第 27 期（2003 年 7 月）

尖刀

X 非程序员
X-Programmer
软 件 以 用 为 本

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

http://www.umlchina.com/

UMLChina 公开课

“UML 应用实作细节”

（2003 年 7 月 26-27 日，北京）

现在，UML、RUP、Rose 等概念已经传播得越来越广，书籍、资料也越来越多，决定在开发过程中使用 UML 的开发团队也越来越多。

在开发团队应用 UML 的软件开发过程中，自然会碰到很多细节问题：“我这样识别 actor 和用例对不对？”、“用例文档这样写合适吗？”、“RUP 告诉我该出分析类了，可类怎么得出来啊？”、“先有类，还是先有顺序图啊？”、“类怎样才能和数据库连起来啊？”...许许多多的细节问题，而每一个细节都和背后的原理有关。

本课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，结合大量练习，阐述如何（只）使用 UML 里的三个关键要素：用例、类、顺序图来完成软件开发。使学员自然领会 OOAD/UML 的思想和技术，并对实践中的误区一一指正。整个过程简单实用，简单到甚至可以只有一个文档。

详情请见>>

UML2.0 正式通过

[2003/6/12]

上周巴黎的 OMG 技术会议上，分析和设计专案小组（the Analysis and Design Task Force）投票通过了 UML™ 2.0 Superstructure 规约，至此，本行业最重要的软件建模符号的主要升级宣告完成。补充的元模型设施 (MOF™) 2.0 规约以及 XML 元数据交换(XMI®) 也被推荐，它们将改进 UML 和 MDA®工具的仓库基础。UML 2.0 元模型和 MOF 元模型的一致性将简化模型间经由 XMI 的转换以及跨工具的互操作。

“基于 UML 5 年多的应用经验，我们对统一建模语言进行了大量研究，UML2.0 代表了我们表示并交流系统规约能力上革命性的进步－它为 MDA 提供了坚实的基础”，Jim Odell 说，他是分析和设计专案小组的顾问、作者以及副主席。

OMG 的主席和 CEO ：Richard Soley 博士补充，“50 多个公司贡献了他们最优秀的技术以及实践，OMG 的工作再一次证明了：一个重要的新标准将对软件开发的未来产生巨大的影响”

升级后的 UML 标准有以下特点：

- first-class 的扩展机制允许建模人员增加自己的元类（metaclass），从而可以更加容易地定义新的 UML Profile，将建模扩展到新的应用领域。
- 对基于组件开发的内置支持简化了基于 EJB、CORBA ® 组件或 COM+的应用建模；对运行时架构的支持允许在系统的不同部分进行对象和数据流建模；对可执行模型(executable model)的支持也得到了普遍加强。
- 对关系更加精确的表示改进了继承、组合和聚合以及状态机的建模。
- 行为建模方面，改进了对封装和伸缩性的支持 ，去掉了从活动图到状态图的映射（译者注：活动图不再是一种特殊的状态图），并改进了顺序图的结构。
- 对语言的句法和语义的简化，以及整体结构上更好的组织。

（自 OMG，袁峰 摘译）

QuickUML 1.1 for Windows、Linux 上市

[2003/6/3]

Excel 公司开始销售 QuickUML 1.1 for Windows 和 for Linux 版本。QuickUML 是一种提供 UML 主要模型之间的紧密结合以及同步的面向对象设计工具。通过卡片窗口的形式提供对用例（use case）、类模型、对象模型、字典和代码的支持。用例用来捕捉系统必须提供的用户可见的本质的功能。类模型描述系统中的对象及其静态关系。顺序图用来描述对象之间的交互并强调事件发生的顺序。项目可以保存为平台独立的 XML 文件，该文件可以在 Windows, Linux 或者 Mac OS X 等各种操作系统相应的 QuickUML 版本上进行编辑，Mac OS X 的版本是最近刚刚推出的。

QuickUML Windows 1.1 版本增强了对 Windows XP 的支持，提供了新的在线帮助系统以及图形输出功能，可以将类图和对象图保存为位图、JPEG 或者 GIF。其中，QuickHelp 取代了 WinHelp ，它提供了内容敏感（context sensitive）的应用支持，在 Windows、Macintosh 和 Linux 各种平台上为用户提供统一的服务接口。QuickHelp 将内容以及主题面板集成到一个可缩放的窗口中。自动的关键词索引以及查找功能使得查找更加便捷。超链接以及、后退按钮提供了迅速的导航能力。QuickUML 窗口的其他加强功能还包括对类和对象模型的视图改进、UNC 文件名的支持，以及放置常用代码管理命令的工具栏。

QuickUML Linux 1.1 版本增加了改进的字体处理、对类和对象模型的视图改进、以及放置代码管理命令的工具栏。QuickUML Linux 拥有和 Windows 版本一样的功能，同样通过 QuickHelp 来提供内容敏感的应用帮助。

QuickUML for Windows 和 Linux 都加强了跨平台和编程语言的支持。QuickUML 的项目文件中存放的是平台无关的 UML 数据以及和平台相关的文本字体、大小、在 Windows、Linux 和 Macintosh 平台下的文件路径。QuickUML 支持各种平台上灵活的文本处理功能以适应 Windows (CRLF)、Macintosh (CR) 和 Linux (LF)等不同的终端要求。在设计中，可以定义诸如属性数据类型以及方法参数列表等语言（如 C++、Java、Delphi、Ada 和 Visual Basic）相关的细节。.这些内容作为编程规约以文本形式列出。

QuickUML 1.1 for Windows 或者 Linux 标价为$495，如果从 1.0. Site licensing 版本升级，费用为$124。关于产品的描述以及在线订购，请登陆 www.excelsoftware.com。

自 1985 年来，Excel 公司为 40 多个国家的开发者提供了各种工具。

（袁峰 摘译）

OMG 推荐模型驱动架构 FastStart 计划

[2003/6/10]

OMG（TM）发布了 MDA FastStart(TM) 计划，这是一种帮助 IT 组织尽快使用 OMG 的模型驱动架构的新途径。最近 OMG 发现很多公司对应用 MDA 很感兴趣，但是在如何实现方面正在寻求外援，因此 OMG 建立了 MDA FastStart 计划。MDA FastStart 帮助企业寻找有经验的服务商，在保证支出合理的同时，帮助企业迅速开始 MDA 应用。“MDA FastStart 计划对 OMG 及其成员来说是一项很令人兴奋的开发，那些对实现 MDA 解决方案和评估其过程的有效性感兴趣的企业，需要通过 FastStart 计划寻求帮助。”OMG 主席和 COO Bill Hoffman 说。

OMG 的 MDA 是使用统一建模语言 UML(TM)进行企业架构建模的有力途径，UML 已经成为应用设计的工业标准。MDA 首次提供了一个标准化的途径将企业级标准背后所有的基础逻辑与特定的中间件，用来实现中间件的消息和中间件语言－CORBA(R)、 Web Services、 Java、XML 等，进行了清楚的划分。当保持和升级其对现有平台的投入时，使用 MDA，IT 企业遇到了新平台带来的完整的挑战。

OMG 计划主管 Michael Guttman 领导 MDA FastStart 计划。Guttman 先生是一位在 IT 企业的架构、设计、开发和使用大规模复杂企业软件系统和基础设施方面具有丰富咨询经验的 MDA 倡导者。在 MDA FastStart 计划上，Guttman 先生及其助手一直与企业专家合作，进行标准化服务计划的设计，寻找适合使用这项计划的服务商，并与其合作。当最终的用户企业要求 MDA 教育和咨询服务时，MDA FastStart 计划办公室将把那些有经验的服务商推荐给他们。需要 MDA FastStart 服务的最终用户在这些服务商中，选择那些不但能够提供 MDA 专业技术，还有高质量的指导、培训和实现支持的服务商。

Guttman 说，“当你跨企业系统地使用 MDA 时，它的作用最大，但你必须从某处开始。我们建立了 MDA FastStart 计划来保证需要从 OMG 获得帮助的 MDA 最终用户，能够找到合格的服务商，他们需要尽快地使用 MDA。同时，该计划应当有助于加速开发 MDA 服务和 MDA 工具的市场。对每个人来说，这都是双赢的方式”。

有兴趣的最终用户和潜在的 MDA 服务商可以通过访问 http://www.omg.org/fast-start/获得更多信息。

（June 摘译）

Borland 向 Eclipse 组织推荐敏捷设计解决方案

[2003/6/10]

Borland 软件公司(Nasdaq:BORL)今天宣布 Eclipse 的 Borland(R) Together(R) 版正式发布，该版本扩展了 Borland Together 的解决方案，为 Eclipse 平台提供了建模、设计和质量保证功能。Eclipse 的 Together Edition 版可完全集成到应用开发生命周期的全部过程中，帮助开发团队迅速创建高质量、高性能的企业应用。

Eclipse 的 Together 版提供了一种敏捷统一建模语言 UML(TM)开发环境，它可以完美地集成到开源的 Eclipse 平台上。Eclipse 平台是用来建立集成的开发环境，可用来生成多种运行环境的不同端到端的计算解决方案。

Eclipse 的 Together 版可把建模无缝集成到开发过程，将诸如自动审计和度量、文档生成和模式的能力提供给开发团队。Eclipse 的 Together 版帮助团队以敏捷方式使用 UML，通过使用诸如 Borland(R) LiveSource(TM)这样的技术，在开发过程中保证模型和源代码保持同步。

最新的 Together 版强调 Borland 的提供 vendor-neutral 的解决方案，这是针对 Java(TM)开发团队跨平台和环境的需要。以用户界面嵌入 Eclipse 为特色，Eclipse 的 Together 版允许开发者使用熟悉的技术，用以增进团队的效率。

“用 Together 的 award-winning 设计技术支持 Eclipse 平台，强调了 Borland 提供解决方案的策略，更容易跨应用开发周期进行集成，” Together 版的副总裁和经理 Tony de la Lama 说， “Borland 准备加速开发可在多平台上运行的新型的、高性能的应用软件。”

“我们很高兴 Borland 已经发布了新的 Together 版，可以将建模集成到 Eclipse 平台上，”HP Enterprise Systems Group，Developer Resources Organization 的倡导者 Mike Rank 说。“现在，Eclipse 平台的用户拥有新的能力去改进应用开发的设计和质量。”

“一个健壮的分析和设计解决方案，像可集成到 Eclipse 体系中的 Borland Together 版，为 Eclipse 团体提供了及时的利益。” Eclipse 倡导者团体的主席 Skip McGaughey 说。“利用 Borland 为开发者开发产品的多年的经验，并把这项技术提供给 Eclipse 开发者，这是令人兴奋的。”

可用于 Red Hat(R) Linux(R) 7.3 or 8.0 、 Microsoft(R) Windows(R) 2000 (SP2 或更高) 、 Windows XP(R) Professional 的 Eclipse 的 Borland Together 版，将于 6 月 10 号问世。可在 http://shop.Borland.com 下载试用版，访问 http://www.borland.com/products/index.html，可获得更多信息。

（June 摘译）

Flywheel：前进的马达在轰鸣

[2003/7/1]

架构设计师和设计人员通常使用 UML 进行设计，而 Microsoft 的开发人员大多使用 Visual Basic 或 C#进行开发，这里有着难以逾越的沟壑。

Flywheel 是 Velocitis 公司提供的以代码为中心的崭新的开发工具。作为 Visual Studio .NET 的插件，它为设计人员和开发人员提供了共同的工作区。Velocitis 总裁 Steve Dadoly 声称，“我们的目标是简化编码人员的 UML 使用，并将其一对一地映射到.NET”。

通过 Flywheel 与 VS.NET 的协同使用，架构师能够使用 UML 形式的可视化方式创建应用架构，并且生成 C# 或 Visual Basic 的代码。因为这些可视化内容是简单的 XML 文本文件，并且是源代码的部分，因此，不存在同步的问题。Dadoly 介绍，设计中所有可视形式的注解或者参数，都会在源代码文件中保留。用户可以选择不同的模板（preference templates），定义代码单元的可视化形式和格式。

Dadoly 介绍，利用 Flywheel，可以为.NET 平台绘制 UML 风格的类结构和依赖图。它可以在 UML 静态结构图和.NET 代码之间建立映射，提供可视化设计和重构的支持。

开发人员可以在 VS.NET 打开架构师的设计文档，可以看到架构师的注解，并据其进行代码实现工作。工具中具有对反工程的支持，当代码发生变化的时候，可视化的设计部分会发生相应的变动，反之亦然。

“维护起来非常方便，因为编码的过程一直是可视化的”，Dadoly 补充，“源代码或者 UML 中的参数都被可视化的表现出来”，因此，开发人员和设计人员都可以很好地理解。

利用 Visual Studio .NET 的浮动窗口工具，Flywheel 允许用户用一个窗口显示编辑中的代码，另一个窗口可视化地显示程序结构。Dadoly 介绍，Flywheel 中的两个浏览窗口—Solution Explorer 和 Type Explorer—提供了代码单元的树状视图，并允许用户方便地进行添加、移动、删除以及拖拉代码单元等操作。

可以在公司网站（www.velocitis.com）购买到 Flywheel，价格：US$799/用户。

（袁峰 摘译）

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

中译本即将上市

样章先行提供>>

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

第一本 UMLChina 训练指定教材

清华大学出版社

UML 相关工具一览（2003 年 7 月版）：I-0

整理

查看评论

接 26 期…

工具（最新版本）	厂商	试用允许	UML 支持								支持代码环境	XMI	平台	备注
			用例图	类图	状态图	活动图	顺序图	协作图	构件图	部署图				
Flywheel	Velocitis http://www.velocitis.com/ VELOCITIS™	100 天试用		√							C#、VB.Net	√	Windows	UML 到.NET 代码的映射，完全结合 VS.Net 2003。
IAR visualSTATE	IAR Systems（瑞典） http://www.iar.com/Products/vS/ IAR SYSTEMS	可以试用			√						C/C++			使用 UML 进行嵌入式系统开发。包括设计、测试。自动生成 C/C++代码和全套文档。
Ideogramic UML 2.3.3	Ideogramic ApS（丹麦） http://www.ideogramic.com/products/ IDEOGRAMIC	有试用版	√	√	√	√		√			Java，C/C++	√	Windows, Linux	关注“用手建模”的 UML 建模工具，强调创造性和弹性。支持电子白板，支持在桌面、

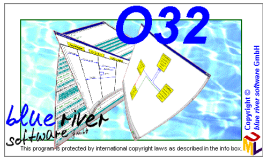
														可移动物体上建模。
INNOVATOR 8	MID GmbH（德国） http://www.mid.de/de/innovator/object/ Enterprise Software Solutions MID		√	√	√	√	√	√	√		Java，C/C++，Smalltalk，Forte，Object COBOL，IDL，VB	√	AIX，DEC VMS，HP-UX，Linux，OS/2，Solaris，Windows	可以和 BPR 工具集成，良好集成版本控制工具（PVCS，Clearcase…），自动产生 Word，FrameMaker，PS 文档。
iUML 2.2	Kennedy Carter http://www.kc.com/products/iuml/index.html KENNEDY CARTER	有试用版	√				√						Windows	自动禁止“无效”模型。模型的执行、测试和调试。支持 MDA。
ISFxIAR	Projexion Netsoft（法国） http://www.projexion.com/index.php?lang=fr&ID=10 Projexion NetSoft										Java			
Iss-UML	Halstenbach（德国） http://www.halstenbach.com/home.php3 ...Halstenbach to be a lit										Eiffel			Rose 插件，提供 Eiffel 双向工程支持。
J2U	NASRA（法国） http://www.nasra.fr/flash/NASRA.html	有试用版		√			√				Java	√	Java	顺序图双向工程，从可执行 Java 代

														码到 XMI，可以被输入到任何兼容 XMI 的 UML 工具。
Javelin 6.5.8	Step Ahead http://www.stepaheadsoftware.com/javelin.htm Step Ahead Software	有试用版									Java		Windows	基于 UML 的图形编程环境，自动保持类图和 Java 代码同步。
JBuilder 9	Borland http://www.borland.com/jbuilder/ Borland USA		√	√	√	√	√	√	√	√	Java	√	Java	结合 Together 的技术后，JBuilder 已经成为一个全程的开发工具。
JDeveloper 9i	Oracle http://www.oracle.com/ip/develop/ids/index.html?jdeveloper.html ORACLE			√		√					Java		Java	结合了 UML 的 J2EE 工具
JVISION 2.1	object-insight http://www.object-insight.com/product/ object-insight	有试用版		√							Java		Linux，Solaris，Windows	
JSequence	Objective Ideas（瑞典） http://www.obj.se/jsequence/jsequencestart.htm	有试用版					√				Java		Java	自动从 Java 代码中产生顺序图。

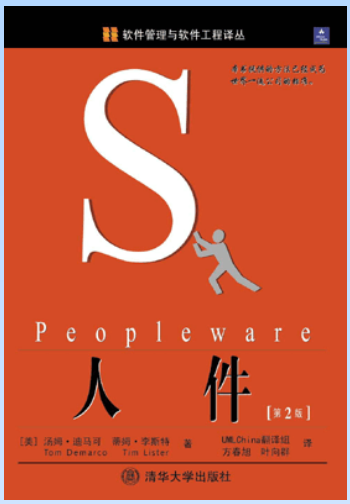
Jude 1.09	Objectclub http://objectclub.esm.co.jp/Jude/jude-e.html 	免费	√	√	√	√	√	√			Java		Java	Java 双向工程，支持 UML 1.4
JUMLi	Kreditwerk（德国） http://www.jumli.de/ kreditwerk		√	√	√	√	√	√			C++、Java、C#		Java	
kUML	kUML http://www.informatik.fh-hamburg.de/~kuml/ 	开源	√	√									Linux	
MacA&D 7.4	Excel Software http://www.excelsoftware.com/ Excel Software Software Engineering Made Easy										C++、Java、Delphi	√	Mac	支持 CRC 卡
MagicDraw 7.0	No Magic, Inc. http://www.magicdraw.com/ 	有试用版	√	√	√	√	√	√	√	√	C++、Java、C#、IDL	√	Java	支持多国语言。支持 RUP 和 WAE 设计模式
MasterCraft	MasterCraft（印度） http://www.tata-mastercraft.com/overvi													基于 UML 的构件建模工具

Component Modeler	ew.asp 													
MEGA 6.0	MEGA Intenational（法国） http://www.mega.com/us/product/overview/ 	有试用版	√	√	√	√	√	√	√	√	Java、VB、C++	√	Windows	系列套件，从业务流程分析到构件设计，基于 UML。
MetaEdit+ 3.0	MetaCase Consulting（芬兰） http://www.metacase.com/ 	有试用版									Smalltalk、C++、Java、Delphi (Object Pascal)、SQL、CORBA IDL		Linux，Windows	元模型工具，让你定义自己的标记
Metamill 3.0	Metamill（卢森堡） http://www.metamill.com/ 	30 天试用	√	√	√	√	√	√	√	√	C++、Java、C#	√	Linux，Windows	多用户建模支持。
MiUML 0.98	SoftwareFarm http://www.swfm.com/miuml.htm	免费										√	Java	此项目目前已停止

														
Modelmaker 7.05	Model Maker（荷兰） http://www.modelmakertools.com/ 	有 demo 版	√	√	√	√	√	√	√	√	Delphi		Windows	针对 Delphi 的工具，支持构件和模式，支持某些“自适应”的方法学。和 Delphi IDE 自然结合。
Model Prototyper	ObjeXion（法国） http://www.objexion.com/											√		从 UML 模型产生 SQL 和 GUI 原型
Model-in-Action 4.0	SodiFrance（法国） http://www.model-in-action.fr/en/index.php?lang=en 	有试用版												专注可裁减的代码生成。覆盖现在使用的各种主流语言。
Modelistic 1.1	Modelistic（英国） http://www.modelistic.com/ 	有 demo 版		√							Java		Java	强有力的 Java 双向工程
Novosoft UML Library	Novosoft（俄罗斯） http://gemini.novosoft.ru/NS2B.nsf/w1/UML_Library	开源	√	√	√	√	√	√	√	√	Java	√	Java	

														
O32 1.3	blue river software（德国） http://www.blue-river-software.com/products/o32/o32.htm 		√	√	√		√				C/C++		Windows	
ObjectArtist 0.1.1	Sven Daumann http://www.objectartist.org/	开源		√							Java	√	Java	支持设计模式
Object Domain R3	Object Domain Systems http://www.objectdomain.com/ 	有试用版	√	√	√	√	√	√	√	√	Java、C++、IDL、JPython	√	Java	支持多用户，支持 Swing ， 使 用 JPython 2.1 作为脚本语言。客户有 Alcatel 等大公司。
Objecteering/UML 5.2.2	SOFTEAM（法国） http://www.objecteering.com/ 	有试用版	√	√	√	√	√	√	√	√	Java、C++、IDL、DDL、Oracle	√	Windows, Linux，Solaris	自动模式支持，产生测试脚本..有专门支持极限编程（XP）的配置。支 持 MDA 和 SPEM。
Object Engineering Workbench（OEW）	Innovative Software http://www.isg.de/ 			√	√	√	√	√	√	√	Java、C++、DDL、ObjectStore&POET		Windows	
待续…														

老板，别把开发人员当成牲口--《人件》热销中



Tom Demarco，Tim Lister

翻译：UMLChina 方春旭 叶向群

http://www.peoplewarecn.com

人件中文版网站

Frederick P. Brooks, Jr--《人月神话》第 19 章

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品。当人件第一版出版时，我写了一份评论，“我强烈推荐你买一份《人件》给你或你的老板，如果你是老板，那么为你部门的每一个人买一份，并给自己买一份”。这建议在 12 年后的第二版依然有效，并且更加热烈。

Mark A. Herschberg

我只学了办公环境的章节，就辞掉了原来的工作！

Joel Spolsky

我想，微软成功的原因之一就是公司里的所有经理都读过《人件》

Steve McConnell

由于本书第 1 版的赫赫声名，新版的《人件》是我不用看就会决定购买的少数几本书之一。

购买>>

UML 状态图指南

Scott W. Ambler 著，梁涛 译

查看评论

UML 的状态图是基于实体在受到事件触发之后的反应来描述它的动态特性，揭示实体当前所处的状态在不同的事件触发下它的不同反应。我们创建 UML 状态图用于：

---揭示 Actor，类，子系统和组件的复杂特性。

---为实时系统建模

1 总论：

1.1 实体状态发生变化时绘制状态图

《敏捷建模》（Agile Modeling）这本书的“最大化 Stakeholder 投资”的原则建议你只有在建模对你的工作有正面效果的时候才进行建模。如果一个类或者组件无论它当前的状态如何，都产生相同的状态变化，那么绘制状态图的功能就微不足道了。

例如“邮寄地址”这样的一个类只是用于表示数据，并且数据只在系统里面被用于处理，对于这样的类，绘制状态图是没有多大意义的。相反，“讨论班（Seminar）”这样的对象则非常值得来绘制状态图，例如根据它当前登记状态的不同，对于新的学生登记就会有不同的反应。请看图 1。

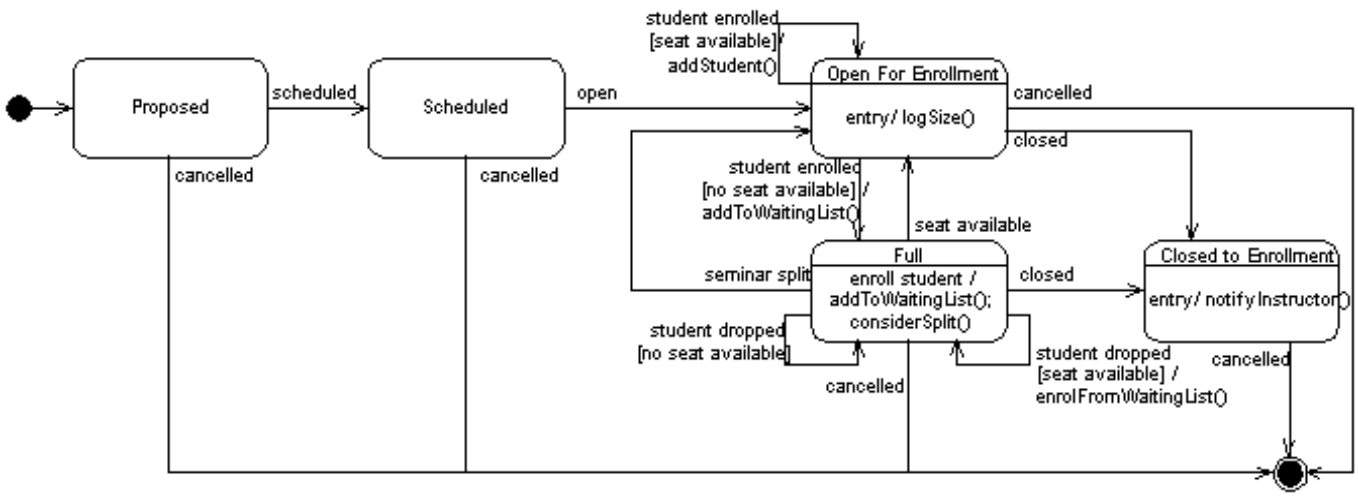


图 1 讨论班（Seminar）在学生登记过程中的 UML 状态图

1.2 将起始状态放置在左上角

用实心的圈代表起始点，放在左上角符合西方文化里的阅读习惯

1.3 将结束状态放置在右下角

结束点用实心黑圈加圆形边界来表示，放在右下角符合西方文化里由左而右，由上而下的阅读习惯

2 状态指南

状态是指一个实体行为模式中的一个阶段。实体的状态是由它属性值来体现出来。

例如在图 1 中，当讨论班（Seminar）这个实体的“登记开放”（Open For EnrollMent）这个属性被标识为确定的（yes or no），那么这个实体就处于“登记开放”的状态，讨论班就有空余的位置可以填充。

2.1 状态名称要简单而且具有描述性

“已开放”和“已计划”（Proposed）这样的状态名称很容易理解，这样就增强了状态图的交流价值。

2.2 质疑“黑洞”（black hole）状态

黑洞状态是指一个状态只有进入的转移而没有出去的转移，这种状态只有对于终止状态是可以的，否则往往意味着你丢失了一个或多个转移。

2.3 质疑“奇迹”（Miracle）状态

奇迹状态是指一个状态只有出去的转移而没有进入的转移，这种状态只有对于初始状态是可以的，否则往往意味着你丢失了一个或多个转移。

3 子状态建模指南

3.1 在目标状态趋于复杂的状况下为子状态建模

由于在图 1 里面的状态图没有为课程登记结束后的状态建模，关于“讨论班”这个实体的状态图是不完全的。图 2 对讨论班的整个生命周期的建模。图 2 将图 1 描述成一个称为登记（Enrollment）的组合状态，又称为超状态，它是由一些子状态所构成的。注意为了简单起见，在转移上的标签都没有标上，通常情况下你必须如图 1 那样将转移上的标签标上。当一个现有的状态展示出它的行为的复杂性的时候，为它的子状态建模是有意义的，这样就激发你研究它的子状态的兴趣。

当几个状态共享一个进入或离开条件的时候，建立超状态是有意义的（Douglass 1999）。

图 1 里面大家看到所有的状态都共享一个关闭（Closed）的转移来进入最终状态。

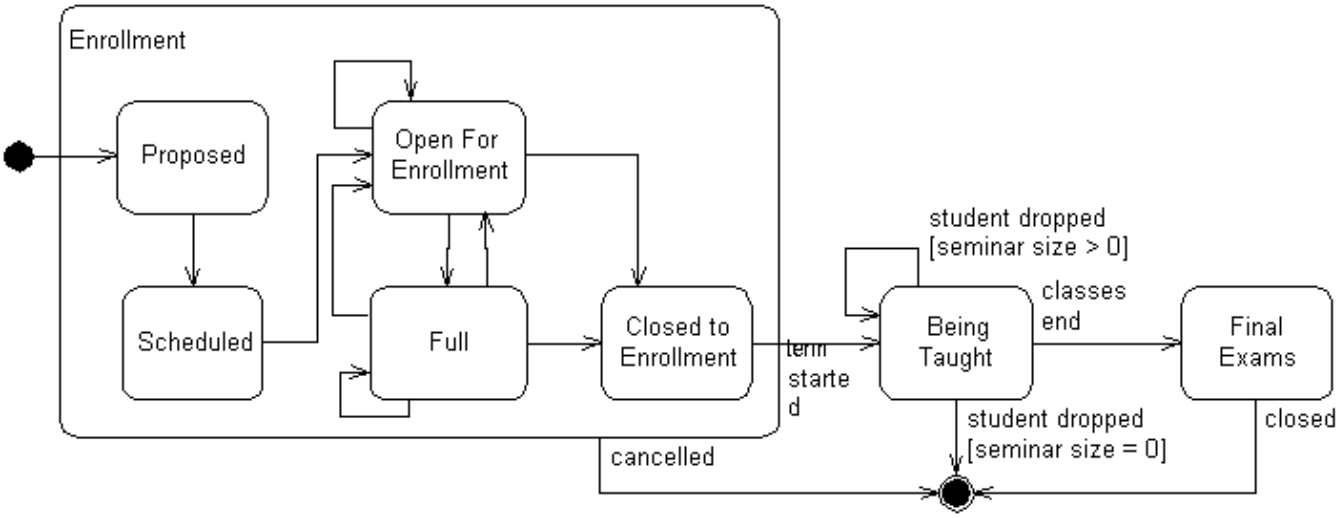


图 2 讨论班的完整的生命周期

3.2 聚集共同子状态的转移

图 2 中“取消”（Cancelled）这个转移是用来描述如何离开登记（Enrollment）这个超状态的。而图 1 中，每个状态都用到了“取消”这个转移。这样就简化了状态图。如果多个子状态共享一个进入转移或者一个离开转移，那么上述的方法同样可以采用。被聚集的多个转移如果有监护条件（Guard）和动作（Action）,那么它们都必须是相同的.

3.3 对非常复杂实体的状态图分级

尽管上述描述状态图的方法使用良好,但是所产生的状态图会变得非常复杂。想象一下在图 2 中，“教授中”（Being Taught）这个状态同样有子状态的话，状态图会变成怎样的情况。一种可选的方法是对非常复杂实体的状态图分级。例如，图 3 是顶级视图，而图 1 是种更细节化的视图。这样分级的好处是如果需要的话：

我们同样可以为“教授中”（Being Taught）这个状态提供更细节化的视图。

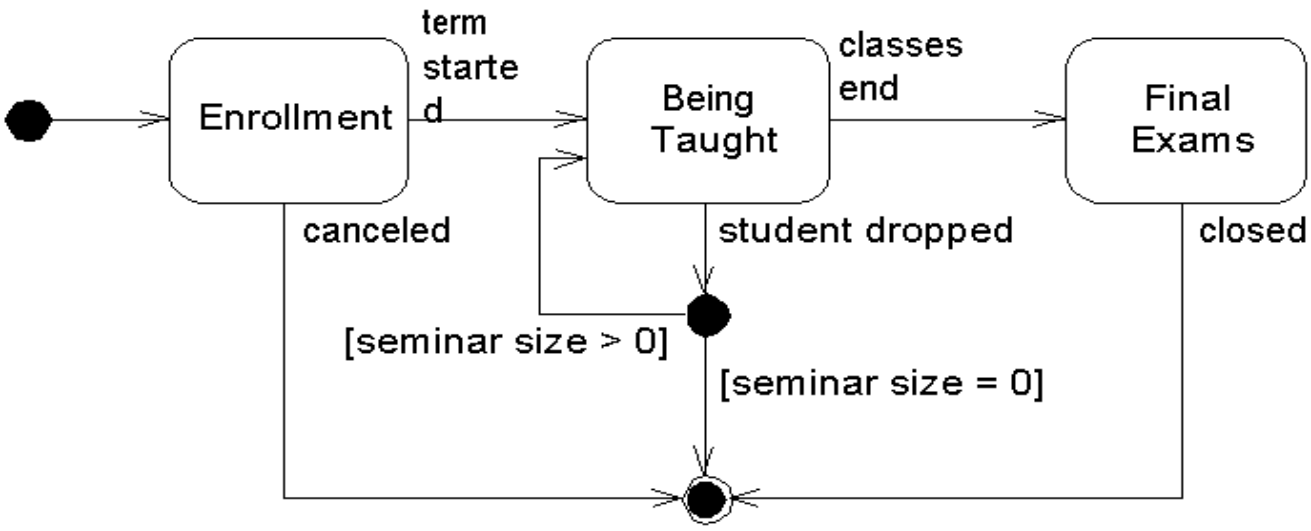


图 3　讨论班的顶级状态图

3.4 顶级状态图通常有开始和结束状态

例如图 3 这样的顶级状态图需要表示出被描述实体的整个生命周期，包括它的开始和终结。低级的状态图通常不需要都包括开始和终结状态，尤其是那些描述处于生命周期的“中间状态”的实体的状态图。

4 转移和动作

转移是由一种状态到另一种状态的迁移。这种转移由被建模实体内部或外部事件触发。对一个类来说，转移通常是调用了一个可以引起状态发生重要变化的操作（Operation）的结果,但是我们需要知道，并不是所有的操作调用都会引起转移的发生。动作（Action）针对于类来说，是指被建模实体所调用的操作。

4.1 遵从实际开发语言的命名规则来命名动作

图 1 中的动作遵从 Java 的操作命名规则，因为计划中这套软件系统是使用 Java 来编写的。如果用另外一种语言，那么动作名就遵从该种语言的命名规则。

4.2 使用平实的文体命名 Actor 的动作

UML 的状态图可以用来建立非软件实体的生命周期的模型，尤其是对于 UML 用例图中的 Actor。例如学生这个 Actor 很可能会有诸如“已录取”（Accepted）,“全日制”（FullTime）,“半工半读”（Half Time）,“本科”,“研究生”,“博士生”,“博士后”等各种状态，每个状态又会展示出不同的行为。当你为现实世界中的学生建模，相对于为软件系统中的学生这样的类建模，需要用较为平实的文体来描述状态间的转移，譬如用“退出讨论班”（Drop Seminar）,“缴费”（Pay Fees）来代替 dropSeminar()和 payFees(),因为现实中的人是做事情，而非调用操作。

4.3 只有对所有的进入转移都适用的时候才标明进入动作

在图 1 中，你可以看到一旦进入登记关闭状态（Closed to Enrollment）,通过进入/动作（Entry/Action）这个标签 notifyInstructor()(通知授课教师)这个操作被调用了。在实际程序的执行过程中，每次进入这个状态，notifyInstructor()就被调用。如果你不愿意这种情况发生，你就需要将动作绑定到特定的进入转移。例如，在“登记开放”（Open For Enrollment）这个状态中，在学生已登记（Student Enrolled）这个转移发生时，addStudent()(加入学生)这个动作会发生，然而对开放（Open）这个转移却没有 addStudent()发生，因为不需要在每次进入这个状态的时候都加入一个学生。

4.4 只有对所有的退出转移都适用的时候才标明退出动作

用退出（Exit/）这个标签标明的退出动作，是和进入动作的工作过程相似的。

4.5 只有当你想退出后再次进入相同状态的时候，才考虑为递归的转移建模。

递归的转移的两个端点是同一个状态。一个重要的含义就是这个实体退出并且进入相同的状态，从而和进入/(entry/)或者退出（exit/）相关联的操作都会被自动调用。

对于“登记开放”（Open For Enrollment）这个状态，每次进入该状态，讨论班的人员数目都会被记录。这种情况就是本小节里面的一个实际的例子。

4.6 用过去时态来命名转移事件

图 1 中的转移事件，如分班（Seminar Split）和取消（Cancelled）,就是用过去时态的。因为事件发生在它所引起的转移之前，过去时态就反应出转移是事件的结果，

4.7 将转移标签放靠近产生它的状态

在图 1 中，无论转移标签放置在哪里，状态图都是复杂的。在该图中分班（Seminar Split）和学生登记（Student Enrolled）这样的标签是尽量靠近产生它的状态的。而且尽可能地左对齐或右对齐来使之靠近产生它的状态。

4.8 根据转移方向来放置转移标签

为了更加清楚地表明标签和转移的对应关系，下面是一些建议的方法：

.....从左到右的转移线，标签放在线上.

... .从右到左的转移线，标签放在线下

.....从上到下的转移线，标签放在线右

.....从下到上的转移线，标签放在线左

监护(Guard)

监护是一种条件，只有它成立，转移才能发生。

5.1 监护条件不可重叠

从一个状态发生的多个类似的转移，它们的监护条件不可重叠。例如 X>0,X=0,X<0 这样的监护条件就是相容的，而 X>=0,X<=0 就不相容，因为它们的条件重叠，当 X=0 的时候，不知道该如何发生转移。在图 2 中的“教授中”（Being Taught）状态的“学生退课”（Student Dropped）的多个转移的监护条件就是不重叠的。

5.2 引入接合点以使得监护条件显得集中

在图 2 中，我们看到由于学生退课（Student Dropped）事件的发生从被教授（Being Taught）状态中衍生出两个转移，而在图 3 中却只有一个转移，而这一个转移导致了一个接合点的产生（实心圆）。这样的好处是，两个转移在状态图上画得靠近了，使得人们容易看到这两个转移的监护条件是不重叠的。

5.3 监护条件不需要形成一个完全的集合

由一个状态所引发的转移上的监护条件有可能没有形成一个完全的集合。例如当发生大额资金存入的时候，银行帐户这个对象会从开放（Open）这个状态转移到需要验证（Needs Authorization）这个状态。然而对于小额资金存入这样的转移却不需要再建模来考虑（资金存入的转移是以资金额度作为监护条件的）。在这里，你是在遵从敏捷建模的简约模型描述（AM Depict Model Simply）的实践，你只是提供了有关的信息。当然关于小额资金存入这种转移是可以推断出来的。

5.4 监护命名要一致

在图 1 里面的监护名称“有空位”（Seats Available）和“无空位”（No Seats Available）,这样的命名是一致的，而 seats left, no seat left, no seats left, no seats, no seats available, seat unavailable 就缺乏一致性而不容易理解。

UMLChina 公开课

“UML 应用实作细节”

（2003 年 7 月 26-27 日，北京）

现在，UML、RUP、Rose 等概念已经传播得越来越广，书籍、资料也越来越多，决定在开发过程中使用 UML 的开发团队也越来越多。

在开发团队应用 UML 的软件开发过程中，自然会碰到很多细节问题：“我这样识别 actor 和用例对不对？”、“用例文档这样写合适吗？”、“RUP 告诉我该出分析类了，可类怎么得出来啊？”、“先有类，还是先有顺序图啊？”、“类怎样才能和数据库连起来啊？”...许许多多的细节问题，而每一个细节都和背后的原理有关。

本课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，结合大量练习，阐述如何（只）使用 UML 里的三个关键要素：用例、类、顺序图来完成软件开发。使学员自然领会 OOAD/UML 的思想和技术，并对实践中的误区一一指正。整个过程简单实用，简单到甚至可以只有一个文档。

详情请见>>

误用例：带敌对意图的用例

Ian Alexander 著，林善茂 译

查看评论

人类甚至在冰河时代的篝火旁，争论捕获一头毛犀牛的危险性时，就已经开始分析不利的情况：如果在它中矛前转身向我们冲过来怎么办？最近更多的情况是：如果黑客们发动了拒绝服务攻击（DoS）怎么办？现代系统工程师能够通过使用误用例（misuse case）——用例的负面形式——来证明和分析这些情况。一个误用例是一个简单的用例，从一个带有敌视意图的角色的视角来看系统的设计。对于用例来说，误用例有许多非常有意思和有帮助的应用和影响。

获取安全需求

由于人们以及他们造出的负面代理（例如计算机病毒）对系统造成了真正威胁，因此存在安全的需求。安全与其他所有的专业领域不同，有些人是故意地造成威胁，以毁坏系统。在系统设计时，部署用例和误用例来建模、分析情况，可以帮助预防威胁，提高安全性。

某些误用例发生在非常特别的情形，其他的则频繁地威胁系统。比如，当汽车停在一个未被人注意的地方时，最有可能被盗；而一个 Web 服务器在任何时刻都有可能遭受到拒绝服务攻击。你可以递归地开发一个误用例和用例。如果需要的话，从系统到子系统，甚至更低。较低级别的用例，能够突出较高级别时未考虑到的方面，有可能加强其他的分析。在任何方面，该方法都提供了丰富的探测、理解和验证的可能性。绘制代理和误用例能清楚帮助开发者，将注意力集中在特定情况下的元素上。

让我们将图 1 中的内容和国际象棋或围棋进行比较。一个团队的最佳策略是提前想好对方团队的最佳的移动和行动方式，并加以阻止。在图 1 中，用例出现在左侧，误用例出现在右侧。误用例的威胁在于盗车贼，用例的参与者是合法的驾驶员；误用例的参与者是盗车贼。如果到车贼能够偷到汽车，那么驾驶员开车的自由就存在风险。驾驶员必须能够锁车——一个最初的需求——来预防威胁。这是最高层次的分析，当你考虑到车贼的反应时，下一层次开始了。如果他/她撬开车门，短接了点火线，这就需要另外的预防方法。比如锁定发动机。显然一个单纯硬件的设计最终将调用软件系统，我们可以按照同样的方式，对电子商务或其他商务系统进行更复杂的威胁性分析。

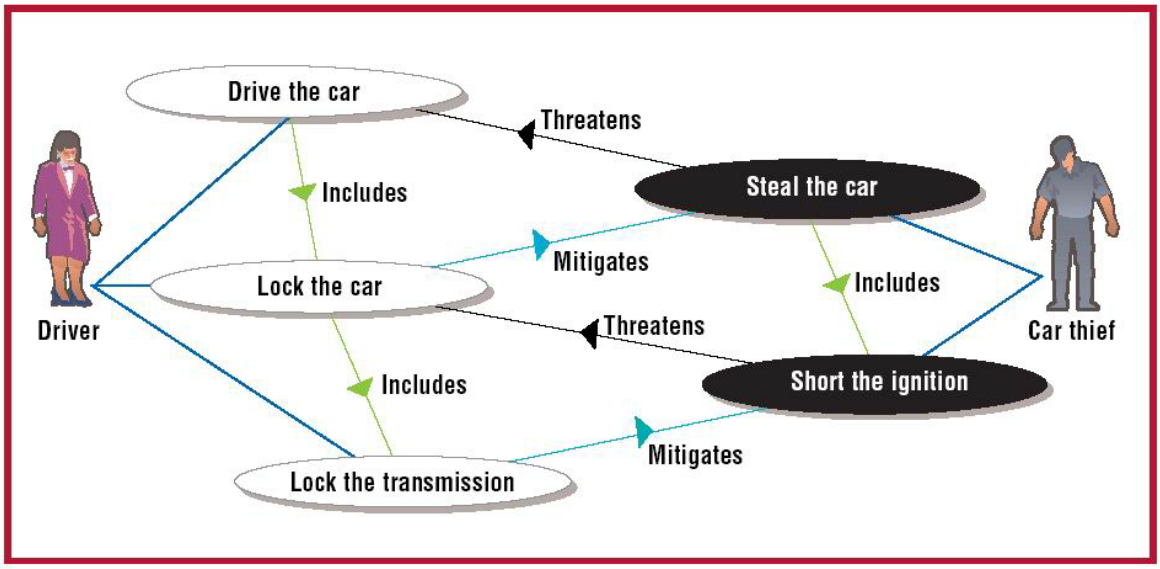


图 1 汽车安全需求中的用例和误用例。用例出现在左侧，误用例出现在右侧

图 1 还显示了威胁与预防在参与双方之间形成一个平衡的 Z 型模式。这个“游戏”表现在开发过程中的调研循环风格中。用例和误用例都可以补充包含各自类型的实例，但是，相反类型实例的关系不是简单的包含。相反的是，误用例以失效的方式威胁用例，适当的用例能够预防已知的误用。

当你知道了预防方法后，你可以将用户的需求（包含误用）和系统限制（比如费用、重量、尺寸和开发进度）结合起来，当你不知道预防方法时，你可以采用自顶向下的形式，同时进行两方面的开发，分析用例/误用例。你可以识别、学习、构造原型、计算，选择预防方法。当预防需要新的子系统或部件时，新的设备通常会反过来造成新的威胁类型。你可以分析这些威胁，来评估将来对策的必要性。由于设计选择的明确化，系统需求变得更加明确，分析和设计变得缠绕在一起。

预防措施极少压制安全威胁。窃贼通过未被怀疑的访问方式凿开锁，进入系统。然而，部分预防还是有用的，因为它们在允许的成本内增加了防护。压制所有的威胁，是期盼中的想法，但不能列入需求中。比如，有些驾驶员在短时间离开他的汽车时没有关闭发动机。设计者能够对这些误用进行防护吗？很明显有很多图 1 以外的实例需要考虑。

从失效实例中获取安全需求

失效模式影响分析和相关评估安全威胁的传统技术。他们将可能影响的原因相关联，作出假设并采取措施，计算可能性，引出一个系统足够安全的断言。然而，分析依赖于对可能的失效模式的精确识别。对于未考虑到的失效，不采取任何措施。因此，误用例分析，寻找、识别系统失效的可能原因，这样的技术能够对系统中似是而非的威胁进行失效模式分析，满足需要。

Karen Allenby 和 Tim Kelly 描述了一个方法：通过部署一种类型的用例，捕获和分析飞机引擎功能性安全需求。由于功能向导是与系统需求紧密联系的，他们发现了一个需求：从一个已知系统功能的继承向导。然而，他们并不建议与用例相关的负面代理。他们的做法是将失效及其原因，类型，影响和可能的预防措施列成表格。预防措施经常包含在子系统中，这意味着一个回归分解。然而，因为他们的用例，描述了潜在的灾难性的失效以及影响，将其称为误用例或失效用例是合理的。

安全需求场景没有必要包含人为代理（虽然可能有不满职工的破坏或是恐怖行为）。消极的代理（如车闸）或是一种单调的外部力量（如恶劣的天气。当驾驶员在冰雪覆盖的路上失控时），通常导致一个安全相关设备的失效。天气作为一个扩展代理来控制刹车是非常危险的。这个方法捕获了一个易于理解的外力，来强调在不同天气情况下控制的需求。因为人的语言含意丰富，所以以这种方式来表达需求是明智的。

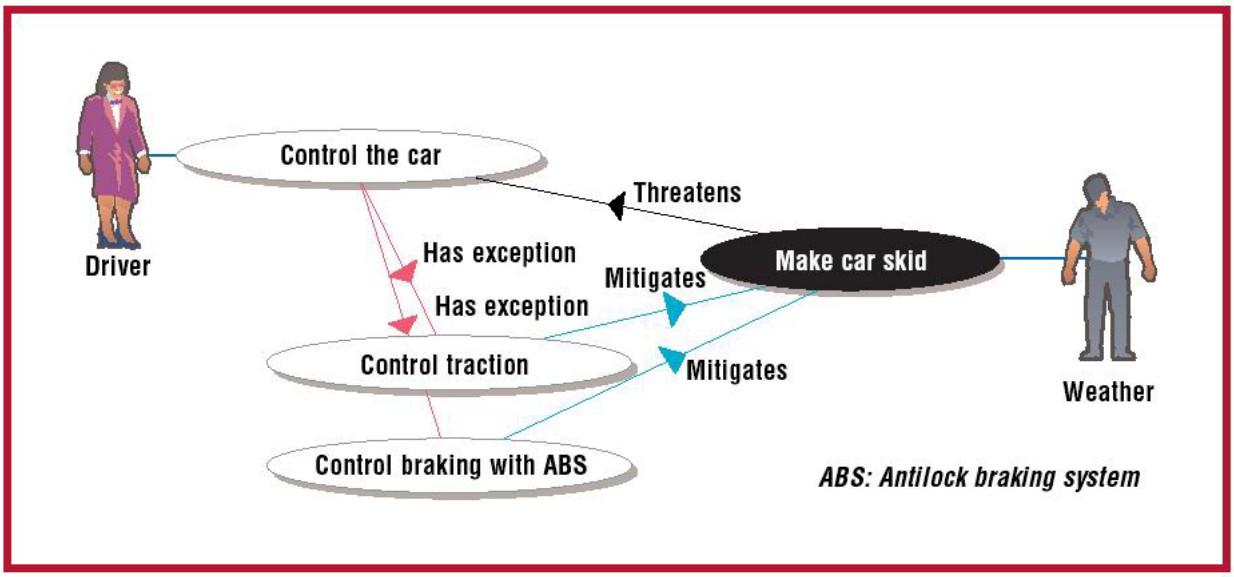
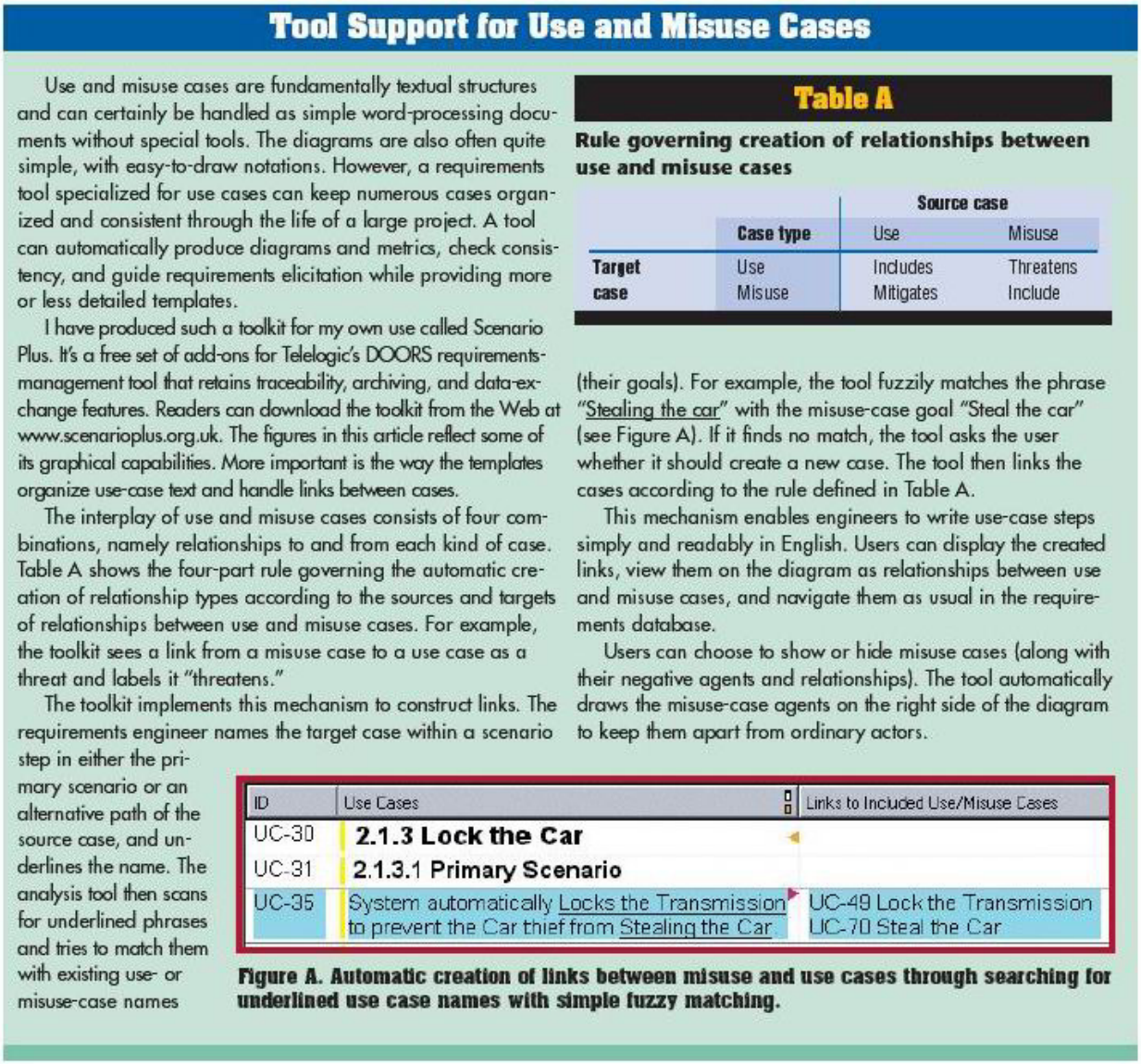


图 2 通过用例和误用例来捕获和分析汽车安全需求。天气是负面的代理

误用例能够以子系统功能的形式，帮助捕获恰当的解决方案，比如在牵引和自动刹车控制中，这些功能通过细致的程序化的反应来处理异常事件（比如刹车）。这些反应，满足了异常处理情况模块的需求。这些情况可以形成较大用例（如“控制汽车”）中的异常处理部分，也可以放在异常处理用例中，正如图 2 所描述的，与新的用例相连，捕获异常。

一旦你定义了这些异常处理用例，你就不需要误用例，除非是想证明设计决策的正当性。在复审时，为了项目设计元素和需求不被推翻，正当性证明非常重要，特别是当时间和费用超出预算时。有必要的话，你可以部署用例工具来很容易地展示或隐藏误用例（参见“用例和误用例的支持工具”）。



Tool Support for Use and Misuse Cases

Use and misuse cases are fundamentally textual structures and can certainly be handled as simple word-processing documents without special tools. The diagrams are also often quite simple, with easy-to-draw notations. However, a requirements tool specialized for use cases can keep numerous cases organized and consistent through the life of a large project. A tool can automatically produce diagrams and metrics, check consistency, and guide requirements elicitation while providing more or less detailed templates.

I have produced such a toolkit for my own use called Scenario Plus. It's a free set of add-ons for Telelogic's DOORS requirements-management tool that retains traceability, archiving, and data-exchange features. Readers can download the toolkit from the Web at www.scenarioplus.org.uk. The figures in this article reflect some of its graphical capabilities. More important is the way the templates organize use-case text and handle links between cases.

The interplay of use and misuse cases consists of four combinations, namely relationships to and from each kind of case. Table A shows the four-part rule governing the automatic creation of relationship types according to the sources and targets of relationships between use and misuse cases. For example, the toolkit sees a link from a misuse case to a use case as a threat and labels it "threatens."

The toolkit implements this mechanism to construct links. The requirements engineer names the target case within a scenario step in either the primary scenario or an alternative path of the source case, and underlines the name. The analysis tool then scans for underlined phrases and tries to match them with existing use- or misuse-case names (their goals). For example, the tool fuzzily matches the phrase "Stealing the car" with the misuse-case goal "Steal the car" (see Figure A). If it finds no match, the tool asks the user whether it should create a new case. The tool then links the cases according to the rule defined in Table A.

This mechanism enables engineers to write use-case steps simply and readably in English. Users can display the created links, view them on the diagram as relationships between use and misuse cases, and navigate them as usual in the requirements database.

Users can choose to show or hide misuse cases (along with their negative agents and relationships). The tool automatically draws the misuse-case agents on the right side of the diagram to keep them apart from ordinary actors.

Table A

Rule governing creation of relationships between use and misuse cases

		Source case	
	Case type	Use	Misuse
Target case	Use	Includes	Threatens
	Misuse	Mitigates	Include

ID	Use Cases	Links to Included Use/Misuse Cases
UC-30	**2.1.3 Lock the Car**	
UC-31	**2.1.3.1 Primary Scenario**	
UC-35	System automatically Locks the Transmission to prevent the Car thief from Stealing the Car	UC-49 Lock the Transmission UC-70 Steal the Car

Figure A. Automatic creation of links between misuse and use cases through searching for underlined use case names with simple fuzzy matching.

用例和误用例的支持工具

由于安全性的需求，你可以递归地开发安全需求。从整个系统到子系统级别，或者更低级别。另外，自底向上或从内到外的方法也是可行的。清楚的误用例使领域专家能更容易、更精确地证实安全性需求的正确性。

设计、功能性和非功能性需求的互动

给出的例子描述了误用例如何与系统和子系统功能互动，但你也能看到一个误用例作为一种敌视行为，针对一个功能性目标，比如“偷车”。这些威胁传统上通过写成非功能性需求和标准管理质量来处理，比如“该汽车将被建造成能抵抗入侵（质量），定义为 STD-123-456”。

这些是否意味着用例定义功能需求，而误用例定义非功能性需求呢？这有可能，但只是在你以传统的方式看的时候。另外，一种了解误用例的方式是：设计者对威胁的典型反应是创建一个子系统（比如锁），来预防威胁。简而言之，可以说误用例捕获了子系统的功能（如图 3 所示）。

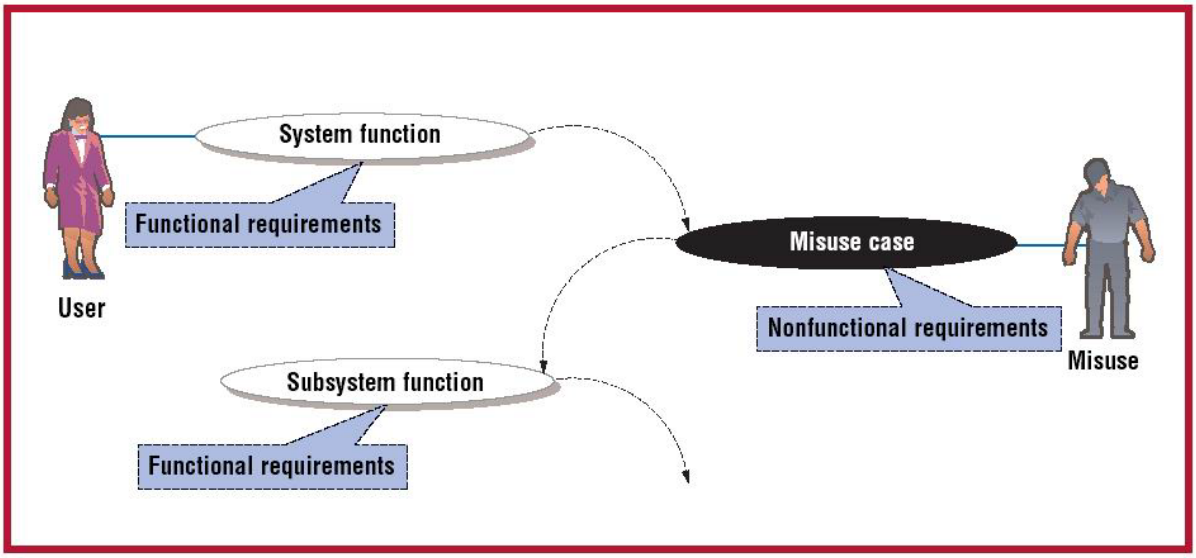


图 3 用例和误用例在功能性和非功能性需求的互动

如果你接受其差别的话，用例和误用例能真正地帮助捕获功能性和非功能性需求。然而，表 1 清楚列出在不同类型需求之间的动态影响，一个用户的非功能属性相反是另一个用户的子系统的功能。如果这样的话，或许用例能够覆盖所有类型的需求，甚至（正如某些人声称的）代替它们。

Table 1

Applicability of use and misuse cases for eliciting different types of requirements

Elicitation through	Use case	Misuse case
Functional requirement	Important mechanism	Useful, but indirect
Nonfunctional requirement	Possible	Important mechanism

表 1 用例和误用例在捕获不同类型需求方面的应用

获取“-ility”需求

误用例能帮助写出非功能性类型或工程师经常称为“-ilities”的质量需求，如：可靠性、维护性、可移植性等等。

可靠性（Reliability）、可维护性（Maintainability）和可移植性（Portability）

尽管安全威胁直接产生于真实的负面代理，但你能捕获和分析可靠性需求，因为引起威胁的代理并不是很智能化。这样的代理包括人为错误，传感器错误、设计错误（比如软件的 bugs）以及通信连接的冲突，这些能导致软件崩溃和其他类型的错误。

维护性和移植性也能从用例/误用例中受益。这里，负面代理可以是一种不可变化的设计或是一个独立的连线设备。这些简单的例子描述了—与“用例只能发现功能”的观点相反—用例/误用例能够被应用于许多类型的需求。

其他“-ilities”

你可以将误用例方案应用于可用性，当一个初学的操作者对用户界面感到困惑时，会成为一个负面的代理。你也可以将这个方法应用于硬件方面，如可存储性（storability）和可运输性（transportability），来解决寒冷天气的威胁，和对精细部件的粗暴操作。

这里，简单就是一个优点。误用例包含基础的情况，影响许多类型的系统。如果误用例只在复杂的微积分中表示，那么它们的应用就有局限性。

虽然我不提倡为所有可能的需求，盲目地创建几百个误用例，特别是当挑战的问题将来会众所周知的时候。技术看起来是可广泛应用的，以捕获和证实不同类型的需求。这是否意味着你可以建立用例模型，以代替写非功能性需求和限制呢？临时应急的时候是对的。有些事物可以容易地用几个词来描述——系统必须在两年内交付，单位成本不能超过$250，失效操作的间隔时间至少必须 10，000 小时——并不是完美地写在场景中。虽然通过场景来思考捕获限制因素和非功能性需求通常是有帮助的，但他们通常更好地总结为陈述。

获取异常

一个异常处理的用例能够描述设计中系统如何响应一个不希望发生的事件以防止可能的灾难性失误。响应能导致正常操作的恢复，或是一个安全的关闭，正如当火车收到一个危险信号后停了下来。误用例分析是避免异常的一种方式。有时一个误用例的场景值得详细写下；其他时候，只要通过误用例的名字就能识别系统需求中的问题。

异常类和引发误用例的负面角色之间存在清晰的关系。这些类属于异常的类，命名简单，通常情况会导致系统崩溃。你可以生成备用的异常场景来捕获需求，防止系统出现已有异常清单中的失效情况。然而，这样的清单太少了。

你也可以部署一个简单的需求模板来捕获异常。好的模板能帮助捕获和证明需求，这仅仅因为他们能提示我们要问的问题，如“这里是否有可移植性的需求？”，以此来捕获异常，可以根据用例中的所有场景来提问，如“这里可能出错吗？”。这种方式是有效的和普遍的，但不能保证找到所有可能的异常。

对于每个模板标题或误用例，都有一些需求，但是如果只找到一项需求—或者如果你确信没有需求—这种方法仍是值得花时间的？换句话说，一个模板，比如一个误用例暗示着一种询问方法。

然而，与简单地通过模板或考虑异常相比，采用误用例来设计威胁和负面代理有时是一种功能更强的技术。这有很多理由：

- 将问题从正常使用转换到误用，开通了一种新的观察方式，帮助找到可能遗漏的需求。
- 在误用例中提问，“谁想要其出错？”和“他们可能会如何操作来使其出错？”可能引导有系统地找出异常，通过使用场景结构自身作为向导，这比普通寻找异常的方法，带有更多特定的意图。

◆ 将捕获作为一种游戏，使得寻找变得有趣，并为其提供一个算法—“团队 1 比团队 2 在最佳移动（策略）思考得更深入，反之亦然”。

◆ 提供一个可视的威胁和预防的展示，使得受影响需求后面的推理立刻变得易于理解。

我发现模板和场景，在需求捕获中寻找异常方面，都同样有用。误用例提供了一种通入“圣杯”（需求的全部集合）的额外方式。

获取测试用例

任何场景都能导致测试用例。好的测试用例超出场景，来搜索大量的情况和异常。

误用例能清楚帮助、识别异常和失效模式，任何一个误用的实例，都值得采用其他方式来测试或验证。对于测试工程师来说，习惯想出负面场景是基本的技能。

用例/误用例分析的产物，可以提供测试计划，包括：

◆ 特定的失效模式（尤其是对于实时系统、嵌入式系统和安全相关的系统，特别有用）。

◆ 安全威胁（尤其对于分布式商务和政务系统）。

◆ 异常处理场景（一直有用，通常直接传向测试脚本）。

一个测试工程师能够浏览这些误用例，就好像真实存在一样，以保证更好的系统测试。一个质量工程师的目的就是提高配送系统的质量。

使用误用例的设计妥协

系统设计的一个重要元素是满足冲突各方用户的意见。由于每个设计选择都将增加新的用例和误用的可能性，因此，情况很复杂。设计者们必须在各种选择中作出妥协。

比如，门户网站的用户必须能够访问其所提供的服务。这个访问受到许多安全攻击的威胁，从游手好闲的不满职工的破坏到复杂的骇客攻击。安全自身也威胁系统用户，如果它是非常严格的，他将阻扰合法用户，导致他们寻找可替代的服务。另一方面，松散的控制容易使这些用户创造误用。图 4 描述了在这些用例中，增加“恶化”和“与其他冲突”左右为难的关系。

一旦设计者们识别到设计选项中的威胁、预防和可能的冲突，他们就能依照系统目标和需求，做出正式的决定。误用例，他们的关系以及最终部分用于正式设计中系统正当性的未实现的用例（也就是说，放松的安全控制）。他们都可以抛弃，唯一的风险在于，在下一次系统升级时，将会重复妥协争论。误用例，在系统设计和目的设计以及服务操作和维护的交替过程中，有了明确的角色。

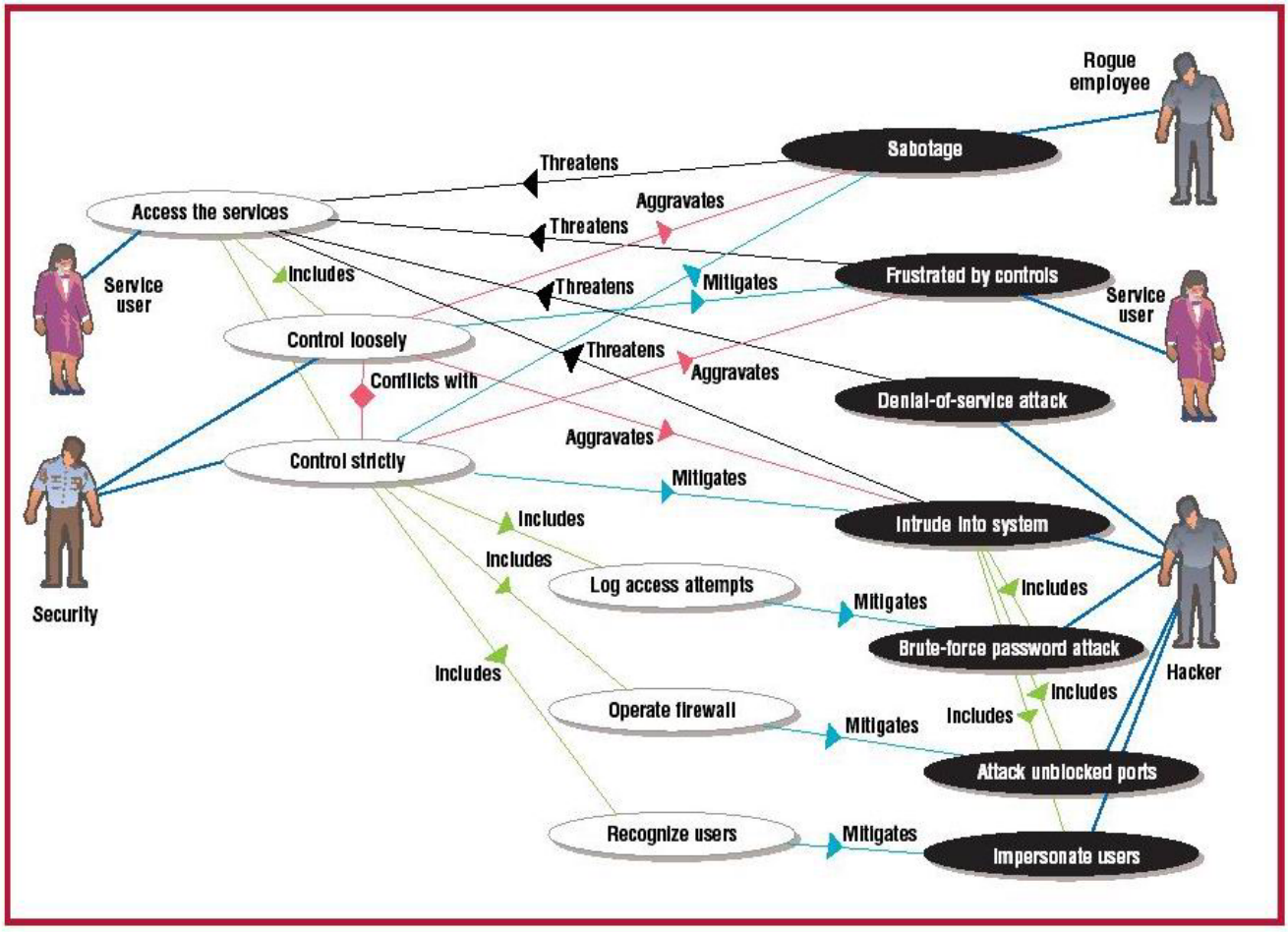


图 4 门户网站安全的用例和误用例

将用例和误用例投入使用

我在 DaimlerChrysler 的同事，和我一起研究用例的恰当形式，来帮助汽车控制软件需求的循环重用。用户不仅能够帮助软件设计，这是流行理由中的主要目的，而且在硬件和各种类型的界面上也有帮助。一个大型的系统包括许多子系统，比如汽车，必须分析子系统模块中连续的许多细节问题，以便处理重复的功能。用例模块也能帮助向不同的听众表达系统的目标和子系统的特征。

工程师可以在系统的任意级别上，应用用例和误用例。这种方法与软件工程生命周期中的递归分解和员工参与（需要循环的方法，邀请人员，通过提问合适的问题，来解决战略问题）这两种方法都非常吻合。用例和误用例帮助用户和工程师交流开发意见。比如，一个开发汽车软件的项目，用于控制娱乐和安全通告的声音，不仅必须考虑到驾驶员的娱乐需要—一个基本用例—而且还要允许交通通告的声音盖过娱乐的声音。确实，如果一个安全的通告的声音将盖过娱乐和交通公告的声音的话，软件系统就必须考虑子系统间的交互作用。

汽车越来越依赖于软件来完成以前依赖于硬件完成的功能。当一辆汽车有些单独的收音机、CD 播放器和预警系统，收音机和 CD 播放器不能干预预警系统，但流星雨的声音可以在预警系统的控制下，以使用户收到警告。因此，设计者开始尽量减少收音机和 CD 播放器的硬件部分，而采用软件来实现其控制功能。这个集成，不仅允许需要的新的行为，如流行音乐，而且为各子系统之间未知的交互作用打开通道。这样，娱乐子系统必须继承整个汽车场景并开发他自己更加细致的用例和误用例来处理。这个流程是个用例，与系统分解和信息隐藏相似。用例/误用例分析对于系统开发的每个阶段都有帮助，同时对于旁边的其他流程也有帮助，这些流程用于识别对象并定义经过他们的消息。

开始使用误用例

最好的开始方式是从一个小的、正式的工作室开始，在这里你和其他涉众识别可能威胁你的系统的负面代理。接着针对每个代理，对一系列误用例进行集体讨论。如果你已经有了用例，开始寻找可以威胁他们的误用例的方式。这将引导你创建更多误用例，或者他将引导出已有用例和误用例间的关系。接着你可以考虑如何预防误用例，看看他们如何上升。这将以用例的形式，引导创建新的子系统功能。接着进行冲突分析，你可能也需要考虑费用以及不同设计方式和可选子系统功能之间的利益。

一些误用的实例几乎不需要文档，其他可能需要一个完整的全过程的用例主场景。在这个流程中创建的子系统用例，通常还需要文档，需要检查对于他们自己的误用例是否是容许的。这需要几次的反复。在项目的后期，你可能需要考虑所有的误用例作为候选的测试用例，来保证所设计的系统按预期运转。

一个建立起方法的大项目可能需要准备包含其他标准的误用例的方针。这些方针应覆盖已应用的误用例，以捕获需求、分析、设计妥协和确认。

用例和误用例在分析过程中的互动，能帮助设计者更加有效的捕获和组织需求。功能性和非功能性需求捕获了类似的互动—以及设计决策—在设计流程中的全过程。考虑误用例将可能获得大量的可能使系统失效或者使项目开发增加时间和费用的争论。用例和误用例分析不仅允诺，而且对已存在的分析、设计和证实实践进行了补充。

参考文献

1. I. Jacobson et al., Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Boston, 1992.

2. G. Sindre and A.L. Opdahl, “Eliciting Security Requirements ments by Misuse Cases,” Proc. 37th Conf. Techniques of Object-Oriented Languages and Systems, TOOLS Pacific 2000, 2000, pp. 120–131.

3. G. Sindre and A.L. Opdahl, “Templates for Misuse Case Description,” Proc. 7th Int’l Workshop Requirements Eng.: Foundation for Software Quality (REFSQ 2001), 2001.

4. K. Allenby and T. Kelly, “Deriving Safety Requirements Using Scenarios,” Proc. 5th Int’l Symp. Requirements Eng. (RE 01), IEEE CS Press, Los Alamitos, Calif., 2001,pp. 228–235.

5. C. Potts, “Metaphors of Intent,” Proc. 5th Int’l Symp.Requirements Eng. (RE 01), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 31–38.

6. I. Alexander, “Use/Misuse Case Analysis Elicits Non-Functional Requirements,” to be published in Computing and Control Eng. J.

7. I. Alexander and T. Zink, “Systems Engineering with Use Cases,” to be published in Computing and Control Eng. J.

8. I. Alexander and F. Kiedaisch, “Towards Recyclable System Requirements,” Proc. 9th IEEE Conf. and Workshop Eng. Computer-Based Systems, IEEE Press, Piscataway,N.J., 2002.

9. A. Cockburn, Writing Effective Use Cases, Addison-Wesley, Boston, 2001.

10. D. Kulak and E. Guiney, Use Cases: Requirements in Context, Addison-Wesley, Boston, 2000.

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

中译本即将上市

样章先行提供>>

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

第一本 UMLChina 训练指定教材

清华大学出版社

RUP 的反模式

Julia Filho 著，沙萌、于宏光 译

查看评论

导言

反模式就是不做什么：有些行为、习惯或者方法似乎很有价值，但对于准备完成的事情来说却没有帮助。本文讨论了多种 RUP 反模式，收集来自 Black Diamond Software 指导 RUP 使用者在各种不同大小，人员组成技术环境和工业项目中的经验。本文讨论了每一种反模式的本质，以及怎样才能避免的措施。

反模式

1. 瀑布 RUP

“我们目前的迭代是需求，下一次迭代才是设计”

对那些一直遵循严格的瀑布开发过程的人们而言，瀑布 RUP 是最容易犯的错误之一。瀑布 RUP 是反模式的原因很简单：它不能帮助降低风险程度，而降低风险是基本的 RUP 原则之一。RUP 迭代式开发要求每次迭代应该是一个应用程序的“小型发布版”。每次迭代有小型的需求，设计，开发和测试环节，并且交出应用程序的一个可运行部分。使用这种方式，需求、设计、实施和测试的问题在每一次迭代中都得到“冲刷”，要求问题越早解决越好（问题越早解决其消耗的代价就越小）。

如果瀑布阶段没有转换成迭代，会怎样呢？功能块就会成为迭代，用例成为功能“片”的基础。通常 RUP 表明了用例的优先性，同时也说明了内在的风险决定用例怎样被划分到迭代中。

2. 准备-设置-RUP

*“公司说每一个人都应该遵循 **RUP**，所以让我们开始吧”*

决定采纳 RUP 作为方法学通常源于高层领导。CIO、CTO 或者其它的高层的技术领导决定所有新的项目必须遵循 RUP。然后每一个部门决定怎样将 RUP 应用到他们的项目中，于是他们购买该方法学，并且相信他们已经做好准备。第一个项目组在接受某些 RUP 培训后热火朝天的开始实施。他们尽可能的严格执行该方法学的要求，但是这好像有些难以容忍－RUP 太庞大了以至于他们深陷在 disciplines, artifacts, milestones 之中而迷失了。如果项目的完成日期迫近，而他们仍然陷入方法学的泥潭中，他们就会绝望，就会退回到他们原来知道的但是不是 RUP 的方式来解决问题。这个项目组从前有很好的意图，但是为了能够在最后日期前完成任务，最终退回到他们原来了解的最好的方式。

RUP 是一种相当大的方法，而且独自解决它是项艰难的任务。对于已经适应不同于 RUP 方法学的组织，例如瀑布模型，第一个 RUP 项目是一个巨大的挑战，超出典型的项目挑战。当它被完全贯彻下来，你就会发现采纳 RUP 的原因，才会发现 RUP 的功效，所以退回到原来的处理的方法是不能被接受的。

那下一步应该做什么呢？下一步要做的就是在整个组织范围内贯彻 RUP。尽力建立一个支持结构，目的是使项目组成员不要经常偏离他们努力的目标。你也许会发现你需要其他具有丰富 RUP 经验人的帮助，这意味着招聘一个具有 RUP 背景的雇员或者有从外部获取 RUP 帮助的渠道。其实你真正所需要的是决定怎样将 RUP 应用到特定的项目中，和怎样使用 RUP 的相关技术，例如用例开发等。有时一点指导会帮你少走很多弯路。

3. RUP－全部工具的总和

“我们拥有 Rational 的工具，所以我们已经做好开始的准备”

这是另一种“准备－设置－RUP”，是对工具的扭曲认识。在这种反模式里，项目组拥有 RUP 方法学和 Rational 的工具，他们已经准备好开始一个 RUP 的项目。问题是 RUP 不是所有工具的和。这些工具可以帮助方便实施 RUP 的很多活动，但是利用这些工具不能确保能够有效的实施 RUP。你知道准备应用的这些工具是做什么的吗？你知道你怎样做才能适合软件开发过程吗？举一个例子，如果你使用 Requisite Pro™ 来进行需求管理，那么你知道怎样保证需求信息被整个项目组共享吗？怎样将 Requisite Pro™ 正在收集的需求信息反馈给用户呢？知道怎样利用 Rational 工具是很重要的，但是理解这些工具所支持的活动以及这些活动之间的相关性也是必须的。简单的说，实施 RUP 可能会驱使你使用这些工具，而不是因为有了工具而使用工具。

要熟悉 RUP 的理念。要思考这些理念与你以前熟悉的方法学的理念有什么相似和不同之处。要熟悉 Rational Tool Mentor，这些指导为你使用 Rational 工具来实施 RUP 提供很多宝贵建议。

4. IT 是一座孤岛

“IT 遵循了 RUP，这就足够了，对吗？”

你所在的 IT 组织已经决定使用 RUP 实施一些项目了。项目组的成员都接受 RUP 的培训，而且拥有必需的工具，并且聘请了一位具有丰富 RUP 经验的专家，所以现在准备实施了，对吗？你是否考虑过 IT 外部因素对 RUP 实施的影响呢？RUP 的每一步，系统需求都是利用用例来描述的，最好是根据直接用户提供的资料得到的。但是你的用户做好准备花费时间来建立和审查用例了吗？他们知道什么是用例吗？你的 DBA 曾经接受过 RUP 的培训吗？他们准备好用迭代的方法接受数据模型/表变化请求吗？所以要让所有相关的成员知道 RUP 是怎样影响他们，知道 RUP 怎样利于项目成功，是很重要的。

5. 我们什么时候开始编码

“用例太耗费时间了，所以我们马上编码吧”

用例是花费时间的。的确。但通过不停的编码，编码，再编码找到“隐藏”的需求的办法就不花时间吗？在开发中出现的需求问题的频率又怎样呢？虽然建立用例也不能确保所有的需求被覆盖，也未必能满足所有用户的需求，但是使用这种方法比起“得到一点需求就进行编码”要高效的多，因为毕竟这种方法是在开发前进行需求分析的，而在开发中调整需求的代价则是巨大的。

项目经理比开发人员更渴望结束用例并开始编码，这并不奇怪。通常开发人员会说“现在已经拥有很多的信息了”，而此时项目经理则会检查他的日程，关心是否能够按时间完成项目。在建立完用例后，完成用户业务目标的用户/系统的迭代模式就存在了。这种用例模型除了可以作开发者的系统需求外，还可以是测试计划和培训材料的重要的输入材料。

6. 什么时候完全做好呢？

“我们要在开始时为所有的迭代做一个项目评估并且坚持将评估贯穿整个项目”

有多少次你被要求对你的项目做大致正确的评估，假设这种有漏洞的评估是基于非常有限的信息并且可能在项目的进行中发生巨大的变化，然后无论他们是多么的不正确，这些评估总会被保留下来？保守地说，在某种程度上，我们大家都是如此。

RUP 的目的是消灭这种做法，它要求项目要做初始的全局性评估，要求每个迭代应该在被执行前进行评估。开始当信息了解还比较少时，可以做一个“低分辨率”的项目评估，随着项目信息的了解深入，它可以进行修改。从每一次迭代评估中得到的知识都被输入到下一次迭代评估或者整体评估中。另外“时间盒”（time-boxing）可以用管理范围。这种方法提倡在预算时间内对完不成的功能进行删减或者延迟，而不是拖延预算的时间。

这种方法行得通吗？答案是，这种方法只有公司高层领导和用户都赞成 RUP 时才能实施，因为他们决定批准或者否决项目的预算、项目完成日期的延迟和项目的应用范围的变动。我们再次强调“IT 是孤岛”反模式，外部因素对项目组的成功影响巨大。

7. 食谱 RUP

“RUP 的规则在那里？我需要手把手的指导”

RUP 提供象食谱一样的方法了吗，“RUP 食谱”中每一个烹饪法都对应一个不同的项目？答案是没有。RUP 只是提供了指导，并没有需要遵循的固定不变的规则，因为任何一个项目都是不同的。特定的用户群，以及与目前存在问题相关的项目组的核心能力都是动态的，甚至是项目成员物理位置的变化范围都不是任何一本食谱能够完全描述清楚的。所以根本就不可能存在这样一本万能的食谱。

那么你怎样实施你的项目呢？第一步你需要花一些时间为你特定的项目定制 RUP，下一步为你的项目建立开发案例。这个工件在初始阶段建立，描述了怎样定制 RUP 以满足项目的具体需要。

8. RUP 是万能的

“我们遵循 RUP，所以所有项目管理、团队和组织的问题都会被解决”

RUP 为管理和实施项目提供了有价值的指导，但是它不是万能的。它可以告诉怎样才能成为高效的管理者、好的程序员、有用的团队成员或者高效的组织。例如，RUP 的项目管理向导可以根据 RUP 方法学解释怎样进行组织、计划和实施一个项目，但是项目经理需要知道怎样管理他们的职责，成员、完成日期等。RUP 提供设计和构建的向导，但是程序员需要知道怎样进行编程。对于已经采纳 RUP 的组织需要知道怎样整合和支持新的方法学。不是所有的组织、项目和团队成员都具有这样的能力。但是这并不意味 RUP 不能被采纳，而是意味着需要面临一个曲折痛苦的学习过程，或者说需要额外的帮助。

9. 过分的-强制的 RUP

*“我们必须完美和彻底地建立所有的 **RUP** 工件”*

这样做的后果是你永远不可能完成你的项目。

和很多其它的方法学一样，RUP 也包含很多的活动和工件，这些活动和工件在合理的时间范围内不可能被完成。RUP 并不是要求项目必须遵循它所定义的所有的活动和工件。实际上对于新项目，RUP 最初的任务是建立开发案例－用于确定所有需要建立的工件。这里可以根据项目组织情况，技能水平等不同的因素确定哪些工件是重要的，例如当进行设计的时候，你需要创建序列图，协作图，类图和状态转换图吗？也许没必要。也许你在每一个用例开发中真正需要的是类图和用来表示复杂逻辑的序列图；也许你正在开发一个工作流项目，描述状态转换的信息就是很重要的，这时候你需要状态转换图。总之，也就是说每一个项目都有自己侧重的工件需求。

决定在某一科目（disciplines）中选择或者排除哪些工件也许是很困难的。这里的关键是你要理解项目组织的需求，工件的阅读对象和全面的项目任务；另外需要理解工件需要的、能够作为迭代过程的因素。项目组在进行一次或者多次迭代后，需要考虑增加一些在初始阶段认为并不重要但是现在看起来十分必需的工件。

另外，创建工件虽然有些苦头但是并不是很困难的。每一个工件的细节层次以及信息数量都要和工件要解决的问题的复杂性相适应。例如，对一个需要花费 2 个月时间完成的项目，其业务用例不可能象企业范围的项目那么全面－如 SAP。

我们已经讨论了一些 RUP 的反模式，在遵循 RUP 方法时试图避免它们。我们已经讨论了克服或避免产生这些反模式的做法。需要记住的是采用新方法需要时间，同时尝试与出错往往是学习的最佳途径。可以在一些不是很重要的项目中实施 RUP，从中学习经验并且将其与它人分享。

领域建模

Paul Oldfield 著，陈明勇 编译

查看评论

摘要

领域建模的概念有很多形式，本文的内容只介绍其中一种，包括了基本概念，设计理念，用法与优点，如何用于实践的简单介绍。没有涉及更深的内容。

导言

不同的人对领域建模的认识不同，它是什么，有什么用，怎么去做，为什么应该做，在什么情形下要做。本文中都做了一定深度的回答。

从业务到分析模型：两种方法概览

在所有的面向对象系统开发中，如何从用例模型导出合适的对象模型的问题都值得关注，下面是两种方法的说明。

Robustness Diagram 方法

使用边界、控制器、实体对象等概念建立用例模型，起初有关系统行为的责任都被分配给控制器，随着开发的进行，这些责任被实体对象所承担，最理想的状态是控制器对象简化到以脚本的形式实现一个特定用例。

有关Robustness Diagram 方法可参考http://www.sdmagazine.com/documents/s=733/sdm0103c/0103c.htm

类－责任－协作方法（CRC）

领域模型和它应用时所绑定特定系统应相互独立，责任被分配给抽象的概念，当用例映射到领域模型时，应注意责任与所实现特定的用例之间的关系。

什么是领域模型

领域模型是有关一个领域的模型，他说明了企业如何经营它的业务。一个企业的领域模型也应该适合于这个领域中的其他企业。再追究细节，有很多不同的观点，这是因为模型和不同的方法学相关。

分析模型中的类和逻辑模型

领域模型往往是指在分析模型中的类图，确实模型应该包含带有属性和操作的类、以及相关的序列图。.这反映出类是为当前系统的特定用途而构造的观点，责任被分配到属性与操作，从支持将来变化的角度来看这种责任分配未必是最佳的，但序列图表明这种安排支持用例所说明的当前需求。一般说来，在属性与操作被设计、抉择时设计者不太考虑责任的变化，方法学未必有一项任务来保证责任分配到了合适的类上。以系统构造的序列图为指导，责任以类操作的形式被分配到特定的类。

RUP 观点

RUP 将领域模型描述为不完整的业务对象模型，关注于产品、运输、事件这些对业务模型很重要的东西。这个模型没有包含人所承担的责任，也可理解为词汇表。没有关注责任、人，实体等要素，实体被认为是在处理企业业务过程中工作者（worker）所操纵的哑对象。

CRC 观点

一方面有行为，一方面有事物，或者说实体，所有的系统都将把行为联系到事物，编程者起初关注在行为，在需要时引入实体，这在过程化编程中很典型，如何组织行为是主要的驱动力，实体从属于行为。

面向对象编程使用一种不同的思路，首先关注实体（对象），行为从属于实体，然而，需求往往用行为语言表达，混在一起的行为被分割，分配到不同的对象，从早期开始，”将责任分配到合适的位置”成为 OO 编程的信条，如果我们做好了这一步，构建和修改系统将会更容易和易于成功。参见“责任分配到合适位置”一段。

这些年中，在分配责任方面比较成功的一项技术是 CRC 讨论，举行这样的讨论有很多形式，都得益于使用卡片比在计算机上灵活，每个类写在一个索引卡片上，类名下面简短写出它的责任，讨论者扮演某种角色，以类如何交互、如何相互匹配来描述系统中的各种复杂行为..

不变属性

你或许会问，既然需求以行为方式表达，为什么我们不以这种方式来构造系统呢？

领域模型试图分离出系统中变动部分和非变动部分，在相当抽象的层次上，领域中的实体以及实体间的关系在相当长的时间内非常稳定（可能达数个世纪），我们假定这种关系在将来变化时仍将保持，如果程序的逻辑结构和领域内的稳定结构相匹配，那么程序的结构就可以适应变化，因此如果我们希望程序能够适应变化，它的结构必须和领域保持一致。

同样，领域中的实体所承担的责任也是长时间不变的，至少在基本的、抽象的层面。因此如果系统基于实体、实体关系、实体责任构建，程序的逻辑结构就可以具有一个稳定的核心。虽然变化的属性：比如系统架构方面、系统使用方面可能会发生变化。

系统架构方面，比如持久化、表现层、系统管理等在实现时会有很多方式，或多或少可以从领域模型的稳定部分分离出来，比如说，系统的非功能性变化会引起系统架构方面的变化，转换基于系统架构和领域模型的完全分离，一种系统架构可以适应不同领域的系统，一个领域模型也可以因操作环境的不同而使用不同的系统架构来实现。

企业需求变化时，系统的使用也将发生变化。如果系统结构适应用户行为的结构，当用户需求变化时，系统的结构也要相应变化，如果用户行为需求映射到领域模型的逻辑结构，当用户想得到不同的行为时，只需改变这种映射，用户想得到新的行为是可能会发生的，但只是在现有系统结构的合适部分上增加，而不是对整个系统进行重构。

如何领域建模：基本知识

谁、什么、哪里、何时、为何、如何做

谁：领域建模者对领域建模，他应该是分析人员，不同的人对这个角色称呼不同，因此下列角色应该是领域建模的合适人选:业务分析者、系统分析者、技术分析者、系统架构师，OO 分析人员。

什么：本文说明什么是领域建模，这一段说明如何进入领域建模

哪里：主要在领域专家的公司，但也需要其他的分析人员，参见：领域专家离开了

何时：任何时候都可以领域建模，最常见的是在业务用例和系统用例产生之后。

为何：参见从业务到分析模型：两种方法概要与优缺点分析

如何做：本段和另一章“如何对领域建模”对此进行阐述

责任分配在合适的位置

OO 方法论中一个很大的呼声是将责任分配在合理的位置，这个问题是个很重要的问题。广泛使用的 OO 系统必须保持灵活，但实际上，往往并非如此。**OO 系统保持灵活性的能力在于以不同的方式使用已有对象，因此如果你想以不同的方式使用系统，你所要做的就是让系统以不同的方式使用系统中的对象，然而，这依赖于一个假定，对象可以以不同的方式使用，如果责任分配合理，对象表达了正确的概念，它就可以以不同的方式使用，如果责任分配不合理，那么改变系统对象的使用方式往往需要改变系统内核。**

如果系统不需保持对未来变化的灵活性，那么责任分配问题就不需关注,建造这样的系统可能会容易些，但修改系统可能要花费更大力气。

这是一个真实的故事，我参加了一个项目，没有依照我的意见，选择了基于数据模型来设计 OO 系统，并没有考虑责任分配问题，尽管从客户的需求分析来看系统应保持对需求变化的灵活性。经过几次因需求变更引起的大规模重新设计之后，项目被取消了，没有向用户提交任何有用代码，很明显设计进程不能跟上需求变化的速度，永远如此。

抽象

让我们看看领域模型的基本构造块。

一个抽象是一个概念，一个和要建模的领域相关的概念，一个概念代表了领域中抽象或具体的实体，而不是关系和约束。

如果你熟悉 OO 编程将会认为将一个抽象作为一个类，这是一个好的起点，但是要注意领域模型中的一个抽象代表了一个类、类的实例，一组实例的集合。它也是一个占位符，代表了行为所需的辅助类，用于支持系统架构，比如持久化，表现，系统管理，对象生命周期管理，分布等，所有这些都要从领域模型中抽象出来

抽象在将责任分配到合理位置的过程中非常重要，因为抽象就是责任要分配的位置。

使用 UML 绘制抽象模型时，我们使用和类一样的表示，有些人会采用一个构造型，另外一些人则认为在领域模型中出现的类一定是抽象。

责任：知识，行为

责任从属于抽象，他们从属于正确的抽象，或者说我们把它放在合适的位置，抽象有两种责任：知识和行为。也就是说抽象具有保持某些信息、做某些事情的责任。

我们看一个例子：在航空运输领域，轨迹是一个抽象，它保持了对飞机的测量，在其他责任中一种是知识责任，它知道飞机属于谁，一种是行为责任，在一个特定时刻，它能告知飞机的位置，和预计位置。

在 UML 中没有对责任表达的直接支持，我们可以选择，我们的选择可能和建模工具的支持有关。

一种方式是有一个指定责任的部分，可能是两个部分，一个表示知识，一个表示行为，这些部分被命名，责任列在这些部分之中，我喜欢使用短句，一项责任一句，有些人可能喜欢将一句话命名表达单个责任，如果你的建模工具不支持例外部分，你可以用属性表达知识责任，用操作表达行为责任，根据命名惯例这个类是一个标记责任的抽象而不是属性或操作。

抽象、责任和关联

在前面我们描述了轨迹和飞机的相关性，轨迹和飞机有一个关联，在关联者之间的一条线表示一个关联。

图示很简单，在实际的领域模型中，轨迹和一个不同的抽象关联，因历史原因称为目标，飞机是目标的一个子集，作为一个普通的航行者可能会认为飞机在地上时不是一个目标，这依赖于你的领域边界，作为一个领域建模者将飞机作为目标的子集让我不安，因为将领域模型扩展到地面运动时将会无效，这种事情可能发生。

那么在领域模型中如何表达呢，我们说目标可能是一个飞机，飞机可能是一个目标，没有规则反对这样做，如果需要我可以说飞机是一个目标，目标是一个飞机，习惯上，IS-A 关系被实现为继承，很多语言不支持多重继承，很少支持相互继承，这不是我的问题，这是其他人的问题，当我带上设计者的帽子时才成为我的问题，但现在我只是讲出领域需求的实际。

控制器对象？不，谢谢

为什么轨迹不仅仅作为一个数据对象，使用其他的对象建立轨迹和飞机的关联，并拥有知识的责任，我将给出两个理由，从一个不那么重要的理由开始，

这样的对象称为控制器对象，在 OO 世界中被反对，因为一个轨迹必须和一个目标关联，即使目标是未知的（你可以认为是未定义的飞行物），每个轨迹必须有一个控制对象和目标关联，轨迹移动了，控制器必须跟随，如果控制器做很多工作，它所关联的工作也必须跟随，为避免这样，我们将控制器的责任分割，只将和轨迹于目标相关的责任保留，这样我们有一个轨迹，就有一个控制器，除非有人犯错，我们将轨迹和控制器合二为一来保证不会犯错。

这是 Robustness Diagram 方法的逻辑，用于从业务模型转向分析模型，实际上不使用臃肿的控制器对象有更好的理由。

另一个更简单的原因是我们说轨迹和目标相关联就可以了，如何实现是其他人的问题。

协作

一些抽象在一起工作（协作）来支持不只一个抽象完成的组合责任所定义的行为，例如我们希望得到飞机报告的位置，飞机可能能够提供，然而如果我们要得到预计位置，我们将这个责任分配给轨迹，但标示飞机属主身份的责任在飞机对象中，提供预计位置需要协作，需要标示身份和位置预计，需要飞机和轨迹的责任，必须协作。

在原理上，轨迹可以导航到飞机得到身份标示、提供位置预测，飞机也可以导航到轨迹请求它的预计位置，但这时支持此行为的消息序列是不清楚的，在领域调查中我们发现飞机可以和飞行计划协作得到一个预计位置，这个位置可能数值不同。因此这些协作和在设计时使用的协作图有不同的数量，在设计时给出一组对象如何提供行为的细节。.

不幸的是 UML 事实上不支持领域建模，我们必须用我们在设计时用的那些表示法，必须意识到我们现在所使用的图和在设计期使用的协作图是不同的，在领域模型中，我们选择一个“客户”使用轨迹得到预定位置，这只表示协作的一种可能用法，重要的是指出一个抽象到哪里寻求功能支持，哪里提供何种类型的责任。

规则与约束

看一个简单的业务约束：复式记帐法，一个账户上借出多少钱必须有另一个账户上有相同的贷入额，反之依然。如果我们将这项责任分配给一个抽象，我们有一个选择，可以说账户具有知道钱从那里来、用到哪里的责任，也可以说钱有知道它属于那个账户的责任，它被约束为在任何时刻属于且仅属于一个账户。

除了以上做法，我们可以在模型中置入一条规则，将它和钱与账户之间的关系关联，在这个时刻不把这个规则分配给特定的实现，在这个特定例子里，我更喜欢使用一些比关系多重性更强和更明确的表示方法，虽然实际上我们都是想说“钱属于且仅属于一个账户”。

当写规则和约束时，我建议使用两种形式：一种正规的语言定义（比如 OCL）和一种非正规的自然语言，正规定义很精确去除了含糊语义，但不易读，非正规描述易读但却有含糊和不精确的弱点，在描述细节时很罗嗦。

如何对领域建模：要点

需求捕获

需求捕获不是本文的重点，只是对其使用的一些简要说明，我发现考虑这三种需求是很有意义的：从用户和其他涉众得到的需求，它定义了系统要提供的任务和系统要如何被使用，这称为使用需求，最好使用用例来捕获，另外还有非功能性需求，可靠性、性能、安全性、可恢复性和其他一些东西，最后就是领域需求，使用需求一般用用例捕获，并产生分析和早期设计，刻划用户要见到的功能，驱动生成交互图（一般是序列图）来实现这些需求，非功能性需求产生设计的架构部分，驱动我们作出某些选择，领域需求产生领域模型，在作为用例抽象实现的交互图生成时使用需求映射到领域模型。

需求分成这三种类型是有价值的，因为捕获它们的技术和时机是不一样的，目前认为用例是捕获使用需求的最好方式，将非功能性需求分离出来作为一个补充的需求说明。我将举出一个例子说明为何我不喜欢这种方法，这个例子取自广为人知的（Kruthen(2000)）比如描述 ATM 机取款的事件流的用例，从用户角度是很完整的，然而它没有指明用户所取的金额应从相应帐户中扣除。这是一个很典型的场景，用例对描述交互是很好的，但对捕获因业务规则而引起的伴随行为上显得过于随意。【编注：此处作者可能对用例有误解，有关“扣钱”问题请参见“编写有效用例”、“有效用例模式”或 22 期的“用例，十年风雨”】

用户只是描述他所感兴趣的部分，而对在情景中的业务规则或不感兴趣、或完全不知道。

捕获业务规则的理想办法是和领域专家对话，将业务规则直接置入领域模型，构建领域模型的过程中将考察所有的实体和关系，用以发现业务规则。

领域专家离开了

有两种方法进行领域建模：一种是纯粹的形式，一种是不纯粹的形式。对系统分析的一种约束是：分析只能在领域专家在场时进行，换句话说如果我们在领域专家不在场时发现了某些事实，或者他无法确认这是领域内的实际现象，那我们所作的就是猜想、而不是分析，因此领域建模的纯粹形式是完全基于领域专家提供的事实，在领域专家不在时我们所能做的是对先前得到信息表达形式进行调整。

然而，几个分析员也可以在一起使用角色扮演的 CRC 卡进行讨论，研究责任、以及责任在抽象间的分配，这不是纯粹的分析，因为领域专家没有确认或否认这种责任分配。

实际上，两种方法相互补充，使用 CRC 确定或找出领域专家提供信息的不足，在 CRC 讨论中分析员往往可以获得足够的领域知识、正确地分配责任，领域专家往往不能在责任分配到抽象时进行推理。他更适于描述他所熟悉的企业中谁执行了何种工作。

使用无关性

领域建模的一个重要方面是对重用能力的支持，因此它没有对它所描述的实体、关系、协作如何使用作任何假定，因此它有助于在领域建模阶段忽略在系统开发时要关注的系统需求，分析员使用抽象术语和领域专家交谈，有那些实体，它们的行为，它们之间的相关性。不对这些建模元素如何使用作假定可以避免在假定错误时崩溃，在核心结构中不涉及这些假定的系统在假定错误时核心结构无需改变。

不幸的是我们为构建特定的系统收费，我们限制自己只作和要开发的系统相关的领域部分。

使用角色：单独考虑每个抽象

在领域模型中的每个实体，和它和其他实体之间的关系，思考一下其他实体对这个实体意味着什么，比如说你，可以是任何一种人，但对我而言只是一个读者，而我是一个作者，这是因本文引发的协作产生了这些角色。

使用角色的重要性表现在关联另一端的实体是任意的，只要实体可以充当这种关联中所指定的角色，这意味着我们可以单独考虑每个实体，以及它交互的角色之间的关联实体不再和关联中充当角色的特定类型的实体耦合。

尽管这对系统的开发没有多少直接的效益，它能帮助建模者理解领域和领域中实体，对分析模式的捕捉和应用也有价值。

“感觉良好”的因素

在每个新领域建模的案例里，都会有一个时刻充分理解了领域并能够构建领域的顶层模型，这一般包含 5-30 个类，你打量这个模型，然后说：这就是领域的真实体现无需渴望、无需拒绝，这一刻在你准备好时就会到来，但是你应该意识到这里面有感觉良好的因素，你只知道模型是对的。

这种因素在协作和为类分配责任时也存在，大多数情况下你需要先得到领域的顶层结构，其他的东西依照上下文关联到合适的位置，某些关联和协作并非如此，这些是通常的分析模式，在我们学习认知时经常发生，我们不能保证在所有的协作中都会达到这种感觉良好的状态，不管如何我们需要前进，采用我们已经拥有的已经足够。

控制范围

通常领域模型的开发是基础工作的一部分。我们为一个特定的任务而建模，而构建系统的任务是为了满足某种用户需要。我们来考虑需求被用例描述的情形，在用例中提到的实体集合是领域建模中初始包含范围。这些实体中一部分是同义的，或有子集关系。然而为了生成满足使用需求的系统我们必须实现一些在用例中没有涉及的概念，为了理解领域，写出顶层模型，我们需要进一步扩大实体集的范围，当然我们调查的资料可能被删到一个最小集合，被记录下来，我们建模是为了理解，一旦理解了一些材料就可以被丢弃，当然我们要选择一些结果以备将来使用。

如何领域建模：秘诀？

不要被标题所吸引，很多读者可能应跳过这一段

前置条件－后置条件映射

如果你对描述行为的正规方式有过研究应该熟悉前置条件和后置条件，它们是指在行为发生前一些声明必须为真（或一些事必须发生），在行为结束时什么会发生一般表达为那些变动发生。这与发生那些行为，这些行为如何运作没有关系。前置条件和后置条件之间的关系我们称之为条款。

最正规的形式，前置条件为真，行为发生，后置条件被满足，在行为中可能没有条件分支，若存在条件分支，行为可能被表述为多个条款，在各个分支上的守卫条件加入前置条件集。

这种深奥的行为文档化方式会影响领域模型，如果分析是以一种很正规的形式，模型中没有可选性和代表性。将行为组织到操作中是一个选择，这被看作设计而不是分析，当然，很多情况下应更灵活地将前置条件和后置条件的映射进行组织，使得这些行为代表一个单一的职责。应该记住的是为类定义操作是设计，而不是分析。

并行建模

在敏捷方法中的现代思想要求建模者在一起工作，共享思想，一般是结对建模，这样做的原因是在任何时候对待问题都有两种思路，可以得到立即反馈，然而一些领域建模的支持者、包括作者会给出不同的建议，当然是在特定的情况下。

如果有可能，在领域建模的早期阶段最好是两个或多个分析师独立开始建模，不要交流，而不是两个建模者一起工作，相互交流。当模型有了一定进展，将建模者聚集在一起分析模型的相似点和不同点，对不同点进行讨论，选择一种最好的做法。

这样做的根据是建立一个领域模型的各种想法需要时间来成熟，在建模者的脑海里组织分析，如果开始就相互交流，这些想法就无法形成，这时候，第二种意见没有多大意义，因为它也需要时间成熟，反而两种意见会相互干扰，伤害了那种有价值和效率的意见。

领域专家学习领域建模

开发一个领域模型的一个重要问题是对其进行验证，建模者读这个模型将意思传达给领域专家，然而这种例外的翻译是低效的。一种选择是建模者根据领域专家提供的专业知识建模时让领域专家学习如何读领域模型，这是有意义的，因为如果领域专家可以阅读模型，他就可以指出哪些知识是不完备的和不正确的。他将意见传达给建模者，建模者更新模型或者对领域专家作出解释。最后领域专家审阅模型并认可这是他知识的最好表达。

这是非常有用的，因为领域专家往往在建模者询问、学习、组织领域知识时可以得到对领域和组织的新的认识，表达领域知识时往往有一些平常没人会问到的问题，对这些问题的考虑使领域专家以新的方式考虑问题，不过我们不能依赖这种影响，有些专家确实对他们的领域非常精通。

哲学

领域模型是对业务和系统是什么的抽象，我们为一个领域建模时，通过开发一个可计算的逻辑模型使它更具体，也可以通过企业的业务对象模型使它更具体，面向对象的系统要获得灵活性，职责要由实现领域抽象的类负责，在业务对象模型中，这些职责被推出去让工作者（Worker）来实现，实体是哑的，工作者操纵实体完成各种灵活的行为，实体对象就和数据持有者差不多。

如果领域模型中的职责用工作者以外的方式来实现，领域模型就可以支持业务流程重组（BPR）,领域模型不是对 BPR 支持的好方法，但它可以来验证 BPR 结果是否涵盖了所有必须的职责。

优点和缺点

重用

领域建模没有包含在一个具体系统中如何使用的信息，它可以帮助解决在该领域中任何系统的开发，在这个领域中的企业可以重用它来开发系统或集成该领域中其他企业的系统，软件开发商可以在给该领域中的任何企业开发系统时使用领域模型。它可以辅助 BPR，在多个重整过程中都可以重用。

灵活性

系统构建如果围绕领域中稳定不变的部分构建核心的逻辑结构，就能够适应领域使用的变化，这是 OO 系统获得灵活性的基础。

扩展而不要改变

当然无法保证建模者第一次就可以得到正确的模型，常见的现象是在领域模型的使用中不断扩展，不要改变已形成的东西，领域中记录的事实就是领域中的事实，注意这是一种大方向而不是一个硬性的规则，如果模型中确实有不完美的地方，在后续的迭代中可以改变，模型过于简单，不足够严格，在后续的迭代中可以变得更严格，除了这些例外，在后续的迭代中的主要工作是确认和扩展已有的东西。

假定这样的情形，我在实际中并未观察到，一个领域模型包含了比任何一个领域专家多得多的知识，在专家系统中，系统给出的建议可能好于单个专家，因为它编入了很多专家的知识。

这个原理也可以假定到领域模型。但是要使系统满足各种角色的应用还是需要精雕细琢。任何一个模型在另一个场合都会有不尽人意的地方，这可在后续迭代中改善，不要试图追求完美，够用即可。

代价

领域模型可以在任何严格程度上构建，简单的不严格的模型可能是几个类和关系的草图，用于下一步工作的想法。或者系统开发的路线草图，最严格和详细的模型可以转换为应用，草图的代价最小，不管怎样对要做的工作的知识有个草图还是必要的。

一般来说，产生一个近似正规的模型的代价在单个系统开发中很难得到补偿，但所有的文档都会对后续工作有帮助，明智的做法是如果不确定模型是否需要重用就先做一个相对不正规的模型，在后续的迭代中扩展并越来越严格和正规。

总结

领域建模对系统建模者和业务建模者都有吸引力，在很多情况下，领域模型只是领域中一些关键概念的草图，可以用来传递和分辨想法，这种工作是价值的，一般来说付出越多，得到越多，本文没有完整给出作出选择的准则，但希望读者已得到足够的启示去作出自己的结论。

参考文献

Beck (1989)

Beck, K. and Cunningham, W. (1989). *A Laboratory for Teaching Object-Oriented Thinking*. Proc OOPSLA '89, pp 1-6

Bellin (1997)

David Bellin and Susan Suchman Simone (1997). *The CRC Card Book*, Addison-Wesley Object Technology Series

Cockburn (2001)

Alistair Cockburn (2001). *Writing Effective Use Cases*, Addison-Wesley Agile Software Development Series

Fowler (1997)

Martin Fowler (1997). *Analysis Patterns: Reusable Object Models*, Addison-Wesley Copyright 2002,

Object Technology Series

Kruchten (2000)

Philippe Kruchten (2000). *The Rational Unified Process An Introduction Second Edition*, Addison-Wesley Object Technology Series

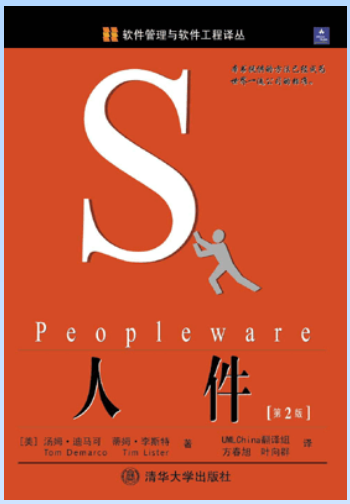
Warmer (1999)

Jos Warmer and Anneke Kleppe (1999) *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley Object Technology Series

Wirfs-Brock (1990)

Wirfs-Brock, R., Wilkerson, B., Wiener, L. (1990). *Designing Object-Oriented Software*, Prentice Hall

老板，别把开发人员当成牲口--《人件》热销中



Tom Demarco，Tim Lister

翻译：UMLChina 方春旭 叶向群

http://www.peoplewarecn.com

人件中文版网站

Frederick P. Brooks, Jr--《人月神话》第 19 章

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品。当人件第一版出版时，我写了一份评论，“我强烈推荐你买一份《人件》给你或你的老板，如果你是老板，那么为你部门的每一个人买一份，并给自己买一份”。这建议在 12 年后的第二版依然有效，并且更加热烈。

Mark A. Herschberg

我只学了办公环境的章节，就辞掉了原来的工作！

Joel Spolsky

我想，微软成功的原因之一就是公司里的所有经理都读过《人件》

Steve McConnell

由于本书第 1 版的赫赫声名，新版的《人件》是我不用看就会决定购买的少数几本书之一。

购买>>

角色建模——实用的系列分析模式

Francis G. Mossé 著，samying 译

查看评论

在进行面向对象分析时，我们经常遇到与角色有关的问题。角色是与某些概念(或类)所在的上下文有关的一个概念（或类）。举例来说，“公司”可能是某些特殊“产品”的“供应者”。“供应者”是一个角色。通过选择以下介绍的五个角色模式，所有的角色问题都能很容易地被解决。这五个模式是:继承角色、关联角色、角色类、泛化关系角色类、关联类角色。每个角色模式都综合了它对能力、灵活性和复杂性方面的考虑。这些模式一起，提供了对所有角色问题的完整解决办法。

教授OOAD和UML课程是一种非常有益的经验。参加课程的人感觉他们能够在以前没有从事的问题域中建模。当他们协助公司建模时，我了解到他们遇到的挑战大部分是：角色的建模。

角色问题是很常见的问题，但是至少有半打方法可以解决这个问题。我们把这种方法叫做角色分析模式。我们将它叫做典型角色问题的“预制”解决方案。每一个模式用简单的UML静态建模元素表示：类、继承、协作和协作类。

在你看了这篇文章之后，你应该感觉到你能很容易地识别一个角色问题，并且精确确定哪一种角色分析模式最适合这个问题。

1 继承解决方案

我们可以简单的将角色看作类型。例如：一家供应产品的公司被看作厂商。另一方面如果它购买产品，那么它也是一个已知的客户。这样它构成两种类型：厂商和客户。这个概念就产生以下提到的模型：

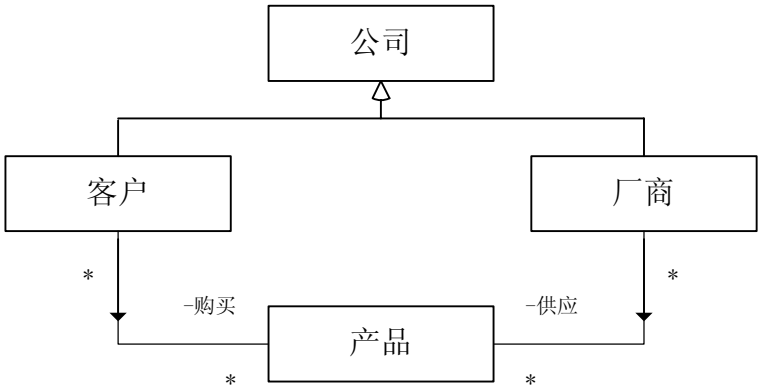


图1 继承角色模型示例

简单来说，这个模型告诉我们存在两家不同类型的公司:客户和厂商。客户购买厂商供应给他们的产品。因为使用了继承，这个模型也说明一家公司不可能同时是一家厂商和一名客户。最终确定的结果，这个公司要么是一家厂商要么是一名客户。

这样的模型是优是劣？它全都依赖你正在尝试解决的问题域。一些业务规则可能指出了某种限制；举例来说，不可能允许由客户来供应产品。其他的业务规则可能描述相反的事物:任何人随时可能购买或供应。我们挑选满足角色限制的最好模型。

上方显示的模型是继承解决方案的一个具体例子。现在，让我们一种抽象的方式来表示这个例子同样的解决方法——使用元模型。元模型提供一个一般性的答案，而描述具体例子的模型是元模型的一个实例。因为元模型给出一个一般性的答案，所以它被称为模式。

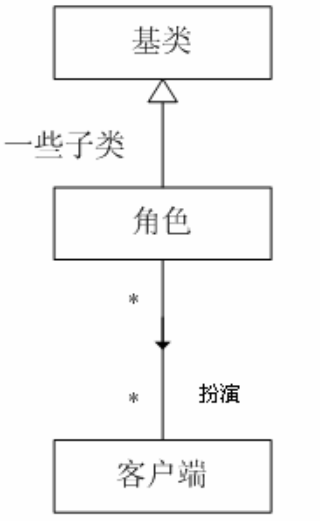


图 2：继承角色元模型

现在，继承元模型不允许多种角色被相同的基类实现。换句话说，一家公司不是一家厂商，就是一名客户，但不能两者都是。同样的，一个人不是一位牙科医生，就是一位房东，也不能两者都是！很明显，这种情形下有太多的限制，无法满足你面对的问题域的需求。

2 关联角色解决方案

仅仅根据与其它类的关系中他们是做什么的，基类有了不同的角色名字（公司，人，文件）

我们可以简单地说:在与他类的上下文关系中，一个类扮演着一个角色。UML关联提供关联角色，如下图所描述。注意厂商和客户是简单的关联角色。

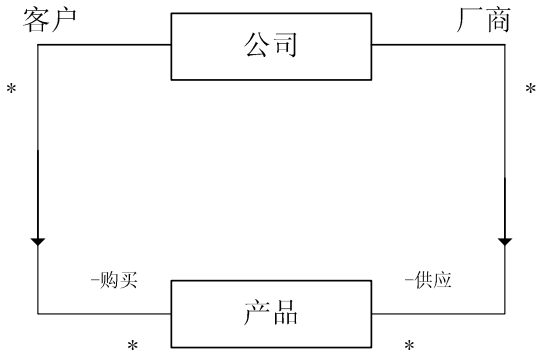


图 3: 关联角色模型示例

依照这种观念，我们开始这个新的模型，一家公司的角色是根据它做的事情而建立，而不是根据它是什么而建立。 图 4(在下面) 是第二个角色模式的元模型。

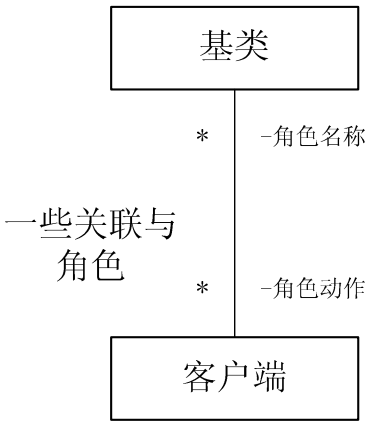


图 4: 关联角色元模型

图4显示，“基类”是象“公司”这样的类的抽象化，“客户”是产品的抽象化。允许有多个基类实例和客户类实例。元模型的角色部分用关联及其自身角色来处理。关联角色元模型是我们的第二个角色模式。

这个关联角色模式显然是处理角色的一个非常简洁的方式，你会惊奇地看见许多问题能用这样一种特征来解决。由于这种方法如此高效和有用，每一个建模者都应该非常熟悉关联角色。

下列的图表（图 5）显示了另外一个使用关联角色的例子。它也呈现了关联关系完整的记号法，其中包括角色。这张用UML记号法表示关联和关联角色的图片给读者一个熟悉UML的机会。

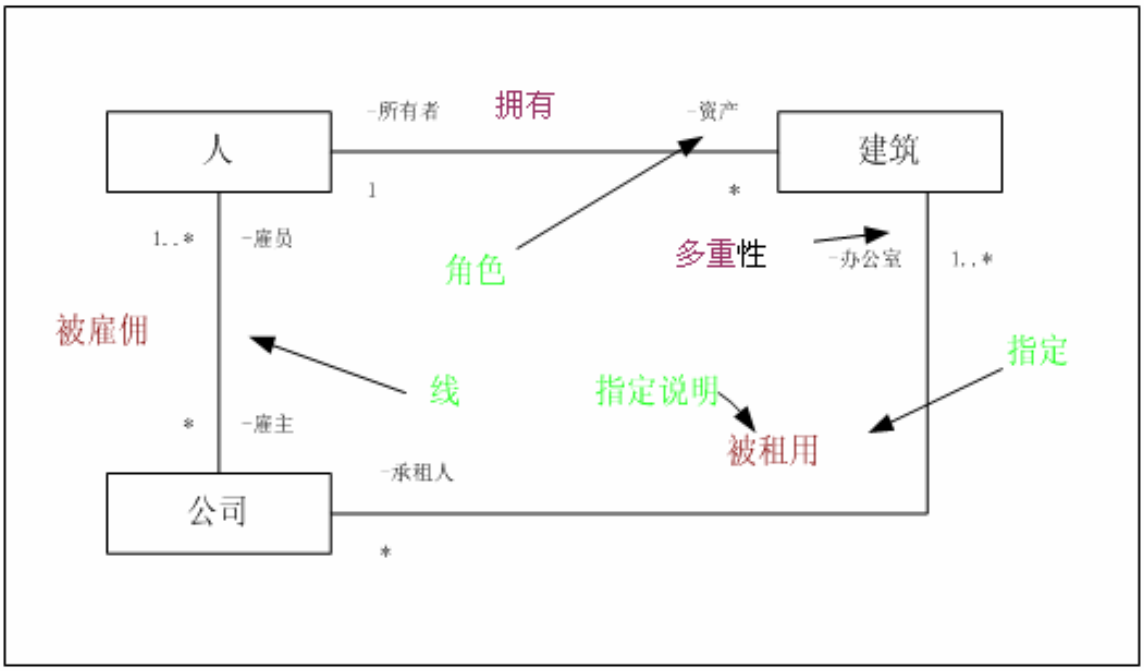


图 5：UML关联一般的语法（包括角色）

关联角色的一个明显缺点是他们不允许有任何的属性。回到我们的最初例子，如果厂商或客户需要特殊的属性，那么我们将必须提出类角色解决方式，因为类能支持属性。角色类是完成建模的又一个方法。

3 角色类解决方案

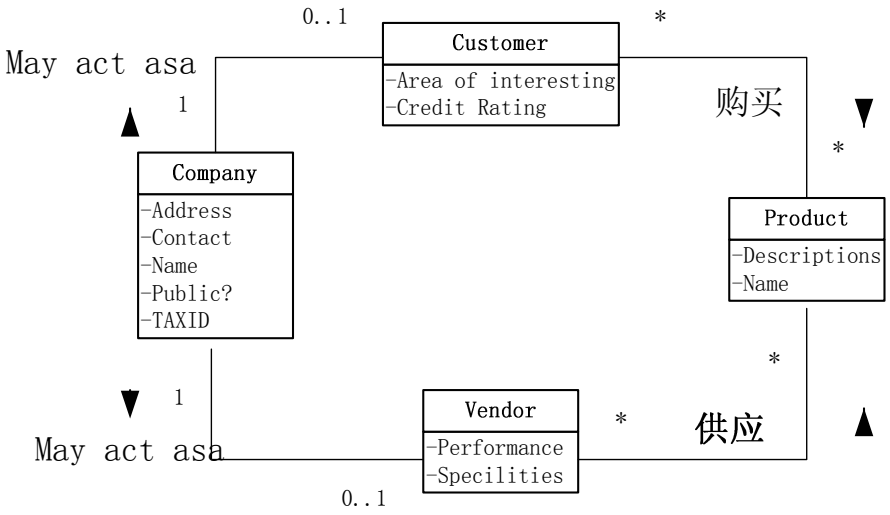


图 6：角色类解决方案示例

这个模型示例允许你连接任何一个角色到任何的公司。

这里用类来对角色建模，而不是用继承－－－在图一中，我们用继承来建模。

这种方法允许我们在需要的时候，对于零到多个角色提供适应性。这些角色有特殊属性或类操作。

图 7 显示了角色类元模型，这是我们的第三个角色模式。

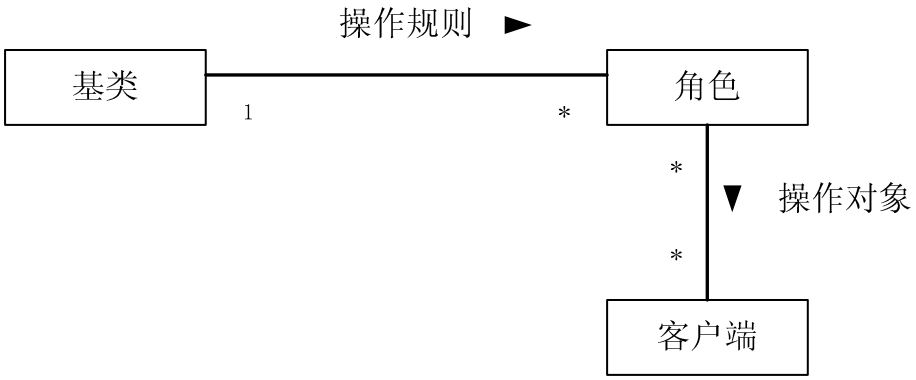


图 7：角色类元模型

这个元模型展现角色类像是客户和基类之间的一个人。在客户端来看角色，仿佛角色类就是基类本身。

在我们的早先例子（图 6）中，产品对象需要有客户对象的联系地址。而客户对象将会需要转向公司对象，然后从公司对象那里得到地址。委托调用时常会发生。相同的方法可能会在不同的类中被发现。(在图 8 中“getAddress” 和 “getName”)

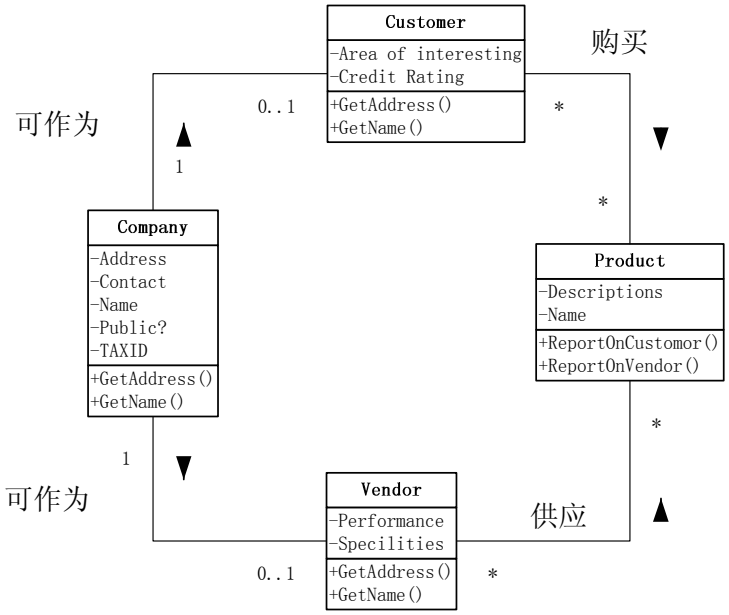


图 8：有操作的角色类模型

4 角色类泛化解决方案

图 8显示重复的类操作；举例来说，getAddress() 和 getName() 操作在厂商和客户角色类两者中被看到。重复要求泛化关系。 图 9显示了一个例子，连同一个附加的类角色，需要重点泛化的经纪人。

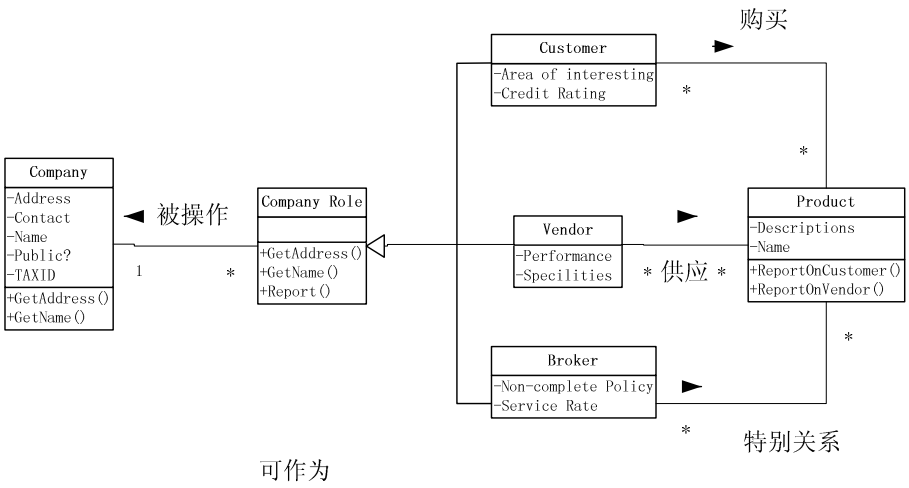


图 9：带有角色类泛化关系的角色类模型示例

凭借角色类泛化关系解决方案，所有的角色可能被建模并且随时动态地被指定为任何公司。支持特殊角色操作，但是不重复，这就需要感谢UML泛化关系特征。这些特殊角色操作只是在一个地方：公司角色泛化关系类。对于公司它是一个考虑所有代表的类。所有的操作（方法）中只有一个是需要的，通过继承，它将服务所有的类角色。图 10 显示了这种解决模式的元模型。

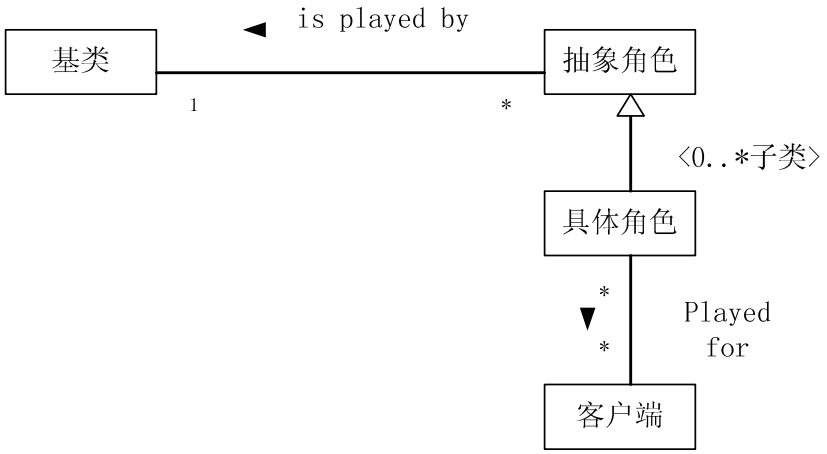


图 10：角色类泛化元模型

这个元模型描述角色类泛化关系解决方案，我们的第四个角色模式。

5 角色关联类解决方案

我们迄今看到的例子，所有的角色在分析的时候都被知道。当系统被实现的时候，仅仅是使用这些角色。举例来说，一个公司只能承担一家厂商的角色、一名客户角色或一个经纪人角色。 这些不同的角色是被动态地分配得到的， 但是不是被动态地构造出来的。我们能分配任何的角色到一家公司，但是它必须是已经存在于系统中的一个角色。用另一句话来说，它必须是一个已存在的类。

尽管对象可以被动态建立，但是类不可以被动态地建立。下面的例子(图11)显示了你怎样能够用对象动态地指定角色关系。

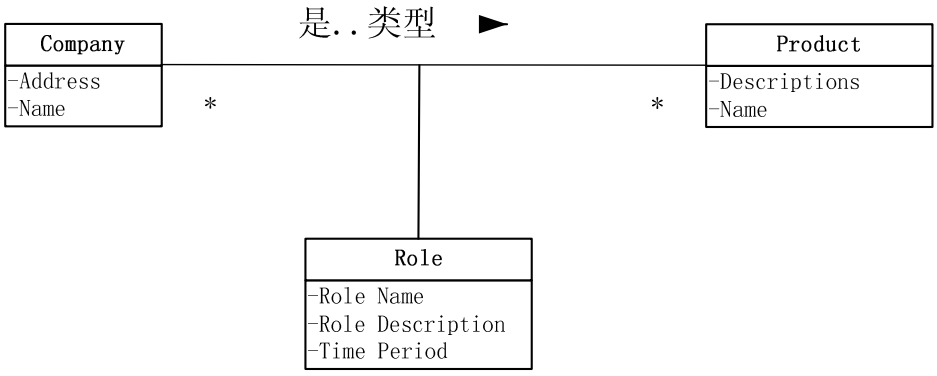


图 11：关联类角色示例

这个例子中，角色在需要的时候被建立。为了指定一家公司供应一种特定的产品，我们需要创建角色的一个实例——连接对象，从而把适当的产品和正确的公司相连。那个角色联结对象能被命名成我们希望的任何东西：厂商，客户，经纪人。

注意这个角色关联类的另外一个有趣的特征：时间周期。许多业务规则需要描述关系的时间周期。

关联类角色解决模式是柔性的和有力的。注意，是这种角色的概念被建模，而不是角色本身。这个模型比一些表面的角色模型更抽象。总的来说，你将会看见比较高层次的抽象化不但带来很好的简洁，而且带来很好的能力和柔性。

然而，注意更抽象的模型需要更好的对象管理方法。举例来说， 解决我们图 9关于厂商、客户、经纪人的问题，应用这种模型的人需要为每个角色产生三个连接对象。

对关联类角色解决模型，需要附加更多的东西。 举例来说，图 11 中所呈现的模型需要为同样的角色产生多个同样的联结对象。 举例来说， 20个厂商角色就需要20个角色连接对象，这些连接对象名字和描述属性值是相同的。我们再一次遇到重复。重复就需要泛化关系。

这个泛化关系不能够使用继承，由于需要被用在和类相对应的对象上。 图 12 显示了一个解决方案，用通用的和可重用的角色类型概念来建模。作为一个对象，每个角色类型可以动态建立，然后连接到属于他的类型的任何数目的角色连接对象上。

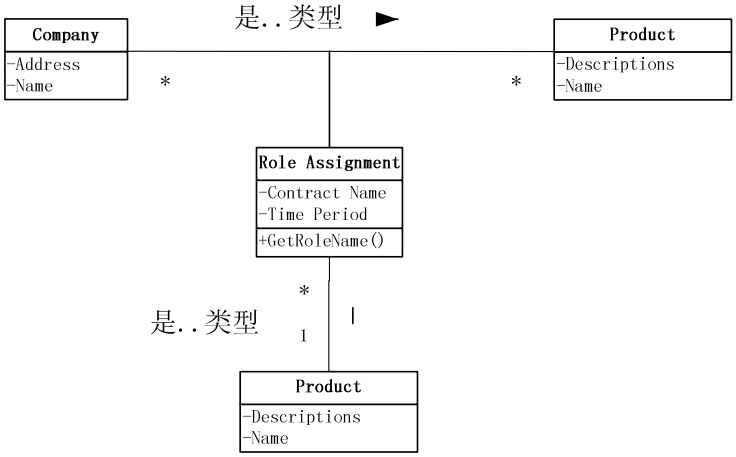


图 12：有角色类型的关联类角色示例

这个解决方法允许所有的角色和角色类型被动态建立和引用。

和类相比，这个模型抽象、有力、有柔性和强烈对象化。

在前面的模型中你会需要每个角色一个类。 在这个新的模型中，类（ 角色类型）包括所有的情况， 但是每个真实的角色类型（举例来说：客户，厂商 ， …)仍然需要创建角色类型对象。

图 13显示元模型为关联类角色的解决方案。我们的第五个角色模式。

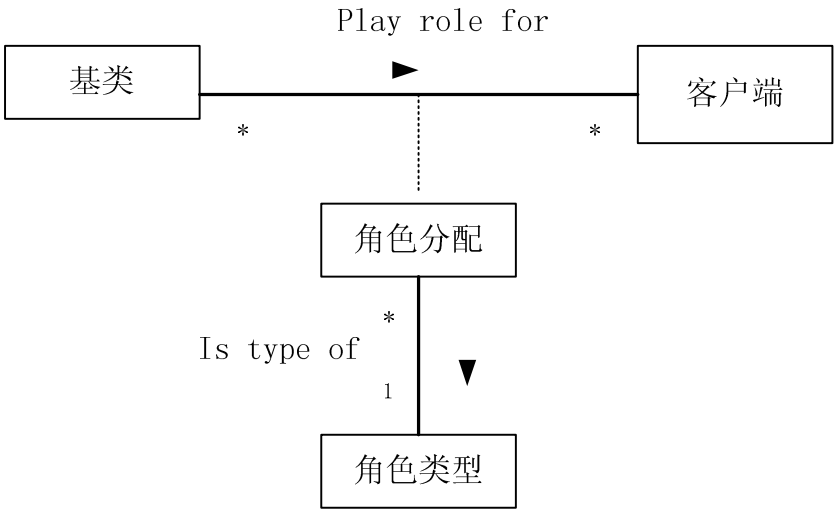


图 13: 有角色类型的关联类角色元模型

6 结论

我们已经看到了一系列的角色需求和他们的对应解决方案，范围从简单的关联角色到角色类和角色关联类。我们已经考虑了每种解决方法的优点和缺点。角色问题经常存在。通过理解本文介绍的解决方法（我们称之为模式或元模型）和它们的特性，我们就能够很快地为任何的角色情况提出最适宜的模型。

征稿

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

UMLChina 公开课

“UML 应用实作细节”

（2003 年 7 月 26-27 日，北京）

现在，UML、RUP、Rose 等概念已经传播得越来越广，书籍、资料也越来越多，决定在开发过程中使用 UML 的开发团队也越来越多。

在开发团队应用 UML 的软件开发过程中，自然会碰到很多细节问题：“我这样识别 actor 和用例对不对？”、“用例文档这样写合适吗？”、“RUP 告诉我该出分析类了，可类怎么得出来啊？”、“先有类，还是先有顺序图啊？”、“类怎样才能和数据库连起来啊？”...许许多多的细节问题，而每一个细节都和背后的原理有关。

本课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，结合大量练习，阐述如何（只）使用 UML 里的三个关键要素：用例、类、顺序图来完成软件开发。使学员自然领会 OOAD/UML 的思想和技术，并对实践中的误区一一指正。整个过程简单实用，简单到甚至可以只有一个文档。

详情请见>>

建立稳定分析模式的模式语言

Haitham Hamza 等 著，wnb 译

查看评论

摘要

一般认为，软件分析模式在减少开销和缩短软件产品生命周期等方面会起到重要的作用。然而，分析模式的巨大潜能还未被充分发掘。缺乏稳定性是当前分析模式存在的主要问题。多数情况下，为特定问题建模产生的分析模式缺乏一般性，即使是为同样的问题建立的分析模型当环境发生变化时难以产生作用，这一情况使得软件开发者不得不从头开始建模工作，难以实现模式的可重用性。为此，本文提出了一种建立稳定分析模式的模式语言，目的在于提出一种在构建分析模式时能够获得稳定性的方法。

1 介绍

分析模式是一种概念模型，用于对问题的核心知识建模。因此，人们希望针对特定问题建立的模式应该能够方便、容易地重用于建模同样的问题，而无论这些问题出现在何种环境中。实际应用中，人们的这一希望还难以实现。事实上，目前对许多领域中众所周知的问题建模所产生的分析模式，当将其应用到不同的环境中时，并不像人们所希望的那样能够被重用。为此，软件开发人员通常喜欢从头开始建立他们的分析模型。

在建立分析模式时考虑稳定性将有助于建立有效的和可重用的模式。这些稳定的模式可以建模问题，无论这些问题出现在何种环境下，也就是说与问题存在的环境无关。文献[1][2]表明了在软件重用和生命周期改进方面软件稳定性具有重要的作用。软件稳定性模型应用了“EBT”（持久性业务主题）和“BO”（业务对象）的概念。本文将稳定性概念应用到分析模式中，提出了稳定分析模式（Stable Analysis Patterns）的概念。稳定分析模式隐含的思想是在进行问题分析时，根据问题的 EBT 及 BO，从增加稳定性和扩展重用性的目标进行考虑。本文所提出的模式语言展示了建立稳定分析模式所需要的主要步骤。

本文的其它部分组织如下：第 2 部分对所提出的模式语言以及不同模式间的交互进行了概述。第 3 部分详细描述了每个模式。第 4 部分对全文进行总结。

2 模式语言概述

本文提出的模式语言包含 8 个模式。这些模式表明了建立稳定分析模式需要的步骤。8 个模式分类为三个主要的层次。每个层次有其主要的目标。图 1 显示了模式语言的三个层次以及每个层次所对应的模式。每个模式包含两个数字。第 1 个数字表明了其所处的层次，第 2 个数字表明了模式号。

第 1 层称为“概念模式”（Concept Patterns）层。该层的模式提供了为理解其它模式语言所涉及的主要的概念。概念模式层包含两个模式：**有效可用的分析模型（Efficient Usable Analysis Models）**[1.1]，该模式描述了建立有效的、可用的分析模型所需要的主要的、基本的属性；**软件稳定性模型（Software Stability Model）**[1.2]，该模式提供了作为获得稳定性的方法所需要的软件稳定性模型的基础概念。

第 2 层是“问题分析模式”（Problem Analysis Patterns）层。该层中的模式表明如何分析需要建模的问题。问题的分析包含四个步骤，每个步骤为一个模式。模式 1-**确定问题**[2.3]；模式 2-**确定持久性问题**[2.4]；模式 3-**确定业务对象**[2.5]；模式 4-**将各个元素 放入适当的抽象层次**[2.6]。

第 3 层是“建立过程模式”（Building-Process Patterns）层。在先前层次对问题进行分析的基础上，需要知道如何**建立稳定的分析模式**[3.7]以及如何**使用稳定分析模式**[3.8]。

语言模式之间的关系参考图 2

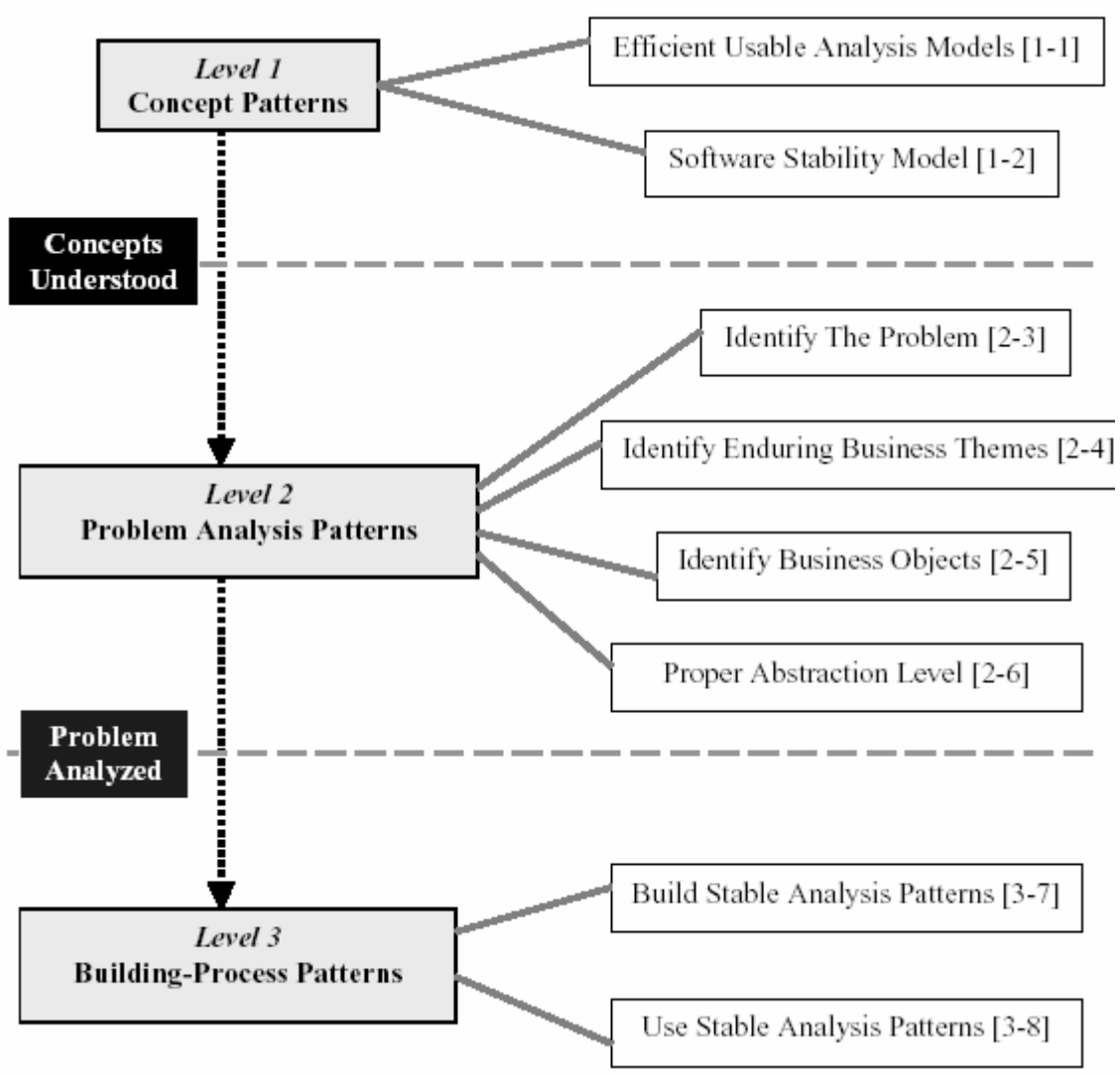

Level 1
Concept Patterns
Efficient Usable Analysis Models [1-1]
Software Stability Model [1-2]
Concepts Understood
Identify The Problem [2-3]
Identify Enduring Business Themes [2-4]
Level 2
Problem Analysis Patterns
Identify Business Objects [2-5]
Proper Abstraction Level [2-6]
Problem Analyzed
Build Stable Analysis Patterns [3-7]
Level 3
Building-Process Patterns
Use Stable Analysis Patterns [3-8]

图 1 模式语言描述

Pattern	Problem
Efficient Usable Analysis Models [1-1]	What are the main essential properties that affect the usability and the effectiveness of analysis models?
Software Stability Model [1-2]	How to accomplish software stability?
Identify The Problem [2-3]	How to focus on a specific problem that the analysis pattern will model?
Identify Enduring Business Themes [2-4]	How to identify the enduring business themes of the problem?
Identify Business Objects [2-5]	How to identify the business objects of the problem?
Proper Abstraction Level [2-6]	How to achieve the proper abstraction level?
Build Stable Analysis Patterns [3-7]	How to assemble the problem model components to build the stable analysis pattern? How to define the relations between the identified EBTs and BOs?
Use Stable Analysis Patterns [3-8]	How to use the contracted stable analysis pattern to model the problem within a specific context?

表 1 模式语言小结

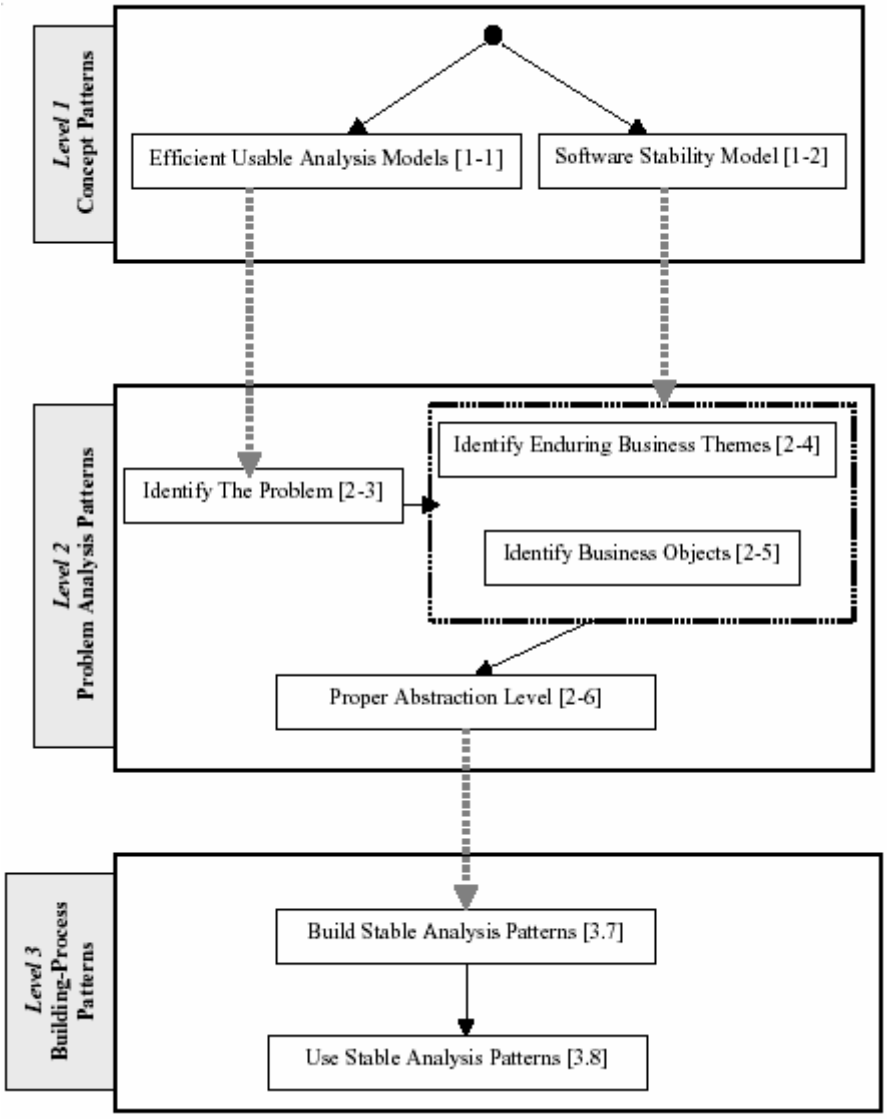


图 2 图示模式语言

3 建立稳定分析模式语言描述

第 1 层 概念模式：首先，理解有效可用分析模型[1.1]。其次，理解软件稳定性模型[1.2]的基本概念。

模式 1.1 “有效可用的分析模型”（Efficient Usable Analysis Models）

目的

表达有效可用分析模型的基本属性。

上下文

分析是解决问题的第一步。分析模型要达到有用和可用的目的需要一些基本的属性。在建立分析模型时应用这些属性将会极大地提高这些模型的质量。

问题

影响分析模型可用性和有效性的基本属性是什么？

约束

- 分析模型作为模型必须满足文献[10]所提到的六个基本的模型属性。即简单、完整并尽可能准确、可测试、稳定、可视表示、易于理解。鉴于分析模型的特性，以上的六个属性还不太充分。
- 在理解某些模型时会产生一些误解。例如，模型的基本属性易于理解不等于简单，许多分析模型易于理解，但实际上相当复杂。类似的混淆将对建立的模型产生严重影响。

解决方案

本文引入了 8 个基本属性覆盖了分析模型的主要品质。满足这八个基本属性并不能保证实现一个有效可用的模型。但是，在实践中，缺乏这些属性的任何一个都可能影响模型的重用性和有效性。

1. 简单：一个模式难以表达整个系统。它用于建模大系统中经常出现的特定问题。系统，从其特征来看，是由多个问题混合而成。因此，对大系统的建模通常形成一个分析模式的集合。实际上，每个分析模式专注于表示一个特定的问题，否则将会产生许多问题。如果不能将一个系统分解为多个组成部分，模式将会非常复杂，模式的一般性将受到影响并且将非常难理解。如果建模系统的主要部分由一个模式实现，将会牺牲结果模式的一般性。应该记住：所有问题同时发生的可能性比每个问题单独发生的可能性要低得多。例如，对payment问题及Buying a car问题进行建模没有什么益处，因为payment问题可能会出现在许多问题中。当在不适当的解决层次上建模系统时，模式的完整性也难以满足。分析师的关注点不是在特定的问题上，系统的重要特点以及子部件极有可能被忽略了。

2. 完整并尽可能准确：与简单性概念有关，该属性保证所有需要的信息都被表达。为了实现完整性，模型不应该忽略任何组件。模型必须能够表示其属性的基本概念。例如，试图建模整个租赁系统的所有属性将使我们忽略系统的一些部分。租赁汽车与保险有关，它与从图书馆借书是不同的情况。结果，想要为整个的租赁系统建立一个模式，除了失去简单性外，还会失去完整性和准确性。

3. 可测试：模型要能够被测试，它必须是特定和清楚的，即模型中所有的类和关系应该是经过验证的且是有效的。

4. 稳定性：稳定性影响模型的可重用性。稳定的模型易于适应和修改，而且不存在退化的风险。

5. 图示化：概念模型难以图示化。但是可视表示方法有助于对模型的正确理解。

6. 易于理解：由于其表达高度抽象，因此概念模型比较复杂。因此，需要对分析模式进行良好的描述，以便更好地理解系统。否则模式的使用既没有吸引力也没有效率。

7. 一般性：一般性是确保模型可重用的基础。缺乏一般性的模型没有什么用处。面对缺乏一般性的模式，分析师们更趋向于建立新的模型而不愿意为适应应用而花费时间和精力去改编一个难以驾驭的模式。一般性意味着针对特定问题进行建模产生的模式应该能够非常容易地用于处于不同环境中的同样问题中。模式一般性可以分为两大类：用于解决在不同环境中经常出现的问题的模式（领域无关模式）；用于解决在特定环境中经常出现的问题的模式（特定领域模式）。对后者而言，即使该类模式仅仅使用在一定的环境中，仍然可以认为它具有一般性，但在该情况下，应当确保该模式不会出现在其它环境中。

8. 容易使用和重用：分析模式应该以清晰的方式表达以利于重用。认识到模式将应用于更大的模型这一点是非常重要的。非常容易地使用和设计能够重用的模式常常意味着该模式能够更加容易地被重用。

下一个模式

在分析和研究了影响分析模型的可用性和有效性的基本属性后，需要理解**“软件稳定性模型”**的基本概念。

模式 1.2 “软件稳定性模型”

目的

描述软件稳定性模型的结构及其基本概念（持久的业务主题“EBT，业务对象BO，行业对象IO”）。由此表明这些因素之间的关系以及如何将它们协调起来建立稳定的模型。

上下文

稳定性对任何工程系统而言都是极为重要的特性。在软件过程领域，拥有稳定的分析模型、稳定的设计模型、稳定的软件架构、稳定的模式将减少开销并提高软件工程产品的质量。

问题

如何获得软件稳定性？

约束

- 对问题进行建模时很难将分析、设计、实现问题完全区分开。通常，分析师在头脑中分析问题（问题领域）及设计问题（求解空间）。由于对同样的问题会产生多个不同的解决方案，因此混淆问题建模和解决方案建模将影响该模型的可重用性。这样做的后果是，当希望对同样的问题采用其它的解决方案时，需要从头开始对问题进行建模。所以许多分析模型缺乏稳定性。
- 另外，在建立分析模型时需要获该问题的核心知识。

解决方案

在文献[1][2]中介绍的软件稳定性概念表明了其在软件重用和软件生命周期开发中的重要作用。图3给出了SSM（软件稳定性模型）的结构。在SSM中系统模型有三个层次：持久性业务主题（EBT）层、业务对象（BO）层、行业对象 (IO) 层。

系统中每个类按照其属性被分类到三个层次中。例如，表示系统持久性和基本概念的类分类为EBT。因为EBT表示系统的持久性概念，它们是非常稳定的，并成为SSM的核心。

确定且稳定可以进行内部的修改的类，分类为BO。例如，人类是BO。该类从外部来看是稳定的，他们可以在内部发生改变（人类可以结婚、或者生病）。EBT和BO构成了SSM的核心。

IO层包含不稳定的类，因此它们可以被修改、增加、甚至移出系统而不会影响系统的核心。

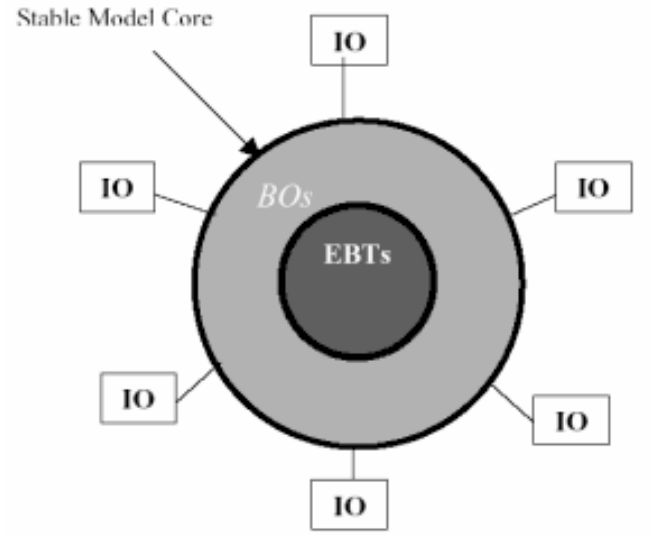


图3 软件稳定性模型结构

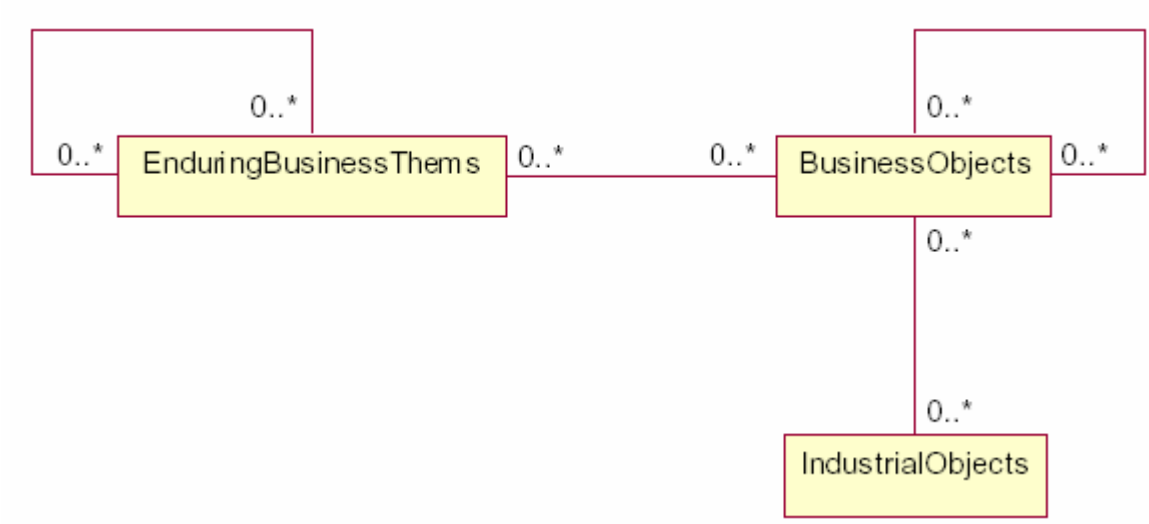


图4 SSM要素之间的关系

示例

示例展示了如何在简单问题中应用软件稳定性概念。

问题：建模小型船运公司的业务。该公司提供几种不同的船运服务。

解决方案：使用软件稳定性概念建模该问题。首先，需要识别该问题的EBT、BO、IO以建立其稳定的模型。在该问题中，EBT可能包含：

- “transportation ”表示了该业务的主要目的；
- “Scheduling”表示如何管理船运日期、目的地等。

以上所提出的EBT比较抽象，它们描述了系统所以存在的原因。为了使这些抽象的、模糊的对象变得切实，需要将它们映射到更加切实的对象。识别问题的BO将完成该映射。在前例的问题中可能的BO是“调度”。该BO将EBT“规划”映射成更加切实的对象“调度”。该BO是外部稳定的，当然，其内部仍然可以发生变化。例如，调度可能天天发生变化，然而，这样的变化不会影响作为问题模型中一个对象“调度”的存在。

进一步需要能够物理地映射BO成为切实的更加具体的对象。这些对象是系统的IO。在该问题中，可以考虑BO“调度”的不同实现的情况。例如，可以用一页纸表示调度，也可以使用一些复杂的软件程序来表示，或者两种方法都采用。通过采用稳定性的概念来建模该问题，类似的变化将不会影响模型的核心。

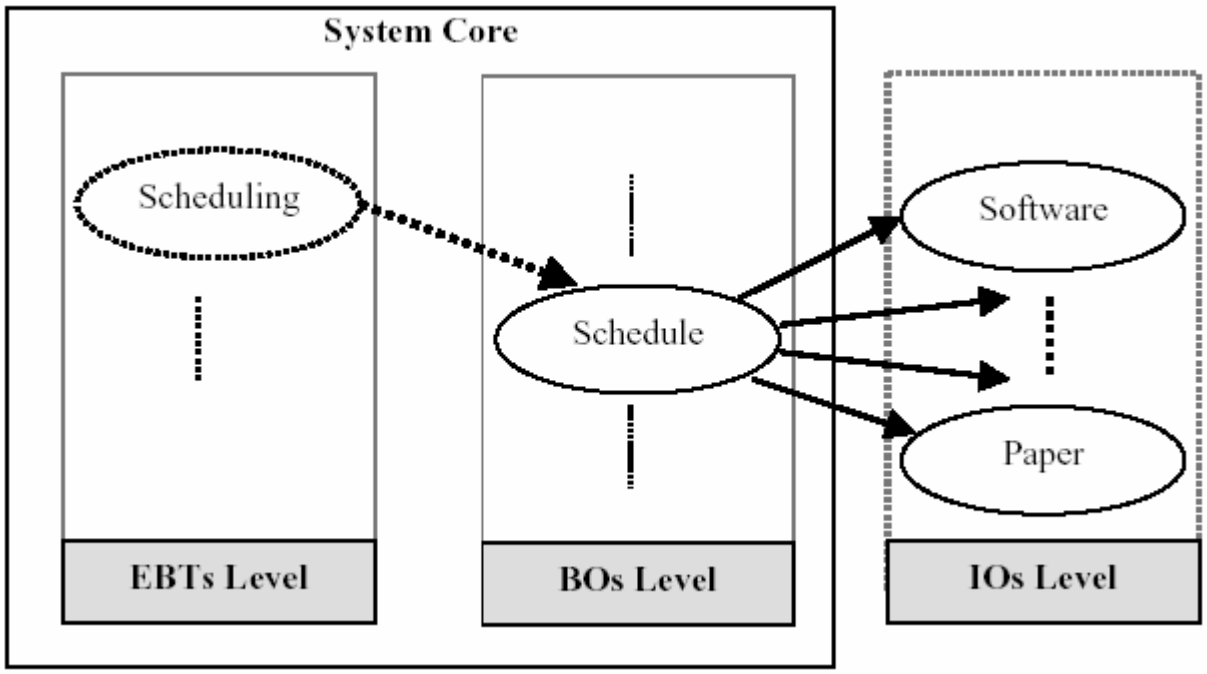


图5 在EBT、BO、IO之间实现映射例

图5显示如何将EBT“规划”映射为更加切实的对象（BO）“调度”并最终成为具体的对象（IO）“纸”和“软件”

下一个模式

在对概念进行了描述和理解后，下面将对问题进行分析。对问题进行分析，需要进行包括“识别问题”、“识别持久性业务主题”、“识别业务对象”并完成“适当的抽象层次”等方面的工作。

第 2 层 问题分析模式：首先，确定问题[3]。然后确定持久性业务主题[4]。确定业务对象[5]。最后，将 EBT 及 OBs 放入适当的抽象层次。

模式 2.3 确定问题

目的

如何将关注点集中于需要分析的问题。

上下文

分析模式的可重用性与其建模问题的数量有直接的关系。如果某个模式可用于建模多个问题，则其一般性将比较差，因为所有问题同时发生的几率比每个问题独立发生的几率要低得多。专注于某个特定的问题是实现模式可重用的关键因素之一。

问题

如何专注于分析模式将建模的特定问题。

约束

- 许多问题一起出现在多个环境中，将这些问题作为一个问题来建模。
- 有时将一个较大的问题分解为多个小问题非常困难。
- 实际上，并不能简单地认为将一些小问题组合起来就能适合于一个大型的问题。

解决方案

在对一个问题建模前，首先需要检查该问题是否能够分解成更小的实际的问题。对下列问题的回答有助于完成这一工作。这些问题包括：“需要解决的问题是什么？”“是否可以将该问题分解为许多个小问题？”“在已知的情况中，在哪些情况下有这些小问题实际存在？”

如果能够找到小问题存在的实际的环境，则需要独立地对这些小问题建模。如果小问题没有实际的应用，将其放在一起建模。

示例

以下我们将通过考虑“帐户问题”，详细描述以上问题分解的思想。事实上，不久前，帐户问题仅仅用于指明银行和金融帐户。目前，如果不与一定的词汇合成使用，术语“帐户”就成为一个比较模糊的概念。例如，除了所有传统已知的业务和银行帐号外，目前还存在电子邮件账户，在线商场账户，在线学习帐号，订阅帐号以及许多其他的帐号。

Flower[7] 曾提出了一个解决帐号问题的模型（Account Pattern）。图6显示了Account Pattern的类图。该模式同时建立两个不同的问题。第1个问题是“帐号问题”，第2个问题是“簿记问题”，这是两个独立的问题。即使它们在许多环境中同时出现，但有时只有簿记而没有帐号，或者相反。图7显示了有帐号没有簿记的情况，图8显示了有簿记没有帐号的情况。结果模式的一般性比较有限。这些特征对模式的可重用性起到相反的作用。

帐号问题的稳定模型在本文后续部分将一步一步完成。从此观点出发，将显示在哪些模式有助于建立稳定的帐号模型。

图6 文献 [7]提供的Account Pattern

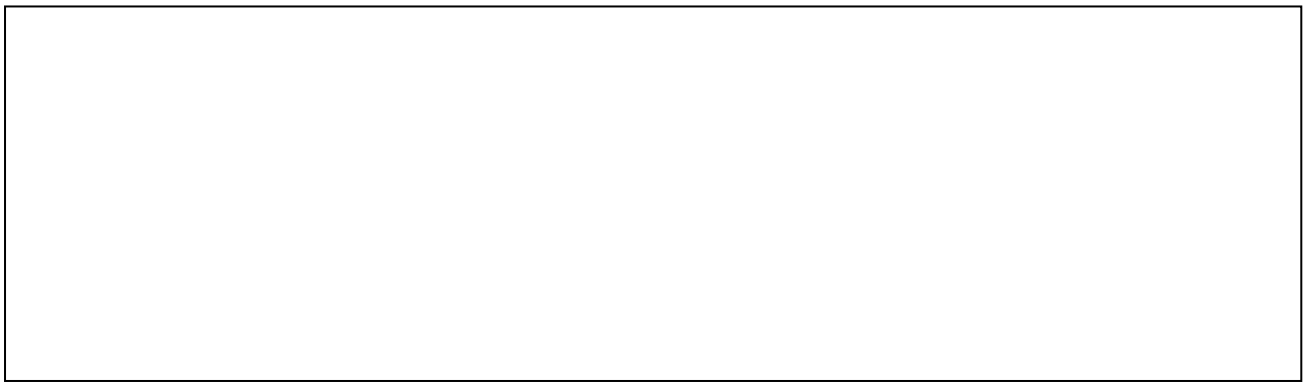


图7 有帐号无簿记例子

下表包含关于Nebraska-Lincoln学院2002年春季课程类调度的信息。在该表中，每块信息构成了一个表的簿记。不需要帐号信息来维持该信息。

Call #	Course Title	Course #	Cr Hrs	Sec.	Time	Day	Room
2850/2867	Computer Architecture	430/830	003	001	0230-0320p	M W F	Freg 112
2855/2873	Software Engineering	461/861	003	001	0930-1045p	T R	Freg 111

图 8 有簿记无帐号的例子

下一个模式

在确定了模型的特殊问题后，需要确定该问题的持久性业务主题。

模式 2.4 确定持久性业务主题

目标

确定问题的持久性业务主题。

上下文

在问题建模过程中使用软件稳定性概念时，首先需要确定问题的核心元素。即：问题的持久性概念。

问题

如何确定问题的持久性业务主题。

约束

- EBT 获取问题的核心知识，然而一些 EBT 获取的是在特定环境中问题的核心知识。这样的 EBT 应该从模型中去除掉。
- 遗憾地是，领域经验并不能总是帮助我们精确地获得有关的 EBT。
- 即使选择的 EBT 初看起来确实与问题有关，实际上其多数却与建模的问题无关。
- 问题的 EBT 应该尽可能小。获取与问题确实有关系的 EBT 通常比较困难。

- 一些 EBT 可能缺乏 EBT 的基本属性。在此情况下，应该将其作为 BO 或 IO 重新确认。

解决方案

一种获得问题的适当 EBT 的方法可以分为以下三个步骤：

步骤 1 建立初始 EBT 列表。从回答下列问题入手建立初始的 EBT 列表。“该问题用于处理何种情况？”换句话说，“该问题存在的原因是什么？”该步骤的输出是问题的初始 EBT 表。这些 EBT 仍然是暂时的，一些 EBT 与问题的关系并不象看起来那样强。

步骤 2 过滤 EBT 列表。该步骤目的是从初始列表中消除冗余和不相关的 EBT。因为在建立初始 EBT 时，建立者通常带有某些偏见，因此该步骤非常重要。该步骤的输出是经过修改的 EBT 列表，通常比初始列表要小。

步骤 3 检查主要的 EBT 属性。该步骤将获得的 EBT 与 EBT 主要的基本属性进行对照。一般过程是回答下列问题：（包含希望的回答）

- 可以用其它 EBT 替换该 EBT 吗？不。
- 该 EBT 是内部稳定的和外部稳定的吗？换句话说，该 EBT 反映了正在建模的问题的核心知识吗？是
- 该 EBT 属于特定的领域吗？不
- 可以直接从物理上表示该 EBT 吗？不能。应该注意 EBT 不应该有直接的物理表示（IO）若是则应该考虑为 BO（参考图 3 所示的软件稳定性模型）。例如对“协议“我们易于将其理解为 EBT。然而，协议有直接的物理表示（例如，合同）。因此，协议不是 EBT，而是 BO

不能满足这些属性的所有 EBT 应该从列表中删除，图 9 对确定问题 EBT 的三个步骤进行了总结。

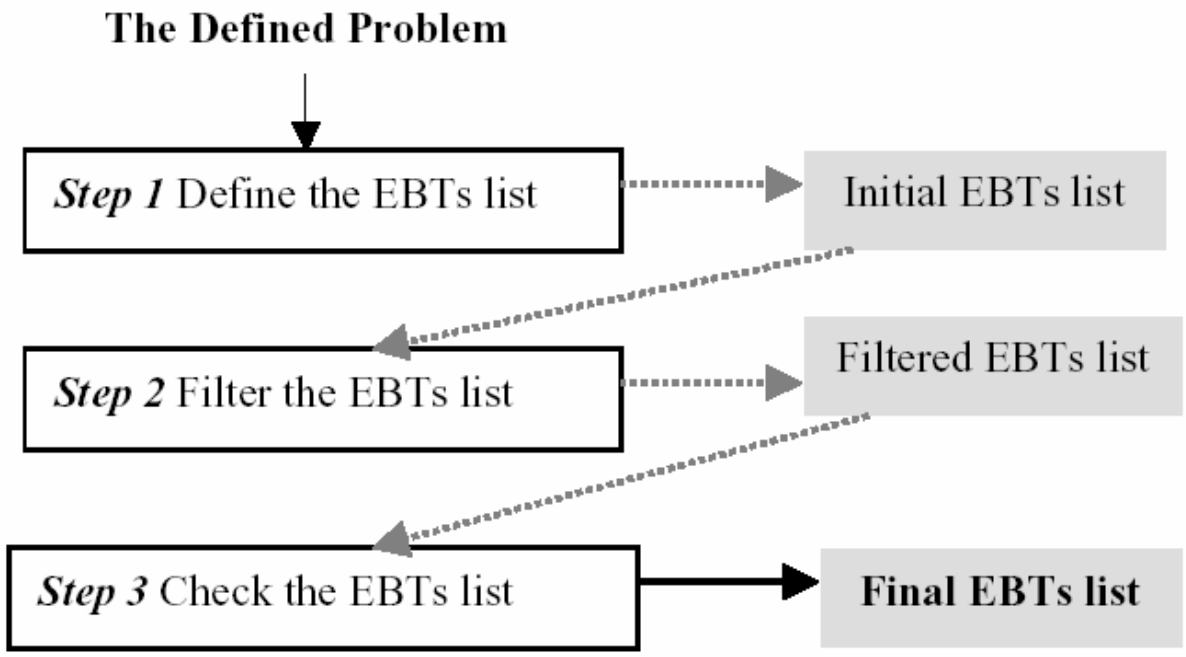


图 9 确定问题 EBT 的步骤

示例

该示例演示了关于帐号问题的区分步骤：

步骤 1 建立初始 EBT 列表

需要回答问题“该问题用于处理何种情况？”初始的 EBT 包含如下：

存储、拥有、可跟踪、记录。

步骤 2 过滤 EBT 列表

获取帐号问题的最适当 EBT。为此，需要检验列表中每个 EBT 并检查它是否反映了建模问题的持久性概念。

因为只将关注点放在帐号问题上，可以看出多数定义的与帐号有关的 EBT 有簿记。例如，存储、跟踪能力和记录都依赖于帐号内存在的簿记。如果帐号没有簿记，则存储、跟踪能力和记录都没有存在的必要。因此，需要从 EBT 列表中去掉存储、跟踪能力和记录。现在，只存在一个 EBT，拥有者。

步骤 3 检查主要的 EBT 属性

可以用其它 EBT 替换拥有者吗？不能

拥有者是内部和外部稳定的吗？是

拥有者属于特定应用或领域吗？不是

可以直接物理表示拥有者吗？不能

讨论

1 获取问题的 EBT 可以采用多种方法。然而，经验表明提出的方法是有效的。

2 对两个重要因素的争论和讨论更明确有效地获取 EBT。

3 对小问题来说，应该尽量减少 EBT 的数量，例如 1 至 2 个。这通常使得模式更加专注和有效。

4 目前获得的的 EBT 列表所包含的 EBT 不一定是 模式结构中最后的 EBT。在后续的步骤中仍然还需要对 EBT 进行分析和考虑。

下一个模式

在完成了对问题 EBT 的确定工作后，需要“确定业务对象”。

模式 2.5 确定业务对象

目的

展示如何确定问题的业务对象。

上下文

在建模问题时使用软件稳定性概念确定 EBT 后，下一步是确定业务对象。

问题

如何区分问题的业务对象。

约束

- 在某些情况下，一个对象到底是 EBT 或是 BO 很难确定。例如，协议由于是表示一个概念因此可以视为是 EBT，但实际上是 BO。
- 在对问题的 EBT 进行确定后，概念化关联更加含糊，因为问题的 BO 必然是基于 EBT。这将导致 BO 的获取更加困难。
- 通常在问题的 EBT 与问题的 BO 之间并没有一一对应的关系。对 EBT 来说可能存在到 BO 的直接映射，而 BO 却没有到 EBT 的直接映射。另外，一个 EBT 可能被映射成几个 BO。
- 除了可以识别问题的主要 BO，还可能存在一些隐含的 BO。这些隐含的 BO 与定义的 EBT 没有直接的关系，它们可能与主 BO 有关与问题中其它隐含的 BO 有关。

解决方案

一种识别 BO 的方法，可以采用下面的四个步骤：

步骤 1 确定问题的主 BO。通过该步骤确定主要的直接与每个 EBT 有关的 BO 的集合。可能有一个或多个 BO 与问题中每个 EBT 对应。然而，一些 EBT 可能没有对应的 BO。问题 BO 的主集合可以通过回答下列问题确定（注意，一些问题不能应用到某些 EBT 中。主要是与每个 EBT 的属性有关）。

- 如何达到 EBT 表达的目标？（例如：为完成 EBT“调度或组织”的目标，可以使用 BO“调度”。另外的例子，对 EBT“谈判”，需要 BO“谈判环境”和“完成谈判的谈判媒介”）
- 使用该 EBT 会产生什么结果？（例如：对 EBT“谈判”，最后的结果是“达成共识确认某 BO 是对 EBT 的映射”）
- 谁将使用该 EBT？（例如：BO“Party”使用“谈判”。该”Party”可以是一个人，一个公司，或一个组织。因此，“Party”是可能映射 EBT“谈判”的 BO）

步骤 2 过滤主 BO 列表。该步骤进一步精化以前步骤确定的 BO。该步骤的目标是从初始的列表上减少、消除冗余或无关的 BO。实现该目标的方法是与小组就所列出的 BO 进行讨论。

步骤 3 确定问题隐含的 BO。该步骤确定问题中隐含的 BO。所谓隐含的含义是，这些 BO 与问题的 EBT 没有直接的关系。由此，不能在前述的两个步骤中获取隐含的 BO。

例如，假定建模一个能够提供不同类型材料（如天然气、水等）的运输服务的系统。其中可能会建立一个 EBT ”运输服务”。而映射该 EBT 的 BO 是“运输”，IO 是“卡车”。在此情况下，“材料”可能是一个隐含的 BO。没有 BO“材料”直接映射的 EBT，然而，在两个 BO“运输”和“材料”之间存在明确的关系。

在考虑问题中隐含的 BO 时，设想一个临时场景包含每个 EBT 以及相应的 BO。然后回答问题“在该问题中还缺什么？”通常对该问题的回答包含了问题隐含的 BO。一些问题没有任何隐含的 BO，特别是在小规模问题的情况下。

步骤 4 检查 BO 的特征：该步骤用于确保识别的 BO 满足主要 BO 的特征。这些特征可总结如下：

- 业务对象部分确实（？）；
- 业务对象是外部稳定的，它们应该在整个问题的生命周期中一直保持稳定；
- 业务对象是可修改的，可以进行内部修改；
- 业务对象有直接的物理表示（IO）。

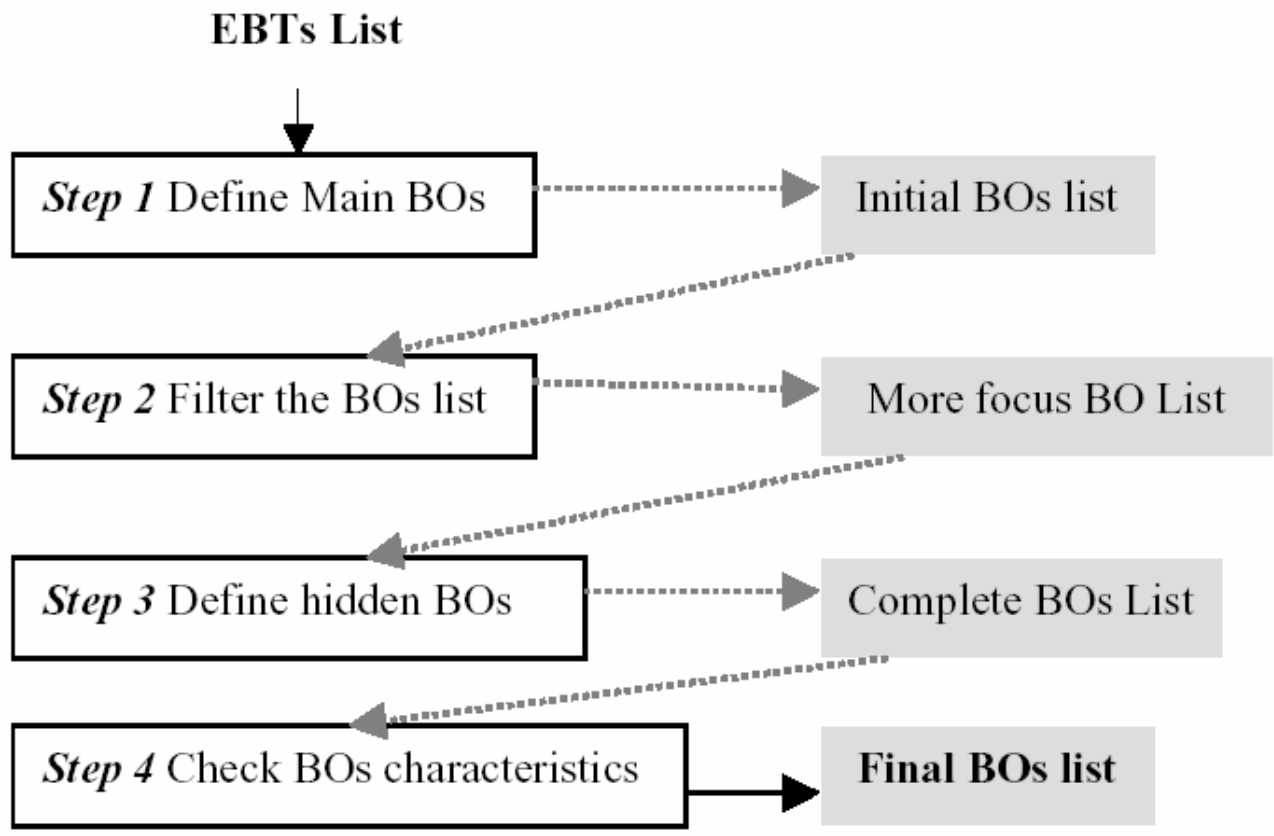


图 10 确定问题 BO 的步骤

示例

在该示例中，需要确定“帐号”问题的 BO。为了确定该问题的 BO，将应用以上提出的四步方法。

第 1 步的输入，EBT 列表包含一个 EBT，“拥有者”

步骤 1 确定问题的主要 BO

- 如何能够达到“拥有者”的目标？在帐号系统中，为了实现拥有需要获得、拥有一些事情。“帐户”本身带有“拥有”的意思。
- 使用“拥有”的结果是什么？分析拥有的意义，可以获得“私有”的概念，但其不是一个 BO，是一个多余的 EBT，因此，排除该概念。协议也可能是系统的 BO，当拥有一个帐户，你一定要同意其制定的规则。
- 谁使用“拥有”？对能够使用帐户的拥有者来说，它应该遵守为此制定的规则。由此，主要 BO 的列表包含：拥有者、帐户和协议。

步骤 2 过滤主 BO 列表

对主列表中的 BO：拥有者、帐户、协议。当对该问题进行建模时，曾讨论过对 BO“协议”的准确使用问题。协议在许多环境中是一个通用的术语。例如，在谈判问题中，需要 BO“协议”。因此，将协议作为帐户模式的 BO 将违背模式属性简单性的原则。协议是一个可发生于多种环境中的单独的问题。因此，将协议从 BO 主列表中消除。过滤后的 BO 主列表包括：帐户、拥有者。

步骤 3 确定问题中隐含的 BO

在确定和过滤主 BO 列表后，回答问题“为完成帐户问题还需要其它工作吗？”。有一个帐户、其拥有者。对建模任何基本的帐户问题来说它们正是所需要的。

步骤 4 检查 BO 的特征

- 业务对象应该是部分确实的，帐户和拥有者满足该特性。
- 业务对象是在其整个生命周期中是外部稳定的。帐户和拥有者总是稳定的，换句话说，在模型中不可能存在一个没有这两个对象的帐户。

- 业务对象是可修改的，可以在内部对其进行修改。帐户和拥有者可能进行内部修改，例如，可以对银行帐户的某些属性进行修改（例如增加透支保护服务）。但这仅仅是帐户内部的改变。对外部而言没有任何改变。另外，对于业务对象拥有者来说，他可能生病了，但仍然是帐户的拥有者。
- 业务对象可能直接表达 IO。对业务对象帐户来说，依据不同的环境可能有不同的表示。帐户可能是复印机上的一个代码。而业务对象拥有者是一个物理存在的人，他拥有该帐户并使用它。

讨论

1 获取问题的 EBT 有几种不同的方法，实践表明上述提出的获取 EBT 的方法是有效的。

2 对小问题应该尽量减少其业务对象的数量。通常，对小问题有 2-4 个集中关注的业务对象。

3 至于在该模式中最后定义出的EBT，业务对象列表在下阶段中将被进一步增强和修改。

下一个模式

在确定了问题的 EBT 及 BO 后，需要完成“适当的抽象层次”这一工作

模式 2.6 “适当的抽象层次”

目的

帮助完成分析模式的适当的抽象层次

上下文

如果模式缺乏适当的抽象层次，该模式的可重用能力将难以保证。建立适当抽象层次的最终目标是使模式尽可能用于所有可能的环境中。确定问题的 EBT 和 BO 是实现适当抽象层次的前提条件。在确定 EBT 和 BO 的过程中，已经将与特定领域有关的 EBT 和 BO 都排除在外。但该抽象层次对需求来说还不充分。因此，为增强模型的抽象层次，还需要做更多的工作。

问题

如何达到适当的抽象层次？

约束

- 通常从前述的阶段中获得的 EBT 和 BO 的名字还不够精确；
- 即使能够找到问题中每个 EBT 和 BO 的适当的名字，通常为每个类定义其满足所有环境的属性和操作非常困难。

解决方案

在确定了问题的 EBT 和 BO 后，确保这些元素有适当的抽象层次。实现该目标的方法总结如下：

- 不要为 EBT 和 BO 分配任何特定的属性或操作，即使若没有这些特定的属性和操作可能会导致在使用模式时更加困难。分配特定的属性和操作将导致混淆不利于对模型的理解。
- 进一步检验已经获得的每个 EBT 和 BO 的名称和结构。在该步骤中，BO 的名称极有可能需要改变。进行检验的一个指导原则是将该问题的 EBT 和 BO 与出现在与建模环境中不同的情况进行比较。同时考虑异常情况并检验是否已经确定的 EBT 和 BO 有助于抽象模式并使得模式具有一般性。以下将用示例进行描述；
- 如果在某些情况下确定的 EBT 和 BO 在改变名称或修改结构时不能被操作，试图解决的问题要么太大要么太小。因此，需要对问题进行重新定义。

示例

仍然以帐户系统为例。到此为止，已经确定包含如下的 EBT 和 BO：

EBT：“拥有”

BO：“拥有者”、“任一帐户”。

粗略看来，BO“拥有者”的名称似乎非常适合。然而，当仔细考虑在帐户系统中可能出现的所有环境情况后，却产生一个问题，当许多用户共享同一个帐户时会出现什么情况呢？例如，考察信用卡的情况，个体成为帐户的拥有者，但该个体可以有限制条件地允许其它用户使用该信用卡。在此情况下，使用“拥有者”作为 BO 限制了该模式，在此模式下，特定的帐户只有一个使用者可以使用，该使用者是帐户的拥有者。

通过以上的分析，显然 BO“拥有者”缺乏适当的抽象层次来协助处理不同情况的问题。因此，需要以某种方式重新定义 BO“拥有者”，使得该 BO 更加一般化。

对该问题的解决办法是，将 BO“拥有者”的名称改为“持有者”，后者具有更一般的意义。并据此改变继承结构以获取不同帐户持有者的不同角色。

图 11 显示了用于考虑该问题的详细步骤。

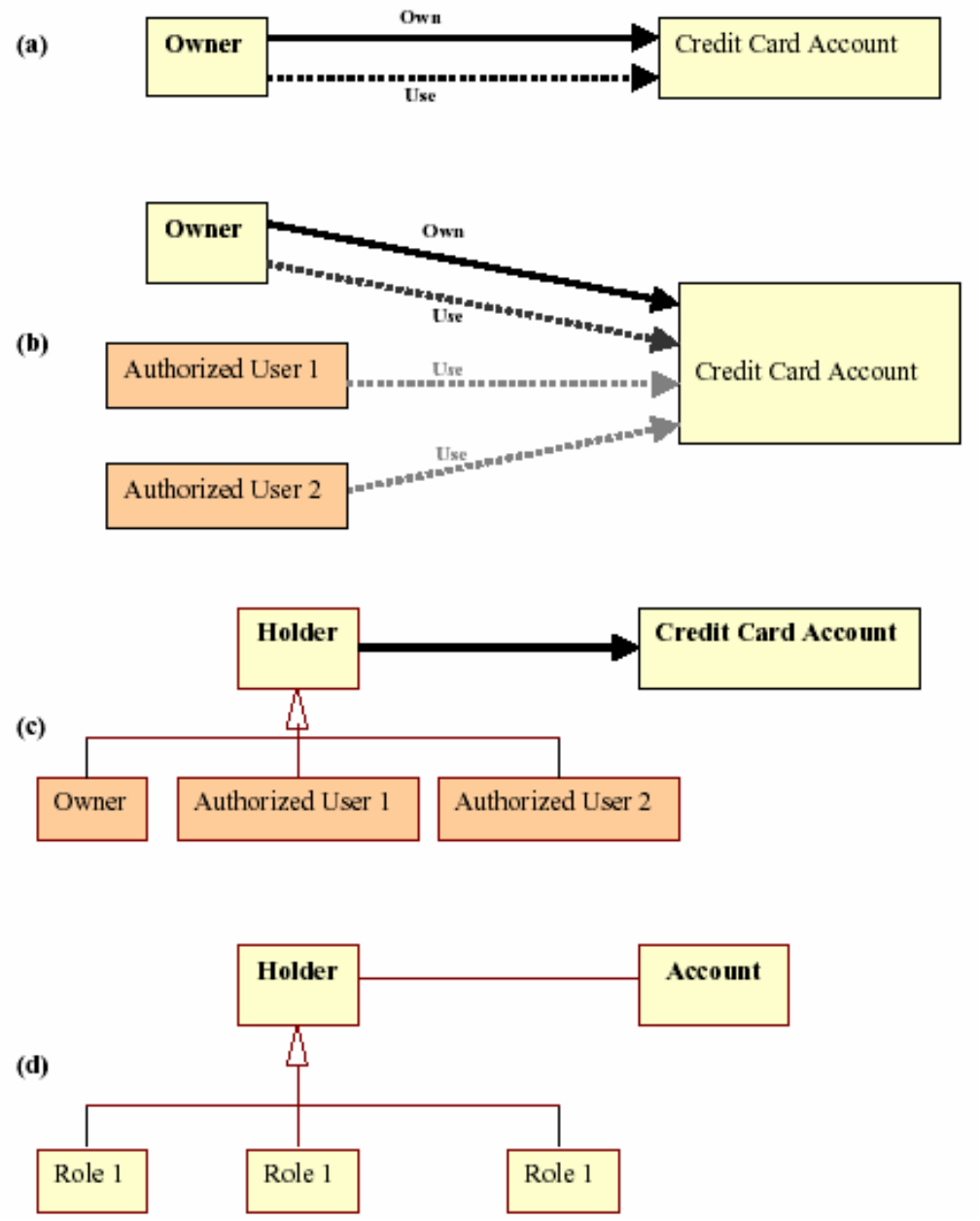


图 11. (a) 显示了“拥有者”与其“帐户”；(b) 显示了目前的表示难以处理的一种情况；(c) 显示了处理该问题的一种方法；(d) 给出了处理该问题的最终的抽象层次。

讨论

抽象层次确定了模式能够使用的范围。但是，过分抽象会产生负面作用。一方面，在对模型的灵活性与可重用性的取舍方面需要进行仔细的权衡；另一方面，对复杂性与模型重用的理解亦需仔细考量。在此平衡中，抽象层次将发挥重要的作用。抽象程度低会降低模型的可重用性，采用这样的模型，工作不会轻松。而将其与在更高抽象级别上设计的同一个模型进行比较，则相对简单。相反，高度抽象的模型会使模型非常复杂。由此，最好的方法是一个概念模型，该模型足够简单，所以非常容易使用和理解，由于抽象程度高，因此比较一般，重用性更强。目前对处理抽象层次问题也存在几中不同的方法。一些观点描述如下：

- 在获取分析模式的时，一种方法是人们根据他们参加某些项目获得的经验中抽取出来的，他们认为为将分析模式重用于其它的环境中，对获得的分析模式进行深入的抽象是不安全的。在图 6 中所显示的帐户模式是该方法的一个示例。因为没有人能够成为所有领域的专家，所以领域专家型分析师通常获取特定领域模式，即使这些建模的问题可以在多个环境中存在。例如，按照此种方法可以为不同环境中的帐户问题建立多个模式。例如为银行帐户建立的模式、为 WEB 应用建立的模式等等。如果能够建立一个通用的模式-AnyAccount，该模式获取不同类型帐户的核心结构，则该模式将更加有效。无论在何种环境中都可以应用该模式。

- 类推在抽象问题的另外一种观点。使用该方法，在一种环境中建立的模式可以通过在模式和新的应用间进行类推而获得。通过类推，将模式中类的名称按照新应用的要求进行类推并修改。有时为了使模式适应新的应用可以增加或删除模式中的类。对完成抽象层次的任务该方法结果将建立模板而不是模式。图 12 显示了一个应用成组模式的示例[11]。显示的模式目标是提供一种模型，该模型可以作为一种可重用的模型应用到所有与租用资源有关的问题中去。图 13 显示了一个使用该模式应用于图书馆应用服务的示例[11]。

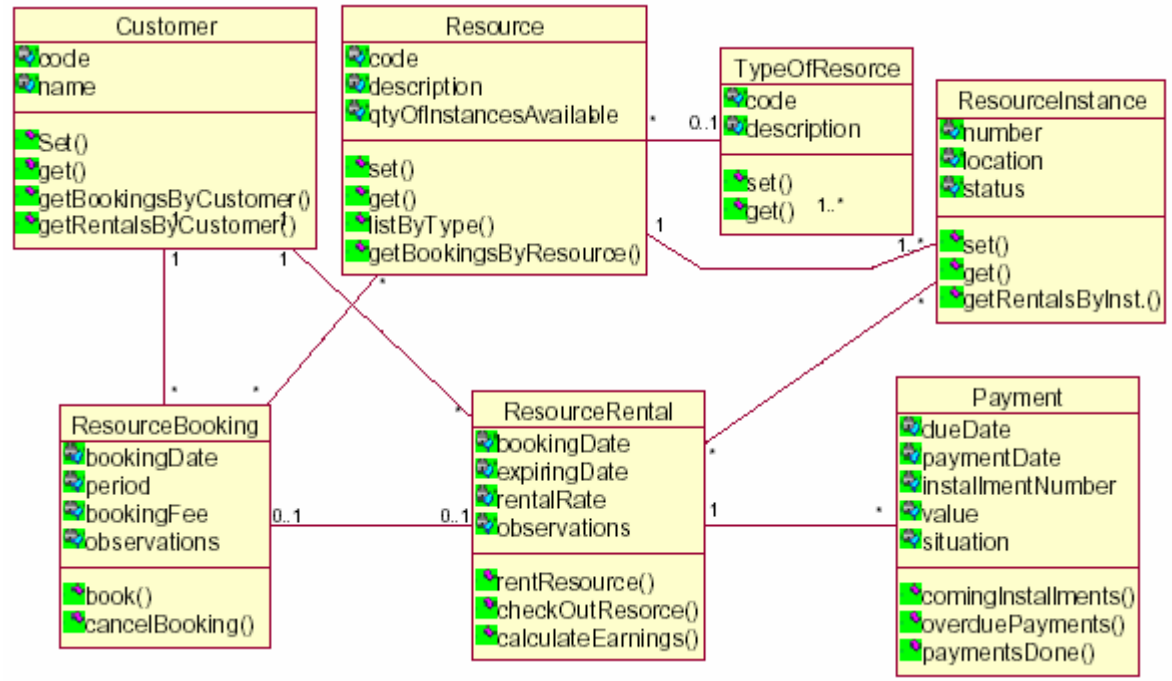


图 12 资源租赁模式

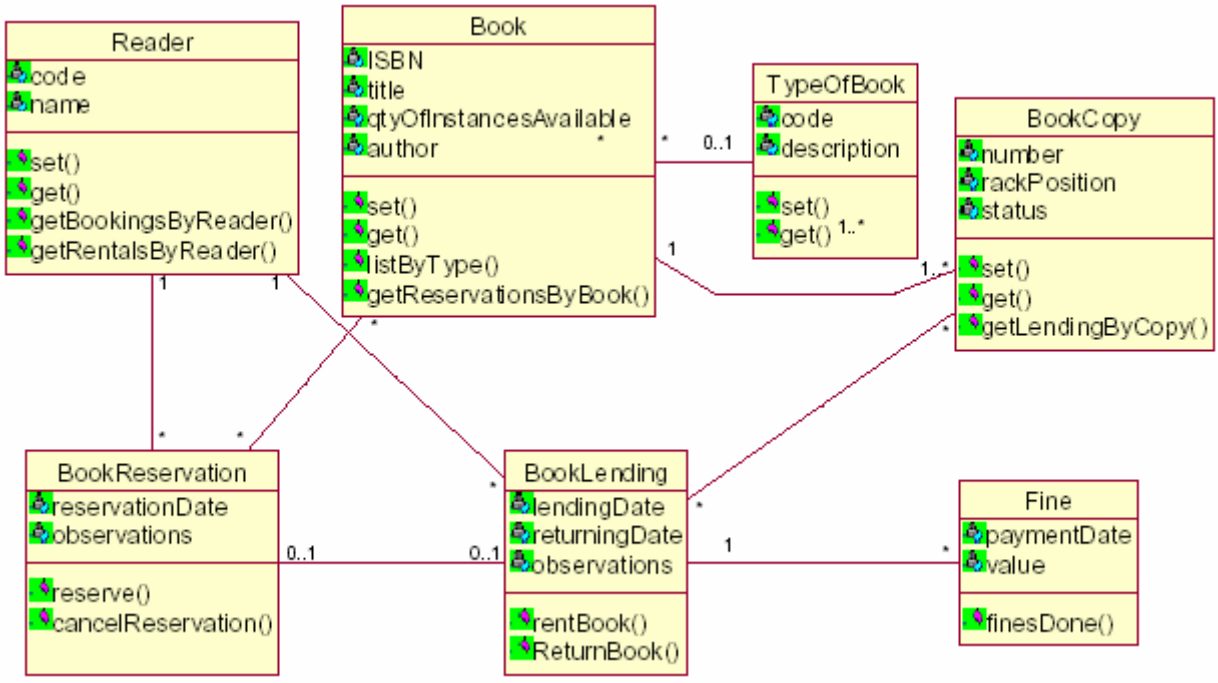


图 13 资源租赁模式应用于图书馆服务的示例

下一个模式

在确定了问题、问题的持久性业务主题、问题的业务对象和适当的抽象层次后，下一步是建立稳定的分析模式以及使用稳定的分析模式.

层次 3 建立过程模式：在该层次中，首先建立稳定分析模式，然后使用稳定分析模式[8]。

模式 3.7 建立稳定分析模式

目的

在定义了问题、问题的 EBT、BO 以及适当的抽象层次后，需要将所有建立的模式组合在一起，建立建模该问题的稳定分析模式。

上下文

以上主要关注于特定问题并且定义了该问题所有的稳定性模型涉及的要素。以下将理解如何确定和区分这些要素之间的关系。

问题

如何将问题模型的各种组件组合起来以建立稳定的分析模式？如何定义确定的 EBT 和 BO 之间的关系？

约束

- 模式元素的抽象层次对于定义元素间的关系产生了困难；
- 有一些 EBT 没有对应的 BO；
- 需要对问题主要的和隐含的 BO 之间的关系进行定义和区分。

解决方案

为确定问题的 EBT 和 BO 之间的关系，其中一种可采用的方法是将这些模式放入一种环境中，以使得我们可以图形化这些关系。为了确保模式的一般性，也需要将这些模式放入不同的环境中。通过这样的方法可以帮助我们增强定义关系的准确性以及元素之间关系的多样性。一种确定问题的不同 EBT 和 BO 之间关系的方法描述如下：

首先将 BO 表中的 BO 加入到 EBT 表中对应的 EBT 中。第二步定义问题中 EBT 之间的关系。最后定义问题的主要 BO 之间的关系，以及主要 BO 与隐含 BO 之间的关系。图 14 显示了问题的 EBT 与 BO 之间的关系。

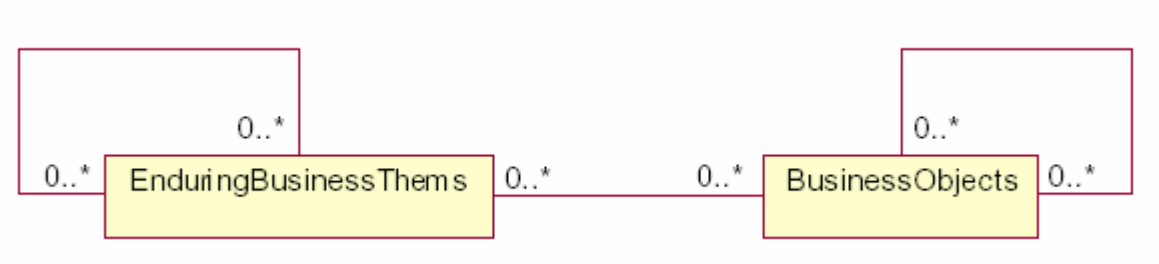


图 14 EBT 与 BO 之间的关系

示例

示例 1 建立“AnyAccount”模式

该示例表现的是如何将帐户问题中的 EBT 和 BO 连接起来以建立稳定的分析模式 AnyAccount。以下首先将至此为止已经存在的关于帐户问题的信息。

Element	Description
Problem Definition	Modeling any account for any context
EBTs List	“Ownership”.
Main BOs List	“Holder”, and “AnyAccount”.
Hidden BOs List	None

定义EBT和对应的主要BO之间的关系：

EBT“拥有”与BO“拥有者”之间的关系是什么？

EBT“拥有”与BO“AnyAccount”之间的关系是什么？

“拥有”表示了拥有帐户的原则，可以管理零个到多个帐户。例如，在银行帐户系统中，预定义的原则是将帐户分为支票帐户和存取帐户。因为该原则的各个部分规定了帐户拥有者的责任和利益，因此说EBT“拥有”与BO“拥有者”具有一定的关联。

定义问题中 EBT 之间的关系：

由于在该问题中没有其它 EBT，因此直接到下一步。

定义问题中 BO 之间的关系：

在 BO“拥有者”与 BO“ AnyAccount”之间的关系是什么？该关系是比较清楚的，“拥有者”使用“ AnyAccount”。因此在这两个 BO 之间必然存在一定的关联。“拥有者”可以有有多个“ AnyAccount”，“ AnyAccount”也可以属于一个或多个“拥有者”。在前面的讨论中，已经知道对“ AnyAccount”的拥有者可以是其“Owner”，也可能是其“Owner”和其它权限的拥有者，例如在信用卡的例子中。图 15 给出了“AnyAccount”模式的类图。

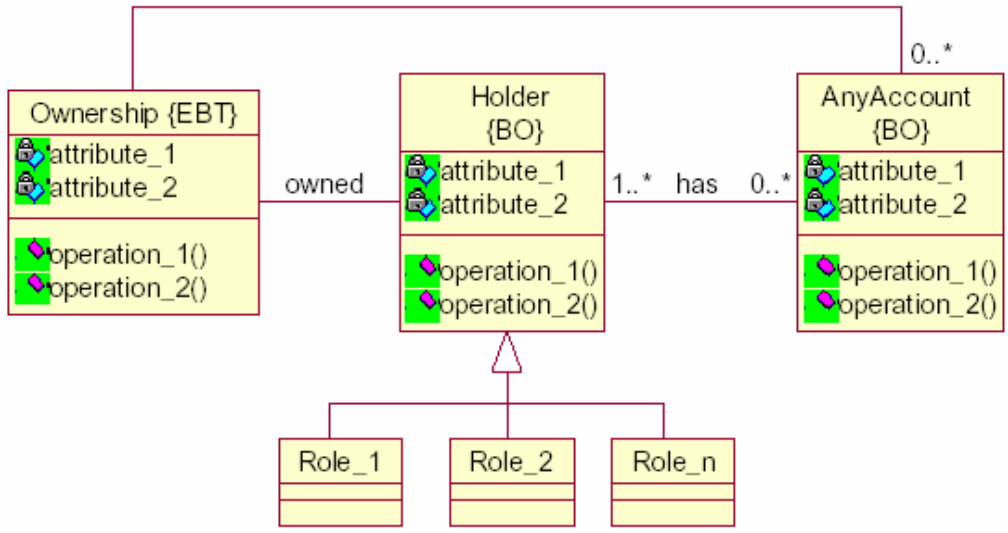


图 15 “AnyAccount”模式的类图

示例 2 建立“AnyEntry”模式

通过与实现“AnyAccount”相同的步骤，构筑另外一个称为“AnyEntry”的模式。该模式获取任何独立于问题环境的核心元素。图 16 给出了“AnyEntry”模式的结构。

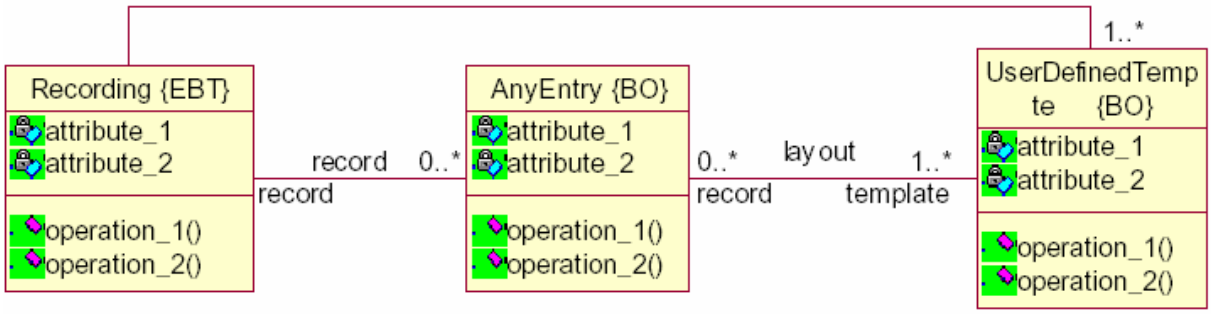


图 16 “AnyEntry”模式的类图

下一个模式

在建立了稳定分析模式后，需要考察如何使用建立的稳定分析模式。

模式 3.8 使用稳定的分析模式

目标

将已经建立的稳定分析模式应用到实际环境中。

上下文

将稳定的分析模式应用于实际中的建模问题。

问题

如何使用建立的稳定分析模式在不同的环境中建模问题。

约束

分析模式通常具有较高的抽象层次，因此，需要对其进行适当的示例以将其应用于特定环境中。在许多情况下，在建立一个复杂问题的模型时，需要集成多个稳定的分析模式。

解决方案

为了利用建立的稳定分析模式，应该注意以下几点：

1 基于应用的环境为问题中每个 EBT 和 BO 选择适合的属性和操作。

2 基于系统中确定的 BO 来确定 IO。无论是主要的或是隐含的 BO 一般都有对应的 IO 用于物理地表示它们。但是，在某些情况下，在问题的 BO 和 IO 之间没有一对一的映射关系。

3 为被建模的问题或系统确定其它的 EBT、BO、IO。

4 在应用时，涉及多个模式情况下，需要定义不同的模式连接的层次（连接应该发生在 EBT 层，还是 BO 层，或者是 IO 层依赖于问题本身的属性）。

讨论

为模式中每个 EBT 或者 BO 选择的名称将保持一致性。然而，在一些情况下需要修改 BO 的名称以使意图更加明显。而 EBT 的名称一般不要进行修改。

示例

此部分将提供两个示例显示如何在特定环境中使用开发的模式（“AnyAccount”“AnyEntry”模式）

示例 1 建模复印机帐户系统

该简单问题表明如何使用“AnyAccount”模式建模大学中的复印机帐户问题。大学中每个学生有一个帐号，用于访问集中控制的复印机。

图 17 给出了复印机帐号的对象图。映射问题 BO 的 IO 都已经确定。对 BO“Student”来说，“AnyAccount”模式没有继承可用，因为每个帐户应该只有一个持有者。对 BO“Account”而言，一个可能存在的物理表示是 IO”Code”。每个学生有一个“Code”以便使用复印机。如果对 BO“Account”有其它的物理表示当改变当前的 IO“Code”时，所有的 BO 需要建立，并将新的 IO 不影响核心地插入模型中。在该问题中，没有额外的 EBT、BO、IO。

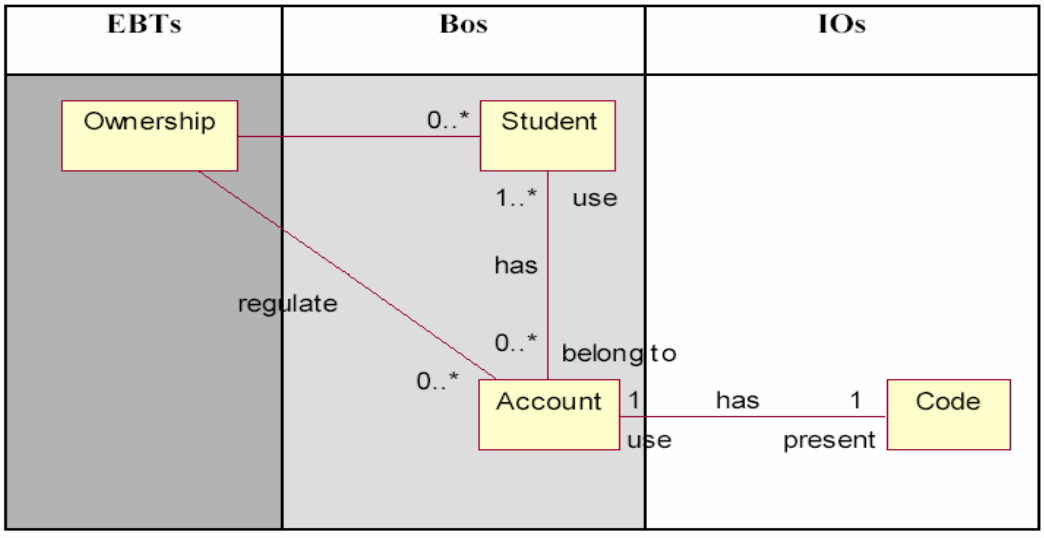


图 17 复印机帐户系统对象图

示例 2 建模 HotMail 帐户

该示例表明如何在一个模型中集成多个模式以解决复杂问题。问题的目标是利用两个构建模式:“AnyAccount”“AnyEntry”模式建模简单的 HotMail 帐户。为简化，仅列出了问题模型的对象图。应该注意到该模型尚未完成。使用它主要是为了能够说明问题。图 18 给出了其对象图。

对每个 BO 都将其对应的 IO 表示出来。例如，HotMail 是 HOST 的可能的物理表示，但是，从一般性考虑，该 IO 在任何时候都可以发生变化以表示任何可能出现的 HOSTS，而且不会影响模型的核心。在该示例中，两个模式中所有的连接都仅建立在 IO 层。

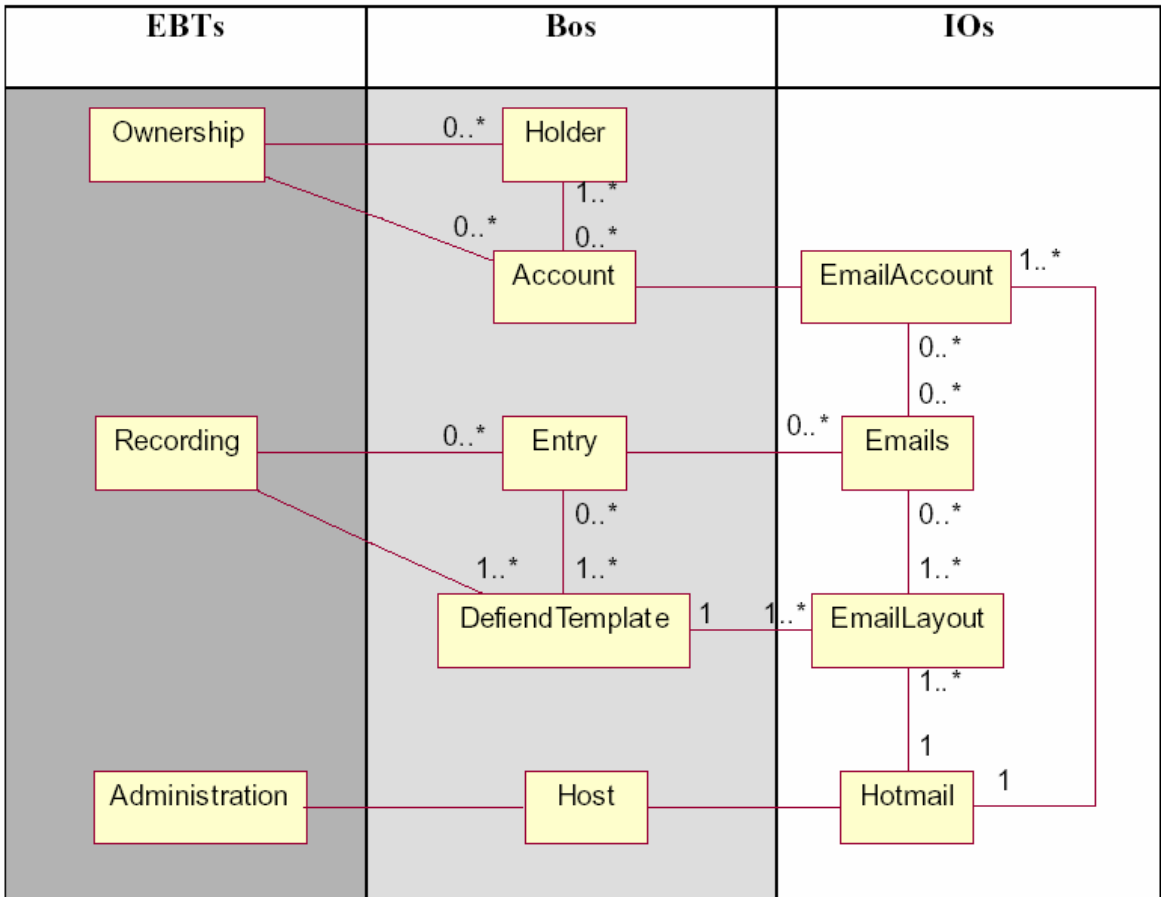


图 18 HotMail 帐户系统对象图

结论

该模式语言提供了一种在建立分析模式时获得稳定性的方法。通过该方法，可以增加有效性，提高分析模式的可重用性。未来的工作包括通过将该模式语言应用到不同的应用环境中来证明该有效性。

参考文献

[1] A. Geyer-Schulz and M. Hahsler, “Software Engineering with Analysis Patterns”, Technical Reports 01/2001, Institut fur Informationsverarbetung und-wirtschft, Wirschaftsuniversitat Wine, Augasse 2-6, 1090 Wine, November 2001.

[2] E. B. Fernandez and X. Yuan, “An analysis pattern for reservation and use of reusable entities”, Pattern Languages of Programs Conference, Plop99. http://st-www.cs.uiuc.edu/~plop/plop99.

[3] E. Gamma et al., “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.

[4] F. Buschmann et al., “Pattern-Oriented Software Architecture, A System of Patterns”, John Wiley & Sons Ltd, Chichester, 1996.

[5] H. Hamza, and Mohamed E. Fayad “Model-base Software Reuse Using Stable Analysis Patterns”, ECOOP 2002, Workshop on Model-based Software Reuse, June 2002, Malaga, Spain.

[6] M. E. Fayad, and A. Altman, “Introduction to Software Stability”, Communications of the ACM, Vol. 44, No. 9, September 2001.

[7] M. Fowler, “Analysis Patterns: Reusable Object Models”, Addison-Wesley, 1997.

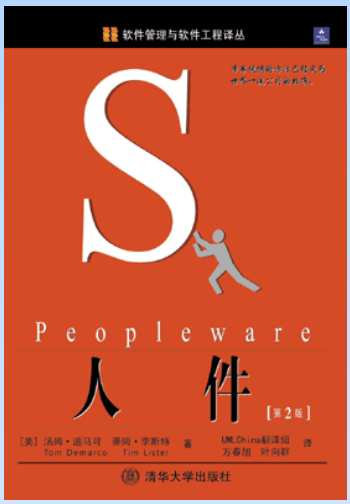
[8] M.E. Fayad and H. Hamza. “Introduction to Stable Analysis Patterns”, Communications of the ACM, Thinking Objectively, Vol. 45, No. 9, September 2002.

[9] M.E. Fayad and H. Hamza. “Comparative Study of Analysis Patterns”, Communications of the ACM, Thinking Objectively, Vol. 45, No. 11, November 2002.

[10] M.E. Fayad and M. Laitinen. Transition to Object-Oriented Software Developments, New York: Wiley & Sons, August 1998

[11] R. T. Vaccare Braga et al., “A Confederation of Patterns for Business resource Management”, Proceedings of Pattern Language of Programs’ 98 (PLOP’98), 1998.

老板，别把开发人员当成牲口--《人件》热销中



Tom Demarco，Tim Lister

翻译：UMLChina 方春旭 叶向群

http://www.peoplewarecn.com

人件中文版网站

Frederick P. Brooks, Jr--《人月神话》第 19 章

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品。当人件第一版出版时，我写了一份评论，“我强烈推荐你买一份《人件》给你或你的老板，如果你是老板，那么为你部门的每一个人买一份，并给自己买一份”。这建议在 12 年后的第二版依然有效，并且更加热烈。

Mark A. Herschberg

我只学了办公环境的章节，就辞掉了原来的工作！

Joel Spolsky

我想，微软成功的原因之一就是公司里的所有经理都读过《人件》

Steve McConnell

由于本书第 1 版的赫赫声名，新版的《人件》是我不用看就会决定购买的少数几本书之一。

购买>>

亮出怀疑的尖刀

Peopleware 著

查看评论

《人月神话》关注“软件开发”本身，《人件》则关注软件开发中的“人”。《人件》中译本封面上说“伸张开发人员的权利”，网络上还有一句更激烈的话“老板，不要把开发人员当成牲口！”。中译本出版后，有人叫好—“每个开发人员都应该看一看”，有人哀鸣--“在中国，这些做法行不通的”--这本书出版后，是否就能够被很快广泛接受，使软件开发人员的环境带来显著改变呢？答案看来是否定的，原因：

1. 积习难改——软件公司里不断的加班，成堆的口号，狭小的空间，嘈杂的环境已经成为了我们的习惯。现状即使再不合理，改变也是困难的——对这一点，《人件》的第 30 章作了专门阐述——旧的现状必定要经过中间的混乱阶段，才能到达新的现状。幻想一步登天，最终收获的只能是失败和失望。软件公司的改革者往往会跌倒在中间阶段，因为他“以从公司旧秩序受益的所有人为敌，却只拥有冷淡的辩护者”，而“抵抗变化”联盟的强大出乎意料，害怕改变本是人类天性，人们宁可逃避现实（就象“黑客帝国”中的叛徒宁可回到发电厂），或者虚构一些“容易”的解决方案，这无异于“在一条黑暗的街上丢掉钥匙，却到邻近的另一条街上去寻找。因为这条街上的灯比那条街上的要亮一些。”（《人件》第 1 章）

2. 诱惑难当——在公司里做几年，媳妇熬成婆后，又可以开始去压榨下一批开发人员，从中获得乐趣。如果说意识深处我们确实讨厌被压榨，却不讨厌压榨别人。各种软件公司的习惯得以维持，并非仅仅依靠口号和强制，而在于黑暗面的诱惑（正如《人件》第 29 章中虚构的“星球大战”片断），它总是诱惑那些企图摆脱被“管理”命运的人们说：“成为人上人，这是唯一的道路！”所以它才会有源源不断的追随者和拥护者，它就是《黑客帝国》中 Matrix 的发电厂，源源不断地为 Matrix 提供奴役人类的能源。这种诱惑象病毒一样可以传染和遗传，使得软件公司里改变现状的努力一次又一次成为泡影。

（如果您是一位经理，想要寻找“管理”开发人员的牧羊之术，不要往《人件》里找，中国的史书就是一个最大的宝库——《资治通鉴》里的整人智慧博大精深。）

如此看来，象《人件》所提倡的一般改进开发人员的环境前路茫茫。一个普普通通的开发人员能做什么？首先能做的只有怀疑，用怀疑的尖刀一次又一次轻轻地刺激梦魇中的 Neo（就象《人件》26 章中的“唤醒霍尔加”）。怀疑加班，怀疑进度表，怀疑口号，怀疑开会，怀疑老板的说话，怀疑家具警察，怀疑工作环境...仅仅是怀疑而已，但，仅这一点就已经是致命的。我怀疑你的时候，你说起那些话来，必定不自然，你做起那些事来，必定不

自然，而且要下意识地做一些事情掩饰，当大家的怀疑到了极点，老板必定会狼狈不堪。

这些怀疑什么时候产生改变？借用《倚天屠龙记》里的一段话：“五弟，那日在船中你跟我比拚掌力，我所以没伤你性命，就是因为忽然间想起了空见大师。”

这样的怀疑已经开始，我们在 CSDN 上不断看到类似下面的话：

“已经做过两年比牲口都不如的工作！”

“就是变相软禁！”

“现在提起程序员就是加班、熬夜的代名词！”

“我只是一台被老板百般盘剥的编程机器罢了……”

“禁止工作狂，这是危险和短视的，长期工作会让人失去思考的能力！”

“有阳光的地方被老总占了！”

请参加 CSDN 的讨论，亮出你怀疑的尖刀！>>

信息时代的技术阅读

刘天北 著

查看评论

据说，我们生活在 一个“知识爆炸”的时代。作为专业软件开发者，我对此颇有些切身体会——事实上，我所在的行业就是这次爆炸的重灾区。选择开发软件，就是选择被弹坑包围的、近乎偏执的思维方式：在这样一个产品按照摩尔定律更新，技术以井喷速率向我们涌来的产业中，永不停顿的学习和追赶技术前沿似乎成了唯一可行的生存之道。

要是在这种语境里谈论“教材”，未免会显得笨拙和老套。“计算机教材”，在不少人听来简直是一个矛盾修辞（oxymoron）。每天，甚至每小时都会有几种新技术、或是核心技术的新版本出现，大多数开发人员都在为赶上趋势疲于奔命，若干年前出版的一本旧书与我们何干？教材至多是一瓶过了气儿的酒，代表着那些早该归档入库的旧概念、旧思路。是呀，在知识爆炸的弹坑里为计算机教材辩护，不啻于号召大家搬进博物馆办公。

毫无疑问，如果上述种种全都属实，那软件行业所面对的就只能是一幅狰狞的图景。任何平凡的个人，在无止境的逐新挑战下，都终有一天会力不从心、败下阵来。素闻软件开发是“青春饭”，30 岁是开发人员的退休年龄，这些都是上面的知识规律的有效推论，并也在国内不少地方已经成为令人难堪的现实。

我反抗事物的这种状况。依我看来，上面给出的，只是关于知识的一幅似是而非的漫画，远没有描绘出软件业内恒常与革新之间、原理与应用之间、教材与实践之间种种复杂而细微的张力和互动。事实上在不少软件业发达的国家，企业的技术骨干往往是 30 至 40 岁的开发者；相应地，不少十几年、几十年前出版的教材至今仍被奉为圭臬。一个例子？Marc Fleury，JBoss 的主要设计者就曾坦承，这种当前构架最先进的应用服务器中的核心技术“客户端拦截器（client side interceptor）”，其灵感源自 60 年前 Allan Turing 关于自动机的论述。从这个意义上讲，一本书页发黄的教材中包含的现实性，远比不少刚出锅的资料更丰富。

近年来“复用性”是软件开发追求的重要目标。在学习领域，研读教材不失为提高学习过程可复用性的上选策略。所谓举一隅而反三隅，我见过一些基本功扎实的开发者，学习任何新应用的速度都远逾常人。可见多读教材，熟知技术应用背后的原理，能使人以少许胜多许，进而降低学习新事物的成本，摆脱面对前沿技术的被动态度。

要说明上面的道理，恐怕首先得看看“教材”和“教程”的区别。教材（textbook）是某一学科领域的经典读物。我们称赞一个漂亮的解决方案，常说“这可以写进教科书”，含义正在于此。与之相对，教程（tutorial）是对特定技术的使用介绍。在一些短视的、急功近利的语境中，这二者往往容易混同，甚至不少人轻教材而重教程，看到内容稍微详尽的教程就大呼“经典”——这只能是心态失衡引发的谵妄。教材和教程虽然名称相近，实质上却代表了知识的两种极端样态：教程让人知其然（know-how），教材令人明其所以然（know-why）；教程使用概念，教材塑造概念；教程倾向于灌输，教材偏爱启发；教程重配置，教材重原理。俗套的说法还爱用主食/副食，或是武侠小说中的内功/招式的对比来描述这二者间的关系。依我看来，这些隐喻未必搔到了痒处，但都肯定了一点：无论对于个人还是行业，单纯推崇教程而忽略教材，都只能将自身引向灾难。

据说不少人由于小时被迫背诵诗歌，成人后患上了叫做“恐诗症”的心理疾病。我猜想那些反感教材的人们，也是国内可悲的教育体系的牺牲品。这些人想起教材，脑中浮现的首先是一种难以下咽、无法消化的锯末类食物。但在我个人接触的国外技术读物中，好的教材远比好的教程可读。国外教材中的名著往往为大师巨匠所作，发达国家的软件业看重表达能力，光有内在美，不善交流也不成——所以越是名家手笔，文章越平易，表述越直观。大师之所以为大师，在于他们对概念把握得更准确，并能以最明晰的方式传播给世人——当他们在著作中首次引入某个概念时，就更在意表达方式是否易于接受。所以在你自己足够清醒的时候，好的教材往往并不使人如坠雾中，反而会时时引发惊叹：我怎么就没想到这个！

教材的可读性优势还在于它对“参与”的鼓励。一般我们看到的各种教程，多为国外企业的商业文档的衍生物。所谓商业文档，大致是使用手册（user manual）或是速成指南（quick start）性质的技术资料，叙述风格以线性的平铺直叙为主，喜欢“手把手”的灌输。这是麦克卢汉（Marshall McLuhan）所说的“热媒体”，学习者的参与程度降到了最低。相反，好的教材的叙述总是启发、对话式的，主题往往在循环中得到深化。故此教程往往诉诸运动神经，它是一种指令，让人不由自主的跟从和忙碌，而教材则使人从盲动中抽身而退，赋予人独立思考的能力。

以我个人所见，技术行业里，大师巨匠大多自信、安恬，温和有趣，水平低得多的技术人员反而常常自矜于一得之见，对旁人摆起面孔。相应地，很多教材名作也都面目有趣，语言温醇，甚至不少段落漂亮得都可以拿去做英语范文。而你很难从大多数教程作者那里指望考究的文风，他们的典型句式是：“读完下面的 10 页，你将学会 a…b…”——要是再加上一个电话号码，这就能构成不错的电视购物广告。

在读书方法上，据说，波普尔（Karl Popper）刚到伦敦大学经济学院（LSE）时，发现学生课程里有一门叫“速读（speed-reading）”。他先生居然跑去问院长，他能不能开门课教人“慢读”。即使是对于计算机教材，卡尔爵士的建议也应该有效：别指望今晚读的教材明天就变成现钱，扫描仪式的线性摄入在这里不起作用。越是名著经典，阅读时越需要享受的、反复体味的态度：应该像吃人参果一样善待这里的一字一句，而不是老想着把它当成果味 VC，一口吞了。另外，依个人经验，我觉得带着问题读教材往往更合理——这更容易使读者进入富于启发的对话过程。

怎样从大量的教材中找出合适的那本，也是一个困难的选择。人生苦短，读书作为时间上的投资也须慎重，开卷之前看看权威的意见——目前国外有专门的网站（而不只是网上书店）作这件事——对选择阅读，对读书时更好地把握该书的长处和弊病，都会有一定帮助。此外，我个人（不无偏颇）的倾向是：首先考虑国外原文教材。语言远不止是思想的载体。一个核心概念与表达它的语言总是处于一种非此不可的关系中。英语是计算机科学的世界语（相当于拉丁语对于神职人员的作用），要在问题的核心考虑一项技术，再精确的译文都是离题万里的。加之国内技术书籍的翻译现状远逊于一般文献的水平，除个别情况外，译者往往不是国内特定领域的首选专家，无论技术水平还是语言能力都与原作者相差太远。在原文里一个微妙而尖锐的段落，经过译者强作解人的滤波和转换，往往平庸得可以去作社论，对于塑造核心概念的教材就更其如此。

Habent sua fata libelli——西方的很多图书馆内都有这句拉丁文，大意是，书籍自有其命运。而我看来，我们能读到什么书，甚至进一步说，我们如何阅读、如何思考，很大程度上取决于出版/传媒界的生态状况和图书从业者的素质。近年来，计算机教材在国内似乎运数不差。随着著译增多，人们渐渐也对何谓佳作、何谓经典有了趋于合理的认识。前些年大家把一些具体语言开发教程奉若神明的风气，似乎也有了一定转变。其实，在软件业发达的国家中，人们对技术书籍早有成熟的衡量标准——这是一套严密的价值体系，新书的出版，只能依据这个体系找到特定的位置。好比说书名中包括“*** Bible”、“*** in 21 Days”的，究竟针对什么样的读者，大家心里自然有数。而国内出版者对这一套潜在的规则的理解也许还有待加深，很多时候还是依靠一些所谓“buzz words”来判断是否引进。事实上，在信息全球化的今天，图书从业者掌握类似的知识无需成为技术专家。这里只要恰当的英语能力，也许再加上一点对图书的热爱就已足够。在一篇谈软件工程经典《人月神话》的文章中，我曾经回忆在一次印度之行中购买影印版教材的情形，我也愿意以此结束本篇：“…第三或四层是技术专区——我开始只找到了 Jacobson 们的 OOSE（有点困惑于复杂的书店布局）——看到另外一人手中的 Software Project Survival Guide——我求助于一个店员，他立刻拿给我那本书，和一本《人月神话》——另一个店员看到它们，又递来一本《人件》…”。对于一个技术读者，这是难以忘怀的体验。**（本文原载于《中华读书报》）**