第 26 期（2003 年 6 月）

工具

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/

Microsoft 希望与 OMG 重修旧好

[2003/5/12]

经过了和对象管理组织的连年对峙之后，Microsoft 现在似乎想要和 OMG 重修旧好了，它希望加入该组织，利用 OMG 的架构技术。

有消息称，Unisys 公司打碎了 Microsoft 和 Needham 之间的坚冰，这两者曾一度关系紧张，尽管 Microsoft 仍是后者的成员之一。双方在 90 年代后期分处两大对立阵营，一方是 OMG 支持的公共对象请求代理架构 CORBA（Common Object Request Broker Architecture），另一方是 Microsoft'的 COM（Component Object Model ），这两个都是分布式计算模型。

消息称 Microsoft 正在向 OMG 靠拢，用 Redmond, Wash 的话说，企业目前都在投入更多的精力钻研建模和架构技术，而其中的两个领域 OMG 都掌握着关键的规约（specification）和技术： UML (Unified Modeling Language) 和 MDA (Model Driven Architecture)。今年，Microsoft 已经赞助了两个为期四天的 OMG Web services workshops ，一个在德国的 Munich，另一个在上月于 Philadelphia 召开，会上 Microsoft 的代表做了关于 面向服务架构（services-oriented architectures）和 MDA 的讲演。Unisys 是 OMG 成员之一并且和 Microsoft 有着紧密的合作伙伴关系，他们和 Microsoft 举行了联合报告，题目是“Microsoft 的 'Jupiter' 和 Unisys 的 MDA 过程”。

“Microsoft 最终是需要 MDA 来简化其.Net 的”，Grover Righter，Kabira 技术公司主管技术的副总裁，这么认为。一位名为 San Rafael Calif，网络服务和其它环境的底层架构软件和框架的提供商认为：“这个投资非常必要，它不仅是明智的，而且同样是对.Net 的一种保护：避免受到来自较老一些的语言（第三代语言）的威胁”。

现在，Microsoft 已经在其 Visual Studio .Net 企业架构版本中支持 UML，并且有消息称，Microsoft 计划在 Jupiter 中支持 MDA ，Jupiter 是其即将问世的电子商务软件（e-business suite and collaboration portal software）的代号。.

另外一个和 Microsoft 关系密切的消息来源称，由于企业系统的日益复杂，该公司对模型架构的兴趣日益浓厚。

对模型和 OMG 的示好并不代表一切，酒桌下的短兵相接依然存在。据有关消息，Microsoft 对 OMG 在 MDA 上的有些做法如代码自动生成等并不赞同。

OMG 的 CEO Richard Soley 对 Microsoft 是否重新加入组织不置可否，但他认为， OMG 将欢迎该公司作为成员加入。Microsoft 的会员资格已经在去年作废了。

“这不是谁喜欢谁的问题，关键是谁干成了”，Soley 说：“Microsoft 是 IT 行业最重要的开发商之一，他们自然明白在一个开放性标准组织中交互和共同工作的价值，他们会是一个非常受欢迎的成员。”

截至发稿时间，Microsoft 官方并没有对此作出任何评论。

（风自由 摘译，不得转载用于商业用途）

Rational, WebSphere 和你

[2003/5/30]

Rational 已经成为 IBM 的分部，对 WebSphere 而言，这意味着什么？ 请看 WebSphere 顾问对 IBM Rational 的 Eric Schurr 的采访。

采访者：Ellie MacIsaac 主编，WebSphere 顾问

2003 年 3 月，IBM 完成了对 Rational 的收购，相关整合正在全力进行之中－虽然这次 IBM 采取了与收购 Lotus 以及 Tivoli 相当不同的方案。相信在不久之后，Rational 会作为 WebSphere 工具集的一部分出现。

Eric Schurr，IBM Rational 市场部副总裁，将和我们分享关于 Rational 将对 WebSphere 产生怎样的影响这一问题上他的个人观点。并且还将透露 Rational 新产品的特点以及在简化 WebSphere 开发上的支持。

顾问: Rational 成为了 IBM 的分部，这对 WebSphere 意味着什么?

Schurr：Rational 是 IBM“按需计算”策略的软件开发平台。因此，如果你需要选择全面的软件开发商，Rationa 可以用于“构建（build）”， WebSphere 和 DB2 可以用于“运行（run）”， Tivoli 可以用于“管理”，而 Lotus 则可以用于以上这些的协作。

对 WebSphere 的开发者而言，这意味着更全面的整合的并且是松藕合的解决方案，可以帮助简化服务的提供。最终的目的是要为 WebSphere 的用户提供更好的服务，因为现在我们已经是 IBM 的分部，这样我们可以优化我们的资源使用，彼此共享任何信息来保证这些产品整合得更好。过去我们也曾和 WebSphere 的团队合作多年，但现在，正如你们所看到，关于整合的计划和创造力都有飞跃性的进展。

顾问：请告诉我们 Rational 现在为开发人员都提供了哪些产品。

Schurr：我们正在发布一些新的产品，并加强我们现有的产品线，主要可以分成三大类：快速应用开发，扩展的开发，以及发布一个更易于配置的开发过程。

快速应用开发

使用 IBM Rational 的快速开发工具，拥有各种开发人员的组织可以很方便地开发 N 层应用。其中一些开发人员也许有着丰富的 J2EE 开发经验，而重要的是：大部分人没有。分析家指出，目前全世界大约有 1.5 万 Java 开发人员，但是下一代 Java 开发人员中大多都不会是 J2EE 专家。他们的成功，需要技术来帮助他们创建基于 J2EE 的应用，而无需了解 J2EE 的技术细节。

顾问： 能否解释一下“架构的 RAD（architected RAD）”的概念?

Schurr：“架构的”意味着这是一个基于模型的产品，也就是说，采用 UML 的元素来创建模型。这样，你可以保证应用是可靠并且可伸缩的。90 年代的 RAD 应用的构建确实很快，但并不一定就好，最后受害的还是使用者。IBM Rational 的快速开发工具将帮助你将 RAD 技术用于快速地构建应用，而且由于是基于架构的基础进行构建，应用的可靠性、伸缩性以及可管理性都可以得到保证。 为了简化与遗留系统的集成，可以联系 Rational Rapid Developer 得到各种现有数据库和遗留的应用以和现有的应用集成并进行扩展。今天，开发新产品的时候，更多的组织努力重用过去的工作成果。

顾问： IBM 宣称，年底之前将在 WebSphere 产品家族中集成更多的 RAD 特性。IBM Rational 的 Rational Rapid Developer 是否正在朝着这个目标努力?

Schurr：差不多吧；IBM 一直都在努力为用户提供更多的 RAD 特性。Rational Rapid Developer 是帮助那些不是 J2EE 专家的开发人员的 WebSphere Studio 的姊妹产品。在一个项目中可以同时使用 Rational Rapid Developer 和 WebSphere Studio。例如，对于一个 10 个人的开发团队，2 到 3 个 J2EE 专家就够了，他们使用 WebSphere Studio 和 Rational XDE，其它人使用 Rational Rapid Developer 开发其它部分，最后将他们的工作整合在一起。

扩展的开发

为了体现 IBM 对开发人员的承诺，XDE 面向 WebSphere Studio 和 Eclipse Rational 的插件被更名为 Rational XDE Developer。Rational XDE Developer 内置于 WebSphere Studio 5、Eclipse 2 以及 Microsoft Visual Studio .NET 中。为了扩展开发能力，Rational 的 Purify 插件被集成到 Rational XDE 中， Purify 中包括了内存管理、运行时分析、代码覆盖（code coverage）以及其它优化代码的工具。这些功能都整合进了 WebSphere 中的 XDE 中，以帮助开发人员解决头疼的调试问题。另外，还可以利用 Rational XDE Developer 来构建 UML 模型，进行可视化追踪（trace）和调试。

对 JAVA 测试人员来说，同样有新的产品：Rational XDE Tester。它是自动化的 Java 和 Web 应用测试产品，可以作为 WebSphere Studio 5 和 Eclipse 2 的插件运行。

开发过程

过去几年中，Rational 将 Rational Unified Process (RUP) 改进成为了一个组件化的架构，使用者可以根据组织的需要增加或者删除组件。 开发人员都对 RUP 感兴趣，因为它给予了软件开发统一的基础、术语和工作定义。他们同样认识到每个项目和组织都是不同的，不同的组织需要将那些对所有组织都适用的内容应用到具体的环境之中，并根据特定的项目进行优化。Rational 已经对 RUP 进行了优化，组织可以在多个层次上对其进行配置 - 根据组织、项目以及个人的情况。

顾问： 各个层次都可以配置哪些功能？

Schurr：在组织层，我们对组织对过程进行重构的能力进行了改进。可以在基础的层次上进行改造，以生成对全组织适用的组织过程，可以对 RUP 进行全面的定制。

而各个项目又各不相同，RUP 现在包含一个加强的组件 RUP Builder，可以帮助用户来定制 RUP 以及添加插件。这里所说的插件是指用户创建来为 RUP 增添内容的模块，现在可用的有 30 个左右。详细信息可以访问 http://www.rational.com。

从项目中的每个开发人员角度考虑，每个人都只关心 RUP 中相关的方面。我们增加了一个新的工具 MyRUP，它可以帮助用户根据个人的角度来定制 RUP，从用户特定的角色出发关注 RUP 中的特定方面，可以更改或者删除内容。

更多

Rational 正在为其产品线作各种改进。例如 Rational ClearCase 现在加强了对团队开发的支持并和 WebSphere Application Server 5 紧密结合在一起。Rational 声称其工具套件也被改进，其安装、管理和配置都更加简化。其它的改进包括 Rational Project Console（Rational 的项目管理系统）中提供了新的报告和能力。

（风自由 摘译，不得转载用于商业用途）

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

《有效用例模式》
中译本即将上市

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

第一本 UMLChina 指定教材

清华大学出版社

UML 相关工具一览（2003 年 6 月版）：A-H

整理

查看评论

两年以前，《非程序员》第 2 期曾经刊登了“UML 相关产品价格”一文。两年过去了，UML 相关工具的发展又如何呢？以下总结了全世界的各种 UML 相关工具，都放上来供大家选择。工具的品种相对于两年前多了许多，总共有一百多种，所以我们以连载的方式刊登。按工具名称字母排序，本期刊登 A-H。

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				
4Keeps v4.2	4Keeps, Inc.  http://www.4keepssoftware.com/4Keepsdm.htm	30 天试用	√	×									Windows	Visio5.0、2000 的插件
Advantage Joe 3.0	Computer Associates  http://www3.ca.com/Solutions/Product.asp?id=257										Java		Java	就是以前的 COOL:Joe。针对 UML 构件图的 EJB 工具。

AllFusion Component Modeler 4.1	Computer Associates  http://www3.ca.com/Solutions/Product.asp?ID=1003		√	√	√	√	√	√	√	√	Java、Visual studio、Forte、PowerBuilder、C++、IDL、VB、DDL-SQL、Smalltalk。	√	Windows	就是以前的 Paradigm Plus。支持 Catalysis 方法，和 BPwin、Erwin、ModelMart、Jasmine 实时集成，支持过程建模。
alma 0.39	欧洲 Alma Team  http://www.memoire.com/guillaume-desnoix/alma/index-en.html	开源	×	√	×	×	×	×	×	×	Java, C, C++, BDL,Delphi, Eiffel, Python, IDL, ODL, SQL, Lisp, HTML, XML, XMI, Fortran		Java	可以读取 Rose 的 mdl 文件
AnyStates 3.0	XJ Technologies（俄罗斯）  http://www.xjtek.com/products/anystates/	有评估版下载			√						C++, C#, Java, J#, gcc/pgcc(Linux)	×	Windows, Windows CE	UML 状态图编辑器和调试器。可以和 Visual Studio 集成。
AP Studio	Kedar Patankar, Binoy Samuel and the Demeter Group（Northeastern University） http://www.ccs.neu.edu/research/demeter/APStudio/ap-ad.htm			√							Demeter/Java		Windows, Solaris	开发 Demeter/Java 程序的 UML 工具
ArcStyler 3.1	Interactive Objects Software（德国）										Java, C#, Web Accessors, EJB 1.1, EJB 2.0, BEA WebLogic 7.0 (EJB			覆盖 J2EE/.NET 系统开发工作流程的套件，遵循 RUP，有针对 Rose 的插

	 http://www.io-software.com/products/arcstyler_overview.jsp									2.0), JBoss 2.4.4,, ASP .NET			件，OCL、MDA 支持。
ArgoUML v0.13.1	Tigris.org  http://argouml.tigris.org/	开源	√	√	√	√	√	√	√	√	Java	√	Java	最流行的开源 UML 工具，支持 OCL，支持认知式开发，不再只是画图，例如可以自动评价设计、自动更正…等等。
ARIS 6.1	IDS Scheer（德国）  http://www.ids-scheer.com/english/index.php		√	√	√	√	×	√	√	×	Oracle，SAP	×	基于 Web，平台无关	强有力的业务流程套件，为业务流程设计引入 UML。
BetterState 6.1	WindRiver  http://www.windriver.com/products/betterstate/index.html	有 Lite 版			√						C, C++, Java		Windows	在 UML 状态图或 PetriNet 和代码双向工程，可直接运行在嵌入式操作系统如 VxWorks、OSEKWorks 平台上
Bold 3.2	BoldSoft(Borland)  http://www.borland.com/		√	√	√	√	√	√	√	√	Delphi, C++, COM, XML, SOAP		Windows	BoldSoft 原为瑞典公司，2002/10 已被 Borland 收购。功能已并入 Borland Delphi 和 Borland

														C++ Builder 中
BridgePoint 5.1	Project Technology PROJECT TECHNOLOGY INC. http://www.projtech.com/	有评估版	√	√	√	√	√	√	√	√	C, C++		Windows, Solaris	使用 XT UML（Executable and Translatable UML），支持 Shlaer-Mellor 方法，在生成代码之前可以对 OOA 模型进行测试和验证
Cittera 6.12.01	Canyonblue Canyon Blue http://www.canyonblue.com/	可以试用	√	√	√	√	√	√	√	√	Java, C++		Java	基于 Internet 的工具，支持 UML1.3，支持协作建模和知识共享。还有 Real-time 版本可以选择
ClassBuilder 2.4	Jimmy Venema（荷兰） http://members.lycos.nl/JimmyVenema/ClassBuilder/ClassBuilder.htm	免费	×	√	×	×	√	×	×	×	C++	×	Windows	专门针对 C++用户。精细的顺序图编辑器。可以以 RTF 和 HTML 格式产生文档。
Codagen Architect 3.0	Codagen Codagen http://www.codagen.com/products/architect/default.htm	15 天评估	√	√	√	√	√	√	√	√	VB, C#, C++, Java		Windows	遵循 MDA 流程，能产生超过 90%的 J2EE 和.Net 平台代码。2003 年获得 Jolt Productivity Award software development 13th annual productivity award

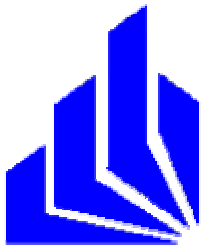
														支持 Rational Rose 2000 或更新版本，Borland Together ControlCenter 6.0，或带 Visio 的 Visual Studio .NET for Enterprise Architects。
Code Logic 1.4	Logic Explorers  http://www.logicexplorers.com/products/codelogic/details.html	有试用版		√			√				Java		Java	强有力的、动态的双向工程，特别是顺序图自动生成。
CodeModeler 1.5.1	Aruba Development  http://www.arubadev.com/	有演示版	√	√	√	√	√	×	×	√			Windows	
Cohesion 1.0	Team Synergy（澳大利亚）  http://cohesion.it.swin.edu.au/teamb/cohesion.shtml	开源											Java	元模型建模工具，可以使用 UML，OCL，或者其他喜欢的建模符号。
Composum 1.3	IST	可以试用	√	√			√					√	Java	可以导入 Rose 模型

	http://www.ist-dresden.de/													
Cradle 4	3SL（英国） http://www.threesl.com/													强有力的需求管理和系统工程环境，支持 UML。
Development Accelerators	Blueprint Technologies http://www.blueprinttech.com/Products/Accelerators.asp													模式&框架工具，与 Rose 整合，支持 Gamma、Buschmann、Fowler、Hay 的模式，也可以自定义模式和框架，支持模式的模糊查询。
devine	Tom Jones http://www.tojosoft.de/Produkte/dvine/dvine.html	有试用版		√							Delphi		Windows	在 Delphi 代码和 UML 之间转换
Delphia Object Modeler（D•OM）	Atos Origin（法国） http://www.si.fr.atosorigin.com/rhone-al		√	√	√	√	√	√	√	√		√	Windows, Unix (Solaris, Linux, AIX, HP-UX)	UML 原型工具，支持到目标架构的转换规则。支持 HTML 和 RTF 文档化。

	pes/Dom/													
Describe 6.0	Embarcadero  http://www.embarcadero.com/products/describe/dedatasheet.asp	15 天试用	√	√	√	√	√	√	√	√	C++, Java, CORBA IDL,	√	Windows, Solaris	自称为“下一代的 UML 工具”。C++, Java, IDL 的双向工程以垃圾代码少而颇受好评。提供对 Visual Studio .NET 2003 和 Sun One Studio Java 开发平台的集成支持。还可以和 ER/Studio, Caliber, DOORS 集成，支持协作开发。
Dia 0.91	Alexander Larsson http://www.lysator.liu.se/~alla/dia/	开源	√	√			√			√	C++, Java, CORBA IDL,		Linux(Debian, Redhat), Windows	类似 Visio 的工具。还支持 E-R 图
DocExpress 3.0	Telelogic  http://www.telelogic.com/products/additional/docexpress/index.cfm													从 ATA 购得后，DocExpress 已经成为 Telelogic 套件的一部分，可以与 Tau，Rose，Paradigm Plus 和 Aonix StP 结合产生可裁剪的 Word、Interleaf、Framemaker、HTML 文档。
Documenta	Henk Rippen（德国）												Windows	可以从 Rose 或

tor	http://www.rippen.de/index_e.htm													Select Enterprise 产生 Word 文档。
DOME (the DOmain Modeling Environment) 5.3	Honeywell Honeywell http://www.htc.honeywell.com/dome/index.htm	开源	√	√	√	√	√	√	×	×		×	Linux, MacOS, Windows, Solaris	元模型工具，如果需要添加你自己的标记法时很有用。有 ftp 站点供大家交换模型。
EclipseUML 1.2	Omondo OMONDO THE LIVE UML COMPANY http://www.omondo.com/index.jsp	有免费版										√		可以插入到各种 UML 工具中，
EctoSet Modeller 2.2	EctoSet（澳大利亚） http://www.ectoset.com/	有试用版	√	√	×	√	√	×	√	√	Delphi/Kylix, C++ Builder, Java, VB	√	Windows, Linux	强有力的内嵌工具
Elixir CASE 1.2.4	Elixir Technology（新加坡） Elixir Technology http://www.elixirtech.com/ElixirCASE/index.html	有试用版	√	√	√	√	√	√	√	√	Java	√		
Ensemble Suite	Ensemble Systems http://www.ensemble-systems.com/pro	有试用版												Rose 插件套件，支持 BEA, Weblogic, Tuxedo, SAP

	ducts.html													
Enterprise Architect 3.51	Sparx Systems（澳大利亚） http://www.sparxsystems.com.au/	有试用版	√	√	√	√	√	×	√	×	C++, Java, C#, Delphi, SQL-DDL, VB.NET, VB	√	Windows	便宜又好货。还包含需求管理、项目估算、测试支持。
Enterprise FrameWork	Ptech Inc http://www.ptechinc.com/													基于知识库的业务流程建模、BPR 工具。
ESS-MODEL 2.2	Eldean AB（瑞典） http://www.essmodel.com/index.html	开源	×	√	×	×	×	×	×	×	Delphi/Kylix, Java	√	Windows	支持 Delphi/Kylix, Java 双向工程
FL 0.4.12	Novosoft（俄罗斯） http://gemini.novosoft.ru/NS2B.nsf/w1/FL_technology	免费		√							DB/2, Oracle, Interbase, Microsoft SQL Server, MSAccess, PostgressSQL		Java	Rose 插件，简化基于 Java 的关系数据库应用开发。简化持久对象的生成，使用 OQL（对象查询语言）访问数据库。
FreeCASE	FreeCASE Project http://www.freecase.seul.org/details.html	开源											Windows, Linux	另一个开源项目，不过好像暂停了。
FUJABA 4.0	University of Paderborn Software Engineering Group（德国）	开源	√	√	√	√	√	√	×	×	Java		Java	学院派作品，支持 Patterns

	 http://www.uni-paderborn.de/cs/fujaba/													
GRADE Modeler 4.0	GRADE Development Group（拉脱维亚） GRADE tools http://www.gradetools.com/default.htm	有试用版	√	√	√								Windows	业务分析和系统分析工具，擅长复杂模型图的界面显示、界面操作，为你自动整理纷乱巨大的图－还带语音功能。可以和 Rose 交互。
HAT	E2S（比利时） HOORA http://www.hoora.org/		√	√	√	×	√	√	×	×	C++	×	Windows	HOORA（Hierarchical Object Oriented Analysis）方法原来是为欧洲太空总署（ESA）开发的一种面向对象方法m，提供了如何使用 UML 来开发软件的清晰指南。HAT 严格遵循 HOORA，可以和 Rose 交互
待续…														

新闻

《人件》已经上市>>

相关文章

《人件》实践之：SAS公司

关注程序员自己的文化——专访Tom DeMarco

别把开发人员当成牲口

Brooks在《人月神话》中的评论

Alan Cooper的评论

办公室空间，下一场革命

《人件》在计算机行业的实践（待续）

人的问题：关于《人件》

Edward Yourdon的评论

开发人员是人吗？

各版本封面

英文，日文，德文

现代需求规约

Donald Firesmith 著，wnb 译

查看评论

摘要

作为需求工程的重要任务，需求规约适当地文档化需求，以方便其使用者。从传统观点来看，该过程包含需求分析组在项目需求阶段使用字处理程序建立一个单一的需求规约文档。然而，系统开发的发展趋势使得该方法暴露出一些严重的问题。需求工具的改进和发展不仅使需求管理得到更好的保证，而且可以保证建立一致的、最新的、面向不同使用者的需求规约，更好地满足不同使用者的需求。

1 传统需求规约

70-80 年代，需求规约乍看起来是一项简单的任务。在项目最初的需求阶段中，需求工程师使用类似结构化分析的功能分解方法从项目涉众中捕获功能需求。然后使用简单的字处理软件将这些需求文档化，形成一个单一的需求规约。这样的需求规约在经过评估和简单的迭代后，放入配置管理控制范围内（也就是进入需求冻结状态），并分发给其用户。每个项目参与者都将其工作建立在同样的需求规约之上，并且确信该规约在接下来的设计、编码和测试阶段都不会发生显著的变化。如此来看，需求规约在瀑布式开发周期的最初阶段中是一个使用手工方式开展的任务，在项目进行的初期就基本完成。需求规约也是一个基于纸质文档的处理过程，即便是有工具支持也基本上不被采用。遗憾地是，目前许多项目仍然使用类似的方法产生需求规约。

存在的问题

当然，前述的需求规约任务的实现方法存在一些众所周知的问题。需求规约文档作为手工处理的结果，需要耗费大量的时间，完成该任务代价很高，并且在放入配置管理控制后很少对其进行维护和修改。这样产生的需求规约非常容易出错。一般来说，也难以保证其完整性和一致性，许多问题不清楚，阅读也非常困难，鲜有修改。采用此方法，想为不同的用户建立多个版本的需求规约太过昂贵，因此只能使用一个版本，即使是面对不同的使用者。这样的后果是：对某些使用者（如管理人员），该文档太细。而对另外的使用者（如独立测试人员），该文档又太粗。需求规约文档一般也没有包含关于需求的元数据（如调度信息、对开发者的任务分派、状态等等）。凡此种种，使得需求管理非常困难。

尽管这些问题早就存在，但需求工程师除了忍受外别无它途，因为其它的替代方法要么不为人所知、要么不切合实际。随着时间的推移，系统开发的发展趋势使得过去的方法不再可行，必须要采用新的方法完成需求工程的目标。该目标主旨在于建立的需求规约应当是正确、完整、内外一致、反映当前情况以及得到用户认可的。

2 影响需求规约的趋势

现代开发周期

也许影响需求规约任务最主要的趋势是从传统的瀑布开发周期到现代迭代、增量、并行和时间窗开发周期的转换。传统的瀑布式开发方法，需求的主要部分在需求阶段结束，架构、设计、实现、集成、测试开始之前就基本完成并进入冻结状态。这种方法难以取得成功的原因在于它基于诸如在周期的初期阶段需求就能够稳定和完全获取的假设。而实际上这两个假设基本上是不正确的。在现代开发周期介入后，需求工程取得了巨大的进步。目前，行之有效的开发周期具有下列特征：

- **迭代式的需求工程**：当认识到重要的制品（如需求规约）是尝试性的，并且由于人为错误和忽略在最初存在缺陷是不可避免的时候，人们开始使用迭代开发方法。这些重要的制品必须采用迭代的方法以发现和修改错误。部分开发过程（如需求捕获、分析及规约任务）是在已有制品（如需求、需求模型、需求规约等）上重复以改进它们。

- **增量式的需求工程**：当认识到许多制品由于太庞大和复杂而不可能通过一次重拳出击的方法能够完成的时候，人们采用了增量式的开发方法。比较复杂的应用，毫不夸张地说，可能包含相当数量的需求，因此，需要大量的时间去获取、分析和定义。由此，需求捕获、分析和规约任务是典型的需要增量完成的工作，在开发周期的时段内，新的需求按照天或周的时间段完成并增加到基本的需求中去。这就意味着，增量开发就是在开发过程的一些时段对某些工作反复进行以将新的工作成果增加到已经存在的制品中去的方法。

- **并行化的需求工程**：当认识到如果一个应用在任何合理的时间框架内完成，其包含的多个活动和任务需要并行进行时，需要采用并行化的开发周期。即使需求工程的各种活动尚未完全完成，也可以进行架构、设计、实现（如原型）和测试（如测试规划、测试用例开发）等工作。同理，我们并行地进行诸如需求捕获、分析、规约和管理等需求工程任务。这样带来的好处是可以使更多的项目参与者更早地开始承担任务，由此提高生产率。对需求、架构、设计、实现和测试的并行化开发增加了需求的迭代。

- **分段式的需求工程**：当任务或制品被细致地规划安排以便能够按照特定的时间期限完成时，我们称之为分段式开发过程（或者称为基于时间窗的开发过程）。瀑布式开发过程中传统的里程碑是建立在某个阶段主要任务基本完成的的基础上，而一个迭代的、增量式的、并行化的、基于时间窗的开发过程倾向于在一个新阶段中基于多个前期安排工作开展以建立一个新的里程碑。

迭代的、增量式的、并行化的、基于时间窗的开发周期对需求规约有较大的影响。这样的开发周期注意到了需求和需求模型不断发生变化的问题，并由此引起需求规约不断发生变化。这也是保持需求规约，特别是传统的手工建立需求规约的情况下一致性和最新性的困难。

具有不同需要的项目参与者

当需求规约由手工建立纸质文档时，需要花费相当大的代价。使得保持一个完整、最新的需求规约非常困难。就更不用说为不同的使用者建立多个不同的需求规约版本了。因此，每个项目参与者，无论其在项目中承担何种责任，都阅读和使用同一份需求规约。这样产生的后果是对管理者来说，这样的需求规约过细过多，而对独立测试组人员来说，这样的需求规约又太简单，难以据此建立测试用例。

应该认识到需求规约面对的是众多的项目参与者，不同的项目参与者有不同的需要，需要不同范围的、不同格式和粒度的、不同抽象层次、不同元数据的需求规约。例如，下面列举的不同项目参加者对需求有不同的需要：

- **执行经理**。通常他们对需求规约的使用主要是将其作为制定执行决策的基础。因此，需要的是简短、准确的总结性文档，而且能够被非技术读者理解。
- **项目管理者**。使用需求规约管理项目范围和评估计划进展情况以及资源使用情况。他们需要的需求规约比执行经理需要的更详细一些，但仍然在较高的抽象层次上，以便能够站在较高的层次上掌握需求，使得需求能够按照计划无一遗漏地完成。
- **主题专家**。领域专家需要较为深入地分析和定义需求的各种资源（业务对象模型、业务过程模型）。他们需要的需求规约更加详细，焦点更加集中于主题领域。
- **架构师**。架构师需要快速区分显著的需求。他们需要比管理者更加详细的结构方面的需求及参考。但不能够过分地陷入没有结构特征的底层需求中。

- **设计实现人员**。实际的开发者需要比架构师更多的完整和细节方面的需求。他们还需要组织化的需求以方便他们对具体部件的开发。当他们所关心的需求发生改变时，应该得到通知，可以通过类似“分布与订阅”功能获得需求变化的信息。应该注意到这些通知应该是细粒度的，以便他们能够只获得与其有关的需求发生改变的情况，而不是将所有需求的改变都通知他们。

- **测试人员**。与设计与实现人员一样，测试者也需要完整和详细的需求，因为他们要为测试用例建立完整的测试套件。例如，如果项目管理者和架构师获得的是用例信息，设计和实现人员得到基本用例，测试人员需要详细的包括前置条件和后置条件的用例路径信息。

以上的描述让我们清楚地看到，一种需求规约并不能够适应所有的项目参与者。采用一个需求规约通常会导致混淆和争议。例如，项目管理者可能会要求将一些细节信息去掉，因为这些信息可能对需求评价没有什么意义，而这些信息对开发和测试人员来说是至关重要的。

不断增加的应用范围和复杂性

自上世纪末以来，典型的应用系统一直向庞大及复杂发展。应用由单一型发展到客户/服务器结构，进而发展到多层结构。单一的应用让位于高度集成及具有互操作能力的应用，例如企业应用集成（EAI）。软件嵌入几乎每一个可想象的硬件中（如电话、电视、家用电器、汽车、ATM 等），可以说如果没有今天成百上千的计算机替我们工作，人们之间的交互要困难许多。应用也以超乎寻常的方式发展，例如，简单的信息 WEB 网站被电子商务以及电子市场 WEB 网站所替代。

由于应用变得更加庞大、更加复杂、对安全要求更高、更加具有一般性，其需求在涉及的范围、复杂程度和类型上也在不断提高。需求工程师不能仅仅只关心功能性需求，也应该将关注点放在数据需求、品质需求、应用程序接口（API）需求，以及各类架构、设计、实现和测试、业务规则、有关的法规等方面。实际上，对质量的迫切需求如操作可用性、性能、互操作能力、可扩展性以及安全性对于架构、开销和调度的影响比功能需求大。

大型复杂的应用需要更加复杂的需求规约文档。这样的需求规约文档的确变得越来越庞大、难以理解、审视和使用。对于这样的大型文档，难以形成一个单一的文档结构或抽象层次来进行刻画，特别是在考虑不同使用者的情况下。

应用于需求工程任务的更好方法和标记

在 70 年代我开始进行需求工作时，需求工程是一件很普通的工作。多数需求不适合于进行分析和分类。如果能够被允许使用结构化分析方法做一些自顶向下的功能分解或是数据流分析，对于多数需求工程师来说会感到非常幸运。

当然现在情况大不相同。目前我们已经认识到许多有用的需求任务例如业务分析、应用前景、需求捕获、需求分析、需求规约以及需求管理等等。也拥有众多的需求分析方法，多数方法（例如用例分析）比以往的技术有显著的改进。这些需求方法使用多种类型和抽象层次为不同的使用者提供不同的需求视图。拥有更多更好的需求模型，这些模型包括包含不同类型需求的比较完整的分类和层次。尽管不太完美，我们毕竟有比较好的建模语言（例如 UML），较好的规范的、半规范的需求方法，对需求规约也有比较好的标准内容。

以上种种对需求规约的影响主要在对定义数据类型、内容和规范，以及如何组织需求规约方面。我们应该采用多种形式定义需求，包括自然语言文本需求、采用图形建模语言和决策表的需求模型，采用定义描述语言规范地定义需求等等。

更好的需求工具支持

鉴于需求工程师的工作越来越复杂和具有挑战性。更加强有力和用户友好的需求工具的出现抵消了先前提到的负面影响，甚至首次采用了一些新的、有效的方法。目前我们有集成的需求工具用来支持更多的需求工程任务如需求捕获、分析、定义及管理。这些工具与其它工具的有效集成可以支持与需求工程活动有关的其它相关任务，例如范围管理、版本控制、配置控制（配置管理）、品质保证及质量控制。这些工具目前还能够支持分布式多用户共同工作。

更为重要的是，这些工具向基于知识库（通常建立一个对象数据库或扩展的关系数据库）的方向发展，用以支持良好的粒度级别。个别的需求可以以增量方式加入到需求知识库中，需求可以被迭代，需求元数据（例如优先级、可跟踪信息、组件及小组的任务分派）可以与有关的需求存储在一起，当有关的需求修改时，开发者能够得到该信息。选择适当的标准和模板，需求规约的各种类型、内容和细节层次可以从需求知识库中自动建立。

观察该发展趋势的一个有效的方法是看这些方法在处理大型、复杂的需求集合时的演化过程：

1. 字处理工具，如 MS WORD；
2. 报表，如 MS EXCEL；
3. 内嵌报表的字处理工具；
4. 具有特别报告产生能力的数据库；
5. 需求管理工具；
6. 支持更多需求工程任务的需求工具；
7. 需求工具适当地集成到集成开发环境（IDE）中。

遗憾的是，当前需求工具还不能够支持所有行为和任务，并且不同工具所能支持的范围也大不相同。我将在本文中列出一些建议用于评估需求工具

3 改进需求规约任务

针对先前提及的影响需求工程的困难和趋势，我们应该做些什么呢？本文将提出以下的建议，用以改进需求规约任务中需求规约的建立。

需求知识库

将需求保存在需求知识库中而不是纸质文档中。保持需求知识库的细粒度以保证个别的需求能够保存于其中，并能够进行迭代、核准、配置管理、分布、管理以及跟踪等。将需求视为独立的对象并将完整的需求规约以二进制大对象（BLOB）形式保存。确保需求知识库存储于需求有关的所有信息，例如独立需求、需求元数据（需求对象的数据），需求模型（聚集对象），以及有关的图表。需求知识库应该基于对象数据库、XML 数据库或者可扩展的关系数据库。例如需求工具 CaliberRM、DOORS 都是基于对象数据库。应该认识到该建议对于其它建议是基本的和必须的。

规约自动建立

确保从需求知识库中自动建立需求规约。这将超越仅仅是简单地使用公认的（如 IEEE830-1998）、工业的（如 OPEN、RUP）、和企业内部的模板来建立一个或多个需求规约。也能够为需求管理和其它目的产生任意的需求报告。还能够产生可发布的完整的需求规约，不需要在建立需求规约中部分工作采用手工过程获得。保证建立的需求规约与知识库中正式的需求一致。采用此种方法将使需求规约或报告的建立不需要大量的纸质文档，有助于解决迭代开发过程中由于需求不断变化所带来的需求管理困难的问题。

以上两条建议都认识到将模型和视图加以区别的的重要性，这样做有助于产生图形用户接口（GUI）。也认识到将同一模型中需求（模型）与潜在的多视图（规约）分开的重要性。因为需求规约的发布应该包含按照时间点的、特定使用者甚至个性化的信息，需求工程领域也提倡通过信息网站考虑内容管理活动。

应用于不同用户的规约

一旦将需求存储在组织良好的需求知识库中并具有从需求知识库中自动产生需求规约的能力，就可以为不同的用户建立需求规约。认识到对同一个需求模型的不同方面有不同的需求这一问题是非常重要的。这种情况包括对不同类型需求视图的需求规约和报告（例如功能需求或是接口需求），需求细节不同层次（执行经理的需求与测试者的详细需求），预定义规约与随机报告，基于元数据的规约（基于需求的规约或报告，包含日期、状态、拥有者、最新修改日期）等等情况。

需求工具

除了拥有一个能够自动产生需求规约和报告的知识库外，还需要一个或多个基于该知识库的需求工具。这类工具应该有如下的关键特性：

- 用户接口：输入需求以及它们所涉及的元数据的最好的办法是使用对用户友好的图形用户接口，通过该接口可以方便容易地输入和维护具体的需求及元数据，包含文字、图示和表的需求模型。应该注意到这种方法与要求需求组首先使用类似 MS WORD 等字处理系统建立简单的需求规约，然后将需求规约输入到数据库中是有区别的（虽然说这样做至少聊胜于无，有一个可重用的东西）。用户接口应该理解基本的需求模型，这些需求模型包括需求类型、公共的及特定的元数据、不同类型的模型、不同类型的图示等等。用户接口不仅应该支持输入及维护需求，也应该支持建立需求规约和需求报告，例如模板建立、查询和实际建立。

- 支持需求工程：需求工具要能够支持需求工程所涉及的包括业务分析、开销/效益分析、应用前景分析、需求捕获、分析、定义和管理等所有任务。应该支持多种需求分析方法以方便定义多种类型的模型（例如，用例、决策表、状态模型、关联模型等）。还应该支持包括所有需求类型（功能需求、数据需求、质量需求、接口需求、各种约束等）的完整需求模型。需求管理工具还应该具有对架构、设计、实现和测试以及迭代等工作进行跟踪的能力。

- 支持有关的活动：需求工具应该支持其它与需求工程相关的活动。这些活动包括范围控制、配置管理、质量工程等。实现管理、配置管理、质量工程、建模、前向或后向测试工具之间进行互操作的能力。因此，需求工具不应该是孤立的，而应该是集成开发环境中关键的组件。

- 支持成组开发：需求工程最好是由交叉功能需求组来完成，该需求组能够提供获取所有需求并以迭代方式及时完成工作所需要的经验。尽管在联合需求工程过程中由技术专家形成需求模型以及使用建模工具获取最初需求非常重要，但所有需求组成员应该能够同时工作于同一需求。同时，其它成员在学习、改进和理解需求时可以同时进行存取。

- 支持安全性：多数应用的需求包含私有信息、商业机密、甚至国家机密。任何需求管理工具应支持需求的安全性管理，包括采用确认、认证、权限等完成针对不同用户的特定任务的安全检查。也包括需求的私有化、完整性等的保证。

- 其它质量因素：与其它应用类似，用于需求定义的需求工具在安全性和互操作能力之外还包含其它质量因素。因此，在考虑选用这类工具时，应该考虑诸如完整性、国际化、性能、可扩展性、可用性和用户友好等特性。

- 支持分布式开发：目前，应用由分布在多个地理位置的多个小组和机构建立的情况非常普遍。需求工具应支持由分布用户完成定义的需求工程。

- 需求重用：需求工具应该保证将存在的需求方便地加入知识库中，以便每项工作都不需要从头开始。由于这些需求可能是使用传统方法建立的，需求工具应能够分析以前的需求规约，识别潜在的需求并以某种方式合并，使他们能够方便地被审视以确定接受他们，或是修改后接收或放弃。该建议的主旨是将所有的需求作为独立的对象以较高的细节程度存储在知识库中。

- 不仅仅是一个 CASE 工具：合适的需求工具不应该仅仅是独立的建模工具或需求知识库。它应该成为集成开发环境（IDE）中的一部分，而不仅仅是简单的传统意义上的计算机辅助软件工程（CASE）工具。

显然本文的主题是现代需求规约，目前比较清楚的是，在讨论需求规约时，应该涉及需求工程的其它任务。仅仅讨论使用基于知识库的需求工具来支持自动建立特定角色的需求规约而不解决此类工具的其它问题也是没有意义的。

4 结论

本文提出了传统的基于纸质文档的需求规约方法存在的几个问题以及软件工程发展趋势如何加剧了这些问题。正如下面图例所描述的，解决这些问题的需求规约方法是采用基于细粒度需求知识库的现代需求工具。使用基于这些建议的适当工具，可以自动化地、方便地、廉价地建立各类高质量的需求规约，这些需求规约能够满足不同使用者的特定需要。

致谢

感谢 Ivars Auzins, Keith Collyer, David Gelperin 和 Hiam Kilov 的审阅并提供了很有用的建议。

征稿

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

UMLChina 公开课

“UML 应用实作细节”

（2003 年 6 月 21-22 日，广州）

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，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，非常适合中小团队。

详情请见>>

安全模型的一种模式语言

Eduardo B. Fernandez 等 著，Happy 译

查看评论

摘要

安全问题是 Internet 中比较重要的问题，对于设计一个包含安全模块的新系统，安全性也是必需的。模式是一个很好的工具，可以帮助设计者构建安全的系统。我们在这里将讨论三种模式，它们都是最通用的安全模型：授权（Authorization）、基于角色的访问控制（Role-Based Access Control）和多级安全（Multilevel Security）。这些模式可以应用于系统的所有级别，我们将展示它们在文件授权模式定义中的使用。

1、简介

安全是任何计算系统非常重要的方面，并且在机构把他们的数据库向 Internet 开放后，这已经成为日益严重的问题。大多现在正使用的 Web 系统在设计时并没有过多考虑安全性，即使打上补丁，也还是不能有效抵御攻击。所以，开发系统过程中，在设计的每个阶段都考虑到安全性是很重要的，不能够仅仅满足功能性需求，还要满足安全性需求[Fer00a]。为做到这一步，我们要从表示机构安全策略的高层模型开始。现在，大多数系统使用三种模型：访问矩阵、基于角色的访问控制（RBAC）模型和多级模型。

安全性包含在系统所有的架构层中[Fer99]，因此，分层架构(Layer)模式[Bus96]是本模式语言的入手点。这个模式提供了一个结构，在这个结构中，我们可以为所有层定义模式，共同完成一个安全的系统。它的主要概念是将系统分解成若干抽象的层次级别，高层使用低层的服务。这里给出一种全面看待事物的方式，并且描述了每层所需的机制。我们在以前曾讨论过 [Fer99]：为什么所有这些层必须协调起来，确保其安全性。图 1 展示了我们所考虑的特定分层，这个图展示了每层的一些参与者和它们相应的交叉层。

由于问题的复杂性，我们需要一系列模式语言，每层一种。本文我们从抽象模型的一种模式语言开始，它包括三种基本的模型，上边提到了，这些模型为应用程序架构最上层定义了安全性约束，但在底层实施。这些模型已经被安全性社区广泛研究[Pfl97,Sum97]，我们不想再增加新模型或扩展现有模型。我们的目的是将这些已被接受的模型表示成面向对象的模式，以能够作为构建安全系统的指导，同时，也展示这些模式如何应用于所有系统级别。

首先，我们提出描述资源访问控制的授权模式，接着在第 3 节描述 RBAC 模式，它是授权模式的一种扩展。第 4 节是多级安全模型。在这之后，我们将讨论这些模式如何应用于底层，并展示一种文件访问控制的模式，作为授权模式特殊形式的例子。最后，我们将讨论这些模式的一些共同方面。

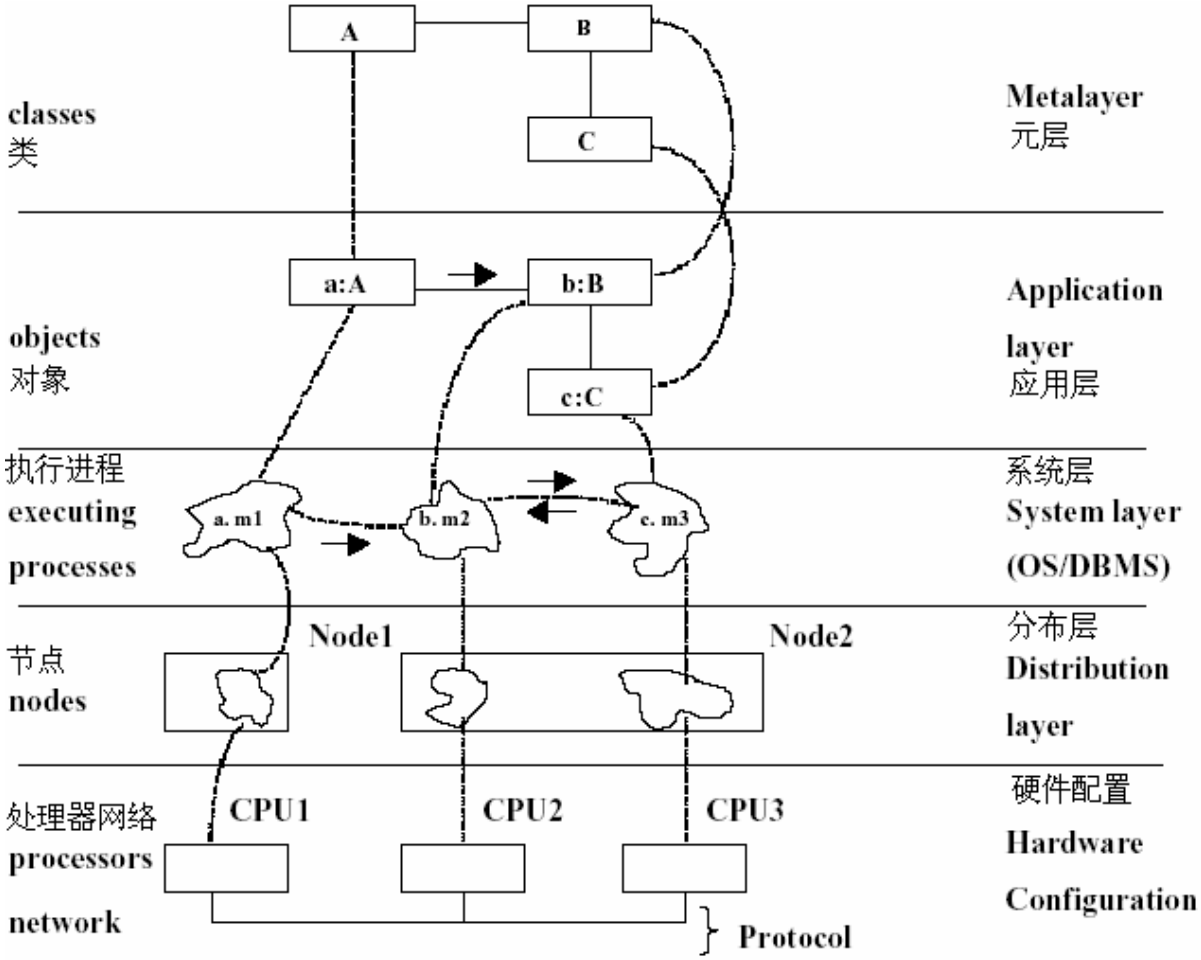


图 1 分层模式的一个实例

2 授权模式

上下文

任何存在主动实体请求某种受控资源的计算环境。

问题

如何描述主动计算实体（主体）对被动资源（保护对象）可允许的访问类型（授权）。

先决条件

- 授权结构必须独立于资源类型，例如对用户访问概念实体、程序访问操作系统资源，要以统一的方式来描述。
- 根据特定条件，谓词或保护措施可能要限定授权的使用。
- 有些授权可以被它的持有者委托给其他主体。

解决方案

用类和类的关联来表示授权规则的元素。类 **Subject** 描述一个主动实体，类 **ProtectionObject** 描述一个被请求的资源，一个授权规则由这两个类的关联来定义。一个关联类 **Right**，包含访问许可的类型（读，写等），包含一个谓词，对授权持有者来说必须为 true，还包含一个复制标志可以为 true 或 false，表明这个权限是否可以转让。另外还有一个操作 check_rights 用来让主体或对象检查请求的合法性。图 2 展示了涉及到的元素。

图 3 是一个时序图，展示了一个请求的验证。ActiveSubject 是一个角色（Actor），例如一个请求某种资源的执行进程，这个请求被一个 Reference 监视器解释。

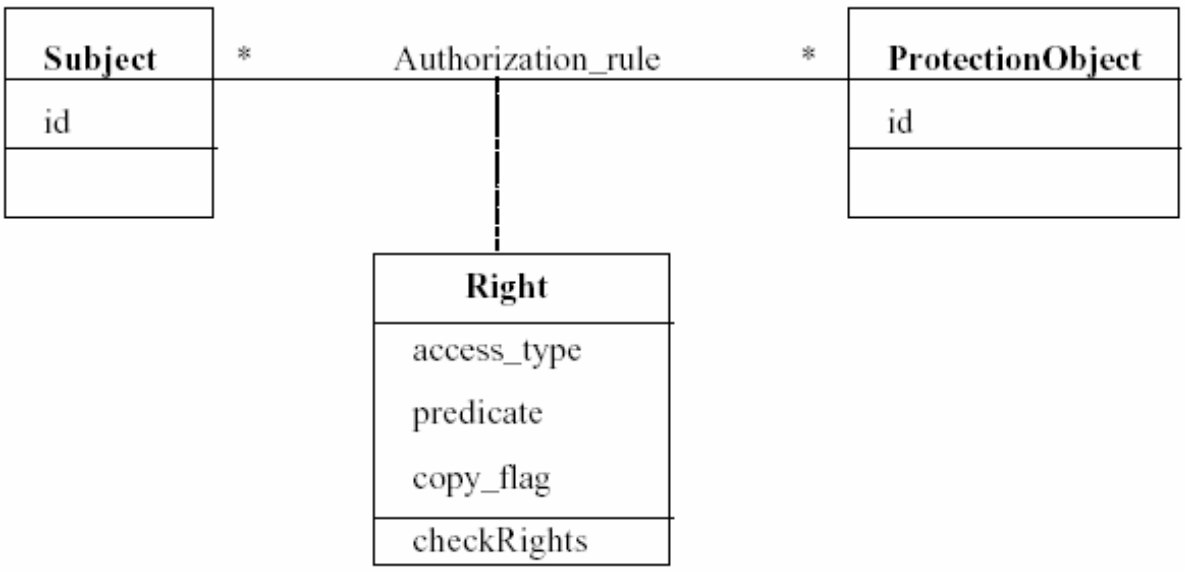


图 2 授权模式

结论

这个模式的优点包括：

- 可以应用于所有类型的资源。主体可以是执行进程、用户、角色、用户组等。保护对象可以是事务、内存区域、IO 设备、文件或其他 OS 资源。访问类型可以是读、写、执行或对象其他高层的某种方法。
- 对于任何可以限定规则应用的条件，谓词是一种通用表示。
- 规则中的复制标志控制权限的转让。不过有些系统不允许转让权限，那么这儿的复制标志将总是为 false。
- 有些系统把管理员授权和一般用户授权分开，以满足更高的安全性（职责分离原则）[Woo79]。
- 请求无需在规则中指定确切的对象，它可以由一个已有的保护对象隐式指定[Fer75]。同样，主体和访问类型也可以隐式，这样当进行一些附加处理时，能提高时间花费的灵活性（引出所需的特定规则）。

已知应用

此模型对应访问矩阵的成分，访问矩阵是一个基本的安全模型[Sum97]。它的首个面向对象形式出现在[Fer93]。后来，它在其他若干论文和产品中也出现过[Ess97,Kod01]。它是多数商业产品访问控制系统的基础，例如 Unix、Windows、Oracle 和很多其他产品。

相关模式

在后面提出的 RBAC 模式就是这个模式的一个特例。在第 5 节，我们将展示另一个特例——操作系统文件访问控制模式。

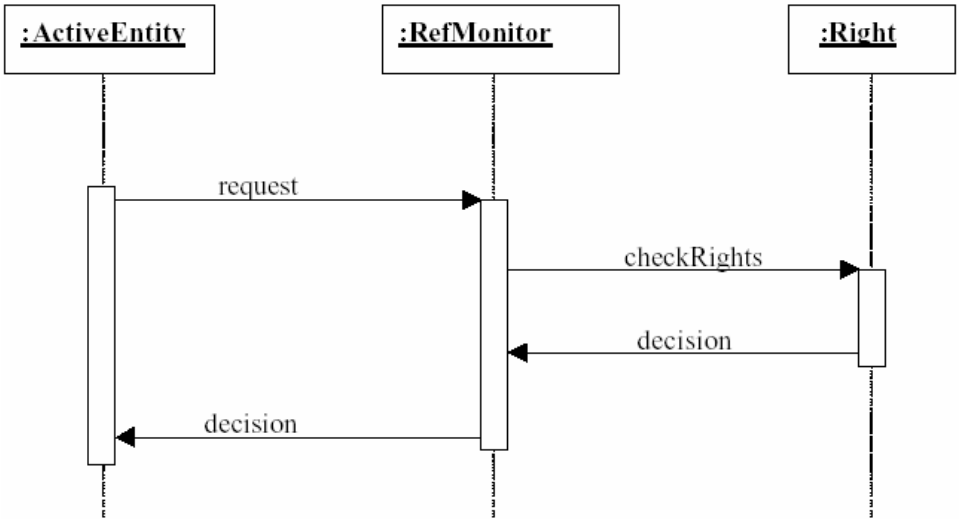


图 3 验证请求

3 基于角色的访问控制（RBAC）模式

上下文

多数机构具有很多不同的工作岗位，需要不同的技能和责任，为了安全考虑，用户应该根据它们的工作岗位获得权限。这对应一种基础安全策略，need-to-know 原则的应用[Sum97]。工作岗位可以视为人们在执行职责时扮演的角色，尤其在基于 web 的系统中，有很多不同的用户：公司职员、客户、合作伙伴、搜索引擎等等。

问题

在一个机构中，如何根据角色为用户分配权限。

先决条件

- 机构中的人根据他们职能不同，对信息的访问有不同的需求。
- 我们想协助机构基于 need-to-know 策略，为它的成员定义精确的访问权限。
- 为单个用户赋予权限需要存储很多授权规则，并且管理员很难跟踪这些规则。
- 用户也许拥有不止一个角色，并且我们可能想实施诸如职责分离之类的策略，那样用户不能在同一个会话中扮演两个角色。
- 一个角色可以分配给单个用户或用户组。

解决方案

延伸上一个模式的概念，把角色当作主体，图 4 展示了基于角色访问控制的一个基本模型。类 User 和 Role 分别表示已注册的用户和预定义的角色，用户被赋予角色，按照角色的职能，赋予用户特定权限。关联类 Right 定义一个角色用户被授权访问保护对象的权限类型。实际上，综合 Role、ProtectionObject 和 Right，就是一个授权模式的实例。同样，谓词表示依内容而定的限定，用来选择特定对象，copy_flag 属性表示一个布尔值，true 或 false。

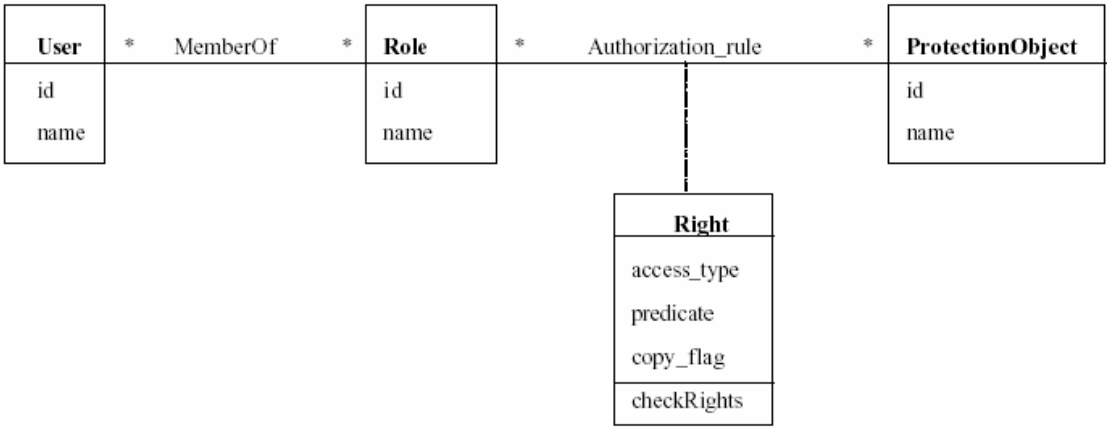


图 4 基本 RBAC 模式

图 5 中的模型考虑到复合角色（Composite 模式的应用），以及管理权限和其他权限的分离（职责分离策略的应用），这个模型同时也包含了一个约束条件来实施职责的分离。管理员具有为组和用户赋予角色的权限，它是一个特殊的用户，可以为用户赋予角色和为角色分配权限。安全管理的权限通常包括：

- 角色授权规则的定义。
- 用户组的创建、删除。
- 用户的角色赋予。

图 5 还包括了会话（Session）的概念，对应于执行时，用户排他方式地扮演某个角色。最后，Group 类描述一组可以被赋予同样角色的用户。

结论

这个模式的优点如下：

- 允许管理员降低安全性的复杂度，用户要比角色多得多。
- 机构关于工作岗位的策略可以直接反映为角色的定义和用户的角色赋予。
- 为得到更大的灵活性和规则的简约性，角色可以进行结构化。
- 为了功能的灵活性，用户可以同时激活多个会话，有些任务可能需要多个视图或不同类型的行为。
- 我们可以增加 UML 约束，以表明有些角色不能用于同一个会话或被赋予给同一个用户（职责分离）。

- 用户组可以作为角色成员，这样可以大大降低授权规则和角色分配的数目。

可能的缺点包括：

- 附加的概念复杂度（新的角色概念，以及多重角色赋予等概念）。

另外还有一些可能的角色结构[Fer94]，在特定环境中比较有用。同时，使用角色来扩展第 5 节的多级模型也是可以的。

已知应用

我们的模式使用面向对象方式表示在[San96]中描述的模型，那个模型已经成为多数研究论文和实现 RBAC 概念的基本原理[FBK99]。RBAC 在很多商业系统中都有应用，包括 Sun J2EE[Jaw00]，Microsoft 的 Windows 2000，IBM 的 WebSphere 和 Oracle 等等。Java JDK1.2 的基本安全模块已经表示可以支持很多种 RBAC 策略[Giu99]。

相关模式

在[Fer93]和[Yod97]中出现一个更简单的版本（相对于图 3），在[Kod01]中，有一个软件实现的模式语言（没有考虑合成角色、组和会话）。图 5 的模式包含了授权模式和合成模式，其他相关模式还有角色（Role）模式[Bau00]和抽象会话（Abstract Session）模式[Pry00]。

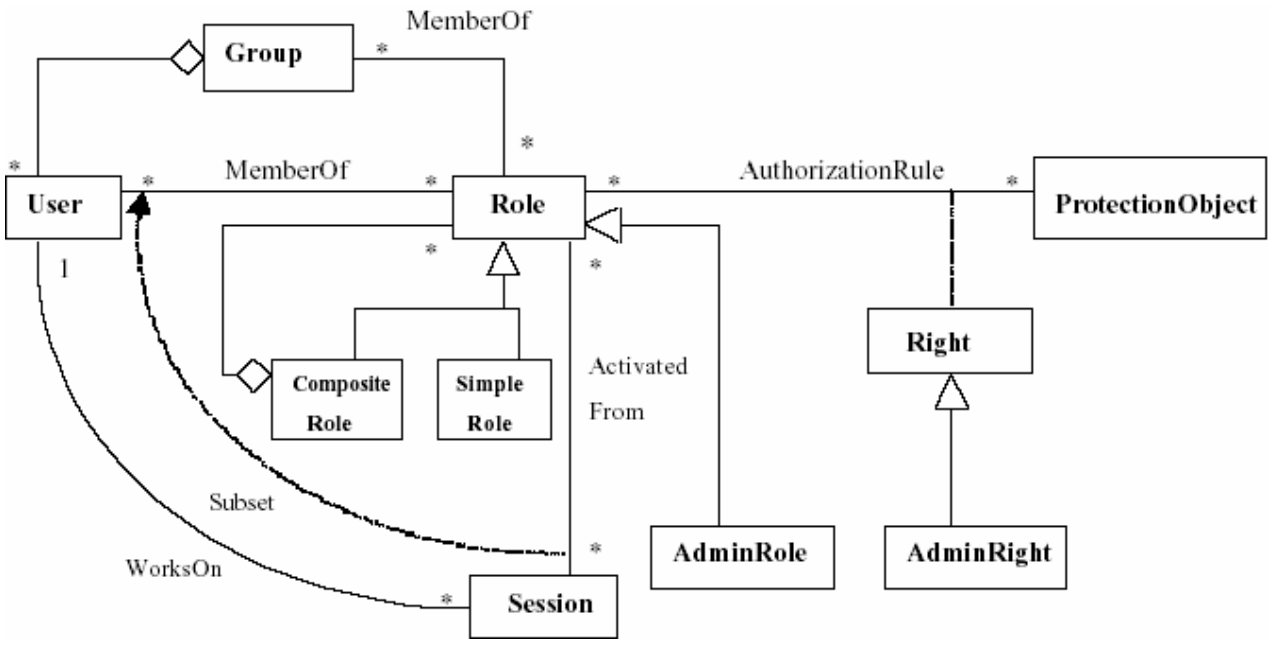


图 5 RBAC 模式

4 多级安全模式

上下文

在某些环境中，数据和文档具有不同敏感级别，例如秘密级、机密级。用户可以凭借许可证访问文档。

问题

如何在安全分级的环境中决定访问方式。

先决条件

- 模型应该根据数据的敏感度来保护它的机密性和完整性。
- 模型应该能够用于所有的架构层。
- 能够有不同的规则组来决定访问方式。
- 必须有一种便利的方式为用户和数据分配分类级别。

解决方案

在这个模型中，用户和数据被赋予分类或许可证。分类包括级别（高度机密、机密等）和分区（工程部、市场部等）。根据 Bell-Lapadula 模型定义的规则实现用户对数据的机密性访问，通过 Biba 模型[Sum97]定义的规则来实现完整性。图 6 展示了这个模型的基本结构。

但是，基本模型不允许级别分配，所以需要一组特定授信的进程来分配级别和修改分类，也就是完成所有管理的功能。

结论

优点包括：

- 用户和数据的分类比较简单，可以根据机构策略进行。
- 在可靠假设前提下被证明是安全的[Sum97]。

可能的缺点包括：

- 通过为数据标上标签以表明它们的分类，这样确保了安全性。但如果没有这么做，将反而降低整个安全度。
- 我们需要附加的受信程序为用户和数据分配级别。
- 数据需要能够结构化成层次化的且具有敏感分级的，同时用户也要能结构化成许可证形式，这通常难以做到，或者在商业环境中是不可能的。

已知应用

这个模型已经用于若干军事项目和一些商业产品中，包括 DBMS（Informix）和操作系统（Pitbull[Arg]和 HP 的 Virtual Vault[HP]）。

相关模式

角色的概念也可以用在此处，角色分类可以代替用户分类。

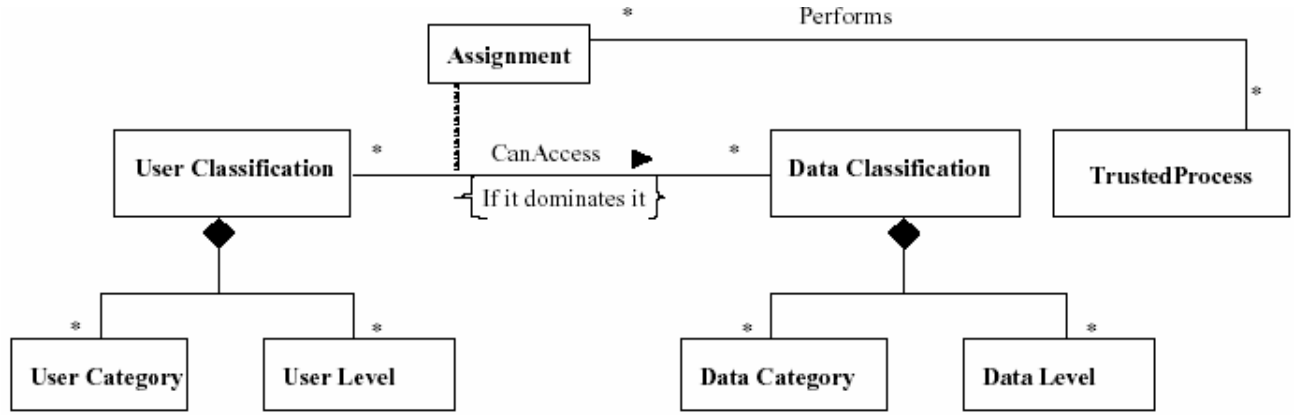


图 6 多级安全模式的类模型

5 底层

上面这些模式定义了抽象的模型，可以应用于系统的所有层。在每一层可以使用该层具体的实体来表示抽象模型的概念，以形成更加具体的模式。为得到一个安全的系统，有必要为每层定义模式语言，这将留作以后的工作，这里提供一个示例。

第 3 节的授权模式可以应用于操作系统中用户从工作站访问文件的情况，这是一个图 1 中系统层的模式。

5.1 文件授权模式

上下文

操作系统的用户需要定义文件，这些文件可以从被不同授权的工作站访问，并且对文件的访问只限于授权用户。

先决条件

对于这种情况，具有如下先决条件

- 可能有不同类型的主体，如用户、角色和组。
- 主体可以被授权访问文件、目录和工作站。
- 一个主体在每个已授权工作站上都有一个 home 目录，但是同样的 home 目录可以在多个工作站或主体之间共享。
- 用户可以形成组来访问。
- 有些系统使用角色代替用户作为主体，或两者都作为主体。
- 操作系统中文件系统具有不同的实现方法。

解决方案

把授权模式特殊化，以文件代替对象，以访问许可代替权限，如图 7。它定义了文件访问，工作站方式也是类似地应用授权模式定义。文件和目录是递归结构的，结果是一个语义分析模式（Semantic Analysis pattern SAP），组合了两个版本的授权模式成为一个合成模式。一个 SAP 是对应于一组基本用例的分析模式[Fer00]。

结论

这个模式可能的优点：

- 它可以使用于多种主体，包括用户、角色和组。
- 被访问的对象可以是单个文件、目录，或者是递归结构的目录和文件。

- 隐式的授权是可能的，例如，访问一个目录可以隐式指定以类似的方式访问该目录下所有的文件。

一些缺点如下：

- 该模式的实现不必依据访问矩阵模型。例如，Unix 使用一种伪访问矩阵，使用 need-to-know 策略并不恰当。但可以为这个模式加上约束，强迫模式所有的示例都遵从一个访问矩阵模型。

其他方面包括：

- 有些系统对工作站不能使用授权。
- 多数操作系统使用读、写、执行作为访问类型，更高层类型的访问也是有可能的。
- 多数操作系统有所有者的概念，它是一个特殊的用户，对它创建的文件拥有所有的权限。
- 有些系统中，文件映射到虚拟内存地址空间，这个模式同样适用于这样的情况。

已知应用

这个模式出现在 Unix、Windows、Linux 和多数现代操作系统中。

相关模式

这是授权模式的一个特例，如果使用角色，就还和基于角色访问控制模式有关，文件系统使用了一个合成模式。

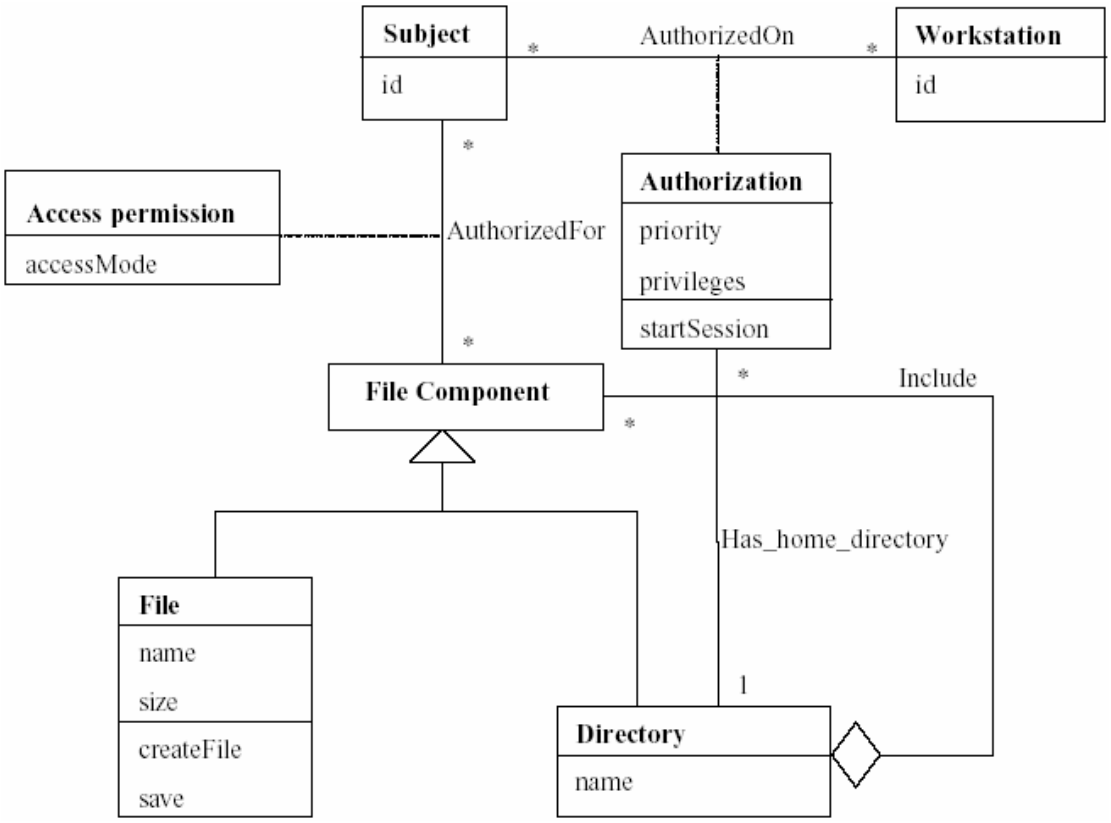


图 7 文件授权的一种模式

6 模式的一般讨论

以上这些模式的具体实现要看它应用在架构中的哪一层，其中一个重要的问题是如何以较低的负荷来进行访问控制。授权模式可以通过访问控制列表（ACL）来实现，多数操作系统就采用这种方法。另一种方式是使用 Capablities，硬件和操作系统层控制资源即采用这种方式。近来有些文章描述应用控制对类的访问[Fra99]，使用元类和反射机制来实现这些模型是另一种有趣的方式[Wel99]。对资源的请求必须被 ACL 或 capabilities 中的信息解释和验证，这需要一个特殊的程序（另一种模式），叫做 Reference Monitor[Sum97]，一个可能的实现就是 POSA2 中的 Interception 模式[Sch00]。

已经提出的某些架构层的模式：

- Yoder 和 Barcalow[Yod97]描述了单访问点（Single Access Point，一种通用安全系统设计模式），检查点（Check Point，一种 Reference Monitor）、角色（见第三节）以及其他等。
- Fernandez 讨论了元数据和模式之间的关系[Fer00a]。第三节的 RBAC 模型最先出现在此。

- Cryptographic 模式描述在[Bar00]。
- 关于安全模式，包括一些网络安全模式的一般性讨论位于[Sch01]。
- 一个访问控制、过滤分布对象的框架，组合了若干模式，位于[Hay00]。
- Java 的若干安全模式位于[Jaw00]。

对于在每一层增加更多的模式，收集和统一这些模式的工作还有待继续进行。

还有一些基于其他策略或策略组合的安全模型。其中重要的两个模型是 Clark-Wilson 模型和 Chinese Wall 模型[Sum97]。Clark-Wilson 模型着重强调完整性，并定义两个基本原则：

- 组织良好的事务——确保对数据的修改保持在一致的状态
- 职责分离——改变必须至少被两方认可。

Chinese Wall 模型与其说是模型，不如说是一种策略。信息被分组成若干利益冲突的类，一个人在每个类中至多允许访问一组信息。

目前有 Clark-Wilson 模型的模式[Wel99]，但是没有 Chinese Wall 模型的模式。

参考文献

[Arg] Argus Systems Group, "Trusted OS security: Principles and practice", http://www.argus-systems.com/products/white_paper/pitbull

[Bau00] D.Baumer, D. Riehle, W. Siberski, and M. Wolf, "Role Object", Chapter 2 in *Pattern Languages of Program Design 4* (N. Harrison, B. Foote, and H. Rohnert, Eds.). Also in *Procs. of PLoP'97*, http://jerry.cs.uiuc.edu/~plop/plop97

[Bra00] A. Braga, C. Rubira, and R. Dahab, "Tropyc: A pattern language for cryptographic objectoriented software", Chapter 16 in *Pattern Languages of Program Design 4* (N. Harrison, B. Foote, and H. Rohnert, Eds.). Also in *Procs. of PLoP'98*, http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal., *Pattern- oriented software architecture*, Wiley 1996.

[Ess97] W. Essmayr, G. Pernul, and A.M. Tjoa, "Access controls by object-oriented concepts", *Proc. of 11th IFIP WG 11.3 Working Conf. on Database Security*, August 1997.

[FBK99] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn, "A Role-Based Access Control model and reference implementation within a corporate Intranet", *ACM Transactions on Information and System Security*, Vol. 2, No. 1, Feb. 1999, pages 34-64 .

[Fer75] E. B. Fernandez, R. C. Summers and T. Lang, ``Definition and evaluation of access rules in data management systems,'' *Proceedings 1st International Conference on Very Large Databases,* Boston, 1975, 268-285.

[Fer93] E.B.Fernandez, M.M.Larrondo-Petrie and E.Gudes, "A method-based authorization model for object-oriented databases", *Proc. of the OOPSLA 1993 Workshop on Security in Object-oriented Systems* , 70-79.

[Fer94] - E. B. Fernandez, J. Wu, and M. H. Fernandez, ``User group structures in object-oriented databases", *Proc. 8th Annual IFIP W.G.11.3 Working Conference on Database Security,* Bad Salzdetfurth, Germany, August 1994.

[Fer99] E.B.Fernandez, "Coordination of security levels for Internet architectures", *Procs. 10th Intl. Workshop on Database and Expert Systems Applications*, 1999, 837-841.

[Fer00] E.B. Fernandez and X. Yuan, "Semantic Analysis Patterns", *Procs. of the 19th Int. Conf. on Conceptual Modeling (ER2000),* 183-195.

[Fer00a] E.B. Fernandez, "Metadata and authorization Patterns", Report TR-CSE-00-16, Dept. of Computer Science and Eng., Florida Atlantic University, May 2000.

[Fra99] G. Frascadore, "Java application server security using capabilities", *Java Report*, March 1999, 31-42.

[Giu99] L. Giuri, "Role-Based Access Control on the web using Java", *Procs. of RBAC'99*, ACM 1999, 11-18.

[Hay00] V. Hays, M. Loutrel, and E.B.Fernandez, "The Object Filter and Access Control framework", *Procs. Pattern Languages of Programs (PLoP2000) Conference*, http://jerry.cs.uiuc.edu/~plop/plop2k

[HP] Hewlett Packard Corp., Virtual Vault, http://www.hp.com/security/products/virtualvault

[Jaw00] J. Jaworski and P.J. Perrone, *Java security handbook*, SAMS, Indianapolis, IN, 2000.

[Kod01] S. R. Kodituwakku, P. Bertok, and L. Zhao, "A pattern language for designing and implementing role-based access control", *Procs. KoalaPLoP 2001*.

[Pfl97] C.P. Pfleeger, *Security in computing*, (2nd Ed.), Prentice-Hall 1997.

[Pry00] N. Pryce, "Abstract session: An object structural pattern", Chapter 7 in *Pattern Languages of Program Design 4* (N. Harrison, B. Foote, and H. Rohnert, Eds.). Also in *Procs. of PLoP'97*, http://jerry.cs.uiuc.edu/~plop/plop97

[San96] R. Sandhu et al., "Role-Based Access Control models", *Computer* , vol. 29 , No2, February 1996, 38-47.

[Sch00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture, vol. 2 , Patterns for concurrent and networked objects,* J. Wiley, 2000.

[Sch01] M. Schumacher and U. Roedig, "Security engineering with patterns', submitted to PLoP 2001.

[Sum97] R. C. Summers, *Secure Computing: Threats and Safeguards*, McGraw-Hill, 1997.

[Wel99] I. Welch, "Reflective enforcement of the Clark-Wilson integrity model", *Procs. 2nd Workshop in Distributed Object Security*, OOPSLA'99, ACM, November 1999, 5-9.

[Woo79] C. Wood and E. B. Fernandez, ``Authorization in a decentralized database system," *Proceedings of the 5th International Conference on Very Large Databases*, 352-359, Rio de Janeiro, 1979.

[Yod97] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security". *Procs. PLOP'97*, http://jerry.cs.uiuc.edu/~plop/plop97 Also Chapter 15 in *Pattern Languages of Program Design*, vol. 4 (N. Harrison, B. Foote, and H. Rohnert, Eds.), Addison-Wesley, 2000.

减少耦合

Martin Fowler 著，李巍 译

查看评论

耦合（coupling）是设计质量的最初标志之一。它与内聚（cohesion）一起出现在最初的结构化设计中，就从来没有消失过。我在考虑软件的设计时总是能够想起它。

描述耦合的方式有许多种，但可以归结为：如果改变程序中的一个模块需要改变另一个模块，那么便存在着耦合。这两个模块很可能在一点上做着相似的事情，因此一个模块中的代码实际上是另一个模块中的重复代码（duplicate of code）。这是代码重复最主要和明显的一个罪证。重复总是意味着耦合，因为改变重复代码的一部分便意味着要改变另外的部分。而且重复代码也很难被发现，因为这两段代码之间的关系可能并不明显。

当一个模块中的代码使用其它模块的代码（通过调用一个函数或存取一些数据）时，同样会出现耦合。有一点要清楚，你不能像对待重复代码那样，总想着去避免耦合。你可以将一个程序分成多个模块，但是这些模块将需要以某种方式进行通信——否则，你只不过是有着多个程序而已。耦合是需要的，因为如果你禁止了模块之间的耦合，你就不得不要将所有的东西都放到一个大的模块中。那么，将会出现大量的耦合——只不过它们都藏在背后而已。

因此，我们需要对耦合进行控制，但如何进行呢？我们在任何地方都需要担心耦合吗，或者是否耦合在一些地方会比其它地方更为重要呢？哪些因素会使耦合变糟，而哪些则能够被容许呢？

如何看待依赖

我自己最关心的是位于顶层模块上的耦合。如果我们将一个系统分成一打（或更少的）大型模块，如何把这些模块耦合起来呢？我关注那些粗粒度（coarser-grained）的模块，因为在任何地方都去担心耦合会令人无所适从。最大的问题来自未经控制的上层耦合。我不担心耦合在一起的模块数目，但是我看重模块间依赖关系的样式。同时，我发现图（diagram）对我们很有帮助。

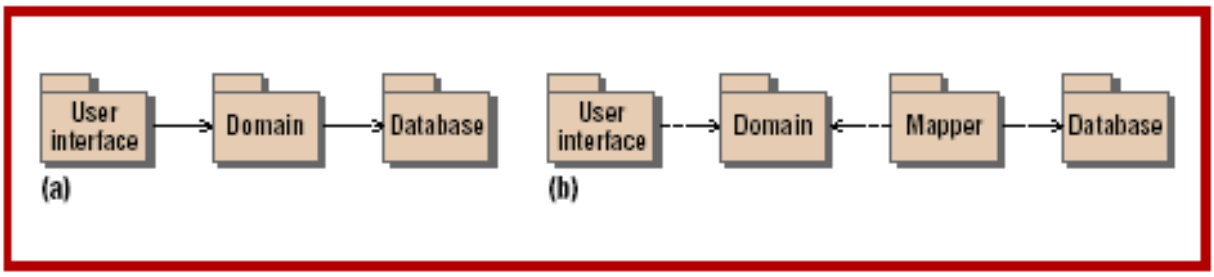


图 1.(a) 一个简单的包图，(b) 一个隔离领域和数据库的映射器包

关于术语*依赖（dependency）*，我会按照统一建模语言（UML）中的定义来使用。因此，如果任何 UI 模块中的代码引用了任何领域模型中的代码——通过调用一个函数，使用一些数据，或使用领域模块所定义的类型，那么 UI 模块便依赖于领域模块。如果有人改变了领域模型，UI 模型也将有可能需要相应做出改变。依赖是单向的（unidirectional）:UI 模块通常会依赖领域模块，反之则不是。如果领域模块也依赖 UI 模块，将会出现第二个依赖。

UML 依赖也是非传递（non-transitive）的。如果 UI 模块依赖领域模块，并且领域模块依赖数据库模块，我们将不能假定 UI 模块依赖数据库模块。倘若真是这样，我们需要将其描述成额外一个在 UI 和数据库模块之间的直接依赖。这种非传递性非常重要，因为它使我们看到领域模型能够将 UI 从数据库的变化中隔离出来。因此，如果数据库的接口发生变化，我们不必立即就担心 UI 的变化。只有当数据库的变化导致领域发生足够大的变化，以至于领域的接口也发生了变化，UI 才会相应地变化。

图 1a 表示了我如何使用 UML 标记来图示这种情况。UML 是为 OO 系统而设计的，但是模块和依赖的基本概念适用于大多数类型的软件。这种高层模块的 UML 名称为*包（package）*，因此从现在开始我将使用该术语（这样 UML 监督者便不会逮到我!）。由于这些模块是包，我将这种图称作*包图*（尽管 UML 严格地称其为*类图*）。

在这里我要描述的是一个多层的架构（layered architecture），对于任何从事信息系统的人来说应该是比较熟悉的。信息系统中的层（layer）为我们提供了良好的素材，用来描述我们在考虑依赖时所须顾及的东西。

关于依赖结构（dependency structure），通常的建议是要避免循环。依赖循环将会引发问题，因为它们表示你陷入了这样一种境地，即每个变化将会引发其它的变化，而这些变化又会回到最初的包中。这样的系统将更难于理解，因为你不得不重复地循环多次。我认为不需要把避免包间的循环作为一个严格的规则——如果它们只是局部的，我将会容忍它们。应用程序同一层内的两个包间的循环则不算什么问题。

映射器包

在图 1a 中，所有的依赖都在同一个方向上。这标志着一个依赖集合控制良好，但其并非必需。图 1b 表示了信息系统的另一个常见特征，即使用映射器包来分离领域和数据库。（*映射器（mapper）*是一个提供双向隔离的包。）映射器包提供双向的隔离，以使领域和数据库能够相互独立地进行变化。因此，这种样式经常出现在一些较为复杂的 OO 模型中。

当然，想想你在数据存取时所发生的情形，便会意识到该图并非完全正确。如果领域中的一个模块需要从数据库中获取一些数据，将如何进行请求呢？它不能请求映射器，因为如果这样的话，将会引入一个从领域到映射器的依赖，从而导致依赖循环。为了避开这个问题，我需要一种不同类型的依赖。

到目前为止，我已经从使用其它部分代码这方面讨论了依赖。然而，还有另外一种依赖——接口及其实现之间的关系。实现依赖其接口，反之则不成立。特别地，任何接口的调用者仅依赖于接口，即使实现它的是一个单独的模块。

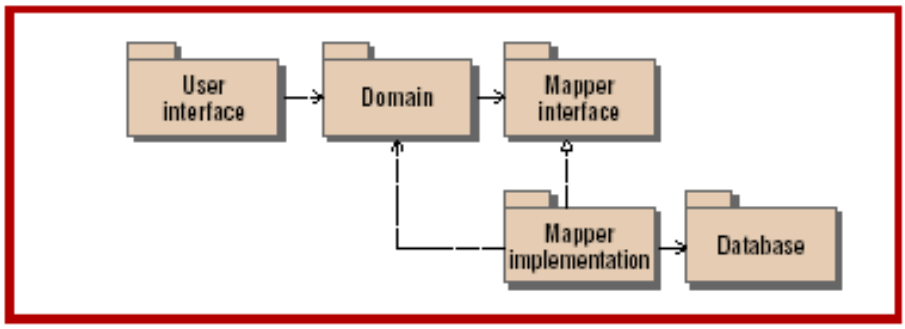


图 2. 引入一个与实现相分离的接口

在图 2 中描述了这个想法。领域依赖于接口，而不是实现。如果没有实现某种映射器，领域将无法工作，但是只有接口发生变化时才会导致领域的变化。

在这种情况下，会出现一些独立的包，但这并不是必需的。图 3 表示了一个包含在领域内部的存储包（store package），而实现它的存储实现（store implement）则包含在映射器中。在该情形下，领域定义了映射器的接口。一言以蔽之，领域包将会与任何实现了存储接口的映射器来一起工作。

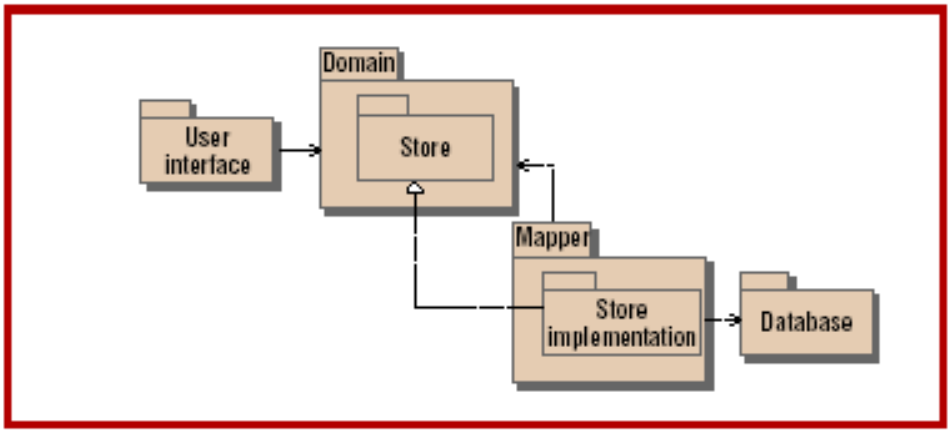


图 3. 在一个包中定义接口，在另一个包中进行实现

在一个模块中定义往往是由其它独立的模块来实现的接口，这是一个打破依赖和减少耦合的基本方法。这种实践有很多种形式，最原始的形式就是*回调（callback）*。在该情形下，要求调用者提供指向一个具有特定签名（signature）的函数的引用，其随后将会被调用。在 java 世界中一个常见的例子就是监听器（listener）。由于监听器是类，它们会更加明确，清晰易懂。

另外的例子是一个模块定义了能够向外传递的事件，以使其它的模块做出反应。你可以把事件看作是定义了监听模块所遵循的接口。回调函数的调用者，监听接口的定义者，以及事件的产生者并不知道实际上调用的是哪个模块，因此这里没有依赖。

我觉得需要总结一下，因为我所说的很多都包含了像“控制良好的依赖”这样模棱两可的词。当我试图定义一个控制良好的依赖集合时，发现很难有一个硬性的指导原则。当然，这里指的是减少耦合的数量，但这并非是问题的全部。依赖的方向以及流程也同样重要，如避免出现大的循环。此外，我会按照相同的方式来对待所有的依赖，而不考虑接口的宽度。如此看来，存在依赖要比过多地担心你所依赖的要更加重要。

我所遵循的基本规则是通过分离接口与实现来打破我不想要的依赖，从而将高层的依赖可视化，然后使其合理化。像许多有关设计的研究一样，这看上去还很不完善。然而我发现它还是很有帮助的——我要说的就是这些。

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

《有效用例模式》
中译本即将上市

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

第一本 UMLChina 指定教材

清华大学出版社

规则对象

Ali Arsanjani 著，Happy 译

查看评论

摘要

因为业务需求迅速变化着，规则也每天都在变化。如何处理这些变化，从而使我们的系统更加有效的可维护、可重用和可扩展？如何为这些规则建模以及处理（表现）它们，以使系统得到更好的重用性、可维护性和性能？

业务规则的变动往往比它们关联业务对象的其他部分要频繁，这些规则通常在一个业务对象的规则方法中实现，并且它们也引用该业务对象周围相关的其他业务对象，这就建立了一个隐含的网络，它们的依赖关系日益增加并难以维护。这种情况下，改变一条业务规则会影响一系列依赖该规则的对象，特别是当实现一条规则的代码在分散类的若干方法中，甚至是若干协作类的方法中，平均信息量就将大大增加。缺少集中控制导致了波纹效应，并且改变一条规则的 if-else 语句元素还会产生副作用。

规则对象模式语言包含 18 个模式，这里只介绍主要的模式：规则对象以及另外两个与之密切相关的模式，判定器（Assessor）和动作（Action）。其他模式只作简单描述，不在本文重点讨论。这个模式语言权衡上面问题域的先决条件，并提供一组连贯且不断展开的模式，来描述处理可伸缩性、可适配性和复杂性的渐进需求。对于服务提供商域模式（Service Provider domain pattern）的上下文环境中[Arsanjani99a;b]，规则对象模式语言可以作为解决问题的最佳方案。

简介：规则设计

在软件系统中大部分都会涉及到规则，有各种各样的类型、领域和规模。为了不断满足新的需求，这些规则经常需要改变。很多作者曾提出了规则分类法，特别是针对业务规则[Odell96]的分类方法。此外，随着日益加快的业务变化和演进，规则也在不断进化。

让我们在一种编程语言的上下文中定义一个简单规则，形式为："if <condition> then <action> else <action>"。如果用这些嵌套的 if-else 语句长链用于代码的不同部分来实现类，将导致代码非常混乱，在需求不断重复变更时，代码实质上是不可维护的。为了避免这种波纹效应，基于面向变更的设计原则[GHJV95],[Arsanjani99;b]，我们以一种非统一的形式把那些经常变化的部分封装起来。同时，我们将 if 语句和 case 语句链从应用程序的层代码（例如业务逻辑）中解脱出来，使得规则结构能够根据业务需要迅速替换。

这使我们在很大程度上隔离系统，从而避免因改变个别对象或为完成某个业务目标而协作的一组对象而产生的波纹效应。

规则具有多种不同的类型，下面有一个场景列表，提供了定义一种规则分类法的依据，这个分类法是基于复杂度、规模、分层和功能的。特别注意，1-7 条指出了依赖于分层的规则类型，8 包含一些非层特定的规则列表。

1. 表单上的输入域需要校验是否非空、是否正确的域值等。这里有专门对象来校验，UI 可通过向这个应用对象发送一个消息来校验这些数据值。

2. 一组互相依赖的字段必须一起校验。

3. 跨表单的复合数据需要交叉校验。一个表单的数据不应和另一相关表单的其他数据冲突。这里可以使用一个中间人（Mediator）模式来实现，它将检查是否所有的数据都是完整的，而不需要各个表单互相知道对方。

    3.1 非法的组合需要识别出来并报告或完全禁止；

    3.2 合法的组合要测试它们的有效性；

4. 在中间层进行策略检查和业务逻辑验证以形成大批应用（或业务）逻辑，它们可以被多种不同类型的客户端访问：例如 PDA、瘦客户端、胖客户端等。

5. 在数据层用触发器和存储过程的方式检查规则。

6. 中间件通讯规则。

7. 多种安全性规则；授权、验证和认可等。

8. 不管所在何层，规则都能关联于计算方法（函数、映射、转换，通常可以由数学公式来表示）。例如：当选定一个保险险种（加入到一个保单），它可以与损失原因相关联，这是固定不可改变的。如果一个险种有若干损失原因，当计算该险种保险费时，计算每种损失原因的保险费并汇总。如果一个险种没有损失原因，那么该险种具有$0.00 保险费。

8.1 一些规则限制并约束业务过程以及输入中驱动规则的数据；

8.2 一些规则属于工作流顺序、“路由”、“规则”和“角色”；

8.3 一些规则设立“同意条款”或“前提条件”，例如实际中更具法律性质的“服务级别协议”；

8.4 特定类型规则的执行导致“正包含”，如一个险种当且仅当...是合法的。有时，还有“负包含”：例如如果申请者的邮编在<高欺诈区域邮编列表>中时，那么我们将必须做一个预付费者的名片。另外一些时候，我们还有“非法组合”情况，它联合一组条件导致一个非法的状态，如假设申请者欲申请其第二栋房子的购房贷款，而他的主房屋位于高地震率地区且没有保险，再如果他的信用值低于<xxx>值，那么我们将不得不拒绝这个贷款。

9. 规则的设计和实现是两码事。例如，规则可以设计成业务约束，而在数据层中，可以使用参照完整性规则来实现，用特定类型的关联/关系来建模。

模式 1：规则对象

意图

使计算机化的业务过程设计和实现具有可扩展性、可适配性。通过可插拔的规则，使整个业务过程不因规则的内部变化而受到影响。

动机

例 1

写一个程序实现温度的摄氏与华氏之间的互相转换：F→C 和 C→F。

第一眼看来这非常简单。实际上，在实现这个设计过程中，有很多“规则”出现。输入一个数，点击 OK 按钮，在另一个文本框中显示另一种单位的温度值。可以实现如下：有两个文本框（一个用于显示摄氏，一个显示华氏）和一个按钮（用于开始计算）。你可以在某个文本框中输入数值，并点击“计算”按钮。计算过程要检查在另一个文本框中是否已经有一个被转换的值，如果有，它是否是一个合法的转换；如果不合法（或没有数值），重新转换；否则，弹出一个信息提示“已经转换。”

所以，在真正转换前，需要做一些验证和检查。对象经常不会“无知地”执行某些动作（执行方法），更确切地说，通常要检查一些条件，如果那些条件满足了，对象才能执行动作。当然，对于更大型软件系统中更重要的问题，这是一个有点微不足道的示例。

接着，我们决定加入开氏温度转换。首先检查特定文本框中是否已经输入值并且另外两个是否为空。如果其他文本框非空，要提示用户清除。而在提示清除文本框内容前，如果它们不是合法的转换，那么它们将无法获得焦点。

想象一下如果不是温度值，而是货币值，那么问题是类似的，解决方法也是类似的。

例 2

你正在设计一个财产保险的应用。保险公司有一些客户（保户），它们购买了一些保单。每份保单都关联着若干保险险种，以针对某些损失原因（COL）来保护客户的建筑物和其他财产。建筑物以位于一个地理区域为前提，保险险种可以分成四种：房屋保险、个人财产保险、Debris Removal 和 Special Provision，每种分类都有一组 COL 关联着。

这个需求的核心部分是业务规则，你会发现无数的规则操纵着一个业务应用程序，这个也是一样。这些规则往往分散在各个层中，如 GUI、支持业务过程的中间业务逻辑层和数据库层，每层都可以有各自的规则集合。

让我们看看一些对域问题无需过多说明的业务层规则：

- 没有重复的保险险种；
- 为了在保单中增加一个个人财产保险，必须已经存在房屋保险；
- 或者加入 InflationGuard 保险， 这是一个附加的险种，可以用于当房屋保险是保单的一部分，并且没有包含一个 Debris Removal 保险的情况；
- 保单中不应包含重叠日期的险种；

保单、险种和单个险种关联的规则经常发生变化，为了确保市场份额和收益，需要经常改变这些规则以适应迅速变化的业务需求和竞争。为了快速改变业务需求、分析不变量（analysis invairiants）（全体业务规则）的设计，我们需要在不产生副作用的前提下定位和修改规则。具有代表性地，我们也许希望在以后某个时间恢复规则先前的某个条件，例如当一种优惠过期。因此，为了适应需求的变化，必须要让规则能够可插拔，可适应。为此，必须要很好的组织规则。如果规则分散在应用程序的各个角落，那么改变它们和避免副作用的代价会很昂贵，那样不仅麻烦，在某种程度上，根本就是不可行的，这样业务过程的可适应性也遭到破坏。

使用一组硬编码的 if 语句来构成规则是难以维护和改变的。所以我们建立了一个小型框架，允许我们进行规则检查，它可以根据业务需要，对条件和动作进行插拔。

不过规则一般不需要全部改变，有时仅需要增加新条件来增强原来的条件判断，新的和修改过的动作需要被替换或重新组合成新规则。因此，为了可插拔性的要求，我们对规则进行组件化。我们从封装它们的要素部分开始：条件（condition）、动作(action)、传递给它们的属性(property)（上下文）、结果对象(result)，它们构成一个规则对象，它本身就是一个可适应的组件。

这让我们可以迅速改变、扩展、重组和重用规则、规则要素以及组件。例如，为了检查一种新的财产保险类型，可以方便地插入一些条件，如个人财产。如果新条件应用了，我们将执行一些动作，即使是一些简单的，如显示一个警告信息告诉用户它们输入了一个日期重叠的险种，因此不能加入到同一个保单中。

典型业务规则实现

规则在一个对象的实现中通常位于多个方法的函数体里，它们通常是分散的，并且通常是嵌套的 if-else 语句，这都造成代码混乱且不够智能，且难以维护。

规则作为方法（规则方法）

为了可以适当地实现，首先将确定下来的业务规则分列出来、分类，然后实现成一个单一方法，常常返回一个布尔值表示这个规则是否应用。如果规则过于复杂，可能会调用其他的规则方法，并试图和其他业务对象进行协作执行它们的规则判定。例如，在财产保险例子中，有一个叫 Debris Removal 的保险，这是一个附加的险种，可以用于以下情况：

- 保单中已包含房屋保险；
- 保单中没有通货膨胀保护；
- 对业务服务没有应用 SpecialProvision，并且 DebrisRemoval 限度已经增加。

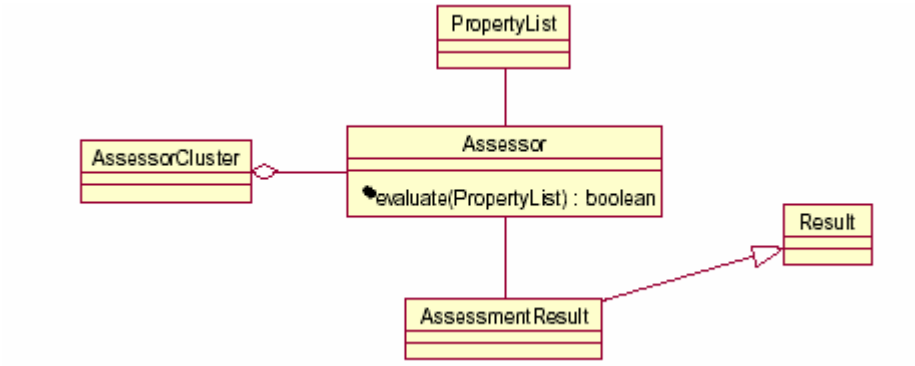
规则对象对等模式（Peer Pattern）概览

对等模式

规则对象在判定器(Assessor)、动作(Action)、属性(Property)（上下文(Context)）和结果之间充当中间人(Mediator)的角色。规则对象决定哪些 Assessor 用于哪些 Property 的判定，并在判定结果中产生可能的 Result。如果判定成功，相应的动作将被调用，这可能会更新或改变属性的状态，结果也将记录入一个动作结果对象。

判定器

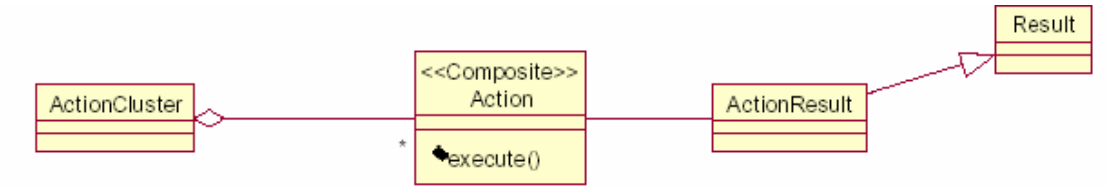
Assessor 封装一组输入 Property 并基于它们进行条件判定，并将判定结果记录在它们的 AssessmentResult 中。Assessor 类似于命令（Command），一个 Command 执行一个操作，而一个 Assessor 基于输入状态或属性进行条件判定，不同的地方在于它拥有一个 evaluate()方法而不是 execute()方法，并且与一个属性列表、一个错误日志（ErrorLog）或结果集一起协作。



图：判定器（条件求值器）

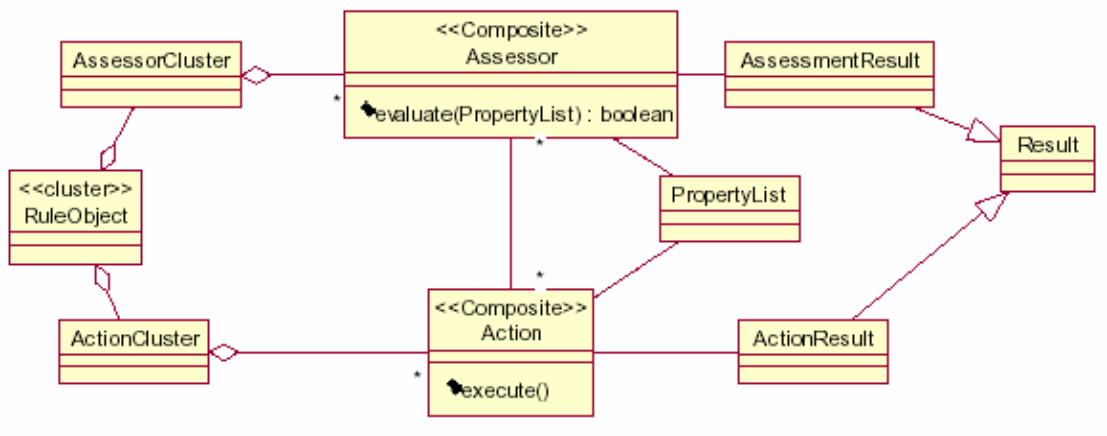
动作

一个 Action 通过状态的更新或调用其他协作对象的行为执行，执行结果将记录入一个 ActionResult 对象中，这个对象可以包含一个 ErrorResult 对象或 ErrorLog 对象。动作对象可以按策略（Strategy）模式实现，也可以按访问者（Visitor）、解释器（Intepereter）、命令（Command）等模式实现。下面是它的集群结构：



图：Action 模式

动作对象和判定器对象的关系如下：



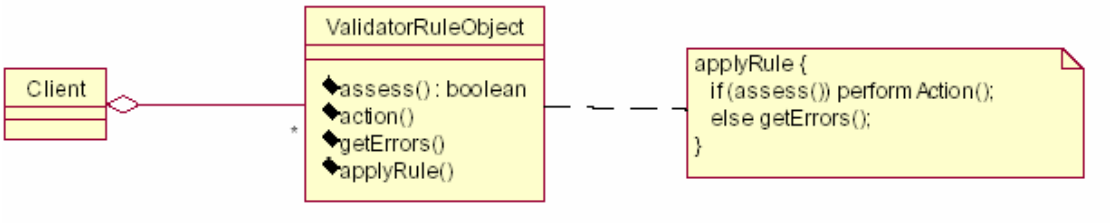
图：动作和判定器

规则对象的渐进展开：简单规则对象

如何提供一种可以用于简单情况的设计同时又具备一定的伸缩性呢？规则对象根据复杂度的渐进展开可以得到最好的理解，平衡各个附加层的复杂度和功能性。

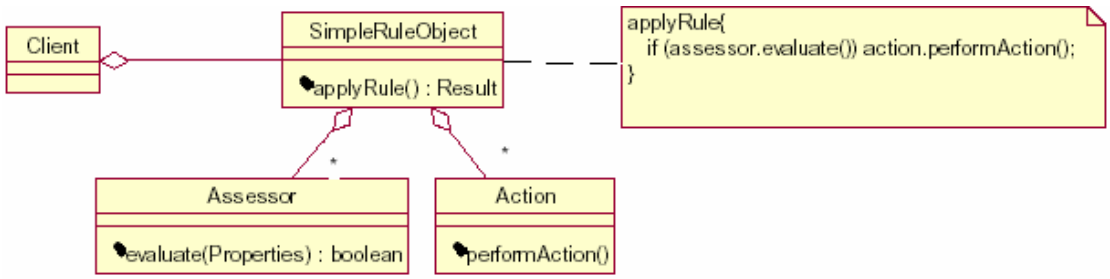
通过透明地嫁接规则对象，能让规则对象可扩展地、可适应地管理对象行为。每个规则对象表示一条对象在域中必须实施的规则。对象动态地管理它的规则对象，以独立的对象来表示规则，不同的业务过程流与管理规则以及它们的交互都互相独立开来，这样改变它们的过程就简单的多了。

规则对象从一个简单的校验器入手，它使用方法来进行条件判定和动作执行，校验器有一个判定条件的方法，当所有条件值为真值，那么调用一个和更多的动作方法。



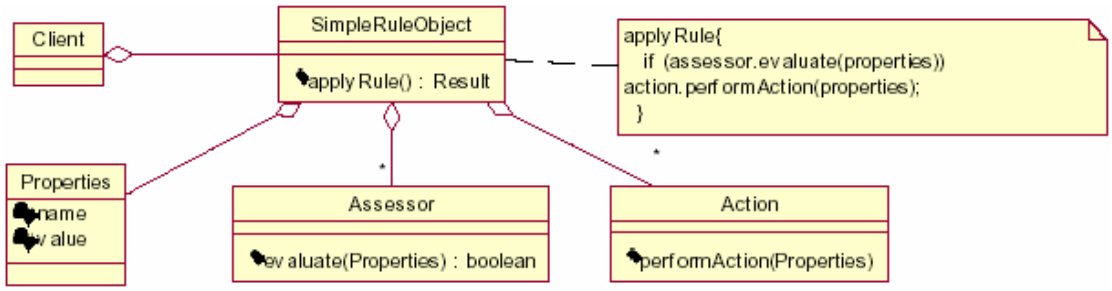
图：校验器

下一步是一个简单规则对象。现在你发现自己开始不得不定义数量很多或复杂的方法来判定条件，并且这些条件有些还正在变化。所以，你想将这些方法封装一下，以易于修改，动作也是如此。所以决定将条件抽象成 Assessor 对象，让动作成为 Action 对象，规则对象作为 Assessor 和 Action 的中间人（[GOF]）。这可能会使用一个工厂模式（Factory）（构建者（Builder）或抽象工厂（Abstract Factory），亦或是单单工厂方法（Factory Method））从序列化对象[Riehle98]来创建合适的 Assessor 和 Action 对象。另外还有一些在规则集群模式语言中的模式，解决对规则对象、或它所包含的 Assessor 及 Action 对象进行“哈希和缓冲（Hash&Cache）”的问题，以支持按需创建或插入规则对象。现在你得出了如下设计，请注意 SimpleRuleObject 是一个抽象类，Assessor 和 Action 可以是接口和抽象类，具有一些缺省的行为。



图：有判定器和动作的简单规则对象

第三级复杂度/伸缩度可以满足经常改变需要进行判定的字段或属性的目的，使用 Beck 的可变状态（Variable State）和 Yoder&Foote 的属性列表（Property List）模式可以很方便地创建一组名－值对，把它们传入到 Assessor 中进行判定。这样 Assessor 的代码也不会因引用个别（视图级）的对象实例而过于混乱，它将引用一组模型级的名－值队实例，反映输入到规则对象 Assessor 中的字段和属性集合。同时也可能发现你的 Action 对象需要以某种方式更新字段，如将一个州名缩写扩展成全名，根据一个邮编得出州名等，另外一些步骤的复杂度将可以得到如下的设计：



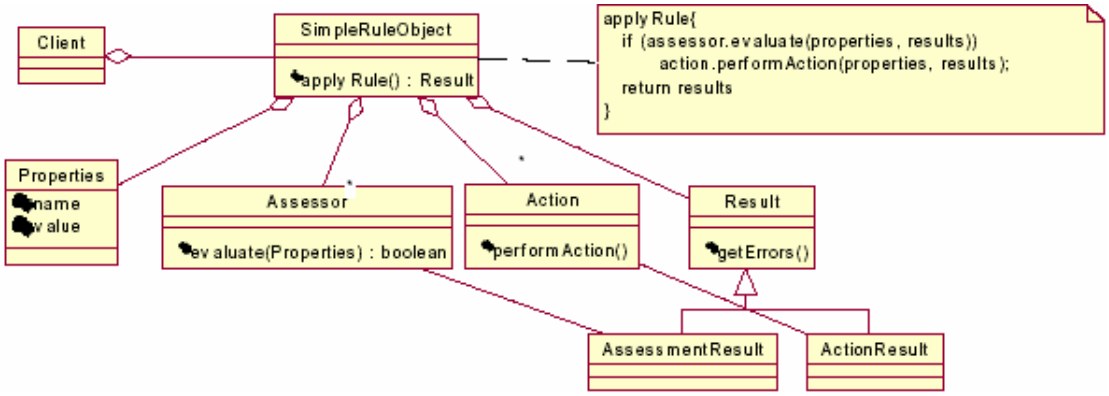
图：具有 Properties 的简单规则对象

第四步是你何时需要记录和返回智能的信息以表示处理的结果。这样，当一个规则对象判定了一组属性，结果需要被记录成一个 AssessmentResult，例如：

```
if( assessor.evaluate("BillPayMethod") )
    action.performAction();
else
    assessmentResult.addErrorFor( "BillPayMethod");
```

用户更关注在它们的数据录入时什么地方出错了，以便于纠正。还有很多更多原因需要记录和报告结果，例如调试（为程序员）、性能（架构师）、数据挖掘（市场、业务人员）、问题解决（用户）等等。

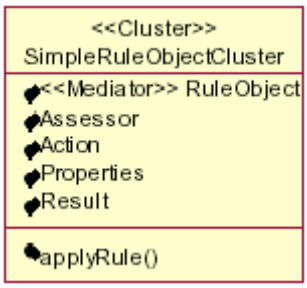
下图是对简单规则对象的简单装饰：



图：具有 Result 体系的规则对象

这个模式实现的变种包括规则对象拥有指向 AssessmentResult 和 ActionResult 的引用，而不是通过 Assessor 或 Action 对象引用。

让我们将上面类的集群称作简单规则对象集群，表示如下：

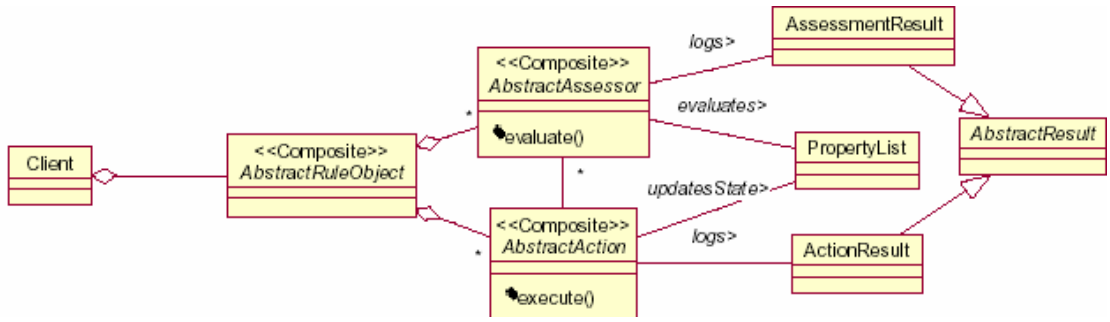


图：规则对象集群

这意味着集群充当一个 facade 的角色，拥有方法 applyRule()，以及要素成员或协作的规则对象、判定器、动作、属性和结果对象。这个集群可以实现成一个组件，可以相应调用参与者的公共方法。

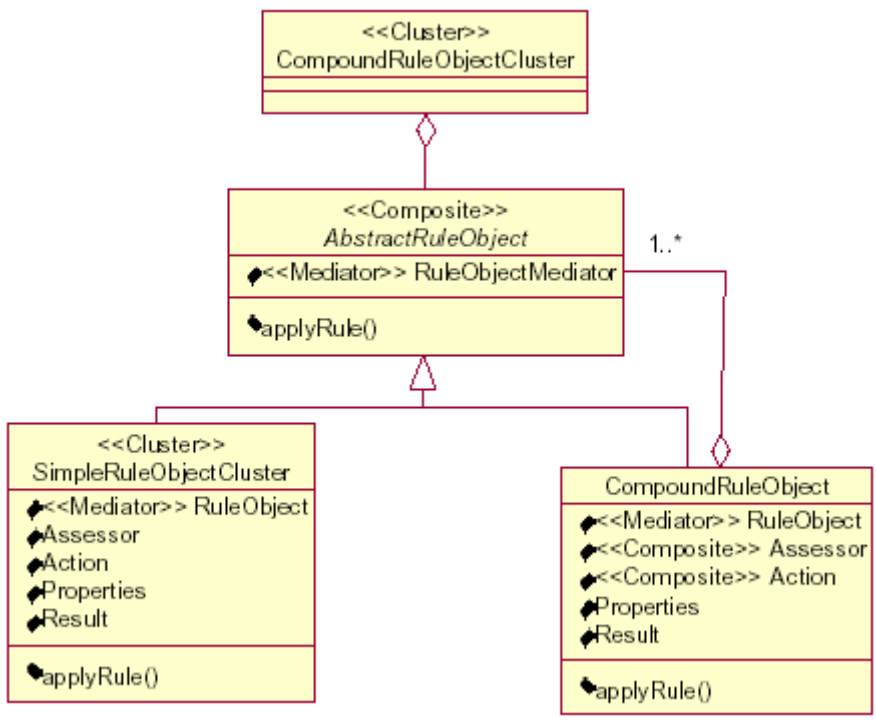
复合规则对象

至此，我们已经讨论了组成一个简单规则对象结构的四个复杂度级别，对于更复杂的情况，当一个组织想存储规则并能够访问一组可重用的条件（判定器）和动作时，可以使用合成模式（Composite）来设计这样一个复合规则对象：



图：复合规则对象，视图 1

下图是一个更加展开的视图：CompoundRuleObjectCluster 包含一个复合抽象规则对象。它的叶节点是简单规则对象，这在上面已经讨论过了，它的部件可以包含嵌套的规则对象聚合结构，这些规则对象可能合成 Assessor 和 Action 对象。这种更加复杂的方案用处之一是用来描述类似电信条款的规则，这需要一组嵌套的规则对象，以及它们相应嵌套的判定器和动作对象组。



图：复合规则对象集群

先决条件

- 业务需求的改变致使规则设计实现的变化。业务规则通常比业务对象其他部分的变动更加频繁。在分析、构架、设计和实现阶段，业务规则可能需要重新定义和更新以反映策略和业务的变化，并且随着新业务需求增加而不断演进。改变已有的规则必须保持未用规则的完整性并与之保持一致的状态，一致性意味着对一组条件的修改不会影响到系统的其他部分从而造成不必要的副作用。

- 规则也许对时间非常敏感。当规则在一个受时间限制的域中就是如此。例如，应用于服务提供的规则只在某个限定的时间段内有效。在这种域中，拥有一些迅速扩展和变化的规则，它们可能频繁改变，甚至有时候可能每天都会变化。虽然在程序中改变规则是有必要的，但是代价也颇昂贵，这种修改通常扰乱代码并具有副作用，而且需要进一步的调试和测试。进行扰乱代码的修改是不安全的并且代价昂贵，潜在的副作用导致需要更广泛的回归测试和组件的重新鉴定。

- 规则可能被集中化，以使它们易于定位和修改。通常，规则是分散在设计和实现过程中，一般都是 if-then-else 语句的形式，并且伴随很多依赖关系。把规则需求跟踪到规则的设计和实现阶段中，将是一个非常易错的并且是资源密集型的项目。

- 规则可能被非集中化，将它们分配到相关的类和集群中。对象具有 manner，manner 管理对象的哪些方法被调用，以确保状态集的一致性和合法性。这样，规则可以分配到类中，每个类（业务对象）有一组方法，规则管理它们合法的状态，以及发送给协作对象或自身方法的合法消息序列。规则通常检查一个对象或一组协作对象的状态，操作于在上下文中提交给它们的数据，我们需要跟踪对规则应用于这些数据的状态。规则应该知道相关的属性（数据），但是对于从业务角度，这些属性应该被那些用于检查它们完整性的规则所忽略。

- 规则应该具备伸缩性。为一个小型应用而设计的规则结构应该也适用于一个更大型应用的需要和非功能性需求。初期，它们可能只应用在很小的环境中，而很快就需要处理更大规模的事务，因此，规则的设计要求简单而又可缩放。

- 规则类型的非统一对待。对不同类型规则的不同对待导致很多不必要的后果，因此需要尽量一致对待，不过也要注意每个业务对象都具有各自的 manner。从不同角度看，有不同类型的业务规则应用于不同的应用层，如用户界面、应用逻辑层和持久层。规则要对所有类型有一个整体的视图，无论它们是否是一组数据元素，都可以在应用服务器的中间层来确定（报告）它们合法和不合法的组合、值或复杂的业务逻辑。

- 代码混乱：嵌套的 if-then-else 语句扰乱了代码，使之难以维护。规则经常通过嵌套的 if-then-else 结构来实现，但实际上规则的增加是一段一段的，并非自顶而下的设计。

- 架构分层和规则：不同的规则可以应用于架构的不同层中，规则可以应用每个不同的层：GUI、Webserver、应用层、中间件、数据库等。位于不同层中的规则有不同的实现机制，如 GUI 中的简单校验、应用层中复杂的交叉规则检查、数据库层中的存储过程和触发器。但是它们基本的结构要类似，以易于跟踪和一致的应用。

- 业务域包含由一组业务规则集控制的业务过程。这些规则从公司的策略、工作流、操作和手续过程中提炼而来。虽然要求是标准的过程，但仍然经常需要改变。这些规则通常设计于那些要随着业务需求变化而改变规则的信息系统中。

- 规则由管理层创建且是可见的，规则以及规则修改对程序员也要是可见的。在业务人员改变业务规则时，开发人员将它们实现成代码，管理层必须知道哪些规则已经实现了。

- 规则是一个域中的对象相互交互和改变状态的合法途径，它们应该被看作是对象思维模式（object paradigm）最好的概念，并且早在分析过程中就要被标识出来，而非事后的想法。在这个新的思维模式中，除了有对象标识、状态和行为，一个类还有 manner（控制行为或方法的规则）。Manner 是规则加上方法（对象行为），再加上应用规则和对象中控制交互和协作协议的元数据。

应用性

使用规则对象

- 为业务对象动态且透明地增加/修改条件、动作和规则，就是说，增改时不会影响其他对象和规则。

- 当复杂性和规模使得规则方法不可再用时，规则可以通过 if 语句或返回布尔值的方法（如 isCompatible()）实现。而当它们增长到不再是简单的检查时，就要将他们实现成规则对象了，特别是需要经常改变规则时更为适用。

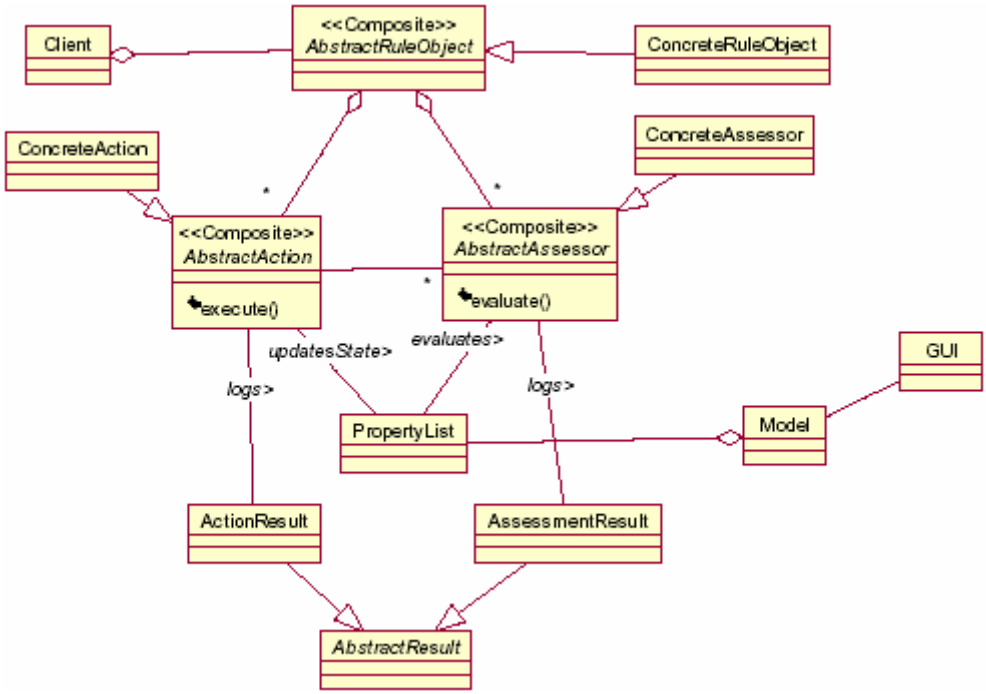
- 当通过实现需求变化来维护一个系统时，需求可以被分析成规则、不变式(invariant)、业务需求（谓词和条件）、策略、协议条款等，这样我们可以进行不扰乱代码的修改。重用现有的条件和动作来创建一个新的规则，只稍稍异于现有的规则，但是它只在短期内起作用（高度易变的需求）。

- 业务对象应该知道它们自己的 manner：如何使用它们的方法和其他协作的业务对象一致工作，什么是合法的状态，什么是业务对象间非法的状态组合。这样，控制方法使用的规律，需要元数据来存储这些信息，以及和具体表现这些规律的条件和动作一起称作业务对象的 manner[Arsanjani99;a]。规则不是作为每个业务对象中逻辑通过它的表达式来单独访问，而是被整理在各自的对象中，以利于非干扰式的修改和插拔，因此业务对象将包含一系列业务规则。

- 当描述一组业务对象的特性时，需要根据不同规模和复杂度的限制进行改编和定制，规则对象可以对 GUI 数据录入字段简化成一个简单的校验器，或者可以升级到一个具有复合判定器（条件）和动作，具有属性和结果的合成规则对象，下面有完整的图示。

结构

业务规则要比业务对象中其他部分的变动要频繁，如果这些规则封装在自己的类中并且分离控制，插拔地使用或重用，那么这些变化可以减少到最少。因此，我们对规则和它们的条件、动作进行抽象，以使它们可以互换、可以插拔。



图：复合规则对象

通过对规则对象的非干扰式修改来维护规则，增加和修改已有的条件和动作，通过使用规则对象增加属性（上下文）来检查合法或非法的状态组合。一个规则对象集群是条件（Assessor）和动作，以及一些辅助类的合成物。辅助类包括提供“字段”的名－值对，这些“字段”用于条件（Assessor）的判定，判定结果记录在一个 Asscessment Result 中，如果这些条件都应用了，那么 Action 将被执行，以改变对象或其他对象的状态。这些动作产生的结果记录在一个 Action Result 中，以供报告或分析用。

参与者

- 抽象规则对象

定义对象的接口，可以让规则动态地加入。

- 具体规则对象

定义对象，让附加的规则可以加入。

- 规则对象

条件和动作的中间人，应用规则

- 判定器（Assessor）

检查条件并存放结果

- 动作（Action）

基于相应判定器的成功或失败执行动作，记录结果并且改变协作对象的状态

- 属性（上下文）

传入的信息或状态，用于条件判定（判定器）或状态更新（动作）。

- 结果（Result）

提供一个 AssessorResult、ActionResult、Error 的父类，其他的参与者可以在此记录结果并报告给客户端。

- AssessorResult

Result 的子类

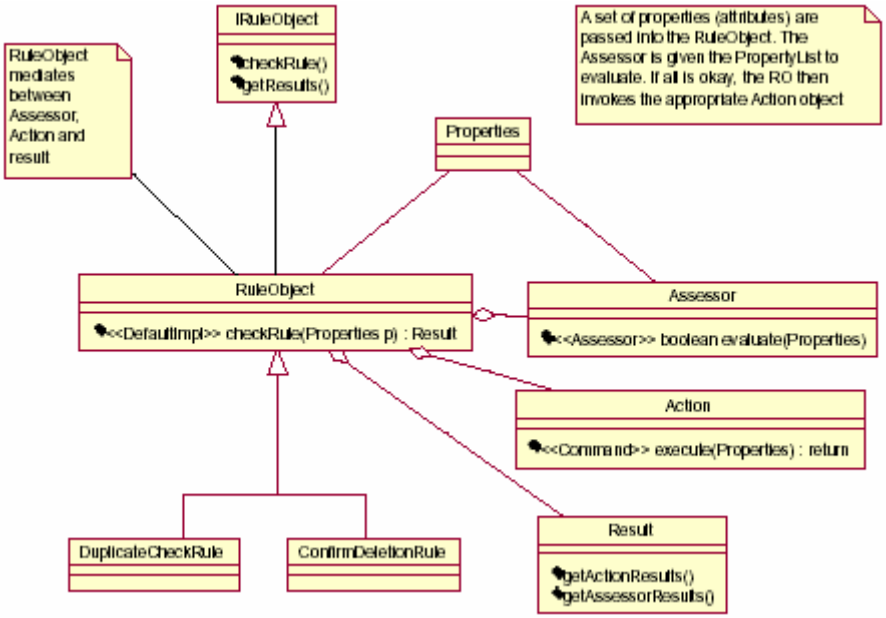
- ActionResult

用于记录动作结果，Result 的子类

- MediationStrategy

在复合规则对象、判定器和动作的情况下，决定我们将如何执行判定器和动作，或是规则对象。

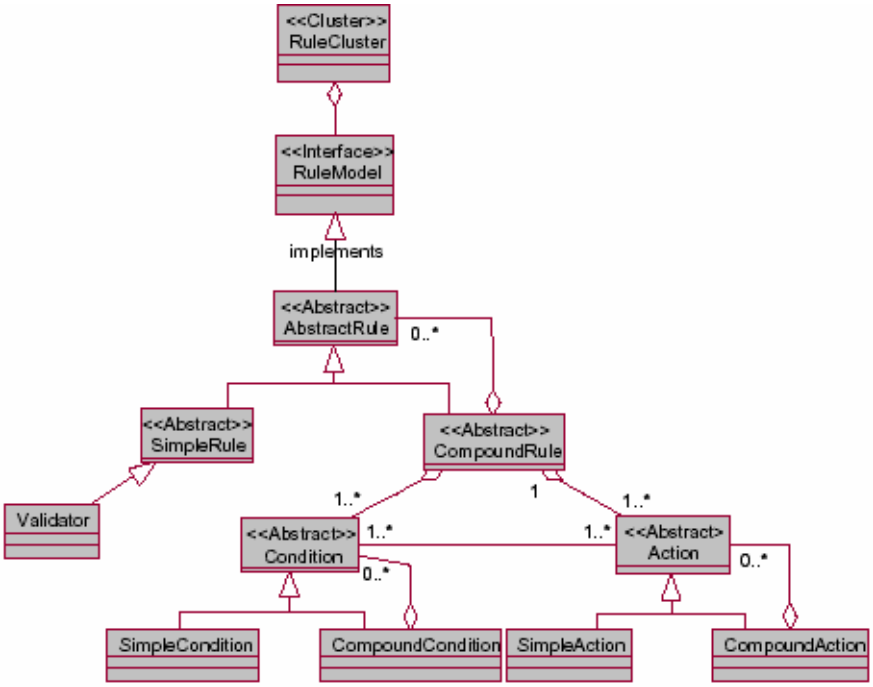
简单规则对象：静态视图



图：规则对象及协作者的静态视图

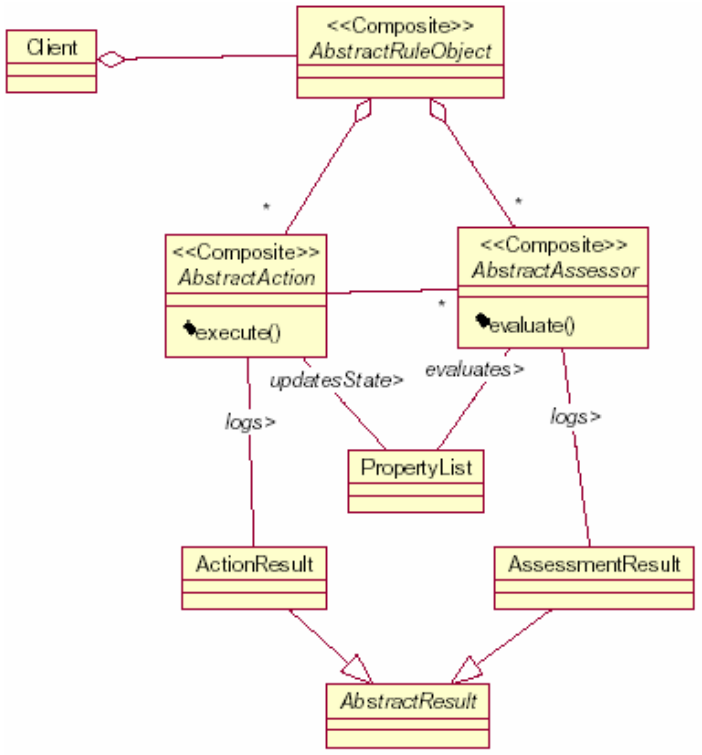
复合规则对象：静态视图

规则集群持有合成规则的一个集群，每个合成规则都有它们各自可能的合成条件和动作。一个简单规则是合成规则的一个叶节点，它可以独自存在来处理一些诸如 GUI 字段的编辑和校验。



图：复合规则对象和合成体系的静态视图

下面还有一个复合规则对象的另一种视图，着重表现规则对象以及它的组成要素的动态可插拔性，而上图主要表示它的合成结构。



图：复合规则对象的另一种视图

协作

下面是一组示例协作：

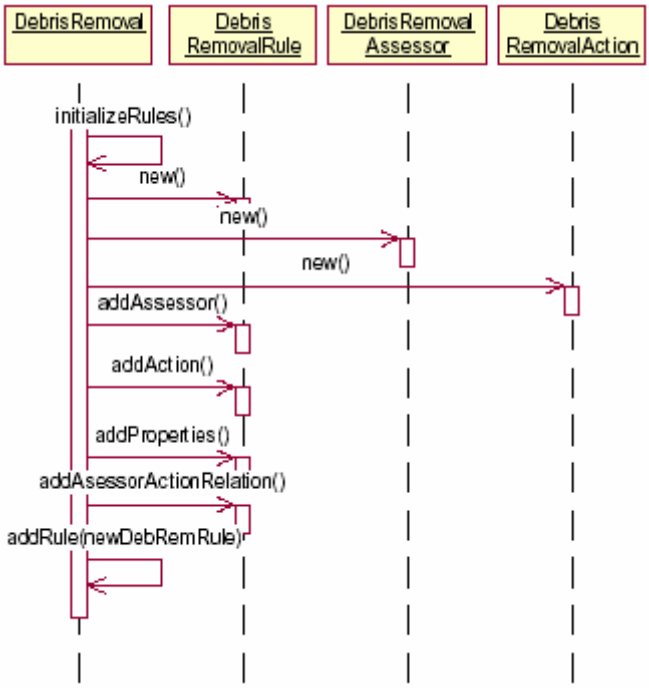
1. 设定规则（例如缓冲到一个哈希表中：“缓冲和哈希”）

2. 提交候选的状态（传入属性或哈希表，或仅将你想检查一致性的状态作为参数）

3. 使用判定器。基于提交的状态检查条件，一个判定器通常会遍历列表或哈希表以检查每个条件，或者有一个策略，使用某种算法来检查每个条件，最简单的就是一次循环进行。你可以通过创建、扩充一个特定判定策略来选择或定义你自己优化的判定器算法。

4. 对每个判定器组，会有一组对应的动作组，它们必须被执行，或有一组 ResultState 或 ErrorLog 对象，必须写入、创建或反馈，对于不重要或未实现 ErrorResult 或 ResultLog 的情况，它们可以是 Null 对象。

下面的时序图描述了上面设定规则的步骤：



图：设定规则

下面是一些缺省协作的变种：

规则对象的一个策略（多对多）
if(assessorList.assess(instate))
actionList.perform(inState, outState);
else
errorLog.reportOutcome(outState);
	简单策略：（一对一）
	if(assessor.evaluate(instate))
	action.performAction(inState, outState);
	else
	errorLog.reportOutcome(outState);

合法的组合：
一个组合是一个状态，此状态耦合于一个策略和一个规则对象。当规则应用了，一个初始化的 InvalidState 状态迁移到 ValidState 状态，否则，ErrorLog 记录非法状态错误和原因
	非法的组合：
	如果这里 inState 是一个 InvalidState 状态，那么，调用 errorLog.logThisAsInvalidState()
	遍历所有的规则对象（使用特定或缺省的策略），看这个 inState 是否匹配一个 InvalidState 组合，如果是，reportInvalidState
	outState = assessor.assess(inState);
	if(outState.isInvalidState())
	errorLog.reportError(outState);
	else
	continue;//继续下一可能非法状态的检查

结论

- 规则变得更易于改变和重用，简化了维护工作，非干扰式的修改可以用来维护规则以及它们的判定器和动作。单独的判定器和动作可能在多个不同的场景中被重用。以这种方式构建的系统更加符合开放封闭原则(open/closed principle)[Meyer84]。对业务规则的改变将很少有波纹效应，它们封装在一个规则对象中。通过在一个规则对象的上下文中创建复合规则，或在简单规则的情况下改变检验器的策略，就可以通过增加更多的规则对象来得到一个新的规则。

- 规则可伸缩性提高了。对于规则、判定器和动作的合成物，它们可以存储在数据库中，或者根据增长量、需求和适用性进行缓冲，与单个规则的增长相对称。

- 规则类型的统一对待。设计和实现业务规则没有一个最好的方式，实际上，规则具有多种类别：业务规则、校验规则、使用规则、协作规则等。校验规则多用于 GUI，一些输入域是必填的，有些输入域的值必须在某个范围中，有些则与其他输入域有相关的依赖关系，还有输入特定的值，其他的某些输入域变成有效或无效等。业务规则检查合法的输入组合及非法组合。每一层的规则都以相似的方式对待，即使每层在定制规则时都有它们各自的独特性。例如，在 GUI 中校验一个文本框可以不用合成的规则，而使用简单的校验器。而一个具备合成判定器和合成动作的合成规则可以用来实现一些客户关系和电信条款的需求。

- 规则变得易用测试。对业务规则严格要求也意味着每个捕捉到的业务规则必定有测试手段。

- 需要新的子系统来管理规则对象，并且让特权用户来改变这些规则。

- 公司的策略库。规则对象为软件开发组织或公司准备了一个带有集中规则库的基础设施。虽然规则（和公司策略）分散在组织的结构中，它们可以被集中管理和浏览，在一个集中的地方进行定义和修改，允许所有对规则或规则类型感兴趣的人收到它们改变的通知，这可以通过观察者（Observer）或发布者－订阅者（Publisher-Subscriber）模式实现。

- 规则对象库可以让公司的业务主管通过基于 GUI 的规则浏览器来定义和操纵规则和策略，接着，它们可以发布到整个组织结构中，这样，负责不断将规则实现成业务对象的开发人员有了共同的基本原则和可追溯的参考点。

实现

考虑下列实现问题：

- 避免将简单规则的逻辑放入规则对象，除非它们需要经常变化。在规则不需插拔、适应或扩展的情况下，规则方法是一个更简单的方式。

- 重用现有的条件和动作相对于重用一个规则对象而言，是一种更好的选择。

- 设定规则对象可能需要一些工作，不过一旦框架设置完毕（见下面的代码），那么定义新规则、动作和条件将非常简单。

- 策略模式可以用来检查规则，因为可能会有一组相关的规则系列，根据特定对象的状态而应用。命令模式也可以用于检查规则，或在条件检查后执行一个动作。判定器抽象了一组需要检查的条件。例如，一个集群的状态由组成它的对象状态构成，每个状态都需要进行合法条件检查，或可选地检查非法条件的存在，因此需要检查合法与非法条件的排列。

- 规则应用的结果。规则对象的应用可以被跟踪，表现成规则应用的结果，因此它们的条件和动作都需要日志记录下来。当然为效率考虑，可以禁止为每个简单条件和动作组进行日志记录。

- 规则复杂度：规则可以很简单，或很复杂（复合），它们可以看作合成物，或简单的校验器。可以使用一个合成、策略或命令模式来实现。

在业务规则设计中较少考虑到的一个因素是错误处理是与之紧密耦合的，这非常重要，也许只是为了知道为什么一条规则失败了或成功了，因此，错误处理将作为实现部分考虑的一部分。

实现指导

每个业务规则封装在它们各自的类中，可以是简单规则对象或复合规则对象。一个合成的规则对象由合成结构构成，包含判定器和动作。RuleObject 包含一个合成判定器、一个合成判定器和 ErrorResult，ErrorResult 是为了记录在判定器求值后，禁止规则触发动作的错误。一个判定器类似于命令，他有一个方法"assess()"，返回布尔值，如果是一个合成结构，判定器在返回 true 前必须成功判定所有的构成元素，如果有一个问题，判定器就将一个错误信息/条件记录入 ErrorResult。

如果所有的都没有问题，判定器也已经求出条件的值为 true，接着将执行动作。动作是一个命令，也可能是一个合成结构，会改变当前正使用或包含规则对象的客户对象的状态，还有可能是和其他对象协作，根据判定器的求值结果为系统创建一个合法的状态。在任何时候有错误发生，它们都将记录到动作对象引用的 ErrorResult 对象中。

一个规则对象通常构成一个集群，这个集群由 RuleObject、Property、Assessor、Action 和 Result 构成。传入 RuleObject 的属性或状态被判定器进行判定和求值。如果这个判定成功，RuleObject（在 Property、Assessor、Action 和 Result 之间充当中间人）将判定结果放入结果对象中，然后在向 Action 请求执行，以改变某些 Property 的状态。

动作执行的结果也放入 Result 对象，这可以为一些实际情况提供元数据。例如，一个系统状态发生变化，属性通常从一组协作或单一对象的属性值封装成状态。因此，我们不仅只想对一组属性进行判定，根据结果执行动

作，我们还想得到求值的中间结果和动作执行的中间步骤，这些信息记录在 Result 对象中。状态或属性组，以及那些作为判定器求值基础的值，把它们传入规则对象或规则对象集合中，并创建它自己的 Property 对象，再在需求驱动的基础上，把它们传入它的 Assessor 和 Action 对象中。

示例代码

这里有一个应用于 GUI 层的简单规则对象例子，虽然如此，规则对象可适用于所有的层：应用、协议和持久层等。

```
class BusinessRule implements ActionListener{
  private JDialog theDialog;
  private Frame theFrame;

  public void setTheDialogJDialog( aDialog){
    theDialog = aDialog;
  }
  public BusinessRule(){
  }
  public boolean assess( int number ){
if (MIN_SERVICES <= number && number <= MAX_SERVICES )
  return true;
return false;
  }

  public void actionPerformed(ActionEvent event){
if (event.getActionCommand() == "SubmitButton"){
  theDialog.dispose();
}
  }
}
```

这是一个最简单的例子，规则对象仅仅是一个类，作为一个 Listener 使用，每次在对话框中的一个输入域按键，assess()都将被调用来判定是否输入了合法的值，这个例子中将检查它的范围。

下面是使用的实例

```
/**
 * 注册输入人数的输入域
 * @param field 输入人数的文本框
 */
public void registerNumberField( final JTextComponent field){
  numberOfServices = filed;
  DocumentAdapter documentAdapter = new DocumentAdapter(){
protected void parseDocument(){
  int count = 0;
  try{
    count = Integer.parseInt( filed.getText());
  } catch (NumberFormatException e){
  }
  if (rangeRule.assess(count))
    serviceCount = count;
  else
    serviceCount = 0;
} // parseDocument()
  };
}

将规则对象注册成 ok 按钮的 listener
public void registerOKButton(final JButton btn){
  submitButton = btn;
  submitButton.addActionListener(rangeRule);
  submitButton.setActionCommand("SubmitButton");
  rangeRule.setTheDialog(myParentDialog);
}
```

示例代码：规则对象框架

定义一个规则对象，包含它的 Assessor 和 Action，Property 和一个 ActionAssessor Map 辅助决定哪个 Action 和哪些 Assessor 相关。

```
public abstract class RuleObject
{
  private Vector assessorVector;
  private Vector actionVector;
  private RuleProperties ruleProperties;
  private ActionAssessorMap actionAssessorMap;

  public RuleObject(){
actionVector = new Vector();
assessorVector = new Vector();
actionAssessorMap = new ActionAssessorMap();
ruleProperties = new RuleProperties();
  }
  public Boolean applyRule(){
Boolean assessorReturns = true;
Assessor tempAssessor;
Action tempAction;
//
Vector tempVector = getAssessors();
Enumeration e = tempVector.elements();
```

```
// 对每个判定器，调用求值方法返回值
while (e.hasMoreElements()){
  tempAssessor = (Assessor)(e.nextElement());
  assessorReturns = assessorReturns &&
     tempAssessor.evaluateAssessor(getRuleProperties());
  // 决定这个判定器是否有动作执行
  Vector tempActionVector = getActionForAssessor(tempAssessor);
  if(tempActionVector != null){
    Enumeration tempEnum = tempActionVector.elements();
    while(tempEnum.hasMoreElements()){
      tempAction = (Action)(tempEnum.nextElement());
      tempAction.performAction(getRuleProperties(),
                          new Boolean(assessorReturns));
    }
  }
}

return assessorReturns;
  }
}
```

已知应用

本文作者曾参与的多个团队的多个项目中都使用了规则对象。包括电信领域（客户关系和计费），医药保健、保险、汽车、高等教育、销售、以及电子经纪等领域。

规则对象已经用于 Java 业务框架[Arsanjani99b]的实现中。

IBM San Francisco 用到的策略公共业务对象和模式(Policy Common Business Object and Pattern)，使用类似的概念。

规则对象被 Paul Corazza 应用于“If-Then-Else”框架中。

IBM WebSphere Application Server 企业版中，组件代理的 Managed Object 框架中实现了规则对象。

David Taylor 在它的对象杂志专栏中提到对业务规则相似的结构。

相关模式

对等模式

规则对象作为判定器和动作，属性（上下文）和结果的中间人，决定哪些判定器应该用于属性的判定，并可能将结果记录入一个判定结果中。如果判定成功了，相应的动作被调用，这可能会更新和改变属性的状态，它的结果也会被记录入一个动作结果中。

其他模式

规则对象使用了若干个基本设计模式。因此它可以被认为是一个复合模式（或一个合成模式），但是并不是所有的复合模式就是模式本身，复合模式仅仅是一个名字空间，标记一组在很多不同场合下重复使用的模式集。为了实现 Condition 参与者，我们建议使用一个 Assessor[Arsanjani98]，为实现 action（简单或复合的），使用 Command。复合本身是一个 Composite 模式。SimpleRule 使用一个 Strategy 来实现它的 Validator 参与者。规则集群有一个 Builder，它使用一个 Abstract Factory 来创建规则、条件和动作的单个实例。

Assessor 实际上是 Command 的一种特殊形式，在多种环境中可以发现，它并非是执行一个命令，而是有一个 assess()方法，返回一个布尔值（在简单判定器的情况下）或者返回一个合成物（在复合判定器的情况下）。判定器还可以进一步实现成 Interpreter 模式，如需要判定一个“规则串”的合法性的时候，即一个字符串包含一个“规则语言”的句子。

规则对象和 Visitor 相联，在很大程度上共享了下面的适用性：很多不同且无关的操作需要在对象结构中被执行，你可能不想让这些操作"污染"你的类，Visitor 让你将相关的操作放在一起定义在一个类中。当对象结构被很多应用共享时，使用 Visitor 就可以让只有使用特定操作的应用才拥有那些操作。

另外，判定器还可以使用在面向语法编程（Grammar-Oriented Programming [Arsanjani89]）的环境中，它的域分析确定了一种域语言。这种域语言以域语法来进行描述。域对象的交互完全由域语法来描述。触发协作、触发域语法和消息的用例以输入流的形式传入到解析器中，这个解析器负责解释和分解语法。对象的 manner 以元模型来描述并表示为语法。

参考文献

[**Arsanjani99**;a] Ali Arsanjani. "Service Provider: A Domain Pattern and Its Business Framework Implementation," presented to PloP '99. http://st-www.cs.uiuc.edu/~plop/plop99/proceedings/Arsanjani/provider3.pdf

[**Arsanjani99**;b] Ali Arsanjani. "Analysis, Design, and Implementation of Distributed Java Business Frameworks Using Domain Patterns" in Proceedings of Technology of Object-oriented Languages and Systems 30, *IEEE Computer Society Press* 1999, pp. 490-500.

[**Arsanjani89**] Concepts of Grammar-Oriented Programming, Azad University Technical Report, 1989.

[**Corazza**] Paul Corazza. Using the if-then-else framework, Part 1: Code maintainable branching logic with the if-then-else framework. Available at: http://www.javaworld.com/javaworld/jw-03-2000/jw-0324-ifthenelse.html

[Fow96] Martin Fowler. *Analysis Patterns.* Reading, MA: Addison-Wesley, 1996.

[**GHJV95**] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Reading, MA: Addison-Wesley, 1995.

[MRB98] Robert Martin, Dirk Riehle, and Frank Buschmann (eds.). *Pattern Languages of Program Design 3.* Reading, MA: Addison-Wesley, 1998.

[Odell96] James Odell and James Martin, *Object-oriented Methods: Pragmatic Considerations*. Prentice-Hall, 1996.

[Riehle98] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. "Role Object." In *Proceedings of the 1997 Conference on Pattern Languages of Programs* (PLoP '97). Technical Report WUCS-97-34. Washington University Dept. of Computer Science, 1997. Paper 2.1, 10 pages.

[VCK96] John M. Vlissides, James O. Coplien, and Norman L. Kerth (eds.). *Pattern Languages of Program Design 2.* Reading, MA: Addison-Wesley, 1996.

附录 A：规则对象：一种用于可适应、可伸缩规则设计和构建的模式语言（管理）

下表是本模式语言的模式的总结，为每种模式提供初始定义和上下文环境，下一节列出此模式语言的模式图，表示各个模式之间的关系，并给出转换标准（从一种模式到另一个模式），以及在转换前和转换后所遇到的问题。

ROPL 模式

1、规则对象－为业务过程提供扩展性和适应性，不会因干扰式的修改而影响，规则可以插拔使用。
2、Assessor－基于一组输入的属性进行条件求值判断，记录求值结果。
3、Action－在条件判定后执行动作，记录结果并更新属性以及相关协作对象的状态。
4、Rule Cluster－组件化规则对象的合成定义和应用，通过一个规则应用策略的定义优化规则应用。
5、Rules have State － 维护规则检查和应用之间的状态
6、Rules are Tracked － 跟踪历史、修改和条件/动作对。
7、Document Rules as Patterns－把规则捕捉成模式，跟踪报告问题解决方法和结果的原因
8、Rule Object Repository－把规则集中在公司知识库中
9、Rules Access Rights－经理应该能够创建规则，提供访问权限，以控制不必要或滥用的规则。
10、Rules Change Process － 新规则影响旧过程
11、Clusters Have Manners－协作对象的集群具有管理对象行为的规律以及这些规律的元数据。
12、Rules as First-class Constructs － 基于对象“manner”进行分析和设计
13、Rules as Production Rules 作为应用域语法的产品规则－面向语法的对象设计；为某种域设计一个域语言，使用解析器来实现，接收来自域中的应用程序输入。
14、Persistent Rules－以数据方式处理子类和对象的增殖。
15、Hash and Cache－当子类和对象数目增加很多时，提供高效、快速的访问。
16、Remedy Rule Proliferation－处理对象增殖综合症。
17、Rules Evole－为业务拓展进行规则演进
18、Rule Change Imapct Architecture－信息系统构架、功能和非功能性需求受规则改变影响。

模式语言图

规则对象模式语言能够简要地用以下方式来表述，接着我们来看看每个模式之间地关系，以及如何转换到另一个模式。

注意有些模式表面上是小型模式语言：规则对象由 Validator、简单规则对象、Assessor、Action、ErrorResult、Property、复合规则对象和 MediationStrategy 组成。（表示用什么算法来依次应用规则、条件或动作，这可以是简单的循环或更复杂的算法，如 Rete 算法，缺省情况是简单循环。它将遍历规则列表，并依次应用。你也许想有一个加权的 vector 或哈希表，可以有优先顺序地应用规则的判定器或动作，它们可以是集合结构或者是合成结构）

你可以从 ROPL 模式的不同部分入手，以你自己的方式应用模式来解决问题域的问题，对我们的例子，这里有若干用例，描述本模式语言中模式应用的过程和解决引起问题的先决条件。

使用实例 1：你想构建一个保险应用程序，需要实现作为需求规范中特定部分的业务规则。

使用实例 2：组织想整理它的业务规则，你正收集业务规则并将它们记录成模式，创建一个公司规则库以存放规则。

使用实例 3：你有一个公司规则库，并想配置成通用库。为经理和需要访问库中规则的开发人员分配规则访问权限（Rule Access Right）。当管理变化时，你对规则变化进行日志记录，以满足市场和操作需要。这个库影响着业务如何进行，所以新规则改变旧业务过程（Rules Change Business Process）。当对规则进行改变，架构也受到影响，也许规则集中在一个中间层而不是分散和重复在若干中间层中，如数据库触发器、GUI 等。

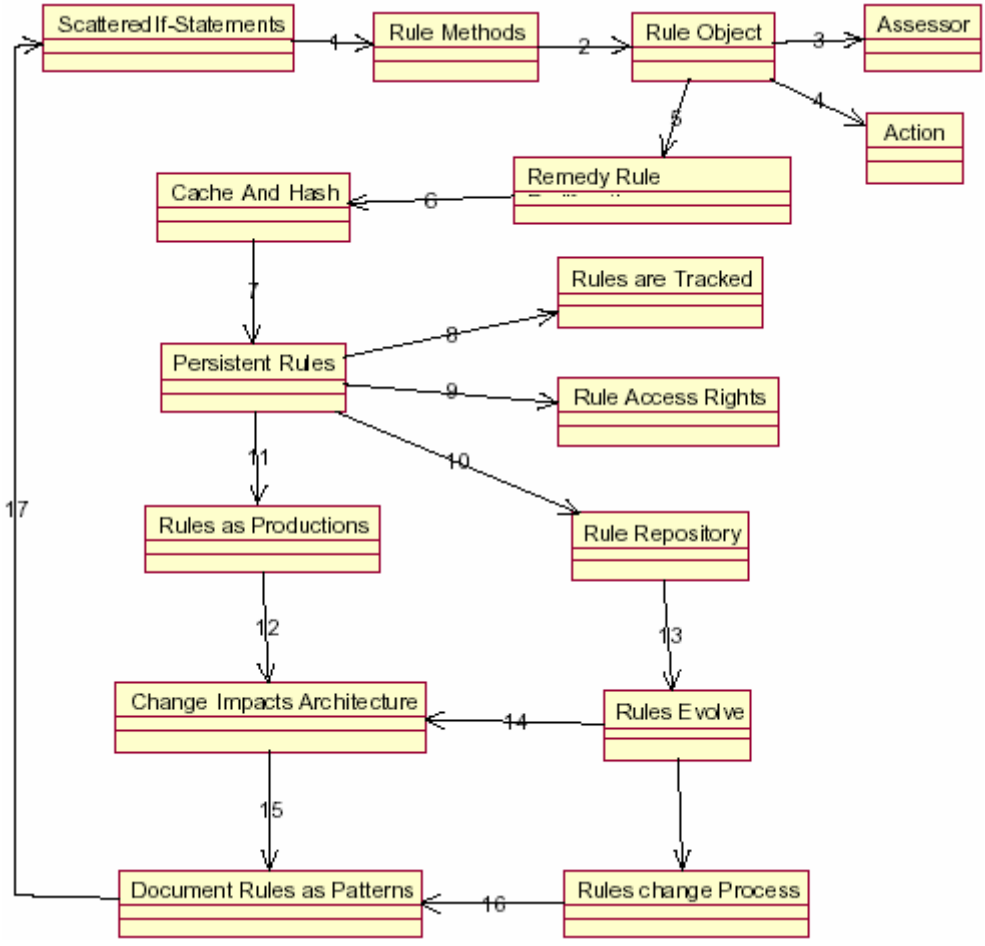
使用实例 4：在规则对象开发过程中，规则演进（Rule Evolve）和改变可以通过创建新的 Assessor 和 Action，或重用现有对象进行重组来反映新规则和过程。Property 被创建成判定器的原始输入，Action 可能会更新 Property，产生一个 Result 和 ErrorLog 向用户报告，或者记录到持久存储中。

使用实例 5：当规则开始增殖，我们通过将相关规则封装在规则集群(Rule Cluster)中，并用简单规则组成复合规则，开始重用 Assessor 和 Action，我们可以把它们哈希和缓冲（Hash and Cache）在内存和一个中间层中，以优化性能。当它们增殖时，我们也许想持久化 Assessor、Action 和 Rule、Property 和 ErrorResult 等，于是创建了持久规则对象（Persistent Rule Object）。

使用实例 6：在使用规则对象的过程中，我们也许需要跟踪规则状态（Track Rule State），这可以通过将它们作为 Momento 或其他机制来实现，也许一个简单的静态 AssessorResult 就可以做到。

使用实例 7：当我们进行更多需求分析并创建更多的规则对象后，我们发现可以有协作的类形成集群共同工作完成某个业务目标。这些业务对象组成了集群（Cluster）。这些 Cluster 让规则管理它们的交互，我们称之为它们的“Manner”，于是 Claster 和业务对象都具有 Manner。

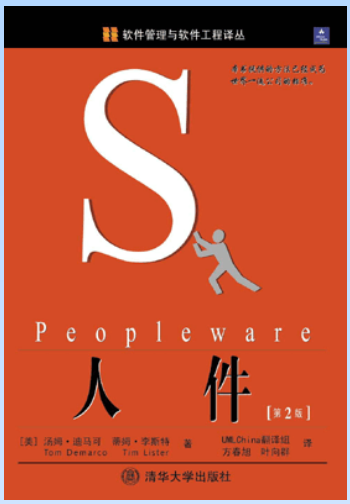
把上面的使用实例记在脑中，下面的模式语言图将帮助你在语言中选择路径：



图：规则对象模式语言图

序号	转换标准、前提
1	复杂性、可适应性、可合成性、组织结构
2	规则频繁变化的需要；设计、规则、条件和动作的重用；当复杂度增加时的维护问题；伸缩性。
3	使用；可互换的条件需要
4	使用；动作的可互换、重用和重现，如错误信息、数据库更新等。
5	子类和对象增殖
6	大量增殖（对象增殖综合症）；对对象更快速地访问；需要管理和维护对象资源。
7	更大量地增殖；更快地访问；将对象当作数据，数据库可以很好地处理它
8	需要跟踪变化并报告
9	很多人访问，需要被控制；法律和安全目的；稳定性和控制
10	公司规则的公共入口；有组织、可浏览；开发团队需要对新规则变化更新的参考框架；动态，交叉域重用；
11	多域重用；产品线架构；创建一个域语法；使用面向语法的对象设计[Arsanjani90]
12	架构和业务过程由规则驱动
13	改变规则和业务；需要保持业务，维护市场份额和受益；业务促进、新服务、新供应；新法律；违规等。
14	在功能性和非功能性需求（可用性、安全性、性能、持久性、伸缩性等）方面，规则变化影响架构
15	公司需要知道规则背后的原理，发布这个规则是为了解决什么需求、什么原因，这些都需要文档记载下来；这些规则如何解决问题；会导致什么样的结果。
16	<同上>
17	从最适当的设计或实现机制开始实现规则

老板，别把开发人员当成牲口--《人件》已经出版！



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>>

基于设计模式的网站设计

Francisco Montero 等 著，lanmobj 译

查看评论

摘要

本文继续收集网站设计方面的模式。传统意义上，网站主要是作为信息交换的媒介，因此绝大多数网站被设计得像书一样，允许人们从一个页面跳到另一个页面。在开发网站时，应该考虑可用性准则[Nie00]。本文中这组模式就是基于可用性准则的。

前言

由于使用超文本和多媒体来提供各类资源，WWW已迅速成为了英特网上占主导地位的工具。网站上的每一块内容都应该有它的实用价值。用户浏览一个网站，要么是希望能执行一项特定的功能，要么是希望获取一则特定的信息。我们设计的网站，应该尽量让用户快捷和方便地找到他们想要的东西，否则，他们会离开我们的站点到另一个站点上去找。网站是新的媒介，需要新的设计方法[Nie00]。

就像在设计其它的用户交互界面一样，我们应该首先弄清楚：用户是谁？任务是什么？但是回答上述问题，并不容易。任何人都可能访问我们的站点，网站的访问者中可能有小孩，年长者，老人，甚至可能有残疾人。所有的信息资源对他们都是可访问的[Con01]，任何人都能毫不费力地在浏览它们。无论你在做什么，都应该是以用户为中心的。在网站设计的整个过程中，时刻要考虑到用户的需要。不要在网站建成后，测试和修复网站时才考虑可用性问题。如果那样的话，很难有效率，也不可能产生好的结果。

Shneiderman [Shn98]说过 “要想使用经验，试验以及假设检验来理清这些内容至少要花上十年时间”，同时他警告道 “由于缺少经验数据来验证和锐化你的洞察力，一些指南总是起了误导作用”。然而，市面上还是出版了许多网站设计的指南。许多指南能够帮助我们提高网站设计的水平。大多数给网站设计者的建议靠的不是研究，而是直觉。它们依赖于设计者的经验。一般而言，界面设计经验通常以指南的形式来收集和整理，但也可以用模式。

模式语言的概念是由Christopher Alexander和他的同事们在建筑和城市设计中提出的[Ale77, Ale79]。简单地讲，模式语言就是一个不同层次上的模式网络；每一个模式都嵌在一个具体的原型里，并且支持相关的更高层次上的模式，同时由较低层次上的相关模式来支持。模式语言的目的就是捕获在它的上下文环境中的模式，并且提供一种机制来理解非本地设计决策的结果[Eri97]。

一种网站设计模式语言

这种模式语言，不同于某种说明性的模式语言或指南，它描述的是如何能够得到可用性的设计结果。使用这种语言描述的模式可以按层次级别分为三组。

网站级，人们期望网站能够提供信息，具有交互功能而且能够满足需求。网站还应该看着比较舒服，能提供快速下载，界面友好，易于使用和浏览，如果能够呈现出专业风格就更好了。网站是由一系列的文档，图像和其它文件构成。这些通常存放在因特网上的某个特定的位置。我们能够找到这些网页，是因为它们都有一个地址。

网页级，网页是指一个指定网站上的某个页面。对于一个网站而言，主页显得特别重要，因为它标识了整个网站。

修饰级，在一个网页上，通常有些不同的元素。这些元素为实现指定功能提供了有用的支持，提高了网站的可用性。

这种三级分类法源于[Ale77]。Alexander的模式语言中包含的253个模式，被分为3大类，城镇，建筑物和结构。最初的94个模式都收在“城镇”这一条目下。不过，这些模式覆盖的范围远远超过了单纯的城镇规划，而且每个模式都是以一个全景视角来审视设计者的职责开始。深入对区域和建筑群的研究，Alexander调查了构成城市景观的建筑物，提出了100多种模式，这100多种模式集中解释了什么构成了一个好的建筑物设计。Alexander确定了建造建筑物的合理的方式后，停止了对模式的收集整理。在他的模式语言的最后章节中加以叙述的这些模式，表明在整个设计过程中是多么需要慎思和创新。Alexander的一个主要的目标是在都市，城镇和建筑物的规划设计中，提高一种难以名状的品质。Alexander想创造一种适于人们的结构，并且这种结构能够对提高人们的舒适度和生活品质有积极的影响。当我们开发一个网站时，这同样是我们的目标。我们希望设计出高质量的网站，但高质量又指的是什么呢？质量可能指可用性，易懂性，易学性，易操作性，适应性和可访问性等等。我们认为质量应该是上面各种性能的综合。

我们将通过一个特别的例子来介绍将要给出的网站模式语言。这里收集的所有模式如图1所示，在本节的末尾处使用了3个表来做总结，每个表中描述一个层次上的模式。

每个模式中提出一个问题，然后在给定的上下文环境和背景下，给出解决方案[Mes94]。问题是与用户需求相联系的，解决方案则着力寻找提高网站可用性的方法。可用性[ISO/IEC 9126-1]是指易学性，易懂性，易操作性。这些性能的提高可以通过提供导航，功能性，控件，语言，反馈，一致性，避错，以及良好的可视性。

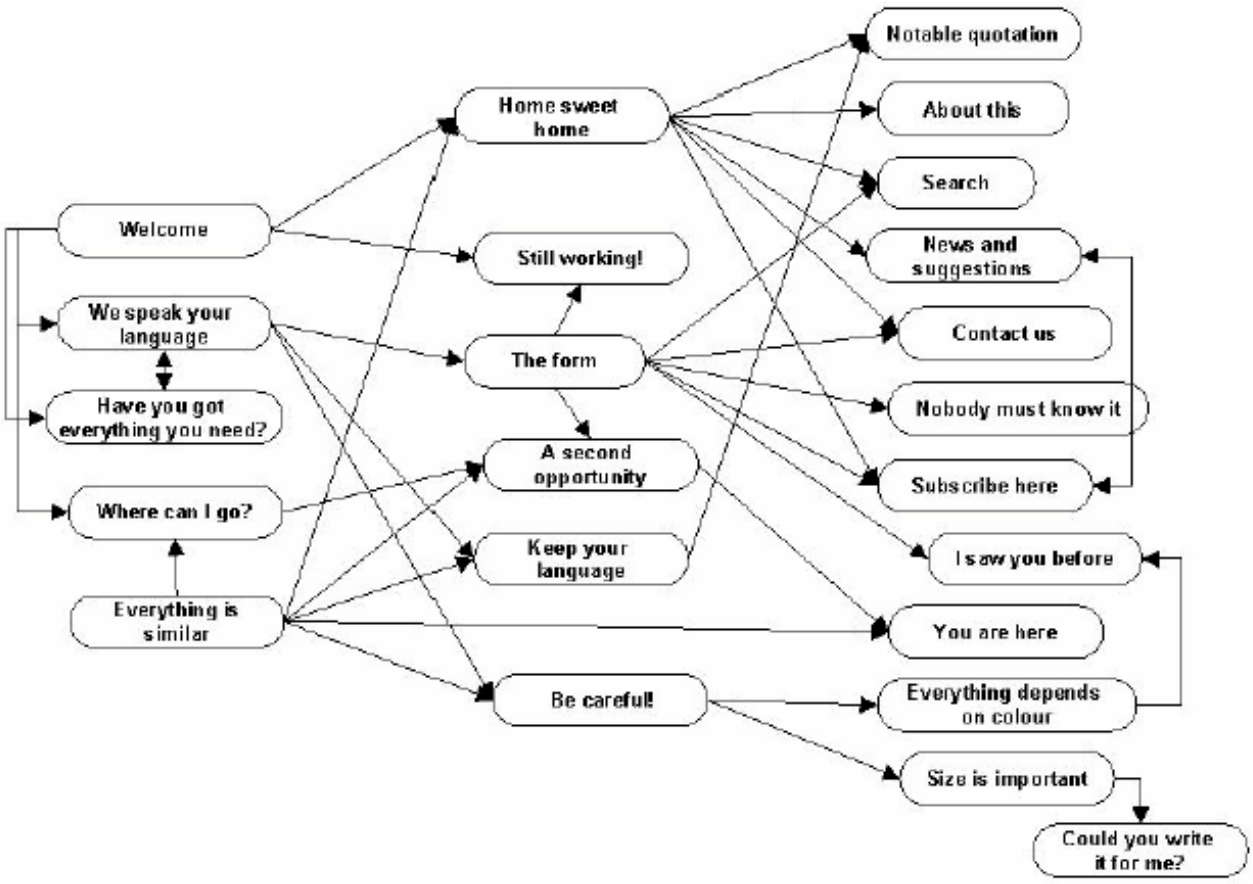


图 1. 建议的模式语言

图1显示了建议的模式语言。它具有网络结构，左边一栏中列出的是网站级的模式，中间一栏列出的是网页级的模式，在最右边一栏中列出的是修饰级模式。

接下来的表中概括了这种模式语言中涉及到的各个模式，以便参考。这些模式可以在方法学层面上进行集成，并用来开发类似IDEAS [Loz01]的用户界面。它们位于需求管理层次，可以用在基于可用性的迭代式生命周期的早期。使用这些模式后，可以让大家参与到设计中来，在可用性准则下进行评价，使围绕着网站开发的利益攸关者间的讨论更为容易。

网站级模式		
问题	解决方案	模式名
用户如何知道他在哪里？	设置一个接待点，让用户能够得到当前网站的信息	欢迎（Welcome）
用户如何知道他能到哪儿去，以及那里有什么？	网站必须提供这样的需求机制，依靠这种机制用户可以从一个地方换到另一个地方	能到哪里去（Where can I go?）
用户怎样才能有效使用一个网站，并且自主访问相关信息	使用用户的语言“为所有人设计”	我们使用与你一样的语言（We speak your language）
用户怎样知道他在哪里？用户怎样知道他是否在访问同一个网站？	网站设计时应注意使用相同的标准：颜色，字体，导航栏，布局	所有的东西都相似（Everything is similar）
用户怎样知道他是否还需要网站提供的其他信息？	应该告诉用户访问该网站能提供用户需要的什么东西？	你是否已经得到了你想要的所有东西？（Have you got everything you need?）

网页级模式		
问题	解决方案	模式名
用户怎样知道他在哪里？	提供一个检查点，让用户感觉像家一样	主页（Home sweet home）
用户怎样才能以一种简单而恰当的方式来访问网页的内容？	让你的语言文明且不具有伤害性	注意你的语言（Keep your language）
用户怎样知道他的操作什么时候才结束，他当前的状态？	显示一些用户状态信息，告诉用户还有多久才能结束当前操作	正在运行中（Still working）
用户怎样才能以他习惯的节奏来访问网站？	提供一个返回路径	再给一次机会（A second opportunity）
怎样才能让用户提供预先格式化的信息？	采用填空的方式，并且告诉用户应该填什么样的内容	使用表单（The form）
怎样才能让用户访问网站时，不被打断，也不会被弄得晕头转向	以合理的方式使设计适合于不同的技术和知识层次的所有人，	小心（Be careful!）

修饰级模式		
问题	解决方案	模式名
用户怎样知道网站所有者的主要特征？	使用标语，简明扼要地给出网站或公司的性质	显著标语（Notable quotation）
用户怎样得到信息的打印稿	提供一个可以直接打印的版本	你能帮我把它写下来吗（Could you write it for me?）
用户怎样访问附加的和定期的信息	提供用户在线订阅处	在此订阅（Subscribe here）
用户怎样才能得到产品和文档的其它信息	在主页上使用一个“联系我们”的链接	联系我们（Contact us）
用户怎样才能查找到特定信息	提供一个搜索引擎	搜索（Search）
用户怎样知道他曾经访问过哪些地方？	使用一种方法来维护Web上的状态变量，比如使用*cookies*	我见过（I saw you before）
用户怎样以合适的方式访问网站的内容？	使用合适的颜色	完全使用颜色来区分（Everything depends on color...）
用户怎样以合适的方式访问网站的内容？	*按需提供*	大小很重要（Size is important）
怎样以安全的方式提供私密信息	*在网站上实现安全性*	没有人知道（Nobody must know it!）
用户怎样知道网站上有哪些新闻	*在主页包含一个新闻和建议部分*	新闻和建议(News and suggestions)
用户怎样知道他在那里	*在网站上使用位置引用*	你在此处(You are here)
网站的所有者是谁？	*包含一个“关于我们”的链接*	关于(About this)

在接下来的章节中，我们将在实例中描述上面的部分模式。实例中将使用这些模式来解决创建具有高可用性的网站中遇到的一些问题。在实例中，我们将比这一节更多地考察与模式相关的细节。

在介绍每一个模式时，最后一部分“实例和实现细节”，提供了相关的参考(urls)和简短的注意事项，来帮助我们如何在网上浏览时识别这种模式。

实例：使用网站模式语言

假设你被提升到了某个机构，一个业务或者是研究小组。创建一个网站，让用户可以找到该机构提供的服务，产品，新闻，可能是一件有意义的事情。

你负责最初的设计，而且你知道拥有一个好的网站是非常重要的，但是你并没有网站设计方面的丰富经验。虽然在网站设计领域汇集了指南，经验和知识，但在设计中应用它们并不容易，并且在某些场合，它们还相互冲突。无论如何，语言是最有力的交流工具，虽然语言中有单词，句子，以及单词和句子之间的结构关系。另一方面，模式可以用来记录设计经验。一种模式语言在两种意义上是有用的：首先，作为一种将经验形成文档的工具，其次，作为一种在项目的利益攸关者（Stakeholder）（包括最终用户）之间交流的工具。

怎样使用这个模式集合（模式语言）？

1. 阅读模式的概要列表
2. 查看列表，寻找那些最能描述项目的全景或者适合你当前急待解决的问题的模式
3. 阅读最初始的模式，将所有排在前面的模式作上标记，忽略所有排在后面的模式；
4. 对每一个模式，仅标记上相关的排在前面的模式。
5. 持续操作第4步，直到你标记出了你的项目中所需使用的所有模式。
6. 调整顺序，在你没有找到相对应的模式的地方添加你自己的素材。
7. 如果对于某个模式，你自己有更贴切的版本，就将该模式替换成你自己的版本。

举个例子，在阅读过模式列表之后，我们从中选择一些模式，比方Welcome, Home sweet home, Notable quotation About this, Search, News and suggestions, Contact us, or Subscribe here。这些模式之间的关系如表1所示，它们处于不同的层次级别上。接下来，我们更详细地探讨这些模式。

欢迎（Welcome）

动机：

用户到达某个站点，就象他到达某个都市，城镇或是重要建筑物一样，需要知道他在哪里，能够做什么，以及通过访问网站他能得到什么。

问题陈述：

用户怎样知道他在哪里？用户怎样知道他能到哪儿去？站点的管理者是谁？这个站点的目的是什么？

约束:

用户想知道他在哪儿

用户想知道他接下来能到哪里去

复杂的网站往往会弄得用户晕头转向

熟悉网站内容结构的用户能够直接跳转到他想去的地方

解决方法：

设置一个接待点，供用户找到关于站点的信息。从这个接待点，用户能够直接进入主页(Home sweet home)。在这里还将收集用户所使用的语言和显示器尺寸等信息，这些用户信息将用来给用户提供合适的站点服务(We speak your language)。这一模式的缺点在于用户需要知道访问站点的最佳条件 (Have you got everything you need?)。用户在该页面上可以知道该网站的内容(About this)和网站的所有者(Contact us)。在多数场合中，接待点就是主页。

结果:

在导航，功能性和反馈方面进行改进

举例与实现细节:

http://www.aosa.es, http://www.alanismorissette.com

上面列出的这些站点，和网络上的许多其它站点，都设有一个初始页面来迎接用户。这些页面都有一些主要特征，如页面下载时间短，用户可以自定义使用的语言，提供网站所涉及的诸如人物，事件，时间和地点之类的基本信息。

主页（Home Sweet Home）

动机:

网站可能会被随意地访问，但通常这些网站都会有一个参考点。用户到达某个站点，就象他到达某个都市，城镇或是重要建筑物一样，需要知道他在哪里，能够做什么，以及通过访问网站他能得到什么。主页是网站的重要组成部分。有关网站设计到的人物，事件，时间和地点之类的基本问题，都能在这里找到答案。

问题陈述:

用户怎样知道他在哪里？用户怎样知道他能到哪儿去？站点的管理者是谁？这个站点的目的是什么？

约束:

用户想知道他在哪儿

用户想知道他接下来能到哪里去

复杂的网站往往会弄得用户晕头转向

熟悉网站的内容结构的用户能够直接跳转到他想去的地方

解决方案：:

*提供一个检测点，让用户感觉像在家一样。*当用户在网站中弄得晕头转向时，他可以退回到主页中。主页的布局中，将重要的信息放在顶部（News and suggestions），此外还设有logos (Notable quotation)，搜索的方法(Search)，和联系信息(Contact us, About this, Subscribe here)。

结果:

在功能性，控件和导航方面进行改进

举例与实现细节:

http://www.apple.com, http://www.ieee.org

任何网站都有主页。主页很有特色。从主页上可以直接链接到网站的不同部分，如新闻区，联系方式，关于我们。网站的每一个页面上都有主页引用 (A second opportunity)。

显著引用（Notable quotation）

动机：

当你设计一个网站时，应该提供关于网站用途的信息。

问题陈述：

用户怎样知道网站的拥有者的主要特征？

约束:

用户很匆忙

用户不认真阅读网页，他们随便翻翻这些网页

解决方法：

使用标语简明扼要地给出这个站点或者公司是干什么的。在一个窗口中给出该站点的简单描述。

结果:

提供在可视性，功能型和反馈方面的改进。

举例与实现细节:

http://www.coolhomepages.com, http://www.bbva.es

这些网站使用图像和标语来实现这个模式。一条标语就是一个短语，它给出了有关网站的所有者，以及该网站是干什么的信息。

下面的元素可以构成有效的标语：

主体＋受众＋组织

关于（About this）

动机：

无论公司是大还是小，所有这些公司的商业性网站都需要提供清楚的方法来供用户查找关于公司的信息。

问题陈述：

用户怎样知道网站的管理者是谁？

约束:

人们想知道他们是在和谁做生意

获取公司信息可能是用户访问该网站的唯一原因

许多用户想知道是谁在提供这些服务

解决方法：

使用链接指向“关于我们”，在那里给出关于网站所有者的概述，还可以给出关于产品，服务，公司评价，业务专题，管理团队等等相关细节的链接。

结果:

提供功能性和反馈方面的改进

举例与实现细节:

http://www.sunspot.net, http://www.ireland.com

这个模式的实现是通过增加一个页面或者一个部分来实现的，在这些地方给出所有者的有关信息，然后在主页中设置指向这些地方的链接。

搜索（Search）

动机：

搜索功能是主页（Home sweet home）上最重要的元素，让用户能够很容易地找到它，并且方便地使用是很关键的。

问题陈述：

用户怎样才能找到指定的信息？

约束:

用户想知道是否能从该网站上搜索到有关信息。

用户并不想阅读整个网站，他只是想随便看看。

解决方法：

提供一个搜索引擎。在主页上给用户一个输入框来输入查询条件，而不仅仅是给出一个指向搜索页面的链接[Nie02]。在主页上的搜索应该默认为搜索整个站点[see Wel01]。

结果:

提供功能性和控件方面的改进

举例与实现细节:

http://www.paginasamarillas.es, http://www.microsoft.com

这个模式是通过提供一个搜索的表单来实现的。搜索的表单是搜索引擎的用户界面。它既可以简单得只有一个文本框和一个按钮，也可以是整个页面，并将相关的链接添加到导航条中。可以为用户提供高级搜索功能。在高级搜索页中，可以提供短语选项，多重值域，特殊集合，以及时间范围，以便搜索能够更精确。

新闻和建议（News and suggestions）

动机:

用户想知道网站上是否有新的主要内容。用户接受建议并且想知道优惠和推广活动。

问题陈述:

用户怎样才知道网站上有什么新闻？

约束:

用户并不想阅读整个网站，他只是想随便看看。

用户很匆忙。

解决方法:

在主页上包含新闻和建议部分，用户可以快速访问到由网站提供的新服务。

结果:

提供功能性和导航方面的改进

举例与实现细节:

http://www.microsoft.com, http://www.terra.es

这个模式是通过把最新的新闻和建议放在主页的显著位置来实现的(Subscribe here)。

联系我们（Contact us）

动机:

所有的商业网站都需要提供一个联系网站所有者的简明方法。

问题:

用户怎样才能得到关于产品和文档的其它信息。

约束:

人们想知道他们是在和谁做生意

获取公司信息可能是用户访问该网站的唯一原因

许多用户想知道是谁在提供这些服务

解决方案:

使用链接指向“联系我们”，在那里给出你们公司的联系方式(About this)。

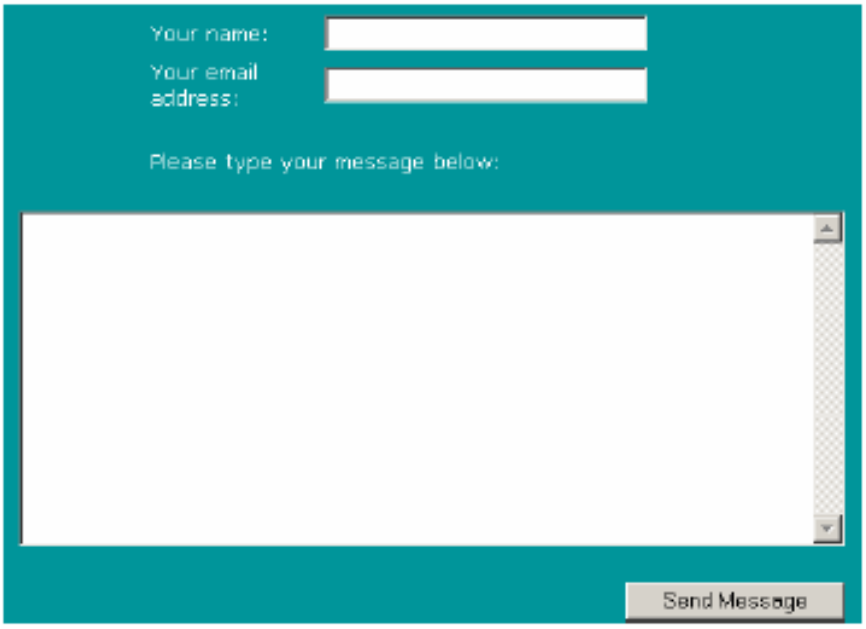
结果:

提供反馈方面的改进

举例与实现细节:

http://www.intel.com, http://www.lucent.com

这个模式的实现是通过增加一个页面或者一个部分来给出联系方式。很多情况下，这类信息会放在该站点所有页面的底部。也有其它一些情况，会给出一个表单，表单中提供输入文本的区域或者是文本框供用户输入他的邮件地址和其它意见。



在此订阅（Subscribe here）

动机:

用户想天天访问某个网站，他们希望在网站有新的产品或是新闻的时候，能得到通知。

问题:

用户怎样才能访问附加的和定期的信息？

约束:

用户很忙

用户希望得到通知

解决方案:

*提供一种方法，用户提供电子邮件地址，就能在线订阅。网站的所有者可以给登记的用户发送关于*新闻和公告的*信息*(News and suggestions)。

结果:

提供反馈方面的改进

举例与实现细节:

http://www.prenhall.com, http://www.sun.es

这个模式的实现是提供一个简单的表单，供用户在上面填写邮件地址。在其它一些场合，可以会要求用户提供更多的有关用户基本情况和喜好的信息，以便给用户提供个性化的信息。此外要注意提供取消订阅的功能选项。

这时我们有了一个开发网站的出发点。使用一种模式语言来编写模式，包括对其它低序模式的引用。如果此时，我们的初始设计者确定了从用户处获取信息的需求，就要阅读The form pattern，以及与此相关的其它模式Still working, Search, Contact us, Subscribe here , Nobody must know it and I saw you before。接下来，我们将解释其中的部分低级模式。

表单（The form）

动机:

用户需要提供有关的信息，通常这些信息是针对问题的简单回答。

问题:

用户怎样才能提供格式化的信息？

约束:

用户需要知道他需要提供什么类型的信息

一般来说，用户不喜欢以这种方式提供信息

必须说清楚什么是必须要填写的信息，什么是可选的

用户很忙

解决方案:

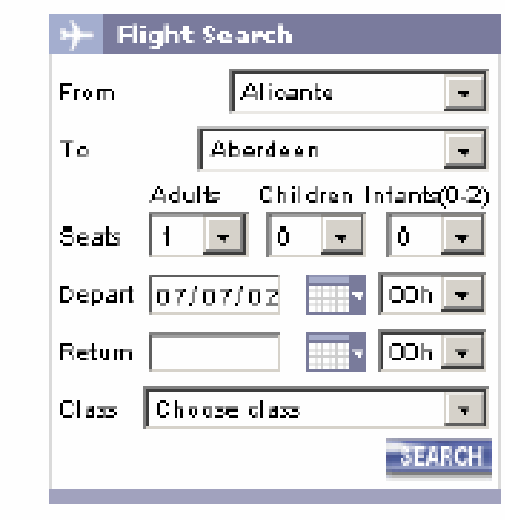
提供适当“表格”供用户来填写，表格中清楚而明白地提示应该填写什么样的内容[Tid98]。Search, Subscribe here, Contact us就是表单的例子。在某些情况下，一个表单可能占满整整一页。用户需要知道他的提交是否正确地进行了(Still working)。有时，我们需要获得一些用户的机密信息，同时必须提供附加的安全性（Nobody must know it!）

结果:

提供功能性方面的改进

举例与实现细节:

http://www.iomega.com, http://www.iberia.es 这些网页和类似它们的其它一些网页都实现了这个模式，设置了让用户提供信息的地方。作为文档的一部分，一个HTML的表单含有常规的内容，标记，被称为控件（检查框，单选按钮，菜单等）的一些特殊元素，以及在那些控件上的标签。用户通常修改这些控件（输入文本，选择菜单选项等）来完成表单，然后将这个表单提交给进行相关处理的代理机构（比如，Web应用服务器，邮件服务器等）。



这个模式的特殊例子是Contact us, Subscribe here。

正在运行中（Still Working）

动机:

从网站上，用户可以下载信息，图片，文件或者是应用程序，但是这种下载会花去大量的时间，造成明显的延迟，也许这种下载应该通过不同的方式来完成。

问题:

用户怎样才能知道他的操作什么时候结束，以及他当前的状态？

约束:

用户希望知道处理过程还要多久才能结束

用户希望知道处理过程有多块，尤其当处理速度变动时。

有时要明确指出处理过程要进行多久几乎是不可能的。

解决方案:

给用户某种类型的状态信息，指示实际上需要多长的时间才能够完成这个过程。应该获得用户要下载的图像，文件或其它东西的大小，这样用户可以知道整个下载过程需要花多少时间。图像和文本应该按需下载（Size is important）

结果:

提供功能性，反馈和避错方面的改进

举例与实现细节:

http://www.google.com, http://www.acrobat.com

许多网页需要装载插件以获得良好的可视性t，使用进度条可以提供这样的信息。有时在一个网站上执行完成一个任务需要花去一些时间。一个具有可用性的网页会指示给用户这个任务会执行多长时间，以及已经完成了多少工作。如果你不知道或者不想知道，任务完成了多少，你就能使用一个游标或动画来指示仍有一些工作在处理中。如果，另一方面，你想传达出任务完成了多少，那么你可以像这样来使用进度条(http://java.sun.com):

有时，你不能立即确定需要花多长的时间来完成一项耗时的操作，你可以将进度条设置成一种不确定的方式。使用这种模式，进度条以动画方式指示出工作正在进行。以Java的视感风格，不确定模式的进度条看起来应该象这样。

没有人知道（Nobody must know it!）

动机:

如果用户提供私密的信息，他具有保密的权利。通讯上的快速进步更强调在Internet上的安全需要。

问题:

怎样以安全的方式提供私密的因特网信息？

约束:

用户需要安全

用户并不想知道技术方面

解决方案:

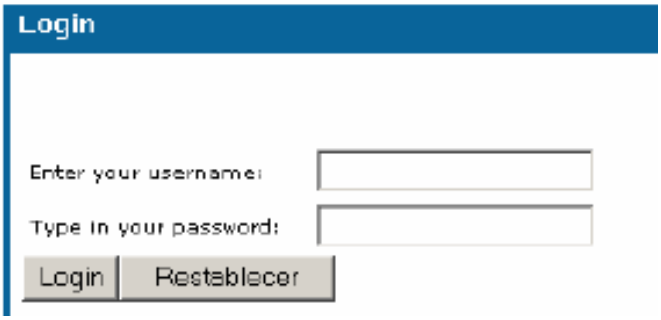
网站安全性的实现。用户要访问网站中的私密部分时，需要进行登记，但有时仅仅依靠登录和密码进行安全性控制是不够的。[see Yod98]。

解决方案:

提供反馈和控件方面的改进

举例与实现细节:

http://www.bankofamerica.com, http://www.cdnow.com 使用这个模式需要使用登录的表单。在登录的表单中，使用php或者asp询问用户的用户名和密码。但是，除非你的表单被放置在一个安全服务器上，而且php脚本开始运行，你才可以使用明文来传输这些信息，并且无需加密。



以前见过（I saw you before）

动机:

当用户返回网站时，他应该知道他已经访问过哪些地方，下载过哪些文档，以及该地方自上次访问以来是否有什么改变。

问题:

用户怎样知道他曾经到过什么地方？

约束:

用户不想浪费时间

用户想得到个性化的信息

解决方案:

使用类似cookies的方法，来维持WEB访问的状态变量。因为HTTP是一个非持久化的协议，所以除非服务器可以以某种方式标记访问者，否则要区分对网站的访问是不可能的。

结果:

提供反馈和避错方面的改进

举例与实现细节:

http://www.kinkos.com, http://www.americanairlines.com

这个模式使用cookies 来实现。Web cookies 只不过是当你访问网站时，放置在你的机器上的一些代表软件的位bits of software。通过Web cookies，可以在用户重新访问某网站时，让该网站识别出用户的机器。Cookies可以提供如下的一些好处，比如，你登录某个网站或者进行在线购买时，你是否注意到，当你重新返回时，无需再次进行签名认证。这时因为你的密码和ID都存放在你的机器的cookie里了。这样可以减轻用户的负担。

结束语

本论文首次提出了一种网站设计模式语言的方法。它的主要目的是收集整理在网站设计方面的经验，并提供一种供项目的利益攸关者进行交流的工具。这种模式语言区分三种层次：网站级，网页级和修饰级。这种贯串模式语言的原则可以帮助用户提高可用性。

网站模式与一些公共特征相关联，这些特征可以在许多网站上找到，也可以从其它不同的上下文中外推出来。用户需要知道他在哪里(Welcome)以及他能到哪里去(Where can I go?)。用户希望以一种合适的方式来访问网站(We speak your language [可参考 Lya00], Have you got everything you need?, Everything is similar)。网页模式引入与网页设计相关的设计模式。它们在网站设计中都是一些常用的，很为用户着想的特征功能。在这些层次结构中，主页是必需的(Home sweet home)。在某些场合中，用户使用填写表格(The form)的方式来提供信息，而且用户希望使用控件(Still working, A second opportunity)，希望以他自己的方式来访问网站(Keep your language, Be careful!)。修饰级模式引入了一个网站的装饰的特征功能，它们可以提高任何一个网站的通用可用性。它们恰当地使用了颜色(Everything depends on colour...)，文件大小(Size is important)，安全性(Nobody must know it!)，提供了链接位置(You are here, Contact us)和相关信息(I saw you before, News and suggestions, Subscribe here)。在web 设计中模式作为一种交流的工具和评价网站可用性的测度表，更适于产生以人为中心的设计，这一新的思想还将不断地进步。

致谢

本论文得到了西班牙CICYT TIC 2000-1673-C06-06 和 CICYT TIC 2000-1106-c02-02部分资助。

参考文献

[Ale77] Christopher Alexander. “A Pattern Language”, Oxford University Press, 1977.

[Ale79] Christopher Alexander. “The Timeless Way of Building”, Oxford University Press, 1979.

[Con01] Constantine Stephanidis, Anthony Savidis. “Universal Access in the Information Society: Methods, Tools and Interaction Technologies.” Springer-Verlang. 2001.

[Eri97] Thomas Erickson, “Supporting Interdisciplinary Design: Towards Pattern Languages for Workplaces”. 1997. http://www.pliant.org/personal/Tom_Erickson

[Loz01] María Lozano, Isidro Ramos, Pascual González. “User interface Specification and Modeling in an Object-Oriented Environment for Automatic Software Development”. IEEE 34th International Conference on TOOLS USA. 2000.

[Lya00] Fernando Lyardet, Gustavo Rossi. “Web Usability Patterns”. EuroPLoP, 2001. http://hillside.net/patterns/EuroPLoP2001/papers.html

[Mes94] Gerard Meszaros, Jim Doble, “A Pattern Language for Pattern Writing”, in Martin, Riehle, Buschmann, PLoP Design 3. Reading, Mass: Addison-Wesley, 1998.

[Nie00] Jakob Nielsen, “Designing Web Usability: The Practice of Simplicity”. New Riders Publishing. 2000.

[Nie02] Jakob Nielsen, Marie Tahir. “Homepage Usability: 50 Websites deconstructed”. New Riders. 2002.

[Shn98] Ben Shneiderman. “Designing the User Interface: Strategies for Effective Human-Computer Interaction”. Addison Wesley. 1998.

[Tid98] Jenifer Tidwell. “Common Ground: A Pattern Language for Human-Computer Interaction”. http://www.mit.edu/~jtidwell/. 1998/99

[Wel01] Martijn van Welie. Interaction design patterns. http://www.welie.com/. 2001

[Yod98] Joseph Yoder, Jeffrey Barcalow. “Arquitectural Patterns for Enabling Application Security”. PloP’97 D-4 book. 1998.

[Guides] Yale C/AIM WWW Style Manual http://info.med.yale.edu

Apple’s Web Design Guide http://applenet.apple.com

IBM Web Design Guidelines http://www.ibm.com/IBM/HCI/guidelines

Mary Evans. “Web Design: An Empiricist’s Guide”. University of Whasington, Seatle, Washington. 1998.

UMLChina 公开课

“UML 应用实作细节”

（2003 年 6 月 21-22 日，广州）

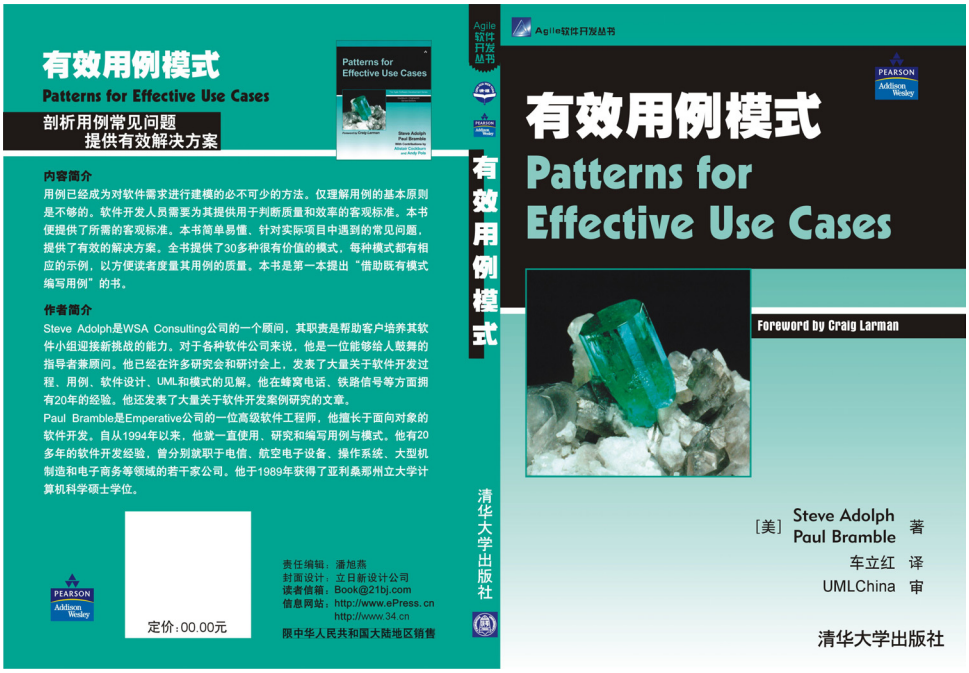
现在，UML、RUP、Rose 等概念已经传播得越来越广，书籍、资料也越来越多，决定在开发过程中使用 UML 的开发团队也越来越多。

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，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，非常适合中小团队。

详情请见>>

《有效用例模式》中译本样章（草稿）



审校者记

用例的概念在 1986 年由 Ivar Jacobson 正式提出之后被广泛接受，迅速发展。现在，“用例驱动”已经成为统一过程的基本原则。

Alistair Cockburn 等人在探索用例的本质和如何使用用例上，作出了大量的努力，使用例变得更加朴素、更加实用。本书可以看作是 Alistair Cockburn 的“Writing Effective Use Cases”的续集，它把用例开发过程中的要点组织成了 36 个模式，使读者更加容易理解。

UMLChina（http://www.umlchina.com）已经决定把本书作为 UMLChina 训练的指定教材。UMLChina 从 1999 年开始就已经在中国推广用例技术，迄今为止，实践和指导实践用例技术的项目已超过 20 个，我们深深感到用例技术带来的强烈变化，正如面向对象为分析设计所带来的，用例也是软件开发的一种“返璞归真”。

2003 年 5 月

第 8 章　编辑现有的用例

当时听起来像个好主意

您与Sitra坐在Wings Over the World的小会议室里。Sitra一直在与Yitka开发创建新客户帐户的需求。看起来进展不错，直到有一天Yitka把Sitra编写的用例删掉了一部分。现在，Sitra 请您帮助她修改用例。

Sitra: 当我开始与Yitka共同开发新的客户帐户特性时，我构建了这个小的UI原型，并和她一起检查了其中的每张表单。她对结果感到很满意，但当我详细编写完用例后，她把用例扔回给了我并说道："这很像小块的拼图碎片，这里没有故事"。我认为没有人能使她满意。

您从桌子上拿起用例看了一下它们的名称：打开帐户、填写客户个人资料、填写客户旅行偏好，保存客户帐户。

您：为了创建一个新的客户帐户，您好像为用户填写的每个用户界面表单都创建了一个用例。

Sitra: 我认为这样做很不错，因为我可以把用户界面表单展示给别人，然后展示驱动它的用例。我认为用例是在当表单填写错误时，处理会出现的所有不同错误的一个很好的方法。

您：对，很多人都会遇到这一问题。通常情况下，对用户来说，填写一个用户界面表单并不表示一个CompleteSingleGoal（5.1节）。此处的用例"打开帐户"实际上没有什么价值，除非还接着"填写客户个人资料"、"填写客户旅行偏好"，然后"保存客户帐户"。在这些用例中，没有任何一个用例可以独立存在，也没有任何一个用例有足够上下文使评审人员能够轻松理解。

Sitra: 我应该做什么？

您：与参与者有一个持续对话没什么错，为用例提供几个表单也没什么错。实际上这很好，因为用例为表单提供了上下文。将您编写的每个用例看作是有一个有用目标的真实用例的片断，如果喜欢的话也可以把它叫做小滴（droplet）。把这个新合并的用例称为"创建新客户"。我们有一个应用于此类情况的称为MergeDroplets（8.2节）的指导原则。

您注意到Sitra快速地计算了一下新用例的步骤。

Sitra: 如果把所有这些用例合并在一起的话，在主成功场景中差不多有20个步骤。

您：是的，所以我们可能必须对新用例进行编辑。难办的是大多数低级的步骤，因此如果它们仍然相关，那么我们就可以通过将它们移到更低级的用例中RedistributeTheWealth。

分析是一个发现的过程，其中会有许多不成功的开始和僵局。当时看上去不错的一个主意可能会在以后变得多余或过时。一些用例会过时，不能正确描述正在构建的系统，这是不可避免的。但这并不奇怪。首先，需求说明并不准确。系统需求本身反复无常，受不断发生的变化的影响，因为编写用例常常是一个揭示的过程。由于您发现了关于系统的更多潜在信息，因此，开始编写用例时有意义的思想可能会变得不再合理。其次，随着系统在无法预料的方向上日益成熟和演变，跨越多个开发周期的用例很可能会跟不上发展。结果是，随着对系统理解的加深，用例可能会无法维持下去。

用例可能会以几种方式退化。这些退化可能包含从描述系统的最初尝试中留下的过时信息。或者，它们可能会以一种不再支持系统愿景或业务情况的方式对系统进行划分。一些在以前代表了有用事件序列的概念，现在可能仅能说明某些不相关行为的一个片断。

相反，一些用例可能过大，或者包含太多的功能，描述了过多的操作。

您可以用几种方法来纠正这些问题。您可以决定现在仍然不是QuittingTime（3.5节），您需要增强甚至编写更多的用例。另一方面，您可能发现用例是比较完整的，只需要花费一些时间重新组织以进行改进。废弃的用例是最容易处理的，因为只需要通过丢弃它们来CleanHouse（8.3节）。

在探讨重构面向对象软件的书籍《*重构*》中，Martin Fowler（1999）把软件可能要求重构的标记叫做“坏气味（bad smell，一种不好的代码形式）”。在用例中应该一直注意的一个“坏气味”是用例过长。一个好用例的主成功场景通常为3～9步。包含9个以上步骤的主成功场景可能说明用例或者有多个目标，或者包含较低级的细节。在任何一种情况下，您都应该RedistributeTheWealth（8.1节），合理地将较大的用例划分为较小的、内聚的用例。

用例可能导致的另外一个坏气味是不能实现CompleteSingleGoal（5.1节），因此仅为主参与者提供了UserValuedTransaction（4.4节）的一个片断。“坏气味”通常来源于CRUD格式的用例或系统，在这种系统中，编写人员为每个用户界面表单创建了用例（Lilly 1999）。

本部分描述了3种将“坏气味”从现有用例中删除的模式。RedistributeTheWealth说的是拆分过分复杂用例，把它们的独立部分分成可以独立存在的更简单用例。任何一个动作都不是任意的，而是包含了一些规则的应用，所以得到的用例不仅内聚而且目的单一。最后，CleanHouse建议消除那些不提供价值，或不能帮助我们了解参与者如何实现目标的用例。

（插图一幅）

每天晚上9点，孩子们排队领取由城市慈善会堂提供的汤

8.1 RedistributeTheWealth

您正在发现系统的UserValuedTransactions（4.4节）。

过长的用例不实用而且难以使用，会分散用户的注意力，并使他们失去重点。

在60年代和70年代，许多公司都合并成了在各种不相关的业务范围中运作的集团。例如，加拿大太平洋集团就有铁路、航空客运、航运业务，并开发房地产。跨国石油公司Exxon甚至成立了一个计算机分部。集团业务的多样性是一种保护方法，用来保护股东利益免受某一具体市场在低迷期所造成影响。这些集团所信守的原则可以归结为一句谚语“不要把所有的鸡蛋放在一个篮子里”。

问题是集团的股价和收入很可能会落后于那些聚焦更窄的公司。通常问题的根源是，经理们很难为公司的各个部门制定一个统一的愿景。最终在90年代，大多数集团都分解为独立进行贸易的公司。在多数情况下股东资产的价值都提高了。

用例的目标对涉众必须清晰。 理想的用例是清晰、准确、明确的，因为它在开发人员和涉众之间形成了一个隐含的合同（甚至可能是一个法律合同）。用例必须描绘一个清晰的系统概貌，证明普通用户通常会如何以一种他们能够理解的语言来使用它。

添加新用例的代价是昂贵的。 新用例不会很快产生，通常需要几个人遵循一个特定的过程为之付出努力。必须有人仔细审查需求，或与客户谈话来理解特性的基本特征，然后，他们必须彻底识别所有可能的错误条件，以及分支。需要一个人或多个人组织该信息，并编写文本，同时，其他人还必须对其进行评审。提供新用例意味着要进行额外的开发工作，因为开发人员必须实现新特性。所有这些工作都需要投入时间和资金，因此仅编写那些提供可度量价值的用例是有意义的。

过多的细节可能会使用例难以阅读和理解。 每个用例都应该仅解决一个目标，并清晰地描述系统如何帮助参与者实现该目标（CompleteSingleGoal，5.1节）。如果用例试图超出这一范围，读者甚至编写人员都很可能会迷失在那些转移注意力的情节和添加了变化的情节中，忽略了重要的事情。描述超越其主要目标的行为的用例会模糊用例之间的边界，使用例重复甚至互相矛盾，并会使用例更加模糊且更难理解。

因此：

将冗长、难以处理的内容或过分复杂的扩展转移到它自己的用例中。

有效的用例将实现一个CompleteSingleGoal。尤其要提防冗长的用例。尽管用例本身的大小可能不是问

题，但一个长的主成功场景可能是预示用例解决了多个目标的“坏气味”。发现用例有多个目标时，将该用例划分为几个单独的用例，每个用例对应一个目标。如何重新组织取决于这些目标的级别以及它们与其他用例的关系。

将片断重新放到其他用例中。 如果这些额外目标的故事是另一个用例中的行为片断，那么，将它们合并到这个用例中。通过把这些片断放到适当的用例中进行MergeDroplets（8.2节），这样做的前提条件是不妨碍其他用例的CompleteSingleGoal。

创建一个新用例。 如果额外的故事描述了不仅单独存在，而且实现了参与者目标的行为，那么，为该行为创建一个新的用例。该方法需要的不仅仅是要求把额外的内容移到另一个文档中，并把它称为一个用例，因为结果必须满足用例的所有准则（CompleteSingleGoal，有一个VerbPhraseName，5.2节）。它还包括通过从最初用例中提取相关信息使新用例完整，以及识别新的分支场景。如果所得到的用例是紧密连接的，那么这可能预示着用例之间存在潜在的关系（PromotedAlternative，7.3节；InterruptsAsExtentions，7.2节；CommonSubBehavior，7.1节）。

创建一个新的较低层的用例。 如果有一些描述较低层行为的额外故事步骤，那么，通过创建具有它自己的CompleteSingleGoal的较低层用例，对用例步骤进行分级（LeveledSteps， 6.2节）。将较低级的步骤放到新的用例（EverUnfoldingStory，4.5节）中，并从最初的用例中引用较低层的用例（这就是“包含”关系）。

将多余的片断重新定位到补充说明中。 如果要包括多余的故事片断以帮助澄清用例描述，那么，将它们保存为Adornments（5.5节）。

示例

呼叫处理

用例8.1从服务提供商的角度对拨打电话进行了简要描述。

用例 8.1　用例的噩梦：具有多个目标的“处理正常呼叫”用例

拨打电话

主参与者：主叫方——拨打电话的人

被叫方——接听电话的人

次要参与者：交换网——处理电话间呼叫的设备

级别：用户目标

主成功场景

1. 主叫方摘机。
2. 主叫方拨号。
3. 交换网转换数字符号，连接主叫方和被叫方的电话，并发送振铃信号。
4. 被叫方应答（摘机）并与主叫方交谈。
5. 被叫方和主叫方挂断电话。
6. 交换网记录此次呼叫的计费信息。如果这是主叫方在当天拨打的第一个电话，那么，它就会初始化客户文件中的日使用报告条目。如果是本地电话，它还把它记录在服务提供商的日志中。如果是长途电话，它把条目记录在相应长途电话提供商的日志中。
7. 交换网切断连接，释放呼叫要求的所有组件。
8. 交换网释放了在呼叫中使用的所有组件后，用例结束。

该用例的长度是可接受的，但它违反了CompleteSingleGoal（5.1节），因为它描述了两个单独的行为：拨打电话和计费。一旦发现这种情况，可以创建一个新用例，也可以创建一个新的子用例，甚至将额外的信息作为Adornments（5.5节）包括在内。在这种情况下，“记录计费信息”似乎实现了一个较低级的目标，因此，创建一个子用例比较合适（EverUnfoldingStory，4.5节）。要RedistributeTheWealth，应该：

1. 从用例中提取话费记录信息，并使用它创建一个新的子用例，参见用例8.2。

2. 修改“处理正常呼叫”用例中的第6步，以引用该新的子用例，如用例8.3所示。

用例 8.2　通过创建“记录计费信息”子用例来 RedistributeTheWealth

记录计费信息

主参与者：主叫方——拨打电话的人

支持参与者：交换网——处理电话间呼叫的设备

级别：用户目标

主成功场景

1. 交换网记录此次呼叫的计费信息。如果这是主叫方在当天拨打的第一个电话，那么，它就会初始化客户文件中的日使用报告条目。如果是本地电话，它还把它记录在服务提供商的日志中。如果是长途电话，它把条目记录在相应长途电话提供商的日志中。

用例 8.3　对“处理正常呼叫”进行 RedistributeTheWealth，以使用例有一个 CompleteSingleGoal

处理正常呼叫

主参与者：主叫方——拨打电话的人

被叫方——接听电话的人

支持参与者：交换网——处理电话间呼叫的设备

级别：用户目标

主成功场景

1. 主叫方摘机。
2. 主叫方拨号。
3. 交换网转换数字符号，连接主叫方和被叫方的电话，并发送振铃信号。
4. 被叫方应答（摘机）并与主叫方交谈。

5. 被叫方和主叫方挂断电话。

6. 交换网记录计费信息。

7. 交换网切断连接，释放呼叫要求的所有组件。

8. 交换网释放了在呼叫中使用的所有组件后，用例结束。

（插图一幅）

在威斯康星州Alvin家中举行的缝被子聚会

8.2 MergeDroplets

您正在发现系统的UserValuedTransactions（4.4节）。

描述细小或孤立行为片断的用例不能传达足够的信息，来帮助读者理解系统如何提供UserValuedTransactions。

许多人都喜欢玩拼图游戏。将数百，有时甚至是数千块碎片拼到一起是一项很有挑战性且富于娱乐性的活动。想像一下在桌子上散落着数千块拼图的情景。拿出一块与盒子上的拼图进行比较，将其放在您认为它大致应在的位置。慢慢地，您已经拼出了更大的一块。很快，您已经拼出了天空、地面和山上的城堡的很大一块。一些热衷于拼图的人甚至都不用看盒子上提供的拼图。他们把所有的拼图块都翻过来，不用匹配颜色和图像，就能够把拼图块拼在一起。

用例不是拼图。用例代表了一个CompleteSingleGoal（5.1节），读者不应该象拼图那样对待用例，在心里把几个用例合并，以看到完整的概貌。

不完整的用例不能讲述整个故事。 单个用例应该为参与者提供一个“价值的完整使用”。用例片断或不完整的用例将会使读者忘记系统目标，并忽视重要的特性。为理解一个完整的行为，读者被迫寻找其他用例（如果能找到的话），或试图弥补用例本身的缺陷。通常的结果是，读者遗漏了重要的特性或添加了不必要的特性。

最好是在任何可能的时候定位信息。 使故事线索贯穿于一组用例会使读者很难理解叙述。将有关一个特性的信息定位到用例中，有助于读者理解主参与者的目标以及功能需求的上下文。

开发人员通常在每个用例的基础上实现系统。如果在多个用例间分配操作，就提高了开发人员遗漏某

些特性的可能性，或者多个组在无意中开发相同特性的可能性，这不仅浪费了大量开发时间，而且很可能会拖延项目。

最好使用例的数量最少。 必须实现的每个用例都提高了项目的成本，要求您进行管理、设计、编写、评审和测试更多的代码。每个额外的特性都增加了产品的风险，因此在构建系统时要实现尽可能少的用例（满足需要即可），在上市时间非常重要时更是如此。

较小的用例易于理解和使用。 理想情况下，您应该编写简洁短小的用例，以恰如其分地告诉读者他们需要知道的关于系统的任何事情。每个用例都应该充分描述系统如何帮助参与者满足一个目标（CompleteSingleGoal，5.1节），没有不必要的细节或技术细节（TechnologyNeutral，6.5节），否则，读者很可能会感到厌烦。

因此：

将相关的小用例或用例片断合并到与相同目标相关的用例中。

合并用例并不仅仅是只把几个步骤放到一起，它要求做更多的工作，因为最终得到的用例必须仍然包含一个好的用例所具备的模式。最佳方法是把步骤仔细编辑为描述CompleteSingleGoal的一个内聚单元。如果得到的用例太大或太详细，那么，对其步骤进行精化，以保持ActorIntentAccomplished（6.3节），或者对它们进行分级，以使其更加容易理解（LeveledSteps，6.2节；EverUnfoldingStory，4.5节）。很可能还需要仔细评审分支路径，删除一些路径并添加一些路径。

如何重新组织不完整的用例取决于其假定的目标和与其他用例的关系。

合并片断以创建一个新的用例。 如果几个不完整的用例描述了与相同目标相关的行为，那么，合并这些用例。当这些不完整用例以特定的顺序运行时，合并它们尤其有意义，因为按顺序排列的用例很容易重复信息。所得到的用例必须仍然满足用例CompleteSingleGoal的所有准则。

将片断合并到现有的用例中。 如果用例描述了属于另一个用例的行为，那么，将其放到该用例中。对于RedistributeTheWealth（8.1节），仅应该在不破坏用例的完整性并且仍然能提供CompleteSingleGoal的情况下才将其合并。

示例：呼叫处理

让我们再看看拨打电话的例子。人们可能想把用例分解为更小的、按顺序排列的用例，每个用例都描述一个不同的功能操作，如用例8.4中所显示的简化的例子。

用例 8.4　用例的噩梦：拨打电话的一组小用例

主参与者：主叫方——拨打电话的人

被叫方——接听电话的人

支持参与者：交换网——处理电话之间呼叫的设备

级别：用户目标

用例 1：拨打电话

前置条件：主叫方和被叫方的电话挂起

成功保证：系统收集了主叫方拨过的所有号码

主路径：

1. 主叫方摘机时用例开始。
2. 交换网发送拨号音。
3. 主叫方拨号。
4. 交换网转换数字符号。
5. 交换网将呼叫连接到被叫方的电话上。
6. 交换网向被叫方的电话发送振铃信号。
7. 交换网向主叫方的接收器发送回铃信号，用例结束。

用例 2：在电话中交谈

前置条件：用例“拨打电话”向被叫方的电话发送振铃信号

成功保证：用户完成呼叫

主路径：

1. 被叫方的电话响后，用例开始。
2. 被叫方应答（摘机）。

3. 交换网完成主叫方和被叫方之间的连接。

4. 被叫方和主叫方通话。

5. 被叫方和主叫方挂断电话，用例结束。

用例 3：结束呼叫

前置条件： 通过“在电话中交谈”连接呼叫。

成功保证： 交换网切断了呼叫，所有组件都恢复到了呼叫前所处的状态。

主路径：

1. 主叫方和被叫方挂断电话时用例开始。

2. 交换网释放在呼叫中使用的所有组件。

3. 交换网“记录计费信息”，用例结束。

这种程序化说明对编写代码不错，但不适用于用例。这些用例仅描述了呼叫的一部分，因此是不完整的，而且除了第一个用例外，所有用例都依赖于其他用例的完成。而且，每个用例都满足了主要目标拨打电话的一个子目标。这些因素表明我们应该合并这些用例。

编写该用例的更好的方法请见用例8.5。

用例 8.5　将小用例合并为新用例

拨打电话

主参与者：主叫方——拨打电话的人

被叫方——接听电话的人

支持参与者：交换网——处理电话之间呼叫的设备

级别：用户目标

前置条件：主叫方和被叫方的电话挂机

成功保证：交换网切断了呼叫，所有组件都恢复到了呼叫前所处的状态。

主路径：

1. 主叫方摘机时用例开始。
2. 交换网发送拨号音。
3. 主叫方拨号。
4. 交换网转换数字符号。
5. 交换网将呼叫连接到被叫方的电话上。
6. 交换网向被叫方的电话和主叫方的接收器（以使他们可以听到铃响）发送响铃信号。
7. 被叫方应答并与主叫方通话。
8. 被叫方和主叫方挂断电话。
9. 交换网切断连接，记录计费信息，用例结束。

对于读者来说该用例很容易理解，因为它包含了拨号、通话和结束电话呼叫需要的所有信息。读者无需通过阅读几个用例来发现关于该行为的必要信息，所有信息都在那儿，而且是按顺序提供的。

我们还注意到在该示例中，将用例合并后，表达行为所需要的步骤的总数减少了。合并后的用例包含

的信息与3个不完整用例一样多，但它不需要重复设置数据或提供信息来帮助读者将单独的更小用例连接在一起。

（插图一幅）

紧挨着垃圾堆的房子，威斯康星州密尔沃基

8.3 CleanHouse

您正在发现系统的UserValuedTransactions（4.4节）。

对于整体没有什么价值的用例是分散注意力的，并可能会使读者误入歧途。

文化是历史的积淀。北美那些储藏室、地下室和阁楼证明了我们积累财产的能力，大多数在很早以前就被遗忘了。车库中的车太多，我们都没法把车停进去了。我们花费了很多时间清理和重新安排物品，以发现储存这些财产的更好方法，甚至发展到了租借昂贵的储存室，来积累更多的财产。难以置信的是，我们花费了更多的时间和资金把这些物品包装起来，搬家时从一个国家搬到另一个国家，然后又把这些东西放在了新车库、储藏室、地下室和阁楼中。大多数东西我们永远都不会再用，但却一直留着它，因为“某一天它可能会有用”。但我们真的想让聚酯重新流行起来吗？当把东西扔掉后，人们的感觉通常会更好，但我们搞不明白他们为什么不在几年前就把东西扔掉？

与此类似，不需要的用例没有什么价值，只会造成混乱，将读者的注意力从系统的主要目的上转移开。更糟糕的是，他们可能会浪费大量的时间和精力开发没有人想要或需要的用例。

清除已有的用例会占用时间和精力。如果不理解用例或用例对系统的价值，就不能修改或废弃现有的用例。新用例通常还很新鲜，相对来说易于理解，老的用例通常就要求进行一些工作。重新学习需要时间，用例越老，所花费的时间就越长，因为在能够准确地进行任何改变或决定不再需要某些用例前，很可能需要评审整个用例集。

未使用的用例搞乱了工作空间，并且会浪费精力。 创建的每个用例都需要付出一些努力来维护，并且会提高项目的成本。用例越多，对读者来说就越难阅读。用例的集合应该仅包括那些描述了各种参与者和系统之间有意义交互的用例。无意义的特性不会为系统添加任何价值，相反，它们只会产生不必要的工作。不必要的用例可能会变成没有人能够理解，但每个人又害怕将其删除的用例，因为它们可能是重要的（请看Lava Flow 模式，Brown等人, 1998），它们成了系统生命的枷锁。

删除用例避免了大量的开发工作，并节约了精力。 必须实现的每个用例都需要更多的时间，这会提高成本，并且会增加项目的风险因素。您应该仅实现需要的用例，每个从一开始就可以消除的用例都会极大地降低开销。

将不必要的用例留在用例中的后果是，有人可能会尝试去实现它们，这样就会在您并不需要的东西上浪费宝贵的开发时间，并且会拖延产品的上市时间。大型开发组织的编写人员和开发人员缺乏交流，因此很有可能会出现这种情况。

因此：

删除不会为系统添加任何价值，或者已经不在现有用例清单中的那些用例。

确定了某些用例不再为系统提供任何价值后，把它们从集合中删除（UserValuedTransactions，4.4节）。可以将觉得有用的任何用例作为Adornments（5.5节）保留，但要立刻停止编写这些用例。确定是否放弃用例的一般规则是“如果有疑问，就放弃它”。如果废弃了在后来证明是有价值的用例，不要苦恼。如果用例是有价值的，您总是能够在以后把它重新放到用例集中。

根据意识到的价值权衡用例的实现成本，删除那些与其开发成本相比，目标太小的用例。有些用例的开发是得不偿失的。考虑删除这些用例，节约一些资金。您无论无何也不会再实现它们了，因为您将维护和增强更重要的特性。

示例

医疗索赔

考虑一下如下4个为医生的服务支付医疗保险索赔的用例：

用例1：支付住院索赔

用例2：支付门诊索赔

用例3：支付出诊索赔

用例4：支付到私人诊所看病引起的索赔

每个可能的用例都描述了不同的情况，每个用例都有它自己的微妙之处、支出以及形式。但现在医生很少出诊，因此用例3不太切合实际。它不属于该用例集，因为对整个系统来说它没有任何价值，尽管它可能包含一些其他用例中并不包含的信息。如果它提供了有用信息的话，可以作为Adornment（5.5节）提供该用例。

CleanHouse 解决了一个除软件开发以外，存在于许多行业中的问题。例如，在编写本书时，我们识别了几种最终决定不再采用的模式。有一些模式相当好，但随着编写的进行，我们发现它们不再那么有价值，因此我们放弃了这些模式。作出这种决定很不容易，我们就是否放弃某几个模式进行了多次讨论，但最终还是觉得没有这些模式本书会更好一些。

8.4 折衷与协作

编写高质量的用例是一个相当复杂的过程，潜在需求的不确定性加剧了过程的复杂性。目标发生变化，曾经是高质量的用例，现在可能只是行为的不完整片断，或者是解决了太多问题的大块。我们可能还编写了不再需要的用例。简而言之，一些用例未能满足CompleteSingleGoal（5.1节）标准，虽然通常不是我们自己造成的。

本章的模式是CompleteSingleGoal的助手模式，提供了对现有用例进行编辑，以使其满足该标准的原则。每个模式都依赖于CompleteSingleGoal。CompleteSingleGoal论述了用例应该在其级别上实现一个且只有一个目标，这是确定在一个具体的用例中应该包含什么的标准。我们可以用两种方式使用该原则：确定某些内容是否应该包含在一个特定用例中，如果不属于，应该把这些信息放在哪儿。这些编辑模式包括如下3种情况：

1. 将过大的用例分解为可管理的长度（RedistributeTheWealth，8.1节）
2. 将不完整的用例片断放到新的或现有的用例中（MergeDroplets，8.2节）
3. 删除不必要的用例（CleanHouse，8.3节）

更重要的是，这些模式告诉了我们如何完成这些操作，以及如何处理剩余的用例。分解过大的用例包括识别并移走与用例的目标不符的用例。我们没必要删除该信息，通常的做法是重新分配该信息，或者到新的用例中，或者放到现有用例中。将片断组合为内聚的单元能够使读者更容易理解具体的人物，无需通过阅读整个系统来理解具体的功能。把紧密相关的行为放在一个用例中还有如下好处：减少一些知识不太多的读者遗漏隐藏在另一个从表面上看不相关的用例中的细节的风险。最后，通过缩减用例的大小和数量来管理和减少实现人员的工作，可以使每个人变得更轻松。

有时我们可能认为一些片断甚至整个用例都不再适合用例的模式，但它们仍然包含了我们希望保存的信息。处理方法是把这些用例作为Adornments保存，无需把它们直接包含在用例中。

最后要说明的是，编辑用例涉及面非常广，在开始编辑前，您要理解系统和大部分用例。这些知识能够帮助您识别不合适的用例，以及应该把它们放在哪儿。否则，您很可能会遗漏系统中包含的一些微妙细

节，并使情况变得更加糟糕。如果您还没有这种经历，花一些时间来熟悉一下您正在编辑的所有用例，这样，您就可以在开始编辑前更好地理解它们。

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

《有效用例模式》
中译本即将上市

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

第一本 UMLChina 指定教材

清华大学出版社

烧毁这本书，别让员工看到—《人件》评论集

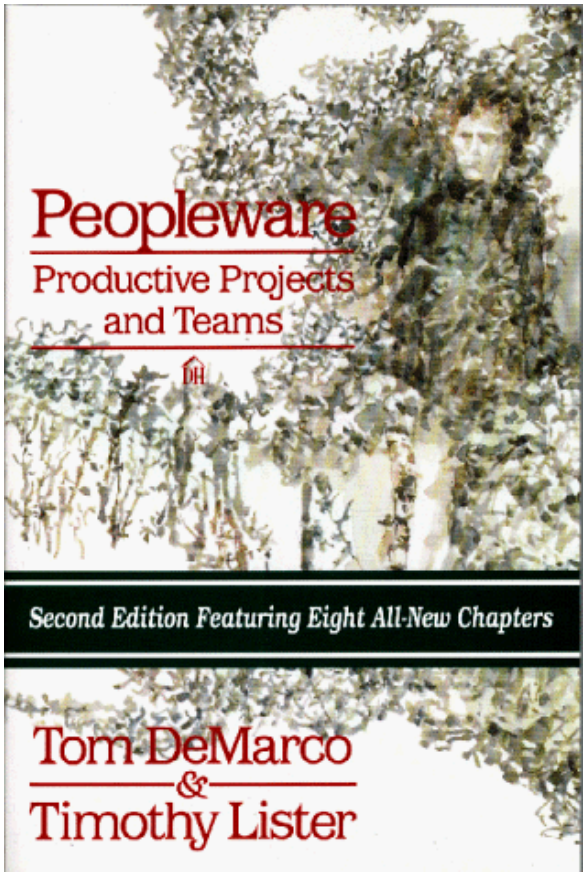
柳林 编译

查看评论

Tom Demarco 和 Tim Lister 的“Peopleware：Productive Projects and Teams”（人件：高效的项目和团队）第一版于 1987 年出版，专门讨论了软件开发和维护的团队管理问题，向传统的管理方法提出了挑战，推崇人本管理思想，给予软件工人自由和信任。和《人月神话》一样，该书现在已经成为软件团队管理的经典之作。**《人月神话》关注“软件开发”本身，《人件》关注软件开发中的“人”。**1999 年 2 月，《人件》第二版出版，增补了 8 章新内容。这些增补的内容视角更加宽广，对比较大型的组织中的团队如何运作进行了探索。《人件》的中文译本将于 2002 年 12 月发行，译者为 UMLChina 翻译组的方春旭、叶向群。

（老板，）在其他人看到这本书之前烧毁它。

——《人件》第二部分



Frederick P. Brooks, Jr

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。

--《人月神话》20 周年纪念版第 19 章

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品，特别是一些组织，开始认识到软件开发问题与程序语言、软件工程方法、软件过程成熟度无关。正如 Bill Clinton 所言，“是人，笨蛋！”你怎样去招聘、面试、挑选最好的软件开发人才？你怎样去奖励与激励他们？你怎样将他们组织为有效的团队？在这些事情中管理扮演什么样的角色？每一个对此都有不同的观点，但很少有人去评价自己的组织，或者问一问自己是否愿意为 Scott Adams 经常在漫画中进行讽刺的组织工作。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，如果你是一个老板，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 年后依然有效，并且更加热烈——新的版本增加了 8 章，覆盖了关于竞争、过程改进程序、“辞职员工再访”、组织学习、“首要的人”的要领、关于“终极”管理的讨论和一些怎样最好地创建软件开发“群落”的极好建议。

Jason Bennett

这本书说什么？

人件是两位资深项目管理者丰富知识的结晶。他们对管理者在对待下属与工作环境方面的错误尤其深有见地。这本书是从软件项目管理方面入手的，但它对任何 “智力工人”都有价值。也就是说，这些人需要集中精力进行工作而不是间断性的工作。

人件分为很多相互关连的章节。我主要从以下几个主题进行说明：

I. 管理人力资源

II. 工作环境

III. 适当的人

IV. 建造有生产能力的团队

V. 使每一个人工作愉快

“管理人力资源”论述的是如何对待智力工人。这个主题在书的第四页概括：“我们工作中主要的问题并不是现实的技术问题”。最后项目失败并不是他们不能解决技术问题，而是他们自己解散了。员工必须作为一个人对待，而不是一个机器部件。很多管理理论采取“装配线”(如汽车自动生产线)的方式来对待非智力工人。用这种对抗性，无理性的方法对待这些请来进行脑力劳动的人，会破坏工作环境，因为每一个人都是唯一的，不可替代的。你不能从地下挖掘出人力资源。

“工作环境”这章是我最集中，影响最大的部分。D&L 推翻了便宜，开放空间的理论，推翻了应该在工作空间上节约钱，环境不会影响人们工作的观念。想想一个人工的办公费用仅为他呆在公司工资的 1/20。如果由于糟糕的环境影响的员工的生产率，则在办公环境上节约的金钱与员工加班完成任务的贡献相形见拙。事实上，在最好的组织与最差的组织间生产率是 11:1。难道你能工作 10 倍的时间？

“适当的人”的说法后面是说，尽早雇佣最好的员工，因为管理者不可能自己去塑造员工。因此管理者应雇佣最适合于工作的人，而不是满足于一些标准的人。通过允许团队成员相互帮助，保持长期的稳定有助于这种情况。

“建造有生产能力的团队”着力于“整体团队”的思想。这些团队的每一个成员都关注目标，比单个的成员更有生产力。这种团队很难建造，但很容易破坏。如果管理者给予成员足够的自由，就更容易产生优秀团队。专注于质量，精英和保护团队的优秀组织容易产生整体团队。

“使每一个人工作愉快”的目的是：工作很愉快。那不是说理想的工作令人愉快，而是说好的管理者尽他的责任使工作更加愉快。做不同的事件，如战争游戏和飞行项目，并给予员工足够的自由，让他们用自己的方式去完成工作，这是一个使工作愉快的好方法。现在正是这种情形。

什么是好的？

当我读这本书的时候，我感到每一个章节都内容充实。D&L 不停地抛出以数据与自己的经验为支撑的重要观点。每一个与“智力工人”打交道的人都能从本书中获益，书中的观点可以提升工作环境。拿起这本书，读它，并找到改进的内容。你的合伙人将会因此而感谢你。

什么是差的？

我只能说这本书都很好，只是读者群比其它我评论的书更局限于独立承包人，或者其它喜欢按自己的方式工作的人。那些不在桌边工作的人可能与本书关联不多。

那些是写给我们的？

Redhat 更适合这本书，或者… 不，我只是在开玩笑。严肃的说，应该用“开放源代码”更中肯，我相信，你愿意将你的虚拟办公室设定在任何地方。无论如何，一个整体的团队比一群随机组成的团队更有生产率，以团队为导向加强了管理。

Compendiumdevelopments

扩展后的第二版在第一版出版 12 年后才出版，此时第一版已经成为经典著作。

令人不安的前景是，由于本书太过于真实，它的内容要么还没有被读到，要么被忽视了。

作者轻轻地引领读者，偶尔来一些幽默， 完成 34 章的内容。指出了偏离计划的动态原因。

这种书主要是针对于管理，读者主要针对管理者，但它对于被管理者也有用。团队成员可以通过设身处地的对比来帮助管理者和组织。

人是重要的，他们都是独立的个体，不断努力提高自己，他们有坚强的一面，也有虚弱的一面。如果没有被挑衅很少怀有恶意。他们享受教育，建立联系，他们愿意工作，管理应该是帮助他们而不是阻碍他们。

你应该尽可能地去读这本书。

如果你发现你的组织与书中的情形类似，就应该开始改变。

Sue Petersen

对于新出版的《人件》这本书，我除了“买下这本书”这句话外，真不知道还有更好的话。如果你还没有看过这本关于管理实践的杰作，那你应该在以后的学习中得到很多。如果你已经看过，DeMarco and Lister 通过新增的八章内容，从变更过程——人们怎样变更和为什么他们经常不，到“人力资本”和组织学习。在第一版出版后的十一年内他们又形成了两个关于团队自杀的方式。（Dilbert 的老板将感到骄傲。剩下的我们也密切注意）。他们在“大 M 方法论”中的思想可以使 SEI 中的人有所动作，正如帮助我们在过程改进努力中得到真价值。但我喜欢书的这些部分，完全重构了我对团队工作的图象，就是第 28 章。“竞争”。在这章中，他们以运动作比喻来说明我们的行为，然后指出为什么一个合唱团俱乐部更接近于一个“团结的”工作队伍。最后“如你的合唱队不能完美的作为一个整体，你绝对得不到人们的喝彩”。如果你仅仅是只做了你工作的部分，这对于你的公司和顾客没有好处。

Pmnetwork

这是一本关于项目管理的杰作的 1999 新版。基于人们－－你的下属是独特的、不可互换的这个事实。

这本书，最初在 1987 出版，引入了很多管理技术团队的的杰出思想。两个最杰出的是——

“团队自杀”，大公司事实上的政策与官僚习惯阻碍了团队的接合与连续性。还有

“家具警察”表示结构办公空间式的管理方式好象监狱（损害生产率和质量）

我在培训中特别引用一该书中的一句话：“在今天某个地方，一个项目失败了”

这本书的力量是它的预见性和易读性。每章都是独立的短文，尽管在通用观念与认识上有一些交叉，你都可以轻松地在几分钟内读完一部分。

对这个新版本，作者新引入了 8 章内容，而对原版本没有进行大的改变。新引入的每一章比原来的章节有更多的检查倾向，如减少规模，过程改进程序和管理变更。在这些主题中我发现了与原来一样的预见性价值。

正如作者就：“大多数管理者愿意承认他们对人的忧虑大于对技术的忧虑。但他们很少管理人的方面”这本书对一个试图从技术转向到团队管理的专业人士来说，是可理解和有价值的工具，我推荐它。

Mark A. Herschberg

这是我一直喜爱的软件工程书籍。《人件》正确指出软件工程是对人，而不是针对技术。它看到在软件开发过程中人的许多方面，并指出人并不是软件开发机器中简单的小齿轮。

这本书花费了许多的时间讨论团队，使你认识到团队的价值。但它没有一般的管理书籍列出“好团队”的标准，而是论述如何创造一个好团队，并指出它有多难。对于一个尽力去建造一个团队的管理者，这本书帮助他认识到成功的技术与技巧。它并不教你关于开发过程的知识，而是通过教你认识到在软件开发中人的价值去管理开发过程（但它并不仅仅论述管理者，我强烈推荐这本书给每一个人，从低级工程师到 CEO）。

这本书也包含涉及办公环境的章节，并提供证据说明为什么通常的观点并不适用于软件开发。我只学了这些章节，就促使我辞掉了原来的工作！

哟，这本书有一点不同于其它的书，它提供了更多基于多年研究的证据。

这本书改变了人对于软件工程的观点。

Shorewalker

脑力劳动需要办公室。

DeMacro 和 Lister 拼凑了一种理论：管理者应该帮助程序员、设计师、作家和其它的脑力劳动达到一种心理学上称为“灵感”的状态——人们可以静思以达到解决复杂问题的重要飞跃。你开始工作，抬起头来时发现三个小时已经过去。但这需要时间——平均十五分钟——进入这种状态。DeMacro 和 Lister 说现在的嘈杂，狭小的办公室很难使人有十五分钟内不受干扰。也就是说，在世界众多的地方，那些高薪并专注的程序员和创造性的艺术家花费一天时间而不能完成任何真正核心的工作。

Alan Cooper（交互设计之父）

这本美妙的书是关于管理编程项目的，无可否认，它有些偏离我们的主题。但是你要理解它就得将管理项目看成是一种人类的活动过程。作者用与以设计为中心的其它其它书籍同样的隐喻，所以他们的思考过程对我来说是一样的。通过这本书，作者展示通常对于办公室与办公人员的至理名言不一定正确，尤其当工作人员是程序员时，为什么？当我们更深地进入信息经济时代，所有的办公人员将逐渐都象程序员：原始的知识员工。因此，程序员的有意义的工作方法大多数都与设计二十一世纪的商业软件相关。

ROI

投资回报

“15-25%的项目被中途取消或流产…而失败的原因大部分归咎于人的问题。

(人件, DeMarco and Lister 第二版. 1999)

投资回报在现今社团社会中是第一位的问题。通过减少并使规模适度，公司对各部门进行考察以确保他们能有回报。团队中冗余的部分被排除出组织，管理要求行销项目能提供可以看到的投资回报。

激励程序的一个重要价值之一是，浪费是可知的，并且回报可以追踪。即使利益是无形的，例如员工满意度，也有方法去证明激励程序的效果。

PNC 银行集团和 PAF(公众事务论坛)的一次调查显示：

- 全美国发挥了全部潜力的员工少于 25%.
- 50%的人只按吩咐去做
- 75%的人认为他们的工作可以提高效率

- 80%消费满意度来自于知识渊博和专心的员工

无论如何，Demarco 和 Lister 补充，大多数的人喜爱他们的工作，有时非金钱的奖励更能激励他们。这就是激励程序的出发点。

- 一项美国补偿研究表明对每个员工花费$2000/人的激励程序可以使销售增加 20%。
- 总的 88%到 95%的激励程序达到或超过预定目标。
- 平均的销售激励 POI 是 134%。
- 非销售员工的激励 POI 一般为 200%。

还有，Loyalty Effect，Harvard Business，Forum Corp 在全球开展的关于顾客/职员关系的研究报告表明：

- 10-30%的客户流量/每年
- 50%的客户流量/5 年
- 50%的雇员为每 4 年一换
- 50%的投资者为每年一换
- 70%的消费者流失是因为服务不好
- 5%的消费者保持增长能够使从一个消费者的回报提高 75%。

Raghavendra Gururaj

我们终于有了一本关注论述软件工业中人的因素的著作，这可是一个好消息。当我第一次读完这本书时，我兴奋异常，即使现在再去读它也会激动。我们工作的主要问题并不是技术问题而是社会问题。我们在前面错误认为这如同给我们带来损失的政治。但是作者解释软件比政治更接近自然。我不禁嫉妒作者有如此丰富的学识与经验。

我们不常去阅读这本书的一个重要因素是由于软件工业所需要的管理风格。作者指出（非常直接），相对于生产环境，开发环境需要不同的管理风格。一些常见的错误是因为人们没有理解这种不同所造成的。设定不切实际或不可行的期限，使人们加班加点工作，追求华丽的，不现实的外表，追求高生产率，取悦高层管理者，等等。这些问题的叙述都令人难以忘怀。作者一定有卷入到事务的泥潭中不得不进行妥协的切身经历。这部分内容读起来非常有趣，作者还列出了项目管理者在管理中的一些错误期望。

作者在文中非常直接地指出管理者的功能不是使人们工作，而是使人们有工作的可能。强烈强调管理者建立一个健康，有益于工作的环境，管理者认为雇佣人才更重要。以上的三个论点令人愉快，读者可以从中受益。众所周知，整体比它的部分的和作用更大，毫无疑问，一个成功运作的团队能克服巨大困难。带领这样的团队达到

目标真是太好了。作者也指出团队的目的不是目标达成，而是目标安排。只有管理者能接受作者关于团队的观点，并开始按此去做，我才会非常高兴。通过强制要求每一个项目管理课程/学习/培训都阅读本书，我们才能公正地对待这项伟大的贡献。

简而言之，这本书是关于软件工业的最好的著作之一。它给软件工业的管理带来了显著的价值，因为该书论述的是人，人，更多的人是这个软件工业最大的资产。买下这本书，更重要的是尽可能地多次去读它。

http://flyingchihuahuas.editthispage.com/faq

我不是一个伟大的程序员。尽管我可以用特定的专长完成一些工作，但在最好的程序员与平均的程序员生产率为 10:1 的差距上，我只是达到平均水平的人，我处于人件金字塔的底端。

在《人件》的第二版出版的前夕，有人问 Tom DeMarco 和 Timothy Lister，软件开发的导师，软件经理应该从书中获取些什么？

DeMARCO: 好的管理者应该具有足够的才能熟练地建立一个社区。

LISTER: 我补充一点，人们都渴望把工作做好。如果你用别的方式管理他们，你会感到混乱。对我来说，渴望的是成为独立的贡献者，而不是，为什么我要从事我现在的工作。

MFESD

这本书最早大约出现在十几年以前，但它的内容在现在也还是适用的。

如果你相信雇用优秀的员工，为他们提供良好工具、设施、环境是软件开发成功的途径，则这本书是真正你所需要的。如果你认为只有自己才能完成成功的软件，则这本书将给你一些启发（可能有些争议）。

作者在文中涉及时间估计、空间与工作环境（使其安静）、电话（切断电话）、开放思想、雇佣、娱乐等内容。

这本书是一本经典——如果你重视软件的开发并希望你的团队成员作出最好的表现，而且你还没有读《人件》，那么请立即读这本书——许多智慧与实践的方法等着你去获取。

别把开发人员当成牲口

Mike Gunderloy 著， think 摘译

查看评论

Tom DeMarco 和 Timothy Lister 在 1987 年写了《人件》。1999 年，《人件》第 2 版由 Dorset House Publishing 出版，新增了 8 章。该书基于 1970 年代早期开始的研究，某些东西已经有四分之一个世纪的历史。对程序员的岁月来说，这是很长的时间。我好像回想起我们过去用古老工具编程的情景，但记忆相当模糊。

现在，那些炙手可热的新一代 dot-com 开发人员能从这本书中学到什么吗？足够令人吃惊——答案是 Yes！这本书是软件开发领域的经典，只有 200 页多一点，但值得每个团队负责人和经理阅读。

注意，这不是一本编程书籍。该书的子标题是“高产的项目和团队”。DeMarco 和 Lister 所关心的是生产力。用任何尺度来度量生产力，在软件工业的不同团队之间有 10:1 的差距。换句话说，最好的团队开发一个软件需要 1 年，最差的则要十年。原因看起来不止是最好的团队拥有最好的程序员（虽然最好的程序员趋向于加入有培养优秀团队环境的公司）。那么差别在哪儿呢？

DeMarco 和 Lister 认为差别在于：优秀团队有良好的工作环境，不必受制于那些对开发软件毫无益处的荒谬措施；经理不会挡团队的路（如果必要，经理还会把挡住本团队的路的其他人推开）。沿着这条思路，他们谈及大方法论如何使人变得愚笨，必须带门和窗的办公室，在工作中寻求乐趣，必须存在一些混乱，为什么有的组织能够学习，有的不能…

提防家具警察

作者认识到的很多问题归结为管理层把开发人员当成牲口，而不是使用大脑生存的人。例如，作者花了好些段落来把家具警察送上烤肉架。家具警察（说起来很悲哀）在很多成型的公司里都有。如果你曾经在一间“小隔间牧场”工作，下面的文字会为你敲警钟：

用家具警察的眼光来看，地下室的空间确实更好，因为它本身为统一布置提供了更充分的条件。但是人们在自然光下工作得更好。他们在有窗户的空间感觉更好，并且会把这种感觉直接转化为更高的工作质量。人们也不想在一个完全统一的空间工作，而是希望按自己的喜好和品味来布置自己的工作空间…工作空间总是充满了噪声、干扰，没有私人空间，并且没有多少办公用具。有些公司的办公场所比其他一些公司的要好看一些，但是实用性并不强。没有一个人能在那里把工作做好。那个人本来可以像一只海狸一样，在一个带有两个大的折叠桌子和一个关着的门的安静舒适的地方工作…。

幸运的是，在某些公司或者在某些领域，事情还是好转了。Microsoft 因为在办公设施方面“浪费钱”而“声名狼藉”：给开发人员配备带有门和窗的办公室，还提供免费饮料，休息区，还有其他很多无聊的东西。结果呢？Microsoft 的人们确实喜欢呆在办公室，自由自在地集中精神，写出高质量的代码。事实上，Joel Spolsky 曾声称：微软成功的原因之一就是公司里的所有经理都读过《人件》。Joel 推荐软件经理每年重读这本书一遍——这主意不坏。

家具警察只是更大问题的症状和伪装。问题在于对开发成本的错位认识，导致采取削减成本的行为，实际上从长远来看反而多花了钱，因为这些措施阻碍团队的形成，阻碍软件开发。一些其他的例子：

吝啬鬼管理层用那些可憎的“宣传工具”（你知道的，经常是一些全彩色照片加上一些类似“把灵魂献给公司是最高利益所在”的标语）取代了真正能起到激励作用的东西（如更高的薪酬和有门的办公室）。

制度上执着于过程改进程序（特别是 CMM），把精力集中在把事情做得更熟练而不是做市场需要的东西。如果 CMM 是现实的反映，那么完美的泥馅饼应该比缺角的椰奶馅饼好卖。

团队被分散在公司区域的不同角落，因为为他们找到一片连着的区域对家具警察们来说太麻烦。

现在你知道了，问题成堆。DeMarco 和 Lister 勇敢地把它们编辑成册。

不可思议的顺流状态

攀岩者会进入称为“顺流”的状态，在这个状态下，他们精神集中，在岩石上移动是如此的轻松和稳定，整个身体已经溶进攀登之中，每件事进行得都那么顺利。当然，我们可能会辨称，软件开发和攀岩不同，至少我们的代码崩溃的时候不会飞沙走石。但有一件事是相同的：顺流。书中对这种状态有很多描写。如果你在承担严峻的软件开发任务时，没有被各种琐事打断，你就会知道顺流状态什么样。这种状态不只感觉良好，而且也能使你得到好的代码。

顺流是管理层所不了解的东西，这就是家具警察能被授权控制你的工作环境的原因。正如 DeMarco 和 Lister 所指出的，顺流状态完全不为管理层所理解，因为管理者在工作中被经常打断是自然的。管理正是不断应付各种打扰的艺术。不幸的是，软件开发不是。在 5 分钟的电话之后，需要开发人员花 15 分钟或更多，以回到顺流状态。如果 5 分钟的电话来上 12 次，你就死定了。作为一个旁观者，我不喜欢看到有人煞有其事地研究为什么杰出的程序员进入顺流状态比其他人快得多。

记住，不止是别人强制我们不能进入顺流状态，我们自己也在这么做。如果你有顺流状态方面的问题，试着关闭你的 email 客户端几个小时。世界不会因此完蛋，而你就不会每 30 秒受到一次干扰了。

有趣的阅读

如果你已经看过了《人件》，那么现在怎么办呢？两条建议。首先，浏览一下 Atlantic Systems Guild (http://www.atlsysguild.com/)，该公司由《人件》作者帮助建立。其次，把它再读一遍！好好看它，这本书简单有趣。写这篇文章时，我很高兴又看了一遍。这样的精华语句百读不厌：

如果老板特别需要，仪式会议的负担几乎会不受约束地增加。例如我们知道一家公司，每天开两个小时的会议是公司的准则。如果开会时与会者离开会场，就用扩音器叫他们进来希望他们参加会议的全过程。不参加会议被视为一种威胁，要受到严厉的处罚。

你在书中不会发现一行代码。但如果你正在管理一支团队，你会发现聆听 DeMarco 和 Lister 的教诲会使你最终更快地在产品中得到更多更好的代码。

你的老板有这本书了吗？或者你想在他的桌上放一本作为礼物？

（2002 年 3 月）

人的问题：关于《人件》

Jacques Lebrun 著

查看评论

Il nome della rosa （玫瑰之名）

名称，神秘主义者们认为，与事物之间具有内在而非偶然的联系。在“现实”中很容易对此做出归谬——叫 Mark 的不一定都好战，叫 Emma 的不一定都爱提建议。对于书籍而言，就更其如此：奥维德和卡夫卡的名著除了类似的书名（Metamorphoses 和 Die Verwandlung 在中文里碰巧都是“变形记”）之外，之间并没有太多可以分享。

但神秘主义者仍可以将一本书引为例证：《Peopleware》（中译《人件》）的名称似乎就孕育了该书其他的部分，像是沙粒里蕴藏着整个世界。当然，我指的不单单是标题里同时包含了“人”和“软件”二者这回事，虽然也许据此我们就能推断出书中的核心论断：软件开发中，归根结蒂还是 “人的因素”占主导地位，因此，项目成败的成败、企业的存亡也更多的不在于技术，而在于“人”，或者说在于对于人的因素的重视程度、组织方式等等。换言之，本书的主旨固然可以从这个标题中“分析地” （取这个词在德国人那里的意思）得出，但我指的，更多的是该标题中包含的一种矛盾，而在我看来正是这一矛盾有效地推动、同时又内在地颠覆了该书的论述。

在我书房的一角，《Peopleware》尴尬地和其他原版“技术书籍”分享着位置。尴尬，首先就从书名中显露出来，这个自造的表达方式置身众多的“Patterns”“Models”和“Components”之间显得可疑、孤单、缺乏一目了然的“技术特征”。People 和 software 之间这个狡诈而轻快的拼搭，带有明显的“头脑风暴（brainstorm）”的痕迹（你还能在书中找到其他的例子，比如“teamcide（队杀？）”）。通常来说，类似的表达方式应该由一种人（“市场人员”）生产，另一种人（“管理阶层”）消费，与“技术”了无关系。——且慢，再仔细看看副标题：“Productive Projects and Teams”——这似乎又与“我们”有些牵涉？无论如何，“项目”和“生产率”在软件业中一直被认为是与“技术”有关的，类似的实践在词汇光谱中更偏近于“技术”一端而不是“行政”或广义上的“管理”。

叶饰中的人脸

而如果你打开本书，从随便的地方开始读上几页，你的反应也许就能证明你是哪一种人。可能对任何人而言，这本书读起来简直都太“容易”了，轻滑绵软，入口即化。但是就像止咳糖浆对一些人是良药，对大多数人就是

受罪一样，我预料不是所有人都能顺利地接收以这种方式传递的这样的信息：图表都是手绘的；不时穿插着《读者文摘》风格的小故事；显而易见的煽动性表述和故作惊人之笔；最后，是技术细节的缺乏和常识错误（e.g. 第186 页上提到了“…building PERL applets at Yahoo”）。这是一份加料过火版的呆伯特（Dilbert）读物（不幸地缺乏Dilbert 的冷隽和细微）。如果你（像我初读时那样）相信这是一本“软件工程的必读书”，尤其是指望在其中找到具有可操作性的方法，那也许你会在读后大呼上当。

这种风格（并不像我曾以为的那样）能用作者们的技术背景加以解释。两个作者中，虽然 Lister 简历上没有留下任何软件开发的痕迹，但 DeMarco 却确实曾任职于贝尔实验室，甚至还写过一本关于结构化软件分析的书（Structured Analysis and System Specification，1979）。据说，从前（面向对象被引入以前）脍炙人口的“数据流图（DFD）”就是那本书的发明。因此上面谈到的甜腻风格未必是某种缺憾（知识或能力上的，比如说）的结果，而是作者们有意为之：这是他们处心积虑的对推销员风格的模仿，而目标的消费群，像或许你已经猜到的那样，正是所谓的“管理阶层”。我读到的一篇（显然是有倾向性的）书评阴险地说道“我强烈推荐这本书给每一个人，从低级工程师到 CEO”,另一个有害的推荐说：“我强烈推荐你买一份给你或你的老板，如果你是一个老板，那么为你部门的每一个人买一份，并给自己买一份”，似乎这说的是什么营养品或特效药。在此我（谦卑地）提出背道而驰的推荐：如果你的老板还没有读过，你也一定不要读，除非你碰巧想换工作（原因内详）。

然而上面的区分仍然是过于幼稚的，甚至，本书自身就抵制这种区分。书中正确的指出，软件开发的管理人员，往往原本也是“技术”人员。因此他们似乎倾向于技术性地，而非“人性地”考虑问题：无论是一个项目，一个团队，还是一个企业。

以上论断，应该说是本书的主要成就，它描绘了（尽管是漫画式的，尽管面目模糊）一个角色，介乎“管理阶层”与“技术人员”、“行政官僚”与“狂人程序员”之间，正在“蜕变”但又旧习难改，非此非彼而对二者又都若即若离的一个人群。它试图捕捉这个飘忽的影子，给这个此前无人谈及的角色定位，并签发一张（在我看来是可疑的）诊断证明。如果手上没有原书，你可以去亚马逊网站看看该书的封面：白底上散乱的一幅绿色水粉画，大量的模糊叶饰中，隐约显出一张人脸。我的（不失独断的）假设是：这个面目难辨的人，就是上面提到的“软件开发管理人员”的肖像。书中的不少观点，原本的读者群可能是最上层的决策者，可是由于提出时的语气，每每不由自主地降落在我们谈到的这个含混的目的地。

但也许对于年轻的软件业而言，这张脸后面隐藏着的形象，还是一个过于不可测度的来客，很少有人能够在它面前能够感到自在，要对它开口说话，更是一项专门的学问。就个人而言，我感到本书的作者们就没有把握好语调：他们似乎努力显得生动、不教条，但结果却可悲地滑向了另一种风格，在德语（以及后来的英语）里，人们称之为 kitsch，这指的是一种由于故作高雅而带来的俗气——在讨论“办公空间”的部分援引《建筑的永恒之道》的一些段落加重了这种效果。然而如果仅仅针对各种觊觎或已陷入软件开发行业的“商务人士”，本书却确实提供了一个近乎理想的起点。上面提到的种种特性保证了它对匆忙而好奇的眼睛具有难得的捕捉作用。

软件企业的悖论

在我看来，本书（源自书名）的张力由两组近乎正交的对立构成。首先是上面提到的“软件开发管理人员”的身份矛盾。其次，则是“软件企业”自身隐含的悖论。

根据本书，软件企业似乎特别倾向于错误地理解自身。比如说（似乎是出于软件的组件化结构），管理者总认为开发者都是匀质的、可替换的；再比如，管理者也乐于认为开发项目的组织更多的是一个技术层面的问题，而与“人事”方面无关。以上两种认识，都似乎认为软件业脱离了各种传统企业的运作模式，只需要采用机械的，或者至少是技术的管理思路就可以成功的组织。针对于此，作者们论证道：软件生产，作为一种人类活动，面临主要问题并不是技术问题而是社会学（sociology）问题。软件开发者也并非像机械部件那样可替换的无面目的劳力，而是血肉之躯。我们尤其不应被所谓的“高科技”外表所迷惑，错以为高科技的介入就能使普通的管理模式在软件开发这里失效。因为（据作者们说）各种技术的发明过程确实是属于那种人迹罕至，需要绝世天才的高科技领域，但是软件企业中多数的情况只是在“应用”他人的成果而已。这种应用，本身与其他人类实践一般无二。这样，似乎软件企业也将需要与其他行业同样的管理科学和组织技巧，需要与大量“人”的问题打交道。那么本书的特殊处又如何体现呢？为什么不在完成以上论证后结束，并推荐一部《管理学引论》呢？

这里作者们又引入了另外的观点：软件企业终归有其特质，尤其是软件开发过程本身有其特质。管理者们往往又容易陷入轻信的圈套。他们（错误地）相信，开发者身上压力越大越好，加班越多越好，开发者都像他们自己一样不在乎电话打扰，开发者只配与其他员工（比如营销、客服人员）一样坐在开放的、由格子分割的办公空间中工作，开发者自身并不在乎产品质量高低等等。而（作者们提出的）事实是：压力和加班只能破坏项目进度；由于软件开发工作的特质，应该尽量保证开发人员不受打扰并拥有足够空间；开发者最关注产品的质量，甚至视其为荣誉攸关的大事。

如果以上对软件开发人员的报告全部属实，那么我们似乎又要重新建立关于软件开发的一种神话，软件业似乎又与所有其他行业重新天差地别、判若云泥。但是，就像摄影中正片和负片传达的信息量几乎相同，把一种完全错误的实践颠倒过来也不一定就是正确的。比如我就看不出，作者们鼓吹的软件业的各项“独有特征”，究竟有哪一种不能适用于其它行业，比如说，建筑师事务所。如果这个论点正确的话，也许可以把同一文本按照特定的词汇表作全局替换，然后推出一本新书《Peopletecture》？

也许一直以来，软件业对自身的理解仍停留在一个幼稚的阶段。究竟在哪些区域，哪些实践中，软件业有别于“传统行业”，而在哪些方面我们仍应保持对“人类活动”的忠诚？这种同一/差异间的对立，加上上文提到的“谁在管理”的困惑，形成了本书中推动而又吞噬着主题的隐形漩涡。作为牺牲者，作者们轻佻的风格没有允许他们看到位于漩涡中心的（可怖而壮丽的）图景。

Menschliches, allzumenschliches（人性的，太人性的）

恐怕我已经给人一种“本书的敌人”的印象。作为挽回和补偿，也许我应该表明，我赞同书中绝大部分具体结论。我身上幸存的开发者的良知告诉我，作者们关于加班、电话、空间和离职等问题的想法都是对的。

我尤其喜欢其中对开发者“沉思（immersion）”状态的强调：任何程序员想要进入富有“生产率”的状态，都有一个缓慢的过程（通常在15分钟以上）。进入该状态之后，他/她会感到工作特别上手，劳动效率极高。因此作为领导，出于利润最大化的考虑，管理者也应该尽量促进、而不是破坏这一状态。尽量少打扰开发者；尽量将同一项目组安排在封闭的、有足够空间的办公室里；尽量避免电话对这样的办公室的打扰（采用电子邮件，甚至语音信箱）；尽量职责明确（这一点可能是我加的，因为同一人有不同的职责意味着他要在多种沉思状态下切换）。虽然这里的“沉思”概念听上去类似于梦游，但这些并不意味着神话化软件开发过程，因为我们当然可以对建筑师事务所说同样的话。而如果我们忽视这种创造性劳动者（或者，用一个著名的矛盾修辞，“脑力劳动者”？）所必需的沉思，那么8小时的劳动时间往往会被侵蚀得所剩无几（考虑一下，如果你刚要睡着就被人唤醒，连续几次，一晚的睡眠就毁了）。这样就将带来两个问题：管理者在办公空间等细节上千方百计地节省开支，却忽略了他们所支付的最大开支（劳动报酬）的回报率；比这更坏的是，企业中流传着“在9点到5点之间什么都开不了”的说法，而这将导致加班—离职的恶性循环。

类似的精彩论断还有很多，如果你是一个管理者，尤其是，如果你本人并不熟悉软件开发过程，那么确实有必要认真体味这些珍贵的思考。我想，核心的意思是要打消一个错觉：如果你的下属喜欢他们的工作，那并不表示你的工作出了问题或是企业的效率降低了；恰恰相反，这往往是企业进入良心循环的标志。而如果为了节省（多少无知的举动以此名义而行）或其他荒谬的原因（书里的例子是“这显得不专业”）而给员工造成干扰、妨害、限制以至挫伤，那么最终的结果只能是士气低落、效率低下、离职率高，简言之：灾难。

是的，人性的，太人性的。所有的建议，包括提出的语气，都包含了这种“人性的”考虑。无怪乎一些书评称本书为“人道主义的（humanism）”。在这里人性本善，麻烦只是因为庸人自扰或是生性吝啬的经理们造成的（你见过Dilbert的上司头上的角吧？）劳动和资本的利益原本是一致的，只要实行黄老之道，员工的积极性就将被调动，一切都会进入良性循环。

书中对于团队精神（teamwork）和质量的论述尤其体现了这种人道的思想。根据作者高见，提高生产率的最重要方式就是形成有效的团队。但是，团队的形成，包括高质量的达到，都无需也无法通过明显的行政干预实现。比如，四处张贴“我们要的就是高质量”的口号无助于真正达到这一目标。据说开发者们心目中原本就追求团队合作，追求高质量，所以与其用外力加压，不如潜移默化地促使团队“成长”起来，而如果工期合理，开发者的骄傲感决不会允许任何低于自身质量标准（往往比经理们的要高）的产品出炉。

类似的乐园化论述不绝于耳。如果你认为这难以置信，作者们会拿出一系列“硬数据”说话：他们一直经营着一个面向实际企业的编码演习（“coding war game”）测试。上述观点，均已被这些测试证实。作者们先前的另一文章（Programmer Performance and the Effects of the Workspace，1985）系统地分析了这些测试，并得出了更严肃的结论。

虽然我愿意同情所有这样的论述，但是，正像上文提到的摄影术比喻一样，正片并不比负片包含更多的信息。“恶魔式”的老板的失败，也未必证明了“天使式”的老板就一定成功。简单地颠倒前者的实践，带来的多半还是灾难。比如说，较之于对个人感受和安逸的强调，团队的形成更依赖于好的交流，无私的互助以及核心成员的性格因素。单纯注重“人性”，未必就能形成团队，就像单单是领导的吝啬颟顸也不一定就会毁了团队一样。

我对类似言论的反对（或者称为“疑问”更好）也许更多地是“形而上学的”而不是“理智上”的。“人”，这个“人道主义者们”诉诸的终极对象，终归仍是一个被寄予了太多感情因素的抽象物。我们听到它被作者们用动情而不乏得意的声调多次提及，但是它真的像他们设想的那样，是软件开发的核心吗？人不也受到它卷入的某个特定的进程定义吗（就像恋人们被爱情定义，团队成员被团队定义）？就“核心”而言，“人”不应更谦卑地让位于此吗？除非我们满足于把一次思考等同于一项“商业操作”（那里，片面、粗疏甚至轻微的自我陶醉都是必要的策略），否则仅仅一个“人”字并不意味着最终的答案。

我的感觉是，与《人月神话》相比，本书在很多细节上有所不及，未能给出有效的出路（比如，《人月神话》中的“外科手术式队伍”就比本书中民主乐园式的团队更有可操作性），这可能仍然与本书摇摆不定的读者定位有关：再说一遍，这是为那些钟情于“头脑风暴”甚于具体操作的人准备的书。它致力于指认某些幻觉、扭转某些态度。如果你恰好处于作者们设想的位置、并也走在相应的方向上，作为一个殷勤的侍者，它会为你打开一扇特定的门。但似乎不应指望，它能领你一路回家。

参考资料

- The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition) by Frederick P. Brooks, 1995 Addison-Wesley
- Peopleware: Productive Projects and Teams, 2nd Edition, by Tom DeMarco and Timothy Lister, 1999, Dorset House, 245pp

本文首发于《程序员》杂志2002年第12期

开发人员是人吗？

Dr. Jakob Nielsen 著，Windy J 译

查看评论

由于正在为一个开发人员站点写这篇文章，我想我将挑战这个古老的问题：开发人员是不是人？我确实希望能从广大读者的血样中来个 DNA 测试，证明他们属于人类这个物种，但那并不是我关心的，相反，这里的问题是开发人员和其他人类是不是一样，或者，到底差了多少。

是的：开发人员是人。

在经典作品《人件》中，Tom DeMarco 和 Tim Lister 介绍了对程序员生产率和他们工作环境特征之间关系的研究，结论很清楚：人性因素对程序员们影响很大。

DeMarco 和 Lister 比较了程序员们在编程生产率上不同的表现级别，生产率最高的程序员们在拥有安静的工作空间方面得分很高，在拥有电话转移能力上得分更高，并在他们被打断的频率上得分相当低。相反，生产率最低的程序员们在安静工作空间方面得分比较低，在电话转移能力方面得分非常低，并在被打断频率方面得分很高。

很明显，（工作）被打断是程序员们生产率低的最主要原因。为什么？问题不是需要处理打断所需要的时间，而是被打断以后回到编程问题所需要的时间。任何人，不管做什么，在被打断以后回到原来的工作都会面对再适应时间的问题。当你正在阅读一篇期刊文章时，如果转去寻找某个问题的答案，然后阅读下一段，这样将比你连续阅读需要花更多的时间。

程序员们处境堪忧。他们需要在大脑里建立复杂的模型，对编程问题，对不同变量的状态，等等。这就是为什么大多数人不能成为好程序员的原因，但就算是最好的程序员在被打断时也会陷入困境。嘭的一声，大脑里刚才小心建立的模型就塌了下来，而回到刚才高效编程的状态很容易就要花掉 15 分钟，因此，一个两分钟的电话就会毁掉大约 17 分钟的生产率。

许多其他著名的发现证明，人性因素对开发人员非常重要：

Fred Brooks 的著名论断说，“给一个已经延期的项目增加人手只会使它延期得更厉害。”（《人月神话》）

好的程序员和差的程序员之间显著的差别：

通常在生产率方面有 10 倍或 20 倍的差距（同样，生产率更高的程序员通常提交最好的代码。）

在其他领域，人类的效率差别小得多：我过去常能在 2 倍世界记录时间内跑完 100 米。

要改进公司的软件工程水平，请遵循以下七个步骤：

- 只雇用最好的程序员，哪怕他们更昂贵。一个好的程序员胜过 10 个差的程序员。
- 为每个程序员准备一间关上门的私人办公室（不是小隔间）。
- 为电话提供秘书接听服务（或者至少提供一个转换系统允许将电话转换成语言邮件而不直接响铃）。
- 禁止邮件（或更实际一点，建立一种文化，在繁重编程的时候允许一两个小时不回复邮件）。
- 给每个程序员一台 21 英寸甚至更好的显示器（大屏幕可以更好地纵览复杂数据）
- 200dpi 的显示器一上市，就为程序员们配备（任何时候花在阅读屏幕上的时间都可以自动增加 20%的生产率）。
- 把所有程序员送去参加指法练习班。

当然，好的软件工程方法学一样很重要，但人性的角度经常被人们忽视。

不：开发人员不是人。

假设你属于这个开发人员站点预期访问者，作为我的一个未来读者，你可能属于那 1%最了解计算机的人之一。你很聪明，有着非常优秀的抽象推理技巧，你甚至能写一个用在搜索引擎上的布尔查询。

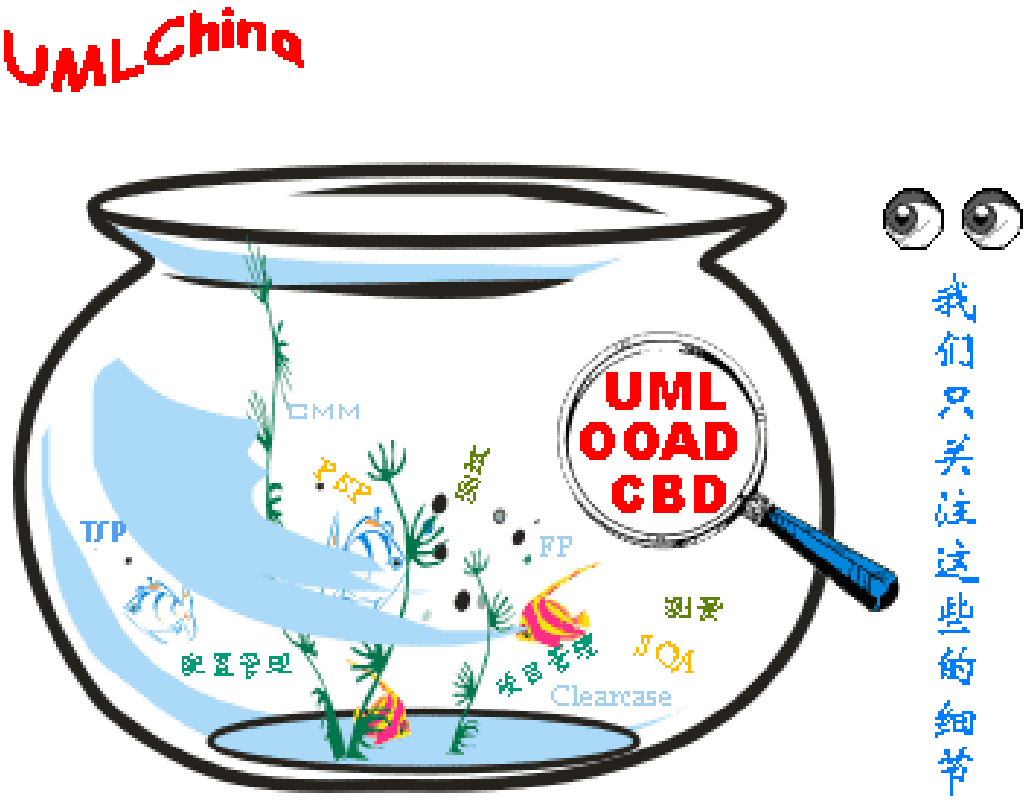
在恭喜你自己得到这么高的评价之前，我不得不指出，这些因素在评估你作为一个普通用户的经验时，恰恰是绝对的不合格。从定义上来说，绝大多数人属于那 99%计算机知识比你少的人。他们也非常聪明，但他们通常有着跟计算机不同的思维方式。而且，如果你给他们提供一个有着太多高级特征的搜索引擎，他们将找不到想要的东西。

不好意思，对搜索引擎说了这么多，但我非常讨厌的是这种倾向：用一些只有两种人才能正确使用的特征对搜索界面进行过度精化，而这两种人就是搜索引擎工程师和经过四年教育的图书管理员。象 Google 那样的简单搜索框就好多了，然后适当提供经过选择的高级特征，不要让用户输入一个有着成百上千个空格的页面，那些空格之间的相互影响也许需要大学教育程度才能理解。

幸运的是，也有一些方法允许开发人员绕过他们自己的才华并象普通人一样来感受他们的产品。在前一篇文章里我讨论了可用性生命周期，也建议了许多不同的阶段去主动采取措施收集用户数据。

现在，让我介绍一种证明很有用的简单方法：用户测试，找来普通客户，让他们使用软件，当他们觉得有问题时，应有的反应不能是“这个用户一定很笨”，而应该是“开发人员犯了错误”。那没问题，所有经验证明，没有人能第一次就设计出完美的用户界面，所以真正的错误是，因为你个人认为容易处理就不去改正可用性问题。

总之：你不是用户。



办公室空间，下一场革命

Diane Pettus 著，Windy J 译

查看评论

现在，许多三角区域的公司打算评估他们的办公空间需要，并决定搬迁他们的办公室，或者对他们现在的工作空间作一些改变。促使他们作决定的因素将会是营业间接成本，生产力方面的关注，和/或员工的方便性。

但是在这个抢座位的游戏中，那些作出明智选择（这些选择基于如何使员工工作最好的持续研究和这方面的知识）的企业主人数，将远远少于那些把临时解决方案拼凑起来的企业主人数，而他们的那些临时方案极少从雇员那里征求意见，并把布置的费用作为底线。

在他们开创性的作品《人件》中，Tom DeMarco 和 Timothy Lister，说这些“拼凑者们”将会吃惊地看到实际的成本：将立即导致生产力的损失和雇员的挫折感。

《人件》详细说明了精心设计的办公室空间将带来的成本节约，一是专业的气氛将使工作完成得更快更有效率，而且员工在这样的空间将感到自豪和被尊重。作者们主张企业必须对这些严重损害生产力的因素引起重视：拥挤，不合理的工作空间只会导致干扰，噪声，私人空间和会议空间的缺乏。

DeMarco 和 Lister 支持这样的观点，那就是企业主们可以通过合理规划的办公室环境，对他们企业的人月计算公式——期望每个员工在 30 天的工作周期中完成的工作——产生极大的影响。

设计能用来工作的空间

一个仔细考虑了办公室空间设计的真实例子是 Interface Technologies, Inc. (ITI)，一个定制软件开发公司，去年十二月刚刚搬到罗利（Raleigh）Six Forks 大道 Forum I 大厦一个大概 11000 平方英尺的单位。他们的总裁 Kelly Campbell 说他搬迁的目标是要建立一个能象加满了润滑油的机器一样运转的地方，一个因为尊重开发人员的日常工作模式，因为认识到客户们对效率的期望，从而不断增加能量的地方。

因为 ITI 的程序员们在项目开发中既要单独工作又要进行团队合作，它的大部分工作空间是大型私人办公室和配备完全的测试实验室的结合。

Campbell 说：“我们的目标是把干扰最小化。我了解到开发人员被打扰后甚至要花 15 分钟才能回到原来的工作。在我们的工作线上，当项目以小时计算成本时，我们需要开发人员能够集中注意力，能够找到一个私人的空间，关上门，关掉电话，思考问题，并且完成他们的工作。我们不希望员工因为在白天什么事也不能做而在这里呆到很晚。”

ITI 还有三个会议室，一个大的开放式多功能房，里面有舒服的家具，可以自由拼凑的桌子和椅子，这个多功能房里头还包括一个厨房，一个休息室，一个器材室，和一个正式的接待室。

“因为在我们的空间里，我们同样需要以团队的方式工作，既有公司内部的团队，也有同客户组织的团队合作，我们需要用来将开发人员和测试人员集合在一起的实验室环境”，Campbell 强调，“我们需要脑力风暴区域，阅读/研究区域，还有工作完成时集会和社交的空间。”

洞穴和草原

在《人件》里，DeMarco 和 Lister 通过报道软件开发人员如何度过他们工作时间的有趣发现进一步补充了 Campbell 的观察结果，根据作者们的说法，开发人员 30％的时间用于独立工作，50％的时间需要跟另一个人合作，还有 20％的时间跟两个或两个以上的人合作。

对上面的分析更进一步，他们基于空间员工密度画出了噪声和干扰程度，不出意料，DeMarco 和 Lister 发现噪声和密度直接成比例。作者们建议，既然 30 平方英尺空间的噪声比 100 平方英尺空间噪声的三倍还大，一个软件公司应该在产品开发时适当调高这方面的风险，因为不同的开发团队协作时产生的嘈杂将不断带来相互干扰和阻挠。

ITI 对此做出的回应是，把它的办公室配置成一个空间规划师所说的那种“洞穴和草原”。它的员工们，根据他们的任务，相对各自隔离（在洞穴）或分布在一个开放空间（草原），在这两种情况下，始终进行着有意义的工作。

所有的 ITI 办公室都“在玻璃上”，意味着每个办公室都有窗户，由两个开发人员共享。这种制度允许开发人员在新项目需要的时候可以容易更换办公室伙伴而不必感到自己在作出不公平的牺牲。四分之三的墙营造出 10×14 平方英尺的实验室空间，非承重隔栏形成草原空间，在办公室创造性地使用玻璃延伸了户外光线的影响，而墙上布满的白板给每个空间增加了多功能性。

遵循下列要素

ITI 的正式空间随着时间扩大到两个办公室套间，通过一个 5300 平方英尺的大厅相连。套间象一个布满独立办公室的迷宫，外面是实验室，以及有着等待列表的两个会议区域，在 11000 平方英尺的可用空间里，ITI 的 20 个员工组不再互相干扰了，而且，Campbell 在这次搬迁时着眼于未来，他从经验中知道，公司的发展可以破坏哪怕是最好的格局布置计划。

为了这个新的办公空间，LLC 理查德商业资产（公司）的房地产代理商 Olivia Moore，和 ITI 一起致力于找到一个地方，让它不但可以适应公司的发展和多功能性，而且对员工们来说，也将是一个方便的地点，于是最终因为它的配备和接近交通干道的模式而选中了 Forum I，好处还有室内停车场，现场的食物供应服务，靠近大型购物中心，这些都是考虑时的加分因素。

“当一个公司租用场地时，它付出的不止是每平方英尺的成本，” Campbell 承认，“在决定把我们的钱花到哪里的时候，我们决定听取我们员工的需要，以及客户的要求。我们成功地给他们传达了两个明确的信息：首先，你们对 ITI 真的很重要，其次，我们有一个专业的办公环境，在这里可以完成很多工作。”

从 Campbell 开始跟地产代理提起办公空间的话题到 ITI 正式搬入 Forum I 花了大约 6 个月的时间，在这段时间里，一个月用来跟建筑师商量布局，另外三个月是总承包人完成工程建设的时间。尽管精心布局和看似复杂，ITI 的办公空间却一点也不浪费，“我们花钱来影响我们公司，我认为这样已经达到跟我们经营公司时同样的专业水准。” Campbell 补充道。

底线

ITI 的新办公环境正在达到预期的效果，开发人员之间的沟通改善了，团队工作更有效率，而且 Campbell 说，当他的客户们看到一个“能工作的地方”时常会啧啧称赞。

ITI 的内部设计没有多余的装饰，但是 McGee 设计室可以通过颜色，织物，家具，艺术品和大大的标志，以及接待室的陈列物品来展现公司的外观和感觉。Campbell 反映说，“当内部设计师开始谈到某个展览设计师，我说，OK，先生们，你们必须给我看一些数字，因为我不知道我们是不是支付得起。让人惊喜的是，每项价格都很合理，他们在预算内工作，以求达到最好的效果。”

对 ITI 来说，一个意外的好处是公司已经在进行额外空间扩展，数年以来，公司一直在网站上为开发者们提供免费教程，ITI 可以在这个外延的空间里主持现场研讨会，例如最近为项目管理研究院当地分部组织的"Water Cooler"活动，就吸引了将近 90 名参与者。

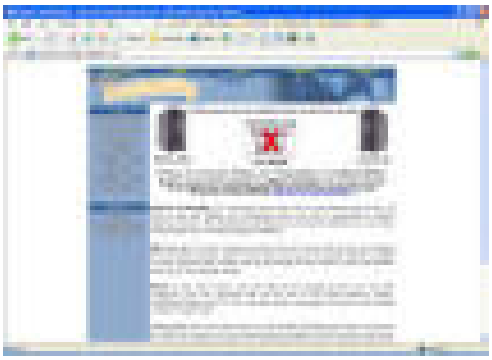
环境，文化，形象

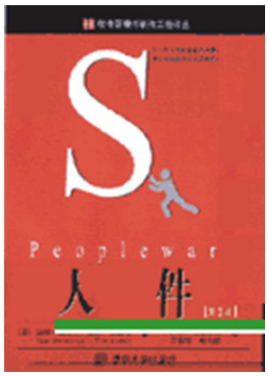
在每个行业中，都有用来定义生产率和员工满意度的可测量工作模式。这些模式已经成为企业文化和它最终外界形象的同义词。用一个不合理的空间设计破坏工作过程是企业每平方英尺办公空间上最大的隐形成本。

作为一个在节约客户开支的同时优先考虑巩固员工努力的企业主，Campbell 提供了一种终极思考方式，“设计办公室空间时，考虑你正在传达的讯息；作决定的时候，不要只询问会计师，研究一下团队生产力，尽量理解公司的工作方式；不要低估了良好环境将带来的收益。”

要得到关于工作空间生产力方面更多的信息，请参阅：

《人件》第二版 Second Edition，Tom DeMarco 和 Timothy Lister 合著, Dorset House Publishing Company, 1999.





http://www.peoplewarecn.com

人件中文版网站

《人件》在计算机行业的实践（待续）

UMLChina 摘录整理

查看评论

公司名	业务	员工受培训小时/年	举措	《人件》章节
Xilinx	可编程逻辑器件	不详	技术低迷期不解雇员工。	17 章 自愈系统
Adobe	多媒体处理软件	51	同事间的友谊是这家以图形产品闻名的硅谷公司的代名词：经常举行的有全体职员参加的会议，岗位轮换，周五晚上的啤酒聚会。每五年有三周带薪假期。	22 章 意大利通心粉晚餐 16 章 很高兴在这里
CDW	计算机产品	68	从来没有解聘过雇员。	17 章 自愈系统
高通	通信和信息技术	20	对领取薪水的雇员来说，休病假的制度是荣誉性的。不存在雇员滥用的风险；90% 的雇员期盼病愈后马上上班。公司餐厅为员工们供应寿司，晚上加班，可享用免费晚餐。	22 章 打电话说身体好了
SAS	企业软件	32	每月 300 美元的儿童保育补贴、公司餐厅（有钢琴师）、健身房、还有诊所，这一切都降低了离职率。	34 章 社区的形成

微软	软件	不详	这家巨型公司将最聪明的人材吸引到公司富有田园风味的厂区。厂区内设有垒球场、篮球场以及带淋浴设施的衣帽间。如果你愿意，你也可以免费得到非公司健身房的会员资格。	34 章 社区的形成
思科	网络	40	规模庞大的儿童保育中心看护 400 多名儿童——这在硅谷的同行中很少见。	34 章 社区的形成
思科中国	网络	/	把“看得见风景”的窗口留给员工；允许员工调换岗位；不限制员工的办公地点和时间；帮助员工在家里安装宽带；员工平均每年参加 6 个培训班。	第二部分：办公环境（7-13 章） 2 章 做吉士汉堡，卖吉士汉堡 16 章 很高兴在这里
英特尔	芯片	45	没有专用车位、私人办公室、经理餐厅	7 章 家具警察
SGI	三维图形软件	23	工作四年后可以享受六周带薪假期。同时在公司内还提供按摩、波足桌、咖啡机、享有补贴的自助餐厅，和沙地排球场。	第二部分：办公环境（7-13 章）
SRA	IT 咨询	30	所有新雇员都能够得到 30 股股票。几乎 100%的雇员都认为公司管理层是诚实的。用一位雇员的话来讲,“这里的确以理服人。”	第四部分：培育高生产力团队（18-23 章）

IBM	计算机产品	50	在全球拥有 67 个日间托儿所（在美国境内有 61 个）。自 1984 年以来，公司已在包括子女保育和老年保健补贴在内的家属照护方面花费二亿多美元。	34 章 社区的形成
IBM 中国	计算机产品	/	晚上 6 点以后，公司放音乐，催促员工回家。如果主管同意，员工一周内可以有几天在家办公。家有学龄前儿童的女性员工可以停薪留职，请假一两年在家照顾小孩。	3 章 维也纳在等着你（世上没有加班这回事）
Network Appliance	网络存储	40	雇员行为古怪，他们自己深知这一点：一个标示牌欢迎来访者进入公司总部，而以往的销售活动都是以与真人大小相仿的身着《星际探险》（Star Trek）剧中人服装的公司主管的剪影像拉开序幕的。	14 章 霍恩布洛尔因子（爆米花不够专业） 19 章 黑衣团队
Autodesk	设计软件	50	雇员们用当场颁发 1,000 美元奖金的方法为一位同事带来惊喜。雇员们还可以享受长期的福利：在工作四年后可以享受六周的休假，同性恋配偶与政府承认的配偶享受同样的待遇。一位雇员说：“你怎么能不喜欢为一家每天都能把狗带来上班的公司工作呢?”	25 章 “自由电子”
IDG	计算机媒体	12	办公室增添了一个拱廊；另外一个办公室让编辑人员负责进行聘用面谈。	第二部分：办公环境（7-13 章） 15 章 雇用一个变戏法的人

Acxiom	综合数据服务	38	没有繁杂的行政管理，可以选择弹性工作时间，也可以在家办公。	7 章 家具警察 25 章 “自由电子”
Intuit	软件	不详	谁在度假时还想到老板？Alan Hampton 可能如此，因为雇主 Intuit 公司会为他的休假付费。雇员们在回报社会方面作出了重要的贡献：雇员们每年有 32 个小时的带薪志愿服务时间，去年社区服务的时间达到 7,600 小时。	24 章 混乱和秩序(培训、旅游、会议、庆祝和休养所)
Sun	计算机软硬件	40	95%的雇员采用弹性工作时间；100%的雇员可报销最多一万美元的学费。	16 章 很高兴在这里 25 章 “自由电子”
Electronic Arts	游戏软件	26	雇员们在感到充满压力的创造力下降的时候，为使自己精神放松，便到直径为 81 英尺的迷宫中徜徉。“在像我们这样的增长型公司，我们不缺乏灵感，我们所要做的 就是为找到灵感创造合适的环境。”公司的游戏开发者们也可以 享用免费的卡布其诺咖啡，或者打排球和篮球，以恢复精力。如果想要休息的话，在工作七年后可以享受七周的休假，在工作 12 年后可以享受 12 周的休假。	10 章 脑力时间 vs. 体力时间（顺流）
德州仪器	计算机产品	34	雇员们着装随便，礼仪也如此：从首席执行官开始，每个人都以名字相称。	第四部分：培育高生产力团队（18-23 章）

注：以上有关公司的资料摘自《财富》