这一期出刊晚了，而且质量也不满意，下一期一定改进！

——《非程序员》

第 30 期（2003 年 10 月）

职责

X 非程序员
X-Programmer
软 件 以 用 为 本

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

http://www.umlchina.com/

*拥有一个已经注册半年的 UMLChina 讨论组 ID 或者《非程序员》曾刊登过您的译稿

*32 岁以下

*本科以上

*在软件组织（软件公司或行业公司的软件部门，不包括高校和纯咨询公司）中有至少 5 年的全职分析、设计、编码的经历；在这期间，至少有两年是主要从事编码工作

*现在还在从事分析、设计、或编码的工作

*谦逊

*有重构自己现有知识的勇气和能力

*有研究精神

*英语很好（阅读、翻译、表达）或者认为自己在需要的时候能迅速使自己的英语水平达到该有的水平

*对自己的表达能力和演讲能力有信心

如果您符合以上的全部条件，而且对成为一名 UMLChina 专家感兴趣，在方便的时候请下载此表填好寄往：think@umlchina.com。

特别说明的是：

*这不是招聘广告，我们不能承诺会给谁一份 offer。寻找的人选只需 1-2 名，即使最后我们选中您，也需要花 1 年时间通过训练和实践来重构您的相关知识。所以，如果想“找工作”，这不是一个机会。

*虽然 UMLChina 专家最终只专注于 UML 相关技术，但上面条件中并没有 UML 在内。

Borland 将 UML 引入.NET 世界

[2003/9/11]

Borland 公司正在开发其 Together 工具的新版本，要为.NET 开发社区带来建模和设计的能力。

Together 的.Net 版本中把 UML 建模设计和 Microsoft .Net 框架下的代码结合在一起，并降低 UML 的复杂程度，Todd Olson，Borland 公司 Together 产品线的首席科学家，如是说。

例如，Borland 的 LiveSource 功能可以保持模型和代码之间的同步，保证用户对模型所做的任何修改可以在代码中自动对应。

“.Net 程序员还没有真正使用过 UML，不过，这一天将很快到来”， Jim Duggan，一位 Gartner 的分析员称，“随着应用开发规模的越来越大，设计驱动编程将会更加流行”。

与.NET 相比，使用 JAVA 开发的项目尤其如此，他补充说，J2EE 的开发人员的工作中实际上已经包括了建模的部分。

微软的建模支持是通过 Visio 提供的，他们对 Together 的评价是 “构建的模型是和应用开发的代码紧密结合在一起的”，微软面向开发人员和平台部门的高级产品经理 Prashant Sridharan 介绍。

Sridharan 认为微软将会在十月下旬召开的 PDC 会议（开发人员大会，Professional Developer's Conference）上深入地讨论在建模工具方面自己的计划。

“Together 的产品符合微软在建模方向上的策略” Duggan 说，“尽管它和微软自身的建模设计有些差异，后者将包括 BizTalk”

Olson 认为，对 Together for Visual Studio 来说，模式支持同样是一个新课题。后者提供了可以支持重用，以及自动文档生成的应用设计的能力，它们使得文档更加容易得到并共享。

（自 InfoWorld，袁峰 摘译，不得转载用于商业用途）

Borland 开始给予 C#平等对待

[2003/9/11]

Borland 正在展示其整合微软开发环境的设计和建模工具。经历了多年在 Java 工具领域的成功，Borland 公司现在开始给予微软的 C#语言平等的对待。

在 Orlando 召开的 Visual Studio 会议 VSLive 上，Borland 展示了它和微软开发环境完整结合的第一个设计和建模工具。据产品管理负责人，Michael Faisst 介绍说，这是 Borland 在 C#方面的第一个版本：Together Edition for Microsoft Visual Studio .Net。

这个建模和设计工具来自于 Borland 收购的 TogetherSoft 公司。Borland 为此投入了八千万二百五拾万美元，这还不包括九百万的 Borland 股份交换。这次收购给 Borland 的代码开发工具带来了全套的建模能力，为其在和微软及 IBM 相关工具的竞争中占据了有利的位置。后者去年收购了 Rational 公司。

Borland 已经为 Java 开发者提供了建模工具中代码的同步功能。现在他们正在为 C#开发者提供同样的功能。Faisst 介绍，Borland 的 LiveSource 是 Together 版本的一部分，它能够保证代码的变化映射到模型，反之亦然。

“代码和模型的同步是一个大问题。开发人员在改变了源代码之后并不乐于返回头去修改模型”，Faisst 说，Together 中存储模型和开发人员的源代码，并自动维护二者之间的同步。

另外，Together 为 C#开发者带来统一的建模语言符号和语法。使用 UML，可以由应用的模型自动生成其代码框架。今年二月份，Borland 首次在其 Jbuilder Java 工具中集成 TogetherSoft 的建模功能。

Together Edition for Visual Studio .Net 中引入了 JAVA 开发者社区中另外一个普遍的概念－模型的应用。所谓模式，就是从解决编码难题的编码经验中总结出来的最佳实践。Faisst 说，在解决同一个问题的多种解决途径中，Together Edition 中加强了一种模式的应用，以保证得到高质量的代码集合。

由于 C#还处于发展阶段，“和 Java 比较而言，我们的模式数目还有限”，他说，但是当更多的好的模式被发现时，建模工具将适时地加入这些模式。“（译注：通过模式的应用，）我们不需要重复编码，如果你能够保证模式的正确，那么应用的质量将得到提高”。

（自 informationweek，袁峰 摘译，不得转载用于商业用途）

Duvessa 发布基于用例的估算工具

[2003/9/16]

Duvessa Software 今天发布了 Estimate Easy UC，一个强有力、低成本和灵活的工具，它可以使用用例点来估算软件开发的工作量。

Estimate Easy UC 用于捕获用例数据，它可以在需求阶段很早的时候进行，不必等到详细的需求被发现之后。Estimate Easy UC 是第一种能够定制用例点估算因子的工具。目前的研究表明，用例点能产生可靠的估算，和实际的工作量很接近。

Estimate Easy UC 允许手工输入用于估算的 actor、用例、技术和环境因子，也能够从 Rational Rose，XMI 1.1，Microsoft Visio 等工具输入。用例数据也可以从 Rational RequisitePro 输入。Estimate Easy UC 允许修改所有数据和产生多种估算。估算得越早，团队意识到风险越早。“Estimate Easy UC 很容易使用，它允许尽早估算，然后随着需求阶段进程不断调整”，Duvessa 的首席技术官 Rahmanian 说。

Estimate Easy UC 的免费试用版在 www.duvessa.com 下载

（自 emediawire，不得转载用于商业用途）

Gentleware 领跑 UML2.0 建模工具

[2003/9/29]

依助本月问世的 Poseidon for UML 2.0 ，Gentleware 在实现了新的 Unified Modeling Language (UML) 2.0 规约的建模工具领跑者中占得一席之地。

Marko Boger 是总部位于德国汉堡的 Gentleware 的 CEO 和创业者，他认为他的公司是实现 UML 2.0 这场竞赛中的领头羊。Boger 介绍，新的工具集中允许所有模型图的交互，依此提供了和其它 UML 工具的交互能力。

Boger 告诉 eADT，四月份 OMG 成员率先认可这种实现方式弥补了先前版本中无法进行图形共享的缺陷。“XMI 标准本来是来解决这个问题的，但自己却有问题。” Boger 说，“OMG 希望在 UML 2.0 中修正这一点。”

Boger 在大约三年前创办了 Gentleware，目标是在改进开源的 ArgoUML 工具集的基础上提供一个建模工具，前者在技术和市场两方面都有局限。Boger 参与了 ArgoUML 的开发，但他很早就意识到这个技术“产品化”准备的不足。 一年以后，Gentleware 发布了其第一个 UML 建模工具，并迅速赶上业界领袖 Rational 和 Togethersoft 的步伐，这两家公司已经分别被 IBM 和 Borland 收购。

观察家们预测，包括 Borland, IBM 和 Telelogic 等各家公司推出的各种 UML2.0 的工具将在几个月之内带给市场巨大的冲击。不过 Boger 坚持认为，与其它公司相比，Gentleware 的“Internet 市场和销售策略决定了其价格策略的与众不同。”

开发人员标准版本的多平台 Poseidon 工具集被标价为$199 ，高端的专业版本价格为$699。嵌入式系统开发使用的版本价格为$1,249。这些工具都可以从 www.gentleware.com 下载得到。

（自 adtmag，袁峰 摘译，不得转载用于商业用途）

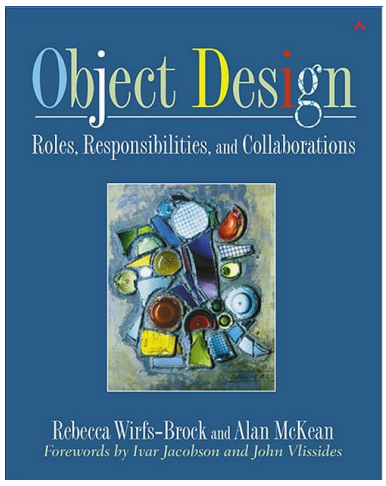
Rebecca Wirfs-Brock 访谈

ObjectbyDesign 整理，李巍 译

查看评论

介绍

Objects by Design很高兴能将这次对Rebecca Wirfs-Brock的专访带给我们的读者，Rebecca Wirfs-Brock是一位面向对象软件设计和开发领域非常知名的专家和顾问（编注：她在1990年出版的”Designing Object-Oriented Software”是最早的面向对象设计书籍之一，该书中译本《面向对象软件设计经典》已于2003年9月由电子工业出版社出版）。她在新书*Object Design* (Addison Wesley - November, 2002)中很好地融合了其对*CRC卡*、*协作（Collaborations）*和*灵活性（Flexibility）*这些主题的见解，而这也正是我们这次访谈的主题。这本书以其独到、新颖的见解而倍受推崇，这些也定将为我们的读者所赏识。



CRC 卡

SZ：目前，CRC 卡成熟已有一段时间了。Kent Beck 和 Ward Cunningham 在 OOPSLA'89 大会*（译注：OOPSLA 全称是 Object-Oriented Programming,System,Languages,and Applications）*上发表了一篇介绍 CRC 卡的论文（http://c2.com/doc/oopsla89/paper.html）。我知道你本人是一位 CRC 卡的忠实支持者。你是否仍然发现它们在你的工作中非常有用呢？为什么 UML 工具没有对 CRC 卡提供任何支持呢？这是不是因为它们并不符合 UML 的标记（notation）？

Rebecca Wirfs-Brock：我们主要是在我们的培训中使用 CRC 卡（以及有时候会在分析和设计工作中使用）。是的，它们非常有用。尤其是在描述一个候选对象的基本特征和行为的时候。

以前，CRC 表示的是类-职责-协作者（Class-Responsibilities-Collaborators）。我当前的思路是将第一个 C 用来代表“候选者（Candidate）”，并有意识地不把每张卡片表示成一个类。我想通过支持「定义由类所实现的接口」，我们便有机会将（设计）决定映射成实现，而不应该过早地在「“候选者”是一个类，还是一个由多个类所实现的角色（role）」上悬而不决。

我不会在「使用实际的卡片建模」上太过教条，但是我会在「表达正确抽象级别上的职责和协作以及对象的角色」上变得非常教条。CRC 卡是一种保持事物简明扼要的良好方式。因此我比较愿意使用白板来建模「对象模型的 CRC 卡级描述（CRC card-level descriptions）」，然后将这些描述转换成 MS Word 中的表格，或甚至是我的客户所喜欢的设计工具中的类。然而，这种「描述性的职责」和「它在你喜欢的建模工具中的表现形式」之间的转换并不总是那么直接。

你问了一个很好的问题：为什么 UML 工具没有对 CRC 卡提供任何支持呢？我更愿意把它说成「为什么 UML 工具没有对表达不同抽象级别上模型元素的行为和特征提供良好的支持」。我想这是因为大多数工具（尤其是那些支持双向工程（roundtrip engineering）的工具）专注于提供类的一个实现级别上的视图。Fowler 谈论过三种关于抽象的级别（相信可能会有更多，但这对理解已足够好）：你的设计的概念视图（conceptual view），规约视图（specification view）和实现视图（implementation view）。尤其是好的 CRC 卡级描述相当于一个概念视图。我想大多数「致力于将你的规约转变成实现的工具」不会看到这种价值（或者可能不会有一个清晰的进行支持的方式），它们掩盖了一个设计的这些不同的视图。我希望他们能够去做，并且我很高兴能与工具厂商合作来使他们支持多种视图。

我知道工具中的这种能力一定会为我的客户所赏识。

SZ：你能阐述一下你所认为的“概念视图”的基本元素吗？

Rebecca Wirfs-Brock：当然。一个对象设计的概念级别视图描述的是关键的抽象。而有些人正好将关键抽象想象成“候选类（candidate classes）”的高级描述，我更愿意从一个稍微不同的角度来考虑概念设计（conceptual design）——我是在一个稍微不同的级别上考虑设计的。

一个面向对象应用是一组相互交互的对象。每个对象便是一个或多个角色的一个实现。一个角色支持一组相关的（具有内聚性的）职责。一个职责便是执行一个任务或知道某种信息的一种义务（obligation）。而且对象并不是在孤立地工作，它们与同一群体中的其他对象相互协作来执行整个应用程序的职责。因此在一开始概念视图至少要是关键对象角色及其职责（较高级别的陈述）的精华所在。许多你最初建模的候选者将很有可能（除非你建立了分级的层次体系并使用继承和组合方法）直接被映射成某种继承体系中的一个单独的类。但是我想让这种可能性显现出来，通过在一开始就考虑角色和职责，然后第二个步骤是转向规约级别视图，将这些候选者映射成类和接口。

SZ：请试着描述一种 UML 工具能够支持 CRC 卡的具体方式。比如，它是否有助于打印出 CRC 卡的三个信息元素：名称，职责，协作？

Rebecca Wirfs-Brock：我想到一种方式——不要坚持我必须总是具体地从操作（或在类的情形下一它的属性）的角度来规定一个类或接口（UML 将这些称之为分类器（classifier））。*除了*下一个细节级别一属性和操作，请让我从职责（陈述的是它知道什么或能做什么）的角度来规定类或接口的行为。打印 CRC 卡可能轻而易举，但更重要的是，应该有一种方式能够支持从角色和职责的角度描述你的设计。UML 中角色（即关联（association）线末端的名称...但这种视图过于局限一一个类实现了一个或多个角色，能够很容易地说明其所扮演的那些所有角色——而不必通过察看那些连接（link）加以推测）的概念比较弱。

另外我希望工具提供一种方式使我能够对未分配的职责进行追踪，并且使我能够说明那些经常会被我放到 CRC 卡另外一面的东西——陈述一个候选者的意图，其角色的构造型（stereotype），以及可以使用的模式角色。不过我有些跑题。总结一下我对你问题的回答，我更愿意看到 UML 工具厂商能够把职责驱动设计（responsibility driven design）的概念集成到「表达一个设计」的整个过程中来，而不是添加 CRC 卡的模板。对我而言，捕获我最初的职责驱动设计的思想，并将它们不断地织进（weave into）当前所进行的设计中，要比「工具对“卡级”视图提供支持」更加重要。这种价值不同于用手来书写 CRC 卡——它们是非正式的，它们能够连续进行工作，并且你能够在它们上面写下任何你要的想法。工具生成的卡片可能不会这样灵活。

协作

SZ：你在书中对协作的论述是集中在预排（walk-through）的价值上面。我的一个非常成功的 OO 项目就是在编写任何代码之前，对所有的协作进行预排－这真让人愉快！敏捷软件（Agile Software）提倡白板会议（whiteboard session），这对于协作的预排很有好处。你是如何召开这些会议，以及在实践中成功地发现它们的呢？UML 工具将会以什么样的方式来更有助于协作的预排呢？

Rebecca Wirfs-Brock：对协作进行挑选（以及开发一个统一控制风格的设计）对于大多数设计工作的成功非常重要。我喜欢致力于协作建模的工作。如果你只是随意地去考虑谁与谁进行协作（以及为什么），那么你可能不会从你的建模工作中获得任何实际的好处。很多人并没有一套指导原则能够促使他们做出协作方面的决定...从而他们会陷入以下境地：他们往往非常随意地在协作者间进行职责的划分。我有许多想要应用的指导原则，并且也坚信委托控制（delegate control）的风格是一个好的想法，它能使交互（interaction）变得清晰和简单。

我认为工具所欠缺的一个地方是有助于协作建模（collaboration modeling）的方式，或者更好的说法便是，有助于讲述适当的设计故事（design story），一旦你对协作模型做出决定。如果一个工具能够允许我建立一个协作（通过预先装载专门的测试案例），然后能够让我“运行一个协作”并且拍下位于正确精度级别上对象之间的交互，那我将真正喜欢上它。这比「模仿对象的交互，然后绘制出示意性的图片」要好上许多（而且会更加正确）。为什么没有工具能够允许我作一些原型（prototype）工作，建立一个协作，然后帮我对结果进行归档。

我到过很多家组织，为了演示对象的协作他们所必须绘制的图（diagram）的数量真是看起来教人畏惧。通常我想要消除「图越多，就意味着设计越好」的想法，而且一个设计能够为人所理解的关键是要知道什么该说（以及什么该省）。人们应该认识到有许多交流设计选择的方式，要比「绘制许许多多看上去几乎相同的图」紧凑得多。在一个典型的协作图中装扮上图表或表格来说明异常情形（如它们是如何被处理的，以及对用户的效果），远远要比绘制 30 个图更加有效。如果一张图片能使头脑麻木并且无法传达正确的信息，那么它的价值尚不如 1000 个单词。

SZ：最近，模式已经能帮助软件开发者以富有效率的方式组织他们工作的交流。你有没有想过把你的协作策略（collaboration strategy）组织成模式，使之更加容易地为人们学习和采用？我相信这是一个真正需要开发的未知领域。Larman 的 GRASP 模式是一个好的开端，然而却没有人将其深入下去。

Rebecca Wirfs-Brock：你关于「将我们的协作策略重新映射（recasting）成模式」的建议让人很感兴趣。Larman 的 GRASP 模式（General Responsibility Assignment Patterns，通用职责分配模式）给出了非常直接的标准来分配对象的职责：其他对象的创建，事件处理，对象相互之间的解耦，等等。所有这些基本的模式都是好的设计原则。我尤其喜欢看到把 GRASP 模式映射（cast）成一系列的问题。对于把设计原则（特别是对象间的协作设计）提炼成一组设计的提问而言，尚有许多要说的。我认为在我们书中的论述里面潜伏了多个有关形式良好（well-formed）的协作，典型的角色构造型间的协作，恢复（recovery）策略，以及可信任边界（trust boundary）的模式。所有我现在要做的就是提出便于记忆的模式名称，一组明确的用于提问的问题，以及需要权衡的约束力（tension）。

灵活性

SZ:有关灵活性的章节写得非常好。你能说明一下如何能将灵活性设计到软件中去呢？你与客户在灵活性设计时有哪些成功的案例呢？

Rebecca Wirfs-Brock：谢谢。写这一章节时给了我很多的乐趣。灵活性既不容易进行规定，也不容易很好地进行实现。我有许多与客户一起的成功案例，帮助他们一起把所需要的灵活性点（point of flexibility）整合（连接，articulating）起来，然后设计出一个合理的解决方案，使其能够对他们软件的能力进行扩展，从而具有伸缩性。

关于正确的灵活性数量构建的关键主要包括三点：理解需要精确进行伸缩的是什么，理解对软件进行适配（适应调节，make adaptation）的是谁（是最终用户、开发者，还是某些能修改数据描述信息的实施人员，并且软件被设计用来读取和解释这些数据描述信息？），然后给出这些适配正确的“转动把手”，使软件能够伸缩自如。有时候你可能需要构建工具和设计额外的软件装置（fixtures）来强制人们进行适配，有时候你可能必须编写处方来说明如何进行扩展。

灵活性的获得并非没有代价，只是因为你正在使用对象技术，或者应用设计模式。这总是包含着额外的代价，而且所引入的灵活性常常会使软件变得更加复杂。因此我可以根据实际需要进行填充，并提出最简单、最具成本效率的“伸缩点（flex point）”解决方案，从而得到非常满意的成果。在这个世界上人们不能承担太多「将错误的灵活性构建到系统中」的“冒险”，这对于了解「构造一个系统并设计合理的解决方案所花费的代价」的确非常重要。

SZ:我们的读者喜欢现实生活的例子。你能给出一些你帮助客户开发灵活性方案的具体细节吗？

Rebecca Wirfs-Brock：我为一家电信公司作了两年多的关于应用集成框架（application integration framework）的架构师、设计者和导师。该系统在 GemStone/J（一种 Java 应用服务器）中进行实现并被投入生产。该系统被设计成能够灵活地集成结账（billing）、订单（order taking），供应（provisioning）和供应规划（provisioning planning）应用。

其基本思想非常简单：这些每个应用彼此之间无需直接相连，我们的软件则位于其间。通过使 telco 能够集成新的应用以及支持更加完善的业务集成，来完成提供业务价值的目标。我们必须要对付多种结账系统，多种订单和客户服务应用，以及不同的供应系统。

作为该项目的一部分，我写就了一份详细的“热点（hot spot）”文档，其描述了我们想要使软件能够伸缩的主要方面。这仅仅只是个开始。在系统中有许多灵活性点。但是主要的出发点则是整个架构。系统被划分成多个与系统“核心”进行交互的适配器（adapter），它们的主要职责是协调每个被集成应用中的工作（如订单接受和处理的结果）。每个适配器负责传递外部应用与核心部件之间的请求和信息。

我们定义了一个通用机制，来定义与下订单、修改订单和取消订单相关的业务处理任务。我们开发了一个声明式（declarative）的商品和结账模型。我们定义了一个通用的命令集来与系统的核心进行通信。我们定义了对那些外部应用所维护的资源进行定位的机制。大多数时间中，我们没有可依赖的既有架构模式。因此，我们在实现我们设计目标的过程中必须要有所革新。在多个设计领域工作会为你带来许多乐趣。更为重要的是，能够成功地进行产品化。

UMLChina 公开课

“UML 应用实作细节”

（2003 年 11 月 15-16 日，上海）

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>>

UML 相关工具一览：U-X（截止至 2003 年 10 月）

整理

查看评论

接 29 期…

工具（最新版本）	厂商	试用允许	UML 支持								支持代码环境	XMI	平台	备注
			用例图	类图	状态图	活动图	顺序图	协作图	构件图	部署图				
UML Diagrammer 4.15	Pacestar Software http://www.pacestar.com/uml/index.html	30 天试用	√	√	√	√	√	√	√	√		√	Windows	
Umbrello UML Modeller 1.1.1	Umbrello Project Team http://uml.sourceforge.net/index.php	开源	√	√	√	√	√	√				√	Linux	
UML2COM	Arion（希腊） http://www.arion.gr/uml2com/index.htm	有试用版		√							C++, VC		Windows	Rose 插件，把 C++ 或 VC 代码转成

														COM/COM+组件
UMLAUT 1.8	Triskell Project（法国） http://www.irisa.fr/pampa/UMLAUT/ 	免费									Eiffel, CDIF, OCL	√	Linux, Solaris, Windows	一个研究项目的一部分，支持许多新特性。可以读取mdl，可以作模型检查。
UMLGraph 1.2.4	Diomidis D. Spinellis http://www.spinellis.gr/sw/umlgraph/ 			√							Java		Java	
UmlNICE 1.0	Intecs Sistemi（意大利） http://www.etruscan.li.it/UmlNICE/HTML/features.htm 	有试用版	√	√	√	√	√	√	√	√	IDL, Java, Ada	√	Java	
Unimodeler 1.4	Unimodeler http://www.unimodeler.com/ 	免费	√	√	√	√	√	√	√	√			Linux	
UML Pad 1.15	Luigi Bignami（意大利） http://web.tiscali.it/ggbhome/umlpad/umlpad.htm 	GPL		√	√	√	√						Windows	
VB CASE	VB CASE Project http://www.quitetheberries.com/vbcase	开源									VB		Windows	专用于 VB 的 CASE 工具。现在

	/													已经停止开发。
Visible Analyst 7.6	Visible Systems http://www.visible.com/Products/Analyst/vaooedition.html 		√	√	√	√	√	√						
Visio 2003	Microsoft http://www.microsoft.com/office/visio/ 		√	√	√	√	√	√	√	√	IDL, C++, C#, VB	√	Windows	微软的主力 UML 工具
Visual Classworks 6.10	Step Ahead Software（澳大利亚） http://www.stepaheadsoftware.com/vcwf.htm	可以试用		√							C++		Windows	
Visual Case 2.6	Artiso（加拿大） http://www.visualcase.com/ 	30 天试用	√	√	√	√	√	√	√	√	C#, VB, Java, Access, Pervasive SQL, Interbase, PostgreSQL, Oracle9		Windows, Linux, Solaris, Mac OS X	强力的 UML-数据库双向工程。
VP-UML 2.2	Visual Paradigm http://www.visual-paradigm.com/vpuml.php 		√	√	√	√	√	√	√	√	Java	√	Java	支持用例文档，文本分析。
Visual UML 3.0	Visual Object Modelers http://www.visualuml.com/products.htm	30 天试用	√	√	√	√	√	√	√	√	IDL, C++, Java, C#, Visual FoxPro, VB, VB.Net,	√	Windows	有 VS.NET2002-2003 插件。支持 GoF

	Visual UML® Object Modeling using the Unified Modeling Language (UML™)										Access, SQL Server, SQL Anywhere, Oracle, MySQL, VS.NET, VBScript/JScript, VBA, UML Web扩展			模式。
Wilde 1.0	WildeTechnologies http://www.wildetechnologies.com/products_intro.html WildeTechnologies evolution by design™										XML Web Services, .NET components, COM components.		Windows	基于 UML 的可视化组件装配
WinA&D 3.5	Excel Software http://www.excelsoftware.com/quickumlmac.html Excel Software Software Engineering Made Easy			√	√	√	√	√	√	√	C++, Java, Delphi		Linux, Mac, Windows	支持 CRC
WithClass 2000 v6	MicroGOLD http://www.microgold.com/ The Design Tool Company MicroGOLD		√	√	√	√	√	√	√	√	C++, Java, Delphi, VB, IDL, Perl, PHP, C#, VB.NET, VBA, PHP, ODBC, Smalltalk, VDHL		Windows	
XCoder/J 1.03	Liantis（德国） http://www.liantis.com/Downloads/index.html										Java		Java	支持极限建模

	LIANTIS													
XDE	IBM Rational http://www.rational.com/products/xde/xdedev.jsp Rational. software	可以试用	√	√	√	√	√		√	√	C#, Java		Windows	完全整合到 VS.Net 及 Websphere Studio J2EE 平台中。支持模式。

征稿

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

The Addison-Wesley Signature Series

Patterns of Enterprise Application Architecture

中译本即将上市

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

UMLChina 负责中译本最后审校，并指定为训练用教材

基于职责建模（上）

Alistair Cockburn 著，李巍 译

查看评论

这是 Humans and Technology 技术备忘录 HaT TR.99.02 (dated 99.03.11)的一份非正式草稿。以这种形式用于同行之间进行交流（peer communication）。欢迎你阅读本文，并将所学到的运用于你的项目中。或许你记下了指向它的一个链接，但要知道链接可能会变化或消失（就像任何的 web 链接那样）。本页面是由 Word 自动生成的，因此它看起来并不那么优美。

职责（Responsibilities）是陈述系统设计基本原理的一种方式。整个系统的职责鉴别和分配是业务模型和软件的主要设计活动。作为一个主要的活动，职责鉴别和分配会相互密切追随并伴随着既有组件的重用。在面向对象的建模和设计中，职责的定义和分配是同时进行的，与其他的方法不同，例如结构分析（structured analysis），它们不需要在定义时进行分配。

人们看上去已经很自然地准备好了与职责及其分配一起共事，或许这正像我们的社会所构建的那样。J. Fisher 博士，Towson 州立大学的退休教授写道，“一个理性的社会，会使其成为一家公司和一个国家，如果以个人的职责和义务（accountability）为核心，则仅需要维持其自身；即从上到下，必须使每一个主体或公民能够主导她（他）的角色并对其行为负有全部的责任。”我怀疑生活在这样社会中的人们会不断增强对职责分配出现分支时的敏感度。不管什么原因，在不同国家教授面向对象的个人体验为那些地方的每个高级设计者所支持：并非每个人都能够很好地创造和分配职责，但是大多数人能够迅速地与下列问题联系起来并回答它们，

1. “该对象的职责真的是处理这个请求吗？”

2. “它的职责是要知道所有的信息吗？”

甚至对象设计的初学者都能快速地感觉到所提议的职责分配是合适的还是让人感到不舒服的。这种感觉非常适用于评估富有经验的对象设计者。我们似乎做好了准备来说，“追踪那个信息并非我的职责。”或者，“是的，处理该请求是那个组织的职责。”

在这里描述的基于职责建模（RBM, Responsibility-based modeling）在本质上与职责驱动设计（RDD, Responsibility-driven Design）是相同的，RDD 由 Rebecca Wirfs-Brock 描述并使用 Beck 和 Cunningham 的 CRC 卡技术。第一点不同之处是在 RDD 中专注于创造软件类，而 RBM 则工作自既有的业务事实和模型。第二点是 CRC 卡已经作为一种特定的预排（walk-through）和大脑风暴技术被分离出来，其与模型中职责的讨论和分析明显不同。进行 RBM 需要观察基本的业务关系，或业务模型，或一个对象的实例图，或一个对象的交互图。

尽管基于职责建模在最初的文章和图书中都进行了很好的描述，但开发者一致评论需要对象间进行交互的文档，以展示职责的递交（hand-off）。职责分配是一个设计的方法，而交互图则归档了设计的成果。幸运的是,大多数的 OO 方法论支持以某种方式对交互进行归档，可通过交互图，对象图，或事件追踪。Jacobson 提供了很好的关于交互图的介绍和讨论。

基于职责建模不只适于软件类的设计，同样能够很好地应用于一个系统的子系统划分。子系统正是一种类型的一个对象，往往是其类型的唯一实例。基于职责建模适当地推迟了对子系统内部结构的专注，并将主意力集中在它的角色以及与其合作者之间的交互上。

原则和假定

该方法的指导原则是通过询问每一部分的职责使之成为整体，来捕获围绕着“如何对系统进行划分”这样一个中心问题。该问题会涉及到功能，功能分布，通信，控制的局部性（locality），与变化相关的健壮性等问题。

该方法的成功依附在人们能够直觉地对职责分配进行有意义的价值判断的假定上。似乎应该维持该项假定。不同层次上的面向对象初学者，程序员和业务人员，已经接收了 CRC 的训练。他们对是否能够简洁地描述和放置一个职责的直觉评估可以非常好地与富有经验的 OO 设计者的评估相符。人们看上去受过了良好的社会训练来回答如下两个前面所提到的问题：

- “该对象的职责真的是处理这个请求吗？”
- “它的职责是要知道所有的信息吗？”

这些是设计者将会不断询问自己的问题，以作为他们对划分进行评估的一部分。这并不意味着设计初学者将会发现合适的起始定位（placement from the start），但是当提出两个设计时，他们将更应该从他人那里提高自己。这样的能力可使一个人不断地找到机会去提高自己。

职责可作为子组件的需求。该方法的原则是设计者负责确定它们是已经存在还是能够被构建。实际上，设计者说，“如果我们拥有的这些组件具备这些能力（capabilities），我们便能交付该项功能。我们会证明这样的一些组件或者已存在或者能被构建。”。

该方法所解决的设计自由度

(1) 来自所构造的系统的组件。

(2) 它们所提供的职责和服务。

(3) 它们满足用例中所陈述的需求的方式。

系统开发到现在，需求已被表示成用例，基本业务关系，和一个候选业务模型。这时便可以检查、发现或创造大部分的健壮组件，这些组件能够协同进行工作以交付用例所要求的行为。

团队负责研究并最终知道对设计有用的素材。其包括任何已有的业务或数据模型，陈述业务规则的文档，设计模式，框架，和程序组件。业务或数据模型提供了组件的候选名称，并规定了基数关系（cardinality relationships）。业务规则提供了有关协作的信息和可能会发生变化的区域。设计模式提供了来自前人设计的思想。框架和程序组件提供了既有的工件，这些工件能够简化必须做的新工作。这是为了对团队进行划分，以确定哪些新的组件会被创建并被引入到系统中来，以及尽可能地简化设计或设计任务。

基于职责建模是一项递归的方法。团队将很有可能会创建一组需要进一步划分的组件。这些组件可能会由同一个团队或另一个团队进行划分。每一个团队需要对质量进行负责，包括他们的组件协同工作方式，以及他们简化、简易或保护设计的方式。

根据职责进行设计的讨论

该方法使用五种活动：准备（preparation），创造（invention），评估（evaluation），合并（consolidation），文档（documentation）。

在准备活动中，将会收集用于设计会议期间的用例。在会议开始之前，团队就已经确定将会设计系统的哪一部分（限制设计活动的广度），会涉及到哪一个层面的设计（限制设计的深度），以及需要哪些用例来涉及（考虑）在这样的广度和深度下的系统设计。

在创造活动中，会提出（形成）对象的类型，以及任意地命名组件，这是由于看起来它们有助于场景（scenario）的实现。可以临时性地进行职责分配。名称可以进行改变。所命名的组件要比最终用到的多。有时组件会从其他层面上获得名称并一直用到层面的差别被发现，它们将会被放到一边，留待以后使用。

在评估活动中，为了着重对设计进行测试，会提出（显现出）一系列的问题和场景。显现的问题用以检查有效性，组件的命名和长期的可用性。显现出的备选场景为“变化分析（variation analysis）”，有时候称为“健壮性评估（robustness evaluation）”。对于需求和实现技术的假定是千变万化的，这要看有多少设计会因此改变来适应它们。如果所需的变化能够被限定在很少的组件内，便认为这是一个好的设计。

在合并活动中，会收集在前两个活动中留下来的组件并对它们的名称和层面（级别）重新进行检查。标记来自较低层面的组件并将它们放到一边留待以后使用。检查名称的意义性，稳定性和助记价值。

在文档活动中，会记录下创建职责的一个具体划分的原因, 连同演示该职责划分使用的场景。仅使用位于设计级别的组件，绘制这些关键场景的交互图。标识已经存在的组件；详细说明需要进行设计的组件。

设计会大致依次使用这些活动。必须在设计被完全考虑之前使用所有这五种活动。也的确会存在未按次序使用活动的情况。也就是说，通常会在组件创造期间引入新的场景，进行变化分析以在两种设计选择中做出决定。只要有人发现了一组位于不同层面上的组件，便可进行合并活动。只要团体感觉有必要，便可对一个具体职责分配的原因进行归档，以使该原因不被遗忘。有些人喜欢立刻对交互进行归档，而另外的人则倾向于当设计稳定后再去进行。在所有情形下，出于对完整性的考虑，合并和文档都必须在最后进行检查，以确保团队达成一致。

使用术语

系统：设计下的系统。该术语能够指代整个应用程序或主体可交付部分；它能够指代主体可交付部分分解而成的其中一个组件；它能够指代一个子系统。它能够指代一个人民团体，计算机硬件，或软件，或一个组合体。无论如何，它是能够被进行设计的东西。唯一的要求是系统要由多个通信部件组成，因为职责驱动的划分（responsibility-driven partitioning）需要与消息和系统各部件间发送的信息协同进行工作。

组件（component）：任何组成系统的部件都会与系统的其他部件进行通信。组件可能是一个具有自主能力的系统，一个人民团体，一个人，计算机硬件或软件。它可能是一个类型，一个面向对象系统的类或对象。它可能是一个购买的厂商包，必须集成到系统的其他部分中。它可能是一组操作系统服务或一个数据库。它可能是一个没有采用面向对象技术进行设计的程序。在本文中，*组件*被用作以下陈述：一个系统被划分为多个组件；该技术展示了如何来详细说明构成系统的这些组件。

职责（responsibility）：一组约定的服务；在一个系统中组件的角色。职责是一个组件对系统所做的贡献，因为它们是系统设计者和使用者依赖于组件所完成的服务。组件的角色概括了它在系统上下文中所提供的服务。

能力（capability）：提供一组服务的可能性（possibility）；一种从上下文中提取出来的职责。在一个运行系统的上下文中，每个组件都具有一种职责来实现完整的功能。当组件被放进组件库中时，这些职责将会脱离上下文而存在。对于后来的设计者而言，它们看起来就像组件的能力。后来的设计者将会考虑这些能力如何在设计期间充当系统上下文中的职责。在本书中，为了保持一致性和简单性，会尽可能地使用单词“职责”。单词“能力”则用来指库中的组件，它脱离了一个具体设计的上下文。

“基于职责建模”。该项技术的别名。基于 Wirfs-Brock 书中的“基于职责设计”（RDD）技术，在某些方面进行了修改，并在该技术多年经验的基础上进行简化和引申。术语“职责驱动（responsibility-driven）”描述了激发（驱动）原则，即对指派到系统组件的职责进行分配和评估。于 RDD 不同，这里所描述的这项技术没有使用“契约（contracts）”，没有对属性（使用标记“联系点”代之）进行讨论，以及没有规定继承的层次（使用委托代之）。

CRC 卡：组件－职责－协作者（Component-Responsibility-Collaborator）卡。在 4"x6" (10cm x 15cm)空白索引卡上可以书写组件的名称、职责以及为完成职责所必须进行协作的组件的名称。字母 CRC 是专为用于类层次上的面向对象设计而发明的，因此 CRC 字面上是指“类－职责－协作者（class-responsibility-collaborator）”。我们正是在系统分解（system decomposition）的更高层面上使用 CRC 卡，因此这些字母意为，“组件－职责－协作者”。

层面（level）：在某一点上对设计进行讨论的关注类别（nature）。如果一个组件具有与此刻所讨论的类别相应的职责，那么该组件在这个层面上是可见的。实现该设计方法的主题之一就是关注层次（级别），追踪与当前所设计的层次相关的组件，以及将其他层面的组件放到一边。

变化分析（variation analysis）：在该设计阶段中，假设需求和实现技术是变化的，从而看看设计须要进行多少改变才能处理这些变化。

职责讨论

有两种类型的职责，*做*某事的职责，和充当一个信息接触点的职责，实际上是作为信息传递的中介。在第一种情况下，使用动词的主动形式来描述职责，并且职责的含义非常清晰。第二种则需要进一步进行论述。

通常一个组件负责向其他组件提供信息，并保持信息的流动。这是一个有效的信息“联系点（contact point）”，并且其职责就是维持一个有效的信息联系点，无论信息如何演变，以及无论设计者最终决定是对其进行存储还是计算。

有些参加设计会议的人们会思考一个组件所保持的“状态数据”，或一个实体所具有的“属性”。对于基于职责建模而言，这些思考都是不适当的，因为它仅关注组件所相互提供的服务。可将属性和状态数据对等地考虑成一个联系点。

主动和联系点职责之间没有外在的不同。核查账号组件（checking account）可能具有“知道账号余额”的职责。此外，其可能具有“知道如何获得账号余额”的职责。第二种职责可能向人们意味着其没有存储一个信息流的拷贝，而是计算得出；而第一种则向人们意味着其存储了一个信息流的拷贝。事实上并没有分别，因为可以通过对其进行计算而“知道余额”,或者通过对其进行本地存储而“知道如何获取余额”。这种差别相当于直接属性（direct properties）和派生属性（derived properties）的差别。

淡化“知道”和“知道如何去做”之间的差别是经过深思熟虑的。确定实现职责的方式是在一个单独时间上涉及到的设计决定，该决定还会涉及到心中其他的设计考虑。

当设计团队熟练运用职责时，他们能够通过仅仅写下最重要的或摘要性的职责，以及当他们最终归档组件时来填充剩下的部分。但是，在刚开始时，团队可能想写下每一件事情，而不至于遗漏。保持顶层上的主动职责则是很有帮助的，因为它们是对系统进行划分的关键。

一种职责通常包含其他的职责，因为一个服务可以包含其他的服务。

从该技术应用的观点来看，概要或详细的职责列表都可用于进行划分。最初，设计团队可能希望与详细的服务列表一起工作，以确保不会出现缺口。最终，他们应该与概要报告（陈述）一起工作，因为这会更加快速。最终将会做出详细的服务列表。

例如，一个银行账户可能具有处理和追踪所有与该账户相关交易的职责。个别服务包括添加一次交易，删除一次交易，创建一次交易来处理月费用或月利息的计算，等等。在职责划分的过程中，“处理和追踪与该账户相关的交易”将更加易于书写、阅读以及与其一同工作。能够随时创建完整的列表。

组件讨论

组件的创建通常是两种事物之一的一个断言（assertion）：

(1) 组件表示可被业务追踪和管理的事物，

(2) 组件是一个设计变化（design variation）的点，负责捕获可能存在的多个解决方案的共同特征。

如果业务管理“客户”和“订单”，那么则需要出现 Customer 和 Order 作为某个设计阶段上的组件。缺少它们意味着设计是不完整的，意味着它们将出现在其他级别的讨论中，或者该术语恰恰是由业务所实际管理的某个其他事物的一个别名。

组件经常作为一种可能性选择的占位符而被创建。它如此常见，以至于当对设计进行检查时，发现对于每个组件会存在一种可能的备选实现。基于职责建模和面向对象实现的优势之一就是组件可以表示一组能够隐藏多种实现的服务。将这些服务定义成组件使得设计者能够随时改变实现，而无需修改服务的客户端。即，组件作为设计变化点（*point of design variation*）。

不能过分强调组件作为设计变化点的价值。检查一个组件，看看其所代表的类是不是有可能随时进行实现。另外，建议设计者只要在讨论需求或实现可能出现的变化中陷入了困境，就考虑创建一个新的组件。新的组件将会表现所需服务的特征，并获得决定能够发生变化的地点，提供将来的安全并使得设计会议得以继续。

关于“组件”的最后一点。该技术主要以面向对象的设计为目标，其中最终设计会被实现成类以及这些类的实例。这样的一个实例便是组件，而类则定义了这些组件。然而，并不是所有系统中的组件都会作为这些类的实例。位于最低级别上的一些组件可以是程序设计，操作系统或网络或数据库服务。在该方法最后，必须对它们进行详细的说明。在最低的级别之上，组件可能是一个必须要一起进行工作的对象类型和实例的集合。在这里，该集合可被视为一个单独“事物”，一个组件，必须要对它作进一步分解。任何种类的框架都可作为一个组件来看待；子系统可以是一个更大系统中的组件。如果项目不是一个应用程序的开发项目，那么组件可能根本就不会最终作为 OO 类而存在，而是人和程序的集合。基于这些原因，将会贯穿始终地使用单词“组件”，直到最后，将会提出这些作为类型和类型实例的组件的类型规约。

创建组件 vs. 重用组件

该方法致力于对既有组件的使用。只要所有场景都能够结合使用既有和新指定的组件进行交付，该方法便告结束。这称之为“协同重用设计（design with reuse）”（与其相对比的是“面向重用设计（design for reuse）”，它希望其他的一些项目将会用到该设计的结果）。以下对于发现可供使用或重用的最佳组件是非常有用的：

使用该方法，设计者的职责就是识别可供使用的既有组件的最佳组合。

获得真正的效率来自于对既有组件的使用。在此处，设计团队的职责是要知道当前可用的组件类型。有时候可以找到能基本上满足需要的组件。直接使用该组件，还是创建一个使用它的新组件，或是不使用它，这是一个设计问题。

发票（invoice）这个对象类型是作为表示系统中的一个业务工件而存在。账户日记账（account journal）则用来记录在该账号上所发生的交易。甚至可以根据用途来命名一个 ordered collection（有序集合），其存储了多个次序排列的组件。它提供了与其用途相适合的服务：first（第一个），next（下一个），last（最后一个）等。根据用途对每个对象进行命名。然而，在一个设计场合的上下文中，组件的名称可以反映其此刻所扮演的角色，也可以反映其所提供的一些能力。“Journal”，“strategy”和“broker”命名了角色。“Fraction”，“ordered collection”以及任何其他集合，或者“window”则可能会被用作能力的集合。每种对象类型都具有自己的价值。

在从组件的角色方面描述一个所需组件的名称时，需要寻找已经具有这些职责的既有组件。从而能够以直接或间接的方式使用既有的组件。

根据能力命名的组件可被用于多种意图。这种方式效率非常高。但是，其名称有可能无法准确地与所需的抽象进行适配，因此，尽管它有非常充分的理由作为数据存储的设备，但其对存储需求的变化却很敏感。如果它们发生变化，那么不但其本身将会变化，而且依赖它的客户端组件也可能必须要变化。因此，其对于变化是脆弱的。为了补偿这种脆弱性，设计者必须将合理抽象所具有的一些职责赋予用来进行数据存储的对象。这样使得客户端看不到数据结构的变化（示例见后）。

另外一个选择是为所需的角色专门添加一个组件。其将会与所需的抽象进行适配，并且将封装其各种实现选择，从而当存储的需求变化时，客户端可由已定义的接口保护起来（这对于变化是健壮的）。然而，新的组件往往具有较差的重用性。它精确地以其任务进行命名，并且不太能够适合其他的情况。同时，设计者在系统中引入一个新的组件增加了系统的复杂度。必须要对这个新组件进行设计，测试，归档，维护，后来的设计者也要学习它。

在「根据能力来重用一个既有的组件」和「引入一个新的、专门的组件」之间的权衡，必须要由所划分的团队来管理。比较「修改客户端代码的成本」和「向系统引入一个新组件的成本」，从而做出选择。这使得成本的对比更加清晰。

一个示例，两种结果：

账号需要一个日记账，其具有的职责是添加和删除交易，以及作为交易历史的中介。团队命名一张卡片，“Journal”，然后确定可直接使用一个既有的对象类型，“OrderedCollection”。

结果 1：重用。

该应用程序并非用于银行。账号是系统中一个相对较小的部分，而且仅在少数地方使用账号和它的日记账。所有日记账需要的服务由外围组件 account 产生。账号将会打印日记账，查找某一确定日期之前或之后的实体，产生一个日记账的拷贝，等等。没有为 Journal 创建新的组件。将 Journal 卡注释为“OrderedCollection”，表示其已经存在。

结果 2:创建（与重用一起)。

团队确定日记账的概念对于业务而言非常重要，将会大量地用到它，而且日记账的表现形式可能随时需要变化（出于性能的考虑或是由于添加新的职责）。他们创建一个名为 Journal 的组件，并列举其服务：以各种方式表现其本身，查找某一确定日期之前或之后的实体，等等。对象类型的设计者最终会考虑需求并且确定将使用 OrderedCollection 来保持数据（初步地），然而这是以后的一个设计问题。

通用提示

如果组件是一个模型中的主要概念，被广泛地使用，并且有可能发生变化，则需要创建一个新组件，并使一个当前可用的组件实现它。引入一个新组件的成本可能会低于修改客户端代码的成本。可将既有组件标识成一个协作者。

如果新组件所需的仅是一个既有组件能力的子集，最好是将其作为一个单独的组件分离出来。既有组件提供的额外服务有可能会危害到所需要的组件。新组件能够隐藏既有组件接口中不适当的部分。可将既有组件标识成一个协作者。

如果组件不是一个模型中的主要概念，并且没有被广泛地使用或者发生变化的可能性不大，则直接重用这个以能力而命名的组件。可以考虑将一些职责移到外围的组件中去。

级别和子系统的讨论

由于一个组件将会经常包括其他的组件，控制哪些组件需要讨论，哪些组件不需要讨论是非常重要的。想想一组对用户输入的文本进行读取和解释的组件。在某个讨论级别上，这组组件是一个单独的事物，一个低级别组件的子系统。在这个讨论级别上，足以能够将该集合作为一个组件来看待，并且讨论该组件在整个系统中的职责。如果出现了一些关于「是否能够实际设计出这样的系统，或者其性能」这样的问题，则可以将该组件展开成其多个子组件，并进行检查。检查之后，将子组件一起隐藏或折叠起来是非常重要的，并且它们会重新作为单独的一个组件与子系统一起工作。使用该方法的关键是要对积极讨论下的组件数量进行管理。

子系统是任意具有统一意图的组件集合。通过对重叠的组件进行分组，不同的子系统可以不同的方式来切分系统。例如，“用户界面”组件构成了一个子系统。“网络”组件同样如此，并且可能与用户界面组件不相交。但是，“客户”组件很可能会具有一个用户界面组件，一个主动处理组件（active processing component）和一个数据库组件。客户组件是一种子系统。它和其他业务领域的对象，会与用户界面和数据库发生重叠，或许还有网络组件。基于讨论的级别不同，“客户”可以作为一个单独的组件来对待，或者是被划分成一组组件。重要的是要清楚讨论的意图，并且在与讨论相关的合适级别上，只将那些组件拿到桌面上来。

与不同级别以及子系统工作的结果是，有时候会出现两个组件具有相同的职责。两个组件工作在不同级别上，外部一个，内部一个。在外部级别上，内部组件为不可见，因此不能对其发送消息。对于其他外部级别的组件而言它是不“可见的”。因此，外部级别的组件扮演的是一个守护者（gatekeeper），或者是该职责的联系点。然后它只是将职责委托给其他内部组件。该种职责的委托（delegation）是适当的。外部组件充当一个子系统，作为一个位于外部级别上的单独组件而出现，并且还是内部级别上一组组件的一个成员。

考虑一下银行账户。它具有追踪其交易的职责。通过近距离观察，账号有一个协作者，“日记账”组件，其职责是对交易进行实际追踪。在将账号作为一个组件的讨论级别上，将日记账作为一个独立的组件进行讨论可能并不合适。事实上，关于日记账是不是一个完全独立的实体的决定，可能是一个随时变化的设计决定。为了保护这个决定，赋予账号具有追踪交易的职责。至于是账号本身对它们（交易）进行追踪，还是把它委托给日记账，则是其自己的设计决定，这在外部级别上是不可见的。

继承和多态的讨论

继承（Inheritance）

在基于职责建模中，可以只在局部上确定「是否使用继承」。在这个方法中它被推荐使用，并且会在以后最终确定组件时进行定案，或者在框架设计中也同样如此。关于考虑继承的推荐方式来自「基于职责建模」，而这依赖于相似组件上面所需的通用服务。

可将多个组件之间所共有的一组职责收集成一个新的组件，一个通用的版本。新的职责集合最终将会作为一个具有自我行为的单独类型的组件，并且不是一个提供职责的通用版本的组件。象任何其他组件一样，必须要评估新组件的稳定性以及所提供的功能。

如果还存在着通用组件（generic component），可将其造型（cast）成多种方式实现的一种。

专门组件(specific component)可以发送消息来请求通用的服务（委托），专门组件可以继承其服务（继承）。

继承是两个组件之间的一个重量级（heavyweight）关系。它并不总是一个作为「实现通用和专用组件之间关系」的最佳选择。而且也不是所有的实现技术都支持继承。因此，至于使用继承、委托还是某个其他的实现方法，都可以作为一个选择由组件设计者来确定。团队可以针对组件间的相似性准备一些有助于设计者的建议。

多态（Polymorphism）

多态是用来表达「两个组件提供相似服务」的 OO 术语，例如，订单项（order line item）具有一个数值，并且订单（order）本身也有这样一个数值。订单可被要求提供这个数值，完成这个功能它需要请求每个订单项的这个数值，然后将它们加在一起并根据税法修改总额，等等。“提供其数值（provide its value）”服务在订单和订单项之间是多态的。作为一个动词，“value（求值）”更是各不相同，因为许多不同的事物能够以多种方式具有和描述它们的数值。

多态可以对概念上的复杂度提供存储。同一个动词短语可用于多个组件来传达相同的意图，即使每个实现各不相同，这意味着需要学习更少的动词概念来理解系统的设计及其实现。在命名职责时，职责划分团队应该考虑多态的价值。

设计者的问题是要考虑服务的意图是否相同。在英语中一个合适的例子是“answer”，可以 answer the door，answer the phone，answer the letter，以及 answer the question。一个不恰当的例子是英语俚语“flog”，其既可用来表示鞭打，又可用来表示做广告；这两个具有不同的意图，因此应该使用两个不同的动词，如 advertise（做广告）和 whip（鞭打）。使用相同的动词短语能够带来好处还是造成混乱，通常是比较明显的，因此常识便已足够。

设计目标

1. 在组件中命名的抽象的纯度和简单性，
2. 使用既有的组件，
3. 防止遭受需求变化的影响，
4. 防止遭受实现技术变化的影响。

结合职责的场景

场景（Scenario）和职责分配紧紧地纠缠在一起。场景可以通过其目标来进行刻画，即基本 actor（primary actor）*想要系统完成的功能*。系统在自己的角度上承诺完成某些功能，这样便使得 actor 能够实现其意图。例如，银行雇员希望“登记客户的一次交易（register a customer's transaction）”。这是 actor 的目标。需求团队（requirements team ）也赋予系统具有「根据日期记录交易，以及更新和记录账户的余额」的职责。这些职责会在场景陈述（scenario statement）中看到。

在基于职责建模中，系统被化分成多个能够完成系统职责的组件。可以对组件之间的交互进行归档。对于每个组件而言，它和它的协作者之间的交互可以作为一个场景出现！也就是说，actor 请求一个服务或者发起一系列组件必须要做出响应的相关信息。在进行组件设计时，每个请求和消息序列将会被看作是组件的场景。

这种（场景 – 职责 – 交互）的循环本身便是一个不断具体和详细的重复过程，直至出现以下这些情况：

(1) 组件被发现能够完成职责。该组件可以是一个对象类型，一个外部服务（如一个操作系统，数据库或网络服务），一个复杂的子系统，或者甚至一个人民团体或人。

(2) 到达“对象类型”（面向对象中的观点）的级别。

(3) 到达「必须要设计和实现一个非面向对象服务」的级别。使用一个与实现技术相适合的设计方法来设计该服务，例如对由人所组成的组件进行组织设计（Organizational design）。

因此，场景和职责分配相互一起构成了一个完整的设计方式，并且根据设计的方法可自由地确定组件的名称，它们的职责以及它们相互协作来交付所需系统功能的方式。

交互图

交互图的功能等效物可以文字形式进行书写，绘制成一个水平箭头的列表，或者适用一个图形编辑器或专门工具来绘制。如果绘制的话，可以拓扑视图（topological view）或时间视图（time view）的形式进行绘制。

交互图描述了为解决一个特定情形组件之间进行交互的序列。在文本形式中，语句将以产生的次序列出，一个语句表示一次交互。在时间视图中，每次交互被表示成一个箭头，从「消息发送者」栏到「消息接收者」栏。根据发生的次序来列出交互。在时间视图中，能够显示出平行或无序的活动。在拓扑视图中，无论作者想用何种方法，都将组件布局在页面上。交互被显示为一个从发送者到接受者的箭头。必须对每个箭头进行编号以显示（交互）次序。

征稿

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

Agile软件开发丛书

PEARSON Addison Wesley

有效用例模式

Patterns for Effective Use Cases

Foreword by Craig Larman

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

UMLChina 指定教材

清华大学出版社

防火墙的模式语言

Eduardo B. Fernandez 等 著，yoochen 译

查看评论

简介

各种组织和个人对计算机信息安全的要求日益增长。因此，必须保护计算机系统免于攻击[Fer01b]。对于与局域网连接的计算机来说，攻击可能来自外部网络或是其他局域子网。保护局域网的一个普遍方法是建立防火墙，用来作为网关 [Zwi00]。

通过建立控制进入（和离开）局域网信息的扼制点来保障信息安全，防火墙被证明非常有效[Bar99]。因此防火墙限制未被认证的用户访问局域网，并限制局域网访问那些未被信任的外部站点。防火墙可以用作一种执行安全措施和决策的机制，并允许被保护的网络对外部开放有限信息。简言之，防火墙允许认证的通信，并拒绝未认证的或未授权的通信。

我们开发了一种模式语言来描述防火墙的功能。图1标明模式语言中的模式和模式之间的关联和依赖关系。地址过滤防火墙（Address Filter Firewall）定义了基于网络地址的基本过滤功能。基于代理的防火墙（Proxybased firewall）用于应用层，控制对应用的访问。基本防火墙和代理防火墙都可以用状态过滤（stateful filtering）来补充。基于内容的防火墙考虑基于文档内容的过滤。本文中，我们仅讨论地址过滤和代理防火墙模式的细节。

基本上，防火墙是一个单元或是一组单元，执行网络间的访问控制策略。基于防火墙的网络保护和安全策略实施的发展主要集中在建立适用于特定网络、操作系统和计算机的组件[Eps99, Hen01]。然而用于不同系统的防火墙的基本底层架构非常相似。

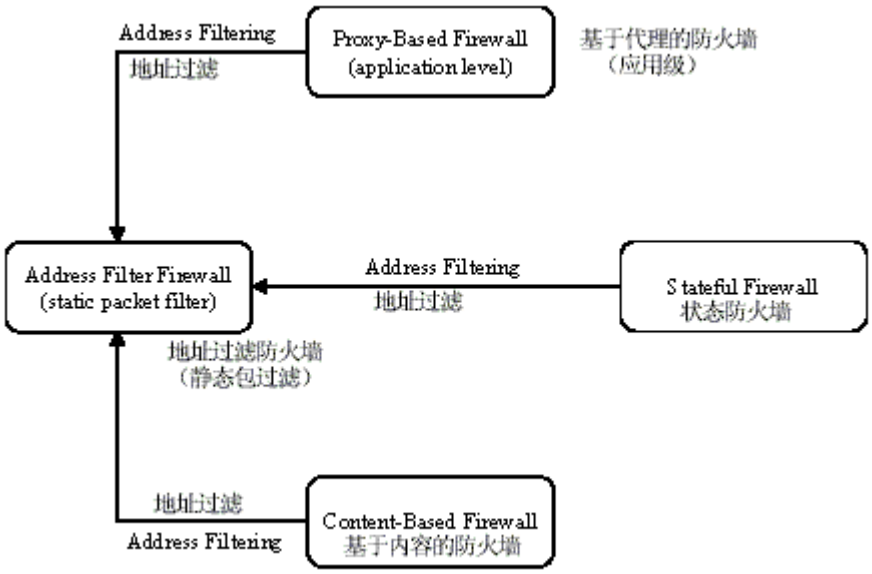


图1 ：防火墙模式语言

地址过滤防火墙（Address Filter Firewall）

目的

根据网络地址过滤流入流出的计算机系统的网络通信.

环境

与互联网和外部网络相连的局域网中的计算机系统

问题

局域网通常被外部（外部网络）攻击。局域网可以被分区，因此攻击可能来自其他局域网。

约束

- 以用户透明的方式过滤计算机系统的输入输出通信
- 网络管理员部署、配置多种防火墙；因此明了要过滤什么和怎样过滤的模型很重要
- 防火墙的配置必须反应机构的安全策略，否则，决定过滤什么很困难。
- 要过滤什么总是在变化：因此以改变防火墙的配置来适应过滤什么的变化要容易实现。

- 规则指定了允许，阻塞，抛弃哪种类型的通信。否则实现特殊的策略很困难。
- 为实现审核和防御，记录客户端请求日志是必要的。

解决方案

仅当存在批准来自客户端地址的通信的规则时，该客户端才可以访问局域网。因此，每个客户端与局域网的连接都被某个规则控制。防火墙由一系列访问规则组成，这些访问规则由机构（属于该局域网）根据它的策略制订。局域网可以有一个或多个防火墙。如果一个特殊请求未满足任何一个显式的规则，那么将应用缺省规则。

动态特性

我们用序列图描述基本防火墙模式的动态特性，序列图对应于两个基本用例。

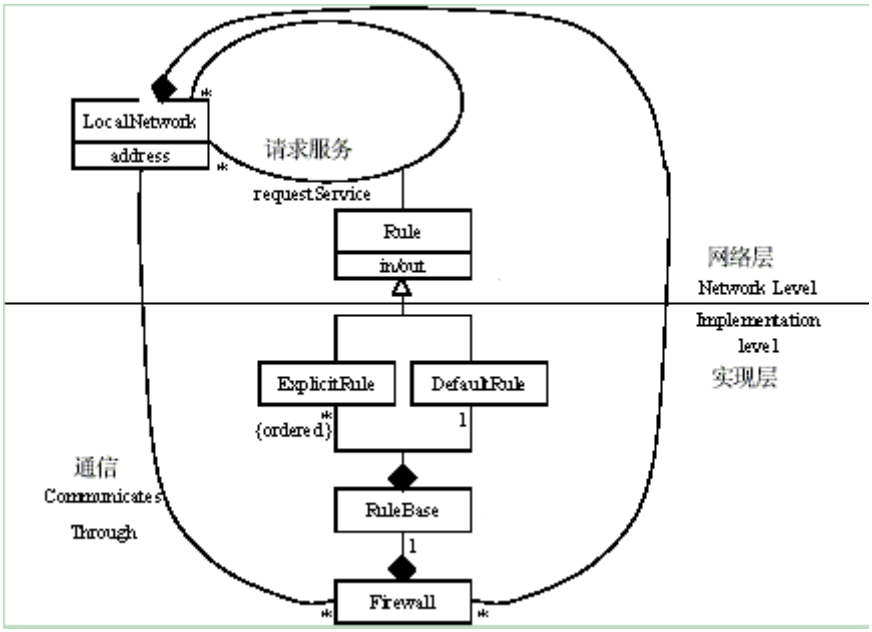


图2： 基本防火墙模式的类图

过滤客户端请求

- 摘要：远端网络要访问局网，不仅要传输而且要获取信息。该访问请求穿过防火墙，防火墙根据规则集合决定接受或是拒绝请求，即防火墙过滤该访问请求。图3说明了这个用例。
- 角色：外部客户。
- 前置条件：防火墙中必须有过滤请求的规则集。

- 描述：

  a) 外部网络请求访问局网。

  b) 防火墙根据规则集过滤请求。如果规则集中没有与之匹配的规则，将用缺省规则来过滤请求。

  c) 如果请求被接受，防火墙允许对局域网的访问。

- 分支流：如果请求被拒绝，防火墙拒绝外部网络对局网的访问请求。

- 后置条件：防火墙接受信任客户对局网的访问。

定义新规则

- 摘要：防火墙的管理员在规则集中增加新规则。防火墙检查要增加的新规则是否已在规则集中，图4阐明了这个用例。

- 角色：管理员

- 前置条件：管理员必须有增加规则的权限

- 描述：

  a) 管理员触发增加新规则

  b) 如果规则集中没有该规则，新规则被加入

  c) 防火墙接受新规则的加入

- 分支流：如果规则集中已有该规则，新规则不加入规则集。

- 后置条件：新规则被加入防火墙中。

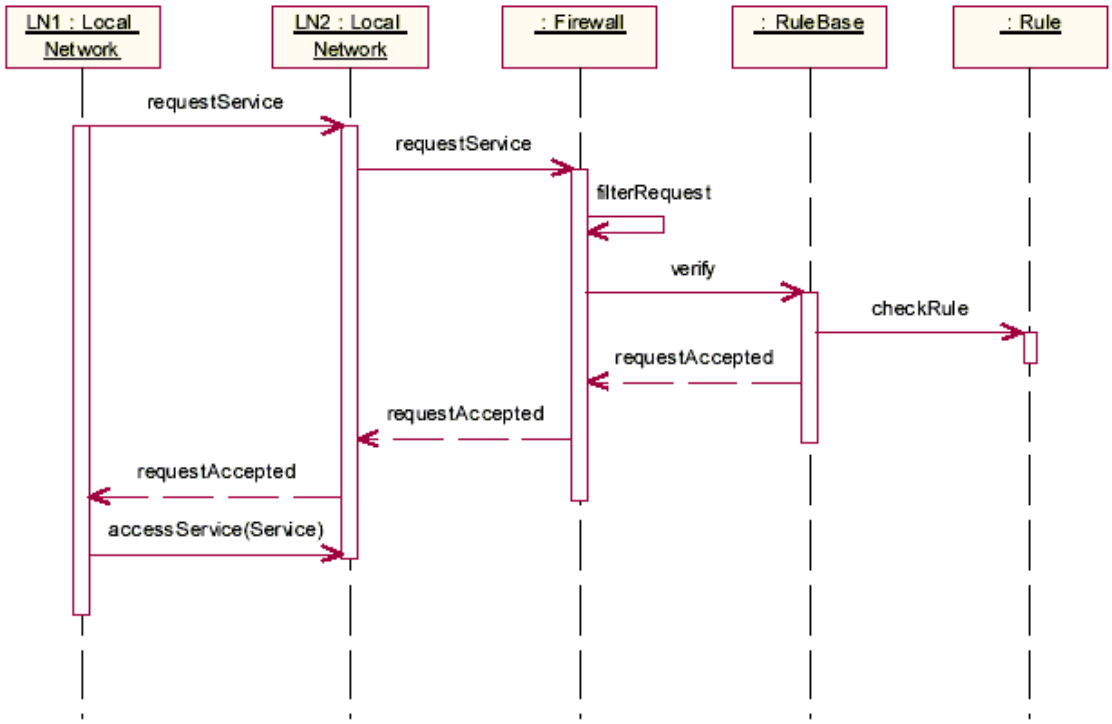


图3 ： 过滤客户请求的序列图

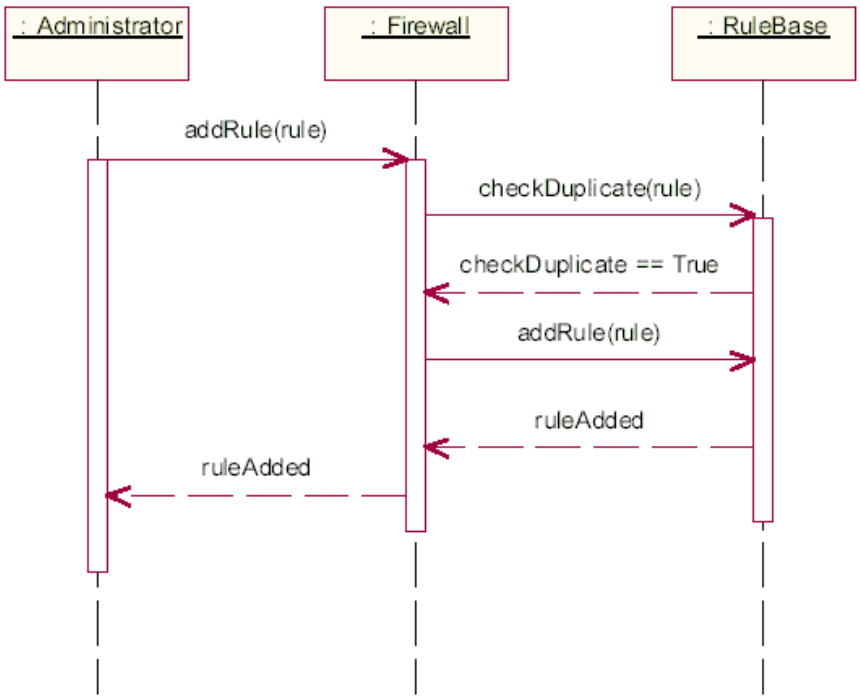


图4：定义新规则的序列图

结论

基本防火墙模式有一下优点：

- 基于网络地址，防火墙过滤所有经过的通信，并对应用透明。
- 可以通过防火墙的规则表明机构的过滤策略
- 防火墙有助于发现可能的攻击，有助于正式用户对其行为负责[Sch03]。
- 防火墙适于记录流入、流出消息的系统日志
- 成本低，许多操作系统将防火墙作为系统的组成部分
- 高性能，防火墙只需查看报文包头。

基本防火墙模式（可能）有以下不足：

- 防火墙的有效性可能受限于规则集（优先级）
- 防火墙的有效性限于局网的入口点，一旦潜在的攻击者通过了防火墙，系统的安全将遭到破坏。
- 基本防火墙仅对通过防火墙的网络通信实施安全策略。
- （基本）防火墙无法阻止更高层次的攻击
- 防火墙通常对被保护系统的可用性，性能和成本造成负面影响[Sch03]。
- 不同站点、网络和系统所执行的防火墙策略是不同的。新增的规则可能妨碍规则集中已有的规则；因此，必须小心谨慎地增加和更新访问规则。
- 非状态察觉
- 包过滤不能识别伪造的地址，因为包过滤仅检查IP包的报文头
- 黑客可以在不用于路由的报文头和有效负荷中加入恶意命令或数据。

已知应用

这个模型对应于基本（包过滤）防火墙架构，该架构用于商业防火墙产品，如：基于Network Associates' Gauntlet Firewall[Eps99]的ARGuE (Advanced Research Guard for Experimentation)；OpenBSD Packet Filtering Firewall [Rus02]以伯克利分布式软件系统（Berkeley Software Distribution system）的基本防火墙架构为模型；还有Linux Firewalls [Zie02]以 linux操作系统的基本防火墙架构为模型，基本防火墙模型被用作其他包括了更多高级特性的防火墙的底层架构，举例来说：CyberGuard [Hen01]，主要采用了全状态检测（stateful inspection）防火墙架构。

相关模式

认证模式[Fer01a]定义了基本防火墙模式的安全模型。基于角色访问控制模式，是认证模式的特殊化模式，适用于根据角色和权限定义网络和访问规则的情况[Fer01a]。防火墙模式是个特殊的Single-Point-of-Access [Yod97]的例子。另一个防火墙模式在[Sch03]中描述。

应用代理防火墙

目的

根据访问应用的类型，检查（过滤）网络的流入流出通信

环境

与互联网和外部网络相连的局域网中的计算机系统，需要比包过滤更高层次的安全

问题

地址过滤防火墙仅仅通过检查网络地址决定对消息的访问。然而，潜在的攻击可能包含在包的数据段中，这些包的网络地址被证实是可访问局网的[Sch03]。另外，地址过滤防火墙无法阻止IP欺骗(IP spoofing)。需要将局域网和外部客户网络虚拟地分开，才能彻底检查网络通信。

约束

- 地址过滤的约束仍是有效的
- 网络管理员部署、配置不同的防火墙：因此明了要过滤什么、怎样过滤和应用代理如何实现的模型很重要

- 防火墙的配置必须反应机构的安全策略，否则，决定在应用数据中检查和修改什么很困难
- 要过滤什么总是在变化：因此以改变防火墙的配置来适应过滤什么的变化要容易实现。如果要请求对防火墙不支持的代理的服务，那么将阻塞这个请求。
- 为实现审核和防御，记录客户端请求日志是必要的。

解决方案

客户只与请求服务的代理交互，代理再与受保护的服务通信。受保护的服务只与应用代理防火墙通信。仅有请求服务的应用代理时，客户才可以从服务器接收服务。每个应用代理都有自己预定义（由管理员声明的）访问和修正规则，这些规则被用于检查、改变并过滤流入（或流出）的消息。

图5说明了这个模式的类图，是图2的扩展。图5包括在每个局域网中现有的分散服务，和过滤请求服务的应用代理。

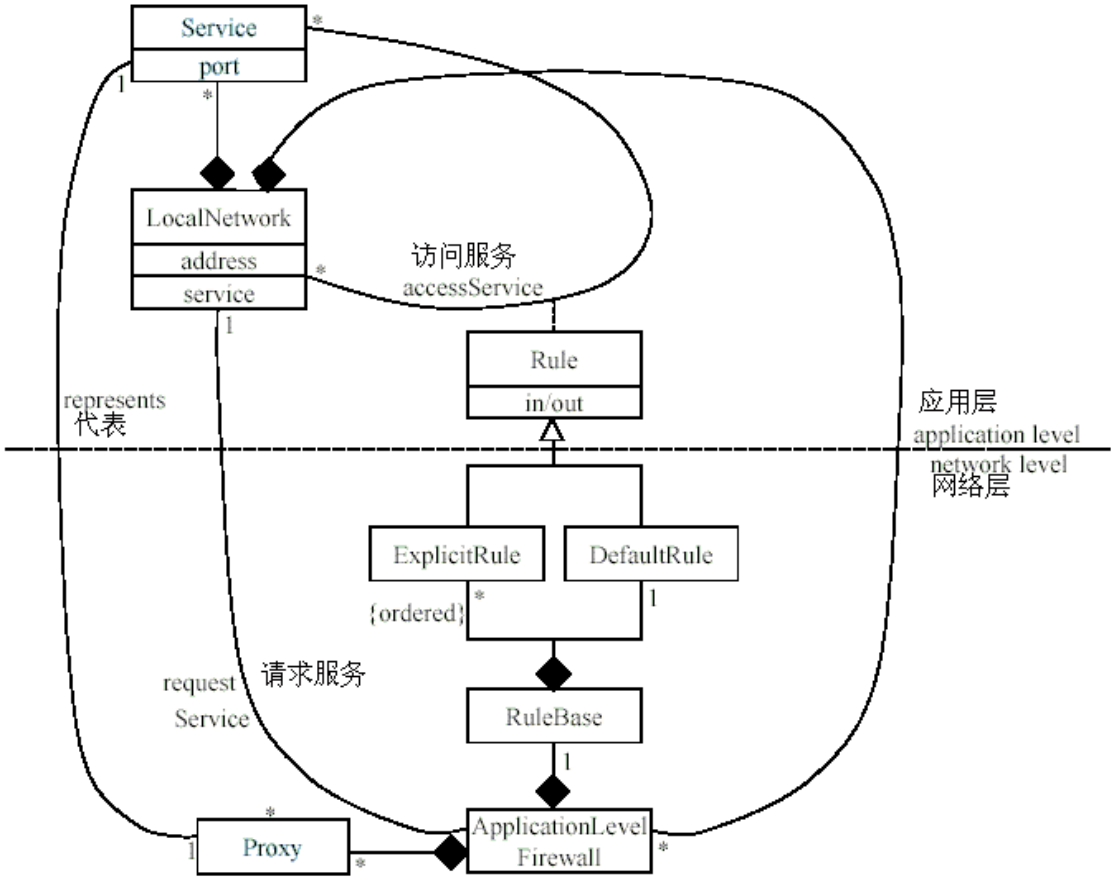


图5：应用代理防火墙模式的类图

动态特性

这里展示了一个过滤服务请求的用例。

为客户提供服务的用例

- 摘要：外部客户想访问本地服务器上的服务。访问穿过防火墙，防火墙根据应用代理和规则集，决定是否拒绝或接受请求（数据内容修改或不修改）。图6说明了这个用例。
- 角色：外部客户。
- 前置条件：无。
- 描述：

a) 外网向应用代理防火墙发出访问服务的请求。

b) 防火墙根据应用代理和访问/修正规则过滤请求。如果没有满足规则集中任何规则，那么将应用缺省规则过滤请求。

c) 如果未经修改或修改后的请求被接受了，那么将允许客户通过应用代理访问服务。

- 分支流：如果应用代理防火墙不支持服务请求，或者防火墙认为客户不可信，那么防火墙不批准该服务请求。
- 后置条件：防火墙接受来自可信客户对局域网的服务请求。

结论

应用代理防火墙模式有以下优点：

- 根据对用户透明的预定义应用代理，防火墙检查、修改（如果需要）并过滤所有请求。
- 可以通过应用代理和规则表明机构的过滤策略。
- 在包中的数据段中包含了可疑命令时，可以修改信息的特定部分
- 防火墙有助于发现可能的攻击，有助于正式用户对其行为负责[Sch03]
- 防火墙可以避免内部系统[Sch03]的协议栈中可能的执行错误。内部网络的IP（互联网协议）地址总是对外部网络隐藏。

- 防火墙适于记录流入、流出消息的系统日志
- 因为防火墙检查包括头和数据段的整个包，所以可提供高级别的安全

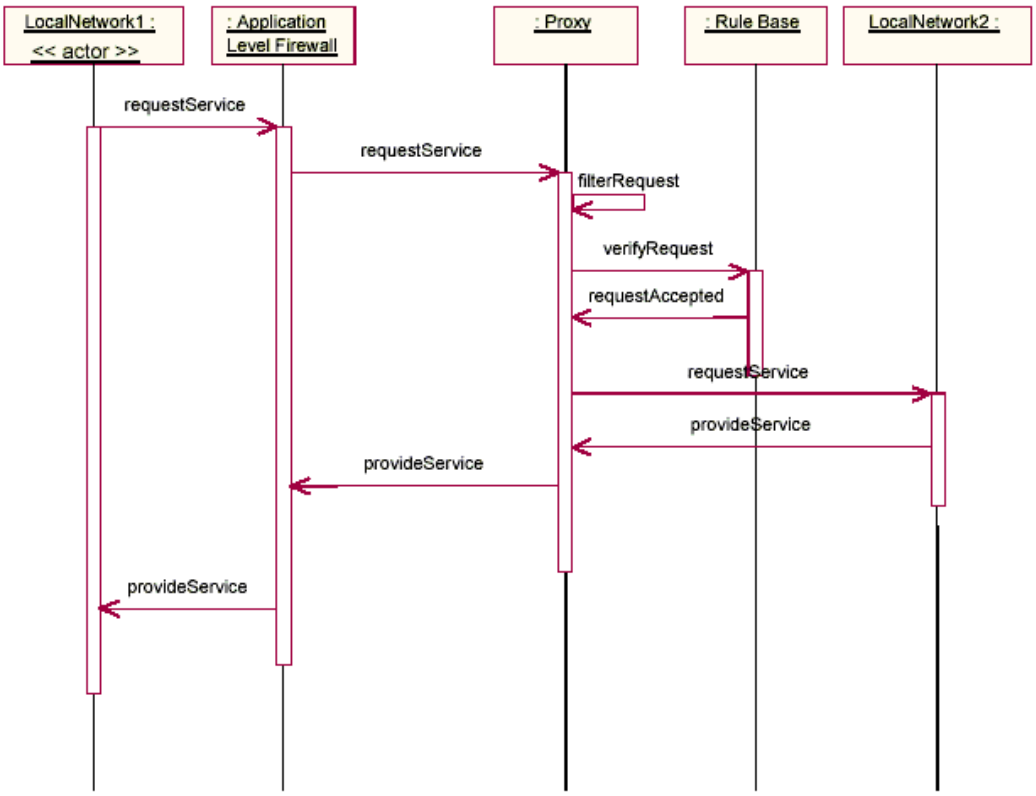


图6 过滤服务请求的序列图

应用防火墙模式有以下不足：

- 对特殊的代理有额外的执行成本，另一方面，普通服务的代理已存在
- 应用代理的开销和对包数据段的检查导致速度变慢
- 防火墙复杂性增加，在应用和/或用户和系统交互中，应用代理防火墙可能需要改变。
- 不同站点、网络和系统执行的防火墙安全策略不同。对特定应用代理增加新规则，可能干扰规则集中的已有规则，因此，必须小心谨慎地增加和修改访问规则。
- 非状态察觉

已知应用

应用代理防火墙使用在大多数防火墙产品中使用的基本地址过滤防火墙模型，如ARGuE Guard [Eps99].一些使用应用代理的特殊防火墙产品有Pipex Security Firewalls [Pip03] 和 InterGate Firewall.

相关模式

地址过滤防火墙是应用代理防火墙模型的基础。

参考文献

[Bar99] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. FIRMATO: A Novel Firewall Management Toolkit. In *Proceedings of the 20th IEEE Symposium on Security and Privacy*, pp 17-31, Oakland, California, April 1999.

[Che03] W. Cheswick, S.M.Bellovin, A.D.Rubin, *Firewalls and Internet security,* 2nd Ed., Addison-Wesley, 2003.

[Eps99] J. Epstein. Architecture and Concepts of the ARGuE Guard. In *Proceedings of the 15th Annual Computer Security Applications Conference*, pages 45-54, Phoenix, Arizona, December 1999.

[Fer01a] E. B. Fernandez and R. Pan. A Pattern Language for Security Models. In *Proceedings of the PLoP Conference*, 2001. http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/acceptedpapers.html.

[Fer01b] E. B. Fernandez. An Overview of Internet Security. In *Proceedings of the World's Internet and Electronic Cities Conference (WIECC 2001)*, Kish Island, Iran, May 2001.

[Gat00] B. Gatliff. Web by Proxy. *Embedded Systems Programming Magazine*. May 2000. http://www.embedded.com/internet/0005/0005ia1.htm

[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

[Hen01] P. Henry. An Examination of Firewall Architectures. CyberGuard Corporation White Paper, April 2001. http://www.cyberguard.com.

[Nou00] N. A. Noureldien and I. M. Osman. A Stateful Inspection Module Architecture. In *Proceedings of TENCON 2000*, vol. 2, pp 259-265, Kuala Lumpur, Malaysia, September 2000.

[Pip03] Pipex Firewall Solutions. http://www.security.pipex.net/about.shtml.

[Rus02] R. E. Rustad, Jr. Guide to OpenBSD Packet Filtering Firewalls. Kuro5hin Article, Nov. 2002, http://www.kuro5hin.org/story/2002/11/23/14927/477.

[Sch03] M. Schumacher. Firewall Patterns. In *Proceedings of the Euro PLoP Conference*, 2003.

[Vddd] InterGate Firewalls from Vicomsoft. http://www.vicomsoft.com/

[Wal01] B. Walder. Firewalls. *Computer Weekly Online*, 2001. http://www.nss.co.uk/Articles/firewalls01.htm

[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.

[Zie02] R. Ziegler and C. Constantine. Linux Firewalls: Packet Filtering. *News Riders*, March 2002. http://informit.com.

[Zwi00] E. D. Zwicky, S. Cooper, and B. Chapman. Building Internet Firewalls.O'Reilly and Associates, 2nd edition, 2000.

*拥有一个已经注册半年的 UMLChina 讨论组 ID 或者《非程序员》曾刊登过您的译稿

*32 岁以下

*本科以上

*在软件组织（软件公司或行业公司的软件部门，不包括高校和纯咨询公司）中有至少 5 年的全职分析、设计、编码的经历；在这期间，至少有两年是主要从事编码工作

*现在还在从事分析、设计、或编码的工作

*谦逊

*有重构自己现有知识的勇气和能力

*有研究精神

*英语很好（阅读、翻译、表达）或者认为自己在需要的时候能迅速使自己的英语水平达到该有的水平

*对自己的表达能力和演讲能力有信心

如果您符合以上的全部条件，而且对成为一名 UMLChina 专家感兴趣，在方便的时候请下载此表填好寄往：think@umlchina.com。

特别说明的是：

*这不是招聘广告，我们不能承诺会给谁一份 offer。寻找的人选只需 1-2 名，即使最后我们选中您，也需要花 1 年时间通过训练和实践来重构您的相关知识。所以，如果想“找工作”，这不是一个机会。

*虽然 UMLChina 专家最终只专注于 UML 相关技术，但上面条件中并没有 UML 在内。

检测网站检索的可用性

Jennifer English 著，沙萌 译

查看评论

摘要

在网站设计中，最迫切的可用性问题之一是如何改善导航和检索。我们对此问题所进行的一系列可用性研究，主要集中在一些包含大量松散信息集合的网站。本论文描述了我们的方法并且展示一个初步的研究结果，即 faceted metadata 的使用对初始级的高限制检索和中间级的较低限制检索任务都是有效的。我们还发现，用户在使用不同的检索界面类型来支持不同的搜索策略时表现出了兴趣。

介绍

尽管一般性的 Web 检索一直在稳定发展[14]，研究表明，检索依然是网站设计中主要的可用性问题。Vividence Research 的最新报告在分析了 69 个网站后发现，最普遍的可用性问题是拙劣的检索结果，影响 53%的网站，其次是拙劣的信息体系结构，影响 32%的网站。

尽管在现实中网站导航的精髓在于超链接和检索结果的相互影响，但是几乎没有什么可用性的研究来揭示如何建造高效的网站检索界面。针对这一技术缺口，我们从事一系列的可用性研究，目标就是为这一问题给出经验性、基础性的建议。我们同时也为不同类型的用户界面的灵活设计开发软件工具套件，帮助比较和分离出不同效果的特征。所以，我们可以说哪些界面工作最好，为什么（多年来设计和评价检索用户界面的经验表明）这样安排可以让我们客观地、系统地决定哪些特征好，哪些特征不好。

现在，我们来考察如何使用 metadata 把检索结果和网站的信息体系结构联系起来的。Metadata 既可用来创建网站的导航结构，又可用来组织检索结果。它还可通过预览[10]让用户在一个搜索时段内作出以后的几步操作。

接下来，我们将描述如何把 metadata 应用于检索界面，访问网站检索界面的方法和一个小型的比较三种界面可用性的研究结果，三种界面分别是：标准检索型，faceted metadata 检索型和利用 faceted metadata 与动态预览的浏览型界面。我们最初的研究结果是基于 metadata 的方法要优于标准检索，进一步可以得出用户愿意根据不同的搜索策略使用不同的界面类型。

检索过程的特性

许多研究人员发现，检索目标存在不同的种类，随之产生不同类型的检索策略。研究证实，这一思想的逻辑延伸是不同的界面配合不同的检索策略。实际上，我们最初的结论支持了之一假设。特别地，“known-item”检索和高限制的检索往往适合于基于键盘的检索界面，而低限制或可修订的任务往往适合于那些支持信息结构导航或键盘查询和导航相结合的界面。根据检索策略把检索界面区别开来这一需要曾长期受到怀疑，但是至今还没有哪种界面可以支持多种类型的检索行为（这一种类的研究和一般性的检索用户界面之评论，参见[6]）。

另一个常常被提及的检索用户界面的不足是它们不能支持中间级的检索。如象棋游戏，检索过程有开局，中盘，末盘。开局时的选择是至关重要、反复研究的，对整盘棋都有深远影响。末盘棋则要根据棋子及其所在的位置精心设计把对方将死。中盘更难描述，更需要灵活地、有机地把战术和策略结合起来。

一开始，Web 检索用户界面通常包括下列之一：一个或多个输入区域，用户可以敲入查询的词汇；一套超级链接，允许在检索的子集中导航；或者一套可钩选的项和清单选项，允许特别说明哪些被选择。末盘通常包括一个或几个内容的详细显示 - 如一篇文章，一张描述某产品的 web 页面，或者一套图片等。

联系前后的中盘一般是不好弄懂的。标准检索界面的中盘是把与查询得到的信息按匹配程度等级的顺序显示出来。这种一次性的、无状态的方式可能对良好限制的任务来说是足够了，但是一般认为，当前的检索界面不能很好地支持中级的结果[4]。越复杂的界面需要满足用户越不确定他们所要查询的信息的情况，并且帮助他们在查询过程中了解更多的信息集合和主题内容[1,8]。

我们试图回答的一个问题是如何最好地支持中盘 - 检索过程的中间级。我们通过比较不同的方法和探测从事这项任务的参与者的心理状态来做到这一点。 这包括回答关于参与者是否明白下一步如何进行，对现在追求的路径有多少信心，有多少次撤销操作或者找不到结果等等方面的问题。

我们同时也调查网站信息体系结构设计中“气味”的概念。我们把寻找信息的过程和跟踪野兽的踪迹做一个类比。如果气味消失了，踪迹变冷了，那么我们不再能确定下一步如何走。在网站的导航结构中，如果用户常常可以了解下一步要点击什么，清楚现在所在的位置和他们当前的目标，那么该站点的气味好。只要清楚了接下来的步骤，用户就不会变得没方向感或者沮丧。Jared Spool 证明说，用户找寻信息所需要的链接数目不如气味在各搜索级别是否丰富来得重要[13]。Xerox PARC 的工作同样研究气味在 Web 用户界面设计中的角色[3]。但是，这些研究人员仅通过研究现存的网站的日志试着推断用户任务和达到目标所采取的途径。

相反，我们的研究是活生生的可用性研究，参与者的目标明确并且不同的界面直接地比较。因此，我们能提出在每一种任务中，哪种界面类型更好或更有效率的问题。

使用 METADATA 集成检索与导航

过去的研究显示，用户一般以往看到关键词的查询结果按一定的方式组织起来。许多研究者已经把文本集中化应用于检索结果；这种方法从实现的角度看是有吸引力的，因为分组可被自动地决定。不幸的是，研究表明普通用户发现分组的结果难以被理解。相反，他们更喜欢按分类等级组织的预测结果[11,2]。

组织结果的不同方法是将它们按分类的 metadata 组织[7]。将查询的词映射到 metadata 分类的大量工作已经完成，例如，把查询“心脏病发作”转换成预定义的“心肌梗塞”，并且增加可能的相关匹配，但是却带来了一个我们感兴趣的不同问题，即如何将 metadata 直接用在网站检索用户界面里，即作为检索的起始点，又作为可以组织基于关键词查询结果的结构。表 1 定义了 metadata 和文章剩余部分使用到的相关术语。

item	集合中某个片断的内容。
metadata	关于某个特别 item 的分类数据，通常用一套 attributes 表达。
attribute	关于某个特别 item 的谓语，如“值 57 美元”或“画于 1923 年”。
facet	attributes 组织下的主观分类，如“价格”或“时期”或“地点”。（attributes 是固有的，而 facets 是设计的）
facet 值	一个在特别 facet 语境下的属性。例如，“少于 100 美元”允许价值 57 美元 item 的“price”facet 值进入。
flat facet	按列表组织值的 facet。
hierarchical facet	按分层级别组织的 facet。

表 1 全文所使用的术语

面向内容分类的 metadata 近年来变得越来越重要，许多人对不同领域描述内容的表转产生了兴趣。Web 目录如 Yahoo 和 Open Directory Project 是使用 metadata 作为导航结构的熟悉例子。Web 搜索引擎已经开始把分类标签上的搜索点击插入其他搜索结果中。此外，Web 作为整体，许多独立的集合已经拥有了为它们所分配的 metadata；例如，生物医学杂志的文章平均拥有数十个甚至更多与之相关的 attributes。

Metadata 在组织内容集合上的使用可按几种方式分类。值得注意的是一些 metadata 分类系统中表示的或未表示的属性。

• 第一，metadata 可以被 faceted，即由分类的正交集组成。例如，在建筑图像领域，一些可能的 facets 是材料（混凝土，砖，木头等），风格（巴洛克，哥特式，明朝等），视角类型（内部视角，外部视角等），人（建筑师，艺术家，开发者等），位置，时期，诸如此类。

• 第二，metadata（或单独的 facet）可以是分级的（“位于 Berkeley，California，United States”）或者是平面的（“by Ansel Adams”）。

• 第三，metadata（或单独的 facet）可以是单值的或多值的。就是说，数据可被限制以便最多有一个值赋给一个 item（“36 厘米长”）或者允许多个值赋给一个 item（“使用油画，墨水和水彩”）。

我们相信 metadata 在检索结果中使用的充分潜力还没有被实现。我们怀疑如果用得正确，faceted metadata 可以明显地改善网站检索，特别是对那些相似风格 items 的大集合（如产品目录站点，或者是包含像药品或法律主题的图片或文本档案集合的站点）。然而，目前没有经验证明在什么时候，用什么方式让用户发现这种组织是有效的。接下来的部分我们描述了一个网站，它在检索界面中以灵活的方式运用了 facted metadata。

所以，我们要调查如何最好地把描述性的 metadata 包含到检索过程中，包括检索的初始表达和结果列表的集合。下列问题是特别吸引我们的：来自不同级别 metadata 的灵活重组比静态等级更可取（同样可理解）吗？faceted 等级比严格的等级更可取吗？得出的结果应该按照这种 metadata 的结构来组织吗，如果是，方法呢？人们更愿意遵循基于分类的超链接吗？他们更愿意发出基于关键词的查询和按照结果类标的排序吗？存不存在个体差异？最好的方式依赖于手边的任务吗？

我们注意到，一些问题与 metadata 本身的产生相关，在此我们没有涉及到。最迫切的问题是如何决定哪一种描述是信息集合的正确描述，或者至少是最恰当的描述。另一个问题是如何把 metadata 的描述分配给当前还没有被分配的 item。我们不会阐述这些问题，一部分原因是很多别的研究人员已经做了，而且实际上，仍有一些现存的分级 metadata 集合是通过手动，有时用半自动的文本分类方法分配的。

EPICURIOUS

我们想肯定的想法是使用 metadata 组织检索结果在书写我们自己的代码来支持这一功能之前是可行的。为此，我们开始调查一个曾获奖的网站，叫做 Epicurious，包含大量的菜谱。这是一个很好的演示集合，因为菜谱的内容和他们相关的 metadata 对大多数人来说是熟悉的。Epicuriousde 所使用的 facets 是主原料，烹饪法，准备工作，配料和菜肴。每个都有一级分类，如菜肴包括开胃菜，面包，甜点，三明治，沙司，配菜和素菜。该集合包括超过 13000 条菜谱。

Epicurious 提供了三种界面来检索菜谱：

- 基本检索表单，让用户输入若干词并按任意顺序显示题目列表的结果（见图 1）。

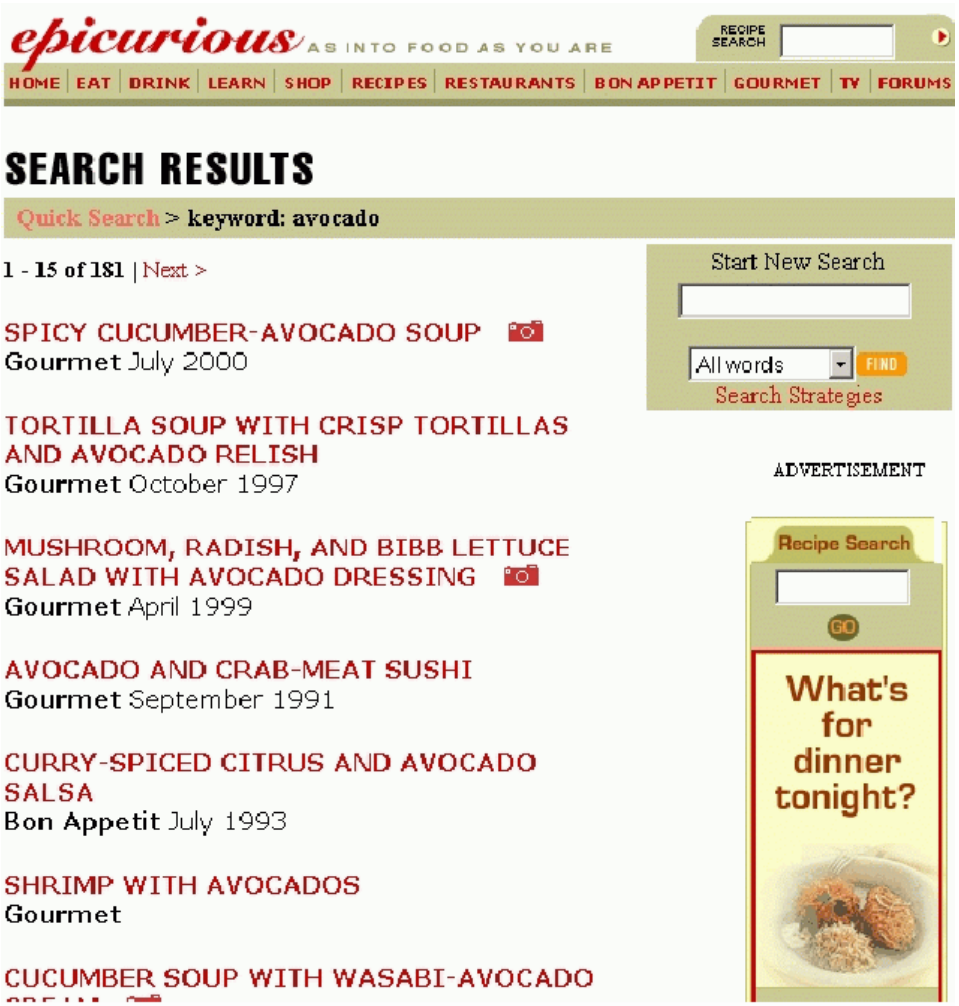


图 1 基本检索表单

• 高级检索表单，以打钩选项和下拉菜单的形式展示大量 metadata 给用户，用输入框中的关键词作为补充（见图 2）。Radio 键说明在 facet 中的选项是被“与”还是被“或”起来。所有的 facet 选择之间是用“与”相连的。结果和基本检索一样是用题目列表显示的。

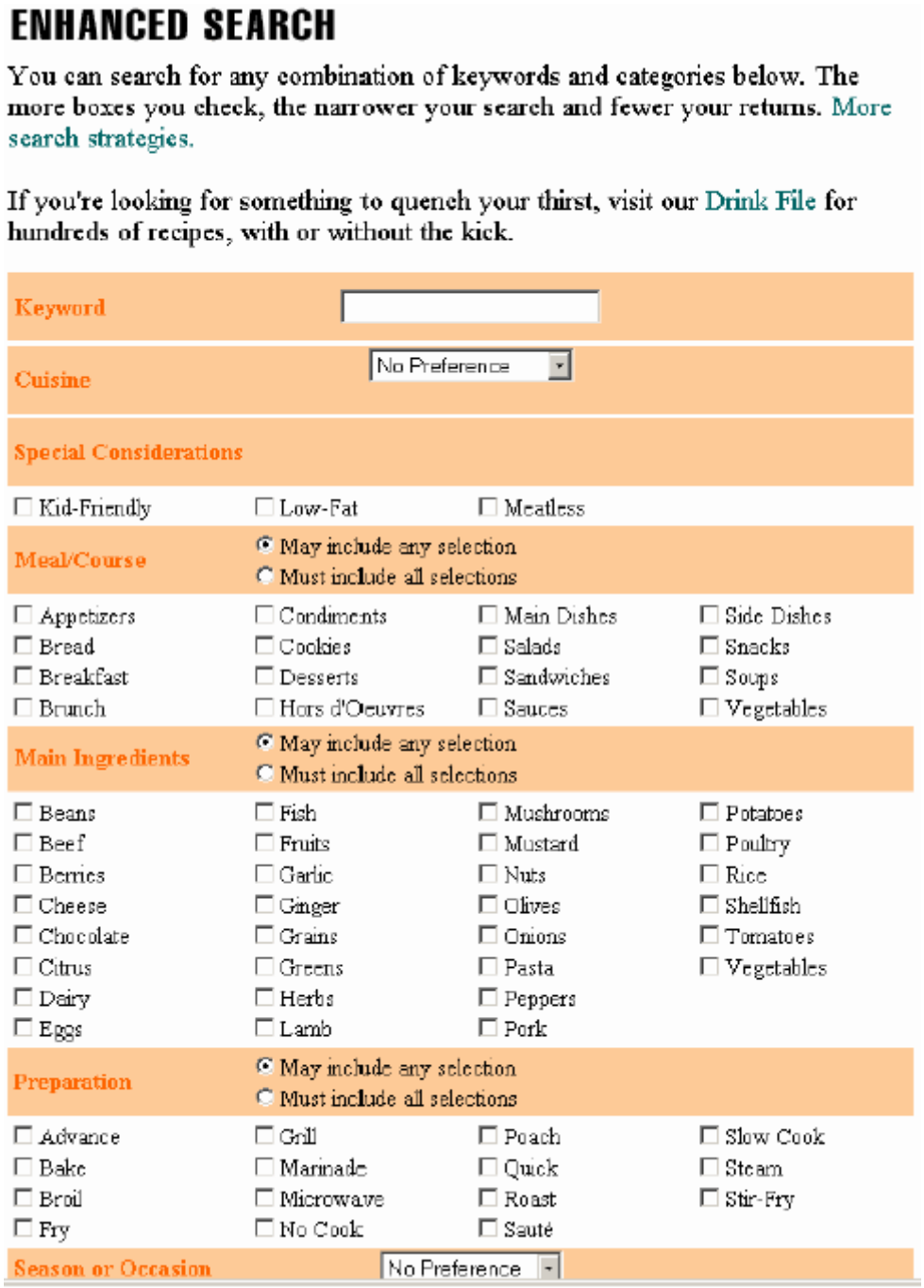


图 2 高级检索表单

• 浏览界面允许用户在集合中浏览，暗含了一个包括 facets 之间“与”操作的查询。（见图 3-5）。在一个 facet 中选择一类（如，主原料中的 Pasta）缩小需要显示的菜谱集合。在此过程的每一级，用户看到的是与题目匹配的菜谱列表预览和余下 facets 中对应的菜谱数目。例如，在原料 Pasta 选择之后，浏览界面的烹饪法 facet 显示 694 个菜谱中 202 种是意大利的，58 种是亚洲的，5 种是中东的，等等。用户可以选择切换到准备工作 facet，这里显示的是 234 种是快速的，117 种是煎炒的，结果按照菜谱的准备方式来精炼。如果当前的是烹饪法，用户看到选择意大利式，那菜谱数量就从 694 减少到 35。用户可以在输入表格内输入关键词检索精炼后的结果，但是检索结果将列表显示出来而不再和预览的信息相关了；所以，查询最终以 metadata 组织的结果视图结束（见图 5）。最后，用户可以在界面顶部“面包屑”中看到所选的 facets 和关键词的历史（如，Pasta〉 Saute〉 Italian）。

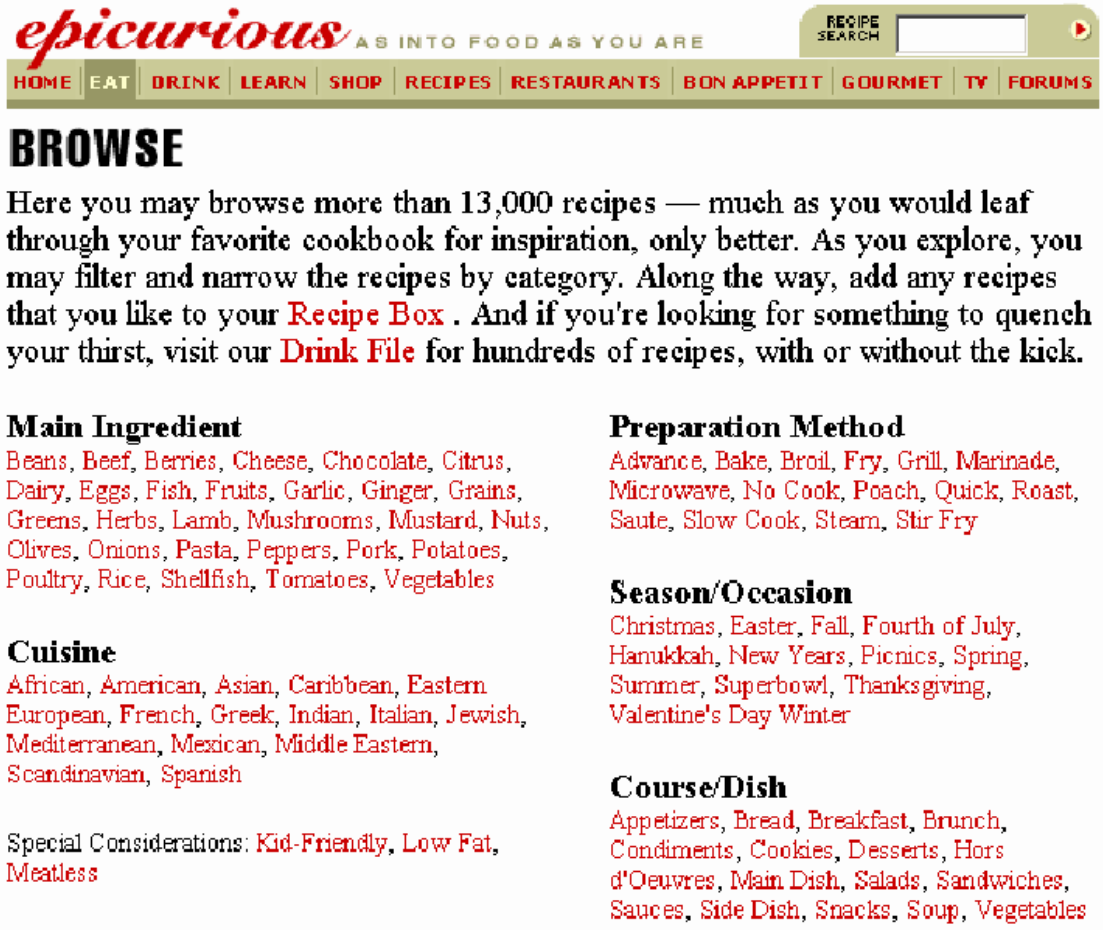


图 3 浏览界面

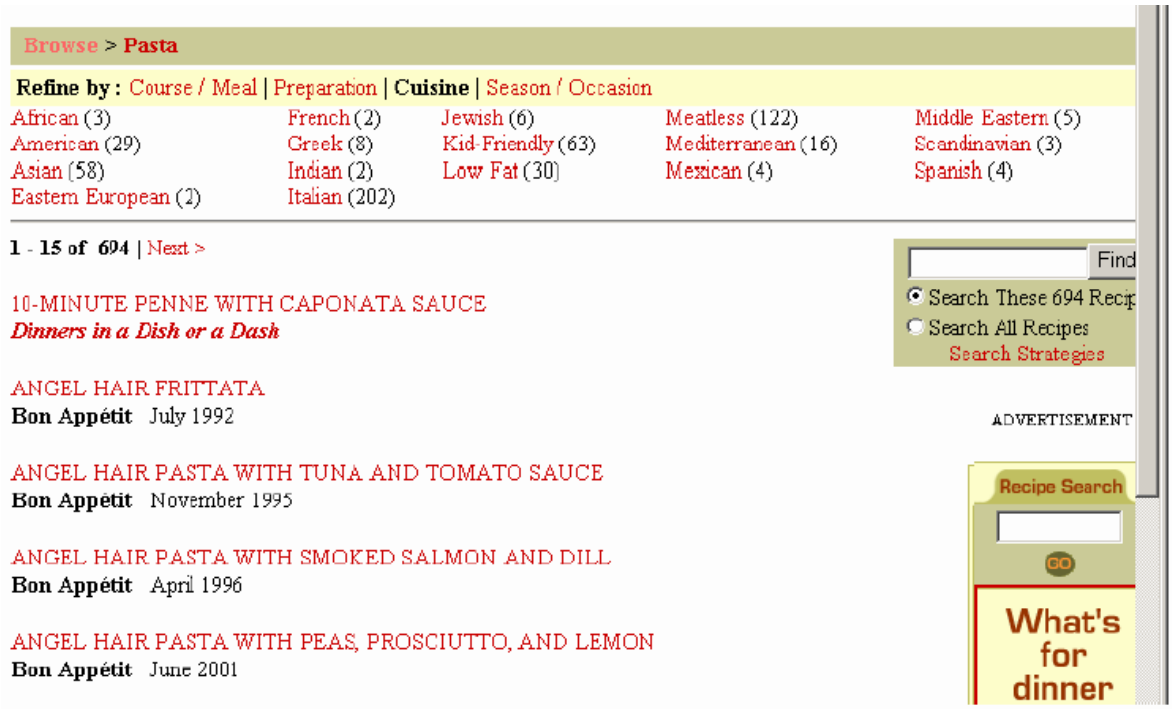


图 4 经过一次精化操作后的浏览界面中盘

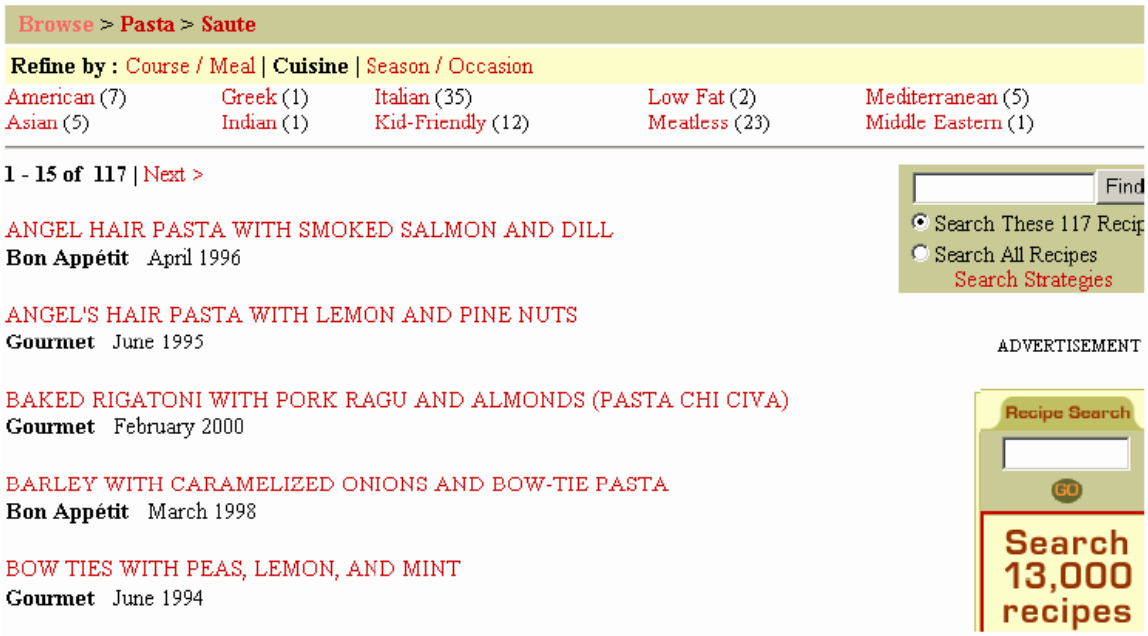


图 5 再经过一次精化操作后的浏览界面中盘

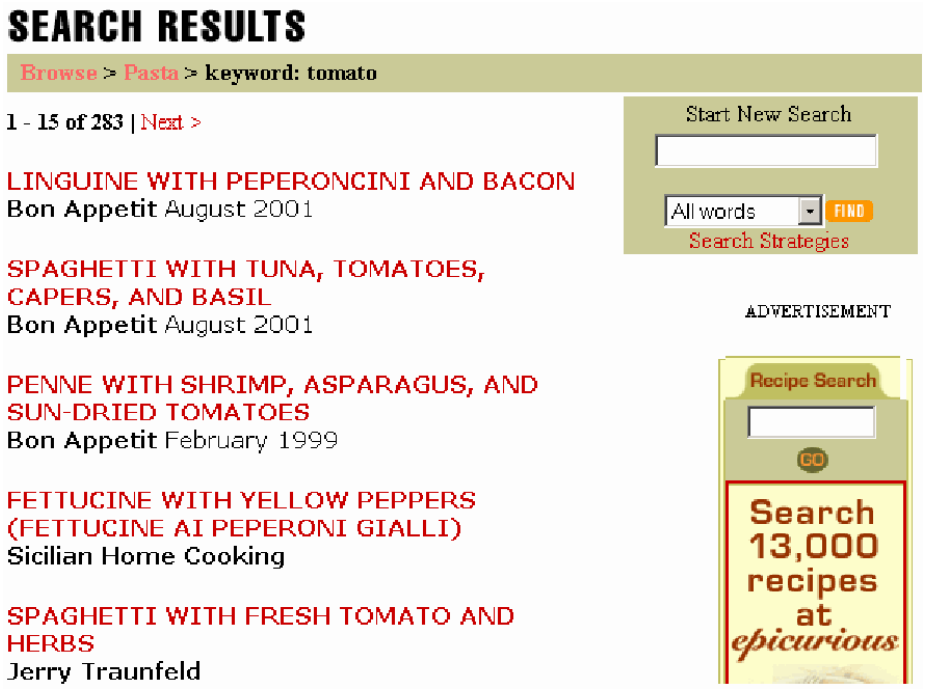


图 6 一次查询会话中的查询结果

浏览界面有一些有趣的特点。与 Yahoo 使用标准的目录不同，用户可以决定所选 facets 的顺序，而且可以动态预览选择的结果。每次从一个 facet 中选择一类都会缩小结果集合，facets 之间利用了暗含的“与”操作。为了扩大查询，用户实质上必须返回，要么通过点屏幕顶端基于历史的“面包屑”中较早前的状态，或者使用后退键。用户可以在保持结果集合不变的同时切换视图，所以能看到不同 facets 中相同的子集。但是，如图 7 所示，当使用浏览模式的时候，用户不能从一个 facet 中选择多个 items（尽管在高级检索可以做到）。

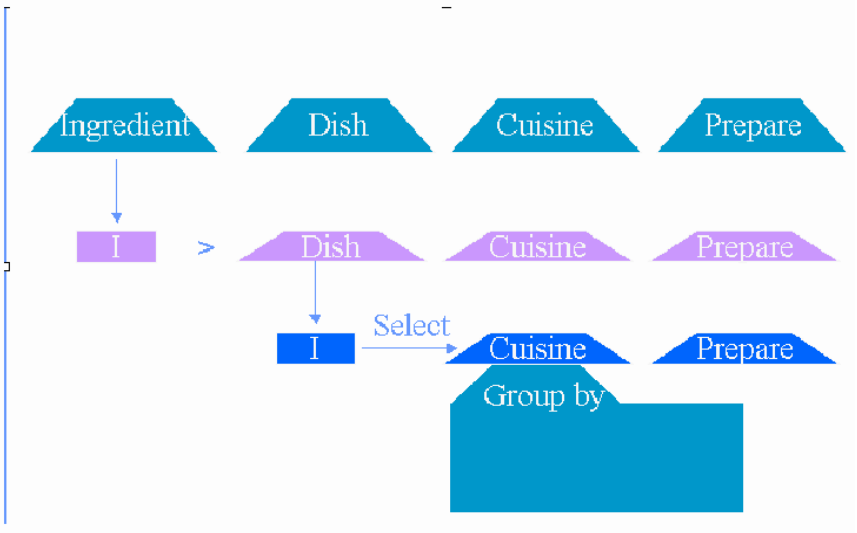


图 7 浏览界面中 metadata 的使用和查询预览架构示意

这种设计选择比起那些需要用户从每个 facet 中选择不止一个 item 的界面要更简单；但是有时也迫使用户在浏览界面下做关键词的检索，因为检索的菜谱经常不止一种原料。“在结果中检索”的界面不会把 metadata 预览中的菜谱包括进来。我们怀疑这是系统设计的漏洞，用户会更愿意在结果中包括 metadata 预览的菜谱。

最后，Epicurious 通过把每个 facet 限制在一个等级来简化用 metadata 浏览的问题。比如菜肴 facet 只有一级，所以甜点和甜饼干在同一级描述，没有糕点分类。这可能是有问题的，过去的研究显示用户是在意这种模糊的分级的[2]，facets 级别简化确实能大大简化界面设计。

可用性研究

研究目标

研究中，我们来比较 Epicurious 的三种界面。我们想知道浏览界面是否能让用户理解，用户是否要在视图中切换，预览是否有用。我们还想知道用户是否觉得不同的界面对不同类型的任务有用。经过剧本测试，我们假设高级检索对于高限制的任务更有效，而浏览界面对开放的、低限制任务更有效。最后，我们想发展怎样设计才是最好的检索界面的假设。

方法

参与剧本测试的一共有 9 人，所有人都使用 Internet 进行普通的信息检索，但技术能力差别明显。9 人中的 7 人是女性，年龄在 20 至 40 多岁之间。在 Spool 劝告的驱使下，可用性研究的参与者们确实关心他们所找寻的东西，所以我们肯定参与者都对烹饪和菜谱感兴趣。他们的参与拥有 12 美元一个小时的酬劳。

过程

我们请参与者在测试到来之前使用 Epicurious 界面，让他们对界面、环境、步骤形成各自独立的观点。这帮助减少他们在做接下来的有测试人员注视的部分时的尴尬和压力。我们请参与者完成三个“找宝物”的任务（每个界面一个任务），并且把菜谱 email 给测试提供者。（寻宝任务是给用户一个特别的菜谱，让用户找到它。例如，“找一个烤鸡三明治的菜谱”或者“找土豆沙拉的菜谱”。）

接下来，参与者来到我们的实验室进行测试。因为动机的问题，在实验室环境下是很难进行检索界面的现实性研究的[13]。如果参与者不想找那样东西仅仅是完成任务的话，他们可能会满意地一次搜索到的结果而不是努力找到他们真正想要的。为了解决所测试的剧本中会出现的这一问题，我们从两方面修改了测试协议。第一，我们让参与者自己定他们的搜索目标。我们开始询问有关他们烹饪习惯和使用菜谱网站的一些一般性问题。然后，我们和参与者一起制定出一两个场景，描述他们寻找可能的新菜谱和列出他们可能准备的菜名的情况。（为了便于实验的控制，我们指导参与者制定的场景能适应测试。）第二个调动参与者的方法是让他们保存各自查到的菜谱。（我们许诺为每个参与者制做一本菜谱格式的小册子）通过这两个简单的技术，参与者积极性变得和现实环境下一样高涨了。

于是，参与者根据场景中的步骤使用 Epicurious 网站进行菜肴的检索了。参与者为每样菜查找三遍，使用不同的界面（基本，高级，浏览）。使用顺序让用户随机决定。

参与者每得到一个中间结果，都会被问到他们是否觉得现在的位置离目标更近了，更远了，还是没变。我们记下每次中间结果的菜谱数目。

在参与者完成检索之后，我们会问他们喜欢，不喜欢，想如何改变检索方法和几个关于对结果满意度的问题。

除了自生成的检索外，参与者还使用基于 metadata 的界面（高级和浏览）完成了几个结构化的任务。结构化的任务根据限制的级别和需要结果的数量来区分。高限制、单结果任务的例子是“找到烹制烤茄子奶酪三明治的菜谱”。低限制、多结果任务的例子是“你的厨房下个月要改建了，下个月你只能做烤制食品。找到你下个月能够做的菜谱。”参与者得到四个假设的不同限制级别的任务，并被问到哪种检索界面是他们更愿意使用的。

最后，我们进行了测试后的访问，问到采访者将来使用 Epicurious 网站和三种界面的可能性。访问同时集中于专门的系统特征，问用户是否注意到了这些特征，如果是，是否感觉它们是有帮助的。

评测

下面是这次测试几个独立的研究项目：

1. Epicurious 界面（基本 vs 高级 vs 浏览）

2. 研究类型（已知项目的检索 vs 启发主观的浏览）

3. 查询限制性的程度

4. 所需要结果的数目（1 个 vs 很多）

相关的项目在结果部分讨论。

结果和讨论

我们的结果分为三类：参与者对 metadata 的理解，对 metadata 的使用，对三种界面的选择。我们还用研究结果描述了所测试的三种界面，并为提高每一种界面的效率提供了设计建议。

Metadata 的理解

从检索界面设计者的观点看，基于 metadata 的检索和基于关键词的检索是很不一样的。但是从用户的观点看，两者常常没有区别。一些界面展示了关键词和 metadata 的区别，但大部分没有。两个例子说明了我们测试的参与者心中的困惑。一个参与者试图使用关键词“kosher”并收到一个奶油鸡块加烤肉的菜谱（原料中包含着“kosher salt”）。他的理解模糊了，对关键词的搜索开始不相信。这个例子中，参与者确实需要使用“kosher”作为菜谱的描述（或 metadata）。

几位参与者在高级检索界面使用关键词的时候也发生了困惑。普遍地，参与者会在搜索栏内输入一个查询信息（如三明治），然后注意到页面余下部分与查询相同的单词。参与者不能确定他们是否应键入关键词并检查搜索栏，或者两者都做。

我们认为，上述两个例子从用户的角度说明了使用关键词和 metadata 是十分相似的限制搜索的方式。为此，尽管我们主要的兴趣是使用 metadata 限制搜索，但是在下面的分析中我们不用把 metadata 和关键词的使用分开。

另外，一些参与者想要 metadata 含义的更多解释。例如，两个参与者对每个菜谱中的脂肪含量和营养价值感兴趣；他们想确切地知道如何使用“低脂”来限制他们的检索结果。

值得注意的是，基于 metadata 的检索界面仅仅和它后面的 metadata 一样好。我们有几个对目前的 metadata 不支持的 facets 感兴趣的参与者。对这些用户来说，决定合适的 metadata 变得像一种猜谜游戏，而关键词的检索才是最有效的方法。

Metadata 的使用

一旦参与者对他们找到的合理结果表示满意，我们就要求他们停止检索。尽管三种界面提供了不同的方式设置限制，但我们想了解是否每次检索设置的限制数目会有不同。（在使用三种界面时为了决定限制的个数，我们考虑所有用到的关键词和选到的 facets。）

图 8 说明为了达到目标，参与者在使用高级检索界面时需要使用比浏览界面更多的限制。高级模式下限制的频繁使用是因为检索屏幕提供了 9 个 facets 共 68 种选择。参与者起初对这种控制的程度很兴奋，并在检索中选择很多钩选框。参与者有 27%的时间花费在缩小过多的检索结果，而且最后一无所获。（相反，在基本检索时，参与者搜到空结果只占 12%的时间，而浏览模式仅有 4%）。

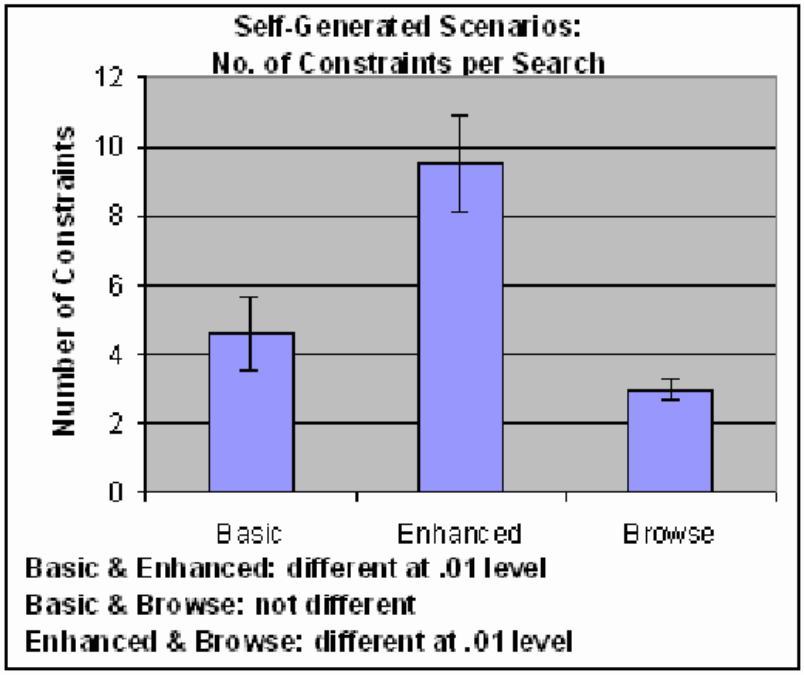


图 8 对每种查询界面，每次查询平均所用的约束条件

很可能是为了避免得到空结果集，在使用高级检索时，参与者慢慢学会在每次查询时间减少限制的数量。第一次使用高级检索时，他们输入的第一步查询的限制数目平均为 5.13，后来的几次他们第一步查询的限制数目平均为 3.39。

检索界面的选择

总体选择：当被问及哪种检索方法他们最偏爱时，参与者的选择非常平均：4 个偏爱高级检索，而 5 个偏爱浏览检索。这一点可以通过他们对使用浏览和高级检索界面指出的满意程度来解释。（见表 2）

界面	非常满意	满意	都不	不满意
基本	32%	50%	9%	9%
高级	43%	43%	4%	9%
浏览	35%	52%	4%	9%

表 2 检索结果的满意度

三种检索类型的满意度结果是很高的。我们认为这部分的原因是 Epicurious 是获奖的站点，并且我们大多数的参与者喜欢烹饪，他们乐于发掘 Epicurious 上的东西。

基于任务的选择：我们同样有兴趣弄清参与者是否会觉得特别的检索界面更适合特别的任务。为了得到最终的测试结果，我们给参与者展示假设的检索场景，并问他们喜欢哪一种界面。我们的假设场景是根据不同的限制级别制定的。表 3 显示了不同任务限制下不同界面的选择函数。参与者偏爱在高限制的任务下使用高级检索界面，在低限制的任务下使用浏览界面。只有一个参与者愿意在高限制任务下使用基本检索。这一结果表明，参与者懂得了不同的界面对于不同任务的价值，即使只是在很短的（一个小时）测试之后。同样地，参与者在这一点的判断上完全一致。我们认为，这一发现可以证明不同的基于 metadata 的检索界面满足不同的检索任务的需要。

	基本	**高级**	**浏览**
低限制	0	1	8
高限制	1	8	2

表 3 界面选择作为任务类型的函数

对系统特征的反应：我们询问了参与者关于界面的几个特别的方面 - 他们是否注意到了这些特征，如果是，觉得这些特征在多大的程度上帮助了他们的检索。表四概括了参与者对专门的系统特征的反应。有趣的是，只有一个参与者发现搜索选项如“布尔查询”和“精确词组”是有用的，而且仅有两人发现“可以包含”和“必须包含”选项是有用的。仅有一个特征，参与者在检索过程中感到兴趣的，是高级检索界面中的“在同一屏幕中设置所有的标准”。这可能和过度限制高级检索带来空结果的频率有关。

	没注意	有用	没用	干扰
浏览：查询预览	1	5	1	-
高级和浏览：含有完整的原料清单	-	7	1	-
浏览：在结果中检索	-	6	1	-
浏览：使用超链接精炼	2	5	-	-
高级：在同一屏幕中设置所有的标准	-	3	2	2
高级：“可以包含”和“必须包含”选项	1	2	4	-
基本/高级：“所有单词”，“任意单词”，“精确词组”和“布尔操作”选项	3	1	3	-

表四：专有特征的反应

三种界面的交互描述

基本检索界面

Epicurious 基本检索界面的操作和标准的检索引擎界面类似。大部分参与者都感觉它容易使用。从满意度的数据来看，参与者喜欢这种界面。但是，他们也认识到它的局限，因为当我们问及在低限制和高限制任务下基于任务的选择时，只有一个参与者认为它比高级检索和浏览检索更适合。

Epicurious 基本检索界面是非常局限的，因为不提供任何形式的中盘。参与者只能发出新的查询，然后接受或者放弃检索的结果。使用后退键是唯一可以精炼检索的方法。

我们认为基本检索界面在基于 metadada 的检索界面中能够提供简单的输入点。用户熟悉并能毫不费力地使用它。但是，基本检索界面的中盘应该提供选择来弥补它的不足。

高级检索界面

使用多个限制：Epicurious 高级检索界面鼓励多个限制的使用。参与者在每次使用高级检索时平均尝试 9.5 次的限制（包括关键词）。在此之前，全部的 metadata 选项让参与者觉得被迫要他们选择一两个。这限制了检索，所以常常造成空的结果。参与者会回到高级窗口，在得到可接受数目的结果前渐渐放弃使用限制。一些参与者觉得这是很让人沮丧的；另一些人却并不介意。上面的表显示，参与者有 28%的时间是以空结果告终。

时间：每次检索的平均时间是 168 秒。这比浏览和基本检索的时间要长，但是在统计上没有明显的差别。导致这一结果可能的原因是参与者使用高级检索需要花更长的时间来制定查询。每一步的平均时间也最高（68 秒）。但是所用的步骤比浏览要少。

控制感：我们假设基本查询，高级查询和浏览查询产生不同的控制感。一些参与者提出他们喜欢高级查询的原因之一是它所提供的控制程度。用户能够选择他们感兴趣的特别的 facets（通过点击钩选栏）。带来控制感的另一个原因是改变检索或者开始新的查询只需要从结果页面点出去即可。

我们认为高级检索界面是自然的、功能强大的界面。但重要的是把某种反馈机制结合到这种界面来以便在参与者的检索将导致空结果的时候提醒他们。问题是这种检索形式（基本和高级）不能洞察被检索的数据集。因此除了通过尝试和犯错，用户没法了解数据集。

一个可能的补救办法是界面提供一些专门的限制性反馈，或许是以一种查询预览的方式。另一个可能的设计方法是在第一次查询时提供较少的 facets，再用剩下的 facets 组织结果。Epicurious 高级检索界面的另一个问题是它不能提供方法来纠正错误操作。当参与者选择过多或过少的 facets 时，他们必须返回开始页面去修改。一个强大的中盘能够允许用户修改或者精炼查询而不用完全重新开始。

浏览界面

Epicurious 浏览界面拥有比基本/高级检索界面更强大的中盘。参与者能够通过选择一个 facet 开始。一旦他们已经做好了初始的选择，在他们前面就有整个范围的选择。他们可以在结果中检索，回溯，精炼，切换。参与者利用全部这些选项，但比其他两者更加易懂。

在结果中检索：参与者频繁选择这个选项（31%）。这是最容易懂的选项。

回溯：参与者通过后退键或“面包屑”可以返回。在大多数情况下，参与者使用后退键；“面包屑”是偶尔使用。后退键一般回溯一步或两步，然后再用不同的 facet 精炼结果。

切换 facets：Epicurious 还为用户提供切换 facets，效果是让他们根据不同 facets 的查询预览可看到相同的结果集合。一开始，在他们使用了一些次数之前，用户不注意或不理解这一特征。但是，一旦他们发现这个特征，就会觉得它非常有用。

Epicurious 浏览界面是一个高效的、设计良好的检索中盘的例子，并且我们猜想基本检索和高级检索界面应该通过结合浏览界面的中盘特点得到改善（如在检索之后的结果中组织输出）。

但是，这种界面仍有不少问题没有解决。最迫切的问题是如何处理按等级分配的 facets 的 metadata。

结论和未来工作

此次初步性研究的结果说明了基于 metadata 的界面和动态生成的基于 metadata 的查询预览是检索和浏览松散信息大集合的高效界面。结果还显示用户认识到利用检索界面之间的切换，让界面最好地支持合适的检索策略。最后，对于像菜谱这样一致性的集合，利用 metadata 形成查询并浏览中间结果是检索用户界面设计的一条捷径，远远好于基于关键词配合结果列表的标准检索。

研究还是太小，不能从中得出什么结论，特别是关于总体选择的结论。而且，Epicurious 站点的性质给我们留下了许多未解答的问题。集合与 metadata 还只是中等大小，metadata 也只是用平面 facets 组织的。还有，用户在浏览模式下不能从一个 facet 里选择多个 items，而且对扩大查询的支持也不够。我们有兴趣学习如何使这种方法符合非常大的集合（所以需要分等级的 metadata facets），支持多 facets 下多 items 的选择，支持简炼和扩大的检索，检测更丰富的信息需要。

为此，我们正着手从事另一个更大的研究，使用我们自行开发的软件，容许我们自己直接对不同的界面特征进行比较。（图九显示的是做比较的界面之一。）虽然现在有针对 faceted metadata 集合的软件，但是这次初始的测试集合包括超过 40,000 张建筑图库的图片，并且设计的任务是要满足现实中建筑师观点下的信息查询的需要。每次浏览包含大约 10 项 items，而且 metadata 级别中包含了数千个词和九个 facets。这些研究将帮助我们进一步了解如何改善网站检索的用户界面。

图 9 以 metadata 为中心的界面图片架构

参考资料

1. Marcia J. Bates. The berry-picking search: User interface design. In Harold Thimbleby, editor, User Interface Design. Addison-Wesley, 1990.

2. Hsinchen Chen, Andrea L. Houston, Robin R. Sewell, and Bruce R. Schatz. Internet browsing and searching: User evaluations of category map and concept space techniques. Journal of the American Society for Information Sciences (JASIS), 49(7), 1998.

3. Ed H. Chi, Peter Pirolli, Kim Chen, and James Pitkow. Using information scent to model user information needs and actions on the web. In Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, Seattle, WA, April 2001. ACM.

4. Steve B. Cousins and Ken Pier. Atask-orien ted interface to a digital library. In Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, Conference Companion, Vancouver, Canada, May 1996. ACM.

5. Fredric Gey, Hui-Min Chen, Barbara Norgard, Michael Buckland, Youngin Kim, Aitao Chen, Byron Lam, Jacek Purat, and Ray Larson. Advanced search technologies for unfamiliar metadata. In Meta-Data '99 Third IEEE Meta-Data Conference, Bethesda, MD, April 1999.

6. Marti A. Hearst. User interfaces and visualization. In Ricardo Baeza-Yates and Berthier Ribeiro-Neto, editors, Modern Information Retrieval, pages 257–323. Addison Wesley, ACM Computing Series, 1999.

7. Marti A. Hearst. Next generation web search: Setting our sites. IEEE Data Engineering Bulletin, 23(3), September 2000.

8. Gary Marchionini. Information Seeking in Electronic Environments. Cambridge University Press, 1995.

9. Vicki L. O'Day and Robin Jeffries. Orienteering in an information landscape: how information seekers get from here to there. In Proceedings of the INTERCHI '93, Amsterdam, April 1993. IOS Press.

10. Catherine Plaisant, Ben Shneiderman, Khoa Doan, and Tom Bruns. Interface and data architecture for query preview in networked information systems. ACM Transactions on Information Systems, 17(3):320–341, 1999.

11. Wanda Pratt, Marti Hearst, and Larry Fagan. A knowledge-based approach to organizing retrieved documents. In Proceedings of 16th Annual Conference on Artificial Intelligence (AAAI 99), Orlando, FL, 1999.

12. Vividence Research. Tangled web 2001. http://www.vividence.com, June 2001.

13. Jared Spool. Web Site Usability: A Designer's Guide. Morgan Kaufmann, 1998.

14. Danny Sullivan. Npd search and portal site study. http://searchenginewatch.internet.com/reports/npd.html, July 6 2000. NPD's URL is http://www.npd.com.

UMLChina 公开课

“UML 应用实作细节”

（2003 年 11 月 15-16 日，上海）

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>>

软件开发 以人为本

乐林峰 著

查看评论

对于广告，我向来是左耳朵进，右耳朵出。但有一个国外品牌手机的广告例外，广告语是“科技 以人为本”，听了之后就记住了，而且至今还没忘。

写这篇文章之前，我琢磨着怎么也得先起个题目吧，不然光叫“《人件》读后感”？ 太像小学生作文（虽然我的写作水平比小学生也强不了多少）。唉，那广告就从脑子里冒出来了，稍加修改“软件开发，以人为本”，下边再加一小副标题，怎么看怎么觉得不错，透着那么专业，得，就它了。

对着题目看了一个多星期，内容一个字儿也没写出来，不知道怎么写。我一琢磨，这光有个题目不行啊，得想办法整点东西出来，再说了《人件》也不是没的可写。实际上，这本书值得大书特书的地方太多了，想敞开的夸一把吧，只是不知道从哪里下嘴好。

既然不知道怎么写，那就还从题目开始吧，题目我拿手。

人件，英文是 peopleware,应该是由 people(人) 和 software（软件）拼凑在一起的合成词，发明这个词的人(不知道是不是《人件》一书的作者)可能是为了强调软件与人之间关系的重要性。不过，这里的人所指的是软件开发人员。《人件》整本书都在讲开发人员，以及跟开发人员有关的事情。主要有以下几个方面：非模块化的人力资源管理方法、工作环境对开发人员的影响，人员雇佣，如何组建高生产率的开发团队，以及如何使工作成为一种乐趣。作为第一版的补充，本书第二版又增加了过程改进等新的内容。

以前听说过这本书，如何如何牛，一直书店里也没见着有卖的。而且老外对版权的态度比咱们“小气”得多，在网上一直也没找到完整的英文电子版。等好不容易出了中译本吧，拿到书之后却犯了愁，没看功夫看。上班时间？肯定不行，项目进度压着，兄弟们大多被发出去现场开发，过朝九晚午(早上九点晚上到午夜十二点)的幸福时光去了，幸存下来的也是每天加班，让你觉得正点儿下班就跟偷了公司东西似的，更别说上班时候抽空看书了，再说耽误了工作被老板炒鱿鱼就更不爽了。晚上回家看？也不行。老婆说了，你老出差我就不追究了，加班回家晚我也不追究了。噢，好容易回家还想自己看书不理我？不行，陪我看电视！人家说得在理儿啊，得，陪着看吧。电视台那帮哥们儿也是，我都这样了，也不放点大家爱看得让我高兴高兴，弄的我每次看完电视都后悔晚饭吃多了，胃里有东西往上顶。

扯远了，还是回到书上来吧。既然谁都惹不起，那只有动用自己的私房时间了。光听说有私房钱，时间还有私房的？没错，这得感谢北京地儿大人多交通不便，到哪儿上班，路上花个把小时都普普通通。拿我来说吧，上班在现代城，住丽水桥，每天早上先公共汽车、倒轻轨、再倒地铁环线最后倒一线，需要一个半小时，除去走路倒车一来一去也有 2 个多小时空闲，这就是我的私房时间。这段时间里我最喜欢干的事情就是看武侠小说，而且只看金、古二位大侠的书。车上看书最大的好处是没人打扰，人虽然多，但不会有人问你这个问题怎么解决那个程序什么时候完成，手机也可以不开，要是问起来可以说在地铁里呢没信号。看着不需要动脑子的书，还没人打扰，享受啊,美中不足的是站着比较累。

那天上班包里的武侠换成了《人件》。在车上找了个旮旯看将起来，刚翻了几页，心里一动，嗯，不错啊。请注意我用的词“心里一动”，简单点说就是心动，我最后一次心动是头一次看到我老婆的时候，都好几年了。越往下看，心动的次数越多。凭我多年看武侠小说的经验，这本书即使称不上是葵花宝典，也够资格进少林寺藏经阁，绝对是当今软件武林中不可多得的上乘绝学。一下子爱不释手，摇头晃脑的一路读下去，弄得好几次差点坐过站。

开始看书的心情是高兴、痛快、解气，就跟当年穷苦老百姓遇到党似的，可有人站出来替我们说话了，可见着亲人了。本来吗，工作中遇到让人觉得不妥、不爽的事情多了去了，可嘴笨，就是说不出个所以然来，等打开书一看，原来上边都有，而且说得还特别到位，您说我能不乐吗。比如说吧，给开发人员施加压力是最常见的做法吧，你说做这个软件需要写 1 个月？好，给你打 5 折，2 周交活，行不行？敢说不行吗。拼死拼活干出来了，他来话儿了，瞅瞅，这不 2 周也做完了吗？你还没脾气。看看书上是怎么说的

“人们在受到时间的压力的时候不是工作得更好，而是工作得更快”。

多精辟，而且还透着那么有哲理，我怎么就琢磨不出来这样的词儿呢。仔细想想，工作好几年了，遇到紧急情况，采取的对策就是两个，一是加班，二就是想辙,或者说想办法应付。

讲一件印象比较深的事情，一年前我为某个公司工作，他们让我负责开发一个开发辅助工具，以减少开发人员重复劳动。我一想，工具是给大伙用的，真能用起来可以节省大把时间，另外还能显着自己水平高，得好好干。满腔热情又是分析又是设计的，框架图没几天就设计出来了,还没等编码呢，上边说了，马上有新项目，那个工具抓点紧，下周最好能用。得，一个多星期，要是按照设计好的方案开发，一天工作 24 个小时也肯定没戏，因为那个方案考虑得比较周密，采用什么系统结构、开发模式，将来扩展的可能性等等，实现起来肯定费时间。没辙，只得从网上找了个比较接近的有源代码的程序，至少界面代码有了，原有的功能用就用,没有的自己再写。等完了再一看，好嘛，这可热闹了，至少有一半代码是人家原来的，用不上也择不出去，跟那搁着吧；有一部分是在原有代码基础上改的，剩下的才是我自己写的。整个一大杂烩，不过该有的功能都实现了。别说，后来那个工具在

公司内还真用起来了。之后有几个使用那个工具的人有意无意的跟我提起，要是能增加这个或那个功能就更好了。我心里想，这些功能当初我不是没想到啊，要按照我当初设计的框架，加上去并不难，现在呢？由于全是拼凑起来的，结构混乱，即使是自己写的程序，也为了省事儿哪里方便放哪，东一堆西一块儿的，要加点东西进去可就费劲了。遇到这种情况，我要么借口现在手里的活紧搪塞过去要么装聋作哑，但心里总不免遗憾。类似的事情很多，之所以对这件事情印象深刻，我想可能是因为我对自己当初的设计方案特别满意的缘故吧。现在我还保留着设计文档，有时打开看看，借鉴借鉴原来的思路，同时自我满足一把。那个工具的代码也留着，但从来没看过一眼。

大多数软件开发人员是很好的“厨师”，都希望、同时也有能力做一桌大餐，但被时间挤兑得只能是给客人上一份杂烩菜。虽然也能满足需求—吃了不饿，但最终的效果可差不少呢。

接下来越看心里越不是滋味，好像书里提到不好的事情，我要么亲身经历过、要么听说过，但几乎没赶上什么好事儿。就拿书中工作环境对工作效率的影响来说吧，书的作者认为开放式的办公环境不利于工作。一点也没错，我曾经供职过的一个公司就采用了开放式办公室格局。几个篮球场那么大的一个房间，以前据说是做库房用的。每四张办公桌被放到一个格子里，而且背靠背坐,即每个人面向一个格子的一角。挡板很矮,坐着的时候对四周的景色也是一览无遗。整个办公室就像个大集贸市场，电话、讨论，人走来走去。我那时候也是头一次在开放式办公室工作，开始觉得别扭，想可能适应适应就好了。可后来发现，根本就不是适应不适应的事儿。在那种环境中，很难高效率的工作，特别是在设计的时候，静不下心来思考。往往一天下来很累，但仔细一想又没做什么事情。那段时间留给我的记忆是效率低下、错误百出，觉得自己特别失败，有很强的挫折感，脾气也暴躁。现在想想，可能跟当时的办公环境有很大关系。如果当初那个公司的经理看过《人件》这本书的话，估计他可能就不会采用开放式办公室了吧。

说到这儿，您是不是觉得这本书是一本反面教材啊，说得全是这儿不好那儿不好,把开发人员说得个顶个跟受气的小媳妇儿似的。不全是，书中也从正面，从管理者的角度，介绍了许多很好的管理方法和经验。“书中推荐的许多方法已经成为当今一流公司的标准”（这是 IEEE software 的 Steve McConnel 说的）。比如在开发团队培养方面，作者提出并倡导的“胶冻团队”概念(胶冻团队是一群紧密结合在一起的人，其整体生产力大于每个成员独立工作时的生产力的总和)，并且集中讨论了“胶冻团队”的主要标志，以及如何组建胶冻团队。内容很多，无法一一详细介绍。对于这些内容，我只是觉得很有道理，颇有点长了见识的感觉，但却没有引起我多少共鸣。原因是我的管理经验不多，充其量也就当过只有几个程序员的开发小组的小头目而已，所以没有太多的发言权。这些内容现在可能对我没有太大帮助，但对那些管理人员来说我想可能就不一样了，绝对值得好好研究一番。

看完了书之后，我有一种冲动，那就是自己掏钱买几本《人件》送给我现在公司的几位总们。但也就那么一想，没敢真送。真送肯定是自找倒霉。对公司不满是吧？不满好办，走人啊你。再有了，我有问题自己发现自己改可以。你做下属的提出来我岂不是很没面子？本来就不怎么锦绣的前程，再让我自个花钱给断送喽，这不是冤大头嘛。当然了，我这也是以小人之心度君子之腹，我们公司的领导未必是这样，不过谁让我以前在别的公司碰到过这样的事儿呢，还是小心为好。

别看没给我的上级送书也没向他们推荐，但我希望所有软件公司的领导们都能看看这本书。搞软件的都知道微软，甚至有些还梦想着有一天能赶上甚至超过微软。我听说微软的管理人员都看过这本书，所以如果您是一位管理者或者自己就是老板，我强烈建议您找一本《人件》看看，并且让您手下凡是管理 2 个人以上员工的人也都看看，说不定哪天您真成了比尔盖茨第二呢。呵呵，说得有点过，这本书可能没那么神，但对软件公司确实有很大帮助倒是勿庸置疑的。

对于广大像我一样，默默无闻勤勤恳恳工作在软件一线的开发人员同志们。我要说的是，如果您有希望或者自认为有希望成为管理者，那么建议您也读读此书，提前准备准备。有朝一日翻身当家做主人了，给穷苦兄弟们多谋谋福利，而且还提高自身的管理水平，使事业更加一帆风顺，何乐而不为呢。

对于那些已经对现状不满、愤世嫉俗而且心理承受能力比较差的兄弟，建议还是别看这本书了。活得不好并不可怕，可怕的是你清楚的知道自己活得不好，又无力改变现状而且还不能笑对人生。我们北方那个邻居朝鲜，成天吃不上喝不上的还饿死人，但照样高唱“全世界人民都羡慕我们”之类的赞歌，一有点什么事儿，呼啦一下就上街游行还载歌载舞，看着个个满脸都写着幸福，那不也挺好嘛。

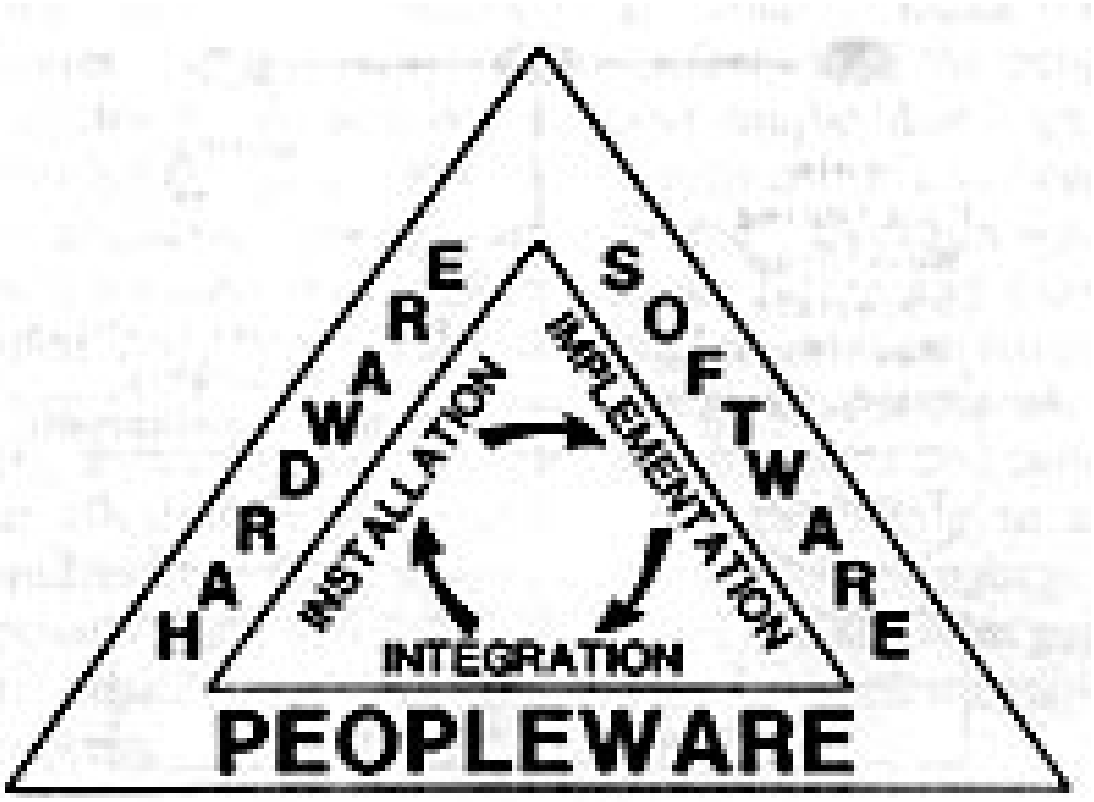
文章的最后，我想以我经历的一个小段子作为结束。前不久跟几个做软件的哥们儿聊天，大家一起说起将来的出路问题。一个兄弟说搞软件太累，没什么意思，等过几年实在写不动程序就去街上推车卖烤白薯。另一个说，你也太没追求了，好歹也算个白领，怎么能卖烤白薯呢？怎么着也得开一个饭馆啊，十几条板凳七八张桌子的规模。我说我比你们都有追求，我要开一个饭馆，门口卖烤白薯。您也看出来了，我讲这个故事并不是想说我们几个打算从事餐饮行业，只是想从一个侧面来反映一些软件开发人员的心态。

工作几年，身心疲惫，没成就感，心情压抑、对前途没信心，但也没彻底放弃，至少还笑得出来。对于现状，我想肯定会有所改变，会越来越向好的方面发展。当然了，光有个美好的愿望是不行的，还需要大家有所行动。

按照我们的习惯，在讨论某些事情的时候最后都要进行总结，我的总结如下：

如果更多的业内有识之士读《人件》这本书，并把书中的伟大指导思想和行动准则广泛应用到具体的软件开发生产实践活动中去，走外国先进管理思想与国内软件行业相结合的道路。我们的事业、收入和生活质量会更上一个新台阶，我国的软件产业也会出现一派欣欣向荣的新气象（本段内容为笔者杜撰，如有雷同，纯属巧合）。

乐林峰，系统分析员，《00 项目求生法则》的译者。



The Addison-Wesley Signature Series

Patterns of Enterprise Application Architecture

中译本即将上市

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

UMLChina 负责中译本最后审校，并指定为训练用教材

《人月神话》中译本修订版对每章前引语的修订

第1章

A ship on the beach is a lighthouse to the sea.

原译：岸上的船儿，如同海上的灯塔，无法移动。

修改：前车之履，后车之鉴。

第2章

Good cooking takes time. If you are made to wait, it is to serve you better, and to please you.

美食的烹调需要时间；片刻等待，更多美味，更多享受。（保留原译）

第3章

These studies revealed large individual differences between high and low performers, often by an order of magnitude.

原译：这些研究表明，效率高和效率低的实施者之间具体差别非常大，经常达到了数量级的水平。

修改：这些研究表明，每个人工作效率的高低存在着很多个体差别，通常会有一个数量级（10倍）的差距。

第4章

Aristocracy, Democracy, and System Design

贵族专制、民主政治和系统设计（保留原译）

This great church is an incomparable work of art. There is neither aridity nor confusion in the tenets it sets forth. . . . It is the zenith of a style, the work of artists who had understood and assimilated all their predecessors ' successes, in complete possession of the techniques of their times, but using them without indiscreet display nor gratuitous feats of skill.

It was Jean d 'Orbais who undoubtedly conceived the general plan of the building, a plan which was respected, at least in its essential elements, by his successors. This is one of the reasons for the extreme coherence and unity of the edifice.

大教堂是艺术史上无与伦比的成就。它所宣扬的理念既不乏味也不混乱……它是一种风格上的极致，要完成这样一件艺术品，建筑大师要首先融会贯通其前辈建筑师的成果，同时完全掌握他们那个时代的建筑技术，并在运用这些技术时做到恰如其分，避免轻浮的炫耀和无端的耍弄花招。

无疑当初正是Jean d'Orbais构思出了这个建筑的整体设计，这个设计得到了其后继建筑师的认同，至

删除的内容: 的原则

删除的内容: 真正达到了

删除的内容: 件

删除的内容: 作品的艺术家们，完全领会和吸收了以往的成功经验

删除的内容: 也

删除的内容: 了

删除的内容: 而且

删除的内容: 应用的

删除的内容: 候

删除的内容: 了

删除的内容: 绝不轻率，也绝不花哨

删除的内容: 了

删除的内容: 者

> 少在建筑的基本要素方面是如此。这就是这个宏伟的建筑能够如此和谐统一的原因之一。

删除的内容: 本质上

删除的内容: 也

第5章

The Second-System Effect

> 画蛇添足（保留原译）

Add little to little and there will be a big pile.

> 聚沙成塔，集腋成裘。（保留原译）

第6章

Passing the Word

> 贯彻执行（保留原译）

He'll sit here and he'll say, "Do this! Do that!" And nothing will happen.

> 他只是坐在那里，嘴里说：“做这个！做那个！”当然，什么都不会发生，光说不做是没有用的。（保留原译）

第7章

Why Did the Tower of Babel Fail?

> 为什么巴别塔会失败（保留原译）

Now the whole earth used only one language, with few words. On the occasion of a migration from the east, men discovered a plain in the land of Shinar, and settled there. Then they said to one another, "Come, let us make bricks, burning them well." So they used bricks for stone, and bitumen for mortar. Then they said, "Come, let us build ourselves a city with a tower whose top shall reach the heavens (thus making a name for ourselves), so that we may not be scattered all over the earth." Then the Lord came down to look at the city and tower which human beings had built. The Lord said, "They are just one people and they all have the same language. If this is what they can do as a beginning, then nothing that they resolve to do will be impossible for them. Come, let us go down, and there make such a babble of their language that they will not understand one another's speech." Thus the Lord dispersed them from there all over the earth, so that they had to stop building the city. (Book of Genesis, 11:1-8).

第8章

Calling the Shot

> 胸有成竹（保留原译）

Practice is the best of all instructors.

> 实践是最好的老师。（保留原译）

Experience is a dear teacher, but fools will learn at no other.

> 实践是最好的老师，智者还能从其他的地方有所收获。

删除的内容: 但是，如果不能从实践中学到东西，再多的实践也没有用。

第9章

Ten Pounds in a Five-Pound Sack

削足适履（保留原译）

The author should gaze at Noah, and ... learn, as they did in the Ark, to crowd a great deal of matter into a very small compass.

负责筹划的人应该瞪大眼睛紧盯着诺亚，然后……好好学习一下，看他们一家人当时是怎样把那么一大堆东西塞进一个小小的方舟上的。

删除的内容: 他

删除的内容: 多

删除的内容: 装到

第10章

The Documentary Hypothesis

提纲挈领（保留原译）

The hypothesis: Amid a wash of paper, a small number of documents become the critical pivots around which every project's management revolves. These are the manager's chief personal tools.

前提：

在堆积如山的文件资料中，少数文档是关键枢纽，每一件项目管理的工作都围绕着它们运转。这些文档是项目经理最重要的个人工具。

删除的内容: 一片文件的汪洋

删除的内容: 形成了

删除的内容: 它们

删除的内容: 们的主要

第11章

Plan to Throw One Away

未雨绸缪（保留原译）

There is nothing in this world constant but inconstancy.

It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.

不变只是愿望，变化才是永恒。

普遍的做法是，选择一种方法，试试看；如果失败了，没关系，再试试别的。不管怎么样，重要的是先去尝试。（基本可以保留原译）

第12章

Sharp Tools

干将莫邪（保留原译）

A good workman is known by his tools.

巧匠因为他的工具而出名。（保留原译）

第13章

The Whole and the Parts

整体部分（保留原译）

I can call spirits from the vasty deep.

Why, so can I, or so can any man; but will they come when you do call for them?

我能召唤地下的幽魂。

这我也会，什么人都会，可是当您召唤它们的时候，它们果然会应召而来吗？ （保留原译）

第14章

Hatching a Catastrophe

祸起萧墙（保留原译）

None love the bearer of bad news.

How does a project get to be a year late? ... One day at a time.

带来坏消息的人不受欢迎。

项目怎么会被延迟了整整一年的时间？……延迟的时间是一天天积累下来的。（保留原译）

第15章

What we do not understand we do not possess.

不了解，就无法真正拥有。 （保留原译）

O give me commentators plain, Who with no deep researches vex the brain

哦，赐予我朴素的评论者吧，他们不会有深奥的钻研让人困惑不解。

（这句话因为涉及到古英语诗词，具难翻译。多亏了在英美文学、尤其是18世纪和19世纪的英语文学方面颇有研究的Aliza Zhao博士的帮助，这句翻译自出版以来还没有收到来自读者的修订意见。）

第16章

No Silver Bullet – Essence and Accident in Software Engineering

没有银弹——软件工程中的根本和次要问题（保留原译）

There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

在未来十年内，无论是在科技还是管理方法方面，都看不出有什么突破性的发展，能够独自保证在十年内大幅度地提高软件的生产率、可靠性和简洁性。

删除的内容: 没有任何一种单纯的技术或管理上的进步，

删除的内容: 独立地承诺

第17章

Every bullet has its billet.

Whoever thinks a faultless piece to see, thinks what ne’er was, nor is, nor e’er shall be.

生死有命，富贵在天。

任何人若想看到一件完美无暇的作品，他所想的那种作品过去不存在，现在尚未出现，将来也不会出

现。

删除的内容: 那些想看到完美方案的人，其实在心底里就认为它们以前不存在，以后也不可能出现。

第18章

For brevity is very good, where we are, or are not understood.

不论别人是否明白我们的意思，描述都应该简短精练。

删除的内容: 我们理解的也好，不理解的也好

第19章

I know no way of judging the future but by the past.

You can never plan the future by the past.

只能根据过去判断将来。

然而永远无法根据过去规划将来。 （保留原译）

Agile软件开发丛书

PEARSON Addison Wesley

有效用例模式

Patterns for Effective Use Cases

Foreword by Craig Larman

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

UMLChina 指定教材

清华大学出版社

《人月神话》国内评论集之二

UMLChina 收集整理

查看评论

以下文字均摘自网络，原文中的错字不作修改。

Jplateau

我女朋友是做外贸的，一天晚上她一直捧着我书架上的一本书在看，很投入的样子，后来她叫住我很认真的说：“这本书写的很好，我明天可以带到办公室接着看么？”那本书是《人月神话》。

http://kaka.rootcn.com/shadowstar/essay/engineering/thinking.html

关于我们的思考——“项目开发”及读《人月神话》有感

项目开发终于结束了，按项目流程，我应该写一份《项目开发总结报告》。拿来“GB856T——88”标准文档，有框架指导我该写些什么，但是怎么能让这些框架协束缚了自由的思想呢？于是决定换一种形式。

项目开发接近尾声的时候，也是最关键的时候，有人送来一本《人月神话》。我很幸运能在这个时候读这本书，因为会有很多思考，关于项目，关于团队，关于我们……

……

四、完美与放弃

在《人月神话》中有一段被截取，称为“程序员的苦与乐”，在网上广为流传。Brooks 大师用简短的篇幅，描绘出整个程序世界的苦与乐。在这次项目开发中，有两个苦恼体现的比较明显。

一是追求完美。“因为计算机是以这样的方式来变戏法的：如果口语中的一个字符、一个停顿，没有与正确的形式一致，魔术就不会出现（现实中，很少的人类活动要求完美，所以人类对它本来就不习惯）。实际上，我认为，学习编程最困难的部分，是将做事的方式向追求完美的方向调整。”

看来我再次做出了正确的选择，我是个追求完美的人，而且我也一直认为程序员就应该有这种本性。在写程序的时候，甚至对一个变量名我都要反复斟酌，直到选择一个我认为可以表达这个变量的意义的。我并不认为这是在浪费时间：一是有助于对程序的理解和维护，好的程序本身就是注释；二是减少错误发生的可能。这次开发因为时间短，我尝试采用设计->编码->编译的方式来写程序，经常把一个几千行代码的模块写完之后才开始调试。效果不错，因为对设计考虑的比较充分，基本上都是一些拼写上的错误。不过有一个错误却另我苦恼了很久。因为一个结构的成员变量名与函数参数的变量名一样，而这个参数又在多处使用，写的时候，也可能是拷贝代码的时候，很容易把结构名给忘记或多加了一个结构名，而这时又不会有语法上的错误。吸取了这次教训，我把整个程序检查了一遍，并做了一些修改。这段程序我参考的一位大师的原型，令我欣慰的是，他的下一个版本和我的程序做出了同样的一些修改。

另一个“苦恼来自设定目标、供给资源、提供信息。编程人员很少能控制工作环境和工作目标。”后面章节又提到“结构师获得了所有的创造发明的快乐，剥夺了实现人员的创造力”。“实现同样是一项高级的创造性活动。具体实现中创造和发明的机会，产东会因为指定了外部技术说明而大为减少，相反创造性活动会因为规范化而得到增强，整个产品也一样。”程序员要学会放弃一部分乐趣，整个项目也一样，这是我读《人月神话》感触最深的一点。

设计网络认证的时候，本打算用一种简单的认证方式实现，因为这不是重点。Y 提出用证书加自己写的 Socket 类实现认证的通讯过程。这是一个很好的创意，但 VC 写出类无法在 Delphi 里使用，BCB 也不行。封装，回调……大家都搞晕了，只剩下两个字：“放弃”。

由于时间比较紧，产品的功能上也放弃了一些认为是很有创意的，但需求中没有提到的部分。后面的开发中也有类似的问题，每个人都按自己的喜好调用和提供接口，可以说是八仙过海，各显神通。问题主要在于交流。

五、善于交流

……

2．交流方式

人月之所以不能成为神话，正是因为增加人手的同时也增加了人与人之间的交流。

我们在项目一开始就通过 QQ 群组进行沟通，后来又搭建了企业内部协作平台，而且还有不定期的会议。与大师所说的三种交流途径——非正式途径、会议、工作手册——不谋而合。但是，执行的效果却不很理想。是大师说的不对？不是。是我们没有利用好这几种途径。我们很好的利用会议，解决很多细小方面的误解；过多的使用 QQ 群组，一是在登陆 QQ 的时候消息太多而没看仔细，二是这种方式针对小的误解是很有效的，但对于一些系统的全面的理解必须通过文档；但是很少使用协作平台，没有仔细的阅读文档的习惯，也没有写出一个完整文档的习惯。

即使在 QQ 群组的交流中，也没有把问题表述清楚。经常看到：“某某：你那里有问题？”哪里？什么问题？QQ 都是通过 UDP 传输的，这里就不需要发送 ACK 了，对方不在线可以通过服务器中转，到论坛发一个贴子或直接发送邮件。接着有人回答：“不可能啊？”什么事情都可能发生的，只要能满足条件。域名都会不易而飞，还有什么是不可能的呢？出了问题首先从自身找原因，发现一切的可能。

……

九、结束语

本文只是我在项目开发和读了《人月神话》之后的一点感想，没有谁的对与错，对于本文一些观点的正确于否欢迎与您一起讨论。本文只是按照项目流程，挑主要的部分叙述了在开发中体会到的一些感想。这次开发让我学到太多太多的东西，将使我终身受益。还有很多的想法，如团队建设、整体设计、开发模式……希望以后能与大家多多交流，相互学习。

《人月神话》这本书推荐大家都去读一下。它不是良药，不能为你解决实际工作中的问题，但它可以给你带来很多思考，让你变得更加成熟。软件工程是为开发软件服务的，标准不是目的，只是手段。《人月神话》没讲标准，但它为怎样做好一个项目提供了一些参考。它会增强你的自信心，当有人反驳你的观点的时候，你可以告诉他：“Brooks 大师如是说^-^”。

fondtiger

不同的社会经验，不同的思想状态，对读本书的心得也不一样，我在此说说我的读后感，书中有许多非常好的观点，但我只把我感触最深的写下来。这确实是一本很值得多次阅读的好书，每次阅读可能都能从中得到一些提示。

1. 外科手术队伍 The Surgical Team

项目经理在项目的初期必须清楚的估计项目的人月运作模式（时间、人力在项目各阶段的分配），例如什么时候需要出什么样成果，决定了什么时候需要什么样的人加入项目，这是项目经理的责任。

2. 贵族专制，民主政治 Aristocracy,Democracy,System

要获得概念的完整性，设计必须由一个人或具有共识的小组来完成。

有四个问题：

1 如何得到概念的完整性

2 是否要有一位杰出的精英，或者说是结构设计师的贵族专制.....

3 如何避免结构设计师产出无法实现或代价高昂的技术规格说明，使大家陷入困境

4 如何才能与实现人员就技术说明的琐碎细节充分沟通，以确保设计被正确地理解，并精确地整合到产品中。

对 1, 2, 4 的回答基本上都可以找到，但第 3 个似乎找不到。

3. 画蛇添足 The Second-System Effect

讲述的基本都是基于 IBM 360 操作系统以及编译程序等方面的经验，讲述如何避免开发第二个系统的风险，作者认为开发第二个系统的设计师设计出来的系统是最危险的,因此要求他们自律。

4. 贯彻执行 Passing the word

印象比较深刻的是"体系结构设计人员必须为自己描述的任何特性准备一种实现方法，但他不应该支配具体的实现过程。"

5. 为什么巴比伦塔会失败 Why did the Tower of Babel Fail?

讲述巴比伦塔会失败的原因是缺乏交流。

6. 胸有成竹 Calling the Shot

主要讲述如何计算编程时间，以及提出几个人的经验算法。

讲述的各种算法可能都不太适合与现在的高级语言，但 Portman 的观点仍然适合现在，即编程人员实际的编程时间只有 50%，其他的时间都花在了无关的琐碎事情上。

7. 削足适履 Ten Pounds in a Five-Pound Sack

主要讲述程序占用的空间等，在 70 年代比较突出，但现在好多了。

8. 提纲擎领 The Documentary Hypothesis

说明文档的作用

9. 未雨绸缪 Plan to Throw One Away

唯一不变的是变化本身。

在大型项目中，项目经理需要有两个和三个顶级程序员作为技术轻骑兵，当工作繁忙最密集的时候，他们能急驰飞奔，解决各种问题。 讲述技术人员与项目人员的互换是，对我有一定的提示，但图中 IBM 的两条职位晋升线，不太理解。

10. 干将莫邪 Sharp Tools

主要讲述项目中管理好各种工具的重要性，项目经理首先要制定一种策略，让各种工具成为公用的工具，这样才能使开发、维护和使用这种工具的开发人员的效率更高，这种工具可能是开发人员开发出来的，也可能是使用现有的，可能是通用的，也可能是专用的或个人偏好的。比如：文档编写工具、开发工具（包括各种不同开发平台）、调试工具、测试工具、数据库工具、版本管理、项目管理工具等。

11. 整体部分 The Whole and the Parts

一读这一章，就让我感触颇深，特别是这句话"BELL 实验室监控系统项目的 V.A.Vyssotsky 提出,'关键的工作是产品定义。许许多多的失败完全源于那些产品未精确定义的地方',细致的功能定义，详细的规格说明，规范话的功能描述说明以及这些方法的实施，大大减少了系统中必须查找的 BUG 数量"。虽然这句话的意思只是说明精确定义产品将减少 BUG 的数量，但我看到了系统分析的最重要的工作——产品定义。现在，许多 开发人员嘴里口口声声说也做过需求调研、系统分析、系统设计，但大多数没有涉及到产品定义的深度，严格意义上不能叫做系统分析。这句话对我的以后想从事系统分析工作有很大的帮助。

这一章余下的内容，也值得一看，虽然有些地方有些过时，但剔除 BUG 的设计以及部分测试/调试方法仍值得一看。

12. 祸起萧墙 Hatching a Catastrophe

这章节说明使项目进度拖后的最大原因不是重要的事件，如新技术、重组等，而是一些琐碎的小事，每件小事只耽误半天或一天时间，但这种小事多以后，将使项目的进度严重拖后。

项目对于公司就如程序对测试工程师一样，如果不了解它，它就是一个黑盒子，如果不打开这个黑盒子，你可能永远不知道盒子里面有什么。

这部分描写项目经理以及小组主管的一些心理，值得一看。

13. 另外一面 The other face

本章说明程序的另一面——文档。

不了解，就无法真正拥有——歌德，作者引用的歌德的话来描述文档对客户的重要性，提出客户需要什么样的文档以及文档的格式和包含的内容，指出当时存在的大多数文档只描述了树木，形容了树叶，但没有整个森林的图案。

想想，这种情况在现在仍然没有改变。于是作者提出了两个观点：

1. 流程图：流程图是被吹捧得最过分的一种程序文档。许多程序甚至不需要流程图，很少程序需要一页以上的流程图

2. 自文档化（self-documenting）的程序：提出文档与程序合为一体，能很好的解决文档与程序分开造成的文档过时的问题，并说明了在程序中加入文档的一些方法和技巧。2002 年，我看到一位网友关于文档与程序合一的文章，当时就觉得是个好方法，没想到 70 年代，老美已经提出来了。

14. 没有银弹-软件工程中的根本和次要问题(No Silver Bullet-Essence and Accident in software Engineering)

这是一篇论文，发表于 1986 年,我自认为我的理论水平没有上升到可以对他的论点和论据做出怀疑或质疑的结论，我只是说说我的感想。

人狼是传说中的妖怪，只有银弹才能杀死他。作者认为软件项目具有人狼的特性，因为软件项目也可能变成一个怪物，一个落后进度、超出预算、存在大量缺陷的怪物。

作者通过软件系统的内在特性复杂性、一致性、可变性和不可见性来分析说明了软件天生就没有银弹。

作者试图通过分析软件问题的本质和很多侯选银弹的特征来探究其中的原因。他行动的第一步是将大块的“巨无霸理论”替换成“微生物理论”。这个变化的过程告诉你，进步是逐步取得的，伴随着辛勤的劳动，对规范化过程应进行持续不懈的努力，而这个努力的过程相应的就诞生了软件工程。作者对软件工程诞生的原因做出这样的解释，我觉得符合外国思维的特点，这正是国人所缺乏。记得有一位朋友说过，中国妈妈与德国妈妈的区别，他说，如果手里拿的针掉到地上了，中国妈妈的第一反应是估计针掉下去的范围，然后在这个范围里面找，可能很快就找到了，也可能一直都找不到；但德国妈妈不同，她会拿一根粉笔来，把整个屋子画成一个大圈，接着把大圈分成许许多多的小圈，然后再到每个小圈里找，虽然比较慢，但最终肯定可以找到。仔细想象，大多数情况下，中国妈妈都会找到得比较快，这确实符合大多数中国妈妈的思维习惯，每个中国妈妈都这样找，这好象是与生俱来的本事，但为什么德国妈妈没有这个本事呢？是德国妈妈笨吗？为什么中国妈妈也有找不到的情况？而德国妈妈，虽然速度慢了点，却始终能够找得到？如果把这件故事推而广之，多年以后，德国妈妈创建了找针工程，她通过多次找针的实验数据，分析出针掉到整个房间中各个小圈的概率，总结出针在哪个小圈的概率最大，很快就可以找到针，找针速度早已高过中国妈妈，而中国妈妈还在依循与生俱来的本事。你能说德国妈妈笨吗？为什么中国妈妈和德国妈妈会有这么大的区别？是德国妈妈把大块的“巨无霸理论”替换成“微生物理论”吗？我觉得是是，你说呢？作者在后面的论述中用数学和物理的发展为例子也说明了，这种思想的成立。

余下的作者把软件工程按“巨无霸理论”替换成“微生物理论”的过程详细的说明，值得看，我关注的不是具体的内容，具体内容可能有些不合适宜，我关注的是作者的思考方式以及处理方法，这是非常重要的。

在“以往解决次要困难的一些突破”和“银弹的希望”章节，从概念上讲述了软件的发展，其中讲到“专家系统”时，使我想起一部科幻电影，忘了电影名字了，有个情节大致是这样的，一位非常有经验的主管死后，有一名较优 秀的下属接任，但这时出现了一位非常厉害的敌人，这位新主管无论如何也战胜不了敌人，这时想起了以前的主管，心想前主管一定有办法对付这个敌人，而前主管的大脑就存放在系统里，于是新主管调出前主管的大脑，把敌人的各种特征都描述给'他'听，就好象前主管仍然活着一样，他与前主管的大脑通话后，前主管的大脑告诉了他对付敌人的方法，后来通过这个方法真的把敌人打败了。这是否专家系统的最佳境界呢？

还有讲到“自动”编程章节时，使我想起我以前也有过类似的想法，但没想到这些想法竟然早就有人提出过。还有记得“图形化编程”好象也风行过一段日子。

15. 再论《没有银弹》No Silver Bullet Refired

看完再论《没有银弹》后，虽然作者说有不少人对他的观点持反对或不同意见，但我始终觉得他的观点是对的——根本和次要问题的划分以及定义。作者认为软件开发困难的部分是概念的结构，如规格化、设计和测试等概念的结构，而不是概念的表述和实现概念，虽然实现概念可能占用了小于 90%的时间，就如现今的软件开发一样，系统分析通常占用的整个项目开发时间不超过 20%，而 80%的时间花在编程上一样。

http://www.ccnews.com.cn/03.2/h.renyue

好好读书 [::坚持日记::]- 叶 子 @ 18:22:35

呜呼,欢乐无声,幸福是卡片

下午去逛书店,本想选几本适合中学生的读物给我表妹文怡

不想看到了< 人月神话>,好熟悉

看介绍不错,买了

刚才网上查了查批评的声音居多，主要是对译者不满意,大家都更信任 ENGLISH VERSION，尽管可能真正看过原版的人寥寥无几，我还没看中文版的,英文版的自然不用说了，看之前已经很好奇了,是炒作还是真的有魅力?

http://www.dogn.net/artdisp.php?id=40022

……

第三，对于开发人员的选用，我发现，美国人是非常注重选用优秀的开发人员的。Martin Fowler 曾经开玩笑的说，如果给他一批水平不高的开发项目，他会考虑全部解雇，重新招聘。《人月神话》中也说，如果 200 人开发一个项目，其中 25 个人最能干，那么会考虑解雇其余的 175 个人，让项目经理来编程（当然，后面还有一些抉择分析，这里断章取义了）。其结论的基础是基于以下研究结果：优秀的开发人员和差的开发人员，其效率之差可以达到数量级。另外，从管理的角度来说，只有人多了，才会有管理问题，当团队规模控制在一定的范围内时，便不会有太大的管理问题。

Zhangmin

……

分工使得每个程序员只关注于自己的工作领域,没有时间也没有精力去关心系统的其他部分,这样就带来了系统开发中的本位主义,更严重的是,系统中存在的错误被详细的分工掩盖了起来,最后往往等到系统测试时才发现错误,这时修改系统已经太晚了,只好在上面进行修补,最后整个系统变得如同泥潭一般,让项目组不能自拔,究其原因,往往是因为分工导致了互不了解,最后大家彼此的工作被相互抵消了,工作的越努力,给其他部分带来的麻烦和缺点越多,这就是软件开发中非常具有讽刺意味的现实.这又往往被理解成为需要更多的沟通,其实不然,因为只有充分了解对方的工作,才能够做到有效的沟通,而分工带来的隔阂使得充分的沟通成为不可能的事情.<<人月神话>>中所谈到的沟通问题,其根本原因就出在分工上面,而作者寄希望于加强沟通来解决问题,我认为也是不现实的.

二.分工导致了对于效率的盲目追求.

由于大家都说:"分工提高了工作效率",于是管理者在此论断的指导下,往往会迫于市场,客户的压力,而盲目要求提高开发的效率.而正是由于分工的存在,每个不同岗位上的程序员都不能意识到这样做的错误性,而反抗这样的决定往往又会背上怠工的罪名,最后大家都听从了.最后,在系统没有明确认识之前,就开始了系统分析,设计,编码,测试.这样做的结果就是"欲速则不达",最后得到的结果往往是,在系统开始时,进行时,似乎都一切正常,等到系统快结束时,问题开始出现,而造成项目一拖再拖. <<人月神话>>中有这样一句话"好的烹饪需要时间."其实软件开发也一样,在对系统充分了解之前开始工作,盲目推进工作,结果就是工作的越多,其中的错误也越多.表面上看提高了效率,实际上是以牺牲软件的质量为代价来追求的速度,而最后当矛盾爆发时,已经太晚了,这就造成了软件危机.究其原因,就是"分工提高了效率"这个似是而非的前提所造成的,正确的说法应该是:"在一定的情况下,分工可以提高开发效率,但在一定的情况下,也有可能降低开发效率."表面上的开发效率提高,并不是真正的开发效率提高,这种表面上的提高往往隐藏了对于系统的不了解,埋下了系统失败的祸根.

http://www.xys.3322.org/xys/ebooks/others/science/report/beijingyanjiusheng.txt

我大四之后，考上了本校的研究生……，我本科的同学，毕业第一年的工资税前大约在 3000－4000，福利什么的不算，在我读研二的现在，他们的工资已经普遍涨到税前 4000－5000 左右。 我们在实验室工作的日子，一周 6 天,每年加班的日子比在公司工作的人多 52（每周六）＋7（五一）＋7（十一）－7（我暑假放假的日子）天＝59 天，寒假和春节假期相抵。 但是还有一点，那就是我们每天都是从早上 8 点工作到晚上 10 点，每天要多工作 4 到 5 个小时。 大家可以算算，我为了这 600 人民币一个月的工资，一个月要加班多少个小时？写＜人月神话＞的家伙如果到我们实验室转转，估计这本书的下一个版本就会多开一个章节了。

http://sdd.ttcone.com/lu0/web/tips/20030305.html

……

这是本栏的第一篇，也是我最有写作冲动的时候. 人月神话这个题目来自于<<人月神话>>一书，讲述的内容也来自于人月神话一书. 只不过讲述的内容换成了我所经历的.

编写代码也已经数年了. 说到编写代码的能力，只怕也不会有一大堆人超越我. 但是，领导终究是领导，始终会有没有学过管理或者尝试以流水线理论来主导软件开发的领导. 而且，在国内，这样的领导并不少见. 这些领导的共同只处在于：目标产品是定下期限要完成的，所有的东西是可以以人手定量的，而人员配备"按照正常情况足够满足开发进度". 在这一前提下，形形色色的变种会出现.

有些是不能明确说出功能列表的；有些则是没有能力知道开发中可能遇到的障碍点在哪里的；有些则不知道团队人员的真正能力的；...；更有些是集以上之大成的. 在有些时候，我也是其中的一员.

说到这里，要说一下人月神话. 我们所有的进度都是以 人月 代码产量来衡量的. 而增加"人"并不能缩短"月"的量.

一个目标产品本身能有多大的代码量大致不会和预算的相差很多. 这时我的经验, 当然也有一些连代码量也估算不准的 LEADER. 我们通常会最终会将代码量分解到每个模块, 并且根据程序员的工作能力来分配进度要求. 在很多情况下, 遇到进度失衡的时候, 第一反应是增加人手. 但是事实上增加人手的项目不到 10%能准时解决. 很多情况下, 增加进去的人手并不能真正进入工作, 因为模块已经无法细分一小块出来给新加入的人手. 又或者新加入人手熟悉现有代码结构的时候已经到达项目终止时间.

而人月代码产量本身就不是一个固定的值. 我的最高写作时刻可以达到 1600 行/天. 真的就是 32000 行/月了么? 不! 更多时刻的代码产量在 200-300 行/天. 也有很多一个算法就花费 1 天. 变得只有 100 行/天的情况. 真正比较客观的状况, 根据最近 3 年的状况, 5000 行/3 月是比较客观的量. 这是 C/C++的速度. 是我的速度, 其他程序员有这样的效率么? 真正能超过的并不多见. 即使是这样的代码效率, 也并不适合将计算进入商业产品的进度考虑.(个人完美产品和商业完美产品将在以后有写作欲的时候写) 因为很多难点并不是因为降低人月代码产量就能够攻克的.

我本人目前比较倾向的时间分配,也是比较真实的时间分配, 没有难点的时间分配

20% 代码编辑

30% DEBUG

30% 文档

20% 保留时间.

这就说明即使在没有已知难点的状况下. 有 20%的保留时间仍然有必要. 因为很有可能 1 个小小的数学逻辑就能让你忙上半天一天. 这并不是不专心, 而是疏忽导致的. 而且从来就没有人能避免疏忽. 而 30%文档时间有时并不能完成很漂亮的文档.

了解了这个神话, 我们就可以采取主动行动.

1.首先, 不要低估任何一个产品的难度, 难度估计得高点总是没有错的.(我曾经犯过多次这样的错误) 这样, 在确定任务进度前争取更多的时间.

2.很显然, 既然有可能在任意时刻发生问题, 为什么不提前多干点呢? 很少有人愿意这样. 但是我的经验是一定要提前多干. 在最近的 2 个项目中, 都是提前很多时候完成了大部分的工作. 90%的东西完成了, 而产品交付时间则剩下 1 个月. 眼看可以轻松了, 却仍然忙着攻克最后的难点, 到了最后一天才真正完成任务. 险得很. 按照时刻表完成进度的程序员都一定会翻船. 不信! 哼, 随便找一个去看看. 我很自信这点的判断.

<<人月神话中>>有着好的程序员可能效率比糟糕程序员高 10 倍的可能性.在我的人月神话中确实有着好的程序员比糟糕的程序员速度快上 10 倍的例证. 当时团队中一天无法完成一个极度简单功能的 PROGRAMMER.(不知到此人现在怎么样) 但是在人月理论中, 这样的人也照样要占着进度表的一条...

……

http://jjhou.csdn.net/feedback-jjhou-csdn.htm

……

“侯捷现象”，真有那 严重吗？前些日子我去书店买《人月神话》，没有遇到一位知道人月是一本计算机书的店员。就算我们在这里吵得热火朝天，社会对此的反应也只是死水微澜而已。其实一个技术作家，即使到 Charles Petzold 这 地步，对大众的影响力也是非常有限的。

……

http://www.techstar.com.cn/gszx/h6-2.htm

……

我们的开发人员结构还需要优化。软件开发团队合理的结构是一般是金字塔形的。为了保证概念的一致性和完整性，大主意必须由一两个人来通盘考虑并做出决定。关于这方面问题，《人月神话》里有非常透彻的描述，包括一些量化的参数。

……

http://www.mypm.net/case/default.asp?caseID=53

icecloud(冰云)

外国大学一般软件工程课有 2 本教材，一本就是《人月神话》，一本是那个《软件工程--实践者的研究方法》。可惜啊，中国只讲了一半。

afiy

该书非常好，希望能读百遍，别的说了也没用，关键在领悟，实践，其他得都没有用，如果你不能做到这 2 点，什么书都没有用

johnsonqian(约翰逊)

这本书挺好的，尤其是作者的基本上所有理论在 20 多年后仍然很有指导意义，这点在 IT 领域是不容易的。

而且，这本书不是具体教你怎么做，而更倾向于告诉你为什么这样做。现在很多的人开发软件要写文档、要做计划、要做评审，而却不知道为什么要这么做，从这本书就可以找到很多这样的答案。

当然，这本书算不算经典就不必评价，经典的书未必真好，好的书未必经典。关键是书是否开拓了你的眼界和思维，从这个意义上说，这本书做到了。

mnm0756

我觉得里面有很多东西很有道理。

最近刚碰到这一件事：开发经理把写详细设计书的任务交给我们来写，有模版但是每个人还是写的都不一样，很多人，认为自己发挥的时候到了，刻意写的和别人不同，最后只有开发部经理自己重写了一边，才算完事。

人月神话中就提到文档一般只能由一个人写，如果很多人写会很乱的，每人的思想都不同，到最后拼凑在一起的东西，只能是垃圾。

richymatin(寸铁)

同意楼上上所说的！我正在看此本书籍，有很多的东西如楼上所说的一样，这本书从 1975 年已经出来了，能到现在还有很多的读者，应该有它的经典的地方！值的一看！

sandygood(斑竹)

我觉得挺好，有很多指导意义，其实许多其他同类的书籍如《快速软件开发》中所以写得如此之好就是因为它踩在《人月》巨人的肩膀上。要想《人月》出生于 20 年前。不过只靠一本《人月》就想有什么质的飞跃是不可能的，还需要借鉴许多其他的经典书籍。

ruihuahan(飞不起来的笨鸟)

翻译水平最差的一本书

xjxie(谢飞扬)

看过了，很有指导作用！

wks9527(空值异常)

我看那本书没有期待中的那么好，说教的太多，例子又是过时的 70 年代的东西。对软件工程的阐述不比 RUP、XP 等书籍清楚、深刻，坦白的说，我看了一半就放下了，我认为这本书的宣传成分很重，且言过其实。当然，也可能是我的水平不够。一家之言...

cccbuiler(建造者)

是真的吗？

我原来就听过这本书，在书店中也见过

但是没有时间看，现在好多书在等着我看

如果好的话，一定要好好研究一下

现在在研究<thinking in c++>，挺有启发的

hell113(一程)

记住，它是

二十年前

的经典!!!

世事变幻，二十年，许多当时的问题，在现在看来，已经不是问题了。

我有电子档，需要的话留言给我。

qiuqiupeng(爱国者)

简单又有说服力，把我从设计人员带到了管理阶段的门口了

一本就是《人月神话》，一本是那个《软件工程--实践者的研究方法》。我 都拜读过我觉得前者对我更有帮助，后者的高级软件工程课题很值得研究哦，特别是静室软件工程，我认为她代表着一种全新 的质量工程思想，有兴趣的朋友可以去拜读一下 MILLS 写的《静室软件工程》机械工业出版社有中文版的

leeapple(小龙)

买了一本,有不小的收获.不过还没有看完.

gosling(小鹅)

我现在只读了 3 章了,但读到第 1 章就被吸引着了好久没有看到这种好书了

chinatcp(平凡)

基本上总结了我们公司大项目失败的原因。

wt13(饿狼．ＳＥＸ)

因为之前看过一本<<快速软件开发>>(或者名称类似吧，不太记得了)

感觉比<<人>>好，里面讲得内容都差不多，而且要比后者有趣得多

所以现在看<<人>>电子版，只是随便浏览了一下

一家之言

fjian37(剑客)

首先这一是本好书，不容置疑;而且只有项目组长或相关的人员才能找到感觉,一般的程序员收获也不大，更别说才入门的。

就我的经验而言，我以前也做过项目组长，但是不很好，现在在一家新公司里学到很多项目方面的东西，而且也能从书中找到我们公司的影子（我所在的公司在国内或国际的某一个领域站有一席）而且也很成功，（国内很多公司没有这种项目管理的方法）

我建议先看先学，先领悟，对你以后的发展绝对有好处!信不信由你

ltf_ty(兔八哥)

我也认为《快速软件开发》比《人月神话》好而且有趣，《人月神话》是本好书，但毕竟是 20 年前的，出版社的大吹大擂是吵作！

zh57(阳光之路)

我只有电子版的，Adams Wang 翻译。

如果时间不多的话，建议看电子版的朋友从后往前看，前边七几年写的东西，确实有很多有价值的论点，经过 20 多年的实践，已经成为共识，佩服作者的眼光。不过书中的例子实在太老，感觉不太好理解，对论点帮助不大。所以建议先读后面 20 年后的总结和回顾，有时间再看前边。

yehuijun(yehuijun)

《人月神话》是本好书，至少要比我见过的国产的任何关于 软件工程的书要好，要实用。我们不应当来不急去批判它，毕竟我们目前国内的多数软件公司和人家 20 年前的公司比还要差的远。我们好好学习吧，把有用的留下，没用的丢了，别客气，也不用藐视人家，人家又不比你差

telescope(望远镜)

幸亏我没买《人月神话》，我就知道带“神”字的书太玄!!

我只买了《快速软件开发》，但一直竖着耳朵关注《人月神话》，因为我不知道它是讲什么的！

听了楼上二位的解释，这下我就放心了，因为我知道《快速软件开发》是干什么的。

Max_LBY(七彩狼)

花了三天的时间，一口气读完了它。感觉受益匪浅。

它讲述了软件开发中存在的最最本质的问题。而这些问题，到目前为止也没有得到很好的解决。

虽然，这本书是二十年前所写，其中相当多的例子在如今看来也是不值得一提。但是，它所反映的软件开发的本质，却一点也没有变化。

正如《没有银弹》中所指出的那样，到目前为止，没有一种方法从根本上解决了软件开发的问题，即使在将来相当长的一段时间内，也不会有。

还有，这本书不是一本教材。

bbface(小费)

花两天实践一口气看完了，有些囫囵吞枣。但还是有些感想，拿出来谈谈：

这本书同时涉及到了编码和管理的经验，确实是一本好书。但作为 20 年前的这本书，显然其中的有些知识已经陈旧了，翻翻 CMM，不得不这样说。回看现在大中专、甚至是大学所教的软件工程课教材，实在是有些不敢恭维。软件工程是一个实实在在的课题，它不单单是技术上的问题，还有管理上的问题。有计划，有规范，有管理，有技术才能把软件做好。我在公司里做事，人家问我能不能做一个好软件，一个大软件，我置之一笑。什么是“好”软件，什么是“大”软件。没有市场的“好”软件、“大”软件意味着什么。做软件的做过市场调研吗，需求分析到位吗？我看现在的软件公司就很难做到这点，更不用说管理了。跳到技术这方面，许多公司都是接到什么订单都做（市场竞争身不由己，原始积累的过程），但对自己的核心竞争力、核心技术却没有一个发展计划。许多在市场上有比较大占有率的软件，其研发公司都具备自己的核心竞争力。所以，做软件在将来也会出现专业化、专业领域的分层，不是讲我使做软件的，而是讲我是做什么方面的软件的。可以看到，软件的开发和管理矛盾重重，不管是 UML、CMM 或是其它都任重而道远，现在是不过是一个开端而已。

weizhenyu()

《快速软件开发》(斯蒂芬.迈克康奈尔)和《人月神话》除了都是经典著作，都是讲软件工程范畴的东西外，比较意义真的很强吗？？？

tao637(彼岸)

她让你思考。。。。。。

cenwangsky (屹康)

《人月神话》书中对首席程序员、副手、管理员定义如下：

首席程序员。他亲自定义功能和性能技术说明书，设计程序，编制源代码，测试以及书写技术文档。他使用例如 PL/I 的结构化编程语言，拥有对计算机系统的访问能力；该计算机系统不仅仅能进行测试，还存储程序的各种版本，以允许简单的文件更新，并对他的文档提供文本编辑能力。首席程序员需要极高的天分、十年的经验和应用数学、业务数据处理或其他方面的大量系统和应用知识。

副手。他是首席程序员的后备，能完成任何一部分工作，但是相对具有较少的经验。他的主要作用是作为设计的思考者、讨论者和评估人员。首席程序员和他沟通设计，但不受到他建议的限制。副手经常在与其他团队的功能和接口讨论中代表自己的小组。他需要详细了解所有的代码，研究设计策略的备选方案。显然，他充当外科医生的保险机制。他甚至可能编制代码，但针对代码的任何部分，不承担具体的开发职责。

管理员。首席程序员必须在人员、加薪等方面具有决定权，但他决不能在这些事务上浪费任何时间。因而，他需要一个控制财务、人员、工作地点安排和机器的专业管理人员，该管理员充当与组织中其他管理机构的接口。Baker 建议仅在项目具有法律、合同、报表和财务方面的需求时，管理员才具有全职责任。否则，一个管理员可以为两个团队服务。

我的问题是：目前中国的状况是否一般将副手与管理员合一？若是，副手与管理员是否可以合一？这样做的利弊分析？

wuguangyao(小武)

我的看法：

首先，我们都承认，人约神话中关于这一段的描述是正确的，也就是说这种工作方式是科学的。

其次，在我们这个公司是肯定不可能这样去做的。

第三，除了那些作坊式的开发团队，由于他们本身可能只有一两个人，或者两三个人，否则，我怀疑是否有哪个公司的开发团队或者开发小组真正采用了这种工作方式，似乎大家都是在遵循这书中所说的错误的开发方式，（用一大堆人来解决问题，问题又没有被明确的划分成不同的部分，或者不同的部分之间的接口描述的不清楚）。

第四，作为技术口上的负责人要想对于财务方面（例如人员的薪水和奖金进行控制）在中国好像本身也是个神话吧？更别说让管理员来控制了，当然如果情况比较理想的话，在有些时候，有些企业，或许首席程序员可以对有权利决定人员待遇的领导提出相关的建议，至于听不听，那就是另一回事了。

第五，我个人的看法，如果遵循书中的描述的话，副手和管理员显然是不能合二为一的，因为前者的工作内容是围绕技术管理的，后者这是围绕人员和财务管理的，性质是完全不同的。对于相应人员的技能要求也是完全不同的。

最后声明，因为在下自己也是个没有什么经验的人，同时没有在规模较大的公司待过，所以上面的观点很可能非常的片面或者浅薄，谨代表个人意见，请各位高人指点。