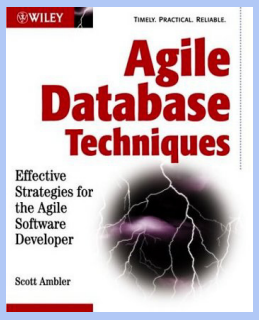
第 38 期（2004 年 6 月）



敏捷数据

X 非程序员 -Programmer
软 件 以 用 为 本

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

http://www.umlchina.com/

微软计划为 Visual Studio 增加测试建模等工具

[2004/6/15]

微软上月晚些时候勾勒了一个野心勃勃的计划，要为其集成开发环境提供企业建模、测试以及变更管理等功能。最早也要到 2005 年前半年才可能真正发布。

在本地举行的 Tech-Ed 会议上，微软发布了其 Visual Studio 2005 Team System。微软开发部高级产品主管 Prashant Sridharan 声称，这套生命周期管理工具将期待其 Visual Studio 的 Enterprise Architect 和 Enterprise Developer 版本。

Sridharan 介绍，VS 2005 TS 包括四个部分—团队架构师（Team Architect）、团队开发者（Team Developer）、团队测试（Team Test）和团队基础（Team Foundation）。该系统不仅服务于开发人员，同样服务于架构设计师、操作主管、软件测试人员以及项目经理，通过这个系统，所有这些人将协作起来。

Team Architect 将包括 Web services 设计工具 Whitehorse，它和一个类设计工具一样，是用于代码建模和生成的。同样，Architect 将帮助 IT 管理人员判断开发的应用是否可以成功地部署到公司已有的 IT 架构上。

Sridharan 介绍，和微软当前的建模工具 Visio 不一样，新的类设计工具将帮助代码和模型保持同步。Visual Studio 将继续支持 Visio 以及统一建模语言 UML。但是计划中为架构师设计的建模工具将不是基于 UML 的。“微软正在创建自己的建模标识，它将和 UML 不同”。

他强调，VS 2005 TS 将支持一些通用的 UML 图形，微软的合作伙伴也将基于微软的建模引擎和架构提供完整的 UML 实现。

Team Developer 将包括单元测试、代码覆盖以及静态代码分析等工具。它们将帮助开发人员在程序完成之前捕获错误，发现诸如缓冲区溢出等潜在的安全漏洞。

Team Test 许诺将包括加载－测试等功能，这将帮助测试人员模拟多个用户同时执行一个执行的场景。

Team Foundation 将提供源码控制等功能。将跟踪 bug 以及代码变更，并提供报告和分析等功能，帮助团队成员在整个项目生命周期中的工作管理。Visual Studio 2005 Team System 将继续支持微软当前的版本管理系统 Visual SourceSafe。Forrester 研究公司主任分析员 Uttam Narsu，认为，SourceSafe 适用于五个左右的开发团队，而 Team Foundation 的目标是为更大规模的团队提供变更管理功能。

呼吁第三方的参与

尽管微软计划为 Visual Studio 增加自己的建模、测试以及其它工具。微软同时也鼓励第三方公司的参与 VS 2005 TS 的工作。因为这个团队系统是微软将其开发技术扩展到企业计算环境的努力。因此其它开发商的参与将是非常关键的。Narsu 分析，“他们意思到要求全微软的环境是一个谬误，因此在积极地鼓励其它企业客户。” 在 Tech-Ed 会议上，微软发布了一个 VS 2005 TS 的 prebeta（beta 版之前）的预览版本。其 beta 版本承诺在 2004 年底之前推出，完整的软件将在 2005 年上半年发货。但是，Forrester 的 Narsu 认为，该公司并不能在 2005 年发布团队版本，因为这个版本的很多功能－尤其是建模方面的功能－都依赖于 Windows XP 的下一个版本，Longhorn。微软的 Sridharan 在一个声明中否认了 Narsu 的推测，并指出，Longhorn 和 VS 2005 TS 的开发是相互独立的。

同样是在 Tech-Ed 会议上，微软宣布了 Visual Studio 和.NET 框架的一个免费插件，Web Services 加强（Web Services Enhancements ，WSE）2.0 框架，从 msdn.microsoft.com/webservices/downloads/default.aspx 可以得到。微软 Web services 的产品主管 Rebecca Dias 介绍，它帮助开发人员更轻松地开发符合 WS-安全协议的 Web services，这是最近刚被 OASIS （the Organization for the Advancement of Structured Information Standards ）通过的一个协议。同样，微软也提供了 WSE BizTalk adapter 的一个技术预览，正式发布将在数月之后。

另外，微软还发布了其 Office Information Bridge Framework 的 beta 版，它将帮助开发人员应用 Web services 来连接 Office 应用与业务应用。

最后，针对顾客们的抱怨：产品还没有升级，产品支持计划已经到期了。微软将其所有的业务和开发软件的支持服务延长到 10 年。该政策将从 6 月 1 日起生效。

（自 sdtimes，UMLChina 袁峰 摘译，不得转载用于商业用途）

IBM 生命周期管理软件决胜 Visual Studio 2005

[2004/6/2]

下个月的全球开发者大会上，IBM 将展示其生命周期管理软件。以和微软最近发布的 Visual Studio 2005 团队系统一争胜负，后者是微软面向应用开发全生命周期的新版本工具。

微软在 TechEd 2004 上宣布其 Visual StudioTeam System，在此之前的一次采访中，Grady Booch，作为 IBM Rational 分部的首席科学家，认为 Rational 在全生命周期管理领域的漫长历史，以及诸如建模管理、团队开发支持等功能，保证了它在该领域的第一的地位。

在其全球开发者大会上，IBM 将会展示 Rational 技术集成到 IBM 中间件 portfolio 上的一些例子。2003 年 2 月 IBM 收购 Rational 至今，这是第一次 IBM 的技术会议和 Rational 软件开发者会议合并在一起。该大会将于 7 月 18 至 22 日期间在 Texas 召开，会议中心位于 Grapevine。

周二，IBM 宣布这个会议的同时，还发布了其开源的面向方面的 Java 扩展的新版本。

IBM 宣称，大会将包括超过 8 个主题的 200 多个会议。IBM 将在会上介绍其 IBM 软件开发平台的新的计划，以及在 IBM WebSphere Studio、Eclipse 方面新的策略，对 Linux 的支持，以及更新的 Rational 产品的设想。

会议同样将会给开发人员提供各主题领域的学习机会，包括业务驱动开发、MDA、UML2.0、面向服务架构（SOA）。并介绍软件开发在创建一个按需操作环境中所扮演的角色。

在对该会议的声明中，IBM Rational 的主管 Mike Devlin 说，“IBM 在全球开发社团中日益增大的影响力是将这两个广受欢迎的业界盛事结合在一起的关键。IBM 将向全球开发人员展示最新的技术、资源，以及程序”。

同时，IBM 也宣布了 AspectJ 的新版本，这是 IBM 牵头的项目，为 Java 语言提供 AspectJ 的面向方面的扩展。AspectJ1.2 加强了编译器以及工具方面的功能。其编译时间比 AspectJ 1.1.1 要快 1 倍。另外，还提供了一个“ajdoc”工具以为 AspectJ 程序生成类似 javadoc 的文档。

AspectJ 项目起源于 Palo Alto 研究中心公司 (PARC)。该公司隶属 Xerox 集团，由 Gregor Kiczales 领导，他现在是 British Columbia 大学的教授。为了推动 AspectJ 技术和社团的发展，PARC 在 2002 年 12 月将 AspectJ 转交给开源的 Eclipse 组织。

（自 eweek，UMLChina 袁峰 摘译，不得转载用于商业用途）

Borland Together 新版本推动.Net 开发

[2004/5/26]

“为大众提供买得起的建模工具”，这是 Borland 对其新的编程支持工具，Together Edition for Microsoft Visual Studio .NET v2.0，的要求。

该公司继续其对.Net 平台的全面支持，并在其基于 UML 的软件开发设计与建模工具第 2 版中支持 Visual Studio .Net。

“由于.NET 所提供的竞争优势，许多公司都在基于.NET 进行研发。但是，他们面临着日益复杂的应用开发环境”，Borland Together 业务部门的主管，Raaj Shinde 指出，“（在降低这些开发风险方面，）建模充当了一个关键的角色。Together for Visual Studio 版本是第一个.NET 技术方面的高级建模环境，它成功地给大众带来了买得起的建模能力，在此基础上，我们正在努力为大家提供更多”。

面向 C#和 Visual Basic .NET 的开发团队，2.0 版本中的新功能包括对 VB.NET 项目的双向工程支持，对 C#项目的审核信息以及加强的文档生成功能。

Together 产品的欧洲产品线主管 Paul Kuzan 强调两点，第一，和传统建模工具相比，其产品的低价位（135 欧元/license，含税），第二，Borland 的角色，是一个独立的第三方.Net 工具提供商。

“作为一个独立的厂商，我们不会固定在某个特定的语言和平台上”，他认为，这种灵活性有助于规避未来快速变化的开发环境的风险。Together 已经支持 Eclipse (Java)、Jbuiler 和 Cbuilder，现在，又本地化地支持了 Visual Studio.Net。

该工具使用的是 UML1.3 和 1.4，随着 UML2.0 的即将批准，Borland 承诺将在 2004 年底之前支持 UML2.0。

Together Edition for Microsoft Visual Studio .NET 2.0 要求首先安装 Visual Studio .NET 2003，并要求操作系统为 Windows 2000 (SP2 或更高)、XP Professional 或 Windows Server 2003。你可以从 Borland 网站获得关于该产品的更多信息。

（自 PC Pro，UMLChina 袁峰 摘译，不得转载用于商业用途）

Ivar Jacobson 博士最近忙什么？

[2004/6/6]

以下内容摘自 Ivar 的 postcard：

2004 年 6 月 3 日，我在新加坡举行了一次由 200 人参加的研讨会，发起了一次新的冒险——创办了 Ivar Jacobson 责任有限公司。我们帮助组织、实施统一软件开发过程的最佳实践（译者注：最佳实践指：迭代开发、需求管理、基于构件的构架、可视化建模、持续不断的验证软件质量和控制变更）。我们不仅会帮助实施现有的最佳实践，还会扩展更多的最佳实践。例如，我们将培训、指导面向方面的软件开发和主动软件开发（使用智能代理）。

软件最佳实践之一是关于方面。我第一次听到术语“面向方面的编程”是 1997 年，我立刻就发现这是一门有趣的技术，但那时，我没有时间仔细研究它。因为，我们正在开发 UML 的第一个版本，继续开发 RUP，并在主动软件过程方面作初始的研究工作。终于，我有时间看一看方面，已经是 2002 年的 9 月了，我下载了许多论文，连续多日学习。后来，我与 IBM 研究机构的 Harold Ossher 和波士顿东北大学的 Karl Lieberherr 取得联系，他们都是这个领域的带头人。在方面这个领域最知名的人物是 Gregor Kizcales，我试图与他联系，可是这家伙那时候太忙。

2002 年 11 月，我会见了 IBM 的研究人员，我们讨论了一天，帮助我理解他们的工作。他们的工作令我印象深刻，感到十分激动。离开了他们的办公室，我飞奔到了机场，踏上了回斯德哥尔摩的路。坐上了飞机，我要了一杯香槟酒，开始思考问题，突然一个念头向我袭来，这些事情以前似乎我也做过！我还写了一篇论文在 OOPSLA’86 上发表，那是 OOPSLA 的首次大会。

回到了斯德哥尔摩，我开始寻找那篇论文。我记得这篇论文讨论的主题是 1985 年我的博士论文的后续工作。当时，我对那篇论文的思想没有兴趣，就放弃了这个主题，我觉得，要想推动这些思想为时尚早，就把它忘了。我在使用对象和用例进行基于构件的开发方面取得了巨大的成功，因而没有新发明的空间了。但现在，我想找到那篇论文，在发布者的网站上，我找到了它。但下载这篇我自己的文章，竟然花了我 95 美元!!

这篇论文的标题是《大型可变更实时系统的语言支持》，此文中我引入了若干个概念："存在"表示基础系统，"扩展" 表示希望增加到基础上的独立功能。在不改变"存在" 的情况下调用"扩展"，我们需要这样的机制。因而，从"存在" 的视角上看，不需要增加任何修改，意味着你可以一个又一个的增加"扩展"，而无需改变"存在"。关键思想就是：在开发人员的视角上让"扩展"保持独立，系统就容易理解，容易维护和升级。

这种思想已经很接近面向方面的研究所要达到的目的了。但我还需要再确认一下，两个小时以后，我把这篇论文发送给了 Karl Lieberherr，他回信："Ivar，这是一篇面向方面的早期论文吗？"他还询问我是否还有更多的东西。于是，我扔下了所有的工作，第一个念头是我似乎没有别的东西了。但令我振奋的是，我的思想把我拉回到了那篇论文以前的时光，我的记忆提醒我：看看你的专利文件吧，有没有类似的工作？

那份专利文件是 1981 年我在爱立信工作时由别人起草的，我打电话到爱立信的专利部，询问他们是否还存有那份申请。一周以后，我得到了他们的寄来的瑞典语申请书拷贝。这份申请书是用典型的专利语言撰写的，实际上，我根本没有弄懂，这是一位专利工程师书写的。申请书的附件是两份爱立信的内部论文，详细描述了整个思想。这两篇论文是瑞典语写成的，我把它们交给一位专业的译者，翻译成了英语，可以在 www.ivarjacobson.com 上找到他们（查找已发表的论文和面向方面的软件开发）。

这个专利是关于我们称为序列变化器的，它工作在微程序指令级。设计的亮点是：程序包括一组指令，对于每个指令我增加了标志。如果标志打开了，表示这个指令点上需要执行"扩展"。则序列变化器从"扩展" 中读取指令，然后继续读取原来的下一个指令。分支由序列变化器处理，原始指令的开发人员无须对分支编码。

好吧，长话短说，Karl Lieberherr 和 Harold Ossher 看上了我的早期工作，Karl 还写了一份邮件给我，他对我的早期工作与现在的面向方面做出了对比：存在对方面，扩展点对加入点，等等。之后，我写了两篇关于方面和用例的论文（察看 www.jaczone.com/papers），因而，我被邀请到面向方面的国际会议上做了主题演讲。我很高兴我的早期工作受到了承认。现在，我正在与新加坡的同事 Pan-Wei Ng 合作写一本关于如何把用例和方面结合在一起的书，很快你就能看到它了。如果你买了它，我会很高兴；如果你阅读它，我会更高兴。你可要求我再做一些介绍，毕竟，我的工作是把软件开发实践改进到最佳状态。

（自 jaczone，陈星 摘译，不得转载用于商业用途）

Bertrand Meyer“面向对象软件构造”讲座即将举行

时间：2004 年 7 月 7 日（周三）下午 16:00-18:00

地点：此次讲座通过网络远程音频视频进行。具体操作方法见报名后的具体通知。

人数：100 人（人数超过以报名顺序为准排列）。

费用：免费

内容：

*面向对象软件构造

*按契约设计

演讲人：

Bertrand Meyer。对象技术大师，发明了 Eiffel 语言和按契约设计（Design by Contract）的思想，名著《面向对象软件构造》的作者，法国工程院院士。目前，他除了担任 Eiffel 环境和工具开发公司 ISE 的 CTO 之外，还是爱因斯坦的母校苏黎世联邦工学院计算机科学系教授，担任软件工程项目主席，同时还在澳大利亚 Monash 大学任教。除了面向对象技术外，早年他还参与了 Z 形式规约语言的设计。他现在的兴趣主要是软件工程的中心问题：可信构件（trusted component）。

预备学习：《面向对象软件构造》

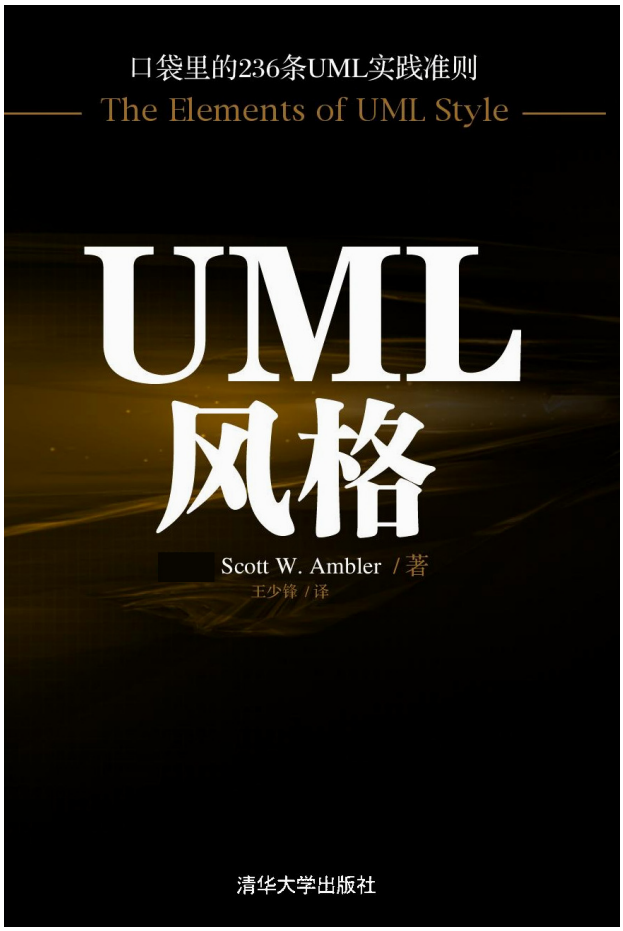
报名：

请在 http://www.umlchina.com/News/seminar040707.doc 下载报名表，寄到 seminar@umlchina.com 预定座位。

注意：

*请尽量不要用 163 信箱和 sina 信箱报名，这些信箱和 umlchina 信箱沟通不畅。

*每次讲座的报名是独立的。即使您在以前的讲座报过名，还是需要再报名。


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


mobile

《UML 风格》读者意见反馈

填写反馈信息

获赠 UMLChina 编著的《软件以用为本——UML 实作细节》

亲爱的读者：

感谢您阅读本书，也感谢您一直以来对清华版计算机图书的支持和爱护。为了今后为您提供更优秀的图书，请抽出宝贵的时间来填写下面的意见反馈表，以便于我们对本书做进一步的改进。同时，**只要您填写并提交下面的表格，即可免费获赠由 UMLChina 编著的《软件以用为本——UML 实作细节》，该手册专为《UML 风格》的读者定制，它图文并茂地描绘了应用 UML 的整个过程，点出了过程中的关键要点，非常适合与《UML 风格》配合阅读！**

请访问 UMLChina（http://www.umlchina.com）了解详细的活动细则。同时，如果您在使用本书的过程中遇到了问题，或者有好的建议，也请来信告诉我们。

地址：北京市海淀区双清路学研大厦 A 座 517（100084） 市场部收

电话：62770175-3506

电子邮件：jsjjc@tup.tsinghua.edu.cn

个人资料

姓名：_____ 年龄：_____ 所在单位：_____

文化程度：_____ 通信地址：_____

联系电话：_____ 电子信箱：_____

您使用本书是作为：☐选用教材 ☐参考读物

您对本书装帧设计的满意度：

☐很满意 ☐满意 ☐一般 ☐不满意 改进建议_____

您对本书印刷质量的满意度：

☐很满意 ☐满意 ☐一般 ☐不满意 改进建议_____

您对本书翻译质量的满意度：

☐很满意 ☐满意 ☐一般 ☐不满意 改进建议_____

您对本书的总体满意度：

从语言质量角度看 ☐很满意 ☐满意 ☐一般 ☐不满意

从科技含量角度看 ☐很满意 ☐满意 ☐一般 ☐不满意

您认为是哪些原因让您购买本书？（可附页）

_____

_____

您认为本书在哪些地方应进行修改？（可附页）

_____

_____

您希望本书在哪些方面进行改进？（可附页）

_____

_____

转化用例为设计

James Bielak 著，李胜利 译

讨论

许多软件开发团队信奉描述系统使用场景的用例表达系统需求。书写良好的用例的强烈优势是对话式的用户怎么做所以系统如何响应的事件序列，这样就能促进非技术和技术读者的理解。

对许多UML和统一过程的从事者，特别是新手来说，从一个类似于故事的用例集生成详细的、充实的设计模型可以证明是相当漫长的一段路程。从陈述的需求建模类和组件之间的交互，这些类和组件每一个都具有各自的属性，方法和责任，而且还要建立明确的架构，这是令人相当为难的。在网上以设计为主题的论坛上经常被问到的一个非常普遍的问题是“我怎样转换这些用例为设计？”

Robustness分析，也称用例分析，是一种从表现为用例模型的需求和表现为设计模型的系统规格说明间转换实践中得到的分析方法。Robustness分析的直接输出是分析模型，比较各种各样的分析类和描述单个用例实现的UML交互图形。

旧事重提，焕发生机

Robustness分析作为一个正式的精确模型出现，用来：

- 确定干扰形式的准确性和稳定性
- 理解复杂系统中的不确定性
- 辅助决策分析
- 分析组件交互

赋予Robustness分析模型背后的这些传统动机，多半是适当的，如果不是彻底的革新，软件产业借用这个术语来标注用于以下目的的技巧：

- 提升对于不确定的，干扰性的和模糊的需求的理解
- 把握复杂软件系统描述的不确定性

- 设计模型的辅助定义
- 分析软件组件的相互作用

RUP2002（Rational统一过程2002）最近重新命名Robustness分析为“用例分析”。这个改动可能非常合理，因为新名称对于Robustness分析相关内容描述得十分恰当。然而，术语“用例分析”对于不同的人们意味着许多事情。在Internet上的快速搜索显示无数的“用例分析”论文和以用例方法学特别是关于执行者，前置后置条件，用例命名和文本格式，<扩展>和<包含>的使用等等的多种形式出现。带有设计准备目标的最普通的术语“用例分析”描述在生成用例，而不是作为结果的用例的分析时使用的分析方法。为避免混淆，术语“Robustness分析”将贯穿本文。

在 Objectory 技术领先的 OO（面向对象，以下同）技术发展的早期，Robustness 分析与 Ivar Jacobson 的工作紧密相连。在 Jacobson 的 OOPSLA ’87 的研讨会论文“Object Oriented Development in an Industrial Environment”（产业环境中的面向对象技术发展，Jacobson,1987,and 2003）中，介绍了用例分析，类交互和实体。OO 分析技术的进一步发展出现在他的“*Object-Oriented Software Engineering: A Use-Case-Driven Approach*”一书中详细表述（Jacobson et al., 1992），书中包含了分析模型、用例和设计模型的特殊元素。

在早期的UML规范（1.3版，OMG,1999）中，OMG公布了“软件开发过程的UML概要（Profile）”，概要中给出了分析模型，分析类版型和分析类交互的指示规则。Kendall Scott和Doug Rosenberg (1999a, 1999b, 2001)在用例模型技术集中的期刊论文和教科书中进一步发展了Robustness分析技术。

理解包括用例或Robustness分析活动在内的分析模型是统一过程(Jacobson, Booch, Rumbaugh, 1999)和Rational统一过程或者称RUP产品（2002）的一个强焦点。这些描述了一种在正在开发的系统中学习和分析用例模型，生成分析类元素，详细阐述角色和分析类交互的方法。

所以虽然Robustness分析已出现十多年，仍有很多开发者没有意识到它的益处或者怎么执行，并且他们一直在问“我怎样从用例得到设计模型？”。本文将解释如何实践Robustness分析，连同在一个项目中得到的实际结果，包括：

- 完成的和正确的用例描述
- 开发前的高级架构视图
- 工作量的早期评估

分析如何开始

Robustness分析转换用例元素为分析模型-一个应当坚实的，稳定的和可维护的自由实现框架。在交互的UML协作图和序列图中，用例模型元素（角色，用例，关联和关系）被转化为边界，实体和控制。这些分析元素共同描述系统行为，概要协作和模型元素间交互，指定单个责任。因为Robustness分析集中于建立一个系统假定的理想映像，而不是工作准确实现，所以我们鼓励构想一个无限的内存和处理速度，不受常规限制或不考虑设计时我们通常必须考虑的硬件实现，来练习预想系统的技术（Jacobson, 1992）。

一个分析模型利用按系统地考察被提议系统的功能性和使用场景的用例优先次序排列的集合创建。关联的用例叙述通常是黑盒（显示非常少的要建立系统的内部工作）或白盒（描述多层次的系统内部细节）描述。当对较少关心系统细节而较多关注用户交互和想要达成的特定目标的预期用户描述用例行为时，黑盒描述通常是合适的。

根据分析团队的经验水平，用例可能优于Robustness分析，需要全面的细化。在能够设想和描述协作的分析元素集合之间，分析可能需要额外的系统细节。换句话说，有经验的分析员为了实现黑盒用例描述可能推动Robustness分析结果，并且增多提供给设计团队的信息量。

例子：

假设你正在分析一个涉及到在企业Intranet上的基于浏览器的雇员应用程序的用户访问的用例。因为雇员已经登录到内部的计算机系统，他们的用户标识自动用来访问这个浏览器应用程序。一个典型的黑盒用例可能是：

- “用户从选择列表中选择应用程序。”
- “系统在雇员的浏览器中显示应用程序。”

然而，在雇员的浏览器中，系统不仅仅负责调用一个应用。使用上面的用例步骤，我们没有一个非常清晰的系统图像，该系统必须在调用应用程序之前首先标识和认证用户。白盒描述应当是：

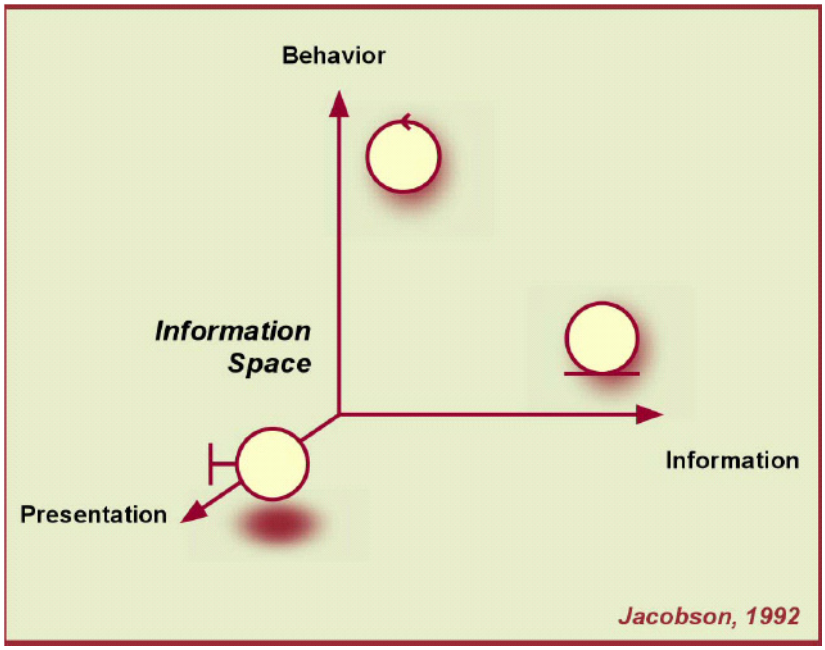
“系统企图通过发送登录用户的雇员概要信息给系统验证程序来验证用户合法性。验证程序校验雇员存在并且授权访问Mgr. III级应用程序。系统通过设置访问超时定时器响应，并且显示雇员浏览器中的应用程序。”

增强的详细级别提供关于潜在的分析对象，控制和责任的更多信息。第二种叙述显示对不在最初描述里的安全和认证的需要，但尽量避免描述系统如何设计或实现。根据分析员的经验，这种类型的描述可能为了进行分析而需要。另一方面，在设计准备中，增强的细节层次可能实际上是分析自身的结果。

在Robustness分析过程中的一个原始行为执行从用例行为中决定分析类。从上面的例子中，我们可以设想一个用户，一个雇员简要表，与系统验证相关的事物，一个访问定时器，授权级别等等。这些用例状态提供了我们需要生成分析类的信息。

信息空间

1992年，Jacobson et al.设想OO系统组件存在一个“信息空间”中，系统各部分位于由信息，行为，表现定义的三轴坐标系中。尽管不必要直接定位到一个轴上，设计良好的对象典型地与三轴中的一个对齐。例如，组件封装信息通常包含许多行为，它们可能响应关于它们状态的信息或者直接改变状态。这个特征阐明信息空间中的三个图表（UML版型），每一个都受控于某个轴或其他轴的吸引力。



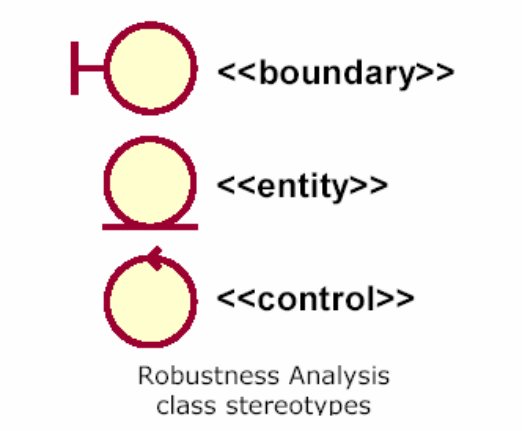
鉴赏这个三维的OO信息空间概念，考虑一下一个主要由函数和数据定义的程序系统。这个信息空间可能只有信息（或数据）和函数（或行为）两轴，并且系统组件的设计或描述可能被限制在二维平面内某处。利用Jacobson的信息空间，通过从描述系统行为和数据的二维平面提升表现信息，第三维（表现）拓宽了在OO分析模型中分离关系的潜力。

边界，实体和控制

Jacobson的信息空间中的三个分析类是：边界，实体和控制，对于软件开发过程来说通过UML简要表中的版型表达。你的第一个想法可能是“为什么限定我的分析过程到只有这三种类型？”。通过限制你的分析到这三种类版型，你被迫考虑定义的每个对象落入系统信息空间的哪个位置，因而承认对象确定的行为和要求确定的责任。进一步说，指派一个分析元素到三个类版型之一自动定义通信类型（因果关系，聚合关系和耦合关系），每一个元素陈列于系统的其他部分。

边界对象表现系统与所处环境通信和连接的系统元素。边界元素存在于系统的边上，而不在其内部。他们与系统外的角色交互，也同系统内的实体，控制和其他边界对象交互。

实体是包含域信息的类，是典型的长生命对象。实体是被动类，也就是说，它的对象不与自身发起交互。一个实体对象可能参与许多不同的用例实现并经常比任何单一交互长久。



控制对象通常与用例特定的行为关联。该类对象管理其他对象间的交互。控制经常有对一个用例的特殊行为，并且它们可能代表特殊子系统的行为。

在完成一个用例并达到其目标期间，分析员被限制使用这三种版型做全部描述，或完成系统全部行为。

这三种分析类版型在系统内部以一种适合他们通常责任的形式互相通信。类间通信意味着界面知识的一些层次（从没有任何知识到其他类界面的全部知识）。分析类可以以下述通信类型参与：

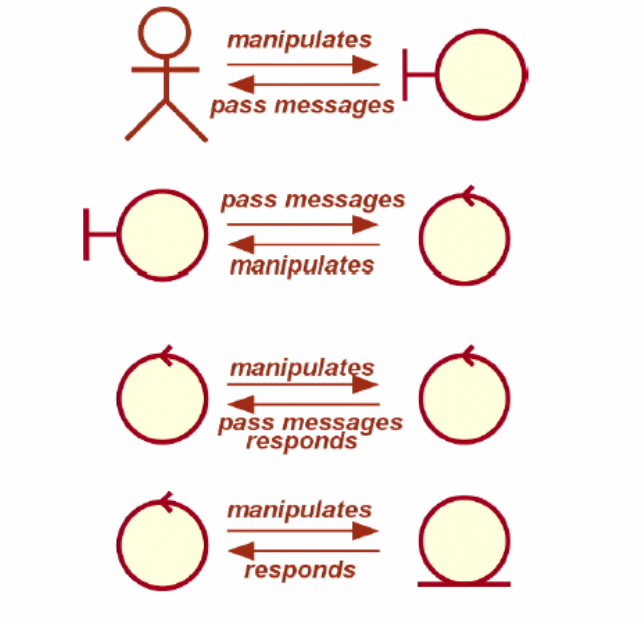
简单响应

传递消息或

处理消息

简单响应包含提供内部状态信息和受指示时执行行为的能力。实体典型地具有这种类型通信的能力，例如，一个开关能报告它的状态（关）或者这个开关可能直接开或关。响应对象不需要知道消息的任何来源，它仅仅报告或执行。

传递的消息需要一些通过消息的界面知识，尽管对通过消息对象来说界面可能是完全抽象的。例如，如果给你一刻钟告诉你打一个特定的电话号码，当IBM达到135美元时说“卖”这个词，你应当知道如何操作电话，但是你应当不知道与你通话的当事人的任何信息或者发布消息后的反应。边界对象通常希望传递消息。用户界面屏幕通常不对用户敲击的信息有深入的理解，但是它知道被定向后传递消息到系统“更智能”部分。该类型行为可能需要一些被传递消息（正如观察对象知道观察者一样）对象的抽象界面知识，而不需要更多。



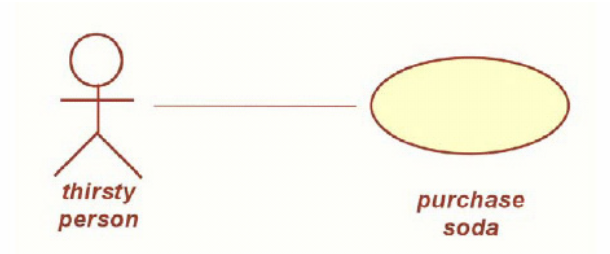
消息处理通常是在角色和控制类的域内。控制类是系统内“使事情发生”的对象。他们通常查询实体得到实体内部状态信息，查询边界对象得到与外部环境交互期间获得的信息，并都（与其它控制类一起）直接执行特定函数。为了完成这些任务，控制类需要对被处理对象的详细理解。

用例被所有相互协作完成每一个用例特定目标的边界，实体，控制的集合实现，这就是Robustness分析的目标：

- 发现分析类
- 在各个类之间分布系统行为和
- 描述类责任和协作

用例：购买苏打

我们第一个例子将阐述引导Robustness分析时的雇员机制。我们将分析一个普通体验-从一个售货机中购买苏打。在第一个例子中我们为了阐明和聚焦于概念探讨和发现的一些创造性方面，将故意避免和软件开发打交道。



目标：完成购买苏打的交易

主执行者：有点钱的口渴的人

事件流：

1. 口渴的人走进苏打售货机投币
2. 系统识别投入足够的币
3. 人做选择
4. 系统发放选择物品的一个单位
5. 用例结束。

发现分析类和分布式行为：

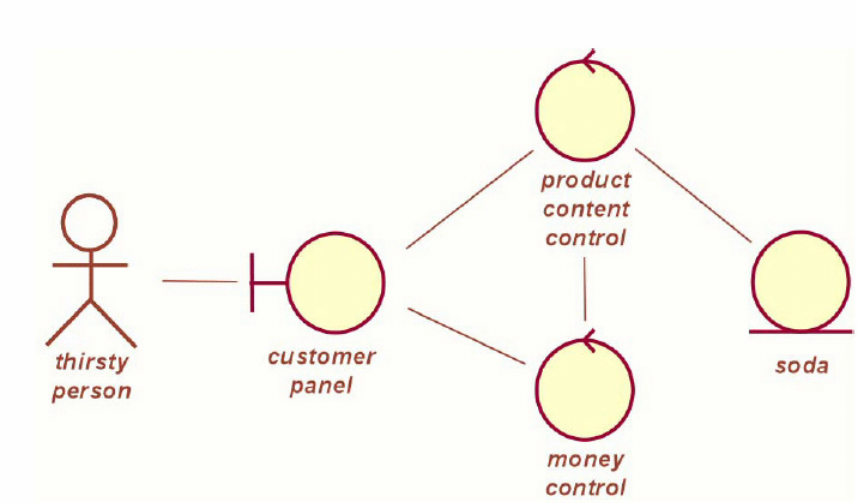
第一步，售货机大概有一些描述可售商品和价格显示排列。今天大部分售货机有一个投币口，另外一些可投入钞票。现代的机器将告诉你已投入多少钱。如果你幸运的话，在投币前（过去，你只有在投币后才能发现你的选择是空的，当机器里有货可卖时，出币口不会退钱给你）机器会告诉你要买的商品是否有货。我们称这个组件为“顾客面板”。

然后，我们将定义一个负责你投入多少钱，投入的钱是否足够买一个商品，如果有，多少零钱找回的组件。我们将称这个组件为“货币控制”。

为了在顾客面板上打开“做其它选择”灯做下一个选择，机器内的一些事物必须知道能分发的任何商品。我们可以设想当接到要求时这个设备具有一次发送一听苏打的能力。我们将称这个组件为“商品内容控制”。

最后，苏打听本身。

一个简单的实现购买苏打用例的协作图应该是这样的：



这时，我们假定我们已经“发现”所有系统组件（实际上，下面我们还会“发明”它们）。我们的下一步就是在每个类间分布系统行为。

描述类责任和协作：

顾客面板：这个边界类负责向用户提供信息，并且从用户向货币控制和产品内容控制传递消息。这时，我们将假定面板不知道货币的任何信息，如何计算，机器里有什么商品等等。这个面板有几个选择按钮，几个不同的投币口和几个表明卖出状态商品的指示灯。顾客面板提供的主要责任是：

- 从内部组件向顾客传递信息和
- 从顾客向同样的内部组件传递消息

货币控制：这个控制类有几个责任：

- 识别美元钞票和不同的硬币
- 维护最后一次交易后共投了多少钱的知识
- 知道每一种商品价格
- 当足够钱剩余时，有能力传递一个“出售”消息给商品内容控制
- 知道如何找零
- 知道把握给顾客提供多少硬币零钱
- 知道如何发布投入钱的数量给顾客面板

货币控制类大概不能识别可乐或百事可乐。它存在的基本目的是收集钱，当有足够的钱剩余时给出“go”信号，并且正确找零。

商品内容控制：这个类：

- 跟踪提供的每种商品的可售数量
- 必要时，告诉顾客面板在合适的位置点亮“空”指示灯
- 当足够的钱剩余时从货币控制通知获得消息
- 从顾客面板出售哪个商品状态获得消息

虽然商品内容控制部分不一定是这一模块中最亮的点，它仍执行一些重要的功能。机器主人应当感到自信，商品存货是安全的，当口渴的人投入足够资金时提供苏打。

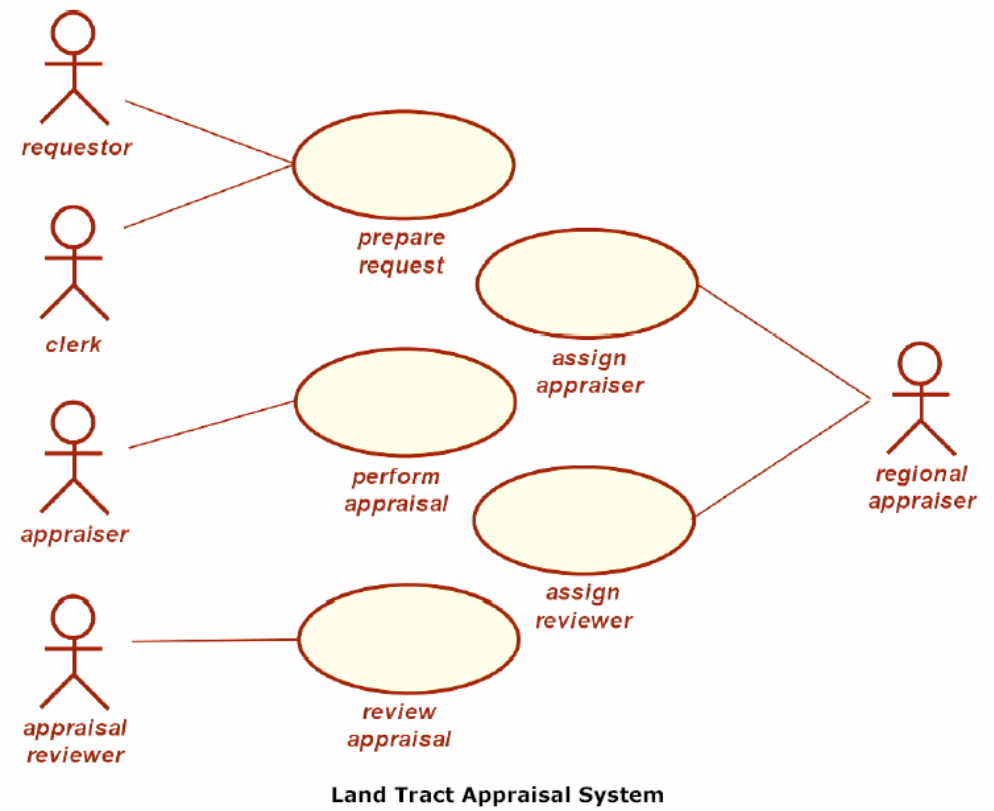
最后，我们设计苏打实体。当商品内容控制发送听或瓶时，它的责任被限制在遵循将其落到机器外的重力法则上。

在这点上，我们可以选择改善协作图或者构造一个序列图。给出迄今我们已经定义的内容，我们准备去测试能否我们的分析类来走通我们的用例。或者为了提供我们定义组件内部的行为期望的额外细节（白盒信息），我们能选择细化更全的用例步骤。

用例模型：国有土地评价系统

一个政府部门负责管理个人土地旁边的国有土地。农场主想用来放牧，能源公司想修建管线和传送塔，制片商想用来做拍片背景。政府需要掌握土地使用费用，但这个费用是公正的也具竞争力的。该部门雇用几个评价师测量提议使用土地申请指定的地域。这些评价师决定了一个提议使用类型的公平价格。然而，这是一个政府部门，有一个官僚政治程序。每个请求首先被审核，然后指派一个评估师，这个工作必须完成，并且评估结果也必须审核。我们的软件系统将帮助该部门更有效地引导他们的工作，并减少纳税人的整体费用。

这里提出的部分用例模型表现标识参与上述相关内容的一些角色和用例：



Land Tract Appraisal System

用例：指派评估师

目标：给职业评估人指派评估任务

主执行者：地区评估师

事件流：

1. 系统提出一个对地方评估师的明确要求列表

2. 地方评估师审核当前待处理的项目，并从列表中选择一个申请

3. 系统给出申请的详细内容

4. 地区评估师审核申请的详细内容并决定评估被批准

5. 地区评估师选择一个评估师指导工作，并给他指派当前申请

6. 系统传输所有权和责任给评估师，记录指派时间

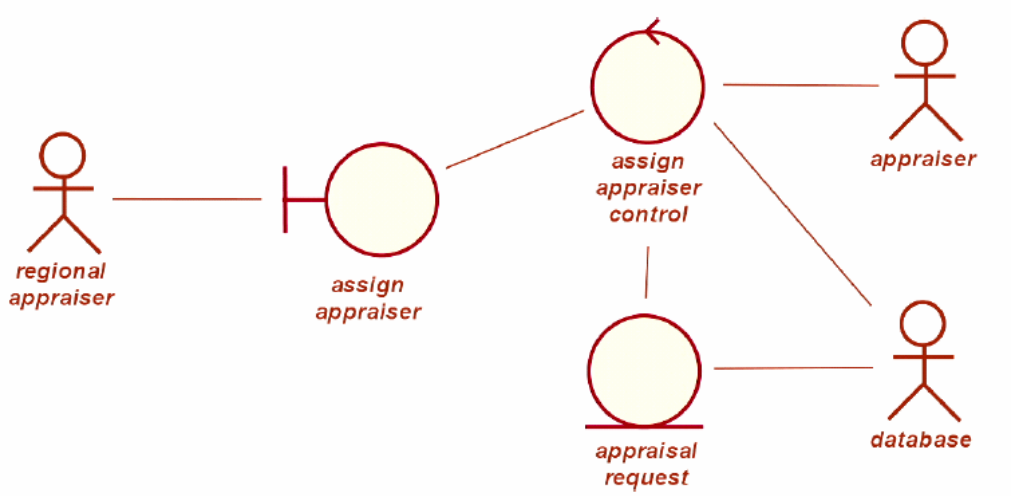
7. 用例结束

第一步：发现分析类

这一步中操作一般规则：

- 命名一个负责全部用例行为的控制类
- 标识每一个独立存在的用户界面窗口或对话框作为一个边界类
- 标识需要持久的独特条目的业务对象作为实体
- 标识额外的角色

在初始协作图中我们的第一个传递结果：

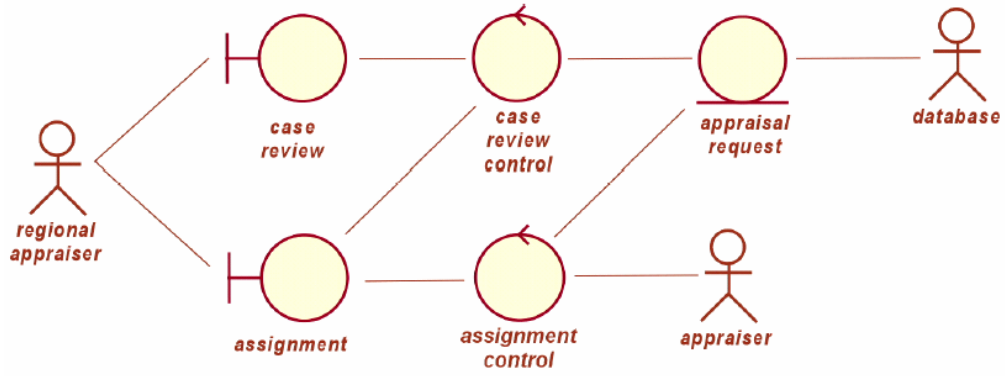


Discovering analysis classes

- 地区评估师是初始角色
- 指派评估师的边界类给这个用例功能提出GUI界面
- 指派评估师的控制类“知道如何做”
- 评估申请是这个用例的重要业务，必须存储于
- 数据库
- 还有，评估师

虽然这是一个好的开始，如果我们检查用例模型和与用例更紧密相关的步骤，我们能在行为分离和探索重用潜力上做得更好。一般的“指派评估控制”满足我们第一个单凭经验的方法，但是对对方评估师完成工作的更广泛的理解显示：1）她使用一些“列表”排序来评估当前待处理的案例；2）她以更仔细的方式单独评估单一特定案例；3）她在包含可用工人的列表池中选择一个工人安排工作。用例模型的外观显示本地评估师在这个用例中指派评估师，而在另外的用例中指派审查者。我们可能能够从指派功能中分离未处理案例的审查功能，分离相关的行为和业务规则为单独未处理案例的审核控制和指派控制机制。这类责任的高层抽象与分离可能呈现重用的机会。

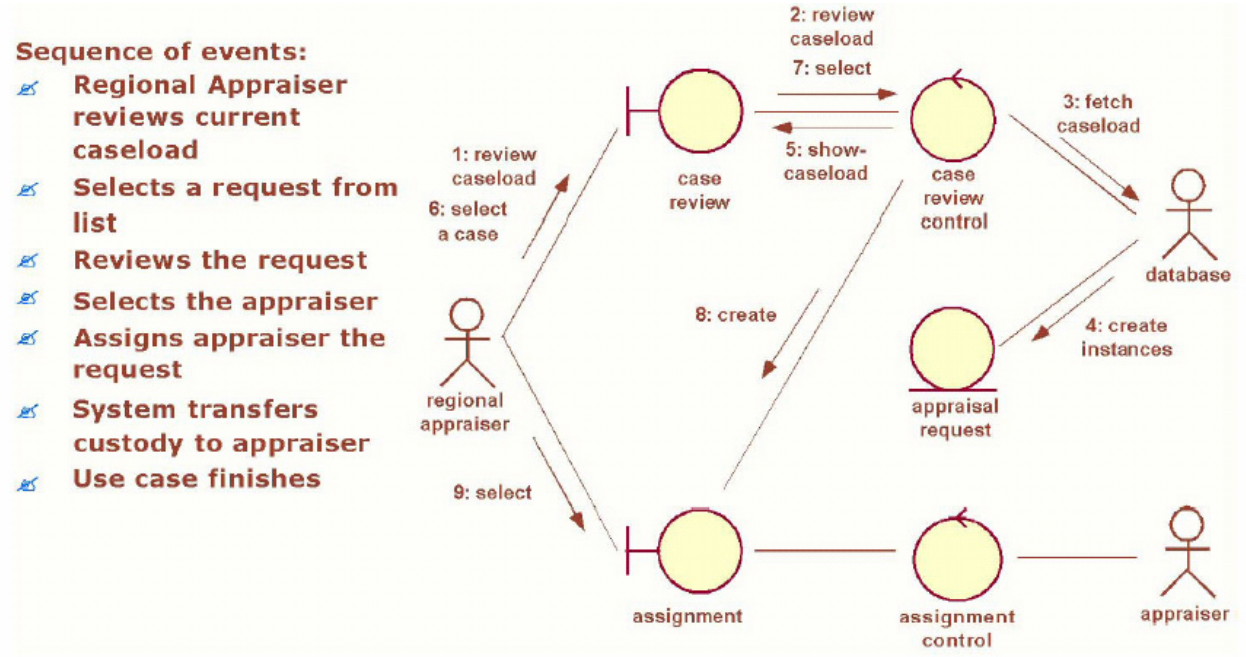
回到白板重画我们的分析类，我们也决定为每一个新的控制类关联单独的边界类（代表GUI对话框）：



Discovering analysis classes, second pass

第二步：分析类中的分布式系统行为

对我们发现所有的分析类并满足基本的单凭经验的方法感到自信，我们开始在每一个类间分布系统行为。我们决定改进阐述我们作为消息的类间交互的协作图。要做到这些，我们将需要用例中的事件流：



时间序列：

- 地区评估师评审当前未处理的案例
- 从申请列表中选择
- 评审申请
- 选择评估师

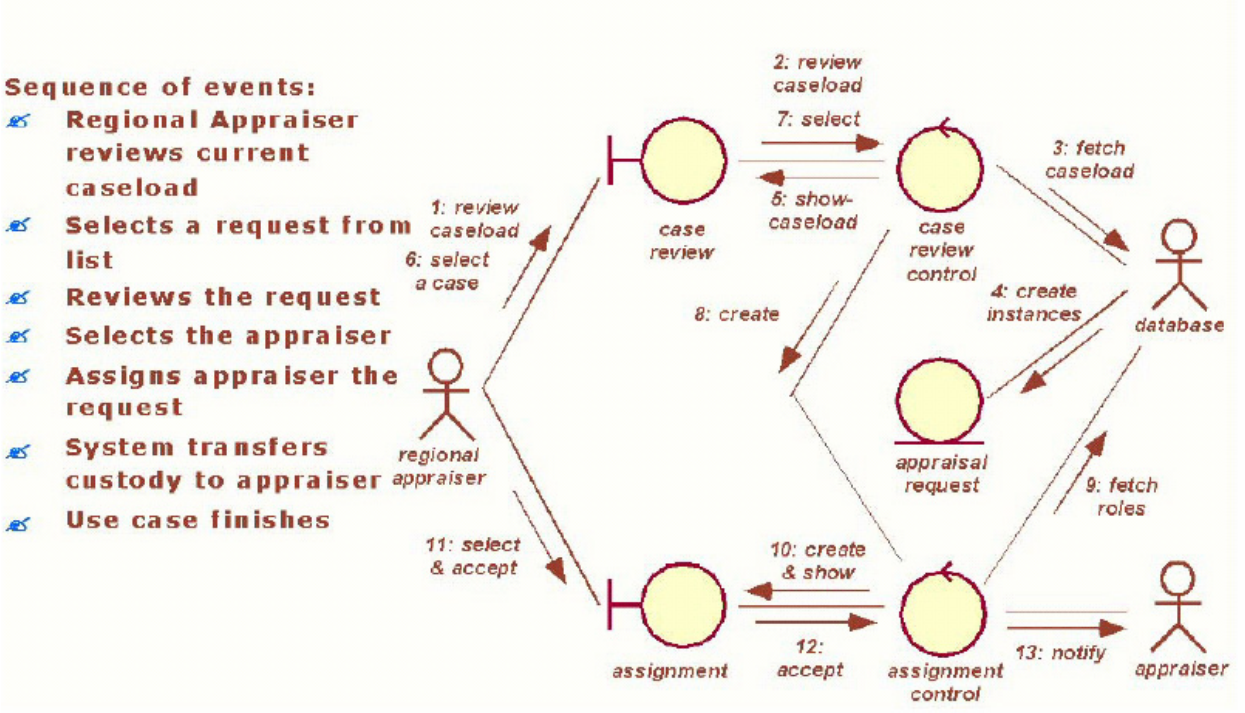
- 为评估师指派申请
- 系统传送保管内容给评估师
- 用例完成

迄今为止，任何事情都似乎井然有序，但是“地区评估师选择评估师引导工作”的用例步骤让我们暂停一会。为了为RA(注：地区评估师)选择人员去工作，系统必须具有现实可用工人缓冲池的能力。如果这实际上是案例，然后在任何给定的时间内系统必须具有“知道”工人是谁，他们单个工作角色和他们的可用性。这是一个带有雇员数据库、它们角色等等的日程应用程序的出现的开始。没有人曾经说过关于雇员日程的任何事...。这就是客户所想的吗？

Robustness分析开始获利。我们需要额外的覆盖雇员日程和雇员角色的用例是可能的。系统将如何知道谁做什么工作？系统将如何知道个人能力？有人退出或改变工作时发生什么，管理部门如何保存当前正确的雇员信息？

对我们客户显示的调用，实际上，他们确实希望雇员出现在屏幕上的选择列表里。工作只是变得更大。现在发现这一点，而不是去做错误的假设：对地区评估师来说通过敲入他们的email地址指派评估师是很合理的。

已经反射出我们将需要定义的新用例，我们认识到在我们的协作图中一些额外的细致处理是必要的。



时间序列：

- 地区评估师评审当前未处理的案例
- 从申请列表中选择
- 评审申请
- 选择评估师
- 为评估师指派申请
- 系统传送保管内容给评估师
- 用例完成

第三步：描述类责任和行为

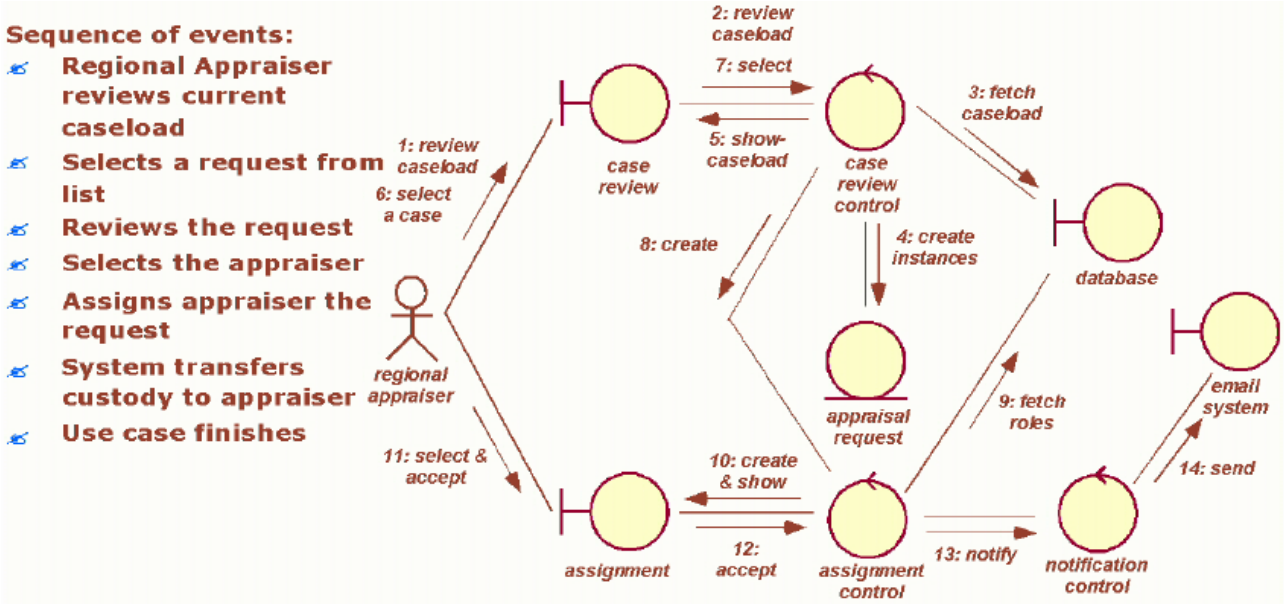
实现我们提出的案例评审控制类或许没有业务生成指派边界类，我们决定案例评审控制或许生成（或至少通报）指派控制类（负责生成他们自己的GUI元素）。为了生成一个地区评估师将选择的列表，指派控制类现在必须持续通信已确定评估角色中什么工人可用。唯一遗留的未解决问题是系统将如何通知已被指派的任务的被选择的评估师。

一个分析类的总结，在表格中它们的责任和行为能够容易地阐述：

分析类	责任和行为
案例评审<<边界>>	显示列表中当前待处理案例 允许单个案例选取 提出选择案例的细节 传递GUI控制消息给控制器
案例评审<<控制>>	询问用户待处理案例数据库 为了表达，为GUI提供案例 为了表达，为GUI提供案例细节 需要时生成指派控制器
评估请求<<实体>>	维护单个案例信息，摘要和细节
指派<<边界>>	在列表中提供可用于特殊角色的工人 允许选取单个工人 传递GUI控制消息给控制器
指派<<边界>>	为工人信息询问数据库 为表现向GUI提供工人信息 当被指派案例时通知工人

对指派控制的最终责任遗留一个问题：我们将如何准确地将工人的指派通知他们。这有几个可能的方式，我们可以在他们登录时，通过email或文章的形式通知他们。与客户的其他协商显示他们需要以email的方式通知工人，这为我们的理解提供了额外的信息。

我们已经获得了很深的理解到特定的用例中。一旦我们为一个额外的“通知控制”类建模，我们将花费时间并参与到我们以前没有涉及到的细节。我们检查分析类版型间发生的消息类型以保证通信的准确级别，并且我们用保留在我们系统内部的新的边界类替换一些外部的角色。我们的协作图，一旦完成，看起来是这样的：



Final collaboration diagram

时间序列：

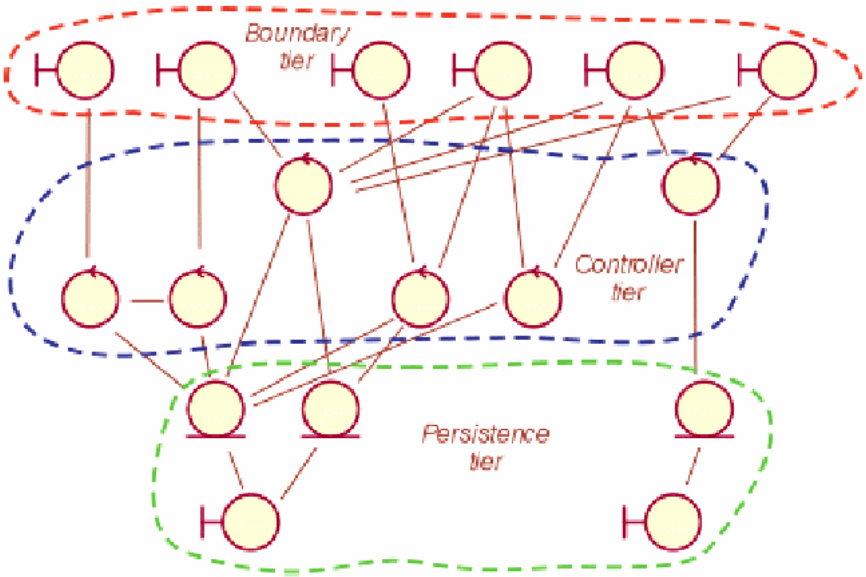
- 地区评估师复审当前未处理案例
- 从列表中选择申请
- 复审选择的申请
- 选择评估师
- 指派申请给评估师
- 系统传输保管内容给评估师
- 用例结束

架构形成

通过对一系列用例的分析和工作定义额外的分析类。许多这样的类是用例特定的，但有些代表整个企业业务对象的类在整个应用中被使用，有些是提供交叉应用服务的控制机制。

一旦我们的协作图集接近完成，当我们将所有定义的分析类放到一个大型的“参与类视图”的图中时，我们能够开始看到架构形成。它允许我们检查迄今所有边界，实体和控制类的范围。扫视一下我们能够估定他们如何相互影响，耦合程度由他们协作类型和某些类希望参与或在多个用例中执行的“热点”表示。

根据你的架构方针，你也可以根据已经建立好的架构来安排分析类。下一个描述显示由一个分析模型的Robustness分析转换的用例模型如何阐述表现（边界类）、业务逻辑（控制类）和持久（实体类）间的分离。使用这种方法分析的应用开始显现（或加强）一个潜在的架构，并为后续设计的努力做了一些工作。



Emerging architecture

工作量估算

Robustness分析帮助需求分析员和设计者生成一个工作量和成本的初步估算。理解多少预想的不同复杂性的对象和它们之间的相互影响以及整体架构允许你至少能计算组成应用的事物，评估开发难度，并且提供一个工作的初步的详细目录。通过分离和指派需求中全部的应用行为为离散的分析类，你就有了设置优先级和从设计选项中筛选的更多信息。

这不是真正的设计吗？

“你不正在做设计，而是分析吗？”-这个问题经常出现。记住练习Robustness分析，尽早开始：

- 检验完整的和正确的用例描述
- 明确表达一个开发前的高层框架视图
- 生成一个早期的工作量估算。

作为Robustness分析的结果，我们有一个我们用例的比较完善的，明确的和正确的看法。我们识别协作分析类，并且理解它们希望的责任和行为。我们自己断定在分析模型中的每一个分析类得感谢用例模型中的一些行为状态的存在。分析类的集合给我们一个架构视图，并且提供生成估算的信息。此外，我们除了需要GUI屏幕的标单，不指定实施细节和可能的选择列表。

设计努力将确保选择和使用合适的实现语言，适配分析类中定义的行为为单个设计类。紧记Robustness分析不暗示或影响任何从分析类到设计类的一对一通信。实际上，你的设计者随意分离和聚合分析类属性和行为到设计模型以最好地利用给定的实现技术。边界类可能变成界面类，或者GUI窗口和元素的集合。控制类可能看到他们的分布在不同对象中的行为，作为算法，业务规则，GUI控制器或者工厂。实体类可能进一步被细分或选择聚集到明智的实现对象。

要点是分析对象为需求和分析人员提供一个能够被指派行为、责任、协作甚至可能是数据属性的面向对象元素的丰富集合。这个活动使分析团队确保用例模型是能充分理解的，完善的，正确的和明确的。

Robustness 分析适合于所有人吗？

一个优秀的分析员应该反映，“那取决于”。第一个答案取决于你的特定目标的理解。首先和最重要的是，技术应该被用来探索和清晰理解你的用例模型，发现任何不正确的和丢失的信息，声称框架的早期视图，并且评估设计和实现中的多少工作量。因为“涉及到UML”和“就在那”，Robustness分析不应当被简单实践。进一步说，这项技术消耗掉一些时间，尽管不比其他形式的需求分析多，但是必须插入日程安排。明智的分析和设计团队做所需要做的一切来取得进展，正如反对仅仅做全部繁忙的工作一样。所以，在这种情况下你不应该沿着这条路开始吗？

2002年，当人们寻求学到新技能时，Alistair Cockburn评论了开始活动的行为的三个层次。他认识到这三个阶段为：跟踪，分离和熟练。

在“跟踪”阶段的人们需要有一个清楚的例子去学习，并且希望给定例子能生成正确结果。回想起你学习程序时可能在文本编辑器中敲入hello.c，然后视图编译它。在那个阶段，你就在“跟踪”。记起当你的程序不能编译时你的感受，并且当你发现你的打字错误时的感受减轻。对于在UML学习早期的人们来说，当你在跟踪学习时，学习不同UML图形通信，如何创建，使用什么符号。

下一个阶段“分离”，描述当你探讨你已经获得的理解和规则的边界和限制时的做法，例如，当你使用一种特定语言学习编程时。对于UML用户，在这个阶段，你意识到以确定的方式使用特定的图形去表达信息和如何使用确定的图形解决特殊的问题。你已经获得了使用UML沟通你的想法的信心，而不是被你可能使用的不同的方法迷惑。

最后一个阶段“熟练”，可以描述为大量理解UML长处和限制的工人，他们为了实现目标而发现延伸规则或利用其他表达方式的路径。当你必须对特定沟通需要感到满意时，你对“在盒外”步骤感到足够有信心表明你对你熟悉你的工具。

因为仍被问到“我怎样从用例得到设计？”的问题，我相信在“跟踪”阶段的人们坚持从Robustness分析中得到最大的好处。“分离”阶段的UML练习者将发现Robustness分析帮助他们得自经过深思熟虑的设计类，并促进对出现的与早期生成、更精确估算的成效相连的应用架构的理解。

在 UML 技术上熟练的工人也应能从 Robustness 分析中得到好处。在开始设计前花费时间保证你的需求是正确的和完整的表达只能够帮助预防移交不能满足客户需要的优秀软件的普遍错误。生成首次通过的行为分解为分析类总能帮助在设计上取得成效。这项技术对转化用例文本状态为拥有其他协作类的行为和知识的 UML 类提供逐步指导。

结束语

Robustness分析能为实践这项技术的人提供无数的益处。你学习这项技术的努力将在增强你理解企图描述和解决问题方面得到回报。回顾一下，Robustness分析技术为转化用例为设计提供无数的益处，包括：

- 不同的分析类版型和从用例文本中定义他们
- 分析类行为、交互和通信规则

- 用例步骤和系统行为合法性检查机制
- 检验需求中的正确和完整
- 发现额外的、以前没注意到的用例和需求
- 发现新角色
- 识别潜在的重用
- 从客户那里发现额外信息的需要
- 识别业务规则定义的需要
- 发现丢失的需求
- 提供估算的基础
- 阐明潜在的架构

任何帮助你提升理解客户需要的技术应有一次验证的机会。花些时间练习Robustness分析，最终掌握这项技术将提高你转化用例到分析模型、到设计的能力。

参考文献

[1]Alistair Cockburn, Agile Software Development, Pearson Educational, Inc., 2002.

[2]Ivar Jacobson, Grady Booch, James Rumbaugh, The Unified Software Development Process (Addison-Wesley Object Technology Series), Addison-Wesley, 1999.

[3]Ivar Jacobson, Magnus Christerson, Patrik Johsson, Gunnar Övergaard, Object-Oriented Software Engineering: A Use-Case-Driven Approach, Addison-Wesley, 1992 (Revised fourth printing, 1993).

[4]Ivar Jacobson, "Use Cases – Yesterday, Today, and Tomorrow," Rational Edge, March 2003, www.therationaledge.com.

[5]Ivar Jacobson, "Object Oriented Development in an Industrial Environment," OPSLA '87 Proceedings, October 4-8, 1987.

[6]Kendall Scott and Doug Rosenberg, "Successful Robustness Analysis," Software Development, Vol. 9 No. 3, 1999.

[7]Doug Rosenberg and Kendall Scott, Use Case Driven Object Modeling With UML: A Practical Approach (Addison Wesley Object Technology Series), Addison-Wesley, 1999.

[8]Doug Rosenberg and Kendall Scott, Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example, Addison-Wesley, 2001.

[9]OMG Unified Modeling Language Specification, Version 1.3. Object Management Group, Framingham, MA, 1999.

征稿

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

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

实用用例：事件建模使用例变得严密

David A. Ruble 著，王志航 译

讨论

RUP 在软件工程界得到了广泛的接受。象任何想赢得世界上软件工程师的心的方法学一样，RUP 也有很多的支持者和批评者。本系列的文章不是关于 RUP 的指南，也不公然反对 RUP。准确地说，这些文章的目的是反映我们在 OCG 对实际的一些项目实施 RUP 时切身遇到的问题，同时提供了对我们来说起到过作用的减轻负担的策略。这些策略已经被证实可以降低项目的风险。我们经验的主要内容集中在分析、设计和实现大型的业务系统上，尤其是使用关系数据库的业务系统。第一篇文章将特意讨论用例。第二篇文章将集中在数据分析上，第三篇集中在用户图形界面(Graphical User Interface)设计和规格说明上。

在过去的几年中，OCG 参加了许多客户（的开发）。这些客户都使用用例作为主要的分析工具。通过使用用例模型的经历，我们发现了两个主要的问题。第一个问题是决定什么组成用例很困难。第二个是如何更好地使用文字描述你所关注的用例的细节。

到底什么是用例？

考虑用例的原始定义：

用例是系统中的一系列操作，这些操作对于系统的单独执行者产生可测的价值。

---- Jacobson et al., 1995

就象经典的罗夏墨迹测试（Rorschach test）一样，一个人可以只盯着定义这一点并考虑其他的任何东西。用例的定义非常笼统因为它想覆盖几乎各种系统中的各种交互。当人们想定义用例的时候我们遇到的第一问题是关于什么是什么不是用例的指导标准很少。对于新的业务系统的开发来说，这个很笼统的定义是简单的，不充分的。

有一家客户，一批程序员被派去参加一个半天的关于用例的讨论会，在那里他们被训练在（大量的）需求文档中寻找用户说的动宾短语。”这些将是你们的用例，”年轻的新的指导员微笑着说。带着受启迪的激情，小组一起回到他们的房间并创建了（用户需求文档所包含的）一栏用例：

摧毁障碍物

和其它系统交互，和

对用户友好[1]

[1]第一个和最后一个用例是否是互斥的并不清楚。

在其他用户中，我们遇到了具有相似的含糊的动宾短语，包括一些我们喜欢的短语：

提出要求，和

得到答案

对任何项目时间（都）是精确的。在起跑线上从不确定的阴云下（开始）转动你的车轮会产生一个以任何（开发）速度都不安全的项目。所以什么是获得一组你的分析所依赖的好的、可靠的用例的解决方案呢？

答案是事件建模。

事件建模的简史

对于软件工程来说事件建模不是新生事物。McMenemin 和 Palmer,作为划分大型过程模型的方法，在1984年第一次提出了事件建模。当时的分析实践（方法）是数据流图（法）。McMenemin 和 Palmer 发现在大型项目中，跟踪事务很困难，因为事务在代表需求处理的圆弧和箭头区域之间流动。他们通过将模型分开来说明系统对于具体的业务事件的反映的方法使得表面的混乱变得有序。

这种技术在实践上和理论都行得通，并且很快被软件工程师采纳。事件建模也得到了早期面向对象使用者的注意。使用激励/响应范例聚焦于系统动作建模的观点和对世界采用面向对象（建模）的观点吻合得很好。在20世纪80年代，Meilir Page-Jones、Steven Weiss 和 Larry Constantine作为他们早期在面向对象技术方面研究的部分工作，进一步形式化了该学科。在20世纪90年代，OCG的顾问们开始在大型的客户/服务器模式的系统中使用事件建模（技术）。结果是令人振奋的。这种方法奏效了！不幸的是，关于这种技术的出版物直到1997年才出现[2](Ruble, 1997)。

同时，Ivar Jacobsen 关于用例的 1995（出版）的书籍（当时）正在软件工程界产生轰动，随之采用基于行为（behavior-based）的方法来组织需求的概念产生了。Jacobsen 的工作和事件建模很像，但是忽略了许多基础的形式化的东西。这些共同的努力在一开始就没有正式合并在一起真是一件惋惜的事情。事件模型的严密和试探法正是用例技术弥补一些我们已经在现实世界目睹的失误所需要的（东西）。

什么是事件？

计算机系统，如果放着不去管它，没有任何活力。只是对外界的刺激响应时它们才具有了生命。当它们具有生命时，它们按照程序以一种可以预见的和重复的方式工作—它们的行为是它们要完成的业务规则的反应。因此，业务系统的期望行为可以通过陈述系统应该怎样对外界的事件反应的方法来建模。

事件以主谓宾的格式陈述。某一执行者对某件事物做了某件事情，例如.，“客户下订单，”“销售经理否决贷款申请，”“市场部改变价格。”和从我们的顾客的启发式的用例指导员得到的动宾组合不一样，一个事件在被接受列到该项目的事件列表上之前必须通过 5 个测试：

1. 事件在具体的时间发生。
2. 事件是在环境中而不在系统内部发生。
3. 事件受环境而不是系统的控制。
4. 系统可以检测到事件的发生。
5. 系统和环境是相关的，意味着系统是设计来做环境中的某些事情。

现在我们已经获得了从行为的观点列出需求所需的可靠资料。事件模型在得出用例目录之外要进行大量的推测。

你应该走多“低”？

烦人的粒度问题不仅困扰着事件建模，也困扰着用例模型。一个叫做“使用系统”的用例层次明显太高了。同样的，“用户点击鼠标左键”的事件层次太低了。答案在于采用业务人员的观点。一个事件应该代表一个内聚的业务处理。这个业务处理完成了从事件触发者的角度可以看作是一个单元的工作。

事件应该也是技术透明的(technology-agnostic)。例如，“顾客需要发货状态”事件可以通过多种技术途径到达企业，顾客可以通过给消费服务代表打电话获得他们的发货状态。接着，消费服务代表从主机中查询这些信息。他们(也)可以通过 Internet 站点获得发货状态。他们也可能通过交互式语音应答(Interactive Voice Response IVR)系统的途径、通过短信或者发送无纸贸易(EDI)（的途径获得发货状态）。

不管技术怎样，事件的本质的业务规则保持不变。启发数据（the stimulus data）是相同的，处理过程也相同，并且响应数据(the response data)也相同。捕获业务规则的本质是事件建模的目标—并且将对话或者交互的设计推迟到技术被声明的时候。

事件和许多课本中的用例的定义是显著不同的。它是精确的。因为我们花费了我们大部分的时间来设计新的业务系统。因为我们认同以下观点：一个过早的陈述交互设计的用例很可能重复以前系统中的过失，并且错过对业务处理重新设计的机会。

通过坚持关键事件建模的原则，你可以避免交互设计的过早僵化—你最初的错误在记录需求上面，而不是在潜在的次优的（sub-optimal）交互设计上。相反，你将建立一种最后对业务规则更有价值的分析说明，因为它可以采用多种技术实现。

高级事件建模技术

事件建模的原则包含了一些鼓励发现（需求）的非常有用的技术。首先，分析员可以根据事件是“可预料的”还是“不可预料的”对事件进行分类。大部分事件是“不可预料的”，意味着业务（或者系统）从不知道什么时候一个事件的特定实例将要发生。在大部分行业，“顾客下订单”是一个典型的不可预料的事件。

在另一方面，可预料的事件，是一些前任事件（这些事件已经在系统中建立了预期的窗口，在这个系统中一个事件的特定的实例可以预料发生）的结果。例如，“仓库装运订单”（事件）要在以前的事件“顾客下达订单”已经通知了仓库去装运（订单）的基础上发生。可预料的事件吸引人因为它们的失败在期望的窗口发生。这些失败能产生在需求获取阶段由分析员经常忽略的各种各样的复杂的业务规则。

在我们“仓库装运订单”的例子中，如果我们假定这个事件在一段时间内发生，我们必须问我们自己（和用户）“在什么情况下我们宣布事件发生失败？” 因为它们的失败可以及时地在精确的时刻宣布发生，所以这些失败，或者（叫做）“非-事件”也需要通过了事件石蕊测试。（在这个例子中，如果仓库在 5 天内没能装运货物，客户服务部门应该听到警报声。）

另一特殊类型事件是时间触发的或者暂时的事件。暂时的事件总是可预料的事件因为它们是超过了由以前的事件在系统中建立的计划表的时间而顺延的结果[3]。

[3]你将注意到“time to ”短语与主谓宾的语法结构稍有不同。然而，时间的顺延，必定在环境的控制下，而不是在系统的控制下。

通过这个关于事件建模的简短介绍你可以发现，这种可以帮助分析员创建第一套合理的用例草稿的技术具有充足的严密的等级—这种技术以业务如何对现实世界的事件做出响应为基础。

支持好用例原则的下一步工作是查看用例书写的详细程度。

文档化用例

下面的部分包括我们在 OCG 开发时确保用例书写质量的最好的实践。

集中在做什么，而不是怎么做：用例应该聚焦在业务对于事件必须做什么响应，不需要描写详细的交互设计或者技术（问题）。交互设计是在系统的用户和系统本身之间复杂的对话塑造—并且是设计活动，而不是分析活动。

好的交互设计会考虑将来用户的技术水平以及采用的技术的优点（或者缺点）。使用诸如屏幕导航图(screen naviagion diagrams)和页-窗口（page & window）模型（进行交互设计）通常效率会更高，而不是长篇地将其写出。

包含交互细节和导航信息的用例在测试场景（中使用）比在分析文档（中使用）更合适。

区分数据和加工：我们在用例描述中遇到的一个问题是描述经常是输入数据、处理步骤和输出数据的混乱（堆砌）。事件建模规定从响应数据(response data)中抽取出处理过程的启发数据（stimulus data）。这种早期对数据的关注可以帮助你很快建立数据模型（也叫做领域类模型）－本系列第二篇论文的题目。

为了阐明这个问题，让我们看一个简单的取款机交易的例子，“账户持有者取款”。启发物是“取款需求”—由一系列数据元素组成。列出数据元素是分析人员的责任，并且应该确保这些数据元素在项目的数据模型中。因此，使用和数据模型一致的数据对于用例是重要的。

处理步骤以一种非常中立的方式叙述，避免了（对）当前取款机实现的对话交互过程（的描述），而取而代之的是集中在对取款的处理过程（的描述），不管交易是从取款机发起、从出纳员或者通过其他的电子方式（发起）。

在这个事件中从系统中得到的响应数据可以有以下两种结果之一——成功取款的确认信息或者拒绝取款。两种响应都和它们各自相应的数据一起列出。

事件：账户拥有者取款

启动：取款需求：

账号

密码，

账号类型，

交易类型，

交易数量。

处理：

检查账户余额

如果账户持有者的账户余额>=交易的数量

创建取款交易

计算现金

发送现金

创建取款确认信息

否则（otherwise），（余额<交易数量）

创建取款请求拒绝信息

结束（end if）

响应：（现金）和 取款确认：

-账号

-交易日期

-交易数量

-剩余余额

-交易编号(ID)

-地点编号(ID)

-现金

取款需求拒绝信息

-账号#

-拒绝日期

-交易数量

-拒绝原因

-地点编号(ID)

你将注意到处理规格说明的结构化性质。这导出了下面的我关于顺序、选择和迭代（的讨论）。

使用顺序、选择和迭代来描写过程：用例描写业务过程。因而，采用过程说明的三个基本结构是必需的。

（1）顺序：按照事件发生的顺序列出事件的步骤，

（2）选择：业务逻辑中的自然的事件分支应该在用例的主要流程中描写下来。

（3）迭代：经常，为了一组目标一个过程会反复地执行很多次。程序员管这个（结构）叫循环。在业务领域，这种结构会自然出现并且每次出现的时候应该特意指出。

使用合适的缩进帮助读者理解用例的结构。这种技术的批评者称缩进使用例看起来太象伪代码[4]（Kulak, 2000）。然而，我的经验告诉我合适的缩进带来的视觉上的清晰效果可以帮助人们更好的理解用例的陈述，就象合适的缩进可以帮助程序员理解其他程序员的代码一样。

避免标记数字的步骤。我倾向于不对我的处理步骤标号。如果你觉得必须要对步骤编号的话，一定要抵制住使用数字对步骤进行编号的诱惑。以后的插入或者删除步骤会搅乱你的卡片（上面的）索引，从而产生了大量的可怕的维护工作。同样的，臭名昭著的”go to ”语句被嘲笑为低劣的编程实践就是因为（使用"go to"语句带来的）对于程序的控制流程理解的困难性。和好的编程实践一样，解决办法是通过名字对处理做引用，例如，“检查顾客信誉”，而不是“跳转到 14.b”。

只向前引用—不向后引用：有一个狂热的用例技术支持组织已经发布了关于 if/then/else（选择）结构的责难。这个学派的学生被教导书写用例的时候好像（需求中）没有选择和意外发生。基本的流程，也可以叫做“快乐日子的场景”，首先被书写出来，接下来是所有的令人头疼的意外事件。如果幸福在那天离开了他们，这些意外事件会降临到这些执行者身上。

我努力的奋斗了很长的时间以试图使这种实践生效，并且得出了下面的结论：在选择或者分枝发生的时候通知读者会更有作用，而不是保留着它们不被处理直到最后的附录。

如果不是故意的从用例中根除 if/then/else 逻辑，会有一些奇怪的后果。”快乐日子的场景”对于不采用分枝逻辑。然而，当你阅读意外流程的时候，你会发现，如果情况 x 发生，你将到基本流程中的“第 4 步”寻找意外处理路径[5]。在意外步骤的末尾我们经常遇到 ”go to”语句来重组我们基本的流程。这是一种纯粹的混乱（的组织方式）。

当在系统设计中使用用例的时候，我们必须精确地将逻辑分支重新组装起来。我看到过许多用例是如此的零碎以至于所有的人（注：原文 所有的国王的马和人）都不能重新组装它们。

下面是一个使用数字编码步骤、没有逻辑分支和向后引用的用例的例子。它也包含了许多交互设计—和用例搅和在一起，使得它仅适用于一种具体的实现。（比较这个用例详细说明和事件建模风格的用例“账户持有者取款。”）

[5]注意if语句没有根除，他们只是被放到文档的末尾。

用例：12.取款

基本流程：

1. 账户持有者插入现金卡
2. 系统提示输入密码
3. 账户持有人输入密码
4. 系统验证密码
5. 系统提示选择账户类型(核算/存款)
6. 账户持有者选择账户类型
7. 系统提示选择交易类型

8．账户持有者选择“取款”

9．系统提示以 5 的倍数输入（取款）金额

10．账户持有者输入（取款）金额

11．系统对账户持有者的账号取款

12．系统计算并吐出现金

13．系统打印取款确认（单）

14．系统退回现金卡

15．用例结束

可变的路径：

12.1 密码不正确

12.1.1 在基本路径的第 4 步，密码不合法

12.1.2 系统显示“密码错误”

12.1.3 系统提示输入密码

12.1.4 密码 3 次错误后，系统没收卡，通知用户给银行打电话

12.1.5 用例结束

12.2 存款不足

12.2.1 在基本路径的第 11 步，账户余额比取款金额少

12.2.2 系统拒绝交易

12.2.3 在基本路径的第 14 步重新开始用例

这个例子很小，看起来差不多可以管理。然而，在一个真正的项目中，数字编码机制和向后引用会很快给生产率带来灾难。

还有几个其他的可以帮助你书写好用例的技巧。他们包括：

通过执行者的“角色名字”引用执行者（例如，不使用“用户”而使用“客户服务代表”）。

当主要的数据实体（也叫作域类）创建、读取、更新或删除(CRUD 功能)时应该详细（的介绍）。

指出可以在“包含用例”中重用的过程。这些仅仅是被多个用例引用的内部过程的详细描述。

改变“状态”值的事件应该在状态转移模型中标出。

不要企图使所有的情形都符合用例图。一些问题是重数据(date-rich)轻处理(process-light)型，包含很少的动作或者没有动作。选择手头适用于这种问题的最好的模型技术。

在好的面向对象设计中使用用例

我通过关于使用用例的一些介绍结束本篇论文。Bertrand Meyer，面向对象的创建人之一，写道：“用例方法类似功能方法，以过程（动作）为基础。这种方法和面向对象的分解相反，它集中在数据抽象上面；该方法具有在面向对象开发的幌子下回归到传统的功能设计形式的严重风险。”

Meyer 所说的是：为了实现向不同的设计/编程小组分配用例势必会造成灾难。相反，在用例中的每一个处理步骤不得不根据在面向对象的类模型中的最好归宿来评估。在业务系统中的用例详细的描述了业务怎样对于具体的业务事件做反应。那些事件对许多类起作用—并且反过来，同一个类会有许多看起来毫不相关的事件对它起作用。

一个单独的，内聚的面向对象的设计小组应该决定哪一个处理步骤应该分配给在数据密集、持久稳固的类上面的操作，哪一个处理步骤需要用来处理业务规则的新的“经理”类。因此，面向对象的设计小组必须管理过程对类的分配，以及在整个面向对象的系统架构中设计模式的应用，以获得面向对象技术提倡的重用和扩展的优点。

总结

作为许多软件开发组织选择的分析形式，用例已经（非常）出名。作为一个学科，用例模型和以前出现的处理和行为模型学科具有很多共同之处。实际上，可以通过应用事件建模的方法创建用例来消除伴随用例模型的模糊性。

事件建模使用技术-独立的启发-响应模型技术来书写本质的业务需求，推迟交互设计。通过分析事件是可预料的还是不可预料的，诸如可预料事件的失败的发生、定时事件等重要策略通常在项目的早期就显现了出来。

为了修改用例模板以适应事件建模，只需将启发数据、处理步骤和响应数据分到三个部分。以这种方式书写的好的用例将详述输入和输出数据元素。就像在数据模型中一样，它将通过相同的名字引用数据。

用例步骤应该使用顺序、选择和迭代—编程逻辑的三个基本结构，来描述业务规则。我建议不要使用数字对用例步骤编号，并且我坚持步骤序号不应该与行号紧密耦合。规格说明可以包含关于异常的向前引用、注释或者包含用例，但是不能有向后引用。

最后，用例本质上是处理过程的规格说明。因为许多用例可以依据同一个给定的业务目标，所以具有对用例的整体的认识对于面向对象系统的设计者很重要—因而应该避免建立分离设计小组的机制，否则会导致面向过程的系统，达不到面向对象的目标。

关于作者

David Ruble 是一位分析家、设计家、作家和教育家，信息建模、对象建模和 GUI（图形用户接口）领域公认的专家。他一直是许多紧急-任务客户/服务器信息系统、电子商务系统以及公众安全领域的主要分析者和设计者。作为一个教育家，在美国他已经给成百上千的学生教授了软件工程。他是畅销书籍客户/服务器与图形用户接口系统的实践分析和设计的作者。该书由 Prentice-Hall 出版。David 是奥林匹克咨询集团（Olympic Consulting Group）的负责人。

参考文献

Anderson, David. “Are Use Cases the Death of Good Design?” February 24, 2001, uidesign.net

Cockburn, Alistair. *Writing Effective Use Cases*, Addison-Wesley Pub Co; First Edition 2000

Kulak, Daryl, and Eamonn Guiney. *Use Cases: Requirements in Context*, Addison-Wesley Pub Co; 1st edition, 2000

McMenemin, S. M., and J. F. Palmer. *Essential Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

Meyer, Bertrand. *Object Oriented Software Construction 2nd Edition*, Englewood Cliffs, NJ: Prentice Hall PTR; 2nd edition, 2000

Page-Jones, Meilir. “Synthesis” Seminar Course Notes. Bellevue, WA: Wayland Systems, Inc., 1992 Ruble, David A. *Practical Analysis & Design for Client Server & GUI Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1997

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

复发责任分析模式

Lubor Sesera 著，Trio 译

讨论

摘要

这篇论文讨论关于在复杂系统中金融责任的重复建立问题。我们建议把“职责”（obligation）从 party（系统）中去耦，在一定的时间点上由职责生成责任（duties）。所生成的责任包含所有必需的数据以及分配给它的支付（payment），这样便于时刻核查责任和支付状态。本分析模式虽然是受保险业务的启发，但它也可以应用于其它的行业，如贷款、分期付款及国家社会保障系统。复发责任分析模式不只关心收入，也关心债务。

1 动机

保险公司和客户签订保险合同获取保费。该合同在一定期间内有效，而客户不必一次付清所有保费。保险公司需要时刻知道实际的保费总额和已经支付了多少。为使这项询问简单高效，查询保费的命令要能够在任何时候可执行（例如每天晚上）。图 1 是用 UML 描述的将查询保费命令从保险合同中去耦的类图示例。[Booch+ 1998]

2 上下文

当一个组织同另一个组织间有重要的金融责任时，这些责任由组织间签订的订单、合同、协议或者其它详细说明的法律文件产生（如一份已被接受的关于社会津贴的申请表）。本文使用抽象概念“职责”（obligation）代表所有这些文件。责任经常重复出现，其支付总额在不同的条件下会有少量的不同。每一个组织都可以和多个对方组织产生并承担多个职责。本软件系统可以分别安装在支付的组织和收取支付的组织。

3 问题

如何在任何特定的时间点上，从这样一个复杂的系统中更加方便、快捷、高效地找出责任和它们的支付？

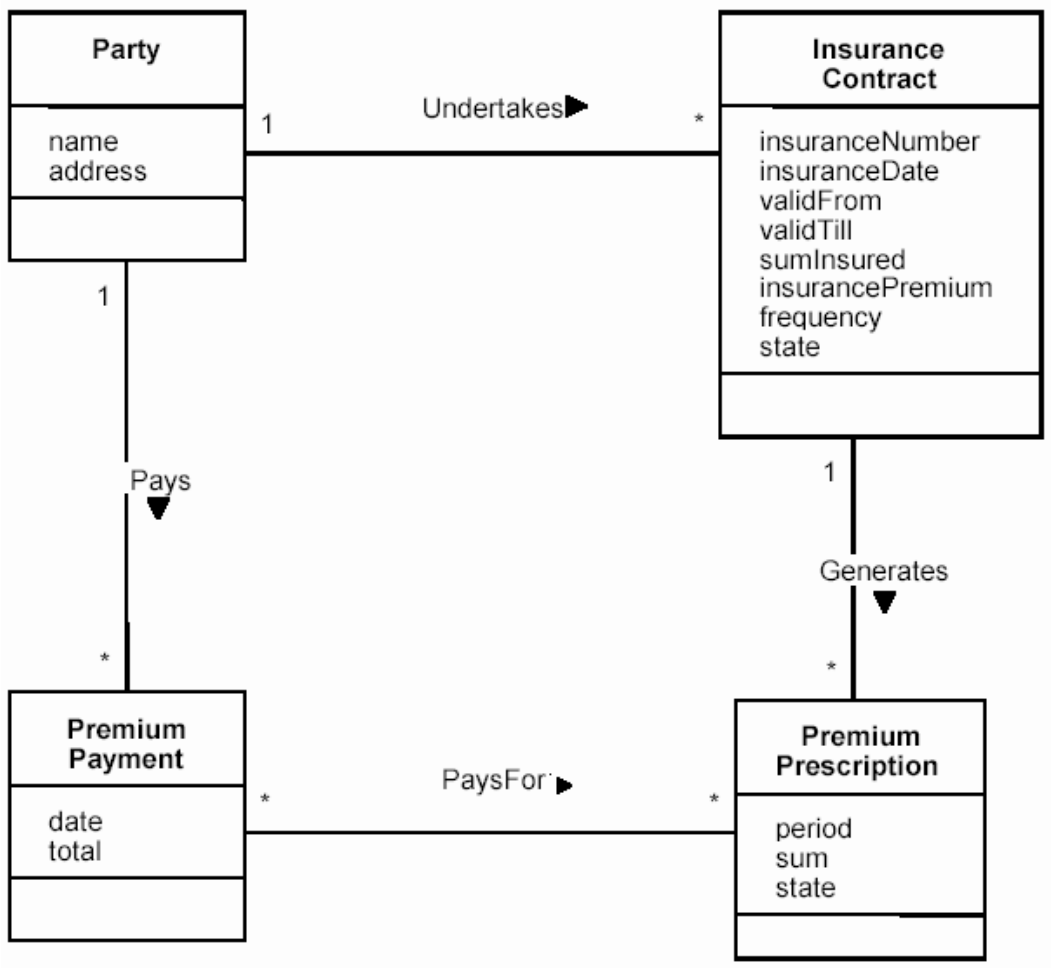


图 1 保险的例子

4 约束

- 这个系统相当复杂：一个组织可以同时有多个责任，每个责任又依赖于不同的条件，而这些条件时刻都有可能改变。
- 用户希望简单的模型，不希望模型有太多的类而难以理解和维护。
- 责任及它们的支付必须在用户进行交互工作时被及时、正确地发现。
- 模型要足够灵活以适应多样化的职责、组织、支付。

5 解决方案

将责任从职责中解耦。包括在任何时间间隔生成责任的操作。这个操作的每一次执行生成责任，该责任在这个时间段内有效。记录所有必需的数据以免日后再算。将支付分配给这些责任。

6 需求

1. 定义一个职责类型

由领域专家定义职责的类型包括约束条件和金额数量。

约束条件包括：

a. 组织角色类型的约束。

b. 用来确认职责的条件。

c. 当职责类型和多笔金额相关时，决定金额总数的条件。

d. 用于责任生成的条件。

2. 管理一个职责类

一名登记员建立职责并管理它的更新。除了管理职责类之外，还要管理这个职责本身，包括的组织类、组织类的角色以及支付方式。

3. 生成责任

由会计（或者由系统时钟）运行自动生成责任的操作，该操作基于职责类中的有效数据和频繁价格。

4. 处理支付

由会计负责处理责任的支付，系统帮助确认一个组织及责任支付是否成功。

7 结构

子模式（核心理念）

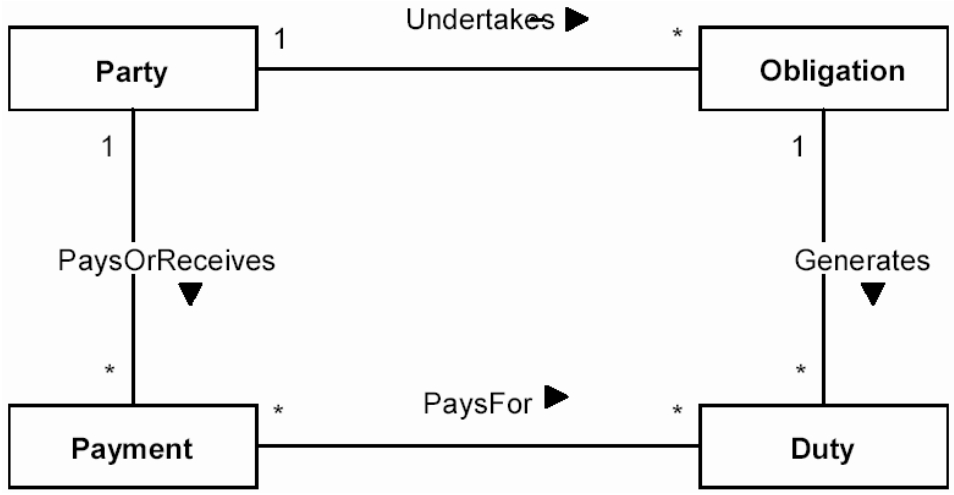


图 2 复发责任的核心子模式

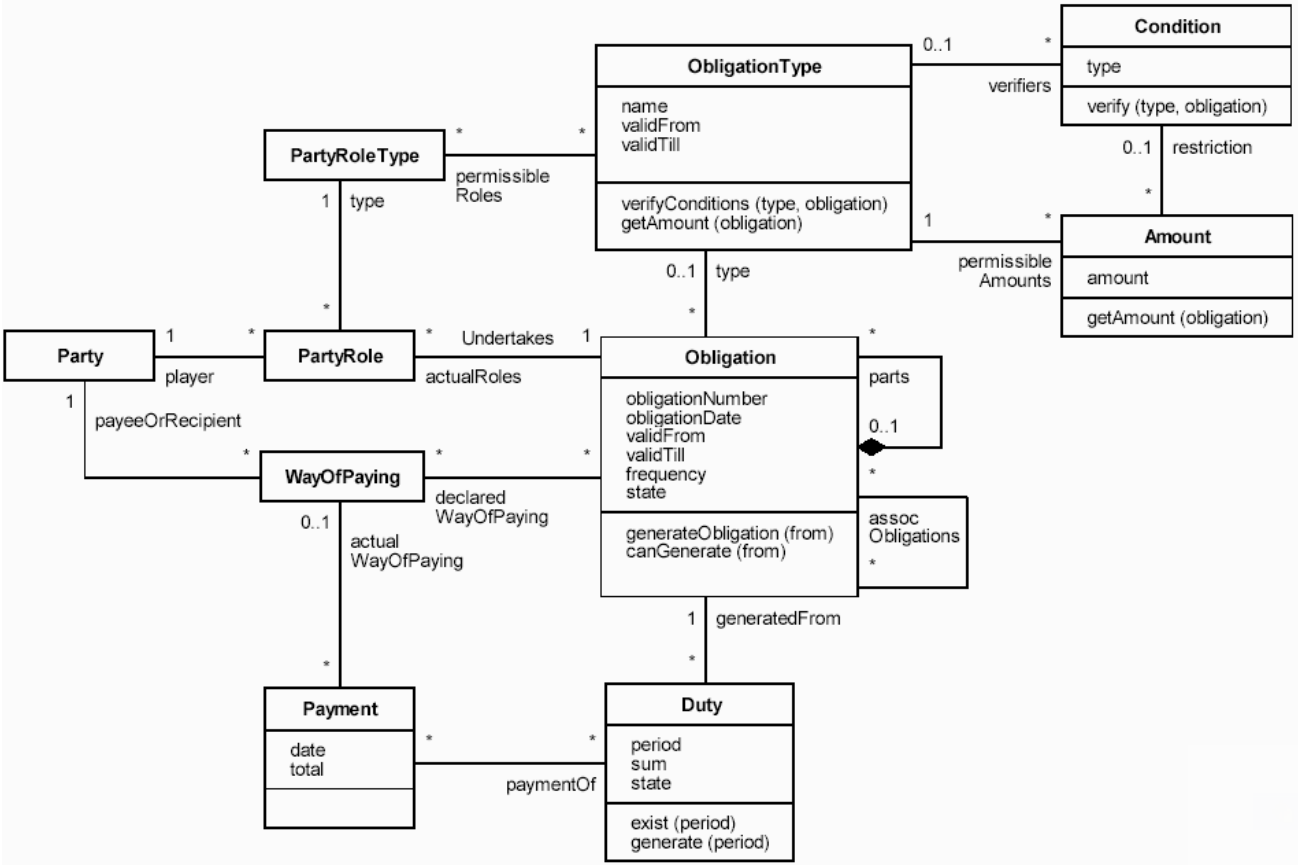


图 3 完整的复发责任模式

8 参与者

模式包括两个抽象概念层次的类[Fowler 1997],知识层（职责类型、组织角色类型、条件、金额总数）和操作层（其它所有类型）。在知识层，职责类型与不同的条件相关联应用于该职责类型生命周期中不同的阶段。条件有很多种功用：确认一个职责、选择责任数量以及当生成一个责任时进行再次确认。更进一步，职责类型像责任一样被分配了允许范围内将要支付的金额数量。总额可以是一个绝对值也可以是一个参数函数的实参（如保险业中的百分比税表）。假如很多种金额数量被分配给了同一个职责，它们将按照生成责任时的评估条件区分。最后，职责类型指定许可的组织角色类型。

操作层由三个以用例需求描述的类集合组成：

1．职责和相关数据

如适用性部分所说，职责是一个从各种各样的（法律）文件中抽象出来的抽象概念（如合同、订单、申请表）。将它和组织解耦是因为一个组织可以同时承担多个职责（如一个客户可以和同一家保险公司同时签有多份保单）[Fowler 1997]。职责可以有多个组成部分：如一定的风险也属于保单的一部分。我们在图 3 中通过递归聚合说明了这种组合的可能性。

职责通过组织的角色和实际的组织相关联。通常，一个职责和多重组织角色相关联。举一个国家社会保障的例子，在一份家庭津贴申请书中，必须根据家庭成员的不同角色填报该家庭中的所有成员。

为了使支付能够被确认及让对方组织知道如何付款，由主要的对立组织，如支付者或收付者来声明支付该责任的方式。支付方式是一个抽象类，银行账号、信用卡号或者邮寄地址是支付方式不同的具体子类的实例。

2．责任

责任是一个组织对另一个组织的一份债务或一条应收款项。责任由职责生成。责任和确定的时间段相关，并包含所有必需数据，以避免低效且有风险的重复计算，特别是对于再计算金额的总数来说。

3．支付（Payment）

支付掌握组织的实际支付。从应收款项的角度说支付被分配给支付者，从债务的角度说支付被分配给接收者。由于组织有多种交付方式，我们间接通过交付对象来指定支付的分配。更进一步，支付和它所支付的责任及和这个责任相关的其它责任相关联。

支付作为应收款项，背后包含了具体的运算法则。它分成两个步骤：首先确定支付者（利用数据库中声明的支付方式），然后找出支付者应承担的责任。每一步分别建立相应的连接，如分别给支付方式或给责任建立连接。

9 协作

在序列图中（图 4）描述了“生成责任”的步骤和条件。为简单起见，该图没有说明外延部分和例外情况。一条被引用的参数消息直接描述了它的数值（相对于还要通过计算参数才能得到的消息来说，例如一条表达式或一个变量的值）。

会计启动由职责类完成的一组操作。职责集合负责生成责任的操作。首先它试图找出在给定的时间段内责任是否应该被生成（职责有可能过期或者责任已经生成），然后，评估条件（一些属性的有效性可能过期），接着，算出金额的总数。如果有多笔金额，实际金额总数将依赖于其它条件。至此，责任就被生成了。

图 4（注：文中缺）中的消息用来定义图 3 中类的操作。

10 结论

复发责任模式对于一些债务也同样有用：

1．将职责从组织中解耦，这种方法在大部分应用领域里是清晰的，并且大部分领域专家和软件开发人员都能接受它（如保险业中的直接付款合同）。但在其它领域这样做可能会让人有些迷惑（例如在公司向国家劳动局缴纳税费，而没有合同的情况下）特别当一个组织同时有多重职责的时候，职责和组织的解耦是必要的。否则这些实体就混合在一起了。

2．职责可以包括有各自角色的多个组织。但这似乎很少被真正用到（如之前提到的家庭津贴的例子）。大多数情况下一个组织就足够了，也就是一个组织充当另一个安装了软件系统的组织的对立方，如此可以简化模式。

3．该模式描述了一个说明性的条件表示法和金额总数，但没有涉及太多的细节。在国家社会保险的软件系统中，我们开发着这些条件，使它们被详细说明，并加以更加详细的应用[Phare 1999]。不幸的是，那些表示法过于关注应用细节而几乎变成现今的一条条规矩了。

4．该模式假定支付被分配给了责任。而在某些系统中这是不必要的，它们只需要通过所有的责任总数来平衡所有的支付总数。然而另一些系统则不得不做得更精确些。将支付分配给责任可能是一个复杂的任务，一项支付可能支付了多个责任，而一个责任也可能涉及多项支付（图 3 解释了责任和支付间多对多的基数关系）。由于支付

如何分配给责任的算法涉及特定的领域或企业专利的问题，这里只将可能性做了大概说明。

5．该模型中没有描绘历史情况。例如，一个组织日后可能要更改一些职责的属性，那时根据这些变化，责任将以不同的方式计算，但并不影响先前的责任。变化前的属性值应该保留，以便变化前的责任在将来的任何时候核对它们。因此历史数据也应该被保留在知识级的类中（如条件、总额等）。

11 已知应用

复发责任分析模式的应用在保险业中相当普遍。我们在我们企业的软件系统开发中也使用了该模式，包括传统保险系统（如财产保险）[Koop 1999], [Sesera 2000b]和社会保险系统（如健康保险和失业保险）[Health 1996]。

该模式还可以用于贷款和分期付款领域，这里我们没有提供明确的参考资料。

所有以上提到的模式都涉及到收入（应收款项）。然而，模式也可以用于债务（未付款项）。例如复发责任分析模式在 ISPO 国家社会保障信息系统中的应用[Phare 1999], [Sesera+ 2000a]（见图 5）。

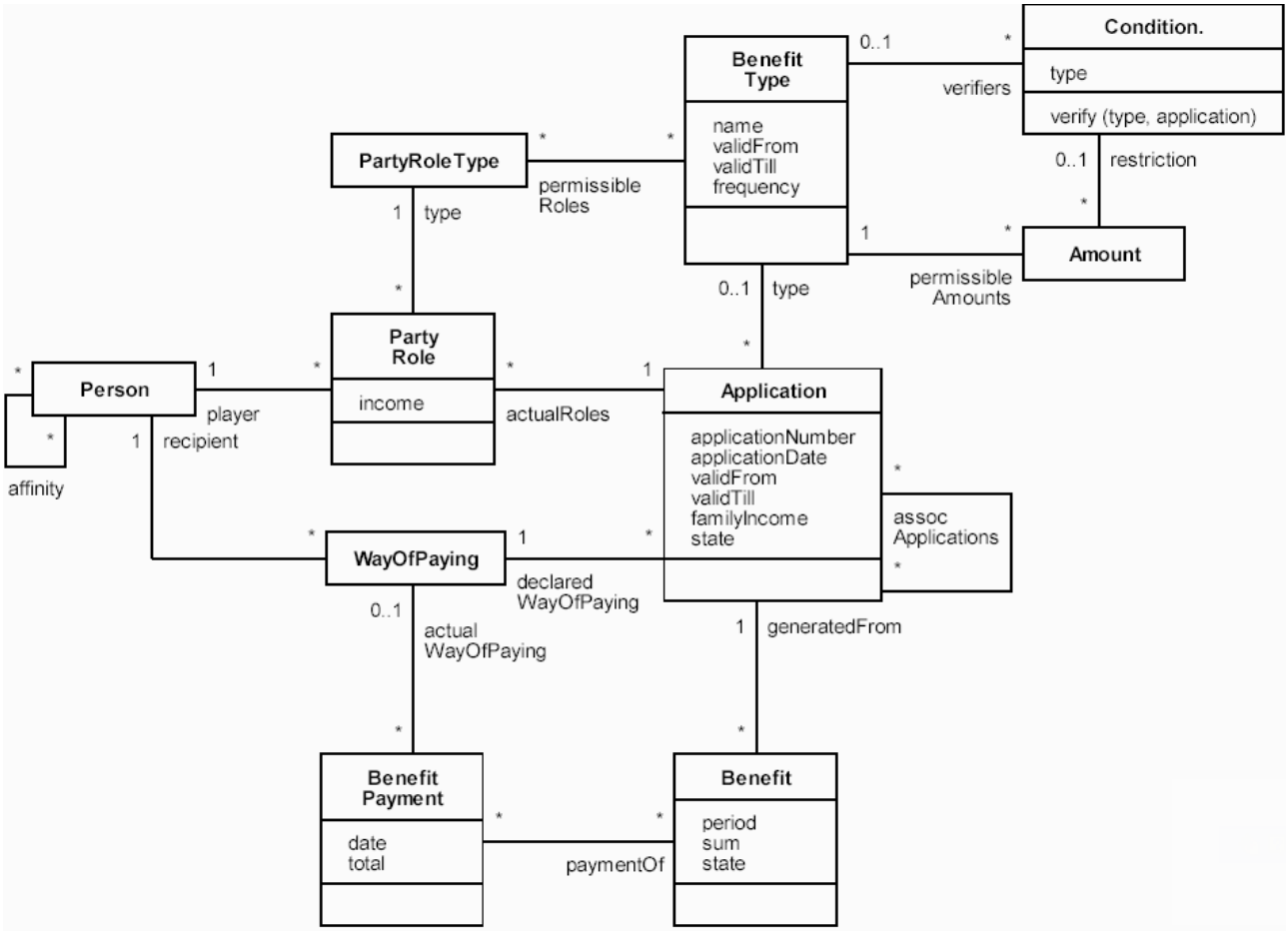


图 5 在国家社会保障领域的应用

12 相关模式

复发责任分析模式中用到了 Fowler 的有责任模式（Accountability pattern）作为它的子模式([Fowler 1997] p. 25)。即职责相对于有责任，职责类型相对于有责任类型。我们的模式允许职责和它的组织之间有 n 维关系，也就是将有责任模式改变成了（Contract Roles pattern）合同角色模式（[Hay 1996] p. 107）。

Martin Fowler 还提出了另一个“复发”（recurring）模式，叫做复发事件（Recurring Events）[Fowler 1996]，但他的模式和我们模式的目的不同。复发事件模式专注于多种不同的事件在不同的时间的重复发生，而与任何（金融）责任无关。相反的，我们的目标是金融责任，而不是它们发生的时间如何安排。不同类的责任往往有规律地一起产生（如每天或每星期）。假如需要，这些模式可以一起（组合）使用。

复发责任模式也可以用于暂时性的产生。职责和责任的价值和它们的实效性相关。职责和责任是明确的，它们二者都是暂时关联模式（Temporal Association pattern）中的暂时关联实体的实例[Carlson+ 1999]。

复发责任模式的思想来源于保险业。Wolfgang Keller 可能已经为这个领域提供了一些有价值的模式[Keller 1998]，但他的模式性质不同。首先，他们虽专注于保险业务，但我们的模式更加一般化。其次，他的模式一般粒度较大（见例子—保险价值链），而我们的模式是非常精粒度的。我们主要的区别是兴趣方向不同。Keller 属于知识级，关注于产品定义（见模式：产品树、对象事件保证等）和特定情况下的政策（政策作为产品实例），而我们的目标是操作级，也就是关注于收入。

感谢

我感谢我的导师 Ed Fernandez 和他的有价值的评论、建议以及所有当论文还在修改阶段时对我的帮助。我还要向 Stephen Sebesta 说谢谢，谢谢他的评论。

参考文献

[Booch+ 1998] Booch, G., I. Jacobson, and J. Rumbaugh. Unified Modeling Language User Guide. Addison-Wesley, 1998.

[Carlson+ 1999] Carlson, A., S. Estepp, and M. Fowler. Temporal Patterns. In: Pattern Languages of Program Design 4. Addison-Wesley, 1999.

[Hay 1996] Hay, D. Data Model Patterns: Conventions of Thought. New-York: Dorset House, 1996.

[Fowler 1996] Fowler, M. Recurring events, PLoP'96.

[Fowler 1997] Fowler, M. Analysis Patterns: Reusable Object Models, Reading, MA: Addison-Wesley, 1997.

[Gamma+ 1995] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995.

[Health 1996] Information System of the General Health Insurance Company. 1996 (in Slovak).

[Keller 1998] Keller, W. Some Patterns for Insurance Systems. PLoP'98. Also at: http://ourworld.compuserve.com/homepages/WofgangWKeller/

[Koop 1999] Software system for the Kooperativa insurance company. 1999 (in Slovak).

[Phare 1999] Information System of the State Social Support. Phare # 951801, 1999.

[Sesera+ 2000a] Sesera, L., A. Micovsky, J. Cerven, J. Architecture of software systems: Analysis Data Patterns. Textbook. Slovak Technical University, 2000 (in Slovak) (in print).

[Sesera 2000b] Sesera, L. Analysis Patterns. (Invited talk.) In: SOFSEM'2000. Lecture Notes in Computer Science series, Springer-Verlag, 2000 (in preparation).

The Addison-Wesley Signature Series

Patterns of Enterprise Application Architecture

中译本即将上市

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

UMLChina 训练教材

状态图模式语言

Sherif M. Yacoub、Hany H. Ammar 著，戴远志 译

序论

有限状态机及其扩展的状态图广泛应用于应变系统中。为了描述实体的复杂行为，David Harel[Harel87]对有限状态机进行扩展，提出了状态图的概念，其表示形式及规约已经在许多系统中得到应用。在面向对象的应用系统中实现一个实体的状态图规约时，系统设计人员会经常遇到一些设计问题，包括如何处理状态分层、正交以及事件的广播。本文将使用状态图模式讨论这些问题，并提出解决方案。由于状态图在软件系统设计中应用很广，本文将有助于系统设计人员在系统设计中直接应用状态图模式语言。

下面先简要回顾状态图概念的发展背景，然后通过一个模式关系图概述状态图模式及其与有限状态机模式之间的关系，余下各节详细论述各个模式。

背景

为描述实体的复杂行为，规范图形符号的表示方法，David Harel[Harel87,Harel88]对有限状态机进行扩展，提出了状态图的概念，后来Derek Coleman *et.al.*[CHB92]以状态图为基础提出了对象图，展现如何在面向对象的环境中使用状态图描述对象的生命周期，Bran Selic [Selic98]也采用状态图描述了递归控制的机制。此外，我们知道，状态图是一种经常被用作实体行为表示的技术，采用面向对象的状态图设计方法使得设计灵活、易于维护，我们将论述如何采用可重用设计方案，解决状态图概念的常见问题。本文作者建议对状态图不熟悉的读者，在继续阅读以下内容之前，首先参考David Harel[Harel87,Harel88]的图形化符号说明。附录B概要列出了状态图的原理与属性。

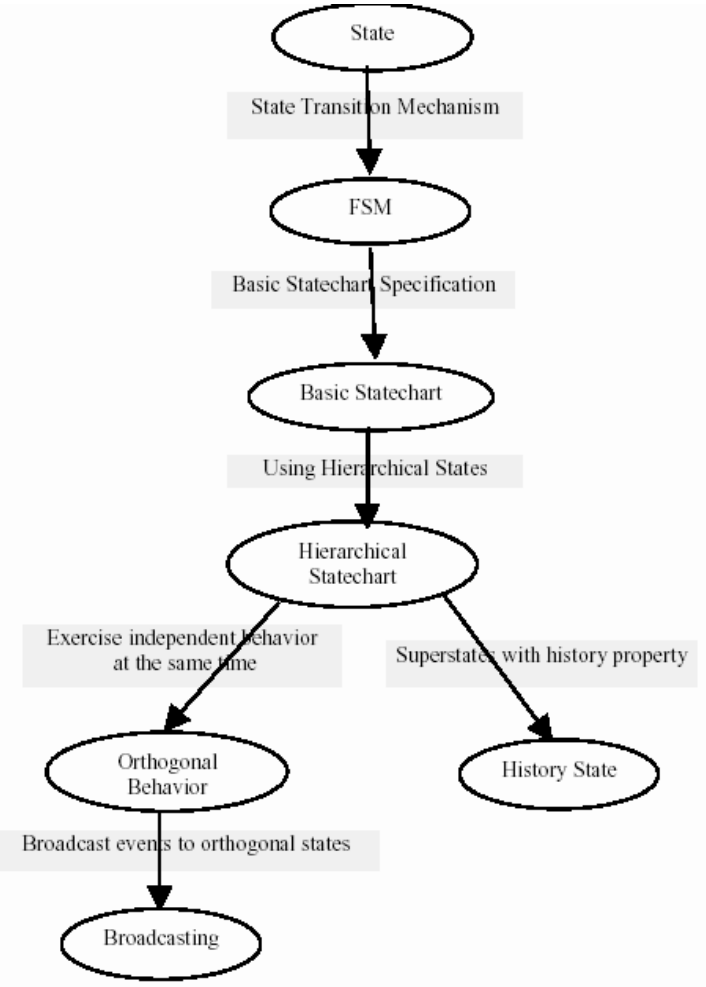
模式概述

对于在实现有限状态机过程中反复出现的设计问题，已有文献提出了解决方案。在Erich Gamma *et.al.*[GHJV95]的论述中，状态模式是一个基本模式，体现了实体中的状态相关行为，以区分状态类并在实体的接口上封装对象的当前状态。Paul Dyson及Bruce Andreson [DA98]在扩展并细化基本模式的基础上，提出了多个模式，用于解决数据成员在各

状态类间的分配、实体状态的显露、缺省状态的实现、以及状态转移机制的可选实现途径的问题。

Alexander Ran [Ran96]探讨了用以实现复杂行为实体的一系列模式。他讨论了设计一个面向对象状态设计模式(MOODS)的树型结构的可能性，该树型结构能够支持类方法及抽象状态的分解、通过条件或者状态转移方法实现状态转移的可能方式，并且说明了如何使用单一继承或者多重继承组织实体的状态类。

一套为解决状态图实现过程中面临问题的解决方案，从有限状态机模式发展而来。基本状态图模式把状态图规约的要素映射到面向对象设计中。分层状态图模式利用状态图分层原理，扩展了基本状态图模式，支持分层的状态关系，一个超级状态可以由几个状态组成。有时通过状态图的与分解规约，实体行为可以描述成非抵触的正交关系行为，这样就从分层状态图模式演变出正交行为模式，它支持正交状态的事件处理。使用正交行为模式，可能导致一个状态事件的效应被传递到另一个正交状态中，这就是广播模式。广播模式是正交模式的扩展，支持事件广播。最后，在分层状态图模式的超级状态中，有时会出现状态图的历史规约，称为历史状态模式。



State——状态；

State Transition Mechanism——状态转移机制；

FSM——有限状态机；

Basic Statechart Specification——基本状态图规约；

Basic Statechart ——基本状态图；

Using Hierarchical States——使用分层状态；

Hierarchical Statechart——分层状态图；

Exercise independent behavior at the same time——同时存在相互独立的行为；

Orthogonal Behavior——正交行为；

Broadcast events to orthogonal states——广播事件给正交状态；

Broadcasting——广播；

Superstates with history property——超级状态包含历史属性；

History State——历史状态；

图1 模式关系图

本文采用Robert Martin [Martin95]自动门禁闸机的示例，通过逐步演进的方式对上述各种模式逐一讲解，这是一个用来阐明如何使用状态图模式解决设计问题的恰当例子。附录A列出利用状态图模式可以解决的问题及其解决方案。

基本状态图

场景

在一个应用系统中，实体的行为取决于实体的状态，状态的变化与系统事件相关，而状态的转移由实体规约确定。我们选择状态图来表示实体行为。

问题

如何在设计中表示状态图规约？

动机

状态图是一种规约语言，可用于构造并检查模型。有些Case工具可以直接从规约生成代码，但是这种自动生成由于代码臃肿、不易理解，用处不大。因此我们希望能够把规约映射到设计中，这样可以提高系统维护的层次，同时可以把该设计合并到总体设计里。

解决方案

在面向对象设计中表示状态图规约时，根据规约中定义的状态类型，把实体的状态分别封装成类，确定每个状态类的事件、条件、动作、进入与退出过程。图(2)用UML符号[UML98]表示出该设计方案的结构。

参与者

Events

“Events”类声明实体欲响应的事件，不同状态下实体对事件的反应是不同的。对于在不同状态下实体反应不同的事件，在事件类中声明为虚拟方法，而在各状态类中实现各自的事件反应。对于在所有的状态中实体反应相同的公共事件，可以在状态基类中实现事件反应，而各状态类继承状态基类的实现(比如错误处理器)。

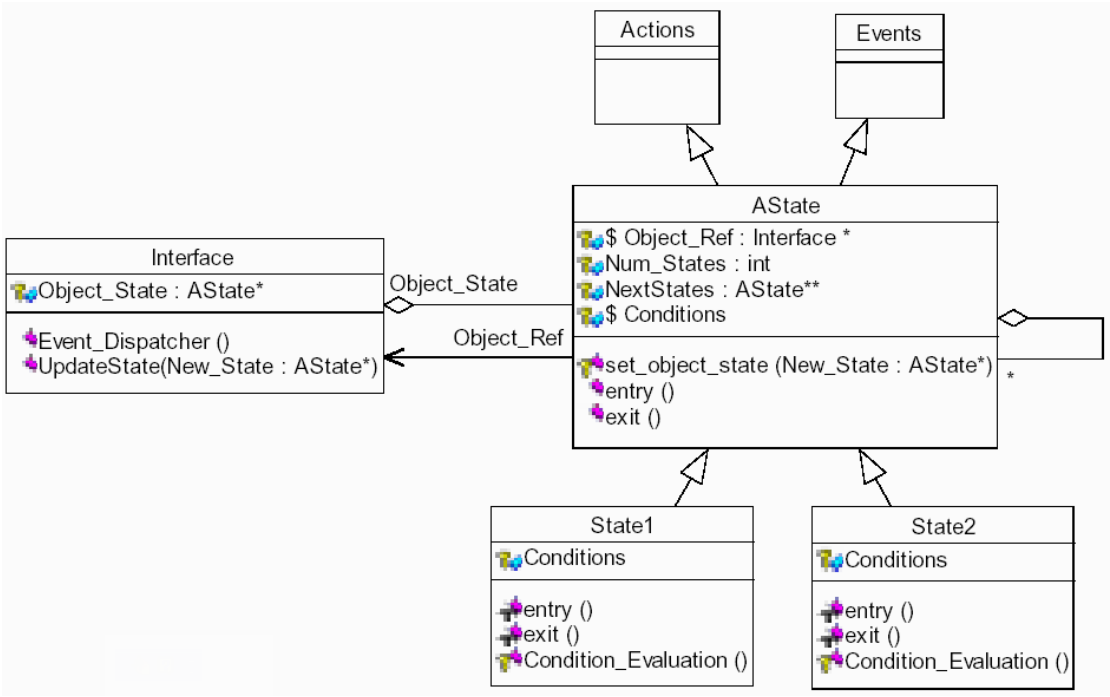


图 2 基本状态图模式结构

Actions

“Actions”类声明由实体调用的外向型方法，方法执行后可影响系统环境。状态类在事件反应方法中调用动作类方法，通常，动作类方法是状态类继承的静态方法。而只对于个别状态类有意义的私有动作方法只在其自身的状态类中声明。

AState

抽象状态类“AState”分组动作与事件，封装状态驱动转移[DA98]机制，在类中通过调用“set_object_state”方法改变实体状态。由于该方法在抽象状态类中声明，任何状态类均可调用以改变状态。该类具有一个指向自身的指针数组“NextStates”，每个状态类用它指向其它可能的状态类，用于实现状态转移。在设计时也可以采用自身驱动转移机制[DA98]，采用这种方式时不需要上述指针数组。抽象状态类通过对“Actions”类和“Events”类的多重继承建立状态类的行为，这两个类也可以直接合并进“AState”类中，不过对于有很多事件与动作的大型系统来说，把动作与事件区别对待，可以使设计更加清晰、易于理解。

States

这些类具体实现实体的各个状态。他们从“AState”类派生，并完成下列任务：

- 对于状态欲响应的各个事件，分别实现其具体的事件方法，触发适当的动作。
- 明确即将进入的状态，通过调用“set_object_state”方法完成状态转移。
- 保存识别状态的条件。可以采用以下形式：

a. 布尔数据成员。例如，可以使用一个简单的布尔数据成员“LinkEstablished”表示两个通讯实体间是否已经建立通讯联系。

b. 数据成员变量。例如，一个简单的条件[数量>=50]可以用一个数据成员(Amount)与一个条件检查语句(If (Amount>=50){..})来表示。状态类的事件方法使用该条件检查语句决定下一步的动作。本文后面自动门禁闸机示例中的硬币插入事件就是一个例子。

c. 复合条件。与上述的判断方式类似，但是语句更长，需要更多的计算。如果条件判断很频繁，为了简化最好把它放在一个条件判定方法中，该方法返回“真”或“假”，这样可以避免在其他方法中重复编码。例如，队列类中的“IsEmpty”方法就是一个条件判定方法，用来检查队列当前是不是空的。

对适用于所有状态类的条件，可以在“AState”类中使用静态数据成员来表示。

- 状态的进入与退出规约在类中体现为“entry”与“exit”方法，这两个方法在进入或退出状态时被调用。实体接口的“UpdateState”方法调用旧状态的“exit”方法，以及新状态的“entry”方法。

Interface

“Interface”类起到对外接口的作用，封装了状态图模式的处理逻辑。该接口保持当前实体的状态，其“Event_Dispatcher”方法从应用系统环境中接收事件，然后调用状态类的相应事件方法。“AState”类的“set_object_state”方法调用“Interface”类的“UpdateState”方法更新实体的状态。

示例

考虑一个最简单的自动门禁闸机的例子，下图表示其状态图规约。

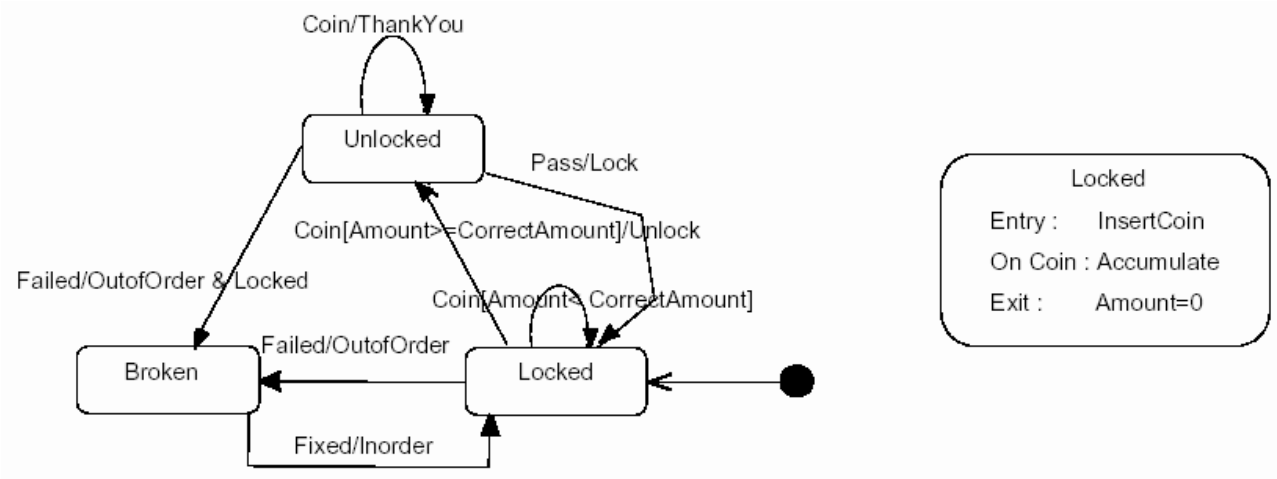


图3 自动门禁闸机状态图规约

下面叙述其中的参与者：

1. 状态类：“Locked”、“Unlocked”与“Broken”。

2. 事件方法：“Coin”为插入硬币事件，“Pass”为客户通过事件（？？？），“Failed”为故障事件，“Fixed”为故障排除事件。

3. 动作方法；图中显示了状态机在各种状态下的动作：“Unlock”方法允许客户通过，“Lock”方法防止客户通过，以及显示“Thankyou”消息的方法，显示“Alarm”警告的方法，故障时显示“Outoforder”消息的方法，故障排除后显示“Inorder”消息的方法。“Actions”类的方法实现上述动作。

4. 每个状态类以方法的形式实现进入与退出规约。例如，自动门禁闸机应跟踪记录插入的硬币数量，因此，在“Locked”状态中，自动闸机使用“Accumulate()”方法计算硬币数量。进入“Locked”状态时，自动门禁闸机显示一条消息，提示客户要先插入硬币再通过，这个消息在“Entry()”方法中显示。自动门禁闸机离开“Locked”状态时应把累计的硬币数量值清零，该清除动作在“Exit()”方法中实现。

5. 确定每个状态类的条件。例如，当硬币累计之和大于一个预定值(CorrectAmount)时，条件“Amount>=CorrectAmount”为真，因此我们在“Locked”状态中声明一个“Amount”属性来保存该累计值，并且在硬币插入事件(“Coin()”方法)中检查该条件。如果条件为真，允许客户通过，即调用“Unlock”动作方法，状态转移进入“Unlocked”状态，否则自动门禁闸机仍停留在“Locked”状态。

图4展示了采用基本模式结构的自动门禁闸机系统的状态图设计，为阐明问题起见，图中只提供了整个设计图的一部分。

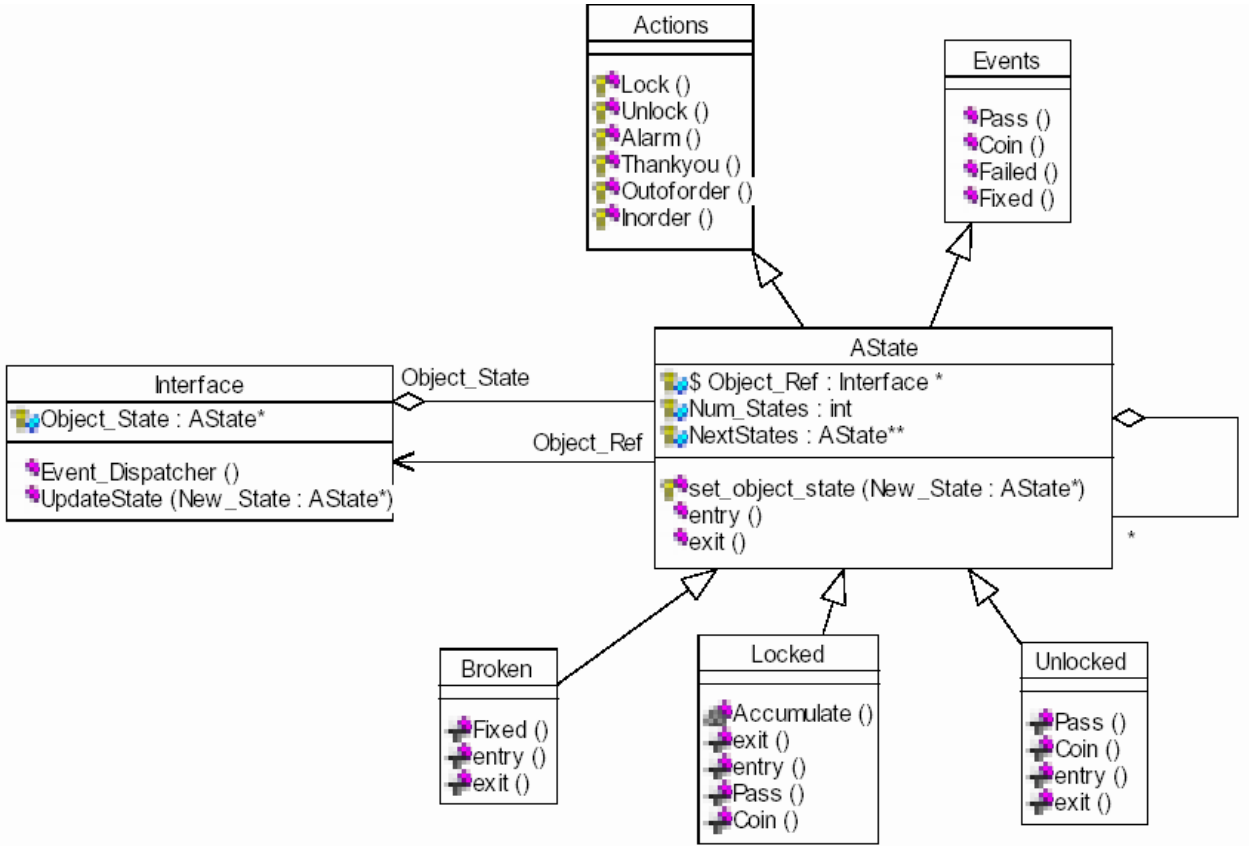


图4 采用基本状态图模式的自动门禁闸机系统设计

相关模式

从行为角度看，基本状态图模式与状态机模式相关。Robert Martin的《三级有限状态机模式》[Martin95]、Paul Dyson和Bruce Anderson [DA98]、以及Alexander Ran [Ran96]论述了有限状态机模式。然而，基本状态图更注重于状态图的形式化要素，比如状态的退出与进入方法、以及条件规约如何实现、在何处实现，这些问题在有限状态机模式中没有论及。每个状态类在同一时间只能有一个实例存在，因此状态类也可以看作是单例式[GHJV95]。

分层状态图

场景

你在使用基本状态图模式。系统很大而且系统中的状态似乎有层次。

问题

如何在设计中对状态分层？

动机

在使用基本状态图模式描述实体行为的时候，如果状态数量众多，状态之间就是一种平面结构的关系，这种方式不能准确表示状态之间的关系。如果利用状态图规约的分层特性，简化状态的表示方式，就会更容易理解。对状态图分层产生了超级状态的概念，超级状态是一个包含其他简单状态的状态。基本状态图模式不支持分层概念，需要修改设计以便可以在超级状态中装入其他状态。

解决方案

在设计中要把状态分层，首先必须确定状态的不同类型：

- 简单状态：不属于任何超级状态的一部分，并且不包含任何子状态的状态。(没有父状态也没有子状态)
- 叶状态：属于某超级状态的一部分，但是不包含任何子状态的状态。(有父状态但没有子状态)
- 顶层超级状态：封装了一组其他状态(子状态)，但没有父状态的状态。
- 中间超级状态：封装了一组其他状态(子状态)，并且有父状态的状态。

所有状态类型均继承自“AState”类，在类中使用“MySuperState”和“CurrentState”两个指针来表示父/子关系。图5表示分层状态图模式。

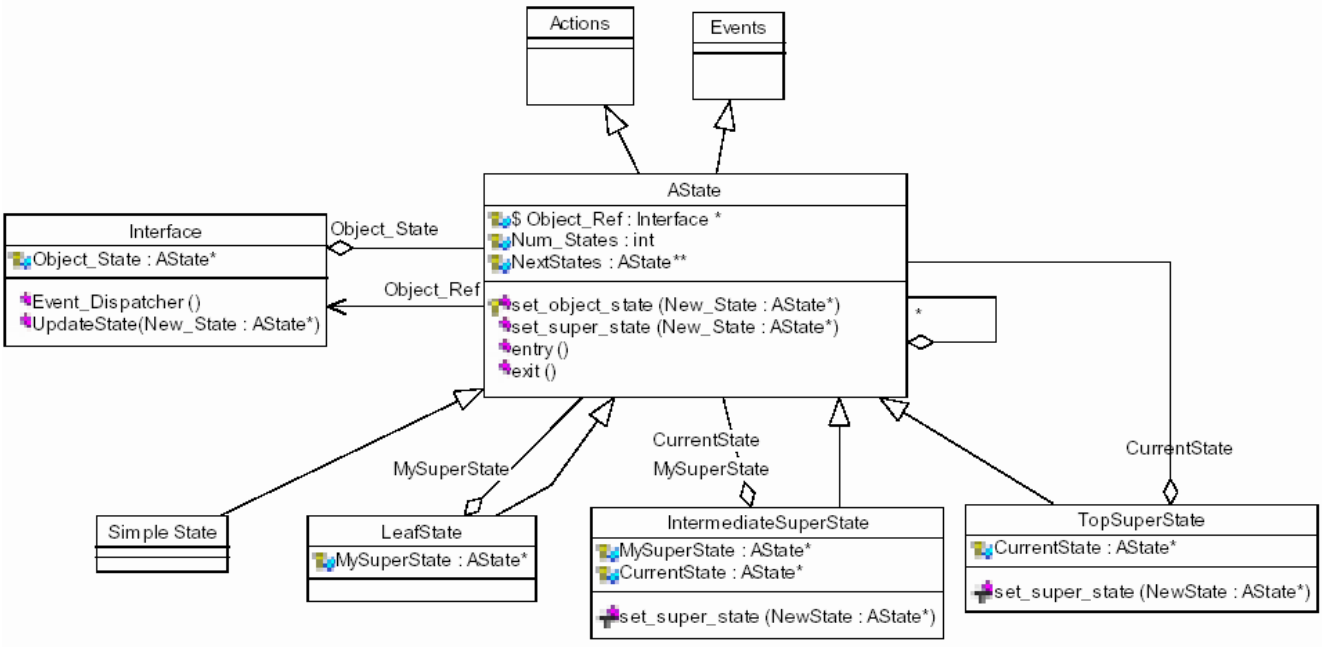


图5 分层状态图模式

参与者

除了基本状态图模式中的参与者之外，分层状态图模式还有以下参与者：

SimpleStates

即基本状态图模式中的“States”参与者。

IntermediateSuperState

同时具备TopSuperState与LeafState功能的状态。

TopSuperState

- 由于超级状态包含一组状态，使用“CurrentState”指针跟踪当前活动状态。
- 接收该组状态应响应的事件，并转发给当前活动状态去处理。
- 为各状态生成公共输出结果，也能为各状态实现公共事件处理方法。
- 完成从自身到下一个状态的状态驱动转移。
- 为整个超级状态实现进入与退出方法。

LeafState

具有与SimpleState相同的功能，并且

- 使用“MySuperState”指针修改其父状态的当前活动状态。

示例

同样使用自动门禁闸机作为示例(这个示例比较简单，不需要分层，这里为了解说的目的我们对它分层)，增加“Unlocked”类与“Locked”类的超级状态类“S_Functioning”。图6表示该示例的状态图。在图中，“Broken”是简单状态类，“Locked”与“Unlocked”是叶状态类，“S_Functioning”是顶层超级状态类。图中省略了条件与动作。

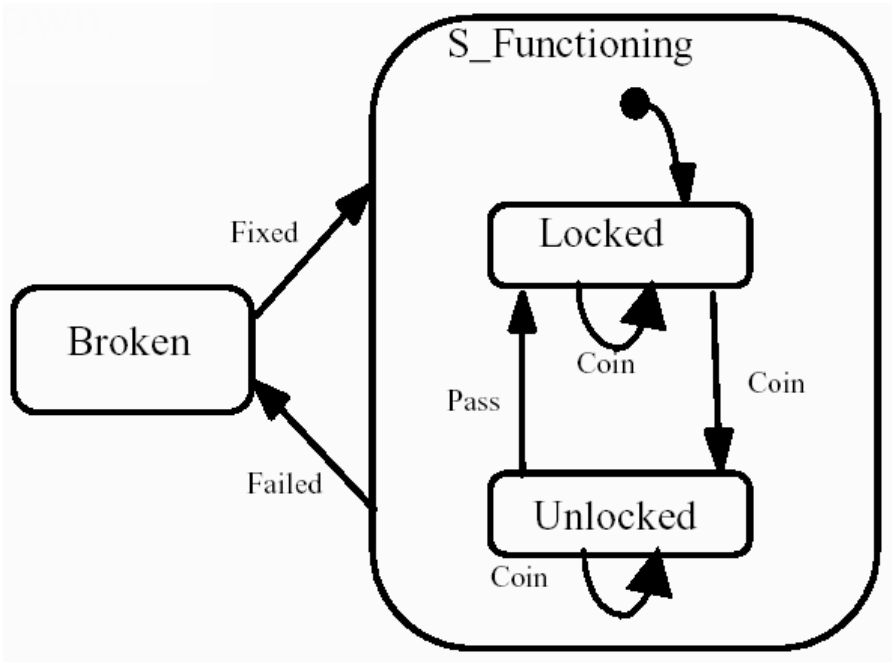


图6 自动门禁闸机示例的分层状态图

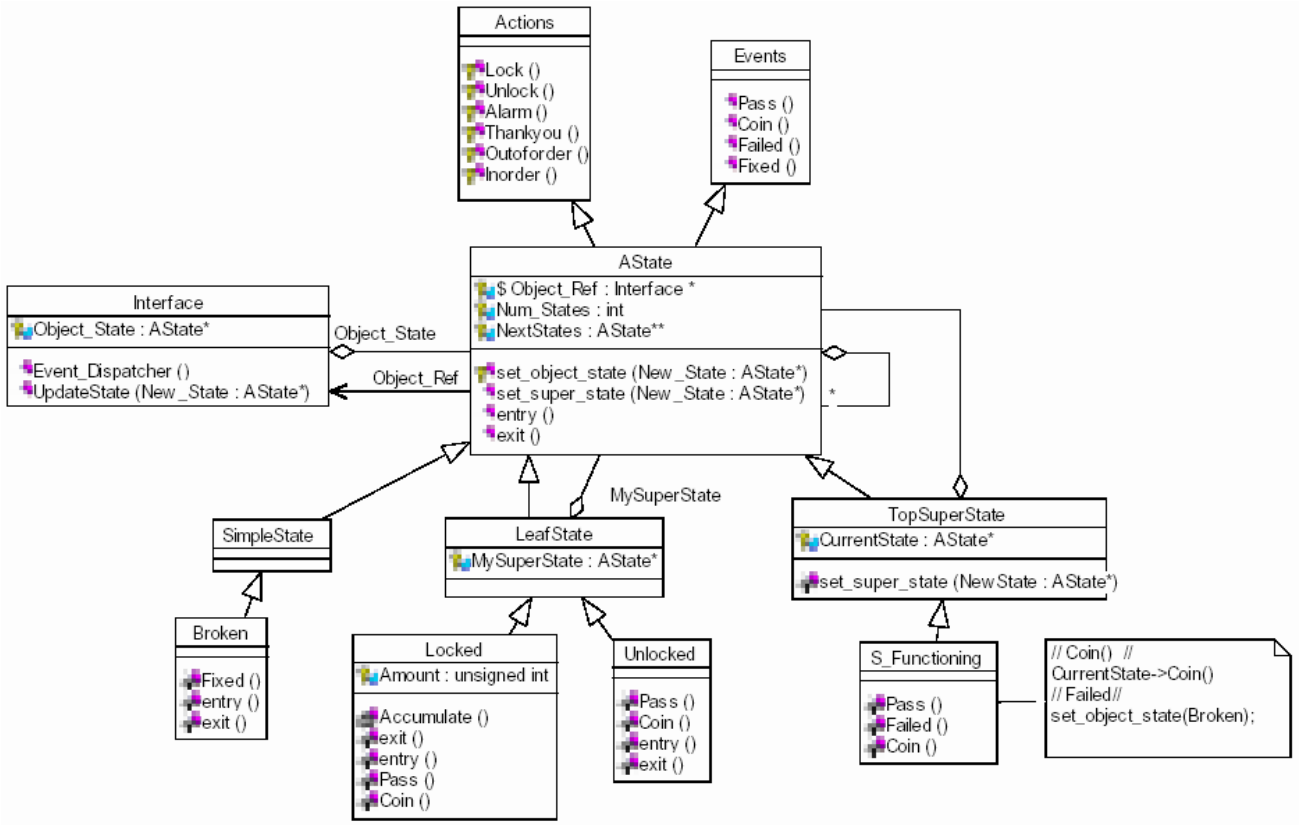


图7 使用分层状态图模式的自动门禁闸机系统设计

相关模式

分层状态图模式可以看作组合模式[GHJV95]，其中组合类(超级状态类)由相似抽象类型(AState类)的子类(叶状态类或其他超级状态类)组成。

正交行为

场景

使用分层状态图模式做系统设计时，实体具有同时存在的相互独立的行为。

问题

设计中如何编排实体的正交行为？

动机

把相互独立的状态进行分组，使每个个别行为只表示实体行为的一个侧面，以简化对实体行为的描述。以往的状态图做不到这一点，因为他们都是顺序出现的，非此即彼，不能同时出现，因此需要用到状态图的正交原理。分层状态图模式不支持正交行为，应在设计中添加进去。

解决方案

参考状态图规约，明确那些具有正交关系的超级状态，把事件转发给这些状态。定义一个“虚拟超级状态”，表示能处理相同实体事件的超级状态的集合。设计人员在虚拟超级状态类中对超级状态分组，其事件方法调用超级状态的事件方法。图8就是这样一个例子。图中虚拟状态“V”从实体接口接收事件，再转发给超级状态“A”和“D”。采用虚拟超级状态类，设计人员在实现状态正交时更加灵活，顺序正交可以通过一次调用一个正交状态来实现，并发正交可以通过触发每个状态对象的事件，每个触发动作启动一个线程来实现(假设底层操作系统支持线程)。这个方案解决了正交状态以及Harel et.al[HG97]论及的并发对象问题，并可灵活实现。需要注意的是，名称“虚拟”并不意味着该状态在规约中不能真实存在，它可以是一个有意义的超级状态，使用它仅仅是因为它“虚拟地”把正交状态进行了分组。图9示意了该方案的设计。

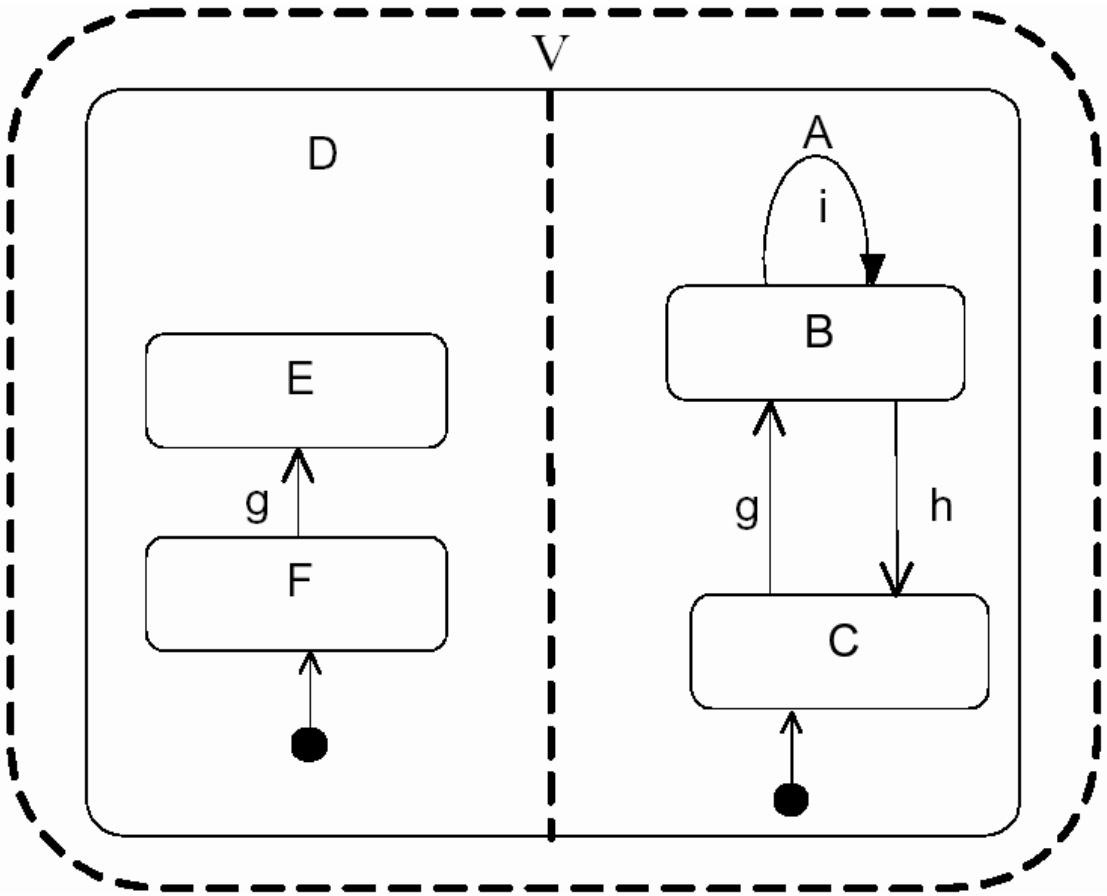


图8 虚拟超级状态示例

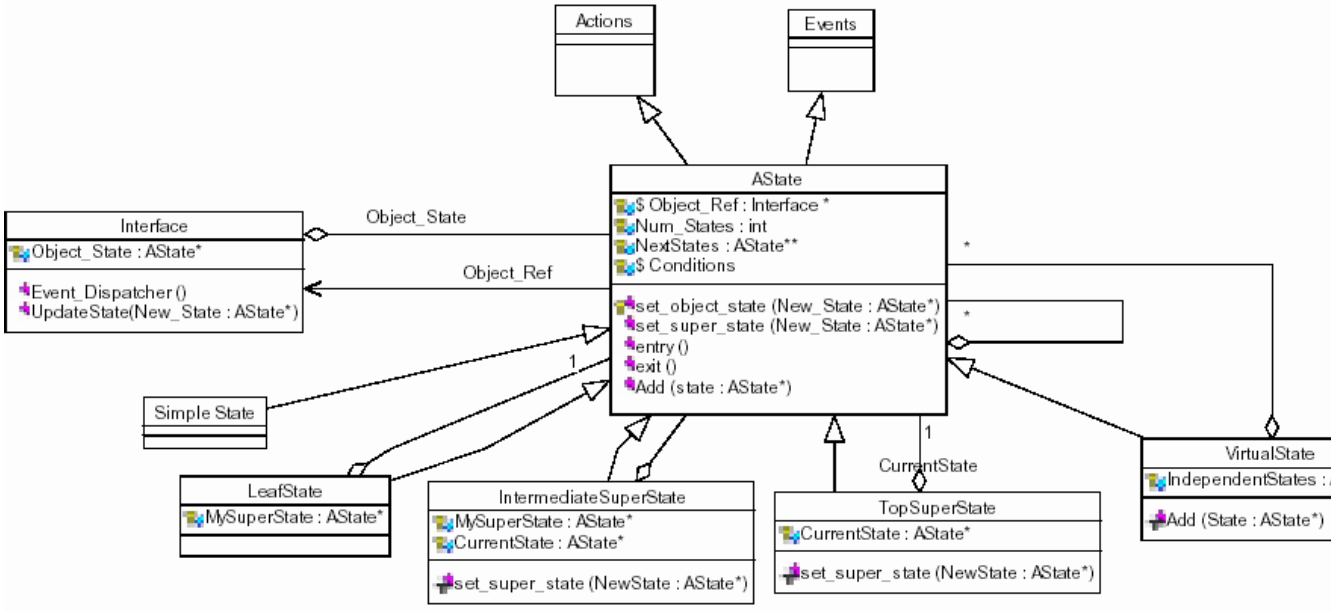


图9 正交状态图模式

示例

继续考虑上述自动门禁闸机，我们要给系统增加一个警报功能，当发生故障的时候发出警报，比如打开警报灯、显示一条信息、或者给管理者发一个通知。这样，我们就有了一个表示警报动作的独立行为，如图10所示。映射到设计中，需要新建一个虚类“V_CoinMachine”，该类包含两个超级状态“S_Warning”和“S_Operation”，它接收从接口传来的事件，再转发给两个超级状态类。图11表示了该设计的类图，图中省略了“Broken”、“Locked”等类。

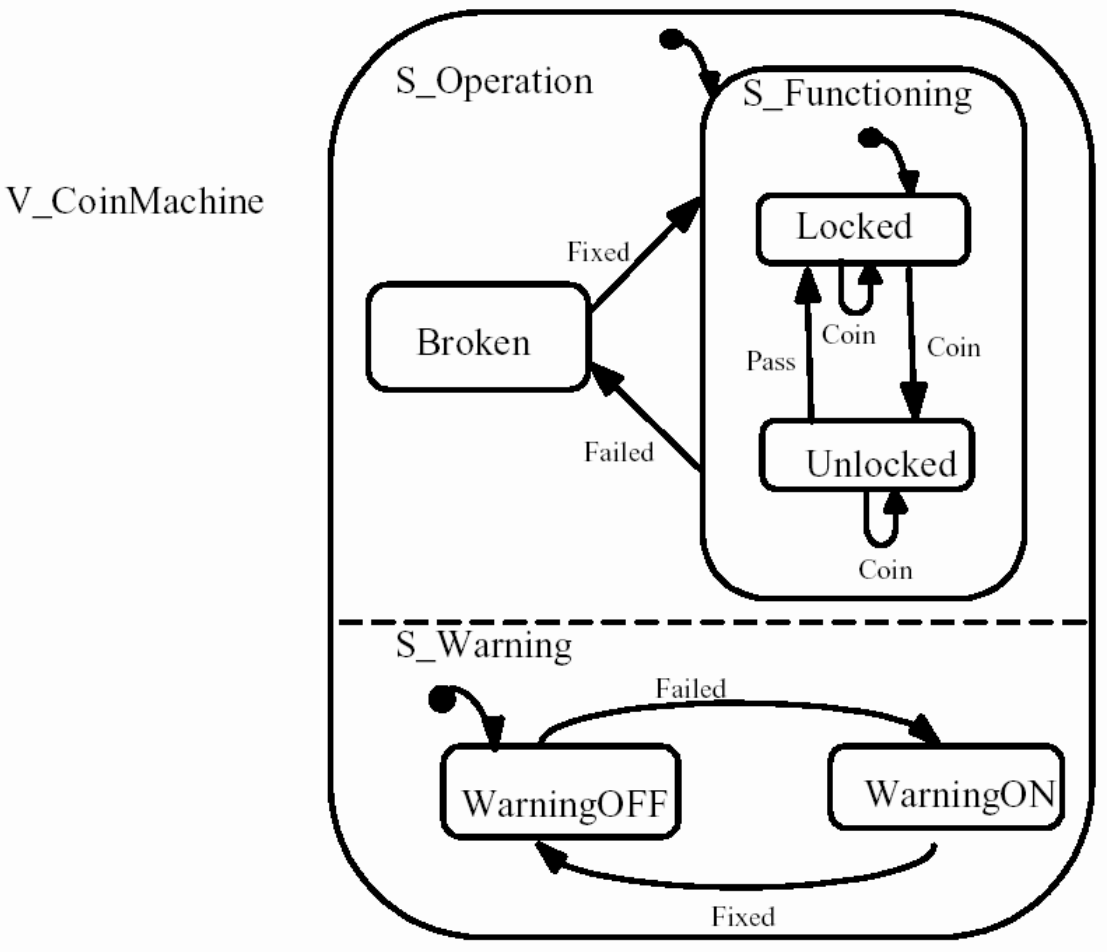


图10 自动门禁闸机的正交性

相关模式

正交行为模式可以看作组合模式[GHJV95]，其中组合类(虚拟类)由相似抽象类型(AState类)的子类(超级状态类)组成。

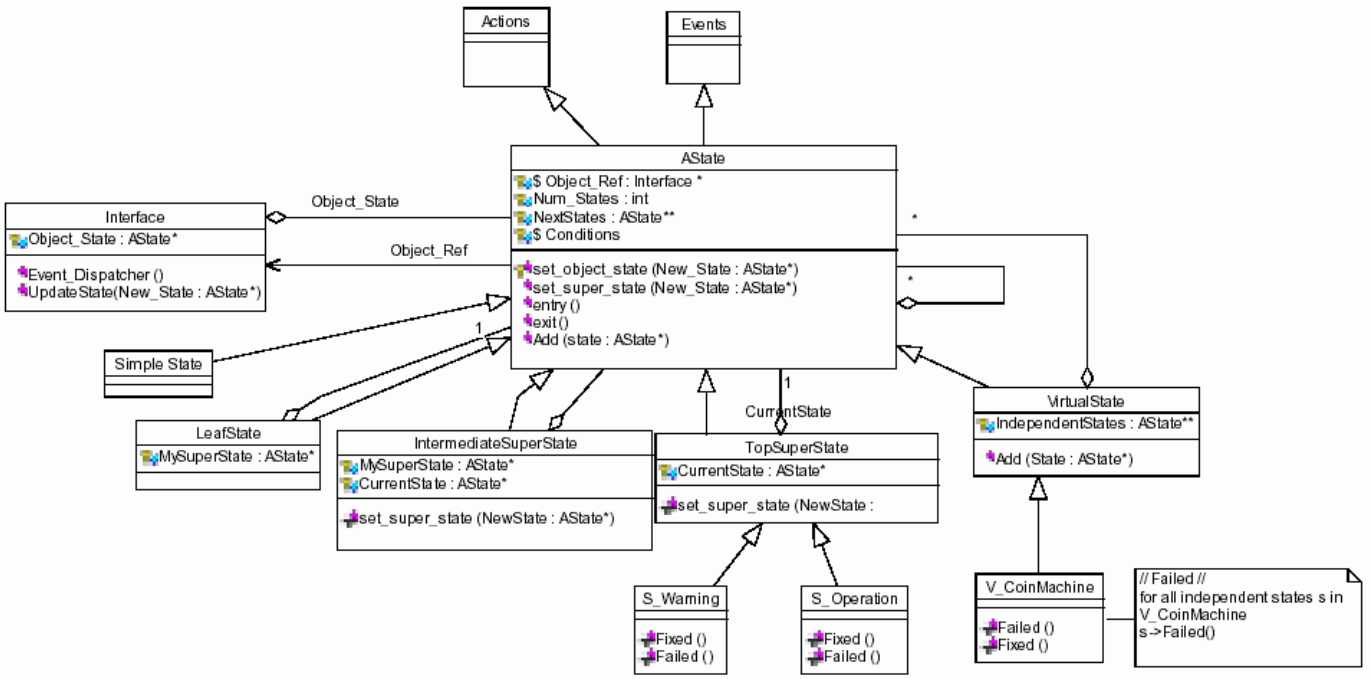


图11 使用正交状态图模式的门禁闸机系统设计

广播

场景

使用正交状态图做系统设计时，一个状态发生的事件引发正交超级状态的事件。

问题

发生事件时如何通知监听状态？

动机

对一个实体来说，正交行为描述了实体行为的一个侧面，逻辑上，正交行为之间仍然是有关系的。在使用状态图正交性的时候，一个超级状态的动作有时会触发另一个超级状态的事件。上述各种状态图模式都没有明确涉及到这种情形，但是利用其模式结构，我们仍然可以实现事件的广播。

解决方案

当一个新的事件发生时，广播状态可以直接把事件导入实体接口中，然后该事件就可经过虚拟超级状态传递到正交超级状态。这里使用的是正交行为模式结构，只有需要广播事件的状态调用“Interface”实体。

示例

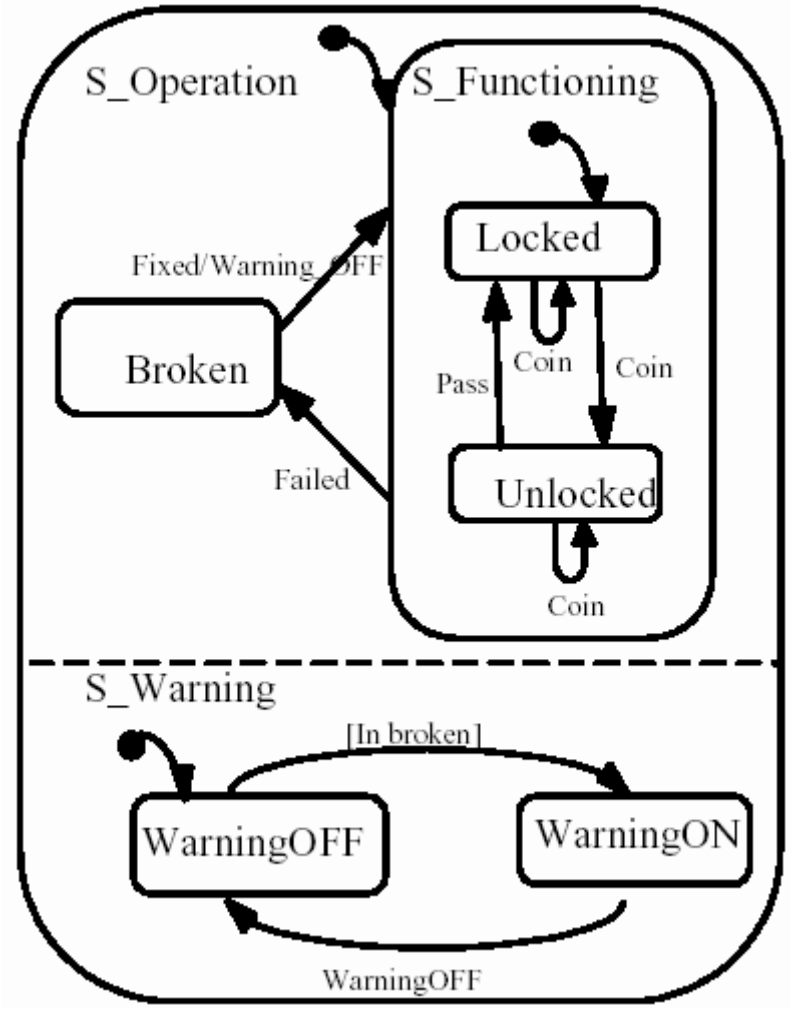


图12 带广播的自动门禁闸机状态图

在自动门禁闸机示例中，假设“Fixed”事件将激发“Warning_OFF”事件，该事件将把“S_Warning”超级状态的当前状态从“WarningON”转变成“WarningOFF”。这种情况下系统结构与正交行为模式相同，只是“S_Warning”超级状态实现的不是“Fixed”事件，而是“Warning_OFF”事件。“Broken”状态的“Fixed”方法通过调用实体接口广播“Warning_OFF”事件。代码如下所示：

```
void Broken::Fixed()
{//…
Object_Ref->Event_Dispatcher(Warning_OFF);
}
```

讨论

我们使用广播模式处理正交超级状态间的事件广播问题。事件发生有多种起源，有的事件是由前一个事件的动作引起的，有的是由于一个状态的某个条件成真引起而影响其正交状态，而有的是由于进入了一个新状态(In(状态)事件)。在这里我们只讨论了一种情形及其解决方案，其他解决方案也可以在本模式中实现，比如事件发生时对其排队直至状态转移完成这种方式。

历史状态

场景

使用分层状态图做系统设计时，一个超级状态有必要记住它在退出之前所处的状态。

问题

设计时如何保存超级状态的历史信息？

动机

当实体从一个超级状态转移到另一个，又返回原先的超级状态时，进入的可能需要是其退出时的状态，而不是其缺省状态。我们可以使用状态图规约的历史属性做到这一点，具体实现时超级状态应知道它最后的状态对象。对于不使用历史属性的超级状态，进入时“CurrentState”指针指向的是其缺省的内部状态对象，如下所示：

```
Void superstate::entry()
{CurrentState = DefaultState;
}
```

这个缺省状态是在超级状态构造方法中初始化的。而对于使用历史属性的超级状态，上述代码会使它丢失最后的内部状态。

解决方案

在超级状态类的创建方法中初始化“CurrentState”指针，而在“entry()”方法中不要重新设置，这样在进入该超级状态时该指针就保持了退出时的状态。

示例

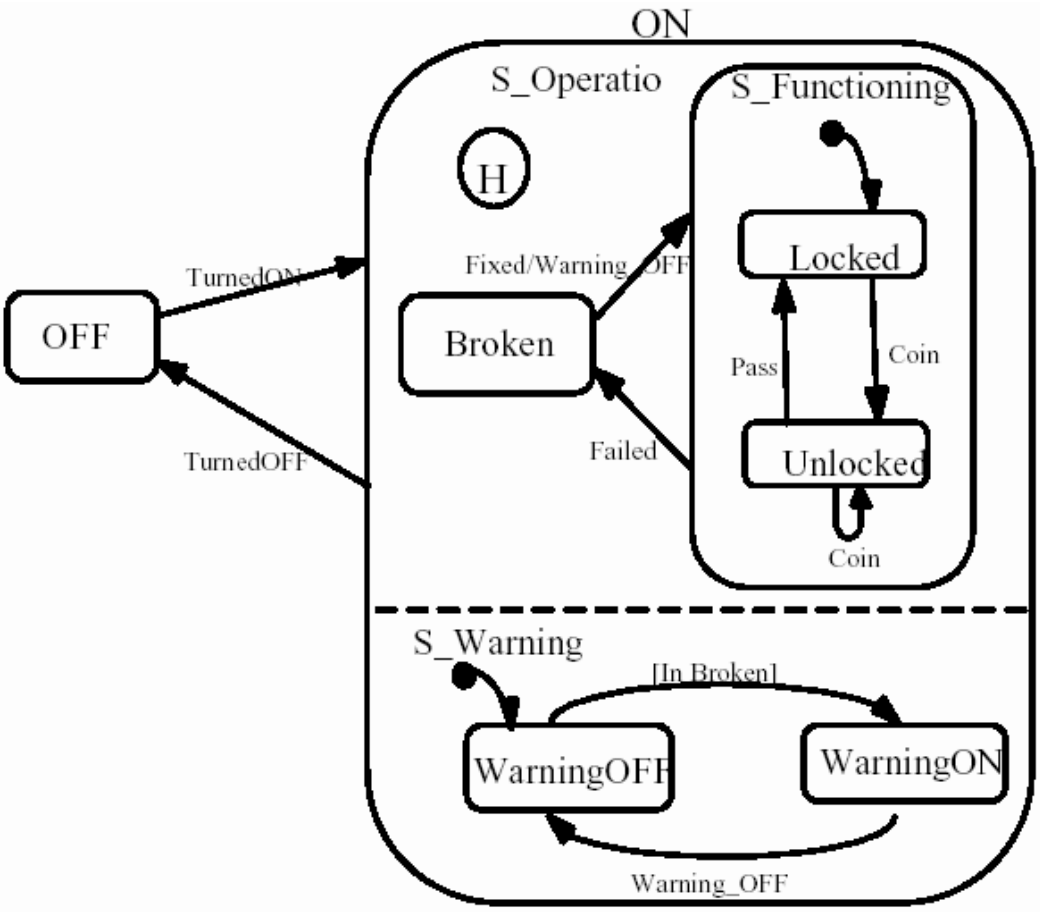


图13 带历史属性的自动门禁闸机状态图

在自动门禁闸机示例中，假设闸机可以开启、关闭，并且关闭时闸机能记住之前的状态，这样如果关闭之前闸机有故障，开启之后闸机应进入“Broken”状态，否则开启之后就应进入“S_Functioning”状态。这种情形下，与在分层状态图模式中的缺省状态实现类似，“S_Operation”超级状态的“entry()”方法也不需要设置“CurrentState”指针。

总结

有限状态机设计是系统设计人员面临的共同课题，在通讯系统中经常会用到。在这些系统中，两个或多个通讯实体间相关联的情况会限制上一层应用系统的行为。有限状态机广泛应用于控制系统中，比如在自动火车、升降机控制、以及自动火车通道控制中的运动控制系统。Gamma et.al[GHJV95]提到了在图形用户界面上的应用，Dyson et.al[DA98]也提到了在图书馆系统中的应用。自动取款机是最著名的例子之一，经常用于阐释状态在操作流程中起重要作用的应用系统。

已经有著作[DA98,Martin95,Ran96]论述了有限状态机模式，其中部分模式也适用于状态图，因为两者具有一定的共同特性。例如，Paul Dyson与Bruce Anderson [DA98]描述的缺省状态模式、显露状态模式、以及纯状态模式均可适用于状态图中的状态类。结合状态模式[GHJV95,DA98]、有限状态机模式[Martin95,Ran96]、以及状态图模式，可为解决实体行为表示中的问题提供一个全面的模式语言解决方案。

致谢

感谢Dennis DeBruler, 感谢他在修订模式和改善模式质量上的有价值评论和大力帮助。

参考资料

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software” Addison-Wesley 1995

[DA98] Paul Dyson and Bruce Anderson, "State Patterns". In Robert Martin, Dirk Riehle, and Frank Buschmann (edt.) Pattern Languages of Program Design 3, Addison Wesely Longman Inc 1998, chapter 9, pp125

[UML98] UML Resource Center. http://www.rational.com/uml/documentation.html

[Martin95] Robert Martin, "THREE-LEVEL FSM". In James Coplien, Douglas Schmidt, (edt.), Pattern Languages of Program Design. Addison-Wesely,1995, chapter 19, pp383

[CHB92] Derek Coleman, Fiona Hayes, and Stephen Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Designs," IEEE Transactions on Software Engineering, Vol 18, No. 1, January 1992.

[Harel87] David Harel, "Statecharts: a Visual Formalism for Complex Systems," Science Computer Program, Vol 8, pp 231-274, 1987

[Harel88] David Harel, "On Visual Formalism", Communications of the ACM, Vol 31, No 5, May 1988

[Selic98] Bran Selic, "Recursive Control". In Robert Martin, Dirk Riehle, and Frank Buschmann (edt.) "Pattern Languages of Program Design 3", Addison Wesely Longman Inc 1998, chapter 10, pp147

[HG97] David Harel and Eran Gery, "Executable Object Modeling with Statecharts", IEEE Computer magazine, July 1997, pp31-42

[Ran96] Alexander Ran, "MOODS: Models for Object-Oriented Design of State", In John M. Vlissides, James O. Coplien, and Norman L. Kerth (edt.) Pattern Languages of Program Design 2, Addison Wesely Longman Inc 1996, Chapter 8, pp119-142

附录 A：状态图模式一览表

模式名称	问题	解决方案
基本状态图	在一个应用系统中，实体的行为取决于实体的状态,我们选择状态图规约来表示实体行为。状态图规约在设计中应如何表示？	使用面向对象设计，根据规约中的定义，把实体所有的状态分别封装成类，识别出每个类的事件、条件、动作、进入与退出过程，并定义为状态类的方法与属性。
分层状态图	使用基本状态图模式设计一个大型系统，系统中的状态可以分成不同的层次。如何在设计中对状态分层？	从抽象状态类派生出超级状态类，使用组合模式，在超级状态中包含其他状态。超级状态应明确当前的活动状态，以便事件发生时可以转发给它
正交行为	使用分层状态图模式做系统设计时，实体具有同时存在的相互独立的行为。设计中如何编排实体的正交行为？	明确那些在状态图规约中具有正交关系的超级状态，定义一个处理同样事件的“虚拟超级状态”，起超级状态的集合的作用，以把事件转发给每个状态
广播	使用正交行为模式做系统设计时，一个状态发生事件时如何广播出去，触发其正交状态事件的发生？	当一个新的事件发生时，使广播状态直接把事件导入实体接口中，经过虚拟超级状态，事件最终转发到虚拟超级状态中所有的正交状态
历史状态	当超级状态具有历史属性时，设计时如何保存该信息？	在超级状态对象创建时初始化当前活动状态类指针，并用于实体的整个生命周期，在超级状态进入方法中不要重新设置

附录 B：状态图原理

状态图要素

状态：状态图的静态要素，描述实体在某一状态下的行为。状态有进入与退出过程，以及处理事件的方法。

触发器：状态图中引发状态转移或反应的动态要素，可以是事件、条件，或者两者都是。

事件：在某一时刻发生的事件，可以发生在状态图内部或者外部。

条件：布尔表达式，取值真或者假。条件可以是简单要素或者是复合要素，复合要素指由简单要素经过“与”、“或”操作所形成的表达式。

动作：状态图中由事件引起的操作。

活动：某一状态下实体执行的操作。

状态进入：一种特殊的事件类型，表示实体进入某一特定状态时应执行的动作。

状态退出：一种特殊的事件类型，表示实体退出某一特定状态时应执行的动作。

状态图原理

状态图是对有限状态机的扩展，支持以下基本原理：

分层

分层原理引入了一个称作“超级状态”的更具全局性的状态，它可以包含几个其他状态。分层原理有时也称作“深度”原理或“异或分解”，因为在同一时间里超级状态中只有一个状态能代表超级状态的实体行为。例如，在图14中，实体处于超级状态“A”时，其实是处于简单状态“B”或者“C”。

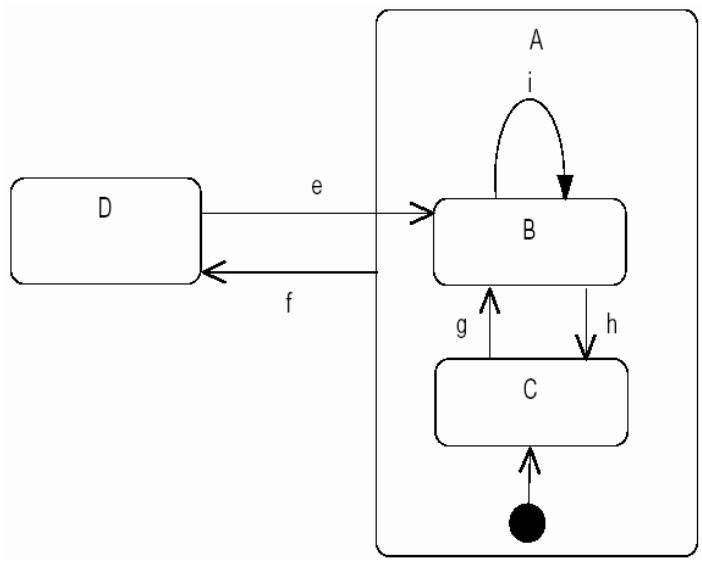


图14 状态图分层原理

正交

正交原理表示实体在同一时间可以有多种行为，一个事件可以同时导致几个超级状态发生状态转移。正交原理有时也称作“与分解”原理，因为多个超级状态可以同时发生变化。例如，在图15中，事件“g”导致实体在超级状态“A”中的行为从“C”转变为“B”，并且在超级状态“D”中的行为从“F”转变为“E”。

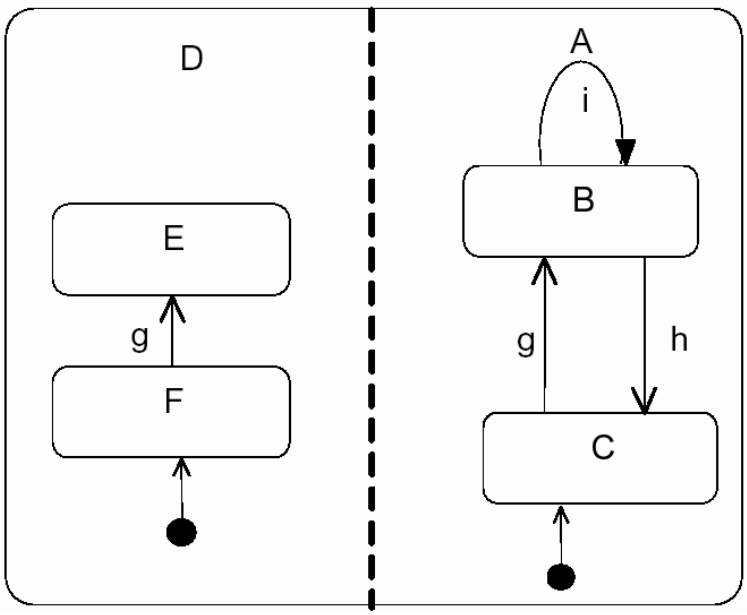


图15 状态图正交原理

广播

一个超级状态中发生的状态转移会引发另一个事件，该事件反过来又会触发另一个超级状态的状态转移。例如，在图16中，事件“g”触发了事件“k”，而事件“k”又导致了超级状态“A”从状态“C”转移到状态“B”。

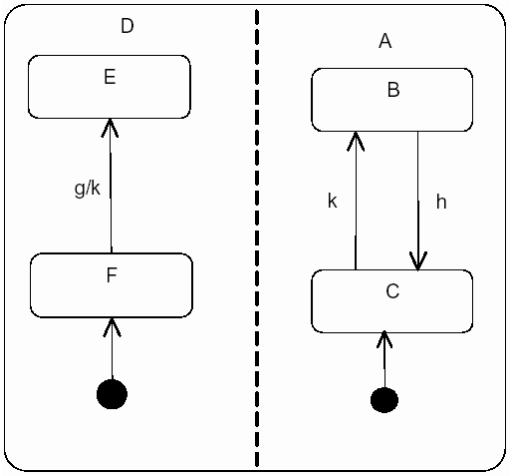


图16 状态图广播原理

历史

当实体从一个超级状态转移到另一个超级状态，然后又返回时，有时需要返回到前一个超级状态退出时的状态，而不是其缺省状态。例如，在图17中，假设当前状态是“B”，事件“m”使状态转移到“E”，这时发生了事件“p”，缺省情况下实体当前进入的状态应该是“C”，但由于超级状态“A”记住了其退出时的状态，因此当前状态变成了“B”。

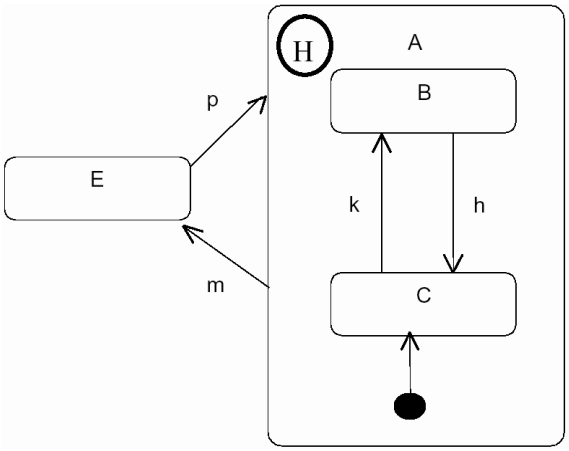


图17 状态图历史原理

综上所述，状态图＝状态机＋深度＋正交＋广播＋历史

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

设计模式中的可分析性和可更改性

Javier Garzás、Mario Piattini 著，黄蕾 译

讨论

摘要

面向对象的方法已经出现很久了，设计师们在OO系统的设计和构建方面也已经积累了很多经验。然而，目前单单使用模式（*patterns*）还不足以作为一种正规方法来指导设计。面向对象的设计方法中存在两个重要的质量因素：可分析性和可更改性。在使用设计模式的情况下，一些基本法则（*Principles*）可以使我们的分析工作更加简单。中间层（*Indirections*）的存在提供了设计模式的可更改性。为了兼顾这两方面的内容，本文建立一套度量标准，以确定在不丧失设计的可分析性前提下可获得多大的可更改性。

1 引言

80年代末期，在OO的诸多方法中，模式应用开始显露头角，通过Coad (1992)、Gamma *et al.* (1995)、Buschmann *et al.* (1996)、Fowler (1996) 和 Rising (1998)几位大师的努力，模式在OO系统中的地位逐步巩固。采用模式的目的是用于传递面向对象设计的知识（OODK）(Garzás and Piattini, 2001)，这些知识是多年实际工作经验的总结。自从有了模式之后，设计者们只是阅读和使用模式，充分享受前人经验带来的种种好处。

然而，目前模式本身还不足以作为一种正规方法来指导设计。设计者必须具有一定的经验，才能避免过载、因无知而不用或误用模式、或任何其他导致不能正确应用模式的问题。在应用模式的时候会出现以下几类问题 (Wendorff, 2001; Schmidt, 1995)：

- 难以应用
- 难以理解
- 试图把所有的东西都改写为模式
- 模式过载
- 无知

- 模式目录本身的不足：查询和复杂应用、对编程语言的高度依赖、其他类似问题等等

从原理上说，使用设计模式将提高设计质量。人们在设计质量及度量标准体系方面做了很多工作，例如：(Genero et al., 2000)， (Brito e Abreu and Carapuça, 1994)， (Briand et al., 1999)， (Henderson-Sellers, 1996)，等。既然设计质量可以通过指标来衡量，那么运用设计模式应该获得更好的指标。但是，许多面向对象设计的通用指标都显示运用模式的设计是低质量的设计。 Reibing 曾说过，如果对于同一问题有两种类似的设计 A 和 B，B 使用了设计模式，A 未使用，那么 B 的质量应比 A 高。但是，如果我们使用传统的标准来衡量这两个设计，指标显示 A 的质量更好——主要因为 A 中类、操作、继承、关联等更少。到底谁错了？是指标？还是模式社区？是我们使用了错误的质量标准来度量面向对象的设计？还是使用模式确实使设计变得更糟而不是更好？人们期望使用模式提高质量，而使用模式后质量指标却变差了，到底是什么原因导致了两者之间的矛盾呢？

在以下的章节中，我们将首先针对软件维护、设计模式以及可分析性和可更改性之间的关系进行更详细的讨论。然后，我们将提出一种度量标准以衡量模式的运用对设计质量的影响。最后几个章节分别是致谢、结论和参考书目。

2 软件维护和设计模式

根据 ISO/IEC9126-1999“软件产品评价－质量特征及使用指南”，可维护性可细分为可分析性、可更改性、稳定性、可测试性和一致性五个子特性。

软件整个生命期的费用大部分是花在软件维护上。(Pigoski, 1997;Bennett and Rajlich, 2000).不能迅速和可靠地更改软件意味着商业机会的丧失。因此，近年来人们在研究中越来越重视这些问题。

正确的 OO 设计可以提供更好的可维护性。按照 9126 的标准，对于面向对象的设计的可维护性来说，存在两个重要的质量参数：

- 可更改性：可更改性使设计容易修改，这一点在需要给已有系统添加新功能的时候显得尤其重要
- 可分析性：可分析性使设计易于理解。这是在可接受的时间段内完成设计修改的必要条件。

更细一点说，模式的使用对软件设计会产生两种完全相反的作用力，而这两种作用力均与可维护性直接相关。一方面，我们建立起来一套公共术语以便运用模式，也有了一些成功的解决方案。另一方面，我们所使用的解决方案可能会非常复杂，这意味着该设计不容易被人理解，对该设计的改动会更加困难。(Prechelt,2000)。因此，延续前面的作用力的概念，可分析性和可更改性之间显示出一种很奇妙的关系：增加设计的可分析性会降低其可更改性，反之亦然。

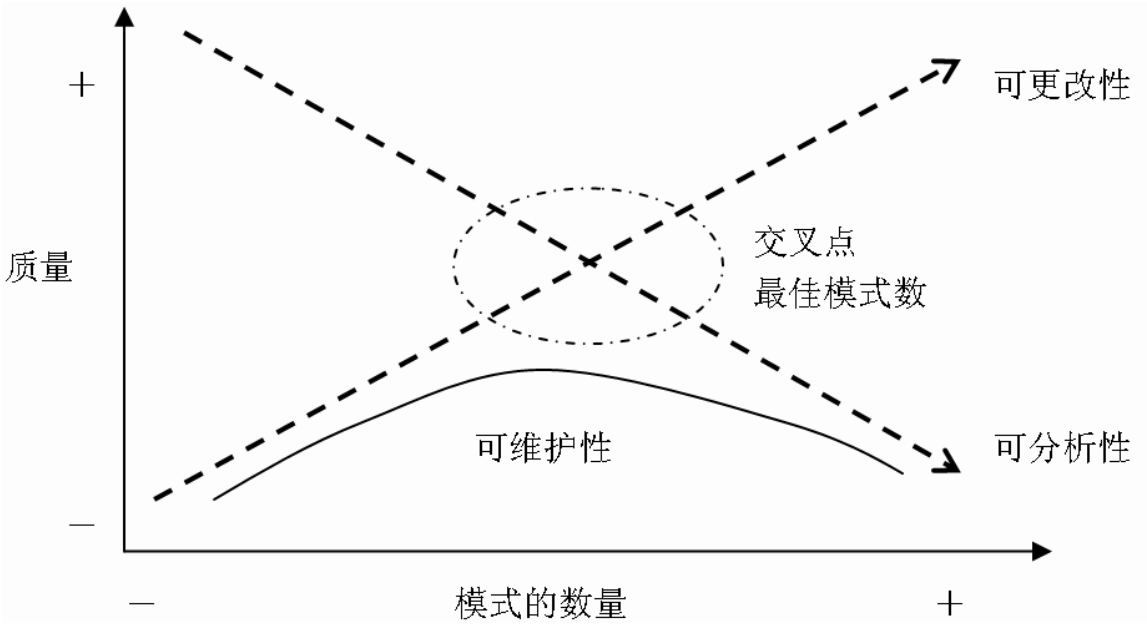


图 1 可分析性和可更改性之间的关系。交叉点是最佳模式数，也是可维护性最好的点

图一显示了使用模式时，可更改性和可分析性之间的关系。该图说明了：

- 如果某设计使用了大量的模式，则该设计具有很好的可更改性
- 如果某设计使用了很少的模式，则该设计具有很好的可分析性

2.1 可分析性相关问题

一般情况下，确定使用哪种模式是一件很困难的事情。本小节将说明如何依据一些原则来选择适当的设计模式。这种方法使我们以一种更简单方式来进行分析，确定使用哪些模式。

Garzás 和 Piattini (2001)说过，OOD 原则是一系列从已有经验中获得的建议或真理，这些经验构成了 OOD 的基础，其目标是控制选择和运用模式的过程。下面列出了其中的一些原则（除了这些原则之外，还有很多，限于篇幅，本文不再列出）：

- 开关原则（**Open-Closed Principle ， OCP**）：一个模型可以有扩展，但不能有更改。
- 替代原则（**Substitution Principle ， SP**）：子类必须能够被他们的基类所代替。
- 依赖倒置原则（**Dependency Inversion Principle ， DIP**）：依赖于抽象，而不要依赖于特殊化
- 接口隔离原则（**Interface Segregation Principle ， ISP**）：多个面向特定用户的接口好于一个通用接口
- 缺省抽象原则（**Default Abstraction Principle ， DAP**）：在接口和实现接口的类之间引入一个抽象类，这个类实现了接口的大部分操作。
- 接口设计原则（**Interface Design Principle ， IDP**）：规划一个接口，而不是实现一个接口。
- 黑盒原则（**Black Box Principle ， BBP**）：多用对象聚合，少用类的继承
- 不要构造具体的超类原则（**Do not Concrete Superclass Principle DCSP**）：避免维护具体的超类

一般而言，为了使 OO 系统达到一定的质量，不应当违反任何原则。另一方面，模式的使用有利于提高设计的效率，但是，原则和模式之间的确切关系大体上是未知的，具体的说，我们不知道哪些原则适用于哪些模式。

因此，举例来说，为了遵守DIP原则，设计可能采取的一个策略是使用抽象工厂模式。但设计中使用其他模式（如原型、工厂方法等）的目的与其说是为了直接遵从一个原则，更不如说是实现抽象工厂模式。从而我们可以得出结论：某些模式会与原则直接关联，而另一些模式则更多的与其他模式而不是原则相联系。因此，模式可根据他们所遵从的原则来进行分类，甚至可以通过原则来创建一种不同于现在的模式目录（很多时候，它们只是简单的按字母顺序排列）。也可以总结一份原则清单，用于评估设计质量，并给我们提供符合设计原则的解决模式。我们可以更进一步把原则和模式之间的关系划分为以下三类：

方式一：	方式二：	方式三：
对于应用某种原则建立起来的模型，模式的使用可以提供更好的解决方案	模式实现了或者包含了某些原则	对于先前已应用了模式的解决方案，可使用原则提升其质量

表一对前面章节列出的各项原则进行分析，以及各项原则和Gamma et al. (1995)提出的各个模式之间的相互关联的不同方式。

下表中各模式按字母顺序排列。这是一种比较客观的排列方式，后面我们将基于原则对它进行分析。

原则	OCP			SP			DIP			ISP			DAP			IDP			BBP			DCSP		
模式	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
抽象工厂		■			■		■	■				■			■		■						■	
适配器 类		■			■			■				■			■		■						■	
适配器 对象		■			■			■				■			■		■			■			■	
桥接		■			■			■				■			■	■	■			■			■	
生成器		■			■		■	■				■			■		■			■			■	
		■			■			■				■			■		■						■	
命令		■			■			■				■			■		■			■			■	
组合		■			■			■				■			■		■			■			■	
装饰		■			■			■				■			■		■			■			■	
外观																								
工厂方法		■			■		■	■				■			■	■	■						■	
享元		■			■			■				■			■		■			■			■	
解释器		■			■			■				■			■		■			■			■	
迭代器		■			■			■				■			■		■						■	
中介者		■			■			■				■			■		■						■	
备忘录																				■				
观察者		■			■			■				■			■		■						■	
原型		■			■		■	■				■			■	■	■						■	
代理		■			■			■				■			■		■			■			■	
单件																								
状态		■			■			■				■			■		■						■	
策略		■			■			■				■			■		■						■	
模板方法		■			■			■				■			■		■						■	
访问者		■			■			■				■			■		■						■	

表1 原则与模式之间的关系

从上表中可以得到模式使用的一些参考因素和研究曲线，下面列出了其中的一些例子：

- 可以将每个模式分为更小的部分，以便研究那些各具特性的模式所共有的元素，我们称之为“模式的模式”或者“元模式”。

- 上表可以指导我们如何运用模式。因为，通过上表可以很容易知道如何正确的使用模式， 而且一旦在设计中运用原则，也容易导致模式的使用。这有利于更好的运用模式。举例来说，使用NSCP原则就意味着要使用创建型模式，这保证了系统是用接口函数写的，而不是实现函数。

- 我们可以对微结构进行正式的研究

- 通过上表，我们可以看出哪些原则和哪些模式是匹配的，是如何匹配的。根据原则对模式的影响方式的不同（方式一、方式二、或方式三），他们的匹配关系也具有不同的特点：

- 可以看出，抽象工厂、生成器、工厂方法和原型模式和所有核心原则匹配，但是单件模式不适应于任何原则。单件模式不是一个微结构（它只描述了一个类）。单件模式涉及对象的生成，但是它生成对象的特性和抽象程度与其他四种创建型模式不同，这是引自Buschmann et al. (1996)的文章，该文章中将单件模式作为一种代码习惯。至于剩下的四个创建型模式，与它们相关的原则是一样的，除了生成器模式外，这是因为生成器模式的构造策略与前三者不同。可以看出，对于模式和原则关系的研究使我们可以对模式进行更细致更基础的分类。

  - 我们考察的模式中，任何具有继承关系的微结构，都至少遵从下列原则：OCP、SP、DIP、IDP和DCSP。

  - 与状态模式、策略模式具有相同结构特征的那些模式都遵从相同的原则，并具有相同的特征

  - 所有行为型模式都具有以下特征：按照方式二使用OCP、SP、DIP、IDP、DCSP原则，按照方式三使用ISP和DAP原则，不实现BBP原则（按照Gamma的书）。

- 我们通过判断是否使用了元模式来寻找和/或验证新的设计模式

通过研究模式建立的基础以及模式与设计的关联方式，上述的原则使我们获得了既好又实用的OO设计经验。

2.2 可更改性相关问题

使模式具有可更改性的因素是所谓的中间层次。Nordberg (2001)说：“许多设计模式的核心是建立服务提供商和服务消费者之间的中间层次。对于对象来说，中间层通常通过抽象接口实现。”不幸的是，任何一个中间层都使软件离真实世界或问题的分析视图更远，建立相对更多的中间类，在设计的可懂度、实现的调试方面产生额外的管理费用。有关前面说的这些因素，可总结出下列几条规律：

- 每次使用模式的时候，设计中至少出现一个中间层，而且这些中间元素或者不在业务逻辑内或者不占主导地位，例如：通知、观察者类、更新方法，等等
- 每次我们添加一个中间层后，软件离设计就又远了一步。添加了中间层或设计类后，设计变得更难懂、更难分析。
- 中间层能够提升设计的可更改性。

3. 最佳模式数据的衡量体系

上面说了这么多，我们仍有一个问题：我们的设计应提供多大的可更改性，而同时又不丧失可分析性呢？建立一套衡量体系来回答这个问题将作出很大的贡献。

定义一个参数，量化地表示设计的可更改性与引入的中间层的关系。

$$可改动数CN＝中间类数ICN \qquad (1)$$

$$Changeability\ Number\ (CN) = Indirection\ Classes\ Number\ (ICN) \qquad (1)$$

另一方面，衡量设计的可分析性的值必须要考虑设计中引入的类的数目：有些是简单类，有些是重用的类（如观察者模式中的目标类），有些是中间类（如观察者模式中的观察者类）。因此：

$$可分析数AN＝域类数DCN－中间类数ICN－简单类数SCN \qquad (2)$$

$$Analyzability\ Number\ (AN) = Domain\ Classes\ Number\ (DCN) – Indirection\ Classes\ Number\ (ICN) – Simplify\ Classes\ Number(SCN) \qquad (2)$$

从公式2可以看出，设计图的可分析性是最高的。进入设计阶段，在设计图上加上中间制品，模型的可分析性下降了。同时可能看出，某些模式对可分析性的影响比其他模式大，这与该模式引入的中间类的多少相关。

现在，我们可以计算最佳模式数如下：

可改动数CN＝可分析数AN (3)

Changeability Number (CN) = Analyzability Number (AN) (3)

中间类数目ICN＝域类数目DCN－中间类数目ICN－简单类数目SCN （4）

Indirection Classes Number (ICN) = Domain Classes Number(DCN) – Indirection Classes Number (ICN) – Simplify ClassesNumber (SCN) （4）

中间类的数目ICN＝[域类数目DCN－简单类数目SCN]/2 （5）

Indirection Classes Number (ICN) = [Domain Classes Number(DCN) – Simplify Classes Number (SCN)] / 2 （5）

在公式（5）中，设计阶段 DCN 参数是一个固定值，剩下的参数依赖于选用哪种模式或者引入多少中间制品。

我们不久后将对此公式进行较大的改进，使该公式能反映出其他因素的影响，如方法的质量。

3.1 例子

下面的例子（图 2）显示了应用的最佳模式数（可更改性的数目 CN 和可分析性数目 AN）。

起初，只有三个领域实体，没有使用模式，也没有中间类，因此 CN=0。由于只有三个域类（Domain Class），所以 AN=0。（中间类和简单类的数目均为 0，只有域类数目不为 0）

然后，引入设计模式（图中的 b）。这个模式具有一个中间类（观察者类），这个中间类引入一个简单类（目标类）。CN=1，AN=3-1-1=1。

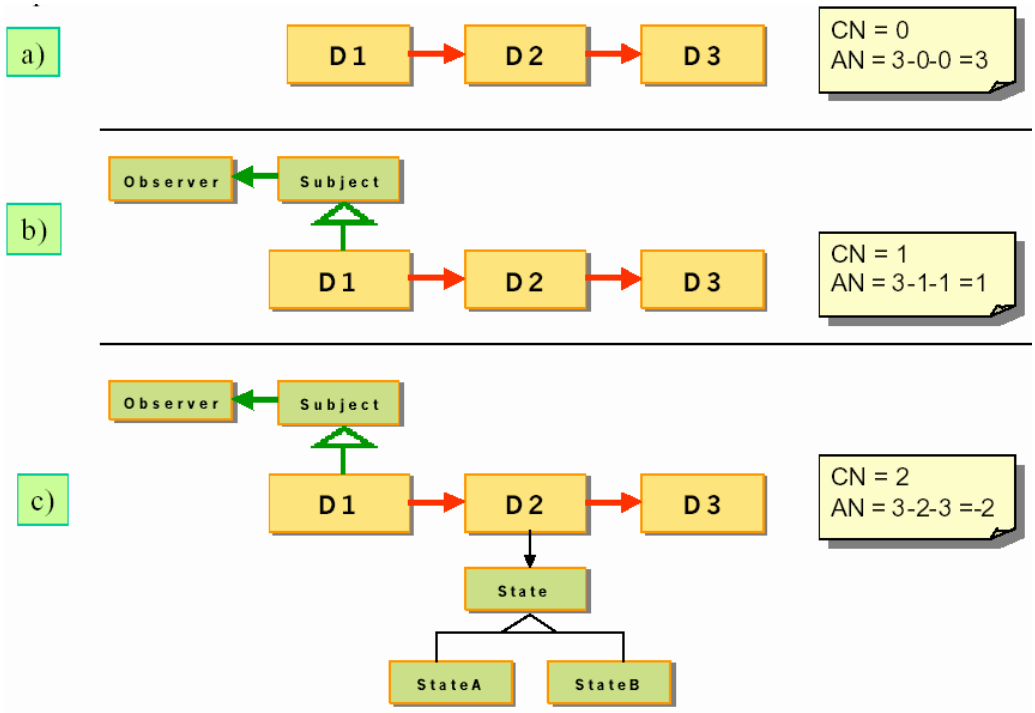


图 2 本例显示了使用模式时 CN 和 AN 的变化

最后，我们引入状态模式，该模式除了增加一个中间类外，针对我们这个例子，还增加了两个简单类（状态 A 和状态 B）。CN=2，AN=3-2-3=-2。

这时，CN 数大于 AN 数。根据前面章节的分析，这时，如果想保持设计的可分析性的话，我们应当引入更多的模式。也许将来确实需要引入更多模式，但是目前这个阶段我们只是提升设计质量，没必要再引入模式。CN 和 AN 之间的关系提供了一种控制模式数量的合理方式。

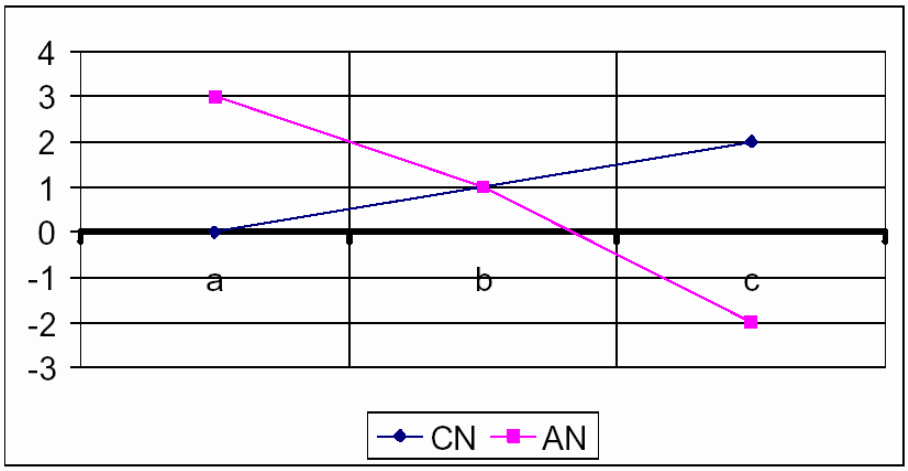


图 3 本图显示了 CN 和 AN 的变化

4 致谢

本研究项目是 CICYT 支持的 DOLMEN 项目的一部分。同时我们也对 ALTRAN SDB 表示感谢，他们一直支持此项研究。

5 结论和以后的工作

专家们总是使用已被证实的观点。近几年，这些观点逐步发展，形成模式的概念。由于模式社区致力于发现、分类、推广各类模式，因此模式的概念得到了广泛的认同和使用。

模式是一种非常有用的设计元素。但是，要想合理运用模式，还需要研究许多其他元素。首先要明确的是质量这个词对于设计模式的含义。用质量这个词描述模式时必须慎重。如果质量的定义包含多个方面，需要选择其中合适的方面。对于维护质量来说，两个合适的方面是可更改性和可分析性。

另一方面，除了模式相关的知识外还存在许多所谓的隐形知识。我们在 OODK 中命名、区别、划分以下几类概念：原则、启发式，模式和重构(Garzás and Piattini, 2001)。但是前面那些元素存在着很大的不确定性。实际上，这些知识元素从来没有被作为一个整体来研究过，没有人研究它的兼容性，也没有人研究建立在这些知识基础上的方法。要系统化地研究这些 OO 设计相关的知识，并把它以一种实用的方式提供给设计者，还有许多工作要做。

6 参考文献

Bennet K. H. and Rajlich V. T. Software Maintenance and Evolution: a Roadmap, in Finkelstein A. (Ed.) The future of Software Engineering, ICSE 2000, June 4-11, imerick, Ireland, pp 75-87.

Brito e Abreu F. and Carapuça R. Object-Oriented Software Engineering: Measuring and controlling the development process. 4th Int Conference on Software Quality, USA, 1994.

Briand L., Morasca S. and Basili V. Defining and Validating Measures for Object-Based high-level design. IEEE Transactions on Software Engineering, 25(5), 722-743, 1999.

Buschmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M., A System of Patterns: Pattern-Oriented Software Architecture, Addison-Wesley, 1996.

Coad P., “Object-Oriented Patterns”, Comm. ACM, Vol. 35, No 9, Sep. 1992, pp. 152-159. Fowler M. Analysis Patterns. Addison Wesley, 1996.

Gamma E, Helm R, Johnson R and Vlissides J. Design patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1995.

Garzás J., Piattini M. Principles and Patterns in the Object Oriented Design, OOPSLA 2001 -Workshop “Beyond Design: Patterns (mis) used”. Octubre 14-18, 2001. Tampa Bay, Florida, USA.

Genero M., Piattini M. and Calero, C. Early Measures For UML class diagrams. L´Objet. 6(4), Hermes Science Publications, 489-515, 2000.

Henderson-Sellers B. Object-oriented Metrics - Measures of complexity. Prentice-Hall, Upper Saddle River, New Jersey, 1996.

Nordberg M. E. Aspect-Oriented Indirection – Beyond OO Design Patterns. OOPSLA 2001 -Workshop “Beyond Design: Patterns (mis)used”. Octubre 14-18, 2001. Tampa Bay, Florida, USA.

Prechelt L., Unger B., Tichy W. Bossler P. A controlled Experiments in Maintenance Comparing Design Patterns to Simpler Solutions. IEEE Transactions on Software Engineering, September 2000.

Pigoski, T. M. Practical Software Maintenance. Best Practices for Managing your Investements. Ed. John Wiley & Sons, USA, 1997.

Reibing R. The impact of Pattern Use on Design Quality. OOPSLA 2001 – Workshop “Beyond Design: Patterns (mis)used”. Octubre 14-18, 2001. Tampa Bay, Florida, USA.

Rising L., The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, 1998.

Schmidt D. C., Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software, Communications of the ACM 38,10, October 1995, pp 65-74.

Wendorff P., “Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project”, Procedings of the Fifth European Conference on Software Maintenance and Reengineering , CSMR 2001, IEEE Computer Society.

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

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

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

UML 之“四书五经”

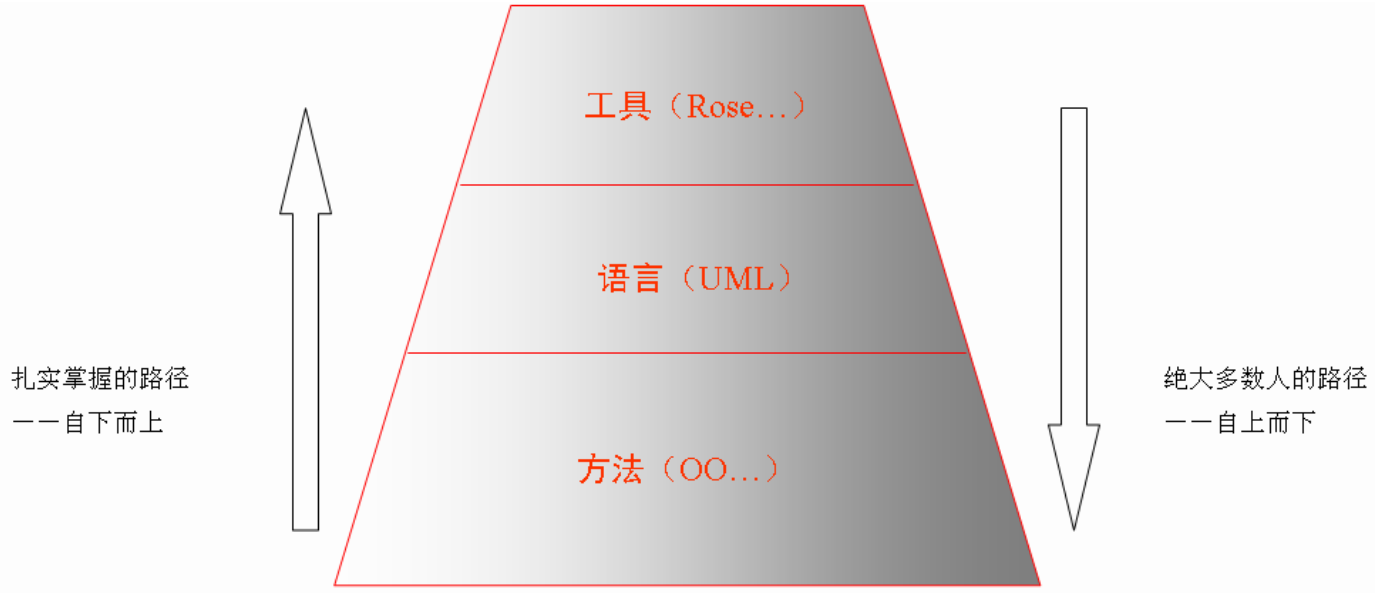
think 著

讨论

正如李维先生所说，IDE 的发展趋势，使得开发人员已经不能满足于掌握一门语言或 API，UML、软件过程、模式等将成为必备知识。

怎样来掌握 UML，真正把它用到开发中去？开发人员缺乏有效的指引。不少人抱着《UML 参考手册》、《UML 用户指南》作为入门读物，结果半天不得其门而入。软件以用为本，软件技术也要以用为本。用不上的技术，不要怪开发人员不开窍。书也是一样。不能真正帮助开发人员的书，再说好也意义不大。

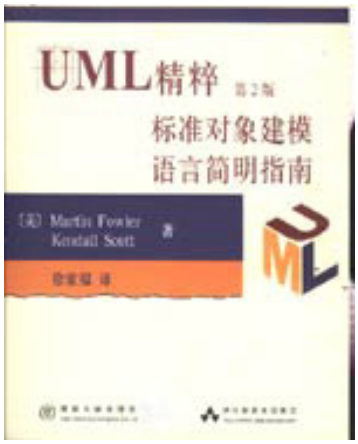
我们说掌握 UML，不是只为了掌握符号本身。正如一个积分符号背后隐藏的数学知识体系一样， UML 涉及的知识体系可以大致表述如下：



上层以下层为基础。上层出现问题，往往根源在于下层没有掌握好。从道理上说，似乎应该自下而上的学习，先打好基础再向上走。但笔者不赞同现实中采取这样的学习路径，我们的绝大多数人也不会采取这样的学习路径。很多人学习 UML 是先找到一个工具，先画几个小人圈圈再说，并不是先去打什么“基础”。所以笔者的建议是一切从实战出发，带着问题去求索，才能激发人的学习动力，学到的才是真东西。

UML 相关的书有很多，在 dearbook 或 china-pub 查一下 UML 就知道了，国内出版的已经超过 60 本，在加上面向对象分析设计，统一过程之类的书名，上百本是没问题的。笔者恰好以 UML 为业，这些书绝大多数都买了。下面根据笔者的理解，按照一个实用的学习顺序，挑选出下面几本书。

A. 《UML 精粹——标准对象建模语言简明指南（第 2 版）》，Martin Fowler, Kendall Scott，清华大学出版社，2002。



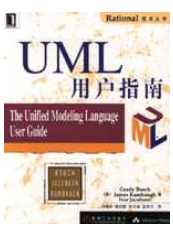
对于刚听说“UML”这个字眼的开发者，面对复杂丰富的规范，经常会不知从何处下手了解。Martin Fowler 和 Kendall Scott 的这本小书，就是开发者了解 UML 的最佳读物。正如书名“精粹”二字所言，书中提炼出了 UML 最重要的元素，用作者一贯亲切轻快的文笔，使得那些稍显抽象复杂的概念显得尽量平易。各章后面还设了“何处寻找更多资料”的段落，引导读者进一步探索。

这本书在国外不断重印，也说明开发者用钞票投票，使它战胜了其他类似的书。

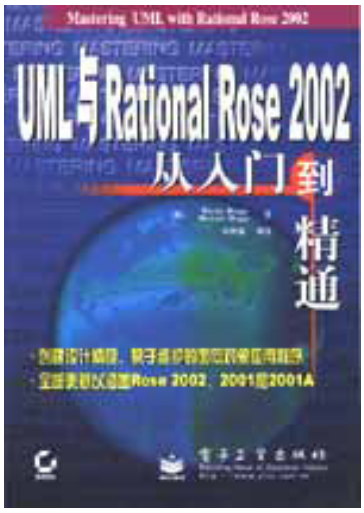
本书第 2 版中译本很多术语的译法，在开发人员圈子中受到很多争议。此书英文版第 3 版已经有了，希望在国内能尽快有中译本。

类似的书：

《UML 用户指南》，Grady Booch, James Rumbaugh, Ivar Jacobson，机械工业出版社，2001。偏重理论，适合想深入研究有关概念的读者。



B. 《UML 与 Rational Rose 2002 从入门到精通》，Wendy Boggs, Michael Boggs，电子工业出版社，2002。

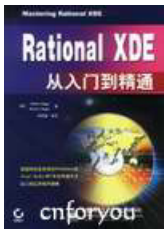


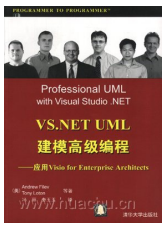
什么？推荐这本“烂书”？是的。这本书从原书内容到中文翻译，被人骂得太多了。问题在于刚听说 UML、Rose 等新字眼时，你以为人人都有耐心去“打基础”？不是，这个时候最想的就是先动手试试再说。不管怎么说，这本书至少系统地介绍了 Rational Rose 的使用，看着书照做，至少可以画出图来，坚定下一步的信心。虽然书中的 UML 概念有的地方值得商榷，但是类似的书很少，它算是最不差的，你让读者怎么办？估计作者还会紧跟工具版本更新，不断推出此书的新版本，希望一版更比一版好。

类似的书：

《Rational XDE 从入门到精通》，Wendy Boggs, Michael Boggs，电子工业出版社，2003。

《VS.NET UML 建模高级编程——应用 Visio for Enterprise Architects》，Andrew Filev, Tony Loton，清华大学出版社，2003。





C. 《UML 和模式应用：面向对象分析与设计导论》，Craig Larman，机械工业出版社，2002

好，基本了解使用一种 UML 工具了（也许你不用 Rose，甚至不用电子化工具，用免费的纸和笔？）。现在我们要来从头到尾开发一个软件，在各个开发工作流怎样用上适当的 UML 元素呢？

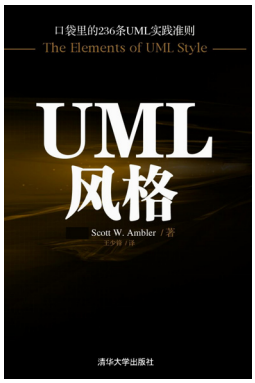
Craig Larman 的这本书通过一个 POS 机的实例，循序渐进地介绍了使用 UML 进行需求、分析、设计和实现的全过程，并阐释了迭代开发的要义。此书被全球诸多培训咨询机构选用作为 OOAD 课程的教材。机械工业出版社 2002 年发行的中译本是第 1 版，笔者推荐读者争取去看第 2 版（中译本即将由机械工业出版社发行）。相对第 1 版而言，Larman 在第 2 版中选择并接受 UP（统一过程）作为开发流程，遵循用例驱动、架构为中心、迭代和增量的原则，向读者展示了一幅全景画面。

类似的书：

《面向对象软件开发教程（原书第 2 版）》， Scott W. Ambler，机械工业出版社，2003。其实就是那本著名的 Object Primer，可惜中文名起得象某个大学教授为评职称编写的教材。本书英文第 3 版已出，并加了副标题：Agile Model Driven Development with UML 2。期待！

《UML 用例驱动对象建模：一种实践方法》，Doug Rosenberg, Kendall Scott，清华大学出版社 ，2003。也是通过一个案例介绍使用 UML 的 ICONIX 过程。这本书比上面两本都薄，而且通俗易懂。

D. 《UML 风格》，Scott W. Ambler，清华大学出版社，2004。



这是一本特别的小册子。它不讲概念，它假设读者已经懂了概念。它简单地告诉读者“在递归关联上指明角色名”，不详细解释什么叫“递归”、“关联”和“角色”。概念可以去看上面推荐的 A 一族。它不讲工具，它假设读者已经了解某种工具。工具可以去看上面推荐的 B 一族。它不讲过程，它假设读者已经了解某种开发过程。过程可以去看上面推荐的 C 一族。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。

类似的书：

尚无。至少笔者还未发现。

E. 其他……

针对 UML 的各种元素，可能都会有一些书籍着重阐述。笔者以各种 UML 图强行分类（虽然比较别扭），简要推荐如下：

用例：《编写有效用例》（机工）、《有效用例模式》（清华）

类/对象/顺序/协作图：《分析模式》（机工）、《Oracle 8 UML 对象建模设计》（机工）、《设计模式》（机工）、《面向对象软件设计经典》（电子）、《敏捷软件开发：原则、模式与实践》（清华）

构件/部署图：《企业应用架构模式》（机工）、《面向模式的软件体系结构》（机工）

状态图：《实时 UML——开发嵌入式系统高效对象》（电力）

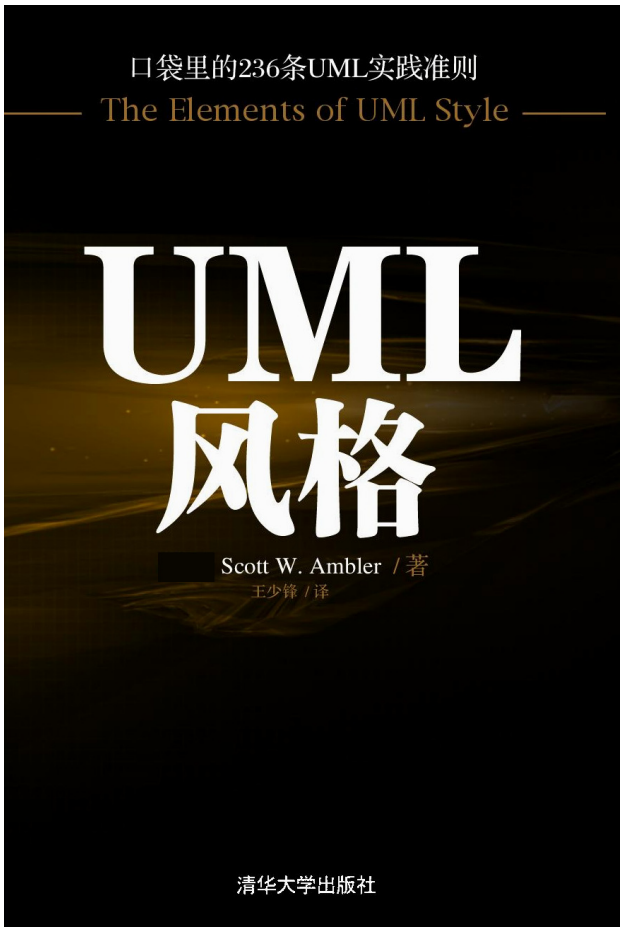
活动图：目前尚未有好书

这篇文章只是在您茫然不知何处着手的时候给您一些建议，如果您正在看某一本 UML 书，觉得对您很有帮助，尽管去看完，不要管上面有没有提到它。也许您目前的工作只要求您了解 UML 的某种元素，例如您在做实时嵌入应用系统，可能只想引进状态图来帮助提升系统的可靠性，您可以看《实时 UML》（电力）；或者您只是想使用 UML 帮助理清组织的业务，然后把整个软件开发外包，你可以看《UML 业务建模》（机工）。总之，一切从您的现实出发，别把我的这些话太当回事。

本文最初发表于《程序员》2004.06 期，经《程序员》同意刊登。

征稿

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


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


mobile

《UML 风格》读者意见反馈

填写反馈信息

获赠 UMLChina 编著的《软件以用为本——UML 实作细节》

亲爱的读者：

感谢您阅读本书，也感谢您一直以来对清华版计算机图书的支持和爱护。为了今后为您提供更优秀的图书，请抽出宝贵的时间来填写下面的意见反馈表，以便于我们对本书做进一步的改进。同时，**只要您填写并提交下面的表格，即可免费获赠由 UMLChina 编著的《软件以用为本——UML 实作细节》，该手册专为《UML 风格》的读者定制，它图文并茂地描绘了应用 UML 的整个过程，点出了过程中的关键要点，非常适合与《UML 风格》配合阅读！**

请访问 UMLChina（http://www.umlchina.com）了解详细的活动细则。同时，如果您在使用本书的过程中遇到了问题，或者有好的建议，也请来信告诉我们。

地址：北京市海淀区双清路学研大厦 A 座 517（100084） 市场部收

电话：62770175-3506

电子邮件：jsjjc@tup.tsinghua.edu.cn

个人资料

姓名：_____ 年龄：_____ 所在单位：_____

文化程度：_____ 通信地址：_____

联系电话：_____ 电子信箱：_____

您使用本书是作为：☐选用教材 ☐参考读物

您对本书装帧设计的满意度：

☐很满意 ☐满意 ☐一般 ☐不满意 改进建议_____

您对本书印刷质量的满意度：

☐很满意 ☐满意 ☐一般 ☐不满意 改进建议_____

您对本书翻译质量的满意度：

☐很满意 ☐满意 ☐一般 ☐不满意 改进建议_____

您对本书的总体满意度：

从语言质量角度看 ☐很满意 ☐满意 ☐一般 ☐不满意

从科技含量角度看 ☐很满意 ☐满意 ☐一般 ☐不满意

您认为是哪些原因让您购买本书？（可附页）

_____

_____

您认为本书在哪些地方应进行修改？（可附页）

_____

_____

您希望本书在哪些方面进行改进？（可附页）

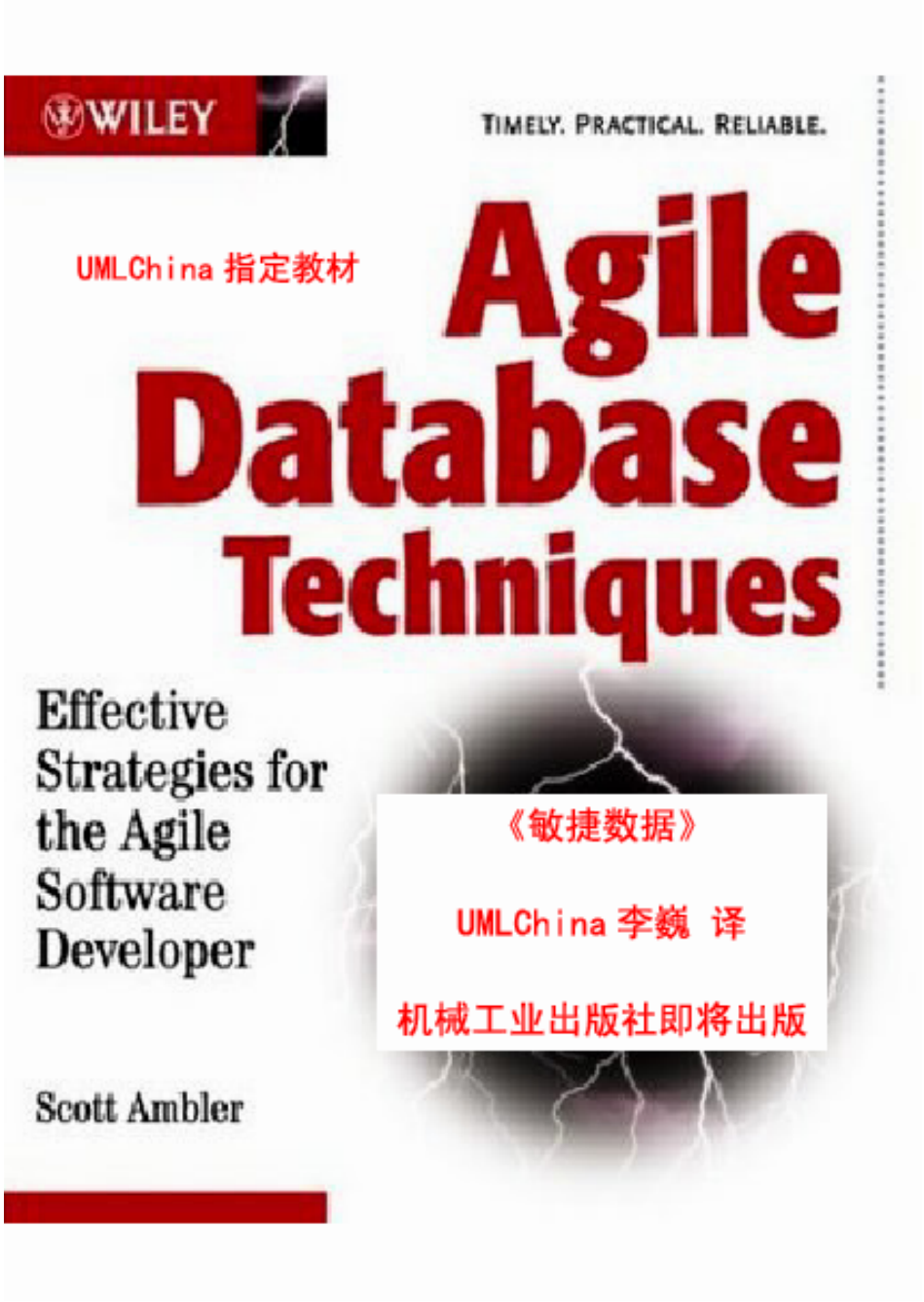
_____

_____

《敏捷数据》中译本样章（草稿）

Scott Ambler 著，UMLChina 李巍 译

讨论



第一章 敏捷数据方法

可以采用一种面向数据开发的敏捷方式。

第一步是要选择这样的方式进行工作。

数据无疑是基于软件的系统的一个重要方面——这为我们近十年来所共知，然而许多组织机构依然在与他们解决「其软件过程中的面向数据问题」的方法作斗争。

敏捷数据（AD）方法的目标是定义「使 IT 专业人员在软件系统的数据方面能够相互有效工作」的策略。这并不是说 AD 是一种“放之四海皆准”方法论。相反，可以将 AD 看作一组观念集，其将使你们组织机构内的软件开发者在基于软件的系统的数据方面能有效地一起工作。尽管本书着重于敏捷软件开发中那些经证实的技术，但定义一种基本的方法基石是必不可少的。

在本章中，我会通过考察以下主题来帮助你们理解 AD 方法：

- 缘何当前难以相互合作
- 敏捷运动
- 敏捷数据的哲学
- 敏捷数据概述
- 敏捷数据能够解决我们的问题吗？

缘何当前难以相互合作

在许多组织机构中，数据专业人员和开发人员之间的关系常常不那么理想。是的，在有些组织机构中这两个群体能合作得比较好，但总是会有一些紧张关系－团队间存在的某些健康的紧张关系可使你们的组织机构从中受益，然而当紧张关系变得有害时，这些分歧便往往会导致冲突。数据专业人员和开发人员必须要克服的挑战包括：

不同的愿景和优先考虑。开发人员通常专注于一个单独项目的具体需求，并且往往致力于完成尽可能多的与组织机构其他部分相互独立的工作。数据库管理员（DBA）则专注于他们所负责的数据库，往往是通过最大限度地减少对数据库的改动来对其进行“保护”。数据库管理员和数据架构设计师专注于整个企业的数据需求，有时候会将项目团队的直接需求实际排除在外。毫无疑问，每个团体的边界不同，它们所优先考虑的事情不同，以及团体所处理的问题不同。更糟的是，你们的项目涉众（stakeholder），从直接使用者到高层管理人员，都具有不同的

优先考虑和愿景（vision）。

角色过于专业化。专家（specialists）所关注的往往过于狭窄；他们非常努力地去掌握软件开发那一小块部分需要了解的所有事情，而忽视了其他的东西。例如，经常会发现高级 Java 开发人员从来都没有听说过数据规范化（data normalization，在第 4 章中予以论述），或者甚至不理解为何你要去做这样一件事情，以及数据架构设计师不能看懂统一建模语言（UML，Unified Modeling Language）的状态图（state chart diagram，在第 2 章中予以论述）。由于这些角色过于专业化，扮演这些角色的人常常很难与其他人产生联系。该问题的另一端就是通才（generalists），他们能够了解整个全景却无法给开发团队提供任何具体的技能。我们需要找出这两种极端情况之间的最佳点（sweet spot）。*敏捷建模*（Ambler 2002a）的基本观念是软件开发人员应该能全面地了解整个软件过程*并且*对一个和多个领域有专攻。由于敏捷建模者是通才，他们所了解的与“软件游戏”相关的问题范围很广，而且还会为他们的团队提供具体而有价值的技能。

过程阻抗失配。诸如统一过程（Unified Process，Kruchten 2000；Ambler 2001b）、极限编程（XP）（Beck 2000）、Scrum（Beedle 和 Schwaber 2001），DSDM（Stapleton 1997），Crystal Clear（Cockburn 2001b）特性驱动开发（FDD）（Palmer 和 Felsing 2002）以及敏捷建模（AM）这些过程所具有的一个为数不多的相同之处是，它们都是以一种演化（迭代和增量）的方式来进行工作的。不幸的是，许多数据社区内的人们仍然把软件开发看作是一种串行或接近串行的过程。无疑，这里存在着阻抗失配（impedance mismatch），这表明数据社群需要重新思考他们的方法。你们将会在本书的第 II 部分看到针对数据所可能采用的一种演化式的方法，这种变化需要进行文化和组织机构的调整方能成功。

技术阻抗失配。开发人员与对象和组件一起工作，而数据专业人员则是与数据库和文件打交道。软件工程的原则形成了对象和组件的底层基础范型（paradigm），而集合论（set theory）则形成了关系型数据库（目前最为流行的数据库技术）的底层基础范型。由于基本范型不同，这些技术无法一起完美地合作，并且存在阻抗失配。这种阻抗是能够克服的，尽管这样做需要非常强的技能组（skillset）（第七章将会涉及到这个主题）。

管理守旧。软件开发人员所使用的技术和方法日新月异，这是我们都非常清楚的事实。随着人们在公司阶层上的发展，他们需要处理更少的技术问题和更多的人际问题，最终便会导致许多管理者失去他们的技术优势。这意味着管理人员先前的开发经验－他们基于这些经验来做出技术上的决定－将不再适用。这种情形曾在我们从过程化转向面向对象技术时曾经经历过－那些可能对于一个 COBOL 项目而言比较好的决定常常会被证明是一个 Java 项目的死亡之吻。随着转向敏捷软件过程，我们清楚地看到这种问题一再发生。管理需要与时俱进。

组织机构上的挑战。一些常见问题，如个体和群体之间糟糕的交流或政策，会对软件开发的数据方面造成伤害，就像它们伤害其他成果一样糟糕，它妨碍了每个人在一起有效地工作。

糟糕的文档。大多数文档看起来会是下列极端情形的一种：文档很少，或者没有文档，或者文档过于复杂以至于没有人能够读懂。相互认同的开发标准和指导原则、遗留系统文档、，遗留数据库文档以及企业模型，在编写良好的时候才是有价值的资源。在第十章中会介绍编写文档的敏捷策略。

低效的架构工作。涉及到企业的架构，大多数的组织机构会面临许多意义重大的挑战，其中最常见的就是他们不知道「从何处开始」。产生偏差的企业架构会过于专注于企业的一个视角，从而导致架构无法充分地涉及到组织机构的真正需求。正如 Zachman 框架（ZIFA 2002；Hay 2003）所指出的那样，你需要考虑许多潜在的视角。这些视角有数据/结构，功能/过程，网络，人员，时间和动机。象牙塔架构－项目团队会在日复一日的现实工作中将他们用公式表示出来的这些东西自己清除掉－在纸上看着很好，但不幸的是会在实践中失败。此外，开发人员需要承认「他们的成果必须要反映与符合其组织机构环境强加在他们上面的约束」。

低效的开发指导原则。许多组织机构致力于提出一组所有软件开发人员工作所需要的开发指导集。这样做的原因非常之多，包括人们不理解遵循这样一些指导原则的需求，人们不愿意遵循其他人的指导原则，指导原则过于复杂，指导原则过于简单，“一概而论”的态度会导致指导原则无法适用于一个具体的平台，以及不愿意随着时间的变化来改进指导原则。如果你们当前有一组有效的指导原则集可用，并且（这是关键）*每个人都能正确地理解和应用它们*，那么便会显著地提高你们软件开发工作的效率。

低效的建模成果。这往往是前面所提到的几个问题的结果。专注于一个具体开发方面的人所提出模型通常将会很好地反映那个狭小视角的优先考虑，但是却无法考虑到现实的其他视角。企业数据模型可以展现出一个组织机构所需数据的极佳愿景，但是反映一个组织机构的数据、功能、用途和技术需求的企业模型则往往更为有用。UML 的类图（class diagram）可以反映出一个单独项目的需求，但是如果它无法反映其所访问的遗留数据源的状况，那么其在实践中价值甚微。通常，建模人员和软件开发人员需要在一起工作并得到一个真正有效的全景。

发现你的问题

对于组织机构而言，他们很容易拒绝承认自己遇到了问题。高层管理者很难在一开始便能觉察出问题所在，因为他们需要听取的坏消息早在到达他们之前便已被过滤掉。同样地，组织机构中其他地方的人们很难觉察问题－－在他们看来可能一切都还相当不错－－而不幸的是，他们用来判定局势的价值系统并不理想，使得他们对其所引发的问题视而不见。

作为一名顾问，我具有为各种组织机构工作的特殊条件，并且在我看来在十个组织机构中，大约会有一个进行面向数据活动的方式相当成功，十个之中大约有六个认为「他们做得很好而事实却并非如此」，而十个里面的其余三个则知道他们遇到了问题，却不清楚该为此做些什么。事情并不是非要到如此境地。

因此你如何才能知道自己碰上了问题？企业数据专业人员——包括数据架构设计师和数据管理员——将会因为项目团队的项目开发人员忽视他们的建议、标准、指导原则和企业模型而感到灰心丧气。更为糟糕的是，应用开发者通常甚至不会首先去了解这些人员和事情。开发人员将会因为「企业数据专业人员在做出或认定一些看上去非常简单的改变时缓慢的节奏」而感到沮丧（这通常是合理的）。DBA 常常发现自己陷入到这两个派别的纷争之中，在努力保持和睦相处的同时，尽力完成他们的工作。如果在你们的组织机构中经常出现一个或多个这样的问题，则意味着你们遇到了麻烦。

下面是一个潜在症状的列表，其表示了你的组织机构所面临的一个或多个敏捷数据方法可以帮你解决的挑战：

- 人们对工作的挫折感非常强，或者因此需要一个或多个团体
- 软件无法进行开发，或是其花费的时间过长或代价过于昂贵
- 你会听到诸如“数据管理员在阻碍着进程”或“开发人员没有遵守共同的指导原则”这样的指指点点，更为糟糕的是，指指点点的人往往不会意识到他或她也是问题的一部分。
- 制度上的问题（Political issues）会比「一起开发、维护和支持基于软件的系统」享有更高的优先级。
- 在人员和团体之间存在着旷日持久的斗争。那些以“你总是”和“你从来都不”开头的句子便是这个问题的良好佐证。
- 你们组织机构中众所周知问题并未得以解决。此外，有关改进的建议往往会被忽略，什么都不做，以及不会提供任何原因就加以拒绝。
- 人们的工作时间过长，而报酬太少或没有。
- 似乎是以一种武断和自大的方式来做出影响团队——特别是项目团队——的决定。

我们需要找到一种能够在一起有效工作的方式。数据和开发阵营之间以及项目和企业阵营之间存在着明显的不同。我们所谈论的不同阵营的事情也是问题的一部分，可被论证成为一个根本原因。有一个基本决定需要你来做：你们应该将这些不同点用来作为一种托词以加剧组织机构中现存的问题呢，还是应该沉迷于这些不同点并找到一种利用它们的方式呢？我倾向于后者。我的经验是敏捷运动（agile movement）的价值和原则形成了一种「相互合作的有效方式」的基础。

敏捷运动

为了解决软件开发者所面临的挑战，一个最初由 17 位方法学家所组成的团体建立了敏捷软件开发联盟（Agile Software Development Alliance, www.agileallcance.org），通常简称为敏捷联盟（Agile Alliance）。关于该团体的一件有趣的事是「所有成员皆来自不同的背景」，然而他们还能就一些方法学家往往无法意见相同的问题达成一致（Fowler 2001a）。这个团体的人们制定了一个「用以促进更好的软件开发方式」的宣言，然后基于这个宣言，正式确立了一组「定义敏捷软件开发过程（如 AM）的标准」的原则集。

敏捷软件开发宣言

该宣言（Agile Alliance 200la）被定义成四个简短的价值观语句，需要理解的一个重要思想是「尽管你应该重视右侧的概念，但是你甚至更应该去重视*左侧*的事物」。该宣言的一个好的思考方式是「它定义了优先项（preferences），而非可选项（alternatives）」，其鼓励人们去重点专注某些领域，但并不是要忽略其它的方面。敏捷联盟的价值观（value）如下：

***个体和交互*胜过*过程和工具*。**团体人员负责构建软件系统，而且做的是需要他们一起有效工作的事情——团体的组成并不仅限于程序员，还包括测试人员、项目经理、建模人员和客户。你认为下面哪些人会开发出更好的系统：是五个软件开发者在一个单独房间内使用他们自己的工具在一起工作；还是五个低熟练度的“hamburger flipper”使用定制良好的过程、当前最复杂的工具，以及金钱所能买来的最好的办公室？如果项目非常复杂，我的金钱将会花在软件开发人员身上，你们的呢？这里是指你需要考虑的最重要的因素是「人员以及他们如何在一起工作」；如果你不能正确地理会这一点，那么最好的工具和过程也将无济于事。工具和过程是很重要，不会使我犯错，但仅此而已，它们不会像「有效地在一起工作」那么重要。记住那句古老的格言，*拥有工具的傻瓜依然是傻瓜*。管理者可能很难接受这一点，因为他们往往认为人员和时间，或者人和月，是可以相互替代的（Brooks 1995）。

***可以工作的软件*胜过*面面俱到的文档*。**当你询问一个用户他或她是需要一份长达五十页的文档来描述你所构建的是什么呢，还是实际软件本身，你认为他们会挑选哪一个？我猜在 100 次中会有 99 次用户将选择「可以工作的软件」，当然这里假设他或她认为你能够实际完成交付。倘若真是这样，更有的意义的工作难道不是快速和经常地产出软件，给予你们用户他们更想要的东西吗？此外，我想用户能够非常容易地理解任何你所产出的软件，而不是那些用来描述其内部的工作原理，或描述其用途纲要的复杂的技术图表。文档有其自身的用处，如果编写合理，它是一份非常有价值的指南，可以指导人们去理解「系统是如何构建的和为什么要这样构建，以及如何与系统一起工作」。然而，永远不要忘记软件开发的首要目标是创建软件，而非文档——否则，将会称之为文档开发，难道不是么？

客户合作胜过合同谈判。只有你的客户才能告诉你他们想要什么。No，他们很可能不具有对系统完全进行阐述的技能。No，他们开始很可能会理解错误。是的，他们往往很可能会改变自己的想法。与你的客户在一起工作是很困难的，但工作的现实便是如此。与你门客户签署一份合同固然重要，但是合同并不能成为交流的替代品，虽然对每个人权利和义务责的理解可能会形成合同的基础。成功的开发人员会与他们的客户一起密切地工作，他们会花力气去探寻「他们的客户需要的是什么」，并且他们会培养他们的客户与其一起前行。

响应变化胜过遵循计划。出于各种原因，人们会改变自己优先考虑的事情。随着在你们系统上面的工作不断进展，你们的项目涉众会在理解「问题域（problem domain）和你们所创建的系统」上面产生变化。业务环境会发生变化。技术随时会发生变化，而且并*不*总是会变得更好。变化是软件开发的现实，你们的软件过程必须要对此有所反映。制订一个项目计划并没有错；事实上，我将会为任何没有计划的项目担忧，但是项目计划必须具有可塑性；也就是说，必须能够在你们情况发生变化时留有改变的空间，否则你们的计划将很快会变得无关痛痒。

关于这些价值观语句的有趣之处是，几乎每个人都会很快地去认同它们，然而却很少有人坚持将其付诸实施。高层管理者总是宣称他们的员工是组织机构中最为重要的方面，但是他们却常常遵循符合 ISO-9000 的过程，并将他们的人员作为可替换性的资产。甚至更为糟糕的是，管理者经常会拒绝提供足够的资源以执行他们坚持要项目团队遵循的过程——底线（bottom line）。管理者需要吃他们自己的狗食（eat its own dog food.，意为“即使再糟糕也要身体力行”）。每个人都会欣然同意「软件开发的首要目标是创建软件」，但是仍然会有许多人坚持要把时间花在生成那些描述「软件是什么以及如何对其进行构建」的文档，而不是卷起袖子去实际构建它。你知道——人们往往是说一套做一套。现在必须要停止这么做。敏捷建模者会做他们所说的，说他们所做的。

敏捷软件开发原则

为了有助于进行敏捷软件开发的定义，敏捷联盟成员把在他们声明中捕获的哲学观加以提炼，提出一个包括敏捷数据（AD）在内的方法学所应遵循的 12 项原则集（Agile Alliance 2001b）。这些原则是：

- 我们最优先要做的是尽早地、持续地交付有价值的软件，以使客户满意。
- 即便是到了开发的后期，也欢迎改变需求。敏捷过程利用变化为客户创造竞争优势。
- 经常性地交付可以工作的软件，交付的间隔可以从几个星期到几个月，交付时间的间隔越短越好。
- 在整个项目开发期间，业务人员和开发人员必须天天在一起工作。
- 围绕被激励起来的个体构建项目。为他们提供所需要的环境和支持，并且信任他们能够完成工作。

- 在开发团队内部，最具有效果和富于效率的传递信息的方式，就是面对面的交谈。
- 可以工作的软件是首要的进度度量标准。
- 敏捷过程提倡持续性的开发速度。责任人、开发者和用户应该能够保持一个长期而恒定的开发节奏。
- 不断地去关注技术上的完美和良好的设计来增强敏捷能力。
- 简单——是将未完成的工作最大化的艺术——是根本的。
- 最好的架构、需求和设计出自组织机构本身的团队。
- 每隔一定时间，团队会在「如何才能更有效地工作」方面进行反省，然后相应地优化和调整自己的行为。

稍停片刻并想想这些原则。这是不是你们软件项目实际工作的方式呢？这是否就是你所认为的、项目应该的工作方式呢？再读一遍这些原则。他们是不是一些人所宣称的激进的、不可能实现的目标呢？它们是不是一些诸如妈妈和苹果派那样毫无意义的话语呢？或者它们仅仅是些常识而已？我相信这些原则构成了在成功的软件开发工作中你们可以依赖的常识基础，一个用来指导软件开发者进行面向数据工作的基础。

敏捷数据的哲学观

首先，敏捷数据方法赞同敏捷联盟的价值观和原则。尽管这些建议是一个非常好的起点，但仍然需要对这些哲学观进行扩展，以反映数据专业人员所面对的现实。敏捷数据的哲学观为：

数据。 数据是基于软件的系统的多个重要方面之一。毕竟，大多数（如果不是所有的话）应用都是基于对某种类型的数据的迁移、利用，或者其他方式的操纵。

企业问题。 开发团队必须要适当考虑和解决相关的企业问题。通过符合通用的企业架构（或者至少在将来会达成一致的架构），通过遵循通用的开发标准，以及通过尽可能地重用现有的遗留资源，他们的应用必须能够适合更大的行事规划（schema of thing）。

企业团体。 正是由于企业团队的存在，才得以孕育企业的财产并支持着你们组织机构内的其他团体，如开发团队。这些企业团体应该以「一种能够反映他们客户预期的敏捷方式」和「他们客户的工作方式」来行事。

每个项目皆有所不同。 每个开发项目皆有所不同，需要用一种灵活的方法进行裁剪以满足其要求。一种软件过程并不能适用所有情况，因此要考虑到有关「数据基于问题特征变化」的重要性。

团队合作。 软件开发者必须有效地在一起工作，积极努力地克服那些为此制造困难的挑战。

最佳点。 你应该积极努力地去找到任何问题的“最佳点”，避免陷入黑和白这两种极端情况，从而找到对你们整个情形而言效果最佳的灰度。

有趣的是，大多数的这些哲学观并非特定于数据；相反，它们往往适用于所有的软件开发工作。正如第一个原则所揭示的那样，你需要着眼于整个图景而不仅仅是数据；因此，特定于数据的原则很可能不是非常地适合你们，这是旁道邪说否？No。只是常识而已。

敏捷数据概述

理解 AD 方法的最佳方式是研究它的四种角色——敏捷 DBA（agile DBA）、应用开发者（application developer）、企业管理者（enterprise administrator）和企业架构设计师（enterprise architect）——以及他们是如何彼此交互的。前两种角色为项目级别的开发角色，作为本书关注的重点。后两种角色是企业级别的支持角色，由于他们的重要性，在书中会将其放在显著的地方加以描述。敏捷软件开发者可以担当一种或多种这些角色，尽管他或她很可能会专注一种或两种项目级别的角色。

让我们更为详细地研究每种角色。

敏捷 DBA

*敏捷*DBA（schuh 2001）是指任何积极参与一个或多个应用的数据方面的创建和演变的人。该角色的职责包括与“传统角色”——数据库程序员、数据库管理员（DBA）、数据测试员、数据建模员、业务分析员、项目经理和部署工程师——相关的典型职责，但并非仅限于此。小型组织机构中的 DBA 往往会发现他们或她们自己处于这样一种角色：某种“对数据杂而不精的人”。

敏捷 DBA 的首要客户是应用开发者，尽管他们与企业管理员和企业架构设计师这两者的关系也很密切。在敏捷 DBA 被要求向业务群体提供支持时，他们的首要客户也会包括直接的最终用户以及他们的管理者。尤其是当敏捷 DBA 在支持那些以数据为中心的应用（特别是报表应用）时更是如此。

敏捷 DBA 将会与应用开发者的工作密切相关，这种情况往往是对一个单独的大团队或几个小团队提供支持。敏捷 DBA 可能通常会负责多个数据源（例如数据库，文件，XML 结构等等），或者至少是共同对它们负责。譬如，如果两个开发团队访问同一个数据库而且每个团队都拥有自己的敏捷 DBA，那么这两个人将需要在一起工作以随时地改进这个数据库。这一点与 Schuh 最初的敏捷 DBA 的构想稍有不同——他关注的是一个 DBA 如何才能有效地在一个单独的团队上发挥作用，而 AD 方法则着眼于整个企业。重要的是你要以一种适合你们环境的方式工作。

传统 DBA 向敏捷 DBA 转变的最大的潜在变化是，他们需要学会以一种不断演进的方式进行工作。当今的开发过程，如统一过程（Unified Process, UP）或极限编程（Extreme Programming XP），不会事先就提供详细的需求，它们也不会将注意力放在详细的模型上面，（当然事前也不会有详细的数据模型）。相反，他们会随时演变自己的模型，以反映出「他们对问题域理解的变化」以及「他们的涉众对需求的变更」。有些项目团队可能会选择一种更为连续的方式来工作，他们甚至可能在项目生命周期的初期就提出一个详细的概念数据模型（conceptual data model），然而这些团队会或多或少地介于这两者之间（尽管你们也希望能够支持他们）。敏捷 DBA 将需要传达遗留数据源所施加的约束（在第 8 章中予以论述），与应用开发者一起工作来理解这些约束并进行正确的工作。

敏捷 DBA 将会随时演变他们遗留数据的样式（schema），适当地应用常见的数据库重构（database refactoring，在第 12 章中予以论述）以及使用新的工具进行工作以随时迁移他们的数据样式。这是一个艰难但必需的任务。敏捷 DBA 也需要与应用开发者一起工作来建模他们的数据需求，与某些项目团队一起使用基于 UML 的工件（例如类图），而与另外的团队则使用概念数据模型。敏捷 DBA 将会与应用开发者一起工作来编写和测试数据库代码（如存储过程），在应用内部与其数据源交互的面向数据的代码，以及甚至是从应用样式（application schema）到数据样式（data schema）的映射。性能调优（在第 15 章中予以论述），包括数据库和数据库映射，是该项工作的重要方面。

敏捷 DBA 通常会与企业管理员一起工作，企业管理员负责公司元数据（meta data）的维护和演变工作，而公司的元数据用来描述企业和公司的开发标准和指导原则。敏捷 DBA 将会使用该信息并且遵循这些标准和指导原则，以及提供有价值的反馈。敏捷 DBA 也将会与企业管理员和其它的敏捷 DBA 进行交互，以随时演变企业的各种数据源，包括关键性的元数据。

敏捷 DBA 也会与企业架构设计师一起工作，以确保他们的工作能够适合整体的图景，以及有助于随时演变企业的架构。

本书所阐述的大多数素材都是从敏捷 DBA 和应用开发者的角度来加以描述的，我这样写是为了要尽可能保持一致和简单。

应用开发者

对于敏捷数据方法而言，*应用开发者*是指任何积极参与软件应用的非数据方面的创建和演变的人（记住，任何所指定的人都可能会担当多种角色）。应用开发者首要关注的是分配给他或她的一个单独的系统或产品线。该角色的职责包括与“传统角色”——程序员、建模员、测试员、团队组长、业务分析员、项目经理和部署工程师——相关的职责。

正如前面指出的那样，应用开发者的工作与负责一个或多个应用的数据方面工作的敏捷 DBA 有着非常密切的关系。应用开发者的首要客户包括他们系统的潜在用户、他们的经理、以及他们组织机构内部的操作和支持团队。第二类客户包括其他的项目涉众，例如高级管理者、企业管理员和企业架构设计师。

重要的一点是，应用开发者知道尽管他们首先关注的焦点是完成直接项目涉众的当前需求，而他们的项目则位于更大范围的组织机构内部。这种哲学观反映出 AM（在第 10 章中予以论述）的*软件是你的首要目标（Software Is Your Primary Goal）*和*让下一个工作成为你的第二目标（The Next Effort Is Your Secondary Goal）*原则——在这种情况下，接下来的一部分工作就是确保你的项目符合整个企业的愿景。应用开发者最好能够认识到他们正在工作的项目只是其组织机构内多个项目中的一个，在他们项目之前和之后都会有许多项目进行，因此，他们需要与其他角色的人一起工作，以确保他们做正确的事情。

应用开发者将会采用并遵循敏捷软件开发过程，例如 FDD、DSDM 和 XP。当进行建模和文档工作时，他们很可能会使用 AM 的原则和实践来改进这些过程。所有这三种过程都是敏捷的，都是在恳求开发者要与他们的项目涉众进行密切的工作。这暗示着开发者要负责在软件开发的基础原理上帮助教育他们的涉众，包括用户和管理者，从而有助于他们做出更有见地的、与技术相关的决定。

一个组织机构的遗留系统，包括遗留数据源（在第 8 章中予以论述），将会制约应用开发者的工作。通常将难以对这些系统进行演变，并且如果它们能够演变，那么进行的步伐往往非常缓慢。幸运的是，敏捷 DBA 能够帮助应用开发者解决因为遗留数据源而强加在他们身上的那些现实问题，但是他们将需要与企业管理员一起工作，而且更多的时候要与企业架构设计师一起，以确保他们的工作能够反映你们组织机构的长期需求。同敏捷 DBA 一样，应用开发者也需要认识到他们要遵循其组织机构的开发实践，包括企业管理员所支持的指导原则和标准。期望应用开发者能够提供有关这些标准和指导原则的反馈；组织机构中的每个人都应该这样做，并且要做好与企业管理员一起工作的准备，来开发组织机构的新开发环境的指导原则。

应用开发者也需要与企业架构设计师进行密切地工作，以确保他们的项目能够利用现有的企业资源并合理地适应到整个企业的愿景中去。企业架构设计师应该能够提供这种指导，并且将会与你们的团队一起工作来规划（architect）甚至是构建你们的系统。此外，应用开发者应该希望接受“高级”技能方面的辅导，例如架构和建模。这种方式使你们团队能够很容易地支持企业的工作，并有助于使企业架构保持充分的根据，因为他们很快便能发现自己的架构在实践中是否真正有用。

企业管理员

*企业管理员*是指任何积极参与确定、文档化、演变、保护以及最终收回企业 IT 财产的人。这些财产包括企业数据、企业开发标准、指导原则，以及诸如组件、框架和服务这样一些可重用的软件。该角色的职责可能包括与传统的数据管理员、网络管理员、重用工程师和软件 过程专员角色相关的职责，但并非仅限于此。企业管理员与企业架构设计师的工作密切相关，尽管他们的首要客户团队是高级管理人员和项目团队。在很多时候，企业管理员是“公司大门的保卫者”，提供对项目团队的支持，而同时还要引导他们以确保完成公司的长期愿景。保卫和改进公司财产的质量是一个重要的目标，这些财产并非仅限于数据。好的企业管理员是具有一种或多种专长的通才，数据管理（data administration）可能只是其中的一项专长，他们通晓范围广泛的与企业相关的问题。

企业管理员知道这项工作要比数据管理更多，并将以一种演进的工作方式来支持敏捷开发团队。这是因为企业管理员的工作与敏捷 DBA 密切相关，而且针对更小范围的应用开发者，他们便是以这种方式进行工作。企业管理员与敏捷 DBA 一起工作，以确保他们的数据库能够反映企业的整体需求和方向。企业管理员将会找到「向敏捷 DBA 和应用开发者传达他们角色的重要性」的方法，并且这样做的最好方式就是专注于那些能使他们在工作中更有效率的东西——很少有人会拒绝援助之手。试图通过繁重的过程或管理法规来强迫你们的意志将很可能不会奏效。

企业管理员要与敏捷 DBA 和应用开发者一起工作，以确保这些群体能够理解他们所应遵循的公司标准和指导原则。然而，他们的角色是对标准和指导原则进行支持，而非强制。一个好的经验做法是如果你需要担当“标准警察”的角色，那么你就失去了这场斗争。此外，这次失败很可能是你的错误，因为你未能很好地传达标准，没有赢得支持，或者试图去强制执行那些不切实际的指导原则。如果这些标准和指导原则有意义，它们被编写良好，而且易于遵守，那么数据和应用开发者便很愿意去遵循它们。然而，如果不是这种情况，当标准和指导原则不适合或是对项目施加过度的负担，企业管理员就应该能预见到这种回坐力（pushback）。是的，有些个体可能会开「遵循标准和指导原则」的玩笑，但那是项目指导员/经理需要处理的事情。

当出现回坐力的时候，企业管理员要与项目团队一起工作来研究和解决问题。他们要做好随时对标准和指导原则进行演变的准备，以反映所吸取到的教训和组织机构变化的事实。一种尺寸无法适用于所有情形——你们关系型数据库的命名规范可能与你们的 Java 命名规范有着非常多的不同，这是因为它们是具有不同优先考虑集的不同环境。

企业管理员要与构设计师密切地进行工作，来向架构设计师传达当前环境所强加的约束。更为重要的是，企业管理员需要理解企业架构设计师所展望的未来方向，以确保他们的努力能够支持组织机构的长期方向。

企业架构设计师

*企业架构设计师*是指任何积极参与创建、演变和支持/传达企业架构的人。架构常常会被描述成一个模型集。这些模型描述了各种视图，其中一种视图可能就是面向数据的，尽管网络/硬件视图、业务视图、用途视图（usage view）和组织机构的结构视图（举几个例子）同样具有价值。该角色的职责包括与传统角色——企业数据架构设计师，企业过程架构设计师，企业网络架构设计师，等等——相关的职责，但并非仅限于此。

与企业管理员的角色一样，企业架构设计师的角色与仅仅处理数据相比具有更大的空间——相反，他们着眼于企业的整个图景。企业架构设计师的主要工作是洞察未来，来努力辨别组织机构以后的发展方向，从而确定其 IT 基础设施需要如何演变。企业架构设计师会很自然地受到组织机构当前所处的形势、其周遭环境以及其演变能力的限制。企业架构设计师要与企业管理员密切地工作，以确保他们理解当前的环境，并传达他们对未来的愿景。企业架构设计师的首要客户是组织机构的高级管理者，包括 IT 管理者和业务管理者，他们与这些人一起工作来演化企业的愿景。项目团队也是主要的客户，因为他们的工作应该能够反映整个企业架构，并且他们会对架构提供重要的反馈。

企业架构设计师专注于各种类型的架构问题，数据只是其中的一部分。他们的主要目标是开发企业架构模型，然后对其提供支持。对于企业架构设计师而言，仅仅提出好的模型是不够的，他或她必须能够宣传（evangelize）这些模型，与开发团队一起工作，以及对高层管理者进行「能够揭示与系统架构相关的一般性问题」的培训。除了你们组织机构的 CIO 和 CTO 之外，你们的企业架构师与高层管理者打的交道很有可能最多；因此，他们需要做好「帮助高层管理者制定策略上面决定」的准备。

企业架构设计师会与敏捷 DBA 和应用开发者一起工作。架构设计师所能做的最重要的事情是“言行一致”，而且卷起袖子去积极地参与项目。这将会赢得开发者的尊敬，并能显著地增加「他们真正理解和遵循企业架构的愿景」的机会。这种方式的好处是其对「架构是否真正有效」提供迅即而具体的反馈，以及对「架构需要如何演变」提供有价值的见解。

企业架构设计师需要做好以一种迭代和增量的方式进行工作的准备。他们试图事先创建一个包含所有企业模型的集合是失策的。相反，创建一个初始的、高层的架构，然后与一个或多个开发团队密切工作以确保其能够发挥作用。AM 包含了一个被称为*小增量建模（Model In Small Increments）*的实践，该实践基于这样一个假设：你在建模时未收到具体反馈的时间越长，如一个实际项目所提供的反馈，则你的模型无法反映你们组织机构真实需要的机率就越大。敏捷企业架构设计师要避免这样的象牙塔式的架构。关于企业架构的敏捷方法的描述可参见如下网页：www.agiledata.org/essays/enterpriseArchitecture.html。

敏捷软件开发者

AD 方法的一个基本假设是你们的组织机构想要采取一种敏捷的软件开发方式。敏捷软件开发反映了一种思想模式的转换，这是一种新的考虑问题的方式。为了成功地运用 AD 方法， 充当前面所描述的四种角色的人们必须要有这种思维模式（mindset），关于该思维模式的特征描述如下：

团队协作。 敏捷软件开发者认识到与其他人一起有效工作的重要性并将依此照做。他们能够谦逊地去尊重和重视他人的意见与能力；如果不具备这种谦逊的态度，他们不太会愿意选择与其他人相互协作。一个重要的提示是每个人将必须要重新思考他们的工作方式，并且愿意为更美好的方式做出改变。“我的团队是世界的中心，并且每个人必须符合我们的愿景以及遵循我们的过程”，这样的态度不会有好的作用。

通用、有效的过程。 敏捷软件开发者积极寻求定义一种每个人都赞同的一般性的方法。我的经验是上层所强加的过程很可能会失败，因为这会导致一个团队拒绝该过程而“成为淘气者”。一个较好的过程改进策略是有机地培养一种能够工作的软件过程，并使其能映每个相关人员的需要。由于软件开发者是具备有价值技能的聪明人群，你很可能发现「每个人都赞同的原则集」常常是一个有效过程的最重要的部分。在 AD 方法的情形中，这些原则就是敏捷软件开发的价值观和原则，以及 AD 哲学观。

位置共享。 敏捷软件开发者愿意根据需要与其他的人共处一地。你可能需要在一段时间内放弃自己舒适的隔段或办公室，而在一处团队所共用的地方进行工作，或甚至让某些人分享你的办公室并针对一个项目与你一起工作。这个事实反映了交流和协作对于你成功至关重要；与其他人一起工作远远要比你一个人工作更为有效。

通用型的专家。 敏捷软件开发者是具有一项或多项专长的通才。这就意味着每个人都需要具有广泛的技能，并且愿意与其他人一起工作来改进既有的技能以及学习新的技能。（在第 23 章中会更加详细地探讨这个观念。）

过程的灵活性。 敏捷软件开发者也做好了对他们的方法进行裁剪的准备，以满足他们所参与的项目的需要。例如，工作在报表数据库（reporting database）上面的项目团队所采用的方法，很有可能与工作在使用 Java 或 C++ 编写的在线应用（online application）上面的团队有所不同。没有哪种方法能够适用于所有情形。

充足的文档。 敏捷软件开发者认识到文档在他们的工作中是必需的，如果他们愿意他们可以非常有效地去做这些事情。例如，企业架构设计师认识到企业建模的目标是生产有效的模型，以满足他们读者的需要，而不是生产大量的文档。他们发现许多传统的架构成果之所以失败，是因为开发者不愿意花费时间在文档中跋涉并依此来了解架构。应用开发者知道系统文档必须要支持以后的改进工作，而且敏捷 DBA 发现文档必须能描述它们所支持的数据源。敏捷软件开发者将会采取一种敏捷的文档方式（在第 10 章中予以论述），并生产出编制良好和简明的文档，它们恰到好处。

敏捷数据能解决我们的问题吗？

一个重要的问题是，在本书中论述的这些哲学观和所暗示的文化上的转变，能否解决组织机构所面临的有关软件开发数据方面的问题。以下列表展示了这些实际情况，并论述本章前面所提到的每个潜在的问题，以及 AD 方法所建议的解决方案。

不同的愿景和优先级。 敏捷数据恳求软件开发者要相互合作，并且理解和尊重他们合作者的观点。

角色过于专业化。 敏捷数据要求软件开发者去寻找位于「成为一名通才」和「成为一名专家」两种极端情况之间的最佳点，理想的情况是成为一名具有一项或多项专长的通才。

过程阻抗失配。敏捷数据明确表示，企业和数据专业人员必须做好遵循一种增量和迭代方式进行工作的准备，这是大多数现代开发的准则和敏捷软件开发的事实标准。同样显而易见的是，应用开发者必须要认识到现有环境和组织机构的未来愿景会制约他们的工作。

技术阻抗失配。 敏捷数据要求软件开发者彼此密切地工作，同时在这样做的过程中相互学习。敏捷 DBA 要具有把应用样式（application schema）映射为数据样式的能力，并且能够编写面向数据的代码，以及对他们的工作进行性能优化。

管理守旧。 敏捷数据要求企业架构设计师要与高层管理者一起工作，并且就现代软件开发的现实培训他们。同样地，应用开发者应该与各个层面的管理者一起工作，并帮助培训他们。

组织机构上的挑战。 敏捷数据要求软件开发者与其他人以及你们的项目涉众一起工作，尊重他们，并且积极致力于相互有效地工作。

糟糕的文档。 敏捷数据指导软件开发者去遵循敏捷文档（Agile Documentation）的原则。（在第 10 章中予以论述）。

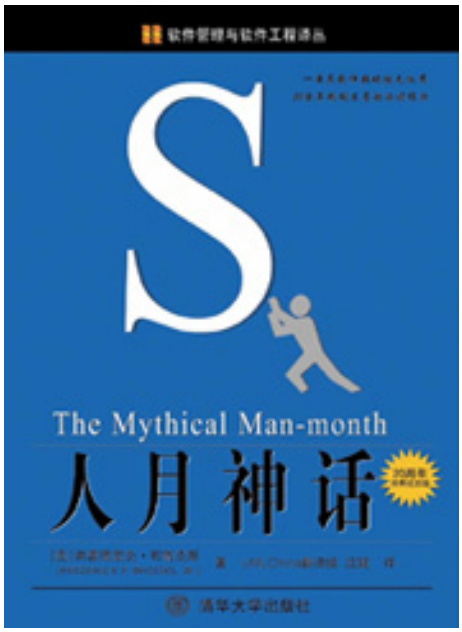
低效的架构工作。 敏捷数据建议企业架构设计师对架构采用一种多视图/模型的方法，并积极地工作在一个项目上面来支持和验证其架构的有效性。从而来自这些工作的反馈应该能被反映进该架构的后续迭代中。

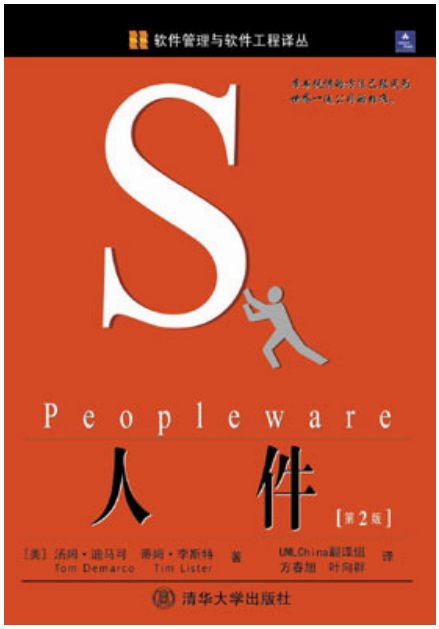
低效的开发指导原则。 敏捷数据恳求企业管理员要编写清晰和适用的标准和指导原则，而且做好落实来自开发团队的反馈的准备。

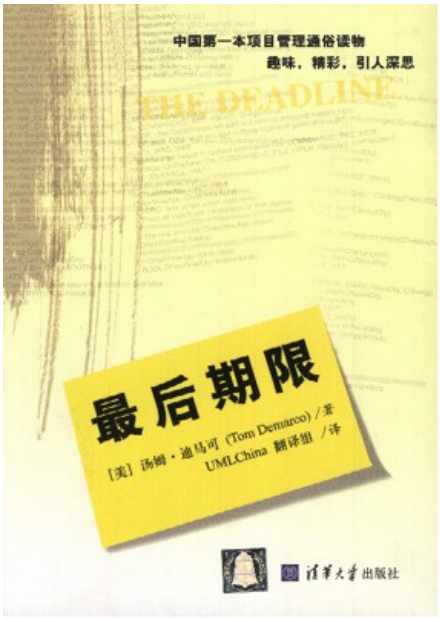
低效的建模工作。 敏捷数据指导软件开发者去遵循 AM 方法学的原则和实践（在第 10 章中予以论述）。

总结

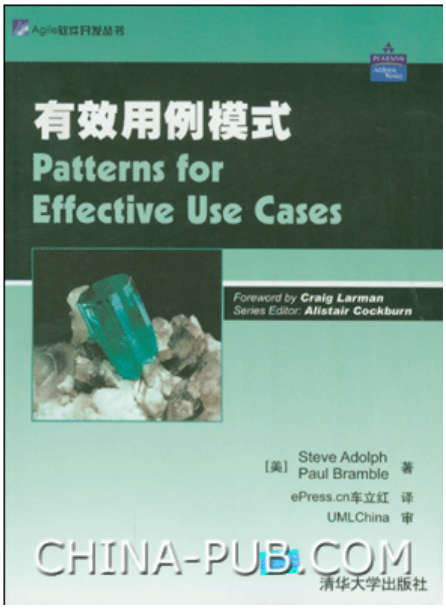
敏捷数据方法的核心是它的哲学观，以及这些哲学观揭示出的软件开发者所采取的工作方式的变化。首先要认识到你遇上了问题；许多组织机构有这样一个严重问题：他们的应用开发者和数据专业人员如何在一个层面上相互合作，以及项目团队成员如何与企业团队成员在另一个层面上相互合作。然而对于敏捷数据方法而言，仅仅提出一个哲学集是不够的，它必须还要描述那些软件开发者能实地应用在工作上面的、经证实的方法。你们应该思考这些方法，挑选那些听起来能使你受益的方法，对它们进行裁剪，并将它们应用到你们的环境中。软件开发者能够有效地相互合作，但是他们必须要选择去这样做。

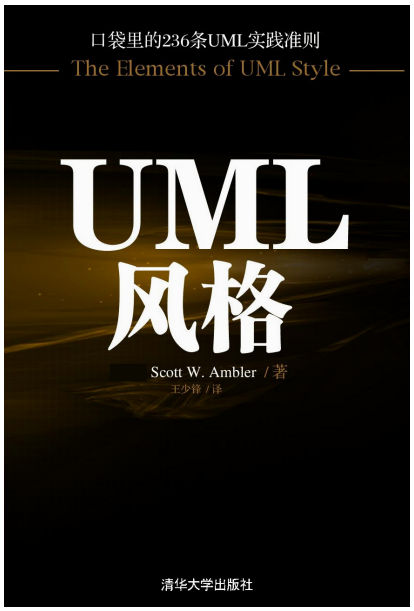


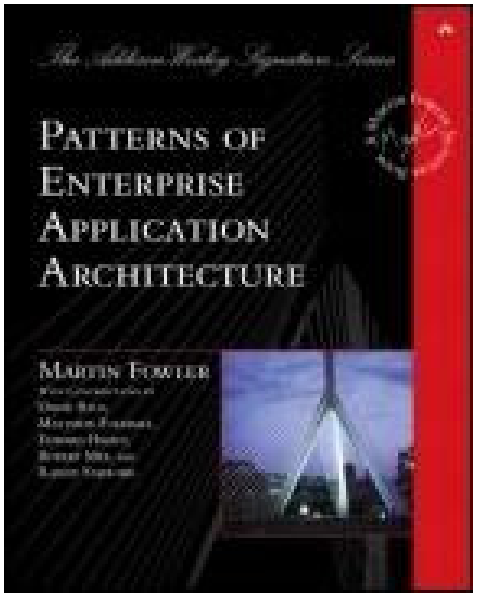


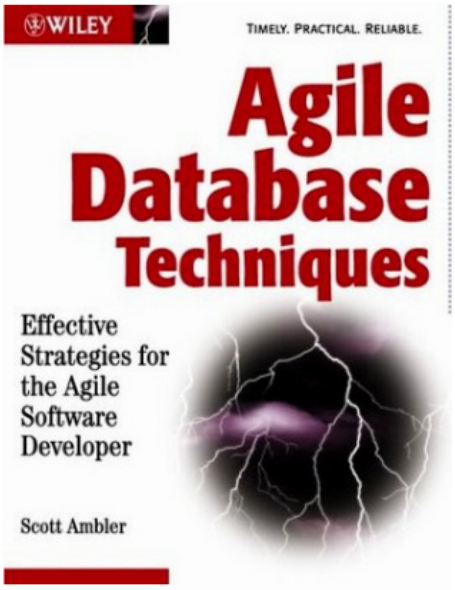


- 2002 年《人月神话》创下销售记录
- 2003 年《最后期限》、《人件》、《人月神话》分别排在 china-pub 年度总销售榜的 1、3、4 名









业务建模 vs 系统建模

业务建模可以看作软件开发之前的第零步工作流程。随着统一过程被越来越多的团队所采纳，基于用例和对象技术的业务建模成为团队所关注的重要技术。以下是笔者被问到过的一些有关业务建模的问题。把它们归纳出来，加上我自己的理解，希望对大家能有帮助。

全文内容刊登于2004年7期《程序员》杂志>>

勘误：第一个图中的标注，“业务对象”应为“业务实体”。