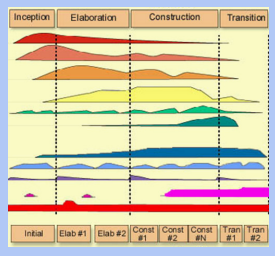
第 37 期（2004 年 5 月）



UP

X 非程序员
-Programmer
软 件 以 用 为 本

投稿：editor@umlchina.com
反馈：think@umlchina.com

http://www.umlchina.com/

了解 UMLChina

软件以用为本。不能为客户提供所需价值的软件是可悲的。以前，开发人员以“专家”自居，客户心存敬畏，认为自己“不懂”，不敢提需求，不敢评判。最终的结果是产生了许多“鸡肋”软件——软件做出来了，也能运行，却不提供想要的价值！现在，客户的胆子大了！软件只不过是帮助我完成业务、提高效率、赚取利润的工具而已！我要的软件要能够为我的业务提供价值，运行架构还要符合我的实际情况，还要有足够的可靠性、灵活性、可用性…。软件组织的“好日子”不再有，不得不关心这些问题：如何捕获有价值的用户需求，如何组织需求，如何获得良好的软件结构，如何获得稳定的性能，如何提高软件的可用性…。

软件（技术）以用为本。不能为软件组织实际使用的技术是可悲的。为了提升软件质量，许多软件组织都在谋求技术的改进：需求的技术、分析设计的技术、增量迭代的技术、构件组装的技术……我们搜索，我们买书，我们看书，我们讨论，我们试用...但学到的技术如果不能最终用于自己的项目之中，将是极大的浪费。而这最后一段路最是艰难。

相信【软件以用为本】。为此倾注全力，不断实践，苦心研究，采用一切可能的方式，帮助软件组织朝着这个目标进发——让软件变得好用，让技术变得好用：

- UMLChina 训练——“聚焦最后一公里”，专注于上门为企业作 UML/UP 团队训练。训练完全和受训团队当前项目结合，【训练的过程就是指导团队用所授 UML/UP 方法实作团队当前项目的过程】。UMLChina 在团队训练方面已经有了丰富的积累。从东北到西南，已上门为超过 50 家企业提供服务，涉及各种类型的软件开发项目。
- 项目指导——是 UMLChina 训练的延伸。UMLChina 专家带领团队用 UML/UP 方法做项目，使团队顺利行走在 UML/UP 的路上。
- 专家讲座——请来国外国内的顶级专家，传播他们的知识和经验。著名专家有：Gerald Weinberg、James J. Odell、John Vlissides、Kent Beck、Alistair Cockburn、Martin Fowler、Scott W. Ambler、Bruce Powel Douglass、Alan Cooper、Roser S. Pressman、Kendall Scott...等。
- 新闻——全球如发生和 UML 相关的事件，都尽力为您及时报送。
- 《非程序员》——和《程序员》同年（2001）诞生，却一直保持自己的独特风格，为众多高阶软件人员所青睐。每月发行一期，杂志将长久保持免费下载的电子形式。
- 书籍——继 2002 年《人月神话》创下记录之后，2003 年《最后期限》、《人件》、《人月神话》分别排在 china-pub 总销售榜的 1、3、4 名。随后 UMLChina 把精力集中到 UML/UP 相关图书：《有效用例模式》、《UML 风格》、《探索需求》、《PEAA》、《敏捷数据》…
- 讨论组——2000 年下半年开始设立，目前注册用户已经超过 40000 人。

UML 被 SUN“本地化”

[2004/5/21]

总部位于 Calif 的 Santa Clara 网络计算机制造商和 Embarcadero 科技公司达成的协议将在其 Java Studio 企业平台的下一个版本上增加 Describe UML 6.1 版本。

UML 是一个用于标识和可视化复杂软件设计通用符号语言，尤其面向大型、面向对象项目。UML 基于过去的符号方法诸如 Booch、OMT 和 OOSE 创建。

SUN 声称，UML 的 Descirbe 部分将缩短开发者的学习周期。另外，代码将和 UML 模型保持同步，这将加快开发者直接从图形来浏览和维护代码的速度。

SUN 的一个产品线主管 Robin Smith 告知 internetnews.com ，加强了 UML 的版本将是今年稍后会发布的 Java Studio Creator 2.0，早期的估计时间是在 6 月会发布。

Smith 说，“UML 将使得开发人员可以从模型生成代码，架构师可以通过可视化的工具来设计应用结构，并生成代码，而且同样可以从代码对模型产生影响，这是我们的目标。”

和 Embarcadero 的协议中给予了 SUN 无限制分发和使用 Describe 的基于标准的建模技术的权利。之前，开发人员只有通过分别付费来分别下载这些软件。SUN 表示，目前，5 美元每个 license 的价格中将包括使用软件的权利以及享受开发人员的服务，并每年更新。或者 1895 美元可以购得永久的 license（perpetual seat license）。

Smith 说，另一个障碍就是，Embarcadero 只面向 Windows 平台。合作将使得 Java 开发人员可以使用 Java 语言在支持 Windows、Linux 和 Solaris 的多平台上使用该 IDE，这项功能将直接由 SUN 支持。

“顾客们会说，啊，IBM 的 WebSphere Studio 里面有 UML 了，Borland 的 Jbuilder 里面也有了，SUN 的呢？”Smith 说，“现在，我们可以看着他们，说，我们也有了可以在所有这些平台上台运行的集成的 UML 功能，并且和我们的 Java 企业环境绑定接口。现在我们具有了和其它厂商同样的功能，在某些方面，还超过他们一些。”

Smith 说，增加 Describe UML 同样有利于 SUN 的其它各种 Java 企业系统，包括 Java 系统目录服务器，Java 系统标识服务器、Java 系统 Portal 服务器，Java 系统消息服务器以及 Java 系统 Web 服务器。

（自 internetnews，UMLChina 袁峰 摘译，不得转载用于商业用途）

征稿

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

业务驱动开发者的趋势

[2004/5/17]

在软件开发领域几个重要的趋势背后，是对简化开发以及使得软件开发人员和业务人员更紧密的合作的推动。

第一个趋势是业务和系统分析人员对 UML 更广泛的应用。第二个就是敏捷开发和极限编程的日益被接受，这两个相互关联的趋势正使得业务人员更多地参与到开发过程之中，给予更多的反馈。最后，还有面向方面开发的趋势，它使得开发人员可以为一个或者更多的应用提供例如安全性等重要的功能。

UML

UML 的最大潜力就在于它具有从软件开发中剔除众多“臆想（guesswork）”的能力。UML 是定义良好的符号集、图形表达方式以及语法。它使得业务和系统分析人员可以通过画图来向软件开发人员描述预期的应用程序。这其中，编程的专门技术并不是必不可缺的，但了解系统工作的知识是（不可缺少）的。

UML 起源于 1997 年，是由 OMG 推出的一个标准，但现在才渐渐成熟。UML 开始被各企业的开发人员所使用并已经被植入多个产品中，诸如 IBM Rational 的 XDE 以及 Borland 公司的 Together Control Center。

UML2.0 的最终版本尚未完全确定（is pending），但其关键部分已经在去年秋天被 OMG 投票通过。UML2.0 克服了 UML1.0 的缺点，UML1.0 只能表示一个单独的软件应用，而不能表示一系列相关联的应用或者系统。

IBM 历史上著名的工程师 Alan Brown 认为，UML2.0 提供了业务人员和软件开发人员之间的桥梁，“他们并不是完全同步的”，可以在一个业务需求工具中创建得到一个 UML 模型，并将其导入到开发工具中。

但对于 UML2.0 更高版本，大家有着更多的期待。开发工具们正在朝着更加可视化的建模环境努力：对象可以拖拉，并通过线条连接。在 UML 的未来版本中，开发中系统的错误将会被可视化地显示出来，并在更早的阶段被捕捉。PivotPoint 科技公司的 CEO 和 OMG 分析和设计专家组（该小组负责制定 UML2.0 规约）的副主席，Cris Kobryn，认为，“如果你要对 1 万行代码负责，要是有一个清晰的表或者图来帮助的话，不是要清晰得多吗？”，因此，开发人员将不会去翻看 20 页长的头疼的代码，而会选择去看一个详细的 UML 图，来发现错误，并进行改正。

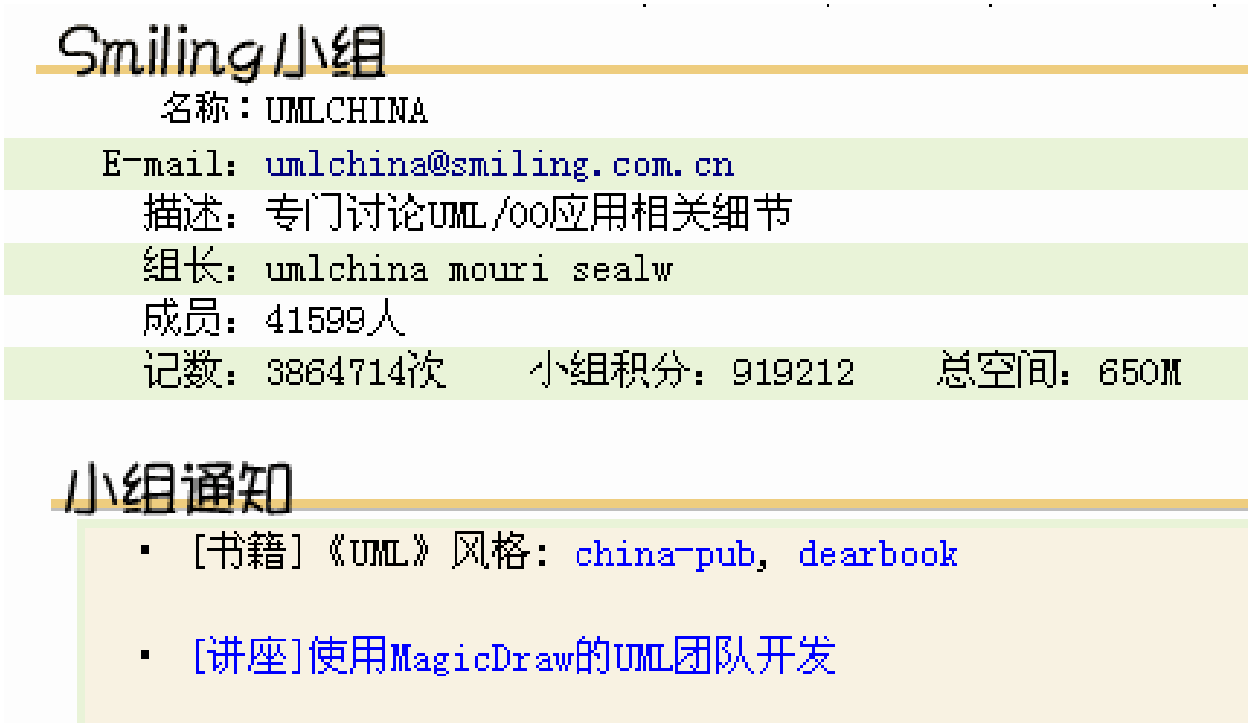
敏捷开发以及极限编程

敏捷开发是一个年轻的运动，在技术上并没有太多的变化，只是通过一个结构化的方式拉近了开发人员和业务人员之间的距离。其方法依赖于业务人员基于软件原型的频繁反馈以及保证软件满足业务需求的频繁测试。

极限编程，其中有：两个开发人员结对编程，一个负责开发代码，另一个负责评审开发者的工作；检查是否满足业务需求；紧密地和业务人员合作。这里有一个基本的认识：当两个开发人员从各自不同的观点来一起工作时，会加快开发速度并尽早发现错误。敏捷开发和极限编程有一个共同的目标：在开发过程中尽早地满足业务需求。

顾问公司 J. Walter Thompson 的助理开发主任 John Tripp 在其同事中进行了一个长达一年的极限编程测试，发现开发人员在该方式下“开发了多得多的代码”，John 的团队开发了一个协作过程入口（portal），提供给分布在世界各地的 2600 个雇员使用，来管理项目工作流、时间表以及项目经费。

（自 informationweek，UMLChina 袁峰 摘译，不得转载用于商业用途）



Cutter 会议：敏捷在升级

[2004/5/7]

几年前，随着 UML、RUP 以及极限编程的风行，敏捷建模曾经引起大家的注目。现在，它哪去了？回答这个问题，没有比 Jim Highsmith 更合适的人选了，他是《自适应软件开发》、《敏捷软件开发生态系统》的作者，敏捷联盟奠基人之一。

实际上，作为 Cutter 协会的会员，Highsmith 认为他看到敏捷建模的概念正在管理领域得到应用。Highsmith 在本周的 Cutter 在剑桥的高层会议上指出，过去相对较窄的敏捷建模的概念中，将加入越来越多的敏捷实践。

在敏捷建模领域，XP 概念中迭代的开发技术带有软件项目快速初步计划的概念。可以说，敏捷建模在极限编程方法上增加了一些结构，但没有变得“过于结构化（over-structured）”，这是某些人对传统的软件项目管理的评价。Highsmith 认为，“敏捷就是平衡适应性和结构这二者的能力”。

“我们看到敏捷实践从最早被接受到成为主流，我们现在要做的是面对它。”，对于有些人的问题，是否需要从极限编程转为温和的编程，Highsmith 进行了斥责，“我们必须坚持敏捷系统中一些基本的价值”。

软件应当被看做是一个更大的团体过程中的一部分，产品的单元应当可以后来者所重用。Highsmith 认为，随着这样的一个认识的被接受，敏捷实践的倡导者们已经开始从“项目” 角度逐步转向到从“产品”角度看问题。

Highsmith 说，对一些敏捷拥护者而言，下一步将是在组织的探索项目中取得成功。他用到了石油行业的情况作为类比，这个行业的项目通常分为两大类：勘探性的开采，特点就是适应性和风险管理；产品性的开采，公司对已知的油田进行开采，项目更多地关注在优化和操作上。

另一位在 Cutter 会议上发言的是 Qwest Communications International 公司的方法学家 Jean Tabaka。她介绍，她和她的同事正在将敏捷实践应用于更大的团队。在一些可以描述为中型的项目的基础上，对于更大的敏捷项目而言，专家指导以及合作的观点是成功的关键。另外，还需要清晰的反思和回顾，“你必须知道哪些（实践）是有用的，哪些是没有用的。”

（自 adtmag，UMLChina 袁峰 摘译，不得转载用于商业用途）

Select Business Solutions 引导新的模型驱动架构解决方案

[2004/5/7]

今天 Select Business Solutions 宣布了其最新的开发工具集，基于 MDA 的 Select 解决方案的诞生。Select Business Solutions 在业务过程建模、UML、数据挖掘、基于组件开发（CBD）、面向服务架构（SOA）以及软件重用/工件（Artifact）管理等软件设计和开发行业的各个领域占据了 10 数年的领先地位。

Select 的 MDA 解决方案基于 Select 的 UML 建模工具及其 CBD/SOA 软件开发过程，为 IT 组织提供平台无关性、软件应用的高质量以及更高的开发效率。引人注目的是，Select 是在通过利用预载的转换和模式来生成高达 80%的代码的情况下达到的这些成就。

国际对象管理集团 OMG 的主席和 CEO，Richard Soley，说： “OMG 很高兴看到 Select 在其新产品，Select Solution for MDA，中结合了我们的模型驱动架构标准”。

Select 支持 OMG 并力助其基于 MDA 的解决方案。诸如 viewpoints 和自动转换等很多已为 Select 顾客所熟知的概念现在已经深入到 MDA 标准之中。现在，作为 MDA FastStart(TM) 项目成员，Select 将继续加强与 MDA 的紧密合作关系。MDA FastStart 是 OMG 设计和管理的一个特殊项目，目的是帮助 IT 组织熟悉 MDA 的概念并快速将 MDA 应用到他们的任务－关键软件开发活动中。

Select Business Solutions 的 SVP 及主管 Hedley Apperly 介绍，“共享架构师的知识，并提供给他们平台独立的设计支持，使得这些设计在今天和将来都能够保持复用，通过这些，Select Solution for MDA 变革了 IT 组织的软件开发方式。开发人员可以自动化地将一个抽象的业务模型在平台和技术相关的设计上描述出来，甚至生成代码。我们已经打破了业务和软件开发人员之间的沟通鸿沟。现在业务人员可以保证他们的 IT 开发人员不仅在正确地开发系统，还可以保证他们在开发正确的系统。Select Solution for MDA 不仅解决了这个沟通的问题，也支持基于组件和服务的开发。Select 的软件资产库将继续支持紧密结合 MDA 的供应商/顾客（Supplier/Consumer）开发过程。”

Select Solution for MDA 是一个崭新的基于 UML(TM) 的建模和转换工具，可以正向、反向工程，并保证模型和代码的同步。Select Solution for MDA 帮助用户保证计算无关模型 CIM (Computation-Independent Model)、平台无关模型 PIM (Platform Independent Model)、平台相关模型 PSM（Platform Specific Model）之间的同步并支持进行中的开发、维护、加强以及集成等任务。所有模型和代码都可以并行工作，通过双向同步提供了随时的变更协调。

Select Solution for MDA

*加速开发生命周期，自动生成各种模型和诸如 Microsoft Visual Basic、Microsoft Visual C++ 和 Microsoft Visual C#等各种语言的代码。

*帮助用户从模型中生成更多的代码，从而节省开发时间。

*提供架构和技术的无关性，保证用户的模型和解决方案具有更高的适应性，可以应付未来的变化。

*缩短提出要求的业务人员和软件实现的开发人员之间的“技术鸿沟”。

（自 Yahoo，UMLChina 袁峰 摘译，不得转载用于商业用途）

征稿

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

OASIS 委员会通过通用业务语言（Universal Business Language）草案

[2004/5/4]

OASIS 委员会通过通用业务语言 UBL（Universal Business Language）1.0 草案。现在可以免费使用，UBL 定义了一个用于业务文档的通用的模块化的 XML 交换格式，并可以根据特定的行业要求进行扩展。

该规约的这个版本经历了六年的制定过程并有一年时间用于公开评审。OASIS 在一个声明中提到，“最终结果将用于商业实施以及开源软件开发。”

UBL1.0 委员会草案现在已经提交给 OASIS 以作公共评审，之后还会有一些标准化的规程要进行。

UBL v1.0 中包括一个用于可重用数据组件的 XML Schema 库、用于通用业务的 XML Schema 的小的集合、通用订单到发票（order-to-invoice）贸易领域的有用的文档，以及对特定的贸易关系的 UBL 自定义的支持。通过 XML 识别的途径，UBL schema 拥有模块化、可重用以及可扩展等特性。

目前，UBL v1.0 正在广泛分发以作公共的评审，在一个 zip 压缩包中包括了文档、XML Schema、UML 图形、spreadsheet 模型、格式化的规约、样例实例以及其它的组件。

（自 cmpnetasia，UMLChina 袁峰 摘译，不得转载用于商业用途）

Gentleware 推出 Poseidon for UML 2.3 版本

[2004/5/4]

总部位于汉堡的国际性建模软件开发商 Gentleware AG 今天向市场推出了 Poseidon for UML 的最新版本。和先前版本不同之处在于，新版本中包含了对 UML2.0 标准元素诸如接口（ports）、连接器（connector）以及新的图形的增强支持，Poseidon for UML 是第一个允许地理分布的开发成员可以同时操作同一模型并可以看到同事的实时修改的 UML CASE 工具。该软件现在已经可以从 www.gentleware.com 得到。

在三月份的 CeBIT 展会上，Gentleware 推出了企业版，并展现了对 Poseidon 的团队功能的巨大兴趣，接下来的重大举措就是在 UML2.0 的图形支持以及对新的 UML2.0 元素诸如接口和连接器等的支持，这些举措都使得 Poseidon 相对于其它 UML CASE 工具的不同之处更加突出。

Gentleware AG 总裁 Marko Boger 博士认为，“通过 2.3 版本，我们再次证明了：和大多数其它开发商所不同，Gentleware 已经符合了即将到来的 UML2.0 标准。我们将坚定不移地沿此路前进，今年下半年，你们将看到 Poseidon 已经 100%地基于了 UML2.0 的元模型仓库。”

Poseidon for UML 目前有共享版（免费），标准版本（199 欧元），专业版（699 欧元），企业版（1249 欧元另加服务器费用），以及嵌入式版本（1249 欧元），可以从 www.gentleware.com 得到。

从下面地址可以下载可打印的 UML2.0 图形样例：

http://gentleware.com/company/pressReleases.php4

（自 yahoo，UMLChina 袁峰 摘译，不得转载用于商业用途）

Quocirca 直言：编程已死？

[2004/4/23]

有一些软件工具厂商已经走在了 OMG 的前面。比如说，新出来的 Quovadx 公司在应用模型生成代码方面已经到了崭新的水平上，书写代码已经变得越来越失去需要。不可否认的是，Quovadx 目前的大部分示例都是和特定的行业相关，例如医疗和金融服务等。但是它使人们再次关注－我们离完全告别代码的时代还有多久？

答案是，实际上已经不远了，如果 Select Business Solutions 提供的最新信息可信的话。SBS，曾经是 Rational Rose 公司在英国的最大竞争对手，他们目前已经把 MDA 的概念和设计模式结合在一起，设计模式实质上就是将过去你写过的不错的代码结构，详细地说明并以建模术语的形式进行复用。通过选择这些设计模式并作为代码生成器的输入，就有可能生成绝大部分的代码，即使不是全部。SBS 并不是唯一一家这么想的公司。但是，它是第一个实际实现了这个思路的公司。

当然，事情总是说起来容易，要做结论还得进行仔细的测试。也许问题不是谁获得了胜利，SBS 还是 Quovadx，Rational 还是 Borland。而是，他们将我们置于这一无情的事实中：大部分的代码都将自动生成。当然，会有人抱怨（自动生成的代码）效率太低，也就是说总是需要专家来开发高质量的代码的。但是，对其余（非专家）的人而言，无疑是致命一击。

同样，代码自动生成也意味着程序员不需要考虑甚至不需要知道这些代码。由于 IBM 的软件部门包括软件开发和企业管理，这时候它的优势就显示出来了：比如说，他们可以关注标准的事件日志如何内建到代码生成器中，支持自动的计算系统。

同时，我们应该在主要的应用开发商的角度来考虑这些。他们都会停止开发那些可以集成在应用中的功能。另外，一些独立的解决方案提供商将会致力于提供面向特定领域和市场的垂直解决方案。他们都将从 MDA 以集成为中心的观点中受益，在这一观点中，模型就是那些预制的组件和从专门机构那里购买的服务之间的粘贴剂。

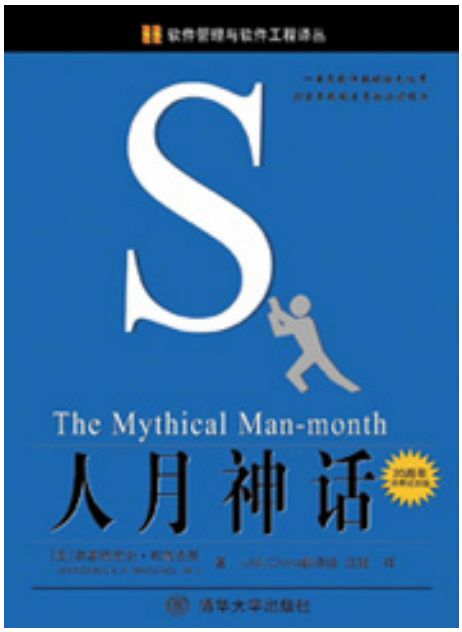
没有程序员会因为 MDA 丢掉工作，原因很简单：总是有新的有趣的事情等待开展。但是，MDA 将会带来非常大的变化。未来，如果模型是国王（译者注，以模型为中心），编程的责任就会转移到业务上，并进入 IT 客户的市场。这不是坏事：保险公司关心保险的事情，制药公司努力研制新药品，而不是花掉大半的预算去重新升级旧的代码或者努力开发新员工喜欢的新的系统。

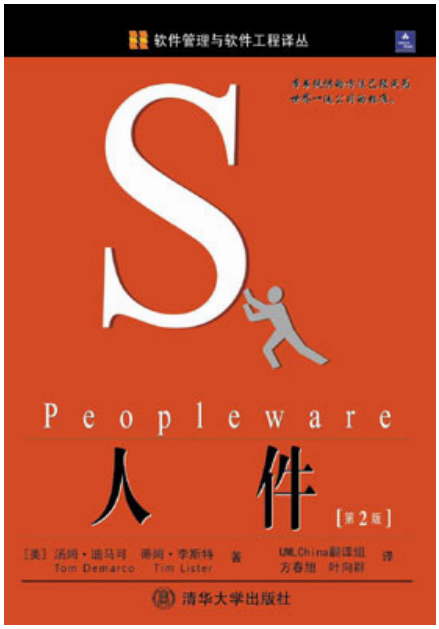
MDA 还有路要走－随着 MDA 被各种决心使用的它的业务领域所采用及接纳，MDA 将会获得一致认可，但，还是需要时间。

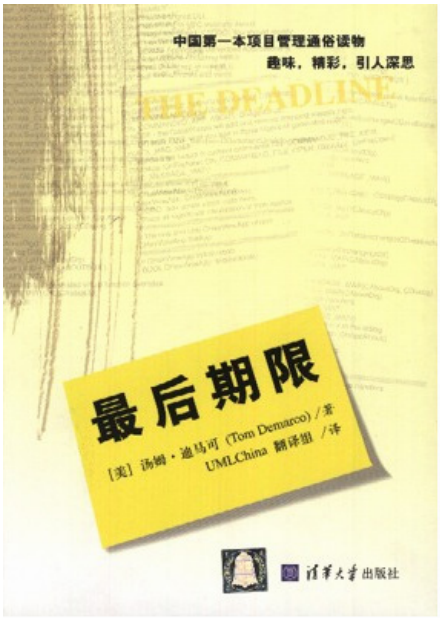
（自 silicon，UMLChina 袁峰 摘译，不得转载用于商业用途）

征稿

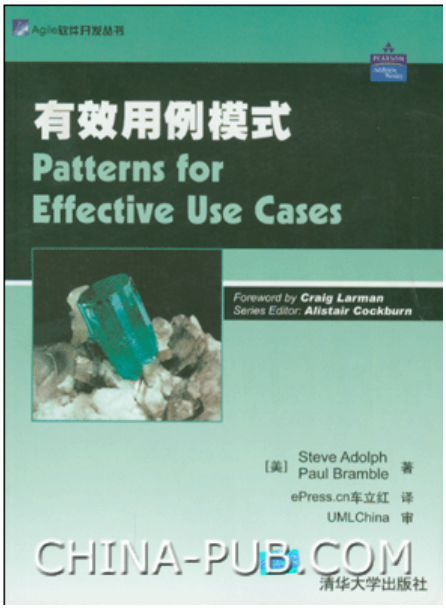
http://www.umlchina.com/xprogrammer/xprogrammer.htm

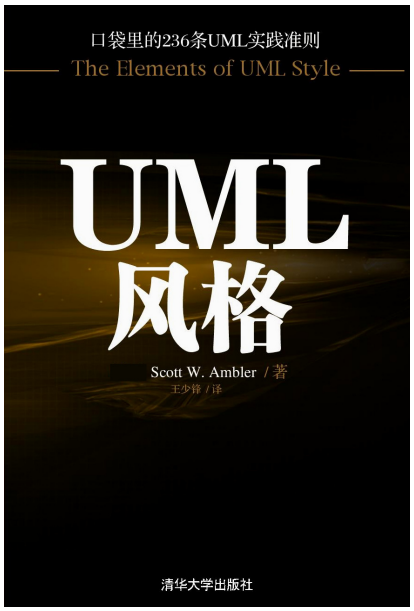


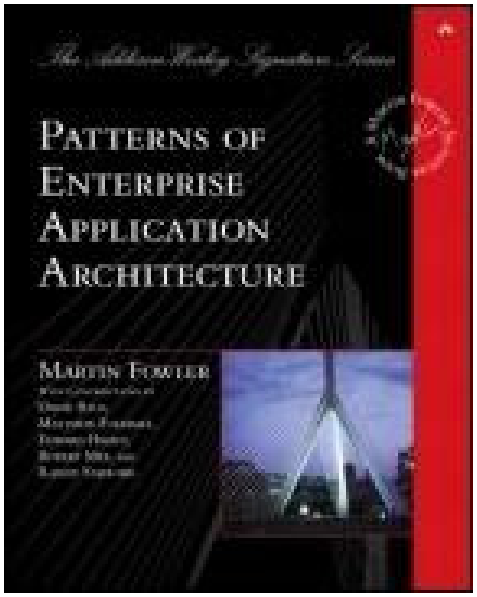


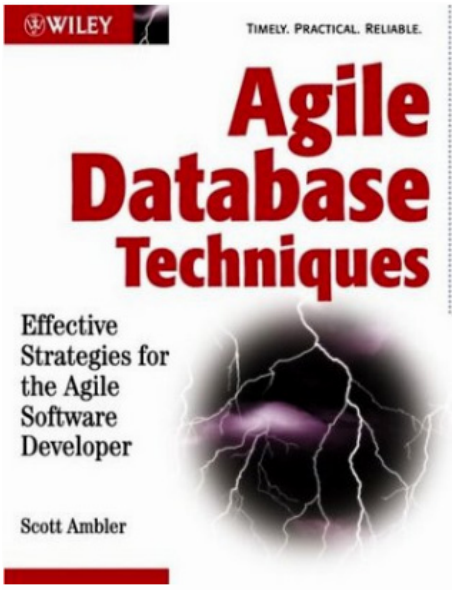


- 2002 年《人月神话》创下销售记录
- 2003 年《最后期限》、《人件》、《人月神话》分别排在 china-pub 年度总销售榜的 1、3、4 名









MDA 辩论

车皓阳 译

查看评论

Axel Uhl

Scott W. Ambler

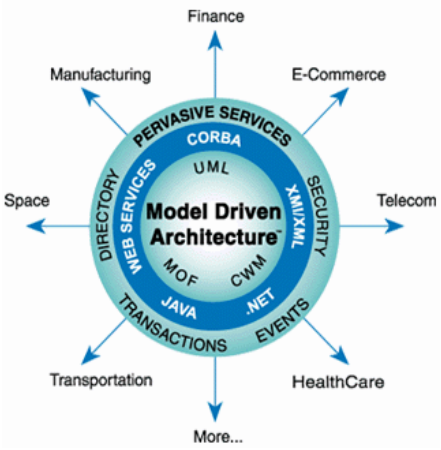
正方：模型驱动架构正在步入成熟期

MDA 是下一轮逻辑演化中的一步，它补充了软件工程行业中的 3GL。

Axel Uhl在Interactive Objects Software公司负责IDE ArcStyler开发的小组中担任软件架构师。他也在积极地为OMG MDA的标准化而努力着。可以通过邮件axel.uhl@io-software.com与他联系。

一位芯片设计师使用他在工作站上的工具来对微处理器的逻辑进行建模。这个工具可以帮助他使用预先创建的组件和模式，并验证设计的可靠性，帮助找到一条好的路径和芯片布图。这位设计师测试了模型，然后直接把它发送到晶片生产部门。在那儿，基于模型，创建掩膜、相移、蚀刻等操作都是自动进行的。如果芯片设计师是软件工程师，一些人就会希望他把模型打印出来，或者丢掉直接找到晶片生产部门的工程师，和他们谈论芯片的外观，以及诸如此类的事情。当然，我现在把它弄得很简单，但你已经掌握了其中的意思。

软件工程师都曾经历过类似芯片设计和其它制造领域的痛苦的黑暗时代，现在是个好时机，可以转换到相同的成熟层次。汇编语言的复杂性、表现力的匮乏以及移植性问题告知我们要使用第三代语言（3GL），我们开始认为它们与相应的编译器一起，理所当然地都是功能强大的、不可或缺的软件开发工具。它们帮助我们处理注册任务、跳转预测、内存管理代码，等等。我甚至还没有讨论过跨操作系统平台移植的问题。



我们正处在向OMG模型驱动架构（MDA）转型阶段的边缘，这样说的原因在于，在过去几年中我们使用的技术变得更加复杂。而且，技术的变化比技术支持的业务来得更快。如果我仅仅需要在UML图中的两个分类器（classifier）之间绘制一个关联，那么为什么还要费尽周折在两个EJB组件之间实现双向可导航的关联呢？而且，如果这样做的结果并不可移植，为什么我还要陷入因手工实现关联而带来的技术细节中呢？结果是，你可以部署3GL以确保这些平台的应用，并可以使用汇编语言创建一套基于Web的客户关系管理系统；只不过它们不是表达系统结构和行为最适当的方法而已。

我不是说我们必须用一个或多个UML模型来表达系统规约中的每一项细节。但这比我们只用3GL的状况肯定会好许多：模型不一定非要变成平台依赖的。合理的MDA工具可以让你在不同的抽象级别之上增加系统规约，并保持它们同步。marks（可以附加在模型元素之上的特定映射的注解）的概念使模型自身免除不必要的平台特征。使用这种技术，我们可以用最适合此目的的方式提供全部的系统规约。此种工具的例子有Interactive Object公司的ArcStyler(www.arcstyler.com)和Compuware公司的OptimalJ (www.optimalj.com)。

MDA有一个坚实的基础，包括MOF和UML，两者都是流行的、业已成熟的形式化方法，用以确定元模型和模型。与协定使用ASCII编程的日子比起来，约定大多数建模活动采用UML让我们今天可以走得更远。现在，对于用MOF给定的元模型，你可以自动地创建UML Profile。这几种强大的技术，特定领域的元模型以及相应的UML Profile都已经标准化了，例如，企业分布式计算或者J2EE的Java Specification Request 26。较好的MDA工具可以让用户扩展并定制现有的模型转换规则。同样，确定可移植模型转化的标准也在进行中（见MOF 2.0中查询、视图和转换等相关工作）。

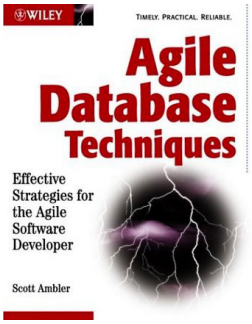
许多工程项目使用适当的工具支持，成功地部署了MDA（参见www.omg.org/mda/products_success.htm）。UML库和工具的开发自身也是极好的应用MDA的例子。你可以从OMG网站下载模型，直接将它们用作MDA模型转换的输入。在这个场景中，UML元模型的尺寸符合使用MDA方法产生的巨大节约。

软件工程师感到正被强力拉向以模型为中心的开发。大学里也已经开始教授UML，书架上也开始充满MDA的书籍。MDA正流行起来，就象几十年前3GL那样（现在依旧如此）。我们现在正在迈开下一步。从我所见到的事情来看，我确信MDA是发展方向。它正在步入成熟期。

反方：敏捷建模已经足够好了

已经有十年了吗？观点“超级建模工具”再次扬起了它丑陋的头颅。

Scott W. Ambler是Ronin International公司的高级顾问。他是敏捷建模方法学（www.agilemodeling.com）的精神领导者，Software Development的专栏作家，Flashline（www.flashline.com）软件开发生产力协会的成员。他最新出版的图书是《Agile Database Techniques》（John Wiley & Sons，2004）（本书中译本将由机械工业出版社出版，译者为UMLChina李巍。）。可以通过邮件scott.ambler @ronin-intl.com与他联系。



对于模型驱动开发采取的方向，我还并不是太相信。不要误解我了，我是建模的坚定信徒。只是我在想，对于开发来说，已经有了更多。下面是我的观点：我们需要分辨产生式MDD和敏捷MDD。产生式MDD，根据OMG MDA的概括，基于这样的观点，人们会使用复杂的建模工具创建非常复杂的模型，它们可以使用这些工具自动转换，以反映不同的部署平台下的实际情况。伟大的理论，就如同是世界是平面的这种想法一样。我的观点是，产生式MDD对于这一代程序员来说是产生迷惑的原因。敏捷MDD仍然需要努力才会成功，但至少它会有机会成功。

我相信建模是在编码之前思考问题的一种方法，因为它可以使你在更高的抽象层次上进行思考。你也可以沿着测试驱动的开发主线，在编写功能代码之前写出测试。但本次讨论主题并不是TDD，因此我也不会多说什么。

我也由衷地相信被我称为敏捷建模驱动开发的东西（AMDD）。敏捷建模已经足够好了，它可以满足目标。因为这种好也只是相对地好，在适当的情况下，你可以把一张草图，一张UML状态图，或者详细一点儿的物理数据库模型想象成敏捷模型。遵循AMDD方法，当我和用户一起工作探索和分析他们的需求时，我一般使用非常简单的工具，例如白板和纸。简单的工具容易使用，包含广泛（涉众可以积极地参与到建模中），灵活，没有约束。它们正是我们在探索问题领域和确定系统架构时所需要的。

谈到详细AMDD设计建模，我经常会使用复杂的建模工具，例如Together ControlCenter（www.borland.com/together/controlcenter）或Poseidon（www.gentleware.com）用于对象建模，Erwin（www3.ca.com/Solutions/Product.asp?ID=260）用于数据建模。当你试图生成代码时，这样的工具就会有意义。AMDD恳请你用最简单（不只是简单）的工具工作，依赖于你的目标，有时最简单的工具会变得最为复杂。然而，许多开发者选择不使用这类工具进行工作，他们宁愿只使用象Eclipse（www.eclipse.org）或Visual Studio（http://msdn.microsoft.com/vstudio）这样的集成化开发环境工作。每个人都不需要，或不想要复杂的建模工具。特别是使用AMDD，程序员编写代码与模型开发的步调一致，AMDD推动演化方法，实现是迭代和渐进地发生的。

对于我来说，产生式MDD的基础是不可信的且不稳定的。首先，我们仍然没有一种标准的建模语言，可以满足真实世界的需求，这使得开发者很难在一起有效地工作。我曾经创建的每一个系统都有前端用户界面和后台数据库，但UML并没有描述这些问题。是啊，你可诡辩说，你只需要在现有UML图上应用一些构造型，但正如我在UML数据建模概述中的工作那样（参见www.agiledata.org/essays/umlDataModelingProfile.html），对此来说，含义更多，我只是触及表面而已（我热切希望收到有关我的工作的反馈意见）。

再者，人们只是没有建模技巧。教授人们学习如何在白板上创建时序图和状态图，已经足够困难。在我们行走或奔跑之前，我们首先需要学习爬行。在未来十年中，如果大多数开发者开始这样绘制，我就会被深深感动。

第三，工具还没到那儿。有一些有趣的工具，例如Bridgepoint（www.projtech.com/prods）和Codagen（www.codagen.com），但它们用得还不广。OMG似乎还在支持这种想法，通过XML元数据交换标准，你就可以组合出一套工具集，但我同样会提出疑问。如果想要这种观点工作起来，工具供应商需要XMI规范如定义那样产生实际的支持。过去使用CORBA ORB（对象请求代理）互操作的经验表明供应商常常是表面宣布支持标准，实际上却为了竞争实现自己的版本。很有意思，大量建模工具供应商都支持XMI，但我还不能找到一套工具可以让我在两者中建模并可以来回导出不丢失信息。

现在的问题是，AMDD是它对于大多数开发人员来说只是个草图，最多，可执行MDD的观点仅在几种情况下是可行的。MDD社区应该关注实际可用的东西，而不是象牙塔中的理论。

Axel 的回应

Scott和我都认为建模总的来说是好的。但是，我不同意Scott所说的产生式MDD，或者说模型驱动架构是造成迷惑的原由。好多工业软件开发项目已经证实了这一点。比方说，德国银行Bauspar和澳大利亚国家铁路报告第一期MDA项目节约了40%。有了MDA，通过MOF，你可以集成任何领域专家与你一起使用的建模语言，就可以从自动模型验证和转换中得到好处。

MDA并不阻止迭代和渐进开发的演化方法。我们用MDA工具ArcStyler去开发ArcStyler，迭代前进，没有出现问题，尤其是没有MDA相关的问题。

所有的MDA建模语言都由同一元建模语言MOF来形式化确定，MOF的基础很坚实。用于模型表示的UML Profile也被标准化了，正如用于模型转换的规约语言一样。

UML是有意保持那么简洁的，没有特定于领域的内容。相反，UML提供轻量可扩展性，允许为MOF形式化确定的元模型创建UML Profile。你甚至可以使用MDA自动创建Profile。

我认为AMDD造成了建模成本，但在收到真正成效之前就已经停止了。我会一直从使用MDA的模型中得到好处，正如Scott从测试中得到好处一样。我在大量真实的项目中见到MDA赢利如此多次，因此，我坚持认为它会起作用的。

Scott 的回应

唉。好消息是Axel和我的意见并不一致。本来在许多情况下，那会是令人头疼的事情，我关注眼前，Axel展望未来。我对MDA并不热火的原因是因为我过去曾经听说过许多类似的想法，它们全都败得很惨。

- 集成机辅软件工程。I-CASE，在二十世纪八十年代出现，失败的原因是因为它们不能与技术变化保持同步。很少会有开发人员拥有相应的建模技能，或想要学习它们的欲望。尽管这些工具生成了80-90%的代码，但剩下的10%往往会花费90%的功夫。
- 应用开发周期。不仅市场没有准备好迎接AD/Cycle，透过IBM我也看到AD/Cycle真正的目标在于销售产品和服务，而不是由利他主义来启动的。
- 公共对象请求代理架构。尽管大多数CORBA供应商使用这个标准完成编译，至少它们部分上让它们所谓“标准的”ORB在一起工作是非常困难的。很明显，供应商宣传它们的工具“兼容CORBA”，他们在其中发现了巨大的市场利益，但如果使得它们可以和竞争者的产品一起工作，则会获利很少。

谈到MDA时我已倍感疲倦，因为我并没有看到如何摆脱和I-CASE、AD/Cycle和CORBA相同的问题。不要误解我，我很乐意看到MDA成功，因为我象大多数开发人员一样，都会提前建模，喜欢使用那些可以增加生产力的好工具来完成工作。我只是不想屏住呼吸一直等待而已。

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

用例和方面——无缝的协作

Ivar Jacobson 著，Brian Sun 译

查看评论

摘要

面向方面的编程（Aspect Oriented Programming, AOP）是一个“缺失的连接”，它允许你逐个用例地切分系统，并适用于所有的生命周期模型。它将戏剧性地改变我们理解复杂系统的方式，新特性如何被添加到系统的方式，以及系统如何被实现和测试的方式。AOP也能在软件开发中加入一个新的重用维度。而且这些已经开花结果了。

1 引言

用例（Use Case）普遍地适用于需求说明。用例开始于需求，在分析与设计阶段被翻译成协作图，在测试阶段被翻译成测试用例——这是用例驱动开发背后的中心思想。我们可以把系统看作一个面包，将它切成若干片。借助用例，我们可以将系统切分成多个用例分片，且几乎包含了来自生命周期每个阶段的元素！它近乎是正确的，因为当前(1)我们编写一个组件或类需要整合源自多个用例的代码，这样独立的分片就被分解，无法识别。(2)当前的UML版本在使用用例时支持扩展机制(使用<<extend>>等等)，但它既没有在协作图、组件图和类图等分析和设计元素中，也没有在传统的实现环境如Java或C#中得到支持。根本的问题是当前所使用的语言的限制。

面向方面的编程（即AOP，这里指的是一种实现技术）是一种“缺失的连接”。它允许我们清晰地逐个用例地将切分系统，并适用于多种模型（用例模型，分析模型，设计模型，实现模型等），因此用例自始至终直至代码阶段仍是分离的。这样我们就实现了“关注点分离”。事实上，我们是通过引入与组件模块横切的分片（称为用例模块）实现这种分离的。以后我们可以重组或编织这些分片，来回归到一个一致的整体——已部署的系统。AOP是基于与可扩展机制类似的概念，对其中部分机制的支持现在已经在UML中通过用例扩展实现了。这样，幸亏有了AOP，这些概念现在终于可以开花结果了。

所以，如果这篇文章专注于扩展机制的产生而与基础无关，可以使用下面的标题：

UML的扩展机制 ≈ AOP的Aspect

如果我们想描述一个更广泛的技术，用来跨越整个生命周期，也可以叫

AOSD 和 Use Cases

（AOSD，Aspect-oriented software development，面向方面的软件开发）

但我们的动机是在用例中加入方面，从而能够逐个用例地切分系统，并适用于所有相关的软件生命周期模型。

在爱立信，我们成功地设计了一个系统，可以不断修改很多年，并能继续使用。

通过定义良好的接口，它完全由互连的组件组成。通过用组件配置系统，我们满足了客户的需求。通过加入新的组件和修改部分原有的组件，我们提供新特性或用例。我们所设计的一个电信交换系统，被公认为优于其它竞争产品。大家对此都很满意。那时是1978年。

但是，由于下面的原因我并不满意：

1.组件通常不仅包含实现主用例[1]的代码，而且包括实现许多其它用例的片断（通常是小片断）代码。

2.用例通常通过分配在若干互连组件中的代码实现。

这两种分解效应现在通常被称为“纠缠（tangling）”和“散乱（scattering）”（参见Peri Tarr等，[1]）。这两个著名效应在1967年被人用来作为攻击组件技术的炮弹。幸运的是，由于众所周知的原因，组件技术还是赢了。然而，我们还是不得不接受和面对纠缠和散乱。每次我们修改一个用例或加入一个新用例时，都要修改好几个组件。同样，每次我们升级底层系统软件时（例如为使粒度更细而升级恢复机制，或加入日志功能），都不得不对很多组件做细微的修改。

这篇文章描述了一种动手处理这一主要软件问题的方法。这种解决方法很多年前就已起步了，但我们自始至终未能明确它的实现方案。由于新的编程范型，AOP，“缺失的连接”开始显现并帮助我们“终止循环”。我们关于基于组件开发和面向对象编程的长期工作能够获得另一次飞跃。

我用术语“AOP”[2]，表述一种通用目的的技术，相对于特定的编程语言[2][3][4],你即将发现它是“天上掉馅饼”。这种通用的AOP是通用软件工程技术AOSD的一部分[5]。并且，（抱歉引入更多的术语）为了完善这幅图景，“用例驱动开发”作为AOSD的一种方法——不是很传统的方法——正在被精练中。但在介绍更多的东西之前，让我们先更好地理解这个问题。

2 问题的提出

1978年为了替我心中的疑问找到实际的量化证据，我为一个由成百个子系统组成的电信交换系统主持了一个小型案例研究。不同的客户（当时通常是一个一个国家）可以重用多数的子系统，但是通常每个客户都有自己的一套通信协议。所以，我们为每个协议设计了客户相关性子系统，实际上是两个子系统：一个用于向交换机拨入电话，另一个用于从交换机接出电话。在研究中，我选择了一个用于从交换机接出电话的子系统，发现如下事实：

◇子系统的基本功能实现了用例“用协议X拨打电话”的一部分。该用例是由许多子系统实现的，但协议相关的部分是由我所研究的这个子系统实现的。有趣的是，子系统只有40%的代码是用来实现这个基本功能的。

◇代码其它的部分用来编写实现其它23个用例的一小部分，例如，编写代码来阻塞使用此协议的电话线，编写代码来监管使用此协议的一组电话线的报警级别，编写代码来测量这些电话线上的流量，编写代码在软件错误发生时重启线路，以及编写代码支持子系统在多个计算节点上的分布。

◇大约80%的代码通过模板生成。组件设计者（程序员）只知道必须要用的模板，而不用深入理解它们的意图——没什么创造性的工作，至少是这样。

◇我们预计未来子系统中的新用例数目将持续增长。

◇报告被接受了，并引起了极大的兴趣，但反响却是“这就是软件的本质——用例跨越组件——无需什么改变”。我对此不敢苟同。

翻阅这个案例中其它23个用例的小片断，我做了一些笔记：

12个用例片断是非干扰性的：这些片断是非常简单地添加到子系统中的：它们的加入不会影响任何其它片断的行为；它们仅仅需要与其它片断共享的对象的读取权限。

8个用例片断是基础用例的扩展，但并不改变基础用例：这些片断需要有访问基础用例片断（拨打电话用例）中执行序列的权限。当执行基础用例并经过代码中的某个点时，需要调用这些用例片断。执行将总是返回调用点。

3个用例片断对基础用例有主要影响：这些片断需要有共享对象的写权限，而且会改变基础用例片断的执行序列。

解决方案很明显：

如果我们保持用例的独立性，即使它们会跨越一些组件，只要这种独立性在从需求到测试的所有生命期活动——经历分析、设计、实现、测试，当然包括运行时——都能保持，我们就能得到一个理解、改变和维护都变得极为简单的系统。

但是我们如何才能达到这个目的呢？在本文中我将分三个部分来解释：

Part I　　基本思想 介绍主要的解决方案

Part II今天——使用用例 简要陈述用例驱动开发的技术的现状，和由于我们现在不完善的用例分离能力所面对的问题——不支持方面

Part III　　明天——使用用例和方面 描绘明天我们如何使用用例，当用例能够贯穿生命周期而保持独立时——支持方面

3 PART I：基本思想

自始至终地保持用例独立

在爱立信时，我们的组件提供了组件模块性。组件在提供系统整体视图和描述系统静态结构方面工作得很好。通过一种类似于人类传统上组织复杂问题的方法——层次性分解，它们有助于我们理解，设计，实现，分布，测试，和配置系统。但是组件并不能帮助我们描绘静态系统在做什么——即它的功能行为，因此我们使用用例。但是问题是用例并不真是分离的：他们横切组件。我们试图将它们指定和设计为分离的单元，但是在实现用例的时候，它们都会被集成到一块，无法分辨哪个用例是被哪块代码实现的。或者可以这样说，**用例会溶解在代码中，而想把它们再提取出来，就决非易事了。**

我一直在寻找组件模块性之外的新模块性。这些新种类的模块应该跨越组件，能够与同种类的其它模块性组

合，提供系统完整的功能性行为。组合可以在各种层次上进行，在一个组件内，或在所有组件的整体之上。我寻找功能模块性，但那是在用例之前——今天，我更愿意称之为用例模块性[7]。用例模块性很重要，所以我将在“我们从方面中得到什么”一节中讨论。

为了实现这个梦想，我需要两种机制：

1．用例分离机制

用例的设计目的是将系统分割成多个与使用相关的部分。这对于同级的用例来说非常有效：没有哪个用例比其它用例更基本、更必需或者更可选。一个例子是拨打电话，这是电信系统中一个基本的用例。有很多通话的种类：本地通话、到另一个地区或另一个国家的通话、来自另一个地区或另一个国家的通话；这些不同的种类都是同级用例（peer use cases）。

但是，有些用例要依赖其它更基本的用例来工作。为了分离这些用例，我们需要一种扩展机制，能够从一个基础用例开始，然后用更多行为不断地扩展它，以开发（分析、设计、实现、集成和测试）复杂系统——其中地这些行为就称为非干扰（nonintrusive）扩展。首先描述基本行为，然后加入附加的行为，附加的行为不需要理解基本行为[3]。我们的目的是通过从一个基础开始组织设计和代码，在与基础无关的语句（即使对于其它行为非常重要）不添加到基础中的情况下，使系统生长，从而获得容易理解的设计和代码。

这种扩展机制应用于用例时，能给我们带来“扩展用例（extension use case）”。在一个包含用例的基础之上，我们添加扩展用例，当组合两个基础时，我们将得到一个新的基础，带有新的扩展用例，如此类推。通过这种扩展机制，我们可以保证大多数用例在转变为代码，甚至是可执行文件的过程中始终保持分离。

2．用例组合机制

为了使系统能工作，我们需要将多个分片（也就是分离的用例）组合或者集成为一个一致的整体，以获得可执行代码。我们有几种选择：例如在编译前、编译时或者执行时将它们编织起来。

将扩展用例与其基础组合起来，如果给定当时我使用的扩展机制，这是相对比较容易做到的。但是，我们还需要将不是通过扩展分离的同级用例组合起来。这里我们必须将有重叠行为的用例组合起来，比如，两个有操作类似但是不相同的同级用例。这是一个更复杂的问题，也是一个更一般性的问题，因为组合扩展用例只是组合同级用例的一种特殊情况。

在两种机制中，我优先考虑通过扩展机制分离用例的能力，因为通过它，我们能向下直到组件的代码层次

都保持用例的分离。组件开发人员就能将分离的用例集成起来。这种机制通过向开发环境添加一个预编译器就能非常简单地引入。而且，从我的经验来看，即使是这么小的技术改变也能获得巨大的回报：许多新特性只需要设计简单的扩展，这些扩展能用奇迹般更简单的方式测试。我们因此能节约更多成本。

所以，我的工作集中在用例分离机制和用例组合机制这两种扩展上了，而复杂得多的同级用例组合的研究工作就得留待将来了。

首先，最初的扩展

为了解释这一概念，我使用了下面的例子。斜体显示的文字直接引用自我1979年的论文[7]，做了无足轻重的修改[4]。

我们的……语言构造必须补充明确改变“控制流程”的能力。我们将用一个例子说明这一点：首先，是我们今天（指1979年）的做法，然后是未来可能的进展：

例子：假设我们有两个用例，“通话处理”和“流量记录”。你可以认为前者的存在依赖与后者，反之不然。但是，构造“通话处理”用例你必须也具有“流量记录”用例的知识。

结果可能像这样（非常简化）[5]：

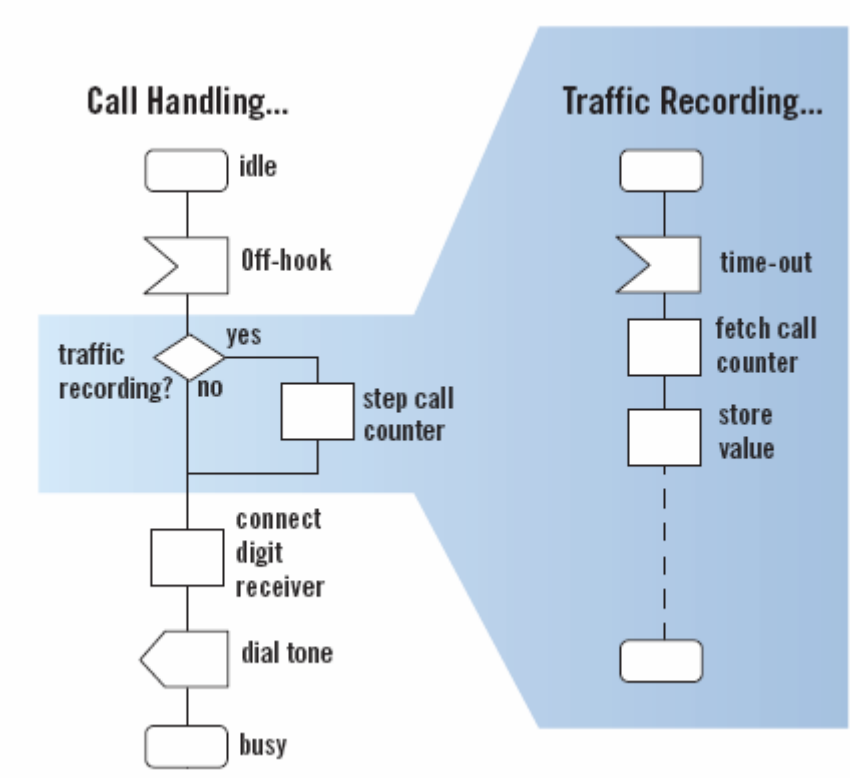


图1：“通话处理”用例无法忽视“流量记录”用例所请求的活动（比如“递增通话计数器”）[6]。

在“通话处理”用例中，我们必须包含与流量记录相关的用例片断。不幸的是，用例应该独立。

通过使用简单的技术……这种情况是可以避免的。

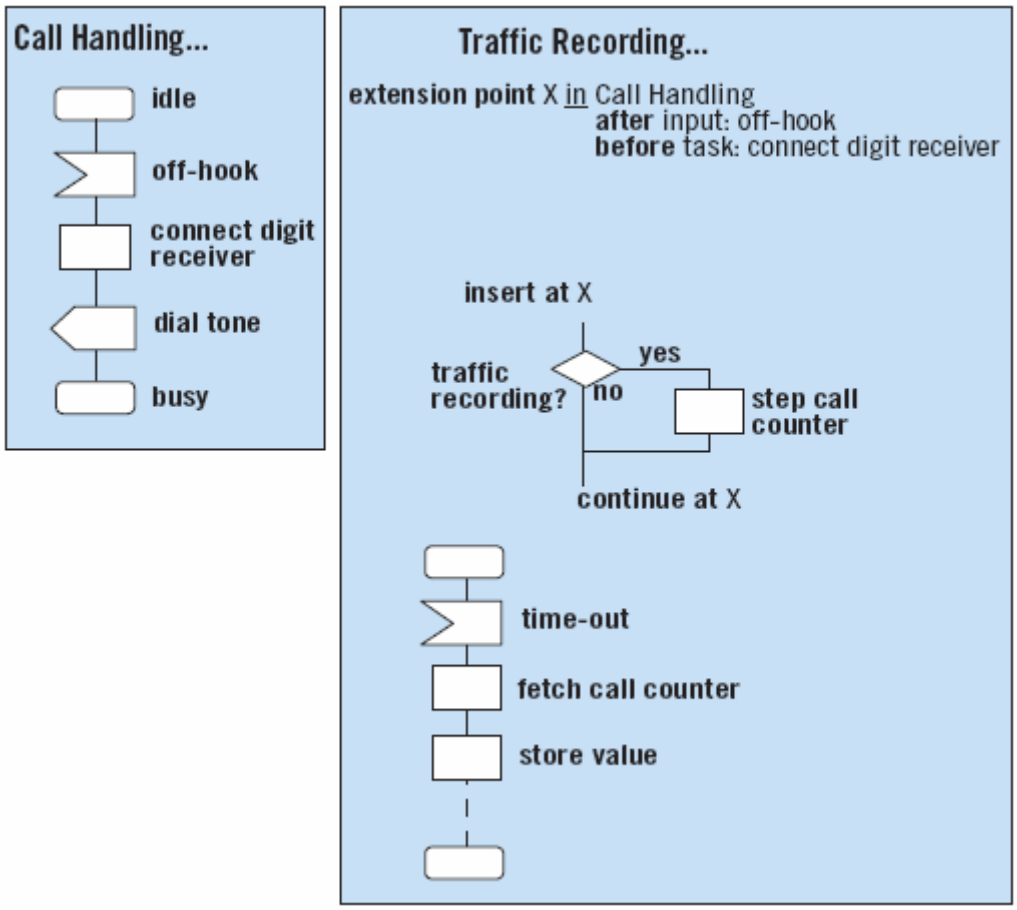


图2：通过一种简单的技术——用扩展和扩展点分离关注点

——“通话处理”用例能够不再理会“流量记录”用例了。

这样，我们就引入了一种可能性，能够没有二义性地引用另一个用例描述，并改变一个用例的控制流程。“通话处理”和“流量记录”现在可以用用例模块性来描述了。

下面来自同一篇论文[7]：

在用例描述中的一条语句执行之前（已经编译成目标代码），我们假设微编程会（同时）检查外层中的一个用例是否引用了当前的指令地址。如果是这样的话，当前用例描述的执行将被中断，而执行引用方用例描述所插入的序列。

我为这种概念的可微编程实现申请了专利[8]，它没有被批准，原因等一下解释。

因为一个大的扩展类可以安全地引入，无需干扰基础，对于这个类而言，回归测试在有正确的工具支持下就不再需要了。这将有望大大节约测试所需要的人力和成本。

为了实现这种机制，我引入了几个简单的构造：

◇可以用来无二义性地引用另一个用例描述（在描述中使用**before**或者**after**声明）的扩展点（extension point）。术语“描述（description）”的意思是一张图或者一段代码。

◇两个语句“insert at <extension point>”和“continue at <extension point>”，可以无需在基础中说明，就能扩展一张图或者一段代码。

在“用例模块性”一节[7]中，我写道：“用来实现这种概念的技术，从原理上将令人吃惊地简单：运行时编辑。”我认识到，这一提议正是一种新技术的发端，而此提议一旦得到接受，将会按自己的轨迹继续发展。

几年之后（1986年）我将“扩展”[9]这个观念概念化（见下面的斜体字）。抱歉我引用了1986年论文中冗长的部分，但它依然有效[7]。我引入了一个新词“存在（existion）”作为对象的原有集合（基础），并与扩展形成对比。

在“存在”和“扩展”之间只有两种关系：

○ *“扩展”要么无需访问“存在”的权限，要么只有读访问权限。*

○ *“扩展”需要有来自“存在”的控制权限（也就是说，扩展将使附加指令执行），但不能改变“存在”。*

前一种情况意味着扩展可以使用对象实例现有的操作，只要这些操作不改变对象实例的状态。

第二种状态意味着，虽然存在的执行是原子的，但是扩展可以在某些指定的点进行干预。当扩展执行后，控制将返回给存在让它继续执行原子操作[8]。在存在执行当中，可以有一个以上的扩展进行干预。扩展点指定了需要对存在的执行进行干预的位置。

我们必须提供这样的构造，才能在不改变存在的情况下描述扩展[9]……

这里的要点是提供一种扩展……通过一系列扩展点（图3）。扩展点指定了插入点……在转换路径的解释期间，……存在中的一个对象实例允许扩展需要的语句……进行干预。

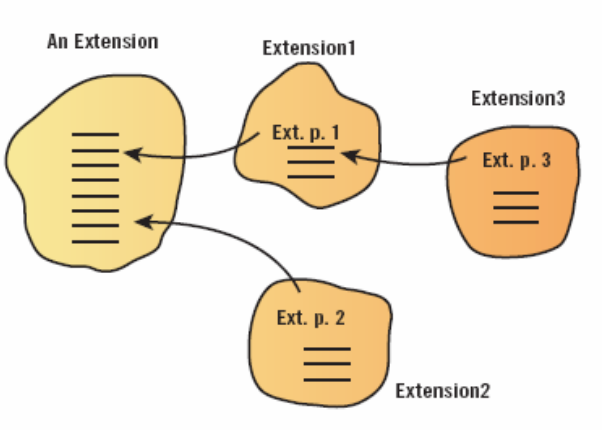


图3：功能扩展

扩展本身可以看成是一个存在，可以被其它扩展所干预。

因为功能扩展并不改变现有服务的行为，这种改变可以通过一部完成。

从语言上讲，扩展乍看起来可以视为继承其存在的新类。……

但是，类继承不是这里想要的行为。相反，扩展必须与其存在共存；它总是需要存在先行安装，只在存在执行的时候执行。

这是1986年。这之后又发生了什么呢？我理所当然对把用例模块加到原有的组件模块上的理念感到激动。我把组件结构视为我们添加用例模块的基础——一个一个的添加（在图4直观地呈现出了）这幅图中的每个小图更细致地展现了用例模型中的每个用例是如何被部分相互作用的用例所实现的。

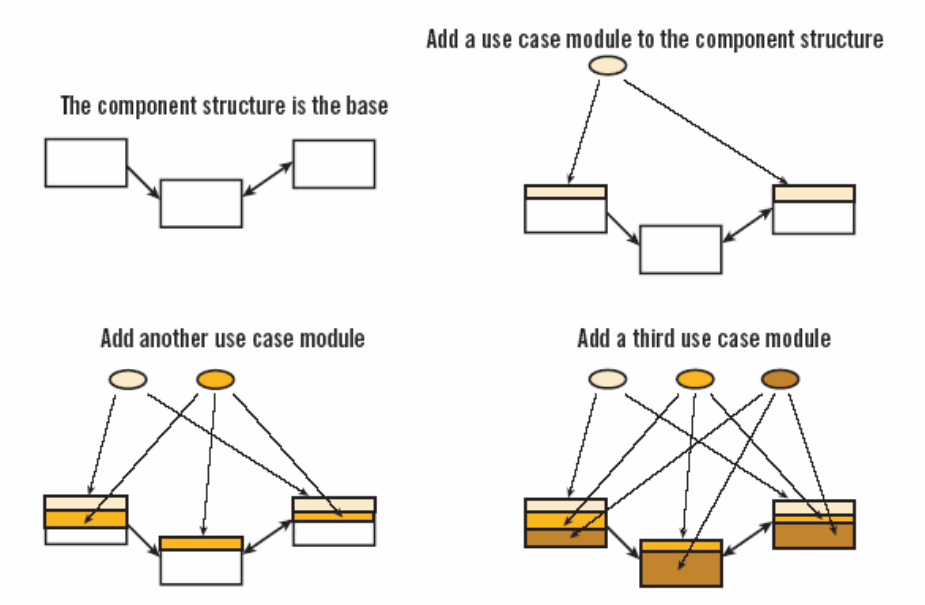


图4：在组件模块之上增加用例模块

不幸的是，这种概念与软件补丁技术太类似了——我一直不得不为这种相似性而抱歉。前面所说的专利申请没有批准，因为已经有一个软件补丁专利了，我的提议将侵犯它的权利。

在OOSE[6]方法中，我们继续支持在需求和分析中的扩展，我们说明了扩展怎样能够用传统的面向对象编程语言来实现。《重用》[10]这本书详细阐述了变化点是扩展点的一种泛化。许多这些想法都继续发展融入了“可重用资产规范”（Reusable Asset Specification，RAS，[11]）。

实现扩展的第一次认真的尝试，是在20世纪90年代初爱立信新一代交换机的开发中。扩展被加入到一个新的名为Delos的开发环境中，它能够自始至终直到代码阶段支持扩展。[10]

这是一种开创性的新概念吗？显然不是，它所依赖的是修补几十年来已经久为人知的软件补丁技术。但其中新颖之处是这样的概念，“正确地修补”是理解复杂系统最自然的方式。.

就像“天上掉馅饼”——Aspect-Oriented Programming

人类通常是从对具体事物的理解中进行抽象。而与此相反，将抽象概念转换为具体事物，则需要近似于宗教般的信仰。在使抽象概念变得别人可以相信之前，需要为此反复宣传。这种工作多年来我已经做过无数次。当我们的实现环境还是汇编程序的时候，将组件和接口具体化几乎是不可能的，但是我们做到了。现代的语言，比如Java和C#本身就已经将这些抽象（组件和接口）具体化了，因此我们没有必要再花功夫宣传了。将一种扩展机制具体化也曾是一种挑战，因为当时我们没有能为它提供支持的编程语言。即使现在UML已经通过用例建模提供了支持，许多专家还是直言不讳地反对使用它。但是，借助面向方面的程序设计，我们终于有了具体地解释扩展的工具，因为通过像AspectJ[2]一样的语言，可以在整个开发过程中自始至终直到代码支持扩展。即使它只是我们所需要的一部分，它也已经改变了很多。

用一句话概括：**为了在整个开发过程中自始至终（从需求直到可执行文件）分离然后再组合用例，我们必须能够处理同级用例（peer use case）和扩展用例（extension use case）。**扩展用例的概念已经提出20多年了，现在才通过AOP得到了支持。迄今为止开发人员都是手工管理行为互有重叠的同级用例。而我们将看到，AOP专门工作在关注点的多维分离（Multi-Dimensional Separation Of Concerns，MDSOC，[5]）上，而HyperJ[3]则允许同级用例分离。

我对AOP有充足的兴趣是自从在OOPSLA’97[12]上Gregor Kiczales的主旨演说上听说了这一概念。早先，在1991年，我经由Karl Lieberherr的大作获知DEMETER法则和传播模式[13]。在1994年我到约克镇拜访IBM时，会晤了Harold Ossher和Bill Harrison，并翻阅了他们的关于面向对象编程的书目[14]。

回到我的OOPSLA'86的论文，基础构造在AOP中都能找到对应物。表1中的映射是Karl Lieberherr在一次私人通信中原创的。

OOPSLA'86 paper	AOP Equivalent
existion	base program
extension	aspect
extensions doesn't change the base extension points	base oblivious of aspect join points
list of extension points	set of join points

表1：OOPSLA'86的论文到AOP等价物的映射

稍后我将详细解释。

4 PART II 今天——使用用例

为了理解 AOP 能做些什么，我们首先需要理解用例驱动开发方法当前的状态。**用例驱动开发也是模型驱动的。**它最简单的形式就是从用例到设计再到实现依次顺序进行。

在软件生命周期的每一次迭代中，开发小组要经历下面的一系列活动。

1. 找到用例并明确说明每一个
2. 设计每个用例
3. 设计、编码和测试每个组件
4. 测试每个用例。

以下，我将使用组件作为一般性的术语来指代类、子系统、服务（如 Web 服务）和物理组件等实现元素。

通常，这四种活动每个都表示由一位小组成员承担的一项工作除了活动 3（设计和实现每个组件）之外，所有活动都是基于用例的，因此产生了术语“用例驱动开发”。

当然，在一次开发迭代中，有更多工作要做，例如，架构的分析与设计，但是就我们的目的而言，不需要超出用例、用例的实现和组件之外。

用例

说得通俗一点，用例就是系统执行的一系列操作，能够产生对特定用户有价值的看得见的结果。

说得正式一点，用例是类似于类的一种构造，描述特定参与者或者用户对系统的一组相关联的使用。

用例可以是具体的也可以是抽象的。具体用例可以实例化。例如，假定“拨打电话”这个用例是一个抽象用例，那么“使用协议 X 拨打电话”就是一个具体用例。当用户拨打电话时，就创建了后一个用例的实例。

用例模型由参与者、用例和两者之间的关系组成，这是一种需求模型。为了使用例模型简单，要对用例之间可接受的关系类型上施加一个语言约束。其目的是分离关注点[1]：每个用例表示一组相关人员的一个关注点。这样，用例（作为类似于类的现象）之间的关系就是唯一可接受的关系了。

现在，让我们把注意力集中到两种关系上：泛化（generalization），即一个具体用例与一个抽象用例的关系；扩展（extension），即在不改变基础用例的情况下，在一组扩展点上给基础用例添加行为。所添加的行为在扩展用例中指定。在基础用例中，扩展点明确指的是用例描述中的一个点，可能使用 before 或者 after 限定符来指定。在这一点，扩展用例中指定的扩展行为将在解释用例时插入。

用例实现（use case realization）

用例模型是系统的一种外部视图——它不代表内部的构建块。系统的内部构造在设计模型中提出。用例模型中的每个用例都是通过设计模型中的一个用例实现来实现的。用例实现就是一个 UML 协作图描述（例如使用序列图）参与用例的组件，它们如何相互操作，以及在实现用例上都承担什么责任。因为每个用例在用例模型中都是不同的关注点，每个用例实现也是设计模型中不同的关注点。一个用例的实现往往涉及许多组件（散乱，scattering），而一个组件会含有多个用例实现的片断（纠缠，tangling）。

图 5 通过为一个三用例（不同关注点）的 ATM 系统建模，说明了散乱和纠缠的概念。每个用例都已经设计好了，其结果就是一个用例实现。最后，参与实现用例的每个组件都要设计、实现并进行单元测试。

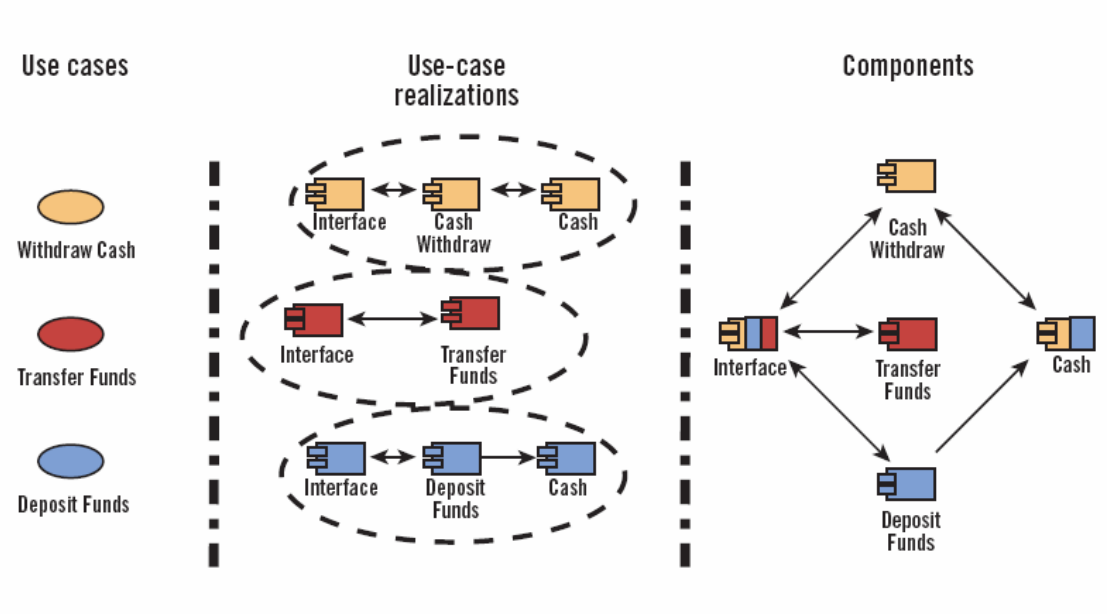


图 5：散乱和纠缠

散乱来自一个用例实现接触多个组件，而纠缠来自一个组件包括多个用例的混合片断。[10]

用例之间的泛化机制传承给了 UML 中的协作图。因此，用例实现可以重用更抽象的用例实现。

但是用例之间提供的扩展机制并没有传给协作图。对于这一点，在 AOP 之前我连例证都找不到，因为没有主流的程序设计语言支持扩展的实现。因此，当然还不可能在设计和实现中将扩展用例与基础用例分离。扩展用例的实现必须分散在基础用例的实现中，而基础用例也不能不考虑扩展用例。这样我们就无法完全无缝地从用例建模转换到设计——扩展用例的实现必须与基础用例的实现交织在一起。

组件

一种工具，能够通过收集所有用例指派给组件的责任，根据用例实现为每个组件生成规范。这很简单，因为责任直接来自组件参与实现的用例。

现在，组件的开发者必须在一个协调的整体中将所有责任组合和实现。问题就来了：协调用例的不同需求，例如，协调互相重叠的操作；解决冲突，例如由于并发引起的冲突（如死锁问题）。最后，每个组件还要根据其规范进行单元测试。

目前，主要的问题是，传统语言并不支持“关注点分离”，因此在一个组件上不同用例之间的影响无法分离——也就是说，无法维持各个分片。

5 PART III 明天——使用用例和方面

通过使用方面，我们的方式发生了本质性的改变。我们将能够自始至终保持用例的分离，直至可执行文件。但是，首先让我们重温几个关键构造，并研究它们与 AOP 的关系。

用例

与用例最直接对应的概念是横切关注点[15]（一种催生了方面概念的实际构造，但它不是一种语言构造），因为它将实现自己的多个组件分片了。与组件一样，用例可以按照一定的标准组织成层：特定于应用的（比如银行业务系统中的货币管理），应用通用的（可重用的银行业务框架），中间件（middleware）或者系统件（systemware）。我们通常认为用例是与应用相关的，属于应用相关的或者应用通用的层。但是，位置更低的层或者说是基础设施中的软件也可以含有用例。这种用例与分布性、持久存储、调试、性能监控、审计等等有关。

AOP 中还没有构造与用例对应——也不应该有，因为 AOP 只是一种程序设计技术。但是，AOSD，或者说得更具体一些，MDSOC[5]和 HyperJ[4]，都有对应于用例模块的构造。用例模块中包含一个用例实现和一组参与实现这个用例的组件分片。每个组件分片又包含组件实现的用例片断的设计和实现（即代码）。超分片（hyperslice）是 MDSOC 中对应于用例实现（即代码）的构造。

用例实现

一个用例的实现一般会分配到系统的许多组件中，在用例实现中建模，用例实现是一种 UML 中的协作图，可以直观地看成是与系统实现的一些组件横切的设计模型中某种模块化单元。在 AOP 的上下文中，

◇它既是基础程序（base program，就和同级用例是实现主要分解的基础一样）的一部分，

◇又是一个方面（就和用例扩展的实现中一样）。

方面是横切实现的模块化单元。而扩展，正如“可变的大型实时系统的语言支持（Language Support for Changeable Large Real Time Systems）”一文中所描述的，直接对应于方面。

UML 现在需要在用例实现（UML 协作图）中增加对扩展的支持。扩展点将从用例传递到流行的用例实现（UML 协作图）。

组件

而且，我们还必须在 UML 中的组件上增加对扩展的支持。这很容易接受，因为我们现在的编程环境中已经获得了有效的扩展支持。扩展点将从用例经过用例实现传递到参与实现用例的各种组件（类、子系统、物理组件等等）。

有些用例实现将需要多个扩展点，在不同组件中或者在同一个组件内。例如，扩展基础用例使其具有调试踪迹，可能意味着所有参与的组件都要扩展，因此需要在许多组件中都加入扩展点。

扩展点

扩展和扩展点在 AOP 中都有对等物：

1．扩展点在 AOP 中就是连接点（join point）。连接点就是程序执行中有明确定义的点。不是每个执行点都是合法的连接点，为了保证应用良好的编程实践，一些限制是必要的。

2．附加在一个用例实现上的一组扩展点及其参与的组件在 AOP 中就是一组连接点。它们是在连接点模型（join-point model）[16]中定义的，其中包括一个机制（例如，使用正则表达式）选择一组连接点。这是对用例驱动方法的主要贡献，也是我 1986 年论文中缺少的东西。

3．扩展行为（一个用例片断），在 AOP 中就是通知（advice）。通知是要在连接点模型中定义的每个连接点处执行的代码。通知可以以三种方式执行：before 连接点，after 连接点和 around 连接点。扩展行为/通知在扩展元素/连接点的上下文中执行。

为了迎接明天，我们必须在 UML 中添加至关重要的元素：其中最必须的构造，是通过扩展点将扩展关系从用例传递到协作、类、操作等等——事实上，要传递到 UML 每个重要的语言特性。

我们从方面中可以得到什么

新的一种模块（MDSOC 中超分片的特例）出现了：用例模块。正如前面讨论的，我们需要两种模块：用例模块和组件模块。用例模块和组件模块是横切关系。两种模块将共存，扮演不同的角色。组件模块提供了系统的静态结构；用例模块提供了在此结构上增加的动态行为。开发人员能够选择观察系统的视角——用例视角或组件视角，并由此开始工作。编程环境将把变化从一个视角传递到另一个视角。

具备了保持用例分离的能力之后，我们仍然可以执行“今天——使用用例”一节中讨论的活动：1．找到用例并明确说明每一个；2．设计每个用例；和 4．测试每个用例。但是活动 3．设计和实现每个组件，将被 3a．为每个用例编写代码，和 3b．组合每个用例的组件分片（参见图 3）代替。因为活动 2 和活动 3a 是同一个人——用例设计者完成的工作，生成的活动序列将变为：

1．找到用例，明确说明每一个

2．设计每个用例并为其编写代码

3．组合用例分片（一个组件活动）

4．测试每个用例

组件开发者的工作将有极大的改变：虽然不同编码或者测试组件（编码将由用例设计者完成，测试将与用例测试集成），他必须协调同级用例的不同分片。未来，我希望这种活动能够通过新的工具支持和相关用例设计者之间的协作而减少。编程环境将把每个组件中的横切用例编织为一个协调的整体，这个整体的性质取决于编织的时机——整体可能是一个预编译组件，一个可执行组件，或者一个正在执行的组件。而且，组件开发者必须负责组件“数据”（更低层次的类或者属性）的整体性。

这将减少或者可能消除组件的设计、实现和测试等不同活动之间的隔阂，从而使软件开发过程更加流畅，更有效率。**如果我们能够整个消除组件工作，4 个活动就只剩下两个：**

1．找到用例并详细说明每一个

2．设计、编码和测试每一个用例

这样，每次迭代将自始至终（逐个用例地）进行到代码层次（可能还包括动态的运行时）。现在我不知道我们能走到多远，但是我确信使用方面将用例切分到代码层次，可以充分地减少成本，简化开发。

6 未来

综上所述，很明显，我把AOP和AOSD视为对整个软件业界的非常重要的贡献。

能够自始至终地、直至代码阶段地支持扩展，并在稍后组合它们。这将在各个度量标准上给我们带来巨大的突破：成本，质量和工期。

沿着这一思路展望未来，一旦我们投身于这一新技术，这将是一次超越我们本身的更大的进步，我们将开始软件研发的新纪元。在这一新纪元我们可以说“元扩展性”，因为软件将具有天赋的可扩展性。

首先，正入我们所了解的，“系统研发是一个正变化的过程”[6]。十多年以来，我更加地明晰并直言“系统研发是一个改变的过程：从‘此物’变成‘彼物’;而研发的第一步不过是一个将‘无物’变成‘有物’的特殊情形”。未来依存于此种软件观。我试图用下列语句刻画未来：

◇**软件构建于扩展**[11]；甚至第一步构建都是一种扩展——对虚无的扩展。这样我们可以摈弃基础编程或existion等术语，相反，我们可以把它们视做一种扩展[12]。

◇**扩展将有几种类型**，例如：

○使用者要求的新商业需求或新特性所引起的新用例和用例修改

○对平台或基础设施修改

○体系结构的重构或其它改进

◇**系统研发将被规划为一个连续的扩展**；这将使系统更容易理解，成长，收缩和维护；成本，质量和工期标准将引人注目地改进。

○为了更精确，我把连续的扩展指代为在一个存在之上或之下。

○存在是自身地在原存在之上或之下的一个扩展。当它为空集时[13]，这一递归停止。我将在稍后回到这一点。

○“在其之上”指代增加高层特征的扩展，“在其之下”指代增加低层特征的扩展。

◇**扩展将在某一时点被组合**，也许会晚到运行时，来提供被系统整合的行为。

◇**扩展发生在整个系统的生命周期**；跨越所有的版本、迭代和构造。

◇**扩展必须在对已部署的系统无任何破坏性的操作下实现。**

◇**如上所概述的基于扩展的软件，其语义通过代码一直可以延续到运行时环境。**

实际上，我认为编程语言应当定义修改的概念，而不是像现在这样仅把软件的基本使用权留给操作系统买主处理。在我的论文中[17]，我定义了一个天生就基于扩展的小型演示语言。从语义学角度讲,一个系统实例建造时为空（空集），然后用用例实现（参与其中的类）扩展。当系统实例的用例实现被实例化，且这些类的实例被创建时，它进入执行模式。这是一个很有趣的话题，但它超越了本文所研讨的目的。

7 总结陈词

用例驱动开发或者AOP都不是银弹，它们只是代表两种最佳实践。但是，我相信它们的结合将极大地改进我们开发软件的方式。就眼前的意义而言，我们能够在一些最有趣的生命周期模型中逐个用例地切分系统。并自始至终保持用例的分离直到编码阶段。在将来的某个时候，我们将能够重新将这些分片组合成一个协调的整体：一个执行系统。就长远的意义而言，我们将获得更多基于扩展机制的软件——扩展从需求阶段自始至终到编码阶段和运行时阶段；扩展发生在每一个软件层，比如应用层，中间层，系统层；扩展也可以透过这些层。

我们将获得基本上各个角度都更易使用的软件。我们将获得更好的软件（高质量），当然☺，我们也将获得更便宜和更快的软件。

现在是我们动手让这一切继续，让梦想成为现实的时候了。

鸣谢

非常感谢Pan-Wei Ng对本文的深入评审。我也想谢谢Karl Lieberherr和Harold Ossher,和我的同事 Mauro Assano，Kurt Bittner，Magnus Christerson，Gunnar Overgaard，Ian Spence和Dean Whampler，他们为本文的一个较早版本提出过建设性的反馈。 感谢Catherine Southwood编辑本文。

注释

1 在早些年我使用术语“功能”来近似地表述“用例”的含义，但是，为简单起见，在这篇文章中，我用“用例”代替“功能”。我第一次引入“用例”一词是在1986年。

2 不幸的是，没有单一的AOP或AOSD的参考就像没有单一的OOP（object-oriented programming，面向方面编程）或CBD（component-based development，基于组件的开发）的参考一样。这些术语是一类思想的保护伞。比如AspectJ和HyperJ是AOP语言，但MDSOC和“用例驱动开发”是AOSD的方法。

3 这种分离机制不仅是一种描述可选行为（对客户视角而言可选）的技巧，也是一种计划用结构化方法描述强制行为的技巧。

4 在1979年论文中，我用术语“用例（use case）”替代了“功能（function）”，用“扩展点（extension point）”替换了“参考点（reference point）”。括号中是一些简单的解释，而不相关的文字就用省略号代替了。

5 读者对电信系统可能不太熟悉。“通话处理”用例实例在通话发起人将电话从支钩上拿起来的时候调用：将有一个离钩信号从电话传给用例实例。用例实例检查是否请求了流量记录。如果是，就递增计数器。这个计数器在通话进行时递增，当通话终止时随即停止递增。下一个操作是“连接数字接收器”，“发送拨号音”给通话发起人，对于本例而言，无需考虑更多了。

另一个用例是“流量记录”，其目的是测量一段时间内（比如说15分钟）来自发起人的平均流量。为此，它有两个流程，图1中只显示了一个。首先，每10微秒它都要访问系统中的通话计数器集合。计算递增了的数目，然后用计数器的总数去除它——结果就是在10微秒的时间内流量的快照。记录这个数字和在15分钟内类似的其它数字。另一个流程将计算周期内的平均流量。

6 图1中所用符号是开发AXE产品时采用的。1976年这些符号融入了SDL，经无足轻重的修改现已成为UML中的一部分。

7 我把术语“探针（*probe*）”改成了术语“扩展点（*extension point*）”，这是无关紧要的修改。我还把一些无关紧要的文字用“……”代替。

8 扩展的执行伴随着被存在触发的原子操作。

9 存在无需关心扩展。

10 图4显示散乱和纠缠，但是，并未显示同级用例的重叠。未显示现金组件、现金取款组件和现金存款组件的重叠部分，或者这三个用例与界面组件的重叠部分。如果用例不重叠，我们的问题更容易解决。但是，重叠必须得到解决。

11 增加了一个新名词

12 从数学上讲，设Si为系统发布i，Ei为扩展i，则S0 = null; S1 = S0 + E1 = E1; S2 = S1 + E2 = E1 + E2; S3 = S2 + E3 = E1 + E2 + E3。即每个Si都是Ei的和或集合。因此一个基础程序或存在就是一个扩展。

13 已经回到了1978年，在主持完案例研究会之后，我总结发现电信系统中超过80%的软件可以设计成简单的扩展，而不是基础系统。在更多细节技术在AOP中讨论之后，这一数字还会变得大得多。

参考文献

[1] Peri Tarr, Harold Ossher, William Harrison, Stanley Sutton. “N Degrees of Separation: Multi-Dimensional Separation of Concerns”. *ICSE 1999 Conference Proceedings*. pp 107-119, 1999.

[2] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. “An overview of AspectJ”. *Proceedings of the European Conference on Object-Oriented Programming*, Budapest, Hungary, 18-22 June 2001.

[3] http://www.eclipse.org/aspectj/

[4] http://www.alphaworks.ibm.com/tech/hyperj

[5] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach”. *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[6] Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Overgaard. *Object-Oriented Software Engineering, A Use Case Driven Approach*. AddisonWesley, 1994.

[7] Ivar Jacobson. “Use Case Modularity”. *Ericsson internal document X/Tg 2618*, 1979-12-13.

[8] Ivar Jacobson (inventor) & Ericsson (applicant), Address Sequence Variator, 1981-09-21.

[9] Ivar Jacobson. “Language Support for Changeable Large Real Time Systems”. *Proceedings of OOPSLA’86*. pp 377-384, Sept 1986.

[10] Ivar Jacobson, Martin Griss and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley, 1997. (Section 6.7.3)

[11] http://www.rational.com/rda/ras/preview/index.htm

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. “Aspect-oriented programming”. In *ECOOP'97—Object- Oriented Programming*, 11th European Conference. LNCS 1241, pages 220- 242, 1997.

[13] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. “Adaptive object- oriented programming using graphbased customisation”. *Communications of the ACM*. pages 94-101, May 1994.

[14] William Harrison and Harold Ossher. “Subject-Oriented Programming (A Critique of Pure Objects)”. *Proceedings of OOPSLA’93*, pp 411-428, 1993. USE CASES AND ASPECTS – WORKING SEAMLESSLY TOGETHER **28** JOURNAL OF OBJECT TECHNOLOGY VOL. 2, NO. 4

[15] Siobh´an Clarke and Robert J. Walker. “Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J.” *Technical Report TCDCS 200115* (Trinity College Dublin) and *UBCCSTR 200105* (University of British Columbia), 2001.

[16] M. Wand, G. Kiczales, and C. Dutchyn. “A semantics for advice and dynamic join points in aspect-oriented programming”. Work supported by the National Science Foundation under grant number CCR-9804115, January 2002.

[17] Ivar Jacobson. “Concepts for Modeling Large Real Time Systems”. Department of Computer Systems, The Royal Institute of Technology, Stockholm, Sept. 1985.

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

需求之于开发，就像婚姻之于人生

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

电信行业的软件开发过程案例研究

WHM Theunissen、DG Kourie 著，李腾 译

查看评论

概要

本文分析了对 Telkom SA 主要软件开发部门采用的软件开发方法的一项研究。这个部门的职责是为内部客户开发大部分的软件和硬件产品。考虑到它的活动范围，该部门的工作可以被认为在整个南非电信行业软件开发过程中所具有代表性的一个。调查的整体目标是确定这个行业当前所使用的开发方法同敏捷方法的兼容程度，以及了解电信行业是否可以从敏捷社区所流行的一些思想中获益。

I. 背景

为了确定敏捷方法在电信行业的适用性，我们对一家电信公司的软件开发过程进行了研究。该案例中的电信公司 Telkom SA Ltd.是目前南非唯一的固话运行商。作为第一个步骤，Telkom 中同软件相关的一些部门被确定出来。然后这些部门中最具影响力的被挑选出来以进行下一步的研究。这个部门被称之为 TDL，它进行内部的软硬件开发以满足电信方面的要求。选择这个部门的原因是：一、它参与了范围广泛的电信项目，这种广泛性同时体现在规模（成本和时间）和使用的技术上。二、这个部门所参与的项目的复杂程度超出其他部门。一些项目不仅需要软件开发，而且同时需要进行相关的硬件开发。

本文分析了对 TDL 软件开发过程的调研。第 2 章说明了调研中使用到的一些方法。第 3，4 章讨论了调研所得到的一些发现。调查中发现的问题以及可能的解决之道在第 5 章进行讨论。第 6 章提供了使 TDL 主要采用的 RUP 更为敏捷的一些指南性建议。第 7 章则做了一些结论性的说明。

II. 调研方法

调研分三个阶段进行。第一阶段是确定 Telkom 的软件开发部门。我们同来自不同部门的软件开发人员进行了接触，以确认他们是否对当前的 Telkom 的软件开发过程的研究有意义。根据反馈的意见，TDL 被认为在电信相关的软件开发中最具代表性的。而且，也没有必要在后面的调研中在多去找一个部门研究。

第 2 阶段包括了从 TDL 收集软件开发过程的信息。这主要是通过座谈和现场观察。开始的时候，我们同一位高级程序员进行了一次交谈。谈话的主要内容是对这个部门的开发过程做了一个整体的梳理性的描述。我们同时还观看了开发时所使用的一些不同的软件开发工具的使用过程。还有一部分时间我们花在了观察开发环境中产生的一些活动。观察的过程中，我们同一些部门成员也有一些非正式的讨论。这些信息被收集起来，变成一些将来会被提出和讨论的问题。一些准备好的问题在同高级程序员座谈时作为关联问题被提出来。这些问题和他们的回答我们将在第 4 章进行讨论。

调查的第三阶段集中分析了通过座谈和观察作得到的信息，并作出了一些结论。第三到 5 章讨论了最重要的一些发现。座谈和观察为 TDL 的开发过程提供了一个很好的视角。这个过程我们将在第 3 章描述。另外，第 4 章讨论了很多问题（它们被用以获取 TDL 的背景信息），以及这些问题在座谈时所得到一些回答。

最后，第 5 章讨论了 TDL 经常碰到的一个问题，并提出了一些可能性的建议。

III. TDL 软件开发过程

TDL 采用了一种改进的 RUP 来开发他们的软件。图 1 给出了 RUP 的一个框架。纵轴是一些称之为科目（Discipline）的东西。每个科目由一些逻辑上可以归为一类的活动构成。横轴为时间坐标。它展示了过程的生命周期如何在不同的迭代周期内展开，而这种迭代周期包括 4 个阶段：起始，细化，构建和交付。

RUP 是一个软件项目开发过程的框架。它集 Rational16 年的开发经验以及行业内其他企业经验之大成。RUP 包含 100 多种工件，40 多种角色和 9 个科目[Kruchen 2001], [Hirsch 2002]。因为它是个框架，因此实施时也需要从框架内选取合适的角色和工件。但内部的方法，角色和工件也可以被加入进这个框架。为了满足使用者本身的要求，通常也需要进行修改。例如业务建模阶段就被 TDL 取消了，因为 TDL 的大多数项目不要进行业务过程的调研。用户往往是带着已经明确的问题来找 TDL。这样就使 TDL 可以关注在起始阶段确认已经提出的需求。

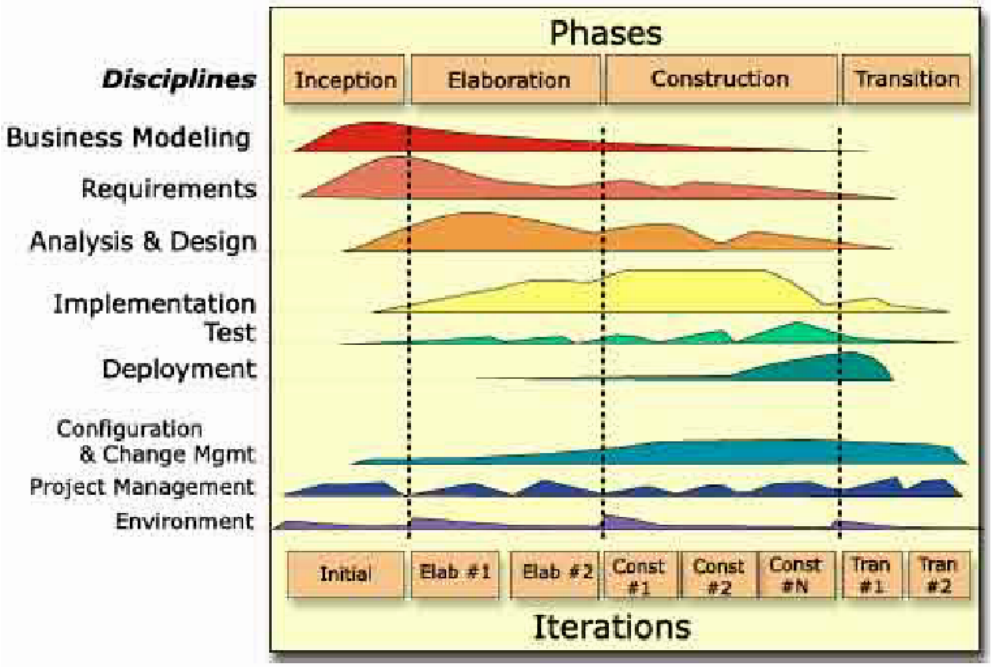


图 1 Rational Unified Process 的元素总览

项目的起始和细化阶段要花费大量的时间和精力，为的是确保一个高质量系统架构设计。普遍认为，起始和细化阶段良好的架构会对后面的系统扩展性起着促进作用。

开发团队遵从下面的过程：

一个团队首先从同客户交互和整理系统需求开始。通常这包括和全部项目相关人员举行一些设计会议（planning session）。同时，对于项目中会用到的一些技术会进行研究，包括网络协议，构成系统的硬件，以及第三方或者硬件厂商提供的类似相关解决方案的研究。

需求被建模，并整理成用例。当设计会议产生了足够的信息量时，这些信息就会被 Rational Rose 捕捉和建模。一旦需求被最终确定下来，文件就会提交给客户并签署，作为今后开发的基础。

在需求规范完成后，一个系统分析员的团队（2 到 3 个系统分析员组成）会开始进行系统分析。这个阶段一般结对进行，因为经验证明结对会提高建模的质量。用例被用来产生架构。依照 RUP 的理论，会花很大的力气去确认最有效的用例，尤其是那些进行起来会有比较大风险的。这些高风险的用例对架构起着最为重要的影响，在分析阶段也会首先去解决。作为原则之一，开发团队会首先实施高风险的用例，以尽早降低项目的风险。系统分析的成果是分析模型，包括高级的类图和序列图。

在 Rational Rose 的帮助之下，会从分析模型得到一个设计模型，它是对实施阶段所生成的代码的高级描述，可能会包括：详细的类图，序列图，状态图，这些都被 Rational Rose 所支持。作为实施的基础，如果适当的话，这一阶段会考虑尽量采用面向对象的设计模式。

在实施阶段的开始，系统的基础代码会由 Rational Rose 使用前面阶段所创建的模型来生成，主要使用的模型是设计模型。通常生成的代码是 java 或者 c++的，视项目的情况而定。然后这些骨架代码被填充起来，以完成需要的功能.在实施阶段，开发人员使用下面的方法：

- 编码时采用结对编程。经验证实结对编程可以产生质量更好的代码。
- 开发人员需要遵从 TDL 的编码约定。这个约定是基于 Rational 的指南编制的。
- 开发人员自己编写单元测试，这些测试被用来保证类功能的有效性。测试由下面会谈到的测试人员来进行。当他们认为需要时，测试人员可以自己对测试进行扩展。
- 使用 Rational Rose，开发人员被要求进行反向工程，以保证设计模型和代码的一致性。
- 当系统的任何一个部分可以进行测试时，测试阶段就开始了。主要的测试工作由测试小组进行，他们专门进行系统测试用例的编写和测试。大多数的测试通过 Rational Rose 提供的自动测试工具进行。包括，单元测试，集成测试，功能测试，用户界面测试，业务周期测试，性能测试，加载测试，配置测试以及安装测试。
- 单元测试主要是由开发人员编写的、对每个类进行的测试。
- 业务周期测试包括不同时间状态下对业务活动的模拟，在这种模式下，时间被压缩了，目标是确认系统在业务环境中的可用性。
- 性能测试通过 Rational Quantify 和 Purify 进行。这些测试被用来检查系统的内存泄漏，代码覆盖以及性能分析。

如同图 1 所示，RUP 是一个迭代的过程。也就是说上面描述的每一个阶段都是不断进行的。每个阶段都是在上一次构建的基础上以渐进的方式进行。

如前面所言，TDL 的整体战略中的一个重点就是开发一个具有良好扩展性的系统。TDL 认为这种战略可以使开发人员以很少的代价进行变更。经验说明，一旦有了良好的架构，开发人员就能够进行大多数的变更，而且对系统进行升级也会变得很容易。

IV. 探讨的问题

在本章中，我们列出了 17 个问题。这些问题用来帮助我们更好地理解 TDL 的开发过程.问题在会谈中被提交给具有代表性的 TDL 高级软件开发人员。在每个案例中，提供了对上述问题回答的一个总结。问题是关于一些普遍的软件开发的话题，包括：管理，过程和环境因素。而答案主要反映了在 TDL 开发是如何进行的。

A. TDL 总的组织结构是怎样的？

TDL 是“技术产品开发部”的一个部门，而后者也是“核心网络运行中心”的一个部门。TDL 被分作 6 个小组。小组一（ISS and Quality & Product Certification Ser-vices）负责测试，校核硬件度量工具，因此同软件开发没有什么关系。二组由被指派到每个项目上去的项目经理组成。剩下的四个组是开发团队，由一位小组经理管理，一个项目经理作为助手。具有代表性的一个小组有 11 个人，包括高级，初级的程序员和测试人员。图 2 是一个 TDL 的组织框架图。

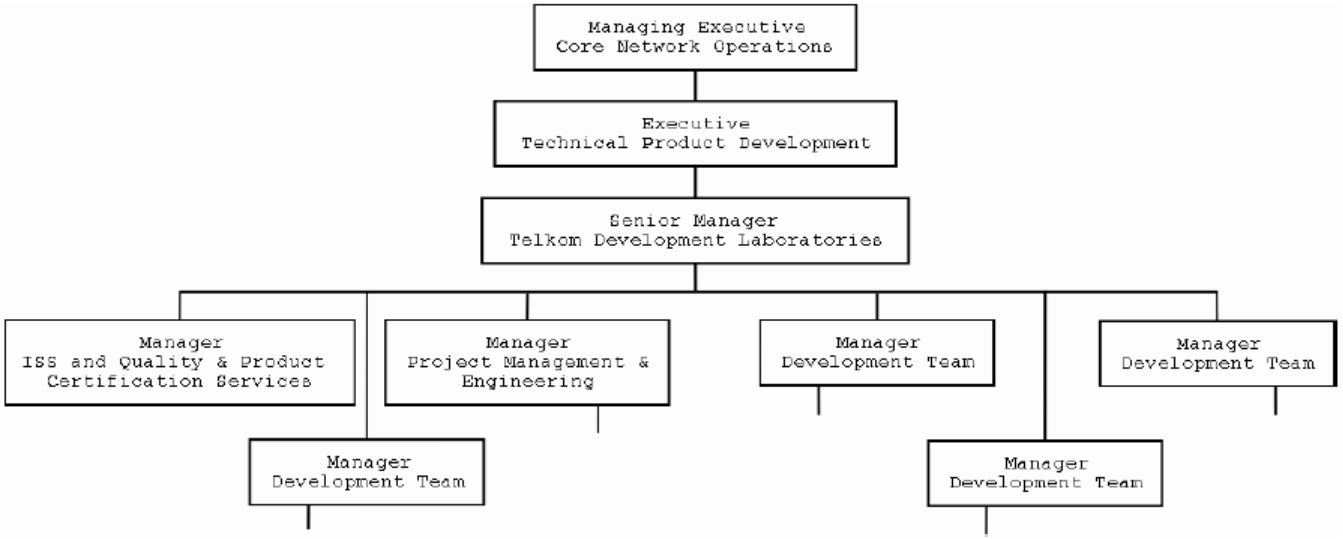


图 2　TDL 组织图

B. Telkom 和 TDL 主要承担什么类型、范围的软件开发项目？

这个项目各种各样，很难按某种标准去分类。但是，C/S 的应用是非常普遍的，项目也经常包括软硬件共同绑定到单一产品上的开发。还有一些多系统的集成项目。这些项目大多是内部使用，基于电信行业的标准。

C. 适用什么样的软件开发过程、或者准则？

Telkom 不同的开发部门使用不同的软件开发过程。一些部门使用配合“方案价值链”（一种基于瀑布式模型的开发方法）的定制过程，一些部门使用 ad-hoc，还有的使用快速应用开发方法（RAD）。TDL 部门使用的是经过改进的 RUP，见第 3 章所述。

D. TDL 必须符合一些什么样的标准？

TDL 的开发人员没有任何必须要符合的标准，仅仅是要遵从一些 RUP 的指导原则。

E. TDL 的项目成功率如何？

从技术和开发人员的角度来看，成功率是非常高的，几乎是 100%,这归功于 RUP 过程的实施。在问题域的早期调研上花费些时间以及定义一个良好的架构以使开发人员尽早确定项目的可行性。这样，一个项目（如果不可行的话）就可以在实施之前就被停止下来。

某些项目如果从商业角度来看也许是失败的，因为它们没有带来商业上的回报。产生这种问题的原因往往是非技术的，而且和政治或者市场因素更加相关。政治因素驱使用户和（或者）管理者们在系统交付后而不去使用。市场因素，例如供货商的政策拒绝使用定制的解决方案，可能会让用户去从硬件厂商那里再去购买解决方案。因此要说明的是，商业客户需要在同 TDL 接触之前，需要事先通过需求调研以确定自身的策略。管理层需要在把项目提交后，仍然坚持他们使用 TDL 解决方案的决策。

F. 项目评审是否为你们开发过程中的一个部分？

每个项目完成后都会有个项目总结。我们会对方法和技术的有效性进行评估。发生的变化会被考虑并且记录下来以供今后的项目使用。其中包括对方法的调整，以及工具上的配置变更，例如 ROSE 中表格和报告的调整。回答者还提到在迭代的过程中会缺少评审，对于单个项目而言，这可能会丢失一些有用的经验。

G. 目前的代码权限的策略是怎样的？

整个团队都可以访问到代码，如果需要可以进行修改（代码是集体所有的）。这些通过 ROSE 的版本控制工具来进行管理。团队间共享代码很少见，但是一些由同一个团队完成的不同项目中的代码共享是有的。

H. 硬件的开发环境有哪些特征？

每个团队在同一层楼的同一个办公区。每个成员有自己的隔间或办公室。测试人员被安排在一起，以便他们可以更密切地合作。不同的团队都在同栋楼里。

I. 白板在怎样的一个范围内使用？

白板的使用一般局限在起始阶段。主要的规划和建模在白板上进行。然后信息就都交给 ROSE 去记录了。

J. 当项目需求分析完毕后或者已经在实施阶段时是不是还会有变化？

确实在项目的实施阶段时需求会有变化（软件项目中这个现象很普遍），但是最多的还是用户界面的变化，而需求的其他部分还是稳定的。这种稳定要归功于通过同客户多次迭代而获取的一个比较全面的设计模型。

在很多项目中，大量用户界面的变更被认为是最主要的问题。这个问题以及可能的解决之道将会是第 5 章的主题。

K. 重用在 Telkom 得到的鼓励程度如何？它被提出的频率如何？

在 Telkom 并没有整体的原则或者正式的策略来推动重用。TDL 试图在它管理的团队和项目间推行重用。在团队水平上的代码重用的努力有着积极的效果。在有可能的情况下，项目间的代码重用也是这样。

L. 同客户的交互水平如何？

开发团队在开发过程中尽可能地使客户能够参与进来。我们鼓励高水平的交互和交流。这个目标是主动追求的。

M. 每个项目保存有哪些统计资料？

通过使用 ROSE，大量的数据被收集起来了。这些数据然后被 Rational 的工具转换成一系列的报表，包括：缺陷率，任务所花费的时间和测试结果。管理者们根据自己的判断来使用这些报表。

N. 典型的项目中 RUP 阶段的时间是如何划分的？

对于这个问题的回答是通过对他们认为是比较典型的项目进行讨论进行的。他们估计时间的划分大约是:前期设计和详细设计约占 50%，构建阶段为 40%，交付过程 10%。从这一点来看，在设计一个高效的架构方面是花费了很大的工作量的。

O. 有多少时间花费在新的方法，工具以及软件工程的创新方面？

团队的成员们会经常性地参加培训，目的是使团队掌握解决问题的新方法。这样做也使团队成员们可以跟得上行业里的潮流。

对于可能对现在或者将来的项目有用的新技术的研究同样也是鼓励的。在项目中新技术和新方法的使用会证实其可行性。在总结的时候，对于新技术的有效性会进行评估和讨论。

如果一项未知的新技术将在项目中采用，会在项目的开始阶段摸索。

P. 项目会经常迫使开发人员去熟悉新的领域和（或者）技术么？

大部分的项目会涉及到新的技术或者领域。这只不过反映了我们接触的内部用户涉及到的领域和技术是多样的。还有一个原因是这些项目经常是第三方不能完成的一些开发项目。从第三方购买解决方案的成本也可能是让 TDL 完成这些项目的原因，因为这些第三方公司的地理位置以及他们所谓的专业可能会导致惊人的价格。

Q. 有没有进行任何的开发成熟度评估（象 CMM 或者一些类似的标准）？

TDL 没有进行过正式的能力成熟度评估。部分原因是这个部门主要强调满足内部需求的开发，因此没有压力去进行此类的评估

V. TDL 的关键挑战

就像其他的软件开发队伍一样，同我们交流的开发团队认为他们也正在经受这个老问题,那就是必须满足不断变化的用户界面要求。就像很多包含用户界面的项目，工程师们总是面临这种问题。

导致这种问题出现的主要原因有 2 个。第一个是客户对系统的首要的印象和体会来自于用户界面。第二点在于人的性情和个性。一个人的主意每一天总会变化。昨天看到一个挺不错的方法今天可能就变得令人讨厌和用户不友好了。另外，每个人都有各自的喜好和厌恶，以及做事的方法，结果就会导致当不同的人来定义用户界面时，意见不一致就不可避免了。

人机界面理论规定用户界面的设计必须由专家进行，而不是用户。坚持这一观点意味着不能让来客户定义界面。虽然实际上不可能让客户对界面设计一言不发，但是上述观点还是应该尽量坚持。

处理变化的用户界面需求这一难题似乎是一个主要的绊脚石，同过程和方法论无关。一些过程试图通过尽可能地冻结需求说明来解决这个问题。主要做法是将用户界面的规范说明归档，要求客户在上面签字。因此，需求的变更要么是完全不允许的，要么客户会付出很大代价（罚款）而得到的。但是这种做法违背了“客户就是国王（Customer is the King）”这条业务规则。客户应当对系统保持满意，否则系统就不会投入使用，而项目也会被认为是失败的。这又让我们回到了第一个问题。怎样一方面让顾客满意，另一方面又可以限制用户界面需求的变化？

一个可能方法是在项目开始的时候确定一系列的“项目规则”。这些规则应当包含一个总的处理变更的程序，尤其是用户界面的变化。用户界面设计的变更应该清晰地被反映到成本和时间上。这种方法符合 ISO/IEC 12207: 1995“问题解决过程”的思想。这样可以帮助客户理解需求的变化会导致时间的延期以及增加项目的开销。

敏捷方法，象极限编程（XP）,Crystal 和 SCRUM 都会使用需求日志。使用这种方法，任何一项变更都会被当作系统新的需求，并且会加入到系统的下一个版本中去。在一次迭代的开始，客户会被允许选择哪些需求在下一轮的版本发布中必须实现。这种选择是基于功能的优先性并结合考虑迭代周期而确定的。现场客户也同样会使问题解决得更好，当这些现场客户本身就是人机界面专家时尤为如此。现场客户可以使开发队伍在进行界面开发时尽快地收到客户的反馈，这样同时也减少了进行变更的工作量。

VI. 使 RUP 更加敏捷

RUP 的一个实践就是它应当被量体裁衣，以符合项目的需要。通过这个实践，一个团队或者组织可以将 RUP 实施得更为敏捷。敏捷使团队可以获得快速响应变化的益处，可以提供不断增长的解决方案。

有一些文章提供了如何将 RUP 改进，以达到更为敏捷的指南。包括：[Kruchen 2001], [Hirsch 2002], [Pollice 2001]。值得一提的有：

工件的使用。产生的工件应当基于它们所能提供的价值而进行选择。控制工件的数量可以降低维护它们的工作量和花费。工件的详细程度也应当被考虑，尤其在 TDL 的案例是这样。

在规划时的详细程度应当尽可能低，以便接受变更。应该一次将详细设计的范围限定在一个迭代周期内。减少前期设计花费的过多时间，最大程度地利用迭代开发对 TDL 可能是有效的。

用户需要尽可能地参与，以确保一个稳定，快速的反馈环境。尽管 TDL 已经在尽力同客户协作，这个目标终归还是应当去主动追求并且尽量完善的。

尝试使用固定的迭代周期，每一次都包含一个可用的软件工件。固定的迭代周期，例如一个月，产生一个内

部或者外部的一个功能有限的但是可工作的软件产品。应该在迭代之间进行迭代评审（Iteration reviews），这些评审分析所使用的过程和实践。这种调查会产生一些新的想法，来微调流程，达到提高生产力的目标。

VII. 结论

通过调研，显示出敏捷方法所强调的一些技术和实践确实被同我们交流的团队正在使用着。这些技术包括结对编程和自动化测试。

但是，RUP 的基本原则要求进行大量的前期设计，这种面向设计的方法（也被称作“大前期设计”或者 BUD），是大多数传统的开发方法所要求或者鼓励的。但是，在敏捷社区，这种方法是不被鼓励的。它认为跟从一个设计没有响应变更来得重要。敏捷的观点认为开发过程中大量的系统建模会让开发者过于偏重设计，而不是系统真实的编码。

敏捷方法认为使用过多的建模来确定系统的可行性并非软件开发最有效的方法。相反的，它认为，系统的可行性可以在将风险较高的部分需求进行实施并且实际测试时就变得比较明显了。采用这种方法，客户可以在不用付出过多的努力和时间的情况下，就是否取消一个项目作出决定。例如，在一个采用 BUD 方法开发项目的情况下，一般需要 2～3 个月项目才能进展到确定其可行性的阶段。而一些敏捷方法，比如极限编程可能在第一或者第二次迭代试图实施关键的高风险功能的时候就发现项目的不可行因素，一般只要大约 4～6 个礼拜。

就 TDL 实施的项目类型来看，使用敏捷方法看来是可行的。但是，采用一种适当的方法需要考虑下列因素：

首先，项目的需求的变化的可能性有多大？项目需求从始至终保持稳定的情况下可能不会从敏捷方法中获益多少。

其次，可以将系统向用户以渐进的方式交付吗？可以将一个功能有限的系统尽早交付用户使用，而将后续扩展的版本在这一基础上不断完善提交。这种方法可能对客户会带来好处，而且可能对他们也很重要。敏捷方法的特征之一就是以迭代的方式提交可工作的系统。

再者，项目需要进行风险分析，采用这些方法可以减少风险。

最后，对于每个项目采用一个方法论的实践（如同 Cockburn 所推崇的）确实是有意义的方法。考虑一个项目，“连同它的限制条件和环境，以促使你选择最有效的解决方法”，而不要让解决方法来驱动项目！

今后作者会研究他们的建议在 TDL 的实施情况。

参考文献

[Cockburn 2002] Cockburn A, Agile Software Development, Addison-Wesley, 2002

[Hirsch 2002] Hirsch M, Making RUP Agile, OOPSLA 2002 Practitioner Report, 2002.

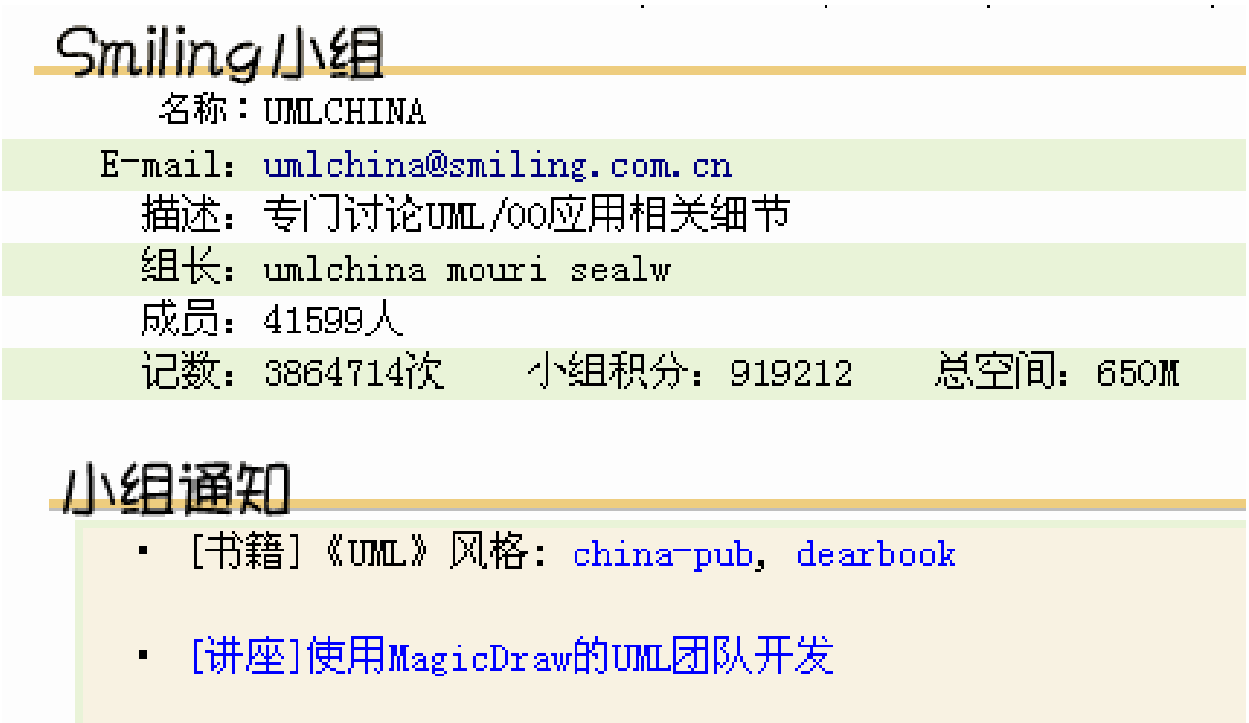
[Kruchen 2001] Kruchten P, Agility with the RUP, Cutter IT Journal, Vol. 14 No. 12, December 2001.

[ISO 12207:1995] National Committee TC 71.1 (Information technology), SABS ISO/IEC 12207:1995, Information technology -Software life cycle processes, 1995/08/01.

[Pollice 2001] Pollice G, Using the Rational Unified Process for Small Projects: Expanding Upon eXtreme Programming, Rational Software White paper, 2001

[Shneiderman 1997] Shneiderman B, Designing the User Interface, 3rd edition, Addison-Wesley, 1997.

[Manifesto URL] http://www.agilemanifesto.org



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

Scott Ambler

使用用例探索需求－－一次解决需求问题的对话

think 著

查看评论

本文根据录音整理改编。对话时间为2004年5月。隐去具体环境人物，尽量保持原汁原味。虽然不完美，甚至很乱很狼狈，但这样的东西是真实的、管用的。UMLChina把“聚焦最后一公里”作为一个目标，就是要抛弃“精心设计的案例”这种方式。

一些解释性的补充说明用补注的方式在括号中说明。

D－－开发人员

U－－UMLChina专家

D：这个用例文档你看着写得怎么样，给提提意见？

U：我看了。但是我想先不说这个用例文档，你先回答我一个问题：为什么要开发这个系统？

D：这是一个基于web的类似工作流的系统。现在企业里面的各种MIS应用越来越复杂，用户使用这些MIS系统的时候会出现各种问题，需要MIS部门的工程师解决。通过这样一个系统避免用户的请求没有人去负责，提高处理的效率，合理配置人员，对工作的情况有一个报告，有多少已经处理，有多少没处理等等。

U：好，我们来看看，你刚才说的就是系统的**愿景（Vision）**。这里要提醒你的地方是，你说的这几个愿景目标，能够有办法度量吗？比如你说“提高处理效率”，那么现在的效率情况如何，平均一个请求需要多长时间解决？上了这样一个系统，希望能缩短到多少才算系统没有白上？等等。这些客户或者你们老板心里有没有数？这个可以回去考虑一下。

（注：有关愿景的重要性，参见《有效用例模式》的SharedClearVision模式，Philippe Kruchten曾说，“如果仅允许设计一份文档、模型或工件来支撑项目，我会选择愿景文档”。）

U：这个系统涉及到哪些人，他们对系统有什么希望？

D：可能有这么几种人，第1种是用户，也就是使用MIS遇到问题的企业员工。这个系统主要就是为这些人服务的（注：是吗？难道没有这个系统，用户的问题就不解决了吗？），他们希望能通过一个友好的界面把请求提出来，提出来以后还能跟踪处理情况，觉得不满意的时候还能找到处理的人，让他把进度加快。第2种就是处理这些请求的MIS部门工程师，他们希望这些请求能够有分类地发到他们手里，例如负责处理email问题的工程师，不应该有打印机的问题发到他那里去。而且分配的任务是比较清楚的，要给我一个清楚的任务列表。我每天一上班打开机器，我的任务就出现在我的面前，优先级也很明确。这样，就能使自己更有效率地去处理用户的问题。第3种就是部门经理了，希望能通过系统掌握工作量，某个时间段的工作量，例如4月份，接到多少和IT有关的求助请求以及工程师对请求的响应情况。比如说提出了100条，谁完成了多少条，还有多少条没完成，完成这些东西消耗了工程师的多少时间，由此产生了多少成本，希望对部门的工作量有好的统计报告产生出来，让他能够通过这些统计分析来调整部门内部的工作流程或资源配置。给你的是和MIS工程师相关的用例，用例技术刚刚尝试使用，可能有些地方写得不是很规范……

U：规范不规范倒不要紧，关键是有没有写出有价值的东西，捕获了该捕获的信息。我们现在一个一个来看具体问题更好一点。比如2.6.2这个，“查看任务管理员发送的用户请求和其他信息”。任务管理员是谁，是系统上马以后设想的一个角色吗？

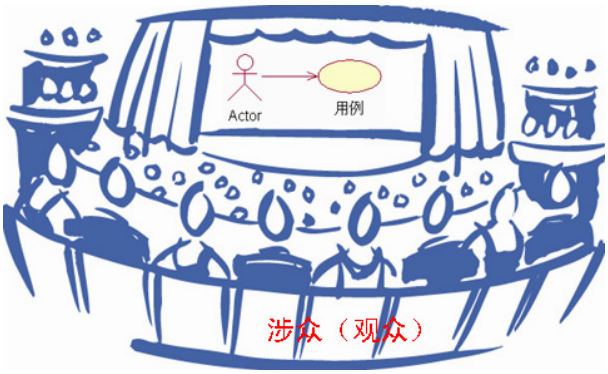
D：系统上马以后，工程师里面有分工。其中有1-2个工程师担任任务管理员的角色。这个角色怎么定义的呢？就是说一个用户提出一个请求，用户可能不知道他是属于哪一类的问题，email还是操作系统还是应用软件，提出问题以后，首先就到了任务管理员这个地方。任务管理员根据请求的性质转给相应的MIS工程师去处理。任务管理员就是起到一个类似接线员的作用。

U：好。你现在假设了一个流程，流程里所有的任务要通过任务管理员派发。也隐含了一个系统需求，任务管理员使用系统来派发任务。这里面就要和客户一起考证、一起思考了。为什么要有这个用例，没有这个用例妨碍不妨碍刚才提到的愿景目标：提高效率，透明度，明确责任。设置任务管理员这个执行者并给他一个用例“分配任务”是否合理？能不能系统自己智能分配，或者让用户自己挑工程师？这些都是可以讨论的。每提出一个用例，都需要思考它是否有存在的理由。这样才能防止需求的蔓延，计算机可以做这个，可以做那个，需求不知不觉中蔓延，这些需求是要钱的，而且无价值的“需求”和真正的需求混在一起，会影响真正需求的实现。（注：类似这样的事您一定也见过。一个人事系统，开发人员先花大量的时间去研究权限分配，RBAC什么的，最后没有时间去搞真正的业务，解决那些真正让职员头痛的问题，其实那个系统的用户很少，就算是用钥匙锁办公室的门也能实现权限管理。）如果没有经过考证，引进了一个用例，这个用例有可能还排在比较优先的级别，到最后才发现这个用例可有可无，这时很多时间都浪费掉了。写用例之前的第一步实际上是要论证这个用例存在的必要性。依据就是现实业务和系统的愿景。

D：这些问题要好好考虑一下……

U：现在先假设这个用例“查看任务管理员发送的用户请求和其他信息”的存在没有问题。我们来看看用例的内容。名称长一点，不过一开始长一点没事。（注：随着需求工作的进行，对系统的理解逐渐深刻，可能会有更好的名称）操作者（注：执行者，Actor）也可以，描述也无所谓。**前置条件**，这里应该是系统能够判断的一个状态，不能判断的前置后置条件是没有意义的。例如，这里的“任务管理员将用户请求分解并发送给相关MIS工程师”这个是可以判断的。**后置条件**有问题，这里写“MIS工程师查看任务管理员发送的用户请求和其他信息”，后置条件是用例结束之后系统的状态。前置条件是说开始时怎么样，后置条件是说结束时怎么样，应该尽量用“已××××”这种描述状态的语句来表述。好，现在起点终点定了，那么中间的路怎么走呢，这不是MIS工程师一个人就能决定的。我现在说一个通俗的例子，取钱。周末我想到楼下超市去买东西，会先看看家里的抽屉有没有现金，如果有，就拿一些出来到楼下超市买东西，如果没有，我就会拿上卡，到楼下取款机去取钱，这里就存在问题，为什么同样是取钱，家里的抽屉没有找我要卡要密码（注：如果读者您是气管炎，则是例外），楼下的取款机找我要卡要密码？而且很可能抽屉里的钱就是我昨天在取款机那里取的啊！找取款机取钱的用例写得象抽屉或者找抽屉取钱的用例写得象取款机行不行？这里就涉及到用例的下一个内容：**涉众利益**。假设MIS工程师在“查看任务管理员发送的用户请求和其他信息”的时候，有哪些人会关心这个过程。或者说他搞砸了或者乱搞，谁会担心可能会损害他的直接利益？从这些利益出发，才能推导出下面的路径、步骤、补充约束。我们来用一下幻灯片里的“醉酒法”（注：指UMLChina训练用的一个幻灯片，http://www.umlchina.net/training/umlchina_2.pdf），要是MIS工程师喝得醉醺醺的在做这个事情，谁会担心自己的直接利益受损？为什么他会担心？

（注：关于用例，Ivar Jacobson的定义是：用例实例是在系统中执行的一系列动作，这些动作将生成特定执行者可见的**价值结果**。一个用例定义一组用例实例。Alistair Cockburn进一步发展了用例思想，正如《编写有效用例》开篇所言，用例是什么？用例是涉众之间就系统行为达成的**契约**。笔者尝试把它补充一下，作为用例的进阶定义：用例是涉众之间就系统行为达成的契约，以执行者为达成特定目标和系统交互的方式演绎。笔者认为：只有在这个层次上理解用例，才能发挥出用例的最大价值。有关用例的历史，不妨看看《程序员》2004年5月的《用例：十年风雨》）



D：我觉得部门经理会担心。

U：担心什么呢？

D：（沉吟）……我分配给他的任务，他做不了，就把它接受下来了。或者是他能做，却把它拒绝了。如果他做不了，就把它接受下来了，任务管理员就会认为这个任务已经有人负责了，就不会去管它了。而这个人可能接下来之后就不去做它。这会导致用户的请求迟迟得不到响应。用户不会去抱怨某个MIS工程师，他只会去抱怨MIS部门，“反应给你们部门这么久，我还是不能打印”，这种抱怨多了，老板对MIS部门的经理肯定会有意见，认为他不称职，可能会处分他或撤了他的职。

U：好。**经理——希望MIS工程师不要乱接受，确认能做才接受。**那么，MIS工程师呢？

D：他也**希望系统能帮他把把关，看看他能不能搞定**。

U：这里我插一句：涉众的利益我们不能假设，只能去问涉众自己。也许他希望接得越多越好，能不能搞定再说。这种情况也不是不可能啊，如果公司的规章是按接的活多少发钱的话。因为一个问题能不能解决，谁也说不准。比如打印机坏了，我说我行，那可能性也大约就99%，说绝对能搞定估计是没有的。要去调查，把这些真正的利益关系搞清楚。

D：还有用户，看到他喝得醉醺醺的，会担心自己的请求能不能得到及时解决。

U：从这些利益就可以推出一些系统的需求了。为满足部门经理的利益，系统可能需要提供一种帮助判断MIS工程师能否搞定一个任务的价值，当然，可能不是这个用例的内容，可能是“任务管理员→分配任务”这个用例里面的内容。例如：在“任务管理员→分配任务”里面，可能需要一个步骤，系统告知任务管理员以前谁解决过类似问题作为参考。

D：任务管理员希望责任要清楚，透明一点，因为他和用户接触，用户看到问题迟迟未解决，会责怪他。

U：针对这个利益，就有必要让用户及时知道任务已经被分配给谁，这样他就不会怪任务管理员。这就是新的需求，可能也是在“任务管理员→分配任务”里面加一个步骤，通知用户（主动的），或者保存分配的细节，让用户及时跟踪。这就是探索需求的思考方法。

D：这里我有一个问题，我这个用例的格式是从一本《软件需求》的书上摘下来的，这样格式会不会有问题。

U：格式无所谓。当然有一些形式上的原则，但这个问题好解决。难解决的问题在于，可能你的用例写得规规整整，但内容不对。用例值不值得存在，值不值得花钱去做它，根据是业务和愿景。格式本身只是代表了用例应该有哪些内容，通过充填这些内容强迫需求工程师去思考。

U：下面我们来看看一般流程（基本路径），第1句，任务管理员把任务分配给MIS工程师。实际上这一句不应该属于这个用例，用例就像一份合同，它声明了只要满足前置条件，按照里面的路径步骤走，就能到达后置条件。这几个字不应该属于这份“合同”。任务管理员→把任务分配给MIS工程师可能是另一份合同，里面也会有步骤。另外半句，系统提示MIS工程师。这个也可能是另外一个用例，这里面涉及到提示的方式，是MIS工程师登录以后提示还是每隔一段时间自动提示还是二者兼有。

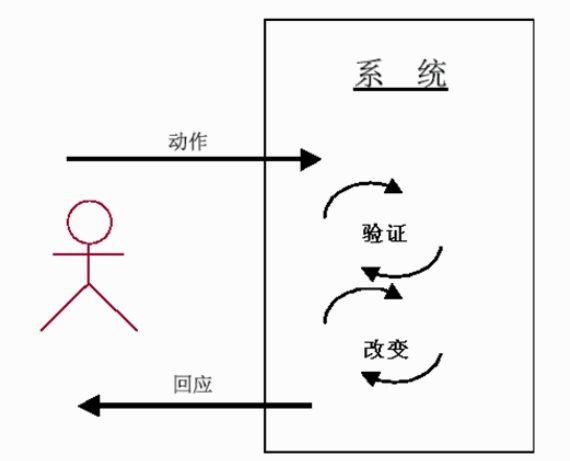
D：第一句是不是一定是执行者的动作？

U：是的。而且后面步骤的主语也只能是执行者或者系统。因为这份合同是通过执行者和系统交互的细节来演绎的。（补注：针对的是描述需求的系统用例。）

D：如果这里是自动产生提示，执行者应该是什么？

U：“时间——检查未查看任务”。（补注：时间，不是系统时钟，时间执行者在外面，系统时钟是实现时间引发用例的一种设计机制，如果用面向对象方法来设计，它是一个边界类对象的成分。）其实这个“系统提示有任务”的步骤可以出现在多个用例中。登录的时候提示，就可以出现在“MIS工程师→登录”用例中，在线的时候一分配就提示，就会出现在“任务管理员→分配任务”用例中。

U：我们看第二句，“MIS工程师查看任务列表，任务列表以明显的方式显示”，实际上这一句和用例的名字一样，但是步骤写的应该是为了达到用例目标，执行者和系统的动作。这些动作按照回合制书写：



所以要把这一步改成一个动作，“MIS工程师请求查看任务列表”，中间没有验证、改变，但需要系统的回应，“系统显示任务列表”，这样，就把责任区分开来，请求是MIS工程师的责任，不请求要怪MIS工程师，显示是系统的责任，请求了不显示要怪系统。下面的“以明显的方式”几个字，这几个字不象需求，因为这几个字是不可验证的。明显是相对不明显而言，这和个人体验、环境等很多因素有关。这里就有必要找MIS工程师问清楚，什么是明显。这可以看做一条涉众利益，但需要具体的需求来体现。

D：我想可能是用某种方式表示出来，比如，“某些字段用黑体显示”，这种是不是需求，怎么表示。

U：这可以看作绑定到这一步的一个约束，你甚至可以让涉众把他想要的“明显的”方式描述清楚，比如要声音？颜色？界面布局。可以让他把布局画出来作为一条约束绑定到“系统显示任务列表”这一句。但这两句需求要分开。前一句是功能需求（系统做什么），后一句是设计约束。稳定性不同，后者绑在前者之上，比前者易变。这里的关键是要和MIS工程师交谈，弄清楚明显的真正含义。刚才说的声音、字体、颜色什么的约束，也许并不一定是这样，也许MIS工程师的要求是某些显示字段，该出来的出来，不该出来的隐藏，这个就叫明显了。这种情况下，这条需求就属于“字段列表”补充到“系统显示任务列表”中。

（

注：用例的内容有以下这些：

用例编号：用例名

执行者

前置条件

后置条件

涉众利益

基本路径

1….××××

2……××××

扩展

2a.××××：

2a1….×××××

字段列表

业务规则

非功能需求

设计约束

）

D：我怎么区分设计和设计约束？例如刚才的“用黑体显示”。

U：需求的意思是如果系统做出来不满足它，某些涉众的正当权益就会受损害。如果涉众已经明确提出要用“黑体显示”，这是需求，因为系统如果不满足这一点，就违反了涉众的这个意思。如果涉众只是要求“某些字段要和其他字段在显示格式上区分开”，那么“用黑体显示”就是一种设计，由界面设计人员根据自己的知识和经验来为涉众选择的一种解决方案。

（补注：经常有人会问，使用了用例文档以后，怎么感觉那么细，这是不是设计呢。其实原因只是以前的需求做得不过关，漏了很多需求而已。设计指的是用什么平台、语言、架构来实现需求。涉众提出“用黑体”、“通过××协议与支撑系统通讯”，“购物满1000元获得VIP资格”，这是需求，不管用什么语言平台架构，都要满足这一点。还经常有人问，怎样和客户讨论设计？答案是你没有必要和客户讨论设计，你想要和客户讨论的多半也不是设计，只是做需求的时候没做好，一些利益、业务规则、概念关系没有搞清楚，拖到设计阶段出现问题了才不得不和客户交流。）

（补注：这里又引出我见到的另一种理解偏差，“反正需求是不能一次弄清楚的，这是为什么要迭代开发的原因。所以我们要敏捷，不用太认真去做需求（甚至分析设计），先编码再说”。这里我采用一个比喻：治病。要治好一种稍为复杂的病，医生不能也不应一开始就定好所有的治疗方案细节，而是要“拥抱变化”，根据上一个疗程的治疗效果来调整下一个疗程的治疗方案。虽然承认“这个病不能只用一个疗程就治好”的事实，但是医生在每一个疗程都需要根据现有情况尽心尽力。敏捷是积极的行动，不是偷懒的理由。）

U：下一句“MIS工程师点击任务列表”。这句话也有问题。暗示了MIS工程师用了某种“点击”的设备来工作，比如鼠标、触模屏。MIS工程师的目的可能是“选定某个任务”，不管他以任何方式选定，能满足他的要求就行。即使MIS工程师明确提出要用鼠标，也不能出现在这里，因为这是两类不同的需求。

D：这是不是意味着用例要写得尽可能抽象一些？

U：不，不是这个意思。我们经常看到有人讨论用例应该写得多细，用例的“粒度”应该多大的问题，实际上这些讨论是没有意义的。用例或者说需求不是面团，要怎么捏就怎么捏。这里面实际上就是四个字“实事求是”，用例是一种实事求是的技术，反映出“真正想要的是什么”。“选定某个任务”和“点击某个任务”之间，哪一个是需求，要去和MIS工程师交流，得到真实的答案。也许他们真的有需求，希望通过“点击”来锻炼手指呢？这个时候，“点击”就是需求。所以，不是“尽可能抽象”，也不是“尽可能具体”，而是“尽可能真实”，用例就是引导你一步步去探索真实需求的技术。

下一句“系统显示任务详细信息”，这一句可以，但紧接着的后半句“详细信息包括请求者…”可以分离到后面的字段列表栏目，一方面可以保持路径步骤的可读性，另一方面，“系统显示任务详细信息”及“该显示的详细信息有……”是稳定性不同的需求，前者比后者稳定。那么现在我们在路径里有四步：1. MIS工程师请求查看任务。2. 系统显示新到达任务。3. MIS工程师选择一个任务。4. 系统显示任务详细信息。这四步就完成了一个查看的目的。

D：那我应该是把查看作为一个用例来写，还是和接受任务合在一起写？

U：这里的根据是，光是查看，是否为MIS工程师提供了一个完整的价值。这就要和MIS工程师交流，问他：你认为光是这么看一看算是完成了一个事情吗？（就像你到取款机那里不是为了插卡输密码一样）或者是不是可以回答老板，“老板，我今天的成绩是看了100个任务…”。细节上加起来是一样的，例如到取款机取钱的用例，也可以在输完密码后砍断变成两个用例，里面也有路径步骤，问题在于我们和取款机交互的目的是为了取钱。

U：下面我们来看扩展。扩展就是步骤里出现的分支或失败，而且系统要处理这种意外或失败。第一个你写的是“超时”，从上面来看不属于这个用例的范围，没有哪一步能引申出这个扩展点。这里可能隐含着一个时间引发的用例。寻找扩展点要从步骤来看，看每一步有没有可能失败，而且这种失败是系统要处理的——因为这才是系统的需求，用例文档里写的都是需求——例如MIS工程师心脏病发作，停电等，是这个要开发的系统无法也不应该去处理的。第2步，“系统显示新到达任务”，这里可能就有一种例外情况，没有新到达任务。这时系统怎么处理或者干脆就结束用例。你看看那几步还有没有什么地方会有扩展？

D：好像没有了。

U：那好。现在我们的路径步骤写完了。接下来就要为这些路径步骤补上一些约束。包括字段列表、业务规则、非功能需求、设计约束。例如，“系统显示新到达任务”，该显示哪些信息？什么叫“新到达任务”？根据什么判断规则？有没有性能上的需求？你需要把这些调查清楚，补上去。

U：这个时候再回头看看涉众利益。MIS工程师希望系统能帮他把把关，帮他判断他能不能搞定。这个利益这个用例能体现吗？

D：系统可以把以前类似问题的解决方案列出来给MIS工程师看。

U：这就有必要添加一个步骤，“系统显示类似任务的解决方案”。还有用户，他希望MIS工程师不要搞错，信息要透明。我们看看这个用例里能不能体现。

D：“不要搞错”这个利益可以体现在刚刚增加的那个步骤上。

U：“透明”这个利益能不能体现呢。

D：可以。我们可以把谁查看了、什么时候查看的记录下来，让用户知道。

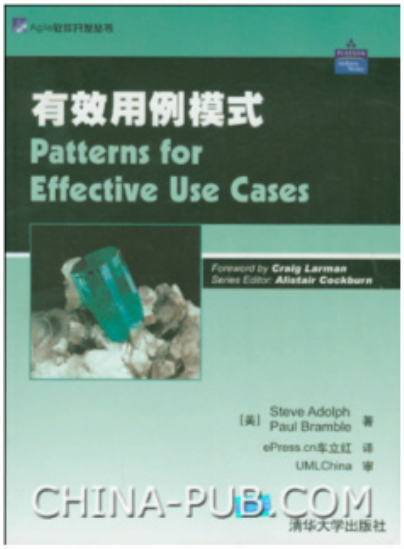
U：那么应该添加一条需求，作为步骤出现，“系统保存查看的细节”，外加一条字段列表的补充说明，“需保存的细节包括MIS工程师、时间、任务”。再看看部门经理，希望能接就接，不能接不要乱接。这个利益能体现吗？

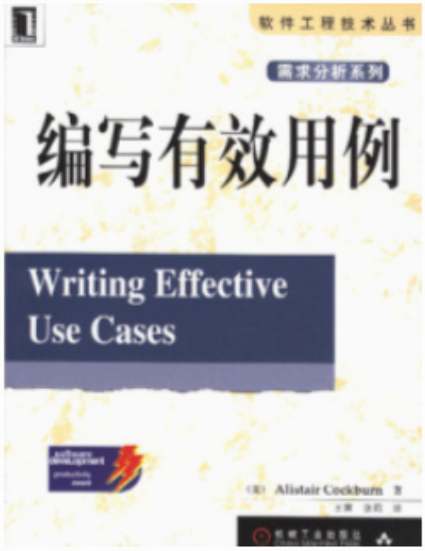
D：这个用例没有接任务的内容，不好体现，应该在“接受任务”里面体现。但是部门经理希望一个任务被分配给MIS工程师以后，MIS工程师能尽快查看并做出判断接还是不接。

U：这个用例第一步就已经是请求查看了，所以这个利益也不是体现在这里，但可能会体现在“时间→提醒有新任务没看”这个用例里面，例如提醒时间设得短一些，提醒方式强烈一些。但是要注意这又有可能和MIS工程师的利益相抵触，你不断地强烈提醒他，可能他正需要时间思考来解决一个难题，这实际上对他形成了干扰。这就需要MIS工程师和部门经理之间有一个妥协，要去问清楚，这是需求工程师的工作。怎么妥协，包身工式的公司和人性化管理的公司是不一样的。总之，一切需求的来源是涉众，这个项目涉及到的人，包括用户，也包括我们开发人员自己。

D：谢谢你的讲解。有没有什么好书可以推荐？

U：温伯格的《探索需求》马上就要出版了，你可以买来看看，里面有很多有用的技术。





D：我现在看一本陆丽娜翻译的《软件需求》。

U：我知道，机械工业那本。

D：对，我那个用例模板也是从那本书上来的，我看那书上和你说的好像有些地方不太一样。

U：没有本质上的不同，思想都是一致的。目前来说，用例是帮助贯彻这些思想的一个比较好的利器。至于模板什么的无所谓的，只要你能把该捕获的需求写出来，写在草纸上都可以。

D：书里面没有提到涉众利益。

U：我看看，比如说57页，第7章，寻找客户的需求，这里面的用户类、寻找用户代表，还有谁做出决策，不就是我们说的涉众吗，书里面翻译成风险承担者的就是。他可能没有写在模板上，但是你要知道这些是需求的来源，不能猫在办公室里“写用例”，编需求。而且也要让涉众来验证。69页那里也描述了我们刚才说的那些需求的类别。这本书没有强求你一定要使用用例这种方式来组织它们，但用例技术确实能【强迫】你去思考这些问题。

笔者为  首席专家。1999年创建UMLChina，潜心研究UML/UP相关技术的应用。这些年，笔者已经上门为50多家企业提供关于用例驱动的面向对象分析设计技术的指导。

The Addison-Wesley Signature Series

Patterns of Enterprise Application Architecture

中译本即将上市

Martin Fowler
With Contributions by
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee, and
Randy Stafford

UMLChina 训练教材

七步搞死 RUP

Craig Larman、Philippe Kruchten、Kurt Bittner 著，李涛 译

查看评论

摘要

RUP 为软件开发提供了一个非常有价值的框架。然而，RUP 并不是无所不适的过程，作为一个框架，它必须适应具体项目、具体团队及其环境的要求，也应符合轻量和敏捷的风格。本文描述了开发团队在具体环境中应用 RUP 时遇到的一些常见错误。

事实上 RUP[1][2]已经成为一个标准的现代软件开发过程。它结合了适应、迭代和风险驱动等最佳实践，由业界一流、兼具大小型系统开发经验的领军人物开发，它的应用和扩展十分灵活，文档和资料也非常丰富

然而，仍有一些因素限制着 RUP 的成功应用，并且导致了非常不尽人意的结果。这些失败的案例中蕴涵着一些共同之处，如果你想要在应用 RUP 时遭遇失败，不妨照其行事。

第一步：加入“瀑布”思想

你的开发过程类似于如下所述吗？

1. 尽量使需求确定下来，并签字认可。

2. 基于已确定的需求再进行详细的设计。

3. 基于已确定的详细设计再进行实现。

4. 集成，系统测试，部署。

这是一个线性的“瀑布”生命周期的例子，也是导致 RUP 失败的罪魁祸首。如果你的过程与其类似，那么你已经成功地背离了 RUP。

目前存在这样的争论：瀑布生命周期产生的许多严重的失败并不是由于其本身造成的，这些失败更应归咎于对瀑布生命周期错误的理解和不当的应用。即便如此，我们还是要探讨一下如今普遍应用或者说是误用的瀑布方法。

考虑到失败的标准，我们来看一下 1994 年关于项目成功和失败的一个分析[4]，当时大多数的开发是特别单一的，或者是基于瀑布过程的，该分析估计只有 70%的软件项目能够完成，超过 50%的项目预算翻倍，在美国，耗费在取消掉的项目上的金额就达 810 亿美元。这是个惊人的数字，情况依然糟糕。

值得赞扬的是，美国国防部，作为软件开发主要的客户，最初宣扬瀑布过程和思想，在看到如此多的失败后，不仅搁置了这种需求，而且在 1994 年的一份报告中开始积极地鼓励用现代迭代生命周期去取代瀑布生命周期[5]。然而，惯性却使得“瀑布思想”依然留存。“太好了，国防部将在一个项目中使用迭代的过程，现在是第一步：提交完整的需求，签字认可；然后再确定设计，……”

瀑布过程是对 1960 年代前的 ad hoc 方法的一种反应，比起之前的缺乏结构，瀑布过程的出现是合理的，此时的瀑布方法被称为“结构化”方法以突出其要旨。这种方法从我们所熟知的其它领域中的工程和构造方法中得到启发。例如建筑行业，分析－设计－建造。然而，人们采纳结构化方法的同时并没有深入地研究其是否真正适合软件开发，好几代人只是在墨守陈规。

有些东西应该像建造房屋那样去构建，然而，软件不是。

原因是多方面的。其中最根本的原因是：错误的假设大多数需求可以在项目的初期确定下来是。Copers Jones 的研究击碎了这一神话[6]。如表 1 所示，通过对 6700 个项目的研究，表明需求是不断增长的，随着项目规模的增大，需求的增幅也从 25%达到了 50%。Boehm 和 Papaccio 也得出了相似的结论，见参考文献[7]。

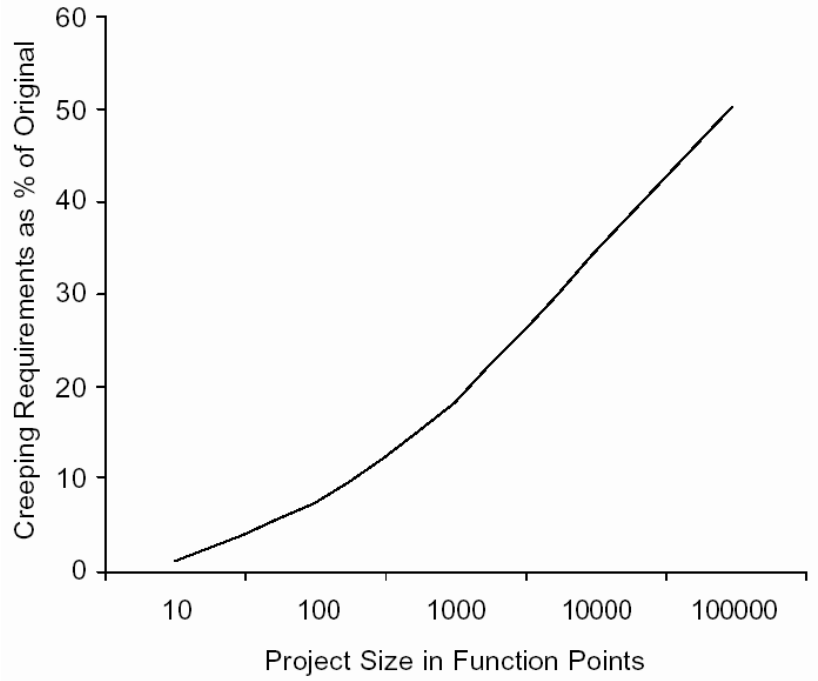


图 1 需求变化是正常的

瀑布思想坚持认为，通过足够的努力和技能，需求和设计可以被正确地定义和确定下来，也可以在项目中完全的、无二义的实现。

造成在软件开发中不能限制需求变化的原因有很多：

- 灵活性和可选择性的不平行
- 在心理和组织上限制了全面、精确和恰当地定义软件系统的能力（没有反馈-适应循环）
- 不准确的描述语言
- 有缺陷的设计和实现
- 市场的快速变化

无论什么原因，竭力将需求固定下来（像瀑布模型那样）都不是良策，而是要象 Kent Beck 所提倡的那样，“拥抱变化”，并将之作为过程中的核心驱动力。

因此，RUP 的开发过程建立在一系列迭代之上，每次迭代都有一个固定的时间限制（例如四个星期），称为“时间盒”，每次迭代结束的时候都发布一个稳定的小版本，该版本是最终系统的子集。“时间盒”是迭代开发中的关键概念：它意味着迭代周期的期限是固定的，如果目标没有完成，则放弃本次迭代的需求，而不是延长迭代的时间。

一个迭代周期有点像一个小瀑布。首先，选择需求的一部分（通常选择高风险的或核心的）进行充分的分析；接着，花一些时间进行设计；然后，开发团队迅速进行开发、集成、实现及测试。每次迭代都产生一个可运行的系统，该系统是将来需求和设计的反馈。随着时间的推移，一次次迭代和反馈揭示了恰当的需求和优良的设计实现。值得主意的是，这里以周为单位应用了瀑布方法，而不是以月或者年为单位，并且有内嵌的反馈和适应机制。以周为单位，一切正常；但是当迭代的周期规模变大时，将难以奏效。

由于现实中在设计和实现阶段需求是常常变化的，而瀑布模型将重要的风险推迟到后期去处理，常常一切已晚。瀑布方法与其所取代的先前的方法一样先天具有失败的倾向，只是瀑布方法较为有序（在惨遭失败前，项目看起来一切良好）。然而，由于不了解更好的方法，很多人认为自己别无选择。甚至整个组织也习惯于瀑布原则造成的失败和困难。不幸的是，直到今天，在我们知道瀑布方法的缺陷十多年之后，仍然有咨询公司、经理、教师及作家提倡瀑布模型。

因此，瀑布思想遍及我们的开发，RUP 实践者也常常在应用 RUP 的过程中卷入瀑布模型的价值观和实践。

瀑布模型的价值观和实践：

- 首先，确定大部分的需求，假设用户在看到其产品之前，就可以妥善的定义需求；
- 第二，进行详细的设计，假设对于一个定义糟糕的问题，可以为其找到一个精确定义的解决方案；
- 第三，在设计未经证实或者无法证实之前实现；
- 第四，集成、测试和部署。

瀑布模型的一些活动：

● 尽量完全地实现一些阶段产品，例如需求或者设计，然后再向前进行。尽量使这些阶段产品完备和稳定，并精益求精。

● 随后发现，先前我们竭力确定下来的需求、模块或者设计必须要进行重要的变化，这是失败的迹象。为了使需求或者设计接近于我们先前已经确定下来的结果，尽力弥补。

● 认为基于新的技术为新的系统交付可信的评估和计划是正确的，特别是在项目的早期。不能做到这一点就意味着缺乏技能。

我们很多人都被灌输了上面这些实践和观念。然而，这些实践和观念并没有被 RUP 和其它迭代过程（例如 XP）所采纳。丢弃这些实践和观念是由于其拒绝变化，不能包含变化并将变化作为过程的核心驱动力。

瀑布方法的误导增加了我们失败的风险(而不是降低了此风险),它违背了我们日常的教育和软件开发的常识。如同我们不能改变物理定律一样，我们也不能通过要求客户在规范说明书上签字来阻止客户改变需求。事实上，如果需求是错的或者业务变化了，我们需要一种灵活的方式来开发系统，这种方式允许我们逐步求精。在 RUP 和迭代开发中克服有意识的或者无意识的瀑布价值观念是项目组面对的最大挑战。真正采纳 RUP 最关键的地方不是在技术技能方面，例如用例分析、对象建模等等，也不是在于学习 RUP 提供的各种工件（artifact）和活动（这些工件和活动都是可选的），而是在于调整基本的态度和期望。

下面将描述瀑布价值观念和实践如何导致 RUP 应用的失败。

在 RUP 阶段中加入“瀑布”阶段

RUP 开发周期由四个阶段组成：初始、细化、构建和发布。导致 RUP 失败最常见的策略就是在一定程度上认为 RUP 阶段和瀑布模型的阶段相似。即，

1. 初始－分析确定大部分的需求

2. 细化－进行详细的设计和建模

3. 构建－实现

4. 发布－集成，系统测试，部署

上述描述是完全错误的，这是对 RUP 阶段常见的有意识或者无意识的误解。在各种各样的书籍、文章、演讲和咨询建议中不难发现这种误解。这种误解的标志就是把这些阶段名称用作动词（“初始做完了吗？”）或者状态（“所有的关键用例被细化了吗”）。

对 RUP 阶段细致的解释不在本文的范围之内，但是下面将给出关于 RUP 阶段的一个简洁和准确的描述：

1. 初始－开发系统的业务用例；要求探索少量但是重要的需求（大约 10%），以便获得范围、关键风险的尺度，并且决定是否进入细化阶段。

2. 细化－迭代地构建核心体系结构和解决技术风险。构建体系结构意味着真正的编程、集成及测试—这不是纸上谈兵或者丢弃原型。细化阶段，我们需要迭代地详细地探索大部分需求（大约 80%），同时实现系统的核心风险部分。在整个细化阶段需求都可能是变化的，通过不断的“反馈－适应”循环，评估已实现的部分。可以看到，这与传统的瀑布风格的需求定义不同，其大部分需求是在开发核心体系结构的同时细化得到的，并且其从实际的开发中得到反馈。我们也能够以此为据来决定是否继续此项目。

3. 构造－迭代地构建细化阶段没有做的元素；迭代地集成和进行质量保证；准备部署。由于大部分需求的不稳定性已经在细化阶段澄清，所以在构造阶段需求的变化较少。

4. 发布－完成 β 测试，确定版本，部署系统。

迭代周期太长或太短

有些组织在应用 RUP 的时候，迭代周期的平均长度为 6 个月，甚至 12 个月。在大多数情况下，这并不合适。RUP 规则推荐“迭代周期的长度是 2－6 周”。

迭代开发和 RUP 的本质是采取小步骤，对于可能不完美的实现，迅速集成，质量保证，测试，及时获得反馈，然后根据反馈，调整需求、设计和实现。小步骤、反馈和调整是核心概念。迭代周期过长与此背道而驰，必将导致 RUP 应用的失败。

另一方面，对于 300 人参加的项目，由于管理大团队等各方面的费用，2 周的迭代周期则难以运转。为了保证生产率，迭代周期应该是 8 周或者 10 周。但是可以看到，小的、暂时独立的子系统小组可以把大型项目 8 周的迭代周期划分成 2 周的迭代周期。

设计开始前求精需求

前文已经提到，在缺乏反馈的情况下定义和确定需求的想法是与软件开发事实背道而驰的。试想买衣服的情况，瀑布方法无异于让我们如此去做：在看到能够得到什么之前，就精确地描述想要什么；甚至不能先试试衣服看看效果怎样。在拿到衣服之前，必须准确的指定所有尺寸。如果想法改变，只有等待上帝出现。这就是瀑布方法的过程—在设计和实现之前定义大多数需求。

如果不能这样买衣服的话，怎能认为可以这样开发软件呢？

事实上，我们是一个矛盾体，是一个永恒的关于需要和想要的矛盾，何种选择是正确的常常是一个复杂的对比，需要我们试验不同的方法以辨别哪种最为有效。最可靠的做法通常是给出一些类似的方法，然后再从中进一步挑取。迭代方法允许这样直接的试验和创造；而瀑布方法却强迫我们在第一次的时候就足够明智。这与人类的本性，甚至是软件项目研究背道而驰。

在项目开始的时候指望可信的估计和详细的计划

想象一下在石油公司中的一个场景。一名经理把你叫入办公室对你说“我听说 ForBarKHan 有石油。我想要。请回去用一周的时间写一个报告，告诉我那里有多少石油，架设油田需要多少成本、花费多少时间，我们需要什么资源，以及一个详细的计划。”

上述在石油公司中是荒唐可笑的。但为什么这样的事情在软件工业中却被接受了呢？在石油行业中，大家知道不经过探测钻机的充分调查是无法得出上述问题的答案的。只有探明真实的储量和地质之后，才能做出估计和可行的计划。

然而，尽管和石油业有相似的不确定性和很高的复杂度，我们却指望能够不经过实际的调查就能估计和计划软件项目。

如果你之前没有做过某件事，那么很难清楚完成此事需要多长时间。有时这一事实成为了反对迭代方法的一个理由—如果你做的事情非常新奇，那么很难知道会用多少时间。由于害怕这些未知的因素，人们退却到了瀑布方法的温柔幻想中。我们可以制定计划来明确步骤、里程碑，但是并不意味着可以真正做到。

在缺乏工作量真实信息的情况下设立确切的计划日期是容易的—这是作计划遇到的典型问题：计划一经发布就没有用了。如果项目发布需要十全十美的计划，那么失败也就注定了。

迭代方法允许我们边学边走；随着迭代的进行，我们得到越来越多的真实的需求，更加客观的风险，以及完成该项目的更加准确的能力估计。简言之，经验使我们成为更好的计划者。

有时固定成本成为瀑布方法的辩护者，似乎这可以证明在缺乏真实信息的情况下做出计划是正确的。事实上，如果你从未做过此类项目，就被要求给出一个固定成本，你只能胡蒙乱撞。如果想成功，最好尽可能地多做预算，尽可能多地雇用此领域中有经验的人员，并期待最好的结果。在缺乏信息的情况下做出大部分的决定总是有风险的。在迭代方法中，最初的估计你不会做得比上面更好，但是不久你就会十分清楚你做得有多么好，你可以开始协商，预料，更早的管理范围，那时，最初的估计会有所帮助。

RUP 所提倡的解决固定成本及合同的合理方法是两步走策略。第一步，约定初始的迭代周期（例如四周）的固定成本，进行充分的调查和试验性的编码，以得到更具观察性的范围、风险和需求说明。这一改进的说明随后作为第二步中估价的基础（完成开发）。第一步调查的结果将会减少第二步的工作，并且有助于得到更加现实的和约。

第二步：将 RUP 作为重型的、预见性的过程去应用

方法学家认为过程有重型和轻型，预见性和适应性之分。重型过程是被人蔑视的，它具有如下特征：

- 刻板而控制严格
- 很多活动和工件官僚主义气息浓重
- 文档繁多
- 长期而详细的计划
- 过程高于本质核心的工作
- 面向过程而不是面向人；把人看成机械方法中的插件
- 预见而不是适应

预见性过程是指在一段相对较长的时间内试图详细地计划和预见活动以及资源的分配，例如大多数项目都是这样。其价值观倾向于瀑布模型，强调首先定义需求，然后详细的设计，最后实现。

相反，轻型或者敏捷过程意味着“倾斜和平均”，其特征如下：

- 抛弃不必要的过度的官僚主义过程，消除低价值的及无思想的文档；
- 集中在人本身，令软件开发充满乐趣；并且
- 是适应性的。

如果想在应用 RUP 时失败，不妨采取重型的、预见性的过程的实践：

- 详细地制定项目的整个迭代计划。在早期就定义所有迭代期限，并指定每次迭代结束时会发生什么。
- 创建大部分，或者是尽可能多的创建 RUP 的工件
- 添加大量项目和过程的形式，甚至几个项目委员会。
- 在项目中坚持机械和钟摆式的运作，将人看作项目系统中专用的齿轮。

RUP 的作者并不认为 RUP 是重型的或者预见性的，这种认为 RUP 是重型的或者预见性的错误观点是由于加入了不正确的过程思想或者对 RUP 的误解，RUP 本身提供的大量详细的过程文档也加剧了这种认识，以至于 RUP 被错误的刻画或者蹩脚的实现。RUP 作者的本意是用轻量、敏捷和适应性的过程精神来应用 RUP。如何这样做的一些指导如下：

- 创建尽可能少的 RUP 活动和工件；仅仅是那些真正有价值的。随着项目的进行，如果任何过程活动没有真正价值，剔除它。

- 不存在所有迭代的详细计划。有一个高层的计划（称为阶段计划）估计项目的结束时间和主要的里程碑，但是对于这些里程碑，它没有详细到确切的方法。详细计划（称为迭代计划）仅仅预先制定了一个迭代周期的详细计划。详细计划的制定是适应性的，从一个迭代周期到另一个迭代周期。这并不暗示不能为将来的迭代周期推测地分配一些工作，但这样做是侥幸的，它增加了项目在将来的不可靠性。调整最初的计划并不是制定计划或者执行的失败，而是对于项目现实情况的合理调整。高层的里程碑日期和目标是已知的，但是实现各个里程碑的方法是适应性的发展的。

第三步：避开对象技术能力

虽然 RUP 的大多数实践可以应用于其它技术，但是 RUP 针对的是面向对象的系统的开发。一些年来，我们看到了（直接或者间接）对象技术的项目失败和面临的严重问题，通常的威胁是缺乏真正熟悉对象思想、对象设计、对象模式和面向对象编程的人员。如果我们没有熟练的对象技术开发人员，再多的过程也无法挽救项目。熟练的对象技术开发人员是极为重要的、关键性的成功因素，相比而言，采用 RUP（或者任何过程）是相对次之的因素。

如同 Grady Booch 所说，“人比过程重要的多”[10]。相类似，《面向对象项目求生法则》（中译本由清华大学出版社出版）的作者，Alistair Cockburn 的描述更为简洁“过程的影响是第二位的。”

因此，想要失败的话，忽略对开发人员高级对象技能的培养。更深层次的意义是，对象技术的培养不意味着一周的 Java 课程培训，接着一周面向对象的分析和设计的技术培训（对于大量深奥的对象技术原则，这是远远不够的），而是应当对软件工程师们进行如八周集中式的、教师为主导的培训，而后进行六个月以上的扩展，接下来是十二个月的专家封闭指导。

这不是宗教或者天才的蛊惑，而是公认：对象技术的开发技能是十分有价值的，成功的对象设计和编程需要受过良好培训和高级的开发者，而不是新手。

对于雇用或者培养多少熟练的对象技术人员缺乏实际的调查，或者心存侥幸，认为只需较少熟练的工程师就够了，这是保证失败的极好的办法。

最后，我们想要强调集中培训的需要以及对象设计和设计模式的技能，而不仅仅是对象编程。引用 Davis Parnas 的话[12]:

代替教给人们 OO 是一种设计风格，而是给予他们设计的原则，人们会认为 OO 是使用特定工具。我们可以用任何工具写出好的或者坏的程序。除非我们教给人们如何设计，否则语言无关紧要。结果是人们用这些语言做着差的设计，并且没有从中得到什么价值。

第四步：低估适应性迭代开发

RUP 的核心思想和关键实践是：

- 需求管理
- 迭代开发
- 变更控制
- 质量审查
- 以体系结构和组件为中心的设计
- ……

不了解迭代过程和 RUP 的人看不出上述列表内容的实质所在，认为每一条的重要程度都差不多。然而，相比较而言，迭代开发对于我们如何思考和实践软件开发的影响是最深的。我们应当如下看待上述列表：

- 需求管理
- 迭代开发
- 变更控制
- 质量审查
- 以体系结构和组件为中心的设计
- ……

如果一个组织想要抛弃瀑布价值观和实践，那么迭代开发就是一场革命。事实上，如果一个组织正在从瀑布方法转向 RUP，并且在很多关于如何思考软件开发的层面上没有经历沉痛的、或者是创伤式的改革，那么这很可能意味着他们没有真正领会迭代开发的本质，并确实采纳了迭代开发。

有些做法忽视了迭代开发的内涵，如反复引入瀑布态度，以及下面一些所强调的：

拥抱刻板的和预见性的态度

这意味着尽量去冻结需求和设计，而不是拥抱变化。尽力去计划所有将来的迭代，而不是适应性地去调整。

改变组织或者团队的过程是非常大的改变—有潜在的分裂，有潜在的鼓舞，但总的来说是一个挑战。这种改变没有捷径—你必须直接并且深思熟虑。改变会威胁到某些人，他们会抵制这些变化，即使这些变化从长期来看对他们是有益的。项目推陈出新的力量对比也会随着改变而变化。因此，改变必须要小心谨慎，深思熟虑。

瀑布模型也面临着变化的挑战。瀑布模型不考虑未控制的变化，这是非常极端的偏见。某种程度上，瀑布方法是极度反对变化的—一旦需求被确定，如再变化则相当的繁琐和困难，即使发现已经确定的需求是不正确的。瀑布方法好像在对我们说“你既然同意过这些需求，那么请安静的拿走我给你做的软件吧。”

为了过渡到迭代生命周期，必须在两个层次上包含变化：组织层和项目层。在组织层上，必须将包含变化作为做事情的新方法。在项目层上，必须允许反馈和持续学习，接受响应新信息。

误解迭代开发

如同 Andy Grove 观察到的，“完成的是被度量的。”如果你为了“冻结需求”、“冻结设计”等而度量和奖励员工，那么无论你怎样为各个阶段命名，你进行的都是瀑布生命周期。如果你拒绝变化，无论你如何辩解，你进行的都是瀑布方法。

因为瀑布生命周期看起来是稳定和确定的，比如：瀑布生命周期模型很可能会说“12 月 2 号到了，我们已经分析完了所有的需求。”所以项目经理们也似乎喜欢这种表面的确定。然而，问题是这种说法其实是一种错觉—你只是说你已经做完了，但是在某种程度上你知道这并不是真的—因为还没有实现任何东西，没有可靠的方法来确保你真的是做完了。这些只是通过测试你关于一些可以估计对错的东西的假定以及通过实现一些你知道将会怎样运转的东西来估计。

在项目中，我们都在玩游戏—我们作出假定，预测事情将会怎样，解决方案将会如何，我们对自己所说的装作非常确定。但是，事实上我们真的不知道究竟有多么肯定，我们虚张声势，幻想最好。结果常常会表明，我们行动于不完全的信息之上。

明智的项目经理会重视这一问题，为可以证实的假设建立检查点，履行活动以获得更有效的信息。对于此，迭代方法是完美的：允许收集信息改进计划、改变方向。有这样一句谚语“计划不可能原封不动地实施。”明智的项目经理知道其含义，不失时机地根据新的信息调整计划。

也许瀑布方法最大的失败正是其最普遍的做法：孤立于现实制定计划，并且不根据新的信息更新计划。更改计划被认为是预测的失败。但是，没有人可以预知未来，与其认为调整计划是预测的失败，不如将其看成是改进的机会。

不就迭代开发内涵培养涉众

客户和管理者已经适应了瀑布文化。客户希望他们能够在移交需求之后就万事大吉，直到最终版本发布。

管理者习惯性的认为他们能够在项目的第一天就制定出完美的计划，然后仅仅是去执行，可以通过最小限度的调查以及实质上没有意义的探索性编程和概念证明来得到可靠的估计。如果项目失败了，他们不会认为是糟糕的计划造成的，相反，他们会认为是计划没有被执行或者作计划和作估算的人缺乏技能而造成的。这种想法也自己败坏了自己的名声。

为了保证迭代开发的有效，客户必须参与。迭代开发的核心和本质就是基于反馈进行调整，而不是推测。如果客户没有兴趣，并且也不积极地参与确定他们究竟想要和需要什么样的系统，那么他们会自作自受。构造软件中最困难的事情在于构造正确的东西，使功能和可用性一致。开发者很少是领域专家，他们需要帮助以理解什么是解决问题所需要的，来构建正确的系统。瀑布方法剥夺了开发团队和客户之间意义重大的交互。

必须再次教导客户，以使他们明确应该在定义需要开发什么时起到积极的作用，必须让他们理解，在系统需要做什么和新技术之间存在相互的影响。也许他们要求某种“传统”系统，没有意识到无线和移动网络设备能够带来可视性。技术改变了问题的性质。互联网就是非常好的例子：它允许我们用不同的途径处理问题，但是这些途径对于完全使用传统商业过程的人来说常常是不明显的。

理想的方法是，以对需要解决的问题的理解为开始，进化式的探询问题（既是挑战，又是机会），开发者和客户/用户之间是合作伙伴关系。迭代方法允许这种适应性的反馈，适应性的反馈是这种开发风格的精华。

过度建模和创建很多低价值的工件

RUP 是解决很多不同问题的框架。然而，你不可能遇到所有这些问题。因此，你不可能在项目中使用 RUP 的全部。

另外，迭代开发的思想是要在只完成了部分需求和设计的时候，快速开始编程，这样做是为了得到实际的反馈，而不是去推测需求和设计。

因此，让 RUP 阵亡和迭代开发失败的有效方式就是创建所有的 RUP 工件，尽力细化和详述每一个工件的特征。开始编程前画出至少二十页 UML 图。

RUP 像是一个药橱或者药店，大部分人都不会去想进入药店之后，购买每一种药，然后全部服用。我们或许会生病（也许更糟）；但我们知道应该服用需要的，有时听从专家的指导。

有人说 RUP 太庞大了；这就如同说药店的药太多了。有很多药，人们需要而我们却没有，即使大多数人根本就不需要很多药（也许只是一些偶尔的疼痛）。真正的问题是人们需要更加清楚的知道：怎样弄清楚自己需要什么。

考虑使用多少 RUP 内容的一个好方法是以风险缓解作为指导。

- 存在开发人员不理解系统将要做什么的风险吗？使用一些“需求管理。”
- 存在包含复杂控制流或者难以理解的行为吗？使用“用例。”
- 系统的技术方案是新颖和复杂的吗？利用“体系结构。”

你开始明白如何决定做何选择。理解你面对的问题，然后选择有助于简化这些问题的技术（药物）。

第五步：避开深谙迭代开发的顾问

没有什么比没有经验、误解和忽视更能造成新的实践失败的了。因此，对于你的第一个迭代的 RUP 项目，只用一个从未参与过适应性迭代开发的团队—特别是这样的项目领导人。寻找习惯于瀑布态度和实践的人员。一点题外话，为你的首个 Java 技术项目选择一个既不是 Java 专家，也不会 Java 编程的体系结构师。

干脆不要那些懂得如何迭代开发的专家顾问。如果一个知识丰富的顾问不幸加入了这个项目，要确保其不会带来什么影响，并且其建议会被争论、怀疑，最终被拒绝。其本人也受到嘲讽。如果必须要顾问加入这个项目组，那么就找一个不理解 RUP 和能够巧妙地引入瀑布预见性态度的人，例如，认为细化阶段是要阐明在构造阶段所要实现的模块，或者认为所有迭代的工作应该在项目开始的时候计划和分配。这样的人很容易找到。

第六步：轰轰烈烈地采用 RUP

如果可能的话，在某个星期一向整个 IT 部门引入 RUP，并且要求所有的项目到星期三为止都要按照 RUP 去做，但没有向任何人讲述任何细节。如果这样不可能的话，那么在组织中强制培训，只培训开发人员（而不是经理和高级执行者）。如果管理者是瀑布态度，而开发人员被培训要抛弃瀑布习惯，必将十分有趣。如果所有人员必须参加 RUP 的培训课程，尽量培训六个月以上，以使大家在采纳 RUP 之前完全忘记培训的内容，培训形式为速成式的一天之内的讨论，并且培训师有很深的瀑布思想。如果必须要在培训之后采用 RUP，那么培训至少要在整个组织中进行，并且同时让所有人、所有项目一起转变。不要在资深顾问的指导下用一个小的项目进行如下试验：使用最小限度的 RUP 实践，从经验中学习，逐步采纳 RUP，基于试验所得的经验前进到第二个项目。如果有人这样建议的话—让他走人。无论如何，都不解释采用 RUP 的原因—这样能够合理的证明代价是值得的，并且能够减轻变化的痛苦。找出 RUP 的怀疑者和瀑布的倡导者，让他们进入已经采用 RUP 的项目中去。

第七步：采纳错误的建议

即使在一些出书、写文章、培训及演讲的所谓 RUP 专家当中，也有一些并没有真正领会迭代开发及其对瀑布价值观和预见性实践的根本否定。他们所表达的 RUP 信息是不正确的，并且常常掩盖了潜在的瀑布心理。仔细深入地研究 RUP 的原始文档，理解迭代开发的真正价值，然后与自己进行对照。

近来我们至少在如下一些地方看到了对 RUP 的错误解释：

- 一本面向对象项目管理的书
- 一本关于服务器端编程的 Java 的书
- 一次 UML 会议中的一次演讲

错误信息最典型的特征是类似于如下描述 RUP 的四个阶段：

1. 初始－分析确定大部分的需求
2. 细化－进行详细的设计和建模
3. 构建－实现；将模型转换成代码
4. 发布－集成，质量保证，部署

我们还发现了所谓的 RUP 专家的如下错误的论述：

- “对于 RUP 来说，在开始设计前获得 100%的需求定义是非常重要的。”
- “一个良好的、典型的 RUP 迭代周期大约应当是 6 个月。”

为了增加应用 RUP 失败的机会，建议你不加辨别地接受你所看到和听到的关于 RUP 的信息。目前有很多错误的信息可以误导你。

你要明白你并没有真正理解 RUP，如果……

为了保证在应用 RUP 时完全曲解和失败，建议如下去做。当然，你做得越好，就越易失败。

你要明白你并没有真正理解 RUP，如果……

- 认为初始=需求；细化=设计；构造=实现。
- 认为细化阶段的任务是充分而仔细地建模，然后在构造阶段将此模型转换成代码。
- 认为细化阶段只是创建了原型，而事实上，在产品质量中有风险的体系结构的元素的核心应该在细化阶段实现。
- 设计和实现之前尽力确定大部分需求。
- 实现之前尽力确定大部分设计。
- 开始编码前，有很长的一段时间用于需求和设计工作。
- 组织上认为合适的迭代周期单位是月，而不是周。
- 认为编程前的阶段—用 UML 制图及进行设计活动前的那段时间，应该充分、准确、详细地进行设计和建模，认为编程只是简单而机械地将这些设计和模型转换成代码。
- 尽力从始至终地为项目制定详细的计划，为每一个迭代分配工作；怀着侥幸心理去预测所有的迭代周期会发生什么。
- 在进入细化阶段之前，组织上就要求可靠的项目计划和估计。
- 组织上认为采用 RUP 意味着要进行很多活动，创建很多文档，贯彻 RUP 要遵循很多固定的步骤。

结论

我们确信，遵照本文的七个步骤和上述的一些曲解，你在应用 RUP 和迭代开发时注定会一团糟。

参考文献

1. Kruchten, Philippe. The Rational Unified Process—An Introduction, 2nd ed. Reading, MA: Addison-Wesley (2000).

2. Jacobson, I., Booch, G., and Rumbaugh, J. The Unified Software Development Process. Reading, MA: Addison-Wesley.(1999)

3. Royce, Walker. Software Project Management—A Unified Framework. Reading, MA: Addison-Wesley.(2000)

4. Johnson, Jim. Chaos: Charting the Seas of Information Technology. Published Report. The Standish Group (date?)

5. Druffel, Larry and Heilmeyer, George. Report of the Defense Science Board Task Force on Acquiring Defense Software Commercially. (USA Department of Defense)

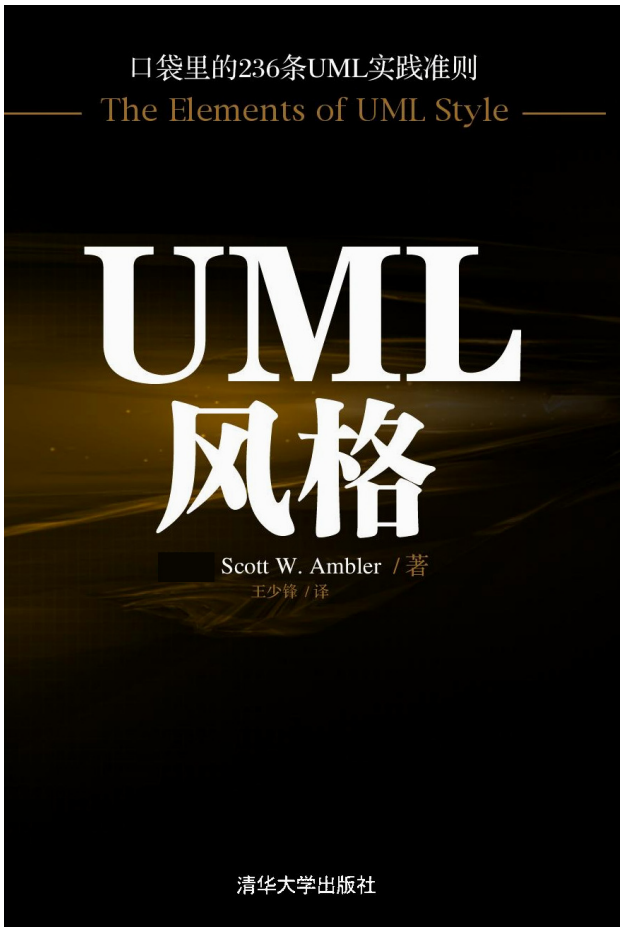
6. Jones, Caper. Applied Software Measurement. NY, NY: McGraw-Hill.

7. Boehm, B. and Papaccio, P. "Understanding and Controlling Software Costs," in IEEE Transactions on Software Engineering. Oct 1988.

8. Beck, K. Extreme Programming Explained—Embrace Change. Reading, MA: Addison-Wesley.(2000)

9. Fowler, Martin "Put Your Process on a Diet," in Software Development. Dec 2000. CMP Media.

10. Booch, G. Object Solutions: Managing the Object-oriented Project. Reading, MA: Addison-Wesley.(1996)


口袋里的236条UML实践准则
The Elements of UML Style
UML
风格
Scott W. Ambler / 著
王少锋 / 译
清华大学出版社


mobile

关于重构研究与实践的一次论述

Bart Du Bois、Pieter Van Gorp、Tom Mens 著，风之影 译

查看评论

摘要

在本文中，我们从理论和实践的两个方面，对软件重建（restructuring）和重构(refactoring)领域进行了详细的综述。首先对重构的当前应用和支持工具进行总结。然后对重构许多重要的最新研究进行讨论。最后，我们着重指出重构研究正在前进的关键方向。

关键字

重构、调查(survey)、重建、模型变换（model transformation）

1 导言

在现实环境中，软件的一个本质特征就是需要不断地发展。随着软件不断地增强、修改和适应新需求，代码变得越来越复杂，慢慢地偏离了最初的设计。因此，整个软件开发的开销的主要部分，集中于软件维护阶段了[1,2,3]。更好的软件开发方法和工具也不能解决这个问题。因为在同一时期的框架下，它们增强的能力是用来实现更多的需求，这又使得软件变得更加复杂。

为了解决软件复杂性的螺旋发展，迫切需要一种技术，通过增量式地改进软件的内部结构，来降低软件的复杂性。解决该问题的研究领域称为重建（Restructuring）[5,6]，在面向对象软件开发中称为重构（refactoring）[7,8]

依照Chikofsky 和 Cross的逆向工程（reverse engineering）分类学[9],重建定义如下：

重建是在相同的相关抽象层次上，从一个表现向另外一个表现的转换，同时保留主题系统的外部行为（功能和语义）不变。重建转换通常是一种表现，例如修改代码来改进传统结构化设计中的结构。在重建对主题系统进行实现或提议修改而产生的新版本中，通常不会涉及为新需求而进行的修改。然而，重建通过改进系统各个方面，也会使主题系统具有更好的可观察性（observation）。

重构的定义也基本相同：“*改变系统的过程，通过这样的一个过程，不会改变代码的外部行为，但可以改进其内部结构*”[8]。这里的关键思想是重新分配类、变量和方法，使得将来的适应和扩展更加容易。

本文如下组织：第二部分描述了在当今软件开发过程和工具中，重构所处的位置。第三部分概述了重构基本和实际问题的最新研究。最后，第四部分是通过一系列基本问题预示着重构研究的未来趋势。

2 当前应用

为了阐述重构的当前应用，我们论述一下重构在现今软件开发过程中的角色，并纵览最近的支持工具。

2.1 重构在现今软件开发过程中的角色

重构被应用于敏捷软件开发和再工程（reengineering）中。

诸如eXtreme Programming(XP)的敏捷开发过程能够依靠重构，是因为其短的迭代开发周期[10]。也因为相同的原因，对于直线瀑布或螺旋的软件开发过程模型，重构就不是很合适[11]。因此，一直存在一个公开的问题，重构是否能被更多的传统软件开发过程中所采纳，并如何来做。

文献[12]描述了重构如何适应于软件再工程(reengineering)过程。在软件再工程中，经常用来重建把原有代码转化为更加模块化或结构化的形式，甚至把代码移植到不同的编程语言，甚至是不同的语言范畴(language paradigm)中[13]。

像Smalltalk VisualWorks[14]、Eclipse[15]和IDEA[16]等集成开发环境(IDE)对XP进行相当大的支持，其是通过组合XP的两个核心活动（组合重构支持和单元测试(unit testing)）来完成的。

2.2 支持工具

虽然也可以采用人工方式重构，但能够采用支持重构的工具来支持重构也是至关重要的。如今，众多工具可以用来实现重构各方面的自动化（想进一步了解重构工具的最新信息，请参看 http://www.refactoring.com）。根据工具和所提供的支持种类不同，自动化程度会有所不同。

诸如Refactoring Browser[17]、 XRefactory[18]、jFactor[19]此类工具，可以支持半自动化方法进行重构。

一些研究者示范了全自动化重构的可能性。例如，Guru就是一个全自动化重构工具，其能够实现SELF程序的继承体系（inheritance hierarchies）重建和方法重构[20]。文献[21,22,23,24]提出了其它全自动化重构方法。

把重构工具直接集成到工业级集成开发环境中，现在也是一种趋势。例如Smalltalk VisualWorks v7 [14]、Eclipse v2 [15]、Borland Together Control-Center v6 [25]、IntelliJ IDEA v3 [16]、Borland JBuilder v6 [26]等等。

所有这些工具的焦点是根据用户的需要来应用重构。正如我们将在4.4章节所论述的，这些工具对诊断何处、何时应用重构却很少支持。文献[27]陈述了通过度量的方法来达到，而文献[28]则通过使用Daikon工具自动诊断程序不变式(invariants)，指出哪里可以应用重构。这种方法是基于运行期行为的动态分析，可以作为其他方法的补充。

3. 最新研究

3.1 什么技术能被用来实现重构

正如2.2章节所提到的，自动化水平是根据重构工具的不同而不同。依照不同的需要，不同的工具可以基于轻量级或复杂而强大的实现技术。

3.1.1 模式匹配(pattern matching)技术

早期的尝试者仅仅利用shell-based正则表达式工具（例如sed）和编译器的语义分析器的组合使用来重构代码。这种方式非常灵活，但也只能小步前进。

3.1.2 元建模（metamodeling）技术

那些对重构明显支持的工具把执行重构看成是模型转换。在执行之前，需要验证重构的前提条件，保留编译能力（compilability）和行为特征不变。这都是依靠从程序源码中建立模型的解析器。

命令式(Imperative)

就像Java能够抽取出表示Java抽象语法树（abstract syntax tree）的类模型一样，以代码为核心的开发环境可以实现重构。这种方式有一个缺点，像Java这样第三代语言写的程序，推论就变得不切实际了。因此，Tip, Kiezun和Baumer给开源的Elipse IDE提出了一种类型约束符号，用以推论重构的正确性[29]。

说明性(Declarative)

研究者们正在研究说明性的形式体系（formalism）如何用于重构，并立刻推论出他们的行为。例如，图重写适合用来推论重构已经得到证实[30,31]，并能从一个宣言式定义中生成可执行的重构代码[32]。在UML中，OCL表达式被用来说明重构契约，例如有关重构的代码味道、不变式、前件和后件[33]。最后，logic meta programming技术也能被用来翻译和推论面向对象程序[34]。SOUL已经集成了重构的“代码坏味道侦察”，并能自动化地提示什么Smalltalk重构应该被应用，用来解决那些被PROLOG规则描述的有问题代码[35]。

3.2 我们如何以可扩展的方式组合重构

在一个大规模的再工程项目中，复杂重构(complex refactoring)的支持变得很重要。但目前为止，我们的开发工具和形式体系(formalism)却仅仅支持基本的重构。为了能够实现支持工业软件的复杂重构工具，首先得要求，基本重构组合成复杂重构是可能的，而且是容易的[36]。

复杂重构的一个优点，就是它比等同的一系列基本重构更加有效。例如，对于复杂重构，我们只需要验证一次前提条件，要比分别为前后的每个基本重构来一次好得多[37,38,39]。

为了验证复杂重构的有用性，使用了许多研究方法。正如文献[40]那样，发展中的软件系统能被检测出复合式重构(composite refactorings)。同理，基本的设计模式转换也能够进行组合。文献[41]描述了三种组合方式：序列化(sequencing)表示按次序一个接一个进行转换；集合迭代(set iteration)意味着在程序元素的集合中，迭代地执行转换；并发（concurrency）表示一系列转换并发执行。这种组合也能基于代码味道，通过学习味道如何解决也能引入其他东西。

通过对OCL重构契约的评测，自动化提示UML重构和序列化基本重构能够被实现。应用UML/OCL标准的一个好处，就是广泛的使用者能够调整出一个自动化重构工具的组合规则。

3.3 我们如何分析重构之间的依赖关系

在建立复杂重构的时候，有一件事也很关键。就是需要确定哪些重构是相互独立的，哪些重构是必须按次序应用的。重构的平行应用通常会导致一些意想不到的冲突[38]。一个典型的例子就是，面向对象框架的重构可能导致它的不同实例之间产生冲突。合并(Merge)技术在这里可能用得上[42]。

因此，检测和解决这种冲突的一个规范原则，就显得非常重要[43,44]。从理论的观点来看，我们可以利用一些现有成果，就是图转换系统[45]和临界对分析[46]中的并行和并发技术。

检测重构之间的先后依赖关系，在解决变化传播中时也很有重要的[47]。因为所谓的“波纹效应(ripple effect)”，一个重构的应用往往也需要其他许多重构被应用。

3.4 我们如何以语言无关的方式来定义重构

重构的工具或规范的模型应该具有足够的抽象，以便于应用于不同的编程语言。同时也应该提供必要的钩子(hook)，用来添加各个语言所特有的行为。多语言支持是令人期盼的。它不仅能在IDE中防止重复重构，而且更为重要的是，它能使工具使用者尽可能地共享不断调整的代码坏味道定义。

FAMIX是第一个再工程多语言（例如Smalltalk、C++和ADA）的元模型 [48]。其中一个扩展版本，能够支持高级Java重构，被映射为一张类型图(type graph)[30]和UML元模型[33]。

除次之外，还有就是文献[49]所描述的，一个开发语言无关的重构定义框架。这个框架在结构化编程语言中是应用通用函数。

重构在比编程语言更依赖于实现的层次上也很有用处，比如二进制码的重构，甚至是可执行代码的重构。这方面的工作是由编译器的开发者和编译生成器[compiler generators]来做的[50,51]。在这里，关注点不是优化效率而是一种转换二进制代码的技术，使得其能够跟重构定义所兼容。根据应用领域的不同，这项技术也能被用来提高一个系统的其他方面(安全性、耗电特性、代码长度、内存使用情况等等)[52]。通过对运行时编译器(runtime compiler)[53]的研究，结合重构技术，就能够让正在运行的应用程序在不用重启的情况下就可以得到改变。

3.5 如何让重构在更高抽象层次中应用

重构在比源代码和低级别设计模型更高抽象层次上，也是有用的。研究者们正在研究高级别模型的重构，设计模式和软件架构的重构：

设计模型(design model)可采用统一建模语言（Unified Modelling Language）来明确定义了。目前已经完成了很多尝试，提供对UML模型和图的重构支持[45,55,56,57]。分层图(hierarchical graph)和分层图重写(hierarchical graph rewriting)[58]提供了一种有趣的形式来表示更高抽象层次的程序转换。我们已经研究了对相同抽象层次的产物（例如交互图和类图）之间进行结合[56,60]，但是对不同抽象层次的软件产物(图和代码)进行结合的集成方法目前还欠缺。

设计模式是一种在高级别抽象层次上描述软件结构的方法[61]。重构通常也会引入新的设计模式到软件中[23,24,62,63]。设计模式也能添加约束给软件结构，其能够限定每个特定重构的适用性。为了检测这个，我们可以采用逻辑规则[34]。文献[22]建议用图转换技术把出现坏设计模式的地方重构或替换为好的设计模式。

为了处理软件架构的重构，文献[64]提议了一种方法。这种方法认为，重构的规则应该直接基于一个系统架构的图形表示。这种规则利用组件行为之间的因果关系来保持行为。从而这种方法似乎有前途的。一种更加实效的方法在文献[36]中提起：两个软件系统的架构改变通过执行一系列基本重构来完成(第一次个案研究的81个重构，第二次个案研究的800个重构)。

3.6 如何让重构涉及到其他软件工程技术

在消除软件发展过程中的障碍时，重构仅仅是被用来工程化软件系统多种技术中的一种。重构被排到软件工程技术中的软件分析(3.6.1)和测试(3.6.2)中。通过对重构的划定，把应用程序的范围限定到重要问题的解决和需求的满足，从而提高投资的回报。

3.6.1 程序分析

程序分析的目的就是在编译时自动测定程序的特性。这些特性包括内部质量特性、代码级别的问题（如重复代码）和潜在的优化等。发现这些特征需要使用检测技术。在这里，我们讨论一些具有代表性的检测技术：

Software Metrics[65,66,67]认为用真实数据标识了软件内部特性，从而指出质量特性的当前得分。软件质量能够被用来指出源代码方面的问题，包括尺寸、复杂度、耦合度、聚合性等等[68,69,70]。因此，软件质量技术能够用来检测一个软件系统哪里需要进行重构。

通过计算重构前后的软件质量，能够检测出特定质量特性的变化。软件检测原理认为，可以鉴别出内部和外部（诸如可维护性）质量特性之间很有意义的关系。例如，文献[71]定义了多项式测量方法。这种方法提供了简单的可维护性指数。基于这些外部特性的指示器，重构的效果可以被评测出来。

利用这些检测技术的自动化能力，可以分析出一个软件系统不同版本之间的差异。文献[40]提议，应用变化度量技术来检测一个软件系统连续两个版本之间的重构。

Software Visualization用图形化的方式来表示一个应用程序的内部结构。它有助于软件的重建（Restructuring)。文献[72]建议利用星形图（star diagram）来达到这个目的。，DupLoc就是一个基于克隆展示来检测重复代码的图形化工具[73,74]。

元建模(Metamodeling)能够让重构更少地依赖于实现语言。这个正如3.4节讨论的那样。相同的技术也能被用来制作语言无关的程序分析工具。

Program slicing[75,76]分析系统各个部分的依赖关系。这部分信息也可以被利用来处理特定类型的重建(restructuring)：函数和过程萃取[77,78]。这种基于系统依赖图的技术，可以保证一个重构保护了一些所关心的行为。相似地，文献[79]采用了更非正式的方法。在控制流图中，利用运算法则一起移动被选择的一些节点，使其具有更多扩展性，同时又保护了程序语义。

规范概念分析(Formal Concept Analysis)技术能用于处理重建(restructuring)[80]。文献[81]中应用概念分析来重建面向对象类阶层体系(class hierarchy)。其是基于对一系列程序类层次的使用。重建后的结果也可以得到保证,跟原来的类层次行为上保持相同。文献[82]中也利用该技术来重建(restructure)软件的模块。文献[83]中使用概念分析，通过对已有的数据结构进行半自动化重建来识别对象。

3.6.2 测试

重构最关心的是在转换之后行为保持不变。但没有规范的证明，来确保重构工具的正确性（参考 4.1 节）。语义测试是验证程序行为保持的唯一方法。事实上，自动化单元和集成测试可以让回归测试（regression test）在重构之前和之后运行一次，从而让我们在增量式程序转换过程中感到安全。

在重构已有系统的过程中，识别出系统哪些部分首先需要编写测试是一个重要的问题。文献[12]提议从最关键的组件开始，慢慢地增加你的测试用例。正如文献[84]所建议的，重构本身能被用来提出测试。相反，回归测试系统是限定需要运行的测试数量。这在短迭代周期的软件系统中，面对大量的测试用例时，是有好处的[85]。

测试也能被用来掌舵重构过程。文献[86]中就在回归测试过程中，使用动态切割(dynamic slicing)来定位特征实现，目的是萃取功能。文献[87]介绍了测试优先重构的观点，在这里，测试代码是用来分析产品代码，以便发现需要改进的地方。它们也认为，测试代码的重构是不同于产品代码的重构，表现为两种方面：(a)它们各有一套明显的坏味道。(b)改进测试代码还涉及额外的测试特定的重构。

3.7 如何比较重构的工具、技术和形式体系(formalism)

为了能够评测一种重构工具或形式体系(formalism)比另一个是否更适应，它们需要严格地进行比较。这个可以通过相应的发展依据(evolution criterion)分类学为基础来实现。文献[88]进行了第一个努力，给比较软件发展工具定义了大体的分类法。文献[89]应用这个分类法来比较四种不同的重构工具。

4. 未来趋势

4.1 什么是行为，重构是如何来保持的

重构意味着程序的“行为”受到保护。但是行为的准确定义却很少被规定，或者定义地太不充分以至于不能在实际中被检验。

另外一个问题是，对可观察行为等同有一个直觉上的定义，认为是“对于相同的输入，我们应该获得刚好的相同输出”。其实这是不够的。在许多应用领域中，可观察到的输入-输出语义不仅仅是这么一回事：

* 对于一个实时系统，行为的基本表现就是操作（序列）的执行时间；

* 对于嵌入式系统，内存限制和耗电性也是需要被考虑的有关行为的重要方面；

* 对于安全关键（safety-critical）系统，存在具体的安全概念，这也是重构过程中需要被保护的。

因此，在现实的世界中，重构应该也能够保留这些特性。实际上，不是所有的系统都需要保护所有这些特性。所以我们事实上对行为保持有广泛的理解。

从实践的角度来看，对待行为保持需要以非常实效的方式，例如依靠严格的测试规则。如果我们有了相当完备的测试用例，而且重构之后它们都能运行通过，那是很好的证据认为重构是行为保持的。

从规范的角度来说，可以尝试确定一种强表达能力的规范语言。我们可以用它来表达程序的不变式，而且把它与一系列重构关联起来，能够保证所有可表达的属性受到保护。这与目前比较有名的Courcelle中关于单值二阶谓词逻辑相似[90]。

在研究文献中，不同观点的行为保持在重构中被应用。文献[91]定义了对象保持类转换(object-preserving class transformation)。文献[92]应用特别类型的图转换，它具有语言保持(language-preserving)的特性。所有可接受的程序输入在转换之前和之后必须保持一致。同样地，它也提供了基于规范化语言理论的用于重建的框架。

4.2 重构在模型驱动软件开发中地位如何？

OMG的Model Driven Architecture(MDA)能够平衡UML profile，OCL和MOF的相关标准。使得编程语言和组件模型的集成发展变成可能[93]。通过明确地把平台依赖关系从Platform Specific Model分离到对Platform Independent Model的映射中[94]，实现了平台无关性。MDA代码生成器是基于model-model和model-code转换，这个转换是可以通过所谓的“模式”或“模板”编辑器来配置的[95,96]。关于重构，几个问题需要更深入地研究。

正如3.5节所描述的，如何集成程序代码和更高抽象层次的模型，并没有统一的意见。在MDA中，转换应该确保生成的模型和代码是保持一致的。然而每次重构之后重新生成整个系统，对于增量开发（incremental development）来说需要太多的资源。第二，重构一个系统可能不仅仅是在最抽象的层次上。最后，还缺少一种优于现在编程语言的领域工程语言。因此，大部分MDA应用程序是通过手工编写程序代码来完成的。因此，抽象模型的重构也需要映射到基本的源代码重构。基于以上所有问题，我们需要规范的原则来详细分析MDA代码生成器转换之间的关系[32]。

高级的MDA工具能够逆向工程，即把现有代码转变成抽象模型，从而实现平台集成[97]。显而易见地，这种解析方法只有在现有代码遵守一个预定义结构时才能工作。这样，重构能够被用来把现有代码转换为MDA工具所能理解的规格。而在实践中如何扩展这种方法工作以及扩展什么，这方面还需要更多的研究。

第四代语言(4GL[98])CASE工具的有限接受，则表明软件开发者对较高抽象层次的编程还要求其再具有一些（也许是相当大的）灵活性。在MDA中，当他们遇到模型转换和代码生成模板时，就把代码生成器看成白盒子。对于这种代码生成器，还有一个问题有待研究，就是我们应该用什么重构，以便其能够适应新的平台和优化生成的代码的效率。

4.3 重构在面向方面软件开发(AOSD)中地位如何？

在面向方面领域中，重构面临双重问题。首先，为了具有更好可进化性(evolvability)而对程序的重构，将会涉及到作用于程序的所有Aspect。例如Ecplise[15]的环境对重构和面向方面编程都提供支持。但是当在改进过程中，添加出现新的连接点时，会忽略对Aspect的改进，尤其是Pointcut（构造器描述了哪些程序点(propgram join)、join points、aspect code被组织）的改进。此外，关注点(concern)的明显分离也会造成Aspect对基础程序的所有变化不会关心。这也可能导致基础程序和它的Aspect相互矛盾。

文献[99]中正在开发一种pointcut语言，其具有更多表达能力的pointcut。这样的语言允许通过良好定义的模式(well-defined patterns)来识别join points，而不会导致基础程序和Aspect之间的紧耦合。

其次，通过对Aspect的引入，诸如代码混乱和代码分散的一些问题可以得到很好地解决。把一个程序的横向关切点(crosscutting concerns)(例如错误处理、并行控制等待)萃取(extraction)成Aspect也是重构的一种形式。这种类型的萃取和aspect code的自身改进，被认为是面向方面重构（aspect-oriented refactorings)。

到目前为止，支持面向方面重构的工具，主要局限为横向关切点的探测。Aspect Browser[100]帮忙查找和管理crosscutting concern；Aspect Mining Tool（AMT）[101]能够在现有系统中挖掘Aspect；Eclipse插件，the Feature Exploration and Analysis Tool (FEAT)[102]帮助浏览现有系统中的crosscutting concern。

现在，Hannemann,、Murphy和Kiczales正在做一个项目。这个项目旨在开发一个是基于对话的Eclipse插件工具，提供半自动化面向方面重构。

4.4 我们如何决定哪里和什么时候应用重构？

诸如重构这样的强大技术，只有在使用时获得反馈，才能发展到最大。在过去的十年，在优秀的面向对象设计领域中，最多实践形式的反馈一是模式[104,61,12]、方针（guidelines）和启发（heuristics）[105,106]。

在重构使用的反馈中，获得了分析如何重构来改变系统。例如，文献[107]学习如何使用重构来提出设计模式。在同一个层次，文献[108]研究如何重构来改变内部系统的语义。

为了跟踪软件外部质量特性（例如维护性、效率），实用的研究还是很有必要的。文献[109,110,111]研究了启发设计在维护阶段的价值。同时这些文献的实用性研究也增加了证据，证明重构的实用型研究是很有价值的。

总而言之，要解决改进软件系统的这个问题，其关键之处在于既要知道在哪里也要知道为什么要改变系统，以及知道重构是如何影响一个系统的。

5. 结论

软件重建和重构的研究还会保持活跃。虽然商业的重构支持工具正在快速增长，还是有许多公开问题继续存在，需要得到解决。我们认为，抓住行为保持（behavior-preservation)，收集针对哪里使用和为什么使用重构的反馈，对MDA和AOSD的集成，作为重构以后主要发展方向。

大体上来说，需要形式体系、程序、方法和工具，以更为一致的、更有指导性的、更可升级的、更灵活的方式，来记录重构。

6. 参考文献

[1] D.M. Coleman, D. Ash, B. Lowther, and P.W. Oman. Using metrics to evaluate software system maintainability. IEEE Computer, 27(8):44–49, August 1994.

[2] T. Guimaraes. Managing application program maintenance expenditure. Comm. ACM, 26(10):739–746, 1983.

[3] B. P. Lientz and E. B. Swanson. Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations. Addison-Wesley, 1980.

[4] Robert L. Glass. Maintenance: Less is not more. IEEE Software, 15(4):67–68, July/August 1998.

[5] Robert S. Arnold. Tutorial on Software Restructuring, chapter An introduction to software restructuring. IEEE Press, 1986.

[6] William G. Griswold. Program Restructuring as an Aid to Software Maintenance. PhD thesis, University of Washington, August 1991.

[7] William F. Opdyke. Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[8] Martin Fowler. Refactoring: Improving the Design of Existing Programs. Addison-Wesley, 1999.

[9] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. IEEE Software, 7(1):13–17, 1990.

[10] Kent Beck. Extreme Programming Explained: Embrace Change. Addison Wesley, 2000.

[11] B. Boehm. Software engineering. IEEE Transactions on Computers, 12(25):1226–1242, 1976.

[12] Serge Demeyer, St´ephane Ducasse, and Oscar Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann and DPunkt, 2002.

[13] Richard Fanta and V´aclac Rajlich. Restructuring legacy C code into C++. In Proc. Int'l Conf. Software Maintenance, pages 77–85. IEEE Computer Society Press, 1999.

[14] Cincom. Smalltalk VisualWorks, January 12 2004.

[15] Eclipse. Eclipse, January 12 2004.

[16] IntelliJ. IDEA, January 12 2004.

[17] D. Roberts, J. Brant, and R.E. Johnson. A refactoring tool for Smalltalk. Theory and Practice of Object Systems, 3(4):253–263, 1997.

[18] XRef-Tech. XRefactory, January 12 2004.

[19] Instantiations. jFactor, January 12 2004.

[20] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In Proc. Int'l Conf. OOPSLA '96, ACM SIGPLAN Notices, pages 235–250. ACM Press, 1996.

[21] E. Casais. Automatic reorganization of object-oriented hierarchies: a case study. Object Oriented Systems, 1:95–115, 1994.

[22] Jens H. Jahnke and Albert Z¨undorf. Rewriting poor design patterns by good design patterns. In Serge Demeyer and Harald Gall, editors, Proc. of ESEC/FSE '97 Workshop on Object-Oriented Reengineering. Technical University of Vienna, 1997. Technical Report TUV-1841-97-10.

[23] Benedikt Schulz, Thomas Genssler, Berthold Mohr, and Walter Zimmer. On the computer aided introduction of design pattern into object-oriented systems. In Technology of Object-Oriented Languages and Systems, pages 258–267. IEEE Computer Society Press, 1998.

[24] Mel ´O Cinn´eide. Automated Application of Design Patterns: A Refactoring Approach. PhD thesis, Department of Computer Science, Trinity College, University of Dublin, 2000.

[25] Borland. Borland Together ControlCenter, January 12 2004.

[26] Borland. Borland JBuilder, Janurary 12 2004.

[27] Frank Simon, Frank Steinbr¨uckner, and Claus Lewerentz. Metrics based refactoring. In Proc. European Conf. Software Maintenance and Reengineering, pages 30–38. IEEE Computer Society Press, 2001.

[28] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In Proceedings of the International Conference on Software Maintenance, pages 736–743. IEEE Computer Society Press, 2001.

[29] Frank Tip, Adam Kiezun, and Dirk B¨aumer. Refactoring for generalization using type constraints. In Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, pages 13–26. ACM Press, 2003.

[30] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In Graph Transformation, volume 2505 of Lecture Notes in Computer Science, pages 286–301. Springer-Verlag, 2002. Proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.

[31] Paolo Bottoni, Francesco Parisi Presicce, and Gabriele Taentzer. Specifying integrated refactoring with distributed graph transformations. In AGTIVE '03 - Applications of Graph Transformation with Industrial Relevance, pages 227–242, 2003.

[32] Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing refactorings as graph rewrite rules on a platform independent metamodel. In Holger Giese and Albert Z¨undorf, editors, Proceedings of the 1st International FUJABA Days, pages 17–24. University of Kassel, Oktober 2003.

[33] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In Proceedings of the 6th International Conference on UML – The Unified Modeling Language., 2003.

[34] Tom Tourw´e. Automated Support for Framework-Based Software Evolution. PhD thesis, Vrije Universiteit Brussel, September 2002.

[35] Tom Tourw´e and Tom Mens. Automatically identifying refactoring opportunities using logic meta programming. In Proc. Int'l Conf. Software Maintenance and Re-engineering (CSMR), 2003.

[36] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. Automated Software Engineering, 8(1):89–120, 2001.

[37] Don Roberts. Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[38] Tom Mens. A Formal Foundation for Object-Oriented Software Evolution. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 1999.

[39] Tom Mens. Transformational software evolution by assertions. In CSMR Workshop on Formal Foundations of Software Evolution, Lisbon, March 2001.

[40] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In Proc. Conf. Object-oriented Programming, Systems, Languages, and Applications, volume 35(10) of ACM SIGPLAN Notices, pages 166–177. ACM Press, 2000.

[41] Ladan Tahvildari and Kostas Kontogiannis. A methodology for developing transformations using the maintainability soft-goal graph. In In Proceedings of the 9th IEEE Working Conference on Reverse Engineering (WCRE), pages 77–86. IEEE Computer Society Press, 2002.

[42] Tom Mens. A state-of-the-art survey on software merging. IEEE Trans. Software Engineering, 28(5):449–462, May 2002.

[43] Tom Mens. Conditional graph rewriting as a domain-independent formalism for software evolution. In Proc. Int'l Conf. Agtive 1999: Applications of Graph Transformations with Industrial Relevance, volume 1779 of Lecture Notes in Computer Science, pages 127–143. Springer-Verlag, 2000.

[44] Tom Mens. A formal foundation for object-oriented software evolution. In Proc. Int'l Conf. Software Maintenance, pages 549–552. IEEE Computer Society Press, 2001.

[45] P. Baldan, A. Corradini, H. Ehrig, M. L¨owe, U. Montanari, and F. Rossi. Handbook of Graph Grammars and Graph Transformation, chapter Concurrent Semantics of Algebraic Graph Transformations, pages 107–188. World scientific, 1999.

[46] Reiko Heckel, Jochen Malte K¨uster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In Graph Transformation, volume 2505 of Lecture Notes in Computer Science, pages 161–176. Springer-Verlag, 2002.

[47] Vaclac Rajlich. A model for change propagation based on graph rewriting. In Proc. Int'l Conf. Software Maintenance, pages 84–91. IEEE Computer Society Press, 1997.

[48] Sander Tichelaar, Stephane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta model for language independent refactoring. In Proceedings ISPSE 2000, 2000.

[49] Ralf L¨ammel. Towards Generic Refactoring. In Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02, Pittsburgh, USA, October5 2002. ACM Press. 14 pages.

[50] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. ACM Letters on Programming Languages and Systems, 1(3):213–226, September 1992.

[51] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison Wesley, 1996.

[52] Marijn Temmerman. Towards energy-conscious class transformations for data-dominant applications: a case study. Third PA3CT-symposium, 2003.

[53] Dries Buytaert and Frans Arickx. A selective runtime compiler for the wonka virtual machine. Presentation at PACT Symposium at University of Ghent, September 9-10 2002.

[54] Dave Astels. Refactoring with UML. In Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering, pages 67–70, 2002. Alghero, Sardinia, Italy.

[55] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring browser for UML. In Proc. 3rd Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering, pages 77–81, 2002. Alghero, Sardinia, Italy.

[56] Paolo Bottoni, Francesco Parisi-Presicce, and Gabi Taentzer. Coordinated distributed diagram transformation for software evolution. Electronic Notes in Theoretical Computer Science, 72(4), 2002.

[57] G. Suny´e, D. Pollet, Y. LeTraon, and J.-M. J´ez´equel. Refactoring UML models. In Proc. UML 2001, volume 2185 of Lecture Notes in Computer Science, pages 134–138. Springer-Verlag, 2001.

[58] G. Engels and A. Sch¨urr. Encapsulated hierarchical graphs, graph types and meta types. Electronic Notes in Theoretical Computer Science, 2, 1995.

[59] Niels Van Eetvelde and Dirk Janssens. A hierarchical program representation for refactoring. In Roswitha Bardohl and Hartmut Ehrig, editors, Electronic Notes in Theoretical Computer Science, volume 82. Elsevier, 2003.

[60] Ragnhild Van Der Straeten, Jocelyn Simmonds, and Tom Mens. Using description logic to maintain consistency between UML models. In Proceedings of UML 2003 – The Unified Modeling Language. Springer-Verlag, 2003.

[61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Languages and Systems. Addison-Wesley, 1994.

[62] Lance Tokuda and Don Batory. Automated software evolution via design pattern transformations. In Proc. 3rd Int. Symp. Applied Corporate Computing, October 1995.

[63] Pieter van Winsen. (re)engineering with object-oriented design patterns. Master's thesis, Universiteit Utrecht, November 1996.

[64] Jan Philipps and Bernhard Rumpe. Refinement of information flow architectures. In M. Hinchey, editor, Proc. ICFEM'97. IEEE Computer Society Press, 1997.

[65] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object-oriented design. IEEE Trans. Software Engineering, 20(6):476–493, June 1994.

[66] Norman Fenton and Shari Lawrence Pfleeger. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, London, UK, second edition, 1997.

[67] Brian Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, 1996.

[68] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In Qiaoyun Li, Bertrand Meyer, Gilda Pour, and Ruchard Riehle, editors, Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS 39), volume 1, pages 173–182. IEEE Computer Society, 2001.

[69] Houari A. Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In Proc. International Conference on Software Maintenance, pages 154–162, october 2000.

[70] Michele Lanza. Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization. PhD thesis, University of Berne, Switzerland, 2003.

[71] D. Coleman, P. Arnold, S. Bdoff, H. Gilchrist, F. Hayes, and P. Jeremaes. Object-oriented Development: the Fusion method. Prentice Hall, Englewood Cliffs, NJ, 1994.

[72] William G. Griswold, Morison I. Chen, Robert W. Bowdidge, and J. David Morgenthaler. Tool support for planning the restructuring of data abstractions in large systems. In Proc. 4th Symp. Foundations of Software Engineering, volume 21(6) of ACM SIGSOFT Software Engineering Notes, pages 33–45. ACM Press, October 1996.

[73] St´ephane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, Proc. Int'l Conf. Software Maintenance, pages 109–118. IEEE Computer Society Press, September 1999.

[74] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques. In Proc. Int'l Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), pages 25–36, 2003.

[75] F. Tip. A survey of program slicing techniques. Journal of Programming Languages, 3(3)(3):121–189, 1995.

[76] D.W. Binkley and K.B. Gallagher. Program slicing. Advances of Computing, 43:1–50, 1996.

[77] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. Trans. Software Engineering, 23(4):246–258, April 1997.

[78] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. In M. Harman and K. Gallagher, editors, Special Issue on Program Slicing, volume 40 of Information and Software Technology, pages 677–689. Elsevier, 1998.

[79] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 2000.

[80] B. Ganter and R. Wille. Formal Concept Analysis: Mathematical Foundations. Springer-Verlag, 1999.

[81] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In Proc. Foundations of Software Engineering (FSE-6), volume 23(6) of SIGSOFT Software Engineering Notes, pages 99–110. ACM Press, 1998.

[82] Paolo Tonella. Concept analysis for module restructuring. Trans. Software Engineering, 27(4):351–363, April 2001.

[83] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. 1998.

[84] Wright Graham. extreme programming in a hostile environment. Sessionpaper at Int. Conf. on eXtreme Programming, 2002.

[85] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In International Conference on Software Engineering, pages 211–220, 1994.

[86] Alok Mehta and George Heineman. Evolving legacy system features into fine-grained components. In ICSE 2002. ACM, 2002.

[87] Arie van Deursen and Leon Moonen. The video store revisited – thoughts on refactoring and testing. In Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering, pages 71–76, 2002. Alghero, Sardinia, Italy.

[88] Tom Mens, Jim Buckley, Awais Rashid, and Matthias Zenger. Towards a taxonomy of software evolution. In Proc. Workshop on Unanticipated Software Evolution, March 2003. Warshau, Poland.

[89] Jocelyn Simmonds and Tom Mens. A comparison of software refactoring tools. Technical Report vub-prog-tr-02-15, Programming Technology Lab, November 2002.

[90] B. Courcelle. Graph rewriting: an algebraic and logic approach. In J. Van Leeuwen, editor, Handbook of Theoretical Computer Science, Vol. B, pages 193–242. Elsevier, 1990.

[91] Paul L. Bergstein. Object-preserving class transformations. In Proc. Conf. Object-oriented programming systems, languages, and applications, pages 299–313. ACM Press, 1991.

[92] Paul L. Bergstein. Maintenance of object-oriented systems during structural evolution. Theory and Practice of Object Systems, 3(3):185–212, 1991.

[93] David Flater. Impact of model-driven standards. In 35th Annual Hawaii International Conference on System Sciences (HICSS'02), volume 9 of Lecture Notes in Computer Science, page 285. IEEE Computer Society Press, 2002.

[94] Jon Siegel and OMG Staff Strategy Group. Developing in OMG's model-driven architecture. Technical Report White Paper Revision 2.6, Object Management Group, November 2001.

[95] Compuware. Optimalj pattern-driven generator. 2002.

[96] Codagen. Codagen architect, 2002.

[97] Interactive Objects. ArcStyler. http://www.arcstyler.com/, March 2003.

[98] Ivan Verborgh. De stille kracht van 4GL's, getoetst aan de harde praktijk van de RAD Race. Database Systems 2003, 2003.

[99] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. Proceedings of the 2nd international conference on Aspect-oriented software development, pages 60–69, 2003.

[100] UCSD. Aspect browser, 2002.

[101] Jan Hannemann. Aspect mining tool, 2001.

[102] Martin Robillard. Feature exploration and analysis tool, 2002.

[103] Jan Hannemann, Gail Murphy, and Gregor Kiczales. Dialogue based aspect-oriented refactoring, 2003.

[104] W. Pree. Design Patterns for Object-Oriented Software Development. Addisson-Wesley, 1994.

[105] P. Coad and E. Yourdon. Object-Oriented Analysis. Yourdon Press, 2nd edition, 1991.

[106] Arthur J. Riel. Object-Oriented Design Heuristics. Addison-Wesley Publishing Company, April 1996.

[107] Joshua Kerievsky. Refactoring to Patterns, to appear. Addison Wesley Professional, 2004.

[108] Bart Du Bois and Tom Mens. Describing the impact of refactoring on internal program quality. International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), ICSM-workshop, 2003.

[109] F. Brito e Abreu, M. Goulao, and R. Esteves. Toward the design quality evaluation of object-oriented software systems. In Proc. Int'l Conf. Software Quality, pages 44–57, October 1995.

[110] Fernando Brito Abreu and Walcelio Melo. Evaluating the impact of object-oriented design on software quality. In Proc. 3rd Int'l Symp. Software Metrics, March 1996.

[111] L. Briand, C. Bunse, and J. Daly. An experimental evaluation of quality guidelines on the maintainability of object-oriented design documents. Empirical Studies of Programmers (ESP), 1997.

斐力庇第斯从马拉松跑回雅典报信，虽然已是满身血迹、精疲力尽，但他知道：没有出现在雅典人民面前，前面的路程都是白费。

学到的知识如果不能最终【用】于您自己的项目之中，也同样是极大的浪费。而这最后一段路最是艰难。

UMLChina 聚焦最后一公里，所提供服务全部与您自己的项目密切结合，帮您走完最艰难的一段路。

积极主动的软件工具使应用开发潜力最大化

fgonline 著，李胜利 译

查看评论

罗伯特弗朗西斯集团公司（RFG）相信，被看作角色与职业合集的软件开发，由于新型的技术和市场需求，将发生根本性的变化。尽管软件自身和支持开发的工具软件无处不在，但与前几年的基于代理的工具软件方面有了根本性的改进。关于软件发展的所有方面的观点变化也正在促使对这些活动所需技巧及其价值进行重新评估。IT 主管们应关注新兴的工具，依靠 IT 发展作用，来评估他们发展目标设想，并策划一个与新市场现实和可用资源相容的进程。

商业推介

承诺提高专业开发者生产率和保证产品质量的新的基于代理的应用开发工具正在进入市场。IT 主管应当评估这些产品与自身的人力资源和需求间的前后关系，以此来看这样的投资是否被批准。

降低软件成本以及改进软件质量的压力正迫使企业重新检验有关个人开发者能力产生价值的假设。IT 主管应忠实地评估他们的开发资源，并制定一个承认并奖励个人贡献的计划。

软件开发者工作在一个瞬息万变的环境中，因而需要时常更新他们的技能。无论通过讲授、阅读还是来自软件产品的支持，转化自工业思想领袖及更有经验同僚的知识，说起来比做起来容易。IT 主管们着手制定一个预算分配计划，为满足适当的技术增强的产品和服务的结合，以使合适的人员在适当的时间获得最优的结果。

新兴的工具采用积极主动的方式

职业生涯超过二十年的 IT 专家们无疑意识到许多技术被炒得过热，然后在实施中最终产生一些不利反应。这两种技术—人工智能和计算机辅助软件工程（CASE）工具，现在牢固地扎跟在我们共同的 IT 意识之中，以至于我们不再把他们认为是可用的商业规则。今天，在我们中间，谁会因为去参加 CASE 工具会议而申请差旅费呢？然而，吸收人工智能和 CASE 工具精华的下一代应用开发工具，为各种技能水平的开发者提供新层次的技术支持。

软件开发—包括从需求及由此引出的分析、设计、实施和集成的所有任务，要求一系列有专门技能要求的角色。随着世界一流系统在复杂性上的增加，其运行所需的知识也极大地增加了。在 1970 年或 1980 年代定义一个完全应用的功能，现在必须使用易于理解的图形界面分发到分布式系统中，这在短短几年前还是不可想象的，然而人类的平均脑容量没有增长。

早期的支持软件开发的工具集中于有效的分解和包装以分发系统。从结构化方法和早期的 CASE 工具到 UML 和模型驱动架构，这些方式是静态的或被动的。（参见 RFG 研究文档“模型驱动 IT：可视化到实现”和“用户需求指南：怎样得到你想要的，并且所见即所得”）用丛书中和培训班获得最新技术武装起来的开发者一般在准备使用时都试图用成批的模式查询参考书来学习新技术。

在二十世纪八十年代，人工智能方面的学术研究应用于程序开发工具及环境问题上。程序员入门系统被使用基于知识的系统技术构建，但商业诺言从来没有兑现。

尽管不受前者负担的影响，几个知名品牌服务于今天的 CASE 市场上。Borland 软件公司、CA 国际公司、微软公司和 IBM 公司的 Rational 品牌主导这个市场。然而，许多小的 ISV 厂商也一直提供解决方案，其中一个开始着手实现 KBS 和 CASE 的诺言。

Jaczone AB 是一家来自瑞典的具有值得羡慕的出身的新软件工具商，公司由 Ivar Jacobson 和他的女儿 Agneta 创建。该公司正用他的第一个叫做 WayPointer 的软件产品进入美国市场，该产品预示着新一轮智能软件工具浪潮。

Ivar Jacobson 发展了对 UML 和 RUP 的进化至关重要的“use case”（用例）的概念，他还在软件组件重用专题上发表了无数的文章，将重用概念应用于包装可重用的过程知识组件上。在与经常被 RUP 知识主体制服的客户打交道多年以后，Jacobson 团队决定法律化这些知识，使其在基于内容的系统中易于访问，多年的结果就是 WayPointer。

简单地说，WayPointer 是一个基于代理的 Rational Rose 伴侣，能在给定的时间内提供用户所需大量的前后关系的指导。这就是说，通过评估用户行为和前后关系（例如，正在构建的框图类型），WayPointer 能通过流程主动地参与提供帮助和指导。不像读一本书或听一堂课，这种知识通过系统而不是要求用户给予回应来“显现”。界面是和谐的，因此，它不像大多数基于代理的商业化帮助系统，在实践中不引人注目。正如下表说描述的，这种方法重要的潜质所在是能够作为书面和传统培训方法的补充，提供较高的回报。

表 1 知识支持方案

知识源	性　质	可用性	成　本	效　率
代理	主动或被动	低	低	高
讲授	主动或被动	低	高	高
CASE	大多数被动	高	中等	中等
书本	被动	高	低	低

来源：罗伯特弗朗西斯集团公司

这个工具对于任何一个从事 UML 和 RUP 技术的企业来说都值得学习。然而，随着这项技术的应用，这个名称可以被认为是一个双重协定，这一技术包括为一项新的子产业指出技术路线：通过利用在过程密集领域的广泛知识提供智能助手，这一领域在软件方面已有实例。随着这一技术配置更加简单，将给这一系统和其他系统的应用者用来捕捉和提出相关建议，“WayPointer for SAP”将是一个逻辑分支。

实际上，像带有灵巧工具的 SAP 以及类似的来自于 IDS Scheer AG 和 Intelli 公司的软件工具都试图支持企业应用，但是 Jacobson 使用灵活的知识基础的集成化方式是新奇的和具有商业感染力的。

全方位的开发技能—多种而不是一种开发方法

评估新工具改进开发者生产率和质量引发了关于开发者、开发工具和为他们付费的经理间的关系的古老问题。

评估软件开发者技能一直以来是都需要更多的艺术而少一些科学，但是一些真理不言而喻。一个简单的事实就是在大多数最高产和最不高产的开发者间存在较大的差异。典型的比例为 20:1。如果一个企业拥有一个较大的开发者及开发能力的储备库，像功能点一样明确地根据产出单元衡量，均匀地按十个十位数分布，结果是惊人的。将使前 10%的程序员效率增加 20%，会使后 20%的程序员成为多余。

从历史上看，这种认识已经产生如下设想：工具及改进的实践方法能满足持续的软件危机（供不应求）。能为不同水平的开发者所利用的工具首先交付给最大产量的人，然后通过等级向下寻求经济可行性。然而，当今的形式已改变，并且随着工具由于费用及质量原因而不断地向国外转移。可以根本性改变开发目标的工具的政治的后果可能引起详细审查。（参阅 RFG 研究文档“向国外采购：可变的选择方法”，“与向国外采购相结合的经营风险”及“在外购运作中保持员工士气”）

事实上，几十年前，Philip Kraft 在《Programmer and Managers》杂志发表评论：计算机编程程序化（1971，Springer-Verlag），写道：向结构化的编程转移可能被看作对低技能工人的管理技术，之后，为了降低费用施加更多的控制。在后来的几年里，当许多社会科学研究者研究这样的一个案例时得出了相反的结论—改进的实践方法及支持性工具解放了最好的程序员，使其把精力集中在工作中有创造性的方面，同时也改进了低层次的工作程序。新的积极的助手的出现，能够重新回顾此争议。

由于把他们的知识汇编成典，威胁到专家的利益，早期基于知识的系统受到了一些抵制。有时的解决方案就是从已出版著作或退休或期满的专家那里推导出规律。国防部在空-地战役中使用这种处理程序，这种程序试图从军事领导们那里捕捉到战斗的策略，并且能够使作战部队估计事态及询问诸如“在这种情况下如何做”的问题。

如果像 WayPointer 这样的新系统为知识生产者提供书籍及课程的可供选择的商业销路，他们能够通过分发插件的模式为作者创造新的市场。然而相对谦虚地说，这些系统的确仍然在他们的控制之下，尽管典型地由开发者自己掌握着书本里的可用知识。IT 主管应该考虑这样一种综合技能管理程序，它平衡最有特色的代理、书刊、讲师或团队培训和不断奖励那些提升自身技能的个人贡献者。（参见 RFG 研究文档“能力中心：经验是最好的老师”，“维持企业培训程序”和“使员工职业管理走向正规”）

不断提升承诺升值的技能

随着软件工具及系统生成技术的改进会使应用性开发者地位丧失，这种推测越来越广泛，但应用需求曲线和这些工具的相对不成熟很可能使角色集合至少延续十年。

今天的有效开发组织很可能是分布式的、层次性的和多文化的，具有制度化的从讲师到下级员工的知识转化。最好的团队使用可持续改进的进程。（参见 RFG 研究文档“建造一个讲授程序”和“个人 CMM：改进人力资源管理”）提供一个及时的、连贯的技能增强机会和对于将来得到和维持世界一流的生产率和质量是重要的。不管是在今天还是在可预见的将来，基于代理的主动的帮助工具是这种方式中最吸引人的选择。

RFG 相信为专业的软件开发者提供及时的知识转化和加强的基于代理的工具目前已准备就绪。如许多 IT 工作变得不需要技能和向海外转移一样，这些工具的影响将增强他们支持应用的范围。IT 主管应评估和监控这些新开发来决定他们环境中的可能影响的前后关系。随着多应用技术的知识基础的成熟，替代提供像分析和设计工作的重要任务的企业内部讲师的高成本方案的技术将变得更有价值，像 WayPointer 这样的工具将允许企业在拥有长期竞争力的基础上保持和提升在同行业中的竞争能力。