第 30 期（2003 年 11 月）

【新闻】

1 Intelliun使Executable UML成为现实…

【访谈】

9 Alistair Cockburn与UMLChina交流会

【方法】

13 UML中文FAQ

21 基于职责建模（下）

38 用例随笔

40 更多的操作系统访问控制模式

59 远程认证者和授权者模式

67 帐务模式

103 《企业应用架构模式》评论

【人件】

108 国内中文书对《人月神话》的引用精选（二）

交流

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/

本电子杂志免费下载，仅供学习和交流之用
文中观点不代表电子杂志观点
转载需注明出处，不得用于商业用途

UMLChina 公开课

“UML 应用实作细节”

（2003 年 12 月 6-7 日，北京）

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

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

本训练课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，结合大量练习和项目实践，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。

详情请见>>

Flashline 加入 OMG

[2003/10/7]

关注于支持重用资产规约 RAS（The Reusable Asset Specification）成为行业标准

Flashline，企业软件资产管理方面的领先的提供商，今天宣布加入国际对象管理组织（OMG(TM)），OMG 是在分布式对象计算领域的国际标准化组织。作为 OMG 的平台成员（Platform member），Flashline 将参与支持模型驱动架构（MDA）的各种技术规约的创建、扩展以及修改过程，其中包括重用资产规约(RAS)以及统一建模语言(UML(TM))。

Flashline 和 IBM 的 Rational 向 OMG 联合提出 RAS 标准。在加入 OMG 之后，Flashline 将继续致力于协同各种软件社团，推动 RAS 使之成为描述软件资产，例如代码、框架以及模型等，的工业标准，以实现更多的重用。另外，Flashline 将加入遗留资产转换平台特别兴趣组（the Legacy Transformation Platform Special Interest Group），致力于创建提高在遗留资产转换领域工具和过程的互操作性的标准。Flashline 软件开发产品委员会的 William M. Ulrich 同时也是遗留资产转换小组的副主席。

“Flashline 自从 2000 年创立之初就致力于 RAS 的发展。RAS 提供的使用开发性工业标准来描述软件资产的能力将促进软件组织在小组之间或者跨企业地理解及共享软件资产的流程”。Flashline 首席执行官，同时也是 OMG 的代表，Charles Stack 指出，“加入 OMG 将加强我们在软件资产重用领域的责任，我们将继续推进 RAS 及其他标准的发展，驱动软件行业的重用。”

“Flashline 为 OMG 带来了和各种组织协作地成功重用的有益经验，另外，Flashline 还带来了资产重用方面的专家队伍”，OMG 的主席和首席执行官，Richard Soley 博士指出，“对于今天将重用的概念应用到各种软件资产（这里的资产不仅是组件）的公司来说，为重用资产创建统一的工业标准是非常关键的。我们欢迎 Flashline 加入 OMG，并期待它的贡献。”

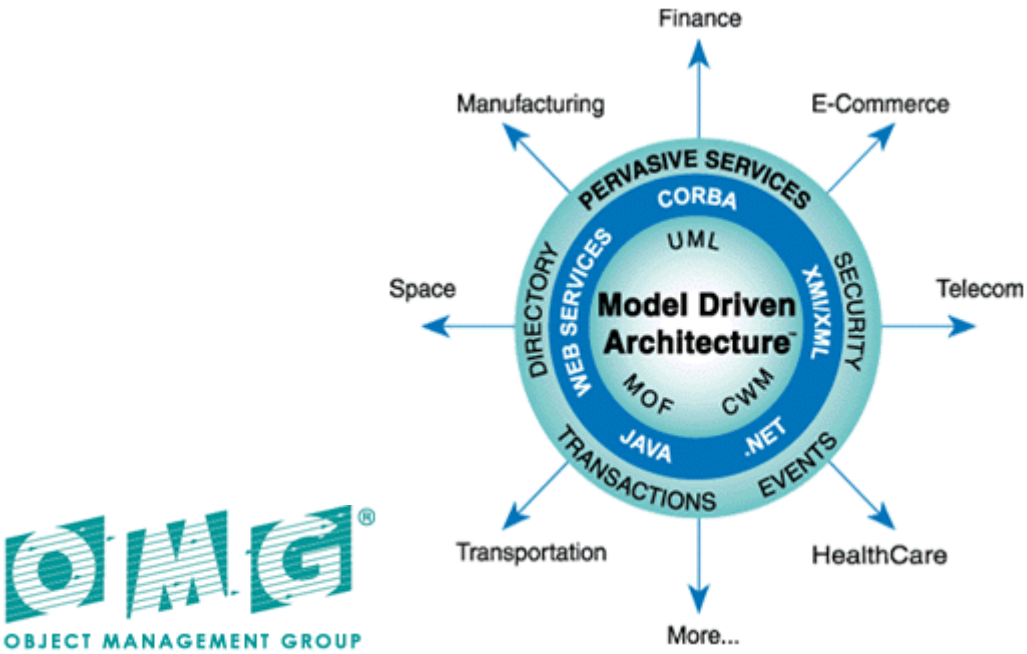
（自 Yahoo，袁峰 摘译，不得转载用于商业用途）

OMG 的 MDA FastStart 计划进展迅速

[2003/10/23]

国际对象管理组织 (OMG) 宣布，模型驱动架构 MDA 的 FastStart 计划获得了极大的进展。

FastStart 计划的授权服务提供商（Qualified Service Providers ，QSP）名单迅速增长，又加入了 19 家新公司：88 Solutions, ADA Software, Advanced Concepts Center, ATC Enterprises, Inc., Cephas Consulting, Cogenture Limited, Compuware, Data Access Technology, Finantix, Herzum Software, Inherit, LLC, iO Software GmbH, M2VP, Metanology, NT/e, Object Computing Inc., Osellus, Pathfinder Solutions 和 X-Change Technologies。



MDA 为帮助其最终用户（软件组织）在其 mission-critical 软件开发活动中集成 MDA 而专门设计了 FastStart 课程，其中清晰地演示了 MDA 在一个约 5 周的小规模项目中的实际价值。每个 FastStart QSP 都有一个已经证明在 MDA 软件开发方面富有实践经验的服务组织，并且承诺遵从 OMG 设计的这个公共的 MDA FastStart 课程。

OMG 的 FastStart 计划负责人 Michael Guttman 认为，“MDA FastStart 计划的效果是显著的，在“The Fastest Way to MDA'” 这一口号的指导下，我们组建了专家团队，最终用户可以在他们的帮助下启动 MDA 项目并使之良好运转运转。更重要的是，这个项目可以在很短的时间内示范其巨大的商业价值。MDA 已经开始起飞，这个时候，正是我们的用户需要 OMG 帮助他们起步的时候”。

OMG 认为 MDA FastStart 计划中包括了在某个公司或其他 IT 组织中使用 MDA 获得成功所必须的所有要素：

评估（Assessment） - 对业务和技术领域的彻底评价。

概览（Overview） －由经验丰富的 MDA 架构师展开的长达一周的分析，用来决定基于组织当前架构的更多技术细节。

交付（Transition） －长达一周，产生一系列建议以及对将 MD 引入到目标组织中的较高层次的一个计划。另外，还包括 MDA 的研讨会以及一个 MDA 实习科目。

OMG 的标准涵盖了从设计、开发到部署、维护等软件开发的各个领域，并且还提供了对未来平台的支持。基于这些标准，OMG 支持最大化投资回报率 ROI（IT 企业成功的关键）的全生命周期的企业开发过程。OMG 的标准跨越了各种操作系统、编程语言、中间件及网络结构以及各种软件开发环境。其中各种建模标准是 MDA 的基础，包括统一建模语言 UML（Unified Modeling Language）、公共数据仓库元模型 CWM（Common Warehouse Metamodel ）以及公共对象请求代理架构 CORBA（the Common Object Request Broker Architecture），后者是 OMG 标准的开放平台，目前应用非常广泛。

OMG 组织的总部位于 Needham, MA, USA 在华盛顿特区有一个美国代表处，在日本、英国以及德国都有国际市场代表处，它是一个国际性的、开放会员资格的、非盈利性的电脑行业规约组织。其成员组织遵循一个成熟而开发的流程参与各种标准的撰写、采纳和维护。

（自 ebizq，袁峰 摘译，不得转载用于商业用途）

SAP 和 IDS Scheer 达成合作

[2003/10/28]

WALLDORF / SAARBRUECKEN, Germany – 世界头号业务软件解决方案供应商 SAP 和领先的流程管理方法和工具供应商 IDS Scheer 今天宣布，两家公司已经达成协议，一起创建业务流程管理（BPM）的全面解决方案。

SAP THE BEST-RUN BUSINESSES RUN SAP IDS SCHEER

IDS Scheer 的 ARIS 流程平台将集成到 SAP NetWeaver 中，SAP 的客户们将第一次能够把业务流程的建模和优化与这些流程的物理构造和执行连接起来。作为 SAP NetWeaver 的一部分，BPM 解决方案提供了一个建模环境，支持各种业务流程模型，如 SAP Exchange Infrastructure (SAP XI)和 UML。

（自 pressi，不得转载用于商业用途）

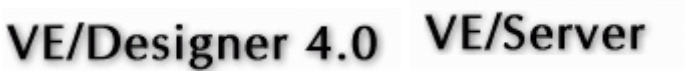
征稿

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

Intelliun 使 Executable UML 成为现实

[2003/10/29]

简化 J2EE 开发的领先者 Intelliun 公司发布了其旗舰产品 Virtual Enterprise (VE) 的 4.0 Beta 版。



VE 是一个快速开发、集成和部署 web 应用和 web 服务的平台。VE/Designer 是一个使用 UML 模型构造完全可执行应用程序的集成开发环境。VE/Server 是一个执行引擎，位于 IBM WebSphere, BEA WebLogic, Sun ONE Application Server, JBOSS Application Server 和 Apache Tomcat 等 J2EE 应用服务器的顶端，提供即时执行 UML 模型的服务。

VE 4.0 达到了软件工程演进史的一个新的里程碑。“作为一名终生致力于把软件开发从艺术转变为工程的软件工程师，我为这一次发布感到自豪”，Intelliun 的创始人和 CEO Iyad Jabri 说，“我们最终把清除了低水平编码的面向对象技术带给了大众。”

VE Standard Edition (SE)将在 2003 年 11 月 11 日提供下载，完全允许免费使用和内部分发。Professional Edition 和 Enterprise Edition 将在 2004 年发布。更多详情请参见 http://www.intelliun.com.

（自 emediawire，不得转载用于商业用途）

Borland 的 企业核心对象“ECO”技术加快软件快速开发的进程

[2003/11/4]

来自 2003 Borland 大会

新的模型驱动运行时平台为企业基于微软.NET 框架的开发提供了 MDA 的快速自动化实现能力。

Borland 软件公司 (Nasdaq NM: BORL (译注：纳斯达克股市编号)) 今天宣布了 Borland ECO （Enterprise Core Objects，企业核心对象）的发布，这是一种用于快速应用开发的崭新的模型驱动的运行时平台。它提供了帮助软件组织将业务解决方案快速推向市场的应用框架。ECO 是 Borland 应用生命周期管理（Borland application lifecycle management (ALM)）的一个重要补充，帮助组织保证业务需求和软件功能的一致性，从而提高其产品的竞争力。

为微软.NET 框架提供的快速 MDA 解决方案

面向使用微软.NET 框架的应用架构设计师、设计和开发人员，ECO 应用程序集有助于由业务蓝图直接创建和维护解决方案。ECO 技术已经整合到新的 Borland(R) Delphi(TM) 8 for the Microsoft .NET 以及 Borland(R) C#Builder(TM) for the Microsoft .NET 之中，为基于.NET 的开发提供了快速的 MDA 解决方案。

ECO 技术被设计来定义应用的业务功能、支持 OMG 的 MDA 驱动，并实现 MDA 中很多手工操作步骤的自动化。MDA 中的相关规约包括统一建模语言(UML(TM))，对象约束语言(OCL)以及 XML 元数据交换标准 (XMI)。

业务价值

软件组织通常通过标准化其内部应用框架的方式来提高开发生产率和保证软件产品的质量，这种方法的成本是很高的。ECO 基于 Borland 公司 2002 收购的 BoldSoft MDE Aktiebolag(2)的技术，该技术是已经得到了实践验证的，Bold 被设计来实现在任务关键环境（mission critical environments）下业务应用的自动化开发。

ECO 提供了一个通用的应用架构。使用 ECO，开发团队可以将时间和精力集中在应用的业务目的上，而不需要花费大量时间来设计应用架构。ECO 平台提供了封装有各种典型业务应用的底层应用服务，开发团队只需要集中精力设计实现业务需求即可，这将显著提高开发队伍的竞争优势。在针对.NET 框架进行修改之后，ECO 现在可以很好地支持包括 C-S 应用开发在内的各种系统开发和业务应用开发。

“Borland 积极参与到各种标准的提案之中，为建模技术增加更多的业务过程规约。许多大型的软件组织不得不开发自己的应用框架，这项工作成本很大。使用 ECO，通过崭新的应用开发方式，我们致力于提高软件开发的生产率，提供一个成熟的平台，基于该平台可以自动化 MDA 中许多需要手工操作的步骤，同时引入加速业务应用开发的实用的高级别的解决方案。”，Borland 软件产品的副总裁，Boz Elloy 认为，“ECO 不但可以减少编写和维护的代码量，同样可以减少风险，支持更高的软件产品质量，我们相信 ECO 至少要比市场上的同类产品领先一年。”

快速的 MDA

MDA 是 OMG 推出的，用来将软件开发提升到一个新的层次，降低应用规约的重要性，甚至消除其影响。用 MDA 的眼光来看，ECO 的目标是提供很多和应用目的相关的核心对象（业务功能），抛弃复杂的实现细节。为了简化实现，基于 ECO 的应用通过定义和提供业务应用开发的各种服务诸如持久性、事务、版本以及整合管理等提供了 MDA 的能力。ECO 提供了一系列的向导（wizards）和组件以支持模型驱动的开发，另外，ECO 还可以与任何 XMI 兼容的基于 UML 的设计工具一起实用。Borland(R) Together(R) 的 UML 类图设计工具已经集成到 Delphi 8 和 C#Builder 之中，它提供了 ECO 运行时平台上 UML 业务模型的创建和管理能力。

ECO 的一个主要优点在于其应用的变更管理，它通过依据模型信息自动更新数据库和必要的源代码减少了应用维护的负担。

ECO 相关资料下载：http://dotnet.borland.com/bdntv/csharpbuilder/eco.zip

（自 businesswire，袁峰 摘译，不得转载用于商业用途）

UMLChina 公开课

“UML 应用实作细节”

（2003 年 12 月 6-7 日，北京）

本训练课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，结合大量练习和项目实践，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。

详情请见>>

简化应用集成的工具

[2003/11/14]

正在迅速发展的面向服务架构（service-orientated architecture）技术将使旧应用与新应用的集成更加简易。

销售商们推出了各种各样的新技术，包括效用计算（utility computing）、自动化计算，网格架构以及 web service 等。所有这些新名词，用一个名字就可以概况，那就是面向服务架构 SOA(service-oriented architecture)。各种 SOA 技术的一个共同主题就是要简化新旧应用的集成。

Microsoft 似乎已经认识到了 SOA 的重要性，并且最近已经做出了响应，宣布其 SOA 架构 Indigo 基于很多新的开发和部署工具，它们将被包含在计划于 2006 年发布的代号为 Longhorn 的下一代 Windows 里。但是，微软还需要和很多已经提供了同类工具的公司进行激烈的竞争。

SOA 实现要求新的应用通过 web services 与旧系统进行集成。OMG 的模型驱动架构将会被广泛接受，因为它支持端到端（end-to-end）的应用开发方式。

这也正是分析公司 Butler 集团的 Teresa Jones 的看法，他最近写了一篇名为“应用开发策略”的文章。Jones 提到，有很多的工具都可以提高编程人员的生产力，但是它们往往都忽视了一个关键的领域。

Jones 认为，“（这些工具）没有提供端到端（end-to-end）的开发环境，而在目前，这一点甚至要比编程人员的生产力更为重要。” Jones 还认为开发环境必须要在为企业需求建模及支持设计、开发、测试、部署和管理方面作更多的工作。“企业的环境是一直在变化的，因此应用也要随之变化。”

她认为当前的建模系统大多存在问题。MDA 将会带来改善，因为它用到了 OMG 的统一建模语言 UML。UML 模型经常被用来生成软件的源代码，因此模型的变化可以直接反映到代码上。“这样，应用的功能开发和改进就得到了简化，因为在这种建模过程中业务人员可以理解建模的内容并且能够在其中有所贡献”Jones 认为，“这使得后期的更新可以更快和更可靠，并使得开发过程更加可控”。

这个观点得到了来自 Compuware 公司的 Edwin Schumacher 的支持。该公司的 OptimalJ 正是 Butler 集团评出的三种最好的 MDA 工具中的一种。另外两种是 IBM 的应用开发解决方案（Application Development Solution）和 Interactive Objects 的 ArcStyler 4.0。

在解释 MDA 和这些工具的好处时，Schumacher 说，“如果你观察现在的开发过程，你会发现，包括业务人员、模型设计师、IT 部们及各种人员都参与进来了，每一步由不同的团队来完成，而他们之间的转交往往是手工完成的，这必然会产生很多无法避免的错误。MDA 将这些过程自动化，并提供了自动化的转换方法”。

（袁峰 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

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

UMLChina 指定教材

清华大学出版社

Alistair Cockburn 与 UMLChina 交流会

think 著

查看评论

2003 年 11 月 8 日，Alistair Cockburn 到京为一家跨国公司的中国分部作指导。当天 UMLChina 也在北京举办“有效用例实践”讲座，在用例技术上，UMLChina 一直是 Cockburn 的追随者。所以利用这个机会，邀请 Cockburn 先生和参加 UMLChina 讲座的开发人员一起交流，并举行了签书活动。

活动于下午 14:00 在海淀图书城富益视点图书中心举行。Cockburn 先生一进门刚好看见他的书被集中到一起，放在显眼的位置，于是持书留影：

图 1 在自己著作前留影

然后上楼和大家交流。Cockburn 先解释了活动会场欢迎背景中的 Actor 小人的故事，“UML 里的火柴杆小人对我来说太难画了，所以我创作了这样一个。”

图 2 Cockburn 画的小人

然后做了开场白：

“在我的主页上有三个汉字，‘守’‘破’、‘离’，日文发音是 Shu-Ha-Ri，和中文发音不同。

图 3 http://alistair.cockburn.us/上的三个汉字

为什么我要把这三个字放在我的主页上呢？因为对于用例（Cockburn 用汉语发音说出了‘用例’二字）的初学者来说，肯定要先遵守一定的规范，经过一定时间的使用以后，用例编写者就会尝试使用不同的方法来编写用例，在这本绿色的书（注：指《有效用例模式》，中译本由清华大学出版社出版）里面，第二部分介绍了如何用不同方法编写用例。在这本蓝色的书（注：指《敏捷软件开发》，中文注释本和中译本由人民邮电出版社出版）里面，介绍了使用敏捷（Cockburn 用汉语发音说出了‘敏捷’二字）软件开发方法来编写用例，这个时候就要用到第三个字‘离’，这个时候，你要用最快的速度和最低的代价来编写。那么，用一个具体的做面包的例子来说，守就是一个从来没有做过面包的人，就需要有一个菜单似的东西告诉他你具体应该怎样去做，第一阶段‘守’最重要的是我要知道菜单给我提供的是一种正确的方式，我只要照着去做，就能够做出面包来。那本紫色的书（注：指《编写有效用例》，英文版是紫色的，中译本是白色的），对，是白色的书提供的就是象处方那样的东西，你只要照着处方一步步去做，就能够写出正确的用例。到了第二个阶段，也就是‘破’的阶段，这时你不满足做普通的面包了，你想做松酥的面包、薄一点或是厚一点的面包，这个时候就需要不同的处方，根据不同的方式，做出不同的面包来。而象我的妻子那样做面包，就和那两个层次不一样了，她只需要凭她的手感，抓一些面粉，倒一些水，弄点鸡蛋在里面揉揉就能做出她想要的面包，这就是‘离’的阶段。到了她那个阶段，她既知道如何根据处方来做，也知道不看处方怎么做。到了敏捷开发阶段，就要求你处在第三层次。就象编程序，一开始要老老实实从第一步做起，到了技术越来越好时，你自己就能判断用什么方法来使编程能满足你的需要。这样我们很容易理解为什么一些人会碰到这样的情况，一名老练的程序员，他熟悉各种编程技术，用这种也可以，用那种也可以。当一个初学者来问他怎样编程时，他会说，你可以这样，也可以那样。这不是初学者想要听到的。初学者总是会问，‘什么是最好的办法‘，老程序员说，没有最好的方法，这样也行，那样也行。其实自然的规律总是有一个‘守’‘破’、‘离’的阶段，初学者总是在‘守’的阶段，随着时间的推移，会到达‘离’的阶段。”

图 4 上文中 Cockburn 提到的著作

接下来大家就平时使用用例技术时碰到的各种难题和 Cockburn 先生展开了交流。最后，举行了简短的签书仪式并和大家合影。

图 5 在回答问题时当场用白纸制作 CRC 卡

图 6 签书

图 7 请中译本译者乐林峰为他签名

图 8 合影

交流会结束后，继续举行了UMLChina“有效用例实践”讲座，由于场地不允许太晚，所以把时间提前到17:00-19:00（即：整个活动是14:00-19:00），临时已经来不及通知。也有朋友来信表示不满，说赶到那里已经没人了，在此在深表歉意的同时，也要说一句：活动14:00开始，您19:00才赶到，猛将兄您是不是也太猛了？

The Addison Wesley Signature Series

A MARTIN FOWLER SIGNATURE BOOK

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

UML 中文 FAQ--V1.0

张恂 著

查看评论

一直很难找到中文的、适合国人看的 UMLFAQ，而英文的 UMLFAQ 往往散落于各处不易收集和消化，导致很多看似简单却很基本、很关键的问题在网络论坛、BBS 上被反复地提出，既浪费了提问者、解答者不少的时间和精力，也给大家学习总结、进一步提高带来了困难。于是，结合自己的所学所想试着编写了一个 UML 中文 FAQ 供各位参考。众人拾柴火焰高，有什么不足之处，欢迎大家多多提出宝贵意见，不断补充，一起来完善！

联系邮箱：zhangxun2001@hotmail.com

什么是 UML？

Unified Modeling Language（统一建模语言）是国际对象管理组织 OMG 制定的一个通用的、可视化建模语言标准，可以用来描述（specify）、可视化（visualize）、构造（construct）和记载（document）软件密集型系统的各种工件（artifacts，又译制品）。

UML 有哪些特点？

UML 的“通用性”主要是指不仅仅可以用它来描述软件，而且还可以用它来描述一般企业或组织的业务流程以及由软、硬件共同组成、以软件为主的复杂系统（即所谓的软件密集型系统），甚至还包括非软件系统。

UML 的“可视性”是指可以通过 UML 一系列的图形符号，组成多种视图（view）来直观、清晰地表达系统分析设计中方方面面的、许多复杂的概念。UML 主要是为了人的阅读和使用而设计的，所以它采用了半形式化的，易于人们理解、交流的形式。

UML 是一种分析设计专用的建模语言，它本身不是编程（programming）语言，不能直接用来生成可执行的软件。UML 是一种抽象层次比 C、C++、Java、VB、Delphi 等文本高级语言更高的图形语言，通过它我们可以抽象地表示用高级编程语言编写的文本程序的逻辑结构和行为。相比传统的高级编程语言，UML 能够更加高效、准确地反映软件设计的方案和思路，是真正用来“设计程序”（design programs）的语言。从这个意义上看，不妨称 UML 为“甚高级”程序设计语言。

UML 基本上不能算作全新的发明，它并非学者教授、科研机构的研究成果，而是直接来自于产业界、工程界的实践总结，是在归纳基础上进行理论升华的产物，其核心内容反映了 30 多年来全球软件工业的领导者在软件设计构造领域的最佳实践和成功经验，因而具有很高的实用价值。

实践证明，OO 分析设计(OOAD)方法比传统方法能更加准确、全面地描述现实世界。UML 是用来表述 OO 概念的一种语言工具，而很奇妙，它本身作为一件产品同样也是用 OO 方法来设计的，这使得 UML 具有传统建模语言所不具备的很强的语义表达能力和非常灵活的可扩展性。

UML 有什么用？

UML 的用途非常广泛，可以概括为“描述、可视化、构造、记载”4 种基本功能，在业务建模、需求分析、系统设计、实现和测试、数据建模、项目管理等阶段任务中均可根据需要采用。

UML 建模是建立软件开发文档的一个有效手段，通过 UML 可视化地描述系统需求，记载软件构成，能够显著地提高文档的质量和可读性,减少编写文档的工作量。

UML 实质上是一种系统分析设计专用语言，通过可视化的图形符号结合文字说明或标记可以帮助业务/系统分析员、软件架构师/设计师、程序员等各种建模者有效地描述复杂软件（或业务）的静态结构和动态行为，包括工作流（数据流和控制流）、功能需求、结构元素及关系、架构组成、设计模式、对象协作、事件响应和状态变化等等。

UML 不能做什么？

UML 不是高度形式化的语言，一般不能用于定理证明。

UML 是基于 OO 方法的通用建模语言，不适合用户图形界面设计、超大规模集成电路（VLSI）设计、基于规则的人工智能等专业领域。

UML 是一种离散型建模语言，适合对由软件、固件或数字逻辑构成的离散系统建模，不适合对工程和物理学领域中的连续系统建模。

本身作为语言，UML 仅仅是一种表达形式，不是建模方法，在实际的软件项目开发中仅仅掌握一套标准的图形符号是远远不够的。用好 UML 首先需要掌握 OOAD 的基本原则和方法，并在一定的软件开发过程（如 UP/RUP、XP 等）的指导下进行有取舍的运用。

为什么要学习和掌握 UML？

首先，UML 对于当前大多数希望进一步改进质量的软件开发团队来说是必不可少或必须的。为什么这样说呢？因为，C++、Java 等源码并不能直观、方便地反映复杂程序的设计：如内部逻辑结构、各种隐含的依赖关系、运行时的状态改变和特殊行为等等。写好的代码仅仅是一种实现方式，很难反映出现象背后的真实本质——设计，因此对于大多数稍稍复杂点的项目来说，仅有代码是不够的。

可视化建模历来是一种成功的工程实践做法，并非软件行业所特有。软件的设计方案在用 C++、Java 实现之前通常隐藏在人们的头脑当中，而设计正确与否是决定软件质量好坏的要害，一个前提是必须用适当的方式把它表达出来。通过程序代码来表达、讨论、评估和选择复杂软件的设计往往是很笨拙的，这一表达方式的空欠唯有通过类似 UML 的建模语言来填补。

其次，OO 方法是当代主流的软件开发技术。世界先进的软件团队和个人早已摆脱了对如何使用具体的平台 API、个别编程语言特性的纠缠，而把更多的精力放在了需求、架构、设计机制和模式等对软件质量有重大影响的核心要素——分析与设计上。UML 作为 OO 建模语言的事实上标准和主要的表达媒介，在这些场合能发挥关键的作用。所以，熟练地掌握并运用好 UML 是当今系统分析员、架构师/设计师以及程序员等软件工程师所必备的一项基本技能。

什么情况下不需要或不适合用 UML？

对于一个特定的软件公司或开发团队，在下列情况下建议不采用 UML：

1）传统的做法已完全适用，对 OOAD 的要求也不高，项目非常成功，无任何改进的必要;

2）开发的系统比较简单，直接用源码配上少量的文字就能解决问题，软件开发文档也无需添加图形来辅助说明；

3）开发人员更习惯于直接阅读源码，用源码交流，这样做不影响工作效率和质量；

4）开发的系统本身不属于 OO 方法、UML 适用范围。

什么情况下应该用 UML？

对于一个特定的软件公司或开发团队，在下列情况下建议采用 UML：

1）OO 方法是项目决定采用的方法论，是整个项目或产品成功的关键；

2）开发人员感觉用源码说明不了真正的问题，希望利用可视化建模语言简化文档，提高交流的效率，准确抓住问题的本质；

3）系统的规模和设计都比较复杂，需要用图形抽象地表达复杂的概念，增强设计的灵活性、可读性和可理解性，以便暴露深层次的设计问题，降低开发风险；

4）公司希望记录已成功项目、产品的公共设计方案，在开发新项目时可以参考、重用过去的设计，节省投入，提高成功率。

促进 UML 普及和应用对于加强我国软件业的实力有什么重要意义？

OO 方法自上世纪 80 年代以来已经成为软件开发的主流技术，标准 OO 建模语言 UML 的问世说明 OO 技术的发展达到了一个新的高峰。推广普及 UML 的应用可以使我国软件开发人员、软件企业和客户乃至整个行业都从中获益：

1）个人

UML 相关知识体系蕴含了非常丰富的当代软件工程先进知识。软件开发人员通过学习和掌握 UML 概念、表示法及相关的软件过程、软件工程技术，能够加深对 OOAD 原则、方法的理解，提高抽象思维能力，从而站在更高的层次上分析问题、解决问题，这是一条快速提高个人软件设计能力的有效途径。

2）企业

对软件企业内部，用好 UML，不但能直接提升企业的软件设计开发能力，而且由于 UML 能形象直观地记录软件设计的核心思想，可以使软件开发管理透明化，促进企业知识资产的保护和增值，促进软件重用和整体效益的提升。对外，由于 UML 是通行的软件行业国际标准，企业在业务交往中有效运用 UML，无论对于开拓国内外产品市场还是保障工程承接、项目外包等业务的顺利开展都大有裨益。

3）行业

积极采用国际通行的软件描述和设计语言 UML，一方面能增加信息透明度，显著降低软件企业之间、客户与开发商之间的沟通成本，减少项目失败的风险，另一方面能促进行业市场的规范化和标准化，增进国际技术交流，整体提高我国软件业的技术水平和参与国际市场竞争的能力。据悉，与 UML 有关的国家级标准目前正在考虑制定过程当中。

UML 的统一性表现在哪些方面？

UML 的统一性至少表现在以下几个方面：

1）随着 OO 技术的蓬勃发展，到上世纪 90 年代初 OO 方法已经多达 50 余种，它们之间既有很多共通之处也存在许多没有必要的细节差异，这妨碍了技术进步，不利于产业的发展。UML 统一了多种互补的、最具代表性、最受业界欢迎的主流 OO 方法，这既是历史的必然，也 OO 方法成熟的一个重要标志。UML 及与其配套的 OO 统一过程（RUP）在实现“合并同类项”的基础上又向前迈出了一大步，不愧为当代 OO 建模方法的集大成者。

2）UML 适用于各个行业的信息化工程，包括电信、银行、保险、税务、办公自动化、电力、电子、国防、航天航空、制造、工业自动化、医疗、交通、商业、电子商务等诸多领域的业务建模和软件分析设计，尤其适合对大中型、复杂、分布式应用系统或软件产品建模，在这些广泛的领域中都可以统一使用一套标准的建模语言。

3）作为一种独立于具体实现的、抽象的表述方式，UML 广泛地适用于各种现代程序设计语言、数据库和开发平台。

4）有了 UML 标准，面向各种不同的软件开发方法和过程（如重载/轻载，瀑布式/迭代递增式），在软件开发生命周期各个阶段的工作（如业务建模、需求分析、设计、实现、测试）中，都可以采用一套统一的概念和表示法，避免了语言转换的麻烦。

5）UML 明确定义了一套公共的内部概念，建立了统一的关于建模语言的元模型，反映了在软件和信息建模技术领域的最新成果。

UML 是如何诞生的？

识时务者为俊杰。为了突破上世纪 90 年代初 OO 方法论混战的局面，1994 年杰出的 Rational 公司 OO 大师 Grady Booch 邀请通用电气公司著名 OO 大师 James Rumbaugh 博士加盟 Rational，启动了 OO 方法的统一历程。他们于 1995 年发表了“统一方法 0.8”。与此同时，另一位超一流 OO 大师、爱立信公司的 Ivar Jacobson 博士也在该年加入了二人的行列。

1996 年，3 人正式把他们的统一成果命名为“统一建模语言”，UML 于此诞生。同时，他们还做出了一个非常重要的决定——把 UML 提交到非赢利性的 OMG 进行标准化，让全世界的软件开发人员都可以自由地分享这一软件史上的重大成果。

于是，在全球软件界具有广泛影响力的国际对象管理组织 OMG 从此开展了一系列 OO 建模语言的标准化工作。1997 年 11 月，UML1.1 经 OMG 各成员投票被正式采纳为行业标准。

UML 是一家之言，或少数派的观点吗？

否，UML 是全球工业界和学术界的领导者协同努力的成果。自从进入 OMG 程序后，UML 就不再由 Rational 一家公司所有或由少数人控制，而成为凝结了百家之长的公共知识结晶。具有丰富企业信息系统和信息工程经验的 OO 大师、Martin/Odell 方法的领军人物 James Odell 为此曾表明放弃自己的方法，并直接参与领导了 UML1.x 系列标准的制订工作。另一位 OO 大师 Coad/Yourdon 方法的创始人之一 Peter Coad，虽然没有直接参与 UML 制订，但却独具慧眼创办了 TogetherSoft 公司（已被 Borland 收购），开发了著名的 UML 集成开发环境 Together ControlCenter，成为 Rational Rose 的有力竞争者。事实说明，UML 的确促进了各种 OO 方法和流派的大融合，在 OO 建模语言领域具有不可替代的地位。

UML 之父是谁？

UML 之父有三位:他们是 Grady Booch（Booch 方法），James Rumbaugh（OMT 方法）和 Ivar Jacobson（OOSE 方法）。人们亲切地称他们为“3 amigos”（即“三高”，类似于大家给予合作举办世纪音乐会的，世界上 3 位顶尖男高音歌唱家的称谓）。

UML 标准有哪些最新进展？

UML1.x 系列的最新版本是于 2003 年 3 月发布的 1.5 版本（www.uml.org）。OMG（www.omg.org）从 2000 年起启动了 UML2.0 标准的制定工作。U2P 组织（UML2 Partners Consortium，www.u2-partners.org）在 UML2.0 标准的制定过程中发挥了主导作用。目前 UML2.0 的上层结构（Superstructure）规范已经在 2003 年 6 月 12 日获得通过。

OMG 已经与国际标准化组织 ISO 开展了合作，预示着 UML 将来有望成为 ISO 标准，值得关注。

初学者如何开始学习 UML？

无外乎几种方式：读书、上网、实践、培训。

最近几年国内一窝蜂地出版了不少与 UML 有关的中英文书籍，但说实话，有点良莠不齐。初学者不管自学还是参加培训，选择合适的 UML 教材/读物是很重要的。

Craig Larman 的《UML 和模式应用》是一本非常好的内容丰富、真正实用的入门教材，在国际上也是用得最多的一本，无出其右者。纵观全书，以实案为中心，脉络清晰，组织老到，深浅适当，循循善诱，非常适合 UML、UP、设计模式的初学者和一直对 OOAD、UML 的价值存有疑虑的结构化人士一读。对于熟悉 OO 的人来说，阅读此书也是再一次享受梳理知识、进行系统性训练的美妙体验。本人尤其推荐第 2 版（内容更新了不少，听说正在引进翻译，值得期待）。

如果希望与大师对话，全面深入地掌握 UML 的基本要领，通过领悟 UML 设计者的思想和意图来达到在实战中得心应手运用 OO 建模技术的目的，建议一定要阅读 UML 之父 Grady Booch 亲自撰写的《UML 用户指南》。本书相当全面，偏重理论分析和概念阐释，这些内容和抽象技术对于真正理解 UML 是非常基本、必不可少的，所以它适合喜欢认真探究一切的读者。

用好 UML 离不开有好的过程作指导。RUP 极其丰富的内容令人生畏，Ivar Jacobson 大师在《统一软件开发过程》一书中从管理者和系统架构师的角度，通过实例分析系统地讲解了将 UML 用于分析设计实践的完整过程，深入浅出，言简意赅，可以说此书正是 RUP 的精华所在。带领自己的团队用好 UML，得此书足矣。

此外，IT 之源（www.iturls.com）、UMLChina（www.umlchina.com）等网站提供了非常丰富的学习资料和参考文章。

UML 规范、《UML 参考手册》内容深、篇幅大，主要面向 UML 工具开发者、专家和研究人员，不适合初学者阅读。

世界上有哪些著名的公司、组织参与了 UML 标准的制订？

历年来，参与 UML 标准制订的一些核心公司和组织包括（1.x、2.0）：

全能型 IT 公司：HP, IBM, Unisys

大型软件公司：CA, Microsoft, Oracle

CASE 厂商：I-Logix, Rational（已被 IBM 收购）, Telelogic

电信供应商：Alcatel，Ericsson，Fujitsu，Motorola

行业组织：OMG

IT 系统集成商：EDS

等等

UML 在业界的影响力和地位由此可见一斑。

感谢您阅读此文！纸质媒体如需转载请与作者联系；本文版权所有者为张恂，保留所有权利；您可以从 IT 之源上获得本文的最新版本和相关资料；以上言论仅代表作者本人观点，与作者服务的公司无关；欢迎转载本文电子版，转载时请注明出处并保留所有原始信息。

*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。

基于职责建模（下）

Alistair Cockburn 著，李巍 译

查看评论

RBD 中的步骤

概述

1. 识别使用的场景；限制设计的边界。识别边界内的场景。排序所应用的场景。从主要场景开始做起，备选场景（alternate scenarios）可被用作变化（variations）。

2. 角色扮演场景，对职责进行评估。

3. 在每个所需承担的职责点上，对场景进行命名以此作为总结。命名一个既有的组件或创建一个新的组件来承担职责。转向业务模型，对象实例图 CRC 卡，或保持对设计进行讨论的任何东西。

4. 确保每个服务提供者具有足够的信息和能力来承担其职责。

5. 考虑场景的变化，以检查职责分配的稳定性。重新扮演一遍最初的场景，以验证其是否可用。

6. 根据测试问题和变化分析对组件进行评估。

7. 询问每个组件是否能够免受未来变化的影响，或者其是否能被业务直接管理。检查组件的生存周期：创建和删除。

8. 产生变化：需求或实现可能随时会在系统的生存期中发生变化；备选路径场景和错误条件。

9. 预演（Run through）不同的场景来审查组件和职责的稳定性。为强化设计而进行所需的修正。

10. 尽可能地进行模拟。

11. 根据级别对组件进行合并（Consolidate）。

12. 如果一组组件位于不同的抽象级别，标记与它们相关的主要卡片及其意图。随后将这组卡片设计成一个子系统。为需要这些卡片的场景取一个有助于记忆的名称，便于日后容易回想起来。

13. 对设计原理和关键场景的处理进行归档。或是在最后，或是在一组相关场景处理之后，文档将会趋于稳定。

14. 确定要归档哪些场景（主要，错误，感兴趣的那些）。

15. 对每个所选的场景以及为何要那样分配职责的原因进行归档。

16. 列出已经存在的可用组件。

17. 详细说明该设计会议级别上的每个新组件。

收集相关场景

在广度和深度（级别）上限制设计的边界。预先确定哪些需要设计，哪些不需要，以及如何知道何时便能完成设计。确定设计在何种级别上能够完成，以及哪些主题（topic）与设计相关。

收集和阅读与当前设计边界相关的场景。在系统的最外级别上，选择与用户一起开发的场景。如果是一个内部子系统，则从在前面设计会议中生成的场景、职责和交互图开始。如果没有任何场景，并且这是项目的入口（entry），则对场景进行全面的设计。

复审“事实分析（Fact Analysis）”方法，并遵循相同的步骤理顺业务模型中的术语，概念和设计片断（design fragments），使其与需求模型以及本次 RBD 会议所建议的保持一致。

首先选择一个简单的场景，其构建了部分系统。添加难度更大的场景。难度越大则需要确定的点就越多，而且应该尽可能早地到达该阶段。

如果你能够非常自如地运用问题域和 CRC 卡，首先选择一个处理较为复杂情形的场景。接下来根据它们应用子系统的顺序选择一系列的场景。选择每个类似分组中最复杂的场景。

意图是为了展现尽可能多的决定点（decision points），以及展示组件间最为复杂的交互。情形越复杂，就越需要尽快这样做。

示例：对于一个为房屋保险的保险公司来说，生成一套位于不同州或国家的房子报价是复杂的，并且需要尽快暴露出这个问题。

如果最复杂情形被证明过于困难，便回退到一个难度稍差的情形上，但是要尽快地重回到这个困难的案例上来。

首先选择主要的成功场景。主要场景能够以最直接的方式来交付主执行者的目标。可适当地将备选场景用来作为一次预排（walkthrough）和角色扮演之内的变化。任何在角色扮演中未涉及到的场景，将会被应用到一个单独的预排和角色扮演中去。如果备选场景展示了主要场景所用组件中令人感兴趣的决定，可将其看作是在主要场景预排期间的一个变化。如果需要发明在主要场景中未展示的组件，便可单独对备选场景进行处理。可以将失败场景（failure scenarios）作为对设计的测试，应用到相关的成功场景中；最后确定失败场景的设计。

识别每个场景的组件及其职责

结合使用“主动”和“联系点”职责。主动职责以一个主动动词作为起始。

示例：

“compute new balance”，

“find all customers with given characteristics”，

“refresh the screen”，

“maintain consistency of customer addresses”，

“introduce transaction”

使用全称形式来陈述主动职责，从而能够识别出相似的职责。

示例

较好: "display"，或"display on screen".

较差: "display triangle on screen."

推论。在设计完成时，许多组件将很可能会说 ，“Display on screen.”在职责复审中，将很容易发现这些组件。复审这些具有相似职责将会产生下列一种设计决定：

(1) 有一个通用组件尚未被识别出，并且这些组件能够将它们的职责委托给这个通用组件，便可将其标记成使用继承的一个可能位置。

(2) 没有通用组件。而是有多态的行为。应该再对职责进行复审以确定它们传达的是相同的意图，并且客户端在访问该职责时能够安全地对组件的行为进行假定。

识别组件所传递的信息（因为它是联系点）。使用联系点代替“状态数据”或“属性”进行论述。可能还没有到「确定组件必须拥有某些数据或属性」的时候。实现很可能会随时演变，使得这些假定无效。这些属性可能会从原始的组件中脱离出来，例如，组件将会被分解开来。

示例

「知道账户上的余额」是账号的一个联系点职责。余额可能是通过动态计算得到的。记录银行账户是余额的一个联系点 (a) 记录所需的职责，(b) 关于余额如何与何时计算，以及结果在何处与是否保存的决定，则保持开放。

要注意重用既有组件的机会。设计团队负责考虑重用与创建新组件的多种选择。无论何时组件的名称都要描述它的*角色（role）*，而不是它的*能力（capabilities）*，寻找已经具有这些能力的既有组件。根据引用它的其他组件的数量以及发生变化的可能性，从而确定是创建一个新的组件（保护设计），还是重用既有的组件（节省开发）。

通用提示

(1) 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，则需要创建一个新的组件，并使一个当前可用的组件实现它。改变引用它的组件的成本很可能会大于引入一个新的组件。如果合适的话，可将既有组件标识成一个协作者。

(2) 如果新组件所需的仅是一个既有组件能力的子集，那么最好是引入一个新的组件。既有组件提供的额外服务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。

(3) 如果组件不是一个模型中的主要概念，而且未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。变化的成本可能要小于引入系统新组件的成本。可以考虑将一些职责移到外围的组件中去。

下面的这个例子具有两种结果：

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。

要注意出现需要一个新组件的信号。

需求模型和业务模型充斥着候选的业务术语和概念。不会将这些接受作为当前设计级别上的组件，除非它们被证明具有对该讨论级别的场景非常重要的职责。这往往会与「组件被展现在不同的级别上」同时发生。有时候设计团队要确定那些在公司内部使用时能够产生歧义的术语，并根据这些不同的意义提供不同的组件。

通常会为一个组件或组件的名称召开竞争提名。有时团队会达成一致，即两个名称中的一个能够更好地传达所需的职责，因此将会放弃第二个提名。有时候，两个提名都无法完全占据上风，则宁愿将这两个都作为备选。在以后案例中，设计团队需要考察其背后是否存在着一般的抽象，或者可能有第三个抽象可以提取这两个一般元素，或者是不是错误传达了这两个名称。

组件可以保护设计的变化。这些才是最具价值的组件。

讨论可能会在某个系统部分将“这样”或“那样”演变上陷入泥潭。如果其他部分的建模将影响到该部分设计，这时便需要考虑命名一个组件来表示设计的另外一个部分。通常会有一个有意义的抽象，其承担的职责和服务可应用于所有建议的备用实现。该抽象会作为保护设计决定的关键组件。其保持以后对决定进行改变的权利，或者甚至在运行期间能够动态改变。其定义了一个必要的组件客户端所需的服务接口，而无需关心它的实现。可以证明这样的设计点对于随时保证系统的稳定性和健壮性是至关重要的，并且是职责驱动、基于对象和面向对象系统的关键。稍后在会议中，将会检查每个所提名的组件，来看看它们是否提供了这种价值。

示例

在地图系统的设计中，团队会涉及到以下讨论，路线（route）是预先计算，还是动态计算，将如何对它们进行表现，等等。在此处，将引入组件“Routing Strategy（路由策略）”。Routing Strategy 组件具有「根据一个位置列表获取一条路线」的职责。该组件一经创建，关于计算和存储路线的主题便可被推迟（deferred）。事实上，在原型最初，将一个预先选择的路线列表硬编码到 Routing Strategy 组件中。在最初版本中，可以计算飞行的路线。在后续版本中，则结合了动态计算的路线和预先计算位于数据库的路线。Routing Strategy 对象定义了对所有实现都通用的服务，使系统能够不断地发展。

设计模式。

Gamma，Helm，Johnson，Vlissides 编著的设计模式书中包含了许多用来「保护设计决定」的思想。在书中就包含了刚才引用的“strategy（策略）”的思想。熟练掌握该书，并学习什么时候能够从设计模式书中引入一个组件。

同类集合（Homogeneous collections）。

在所有设计级别上，相同类型对象的集合会重复出现。有时候，集合中的项并不是那么令人感兴趣，令人感兴趣的行为会出现在集合中（例如，分类帐行项（ledger lines）是无趣的，分类帐集合则令人感兴趣）。有时候集合能够引导用户界面（如列表框），有时它们会引导数据库（如持久化的集合）。做好在你的组件引入同类集合的准备。在基本业务关系中已经命名了这些许多集合。

当使用一个集合时，需要关注哪些组件会“拥有”它，创建它，初始化它，保持其内容的信息一致（例如，假设会有许多其他组件在各个阶段来浏览或更新它）。

有时候，良好的抽象会在设计之后才被发现，而不是在设计之前。这就是说，只有在与场景和组件工作了一段时间之后，才会发现许多组件可被看作是一般抽象的一个变化，或者他们会对其他抽象进行大量的委托。

如果有一些组件具有通用的职责而未被共用，则考虑引入一个新的组件，然后这些组件将它们的通用职责委托给这个新组件。

如果组件看上去是一个通用主题上的变化，或者职责不能进一步地委托，则需要寻求能够作为通用组件的抽象。

要注意某些通用的情形。

不同级别的委托。

组件可以将其职责委托给不同级别上的组件。将会出现两个卡片具有相同职责的情形。这两个组件之间的差别就是第二个组件将无法被发送请求至第一个组件的组件所完全认知。

职责分配中的难点（地方太多或没有地方）。

看上去职责所属的地方太多：

有的组件已被规定得过于广义。重新考虑每个组件，它所表示的抽象，它在系统中的用途。尽量确定一个组件是否被规定得过于广义。很有可能在它的新定义中，不会再以‘yes’作为问题的答案，该组件的职责是否真的就是处理这个职责呢？

需要命名一个新的组件。

看上去职责具有多个家是因为背后隐藏着一个无名组件需要命名。其一旦显露出来，便可将其作为职责的通用部分。

示例：火车发动机（Train engines）挂着多节车厢和一节守车（caboose）。确定是发动机还是守车具有「连接发动机到守车」的职责则是一个难点。**解决方案：**有一些抽象被遗漏了，其中有关于火车结构（configuration）的，有关于火车的。系统很可能想要管理发动机，车厢和守车的结构，并且想要随时能够变化。引入“train configuration”（火车结构，或运输行业的对等物、准确术语）可能是正确的。将「注册每个发动机，车厢和守车的附属物」的职责赋予火车结构是正确的。

所做出的设计决定必须要简单。有时候必须要简单地做出选择。有些案例不需要进行进一步的考虑，而且对等出现的两个组件能够与任务良好地匹配。设计可能会保留其中的一个决定。可能会在后续的开发中发现哪一个要优先选择。

示例：一个仓库包含有货箱。是仓库知道每个箱子的位置，还是每个箱子知道它自己的位置呢？每一种方式都能够工作，每一个有具有自己的特点，无法明确确定哪一个“更好”。

看上去职责不属于任何地方：

某一组件被规定得过于狭义。重新考虑每个组件，它所表示的抽象，它在系统中的用途。尽量确定一个组件是否被规定得过于狭义。很有可能在它的新定义中，其对问题会做出“yes”的回答，该组件的职责是否真的就是处理这个职责呢？

需要命名一个新的组件。职责看起来不属于任何地方的原因是需要发现一个未被命名的组件。一旦发现，便能够看到其所表示的抽象并对其进行命名。

评估组件和职责

考虑备选假定和场景来检查和改善设计。将附加场景（additional scenarios）作为必要的假定。

检查备选场景，以找出结果，创建和错误条件中的变化。

对于每个场景，预排每个变化，以检查所命名的职责及其分配能否正确工作。对于最前面的几个，将很可能必须命名新的组件。随后，场景将会以一种非常显而易见的方式使用已命名的组件，从而使预排变得简短。

完整性标准

没有必要预排每一个数据条件的每个变化。只需要预排能够揭示有关组件的工作原理的新东西的场景。一旦场景的预排已经深入地进行讨论，协调员（facilitator）便会说，“...我们已经建立的可以工作了”。如果在此建立了能够正确交付的场景，该场景便告结束。如果场景后来是以一种不同的方式继续的，在预排过程中会选取那些出现新东西的点。

考虑最初的场景（initial scenarios）和错误引发的场景（error causing scenarios）是非常重要的。错误条件场景定义了系统本身必须能够防止的错误。初始场景（Initialization scenarios）则揭示了哪些组件能创建哪些其他组件。这两种都需要归档。

示例：在汽车租赁中：客户租借一辆汽车，跨越一个时区并在汽车被出租前返还。在银行中：一个新的客户申请一张 ATM 卡。

检查可能存在的需求变化

考虑系统的演变。客户在使用该系统后可能会提出哪些改进。典型地，最初的需求陈述包含了对第一个版本的简单化的假定，它将会在以后的几个版本中进行修改。预排非常适于讨论某些将来可能的改进或需求变化。可以放置一个新的组件来防止某个特定的需求变化（如前所言，“组件保护设计的变化”）。组件将会使第一个系统能够通过简化的假定进行构建，但是其包含着一个留待改进的空间。

设计团队将必须确定在设计工作范围内需要考虑哪些需求的变化。像“考虑涵盖世界上每个人”这样的需求变化，可能对于一个本地办公系统不是很有意义，但对于一个电话系统就有可能是适合的。

示例：在前面用到的地图系统示例中，需求可能是“从数据库中获取一条位于两个位置之间的路线”。团队需要考虑 (a) 以后的改进将会具有能遍历多个位置的路线（如，“发现一条从 Boston 到 L.A.的路线，中间经由 Chicago 和 Santa Fe”），(b) 以后的改进将使用动态计算的路线。团队引入一个“List of locations（位置列表）”组件以替代「总是有两个位置」的假定，以及一个“Routing Strategy（路由策略）”组件，其隐藏了路线是通过检查还是计算得到。起初，路由算法将只为列表中的两个位置工作，并且总是在数据库中检查它们。当需要进行后续改进时，能够扩展对列表的使用，并且可以添加实现了不同路由策略的备选类。

可能的实现变化。可能的变化包括对结果进行查找还是计算，以及业务中可能的变化。

验证每个组件的名称和职责

“该组件的职责真的就是处理这些请求吗？

直觉能够指导你发现那些与组件所表示的抽象相伴随的职责。运用你的直觉和判断。

“该组件是否已经存在呢？”

反复观察组件和设计模式目录，看看是否能够发现一个能够知道你现在所想的适当组件。

“该组件具有的职责是否过少？”

警惕一个组件仅具有一个令人称赞的职责。组件被假定用来捕获系统中「具有一个意图的抽象」。作为一个有意义的组件，在某一时刻所出现的可能正是该组件本身所保持的一个单独的职责。可以把这个职责分配给一个组件。

此外，职责很少会被描述成组件所扮演的一个“角色（role）”。寻找能够扮演该角色的组件。

寻找其他做类似工作的组件，看看这两个组件中的哪一个能更好地传达系统的抽象，以及看看是否能排除这两个组件中的一个。

“该组件是否具有太多的职责？”

警惕将组件最终作为一个充斥着职责的洗碗槽（kitchen sink）。组件被假定用来捕获系统中「具有一个意图的抽象」。一个具有太多职责的组件很可能不是一个单独的抽象，而是多个，混合在一起。与使用一种单纯方式捕获抽象的简单组件相比，一个复合的组件更加难于重用。

在大型系统中，一个大的子系统允许存在大量的职责。创建子系统的一个更可取的原因可能是其具有一个单独的或少数的关键职责。然而，可能会有创建子系统其他的合理原因，使其成为一个职责包。

“该组件能否保护设计的决定？未来的子类型化（subtyping）？实现的变化？”

许多组件可以像前面描述的那样能够专门保护设计的决定，也可以支持多种变化或实现？

如果一个组件无法做到这列工作之一：它可能是一个业务进行管理的必需项，因而需要保持。它的职责可能符合其他一些地方，允许将其删除。可能会更好地对它进行命名，因此一个更好的抽象将浮出水面。

“名称是否能够正确地反映抽象和能力？”

“在以后的重用期间，当脱离系统的上下文来看待组件时，根据组件的名称来发现和使用组件将会多容易？”

名称极为重要。在理解系统以及查找既有的可用组件时，它们将会被人们所关注。

“你必须能够基于组件的名称做出假定.”（Ward Cunningham）。

“如果你的队友不能对你的代码做出假定，那么你只不过是将干草放在了流沙的上面。”（Hayden Lindsay）。

可用的名称和命名习惯：根据所表示的抽象进行命名，如 bankingTransaction。根据它的能力进行命名，而非它在系统中的角色。

需要避免的名称和命名习惯：

“manager”。考虑使用“broker”，“librarian”，或相似者。示例：职责是「获取正确可用的表（table）」。试着用“Table Librarian”来代替“Table Manager”。

"-er"后缀被应用在一个规则（formula）上。考虑使用“policy”或“strategy”。示例：职责是「获取一个计算利率的规则」。试着用“Rating Policy”或“Rating Strategy”来代替“Rater”。

“data”。尽量找出其所表示的抽象。

“其他哪些组件能够控制该组件的生命周期？”

如何，何时，由谁创建它呢？

如何，何时，由谁销毁和清除它呢？

在划分过程中确保这些问题能够得到回答，否则需要往回走并处理它们。

“当前可用的行为有哪些是经过了仔细的检查，以及这些是如何被处理的？”

示例：信用卡公司：客户可以从一个预期者变成一个普通持卡人（plain cardholder），再到一个优先持卡人（preferred cardholder），或是一个拖欠持卡人（delinquent cardholder），再成为一个前持卡人（former cardholder）。信用卡公司客户所处的这每个“状态”具有相对应的不同行为。设计必须考虑到状态及其伴随的不同能力间的迁移。

行为随时间变化的实体常见于业务系统中。要对它们予以警惕。状态上的变化所伴随的行为上的变化并非能通过当前的技术轻易解决。必须要慎重和仔细地来设计它们。

“职责短语是否采用了主动的术语？”

对任何一个组件发问其是否只是作为信息的一个联系点。这有一些是必要的（**例如**，银行交易可能仅被用来与交易数据结合在一起），但有时候可它们赋予更多的职责。

“如果用魔毯来覆盖一个应用程序，要比用毛毯花费的更少。”（Rebecca Wirfs-Brock）。具有主动职责的组件就像一个魔毯。没有主动职责的组件便如同一个毛毯。

使用如下的单词：

“Obtain...”

“Add...”，“Remove...”，“Introduce...”

“Sort...”

尽量避免那些来自计算机专业术语的单词，如：

“Hold...”

“Manage...”（有时候需要该单词）

“know...”（有单独的地方来存放“联系点”信息）

“该组件是否能够实现？”

沿着级别进行检查，以确保一个组件是可以实现的。根据设计，设计团队能够声明，“如果我们拥有这些组件，我们能够以这种方式来交付所需的功能。”该声明便可作为「组件可用或能够构建」的断言(assertion)。

检查通信的模式

在一个组件周围的通信是否密集？

如果一个组件控制着通信的模式，它可能具有过多的职责以及过于了解系统过多的部分（它是一个“manager”）。这将成为一个系统的脆弱部分：如果任何其所了解的部分发生变化，那么它可能不得不随之变化。如果它是一个大而复杂的组件，那么引入错误或变化扩散的机率会更高。考虑来分散它的职责。

两个组件之间的通信是否密集？

相互集中通信的两个组件显然要非常了解对方。

或许应该把它们组合到一个组件中去。

示例：在 Smalltalk/V 中，模型-视图-控制者（Model-View-Controller）的分离使得 GUI 的视图和控制者之间具有大量的通信。根据通常要求，可以把视图和控制者结合起来产生一个交互者（Interactor）。这个新的抽象被很多人认为更加容易使用。

或许这有待发现第三方的抽象。有时候通信表示了另一个组件的信息和职责，它未被命名并被这两个组件一分为二。

在这里，你应该具有一叠改进后的卡片，并标注上职责，它们携带着所需的更优设计或是不用那么新的设计。

在设计会议最后，最复杂的场景会被设计得让人满意，对设计进行原型化来检查流程以及改进设计评估。

通常，对场景进行实现将揭示出职责的传递（hand-off）并非最优，或者在需要的点上该信息并不可见。找出设计期间这样的情形并加以弥补。把电脑看作设计团队的另外成员，这名成员能够对设计提供反馈。

只实现那些需要的组件，从当前最可用的用户界面（例如，直接的程序控制，脚本窗口（Transcript Window），或等价物）开始进行。

预排「系统的执行过程」，来检查所有信息和职责的流程是否正确。

使用用来进行验证的实现，或甚至是创建场景的文档。实现不能具有歧义，而且要确保能够展示从一个到另一个组件的所有信息。有些工具支持从执行路径（execution trace）中创建交互图。

归档设计原理和关键场景

保存每个场景的主要场景。保存每个场景的错误场景。保存任何其他的具有令人感兴趣的原理的场景，或那些展示组件间不明显的通信的场景。这关系到对设计团队部分的判断以及项目文档需求的问题（可追溯性）。过多的与其价值不相称的文档会让人难以承担。而文档过少则会引起混乱。

归档每个所选的场景。在声明的级别上，使用组件来创建一个交互或对象示例图。为每个场景和交互图取一个名称。如果多个场景具有一组公用的交互，则考虑产生一个具有交互图的子场景（sub-scenario），来描述这个公用部分。然后会在交互图的子场景中涉及到这些公用的交互。子场景所扮演的是一个位于更加细节级别上的场景，这在前面已经提到过。使用子场景能够大量缩短文档，而不会丢失细节。

交互图在展示场景时必须从它起始一直到场景结束。该图的意图是要声明这些组件如何协作来交付需要的功能（即使这个功能仅仅是返回一个错误的条件）。因此，该图必须从发起场景的最初消息开始，在场景的最终转变处终止。

描述了为什么要把职责划分成它们所固有的方式，如果存在一个业务或不那么明显的原因。职责的划分可能反映了某些基本的业务过程或假定。这种信息很容易丢失，并且有助于回到后续的工作中。将其书写在交互图的顶部，并最终作为程序代码中一个注释。

示例：“计算位于三种不同地域的住房保险是以第一种住房为基础。保险的利率以第一种住房作为基础，并根据第二种住房的利率进行修改。把「计算基本利率」的职责赋予第一种住房。把「修改利率」的职责赋予第二种住房，其将返回「前面住房与其本身相加的新利率」。”

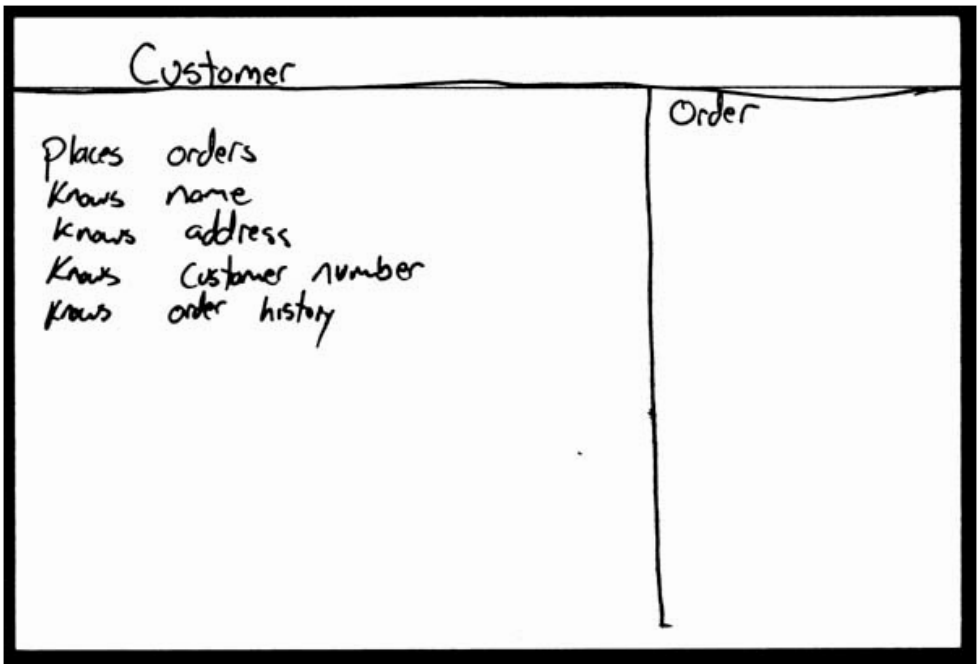
有时候职责分配的工作量会非常大。分配背后的思想不应该被丢失，因为对于其他人它可能并不那么显而易见。

示例：“routing strategy 组件之所以存在是为了能够在存储和计算路线之间自由选择。它具有「获取一条路线」的职责，然而这条路线可能是在存储中，也可能是通过计算得出的。”

有时候组件的名称会使得职责的分配显而易见。这些则不需要进行归档。

演练 CRC 卡

CRC 卡是一个非正式的工作空间，其往往被用来捕获快速变化的信息。当前可以使用各种不同的尺寸（通常尺寸为 3"x5"和 4"x6"，每一种都有相应的支持者）。卡片描述了组件实例所使用的信息。在基于职责建模的活动中，可以将卡片看作一个实例。创建一个把类描述成对象本身的卡片应该是愈发显得必要，将名称标在卡片上，并以单词“class”作为结束。（例如 Checking Accounts class）。



横跨整个顶部写上组件的名称。职责被写在三个分区中。

首先是关于组件职责的一个简要概括，或大纲，其在系统中的角色。这应该是一个短语，或者可能是两个。

其次，沿卡片左侧的列表中是主动职责，有一条向右的线或箭头，其结束于该职责所需的一个协作者的名称。主动职责始于一个主动动词，像“track”，“compute”或“find”。尽可能避免单词“manage”，以及被动动词，如“hold”。

最后是联系点职责，即组件所传递的信息。这些将经常来自于数据或业务模型中的属性。如果对「一个服务属于主动职责还是联系点职责」尚有疑问，可以任意选择，同时稍微倾向于把它放在联系点区。这的确没有太大关系。最后，将平等地来对待所有的职责。具有这些区是为了把注意力放在概括说明和主动职责上，它们是对系统进行划分的主要工具。联系点职责为组件规约（component specification）所需，并且展示了组件如何交付一个已归档场景中的所需功能。

安排活动

使用一张中间能够看到和触及到的桌子。

当前有一叠空白的 CRC 卡可用。把它们放在桌子上任何人都能触及到的地方，或者是每个人发一套。

理想情况下为 2-4 人。如果有一个讨论伙伴，CRC 工作地会更好。4 个人以上会导致人们无事可做以及无聊地坐着，或者讨论的方向转移。

通过发明和识别组件和职责，对一个场景进行预排和角色扮演。从陈述「发起场景的消息和信息以及预期结果」开始。假装设计参与者是单独的组件，并且必须以场景描述的方式来交付功能。人们对组件进行角色扮演。

当感觉到为难时。假装是一个组件而且不知道这个组件应该做什么，开始这会让人感到惊奇。尽管感到惊奇，通过假装成一个组件，能够让人最好地专注于一个职责的正确与否。

询问应该要由哪种组件来处理入口（entry）。选取那个组件的卡片。对职责和组件进行假设，如果需要可以发明新的组件。

注意： 这是你进行重用的大好机会。若有可能，请使用已经存在的东西。

识别组件会需要什么来完成它的本职工作。观察既有组件的目录，寻找已能做这项工作的组件。如果没有这样的组件，则继续询问哪种组件应该具有所需的职责。

使用基于职责建模的方法，用这种方式继续下去，直至场景结束。

技巧：当事物快速变化时，有时候没有时间来写下职责名称或组件名称。在这些时候，可指向一个现有的卡片，或者甚至是桌子上的一个空白地，或者命名这个组件，或者只是说“这个”。如果设计能够工作，组件将会被一致地予以展现，并且能够为它和它的职责找出一个良好的名称。良好的名称非常重要，以至于值得推迟对组件进行命名，直到完全弄清楚它的意图并达成一致。

技巧：注意讨论的级别。正如前面所提到的，最好是要比定期的设计会议的级别更加深入。在其他级别上提名的组件可能会变成最终所设计的子系统的元素。记下属于不同级别的组件，以及或者根据调用这些组件的卡片对它们进行分堆，或者将它们放到一边，从而在有疑问的时候能够转向或重新引入它们。

技巧：只使用一支笔。这样，就不会出现一个人在团队其他人没有注意的情况下改变名称，职责或协作者。可以把这支笔作为一种团队的同步机制。

角色扮演是一种创造性的行动，在其中可以发现设计点，提名组件和职责，以及做出设计决定。在两个名称之间，在是否需要一个组件之间，在分配一个职责的两个地方之间，可以通过对这两种选项与一组场景进行比较做出选择。优先选择最符合场景的选项。可以基于以下几点进行选择：

(1) 以一种更自然的方式来分配职责，

(2) 组件之间的通信模式更加简单，

(3) 针对各种假定的变化处更少。

(4) 有时无法分清两个选项之间的不同。这种情况下，只需选取其中一个并继续。（参见下面的“普通情形”）

因此做出一个深思熟虑的决定通常是恰当的，必要时可以中断一个场景并研究某些变化。这些变化可能尝试关于将来需求或实现的备选假定，或者是它们的用法。

管理卡片

典型地，直接命名一个卡片，但这并非十分必要。

有时候人们想要提名一个组件。很好。把卡片放到桌子上。问它的职责是什么。如果看上去有一个职责，写下它，可以作为一个主动职责，也可以作为一个联系点。使该卡片凭借其自身的价值保留下来。

有时候人们能够说出那里需要一个卡片，但是却不知道它的名称或确切的功能。很好。放在那里一个空白的卡片。随着时间不断增加它的特性直至其名称和职责变得清晰。通常职责将会先变得清晰起来，然后根据职责将会产生名称。

有时候卡片已被命名，但它的职责会演化地不再与名称相匹配。在旧名称上划一道直线并写上新名称。或者，取一个新的卡片，在上面书写新的名称和职责。将旧的卡片放到一边。

去掉关于边界之外子系统的卡片。

讨论经常会走到一个不同的设计级别上。生成的卡片无法应用于当前的设计级别，但却有助于演示一个设计选择的推理过程，或者有助于展示「一个组件将会如何来实现它的职责」。

把那些有助于实现一个职责的卡片集中起来，而不是让卡片把桌子和讨论搞得凌乱。把它们放到组件的后面，这样有助于在一个正确的级别上进行讨论。然后当需要它们的时候，可以把它们拿出来考查，否则不予考虑。

通用组件（generic components）和变量同样如此。有时候，讨论将会集中在通用组件的周围。变量将处于第二重要的位置，但是要在确定它们存在的时候能够出现。将变量放在通用组件的下面，从而通用组件能够运载这种约定（conversation）。当需要的时候，可以将这些变量拿出来。

收集和排列组件的方式：

根据实现级别。这是目前为止所一直在讨论的。

根据私有性（privacy）。如果组件在这个讨论级别上可被公开知晓，或者它们以某种方式保持私有，那么便以不同的方式排列组件。那些私有的组件可能会处于不同的级别。

根据生命期（lifetime）。寻找那些比其它组件的存活时间显著较短或较长的组件。考虑他们是否属于同一级别。有时候，根据生命期来收集组件能够揭示出关于系统、组件或它们间通信的一些令人感兴趣的东西。

去掉那些没用的卡片。

在许多提名的卡片中，有些将不能在整个设计会议中保留下来。如果一个或多个卡片不能看到动作（action），它们会被置之一旁或者返回到工作区中。如果不断显现出它们的重要性，便可将它们拿回来发挥作用。

在会议结束时，如果有一些卡片虽被提名但未被使用，则需要重新将它们提出来以备复审。应该清楚，它们未能保持任何的职责，因而将无法予以实现。如果未能就此达成一致，想保留它们的人必须要找出一个它们能够传达职责的场景。

交互图与卡片一起使用，或甚至用交互图代替它们。

一个对交互图合理分组的设计可以让交互图替代 CRC 卡，来传达所进行的讨论。这是一个个人喜好的问题，因为有的人需要看到消息流，来对交互进行可视化。交互图的时间视图形式与拓扑视图相比，更有利于传达所初步进行的讨论（exploratory discussion）。在一个活跃的设计会议上，CRC 卡仍然能够提供更大的弹性和灵活性。

如果使用交互图来替代卡片，需要在图上书写每个组件关键的职责。

在设计复审时，交互图已经作为设计会议的成果能够得到。可以使用交互图来展示「如何一直传递和调用职责」。必须要在复审期间得到组件的职责清单，可以是一个列表，也可以在 CRC 卡上，或者是交互图上的标注。

在这里，你应该有一叠具备职责的卡片，和一叠展示组件是如何交付场景的交互图。

根据级别整合组件

标识那些与设计会议开始时所声明的设计级别相适合的组件。

系统的设计将在不同级别上向读者展示。为了简化对设计的理解工作，以及隔离未来的变化，应该在一致的级别上对系统的设计进行归档。将会分别收集在一个所声明的级别上代表组件职责的子系统或组件，并且它们不会在所声明级别的系统讨论中使用。

对于一个业务应用程序（business application）而言，第一级别应该只包括那些对业务人员有意义的组件，这些组件致力于「验证系统的工作方式是否与业务的工作方式一致」。每个场景必须要有完整的描述，以及仅使用该级别的组件便能充分进行关联的预排。相关的预排是指，在预排中使用所选的组件显然能够实现所有的职责和需要的信息。

在任何后面的级别上，一个子系统的场景必须要有完整的描述，以及仅使用适合于所声明级别的组件便能充分进行关联的预排。这取决于设计团队的如下评估，哪些组件与该级别相适合，以及哪些组件属于所声明级别的一个组件的实现。

设计团队的直觉、业务知识和常识能够指导他们确定哪些组件是属于一个级别的。

分别收集处于更深级别上的组件。

使卡片能为以后所用。

位于设计边界之外的组件依然有用。或许，设计团队中的一个成员将会参与到使用这些更深入的组件的子系统设计中，以及将会能够使用这些卡片来开始设计。

没有必要对「使用那些更深级别上的组件」进行归档。房间中的一些人可能想对它们的使用进行归档，为将来的设计或将来的参考提供帮助。

征稿

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

UMLChina 公开课

“UML 应用实作细节”

（2003 年 12 月 6-7 日，北京）

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

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

本训练课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，结合大量练习和项目实践，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。

详情请见>>

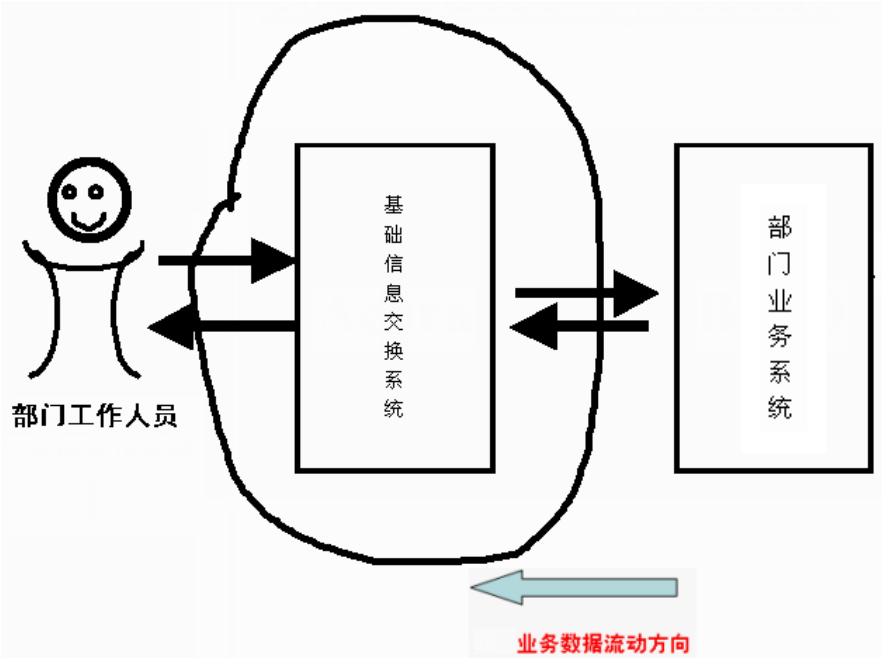
用例随笔

think 著

查看评论

某城市的基础信息交换系统，系统的目标是通过工商部门与国税、地税、质监部门的企业数据之间的比对，发现数据差异，减少漏管户，防止偷逃税。

首先碰到的是一个用例命名的问题。每隔一段时间，每个部门的工作人员要使用基础信息交换系统做某件事情，使得本部门业务系统中的数据进入基础信息交换系统中。很明显，这是一个基础信息交换系统的系统用例。问题出现了：这个用例叫什么？绝大多数开发人员都认为叫“采集数据”，原因如下：

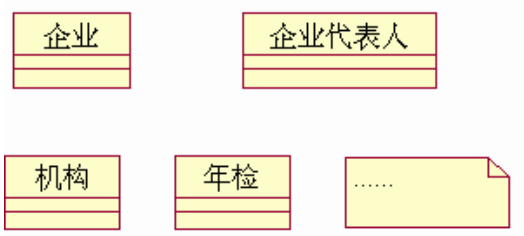


部门工作人员给基础信息交换系统发出命令后，基础信息交换系统就连通部门业务系统，把业务数据搬到基础信息交换系统内。确实是一个命令系统去“采集数据”的过程。但是我们要回过头来想一想：用例是“执行者用系统来干什么”。部门工作人员会怎么看待这个过程？他们会说，这是在“提供数据”，把数据提供出去，作为比对、交换的依据。他们之所以要和基础信息交换系统交互，是为了“提供数据”。这一点，通过对涉众利益的研

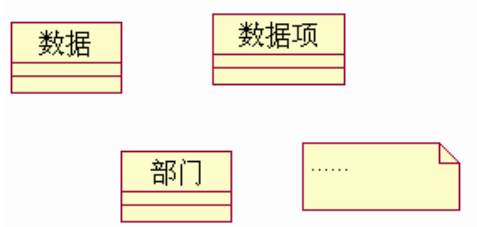
究也可以验证：因为现实中部门的数据是可以用来盈利的（一个企业的法定代表人会接到莫名其妙的推销电话，还直呼其姓“×总”，就是这个原因），特别是工商部门，数据是最全的，白白地送出去给其他部门用，心有不甘，所以部门希望能控制“提供数据”这个行为。如果是“采集数据”，应该是多多益善，不会有什么不甘心。

那么硬是要叫它“采集数据”有什么要紧吗？其实也不要紧，不管是叫它“采集数据”还是“用例 A”，掌握了用例技术的团队知道用例内的路径、步骤、约束都必须受到涉众利益的制约，不会有严重的偏差——不过这样的团队就不太会起这样的名字，因为已经感觉味道不对了。

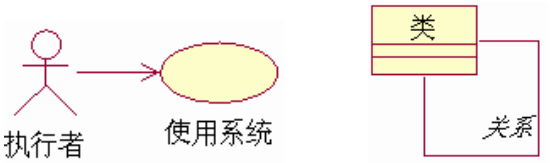
说到味道，我们可以继续往下走，在识别对象时也会碰到。用例文档出来后，下一步就是从文档中抽取出一些分析类。一开始抽取的类是这样一些：



这个时候，也可以感觉到味道不对。用例文档中确实有上图中的这些词汇（出现在字段列表中），但我们要做的不是这些部门的业务系统，我们的系统只是负责帮部门比对数据。“企业”、“年检”……等不是我们关注的核心概念。“数据”、“部门”、“提供”等才是适当的抽象：



这里只是说“适当”。不这样作抽象，也能做出一个可运行的系统来。往极端里走，任何系统的用例和类都可以表示成这样：



如何才是适当，依赖于对业务的理解和对用例技术、对象技术的掌握。

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

更多的操作系统访问控制模式

Eduardo B. Fernandez、John C. Sinibaldi 著，droplet 译

查看评论

摘要

这篇文章描述了操作系统访问控制架构性的模式。这是对上一篇文章[Fer02]中所描述模式的一个补充。这些模式控制主体对资源的访问（资源被表示为对象），其中包括认证模式、进程创建模式、对象创建模式和对象访问模式。

简介

我们这里描述操作系统中的访问控制模式。这些模式是对[Fer02]中所介绍模式的补充。在那篇文章中描述的模式如下：

- **文件访问控制模式**。怎样在操作系统中控制对文件的访问？将授权模式应用到主体对文件的访问过程中。被保护的对象现在是一个文件构件，它可以是一个目录，也可以是一个文件。

- **受控的虚地址空间模式**。怎样控制进程根据一个预定义的访问类型集合去访问它的特定虚地址空间？将虚地址空间划分成与程序逻辑块相对应的段。用一个特殊的描述符描述对这些段的访问权限。

- **引用监视器模式。**当一个进程访问对象时怎样强制它必须通过授权？定义一个抽象的进程，它截获所有对资源的访问，并检查它们是否符合授权要求。

- **受控的执行环境模式。**如何为进程定义一个执行环境？在每个进程上附加一组描述符表示权限，并用引用监视器强制进行访问控制。

在这篇文章中，我们增加以下模式：

- **认证器模式。**怎样确认主体的身份？使用单个访问点接受主体与系统的交互并用相应的协议确认主体的身份。

● **受控进程生成器模式。**怎样定义新创建进程的权限？在进程创建过程中定义它。

● **受控对象工厂模式。**进程创建新的对象时如何确定它的权限？当进程请求一个工厂去创建新对象，这个请求里包含了新对象的特性。在这些特性中包含了对象的访问权限。

● **受控对象监视器模式。**怎样控制主体对对象的访问。用一个引用监视器可以截获所有进程的访问请求。引用监视器检查进程是否有访问对象的权限。

假设资源被表示成对象（这在现代操作系统中非常普遍）。图 1 展示了这些模式所组成的模式语言。比如，认证在文件访问和受控对象访问中是必须的，当一个主体被授权通过某种方式访问某些对象，我们必须确认请求者不是入侵者。其他三个模式完成了受控执行环境的定义，现在对象的创建和访问都是受控的。这个语言同时也表明文件访问被引用监视器所控制。

背景

操作系统是计算机中提供安全保护的一个基础设施。操作系统支持应用程序的执行，所以每一层上的安全限制都应在操作系统中得到加强。操作系统同时要保证自身的安全，因为任何的妥协都会使它的用户帐号或是文件里的数据被非法访问。一个脆弱的操作系统使得黑客不但能够访问操作系统文件里的数据，而且可以访问那些使用操作系统服务的数据库系统里的数据。操作系统通过使进程相互独立来保护进程和与进程相关的文件里的数据[Sil03]。为达到这个目的，操作系统控制对诸如内存地址空间、I/O 设备等资源的访问。许多操作系统采用访问矩阵作为安全模型。一个访问矩阵定义进程（一般成为主体）对特定资源（资源在现代操作系统中常被表示为对象）可以进行何种类型的访问。使用这个模型时，我们必须确定主体在访问资源前是否通过了认证（使用 Authenticator），我们也需要在创建进程时确定进程所具有的访问权限（使用 Controlled-Process Creator），同时需要将进程放在一个受控的执行环境中执行，在这个环境中进程不能做超越它权限的动作（Controlled Execution Environment）。我们也需要定义新建对象的访问权限（Controlled Object Factory），同时控制在执行过程中对对象的访问（Controlled Object Monitor）。后者通过截获访问请求并检查它们的授权信息来达到访问控制的目的。所有这些功能就是在这两篇文档中所介绍的模式所要完成的。

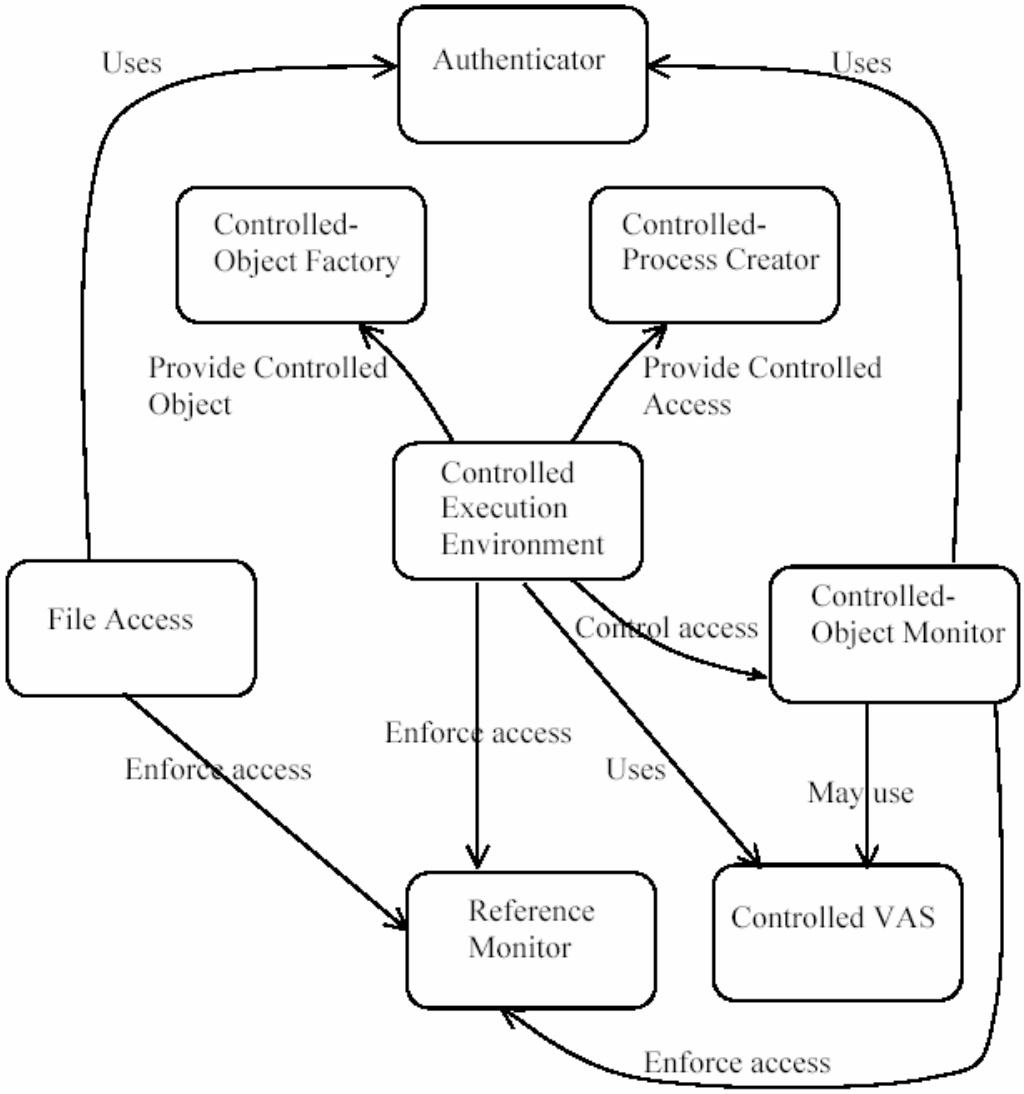


Figure 1. O.S. Access Control Pattern Language

图 1 OS 访问控制模式语言

操作系统在用户第一次登录时认证用户（可能在他访问特定资源时还要求再次认证）。然后用户执行一个程序，这个程序可以由多个并发的进程组成。进程一般通过操作系统提供的系统调用来创建。一个进程请求操作系统创建子进程的同时赋予这个子进程访问某些资源的权限。应用程序在运行时可能会创建一些对象。有些对象在程序初始化时创建，有些在执行过程中动态创建。进程对对象的访问权限必须在进程创建时确定。应用程序也可能需要诸如 I/O 设备这样的资源或者其他一些从资源池中得到的资源。在这些资源创建的时候，相应的权限就应该分配给相应的应用程序。这些权限被定义成授权规则或策略，并在进程访问资源时被检查。这就意味着我们需要截获每个访问请求，这可以由引用监视器来完成。

认证器（Authenticator）

意图

怎样确定用户的身份？

上下文

当主体，通常是一个用户，发起请求时，操作系统创建一个受控的会话。然后，认证用户（表现为用户执行的程序）就可以根据他的权限去访问资源。访问敏感资源时可能需要再一次认证。在分布式系统中如果访问外部节点的资源，需要再一次认证。

问题

怎样才能防止非法用户访问我们的系统？一个恶意的攻击者伪装成一个合法用户去访问资源。如果被伪装者具有比较高的权限，事情就非常严重。

约束

这个模式有以下的约束条件：

- 不同的用户要求不同的认证方式。我们必须能够处理所有的认证方式，否则就会造成系统的安全隐患。
- 我们需要以一种可靠的方式来认证用户。这就意味着我们需要可靠的协议，同时能够保护认证的结果。否则，用户可能跳过认证步骤，或者修改认证结果，使得系统暴露在安全攻击之下。
- 在安全和费用之间需要权衡，越安全的系统，花费也就越多。
- 如果认证是一个频繁进行的动作，性能需要注意。

解决方案

使用单个访问点来接受主体对系统的访问，同时使用协议来识别主体的标识。这个协议可能使用用户的输入作为标识，也有可能是其他更复杂的方法。图 2 描述的是这个模式的类图。主体，通常是用户，请求对系统资源的访问，认证器接收这个请求并根据认证协议认证用户。如果认证成功，认证器创建一个可信标识（这有可能是一个令牌，或是其他的东西）。

动态图

图 3 描述了认证的动态过程。用户请求到认证器。认证器通过认证协议确认用户信息，同时创建一个可信标识。用户取得一个可信标识的句柄。

变体

Single Sign-On

Single Sign-On（SSO）是一个过程，在这个过程中主体的安全标识可以在多个安全域中使用，并且可以多次使用。认证的结果就是用户取得一个安全令牌，在后续的访问中这个安全令牌就代表了认证用户。

PKI Authenticator

公钥加密是一种常用的标识用户的方法。它的认证过程与图 2（图 4）所示的模式过程类似。一个认证类使用包含公钥签名的证书进行认证。认证的结果是用户取得一个可以在后续访问中使用的认证令牌（这同样是 SSO 的一个变体）。

已知应用

- 许多商业操作系统使用密码来认证用户。
- RADIUS 使用集中认证服务来认证网络用户或分布式系统的用户。
- SSL 认证协议使用 PKI 来认证。
- SAML，一个安全 WEB 服务，定义了它在实现 SSO 体系时的应用[sam]。

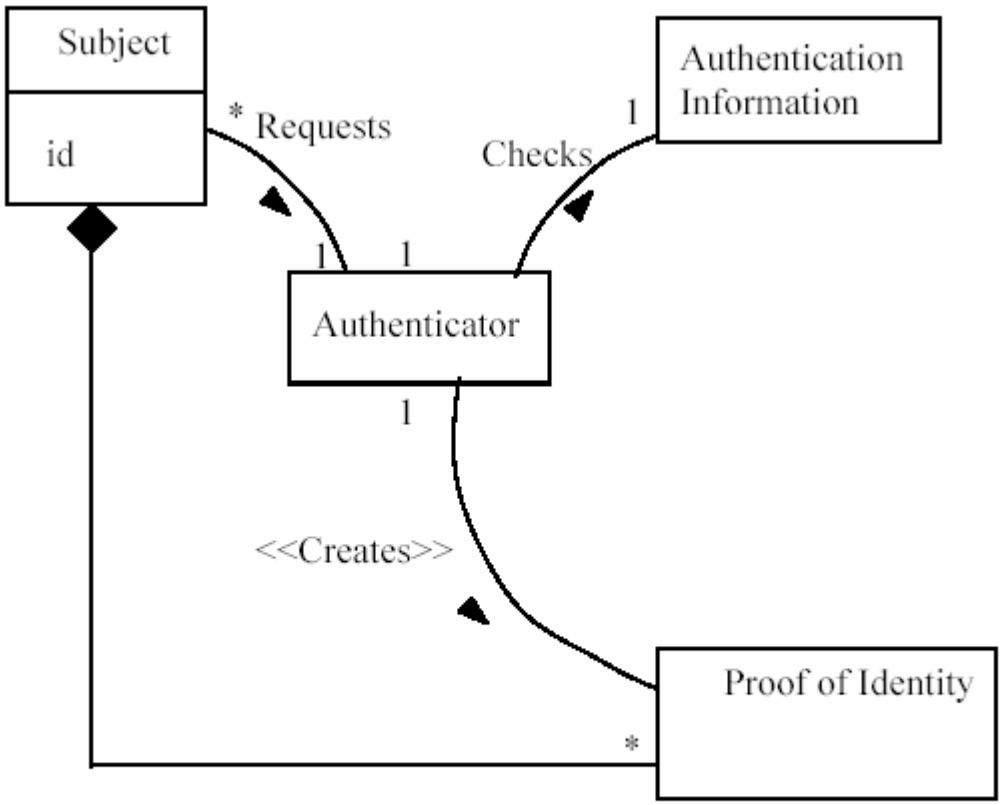


Figure 2 Authentication Pattern

图 2 认证器模式

结论

这个模式有以下几个优点：

- 根据不同的协议和不同的认证信息，我们可以以不同的方式认证不同的用户。
- 因为认证信息是分离的，我们可以把它存储在一个保护的区域内，所有的主体对它最多有读权限。
- 我们可以使用不同强度的算法和协议。这要在安全和花费之间权衡得失。有三个因素需要考虑：用户知道的信息（用户密码），用户有的信息（用户的 ID），用户是什么（用户的身份）。

- 认证可以是集中式的，也可以是在分布环境中。
- 我们可以生成一个安全的用户 ID，这个 ID 可以用在后续的认证中。这样会提高性能。

（可能）有以下几个缺点：

- 认证过程需要耗费一些时间。
- 系统的复杂性和花费随着不同的安全级别将会增加。

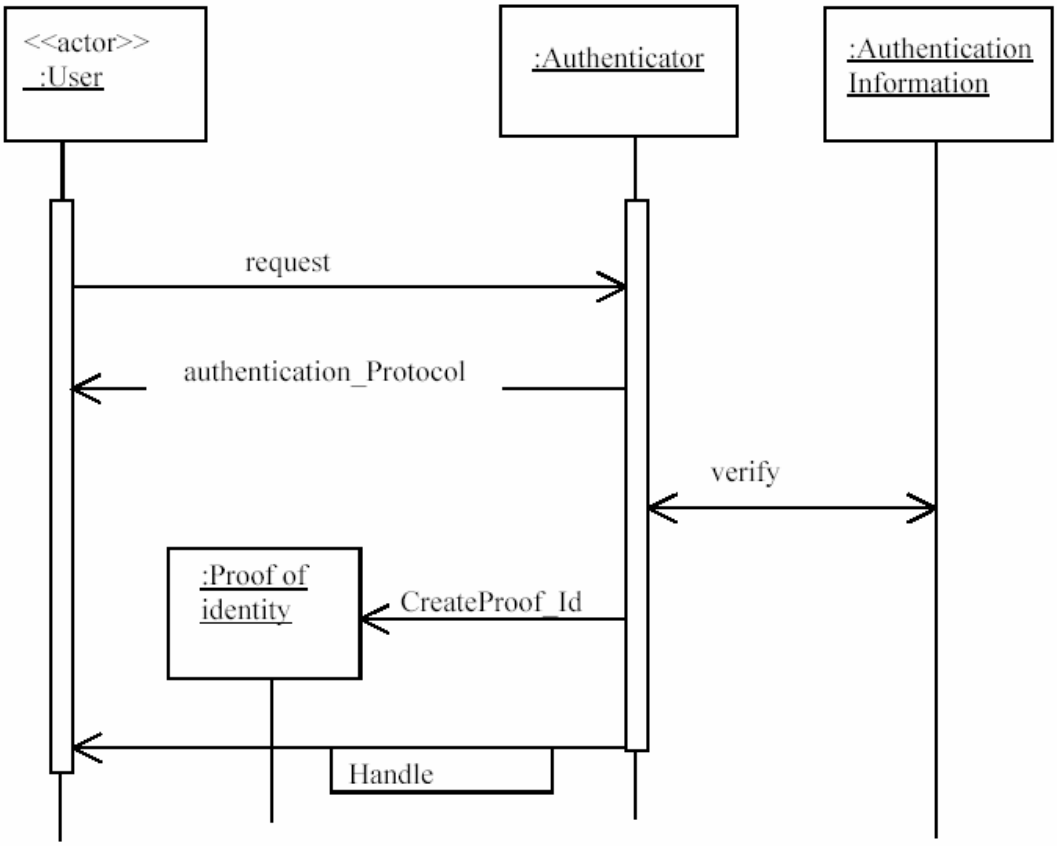


Figure 3 Authentication Dynamics

图 3 认证器的时序图

相关模式

分布式认证器[Bro99]讨论了认证器在分布式环境中的实现。

分布式过滤和权限控制框架包括了认证器[Hay]。

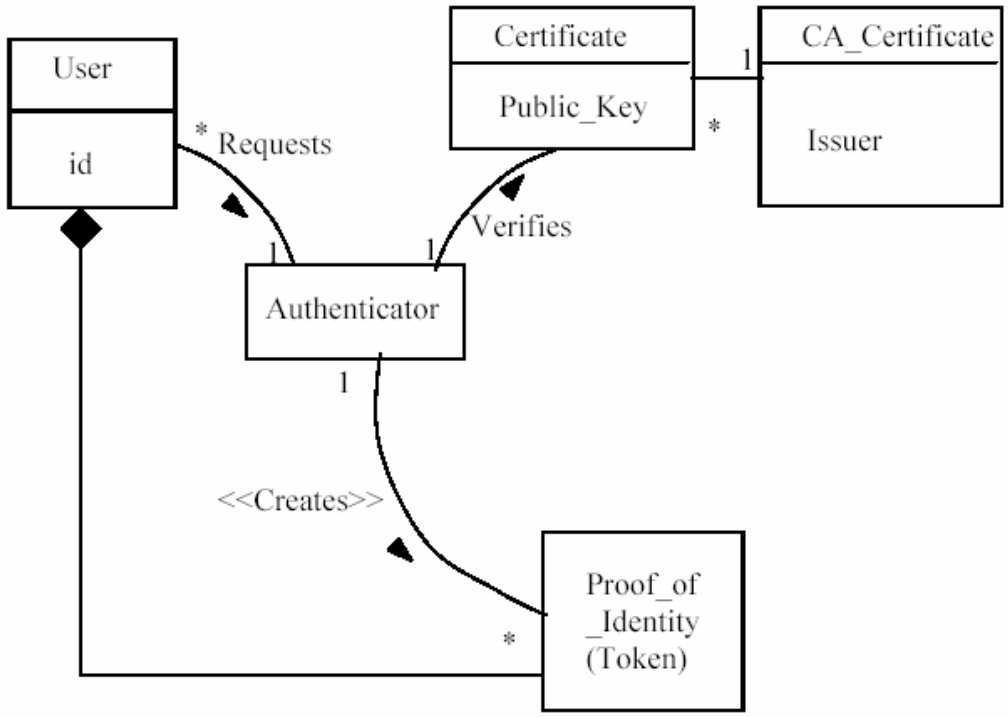


Figure 4. Class model for PKI authentication

图 4 PKI 认证器的类图

可控进程生成器

意图

定义合适的权限并将它赋给新创建的进程。

上下文

操作系统中的进程和线程需要根据应用的需求而创建。

问题

计算机系统中有许多进程或线程。进程根据应用程序的需求而创建，并且操作系统也是由许多进程组成。如果进程不受监控，那么它们可以相互交互，也可以非法访问对方的数据。

所以它们对资源的访问权限必须根据正确的策略来定义，如仅知所需策略。许多情况下进程需要进行的动作可能超越它的权限，比如访问那些它没有权限访问的文件。

约束

这个模式有以下的约束：

- 必须有方便的方式选择策略去定义进程权限。没有根据策略定义权限可能导致相互矛盾的情况出现和没有系统化的访问控制，这样可能会使得权限失去作用。
- 子进程在某些情况下可以伪装成父进程，但是这样伪装必须严格控制，否则一个脆弱的子进程可能泄漏信息或者破坏数据。
- 进程创建子进程的数量应该受限，否则可能引起拒绝服务攻击。

解决方案

由于新进程是通过系统调用是或者发送给操作系统的消息而创建的，我们就有机会决定新进程的权限参数。典型的，操作系统象创建子进程一样创建新进程。有几种策略确定赋给子进程的权限。一种策略是子进程可以继承所有或部分父进程的权限。另一种策略是父进程将自己权限的一个子集赋给它的子进程（更安全的一种方式）。

图 5 展示了这个模式类图。受控进程生成器控制操作系统新进程的创建。创建请求包含了父进程定义的子进程的权限。这些权限是父进程权限的一个子集。

动态图

图 6 展示了进程创建的动态过程。一个进程请求创建新进程。访问权限被包含在创建请求中传递给操作系统。

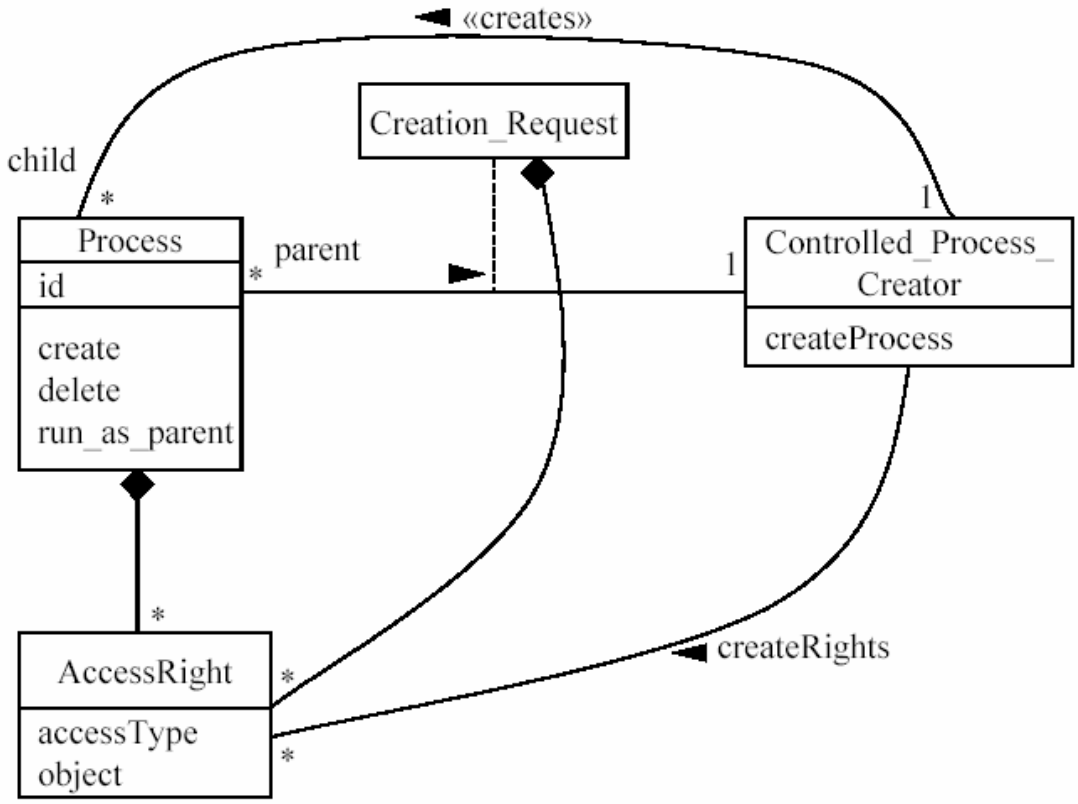


Figure 5. Class diagram of Controlled-Process Creator

图 5 受控进程生成器类图

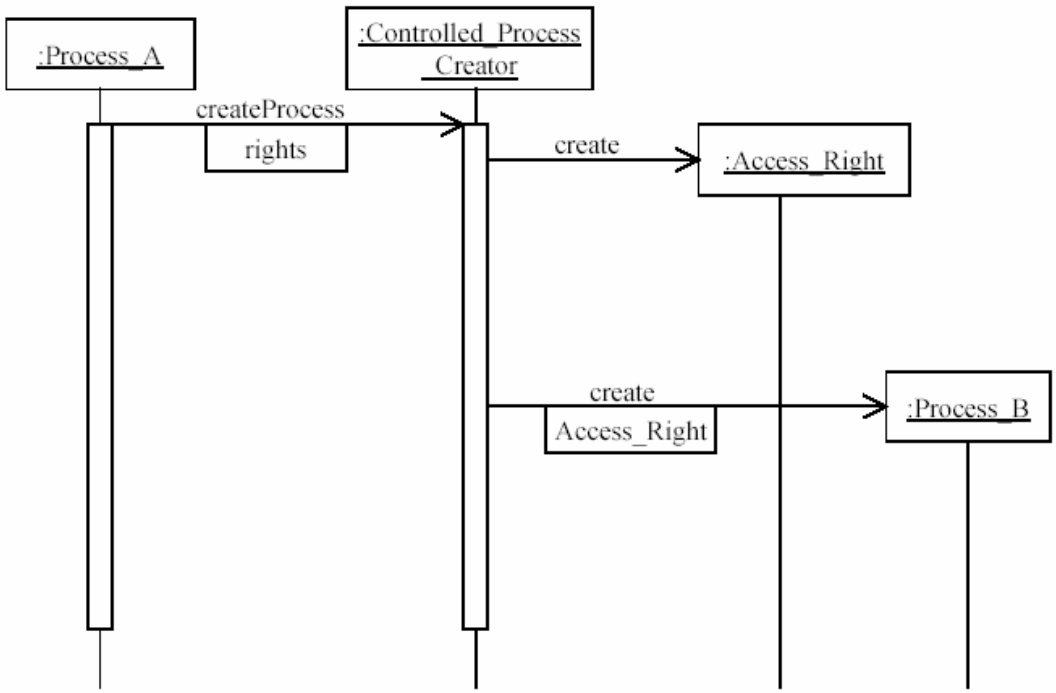


Figure 6. Child creation dynamics

图 6 子进程创建的时序图

已知应用

在许多操作系统中，例如，UNIX，权限是继承自父进程权限的一个子集。一些强化的系统，如Hewlett Packard's Virtual Vault不允许继承，它为每一个子进程重新定义权限集。

结论

此模式有以下的优点：

- 进程可以根据不同的权限策略创建。
- 创建子进程的数目可以控制。这可以防止拒绝服务攻击。
- 权限集可能包括父进程的进程ID，这样可以允许子进程以父进程的权限集运行。

受控对象创建器模式

意图

对象为特定的目的而创建，如果允许其他进程访问它，就必须在它创建时指定这些权限。

上下文

计算系统需要根据新创建对象的重要程度来控制对它们的访问。

问题

在计算环境中，进程会在运行过程中创建对象。有些对象在程序初始化时创建，而其他对象在进程运行时动态创建。进程对这些对象的访问权限需要在进程创建时就指定，否则进程可能会误用它们。

约束

- 应用程序创建不同类型的对象，但是我们需要根据它们的权限以一致的方式去访问它们。当然，使用标准的安全策略相对比较困难。
- 我们允许从资源池中创建对象，它们的权限可动态指定，这里的权限当然不是固定不变的。
- 应该有特定的策略来定义谁可以访问新对象，当我们定义对象的访问权限时就可以使用它。这是一个基本的安全要求。

方案

当进程通过工厂创建一个新对象时，创建请求包括了对象各种参数。在这些参数中，有定义主体对创建对象的访问权限。

动态图

图9显示了对象创建的动态过程。

结论

这个模式有以下的优点：

- 可以根据对象的重要程度定义对它们不同的访问权限。
- 从资源池中创建的对象可以动态添加其权限。
- 操作系统可以使用所有者权限，例如，对象的创建者可以拥有对所有它创建对象的访问权限。

已知应用

WIN32 API允许进程调用不同的系统调用创建对象，在调用时进程会传递一个包含权限信息的结构引用。对象创建时，内核就把相应的权限与它绑定。内核返回给调用者一个句柄用于访问对象。其他操作系统使用预定的权限；比如在UNIX操作系统中，与所有者同组的其他用户拥有对文件相同的组权限。

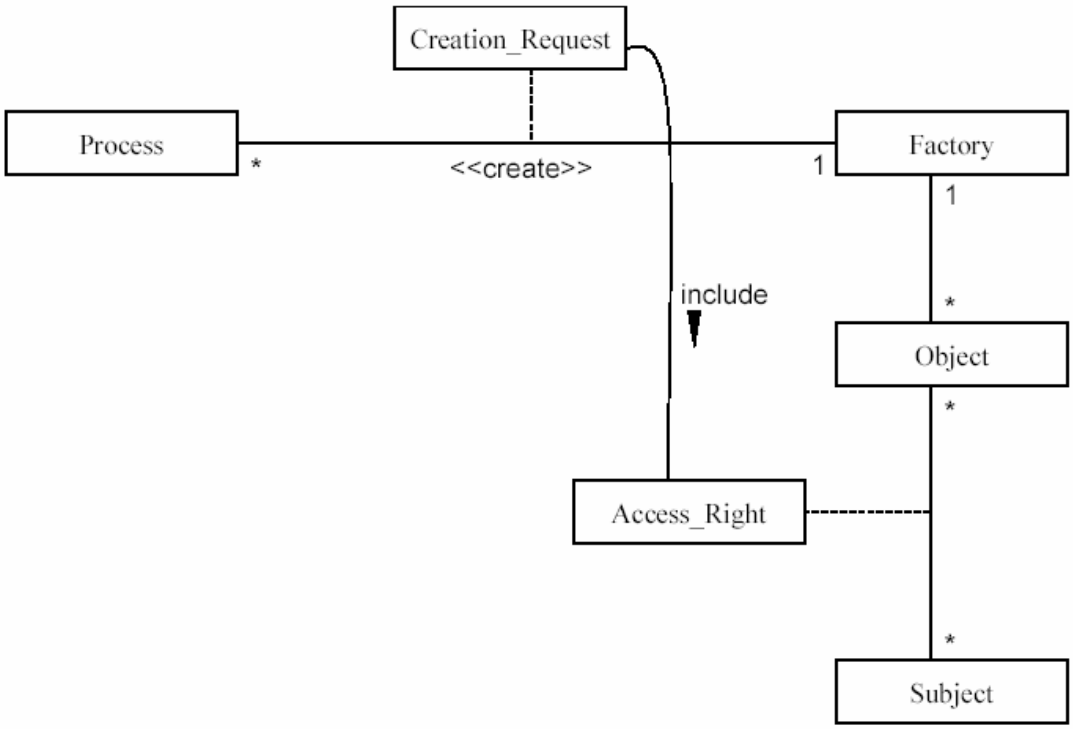


图8 受控对象创建器模式类图

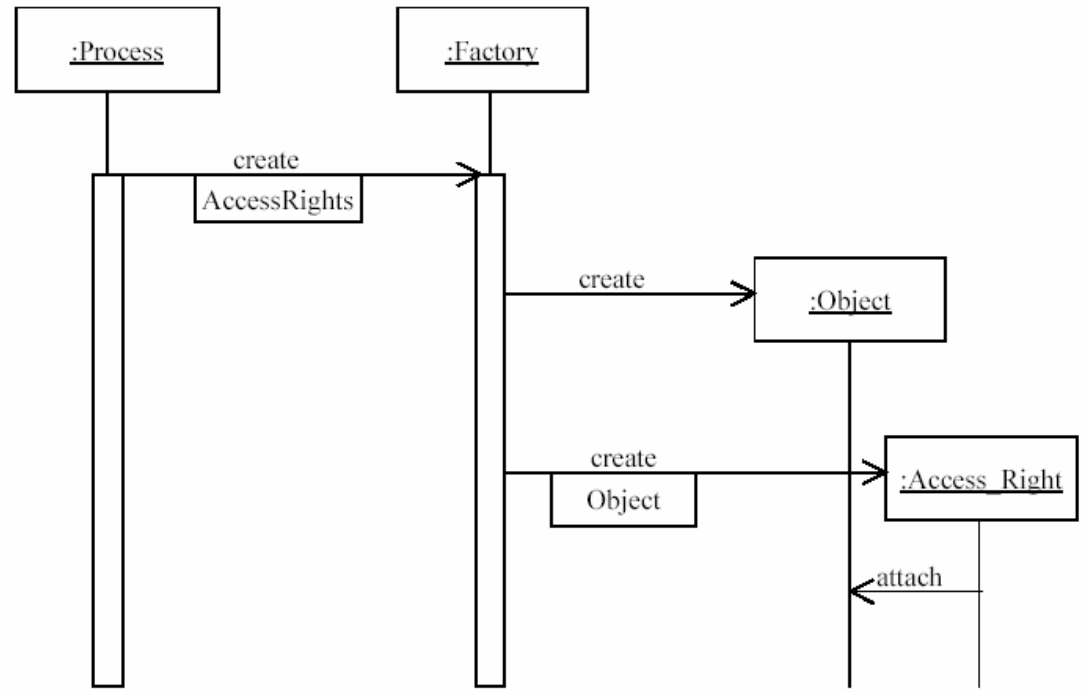


图9 对象创建时序图

受控对象监视器模式

意图

如何控制主体对对象的访问。

上下文

操作系统包含不同权限的对象。

问题

当对象创建时，我们定义进程对它们的访问权限。当进程访问它们时需要强制使用授权规则或是策略。

约束

有以下的约束条件：

- 不同的授权策略定义了不同的访问限制；当进程访问对象时我们必须强制使用这些限制。

- 我们需要控制不同类型的访问，否则对象可能会被误用。

解决方案

使用引用监视器截获进程的访问请求。引用监视器检查进程访问请求的权限是否符合访问规则。

图10显示了这个模式的类图。更具体的引用监视器模式的实现参见[Fer02]。那里的改进展示了系统如何将安全对象和规则联系在一起。

动态图

图11显示了安全主体访问安全对象的动态过程。访问请求发送给引用监视器，由引用监视器检查访问规则。如果规则允许，内核处理请求并向主体返回结果。这里需要注意的是，返回结果中包含了对象的句柄或令牌，对安全对象的后续访问可以直接进行而不需要再检查权限。

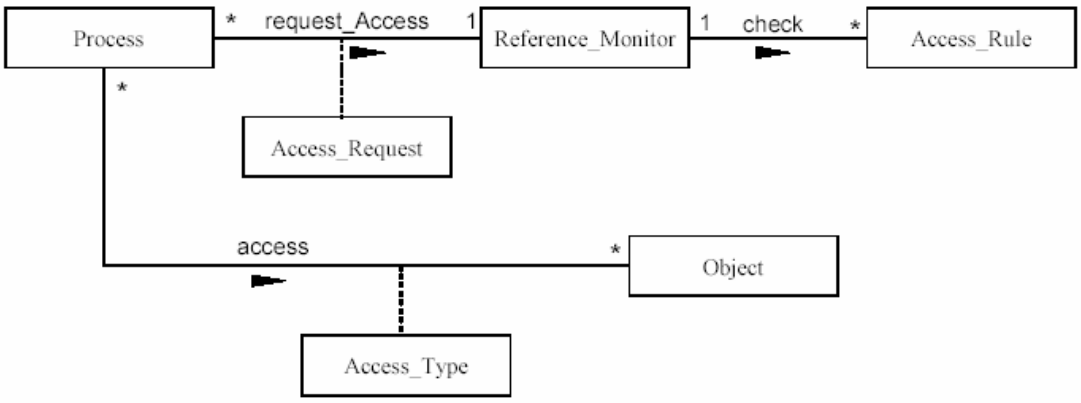


图10 受控对象监视器模式类图

结论

此模式有以下优点：

- 访问规则可以用权限矩阵实现，这样可以对每个主体都定义不同的权限。
- 每个被截获的访问请求都会根据授权规则允许或拒绝。

（可能）有以下缺点：

- 需要保护授权规则。
- 每个访问都需控制，这增加了系统的负担。

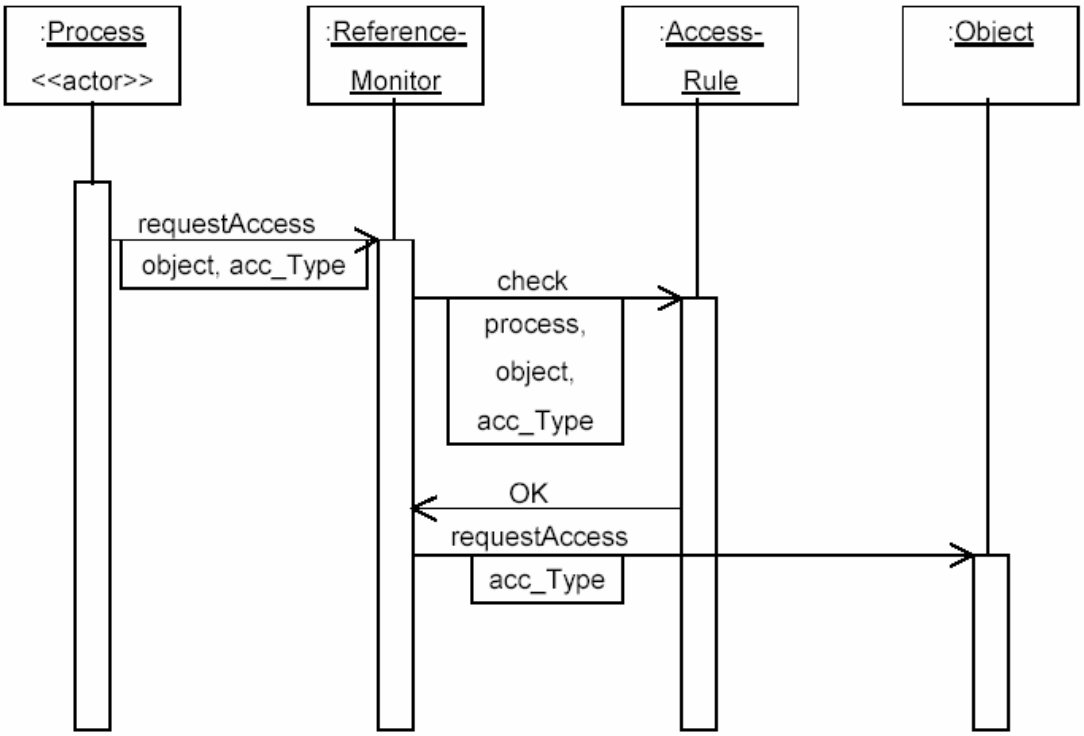


图11 验证访问请求的时序图

实现

一种可能的实现是：在用户登陆时验证用户。新建对象继承了包含在令牌中的原始用户的ID。这个与用户进程相关的令牌决定访问权限。只有通过这些受控的方式才能访问安全对象。

每个用户想要访问的对象都可能有相应的访问权限表。它描述了用户对对象有什么样的访问权限。它同样描述了系统中的其他对象有什么权限。通常，权限可以是允许或拒绝。这就是在Windows环境中常见的访问控制实体（ACE）[Har01,Mic00,Zac99]。这些实体的集合就是在Windows或其他操作系统中常说的访问控制列表（ACL）。

另一种与ACL相反的方式是能力（capabilities）集。一个能力对应访问矩阵中的一行，而ACL与对象相关（访问矩阵中的一列）。能力显示主体有权力访问安全对象。能力可以附带一些认证特征，这样对象可以信任主体提供的能力信息。一个全局表的每一行代表了每个认证用户的能力。或者能力集可以实现为一个用户列表，它包含用户对那些对象有访问权限[Kin01]。

模式的综合使用

下面的例子综合使用了以上的模式：

Windows NT

Windows NT安全子系统使用了上面描述的模式。它包括三个组件[Har01，Kel97，Mic00]：

- 局部安全授权（LSA）
- 安全帐号管理器（SAM）
- 安全引用监视器（SRM）

局部安全授权（LSA）和安全帐号管理器（SAM）配合起来认证用户并创建用户的访问令牌。安全引用监视器运行在内核中提供强制的访问检查。当请求一个对象时，需要比较文件的安全描述符和用户令牌中包含的SID信息。安全描述符由访问控制列表（ACL）中的访问控制实体（ACE）组成。如果对象有访问控制列表，安全引用监视器（SRM）检查每一个ACE以确定是否允许访问。一旦SRM授予主体访问对象的权力，后续的检查就不需要了，因为这时主体已经拥有了对象的句柄。

访问权限包括：不能访问，读，修改，完全控制和特殊访问。对目录的访问又增加了几种权限：列表，增加，读。

更进一步，Windows使用句柄的概念定义对系统中受保护对象的访问。每个对象都有一个安全描述符，描述符中包含了对象的自由访问控制列表（DACL）。同样，每一个进程都有一个安全令牌用于标示进程。它被内核用来确定进程的访问权限。ACL中的ACE描述了每个特定的进程SID有何种权限。内核在收到访问请求时检查对象的ACL。

当进程需要对象句柄时请求访问对象。例如，调用CreateFile()。CreateFile函数用于创建一个新文件或打开一个已存在文件。创建文件时安全描述符作为参数传递给函数。如果打开已存在文件，这个安全描述符就附加在文件句柄上，它描述了文件的访问权限，如GENERIC_READ。如果进程有相应的权限，请求成功并且返回访问句柄。所以，同一对象的不同句柄可能有不同的访问权限[Har01]。一旦取得句柄，后续的访问，如读文件就不需要权限检查了。同样，句柄也可以传递给另一个可信的函数用于后续处理。

Java 1.2 Security

Java的安全子系统用到了这里描述的安全模式。Java的安全控制器根据权限和策略创建访问权限。它用checkPermisson方法来验证每个方法的字节码，并使用策略对象确定相应的权限对象。值得注意的是checkPermisson方法会遍历整个调用栈并确定栈中每个方法的权限。java.policy文件被安全管理器使用，它里面包含了每个字节码的授权信息。

致谢

我们感谢Wolfgang Keller，他的非常有价值的建议使这篇文章更加完美。

参考资料

[And01] R. Anderson, *Security Engineering*, Wiley 2001

[Bro99] F.L. Brown and E.B. Fernandez, "The Authenticator pattern", *Procs. of Pattern Languages of Programs Conf. (PLoP99)*, http://jerry.cs.uiuc.edu/~plop/plop99

[Fer99] E.B.Fernandez, "Coordination of security levels for Internet architectures", *Procs. 10th Intl. Workshop on Database and Expert Systems Applications, DEXA99*

[Fer01] E B. Fernandez and R.Y. Pan, “A pattern language for security models”, *Procs. of PLoP 2001*, http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/acceptedpapers.html

[Fer02] E.B.Fernandez, "Patterns for operating systems access control", *Procs. of PLoP 2002*, http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings.html

[Gar02] S. Garfinkel, *Web Security, Privacy & Commerce*, 2nd Edition O’Reilly 2002.

[Har01] J. M. Hart, *Win32 System Programming*, Second Edition, Addison Wesley 2001

[Has02] J. Hassell, *RADIUS*, O'Reilly, 2002.

[Hay00] V. Hays, M. Loutrel, and E.B.Fernandez, "The Object Filter and Access Control Framework", *Procs. of PLoP 2000,* http://jerry.cs.uiuc.edu/~plop//plop2k/proceedings/proceedings.html

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

[Kel97] M. Kelley, "*Windows NT Network Security, A Manager's Guide,"* Lawrence Livermore National Laboratory, 1997.

[Kin01] C. King, et.al., *Security Architecture*, Osborne McGraw Hill 2001.

[Lan99] C. R. Landau, "Security in a Secure Capability-Based System," *Operating Systems Review,* October 1999.

[Mic00] Microsoft, *Windows 2000 Security, Technical Reference*, 2000

[sam] SAML, http://www.saml.org and http://www.oasis-open.org/committees/security/

[Sch00] D. Schmidt, et.al. "Pattern-Oriented Software Architecture," Wiley 2000.

[Sil03] A. Silberschatz, P. Galvin, G. Gagne, *Operating System Concepts (6th Ed.),* John Wiley & Sons, 2003.

[Vis99] P. Viscarola, "*Windows NT Device Driver Development*," Macmillan Technical Publishing, 1999.

[Zac99] W. H. Zack, *Windows 2000 and Mainframe Integration*, Macmillan Technical Publishing 1999.

The Addison Wesley Signature Series

A MARTIN FOWLER SIGNATURE BOOK

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

远程认证者和授权者模式

Eduardo B. Fernandez、Reghu Warrier 著，杨德仁 译

查看评论

摘要

许多分布式系统包括具有各种计算设备的节点，这些设备要访问共享资源。这种环境需要一个安全和容易管理的认证和授权机制。我们在此描述了一个用远程认证协议实现这个目标的模式。这是一个复合模式，包含2个已知的模式：代理和基于角色的访问控制。

意图

在访问松散耦合的分布式系统中的共享资源时，提供认证和授权便利。

例子

一个跨国公司在不同国家如美国和巴西有许多雇员。支持美国职员的认证和授权信息要保存在美国的服务器中，支持巴西职员的信息要保存在巴西的服务器中。现在假定一个美国职员旅行到巴西，需要访问巴西数据库服务器中的数据。有两种方法可以实现：

1. 在巴西服务器中复制职员用户信息，得到访问数据的授权。

2. 借用有相似权限的巴西雇员的用户名称，用于访问需要的信息。

两种方法都有缺点。系统管理员需要以协调方式在每个被访问的系统中创建和管理用户帐户，以维护安全策略实施的一致性。如果雇员的用户名被借用，就会危及用户责任。

上下文

松散耦合的分布式系统诸如互联网，由许多计算节点组成，其中有些节点需要共享资源。例如，在几个国家具有分部的公司。

问题

在不需要冗余的用户登陆信息下，如何在分布式环境中提供认证和授权？

在过去几年，电信、互联网和电子商务从做生意的一种选择日益发展成主流消费者的活动。对公司数据安全的关心程度急剧增长，对用户注册多个领域和多个服务正在变成需要而非奢侈。具有中央服务的系统能提供容易的管理、更多的责任和安全认证。

约束

在多处存储用户认证和授权信息，产生冗余，难于管理，容易造成不一致。

虽然认证信息能存储在任何地方，但这个位置应该对用户透明。

用户一般工作在角色上下文中，这些角色是标准的，无论跨多少领域，至少在公司或机构内部。

借用本地用户的登陆权限，难于分清用户责任。要在用户访问资源时保存用户标识。

方案

设置一个单点入口，能把用户重新导向正确的服务器，以便用户的登陆和访问信息被确认。

我们用特殊的认证/授权服务器实现这种重新导向。这种服务器用户嵌入式网络设备诸如路由器、modem服务器、交换机等。认证服务器负责接收用户连接请求、授权用户并返回客户端需要给用户传递服务的所有配置信息。图1显示了这种方法。客户端通过代表包含用户登陆信息的服务器的代理服务器请求服务。这个请求被路由到远程服务器，后者根据基于请求主体的角色以及对保护对象的权限来验证。

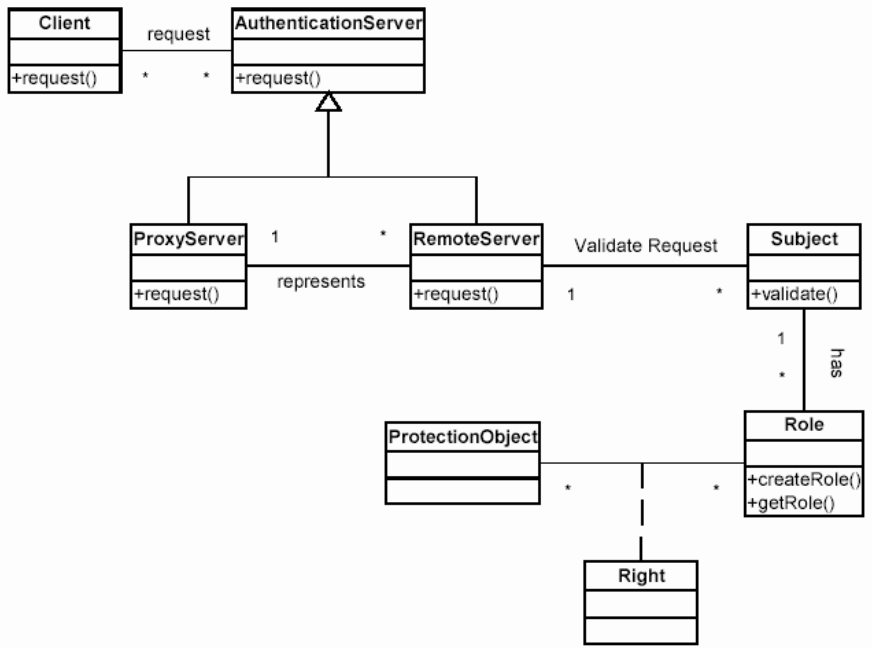


图1 远程认证/授权模式的类图

动态结构

典型系统用下述信息：

访问请求：由客户端发送请求认证和授权网络访问连接尝试。

访问接收：由服务器发送对访问请求信息的响应，通知客户端连接已经被认证和授权。

访问拒绝：由服务器发送对访问请求信息的响应，通知客户端连接请求被拒绝。若信用没有得到认证或连接没有得到授权，服务器便发送这个信息。

访问挑战：由服务器发送对访问请求信息的响应，告诉客户端这个信息需要回答。

帐户请求：由客户端发送

帐户请求：由客户端发送去指定被接收的连接的帐户信息。

帐户响应：由服务器发送响应帐户请求消息。这个消息承认成功的收到和处理帐户请求消息。

消息由头和属性组成。每个属性说明了连接尝试的一块信息。

下述场景（图2）阐明了一个在客户端和转发/远程服务器之间的基于代理的通信：

1. 客户端向转发服务器发送访问请求。
2. 转发服务器把访问请求转发给远程服务器。
3. 远程服务器给客户端发送访问挑战。
4. 转发服务器把访问挑战发送到客户端。
5. 客户端为挑战计算响应并经过第二次访问请求把它转发给转发服务器。
6. 转发服务器把这个访问请求转发给远程服务器。
7. 如果响应与希望的响应匹配，则远程服务器用访问接收回答，否则使用访问拒绝。
8. 转发服务器给客户端发送访问接收信息。

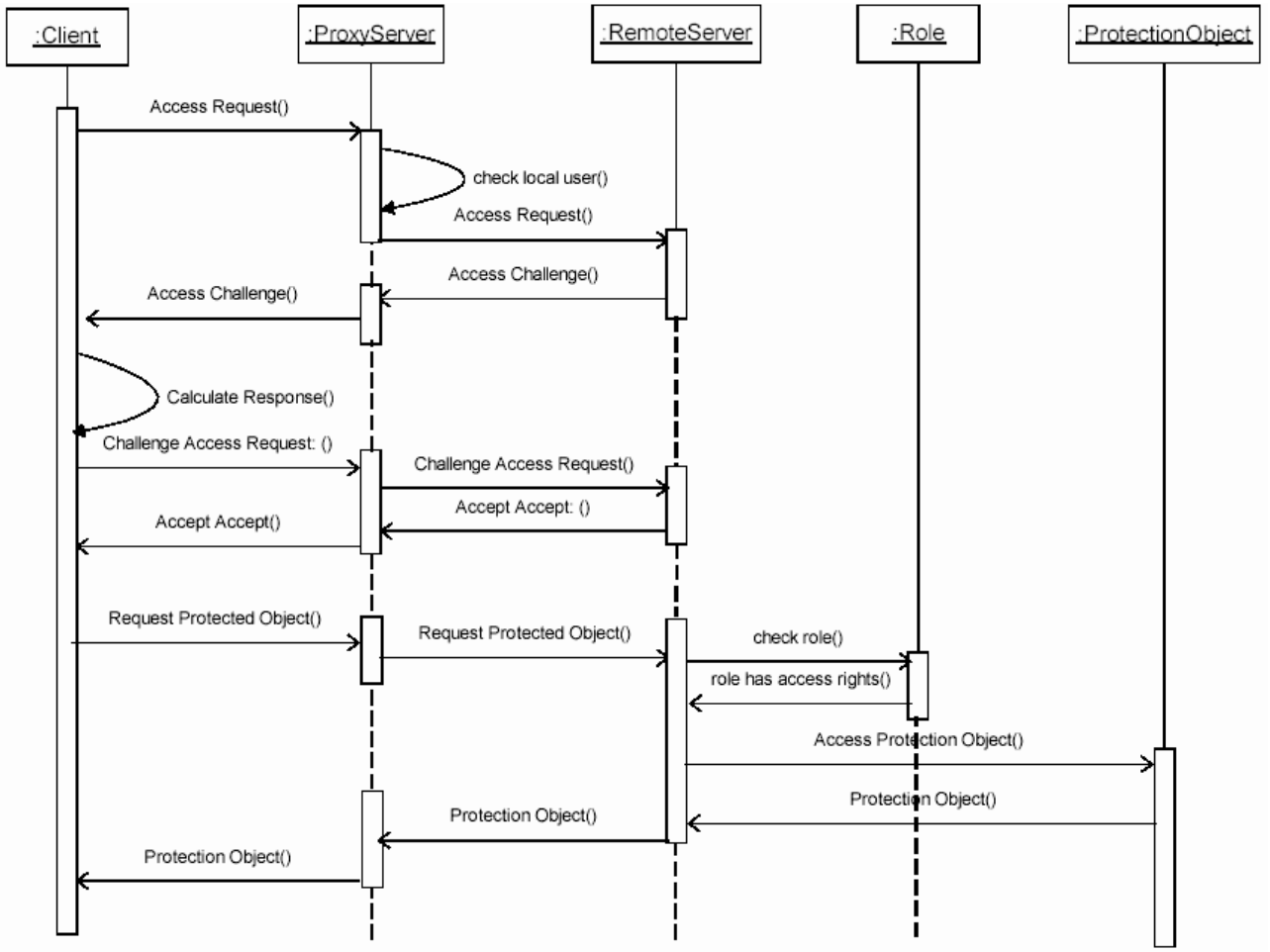


图2 客户端认证的顺序图

实现

一个认证服务器能同时作为转发服务器和远程服务器，有些域作为转发服务器，为其它域作为远程服务器。转发服务器能为许多远程服务器作为转发者。远程服务器能有多个向它转发的服务器，并能为任意数目的区域提供认证。转发服务器能向其它转发服务器转发从而形成一个代理链。需要查询服务去查找远程服务器。

结论

这个模式有下述优点：

漫游准许多个管理实体允许各自用户向对方实体网络拨号请求服务。

在单点存储用户登陆和访问权限使之更安全和容易管理。

用户的登陆ID、口令等存储在内部拨号数据库或能被从一个SQL数据库访问。

用户信息存储的位置对用户透明。

角色和访问权限是标准的，可以跨多个位置。

服务器和客户端应该支持基础协议。

单元诸如活动卡[ACS]允许复杂的请求/挑战交互。

也有一些缺点：

用到的额外信息增加了负载，这样就降低了简单请求的性能。

这种系统比直接认证客户端的系统更复杂。

实例解释

当美国职员旅行到巴西时，他登陆到远程认证者/授权者，后者把请求重新路由到存储其登陆信息的美国服务器上。

已知应用

远程认证拨号用户服务(RADIUS)是一个广泛部署的IETF协议，以完成网络访问的中央认证、授权和记帐[Has02, Rig00]。起初为拨号远程访问开发的RADIUS现在已经被虚拟专用网(VPN)服务器、无线访问点、认证以太交换机、DSL访问和其它网络访问类型[Hil]支持。图3显示了使用挑战响应方法的RADIUS服务器中的客户端的典型认证顺序。

使用代理RADIUS，RADIUS服务器接收来自RADIUS客户端（如NAS）的认证(或帐务)请求，把请求转发给远程RADIUS服务器，接收来自远程服务器的应答，并给客户端发送应答。代理RADIUS的一般用途是漫游。漫游允许两个或多个管理实体让对方用户拨入自己实体的网络请求服务。

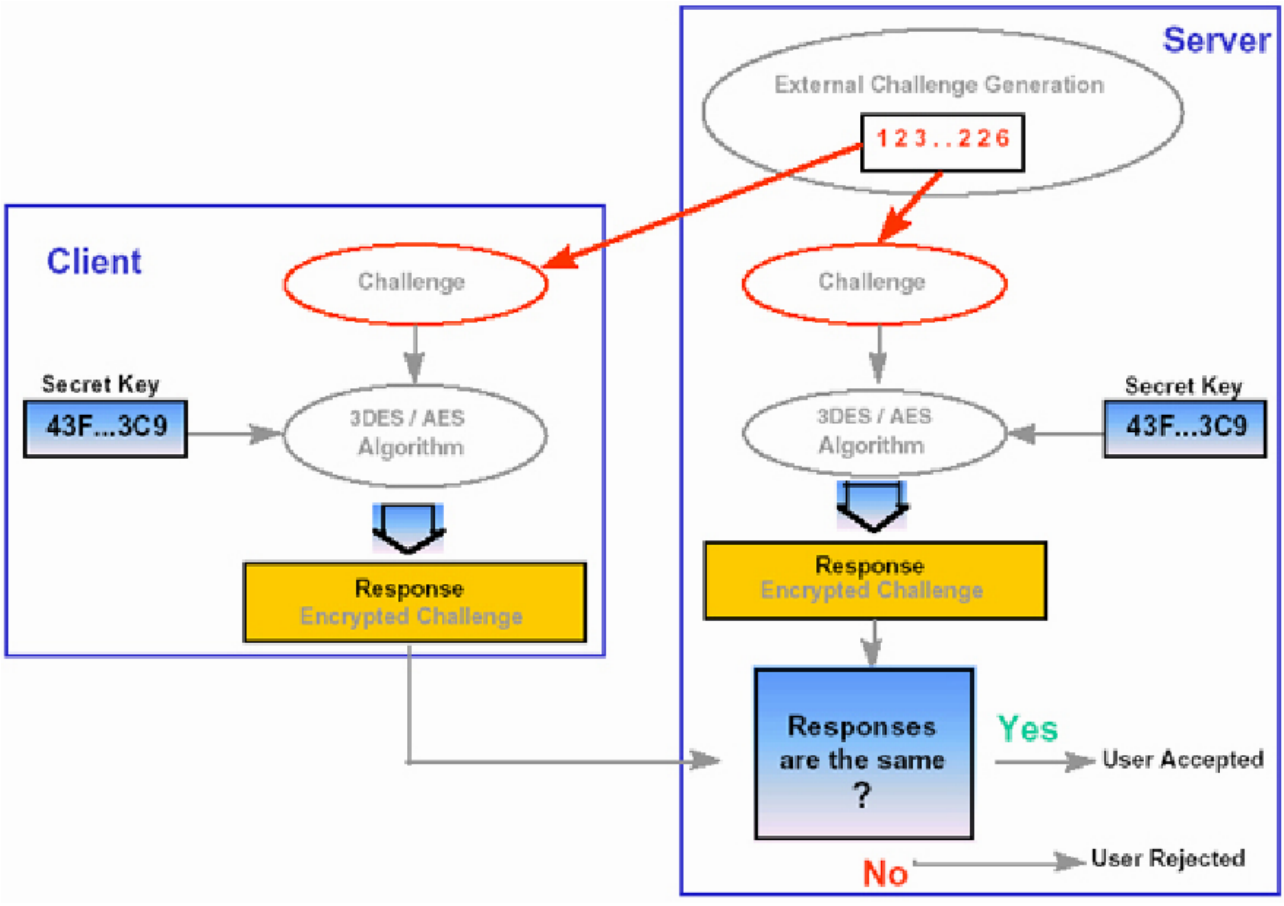


图3 RADIUS挑战/响应模式

今天市面上使用许多商业RADIUS服务器。它们包括：

FreeRADIUS

FreeRADIUS服务器[Fre]用于Unix操作系统的后台程序，允许设置通常用于拨号用户的认证和帐务的radius协议服务器。FreeRADIUS 是开源产品，具有开源的所有优点。

Steel-Belted Radius

Steel-Belted Radius是RADIUS的完整实现。它提供丰富的用户认证、授权和帐务功能，完全支持代理RADUS。它能：

向其它RADIUS服务器转发代理RADIUS请求。

作为处理来自其它RADIUS服务器请求的目标服务器。

把帐务信息传递给目标服务器：执行认证的服务器或另外的服务器。

NavisRadius

NavisRadius是提供认证、授权和记帐证（AAA）服务的RFC标准RADIUS协议的补充。NavisRadius为服务供应商和载体提供一个集成网络范围远程访问安全的方案。NavisRadius支持RADIUS由IETF RADIUS RFC 2865 (RADIUS认证)和2866 (RADIUS 记帐)定义的标准，用于提供广大范围的网络服务。

早期的认证服务器用于CKS, MyNet和Security Dynamics [CTR96]产品中。

相关模式

整个体系结构是单点检查模式[Yod97]的应用。它利用代理模式[Gam95]作为基本组件。最后，利用基于角色的访问控制模型[Fer01, Yod97]定义了用户权限。文献[Bro99]中给出了用于认证分布式对象的一个模式。

参考文献

[ACS] ActivCard Synchronous Authentication (http://www.activcard.com/activ/services/library/synchronous_authentication.pdf)

[Bro99] F.L. Brown and E.B. Fernandez, "The Authenticator pattern", Procs. PLoP99.

[CTR96] “Security: Resellers getting the advantage of growth”, Computing Technology Review, Febraury 1996, 14-17.

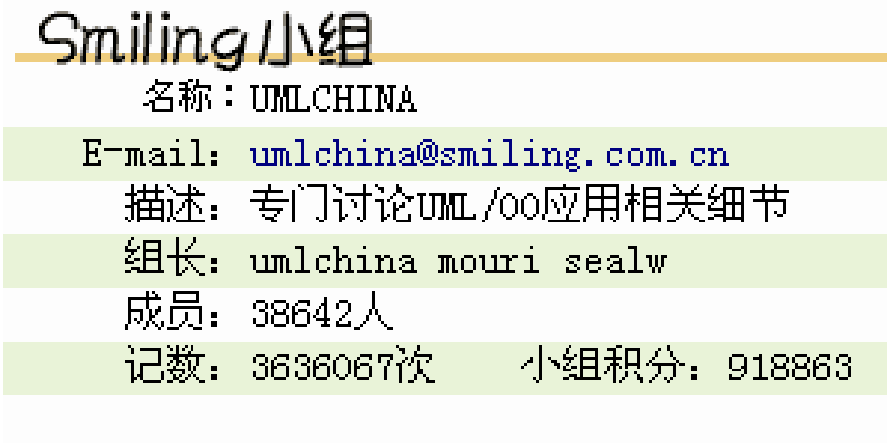
[Fer01] E B. Fernandez and R.Y. Pan, “A pattern language for security models”, Procs. of PLoP 2001, http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/acceptedpapers.html, http://www.cse.fau.edu/~ed/SecModels.pdf

[Fre] http://www.freeradius.org/mod_auth_radius/

[Gam95] E. Gamma, R. Helm,R. Johnson, and J. Vlissides, Design patterns –Elements of reusable object-oriented software, Addison-Wesley 1995.

[Has02] J. Hassell, RADIUS, O’Reilly, 2002.

[Hil] J. Hill, “An Analysis of the RADIUS Authentication Protocol”, InfoGard Laboratories, http://www.untruth.org/~josh/security/radius



*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。

帐务模式

Windy. J 著

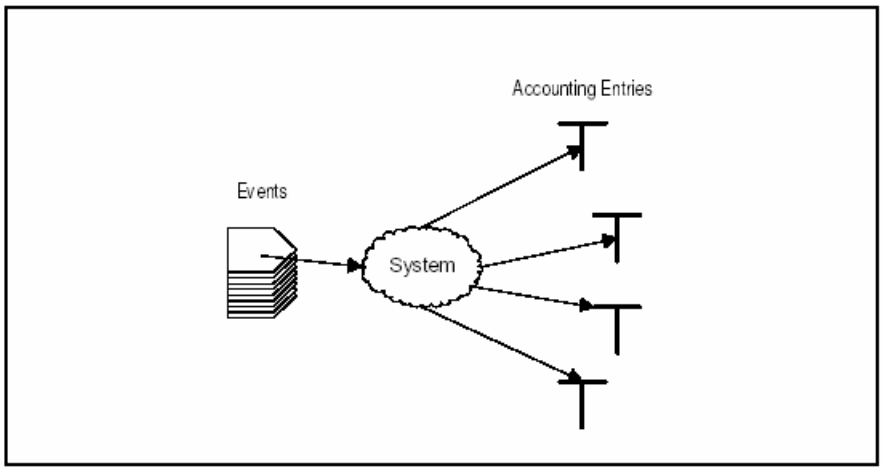
查看评论

下面是从深圳移动公司得到的部分话费清单，上面记录了我在 2002 年 8 月 1 日的几个电话记录，无一例外的是，当电话接通以后，一笔钱就记在我的移动电话号码上了：

序号	呼叫发生地	开始时间	主被叫	对方号码	通话时长(秒)	移动话费	长途话费
1	755	2003-08-01 09:31:36	主叫	0755xxxxxxxx	00:00:16	0.40	0.00
2	755	2003-08-01 11:39:28	主叫	0755xxxxxxxxx	00:01:34	0.80	0.00
3	755	2003-08-01 11:43:33	主叫	0755xxxxxxxx	00:00:02	0.40	0.00
4	755	2003-08-01 11:44:49	主叫	13xxxxxxxxx	00:02:23	1.20	0.00

我们打电话，上网，然后在月底收到一张账单，上面写着这些活动产生的费用，类似的还有水电费，煤气费，有线电视服务费，购物账单等等等等，总之，在某个时候发生了一些事件，打了一个电话，上了一次网……，在这些事件里我们付出或者得到了某些东西（例如劳动，商品，服务等），然后这些付出或得到的东西产生了一些不同程度的费用条目，当然还伴随着我们的某种体验和感觉，开心或者无奈，但这些就不在帐务模式的讨论范围之内了。

下面一个生动的图画描述了事件（Events）通过将要建造的系统（System）产生账单条目（Accounting Entries）的情况：



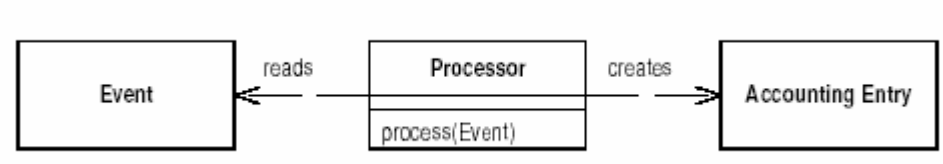
帐务模式（Accounting Patterns）要讨论的，不光是对费用的记录，还包括费用产生的原因和它产生的过程，例如这些事件本身，它们的类型，以及从这些事件产生帐务条目的规则，还有，如果出现错误应如何处理。

Martin Fowler 先生在他的职业生涯中经历了多个跟帐务有关的项目，并从中得到了许多宝贵的模式，正是这些模式，组成了帐务系统的基本模型，它们也是我们在建造帐务系统时不可缺少的参考资料。分析模式中每个系列的模式都非常经典，而帐务模式尤其如此，我们可以看到，Events，Posting Rule，Accounting Entries，Secondary Posting Rules 它们是如何一点点地展开，组成帐务系统的框架，笔者也在阅读了帐务模式以后，得到很多启发，并试着用来改进自己手上的项目模型，真的体会到了那种“蓦然回首，那人却在灯火阑珊处”的感觉。

帐务系统中名目繁多的事件和它们产生的帐务条目引起了很多人的兴趣，但是我们把它们放到后面，先来看看从事件产生帐务条目的整体过程。

处理器（Processor）

从前面那个电话说起，对于电话呼叫而言，假设有个处理器，读入呼叫事件，得到主叫号码，被叫号码，通话时长，并找到对应的价格，例如每分钟 0.4 元，就可以计算出这次电话对应的费用，并创建一个相应的条目。



合同（Agreements）

但世事并非如此简单，移动用户分神州行，全球通，移动梦网等类型，每一种的计费规则是不同的，难道在上面的处理器（Processor）中一一用 if…else 区分开来？如果以后这些类型变更或者增加新的分类怎么办？这样看来上面的模型太粗略了。

等一等，神州行，全球通，移动梦网用户入网的时候签订的合同是不同的，如果可以用签约合同把他们区分开来，这对降低处理器的复杂度来说无疑是个好消息，而且也可以增加新的合同来支持以后的变化：

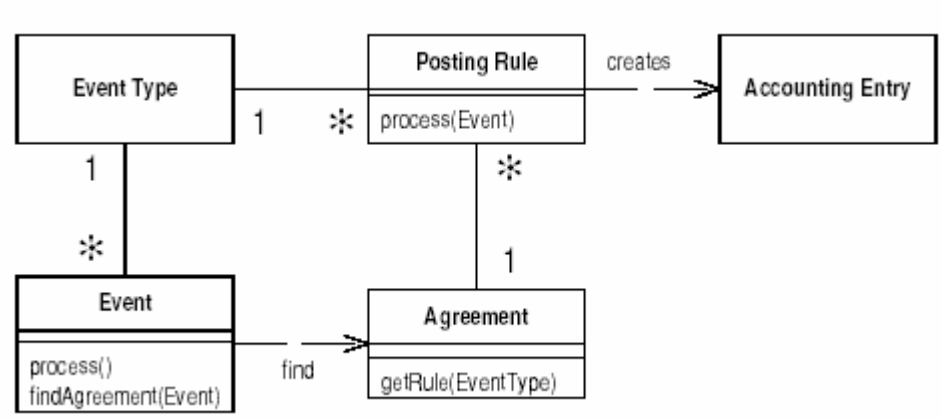
如上图所示，事件（Event）首先需要找到它对应的合同（Agreement），然后由合同里的处理器进行处理，并生成帐务条目（AccountingEntries）。

在这个例子里，合同的种类似乎不是很多，但我们可以想象一下一个大中型的公司，计算薪酬的时候需要用到公司与员工签订的合同，岗位不同，年限不同，部门不同……将会导致合同对象的膨胀程度——将远远超出上面这个例子。

引入合同也有它不足的地方，如果是每个合同内容完全不同，就好了，可是往往会在多个合同中出现同样的内容，那样就需要在多个合同对象里维护这些内容的一致性，这个问题我们将在下面讨论。

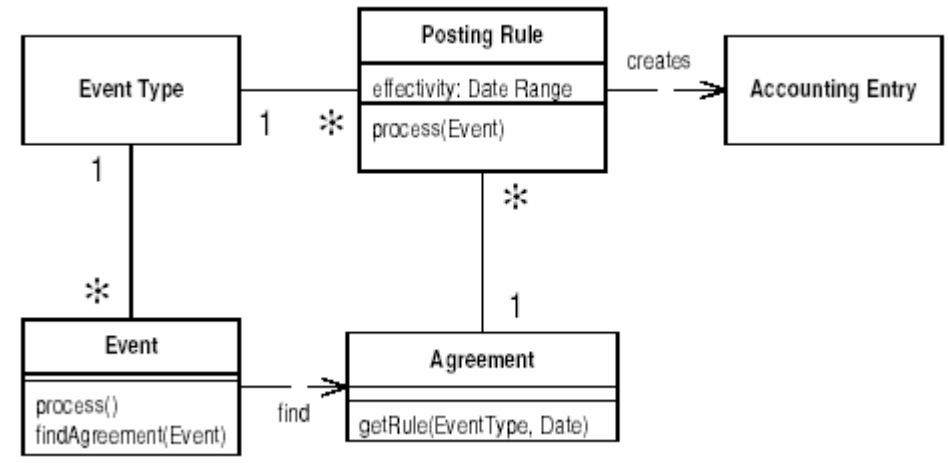
事件类型（Event Type）和处理规则（Posting Rules）

上面事件的例子是电话呼叫，其实还有其他的很多事件类型，例如煤气水电消费，购物，工作时间，等等，针对这些不同的类型，都有着不同的合同和处理规则，同时，我们还可以通过事件类型来查找相应的合同，然后找到具体合同规定的处理规则，对，同时把处理规则从合同中分离出来，就可以让合同支持多个处理规则，还可以在合同之间共享一些规则，刚好解决了上面提到过的合同部分相同问题。



规则的有效时间

还有一个引起变化的重要因素是时间，有一些处理规则本身就基于时间变化，同时，处理规则还会有有效时间限制，这个时候，想起我们在前面提到过的“对象有效期和临时性”等一系列模式，把其中的 Effective 模式应用到 Posting Rule 上就可以了。请看下面的图示：



如果用 if…else 实现

上面的模型是为了解决不同的事件，多个合同，多种处理规则，以及规则有效性的问题，如果我们不用面向对象技术，改用过程技术，将会怎样呢？我们不妨来看看下面模拟过程技术实现的伪代码：

```
Process(Event e)
    if (e.getAgreement().equals(agreement1) )
    {
        if (e.type().equals(eventType1)
        {
            if (e.date().laterThan(aug21)
            {
                multiplyByRate(e,e.getCustomer().getRate(0));
            }
            else if ()
            {…}
        else if (e.type().equals(eventType2)
        {…}
    else if (e.getAgreement().equals(agreement2) )
    {…}
}
```

在上面我们看到，到处都充满了 if…else 语句，如果增加或修改了某种条件，会影响到很多地方，而采用面向对象技术，完全可以实现上述的功能，也将去掉那些条件语句。Event 将根据自己的 EventType 找到它所在合同规定的处理规则（Posting Rules），然后进行处理（例如上面语句中的 multiplyByRate），创建新的帐务条目。

需要新的规则时，不用改变原来规则的代码，只需要建立新的规则对象，然后把它装配到合同中去，这跟修改所有的条件语句比起来，简洁而不易出错。还有，考虑到前面提到的规则有效时间段问题，如果因为有效时间而规则一直在变化，例如上个月 0.3 元一分钟，这个月 0.32 元一分钟，下个月又变成 0.4 元一分钟，那么对 if…else 语句的修改就永无宁日了。OO 技术能有效地减少代码中条件判定语句的数量，同时为变化带来新的解决方案，这些方案不用象原来那样频繁地修改代码，这无论对代码质量、维护工作量以及系统的灵活性来说都有着重要的意义。

还有更重要的意义：根据模型建立上述相关的系统，将让系统变得结构清晰，并把责任合理分离到各个互相协作的类，它们各自专注于自己的责任，这样系统将更加稳定，易于扩展（可以对组成系统的类分别进行扩展），在以往的帐务系统中我不止一次听说过那些 1000 行以上的函数代码，复杂的逻辑，纠集的责任，一再地修改，用 goto 跳过的代码还混在其中，很难得到稳定清晰的结构，结果到最后，往往连这些代码的作者也看不明白了。

在不断变化时，采用面向对象技术仍然可以保证系统结构的相对清晰，将变化影响的范围缩小，为系统提供清晰，良好的扩展点，这是 OO 的最重要的意义所在。

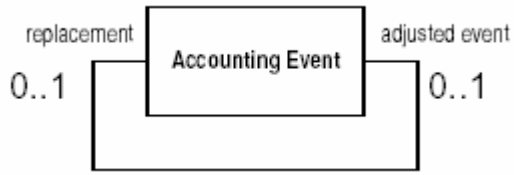
当逻辑简单时，用条件语句当然也更容易，如果发现逻辑复杂，就可以试着进行分析，用区分责任的类和它们的协作来替代这些条件语句，面向对象技术也提供了多态机制来实现这种替代，基于公共父类的接口，用具体子类的实例实现，就可以得到想要的变化。什么时候可以用条件语句，是不是还有更合适的方案，就让我们带着这个问题去分析吧。

错误更正

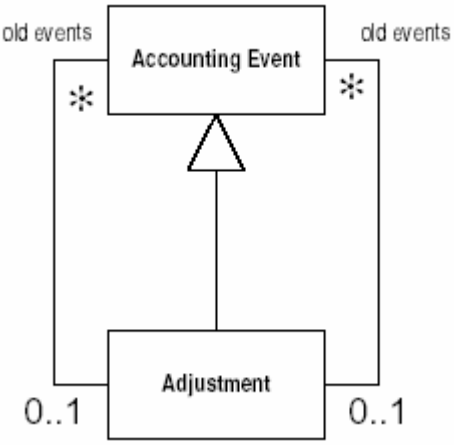
如果处理完以后发现计费的规则已经改变，或者原来的事件数据有错误，又该如何处理（计算机系统应该允许人们犯错误，毕竟，犯错误也是生活的一部分）？这是帐务系统另一个需要注意的地方。

对帐务系统而言，一个重要的原则是对已经处理的事件和已经产生的帐务条目不能擅自改动，一切都要能回溯，从处理结果到原始事件，以及处理过程，都要保持原状。那么，一旦真的出错了，又该怎么处理呢？

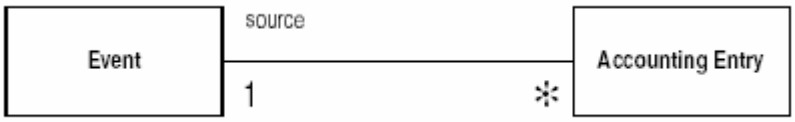
首先，把原始的事件（Event）进行额外的标识，并让它跟另一个事件实例：更正事件（Adjusted Event）联系起来，用这个新的事件进入下一步处理来更正原来事件的影响，如下图所示：



也可以单独让一个更正事件（Adjustment）成为事件（Event）的具体子类，并建立它跟原来事件的关系来完成更正动作：



要保证这个更正事件能完全消除原来错误事件的影响，并进而创建正确的帐务条目，我们还需要得到由于原来错误事件创建的条目，以及需要保证更正的事件能正确进行处理。要得到原来创建的条目，可以在事件和条目之间建立关联，例如：



这样就可以找到它们了。

如何调整帐务条目

一共有三种方法来调整错误事件引起的帐务条目，它们分别是：替换调帐（Replacement Adjustment），对冲调帐（Reversal Adjustment），和差异调帐（Different Adjustment）。

替换的方法是要找到错误事件产生的所有帐务条目，把它们改成正确的值或者用新的条目替换它们，这种方法简单，但不一定可行，因为财务审计检查的需要帐务条目经常是不可改变的，还有，可能后续的财务动作已经完成，例如付款等，这些是无法改变的，因此替换方法在许多时候不适合。

第二种方法是找到所有错误事件产生的帐务条目，并为它们每一个增加一个日期相同，金额数量相同，但为负值的条目，来消除原来错误事件的影响，然后就可以用新的事件来产生正确的帐务条目了。

这种方法的缺点会在系统中产生大量无用的帐务条目（想想每条错误帐务条目就有一个负值条目跟它对应），这可能是一件让人烦恼的事。

第三种方法是差别调整，找到那些错误的帐务条目，把它们跟正确条目之间差别补充进入系统，这样不会象第二种方法那样产生大量的无用条目，也保证了原始的事件和条目依然存在，但它的实现会比第二种方法更复杂一点。

基本上我们可以把这三种方法结合起来使用，按系统的复杂程度，对历史信息进行追溯的要求，以及有没有可能有些动作无法撤销，考虑了整体情况，再决定用哪种调整方式。

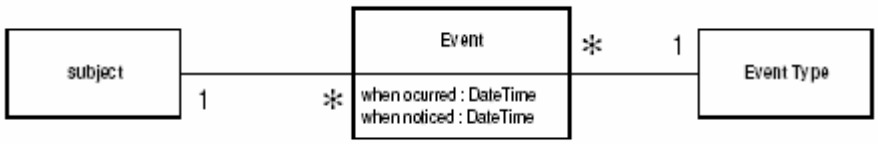
这里只是笼统地介绍了一下三种调整方法，在后面我们还将详细讨论它们，并将给出一些样例代码。

事件（Event）

事件当然也很重要，一来它是产生费用的基础（原始依据），二来是它们构成了我们的生活，而不是钱，想一想，在没有人烟的沙漠里背着一袋金子的感觉，虽然富有，但是连水都买不到，会不会觉得很无奈？

跟事件有关的还有人或团体，事件发生的时间，事件生效的时间（假设一次购物的时间发生在中午，而通知银行的时间发生在下午；又比如全球通的漫游长话通知归属局的时间一般在发生通话当月月底或下月月初），事件类型，当然依类型而定还会有一些详细信息，例如电话记录中的被叫号码，通话时长等。

事件提供了一个记录，除了帮助我们回想当时发生的事情以外，它们还是帐务条目产生的原始依据，同时，它们也是随着时间过去而无法更改的，昨天打过的电话，今天可以再打，但已经是一个新的事件了。这样看来，它们跟日志记录模式中描述的 Log（日志记录）特征一样，那么也可以应用 Log 模式来处理。

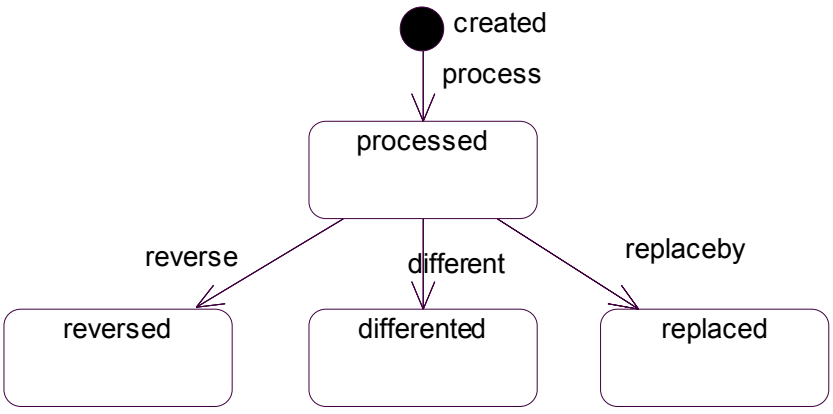


由于事件不能更改，所以系统必须保证它们不可改变，如果真的发现当时由于客观原因事件出现了错误，也不能更改原始事件，而应该用前面提到过的对冲（Reversal Adjustment），和差异调整（Different Adjustment）方法进行调整（上面的调整方法包括产生新的*补充*事件记录），在区分系统数据类别的时候，事件作为系统的源数据，相应的处理结果应为处理数据/结果数据，或者还存在一些用来计算的中间数据（例如目前长途电话 6 秒为计费单位，跟长途电话通话时长 20 秒对应的计费时长——24 秒就是中间数据），后两者都是经过系统加工处理的数据。

和日志条目一样，时间也是事件的要素之一，对时间精确程度的要求依系统而定（例如电话呼叫的发生时间一般精确到秒）；同时，需要一个时间还是两个时间也依系统而定，有时确实只需要一个时间就够了，不过为了明确起见，无论什么时候都最好给时间一个确切的名称。

从前面的模型来看，还有一个跟事件相关的 Subject（主题），事件是围绕谁产生的，谁就是事件的主题。

我还要补充一点的是，事件还有状态，根据错误事件账目调整方式为对冲调整，事件会有有效和无效两种状态，根据系统处理过程的复杂程度，还会有许多中间过程，例如已登记，已定价，已收款等等，可以单独为事件类画一个状态图，例如最简单的状态用下面的图形表示：



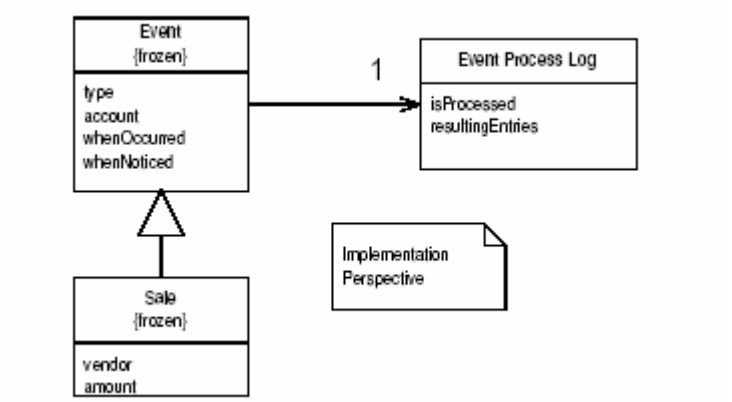
另外，我们试图把事件中完全不可改变的部分和可变的部分剥离开来，可变的部分包括事件状态，它产生的帐务条目引用等等，这样，改变它可变的部分就不会影响到它不能变的因素了。

最基本的事件（Event）接口如下：

```
interface Event
{
    Event newEvent(EventType type, Date whenOccured, Date whenNoticed);
    EventType getType();
    Date getWhenOccured();
    Date getWhenNoticed();
}
```

从上面可以看到，这个接口只提供了读访问，创建以后就无法再更改其中任何属性，这也是我们在前面一直讨论的原始事件不可更改性。

然后可以看看另一个销售事件（Sale Event）：



```
interface Sale extends Event
{
    Sale newSale(Date whenOccured, Date whenNoticed,Vendor vendor,Money amount);
    Vendor getVendor();
    Money getMoney();
}
```

它的类型隐式定义为跟类型一样：Sale，并补充了自己必要的一些属性和读访问接口。

而另一部分可变部分的接口举例如下：

```
interface Event
{
    Boolean isProcessed();
    Set getResultingEntries();
    void addResultingEntry(Entry arg);
    …
}
```

在实现的时候，把可变接口和不可变接口分开可能比较好，概念清晰，而且，不可变部分经常会形成继承体系，那么在整个继承体系中保持不可变的一致性；对于可变的部分，我们可以把它委托给一个独立的对象，例如EventProcessLog（事件处理记录），在这里记下事件当前的状态和处理信息，而这些信息变化并不会影响到事件本身，同时它还可以保持一个事件继承层次统一的可变部分接口。

也可以反过来从一个负责可变部分的类中把事件部分委托出去，这样类似一个管理事件变化的类；还可以用第三个类，让它管理一个事件类（及其子类）和一个处理记录类，具体怎么做看具体的系统了。

实际上，如果不想这么复杂，也可以把两部分接口综合起来到一个类，只给可变部分提供写、更改方法，给不可变部分提供只读访问，就可以满足大部分的需求。

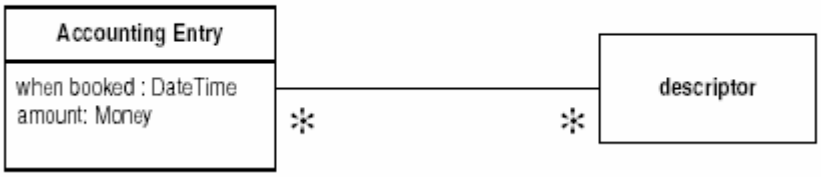
帐务条目（Accounting Entry）

帐务条目是根据事件得到的一个记录，它一定跟一笔某个数目的金钱有关，想想我们从超市出来的时候手上拿的小票，商品编号，名称，数量，单价，金额，优惠金额…这张小票上的清单就是每件商品作为单件商品购买事件，以及它产生的帐务条目，最后的总额就是这次购买事件产生的帐务条目，它也意味着我们购物车里面所有商品的价值（实际上，在超市购物的一个诱惑是，商品那么琳琅满目，购物车又那么方便，对这些东西要付多少钱我们常常要到交款的时候才有明确的认识），如果采用刷卡的方式付款，在银行卡/购物卡的记录里还会增加一个账目，它包括付款发生的时间，和这次购物的金额总和，或者一些别的描述，例如这次付款发送的终端等，如果是个人财务系统里记下这个账目，我们还会加上一点描述，说明在某某超市购买了某些商品。

帐务系统总而言之是处理关于钱的事情（而钱是大家都关心的），关于钱的数目，登记时间，它从哪里来，它作什么用途，在用电脑记录之前，这些内容被记在厚厚的大账本上，相信我们从以前很多故事、电影里都见过。

收入和支出都可以记录成帐务条目，它们表示你收到了一些钞票，又或者从口袋里掏出一些钞票给别人，也有些时候不涉及到具体的钞票，例如转账。

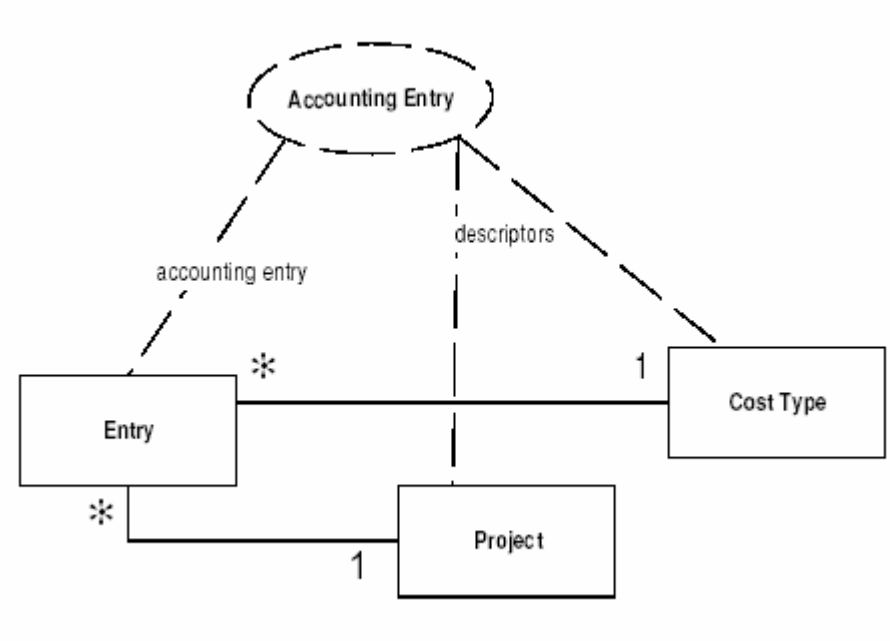
说到帐务条目，不免要提到账户（Account），这是另一个概念，我们会在后面提到它，现在就认为账户是对帐务条目的一种描述好了。



还有一点，有时我们或许会觉得可以把事件（Event）和帐务条目（Account Entry）合并成一个类，尤其是在简单的流水帐系统中，如果打算这么做，一定要看处理规则（Posting Rule）是否非常简单，是否可以忽略事件和帐务条目的差别。因为前面说过，事件有可变和不可变的部分，怎样保证这些得到满足；对帐务条目的更改会不会影响到事件；基本上还是建议将它们分开。

*事件——某件可能产生费用的事情；帐务条目——事件产生的费用；处理规则——从事件到帐务条目的计算规则。*如果再扩展开去，事件到事件或者帐务条目到帐务条目还有一些别的规则（例如统计），那么也可以定义一些规则来处理。

对帐务条目进行分解还可以得到 CostType（费用类型），Project（相关项目，表示该费用用于做什么），这两个部分是对帐务条目的进一步描述，例如前面购物的例子，费用类型是消费，项目是日常用品采购，或别的物品采购；这样来看就更完整了。



对帐务条目的更改也是我们要考虑的问题，一般来说，也要考虑帐务条目状态，当状态是 Open（开放）时候，是可以更改的，当状态为 Closed（已关闭），Booked（已登记）的时候就要保证它不能被修改了。

模式的应用

虽然帐务条目最多的时候是用来记录金钱数量的变化情况，其实它也可以用来记录其他数量的变化，例如库存商品，甚至源代码行的控制。然而金钱流动是商业社会最普遍关注的，所以要跟踪不同所有者之间的金钱流动时，我们就要考虑帐务条目了。

样例代码

```
interface Entry

{

Entry newEntry(CostType costType, Project project, Money amount, Date date);
```

```
        CostType getCostType();

        Project getProject();

        Money getMoney();

        Date getBookingDate();

…

}
```

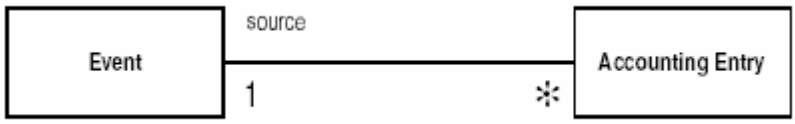
如果考虑到上面说过的状态问题，可以考虑 setCostType()的方法如下：

```
void setCostType(CostType arg)

{

        if (isOpen())

{

        CostType =arg;

}

else

{

        throw new immutableEntryException();

}
```

}

建立跟事件的关联

为了顺利进行调帐，前面提到要建立事件（Event）和帐务条目（Accounting Entry）之间的关联，把事件的角色当作来源（Source），帐务条目当作结果，就很容易看到从什么样的事件产生了什么样的帐务条目，以及很容易回答象这个帐务条目怎么来的这样的问题，如下图所示：

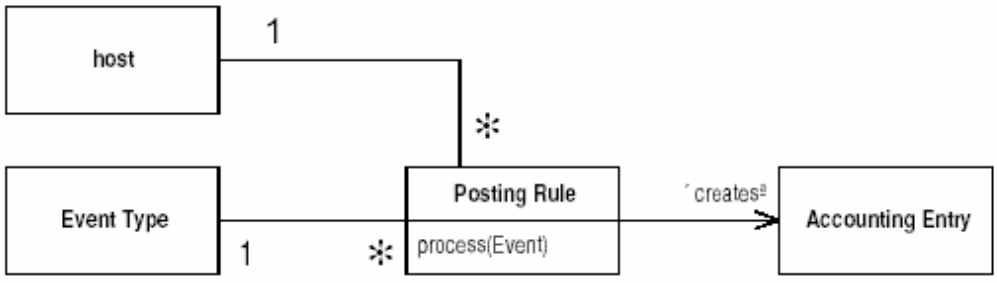


处理规则（Posting Rules）

处理规则是为了回答事件怎样形成帐务条目这个问题。

例如在我跟有线电视公司签订的有线电视服务协议中，关于费用的条款是每月固定收费 16 元，这就意味着我每收看一个月的有线电视（事件），就要付费 16 元（支出帐务条目）给有线电视公司，规则是每月 16 元，这是一个简单的处理规则。又例如上海市实行“峰谷电”电费制度，每度电的电价在晚上×点到早上×点是白天的一半，这样，从用电度数（事件）来计算电费（帐务条目）的处理规则就比刚才复杂很多了。

当然还有更复杂的处理规则，别忘了，为了处理帐务工作，每个公司都要聘请专门的财会人员，计算工资，收入，支出，等等，看这些金钱流动是否应该发生，以及保证它们的准确性。处理规则是这中间最灵活变化最多的环节，因为它是我们人类制定的，只要是人类的脑袋想得出来的，就有可能出现在处理规则中，例如我见过某个地方的电费政策，电力系统的职工可以享有一定的免费用电度数，用电单位从工商行政学校等等的电价各不相同，不同的区域也还有不同的耗电比例，等等，等等。



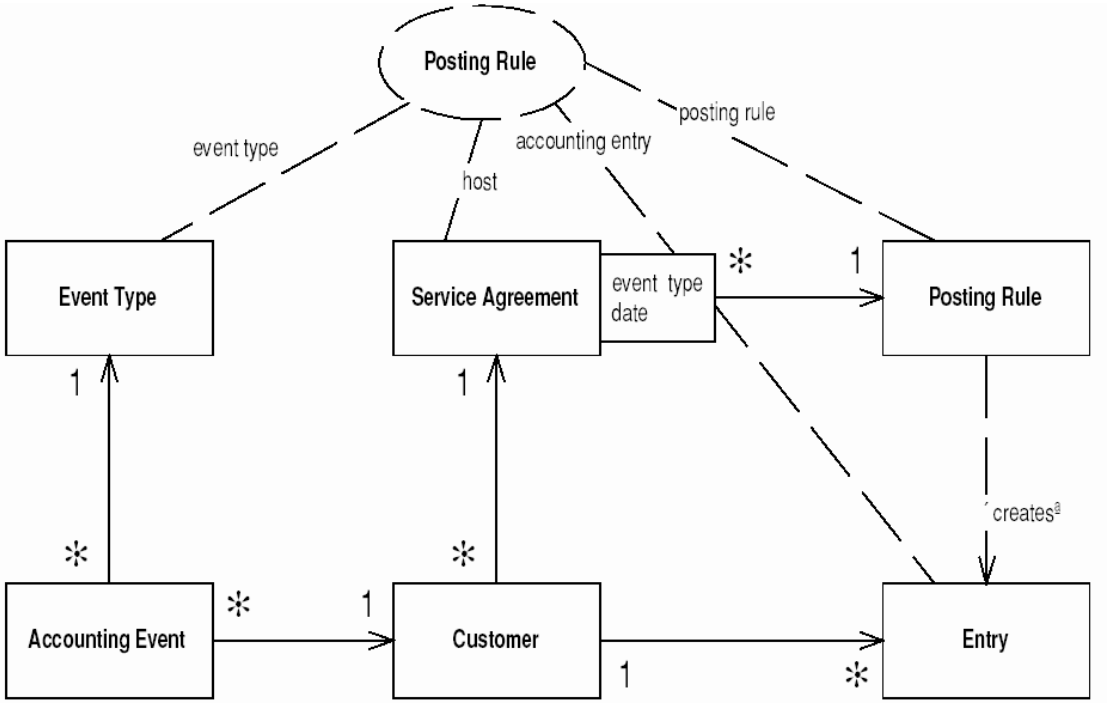
复杂之处可能还跟时间有关，这个在 Range（范围）模式中已经提到过，处理规则存在有效作用时间，而且这也意味着处理规则在某种程度上的不可变性。

模式应用

凡是要从事件计算相应的帐务条目时，应用规则可以处理非常灵活的计算方法，所以处理规则（Posting Rules）是非常重要的抽象，而且可以适应使用过程中的多种变化：跟据通用的接口，只需要补充具体的实现方法就可以了，如果处理规则出现变化不会影响到整个帐务系统，也不会影响到事件，帐务条目，这样的系统框架就相对稳定，虽然起初要多花一些精力来建立这样的框架，但框架建立之后，可以节约大量的系统维护时间和维护费用，一个良好的设计对应用系统建设的意义就体现出来了。

如果是规则简单，而且不会变化（这个条件要再三好好考虑），也可以用一段简单的条件语句来实现这些处理规则，当然，硬编码也可以做很多事，可以更简单直接，是一种快而粗略的方式，它的缺点就是面对变化时比较无奈，可能引起的维护成本非常高。

考虑到现实世界中的多种事件类型，以及各种商业服务的协议，我们建立以下一个相对通用的计费系统模型（假设是一个电力计费系统），其中就包括 Posting Rules（处理规则），用来封装计费系统中不同的规则，同时事件可能包括特定服务，滞纳金产生，设备安装维护等，但我们先来关注使用（Usage）事件，就是“用电/电的使用”：



在这个图里，合同（Service Agreement）充当了所有者（Host）的角色，并由 EventType（事件类型）和 date（时间）来限定一个具体的处理规则，合同属于客户（Customer），事件包括事件类型，并与特定的客户相关，最后，由 PostingRule（规则）处理相应类型的事件，并产生对应的帐务条目（Entry）。

根据上面的模型，下面首先来看一个简单的事件和事件类型的样例代码，其中，事件包括一个哈希表容器，来容纳它将产生的帐务条目：

```
class AccountingEvent
{
    private EventType _type;
    private MfDate _whenOccurred;
    private MfDate _whenNoticed;
    private Customer _customer;
    private Set resultingEntries = new HashSet();
//构造函数
    AccountingEvent (EventType type, MfDate whenOccurred,
                        MfDate whenNoticed, Customer customer)
    {
        this.type = type;
        this.whenOccurred = whenOccurred;
        this.whenNoticed = whenNoticed;
        this.customer = customer;
    }
```

```
Customer getCustomer()
{
    return customer;
}
EventType getEventType()
{
    return type;
}
MfDate getWhenNoticed()
{
    return whenNoticed;
}
MfDate getWhenOccurred()
{
    return whenOccurred;
}
void addResultingEntry (Entry arg)
{
    resultingEntries.add(arg);
}
PostingRule findRule() { /*查找处理规则，将留到后面讨论*/}
```

```
        void process() {/*处理事件，将留到后面讨论*/}
}
class EventType extends NamedObject
{
        public static EventType USAGE = new EventType("usage");
//使用：这里用字符串代表类型，实际情况中会复杂很多
        public static EventType SERVICE_CALL = new EventType("service call");
        public EventType (String name)
        {
                super(name);
        }
}
```

下面是帐务条目的代码，根据我们前面的讨论，有以下属性和方法：

```
class Entry
{
        private MfDate date;
        private EntryType type;
        private Money amount;
        public Entry (Money amount, MfDate date, EntryType type)
        {
                this.amount = amount;
                this.date = date;
```

```
        this.type = type;
    }
    public Money getAmount()
    {
        return amount;
    }
    public MfDate getDate()
    {
        return date;
    }
    public EntryType getType()
    {
        return type;
    }
}
```

帐务条目类型，跟前面的事件类型类似：

```
class EntryType extends NamedObject
{
    static EntryType BASE_USAGE = new EntryType("Base Usage");
    static EntryType SERVICE = new EntryType("Service Fee");
    public EntryType(String name)
```

```
        {
            super(name);
        }
    }
```

客户/顾客的简单定义，它拥有一个合同对象（ServiceAgreement），并用一个列表（List）作为该客户的帐务条目容器：

```
    class Customer extends NamedObject
    {
        private ServiceAgreement serviceAgreement;
        private List entries = new ArrayList();
        Customer (String name)
        {
            super(name);
        }
        //为客户增加帐务条目
        public void addEntry (Entry arg)
        {
            entries.add(arg);
        }
```

```
        public List getEntries()
        {
            return Collections.unmodifiableList(entries);
        }
        public ServiceAgreement getServiceAgreement()
        {
            return serviceAgreement;
        }
        public void setServiceAgreement(ServiceAgreement arg)
        {
            serviceAgreement = arg;
        }
}
```

在合同中定义了一个 Map，来充当 PostingRule 的容器，这个 Map 用事件类型作为主键（Key），而每一个规则对象根据不同的事件类型还拥有一个带有效时间段的规则容器，用来保存该规则一个时间段系列的规则具体设置，合同还拥有一个比例属性，意味着不同的合同规定了不同的计算比例，需要注意的是，复杂的时候连比例都要分离到不同的规则对象中去，并让它成为比例规则：

```
class ServiceAgreement
{
        private double rate;
```

```
private Map postingRules = new HashMap();
void addPostingRule (EventType eventType, PostingRule rule, MfDate date)
{
    if (postingRules.get(eventType) == null)
    {
        postingRules.put(eventType, new TemporalCollection());
    }
    //将规则插入已有相应事件类型的有效时间段容器中
    temporalCollection(eventType).put(date, rule);
}
//下面根据事件类型和发生时间获得相应的规则，其中的时间参数被送到有效时间//段容器，由它来查找得到对应时间的规则：
PostingRule getPostingRule(EventType eventType, MfDate when)
{
    return (PostingRule) temporalCollection(eventType).get(when);
}
    //跟据事件类型不同，返回该事件类型的规则容器
private TemporalCollection temporalCollection(EventType eventType)
{
    TemporalCollection result = (TemporalCollection) postingRules.get(eventType);
    Assert.notNull(result);
```

```
        return result;
    }
    public double getRate()
    {
        return rate;
    }
    public void setRate(double newRate)
    {
        rate = newRate;
    }
}
```

下面是抽象类处理规则，包括它将创建的帐务条目类型，公开的事件处理和私有的创建条目的方法，还有一个抽象的保护方法，用来根据事件计算数值。

```
abstract class PostingRule
{
    protected EntryType type;
    protected PostingRule (EntryType type)
    {
        this.type = type;
    }
    //创建帐务条目（通过事件获得客户，把条目分别增加到客户条目和事件对应的条//目容器中）
```

```
    private void makeEntry(AccountingEvent evt, Money amount)
    {
        Entry newEntry = new Entry (amount, evt.getWhenNoticed(), type);
        evt.getCustomer().addEntry(newEntry);
        evt.addResultingEntry(newEntry);
    }
    //处理过程就是为了创建条目
    public void process (AccountingEvent evt)
    {
        makeEntry(evt, calculateAmount(evt));
    }
    abstract protected Money calculateAmount(AccountingEvent evt);
}
```

上面的几个类定义了整个电力计费系统的框架，下面我们来看看一个实际的系统需要定义的具体类，首先我们想起前面提到要处理的事件，叫 Usage（用电），表示客户用电的数量，单位是度（千瓦时），那么来看看这个具体的事件类定义，它继承了前面定义的事件类，并在构造函数里调用了父类的构造函数，增加了一个表示数量的属性，并增加了一个获取比例的方法 getRate()，这个方法返回该事件对应客户的合同的比例：

```
public class Usage extends AccountingEvent
{
    private Quantity amount;
    public Usage(Quantity amount, MfDate whenOccurred, MfDate whenNoticed, Customer customer)
```

```
    {
        super(EventType.USAGE, whenOccurred, whenNoticed, customer);
        this.amount = amount;
    }
    public mf.Quantity getAmount()
    {
        return amount;
    }
    double getRate()
    {
        return getCustomer().getServiceAgreement().getRate();
    }
}
```

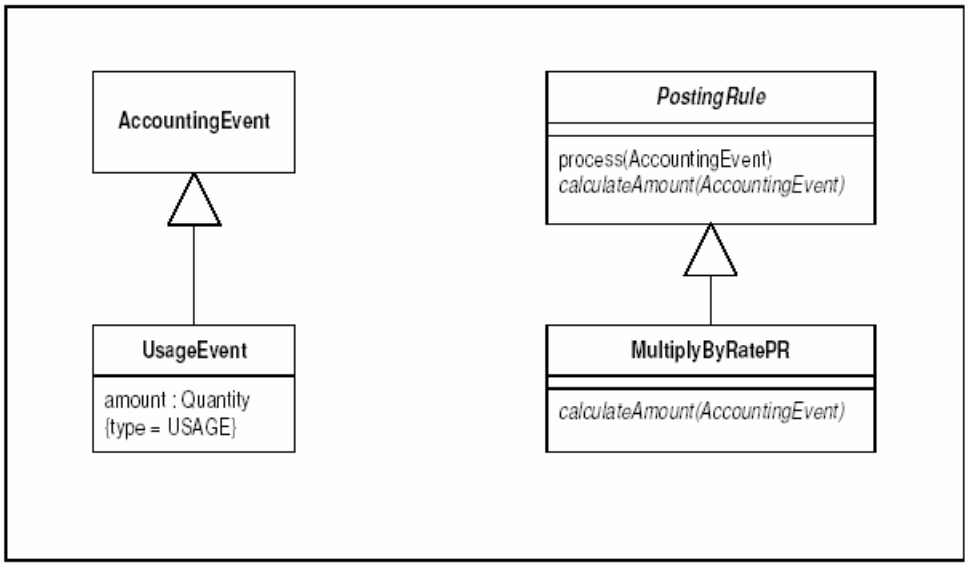
在这里用子类来表示具体的事件，这也是相当通用的方法，因为前面介绍的实在是一个相当抽象的事件接口，为了对具体的业务建模，通常需要对原来定义的事件进行扩展，在上面为了简单起见，把事件类型写在构造函数中，而在实际系统中，通常需要把它当作参数传进去。

然后我们建立具体的处理规则，假设我们的规则是把用电的数量乘上一个比例，并把它的名字叫做MultiplyByRatePR，它沿用了抽象父类的结构和方法，变化之处仅仅是用自己的计算方法实现了父类定义的抽象计算接口 calculateAmount()，代码如下：

```
class MultiplyByRatePR extends PostingRule
{
    public MultiplyByRatePR (EntryType type)
```

```
    {
        super(type);
    }
    protected Money calculateAmount(AccountingEvent evt)
    {
        Usage usageEvent = (Usage) evt;
        return Money.dollars(usageEvent.getAmount().getAmount() * usageEvent.getRate());
    }
}
```

上面两个类的继承关系如图所示：

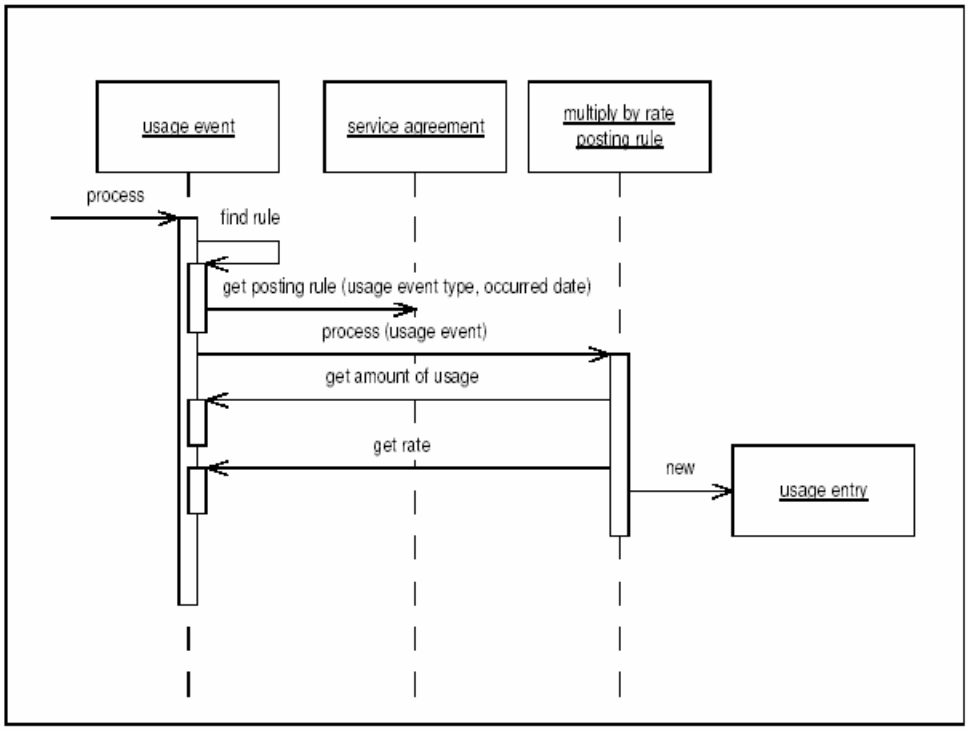


有了具体事件和具体处理规则，怎样用定义好的规则来处理它呢？记得我们在父类事件中定义了一个process()的方法，还有一个 findRule()的方法，当时我们没有讨论它们的具体实现，现在就来看看它们是怎么组织的：

```
class AccountingEvent
{
    public void process()
    {
        findRule().process(this);
    }
    PostingRule findRule()
    {
        PostingRule rule =
        customer.getServiceAgreement().getPostingRule(this.getEventType(),
        this.whenOccurred);
        Assert.notNull("missing posting rule", rule);
        return rule;
    }
...
}
```

process()方法是先调用私有方法 findRule，然后用找到的 rule 对自己进行处理（主要是产生 rule 规定的帐务条目）；而 findRule 依然利用事件跟客户的关联，得到客户对应的合同，然后用自己的时间和时间类型作为参数，从合同类的规则容器中得到处理规则，然后返回。

这个过程的序列图如下所示：



下面来看看怎样使用我们已经定义好的这些类来计算电费，建立一个客户对象，建立一个合同对象，设置合同，事件类型，规则，有效时间：

```
public void setUpRegular ()

{

    acm = new Customer("Acme Coffee Makers");

    ServiceAgreement standard = new ServiceAgreement();

    standard.setRate(10);

    standard.addPostingRule( EventType.USAGE,

                            new MultiplyByRatePR(EntryType.BASE_USAGE),

                            new MfDate(2002, 10, 1));

    acm.setServiceAgreement(standard);

}
```

...

建立简单的测试，事件，处理，并判断最后结果是否正确：

```
public void testUsage()
{
    Usage evt = new Usage(Unit.KWH.amount(50), new MfDate(2002, 10, 1),
                          new MfDate(2002, 10, 1), acm);
    evt.process();
    Entry resultingEntry = getEntry(acm, 0);
    assertEquals (Money.dollars(500), resultingEntry.getAmount());
}
```

好，一个电力计费系统现在可以运作了，我们为了它花了不少心思，但也可能有人想，不就是一个乘法吗，本来我几行代码就写好了，最多再加点对时间的判断，这下可好，建立了一堆类，一堆方法，把简单的问题复杂化了。不过这是一种巧妙的设计，它的意义在于易于扩展，易于应付各种变化，是一种柔性的设计。

不信的话，我们来看它将面对的变化——第二种事件类型，假设我们认为它是一个服务请求，服务部门提供服务（例如线路维修，电器安装等）以后，将产生一笔服务费，具体产生多少数额的服务费当然是由客户跟电力公司的合同规定的，在这里我们先假设计费规则是一半基本服务费再加上 10 美元，好，马上来看新的事件类型，新的事件和新的规则：

```
class MonetaryEvent extends AccountingEvent
{
    Money amount;
    MonetaryEvent(Money amount, EventType type, mf.MfDate whenOccurred,
    mf.MfDate whenNoticed, Customer customer)
```

```
    {
        super(type, whenOccurred, whenNoticed, customer);
        this.amount = amount;
    }
    public mf.Money getAmount()
    {
        return amount;
    }
}
```

这个事件继承了 AccountingEvent，只有一个简单的金钱数目作为新增的属性。可以先写一段测试程序来测试这个事件：

```
public void testService()
{
    AccountingEvent evt = new MonetaryEvent(
                                Money.dollars(40),
                                EventType.SERVICE_CALL,
                                new MfDate(2002, 10, 5),
                                new MfDate(2002, 10, 5),
                                acm);
    evt.process();
    Entry resultingEntry = (Entry) acm.getEntries().get(0);
    assertEquals (Money.dollars(30), resultingEntry.getAmount());
```

```
}
```

接下来提供新的处理规则，从 PostingRule 派生，它增加了乘法系数属性，和一个固定费用属性，并重新实现了 calculateAmount(AccountingEvent evt)方法：

```
class AmountFormulaPR extends PostingRule
{
    private double multiplier;
    private Money fixedFee;
    AmountFormulaPR (double multiplier, Money fixedFee, EntryType type)
    {
        super (type);
        this.multiplier = multiplier;
        this.fixedFee = fixedFee;
    }
    protected Money calculateAmount(AccountingEvent evt)
    {
        Money eventAmount = ((MonetaryEvent) evt).getAmount();
        return (Money) eventAmount.multiply(multiplier).add(fixedFee);
    }
}
```

下面用新的事件，规则建立计算测试代码：

```
public void setUpRegular ()
{
    acm = new Customer("Acme Coffee Makers");
    ServiceAgreement standard = new ServiceAgreement();
    standard.setRate(10);
    standard.addPostingRule(
    EventType.USAGE,
    new MultiplyByRatePR(EntryType.BASE_USAGE));
    standard.addPostingRule(
    EventType.SERVICE_CALL,
    new AmountFormulaPR(0.5, Money.dollars (10), EntryType.SERVICE));
    acm.setServiceAgreement(standard);
}
```

第二种合同

对已经存在的合同，很容易增加新的处理规则（Posting Rule），同样也很容易增加新的合同种类，假设针对某些家庭用户（电力公司对工业用户，商业用户和各种家庭用户的计费规则可能会不同），新的合同规则是：在某个给定的用电量范围内（假设 50 度），电费价格是原来的价格（原来的价格定义在合同类中），超出这个范围的话，将用另一个价格（rate）计算。

看下面的一个新规则类定义，增加了一个新的 usageLimit 属性和 rate 属性：

```
class SomeFamiliesCapPR extends PostingRule
```

```
{
    double rate;
    Quantity usageLimit;
    SomeFamiliesCapPR (EntryType type, double rate, Quantity usageLimit)
    {
        super(type);
        this.rate = rate;
        this.usageLimit = usageLimit;
    }
```

在这里，计算方法不同了：

```
    protected Money calculateAmount(AccountingEvent evt)
    {
        Usage usageEvent = (Usage) evt;
        Quantity amountUsed = usageEvent.getAmount();
        Money amount;
        return (amountUsed.isGreaterThan(usageLimit)) ?
        Money.dollars(amountUsed.getAmount() * usageEvent.getRate()):
        Money.dollars(amountUsed.getAmount() * this.rate);
    }
}
```

下面建立客户，新的合同对象，为合同对象增加规则，然后将合同对象与客户对象联系起来。

```
private void setUpSomeFamilies ()
{
    reggie = new Customer("Reginald Perrin");
    ServiceAgreement SomeFamilies = new ServiceAgreement();
    SomeFamilies.setRate(10);
    SomeFamilies.addPostingRule(
            EventType.USAGE,
            new SomeFamiliesCapPR(EntryType.BASE_USAGE, 5,
                new Quantity(50, Unit.KWH)));
            SomeFamilies.addPostingRule(EventType.SERVICE_CALL,
                new AmountFormulaPR(0, Money.dollars (10), EntryType.SERVICE));
    reggie.setServiceAgreement(SomeFamilies);
}
```

下面是一个小小的测试方法，首先建立两个不同的使用事件，接着分别计算结果，然后分别检验计算结果是否正确：

```
public void test SomeFamiliesUsage()
{
    Usage evt = new Usage(
        Unit.KWH.amount(50),
        new MfDate(2002, 10, 1),
        new MfDate(2002, 10, 1),
        reggie);
```

```
        evt.process();
        Usage evt2 = new Usage(
                Unit.KWH.amount(51),
                new MfDate(2002, 11, 1),
                new MfDate(2002, 11, 1),
                reggie);
        evt2.process();
        Entry resultingEntry1 = (Entry) reggie.getEntries().get(0);
        assertEquals (Money.dollars(250), resultingEntry1.getAmount());
        Entry resultingEntry2 = (Entry) reggie.getEntries().get(1);
        assertEquals (Money.dollars(510), resultingEntry2.getAmount());
    }
```

有一个小问题，新合同的规则中规定了一个价格，而另一个价格放在合同对象中，这样处理是因为合同中的价格更具有通用性，如果想把它们都放到规则中也可以。

因此，每次出现新的规则时，我们都可以使用一个新的规则（PostingRules）子类来完成任务。规则是各种各样的，而且经常会变化，所以用子类实现不同的规则是较好的方案，也可以实现一些比较通用的规则，但刚开始没有也没关系，因为害怕会变化就提前设计好规则，有时候会白白增加劳动量，毕竟，变化是未来才出现的，也是未知的，所以，等它出现的时候再面对它就可以了。当然，遇到重复的情况别忘了抽取它们的共性。

敏捷软件开发

敏捷软件开发系列

Cockburn • Highsmith 丛书编辑

Agile Software Development

Alistair Cockburn 著

[illegible] 译

[illegible] 审校

CHINA-PUB.COM

《企业应用架构模式》评论

李巍 整理

查看评论

Charles Ashbacher

Flower没有给出关于企业应用（Enterprise Application）的精确定义，而是更愿意列举出它们大多数所共有的一组特性。一般而言，它们都是非常大型的系统，具有许多的用户界面屏幕，用来进行并发访问和海量数据的更新。在几乎所有情形中，必须要对数据进行持久化，事实上其往往具有非常强的持久性，这意味着它必须能在整个软件迭代，操作系统更换，硬件变更，以及人员和程序员流动的过程中留存下来。

此外，企业应用通常必须要与其他应用进行通信，这些应用往往同样大型和复杂。例如工资和病例，信用卡处理，保险理赔处理，银行，以及外汇贸易。简而言之，就是大多数驾驭现代全球经济体系的程序，它们许多都是当前所使用的最复杂的软件项目。最后，所构造的程序必须让「那些并非是它们创建者的人们」能够“容易和快速地”对其进行改变，以适应快速变化的条件，并且通常无需程序员任何输入。由于投入如此之多，必定会存在一组最佳的实践，这便是本书所要阐述的内容。

Fowler所论述的软件构造模式通常都比较小，它们描述的是专门的操作，而不是演示一个大的构架形态。每个具体模式的提出方式是，首先用一句话的描述来列出该模式的意图，以及用一个UML图演示它的结构。接下来的部分便描述了模式如何进行工作，何时使用它，以及一个或多个「使用源代码的框架来演示该模式具体实现」的示例。C#和Java都会在示范中被用到，它们不会造成理解上的问题。语言和上下文（context）如此类似，以至于任何人都能理解其中的任何一种，将不会在代码的阅读和理解上有任何的问题。

这是在封面上列出的五十一个模式中的一些示例：

- Lazy load－在这里对象将只装载当前需要的数据，但是要对「需要的所有其他数据的链接（link）」进行维护。
- Front controller－一个单独的处理者（handler）对象，其负责合并（consolidate）所有针对一个web站点的请求。然后它将这些请求发送给专门的服务对象（如安全，国际化问题，以及针对特定用户和位置的具体显示）。
- Optimistic offline lock－用来防止在执行并发业务事务时发生冲突。解决方案是当冲突被检测出时对事务进行回滚。

■ Server session state—将会话的数据以串行化（serialized）的形式保存在服务器上。

尽管这些示例的必要性通常都极为简单，但它们确实展示了针对通用软件开发问题的一些最有效的和业已测试的解决方案。因此，这是一本任何企业应用的软件构造者都不应该没有的书。很难相信会构造出一个没有涉及到解决许多本书所列的这些问题的企业应用。

我唯一要抱怨的就是偶尔会出现一些不好的英文。例如，第100页上的短语，“The only reason I’ve concentrating on Java . . .”，以及第119上出现的短语“One factor that comes into this is comfortable used the development team is . . .”。当然没有一本书不会出现错误，但这种错误如此频繁，不得不让人怀疑最终编辑工作的质量。

没有什么是比「构建永久运行IT相当事物（IT equivalent）的高效软件」更难的事情了。这便是企业应用所设想的，如果你的目标是致力于构建一个企业应用的话，那么让你自己走到正确的轨道上来，并研读本书。你以及其他每个与软件打交道的人将会获得一份更好的经验并作为回报。

Joel Semeniuk

无论什么技术平台，构建企业应用都不是一件容易的工作。当这样去做时，方案架构设计师和开发者需要召集行业中其他人所积累下来的经验。这些设计和实现的最佳实践通常会被表示成软件的设计模式。作为当今面向对象软件开发的一位主要的权威人物，Martin Fowler抓住了这许多模式的本质，并将它们写到他的*Patterns of Enterprise Application Architecture*书中。

Martin Fowler这本书是为致力于设计和构建企业应用的软件架构设计师、设计者和程序员而写。这本书的主要目的是要建立一个工作词汇表（working vocabulary）以及通用企业模式的定义集，以希望读者在开发企业解决方案时能够做出更有见地的决定。注意，本书所表示的模式并不针对任何具体的平台或技术，并且提供Java和C#两种示例，这一点十分重要。但是，作者并没有对「像J2EE和Microsoft .NET这样不同软件开发平台可能更适用的模式」提供一些基本的指导。

如果你参与一个企业应用的设计和实现，便会有诸多「在你的参考书库中应该有这本书一席之地」的理由。尽管我认为它并没有对设计模式的“总体轮廓（big picture）”给出足够的论述，但本书依然是一本组织良好的精品佳作。作者很好地运用了形式化、模型化以及叙述性的混合手法，来定义和描述每个模式的上下文（context）。在书的第二部分会对本书前几章所泛泛描述的模式给出详细的说明，使其成为极佳的参考指导。正因如此，就请步入企业软件模式的世界，本书将带你即刻启航。

评论

当我在他（Fowler）的站点上看到这本书时，我便阅读了其大部分“进行中”的版本，并完全喜欢上了它— 那时我才刚刚开始接触设计模式，因此无法从一个“富有经验”的角度来评论它。尽管如此，基于我所能够记起的（以及现在大幅提高的设计模式的知识），我认为它十分值得推荐。

评论

我现在正在阅读这本书，而且我必须要说的是它让我非常高兴。我想向软件工程领域的每个人高度推荐它（尤其是你正在为每个人谈论的所有这些模式疑惑时）。令人高兴的是为许多我已经实现的模式取了名称（我只是不知道它而已），并且也从中获得了一些好的想法。

Sam Gentile

非常值得一读。我和John Lam在Win-Dev上进行了长时间的讨论，我们认为这是一本具有重要地位的书，或许更甚于GOF一书。GOF这本书在范围上其实比较小，关注软件的构造。Martin的书则将那些更加重要的素材（如何与数据库进行沟通，可伸缩架构等模式）分门别类。非常之好。

Robert Hurlbut

我同意这是一本非常好的书。我一直关注着Martin在其站点上所写的内容，直至最终出版（这时他拿下了所有文章并将它们替换成链接到其书的页码）。绝对是一本“经典”之作！

Phil

的确是一本非常好的书。我仍然没有看到一种模式可以表示复杂的多分支条件（multi-branch conditional logic）逻辑：

if this then

```
{
if that or theother then
{}
else if here then {}
else if now then {}
else {}
}
```

评论

这是我所发现的关于 J2EE 和.Net 模式的最佳图书。我想它注定会成为经典。我发现关于「何时进行分布式，Unit Of Work，Domain Model 和 Data Mapper 模式」的论述极为有用。它改变了我考虑企业应用的方式。

就其范围而言，我想它处于原先 Gamma 等人所著的“设计模式”与一本像“J2EE 模式”书之间的位置。“设计模式”描述了能够适用于任何类型应用的既有模式。“J2EE 模式”则从一个平台的角度来描述模式（尽管它们许多也同样适用于其他的平台）。Fowler 的书描述了一组能够工作于某种类型应用（业务应用）的模式，但它更适用于多种平台。

这本书要好于“J2EE 模式”，“J2EE 模式”这本书没有很好地论述哪些 J2EE 部分需要避免，以及哪些“模式”实际上是为平台本身的问题而工作的（例如，“Composite Entity”模式）。

我非常不同意第一位评论者的观点。Fowler 确实论述了「模式所工作的最佳平台」。书的第一部分给出了一个很好的线路图（road map），以确定能够在你的应用中使用哪组模式。他明确地提到.Net 会引导你进入 Table Module 的方向，而使用 J2EE 你可能不会使用这个模式。

就框架用到的模式而言，我依然发现「了解框架所实现的模式」是非常有用的。这样你便会知道需要选择哪些框架。我们最近正在进行 O/R mapping 工具的选择过程。阅读 Unit Of Work，Data Mapper，Repository，Lazy Load 和 Identity Map 章节，将对这个过程*非常*有帮助。同样，阅读 Front Controller 模式则给了我一些关于「如何最好地利用 Struts 框架」的新思路。我完全不同意“学习与这些框架相关的模式将不会带来很多价值”的观点。在这里，无知绝不是福。

最后，关于「本书“只是”收集和命名某些已经存在的模式，从而降低了它的价值」的观点同样无聊。许多开发者发现这些业经实践的真正模式是非常有用的。对通用的模式命名并进行清晰的描述是非常有帮助的。这正是最初“设计模式”一书所要做的内容。按照这个逻辑，我猜想最初的那位评论者可能只给了“设计模式”三颗星。

这是一本伟大的书。

评论

这本书给出了基于层式架构（他没有涉及过滤器/管道模式）企业解决方案的模式目录。它为许多企业相关问题提供了多种可能的解决方案（使用C#和Java）。基于J2EE/.NET/CORBA框架的应用都会存在这些大多数问题。如果使用者已经阅读了J2EE设计模式，设计模式（Gang of Four）的书，并且在3-4个企业多层应用上工作过，那这本书便十分有意义。这些解决方案非常容易付诸实施，至少在基于J2EE应用服务器的解决方案中是这样。

顺便说一句，这本书在Amazon上已经脱销，我从Barnes & Noble上订购一本并且也获得了不错的折扣。

缺点：

- 没有随书附带的CD:-)
- 没有使用所有（高级）语言工具（使之易于阅读）

评论

这确实是一本关于构建企业应用的非常好的读物。我喜欢作者给出了当前最常用到的软件平台（如Java和.NET）的示例。不管怎样，作者对Java所给出的细节要比.NET更多，并且他论述的模式并不总是具有那样的通用性。并没有很好地涉及Table Module相关的事情。我有一个问题提请注意，任何愿意就这个事情提供帮助的本书读者，请与我联系。

总之，我喜欢阅读这本书，并且我相信它物有所值。

The Addison Wesley Signature Series

A MARTIN FOWLER SIGNATURE BOOK

PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE

中译本即将上市

MARTIN FOWLER

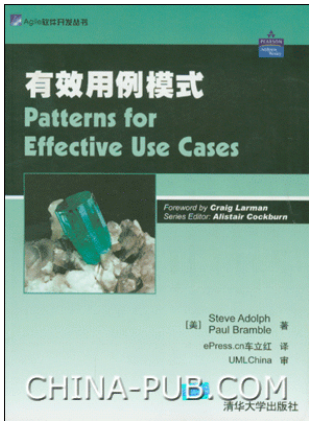
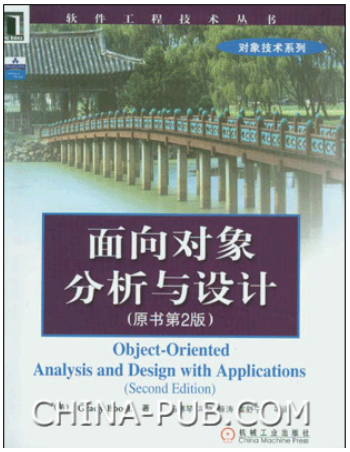
WITH CONTRIBUTIONS BY
DAVID RICE,
MATTHEW FOEMMEL,
EDWARD HIEATT,
ROBERT MEE, AND
RANDY STAFFORD

UMLChina 负责中译本最后审校，并指定为训练用教材

国内中文书对《人月神话》的引用精选（二）

UMLChina 整理

查看评论

书	作者	页码	书中评价原文
	Steve Adolph ，Paul Bramble	1	书中第一章“什么是高质量用例”第一句话就引用了“没有银弹”中的话：构造软件系统最困难的部分就是精确确定到底构建什么。
	Grady Booch	1	一颗濒于坍塌的行将死亡的恒星，一个正在识字中的小孩，一群对病毒发起攻击的白细胞：这些只不过是物质世界中少数涉及巨大复杂性的对象。软件也可能会牵扯到很复杂的元素，但这里所看到的复杂性却是根本不同的类型。正如 Brooks 指出的“爱因斯坦认为肯定有对自然的更简单的解释，因为上帝并不是反复无常和独断专行的，但这样的信念并不能安慰软件工程师。他所要对付的复杂性绝大多数都是任意性的”
		392	分类书目……采用以下格式： [<标号><年代>]。例如 Brooks[H 1975]是指 Brooks 于 1975 年所著的《人月神话》

	Craig Larman	381	参考文献中仅有的几本非面向对象书籍之一
	Alistair Cockburn	29	在标题“创造和沟通的游戏”中，引用了《没有银弹》文章。中文注释版专门作了注解。
		160	MYTHICAL MAN-MONTH REVISITED 的故事
	Scott W.Ambler	5	敏捷软件开发宣言……价值观……“个人和交互高于过程和工具”……这对管理层可能是难以接受的，因为他们经常情愿相信人和时间或者人和月是可以互换的。
待续….			

欢迎您提供您的信息。