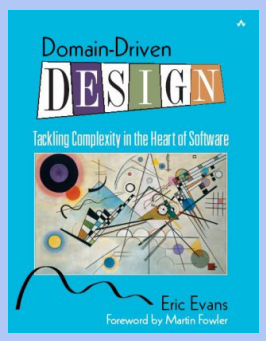
第 39 期（2004 年 7 月）



领域驱动设计

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/

建模工具领域的 IBM 和微软对决

[2004/7/24]

背景：德克萨斯的狭长地带，上周末，IBM 将微软招呼出来开始一场老式的决斗，随着它的“Atlantic”工具集的面纱被揭开。

Atlantic 是 IBM 软件开发平台的下一个版本代号。IBM 计划在设计、建模、开发、测试、部署以及应用维护的组件间实现深度集成。Atlantic 承诺将和 Eclipse 的开源开发平台更紧密的结合，并给予 UML2.0 更多的支持。

IBM Rational 工具集和微软的 VSTS(Visual Studio Team System)的关键区别在于建模的途径。IBM 严格遵循 OMG 提出的统一建模标准 UML，而微软认为 UML 不是必须的。微软 VSTS 主管，Rick LaPlante，认为，对大多数开发人员而言，UML 表示的模型太过复杂，不必要，这不是微软想要走的路。“从现在起的十年内，建模将不再是软件组织中少数人的工作，也不会说专门成立一个特别的部门，里面的人只负责建模，我认为，建模将变得非常普遍”。

IBM 官方宣布，今年底 Atlantic 问世。微软的 VSTS 将在明年中旬发布，还包括 Visual Studio 的“Whidbey”版本和“Yukon”数据库。

专家认为，微软给自己打上的标签是迎合大众、让开发更加容易。其 Team 系统试图为更广泛的用户寻找解决方案，而 Rational 则在解决复杂度上声誉在外。来自 ZapThink LLC 的分析师认为，“UML 的学习曲线较陡，这导致它更适合于经验丰富的架构师和高级开发人员，而 UML2.0 则更加复杂，虽然在 UML 里面还是不完全地定义软件功能。相反，微软在 Visual Studio 方面表现出来的趋势始终是：其工具对初学者来说足够简单，对高级用户来说足够强大”。

IBM 的 Fellow 和 UML 的创始人之一 Grady Booch 认为微软没有遵循标准。“这是一个长期的、可以支撑的战略吗？很难说，不过 IBM Rational 正把赌注压在开放的市场上，主要是因为这方面的应用很多，在交互性方面的要求还有更多。”

但是，最后有人认为有迹象表示两种趋势的合龙。压力导致微软要去支持企业化的应用开发，而 Rational 也在集成和简化他们的解决方案。McGraw-Hill 公司的软件架构师，Cort Bucher，微软和 Rational 产品的共同的用户，认为他已经看到了这种转变。Bucher 说，“用 UML 来表示解决方案架构，这些设计和建模对项目有好处，我们当然要用 UML，但用 UML 建模有几种方法，包括在 Visio 里面或者在 IBM Rational 的工具集里面。UML 在可视化表示应用方面非常有帮助，至于说它是不是 must-have(必须的)，我认为这取决于系统的复杂度和公司对文档的要求”。

（自 eweek，UMLChina 袁峰 摘译，不得转载用于商业用途）

Rational 揭开其工具集升级版的面纱

[2004/7/21]

IBM Rational 向开发人员展示其代号为 Atlantic 的工具集的下一个主要版本，该版本将在今年晚些时候发布。

IBM 软件集团 Rational 主管 Mike Devlin 在 Texas 的 Rational 软件开发者大会上宣称，其工具的新功能将支持 UML2.0 以及 JavaServer Faces (JSF)，后者提供了交互式 Web 页面的构造组件。

Rational 软件的 CTO，Lee Nackman，演示了下一个软件版本，它将和 Eclipse 3.0 开源工具环境集成。IBM 还宣称，在其 Atlantic 发布版中将集成测试和监控环境。

Atlantic 将包括服务数据对象（service data objects），它将帮助开发人员连接应用和数据库，并提供到 WebSphere 的实时部署。另外，还会有一个新的 ClearCase 客户端辅助改进基于团队的开发。

Devlin 介绍，无论从业务、开发、操作和部署的角度，IBM Rational 都将“把它们融为一个单一的过程，以加速开发生命周期。”

为 IBM 提供的开发工具集将成为 IBM 软件开发平台的部分，其中包括 WebSphere Studio Device developer 5.7，它提供给开发人员将应用从桌面拓展到移动设备和便携电脑的能力，将在 6 月 30 号上市。

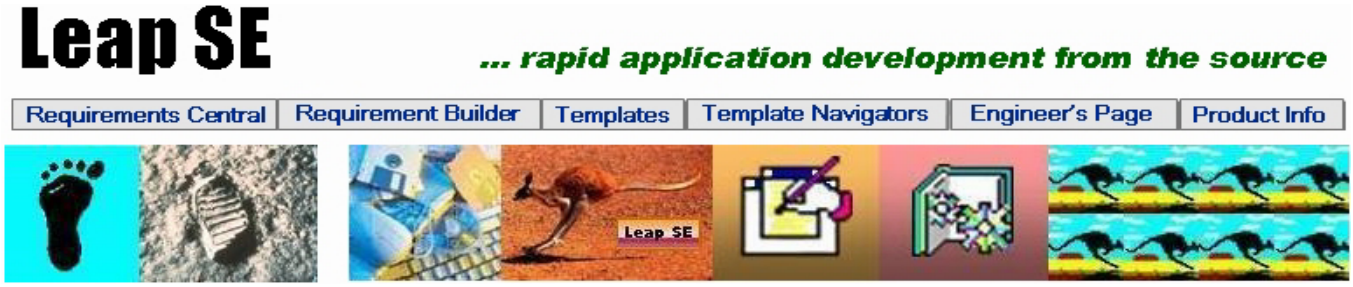
其中同样还将包括的是 WebSphere API 工具集。提供给业务用户的 Workplace Builder 以及企业级别的脚本工具 Workplace Designer 将可以用来创建单独的业务应用。

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

Leap SE 将自然语言直接转换为对象模型

[2004/7/19]

Leap SE 是一种将系统级需求翻译为对应系统的逻辑模型的专门的 CASE 工具。该产品不仅是一个描述“将会发生（shall）”句子的仓库（系统需求规约 SRS（System Requirements Specification)正是由这些 shall 句子组成），同时也为直接从这些需求生成面向对象的头文件提供了一种方法。



通过将自然语言（英语）表达的业务规则转换为一族类、属性、方法、关系以及继承层次，Leap SE 提供了一种“飞跃（leap over）”系统工程中这个重要部分的方法，这个部分对于任何一个软件开发项目都是如此的关键。

每次当一个新的需求保存的时候，Leap SE 的对象模型数据库就自动更新以反映新的实体、关系、属性以及方法。从这个数据库中，可以在任何时刻生成一个头文件目录，它将为软件工程师提供一个设计的良好开始。另外，只需要简单的一步，将这些头文件导入到一个支持反工程的 CASE 工具中就可以快速地得到一族类图。

基于 UML，Leap SE 使用了名为 Deterministic Phraseology 的独特方法，这种方法：

- 输入需求就可以得到模型，无需任何额外操作。
- 自动将自然语言（英语）描述的业务规则转换为逻辑对象模型；
- 强制要求需求的格式为可创建的以及可测试的；
- 暴露系统需求中可能存在的不一致以及模糊不清；

其功能包括：

- 21 种模版，可快速简单地进行组合；

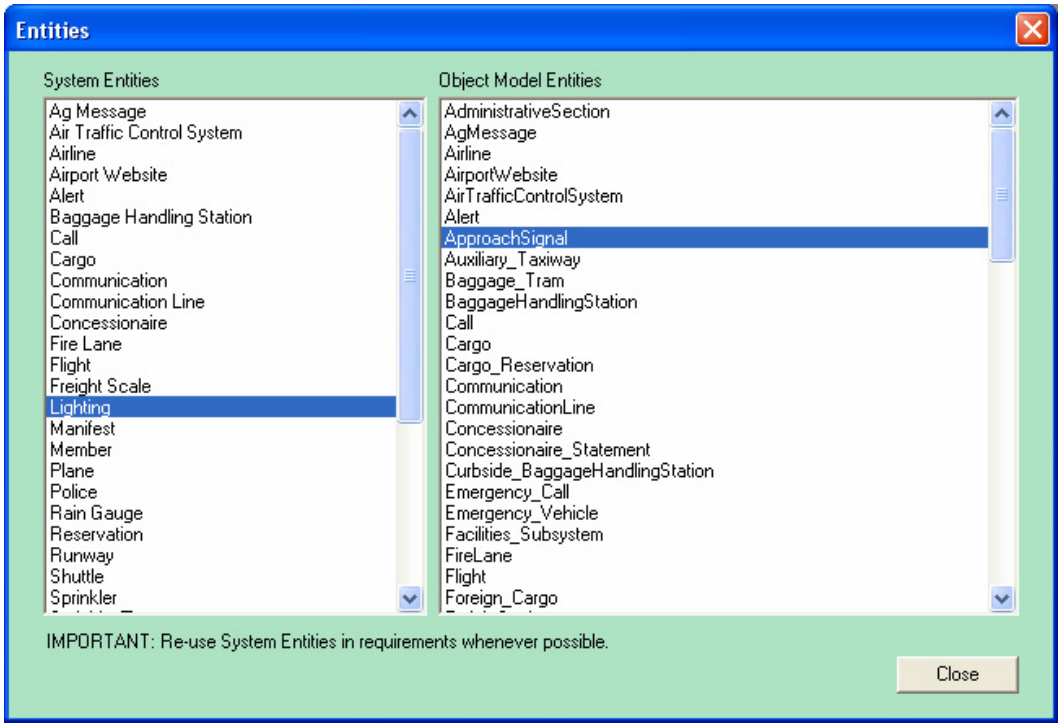
- 创建复杂需求的构造器；

- 伴随每次需求保存的模型增长；

- 从 SRS 的即时对象模型生成；

- 集成的查询和模板示例；

- 为系统规约提供的完整仓库；



公司宣传称：拥有 Leap SE，项目经理可以把他们的系统工程师削减一半，同时还可以保证需求的质量。提供 21 个模板和一个需求构造器以生成快速和灵活的组合。

欲下载免费 demo，获取产品信息，或索取免费试用版，请访问 http://www.leapse.com。

（自 prleap，UMLChina 袁峰 摘译，不得转载用于商业用途）

IBM 为 Rational 打开 Hyades 大门

[2004/7/7]

IBM 公司正在一个 Eclipse 的开放源码框架上标准化来自 Rational 软件的软件质量工具，同时增强和第三方产品的交互性。

IBM 昨天宣布，Rational 的自动软件质量 ASQ(Automated Software Quality)工具集将会基于 Eclipse Hyades 项目或者改进以和 Hyades 交互。

在软件质量这块招牌下，集中了 IBM Rational 的 for Java 和 Web 的 Functional Tester、Robt、Performance Tester、Team Unifying Platform、PurifyPlus、Rose XDE Developer Plus 和 Test RealTime 等工具。IBM 希望今年下半年可以在 Rational 工具集的下一个主要发布版本中 Hyades 可以起到重要作用。

IBM 是 Eclipse 和 Hyades 的奠基者。IBM 昨天宣布的 Rational 向 Hyades 的逐步移入也宣布了 CEQ（Continuously Ensured Quality initiative），它将关注于应用生命周期管理 ALM（application lifecycle management）中的角色、过程、工件和工具集。

通过增强 IBM Rational 及第三方产品间功能的交互性以及工具的数据交换，通过公用的 Hyades 框架，IBM 相信这将增强用户开发的软件产品的质量。

IBM 宣称，构造不佳的软件每年总共消耗软件组织超过 22 亿美元的资金。问题出现在 ALM 周期的早期－特别是在需求获取阶段－这意味着很多项目在编码还没有开始前就注定失败了。

Eclipse 和 Hyades 服务于 IBM 的整体目标，简化 ISV 将他们自己的工具插入到 IBM 和 Rational 的工具中的工作量。理论上，一个简单、开放的框架将减少 ISV 的集成工作，而将潜在地吸引更多合作者。

Eclipse 和 Hyades 同样也为其它的工具集提供一个单一的接口，因此用户无需在不同的环境间切换来去，而避免开发的复杂化。

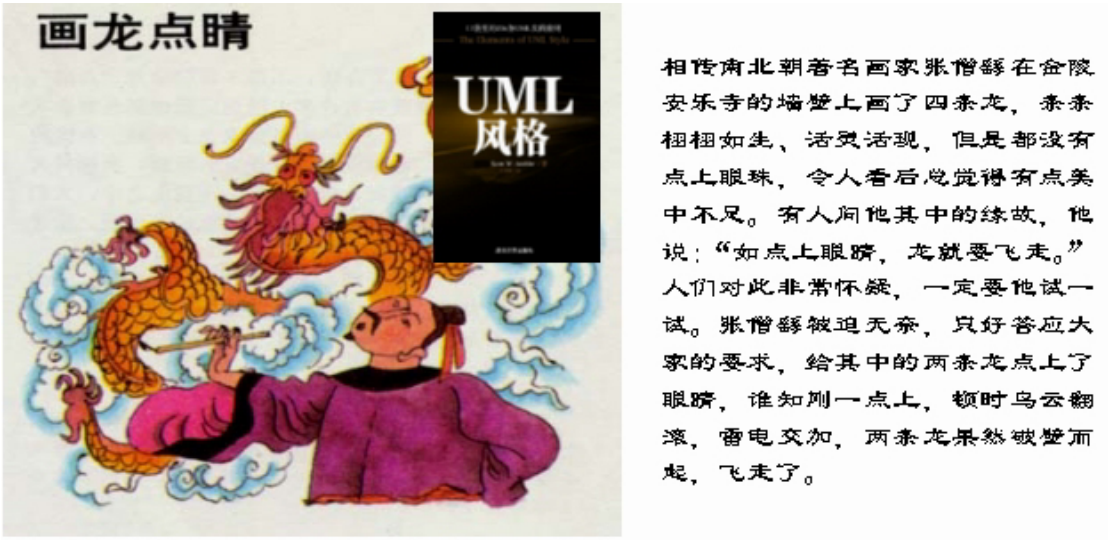
IBM 和 Rational 在 2002 年和 Parasoft、Scapa Technologies 以及 Telelogic 一起创建了 Hyades 项目，从不同的 ISV 那里集成了各种 ASQ 工具。去年 Hyades 开始对 OMG 的 UML2 Test Profile 进行实现工作，这项工作为理解、捕捉、测试和监控数据提供一个结构化的数据模型。

“Hyades 已经创建了一个标准的通用架构，我们可以基于它创建跟踪从一个工具到另一个工具的（开发）资产的环境”，高级产品主管 Serge Lucio 说，“我们需要标准化一个基础平台……以提出解决方案，并扩展我们的产品”。

（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



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。

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

(英) Martin Fowler 著

王怀民 周斌 译

UMLChina 审校

中译本已出版>>

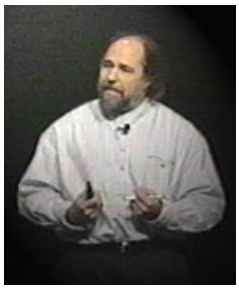
Grady Booch 展望 UML 之路

袁峰 摘译

讨论

Grady Booch，IBM Rational 分部首席科学家，UML 的关键发明者之一。Booch 近日和 Darryl K. Taft 讨论微软的建模策略对 UML 可能产生的影响。

Booch 特别提到了微软主席和首席架构师 Bill Gates 在最近 eWEEK 的采访中的评论。



你对微软的建模技术有什么想法吗？

我一直在跟踪 Bill 的言论，包括一年左右之前微软财政会议上他关于建模的第一次声明。

因此，我实际上是很高兴看到他越来越更多地参与进来－因为，和他一样，我们都完全同意这一点，建模将越来越重要。

现在，我们面对的系统正遇到日益壮大、异构、多平台、多语言、分布、安全、并行以及可信赖的复杂等问题，我们的观点是，建模正是解决之道。

那么，是什么时候我们开始和 Bill 开始共同关注这些的呢，应该是两年前，从在 San Francisco 发布 Visual Studio .Net 算起，有时候建模成了我们相互的“雷达屏幕”。有人曾和我说，“天啊，有人在搞分裂（因为微软采取的是和 UML 所不同的方案）。”

我的观点是，很高兴看到微软认识到建模的重要性。因为我们已经为之努力多年，而且我们也很高兴地看到几乎所有的主流平台厂商都在真正进入这一领域。

看看 Sun 公司的 James Gosling 以及他的 TopHat 项目，这显然是在同一方向上的努力。天啊，当你面对这些对手的时候，有一点是可以保证的：巨大的变化是不可避免的（it certainly indicates there's a sea change）。

现在，我们有着不同的见解。当然，我们是坚定地基于 UML 符号以及其公开标准的。但看起来微软对他们在这一点的立场有些像是在胡扯，尽管我认为现在他们确实是清清楚楚在做一些和 UML 所不同的东西，尽管你必须要了解他们在做什么。

如果你回溯我们和微软的关系，你会发现，实际上他是我们在 UML 联盟中最早的拍档。当 Jim Rumbaugh、Ivar Jacobson 和我（UML 三友）在创建 UML 的时候，我们要让我们的工作稳定可靠，我们意识到我们需要更多的听众，因此，微软是我们联系的第一个伙伴。

事实上它们也对 UML 本身做出了实质的贡献。UML 的接口标识、语义以及符号都是直接基于微软的工作。

现在，不管为什么原因－你将不得不问微软，为什么他们这么做，或者为什么不那样做－直到我们将 UML 成文提交，微软都一直和我们在一起。但当我们把 UML 提交给 OMG 以寻求标准化的时，微软却选择了不参与。但是，从最长的时间来看，微软是参与过 UML 的努力的。

你从微软现在对 UML 的评价中看到了什么迹象？

嗯，Bill 对 UML 的看法有些是我认为不精确的。UML 的标识已经变成了一个复杂的元模型，而且 UML 也在变得越来越复杂……嗯， 如果你打开一个操作系统或者其他任何技术的内部，从外面看，它是非常丑陋的。

而且，坦率地说，UML 的元模型就是 UML 被打开的内部的部分。因此它……我还是不愿意说是复杂；我更愿意说它是语义丰富。但是这对于工具开发商来说是非常重要的，而且 UML 的实际的使用者是永远不会看到这些的。

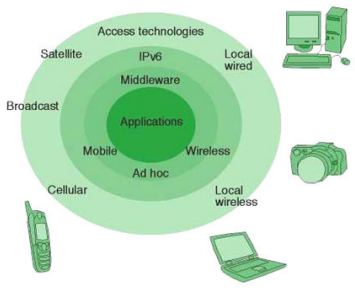
其意义就在于，借助如此丰富的元模型，像我们这种工具开发商可以作很多有趣的事情－这些可以重构的符号，在 Elipse 中比其他 IDE（集成开发环境）中更容易绘制，因为 Elipse 中提供的元模型更加丰富一些。

这使得转换的源的自动化成为可能，如果考察更高的抽象层，转换的形式可以是模式，并且用户可以将他们应用到系统中。

实际上，即使基于 Bill 的建议，我们也有相同的愿景－我认为有一个，尽管有着不同的实现－那就是对这个领域的各种内容的事实的影响。

我们已经看见了语言上的稳定。当然，我们看见了新的 C#，但是，它不能和 Java、Smalltalk 或者 C++相比，C#并不是语言上的真正变革。

因此，这也保证了一些稳定性。底层的平台发生了转移，较少基于裸操作系统而更多是基于中间件层。



Danny Sabbah，IBM 软部门的首席架构师，认为中间件是 Internet 时代的操作系统。因此，这是另一个转移。

因为，我们看到的就是两件事情的发生：语言的稳定和向中间件的转移－以及一个事实：人们在构建越来越复杂的东西。

我们看到了建模、模式和部分基于这些的方面（aspects）标识的影响。我们认为，建模就是一个长长过程的起点、帮助提高抽象层次。

但是，你没有发现一些问题吗？比如，共同的愿望，但是不同的实现？

嗯，我确实失望过，而且将继续失望，如果微软不选择 UML 的道路。UML 是全行业成千上万的人花费多年时间努力的结果，它实际上一个非常可行的标准。

因此，我对微软不选择参与进来表示失望，因为世界上又会出现一种新的编程语言，或者建模语言。

但是，这甚至超出了语言的纯粹语义。你必须认识到，围绕一个语言，有一个大的生态系统－事实就是，有很多相关的书籍和课程，UML 已经编织了一个在高校内部甚至涉及高中课程的网络。

因此，已经有这么一个生态系统支持 UML 成为一个标准，人们将开始使用它、赞成它，并实际了解如何应用它。

但是，如果你应用另外一种不是公开标准的语言。嗯，你必须拥有关于它的一整套教育。调查的结果应该是令人失望的。

你感觉围绕的举动是他们对 Rational 和 IBM 的反应吗？

嗯，我从来不想猜测微软的想法，但 Rational 显然已经置于 IBM 整个工具策略之中。对 IBM，它不仅仅意味着是一个 IDE，而是更加重要是，所有其他的产品生命周期管理工具将围绕它建立。

软件开发最终是一个团队运动。因此，我们将关注编译器的加速以及其他一些事情，但是最终，它是要把项目中涉及到的各种人都紧密结合起来。

尤其是在外包情况下，团队成员的日益分布在不同的地理位置。但在这里我们将不深入讨论这一点，这本身就是一个焦点问题。

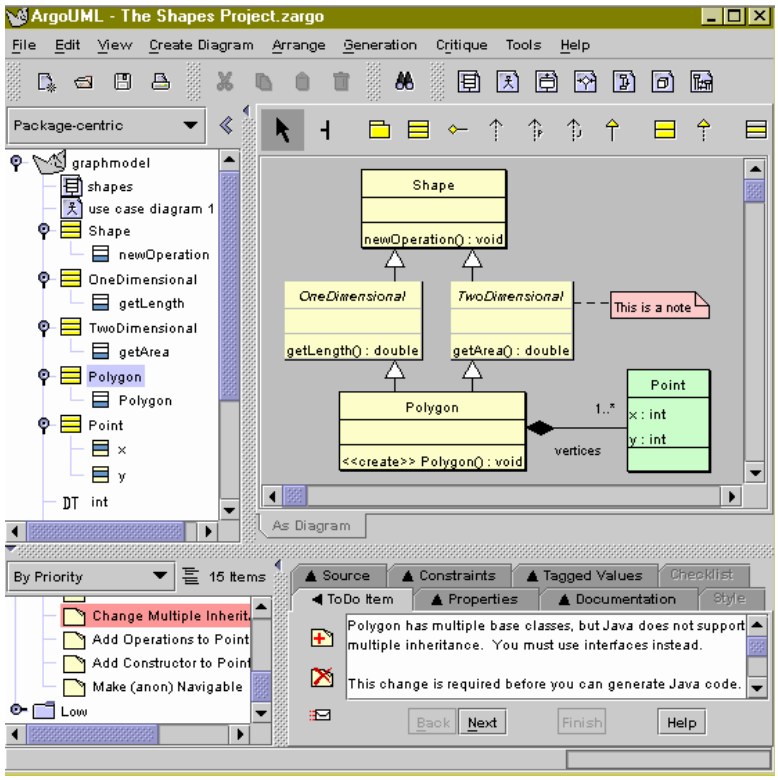
这也意味着，不仅需要具有单个的开发人员的经验，也需要有团队开发的经验。这也正是 Rational 追求已有一阵子的目标了。

而且事实上，作为微软的拍档，我们为它们的顾客提供了很多。但是现在，作为 IBM 的一部分，我仅仅是不得不在微软的策略中留下一个空缺。

在建模领域中，你看到了一些好的事情吗？

嗯，如果你看看涉及到 UML 的公司，它们为小公司们提供了一个整体的宿主（host），所有大的平台开发商都已经在某种程度上参与到 UML 中了。

Borland 公司显然是在这一行列之中的。另外，还有一些关于 UML 的开源项目－ArgoUML 可能是最早的一批之中的。而且你可能认为它们给我们带来了麻烦。



但是，作为同样信仰开源的一员，我知道这将激怒 Redmond 的一些人，我认为这是一个好事情，因此它代表了市场的部分，并且它促使我们增加价值，事实就和开源社区跟我所说的一样，它们的支持者已经遍布天下。

好，那么 UML 目前渗透有多么深入了呢？

你可以在很多地方看到 UML 的渗透，但这并不是一个全部是或者否的问题。最酷的是 UML 的良好设计保证了它可以满足很多不同的涉众的需要。

因此，尽管你阅读规约的话，发现它是如此的乏味，但在实践中的事实是－这是我们所期待将发生而且应该发生的－是，不同的涉众使用 UML 的不同方面。

比如，我最近和某些使用 UML 来描述系统部署的人打电话。因此，我看到了大型的企业系统，其中你要询问自己的一个问题是，所有这些部署的部分中，瓶颈在哪儿？它们的版本是什么？对这些问题，有不同的方法可以来进行可视化。

但有趣的是，你可以用 UML 来进行可视化。因为它意味着我可以使用同样的语义、同样的可视化方法同网络工程师沟通，他对编程一无所知，但这种同样的语言也可以被你的安全人员所理解，它也需要了解这些模型。

因此，我们看到的是－我们在模型驱动开发方面的特别努力在于 UML 确实在一些部分增加了价值，首先跃入脑海的三个就是部署、业务规则和模式。

但这每一个都不是完整的 UML，而只是 UML 的一部分。但事实在于他们都在使用各种 UML 的手段。作为一个开发经验丰富的人，你可能有清晰和唯一的设计思路，但是开发总是存在足够的“噪音”，因此所有可以降低摩擦的方法，以及同一的语言都会起到帮助作用。

因此，微软现在许诺对建模的支持底线是什么呢？

很高兴看到他们现在开始考虑这个。我们已经认为建模是主流，有一阵子了。但事实是微软现在也开始认识到这一点，这是非常非常酷的。

相传南北朝著名画家张僧繇在金陵安乐寺的墙壁上画了四条龙，条条栩栩如生，活灵活现，但是都没有点上眼珠，令人看后总觉得有点美中不足。有人问他其中的缘故，他说：“如点上眼睛，龙就要飞走。”人们对此非常怀疑，一定要他试一试。张僧繇被迫无奈，只好答应大家的要求，给其中的两条龙点上了眼睛，谁知刚一点上，顿时乌云翻滚，雷电交加，两条龙果然破壁而起，飞走了。

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

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

基于模型的通信系统设计

Sari Leppänen、Markku Turunen 著，黄蕾 译

讨论

本研究报告描述了用于协议工程的一种模型驱动的设计方法。该方法覆盖了从前标准化（*pre-standardization*）直到实现的整个阶段的所有阶段。建模是面向服务的，建立在结构（*compositionality*）、外部可见行为和逐层细化的基础上的。本文将使用 UML2.0 作为建模语言。在本方法中用到的许多协议相关的基本概念都已经在文献 7 中介绍了。

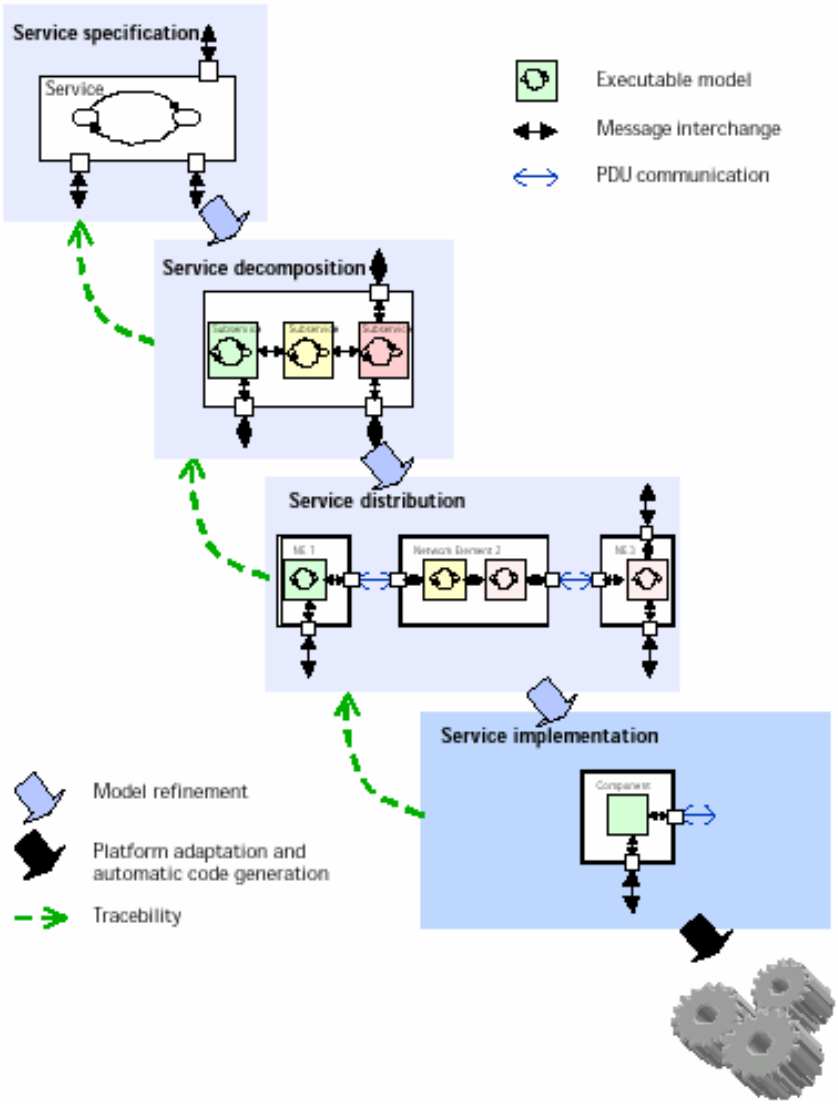


图 1 模型驱动的设计流程

设计过程包括 4 个阶段：服务说明、服务分解、服务分配、服务实现。图 1 大致描述了覆盖这些阶段的设计流。其核心理念是通过逐步细化可执行的服务需求模型推导出系统的最终实现。明确的工作流程和大量的仿真测试保证了模型之间的可追溯性。下文将通过举例对每个工作阶段进行详细论述。

服务说明阶段描述了系统提供的服务和功能需求集。功能需求被建模为有限状态机，可以进行仿真。非功能性需求被分别记录下来。在这个阶段，实现服务的系统被看作一个黑匣子。

服务分解阶段对服务一步一步的进行分解，自顶向下，把服务分为更小的部件，也就是各种服务组件。服务分解阶段提供了系统内部结构在不同的抽象等级上的多张透明视图。

服务分配阶段把服务组件分配到给定的网络结构上。模型的模块性允许系统存在多种分布和配置方式。可以通过组装联系来自底向上地封装各分布式功能，从而定义系统内部的模块化结构。

最后，在实现阶段，将服务组件与目标平台整合，并生成代码。

图2 大致描述了在系统实现时采用组装和分解方式逐步细化服务描述的过程。一个扩展的设计方法将覆盖从服务说明到完全实现的各个阶段。不同的阶段参与的人也不同。举例来说，在协议工程中，设计方法覆盖了从前标准化到协议实现的所有阶段。服务说明阶段与文献【4】中标准化过程的阶段1相对应。服务分解阶段主要是在标准化阶段2中，服务分配阶段覆盖了标准化阶段2和阶段3。

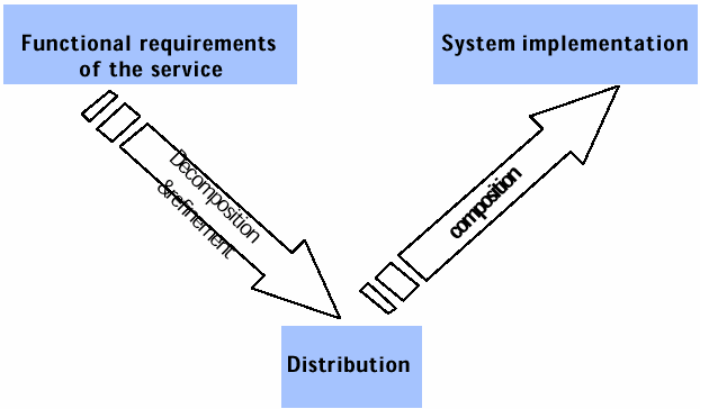


图 2 设计流中的组合分解和精化

下文将对该方法的各个阶段进行更详细的描述，通过类定义、接口定义、内部结构描述和外部行为描述这几个方面来描述每个阶段的内容。同时，还使用一个具体的实例作为例子。该例运用本方法的若干原则对3GPP无线通信协议建模。PCAP(位置计算应用部分)协议是无线接入网络中用户设备定位系统的一部分。PCAP定义了RNC（无线网络控制器）和SAS（辅助全球定位系统移动定位中心）网元之间的通信（见图三）。文献1中详细描述了RNC和SAS通信的功能需求。PCAP标准定义了lupc接口和相应的信令过程，即，通信协议的功能性描述。

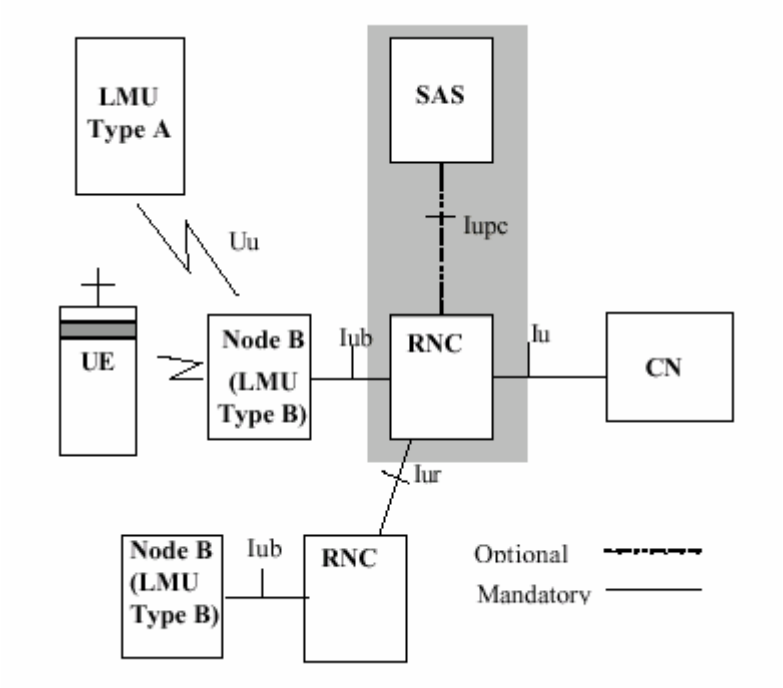
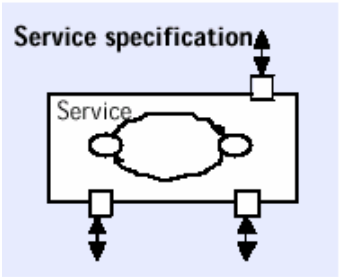


图3 UMTS网络体系中与定位计算相关的部分

1. 服务说明



服务说明阶段利用可执行的状态机来详细说明服务的功能性需求。这些功能需求是通过系统有效的外部可见行为来满足的。不同接口上的信号交互过程代表了系统的外部可见行为。通过定义这些信号交换操作的正确执行过程，可以获得对系统有效的外部可见行为的详细描述。这种描述可通过有限状态机的方式来表达。

要定义系统行为，必须定义一套外部接口，以及该接口上的信号集和信号参数。为系统及其外部环境建立域模型，以便识别系统的外部接口。

1.1 概念

概念定义阶段需要鉴别并命名整个设计过程中会用到的各种基本概念。本文描述的方法是一种可扩展的设计过程。该过程覆盖了从需求分析到完全实现的所有步骤，因此会涉及到各种具有不同背景和不同看法的人。这意味着概念的定义是一个非常重要的基础，所有的后续工作都建立在这个基础上。

需要定义的概念既有设计方法相关的也有应用领域相关的。需要确定各种逻辑的、结构的和物理组成单元的定义和命名，以提高整个设计过程中的相互理解。本模型的应用领域是协议工程。下面的一些概念被定义为UML2.0中的原型，在后面章节中使用，以便对各种类图的信息内容进行分类。

《网元》（«Network element»）

具体的电信实体，可通过特定界面进行管理。例子：3GPP系统中的无线网络控制器网元（RNC）

《用户》（«User»）

外部实体，不属于系统的一部分，但需要使用系统提供的服务。例子：使用3GPP用户设备的人

《服务》（«Service»）

服务提供商提供给用户的一系列可选组件，或者说提供给用户的一些功能。例子：定位服务，用于确定3GPP系统中用户设备的位置。

《服务组件》（«Service Component»）

在任何分解阶段都独立的一个服务。组件提供了一种抽象和信息隐藏机制，一个组件发生变化时可以不影响其他组件。模块之间的交互完全通过通信接口或共享变量来实现。例子： PCService组件。该组件是定位服务的一部分，它封装了与请求、报告、结束位置计算操作相关的各接口和功能。

1.2 服务说明阶段的类

本文中将那些能够接收异步消息的实体建模为主动类。其他概念建模为被动类。处于被建模的系统之外的实体为外部类。

在服务说明模型中，服务作为主动类，服务的用户作为外部类。此外，被使用的服务（用于支撑本服务的其他外部服务）也作为外部类。那些不需要实现的内部组件（如数据库、算法库）也作为外部类。

利用这些已定义好的类来画一个域模型，该模型描述了服务的已知信息和外部环境。域模型可以通过类图来表示，服务和相关外部实体都作为类或原型体现。他们之间的关系通过关联来描述。

图4显示了PCAP服务的域模型。

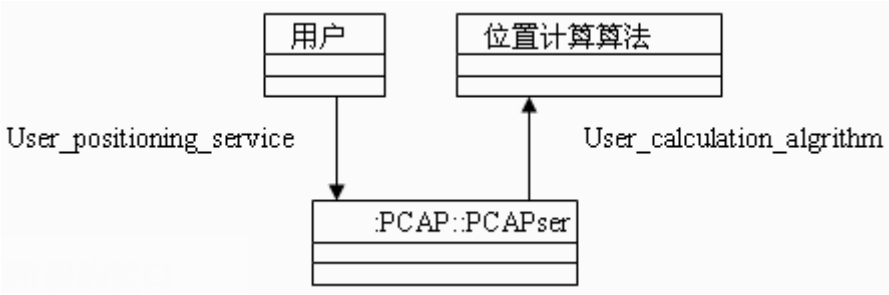


图4 PCAP服务的域模型

1.3 服务说明阶段的接口

根据UML的定义，接口是一种结构化的分类器（*classifier*），不能进行实例化。接口将一组信号封装起来，实现该接口的类可以通过这些信号进行通信。实现了接口的类既接收信号，也可以要求接口发送信号给其他实现了相应接口的主动类。

服务接口是系统及其环境之间双向传递信号的通信点。服务接口的概念包括两方面的含义：一方面是需要实现的接口，系统通过这个接口向用户接口（用户）提供服务；另一方面是已经实现的接口，系统通过这个接口使用外部实体（使用服务接口*(Used Service Interfaces)*）提供的服务。

提供与环境通信功能的系统会用到服务接口。服务接口是系统的外部接口。通过在外部接口上的信号交互，系统的部分行为变为可见的。这种行为称为系统的外部可见行为。

在服务说明阶段，首先确定服务接口的接口定义，如信号、信号的参数等。为了将服务接口和域模型中的类联系起来，需要明确与类中的接口定义相关的通信端口。这些端口称为主动类的交互点。他们确定了该类已实现的接口，和需要其他类实现的接口。

在 PCAP 例子中，PCAPservice 类的外部接口是根据域模型图中的信息确定的。用户接口的通信中用到的信号封装在接口 I_UserToPCAP 和 I_PCAPToUser.中。封装在 I_AlgorithmToPCAP 和 I_PCAPToAlgorithm 接口中的信号主要用于和外部实体的通信，这些外部实体实现定位计算的计算逻辑和算法。用户接口为 user_port，与定位计算实体之间的服务接口为 calculation_port。

1.4 服务说明阶段的体系结构

体系结构图定义了一个主动类的内部运行结构。体系结构图说明了怎样实例化 UML 对象，怎样相互作用共同形成一个系统或系统的一个部件。

服务说明阶段创建的服务类没有内部结构。为了在这一抽象级上定义服务需求，可以把提供服务的系统看作一个黑盒子，隐藏所有的实现细节，包括系统的内部结构。

域模型可以看作服务说明阶段的体系结构描述。它描述了服务环境，以及服务上下文结构。域模型可以通过类图或结构图来描述。PCAP 例子的域模型是通过类图来描述的，如图 4 所示。

1.5 服务说明阶段的行为

外部接口的通信描述了系统的外部可见行为。系统在用户接口上的外部可见行为满足服务的功能需求。通过有限状态机的方式来描述有效的外部可见行为，可以获得对于该服务的可执行的需求说明*(executable requirement specification)*。这份说明中还描述了系统与提供服务的实体和其他外部实体之间的通信。

上一工作阶段已经定义了服务接口的信号，现在定义信号的正确顺序，以便得到对各服务接口的有效行为的说明。可以使用顺序图来描述不同用例下的信号交互过程，不仅描述正常情况，也要描述出错情况。

图 5 中，左边的模型视图显示了正常情况、PC 成功（*PC success*）和两种错误情况下的顺序图。PC PCAP 错误（*PC PCAP error*）是由于协议错误引起的一种错误情况。PC 计算错误（*PC calculation error*）描述了在系统之外的位置计算功能出错的情况。

在顺序图的基础上，建立一个或几个状态机来描述服务接口上的行为。可以把某一用例的某一接口上的需求的一部分用一个状态机来描述。举例来说，图 5 中，两个服务接口上 PC 成功用例的功能可以分别建立两个状态机。图 6 是用户接口的服务说明状态机的一个部分，在用户端口（*user_port*）实现的。完整的描述服务需求的状态机可通过这些小的部分组成。

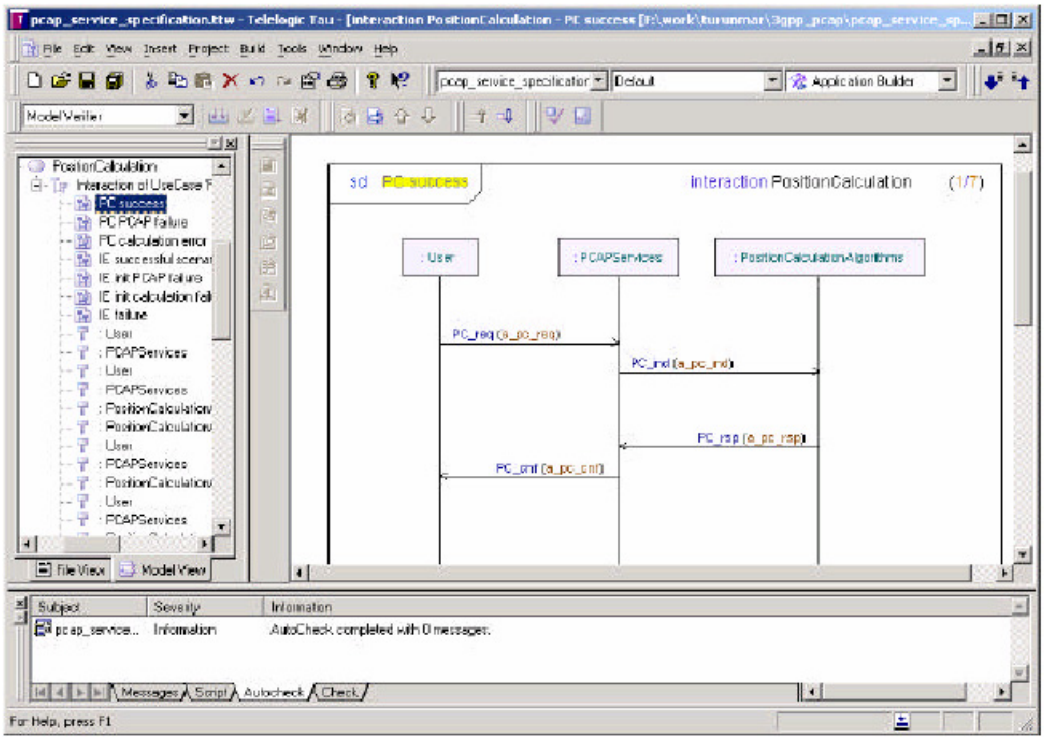


图 5 成功定位计算的场景

由于高抽象级的模型隐藏了所有的实现细节，所以不确定性（*nondeterminism*）是不可避免的。不确定性可通过交互式仿真来解决。在模型自动执行的情况下，不确定性可通过常量或由随机生成器生成的变量来解决。图 6 给出了在状态机中使用不确定性的例子。

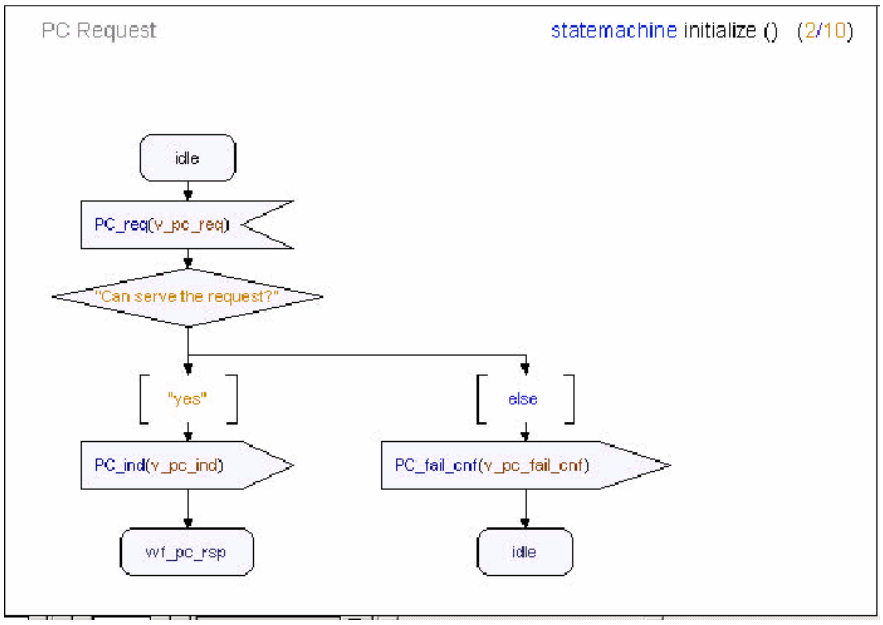


图 6 服务说明中的非确定行为

1.6 服务说明阶段的可执行性说明（Executable specification）

本设计方法的一个非常显著的优点在于模型的可执行性。我们的模型具有在不同抽象级别上执行的能力，这使得新观点、新建议的测试在标准化过程中进行。可执行性也使我们能够对核心功能进行验证和评估。

出于仿真和测试的目的，需要确定系统的正确配置。从服务类的端口中选择那些我们想观察的，把它们加入到配置中去。仿真可以自动运行，也可以交互式运行。对于自动仿真，如为验证而进行的仿真，可以定义独立的用户过程和监控过程，并把两者连接到系统的实现端口。输入输出测试的自动处理使得系统可以进行更详尽的测试，以提高系统可靠性的可信度。

图 7 描述了 PCAP 服务在两个服务接口上的交互式仿真的配置。

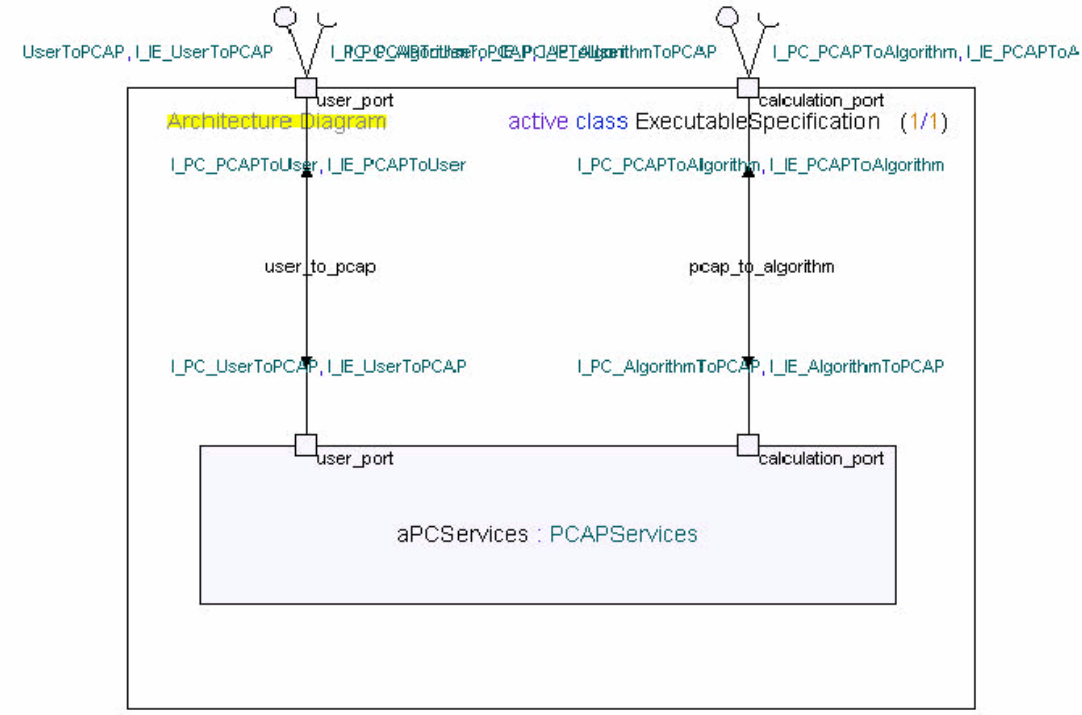


图 7 可执行的服务说明的配置

2. 服务分解

在服务分解阶段，实现服务的各项逻辑功能被逐步分解为更小的服务组件（*service components.*）。分解工作自顶向下进行。每个服务组件都被标记出来，与合理分布（如根据与其他服务的关系）相关的信息也被记录下来。

服务分解模型描述了不同抽象级别上系统内部结构的若干视图。最终目标是生成一系列更小的模型——服务组件，每个服务组件都封装了一个逻辑上一致的功能单元。服务分解使得在不同的网络体系结构中可以运用不同的分布式模型。此外，它还使系统可以灵活定义为不同的配置，并给系统提供了良好的模块化过程和高效的资源分配规划。

2.1 服务分解阶段的类

服务分解阶段首先要为分解出来的服务组件定义类。然后，根据服务组件的接口和定义外部可见行为的状态机，把端口（port）与这些类相连接。

分解是一个递归的过程，始于服务说明模型中定义的服务类。分解过程最终生成一颗分解树，每个节点对应于一个服务组件，一个逻辑上一致的功能单元。服务组件用类来描述，它封装了服务组件的外部接口定义和外部可见行为。分解树的根是服务说明模型中的服务类。如果说深度 n 的节点是某一个服务模块的黑匣子，那么该模块的内部视图可由深度 n+1 的一系列节点组成。分解树的树叶是最小的功能元素（如过程），可以把他们封装成为单独的单元。

PCAPservice 类是分解树的根。服务说明阶段定义了 PCAPservice 类的外部接口和行为。本阶段将 PCAPservice 类划分为两个功能实体，并创建相应的类。PCservice 类包含了与位置计算相关的功能。IEservice 类包含了与信息传送相关的功能，这些信息是位置计算功能所需要的。

2.2 服务分解阶段的接口

服务分解产生并定义了服务组件的外部接口。通常，这并不意味着要引入新的信号，而主要是已有信号的重新组合。

分解过程显示了一个服务组件的内部结构，把外部接口分为几个部分，把不可见的内部接口变为其成员组件的外部可见接口。在分解树上，部分深度 n 上的内部不可见接口成为深度 n＋1 上的外部可见接口。

服务分解阶段定义系统中所有提供服务的内部接口。与服务组件类中的端口实例不同，内部接口是不可见的。在描述服务组件内部结构的体系结构图中，内部接口通过端口之间的连接变得可见。服务组件的外部接口表现为体系结构图外边界上的接口实例，它们和内部接口相连。

当服务组件被分解为更小的服务组件，而且对每个组件都定义一个类时，外部接口也被分解了。在这些更小服务组件的端口定义中要明确实现原有组件的外部接口中的哪一部分。然后，被分解的服务组件的内部结构用体系结构图表现出来。如果内部组件之间需要交互的话，这个阶段就需要定义相应的内部接口。对于“纯”模块化的一个基本要求是组件之间交互的完全封装，只允许通过信号传递来实现组件间的交互。

PCAP例子中，PCService组件和IEService组件之间不存在相互通信，因此分解过程不会产生对新接口的需求。服务说明阶段定义的外部接口根据功能进行分解（见图9）。图8描述了用户接口的PCService部分，PCAPService的user_port 和 calculation_port也被相应地分解为PC相关的部分和IE相关的部分，并分别添加到PCService类和IEService类。

```
interface I_PC_UserToPCAP {
  public signal PC_req       (PC_req_param);
}

interface I_PC_PCAPToUser : I_PC_PCAPToUserFail {
  public signal PC_cnf       (PC_cnf_param);
}
```

图8 PCService组件的用户接口定义

2.3 服务分解阶段的体系结构

体系结构图定义了被分解的服务组件的内部运行结构，这是从其他组件的角度出发所看到的结构。服务分解阶段通常会产生几张体系结构图，每个分解层次至少有一张。

体系结构图可以由多个部件组成，每个部件是针对对组成原始服务组件的各组件而定义的类。每个部件都与端口相连接，在该端口上请求或实现用于外部（部分到环境）或内部通信（部分到部分）的信号。

通过与各部件相连的端口，或部件之间的连接器，接口变得可见。连接器是一种承载内部或外部通信的介质。连接器可以使通信链路变得更直观，但也可以被忽略。在某端口上建立通信的必要前提是：体系结构图中该端口上请求——实现定义是一致的、完备的。连接器可能是单向或双向的，并对每个方向上的信号作出具体规定。如果信号数量很大的话，可以为连接器的每个方向定义一个信号列表或者参见某个接口。

已分解的服务组件的外部接口在体系结构图上表现为外边界的端口。每个外部端口都和一个或几个内部端口相连接。当一个外部接口对应几个内部端口时，内部端口的复合信号集应与外部端口的信号集一致。与内部通信相关的端口也相应连接在一起。

图 9 显示了 PCAPService 的内部体系结构，由 PCService 和 IEService.两个类组成。

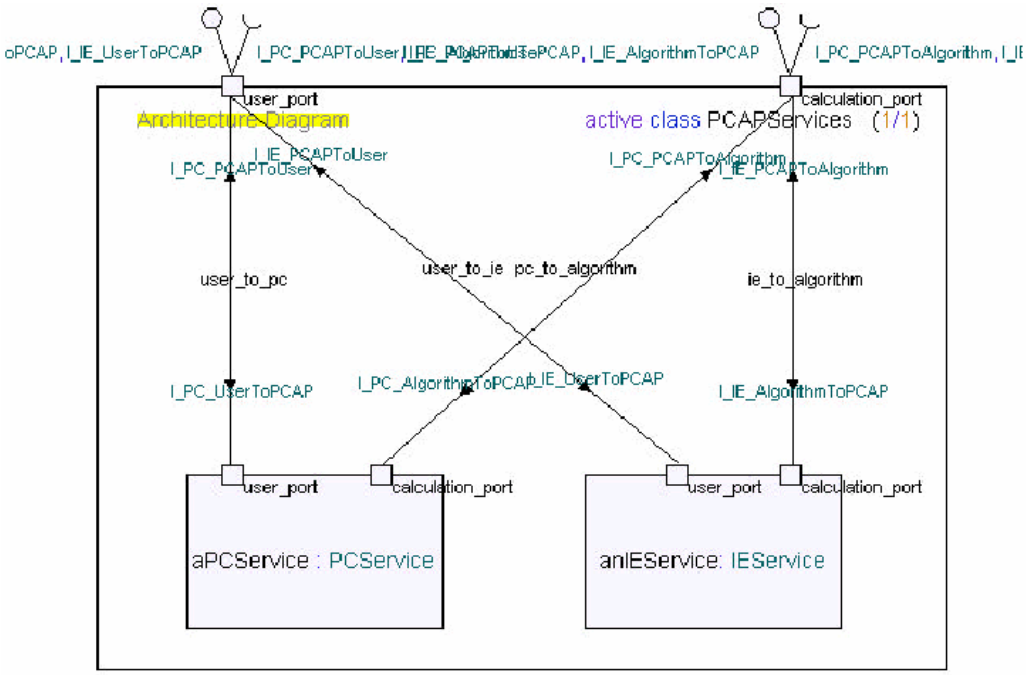


图 9 PCAPService 的内部体系结构

2.4 服务分解阶段的行为

服务分解过程把服务组件的一部分内部接口转化为它所包含的服务组件的外部接口。因此，被分解的服务组件的一部分内部行为被转化为它所包含的服务组件的外部可见行为。

本设计方法的一大原则是，建模阶段只确定外部可见行为。这种确定是逐步求精的，也就是说，当分解过程逐步深入，会发现更多的内部通信接口，出现更多新的外部可见行为。对行为的说明主要是定义信号交互的过程，这个过程通过状态机来描述。

服务分解阶段需要定义所有服务组件的行为。每一步分解至少会产生 2 个新的服务组件。首先服务组件的行为被分解，也就是说，根据外部信号和内部结构的调整对早期阶段定义的功能进行分解。如果分解出的各组件之间需要相互通信，并定义了新的接口，那么该接口上的行为也必须定义好，并和先前定义的行为是一致的。与服务说明阶段描述行为的方式相同，这里也通过顺序图来描述服务组件的不同接口上的外部可见行为。可以采用独立状态机（例如：每个接口建立一个状态机）的方式来描述行为，也可以建立一个扩展的状态机对服务组件的所有接口的外部可见行为进行描述。

图 10 描述了 IE 的一个成功执行过程。重复过程出现在最后一个状态机中，通过自循环的方式表现，在图中被高亮度标出。

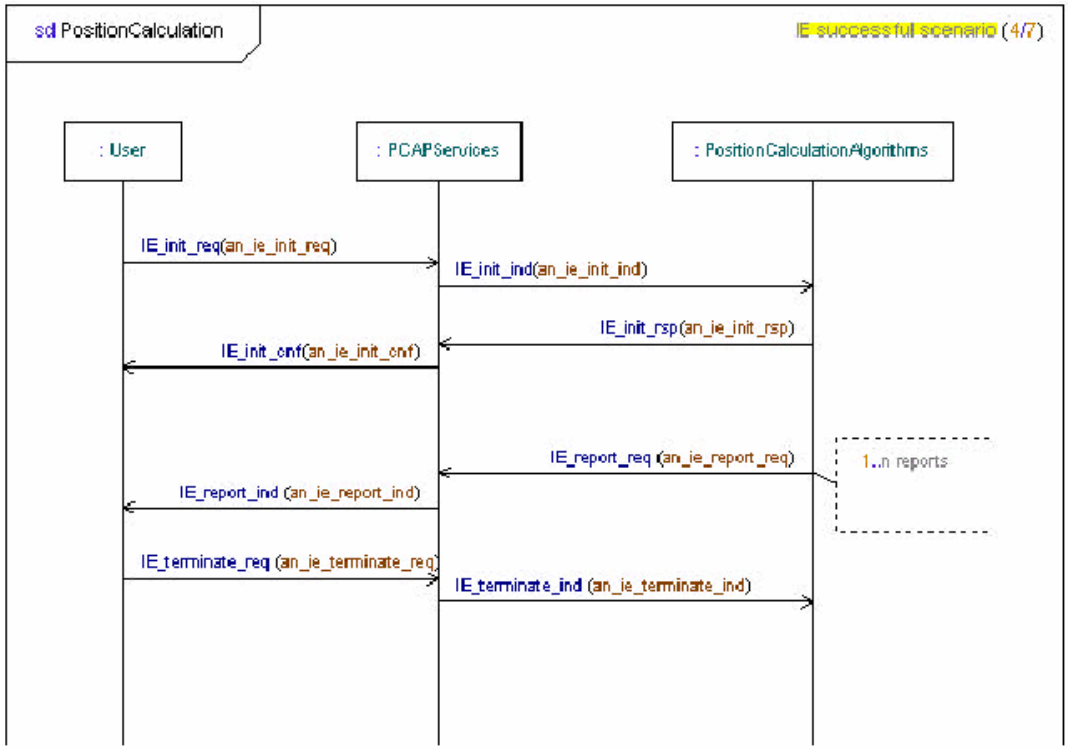
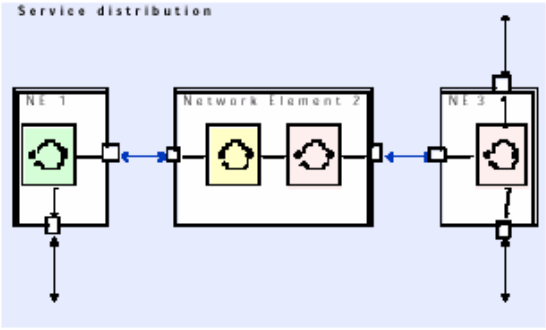


图 10 成功信息交互过程

2.5 服务分解阶段的可执行性说明

出于测试的目的，为不同的服务组件设计不同的仿真配置，即测试台。在确定作为独立模块的各服务组件的功能正确性之后，设计新的仿真配置，对几个服务组件一起进行测试，测试它们的交互和并行功能是否正确。

3. 服务分配



服务分配阶段，系统的功能被分配到特定的物理网络上。这种分配是建立在网络体系结构以及服务分解阶段产生的服务组件的基础上的。根据网络体系结构的不同，服务组件被分配到不同的网络元素和协议层次上。服务分解，以及由此而来的模块化，支持不同网络结构下的不同分配模型，支持系统的灵活配置。

根据服务组件的位置和功能确定相应的结构性元素（*Structural elements*），如协议、计算环境等。换句话说，服务组件自顶向下的构造方式产生了这些结构元素。

本阶段定义位于不同网元但相互通信的结构元素之间的 PDU 接口。新的 PDU 接口会产生新的外部可见行为，需要和结构元素的原有行为进行整合。

3.1 服务分配阶段的类

服务分解阶段，服务组件按照自顶向下的方式逐层分解为更小的服务组件。其目的是定义小的但逻辑上一致的功能模块，这些模块应具有定义明确的接口。

服务分配把前阶段分解得到的服务组件分配到几个物理网元上。分配过程必须同时考虑网络的体系结构和服务分解模型。下面列出了三个重要的情况：

首先，也是最简单的情况，不同的服务组件分配在不同的物理网元上。如果各分布式服务组件间不需要进行通信的话，服务分配就完成了。在服务分配模型中为每个分布式服务组件定义一个类。接口定义无需变化。

第二情况是不同的服务组件之间需要进行通信，而且信号交互过程被内部接口封装。在服务分配过程中，用于内部通信的接口被“公开”，作为 PDU 接口。举例来说，在图 11 中，组件 SC1 和 SC2 之间存在交互，分配完成后，他们之间的内部接口被公开为 PDU 接口。这种情况下，服务分配模型为每个分布式服务组件定义一个类，接口定义的变化在稍后的接口定义阶段描述。

第三种，也是最复杂的情况，一个服务组件被分配在几个物理网元上。服务组件的外部接口即服务接口根据功能分配情况进行分配。为不同网元上的对等实体（分布式服务组件的一个部分）分别定义类。在对等实体间定义新的 PDU 接口，使统一服务组件的分布在不同位置的各部分间可以进行交互和端到端的通信。举例来说，服务组件 SC3 的功能分配到网元 NE2 和 NE3 上。为每个实体定义类 SC3a 和 SC3b。对等实体之间定义 PDU 接口以便进行交互，并在系统层面保证分布式服务组件的功能的完整性。

服务分配阶段使设计流程中最具挑战性的阶段，必须非常小心谨慎。服务分配与案例情况密切相关。除了这里列出的三种主要情况外，肯定还存在其他的情况。例如，一个服务组件可能会复制到几个网元上。

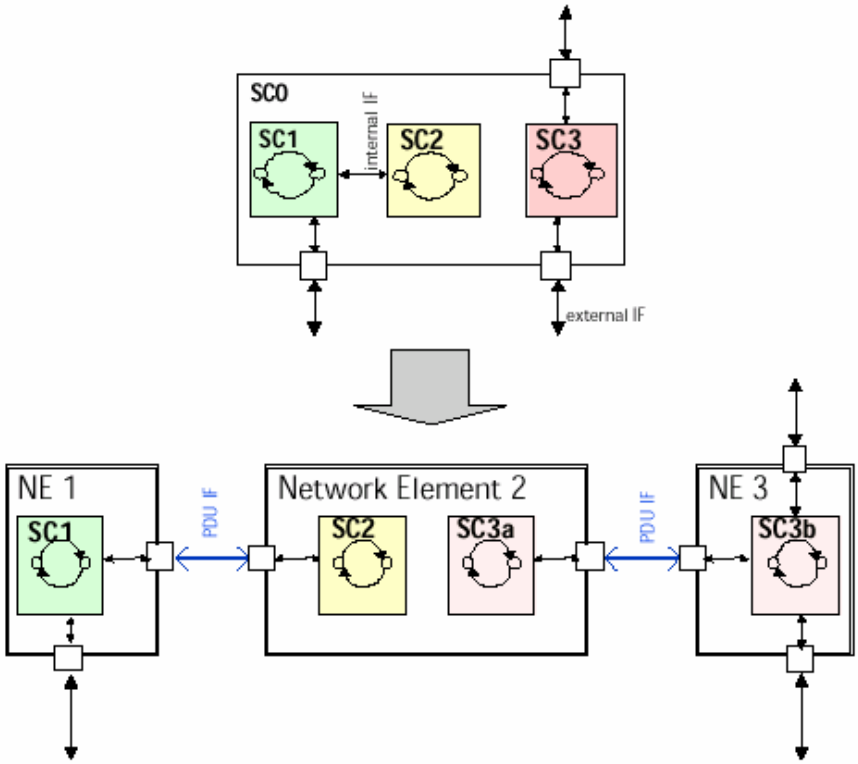


图 11 服务分配

以自顶向下的方式对分布式服务组件进行分类或组装会生成结构性元素，这些元素在分配模型中被定义为类。组装作为分解的反向过程，在各不同的抽象级别上生成结构性组件。举例来说，如果一个网元上的两个服务组件之间联系紧密，且可交错执行，则可以把两者合并到一个计算环境中（*computational context*）。更进一步，可以把参与两个网元之间交互的所有计算环境关系封装为一个子协议（*subprotocol*.）。一个协议（*protocol*）由几个子协议组成，一个层（*layer*）包括几个协议。最终成果是对于分布式服务的内部体系的一个模块化的，定义明确的描述。

PCService 和 IEService 组件分布在图 3 描述的网络体系上。这些服务组件相互独立允许，因此它们可以有自己的计算环境。通常在一个网元内部定义计算环境，因此系统中针对特定网元的计算环境称为 SAS_PCService，RNC_PCService，SAS_IEService 和 RNC_IEService。PCAP 协议的子协议，SAS_PCAP 和 RNC_PCAP，封装了一个网元中的所有计算环境。

3.2 服务分配阶段的接口

服务分配阶段，服务组件所封装的功能被分配到网元上。这种分配会导致接口定义发生一些变化，主要涉及到那些用于服务组件之间通信的接口。

在比较简单的分配情况下，不同的服务组件分布在不同的网元上。如果分布式服务组件之间不需要通信，那么服务分配不会改变接口定义。在组件之间需要进行通信的情况，而且信号交互被内部接口封装的情况下，内部接口作为 PDU 接口被“公开”，或者说，在系统一级变为可见。PDU 接口是用于不同的网元上的不同服务组件之间进行通信的接口。

第三种情况下，一个服务组件分布在几个网元上。服务组件的外部服务接口根据功能的分布而分布。功能的分布产生了位于不同网元上的对等实体。要通过通信方式进行必要的通信，有必要在两个对等实体之间定义一个新的 PDU 接口。PDU 接口的信号定义来自行为描述。根据各结构化元素的组合方式把 PDU 接口组装在一起，最终针对某一协议只存在一个 PDU 接口。

图12显示了PCService组件RNC网元侧的接口。PCService的用户接口部分在I_PC_UserToPCAP和I_PC_PCAPToUser中定义。I_PC_SASToRNC和I_PC_RNCToSAS定义了PCService的PDU通信。当I_PC_SASToRNC和I_PC_RNCToSAS与PCService相应的PDU接口I_IE_SASToRNC和I_IE_RNCToSAS整合时，得到I_SASToRNC 和I_RNCToSAS接口，该接口定义了PCAP协议的PDU通信。

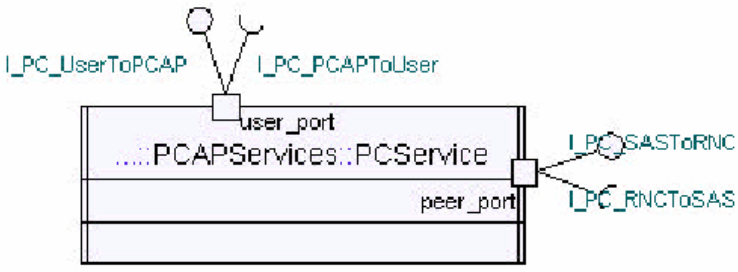


图12 分布式PCService的接口

3.3 服务分配阶段的体系结构

服务分解模型描述了不同抽象层次上的体系结构中服务组件的内部结构。服务分配阶段，体系结构图用来在不同抽象层次上描述结构元素的内部结构，如协议、子协议和层。体系结构图中，定义了更小的结构元素的类被作为局部。把端口与该局部的各个接口相连接。可以使用连接器来使点对点的通信变得可视化。和前几个阶段一样，封装某结构性元素外部接口的端口必须和组成该元素的各结构元素的内部接口相连。

图13描述了RNC侧PCAP协议的内部结构。PCAP协议包括两个计算环境：PCService和 IEService。在PCAP例子中，计算环境也可被称为子协议，因为它不会再被拆分。PCAP的用户端口，即，PCAPServices_RNC类，与PCService和IEService的用户端口相连，该端口封装了特定计算环境的接口。类似的，PCAP协议中封装了PDU接口的peer_port，也与内部相应计算环境的端口所连接。

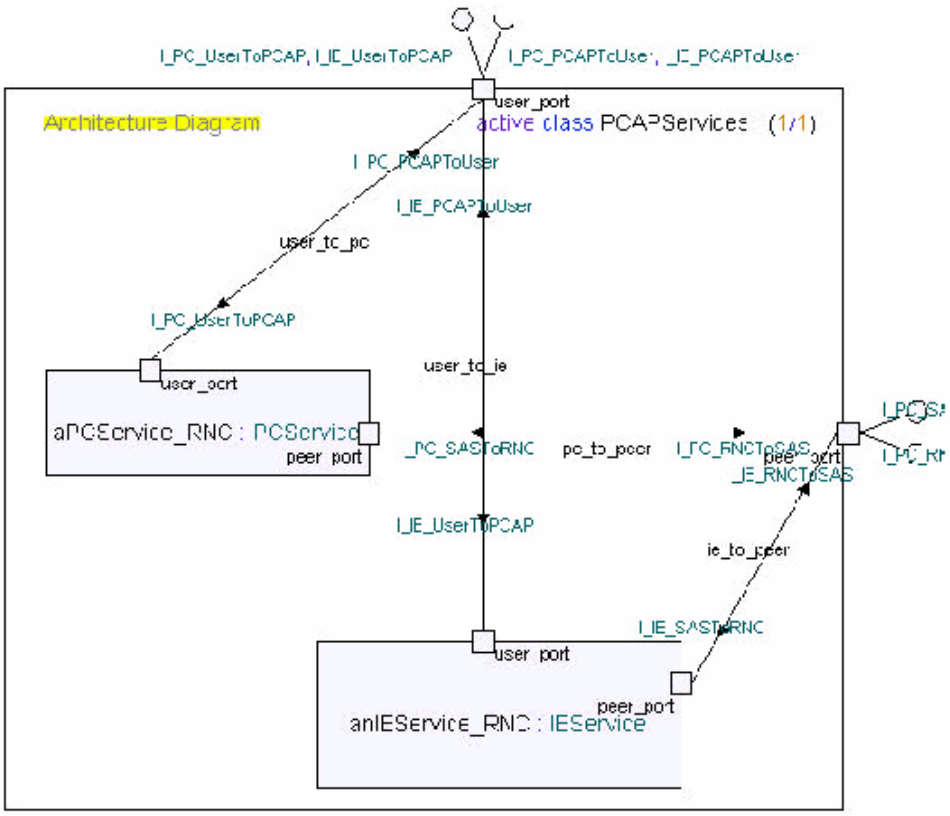


图13 PCAP协议，RNC侧的内部结构

3.4 服务分配阶段的行为

服务分配阶段将服务分解阶段定义的一些行为分解为几个存在交互的部件。在前面的类定义阶段，针对每个计算环境定义一个类。现在，在行为定义阶段，我们把每个计算环境的类用一个状态机来表示其功能。因为对应于不同网元的计算环境通过PDU通信实现相互之间的交互，所以这里定义的状态机是PDU通信的两端的状态机。这意味着一个状态机的发送行为必须对应于对端状态机中的接收行为。

一个计算环境要处理两种不同类型的通信，即，“水平的”PDU通信和“垂直的”协议栈内与用户的通信。水平的PDU通信定义了协议标准以保证不同厂家的网元之间的互操作性，使它们属于同一定义域。垂直的栈内通信是与实现方式相关的。垂直通信的状态机处于协议实体之外。这种错层的通信通过分层状态机来表示。

状态图形象化地表示了状态机。Tau/Developer支持两种画状态图的方式。面向状态的视图给出复杂状态机的一个很好的概观，但是当每个转换包括更多的行为或每个行为必须描述得更清楚时，这种方式不够实用。基于这个原因，也可以用面向转换的方式来描述状态机，可以用明确的符号描述出每个转换中实现的不同操作。在分层状态机中，这两种方式综合使用。与分层状态机的主线——水平PDU通信相关的行为通过面向状态的方式来描述。主线上的各个状态都是复合状态，即，由其他状态或转换复合而成的状态。按照计算环境中的过程来命名各个状态。每个服务状态背后是用于描述与用户之间的垂直通信的另一个状态机。该状态机按照面向转换的方式来描述，因为除了栈内通信外，这个状态机还包括了所有的内部计算。

在水平状态机中，所有状态之间的转换或者是PDU的接收或者是PDU的发送。在垂直状态机中，所有的转换与服务原语（*service primitives*.）相对应。水平和垂直状态机之间的控制交换通过命名出入连接点（*named entry and exit connection points*.）来处理。连接进入点是进入一个复合状态的起始点；连接退出点是离开符合状态的终止点。在水平状态机中接收一个PDU，使状态机通过连接进入点转换到垂直状态机中。在水平状态机中发送一个PDU的动作是垂直状态机从当前复合状态中退出时激发的。在水平状态机中明确复合状态的激发事件，这个事件将导致状态机从复合状态（及其子状态）中退出，进入水平状态机中的另一个新状态。这种特性为特殊信号（如异常或终止）的接收和处理定义了一个灵活简单的方法，而无论当前是什么状态或处于什么样的状态集中。

PCAP例子中，水平状态机的主线描述了IEService计算环境的行为（见图14）。在这个主线状态机中，转换集包括了IEService的PDU接口上PDU的发送和接收。图15描述了“空闲”复合状态后的垂直状态机的状态图。

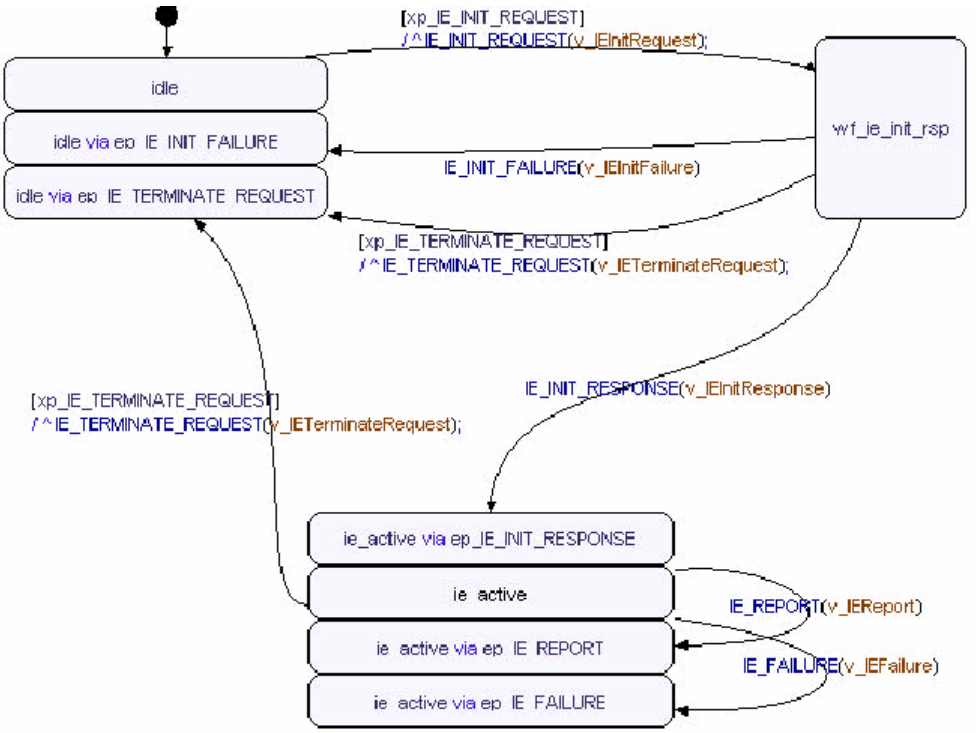


图14 IEService服务的PDU状态机

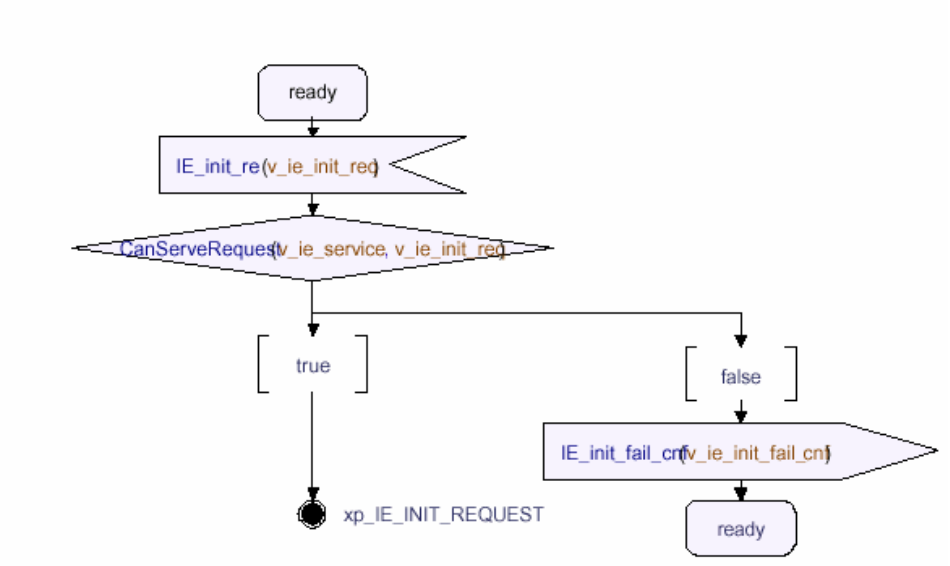


图15 空闲状态的内部情况

3.4 服务分配阶段的可执行性说明

通过不同的配置，可以分别对整个协议实现或仅仅对协议对等体进行仿真和测试。此外，也可以仿真和测试选定的接口，观察其行为。

图16显示是仿真PCAP协议的配置，是实现【2】中定义的Iupc接口时所使用。在这个仿真中，可以只观察用户接口，而隐藏其对等实体间的PDU通信接口。

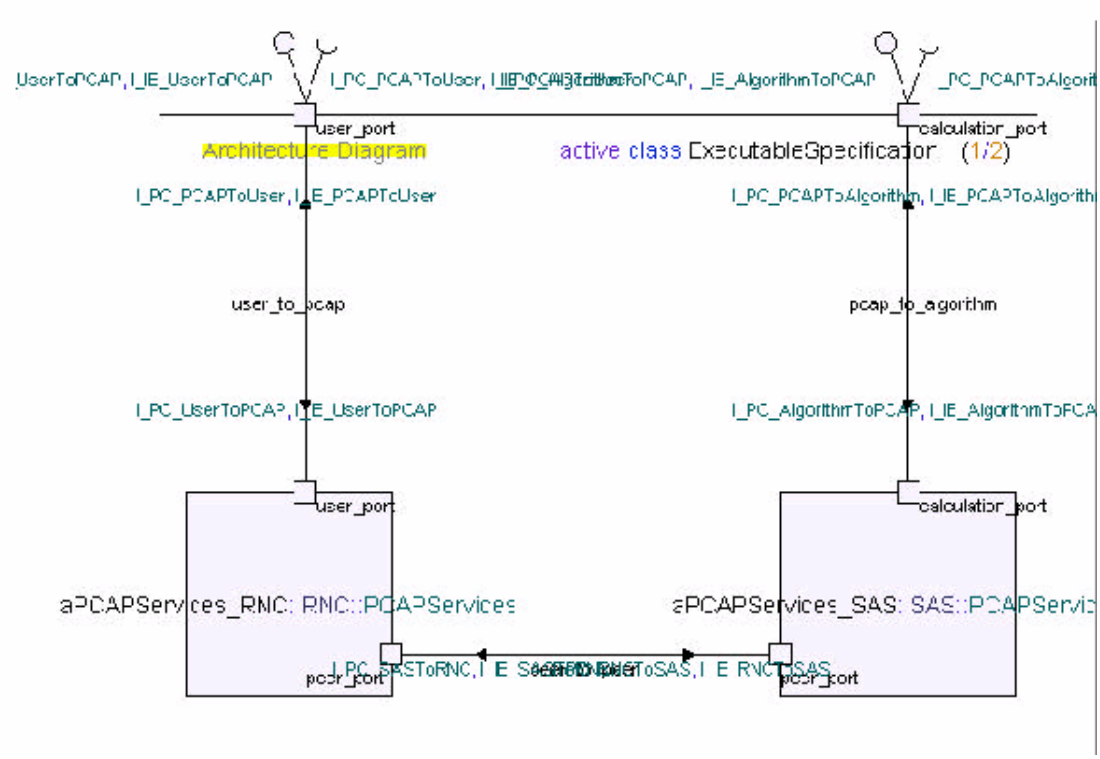
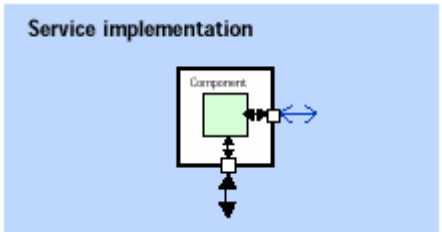


图16 可执行的服务分配的配置

4. 实现



实现阶段的目标是建立实现模型，以便自动生成目标平台上的代码。本阶段可划分为两个子阶段：通用子阶段和平台相关子阶段。在通用子阶段，给模型添加前面遗漏的功能特性。需要添加的特性包括：同一计算环境支持几个并行的通信会话、分布在不同网元上的对等实体间的虚拟PDU通信、错误处理。目标平台相关子阶段需要考虑目标平台及其配置相关的事项。以下各节描述了通用子阶段的操作。

4.1 实现阶段的类

服务分配阶段定义的主动计算环境类（active computational context classes）是本节计算环境实现类的基础。这些类封装了与协议的真正功能相关的计算和算法，即，生成协议提供的服务。

为了使几个通信会话同时并存，且每个对话具有自己的计算环境实例，可以使用主从模式（*Master-Slave*）。一个计算环境实现类作为工人（*worker*）。工人类实例的产生和终止是相互独立的。只有在通信会话期间，工人类才是活动的。Master类生成新的工人实例并维护活动工人的记录。

在最初的服务说明阶段，消息接收者的标识并不重要。消息发送时，它到另一个实体的路径在模型的消息路由中确定。现在，同一个类会同时存在几个活动实例，必须要有路由的功能。该功能决定了到来的消息的适当接收者。这里可以使用消息经纪人模式（*Message-Broker*）。本例中该模式使用了两次。第一次，用于决定接收的计算环境的类型，是PC还是IE。第二次，用于确定同一类型内的不同工人。

PDU不能直接从一个协议实体送到另一个实体，因为两个实体之间没有直接的物理通信路径。不同网元的计算环境需要使用另外一项服务，该服务为两个网元间提供透明数据传输服务。必须要在现存的低层服务的基础上实现虚拟PDU通信。同时，应把PDU通信相关的信息存储在工人类中。解决方法是使用编解码（*Codec*）模式（或者称为对等代理）。每个对端代理（peer-proxy）作为对等实体的代理。它拦截工人类发送的外出PDU，并进行编码生成比特串。编完码的比特串通过低层通信服务发送出去。对于进来的PDU，对端代理接收低层的数据，并解码获得内含的PDU分组。最后，将已解码的PDU分组送给正确的工人。

本例中，计算环境相关的master、消息经纪人和对端代理的功能合并形成两个类，即，PC_RED和IE_RED (路由——编码——解码)。此外，PCAP协议中存在两个子协议，PC和IE，子协议的实现应尽可能保持独立。将环境类型相关的路由（即，不管消息是PC消息或IE消息）从PC_RED和IE_RED独立出来，归入MUX类中，这是因为消息的路由可仅仅基于消息本身提供的信息进行。之所以添加MUX类是因为对于PCAP只有一个服务接入点。如果对于PC和IE两个子协议分别有不同的接入点，那么MUX的功能可能只是对到达的低层数据进行路由。

可能的系统结构的例子如图17所示。

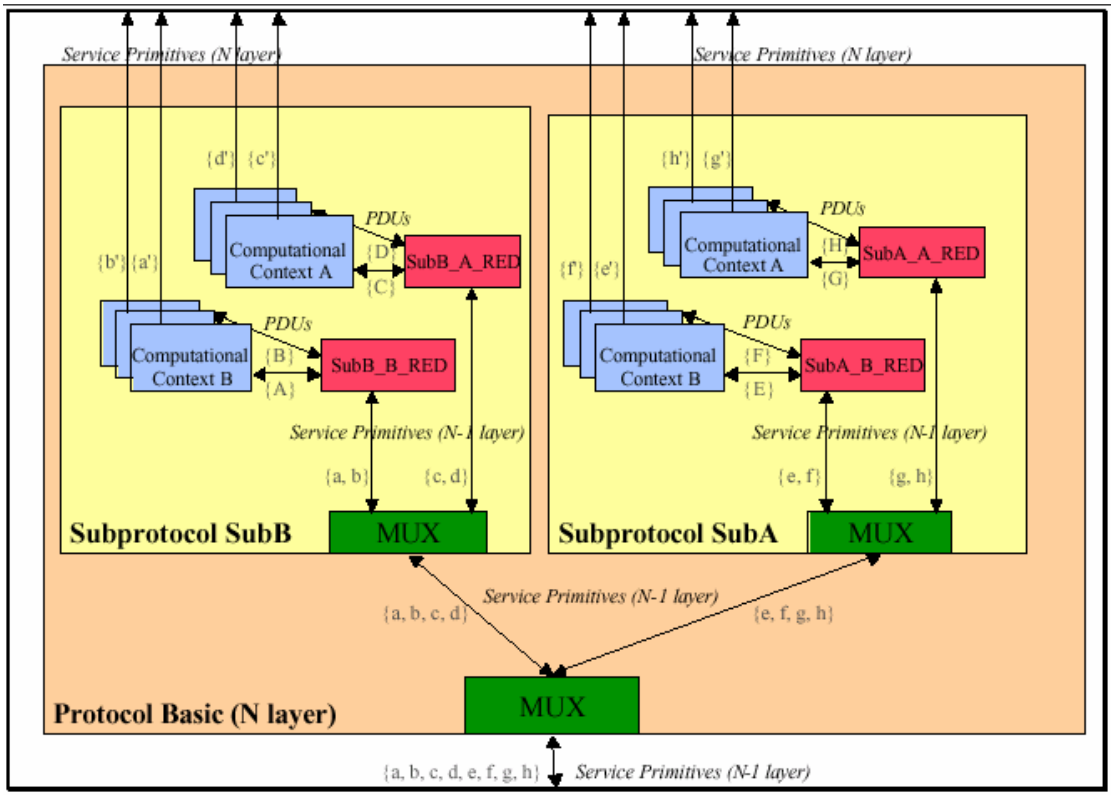


图17 系统体系结构

4.2 实现阶段的接口

为了便于实现位于不同网元的服务组件之间的通信，本例应用了编解码模式。因此，对端代理需要与提供数据传输服务的更低的协议层之间进行通信。高层需要使用低层提供的服务。

PDU接口的计算环境在前面的章节中已经定义。为了使两个对端代理能够进行虚拟PDU通信，对各自的计算环境定义PDU接口。

通过下层数据传输服务交换的服务原语（*service primitives*），PDU从某个网元上的协议实体传送到另一网元上的另一协议实体。通过RED和MUX等实体的使用，消息路由可穿过多个协议层次。通常，协议实体中对外部环境的可见服务原语接口被分解为RED和MUX的几个子接口。图17中，最外层的服务原语接口首先被分解为子协议的服务原语接口，然后再进一步被分解为每个计算环境相关的服务原语接口。

图18描述了PC_RED类及其接口。所有到达的消息都是通过mux_port接收的。pc_port用于PC服务原语交互，peer_proxy_port用于PDU交互。编码后的PDU通过sctp_port和SCTP服务原语发送出去。

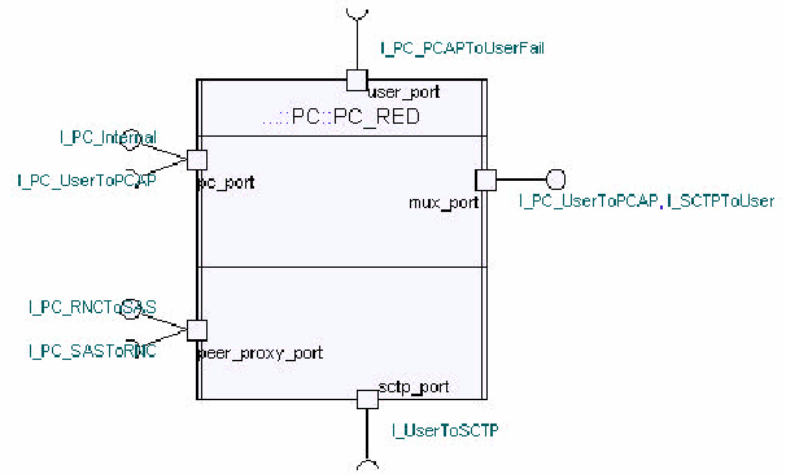


图18 PC_RED类

4.3 实现阶段的体系结构

体系结构图描述了用于自动代码生成的系统配置。作为功能组件的类是其中的部件。类之间的接口通过每部件的端口实例实现，端口之间的通信可通过连接器来表示。外部接口也在图中显示。

图19显示了PCAPServices的内部结构。这个实现模型中只包含了PCAPServices组件和计算环境。

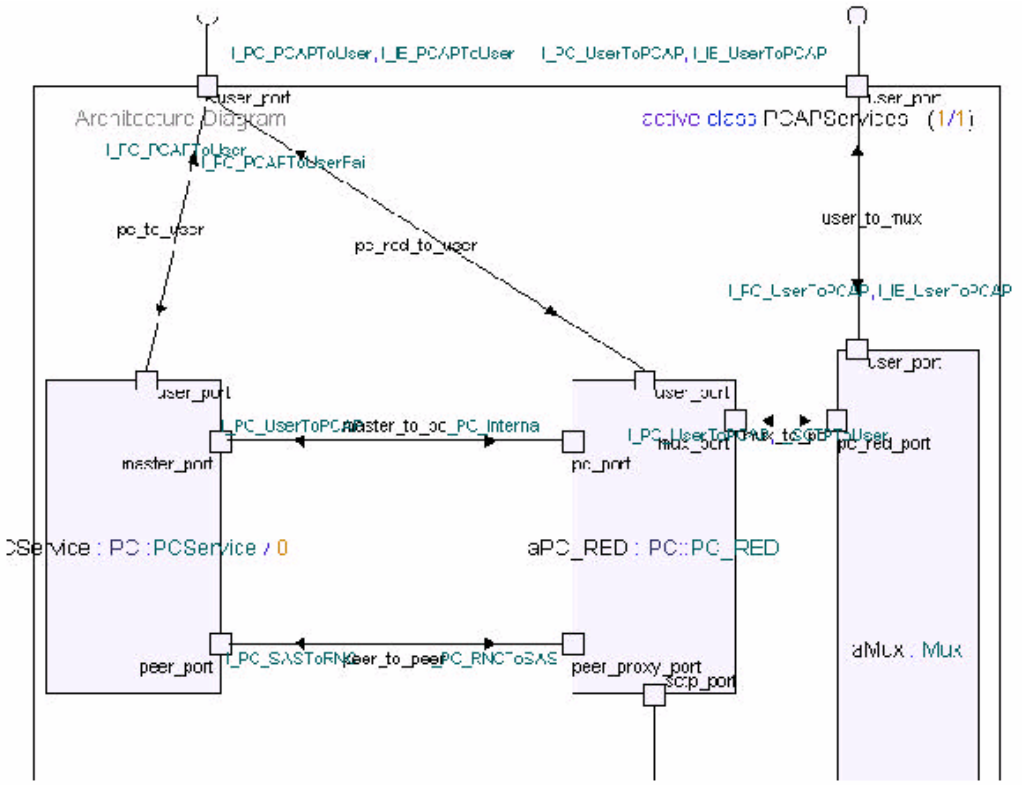


图19 实现的体系结构

4.4 实现阶段的行为

计算环境的大部分行为已在服务分配阶段定义。实现阶段运用了主从模式。一个计算环境的生命周期仅限于一个通信会话，该会话中，工人被创建、激活、最终自行终止。RED和MUX实体的行为应进一步说明，明确在服务分配阶段尚不明朗的细节问题。

图20显示了怎样向PCService类中添加动态机制（对象终止）和错误处理行为（定时器）。图21显示了PC_RED接收到PC_req服务原语时的响应。基本控制流看起来和服务分配阶段一样，但添加了一些细节。真正的实现细节隐藏在符号后面。

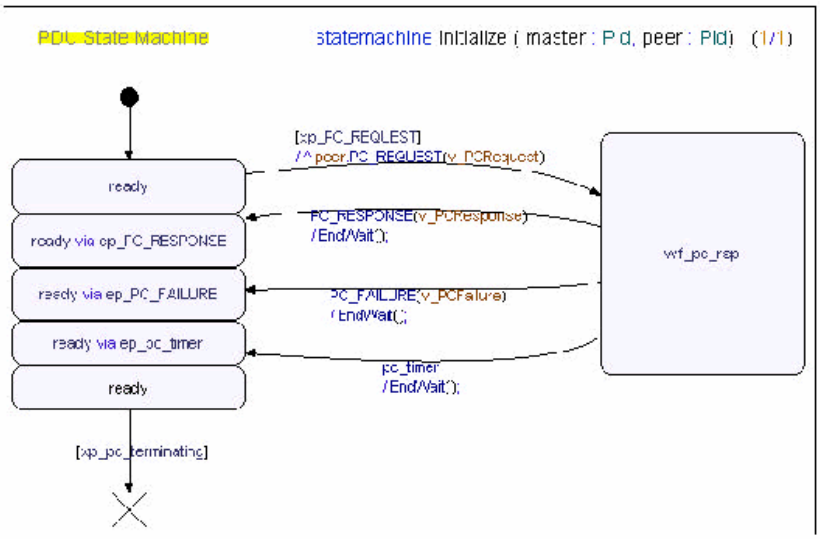


图20 PCService类的状态机

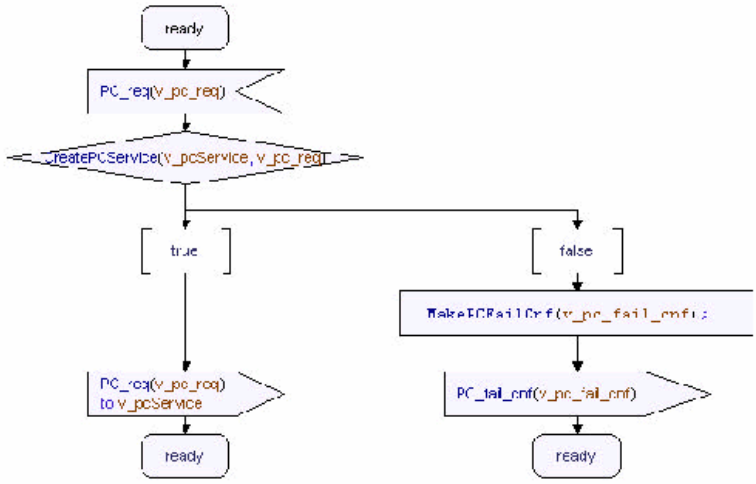


图21 PC_RED状态机的一部分

5.结论

本文描述了用于协议工程的一种模型驱动的设计方法，该方法使高层次需求说明逐步细化为详细的实现模型。高层需求模型展示了对系统有效行为的功能性需求，通过可执行的有限状态机来描述。功能（或行为）的内部结构，即实现系统的软件，通过服务组件来描述。服务组件之间的交互通过定义明确的接口来封装。每个服务组件的行为用状态机来描述，从最高层的含糊说明逐步求精，最终获得详细的实现模型。该模型可自动生成面向目标平台的可执行代码。逐步求精过程中各个模型的可追溯性可通过大量的仿真来实现。每个接口上的外部可见行为在求精过程中应维持不变。在实际当中，不同接口上的信号定义会经常更新，这在逐步精化过程必须要有所考虑。

本方法的目的使提高设计过程的生产率及其生成的软件的质量。不同抽象级别上的模型可通过仿真来进行评估，并为求精过程中的后续模型提供一个可填充的框架。在较早的设计阶段对核心算法进行分析，可降低费用，并从一开始就提高软件的质量。模块化工作使资源分配合理有效，并对软件组件的维护和重用提供了有力支持。封装设计过程中用到的启发式经验，并把它们通过模式的方式进行描述，可以把设计过程中的不同阶段固定下来，并使其具有可重复性。

提升设计工作的抽象级别，也就是说，从实现语言导出说明和建模预研两个功能，将极大地提高问题解决能力。设计者可以把精力集中在最本质的东西上——她或他的代码要解决的问题。

附录 1：UML2.0 的背景知识

统一建模语言UML从1997年成为标准后获得了巨大的成功。它代表了建模战争的结束，并在软件业广为使用。它最初创立的时候是作为一种语言实现“说明、构建、可视化和记录软件密集性系统的中间产物”。

标准建立后的5年间，工具厂商和用户积累了很多UML的使用经验，可以把该语言中最有效的元素和其他不能满足人们期望的部分分开。各种UML工具提供了一些非标准的功能，人们希望这些功能作为标准的一部分。一个典型的例子是模型的可执行性，它可以在早期对系统功能进行验证，而无需去钻研应用代码。

软件产业中，5年是一个相当长的时间，所以新的软件趋势超出了UML的能力也就不足为怪了。尤其是基于组件的开发给UML建模者造成了很多麻烦：不知道如何处理组件框架如COM+、EJB，也不知道处理嵌入式系统开发中典型的层次化分解。

2年前，OMG启动了创建UML2.0的工作——对于这种最流行的建模语言的新的重大修订——以便把上面提到的以及其他方面的问题考虑在内。现在来看看这项工作的成果，它的一些新功能将在后续章节描述。

本研究报告中，我们通过协议工程的一个案例的研究，验证了如何在实际中使用这些功能。在该案例中使用的工具是Telelogic Tau/Developer，它实现了UML2.0的许多功能，包括执行模型的功能。

A.1 主动类和被动类

类是UML中最为广泛认可的概念。在这次重大修订中，人们花了很大的力气使其更适合于大系统的开发。这意味着它们更加适合于嵌入式系统中基于组件的开发类型和基于组件的框架（如COM+、EJB等）的建模。

对于这类系统，其子系统通常分布在网络上，彼此并行工作。换句话说，每个子系统可以看作一个具有自己权利的系统。这种系统中，最好是区分出主动类和被动类，主动类用于描述系统的逻辑体系结构，被动类用于描述数据结构。从符号上说，主动类使用和普通类一样的符号，但是在边上有一条竖线（就象图22中的VendingMachine类一样）。主动类在它们自己的线程内运行，而被动类必须在其他类（如主动类）的环境中运行。（C或C++中的main函数与主动类的行为相似，代表了整个程序，通过由操作系统来运行。）

需要注意的是主动类趋向于具有接收能力，即接收异步信号的能力，而不是直接操作。与此相反，异步类大部分使用同步操作。嵌入式系统设计的主流是使用主动类。

为了降低分布式团队同时开发大型组件的复杂程度，必须在不同部件之间使用明晰的接口。从本质上说，每个组件都可以看作一个构造块，彼此之间可以通过特定的方式组合在一起。

A.2 供应接口和所需接口（Provided and Required Interfaces）

基于接口的设计在现代软件开发中的地位越来越重要，许多编程语言都集成了接口的概念。用户希望把每一部分开发成独立的实体，与系统的其他部分之间无关。要达成这个目标，必须要制定各部分及其环境之间的接口规范。为了方便对此建模，UML把接口分为供应接口和所需接口。供应接口以传统的棒棒糖符号表示，描述了一个类实现的服务；所需接口描述了一个类希望其它类实现什么服务。从符号上来看，所需接口和供应接口非常相象，都是棒棒糖符号，只不过圆圈变成了半圆。

需要注意的是供应接口是实现该接口的类的一种简单表达方式，需求端口是使用这个接口的类的简单表达方式。那就是说，可以通过类和接口之间关系对这些接口建模。从类的角度出发，它并不需要知道哪个功能模块实现它所需的接口对应的功能，任何通过某种方式实现这个接口的方式（并不一定是类）都可以提供相应的服务。

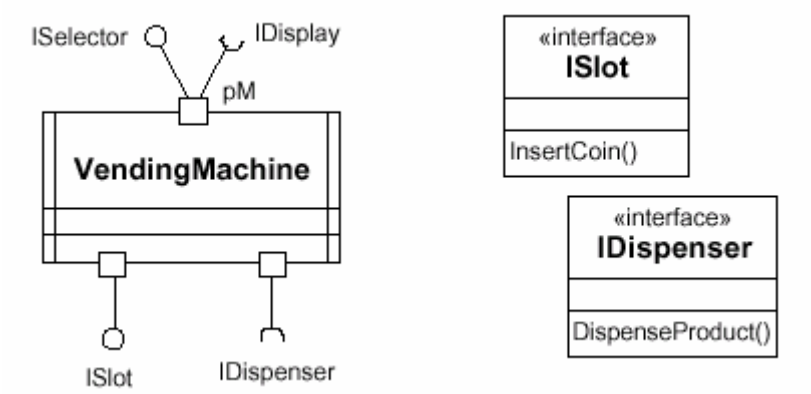


图22 供应接口和所需接口

图22显示了一个主动类VendingMachine的小例子。这里既有供应接口（ISelector 和 ISlot）也有所需接口（IDisplay and IDispenser）。这个类是一个黑盒子，我们不知道它是怎么实现的。但是，我们得到了使用这个类的足够信息，因为接口给出了该类所提供的服务和所需要的服务支持。最好能附加上如何使用该接口的说明，可通过顺序图、状态机或其他方法来实现。这些都可以描述消息是怎样交换的，服务是以什么样的顺序被调用的。类上的方块代表了端口，我们在下一节再进行详细描述。

供应接口和所需接口描述了类之间的交互约定，只有接口相匹配的类之间才能进行通信。接口匹配的情况有两种，或者类型一致，或者一个接口是另一个接口的子类。

A.3 端口

端口具有几种不同但相互关联的目的。首先，它可用于组合具有相关功能的接口。从这个意义上说，端口提供了类的一种视图。其次，端口可作为一个交互点，不同的类之间通过它连接到一起，形成内部结构的一个部件。

大部分情况下，对于那些具有端口的类，只允许通过端口访问它们的服务。端口就象类的封装硬壳上的一个洞，类通过它接收和发送消息。这种类的操作和属性都是非公开的，只能通过类的供应接口访问它提供的服务。图22中的类有三个端口，顶上的那个叫pM，拥有一个供应接口和一个所需接口。这个案例中，端口间的通信是双向的。一般来说，一个端口可以和任意数目供应接口和所需接口相联系。另外两个端口各自只有一个接口，一个是供应接口（IDispenser），一个是所需接口（ISlot）。这两个端口只支持单向通信，这意味着只能向具有供应接口的类发送请求并获得返回值。对具有所需接口的类来说，它可以向任何实现该接口的东西发送请求，并获得返回值。

A.4 由部件和连接器组成的内部结构

层次化分解是构造一个大型复杂系统的有力的也是最常用的方法。在某种程度上，对接口的讨论包含了这样的内容：如果一个项目太大，不能由几个人来完成，那么必须使用模块化方法。每个类可以看作模块的构造块，这些构造块也可以进一步分解为更小的构造块。换句话说，一个类可以把它的功能分解为另外一些类，这些类是先前类的组成部分。每个类都可以进一步分解，直至递推到最细颗粒的类，这种类只有行为，没有内部结构。使用这种方法可以构建任意大的系统。

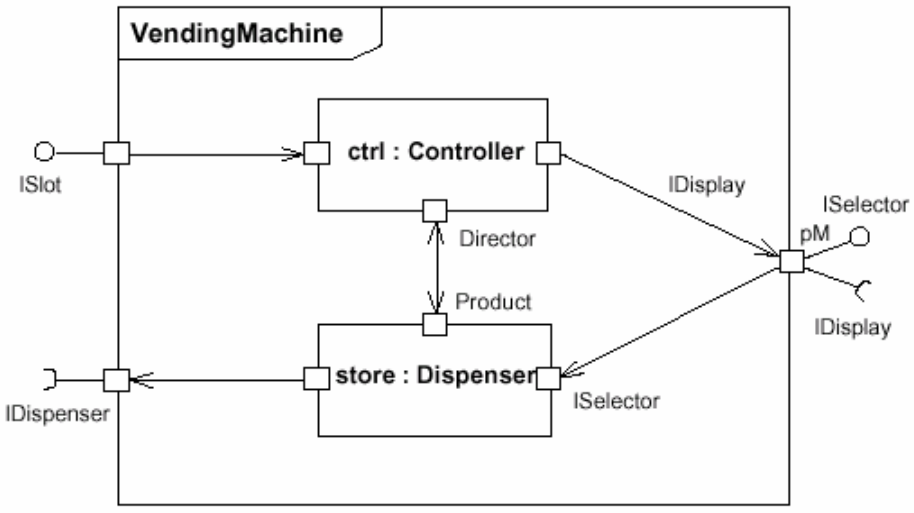


图23 类的内部结构

类的内部结构是由部件和部件之间的连接器构成的。每个部件代表了在容器类环境中的一个类，同一个类可以作为不同环境（内部结构）的部件，就象图23所示。这和图22所示的内部结构有关。也显示了如何运用端口作为连接器的连接点。连接器本质上是上下文的联系（association），只有在它所使用的环境中才有效，内部结构的连接器描述了系统可见部分之间有效通信路径。类也通过其他方式连接，作为其他内部结构的部件。上图中的store和ctrl的部件取决于Dispenser和Controller类，这两个类在这张图中没有表示出来。

正如其名称所示，层次化的分解建立在组装关系的基础上的，图24沿用了图23的例子，显示简单的组装关系。这两个视图互相补充，传达了不同的信息。类的内部结构只是构造体系中的一个层次，有必要将该类缩小或放大，以便从其他层次进行观察。进一步说，内部结构关注不同部件间的通信。图24中只显示图23所隐含的组装关系。这种视图不适合于展示内部结构中环境相关的信息。类之间的组装关系暗示着，容器类和它的内部部件生命周期之间存在着依赖关系：容器类被创建之后，代表部件的类实例才能被创建。同样的，当容器类的实例被删除时，所有部件实例也将终止。

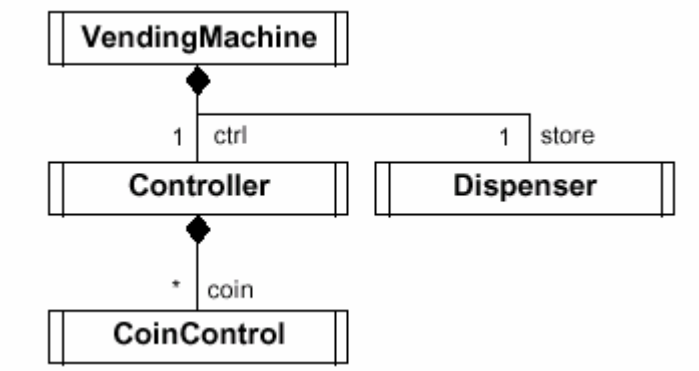


图24 组装关系结构

每个类被看作一个构造块，既可以分解为更小的块，也可以组合起来创建一个更大的块。这是处理遗留系统的一个常见办法，它们通常被看作接口不可更改的构造块。

A.5 行为端口

在前面章节的黑盒视图中，无法获知端口接收到的通信是由类实例处理，还是委派到其他部件类来处理。在显示了内部结构的白盒视图中，你可以指定发送到行为端口的消息不能委派给其部件类，必须由容器类本身来处理。端口之间的差异在黑盒视图中被隐藏起来，因为它代表了类的实现，不应该暴露。行为端口的符号是代表端口的方块上再连接一个小的状态标志。完全在类边界内的行为端口代表了该端口只能从类的内部进行访问（即从其他部件进行访问）。图25显示了行为端口的例子。

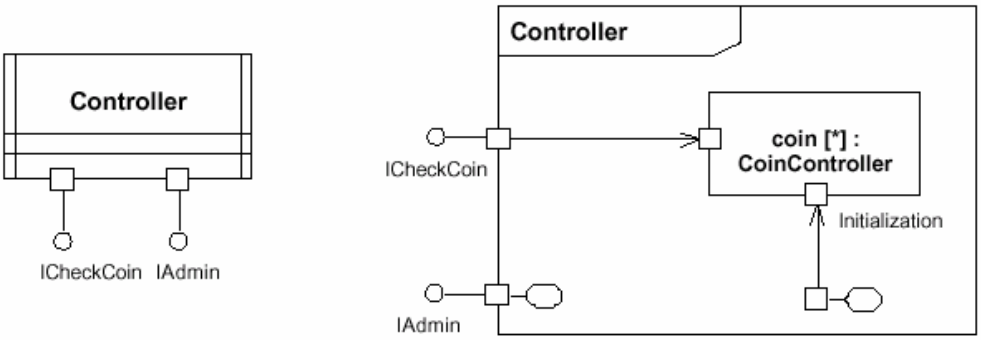


图25 类的行为端口

左图是Controller类的定义，它的内部结构在右边。行为端口直接和类的行为相连，类的行为是通过状态机、活动、交互等来表示的。在这个例子中，容器类Controller负责创建和初始化CoinController类的实例。它有一个受保护的行为端口，仅用于创建实例所需要的初始化信息，从Controller类外部不能访问CoinController类的这个初始化端口。

A.6 行为描述

到现在，我们还只讨论了UML2.0中结构性方面的内容。然而，这种语言在行为方面也做了重大修订，尤其是对交互图和活动图方面做了全面修订。对本白皮书来说，不需要这些新概念进行细致的研究。

尽管交互图颇受欢迎，但它在UML的地位一直被低估。在整个开发生命周期内，都可以使用交互图，而且它们在表达需求、功能和测试方面也表现出色。此外，它们很容易理解，即使对不懂UML的人来说也是这样。图5和图10是交互图的简单例子。不幸的是，UML1.x中交互图的能力太有限了，对于更大的系统来说无法真正实用。UML2.0中的最重要的改动如下：

- 在顺序图中可以引用其他交互图，这样就不必在多张交互图中重复相同的信息。使用这种方法可以很快地在已有行为的基础上生成新的行为，举例来说，生成测试版本。
- 通过在一个框架封装一个以上消息的方式来表达变化的情况。这种变化情况包括：交互、决定、可选项等等。这种方式减少了用于表达动态功能的顺序图的数量。
- 把生命线分解为一张新的交互图，该交互图表示了被分解的生命线所代表的对象的各部件之间的消息流。这项功能使你能够象对待被分解的对象一样放大或缩小交互图。

状态机的变化不象其他行为的变化那么大。最显著的变化可能是元模型的简化（元模型是用于定义语言的），但这并没有对用户造成多大的影响。但是，现在定义复合状态更加简单了，这些状态也都可以有进入点和退出点。这些点的工作方式与端口一样，它们断开了状态内部和状态所处的外部环境之间的连接，使你可以定义特定的进出状态的点。复合状态的一个例子是图14中的空闲状态，它的子状态在图15中显示。

A.7 动作

可执行的模型是UML2.0中一个重要的词汇，使模型的可执行性成为可能的是这样的一个事实：动作定义的粒度可以和大多数编程语言相比。然而需要注意的是：UML并不是作为直接的编程语言使用的，必须要和一个或多个包含不同语法变化点的profile结合，使其具有较大的灵活性，同时这也有利于把UML与具有基本数据类型的数据模型相捆绑。进一步来说，因为语言本身没有提供相应的符号，因此应当使用具体的动作语法符号把profile与动作联系在一起，以便大多数用户可以访问该动作。

动作包括分配、请求、循环、决策等，和其他语言中的statements相对应。

动作最先是在UML的动作语法中定义的，该语法是后来添加到UML1.x中，但UML2.0将该语法和活动图整合，成为更具一致性的语言。动作可用于描述非常细节化的行为，如操作、状态转换、活动等。

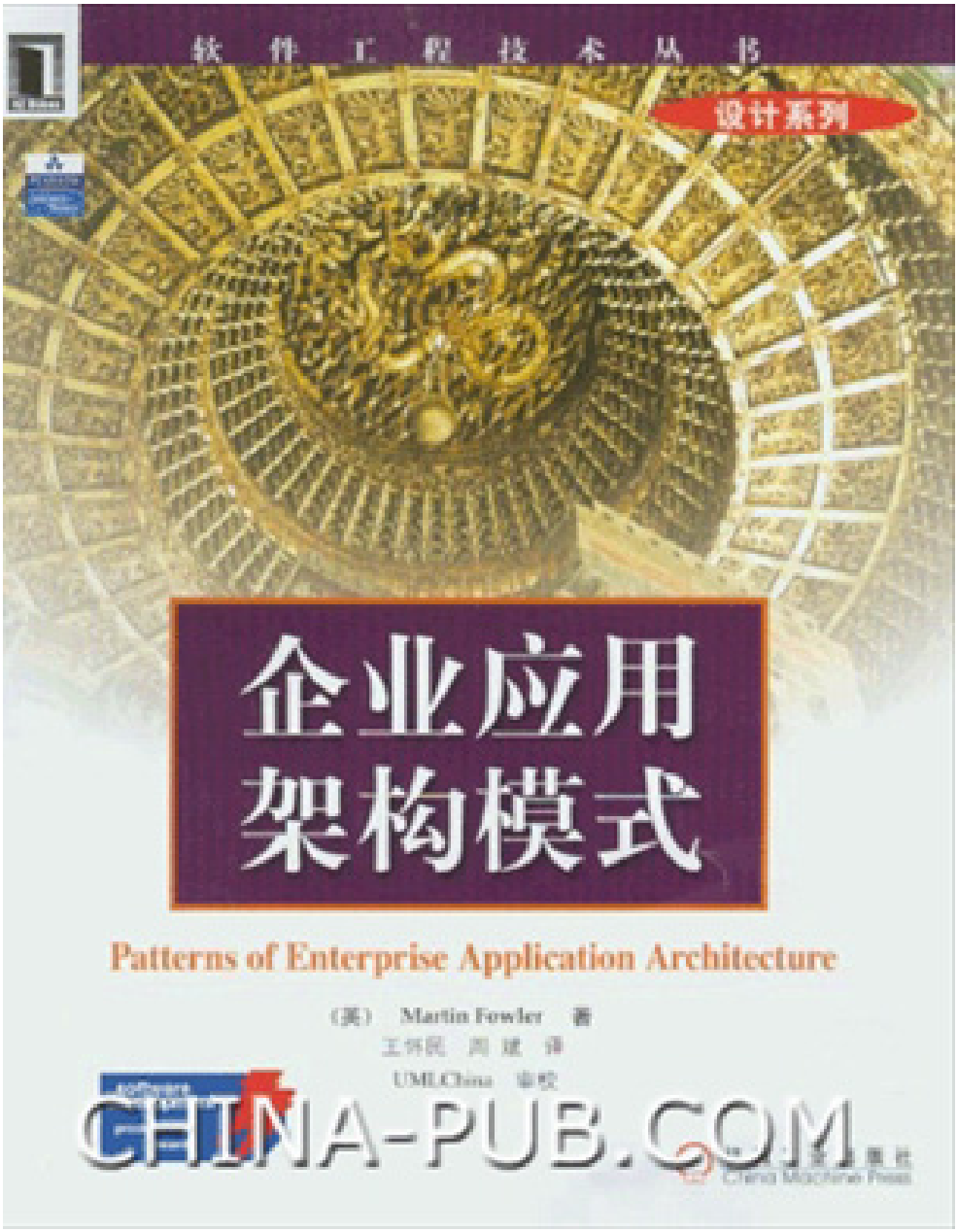
可执行的模型的最大的好处在于它提供了在编码之前验证系统正确性的能力，而且可以在生命周期的较早阶段发现错误。可以对可执行模型进行仿真和调试，也可以在模型上运用各种检查验证技术。同样重要的是：一旦模型正确工作并且你对模型感到满意时，可以通过模型自动生成大部分代码，如果不是全部的话。这使我们可以集中精力于模型的功能定义，让代码生成器考虑内存分配等编码相关事宜。

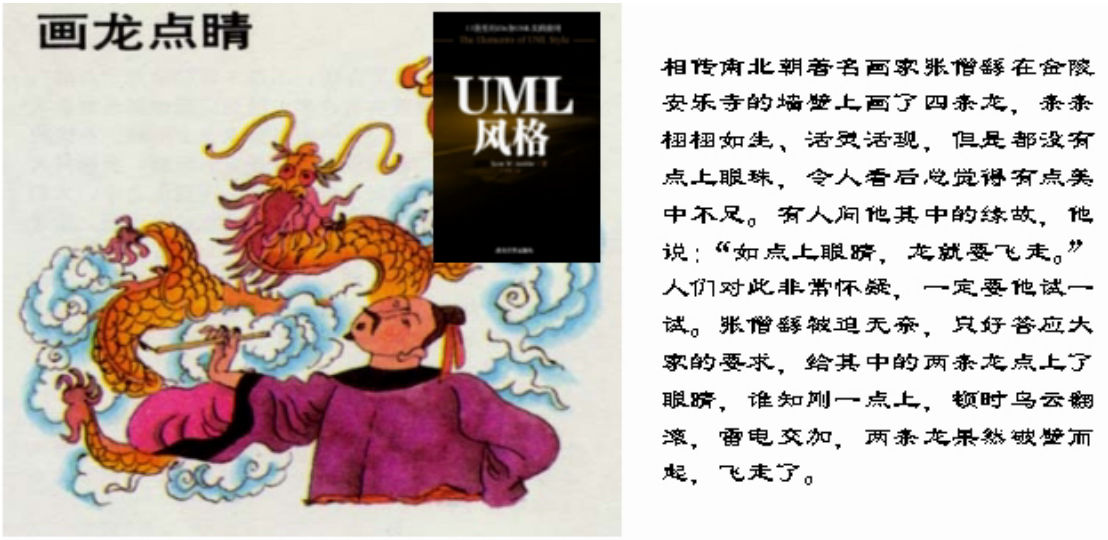
按照这种方法开发系统意味着你可以重新分配开发力量。大部分时间花费在分析和设计阶段，此时也可以经常测试的系统。同时，可降低系统实现的时间，因为大部分代码已经自动生成。实现之后的测试阶段也被大大缩减，因为大部分测试已在早期阶段完成。需要注意的是，在开发阶段的后期发现错误并更正要付出巨大的代价，这意味着能够在早期检测和修补漏洞将节省大量的时间和金钱。

A.8 模型驱动的开发

并不是所有模型都必须是可执行的。实际上，所创建的大多数模型是不可执行的，主要用于在开发过程涉及的各种人员之间传递信息（也就是出于交流的目的）。UML2.0中保留了UML1.0中大部分功能和灵活性，但是使该语言具有更高的内聚性。同时，该语言也被扩展，以便在获取大型系统的行为和结构时具有更好的可扩展性，并在设计行为细节时具有更好的精确性。

UML2.0适应于整个开发过程，允许你对需求、分析、设计、实现、测试进行建模。模型驱动的开发这个词意味着模型是开发的核心，也是应用开发的基础。整个系统用模型表示，使团队成员更易于理解和参与，减少了对关键架构设计师的依赖，也使新成员可以更容易融入项目。进一步的说，这种开发方式把重点放在系统功能上，而不是需要生成的代码上。然而，要真正从模型驱动的开发中获益，模型执行和代码自动生成功能是非常重要的，因为它们提供了更快地建设更好的系统的方法。





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。

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

(英) Martin Fowler 著

王怀民 周斌 译

UMLChina 审校

中译本已出版>>

软件工程与软件工艺

紫云英 著

国内近来流行“软件工艺”的隐喻，很多朋友都立志要成为“软件工匠”，而且曾在网上看到“颠覆软件工程”的说法。

持工艺观的人对软件开发的“机械工程”隐喻的一个主要批判点是，“在机械工程中，把设计转换成产品是需要很高成本的，工程化是为了降低制造成本。而在软件开发中，把设计转换成产品——即编译、链接，以及复制光盘的过程，成本是很低的。所以‘机械工程’的隐喻不适合于软件开发。”也就是说，持工艺观的人认为，可执行的程序或者其载体——光盘，才是产品，而“源代码就是设计”。

但是，若按照这样的逻辑，我们要么承认机器码也是设计，要么承认 3GL 代码和汇编代码，或者汇编代码和机器码，有着质的差别——因为设计和产品显然是两码事嘛。而我们都知道，从机器码到汇编到 3GL 代码并没有质的差别，只是抽象层次的逐渐提高，越来越多的系统知识被封装进语言本身。今天的源码可能就是明天的目标代码（想想 50 年代的汇编和今天的汇编吧）。

所以，以上的批判显然是站不住脚的，更何况，即便承认“源代码就是设计”，那么“机械工程”的隐喻也仍然是适用的。在机械工程中，CAD 软件（计算机辅助设计软件）可以极大地提高设计效率，那么既然有 CAD 为什么不能有 CASE 这类工具呢？

机械工程和软件设计的本质差别并不是设计和产品之间的转换成本。设计也是分不同层次的，转换成本位于所有层次之间。对于智力产品而言，或许把承载产品的光盘看作最终产品恐怕也并非合适。（想想，如果是通过网络发行，岂不是没有产品了？）

软件工艺批判了工程观提倡的“足够好就行”，但不幸的是，从商业的角度看，“足够好就行”确实是合理的。软件工艺的隐喻表明，程序员应该像工匠一样学习和成长，但不幸的是，关于软件开发的知识总量要比打铁工艺或者其他任何工艺的知识量都大出很多，而且还在不断增长中，而很多软件项目的复杂性也非木匠、铁匠所打造的木制品、铁制品所能比，而且随着各行各业的信息化的深入，我们可能会需要越来越复杂的软件。在这样的现实图景中，工艺的隐喻会遭遇重重困难。下面我将从经济的角度和复杂性的角度来对比分析工程和工艺。

批量需求带来批量经济效益，从而引发工程化。任何创造过程，只要人们对创造的结果有大批量的需求，人们就会在经济利益的驱使下把这个创造过程变成工程。即便一开始工程方法不成熟也没关系，巨大的市场需求会促使人们投入大量资金来改善工程方法，直到工程产品的成本足够低，质量足够好（好到可以接受的程度就可以了，但未必超过手工制品）。这是经济力量决定的，和创造过程本身的特性并没有太大的关系。哪怕人们看作艺术的音乐作曲这样的过程，通过计算机作曲软件来把它“工程化”而大批量生产乐曲也是可能的，而且现在的技术已经可以使得计算机生成的乐曲的质量接近大师经典作品（一般人听不出两者区别）了，但是“音乐工程”没有流行，这主要是因为其产品——各种不同的乐曲，属于奢侈品而不是生活必需品，对其没有很大的市场需求。更极端一点，假设人们现在因某种原因迫切需要大量小恐龙，那么资金就必然会大量流入遗传工程的研究，那么很可能《侏罗纪公园》中的一幕会成真，批量生产恐龙的工厂会出现。只要存在强大的经济利益，连养育后代这样事情都可工程化，还有什么是不可以工程化的呢？

大量生产使我们对生产过程更加了解，工程化因此成为可能。批量效应不仅体现在经济利益上，也体现在工程过程的创建之中。正是大批量的需求使得人们得以对足够多的产品进行共性和差异性分析，总结出一定的模式和规律，把一些东西标准化自动化，从而使得工程化成为可能。工程制品未必都是“清一色”的。以汽车工业为例，人们对汽车有大批量的需求，于是汽车生产就工业化了，带来的结果是成本大幅下降，质量也足够好。但现在从奔驰的生产线上下来的奔驰车，可能没有两辆是一模一样的，很多部件都可以替换，更不必说颜色等了。但是这依然是工程的产物，是借助计算机和先进的生产设备从流水线上下来的，而不是手工打造的。当然，价格最贵、档次最高的汽车依然是由顶级工匠手工打造的，要比奔驰、法拉利、劳斯莱斯等都贵上很多，要购买的话可能要提前几年排队，因为生产一辆需要很长时间。但这已经是富人的奢侈品了。而通过这样的工艺是无法支撑起现代社会生活对产品的大批量需求的。

工程未必追求最佳的质量，但是能够控制质量。对工程产品和工艺产品看法的一个误区是工程产品的质量会比较高。其实未必是这样的。比如，很多人就觉得妈妈或女友编织的毛衣（工艺产品）要比商店里买到的机器编织的毛衣（工业产品）要好；手工制作的糕点和馒头要比机器制作的好吃；天然的食品要比工场化养殖/种植的食品好吃，而人工养殖/种植的食品又要比通过生物技术手段生产的人工合成蛋白质食品好吃。当然，前者的价格也往往比后者贵，而质量差距往往其实也没有你想象得那么大。事实上，工业产品的质量一般都在某个中等范围内，不会太差，但工艺产品的质量可能会参差不齐。（可能你妈妈织的毛衣非常精美，但你女友的作品却一个袖子长、一个袖子短。）总的来说，这些产品并不复杂，手工制作是可行的，质量差异也是可以接受的。工程主要扮演了降低成本的角色。若降低幅度并不太大，人们也不介意花更多的钱或者时间/精力，那么工程和工艺产品会在市场上长期共存。

工程是克服复杂性的手段。虽然工业产品的质量未必一定比工艺产品好，但也有一些领域，比如建筑，只有通过工程化才能获得质量足够好的产品，通过手工艺或许能搭出精品茅草屋，但是无法造出现代的摩天大厦。这里，工程化不仅是为了降低成本，更主要是为了攻克复杂性。现代载人航天工程是另一个典型的例子。飞机也是。由顶级工匠历时多年手工打造的超豪华轿车是奢侈品，并且有人愿买敢开，这是因为轿车虽然复杂，但复杂性还在工艺能够应付的范围之内（当然，已经只有少数最顶尖工匠才能应付了，所以物以稀为贵，价格非常非常高）。但相信没人能手工打造出一架波音 747 飞机，即便有人造出来了，我也不相信有人敢坐着它飞上天。神州 5 号飞船和阿波罗登月飞船就更不是手工能够打造的了。

我们再来看一下工程和工艺对经验的不同处置方式。持工艺观的人说，年纪大的、有经验的开发者都离开了开发第一线，所以我们才会反复犯过去的错误。确实有道理，但是，因为软件开发这个领域的发展速度远非传统手工艺所能比，所以讨论这个问题就不得不谈及知识的积累。随着人类发展，知识越来越丰富，而人的进化却相对缓慢，比如人的脑容量的进化速度显然更不上知识增长的速度。那么如何去应付这样的局面呢？目前，人们的做法是抽象（封装）和分工。抽象（封装）的意思是把一些知识用已经证明的定理，或者已经制造出来的组件，或者数控机床这类（半）自动化设备，或者包含了现有最佳实践的标准过程来封装；分工的意思则是个体关注的领域越来越窄。这似乎与软件工艺的图景不太符合。虽然让程序员很早就离开开发第一线不是好的做法，但解决方法也并不是照搬传统手工艺模型，“推迟退休年龄”那么简单。或许，还是该向工程方法寻求解决之道。工程的做法是总结出 patterns/antipatterns 或者 best practices/bad smells，并尽量把这些 patterns/antipatterns 形式化，用工具来封装这些经验，从而可以让工具自动检测甚至改正。我并不是说这可以完全代替人脑，但至少可以避免很多错误，并可以减轻开发人员的很多记忆负担和操作负担。

在工程方法中，工具（生产设备）很重要，流程很重要，因为依靠这些后人就可以在前人的积累的基础上前进，而不是必须自己也走过从学徒到大师的漫长的积累经验的过程（或者至少过程可以缩短）。当然，工匠也倚重工具，但是我们习惯上并不把操作数控机床生产铁制品的人叫做铁匠。工艺观在强调人的主观能动性的同时，也把太多本可以由工具承担的工作交给了人本身，把太多本可以形式化的知识归入了神秘的“经验”之中，这显然无助于提高生产效率，也无助于整个行业的积累和前进。

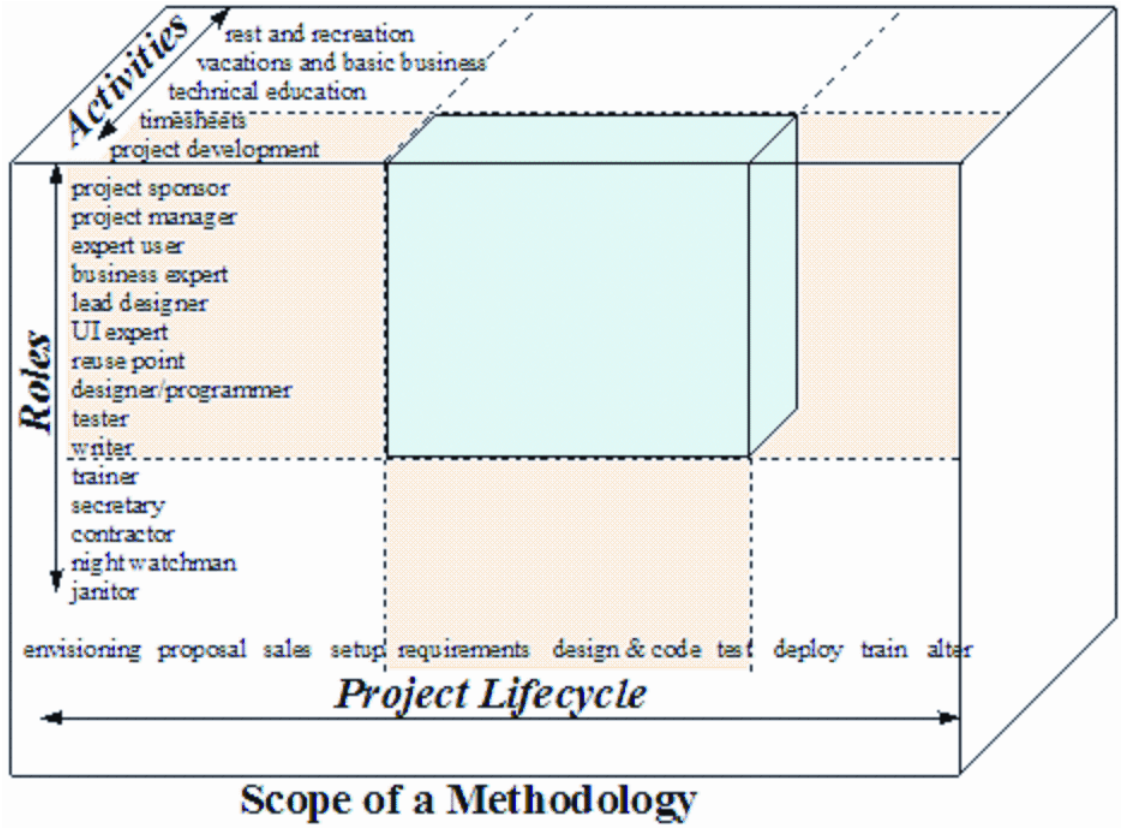
软件工艺的支持者说，工匠应当只选择那些经过较长时间考验被证明为可靠的工具。我记得在《软件工艺》一书中 Pete McBreen 说的是十年。Pete McBreen 在书中有句口头禅“It’s crazy”，我这里忍不住也要套用一下：这种做法简直是 crazy。这不仅意味着我们应当忽视 GP、MDA 这些新进展，也意味着我们不应该使用现代的 IDE 和各种 Framework，以及 XDoclet、Ant 这类可大幅提升效率的工具，而应该用 Emacs 编辑代码，用 GNU 编译器或者其他具有悠久历史的编译器编译，而且还要用 10 年前的版本。那时候好像 C++还没标准化，STL 还没出现，Java 语言和 C#语言还没发明呢。

归根结底，工艺隐喻和快速发展的软件开发行业的现状存在矛盾。工艺观给人这样一幅图景：历经几个世纪缓慢发展着的手艺，慢条斯理但又精益求精地学习、工作着的学徒和大师们。学徒们云游四方，师从不同的大师，几十年过去了，于是自己也成为了大师。是的，手工业给我们带来了高质量的产品，从业者也悠然乐在其中，这很美妙，美得像乌托邦一样，但这幅图景显然和软件业的快节奏以及社会对软件的大批量需求不协调。

我无意否定软件工艺隐喻的有价值之处，而且也喜欢“软件工艺”这个隐喻，因为它给人亲切感。而且工艺观强调人的重要性，这一点都没错，我很赞同。我开始学习编程的时间很早（那时候苹果机还流行，286 是高档电脑），从感情上说，我怀念那个个人英雄的时代，所以连带着也喜欢“工匠”这一隐喻（我将之理解成技艺高超的个人）。但是客观地说，对于工艺和工程之争，我个人觉得，对于讲求实际的程序员们，名称并不重要，内容才重要。叫它工艺也好，叫它工程也好，只要能提高软件开发的生产率就行。但现在有不少人举着软件工艺的旗号在反对软件工程（“颠覆”这个词实在令人印象深刻），以软件工艺为借口而反对罩在“软件工程”帽子下的新知识新工具新方法（特别是和 UML 沾边的东西），这却是很不明智的。下面我们且来看看《软件工艺》一书的作者自己是怎么说的。

Pete McBreen 在第 3 部分的导言中说：“In applying the apprenticeship model to becoming a craftsman, we can draw on the experiences and cultures of other crafts. In borrowing these traditions, we need to allow for differences between software and traditional crafts and to ensure that the advances made in software engineering are not lost. We need to be like blacksmiths, using the extra tools that engineering gives us and using the knowledge gained by science to supplement what we know from practice.（第 3 段）”他还说：“Craftsmanship is not a rejection of science and engineering, but rather a subtle blending of man, machine, and knowledge to create useful artifacts. This blending is the aim of craftsmanship--that is, obtaining mastery over science and engineering so that we can continually refine our craft.（第 4 段）”

以上引文对应的大致中文意思是“我们不应当抛弃软件工程已经取得的成就，工艺学也并不反对科学与工程”。由此看来，软件工艺的主要提倡者都没有“颠覆”软件工程的意思，那么我希望读者也不要因为喜欢“软件工艺”的隐喻而忽略了 Generative Programming、MDA 这些重要的软件工程新进展。



最后，我想借用 Alistair Cockburn 在 Agile Software Development 一书中的方法学空间维度图来阐述我的观点。显然，软件工艺所强调的“人”的因素，和工程所涉及的工具、方法、过程等其他因素在一个方法学中是共存（并互补）的，而不是非此即彼的。工匠也离不开工具的帮助。你见过用拳头钉钉子的木匠吗？你见过不用炉子锤子而用“三昧真火+一阳指”打铁的铁匠吗？又有多少人可以不用锯子而用“降龙十八掌”砍树？要知道，工匠也是非常倚重工具的。所以，请不要盲目拒绝 Generative Programming、MDA 这样的“新型锤子”、“新型斧子”或者“新型数控机床”。

本文最初发表于《程序员》2004.07 期，经《程序员》同意刊登。

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

使用 Together 让你的项目变得更加敏捷（上）

周小辉 著

讨论

为什么要敏捷？

当前，在软件工程领域刮起了一股“敏捷”之风，重量级的软件过程方法不再成为人们热衷的话题，与其相对，诸如 XP、测试驱动这样的轻型方法却越来越受到人们的关注。

这种趋势从图书出版市场也能发现得到，例如，在最新的第十三届 JOLT 大奖中，由大师级人物 Robert C. Martin 历时多年写就的《敏捷软件开发　原则、模式与实践》一书就获得了软件开发震撼大奖。

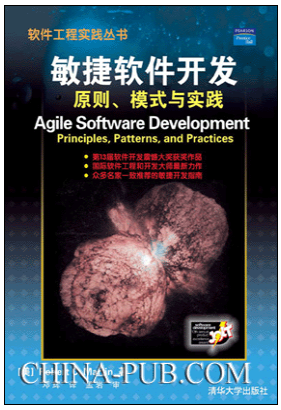


图 1 敏捷开发的巨著

就是雄踞软件开发过程方法多年的 RUP 也在寻求使自己变得更敏捷的方法，例如在笔者于《非程序员》第 29 期所翻译的文章“使 RUP 敏捷”中，就介绍了一个如何裁剪 RUP，使其适合小型项目因而更敏捷的案例。

敏捷方法得到了众多大师级人物的支持，著名的如《测试驱动开发》一书的作者 Kent Beck，《重构》、《企业应用模式》等书的作者 Martin Flower。这些大师目前都活跃在软件业的第一线，他们在软件开发实践中，深刻认识到一些使项目成功所应该遵循的基本原则，并身体力行，不断总结实践，并不断发表文章、书籍等加以宣传，从而形成了今天敏捷方法欣欣向荣，为广大软件从业者所深刻认同的局面，并出现了一大批能有效支持敏捷理论的开发工具。

谈到这里，各位也许要问，“敏捷”方法到底是个什么东西？它真有这么神吗？为什么有这么多的人对其趋之若鹜？它是软件业的又一个银弹吗？

基本上来说，笔者从“敏捷”方法里看到的最鲜明的一个特点，就是两个字“务实”，这对厌倦了花架子和大量文档的软件从业者来说，无疑是最适合他们胃口的。

敏捷方法有多个分支，如 XP 方法，特征驱动的方法等等，他们并不是互相排斥的，就目前来说，XP 方法应该最为大家所熟悉了。在 XP 方法中，提出了 4 个核心价值和 12 个实践活动。四个核心价值是：

1、沟通；

2、简明；

3、反馈；

4、勇气。

12 个实践活动是：

1、计划游戏；

2、测试；

3、结对编程；

4、重构；

5、简单设计；

6、共同拥有代码；

7、持续集成；

8、现场客户；

9、小版本；

10、每周工作 40 小时；

11、编码标准；

12、系统隐喻。

那么，说了这么多，采用敏捷方法，到底能够给软件开发项目带来什么样的好处呢？

由于敏捷方法通过采用务实的方法，不折不扣地贯彻软件开发过程中的最佳实践，缩短了软件的交付周期，因而也就降低了项目开发的风险，使项目成功的可能性大大提高（当然，它不是银弹^_^）。

敏捷需要工具的支持

无法想象这么多的最佳实践，如果没有工具的有效支持，将会成为什么样子。没有工具，开发员将讨厌重构；没有工具，自动化的单元测试将成为空话；没有工具，持续集成也将会因为集成周期过长而成为口号.....

我们需要什么样的工具？[1]

目前，就建模领域来说，已经有很多厂商在这个领域角逐，因而也出现了很多这方面的产品，每家都宣称自己的是最好的，能满足客户的需要。具体的清单可以大家可以参考《非程序员》杂志从第 26 期到第 30 期的连载。

面对这么多的工具，很多人常常会挑花了眼，这时候就会出现错误的选择，有些人或者随大流，或者看价格来确定产品的优劣。笔者在工作中也接触过一些人，言必称“Rose”，似乎 Rose 才是最好的工具，可以满足他的所有需要。笔者使用 Rose 和 Together 多年，也和一些业内资深人士探讨过，相比之下，感觉到 Together 要更为实用一些，尤其是在敏捷特性支持方面，读者可以参考本文后面的讲述。

那么，什么才是选择工具时应该考虑的客观判断标准呢？也就是说，这个工具应该具有什么样的特性呢？笔者认为，它应该具有以下基本特性：

[1]在此所指的敏捷工具主要指以建模为中心的建模工具。

1、能支持团队工作

现在的软件开发不再是以前那样小作坊形式的作业了，而是团队的协同开发，管理复杂性成了软件开发过程中的一个主要问题，因而 CASE 工具应该具有沟通机制，能使团队的协同工作成本降到最低，并能有效提高团队的工作效率。

2、能与软件工程领域的其他工具进行集成

软件开发过程包括五个主要的阶段：需求、分析设计、开发、测试和部署，在迭代、增量的开发模型下，这五个阶段不是截然分开的，而是互相重迭的。因而建模工具要能够与其他各个阶段的工具具有集成能力，从而形成一个各个 CASE 工具协同工作、无缝集成的局面，这将极大提升工作效率，减少需要人为干预的工作量。

3、全面支持 UML

我想这是一个基本要求。

4、具有反向工程能力

反向工程能力是一个很有价值的能力，设想一下让你分析一个缺少文档、原开发人员无法找到的软件，如果面对的是一堆代码，而不能站在更高的抽象层次去观察系统的结构或交互行为，将是一件多么痛苦的事情。

5、能自动保持源代码和模型的同步，无须人工干预

自动保持源代码和模型的同步，是一个建模工具必须具有的特性，否则你将会经常范糊涂：到底是源代码更新，还是模型更新一些呢？

6、支持重构与测试

重构与测试是一对孪生兄弟，XP 方法中将重构和测试作为其 12 个最佳实践中的两个实践活动之一。在 Martin Flower 的《重构》一书中，强调在重构之前，必须具有坚实的测试代码（这里的测试指的是单元测试）。

没有哪一个软件是不需要修改的，没有哪个软件的设计是尽善尽美的（除非从来没人使用），因而能够对软件进行持续改进就显得很有价值了。当对软件的设计改进难于进行时，Flower 建议我们先对软件进行重构，重构就是“在不改变软件的外部行为的情况下，改进其内部的结构，使其更易于修改”。

7、支持模式

模式是历经锤炼、经过证明的最佳设计方式。建模工具应该支持多方面的模式列表，包括通用模式、语言特定的模式（如 J2EE 模式）等。

此外，建模工具的模式特性应该是可扩充的，允许开发员定制自己的模式，并借助于团队特性，从而得以在团队成员间共享。

8、具有强大的文档生成能力

技术人员大多很厌烦写文档，如果建模工具可以做到帮助开发员自动产生所需要的文档，无疑可以极大提高生产效率，将开发员从繁重的文档工作中释放出来。更好的是，具有这一特性以后，就再也不用担心文档与模型或代码不同步了，因为我们可以随时根据模型或代码得出所要的文档。

Together 正是这么一款工具

也许你还在找寻一个能满足上述所有特性的工具，那么，我告诉你，Together 就是这么一款工具。通过多年的不断努力，如今的 Together 已经成为了建模领域的领导者，它具有以下的多个特性：

（一）支持所有的 UML 图形

UML 图形已经成为了建模领域的标准，之所以要建模，是人们天生就希望用尽可能简单的形式交流，而“一张图胜过一千句话”，对应到建模领域，我们可以解释为：

1、一张类图胜过一千行代码；

2、一张用例图胜过千百次无休止的会议。

通过将头脑中的观念用用例图或类图的形式表示出来，我们可以只需要一张图就做到传递大量的信息。

Together 支持全部的 UML 图形，并支持 XMI 标准使得可以将模型以 XML 规范的方式导出。

（二）能自动进行模型与代码的同步

Together 的 LiveSource 技术能帮助你做到模型与代码的自动同步。

源代码与模型的持续同步能力使得程序员摆脱了某些 CASE 工具需要手工去作这些同步的烦恼，这在进行重构工作时也得以充分体现。此外，由于源代码与模型是时时同步的，因而重构不仅可以在代码中进行，也可以在模型图中进行，当然，这取决于你的爱好，也许你喜欢模型图的直观，也许你喜欢修改代码的放心（其实不用担心^_^）。

（三）自动生成文档

Together 具有强大的文档生成能力，包括你可以定制自己的文档模板，从而使开发员可以将更多的精力集中到分析和设计上。

（四）广泛的模式支持

Together 支持业界常用的模式，如 Gof 模式、J2EE 模式等，并可以让开发员定制自己的模式，从而使得模式的复用成为现实，这将极大提高公司所有项目的架构质量。

（五）重构、测试、审计和度量

Together 支持《重构》一书中的多个重构技巧，并具有强大的测试框架生成能力，从而使得可以在一个集成开发环境下完成重构所需要的步骤。

审计可以保证软件的质量，以技术的手段施行企业的软件规范。

度量则站在一个更高的角度，使你可以看到软件设计的质量。

结合使用审计、度量和重构，可以使得你的重构工作更见成效。

（六）支持团队工作　支持与其他 CASE 工具的集成

Together 通过与 SCM 工具（Borland StarTeam、CVS、ClearCase 等）的集成，得以支持团队工作。

除了版本控制工具外，Together 也与领先的需求管理工具进行了集成，其中包括 Borland CaliberRM 和 Rational RequisitePro。

通过集成，Together 将整个软件开发的各个环节无缝地连接到了一起，信息共享变得更加方便，生产力得以提高，这不正是“敏捷”方法所强调的吗？

敏捷特性 1：团队合作与集成

TCC 通过实现与多个版本管理工具的集成实现团队协同工作。各位可能要问，它是怎么做到的呢？

由于 Together 将组成项目的各个元素保存为单独的文件，这样它们就得以成为一个有效的配置项，由于粒度足够细，因而自然就支持团队在同一个项目上工作了。也许在同一时刻，某些人工作于分析设计图上，而其他人则工作于代码上。

下面就以 TCC 与 StarTeam（关于 StarTeam 操作的有关知识，读者可以访问 www.oochina.org 获得笔者撰写和翻译的一些有关 StarTeam 的一些文章）的集成为例，描述一下 TCC 的团队工作特性。由于 TCC 能够与 StarTeam 进行集成，这使你能够在无须离开熟悉的工作环境就能利用到 StarTeam 的版本控制功能，这样你就可以：

1、在 Together 的集成环境下进行建模而同时无须离开环境就可以对项目文件进行检入/检出操作；

2、将每个文件上发生的变化与变更请求或需求、任务等建立链接关系。

在 Together Control Center 中，可以选择多个版本控制工具作为团队合作的基础，下面笔者就以 StarTeam 作为范例演示如何进行团队协作。

1、笔者采用的是 TCC6.2 版本，而 StarTeam 则为 5.3，由于 TCC6.2 只能与 5.4 以后的版本进行集成，因此要实现与 StarTeam 的集成，首先需要安装 StarTeam 的 SCC 集成工具，读者可以到 Borland 官方网站下载；

2、安装完 SCC 集成工具后，接下来的一步就是在 TCC 中将 StarTeam 作为缺省的版本控制工具：

1)、选择主菜单上的“Tools->Options->Default Level”，将打开 Default Options 对话框，从左边的树中选择 Version Control 节点，然后在右边的 Use 下拉列表框中选择 SCC，如图所示：

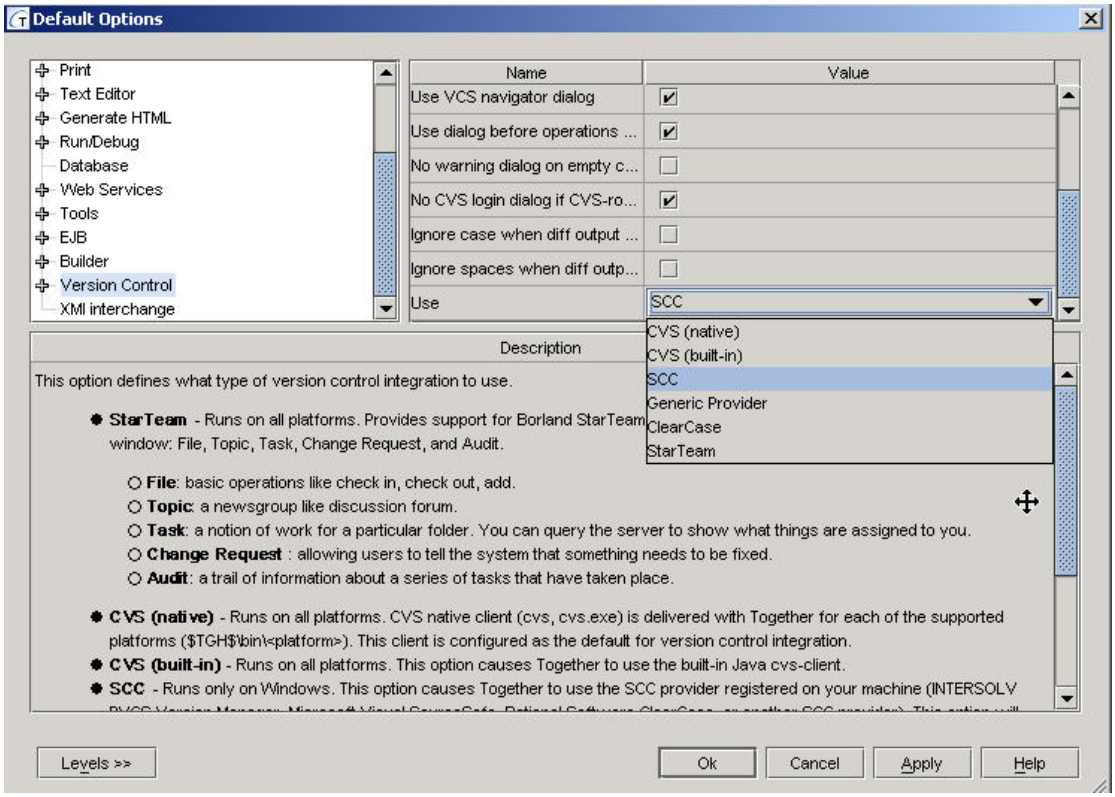


图 2　在 TCC 中将 StarTeam 作为缺省的版本控制工具

2)、点击“OK”按钮。

3、将 TCC 项目与 StarTeam 项目建立关联，既可以将 TCC 项目作为一个新的 StarTeam 项目，也可以将其与一个已有的 StarTeam 项目建立关联，下面就以 TCC 自带的 CashSales 项目为例，演示将其作为新的 StarTeam 项目（读者可以参考相关文档查看如何将 TCC 项目与已有的 StarTeam 项目相关联）：

1）、打开 CashSales.tpr 项目，选择主菜单上的“Project->Project properties”，将打开“Project Properties”对话框，如图所示：

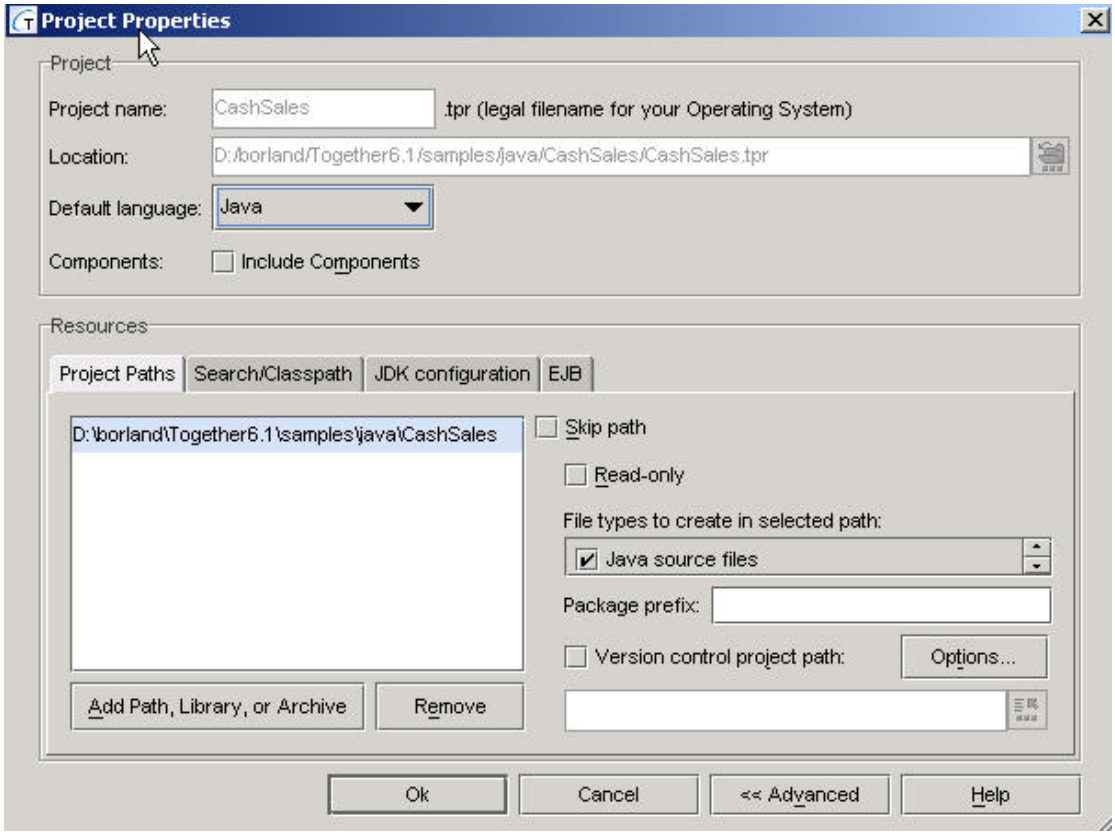


图 3 将 TCC 项目与 StarTeam 项目建立关联－步骤 1

2）、选择“Version control project path:”复选框，然后选择旁边的“Browse”按钮，将打开“Select a StarTeam Project for Source Code Control”对话框，如图所示：

图 4 将 TCC 项目与 StarTeam 项目建立关联－步骤 2

3）、点击 Create New 按钮，将打开 Create New StarTeam Project 对话框，如图所示：

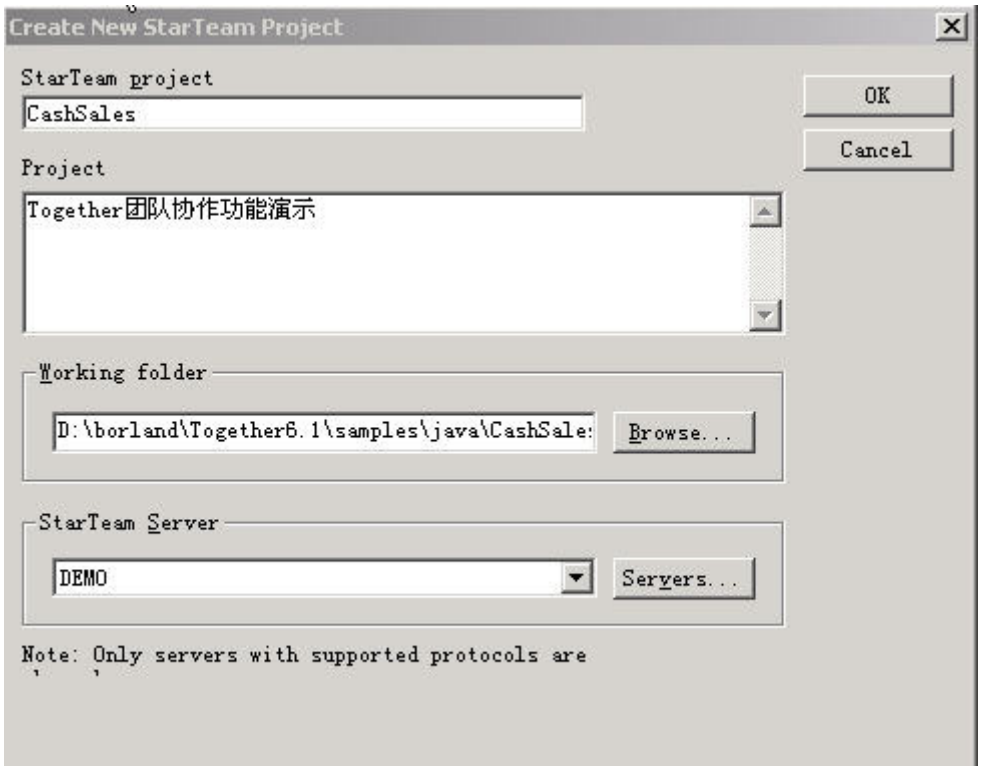


图 5 将 TCC 项目与 StarTeam 项目建立关联－步骤 3

可以在其中选择（或配置）StarTeam 服务器，工作文件夹，输入 StarTeam 项目的名称及描述性信息。

4）、点击“**OK**”按钮，这时将弹出登录对话框，提示你输入用户名及口令，如图所示：

图 6 将 TCC 项目与 StarTeam 项目建立关联－步骤 4

输入正确的用户名和口令后，点击 OK 按钮，这时就回到了“Project Properties”对话框，这时你可以观察到对话框中的文本框中已经多了一行显示版本控制项目路径信息的文本，如图所示：

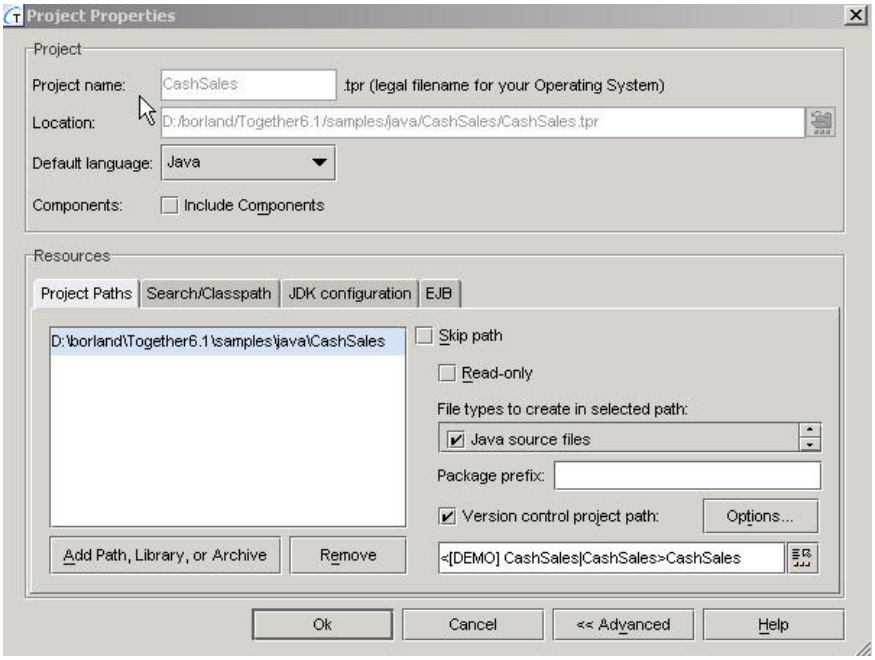


图 7 将 TCC 项目与 StarTeam 项目建立关联－步骤 5

点击“OK”按钮，这样就完成了在 StarTeam 中创建一个新项目的过程。

4、接下来就是将 TCC 项目中的文件加入到 StarTeam 的版本控制之下了：

1)、从主菜单中选择“Selection->Version Control->System…”，将打开“System”对话框，如图所示：

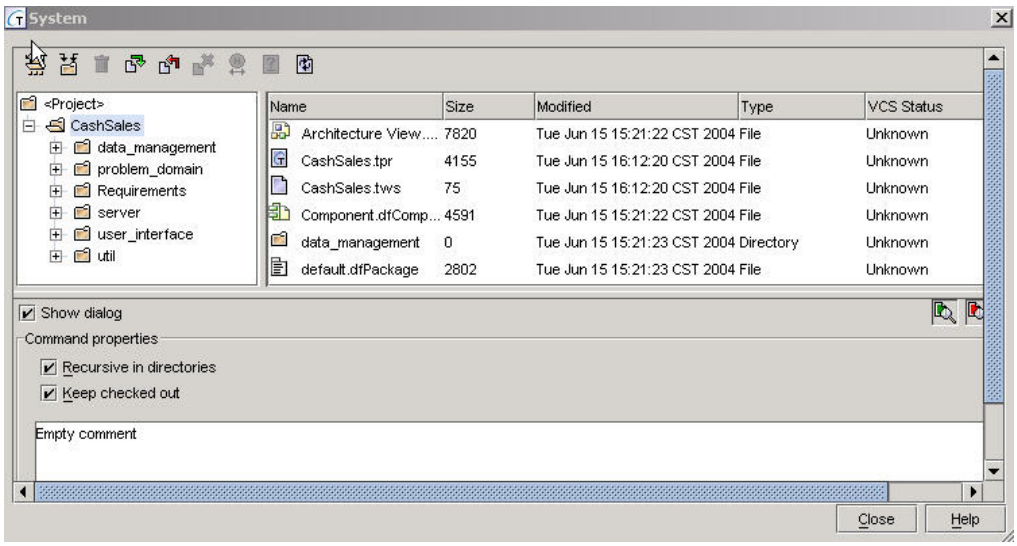


图 8 将 TCC 项目中的文件加入到 StarTeam 的版本控制之下－步骤 1

2）、选择左边的树中的第一个文件夹“CashSales”，然后点击上方的“Add”按钮，这将弹出“Add to Version Control”对话框，如下图所示：

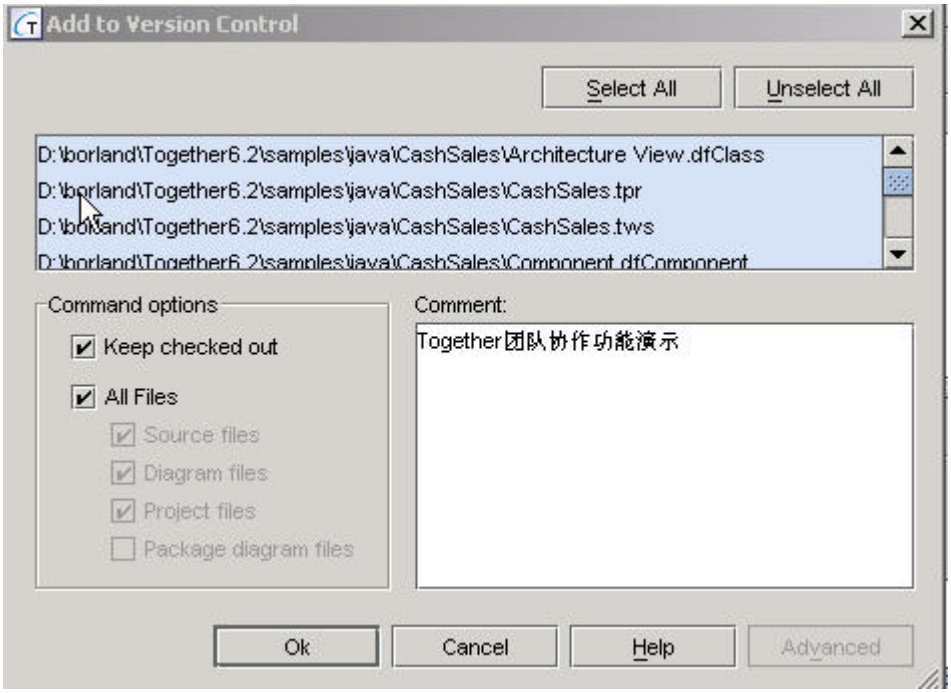


图 9 将 TCC 项目中的文件加入到 StarTeam 的版本控制之下－步骤 2

在其中输入注释，并选择“Keep checked out”复选框，这将使得在 StarTeam 库中为这些文件加锁，然后点击“OK”按钮，这将把这些文件添加到 StarTeam 库中。

依次选择其他文件夹，将它们添加到 StarTeam 库中。

这样整个过程就基本告一段落了，在以后的使用过程中，你可以使用“System”对话框进行常见的配置管理操作，如将新产生的文件添加到版本控制之下、获得文件的最新版本、查看版本历史等等。例如如下图就是在 TCC 中显示文件变化历史的情形：

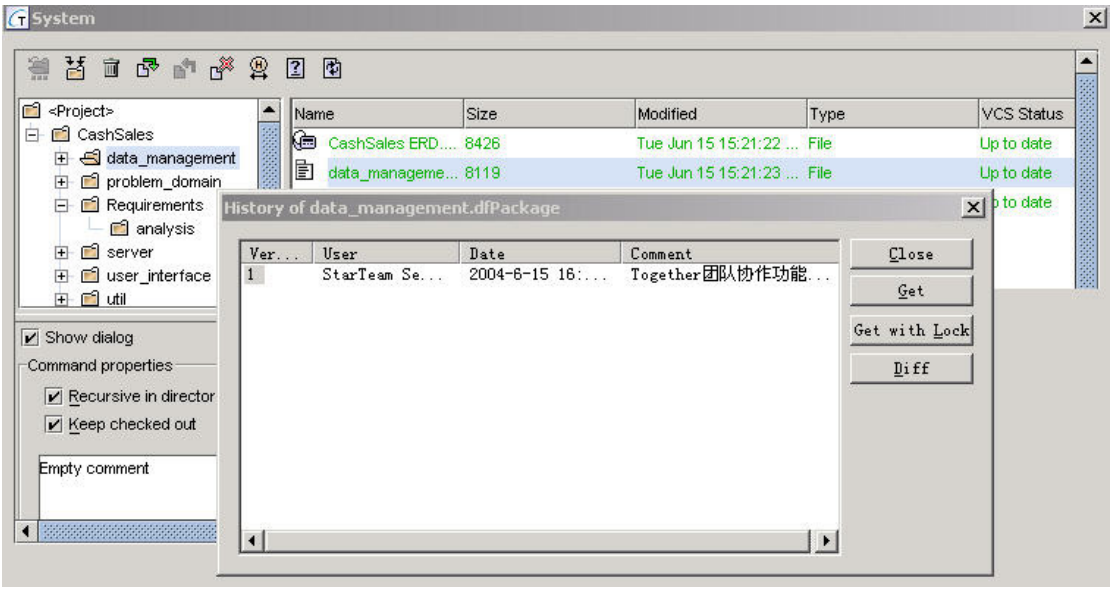


图 10 查看文件的版本历史

除了与配置管理工具进行集成外，TCC 还能够与需求管理工具进行集成，如 CaliberRM,Doors 等，这进一步使得整个软件生命周期的各个活动得以无缝地连接到一起。

敏捷特性 2：建模与文档生成

Together 具有强大的建模能力，包括：全部 UML 图；XML 建模；DB 建模；JSP 建模；序列图反向生成等。

通过建模，加强了团队成员之间的沟通。例如，笔者在进行 PeopleSoft ERP 系统的运行维护时，为了深入了解系统在处理业务时的商业逻辑，更为了与业务人员的沟通，使用 Together 根据对系统的了解绘制了如下的费用报销状态图：

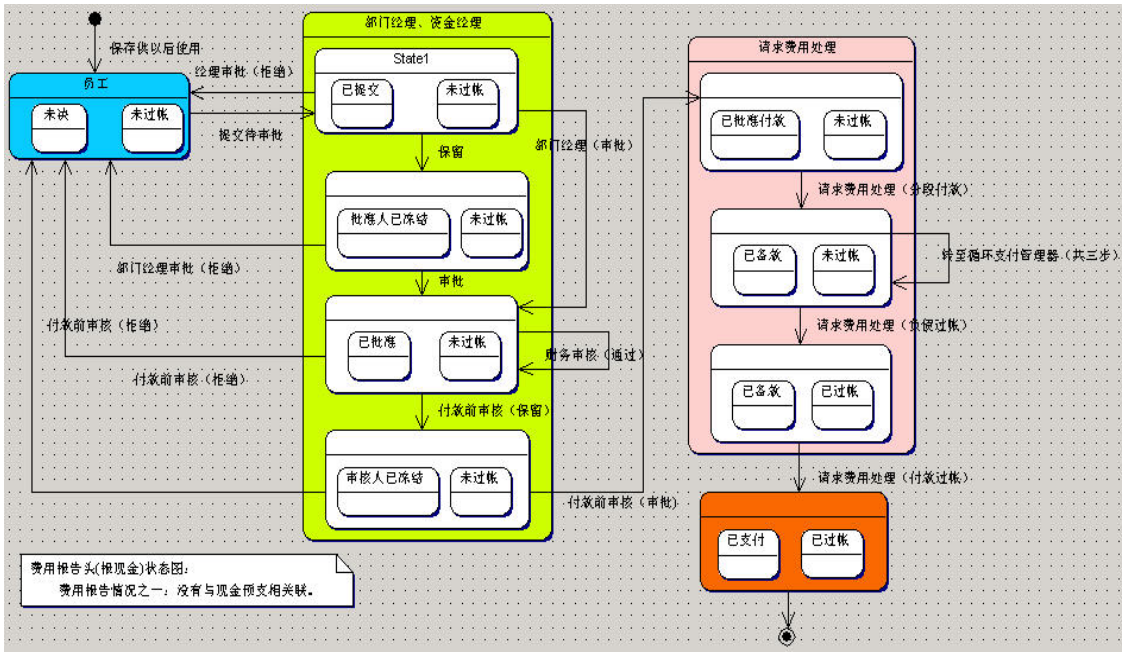


图 11 Together 强大的建模能力帮助理顺业务逻辑

模型建好后，可以使用 Together 的文档生成特性生成文档，例如，下图就显示了笔者使用 Together 所生成的文档的屏幕截图：

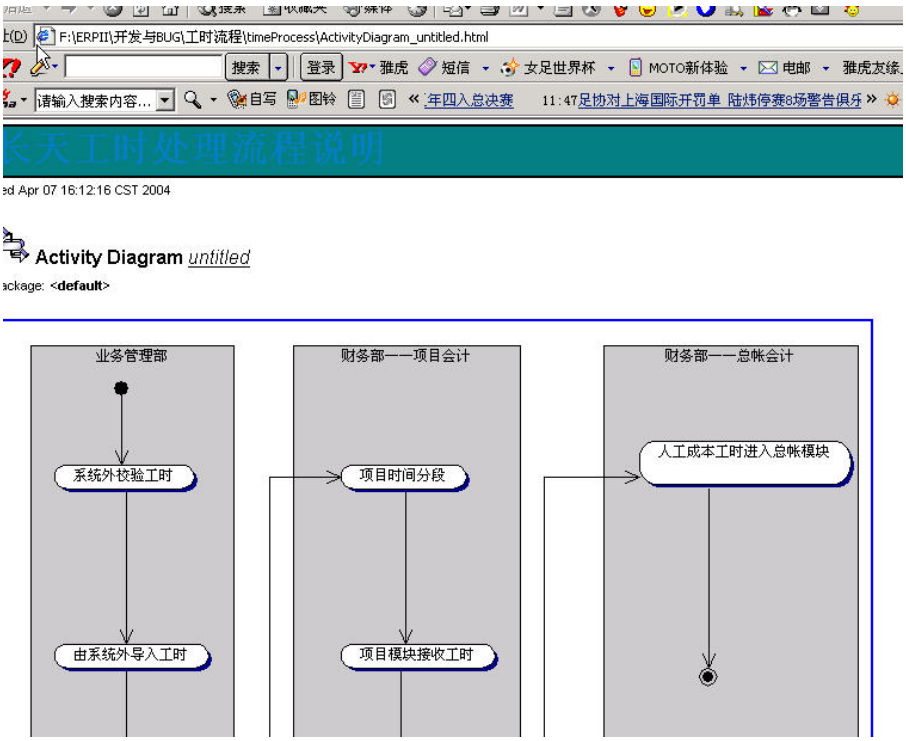


图 12 强大的文档生成能力与建模相得益彰

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

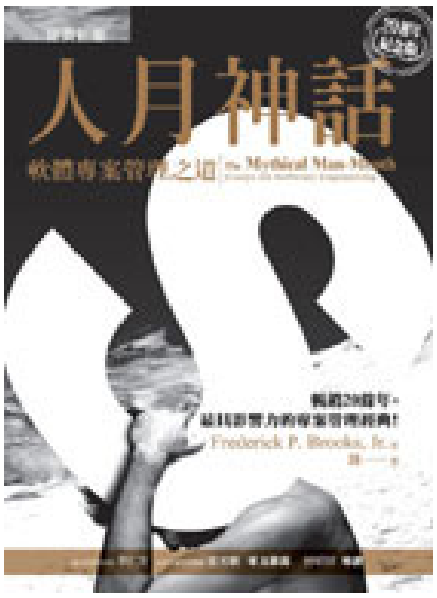
Scott Ambler

《人月神话》、《人件》近期动态

UMLChina 整理

讨论

《人月神话》、《最后期限》台湾版



讀書有感--人月神話

http://blog.forlady.net/archives/000127.html

May 29, 2004

看到小女子介紹人月神話這本書，請不要以為是羅曼史小說或者傳奇故事。

它是一本二十年前出的、講三十年前軟體專案管理問題與經驗的書，原名是 The Mythical Man-Month。其中的很多事例，現在仍存在，其中很多經驗，現在拿來用都還來得及。這次的版本為 20 週年紀念版。

曾經接過急迫且大型的網站建置專案，但是進行上卻還算順利，這是由於系統工程師十分有經驗，在分工、次序與掌控上，都做了很好的安排。

但在最近的一個朋友的經驗中，卻是遇到網站的機制改版，在時程與資源運用上卻不斷延遲，還好是內部機制，不用對客戶交代，否則所蒙受的將不是只有金錢上的損失。

我們知道時程延遲，但原因在哪裡？為何會發生？

人月神話的第二章，應該是這本書的精華，也是書名的由來，看完這個章節，發現作者道出很多軟體專案上的管理關鍵。而軟體專案是不可以完全套用傳統產業的專案管理方式。

第一個關鍵 樂觀

首先作者提出了一件常常發生的事情，這也是一個朋友身上發生的故事:『程式設計師都是樂觀的傢伙。』而且我發現越年輕越樂觀，但是無可避免的這是個年輕的產業，很少會遇到很有經驗的程式設計人員，這些年輕人，即使告訴他這樣是會有問題的，他也不改其樂觀。

所以『他們所做的第一個錯誤假設是一切都會進行的很順利』可是沒有想到一個 bug，就可以花上一天時間也無法解決。然後，就延遲了後面本來假設可以順利進行的部分。

第二個關鍵 人月

人月，是我們預估和排定時程用的，但是作者提出一個前提:『使用人月必須要在人力與工時可以互換的狀況下。而且要當工作可以切割、投入工作的人不用溝通，人力與工時才能互換。』就是說要可以互換，才能使一個人做 30 天，與 30 個人做一天的結果一樣，不然單純用人月去估算時程，一定會有誤差。

為什麼呢？首先軟體專案有其連續性，作者舉了個例子，蠻傳神的，他說，生小孩就是要九個月，你叫多少個媽來生都是一樣。第二點是因為即使工作可切割，但是需要溝通，越多人投入就需要多的溝通與教育的時間，所以一個人做 30 天的工作不可能用 30 個人做一天就完成。

所以，Brooks 定論說，在一個時程已經落後的軟體專案中增加人手，只會讓它更加落後。

因此，要讓一個專案順利進行，首先要有良好的時程規劃，考慮所有的因素，而且，程式人員要有勇氣不要妥協，堅持自己預估的時程。就像是廚師一樣，即使外面的客人在等著上菜，一隻雞要烤多久就要烤多久，不能因為妥協就用大火烤焦了。

外科手術團隊

這個篇章看完突然想到研究所的專案管理課程，三個學分，卻是我們一個學期的研究重心，四個實際運作的專案，大家分組進行，教授不斷的製造專案危機，像是公司被併購、人員流失、專案形式改變、客戶不斷施以壓力等等，但最後大家還是順利的結案。

這是個難以忘懷的經驗，組員們的默契在起初的確造成危機，但是，由於大家的素質還算整齊，很快就可以建立溝通的語言。加上大家熟悉協同科技概念與網路溝通，可以做到分時分地作業，也使得專案進行有效率。

但當我看完這篇外科手術團隊後，我發現其實還有個成功的關鍵在其中，就是同學們各有專長，所以，分工很容易，又因為有選出 leader 作為掌控者，所以，不會亂。大家不會做重複的事情。每個人按照時程交差，就可以順利完成。

作者認為一個外科手術中，有操刀的大夫，有護士、有麻醉師等等，各司其職。而不是像屠夫團隊，每個人都得拿刀。一個團隊中只有一個人操刀，其他人扮演支援的角色。這樣整件事情就會出自一個腦袋不會亂，也不需要不斷的溝通協調。

這兩個篇章是我比較有感覺的，寫出來跟大家分享。

最後，作者還有一個人月的概念我覺得很重要，這會影響到組織的公平性：一個好手，花一天可以做完，但同樣一件事，庸才卻需要三四天以上，甚至加班趕工，也許還領加班費。如果照這樣去計算人月，一定會差之千里。

有程序员把“人月神话”作为其 blog 的栏目

http://www.blogbus.com/blogbus/blog/index.php?blogid=14837&cat=1

其实读一本好书也是有要求的

http://blog.joycode.com/aspdian/posts/6356.aspx

前天，我的朋友 sniper 说他在读《人月神话》，并从中得到了很大的教益。这本书很早以前就听说过，而且还是很受推崇。但自己一直没有去读过，只是感觉自己还没有去读的水平。。如果没有书所设定的认知基础很难读懂，就算是读了也只是过眼就忘。一本好书更值得反复读，“温故而知新，可以为师矣”。为师很难说，但知新一定是有的。

什么是软件开发的核心问题？

http://forum.javaeye.com/viewtopic.php?t=1710

dlee 写道：

按照软件工程鼻祖，《人月神话》作者 Brooks 在“没有银弹——软件工程中的根本和次要问题”一章中阐述的思想，软件开发的核心问题就是如何从概念上对一个复杂的业务系统进行建模。这个建模是含义广泛的，不仅仅包括对象建模，还包括数据建模、算法建模等等一系列的内容。总而言之是要先找到解决复杂问题的突破口（先要搞明白需要做什么，然后再考虑如何做）。至于采用什么表示方法（简单文本、UML 图、E-R 图）、采用什么高级语言、是否一定要用面向对象、使用什么开发工具都是次要的问题。

有些人每做一个新项目，不管三七二十一先画一大堆图出来。其实你看《编写有效用例》这本书中用到了很多图形吗（当然我不否认图形在有些场合的重要性）？很多时候可能你只需要把你的想法简单地在纸上画出来（一只铅笔、一张白纸，成本接近于 0）就足以帮助你理清思路了。如果你的想法都无法清晰地画在纸上，那么 ROSE 这样昂贵的工具就能帮助你了吗？还是你只能装装样子，否则混老板的工资于心不忍呢？

软件开发中，工具是非常重要的，完全有必要非常纯熟地掌握所使用的工具。但是一旦掌握以后，就应该进一步思考一些深层次的问题了。按照我的看法，开源软件解决了工具的问题，然而如果你一味依赖开源软件，而不去利用这些很好的基础解决复杂的问题（建造更加复杂的架构），那么等于是把你的明天拱手相送了。

我有很多计算机书，买的和从网上下载的。我把这些书分成两类：可以增加知识的，可以增加智慧的，并且优先阅读第二类书。几乎所有的书只要你买来都能增加你某方面的知识，但是第二类书可以使你变得更聪明。《人月神话》毫无疑问属于第二类书。你要问我读这本书的感受，我的感受就象是大热天踢球后喝了一杯清凉的饮料，套用一句广告词：“从这到这都舒服”。

jlinux 写道：

软件开发的核心问题？ 我自己的回答是人.

这里面包括很多, 人, team, 公司等等, 从一个人到一群人, 不管是开发软件的人也好, 使用软件的人也好,都是由人为主体的. 而其他不敢是工具也好,还是方法也好, 不都是人在操作和使用吗.

这正是《人件》中所阐述的思想，我们以后会另开一个线索讨论《人件》中所关注的问题。国外资深的开发人员都读过《人月神话》和《人件》这两本书。国内的开发人员却不屑于读这些“没有用处”的书，而只对一些“神奇”的工具感兴趣（那些追捧 .Net 的文章很少能达到 robbin 文章的深度）。一些软件企业的管理者，更是不读书、不学习（那是我手下人的事情）、刚愎自用、好大喜功（昨天我卖机器很成功，今天做软件肯定也没有问题！？）。从企业高层到开发人员对于思考深层问题的漠视直接导致管理水平的低下，产品质量的低劣。以前我们嘲笑西方人只注重工具和各种奇技淫巧，却不知道人家早就不是这个层次了，而我们显然还停留在只注重工具和奇技淫巧的层次上。

软件开发可以说是因人成事，没有合适的人，你就别指望能做和别人相同的事情（没有金刚钻，就别揽瓷器活。假设给你 100 个开发人员，一年时间，你有把握开发出 JBoss 一类产品吗？我是说与 JBoss 一样好用，你不仅开发出来，还要能卖得出去，别人喜欢用）。张三程序员即使水平很差，如果参与过大量开发，一旦跑掉了，他的工作也不是随便找来一个李四程序员短期内就可以胜任的，所以所谓即插即用的“软件蓝领”纯粹是一个谎言。尊重人才，人尽其用是软件企业首先要解决的问题，也是关系到企业生死存亡的大事。

软件开发中的审美疲劳

http://dev.csdn.net/article/28/28962.shtm

surstar [原作]

软件的开发的过程是一个反复的过程，对自己的界面时时刻刻的面对，一种很简单而豪无意思的说法是：换一个角度来看的产品，也许有一些人可以做到，但是他们真的做到吗，这和一个人美术细胞有关，一个修养有关，我认为要做到：吹毛求疵，在《人月神话》里，作者这样说到，程序员是一个天生就有追求完美的天性，这也是在锻炼过程中生成的！（大意），我认为 对自己的产品吹毛求疵是可以做到的，现在你打开一个你自己软件，观看自己的界面：

大小是不是合适，是不是可以再小 1 像素或大 1 像素，

有没有 BOTTON 是不大小不合适，位置怎么样，色彩怎么样

有没有功能上的不完整！

我个人认为审美疲劳不只是软件界面中有这样情况，在软件的代码和功能分析上也有如此的效应，总之，审美疲劳也许是吓唬人，只要我们作到吹毛求疵就可以作到了！

我这几天很烦！(产品概念完整性)

http://www.blogbus.com/blogbus/blog/diary.php?diaryid=226825

2004-06-18

我这几天很烦！

一是因为现在做的项目处于测试阶段，由于一些原因，导致现在发现了很多关于模块交互方面的问题。现在将这些模块“组装”成一个像样的系统，这些问题必须解决，而且目前只能自己与其它开发人员商量解决！

这个项目在设计的时候有七个人参与，应该是开发部所有的开发人员包括在内。当时先是讨论整个框架的设计，然后将各个模块的设计分下来，一人完成一两个，再接着开会讨论，讨论来讨论去，讨论的差不多了，也就是每个人将自己设计的模块接口写成电子表格（模块内部的设计有些有文档，有些没文档，因为每个模块的设计

人员基本上就是这个模块的编码人员，所有其它人不会太多关心），然后每个人就开始实现自己的模块。后来因为人手不够，又招了几个写代码的人员，分给各个模块的负责人员协助编码。事实上在真正进入编码阶段时模块的负责人只剩下三个，其它的人呢？两个离职，一个是公司领导，不参与实际编码，还有一个因为考博，没时间亲自参与开发。而剩下的三个人中有一个人是这个开发小组的组长，协调开发小组的开发工作，当然也参与实际的开发。

可以说在整个开发过程中没有一个真正的产品负责人（或者称之为总设计师），用于负责产品的整体结构设计、各模块的交互关系、功能设计和实现方案的取舍。他必须十分了解整个产品及设计方案，他可以不写代码，因为维护整个产品设计方案的工作贯穿于整个开发过程，工作量是很巨大的。他也就是维护产品概念完整性的那个人！

其实存在的问题还有很多！近期我会抽时间将它整理下来，寻找问题的根源到底在哪，避免再犯同样的错误！

还有我去年买的《人月神话》，当时没看懂。前段时间我拿出来，终于将它完整的看完了，而且感触颇多。

为什么《人月神话》在 20 年后还能再出新版，而且又一次给于业界巨大的轰动，就是因为我们没有吸取前人在 20 年前总结的经验教训，还在一次次的重犯同样的错误！

……

这个项目是一个二次开发包。完了之后紧接着有很多项目都要依赖这个。二次开发包做成这样，不难预测以后依赖此开发包的项目又将使人进入新的“焦油坑”。我现在也不知道什么时候开发第二个版本，至少今年不太可能。

2004 年 6 月 28 日

今天看来又将是一个不眠之夜了，连续鏖战已经有一段时间了，最近这段时间家事国事天下事，扑面而来，真些应接不暇了。

回到加班这件事，记不得在哪本书上曾看到过所谓“应当给软件工程师合理的计划和工作时间，一周工作超过 40 个小时就视同犯罪”，从《人月神话》到《敏捷开发》，从最基本的软件工程方法论和思想到最细枝末节的计划安排技巧和开发技术，有哪些内容能够在提高开发效率、改善产品质量之余减少我们的加班时间、增进我们的身体健康呢？

书香气围绕身边的感觉

http://140.127.22.156/blog/genius/archives/cat_eae.html

April 27, 2004

人月神话...我 Blog 的名称...正如自己所期望的。这是我很想看的一本软体工程的书。而且名字也很优雅，这本书带给我的感觉也相当的不错，在今天我透过诚品的网路书店，买下了这本书。之後要开始阅读这一本，估计在 5 月 1 号开始，因为要等中山大学研究所考完，就拥有短暂自己自由的生活，读书是一种痛苦，读书同时也是一种休闲，真难定义自己呢..我想我还是会一直看书，这是我的喜好，逛书店、图书馆也是我的喜好，真想寻找和我相同的人。哈..我就是爱书香气围绕身边的感觉。

概念完整性

http://www.mypm.net/bbs/article.asp?ntypeid=4&titleid=551&page=2

看了《人月神话》，体会到一点：概念完整性非常重要，扩展开来说，pmbok 中项目管理体系已经是一个经过验证的管理概念，要做好项目必须要关注项目中的各方面特性，就算你不关注，它们事实也存在。所以我认为一个成熟的项目管理人员，必须在头脑中首先有这样一个完整的项目管理概念，但是在小公司中，由于条件的不充分，可以弱化那些优先级不太高的子概念（比如 hr）,但是这仅仅是弱化，不是不存在，在一定时候，你必须关注，它也许会作为最高优先级的项目管理方面出现，毕竟具体项目中的管理是活的。

所以我认为“概念完整性”，是非常重要的，不仅仅在业务方面，在管理方面也是如此。

泥潭

http://www.geocities.jp/baryonlee/weblog/2003_09_01_archive.html

上一个项目已经结束 1 个多月了，新的项目一直没有下来。最近的一个月，yan 一直在设计他的 workflow。workflow 的思想是去年他们做 open cube 的时候学到的。那个系统最终还是成为了他们的噩梦。一周只回家一次，月加班时间达到 300 个小时，持续半年以上。最后的结果还是放弃。完全是《人月神话》里描写的泥潭。一个近百人组成的 team 就像一个困兽掉进了沼泽地，经常半夜 2，3 点钟开会，还一个人不缺，一周也不回家一次的人

大有人在。开始还用 sourcesafe 管理，后来 sourcesafe 都不管用了，改用原始的笔记本，每一个要更新 source 的人都要先登记，在早上 6 点的时候，会看到大家排队，手里拿着软盘，等着更新 source 的场面。我虽然没有经历这些，但我可以想象得到。可笑的是，那个用来登记的笔记本有一天不翼而飞，哈哈哈哈

MVM 的软件开发经验谈

http://hedong.3322.org/newblog/archives/000041.html

July 02, 2004

mvm 的“75 条”，读来深以为然，特转载于此。

5. 你们的项目组用了你能买到最好的工具么？

应该用尽量好的工具来工作。比如，应该用 VS.NET 而不是 Notepad 来写 C#。用 Notepad 写程序多半只是一种炫耀。但也要考虑到经费，所以说是“你能买到最好的”。（编注：干将莫邪——人月神话）

6. 你们的程序员工作在安静的环境里么？

需要安静环境。这点极端重要，而且要保证每个人的空间大于一定面积。（编注：在空间上省钱——人件）

7. 你们的员工每个人都有一部电话么？

需要每人一部电话。而且电话最好是带留言功能的。当然，上这么一套带留言电话系统开销不小。不过至少每人一部电话要有，千万别搞得经常有人站起来喊：“某某某电话”。《人件》里面就强烈谴责这种做法。（编注：电话——人件）

10. 你们的项目组中所有的人都坐在一起么？

需要。我反对 Virtual Team，也反对 Dev 在美国、Test 在中国这种开发方式。能坐在一起就最好坐在一起，好处多得不得了。（编注：黑衣团队——人件）

13. 你们的开发人员从项目一开始就加班么？

不要这样。不要一开始就搞疲劳战。从项目一开始就加班，只能说明项目进度不合理。当然，一些对日软件外包必须天天加班，那属于剥削的范畴。（编注：维也纳在等着你）

22. 你们项目组有 Team Morale Activity 么？

每个月都要搞一次，吃饭、唱歌、Outing、打球、开卡丁车等等，一定要有。不要剩这些钱。（编注：黑衣团队－－人件）

23. 你们项目组有自己的 Logo 么？

要有自己的 Logo。至少应该有自己的 Codename。（编注：黑衣团队－－人件）

71. 你在招人面试时让他写一段程序么？

要的。我最喜欢让人做字符串和链表一类的题目。这种题目有很多循环、判断、指针、递归等，既不偏向过于考算法，也不偏向过于考特定的 API。（编注：雇用一个变戏法的人－－人件）

73. 你们的程序员都能专注于一件事情么？

要让程序员专注一件事。例如说，一个部门有两个项目和 10 个人，一种方法是让 10 个人同时参加两个项目，每个项目上每个人都花 50%时间；另一种方法是 5 个人去项目 A，5 个人去项目 B，每个人都 100%在某一个项目上。我一定选后面一种。这个道理很多人都懂，但很多领导实践起来就把属下当成可以任意拆分的资源了。（编注：朝九晚五无所为－－人件）

《人件集－人性化的软件开发》

http://think.blogdriver.com/diary/think/index.html

《人件集－人性化的软件开发》（The Peopleware Papers）这本书是 Larry L. Constantine 著， 是对《康斯坦丁人件集》（Constantine On Peopleware）的整理再版。

刚读了将近一半，总的感觉不如《人件》作者分析问题更透彻。但还是有很多收获，尽管有一些内容已经有些陈旧，但是并不影响阅读的价值。而且书由许多短文组成，很适合间断的阅读。

作者认为软件开发中，人的进化速度远远低于技术本身的进化速度，因此多一些对人本身包括团队的研究是很有意义的，这一点我十分赞同。其实不光软件开发如此，整个人类社会的进化中，人自身尤其是思想的进化是很缓慢的，最近走马观花看了两千年前先秦的一点东西，感觉其中对一些问题的论述已经非常成熟，丝毫不比今天的人逊色。《登徒子好色赋》中登徒子和宋玉在楚王面前的辩论简直就是古版的“刘罗锅与和珅”。上世纪 30 年代林语堂的《中国人》中描述的中国人，今天读前来丝毫不觉得过时和隔阂。

所以，在软件的短短几十年历史中，开发人员的个体和团队的一些情况，确实没有什么大的变化。

《手机》与《人件》

昨天，看了《手机》的 DVD，恰好这几天也看了《人件》中关于电话和工作环境的论述，感觉两者有很多的相似之处，我不知道

刘震云和冯小刚是否读过《人件》，感觉上几乎能够肯定他们没读过，因为《人件》毕竟是软件行业的书籍，目标读者群就很窄。

《手机》从社会的角度，描述了手机的出现，对个人隐私（不管是好的还是坏的）空间和时间的掠夺，描述了技术跟人性之间的关系，其实技术只是一种工具，在没有手机的时代，人们一样会撒谎，会受到干扰，只是远没有现在这样激化与明显,技术催化了人性恶的暴露，或者说为人性恶的发展提供了更好的途径。

《人件》更偏重于论述电话对人的干扰，对脑力生产力的破坏，生产率的降低。想一下自己的工作环境，果真如此，每天自己的思考要被许多电话打断，以前却没有意识到，这是一个普遍的现象。人们既是办公室噪音的受害者，也是噪音的制造者。《人件》提到人们在思考时，会进入一种“顺流状态”的愉悦状态，而进入这种状态需要一定的时间，如果被打断就只能重新开始，反复的打断会让人产生挫败感。《手机》提供了一个很好的场景：费老给几个人开会，思路从萝卜、狗熊等将要进入“顺流状态”，却一再被众人的手机打断，后来干脆不知道自己要说什么。我们的办公室里，不是也经常有类似的事情发生吗？

以后，我要推荐我们的同事，多用 mail，少用电话。

《人件》中还提到，发明电话时，将规则定为只要被叫方不拿起话筒，电话铃就会一直响下去，而不是给人选择，或者响几声后停掉。这反映了新技术的“霸道”，有很多类似的例子，一种新技术可能彻底改变某种规则，比如：在家里打电话的人比去现场站着排队的人更有优先权，身边很多例子可以证明。我感觉这种新技术的“霸道”，可能有几个原因：1 技术本身的缺陷导致客观上不得不如此，电话铃之所以一直响下去，可能是因为这是最简单、成本最低的实现方法。2 垄断或者技术拥有者商业利益的驱动。手机单向收费就是最好的例子，技术上不存在任何问题 ，还有不同运营商的互联互通。3 新技术的发明者，缺乏深层的人文思想。很多技术型人才往往对技术有着狂热的追求，而对于技术对社会的影响，考虑的相对较少。

总之，《手机》还算值得一看，《人件》更是一本好书。两者有相似，也是英雄所见略同吧。

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

(英) Martin Fowler 著

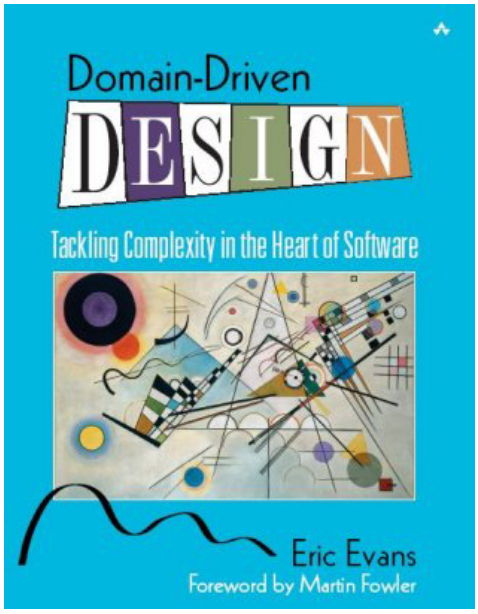
王怀民 周斌 译

UMLChina 审校

中译本已出版>>

《领域驱动设计》中译本(草稿)节选

本书中译本即将由清华大学出版社发行，译者陈大峰，张泽鑫，将由 UMLChina 审稿。经出版社允许刊登若干片断。



序-- Martin Fowler

很多原因都会造成软件开发的复杂性，然而其核心则是由于问题领域本身的复杂性。如果要在复杂的企业中实现自动化，那么你的软件便不可能避开这种复杂性——它所能够做的仅仅是对复杂的问题进行控制。

控制复杂问题的关键是建立一个好的领域模型，它越过问题域的表象介绍其底层的结构，提供给软件开发者所需要的方法。一个好的领域模型有非常重要的价值，但它的建立却不是一件容易的事情。很少有人能够出色地完成，并且建立的方法也很难传授。

Eric Evans 是为数不多能够建立出色的领域模型的人员之一，这是我在与他共事的时候发现的——这种时候往往是发现比你有经验的人的最佳时刻。我们的合作虽然短暂却非常有意思，从那时起我们开始保持联系，我也仔细地研读了这本书。

它是值得我们期待的。

这本书希望达到一个雄心勃勃的目的：描述并建立领域建模艺术的词汇表。使得我们在领域建模的时候能够参考其提供的框架，并且向读者教授这个较难学习的技能。我在这本书中得到了很多新的思想和观念，我相信任何掌握传统概念模型的人从这本书中一定能够得到很多新的思想。

Eric 还巩固了我们已经掌握的很多知识。首先，在进行领域建模的时候，不能够将概念与实现分离。一个高效的领域建模人员不应该只会使用记事本和计算器，还要能够编写 Java 程序。一部分原因是离开对于实现问题的考虑便无法建立一个有用的概念模型。然而概念与实现不可分割的主要原因是：领域模型

最重要的价值在于提供一种通用的语言将领域专家与技术人员联系在一起。

从本书中你还能够学到：领域模型并不是先建模然后接着实现的。和很多人一样，我开始拒绝接受“设计，然后建立”的定向思维。Eric 的经验告诉我们真正有效的领域模型是随着时间慢慢发展得来的，即使最有经验的建模人员也会发现总是在系统的初始版本实现后才会找到他们的最佳想法。

我认为，并且希望它会是一本非常有影响力的书。在向人们讲述如何使用这个有价值的工具的同时增加了这个难于掌握的领域的结构性与内聚力。领域模型会对管理软件开发起到重大的影响作用——不管软件开发使用何种语言或环境实现。

最后还有一点非常重要的想法。从本书中我看到 Eric 最值得我尊敬的一个方面，他敢于讨论还未取得成功的事情。大多数作者都想给读者留下一个无所不能的印象，而 Eric 却明白地告诉读者，他和我们中的大多数人一样，也曾有过成功和失望。重要的一点是他能够从两者中汲取养分——对我们来说更重要的是他会传播他所得到的经验。

Martin Fowler

2003 年 4 月

第一部分 让领域模型发挥作用

P2 图

这张十八世纪的中国地图表示整个世界。中间占据了绝大多数空间的部分是中国，周围寥寥几笔勾勒了其他的国家。这是适应于那个社会的世界模型，它有意地偏重于内部。这张地图表现的世界视图对于外国人是没有太大用处的，当然它也不能够适应于现代的中国。地图就是模型，每个模型都代表了我们所感兴趣的现实或观点的某些方面。模型是一种简化，它对现实进行阐述，只是抽象出对于解决手头问题有关的方面而忽略掉无关的细节问题。

每个软件程序都会与其使用者的活动或兴趣相关。用户使用程序的主要环境称为软件的领域（domain）。一些领域会涉及到实体世界：航线预定程序涉及到现实中登机的人。一些领域是无形的：记账程序的领域就是货币和金融。软件领域通常很少与计算机有关系，然而也有例外：源代码控制系统的领域就是软件开发本身。

要建立对于用户活动有价值的软件，开发团队必须瞄准与这些活动相关的知识主体。所要求的知识宽度可能是令人望而生畏的，信息的容量和复杂度也是不可避免的。而模型正是处理这种过载负担的工具。模型是一种有选择的简化和知识的有意识组织形式。一个合适的模型能够了解信息的含义并聚焦于问题本身。

领域模型并不是某一个特殊的框图，而是框图所要表达的思想。它并不仅仅是某个领域专家头脑中的知识，而是对相关知识进行严格的组织与选择性抽象。一个框图能够描绘并传达一个模型，同样，认真书写的代码以及英语句子都可以做得到。

领域建模并不是尽可能地制作一个逼真的模型。即使在一个可触及的真实世界事物的领域中，我们的模型也只是一个仿真的创造物。它也不是仅仅给出必要结果的软件机制的构建。它更像是电影制作，松散地表现具有特定目的的现实。即使是纪录片也不会展示未经修饰过的实际生活。就像一个电影制片人选择

经验中的方方面面并将它们用一种特殊的方式展现出来，告诉大家一个故事或一个论点，领域建模工作人员也要为它的实用性选择一个特殊的模型。

领域驱动设计模型的实用性

在领域驱动设计中，模型的选择取决于三个基本的用途:

（1）模型与设计核心的相互塑型。正是模型与实现之间密切的联系使得模型与现实相关并且保证对于模型的讨论分析能够应用于最后的产品——可运行的程序。这种模型与实现之间的绑定对于软件的维护和继续开发也很有帮助，因为可以根据对于模型的理解来解释代码。(参见第 3 章)

（2）模型是所有团队成员所使用语言的核心。由于模型与实现是相互绑定的，开发者们可以使用这种语言来讨论程序。他们能够在没有翻译的条件下与领域专家交流。又由于这种语言是基于模型的，我们的自然语言能力也能够用来细化模型本身。(参见第 2 章)

（3）模型用来提炼知识。模型是团队在组织领域知识和辨别最感兴趣的原理时一致同意的方式。在我们选择术语、分解概念并将它们相互联系起来时，模型能够反映出我们是怎样考虑领域问题的。开发人员与领域专家将信息放置于模型这种形式中，这样，公用的语言可以使他们的合作更加高效。模型与实现之间的绑定使得在反馈到建模过程时，软件前期版本的一些经验也同样可以适用。(参见第 1 章)

下面的三章将着手考察这些影响的意义和价值，以及它们相互关联的方式。通过这些方式使用模型能够给具有丰富功能的软件开发过程提供很大帮助，否则可能会需要巨大的投资。

软件的核心

软件的核心是它为用户解决领域相关问题的能力。其他的一些特征，尽管它们也许是必需的，但也是用来支持这个核心目的的。当领域非常复杂的时候，这个任务将非常艰巨，需要有才能和技术的人们共同努力。开发人员需要进入领域之中补充业务知识，他们必须磨练建模技巧来掌握领域设计。

然而，这并不是大多数软件工程优先考虑的方面。大多数开发人员都不大愿意学习他们所处理的特定领域的知识，更少有人愿意承诺去提高他们的建模技巧。技术人员则喜欢能够锻炼他们技术能力的可计量问题。领域相关的工作非常繁杂，并且需要很多难懂的新知识，而这些并不是计算机科研工作者能力范围内的。

技术人员应该在精心描述的框架下工作，用技术解决领域问题。学习建模和领域的任务留给其他的人。必须首先处理软件核心的复杂性问题，否则可能会偏离初衷。

在一次电视谈话节目的采访中，喜剧演员 John Cleese 讲述了在电影 Monty Python and the Holy Grail 中发生的一件事。他们反复拍摄一个特殊的场景，但是不知为什么总觉得不是很有趣。最后，他休息了一会，与共事的喜剧演员 Michael Palin 商量并提出了一个细微的改变。他们又进行了一次拍摄，最后达到了预期的效果并顺利收工。

第二天早上，Cleese 正在观看影片剪辑员对前一天工作的初步剪接，看到他们昨天一直研究的那个场景时，Cleese 发现它很没有趣味，因为使用的是早些的一个镜头。

他问影片剪辑员为什么没有使用指定的最后一个镜头。剪辑员回答说:“不能使用它，拍摄时有人走动。”Cleese 一遍又一遍地观看这个场景，始终没有发现有什么地方不对。最后，剪辑员暂停了影片，指出图片的边界处能够看到一个外套的袖子。

影片剪辑员注意的是他自己专业方面运作时的精密性。他所关心的是其他影片剪辑员看到电影时会从技术完美的角度来评价他的工作。在这个过程中，场景的核心问题丢掉了（*The Late Late Show with Craig*

Kilborn，CBS，2001 年 9 月）。

幸运的是，这个有趣的场景被通晓喜剧的导演还原了。同样的道理，当模型开发偏离了正确走向时，团队中理解领域中心问题的领导者能够使得它退回原处。

本书将说明领域开发能够培养复杂的设计技巧。大多数软件领域的杂乱性实际上是一种有趣的技术挑战。事实上，在许多科学学科中，当研究者着手处理现实世界的繁杂时，“复杂性”是最令人兴奋的问题。当面对一个从未被形式化过的复杂领域时，一个软件开发者也会有同样的期望。创建一个清晰易懂能够简洁地解决其复杂性的模型是一件令人兴奋的事情。

开发者可以使用系统化的方式来寻找并生产有效的模型，对于杂乱的软件应用程序，设计方法能够使其有序发展。这些技巧的培养可以使得一个开发者更有价值，即使在一个最初并不熟悉的领域。

第1章　关键性知识

几年前，我曾经着手设计一个用于设计印制电路板（printed-circuit board，PCB）的专用软件工具。我并不了解任何关于电子硬件方面的知识。我拜访了一些 PCB 设计人员，他们在三分钟内就让我头脑转向一片糊涂了。那么我应该了解多少才能够开始编写这个软件呢？我当然不想在交付底限之前变成一个电子工程师。

我们试着让 PCB 设计人员精确地告诉我们软件所需要完成的工作，然而这么做并不是一个好主意。他们是非常优秀的电路设计人员，但是他们的软件概念通常涉及到阅读一个 ASCII 文件，将其分类并加入注释，然后完成一个报告。这样做显然不能够取得他们所希望的生产率的飞跃。

前几次的会议让人气馁，但是在他们要求的报告中有一些闪光点。他们总是谈及“nets”和其中的各种细节问题。在这个领域中，net 是可以连接 PCB 中任意数目原件并将电子信号传送到它所连接的任何原件的金属导线。我们得到了领域模型的第一个元素。

图 1-1

在我们讨论他们希望软件所做工作的时候，我开始为他们绘制框图。我使用了一种非正式的对象交互图的变体来对情景进行初步描绘。

图 1-2

PCB 专家 1：元件不一定必须得是集成电路。

开发人员（我）：那么我们就只把它们称为“元件”如何？

专家 1：我们叫它们“元件实例”，因为可能有很多相同的元件。

专家 2：“net”方框看起来就像一个元件实例。

专家 1：他没有使用我们的符号。我猜想对于每样东西都是一个方框。

开发人员：很遗憾，是这样的。我想我应该更清楚地解释这个符号。

他们经常地修正我的一些工作，在这个过程中我开始学习相关的知识。我们解决了他们在术语和技术主张差异中的冲突和含糊不清的地方，他们也从中得到了学习。他们开始更加一致和精确地对事物进行解释，我们开始共同开发一个模型。

专家 1：说一个信号到达一个 ref-des 是不够的，我们还必须知道引脚。

开发人员：ref-des?

专家 2：就是一个元件实例。ref-des 是我们在使用一种特殊工具时所使用的名称。

专家 1：总之，一个 net 是将一个实例的特定引脚与另一个实例的引脚相连。

开发人员：我们可以说一个引脚只属于一个元件实例并且只与一个 net 相连么？

专家 1：是的。

专家 2：并且，每个 net 都有一个拓扑结构，用来决定 net 连接元素的方式布局。

开发人员：好的，这样画如何？

图 1-3

为了让我们的讨论更加集中，我们在一段时间内集中研究一个细节特征。“探针模拟”能够跟踪一个信号的传播，这样可以发现设计中可能涉及到的问题的各个方面。

开发人员:我已经明白了信号是如何被 net 传送到附属的引脚上的,但是它们怎样可以传送得更远呢?拓扑结构和它有关系么？

专家 2：不。是元件推动信号继续前进。

开发人员：我们肯定不能对一个集成电路的内部行为进行建模，那样太复杂了。

专家 2：我们不需要这么做。我们可以使用一种简化的方式，就表示成通过一个元件从一些引脚将信号推动到其他可能的地方。

开发人员：是这样么？

[经过反复的尝试和挫折，我们最后共同绘制了一个方案草图。]

图 1-4

开发人员：但是你们究竟需要从计算中得到哪些东西呢？

专家 2：我们需要寻找一些较长的信号延迟——就是说，任何超过三跳的信号路径。这是一个经验法则。如果路径过长，信号可能不会在时钟周期内到达。

开发人员：超过三跳……因此我们需要计算路径的长度。那么怎样算做一跳呢？

专家 2：信号每经过一个 net，称作一跳。

开发人员：那么我们可以一直传递跳数，net 会对它进行递增，就像这样。

图 1-5

开发人员：我惟一不清楚的部分是这种“推动”来自什么地方呢？我们需要在每个元件实例中存储那些数据么？

专家 2：一个元件所有实例的推动行为是一样的。

开发人员：那么元件的类型将决定推动的方式。它们与其他实例是一样的了？

图 1-6

专家 2：我也不能确定它的含义，但是我想象中每个元件存储的推动方式应该就是这个样子。

开发人员：对不起，我问得太过细节了，我只是想把它理解得更透彻一些……那么现在的问题是，**拓扑结构**与它有什么关联呢？

专家 1：探针模拟是不需要拓扑结构的。

开发人员：那么我现在可以不对它进行研究，对么？等我们接触到那些特征时再来讨论它。

这样就完成了模型的建立（实际中要有更多的纠缠和困惑）。在自由讨论中进行细化，在询问下进行解释。随着我对领域的理解和他们对于模型如何在问题解决中发挥作用的理解，模型一步步地发展。下面的类图表示了早期模型的基本形式。

图 1-7

经过两个这样的周期，我觉得我已经有了充分的了解，可以开始进行一些编码工作了。我编写了一个非常简单的原型，它是由一个自动测试框架所驱动，我避过了所有的底层结构。没有持续性也没有用户接口（User Interface，UI），这使得我能够专注于行为本身。再过几天我就可以示范一个简单的探针模拟。尽管它使用的是虚拟的数据并且向控制台书写的是原始的文本，它还是使用 Java 对象对路径长度进行了实际的计算。那些 Java 对象反映了领域专家和我自己共同使用的模型。

原型的具体化使得领域专家更加明了模型的含义以及它与软件的功能是如何相关的。从那时起，我们的模型讨论更加具有交互性，他们能够看到我如何将我新近获得的知识组合到模型中并写入软件。他们也从原型中评价自己的考虑，有了更加具体的反馈信息。

嵌入式系统中的原型要比我们在这里展示的更加复杂，它涉及到我们要解决的 PCB 相关问题领域的知识。它统一了许多描述中的同义词和微小的偏差。它排除了大量的工程人员理解但是并没有直接相关性的事实，例如元件的实际信号特征。像我这样的软件专业人员看到这个图后便可以在几分钟内了解到该软件是关于什么方面的。他或她能够有一个框架来组织新的信息并更快地进行学习，用来推测什么是重要的而什么不是，并用来更好地与 PCB 工程人员进行交流。

当工程技术人员描述他们所需要的新的特征的时候，我让他们与我共同进入场景来看对象是如何进行交互的。当模型对象不能够完成一个重要的场景时，我们集体讨论新的对象或对旧的对象进行改变，提取关键的知识。我们对模型进行精化，代码也随之改进。几个月过后 PCB 工程人员得到的工具比他们预期的更加丰富。