第 40 期（2004 年 9 月）

天线宝宝

X 非程序员
X-Programmer
软 件 以 用 为 本

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

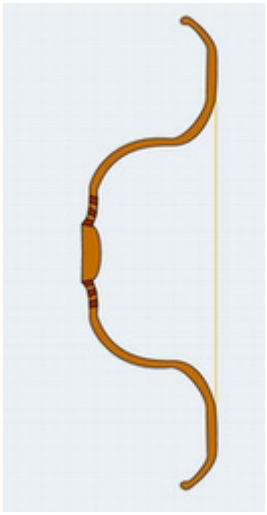
http://www.umlchina.com/

Sun 在应用生命周期管理（ALM）上另辟蹊径

[2004/9/22]

SUN 计划在未来三个月内在 ALM (application lifecycle management，ALM) 领域上好好表现一下，但 SUN 不打算采取和其它厂商一样的策略。

Sun 希望在今年底发布代号为“弓（Bow）”的 Java Studio 企业版 7.0。Bow 从 Embarcadero（今年 5 月被 Sun 收购的公司）那里将 UML 和 Java 代码的重构及配置（profile）集成起来，并增强对分布组织基于团队的开发支持功能。



Bow 将在 ALM 领域开创一个新时代。IBM 的 Rational 计划今年将发布其代码好 Atlantic 的最新的客户和服务器工具包，2005 年上半年微软将发布 Visual Studio2005 团队系统（VSTS），Borland 会发布 Themis，Themis 在该公司的 ALM 工具集基础上增加基于角色的访问控制。

流水线开发（Streamlining development）

这些公司有一个共同的策略就是简化 ALM 以支持流水线软件开发过程，并保证开发能够符合业务目的。另外，他们都致力于为 ALM 争取更多的用户。

微软的 VSTS 主管 Rick LaPlante 在上周指出，微软希望在未来 20 年内开拓一个庞大的 ALM 市场。

但是，Sun 对此存在异议。Sun 负责开发工具和平台的高级产品主管 Dan Roberts 对 ComputerWire 说，ALM 不会占领企业的所有空间，因为企业已经有了过去的开发系统和开发方法。

Roberts 先生认为今天的 ALM 应用是“niche”的（已经形成了一个小的环境），开发人员并不希望已经有了各种熟悉的工具，并不希望把这些都推倒，再强加一个完全的工具集。“革新要慢慢的来，因为(ALM 的改变)是文化的改变，我没见过企业开发人员大批量地改变开发工具……我看到的情况不是这样的”。

插件和合伙（Plug-ins and partnering）

Sun 正在绑定各种插件。Sun 将提供 UML、测试、性能管理以及其它协作功能，同时他们也在注意 Java 的蓝图，jForce 中心将帮助顾客。Sun 也在使用开源的配置管理和需求获取。尽管 Roberts 先生指出在需求获取方面会有潜在的收购可能。

Roberts 先生认为 Sun 的策略还包括通过工具提供各种熟练等级之间的跨越，开发人员可以在 Java Studio Creator 和 Sun Studio Enterprise 之间适应。

Sun 引以为豪的 Bow 的一个功能就是其协作功能。开发人员可以进行在线聊天、共享和编辑代码、共享桌面，在不同开发人员的源代码之间转换，在同样的环境中进行调试。

但是，和所有 Sun 的开发工具所面临的情况一样，Bow 面对是一个已经被各工具割据山头的战场，他们是 Borland 和 IBM Rational。在明年，随着微软最终发布 VSTS，这个领域将会发生地震级的变化。

消除鸿沟

如同其 Java 中间件一样，Sun 的工具试图消除市场和意识上的巨大鸿沟。Sun 在 1999 年买下了 Netbeans 这个 IDE 以加强其 Java 工具集，还买下了开发者社团，如果操作得当，Sun 应该已经在开发者中

但是，直到最近，NetBeans 似乎才稍有起色。而最近 IBM 新的 Eclipse 框架和社团（创建于 2001 年）已经有了大批的追随者，包括各种规模的 ISV（译者注：Independent Software Vender，独立软件开发商）和硬件厂商。

现在，Sun 似乎必须依靠某种推销方式，通过提供一系列的工具，来赢回开发者的支持。Sun 的市场地位的实现，包括其 iForce 机构的，意味着 Sun 要提供一个全面集成的，与 VSTS 相似的平台，这似乎有些不太现实。

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

PolySpace 和 EmbeddedPlus 发布第一个 UML 测试集成环境

[2004/9/14]

PolySpace 科技公司是运行时编译错误自动检测方面世界领先的公司。EmbeddedPlus 是 UML 功能测试解决上的领先者。这两家公司今天在 Boston 的嵌入系统大会上宣布 Developer-EP，为基于 UML 项目服务的集成测试环境，已经基本可用了。

Boston，2004 年 9 月 14 日 - 基于 EmbeddedPlus 和 PolySpace 的技术，Developer-EP 可以从各种流行的开发环境中读取 UML 信息，它可以自动生成测试用例和脚本，以检查 UML 图中设计的功能是否被正确地实现。该产品中还包括了测试结果收集和代码覆盖测试分析报告。

Developer-EP 还可以帮助开发人员分析源码，检查出通常的运行时错误，如数组越界、非法指针废弃等。它并不要求测试用例已经写好，也不需代码可执行，只要代码一完成就可以进行。在源码中的所有错误都会被清晰地标注出来，高亮显示。

EmbeddedPlus 的工程部副总裁 Salah Obeid 说，“保证 UML 设计被正确实施和保证源代码的鲁棒通常会需要长时间枯燥的工作。Developer-EP 通过自动生成功能测试用例、执行代码覆盖分析并彻底分析代码的运行时错误使得这一切变得简单，所有这些都通过一个简单的接口提供”。

“Developer-EP 为 UML 开发人员提供了第一个集成的自动化测试环境”，PolySpace 的主管 Chris Hote 介绍，“它基于 PolySpace 和 EmbeddedPlus 的领先技术，帮助软件开发组织减少在测试和调试上的时间消耗。”

Developer-EP 目前适用于用 C++实现 UML 的组织。

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

Borland 倾力连接软件与业务要求

[2004/9/14]

Borland 年度的 BorCon 大会昨天在 San Jose, California 举行，宣布了其在号称软件开发优化（Software Development Optimization，SDO）上未来 12 到 18 个月内的三个里程碑：Themis, Hyperion 和 Prometheus。(译注：Themis：西弥斯(司法律与正义的女神)，Hyperion：亥伯龙神，Prometheus：普罗米修斯(造福于人类的神))

软件产品部的副总裁，Boz Elloy，认为 SDO 是该公司在已有 ALM 策略和产品上的一次革命，它将加速开发进程、降低成本并降低开发风险。

SDO 是为那些传统上位于软件开发过程之外的用户所设计的，例如业务管理人员，以减轻诸如开发中途改变功能给项目带来的预算和发布时间上的冲击。 Borland 把 SDO 比作 ERP，ERP 使用软件给制造过程带来了可信任、可预测和效率。Elloy 认为，即便在软件已经相当广泛的今天，软件的生产过程本身还相当陈旧。

“我们都是鞋匠的孩子（译者注：为他人作嫁衣裳 ；)”，Elloy 这样告诉 BorCon 的代表们。“软件用来提高了各种其它业务过程的生产效率，但却没有为自己的软件开发过程做些什么”。

引用 Standish 集团的数据，Elloy 指出，30%的软件项目都被取消，44%的太昂贵，60%的被认为不成功，90%的延期。

在 Themis, Hyperion 和 Prometheus 这三个里程碑点，Borland 会给 ALM（Application Lifecycle Management，应用声明周期管理）工具集增加基于角色的访问、过程管理入口、业务智能功能（BI，Business Intelligence），这些将给 ALM 过程的各个阶段带来更多的可控性，并提高过程的可见性。

Themis 将是第一个安装版本，它将为分析人员、开发人员以及测试人员提供基于角色的工具访问入口。Themis 预计 2005 年上旬将发布。

在主题演讲之后，Elloy 告诉 Computer Wire，基于角色的访问意味着，拿架构师作比方吧，他会需要 CaliberRM 需求管理工具的一些部分，以及 Together UML 和 JBuilder IDE 的部分。

“你只使用你需要的”，Elloy 这么解释 SDO。

Hyperion 将会为 ALM 项目引入基于入口（portal-based）的访问，提供基于角色的视图和任务提醒，并最终提供更强的可视性和可预测性。

Elloy 说，“在新的工具中，你登录之后会有一个专门为你和你的角色设置的入口，会列出你需要做的工作，你需要的工具和内容会被正确地打开。比如说，如果你要改一个 bug，那么它会在入口处被自动打开，等着你”。

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

征稿

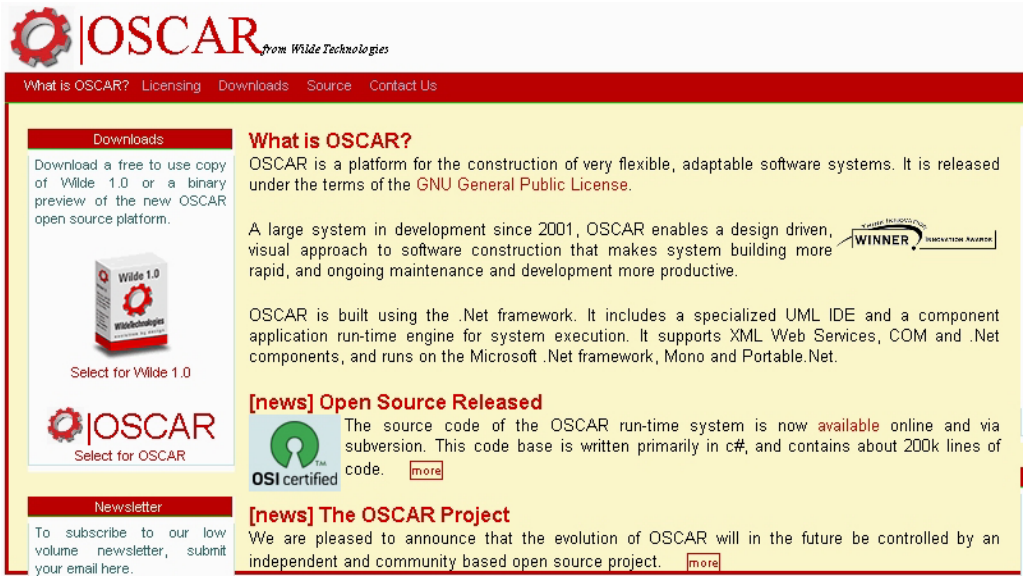
http://www.umlchina.com/xprogrammer/xprogrammer.htm

用 Wilde 来进行软件设计

[2004/8/30]

软件构造的改革家 Wilde 科技公司最近宣布开放其新的架构实现平台的源代码。

这家总部位于都柏林的公司的目标是为希望浏览松散偶合的 IT 架构的架构师、系统工程师以及研究人员提供其 OSCAR(Open Source Component Application Runtime，开发源码构件应用运行时)技术。



行业分析人员认为，在构建更有效率和更有效的企业级 IT 系统方面有着新的可能性和市场潜力。

OSCAR 的独特之处在于应用 UML 来驱动构件软件系统的装配和执行，面向于更灵活和更易于维护的软件系统。

Wilde 支持企业级 IT 系统开发的两个趋势，这两个趋势在最近都非常引人注目。

第一个就是 MDA（模型驱动架构）。它的目的是提供从许多变化的今天主要由业务软件系统来完成的任务导出到可运行的 IT 系统的能力。

传统情况下，软件计划和功能性、面向任务执行的代码结合相当紧密，这一点妨碍了其维护、升级、整合的方便和灵活性。这些工作往往变得困难、昂贵而耗时。Barrett 形容“业务成了 IT 系统的人质”。另外一个趋势是 SOA（面向服务架构），它和 MDA 也是相关的。它的一个基本观点就是，软件不再是以一整个庞然大物的形式提供，过去，功能是封装在“黑盒”中不可访问的而且也不对外界不同任务做出不同的响应，但后者是现在的 IT 系统所要求的。

现在，大多数企业都希望可以通过 IT 系统来满足供应商、分销渠道和顾客之间的无数的交互。SOA 的目标就是要使得软件被作为灵活的功能模块来发布，在需要的时候组合起来执行需要完成的任务，而在下一个新的任务来到的时候，又有新的组合来实现。

Barrett 解释，“OSCAR 可以看做很薄（thin）但是很艰难的一个工作层，它位于 IT 系统的上面。随时都有多个任务作用为计划者、指导（guide）和网站工程师，来提供和复杂度以及变化相适应的灵活性。在我们看来，MDA 和 SOA 都需要大量崭新而富有创造力的输入”。

“OSCAR 为 MDA 的发展做出贡献，并帮助驱动协同的架构。OSCAR 采取的开源策略使得我们拥有了在这些重要领域来自全世界的贡献”。目前，Wilde 已经得到了来自私人投资商和欧洲 R&D grants 的 350 万美元基金，并且赢得了爱尔兰的革新奖。Barrettt 说，OSCAR 在三年开发了 75 万行代码之后选择了开源发布的策略。

“我们支持多种中间件”，基于 GNU GPL License 发布，OSCAR 目前可以通过 Microsoft .NET 在 Windows 上运行，也可以基于 Mono .NET 和 Portable.Net 等在 Unix 上运行。另外，OSCAR 还支持 XML Web Services。

最初三个阶段的目标是基于开源项目开发一个独立的社区，包括马上就可以得到 Wilde 运行时系统，另外，6 周之后还会增加一个功能丰富的 UML 开发环境，并在 2005 年初之前增加对 Java 组件技术如 J2EE 的支持。

组件开发顾问和分析组织 CBDi 论坛对 Wilde 如此评价：“软件开发的圣杯（Holy Grail）在于架构设计和配置的代码之间的联系。但是生成和双向工程总是带来折衷和妥协的要求，而资产管理和重用的解决方案又总是过时，它不是过程之中一个积极的参与者。Wilde 在这方面的解决方案很激进也很独特”，分析专家 Butler Group 认为，“使用 Wilde，企业会得到很多好处，例如节约成本、成功方案的架构级重用以及软件开发的简化”。感兴趣的话，可以在这里得到免费的 OSCAR：www.wildetechnologies.com。

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

UML：大了，小了，还是刚好？

[2004/8/15]

UML 2.0 正在变得过于繁复。建模专家们当然不需要统一 UML 支持者的意见。但是随着微软在提供建模工具集方面消息的刺激，这方面的言论正引起了一场大争论，争论可能会对各方都是有利的：UML 的未来之路究竟该怎么走？怎样进一步推广它？

Cris Kobryn，OMG 的分析设计平台小组副主席，认为，“UML 正在变得肥胖”。PivotPoint 技术公司的 CEO，Kobryn 说，UML2.0 里面有 15 种图，常用的却只有六种：类图、顺序图、用例图、活动图、状态图和组合结构（Composite Structure），“我们要减肥了！”。

IBM 的杰出工程师，OMG 的 UML2.0 底层架构定案小组副主席，Bran Selic 说，“我并不赞同这种说法”。他说，“UML 面对的问题越来越复杂，你无法通过简化事情来解决这些问题”，Selic 认为 UML2.0 也考虑了规模的问题。“我们通过组装 UML2.0 来解决这个问题，它被分解成很多部分，然后组装在一起，它是模块化的。规约包括有很多子语言，如事件驱动的建模和活动建模等，它们是相互独立的，这样 UML 的用户可以只用他选择的那个子语言就可以工作”。

但 Kobryn 看法却不一样，他觉得在实际的项目开发中，UML2.0 工具常常导致混乱。“仔细看看这些图，你会疯掉的。比如说，组件图和组合结构图 90%是相似的”，他说，“从建模的观点来看，为什么需要两个干同样事情的图？应该砍掉其中一个就对了”。

另外，活动和顺序图的子语言也是大体一样的。更复杂的是这两个混在一起，叫做交互图的这个玩意，Kobryn 说，“这个语言太大了，很多人都发现了一些毛病和错误”。

Popkin 公司的奠基人和 CEO，OMG UML2.0 定案小组成员，Jan Popkin 认为，这个说法是否正确还有待考察。如果 Kobryn 的说法－UML2.0 太胖－有水分的话，用户应该让开发商知道。他说，“UML 里面有很多图，你可以发表言论，说它是可以更小的”

谁对谁错并不重要，“他们真正要问的是：UML 的未来将是怎样？”，这是 Popkin 的看法。人们有批评是好事情，这将帮助它进步。Popkin 指出，Kobryn 在批评 UML2.0 太大上有利益考虑，“他说话的时候戴了一顶大帽子”，实际上，Kobryn 的 PivotPoint 公司帮助他们的客户围绕 UML2.0 图的子集来创建模型。

Kobryn 并没有确认是否他的公司在出售基于 UML 子集的产品，但他说，“PivotPoint 对建模语言的快速发展非常感兴趣”。

微软也参与了这场舌战。微软在 5 月宣布了它在 Visual Studio 2005 团队系统方面的计划，其中包括了建模工具集。他们定义自己的建模策略是“UML and more”，并表示，仅基于 UML 的工具无法精确地映射到.NET 框架的 CRL（公共运行时语言）。

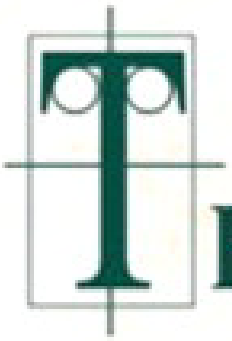
另外，微软还指出，UML 工具集并没有很好地和生命周期开发过程集成在一起。Visual Studio 团队系统的一位架构师，微软的 Jack Greenfield 认为，尽管很多 UML 工具有生成代码的能力，有些还支持双向工程，在今天，UML 更多的用途还是作为一种文档。

“UML 图和软件的创建结合还不够紧密。”而他正在参与开发的 Visual surface 将是一个真正的工具，“它将可以产生我想要的东西：代码、配置文件……”

根据 BZ 上个月对这个观点的调查结果，至少有两种回答，一种：“画画图并不能完成项目”，还有一种是：“简单地画画 UML 并不能直接转换到好的代码”。 调查询问 SD Times 的读者们，在他们的开发团队中是否用到了 UML，如果是，效果如何。58%的回答是在一些项目中有用到。但这些人中，仅有 15%的回答他们在项目的全生命周期中都用到了 UML。

尽管微软的建模计划偏离了 UML，Popkin 还没有看到他们对行业标准的违反。微软是 UML 标准的制订者之一，“微软帮助 UML 变得更有价值，并且还正在帮助 UML 前进，如果他们抛弃了 UML，那就完全是另外一个故事了。”

（自 sdtimes，UMLChina 袁峰 摘译，不得转载用于商业用途）

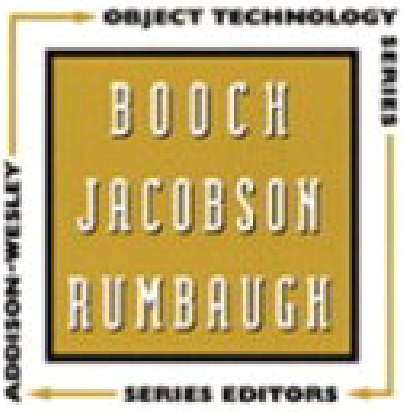


THE UNIFIED MODELING LANGUAGE REFERENCE MANUAL, Second Edition

JAMES RUMBAUGH
IVAR JACOBSON
GRADY BOOCH

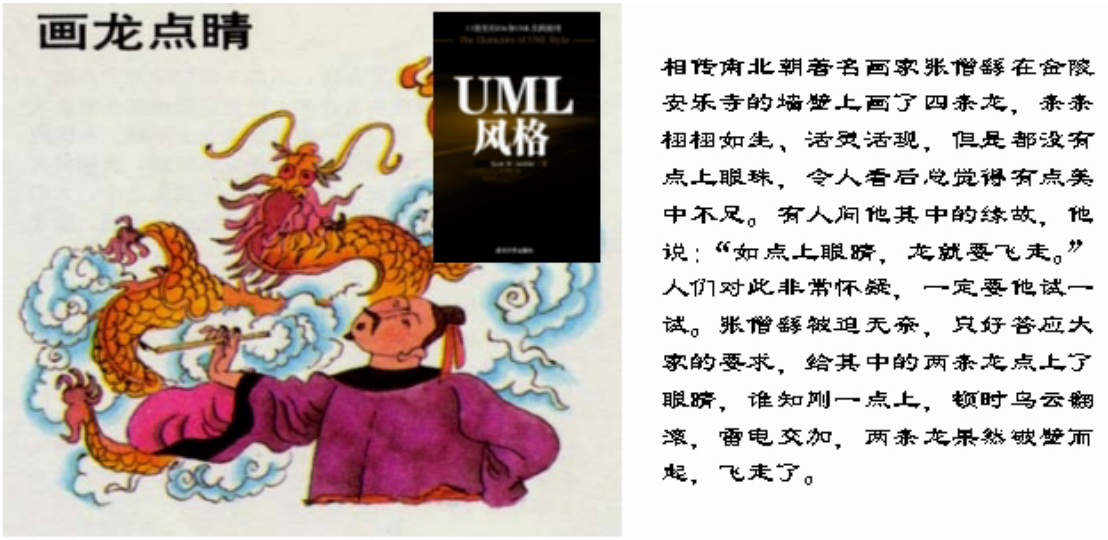
Covers UML 2.0

UMLChina 译
（王海鹏、汪颖）



CD-ROM Included

《UML 参考手册》2.0 版中译本
即将由机械工业出版社出版



它不讲概念，它假设读者已经懂了概念。

它不讲工具，它假设读者已经了解某种工具。

它不讲过程，它假设读者已经了解某种开发过程。

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

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

打开需求之门——在中国实践用例技术的感悟

潘加宇 著

讨论

我这几年的工作主要是为各种软件开发团队作指导，指导团队如何将用例驱动的面向对象方法真真正正应用到团队的当前项目中。贯彻的起点往往是需求技术，即用例技术。在这个过程中收获良多，把一些感悟和思路总结如下与大家共享，比较粗糙，请多指教。

一、用例揭开了疮疤，迫使团队打开需求之门

来自开发团队的声音：**“用了用例技术以后，我不会做需求了！”**。

答案是不留情面的——有可能原本你就不会做需求，或者说你做的“需求”并不是真正的需求。只不过以前这些问题被掩盖起来了，使用用例技术以后被迫暴露出来【1】。使用用例的症状下面隐含着的问题：

执行者搞错——系统的边界不清楚

用例搞错——没有好好思考软件应提供的价值

用例文档不会写——没有考虑过涉众利益，不知道哪些才是真正的需求

开始使用用例技术的团队，经常会经历以下变化：

第一阶段：形式不正确，内容不正确。初次接触用例，团队感到用例“难写”，画的图，写的文档一眼就看出来不正确（明知道不和系统交互也当作执行者、用例羊肉串、明显错误的步骤表达方式...），并不需要从业务上来判断。

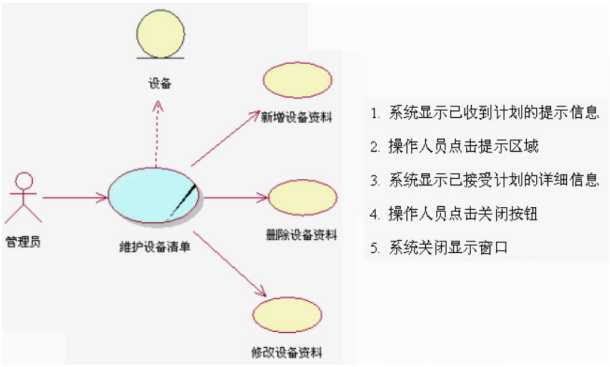


图1 形式上的错误好解决

未经过有效训练的大多数团队会停留在这个阶段，但形式上的错误好解决，只需经过专家指点，团队理解执行。一般来说，很容易改进。

第二阶段：形式基本正确，内容不正确。这个时候团队已经了解了用例的概念，写起来“容易”多了。关起门来写，一天可以写很多个用例。形式上的错误基本上不会犯，但用例的内容十分可疑。

UC1：安排卸货
执行者
操作员
……
涉众利益

（留空）

基本路径
1. 操作员选择一辆货车
2. 系统显示该车上的货柜
3. 操作员为货柜安排搬运组和库位
4. 系统保存安排信息
5. 系统显示当前安排情况

图 2 形式正确，内容可疑

格式上没问题，但涉众利益留空使得需求的内容变得可疑。如果走出办公室，到第一线去和涉众交流，公司总裁可能会告诉你，以前由于安排不当，使公司少赚了不少利润；运输公司可能会告诉你：曾经由于安排不公平，自己的货迟迟不能卸下，耽误了其他运输任务【2】；搬运组工人可能会告诉你，看到别的组老是有任务，我们却闲在一边抽烟。

上面的需求反映出这些涉众利益了吗？看起来没有。按照上面的做法，操作员出于道德操守或技术水平的缺乏，依然会给涉众带来这些伤害。要体现涉众利益，步骤改为“系统自动为货柜合理安排搬运组和库位”，外加业务规则“合理的规则是**********”，这样的需求内容上才有了正确的依据。如果这个需求由于资金（出资人的利益）和技术水平（开发团队的利益）的关系难以实现，次优的选择是在图2的基础上加上另外的需求来约束操作员的行为，例如加上一个步骤“系统记录做出此安排的操作员”。

第三阶段：形式正确，内容正确。这个时候团队已经了解用例只是实事求是地反映需求的一种技术，难度不在于写，而在于写之前的导出工作和写之后的验证工作。又感到用例变“难”了。感到变难是好事！【3】软件开发周期中，需求本来就应该占工作量的一大部分，你感到变难，说明认识到有很多工作要做了，这是值得高兴的。很多时候我们也知道需求应该占大部时间，但分出这段时间后，我们在这段时间里应该做什么？有没有真正把时间利用起来产生真正有价值的需求？很可能没有，缺乏需求技术，在需求阶段“无事可做”或者“无从下手”，往往就使得开发团队有意无意地压缩需求的时间，甚至假“敏捷”的名义直接跳过。正所谓“不是我不明白（需求的重要），这个事太难做！”

二、“直接交流”不是用例当前最重要的价值

经常有团队说，我也知道用例好啊，但客户看不懂用例，甚至都没听说过用例。怎样通过用例和他们交流？也有不少书和文章赞美用例的时候，说用例好啊，一个小人一个圈圈，客户一看就明白。这是比较理想化的想法，换句话说就是不现实的意思。最理想的情况当然是客户自己会写用例；最糟糕的情况是客户不参与，完全由开发人员自己杜撰；最常见的情况是折衷：客户提供素材，开发人员写用例。

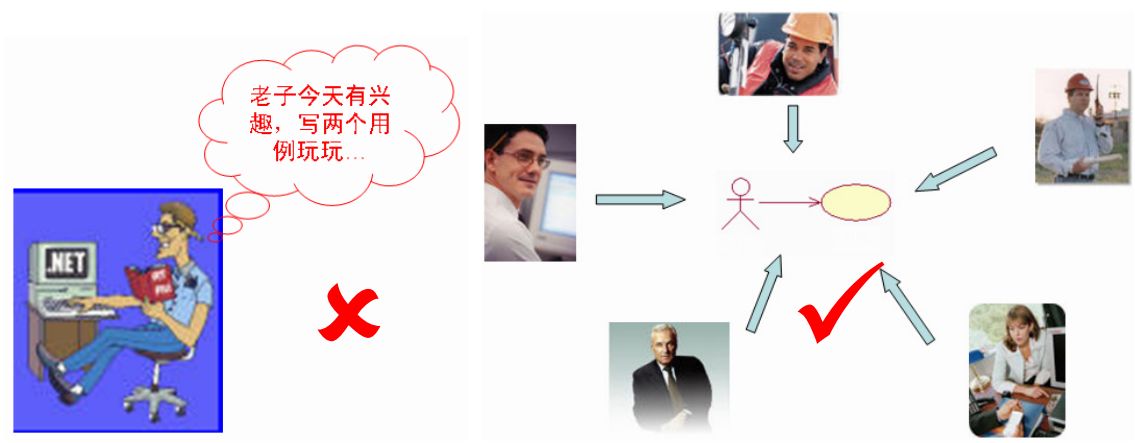


图3 用例不能杜撰，只能从涉众中来

用例不能杜撰，只能从涉众中来，但用例又不能直接从涉众中来。在初级阶段，用例的最重要意义是指导开发团队去向客户探索需求。用例确定的只是交流的**目的**，而不是交流的**手段**。客户并不需要了解执行者、用例这些概念。用例能告诉开发团队“去向客户了解什么”（目的），不能告诉你如何向客户去了解（手段）。把目的当手段，当然行不通。手段可以有很多种，文档研究、问卷调查、访谈、观察、研究竞争对手、开会、原型、场景演示...。总之，用尽一切手段和涉众交流，使用用例思维来指导这些交流手段，会使交流更有目的，更加高效。

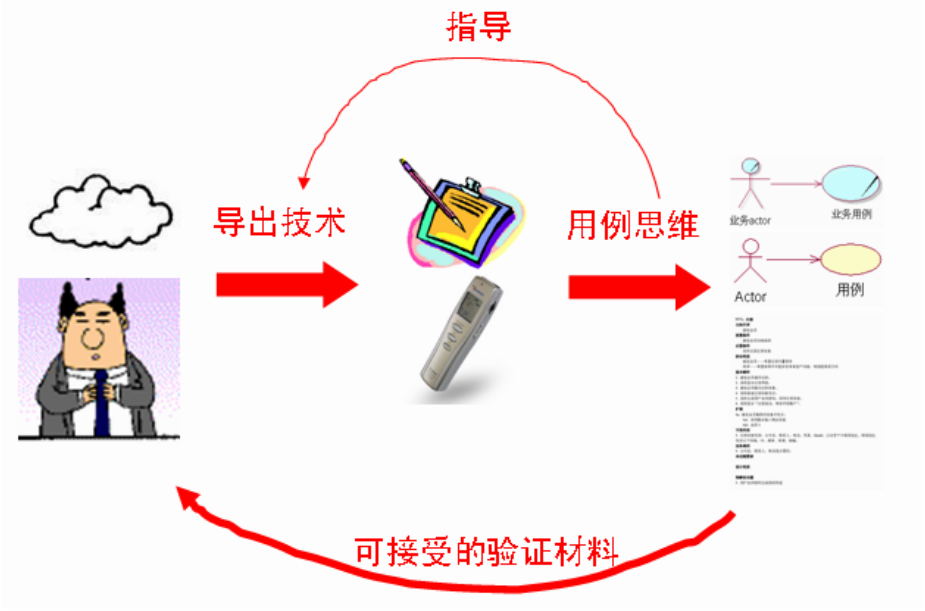


图4 用例不能直接从涉众中来

什么时候才能拿用例文档去和客户交流呢？甚至，客户就能写用例文档？这是一个习惯问题。客户肯定会看某种文档，特别是那种以前司空见惯的形式：一开始先说形势一片大好，然后说一堆我们的系统要体现可靠性、安全性...之类的话，然后再画几个网络图。把这个拿给客户，他觉得他应该看得懂，为什么？因为“周围的人都看，我要是看不懂就太不应该了”。即使这种东西基本上没多少价值。随着用例技术的普遍使用，客户就会形成习惯，“开发团队给我看用例文档，我不看，实在是我的不对”。这个时候就是用例技术能起到“交流”作用的时候了。**以场景方式表达的需求本来就比一条条列出的需求要便于交流。**

三、需要培养专业的需求工程师

我们都经历过各种大小考试，出题人的思维和做题人的思维是不一样的。老师对着大纲苦思冥想出了一道题，我们一会就做完了，一般不会体会到老师出题时的苦心和思路。需求过程就像出题，从各种需求碎片中捏出用例，需要的是合成的思维，而后续的分析设计相当于解决问题，需要的是分解的思维。

例如，需求工程师看到工作人员在填表格，问“为什么你们要填这个表格？”，工作人员回答“这样经理就可以知道所发生的事情”，需求工程师继续问“为什么经理需要知道所发生的事情”？工作人员回答“这样她就可以按需要分配资源”。这个时候需求工程师就能发现“经理应能按需要分配资源”才是真正的价值，解决方案未必是“工作人员填写表格”。

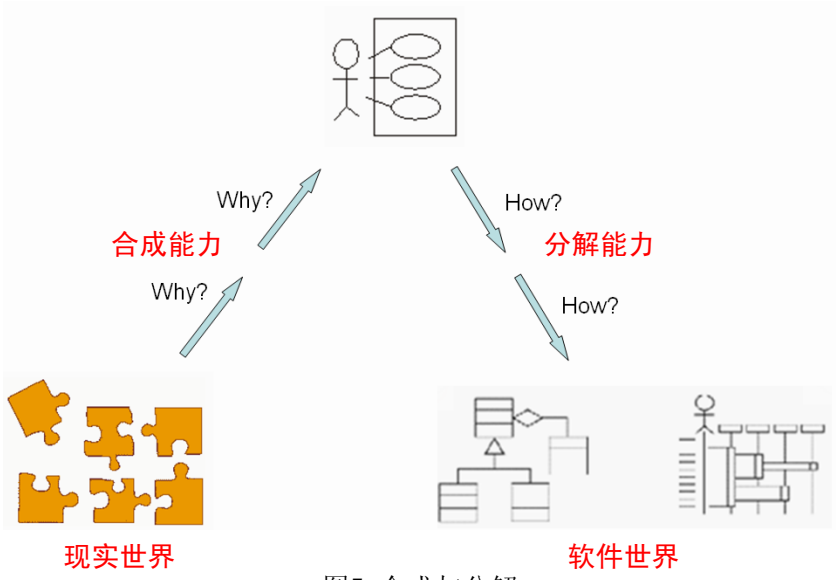


图5 合成与分解

当打的程序员和需求工程师在能力要求上是有区别的。笔者总结如下：

需求工程师需要的能力	程序员经常使用的能力	说明什么问题？
合成能力 （和涉众交流，找出问题）	分解能力 （针对问题，采用某种技术解题）	为什么在我们手里用例的合成会变成功能分解？为什么用例的道理很简单，我们却一次次地摔倒？
和涉众的交流能力	和电脑的交流能力	为什么捕获不到有意义的需求。为什么调研过程总是流于形式？为什么更喜欢在办公室“编”用例，而不是深入第一线？为什么喜欢甲方的信息中心人员，而不是不懂电脑却至关重要的涉众？
自然语言表达能力	编程语言表达能力	为什么寻找问题时总是想到解决方案？为什么习惯于使用技术语言和涉众交流而不是业务语言？

表1 需求工程师 vs. 程序员

由此看来，如果开发团队需要分出角色来，需求工程师这个角色最值得分出来。后面的分析员、设计员、程序员都是解决问题的角色，是否分开则没有那么重要。

一些方法如XP提倡现场客户和程序员坐在一起，这比没有现场客户要好，但存在的问题是一两个人怎么能代表那么种涉众？更不用说不是给人直接使用的系统了【4】。需求工程师应该把捕获需求的责任承担起来，他需要亲临第一线，到涉众那里去，灵活运用各种方法从各类涉众那里得到需求的素材【5】，而不应该对客户有太多的要求，只要客户了解自己的利益，需求工程师就应该有能力完成工作。婴儿也不会直接表达需求，但出色的公司仍然可以探索出“天线宝宝”这样的需求。

四、用例思维对不直接可见系统作用更强烈

有些开发团队做的是与人交互不多的系统（嵌入式开发、电信开发。。。）。笔者和这些团队交流时，他们都会首先表达一个担心：用例技术（包括对象技术）是否适用于他们？有的书上也说：用例只适合和人交互多的系统。笔者认为这是一种误解。正是这样的系统，用例发挥它的价值时，更能形成强烈的对比。

看得见的、给人用的系统，用户摆在那里，交互也比较清晰，需求相对好调研。不使用用例，采用别的方式来捕获和验证需求，很可能也可以达到目的。看不见的系统，需求经常让人感觉无从下手。这正是用例发挥优势的好地方。用例采用拟人化的场景，使“非人”的执行者活起来，在舞台上翩翩起舞，舞台下面有各种观众（涉众），他们的利益正是需求的丰富来源。

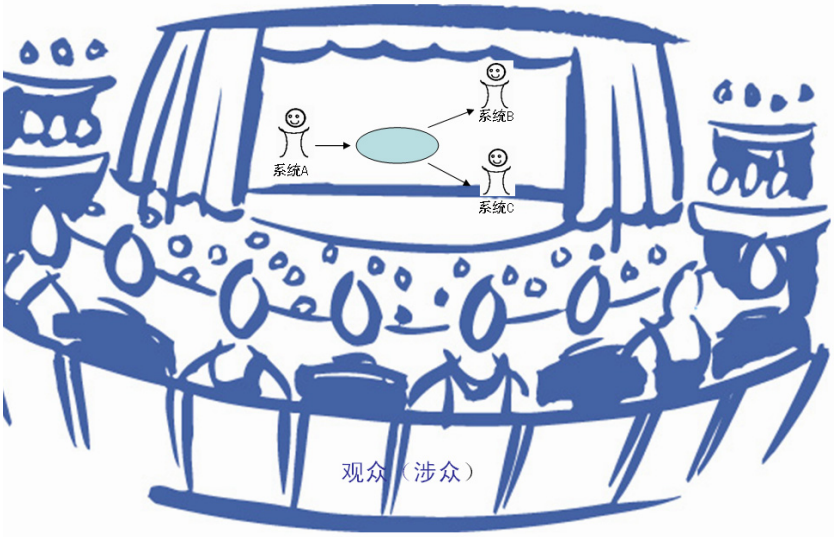


图6 非人执行者像人一样在舞台上翩翩起舞，接受涉众的评判

图6的舞台上，“系统A”向系统发了一个信息，系统请求“系统B”提供一些数据，系统再请求“系统C”做一些计算，然后系统内部做一些计算，把结果返回“系统A”。这个过程并不是枯燥的。“系统A”为什么向系统发信息？它想实现什么价值？要是这个事情搞砸了，哪些人会遭殃，他们为什么会遭殃？通过这样的思考，就可以发现许多需求。

五、用例是朴素的，本应该没有那么多“技巧”

经常有人问：**需求应该写多细才合适？具体一些还是抽象一些？用例粒度该多大才合适？**似乎需求（用例）是面团，关在办公室里面想怎么捏就怎么捏。

容易出现的“技巧”是“复用”用例，例如：

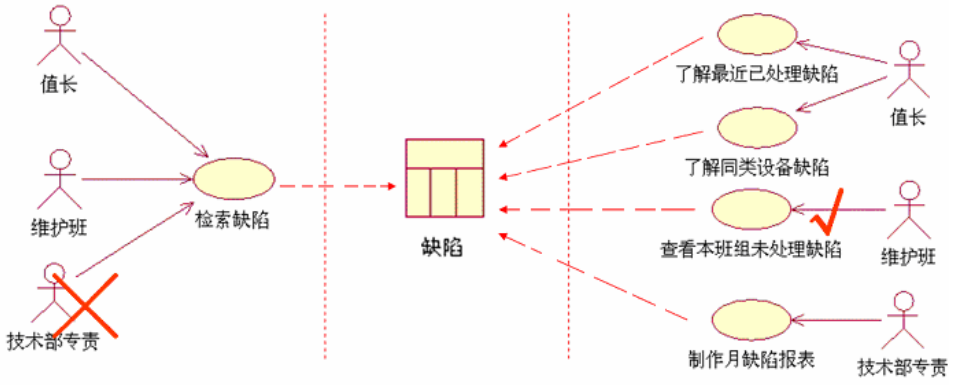


图7 用例和数据表不是一对一的

开发人员认为反正都可以看做“检索缺陷”的一种情况，估计数据库里以后也都是对“缺陷”表的操作，干脆一个用例“检索缺陷”算了。其实，不同的执行者，他们使用系统的目的是不同的，涉众利益也不一样——虽然，在某种解决方案中，很可能实现的时候在数据这一层区别只在于Select语句的Where子句不同。

用例是一种探索真正需求的思考方法，也是如实表达需求的一种组织方式。这里的关键词是：

“执行者”意味着“系统必须要负责和外面这个东西的交互”；“用例”意味着“执行者能够使用系统来实现这个有意义的目标”；用例文档里面的各种需求都意味着“系统必须满足这一点，否则某类涉众的正当利益会被侵害”。不断询问，从涉众的角度来锤炼，使需求逐步逼近问题核心。

图8 不断思考，寻找最佳答案

既然用例是实事求是反应需求，在用例的“粒度”问题【6】上，可以讨价还价的余地很小。是不是用例？几个用例合适？答案只能从涉众那里探索出来。需求人员要学会用一颗返璞归真的赤子之心去探索需求。

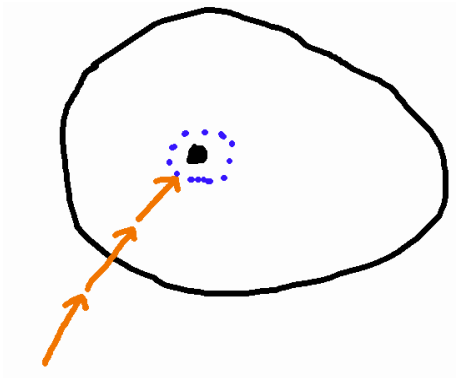


图9 探索需求——增之一分太赤，减之一分太白

用例技术的采用应该增强了团队和涉众的交流，否则，你可能误用了用例。这是判断用例技术应用成功还是失败的标志。

六、用例工具急需改善

《程序员》2004年第7期中有笔者翻译的《对用例工具的期望》，作者已经在这方面做出了很好的阐述。笔者认为，该文还有需要补充的重要之处。该文中更多讨论的是用例级别的“管理”，在最令人头痛的书写用例内容方面却讨论不多。用例不只是小人和圈圈，笔者尝试用类图描绘用例的内容：

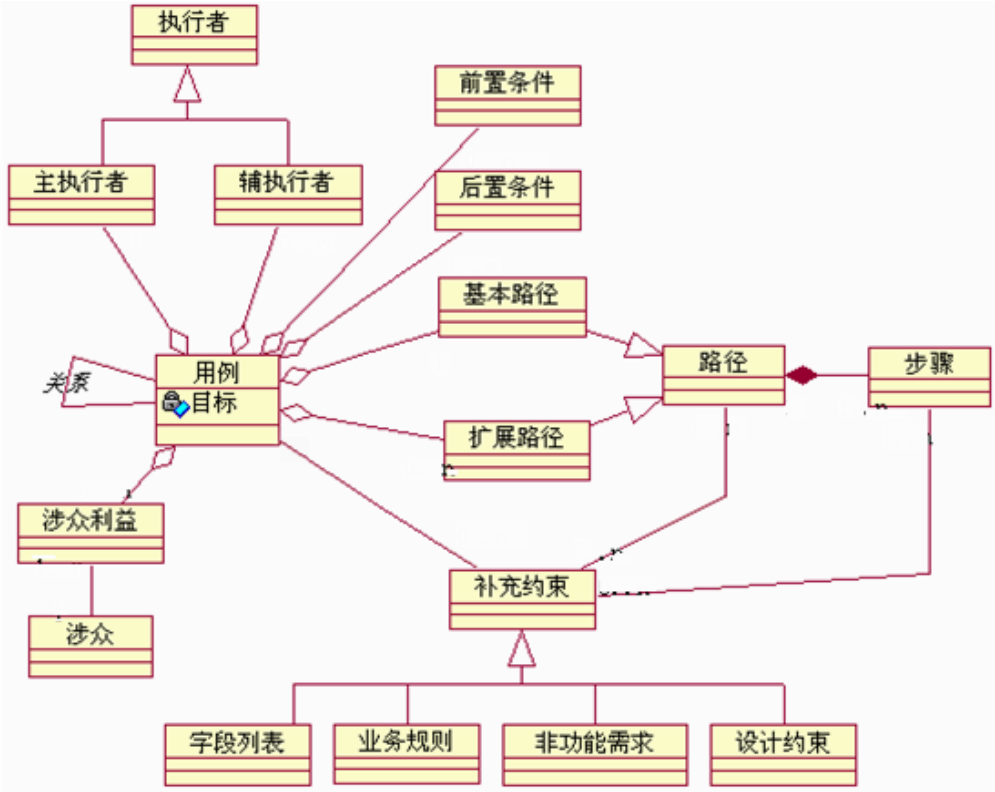


图10 用例不只是小人圈圈

这些就是用例的结构，不管它是以图形方式表示（目前UML并未覆盖全部），还是保存在数据库中，还是用Word写出来，都需要包括这些内容。目前的现实中，一般用Word等文本处理器来书写用例，而用例的各项内容之间是有关联的，扁平的文本形式难以高效地建立、修改和维护这种关联。应该有一种用例书写（未必是书写）工具，能够以一种“立体”的方式来“书写”和保存用例，同时允许需求工程师以文本、图形、表格等各种视图查看或输出用例。

补注

【1】也意味着，如果需求技术过关，转到用例并不是一个难题，或者是不是使用用例这种方式也不是一个难题。

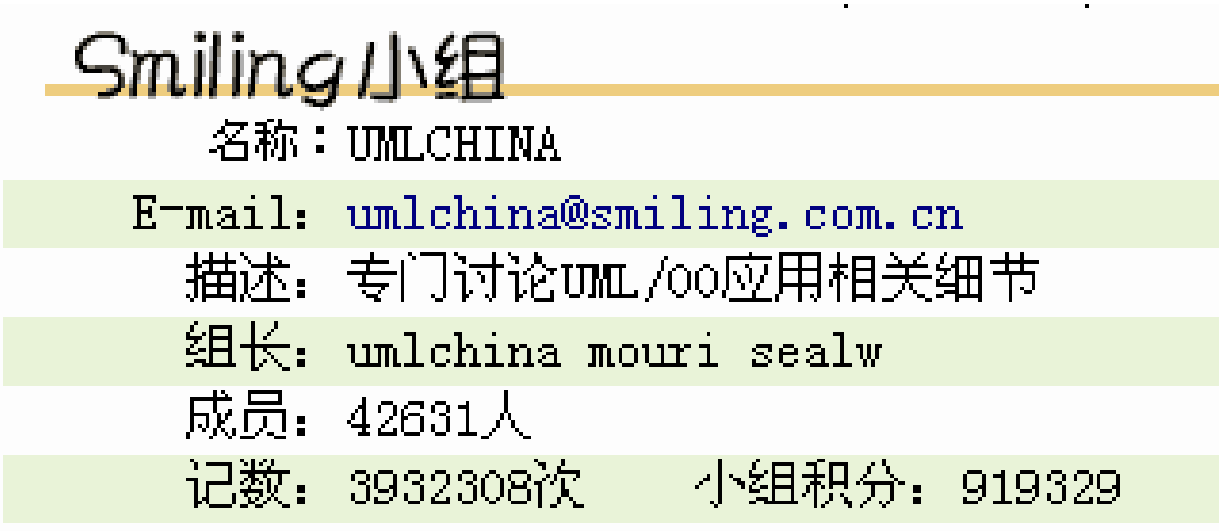
【2】这还可能变成行贿的刺激源。

【3】同理，职业选手的训练和比赛比业余爱好者更“难”，要考虑的问题更多、更细。

【4】“客户”这个词也存在一定的模糊，是领域专家、直接用户还是非用户涉众？

【5】“亲临第一线”，到涉众的工作环境中去捕获需求，比让涉众移步到开发人员的办公室更加合适。被访者在他熟悉的工作环境中，会想起许多工作中的烦恼和不便之处。关于需求导出的技术，我们会在以后的文章里讨论。

【6】有关“粒度”，讨论的最多的是CRUD问题，是4个？还是1个？还有误用“扩展”、“包含”关系引起的“粒度”问题（其实是错误）。这些笔者将在后面的文章讨论。用例关系是用来整理用例文档的，没有文档，关系无从谈起。没有关系，也并不影响用例的内容。关于用例关系，很好的参考资料是《有效用例模式》一书的第7章。



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

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

从基本用例到对象

Robert Biddle James Noble 著，李胜利 译

讨论

从基本用例到对象

Robert Biddle, James Noble, Ewan Tempero / 文

产生核心用例的一个主要动机是用户界面设计的上下文关系。然而在通常的面向对象系统开发中我们仍然在探索核心用例的应用。本文概述了两个涉及核心用例的技术，在需求分析中角色脚本的使用和从核心用例到对象的系统实现的分等。

导言

用例的创始和使用现在已经成为软件开发中的常见惯例。并且用例是一个在抽象语言和开发过程中被认可的概念。基本用例作为一组概念的精炼的简介，最初的目的是为满足在用户界面设计中表达技术独立的需要。我们感到基本用例值得作为一个更广泛的角色，并且在一般的面向对象软件开发中都已在探索基本用例的应用。

为一般系统开发考虑基本用例的动机起始于基本用例一般简单的优点，特别是其抽象使得过早考虑界面细节的危险变。并且使之更为简洁。当我们在开发各类系统应用基本用例时，我们逐渐认识到许多探索的益处。

我们开始使用基本用例作为我们首选的需求收集工具，并且已经在最近三年多的时间里使用这种方法完成了许多系统开发工程。我们认为基本用例普遍适用于面向对象软件开发，并且很大地优于常规用例。

在本文中，我们简要回顾基本用例的本性，然后讨论两个我们正在探讨的新技术。第一个技术涉及需求分析，并且描述基本用例在然支持的用于用例评审的角色作用的使用。第二个技术涉及设计，并且描述由基本用例分解标出的系统职责分开，生成一个组成可跟踪需求的面向对象设计所涉及的对象和他们责任的集合。

背景：基本用例

用例的一般想法是描绘系统(即使尚未实现)和系统外部实体可能的交互的序列。用例在此序中的定义是"对特定的角色来说，系统执行产生一个可观察的结果值的动作包括变量的集合和序列描述"。用例的定义对整个系统开发过程都是有益处的，从早期的分析阶段开始得经测试和维护阶段。

基本用例是用法中心设计方法的一部分，是由 Larry Constantine和Lucy Lockwood开发的，Constantine和Lockwood支持用例，并且深度关于用例优点的许多主张，他们同时也看到了用例的局限性。"在特定情况下，对于仍在设计的用户界面简单，一般用例再型地包含太多的内在假设，它们经常是隐藏的或含蓄的，"无论因为过早做出界面设计决策，还是因为嵌入到这些需求中的决策，这对于用户界面设计来说都是有问题的，使得到后来基于很难或适应。

基本用例被设计来克服这些问题，这个术语"基本"是指基本模式"是通过非技术的，甚然化的和抽象的描述来规程问题本质"，Constantine和Lockwood定义的基本用例如下：

基本用例是一个结构性的叙述，使用应用域和用户语言来表达的，组成一个目标或定义的简单的、一般化的、抽象的、非技术的和与实现无关的描述，从用户的观点来看，它是个角色或与系统相关的一致整合，这个目标或定义是必要的，有意义的和明确定义的，并且使构成定义作用的基础的目标和意思具体化。

基本用例以代表用户和系统互对话框的格式被书写，它类似于Web-Brock1993年所使用的两列格式，在这种格式里，列标题的是动作和响应，尽管Web-Brock讨论过用例中抽象层次框架的问题，然而她提出的这种两列用例包含了用户和系统交互的具体步骤。

相反，基本用例格式标出了用户意图与系统职责列，这份新标准显示基本用例如何通过无需描述用户接口细节而允许交互支持化来实现独立。注意这种抽象并不满足与用例作为一个整体相关，而是更多与用例步骤相关。

详见《程序员》杂志 2004 年第 10 期

敏捷 MDA

Stephen J. Mellor 著，汪磊 译

讨论

模型驱动架构是包含多种不同模型驱动途径的开发方法。更一般地说，人们考虑的模型是充满了代码的蓝图。模型驱动开发能够自动转换这这些模型。也就是说，MDA 被看作是一种支持重量级过程的模型技术，但是，却能做得比重量级过程模型更好。

敏捷模型驱动架构是基于这样的概念：代码和可执行模型在操作上是一样的。因此，敏捷联盟定义的主要原则首先就是测试，直接执行，在一个小的周期内沿着链完成分析到执行的过程，例如可以同样应用到建模。一个可执行的建模，因为其可执行，所以在一个增量的迭代周期里可以构造，运行，测试和修改。

为了达到这个恰当的状态，模型必须足够完全以致能够独立执行。不存在分析或设计模型，因为所有的模型都是均等的。模型被连接在一起，而不是转换，接下来就是映射成一个简单的联合模型，根据简单系统架构原则转化成代码。这种方法就称为敏捷模型驱动架构体系。

什么是 MDA?

对一些人来说，将敏捷和建模放在同一个句子里的概念是毫无意义的。建模者担心敏捷这个词在最坏意义上被认为是骇客的同义词，而敏捷建模者看到的是笨拙重量级过程（也许是笨拙的重量级方法学家）花费大代价交付的错误系统。

这种分离性的原因之一是对确认隔阂的公认，而隔阂来自于当我们写下文档而不能执行的时候。当然，我们可以评审并对其正确性下结论，但是直到我们执行否则我们无法确切知道它们所做的是否真正被需要。另外，在这段时间里它也传达了一个信息，市场和技术已经改变，即使它是正确的却不相关，导致了一个过时的系统。更糟糕的是，有些很狡猾的系统，存在对改变问题的解决方案，里面设置了完整详尽而无效的说明文档。

敏捷方法提出尽可能快的移交工作代码的小片断来解决这个问题。这个工作功能直接用于客户，能提高需要建立系统的相互理解和作用。因为这个交付周期可能很短（一周或两周），所以系统开发过程可以根据条件的改变只移交用户想要的需求。

为了提高用户的参与性，敏捷过程鼓励用户参与甚至在编程的阶段，但是没有建议用户帮助写汇编代码，因为它的抽象性太低了。当然，用户并不愚蠢：他们完全可以学会写汇编代码，但是这种涉及到寄存器地址和堆处理的语言与他们关心的银行业，电话，复印机或者其他他们能运用的东西相去甚远。

Java, Smalltalk 和 C++都是高级语言，但是他们还是无法引起用户的兴趣和关注：表结构，分布策略，和全局调用的美妙等等。为了减少确认隔阂和直接交付系统片断，我们需要的是一个高度抽象的能够集中表示简单主题的建模语言，而无论用户是否对这些主题感兴趣，但是这种语言必须是足够精确和具体而能被执行建立模型。这个可执行模型的语言是比 Java, Smalltalk 更高级的抽象语言，就像 Java, Smalltalk 比 C 或者汇编语言更高级一样。

这个可执行模型既不是草图也不是蓝图，向他的名字一样：运行模型。这个事实减少了确认隔阂，允许我们以小的增量与用户直接交流并交付给运行的系统上。从这个意义上讲，可执行模型担当了代码的角色，尽管他们提供了与用户领域更好的交互能力。它是独立于平台的，用 MDA 的说法就是与平台无关（PIM）。

建立可执行 UML 模型

可执行 UML 是 UML 的一个 profile，定义了一个谨慎选择的最新的 UML 子集的执行语义。这个子集完全是可计算的，所以一个可执行 UML 模型可以直接执行。建模规则不是坚持惯例而是可执行：任意的模型都能被编译和运行，或者相反。

所有的表格（例如，类表，状态表，过程说明）都是映射或者下层模型的视图。不支持执行的 UML 模型，例如使用用例表，也许可以自由地用来帮助建立可执行的 UML 模型。

图 1 说明了可执行 UML 的主要组成和一组使用状态机通信的类和对象。每个状态机都被一组状态变化动作触发来执行同步，数据存取和功能计算。

一组完全的动作使 UML 成为计算说明语言，这种语言能够为创造对象定义抽象语法，发送信号，访问关于实例的数据和执行一般的计算。动作语言能够用具体的语法为表达这些计算提供符号。UML 完全是计算的，它作为 PIM 可以在系统中具体化任何主题事件。

UML 动作语言和普通乏味的程序语言的不同类似于汇编语言和程序语言的不同。他们都能具体完成要做的工作，但是它们是在不同的语言抽象水平上完成的。程序语言概括了硬件平台的细节，所以你可以写出你需要做什么而无需担心目标机器上寄存器的数量，栈的结构或者参数是如何传递给函数等等问题。相对的，动作语言概括了软件平台的细节，所以你不用担心分类策略，列表结构，远程调用等。例如，动作语言不关心数据结构的集的势，把一切都视为组。只有当这些结构执行的时候，我们才选择合适的物理数据结构。类似的，只有在检查了这种访问类型是否真正能被使用后我们才会选择合适的数据访问模式。

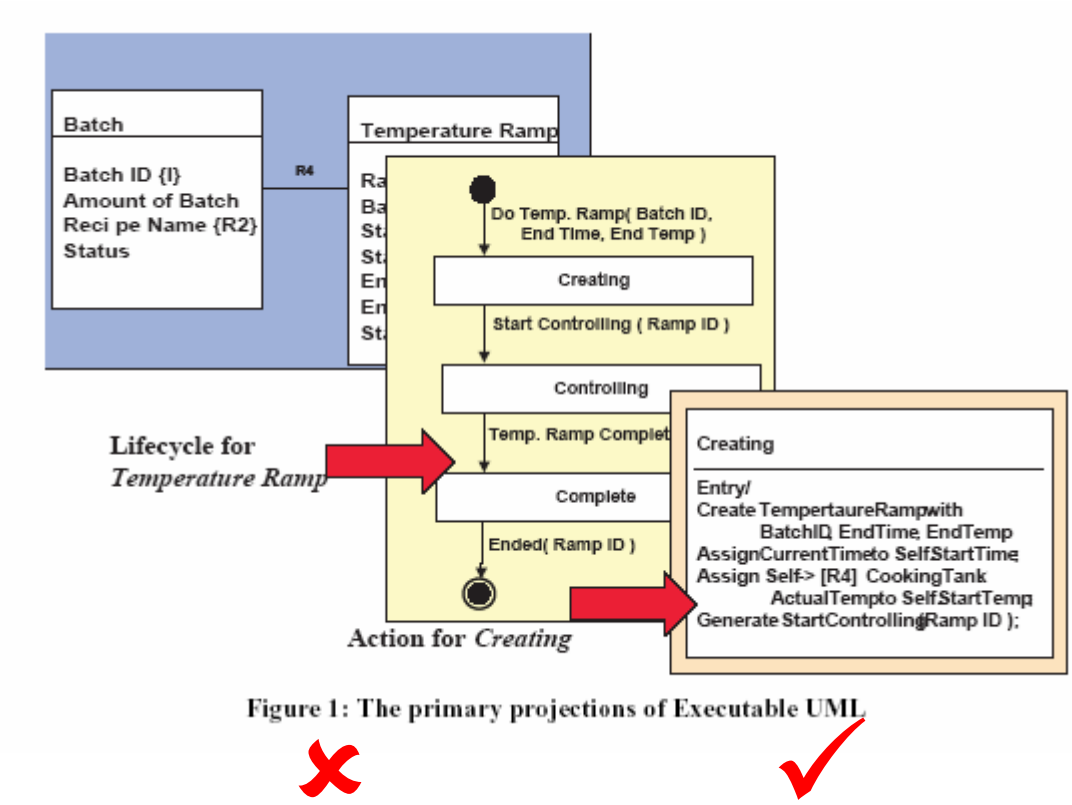


Figure 1: The primary projections of Executable UML

图 1：可执行 UML 的主要映射

就像存在使程序移植于多种硬件平台标准一样，可执行 UML 也存在使模型移植于多种软件平台的标准。

执行可执行模型

图 1 表示了可执行模型的静态结构，但是一种语言只有在赋予了可执行语义才有意义。而可执行 UML 具有特别明确的语义。关于在运行时语言如何执行动态动作的规则用一组通信状态机术语表达的，通信状态机是可执行 UML 程序的唯一活性元素。

每个对象和类（潜在的）都有捕获各自行为的实时状态机。状态机在一个时刻仅有一种状态，所有的状态机根据彼此的状态同步执行。

过程里的动作可以同时执行访问数据和对象，除非被其他的数据或控制流所限制。建模者的任务就是指定正确的顺序和保证对象数据的一致性。

状态机通过信号同步行为，这个信号被接受器的状态机解释为事件。接受到信号的状态机激发跃迁和执行过程，但是在处理下一事件之前，这组动作必须运行完成。

状态机仅靠信号通信，信号指令在发送器和接受器间的一对实例中保存。规则就是声明想得到活动的顺序。当事件在接受器引起跃迁，接受器的目的状态过程就在发送信号后执行。这获取了系统动作里需要的结果。至于保证信号次序正确、连接成功等问题，就像在并行系统里保证指令的执行顺序一样，完全是个别的问题。

代码与动作

为什么使用动作模型而不写代码？因为定义的动作语义在转换的时候不改变数据结构，不影响关键可转换性需求的计算精确度。所以动作语义允许你在不依赖实现知识的基础上定义详细的行为。例如，找到最后十个处理的总数的公共方法是通过数据结构创建总数进行循环。这种方法不可避免地涉及到数据结构的计算，但是如果结构改变了会发生什么？动作语义用这样的方法解决：取得最后十个处理数量，然后累计。这个小变化的结果在视图看可能是存储安排的改变，不会影响到算法，因为仅仅是个累加值而已。

总之，可执行 UML 模型能够转换成任何目标。

因此，任意模型都包含支持它执行，测试、确认和实现独立性的必要细节。没有为执行模型设计开发或增加所需的细节或代码；所以执行正式的测试用例和模型相反，为了确认需求已经被恰当的处理了。这种测试形式可以在每个可执行模型上进行，并对立于其他的模型。

这就是它们的规则，但是真正发生的是可执行 UML 是一种一致性的规范语言。关于同步和对象数据一致性的规则是那种语言的简单规则，就像在 C++里我们执行一个接一个的声明和数据一次访问一个声明一样。我们在这种一致性语言里详细说明这样我们就能把它解释到协作的分布式平台上，也就是同步的单任务环境。

转换模型

可执行 UML 定义数据和行为（类）的分组，实例的过去行为（状态图）和为准确计算的行为（动作）。括号是因为 UML 的不指定实现。甚至，在可执行的 UML 内的类描述可能被作为类实现的数据和行为的概念的组，它可能作为 C 结构体和一组相关函数或者 VHDL 实体来实现。所以类不一定非要作为类实现。可执行的 UML 是可以被转化为任何目标的一种独立软件平台的语言。因此我们也可以使用可翻译和执行的命令。可执行 UML 允许开发者模拟主要问题的基础语义学而无需担心例如处理器数目，数据结构组织或者线程数。换句话说，就像程序语言独立于硬件平台，而可执行 UML 独立于软件平台能轻便地应用于不同的开发环境。

模型 模型 模型

模型至少有三方面的意义，每个方面都指出了不同的用法和包含的不用过程。一是指所谓的模型就是草图。我们概略地描述出啤酒垫后面的翅膀形状，用一些线条表示空气流动，一个或两个等式描述如何互相作用。草图既不完全，也不确定。它的目的就是提炼出一个想法,不能维持或交付使用。

敏捷代表者愿意草拟出他们的类和用例，有时候叫做故事，或许甚至用 UML 去做。这里不存在对抗：甚至最极端的用法也就是为代码勾画出设计的要点。

第二是指作为蓝图的模型。在风洞中的飞机的物理模型就是一例；更一般的，我们可以把蓝图想象成为建立实物所需的关键属性的文档描述：蓝图就是建筑计划的体现。

但是蓝图模型的内涵却引起了冲突：它激起工厂和制造业的想象和枯燥无味的目标。在一个环境里，80%的对象建模需求可以用 20%的符号设计来满足，就像制造业。把蓝图看作将要运行构造的预言性的计划。

重量级过程具有足够的蓝图模型思想；例如，在软件工程协会的成熟度模型（CMM）里反复提到了制造业类推问题。但是我们知道软件是创造性新经济体，根本不像过时的制造业。相反地，软件的关键是创造性，80%的设计需要 20%的构造。这种情况下，开发者需要适应的不是和其他模型关联的预言性计划，而是把模型用作蓝图的挫败。

三是指可执行模型。飞机的模型能够转化为实物。转化需要其他的输入：飞机需要金属板、螺丝和螺旋桨组成机身，并且模型在与塑造实体相关问题的各个方面的细节都是完全的。当我们建立了可执行 UML 模型，就已经描述了系统的行为就像我们写 Java 程序时一样。在这种可执行模型的解释下，用高级语言 Java 写的程序也可以说是一个模型。Java 程序也可以转换成实物（字节代码）。模型的建造者（程序员）在这样的情况下，无须知道 java 编译器如何工作，也无须知道编译器运行程序都做了什么。当然，编译器在一个抽象层次产生的字节代码本身是一个能够用 0、1 代替的模型，并且 0、1 代码在更抽象的层次上定义了硬件想要的行为。

XP（极限编程）和 AM(敏捷联盟)的许多原则都涉及到过程和用户之间的关系和他们的管理而不是代码。例如，敏捷过程的代码构造原则的应用与可执行模型的构造一样。那些特别提及的代码和软件原则对于可执行 UML 的定义就是代码。

在系统构造阶段，概念对象映射成线程和处理机。产生器的工作就维持列入应用模型清单的先后顺序，但是它可以选择分配对象并对其序列化，甚至冗余地复制或分离它们，只要定义的行为是被保护的。

一个 C++程序被完成并执行，但它不能很好地作我们想要的事情直到它转化成直接解释的语言。因此我们通过一系列转换来运行程序，这些转换保存了程序的语义内容（否则在编译器中将会有错误）但是程序的表达是用一种更面向实现的语言。

同样的会发生在当我们建立一个完全的可执行模型时。当我们转换一个模型时，工具为现在用的建模语言组装元模型。为了使我们上面的语言实例达到极限，我们使用 C++程序来组建模型实例，包括类，保护成员，静态成员函数等。下一次转换的结果，是我们有一个 C 的模型，在这里静态成员函数和保护成员将被派成普通函数，尽管有不同的标记。

这样的转换可能持续到它们最后，最低的元模型。在元模型（汇编语言）里的类可能是指令，寄存器，内存地址等等。元模型的实例包括高级模型的所有信息，但在一个低层次的主题抽象上。选择最低层的元模型能产生 C++、Java、C，汇编代码或甚至微代码。最后的层次就是变成在正文表里编译的可以被模型编译器执行的更低级的语言。

融合模型

要产生一个系统，除了使模型完善的连续转换，模型还需要与其他模型编织在一起。在一个升降机系统里，升降机模型可作为可执行模型,但不能解决建造问题直到它与另一个运输主题的问题模型连接在一起。连接后转换成代码，可执行模型就变成了系统。

为了使结合有效，我们定义一个映射函数。（映射通常是融合或表示映射，而不是把一个模型从一个表转换到另一个表的精炼映射。）特别地，我们需要建立两个模型之间的通信机制。一个实例就是银行系统中的出纳类和用户类都能和安全模型的任务实例通信。另一个例子就是每个帐目实例与保护资源实例安全地通信。或控制应用里的训练实例的子集与用户界面范围的图标实例通信。（后两个叫做相似物或副本实例）

这些映射可以在可执行 UML 模型的任意可确认实体间执行。例如，停留在 Train Control 域的状态与 Icon.color 属性的 Red 值通信。这将扩展到动态过程。一个信号就是一个域,也就是说,按下 UserInterface 的按钮，将使 TrainControl、TimeToLeaveStation 的信号与 Train47 的信号通信。信号可以映射为函数,或者相反,函数转换成映射以适应 values.Anything 属性的变化。

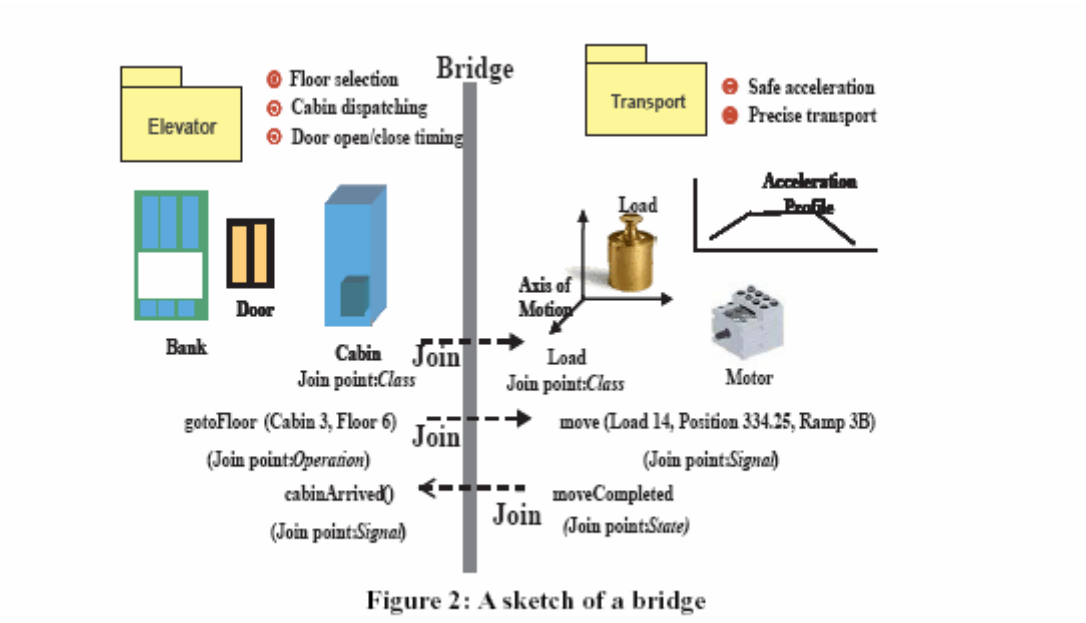


Figure 2: A sketch of a bridge

图 2 桥接略图

映射常常受到与源模型和目标模型相关元素的标记的指示。在 Mellor 和 Balcer[1]的早期著作里，这些相互性（映射函数和标记）称为桥接。

一旦定义了映射我们就准备把所有的模型结合成单一的组装的元模型，并从它产生代码。（这些步骤可以立刻发生）可依赖的机制是编译器，它可以遵循一组简单的架构规则把几个模型编织起来。

模型编译器

模型编译器执行一组可执行 UML 模型并根据一系列规则把他们编织在一起。任务涉及到执行多种源和目标模型的映射来产生一个围绕系统所有结构，行为和逻辑等每一件事情的原型模型。

存在几种方法实现原型模型的最终映射。其中之一是使用原型定义映射函数，特别适合操作文本。

立即把模型编织在一起,涉及到结构失配问题,这个术语是 David Garlan 杜撰的,指那些在不增加通道和粘连代码通道就不能结合的组件,而这正是 MDA 特意避免的问题！模型编译器把一个架构结构作为整体施于系统上。

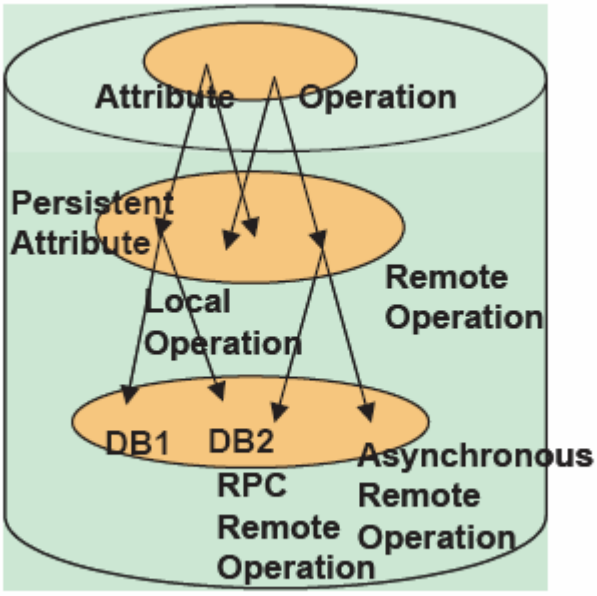


Figure 3 A silo of successive transformations

图 3 连续转换情节之一

添加代码主体

可执行模型并不与代码完全一样，因为他们需要和其他模型编织成一个系统。而且，因为每个模型都是独立的，一旦编织完，系统才算完成。

建立系统的另一种方法是复制 PIM （Platform Independent Model）结构到 PSM(Platform Specific Model)，然后添加代码主体。（显然，这些代码主体应该被保护，因为模型的改动或重建都可能导致代码的消失。）PSM 也可以先用图形模型表示再加入代码细节或结构重组。可能存在几个转换，而在每次转换时都要添加代码。代码在适当的时间和适当的抽象层次上加入，这么做据说是有好处的。

我们也可以通过它对连续转换的依赖区别这种 MDA。这种典型情节用来说明比较低级抽象语言的轮廓，如图 3 所示的椭圆型。例如，UML 的子集可以作为无需全局或局部对象知识的分析模型，另外更大的方面，它可以作为可获取知识的设计模型。然后模型又被转换成更抽象的语言，或许与平台无关，例如 CORBA（Common Object Request Broker Architecture）.

然而这种代码主体天生依赖于与代码密切相关的模型结构。例如，任何为目标元模型写的代码都假设远程过程调用只能应用在使用远程过程调用的环境里，甚至被模型捕获的主题正在倾斜。产生这样的结果因为模型不是通用的，但是代替模型语言（元模型）和主题结合的图表即将到来。这有另一种结构失配结构表。敏捷 MDA 通过平等对待所有模型和立即融合它们来明确解决这个问题。

显然，这种模型不能执行。事实上，这些模型就是重量级过程的自动操作。不能藐视那个用途，因为一些人不得不使用，但它决不是敏捷的！

可执行 MDA 方法就是建立可执行转换模型，它在转换过程中无需更多的干涉和详细的细节就能够使模型通过互连变得完善。

在敏捷 MDA 中，因为可执行模型是与平台无关的并且模型编译器把模型转换成代码，你可能想知道在平台相关模型上发生了什么？平台相关模型在这里已经转换成了代码。代码是 PIM 的元素与所需平台的组合并且能执行。在敏捷 MDA 中，无需非要利用 PSM 或者使它作为模型具体化。我们直接就可以运用 PSM：代码。

总结

我们认识到建模者和敏捷程序员之间的冲突是基本和广泛的，一方面因为不同的技术焦点不同，对预设的流程反应激烈，一部分原因是因为各自的夸大。然而，在现实中，这种隔阂是很小的。敏捷联盟和极限编程的许多思想同时控制着模型，是否我们简单地用可执行模型代替文字编码。无论如何，你的作者已经成为敏捷宣言的签名者就足够了。

为了用敏捷建立系统，应遵循以下步骤。建立测试用例和可执行模型，用模型编译器编译，运行测试用例，增量地移交系统片断给用户。这就和我们建立测试用例，写代码，用编译器编译代码，运行测试用例，增量地移交系统片断给用户一样，除了我们用另一种更高级的抽象语言（一个可执行和转移的模型）代替这里的语言（代码）。

敏捷 MDA 要求使用称为可执行 UML 的不同平台无关模型构造，应用这些使用模型编译器编译出来的模型执行函数映射产生出最有趣和有用的 PSM：代码。

参考文献

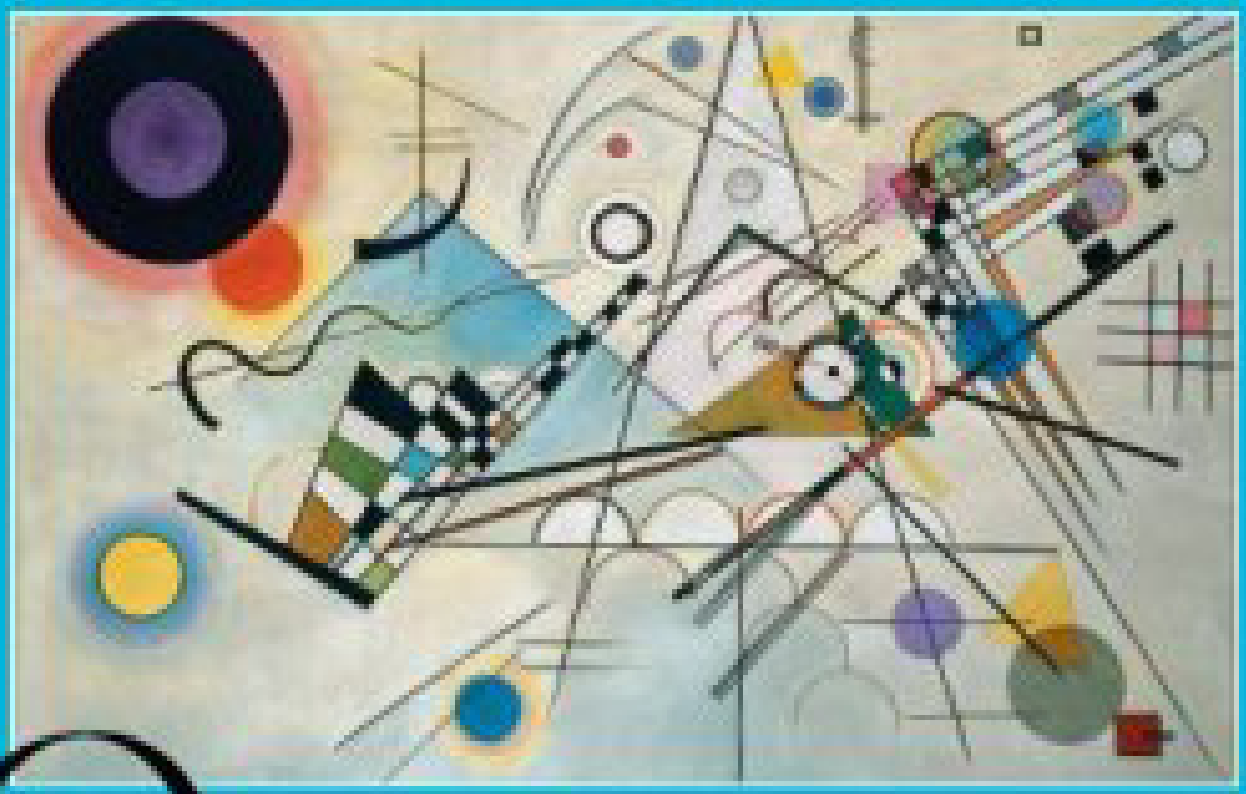
[1] For more information on this usage for bridges, see Executable UML: A Foundation for Model-Driven Architecture, Mellor and Balcer, Addison-Wesley, 2003, Chapter 17: Multiple Domains, and Chapter 18: Model Compilers.

[2] For a description of MDA in general, see MDA Distilled: Principles of Model-Driven Development , Mellor, Scott, Uhl and Weise., to be published in March 2004. This paper was derived from Chapter 10 and from [3].

[3] Agile MDA , Mellor and Wolfe, 2005, Addison-Wesley Figure 2 was conceived by Leon Starr, Model Integration, LLC. San Francisco, CA. 2. For information on the Agile Alliance, see their website www.aanpo.org. The best-known agile method is XP, for which see Extreme Programming Explained by Kent Beck.

Domain-Driven DESIGN

Tackling Complexity in the Heart of Software



Eric Evans

Foreword by Martin Fowler

《领域驱动设计》中译本

即将由清华大学出版社出版，UMLChina 审稿

软件工程技术丛书

设计系列

企业应用架构模式

Patterns of Enterprise Application Architecture

(美) Martin Fowler 著

王怀民 周斌 译

UMLChina 审校

UMLChina 训练辅助教材

WILEY

TIMELY. PRACTICAL. RELIABLE.

UMLChina 指定教材

Agile Database Techniques

Effective Strategies for the Agile Software Developer

《敏捷数据》

UMLChina 李巍 译

机械工业出版社即将出版

Scott Ambler

使用分析模式的软件重用

Andreas Geyer-Schulz、Michael Hahsler 著，刘杰 译

讨论

摘要

本文通过介绍软件生命周期中的分析阶段所采用的各种模式来促进问题域的重用。我们为分析模式的描述建立了一个模板，这一模板能够有力的支持整个分析过程，从需求分析到分析模型，甚至包括由分析模型向灵活的，可重用的设计和执行方案的转化。在这篇文章中，作为例子，我们将提出一类分析模式，来解决协同工作，协同信息过滤与共享，以及知识管理过程中所存在的一系列亟待解决的问题。我们评估这些模式的重用能力的方法是对某信息系统中的若干组件进行分析，该信息系统是为维也纳经济和工商管理大学的虚拟大学工程而开发的。分析结果表明，在分析阶段使用各种模式，采用已经应用于分析阶段的重用，改进分析和设计阶段之间接口，就可以节省大量的开发时间。

引言

目前，Internet仍然以指数态势增长，伴随着这一增长过程，人们对用于存储、组织和显示信息的信息系统的需求也增加了。网络时代，向信息社会转型的时期，技术变化速度之快向系统开发者提出了巨大的挑战，要求他们在越来越短的时间内开发出高质量的符合顾客个性需要的信息系统。每个系统从构思到实现，需求的增加并不意味着系统的强壮，而且，也不能确保要求的质量（软件质量的相关细节，参见Gillies 1992）。代码以库和组件的方式重用，通过框架和设计模式进行重用设计，都成为了应对这一挑战的手段。最近，有关软件重用的研究重点已经转到从领域分析(Neighbors 1984)而来的问题域(Moore 2001)。从问题域中提取出共同的元素使得非代码工件的重用有其可行性，从而使我们能够以更快的速度投放市场，开发出更多的可靠灵活的系统。非代码工件包括说明、用例、分析模型和设计。在这篇文章中，我们将提出和评估一些分析模型，并将它们作为推动问题域重用的方法。

（译注：artifact（工件）一条信息 (1) 由流程生成、修改或使用；(2) 定义职责范围；(3) 受到版本控制。工件可由软件开发流程所生成或使用的一条信息。工件可以是模型、说明或软件。同义词：产品 (product)。）

文章的结构如下：首先，简短的说明我们对分析模型的总体看法，并提出一个适用于对分析模型进行描述的模板结构。然后，列举一些协同工作和信息共享中所用到的分析模型的例子，这些例子都是我们在 1997 至 2001 年之间所进行的一系列虚拟大学工程中所开发的。最后，我们将对两个信息系统进行分析，它们都是以本文中所提到的分析模型的实现为基础建立起来的。通过这些例子，我们将分析问题域重用所引发的代码重用，并采用Boehm提出的著名的 COCOMO 模型（COnstructive COst Model）对开发时间和精力的减少程度进行评估。

分析模式

分析模式这个术语是Martin Fowler（1997）为了描述一种模式而提出的，这种模式用于从应用域中抽象出概念模型，这种抽象工作使重用得以在整个应用软件中实现。与设计模式相比，分析模式侧重于系统的组织结构、内部关系和经济因素三个方面，这三个方面对于需求分析以及系统最终得到认可并投入使用来说是很重要的。

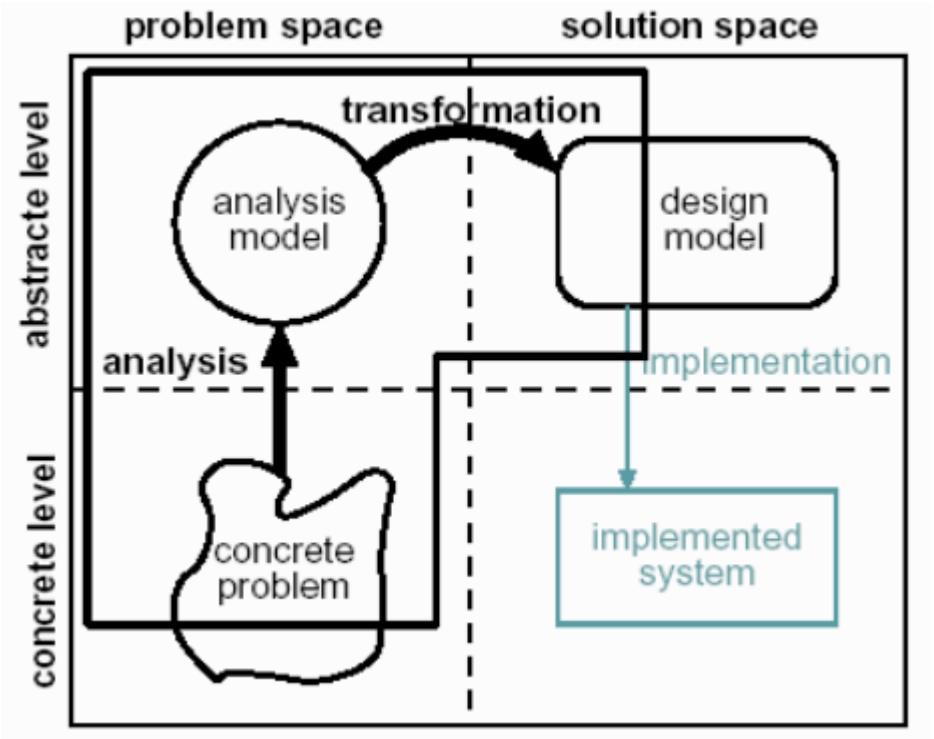


图1 软件开发过程中的分析模式

图1显示了在软件开发过程中分析模式发挥作用的两个主要的工作部分。第一，分析模式提供了可重用的分析模型及实例，并对这些模型的优点和局限性进行了描述，这些都有利于加速抽象分析模型的提取，抽象分析模型用于从具体问题中抽取主要需求项。第二，分析模式使分析模型向设计模型的转化更易于实现。就像Champeaux et al. （1992）和Kaindl（1999）所描述的那样，由问题领域向解决方案领域的转化是一个复杂和费时的过程。

分析模式支持这一转化过程的方式是，提出设计模式和可行的解决方案，为重现问题领域结构而服务。

与Fowler（1997）不同的是，他采用的是一种自由的，非正式的方式来描述分析模式，而我们采用的则是一种统一的，前后一致的方式。Vlissides（1998）和其他作者都强调，依照某种结构来描述模式是非常重要的，因为，对结构化信息来说，一旦其组织方式被理解，它就会易于被传授、学习、比较、描述和使用。表格1介绍了我们为描述分析模式而提出的模板，本文后面部分中的四个模式实例都使用了这一模板。

模式名称(Gamma et al. 1995；Buschmann et al.1996)：模式的名称准确的描述了它的基本内容，成为分析过程中所使用的词汇的一部分。 **目的**：分析模式将要做什么，它所要解决的问题是什么？ **动机**：阐明目标问题的场景，并分析模式在具体场景中起作用的方式。 **力量因素和环境**：对需要由分析模式解决的forces and tensions进行讨论。 **解决方案**：对于有关动机部分所提到的场景，描述由分析模式得到的解决方案，以及各种力量因素之间的平衡。包括分析模式中所有相互关联的结构和行为侧重面。 **结果**：模式如何达到其目的，以及存在的折中。 **设计**：分析模式如何由设计模式来实现，设计建议示例。 **已知应用**：在现实系统中应用的模式实例。

表1：分析模式的模板

然而，请注意，我们所提出的这种描述方式保持了模式典型的环境/问题/解决方案结构。

在模板中，除了关键力量因素、解决方案和设计三个部分，其余各个部分均取自于（Gamma等人，1995），依照分析阶段的特点，我们对它们的涵义作了些微的改动，这些都在表1中做出了说明。

有关关键力量因素的部分是 Alexander 模式的核心内容，软件工程对模式的应用即可追溯至此。这一部分用于讨论需要由分析模式解决的 forces and tensions，包括社会和经济的冲突。

模式中解决方案部分的观点来自于（Buschmann et al. ，1996），它说明了解决问题的方法，以及借助模式来达到各方力量平衡的方式。它都是由图表构成的，这些图表用图形的方式描述了模式中相互关联的结构和行为侧重面，我们使用的是 UML 符号。

分析模式的设计部分讨论分析模式实现的可能性，同时给出一个或多个设计模式。这一部分的目的是，通过为分析模式提供设计实例，从而减少软件开发时间。最好的情况是，通过重用提出的设计解决方案，设计阶段可以跳过去。

一般来说，在模式的使用中需要解决的一个关键问题就是，要求开发人员和模式的用户共同使用一个通用的词汇表，这一词汇表是实现有效沟通，使各方都能够理解有关分析模式的目的的先决条件。然而，有关这个问题的研究并不在本篇文章的研究范围之内，我们这样处理这一问题：第一，我们召开 intensive 的开发小组项目会议，建立一个通用词汇表，并达成共识。在这里，我们依靠组织动力学来消除有关分析模式真正含义的误解。第二，UML 作为一种标记来使用时，我们过于回复到某种程度上更规范的语言，然而，有必要注意的是：这要求那些正式的说明语言要能够被作者与用户所理解。这一需求不能够被认为是理所当然的，因为，知识传播所需要的时间要被预想的长的多。

实例：通过活动代理实现普通插接板向虚拟库的进化

（译注：pinboard是电力上的专有名词，意为插接板，它的构造、用途等是理解下面部分内容的前提。可惜的是，翻译这篇文章的人暂时无法做到这一点。既然标题上实现进化过程的方式是活动代理，但是，整篇文章的后半部分都没有提到活动代理，很奇怪！）

接下来，我们将阐述普通插接板向虚拟库进化的基本原理。这一进化过程的目的是实现交易成本的减少以及可测量性和性能的改进。首先，考虑一下普通的插接板，普通的一组插接板在通过的讯息量保持较小的情况下，能够为小型工作组充当高效率的非正式信息通道。一旦工作组成员的人数或是讯息量增加，他们就需要花费越来越多的时间来理解插接板，相应地，要想快速而及时地找到有用的信息就必须花费更多的时间和精力。普通插接板向结构化插接板的进化能够帮助处理这些测量性问题，这一进化的实现主要通过扩展表现接口和讯息表示，为结构化讯息提供环境。这些改进使得用户可以将注意力仅仅放在插接板入口的小部件上，而且可以极大的提高搜索能力。

然而，值得付出的前提条件是：对信息结构的合理使用，要求，要么用户对他所要填写的表格的各个字段的含义十分清楚，要么必须有一个专业的信息管理员对讯息进行分类。

总之，这种插接板的管理成本会随着它自身的扩展和时间的推移而增加。

另外，通常情况下，用插接板来处理分散型信息对象是很有效的，例如，当一个信息对象的所有者喜欢亲历亲为，当所有者经常改变其信息对象，或者当一个分布式对象群能够在网络交通拥挤的情况下提供更好的负载平衡，或是以更窄的网络带宽提供更好的性能。

所有这些因素都能够减少一个组织的网络构建成本。显然，这些都会推动进化过程向下一步发展。

简单插接板的分析模式

目的

一个分散型团体如何实现有效的共享信息？

动机

如果你在一个跨国公司工作，这个公司有好多分散在不同国家的工作组，每个组有一定的成员。这些工作组需要共同工作来完成他们的工作，因此有效的沟通和信息共享对他们来说是十分必要的。所有的工作组成员经常性的碰头是不可能的，因为这太过费时。一个工作组需要交换信息，异步通信，建立起一个共用的知识基础。更进一步来说，为了避免信息过载，工作组中的成员需要一种方便快捷的方式，来检索到服务于某一具体任务的信息。遗憾的是，现存的通讯媒介如信件、电话、传真、电子邮件等，都不能满足他们的所有需求。一个类似于插接板的系统（如图2所示）就是用来为工作组服务的。所有的成员都能够阅读和组合讯息。所以，每个工作组成员都可以在需要的时候使用信息。

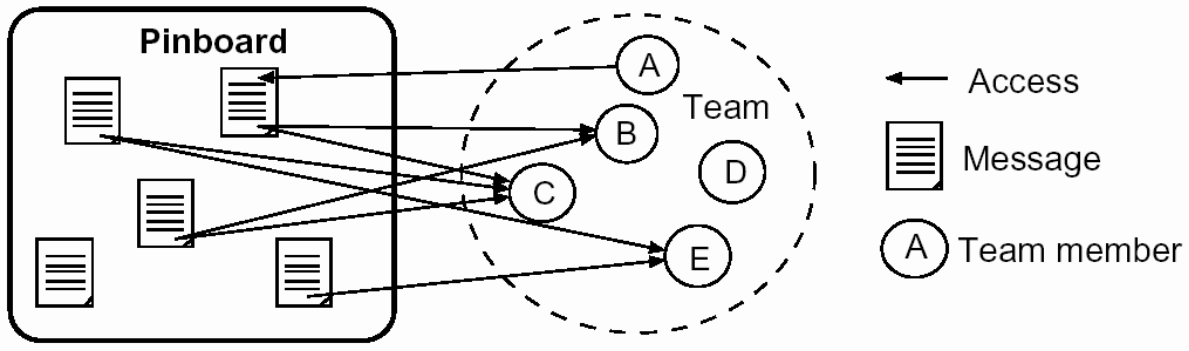


图2 为团队内部通讯服务的简单插接板

关键力量因素

有效的组内交流是至关重要的。

频繁的碰头会议是不可行的。

信息过载的风险时可见的。

解决方案

用插接板来解决组内交流的问题。他从所有的小组成员那里收集讯息并进行存储，并使这些信息们能够被访问。对于普通的插接板来说，一条信息的内容仅仅包含文字描述和作者姓名。要使工作组成员能够选择所需信息，就应该提供对所有信息的全文检索（例如，搜索从某一特定成员那里搜集过来的信息，或是搜索包含某一具体关键词的信息）。表3表明了插接板有三个主要部分构成，用于存储和检索信息的数据库，用户访问的接口，负责管理整个插接板的控制单元。

使用系统的人员包括：

1、 用户，通过插接板进行信息检索。

2、 信息提供者，向插接板传递新的信息。

3、 管理者，负责管理插接板，如删除数据库中过时的信息。

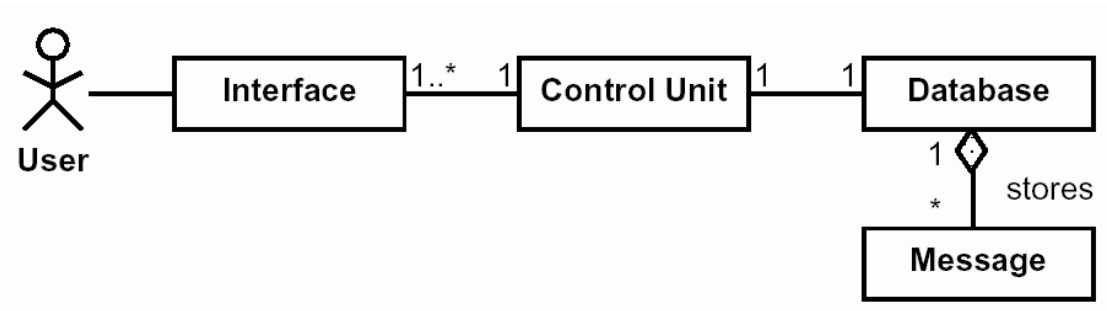


图3 简单插接板结构

结果

插接板模式的优点和缺点在于：

1、减少了信息过载 用户能够主动地查询其所需要的信息。然而，如果某一讯息需要被迅速传递给某一具体人员，问题就会出现，（插接板模式是解决不了这个问题的）。这种情况下，就要采用其他通讯方法，如电子邮件。

2、异步通信 工作不会因为接收电子邮件或是接听电话而中断。

3、协同信息源的自动产生。收集讯息并存档，从而建立一个集体的动态数据库（dynabase）。

4、机密信息 如果通过插接板来实现机密信息的共享，就应该有足够的安全警惕意识。

5、讯息寿命 一条讯息何时过时，谁来删除这一讯息？一个实用的解决方法就是自动保存那些超过规定期限的讯息。

设计

插接板看起来很容易实现，但我们仍然需要考虑一些问题来确保可重用性和可扩展性。

1、系统三个组成部分的划分

对于类似于插接板这种简单的系统来说，许多人会倾向于在一个模块中实现所有的部分，这样可以减少执行的时间，因为不需要考虑各个组成部分之间的接口，但随后，一旦系统发生改变，代价就会很大。例如，只有在接口部分适合的情况下，从私有客户机到基于网络的接口的改变，才可能会是易于实现的。

2、控制单元的可扩展性

要确保能够向控制单元容易地添加新的功能，最好的选择就是采用interpreter模式加上可扩展的指令系统作为控制单元的执行方式。

3、数据库的选择

实现信息存储有很多方式，从数据库管理系统（面向对象型的和关系型的）到使用简单的文件系统，采用哪种存储方式要依据检索性能，license成本和执行的难易程度。然而，如果数据库部分与系统的其他部分分离，随后采用Façade模式实现变更是可行的。

4、接口的选择

这里，我们需要在为私有客户机设置的接口或是标准客户机，如网络浏览器之间进行选择，网络技术最大的优点在于允许客户机对其进行访问。一般来说，这些客户机依据每个新的操作系统实现分类和预设。选择私有客户机的唯一原因是出于安全方面的考虑，但是内置的安全特征（如Netscape的Secure Socket Layer）可以不断的改进，我们甚至可以实现私有接口，如，如果我们将Java applet的客户端与网络浏览器配合使用的话。

已知的应用

新闻网络传输协议

电子公告牌（如，为群件应用服务的，像 BSCW 系统）

网站的留言板

结构化插接板的分析模式

目的

对结构化信息进行收集，分类，显示。

动机

工作组中对简单的文本讯息的收集，并不能使人们可以在要求的时间内查找到所需要的信息。这不仅取决于工作组所需解决的不同论题的数量，而且取决于工作组的规模，因此也取决于所提供的信息的数量。使用户能够迅速判断出某一信息是否包含其所需要的内容，是十分重要的。同样，依照不同的标准（如一些分类方式）对信息进行分组，能够使查找相关信息变得更加容易。简单的插接板仅仅支持非结构化的文本，但是不同的人们可能会使用不同的术语来描述同一个概念。因此，所有的信息检索的问题就浮现出来了。我们使用一个支持结构化信息的插接板来解决这一问题。另外，对于文本信息，你可以对某个主题使用不同的查询域（查询范围），关键词和若干预先确定的类别（如，信息的语言，或者是某个特殊的分类表）。预先确定的格式可以被用来对信息进行组合。

关键力量因素

对拥有大量讯息和主题的大型工作组来说，简单的插接板是满足不了其需求。阅读非结构化的文本会花费更多的时间。对非结构化的信息的阅读是很难的，对其进行自动处理也是很难的。人们在组合一个新信息时会忘记（原来讯息的）一些重要的特征。

解决方案

建立处理结构化信息的插接板。如果使用能够提供若干半结构化讯息类型的系统，智能信息共享是可能实现的（Malone et al. 1987）。图4表明了通过对若干对象域中的信息对象进行组合，可以得到结构化信息。为了提供一个更加灵活的接口，接口使用若干子类来表示所陈述的问题，这些子类使用策略或是模板模式。当图4中的接口A是一个简单的搜索接口时，接口B就会提供相应的访问，如，一个等级分类树，类似于NAICS为业界提供的等级分类树。

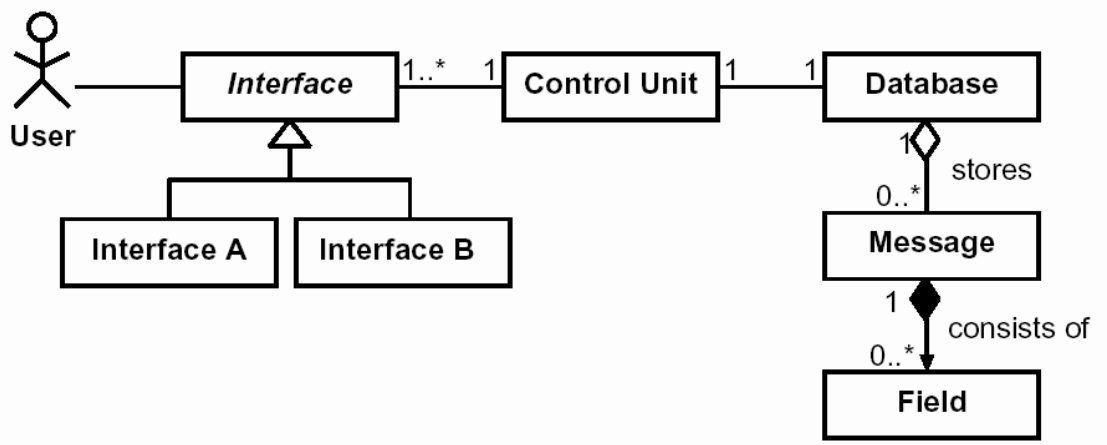


图4 结构化插接板的结构

结果

结构化插接板模式的优点和缺点包括：

1. 结构 相比较于阅读非结构化文本来说，用逻辑结构表示的信息更易于被理解，而且花费的时间也更短。

2. 自动处理 使用内容被限制在一个预先确定的取值范围中的域是可行的。有了这些域，就可以实现自动处理，而不需要复杂的信息组织技术。

3. 信息的完整性 向用户提供包含所有预先确定并予以标识的域的表格，可以避免用户遗漏重要的信息。

4. 不同类型讯息之间的不相容性 使用固定格式来组合信息的缺点在于需要预先定义好所有可能用到的域。用户在写入信息时，那些可能改进信息的可读性的新的域是不能被实时加进来并提供使用的。而且，如果提供一些由不同逻辑结构构成的不同的格式，人们需要找到一个可以解决同质异形的信息类型的方法。

设计

除了简单的插接板所涉及的那些问题之外，下面的问题也很重要：

1、讯息表格的发展 表格的执行要求对所需信息进行详尽的分析。表格的使用方法必须直观，而不需要去阅读使用手册，否则，用户会花费一些时间去思考某一信息需要被归入哪一个域。如果某些域易于被人们误解，应该在表格中对其所指做出解释。

2、接口的可扩展性 采用结构化讯息，就有必要对接口做出较大的改进。如，可以仅用主题或作者的方式来标识讯息，因此，在一个画面上就可以显示更多的讯息。我们还可以作出的改进包括，提供仅在某一具体的域中进行检索的接口，或是提供可以在某一个域中限定仅仅检索包含某一具体的值的讯息的接口（如，在包含讯息日期的域中检索某一关键词时，可以依据日期限制仅仅检索那些在某一周之内的讯息）

已知应用

普通的插接板模式是结构化插接板在某一个具体的域中以文本形式表示讯息的一个特例。因此，所有普通插接板的应用同样可以被结构化插接板所识别。

Internet上的代理服务（如，求职代理的在线服务）

分析模式：虚拟图书馆

目的

虚拟库是用于对分布式信息源提供简单的接口。

动机

在大学中，多数研究和教学材料都在线提供，但是大多数都分散在整个学校网络上，因此查找是很困难的。一般来说，一些材料可以在学院的主页或是讲师的个人网页上访问到。遗憾的是，这些材料通常都在网页的深层，或是无法链接到。即使大多数大学在其网站上都提供搜索引擎，以某种组织方式显示所有的材料，这可以改进材料的访问方式，从而改进网站的使用并提高其价值。

解决分散型信息的一个通用的方法就是将他们集中，比如，集中到一个数字图书馆中。数字图书馆是储存和检索文档的空间，然而，采用数字图书馆可能会引发许多问题，甚至失败，原因如下：

1、 那些经常变化的文档（时事问题演讲材料），一旦其发生变化，就会产生额外的管理费用，用于在中央存储中更新文档。

2、 在数字图书馆中对多媒体文档进行保存并保持其功能性是困难的（如，使用服务器程序可视化显示用户所操作的模型的演讲材料）。

3、 在包含多个相互独立单元的组织中，很难使所有人相信，对每一个组织成员，集中存储信息是能够起到促进作用的，而且，它不仅仅是一个集中力量的方式。

我们采用如图5所示的中心目录，这一目录在校园网（或Internet上）为教学科研材料和信息源提供了统一的访问接口。这一目录包括附加在元信息上的电子索引卡片（如，信息对象的名称，作者，关键词，描述等（参见Weibel et al. 1998），作为网络上的标准元数据），还包括对他们所提供的实际的信息实体的引用（如超链接）。

使用不同的结构化插接板讯息域来存储引用和与它相关联的元信息。一个重要的问题是只有当被引用的信息实体可被访问时，引用才能是有效的。因此，建立一种机制，确保经常检查所有的引用以达到一致性，是十分必要的。

关键力量因素

需要建立一个中央接口，指向分散性信息

有着不同目的，使用不同技术的，相互独立的信息提供者

对所提供信息保持较高的变更率

解决方案

虚拟图书馆使用的是结构化插接板的结构。插接板的讯息被用于存储有关信息实体，以及这些信息实体的引用的元信息。

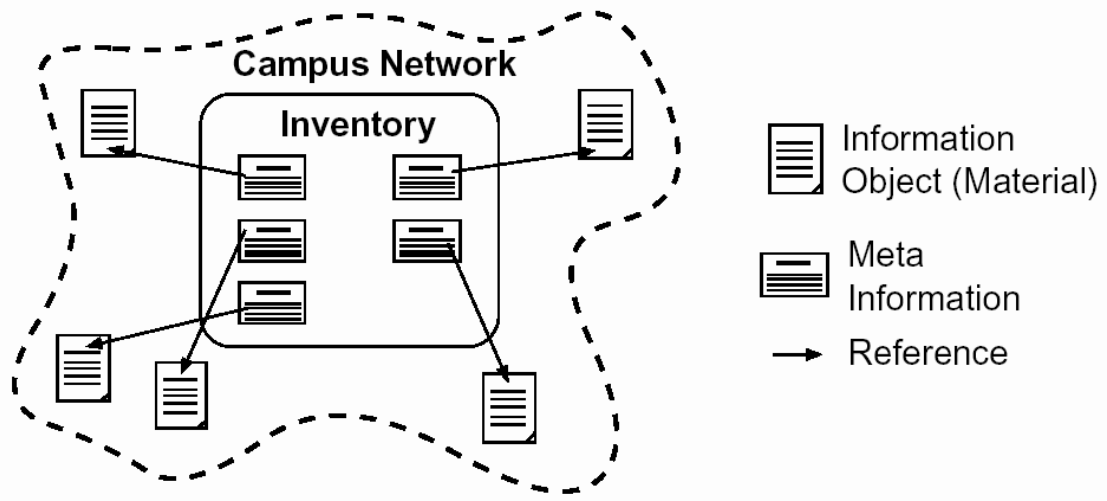


图5 虚拟图书馆

结果

除了已知的结构化插接板模式的优点和缺点以外，虚拟图书馆模式有着以下结果：

1、 信息实体在物理上没有与其所有者分开 因为只有元信息与引用被中央存储，实体的所有者可以无限制地对它进行操作。如果实体需要频繁的进行更新这种方式就具有很大的优势。

2、 一致性问题 信息实体可以被删除或是移动到其他位置，而不用通知中央词典。因此，必须经常检查所有的引用和元信息，以保持系统的一致性。另外，修正无效的引用是一项花费很大的工作。

设计

虚拟图书馆使用的是有着可扩展性控制单元的结构化插接板。因此，无需进行更改而添加新的功能（如，一致性检查，其他管理功能）是可能实现的。对信息实体的引用的一致性检查，以及其他附加功能（如自动关键词抽取），通过使用visitor模式（Gamma等人，1995）都是可以实现的，如图6所示。

已知应用

基于目录的服务（例如 WWW虚拟图书馆，Yahoo!）

开发的虚拟图书馆系统和虚拟大学项目的现场课程(http://vu.wu-wien.ac.at/virlib/)

分析模式的效益

我们使用本文所描述的分析模式，为维也纳经济和工商管理大学的虚拟大学项目，实现了多个信息系统。第一，我们在编程语言PERL和HTML中采用了虚拟图书馆分析模式，然后，我们重用了所开发的软件为虚拟大学项目中的不同任务建立了多个信息系统。

为了量化我们所使用的分析模式的效益，我们列举了两个重用模式执行项目中的信息系统，并将执行这两个信息系统所实际需要的花费，和估计的用于从头构造这些系统的花费进行了比较。

在估计工作中，我们使用了著名的Constructive Cost模型，将代码的行数转换为以人月为度量单位的工作量。对Boehm在the *Organic Mode of Software Development*中提出的COCOMO的基本模式，我们使用了参数，因为我们所分析的信息系统是由小型团队开发的，而且技术说明书在开发之初并没有完全确定。由此，我们得到了如等式一所示的简单的公式，公式中的2.4和1.05都是模式的参数， KDSI是程序千行数，MM是估计出的开发系统所需人月的数量。

$$MM=2.4\times KDSI^{1.05} \quad (1)$$ 等式一

项目中被分析的信息系统的规模，开发团队的规模，以及开发所需要的实际的工作量，都在表2中显示出来了。对于事件日程表（是一个实例），我们改变了虚拟图书馆模式的接口及其元信息的结构，我们还为已经发生过的事件增加了一个自动存档功能。对于数字图书馆（另外的一个实例），我们需要在虚拟图书馆模式上增加文档库，实现对同质异形元信息的支持，我们还要改变和增加许多管理功能（如，上载文档，以不同格式管理文档，对文档集进行全文检索）。

Project	Size in LOC	Team Size	Actual Effort in MM
Calendar of Events	4021	1	0,5
Digital Library	7616	2	6

表2 所分析项目

	Unit	Total Code	Reused Code	Code Reuse
PERL	LOC	3743	3502	93,56
HTML	LOC	278	192	69,06
Sum	LOC	4021	3694	91,67
Sum	MM	10,35	9,46	91,40

表3 事件日历的代码重用

事件日程表：在线提供事件日程表，是为了从网站中协同收集事件而设计的一个软件，这在虚拟图书馆分析模式中都已经涵盖了。因此，我们可以在执行中重用模式的大部分。表 3 总结了被重用的代码所占的比例。PERL中多于93%的LOC和HTML中69%的LOC可以不经修改被重用。通过使用COCOMO从LOC中估计人月，这一重用实现了总体开发工作量的减少，减少了 9 个以上的人月，或者说，减少了90%以上的工作量。

数字图书馆的开发 基于虚拟图书馆分析模式执行的数字图书馆的开发，对原有模式作出一些改动是必要的。虚拟图书馆模式仅仅提供了管理文档的元信息和引用功能。因此，虚拟图书馆的设计必须进行一定程度的扩张，添加必要的用于存储和管理实际文档的部分。图7显示了主要的变更。

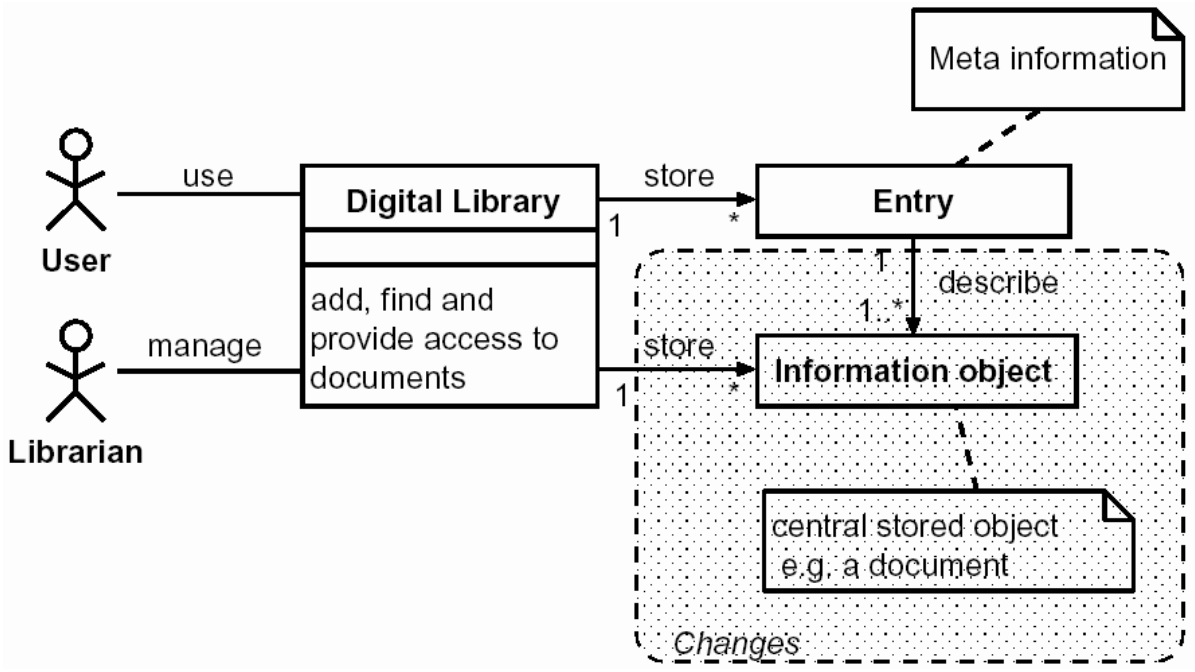


图7 从虚拟图书馆转变到数字图书馆

	Unit	Total Code	Reused Code	Code Reuse
PERL	LOC	6954	3687	53,02
HTML	LOC	662	224	33,84
Sum	LOC	7616	3911	51,35
Sum	MM	20,23	10,05	49,68

表4 开发数字图书馆的代码重用

但是，我们不仅需要对设计作出修改，对虚拟图书馆模式所提供的分析也要进行变动，因为它只有一部分可以被应用于数字图书馆（系统）。虚拟图书馆被用于为分散性信息源提供访问路径，与之不同的是，数字图书馆是对文档形式的信息提供集中式的存储。因而对数字图书馆来说，也就产生了不同的关键力量因素集和结果集。事实上，数字图书馆应该成为它自己的分析模式。

然而，像所有其他模式一样，分析模式的建立是为了引导出更加清晰、灵活和可重用的设计和代码。因此，虚拟图书馆的执行，应该易于被更改和扩展以建立数字图书馆系统。表4总结了由虚拟图书馆而建立数字图书馆的代码重用的结果。PERL中53%以上的LOC和HTML中33%以上的LOC可以不经修改而被重用。这就实现了整体代码51%以上的重用，或是比预计减少了10人月，这大体相当于总体开发工作量的50%。

本部分所示的简单的代码重用分析只能表明分析模式所能够实现的利益的很小一部分。分析模式提供了一个在整个软件开发过程中节约时间和工作量的工具，要想对分析模式所导致的开发时间和工作量的减少程度进行精确的度量，就必须在系统部署之后，抛开使用模式，对所有开发阶段，包括变化，所产生的费用，进行比较。然而这是第一次对代码重用进行评估，其结果提供了强有力的证据，证明已经被用于分析阶段的模式，与设计模式相结合，能够开发出更有活力和更加灵活的系统，这样不仅能够从长远的角度上减少成本，而且可以极大的缩短上市时间。

结论

在今天，不采用以库、组件、框架、设计模式以及其他类似的重用方式而做到软件工程的多产，是不可思议的。然而，最近，更加广阔的问题域重用方法，就像用于整个软件开发过程（从说明书到编码）的信息总和一样的，吸引了人们的注意力。

本文介绍了分析模式，它作为一种途径实现了以结构化方式组织问题域，从而使知识可以重用。通过一些例子，我们阐述了，在所有主要的软件开发阶段。如何使那些用分析模式描述的问题域易于被重用。为了提供证据，证明分析模式可以推动生产力，在这篇文章中，我们对两个虚拟图书馆分析模式的应用，进行了代码重用分析。即使被评估的应用实例是十分不同的（一个是事件日程表，一个是数字图书馆），被应用的模式却表现出了很高的代码重用能力（在49%和91%之间）。这篇文章的结论是可以继续向前研究的。然而，将模式作为组织和重用问题域的方式，对这一做法的效益进行更深层次的和更加具体的研究是十分必要的。一些重要的开放性研究课题包括：

分析模式如何帮助，为开发者和用户提供通用词表？

分析模式如何帮助对文档问题域进行组织和分类，使他们能够方便被访问和重用？

分析模式如何帮助教授有效的分析策略？

分析模式如何帮助描述和理解大型系统？

参考文献

Alexander, C. The Timeless Way of Building, Volume 1 of Center for Environmental Structure Series. Oxford University Press, New York, 1979.

Bentley, R., Appelt, W., Busbach, U., Hinrichs, E., Kerr, D., Sikkel, S., Trevor, J., and Woetzel, G. “Basic Support for Cooperative Work on the World Wide Web,” International Journal of Human-Computer Studies (46:6), June 1997, pp. 827-846.

Boehm, B.W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D.J., and Steece, B. Software Cost Estimation with COCOMO II, Prentice Hall PTR, Upper Saddle River, NJ, 2000.

Boehm, B.W. Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. Pattern-Oriented Software Architecture, A System of Patterns, John Wiley & Sons Ltd, Chichester, 1996.

Champeaux, D., Lea, D., and Faure, P. “The process of object-oriented design,” ACM SIGPLAN Notices (27:10), October 1992, pp. 45-62.

Fach, P.W., "Design Reuse through Frameworks and Patterns," IEEE Software (18:5), September/October 2001, pp. 71-76.

Fowler, M. Analysis Patterns: Reusable Object Models, Object Technology Series. Addison-Wesley Publishing Company, Reading, 1997.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company, New York, 1995.

Gillies, A.C., Software Quality: Theory and Management, Chapmann & Hall, London, 1992.

Kaindl, H. "Difficulties in the Transformation from OO Analysis to Design," IEEE Software (16:5), September/October 1999, pp. 94-102.

Kantor, B., and Lapsley, P. Network News Transfer Protocol, Request for Comments 977, Network Working Group, February 1986.

Malone, T.W., Grant, K.R., Turbak, F.A., Brobst, S.A., and Cohen, M.D. "Intelligent Information-Sharing Systems," Communications of the ACM (30:5), May 1987, pp. 390-402.

Moore, M.M., "Software Reuse: Silver Bullet?" IEEE Software (18:5), September/October 2001, p. 86.

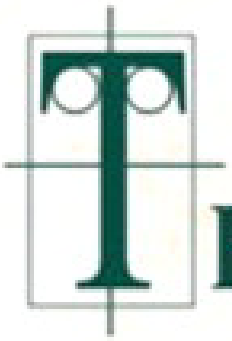
Neighbors, J.M., "The Draco Approach to Constructing Software from Reusable Components," IEEE Transactions of Software Engineeing (10:5), September 1984, pp. 564-574.

NTIS. North American Industry Classification System (NAICS) - United States, National Technical Information Service, Springfield, 1997. ISBN-0-934213-57-7.

Press, L. "Collective Dynabases," Communications of the ACM (35:6), June 1992, pp. 26-32.

Vlissides, J., Pattern Hatching: Design Patterns Applied, Addison Wesley Longman, Reading, 1998.

Weibel, S., Kunze, J., Lagoze, C., and Wolf, M. Dublin Core Metadata for Resource Discovery, Request for Comments

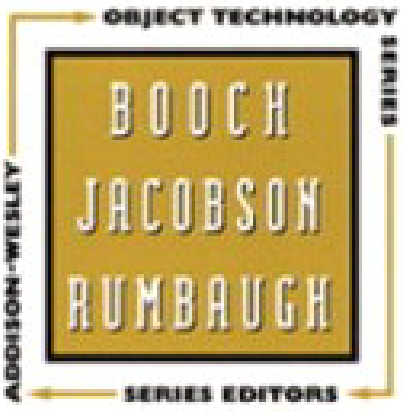


THE UNIFIED MODELING LANGUAGE REFERENCE MANUAL, Second Edition

JAMES RUMBAUGH
IVAR JACOBSON
GRADY BOOCH

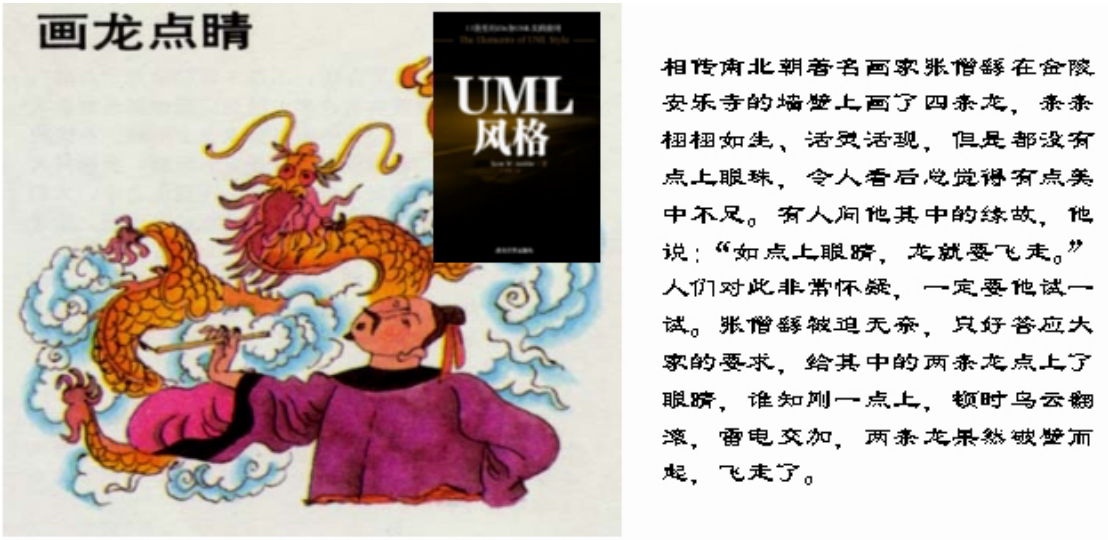
Covers UML 2.0

UMLChina 译
（王海鹏、汪颖）



CD-ROM Included

《UML 参考手册》2.0 版中译本
即将由机械工业出版社出版



它不讲概念，它假设读者已经懂了概念。

它不讲工具，它假设读者已经了解某种工具。

它不讲过程，它假设读者已经了解某种开发过程。

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

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

使用 XP 和 Scrum 通过 CMM L2 和 ISO9001 认证

Christ Vriens 著，李胜利 译

讨论

摘要

本实验报告描述了在一个为期2年的使用敏捷技术通过CMM L2和ISO9001:2000认证的道路。讨论了我们为什么选择XP（极限编程）和Scrum的结合作为我们软件开发过程的基础，以及我们为满足CMM L2和ISO9001的需要不得不增加的“礼节”。也描述了我们的主要挑战和试图解决的方式，进而与大家共享大量的问题和经验。

1. 导言

SES目前拥有26人并且以每年大约10人的速度增加。SES在研究领域执行像环境智能、存储、网络、拷贝保护和机器人技术的软件项目。这些项目的可交付产品既是全球展会上示范的原型，也是以当前研究结果为基础准备商业化的产品。这些项目有以下特征：

- 发行时间固定，例如展会的开始日期
- 模糊的、最少的需求，开发期间也经常改动
- 经常是已经有产品了，客户还不存在
- 技术挑战
- 大多数订约人需要特殊的注意，来防止项目结束后问题和解决方案域知识的丢失
- 2-6个月的短的可用时间框

在这些特征的基础上，我们按优先级递减的顺序给出下面我们项目关键的成功因素：

- 及时发行；简单说，我们不能错过展会的开始
- 质量；如果同意，尽可能在项目开始时与客户一起量化
- 和功能范围

团队成员的数量主要决定了成本。项目时间总是固定的，这大体上意味着范围是我们在开发中能够增大或减小的唯一可变量。

为了明确定位我们部门为一个专业的软件开发机构，并且指导我们的质量活动（保持精力集中和工程进展），我们决定努力具有ISO9001:2000和CMM L2资质。我们提出以下几点作为最大化我们成功的关键因素，并达到我们的ISO和CMM目标：

- 最小化官僚作风（通过薄薄的结果质量手册来限定），为了生成尽可能多的创新空间；编程必须是有趣的！
- 不要出现“车轮的再发明”；使用已经在类似的环境下证明了其自身的实践。
- 为了后期能把它转给其他的飞利浦机构，使用最新的软件开发过程和相似工具

2. 极限编程（XP）

研究学习一些方法（XP, RUP, Octopus, DSDM, FDD）后，我们决定采用XP作为我们的开始。与经常是自顶向下作为管理需要的CMM和ISO的导入相比，正如在许多其他部门一样，XP已经自下向上进入我们的部门。 XP的12项关键实践中的许多内容已经被我们的工程师认为非常熟悉。这些包含简单设计、持续集成、现场客户、小发布、 编码规范、每周40小时工作。而其他的像系统隐喻、结对编程、测试最初设计等仍在不同的不清晰层次上。根据下面的XP的三个成熟级别

- 做记下的每一件事
- 完成事件后，在已有规则中做带有变化的实验
- 最后，不关心是否在用或不用XP（我们仍没达到这个级别）

我们学习了当时可用的书[2]、[3]、[4]，并在日常实践中使用XP。尽管后来者不知道我们起先一直在使用XP，但无论程序员还是客户的结果满意度都是高的。

从这段时间中，我们希望与你共享下面的经验：

- 尽管我们理解系统隐喻的概念和例子，但我们不能提出对我们当前项目的好的隐喻。在这种感觉下，我们当然欣赏Kent Beck在OOPSLA 2002上名为“The Metaphor Metaphor”或非正式称为“Kent最后一次解释隐喻实践”的主题演讲。

- 正如期望的那样，由于相信费用将2倍增长，起初我们的客户拒绝结对编程。在习惯并看到这种较高效的缺陷预防/消除并且减少开发周期的形式的益处后[9]，尽管对于每一个新客户讨论总突然出现，但客户对这一点没有问题。

- 我们不能确定对于单个程序员何时应用结对编程，以及何时接受它。我们目前使用下面的“拇指规则”（经验法则）：

  - 基本上所有的产品代码，包括相关的测试，将结对编写
  - 每一个新任务将以一个小设计会议开始。如果这个会议结果是需求代码相对简单或常规，这允许独立编程
  - 问题报告总是结对解决，因为大多数情况下问题发生在两个或多个模块的接口上
  - 文档可能由一个人编写，并且总是使用Fagan Inspection（检查法）来评审

- 结对编程是一种高能消耗活动。我们尽量限制到最多一天6小时。但每天结束时，给人自信的感觉，关于结对的两个人同意代码的事实。结对编程作为一种防止重复性错误的方式，也非常被认可。

- 无论客户还是程序员都欣赏反复做小改进的工作。客户重视早期发布的商业价值，程序员重视经验学习。在客户定义的订单中及与程序员的合作中，作为需求功能的压力减少和自信增加的两种感觉不断释放。

- 我们体会到跟踪XP实践需要很多纪律，并且我们评估我们同事的压力，让我们停在“敏捷”的轨道上

■ 日常的站立会议（我们只是使用这个名字，我们总坐着）逐渐地取代通常的状态报告会议，这种会议被认为太耗时，并且大体上太多的项目直接范围外的人参与。他也去除了项目领导者的“警察”感觉，因为他每天上午得到所有的需要信息，就不再为了得到信息“不得不作巡回问询”。他得到的信息包括昨天完成了哪些，今天将作些什么和项目领导者不得不快速解决的问题

■ 我们喜欢XP中程序员和客户表达职责的直接方式：客户在听完程序员的反馈后决定软件应该提交什么和以什么顺序，并且面对和接受结果，尽管我们明确地欣赏这里的一些客户的输入，但程序员决定如何开发。

3. XP@Scrum

关于使用极限编程的问题。尽管我们大体上对应用XP非常满意，但我们仍为下列问题而斗争：

1．XP不给你很多的关于文档，建模，UML使用和设计模式。对于这些问题，书[6]和[1]完全补足了XP

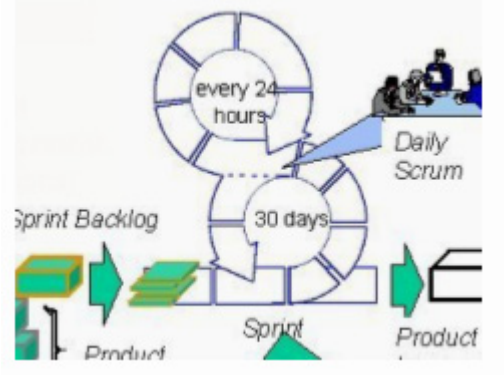
2．新招的软件工程师不得不遵从在XP中致力于体系结构。团队只是在12种实践中简单提到。我们目前相信对每一个新员工来说，需要一个关于XP的良好的介绍课程。他们必须按照书面测试首先对体系结构和设计作每件事情：它强迫从一个高度抽象层次用户的观点来察看代码。并且提供一个获得低耦合和高内聚的规则的实践方式。这是因为测试必须是全部自包含的。此外如Martin Fowler[7]解释的重构允许不仅在项目开始时（正如在瀑布模型中）专注于体系结构，而且提供以高效成本来进化软件体系结构的机会。的确，你所使用的设计原则与边写边改相比必须从一个实际的上下文关系来解释。

3．XP在一个组织中不帮助我们介绍关于XP的知识，不帮助我们如何优化或增强工作的方式和如何与你的管理相互影响。同样对大多数管理者来说，极限这个词也是恐怖的。这里也许我们认为这个词是极好的。当然，我们明白如果XP没有那么一个煽动的名称的话，XP应当不是现在这个样子。

4．我们不能确信我们所有的客户以接受测试的形式提供明确的需求。我们能从他们那里得到的最好结果是执行的场景和一些与关于期望结果相关的句子。尽管在多媒体领域，在用户层次上定义所有需求的自动测试对它本身是一个挑战。现在我们在团队对客户示范目前实现的功能并且一起讨论是否接受的迭代过程后期安排会议日程。

5．为了能增加重要的功能，一个两周时间的迭代对于我们小团队（1-5人）太短了。我们现在使用一个月历的迭代长度，在我们的项目中引入一个美好的节奏：在月初新的需求不得不被确定，在月末这些需求不得不被接受。这容易记住，甚至不用察看我们的日历。

6．我们感到在用户故事列表中不仅需要输入功能性需求，像从一个工具转向另一个，也需要非功能性需求，一个主要的重构或一个特殊的需求文档。毕竟，为它们支付费用的是客户，我们想让它们变得可见和被认可。现在我们也增加问题报告，并且改变需求列表。



然后Scrum来了。经历了一年的仅XP使用的工作后，一本关于Scrum [8]的书引起了我们的注意。我们立即认可了问题（issue）3到6，和由Scrum提供的比得上我们介绍自己的解决方案。我们决定结合Scrum和XP，因为XP注重与工程实践，Scrum注重于管理和组织方面，工作进行得非常平稳。他们由称为Sprint的XP计划竞赛和在Scrum中的计划会议相互交迭，这不奇怪，正如Kent Beck在XP中重用Scrum中的元素。后来，我们意识到经常结合使用XP和Scrum，甚至给它一个名字XP@Scrum。这时，我们不仅使用Sprint评审会议作为与用户会面的认可形式，也使用检讨，它也是认可形式的一部分，作为通向学习型组织的第一步和在不同的程序员团队间沟通最佳实践的一种方式。

4. ISO 9001 和 CMM L2

在我们组织内介绍XP和Scrum的同时，我们致力于通过ISO9001和CMM L2认证。请注意认证不是，从来不是自身的一个目标。我们使用它为了测量我们使用所谓的自评估指南进行的质量活动，在CMM中称为过渡成熟度评估（IME），并且能够作为我们与其他机构合作的软件开发过程的基准。

当我们比较ISO9001和CMM时，令我们惊奇的是ISO（当然是在2000新版本中）明确地提到客户满意，而CMM假设采取坚持不同关键过程域的需求暗中引导到较高程度的客户满意。

与CMM相比，ISO9001有一个较高层次的抽象和一个较宽领域（硬件、软件、处理材料、服务…），它严格专注于软件。这意味着我们不得不讨论怎样说明我们组织的撰写ISO需求。为了在这个讨论中帮助我们和弄清楚需要什么程度的言辞使CMM和ISO审核师满意，我们为这两个质量管理系统分别雇用了一名顾问。对这些顾问的最重要的选择标准是他们自愿从一个敏捷的观点解释CMM和ISO的需要。正如描述组织该做什么和不做什么，以及怎么做一样，存在解释的自由度。

CMM。因为CMM主要关注大项目和大机构，转化不得不做到我们自己的特殊的环境中，对这一点，我们仅使用简单的常识。我们通过为每一个可应用的L2关键过程域（需求管理、项目管理、项目跟踪和勘察、配置管理和质量保证）用至少两页纸定义一个程序。只有在从我们的顾问和经常的激烈辩论中得到明确的要求后，一个程序肯定要超过这两页。这些程序包含了我们该做什么和如何做的指令和指南。现在大多数程序是两页，最大的是四页。在XP和Scrum中只有配置管理和质量保证的过程没有直接的和足够的支持（在CMM和ISO术语中）。为了提供适当可见度的管理，增加一名质量人员对我们的敏捷观点来说甚至是一种促进。我们将在后面的部分进一步关注这个主题。

在这么多过程后面，为了增加不同项目发布的一致性和我们工作方式的效率，定义了许多小模板和检查列表。虽然定义是一件事情而部署是另外的事。我们将在下一部分描述一些挑战，这些是在我们写这篇论文时仍没有解决的。

ISO9001。对于ISO9001的考虑，我们在预定义的格式里只定义了一种过程描述，它在一个抽象层次上描述我们混合使用XP和Scrum。这里需要更详细的描述，我们参考了书[2]和[8]。此外，我们包含了很多已经定义的程序、模板和检查列表的参考。经过文档化处理、内部审核、纠错和预防处理、错误控制和质量注册的所有ISO9001需要的其他过程被我们所有的研究机构的质量系统再利用。

度量。坚持ISO9001和CMM L2也需要介绍度量，作为良好控制的基础之一。目前我们收集到下列数据：

- 每六个月无论客户还是程序员的满意度。通常我们在0-10分中得到7-8分
- 不同编程团队每次迭代的速度
- 每次迭代中测试和功能代码增加的行数
- 作为结果的测试覆盖
- 每三个月，从指导的IME的得分
- 凡个项目每一个月迭代的主要和次要错误的数量

5. 挑战

高级的管理问题。正如从一侧看，管理问题随着团队使用Scrum实践变得更自组织性而增加。但为了满足下面的ISO和CMM需要，更多的问题需要解决：

- 对已定义的软件开发过程的管理承诺
- 建立和充实一个独立的质量保证小组
- 复查和批准所有承诺，并且改变对外部人员的承诺
- 从质量保证检查中解决报告的错误问题，以防在项目范围扩大后逐步升高
- 复查不同的已定义过程的实现
- 检查不同项目在一个周期基础上的时间安排

我们仍然在讨论怎么以一个敏捷的方式专注于这些需求的高级管理问题。

质量保证。无论ISO9001和CMM都需要存在一个高度独立的质量保证小组来提供在使用过程中的合适的可视化的管理。结对编程中的对等压力能对实现标准一致的高层次保证格外有效，但是不必要给出在错误问题中的可视化管理。我们目前正在每个项目的每次迭代的后期部署质量保证检查，但是我们仍不得不定义如何管理逐步增多的团队自身不能适当解决的错误，

欧洲的敏捷培训和导师。随着我们部门的稳步增长（每年10人）和严重依赖大量的“爆炸性”的承包商（大约80%），持续需要合适的培训和指导。这种在欧洲的熟练的导师缺乏和在不同国家飞来飞去的高费用的混合，导致了目前很多项目执行在不同的XP和Scrum成熟度上。

正确的风险评估。目前，只有风险识别存在于我们项目的CMM的项目管理中。对于项目跟踪和勘察，我们正和别的方法一起，如效果分析，研究怎样改进风险评估，与XP注重实效的方法相比不试图预测或控制未来。XP规定一个团队能投入更多的能量保持系统尽可能简单。

6. 经验

只招聘最好的人员。良好的工程师必须是你最先考虑的事。没有过程，无论是严格的还是敏捷的，能够替代才能和技能。作为我们创新的副产物，基于XP@Scrum的开发软件的方式自身已经证明作为一种高质量的工程师对我们部门感兴趣的方式。

允许用户定制的空间。由于关键程度、问题域、应用技术和团队的物理分布不同，不同的项目拥有不同的过程需要。

质量官必须是一个教练而不是一个警察。质量官能在一个敏捷的环境中起作用的最好方式是通过与项目团队工作在一个规则基础上，并且在已经定义的工作方式中指导他们。不服从必须被认为是一个提高项目团队工作方式的机会。通过以单项形式只检查长长的质量问题列表做一个警察，工作会适得其反的。目前我们使用其它团队的领导作为一个特定团队的质量官。

正确使用时间盒。工程师和客户必须领会到：在这个迭代终将实现什么，什么不实现。

专题检讨会。专题检讨会，由Scrum定义，在每次迭代后期是一种使你的程序员团队增长成熟度和效力的敏捷方式。

预算。我们经常听说的一个基本规则是保留资源容量的15%在两年内提升你部门从CMM L1到L2。基本上，由于各种所有项目组织相似的压力，实际上这很可能变为10%。通过再利用其他部门（所有飞利浦内部的或可免费使用的）的现有技术、过程描述、程序、模板和检查列表，并且通过坚持敏捷的工作方式，我们达到了保持预算花费在5%左右。

现场客户。我们体会到在与客户亲密中安排合适的空间是更加简单的，然后说服客户坐下来或经常访问编程团队。这导致编程团队分散在我们园区各处，作为一个结果，缺少技术知识的共享。现在我们组织午餐会议，所有的不同项目团队的成员都在场，通过技术演讲明显地促进了知识共享。＝＝到此＝＝＝

XP不适合每一个人。XP和Scrum是高度纪律性的方法，强调口头交流胜过书面交流。不是每一个工程师或客户都欣赏这种工作方式，并且一些人不能坚持。你应该尊重这一点。后果是一些工程师可能离开你的组织，还有一些客户可能由其它供应商提供更好的服务。

7. 结束语

2002年5月，我们基于XP@Scrum的软件开发过程通过了ISO9001:2000，并且在2002年12月的一个快速扫描显示我们在2003年通过CMM L2认证的良好的轨道上。但是正如在挑战部分中表达的那样，同时仍有许多主要的障碍要克服。

8. 参考文献

[1] Alistair Cockburn, *Agile Software Development*, Addison-Wesley, ISBN 0201699699, December 2001

[2] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, ISBN 0201616416, October 1999

[3] Kent Beck and Martin Fowler, *Planning Extreme Programming*, Addison-Wesley, ISBN 0201710919, October 2000

[4] Ron Jeffries, Ann Anderson, and Chet Hendrickson, *Extreme Programming Installed*, Addison-Wesley, ISBN 0201708426, October 2000

[5] Mark C. Paulk, *XP from a CMM Perspective*, November/December 2001 issue of IEEE Software

[6] Scott Ambler, *Agile Modeling*, Wiley Computer Publishing, ISBN 0471202827, March 2002

[7] Martin Fowler, *Refactoring - Improving the Design of Existing Code*, Addison-Wesley, ISBN 0201485672, June 1999

[8] Ken Schwaber and Mike Beedle, *Agile Software Development with Scrum*, Prentice Hall, ISBN 0130676349, October 2001

[9] Laurie Williams, *Pair Programming Illuminated*, Addison-Wesley, ISBN 0201745763, June 2002

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

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

通过用例整合业务流、工作流和对象模型

Proforma 著，王志航 译

讨论

介绍

我们的行业一直追求一种形式化的连续的应用开发生命周期。我们需要理解我们的业务并且优雅地将对业务的理解整合到对业务系统的设计和实现当中去。我们想让每个生命周期阶段的交付可以向前和向后可跟踪地融入到随后的各个阶段。并且我们不想重组我们的开发人员。

手动的记录和管理生命周期的细节的尝试已经证明需要比我们项目的截止日期可以容许的还要多的努力。结果，团队经常走捷径，比如在设计的初期忽略业务需求。过去的努力缺少一个统一的方法，尤其在分析和设计领域。结果，组织一个能够保持一个统一方法的团队（很）困难。

许多比较好的解决方案是使用单一的 CASE 工具，这套 CASE 工具许多组织不能或者不想遵守。这些集成的工具需要在设备和培训上的大规模投入来“武装”团队中相对来说比较小的部分。

今天，现代化的计算机数量丰富的并且功能强大。团队不需要在特殊的高性能工作站和专门的培训上面进行特别的投入来形成概念-代码的解决方案。通过拖拉应用软件可以更容易地在同一台机器或者在机器之间交换数据。每个人是“连在一起的”并且以电子形式交换数据。

新的建模技术和应用软件从 90 年代就已经出现了，主要是由于大量的 BPR 和 OO 的出现。今天，工作流模型、统一建模语言(UML)、关系技术、面向对象技术(OO)和用例建模技术代表了主要的无异议的标准。总之，这些是连续生命周期的关键组件。

此篇论文描述了在现代应用开发生命周期中用来进行企业建模的 ProVision Workbench 方法和技术。

应用开发生命周期

应用开发“生命周期”从业务（或者企业）开始，以产品代码结束。当我们遍历生命周期的时候，我们发现大量的迭代点，但是最终生命周期过程通过三个主要的（经典的）阶段：概念（模型）、逻辑（模型）和物理模型。

生命周期中概念模型阶段包括理解业务。有时通过概念模型来改进业务过程，其他时候概念模型用来指导应用系统的开发或者购买。在这里的工作经常叫做“企业建模”或者“业务分析”。主要的工作成果通常是“业务需求”。

在逻辑模型阶段，业务需求帮助指导系统组件的设计。这个阶段通常认为是系统架构和设计。主要的工作成果是“系统说明”。

在物理模型阶段将逻辑模型阶段的产品映射到一个具体的平台。这个阶段包括底层的设计、编码、测试以及发布和维护。主要的工作成果是产品系统。

本质上，应用软件开发生命周期包括两个主要的元素，业务部分和它的系统部分。业务流程再造（BPR）和补充的方法代表了用来改进业务过程和系统的现代方法学，并且因此改进业务流程。BPR 挑战现有的业务执行方式，它发明新的执行业务过程的方式。这种新的业务执行方式可以满足市场的需要并且拥有新技术的能力。

我们采用技术以达到业务规则的改变。系统分析、设计和开发的新方法是面向对象技术（OO）。用例建模，作为一个被 OO 技术普遍采用的分析技术，以人和系统功能之间的交互的方式，进一步增加了有效的描述业务情形的方式。

该文表述了如何整合业务过程和工作流、用例和对象模型来为企业提供一种高效理解并表示业务和系统需求的方法。

企业建模

概念模型作为起点

企业模型实现生命周期中的概念阶段。企业就是在这里变成了可以管理的部件。选择部件就是有计划地改进业务流程(BPR 或者业务过程改进)并且/或者阐明业务需求(业务分析)。

企业建模并不意味着研究整个企业。在许多情况下，为了准备新系统的开发或者企业软件购买，企业建模是在一个确定的范围内。

下面的讨论集中在两个模型：工作流模型和用例模型。这两个模型在企业建模中具有主要作用。

工作流模型

企业建模的一种重要工具是工作流模型（或者过程地图）。这种模型描述业务过程中业务活动的优先性。它可以让分析师识别并且纠正（企业中的）瓶颈和无效率（的环节）。除了（描述）优先性之外，工作流模型还可以表达每一个活动由哪个部门负责。这通过将给定部门的活动放在代表部门的“泳道”中来实现。

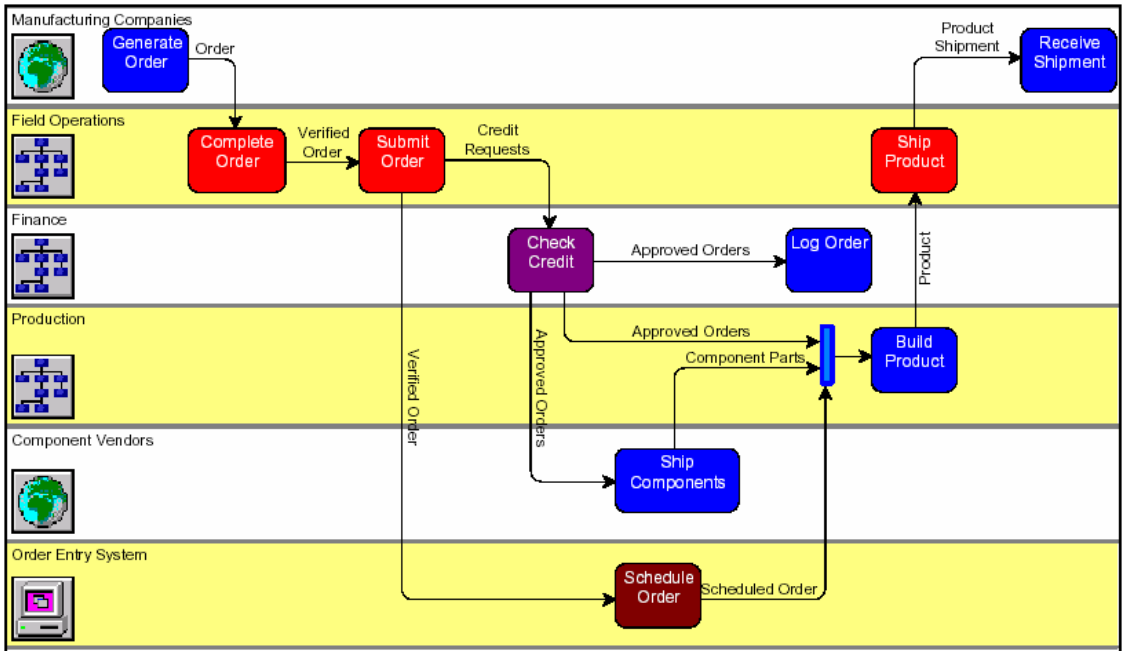


图 1 工作流模型描写了业务流程中活动的流动

上面的工作流模型（图 1）描写了在实现订单过程中的活动和负责每一个活动的部门或者系统。由于业务流程非常复杂，嵌入工作流模型通常比将所有的细节放在一个模型中更加有用。这样就创建了一种层次模型，每一个模型描写了单一活动的细节。

为了达到改进流程的目的，活动和工作流（连接）带有诸如它们需要的时间和金钱等丰富的信息。通过分析过程成本和花费的时间，业务分析师可以再造工作流程，合并或者消除活动等来设计更好的业务过程。

用例模型

用例模型是用来表达人、部门和系统如何交互来完成业务（或者系统）的一部分的一种非常流行的技术。它是一种简单的模型，然而提供了强大的方式表达业务中的连续性来支持系统。

用例模型的本质思想是表达角色（工作功能、部门和外部系统）如何和公认的业务或者系统功能交互。完整的用例模型代表了一种原型的交互。从这种交互可以得出大量的应用场景。

简单性是用例模型流行的一个主要原因。用例模型通常描述在现实世界中观察到的具体的情形。

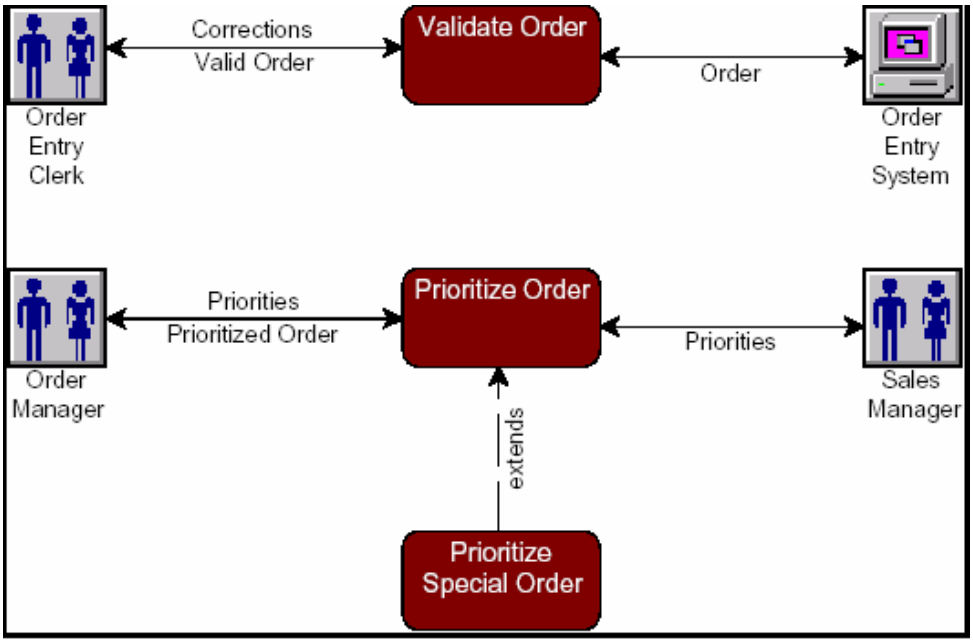


图 2 用例模型描绘一个原型的场景

图 2 描述了完成订单所需要的相互作用。这个相互作用包括验证和优先级分级。订单入口职员和订单入口系统都参与订单验证。订单管理者和销售管理者参与订单优先级分级。有时，订单会要求特殊的处理。用例模型通过“分级特殊订单(Prioritize Order)”的扩展用例“分级特殊订单(Prioritize Special Order)”表示这种情况。

用例模型的非正式性促成了用例的简单性但是阻碍了用例整个的一致性。产生许多用例模型是正常的，每一个用例代表了对业务系统的不同的看法。

用例模型需要仔细的管理。这些模型表达了相当小的范围内的细节并且对其他用例模型关注很少。在大型项目中，用例模型通常相互冲突。也非常容易忽略用例应该使用的领域，在需求中留下漏洞。

重大的项目受益于更全面的模型来帮助避免重叠并且保证用例模型全面地代表要研究的整个范围。

从工作流模型到用例模型

工作流模型是用例模型的自然宿主

在工作流模型中每一个低层次的活动描写业务系统中的一个重要的但是良好划定的部分（或者系统的部分）。用例模型用来表达用户如何使用功能来完成活动。

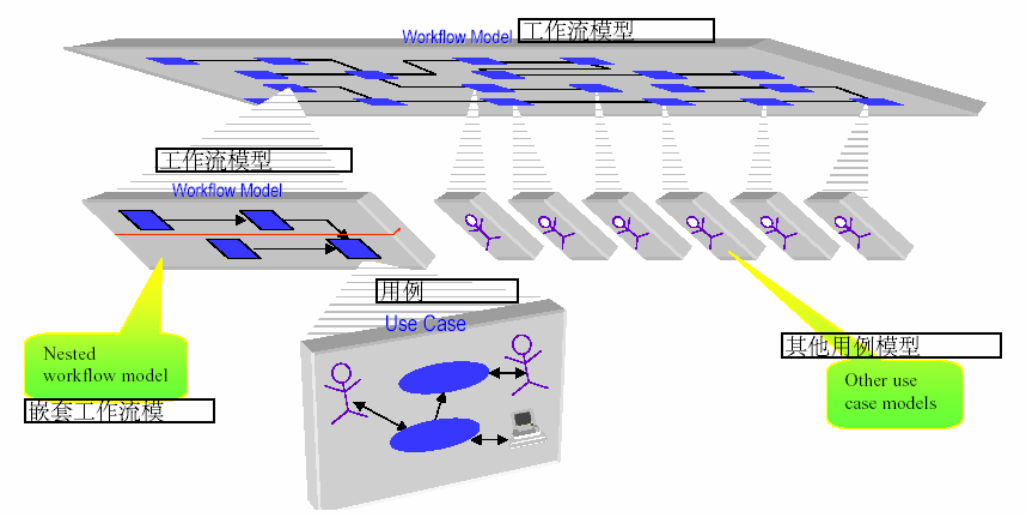


图 3 工作流模型为包含用例模型提供了广阔的空间

图 3 描绘了在工作流模型和用例模型之间的连接。工作流模型覆盖广阔的领域，这些领域包含大量的具有各种各样活动的部门。这个模型自身提供了对业务广阔的和细节的见解。

用例模型补充工作流模型。工作流模型中的每一个活动是用例模型中的一个候选的主题。用例以功能（子活动或者用例）、业务和/或系统实体的方式描写活动中发生的事情。

从概念模型到逻辑模型

从概念模型到逻辑模型的过渡包括识别系统组件来实现概念模型的业务组件。用例模型为这种过渡提供了一种优秀的工具。

用例模型

用例模型在到逻辑阶段的过渡中起了主要的作用。它提供了系统用户（以及外部系统）如何与自动的业务活动交互的抽象视图。这为识别和设计用户和系统的接口提供了一种结构。

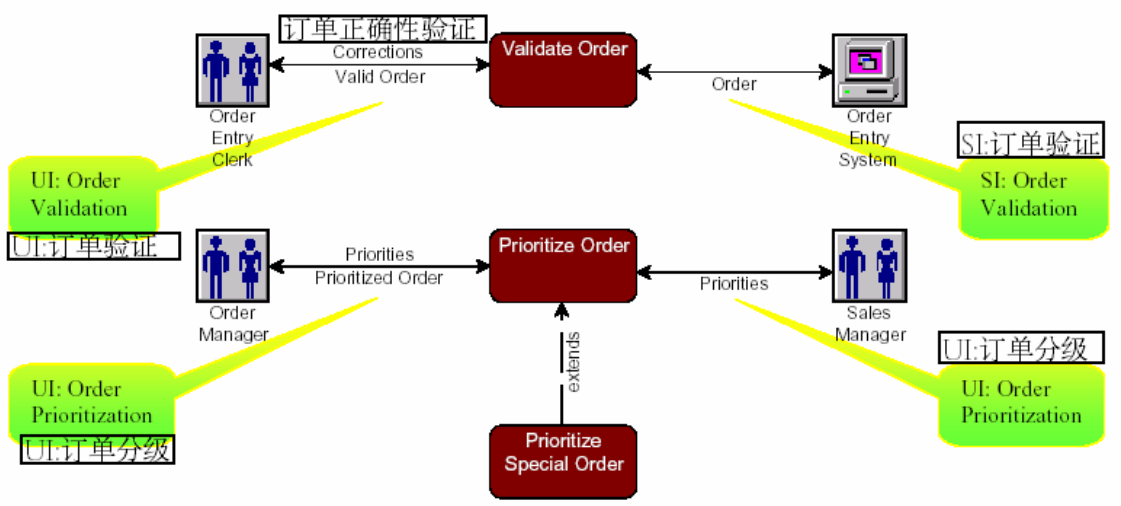


图 4 用例模型表明候选的接口

用户和系统接口是系统架构的典型部分。历史上，这也是生命周期连续性停止的环节。

当在工作流模型的结构下开发时，用例模型描述所有的关键用户和系统交互。设计系统的用户接口包括检查每一个用例模型来确认候选用户和系统接口。当业务实体（例如角色或者部门）和功能交互的时候一个候选用户接口就会出现。图 4 的用例模型描述了三个候选用户接口和一个候选系统接口。

用户接口设计

候选用户或系统接口提供了设计的范围。设计过程包括以提供给用户的数据和功能的方式理解用户接口。

当描述接口的时候有几种模型可供使用。模型和他们的用法主要取决于接口的复杂性。在许多情况下，设计者需要首先将用户接口描述成类模型。类模型描述包含在接口中的类、属性、关系以及方法。在复杂情况下设计者将采用接口模型（或者协作模型）描述接口中对象的动态行为。在所有情形下，设计者将以文本和/或摘要的形式向用户接口提供细节信息。细节信息主要用来和用户接口的开发者沟通设计意图。

图 5 中在订单入口职员和“验证订单”之间的接口在类模型中进一步作了详细说明并且形成了摘要模板。类模型描述在交互过程中对订单入口职员有用的数据和功能。相关的情节模板提供了用户接口应该显现的图形“轮廓”或者粗略的描述。

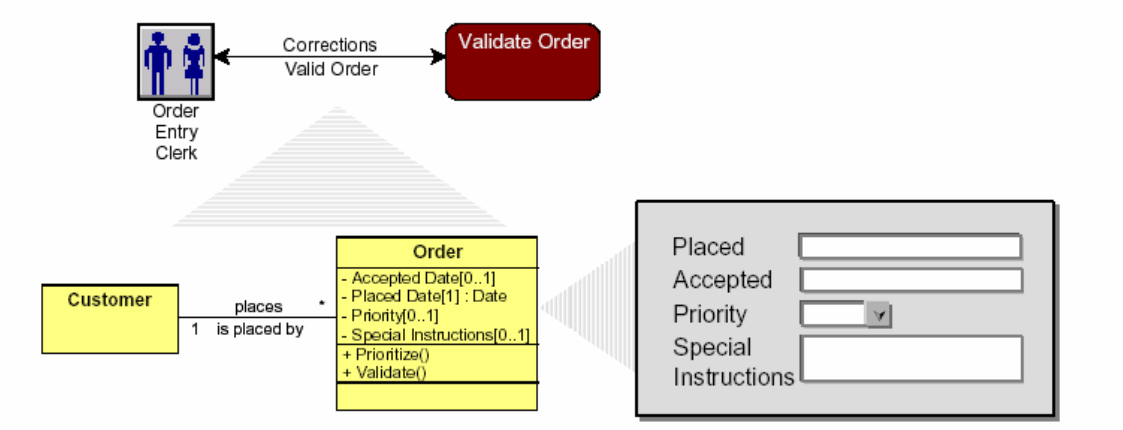


图 5 用例提供了识别用户和系统接口的结构化的方法

功能（部件）设计

设计的第一个阶段提供了一个类模型来澄清实现系统活动的部件。在需要的地方，诸如协作模型或交互模型等更多的模型描述组件如何交互来完成活动。

数据设计

数据体系结构总是为一些数据存储技术设计。不管下面的存储技术是关系型或者面向对象型，设计者必须处理持久性的管理。对于关系型技术来说，良好的持久性机制的设计从将功能体系结构的所有部件的持久数据映射到关系表中开始。以后的步骤包括传统的关系表设计以及提供与功能体系交互用的清晰的抽象。

系统接口设计

接口体系结构，就像表示的体系结构一样，以用例模型作为开始。和外部系统的每一个交互都是一个候选的接口。图 4 描述了在订单确认和订单入口系统之间的接口。这个接口的设计包括通过类模型描述纯数据交换以及以（如 CORBA，批处理文件，进程件通信）等技术为基础的更底层的工程实现。

编码

应用开发周期的目标是产生能够准确地支持业务的可以工作的系统。

系统（适当的）从设计开始开发。系统的设计包括体系结构设计以及关键部件的设计（用户/系统接口，功能部件，关系表等）。

我们行业拥有大量的程序编码/开发工具。这些工具支持低层次的设计和代码的开发。

相应的，这些工具专注于编码，而不尝试提供一种覆盖整个开发周期的集成的方法。一种实用的概念-代码方案是使得概念和逻辑部件可以和在开发组织者中使用的不同的编码系统交互。

ProVision Workbench（PVM）设计用来补充现存的编程环境。PVM 提供了一个整合的库。它可以通过 C++、Java 和 DLL 代码的产生提供对企业模型的密切跟踪。

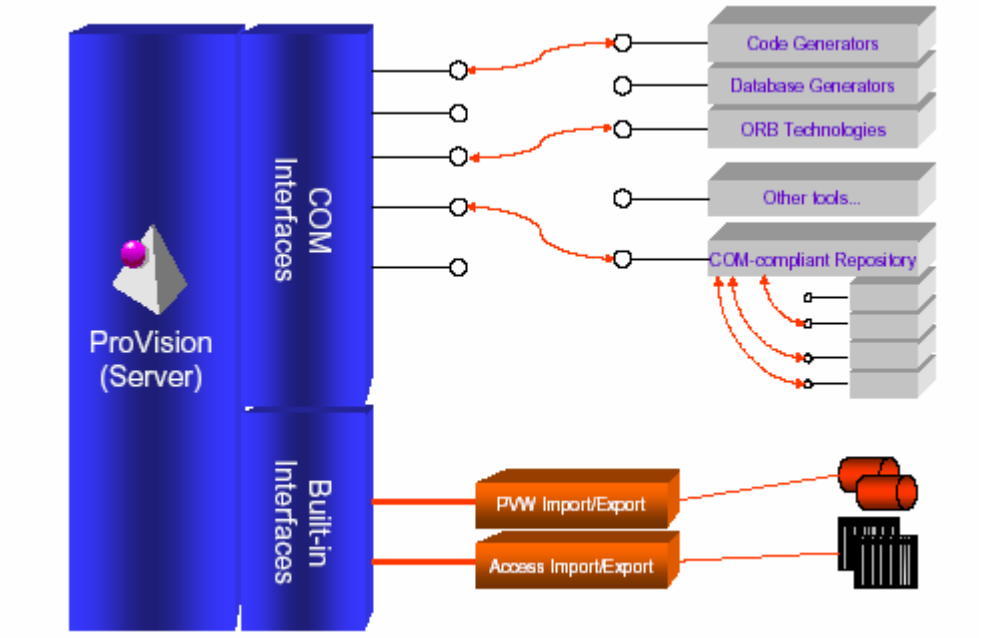


图 6 生命周期技术必须和许多不同的编程环境共存

从逻辑模型到物理实现的过渡本质上包含了可以反映设计意图的代码。从这个角度讲，开发人员使用编程环境来做详细的逻辑设计、编码和测试工作。

由于开发通常揭示在设计和业务需求上的变化，因此需要双向的接口。编程环境的接口技术中的关键技术是微软的 COM 接口。从编程环境中开发的部件可以通过反向工程提取设计信息并自动地更新在 PVW 库中的设计模型。

ProVision Workbench

ProVision Workbench (PVW)是可以使用现有的代码编写工具实现概念到代码的获奖产品。它是市场上唯一可以清晰地将工作流模型绑定到用例和类模型的分析和设计工具。它提供了从业务分析到系统设计的简单的过渡。

PVW 支持使用最流行的工作流方法学从简单到复杂的企业建模。除了工作流模型外，存在的模型还代表了业务需求的其他方面。企业模型是：

业务模型	描述
业务交互模型	提供了业务的阶段性视图。描述业务并且表明了组织的边界以及和内部及外部组织的交互
工作流模型	以部件活动和这些活动中的工作流的方式表示业务过程
目标模型	显示和业务过程以及这些过程中的活动相关的业务目标体系
过程模型	显示将业务领域分解成过程和活动
组织模型	以层次的结构显示企业的组织和企业中的角色
位置模型	显示对企业感兴趣的地理位置
事件模型	组织关键的业务事件和子事件
系统模型	表示支持企业的系统和子系统

从概念到逻辑模型的转变通过用例模型可以最好地表现出来。然而 PVM 支持更多的模型来很详细地描述业务需求。

分析模型	描述
用例模型	描绘原型场景，包括业务中的“执行者”（例如角色、部门、系统）和他们用来完成业务活动的功能。
业务对象模型	显示对业务感兴趣的业务对象（例如订单、消费者）的属性和它们之间的交互

子类型模型	描绘业务对象的基本的层次，揭示超类/子类关系
交互模型	通过消息传递表示对象如何协作来完成给定的功能
状态模型	以状态（对象的生命中的关键阶段）和这些状态之间的转换的方式表示指定类的各个独立的对象的生命周期
方法模型	通过消息表示完成单独一个方法需要的代理

模型，不仅仅是简单的图表

一幅图表示一张图片。每幅图表可以以图形的方式传递思想，但是不支持下面的特点：

- 完整性
- 超越一般的名字和描述的属性
- 共享或者复用其他图表上面的对象
- 智能功能例如完整性检查和跟踪
- 和其他图表的整合（反映所有图表中单独的改变）

每个 PVW 模型是单独整合的元模型的视图。从一个视图的改变可以自动的在其他的视图中反映出来。每个模型可以包含（逻辑上）无数的对象和这些对象之间的连接。

每个 PVW 对象是智能的。它知道如何参加到模型中并且如何和其他的对象交互。除此之外，每个对象包含了除了名称和描述之外的特有的信息。

智能的整合的对象允许一起工作的质量模型在生命阶段中的过渡并且提供了自动的向前和向后的跟踪。

WILEY

TIMELY. PRACTICAL. RELIABLE.

UMLChina 指定教材

Agile Database Techniques

Effective Strategies for the Agile Software Developer

《敏捷数据》

UMLChina 李巍 译

机械工业出版社即将出版

Scott Ambler

使用 Together 让你的项目变得更加敏捷（下）

周小辉 著

敏捷特性 3：重构、模式、测试、审计和度量

对于团队来说，要采用建模工具，除了学习规范以外，还需要进行总结以形成最佳实践。本节所介绍的 Together 的敏捷特性 3，就是一个使用 Together 的最佳实践。

代码评审、审计和度量

我们知道，在 CMM 中，特别强调在项目中进行代码评审，代码评审可以提高软件项目的成功率，因而 XP 提倡结对编程，将这一做法发挥到了极致。Together 的审计与度量特性可以使你的代码评审过程更加富有效率。

代码评审涉及四个主要区域：

1、是否遵循了代码标准，是否违反了团队的最佳实践；

2、是否与设计相违背；

3、是否遵循业务逻辑；

4、代码的写法是否有可能造成性能问题。

自动化的审计和度量虽然大大有助于指出语法和设计的质量，却无法确定软件的业务逻辑是否正确。但判断软件的业务逻辑是否正确，却是一个十分耗时的工作，因而具备自动化审计和度量能力的工具能够使我们在进行代码评审时更专注于去发现软件中所存在的业务逻辑问题。

在实践中，代码评审应该作为 QA 程序的一部分。

重构、模式

重构是迭代开发过程的一部分，并将设计师从想要“一次就将软件完全设计好”的泥潭中解脱出来，也避免了过度设计，这并不是一个新概念，但它对软件项目的长期成功却十分重要。

在 Martin Flower 的《Refactoring _Improving the Design of Existing Code》一书中，提出了超过七十种行之有效的重构方法。

Together 对 Java 项目的重构提供了广泛的支持，某些重构操作也可以应用于其他语言。

重构在Together中是一个可激活的特性。你可以从Together的主菜单中激活它或将它置为非激活。

要激活“重构”特性：

1）、从主菜单中选择菜单“Tools | Activate/Deactivate Features”；

2）、在出现的对话框中，选择“Together Features”页；

3）、选中“Refactoring”复选框；

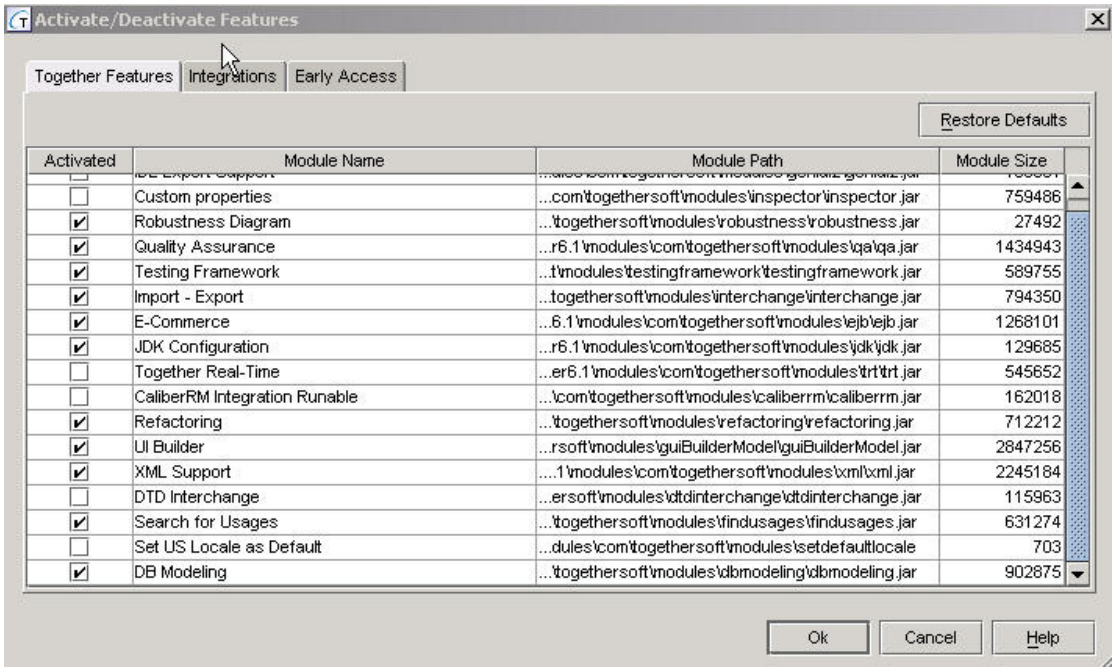


图13 激活重构特性

4）、点击“Ok”按钮以激活所选择的特性并关闭对话框。

当“重构”特性为激活状态后，你可以以多种方式进行代码重构：

1、从Together的主菜单种选择菜单“Tools | Refactoring”；

2、在浏览器（Explorer）或设计器（Designer）中当前选择项的弹出菜单中选择“Refactoring”；

3、从编辑器（Editor）的弹出菜单中选择“Refactoring”，鼠标的位置决定了要被重构的元素（除Extrac Operation外，所有其他重构命令都要求鼠标必须位于要被重构的元素的名称上）。

“Refactoring”命令带有一个子菜单，列出了具体的重构命令。具体的重构命令列表根据所选择的元素不同而不同。“ Tools |Refactoring”菜单包含了重构命令的完整列表，但只有对当前选择项适用的命令是使能（生效）的。

下表中列出了所有的重构命令，你可以从设计器、浏览器、编辑器的弹出菜单和“Tools”主菜单中调用绝大多数命令：

重构命令	应用于…
封装属性（Encapsulate Attribute）	一个或多个属性
提取接口（Extract Interface）	一个类
	多个类
	单个类的多个成员
提取方法（Extract Operation）	编辑器中所选择的代码块
提取超类（Extract SuperClass）	单个类（接口）
	多个类或接口
	单个类（接口）的多个成员
移动类（Move Class）	一个或多个类
上移成员（Pull Up Member）	单个类的一个或多个成员
Push Down Member	单个类的一个或多个成员
重命名属性（Rename Attribute）	单个属性
重命名类（Rename Class）	单个类
重命名接口（Rename Interface）	单个接口
重命名方法（Rename Operation）	一个方法
重命名属性（Rename Property）	单个属性
重命名变量（Rename Variable）	一个参数或局部变量
显示祖先（Show Ancestors）	一个或多个类（接口）
显示派生（Show Descendants）	一个或多个类（接口）
显示实现（Show Implementing）	一个或多个类（接口）
显示重载（Show Overrides）	单个方法

当进行重构时，我们经常会发现，应用设计模式是一件非常自然的事情，由于在 Together 中对模式进行了广泛的支持，因而使得重构和模式得以无缝地连接到一起。下图显示了 Together 中所支持的模式：

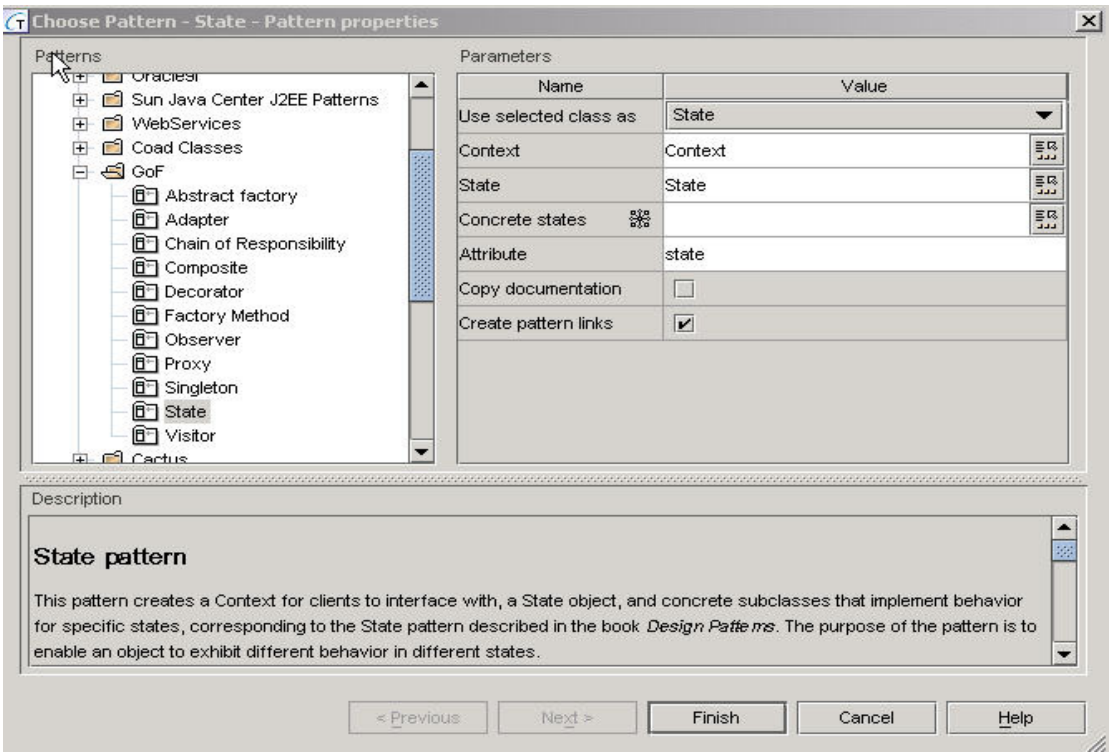


图 14 对模式的广泛支持使得重构与设计变得敏捷

在《重构》一书中，Martin.Flower 向我们演示了一个使用 State/StrateGy 模式对代码进行重构的例子，书中用了很长的篇幅，不过采用 Together 却只需要短短的几分钟即可完成。限于篇幅，这里就不作介绍了，有兴趣的读者不妨一试。

重构与测试

Together 的一个主要特性是它提供了一个强大的测试框架，支持对源代码进行单元测试，以及对 GUI 进行可视化测试。

重构特性与测试框架支持在一个 IDE 环境中的无缝结合，使得二者相得益彰。

还是让我们以一个简单的例子来说明一下 Together 对单元测试的支持吧：

第一步：首先建立一个范例 Together 项目，姑且起名为 TestSample，并在其中创建两个待测试的类；

1、其一为一个简单的计数器类 Counter，代码如图所示：

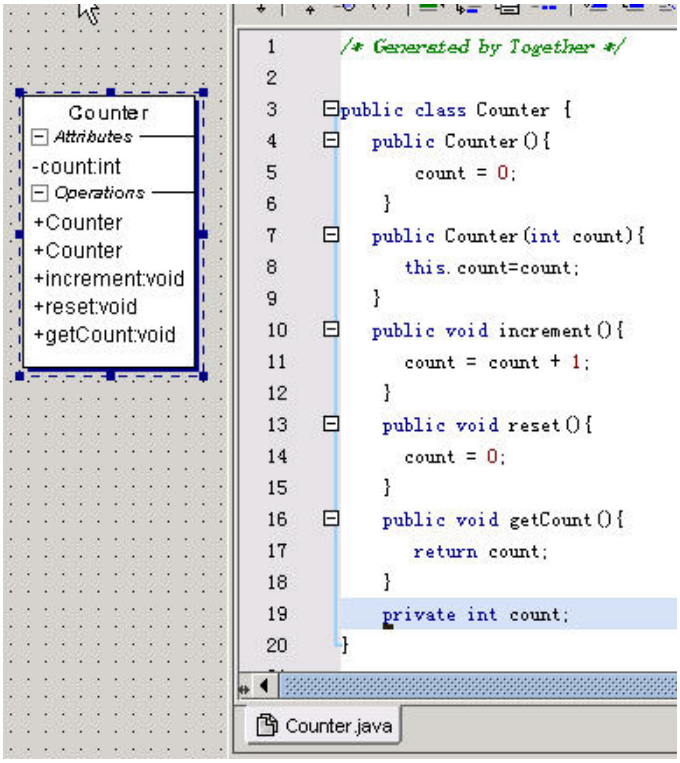


图 15 待测试的 Counter 类

2、其二为一个里程表类 Odometer，它在到达最大值时会从头开始计算里程，如图所示：

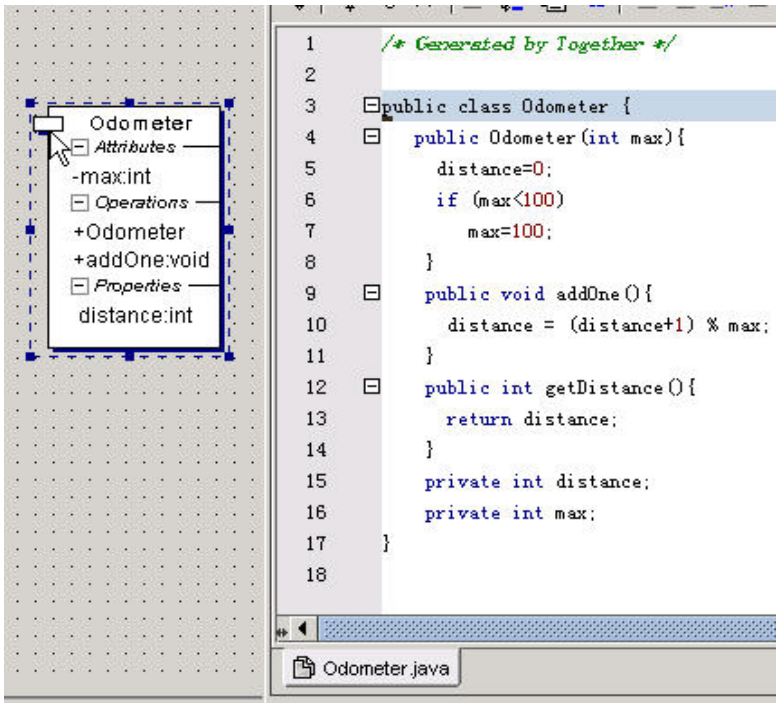


图 16 待测试的 Odometer 类

第二步：点击浏览器窗格（Pane）中的“Tests”标签页，点击下方的“Choose Test Plan Path”按钮，选择测试计划的路径。在 Together 中，测试计划是包含测试资产和测试结果的逻辑集合，这里的测试资产包括：测试步骤（TestStep）；测试套件（TestSuite）；测试数据（TestData）；测试集合（TestCollection）。

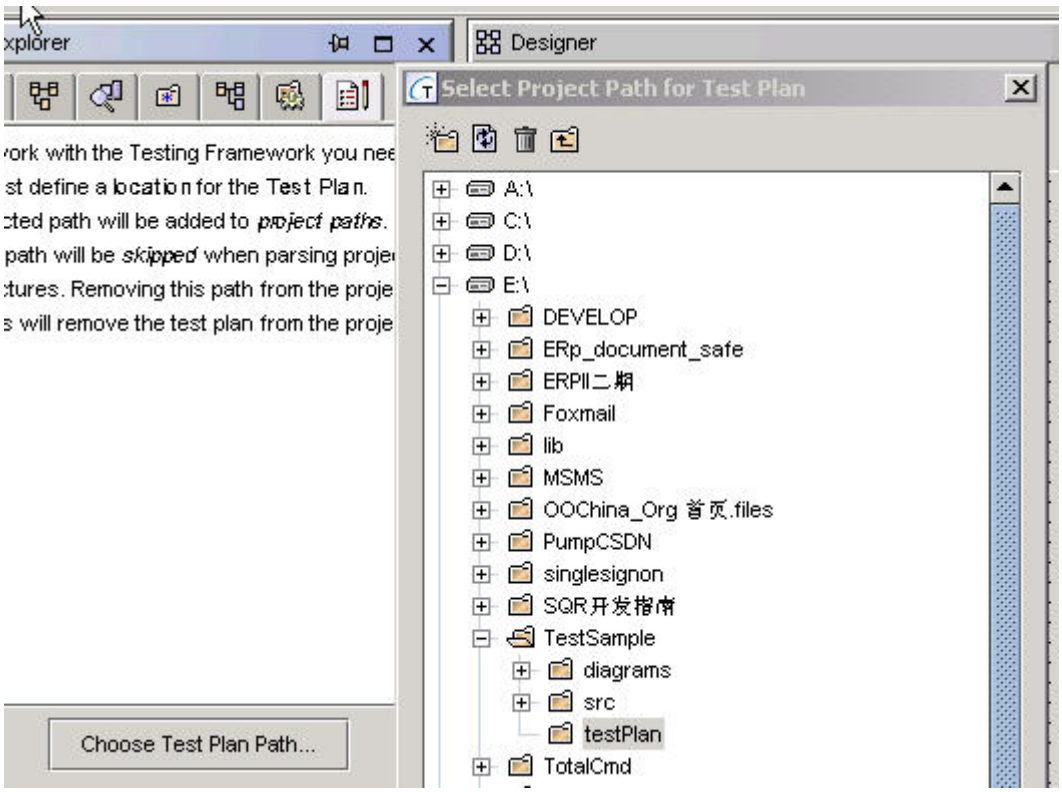


图 17 为测试计划选择测试路径

第三步：创建一个测试套件（TestSuite）:在“Tests”标签页的“Test Plan”节点上右击，选择弹出菜单“Create Collection Suite”

图 18 创建测试套件

第四步：接下来分别为每个类创建 TestCase，以便加入到上一步所创建的测试套件中；

1、右击 Counter 类，从弹出菜单中选择“Test->JU Generate JUnit->Default TestCase”，这将为你产生测试用例的骨架程序代码，如下图所示：

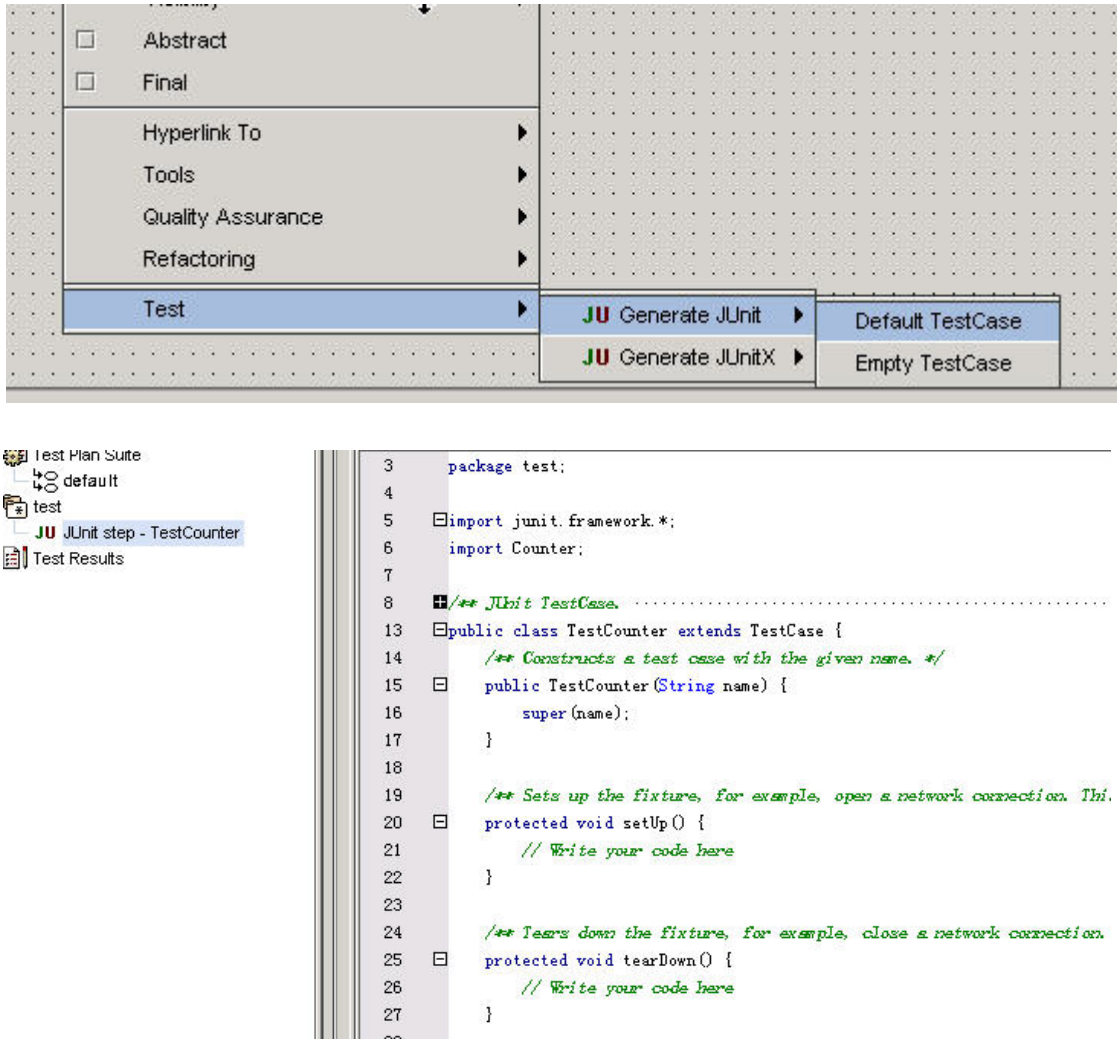


图 19 为 Counter 类生成骨架测试用例代码

2、依法炮制，为 Odometer 类创建 TestCase；

3、编辑 Counter 类的 TestCase：

1)、加入“Count c;”声明；

2)、在 setUp()[1]方法中添加初始化代码

关于 setUp()方法等的原理性说明，请参考 JUnit 网站的文档，也可以到 www.oochina.org 上去参考相关的中文翻译文档。

```
c = new Counter(5);
```

3)、调用 increment()方法以测试 increment()方法的正确性

```
c.increment();

assertEquals(6,c.getCount());
```

4)、下一步故意让 testReset()测试方法失败

```
c.reset();

assertEquals(1,c.getCount());//因为此时 count 的值应为 0
```

整个代码如下图所示：

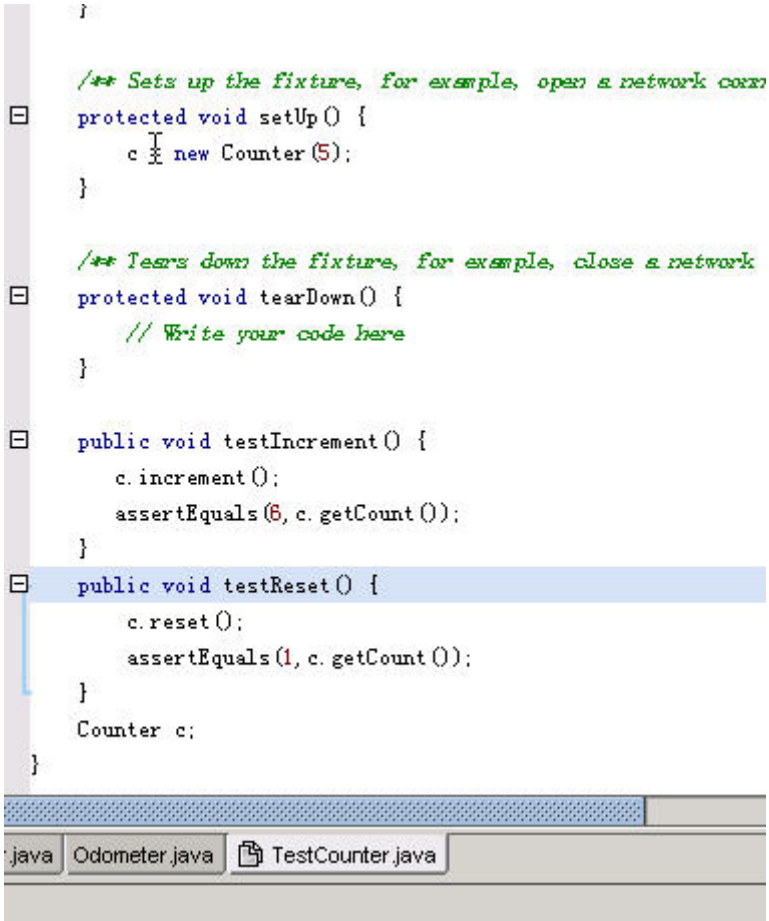


图 20　为 Counter 类的骨架代码中加入相应的测试代码

4、编辑 Odemeter 的 TestCase

1)、加入“private Odometer d;”声明；

2)、在 setUp()方法添加初始化代码

```
d = new Odometer(100);
```

3)、为 addOne()方法添加测试代码

```
d.addOne();

assertEquals(1,d.getDistance());//按照设计，这一步应该成功
```

整个代码如下图所示：

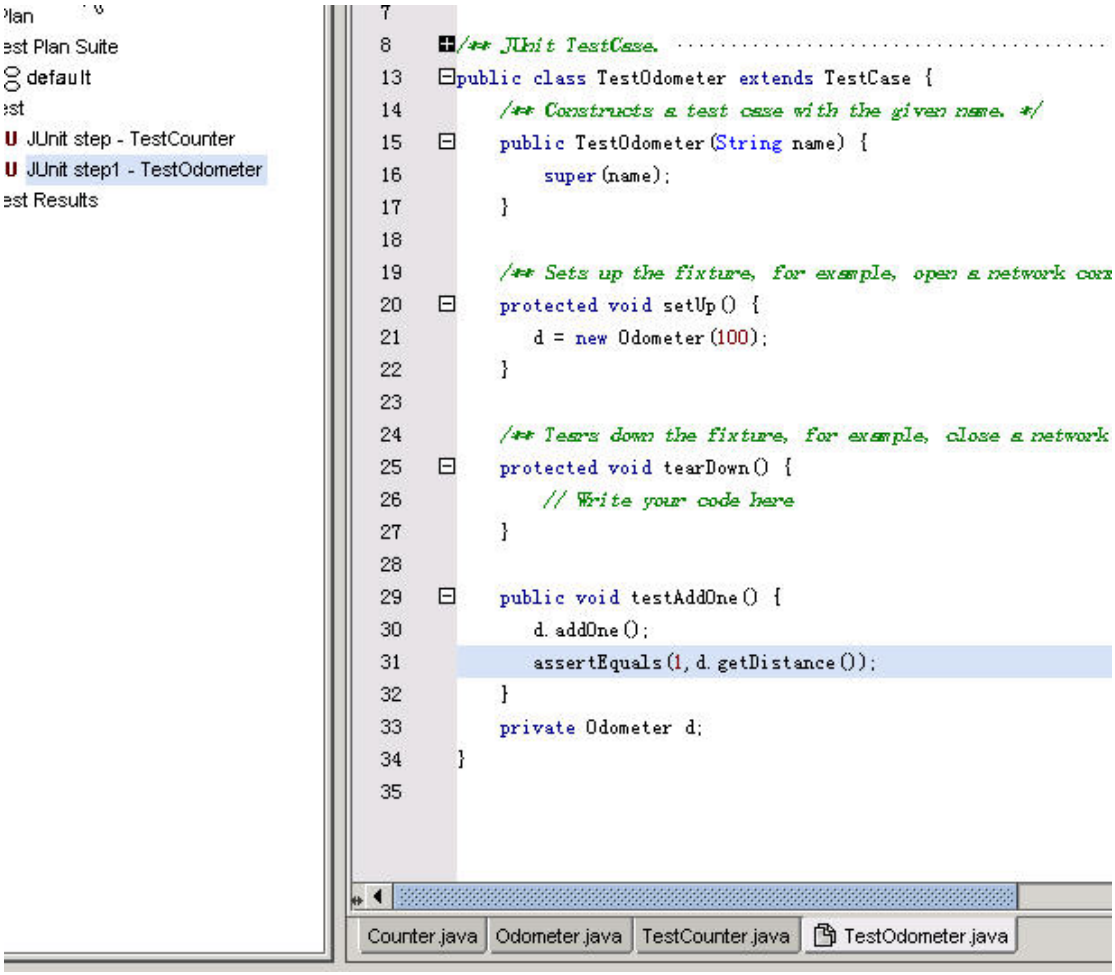


图 21 为 Counter 类的骨架代码中加入相应的测试代码

第五步、将上面创建的 TestCase 添加到 default target 中：从 default 的弹出菜单中选择“New->Step Reference”，从弹出的对话框中选择刚才创建的两个 TestCase（选择时没有先后顺序），如下图所示：

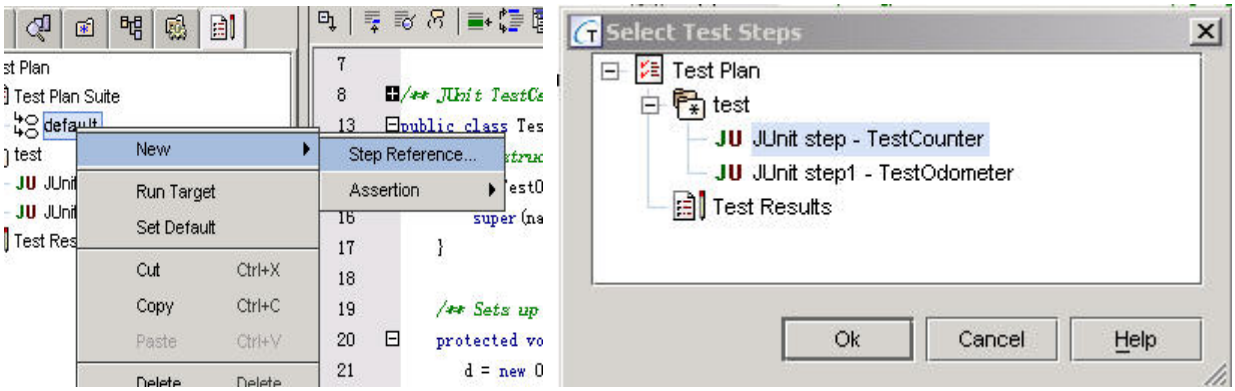


图 22　将 TestCase 中加入到 default target 中

第六步、在 default target 上右击，从弹出菜单中选择“Run Target”来运行此测试套件：

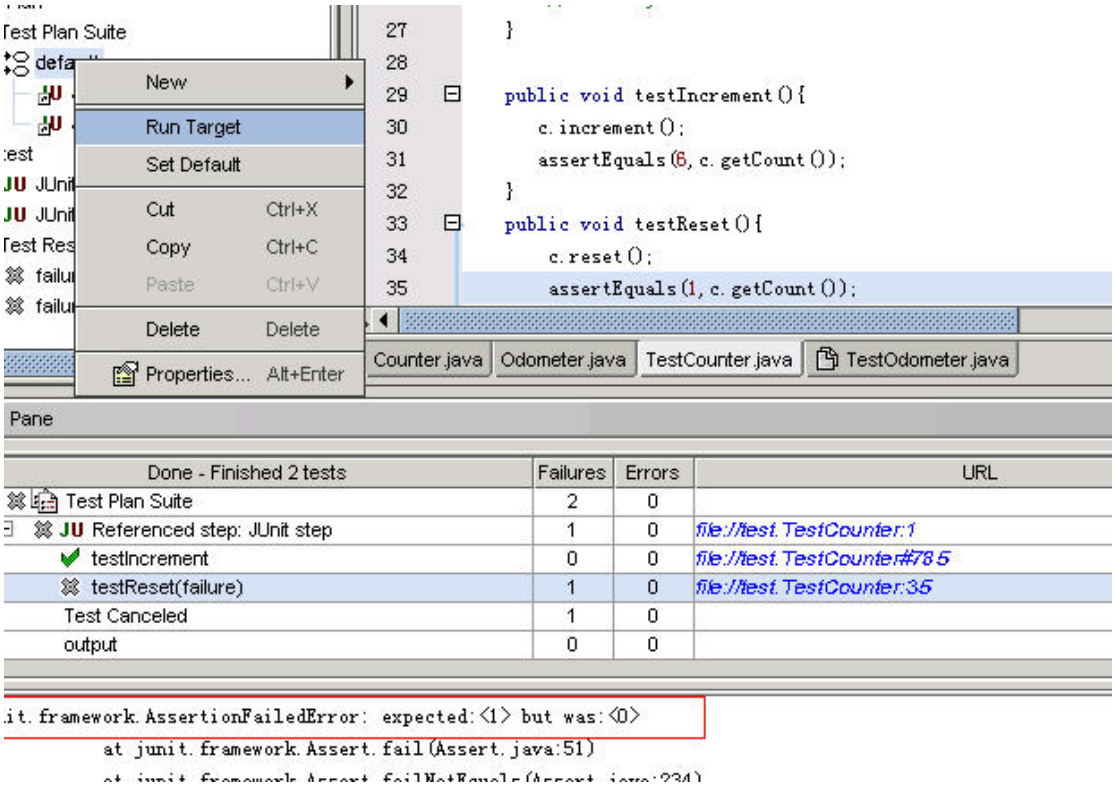


图 23 第一次运行测试套件

大家可以看到，在测试输出结果中，显示了两个失败（failures），选择其中的一个失败，下面可以显示其详细情况（程序期望为 1，但实际值为 0），并将失败的代码高亮显示出来

第七步、让我们修正这一错误，将 testRest()方法中的断言修改为 assertEquals(0,c.getCount());再次运行测试，这时可以发现前面的失败已经被修正了，还有一个错误，如下图所示：

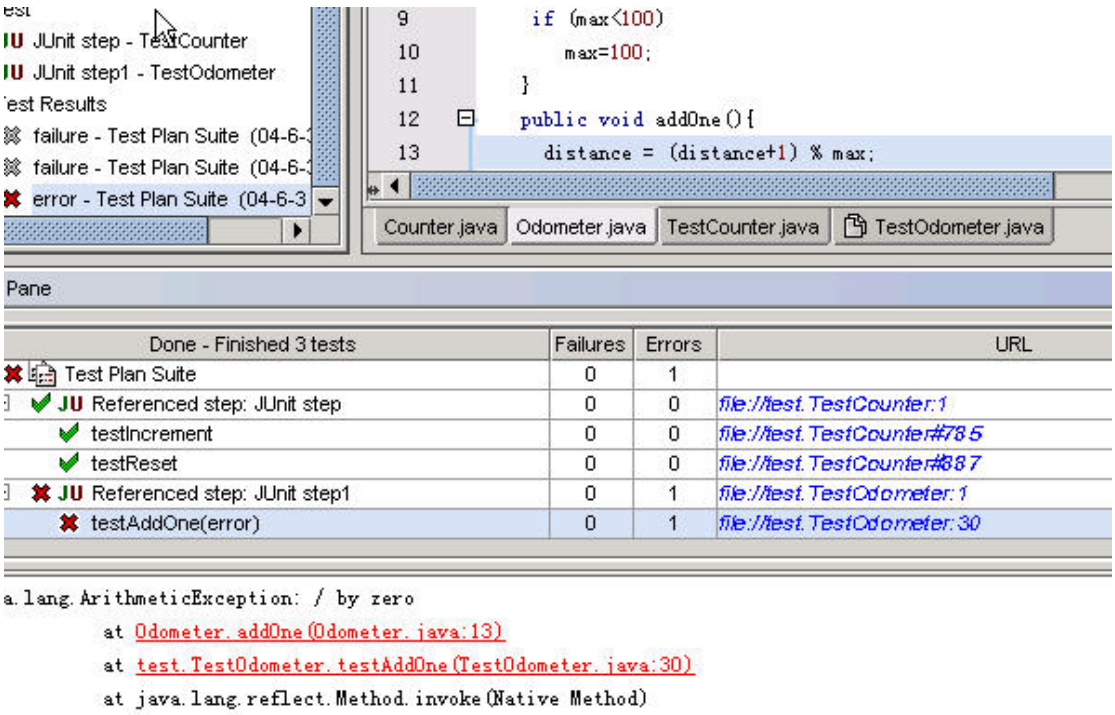


图 24 再次运行测试套件

按照指示，可以发现这个错误是被零除错误，找到对应的代码，经过分析发现是因为 Odometer 的构造器有逻辑错误，没有在不同的输入值时都对 max 属性进行初始化，将 Oometer 的构造器修改为如下所示代码：

```
public Odometer(int max){
    distance=0;
    if (max<100)
  this.max=100;
    else
  this.max = max;
    }
```

第八步、再次运行测试套件，这时你可以发现整个测试案例成功运行，如图所示：

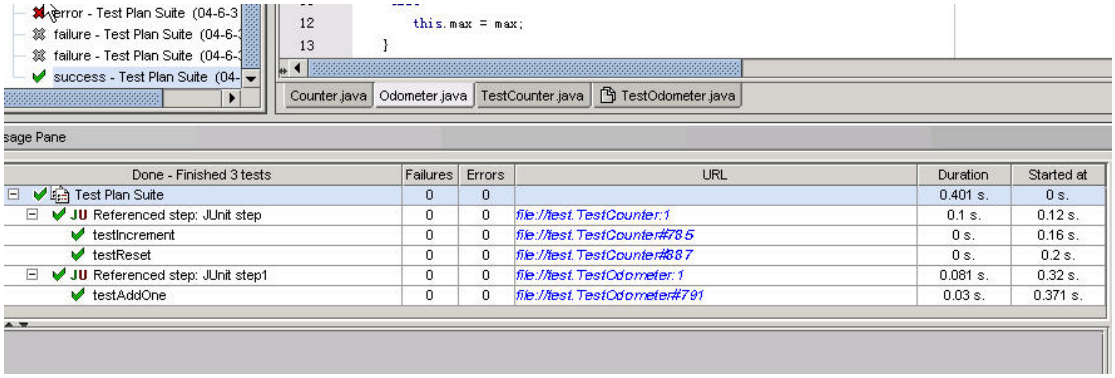


图 25 最后一次运行测试套件（成功）

至此，这个简单的例子就结束了，大家可以看到，在 Together 中进行单元测试是非常简单的，从而有力地支持了重构活动（感兴趣的朋友可以和我联系索取完整的 Together 项目文件）。

不仅如此，Together 还提供了其他测试框架，如 JUnitX（JUnit 测试框架的扩充。），Cactus（Cactus 是一个用来对 servlets、EJBs、标记库（taglibs）和其他服务端 Java 代码进行单元测试的框架。）等，有兴趣的读者可以参考 Together 的用户指南。

审计与度量驱动的重构

审计和度量是 Together 提供的独一无二的特性，它们可以用来帮助用户强制执行公司标准和惯例，获得真实的代码统计，并改进质量，其中审查(Audit)用于检查代码是否符合组织定义的风格和标准；统计(Metric)则站在更高的抽象层次，为开发小组提供了测量软件项目复杂程度、质量与规模的能力。

在此，笔者不想讲述如何去进行审计和度量，读者可以参考相关资料去进行，而是说明我们可以使用审计和度量这两个有力手段实现“审计与度量驱动的重构”。

为什么这很重要呢？读过 Martin Flower 的《Refactoring _Improving the Design of Existing Code》一书的人都知道，作者用一章的篇幅描述了代码中存在的“Bad Smell”，作为判断是否需要进行重构的“事实根据”。可是，“懒惰”的开发员不禁要想，这样去按照标准一个一个去判断，还是太过麻烦，特别是在团队开发环境下，如何去推行代码标准呢？虽然有 CMM 强调的代码评审和 XP 强调的结对编程作为流程上的保障，但如果有这么一个工具能自动替你发现那些不如你所愿的 Bad Smell，世界将更完美！

Together 提供的审计和度量就可以帮你做到这些，从而实现“审计和度量驱动的重构”。在一个已经结束的应用上运行审计和度量可帮助你揭露出软件中需要重构的地方。如果在软件开发过程中运行它们，效果将会更加明显，它可以帮助你强制性地建立最佳实践。

软件审计的种类包括：代码风格；多余内容；特定计算机语言的错误用法等。大多数审计（在《effective java》一书中提供了很多有关审计的例子。）可以通过自动化的工具来帮助完成。Together 就提供了大量的审计规范，并可以使用 OpenApi 撰写自己的审计。

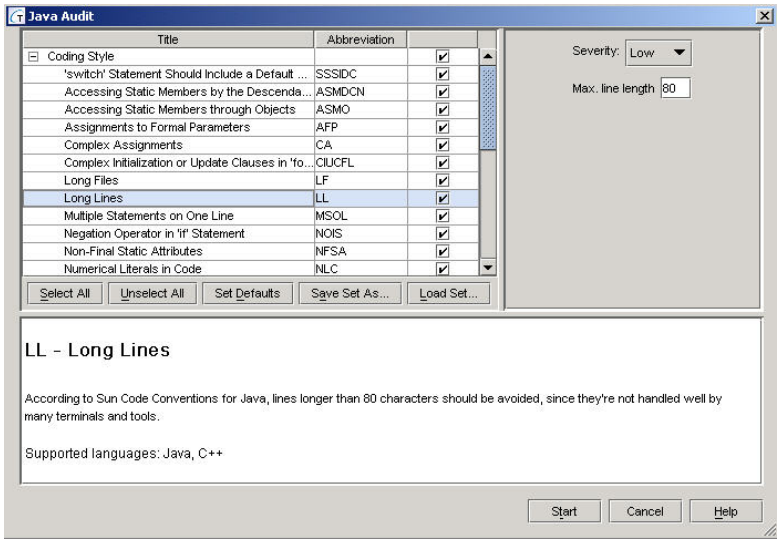


图 26 Together 提供了丰富的审计

如上图所示，就显示了在 Together 中执行审计时的情形，对话框的下部显示了针对每个审计的描述性信息。点击“Start”按钮，即开始审计过程，审计结果可以保存。

度量分析则集中于分析面向对象编程的已知原则，包括：

1)、内聚性（cohesion）；

2)、耦合性（coupling）；

3)、复杂性（complexity）；

4)、继承（inheritance）；

5)、多态（polymorphism）；

6)、封装：encapssulation 等等

如果对度量进行分类，那么可以分为：内部（源代码）；外部。Together 提供了 55 个度量规则，也可以使用 Borland OPENAPI 进行客户化。

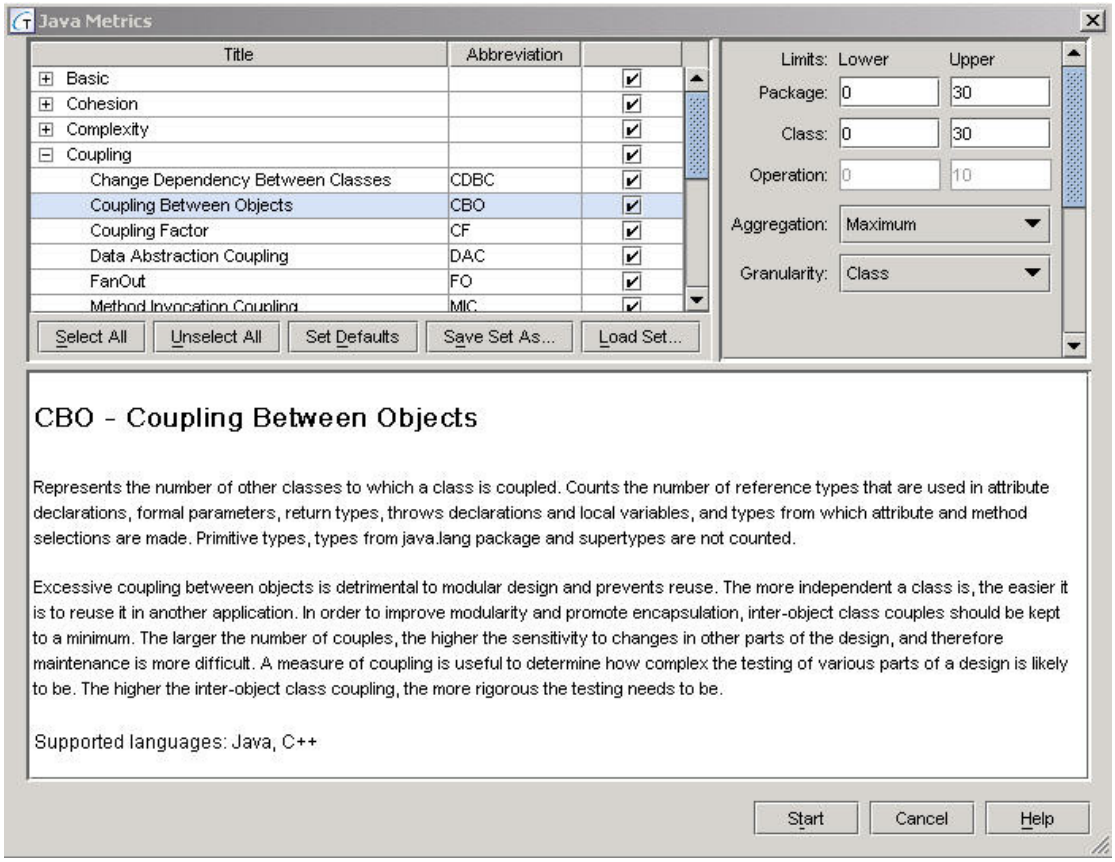


图 27 丰富可用的度量

上图就显示了在 Together 中进行度量的情形，对话框的下部显示了针对每个度量的描述性信息，点击“Start”按钮即可以开始进行度量工作。度量分析的结果也可以保存输出。

度量分析的结果既可以以表格的形式显示，也可以以图形的方式显示，当你希望查看某个度量项在多个类或包中的分布情况时，使用条形图和分布图；当希望查看多个统计项在某个类或包中的分布情况时，使用 Kiviat 图。

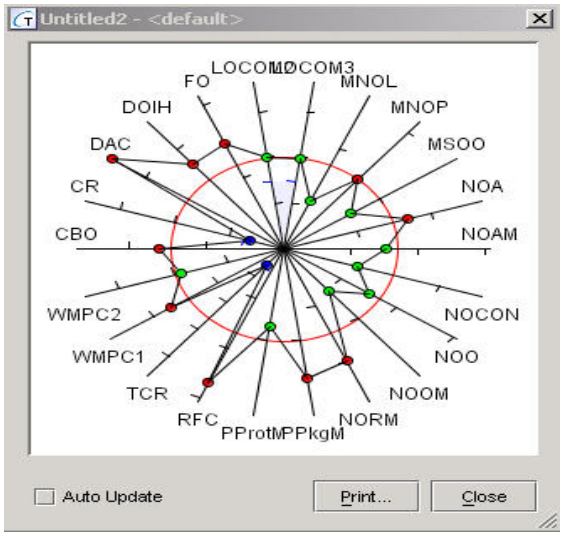


图 28 图形化的方式直观地显示了度量的结果

例如上图即显示了 Kiviat 图的情形，红线外的度量结果即表示其结果超出了规定的范围。

当前，能提供重构能力的开发工具已经不是什么稀奇事了，当面对的是一小块代码进行重构时，开发员不会觉得有多困难，然而，当面对一个大型程序，并且可能还不熟悉其代码的时候，程序员可能就不知道如何下手了。

审计和度量可以驱动重构，而 TCC 就提供了这么一个环境。

重构是一把双刃剑，永远记住一句古老的谚语：“不要想一口吃成个胖子”，而要每次一小步，在重构之前和之后都进行单元测试可以验证功能的正确性，而在重构之前和之后运行审计和度量(度量结果可以保存，这样你就可以在重构前后对同一个目标进行度量，以观察比较其变化。)则可以帮助你确认你的重构是否作对了方向。

这样，当采用审计和度量驱动的重构时，整个工作流程就变成了：

1)、运行单元测试，审计和格式化工具；

2)、针对代码进行度量，并分析度量结果；

3)、检查冲突区域，查找问题区域；

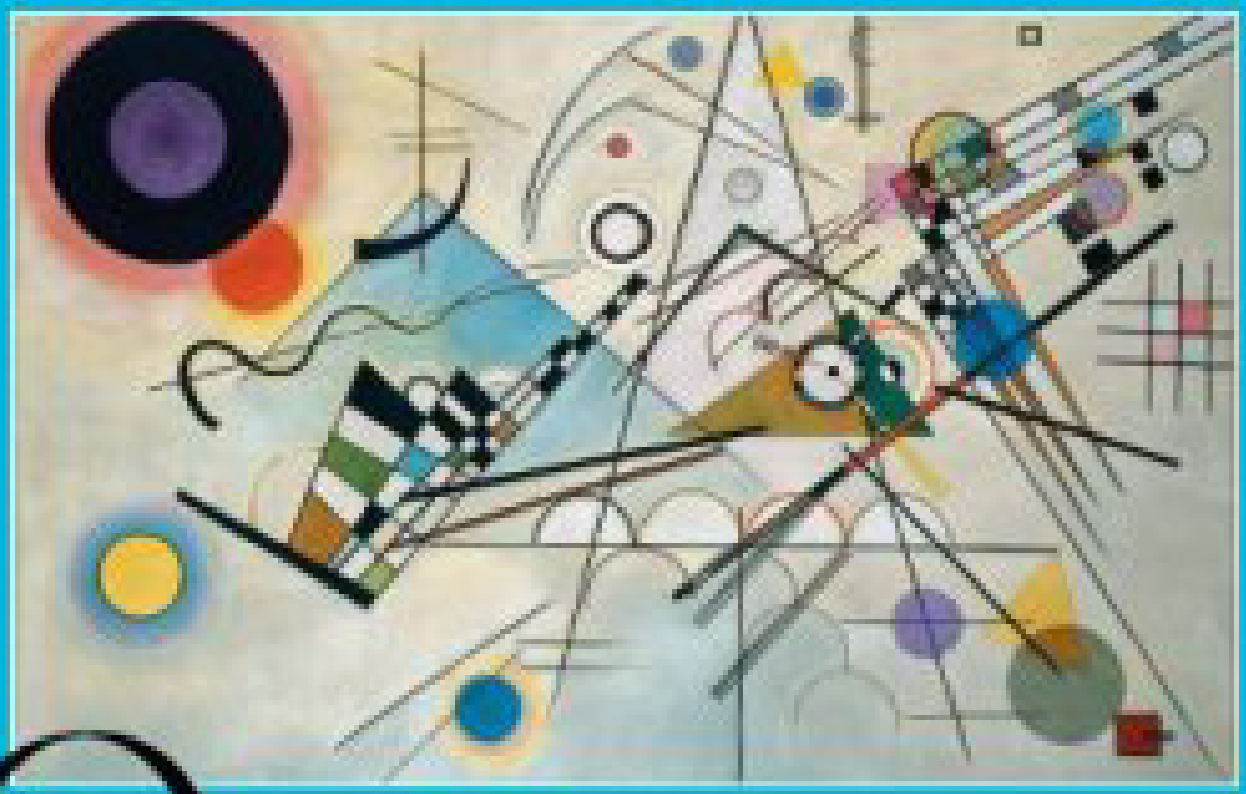
4)、重构代码，如果可能应用设计模式；

5)、再运行审计，度量和单元测试，比较前后变化以进行确认。

在实际工作中，开发员还需要摸索出度量与重构的组合使用最佳实践，比如判断在进行哪个度量后可能需要采取的重构动作，限于篇幅，这里就不举例了。（本文为 Borland 中国公司活动征文，经 Borland 中国授权刊登）

Domain-Driven DESIGN

Tackling Complexity in the Heart of Software



Eric Evans

Foreword by Martin Fowler

《领域驱动设计》中译本

即将由清华大学出版社出版，UMLChina 审稿

软件工程技术丛书

设计系列

企业应用架构模式

Patterns of Enterprise Application Architecture

(美) Martin Fowler 著

王怀民 周斌 译

UMLChina 审校

UMLChina 训练辅助教材

利用模型驱动架构加速嵌入式软件开发

PathMATE 著，sunpengsun 译

讨论

介绍

嵌入系统开发人员在面对需求变更、复杂并且有时脆弱的软件架构和不断进化的平台时，肯定遇到过交付日期紧张的问题。为了解决这个问题，一些机构已经在使用统一建模语言(UML)或者更早的方法如Shlaer-Mellor的应用建模上进行了投资。在很多人期盼着UML建模来改进需求的获取、流线系统的设计和增加组件的重用的时候，实际上它的好处经常被局限于简化设计文档的编写和讨论。

从对象管理组织（Object Management Group®）(www.omg.org)进入模型驱动架构（MDA）。由软件开发专业人员协会创建的MDA提高了你在建模工具上投资的回报。这份白皮书帮你理解MDA是什么、采用它有多么好和在安装了例如PathMATE的自动建模和转换工具时，它能提供的益处。益处包括：

- 更快，软件交付周期的可预见性提高
- 需求变更对开发时间表的影响降低
- 组件重用性和执行兼容性增强
- 架构的更灵活和平台无关性
- 还有其它…

MDA 定义

MDA是一个用来为软件系统建模的标准框架. 当你设计一个符合这些标准的模型时，你获得并且描述嵌入系统的对象、属性和操作。当你使用MDA自动生成和转换工具处理模型时， 你做到了：

你的应用程序的自动操作和执行，为了测试和验证——在写任何代码之前。

模型到测试过的，可配置应用的自动转换。

MDA 通过将系统必须做什么和它是如何在一个特定的技术平台执行的分离达到这些目的。MDA 由两个部分组成：

- 平台无关模型（*Platform Independent Model*，PIM），指定系统要做什么。
- 平台特定模型（*Platform Specific Model*，PSM）， 指定系统如何被执行。

PIM 获取系统的重要特征或者业务逻辑。PSM 确定PIM怎样在目标配置环境上执行。PSM可以被用于多种形式，包括可执行代码，例如C，C++或Java。MDA工具如下一部分说明的那样把PIM转换为PSM。

用于嵌入式系统的 MDA

MDA特别适用于嵌入式软件开发，是因为它把功能逻辑从执行细节分离出来，并且自动生成和测试任何嵌入应用架构都只使用MDA技术。MDA为嵌入式软件开发人员提供一个根本不同的、高水平的方法来适应变化的需求，提高重用率和扩展系统寿命。

PIM向可执行的PSM的转换是脱离环境自动进行的，但仍是可以定制的基于模板的转换技术，如PathMATE。 图1用PathMATE为例说明模型的构造、转换和验证。

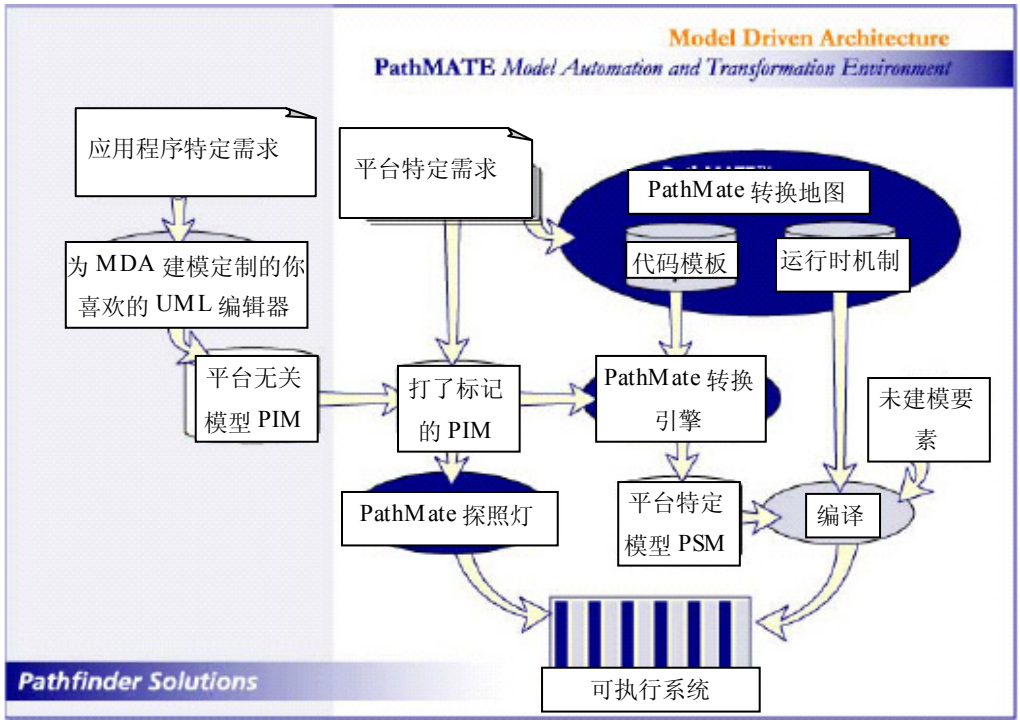


图 1 MDA 工作流程和关键技术要素

下面的步骤解释程序是如何工作的：

1. PIM在支持的几个UML编辑器中构造，例如IBM Rational Rose，它有可以用来和PathMATE集成的自定义菜单项。

2. 一套平台特定标记可以随意地指定到PIM的基础上。

- 标记包含一套属性和立体类型，用来引导转换规则和优化方法，例如生成单线程还是多线程的应用，向有资源限制的配置目标生成多少调试代码。
- 因为标记的存储是和PIM分离的，你能够通过把不同的标记应用到同一个PIM来产生不同的PSM.

3. 完成PIM的扩充后，开发人员调用PathMATE转换引擎。引擎读取PIM并且给它应用一个脱离环境的或者自定义的PathMATE转换地图，这个地图为目标平台代码的生成提供导向。PathMATE带有用于这几种目标语言的转换地图：

- C
- C++
- Java

4. 地图包括一套模板和运行时机制：

- 模板指定了把PIM转换成可执行代码的规则。
- 运行时机制是一套实现生成的代码要求的效用的实现。

模板读取平台特定标记来决定何时应用不能直接从PIM得到的优化或者转换规则。

5. 引擎从标记过的PIM和模板生成表现PSM的源代码。

6. PSM和运行时机制，还有任何未建模的代码（包括脱离环境组件和手工编写的代码）紧密结合来组成可执行系统。

7. 然后用PathMATE Spotlight来调试和测试PSM。

为了早期的、反复的测试，Spotlight在开发环境中执行PSM，或者在目标硬件上为了迅速分离出行为和性能相关的缺陷。

为什么用 MDA?

当你把MDA标准、细心抽取的PIM和自动转换技术应用到你的嵌入式系统开发中，你就从开发过程中除去了固定的下级编码和测试。众所周知，更少的人工编码和更早的隐错检测能够大大提高按时并且不超预算地交付一个高质量系统的概率。你也继续紧密控制你的工程师开发的嵌入系统的开发过程和质量。

对需求变更快速反应

PIM从PSM的分离允许你不必改变PIM就能执行需求，从而对变更快速反应。例如，如果你需要在一个新的平台上进行配置，你不需要改变PIM——简单地应用一个不同的地图就行了。如果你需要改变处理的拓扑，你只需调整模型标记。如果你需要应用新的优化，地图模板和机制是很容易定制的。如果功能性的需求改变了，你可以在PIM级整合新的特征。

充分扩展系统寿命

维护一个依赖于平台的系统时，初始的架构可能不再能满足新的需求。与花时间重新构建相比，大多数团队只有时间应用补丁和错误修正，而这会导致架构变得脆弱。使用MDA， 功能和架构是分开定义的，并且架构的改变是通过转化自动地执行的。架构和功能能够发生根本的变化——与别的无关，这使大多数MDA系统的生命周期得到扩展。

提高开发者的生产力

最好的实践建模技术把系统划分为高度相关的组件。这个划分对MDA来说是基础的，并且简化了各系统组件，正是这些组件始终如一地产生易于建立、配置、重用和维护的实现组件。引擎自动从模型产生高质量的和完全实现代码。开发人员能够把精力集中于在PIM设计另外的客户驱动的功能，或者建立和扩展地图和优化。

使平台无关模型（PIM）大规模重用成为可能

不同的地图和设置可以被应用到相同的PIM，以便在不同的应用语境中产生多种组件的实现。这样PIM就能够在多个系统重用。

更低的维护成本

因为编码是由模型生成的，你知道你的模型和代码总是保持同步。因为拥有一个可靠的、高水平的系统视图，系统的新开发人员能够快速跟上。

简化文档编制任务

当软件变更时保持设计文档随时更新是单调乏味的和强实时性的。使用MDA，模型、代码和文档总是保持一致。PathMATE文档地图生成包含模型及其相关描述的定制文档。

减少质量保证成本

在开发过程中，软件错误发现的越晚，修复它的代价就越昂贵，交付期变得更具危害。MDA模型自动操作和测试工具，如PathMATE Spotlight，帮助开发人员在编码以前，在模型级测试他们的应用程序。结果，设计缺陷和应用逻辑错误在开发过程的上游就被揭示出来。此外，Spotlight能够在目标硬件上自动操作和测试模型来帮助在过程中较早地揭示平台特定的问题。

提高质量

PIM本质上的简单带来系统质量充分的提高。 建模帮助提高成员间的交流和促进缺陷的尽早消除。引擎自动对模型应用代码的模式消除了手工代码产生的缺陷。

开始使用 MDA

通常应用高效的软件工程技巧和技术的最大障碍不是技术的，甚至不是资金的。即使使用标准，例如基于已证实的技术MDA,对进展的重大阻碍还是会从管理和文化要素产生。要推动MDA的应用,Pathfinder解决方案建议下级楔入以减轻应用新技术的风险：

确定你唯一的问题

- 描述出你面对的软件开发的主要挑战。
- 和专业人员一起确定一份这些问题的详细诊断。
- 确定MDA技术和加工在这些挑战的哪些方面有效。

建立解决方案

如果你的关键组织目标向MDA的益处看齐，MDA技术和工具能有效解决你的最大问题：

- 为你的开发环境获得恰当的MDA工具：
  - 与现存的UML工具和基础结构整合。
  - 支持你的扩展和配置语言以及平台。
  - 进展迅速、可注册和易扩展的转换技术。
  - 尽管是受资源限制的，在目标平台测试模型仍然能够实现。
  - 工作无缝隙跨越模型和未建模系统组件。
- 安全的支持，提供专家和方法来帮你确认符合你特有需要的MDA解决方案要素。Pathfinder解决方案已经与多个成功案例讨论，能够帮助你的团队建立一个成功的配置平台。
- 考虑技术和文化的障碍，用一个步进式的绪论设计一个管理复杂问题和风险配置计划。

配置

一旦你已经确定配置MDA这一英明决策,执行：

- 用Pathfinder的用于MDA/UML的方法指南训练来培训团队。
- 通过引导员的指引或者通过关注应用程序片断的MDA配置管理最初的使用范围。
- 通过Pathfinder的专门的从业者为你的职员作顾问、建立有效的建模技术、提供鉴定反馈和帮助你的队伍避免常见的昂贵缺陷来减轻技术风险。

改进和扩展

- 在你初始成就的经验基础上，改进你的解决方案。
- 扩展配置以便在更大范围收益。