这种状态。

即兴表演就是一种减低打扰的手段，因此可以增加脑力时间。你要规定一些固定的时间框架，在这段时间内所有的干扰——外部的或内部的——都被阻止。这意味着没有电话，没有邮件，没有任何事情干扰。只是代码，当然，只有当这些时间框架被公司中每一个人接受的时候，这种即兴时间才有意义。

功能性	。	
可靠性	+	对于任务的高的关注有助于减少可能的错误
可用性	。	
效率	+	更多的脑力时间使得时间最优化。
可维护性	。	
可移植性	。	

表10.1 实践等级 : 即兴表演

在实际计划中，“即兴表演”不要求太多—— 只需要一个每个人都可以看到的日历和每周一次的对于下周的简短计划就可以。图 10.1 是一个周计划的例子。

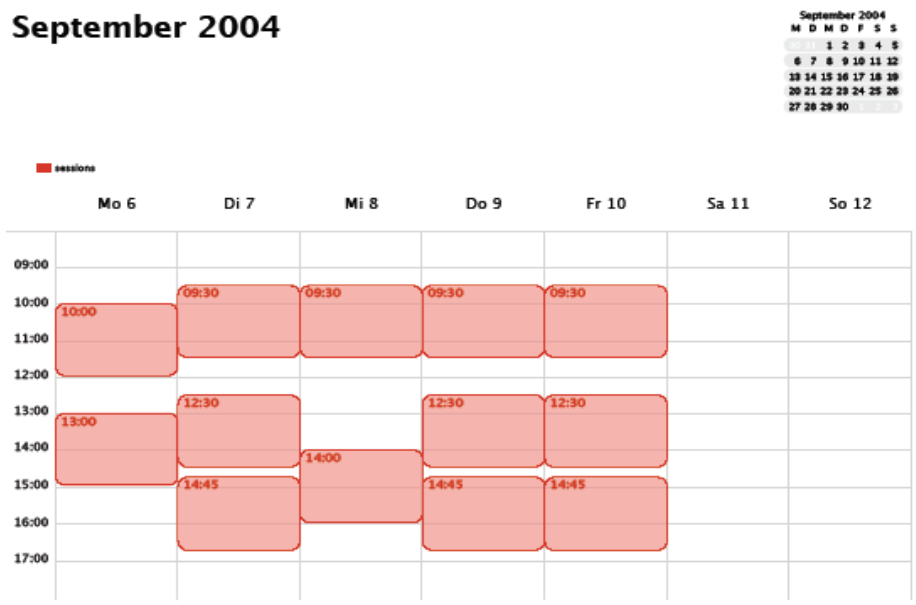


图10.1 “即兴表演”日历

10.1.2 gocept 的经验

尽管我从不认为在 gocept 的工作环境很嘈杂，但是“即兴表演”还是被视为一个最有用的技术，这一点是值得关注的（见表 10.2）。但是不能对此评价过高，当使用“即兴表演”这个技术的时候，我们大部分也使用结对编程，这样会有一些抵消作用。

即兴表演	Min	Avg	Max	Var	Med
这项实践多么有用?	5.00	5.00	5.00	0.00	5.00
你可以多好地使用它?	2.00	3.25	4.00	0.92	3.50
你让它发挥了多大的作用?	1.00	2,50	3.00	1.00	3.00
你喜欢它吗?	3.00	4.25	5.00	0.92	4.50

表 10.2:实践调查：即兴表演

我们在执行“即兴表演”的时候存在的一个问题就是，团队成员会有其他的约会或者要出席其他会议（注：一些员工可能还是学生）。这使得无干扰时间的确定很困难。另外，“即兴表演”也要求我们高效使用剩余的时间。

10.2 短的发布周期

10.2.1 调查

短的开发周期对于软件成功很重要，它可以持续地将功能非常简洁的版本提供给客户。Alan MacCormack[24]的一项研究指出如果客户可以较早测试较低功能的软件版本。最终产品会有较高的质量。MacCormack 也指出如果提供给客户的软件的测试版包括了全部的功能，那么软件的质量会很差。MacCormack 认为更重要的是这个参数（短的发布周期）可以解释大于 1/3 的产品质量变化。

有趣的是 Alan MacCormack 的研究没有显示出测试版的数量并不影响产品质量。

几乎所有的敏捷方法都提倡短的发布周期。XP 以 2—4 周为一个阶段，Scrum 以 30 天为一个迭代周期，水晶方法（Crystal Methods）强调频繁交付。每隔几周或数月就将客户纳入开发。

功能性	++	用户可以经常提供有价值的反馈。
可靠性	+	在某种程度上，交付也意味着固化。
可用性	+	用户可以经常提供有价值的反馈。
效率	。	
可维护性	+	增强发布，升级等问题得知识。
可移植性	。	

表10.3 实践等级：短的发布周期

意识到短的发布周期并不等同于真实的发布软件，这一点是很重要的。发布和可发布之间是有区别的。发布指封装起来在事实上交付给用户。可发布是软件一个已经确定的状态，说明软件可以在短期内封装起来发布。这点很重要，因为当软件因为需求问题还处在可发布状态时，没有必要在软件发布上浪费时间——例如把软件提交给客户。

短的发布周期很重要的另外一点就是 Alistair Cockburn 所指出的“较早成功”[45]。短的发布周期允许开发团队把产品交付给客户，对于交付的产品客户可以表扬或批评。每一种情况都是很有价值的反馈。

10.2.2 gocept 的经验

我们认为短的发布周期非常重要——所获值得付出。不幸的是我们通常没有一个自动化的构建和封装过程。

短的开发周期	Min	Avg	Max	Var	Med
这项实践多么有用?	3.00	4.00	5.00	0.67	4.00
你可以多好地使用它?	2.00	2.75	3.00	0.25	3.00
你让它发挥了多大的作用?	2.00	2.75	3.00	0.25	3.00
你喜欢它吗?	2.00	3.00	4.00	0.67	3.00

表10.4 实践调查:短的发布周期

这使得创建发布比较困难——这也是表 10.4 中“你可以多好地使用它”这一项得分很低的原因。这里给我们的启示就是：引入或者开发一些工具，它们可以简化发布。

10.3 每日项目会议

10.3.1 调查

每日项目会议用来交流项目状况，问题，正在进行的行动，以求完成一系列的目标。有各种不同的会议风格。在 XP 和 Crystal 中是 “Daily Stand Up Meeting”，在 Scrum 中是“Daily Scrum Meeting”。所有的会议共同点就是在同一时间，同一地点，每天，团队所有成员召开一个简短的会议。每一个参加者在会议期间都要积极——例如都需要在会议期间发表个人观点。

会议应该简短，意味着不超过 15 分钟。15 分钟足够每一个人陈述一些事情——但是 15 分钟不留讨论时间。做到这点的办法就是采用一个“直到被问起才发言”的政策，项目管理者是唯一发问的人。如果你有问题，你可以在会后和相关人员讨论。

产生这个会议源于以下想法：当所有的团队成员参加这些短小的会议的时候，对于其他会议的需要就减少到了最低的水平。没有必要召开等级制的会议，大多数参加人坐在一起什么事都不做。每日项目会议可以留出时间做更重要的事情，如理解客户需求，使代码成熟等。即使需要其他别的会议，通常也只是涉及到两个或者三个人，这些会议可以单独召开——可能只在一台电脑面前召开就行，在那里可以浏览代码，可以实验想法[48]。

通常，那些会议召开时要求参加人站成一个圈。这样没有人希望长时间的开会。这种方式存在的问题就是笔记问题。所以要么使用一个剪贴板要么在白板上写笔记，从而使每个人都可以看到。

在 Scrum 中强调管理者也能参加会议——但是他们不允许坐在会议桌旁，也不允许站在参与者中。他们不可以说话，只可以倾听。Ken Schwaber 以一个不是很有趣的笑话的强调了这点：

一只鸡和一头猪在一起，这时这只鸡说“我们开一个餐厅吧”猪想了想然后说“我们给餐厅起什么名字呢？”

鸡说：“Ham n’ Eggs!”

猪说：“不好。我要对餐厅负责，但是名字中只提到你！”

猪们（开发者）都对特定的任务负责——所以他们都会这么说。每个人都是一只鸡。

每日会议的主要益处在于增进交流。这导致对于障碍和需求的较早注意，创造了较早行动的机会。对于开发过程中产生的问题，不是由开发者一个人面对。

为了使这点更清晰：每日会议不是中期或长期的计划。它更多的是关注今天你要做什么事情。

功能性	+	通常，每日会议对于软件质量有影响.
可靠性	+	
可用性	+	
效率	+	
可维护性	+	
可移植性	+	

表10.5 实践等级：每日项目会议

每日会议对于软件质量有直接的影响。每日会议不仅仅要求每个人说出问题是什么，而且还使每一个人都要倾听别人的问题[27，P105-106]。这对于整个团队有积极的影响，因为团队不断的交流有助于个人问题的解决。管理者可以移除障碍，因为障碍被列入会议中。

10.3.2 gocept 的经验

交流获得的收获是巨大的，但是我们对于维持每日会议还存在问题。一个原因是通常一个项目我们会安排 2-3 个人一起工作，他们可能不必通过预定的例会就已经很好的交流了。另一个原因就是每个人都会感觉自己对于目前正在开展的工作不够了解。为了避免这种情况，我们至少应该与公司所有员工召开每周会议，对于公司中每个人在做的事情做一个大略的概述，将结果提供给每个人。

会议	Min	Avg	Max	Var	Med
这项实践多么有用?	3.00	3.75	5.00	0.92	3.50
你可以多好地使用它?	2.00	2.00	5.00	2.25	3.00
你让它发挥了多大的作用?	1.00	2.25	3.00	0.92	2.50
你喜欢它吗	3.00	4.00	5.00	0.67	4.00

表10.6 实践调查 例会

有趣的是，每日会议并不被认为非常有用（从表 10.6 中可以看出）

10.4 现场客户

10.4.1 调查

XP 的核心实践之一就是把一个客户代表整合到开发团队中。Kent Beck 的著作《解析极限编程》出版以后，很多术语发生了变化：现场客户变成了“整个团队”，强调现场客户，客户团队（客户团队包括最终用户，销售力量，文档人员等。它主要是来自客户方（未被列入现场人员）的每一个人。），开发者和质量控制成为开发团队的一部分 （你应该试图把这些尽可能的组成一个整体）[50]。

现场客户必须能够进行诸如回答疑问，提出需求，调整优先权，解决争议等工作。否则他是没有用处的[51]。可以随时接近客户的时候，的确可以减少对于说明书的需要——现场客户就是最好的说明书。从另一个角度看，这个也暗示如果你没有一个现场客户，那么你需要更多的说明书。

功能性	++	让正确的功能被实现是拥有一个现场客户的前提。
可靠性	++	客户可以测试开发的软件。
可用性	++	客户最清楚他如何工作以及他的需求。
效率	。	
可维护性	。	
可移植性	。	

表10.7 实践等级：现场客户

把客户纳入整个开发团队的一些有趣的附加作用：

客户了解你如何工作。对于大多数客户而言，整合到另一个公司是一种全新的体验。他们不可能习惯所有的事情。你要表现出对于项目的真诚的关注，表现出你不会撇下客户自己一个人。这对于建立合作关系（在 JAOO2003 会议上有一个关于提高客户开发者合作的工作组，一份报告值得阅读，参见[49]）。是至关重要的一步。

你了解客户是怎么想的。每天的交互有助于你对他的理解。你逐渐会知道他的习惯和态度——至少会有一点了解。即使他某些时候不在现场，交流也会变得容易些。比起和陌生公司的陌生人交流，和你的新朋友交流显得容易。

客户知道系统如何工作。他或多或少直接塑造了系统。积极的角色认知要求考虑系统，要求在一定程度上了解系统。

让客户一直在现场的变体就是让客户在项目关键时刻在现场 2-5 天。关键时刻是这样的时刻——那时你所有的细小的问题只有客户才可以直接的决定是否应该这样处理，作为一个开发者你无法获得任何关于该怎么做的线索。为了彼此相互了解，在项目初期建立一个工作组是明智的（如[49]中建议的那样）。

功能性	+	这个实践确实有作用，但是让客户全过程在现场效果更好.
可靠性	+	客户可以测试开发的软件
可用性	+	客户最了解他如何工作以及他的需求.
效率	-	当客户在现场的时候你要更关注功能性而不是效率，因为客户只是在很短的时间在现场
可维护性	-	对于功能性的关注可能激发快速的hacks.
可移植性	。	

表10.8 实践等级：客户工作组技术

10.4.2 gocept 的经验

极限编程要求在项目全程中都有一个现场客户。不幸的是这对于我们而言不起作用：到目前为止，没有一个客户能够且愿意投入这样的时间和人力。我们的客户分散在全欧洲，这对于我们来说也是个问题。但是我们对此并没有彻底沮丧。如果处理不当，那么有一个现场客户也会很劳神。而且我们并没有很好的处理能力。因为有限的人力，我们不可能一次只做一个项目，每一个开发人员涉及很多项目（同时陷于多个项目并不是一个好事。但是在小公司你别无选择）。

11 结论

11.1 技术等级小结

表 11.1 总结了本研究的中技术（实践）的等级。

	F	R	U	E	M	P	Σ
结对编程	+	++		+	++	+	7
单元测试	+	++			++	+	6
验收测试	+		+		-		0
简单设计	+	++			++	+	6
持续集成		++			+		3
编码标准		+			+		2
即兴表演		+		+			2
短的发布周期	++	+	+		+		5
每日会议	+	+	+	+	+	+	6
现场客户	++	++	++				6
客户工作组	+	+	+	-	-		1

表11.1 所有技术等级的总结（功能性，可靠性，可用性，效率，可维护性，可移植性，合计由“+”的数量减去“-”的数量；“。”的位置为了可读性保持空白。）

没有一项实践对于质量不起作用。尽管在某种程度上，我期望是那样。我选择这些实践因为它们对质量有明显价值。这些明显的价值在这个评估中得到证明。

本部分的总结显示确实存在一些实践，它们对于提高质量有实质的帮助，它们是单元测试，简单设计，短的发布周期，每日会议，现场客户，其中最重要的是结对编程。其他的实践作用稍微小些，如编码标准，验收测验。

有一个现场客户我们可以获得很大收获。但是像上文讨论的一样，这种情况不可能在任何时间都存在。我们可以不断地安排客户工作组，但是需要确保在工作组期间不会牺牲效率和可维护性。

	Use	Appl	ActUse	Accpt
结对编程	5.0	4.0	3.0	4.0
单元测试	5.0	4.0	3.5	4.5
验收测试	5.0	3.0	2.0	4.0
简单设计	4.0	4.0	4.0	3.5
持续集成		not surveyed		
编码标准	5.0	3.5	3.5	3.5
即兴表演	5.0	3.5	3.0	4.5
短的发布周期	4.0	3.0	3.0	3.0
每日会议	3.5	3.0	2.5	4.0
现场客户	5.0	3.0	1.0	5.0
客户工作组	4.5	3.0	2.0	4.0
总计 (中值)	3.0	4.0	4.0	4.0
总计(平均值)	3.5	3.5	3.8	3.5

表11.2 内部实践评估调查的中值（标题是：有用，适用，实际使用，接受。总计中的中值和平均值通过全部的样本计算，而不是通过表中所列的条目计算。）

11.2 内部实践评估小结

表 11.2 总结了调查数据的中值。我们认为大部分实践有用到非常有用（4.0—5.0）。对于我而言，看到总的中值只有 3 有用，非常惊讶。

此次调查可能会因为不严谨的问题形式而遭到批评。

11.3 结论

23 页的 3.4 部分中，我提到我们存在的最明显的问题。表 11.3 显示了每一个问题相关的质量方面。

问题	相关的质量方面
高的错误率	可靠性
代码难懂	可维护性
代码被复制粘贴	可维护性
很难恢复错误	可靠性，可维护性
客户太远而且很难接触到	功能性
没有和真正的客户接触	功能性
变更系统的成本高	可维护性，功能性

表11.3 问题和相关的质量方面

大部分我们的问题导致可靠性和可维护性困难还有功能性的问题。上页中的表 11.2 显示实践在哪个方面有助于质量的提高。

结对编程，单元测试，简单设计是最有助于改善可靠性和可维护性相关的问题（++等级）。这三个都有助于解决功能性相关的问题（+等级）但是短的发布周期和现场客户更有效。（++等级）

总而言之，通过引入结对编程，单元测试和简单设计我们收获很大。结对编程和单元测试均被认为有用（中值 5.0）可接受（中值分别为 4.0 和 4.5），

11.4 最终的思考

现在我们应该做的就是引入这些实践。我们对于大部分的实践有一些经验，但是我们应该一次只引入一个。我们必须不再犯如下错误，就是我们只是说“我们现在开始做了各种各样的实践”。相应的我们必须对开发者进行培训。对于引入一项新的实践，每周一到两个小时的培训可能就够了。

Gocept 从这项调查中获得的最真实的价值并不仅仅在于这个文章。我读了很多东西，我们的团队讨论了很多东西。现在我对于问题出在哪里以及如何解决问题有了一些理解。如果没有写这个调查，我不会花费那么多的时间在敏捷软件开发上。

最终我要强调的是敏捷方法不是真正的关于实践的方法。它们几乎不是使他们与严格的方法产生区别的那个原因。使他们不同的是它们在评估和原理方面的基础。它们评估那些构建软件和提倡合作的个体技能。对于我们来说，重要的是巩固、评估我们的开发者的技能和改善与我们的客户的合作关系。

让我以 Tim Lister[7]的话作为结束语：

面对面最好。

只从事一个项目最好。

努力工作，然后回家最好。

晚安。

Domain-Driven DESIGN

Tackling Complexity in the Heart of Software

众多问题根源在于
领域模型没有捕获到
业务的深层内涵

Eric Evans

Foreword by Martin Fowler

《领域驱动设计》中译本

即将由清华大学出版社出版，UMLChina 审稿

OBJECT-ORIENTED SOFTWARE CONSTRUCTION

SECOND EDITION

CD-ROM INCLUDED

- *The* Most Comprehensive, Definitive O-O Reference Ever Published
- An O-O *Tour de Force* by a Pioneer in the Field
- CD-ROM Includes Complete Hypertext Version of Book AND Object-Oriented Development Environment

BERTRAND MEYER

《面向对象软件构造》中译本，UMLChina 辅助教材

即将由 机械工业出版社 出版

WILEY

TIMELY. PRACTICAL. RELIABLE.

UMLChina 指定教材

Agile Database Techniques

Effective Strategies for the Agile Software Developer

《敏捷数据》

UMLChina 李巍 译

机械工业出版社即将出版

Scott Ambler

模型驱动软件开发模式（上）

Markus Völter、Jorn Bettin 著，徐异婕 译

一、工作进展

本论文介绍了一项正在进行中的模式收集工作的成果，这些模式主要集中在模型驱动（model-driven）和基于资产重用的（asset-based）软件开发方面。期待您的反馈！

本文包括几个原型模式（proto-pattern），这些模式的名称后面带有标记（P）。它们未必都能发展为成熟的模式，并且它们的描述仍不彻底。本文之所以收集这些原型模式，意在暗示模式收集工作可能的发展方向。我们鼓励人们提出更多的原型模式。此外，我们正在研究 MDSD 方面的版本化（versioning）和测试（testing）的材料。

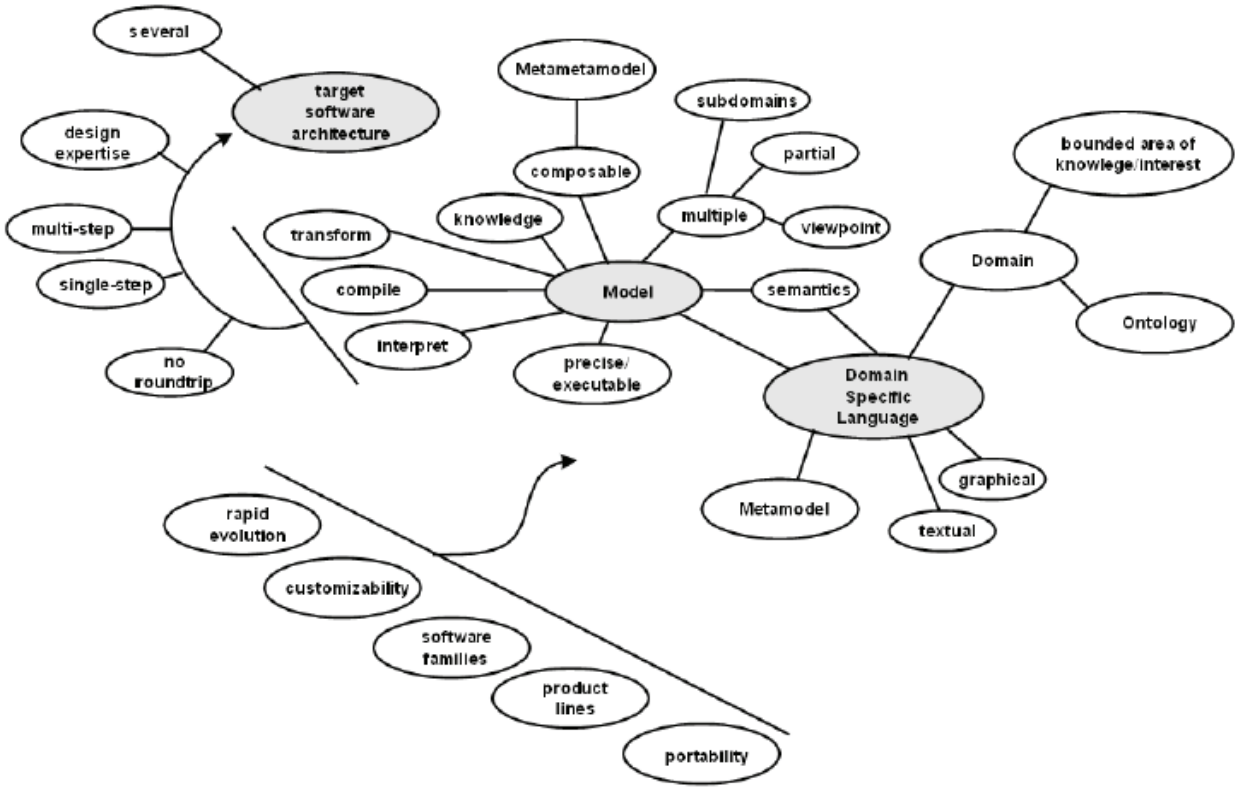
二、什么是 MDSD

模型驱动软件开发（Model-Driven Software Development，MDSD）是一种软件开发方法，旨在通过领域相关模型（domain-specific model）开发软件。领域分析、元建模、模型驱动的自动生成、模板语言、领域驱动的框架设计、以及敏捷软件开发原则，扮演着这种方法的中坚力量，而 OMG 的 MDA 是这种方法的“特有风味”。

下面是一组核心价值，由 OOPSLA 2003 的 BOF 会议定义：

- 确认开发中的软件，胜过确认软件需求。
- 在领域相关资产层面工作，它们包括模型、组件、框架、产生器、语言、技术。
- 努力实现根据领域模型自动构建软件，同时有意识地区分构建软件工厂和构建软件应用。
- 认为软件开发供应链将浮出水面，这意味着和领域相关的专业化、以及大规模定制的出现。

在展开详细介绍之前，请了解下面的思维导图（mind map）提供的宏观概念。



三、模式归档形式

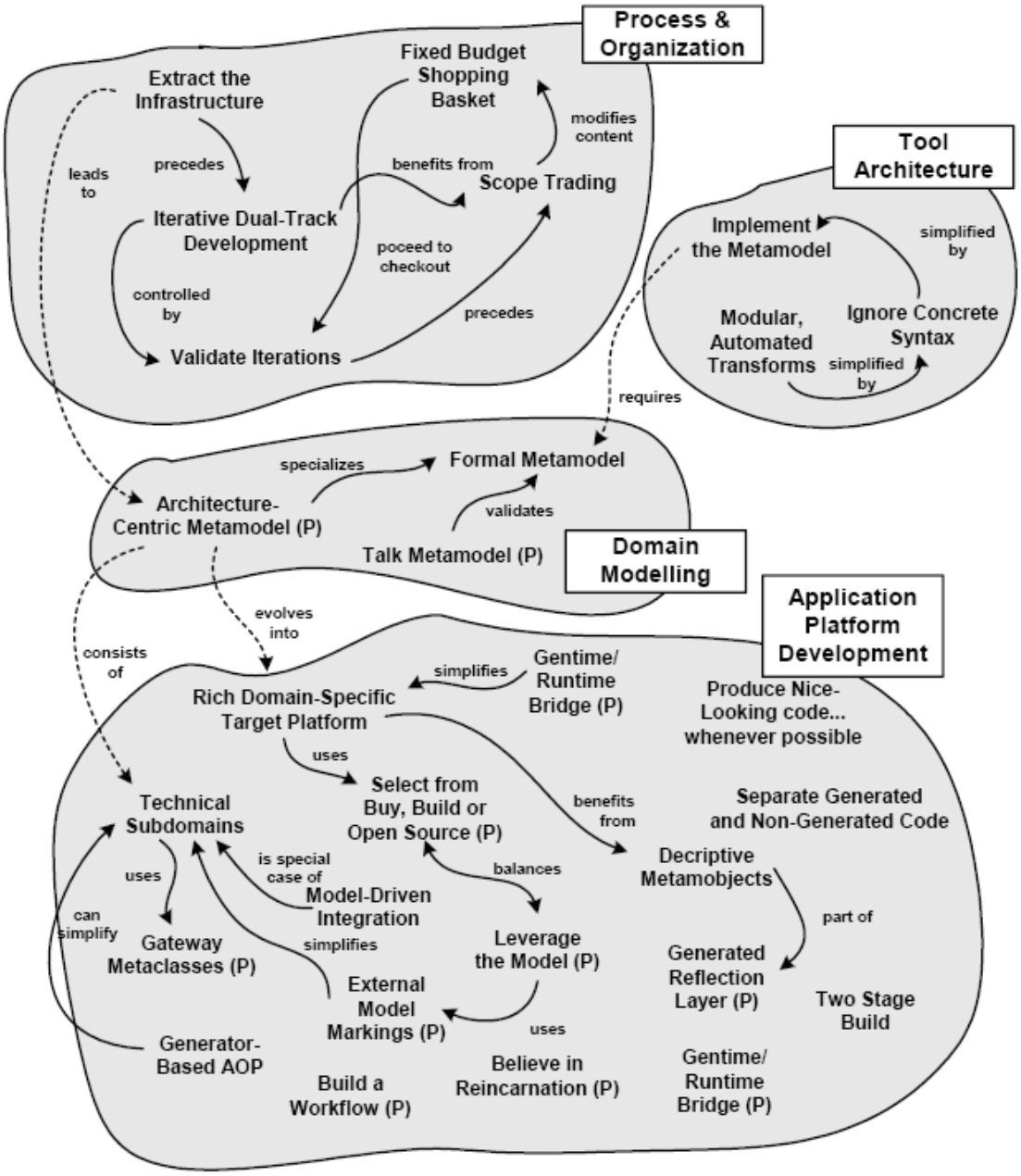
这些模式将采用 Alexandrian 形式来归档。因为这种形式目前广泛使用，并广为人知。读者可参考 Christopher Alexander 有关模式语言的原著[Alexander 1977]。注意，正如 Alexander 的模式语言，我们也为每个模式加了限定符——没有星号、一个星号、或两个星号：

- 没有星号表示我们对模式的内容不是非常的确认。
- 一个星号表示我们认为这个模式是正确的，但我们对细节、公式表达或者所有的强制性约束不能确认。
- 两个星号表示这个模式是实际采用的技术。

模式的已知应用的数量以及质量，与星号的数量是成比例的。

四、综述

模式被分为几组：领域建模、过程和组织、工具架构、以及应用平台开发。下图中，即包含一些模式之间的关系，又包含模式所在组之间的关系。



五、模式详解

过程和组织

MDSD 以明确区分领域工程和应用工程为基础，前者致力于设计和构建应用平台，后者则关注设计和构建单独的应用。软件产品线工程中[CN 2002]，长期伴随着“关注点分离”的思想。

应用开发组和基础设施开发组之间的关系，类似客户和应用开发之间的关系。基础设施开发组需要应用开发组作为现场客户代表，否则，会存在很大风险：基础设施开发组提供的功能不能被应用开发组接受。这并不意味着每个应用开发组都必须有一个永久的基础设施开发组的大使。下面的理解更为合适：每个应用开发组需要一个能够胜任的技术设计者，他参与到一个跨团队的架构组中，这个架构组为基础设施开发组确定需求。跨团队的架构组当然不是什么新概念，但必须保证应用开发组拥有能力足够的的设计师。这样一来，就不必把所有设计高手都集中到基础设施组中了。通过在应用组和基础设施组之间的相互借调（在一个迭代周期期间），有效的技能转换能够在短期内实现。

经验表明，基础设施组常常希望为世界带来新的银弹，他们抱着最好的愿望，违背“众议”而采用新奇技术强行上马，这风险很大。如何控制这个风险呢？方法如下：保证架构组（别忘了其中有来自应用开发组的代表）有权进行“范围协商（SCOPE TRADING）”和“验证迭代（VALIDATE ITERATIONS）”。

在应用开发组中的架构组代表，主要对他们各自的组负责。并且所有基础设施的需求必须源于整个产品/项目的功能需求或非功能需求。

通过借调，可以使领域知识和技术知识更合理地分配。

这种组织结构对于管理比较大的项目非常重要。在小项目中，基础设施开发组可能归结为一两个人，并且架构组的工作开展非常不正式。但无论如何，严格记录“范围协商”的结果还是很有意义的。

尽管 Crystal Orange 并没有特别考虑模型驱动方法，但设置一个基础设施组、以及一到多个应用开发组的组织方式和 Alistair Cockburn 的 Crystal Orange 方法论[Cockburn 1998]是一致的。

迭代的双路开发模式**

你们正采用 MDSD 开发软件系统（或软件系统族）。一到多个组正同时为一到多个应用而工作，你们也需要开发领域相关基础设施（即应用平台）。你们需要按时地、在明确的时间点提交迭代，并且应该尽量减少因进入新的迭代周期而造成的中断。

☆ ☆ ☆

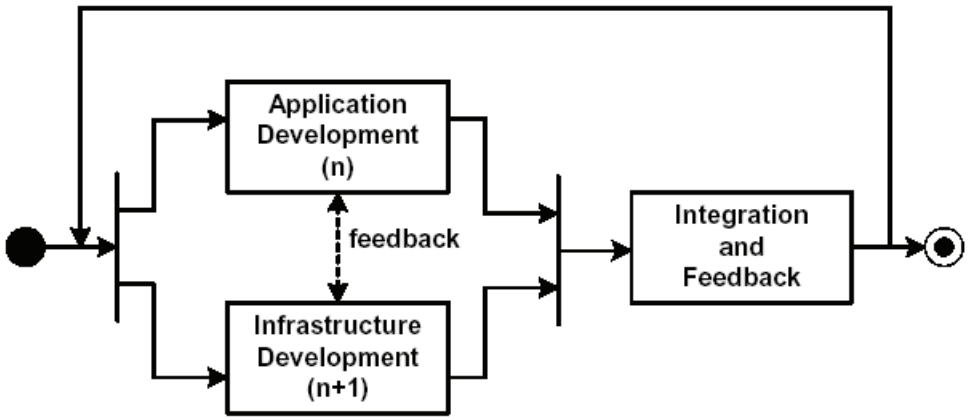
当构建一个新的软件系统族时，实际上你们必须做两件事：一个具体的应用、以及一个有助于基于其上构建应用的 MDSD 基础设施。要对精细的基础设施和应用功能进行并行开发，会对应用开发迭代周期的范围造成影响，因为要根据新的平台不断重构应用代码。

MDSD 基础设施由转换定义、元模型、具体的语法定义以及目标平台组成。

如果基础设施不就绪，就不能构建基于 MDSD 基础设施的应用。同样，如果没有就应用领域的理解达成一致，也不能建立基础设施，而应用领域的理解通常是通过开发多个该领域的应用后才能获得。

因此

基础设施开发至少要和一个应用开发同时进行。应确保基础设施开发者能直接从应用开发者那里获取反馈。无论是基础设施还是应用，都必须进行增量和迭代开发以获得全面的敏捷性。为了解决这个“鸡和蛋”的问题，我们从正在运行的应用程序中“抽取基础设施（EXTRACT THE INFRASTRUCTURE）”。也就是说，每个迭代周期中，首先要进行基础设施开发这一步，并且只在应用开发迭代开始的时候引入基础设施的新版本。



☆ ☆ ☆

在实践中，为了足够敏捷，迭代周期决不应该超过 4 到 6 周，并且最好所有迭代周期都一样长。

注意，基于“同步时间盒（synchronized timebox）”的增量和迭代过程，并不代表在开发之前，你不应该进行[Cleaveland 2001]所描述的领域分析。对领域的深刻理解是开展 MDSD 的有效前提。一旦开发开始，就需要以迭代的方式进行更为深入的领域分析，这是基础设施工作流不可或缺的工作内容。

基础设施组常常希望为世界带来新的银弹，他们抱着最好的愿望，违背“众议”而采用新奇技术强行上马，这风险很大。如何控制这个风险呢？方法如下：保证架构组（别忘了其中有来自应用开发组的代表）有权进行“范围协商（SCOPE TRADING）”和“验证迭代（VALIDATE ITERATIONS）”，于是对于应用开发者来说，正在开发的基础设施成了真正的资产。

在这种方法的后期，要对不同的子过程进行有效的同步，此时版本管理成了问题，特别是如今的 MDSD 工具。另外，升级（重构）应用模型，以适应和使用新版本的基础设施，也并非轻而易举。

注意：

- 一旦 MDSD 在一个组织中建立起来，技术基础设施就需要高度标准化；工作的重点也从标准化地使用技术，转移到构建特定领域应用平台上来。因此，随着时间的推移，术语“应用平台开发”成了更精确的描述。
- 在一个人项目的小案例中，这种模式简化成为如下要求：将基础设施（应用平台）的代码库、从单个应用的代码库中清晰地分离出来。

☆ ☆ ☆

FAST 过程（Family-Oriented Abstraction， Specification， and Translation prcess）[WL 1999]是由 AT&T 开发的，从 1992 年开始被使用，能够明确区分领域工程和应用工程。FAST 传承了二十年的开发软件家族经验，在朗讯科技至少应用于 25 个领域并进一步发展。

本文作者之一从 1994 年开始使用“迭代的双路开发模式”，最初是用于可编程的 LANSA RUOM 模型驱动生成器[LANSA]，后来，用于使用不同 MDA 工具的一些项目中。

另外，b+m GDP（generative development process，产生式开发过程，参见文献[GPD]）也采用了这个原则作为基础，它被运用在模型驱动开发领域很长时间了。它证明了 MDSD 的必要性。

有关产品线（product line）开发组织（团队结构）的更多具体案例，可以参考文献[Bosch 2000]

固定预算购物篮模式**

你正在进行迭代式的软件开发，并要确保固定数额的资金被明智地利用，来构建一个能够满足客户需求的产品。

☆ ☆ ☆

实际经验表明，大型软件开发的初始阶段，常常会给用户和开发组织带来很高的风险。在风险预估期间，双方都试图降低自己的风险：客户坚持固定的价格，而软件开发商试图让固定的价格包含意外情况的风险。你如何来确保用户以固定预算得到一个工作系统呢？

这些过分简单的风险缓解之策是于事无补的，因为大规模软件开发的风险很难——不是不可能——精确预估。为固定价格的合同买保险，也不能保证交付的系统到部署之时一定满足客户需求。同样，预估时充分考虑可能发生的重大意外，也不能保证符合预算、按期交付的系统是功能齐全的。

为了生产用户满意的软件——即完美地支持用户的工作的系统，在软件设计和实现中要考虑得非常周全，但软件的抽象本质使合同细节远远做不到周全。软件需求适用于提供范围（scope）指导，但它不足以保证产品合乎客户心意。

因此：

将固定的预算分散到每一个迭代中，明确可用的资源。使用固定长度的时间盒迭代，至少每三个月要提供一次可以运行的代码。“验证迭代（VALIDATE ITERATIONS）”模式提供了“校验”过程来确认期望的列项，并且将不满意的列项搁置在未决定需求中。随后，商业涉众和选出的最终客户一起，使用“范围协商（SCOPE TRADING）”模式，花费他们的数额固定的“迭代周期预算”装满下一个迭代周期的固定预算购物篮。

☆ ☆ ☆

针对项目初期捕捉到的客户需求进行旷日持久的、反复的评审，未必是好方法；倒是有规律地执行“验证迭代（VALIDATE ITERATIONS）”，通过测试来驱动处于构建过程中的软件产品的开发，来得更为有效。

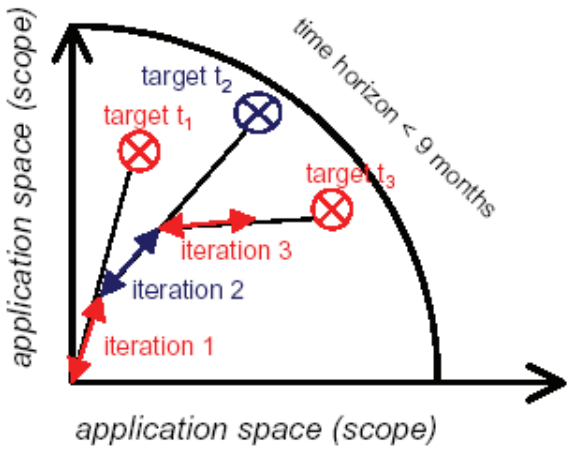
我们对那些没有得到成功确认的需求，加以标注来澄清；并且在旁边罗列出那些在上一个迭代周期中可能已经出现的新需求。随后，开发组针对所有未决定的需求进行实际评估，以支持“范围协商”模式。

两三次的迭代工作可以被打包，并形成中间发布版本。比较短的发布周期使“范围协商”模式更具实效——也就是说，重要的新增需求会代替不重要的需求发生在两个月内，而不是很久以后。发布质量必须达到可交付要求——也就是说，它必须通过所有阶段的测试。因此，一次发布的最后一个迭代周期，将集中在 bug 修复、测试和打包的工作。

下列规则概况了发布和迭代周期的管理方法：

- 采用时间盒迭代不能超过六周，并且由用户/客户进行验证。
- 每三个月至少要提供可交付代码。
- 理想情况下，每三个月将代码部署到产品中，得到实际反馈。

- 新的商业应用的开发，必须在九个月之内“哇哇坠地”，否则就可能危及妈妈（开发组）或婴儿（应用）的生命，如下图所示。



在“应用空间”中，每个点代表一个潜在的系统，该点到原点的距离代表该应用的功能丰富程度。在运行良好的迭代和增量开发中，每个迭代周期都以增量的方式使应用功能更加完善。

☆ ☆ ☆

本文的作者之一从 1994 年开始使用这种模式，随后将此模式精炼为现在的形式，并将此模式应用于多个行业的大量软件开发项目中，涉及技术广泛。该模式在用户和软件开发者之间实现了适当的权利和责任的分配，所有采用此模式的 MDSD 项目以及传统项目都获得了巨大的成功。这个模式甚至被应用在固定价格项目中，例如“Zemindar”项目，很明显，这些项目实际上会变成预算固定、而范围不定的。

范围协商模式**

你需要为下一个迭代周期制定一个（项目）计划，这个计划中需要包含属于下一个迭代周期的全部需求的优先级列表。

☆ ☆ ☆

每个迭代周期的工作内容需要将商业考虑体现在需求优先级上，同时也需要包含具有关键架构意义的列项。每个迭代需要为终端客户和涉众提供一些功能，这些功能从某种程度上能够被验证；同时，架构意义的列项也不能被忽视。在迭代开发中，如果允许增加新的需求，你如何来管理范围（scope）并防止范围的蔓延呢？

迭代方法的首要目的是支持验证和早期反馈，从而能够及时调整项目的方向。依据项目一开始时固定不变的需求列表来衡量进度没有太多意义的。相反，“验证迭代（VALIDATE ITERATIONS）”模式能够每隔一段时间就确定项目进展是否和用户和涉众需求吻合。

没有直接的涉众反馈的“迭代”开发并不罕见，尽管它仍支持尽可能早地排除架构风险，但这没有充分发挥迭代开发的潜能。并且，从用户角度，它依然很象传统的“瀑布”开发，最后结果会出人意料。

因此：

已经进入迭代周期后，不允许新增需求。在每个迭代初期，进行“范围协商（scope trading）”讨论会就此迭代周期的需求范围达成一致，并确定迭代周期的成果。确保出席讨论会的不仅有最终用户，还包含其他相关的涉众，使得优先级达成一致。要将讨论会的结果正式文档化，然后，根据其中定义的优先级来指导时间盒内的开发工作。这样一来，可以避免早期预估时的错误影响最重要的需求和有关键架构意义的列项。

☆ ☆ ☆

范围协商讨论会的持续时间可以根据项目的大小而定，通常需要 1—4 个小时。一旦开发组根据上个迭代周期的工作效率估算了剩余的未定需求的工作量，就可以在“验证迭代”之后立即开始范围协商讨论会。诸如极限编程等的敏捷开发说明了如何务实地度量速度和需求。另外，基于用例点也有类似务实的方法。对于单个需求工作量的估算，我们查阅了[Beck 2000]。参加范围协商讨论会的有项目经理、架构师、关键涉众、以及选出的最终用户。讨论会的目标很简单：

由用户/涉众团体来为每个需求项（用户故事、用例场景、特性——项目所采用方法论的所有相关技术）分配严格的优先顺序。

识别下一个迭代周期中所有极具架构意义的需求，这些需要将被包含在即将开始的迭代周期中。

作为指导，整个迭代周期不应全都用来完成架构意义的列项，除非这些列项都位于用户优先级列表最前面。在项目之初，“抽取基础设施（EXTRACT THE INFRASTRUCTURE）”意味着有大量的架构工作需要完成，这项工作将和经常在项目初期开展的构建原型工作一起配合进行。经过几个迭代周期之后，架构意义列项的数量将减少。这样一来，就做到了由涉众/用户团体来驱动开发的方向。一旦开发组织有个针对特定领域的、优秀的 MDSD 基础设施时，就可以立即开始完成用户需求的新项目了。

所有没有分配到此次迭代周期时间盒内的列项，将被移动到未解决需求列表中，作为后继迭代周期的候选列项。此时此刻（指范围协商讨论会），是最终用户和涉众干预即将到来的迭代周期工作范围的最后机会。当然，如果上述的优先级排序工作做得不错，改变应该是比较小的。

如果最终用户不能同意采用严格顺序，涉众可以决定采用随意顺序。这样，当范围变化或资源有限时，对下一个时间盒可以削减哪些列项有个清楚的预期。

当要开发的产品将针对多个需求不同的客户时，“范围协商”模式很有用。此时，产品开发的预算就那么多，每个客户的需求将得到不同程度的满足。既然这样，可以采用这样的方法来建立一个公开、公平的投票流程：按照每个客户投资数额的比例，分配给他们一组“优先点”，并且允许每个客户将“优先点”分配给产品线上的列项。对优先级的评判，可以采用不记名的方式，由产品供应方甚至可以是独立的第三方组织来推动。推动者的职责很明确，他们推动用户以迭代的方式修订优先点分配，直到没有客户要求更进一步的调整为止。通过使用类似于电子竞标的协同工具，可以保证计划在规定时间内完成。

☆ ☆ ☆

本文的作者之一连续使用此模式已经很多年了，从为期三个月的小型 WEB 应用项目开发，到价值几百万元的企业级应用的大规模软件开发都有。这个模式能够帮助涉众在每个迭代周期结束后引入新需求和优先级，有助于他们有效控制项目。有时涉众会惊讶地发现，优先列表后面的列项竟和前面的一样重要。在“Zemindar”等项目中，这个模式帮助项目在固定期限内交付了可工作的系统。

验证迭代**

你希望经常得到确认：项目是处于跟踪状态的，并希望明确哪些列项满足了客户期望，哪些列项需要进一步明确或重做。

☆ ☆ ☆

最终用户需要有能力通过测试来驱动软件开发，并且和涉众一样，需要有能力提供给开发组有效的反馈。尽管现场客户【注：现场客户是极限编程实践的核心成员，其中的敏捷原则是：贯穿项目始终，业务人员和开发者必须每天都在一起工作】能够确保需求可以每日澄清，尽管挑选的最终客户会不断地通过测试来驱动软件开发；然而，在时间盒内部的范围仍需要保持固定，过程仍需要有规律地定时跟踪，涉众仍需要确认最终用户的建议方向是否准确。

在项目中的任何时间，最终用户和业务专家都可以确定和清晰的说明新的需求，但时间盒内的工作范围需要保持稳定。尽管需求变更时，开发组有必要重新进行工作量估算，但开发组不应该在迭代周期内受到影响。另外，优先级需要很正规地重新安排，又不会对开发过程中的其他部分产生影响。

总之，时间盒结束后，包括对工作量重新估计的“迭代周期结束工作”，时间应该尽量短，以便下面的开发工作时间盒尽早开始。

因此：

正式的验证迭代讨论会通过如下方法决定时间盒：确认进展程度、并且归档用户和涉众可以接受的部分软件内容。让最终用户担当现场客户，并对已经完成的功能进行演示。向最终用户和涉众团体明确传达：在专题讨论会期间，可以随时提出新需求。鼓励涉众对假定场景的探索，并可以在观看演示时，提出新的想法。同样地，架构师可能想借此机会提出一些需求启发未涉及、以及开发中暴露出的问题。

☆ ☆ ☆

“验证迭代”和现场客户是相互补充的关系，两者缺一不可，否则有迭代开发失败的重大风险。现场客户是增量和迭代地进行用户验收测试的基础，而“验证迭代”对用户验收测试结果进行正式确认。

验证讨论会的结果体现在迭代评估文档中，包括：

- 客户认可的需求列表;
- 圆满完成的需求列表;
- 需要返工的需求列表。

“验证讨论会”之后，“范围协商讨论会”之前，需要修订高优先级的需求规约，因此，从而支持需求规模评估。

使用“范围协商”模式的迭代周期结束活动，时间范围从不足一天到最多五天；时间越短越好，不过，该结束活动的级别不能降低。其工作量的大小依赖于下列因素：涉及需求的数量、需要确认的功能点的复杂程度、以及涉众和最终用户达成一致的快慢。

最好由最终用户来演示这个系统，向其他的用户来说明新的功能。作为选择，也可以由业务专家来演示。

光有演示者还不够，还必须有一位熟悉正处在开发过程中的软件的推动者出席，只有这样才能就下列问题达成共识：所实现的特性是否满足需求、要求的调整是否被客户所接受。另外，推动者还扮演着记录员的角色，记录所有特性以及在演示时最终用户认可的特性。特性（feature）被公认为是便于进行小的修改的主题，这些小的修改需要在讨论会上归档。推动者必须归档的东西还包括未被最终用户或业务专家认可的特性的缺点。

在跨越时区和地域的分布式项目中，确认讨论会需要被管理起来，可以采用电子手段——比如基于网络工具的视频会议。有的场合涉众(用户)较多，确认讨论会可能需要由代表来完成，他们来自产品用户组，或是选出的客户代表。另外，可能需要不同的听众参见多个讨论会，这时需小心，拖延迭代周期结束时间可能会耽误下一步的产品开发。

☆ ☆ ☆

本文的作者之一多年以来在多个项目中一直应用此模式。这个模式被成功推荐给了几个组织，然而在最初，这些组织都怀疑时间盒迭代软件开发的价值。

实践项目经验表明，对于用户和涉众，最初的确认讨论会总会有点虎头蛇尾，总结起来有如下两个原因：其一，确认讨论会发现原有存在很多误解，这暴露了软件开发组和软件开发过程的能力不足的问题。其二，在第一次迭代结束时，不会有很多明显的功能是可用的，更多的早期工作被用于排除架构风险、以及构建底层的基础设施。通常是在第二个迭代周期的确认讨论会上，赢得客户的信任，因此此时客户能看到上一次讨论会中记录内容的反馈。

抽取基础设施**

你正在使用模型驱动的开发技术开发一个软件系统。但你还没有一个可用于各个领域的 MDSD 基础设施。

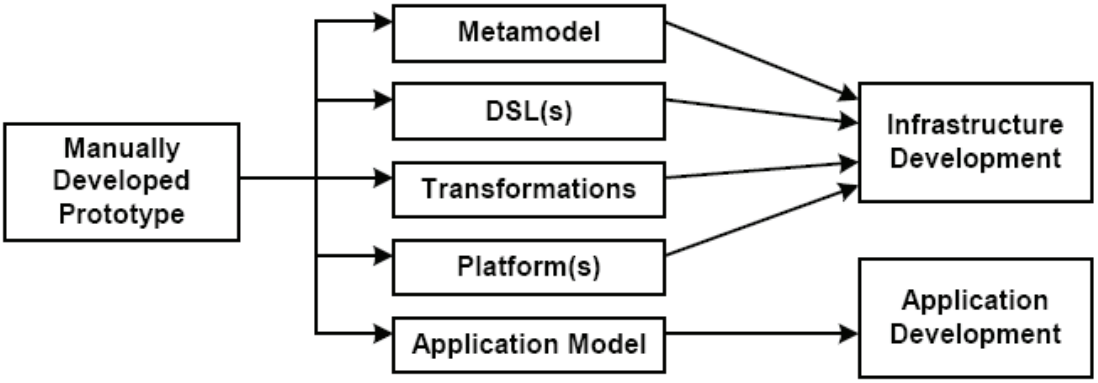
☆ ☆ ☆

在 MDSD 项目的最初，经常不知道该如何开始。你知道应该使用“迭代的双路开发（ITERATIVE DUAL-TRACK DEVELOPMENT）”模式，但却不知道该怎样开始，你希望能尽快得到一个最起码可以运行的应用。

构建 MDSD 基础设施要求按照模型转换和元模型来考虑。必须利用大量的转换说明/规则（代码生成模板等）将实现代码分散。很多人并没有用过这种方法。同时也应该保证：在项目的早期阶段，始终不必调试自动生成的代码。自动生成代码应该有一个明确的最低质量要求，并且确保在实际工作中满足。

因此：

从一个正常工作的应用案例中来抽取转换。按照惯例，从开发一个原型开始，然后构建基于这个正常工作的应用的基础设施。在这个初始阶段之后，再开始“迭代的双路开发”。



☆ ☆ ☆

原型应用应该是每个领域的代表性应用，既不能太简单，也不能太复杂。如果正在构建一个大而复杂的系统，最好只将这个系统的一个子系统作为原型。

该模式有两个特点：

- 如果你开发各自的领域应用已经有一段时间了，而你又希望能引入模型驱动的方法，你可以从以往的开发应用中“抽取基础设施”。
- 如果你开始开发一个新的、复杂的软件系统，你应该真正的开发一个原型来理解这个系统，并且从这个原型中“抽取基础设施”。

必须注意：你应该从具有优质架构的应用中抽取基础设施，因为它将是软件系统族的基础。因此，即使你有一组积累的应用，仍然建议你用新的、改良的、经过整理的架构来写一个新的原型。

根据作者和其他业界人员的经验，抽取基础设施（或者叫“模板化”）大概需要占用开发原型的 20－25%的时间。

这个方法不仅支持抽取转换，还可以帮你抽取合理的元模型，作为 DSL 的基础。和在“迭代的双路开发”模式中描述的一样，抽取一个有表现力的、精巧的 DSL 也需要进行迭代。

同样要注意：一些自动生成器（特指那些使用文本的模板）能很好地支持从正常工作的程序中抽取模板代码。

最后再提醒一点：上文中我们提及的“模板”与 C++模板无关。代码生成器模板是一种特殊的生成器，它还支持运用元模型。它提供了常用控制逻辑的构造、嵌套等。一般而言，它们非常小巧并易于使用。

☆ ☆ ☆

在一个为嵌入式系统开发模型驱动基础设施的项目中，这个系统家族的信息核心将全部由模型自动生成。项目将首先用人工方式为某个特定场景开发完整的核心实现；然后，从这个原型中抽取自动生成的架构。虽然公司有这方面的经验，但依然如此。

该模式的动机源于 LANSA 模板语言，后来在 1993 年它被作为 LANSA RUOM（一个基于模型驱动生成器的客户定制模板语言）的基础。本书的作者之一从 1994 年以来，一直使用原型代码“模板化”来自动完成基于模式的开发，构建了保险系统、配电系统、ERP 系统、以及电子商务系统。尽管现今的 MDSD 工具仍然采用未标准化的模板语言，但基本过程是相同的。作者通过 LANSA RUOM [LANSA]、Codagen Architect [Codagen]、eGen [Gentastic]、GMT Fuut-je [Eclipse GMT]、b+m openGeneratorFramework [GenFW]等确认了这种软件开发模式的正确性。

（待续）

征稿

http://www.umlchina.com/xprogrammer/xprogrammer.htm

The Unified Modeling Language Reference Manual, Second Edition

JAMES RUMBAUGH
IVAR JACOBSON
GRADY BOOCH

Covers UML 2.0

UMLChina 译
（王海鹏、李嘉兴）

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

CD-ROM included

《UML 参考手册》2.0 版中译本
即将由机械工业出版社华章分社出版

采纳 RUP－－缺陷和解决方案

Cogenture 著，李静 译

讨论

统一软件开发过程－－含义

人们可能会假设采纳一个新的软件开发过程只会影响软件项目，现实中，组织的每个层级都有可能受到影响。

一个新的过程在开发和部署过程中典型地伴随着新的工具，技术和方法论。为了确保成功，不可能再依赖现存的能力；软件开发组织面临很多的，并发的学习曲线，显著地增加了项目的风险。

统一软件开发过程和它的相关工具和技术能够为成功的采纳者带来巨大的优点，但是现实是在你认为最不可能出现问题的地方，你将会遇到最主要的问题。

统一软件开发过程必须伴随着开发团队中组织和文化的主要改变，但是更重要的是，如果你想有所收获，你必须

- 发展一个企业级的战略
- 在个体的项目之外安排你的开发
- 改变组织以及它对于项目可交付使用的预期
- 改变评判和资助项目的方式
- 改变开发过程中纳入终端用户的方式

一个主要的改变

瀑布模型到迭代模型

采纳统一软件开发过程意味着在一个与传统的瀑布模型有着显著不同的风险驱动的和迭代开发的基础上重建和管理项目。这样的改变要求有一个组织，这个组织支持管理需求，资源分配，控制过程，作业分配和获取度量的全新方式。

用户为中心的方法

与传统的方法相比，用例驱动需求的管理要求更大范围的和持续的将用户纳入开发过程。类似地，用户验收和测试在项目早期开始而且贯彻项目始终。这影响着用户的组织，除非周密地安排，否则会严重影响项目的交付。

设计

采纳面向对象的分析和设计，如果这是全新的，则代表对于开发团队的一个主要的范式转变。学习曲线可能是极大的。

配置管理和变更管理

同时配置管理和变更管理的原则不会是新的，统一变更管理（UCM）的整合方法以及它在统一软件开发过程中的执行带来了实质性的收益，但是没有对方法，用例和组织进行改变。

开发组织

统一软件开发过程基于一个良好定义的组织模型；只有统一软件开发过程中定义的角色在项目中存在，它才能够有效的展开。同时大部分新的使用统一软件开发过程的项目会对于统一软件开发的角色绘制人员地图，这个改变会引起相当大的张力，除非已经进行了恰当的准备。组织的改变是最难实现的。

统一软件开发过程的最好实践

采纳统一软件开发过程作为一个软件开发过程意味着采纳它的六个最佳的实践：

- 使用基于组件的构架
- 迭代开发
- 模型可视化
- 管理需求
- 管理变更
- 不断检查质量

这些不是随意的，它们对于统一软件开发过程的效果起到基础性的作用。所有的实践都暗示着在组织范围内对于组织的改变，特别是前三个方面。

基于组件的构架

基于组件的方法允许每一个开发项目增加一个可以复用的软件资产的增长式的集合。经过适当的安排，在企业架构的背景下，这点对于软件开发和维护的时间和成本能够产生非常显著的影响；它也将导致开发组织的重大改变，因为为了和组件构架匹配它会随着时间重建。

迭代开发

从运作的立场来看，迭代的方法要求一个全新的资源管理方法。对于项目发起人的影响不太明显，项目发起人必须学习修改他们的预期从而和方法匹配。基本的改变是使用迭代的方法对于项目成本，资源需求和时间表的评估占据了更长的时间，但是最终这些评估也更准确。如果组织想要对改变做出有效的反应，所有的参与人都必须理解更长的评估，更好的准确性和风险降低之间的平衡关系。

模型可视化

统一软件开发过程是模型驱动的。在统一软件开发过程中，模型

- 是一些说明，软件根据它们进行开发
- 是语义，项目成员通过它进行交流
- 定义了企业的构架，而不仅仅只是一个系统

为了使模型真实有效，它必须在一个业务模型的背景下执行。这决定了对于基于组件开发的长期战略，这也是成功执行的核心所在，在企业级上的商业模型在任何一个个体的开发项目之外，必须被分离出来说明。

使用统一软件开发过程和 Rational 工具——引起伤害的地方

过程剪裁

使用统一软件开发过程，“即开即用”是不可能的，有太多的选择。全面的看，统一软件开发过程是 20%的规则和 80%的选择。在成熟的组织中，这可能会被接受，但是对于新的采纳者而言，这些比例反过来是要点。没有剪裁，对于相同的问题做出的不同决定会导致矛盾的产生，对于个体项目是这样，对于不同的项目也是如此。

在一些范围内的局部实践如项目管理，测试，配置管理和变更管理也必须包含在剪裁中。

工具配置和部署

Rational 工具“即开即用”的使用提出了类似的困难。没有增加和改变，标准的计划缺少一个实际项目所要使用的足够的细节，而这样的改变又必须始终在项目中使用以支持项目的内部工作。工具计划也需要裁剪以反映组织的裁剪过程和局部实践，如测试和配置管理。

设立 Rational 工具需要一些数据库，知识库连同用户帐户和访问权限的服务器部署，同时也需要客户端工具的安装和与恰当的服务器建立联系。客户端工具通常需要桌面升级，客户端服务器的构架对于网络的构架提出性能要求。

项目启动

在恰当的时间使用新的过程和工具箱启动项目是困难的。项目团队首次使用统一软件开发过程时所面临的最主要挑战是做计划——为迭代开发、与开发相关联的新的资源配置文件、开发过程和开发工具中恰当培训。及时获得桌面工具，在过程中对于它们的使用进行指导也是一个重要的挑战。

监测和控制

当采纳一个不熟悉的过程时，特别是当它是迭代的过程时，测量项目的进展会出现很多问题。如果组织对于迭代开发的使用不成熟，对于项目进展的测量常常用瀑布法。很多组织把阶段转变作为主要的项目里程碑，这给项目带来了问题。实现这种以时间为基础划分的里程碑的压力与实现以项目所处阶段的质量评估为基础划分的里程碑产生了直接的冲突。

Upskill———一个解决方案

UpSkill 是一个有保证的，固定价格的，关注交付的，技能转移项目。它以固定的价格，提供可以测量的技能提升帮助组织在采纳统一软件开发过程和采纳 Rational 工具过程中控制成本。固定价格允许使用软件行业标准经济成本模型来论证投资的回报。

UpSkill 有助于减少采纳一个新的软件开发过程和工具的风险。通过提供现行项目所需要的支持，提供所需要的技能转移，使用规则的度量方法跟踪和监控过程，提供包含过程，工具和环境的高覆盖性，UpSkill 对现行的使用新的过程和工具的项目交付提供帮助。UpSkill 创建了一个安全的学习环境，这个环境关注项目的交付。

大范围的工作模型

UpSkill 涉及到组织的每一个层级：

- 企业级
- 程序层
- 项目级
- 项目团队成员级

在企业级，UpSkill 关注战略，帮助定义一个与企业结构相匹配的软件资产开发长期框架。

在程序级，效益管理制度确保程序识别和实现采纳统一软件开发过程的目标收益。这些包括对于软件资产的贡献和改进软件开发效率带来的可论证的投资回报。在 UpSkill 的使用过程中，内部指导者和管理能力的发展，对于 RUP 和 Rational Suite 是一个内部的推进能力的发展，正是如此，UpSkill 有助于组织的自我实现和消除对第三方的依赖。

UpSkill 帮助创建环境和必需品以支持采纳 RUP 和 Rational 工具的项目。这点通过创建对于 RUP 和 Rational 的管理认知，通过引入资源计划和与迭代开发兼容的项目过程测量来完成。

在程序级，UpSkill 有助于完成 RUP 推进的持续性，避免经常出现在采纳新过程的个体项目中的技能分散。

在项目级，UpSkill 关注整个项目，以确保整个项目成员获得他们所需要的支持。通过程序级的工作，UpSkill 能够确保成功的项目启动，从项目启动开始工作，提供项目初期所需的关键性的支持。这使得培训和指导被预先安排到了项目计划中。在项目级，UpSkill 关注交付，提供所需的支持以确保项目成功。这包括由专家选取核心角色以降低失败的风险。

在团队成员层，UpSkill 关注技能转移和支持。UpSkill 使用一个围绕学习流的结构化的技能提升模型。技能转移通过技能评估进行测量，当项目成员获得了他们从事工作所需的技能的时候会提供证书。技能转移被安排到个人的学习计划中，交付被安排到了培训，积极指导和持续评估中。

学习流

项目成员被分派到一个学习流中，这个学习流符合 RUP 的原则。这比选择 RUP 角色更重要，它激励团队通过小组学习整合。下表列出了学习流、项目成员在流中执行的关键活动、执行活动所需的技能以及使用的工具。

学习流	关键活动	技能	工具
项目管理	启动项目 识别评估风险 安排阶段和反复 开发迭代计划 启动迭代 评估迭代	迭代的项目管理	Microsoft project ClearCase ClearQuest Requisite Pro
需求管理	获得需求 找出执行者和用例 细化用例 找出业务执行者和业务用例	需求收集 用例建模 UML	ClearCase ClearQuest Requisite Pro Rational Rose
架构	用例优先 架构分析 整合现存设计元素 识别设计机制	架构分析 面向对象分析 面向对象设计 UML C++/Java/VB.NET .NET Framework/J2EE	Requisite Pro Rational Rose ClearCase ClearQuest
分析	用例分析	面向对象分析 UML	Requisite Pro Rational Rose
设计	用例设计 类设计 子系统设计	面向对象设计 UML C++/Java/VB.NET .NET Framework/J2EE	Rational Rose ClearCase ClearQuest
开发	实施构件 进行单元测试	程序语言 单元测试 C++/Java/VB.NET .NET Framework/J2EE	IDE ClearCase ClearQuest
整合	整合子系统 整合系统 提倡基线	配置管理 持续整合 C++/Java/VB.NET .NET Framework/J2EE	ClearCase ClearQuest

测试管理	识别测试动因 获得易测性承诺 评估和提倡质量	测试战略评估 质量保证	Rational TestManager
测试	定义测试细节 定义评估和可追溯性需求 确定测试结果 实现测试套件 实现测试 执行测试套件	测试用例设计 测试用例创作 测试实现 测试执行	Rational Robot
配置和变更管理	编写配置管理计划 执行配置审计 创建部署单元	配置管理 ClearCase 管理 ClearQuest 管理	ClearCase ClearQuest

表 1 学习流

初始技能评估

进行培训和指导之前，每一个项目成员都要进行一个与他在项目中角色相关的初始技能评估。他们当前的技能水平作为一个参考基点，用以测量技能提升，需要提升的技能面，这些记录在个人学习计划中。每一个项目成员被认为是独一无二的，会接受培训，指导以及适合他们的个人情况的支持。

初始的评估给每一个人按如下等级评级：

- 新手
- 提高者
- 胜任者
- 专家

很典型的，需要提供给每一组的服务如下：

等级	提供的服务
新手	达到胜任者水平的培训和指导
提高者	达到胜任这水平的培训和指导
胜任者	达到专家水平的培训和指导
专家	达到指导者水平的培训和指导

表 2 初始评估等级

个人学习计划

初始评估之后，为每一个项目成员准备一个个人学习计划。通过个人学习计划，个人的培训和指导需要被记录，他们的进步也被追踪。它是一个动态的文档，由项目成员和他们的指导者共同拥有，不断更新，直到成员获取他们所需要水平的证书。它会记录：

- 过去的经验和培训
- 所要达到的技能目标以及当前的技能水平
- 行动计划
- 成果记录

随着学习的进行，个人学习计划被技能水平的新的评估等级更新。帮助技能发展的必要行动，特别是项目行动，被识别和同意，成果被记录。当目标技能水平达到时，重新评估开始就绪，预示可以获得证书。

培训

教室里的培训提供给项目成员必要的基本知识，以帮助他们个人学习计划中识别出来的技能发展需求。课程表为所有的项目成员确定了共同的培训课程，专家课程提供给每一个学习小组。

当项目活动需要的时候以及个人学习计划预定好的时候，培训及时进行。课程由 Rational 课程剪裁组成，以反映组织的过程和工具配置。

学习流	课程
所有	Rational Unified Process Fundamentals(2) Fundamentals of Rational ClearCase and Unified Change Management (2)
项目管理	Principles of Managing Iterative Development (3) Managing Software Projects with Rational ClearCase and Unified Change Management (1)
需求管理	Requirements Management with Use Cases (3) Rational RequisitePro Fundamentals (1) Fundamentals of Rational Rose (1) Introduction to Business Modeling using the UML (1)
架构	Fundamentals of Visual Modeling (1) Principles of Architecting Software Systems (3) Fundamentals of Rational Rose (1)
分析	Fundamentals of Visual Modeling (1) Object-Oriented Analysis with the UML (3) Fundamentals of Rational Rose (1)
设计	Fundamentals of Visual Modeling (1) Object-Oriented Analysis and Design with the UML (4) Fundamentals of Rational Rose (1)
开发	C++/Java/VB.NET (5) NET Framework/J2EE (5)
整合	C++/Java/VB.NET (5) NET Framework/J2EE (5) Managing Software Projects with Rational ClearCase and Unified Change Management (1)
测试管理	Test Manager
测试	Principles of Software Testing for Testers (2) Essentials of Functional Testing with Rational TeamTest
配置管理和变更管理	Managing Software Projects with Rational ClearCase Unified Change Management (1) Rational ClearCase Administration (2)

表 3 培训课程

指导

指导由项目工件和活动驱动。指导在初期开始，给项目经理提供帮助，以准备项目计划，项目计划为把培训和指导纳入计划范围提供机会。这有助于培训在需要的时候及时交付，指导与项目活动结合，因此避免了培训和指导被视为是随意的。

指导方法是积极的，围绕指导中心，办公桌指导和复审—项目成员被指导如何执行活动，在完成活动中给与他们帮助，然后工作被检查以观察工作是否全部完成。

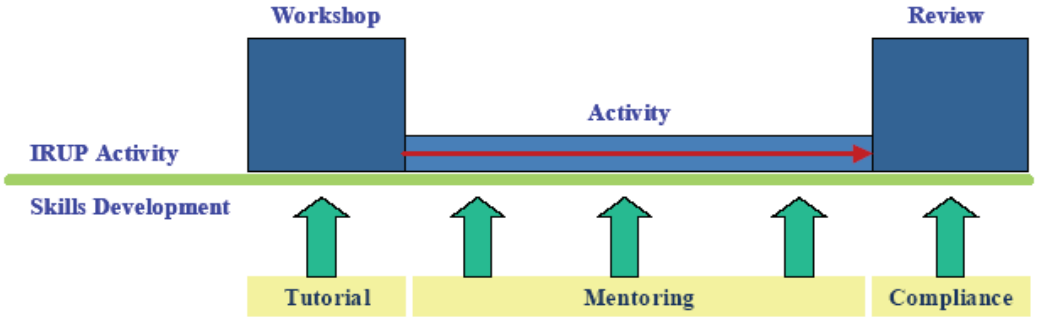


图 1 积极的指导模型

当项目成员开始进入活动，工件进入工作流时指导中心被保留。出席者包括工件的制造者，贡献者和用户。工作中心包括产出工件的走查，针对现实世界的项目状况，指出最好的实践和理论应用。工作中心产出是真实的工件样本，加上对于活动的提示和小技巧。

对于工作，桌边指导在工件生产的全过程发生。典型的是一对一的或者在小组内进行，包括工具指导在内。个人学习计划允许关注项目成员所需的特殊的指导领域。

当工件在基线时，特别是在迭代的末期，来自质量保证过程的复审开始。在学习的环境中，正式复审有助于确保坚持 RUP 和加强指导。

改善知识的保持力

积极的指导方法通过加强培训改善了知识的保持力—有不少证据显示回到工作室后，教室培训会快速被忘记。积极的指导：

- 把培训转化为成果
- 鼓励个人成长
- 搭建能力隔阂的桥梁
- 提升表现不好的人
- 帮助克服学习曲线
- 减少现有项目采纳新过程的风险

下图说明了积极的指导是如何加强正式培训改善知识保持力的：

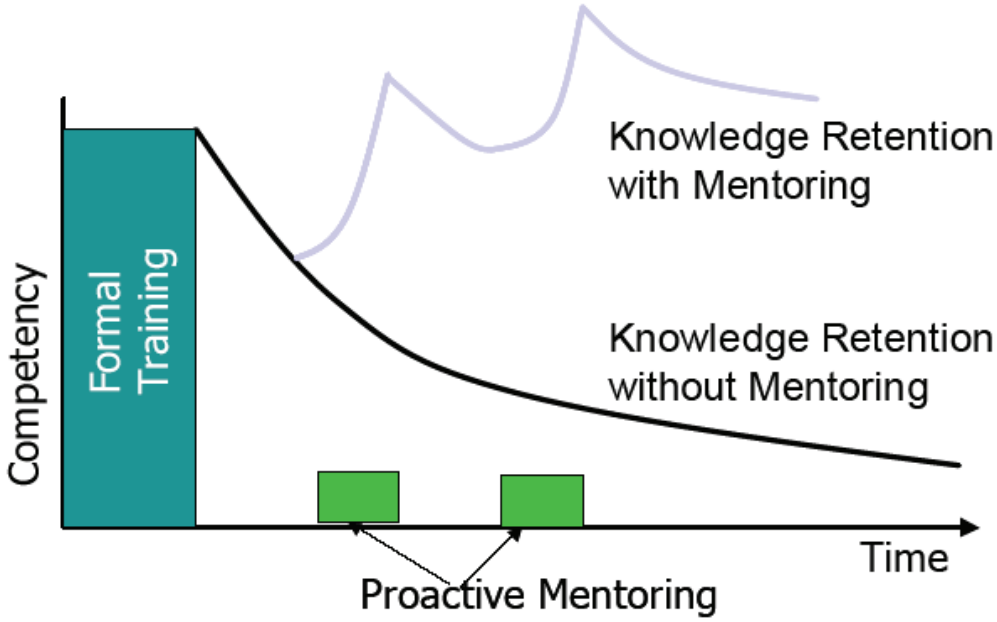


图 2 改善知识保持力

软件与系统思想家温伯格精粹译丛

现代需求技术的基石

探索需求

设计前的质量

Donald C. Gause
Gerald M. Weinberg / 著
章柏幸 王媛媛 谢攀 / 译

Exploring Requirements: Quality Before Design

UMLChina 训练辅助教材

清华大学出版社

Agile软件开发丛书

PEARSON Addison Wesley

有效用例模式

Patterns for Effective Use Cases

Foreword by Craig Larman

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

UMLChina 指定教材

清华大学出版社

需求工程师的素质

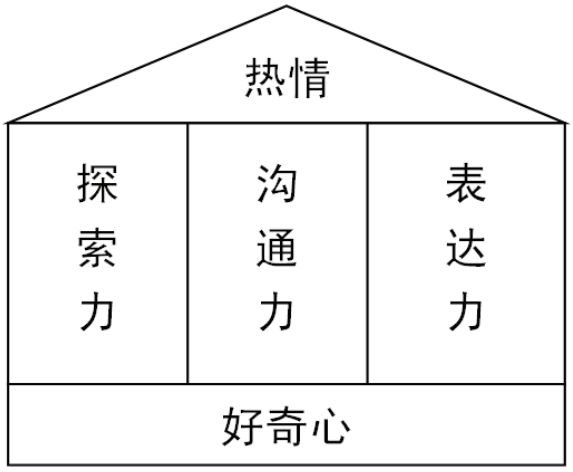
潘加宇 文

讨论

如何改进需求？笔者认为，首先必须意识到需求是一种技能，否则不可能改进。如果团队只把“技术”定义为“实现软件”，认为需求只要愿意去做就能做好，那么“改进需求”永远是一句空话。事实是，很多团队就算愿意去改进需求，也不知道怎样才能做好。

按照需求的各项活动，技能可以分为启发技能、定义技能，管理技能等；还可以分为团队技能，个人技能。本文不讨论访谈技术、用例技术、或者需求变更，这些在以前的文章中已经涉及。本文讨论这些技能的最终执行者——需求工程师，他们为了掌握好这些技能，最好具备什么样的素质。

笔者把一名优秀需求工程师所需要的素质归纳成一所房屋的样子：

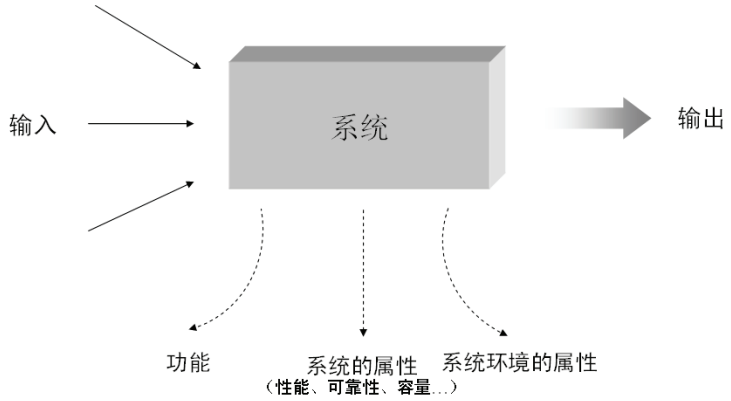


房屋的根基是好奇心，有三根柱子：探索力、沟通力、表达力，以热情作为屋顶。

好奇心

好奇心，首先指对不熟悉的事物提起兴趣的能力。在做项目时，有的开发人员只对项目将要用到的新技术感到兴奋，对项目所涉及的业务领域则不感兴趣。为什么调研过程总是流于形式？为什么更喜欢在办公室“编写”用例，而不是深入第一线？为什么喜欢甲方的信息中心人员，而不是不懂电脑却至关重要的涉众？这就是原因之一。

好奇心，更重要的是从熟悉中发现惊奇的能力。很多时候对业务太熟悉或者存在已有系统，也是捕获需求的一种阻碍。这种情况下很容易就想到系统里有哪几张表，怎么调用，反而钳制住了思维。必须要学会抵制各种想要向里探头的诱惑，尽量跳出来看，从熟悉中发现惊奇。这样才能从涉众提供的资源中，超越涉众的目光，开发出在局中人无法察觉的需求。需求本来就是要把系统看成是黑盒子，不关心它是如何被构造的，只描述它被使用时表现出来的功能和性能：



如何培养好奇心？其实日常生活中充满了惊奇：

一大群人，半夜不睡觉，讨论过去一天的各种事情，然后把它们整理成厚厚的一叠纸，送到你的办公室，只收 1 元钱。

有一种动物很奇怪，每天从壳子钻进钻出，一到晚上，就几个一组钻进一个壳子里盯着一个发出荧光的盒子一动不动。

用另一种心态来观察这些事物，可以培养好奇心。看一些“短路”的漫画，或者做一些不熟悉的事情，例如你喜欢看《体坛周报》，不妨去看看《人民日报》；如果你喜欢看《程序员》，不妨去看看《知音》。

探索力

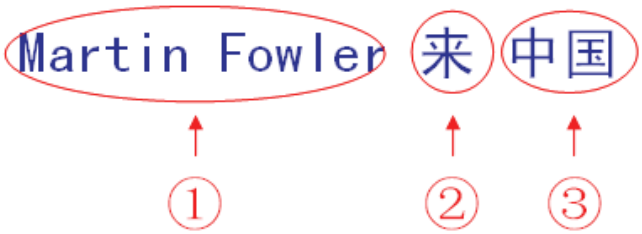
探索力，首先是寻找线索的能力。业务现实就是这样，资料在那里，涉众也在那里。你怎样去寻找线索？拍拍涉众的肩膀，“来吧，说说你对这个系统有什么需求？”这样行吗？

探索力，还包括从线索中归纳出问题的能力。就象侦探一样，需求工程师需要从涉众提供的各种信息碎片中捏出真正的问题。这种探索力更强调的是“合成”，而程序员擅长的是“分解”——针对问题，采用某种软件技术解题。（这里的程序员指广义的软件实现人员。不管他是用图形（如 UML）还是字符（如 C#）来“编程”。）这就

解释了：为什么程序员在“寻找用例”时往往会变成“功能分解”？为什么用例的道理很简单，我们却一次次地摔倒？

某开发团队在一个制作报表的需求上纠缠了很长时间，最后才发现客户方的最高负责人从未要求，也不在意有没有正式的“报表”，他在意的是随时掌控进展情况并做出平衡。如果他确实需要一份“报表”，系统可以给他提供基本数据，他宁愿用自己的人制作任何他想要的报表。可惜，没有人早一点问：为什么需要这个报表？

日常生活中如何培养探索力？例如最近的事件，“Martin Fowler 要来中国”，如果一开始只是知道这么一个事件，并不了解其中细节，可以尝试针对各个环节的信息，通过“反转”、“取代”等手法来探索：



①为什么是 Martin Fowler 来，别人象 Kent Beck、Scott Ambler 也会来吗？

①Martin Fowler 以前来过吗？这次来是为了什么事？旅游？讲学？

①Martin Fowler 来了之后怎么办？会在中国开展业务吗？

②为什么是来，有没有中国公司“去”过 Martin Fowler 那里上门求教？

②是坐飞机“来”吗，能不能从网上“来”？

③为什么是中国，他去过印度吗？日本呢？

探索力的培养方面，Edward de Bono 的书如《六顶思考帽》、《水平思维》等书虽然与软件不直接相关，但很有参考价值。和螺丝刀、拉链一样，软件实际上就是为人服务的一种工具。

沟通力

沟通力包括需求工程师和涉众沟通的能力。**涉众往往表达的并不是需求，而是一种“需要”或者“利益”。**认为客户和程序员一说，就能理解执行得到正确的软件，这种想法是天真的。例如，一名操作员涉众说系统要简单易用。但“简单易用”并不能直接成为需求。需求工程师要耐心和涉众沟通，了解涉众是以什么标准来度量“简单”和“易用”的。

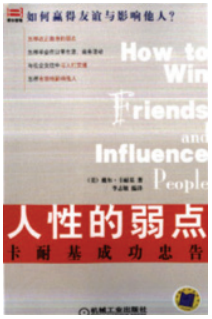
沟通力还包括需求工程师在不同涉众之间协调的能力。涉众往往有很多方，A 类涉众的利益和 B 类涉众的利益可能在一定程度上是冲突的，例如，录入人员希望操作步骤尽量少，但如果因此省略了一些确认和验证的步骤，使用这些录入数据的审批人员、施工人员的利益可能就会受到损害。需求工程师需要平衡各方涉众的利益以得出恰当的需求。

沟通力还包括需求工程师在涉众与程序员（也是一类涉众）之间协调的能力。例如，操作员要求“一键完成操作”，却难为了程序员。

需求启发技术中的访谈、观察、会议等，无不需要人与人之间的沟通能力。而平时程序员更多使用的是和机器的交流能力，这一点也是身兼多职的程序员值得注意的。程序员如果要去承担需求工程师的角色，沟通力可能是木桶上最短的板。

如何培养沟通力？我们习惯于生活在一群有较高学历的软件人圈子内，而涉众往往在学历和技术知识上和我们有差距。要能够和涉众有效沟通，有意识地去改进沟通技巧是必要的。参加一些沟通技巧的训练，或者阅读一

些人际交往的相关书籍，例如《人性的弱点》，这本书可以说是中国引进的最早的“沟通力”书籍之一，10 多年前曾经大热。书中的倾听和赞赏的道理到今天依然没有过时。



表达力

表达力在这里着重指文字表达和组织的能力。需求最常见的形式是以文字方式表达出来的。象用例文档，就是一种规范的、有层次的需求表达形式。用例的写作需要表达力，也培养表达力。

用例（取款）
路径（正常取款）
步骤（系统验证取款金额合法）
补充约束（取款金额必须为50元的倍数）

- 执行者××××××
- 系统××××××
- 系统××××××
- 执行者××××××

用例是一种规范的、有层次的需求表达形式

我们平时习惯的是编程语言的表达能力，写个注释有时也想偷懒，更不用说自然语言表达的文档了。开会时项目主管向大家介绍项目，用词中经常充斥着 “伸缩性”之类的各种字眼，明明只是一个小小的管理系统，却起名叫“××平台”，似乎大家听不懂才说明高深。

如果您希望提高文档的表达力，可以尝试读读这本书——《金字塔原理》。这是一本说明怎样有效写作的书籍，介绍了一种处理写作时文笔不清问题的新方法。

热情

我有好奇心，有探索力，有沟通力，有表达力，也掌握了各种具体的需求技能，就意味着我会积极地尽我所能去向涉众探索需求吗？没有热情作为屋顶，上面提到的各种“力”不能得到贯彻。培养上面提到的各种素质，主要靠需求工程师自己的努力。培养热情则复杂得多。

没有价值感，没有热情。软件公司可能希望通过企业文化洗脑来使需求工程师获得热情。例如讲这样的故事：有个人经过一个建筑工地，问那里三个石匠在干什么？第一个石匠回答：“我在做养家糊口的事，混口饭吃。” 第二个石匠回答：“我在做很棒的石匠工作。” 第三个石匠回答：“我正在盖世界上最伟大的教堂。”故事是很感人，但如果我内心里知道，这个软件项目纯粹是浪费纳税人金钱的政绩工程，它的开发只会有损于社会的总福利，你叫我如何提起热情来，睁眼说瞎话“我在建造世界上最伟大的软件”？

没有利益，没有热情。这个软件系统做好了，我能得到什么好处？做砸了，对我有什么坏处？如果没有这样直接相关的利益，需求工程师是不会有热情去体会涉众的需求的。就象玩斗地主，如果只是扣分，大家只当玩玩而已，没有人会绞尽脑汁，如果扣的是口袋里的铜板，恐怕每一局考虑的就全面多了。

如果您现在所在的软件公司没有给你带来价值感，也没有带来直接利益，就不要用强打精神用“敬业”来折磨自己了，要么请老板改变现状，要么走人吧！

结语

想一想这一种最简单的涉众，婴儿。他哇哇地哭了，他的需求是什么？热了？饿了？要尿尿？有蚊子咬他？还是生病了？应用一下探索力和沟通力，你可能要摸一摸他的后颈，有汗吗？发烫吗？在脑子里查询一下“喝奶日志”，上次喝奶是什么时候？抱起来，摇一摇，哼支小曲……

但是，如果你不是真正爱他的父母，这些事情能做得很棒吗？

（本文首发于《程序员》2005 年第 6 期）

软件工程技术丛书

设计系列

企业应用架构模式

Patterns of Enterprise Application Architecture

(美) Martin Fowler 著

王怀民 周斌 译

UMLChina 审校

UMLChina 训练辅助教材

设计模式精解

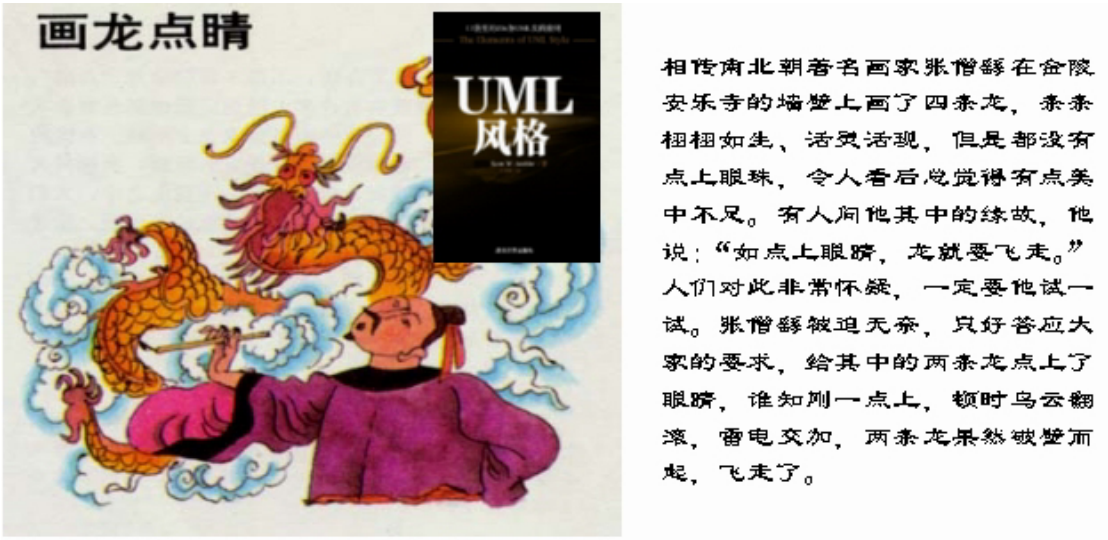
Design Patterns Explained

看不懂《设计模式》？
不用惭愧，别人也一样苦恼。
先把它供起来吧，看这本！

UMLChina 训练
辅助教材

Alan Shalloway & James R. Trott 著
熊节 译

清华大学出版社



它不讲概念，它假设读者已经懂了概念。

它不讲工具，它假设读者已经了解某种工具。

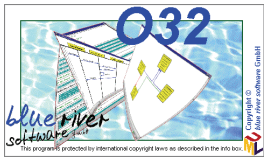
它不讲过程，它假设读者已经了解某种开发过程。

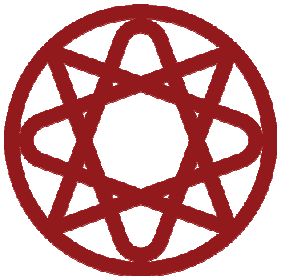
它只是在读者已经了解方法、过程和工具的基础上，提醒读者在绘制 UML 图时需要注意的一些细节。

在这本类似掌上宝小册子中，Ambler 提出了 200 多条准则，帮助读者在画龙的同时，点上龙的眼睛。

UML 相关工具一览 0-P（截止 2005 年 6 月）

整理

工具（最新版本）	厂商&地址	试用允许	UML 版本	支持代码环境	XMI	平台	备注
GModeler 	Grant Skinner http://www.gskinner.com/gmodeler/app/run.html	免费			√	浏览器支持 Flash	线上的 Flash UML 工具。
Kant & Plato 	楚凡科技（中国） http://www.trufun.net/	有试用版		C#, VB.Net, J#, JScript.Net,C++.Net,C, Ansi C++, Delphi, Perl, Php, Python, Eiffel, Java, Caml	√	Windows	
O32 1.3 	blue river software（德国） http://www.blue-river-software.com/products/o32/o32.htm			C/C++		Windows	2004 年停止更新。
ObjectArtist 0.1.1	Sven Daumann（德国） http://www.objectartist.org/	开源		Java	√	Java	支持设计模式。已停止更新。

							
Object Domain R3 	Object Domain Systems http://www.objectdomain.com/	有试用版		Java、C++、IDL、JPython	√	Java	支持多用户，支持 Swing，使用 JPython 2.1 作为脚本语言。客户有 Alcatel 等大公司。
Objecteering/UML 5.3.0 	SOFTEAM（法国） http://www.objecteering.com/	有试用版		Java、C++、C#、IDL、DDL、Oracle	√	Windows, Linux ，Solaris	自动模式支持，产生测试脚本..有专门支持极限编程（XP）的配置。支持 MDA 和 SPEM。
objectiF 4.7 	microTOOL（德国） http://www.microtool.de/objectiF/de/index.htm	有 Demo 版		Visual C++, JBuilder, Visual Café, IDL, SQL, Visual Basic	√	Windows	
ObjectMaker 	Mark V Systems http://www.markv.com/products.html	有试用版					支持大多数建模符号。已停止更新。
ObjectPlant 4.1.3 	http://www.arctaedius.com/ObjectPlant/	共享软件		C++, Java, Objective-C		Mac OS X	
OCL Compiler 1.0	Cybernetic Intelligence GmbH	免费					OCL 检查工具，可以整合到

	http://www.cybernetic.org/prodoc1.htm /bernetic Intelligence						SELECT Enterprise 和 Rational Rose 中。
OCL Parse 0.3 IBM	IBM http://www-306.ibm.com/software/awdtools/library/standards/ocl-download.html	开源				Java	支持语法检查和部分类型检查，无 IBM 官方支持。
OptimalJ 3.3 OptimalJ	http://www.compuware.com/products/optimalj/	有试用版		Java			模式驱动的 MDA 工具。
PLASTIC 2005 AGORA PLASTIC 2005 REUSABLE AND VALUABLE MODELING	Plastic Software（韩国） http://www.plasticsoftware.com/	个人版免费		Java、C#、C++		Windows	
Poseidon for UML 3.1 poseidon for uml	Gentleware AG（德国） http://www.gentleware.com/	Community Edition 免费，其他版本试用	2	Java	√	Java	基于开源项目 ArgoUML 的商业产品，支持多国语言。集成到 Eclipse，使得 Poseidon 利用其他工具的方便性大大增加，减少了许多 import/export 的工作。支持团队建模。下载量已经超过 1,000,000 份。
PowerDesigner 11 PowerDesigner The Leader in Enterprise Modeling Try Now!	Sybase http://www.sybase.com/products/powerdesigner/	有试用版	2	C++, Java, C#, VB.Net,XML		Windows	企业建模、对象建模、数据建模相结合。
ProVision EnterprisePro 4.4	Avoka http://www.avoka.com/proforma/EnterprisePro.shtml	有试用版		C++, Smalltalk, ERWin, VB, SQL		Windows	集成业务建模、需求建模和对象建模工具。

EnterprisePro							
PROSA UML 2004	Insoft Oy（芬兰） http://www.insoft.fi/eng/			C++, Java, C#, COM, SQL		Unix, Windows	可以把状态图和活动图转变成可执行的 C++, Java 代码。支持移动设备开发。
ProxyDesigner 1.0 Developers ProxyDesigner v1.0	ProxySource http://www.proxysource.com/Home.asp	免费					支持 Patterns。而且把 ProxySource 社区集中联合起来，分享各自的模式、架构、设计。
QuickUML 1.1 Excel Software Software Engineering Made Easy	Excel Software http://www.excelsoftware.com/					Windows, Linux, Mac OS X	多平台原生支持的 UML 工具。