X-Programmer 总第1期
软件工程的播种机
www.umlchina.com
非程序员 2001 5

千年决心

访谈
Dassault Aviation 为业务的成功实施面向对象技术和 UML

方法
用 Use Cases 捕获需求
《分析模式》的前言和介绍

过程
参与变革
创建成功的工程
更好地领导一个项目的诀窍

工具
选择一种 UML 建模工具

论坛
OO/UML：Use Case 正解
OO/UML：定时器在用例图里是不是一个角色?
工具：怎样把 Rose 中的 Use Case Diagram 导到 Word 文件中，作为一幅图片
工具：我可不可以不用 RO$E?(并非贬低 Rational)

服务
国内出版的软件工程书籍一览

征稿

详情请见：

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

来稿若有重复，编辑部将择优整合

出版：UMLCHINA
编辑：UMLCHINA
投稿：editor@umlchina.com
业务：info@umlchina.com
广告：adv@umlchina.com

千年决心

Scott Ambler，译：abug(abug@263.net)

在新千年的开始，让我们花点时间为自己的事业下点决心。

在 2000 年 1 月 1 号的零点。不管在 1999 如何如何，依然有电，水还在流，世界上的金融系统仍在加加减减的正常运行。这已经是新千年的开始，但你仍要作些新年誓言。是的，你可以发誓减肥或是更多的在外工作，但你是否可以持续一到两周？不如作些承诺改进做为软件专业人士的职业生涯吧。请仔细想想以下的一些决定:

我不妨碍他人。你不应只想玩玩，就使用技术。你用技术，因为这合情合理，或对解决在手边的问题有用。许多开发者仅仅为了获取经验而建议使用如 EJB（Enterprise Java Beans）或 COM+等流行的技术。你是在浪费你老板的时间与金钱来实现自己的目标。另外，不要为了揭示你的单位软件结构的弱点而引入病毒，或删除数据或程序。如果真的有弱点，通知管理层，不要造成任何损害。

我会适当的重用任何可重用的东西. 因为大家喜欢从零开始，而不是重用已有的成果，这些年，软件专业人士的生产率并没有显著的提高。有很多的软件成果可以被重用，如原代码、部件、文档、文档模板、模型甚至通过应用模式重用其他人的技术。只要可以，就去致力重用其他人好的工作成果，而不是假设你可以从零开始做事，并比其他人更好（不幸的是，这正是许多一般开发员的心态）。现在，重新发明“轮子”，并不是一件荣耀的事。

我只开发基于实际需求的软件。如果你没有需求，你不需要开发任何东西。无论什么类型的系统，你总可以从为它定义需求开始。人们或其他系统如何使用你的系统？它需要怎样好的执行？它必须具有什么样的使用特点？它必须在什么平台下运行？不管你的系统使用什么技术，它是什么样的业务类型，你总可以先确定它的需求。其他都是干咳。

我会在编码前先建模. 最有效率的开发员会首先建模，并只有在他们认为完全理解了要做什么了后，才会开始编码。你在建模上的精力或许简单如只是在餐巾纸上画几幅简图，或是复杂如使用业界领先的 CASE 工具画出一整套图形。其中的含义，非常简单：先想，后做（运筹帷幄）

你要能够辩明你的工作. 你为什么正在开发你的系统或相关部分？你是否知道它技术上的可行性？是否其他人做过该类原型，显示出你正在做正确的工作。你的软件在经济上有意义吗？它是否值得去做，是否可保持你的组织的竞争力或打开新的市场？一旦你开发的产品完成后，是否你的组织能够操作它？你是否有一些人可以操作并维护该系统？是否可以得到操作规程和文档？是否有支持计划？如果你不能辩明你的工作，为什么你还在做它呢？

我会停止重复教条. 给予项目组最大损害的是那些相信如：数据至上，编码最重要，或是我们是用例驱动等一些小教条的人。软件开发非常复杂，这些教条只是覆盖了对于过程的缺乏理解问题。实际上，数据只是整个过程的一个小部分；而为了使软件成功，需要的不仅仅使原代码；对描述需求来讲，仅有用例是不够的。教条只是在人们之间建立了阻碍，并减小了小组的成功机会。

我会从不同的角度来看工作. 不管你在项目中的角色，你应总是同时针对几个相关部件。业务分析员在调查问题域时可以开发用户界面、用例和领域模型。建模者在设计软件时可以开发顺序图、类模型、部件模型、状态图和数据存储模型。程序员在实现软件时可以编写代码、测试用例和编写原代码文档。只关心一个产品部分，或是项目进度、或是用户界面原型、或是原代码或数据模型，将经常导致发布结果达不到软件总体需求。

我不仅仅关注软件的执行. 成功开发不只是做出执行速度快的软件。根据项目的类型、级别不同,创建可扩展、可理解、可维护、可用或重用的软件也许更为重要。软件执行速度仅仅只是软件评价标准的一方面，但是太多的开发员只注重该项，而影响了他们整个的工作质量。

我会尊重客户并同他们紧密协作. 你开发软件的唯一理由是为了支持客户。只有客户能够告诉你他们需要什么，因为他们是业务领域的专家。为了保证你建立正确的系统，与客户紧密工作难道不是非常有道理吗？

我只接受符合实际的项目计划. 组织或市场的压力经常导致项目计划不符合实际。无论何时，只要使用“如果每件事都按我们估计的方式发展，而且我们足够幸运的化，我们就可以将项目作好”来阐明计划的话，那你就有麻烦了。事情不总是按你想象的方式进行，不管多少人力投入一个项目，许多软件的开发的不同工作都会耗费非常大量的时间来完成。记住，九个女人也不能在一个月里生一个孩子。如果你迫于压力，接受了一个不切实际的项目计划，那么你必须回去向老板阐明计划不实际的原因。你的理由也许不被接受，你还得执行该计划，但至少你为项目进行了争取。有不现实计划的项目组常常时采取了捷径，但却导致项目在远远落后于进度时，项目仍在停步不前。

我会持续改善我的沟通技巧. 如果你不能与其他人有效沟通，好点子又有什么用呢？你也许可以写出世界上最好的源程序，但是，你不能写一封 E-MAIL 来告诉你的老板与小组成员你所做的，那么你的杰作可能根本得不到承认。

我会养成学习的习惯. 在你躺在你的荣誉上时，软件工业的技术与技巧在飞速改变。试着每月读几本相关杂志或至少一本技术书籍。我曾经接受到的一些最好的建议是扩展视野，并读一些商业杂志。商业相关的阅读使你具有同用户联系的背景知识，它是成功的一个关键因素，因为软件是为支持用户的使用而开发的。参加与工作有关的课程或会议也应是你学习过程的一部分。

我要测试所有我开发的. 如果你可以构建，你也可以测试。你可以通过查看需求文档、设计、并执行多次测试来检测代码。如果一件事不值得测试的话，那么为什么还要做它呢？

我要文档化所有我开发的. 现代软件开发模式是你做为一个小组成员进行工作，如果没别人能够理解你的工作，那你就没有对工作做出任何贡献。很明显，好的文档能使你的工作容易被理解。实际上，简而言之，好的程序员在开始编码之前，会文档化他们的代码。

我认同软件不仅仅是技术. 技术是有趣的，但它只是开发软件过程中的很小一部分。不管是好是坏，你的工作需要你熟练地与用户、经理、同事、卖主或操作人员打交道。

我认为软件不仅仅是开发. 你必须牢牢记住，你的用户，或是你的上层经理，在你开始工作前，很可能已经花费了极大的精力来挑选和识别软件开发项目。一旦你将软件发布给用户，该软件必须要被维护、使用和支持。你需要全面理解如何成为一个有效率的专业人员。

你正在新前年的开始，一个独一无二的历史时期，几乎每个人都在迷惑如何看日历。你可以或是以头撞墙来向其他人解释，2000 年是千年的最后一年，不是新千年的第一年，或是放弃解释，将注意力转向改进你的软件开发技巧。

依从我这里的一条或几条建议将给你带来成功，同时需要指明，经常去去体育馆也是个不错的主意。

访　谈

Dassault Aviation 为业务的成功实施面向对象技术和 UML

Nasser Kettani

自：Rose Architect，Jun,1999. Think 译

Dassault Aviation 是行业领先的跨国军用和民用航空产品的制造商和复杂系统的开发和集成商，总公司设在法国。Dassault Aviation 以其第四代 Rafale 多用途战斗机和 Falcon 商务喷气式飞机闻名于世。Dassault Aviation 的子公司 Dassault Systemes 是著名的 CAD/CAM 系统 CATIA 的开发商。Dassault Aviation 成立于 50 年前，目前雇员超过 9,000 人，营业额 25.40 亿欧元(27.77 亿美元)。

Mathilde Thibault 和 Guillaume Tran Thanh 是软件支持部门的主管，他们非常乐意和大家分享他们引入面向对象技术和 UML 的经验。

NK：你们开发的项目属于哪种类型？

MT，GTT：我们的 MIS 系统基本上分为三种类型：ERP-based，TDMS-based（现用 Dassault Systemes 的新产品 Visual Product Model 来开发）及一些定制的项目。我们也做包括导弹关键信息系统，实施嵌入系统，模拟系统，基于 Web 的系统（Intranet，Extranet）。我们使用的编程语言包括 C，C++，Java 及 Ada。

NK：这是否意味着那么在这些项目中应用了 UML？

MT，GTT：是的。UML 是我们所有项目的共同标准。我们使用面向对象方法开发已经 3 年，最后决定使用 UML 作为我们的标准。我们要打破我们公司不同类型项目之间的屏障。我们已经在几个项目里使用了 UML，其中包括飞机制造方面的项目，在这方面我们与英国航空公司和 Integrated Modular Avionics 有合作。

NK：能为读者解释一下为什么你们要选择 UML？

MT，GTT：主要是三个原因。首先是统一，我们需要在我们的不同项目之间实施统一的软件工程方法以便管理；其次是国际化，我们是一家跨国公司，在系统集成领域和工业界有很多合作伙伴，包括英国航空公司；第三是标准化，作为一家工业公司在选择软件方法时不能太随意。

NK：你们希望选择 UML 能够带来什么？

MT，GTT：我们希望在以下方面能够得到提高。首先，我们希望能够提高软件交付的效率和质量，由于全公司使用同一种技术，互相之间的沟通得到了改善；另外，我们运用 Use Case 驱动，以体系结构为中心的开发流程提高了系统的健壮性，伸缩性和质量；最后，由于我们应用面向对象的编程语言，UML 能够自

动生成代码，这大大改进了我们开发流程的效率和可追溯性。

NK：你们的 Team，管理和合作伙伴如何适应改变？

MT，GTT：我们显著改善了和定义了全部业务需求的业务小组的沟通，他们对我们的效率感到很惊讶；与合作伙伴及咨询公司的沟通也得到了改善。

NK：你们对你们的开发小组进行培训吗？

MT，GTT：新项目开始之前，开发小组都要进行培训。另外，在开发项目时会得到我们的甚至是咨询公司的支持。

NK：为什么你们选择 Rational 的产品？

MT，GTT：Rational 的合作伙伴和多平台战略（Unix 和 Windows）是我们选择 Rational 的主要原因；另外，我们需要世界领先的工具，Rational 是领先者。

NK：Rational 的工具是如何帮助你们达到目标的？

MT，GTT：Rational Rose 贯穿软件开发的所有环节。通过 Rose 的 Open API，我们开发了自己的代码生成程序；以前我们开发过自己的文档生成器，但最终我们还是选择 SoDA，它真正为我们节省了时间，特别是最终发布产品时。我们把 Rose 和我们自己的软件配置管理系统成功集成，表明 Rose 的开放性。至于希望 Rose 以后会有什么新特性，就是对 UML 的更全面的支持和更好的模型检查功能。不过我们很喜欢新的注册机制，特别是 Floating licenses，减少了我们的支出。我们期望专门为航空电子和模拟系统设计的 Rational Rose RealTime 的出现。

方　法

用 Use Cases 捕获需求

Pete McBreen，译：苏康胜（cancan28@163.net）

概述

开发者们经常通过一些典型的情节去理解系统并知晓系统如何工作，不幸的是他们虽然努力地去做了这些工作却很少以一种有效的方式去说明，Use Cases 正是一种形式化捕获这些情节的技术。

仅管 Use Cases 在一本对象方面的书《*Object Oriented Software Engineering*》中有过定义，是跟那些对象结合在一起的，但这项技术实际上是独立于面向对象的，Use Cases 是既能捕获商业处理流程又能捕获系统需求的有效方法，并且它本身比较简单和容易掌握。

使需求有利于回顾

以正规形式捕获这些情节的原因是有利于用户和开发者进行回顾，这里有 2 点关于一些实用需求符号的明确标准要遵循：

1） 它必须让情节的发起者和回顾者都很容易理解

2） 它不需包括一些关于系统样式和内容的决策

实用的需求是评估设计和最终实现系统的客观需求。

对于这些需求来说，必须要做的是以一种可实现的并不受约束的方式去捕获风险承担者的需要和期望。

Use Cases 使需求有利于回顾

Use Cases 已经得到越来越广泛的应用，它与其它需求捕获技术相比，它成功的原因在于：

- Use Cases 把系统当作一个黑盒
- Use Case 使在需求中看到实现的决定变得更加容易

最后一点源于第一点的补充，一个 Use Case 没有指定任何这些需求相关的系统的内部结构，所以说，如果这个 Use Case 中陈述了“提交改变到定单数据库”、“显示结果到 Web 页面”等的话，那么内部结构是显而易见的，并造成对设计的潜在约束。

为什么这些需求不指定内部结构的原因是，说明的内部结构给设计者带来了额外的约束，没有这些约束设计者们能更自由地建立一个正确实现客观可见行为的系统，并存在出现突破方案的可能性。

Use Cases 的工业接受

Use Cases 第一次被正式的描述是在 6 年前（1992 年），从那以后它就被最主要的面向对象的方法采用，同时作为描述现在和将来的操作模式的有用技术被商业再生工程团体（*Business Process ReEngineering Community*）采用。

Use Cases 最近它们自己取得在 UML（*Unified Modeling Language*）中有效的地位和位置而得到了突出的进步。UML 已经被 OMG（对象管理组织）计划做为对象系统的标准语言。

什么是 Use Cases?

Use Cases 本身是用户或其它系统与正在设计的系统的一个交互，是为了达到一个目标。术语 Actor（*行为者*）是用来描述有这个目标的人或系统，这个术语是用来强调有这个目标的任何人或系统。目标的本身是用行为动词开始的短语表达的，如：“Customer : place order”、“Clerk : reorder stock”等。

情节的角色

在 Use Cases 中不同的情节显示了目标怎样成功或失败，成功的情节是目标达到了，失败的情节是目标没有达到。这个的好处是因为目标总结了系统的各种使用的意图，用户能看到被设想怎样地使用这个系统，并且不必等到出现第一个原型或是比较糟糕的等到系统被开发出来才发现什么时候系统并不全面地支持他们的目标。

Use Cases 将做成多大？

试图决定 Use Case 的大小是一个很有趣的话题，处理这件事的一个方法是将 Use Case 的大小跟它的意图和范围关联起来，对于一个真正大的范围来说，一个 Use Case 并不要在一个系统中处理那么多，但这些系统都用于同一商业领域，称为 **Business Use Case**，它把整个公司看作一个黑盒和 Actor 关于公司目标的说明。这些 **Business Use Case** 的情节不容许假定任何公司内部的结构，一个客户将向公司下一个定单而不是客户服务部门。

对于系统发展而言，Use Case 的范围限制一个单一的系统，这是 Use Cases 最通常的形式，我们称之为 **System Use Case**，它把整个系统看作是一个黑盒，它不指定任何内部结构并且仅受限于问题域的语言描述。

Use Cases 的另一范围是设计子系统和系统内部组件的，称为 **Implementation Use Cases**，它把组件看作一个黑盒，并且这些 Actors 是区分它的成员。例如：可能会用 Implementation Use Cases 去说明应用系统中 email 组件的需求。

给出了这些分类，关于 Use Case 的大小话题变得容易了，设计这些项的范围来调整整个大小。帮助系统设计者，每个 Use Case 只描述没有大的分支的行为的单个线索。违背这个规定，Use Case 看起来通常是不准确的或含糊的，作为测试说明的资源和参考，它也是很难使用的。

看看 System Use Case 的例子，“从数据库中查询低库存的”它太小了，容易跟需求的实现细节混淆。对比一下，作为 System Use Case，“仓库管理”就太大了，它不能实现作为没有大的分支的行为的单一线索的原则，并且从系统的观点来说，它很难说明成功的目标，但它可以做为一个好的 Business Use Case，对于配件部门来说，它可以定义“仓库管理”这一成功的目标（可能是根据库存调整、配件验收、成本操作等）。

这些 Business Use Case 的好处是它们能用于区分其它的 Use Cases，就如：“仓库管理”可用于组织这些用于实际管理仓库的 Use Cases。

Use Cases 的正式定义

Use Case：特殊行为者（Actor）的价值的量化结果的提交

正如前面所说的，Actors 可以是人也可以是正在设计的系统与之交互的外部系统，一个 Use Case 要求有一个量化的结果，从单个线索的需要提交。做为量化结果的组成，目标要么成功要么失败，没有其它的情况。

达到主要 Actor 的目标定义为成功，结果没有迎合主要 Actor 的目标定义为失败，不同的情节显示了成功或失败的途径。

Use Cases 的说明

Use Cases 的好处是一些情节能用不同程度的正规化的文字说明。每个情节涉及 Use Cases 中单一的途径，细节是条件组。

不正规的文本描述也能使用，不过当条件较多和可能失败的情况下它们很难跟随下去。开始试图理解需求时，不正规的叙述风格也是非常有用的，然而随着 Use Cases 的进展，使用更加正规的机制去说明 Use Cases 才是有用的。

下面是客户对 Use Case“下定单”的粗略概略：

“确定客户，找出需要的并且仓库里还有的物品并检查客户信用额是否够用”

结构化叙述的格式已经被证明是非常有效的。这个格式所做的事是描述每一个情节的行为者：目标语句对的顺序。在这个顺序中，每一个行为者：目标的语句对都假设前一个的目标是成功的，右面是一个简单的范例：

1、 职员：确定客户
2、 系统：确定产品项
3、 系统：确认数量
4、 系统：确认信用额
5、 客户：授权付款
6、 系统：派遣定单

延伸部分

1a、客户没有存在
1a1、职员：新增客户
3a、库存不够
3a1、职员：商议数量

Use Cases 认为我们正在设计的系统是一个单一的黑盒，根本没有任何内部结构被记录下来，并且它被认为是一个情节产生的目的及对应单一的行为者（Actor）。这些 Use Cases 没有表示任何关于系统内部的东东，只是表示系统将达到什么样的目标及由什么（人或其它系统）操作和负责。

处理目标失败-----延伸部分

下面要做的是确定以上的每一步中可能会发生的失败，对于这种情况那些可能造成失败的条件做为延伸部分来捕获。这些扩展是通过在失败条件下直到可以重新入轨或失败的情节来处理的。

失败条件的分离使情节变得可读了，一般情况都是以最简单的途径通过 Use Cases 的，它的每一步，行为者（Actor）的目标都是成功的。分开列出所有可能造成失败的条件给予了更好的品质保证。回顾者可以很容易的检查是否所有造成失败的条件都被指定及哪些潜在的条件被忽略。失败的情节要么可以恢复要么不可以恢复，可恢复的情节最终取得成功，不可恢复的情节直接失败。

失败中的失败

当失败情节下还有其它失败发生，需特别的标示出来，也就是说，在延伸部分中用更长的前缀标记更深一层的失败情节，如：1a1b、客户是个不讲信用的，它的恢复通过 1a1b1。

为什么使用结构化叙述格式

结构化叙述的价值在于它是可驳倒的描述，所谓可驳倒的描述是它足够明确，以至可以给人去争辩和提出异议。

“流程不是那样的”

“为什么开单之前不检查是否有用呢？”

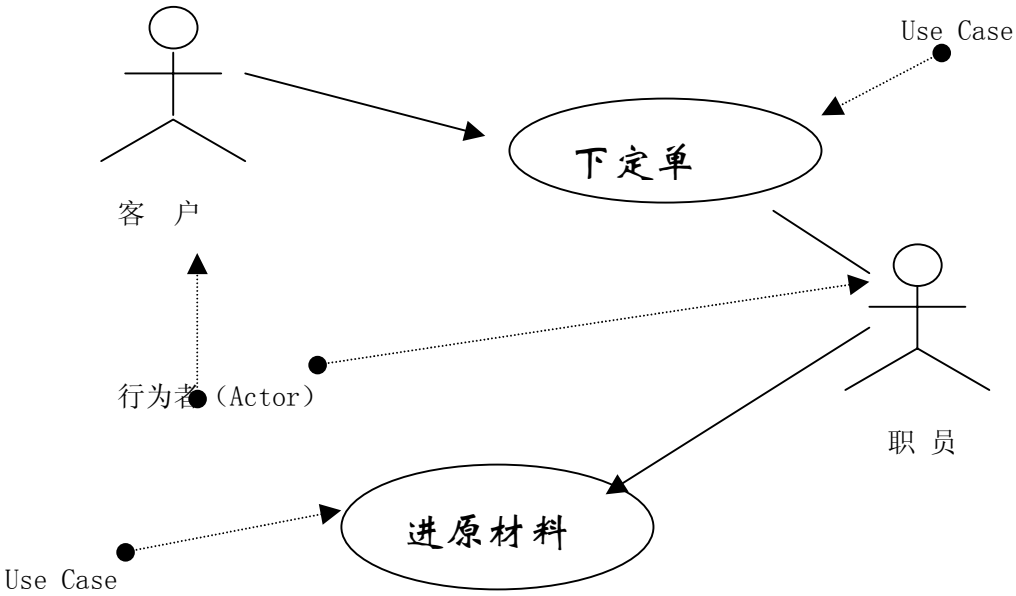
“你少了好几步哦”

对比之下，那种用非正规叙述形式描述的粗略概略是很难去驳倒的，但它有利于早期对问题域的理解。

可驳倒的描述方式的价值的另一种描绘是，当你说明 Use Cases 时期望从用户和开发者处获得关于 Use Cases 品质的反馈。自想在开发过程的中及早的得到修正，它是非常有价值的。用户典型的反馈是不同顺序的问题，可能是平行的或是缺少的步骤。开发者典型的反馈是关系到特殊失败条件的意思和如何检测到它是否清晰的要求。

Use Cases 的图形符号

这里是 Use Cases 的图形符号描述，UML 中一个单一的“Stick-Man”符号标示行为者（Actor），用椭圆标示 Use Cases，如图一所示。这些图对于你想看到 Use Cases 之间如何关联的“大图”和获得系统上下文的大体描述来说是非常重要的。



（图一）

Use Cases 图没有显示不同的情节，它们的意图是显示行为者和 Use Cases 之间的关系。所以 Use Cases 图需求用结构化叙述文本来补充。UML 提供一些可供选择的图来显示不同的情节，这些常规的图形有交互图、活动图、顺序图、状态图等。使用这些图的主要缺点是它们不象文本一样是紧密的，但它们能用于给出 Use Case 的整体感觉。对于这些图的协定的参考请见《*UML Distilled*》，作者 Martin Fowler。

需求重用取得

按行为者：目标（Actor：Goal）的格式来描述 Use Case 的作用是它容许公共的功能性分解出来做为独立的 Use Case。当执行公共部分的时候是指用于主要 Use Case 的。比如：Use Case 下定单中的“确定客户”这一步骤可以用于其他 Use Cases。

Use Cases 之间的另一关系是“延伸部分”，如果 Use Case 有一个失败恢复的步骤是复杂的，通常有三、四步，说是一个 Use Case 去扩展另一个 Use Case。比如：当没有可用库存时，“Issue Raincheck”可能扩展 Use Case“下定单”。

Use Cases 应用当中的复杂性和危险

主要行为者（Actor）和 Use Case 之间没有连结

一些情况下，从 Use Case 中取值的行为者（Actor）和积极参与这个 Use Case 的行为者（Actor）之间没有清晰的连结。如：财务主管能成为“发票确认”的行为者（Actor），但他未必是创建发票的人。这不是什么问题，这个 Use Case 仍然是正确的，它正说明行为者取值和设计的系统的范围外的 Use Case 发生的初始化之间的关系。主要行为者是有用的，因为这个人扮演的角色是当你说明 Use Case 时需要跟他说的人。

情节步骤不需要连续

情节中步骤顺序的情况是没问题的，这里有一些机制去突出可能的并行步骤。在 UML 中活动图是首选的机制，通过非正式地看 Use Case 的情节你可以注意到可能的平行步骤；可以看 Use Case 内一些邻近的

步骤；也可以有相同的行为者（Actor）对步骤负责。之前我们举过的例子里，确认数量和确认信用额可能是平行的。有时候在 Use Case 的说明文档中标记这些可能的平行步骤是有用的。

Use Cases 的大小

当开始做Use Cases的时候有个很显然的危险就是它要么有很多步骤要么就很少步骤。如果在Use Case中有超过 15 个步骤，它可能包含一些实现明细。如果它只有非常少的步骤则检查它的目标是否是达到一个没有很多分支的活动的单一线索。

较少的人类行为者（Actor）

如果 Use Case 有较少的人类行为者，而大多数行为者是其它系统，通常的做法是修改这个 Use Case。寻找系统必须做出反映或公认的事件胜过会见这些行为者。

需求捕获和系统复杂性

总而言之，这些情节捕获到系统复杂度的同时行为者：目标语句对容许大的系统以相对压缩的格式说明。Use Case 的格式的作用是用户和开发者能标志出行为者，然后确认这些行为者工作职责对应（或不对应）的目标，代替一个大的很难读的功能规格说明书。

仅仅这样，用户和开发者就有足够的兴趣进而研究那些情节的细节。

系统不仅仅有应得的功能性需求

一些 Use Cases 并没有捕获所有的客观需求，仅仅是捕获了系统怎么用的那些功能性需求。然而还有许多方面的需求需要去捕获的。其中有的非功能性需求使用关联以至于也能隶属于个别的 Use Case，如性能需求和系统容量的需求。另外的一些不是关联的而是要单独地去捕获，它们是以下的需求：

- 系统范围
- 系统用户的目标
- 用户界面原型
- 一般规则
- 约束
- 算法

运行时期和建立时期的需求比较

一个重要的因数要记住，就是系统的赞助者是大过用户团体的。系统中有许多的风险承担者，Use Cases 仅仅捕获其中一些风险承担者的需要，具体说，Use Cases 仅仅捕获系统运行时期的需求而忽略做为系统开发组织的风险承担者的需求，开发组织最有兴趣的是对建立时期需求的描述。

运行时期需求包括：系统范围、用户组织对产品的期望和目标、Use Cases、其它非功能性需求。

建立时期需求包括：减少开发成本、较少的变更处理、现存组件的重用。

建立时期的需求可以部分的由 Use Cases 把握。但许多方面是需要由开发组织的处理的。

- 项目范围和目标：项目必须提交什么。（和系统范围的区别是它提交的是所有项目的东西）
- 增长性和变更请求：这些可以在捕获为常规 Use Cases 格式中的“Change Cases”
- 开发负责人的约束：包括标准、习惯、工具、品质度量标准、品质保证原则、及品质保证的习惯。

Use Cases 的适用性

Use Cases 首先用于需要响应客观事件的系统。它们能用于提供了一个有很容易理解的目标的清楚的行为者的环境。当结果不可定义或不清晰时不能用 Use Cases。意思是如果目标成功或目标失败不能有一个明确的定义，那么 Use Cases 不能用来捕获需求。

然而说到这，现在大部分对象方法都使用 Use Cases。因为 Use Cases 被证明是捕获需求的非常有效的机制。

总结

Use Cases 以一种可读的、可驳倒的格式捕获需求。Use Cases 是系统客观必需机能的可驳倒的描述。可驳倒的意思是当你说明 Use Cases 时期望从用户和开发者处获得关于 Use Cases 品质的反馈。

Use Cases 并没有从一开始就就明确的定义，它主要的开发顺序如下：

1、 指出行为者（Actor）
2、 指出行为者的目标
3、 指出每一行为者：目标语句对的成功或失败的意思
4、 指出每一 Use Case 的主要的成功的情节
5、 在细化阶段，指出失败的条件及可恢复/不可恢复的情节

只有做到了第四步才能决定哪一些的东西在 Use Case 中逐步开发出来。

总而言之，Use Cases 是非常有效的需求捕获技术，它能使需求变得容易回顾，并且避免在需求中有实现细节的偏好出现。

对照表：

英文	中文
Scenario	情节
Internal structure	内部结构
Measurable	量化
Thread	线索

方　法

《分析模式：可重用对象模型》前言和介绍（笔记版）

Martin Fowler

翻译整理：Windy J（windy.j@163.com）

注：本文采用意译的方法，重在表达原文的意思，而不是逐字逐句的翻译，如有错误，请指正。

前言

不久以前市面上还没有关于面向对象分析和设计的书籍，但现在这类的书非常多以至于人们无法全部一一掌握他们。大部分的这些书致力于讲解一种注记方法，介绍一个简单的建模过程，并提供一些简例来说明。但*分析模式：可重用对象模型*显然不同，它关注的主要是建模的结果——那些模型本身，而不是关注建模过程——怎样建模。

我是一个信息系统面向对象建模咨询/顾问人员，为客户提供建模人员培训和项目指导。我的绝大部分经验来自建模技术知识和怎样运用它们。不过，更重要的是我在实际建模中的经验，以及看到问题经常会重复出现，正是因为这样我可以重用以前建立的模型，改进它们，并用于新的要求。

最近几年越来越多的人们发现了这一现象。我们意识到典型的方法学的书，尽管很有价值，却只提出了学习过程的第一步，这样的学习过程，还必须捕捉实际的事物本身。这样的意识发展成为了“模式（*Patterns*）”运动，但是怎样给模式下一个唯一的定义呢？我的定义是：模式是一种思想，它已经适用于一个实践环境中，并有可能适用于其他的环境。（A pattern is an idea that has been useful in one practical context and will probably be useful in others.）模式可以有多种格式，每一种格式都增加了一些有用的特别化特征。

本书讲述分析中的模式，也就是反映商业过程的概念模式，而不是具体的软件实现。大多数的章节讨论不同的问题域中的模式，这些模式难以按传统的行业分类（例如制造、金融、医疗保健等），因为他们经常适用于多个领域。模式可以帮助我们理解人们对世界的认识，而且基于这样的认识来建立计算机系统，并试图改变这些认识（或者可以称为商业过程重组工程 Business Process Reengineering——BPR）是非常有意义的。

当然，概念模式（Conceptual Patterns）并不能孤立存在，对于软件工程人员来说，只有当他们看到如何实现时概念模型才有意义。所以在这本书里我还提供了可以将概念模型实现成软件的模式，并将讨论该软件如何适合一个大型信息系统的结构，还将给出和这些模式有关的具体实现技巧。

建模人员将在这本书里找到在其他新的领域里有用的思想，这些模式包括有用的模型，设计背后的理由，还有什么时候适合和不适合应用。这些可以帮助建模人员在遇到具体的问题时更好地应用这些模式。

这本书里的模式还可以用来回顾已有的模型——来看其中哪些可以省略，哪些可以找到替代的方式来改进它们。当我回顾一个项目时，经常拿它们和从以前的项目中学到的模式相比较，就这样，我发现模式意识使我更容易应用以往的实践经验，这样的模式也远远比简单的课本更容易揭露模型的要点和本质。通过讨论我们为什么这样建模，将会使我们对如何改进这些模型有更深的理解，即使我们并没有直接运用这些模式。

本书结构

这本书分为两大部分：第一个部分讲述分析模式——来自概念商业模型的模式；他们提供来自贸易、测量、记帐（Accounting）、组织关系等多个问题域的关键抽象。这些模式之所以是概念性的因为他们关注的是人们对业务的思考和认识，而不是计算机系统的设计方法。这个部分的章节着重于可用的可选模式，和这些可选模式各自的优点和弱点。而且尽管每一个模式可用于特定的问题域，那些基础的模式还可以用于其他的领域。

第二个部分讲述支持模式，通过支持模式对这些分析模式提供使用帮助。支持模式描述分析模式怎样适合一个大型信息系统的结构，描述这些概念模型如何转换成软件接口和实现，还有那些特定的高级（advanced）模型构造怎样和更简单结构相关。

为了描述这些模型，我需要一种注记方法。附录中包括对本书所用注记方法的简要讨论，以及符号的意思，还包括哪里可以找到我所用（注记）技术的指南。

每一个部分都划分为章节，每个关于分析模式的章节包括那些在一个松散的主题空间相关的模式，这些模式也会受到产生它们的项目的影响。这样的组织方式说明任何模式都来自于实际的环境。每个模式出现在一章的各个小节，不象其他模式作者一样，我并没有为每个模式提供单独的标题，而是提供了一个类似于描述原始工程的格式，并增加了模式在原始问题域中以及在其他的环境下如何应用的例子。关于模式，其中最大的困难就是如何抽象到别的问题域；不过我认为这个问题最好留给读者自己去思考。

因此，这本书可以当作一个目录，而不是需要从头到尾读完。我努力使得每个章节独立于其他的章节，（虽然这经常是不可能的，所以每个章节如果引用了其他的章节，我会在介绍里声明。）每个章节的介绍部分包括该章节的通用主题空间，总结这个章节中的模式，以及模式产生的项目资料。

怎样阅读本书

我建议先详细阅读第一章，再看每章的介绍部分，然后就可以按照你的兴趣选择阅读各章，不管你用什么顺序。如果你不熟悉我建模的方法，或我采用的注记和概念，可以阅读附录。在模式表格中给出了每个模式的简要总结，以后回顾本书的时候可以查阅它们。非常重要的是这些模式在它们产生的问题域之外也非常有用，所以我建议你阅读那些也许不在你兴趣之内的章节，例如，我发现医疗保健行业中关于观测和测量的模式在公司金融分析中证明非常有用。

谁将阅读这本书

最大的读者群应该是面向对象计算机系统的分析和设计人员；

一个小规模但非常重要的读者群应该是那些建模项目的问题域专家；

希望程序员们可以钻研这本书，尽管本书缺乏代码和有着概念倾向，我建议你们着重注意第十四章，这一章讲述了概念模型和结果软件之间的关系；

数据建模人员；

经理们将发现这本书是开发活动的一个起点。从模式开始有助于简化目标，项目计划也可以获益于广大的模式设计背景；

学生并不是我定位的读者，但他们同样可以阅读这本书。

一本动态的书

书出版之后作者将难以改变书的内容，但是我一直在努力学习，而且这样的学习一定会改变我的原有的看法，所以我希望这些改变也能让读者们知道。

幸运的是 Addison-Wesley 公司为这本书提供了一个网址<http://www.aw.com/cp/fowler.html>，将用来提供更多的资料。使得这本书可以保持动态改变，我希望该网址会包括以下内容：

- 我学到的关于这本书的模式的新的东西；
- 对于这本书的问题解答；
- 其他人关于模式的有用解说；

- UML 注记出现的时候我会重画书中的图表并上载它们；

这个网址将是本书的补充部分，所以别忘了关注它，并可用来告诉我怎样改进和发展本书的思想。

答谢（略）

参考

1. Martin, J., and J. Odell. *Object-Oriented Methods*: A Foundation. Englewood Cliffs, NJ: Prentice-Hall, 1995

第一章　介绍

1.1 概念模型

大部分面向对象建模的书都提到分析和设计，但对于分析和设计的分界点却一直没有达成统一的意见。在对象开发过程中一个重要的原则是需要设计软件，并使得软件的结构反映问题的结构，这项原则的一个结果是从分析和设计产生的模型结束时都有意变得十分类似，从而导致许多人认为这两者之间没有区别。

我相信分析和设计之间的分别仍然存在，但它正日益成为一个重点。进行分析时你需要努力理解面临的问题，对于我来说这不仅仅是用用例列出所有的需求。在系统开发中虽然用例即使不是必需的，也是很有价值的一个部分，但捕捉完它们并不意味着分析的结束。分析还包括深入到表面需求的背后，来得到一个关于问题本质的精神模型（或理论模型：Mental Model）。

例如有人需要开发一套软件来模仿桌球游戏，问题可能用用例来表述表面的特征：“游戏者击中白球，它以一定的速度前进，并以特定的角度碰到红球，于是红球在某个特定的方向上前进一段距离”。你可以拍下几百个这样的事件，测量不同的球速，角度和距离，但靠这些样例要写出好的仿真程序远远不够，因为除了这些表面现象，你还必须了解背后的本质，那就是和质量有关的运动定律，速度，动量，等等。了解这些规律将更容易看到软件可以怎样建立。

桌球问题非常例外，因为那些定律早已成为公理。在许多企业中，规律并没有得到很好的理解，而需要我们努力去发现它们。因此我们建立一个概念模型来理解和简化问题。某种概念模型是软件开发过程中必要的一部分，就连最不受控制的黑客们都这样做，差别只是：建立模型到底是它本身的一个过程还是整个软件设计过程的一部分。

别忘了概念模型是一个人工产物，它们代表了现实世界的模型，但它们是人类创建的。在工程方面，模型让我们更好地理解现实世界，而且，建模人员也可以使用一个或更多的模型。例如，桌球问题中，人们可以采用牛顿模型或爱因斯坦模型，只是它们的精确程度和复杂程度不一样。

建模原则　模型没有对错之分，只有适用性不同。（*Models are not right or wrong; they are more or less useful.*）

模型的选择影响着结果（系统）的灵活性和可重用性。在系统中考虑太多的灵活性将使系统复杂度提高，这是不好的设计。设计需要在建造和维护代价上进行折衷。要得到满足某种目的的软件，你需要建立一个和需求相应的概念模型。你需要你能得到的最简单的模型，不要为你的模型增加不可能用到的灵活性。

最简单的模型不一定是你首先需要考虑的，因为简单的方案需要不少的时间和精力。但这些时间和精力往往是值得付出的，它们不但使得系统易于建立，更重要的是易于维护和将来易于进行扩展。所以值得用可以运行的更简单的软件代替原来运行的软件。

你将怎样表达概念模型？很多人把概念模型建立在他们的开发语言中。好处是你可以执行一个模型来验证它的正确性，和进行后续的研究。另一个好处是你最终将把模型转到开发语言，所以可以省区翻译转换工作。

坏处是，很容易陷入语言细节而忘记原本要面对的问题；还有很容易为特定的语言建模，使得模型向别的语言移植变得困难。

为了避免这些问题，许多人开始采用分析和设计技术，这些技术可以帮助人们关注概念上的而不是软件设计上的问题，问题域专家们也容易学会这些技术，由于它们使用图形，因而显得更有表现力。它们可以是精确/严格的，但并非必要如此。

我采用分析和设计技术的一个主要原因是为了和问题域专家一起工作，因为在概念建模时必须有问题域专家参与。我相信有效的模型只有那些真正在问题域中工作的人才能建造出来，而不是软件开发人员，不管他们曾经在这个问题域工作了多久。不过，当问题域专家开始建模时，他们需要指导。我曾经为客户服务监督、医生、护士、金融商货和公司财务分析员教授面向对象的分析和设计技术，同时，我发现，对于他们来说，有没有 IT 背景其实是无关紧要的，例如，我知道的最好的建模人员是伦敦一家医院的外科医生。作为一名专业的分析家和建模人员，我给这个过程提供有价值的技巧：我提供严格（的标准），我知道怎样使用这些技术，而且我“局外人”的眼光可以质疑模型中看似正确的地方。但是，这些是不够的，在建立分析模型时最重要的是专业知识。

分析技术倾向独立于软件技术，理想情况下，一种概念模型技术最好完全独立于软件技术，就象运动定律一样。这种独立性可以防止技术阻碍对问题的理解，而且产生的模型对各种不同的软件技术一样有用。但是实际上这种纯度不可能达到，例如，（实现）语言的变化也会影响到我们建模的方法。

在这里希望大家注意的一点是，概念模型更接近软件的接口（Interfaces）而不是软件实现（Implementations）。面向对象软件的一个特点就是接口从实现中分离，但在实际中很容易忘记这一点，因为通常的语言在这两者中没有显式的区别。软件组件的接口（它的类型：type）和它的实现（用以实现的类）之间的区别非常重要，《设计模式》一书许多基于委托的重要模式就依赖了这种区别，因此，当实现模型时也不要忘了这些区别。

建模原则 Conceptual models are linked to interfaces (types) not implementations (classes)。

1.2 模式的世界

最近一些年，模式成了面向对象界最热门的话题。它们引发了巨大的兴趣和普遍的宣传。我们也看到一些争论，其中就包括：模式到底是什么；当然难以下一个一般的定义。

模式运动有着不同的起源。近年来越来越多的人发现软件界不善于描述和利用好的设计实践经验。方法学家多起来了，但他们定义了一种描述设计本身（而不是描述实际设计方法）的语言。仍然缺乏描述基于实践设计的技术论文，这些论文可以用于教育和鼓励。象 Ralph Johnson 和 Ward Cunningham 提出的：*“哪怕有了最新的技术，项目也会因为缺乏一般的解决方案而失败”*。

模式也从不同的起点发展而来，例如，来自亚历山大·克里斯托弗的思想，来自为软件体系结构编写的手册，来自 C++的习惯用法等。

94 年《设计模式》一书的出版和 Hillside Group 关于 PLoP（“编程模式语言”）的会议带来了模式运动更广泛的知名度。

当然，软件模式思想并不局限于面向对象技术。

1.2.1 亚历山大·克里斯托弗

许多人看来，软件界出现模式这个词完全是由于亚历山大·克里斯托弗的工作，亚历山大·克里斯托弗是位于伯克利的加利福尼亚大学的一位建筑学家。他发展了一系列建筑学上的模式理论并出版了多部著作。

不过也有很多人否认亚历山大·克里斯托弗在软件模式运动发展中的中心地位，例如，《设计模式》一书的四个作者中有三个在写书之前并不了解亚历山大·克里斯托弗。

1.2.2 文字格式

模式写作的一个显著特征是它遵循的描述格式。一些人遵循亚历山大提出的格式，还有一些人则采用了《设计模式》中的格式。

通常来说，一个模式，不管它怎么编写，有四个必要部分：模式适用的*上下文*（context）陈述；模式提出的*问题*（problem）；形成解决方案过程中的*约束*（forces）；对这些约束的*解决方案*（solution）。这种格式带标题或者不带标题，但是是许多已经发布的模式的基础，这种格式很重要因为它支持对模式的如下定义：“在一个上下文中对一种问题的解决方案”。——一种定义限定模式为单一的*问题-解决方案*型。

一种固定格式使得模式和一般的软件技术作品显著区分开来，但固定格式也有它的缺点，例如在这本书中，我经常发现找不到*问题-解决方案*这种相应关系来说明一个模型单元；而且本书的几种模式都描述了对单个问题的不同解决方式，取决于不同的考虑/代价。

关于模式格式的另一项原则我毫无保留地表示赞同，那就是：模式需要命名。好处是可以丰富开发词汇，设计思想可以得到沟通。再一次强调，这不是模式独一无二的特点，这是技术作品为概念打造术语的常规做法，不过寻找模式促进了这一过程。

1.2.3 作者的抽象级别

模式产生于日常开发而不是学术发明。本书中所有的模式都是一个或多个实际项目的结果和项目中精彩部分的描述。

相信这些模式对别的开发者有所帮助，它们通常不只适用于模型本身的领域，而且经常在其他领域一样有用。例如 9.2 节中的 portfolio 模式，最开始作为金融合同分组而创建，但是，通过定义一个隐含的查询，它可以用来分组任何对象，并抽象到可以用在任何问题域中。

那么我面临的一个问题是我应该作何等程度的这类广泛抽象。如果我发现一个模式可以用于更多的问题域，那么我该让它变得有多抽象？问题是如果抽象一个模式，在使其超越原本的问题域时我不能保证它的有效性。所以我认为你必须判断模式对你的问题域是否有用，这一点你比我肯定，或者你已经接触到相当多的问题域专家。任何超出原始问题域的模式的使用都是实验性的，但是它们可以激发你的想象力，让你问自己：“对我确实有用吗？”

1.3 本书中的模式

模式来自实际项目中具有通用性的部分，所以，可以说发现模式，而不是发明模式。

本书中的模式分为以下两类：

分析模式：一组概念，代表业务建模过程中的公共结构，可能只和一个问题域有关，也可能扩展到其他的问题域。分析模型是本书的核心。

支持模式：本身有价值的模式，在这本书里扮演特别的角色，那就是：描述怎样选择分析模型，怎样应用它们，使它们真正实用。

1.3.1 具体建模样例

离开上下文环境，有很多问题难以理解并需要建模人员有相关的建模经验。而模式提供了一些好的方法来面对这些问题。本书中的许多模式通过考察一个领域中特定的问题来处理通用的建模结果，这样易于理解。例如，可以连接到单一对象实例的方法的处理（6.6 节）；状态图的图表类型（10.4 节）；模型的知识层和操作层分离（2.5 节）；通过查询利用 portfolio 模式分组对象（9.2 节）等。

1.3.2 模式来源

如上所示，本书的模式都基于我在大型企业信息系统应用面向对象建模技术的个人实践，也就是说，模式来自我曾经参与的项目。

我希望这本书跨越不同领域并形成交叉，所以我描述了多个问题域的模型，但我没有详细讨论模型的细节，一部分是由于客户机密的原因。

对于这些模型我并非完全自信，由于几个原因我作了一些改动。我简化了一些抽象；保留了原来的精神但让它更容易解释和接受。我还在一些具体的问题域上对一些模型进行了一些小小的抽象。有些情况下我改动了模型来反映我的意见。

对于模式的命名，我采用了用原项目命名的原则。

1.3.3 跨域模式

不管你工作在哪一个领域，我希望你学习你的领域以外的模式。本书的大部分内容包括通用建模要素和建模领域外可用的经验。其他领域的知识有助于抽象。很多专家不象我这么幸运，可以工作在多个领域。在不同领域考虑模型经常会在不相关的环境中蹦出新的主意。

我怀疑有一小部分高度通用的（商业）过程绕过了传统的系统开发和业务工程边界，诊断和治疗模型就是一个；另一个是记账和盘点模型（见第 6 章）。许多不同的业务可以共用一组相似的抽象过程模型。这对工业界各行业原来发展垂直类库的希望引起了一些大问题。我相信真正的业务框架会沿着抽象概念过程而不是传统的业务线来组织。

1.4 概念模型和业务过程重组工程（BPR）

读者可能认为本书中的概念模型是为了帮助开发计算机系统，其实它们还有其它的用途。好的系统分析员早就知道，采用一个已有的过程，并简单地将它计算机化并不是一种好的做法，计算机可以让人们以一种不同以往的方式办事。但是这种思想并不容易被人们接受，因为系统分析员们的技术好象还是太依赖于软件思考方式。如此看来，IT 业的人们还需要一段艰难的时间才能让商业领袖们理解这样的思想。

和 Jim Odell 的合作经常让我陷入到业务建模而不是软件建模；John Edwards 在 BPR 成为一个热门称号的很久以前就把他的方法叫做“过程工程”（process engineering）。用面向对象技术进行概念建模可以让系统分析和 BPR 变成同一回事。我所教授的问题域专家们很快掌握了它潜在的功能，并且开始用一种新的方式思考他们自己的领域。只有问题域专家们才能真正使用和应用这些思想。

因此本书中的模型在讲述软件建模技术的同时也讲述了业务工程（business engineering）。尽管业务工程的重点是关于过程（process），这些模式的一大部分是静态类型模型，这样的主要原因是我在涉及的领域：在医疗保健领域我们发现尽管我们建立通用类型模型，应用到这个领域的各个部分，却很难建立多的通用动态模型。

类型模型非常重要，我喜欢把它们当作业务定义的*语言*，这样的模式提供了一种方式，带来有用的概念，并使这些概念成为大量过程建模的基础。Accountability 的概念证实在医疗保健行业中的保密政策建模时很有用，在薪酬管理中我也看到建模改变了这个过程的语言和对这个过程的理解。

1.5 模式和框架

如果随便问一个专家面向对象技术的主要好处在哪里，答案通常是*重用*（reuse）。软件界的美好愿望是有一天开发者可以利用实验和测试过的可用组件来组装系统。不过很多这样的愿望实现得很慢，可以说重用才刚刚开始，更多的是在 GUI 开发和数据库交互上，还没有出现的地方是在业务级。

在医疗保健、银行、制造业或其他行业没有组件，是因为在这样的领域没有标准的框架（framework）。为了达到信息系统的组件重用，必须建立通用的框架。一个有效的框架不能太复杂，也不能太松散，应该基于一个大问题域的有效概念模型并可以跨越问题域广泛应用。建立这样的框架非常困难，不管在技术上还是在政策上。

本书不打算为不同的工业领域定义框架，而是描述一种状况下的可选建模方法；框架是关于选择一种特定的模型。希望这本书可以鼓励人们思考这样的框架并影响它们的开发。

1.6 模式使用

模式是软件技术上一种新的发展，我们也在发展新的方法来帮助人们在他们的工作中学习这些模式。面对书中大段的模式，很容易有一种被淹没的感觉（或不知所措的感觉）。

首先需要得到一个大致的方向，看完本章的介绍之后，建议你看看每一章的介绍，从那些介绍中可以了解到章节覆盖的主题。这时你就可以继续并阅读每一个章节了，当然这本书的写作方式并不要求你仔细读完每个章节。如果你在一个特定的领域工作，就可以阅读你认为合适的一些章节。另一种人们建议的方式是浏览书中的图表，如果发现感兴趣的地方，再看看样例，你可以从样例中看出该模式是否会对你有用。模式表也提供了模式的一个小结，可以从它开始或以后用来巩固记忆。

一旦你确定一个可能有用的模式，试一试。我发现要真正理解一个模式如何工作最好的办法就是自己找一个问题进行实验。可以只是在纸上划一个特定的模型，或写些代码看看。要尽量使模式合适（对问题），但也不要花费太大精力，因为可能这个模式并不是最合适的，那样的话没有浪费你的时间，因为你已经学到了关于模型的一些知识，可能还有关于问题的一些知识。如果模式确实不符合你的要求，尽管修改它。模式只是建议，而不是规定。不管有多适合，请详细阅读关于模式的所有说明，以确认你已经了解该模式的局限性和重要的特点。使用之前和使用之后都请别忘了这么做。如果你发现模式中一些没有给出说明的地方，不要只是骂我——发个邮件让我知道（100031.3311@compuserve.com），我非常希望能够看到别人怎样使用这些模式。

当在项目中使用模式时，我必须了解客户的意见。有些客户不喜欢自己和其他任何客户相似，他们自认为与众不同，并且怀疑外来的思想。对这些客户我不展示模式。在某个模式可能适合的地方，我用它来帮我设计问题，用这些问题引导客户走向这个模式，但我不是直接这么做，而是用问题来刺激他们。

另外一些客户高兴看到我公开使用模式，并放心让我重用以前的工作。对这些客户我在他们面前实验模式，并询问他们看他们是否喜欢。对我来说，他们必须明白，我并非把模式看作福音书，而且如果他们觉得不好，我会试试别的（模式）。因为这类客户可能存在不假思索地接受这些模式的危险。

对于回顾你自己或其他人的工作，模式也是非常重要的。对于你自己的工作，看看是否有和模式相近的，如果有，用模式试试看。即使你相信自己的解决方案更好，也要使用模式并找出你的方案为什么更适合的原因。我发现这样可以更好地理解问题。对于其他人的工作也同样如此。如果你找到一个相近的模式，把它当作一个起点来向你正在回顾的工作发问：它和模式相比强在哪里？模式是否包含该工作中没有的东西？如果有，重要吗？我把正在回顾的工作中的模型和我知道的模式相比较，经常发现在这个过程中，当我问自己“它为什么这样处理？”的时候，学到了很多关于模式和关于问题的东西。简单地问问为什么就能学到那么多东西，真让人惊奇。

写一本书经常意味着一定的权威性，读者容易把书当作肯定性的声明。尽管一些作者觉得他们说的相当正确，我却不这样认为。这些模型都来自实践，而且最多我能肯定它们对你有用。然而，我比其他任何人都更痛苦地知道它们的局限性在哪里。要得到真正的权威性，这些模型必须被大量的应用验证，不只是我的实践经验。

这并不表明这些模式没有用，它们代表了大量谨慎的思考。就象它们在我的建模工作中给我一个主要的起点，我希望它们也能对你有所帮助。重要的是，它们是一个起点，而不是一个终点。花些时间理解它

们如何工作，但也要去了解如何开发它们和它们的局限性。不要害怕继续向前，并请努力产生更新的更好的思想。同客户一起工作时，我并不把模式看成教条，哪怕是我自己创造的模式。每个项目的需求都会让我采用、改进和提高这些模式。

建模原则 Patterns are a starting point, not a destination.

建模原则 Models are not right or wrong, they are more or less useful.

过　程

参与变革

发现更多可重用的过程与简单的版本控制相比，更易走向成功

Lisa J. Roberts，译：mirnshi(mirnshi@263.net)

译者序：本文刊登于 SDMagazine，2000 年 11 月。论述了为什么要建立可重用过程以及从中得到的好处。译文中部分语句采用了意译，不妥之处和曲意之处请参见原文。

对开发过程共享实施猛烈的变革和改变是个什么样子？除了可能的时间大量损失（好，其实这是很小的可能，除了正在改变开发过程时），当它们获得了人们支持时就都会成功。

在历史上，人民，社会的劳动者，通过联合推动了社会变革。这就是我们所满意的应用程序，MeshTV，用二维和三维的有限元网格以图形的方式可视化和分析数据。它也能处理多种不同的网格类型，提供各种方式查看数据并去除了大多数的硬件和厂商的依赖，同时以自身图形硬件的速度显示图形。MeshTV 也可以并行工作，你可以想象得到这需要一些组织级别满足程序的复杂性，保持有序。最后说一点，MeshTV 大约有 450,000 行源代码。

这是我所作的全部描述。如果你想了解得更多，查看 www.llnl.gov/bdiv/meshtv，你可以下载可执行程序、源代码和手册。

受约束的混乱

象许多为内部使用而开发的程序一样，在加利福尼亚 Livemore 的劳伦斯 Livemore 国家实验室，对程序必需的修改和增加超出了可用的资源。在 MeshTV 项目中，这导致了混乱，(on the MeshTV project, this led to controlled chaos, where developers implemented new features based on the crisis du jour.)（译者注：这句话不好译）没有人有时间坐下来画出应用流程。我们都在实验室的里里外外，忙于我们客户的贪婪的需求。(有约 150 个文档用户-可能更多，实际上要靠 5 个兼职的开发人员支持）在我们这种状况下，这种过程导致了我们用户更多的抱怨和可靠性匮乏的程序。

三年前，我们的职责很小（几个用户，几个开发人员，很少有广泛的应用功能）允许我们在 CVS 上用非常不正规的过程管理源代码。当用户数和他们需求差异增多时，相应的代码管理的复杂性也增加了。处理我们增长的工作量也变得更加困难，我们知道有些事情必须要改变了。我们决定加入到我们部门的其他开发小组中去，并使用 Rational 软件公司的版本控制系统—ClearCase。从此，我们过程的改进氛围（our process improvement culture）开始改变，变革的种子已经播下了

在我们转向 ClearCase 前，我们小组的一位经理曾经诱导我们更多的集中在过程上，她徒劳了。她看到了增长的压力和用户的不满，她想我们应该尝试用不同的方法提高我们程序的可靠性和在用户那里的名声。不幸的是，她的话从来没有引起重视，同样我们也认识到了这点，但我们不得不忙于作完我们的工作。开发人员认为最好的情况是软件工程学所论述的那样，而在最糟和最可能的情况下，会占用大量的时间，提供众多的文档，用处不是很大。我们的一些老开发人员认为改进我们的过程没有用并且……(Some of our veteran developers saw no use for "improving our process" and would have sooner appeared in public in a tutu rather than utter such a sissy phrase.)（译者注：这句话太难译了，单词也不认识）而且俱乐部所有人，包括我也怀疑我们要收获的巨大好处。我认为 CVS 工作得很好，我们真的不需要更多先进的东西。

在 CVS 工作的同时，ClearCase 工作得更好。我认为在每个软件工程生产力上没有真正的提高，但是可以用我以前不能采用的工作方式工作。这些新的工作方法可以使管理源代码变得更容易，同时也减少了我曾经在 CVS 中遇到的问题。例如，我现在可以轻松的多并发地开发，我可以在我完成后可靠的归并我所作的。新特性弥补了我花在开发和学习新过程上的时间。

真正的产出

过渡不久，一位同事和我与一位来自苹果计算机公司的开发同行进行了一次有趣的讨论。他的工作需要产品开发过程的急迫应用，包括构建方法学和发行版本管理过程。当我们叙述我们通常随意无计划的方式时，他几乎震惊晕倒。后来的讨论，使我们惊奇的了解到了通过改进我们的过程所获得的好处。有一位经理鼓励我们是一方面，但是非同寻常的另一面是听到一位开发同行称赞他发现的好处。这是真正的产出，计算机科学风格的。

我们对其思考的越多，我们越认识到我们需要行动。将新特性和缺少固定发行日期联系起来的狂热，导致了在发行新版本和功能性的匮乏测试间长久的延期。我们的意图是好的，我们想让我们的客户满意。然而，不知何故，我们的期望事实上很糟糕，似乎看起来我们工作得很辛苦，但是，我们听到了更多的抱怨。我们需要改变现状。

首先，我们转换到有目的的发行版本日期，使其包含明确的更改和新的功能。像许多的内部产品，MeshTV 有着明显的直接的用户（MeshTV had users who lived "right down the street."）。新特性的不同的声音淹没在所有的目标回应中，我们尝试经常性的从那些所有从会议厅走下来，停下来聊会儿天的用户那里获取要求。（这些打断也可以避免我们持续工作）荒谬的是在试图获取所有要求中，我们不能满足他们中的大多数，我们失望了。所以我们从这种方法上退回来。或者试图将所有要求放进去（可能匆忙地完成，没有更多明显的 bugs）,或者针对最后的抱怨作一个改正（从而没有旧的要求）。我们需要一个载明新发行版本应体现哪些要求的计划。

这表明另一个过程需要改进。在决定之前，我们通过将 bugs 报告和要改变的要求写到纸上，保证可追踪。这片纸经常“走向了所有事物的归途”，或者偶然的扔进了废纸篓，或者压在了其他所有的纸张下面。（这点上我坚信我已危险地靠近制造我自己桌面中子星）（译者注：黑洞乎）那些纸随着时间的流逝而不能理解，无心的造成了客户输入损失的结果。

我们需要很好的保持我们改变要求的追踪，这样我们就可以为下一个发行版本选择明确的要求。在对几个产品调研后，我们购买了 Pure Atria 的 ClearDDTS，帮助我们管理我们的改变要求，我们试图一年四次发布新版本应用程序，这作为一策略被我们采纳，这样我们就可以很快的清晰地增加新功能，不用更新得过于频繁导致没有时间测试我们的修改。为了达到结束点，我们努力选择一定量的能够在三个月内完成的工作。第一次时，因为我们没有人知道如何预测一个详细的修改需要多长时间，我们彻底失败了。幸运的是，我们可以通过 ClearDDTS 跟踪我们曾经预测的时间和工作中实际花费的时间，并且个别开发人员利用这些数据预测将来。在为其他版本的选择工作中，这获得了重大的成功。变革明显的站住了脚。

我们也决定与目标发行版本并进，着手提高质量的工作。为了完成这点，我们要求所有决意要改变的要求要被不具有开发责任的其他人所验证。当我们知道其他人会检查工作时，我们就都会非常仔细，这多么惊奇。我们也开始采用 Mercury Interactive’s Xrunner 和内部使用的脚本开发了一个自动测试系统。将来，在我们发布一个新版本应用程序前，这些测试必须成功的测试过。

持续的改进

所有这些工作都以我们不能想象的方式的到回报了。我们更好地跟踪我们的改变要求，也就是我们没

有丢掉它们，我们确实能跟得上用户的更新状态了。用户喜欢这点，我们也不再面对来自客户的挫折。他们也喜欢我们更频繁的更新和更加健壮的程序。

我们正在寻找另外的方式，我们可以从改变我们过程中获得好处的方式。现在，我们正在研究软件开发成熟度模型（CMM），看是否能通过遵从 2 级帮助我们提供更好的应用软件（请到 www.sei.cmu.edu 查看更多的 CMM 信息）。我们可以从这种方式中获得好处，但是我们想确认通过 2 级认证能够编写更好的代码，不只是更多的文书工作的要求。我们的兴趣在于进步，不在于核对 boxes。

我们正在进行的另一个改变是，我们能够更容易地在我们每年四个主版本之间发布修订 bugs 的版本。这点可以是我们更快的转向用户的要求，平息用户的愤怒。（我们基于用户认识到的严重性和我们察觉的严重性的比较，选择哪些 bugs 被改正，还包括工作区存在的风格。我们在整体上也为客户整合了全部优先级的感觉）

我们过程改革的更多惊人结果之一是不只影响了程序，更多地影响了程序开发人员。他们停止了改善用户的态度，转向提高应用程序。程序变得更可靠，发行版本变得更可预测的同时，用户对应用软件整体上也更加满意。

但是我们不仅只关心我们的过程的改革。MeshTV 的开发人员同其他的开发小组并同工作着，我们试着同其他的小组分享我们的经验，学习我们的胜利和错误。围绕我们新的过程，我们提供了四个展示，至少有一个小组已经决定采用我们的一些经验。变革在成长。

成功孕育成功

当管理层尝试推动改变过程时，从最初的怀疑和愤怒，我们在这个过程中经历了巨大的变革。这些曾经著名的过程都是那些所有刊物都讨论过，当作福音传授给我们的同事。当我回头看看我们的小组，我惊奇于我们从使用一个版本控制系统改变到几个可重用性、文档化过程，甚至将那些经验出售给别人。什么能够允许我们背离我们正规的商业方式

对于我们来说，在开展新的过程中最重要的因素是一个成功的战士，他酝酿了过程改革中的兴趣。这个战士应该是一个受到尊敬的开发人员，在小组里的，因持续工作而闻名，因渴望经验的增长而受人尊重。在我们的事中，我们有二位战士，我的同事 Sean Ahern 和我自己。在我们兴奋于可能的好处并开始着手于一些改革后，小组的其他人信服了并跟随我们。当管理层决定了过程必须跟随时，来自小组外的压力出现了。对于外来的，组员将可能排斥它，对此我不能强调得足够多。然而，来自里受人尊敬的组成员的狂热，开发人员感到是必须听从。毕竟，这些人们事实上知道到底它象个什么样子。一旦其他团队里的人看到了真正的好处，他们就会跳进这个潮流中，变革也就会良好的进行下去。当开发人员开始思考改进过程的方式，获取真正的好处时，好戏就开始了

你如何开始你的变革？每次一个改变。一旦明显有好处，你的同事就会加入到你的行列中，同你一起征服编程世界。

过　程

创建成功的工程

作者：Bruce Eckel，翻译：Hairui

Bruce Eckel，著名的计算机语言和工程专家，著有《C++编程思想》（Thinking in C++）、《Java 编程思想》（Thinking in Java）等书籍 相关网站：http://www.bruceeckel.com --译者注

以下工程开发指导是我对决定一项使用任何语言的软件工程成功与否的决定因素的一些认识。

1．记住往往事与愿违

纯粹的“轶事工程”（原文为：anecdotal project，其含义不好理解，暂译为"轶事工程"，盼指正--译者）的失败几率总是存在，一些低至百分之五十而另一些高达百分之八十，但是所有的这些都表明：你失败的机会大于你成功的机会。为什么我要从这个令人丧气的预测开始我的话题呢？因为每一天开始时，我都想“今天将会不同，我今天能够完成四倍数量的事情”，尽管（在此之前）有过一系列不间断的例外。对于软件工程来说，过度的狂热往往被那些（只）关心结果人所夸大——“这一次，我们将解决以前从来没有人解决过的问题，只需付出更少的时间和更小的代价”，尽管他们知道，真正的规则是：“你只能从此三者中选择一个”。

记住你身后高高堆积的纸牌[i]非常重要。你手中有一根包含时代力量的魔术手杖或者是挂在悬崖上，会让你做出完全不同的两个决定。如果你懂得你的处境属于后者，你将会说：“是的，这很好。但首先让我们看看我们是否能够在现有的进度和预算情况下完成这一切。”

一个将不稳定形势和对失败的认识放到显著位置的方法是研究过去的失败。一份很好的资料是Roberts Glass（一位爱好研究崩溃的专家）的著作：《软件失控》(*Software Runaways, 出版信息：Prentice Hall 1998*)，以及他其它的著作。此外可以阅读 Tom Demarco 和 Tim Lister 的经典之作《人件——生产性工程和团队》 *(Peopleware: Productive Projects and Team, 出版信息：第二版，Dorset House, 1999)*。

2．切合实际地安排时间

时间安排的“魔法”经常受到非开发人员为满足软件开发实践之外的愿望和期待而产生的想法所驱使。最近我校正了自己的时间安排策略。我先将整个工程显示脑海中，然后闭上眼睛，清理自己的大脑并让它判断这个工程大概需要多少时间。如果不考虑奇怪的技术问题、各种会议和其他分心的事物的影响，得出的这个时间居然非常合理。但我建议将这个合理的时间乘以 3，或许可能是 4，并且加上百分之十。如果这个估计出来的时间将让你失去市场机遇，那么考虑不要进行这个工程。如果你认为像这样计划时间不合理，那么首先请注意，大多数工程将遵循这个规律。其次，试想一下，如果你所在公司的所有工程都很成功进行会带来局面：你将拥有更多的收入；更少的程序员会因为愚昧的工程时间安排筋疲力尽而退出。你也许还会争论说这个时间评估技术非常没有科学性，这一点我同意。然而，所有的软件评估技术都含有臆测和直觉的成分在内，甚至连功能点（原文为：function point，若有其他正规译法，请指正）分析都需要对功能点进行猜测。我的“信封背面”技术将所有臆测结合到了一起，而不试图假装没有猜测。用更少的时间，也许产生更好的结果。但是，我的猜测是建立在我自己的经验之上的。

3．首先让它运作起来

当我试图进行一些无意义的事情时，我最大的创造性成功来临了。铭记最重要技巧——当你开始一个工程时，你好比已经用手指将自己挂在一个悬崖之上；然后你考虑一下能够做什么疯狂的事情简单地让你

的工程运作起来。这并不意味着你需要马上投入进去并用通常的方式开始撰写代码，你只需要尽早尽快找到一个转换周期非常短的工具，用来判断你是否可以做该项工作以及你的工程可行性如何。我在后面将要提到的 Python 语言就是这样一种工具。

将你的计划运作起来有很多好处。凭你的经验，你应该知道，用户只有能够开始使用你开发的东西的时候才能理解你开发的是什么，然后他们会突然产生各种念头并对该软件应该做些什么真正提出要求。一份系统说明书往往只是一份文档，人们往往不会认真地阅读，但是如果你让他们体验一个可运行的程序之后，他们就会确切地明白你的意思。更早地了解用户们真正想要什么岂不是更好？

事情往往会比你想象出来的要复杂四倍以上，所以对你能够完成的东西要尽可能地保守一些。无论何时，一些不可知的因素都在伴随着你的工作（这一点你可以从产品描述中一些“最”中察觉到：“最快”、“最大”、“最新”），原型的价值不能进行夸大。如果在此之前你没有做过类似的工程，那么最重要的事情是尽快地判明该工程是否可以实现，开发一个根本不能发挥作用的程序将会以浪费你的大量金钱而收场。

最后一点，优化。要能够在这个阶段抵抗得了诱惑。牢记 Donald Knuth 说过的话（其中略有一点开玩笑的意思）：“不成熟的优化是所有麻烦的根源”。虽然优化是一些工程的关键因素，但是在确认程序切实可行之前一切优化都是盲目的。在最后建造系统之前浏览一遍所有的问题。每个工程都有一些你没有接触过的东西，你应该首先将注意力放到这个领域，创建一个测试程序或者原型来寻找解决问题的方法。在你知道你是否可以做到并且知道做到的难度有多大之前，你没有其他办法能够得知工程是否能够成功、如何为它安排时间以及它需要多少付出等等。

4．使用恰当的工具

一个工程的早期部分应该是高度探索性和实验性的，因为那个阶段是发现自己不会做什么以及如何去建造程序的阶段。寻找最适合工具的最好方法是去体验一下他们，然后摈弃其中工作效率低下的那些。例如，你可能开始的时候用的是 Rational Rose，后来决定使用 Visio Professional 来创建视图，因为你需要 Visio（或者通过 Versa）提供的一些特性。

用来做工程的恰当工具并不一定就必须是你已经了解的编程语言。当使用一种语言时，你就被局限在该语言所能表示的范围之中了。如果你是一个 C++程序员，你很自然可能想用 C++创建所有的工程管理和工具。但当你需要更加灵活的工具时，Perl 是一种更快速的选择（甚至将考虑学习需要的时间在内）。在你的实际工程开发中，使用 Python 来快速造型或者甚至交付一个内嵌 Python 语言的应用程序将给你带来更好的局面。首先，它是免费的，所以不需要支付任何许可授权费用；同时它对 C 和 Java 有完全兼容的接口，你可以使用 Python 解决所有 Perl 能够解决的问题，所以它是 C++和 Java 的一种完美的辅助语言。

5．接口的设计

在 C++中，接口是一个包含所有虚函数的类；而在 Java 中接口技术被直接支持；在 COM 和 COBRA 中，你没有其他选择，你和所有的抽象打交道——所有的都是接口，没有实现。接口提供了一个更加整洁的设计方式。要想让程序员们确信这一点有些困难，但是它对将 COM 或者 COBRA 指定为构件模型非常有帮助的（COBRA 技术也是与操作系统无关的技术）。它不仅仅提供了工程实现语言的灵活性，并让你能够完全地将工程切割开来。如果你打算在你的开发组或者公司之外实现你的工程的一部分，整洁的接口可以阻止任何与工程其它部分不适当的连接，同时你可以用任何语言来进行开发。你可以采取快速造型来实现所有的接口，稍后才对其中比较特别的部分进行优化。

6．设计时充分考虑异常情况

在 C++中，异常控制并不像在 Java 中那样得到有力支持——这是 Java 在工程管理方面成功之处。在设计、代码编写和模块使用的时候往往会有一些错误，除非软件自身能够通过抛出一个异常来声明这些错

误，否则你将会花费许多小时或月的时间来捕获这些问题。只有通过严谨的异常使用，你才能保证这些问题不会出其不意地让你的工程陷入困境。

7．简洁往往付出代价

虽然很难说服管理部门，但是“简洁”这个词是可维护性和复用性的同义词。不仅如此，一个简洁的程序让人感觉很好。但是因为我们确信软件工程是一种商业行为，目的是为了赚钱，而不是为了感觉，因此很难说简洁的程序比其他非简洁的程序更加有灵气地结合在一起。但是由于软件是一种能够赚钱的艺术实体，在美学和实用性之间必然会一场争论。

8．人与人之间的交流是一个瓶颈

这就是为什么小型的组队往往更加有生产力的原因。当一个工程像火焰一样失去控制的时候，将更多的程序员扔进火焰将使情况变得更糟。这也是为什么简短的小会议往往可以发挥作用而冗长的大型会议却做不到，还有为什么太深的管理机制会导致生疏的原因。参阅《人件》（早些时候提及过）一书了解更多的细节。

解决交流问题的最好办法是免费的：在一台废弃的计算机上安装一个 Linux 服务器，你可以在几分钟内完成这项工作，自动安装将包括一个 Apache 网页服务器。然后将你们所有的文档，从测试分析到用户文档，拷贝到服务器上，以便每个人都能够访问到最新的信息。你可以轻松地加入 Java Servlets 或者 Perl 脚本（http://www.perl.com）或者 Python(http://www.python.org)来收集每一页的内容，然后用一个 List 服务器来向所有的成员发送公告。如果你想用 camera-ready 格式来提供文档，你可以用 Adobe Acrobat 格式来代替 HTML 格式。如果你的工程足够大的话，指定一名成员专门负责维护服务器是值得的。

9．制定一份计划（可以是任何类型的）

我曾经见过许多工程在没有签订任何合同（更别说一份计划）时已经有大量资金流动。哪怕是对于一个很小的工程，你也需要某种计划，甚至它可能只是被写在一个信封的背面或者只存在于主程序员的脑海中。当工程逐渐变大，你需要一个回顾的过程。一个典型的计划包括：分析阶段（包括你打算用程序解决什么问题以及程序将完成什么）、设计阶段（程序如何完成它的任务、程序实现的组成、分析阶段预定目标是否达到的测试使用信息以及发布、安装和培训等事项）。当新的信息被收集时，这些阶段将被重复。根据工程的大小，这些步骤将被缩小或者放大，但你必须像熟悉你的编程语言一样熟悉它们。

10．考虑外部帮助

一种放弃：我的公司提供培训和咨询服务，因此当然我感觉这是一个好主意。然而，如果你的公司内部有经验丰富的人可以担任你的工程的顾问，你可以不必向公司之外寻求帮助。这是一种以知识为基础的商业行为，生产力最低和最高的软件工人之间的生产力差别是很大的。如果你无法雇用那些最有生产力的工人，你可以通过培训提高他们的生产力，通过咨询和代码预排来改进你的工程的分析、设计和实现。对于顾问和客户来说，有一本优秀的书籍是 Weinberg 的《咨询的秘密》（*出版信息：Dorset House,1986*）

另一方面，我曾经见过一些工程中，使用外部的开发组队剥夺了内部的队伍的权利，该项目最后以花费更多的时间和资金收场。这将我们带到我的最后一个提示：

11.了解永远没有银弹（原文为：silver bullets,此处直译为银弹，估计引申含义和 free lunch 接近**）的道理**

这句谚语是由 Fred Brooks 发明的，对于今天仍然适用——尽管有许多“银弹”已经被发明出来了。统一建模语言（UML）就是这样一个例子：它当然是一个很好的通用词汇表和设计符号集，但是 UML 仅仅

轻微地减少了方法学家之间的争论而已。永远不会有不劳而获的事情。你必须艰辛地计划你的对象、它们的接口和结构，然后跨越一道道障碍将工程变成成果。你必须清楚没有任何可以保证成功的方案可以依赖，同时牢记工程的失败的几率让自己更好的瞄准成功。

过　程

更好地领导一个项目的诀窍

Warren Keuffel，自：SDMagazine，1999年9月
Think(think@umlchina.com)译

技术管理就像开车。当你做得正确时，没有人注意，一旦某个环节出错，问题会接踵而来。以下是我11年来作为Interviewing Manager的Team管理体会，排名不分先后，你必须注意每一点。

1. 不要重复过去二三十年来别人犯过的错误

这句话来自Steve Mcconnell，IEEE软件编辑和软件开发畅销书作家。Mcconnell的作品包括经典著作“Code Complete”。Mcconnell认为，“大量阅读”是避免凡重复错误的最好方法。

2. 80%的管理就是选择正确的人选

Scott Adams, Dilbert漫画的作者认为一个好的项目经理必须创造一个人尽其用的环境。所有的项目经理都应该读一读Tom Demarco和Tim Lister的新书“Peopleware：Productive Projects and Teams”（2nd Edition, Dorset House, 1999）。

3. 总是试图雇用比你强的人

不要让你的自负成为项目的瓶颈。组织一支聪明的队伍，给他们足够的资源和解决问题的规则，让员工自己解决问题。

4. 不要浪费时间

Tom Bragg，Intellisys Technology Corp.的首席技术官员，认为太多的项目由于不能如期开始最后陷入麻烦。通常导致延迟的原因包括其它任务的干扰，人事变动，不准时的经理等等。

5. 最优的未必是最大的

Tom Bragg的另一个建议是：密切注意项目开始后发生的事情。Bragg说：“计划好你的工作然后如期进行，过分紧张的工作强度反而往往导致生产率的降低，可能保持每周50小时以内的工作强度是最佳的。

6. 真实的，公正的估计

项目经理应该避免“依照管理者的欲望修改计划”的陷阱。“一个有效的估计的特征是所估计的时间与金钱比实际情况低和高的概率相等”，Bragg说。

7. 使你的组织结构更有效率

很多情况下，你可以采用另外一种与现在不同的组织结构。看一看Apache Web server的开发小组，他们的层次组织并不分明，却开发出了成功的产品。

8. 使用WWW上的免费工具

从http://sunnet.usc.edu/winwin/winwin.html，你可以下载由Barry Boehm的学生开发的，能够把W理论（WinWin模型）和螺旋形模型结合起来的工具。在项目管理研究所的网址www.pmi.org，你可以下载它的联机手册。从www.spmn.com你可以看到从CMM模型出发的一些建议以及两套工具：Control Panel和Risk Radar。Control Panel是Excel表格形式，由于监测生产率和质量；Risk Radar是一个Access数据库，对项目的风险进行量化管理。

9. 不要小看老程序员

重新训练现有的程序员比雇用新毕业的大学生要有价值。老的程序员在以往的多个项目上有丰富的经验，通过新技能的训练后，他们的经验和知识会帮助年轻的程序员（包括项目经理）节约时间和金钱。

10. 为你的项目选择定正确的工作流程

并不是所有的项目都适用一种开发流程。Intel公司有规律地检查每个开发小组的工作质量，如果出现了延迟交付或质量问题，Intel鼓励该小组改进他们的工作流程。

11. 做好你的生存计划

工　具

选择一种UML建模工具

来自objectsbydesign(http://www.objectsbydesign.com)

翻译：think(think@umlchina.com)

以下标准用于评估一种UML工具。当然，除了已被列出的以外，可以用这些标准来评估的产品还很多，但如果你想选择最好的，请花时间按照清单对产品作测试。如果你特别重视某项标准而在清单中没有列出，请告诉我们。

信息仓储支持

对于一个大项目，信息仓储(Repository)对在开发人员之间共享组件设计是必要的。两个以上的开发人员可以共享同一模型的的组件，甚至可以通过在适当级别上定义所有权和共享权来合作进行单一组件的开发。

信息仓储通常用提供数据共享和并发控制等特性的数据库来实现。 通过提供锁定和只读访问，信息仓储允许一个开发人员拥有整个模型而其他人对该模型及其组件只读访问，或者将这些组件结合到自己的设计中。重要的是： 这种工具应该允许你从另一个模型只引入你所需要的组件而不必引入整个模型。

构造信息仓储的另一个令人感兴趣的方法是利用项目的源代码，使用源码控制系统来提供并发控制。这种方法的好处是在源码和模型之间有更高级别的同步，另一个好处是更除去了另一个数据源--别忘了，如果你为信息仓储使用了数据库，你必须对各种存储数据分别备份并完成在模型、信息仓储和源代码之间的三方同步，而不止是在代码和模型之间的两方同步。

有了建模工具对信息仓储的支持，对任何组件的修改将被自动传播到所有引入该组件的设计。

双向工程

对源代码(Java, C++, CORBA IDL)的正向和逆向工程的能力是一项复杂的需求，不同厂商在不同程度上成功地支持这一点。对正向和逆向工程这两方面的成功结合，定义为双向工程。

正向工程在第一次从模型产生代码时非常有用，这将为你节省许多用于编写类、属性、方法代码的琐碎工作的时间。

在以前没有模型存在的情况下，将代码转换成模型；或者在迭代结束，重新同步模型和代码时，逆向工程非常有用。

在一个迭代开发周期中，一旦一个模型作为迭代的一部分被修改，另一轮的正向工程应允许所有加入该模

型的新的类、方法、属性的代码被更新。这个步骤通常不被开发者采用，因为许多工具在这个过程中没有办法管理源代码，问题在于源代码中不只包含与模型有关的信息。工具必须精于对在新一轮正向工程之前已有的源代码进行重新构造。

至少，建模工具应成功支持一开始的正向工程和全过程的逆向工程。同样，建模工具对纯Java语言的逆向工程的支持应该毫无问题。一定要针对你自己的源代码确认这一点，因为我们见到过优秀的工具在对Java的一些特性如内联类(inner classes)等进行逆向工程时失败了，每一次进行逆向工程时，你不得不把讨厌的代码注释掉----确实非常痛苦。

HTML文档化

对象建模工具应能为对象模型及其组件无缝地产生HTML文档。HTML文档提供对象模型的静态视图，以便开发者通过浏览器迅速查询而不需要加载建模工具本身。另外，通过产生HTML文档，所需建模工具的许可证(licenses)会因减去那些对模型只需要有只读权限的人而减少。

HTML文档应包括模型中每个图形的一张位图，并允许通过超链接浏览整个模型。产生HTML文档所需的时间应是合理的。现在许多产品在不同程度上成功支持这一点。再说一遍，你必须亲自测试这个特性，在特征表上有打勾并不能保证成功支持。

完全UML1.3支持

虽然许多工具声称完全支持UML1.3，实际上，这是一项复杂的需求，一些工具并不能做到广告所声称的完全支持。至少应支持的图表有:用例图(Use Case diagrams),类图(Class diagrams),协作图(Collaboration diagrams)，顺序图(Sequence diagrams)，包图(Package diagrams)，状态图(State diagrams)。

类和方法的选择列表

建模工具应在一些关键界面上提供选择列表:

协作图(Collaboration Diagrams)和顺序图(Sequence Diagrams) --工具应允许从模型的类列表中选择一个类，把一个对象分配给它，并允许对象间传送的消息能够从接收消息对象(类)的有效方法列表中选取。

类图(Class Diagram) --工具应允许从别的包或模型的类列表中选择并引入类 。

选择列表特性在直观上对建模工具至关重要，可以看作是必备特性。能够迅速从列表中选择一个对象到另一个对象的消息，给开发顺序图和协作图带来很大的方便。

数据建模集成

对象建模工具应允许集成数据建模工具。有许多方法可以提供这种功能。一种方法是UML工具提供将对象模型转换成DDL(数据定义语言，用于为类创建表的SQL)。另一种方法是UML工具输出元数据到能够输入这些元数据的数据建模工具，并将其作为数据模型的基础。一套先进、完整的工具应允许数据模型和对象模

型之间在每次设计的迭代之后同步。

版本控制

建模工具应允许储存各种版本，以便后续迭代开始时，以前的版本仍然可以得到，并用于重建或保持基于该版本的已有代码。

模型导航

建模工具应提供强的导航支持以允许开发者全盘浏览模型中的所有图表和类。一种方法是提供一个按名字排序的类目录或选择列表，以便设计人员随意跳到图表中想去的类。

对于大的图表，工具应使得在缩放和平移时，能够轻松实现浏览。

工具也应允许在使用双向工程时，对类的源代码轻松浏览。

打印支持

建模工具应允许一张大图表能够准确地用多个页面打印出来，并提供打印预览和缩放功能，轻松地使图表能够在所需页数内放置。允许将一张图表放置在单页中的能力在清单中是高要求。不幸的是，我们发现许多工具很难用无缝的方式完成这项重要的任务。

图表视图

建模工具应能方便定制类及其细节的视图。例如，它应有可能从图表中排除所有的get/set方法，因为它们会对阐明一个图表造成混乱。方法的全部信息应允许容易地根据不同级别细节的需要显示或隐藏。属性和方法的可见性(private, protected, public)是用于选择什么该显示，什么该隐藏的另一个尺度。

输出图表

一个经常被忽略的关键特性是用某种格式输出图表，以便引入到文字处理文档或Web页面中。用于输出的最流行图像格式是GIF、PNG和JPEG。输出时，工具应允许你定义所产生图形的首选分辨率和尺寸。这个功能需求来自那些野心勃勃，需要写一本包括图表的UML书籍的作者，或者希望将他们的工作展示在网站上的人。

脚本

用脚本编程是建模工具应该支持的另一个强大的特性。有了脚本功能，高级用户可以创建能在建模工具内直接访问对象模型的脚本来添加其它功能，例如：为当前开发的项目做的项目管理表格，定制文档，定制代码，报表和度量。一个定制代码的例子是集合类和用于访问集合类的get/set方法。

为了方便使用脚本，建模工具应公开访问自身对象模型的接口，以便在开发时能提供对对象模型组件的访

问。(如果这一句听起来有点绕口，请再读一遍。)例如，脚本编写者应能在整个迭代周期中访问类图中类的集合，从而能够通过类对象的accessor方法来访问类的属性。当然，脚本语言自身应该是面向对象的；一个明显的选择就是Java语言本身，另一种选择就是Python脚本语言。

健壮性(Robustness)

你的UML工具需要象岩石般坚固可靠，以防止设计期间工具崩溃而使用户的时间和生产率在不知不觉中损失，或者在模型没有备份的情况下崩溃。我们曾亲眼见过许多领先的工具因为崩溃或文件损坏而引起数小时的工作成果丢失。如果你是一位开发人员，你知道那种因“生产率高的软件”反而比粗糙的代码工具生产率要低而产生的蔑视感觉。如果你是一位经理，你会看到被要求使用一种不可靠的工具时开发人员的愤恨。

今天，健壮性常被发现于用Java实现的应用程序(JVM运行时保护)或开放源码的项目(在web范围内并行调试)。发现某种特定的UML工具是否健壮的最快方法是在comp.object等新闻组四处询问，你一定会听到许多抱怨的!

可用于此处的另一个策略可以借鉴有效率的办公应用程序，我们也推荐工具开发商采用这种策略。该策略就是让UML工具每隔一定时间间隔就在背后自动保存模型。

平台

为了使你在建模工具上的投资得到最大回报，请慎重地考虑工具将运行在哪种平台上。你需要为Windows还是Unix开发软件？还是二者都要？将在哪种平台上开发？

最近的各种事件一起推翻了这个神话：一流的跨平台图形用户界面还不能实现或者拥有一个"最少共同支配者"的视感。很长时间以来,这是不可能的(除了基于HTML的应用程序之外)，直到最近Java的Swing用户界面的出现。但是，跨平台工具需要在Linux等常用平台得到支持，以大规模地被程序员们采用。

Sun最初几乎没有做什么事情来促进Java在Linux上的应用。但最近工业界元老们，主要是IBM，IBM保证在他们所有的硬件平台上为Linux提供无限广泛的支持，并支持Apache/Jakarta项目，IBM现正快速地在Linux上推广Java。也许因为IBM已经开始为主要的Linux厂商发放它的JDK 1.1.8版本，Sun被迫支持在Linux上的 全功能JDK 1.2 （带Swing的Java2）的发放。通过Blackdown小组的努力，这个Linux上的Java端口大部分已被完成。

迄今为止我们已经测试了一种Linux平台上基于Swing的领先Linux工具，结果优秀。但要告诫的是：128M内存是必需的。

版本更新

你需要选择一种将会不断通过修正错误、改进性能、添加新特性来进行改进的建模工具，毕竟你在时间和金钱上进行了一项大的投资，而且改换到另一种建模工具并不容易。

小心那些已经被大公司拥有的产品。在兑现所有期权之后，最初的开发者常常会离开公司，寻找下一次大的机会。寻找有才能的、能读懂和维护最初并非由其编码的软件复杂模块的程序员并不容易。这种局面也会出现在开放源码项目上。

如何能知道一种产品是否在改进？向销售代表询问下一版本发放的详细时间表以及该产品将来的蓝图。密切观察产品改进和添加新特性的速度。产品计划什么时候支持UML 1.3？图形界面是否支持最新的流行样式？你也可以看看该公司的网站：如果产品发布和外界评论是旧的，就是可疑的。

未来...

现在我们来看看对未来有什么希望。建模工具的当前成熟程度表明，工具厂商准备通过添加高级特性来使产品达到新的高度。我们希望在下一代产品中看到以下特性的出现。

集成编辑器

在模型的迭代开发过程中，将UML图表和相邻窗口的源代码匹配，是非常有效率的。支持这种视图协调的产品可以给模型设计者的工具箱添加一个额外的功能选项，以直接给建模工具添加强大的源代码编辑特性。当建模工具不必作为设计者的首选编辑器时，能够在代码里直接更改方法的名字或原型，并立即反映到模型中。

最想要的特性是类似emacs等流行编辑器的键盘仿真，另一个热门特性是通过改变颜色来突出语言关键字，注释等等，提高了代码的易读性。一个重要特性是在类图中选择一个类、属性或方法时跳转到匹配代码行的能力。 最重要的是，编辑器应该是快速易用的。

作为变通的方法，另一种解决方案是允许建模工具和开发者喜爱的编辑器通信。例如，通过一个热键，允许建模工具从当前活动窗口跳转到伴随编辑器的匹配代码行。

自动生成

我们真正想在不久的将来看到的一个特性是，建模工具帮助产生交互图和状态图的能力。

工作方式是：在一个已有的程序的执行过程中，建模工具应容易生成一个追踪文件，目的是获取对象间互相传递信息时的交互。产生追踪文件后，建模工具将被用于分析该追踪文件，以发现对象交互的模式。建模工具应允许用户从一组类中选择一个来分析，然后展示被追踪文件记录的每个类唯一的一套交互，允许用户为模型选择交互。最后，工具应能够产生一张基于真实记录对象交互的顺序或协作图。

很酷吗？它并不象听起来的那样太过前卫。因为追踪技术已经十分成功地应用在帮助开发人员追捕他们的程序中性能瓶颈的工具中。这类产品一个很好的例子就是KL Group的JProbe，用于分析Java程序的性能。

使用同样的技术自动产生状态图也是可能的。对以前描述过的顺序的修改将允许用户为状态机里的状态指定基类的名称。建模工具将追踪基类的衍生类之间的交互。从这种追踪，建模工具能够通过描绘每次被记

录的状态迁移来创建状态图。

管理工具

如果你是项目经理中的一员，你最有可能想要能够估量你的O-O项目进展如何。一个应被集成到建模工具中的很好的特性是能够输出模型信息到允许你追踪项目设计和实现进程的工具中。由于它的通用性和可塑性，电子表格是实施这个解决方案的理想工具。项目管理工具也是理想的候选。

这个特性如何工作呢？在高层次，通常你想追踪的是模型中的类和负责在这些类上工作的的人。 你想知道什么时候有人开始在该类上工作，完成任务到了哪种水平。在下一层次的细节上，你想要知道每个类的方法。在这一层次，你可能想要知道哪种方法已经包括在交互图中，或在实施阶段，每种方法完成了多少代码量。

要使这个特性起作用，你需要“敏捷”地更新你的项目管理信息。不象报表工具那样总是从头产生一个新报表，你只需要在第一次输出所有东西。产生初始报表后，你的建模工具应该只被要求用新信息来修改你的管理工具。根据用户需要控制的级别，建模工具能在输出之前展现给用户一个修改的清单。

建立一个项目管理链接的一个美妙的好处是，提供把分析和设计阶段的完成日期作为目标的能力。具体方法是通过计算进展速度，并基于完成模型所需的剩余的工作，使用这个速度来计算预期的完成日期。

度量(Metrics)

当你的项目开始成熟时，你可能需要知道你的模型的度量。度量能在一个特殊的模型的生存质量上给面向对象分析员一些即时的反馈。一些感兴趣的度量包括：类层次中的超类数量，每个类中方法的数量，每个类中属性的数量，get/set的数量，方法重载的数量，每个方法的代码行数，public、private和protected方法的百分比，每个类的耦合度（该类知道另外的类的数量），以及被注释方法的百分数。

度量可以通过一个报表界面提供，或者，更好的是，通过一个到电子表格的链接，类似于前面描述的项目管理链接。

SVG：矢量图形

为达到真正的、标准化的矢量图形输出/输入功能，UML工具厂商即将有一种选择。W3C的可缩放矢量图形(Scalable Vector Graphics, SVG)建议是可格式化图形的一种XML语法，成熟的1.0版本规范已经进展到“最后预览”阶段(3/3/2000)。一旦被完全认可，你可以留意HTML浏览器厂商什么时候在他们的下一代浏览器中提供支持。

为什么是SVG？因为一套用这种矢量图形格式输出的UML图表可以被链接到网页上。“over the web”的UML设计文档的读者将能够使用这种图形浏览技术，如在浏览器内缩放和平移，来更轻松地浏览一张大的UML图表。还有，和GIF格式图形相比，这种格式将戏剧性地提高通过web加载大图形的速度。请看今天Macromedia Flash的展示在浏览器中加载是如何之快，就可以证明这一点！

为了强调GIF图形和可缩放图形在出版环境中的强烈差别，我们准备了一个模拟，通过创建包括两个类图实例的Adobe PDF文件，其中一张是输入的GIF图形，另一张是矢量图形。你可以下载这个PDF文档并在Adobe Acrobat中观看。尝试放大到很高的水平如800%或1600%，然后比较GIF图形和矢量图形的结果。这个实验并非不切实际：你可能需要准备一张被缩放到一个可读性提高的水平的演示图。

下载GIF和矢量图形对比的PDF例子文件.

为了展望UML和SVG的未来，我们也准备了一个使用SVG在浏览器中显示类图的演示。为了观看这个演示，你必须首先为你的浏览器下载一个SVG察看器。我们推荐来自优秀的Adobe SVG站点的插件。这样你就可以观看用SVG显示的图形模型演示。

XMI：把所有东西捆绑在一起

对UML开发团体来说，对象管理组织(OMG)的XMI标准是最近最令人振奋的进展。XMI是一种有潜力最终允许在领先的开发工具之间无缝共享模型的交换格式。例如，与其在UML建模工具中书写脚本来创建报表，不如让用户简单地在开发时使用XMI输出该模型，然后引入到一种特定的报表书写工具中。事实上，这个范例将平等地、很好地应用到前面讨论过的特性：O-O度量追踪和项目管理。此外，自从XMI使用XML来表示模型信息，一批XML解决方案很快会出笼，例如为基于浏览器表示提供的XSL格式表和为搜索兼容性提供的XQL查询工具。

XMI标准是复杂的，在被广泛使用之前，它将需要时间来适应许多不可避免的兼容性问题(谁说标准经不起解释？)。但是，既然XMI是由IBM和Unisys等领先的工业巨头开发的，可以预期产品很快会出现。一直到用户团体不断要求厂商来驱动在UML工具中的XMI支持的需求。关于XMI的进一步信息，请看优秀的IBM网页。

作为XMI如何投入使用的例子，请看我们的项目， 转换XMI到HTML. 这个项目展示了XSL格式表如何能被用来产生UML模型的HTML翻译。

读者建议

如果您对在清单中未列出的UML建模工具新特性有什么想法， 我们将很乐意听到。

谢谢曾经把他们的UML工具评价标准Email给我们的读者。他们的原始建议在这儿。

Use Case 正解

Use Case 正解（答 olive, knight, builder, 3nt 和 think 关于 Use Case 的讨论）

发言人：Dr.OO

大家好，下面是我的观点，望能抛砖引玉：

1、好像不少人对 Use Case 存在误解，或者片面的认识。其实作为原爱立信公司的软件总设计师，Ivar Jacobson 对软件界最重要的贡献之一就在 Use Case 上（另一个重大发明是被电信界广范采用的 SDL），所以要真正理解 Use Case，推荐大家有机会读一下 Jacobson 的经典名著“OOSE”（即 Object-oriented Software Engineering - A Use Case Driven Approach）。

不能苟同 olive 的观点。

Use Case 绝不是锦上添花的东西，一方面它可以促进与用户沟通，理解正确的需求，另一方面它可以划分系统与外部实体的界限，是系统设计的起点，而最终应该落实到类和实现代码上。Use Case View 与 Logical View 应该由明确的相关性。UML 中从 Use Case 到类包的关联可以用依赖（或实现）关系描述，好像 Rose 2000 已经可以支持该功能了。

Actor 不是指人，而是指代表某一种特定功能的角色，因此同一个人可能对应很多个 Actor。Actor 是虚拟的概念，甚至可以指像外部系统和设备。理论上我们可以把一个软件系统的所有 Use Case 画出来，但实际运用时只需把重要的、交互过程复杂的那些画出来。

Use Case 是对系统行为的动态描述，它是 OO 设计的起点，是类、对象、操作的来源，而通过逻辑视图的设计，我们可以获得软件的静态结构。

Use Case 是 Jacobson 在设计世界上最大、最成功的 OO 产品——爱立信著名的 AXE 系列程控交换机过程中发明的，可见 Use Case 的重要性和实践意义。

2、OOAD 大部分情况下比结构化设计好。

我认为结构化设计是过时的东西，它强调软件的结构按照功能来组织，一旦功能改变，软件的结构就会不稳定。而 OO 设计把数据流和功能统一起来，因此我估计，IT 行业绝大部分（70-80%）的软件设计（包括数据库设计）可以采用 OO 方法，目前国外流行的趋势也是这样，剩下的少部分有特定需求的可能还会用传统方法。
另外在电信界，用有限自动状态机的 SDL 方法仍占绝大数，但现在 UML 和 SDL 出现了融合的趋势，而 Jacobson

则是幕后戏剧性的重量级人物。

3、Use Case 是可以分解的
最近 Jacobson 出了一本新书: Software Reuse - Architecture, Process and Organization for Business Success. （世界图书出版公司影印版，88 元)。很巧，去年底我在南京成贤街的小书店买到的，感觉如或至宝。该书通过一个网上银行的例子，对 UML 建模作了精辟的分析，大家可以从中了解到 Use Case 的正确用法。
书中在针对软件的分层结构，用了超系统和超用例（Superordinate Use Case）的概念。
分解一般在功能细化的时候进行，相当于子系统的功能分析。分解后当然还是 Use Case 图。
Use Case 驱动很好理解，因为 Use Case 分析总是迭代递增开发过程中每次循环的起点。

4、光有思想和方法论是不够的，最后还要落实到系统的实现上，即代码，这样才能前后贯通，从而对 OOA、OOD、OOP 方法有深刻的认识。这方面 Rose 的集成功能做的较为完善。

5、Rose 仅仅是 UML 设计的一种工具，事实上还有很多其他类似工具，包括嵌入式实时系统的 UML 设计。个人感觉 Rose 的可视化作的不错，使用较方便，就是耗内存，有点慢。据说国外有人用一个 ROSE 模型就有上千个类。大家可以参加 Rational 公司的 Rose 论坛，里面非常热闹，往往有问必答，包括 FAQ 和菜鸟级问题，最大的好处是可以了解世界先进水平的专业级用法。

6、最后，我不是 Rational 公司的。

#:--)

一些补充

发言人：Frank Gu

Actor 是指系统以外的，需要使用系统或与系统交互的东西。包括人，设备和其它系统。Use Case 应该覆盖系统的所有功能性需求，即使是简单的 Use Case，只要它确实描述了这类需求，就应该在 Use Case 图上画出来，并用 Use Case Specification 加以详细描述。Use Case 还是做开发计划，测试计划，设计测试用例的依据，所以 Use Case 一定要识别得充分而完整。

Use Case 与最后设计出来的类的关系是通过 Use Case Realization 来表现的。一般不画它与类包的依赖关系，画出来一定很复杂，且意义不大。Use CaseRealization 可以描述清楚类与类之间是如何协作来 perform 一个 Use Case 的。

USE CASE 多少的问题?

发言人 semuw

好象在北航的那本书中提到了 USE CASE 多少的问题,有人用 20-30 个左右,有人用 100 多个, Frank 的看法大概是 100 多个的那种.从需求定义,系统边界的角度,似乎应该是这样,那么为什么会有少定义 USE CASE

的情况呐?

是否应该定义主要的,用户最关心的,对系统结构有重大影响的,而对于简单的如代码维护等就不必划 USE CASE 图了. 或者,都划,但简单不用 Use Case Realization 来详细描述.

Re: USE CASE 多少的问题?

发言人 Frank Gu

系统的规模不一样，Use Case 的数目当然就不一样，怎么能硬性规定呢？小的系统，可能 7，8 个就可以了，一个大系统，比如一个 ERP 系统，恐怕 100 多个都不止吧？Use Case 的数目还和 Use Case 的粒度有关。粒度大了
数目自然就少了。

我还是认为 Use Case 应该找全，简单的也要找出来。当然可以不用详细的事件流来描述，只写出简单的步骤。Use Case Realization 也应该做，否则怎么保证类找得全呢？类的属性和方法都找全呢？

我对 Use Case 的理解。（也对下面关于 Use Case 的讨论发表看法）

发言人 Frank Gu

Use Case 的用途。

Use Case 分析是分析系统功能需求，确定系统边界的手段。Use Case Model 是系统需求分析阶段的成果之一。Use Case View 是 RUP 定义的系统架构 4+1 五个视图的其中之一。Use Case 是系统分析和设计阶段的输入之一，是分析和设计，制定开发计划、测试计划、设计测试用例的依据之一。Use Case 不仅仅用来和用户交流，也是开发人员之间进行交流的重要手段。

Use Case 是非形式化的吗？

怎么样叫形式化，怎么样叫非形式化呢？Use Case 图中每个符号都有确定的含义，Use Case 图的绘制也有明确的画法和规定。也许很多人对 Use Case 的划分和 Use Case Specification 的写法有不同的看法，但至少在一个项目中应保持一致，而不能每个开发人员各写各的。这方面自由度是不应该很大的。这算形式化还是非形式化呢？

Use Case View 和 Logistic View 的关系。

Use Case View 的主要组成部分 Use Case Model 是需求分析阶段的成果，分析和设计阶段的输入，Logistic View 主要是在分析和设计阶段完成的。由此可见 Use Case View 和 Logistic View 的相关性非常强。Use Case View 是 Logistic View 的设计依据。

在 Use Case 和分析与设计类之间的桥梁就是 Use Case Realization. Use Case Realization 就是用互相

协作的对象类来描述一个特定的 Use Case 在设计模型中是如何实现的。(A use-case realization describes how a particular use case is realized within the design model, in terms of collaborating objects。摘自 Rational Unified Process)。

一个还是多个 Actor?

Actor 是系统以外的与系统交互的人或物。就人这一方面来说，Actor 不是具体的某个用户，而是刻画了一种角色。在现实世界中可能是某个岗位或职位，等等。这种划分是客观存在的。比如一个财务系统的用户可能就有出纳，会计，财务主管等。他们会使用系统做不同的事，自然就会有不同的 Use Case。所以不分青红皂白，只用一个 Actor 是不行的。如果 Actor 与现实世界相符，那用户非但不会搞糊涂，反而会对 Use Case Diagram 理解得更清楚。

只要 Word 就够了吗?

Use Case Diagram 不是可有可无的。在一定条件下，一幅图被长篇大论更能说明问题。Use Case Diagram 可以使我们对 Actor 与 Use Case 的关系，Use Case 之间的关系，系统的边界有一个总体的认识和把握。Use Case Diagram 和 Use Case Specification 缺一不可。就好象我们既要有 Class Diagram 又要有 Class Specification 一样。而如果是开发一个大型项目，有上百，数百，甚至上千个 Use Case 时，我们还需要专门的需求管理工具，就更不是一个 Word 就能胜任的了。

其实关于 Use Case 如何划分，在 www.umlchina.com 中有好几篇文章，都是写得很好的。在 Rational Unified Process 中也有具体的指导。

定时器在用例图里是不是一个角色？

Straybird：

假设一个用例是“查询设备状态”，角色是“设备”，每隔一定时间系统开始执行“查询设备状态”这个用例。但是用例应由角色启动的，而“设备”这个角色只是被动的，这个用例实际上是定时器启动的，定时器应当是角色。可是定时器是系统边界以内的对象，不应是角色。
是不是可以把时间看作角色，定时器作为对应角色的主动对象，这样理解是不是就符合模型了？
刚开始学习用 UML 做分析，许多地方不知理解的对不对，希望大家答疑解惑

worship：

我觉得定时器应是一个守卫条件，不应该看为一个角色。“查询设备状态”用例由其他角色激活，然后启动计时器"守卫"对被动角色“设备”的访问。

Ms.OO：

角色应该是时间。角色有几类：用户，外系统，时间 ，所以，角色应该是时间 。

“是否角色”与“是否在系统内表示”并无关联，例如，一个顾客上网站买东西，顾客是角色，但系统内也会有“顾客”类。
角色是角色，它在系统外部工作，它可以映射系统内的某个类，也可以什么也不是

straybird：

多谢两位指点。从状态图上理解，把定时器作为守卫条件就很合理了，这样启动系统时，用例激活，守卫条件满足时，执行用例，启动系统的操作员是激活用户的角色，是这样吗？
把时间作为角色也应该是合理的，只是这个角色有点抽象，呵呵。

工具

怎样把 Rose 中的 Use Case Diagram 导到 Word 文件中，作为一幅图片

coder2designer：

请问，怎样把 Rose 中的 Use Case Diagram 导到 Word 文件中，作为一幅图片

ken_xu：

It’s a piece of cake.
1. In use case Diagram, "select all"
2. In use Case Diagram, "copy"
3. In word, "paste"
4. You did it.

coder2designer：

我也是这样做的，但是中文会出现乱码！

fact-finder：

中文会出现乱码解决！

步骤:Rose 菜单->Tools->Options->General
Default Font->字符集 Gb2312,字库，大小
Documentation Windows Font->字符集 Gb2312，字库，大小
应用/确定
保存退出

重新启动 Rose，打开文件
Soda 和拷贝粘贴中文就会正常

Mrshiwei：

试一下这个方法:
使用“Web Publisher”这个 Add-in，可以将整个模型输出成 Web 页，
其中的图形可以很方便地导入 Word。我试过，没有中文问题。

我可不可以不用 RO$E?(并非贬低 Rational)

我可不可以不用 RO$E?(并非贬低 Rational)

发言者:mark

公司最近要上市，管 license 很紧，其实我本来就不喜欢 RO$E，于是刻的 ROSE98 let it be 吧。其实我应该可以不用 ROSE. 因为我可以搞到 argo.. 但是破 JDK 太大啦。还有 OTM...算。我还是用 visio. 或者 word..最后我还有白纸和黑板!

人应该有头脑的使用工具，而不是被工具奴役。

For RoundTrip Engineering.

发言者:Rayman

如果只是用来画图，当然不必用 Rose. 但是要是想作 RoundTrip Engineering，一个好的工具是不可少的。不知有没有比 Rose 更好用的工具？我试过 Together Control Center,用起来比 Rose 方便，但功能不够强。但 Rose 用起来太烦。而且我用的时候还是问题多多。

Re: For RoundTrip Engineering.

发言者:mark

同意。我只是不希望被工具左右。 就好象多年前，Turbo C 的使用一样。没有真正了解什么是真的 C 语言，我觉得如果读 K&R 更好。有的时候我是偏激。呵呵。但是觉得很多商业化的东西，容易掩盖事物的本质。

选择好的工具很重要。

发言者:Rayman

就像写信和打电话一样，同样是交流的手段，工具不一样，效果差得远。分析阶段的模型定型后，变化不会太多。但设计模型在迭代过程中变化是常有的事，而且经常会是先改代码，后更正模型。如果没有工具自动完成这个工作，很容易超成模型与代码不一致。还有许多相应的文档如类的说明等，如果靠手工维护也将带来很多的工作。我非常支持尽量减少中间产物的（手工）生成。如各种报告，如果能够自动生成将能提高效率，减少工作量。ROSE 的一系列工具（声称）能很好的符合这个目标。

但在作为学习时，我们不应把工具作为目标。ROSE 只是用来实现 UML 的工具，UML 也只是表达 OO 概念的方法。真正的重点是把握 OO 的分析方法和设计方法。国内的论坛/主页大多数都在讲 UML/ROSE，好像掌握了它们就会系统分析和设计了。

随便说的!

发言者:Newer

一直以为学习是学门语言,可是现在我不这么认为了,会 VC,BC,VB,VJ,PB,ASM.....没用的,只能成为编程的机器而已,人活到那个份上,怎一个"可怜"了得.

一直在找一个工具,让它能把我们的思想转换成代码出来,有时候怀疑会不会有这种工具,假如让我实现该工具,我从哪里下手?

现在类似 rose 的工具很多.但是 visio 绝对不能和 rose 相提并论.激光也有热量,但是不是用来烧饭的.一样的道理.

再同 newer 谈

发言者:mark

其实我的话更缺乏逻辑。望原谅。:) 学习当然不是学习语言。我自己认为一个好的软件开发人员，操作系统，语言，网络(include C/S)，数据库，工程化 的知识都是必不可少的。但是语言还是要学的。

可悲的是，我们很多时候都不知道什么是学习语言，学 Turbo C 但是不知道什么是真的 C 语言(为什么那时候老师不教 K&R),学 VC 担不知道什么是真的 C++(Thinking in C++是本好书)，而更可悲的是把 DOS 认为是操作系统，认为 Foxpro 是数据库，而现在很多人还没有理解什么是软件工程，很多人，把软件工程和过于严格的过程控制划成等号。

另外我同意编写代码并不是一个高级的工作。但是我觉得一个再高级的软件开发人员，也要会编写代码。非常不同意把软件开发工厂化，划分成高级，中级低级程序员的分别。XP 的人性化会让角色划分更加人性化。软件开发再相当长时间会是高智商的工作。很多不能同 ROSE 比的都只是价格。我觉得。呵呵。ROSE 白给我用，我用他，那么贵。我就不用他，如果激光很便宜，但是炉子很贵，我就宁可用激光烧饭。呵呵。玩笑。

服　务

国内软件工程书籍一览
（按推荐强烈程度排序）

OO/UML

《UML 参考手册》，Ivar Jacobson，James Rumbaugh，Grady Booch。姚淑兰，唐发根译。机械工业出版社，2001。★★★★★

《设计模式 可复用面向对象软件的基础》，机械工业出版社，2000。★★★★★

“Software Reuse”，Ivar Jacobson 等著，1997 年出版，世界图书出版公司原版引进。★★★★★

《Oracle 8 UML 对象建模设计》，机械工业出版社，2000 年 4 月。很实用。

《面向对象设计的 UML 实践》（影印版），Mark Priestley，清华大学出版社，2000。

《UML 对象设计与编程》，刘润东，北京希望电子出版社，2001。

《面向对象的系统分析》，邵维忠，杨芙清，清华大学，广西科技出版，1998 年 12 月第一版。

《实用面向对象软件工程教程》，Edward Yourdon, Carl Argila 著，电子工业出版社，原书 1995 年 12 月写就。

《面向对象分析与设计》，杨正甫，中国铁道出版社，2001。台湾人写的，每个方面都泛泛而谈了一下。想知道布什、雅寇森、云豹是谁吗，看看这本书就知道了。

《复杂系统的面向对象建模、分析与设计》，范玉顺、曹军威编著，清华大学出版社、施普林格出版社，定价 38.00，软件较旧、光盘带有欺骗性，但也有其自己的特色。

《UML with Rational Rose 从入门到精通》，电子工业出版社。

《可视化面向对象建模技术--标准建模语言 UML》，刘超，张莉，周伯生，北航出版，1999 年 7 月第一版，可谓是中国第一本关于 UML 的书。

《UML Programming Guide 核心设计技术》，希望出版社，2001。

《可视化对象建模技术》，[美]D.特卡奇 W.方 A.苏著，科学出版社。

《面向对象软件工程》，陈世鸿，彭蓉，电子工业出版社，1999 年 5 月第一版。

《面向对象的分析》，《面向对象的设计》，两本小册子，Peter Coad，Edward Yourdon 著，邵维忠等译，北大出版社，原书 1991 年写就。国内第一本关于面向对象分析和设计的书。

交互设计

《Web 可用性设计》，Jakob Nielsen 著，人民邮电出版社，2000 年。★★★★★

《软件创新之路--冲破高技术营造的牢笼》，英文原书名：The inmates are running the asylum，Alan Cooper 著，翻译：刘瑞挺 刘强 程岩，电子工业出版社，2001。★★★★★

概论

《软件工程－实践者之路》（第五版，影印），Roger S. Pressman，清华大学出版社，2001。前面版本的中文译本见下。★★★★★

《软件工程--实践者的研究方法》，Roger S. Pressman 著，黄柏素、梅宏译，机械工业出版社，1999。

《现代软件工程》，清华大学周之英编著，科学出版社。分上中下三册，上册为管理技术篇，中册为基本方法篇，下册为新技术篇。

过程

《软件需求》，Karl E.Wiegers 著，陆丽娜 王忠民 王志敏译，机械工业出版社，2000。
《程序设计实践》，Brian W.Kernigham 著，裘宗燕译，机械工业出版社，2000。
《小组软件开发过程 TSPi》，Watts s Humphrey 著，人民邮电出版社，2000。
《软件能力成熟度模型》，何新贵等著，清华大学出版社，2000 年 11 月。