2002－8（第16期）

David Van Camp

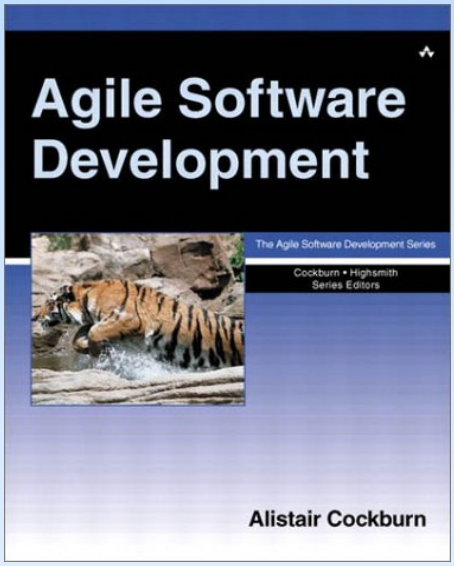
X-Programmer 非程序员
软 件 以 用 为 本

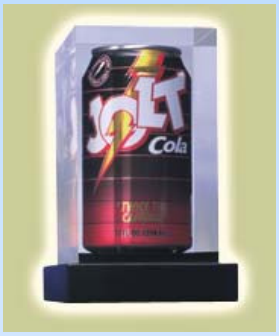
主编：davidqql
审稿：davidqql，think

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

http://www.umlchina.com/

《敏捷软件开发》





2002 Jolt Awards 获奖书籍

Alistair Cockburn

翻译：UMLChina 翻译组 Jill

我是一个好客人。到达以后，我把我的那瓶酒递给女主人，然后奇怪地看着她把酒放进了冰箱。

晚餐时她把酒拿了出来，说道，“吃鱼时喝它，好极了。”

“但那是一瓶红酒啊。”我提醒她。

“是白酒。”她说。

“是红酒。”我坚持道，并把标签指给她看。

“当然不是红酒。这里说得很明白...”她开始把标签大声读出来。“噢！是红酒！我为什么会把它放进冰箱？”

我们大笑，然后回顾我们为了验证各自视角的“真相”所作的努力。究竟为什么，她问道，她已经看过这瓶酒很多遍却没有发现这是一瓶红酒？

中文译本即将发行！

David Van Camp：模式、构架和 XP

查看评论

编注：

北京时间 2002 年 3 月 7 日上午 10:00-12:00，David Van Camp 先生作客 UMLCHINA 讨论组的聊天室。David Van Camp 是对象技术、设计模式、软件复用方面的专家，有 17 年的软件开发经验。经历了各种大大小小的项目，有录像租借店管理系统等小项目，也有 NASA（美国航空航天局）的哈勃太空望远镜控制系统等大项目。他的模式网站是 Pattern Digest。以下是交流实录。由李文华翻译。原文链接。

dongyeye：您能否告诉我如何组织一个小型团队？但是这个项目很大而且需要很长时间，就像一个政府项目。

davidvancamp：如何组织小型团队取决于你的目标，我需要你提供更多信息。至于组织大型项目，应该围绕层次化目标进行组织。

umlchina：NASA 不是一个军方部门吗？

davidvancamp：不，NASA 是一个单独的机构，我相信它跟军方有关，但是基本上被当作一个民用机构。

idlecrook：Dave, 如何实现一个软件过程？

davidvancamp：软件过程应该依赖组织，目标，文化——没有单一的解决方案。

415918：Dave, 以您的经验，您能就软件复用给些建议吗？

davidvancamp：对于复用，我能给出大量的建议，但是需要更具体的问题。每个项目和组织都有特定的一系列的文化、业务目标，利害关系等等。在实现软件过程和复用策略的任何实践中，这些都需要考虑在内。

dshenqw：如何只在维护时复用？

davidvancamp：在维护中复用，在我看来，是个大难题。最好在开发中达到复用。然而，当完成一个项目的时候，组件挖掘（例如寻找现成的代码来复用）是一条很好而且实用的途径。

gigix：但是，如果你想“在开发中实现复用”，你将可能陷入“过度设计”。您如何看待这一点呢？

davidvancamp：过度设计——取决于你使用的方法和你对复用的重视。

dongyeye：当你更新你的版本，你应该复用许多组件，否则，你将浪费很多时间重复工作。

davidvancamp：一个大型的开发过程，起始于分析，并最终通过部署完成，在这种项目中我一般推荐综合多种途径来达到。成功的方法是通过高层的业务分析确定关键业务对象，然后建立一个小组来开发这些可以在未来复用的组件。可是，在多个项目的开发和调节中，我更倾向于基于重构的方法。

415918：在软件过程中，那个阶段对软件复用影响最大？

davidvancamp：从分析到最后配置，开发中的各个阶段对软件复用都有影响。

dongyeye：当您碰到麻烦的顾客，您如何做呢？

davidvancamp：您说的“difficult customer”（ 麻烦的顾客）是什么意思？

dongyeye：在中国，“difficult customer”就是像政府客户那样，他们的计算机水平很差，他们不太了解信息系统。

davidvancamp：尽量使用客户的词汇与他们进行交谈，尽量理解他们所关心的和所需的。避免复杂技术术语（行话）。

homemoon：您能谈谈关于软件设计和技术领导方面的问题吗？

davidvancamp:：我需要具体的问题——软件设计和技术领导是两个巨大的话题。

gigix：客户们总是不能清楚地表达他们的需求。

davidvancamp：要帮助客户更好表达他们的需求，尝试用例和从 XP 中借鉴的“计划游戏”。——这些在实际中效果很好。

huhu71：对于大多数开发者，如何成为一个客户应用的专家也是个问题。

davidvancamp：客户们不应该关注技术细节——你必须引导他们进入分析的过程，以及什么是/不是技术上可能的和是否切合实际的。

gigix：您提到了 XP—您如何看待它呢？

davidvancamp：我采用 XP 多年了，它是最佳实践的集合。

huhu71：您认为 XP 只是针对资深的专业开发者吗？

Davidvancamp：绝对不是。对一个混合着有经验者和初学者的项目甚至更好。我提倡在所有的项目中采用 XP 的大多数实践。

gigix ：“成对编程”也是最好的实践吗？

davidvancamp：对于成对编程，我有着复杂的感情——我不强迫给开发者，可能的话也允许他们自然的结对。

gigix：结对必须紧挨着坐在同一台计算机前吗？

davidvancamp：我并不认为紧挨着坐在计算机前是真正的 XP 方法。

idlecrook： 什么类型的项目适于使用 XP?

davidvancamp： XP 或许并不适合维护或者小规模改进工程。可能理想的情况是新软件开发。

415918：您如何看待 RUP?

davidvancamp：RUP 不错，但不是非常完美。我采用了 RUP 的部分要素并进行扩展，例如增加 XP 实践。

supershan：您能谈点关于.Net 和 java 的吗？

davidvancamp：我还没用过.Net---没有发言权。

415918：以您的观点，什么是软件复用的关键技术？

davidvancamp：复用的关键技术——你的大脑。一个好的复用库和 OO 建模工具也是非常重要的。

gigix：您如何看待设计模式呢？他们可以改善复用吗？它会鼓励过度设计吗？

davidvancamp：设计模式是优秀的教学工具，能够帮助你澄清设计决策，的确很棒。但是如果对它过于依赖则会导致过度设计。

415918：您认为 XP 很棒吗？

davidvancamp：XP 比什么更棒？

415918：XP 比 RUP 更棒吗？

davidvancamp：如果你问 XP 针对 RUP——恕我直言，他们并不冲突——很容易结合在一起。

gigix：你说 XP 和 RUP“可以很容易结合”，真的吗？你可以介绍这方面的情况吗？

davidvancamp：如果你看看我写的《最佳实践》（www.captial.net/~dvc）——在一个大公司的一个项目中我把这些实践同基于 RUP 的环境结合起来了。

shaojl：在哪里可以找到《最佳实践》？

davidvancamp：我的文章和陈述在 www.capital.net/~dvc 可以找到。

415918：Dave，我常想，模式和框架是复用技术很好的表述，你是否如此认为？

davidvancamp：是的，连同组件——它们都有优点和不足。参看我的文章和陈述。UMLChina 也有我的主页连接。

colinshen：你可以告诉我们什么样的项目适合 XP 吗？

davidvancamp：新软件开发我最推荐 XP，特别是使用良好的面向对象语言（Java,C++,Smalltalk 等等）的项目。对于大的项目和/或许多相关项目，我推荐加入部分 RUP 或者其它更大规模的过程。至于复用，在一个大的组织，RUP 必须加以改进。不久我可能就此发表一篇文章。

javalover： davidvancamp 先生，我阅读了许多关于用例的书籍，但是仍然对它感到很疑惑。你可以就此说些什么吗？

davidvancamp：您对用例有什么疑问？用例尽量展示了系统特征和使用系统的用户之间的关系，并且提供全面的实施步骤。

paddychen：我觉得界定参与者非常困难。

davidvancamp：参与者代表了使用系统的用户承担的角色（以及可能与系统发生作用的其它外界系统）。避免细节名词，选择参与者时，专注于执行的角色。

gigix：Dave，您喜欢那种语言？

davidvancamp：最近，我首选语言是 Java，但是我非常喜欢 C++。

shenqw：关于多个项目中复用：复用什么？难道只是一些函数和过程？

davidvancamp：复用可能聚焦于所有有价值的地方，但是高层复用块更具价值。比如，像 EJB 那样，或者那些非常重要的处理准则。

liaofe：我如何大规模地使用设计模式呢？

davidvancamp：设计模式提供可复用模板，这些模板告诉你如何在一定层次上组织系统。构架模式则提供更高层视图。我说的对你有帮助吗？

shaojl：我是用 RUP 和 Powerbuilder，你可以给我一些建议吗？

davidvancamp： RUP-PB—我需要更特定的问题。两者结合使用没有任何障碍。

chi001：你认为 CMM 适合中国的软件项目吗？

davidvancamp：当 CMM 的目标符合你们组织的目标时，采用 CMM 是可行的。每个项目都处于一定的 CMM 层次，知道自己所在的层次是提高开发中可重复性和可靠性的关键。

gigix：我对构架模式感兴趣，您能介绍一下吗？

davidvancamp：我向大家推荐“Patterns of Software Architecture”一书。我现在身边没有，不过我相信 Addison－Wesley 出版了。

liaofe：我有，但是不全。

415918：是否存在通用的途径去构架和设计一个 J2EE 应用？

davidvancamp： J2EE—不，没有通用的途径。但是现在有许多书籍是关于它的。Java 设计模式也能够帮助你。

415918：我认为在开始一个项目前，J2EE 构架就很好地进行预定义。我所能做的就是根据软件质量指标决定组件框架的结构。

gigix： 为什么您喜欢 C++甚于 Java 呢，是因为 C++的自由吗？

davidvancamp：我的回答看来被误解了——C++对比 Java—实际上，由于简洁和可移植的原因我更喜欢 Java。

liaofe：我正试图把设计模式应用在 OA(办公自动化系统)中,您能给我一些建议吗？我现在正忙着写关于 UML 和 OA 的论文，但是，我发现用 OOA（面向对象分析）的观点来分析 OA 系统（使用 domino/notes 平台）是困难的。您能给我一些建议吗？

davidvancamp：你正经历什么困难？比如分析 OA 系统。

liaofe： 哦，我的意思是使用 domino/notes 比较困难。

davidvancamp：我没有为 Domino/Notes 建模的经验，但是我建议使用 UML 活动图建模业务工作流，就像目前它作为一条初始分析途径一样。

liaofe：先生，你认为用 UML 描述工作流有价值吗？因为它可以用 WFDL(工作流定义语言)描述？我现在要做我的 UML 论文。

davidvancamp： 活动图可以用作业务流建模。

colinshen：可以告诉我们如何开始设计 NASA 项目吗？

davidvancamp：NASA 项目（争取进行中的项目）始于分析（非正式）项目的需求、目标、问题，形成我们的看法，并且依据我的经验得出一条看起来适当的途径。随后，我设计一个初始框架，并组织一个开发团队进行开发和目标培训。

shenqw：在您的 NASA 项目中，一共有多少开发者？

davidvancamp：整个 NASA 项目最多有 160 名开发者参与。我的团队有大约 10 名开发者，我还参与指导其它团队并为他们作咨询。

gigix： 75%的中国公司少于 50 人，CMM 和 RUP 合适吗？也许敏捷开发更适合中国公司？

davidvancamp：对任何项目 RUP 都需要进行定制和缩放。你可以使用它的部分元素而不管项目大小。或许，你需要根据你的目标和需求对它加以改进。所有的方法都应该是“敏捷”的：)。在任何软件过程改进中，最重要的是能够适应需求变更并且监控结果。

colinshen：在你的 NASA 项目中，使用 XP 还是 RUP 过程？

davidvancamp：在 NASA 项目中，我采纳了 XP 的许多方面。那个时候 RUP 还没有出现，但是这些大规模的过程是基于需求和经验不断改进的。

gigix：啊？在如此大的项目中用 XP？能够很好控制吗？

davidvancamp：XP 类的实践过去用于团队层次，对，他们发挥了很好的作用。

simaetin：据我所知，XP 对开发者能力有很高的要求

davidvancamp：XP 提倡有经验的开发者和初级开发者结对编程。我一直这样非正式地执行。我也利用每周培训课帮助训练队员。

gigix： 至于重构呢？我认为它是 XP 最重要的部分。

davidvancamp：我极力推荐重构——它是开发优良系统的关键。实际上，如果缺乏重构，一个相当规模的项目能够完成是不可理解的。重构是我职业生涯中见过的最好的名字——我想对其它许多人也同样如此。

paddychen：你如何控制可能在 XP 过程中出现的缺陷？

davidvancamp：缺陷——经常与之交结。启用复杂的机制去鉴别缺乏组织的代码并安排代码复审。在我的《最好实践》的文章中就如何运用代码韵律提出了一些看法。同一文章中我也提供了参考 NASA 的有关代码韵律的论文。

gigix：重构时您采用那种测试工具和集成工具？

davidvancamp：目前，我正在使用 Junit，但是过去，我构建自己的工具。

cajan2： 为什么 C3 项目中止了，据说是 Beck 自己不能成功完成该项目。

davidvancamp： 除了公布的信息之外，我不知道任何 C3 项目的成果。我怀疑“Beck 不能完成”项目的说法。

shenqw：NASA 项目使用黑箱/白箱测试吗？

davidvancamp：NASA 运用了他们所能运用的一切方法去测试。NASA 也采用冗余的设计和实现形成运行时的内部一致性检查。

shenqw： "冗余设计/实现","运行时一致性检查"?你能给更详细的信息吗？

davidvancamp： NASA 分配两支独立的队伍去设计和开发那些相同（关键的）的特性，而两支队伍开发的结果在运行时必须保持一致，否则就报告错误。

shenqw： 谢谢，但是我想：1， 系统只是报告错误，为什么不是 3 个团队， 2：然后可以得到更多有用信息 ：）

davidvancamp：好的，1：1 比 2：1——两者都比通常的 1：0 好。但是，你到了一个太简单太重复的地步。

415918：如您所知，我们大都熟悉面向对象编程，但是缺少面向对象分析和设计的知识，您能告诉我们如何获取这些知识吗？

davidvancamp：有许多好的书籍和文章。可以看看我的主页和 objectmentor.com(Bob Martin)和 korson-mcgregor.com。cetus 是一个提供面向对象信息的优秀资源——如果你需要的话，我可以提供链接。

simaetin：程序员进行逐行跟踪的测试吗？

davidvancamp：我不明白你说的“逐行跟踪测试”是什么意思？

simaetin： ″逐行测试″意为测试对象内部状态而不仅仅是其实现的接口。

davidvancamp：我一般不测试内部状态——我对如何满足公共规则更在行。代码复审通常足够评估实现。

415918： 在开发软件超过 20 年后，你的基本经验是什么？

davidvancamp：我已经开发大约 17 年了，不是二十多年，我有很多经验——更具体一点？

qphu：嗨，我有个关于 CBD 的问题，CBD 在实践中流行吗？

davidvancamp：EJB 现在获得了比 CBD 更多的青睐——它们都很优秀，而且工作时充满乐趣。CBD（component based development）在美国注定要流行。

415918：EJB 的缺点是什么？

davidvancamp：必须仔细考虑 EJB－－在业务对象被许多系统复用的情况下它们是最好的。企业管理费用和过于复杂是其主要缺点。

liaofe：我发现很难找到特定方面的设计模式，不是指《设计模式》一书中的 24 个。

davidvancamp：寻找设计模式——我相信关于这方面的出版物很多——查询书籍资源。我最近可能在我的主页公布一个《模式摘要》。

gigix：在我的主页—— http：//gigix.topcool.net 上也有模式的介绍。

simaetin：请告诉我你的网站地址。

davidvancamp： 我的网站是 www.capital.net/~dvc

qphu：我阅读了一些 CBD 的书籍，但是没有关于组件的一般概念。

davidvancamp：论及 CBD 的书很多——如果你是一个 Java 开发者，一本 Java EJB 的书籍可能对你帮助很大。我不确信，但是 Bob Martin 的新书可能就 C++论及 CBD。否则，我建议看看最近出版的书籍。

simaetin：在美国，COM 技术状况如何？

davidvancamp：我不使用 COM。在我看来由于可移植性的原因，越来越少的项目特地采用 COM。

qphu：David，我怎样才能找到基于组件软件的例子。我想得到直观的感觉。

davidvancamp：组件可以从许多地方购买——其中最大的一个是 www.componentsource.com。

qphu：是不是没有免费版本？开放源码现在已经非常流行，免费组件如何？

davidvancamp：自由软件很好。但是许多美国公司担心维护问题，所以不使用自由软件。

gigix：我有一个关于模式的网站，请问你允许我们使用你的内容吗？我的网站地址是 gigix.topcool.net

davidvancamp： gigix - 欢迎你连接到我的主页。

gigix：我能够翻译你的文章吗？

davidvancamp： gigix—好的，欢迎免费翻译我的文章，只要提供到你翻译文章的链接就行。

gigix：好，没问题，我非常喜欢您的“复用轮子”。

415918：你喜欢在基于 Web 的应用中使用框架吗，例如 Expresso，Struts?

davidvancamp：如果我能确定多个应用的公共构架需求，我就开发这个框架。很多时候都这样。我推荐基于框架的开发方法。

simaetin：但是当有新的需求的时候，框架总是成为整个应用的瓶颈。

davidvancamp：好的框架设计不会成为处理的瓶颈，尤其当框架是 1 到 3 个项目的公共部分并且采用重构和共享代码的方法进行开发时。

415918：如果我们必须自己开发框架，如何获取框架呢？

davidvancamp：设计和构建框架与构建应用相似，但是，专业技术、测试和文档的层次都更高。一般来说，如果你能够找到一个符合你需求的框架，从上述原因以及费用的角度，你应该尽可能购买。

cajan2：如果需要一个新的框架，而且基于此框架的应用很急迫，如何协调项目进度。

davidvancamp：基于框架的方法有个最大的问题就是设计者预先努力创建完美的框架，这是错误的。仅仅设计和实现你所知道的一到三个当前项目的需要并调整开发。做到这一点需要智慧和经验。另外，开发两到三个密切相关的产品作为尝试，并且从中慢慢重构框架。

qphu： david，你是否认为如果使用组件构造软件会更加快捷呢？

davidvancamp：我不确定你想问什么？——开发组件的代价取决于组件规模和复杂度。

paddychen： David，你认为文档优先于编码吗？

davidvancamp：我做编码、测试、重构周期前的高层设计和文档工作。

qphu：有如此多的可靠性模型，他们都有实际的应用吗？

davidvancamp：我并不采用任何正式的可靠性模型。

gigix：几乎每种语言都跟随着某个平台，你对此有何看法？这是有害的吗？

davidvancamp：语言＋平台——取决于如何实现。Java 很好，但是也有问题。我没有使用过 Forte，但是听说过它的优点——对我来说主要问题是拥有它。

qphu：嗨，david，你对软件可靠性有什么看法？它依然和六十年代那么重要吗？

davidvancamp：软件可靠性是个巨大的话题——这不是我的专业。在我的《最佳实践》一文中我的确提供了一些建议。

cajan2：我参与了一个使用新框架的应用程序项目，在框架中存在如此多的缺陷，应用开发者感觉非常痛苦。

davidvancamp：设计不是很好的框架是可怕的，我为你感到遗憾。

simaetin：有个与技术没什么关系的问题：在你需要更多时间进行代码复审的时候，怎样处理来自老板的压力，他并不是很了解技术。

davidvancamp：处理经理和客户不能很好沟通的问题往往非常困难。我能提供的最好建议是努力“立足于他们的观点”，重视他们的所见，并且尽量用他们的词汇表达你的问题。另外就是使用“计划游戏”让他们参与进来，从而更好地理解系统。XP 谈到了促进客户参与开发的方法。基本原理就是你要努力理解客户的问题和想法，学会用他们的语言跟他们交谈。非技术人员总是不能很好理解技术问题，业务人员只懂业务。例如，如果你能说服他们听从你的建议，这将节省成本，提高投资回报（ROI），从而你将获得客户重视。对于我们技术人员来说，也许与非技术背景客户沟通是最困难的问题，但是，如果我们希望影响他们的选择的话，必须学会有效处理这类事情。

seeseax：你对面向代理编程/面向代理软件工程有什么看法？

davidvancamp：在面向代理开发方面我不是专家。

qphu：如何处理 CBD？什么技术是最重要的？是组件库吗？

davidvancamp：除了复用外，我不能界定其它有关 CBD 最重要的核心技术。当可用组件的数量大到无法管理，而你还没有组件库，这时组件库就显得至关重要。首先，建议放慢进程保证质量。

cajan2：在两个项目中有许多接口，但是前一个项目的人员并不总是准备与你合作，有时候只是因为前一个项目的人员离开，你就不能做任何事，他们说“我们没有问题，你完成你自己的吧。”

simaetin：Java 和 C#最大的不同是什么？

davidvancamp：我不确定 C＃的需求会盖过 Java——尽管 C＃有语言上的改进，但是为什么是另一种 C 呢？Java 的确在可移植和简单上面优于 C。

cajan2：如何得到用例的层次？

davidvancamp：你说的“用例层次”是指什么？

cajan2：cockburn 说用例有三个层次：概要、 用户目标、子功能。

davidvancamp：你需要什么“层次”用例取决于你的需求和你觉得哪种层次适合驱动你进一步分析。我一般不那样做。高层的用例有益于理解业务环境（它们完成工作流/活动图）。系统级用例为你提供必须实现的外部接口良好的描述。但是，案例过于细化经常导致过于结构化的设计——不是面向对象——参看 Tim Korsons 在 Korson-mcgregor.com 上关于用例的文章。

simaetin：按照你的看法，既然我们被存在的模式、组件和框架等等包围，那么开发者的创造动机在哪里？

davidvancamp：模式是一般的复用方案，我不认为它们会窒息创造力。框架和组件是部分解决方案。对于创新方案还有大量空间。创新是对问题的求解——框架，组件和模式并不解决特定问题，它们只能帮助你去解决。

colinshen：你倾向于用用例捕获需求吗？

davidvancamp：是的，通过用例捕获需求是非常好的实践方法。

gigix：关于模式，你喜欢那些书？

davidvancamp：关于模式的书，经典的是 Gamma 等人的“Patterns of Software Architecture”，它非常好。另外，PloP 的也不错。其它书也有关于模式和面向对象的讨论。

simaetin：您是否能预言可能不久的某一天，面向对象范例将被其它方法所取代。

davidvancamp： OO 正在发展——CBD 是其中一个方向，敏捷开发是另一个方向。我希望不断进化。我确信 OO 必将为某种技术所取代。至于是什么，我没有水晶球（未卜先知的魔力）。

dingyilay：我不知道如何从嵌入式系统中得到用例，你能给我们一些有关嵌入式系统的建议吗？

davidvancamp：我还没有做过嵌入式系统编程。然而，嵌入式系统往往是同其它系统接口而不是同用户交互，所以，你的参与者应该是那些系统，而你的用例应该是那些交互。

415918：在美国，模式成为 00 技术的主流吗？

davidvancamp：模式不是技术——它们是简单的文档和教学技巧。当你逐渐熟悉模式，你就建立了一个模式“词汇表”。这使你能够更好地寻找模式和应用模式。这是一个学习过程。在美国和欧洲，许多人已经建立“模式讨论组”讨论模式。他们通常提出一个模式，然后大家评阅并相聚讨论。鉴别新的模式是一种艺术，不妨试一试，假以时日，你的能力会得到提高。鉴别一种新的模式，关键是要认识到其他人已经独立地找到了相同的解决方案并且得到确认。适当运用模式能够带来“良好”或改善的结果（不是其它途径能够带来的）。John Vlissides 已经写了很多关于寻找和编写模式文档的文章。

415918：我将试试，因为我认为模式是一种复用形式。

davidvancamp：模式的确是一种很好的复用形式。IBM 已经通过模式语言形式开发了一系列商务模式，以帮助确定业务系统的需求和选择适当的工具和体系结构。

dingyilay：我认为，很难从系统中得到用例和状态改变。

davidvancamp：状态变化应该用状态图描述而不是用用例描述。

cajan2：用例和模块之间的关联是什么？

davidvancamp： 用例和模块的关联——并非全部必要——取决于需求和解决方案。用例通常对应到一系列图。

notyy：关于 AOP，你有什么看法？

davidvancamp：我对 AOP 不在行，但是它列在我的学习计划中。

colinshen：如您所说，“OO 将要被取代”，您能告诉我 OO 的不足之处吗？

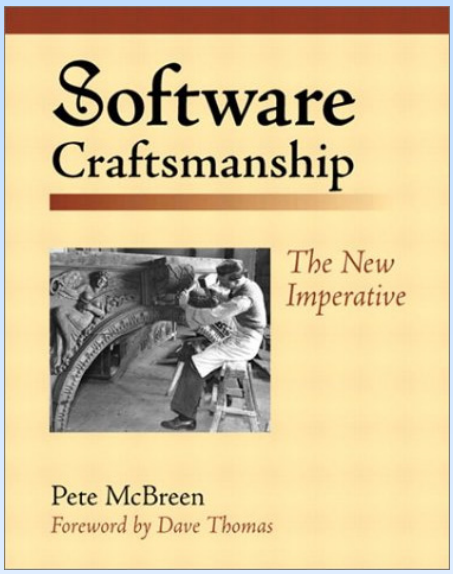
Davidvancamp：OO 的问题——当然，问题不可能没有。OO 很难，非常大的项目把 OO 推至极限——所以有了 CBD。

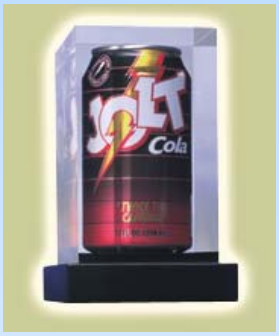
davidvancamp：OK—我们已经超时了——现在是我的休息时间了（美国东岸）。谢谢大家，晚安！

征稿

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

《软件工艺》





2002 Jolt Awards 获奖书籍

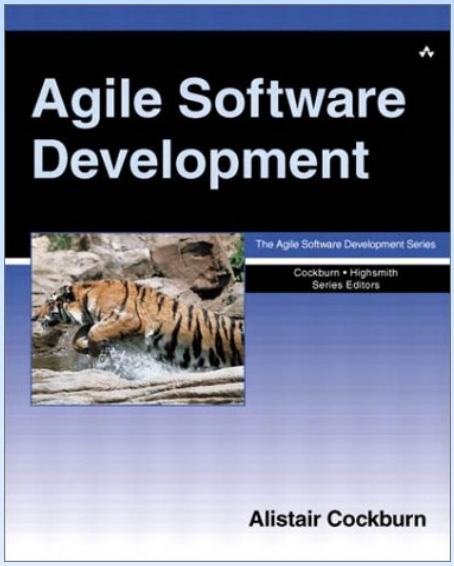
Pete McBreen

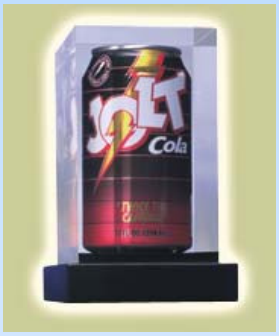
翻译：UMLChina 翻译组

工艺不止意味着精湛的作品，同时也是一个高度自治的系统－－师傅（Master）负责培养他们自己的接班人；每个人的地位纯粹取决于他们作品的好坏。学徒（Apprentice）、技工（Journeyman）和工匠（Craftsman）作为一个团队在一起工作，互相学习。顾客根据他们的声誉来进行选择，他们也只接受那些有助于提升他们声誉的工作。

中文译本即将发行！

《敏捷软件开发》





2002 Jolt Awards 获奖书籍

Alistair Cockburn

翻译：UMLChina 翻译组 Jill

我是一个好客人。到达以后，我把我的那瓶酒递给女主人，然后奇怪地看着她把酒放进了冰箱。

晚餐时她把酒拿了出来，说道，“吃鱼时喝它，好极了。”

“但那是一瓶红酒啊。”我提醒她。

“是白酒。”她说。

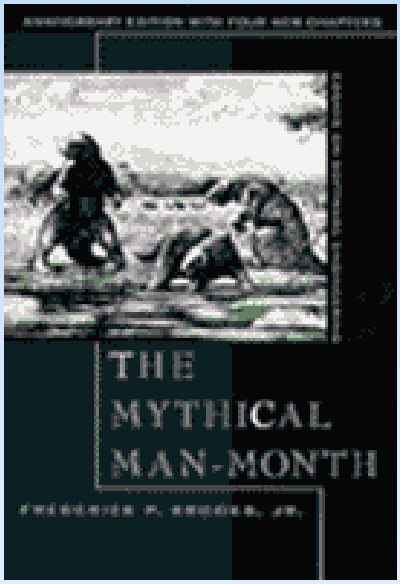
“是红酒。”我坚持道，并把标签指给她看。

“当然不是红酒。这里说得很明白...”她开始把标签大声读出来。“噢！是红酒！我为什么会把它放进冰箱？”

我们大笑，然后回顾我们为了验证各自视角的“真相”所作的努力。究竟为什么，她问道，她已经看过这瓶酒很多遍却没有发现这是一瓶红酒？

中文译本即将发行！

《人月神话》



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 Adams Wang

散文笔法，绝无说教，大量经验融入其中

在所有恐怖民间传说的妖怪中，最可怕的是人狼，因为它们可以完全出乎意料地从熟悉的面孔变成可怕的怪物。为了对付人狼，我们在寻找可以消灭它们的银弹。

大家熟悉的软件项目具有一些人狼的特性（至少在非技术经理看来），常常看似简单明了的东西，却有可能变成一个落后进度、超出预算、存在大量缺陷的怪物。因此，我们听到了近乎绝望的寻求银弹的呼唤，寻求一种可以使软件成本像计算机硬件成本一样降低的尚方宝剑。

但是，我们看看近十年来的情况，没有银弹的踪迹。没有任何技术或管理上的进展，能够独立地许诺在生产率、可靠性或简洁性上取得数量级的提高。本章中，我们试图通过分析软件问题的本质和很多候选银弹的特征，来探索其原因。

中文译本即将发行！

通俗而且深刻的软件开发思想杰作—UMLChina 推荐

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

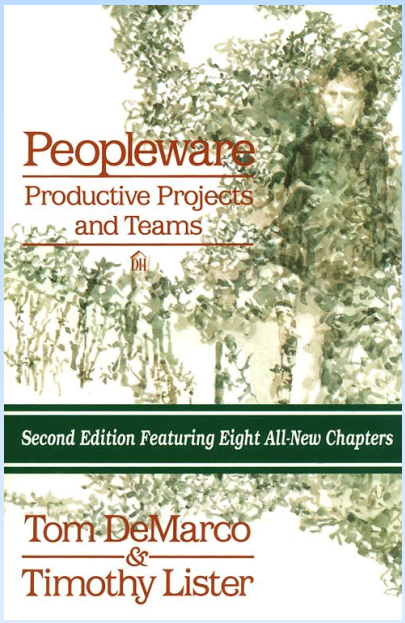
翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

面向对象技术在给我们带来好处的同时，也会增加成本，其中很大一部分是培训费用。经验表明，一个不熟悉 OO 编程的新手需要 3 个月的培训才能胜任开发工作，也就是说他拿一年的薪水，却只能工作 9 个月。这对一个拥有成百上千个这样的程序员的公司来说，费用是相当可观的。一些公司的主管们可能一看到这么高的成本立刻就会说"不能接受。"由于只看到成本而没有看到收益，他们会一直等待下去，直到面向对象技术过时。这本书不是为他们写的，即使他们读了这本书也会说（其实也有道理）"我早就告诉过你，采用 OO 技术需要付出昂贵的代价以及面临很多的危险。"另外一些人可能会决定启动一个采用 OO 技术示范项目，并观察最终结果。还有人仍然会继续在原有的程序上修修改改。当然，也会有人愿意在这项技术上赌一赌。

中文译本即将发行！

《人件》



《人件》第 2 版

Tom Demarco 和 Tim Lister

翻译：UMLChina 翻译组方春旭、叶向群

微软的经理们很可能都读过—amazon.com

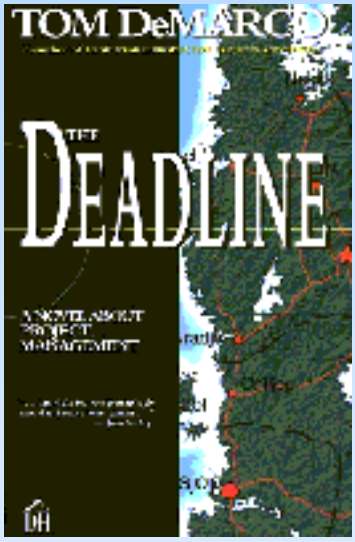
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

许多开发经理采用了类似的态度，他们竭尽全力地使自己确信没有人能够取代自己。由于害怕一个关键人物要离开，他强迫自己相信项目组里没有这样的关键人物，毕竟，管理的本质是不是取决于某个个人的去留问题？他们的行为让你感到好像有很多人物储备在那里让他随时召唤，说"给我派一个新的花匠来，他不要太傲慢。"

我的一个客户领着一个极好的雇员来谈他的待遇，令人吃惊的是那家伙除了钱以外还有别的要求。他说他在家中时经常产生一些好主意但他家里的那个慢速拨号终端用起来特别烦人，公司能不能在他家里安装一条新线，并且给他买一个高性能的终端？公司答应了他的要求。在随后的几年中，公司甚至为这家伙配备了一个小的家庭办公室。但我的客户是一个不寻常的特例。我惊奇的是有些经理的所作所为是多么缺少洞察力，很多经理一听到他们手下谈个人要求时就被吓着了。

中文译本即将发行！

《最后期限》



《最后期限》

Tom Demarco

翻译：UMLChina 翻译组 透明

这是一本软件开发小说

汤普金斯在飞机的座位上翻了一个身，把她的毛衣抓到脸上，贪婪地呼吸着它散发出的淡淡芬芳。文案，他对自己说。他试图回忆当他这样说时卡布福斯的表情。当时他惊讶得下巴都快掉下来了。是的，的确如此。文案……吃惊的卡布福斯……房间里的叹息声……汤普金斯大步走出教室……莱克莎重复那个词……汤普金斯重复那个词……两人微张的嘴唇触到了一起。再次重播。"文案。"他说道，转身，看着莱克莎，她微张的嘴唇，他……倒带，再次重播……

....

"我不想兜圈子，"汤普金斯看着面前的简报说，"实际上你们有一千五百名资格相当老的软件工程师。"

莱克莎点点头："这是最近的数字。他们都会在你的手下工作。"

"而且据你所说，他们都很优秀。"

"他们都通过了摩罗维亚软件工程学院的 CMM 2 级以上的认证。"

中文译本即将发行！

使用 UML 和 Rhapsody 开发导航控制系统

G.R. de Boer 著，ottafei 译

查看评论

摘要

本方案的目的是为了深入了解基于统一建模语言（UML）的开发工具 Rhapsody。另外，在 TMS-320LF2407 主板嵌入已开发软件的目标似乎已经超出了本方案的范围，因为 Rhapsody 是以实时操作系统（RTOS）为目标的，而 TMS 板却不符合条件。

本方案采用嵌入式系统面向对象快速开发过程（ROPES），此过程规定了“周期原型”，是一种即使产品没有完成也允许对分析模型进行早期测试的迭代方法。（Douglas, 2000）。我们选用 Rhapsody 为导航控制（CC）建模，目的是从中发掘 Rhapsody 这种开发工具的各种可能性。

在成功地深入了解了 Rhapsody 的同时，开发出了一个功能完整的导航控制（CC），同时也完成 Rhapsody 开发工具的全面知识的收集并形成了文档。我们建议进一步的研究主要放在改善维持速度的控制规则上，同时开发出一种能够绘制系统输出和易于事件插入的图形用户界面。此外还建议为 Rhasody 开发一种能够嵌入到非实时操作系统（non-RTOS）中的框架。

1 前言

本方案的目的是为了深入了解基于统一建模语言（UML）的开发工具 Rhapsody。使用 Rhapsody 能够设计和构建高水平的嵌入式软件。我们选用 Rhapsody 为导航控制（CC）建模，目的是从中找到 Rhapsody 这种开发工具的各种可能性。目前导航控制（CC）在小汽车上的应用已经非常普遍。因而非常适合作为控制系统在日常生活应用的案例。导航控制（CC）包括基本的循环控制和一些用户接口。导航控制在个人用小汽车系统中的位置如图 1.1 所示：

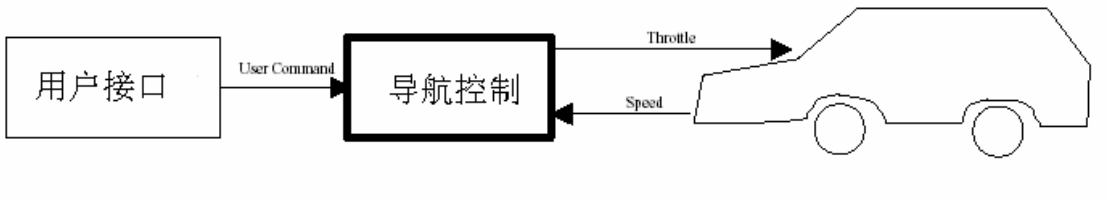


图 1.1 导航控制在个人用小汽车系统中的位置示意图

Rhapsody 为定义系统提供了用例，顺序图，类图和状态图等工具。用例描述了系统可能遇到的各种情况和主要功能。顺序图通过系统地定义系统中发生的事件顺序来了解期望行为。通过对象模型图可以定义模型的类结构。最后通过状态图来完成嵌入式软件的开发。

开发策略采用嵌入式系统面向对象快速过程（ROPES）方法（Douglas，2000）。

此方法规定了“周期原型”，是即使产品没有完成也允许对分析模型进行早期测试的一种迭代方法。ROPES 周期如下图所示：

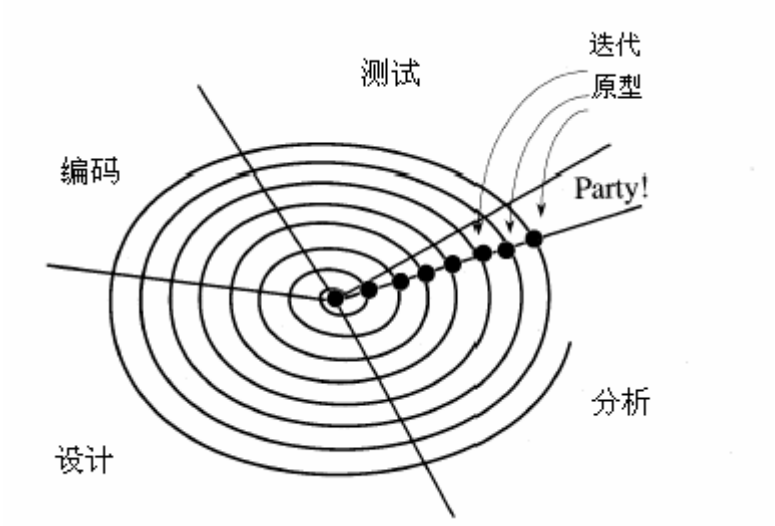


图 1.2 ROPES 周期（Douglas，1999，160 页）

第二章会详细解说这种开发方法。

起初，本方案的目标是使用 Code Composer 在可编程 TMS-320LF2407 主板上嵌入软件。最初项目策略如图 1.3 所示：

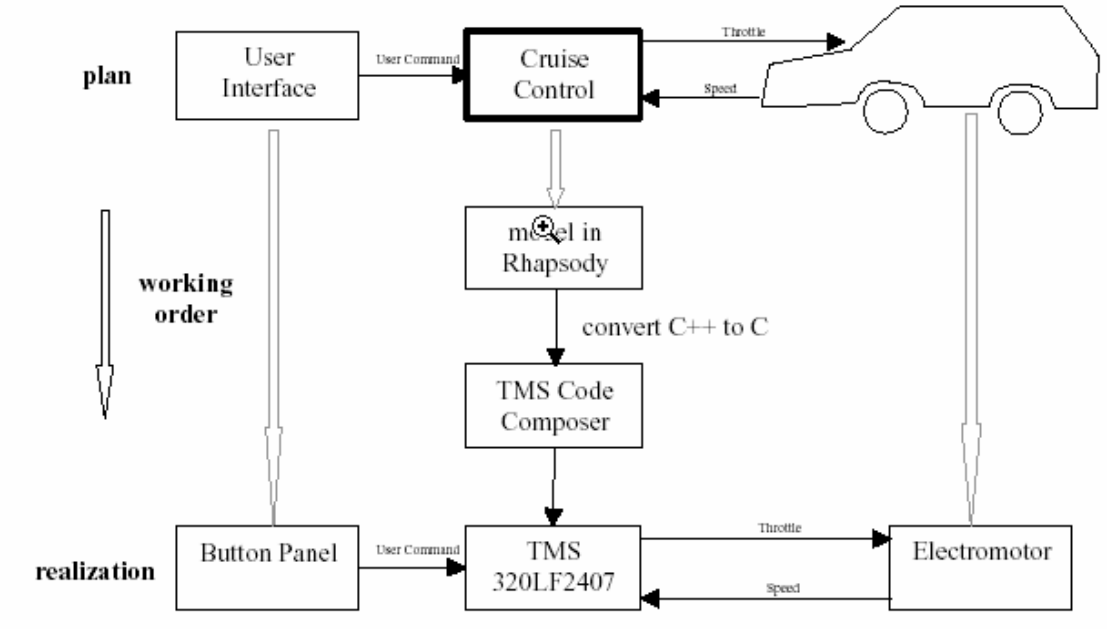


图 1.3 初始模型策略示意图

后来，由于 Rhapsody 的假定输出和 TMS 主板的特性不匹配而放弃了这个目标。Rhapsody 采用的是实时操作系统（RTOS），而 TMS 板却不符合条件。项目目标改变如图 1.4 所示：

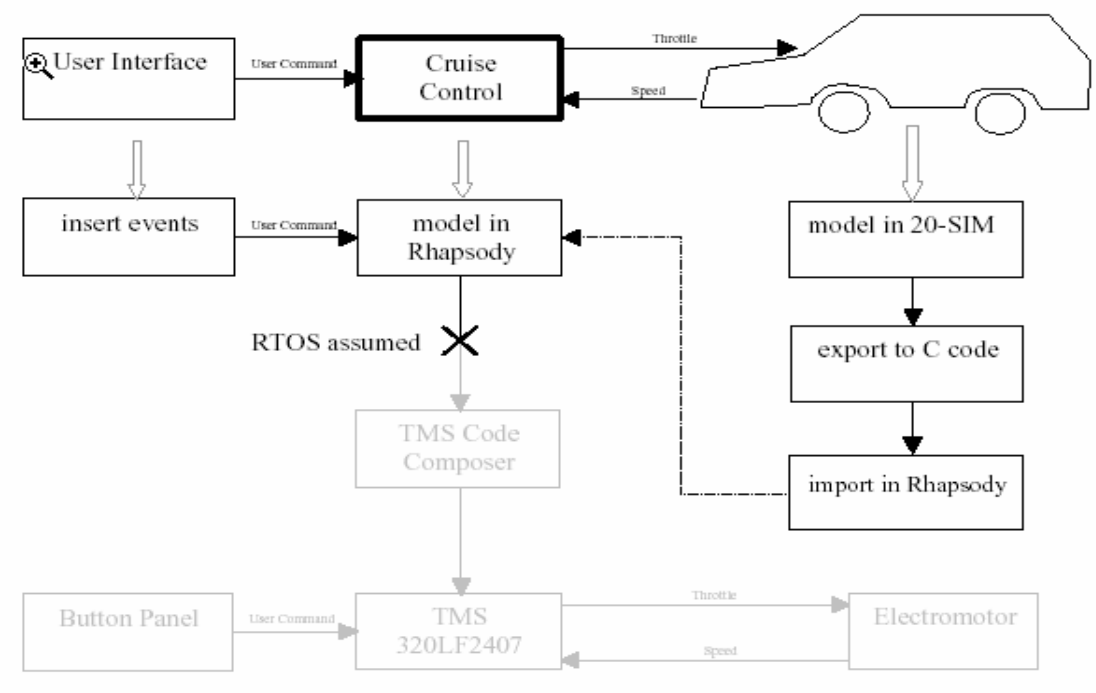


图 1.4 改变的项目策略

如果不写出一个新的框架来代替 Rhapsody 通常包含的 OXF 框架来进行设计，就不能考虑使用 TMS 板。

但这样做会大大超出任务的范围，因此决定使用一个汽车模型。这样，用 Rhapsody 开发出的导航控制（CC）仍然可以进行测试。

第 3 章包含了开发过程的分析。3.1 节对导航控制（CC）做了更详细的说明。3.2 节会回顾一下整个项目设计过程中所需的工具和技能。这些工具除了 Rhapsody 和必要的 C++外，还包括为汽车建模的 20-SIM。第 4 章涵盖了使用 Rhadsody 设计 CC 的全部内容。第 5 章详细说明如何使用 3.2 节中提到的工具来实现 CC，也描述了对 CC 的调整（5.2 节）。第 6 章说明的 CC 的测试和测试结果。第 7 章对项目做了总结并提出了建议。

2 ROPES

本项目采用的开发策略是面向对象快速过程（ROPES）方法。这种方法在本报告的章节安排部分已有说明，更详细的 ROPES 周期介绍如图 2.1 所示：

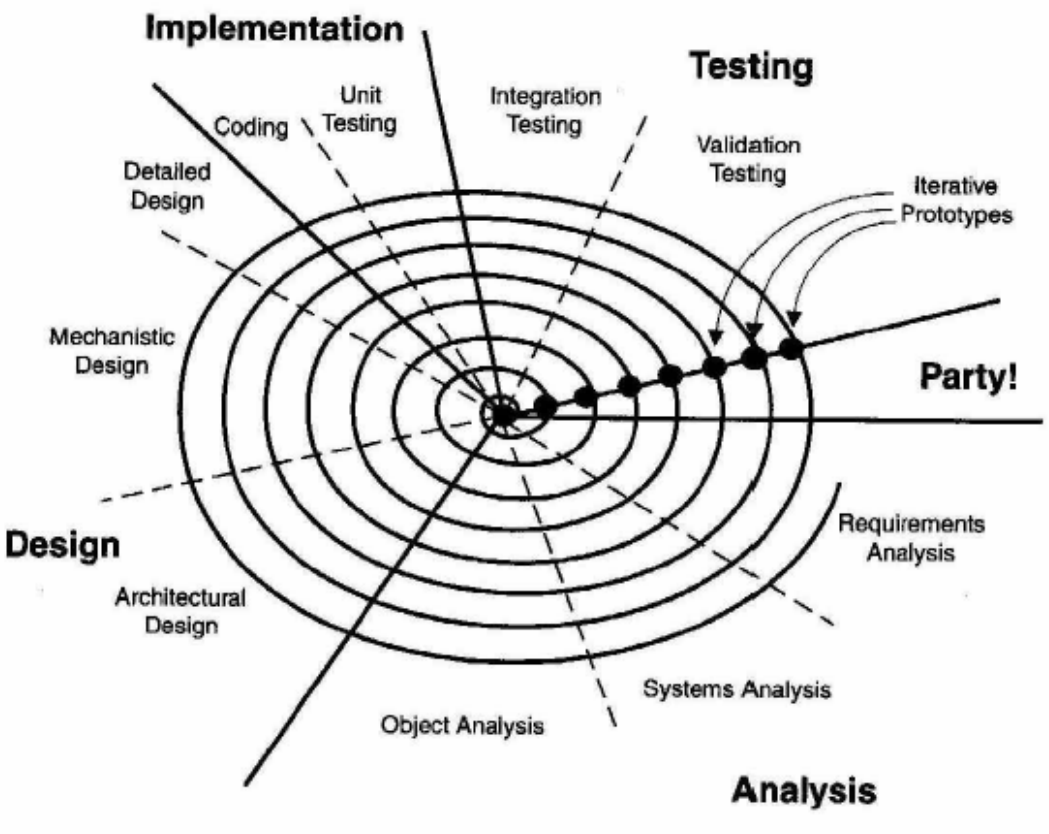


图 2.1 详细的 ROPES 周期（Douglas, 1999, 167 页）

任何开发过程的主要目的都是

（1）提高最终产品的质量

（2）改进开发工作的可重复性和可确定性

（3）在减少劳动的同时开发出合格的最终产品（Douglas, 1999）

如上所示，软件开发过程被划分成四个主要阶段：分析，设计，实现和测试。由于各子阶段都适用于本项目开发，所以本项目开发过程也作如此部署。

迭代的项目开发需要识别出项目中的子目标。每一个周期，新的元素被包括到设计中来，使设计越来越接近于期望设计。为了完成总体设计，识别出下列子目标：

- 定义用例，顺序图，和类结构
- 通过类结构实现数据传送
- 设置速度/恢复速度
- 控制速度
- 把 C++代码重写为 C 代码
- 在 TMS 板上实现嵌入式系统

后两个阶段最后由如下子目标替代：

- 在 200-SIM 中给小汽车建模
- 把模型导入 Rhapsody
- 向汽车发动机施加影响
- 调节控制器，使速度持恒

在第 3 章把各阶段的分析形成文档。第 4 章包含了设计部分。第 5 章描述实现而第 6 章描述测试。这样，报告的结构同样也显示了 ROPES 的结构。

3 分析

本章主要分析了需要使用 Rhapsody 实现的导航控制（CC）系统。首先要定义用例。为了获取 CC 的用例的详细描述，也为使用户全面理解 CC 的一般工作原理，1.1 节用词语描述了 CC 的功能。为了引起对可能表示对象的词的注意，我们给这样的词加下划线。为了能够识别出文本中可能的用例，我们对这部分文本加粗。

3.1 导航控制

为了使**机动车保持一定的速度**，首先需要**知道当前速度**。比如 CC 就通过速度传感器获取速度信息。把当前速度**同期待速度进行比较**就知道速度是否确实被维持在一定水平。为了**维持期待速度**，CC 要**能够改变**流入发动机的**油量**。这样 CC 的输出就是油量大小的信号。伺服系统通过这个信号就能够控制跟发动机相连的油门杆。

在行驶中可以通过按下标有“开/关”的按钮来设置期待速度。CC 单元会在此刻**获取并存储当前速度**。然后此速度会被用作期待速度的值，CC 会持续地把它**同当前速度进行比较**，尽力**维持那个速度**。

然后，要指定**使 CC 停止工作**的条件。司机可以通过再次按下“开/关”按钮简单地**关闭 CC**。更重要的是**在刹车时使 CC 停止工作**。这样，就防止了 CC 此时为了防止速度的消失而发动汽车，在这种情况下应该由司机去发动（所以是必需的）汽车。

在**使 CC 停止工作**后，用户可以通过再次按下“开/关”按钮在期望速度标准上**重新设定新的速度标准**。或者**恢复到早先设定的速度标准。**

最后，也要实现一些额外的功能，例如“CC 加速”或“CC 减速”。用户能够使用这些按钮把**设定的速度精确地调节**到期望标准。

对象和用例的简要说明可以用下面的列表表示：

Objects(对象)	Use Cases(用例)
User	Acquire actual speed
Button panel	Store actual speed
Engine	Compare speed
Speed Sensor	Keep speed constant
Store speed	Activate/Deactivate CC
Calculate Difference/Throttle	Resume speed
Controller	Fine-tune speed

表 3.1 导航控制（CC）中识别出的对象和用例

这些用例和对象几乎都可以在用例图和对象模型图中发现。对象模型图中那些不必要在软件中设计出来的对象已经被省略掉了，如按钮面板、速度传感器和发动机（汽车类会在以后为汽车发动机的行为建模）。

3.2 用例

在这个建模阶段，使用真实的传感信号产生逼真的输出很困难而且也没必要。因此表 3.1 中的对象可以简化为：

- GenerateSpeed（传感器的临时代替者）
- SignalConditionerIn（把从传感器读入的数据转换成可用（可控制）的形式）
- ProcessUserCommand(用户输入过程（由 Input 提供），并且分发给系统相应的任务）
- StoreSpeed（当用户设定速度是捕获当前速度样本）
- Controller（控制汽车的速度，努力使速度维持在设定点上）
- SignalConditionerOut （在伺服速度控制器或电动车作用范围内产生控制器输出）
- Display（临时对象，对系统中重要的信号进行分类并按顺序显示以进行检查）

按照上一章所做说明，进行设计的第一步要定义 CC 系统中的参与者和用例。第二步识别用户和用例的关系以及用例间的泛化。结果如图 3.1 所示：

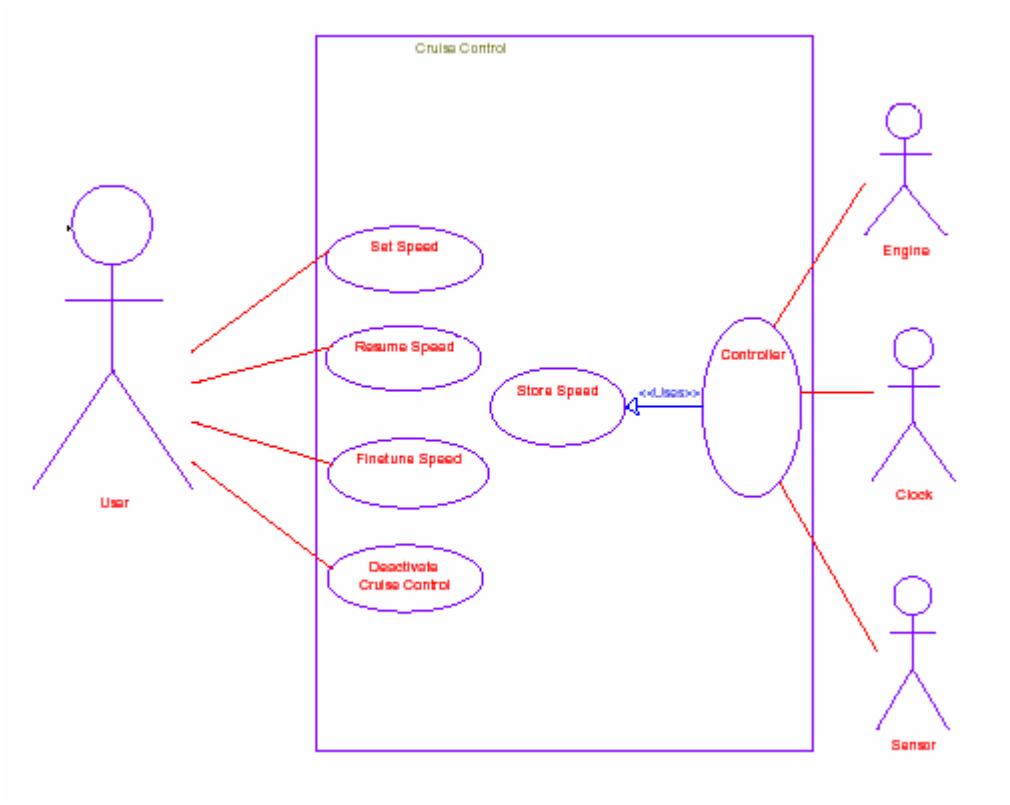


图 3.1 导航控制用例图

如上所示，此图显示了 CC 系统的不同功能以及用户能对其功能进行干预的位置。

此图仅仅是用来进行说明，到目前为止还没有生成代码，问题仅用示意图给概括出来。它不仅能作为对系统整体的参考，也能用作程序员对问题描述的参考（例如给项目经理）。只有当用例名和在顺序图或对象模型图中定义的类名相同的时候，Rhadsody 才建议合并他们。这样，在用例图中定义的关系才会影响代码，但这不是很重要。

顺序图帮我们大致勾画出了 CC 系统可能产生的动作的顺序。一个含有当用户按下每一个按钮时系统产生响应的顺序图生成了。需要再次说明，顺序图不会限制设计，它仅仅把可能发生的动作可视化了。顺序图如图 3.2 所示：

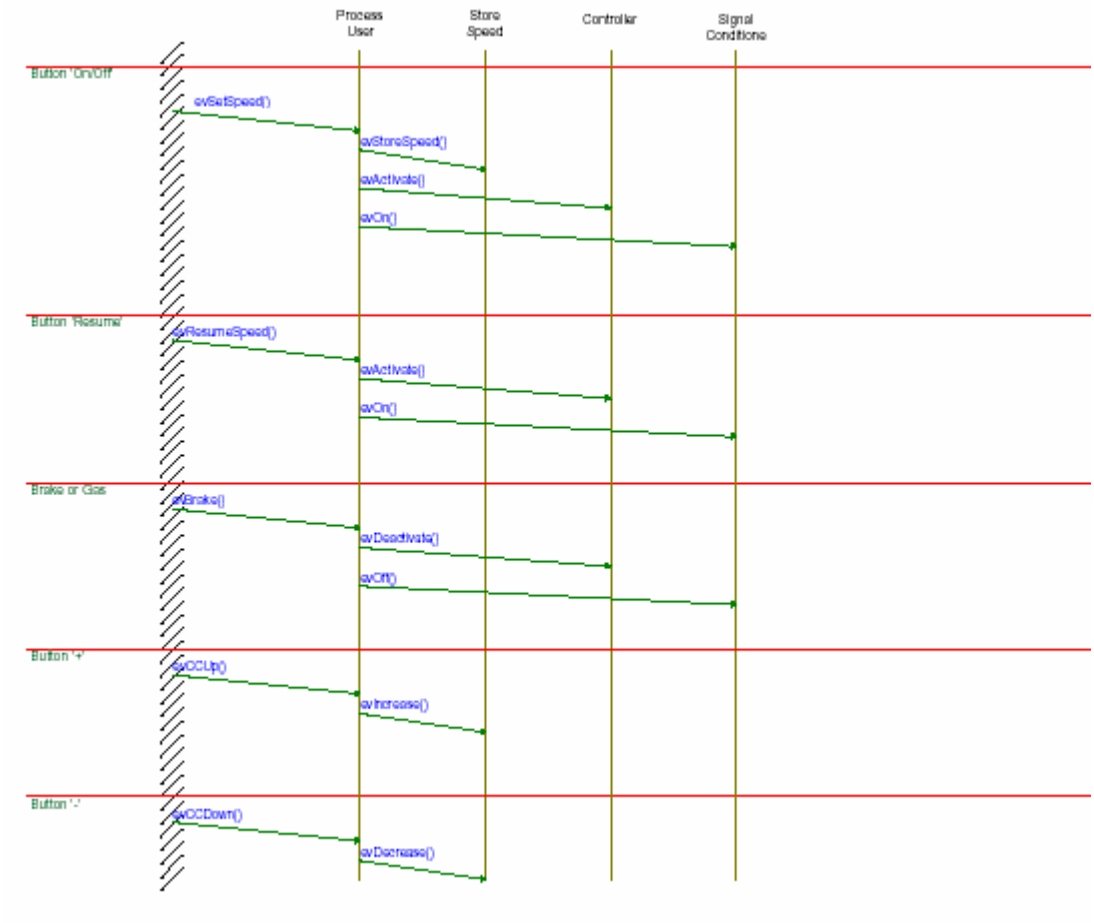


图 3.2 导航控制的顺序图

如上所示，ProcessUserCommand 类把用户命令发布到系统的其它部分。由于 CC 直接对速度施加影响，那么当用户亲自调节速度的时候，evOn 和 evOff 对于 SignalConditionerOut 来说就是必须的了。

3.3 对象模型图

在对象模型图中大体勾画出了 CC 软件的结构。通过对象模型图用户定义了类和类间的相互关系。CC 的类结构如下图所示。

无论什么时候，一个类为了执行操作，需要从其他类接受数据。在对象模型图中这个类必须通过关系或聚合和另一个类连接起来。因此，对象模型图也是定义数据传送的基础结构。

在对象模型图中，直线代表数据流，而曲线表示开关切换，事件产生和监视的信号线，即控制流。这并不是 UML 的特性，但是程序员可以完全自由地划线。采用这种方式是为了使图更清晰，这种方法类似 K.C.J.Wijbrans，1993。

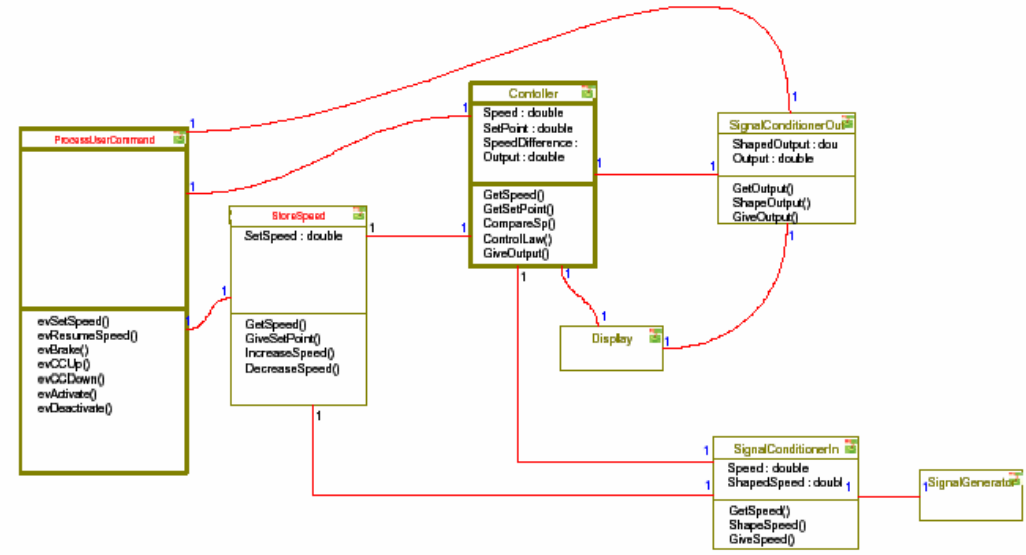


图 3.3 导航控制的对象模型图

在关系的每一端的字符 1 指出了关系的多重性，意思是一定数量的对象只能由一个类产生。

当类结构很清晰的时候，软件 CC 的设计与测试才是完整的。汽车模型也需要在 Rhapsody 中实现，这样就又产生了一个新的对象模型图。新的对象模型图如图 3.4 所示：为了能够关闭控制循环，类已经同 Input 和 Output 连接上了。用户需要在不借助 CC 的情况下一样能够控制汽车（如加速、刹车等），因此把 ProcesssUserCommand 也同 Car 连接。因为需要监视汽车可能存在的内部变量，Car 也同 Display 连接。

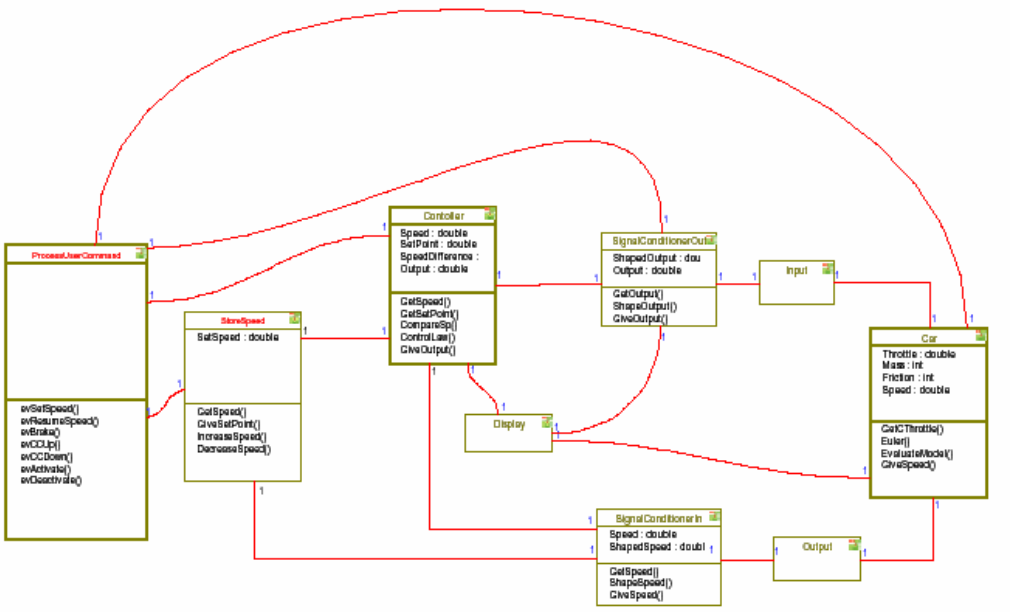


图 3.4 带有汽车模型的 CC 对象模型图

3.3.1 实现备注

当使用“聚合”进行类连接时，只要在“整体”或“聚集”类中“设定”了“部分”类，就完成了数据传送和事件发布。这种做法在指南（Ilogix, 1999, p4-17）中以“设定功能”的方式来定义的。当使用*关系*连接类时，Rhadsopy 会自动设定连接类。但是，Rhadsody 不能设定指向这些类的指针，这需要用户手工完成。对于每一个连接，程序员得写出象这样的代码：

```
p_Class1 -> setItsClass2(p_Class2);
```

代码要被加入到模型中的组件初始化列表，YourModel，Configuration，YourActiveConfiguration.。

记住，根据关系的类型（单向或双向），要对每一个关系设定一个或两个指针。

附录 A 列出了在使用 Rhapsody 过程中会遇到的一些最令人耗费时间和感到困扰的难点。对问题的解决可以参考此附录。

3.3.2 所需技能

使用 Rhapsody 实现 CC 需要用户精通编程。手册和参考指南可以予以帮助，但远远不够。范围广阔的 Windows 特性编程中的一些选项很难要求提供证明。也许对这些选项假定足够的编程背景已经变得不重要了。既然编程背景首先不是很重要，那么能够正确理解和操作 Rhapsod 要求就逐步变得重要起来。因为要定义实际状态功能，对象体，还要手工编程，所以也需要 C++的知识。也许前面提到的不重要选项需要深层次的 C++知识，但这里不需要。在 C++课上很少涉及面向对象编程，大多数的理解都是在设计中获得的。最初四周主要集中在了解 Rhapsody 和面向对象编程上。

用 Rhapsody 开发分三个阶段：

- 定义用例和指定顺序图
- 建立系统对象模型图
- 设计状态图

前两个阶段可以在 ROPES 方法分析部分看到，但最后阶段则属于设计部分。

第一阶段，用例和顺序图，主要让程序员描画出问题的轮廓，限制和精确他正在处理的设计问题。第二阶段，对象模型图，创建类的框架以及类之间的相互依赖，勾画出程序和数据流的大致轮廓。第三阶段，状态图，完成

程序。不同系统必须定义的状态以及和这些状态相关的动作有所不同。程序员可以选择制作大的状态图，也可以定义几个状态图，例如为每一个类的抽象部分定义一个。关于 ROPES 迭代开方法，后一种只是建议。这样，就能很容易逐步精化到实现。

使用电动汽车在硬件中为汽车建模。Rhapsody 设计的 CC 会被嵌入到 TMS 板上。用户接口采用一系列按钮来实现。因为 TMS 板的代码编发器使用 C，所以在 TMS 板上嵌入软件需要 C 的知识。C++转换到 C 可以使用附录 A 的 J.M.van Drunen,2000 完成。

当项目变得清晰的时候，开始让没有 Code Composer 的 20-SIM 工作了。现在已经使用键合图为汽车建模并用 Rhadsody 实现。这样就实现了一个完整的“用户-CC-汽车”系统。

4 设计

既然已经分析了 CC 的功能和需求，设计在软件中就变得重要了。4.1 节介绍定义 CC 期望行为的状态图。4.2 节介绍用 20-SIM 开发汽车模型。

4.1 状态图

使用 UML 编程过程的下一步就是定义状态图。通过定义 CC 会发生的状态、状态转移，以及伴随的动作就能确定 CC 的正确行为。为了防止状态图变得复杂，决定为每一个类分配一个状态图。而且如第三章所述尽可能逐步精化。下列图形显示了创建的状态图。

首先我们会观察一下简单类（SignalConditionerIn 和 GenerateSpeed）的状态图。这些类以单一状态循环，重复执行在状态体中定义的功能。这些类的构成如图 4.1 所示：

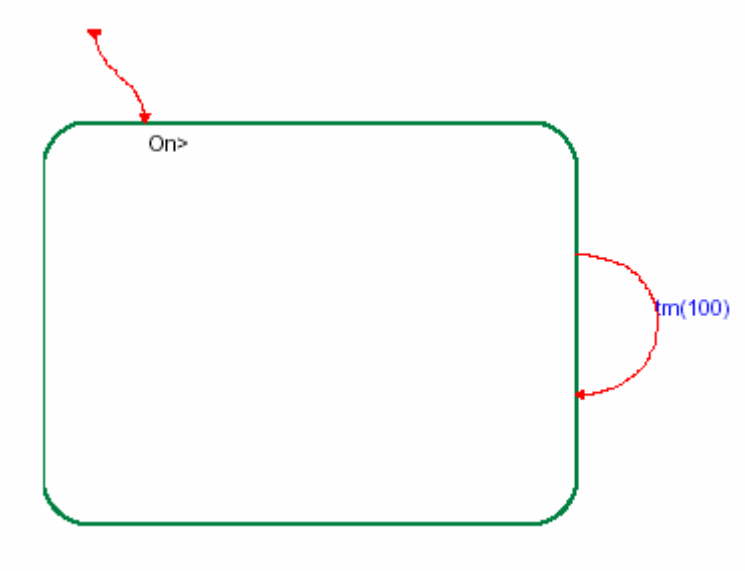


图 4.1 SignalConditionerIn 和 GenerateSpeed 的状态图

整个 CC 设计中的循环时间维持在 100ms（tm(100)）。由于一个信号的输入到输出会经过 5 个类（见对象模型图），这样整个 CC 的循环时间就达到 500ms。下一节会论述此循环时间满足设计需求。

下面的列表显示这些类循环的每一个功能：

- SignalConditinerIn　　GetSpeed(), ShapeSpeed()
- GenerateSpeed　　GenerateSpeed()

由于 SignalConditinerIn 必须有关闭的能力，所以它的状态图稍微有些不同。实现的方式如图 4.2

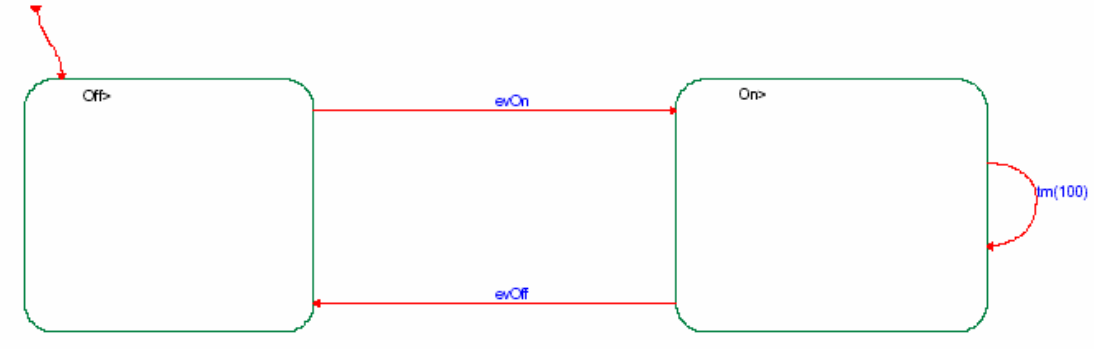


图 4.2 SignalConditinerIn 的状态图

SignalConditinerIn 执行 GetOutput()和 ShapeOutput()操作。

ProcessUserCommand 显示通过插入 5 个事件就可以控制整个系统。

- evSetSpeed（激活 CC 使其维持一个新速度（“开/关”按钮）或使其停止）
- evRestoreSpeed（激活 CC 使其维持先前速度）
- evCCUp（以 1km/h 加速）
- evCCDown（以 1km/h 减速）
- evBrake（当使用刹车时，CC 停止工作）

这些事件如何包含进 ProcessUserCommand 状态图，如图 4.3 所示。

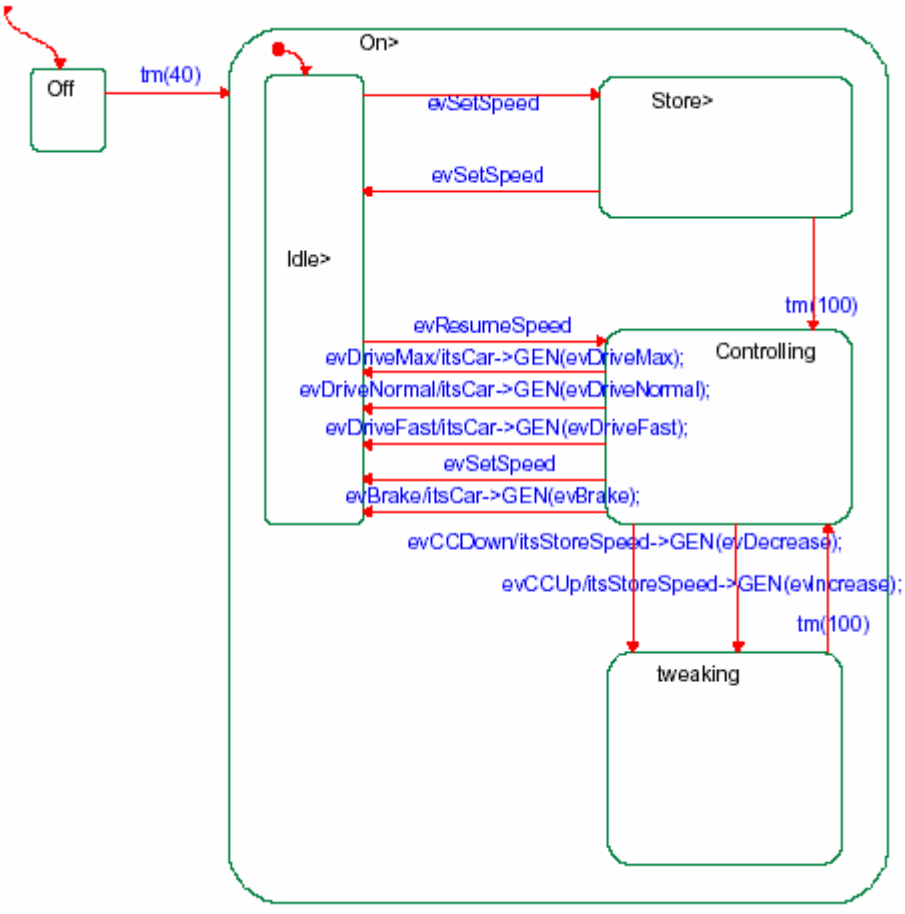


图 4.3 ProcessUserCommand 状态图

ProcessUserCommand 类在整个系统中处于协调位置。其他类就是从这里被唤起。

进入 Idle 状态时停止 Controller 和 SignalConditionerOut，离开 Idle 状态时激活。进入 Store 状态时在 StoreSpeed 类中产生一个事件（evStoreSpeed()）。如上所见，事件 evSetSpeed()导致进入 Store 状态也会相应进入 idle 状态（当 evSetSpeed()在 100ms 内重新发生时。由于在那以后进入了 Controlling 状态）。这样做是为了完整性，因为在 100ms 内用户偶然按下或调整开/关按钮时也会发生。Controlling 状态不含有代码，因为离开 Idle 状态就已经启动了控制器。

如上所示（当处于 Controlling 状态时）事件 evCCUp()和 evCCDown()会在类 StoreSpeed 中产生事件 evIncrease()和 evDecrease()。按下“开/关”（evSetSpeed()），踩下刹车或用其他方法调节速度，都会导致进入 Idle 状态。已经指明仅仅当碰到刹车或者甚至离合，CC 也会作出反应进行加速。只要司机想要用任何方式调节速度，CC 就停止工作。所以按下每一个这些调节器都会产生同一个事件：evBrake()。

StoreSpeed 类定义如下：产生于 ProcessUserCommand 的事件 evStoreSpeed()，从 SignalConditionerIn 获得速度的一个样本并用下列函数存储到变量 SetSpeed 中（在 On 状态下操作 GetSpeed()被唤起）：

```
SetSpeed = itsSignalConditionerIn -> GiveSpeed();
```

SignalConditionerIn 类通过当前速度回应此调用：

```
Return Speed;
```

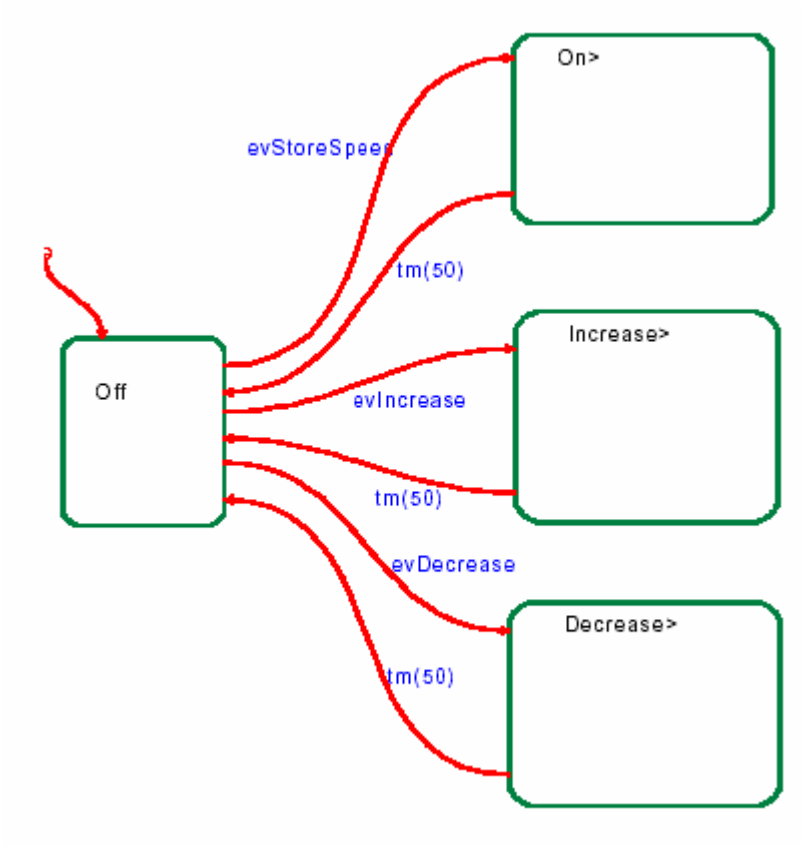


图 4.4 StoreSpeed 状态图

evIncrease 和 evDecrease 的反应分别是 SetSpeed++和 SetSpeed--。这样司机就能够以 1km/h 来调节速度设定。

最后，显示 CC 的主要功能：控制器（the Controller）。控制器每 100ms 刷新对速度控制需要的值：Speed 和 SetSpeed（在控制器中被称为 SetPoint）。用这些值，控制器能持续地在循环周期内比较和控制（因为不知道要控制什么发动机，所以目前没有真实的控制措施是必然的）。在控制器中使用控制规则是一种适当的措施。在 Rhapsody 中实现汽车模型时，控制规则会调节控制汽车的速度。图 4.5 显示了控制器的状态图。可以看到，Controller 类包含 3 个并行进程，每一个的循环时间为 100ms。

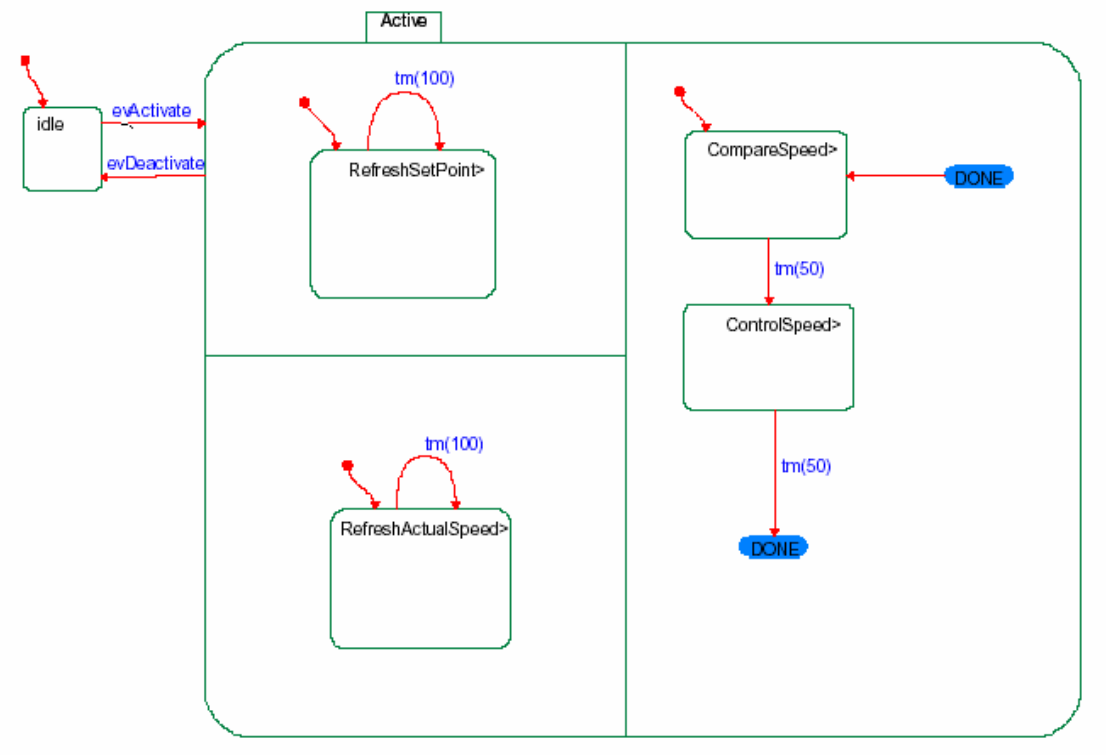


图 4.5 Controller 状态图

这样，一个简单的导航控制就实现了。下一节会讲解系统的动画（一个 Rhapsody 特性）。

4.1.1 仿真和动画

当仿真一个模型的时候，Rhapsody 能够使用动画以图形的方式深入观察系统的操作。用户可以选择任何一个状态图和顺序图来生成动画。在动画过程中可以显示系统事件的顺序。这样，用户就能迅速看出系统功能是否存在和是怎样的，并且检验是否执行了相应的指令。

仿真和动画一个模型可以按指南的描述去做。但是由于指南没有为指令提供综合的解说，因此就一些模糊的问题，在此加以解说：

- 在文件夹 Compponent/Configuration 里，“初始化标签”（‘initialization-tag’）显示了带有复选框的所有构建的类。只有选中的类才会被包含进动画中去。无论何时，如果动画中的一个类具有某种功能，选中这个类的复选框。通常用户可以简单地选中所有复选框，以避免动画失败。

- 只有当仿真运行起来，才能生成动画

- 当一个类的初始状态唤醒了其他类（例如生成一个事件）的时候，要确保在进入初始状态以前有几微秒的延迟。这样，就能够避免由不完整的数据结构引起的不确定错误。

4.1.2 实现备注

直接向其他类（Controller 和 SignalConditionerOut）插入事件前有 40ms 的 Idle，因为处于启动状态，在进入 On 状态之前 ProcessUserCommand 就已经完成了。在这些类被请求之前应该先被创建，如果没有延迟，一个由于不完整数据结构引起的内存例外错误就会发生。

应该实现一个存在检查，控制着程序直到被请求的类产生（仅在有限的时间内，因为是由于类不存在而非启动问题引起的）。

4.2 20-SIM

既然控制一个真实的电动汽车的速度是不重要的，那么 CC 的设计将会在软件中测试。已经使用 20-SIM 为汽车和发动机创建了模型并为之生成了 C++代码，这样就能够在 Rhapsody 中实现它。模型和可应用的简化版会在下一节论述。

首先，已经对汽车和发动机系统的相关部分完整的描述进行了建模。相关部分是发动机的动态属性和汽车的重量及总的摩擦力。但是在这个设计阶段，使用内燃机还是电动机并不重要。目前，只是应用可调节的动力源。结果如图 4.6 的模型键合图所示。

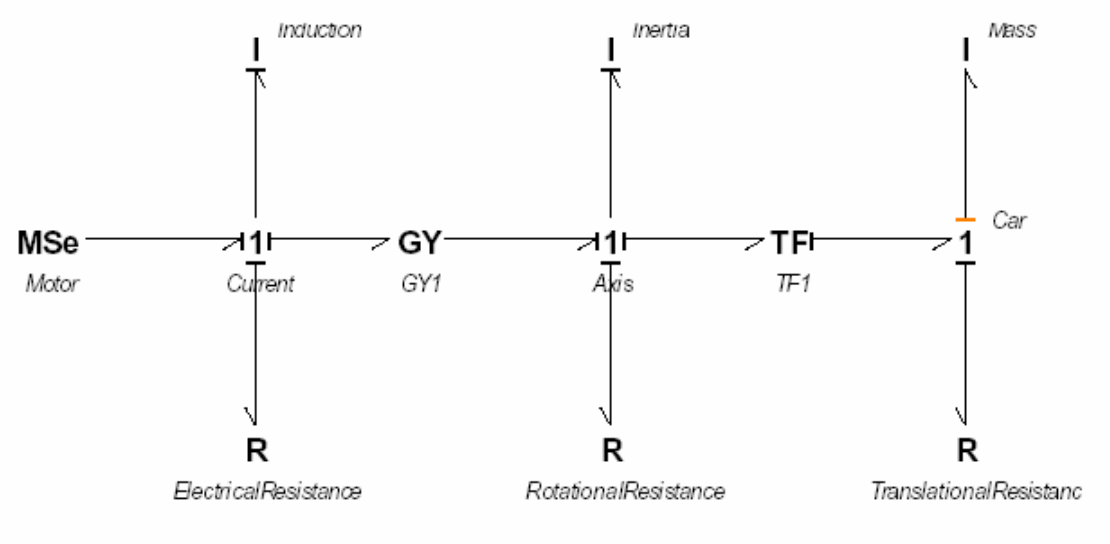


图 4.6 汽车和发动机模型键合图

电动机的实际参数在 1994 年已经被 Breedveld 和 Van Amerongen 发现。当注意到发动机的传递功能时（方程 4.1）就能发现发动机的时间常数。

$$H(s)=\frac{\omega(s)}{u(s)}=\frac{K_T}{K_T^2+R_eR_m+s(R_eJ_m+R_mL)+s^2LJ_m}\approx\frac{1}{K_T}\frac{1}{1+sR_eJ_mK_T^{-2}}$$

（方程 4.1）

应用 Breedveld 和 Van Amerongen，1994，143 页，表 19.1 发现的参数，发动机时间常数的结果大约在 10^{-4} 秒。对于一辆大约 1000 公斤的小汽车，可以通过下列假想的实验得出一个时间常数的近似值。

想象一辆 1000 公斤的汽车，产生 100 公里/小时的速度。试预测汽车需要达到 1/e（≈0.37）倍那速度（100 公里/小时）的时间会是 30 秒和 60 秒之间。现在想象同一辆汽车从静止到 100 公里/小时并且达到 1-1/e（≈0.63）倍那速度（100 公里/小时）时，会在 5 秒和 10 秒之间发生。假设最坏的情况，使用最小时间常数 5s。

第二个实验并不完全有效，因为大多数汽车不会仅达到 100 公里/小时。甚至会超过此速度。但是，只是表明发动机时间常数远远超过汽车的时间常数。因此，发动机的时间常数同汽车时间常数相比并不太重要。根据这些考虑，简化的模型如图 4.7 键合图所示：

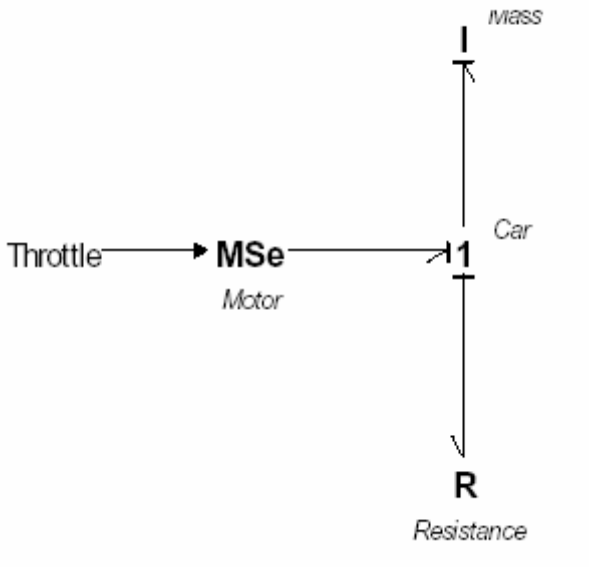


图 4.7 一个简单的汽车和发动机键合图

当应用时间常数 5s 时，汽车的阻力如下：

$$\tau = \frac{1}{RC} = \frac{I}{R} \Leftrightarrow R = \frac{I}{\tau} = \frac{1000}{5} = 200\ {}^{Ns}/_{m}$$

（方程 4.2）

这个模型很合适表示一辆发动机驱动的汽车的方程。下一章会详细解说这个模型在 Rhapsody 的实现。

5 实现

目前为止本项目中一直还没有确实嵌入软件，实现已被简化为在 Rhapsody 中插入 20-SIM 模型。这种在 Rhapsody 中的实现很明显是有问题的，因为一些被包含进来的 C++库文件含有“名称冲突”，例如会发生在 stdio.h。这就导致需要手工实现。5.1 节描述如何完成这个工作。5.2 节详述控制器的调节。

5.1 手工实现汽车模型

分析 20-SIM 产生的代码和方程式，下面一系列方程模型被合并。

<initializtion> //set all the variables zero the first time（首先把所有变量设为 0）

impulse = impulse + h*force （方程 5.1）

speed = impulse/mass （方程 5.2）

rforce = resistance*speed （方程 5.3）

force = eforce - rforce （方程 5.4）

方程 5.1 中对由发动机产生的力 eforce 和由汽车摩擦产生的力 rforce，应用了显式的欧拉积分方法。这些方程就像系统的其它部分，会在汽车模型中连续地循环。循环时间会是 100 微秒。由于汽车的时间常数选择了 5 秒，关于一个样本时间至少要是模型时间常数（Van Amerongen 和 De Vries,1999）的 10 倍的设计需求是可以满足的。当样本时间被设定为 0.5 秒时，正好是控制器循环时间的 10 倍。（参考第 4 章）

现在，为了实现一个逼真汽车模型，已经找到了汽车参数的实值。为了找到汽车模型的实参，应用了 20-SIM 中的模型。首先，确定了汽车的时间常数。当汽车加速到 100 公里/小时的时候，时间常数是在速度为 1/e 倍 100 公里/小时（63 公里/小时）的时间。

下图显示了具有 200Ns/m 阻力和 5600N 力的汽车加速到 100 公里/小时的情况。

图中看出，时间常数的近似值为 5 秒，同在第 4 章的计算相当。

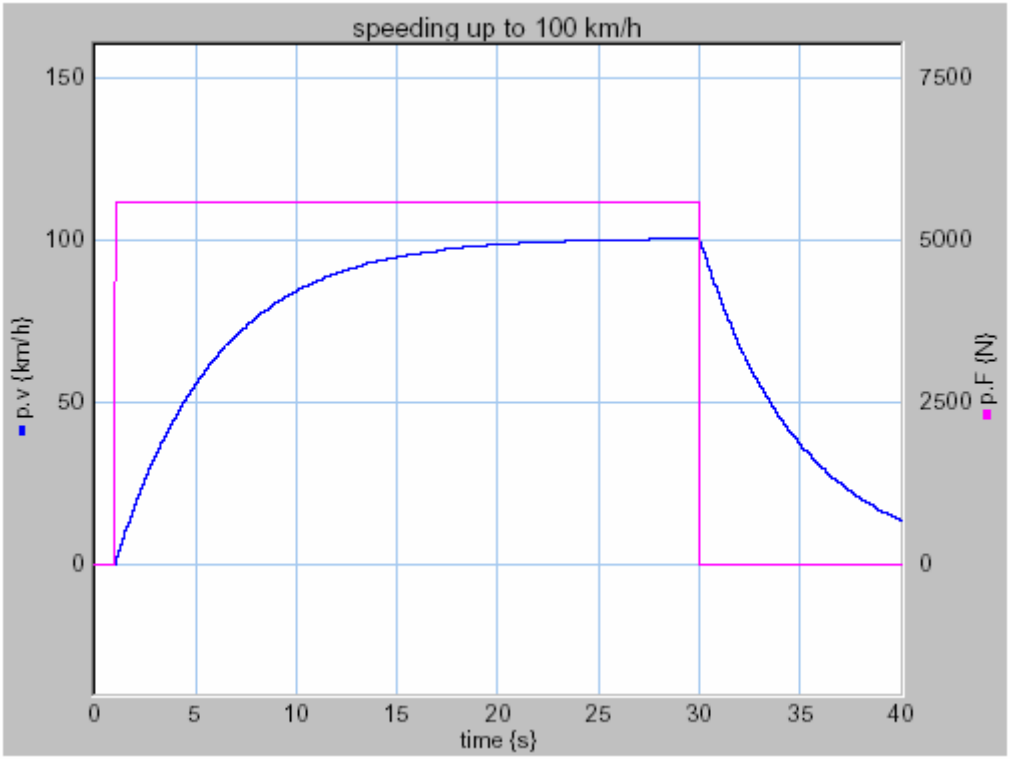


图 5.1　确定汽车的时间常数

假定汽车的阻力为 200Ns/m，最高速度约 180 公里/小时时，就能发现汽车能够释放的最大力。20-SIM 中的模型显示速度达到 180 公里/小时的力为 10,000N。这就是假定汽车发动机能够释放的最大力。

如果把节流阀对汽车的供给从 0 到 100 进行索引，100 同最大力一致，按 1 增加的节流相应的意味着力按

100 = 10000N -> 1 = 10000 / 100N = 100 N　　（公式 5.6）

增加。

最后，外力（风，斜坡）也要插入到模型中。已经选择以每步 1000 改变 eforce。这样，每一步就是可见的速度。用这些值，基于第一个模型，汽车时间常数的估算就能够从方程 5.8 得出。

5.2 调节控制器

既然已经导出一个汽车模型并被包含进了 Rhapsody，CC 需要控制汽车模型的速度了。因为在控制器中应用控制规则是适合的。控制规则的 k 因子应该被优化。为了方便，采用经验来调节此因子。在 3 次尝试后，发现 K 因子最大为 35 或 36。因为 38 会导致不稳定行为，我们希望具有稳定性的余地，所以设定为 36。

6 测试和结果

到目前为止，已经开发出了一个完备的功能系统，需要进行测试。但是动画期间当系统即时反应时模型的输出很难很好的表现出来。测试结果的表现是很大的问题。Rhapsody 对于变量值只有数值提示，还没有开发出合适的结果图形显示的绘图功能。所以测试和结果报告只能采用文字的形式，没有图表支持。6.1 节会详细描述为了测试系统行为而向系统发起的事件顺序。6.2 节会报告系统的反应。 相关的 Rhapsody 输出的半屏输出（手写）会包含进来作为图解。

注意，为了显示公里/小时，内部变量 Speed 已经在 Display 类中被乘以 3.6。

6.1 测试

CC 已经拥有了所有重要的功能，编写了程序，进行了测试。首先检查汽车模型。它象真车一样可以加速和减速，如果汽车没有速度，就会变负，结果和刹车负力一样。第二个测试是，当在正常速度驾驶时，设定速度并观察如何维持设定速度。现在已经设定好了速度，控制器在控制速度，任何由用户所作的调节都会导致速度无效，当提高速度加快行驶和刹车时，但没有失去设定的速度（已测试）。下一步，通过再次先正常行驶然后按下“恢复速度”，用户能够恢复以前设定的速度（已测试）。

模型也能够调节外部力，如表示天气状况或路面的斜坡。这主要通过提高阻力到 1000N 和降低阻力到-1000N 来完成的。

最后，设定速度能以每 1 公里/小时进行调节。测试是通过提高设定速度到+3 公里/小时和降低设定速度到-3 公里/小时来进行的。下面的列表显示了向系统的发起的事件顺序：

- evDriveNormal()
- evBrake()
- evDriveNormal()

- evSetSpeed()
- evDriveFast()
- evDriveNormal()
- evSetSpeed()
- evBrake()
- evDriveNormal()
- evResumeSpeed()
- evResistanceUp()（3倍，因为事件导致+1000N）
- evResistanceDown()（6倍，从+3000前一个事件开始达到-3000N的外力）
- evCCUp()（3倍达到+3公里/小时）
- evCCDown （6倍从3倍CCUp()达到-3公里/小时）

6.2 结果

前述的系统测试顺序列表会被扩展。如前所述，有时会用半屏输出显示系统的反应。结果中，Speed是汽车的当前速度，SetPt是CC设定的速度，Resist是影响汽车的外部阻力，Cthrot是CC为了维持设定速度修正节流阀而计算的值。

- evDriveNormal()　　汽车达到速度约90公里/小时

Speed	SetPt	Resist	Cthrot
90	0	0	0

- evBrake()　　速度在约两秒内减至0
- evDriveNormal()　　汽车再次达到90公里/小时
- evSetSpeed()　　CC设定汽车当前速度，为了维持当前速度激活校正节流阀（注意由于数值数字的减少，校正的节流阀值是非零的）

Speed	SetPt	Resist	Cthrot
90	90	0	-6.57e-7

- evDriverFast() 汽车速度达到约135公里/小时并且校正节流阀值设为0

Speed	SetPt	Resist	Cthrot
134.998	90	0	0

- evDriveNormal() 汽车再次达到90公里/小时
- evSetSpeed() 速度再次被设为当前速度
- evBrake() 速度减至0校正节流阀值设为0
- evDriveNormal() 汽车再次达到90公里/小时
- evResumeSpeed() 早先设定的速度再次被维持
- evResistanceUp() 阻力设为3000N（注意增加校正节流阀的值并且速度不是正确达到设定的速度，这是一个合适的控制器的属性）

Speed	SetPt	Resist	Cthrot
87.081	90	3000	28.3784

- evResistanceDown() 阻力设为-3000N(校正节流阀值现在也为负，速度比设定速度稍微高)

Speed	SetPt	Resist	Cthrot
87.081	90	3000	28.3784

从校正节流阀值精确的大小可以看出模型的线性

- evCCUp() 设定速度以3公里/小时提高（阻力已经被维持了，显示了设定速度提高了，校正节流阀的值也跟着改变）

Speed	SetPt	Resist	Cthrot
94.627	92.999	-3000	-26.802

因为是增加量，设置点不是精确的3公里/小时而是0.2777m/s，要不然会导致舍入错误。

- EvCCDown() 设定速度以6公里/小时减少（低于初始设定速度3公里/小时）

Speed	SetPt	Resist	Cthrot
90.082	87.001	-3000	-29.955

从测试结果可见，CC的功能与真正的CC很相似。既然真正的汽车不完全会是线性的，可能会对一个线性的汽车模型是否足够真实有疑问。但是，这些考虑已经超出了本项目的范围。毕竟不是要去构建一个非常精确的模

型，而仅仅是为了论证用Rhapsody建模的可能性。

参考文献

1. Bruce Powell Douglas, *Doing Hard Time*, Chapter 4: *Rapid Object-Oriented Process for Embedded Systems*, 1999.

2. J.M. van Drunen, *Realization of link drivers implementing CSP-channels on 20-controller*, M. Sc. Thesis, Control Laboratory, Electrical Engineering Department, University of Twente, The Netherlands, April 2000.

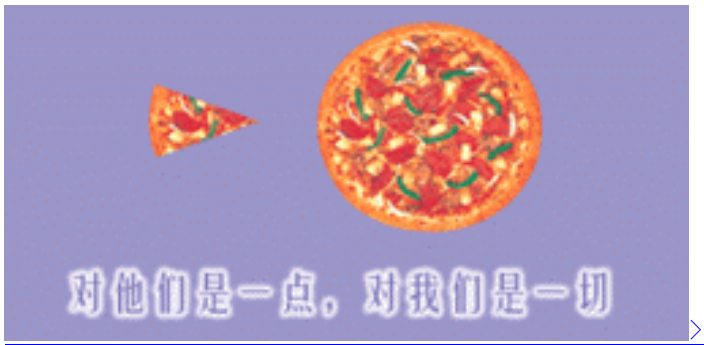
3. Breedveld and Van Amerongen, *Dynamische systemen: modelvorming en simulatie met bondgrafen, cursusdeel 3: Modelleren van systemen in andere domeinen en toepassingen*, pp. 136 - 144, February 1994.

4. R. van Damme, D. Dijkstra, G. Still, C. Traas and G. Zwier, *Numerieke Wiskunde*, pp. 103 -118, November 1997

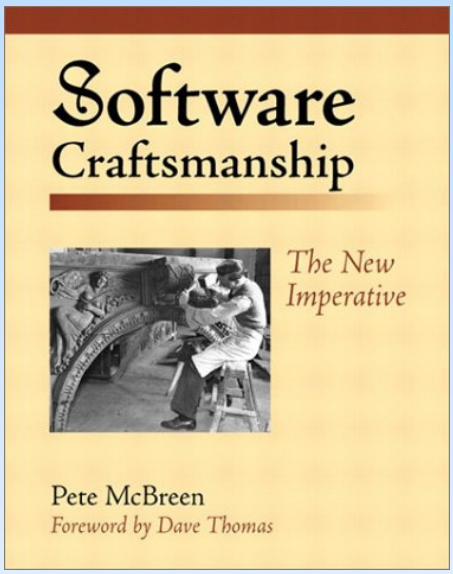
5. K.C.J. Wijbrans, *Twente Hierarchical Embedded Systems Implementation by Simulation*, April 1993.

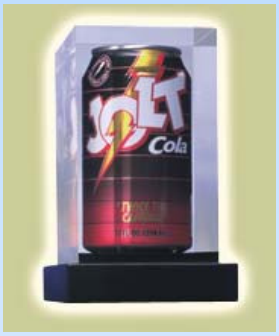
6. Van Amerongen and De Vries, *Meet- en Regeltechniek deel 2*, Leereenheid 6, May 1999

7. Ilogix, *Rhapsody Tutorial*, p 4-17, March 1999



《软件工艺》





2002 Jolt Awards 获奖书籍

Pete McBreen

翻译：UMLChina 翻译组

工艺不止意味着精湛的作品，同时也是一个高度自治的系统－－师傅（Master）负责培养他们自己的接班人；每个人的地位纯粹取决于他们作品的好坏。学徒（Apprentice）、技工（Journeyman）和工匠（Craftsman）作为一个团队在一起工作，互相学习。顾客根据他们的声誉来进行选择，他们也只接受那些有助于提升他们声誉的工作。

中文译本即将发行！

处理对象的特性

Martin Fowler 著，Xu Zhiling 译

查看评论

关键词：特性(property)，动态特性（dynamic property）

几乎每个创建的对象都需要特性：有关对象的一些声明，例如，人的身高，公司的 CEO，航班的航班号。有许多种方法可以模拟特性，在本文中，我将探索其中的一些方法，以及可能在什么时候使用它们。常见到一些模式（pattern）涉及到这个主题，但是它们通常仅覆盖部分图景（picture）。在此，我想广泛地研究这个问题，给出对这些选择的更好讨论。

最普通及最简单的情况是使用*固定特性（Fixed Properties）*，就是仅仅声明类的属性（attribute）。对于大多数类而言，这就足够了。当你有大量固定特性，或可能在运行时需要经常改变它们的时候，固定特性就开始失灵。这些因素将你引向各种*动态特性（Dynamic Property）*。所有的动态特性都具有参数化属性的性质：要查询一个特性，需要使用带有一个参数的查询方法。动态特性中最简单的是*柔性动态特性（Flexible Dynamic Property）*，其参数只是一个字符串。这样，定义和使用特性变得容易，但是难以对它进行控制。如果你需要这种控制，可以用*确定性动态特性（Defined Dynamic Property）*，所带的参数必须是某些类的实例。更深入的控制允许你使用*类型化的动态特性（Typed Dynamic Property）*，它强调动态特性的类型。

当特性结构变得复杂时，应当考虑使它们成为*分立特性（Separate Properties）*，它使特性在其自身权利范围内成为对象。如果需要多值特性（multi-valued properties），你可以考虑*类型化关系（Typed Relationship）*。本文中最复杂的例子是，你想用规则控制什么类别的对象以及它具有什么类型的特性——这需要*动态特性知识层次（Dynamic Property Knowledge Level）*。

如果你想给对象定义一个特性，但不改变支持它的接口所使用的模式，其类别的特性完全属于*外在特性（Extrinsic Property）*。

问题	解答	名称	页码
	给特性一个针对细节的具体属性。这将转换成编程语言的查询方法，或是更新方法。	固定特性（Fixed Property）	2
	提供一个参数化的属性，该属性代表依赖于参数的不同特性。	动态特性（Dynamic Property）	4
	提供用字符串参数化的属性。仅仅是用字符串声明特性。	柔性动态特性（Flexible Dynamic Property）	5
	提供一个用其它类型实例参数化的属性。通过创建这种类型的新实例声明一个特性。	确定性动态特性（Defined Dynamic Property）	7
	提供用其它类型实例参数化的属性。通过创建那种类型的新实例声明一个特性，并指定特性的值类型。	类型化动态特性（Typed Dynamic Property）	9
如何表达一个对象的细节，并记录其相关细节？	为每个特性建立一个分离的对象，这样特性细节就变为对象的特性。	分立特性（Separate Property）	9
如何表现两个对象间的关系？（如何表现多值的动态特性？）	为两个对象间的每个连接创建一个关系对象，给关系对象一个类型对象以说明关系的含义。（类型对象是多值特性的名称）。	类型化关系（Typed Relationship）	14
使用动态特性时如何强制特定类别的对象有特定的特性。	创建包含什么类型的对象，使用哪种特性规则的知识*层次*。	动态特性知识层次（Dynamic Property Knowledge level）	16

如何给定对象一个特性而不改变其接口。	产生另一个对象负责了解该特性	外在特性（Extrinsic Property）	18

什么是特性？

这不是一个无聊的问题。当人们争论“特性”这个术语时，他们可能表示很多不同的事物。对于某些事物，特性是一个实例变量，或者类的数据成员。对于另一些事物，特性是 UML 图中小方框内的一类东西。所以在开始这篇文章前，不得不先定义我对这个词的习惯用法。

对我来说，特性是有关对象的一些信息，可以通过查询方法获得。它可以是一个数值类型（例如 Java 中的 int），或是类的实例。特性可以更新，但并非必须如此。可以在创建对象时设定特性，但是这也不是必须的。特性可以存储为实例变量或数据成员，但不是必须这样。类可以从另一个类直接取值，或是通过进一步的计算取值。因此，我对特性持有接口的观点，而不是实现的观点。这是我在设计中的一个通常习惯：对于我来说，面向对象的本质是把接口和实现分离，并使接口更加重要。

固定特性(Fixed Properties)

固定特性是我们使用最普通的特性类型。*固定特性*在类型接口中声明。声明给出了特性的名称和返回类型。图 1 显示了用 UML 建立的特性模型，表 1 显示了如何用 Java 实现这些特性的查询方法。我选择的例子表明，这个讨论可以同样用于 UML 属性和 UML 关联。它也用于计算出的值（年龄）以及那些合理存储的数据（生日）。

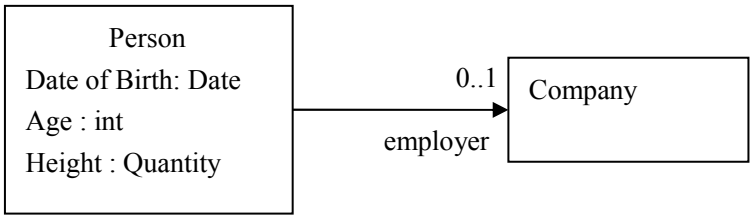


图 1 用固定特性建立人的模型

```
class Person {
public Date getDateOfBirth();
public int getAge();
public Quantity getHeight();
public Company getEmployer();
public void setDateOfBirth (Date newDateOfBirth);
public void setEmployer (Company newEmployer);
}
```

表 1：图 1 对应的 Java 操作

查询操作通常遵循某些命名约定。smalltalk 总是在特性的名字(dateOfBirth)后命名查询。C++没有固定的约定，有些人在特性后面命名，其他人使用“get”约定（getDateOfBirth）。最初 Java 没有特殊的约定，但是现在大多数人采用 get 约定。我个人觉得在读代码的时候“get”让人不舒服，所以我更倾向于不使用，但是 Java 的风格是使用 get，所以在此使用。应当保证无论是使用保存的还是推导的值，都遵循同样的约定。Person 类的客户程序应当不知道或者不关心年龄是保存的或是推导出来的。

修改操作存在与否取决于你是否希望值能够直接修改。若是存在，你要按照某些命名方案提供一个修改操作，例如 setDateOfBirth(Date)。因为不同的返回值存在不同的约定。你可以返回新的特性（Date）值，被修改的对象（Person），或是什么都不返回（void）。我更喜欢修改操作（modifier）返回 void，以帮助我清楚地区分修改操作和查询。

固定特性

如何表现对象的细节？

给对象一个针对这个细节的特殊特性。这种方法将转化为编程语言中的查询方法或是更新方法。

（优点）清楚和明晰的接口

（缺点）只能在设计时添加特性

你可能希望通过为构造函数提供参数来给特性赋值。通常，你想在构造函数中设置足够的特性，以构造一个构成良好的类。不想直接修改的特性应当没有针对它的修改操作。如果你只想通过由生日来计算年龄，对于年龄特性就是这种情况。对于不可修改的特性也是如此：类在其生存期内不变化。但是，在使一个特性不可变时，要考虑人为误差。在现实世界里，生日具有不可改变的特性，在向计算机系统输入的时候有可能会输错，因而它成为可变的。软件通常模拟的是我们所知道的世界，而不是世界本身。

到目前为止，固定特性是你遇到的最普通的形式。其原因是：简单且易于使用。你应当将固定特性作为你表达特性首要的和最通常的选择。在本文中，我将给出固定特性的其它可选方法。这些方法对于特定的情况更合适，但是大多数时间并非如此。在我们研究其它方法的时候，要记住这一点。我在 99%的时间使用固定特性。其它的变化过于复杂，这就是为什么本文的大部分花在这些方法上——也是为什么我不喜欢使用它们的原因！

动态特性(Dynamic Properties)

固定特性的关键是在设计时将它们固定下来，在运行时所有的实例必须遵循这个决定。对于某些问题，这是很笨拙的限制。想象一下我们开发一个复杂的联络系统，有些东西是固定的：家庭地址、家庭和工作电话、email。但是，这些有些细微变化，对于有些人，你需要记录他们父母的地址，另外，一个人有日间工作和夜间工作的电话号码。实现时预测所有这些情况是很困难的，你每次改变系统，都不得不重新编译、测试和发布。为了应付这种情况，你需要使用动态特性。

动态特性

如何表现对象的细节？

提供一个可以参数化的属性，这个属性可以代表依赖于参数的不同特性。

优点：可以在运行时增加特性

缺点：接口不清晰

动态特性有不同的变种，它们都要在灵活性和安全性之间作不同的折衷。最简单的手段是*柔性动态特性*。这个模式的本质是给 person 对象增加一个限定关联，其键是一个简单值，通常是一个字符串（见图 2 和表 2）。如果你想给 Kent 添加一个假期地址，你只需使用表 3 中的代码，不需要重新编译 person 类。你甚至可以创建一个 GUI 或一个文件阅读器，通过它们添加特性而无需重新编译客户端。

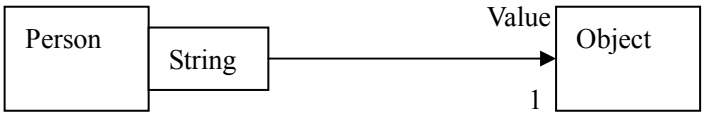


图 2 柔性动态特性的 UML 模型

```
class Person {

public Object getValueOf(String key);

public void setValueOf(String key, Object value);
```

表 3 图 2 的 Java 方法

```
kent.setValueOf("VacationAddress", anAddress);

Address kentVactation = (Address) kent.getValueOf("VacationAddress")
```

表 3 使用动态特性

如前所述，你也许对为何每个人总是使用固定特性感到奇怪，如同此处给出的例子，动态特性给你许多灵活性，当然，这是有代价的，它降低了各个软件部分之间依赖的清晰性。给人添加一个假期地址是件好的事情，但是你知道如何把它取回来吗？用固定特性，你只需查看 person 的接口并查看其特性。编译器能够检查特性，而不要求对象去完成它不明白的事情。用动态特性，你就不能进行设计时检查。而且 person 的接口更不易了解。不仅要查看 person 申明的接口，还得寻找动态特性，但它并不在类的接口之中。你得寻找设置特性的那部分代码（这些代码通常根本不在 Person 类中），并将它提取出来。

柔性动态特性

如何表现对象特性？

提供一个用字符串参数化的属性。申明一个只使用字符串的特性。

它不仅难于找到特性，而且还创建了一个可怕的相关性。使用固定的特性，客户代码具有和 person 类的相关性——一个易于记住的相关性。如果你改变了特性的名称，编译器会让你知道，并告诉你需要修改什么代码来补救。但是，*柔性动态特性*创建了与某些任意代码片段的相关性，这些代码可能属于一个类，它们对客户是完全不可见的。如果有人改变了关键字符串，会发生什么事情？如果有人改变了放入关键字符串中的对象类型，会发生什么？不仅编译器无法帮助你，你甚至不知道从何处着手寻找这些潜在的改变。

柔性动态特性展示了最极端情况的问题。特性可能会在设计时被 Person 的任何客户代码创建。如果 Person 的其他客户代码使用同一个特性，两个类之间就有了相关性，这种情况很难发现。而且特性可以在运行时通过读取文件或 GUI 加入。在运行时不可能找出对于 person 而言哪些是合法的动态特性。诚然，你可以问一个人是否具有假期地址的特性——但是如果没有的话，那意味着那个人没有假期地址，还是没有假期地址的特性？他现在没有这样一个特性，并不意味着在几秒钟后他就不能具有这样的特性。

柔性动态特性另一个主要的缺点是替代它们的操作很困难。封装的主要优点是，客户使用一个特性，但并不知道它是存储的部分对象数据，还是用某个方法计算出来的。这是面向对象方法非常重要的部分。它不仅允许你给两种目的一个标准接口，也使得改变你的想法不需要让客户知道。在子类型定义的时候，甚至可以用父类型来保存特性，用子类计算特性，或是相反。无论如何，如果你要使用动态特性，针对计算可以改变保存数据的唯一方法是如表 4 中所述，在通用访问者对象中为动态特性放置一个特殊的中断。这样的代码很可能是脆弱的，并难

于维护。

```
class Person {
public Object getValueOf (String key) {
    if (key = “vacationAddress”) return calculatedVacationAddress();
    if (key = “vacationPhone”) return getVacationPhone();
    // else return stored value…
```

表 4 用一个操作替换一个动态特性

其他形式的动态特性可以帮助你解决部分但不是全部问题。动态特性的根本缺点是失去了清晰的接口和全部设计时检查。不同的动态特性解决方法给予不同能力的运行时检查。如果你需要动态特性，并且存在明确的情况需要这样做，那么你就必须放弃设计时检查和显式的设计时接口。唯一的问题是，接口有多清晰，以及在运行时可以做多少检查。使用柔性动态特性这两者都不具备。

通常是在数据库中发现动态特性，因为改变数据库方案是件痛苦的事，特别在有很多数据要迁移的时候。分布式组件的接口，比如 CORBA，因为类似的原因使用动态特性。有很多远程客户在使用接口，所以你不愿意改变接口。在这两种情况下，编译时和运行时没有太大的区别，与设计时和产品之间的区别差不多相同。

如果你正在做的所有事情是通过一个 GUI 来显示和修改信息，并且代码没有固定地引用键值（例如：没有看到类似于表 3 那样的代码），那么你使用*柔性动态特性*是非常安全的。这是因为你还没有和某个用作键值的任意字符串建立起糟糕的相关性。否则你应当考虑其他的动态特性处理方法。

朝着更多的运行时检查前进的第一步是*确定性动态特性（Defined Dynamic Property）*。确定性动态特性和柔性动态特性相反，它们之间的关键差别是：动态特性使用的键值不再是某个任意字符串，而是某个类的实例（图 3）。

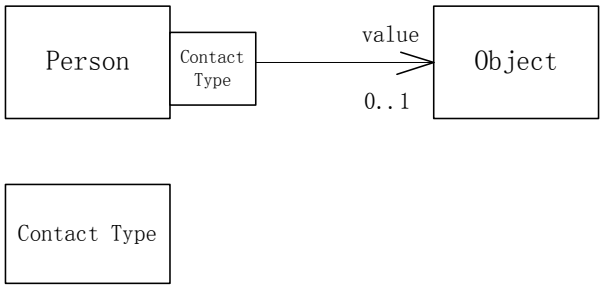


图 3 确定性动态特性

看来使用确定性动态特性不会改变很多东西。事实上，代码接口几乎是完全相同的（表 5 和表 6）。但是，键值的选择不再具有任意性，它由 contact type 的实例限制。当然，它仍然允许你在运行时添加特性——只需创建新的 contact type。无论如何，至少现在存在某些地方让人去查看，得到潜在的键值而无需搜索整个程序。在设计时增加的任何键值可以搜集到 contact type 类的一个装载例程中。你可以容易地提供服务，以便在运行时找出合法的键值。

```
class Person {
public Object getValueOf(ContactType key);
public void setValueOf(ContactType key, Object value);
```

表 6 图 3 的 Java 接口

```
class ContactType {
public static Enumeration instances();
public static boolean hasInstanceNamed(String name);
public static ContactType get(String name);
```

表 6 确定性特性类型的服务

特别地，你现在可以设置一些检查，以防止由于某些人请求表 7 中不存在的动态特性而产生的错误。在这里抛出一个未经检查的异常，因为我考虑 get()的前提条件是客户提供一个合法的 contact type 名称。客户总是可以通过使用 hasInstanceNamed()来履行这个责任，但是，大部分时间客户软件使用的是 contact type 对象，而不是字符串。

确定性动态特性

你如何表达一个对象的细节？

提供用某些类型的一个实例参数化的属性。申明创建该类型的新实例特性。

通常，contact type 保存在字典里，通过一个字符串来索引。这个字典可以是 contact type 的静态字段，我更喜欢使它成为 *Registrar* 上的一个字段。

删除一个 contact type 仍然存在难于处理的后果。除非你写一些复杂的清理代码，在 Java 中，动态特性仍将出现在那些拥有它们的对象中。我通常以不删除确定性动态特性的键值为原则。如果你想使用新的名称，可以通过给 contact type 一个名字，从而很容易地给它一个别名，但是这个 contact type 在定义字典里要重复几次。

```
class ContactType {
public static ContactType get(String name) {
    if (! hasInstanceNamed (name)) throw new IllegalArgumentException(“No
    such contact type);
    // return the contact type
}
// use with
Address kentVactation =
```

(Address) kent.getValueOf(ContactType.get(“VacationAddress”));

表 7 检查合法 contact type 的用法

此时，你可能感到疑惑：这和概念建模有什么关系，毕竟，我写了很多代码并在讨论对设计方法的折衷选择。这是一个重要的概念问题，因为你所做的概念抉择影响实现方法的选择。如果你在概念模型中选择使用柔性动态属性，会导致在实现中使用确定性动态属性或是固定属性变得非常困难。概念建模的原因之一是探寻在用户的概念中，什么是固定的、什么是可改变的。如果完全的灵活性是唯一的目标，那么，我将会总是用图 4。使用该模型可以模拟世界上的任何情况。但这个模型用处不大，其无用性来自这样一个事实：它不能指出什么是固定的。当你进行概念建模的时候，你要明白选择如何影响实现中的事物——否则你就放弃了作为建模者的责任。

人们经常先找到动态特性，然后思考在每个地方使用这些特性。灵活性是如此的美妙，人们能够得到所有他们想要的可扩充性。诚然，有时你需要动态特性，但是不要忘记这是有代价的。只有当你真正需要的时候才使用它们。毕竟，如果你不得不采用的时候再增加动态特性很容易。

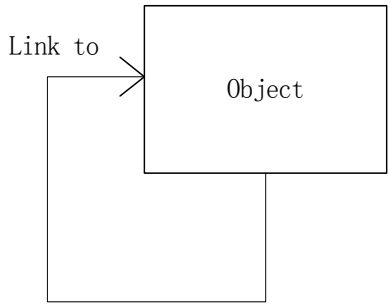


图 4 可以毫无用处地模拟任何领域的模型

*确定性动态特性*允许更多地指明你有什么特性。这些特性仍然是隐式类型的（untyped）。你不能强制使图 3 中假期地址的值成为一个地址。你可以通过使用类型化动态特性（Typed Dynamic Property）完成一些这样的事情。

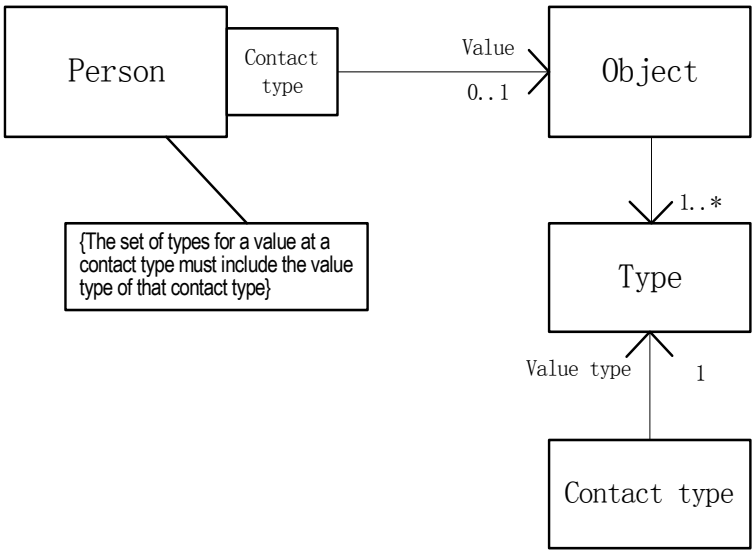


图 5 使用限定关联的分类的动态特性的模型

类型化动态特性向确定性动态特性增加了类型信息（图 6 和图 5）。在这里 contact type 的实例不仅表明 person 有什么特性，而且这些实例也表明每个特性的类型。参照表 9 中的代码，类型限制了值。

```
class Person {
    public Object getValueOf(ContactType key);
    public void setValueOf(ContactType key, Object value);
    class ContactType {
    public Class getValueType();
    public ContactType (String name, Class valueType);
```

表 8 对类型化动态特性的操作

```
class Person {
public void setValueOf(ContactType key, Object value) {
    if (! key.getValueType().isInstance(value))
    throw IllegalArgumentException ("Incorrect type for property");
    // set the value
```

表 9 类型检查

像这样进行类型检查有助于避免错误，但仍然不如固定特性清晰。在运行时而不是设计时进行检查，因此效果并不相同。但优于根本不进行检查，特别是在你习惯于强类型环境的情况下。

类型化动态特性

如何表达一个对象的细节？

提供用某个类型的实例参数化的属性。声明创建类型的新实例特性，并具体指明特性的值类型。

在深入研究动态特性的时候，我们找到丰富的*反射（reflection）*案例，它们是在我们获取能进行自我描述的运行时对象的时候出现的结构模式。[POSA]对反射所做的讨论比我在这里所做的更加细致。

动态特性提供了反射能力，甚至在那些本身不支持反射的语言里也是如此。实际上，使用提供某种反射的语言，动态特性的反射变得更清晰——这样你就可以提供一个更易于使用的接口。

使用限定关联会更难跟踪。提供*类型化动态特性*的另一个方法是使用*分立特性（separate property）*模式。分立特性模式的本质是使特性为一个对象所固有（图 6 和表 10）。你可以找到一个 person 的特性，然后用特性得到值和类型信息。

分立特性

如何表达一个对象的细节，并让有关该细节的其它细节被记录下来？

为每个特性创建一个独立的对象，那么，特性的细节就成为对象的特性。

---

*分立特性*和*限定关联*一直是动态特性两个可用的选择。至此，我已经用有限关联描述了柔性和确定性动态特性，因为有限关联表示了更易于使用的接口。如果你希望的话，你可以和分立特性一同使用柔性和确定性动态特性，在此，我将不讨论这种情况。当我们认识到类型化动态特性的复杂性，进一步的分立特性风格变得更具优势。

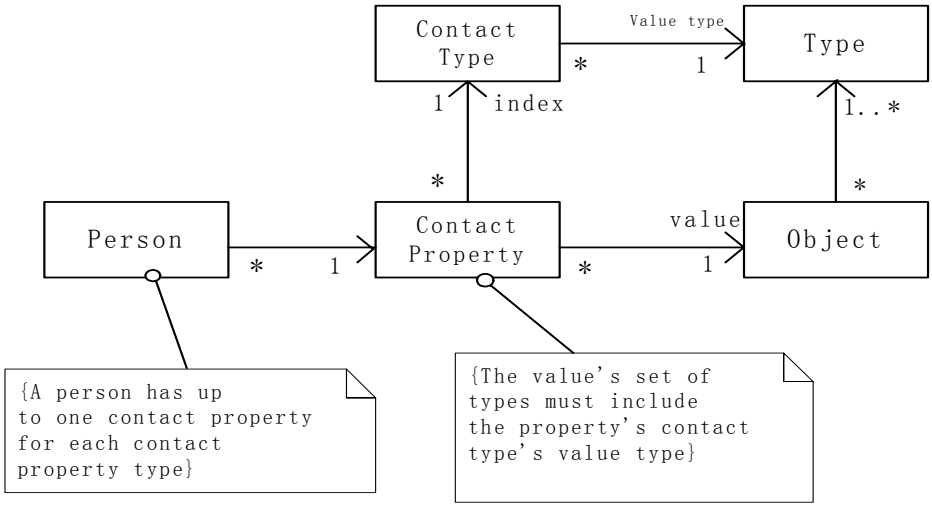


图 6 使用了一个独立特性的分类动态特性的模型

```
class Person {
    public Enumeration getProperties();
    class ContactProperty {
    public Object getValue();
    public Class getType();
    public ContactType getIndex();
```

表 10 对使用独立特性的分类动态特性的操作

分立特性和限定关联不是互斥的。同时提供这两种接口非常容易，这样，你可以利用两者的优点。当然，接口会变得更复杂，所以要先考虑 person 的客户程序需要什么，给他们所需的接口，不要让它过于复杂。但是如果他们同时需要限定的关联和分立特性，那么这是合理的选择。通常情况下，对值使用限定关联，然而它对类型不具有同样的意义。

你在这里也可以考虑接口/实现差别。在本文中，我希望集中于概念性的情形——映射软件的接口，而不是软件的实现。值得一提的是：在实现中使用独立对象时，你可以提供一个限定关联接口。如果 person 的客户程序可以得到 contact property 对象，只能在接口中使用分立特性。通常，隐藏分立特性以简化对客户程序的接口很有用处。

分立特性的优点之一是允许将特性的信息置于特性中。这类信息可以包括：谁决定特性，是什么时候决定的，以及类似的信息。建立在该主题之上的观察模式（Observation pattern ）[Fowler AP §3.5]进行了更深入的讨论。如果你需要分立特性，我在那篇文章中针对观察所描述的很多内容都值得考虑。（我认为把观察模式可看作分立特性的一种应用）。

你也许对分立特性（一种模式）和限定关联（一种 UML 建模构造体）之间的对比感到惊讶。也可以把限定关联看作是一种模式，一种关联的模式。事实上，我在[Fowler AP §15.2]中就是这样做的。我发现将建模构造体看作模式很有益处，它帮我考虑折衷使用分立特性和限定关联。当你把这些构造体和某种更清晰的模式（例如分立特性）相比较的时候特别有用。当然，最初是模式的东西也可以转变为建模构造体，特别在使用 UML 构造型（stereotype）的时候。历史映射模式（historic mapping pattern）[Fowler AP §15.3]就是这样一个例子。我是用历史（history）构造型表示它的。它是模式还是建模构造？也许它还是地板腊和沙丘（desert topping）。

有多值关联的动态特性

上述几个例子都集中于动态特性中的每个键有一个单值的情况。但是也会存在动态特性具有多项的情况。用 Person 时你可以考虑多值的 friends 特性。有两种方法处理这种情况，一个简单但是不能令人满意，另一个令人满意但是（太）复杂。

简单方法就是，动态特性的值是一个集合。那么我们可以像操作其它具有同样接口的对象一样操作（表 11）。这样简单些，因为我们不必对基本动态特性模式作任何改动。（这里的例子有一个类型化动态特性，对于其它任何动态特性，同样可以。）无论如何，它还是不能让人满意，因为它并非我们用来处理多值特性的真正方法。如果 friends 是固定特性，我们想要一个表 12 中那样的接口。我不喜欢暴露这些例子中的向量（vector）。这样做的

话，当我们添加和删除元素的时候，person 类失去了反应（react）能力。也使我们失去了改变使用的集合类型的能力。

```
Person aPerson = new Person();
ContactType friends = new ContactType(“Friends”, Class.forName(“Vector”));
Person martin = new Person(“Martin”);
martin.setValueOf(“Friends”, new Vector());
Person kent = new Person(“Kent”);
martin.getValueOf(“Friends”).addElement(“Kent”);
Enumeration friends = martin.getValueOf(“Friends”).elements();
```

表 12 在分类的动态特性中使用集合值

```
class Person {
    public Enumeration getFriends();
    public void addFriend(Person arg);
    public void removeFriend(Person arg);
```

表 12 对固定多值特性的操作

当存在动态特性时，就能够得到表 12 中的接口吗？答案是肯定的，如果我们努力尝试的话，如图 7 所示。但它是一个复杂模型。使用精巧的编码，我们可以把大部分复杂性隐藏在接口后面（表 13 和表 14），并且非常容易使用（表 15）。但是，客户程序仍然需要知道哪些特性是单值的，哪些是多值的，并且只能在运行时对用法的正确性进行检查。所有这样的复杂性是令人难以忍受的——使用固定特性更令人痛苦。我非常不愿意这样做。

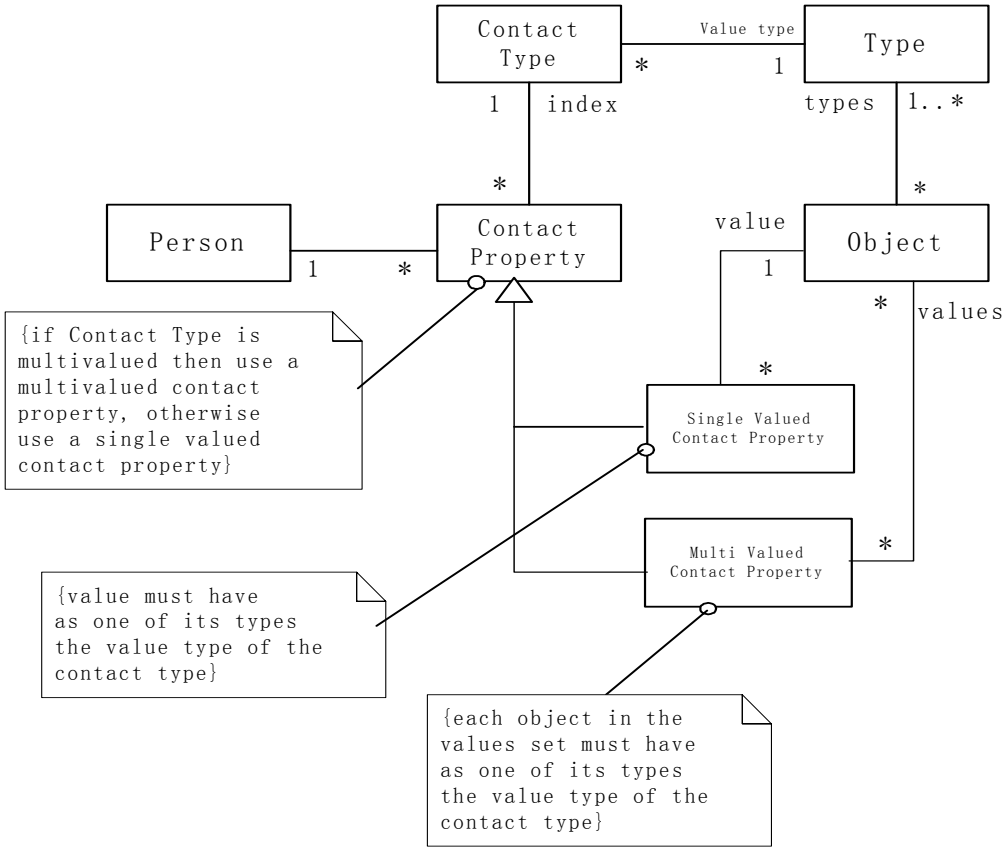


图 7 对多值动态特性令人满意但是过于复杂的支持

```
class Person

    public Object getValueOf(ContactType key);

    public Enumeration getValuesOf(ContactType key);

    public void setValueOf(ContactType key, Object newValue);

    public void addValueTo(ContactType key, Object newValue);

    public void removeValueFrom(ContactType key, Object newValue);
```

```
class ContactType
public Class getValueType();
public boolean isMultiValued();
public boolean isSingleValued();
public ContactType(String name, Class valueType, boolean isMultiValued);
```

表 14 对图 7 的操作

```
class Person
public Object getValueOf(ContactType key) {
    if (key.isMultiValued())
    throw IllegalArgumentException("should use getValuesOf()")
    //return the value
}
public void addValueTo(ContactType key, Object newValue) {
   if (key.isSingleValued())
   throw IllegalArgumentException("should use setValueOf");
   if (! key.getValueType().isInstance(newValue))
   throw IllegalArgumentException ("Incorrect type for property");
   //add the value to the collection
```

表 14 对表 13 中操作的使用进行检查

```
    fax = new ContactType("fax", Class.forName("PhoneNumber"), false);
    Person martin = new Person("martin");
```

martin.setValueOf(“fax”, new PhoneNumber(“123 1234”);

martinFax = martin.getValueOf(“fax”);

friends = new ContactType (“friends”, Class.forName(“Person”), true);

martin.addValueTo(“friends”, new Person(“Kent”));

表 16 使用表 13 中的操作

出现这种复杂性是因为我们同时有多值和单值特性，存在固有差别的接口处理这些特性，因而产生了复杂性。当然我们可以得到只有多值特性的情况。这种情况的通用模式是类型化关系（Typed Relationship）模式（图 8）。在此，一个人可以和不同的公司有不同的雇佣关系（或者和同一个公司有多种关系）。

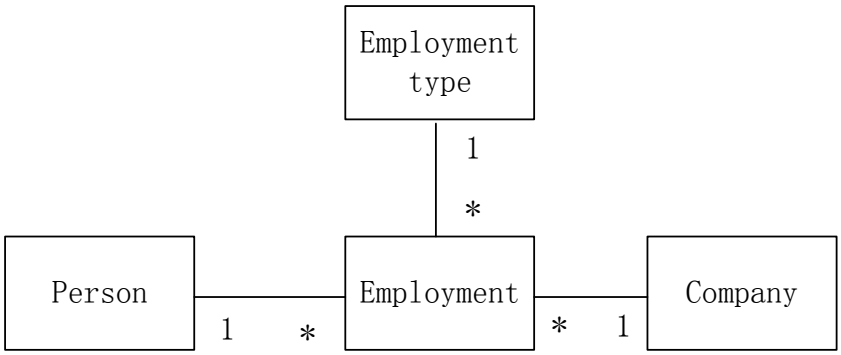


图 8 类型化的关系例子

class Employment {

public Employment (Person person, Company company, Employment Type type);

public void terminate()

…}

class Person {

public Enumeration getEmployments();

public void addEmployment (Company company, EmploymentType type);

表 16 图 8 的 Java 接口

在我们考虑这种情况的时候，你可能很快想到，它实际上与使用一个多值*确定性动态特性*非常相似，只是用一个*分立特性*而不是一个*限定关联*来表达。（或者这句话只是太适当的批评呢？）实际上，图 9 显示了在这种情况下如何使用确定性动态特性接口。这种对事物的看法是真实的，因此类型化关系（typed relationship）不会给这种模式语言添加任何新东西。但是类型化关系在建模周期（modeling circles）中是非常普通的模式，并且很多建模周期无法实现类型化关系和动态特性模式之间的联系。

类型化关系

如何表达两个对象间的关系？

(如何表达多值动态特性？)

为两个对象间的每个连接创建一个关系对象。给关系对象一个类型对象指明关系的意义。（类型对象是多值特性的名字。）

类型化关系的优点是：它和双向关系一起工作得很好，并且它提供向关系添加特性的简单方法。（当然，后者是分立特性的一个特征。）使用大部分与*类型化动态特性*相同的代码，你可以给这个模式添加一个高级的知识层。无论如何，你应当考虑接口实现。类型化关系强迫用户明白 employment（雇用）对象实际上同样使用分立特性。实际上，人们希望特性对象是完全独立的，而不是 person（或者 company）的某个特性。但是限定关联常常为很多目标提供一个简单的接口。所以无论什么时候，你看到或是在考虑使用类型化关系，你应当考虑限定关联形式。可以在一个或两个方向上使用限定关联，可以不使用类型化关系而只用限定关联，或是只使用类型化关系而不用限定关联。

毕竟这两种模型不完全相同。如果你使用图 9，你就表明了雇主（employer）对于特定的雇佣类型（employment type）只能是雇主。除非 employment 有附加的属性，如图 8 中表示的那样，它没有此类约束，尽管大多数建模者暗指了这样一个约束。当然，通常 employment（雇佣）没有附加属性。一个常见的例子是期限（如同 accountability 中的一样[Fowler, AP §2.4]）。

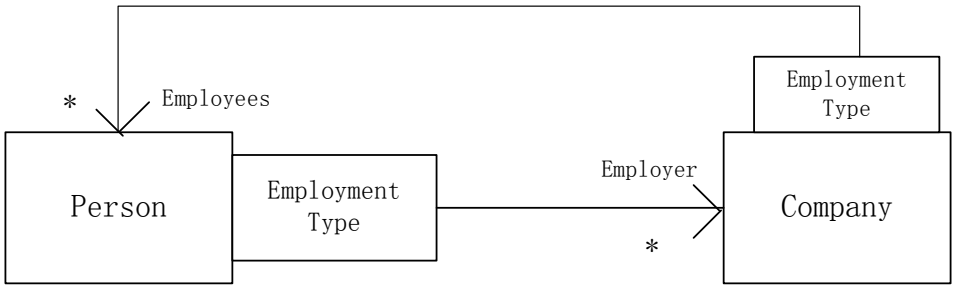


图 10　对与图 8 中相同的情况使用限定关联

public Person

public void addEmployer (Company company, EmploymentType type);

public Enumeration getEmployersOfType (EmploymentType type);

表 17 图 9 的接口

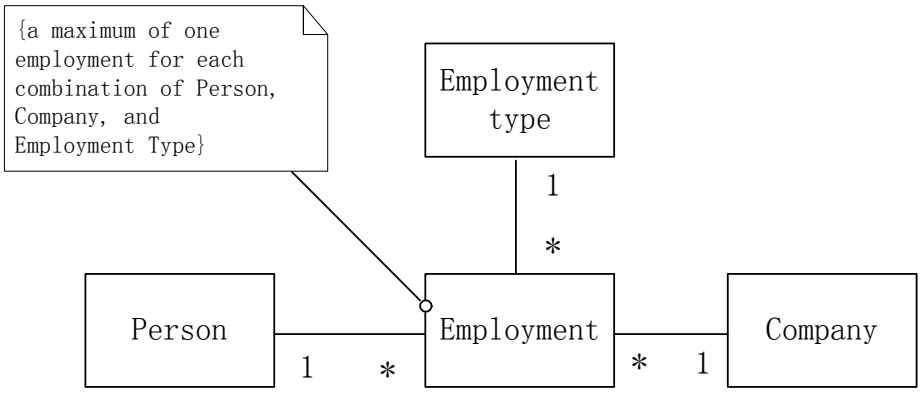


图 10 分类的关系，展示了通常被假定的约束.

不同类型的人

到目前为止，我们都假定只有一类人，我们为一个人定义的任何特性针对所有人。但是，你会碰到不同类型的人的情况，不同类型的人有不同的特性。一个经理可能需要他所管理部门的特性，一个管理人员可能需要管理洗手间钥匙的数目特性（在一个非 90 年代的公司里）。

不用担心，我听到了“使用继承，笨蛋”的喊声。实际上，这是通常用于子类型（subtyping）情况。实际上它比子类型更复杂，特别是当你开始考虑一个人可以承担不同角色的时候。关于对角色建模（modeling roles），我写过一篇完整的论文[Fowler roles]。角色模式考虑那些对操作变化以及固定特性变化感兴趣的情况。但是，

在这个例子中，我想研究具有动态特性概念的不同对象间的重叠。这种重叠产生了*动态特性知识层（dynamic properties knowledge level）*模式（对于我的嗜好而言，该名字太长）。

要使用这个模式，我们给 Person 定义一个 Person type *类型对象（type object）*。然后可以说 person type 到 contact type 的关联表明了对于有 person type 的 people 哪个特性是可用的。如果我们试图使用或请求一个 person 的特性，person type 就可以用来检查用法正确与否。

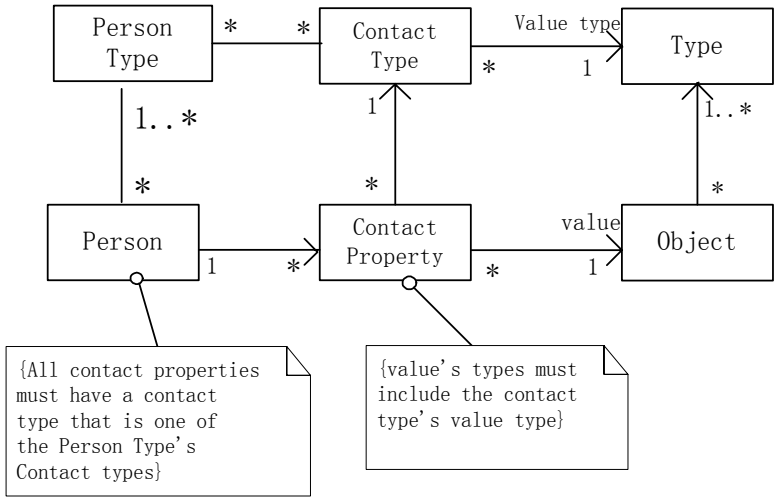


图 11 动态特性知识层（Dynamic Property Knowledge Level）

```
class Person {

    public Object getValueOf(ContactProperty key);

    public boolean hasProperty(ContactProperty key);

    public void setValueOf(ContactProperty key, Object newValue);

    class PersonType {

    public boolean hasProperty(ContactProperty key);

    public Enumeration getProperties();
```

表 18 图 11 的操作

```
class Person {
    public Object getValueOf (ContactProperty key) {
    if (!hasProperty(key))
    throw IllegalArgumentException(“Innapropriate key”);
    //return the value
```

表 19 用动态特性知识层检查一个合适的键值

当我们像这样使用知识层时，分立特性变得越来越重要。在这个例子中，我们很快不再把它看作一个特性，而是把它看作某个对象。什么是特性和什么不是特性之间的界限非常模糊，实际上取决于你对事物的看法。

动态特性知识层

当你使用动态特性的时候，你如何强制特定类型的对象具有特定的特性？

创建一个知识层包含什么类型的对象使用哪种类型特性的规则。

对动态特性的总结

不同类别的动态特性构成了本文的大部分内容。但是，不得不反复强调的是：动态特性是我尽可能避开的东西。动态特性带来大量的缺点：缺乏接口的清晰性，用操作代替存储数据时的困难。它只是某些时候你很少有选择余地，不得不使用的方法。此时，本文应当证明对你是有用的，它会给你一些选择以及在这些选择之间做出怎样的折衷。

动态特性出现在改变接口有很多困难的地方。在分布式对象系统上工作的人喜欢动态特性，至少是在理论上是如此，因为动态特性允许人们改变接口，而无需迁就客户程序——在分布式系统中找出谁是你的客户程序非常困难。但是使用动态特性的时候仍然要谨慎。在为动态特性的一系列键值添加新键值的时候，就是在有效地改变接口。所有动态特性正在进行的就是用运行时检查代替编译时检查。依然存在保持你的客户程序为最新状态的问题。

经常用到动态特性的另一个地方是数据。不仅仅是因为接口的问题，而且因为数据迁移的问题（如果不是主要原因的话）。改变一个数据库表（schema）不仅产生潜在修改使用该表的程序的问题，而且可能迫使你进行复杂的数据转换工作。动态特性允许你改变一些东西而不需要修改数据库表，因此也不需要做任何的数据转换。在

大型数据库中，这是引人注目的优点。

一个不需要知道的特性

最后一个也是很重要的特性，我想加在这篇文章里。就是对象有一个特性但是没有实现该特性的案例。当特性被其它对象以及它和你所关联的方法所暗指的时候，就会出现这种情况。

考虑管理数据库连接的对象。它创建一批数据库连接，并在接收到请求的时候把这些连接交给其它对象。当一个客户程序使用连接完成了任务，它可以把连接还给管理者，以便其它客户程序使用。图 12 显示了使用*外在特性（extrinsic property）*的选择。连接是空闲还是繁忙由连接所在的连接管理器决定。就好像如果你有一个连接，你不能问这个连接空闲还是繁忙，相反，你将不得不询问连接管理器特定的连接空闲还是繁忙。你能说明最初的连接状态，是因为组成关联（composition association）是单向的。连接不知道连接管理器。在某种意义上，空闲/繁忙状态根本不是连接的特性。然而至少在某些意义上它是连接的特性，这也是我为什么提及它的原因。

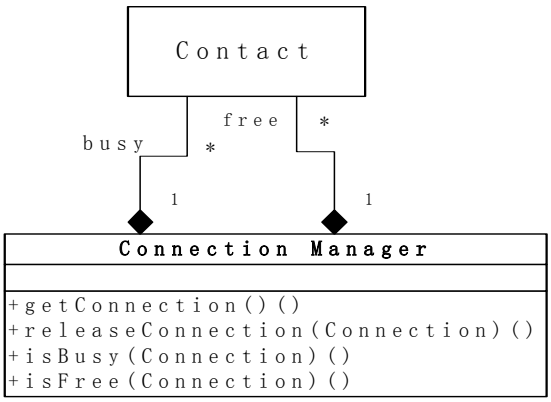


图 12 使用外部集合特性

在纯概念建模的意义上，这个模式没有意义。但有一些可能使用它的现实的实现原因。如果你想让对这个特性所做的所有改变都经过连接管理器，那么这个方法很清晰。特别是在你想让连接管理器给你一个空闲的连接，而你不关心具体是哪个连接的时候，这是很自然的方式。

使用*外在特性*的另一个原因是：如果连接类由其他人提供，而你不能改变其接口。你可以增加新的特性，根本不需要改变连接类。

外在特性

怎样给对象一个特性而不改变它的接口？

构造另一个对象负责了解特性。

使用外在特性的一个大问题是:它导致笨拙的、不自然的接口。通常如果你想知道什么事情，你只要找到合适的对象，并询问它。在此，你需要寻找拥有外部集合的对象，并向它询问合适的对象。在某些情况下它好像是合理的，如这个例子。但是大多数时间，我更愿意让对象知道它们自己的特性（在这种情况下，我把它们称为*内在特性(intrinsic property)*）。

最后的思考

在我完成本文章的时候，再次认为需要劝告你不要使用我在这里写的东西，除非你真正需要它。固定特性的优点非常好。如果你需要别的什么，那么我希望这篇文章可以给你一些想法和某些指导。但是固定特性总是你的首选。

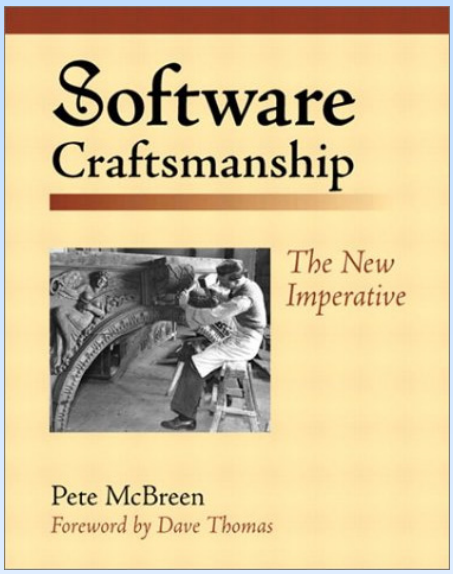
参考文献

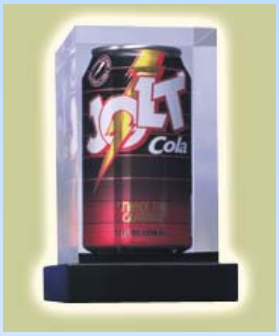
[Fowler, AP] Fowler, Martin. *Analysis Patterns: Reusable Object Model*s, Addison-Wesley 1997

[Fowler, roles] Fowler, Martin. *Dealing with Role*s, http://www.awl.com/cseng/titles/0-201-89542-0/awweb.htm

[POSA] Buschman et al, Pattern Oriented Software Architecture, Wiley 1997

《软件工艺》





2002 Jolt Awards 获奖书籍

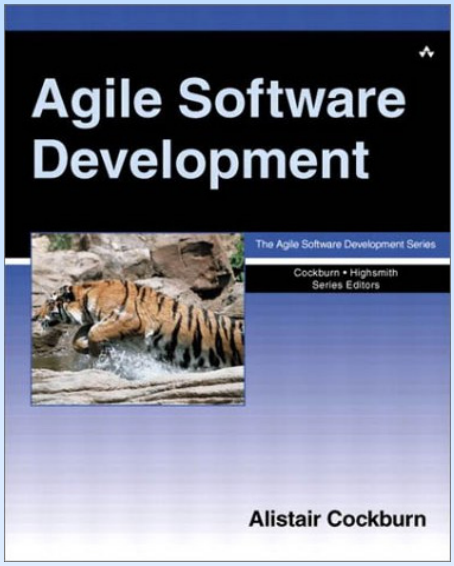
Pete McBreen

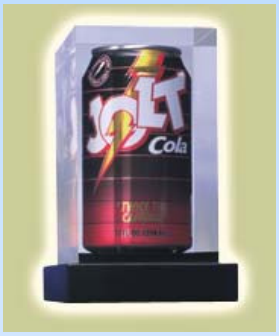
翻译：UMLChina 翻译组

工艺不止意味着精湛的作品，同时也是一个高度自治的系统－－师傅（Master）负责培养他们自己的接班人；每个人的地位纯粹取决于他们作品的好坏。学徒（Apprentice）、技工（Journeyman）和工匠（Craftsman）作为一个团队在一起工作，互相学习。顾客根据他们的声誉来进行选择，他们也只接受那些有助于提升他们声誉的工作。

中文译本即将发行！

《敏捷软件开发》





2002 Jolt Awards 获奖书籍

Alistair Cockburn

翻译：UMLChina 翻译组 Jill

我是一个好客人。到达以后，我把我的那瓶酒递给女主人，然后奇怪地看着她把酒放进了冰箱。

晚餐时她把酒拿了出来，说道，“吃鱼时喝它，好极了。”

“但那是一瓶红酒啊。”我提醒她。

“是白酒。”她说。

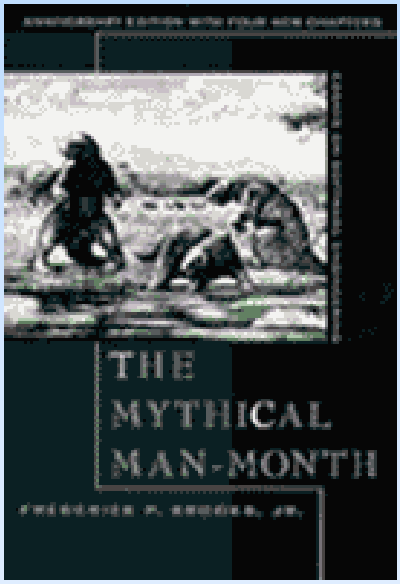
“是红酒。”我坚持道，并把标签指给她看。

“当然不是红酒。这里说得很明白...”她开始把标签大声读出来。“噢！是红酒！我为什么会把它放进冰箱？”

我们大笑，然后回顾我们为了验证各自视角的“真相”所作的努力。究竟为什么，她问道，她已经看过这瓶酒很多遍却没有发现这是一瓶红酒？

中文译本即将发行！

《人月神话》



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 Adams Wang

散文笔法，绝无说教，大量经验融入其中

在所有恐怖民间传说的妖怪中，最可怕的是人狼，因为它们可以完全出乎意料地从熟悉的面孔变成可怕的怪物。为了对付人狼，我们在寻找可以消灭它们的银弹。

大家熟悉的软件项目具有一些人狼的特性（至少在非技术经理看来），常常看似简单明了的东西，却有可能变成一个落后进度、超出预算、存在大量缺陷的怪物。因此，我们听到了近乎绝望的寻求银弹的呼唤，寻求一种可以使软件成本像计算机硬件成本一样降低的尚方宝剑。

但是，我们看看近十年来的情况，没有银弹的踪迹。没有任何技术或管理上的进展，能够独立地许诺在生产率、可靠性或简洁性上取得数量级的提高。本章中，我们试图通过分析软件问题的本质和很多候选银弹的特征，来探索其原因。

中文译本即将发行！

通俗而且深刻的软件开发思想杰作—UMLChina 推荐

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

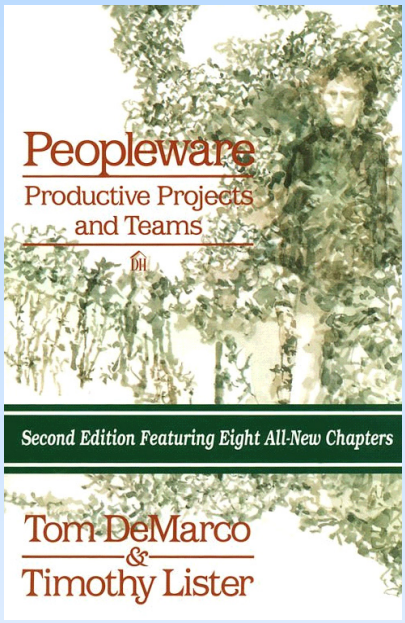
翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

面向对象技术在给我们带来好处的同时，也会增加成本，其中很大一部分是培训费用。经验表明，一个不熟悉 OO 编程的新手需要 3 个月的培训才能胜任开发工作，也就是说他拿一年的薪水，却只能工作 9 个月。这对一个拥有成百上千个这样的程序员的公司来说，费用是相当可观的。一些公司的主管们可能一看到这么高的成本立刻就会说"不能接受。"由于只看到成本而没有看到收益，他们会一直等待下去，直到面向对象技术过时。这本书不是为他们写的，即使他们读了这本书也会说（其实也有道理）"我早就告诉过你，采用 OO 技术需要付出昂贵的代价以及面临很多的危险。"另外一些人可能会决定启动一个采用 OO 技术示范项目，并观察最终结果。还有人仍然会继续在原有的程序上修修改改。当然，也会有人愿意在这项技术上赌一赌。

中文译本即将发行！

《人件》



《人件》第 2 版

Tom Demarco 和 Tim Lister

翻译：UMLChina 翻译组方春旭、叶向群

微软的经理们很可能都读过—amazon.com

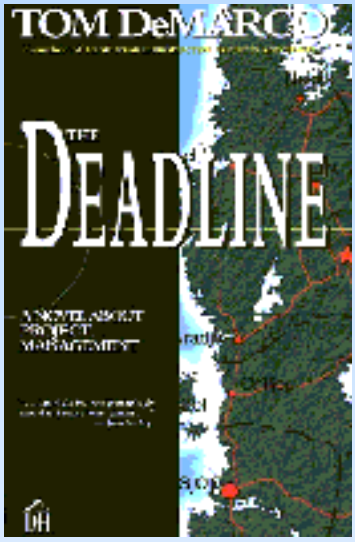
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

许多开发经理采用了类似的态度，他们竭尽全力地使自己确信没有人能够取代自己。由于害怕一个关键人物要离开，他强迫自己相信项目组里没有这样的关键人物，毕竟，管理的本质是不是取决于某个个人的去留问题？他们的行为让你感到好像有很多人物储备在那里让他随时召唤，说"给我派一个新的花匠来，他不要太傲慢。"

我的一个客户领着一个极好的雇员来谈他的待遇，令人吃惊的是那家伙除了钱以外还有别的要求。他说他在家中时经常产生一些好主意但他家里的那个慢速拨号终端用起来特别烦人，公司能不能在他家里安装一条新线，并且给他买一个高性能的终端？公司答应了他的要求。在随后的几年中，公司甚至为这家伙配备了一个小的家庭办公室。但我的客户是一个不寻常的特例。我惊奇的是有些经理的所作所为是多么缺少洞察力，很多经理一听到他们手下谈个人要求时就被吓着了。

中文译本即将发行！

《最后期限》



《最后期限》

Tom Demarco

翻译：UMLChina 翻译组 透明

这是一本软件开发小说

汤普金斯在飞机的座位上翻了一个身，把她的毛衣抓到脸上，贪婪地呼吸着它散发出的淡淡芬芳。文案，他对自己说。他试图回忆当他这样说时卡布福斯的表情。当时他惊讶得下巴都快掉下来了。是的，的确如此。文案……吃惊的卡布福斯……房间里的叹息声……汤普金斯大步走出教室……莱克莎重复那个词……汤普金斯重复那个词……两人微张的嘴唇触到了一起。再次重播。"文案。"他说道，转身，看着莱克莎，她微张的嘴唇，他……倒带，再次重播……

....

"我不想兜圈子，"汤普金斯看着面前的简报说，"实际上你们有一千五百名资格相当老的软件工程师。"

莱克莎点点头："这是最近的数字。他们都会在你的手下工作。"

"而且据你所说，他们都很优秀。"

"他们都通过了摩罗维亚软件工程学院的 CMM 2 级以上的认证。"

中文译本即将发行！

根据合同进行分析--录像店案例研究

Richard Mitchell 著，zhen lei 译

查看评论

本文包括录像店案例研究的一些片段，用来说明根据合同进行分析的原理。本文假定读者已经从其它渠道学习了一些关于根据合同进行分析的方法。

完整的一套模型可以写成一本书。这些选择的片段用来说明开发的某些方面，其中精确的模型（采用 OCL）可以提供真正的支持，包括：

- 限制类型模型的冗余部分，以便容易描述其它定义
- 定义状态模型和类型模型间的关系
- 表述操作的预状态

这些模型从整个系统的角度建立，将系统视作黑箱。假定域模型中定义的所有事件都映射到系统操作中。

这个模型没有经过正式的外部评审，因此应看作可能存在缺陷。我们欢迎读者的反馈。

问题描述

录像店出租销售录像带。只有会员可以租借录像，任何人都可以购买。当录像没有出租价值时，录像店将它标记为可以出售。

如果希望成为会员，你要交纳一定的会费，然后得到一张带有条码的会员卡，条码表示你的会员编号。

录像出租期限是 1 至 3 天。租借天数越多，日租金就越低。如果超过出租期限，需要支付罚金，罚金高于一天的租借费用。在支付罚金之前，会员将被禁止租借。

如果想要的录像带暂时没有，会员可以预订。当录像带可以出租时，预订该录像带的会员就会收到通知，告诉他们可以在三天内租借该录像带。如果超过三天，录像带可以由其它会员租用。

问题域研究

我们可以从问题域中的事物入手对它进行研究，可以分为两部分，“存在的事情（事物）”和“发生的事情（事件）”。

存在的事情（事物）

以下是一些我们从问题域中发现的事物，可以模化为对象类型：

录像带，人，商店，租借合同，收据，日期、预订、通知 … …

发生的事情（事件）

我们可以通过考察场景来研究发生的事情。理想地，每个场景应该是承担一部分或周期性任务的行为，因此我们知道什么时候开始，什么时候结束。例如，我们可以对录像带出租的周期进行建模：

录像带被一个会员租借（或者不允许任何人预订）

另一个会员预订该录像带

第一个会员还录像带（结果包括：录像带归还，给第二个会员发通知）

第二个会员租借录像带

以上场景提出了一个问题，我们出租和预订的是否是同一个事物。实际上，现实世界中一个影片有多个拷贝，影片和拷贝不是同一事物。

域建模

随着列出的事件的增加，我们可以将它们分解到一组主题域中，这样我们可以集中精力分别研究系统特定的部分。例如，我们更愿意将会员如何预订以及如何成为会员分开考虑，虽然这两个部分并不是完全独立的。

事件——按主题域分类

出租（Renting）

会员租借录像带

会员还录像带

会员预订录像带（按片名）

会员取消预订

预订过期

出售（Selling）

顾客购买录像带

雇员将拷贝标记为销售

雇员将某一影片的所有拷贝标记为销售

管理存货（Managing　inventory）

雇员增加一部新影片

雇员增加一个拷贝

雇员删除一部新影片

雇员删除一个拷贝

管理会员（Managing membership）

雇员增加一个会员

雇员删除一个会员

其它需要建模的主题域

钱

支付

时间

推敲列表（Polishing the lists）

随着我们理解的增加，我们希望对一些实体重新考虑。例如，我们如何发现“预订失效”事件？我们可能发现“工作日结束”事件。我们可能发现所有预订都可能过期。我们可能要定义一个操作来确定预订失效的时间。

我们还可能决定，例如，将一个影片的所有拷贝标记为可销售是替代将每个拷贝进行标记的好方法，因此，可以仅关心将一个拷贝标记为销售即可。

从域建模转移到系统建模

当我们转到系统模型，必须考虑我们先前模型的范围。这意味着我们确定的主题域和技术域的范围。如果不能一次发布，这些决定必须与系统哪些部分先发布相关。我们假定出租部分先发布，我们的建模任务就围绕出租为核心，其它模型将考虑设备、用户界面、固定存储设备等。

缩略语

如果我们针对同一类型采用相同的描述方法和变量（如果需要，可以用 x1、x2… …），阅读模型就会更容易。

类型	变量
Copy（拷贝）	c
Duration（期限）	d
Fine（罚金）	f
Member（会员）	m
Notice（通知）	n
Payment（支付）	p
Rental（出租）	r
Reservation（预订）	v
Title（标题）	t
VideoStore（录像店）	vs

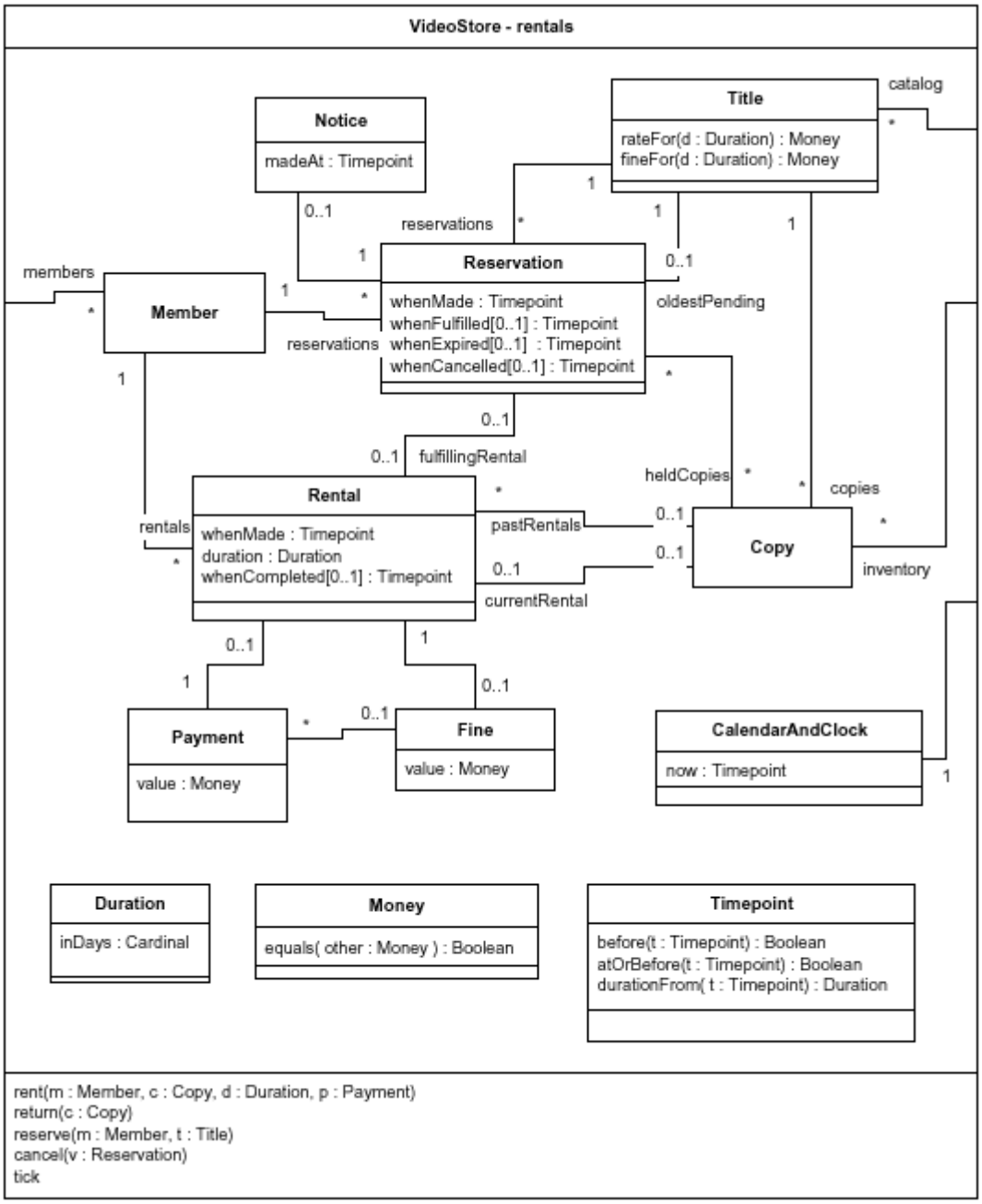
类型模型

接下来，类型模型并非在核心建模开始就进行。许多细节是在其它模型建立过程中出现的，如状态模型、不变性和操作说明。

在此，有些实例有助于说明模型为什么会是这样。

当一盘录像带返还并且有该录像带的预订，这个拷贝就会保留，然后通知预订该录像带的会员，告诉他可以租想要的录像带。因为我们只是为核心建模，所以如何给会员发通知超出了我们的研究范畴。在核心模型中，我们建立一个“通知”对象，留给其它领域考察核心并进行实现，比如，打印一封信。

有许多联想可以帮助我们谈论录像带拷贝保留以及针对预订出租。回忆一下出现在模型中事物的规范，实际项目中的人来确定这些事物。例如，如果我们的客户提到为特定的预订保留录像带，我们必须认真地考虑如何建模。我们必须仔细区分是针对每一个拷贝还是针对同一影片的一组拷贝进行预约保留。第一种情况（每一个拷贝进行预约保留）在没有计算机的情况下容易管理，（也就是说，将每个拷贝编个号，并且说明谁持有它即可）。第二种情况（同一影片的所有拷贝可以进行预约保留）需要计算机支持，拷贝可以放在按标题顺序的保留区中，计算机可以确定一个会员是否可以租借该影片的一个拷贝。



不变因素：方便查询

本节的不变因素所说的查询可以从其它查询中得出，但是它方便定义模型的其它部分。这里是一个例子。

最早的预订

一部影片可能有多人预订，每个预订包括一个日期和时间，一些预订可能还在排队，其中肯定有一个最早的。我们可以通过下面的不变因素获得这个信息。

-- In the context of a title, t,

t : Title

-- if the set of reservations that are still pending is not empty then 如果预订等待队列不为空，则

t.reservations -> select(pending) -> notEmpty implies

-- there is an oldest pending reservation 有一个最早的预订

t.oldestPending -> notEmpty

-- which is one of the title's pending reservations 找出该片名的预订

and t.reservations -> select(pending) -> includes(t.oldestPending)

-- and which was made at the same time as, or before, all of the title's pending reservations 找出最早的预订

and t.reservations -> select(pending) -> forall(v |

t.oldestPending.whenMade.atOrBefore(v.whenMade))

不变因素：约束

本节的不变因素排除某些粗糙类型模型的印象。我们可以非常松散地，通过将对象限制在组内或是按照对象间必然的联系将约束分组（这是一个松散的分组，因为限制在一组内的对象可能互相关联）。

组（Sets）

类型模型允许访问一组对象的两种方法均可访问对象。

录像店的库房是按照影片名称分类的拷贝：

-- In the context of a video store ,

vs : VideoStore

-- the set of copies that form the store's inventory 拷贝集合构成库存

vs.inventory =

-- is the set of copies of the titles in the catalog 按照片名分类的一组拷贝

vs.catalog.copies

录像店有一组出租的录像带

-- In the context of a video store ,

vs : VideoStore

-- the set of rentals held by members 一组会员租借的录像带

vs.members.rentals =

-- is the same as the set of rentals formed from the copies' current and past rentals 与被租借的拷贝是完全相同的组

vs.copies.currentRental -> union(vs.copies. pastRentals)

连接

因为类型模型包含可选功能，它导致一个对象与其它对象互相矛盾。这里有一些限制的例子，说明不允许的连接关系。

出租的录像带不能被预订保留

c : Copy

-- If a copy is rented then 如果一个拷贝已经出租

c.currentRental -> notEmpty implies

-- it is not on hold 就不能被保留

c.~heldCopy -> isEmpty

当前的租借不是一个过去的租借

c : Copy

-- If a copy has a current rental then 如果一个拷贝正在出租

c.currentRental -> notEmpty implies

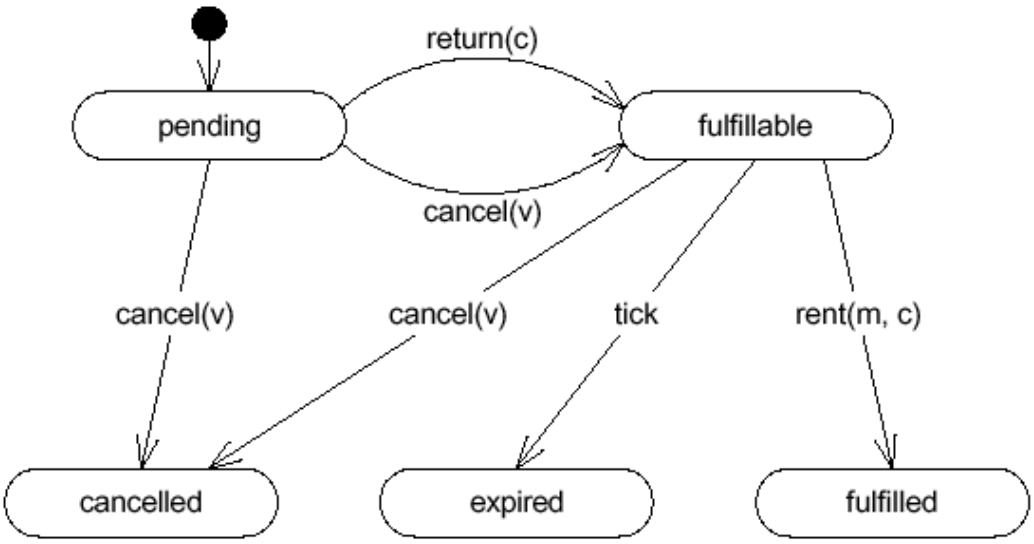
-- the copy's past rentals do not include this current rental 过去的记录不包括当前出租

not c.pastRentals -> includes(c.currentRental)

状态模型

状态模型非常强大，因为它们将注意力从关心所有类型整体的角度转移到一种类型的整个生命周期。这里有两个类型的状态模型。

预订



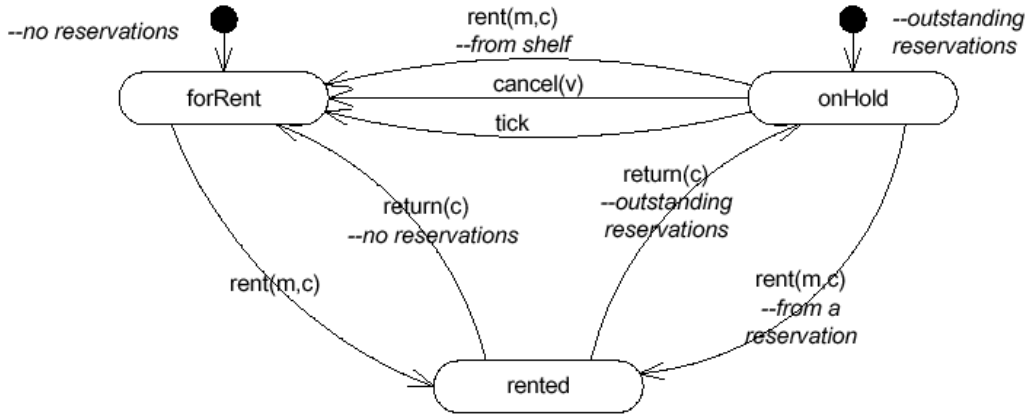
当一个会员预订一部影片时，创建一个预订对象。

预订创建后，就进入“等待”状态。“可满足”预订是指当一个拷贝被保留（当预订被满足，会有一个消息通知会员）。一个预订变为“已满足”是当该会员租借了预订影片的一个拷贝。一个预订过期是指被通知的会员在规定期限内没有租借预定影片。

当一个预订被取消（“等待”状态或“可满足”状态），预订状态变为“已取消”。随着这个动作，另一个预订的状态可能从“等待”状态变为“可满足”状态。

拷贝

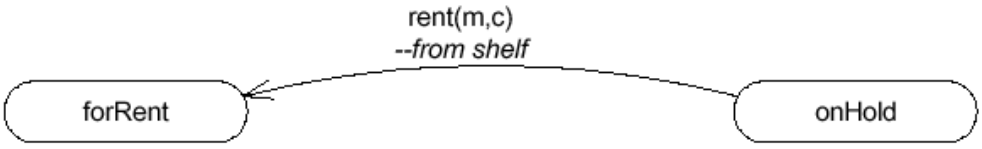
以下是拷贝类型的状态模型。随后用一个示例场景帮助解释。



操作拷贝的动作不属于租借主题的范围。如果没有特别的预订，拷贝对象在“待出租”状态创建，否则在“保留”状态创建。

示例场景

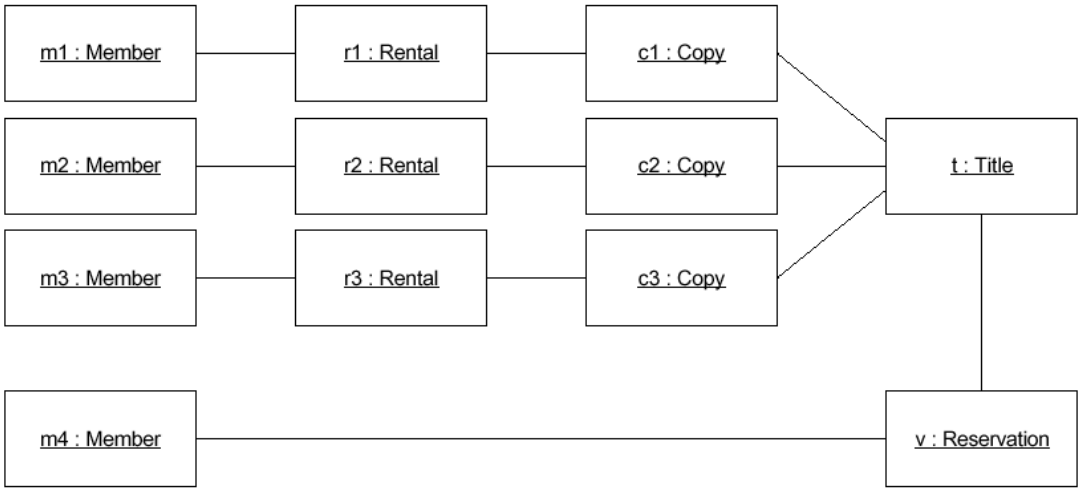
拷贝的状态模型表明，有三个操作触发状态从“保留”到“待出租”。如果会员取消预订，保留的拷贝就可以出租。如果时间超过期限，预订失效，保留的拷贝也可以出租。操作“出租”使拷贝从“保留”变为“待出租”，在后面对比进行解释。



以下场景解释这种转变是怎么发生的。场景由三部分组成：开始状态的描述，场景步骤，结果状态。

开始状态

在场景的开始状态，我们有 4 个会员对象（m1,m2,m3,m4），我们有一个片名对象(t)，有 3 个拷贝对象 c1,c2 和 c3 处于出租状态。会员 m4 有一个影片 t 的预订。以下是这种情况的一个快照。



场景步骤

会员 m1 还回拷贝 c1。

拷贝 c1 为 m4 的预订保留，预订 v 变为可满足。

会员 m2 还回拷贝 c2。

没有等待的预订，因此，拷贝 c2 被放回货架。

会员 m4 进入店中，看到 c2 在货架。

会员 m4 取下 c2，告诉店员租借该拷贝。

拷贝 c2 被 m4 借走。

为 m4 预留的拷贝 c1，被放到货架。

最终状态

会员 m4 租了拷贝 c2（而不是预留的拷贝 c1）

拷贝 c1 在货架上。

连接将场景与状态模型

当一个会员 m 租借拷贝 c：

如果

会员 m 获得了 Cr

并且

有另一个拷贝 Ch 为会员 m 保留

那么

拷贝 Cr 从状态“待出租”到“已出租”

拷贝 Ch 从状态“保留”到“待出租”

状态定义

对每一个带有状态模型的类型，我们必须能够区分类型不同属性的状态（实践中，状态定义的过程引导我们细化状态模型，特别是类似属性这样的细节）。

Reservation（预订）

v : Reservation

-- *A reservation is pending if 一个预订处于等待如果*

v.pending =

-- *there's no copy on hold for the reservation 没有为这个预订保留拷贝*

v.heldCopy -> isEmpty

-- *and it's not been fulfilled or expired or cancelled 没有被满足、失效或取消*

and v.whenFulfilled -> isEmpty

and v.whenExpired -> isEmpty

and v.whenCancelled -> isEmpty

and

-- a reservation is fulfillable if 一个预订处于满足状态如果

v.fulfillable =

-- there is a copy on hold for the reservation 有一个拷贝为这个预订预留

v.heldCopy -> notEmpty

and

-- a reservation is cancelled if 一个预订被取消如果

v.cancelled =

-- there is a timepoint at which it was cancelled 在某一时间被取消

v.whenCancelled -> notEmpty

and

-- a reservation has expired if 一个预订失效如果

v.expired =

-- there is a timepoint at which it expired 在有效期结束的时间

v.whenExpired -> notEmpty

and

-- a reservation is fulfilled if 一个预订被满足如果

v.fulfilled =

-- there is a fulfilling rental 一个出租完成

v.fulfillingRental -> notEmpty

Copy（拷贝）

c : Copy

-- A copy is available for rent if 一个拷贝可以出租如果

c.forRent =

-- it does not have a current rental没有被出租

c.currentRental -> isEmpty

-- and it is not on hold for a reservation且没有为预订保留

and c.~heldCopy -> isEmpty

and

-- a copy is on hold for a reservation if 一个拷贝为预订保留如果

c. onHold =

-- it does not have a current rental没有被出租

c.currentRental -> isEmpty

-- but it is on hold for a reservation但为一个预订保留

and c.~heldCopy -> notEmpty

and

-- a copy is out on rent if一个拷贝在出租如果

c.rented =

-- it has a current rental已经被出租

c.currentRental -> notEmpty

操作规范

这里是一些操作规范。每个操作有一个标记、初始状态和结束状态描述。

会员租借录像带拷贝

VideoStore :: rent(m : Member, c : Copy, d : Duration, p : Payment)

-- Member m rents copy c of a video for duration d, making a payment of p

--会员m租借拷贝c一段时间，支付p

pre:

-- The member and the copy are known to the video store

--录像带店知道会员与拷贝

self.members -> includes(m)

and self.inventory -> includes(c)

-- and the copy is available for anyone to rent or

--并且拷贝可以对所有人出租

and (c.forRent or

-- is on hold for member m (i.e., m has a fulfillable reservation for c)

(

c.onHold

and

c.~heldCopies -> select(fulfillable) -> exists(v | v.member = m)

))

-- and the payment is correct for the duration of the rental

```
--并且支付符合出租期限

and p.value = c.title.rateFor( d )

post:

-- There is a new rental object

--有一个新的出租对象

Rental.allInstances -> includes( r : Rental |

not Rental.allInstances@pre -> includes( r )

-- which is a rental by member m of copy c in return for payment p

--该对象说明拷贝c被会员m租借，支付费用p

and r.member = m

and r.copy = c

and r.payment = p

-- and is the current rental for copy c, but not a past rental

--拷贝c的当前租借，不是历史记录

and r = c.currentRental

and r.~pastRentals -> isEmpty

-- and which was made now for a duration of d

--期限为d

and r.made = self.calendarAndClock.now

and r.duration = d

-- and which has no 'whenCompleted' date and no fine
```

```
--没有到截止日期不需要罚金
and r.whenCompleted -> isEmpty
and r.fine -> isEmpty
-- and, if the copy had been on hold for the member,
--如果该拷贝是为该会员的预订保留，那么
-- against a reservation, then
and c.onHold@pre implies
let
-- the member's reservation gets fulfilled
--该预订被满足
fulfilledReservation =
c.~heldCopies -> select( v | v.member = m )
in
-- it's r that fulfils this reservation
--r满足了预订
fulfilledReservation.fulfillingRental = r
and
-- and the reservation's fulfilled timepoint is now
--满足预订的时间是当前时间
fulfilledReservation.whenFulfilled =
self.calendarAndClock.now
```

```
-- (which means there's a new 'oldestPending')
--（说明有一个新的最早预订）
-- and, if the member was holding a fulfillable reservation
--并且，如果该会员持有一个合乎要求的预订
and
let
fulfillableReservation =
m.reservations@pre ->
select( v | v.fulfillable and v.title = c.title )
in
( fulfillableReservation -> notEmpty
-- but the copy was not one of those on hold for the member
--但该拷贝不是为该会员保留，如果是从货架上取下的，那么
-- (i.e., was a copy from the shelves) then
and c.onShelf@pre ) implies
-- one of the copies on hold goes on the shelf
--保留的拷贝放回货架
fulfillableReservation -> heldCopies@pre -> exists( c2 |
c2.onShelf)
end
)
```

描述方式

以上描述包括以下段落：

-- and is the current rental for copy c, but not a past rental

and r = c.currentRental

and r.~pastRentals -> isEmpty

先前，我们定义了一个不变因素说明当前出租不是历史出租，如下：

当前的租借不是一个过去的租借

c : Copy

-- If a copy has a current rental then 如果一个拷贝正在出租

c.currentRental -> notEmpty implies

-- the copy's past rentals do not include this current rental 过去的记录不包括当前出租

not c.pastRentals -> includes(c.currentRental)

为了效果，我们重复说明了两次。建模者在决定采用什么方式时有两种极端的做法：

- 在不变因素中尽可能多说，在操作中尽可能少说，让不变因素完成剩下的工作。
- 没有不变因素，完全在操作中完成。

这里有一个折中方法：

在不变因素中描述大量有用的事情；操作中不重复，但证明包含在不变因素中（也就是说，没有状态可以引用错误的不变因素）。

会员预订

VideoStore :: reserve(m : Member, t : Title)

-- Member m makes a reservation for title t 会员 m 预订影片 t

pre:

```
-- Member m and title t are known to the video store录像带店知道会员与拷贝
self.members -> includes( m )
and self.catalog -> includes( t )
-- and the member doesn't already hold a pending reservation for that title
--会员没有预订这个影片
and not t.reservations -> exists( v : Reservation |
v.member = m -- the reservation is by member m
and v.pending ) -- and the reservation is pending
-- and there are no copies available for rent  没有其它拷贝可以出租
and not t.copies -> exists( c : Copy | c.forRent )
post:
-- There is a new reservation  有一个新预订
Reservation.allInstances -> exists( v : Reservation |
not Reservation.allInstances@pre -> includes( v )
-- and it is a reservation by member m for title t由会员m预订影片t
and v.member = m
and v.title = t
-- and it was made now  现在创建这个预订
and v.made = self.calendarAndClock.now
-- and it is in the pending state, so ...  预订处于等待状态，因此... ...
-- there are no copies on hold for the reservation  没有拷贝为这个预订保留
```

and v.heldCopies -> isEmpty

-- and it's not been fulfilled 没有被满足

and v.whenFulfilled -> isEmpty

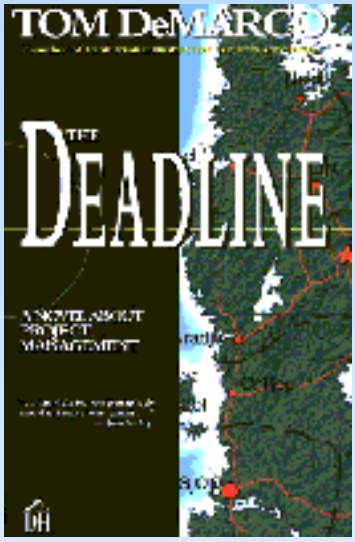
-- or cancelled 没有被取消

and v.whenCancelled -> isEmpty)

征稿

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

《最后期限》



《最后期限》

Tom Demarco

翻译：UMLChina 翻译组 透明

这是一本软件开发小说

汤普金斯在飞机的座位上翻了一个身，把她的毛衣抓到脸上，贪婪地呼吸着它散发出的淡淡芬芳。文案，他对自己说。他试图回忆当他这样说时卡布福斯的表情。当时他惊讶得下巴都快掉下来了。是的，的确如此。文案……吃惊的卡布福斯……房间里的叹息声……汤普金斯大步走出教室……莱克莎重复那个词……汤普金斯重复那个词……两人微张的嘴唇触到了一起。再次重播。"文案。"他说道，转身，看着莱克莎，她微张的嘴唇，他……倒带，再次重播……

....

"我不想兜圈子，"汤普金斯看着面前的简报说，"实际上你们有一千五百名资格相当老的软件工程师。"

莱克莎点点头："这是最近的数字。他们都会在你的手下工作。"

"而且据你所说，他们都很优秀。"

"他们都通过了摩罗维亚软件工程学院的 CMM 2 级以上的认证。"

中文译本即将发行！

Reactor 模式——同步事件复用和处理调度的对象行为模式

Douglas C. Schmidt 著，Tonny Tam 译

查看评论

1 意图

Reactor 设计模式处理来自一个或多个客户并发递送过来的服务请求。应用程序中的每个服务可能包含几个分别由不同的事件处理器所展现的方法(method)，以响应特定于服务的请求。其中有一个用作管理被登记事件处理器的调度器(Initiation Dispatcher，译者注：之所以有 initiation 这个单词，是相对于 Proactor 模式中的充分调度器(Completion Dispatcher)而言的，本身并没有特别的意思。下一篇译文将介绍 Proactor 模式)执行事件处理器的调度，由负责管理已注册事件处理器(Event Handler)的调度器(Initiation Dispatcher)执行事件处理器的调度。服务请求的多路复用则由同步事件复用器(Synchronous Event Demultiplexer)来完成。

2 别名

调度器(Dispatcher),通知者(Notifier)

3 例子

为图解 Reactor 模式，考虑一个如图 1 所示的提供分布式日志服务的事件驱动式服务器程序。客户程序在分布的环境中使用日志服务记录它们的状态信息。 这些状态信息通常包括错误通知,除错跟踪,还有执行效率报表等。日志记录被送到中央的日志服务器，并被日志服务器记录到各种输出设备上，例如控制台，打印机，文件，或者是数据库。

如图 1 中所示的日志服务器处理来自客户端应用发送过来的日志记录和连接请求。在多个句柄(handle)上的日志记录和连接请求将会并发地到达服务器。句柄(handle，在例子中就是套接字 socket handle)在这里标识了由操作系统维系的网络通讯资源。

日志服务器和客户端应用间使用面向连接的协议(connection-oriented protocol)进行通讯，比如 TCP(传输控制协议)[1]。要求记录自己的日志信息的客户应用必须首先发送一个连接请求到服务器。服务器使用在大家共知的地址(和端口)上实行监听的句柄工厂(handle factory)以等待这些连接请求。当一个连接请求到达时,句柄工厂通过创建一个象征连结端点的句柄来建立服务器和客户之间的连接。新建的连结句柄被返回给服务器，服务器即可籍此句柄等待客户的服务请求。一旦客户程序连接成功,各客户程序就可以通过其句柄并行地发送日志记录到服务器，而服务器则可在这些已连接句柄上接收到日志记录。

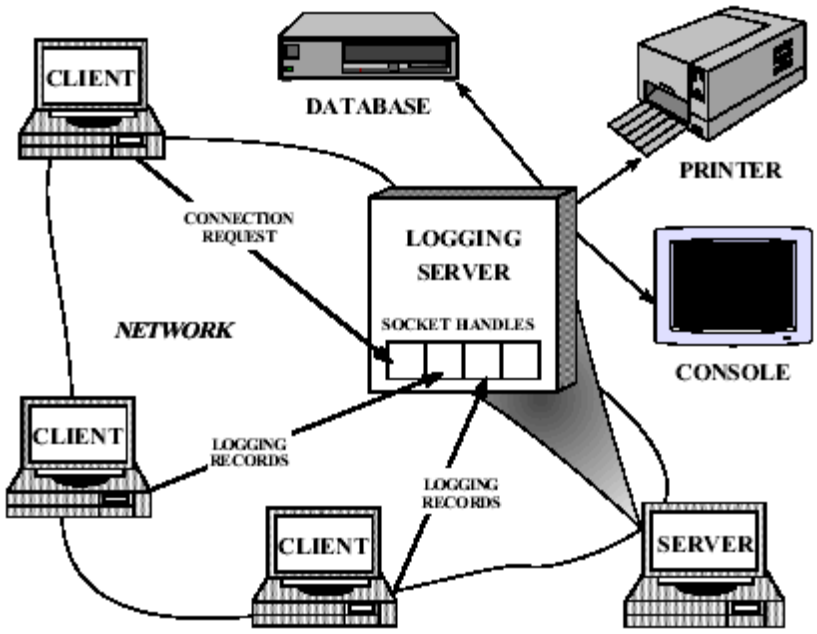


Figure 1: Distributed Logging Service

也许最直接的方法就是用能并行处理多个客户的多线程(multi thread)开发一个并发日志服务器,如图 2 所示. 这个方法同步地接受网络连接，然后为每一个连接创建(spawn)一个线程("thread-per-connection")来处理客户日志记录。

然而，在服务器程序中使用多线程来实现记录日志，将难以解决下列问题:

- 运行效率: 多线程的处理方法导致上下文切换(context switching)，同步(Synchronization)和数据移动[2]这些操作效率很差。
- 简单明了的编程: 多线程编程需要复杂的并行控制策略。
- 可移植性: 并非所有的操作系统中都支持多线程，这带来了程序可移植性方面的麻烦。

由于这些缺点，对于开发一个日志服务器，多线程常常不是最有效的，至少也是一个复杂的解决方法。

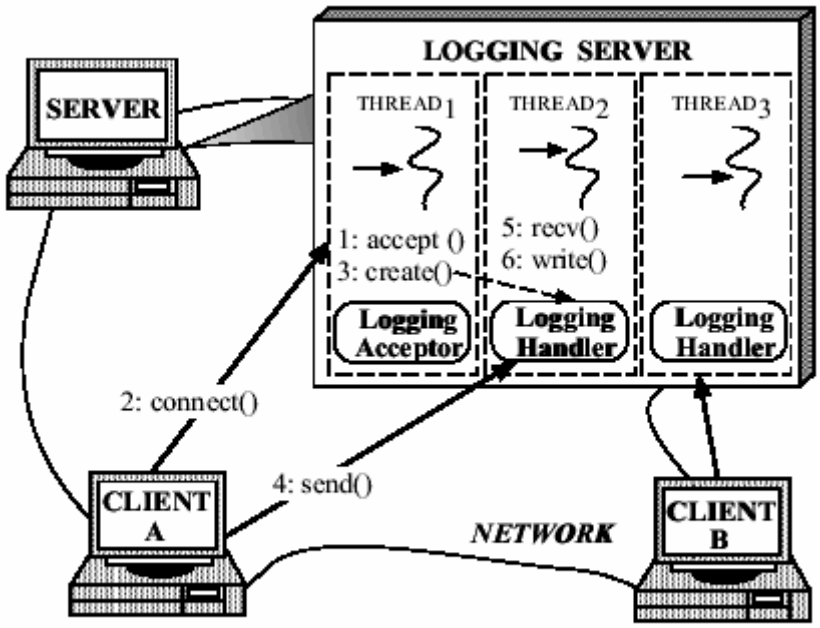


Figure 2: Multi-threaded Logging Server

4 上下文

一个分布式环境中并发地接收来自一个或者多个客户程序事件的服务器应用程序。

5 问题

分布式环境中的服务器应用程序必须处理多个向它发出服务请求的客户。然而在调用相应服务函数之前，服务器程序必须多路复用和将每个到达的请求分派到其响应服务提供者(service provider)上。要开发一个有效的多路复用(demultiplexing)和分派(dispatching)客户请求这样一个架构，需要有效地解决下面几个问题：

- 可用性: 服务器必须可靠地处理进来的请求事件，即使它正在等待其他的请求到达。特别地，服务器不能无限期地阻塞在处理任何单个事件源而明显地耽误了来自于另外客户程序事件源的响应时间。
- 效率: 服务器必须尽量减少响应时间,最大化吞吐量，还有避免不必要地使用 CPU(s)。
- 易于编程：服务器的设计应能简化使用各种合适的并行策略。

- 适应性: 结合新的服务或者增强服务，比如改变日志格式或者加入服务器端的高速缓存。应只导致在现有代码上最小的修改和维护费用。举例来说,实现新的应用服务不应需要修改通用的多路复用和分派架构。
- 可移植性: 把服务器应用程序移植到新的操作系统平台不需很费劲。

6 解决方案

结合同步事件多路复用和相应的事件处理器所处理的事件的分派。此外, 从通用的事件多路复用和分派机制中把特定于应用的调度及其服务的实现解耦(decouple)出来。

对于每个应用程序提供的服务，引入一个独立的事件处理器(Event Handler)以处理确定类型的事件。所有的事件处理器实现相同的接口(interface)。这些事件处理器在调度器(Initiation Dispatcher) 里注册，调度器使用同步事件复用器(Synchronous Event Demultiplexer)等待事件的发生。当事件发生时，同步事件复用器通知调度器，以同步地回调(call back)与该事件相关联的事件处理器。事件处理器将事件递送到请求服务的函数实现上去。

7 模式的构成

在 Reactor 模式中主要的参与者如下:

- **句柄（Handles，在日志服务器里是套接字 socket handle）**

标识操作系统管理的资源，这些资源通常包括网络连接，打开的文件，定时器，同步对象等等。在日志服务器中的句柄标识套接集合点(socket endpoint)，而同步事件复用器则在其上等待事件的发生。就日志服务器而言，它关心两种类型的事件－connection 事件和 read 事件，分别表示到来的客户连接和日志数据。日志服务器为每个客户程序维护着各自独立的连接，每一个连接在服务器中都以一个套接句柄 (socket handle，也就是套接字)表示。

- **同步事件复用器(Synchronous Event Demultiplexer)**

在一组句柄上阻塞等待事件的发生。一旦能够在一个句柄上启动非阻塞的操作，它就会返回。事件多路复用器通常使用 select 函数[1]，这是 UNIX 和 Win32 操作系统上提供事件复用功能的系统调用。Select 系统调用标示应用程序进程中哪个句柄能够被同步调用而不阻塞应用进程(application process)。

- **调度器(Initiation Dispatcher)**

为注册，删除，和分派事件处理器而定义的一个接口。最终为同步事件复用器在等待新的事件发生时使用。当监测到一个新的事件，多路复用器通知调度器回调(call back)特定于应用程序的事件处理器。通常来说，这些事件包括接受连接事件、数据输入输出事件和定时器事件。

- **事件处理器(Event Handler)**

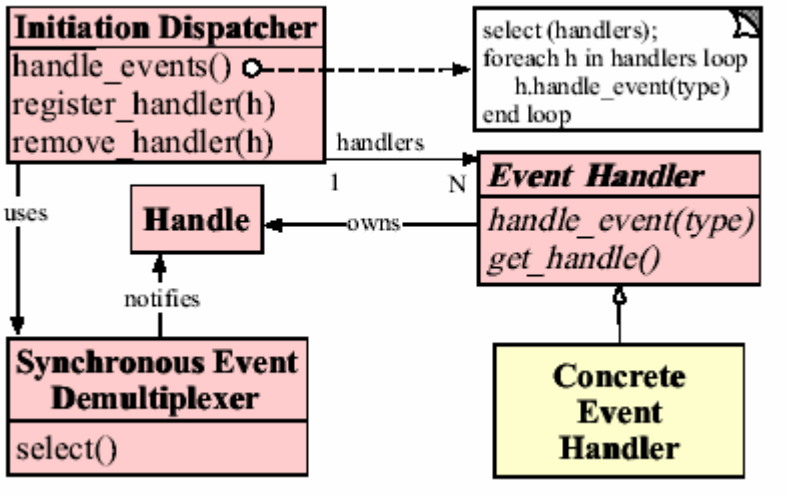
特指包含钩子函数(hook method[3])的接口(在 C++的实现里就是一个抽象类)，抽象呈现了于特定服务之事件的分派操作。这些钩子函数必须在特定于应用的服务里实现。

- **具体事件处理器(Concrete Event Handler)**

实现钩子函数，使它就象以特定于应用的方式来处理事件。应用程序向调度器注册具体事件处理器以处理确定类型的事件。当这些事件到达时,调度器回调(call back)适当的具体事件处理器中的钩子函数。

就日志服务器而言，有两种具体事件处理器：日志处理器(Logging Handler)和日志接受器(Logging Acceptor)。日志处理器负责接收并处理日志记录。日志接受器创建并连接日志处理器以处理后面从客户程序过来的日志记录。

Reactor 模式中参与者之间的结构如下面的 OMT 类框图所示：



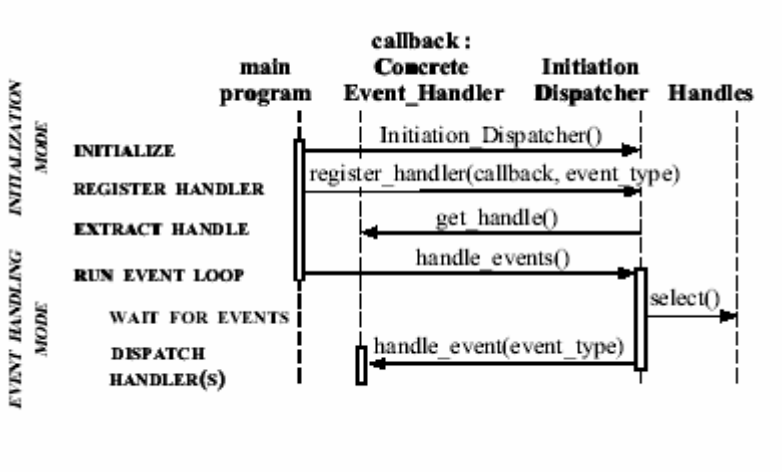
8 动态

8.1 互相协作概要

下列各项互动过程发生在 Reactor 模式里:

- 应用程序向调度器注册一个关注某类事件的具体事件处理器，表明在相关句柄上发生的该类事件，要求调度器通知该事件处理器。
- 调度器要求每个事件处理器返回内部句柄。
- 所有的事件处理器被注册后,应用程序调用调度器的 handle_events 方法以启动调度器的事件循环(event loop)。 在此，调度器收集每个被注册的事件处理器里的句柄，并使用同步事件多路复用器等待在这些句柄上的事件的发生。举例来说,TCP 协议层使用 select 同步复用操作在已连接的套接字句柄(socket handles)上等待客户程序日志记录事件的到达。
- 当和某句柄相应的事件源(event soruce)的状态变为"就绪"("ready")，例如，TCP 套接字"读就绪"("ready for reading")。同步事件复用器通知调度器。
- 调度器触发事件处理器内的钩子函数以响应就绪句柄上的事件。当事件发生时，调度器使用被事件源激活的句柄作为关键字以定位事件处理器，并调度相应事件处理器中的钩子函数。
- 调度器回调事件处理器中的钩子方法(handle_event)，执行特定应用的功能。所发生的事件，其类型作为参数传递给函数并让其在内部中使用以实现更多的特定于服务的多路复用和分派任务。第 9.4 节描述了一个可供替代的分派方案。

下面是应用程序代码与模式中各参与者之间互相协作的框图:

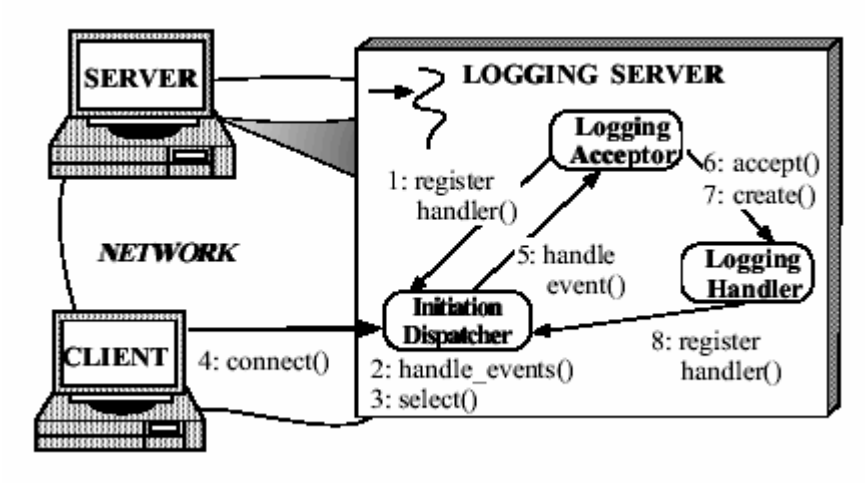


8.2 协作情景

日志服务器中的 Reactor 模式中各参与者的互相协作可被图解为两个情景。这些情景显示了日志服务器是如何设计成处理来自多个客户的连接请求和处理日志数据请求。

8.2.1 客户端连接到日志服务器

第一个情景图解显示当一个客户连接到日志服务器时所发生的各个步骤。

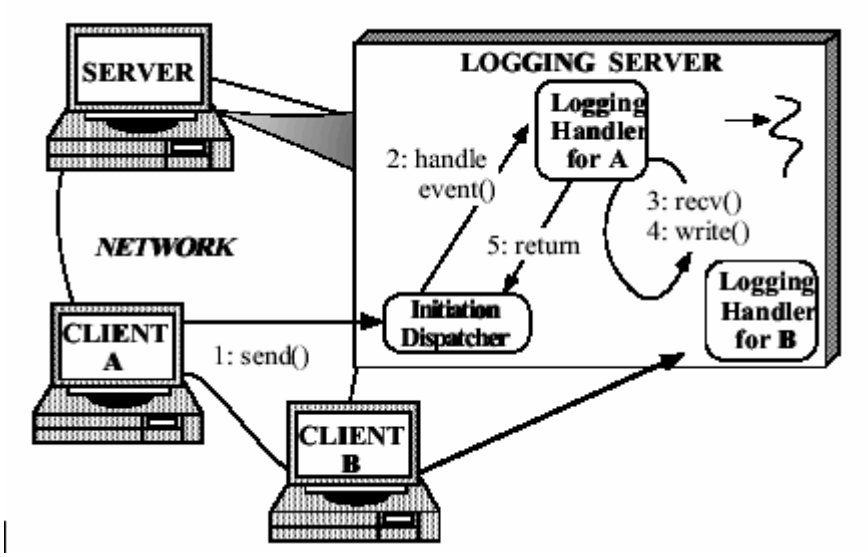


交互的步骤次序总结如下：

1. 日志服务器(1)向调度器注册日志接受器(Logging Acceptor)，以响应连接请求；
2. 日志服务器调用调度器的 handle_events 方法(2)；
3. 调度器调用同步复用的系统调用 select(3)，以等待连接请求或者日志数据的到达；
4. 某客户程序(4)连接日志服务器；
5. 调度器(5)通知日志接受器(Logging Acceptor)有一个新的连接请求。
6. 日志接受器(Logging Acceptor)接收(6)这个新的连接；
7. 日志接受器创建(7)一个日志处理器(Logging Handler)以服务这个新的客户；
8. 日志处理器(Logging Handler)向调度器注册自己以使调度器当套接字状态变为”读就绪”(“ready for reading”)的时候通知自己。

8.2.2 客户向日志服务器发送日志记录

第二个情景图解显示服务器处理日志记录时的步骤次序。



交互步骤先后次序总结如下：

1. 客户程序发送(1)一个日志记录；
2. 当客户日志记录被写入套接字，调度器通知(2)相关的日志处理器(Logging Hander)；
3. 日志记录以非阻塞的方式被接收下来(第 2 和第 3 步反复执行，直到日志记录完全接收下来)；
4. 日志处理器(Logging Handler)处理日志记录并写(4)到标准输出；
5. 日志处理器(Logging Handler)返回，并把控制交还给调度器的事件循环(event loop)里。

9 实现

这一节描述如何使用 C++来实现 Reactor 模式。下面描述的实现方法源自 ACE communication software framework[2]中提供的可复用部件。

9.1 选择同步事件复用器机制

调度器使用同步事件复用器机制以同步地等待一个或者多个事件的发生。通常的实现是使用类似 select 这种操作系统 I/O 复用系统调用。select 调用标识哪个句柄就绪，可以执行特定于应用服务的居于处理器中的 I/O 操作，

而不是阻塞操作系统进程。通常来说，同步事件复用器基于现行操作系统机制。

9.2 开发调度器(Initiation Dispatcher)

以下几步是开发一个调度器(Initiation Dispatcher)必须的实现步骤：

实现事件处理器表(event handler table)：调度器维护着一个表，来存放具体事件处理器。因此，调度器必须实现运行时注册和删除处理器的方法。这个表可以有很多的方法实现，比如，使用哈希表，线性搜索表，或者如果句柄表现为相近范围的小整型值就直接索引。

实现一个事件循环(event loop)入口点(entry point)：调度器的事件循环入口点由 handle_events

函数提供。这个函数籍由同步事件复用器提供的句柄多路选择功能控制句柄，一如执行着事件处理器的分派工作。常有这种情况，整个应用程序的主事件循环由这个入口点(entry point)控制。

当事件发生，调度器从同步事件复用调用(select 调用)返回，并为每个已就绪的句柄分派事件处理器里的钩子函数。钩子函数运行用户编写的代码，并在完成后把控制返回给调度器。

以下的 C++类阐明了调度器公共接口的核心方法：

```
enum Event_Type
// = TITLE
// Initiation_Dispatcher.的事件句柄类型
//
// = DESCRIPTION
// 这些值都是 2 的 n 次方，能有效地进行 or 操作作为联合值。
{
ACCEPT_EVENT = 01,
READ_EVENT = 02,
WRITE_EVENT = 04,
```

```
TIMEOUT_EVENT = 010,
SIGNAL_EVENT = 020,
CLOSE_EVENT = 040
};

class Initiation_Dispatcher
// = TITLE
// 用作响应客户请求的 I/O 多路复用和分派事件管理器
{
public:
    // 注册特定事件类型(例如 READ_EVENT,ACCEPT_EVENT 等等)的事件处理器
    int register_handler (Event_Handler *eh,
        Event_Type et);
        // 删除一个特定事件类型的事件处理器
    int remove_handler (Event_Handler *eh,
    Event_Type et);
    // 互动的事件循环(event loop)的入口点。
    int handle_events (Time_Value *timeout = 0);
};
```

实现必要的同步机制：如果 Reactor 模式使用在只有一个控制线程的应用程序中，那么它有可能排除所有的同步。这种情况下，调度器在应用程序进程中序列化事件处理器钩子函数 handle_event，使之一个一个执行。

然而，调度器也可以作为一个多线程的应用程序中的中央事件分派器。在这种情况下，当修改或者使用处于调度器中的临界区共享变量(比如维护事件处理器的表)时，必须序列化其访问以防止竞争条件的产生。一个比较常用的技术是使用类似信号量或互斥量这样的互斥机制避免出现数据的竞争条件。

为避免自死锁(self-deadlock)，互斥机制可以选择递归锁(recursive lock)[4]来实现。贯穿事件处理器钩子函数运行过程，只要线程持有递归锁就能够有效地防止死锁。拥有锁的线程能够多次获得递归锁而不会导致阻塞线程。递归锁的这种属性在 Reactor 的 handle_events 方法回调特定应用的事件处理器函数过程中非常重要。应用程序的钩子函数代码随后将会通过其 register_handler 和 remove_handler 方法再次进入调度器。

9.3 决定分派目标的类型

作为调度器分派逻辑的目标，关联于一个句柄的事件处理器有两种类型。可以选择以下两种可替代方案中的一种或者全部来实现 Reactor 模式：

事件处理器对象(Event Handler Objects)：一个通常的做法是关联于句柄的事件处理器作为事件处理器对象来使用。举例来说，如第 7 节所描述的 Reactor 实现方案，Reactor 向调度器注册了一个事件处理器子类的对象。这样，使用处理器子类对象作为分派的目标，便于重用和扩展现有的部件(component)。另外，对象结合了服务的方法和属性，并封装在一个单一的部件中，提高了封装性。

事件处理器函数(Event Handler Functions):另一个方法是关联于句柄的事件处理器向调度器注册事件处理器的处理函数。使用函数作为分派的目标，有利于只是注册一个回调函数而不必定义一个继承自 Event Handler 的新类。

Adapter 模式[5]在这里被应用为同时支持对象和函数。例如，一个 adapter 可以被定义为使用一个持有事件处理函数指针的事件处理器对象。当事件处理器 adapter 对象上的 handle_event 方法被调用，它将会自动把调用转递给它持有的事件处理函数。

9.4 定义一个事件处理接口

假设我们使用事件处理器对象而不是函数，下一步是定义一个事件处理器的接口。这里有两个途径：

单方法接口：第 7 节中的 OMT 图表呈现了一个含有名为 handle_event 的单方法的事件处理基本类。被调度器用于分派事件。在这里，所发生事件的类型，作为参数传递到这个方法上。

以下是 C++抽象基类定义这个单方法接口：

```
class Event_Handler
// = TITLE
// 作为调度器分派目标的抽象基类。
{
    public:
        //调度器为了处理事件而回调的钩子方法
        virtual int handle_event(Event_Type et) = 0;

        //返回句柄的钩子方法
        virtual Handle get_handle(void) const = 0;

};
```

单方法接口的优点是无须改变接口的情况下就能增加新的事件类型。然而，这种方法鼓励在子类的 handle_event 方法中使用 switch 语句，这限制了它的扩展性。

多方法接口：另一个实现事件处理器接口的方法是为每一种事件类型分别定义虚拟钩子方法（例如 handle_input，handle_output，或者 handle_timeout）。

下面的 C++抽象基类定义了多方法接口：

```
class Event_Handler
{
public:
    // 由调度器为处理特定事件而回调的各个钩子方法。
        virtual int handle_accept (void) = 0;
```

```
        virtual int handle_input (void) = 0;

    virtual int handle_output (void) = 0;

    virtual int handle_timeout (void) = 0;

    virtual int handle_close (void) = 0;


    //返回句柄的钩子方法

    virtual Handle get_handle (void) const = 0;

};
```

多方法接口的好处是易于有选择地重载(override)基类中的方法，并且避免进一步的多路复用，例如，通过 switch 或者 if 语句实现多路复用。然而，这要求框架开发者尽可能地提前预见在未来可能需要使用的事件处理器方法集。举例来说，在 Event_Handler 接口里众多的 handle_*方法要符合通过 UNIX 系统的 select 系统调用机制可用的 I/O 事件种类。然而，接口里的这些事件处理类型并不足以覆盖通过 Win32 的 WaitForMultipleObjects 系统调用机制而可用的事件处理类型[6]。

以上描述所定义事件处理接口是在[3]中描述的 hook method 模式和[7]中描述 Factory Callback 模式的一个例子。这些模式的意图是为了提供一个定义良好的钩子，以被专门的应用程序和底层分派代码回调(call back)。

9.5 确定应用程序中的调度器的数目

许多应用只用一个 Reactor 模式实例来构建。这种情况下，调度器被实现为一个 Singleton[5]。这种设计对于应用需要在单一位置上集中管理事件的复用和调度很有用。

然而，一些操作系统限制了只拥有单一控制线程的应用中可等待的句柄数目。举例来说，Win32 允许在一个线程中的 select 和 WaitForMultipleObjects 调用等待不超过 64 个句柄。这种情况下，可能需要创建多个线程，每个线程中有一个 Reactor 模式的实例。

需要注意的是，多个事件处理器只会在一个 Reactor 模式实例得到序列化。因此，在多线程中多个事件处理器可以并行运行。这样使如果在不同的线程中的事件处理器访问共享资源令使用额外的同步机制成为必要。

9.6 实现具体事件处理器

典型的情况是，具体事件处理器是由服务器应用程序开发者创建的，以响应特定事件发生时执行特定服务。开发者必须确定当调度器调用相应的钩子函数时什么需要处理。

以下的代码实现了在第 3 节描述的日志服务器的具体事件处理器。这些处理器提供了被动连接建立(Logging Acceptor)和接收数据(Logging Handler)。

日志接受器(Logging Acceptor)类：

这是 Acceptor-Connector 模式[8]中的 Acceptor 组件例子。该模式对服务的初始化任务和初始化之后需要执行的任务进行解耦(decouple)。使应用程序中的特定部分，比如日志处理器(Logging Handler)，完全独立于建立连接的机制。

日志接受器被动接受来自客户程序的连接请求，并创建特定于该客户的日志处理器(Logging Handler)对象，以用于接收和处理来自该客户的日志记录。日志接收器类的主要方法和成员数据如下所示：

```
class Logging_Acceptor : public Event_Handler
// = TITLE
// 处理客户连接请求。
{
public:
    // 初始化 acceptor_并且向调度器注册。
    Logging_Acceptor (const INET_Addr &addr);

    // 接收一个新的套接口流(SOCK_Stream)对象连接并创建一个
    //日志处理器(Logging_handler)对象来处理日志记录。
    //  这是个工厂方法(Factory method)。
    virtual void handle_event (Event_Type et);
```

```
        // 取回 I/O 句柄（日志接受器被注册后，由调度器调用）。
        virtual HANDLE get_handle (void) const
        {
            return acceptor_.get_handle ();
        }

    private:
        // 套接口接受器，接收客户连接的套接口工厂(socket factory)。
        SOCK_Acceptor acceptor_;
};
```

日志接受器类继承自 Event_Handler 基类。这使应用程序能够向调度器注册日志接受器。

日志接受器还包含了一个套接口接收器(SOCK Acceptor)实例。这是个具体工厂(concrete factory)，使 Logging Acceptor 以监听某端口以被动方式接收客户连接请求。当一个来自客户程序的请求到达时，SOCK Acceptor 接受连接并创建 SOCK Stream(套接口流)对象。然后，SOCK Stream 对象被用于在客户和日志服务器之间可靠地传输数据。

用于实现日志服务器的套接口接收器(SOCK Acceptor)类和套接口流（SOCK Stream）类是 ACE toolkit[9]中 C++套接口包裹库的一部分。这些套接口包裹库封装了套接口的 SOCK Stream(套接口流)语义，并使之成为面向对象的，可移植的和类型安全的接口。在 Internet 域（译者注：UNIX 中的套接口分 Internet 域和 UNIX 域）中，SOCK Stream 套接口使用 TCP 协议。

日志接受器的构造函数向实现成 Singleton[5]的调度器以 ACCEPT 事件这种事件类型为参数注册自己，如下所示：

```
Logging_Acceptor::Logging_Acceptor(const INET_Addr &addr): acceptor_ (addr)
{
// 向调度器注册自己，在调度器的register_handler方法里，调用
// Logging_Acceptor::get_handler方法即可取得该
// 日志接受器的套接口。
Initiation_Dispatcher::instance ()->register_handler (this, ACCEPT_EVENT);
}
```

然后，无论何时客户的连接一到达，调度器就会回调日志接受器的 handle_event 方法，如下所示：

```
void Logging_Acceptor::handle_event (Event_Type et)
{
// 断言事件类型是 ACCEPT_EVENT
assert (et == ACCEPT_EVENT);

SOCK_Stream new_connection;

// 接收连接
acceptor_.accept (new_connection);
// 创建一个新的 Logging Handler(日志处理器)
Logging_Handler *handler = new Logging_Handler (new_connection);
}
```

handle_event 方法调用套接口接受器(acceptor_)的 accept 方法，以被动的方式建立套接口流对象（new_connection）。一旦为新的客户建立套接口流对象，一个新的日志处理器(Logging Handler)被动态分配以处理

日志请求。如下所示，日志处理器(Logging Handler)向调度器注册自己，这就多路复用了所有的相关于该客户的日志记录。

日志处理器（Logging Handler）类:

Logging Handler 使用如下所示的日志处理器类，以接收从客户应用发送过来的日志记录：

```
class Logging_Handler : public Event_Handler
// = TITLE
// 接收和处理来自客户应用的日志记录
{
public:
    // 初始化客户套接口流
    Logging_Handler (SOCK_Stream &cs);

    // 钩子方法，处理接收下来的来自客户的日志记录
    virtual void handle_event (Event_Type et);

    // 获得 I/O 句柄(日志处理器被注册后，由调度器回调)。
    virtual HANDLE get_handle (void) const
    {
        return peer_stream_.get_handle ();
    }
```

```
private:
    // 用于从客户端取得日志记录的套接口流。
    SOCK_Stream peer_stream_;
};
```

日志处理器继承自 Event Handler 基类，使得它能够在调度器上注册。构造函数如下所示：

```
Logging_Handler::Logging_Handler(SOCK_Stream &cs) : peer_stream_ (cs)
{
// 向调度器注册自己，并以 READ_EVENT 这种事件类型为参数(当在该句柄上发生 READ 事件，调度器就会通知相关的日志处理器实例)。
Initiation_Dispatcher::instance ()->register_handler (this, READ_EVENT);
}
```

一旦它被创建，日志处理器(Logging Handler)向调度器注册自己，并以 READ_EVENT 这种事件类型为参数。然后，当一个日志记录到达，调度器将自动调用日志处理器的 handle_event 方法，如下所示：

```
void Logging_Handler::handle_event (Event_Type et)
{
if (et == READ_EVENT) {
    Log_Record log_record;
    peer_stream_.recv ((void *) log_record, sizeof log_record);
    // 把日志记录写到标准输出。
    log_record.write (STDOUT);
}else if (et == CLOSE_EVENT) {
    peer_stream_.close ();
```

```
        delete (void *) this;
    }
    }
```

当在一个套接字上发生 READ 事件。调度器回调日志处理器(Logging Handler)的 handle_event 方法，这个成员函数接收、处理并且把日志记录写到标准输出(STDOUT)。同样地，当客户关闭连接，调度器发出 CLOSE 事件，告诉日志处理器要关闭套接口流(SOCK Stream)并删除自己。

9.7 实现应用程序服务器

日志服务器包含一个 main 函数。

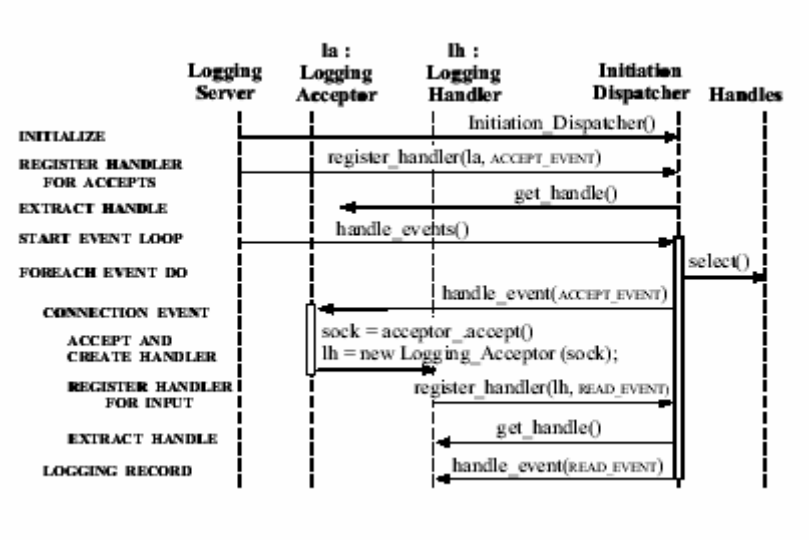
日志服务器的 main 函数:main 函数实现了单线程的，并行处理的日志服务器。它等待在调度器的时间循环上。当来自客户的请求到达，调度器调用适当的具体事件处理器的钩子函数，接受连接并且接收和处理日志记录。日志服务器的主入口定义如下：

```
// 服务器的端口
const u_short PORT = 10000;
int main (void)
{
// 日志服务器的端口
INET_Addr server_addr (PORT);

// 初始化服务器端点，并向调度器注册自己。
Logging_Acceptor la (server_addr);

// 主事件循环(event loop)，回调适当的具体事件处理器
```

```
// 钩子函数，处理连接请求和来自客户的日志记录。
for (;;)
    Initiation_Dispatcher::instance ()->handle_events ();

/* NOTREACHED */
return 0;
}
```

主函数创建日志接受器(Logging Acceptor)对象，日志接受器的构造函数以服务器所在地址和监听的端口初始化该对象。然后进入主事件循环(event loop)里。随后，实现为 Singleton 的调度器使用 I/O 复用系统调用 select 同步地等待来自客户端的连接请求并处理来自客户的日志记录。

下面的交互图表阐明了日志服务器中各参与者的交互协作情况：



一旦调度器对象初始化完成，它就成为了日志服务器中的主要控制流程焦点。所有随后而来的行动都是由那些向调度器注册的，并由之控制的事件处理器(日志接受器(Logging Acceptor)和日志处理器(Logging Handler))中的钩子函数触发。

当一个连接请求到达，调度器回调(call back)日志接受器(Logging Acceptor)，接受一个网络连接并创建一个日志处理器(Logging Handler)。这个日志处理器向分派调度器注册自己，并以参数的形式告诉调度器自己关注 READ 事件。这样，当客户应用发送日志记录，在日志服务器端，相关句柄(套接字 socket handle)读就绪(ready for reading)。于是调度器回调(call back)日志处理器(Logging Handler)的钩子函数以接收和处理从客户端传送过来的日志记录。

10 已知应用

Reactor 模式使用在很多面向对象框架里，包括以下所列的：

InterViews: 分布式 InterViews[10] window system 实现了 Reactor 模式，并称之为分派器(Dispatcher)。InterViews 分派器(Dispatcher)用于定义应用程序的主事件循环(main event loop)，以管理一个或者多个物理上的 GUI 显示。

ACE Framework: ACE Framework 使用 Reactor 模式来进行中心 I/O 事件复用和分派。

Reactor 模式使用在很多的商业项目上，包括：

CORBA ORBs： Reactor 模式被用在很多单线程的 CORBA[12]实现里，在 CORBA 的 ORB 核心层里(比如 VisiBroker，Orbix，和 TAO[13])，Reactor 模式被用于多路复用和向在 CORBA 中实现的服务(servant)分派的 ORB 请求。

Ericsson EOS Call Center Manager System(爱立信呼叫中心管理系统): 该系统使用 Reactor 模式通过 Event Server[14]管理从 PBX（分组交换机）到超级管理员之间的事件路由。

Project Spectrum:医用光谱工程中高速的图像传输子系统里使用了 Reactor 模式。

11 总结

11.1 优点

Reactor 模式有以下优点：

分离复杂的耦合：Reactor 模式对独立于应用的 I/O 事件复用与分派机制和特定于具体应用程序的钩子函数实现解耦(decouple)。这使独立于应用的 I/O 复用和分派机制成为可重用部件，它知道如何复用事件和分派适当的事件处理器(Event Handler)所定义的钩子函数。相比之下，钩子函数里的特定于应用的函数，则必须知道如何执行精确类型的服务。

增强了事件驱动应用程序的模块化，可重用性和可配置性：Reactor 模式把应用程序要实现的功能都分别隔离到特定的类(具体的事件处理器类)中。举例来说，在日志服务器中有两个类：一个用于建立连接而另一个用于接收和处理日志记录。这种解耦能够使建立连接的类可重用于不同类型的面向连接的服务(比如文件传输，远程登录和视频流服务)。所以，修改或者扩展日志服务器的功能只会影响到日志处理类(Logging Handler Class)。

增强了应用的可移植性：调度器由于独立于操作系统的系统调用而可被重用。这些系统调用监测和报告已发生的一个或多个事件可能发生在多个事件源上。通常，事件源包括 I/O 句柄，定时器，同步对象。在 UNIX 系统，事件复用的系统调用有 select 和 poll[1]。Win32 平台上则是 WaitForMultipleObjects。

提供了并行控制：Reactor 模式提供了进程或线程在事件复用和分派层面上的事件处理器中钩子函数调用的序列化。调度器层面上的序列化常常能排除应用程序进程内更为复杂的同步或者锁定。

11.2 限制

Reactor 模式有以下限制。

受限制的适用性：Reactor 模式只能有效地使用操作系统所支持的句柄。一个可能的做法是在调度器中使用多线程模拟 Reactor 模式的语义。例如每个句柄用同一个线程。无论何时，当事件发生，相关联的线程将读到事件并把它压入队列(queue)从而由调度器来处理。然而，这种设计因序列化所有的事件处理器而显得效率低下，从而，增强使同步和上下文切换的费用高于增强的并行能力。

非抢占式：在单线程的应用程序的进程里，事件处理器(Event Handler)执行时并非使用抢占方式。这意味着对于事件处理器而言，由于会阻塞整个应用进程从而阻止了对其他句柄上的客户的响应，而不能在单独的某个句柄上执行阻塞的 I/O 操作。因此，一些持续时间较长的操作，例如传输数 MB 的医学光谱图片[15]，使用 Active Object 模式[17]更为有效。一个 Active Object 能够在调度器的主事件循环(main event loop)里并行地使用多线程或者多进程完成其作业。

难以除错：使用 Reactor 模式写的程序因为在特定于应用的回调函数和框架内部结构之间反向的控制流程而难以除错。这增加了在除错器(debugger)里通过框架的运行时行为来单步执行的难度，因为开发者并不知道或已处在框架的代码中。这好比遇到企图对使用 LEX 或者 YACC 这样的语法分析器产生的编译器代码进行除错的情况。在这种程序中，当控制线程在用户的例程里，这时的除错是直观易懂的，一旦控制线程返回到 LEX 或 YACC 这些工具产生的确定有穷自动机(DFA)骨架，将很难沿着程序的逻辑跟踪下去。

12 相关模式

Reactor 模式和 Observer 模式(观察者模式)[5]在单个主体改变时需要通知所有有关的对象这一点上相似。在 Reactor 模式中，当受事件处理器关注的事件在事件源上发生，该处理器将会被通知。Reactor 模式通常用在与多个事件源相关联的事件处理器上进行事件复用，然而 Observer 模式则常用于单事件源事件。

Reactor 模式与 Chain of Responsibility(CoR)模式(职责链模式)在某请求被委派给服务提供者来负责这一点上相似。区别在于 Reactor 模式为特定的事件源关联特定的事件处理器(Event Handler)，而 CoR 模式在链上搜索和定位第一个合适的事件处理器。

Reactor 模式可被考虑为异步的 Proactor 模式的一个变量。Proactor 支持充分异步事件(completion of asynchronous events)触发的多个事件处理器的复用和分派。对比之下，Reactor 模式负责当一有可能抛离阻塞发起同步操作时可靠地复用和分派被事件触发的多个事件处理器。

Active Object 模式[17]在函数的执行和函数的调用之间实现解耦(译者注：函数的执行在另外的线程中)，从而简化了处于不同的控制线程中的函数调用者访问共享资源。在线程不可用或者当线程的使用将会导致程序过于复杂而显得不适合的时候，Reactor 模式常会用在 Active Object 模式里。

Reactor 模式的实现提供了事件复用的 Facade 接口，Facade 是一种能在具有复杂对象关系的子系统和使用该子系统的应用程序之间加入防护层的接口。

参考

[1] W. R. Stevens, UNIX Network Programming, First Edition.Englewood Cliffs, NJ: Prentice Hall, 1990.

[2] D. C. Schmidt, “ACE: an Object-Oriented Framework forDeveloping Distributed Applications,” in Proceedings of the 6th USENIX C++ Technical Conference, (Cambridge, Massachusetts),USENIX Association, April 1994.

[3] W. Pree, Design Patterns for Object-Oriented Software Development.Reading, MA: Addison-Wesley, 1994.

[4] D. C. Schmidt, “An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit,” Tech. Rep.WUCS-95-31, Washington University, St. Louis, September 1995.

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

[6] D. C. Schmidt and P. Stephenson, “Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms,”in Proceedings of the 9th European Conference on Object-Oriented Programming, (Aarhus, Denmark), ACM,August 1995.

[7] S. Berczuk, “A Pattern for Separating Assembly and Processing,” in Pattern Languages of Program Design (J. O. Coplienand D. C. Schmidt, eds.), Reading, MA: Addison-Wesley,1995.

[8] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Initializing Communication Services,” in Pattern Languagesof Program Design (R. Martin, F. Buschmann, and D. Riehle,eds.), Reading, MA: Addison-Wesley, 1997.

[9] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging,” in Proceedings of the 2nd Conference on Object-Oriented Technologies andSystems, (Toronto, Canada), USENIX, June 1996.

[10] M. A. Linton and P. R. Calder, “The Design and Implementation of InterViews,” in Proceedings of the USENIX C++ Workshop, November 1987.

[11] D. C. Schmidt, “The ACE Framework.” Available from www.cs.wustl.edu/↑ schmidt/ACE.html, 1997.

[12] Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.0 ed., July 1995.

[13] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Brokers,” Computer Communications, vol. 21, pp. 294–324, Apr. 1998.

[14] D. C. Schmidt and T. Suda, “An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems,” IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems),vol. 2, pp. 280–293, December 1994.

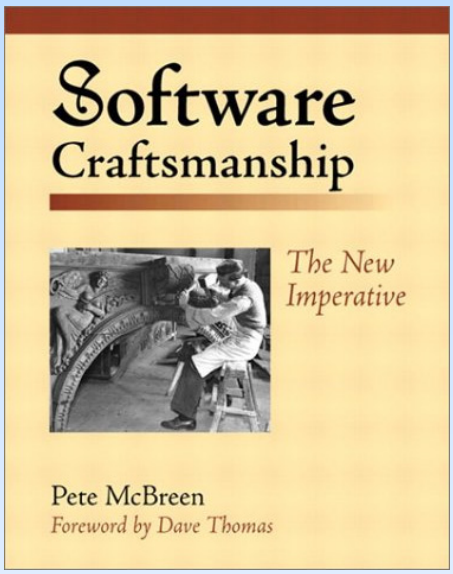
[15] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging,” USENIX Computing Systems, vol. 9, November/December 1996.

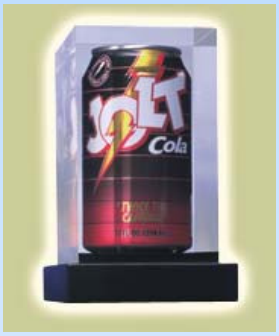
[16] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[17] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *Proceedings of the* 2nd *Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

[18] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, “Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers,” in *The* 4th *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

《软件工艺》





2002 Jolt Awards 获奖书籍

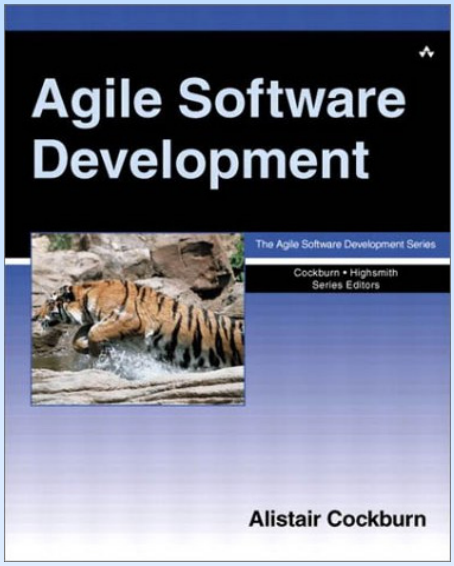
Pete McBreen

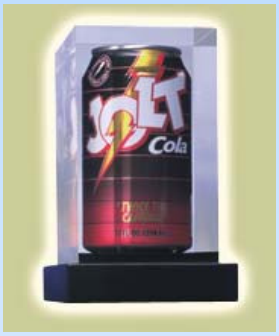
翻译：UMLChina 翻译组

工艺不止意味着精湛的作品，同时也是一个高度自治的系统－－师傅（Master）负责培养他们自己的接班人；每个人的地位纯粹取决于他们作品的好坏。学徒（Apprentice）、技工（Journeyman）和工匠（Craftsman）作为一个团队在一起工作，互相学习。顾客根据他们的声誉来进行选择，他们也只接受那些有助于提升他们声誉的工作。

中文译本即将发行！

《敏捷软件开发》





2002 Jolt Awards 获奖书籍

Alistair Cockburn

翻译：UMLChina 翻译组 Jill

我是一个好客人。到达以后，我把我的那瓶酒递给女主人，然后奇怪地看着她把酒放进了冰箱。

晚餐时她把酒拿了出来，说道，“吃鱼时喝它，好极了。”

“但那是一瓶红酒啊。”我提醒她。

“是白酒。”她说。

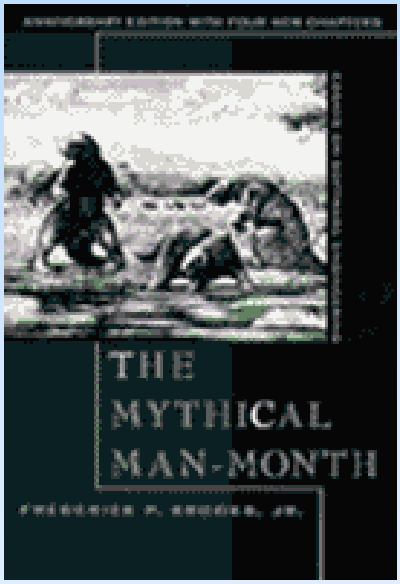
“是红酒。”我坚持道，并把标签指给她看。

“当然不是红酒。这里说得很明白...”她开始把标签大声读出来。“噢！是红酒！我为什么会把它放进冰箱？”

我们大笑，然后回顾我们为了验证各自视角的“真相”所作的努力。究竟为什么，她问道，她已经看过这瓶酒很多遍却没有发现这是一瓶红酒？

中文译本即将发行！

《人月神话》



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 Adams Wang

散文笔法，绝无说教，大量经验融入其中

在所有恐怖民间传说的妖怪中，最可怕的是人狼，因为它们可以完全出乎意料地从熟悉的面孔变成可怕的怪物。为了对付人狼，我们在寻找可以消灭它们的银弹。

大家熟悉的软件项目具有一些人狼的特性（至少在非技术经理看来），常常看似简单明了的东西，却有可能变成一个落后进度、超出预算、存在大量缺陷的怪物。因此，我们听到了近乎绝望的寻求银弹的呼唤，寻求一种可以使软件成本像计算机硬件成本一样降低的尚方宝剑。

但是，我们看看近十年来的情况，没有银弹的踪迹。没有任何技术或管理上的进展，能够独立地许诺在生产率、可靠性或简洁性上取得数量级的提高。本章中，我们试图通过分析软件问题的本质和很多候选银弹的特征，来探索其原因。

中文译本即将发行！

通俗而且深刻的软件开发思想杰作—UMLChina 推荐

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

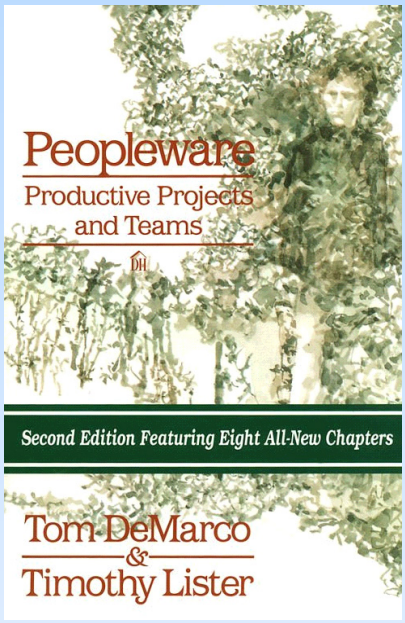
翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

面向对象技术在给我们带来好处的同时，也会增加成本，其中很大一部分是培训费用。经验表明，一个不熟悉 OO 编程的新手需要 3 个月的培训才能胜任开发工作，也就是说他拿一年的薪水，却只能工作 9 个月。这对一个拥有成百上千个这样的程序员的公司来说，费用是相当可观的。一些公司的主管们可能一看到这么高的成本立刻就会说"不能接受。"由于只看到成本而没有看到收益，他们会一直等待下去，直到面向对象技术过时。这本书不是为他们写的，即使他们读了这本书也会说（其实也有道理）"我早就告诉过你，采用 OO 技术需要付出昂贵的代价以及面临很多的危险。"另外一些人可能会决定启动一个采用 OO 技术示范项目，并观察最终结果。还有人仍然会继续在原有的程序上修修改改。当然，也会有人愿意在这项技术上赌一赌。

中文译本即将发行！

《人件》



《人件》第 2 版

Tom Demarco 和 Tim Lister

翻译：UMLChina 翻译组方春旭、叶向群

微软的经理们很可能都读过—amazon.com

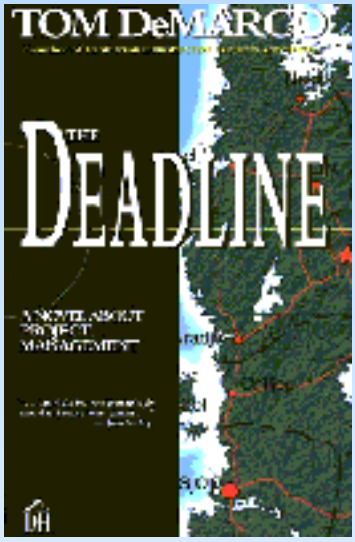
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

许多开发经理采用了类似的态度，他们竭尽全力地使自己确信没有人能够取代自己。由于害怕一个关键人物要离开，他强迫自己相信项目组里没有这样的关键人物，毕竟，管理的本质是不是取决于某个个人的去留问题？他们的行为让你感到好像有很多人物储备在那里让他随时召唤，说"给我派一个新的花匠来，他不要太傲慢。"

我的一个客户领着一个极好的雇员来谈他的待遇，令人吃惊的是那家伙除了钱以外还有别的要求。他说他在家中时经常产生一些好主意但他家里的那个慢速拨号终端用起来特别烦人，公司能不能在他家里安装一条新线，并且给他买一个高性能的终端？公司答应了他的要求。在随后的几年中，公司甚至为这家伙配备了一个小的家庭办公室。但我的客户是一个不寻常的特例。我惊奇的是有些经理的所作所为是多么缺少洞察力，很多经理一听到他们手下谈个人要求时就被吓着了。

中文译本即将发行！

《最后期限》



《最后期限》

Tom Demarco

翻译：UMLChina 翻译组 透明

这是一本软件开发小说

汤普金斯在飞机的座位上翻了一个身，把她的毛衣抓到脸上，贪婪地呼吸着它散发出的淡淡芬芳。文案，他对自己说。他试图回忆当他这样说时卡布福斯的表情。当时他惊讶得下巴都快掉下来了。是的，的确如此。文案……吃惊的卡布福斯……房间里的叹息声……汤普金斯大步走出教室……莱克莎重复那个词……汤普金斯重复那个词……两人微张的嘴唇触到了一起。再次重播。"文案。"他说道，转身，看着莱克莎，她微张的嘴唇，他……倒带，再次重播……

....

"我不想兜圈子，"汤普金斯看着面前的简报说，"实际上你们有一千五百名资格相当老的软件工程师。"

莱克莎点点头："这是最近的数字。他们都会在你的手下工作。"

"而且据你所说，他们都很优秀。"

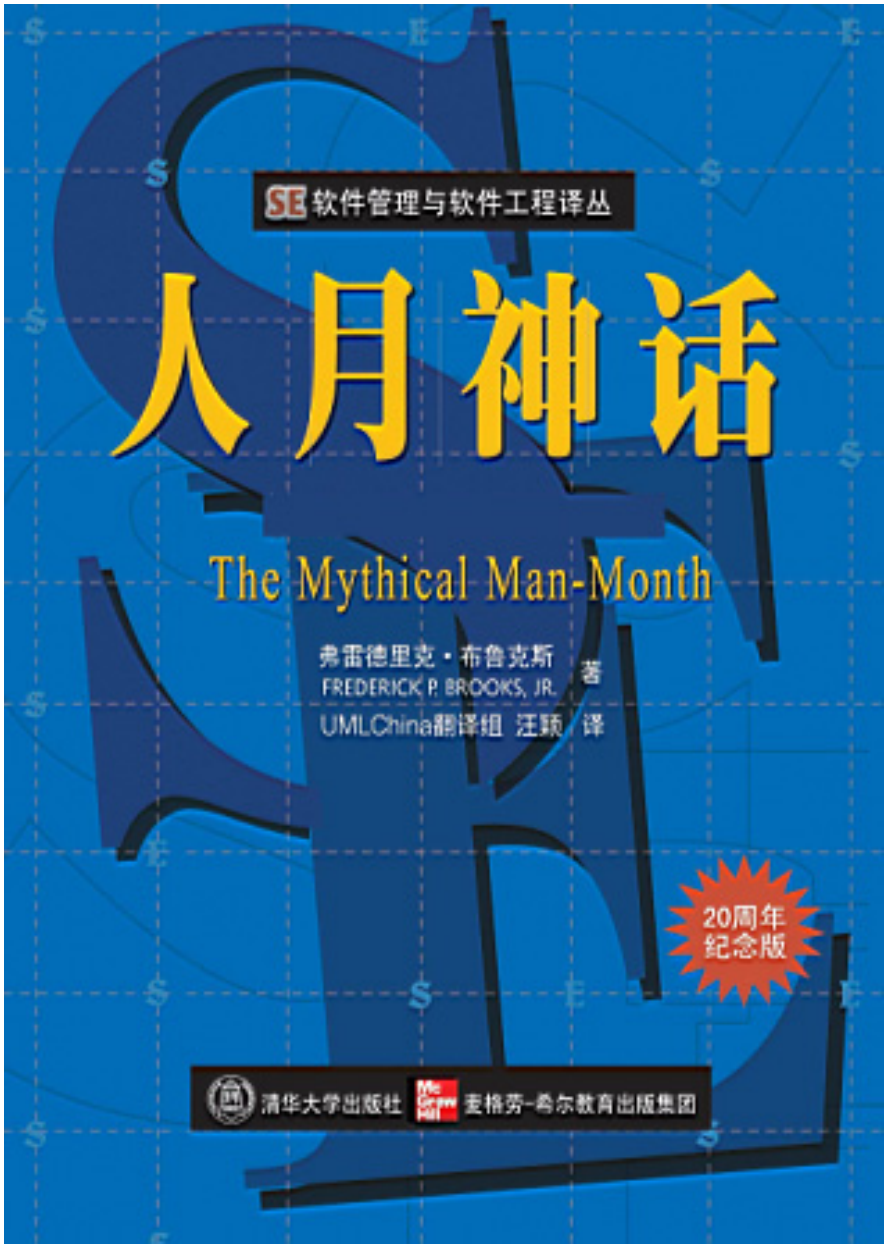
"他们都通过了摩罗维亚软件工程学院的 CMM 2 级以上的认证。"

中文译本即将发行！

《人月神话》20 周年纪念版评论集

陈懋成 编译

查看评论



Brian Kernighan

我唯一一本读过一遍以上的书，是 Fred Brooks 的《人月神话》，实际上我每过一两年都重读一遍。部分原因是这本书文笔很好，部分原因是书中的忠告很有价值，即使是 25 年以后。当然，现在很多细节上的地方，和我们做事情的方法，都有不同。我们的工作更自动化，计算机的“马力”更强劲，但书中依然有许多好的忠告，我非常推崇这本书。这是我唯一能想起来的你能从中体会到乐趣和思想的计算机科学书籍。

Ed Yourdon

有些书对于读者和作者象是年金，它们年复一年的分红。《人月神话》就是这么一本书。我直到在 1994 年接到 Brooks 教授的电话，才真正完全赏识它。

原因来自他的电话，电话中他说，他的出版商请他修订他在 1975 年第一次出版的这本书。我立刻表示了我的一点妒忌、羡慕；因为我的出版商从未叫我修订一本出版接近 20 周年的书。确实，我甚至表示了这样的意见： 当代的软件工程师不会考虑阅读这本如此古老的书，因此，可能无法卖出一本。“哦，不”，Brooks 教授回答道，“《人月神话》到目前为止，每年稳定的卖出 10000 本。”

嫉妒、羡慕和突然的现实：什么是我们拥有的象年金一样的东西。我高兴地说，自此书出版至今，我已读过 1975 版至少四遍；在接到 Brooks 电话后，我再次将它从书架上拿下来，重读了一遍。但这次，是有特殊目的的：实际的原因来自于他的电话，Brooks 教授告诉我，目的是发现自这本书 1975 年出版以来，计算机领域是否有有意义的事件发生。

我必须申明，这样的问题是相当难以回答的。Brooks 继续告诉我，他现在已基本离开软件工程社团，将他的大部分专业精力投身于虚拟现实领域的研究。所以，在准备再版这本书时，他想知道：什么已经改变了，什么还没有？他的原书中哪些是正确的，哪些是错误的，哪些是不切题的。

当然，我不是向他提供同样信息的唯一的人；我的一些同僚、大量的业界领袖、作者、顾问和摇旗呐喊者都被邀请回答这个问题…。我们所有人很高兴地做了。并且，象你所期望的一样，我们的输入被 Brooks 教授处理、分析、过滤、综合进那个非凡的新版本——它是真正的国际财富。

原书的内容仍在那儿，现在新增了四章。它们是 Brooks 教授对他的思想的反思和对我们的反馈的反应。新增的第一章由恰到好处地浓缩的原书主题组成；包括可以被称为书的中心论点的东西：大型编程项目忍受管理问题，这些问题因为劳力的分配而与小项目有本质上的差别；软件产品的概念完整性是如此关键；概念完整性是困难的但可达到的。第二章小结了二十年后 Brooks 对这些主题的观点。第三章是他 1986 年首发于 IEEE Software 的经典报告：《没有银弹》的重印。最后一章是对 Brooks1986 年的断言 “在未来十年，依然没有银弹”的思考。

年轻的软件工程师、吝啬的研究生、懒惰的软件老手常请我标示出当前为止最好的软件图书。“如果我带着仅有的一本计算机书在沙漠荒岛上，”他们问，“应该是哪本书？”这是个荒谬的问题，但人们坚持要个答案。假如你真的被放逐到这样的小岛（或者你决定躲藏到这样的地方去避免 2000 年软件崩溃的恐惧！），《人月神话》应该紧随着你。

Jason Bennett

背景

在我提交我的关于《人月神话》的书评前，我意识到没有介绍此书的第一版是我的疏忽。那是 1975 年。程序员辛苦的在最早的哑终端上工作，这些终端连着来自 IBM 的笨重主机。也许，如果足够幸运，程序员可以工作在一台小型计算机，它来自于业界的新星公司 DEC，和它们的 PDP 系列计算机上。FORTRAN 和 COBOL 是主要语言，夹带着一些 PL/1。C 是地平线上的亮点，刚刚开始离开 Bell 实验室走它自己的路。在整个工业范围中，汇编语言仍在广泛的应用。进入 Frederick Brooks 的书，工业界的原作之一。Brooks 在反思他在 IBM 的时间，特别是他在 1960 年代中期工作于 System/360 和它的操作系统，OS/360（独特的名字，不是吗？）的一段。Brooks 试图指出哪些做对了，哪些没有，特别是如何构造和管理全部程序。他因此开始书写软件项目管理的第一个条约，一个软件工程不可缺少的部分。二十五年后，我们仍然读这本书，学习它。在业界，大部分书六个月后就无用了，这本书则是空前的。记住，虽然，最终会有书能达到它的水平。

哪些是好的？

你可能很想知道，为什么这本书能持续如此长的时间。答案是简单的：书中的技术对大众的教训来说是次要的。简单说，Brooks 指出下列几点：

1. 编程必须进入到软件工程，以便持续改进技艺的状态；
2. 任何好的工程产品，必须是概念和体系结构的完整结合；
3. 软件开发中的焦油坑（the tar pit）可以通过尽责、专业的过程得以避免；

书中有如此之多的经典语录，以致这篇简单的书评不能全部包括。当然，最著名的是 Brooks 定律：加人将延迟工程（隐义：这样做没有帮助）。在 MMM（《人月神话》）中外科开发团队、第二系统效应、文档的重要性等均被覆盖，有些还是第一次出现。通过这本书，Brooks 覆盖了所有因素，这些因素是成功完成一个主要软件项目必须做的；同时，在书的各部分中，他给出了一致的软件工程与项目管理的坚实的基础。事实上，这是第一本主要的正确汇集工程师需要的关于大型软件项目开发知识的书，书中相关知识来自于对已完成项目的看法。除了原书，Brooks 1986 年的随笔《没有银弹》也被包括进去，同时带着 Brooks 在这本书出版二十五年后的思想。这些随笔每篇都值得一读，因为它们给出了当前工业所处位置的精确评价。可以这么说，容易的部分已在我们身后，让我们从这儿开始实际的工作。

哪些是有害的？

关于这本书哪些是有害的，答案是：人们没有读它。没有特别的好理由，他们仅仅是太忙于读最新的关于今天时尚的书了。当然，具有讽刺意味的是，今天的时尚可能是明年的笑话，而 MMM（《人月神话》）将从现在开始下一个十年。如果，学校发给你这本书，再次读它（你可能不是第一次:-)）。如果你从没听说过它，明天开始读。

书中哪些是对我们的有用的？

为什么我们每一人都应该读这本书？对于一串半组织化的小组哪些项目管理是要紧的？在我们为世界大同努力奋斗时，我们是否做的不好？嘿，是或不是。

首先，我们没有商业软件的压力。如果 Apache 明天发布而不是今天，没问题。每个人可能是坏的，所以我们不会做的更糟。因为如此，大部分开放源码项目，依赖一个或几个领导者，他们追踪着每件事和版本。我们没有许多拥有大量不同文档的项目。

其次，我们也依赖于低的期望值。没有人对我们有成功的预期，所以，对于这个运动，任何一个成功都是个胜利。现在，所有的眼睛都盯着我们。如果一个发布版延迟一个月发布，每个人都会说我们在保持时间表上做得不如微软。如果我们因为我们的可怜的设计而重写了一个应用，我们将不能提供有效的替代品。我们将不得不做得比其他人更好以证明我们自己，并且我们不能做面向特定平台环境的产品。如果开放源码社区比工业界学习 MMM 学得更好，那么，我们将获胜。

再次，作为一个工业，我们必须提升我们软件的期望。我们要为精确的时间表努力奋斗。我们需要避免第二系统的影响。我们需要知道为什么你不能加人到已经延误的项目去加速开发。长话短说，我们需要仔细工作以及时产出合格的软件，而不是仅仅将原料放在那儿。

Francis Glassborow

我第一次接触这本书在近二十年前。作为一个学校老师，我毫不怀疑这本书应是我的天才计算机专家的小伙子们需要读的。也许我的成功不能表示它的价值，也许人们不知道它。

计算机图书常常在用旧前很长时间就已过时了，但这部书是一个值得注意的例外。作者描述了管理的失败导致延误了在恰当时候交付的问题，在今天——《人月神话》首次出版以来二十周年后，依旧在发生。加倍的工作力量在今天和他那时一样不能解决时间（表）的流逝。

如果，你已经拥有了老版本，你仍然会为带有额外的四章的新版感到高兴；如果你以前从未读过它，把其他事放在一边，立刻补上这重要的一课。象老版本对暴露关于延迟六、七十次的 IBM 感兴趣一样，这个版本增加了一些关于微软的注释（尽管如此，每晚重建开发项目的策略不能解释为何 Windows95 的 M8 beta 版的版本是 950 版）。

Chris Larson

自 Frederick P. Brooks, Jr. 出版经典著作《人月神话：软件工程随笔》已经 20 年了。在这部注重实效、明晰的书中，Brooks 剖析了许多工程管理的神话，这些神话来自他在年轻的软件工业中有意义的实践。典型的，他抨击了在项目中增加人手可以促进项目的完成的幻想。带着实例、幽默、严密的逻辑，Brooks 展示了这些神话实际上如何给软件项目带来灾难并导致延迟。

你不能用人力换取时间

他的思想恰到好处地应用于文档项目的管理。有多少次查看文档项目的经理思考通过增加人手缩短时间表？这是个简单的导致陷入的谬误，但 Brooks 清晰地表达了这样的思想—“人力可以与时间互换”的实际效果。

Brooks 说，“导致大量软件项目进入失控的原因是时间表的缺漏，这比所有其它因素的组合还多。”他给出了几个原因。首先，Brooks 责备可怜的估计技术，它错误的“搞乱了前进中的努力。”因为估计的不确定性，经理们缺少“礼貌的倔强”去支持那些看上去比其它需要更长的时间线的步骤。一旦时间（表）流逝，倾向于加入更多的人力，而这就像“火上浇油”，会导致一个新的灾难生成的循环。

当然，一个基本的错误就是假定人力可以严格的和时间交换。Brooks 指出，这个公式仅在项目成员不需要和其他人交流的项目中成立，比如：拾棉花。

然而，一旦项目中包含连续任务和依赖关系时，可以交换的思想立刻就土崩瓦解了。“交流的努力，”Brooks 说，“必定导致加入大量要做的工作……。三个工作者要求的成对交互交流三倍于两个；四个六倍于两个。”这个交流的时间（不包括培训时间）吸收了许多增加人力分担任务带来的好处。

那么，答案是什么？

Brooks 认为由一个组织，其结构类似于外科团队的，由一些专家设计并完成全部项目的核心工作而由另外的人力在特定的途径上支持他们的努力，来执行产品开发可以治愈上述病症。

Brooks 说到，这条仅有的路是在一个项目早期完成“概念完整性”的通途。大部分关于这些完整性的重要的表述通过这个规范发生，“那是一本计算机手册加上性能规范……第一批文档中的一篇产生计划中的一个新产品，最后的文档完成它。”

文档是关键

Brooks 继续雄辩的支持文档，列出用户需要的每一项，以免只见树木不见森林：程序的目的、环境、选项、逻辑、“运行时间、”等等。并且认为：“大部分文档需要在程序编写前起草，因为它把基本的计划决策具体化了。”

作为技术交流者，我们需要听到来自工程方面的更多的各种各样的支持。尽管 Brooks 的环境——运行于大机上的大型编程项目——已经转换到我们今天的分布式计算环境，Brooks 的逻辑和清晰的思想依旧吹走了仍然威胁卷入项目管理中的迷雾。

Frank Chance

介绍

出版于 1975 年的《人月神话》是软件开发方面的经典作品。1995 年版包括了令人感兴趣的新的几章，但原来的随笔依然是这本书的心脏与灵魂。在这本书中，Brooks 解决了如何组织和管理大规模编程项目的问题。这些项目要求成百上千的程序员，产生几百万行代码（想想 SAP、Oracle 数据库引擎、Windows2000）。这部书由一系列简明的随笔组成。在这篇评论中我将讨论开篇随笔——我的最爱之一。

焦油坑

Brooks 将大系统编程作比喻作史前的焦油坑来开始他的第一篇随笔：“记忆中，我们看到恐龙、猛犸象、剑齿虎正在挣脱沥青的魔爪。挣扎得越剧烈，陷入的越深，没有哪只野兽足够强壮或熟练，它们最终都沉没了。大系统编程在过去的十年间就像焦油坑，许多大而强有力的野兽在其中已经惨烈地失败了。大部分已实现并在运行的系统，很少有达到目标、时间表和预算的。大和小、厚重和细实，一个接一个的团队卷入了沥青（陷阱）。没有什么事情似乎会导致这个困难——任何特殊的手掌都能被拉出来。但同时并相互作用的因数的相互聚集导致运动越来越慢。每个人似乎都惊讶于问题的难缠，难于面对它的本质。”

记住，这些话写于 1975 年。今天它们仍然可用吗？考虑一下 WindowsNT5.0。第一次计划于 1997 年发布，随后延迟到 1998 年早期，1998 年末，然后是 1999 年（为此它被重新命名为 Windows2000）。这儿是一些公开的估计：

- 5,000 程序员。
- 35,000,000 行代码。

显然，NT5.0 是个大系统编程项目。同样显而易见，Brooks 的焦油坑在今天同 1975 年一样普遍！

让我们继续 NT5.0 的例子。假设最糟糕的情况，全部 35,000,000 行代码都是新编的。有理由假设开发工作大致在 1994 年开始。所以我们有：

- 5,000 程序员 X 5 年 = 25,000 程序员年
- 35,000,000 行代码 / 25,000 程序员年 = 1,400 行 / 程序员年。

如果你是个程序员，或者你只接受过编程课程的教育，这个数字（1,400 行每年）似乎令人惊异的低。我们当中的大部分人都能在一两天内堆积出接近一千行的代码。什么使得 Microsoft 的程序员一整年才产出 1,400 行代码？

两种可能性跃入我们的脑海：

- Microsoft 雇用了 5,000 名不合格的程序员去开发 NT 5.0。

或者

- 写一个大规模的程序系统产品远难于堆砌出单一的程序。

Brooks 将讨论认为后一个答案是正确的。他由定义术语开始：

（1）程序

一个独立的程序是我们两天编程狂欢的结果。它是准备自己运行于我们编程的那台机器上的。如果我们加上文档、通用化代码、编写测试用例、使得代码可以由其他无关的编程人员来维护，我们就有了：

（2）程序产品

另外，如果我们接受我们的程序，并且完整地定义了它的接口使得它达到预定义的规范，并且测试了它和大量的其它组件的交互作用，我们就有：

（3）程序系统组件

并且如果我们都做了（加上文档、通用化代码、编写测试用例、使得代码可维护、定义了接口、测试了交互作用），我们就有：

（4）程序系统产品组件

Brooks 用手边的三倍规则说明在上述每个步骤中的工作要求：

（2）＝3 倍（1）的人力

（3）＝3 倍（1）的人力

（4）＝9 倍（1）的人力

或者，换句话说，开发一个独立的程序仅仅要求开发一个程序系统组件的 1 / 9 的人力。

回到 Microsoft 的例子，如果我们将这个 9 倍的因子乘以 1,400 行每程序员年的生产力测量，我们得到 12,600 行每程序员年（举例来说，假设我们掌握每一程序员，并且使得他们独立工作，堆砌在单一的程序上）。在一篇独立的随笔中，Brooks 引用一个发现这点的经理的话说，平均他的每个程序员仅能将他的一半时间用于开发－－其它时间由文书工作、会议和各种其它任务所占据。把这些因素考虑到 Microsoft 的例子中，我们达到了 25,200 行每程序员年。那么，Microsoft 的程序员开始看来非常可敬。另一个测量自 1975 年来有了很小的改变，Brooks 引用的估计是 1,000 行每程序员年。如果上面引用的 1,400 行每程序员年是精确的，那么，它表现了在 1975 年到 1995 年 20 年间，生产力仅仅提升了 1.75%每年。这个结果证实了 Brooks 的另一个假定－－程序员的生产力相对是个常量，它不受开发所用的语言的影响。因此，实际的生产力收获来自于迁移到高级语言编程，这些语言每行表达了更多的实际工作。尽管目标是大系统项目，Brooks 的解释常常被广泛的应用。例如，这个第一篇随笔用标有“手艺的快乐”和“手艺的悲哀”的小节来结束。在悲哀中，他讨论了荒废的问题：

“…这个人们已经工作了很长时间的产品，显然在完成前将被废弃。同事和竞争者已经在热烈地用新的和更好的主意反击。人们的孩童般想法的取代已经不仅仅在构思，而且付诸时间表。这一切总是似乎比它的实际更糟糕。新的和更好的想法通常在完成之前不被应用；它仅仅被谈论。真老虎永远不能和纸老虎相比。”

小结

Brooks 的随笔涉及到了大系统编程所固有的多种挑战，但对任何投身于软件开发的人来说读这本书都是有用的。题名的随笔（《人月神话》）讨论了许多编程任务的不可分割性，和为什么增加人力到软件项目中无法产生效用。我的另一篇最爱是“贵族、民主和系统设计”（概念完整性的讨论）和“计划和投放之路”（在付运前多次交付的明确计划的益处）。一些问题已经因为技术的进步而废弃，例如关于如何在一个大型团队中分发写好的文档。

然而，你可能惊讶 Brooks 面对的许多问题今天如何阻止我们。另外的益处是 Brooks 简洁、清晰的作品读起来令人愉快。如果你是个程序员，如果你和程序员一起工作，如果你管理程序员，你应该阅读这本书。

TAL COHEN

人们说在计算机世界中每件事都在迅速的变化，然而，在二十多年间，一些重要的事情依旧没有改变。

如果某个工作可以由十个人在一个月内完成，人们说这个工作要求十个人月（man-months，也叫“person-months”）。简单算术表明，如果你分配二十个人去作同样的工作，它应在五个月内被完成。（译注：应该是半个月）我想这个逻辑在许多项目中大概都是正确的，否则，经济学家将不会那么令人喜爱。

然而，在软件设计的世界，这个承诺是个彻底的谬误。没有那条途径可以产生它。一个能由一个程序员在两个月内完成的程序，也许会花去两个程序员三个月的时间。早在 1975 年，当软件工程还是非常年轻的专业时，Frederick Brooks 敏锐地观察到人月概念仅仅是个神话。

在 70 年代，软件工程项目管理的问题背景是：大部分经理是受的经济领域的教育而不是计算机，并且他们熟悉的理论不能简单的用于软件项目。事实上，没有用于软件设计项目的相同的理论。因为直到今天，大部分软件项目仍不能按照时间表发布，所以我们有个大胆的暗示：这个问题象许多 Brooks 在书中谈到的那些问题一样，仍未解决。

在 20 周年纪念版的序言中，Brooks 写到：

> 让我高兴和惊讶的是，《人月神话》在 20 年后依旧流行。

实际上，这真令人羞愧。它指出了在二十年间，软件工业没有学到一些严重的教训，它仍在付着学费。软件工程依然被认为和其它工程专业相比是个很陌生的艺术。真的，我是第一次接纳软件编写中存在艺术的观点。它是一种美丽的、精巧的艺术，仅仅能被那些掌握同样艺术的人所欣赏。我相信，当一个真正的代码艺术家完工时，它比建筑更加迷人、比油画更加引人入胜、比音乐更加令人思潮澎湃。然而，一个建筑师不会让房屋设计的艺术外观将他从房屋可以经受住最起码的地震的保证中转移出来。这个事实正渐渐的被大部分软件专家——那是些认为自己是艺术家而拒绝承认自己是工程师的人，所理解。（一个有趣的解决方案，是由我的好朋友提出的，他认为他自己是个“软件架构师”。）

Brooks 的工作是简单的而且对于那些被认为在软件业务领域是个专家的人是必需的，尤其对于在这个领域从事管理工作的人。这部书不仅仅指出了问题，而且也提出了些有趣的解决方案（例如：“外科团队”）。它指出了这个领域的每个专业人士都应该知道的一些非常重要的缺陷（例如“第二系统”效应）。最后，它提供了许多重要的见解和案例研究。

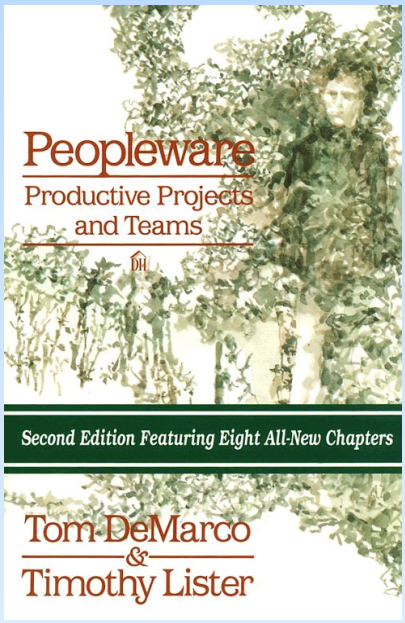
书中大部分材料和今天依然相关，就象它在原书写出来的时候。尽管，众所周知，部分材料已经过~~期~~时。在 20 周年纪念版中，相对于原文——叫做“经典”——的修订，Brooks 聪明地决定加入修改的章节。那些用来讨论暴露于 90 年代灯光下的出现的问题。我个人认为，在读完前面的每一章后应读第 18 章中有关的部分。第 18 章的标题是“人月神话的主题：是或非？”，它包括了从 1 到 17 章的修改信息。

最后，这个新版本包括重印的 Brooks 的著名随笔，“没有银弹”，那是在都柏林 IFIP86 会议上的特邀论文，后来在 Computer 杂志上发表。在这篇文章中，Brooks 推测在他的文章出版（1986 年）后的十年，不会发现有技术可以大大增强软件开发过程。九年过去了，Brooks 悲哀地回顾：他还是正确的。

征稿

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

《人件》



《人件》第 2 版

Tom Demarco 和 Tim Lister

翻译：UMLChina 翻译组方春旭、叶向群

微软的经理们很可能都读过—amazon.com

在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

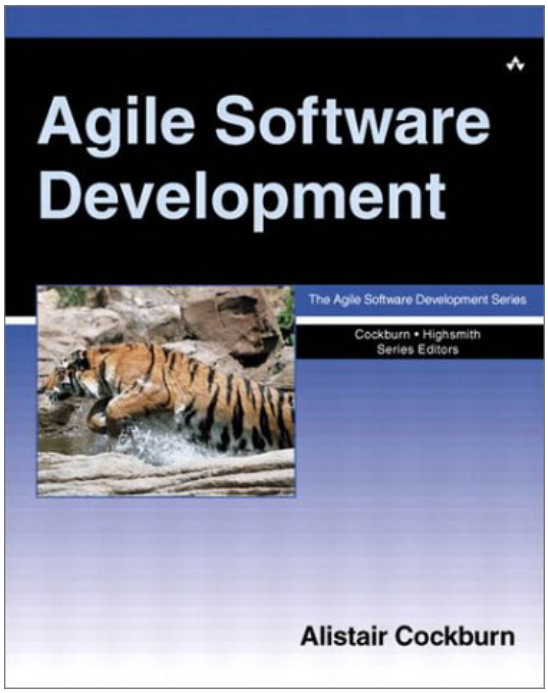
许多开发经理采用了类似的态度，他们竭尽全力地使自己确信没有人能够取代自己。由于害怕一个关键人物要离开，他强迫自己相信项目组里没有这样的关键人物，毕竟，管理的本质是不是取决于某个个人的去留问题？他们的行为让你感到好像有很多人物储备在那里让他随时召唤，说"给我派一个新的花匠来，他不要太傲慢。"

我的一个客户领着一个极好的雇员来谈他的待遇，令人吃惊的是那家伙除了钱以外还有别的要求。他说他在家中时经常产生一些好主意但他家里的那个慢速拨号终端用起来特别烦人，公司能不能在他家里安装一条新线，并且给他买一个高性能的终端？公司答应了他的要求。在随后的几年中，公司甚至为这家伙配备了一个小的家庭办公室。但我的客户是一个不寻常的特例。我惊奇的是有些经理的所作所为是多么缺少洞察力，很多经理一听到他们手下谈个人要求时就被吓着了。

中文译本即将发行！

《敏捷软件开发》翻译草稿样章

查看评论



Alistair Cockburn 所著的“Agile Software Development”2002 Jolt Awards 获奖书籍。它的中文译本《敏捷软件开发》即将由人民邮电出版社发行，译者为 UMLChina 翻译组的 Jill，这是翻译草稿的节选，得到出版社允许试登，仅供学习之用。

第四章 软件工程流程方法论

本章的目的是讨论和概括软件工程流程方法论这个话题，直到能够清楚地说明设计游戏的方法规则和如何来玩游戏。

“软件工程流程概念”一节覆盖了设计和比较方法论所需的基本词汇和概念。包括那些明显的概念，如：规则、技术、标准，还有一些不明显的概念，如：重型、正规性、精度、稳定性和容错性等，按书中第 14 页所描述的“三种水平的听众”来划分，这里的内容适合于第一种水平的人。接下来会有更多高级的讨论话题。

“软件工程流程设计原则”一节讨论了用软件工程流程来指导设计的七大原则，这些原则强调了转向重型方法所花费的成本，以及什么时候接受这种支出。它们还揭示了如何使用工件稳定性来决定应使用多少并行开发活动最为合理。

“XP under Glass”一节应用这些原则来分析一个已存在的敏捷软件工程流程。它也讨论了如何使用这些原则在环境条件稍有变化时去调整 XP。

“为什么软件工程流程如此重要？”一节回顾了前面讨论过程中的关键问题，并展示了软件工程流程方法论的不同用途。

创造软件的生态系统

“软件工程流程是社会建筑，”Ralph Hodgson 在 1993 年告诉我。两年以后我才开始理解他这句话的含义。

你的“软件工程流程”是你在完成软件的过程中每件事情的习惯做法。它包括你雇用什么人，雇用他们来做什么，他们如何在一起工作，他们生产什么，他们如何来共享资源。它是对工作、过程、团队中每个人的习惯的描述集合，是你的特定生态系统的产物，因此是你的组织的独特建筑。

所有的组织都有自己的工作方法和流程：它可能简单地指明你如何完成业务。甚至修车厂的常见的三重唱（proverbial trio）也有自己的工作方法——交换信息的方法，分解工作的方法，一起放回原处的方法——可以发现所有的工作都遵循假定的价值和文化标准。工作的方法包括人们选择如何支配他们的时间，他们如何进行交流，以及如何分配决策权力。

仅有很少的公司为如何记录所有这些习惯做法而烦恼（通常是那些大的顾问所和军用的地方）。少数公司的做法是创建一个专家系统，该系统能够打印项目所需的所有工作方法和流程，包括项目全体成员、复杂性、最后期限等等。我还没有看到一家公司能够获取文化设想或提供不同的价值和文化。

可以用一句话对软件工程流程进行总结和概括：“软件工程流程是你的团队同意遵循的一组习惯做法。”

“你的团队同意遵循的一组习惯做法”就是一个社会建筑。它也是你可以和应该不断回顾的建筑。

软件工程流程中的概念

methodology 这个词来源于 Merriam-Webster 的字典“一组相关的方法或技术。”方法是一个“系统的过程”，和技术很类似。

（牛津英语字典的读者可能会指出在一些 OED 的版本中仅将 methodology 定义为“对方法的学习”，其他的则包括这两种定义。这有助于解释 methodology 这一有争议的词。）

区分软件工程流程和方法是很重要的。读读下面的短语“从用例中发现类的方法”或“不同的方法适合解决不同的问题，”我们能够明白作者的意思是讨论技术和过程，而不是建立团队规则和习惯做法。在协同团队中人们的活动这个大的话题中，可以自由地使用软件工程流程一词。

协同是重要的。独自工作时能够完成相同水平设计的人，合作时常常可以完成更好的设计。反之，所有的聪明人在一起，如果没有协同、合作和沟通，也不能得到成功。我们中大多数人都看到或听说过这样的团队，团队的成功依赖于合作、沟通和协同。

图 4-1 软件工程流程的要素

结构术语

我发现的第一个软件工程流程结构包括七个要素。我画的图中包含了 13 个（参见图 4-1）。这些要素可应用于任何团队的工作，无论是软件开发、攀岩，还是创作诗歌。每个盒子所写的具体内容可能不一样，但这些要素的名字是不变的。

Roles，角色。你雇用了谁，你雇用他们来做什么，假定他们有什么技能。同样重要的是所期望的每个人的个性特点。项目管理者应善于管理项目组成员，用户界面设计者应有天生的外观设计才能并能够认同和理解用户的行为，面向对象程序设计人员应该能够对系统功能进行很好的抽象，顾问应该擅长解释各种问题。

当每个人没有自己应做的工作所需的能力时，项目会变得糟糕（例如，项目经理不能作出决策，顾问不善于沟通）。

Skill，技能。角色应掌握的技能。处于一个角色的人的“个人能力”是他的天赋和所受培训的产物。

程序员参加课程，学习面向对象、Java 编程和单元测试技能。

用户界面设计人员学习如何管理可用性测试和在纸上设计原型。

经理学习面试、激励他人、雇用和临界任务管理技巧。

最好的情况是人们能充分利用天赋的才能，不过在大多数情况下可以通过培训和练习来获取适当的技能。

Teams，团队。在各种环境下一起工作的角色的集合。

在一个小项目中可能仅有一个团队。在一个大项目中，可能会有多个相互交迭的团队，一些负责底层（harnessing）的专业技术，一些负责驾驭（steering）整个项目或系统构架。

Techniques，技术。完成一项任务的知识和过程。一些是对于单个人来说的（如编写用例、管理、设计类或测试用例），其他的则针对一组人（如项目回顾、小组计划会议）。通常，当使用一种可理解的知识对如何完成一个任务进行说明性介绍时，我使用技术这个词。

Activities，活动。团队成员的日常工作。典型的活动有：作计划、编程、测试、开会等。

一些软件工程流程方法论是强调工件的，意味着它们关注于需要产生的工件。其他的是强调活动的，意味着它们关注于人们的日常工作。因此，Rational 统一过程是强调工具和工件的，极限编程是强调活动的。它能获得更好的效率的部分原因是通过描述人们每天应该做什么（结对编程、测试先行的开发方式、重构等等）。

Process，过程。活动如何随时间展开，特别是有前置和后置条件的活动（例如，设计评审应在材料提交给参与者，并产生了改进建议列表的两天后进行）。强调过程的软件工程流程关注团队成员之间的工作流程。

过程图很少用来表现当工作需要重作时的这种回环型（loopback）流程。因此，过程图通常被用作工作流图，用来描述谁应该从谁（其他人）那里得到什么。

Work products，工件。人们所构建的东西。工件可以是临时性的，如 CRC 设计卡片，或是可以被永久化的东西，如使用手册或源代码。

我发现使用可交付成果来描述“符合组织边界要求的工件”很恰当。这允许我们在不同程度使用术语可交付成果：在两个子团队间传递的可交付成果是较大项目的工件。在一个项目团队和为下一个系统工作的团队之间传递的工件是项目可交付成果，需要更认真的处理。

工件在通常的术语中指“源代码”和“域对象模型。”用于每个工件的符号规则在工件标准中被描述。源代码标准的例子包括 Java，Visual Basic，和可执行的可视模型。类图标准的例子有 UML 和 OML。

Milestones，*里程碑*。标记过程或某时刻工作完成的事件。一些里程碑只是简单地宣称某项任务已经完成，另一些可能包括文档或代码的发布。

一个里程碑有两个关键特征：它发生在某个时刻，它可能会完全达到或未达到（不是部分达到）。一份文档可以发布或不发布，代码可以交付或无法交付，会议召开或不召开。

Standards，*标准*。团队对特殊的工具、工件和决策原则所采用的习惯做法。

编码标准可以这样声明："每个函数应在头部包含如下内容…"

语言标准可以这样声明："我们将使用可移植的 Java。"

绘制类图的标准可以这样声明："只显示全局函数的公有方法。"

工具标准可以这样声明："我们将使用 Microsoft Project、Together/J、JUnit、…"

项目管理标准可以这样声明："使用两天到两周的里程碑，每两到三个月交付中间工件一次。"

Quality，*质量*。质量是针对活动和工件而言的。

在 XP 中，团队所提交程序的质量是通过对源代码工件的检查来评价的："所有检入的代码都必须 100%通过单元测试。"

XP 团队成员也通过以下方式来评价他们的活动的质量：他们是否每天召开站立式会议？程序员和他们的编程伙伴多久轮换编程一次？在遇到业务问题时能否有客户进行解答？在一些情况下，质量被量化，以数字方式给出；在其他情况下，它是个模糊值（如：我对上次迭代过程中的团队士气不满意）。

Team values，*团队价值取向*。软件工程流程的其余要素由团队价值取向系统来管理。一个想着"尽快面市"的有进取心的团队和一个重视家庭的团队的工作模式是大不相同的，前者会加班加点，后者则下班就按时回家。

正如 Jim Highsmith 所指出的，一个小组的任务是探测并定位新油田，另一个小组的任务是以最低的成本从一个已探明的油田采集石油并装桶，他们的工作效率评价标准不同，所产生的规则也不同。

软件工程流程方法论的类别

Maier 和 Rechtin(在 2000 年)将软件工程流程方法论分为：标准化、合理化、参与式、启发式 四大类。

标准化软件工程流程方法论基于解决方案或按规程规定已知的工作顺序。例如有关电力系统和其他建筑方面的房屋配线的项目。在软件开发中，可以包含在这个类别中的有状态图 verification。

合理化软件工程流程方法论（与 Rational 公司没有关系）基于方法和技术。它们常用于系统分析和工程规程。

参与式软件工程流程方法论是基于涉众的，强调客户参与。

启发式软件工程流程方法论是基于课程学习的。Maier 和 Rechtin 在航空航天业（空间和飞机设计）中引用它们的用途。

作为一个不断成长的知识体系，软件工程流程中的一部分从启发式转化为标准化，成为典型问题的标准解决方案。如在计算机编程中，搜索算法已经成文，确定下来。而关于把人们放在公共的办公室还是私人的办公室的问题还没有成文化，即没有定论。

大多数的软件开发还处在适用启发式软件工程流程方法论的阶段。

里程碑

里程碑是项目中有趣事件发生的标记。在每个里程碑，担当某些角色的一个或多个人必须聚在一起来评价一个工件是否符合要求。

项目中有三种里程碑，每个都有其特殊的特性。它们是：

- 评审
- 发布
- 宣布（多为口头宣布）

在评审时，一些人检查工件。关于评审，我们关心以下问题：谁来进行评审？谁创造的这个条目？评审的结果是什么？极少有因为评审不合格使项目停止的案例；大部分评审的结果是给出一个应该被合并的建议列表。

发布发生在一个工件被分发或为了 posted for open viewing。宣告发布会的时间，在配置管理系统中检查源代码，在用户工作站上部署软件，这些都是不同形式的发布。关于发布，我们关心以下问题：要发布的是什么？发布者是谁？接收者是谁？是什么导致它被发布？

宣布里程碑是一个人对另一个人或多个人口头通知，该里程碑已达到。对宣布里程碑没有一个客观的测量标准；它只是一个简单的宣告或承诺。宣布里程碑非常有趣，因为它们在团队的社会结构内构建了一个用承诺编织的网。当第一次发现这种形式的里程碑时，我非常惊讶。

Discovering Declaration 发现宣告

第一个宣告里程碑是我在和一个有 100 个人的项目的技术文档编写组的经理的讨论中发现的。我问她如何确定什么时候该分配编写在线帮助的任务给一个组员（在线帮助的产生事件）。

她说当项目经理告诉她，应用程序的一部分对她来说已经“准备完毕”。

我问她“准备完毕”意味着什么，它是否意味着界面设计已经完成。

她说它只意味着界面设计已经相对稳定。其实，项目经理只是作了如下的允诺：

“我们估计我们将来进行的修改与技术文档编写人员的工作相比很小，文档编写人员因为修改而要重做的工作相对于全部工作也很小。所以这是开始编写文档的恰当时刻。”

这种声明纯粹是一种社会承诺。它是由一个受过训练的人所给的承诺，经过他的判断已经达到了一个平衡点，可以开始做后续的工作了。

宣告（“已经准备好了”）常是开发中由编码转向测试、alpha 交付、beta 交付，甚至是部署的一种里程碑形式。

作为一个研究者，我觉得宣告很有趣，因为我没有从以过程为中心的软件工程流程方法论中看到过对它们的描述，以过程为中心的软件工程流程方法论强调过程的入口和退出边界条件。当我们将软件开发看作合作游戏时，这一点更容易理解。在一个合作游戏中，项目团队的内部关系网，以及保持他们在一起的承诺出现的更多。

角色－可交付成果－里程碑图表（如图 4-15 所示）是观察软件工程流程概要的快速方法。通过过程图来观察的好处是它十分清楚地展示了包含在项目中的并行部分。它还允许团队看到关键阶段的完成，工件通过了评审。这有助于他们根据工件的中间状态来管理调整自己的活动，正如一些现代方法论所推荐的那样（Highsmith 2000）。

Scope 范围

软件工程流程的范围由它所试图覆盖的角色和活动的范围组成（如图 4-2 所示）。

最早的面向对象方法论认为设计人员是关键角色，并讨论了与该角色有关的技术、可交付成果和设计活动的标准。这些方法论在以下两方面考虑得不够充分：

- 它们的覆盖面不如所需要的那么广。一个真实的项目包括除 OO 设计者以外的很多角色，每个角色都应进行很多活动，有很多可交付成果，需要很多技术，而不仅仅是这些书中所列举的。

- 它们过分压缩了内容。设计人员需要的不只是他们工具箱中的一种设计技术。

图 4-2 范围的三个维度，软件工程流程是它的一个子集。

有很长历史和丰富经验的团队，如美国国防部、Andersen 顾问、James Martin 和其合作伙伴、IBM、和 Ernst & Young 已经总结出自己的软件工程流程方法论，能够覆盖标准的项目生命周期，甚至可以从项目的销售和项目的建立开始。他们的软件工程流程方法论能够覆盖项目所需的每个人，从团队助理到销售团队、设计者、项目经理和测试人员。

虽然这两类都是“软件工程流程方法论”，但它们所关注的范围是不同的。

一个软件工程流程的范围可以通过三维坐标系来描述：能覆盖的生命周期，能覆盖的角色，和能覆盖的活动（如图 4-2 所示）。

- 能覆盖的生命周期指出在项目的生命周期中，什么时候开始应用软件工程流程，什么时候不再使用它。
- 能覆盖的角色指出哪些角色属于被讨论领域。
- 能覆盖的活动定义这些角色的哪些活动属于被讨论领域。软件工程流程可以考虑填充时间表（一个正常应被包含的部分，项目经理通过它来监督和安排任务），可以忽略休假请求（因为它属于基本商业运作的一部分）。

阐明一种软件工程流程的目的范围有助于 take some of the heat out of 软件工程流程论点。常有这种情况，两种看起来矛盾的方法论分别以生命周期的不同阶段或不同的角色为目标。无法讨论它们的不同，直到阐明了它们各自的目的范围。

从这个观点，我们可以看到早期的 OO 方法论针对的是一个相对很小的范围。它们的典型做法是仅关注一个角色，即领域设计人员或建模者。因为只有一个角色，所以只表现实际的域建模活动，只覆盖分析和设计阶段。在这个很窄的范围内，它们只覆盖了一种或少数几种技术，并概要描述了一个或很少几个的可交付成果的标准。有经验的设计人员认为它们对整个开发过程来说不充分也就不足为奇了。

范围图帮助我们观察软件工程流程的各部分在哪里能很好的结合起来。例如，Constantine 和 Lockwood 的用户界面设计建议（Constantine 1999）和软件工程流程方法论的自然符合，忽略了对 UI 设计活动的讨论（把这部分留给更熟悉该主题的作者）。

如果没有范围坐标轴，人们将会问 Larry Constantine，“你的软件工程流程如何和市场上的其他敏捷软件工程流程相联系？”在 2001 年的一次关于软件开发的讨论会议上，Larry Constantine 说他不知道他正在设计一个软件工程流程，他只是想讨论设计用户界面的好方法。

将软件工程流程范围图放在看得见的地方，我们可以很容易发现它是多么符合 XP 所关注的范围，如图 4-3 所示。注意它缺乏对用户界面设计的讨论。它所关注的“为使用而设计”的范围如图 4-4 所示。通过这些图表我们可以看到，它们非常吻合。对于为使用而设计的方法和水晶系列方法也是如此。

图 4-3 极限编程的范围

图 4-4 Constantine & Lockwood 为使用软件工程流程片断而设计的范围

概念术语

讨论一个软件工程流程的设计，我们需要不同的术语：软件工程流程的大小、正规性、重量（复杂程度）、问题大小、项目规模、系统要紧程度、精度、正确度、相关性、容错性、可视性、缩放比例和稳定性。

Methodology Size 软件工程流程的大小。软件工程流程中所控制要素的数目。每个可交付成果、标准、活动、质量要求、技术描述都是要控制的要素。一些项目和作者会希望较简单的软件工程流程；另一些则希望使用较全面的软件工程流程。

Ceremony 正规性 软件工程流程中的精度要求和控制的松紧度。高正规性相当于严格的控制（Booch 1995）。一个团队可能在餐巾纸上写用例，并在午餐后对它们进行评审。另一个团队更喜欢填满三页纸的模板，然后专门花半天时间对它们进行评审。两个组都编写用例并对其进行评审，前者使用低正规性的方法，后者使用高正规性的方法。

软件工程流程中正规性的高低依赖于系统的要紧程度和方法论作者的恐惧和期望，正如我们所看到的。

Methodology Weight 软件工程流程的重量。是软件工程流程的大小和正规性的产物，等于控制要素的数量乘以每个要素包含的正规度。它是概念性的（因为软件工程流程的大小和正规度没有精确数字表示），但它仍然有用。

Problem Size 问题大小。问题中的元素数目和它们的交叉复杂度。对于问题大小没有一个绝对的衡量标准，因为掌握不同知识的人可能会以简化的方式来看待问题，这就降低了问题难度。一些问题在可讨论的相关量级上明显不同于其他问题（发射宇宙飞船和打印公司的发货单相比是个很复杂的问题）。

决定问题大小的困难常常在于对哪些人需要交付工件上有争议，以及相应的软件工程流程的重量如何。

Project Size 项目规模。指他们的工作成果需要被协同的那些人的数量，即团队成员的多少。根据不同情况决定，需要协同的可能只是编程人员，也可能是包括多个角色的整个部门。

很多人在使用短语“项目规模”上有歧义，它的意思一会儿是团队成员的多少，一会儿是问题大小，甚至在一句话中同时出现这两种意思。这就导致困惑。特别是因为小的、敏捷的团队常常胜过大而平庸的团队。

问题、团队成员和软件工程流程大小的关系将在下一节讨论。

System Criticality 系统要紧程度。由于系统出错造成的影响。如饮料机（本来要咖啡，结果出来可乐） 和飞机（可能机毁人亡）的系统要紧性显然不同。我将要紧程度简单的分为损失舒适性，损失可以自由支配的钱，损失不能代替的钱，或失去生命几类。可能还有其他的分类方法。

Precision 精度。当表述一个问题时精确到什么程度。Pi保留一位小数的近似值为3.1，保留四位小数的近似值为3.1416。源代码与类图相比要更精确；集成代码又要比它的高级源代码更精确。根据作者的想法，某些方法论很早就提倡更高的精度。

Accuracy 准确度。当表述一个主题时的正确程度？说“Pi保留一位小数的近似值为3.3”是不正确的。最终的对象模型应该比初稿更准确。最终的GUI描述应该比低逼真度的原型更准确。软件工程流程方法论覆盖了准确度的变化，也覆盖了精度的变化。

Relevance 相关性。是否考虑一个问题。如在用户界面原型中不讨论域模型。基础结构设计与收集用户功能需求无关。软件工程流程方法论讨论了相关性的不同方面。

Tolerance 容错性。允许有多少偏差。团队规则可能要求或不要求在项目代码中添加版本修订日期。该容错性声明可以为：必须能够找到修订日期，该日期可以手工添加，也可以通过一些自动化工具来添加。一个软件工程流程可以指定什么情况下空行，如何缩排，把它们留给人们自己去决定，或者规定可接受的范围。例如在一个判断标准中规定每三个月（前后可以放宽一个月）必须提交一个工作发布版本。

Visibility 可视性。对局外人来说是否能容易看出项目组是否在按照软件工程流程去做。主动过程如ISO9001很强调可视性问题。因为达到可视性需要额外开销（如时间成本，金钱，或者两者都有），敏捷方法论是一组不特别强调可视性的方法论。至于正规度，不同程度的可视性适用于不同的情况。

Scale 缩放比例。描述一样东西时允许多少组成元素同时出现。Booch以前的“类的类别”提供了一组类的缩小视图。UML中的“包”也允许为用例、类或硬件提供不同的缩小视图。项目计划、需求和设计也分别可以用不同的缩放比例来表现。

缩放比例和精度互相有点影响。打印机或监视器的分辨率限制了可以在屏幕上或纸上显示的细节的数目。不过，即使能把它们全都打印到一张纸上，一些人却并不需要看到所有的细节。他们想看的是一个概要的或称为高级别的版本。

Stability 稳定性。变化的可能性。我仅使用三个稳定性级别：初期大范围的变动，当团队刚开始进行项目开发时；中期变化较小，当一些开发活动已经处于中间阶段；后期相对稳定，刚好在需求/设计/代码评审以前，或产品发布以前。

一个发现稳定性程度的方法是问：“如果我今天和两周后问一个同样的问题，我有多大可能得到相同的答案？”

在初期大范围变动状态下，回答是：“开什么玩笑？谁知道它两周后会是什么样子！”

在中期变化较小状态下，回答是：“稍微有些相似，但细节肯定会变化。”

在后期相对稳定状态下，回答是：“非常相似，尽管会有很少一部分不同。”

判断稳定性的其他方法包括测量用例文本、图、基础代码、测试用例等中的“波动程度”（我没有试验过这些方法）。

精度

精度是软件工程流程中操作的核心概念。对于每类工件都分为低、中、高精度的不同版本。下面是一个关键工件的低、中、高精度的版本说明。

项目计划

项目计划的低精度版本视图是项目导图（如图4-5所示）。它展示了应完成的基本条目，这些条目之间的依赖关系，哪些应该被部署在一起。它可以表现每个条目所需的相关大量信息。它并不显示谁来做这些工作或完成这些工作需要多长时间（这是为什么称它为导图而不是计划的原因）。习惯使用PERT图的人会认为项目导图是粗糙的PERT图，它展示了项目间的依赖关系，扩大了标记的间隔以表现发布活动发生在哪里。

图 4-5 项目导图：一个项目计划的低精度版本

这种低精度的项目导图在组织项目（成员还没有组成项目组，没有建立时间期限）时非常有用。实际上，我使用它来得出时间期限和计划如何组织团队成员。

项目计划的中精度版本是项目导图的扩展，以展现团队和交付日期间的依赖关系。

项目计划的高精度版本就是众所周知的基于任务的甘特图，它能显示任务时间，分配关系，以及任务间的依赖关系。

计划的精度越高，它就越不堪一击。这就是为什么构建甘特图是一件让人恐惧的事情：它强调完成时间，并除去那些小的意外事件的时间。

用户界面设计

对用户界面的低精度描述是屏幕流图，它仅描述每个屏幕界面的意图和界面间的联接。

中等级别精度的描述由界面定义组成，包括字段长度和不同的字段或按钮激活条件。

用户界面设计的最高精度定义是程序源代码。

行为需求/用例

行为需求常被编写为用例。

一组用例的低精度版本是活动者－目标列表，主要活动者和目标的列表，它们与系统有关（如图4-6所示）。在项目刚开始时你需要区分用例优先顺序，并给团队成员分配工作，此时低精度版本视图非常有用。无论何时需要总览项目时它也很有用。

角色	目标
顾客	买产品
顾客	获取退款
Mktg.部门	进行提拔
经理	检查统计数据

图 4-6 一个活动者－目标列表：行为需求的低精度版本视图

中等精度版本由一段用例的摘要大纲，或用例的标题和主要的成功场景组成。中等精度版本包括扩展部分和错误条件，虽然对错误条件命名但没有把它们展开。

最终的最高精度版本包括扩展条件和它们的处理方法。

这些精度级别在Cockburn（2001c）中作了进一步描述。

图 4-7 责任－协作图：面向对象设计的低版本视图

项目设计

在面向对象设计中精度的最低级别是责任－协作图，它是UML对象协作图的一个非常粗糙的变种（如图4-7所示）。关于这种简单的设计表达方式的有趣一点是，人们已经能对其进行评审，并评论其中的责任分配。

中等精度版本是主要类、类的主要意图，和主要协作流程的列表。

中等偏高级别是类图、展示类、属性，和关系cardinality。

高级精度版本是类、属性、cardinality约束关系和功能及功能信号。这些内容常常被列在类图中。

最终的最高精度版本是源代码。

这些不同等级的设计展示了从责任驱动设计（Beck 1987, Cunningham URL-CRC），通过使用UML进行对象建模，到最终源代码的自然行进过程。这三个版本并不象想象中那样互相矛盾，而是自然出现在精度的改进过程中。

由于我们能够从图生成最终代码中获得好处，设计者在图中增加精度和代码生成注释。结果，图加代码注释变成了“源码。”C++ 或Java不再是源码，而变成被生成的代码。

图 4-8 使用低精度版本触发其他活动

为“精度”工作

人们为这些低精度的视图作了大量工作。在项目的早期阶段，他们做计划并评估。在稍后的阶段，他们使用低精度视图来做训练。

我认为到达一个精度级别的标准是，有足够的信息可以让另一个团队开始工作。图4-8展示了项目中的六类工件的演化。项目计划、用例、用户界面设计、域设计、外部接口和底层设计。

观察图4-8，我们发现一旦在适当位置有了活动者－目标列表，就允许草拟项目初始计划。这些连同时间、成员任务、估计目标可以组成项目导图。有了这些，团队可以分为几组来并行捕获用例框架。一旦完成了用例框架，或它们的重要子集，所有的专业团队就可以并行开始工作，进一步完成他们应负责的工件。

关于精度有一个问题需要注意：当精度增加时，所需的工作量会迅速膨胀。图4-9显示了当用例从活动者，扩展到活动者和目标，接着扩展到主要的成功场景。再扩展到不同的失败情况和其他一些扩展条件，最后扩展到修复活动这一系列过程中工作量的增加。对其他类型的每个工件也可以画出一张类似的图。

因为高精度的工件需要更多的精力，而且与低精度的副本相比修改很频繁，一个通用的项目开发策略是推迟，或至少是非常认真地管理它们的构建和发展演化。

图 4-9 由于精度级别的增加带来的工作量剧增（以用例为例）

稳定性和并行开发

稳定性，“变化的可能性”在整个项目过程中不断变化。（如图4-10所示）

图 4-10 在项目过程中降低波动性

团队从不稳定状态开始。过一段时间，团队成员降低了波动性，到达多变阶段如设计阶段。最后，例如在评审或发布前，他们使工作成果达到相对稳定阶段。在该点，评审员和用户向项目团队提出新的要求，使工作重新变得不太稳定。

在很多项目中，不稳定性会在一些不期望的场合出现，例如一个供应商突然宣称他不能按时交付，一个产品不能象预期的那样运行，或者一个算法不象预期的那样有可扩展性。

你可以认为在项目中能够通过努力来获取最大的稳定性。然而，对目标来说合适的稳定性程度是随主题、项目优先级、项目的不同阶段而变化的。关于如何处理工件和项目不同阶段的变化率，不同的专家有不同的建议。

最简单的方法是，“直到需求稳定再开始设计”；直到设计稳定再开始编码；等等诸如此类。它的两大优点吸引了很多人。不过，它充满了问题。

图 4-11 成功的顺序开发比成功的并行开发用的周期长（但总工作天数较少）

第一个优点是它非常简单。按这个时间表工作的人只需按顺序一个活动接一个去做就可以，当上游活动完成，下游活动才开始。

第二个优点是，如果没有特别重要的事件迫使需求和设计作出改变，经理可以通过认真计划人们应该在什么时间完成他们的特定工作，把在项目上所花的工作时间减到最低。

可是，它也有三个问题。

第一个问题是项目所需的时间只是需求、设计、编程、测试等活动的简单合计。这是项目所需的最长时间。在认真的管理下，项目经理使用最低的劳动力成本，代价是最长的项目周期。而对最高优先级是减少所需时间的项目来说，这是个不好的方案。

第二个问题是项目过程中常常会发生意外。当意外发生，就会出乎意料的导致需求或设计的修订，增加了开发的成本。最终，项目经理既没有降低劳动力成本，又没有减少开发时间。

第三个问题是缺乏从下游活动到上游活动的反馈。

在极少的案例中，从事上游活动的人如果没有下游团队的反馈也能做出高质量的工件。然而在大多数项目中，创建需求的人需要观察他们整理的运行版本，以便修正和最终确定需求。

通常，在看到系统运行后，他们会修改他们的需求，这将迫使设计、编码、测试等进行一系列的修改。整合这些修改会延长项目周期，增加总的项目成本。

图 4-12 顺序开发。每个工作组都要等上游工作组达到完全稳定才开始自己的工作。

图 4-13 并行开发。当它的沟通和重做能力指明时每个组就开始它的工作。如图所示，上游组在持续的流中将更新信息传递给下游组（由虚线箭头表示）

选择顺序开发策略只有在以下条件下才会有效，你能确信团队能够一次就完成好的、最终的需求和设计。很少有团队能做到这一点。

一个不同的策略，并行开发，可以缩短项目整体的时间并提供反馈机会，以避免由于重做带来的成本增加。图4-11和4-13证明了这一点。第156页“原则七，从非瓶颈活动中求效率”进一步分析了这一点。

在并行开发中，每个下游活动从某点开始，这个点是通过判断上游团队工作是否达到所需的稳定性和是否完整来确定的（当然，有相同上游组的不同下游组可以从不同的时刻开始）。下游组开始对有效信息进行操作，同时上游组继续工作，不断传递新的信息给下游组。

下游组假设上游组在正确工作，并不会遇到主要的变故，下游组将使他们的工作接近正确。当新的信息到达时，团队需要重做一些工作。

并行开发的关键问题是如何合理判断完成率、稳定性、重做的可能性，和团队内的有效沟通。

并行开发的主要优点有两个，恰好与顺序开发的缺点相对：

上游团队可以从下游团队获得反馈。设计者可以指出某些需求是如何难以实现。程序员可以尽早的开始编码，为需求组提供需求期望值的反馈。

尽管每个下游活动比采用并行开发要花费更长时间，上游团队不用改变它的思路。下游活动可以提前很早开始。这种并行的效果是下游团队比顺序开发完成得早，可能仅在上游活动完成的几天或几星期以后。

这种并行开发在《面向对象项目求生法则》*Surviving Object-Oriented Projects* (Cockburn 1998)一书被形容为淘金热策略。淘金热策略预示好的沟通和重做能力。淘金热策略适用于需求集合可以被预见到接近项目计划中的容错度。很简单，如果下一个团队不得不等待需求的确定，就没有足够的时间做恰当的设计

实际上，很多项目是符合这种模式的。

淘金热类型策略也不是完全免费。需要注意它的三个缺陷：

首先，这是个过分策略，例如，让设计团队走到需求团队的前面（如图4-14所示）。某一天这个团队宣称他们的设计已经稳定了，可以接受评审了。他们仅仅是等待需求团队赶工，生成需求文档。

图 4-14. 保持让上游活动比下游活动更稳定。图中的波浪线体现了需求和设计过程中工件的不稳定性。在良好状况下（如左图所示），虽然两条线同时都在波动，但需求的波动要小于设计的波动。在糟糕的状况下，需求甚至还没有开始确定，设计却已经稳定了。

第二个问题出现在当团队间的沟通渠道不够畅通时。例如，如果团队在地理位置上是隔离的，上游团队很难传递他们的变化信息。只有增加沟通成本，使沟通最终变得更有效，这样在启动下游团队的工作前就知道应等待上游团队，直到他们的工作更加稳定。

第三个问题是对团队的重做能力的估计可能有误差。当一个团队只有很少的空闲时间，或者没有空闲时间。它就必须得到更稳定的输入工件。

16 个 SMALLTALK 编程人员，2 个数据库设计人员。

一个项目有16个Smalltalk编程人员，仅有两个数据库设计人员（DBA）。

在这种状况下，我们可以在需求开始成形时就让Smalltalk编程人员开始工作。同时，我们不启动数据库设计人员的工作，直到对象模型已经通过了设计评审。

只有对象模型通过了包括DBA在内的评审小组的 “稳定性评审”，并确实被评审过，此时DBA才能开始他们的设计工作。

有关什么时候，在什么地方开始并行开发的全面讨论在第156页“原则七，从非瓶颈活动中求效率”。

要理解的一点是稳定性在软件工程流程设计中扮演一个角色。

XP和适应型软件开发（Highsmith 2000）都建议尽可能使用并行开发。因为这两种过程都是时间－市场优先的，需求会在形成最终系统的过程中不断变化。

固定价格的合同常常从使用混合型策略中获得好处：在那些条件下，在进一步设计之前让需求十分稳定是有益的。根据项目混合程度会有不同，有些时候，公司为了竞标可能预先做些设计工作，甚至编码工作。

图 4-15 方法论的角色－可交付成果－里程碑视图

发布软件工程流程

发布软件工程流程包括两个组成部分：图示和文字。

图示

表现软件工程流程设计的方法之一是展示角色如何通过工件来交互（如图4-15所示）。在角色－可交付成果－里程碑视图中，时间从左到右行进，角色用宽的横条表示，工件表示为横条内的直线。表示工件的直线还表现了工件生命期中的边界事件：如它的产生事件（是什么导致了某个人来创建它），它的评审事件（谁来检查它），和它的死亡事件（什么时候它满足了要求，可以停止了。 如果存在这种情况）。

角色－可交付成果－里程碑视图不仅是获取软件工程流程中工件依赖关系的便利方法，它显然也可以用来判断人们什么时候可以休息：

软件工程流程图表可以作为休息时间安排表

有一次我为一个大项目建立了的软件工程流程图表，它非常巨大，占了一面墙。我用角色－可交付成果－里程碑视图来压缩信息，显示组织的软件工程流程的几百个连锁部分的细节。

许多人希望能看到整个的软件工程流程，于是我把它打印出来，挂在一面大墙上。无论什么时候我指向另一个项目角色时，如项目经理或技术文档编写人员，可以看到人们的眼神越过这些部分，当我指向他们自己的角色时，他们的目光才又聚集回来，这非常有趣。

这证明绝大多数人其实只关心软件工程流程中影响他们自己的部分，而不是组织中每个人所做的事。

图示没有表现对组织来说很重要的一些信息，如练习、标准、其他形式的协作。那些信息很难用图来表现，必须以文字形式列出。

软件工程流程内容

在发布形式中，一套软件工程流程用来描述所有被包含角色的技术、活动、会议、质量测量和标准。你可以在*Object-Oriented Methods: Pragmatic Considerations* (Martin 1996) 中和*The OPEN Process Specification* (Graham 1997)中找到例子。Rational统一过程有自己的网址，里面有上千个网页。

软件工程流程的内容是非常多的。在某个级别上没有办法摆脱它的复杂性，甚至一个很精简的方法，只有四个角色，每个角色四个工件，每个工件三个里程碑就有68 (4 + 16 + 48)个要描述的连锁部分，这还不包括任何技术讨论。甚至就拿XP来说，它最初仅有200页（Beck 1999），现在由于为每个部分增加了额外的指南（Jeffries 2000, Beck 2000, Auer 2001, Newkirk 2001），已经扩展到1,000页了。

大多数组织不为他们的每个新雇员出版上千页的关于他们的软件工程流程的文字描述，有两个原因：

第一是Jim Highsmith对差别的精辟概括，“相对而言，文档难以理解。”

真正的软件工程流程是在团队成员的头脑中的，通过他们的行为和交谈的习惯体现。用大段文档来描述方法的根本不能提供同样的理解，所获得的理解也不是必须先有文档。理解很快能被获得，因为新员工通过正常工作很快就可以获得经验。

第二个原因是，组织的需求总在不断变化，保证上千页的文档和项目团队的需要同步，即使可能，也很难实现。

当出现了新技术，团队必须发明新的工作方法来处理他们，那些方法不可能被预先写下来。

组织需要方法来发展不再有效的软件工程流程，以便将一个团队的好习惯传递给另一个团队。当你继续阅读这本书，你将学习到如何进行这项工作。

减小软件工程流程的体积

有几种降低已发布的软件工程流程的物理大小的方法。

提供工件样例

提供工件样例而不是提供模板。象前面所讨论到的，利用人们善于从具体示例中学习并工作的特点。

合理收集不同工件的优秀的样例：项目计划、风险列表、用例、类图、测试用例、函数头、代码段等等。

把它们放到局域网中，鼓励人们复制和修改它们。取代编写用户界面设计标准的文档，将一个友好的界面例子放上去，让人们能够复制并在它的基础上再设计。可能需要为这些样例增加注释，以说明哪些部分是重要的。

这些事情可以减少建立标准所需的工作量，以及为人们使用它们减少阻碍。

少数提到活动和它们的标准的书籍之一是*Developing Object-Oriented Software* (OOTC 1997)，它是九十年代末，由IBM的面向对象技术中心为IBM准备的，后来它被公开了。

删除技术指南

不再试图通过在软件工程流程文档中提供详细的技术描述来教授技术，而是在软件工程流程中简单地推荐应使用的技术，包括任何已知的书籍和教授这些技术的课程。

使用中的技术包括默许的知识。让人们从专家那里学习，就像学徒一样，或通过手把手教的课程，这样人们可以边学习，边实践。

如果可能，让人们在加入项目之前就掌握所需的技术，而不是在项目过程中把教授技术做为软件工程流程的一部分。技术就成为人们所掌握的技能，他们就可以简单运用这些技术完成自己的工作。

根据角色来组织文字说明

编写一个低精度的但又能描述每个角色、工件、里程碑的说明是可能的，将描述与角色－可交付成果－里程碑图表相关联。示例角色描述如下表所示：

角色描述示例	
整体负责人	具有支持和监控项目开发过程的能力。在整体项目级别上负责控制范围、优先级、资金。
跨团队负责人	负责多个团队的开发过程，衡量这些团队的成果，建立团队间的优先级，为不同的团队分配资源。
团队负责人	负责一个团队的开发过程和方向

角色描述示例 （cont'd）	
开发人员	一群掌握技术的人，他们的主要工作是开发产品。包括UI、商业类、基础模块或数据。
文档编写人员	通过媒介如白皮书、共享驱动、intranet 或internet来发布不同主题的技术信息。
技术支持人员	一个或多个人，他们的工作是沟通和调节不同领域的技术人员、客户代表和生产产品的人之间的关系
外部测试人员	一个或多个人，他们的工作是在开发组之外进行与质量相关的功能测试。
维护人员	在产品发布后，对产品进行必要修改的人员。

对于工件，你需要记录谁编写了它们，谁应该阅读它们，它们包含了哪些内容。一个完整的版本应包含一个样例，注明允许的容错性，应达到的里程碑。下面是一个简单的描述。

整体项目计划	
编写者	跨团队负责人
阅读者	整体负责人、团队负责人、新来者
包含内容	跨越所有团队，计划接下来的几次发布，不同团队间的依赖关系，包括内容、计划的开发时间。

依赖关系表	
编写者	团队负责人
阅读者	其他团队负责人、跨团队负责人
包含内容	本团队需要其他团队提供哪些工件，每项所需的数据。可能包括延期计划，以防某些工件不能按时交付。
团队状态表	
编写者	团队负责人
阅读者	跨团队负责人、开发人员
包含内容	团队的当前状态：正在进行的应完成工件列表，下一个里程碑，是什么拖延了进程，每个工件的稳定性级别。

对于评审里程碑，记录谁被评审，谁来做评审，评审的结果。如下所示：

发布建议评审	
评审者	应用程序团队负责人、跨团队负责人和整体负责人
意图	基本的范围评审
评审物	用例摘要，用例，活动者，外部系统描述，开发计划
结果	修改范围、优先级、日期，可能会对活动者列表和外部系统进行修正

应用程序设计评审	
评审者	团队负责人、相关的跨团队负责人，跨团队的顾问，业务专家
意图	检查质量、正确性和应用程序设计的一致性
评审物	用例、活动者、域类图、界面流程、界面设计、类表（如果存在）和交互图（如果存在）
结果	对域模型的实际修正，界面细节。基于质量或一致性方面的考虑所提出的对改善UI或应用程序设计的建议或需求，

从这些简单的段落可以看出，软件工程流程可以根据角色来总结（如下面的两个例子所示）。软件工程流程中的填写表格，根据角色总结，是对每个人的一个检查列表，人们应该符合每个表格的要求，并在个人工作空间中支持它们。这些表格没有什么特别（当第一次读时），它的作用是提醒团队成员那些他们已经知道的东西。

这是一个针对开发人员的稍加删节的例子：

设计人员－开发人员	
编写	每周状态表 源代码 相关说明（等等）
阅读	活动者描述 UI风格指南（等等）
评审	应用程序设计评审（等等）
发布	应用程序配置 测试用例（等等）
声明	UI稳定性

可以看出这不是个抑制创造力的软件工程流程。对于一个新来者，它是个大纲列表，指出他应该如何参与到团队中。对于正在团队中工作的开发者，它是个提醒。

使用微缩流程

已发布的软件工程流程不能表现将内部理解形成默认知识。它不能表现软件工程流程的生命期，它包括很多伴随团队工作的小活动。人们需要看到或亲身体验软件工程流程。

当前我最喜欢的表现软件工程流程的方法是通过一种我称为微缩流程的技术。

在微缩流程中，参与者在一个很短的期间扮演流程中的一个或两个角色。

在我面试的一个团队中，新雇员被要求在第一周开发一个软件的一小部分，所有的需求都已经交付。这种一周练习的意图是给新雇员介绍团队成员、角色、标准、和公司中各种东西的位置。

最近，Peter Merel为极限编程发明了一种一小时微缩流程，称它为极限一小时。极限一小时的目的是给人们一个XP的内部实践机会，以便他们能通过这个基础的仿真练习来讨论掌握XP的概念。

在极限一小时内，一些人被指定为“客户”。在前十分钟，他们为开发者提供需求，并一直在XP计划会议中工作。

在接下来的20分钟，开发者概略地进行设计，并以很高的透明度对其进行测试。第一次迭代的总时间为30分钟。

在后30分钟里，整个循环被重复，这样经历了XP的两个循环仅仅用了60分钟。

通常，极限一小时的主持者会选择一个有趣的任务，如设计一个捕鱼设备，它能够保证鱼在交到厨房的时候还是活的，也能保证啤酒在全天时间里一直是凉的。（当然，在迭代过程中他们不得不缩小范围！）

我们使用90分钟的微缩流程来帮助一个有50个人的公司的团队来体验我们建议的新的开发流程（你将注意到这个微缩流程经验与第57页所描述的信息是多么相似）。

在这个案例中，我们将主要兴趣放在表现我们希望人们使用的编程和测试规则上。因此我们不能使用基于画图的问题，如捕鱼，而不得不选择一个实际编程问题，它可以生成一个WEB应用的运行、测试代码。

微缩过程体验

我们想在90分钟内演示两个完整的迭代过程。

我们想让人们讨论需求，然后创建和测试代码，使用正式的五层体系架构，执行数据库，配置管理系统，正式的Web风格的表单，充分的自动化回归测试。

因此我们不得不选择一个简洁的应用。我们选择构建一个简单的上－下计数器，可以从0到20计数，可以被复位为0。该计数器应使用Web浏览器接口，并在正式的公司数据库中保存自己的值。

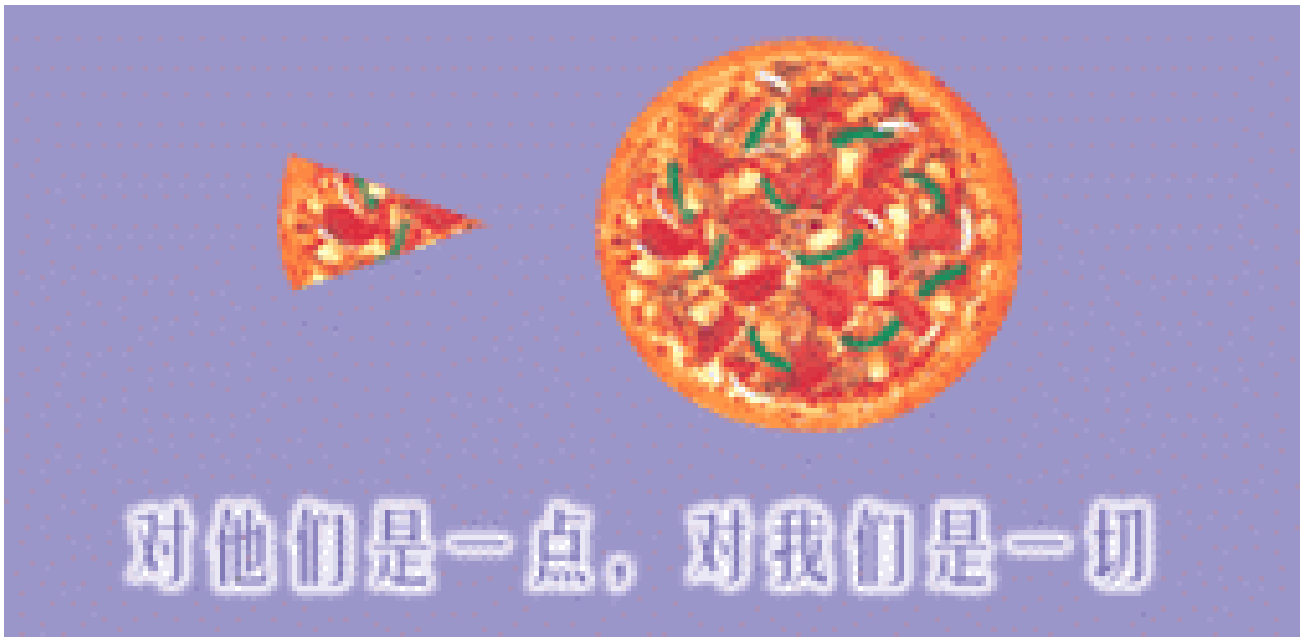
为了让每个迭代超过45分钟的约束条件，我们在设计动作时有一点扩展。市场分析人员被告知要请求超过团队在30分钟编码内能交付的需求（“能为我们提供一个图形化的钟表刻度盘样式的，有三种颜色的计数器吗？”）我们这样做是为了让观众体验在实际生活中遇到这样的问题时，如何进行范围谈判。

我们也演练了程序员在完成第一次迭代时的竞标结果是多少，在迭代中期他们应如何缩小范围，以便观众能在活动中看到这些。

这些片断脚本的重点是给整个公司一个我们想要建立什么的视图，我们想建立在项目运行过程中，正常范围谈判的社会惯例。

我们将实际编程留做活的活动，尽管团队知道任务，他们仍旧不得不把它们实时打印出来，作为体验的一部分。从头到尾看完了包括打印工作的观众，意识到了工作总量，即使它不是一个价值很高的系统。

无论使用什么形式的微缩流程，要计划不断地重演它，强迫你的团队形成社会习惯。许多习惯在文档中是找不到的，如刚刚描述的范围谈判规则，但在演练中能部分捕捉到。



>>>>