2002－7（第15期）

Bruce Powel Douglass

X-Programmer 非程序员

软 件 以 用 为 本

主编：davidqql

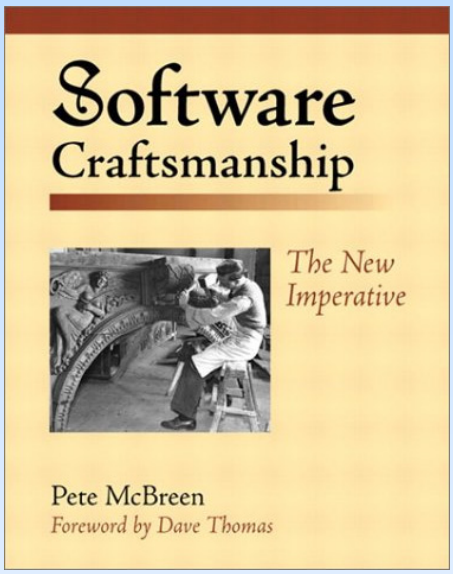
审稿：davidqql，think

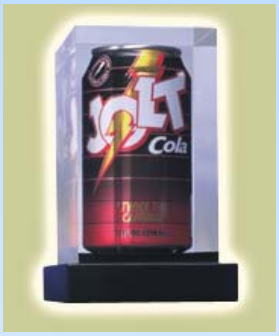
投稿：editor@umlchina.com

反馈：think@umlchina.com

http://www.umlchina.com/

《软件工艺》





2002 Jolt Awards 获奖书籍

Pete McBreen

翻译：UMLChina 翻译组 Jill

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。

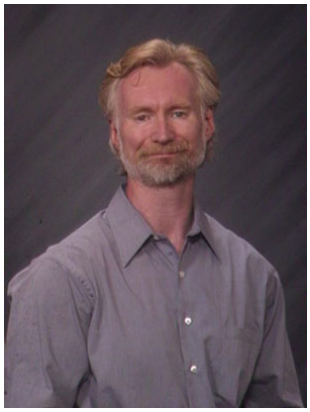
中文译本即将发行！

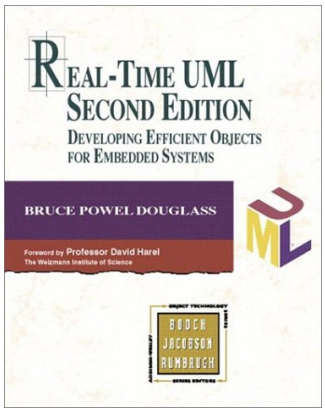
Bruce Powel Douglass： 实时系统和 UML

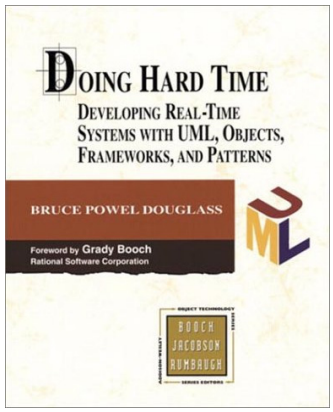
查看评论

编注：

北京时间 2002 年 6 月 18 日上午 9: 00-11: 00，Bruce Powel Douglass 先生作客 UMLCHINA 讨论组的聊天室。Bruce Powel Douglass 在设计安全性要求严苛的即时应用系统设计领域具有 20 多年的经验。著有："Real-Time UML: Efficient Objects for Embedded Systems"和"Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns"。Bruce Powel Douglass 是 OMG 实时分析设计工作组的主席，现为 I-Logix 公司首席科学家。以下是交流实录。由 Windy.J 翻译。原文链接。







bydpdwz：对嵌入式系统来说，需要所有的 UML 图吗？

brucedouglass: UML 实际上只有四种不同的图－结构，结构图有类、对象、包、结点和组件，他们是同一类图。你可以用三种图来建立嵌入式系统－类图、状态图、顺序图。我不认为我们这么讨论有助于对系统建模……因为如果要描述建模方法，除了设计对象之外还需要做更多的事。几个月以前我在 Electronic Design 杂志上发表了一篇名为“UML Lite”的论文，里面讨论了 UML 的最小集合。

umlsz：您认为 Rational's Rose Realtime 怎么样？

brucedouglass: Rational RoseRT 不错，但它不是纯 UML，它基于 UML 和 ROOM。 ROOM 跟 UML 不同，所以要使用 RoseRT，就要同时学习 UML 和 ROOM，而 RoseRT 中只有 ROOM 的部分是有效的，显然这没有必要，因为 UML 本身就够了，例如 I-Logix 的 Rhapsody 工具就是这样。

supergaosong：我们在嵌入式系统中应用 UML 最重要的理由是什么？

brucedouglass: 问得很好 – 最重要的理由...我们为系统建模最主要的的两个理由是...1. 能够从不同的角度来看待系统－结构，行为，功能（需求）； 2.能够在不同抽象程度上考虑复杂的系统，只有源代码还不够，源代码是非常细化的内部结构视图，不能用来建造复杂的系统。

nasapn：ROOM 是什么意思？

brucedouglass: ROOM 是一种方法，叫 Realtime Object Oriented Modeling，来自 1994 年 Gulekson, Selic 和 Ward 的一本书。从这种方法产生了一个叫 Objecttime 的工具，这是一种单一源码的内部语言，Rational 收购了它以后，和 Rose 结合就形成了 Rose RT。

umlsz：少有人用，但它很有意思。

brucedouglass: 你说 ROOM？是的，大多数人喜欢 UML。UML 是标准化的，同时它也更加完整和容易使用。有趣的是，ROOM 自己甚至不讨论时间 ;-)

tinyl：用例（use case）是什么样的？

brucedouglass: 你是说从图形上还是从语义上？ (什么意思？)

tinyl：UML 中的注记（notation ）非常重要对吗？

brucedouglass: 不，我认为注记在 UML 中并不是最重要的，最重要的应该是语义。当我们在 UML 中说到"class"或"event"的时候，它的含义是相当精确的；注记很有用，但语义更重要。

tinyl：您说的语义是什么意思？

brucedouglass: 语义-说话的含义，没有语义，注记就成了没有意义的图形。

umlsz: 您能为实时软件系统推荐一些好的框架性工具吗？

brucedouglass: 例如 I-Logix 的 Rhapsody 就是一个适用于实时系统和嵌入式系统的 UML 工具。

bydpdwz：在嵌入式系统中应用 UML 的典型方式是怎样的？

brucedouglass: UML 非常适合用在实时系统中。首先，使用 UML...其次，细化那些用例-也就是对需求进行准确地陈述，经常会用到状态图，活动图，和/或顺序图。 理解了需求以后，开始识别协作完成这些用例的对象。设计

的时候，我们精化对象模型，增加设计细节。对象是类的实例，对象在运行时存在，而类在设计时存在。好，前面我说了 3 个必需的视图 – 系统结构 (类图), 行为 (状态图，活动图和顺序图)，还有需求，也就是用例。

herman：实时系统和我们常用的 PC 应用差别很大，那么如果（在实时系统中）应用 UML，我们首先会遇到什么问题？

brucedouglass: 我们先来看 PC 应用和嵌入式系统的区别。主要的区别是嵌入式系统更注重“服务质量”(Quality of Service， QoS)，例如最差性能，平均性能，总处理能力，带宽，内存大小，等等。同时我们也进行更多的底层设备驱动和系统编程。

bydpdwz：类图是最终的结果吗？

brucedouglass: 每个视图都很重要。只有类图，你一定会问：“这样到底对不对？”

tinyl：行为 (状态图，活动图和顺序图)是什么意思？

brucedouglass: 只有执行系统的行为（由系统结构图和行为图定义）以后，才会得到答案。行为 – 事物随着时间进行的改变。状态图和活动图描述个别类型的行为，例如- 用例，类，子系统，组件；顺序图描述一组类型的协作；我们使用协作图 – 许多类潜在的结构，来定义对象的协作，这些协作实现用例，或完成一个用例。

umlsz：您更喜欢用 CORBA 还是 COM？

brucedouglass: 在 Windows 里用 COM 就可以了，不过为了在不同操作系统之间的协同工作，CORBA 会更合适。

bydpdwz：您能介绍一下嵌入式系统里面有关协作的细节吗？

brucedouglass: 协作是一组对象来实现一个用例...例如，一个医疗系统中有一个“输送药品”的用例...这个协作就必须包括这些对象： Vaporizer，ventilator，knob, button, text view, icon view, alarm, alarm manager 等等。这些对象一起工作，提供一种安全有效的方法来输送药品。

herman：“输送药品”？ 在制药厂吗？

tinyl：把药品输送到其他商店？

brucedouglass: 不，是向医院手术台上的病人输送药品。

pega：什么（工具）对 QoS 支持最好？

brucedouglass: UML 就对 QoS 支持得非常好。

umlsz：很少有用于实时系统的中间件，为什么？

brucedouglass: 很多人在实时系统中使用中间件...我在 NASA 的最后一个项目用了 Ace/Tao ORB (CORBA)。这取决于你的系统。

umlsz：我觉得 CORBA 规模太大，我们可以剪裁它吗？

brucedouglass: 10 年以前我写过一个叫 CORBAWE (CORBA-Wanna Be)的微型 CORBA 实现，它非常小；对，商用的 ORB 对微处理器来说确实规模比较大。那样的话你可以试试 JINI。

bydpdwz：嵌入式系统都需要 RTOS 吗？

brucedouglass: 嵌入式系统的规模从 4 到 8 位的微处理器到太空船飞行控制系统和核电站管理系统。对 RTOS 来说，它取决于系统规模。在 8 位的系统中我可能不会使用，但规模更大一点的话，我就会用它。它能帮我节约关于内存管理部分和并发管理部分的时间。

tinyl：JINI 是什么？

brucedouglass: JINI 是分布式系统中一种内嵌的 Java 框架。

xiao_ying_ma：UML RT 可被用到并发系统中吗？

brucedouglass: UML RT 是 Rational 公司的一个商标，它不是 UML。UML 能非常有效地用在实时系统中，我把它用在宇宙飞船、机器人技术等方面，还在医疗系统中建造了心脏起搏器。

Bydpdwz，idlecrook：C++呢？ C++会占用更多的资源，内存碎片回收将会导致运行时数据的丢失，对吗？如果不能用 C++，那我们怎样才能应用 UML？

brucedouglass: C++的效率很高，跟 C 的差别不大，使用 C++的原则是，如果用它，就要付出用它的代价。内存碎片回收是个大问题，因为它给系统增加了非常大的不确定性，很多 RT 都难以解决。你可以用 C, C++, Java, Ada... 甚至汇编语言来实现 UML 模型。我曾经用 6502 汇编语言为一个心脏起搏器写过一个多任务操作系统，而它来自一个 OO 模型。

bydpdwz：主要使用 C 的结构吗？

brucedouglass: C 使用结构来表示一个类，对的。

herman：嵌入式系统和实时系统也不同，我曾经为发电所工作，那里大部分的控制系统都是实时系统。

brucedouglass：实时系统和嵌入式系统相当不同，但有着相关的概念。

idlecrook：我怎样选择开发语言？

brucedouglass：语言的选择有很多因素...目标处理器的支持，速度，熟悉程度，实时性，可预测性等等。

tonyhu：Rhapsody 怎样帮助我们节约时间？

brucedouglass：Rhapsody 可以在很多方面节约大量时间...你要知道，你必须经常能够回答这样的问题："这样做到底对不对？" Rhapsody 可以产生代码，并对设计进行有效性检验，甚至在设计还没有完成的情况下 。它既可以在目标硬件上运行，也可以在你的桌面运行。

onerest：如果用 C 的结构来表示类，那是不是会丢失一些象继承这样的东西？

brucedouglass：如果想要的话，可以用 C 实现继承。是有点复杂，不过别忘了以前 C++是用 C 的预编译器产生 C 的代码来实现的。

windy.j：您能详细介绍一下 UML 是怎样对 QoS 进行支持的吗？

brucedouglass：QoS...大多数的 QoS 需求我们认为它们是"无建设性的"，因为它们不影响代码如何编写，但它们影响的是代码编写得是不是正确。 QoS 的具体表述通常放在约束里，UML 里的约束是一个"用户自定义的数量指标"，应用到一个或多个的模型元素。例如这样： {worst case execution time = 20 ms +/- 2 ms }。

umlsz：NASA 使用 VxWorks 的时候是否觉得安全？

brucedouglass：NASA 正在一些项目里使用 VxWorks。

herman：效率和安全性方面必须互相权衡？：）

brucedouglass：所有的设计都是一种权衡——效率，平均经常成本(recurring cost)，开发成本，安全性，可靠性，等等。设计者的工作就是根据它们对系统的重要性找到一个优化所有这些制约因素的解决方案。

umlsz：怎样评估一个 RTOS 的安全性？

brucedouglass：这是个大问题(RTOS 的安全性)...但简要的回答是，不能……在"Doing Hard Time"的第三章中讨论了这个问题。

bydpdwz：您认为在嵌入式系统中使用 RTOS 是一种趋势吗？

brucedouglass：更明确地说，人们会购买商业的 RTOS，而不会自己动手开发，一定是这样的。

wys2051：什么是“Doing Hard Time”？

brucedouglass：“Doing Hard Time”我 1999 年出版的一本书，关于实时系统和嵌入式系统开发。

onerest：哪有卖的？

brucedouglass：你可以从美国订购吗？它由 Addison-Wesley 出版。www.awl.com。Hard real-time 的意思是如果在某个特定的时刻出现问题，它就是一个系统故障。Soft real-time 更关心平均的性能，或平均反应时间，那么可能经常性的丢失一些反应是可以接受的。

skyship：您认为 Rational Rose Real-Time 是开发 RTOS 的一个好工具吗

brucedouglass：就个人而言，我并不是它的爱好者，不过有很多人用它来开发系统。

umlsz：如果有了高速（fast）的 CPU，一个 soft-time OS 是否可以用于 hard-time 系统？

brucedouglass：Fast 跟 hard 不同，这是一个常见的误解。

bydpdwz：如果不用考虑资源的话，在嵌入式系统中使用 RTOS 会不会更好？

brucedouglass：实际上我自己经常使用 RTOS，不过经常是用于比单个微处理器规模大一些的系统。可以用同样的方法开发 MIS 系统或嵌入式系统...对后者只要多多关注 QoS 就可以了。

umlsz：那么分布式系统呢？

brucedouglass：有很多方法对分布式系统建模... 主要的分支是不对称的和对称的两种。

bydpdwz：您能对不同的 RTOS 发表一些意见吗？

brucedouglass：我用过一些 RTOS，喜欢用 VxWorks 和 pSOS。

umlsz：平等对待或者是一些作为主处理设备？有好的模式吗？我指的是多处理器。

brucedouglass：有很多方式实现分布式－共享内存，publish-subscribe（出版－订阅者）, Broker,以及 Proxy 都是好的模式。你可以采用 master-slave（主从）或 peer-to-peer（对等）的方式。

wys2051：可以介绍一下您的书"Real-time UML"吗?

brucedouglass: 9 月份我将出一本新书，"Real-Time Design Patterns" ，这本书讨论了用 UML 对 RT 系统的架构建模的许多方法。

bydpdwz：您认为 UML 覆盖了嵌入式系统设计的所有过程吗?

brucedouglass: 我认识的大多数人开发嵌入式系统的时候，在整个项目中都使用 UML。

longzhifang：您认为 LynxOS 怎么样?

brucedouglass: 我没有用过 LynxOS。

onerest：您认为进行分布式系统建模时 UML 是最好的选择吗?

brucedouglass: 是的，我认为 UML 是分布式系统建模的最好方法。

umlsz：您有计划将您的书翻译成中文吗?

brucedouglass: RT UML 已经被翻译成日文，我听说"Doing Hard Time" 将被译成中文，但我不知道什么时候才开始。

bydpdwz：您能给我一些建议来说服我的老板在嵌入式系统中使用 UML 吗? ：-)

brucedouglass: 要进行技术上的改变，你必须进行商业上的考虑：收益和成本的考虑。在我的经验中，适当应用 UML 可以在更短的时间里做更多的事，同时可以降低缺陷。这是已经是很大的收益了。我工作过的一家公司不得不学习 UML，C++,还要学习一种建模工具的使用(Rhapsody)。我教他们 UML，他们自己进行 C++培训，我也在整个项目中进行指导，每个月几天时间。最后，他们报告说，跟他们采用原来的老方法相比，（用这种方法）开发这个新系统少用了 30%的时间。甚至包括了所有的学习周期。你的具体情况当然可能不同，但我见过的人都获得了生产力和质量上的进步。现在我认为，刚开始转向 UML 的时候很可能会失败，所以你要很小心。可执行的模型很重要，而且一个可执行的 UML 工具也很重要。还有，采用一个迭代的生命周期可以将测试提前，所以可以在缺陷变得不可收拾之前找到和修复它们。

onerest：Master-slave，跟 Client-Server 有什么关系吗?

brucedouglass: Master-slave 跟 client-server 不一定有关系，但它可能用来实现一个 client-server 结构。

bydpdwz: Rational Rose?

brucedouglass: Rational Rose 不错，但它不是一个可执行的工具-也就是说你不能在 Rose 里执行或测试 UML 模型。

onerest：您觉得 AM(Agile Modeling)怎么样？

brucedouglass: 我认为 AM 非常棒，但是还不够完整，它忽视了大时间尺度的问题。对待项目的时候，我使用 3 种时间尺度来衡量：macro（大型）, micro（微型的）和 nano（纤型的）... Macro 是项目的整个过程- 一般是 1 到 *年。Micro 是迭代周期 –一般 4-6 周。Nano 在桌面的执行时间.. 可能几分钟就执行你的模型，因为你不断在修改它……。nano 周期正是 AM 关心的，当然它很好，但是还不够。

umlsz：您认为下一代的 RT OS 将会是什么样的？

brucedouglass: 提供更多的服务，可伸缩性更强，如果需要，可以让人们可以随意包括象中间件这样的服务。它们（RTOS）也会增加更多的 IDE。

bydpdwz：您能为嵌入式系统推荐一种 UML 工具吗？

brucedouglass: Well,我在 I-Logix 为 Rhapsody 工作，我推荐它。对 Rhapsody 你想了解些什么？

wys2051：它有些什么特点？

brucedouglass: Rhapsody 是一种 UML 适用工具，它可以为 C, C++, Java or Ada...生成可分发的高质量代码；它可以让你在 UML 的级别执行你建造好的模型；它有一个用来检查和标识模型问题的调试系统；它的测试向导可以让你以半自动的方式应用测试包对系统进行测试。

wys2051：（Rhapsody）价格怎样？

brucedouglass: 我不知道它在亚洲的价格。到 www.ilogix.com，会有销售接待员跟你联系，他们会告诉你有关的价格。

wys2051：有演示版吗？

brucedouglass: 有，不过下载起来很大。

tonyhu：我们在开发一种视频监控设备，MAP-CA chipset + VxWorks + RTP + HTTP Server + H263/MPEG4 + 模型识别，用 HTTP 服务器进行管理；用这些嵌入式设备，我们怎样应用 UML 来设计 B/S 架构？ Vxworks 的 http 服务器是一个正确的选择吗？还是使用 apache 更好？

brucedouglass: 你的 B/S 是什么意思？OK。用来进行视频处理吗？

tonyhu：设备中的嵌入式的 http 服务器。是的，视频处理。http 服务器用来进行设备管理。

brucedouglass：我知道有些公司使用 UML (还有 Rhapsody)来建立高速电信网络，建立电视和视频处理系统。事实上 Rhapsody 有个"Wedify"界面，它可以将 web 服务器插入你的应用，并建造网页，用来提供系统的观察视图和对系统进行控制。系统只有 3 种视图，可能根据系统的规模，你想要更多的视图。如果不使用 Rhapsody，你当然也可以手工来完成。

tonyhu：如果我使用 rhaposody 的话，是否可以不用购买 vxworks 的 http 服务器（它很贵)？

brucedouglass：Hmmmm 我不能肯定你是不是需要 VXWorks 的 http server；- 我得写信给 support@ilogix.com，问问他们。

blue_monkey：怎样把 SA 变成 UML？

brucedouglass：把 SA 变成 UML ...这是一种头脑的转变...有两个方法..自己学习，或者向别人请教。这是一个可以用金钱购买时间的地方。参加培训并学习 UML，在第一个项目得到一些咨询，或者找一个导师“直接起飞”－这种方法是最有效的 ...在时间上，但它需要花费金钱；你也可以自己学习，但需要非常长的时间才能掌握。所以看你更注重的是什么。你的意思是有些公司可以提供培训？是的 – 很多，我是自己学得多一点。

bydpdwz 对 blue_monkey 说：我想你可以使用 SA 来建立 use case。

brucedouglass：我也认为你可以建立一个 use case 模型，然后用对象或功能来实现它。本质上 use cases 并没有特别的面向对象特点。

tonyhu：Wedify？您有 CD 播放器的例子吗？

brucedouglass：我不太清楚哪些应用配置了 Webify 工具包。

tonyhu：对很多嵌入式系统来说，例如我的视频处理系统， use case 没有用吗？

brucedouglass：Use case 提供一种组织需求的方式， 我从来没有开发过一个系统说那个系统需求是不重要的。

bydpdwz：我怎样来判断嵌入式系统中的参与者？

brucedouglass：参与者是系统之外的对象，并跟你要建模的系统进行交互。例如一个空中交通管制系统会有下面的参与者：控制者(一个人)，各种飞行器，国家领空管理系统等等。

bydpdwz：例如系统定时器？

brucedouglass：系统定时器是设计的一部分，我不会把它们当成系统级的参与者。汽车发动机控制器将有下面的参与者：司机，轮胎，等等；控制器内部是设计的一部分而不是参与者。

smilemac：对，它应该是系统的一个属性。

brucedouglass：不是“属性”，而是系统内部的一个对象。

windy.j：请问实时系统里的 Use Case 有什么特别吗？

brucedouglass：实时系统里不需要特别的 Use Case，如果你感兴趣的话，在 I-Logix 网站上我有一篇论文讲了这个问题。你只要关注 Qos 指标 – 例如端对端的性能等等。

smilemac：我的意思是时间应该成为系统的一个维度。

brucedouglass：你可以把时间放到约束里，并应用于整个 Use Case，或应用到它们内部（用状态图或顺序图建模）...在实时系统里你当然要为你的实时需求建模。实时的意思是“实时可预见的”，所以你建模的时候一定要把时间考虑进去。

bydpdwz：但有些行为需要定时器触发。例如，我有几个探测器。系统定时器定期检查它们。

brucedouglass：是的，但是定时器在你的系统内部，所以它不是参与者，而是你设计的一部分。

tonyhu：“Doing hard time”和“Real Time UML”太厚了，您能否提供一些快速阅读建议？

brucedouglass：太厚了，哈？ ;-)我知道最薄的书是 Martin Fowler 的"UML Distilled"...一本非常好的书，但完全没有讲到实时系统，是一些非常基础的介绍，同时你还能从 www.ilogix.com 下载一些论文。

tonyhu：您是否认为状态机是嵌入式系统中最重要的东西？

brucedouglass：我认为状态机通常是非常重要的，但我在前面说过，一共只有三个关键的视图，而且不是所有的系统都有状态行为。

gabrial：您对 RT java 怎么看？您认为它在近几年会实现吗？

brucedouglass：RT Java 是很有意思的小东西，但它不会应用太广。它有两个规范： Sun 的规范和 J Consortium 规范。对实时系统来说，两个规范都有问题，我认为 J Consortium 的规范更好一点。Sun 的规范仍然存在不确定性

的问题，而且相当严重。同时 Java 占用空间相当大，而且比编译过的 C++慢，这也是一个问题。我喜欢 Java，但虚拟机方法在实时系统里不太合适。

bydpdwz：我读过一篇论文，关于 UML 进行录音机的设计，在那篇论文里，把时间当成了一个参与者。

brucedouglass：很抱歉，我认为把时间当成参与者是没有意义的。;-)。我也见过，但我非常不赞成那么做。(倒不是说我该有个看法或别的;-))

bydpdwz：那么我们怎么判断 UML 设计的质量？

brucedouglass：那，你怎么判断其他任何设计的质量？...正确性，鲁棒性，满足需求，优化系统重要的 QoS 指标。在质量方面，既然我已经想过这个问题，我会把简单性和清晰性增加到清单上。

gabrial：是的，尽管 Java 在低端设备里应用很广，但它的性能是它进入嵌入式领域的障碍，尤其是在控制领域。

brucedouglass：赞成，我见过附加到控制系统的 Java 界面，而控制系统是用 C 或 C++实现的。

tonyhu：用工程的眼光来看，建造嵌入式系统的时候我们可以采取哪些过程？

brucedouglass：well, 工程需要工程师、市场人员、测试人员等等之间一个协作的过程。有很多不同的过程，我推荐 ROPES，它是我创造的，所以我会有些偏心 ;-)

gabrial：我好像听说一些年轻人在钻研 RT Java 的问题，如果他们成功了，将会令人大吃一惊。

brucedouglass：是啊，人们已经在承诺 30 年内要解决内存碎片回收中的不确定性问题，不过现在还没有解决。Sun 承诺 RT Java 将包括 RT 内存碎片回收机制，但是正式发布的时候，内存碎片回收就成了一个提到实时特征时的“用户问题”。

jeersoft：我是一个初学者，怎样才能学好 UML？

brucedouglass：有很多好书，可以从看书或者看论文开始。

gabrial：说实在的，我喜欢在嵌入式系统中使用面向对象技术，但 C++ 在开发和分发的时候都很困难。我相信 Java 和 C#将是这个领域最后的获胜者。

brucedouglass：有无数成功的项目使用 C++，当然，如果你喜欢一种不同的开发语言，它们也会很好用，包括 C 和汇编语言。 C#不是用在嵌入式领域的。

smilemac：RT 应用可以正确运行在分时 OS 例如 NT 或 Linux 下吗？

brucedouglass: RT 应用可以运行在 Linux (TimeSys 有一个实时版本)和 Windows 上，它仅仅取决于你的实时意味着什么。Hard RT？不是在普通的 NT 或 Lunix 系统下。不是 hard real-time 系统，那样的系统有一些需要处理的不确定性问题。嵌入式系统？当然，对它们来说问题比较容易。我实现过一个麻醉医疗系统，结合了 VXWorks 和 Windows NT，尽管 VxWorks 部分完成了全部的实时任务，在 NT 下还是有些不确定性问题。

bydpdwz：您能说说在 UML 里怎样对待数据内存吗？

brucedouglass: 数据作为属性存放在对象中，数据集合就是对象集合。如果需要存储数据，可以利用面向对象数据库。

smilemac：但我怎样证明在分时 OS 中，实时的需求始终可以得到满足呢？

brucedouglass: 在 hard real-time 系统中，速度不是问题，它提供良好的整体处理能力（如果系统是 compute-bound 而不是 IO-bound IO 的），但你也不希望一个飞行器“偶尔”坠毁吧。有一些数学上的技术，例如 RMA (Rate Monotonic Analysis)来证明你能始终达到严格的实时需求。

tonyhu：学校的经验重要还是公司的经验重要？

brucedouglass: 我想工作经历更重要，学校教你字母表，但工作教你怎样写出故事。

gabrial：使用 Java 和 C#，（嵌入式系统的）基础架构将会改变。尤其是 C#，整个 Windows 系统将用.NET 重构，那样的话在语言级是安全的。我相信那将是到现在为止最安全的一个操作系统。

brucedouglass: 对你关于 Java 和 C#的看法，我不是很肯定。基础架构问题，当然，但.NET 不是一种实时环境。如果操作系统在处理虚拟内存时会锁定，那么，这个操作系统就难以应用在实时系统中。回答下面这个问题就可以证明：“我愿意乘坐一个每小时 600 英里的速度，在 35,000 英尺高空飞翔，并用.NET 引导的飞行器吗？”？我肯定不愿意！

herman：对，我想也是这样。：）

brucedouglass: Ctrl-Alt-Del 只能帮你做到那么多;-)如果那样的话，最后我听到的将会是飞行员将通过广播询问：“机上有程序员吗？？？？”

bydpdwz：在嵌入式系统中我怎样使用状态图？

brucedouglass：实时的意思是什么？它指的是“实时可预测”。状态图是对象的行为描述。例如，一个传感器类有下列的状态：“空闲”，“接收中”，“有数据”，“正在配置”，等等。状态转移代表这个对象对输入事件的反应，例如“数据收到”或“配置系统”。有很多方法可以从状态图生成代码...级联的 CASE 语句，一组事件处理操作，状态模式，状态表模式等。空闲是一个状态——对象的一种“存在的情况”。

tonyhu：我觉得在面向对象编程中很难使用 C 语言。

brucedouglass：（可是）从 UML 模型生成 C 代码很常见，人们一向这么做。C 语言很适合面向对象实现，关于这个主题有很多书里都有描述。

tonyhu：从 UML 模型生成代码是不是越来越流行了？

brucedouglass：生成代码越来越流行是因为它的诸多好处。我希望它有用。

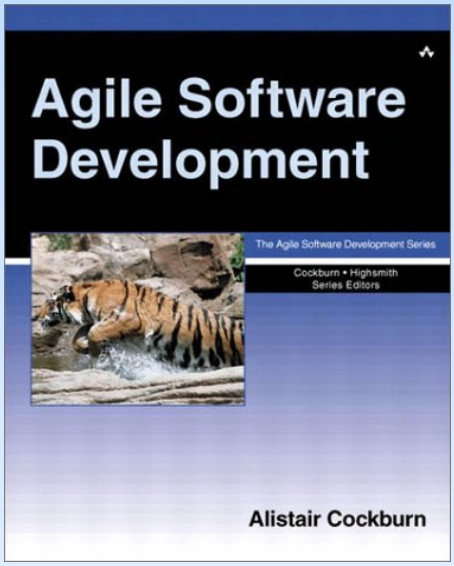
gabrial：我对 Java 和 C#的看法是，发生在 PC 上的变化将一样出现在嵌入式领域和实时领域。在早期 PC 时代，我们必须满足非常苛刻的硬件资源，必须仔细考虑怎样用它们来解决问题，副作用是代码变得非常难懂，而且容易出错。无论如何，现在大多数的 PC 开发人员不用考虑这么详细了。

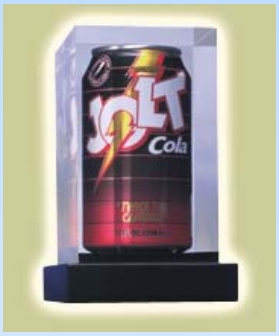
brucedouglass：我同意，嵌入式系统将随着 CPU 的发展变得越来越强大。不管怎么样，作为设计人员我们必须小心谨慎，小心谨慎也不会影响系统的功能。

gabrial：我觉得对实时和嵌入式系统来说，（变化）很快就会发生了。

brucedouglass：变化一直在发生，看看我们的手提电话就知道了。

《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！

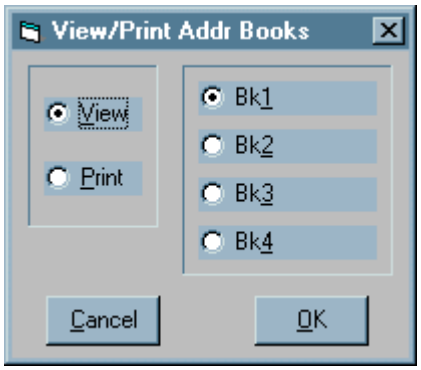
界面耻辱纪念堂--颜色的使用

iarchitect 整理，freeyourmind251 译

查看评论

如果颜色使用得当，可以通过提高界面的美感来引导用户的注意力，大大地丰富界面。另一方面，色彩的不当使用会严重地削弱用户和程序之间的交互能力。

最近更新 1999 年 9 月 19 日



尽管 Windows95 事实上允许用户可以选择多种颜色方案，但是，对于很多开发者，他们还是执着于默认值，那种中度灰色，这是标准的颜色方案。当共享电话联机 Dialog 程序运行在非默认颜色方案的机器上时，可以从其中的图片明显地看出这种现象。Dialog32 的开发者在程序中固定使用一定的颜色来匹配他们自己 PC 上的颜色方案，却没有意识到运行在不同的颜色方案下的最后效果。最后的结果很不幸…。

把这一点作为对所有视窗开放人员的一个警告：马上改变到不同的颜色方案并重新运行你的程序。在你开发的过程中，定期地改变你的颜色方案，并核实你的程序不会遭受这样的不幸结果。当你这样做的时候，决定是否改变成其它显示设置，诸如字体之类会影响到你的设计。

有时候，程序员最好的意图却实现不了。这幅图片是从一个特别的程序借用而来，这个程序在命令按钮例如所有的确定按钮（OK，Yes，Open）中固定文本的颜色为绿色，所有的否定按钮（Cancel，No，Close）都是红色文本。不幸的是，这样做会引发一系列问题。

首先，按钮的背景色由 windows 颜色参数所决定。就像上面显示的那样，固定文本的颜色会使得阅读很困难，在某些情况下，根本看不见。

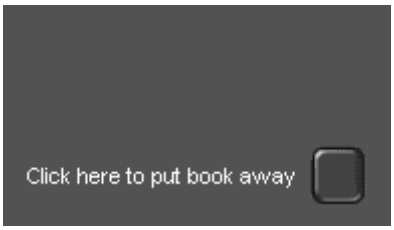
其次，就像这个例子所表现的，绿色/红色——肯定/否定的区分可能在特别的项目中并不一致。在西方国家，用户认为绿色标注暗示着“好”或“正确的响应”。但是在这个例子中，删除所有的记录所要表达的含义远比“*不*是一件好事”多得多。

另外，把你对颜色的特殊联想强加给你的用户可能会与颜色的传统解释不相容。举例来说，在一些东方国家，红色被认为是一种肯定或正面的颜色。让这些用户使用你的颜色联想，具有一种文化影响的迹象。

最后，有很大比例的人存在一定程度的色彩视觉缺陷：这通常会造成分辨红色和绿色的能力减退。你试图去提供不必要的额外信息，其中很大一部分你的用户却不需要，而且还可能引起他们的不满。

为了安全可靠，要避免把使用颜色当作一种解释手段，要让用户自己利用色彩参数。这些参数不仅仅是用户个性化自己 PC 的手段，在很多场合中，在特定的照明和显示环境下，它用来选择把应用程序的可读性设置成最佳。当你使用这些设置的时候，你应该肯定不会冒险去得罪用户。

关于把这个例子放在这里还是放在 Interface Stupidity 那一节，我们有两种观点。这幅图片由 IBM 的用户界面架构和设计小组提供。可是，他们并没有发现它，他们却产生了它；这幅图片是从他们自称是“State of the Art”RealCD 的程序中得来的。



看起来好像蓝色巨人现在把黑色看作是一个真正的现代颜色，以至于用户从它的背景中不能分辨出一个控制按钮，看起来想要变得时髦只要花很小代价。他们必须加上一个标注来指向那个按钮，这给了我们一个很好的启示：在黑色背景上的安放黑色按钮可能不是一个很好的设计技巧。

这个例子仅是 IBM 的 RealCD 应用程序中众多问题之一。想要获得深入了解，查看 RealCD Review。

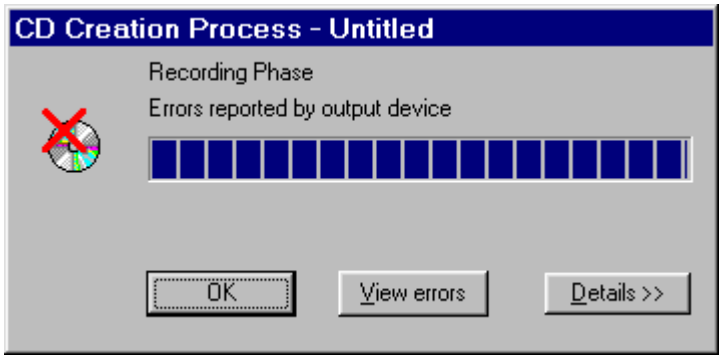
苹果公司的 QuickTime4.0 Player 规定的颜色方案对于很多用户来说都会有严重的后果。因为缺少控制部分的灰色符号和灰色背景之间的对比，我们有理由相信某些用户可能会很难定位于一个感兴趣的控制部件。当年纪大和那些带有轻微视力障碍的用户使用软件时，将会碰到不必要的麻烦。设计者未能考虑到人类视觉这个因素：

通过一位 60 岁老人眼睛的光线数量仅是一个 20 岁年轻人的 1/3。

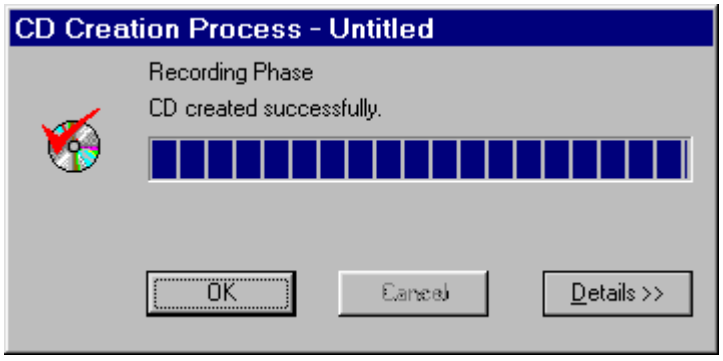
唯一确保这样的用户也能够发现这个符号的方法就是在符号和它的背景之间产生足够的对比。奇怪的是，在用户点击按钮之后，这个对比就改变了。

关于 QuickTime4.0 Player 的详细研究可以在 In-Depth section 中找到。

Joost Vunderink 发送了几张图片举例说明了一个有关 Easy CD Creator 基本的设计问题，这是一个用来刻录 CD-ROM 的程序。在创建一张 CD 之后，有两种可能结果：过程是成功的，或者‘有错误发生’（写 CD-ROM 时不会经常出现这种结果）。

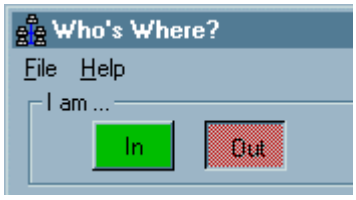


上面显示了错误信息。下面显示了成功的消息。

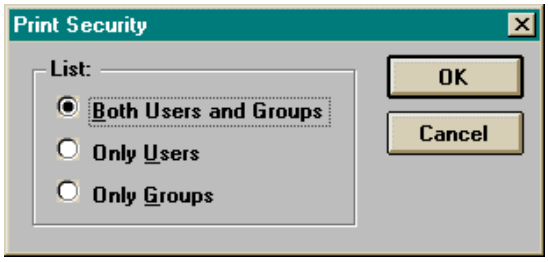


那么问题在哪儿呢？对话框太相似了，两个都利用了一个深红色按钮来表示成功或失败。就像 Joost 提示的：

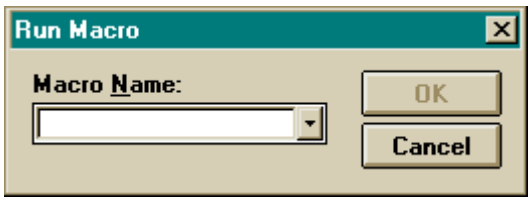
每次完成一张 CD 后，人们看到了红色的东西就会惊慌起来…但是毕竟都很正常。



提得好！我们相信我们决定了 AST 软件的设计者如何去提出他们的 In/Out 白板程序 *who's where?*的名字：这个人是‘in’还是‘out’？很显然，假设鲜红色和凸起的按钮 In，这个人是‘in’。这对于大多数人来说是显而易见的，但是对程序开发人员却不然，实际上，这个人是‘out’。



改变你的窗口颜色方案常常可以发现其它隐藏的界面设计问题。程序应该设计成采纳用户喜欢的颜色方案，以防止当用户和开发者采用不同的颜色方案时发生问题。Microsoft Access 的开发者错误地固定程序中的一些颜色，导致显示了这样的图片。（为了和用户喜欢的颜色方案保持一致，窗口的背景应该和命令按钮的颜色一样，就像下面实现的那样）。



为了公平起见，这是界面设计的一个很容易忽略的方面，需要你在多种非标准颜色方案下特别地检查你的设计。如果忘了的话，程序里重要的颜色信息可能就掩盖了，或者，最终的颜色混合可能没有吸引力，你的用户不大想用你的程序。

就像你的程序应该独立于用户喜欢的颜色方案一样，你的文档也应该独立于颜色。在标准的颜色方案下，带有灰色背景的控制按钮是只读的意思。但是，在非标准颜色方案下，没有任何标注会有一个灰色的背景。

使用颜色不当会影响一个程序的美观。Compuserve’s WinCim2.0 中的工具条提醒我们，一个小孩子用一系列组件作画。这幅图片不仅使外观看起来很不专业，并且会分散注意力。

相反，在 Microsoft Word 里，工具条使用更为局限的一组颜色，并且颜色本身很模糊。尽管有这些不同点，Word 的工具条在更少的空间提供了更多的信息，因为它主要依赖于形状来区分不同的功能。除了表现一个更为专业的外表，明智地使用颜色不大会引起基于用户个人的和文化上的颜色联想，从而产生预料之外的解释。

颜色可以降低你想传递的信息的有效性。这副图片包含一些用在 *Zoc* 中的工具条按钮，*Zoc* 是一个通信程序。在这个例子中，前四个工具条按钮表现了多种不同的发送一个文件的方法。按照显示的顺序，它们分别是：

- 发送
- 不用回车的发送
- 具有引用的发送
- 带有 CIS 引用的发送

图片之间的不同点几乎无法察觉的，每幅图片的显著位置都被蓝色对象充斥（我们都很肯定它是一个电话）。不用工具提示，用户可能分辨不出每个图片的功能；开发者都可能会简单地忽略图片。

这幅图由 *Pirates: Quest for the seas* 中得来。指南忽略了用户显示的偏好，使用了固定的颜色，它们的结合使我们回忆起以前单色计算机的显示器。我们发现这种结合对眼睛特别的有害；三维文本使得它更加难以阅读。

关于 Pirate 的详细评论可以在 In-Depth section 中找到。

这幅图片用于 Webforms 中的状态条，webforms 是由 Q&D 软件开发公司开发的。其视图用来提供描述性信息，帮助用户学会如何使用程序。不幸的是，选用的颜色和使用 3D 字体使得信息几乎不能阅读。

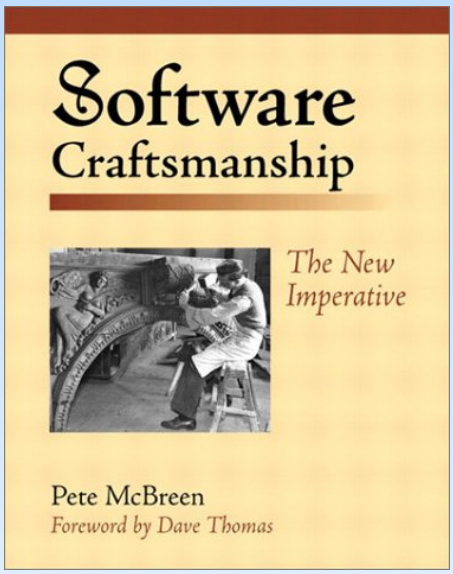
顺便值得一提的是，我们很肯定文本读做：“HTML Browse”或者“HTML/Browse”。“Preview”可能是一个比较好的选择，但是既然你无论如何都看不出来，这也没多大作用。

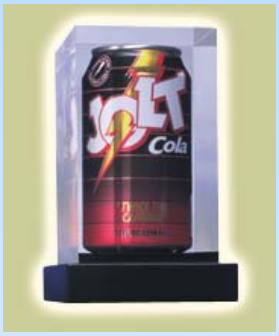
关于颜色设计信息的丰富资料可以在这里找到：Color Contrast and Partial Sight。

征稿

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

《软件工艺》





2002 Jolt Awards 获奖书籍

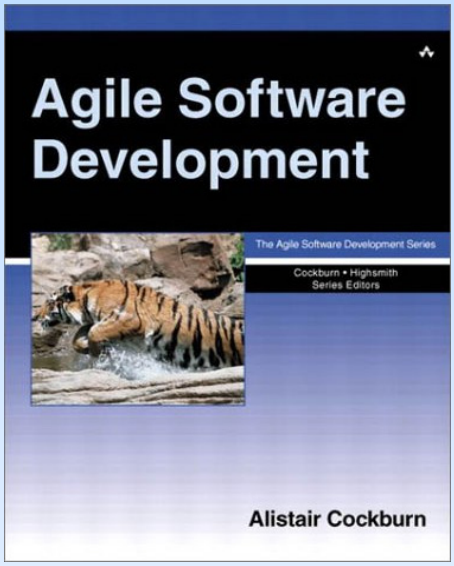
Pete McBreen

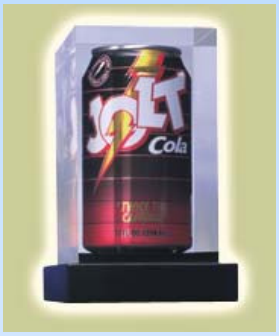
翻译：UMLChina 翻译组 Jill

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。

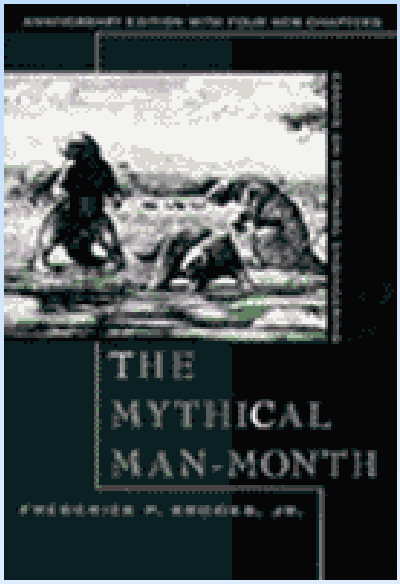
“是红酒。”我坚持道，并把标签指给她看。

“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

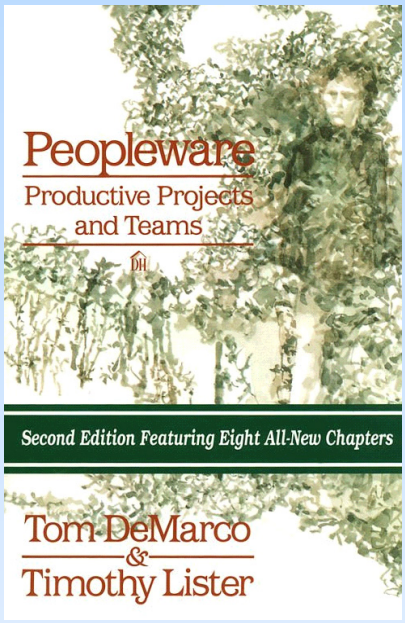
翻译：UMLChina 翻译组乐林峰

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

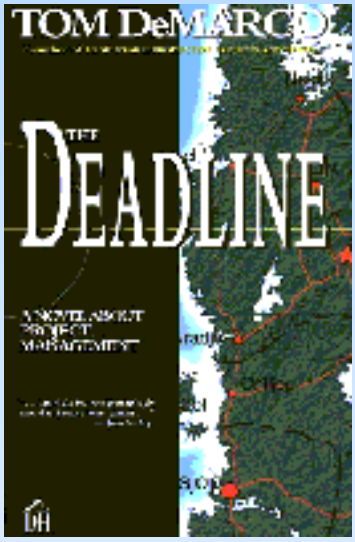
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

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！

对象-关系数据库之间的映射

Scott W. Ambler 著，Adams Wang 译

查看评论

面向对象设计的机制与关系模型的不同，造成了面向对象设计与关系数据库设计之间的不匹配。

面向对象设计基于如耦合、聚合、封装等理论，而关系模型基于数学原理。不同的理论基础导致了不同的优缺点。对象模型侧重于使用包含数据和行为的对象来构建应用程序；关系模型则主要针对于数据的存储。当为访问数据寻找一种合适的方法时，这种不匹配就成为了主要矛盾：使用对象模型，常常通过对象之间的关系来进行访问；而关系理论则通过表的连接、行列的复制来实施数据的存取。这种基本的不同使两种机制的结合并不理想。换言之，需要一种映射方法来解决这个矛盾，从而获得成功的设计。

1 映射对象

- 属性类型映射成域
- 属性映射成列
- 类映射成表。
  - 在关系数据库中实现继承。
- 映射关系。
  - 1 对 1。
  - 1 对多。
  - 多对多。
  - 关联与依赖。
  - 相同的类/表，不同的关系。

1.1 属性类型映射成域

UML 中的属性类型（Attribute Type）映射成数据库中的域（Domain）。

域的使用提高了设计的一致性，而且优化了应用的移植性。简单的域是非常容易实现的，仅仅需要替换相对应的数据类型和数据的尺寸。同时，对于使用域的属性，可能要求为域的约束加入 SQL 的 Check 串。例如，限定域的取值范围等。

枚举域（Enumeration Domain）限定了域允许取值的集合。枚举域比简单域的实现更复杂。下表对各种实现方法进行了比较。

实现方法	*优点*	*缺点*	*推荐*
Enumeration String：定义 SQL 约束来限定取值	简单	不适于数量较大的集合	常用的实现方法
为每个枚举值定义标志：为枚举取值定义布尔字段	解决了命名的问题	冗长——每个取值均需要一个布尔字段	在枚举值之间是并发关系，可同时出现时
枚举表：在表中存储枚举值的定义。可采用为每个枚举定义表或共用单个表的方法	有效地控制了数量大的枚举值。可在不改变应用代码的前提下，扩充枚举值	对于偶尔发生的应用，显得笨重。必须为枚举值的读取定义代码	适合于大数量的枚举值和开放式枚举值的情况
对枚举值进行编码：将枚举值编码为通常的数字	保存了磁盘空间，易于在多种语言中处理	复杂化了维护和调试	仅在处理多语言应用下使用

1.2 属性映射成字段

类的属性映射至关系数据库中 0 个或多个字段。

并不是类中的所有属性都是永久的。例如，***发票（Invoice）***中的***合计（grandTotal）***属性可能只是用于计算而不需保存在数据库中。另外，有时某个对象包含其它对象，如***顾客（Customer）***中的 *Address* 属性（Address

本身映射为数据库表）。此时，属性映射成多个字段。

1.3 类映射成表

类直接或间接映射成表。

除非是非常简单的应用，类与表之间才会存在一一对应的关系。本节列举三种策略在关系数据库中实现继承。

1.3.1 关系数据库中实现继承

在关系数据库中，一个关键问题是表主键的唯一性策略的选取。适当的方案能优化继承、组合及对象之间关系的实现。进一步的讨论参见*对象标识符（OID）*。

考虑类的继承问题。在关系数据库中保存对象时，基本上问题归结于“*如何在数据库中组织被继承的属性？*”该问题的不同解决方案会影响到整个设计。图 1 给出了简单的类层次例子。

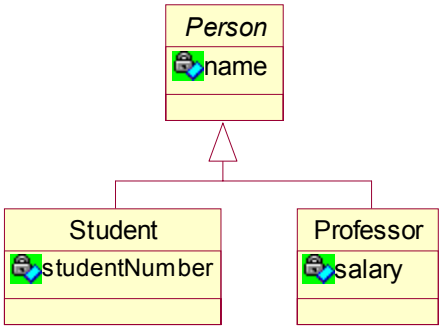


图 1：类层次 UML 示意图

关系数据库中实现继承的方法可划分为三类：

1. ***将整个类层次映射为单个数据库表。***

类层次的所有类映射为单个的数据库表，表中保存所有类（基类、子类）的属性。图 1 层次模型该方法的实现如下图所示：

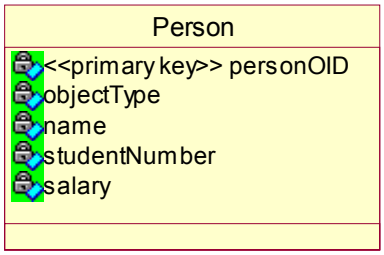


图 2：类层次映射至单个表

优点：

实现简单。

支持多态——对象角色发生变化，或存在多重角色时。

报表操作实现简单：表中包含了所有信息。

缺点：

增加类层次中的耦合。类层次中任何类的属性的增加都会导致表的变更；某个子类属性的修改会影响到整个层次结构，而不仅仅是该子类。

浪费了大量的数据库空间。

可能需要指明具体的角色。

2. 每个具体子类映射成单个数据库表。

数据库表包括自身的属性和继承的属性，每个具体的子类包含各自的 OID。抽象的基类不参与映射。图 3 描述了该方法的实现。其中，Person 由于是抽象类，未映射成数据库表；而 Student、Professor 类映射为相应的表，它们具有各自的主键。

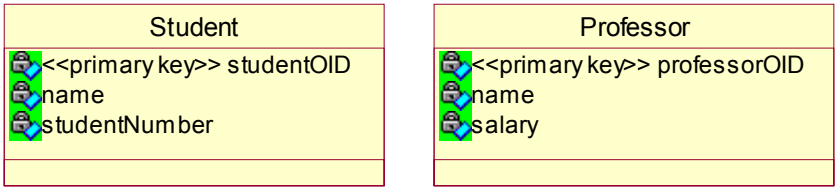


图 3：具体类映射成数据库表

优点：

报表操作实现简单：表中包含了具体子类的所有信息。

缺点：

类的修改会导致相对应的表及其子类所对应表的更改。

角色的更改会造成 ID 的重新赋值（因为不同子类的 ID 可能重复）。

难以在支持多重角色时，保持数据的完整性。

3. 每个类均映射为数据库表。

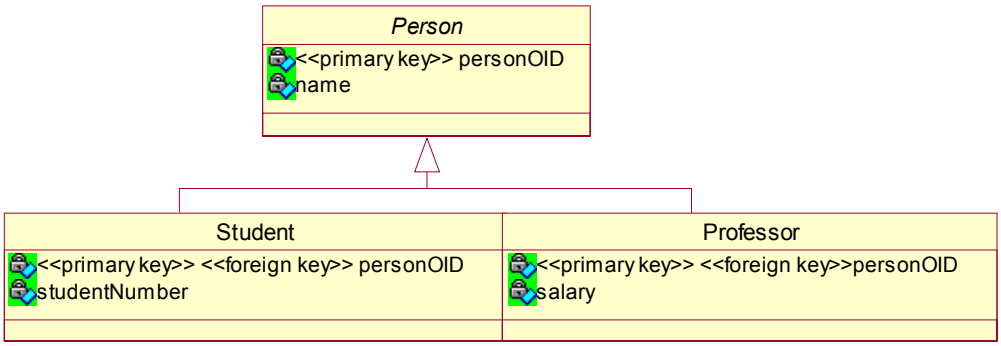


图 4：每个类均映射为数据库表

优点：

与面向对象的概念的一致性最好。对多态的支持最好，对于对象所可能充当的角色仅需要在相应的表中保存记录。

易于修改基类和增加新的类。

缺点：

数据库中存在大量的表。

访问数据的时间较长。

对报表的支持较差，除非定义视图。

以上三种方法各有优缺点，没有一种是完美的。下表为它们提供对比。

考虑因素	类层次——表	具体子类——表	类——表
报表	容易	中等	中等/困难
实现	容易	中等	困难
数据访问	容易	容易	中等/简单
耦合	非常高	高	地
访问速度	快	快	中等/快
对多态的支持	中等	低	高

1.4 关系映射

不仅仅是对象需要被映射至数据库，对象之间的关系也需要映射至数据库。对象之间的关系可分为：继承*（Inheritance）*，关联*（association）*，聚集*（aggregation）*，组合*（composition）*。要有效地映射关系，必须理解它们之间的不同点，如何实现一般的关系以及如何实现特定的多对多关系。

1.4.1 关联与聚集/组合之间的区别

从数据库的角度，关联与聚集/组合之间的区别在于对象间的耦合程度。对于聚集/组合，在数据库中对整体进行操作时通常需要同时对部分进行操作，而关联则不然。

图 5 中有 3 个类，其中 Airport 与 Airplane 之间是简单的关联关系，而 Airplane 与 Wing 之间是聚集/组合关系。从数据库的观点来说，聚集/组合关系往往在访问整体 Airplane 时，往往需要访问 Wing，而对于 Airport 与 Airplane，这种关系并不是十分明显。相应的，在保存/删除对象时，也存在类似的情况。当然，以上讨论的特定于业务规则，但该经验之谈往往在很多情况下出现。

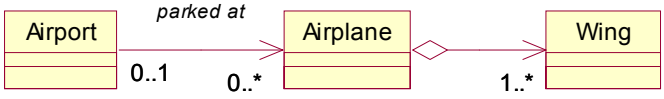


图 5：关联与聚集/组合之间的区别

1.4.2 关系数据库中实现关系

关系数据库中通过使用外键来实现关系。外键允许将表中的某一行与其它表中的行相关联。实现一对一或一对多关系，仅仅需要在表中加入另一个表的主键。

- 可选的 1 对强制的 1

将外键放置在可选的一端，该外键不能为空值。例如，某公司员工使用电脑的情况，要求员工最多能使用一台电脑，且电脑资源应充分的利用。图 6 中，Computer 中放置外键。

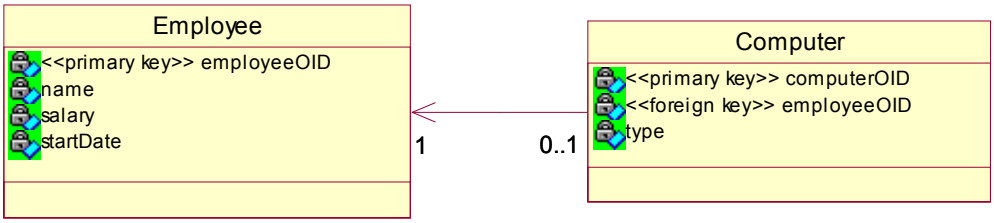


图 6：可选 1 对强制 1 关系的实现

- 其它 1 对 1 的情况

外键可放置在任意一边，具体情况依赖于性能等因素。例子如图 7。注：对于 1 对 1 的情况，不要在两个表中均放置对方的主键。这样，增加了冗余，而且并不会提高性能。对于强制性，一般在业务规则的对应层实现，不在 Persistent Layer 中实现。

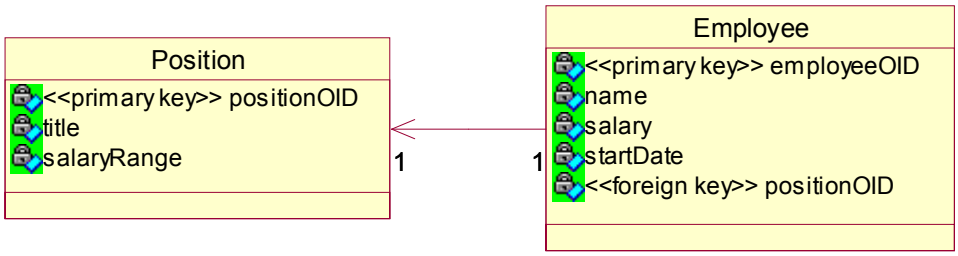


图 7：1 对 1 关系的实现

- 1 对多的情况

将外键放置在“多”的一方。外键的空/非空由对 1 的强制性决定。示意图如图 8。

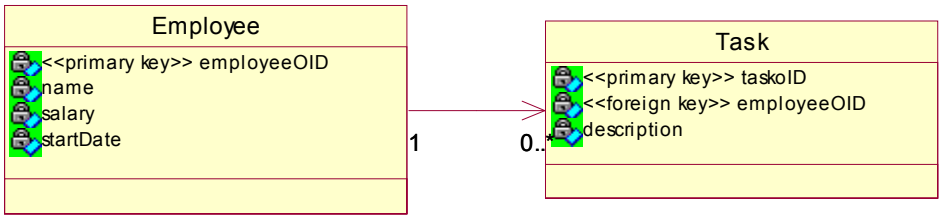


图 8：1 对多关系的实现

- 多对多的情况

实现多对多关系，需要引入关联表 *(associative table)* 的概念。(参见*术语*)

在图 9 中，Customer 和 Account 之间存在多对多的关系。图 10 显示了如何在关系数据库中实现多对多的关系。

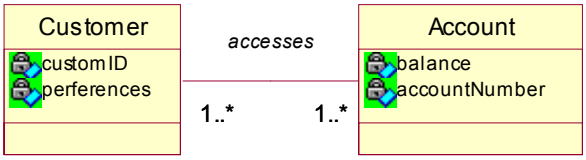


图 9：多对多关系的类

传统实现中，关联表的属性包含关系中两个表的主键，并且关联表的主键往往是它们的组合。另一种实现方法是将关联表视为普通表，使用自身的主键 OID，然后加入实现关系所必需的外键，如图 10 所示。

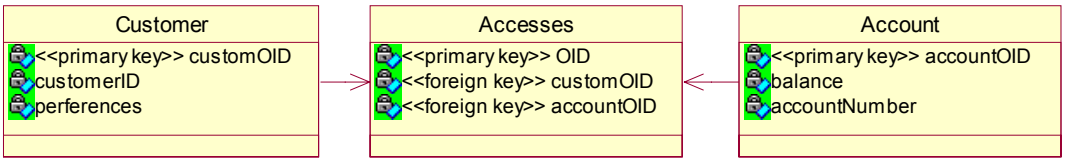


图 10：关系数据库中多对多关系的实现

该方法的好处在于 Persistent Layer 中，所有的表具有相同的形式，简化了实现；另外，提高了运行时效率：一些数据库在连接具有复合外键的表时，性能较差；并且，存在着向 Accesses 表中增添字段的可能，如：需要增加对帐户访问安全性检查，可能某个帐户只允许取钱，而另一个帐户拥有所有的权限。

2. 引用完整性及关系约束检查

在 UML 中，对象之间的关系通常指关联、聚集、组合。其中，聚集是更加严格意义上的关联；而组合则是更加严格的聚集。对象之间的关系反映了具体的业务规则，因此将对象映射到关系数据库时，必须保证对象之间的关系——定义并确保数据库上数据的约束。

在采用了*对象标识符（OID）*中 OID 的策略时（即所有的对象具有唯一的 ID，相应的数据库中所有表的主键均具有唯一性；OID 不具有业务内涵），则在数据库级发生更新时，不会出现完整性问题，但就对象交互及满足业务规则而言，对约束的普遍讨论具有一定的意义。以下就 1 对多的情况考虑限制检查。（1 对 1 关系可以视为特殊的 1 对多关系；多对多关系则可以分解为两个 1 对多关系）

2.1 对象之间的关系和父表操作的约束

- 关联

关联是一种较弱的关系。体现了实体间的联系。关系较松散，可能不需要映射，具体表现为不需要保存对方的引用，而仅仅在方法上有交互；如果在数据上有耦合关系，可能需要映射。

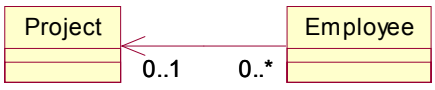


图 11：可选对可选（0-0）约束

图 12：强制对可选（M-0）约束

在 Professor 与 Student 的关系中，存在 M-0 约束（Mandatory-Optional），映射示意图如下：

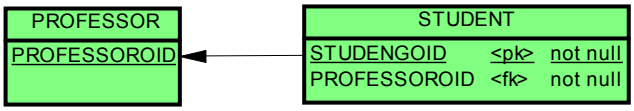


图 13：强制对可选（M-0）约束的实现

父表操作的约束：

父表的 Insert 操作，对 M-0 约束，父表中间的记录可以没有任何约束地添加到表中，因为这种约束中父亲不一定必须有子女。

*父表的键值修改操作，只有子表中其所有子女的对应值均做修改后，才能修改。*即一般采用级联更新的方法。

即：

1. 插入新的父记录，将子表中原对应记录外键更新，删除原父记录。对该操作进行封装。

2. 采用数据库提供的级联更新的方法。

*父表的删除，父亲只有在其所有子女均被删除或重新分配之后该父亲才能被删除。*在 Professor-Student 关系中，所有学生可以重新分配。

对应的实现：

1. 先删除子记录，再删除父记录。

2. 先行将子记录的外键更改，再删除父记录。

3. 采用数据库提供的级联删除操作。

在 Project 与 Employee 的关联中，存在 0-0 约束（Optional-Optional）。映射图如下：

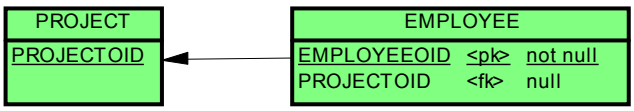


图 14：可选对可选（0-0）约束的实现

父表操作的约束：

*理论上，在具有这种关系的约束中，任何类型的修改都没有限制。*Project 和 Employee 中的行可以按需要而进行修改。具体的处理方法与聚集相同。

- 聚集

关联的一种特殊形式，指定了整体（Whole、Aggregate）与部分（Part、Component）之间的关系。如图 15 中，俱乐部与学生之间的关系。

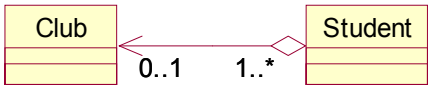


图 15：聚集类型示意

聚集一般映射成为如图 16 所示的结构，注意在子表 Student 中外键可以为空。该约束为 0-M（Optional-Mandatory）形式。

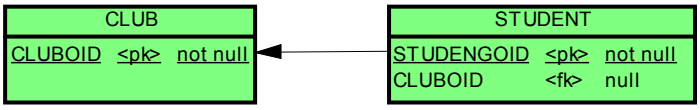


图 16：聚集关系的映射实现，0-M 约束

父表操作的约束：

父表的 Insert 操作，对 0-M 约束，一个父亲只有在至少一个子女同时被加入，或至少已经存在一个合法的子女时，才能被加入。例如，图 16 中的 Club 和 Student 之间的关系，一个新的 Club 只有在已经有学生时才能被加入（或者该学生已经存在或者可以创建一个学生，或者修改一个学生所在俱乐部的值），总之必须要已经有适当的学生存在。

具体的操作：

1. 可以先行加入主表记录，再修改子表的外键。

2. 同时加入主表、子表记录（次序无关），再更新子表的外键。注：如果先加入子表记录时，可能在关系数据库中无法保存加入子记录的数据集。其业务规则更倾向于已有子对象的组合。

父表的键值修改操作，只有当一个子女被创建或已经有一名子女存在才行。也就是说只有当至少一名滑雪者愿意参加 Scuba 俱乐部或者已经有一名学生参加 Scuba 俱乐部时，SKI Club 才可以改名为 Scuba 俱乐部。

针对于实现而言：

1. 在修改的同时将子表外键置空。

2. 子表需要级联修改。

父表的删除，理论上删除父亲是没有限制的。实际上，删除主表记录时，不采用级联删除子表的方案，而采用将子表的外键置空。该方法与聚集的语义是相一致的。

- 组合

组合指具有强主从关系和一致性的一种聚集关系。在合成对象创建之后，可能创建多个成员。成员一旦创建，就与合成对象具有相同的生命周期。成员可以在合成对象终止之前显式的删除。组合可以是递归的。如旅馆帐单与帐单上条目的关系。

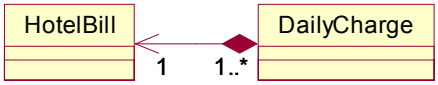


图 17：组合关系示意

组合的映射示意图见图 18，其中子表 DailyCharge 中的外键 InvoiceNumber 为强制性的。更严格的说来，它们之间的约束应为 M-M 约束（Mandatory-Mandatory）。

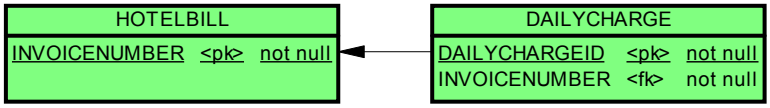


图 18：组合关系的映射实现，M-M 约束

父表操作的约束：

*父表的 Insert 操作，可能随后需要生成子女，即在子表中创建新的行。*也可能通过对子表的重新分配来实施完整性限制，如插入 InvoiceNumber = 2 的记录，接着将 DailyCharge 表中原外键为 5 的修改至 2。即将 Insert 操作封装为原子操作。但根据组合的语义，上述情况更适合使用聚集来描述。

*父表的键值修改操作，只有在子表对应外键的值修改成新值时才能执行。*根据组合的定义，更有可能是先创建新的父表记录，接着修改子表所有原对应的记录，使其与父表的新记录关联，最后删除原父表记录。

*父表的删除，只有在子表中所有相关的行全部删除或重新分配之后，才能删除父表中的记录。*同样，根据组合的定义，一般对子表进行删除操作。

父表上操作小结：

对象关系	关系类型	Insert	Update	Delete
关联	数据无耦合关系则一般不映射			
	0-0	无限制	无限制，对子表中的外键可能需要附加的处理	无限制，一般将子女的外键置空
	M-0	无限制	修改所有子女（如果存在的话）相匹配的键值。	删除所有子女或对所有的子女进行重新分配。
聚集	0-M	插入新的子女或合适的子女已存在	至少修改一个子女的键值或合适的子女已存在	无限制，一般将子女的外键置空
组合	M-M	对插入进行封装，插入父记录的同时至少能生成一个子女	修改所有子女相匹配的键值	删除所有子女或对所有子女进行重新分配

2.2 子表操作的约束

施加子表的约束主要是为了防止碎片的产生。一个明显的区别是，在一些情况下，一个子女（子表中的记录）只有在当其兄弟存在时才能被删除或修改。如在 0-M、M-M 约束中，即最后一个存在的子女是不能被删除或修改的。此时，可以对父记录进行即时的更新。或者禁止该操作。而子表约束的实现，可以通过在数据库中加入触发器；更合理、可行的方法是将子表一方的限制，在业务层中实现。

子表上操作小结：

对象关系	关系类型	Insert	Update	Delete
关联	数据无耦合关系则一般不映射			
	0-0	无限制	无限制	无限制
	M-0	父亲存在或者创建一个父亲	具有新值的父亲存在或创建父亲	无限制
聚集	0-M	无限制	兄弟存在	兄弟存在
组合	M-M	父亲存在或者创建一个父亲	具有新值的父亲存在（或创建父亲）并且兄弟存在	兄弟存在

以上的讨论是从关系数据库的角度出发，但在对象的处理中仍具有意义。在对对象进行增、删、改操作时，与之对应的数据库记录操作应同它相对应。

关系数据库基于关系数学、函数依赖等特性，无法充分体现对象之间的关系和限制。所以，即使使用数据库来强制各种约束，也不一定能保证满足业务规则的需要。其次，数据库的某些约束与生产厂商和版本相关，不利于进行移植。

2.3 小结

ID 的选取：采用*对象标识符（OID）*中的 OID 策略。由于对象标识 OID 同时作为表的主键，并且不具有业务意义。因此，可以避免在数据库操作时的很多限制，而相应的约束的处理由 Persistent Layer 转移到业务层来实现。

数据库表的映射：采用子表外键为空的方案映射对象。

父表的 Insert 操作：如果子记录已经存在，则先插入父记录，再更改子表的外键，如将空值更新为父记录，或者更改外键——此时，可能需要在业务层对子对象（子表）的完整性进行判断。如果子记录不存在，则先插入

父记录，然后根据父记录的主键直接构造子记录。

父表的 Update 操作：如果更新的父记录不存在，则先插入父记录，更新子表，删除原有的父记录。反之，直接修改子表记录，删除原有的父记录。

父表的 Delete 操作：先删除子表记录或将其外键更改（置空或修改），删除父记录。

子表的操作：子表的操作在业务层中实现。

尽量避免使用存储过程或触发器。

应用程序在执行数据约束时有很重要的作用。存在四类约束：M-M、M-O、O-M、O-O。键值的修改可能会改变表之间的关系，而且可能违反一些约束。违反约束的操作是不允许的。前面的规则只是为具体实现提供了可能性。具体的应用必须根据实际的要求和业务规则进行适当的选择。但在设计和开发时，必须考虑所分析的约束。

3 对象标识符（Object ID）

针对于对象，需要能够唯一识别它们的标识符。在关系数据库中，对应的概念称之为键 ***(Key)***；在面向对象的技术中，称之为 OID ***(Object ID)***。OID 在对象模型中的典型实现是作为完整的对象，而在关系模型中，则作为整数来实现，或者对于较大的应用中，以若干整数来实现。

OID 用来在关系数据库中唯一的标识对象。

图 19 描述了 OID 类的可能实现，图 20 则显示了映射机制。

OID 简化了关系数据库的主键方案。尽管 OID 并未完全解决对象间的浏览问题，但是它确实简化了操作。假设你不愿使用遍历的方法来读取聚集对象的成员，例如：发票与发票中的条目，则仍可以使用表关联来实现，至少提供了实现的可能。

使用 OID 的另一个好处是使开发时易于维护对象间的关系。当所有表的主键采用相同类型的列来实现时，非常容易编写使用该特性的通用代码。

3.1 OID 不应具有业务意义

OID 在任何情况下，都不应包含业务内涵。存在业务意义的任何列都有潜在变化的可能。而在关系数据库中，采用有意义的主键是致命的错误。如果用户决定改变字段的业务含义，则需要在所有使用到该信息的地方进行修

改。主键的作用应是保持唯一性和作为外键使用。任何对主键的修改会导致巨大的数据库维护工作量，显然这是不合时宜的设计。就关系数据库而言，OID 策略采用的是代理主键的方法。

3.2 OID 的唯一性

在分配对象标识符（OID）时，需要考虑两个主要的问题：***OID 唯一性的级别和如何计算它们。***

OID 唯一性级别的问题对于面向对象的初级开发者，并不十分明显。OID 唯一性具有三种级别：*具体类中对象标识唯一；类层次中对象标识唯一；所有类的对象标识均唯一。*

例如，考虑下图的类层次模型。假设对 Customer 对象给定标识 OID=74656。则存在以下的可能：

1. 该 OID 可以分配给 Employee，因为 Employee 与 Customer 是不同的具体类。
2. 该 OID 在类层次中唯一，即不能分配给 Employee 对象。
3. OID 在所有类的对象中具有唯一性，从而不能分配给其它任何对象。

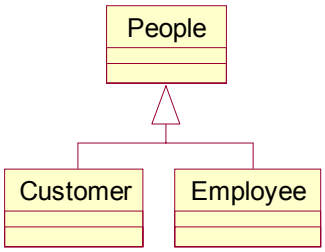


图 19：简单的类层次图

该问题其实归结于多态性 ***(Polymorphism)*** *:*有可能 Customer 对象会成为 Employee 对象。此时，如果采用具体类中的唯一性，为了避免 OID 与 Employee 的标识的重复，会导致对 Customer 对象标识的重新分配。而为了避免 OID 的重新分配问题，至少需要类层次中标识的唯一性，而所有类中对象标识的唯一性可以彻底地解决该问题。

OID 至少在类层次中应具有唯一性，最理想的方案是所有对象标识具有唯一性。

3.3 分配 OID 的策略

第二个问题，是如何决定新的 OID，这个问题对应用程序运行的效率有极大的影响。

3.3.1 在整数列上使用 MAX（）

在整数列上使用 MAX（）函数是一种常用的方法。该方法的基本思想：数据库中插入新行时，在主键列上使用 MAX（）函数取最大值，加 1 作为主键的新值。

该方法的问题是需要在访问时对列暂时加锁，尽管许多数据库对该操作进行了优化。

这种方法仅在每个表中的对象取得了唯一的对象 OID，并没有在所有对象之间保证唯一性。

3.3.2 使用并维护键值表

该策略有两种方法。保存并维护单字段、单行的数据表，用来存储整数计数。当进行插入操作时，对其加锁及增 1 作为新的 OID 值。方法的优点在于避免了调用 MAX（）函数时对整个表的锁定，同时保证了所有表主键的唯一性。缺点则是进行大批量插入操作时，该表会成为性能瓶颈（尽管在内存中，可以对该值进行缓存）以及有效值很快会被耗尽。

另一种方法是，在系统中使用多行键值表，每一行对应一个数据库表。键值表具有两个字段，一个指定相应的表名，另一个用于确定具体的取值。与前面的方法一样，它避免了 MAX（）函数对整个表的锁定，并且由于为每张表使用了键值，阔宽了 OID 的范围。然而，它失去了对所有表主键唯一性的强制；键值表仍然可能成为性能的瓶颈。（尽管可以在内存中，对这些值进行缓存）

3.3.3 GUID/UUID

许多年前，Digital 使用了一种称之为 UUID 的策略。该策略基于哈希计算机以太网卡的物理标识和当前时间来得到唯一的 128 位键值。而对于无以太网卡的计算机，则可以通过在线的文件得到标识数字。Microsoft 具有类似的 GUID 策略来得到 128 位的字符串。

两种方法均能很好的工作，尽管它们具有平台相关性。另外，不使用 Persistence 机制产生 ID，始终是一个潜在的问题。

3.3.4 使用 Persistence 机制提供的功能

许多数据库，如 Oracle，能自动的产生唯一的序列值。尽管该方案可以很好的工作，但它们采用了厂商私有的方法，且在定义时确定，从而无法进行有效地控制。如果面临跨平台移植时，可能成为非常严重的问题。

3.3.5 HIGH/LOW 方法

替代使用较大整数来获取 OID 的方法（要求对单个资源——字段进行访问，从而成为瓶颈）：将 OID 分解为两个逻辑组成部分。在应用程序首次需要创建 OID 时，向数据库的单个字段请求 HIGH 值（或者从某些数据的内建函数获得），对于 LOW 值，初始化为 0，在本次会话随后的请求递增。如果 LOW 值到达了极限，则再次向数据库申请 HIGH 值。由于 HIGH 值互斥的获得，进而保证了唯一性。

HIGH/LOW 方法的优点，每次会话只需与数据库交互一次，减少了流量，使键值表的访问不再成为瓶颈。其次，保证了 OID 在所有对象中的唯一性。

与前面所提到的方法比较，HIGH/LOW 方法是最有效和实现较简易的方法。

1. HIGH/LOW 方法的实现

采用 OO 方法实现 OID 策略，使用类封装 OID 的行为。

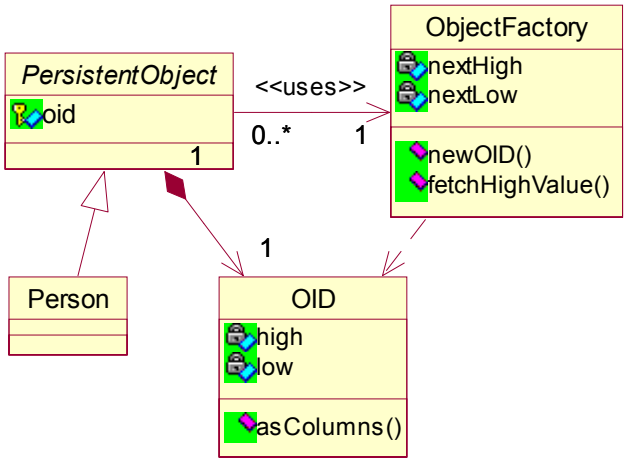


图 20：HIGH/LOW OID 的一种可能实现的类示意图

图 20 显示了实现 OID 的一种方法。其基本思想是在创建永久对象：由 ObjectFactory 对象（可参见创建设计模式中的 ***Singleton***）为它分配 OID，ObjectFactory 的唯一职责创建新的 OID 对象。ObjectFactory 跟踪 HIGH 和 LOW 的取值来得到新的 OID。具体而言，通过访问 Persistence 机制（数据库）来获取 HIGH，而基于 LOW 的当前值返回唯一的 OID。asColumns 方法则以关系数据库存储的对应形式返回 OID 的实例。

图 21 展示了在关系数据库中实现 HIGH/LOW OID 策略可能的实现方法。没有任何一种是完美的，各有优缺点。重要的是选择一种方案，并对数据库中的表一致地实施，使 Persistence 层更加容易开发和维护。

1. 为主键定义单个整数。该方法设计简单，但 OID 的取值范围较小。(0~数据库所允许整数的最大值)

2. 使用任意长度字符串作为主键避免了上述的问题，但增加了运行开销。(许多数据库对整数作为主键进行了优化，而对字符串作为主键的访问较慢)

3. 采用复合主键较方法 2 减少了存储开销，但仍具有某些弊端，具体讨论参见*对象——关系数据库映射的讨论*。

4. 结合方法 2、3，对 OID 对象进行哈希操作，将其转换成字符串。基本思想是将 OID 转化成一系列的 8 位的数字，接着根据数字构造字符，最后将字符连接成字符串。

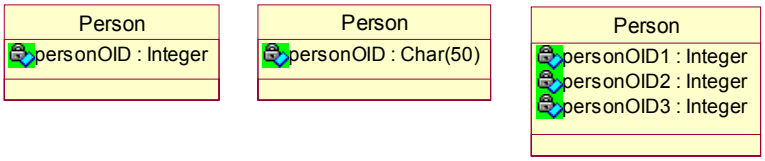


图 21：关系数据库映射 HIGH/LOW OID 的可能实现

推荐方案：

高位（HIGH）采用 96 位的数字（通常是 3 个 32 位的整数），低位（LOW）采用 32 位的数字。接着，将其转换成 128 位的字符串。该方案的最大缺点是许多数据库可能对于定长字符串的主键未进行优化。具体的方法则应根据需求来确定。

2. 分布式环境下的 HIGH/LOW 方法的实现

如前面所提到的，OID 必须唯一。在分布式环境中，客户机从众多服务器中取得 HIGH 值，应如何保证该值的唯一性呢？答案很简单：服务器之间也必须使用 HIGH 值唯一的策略。与之对应有许多方案，如：服务器使用自己的 HIGH/LOW 方法，而从唯一的资源处获得 HIGH 值；或者，服务器从某个集中式的服务得到 HIGH 值块，在 HIGH 值分配完之后重新申请新的块。

3. 多厂商数据库环境下的 HIGH/LOW 方法的实现

另一个相关的问题是，许多大型机构往往使用多种厂商的数据库来实现 Persisntence 层的设计。尽管各种厂商都有特定的机制来产生代理键值，但在分布式的环境中，大多都只能应用于特定的产品。例如：产品 A 的机制产生 HIGH 值 1701，产品 B 也产生了 1701 的 HIGH 值。这会导致两难的局面，此时就要求能将 HIGH/LOW 实现的方法应用到上述环境中，可以使用上节的方法来进行处理。

4 关系数据库常用技术

4.1 索引（Indices）

实现数据库结构的最后一步往往是为其添加索引以优化性能。

通常，需要为每一个主键和侯选键定义唯一性索引。（绝大多数 RDBMS 在定义主键、侯选键约束的同时创建唯一性索引）同时，还需要为主键和侯选键约束未包容的外键定义索引。

在此，对索引的重要性特别加以强调。主键和外键上的索引能加速对象模型中的访问。（实际上，索引用于两个目的：加速数据库的访问；为主键和侯选键强制唯一性）数据库实现必须添加索引，否则，用户将因不良的性能感到沮丧。在数据库设计的早期，就应并入索引，因为它们的实现非常简易，延迟实现并不是一个很好的主意。

数据库管理员（DBA）可能会为经常使用的查询定义附加的索引。DBA 也可能使用特定产品的协调机制。

4.2 存储过程（Stored Procedures）

存储过程是运行在关系数据库服务器端上的函数/过程。尽管 SQL 代码通常是存储过程的主要组成部分，但大多数厂商使用特有的语言。典型的存储过程运行一些 SQL 代码，进行数据处理，接着以零行或多行的形式返回，或者显示错误信息。存储过程是现代关系数据库非常强大的工具。

在向关系数据库映射对象时，假设未使用 Persistent Layer 的前提下，有两种情况使用存储过程较合理。

第一种情况是建立快速的、简陋的、随后将遗弃的原型。存储过程看上去是建立原型最快速的方法。

第二种情况是必须使用已有的数据库。并且，该数据库的设计不适用于对象方法，无法使其为特定的要求服务。则可以适用存储过程，以类似于对象的方法来读写记录。注意，存储过程并不是唯一的方法；相应的，可以使用其它语言来书写，并在数据库外运行代码。（可能仍在 Server 端，以避免不必要的网络流量）

另一方面，有许多充分的理由不使用存储过程。

使用存储过程，Server 端很快成为运行的瓶颈。当某个简单的存储过程被频繁的调用时，会极大地降低数据库的性能。

存储过程一般采用特定于厂商的语言编写。在不同数据库之间进行移植时，甚至在相同数据库的不同版本移植时，会出现很多不可预见的问题。

存储过程极大地增加了与数据库之间的耦合。它降低了数据库管理的灵活性。在修改数据库时，会要求重写存储过程，从而增加了维护的工作量。

底线：存储过程仅仅是一种用于解决短期问题的蹩脚方法。

4.3 触发器（Triggers）

触发器实际上是一种可以自动调用的存储过程。在增、删、改前调用触发器是非常普遍的方法。触发器必须成功运行，否则相应的操作失败。触发器通常用来保证数据库的完整性。

与存储过程一样。触发器使用特定厂商的语言，使它们较难移植。好消息是一些建模工具能基于关系的信息自动生成触发器。只要不修改生成的代码，则可以在跨厂商/版本移植时，为数据模型重新生成触发器。

建议：不使用 Trigger，即使在保证数据完整性时。数据完整性往往相关于特定应用域，可以在 Business Layer 或 MCV 模型中的 View 层次实现。

5 对象——关系数据库映射的讨论

- 对象设计和关系数据库是常用实现标准
- ODBC 和 JDBC 的类是不够的
- 需要 Persistence 层设计
- 硬编码的 SQL 语句是很糟糕的主意
- 必须映射到现存的数据库
- 数据模型不应驱动类设计
- 表连接速度很慢
- 避免使用带业务意义的主键
- 合成主键也是糟糕的主意
- 需要多种实现继承的策略
- 存储过程是一种很蹩脚的方法

5.1 对象设计和关系数据库是常用实现标准

由于对象/关系机制的不匹配，许多面向对象的设计者声称不应使用关系数据库。但事实是 99%的开发环境是面向对象的开发方法和关系数据库的 Persistence 机制，是常用的实现标准。

5.2 ODBC 和 JDBC 的类是不够的

尽管许多开发环境提供了初级的访问关系数据库的机制，它们是很好的一个开始。常用的方法包括 Microsoft 的 ODBC 机制（开放数据库连接——ODBC）和 Java 数据库连接（JDBC），绝大多数面向对象的开发环境提供了封装这些标准方法之一的类库。

这些类库的基本问题，同它们封装了许多对本地数据库的访问一样，是太过复杂。在设计较好的类库中，仅仅提供 ***delete***、***save*** 和 ***retrieve*** 消息来实现基本的 Persistence 功能。数据库中与多种对象协同的界面并不十分复杂。

开发环境所提供的访问数据库的类仅仅是开始，设计实现工作中很小的一部分。

5.3 因而，需要 Persistence 层设计

Persistence 层封装了对数据库的访问，允许开发者专注于业务领域的问题。即意味着，封装访问数据库的类，为开发者提供足够简单的完备接口。另外，对数据库设计也提供了封装，使开发者无需了解数据库的私有实现。Persistence 层彻底地隐藏了存储机制，隔离了可能的修改。

以上的讨论暗示 Persistence 层需要数据字典来提供映射对象所需要的信息。当业务规则发生变化时，Persistence 层的代码应无需改变。另外，如果数据库更改时，可能是安装新的数据库或是 DBA 重组数据表，唯一所需要修改的是数据字典中的信息。简单的数据库的改动不会导致应用代码的变化，如果需要得到易于维护的永久化方法，数据字典非常关键。

5.4 硬编码的 SQL 语句是很糟糕的主意

一个相关的问题是应用程序中的 SQL 代码。使用 SQL 代码使程序与数据库设计耦合在一起，减少了程序的可维护性和升级能力。其问题在于无论什么时候数据库发生改变，可能仅仅是表、列改名或发生移动，都必须修改程序代码。Persistence 层可以使用的另一个更好的办法是：

基于数据字典的信息生成动态 SQL 语句。

诚然，动态 SQL 运行慢一些，但提高了可维护性和灵活性。

5.5 必须映射到现存的数据库…

尽管现存数据库的设计很少能满足面向对象应用的要求，但事实是现存的数据库始终存在。80 年代集中式数据库的使用，现在成为了集中式的灾难：因为许多应用程序依赖于数据库，使数据库的方案难以修改。现实暗示着很少的开发者能够以反映面向对象设计的关系数据库开始，相反他们需要应付现存的数据库。

5.6 …数据模型不应驱动类设计

仅由于需要映射至原有的数据库并不意味着要求搅乱对象设计。许多基于数据模型的设计会导致设计的失败。原先数据库的设计者并未在设计中使用继承或多态的概念，也未利用映射过程中所引入改进的关系技术（见后）。成功的项目使用面向对象的技术建立业务模型、用数据模型对现存的数据库建模，接着引入映射层，对将对象设计映射至旧数据库的逻辑进行封装。有时由于数据建摸过程中过时或不良的设计，使映射代码是螺旋式的，重新设计部分数据库可能比编制相应的映射代码更加容易。

5.7 表连接速度很慢

常常要求从多个表中获取数据来构造复杂对象或一系列对象。关系理论使用表连接来得到数据，这种方法经常被证实是很慢的，或者对于一个活泼、生动的应用程序来说是不可接受的。因此，***不要使用连接***。若干精练的访问过程往往比大的连接更有效——使用遍历来读取数据。使用遍历替代连接部分解决了*对象/关系不匹配*的问题。

5.8 避免使用带业务意义的主键…

同关系理论的基本原则相悖，对象-关系数据库映射的经验要求主键不应包含业务含义。其基本思想是任何具有业务意义的字段都不在控制范围内，从而设计者面临值和规划改变的危险。业务环境琐碎的改动，可能仅仅是客户号长度的增长，都会导致数据库昂贵的修改，因为客户号作为外键多处使用。的确，许多数据库提供管理工具自动完成对应操作，但仍然是易于出错的工作。最终，仅因概念而使主键依赖于业务规则并不适宜。

5.9 …合成主键也如此

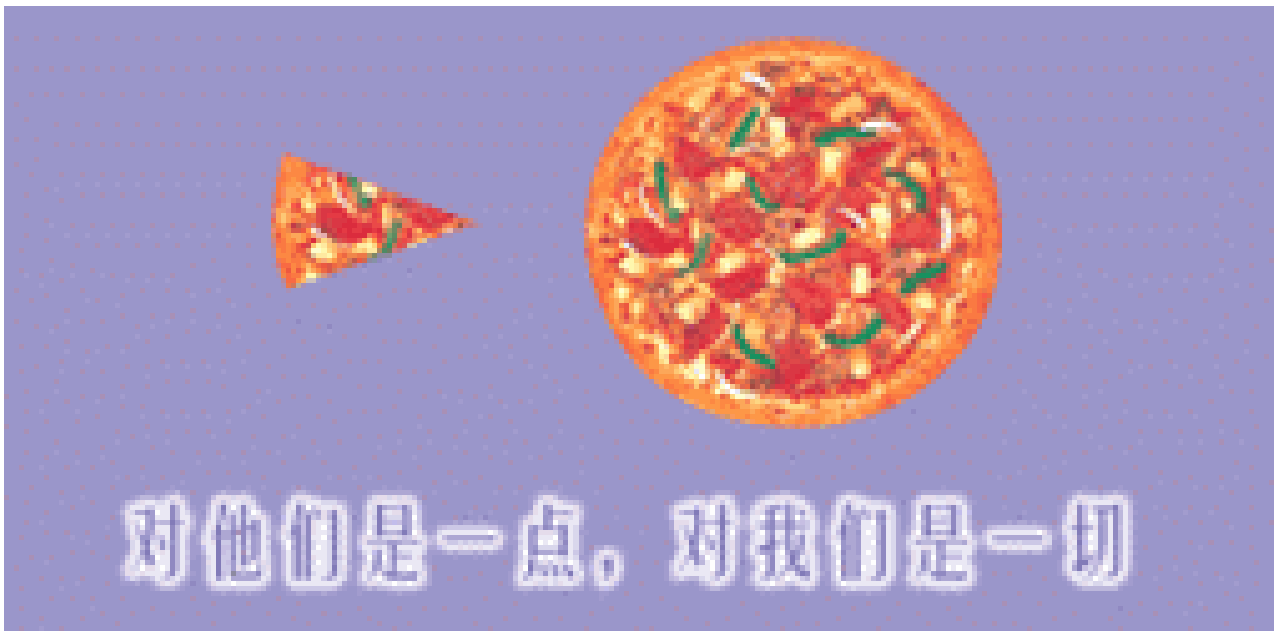
合成主键同样也不宜于使用。合成主键提高了运行开销；增加了设计的复杂性；当涉及多个字段时，往往需要额外的处理。OID——唯一标识对象、单个字段的、无业务意义的主键是非常好的选择。理想的 OID 在企业级数据库中具有唯一性，换言之，任何表中任何记录均有不同的主键。它是一种简单有效的方案。

需要多种实现继承的策略

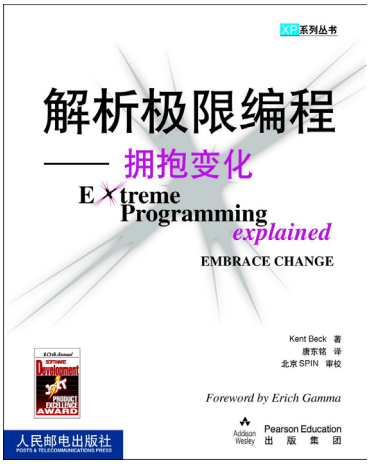
关系数据库中有三种实现继承的方法：整个类层次结构映射至单个表；为每个具体类使用数据库表；每个类均对应数据库表。尽管每种方法都可以很好地工作，但没有一种能适用于各种情况。结果是要求 Persistence 层对三种方法都要提供支持，尽管采用为具体类映射表可能是最容易开始的一种实现。

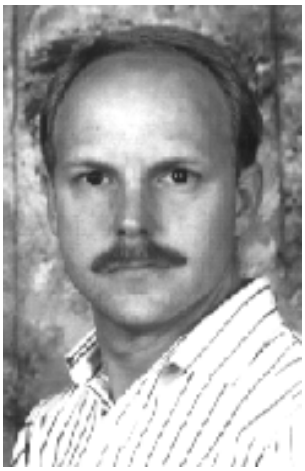
5.10 存储过程是一种很蹩脚的方法

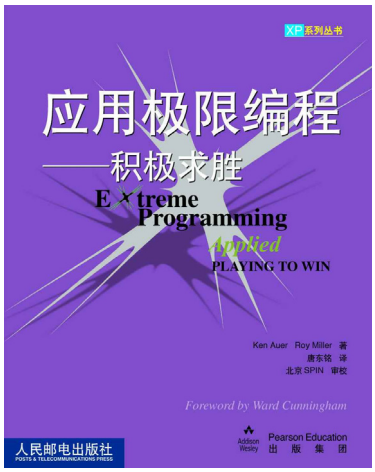
该问题在前面已经讨论，结论是存储过程仅仅是一种用于解决短期问题的蹩脚方法。



人民邮电出版社 XP 系列







如何用状态图进行设计

Dr. Doron Drusinsky 著，Jill 译

查看评论

所有软件实质上都可以认为是一种特殊的状态机。状态机是一个公共术语，用来描述一个系统在某种条件下会做什么以及按什么顺序去做。就像一个程序员写的声明，必须按某种顺序执行，每条声明说明计算机状态的改变。

用可视化对象、状态图来描述系统行为的思想在设计方法中非常流行。状态图可用于系统和子系统行为建模，应用范围从简单商务应用到最复杂的通讯协议。

有两类系统行为：转换和交互，学习状态图的重要一点就是要理解这两种系统行为的区别。

转换（子）系统

转换（子）系统在调用时所有输入信息都已经准备好了，在某个计算过程后，会产生输出信息。如图1所示：

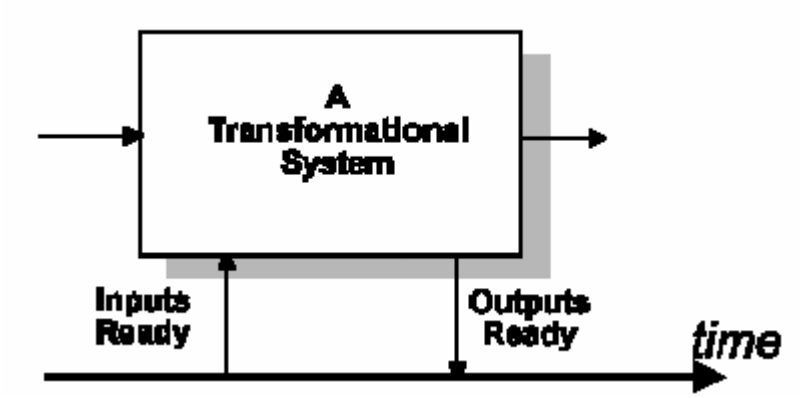


图1　一个简单的转换系统

转换系统的例子有：数据获取系统，音频压缩系统（软件和硬件），或者甚至是一个简单计算输入数值的平方根的过程。自顶向下分解是转换系统通常使用的设计方法论，因为它能够将复杂的输入输出关系分解为简单的、更易管理的关系。同样，传统的编程和系统级规格说明语言经过转换、修改，以适合自顶向下的功能设计方法。

交互系统

易于理解的一个交互系统例子是交通灯控制器。它的所有输入不可能同时存在，输入是没有终结的无穷序列。不可能编写一个转换系统来实现这样一个控制器。事实上，大多数控制器被定义为交互式的，而不是转换式的，其应用领域包括过程控制、军事、航空、汽车工业、DSP、ASIC设计、医药电子和类似的嵌入式系统。

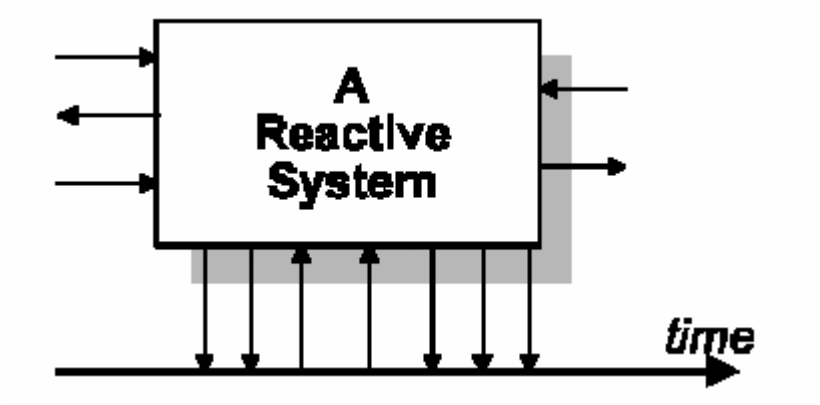


图2　一个简单的交互式系统

其实每个系统都有交互式组件，因为一个系统几乎不可能脱离它的环境单独存在。反之，系统存在的原因就是为了与一些实体或环境中的实体合作或交互。这种合作通过发送、接受、识别、拒绝信号序列——这一系列交互行为来完成

扩展状态图和Petri Nets与交互式系统有关（这是BetterState 的优势）。在本教程中我们称交互子系统为*控制器*。请不要将它与经典的控制理论相混淆。

状态图基础

状态图类别

在我们开始讨论状态图设计方法以前，分清楚两种常用的状态图是很重要的。状态图（State Diagrams）和扩展状态图(StateCharts)（后者是前者的一个扩展）。这两种类型我们都将讨论。BetterState4VisualBasic 和BetterState Pro都支持这两种类型，不过，我们关注的主要是扩展状态图。

传统上，有限状态机（FSMs）和它们的图表副本、状态图用来详细说明和设计交互（子）系统。因为它具有高度可视化和直观的特性，所以被普遍接受。FSMs可以描述有限和无限序列，这一特性再加上它的可视化功能，

使它成为电子工业最受欢迎的形式之一。

传统的状态图

状态图比相应的文本解决方案更易于设计、理解、修改和文档化。传统的状态图在过去几年一直没有大的变化，因此，在今天的交互系统应用中存在局限性。在本节的后面，你将发现扩展状态图弥补了这些缺陷。传统状态图的局限性包括：

传统状态图很**单调**，它们不适合自顶向下的设计和信息隐藏。而且，自顶向下的设计概念需要能够交互的软件，这样可以让用户通过操作和浏览复杂的设计来获取信息。

传统的有限状态机必须是完全**顺序**的。然而，应用程序并不是这样。现代的交互子系统（我们称它为控制器）必须能够对环境中大多数实体发出的信号进行响应。考虑一个电话应答机控制器在接第一个电话时第二个电话等待的情况。一个传统的 FSM 需要计算第一个和第二个打电话者所有可能的状态组合，这会导致我们通常所说的状态溢出现象。

扩展状态图

为了弥补状态图设计的这些限制，David Harel 在他的论文“扩展状态图：解决复杂系统的可视化方法”中描述了扩展状态图。该论文发表在*计算机编程科学*（1987）上。在增加了层次、并行、优先级和同步等功能的同时，扩展状态图保留了有限状态图的可视化、直观等特性。

下文讨论可用于传统状态图和扩展状态图的基础设计方法。后一部分讨论的是扩展状态图特有的设计方法。

状态图符号

状态图的主要组成部分是状态和代表状态改变和转换的箭头。状态通过大量不同类型的符号图形化表示。这些符号包括圆、矩形、圆角矩形等。BetterState使用圆角矩形来表现系统状态。

系统状态

在 Webster 的 *New World Dictionary（新世界字典）* 中对“状态”的定义如下：

“在给定时间、方法和行为的情况下，与某人或某件事相关的一组环境变量或属性集”

状态图用于图形化显示状态以及在任何时候的状态之间发生的交互。如图3所示的简单状态图，包括一个源状态，一个目标状态，以及这两个状态间的转换。

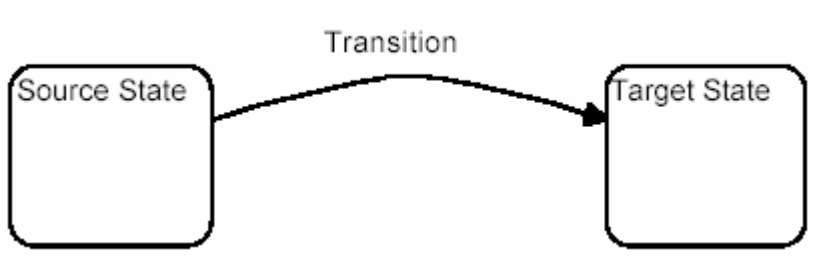


图3　简单状态图

处于如下情况时，系统在一段时间内只能响应一个可视状态：

1、　在外部环境中等待某事件发生。

2、　等待环境中的当前活动（如混合、洗、填充、计算）去改变其他活动。

这并不意味着我们的系统没有执行动作的能力，我们将在本章的后面讨论与动作相关的问题。不过，值得注意的是，系统在当前可视条件下的状态和所执行的动作不同。 因此，一个状态必须呈现为系统的某种行为，这些行为是可见的，而且它们将持续一段有限的时间。

注意：在BetterState中，当你添加了一个状态，将显示一个对话框。在该对话框中你可以输入状态名，所需的动作类型（有关动作的详细内容将在后面讲述），并设置与状态相关的其他选项。有关这个问题的详细信息请参见*BetterState用户指南*，第四章 “使用扩展状态图”

改变状态

一个系统有自己特有的管理自身行为的规则。这些规则指定在什么时候系统会从一个状态改变为另一个状态。一个有效状态改变称为转换或迁移（transition）。一个转换连接有关的一对状态。代表转换的符号是一条带箭头的直线。箭头方向表示转换的方向。

图 4 所示的状态图说明了系统可以从状态 1 转换到状态 2。它还说明处于状态 2 的系统什么时候可以转换到状态 3 或回到状态 1。

不过，根据这张状态图的设计，系统不能从状态 1 直接转换到状态 3。另一方面，这张状态图告诉我们系统可以直接从状态 3 回到状态 1。

注意：状态 2 有两个后继状态（状态 1 和状态 3）。这在状态图中是很常见的。任何状态在不同的环境下都可以有许多相应的后继状态。

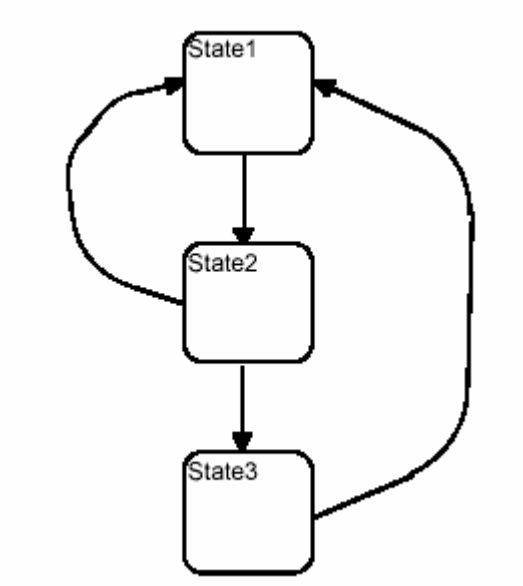


图4 简单的三状态FSM

图 5 告诉我们依赖于时间的系统行为的一些有趣信息。不过，它没有考虑非常重要的一个系统元素。系统的初始状态是什么？根据图 5 的系统模型，该系统将永远运行，并将不停的运行下去。而现实中的系统必须从某个状态开始运行。

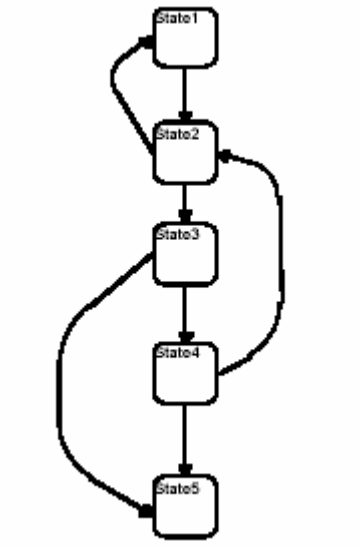


图 5　一个没有初始状态的简单状态图

在 BetterState 中，初始/开始状态用一个缺省的符号（ Ⓓ ）来表示。图 6 是一个和图 5 相似的状态图，它指定了一个开始状态。

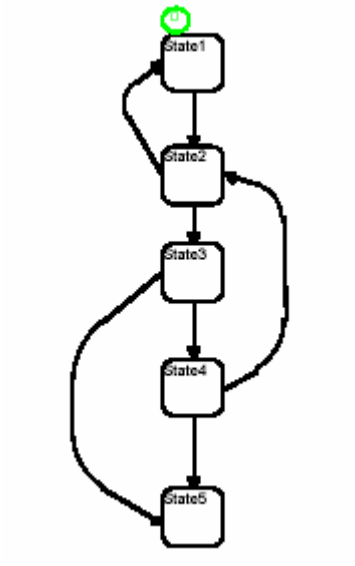


图 6　一个有开始状态的简单状态图

注意：一个简单的FSM可以仅有一个初始状态。但对于扩展状态图，由于要表现层次和并行关系，可能有多个开始状态，并且在任何时候都可能有多个状态机在运行。

条件和动作

要使我们的状态图完整，你可能想增加两个额外的选项：

选项**条件** (conditions)　导致状态的改变

选项**动作** (actions)　表示系统在改变状态时将做什么。

注意：事件驱动的编程语言，例如在Delphi, Visual Basic, 和 MFC中，**事件**会引发状态的改变。有关该问题的详细信息，请参见*用户指南中的事件驱动编程*部分。

转换**条件**是系统能够检测到的外部环境中的一些条件。它可能是一个信号，一个中断，或者一个到达的数据包。它可能是任何导致系统状态改变的东西，使系统从一个*等待状态 x* 改变到一个新的*等待状态 y*，或者从执行*活动 x* 改变为执行*活动 y*。

在改变状态过程中，系统可能执行一个或多个**动作**。它们可能是产生一个输出，或在用户终端上显示一个消

息，执行某些计算等等。在状态图中，**动作**是为了响应外部环境，或执行计算，系统将记录结果以便响应将来发生的某些事件。

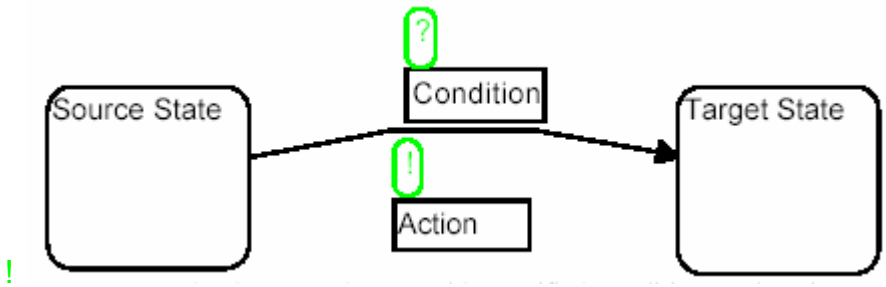


图7：有特定条件和动作的简单状态图

注意：在BetterState中，当你在两个状态之间添加一个转换后，会显示一个对话框，在对话框中可以定义转换的条件以及所需的动作。这部分将在*BetterState Pro用户指南*的第四章详细讨论。

设计第一个状态图

现在，你已经掌握了足够的状态图设计方法的基础知识，可以对系统行为建模了。我们的第一个设计是个小小的挑战，**设计冰激凌销售机的逻辑，它应该能够检测到卖了三个蛋卷冰激凌。**你可以先用笔和纸来设计，这样可以加深对前面所学东西的理解，当然，你也可以跳过这一步，直接使用 BetterState。

注意：请记住教程第二章的目的是告诉你如何设计扩展状态图。我们只用了很少的篇幅介绍 BetterState 的用法。这部分将在第三章详细讲述。如果你想使用 BetterState 来完成这个设计，建议你先学习第三章，它将引导你逐步完成一个类似的设计。

在这个简单例子中，先添加一个初始状态，命名为 Start，然后添加第二个状态，名为 One Sold。One Sold 状态指你的系统在卖出一个冰激凌后的状态。现在你可以从状态 Start 转换到状态 One Sold。然而除非真的卖出了一个蛋卷冰激凌，否则你不希望系统发生状态转换（从状态 Start 转换到状态 One Sold）。因此从状态 Start 到状态 One Sold 的转换条件为 *Cone Sold*（卖出了一个蛋卷）。该过程将继续，直到三个蛋卷都卖出去了。在这个例子中我们没有使用动作，不过我们可以使用。可以为该设计加上一个动作，如每次卖出冰激凌后在收银机上记录销售额。

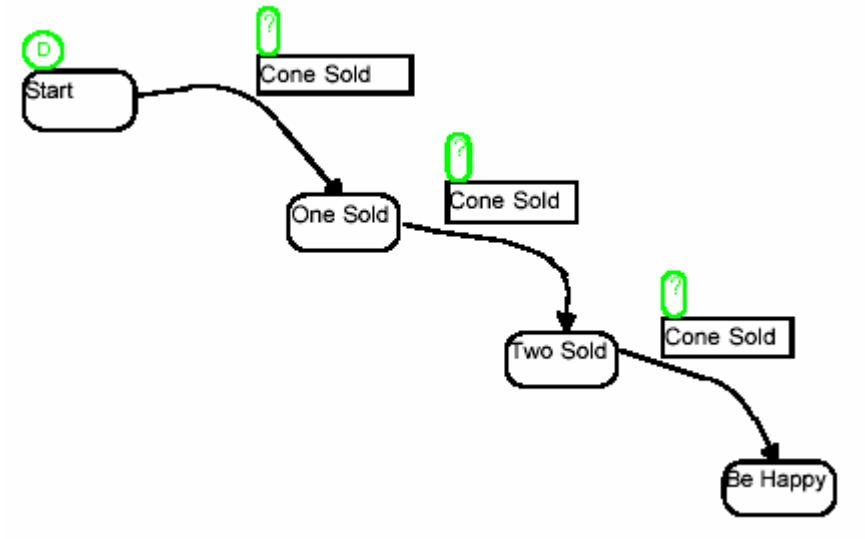


图8 冰激凌销售机计算机系统说明

到目前为止，我们已经讨论了状态图的原理。这些原理对状态图和扩展状态图都适用。第二章后面的部分主要讲述了扩展状态图的扩展功能。我们将围绕这些增强的功能，使你对BetterState Pro的设计能力有很好的了解。关于这些内容和其他有关扩展状态图特性的完整信息，请参见BetterState Pro用户指南。

设计层次结构

在复杂系统中，可能会有许多独特的系统状态。想在一个单独的图中把它们都画出来，即便可能也是相当困难的。分层结构是公认的设计和管理复杂设计的方法。分层就是把相关的状态放在一起。分层结构帮助人们抽象过程，它是现代计算机软件的基本特性之一。

在 BetterState 中，分层表示为把状态放到其他状态内部。图 9 是我们所讲述内容的一个例子。在这个例子中，分层的作用是将相关一组状态放在一起。高层的转换如下图所示，高层状态是状态 1 或状态 2，转换可以实现。如果是状态 2，应具体说明是状态 3 还是状态 4。

在现实世界中，分层是如何应用的呢？让我们复习一下冰激凌机的状态变化的例子。

现在改变我们原来的条件，不仅检查卖了3个冰激凌，还检查在卖了一个或数个冰激凌后又卖了一个冰激凌容器。

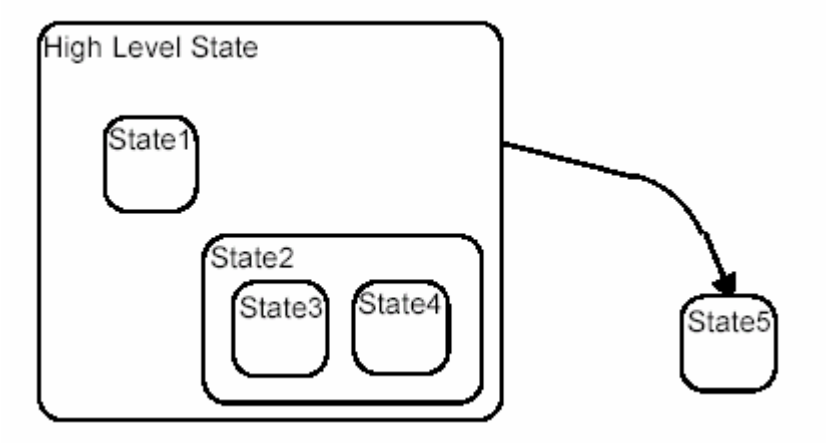


图9 有两个子状态的分层状态，其中一个还有额外的子状态

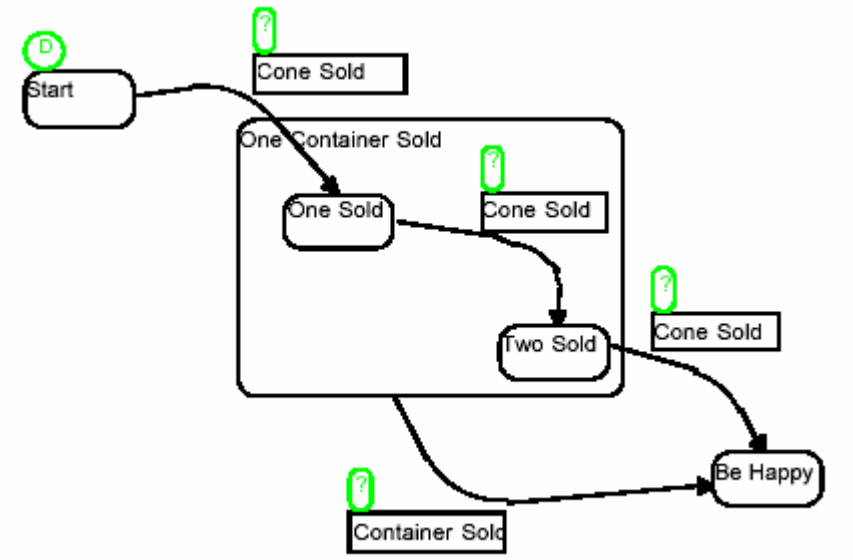


图10 使用分层结构修改后的设计

通过在状态 One Sold 和状态 Two Sold 外简单的增加一个新状态。我们创建了一个高层状态，它代表冰激凌容器的销售。条件 *Container Sold* 称为高层转换。在本例中，如果高层状态是状态 One Sold 或状态 Two Sold 并且满足 Container Sold 的条件，高级转换就会发生。它的含义是如果卖出了一个或多个冰激凌，接着又卖出了一个冰激凌容器，系统将转换到 Be Happy 状态。

独立的控制线程

扩展状态图也提供了获取无序的输入事件的方法。这意味着一个状态开始时，它可能位于一个或多个控制线程的交叉点。控制行为的每个独立线程都类似一个状态机，独自运行，互不干扰。因此，这些控制线程可能会同

时发生状态转换。*（Independence）独立/并行（也称 Concurrency）是一个非常强大的设计功能。有关使用 Concurrency ／ Independence 的完整信息和更多的示例，请参见 BetterState Pro 用户指南第四章，*

为了阐明这个例子，我们扩展我们的冰激凌系统的模型，为它增加*热狗（hot dogs）*的销售。在这个例子中，除了统计冰激凌和冰激凌容器的销售数目，我们还要为那些有需要的顾客供应*热狗*。

你需要设计一个新的状态图，现在有两张单独的状态图。如图 11 所示：

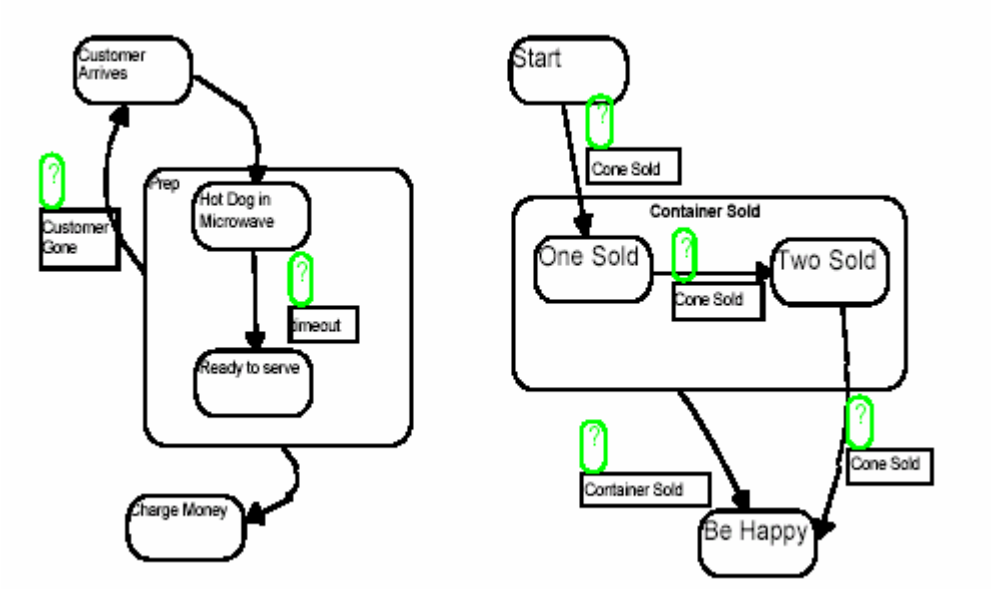


图11　这两个独立的任务在工作时都要执行。

并行为你提供了一个强大的设计工具，它可以显著地简化如图 11 所示的设计。如何用并行来设计呢？先增加一个状态，然后从 Create 菜单中选择 Concurrency / Independence 选项。 在弹出的对话框中为新创建的“控制线程”命名，现在你可以在新的线程里设计状态图了。

图 12 显示了我们的冰激凌/热狗的设计，在 Work 状态中绘制独立的控制线程。

请注意，图 12 是多么易于理解。在图 11 中，我们有两个单独的状态图，很难说明这两个图的关系。而在图 12 中，很容易看出两个子状态图是相关的。这说明，在新的一天开始工作时、工作中——无论任何时候，你都可以在卖冰激凌的同时卖热狗。因此，在下班时，你应同时停止这两个活动。分层和并行让我们能够用更小和更易于管理的状态图来处理和解决复杂的任务。

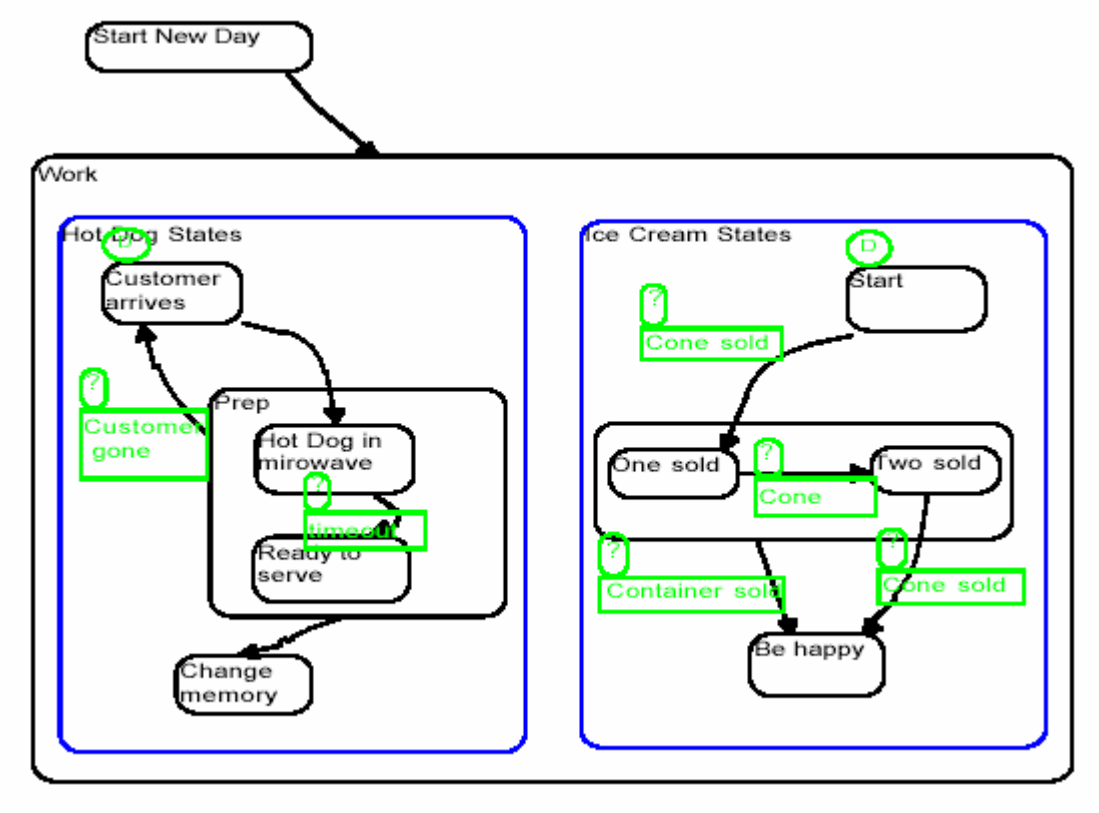


图12 两个控制线程

可视化同步

可视化同步提供用图形表现独立线程之间相互交互的方法。可视化同步功能强大，因为基于消息的文本无法传递到底使用了哪种机制等信息；准确的行为只能从状态图中获得。例如，让我们修改图 12 中的设计，当冰激凌线程完成了 happy sale，强制热狗线程找钱。修改后的状态图如图 13 所示。

可视化的 Switch/Case（决策多边形）

在传统的有限状态图中，一个条件，例如 X，只能是两个可能结果之一。一般为 True 或 False。而在具有可视化 Switch/Case 功能的 BetterState 的流图中，它可以是一个决策多边形，可以有多个结果，没有二选一的限制。理论上它可以支持任意数量的结果。

可视化 Switch /Case 表现为两个或多个目标状态中做选择。*可视化 Switch/Case* 中表达式的值决定能否发生转换。根据表达式的值，状态图转换为多个结果状态中的一个。

在扩展状态图中的任何状态，任何层次，设计者都可以增加一个*可视化的 Switch/Case* ，它以图的方式表现了 C / C++ /Verilog 中的 switch 语句，Visual Basic 中的 Case Select 语句，和 VHDL 中的 Case 语句。

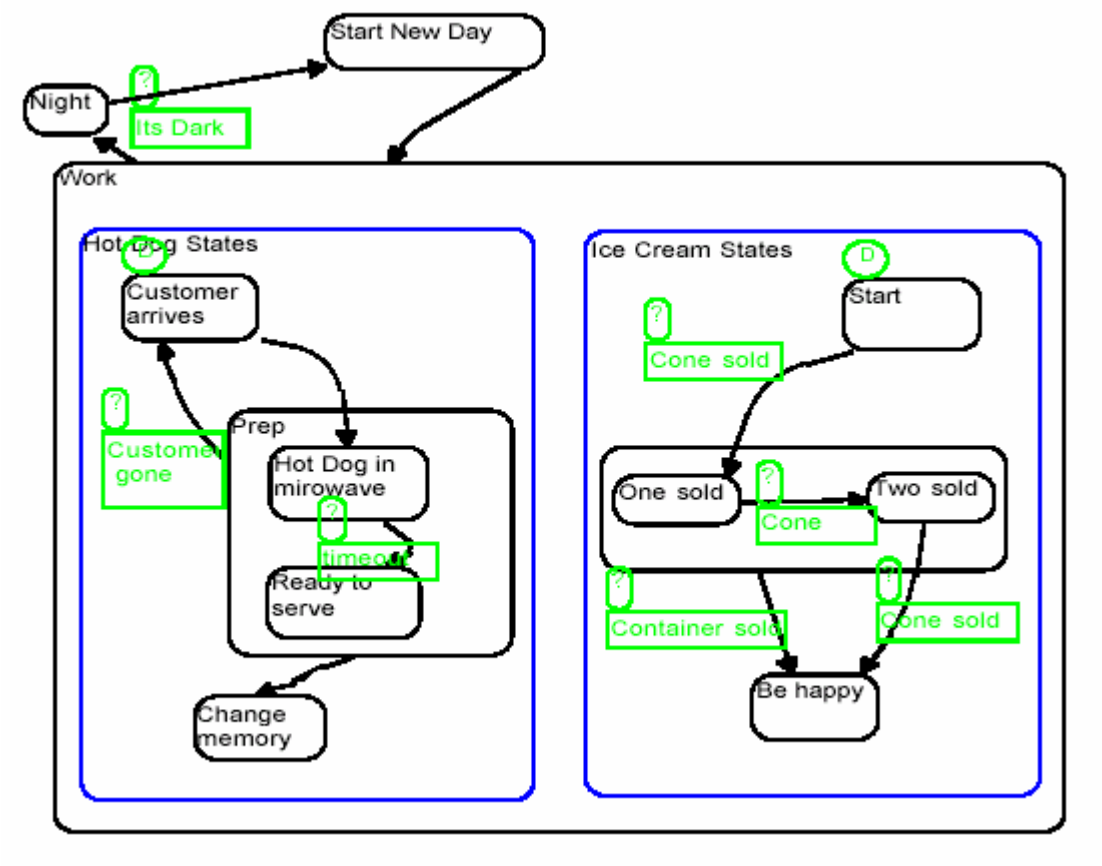


图13 Be Happy状态和Charge Money状态同步

图 14 是一个使用了可视化 Switch/Case 的例子。我们将在状态图中使用可视化 Switch/Case 来描述我们在雨天的行为。在图中，我们的行为将根据雨的大小而改变。如果雨下得很大，我们就回家；如果雨不大，我们就打开伞继续营业；如果是毛毛雨，我们就当作没下雨，一切照旧。

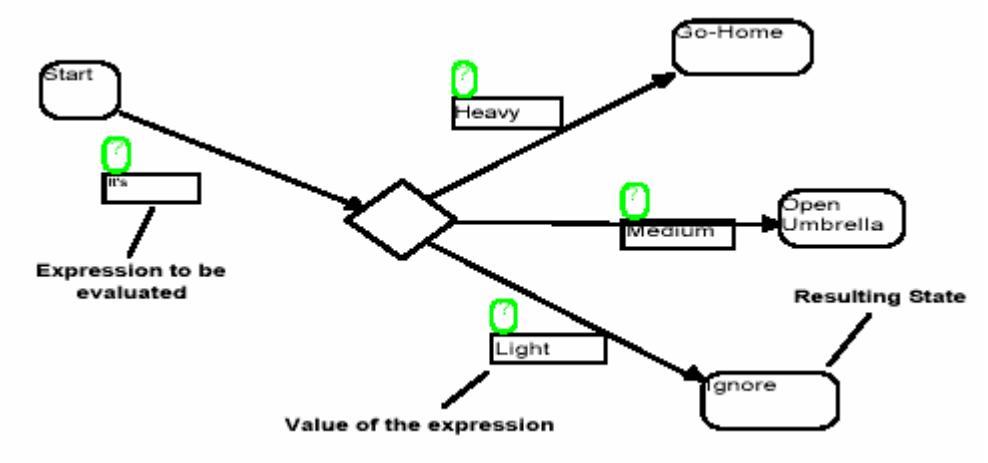


图14 展现我们在雨天的行为的扩展状态图

设计练习

这是一个有趣的练习。设计一个描述接电话系统的状态图。**如果我正忙着，我将隔两个电话才接一个，除非电话是我老板打来的。**同样，你可以使用老方法，纸和笔。或者直接使用 BetterState.

在开始练习之前，我们为您提供了一些有用的提示：

1、你在忙的时候都在做什么？

2、你系统中可能出现的状态有哪些？开始、等电话、有一个电话？

3、当你拿起电话，它是你老板打来的吗？

你完成的接电话练习的行为模型可能如图 15 所示。

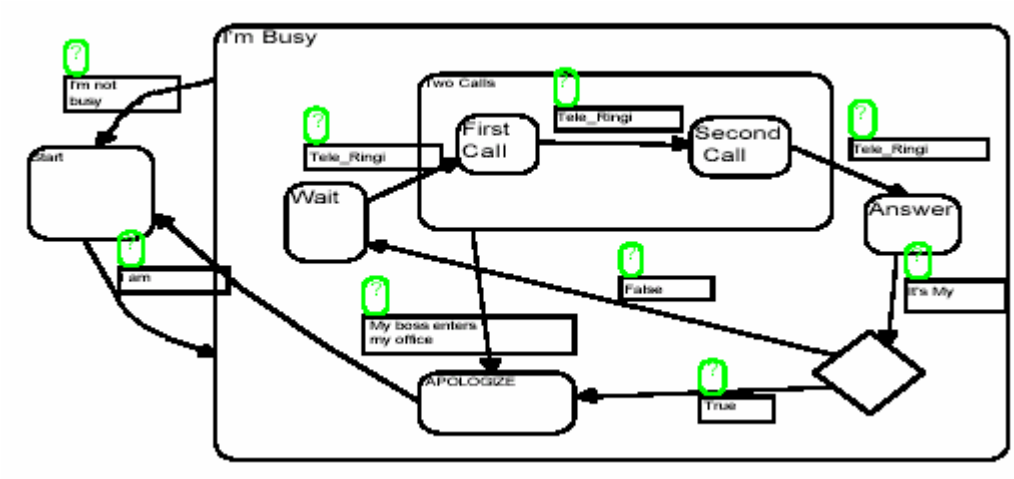


图15　接电话练习

一致性检查指南

当你初步完成状态图以后，你应该执行下面的一致性检查。

- 是否所有的状态都已确定？仔细的观察系统，看看是否还有其他可见的行为或其他条件没有被包括。
- 是否能到达所有的状态，你是否定义了一些状态，却没有路径能到达它们？
- 你能从所有的状态退出吗？只有两种类型的状态不能退出，一个是历史状态，另一个是结束状态。这些状态的用法超出了本教程的范围；有关它们的详细信息请参见用户指南第四章。

- 对于每个状态系统是否都能正确响应所有可能的条件？这是设计状态图时最常见的一个错误。设计者验证了正确条件时的状态改变，但忽略了意外条件下系统的行为。假设设计者为系统行为建模，期望用户在他的终端上按下某个正确的功能键，使系统从状态1改变到状态2，按下另一个功能键，使系统从状态2改变到状态3。但如果用户两次按了同一个功能键，或者按了其他的键，那该如何处理呢？

临界区

临界区（在状态组中用粗线框标记的部分），重叠的两个或多个线程同时访问临界区中的状态。例如，在临界区中，线程 A 中的状态 S1、S2 与线程 B 中的状态 S3、S4 重叠，也就是说当线程 A 访问状态 S1，S2 时，线程 B 不能访问状态 S3，S4，反之亦然。图 16 是展示临界区功能的例子。

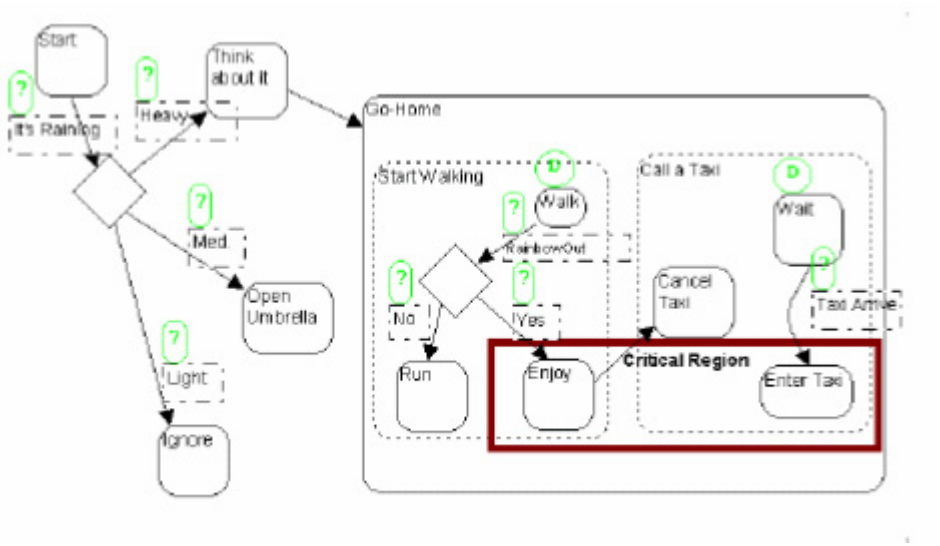


图16 临界区

图 16 展示了我们到目前为止讨论过的扩展状态图的所有功能。在前面的例子中，我们使用决策多边形为雨天的行为建模。图 16 是这个行为的模型。如果**下雨**，将判断表达式，**大雨**是一种情况，如果雨下得很大，我们就**回家**。

图 16 描述了我们将如何回家。我们可能走路或乘出租车。注意并行线程 *Start Walking* 和 *Call a Cab* 可能同时发生。

在这个设计中，我们增加了**临界区的设计**。临界区为同时访问区域内的状态设置了限制。在本示例中，如果乘车就不能享受走路回家的乐趣。

至此，我们讨论了设计扩展状态图的基础知识。BetterState Pro的扩展状态图也支持历史状态，交互状态，和其他很多重要的设计和调试功能。请记住这些是针对BetterState Pro 和 Better State4VisualBasic的前端设计。一旦你的初始设计完成（或在设计过程中的任何时候），你都可以使用BetterState所带的代码生成器自动生成代码。有关代码生成器和BetterState特有的动画调试功能、分析功能的详细信息请参见*BetterState Pro用户指南*。

现在你已经是扩展状态图的设计专家了，请进入第三章，看看 BetterState 是多么容易使用吧。

事件驱动编程

这一章我们将通过 Visual Basic 和 BetterState For Visual Basic 来设计一个简单的例子。这个例子的主要目的是在 BetterState 环境下设计状态图。对于用户来说，就是建立设计的 Visual Basic 前端部分。

在这个例子中我们使用 VB 代码生成器，因为这种代码生成器是每个版本的 BetterState Pro 都提供的。关于使用 Visual Basic, Delphi, 和 MFC 代码生成器的更深入的讨论，请参见*用户指南中事件驱动编程部分*。

考虑一个简单的例子：一个 VB 窗口，包含一个标准的 Visual Basic 定时器控件，命名为 *Timer1*；和一个自定义图片控件，用来表示一个交通灯，命名为 *Light1*。它可以是绿色或红色。假设我们希望信号灯在每次 Timer 事件发生时，在红绿之间变换。这种行为很容易用状态机来模拟，并可视化地表现为一张状态图，如图 17 所示。

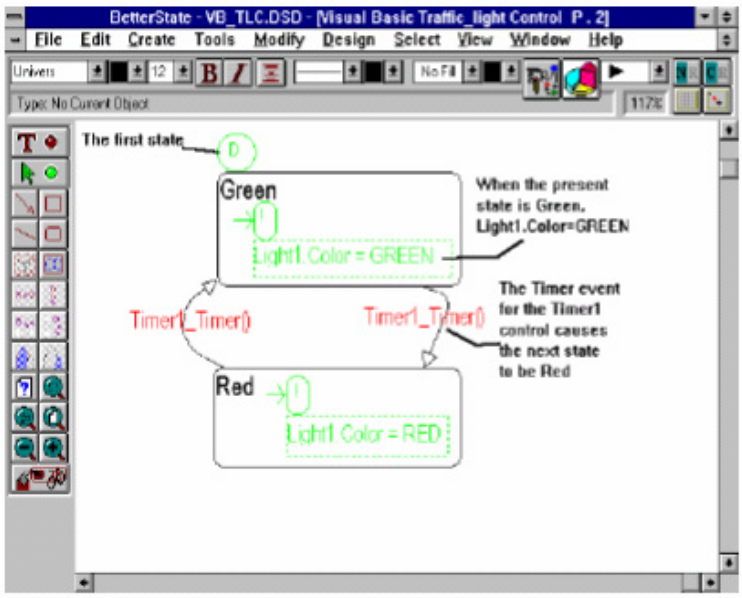


图17 简单的两个状态的交通灯控制器

状态机从 *Green* 状态开始，每次 *Timer1_Timer* 事件（固定时间间隔）发生，状态机在 *Red Green* 两个状态间交替转换，因此适当地改变 *Light1* 的 Color 属性。很显然，我们可以在 *Timer1_Timer* 事件中使用简单的 VB 代码来达到同样的结果，编写子程序，使用一个变量，命名为 *LightState*，它的值为 0 或 1，使用 *if-then-else* 语句交替的改变变量的值。

```
If (LightState=0) Then
  LightState = 1
  Light1.Color = 1 'RED
  Else
    LightState = 0
  Light1.Color = 0 'GREEN
  End If
```

实际上，这就是 BetterState 代码生成器将自动为你所作的工作；你只要画出状态图，代码生成器会自动生成代码。不过，图形化编程手段的好处不仅仅是这些，让我们考虑一个复杂一点的例子。

第一个扩展的例子如图 18 所示，增加了以下修改：

两个控制按钮，分别命名为 *NewTruck* 和 *NewCar*；

点击*NewCar* 按钮3次后*Light1*从Red变 Green，或者在点击一次或数次*NewCar*后再点击*NewTruck*。

图 18 展示了能够描述这种行为的扩展状态图。注意在图中增加了层次结构，连接 *OneORMore* 状态和 *Green* 状态的转换意味着如果点击一次或多次 *New-Car*（令当前状态是 *OneORMore* 的**子状态**之一）。又点击了 *NewTruck*，状态就会变为 *Green*。

这种层次设计功能允许任何 VB 用户练习掌握现代编程的最基本概念之一，即**自顶向下设计**。例如，它允许你设计一个仅由 *Green* 和 *Red* 状态组成的高层状态图，然后再设计 *Red* 状态的内容，或将设计 *OneORMore* 状态内容的任务分配给其他人。

注意扩展状态图是如何记忆**输入事件序列**，并通过可视的图来表现这些序列的。这是各种状态机的基本使命之一。

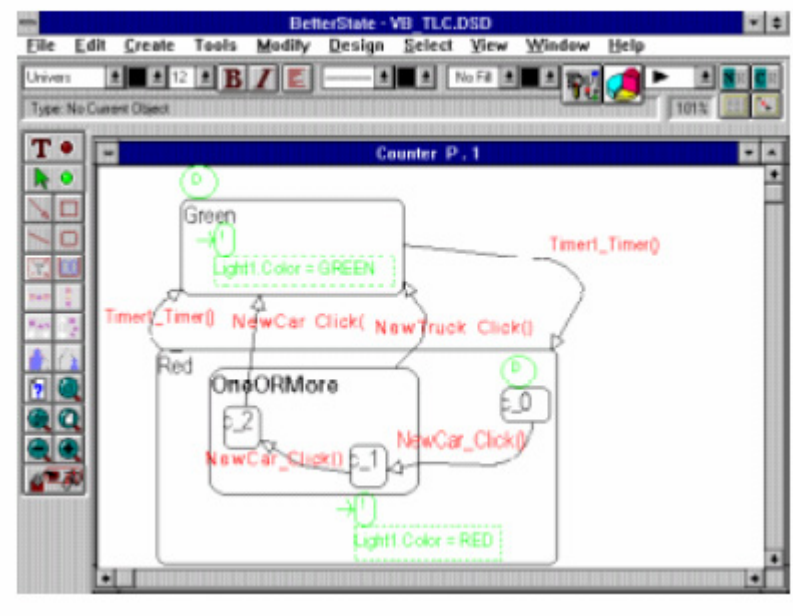


图18　修改后的交通灯控制器

扩展状态图在其他很多方面也很有用，它们提供的另一个重要服务是捕获并发，特别是部分并发的输入事件序列。

让我们更进一步加深我们的例子，说明如下：

Grid 控件，命名为 *Grid1*，增加到 VB 前端设计。

当 *Light1* 为红色时按下任何 Grid1 的键，接着点击 *NewTruck* 按钮，显示一个*照相机*图片。到 Light1 变为绿色时再隐藏该图片。

很显然，这个行为描述了另一个输入事件序列，它用状态图来描述很方便。这种行为与前面讨论的活动相互**独立**，可以用完全独立的状态图表来描述。然而，假设我们希望当 Light1 为红色，又点击了 *NewCar* 两次时*照相机*图片显示约半秒钟。现在两个扩展状态图设计不是完全独立的，第二个扩展状态图的设计必须判断第一个扩展状态图的状态。

有两个独立控制线程的扩展设计如图 19 所示。

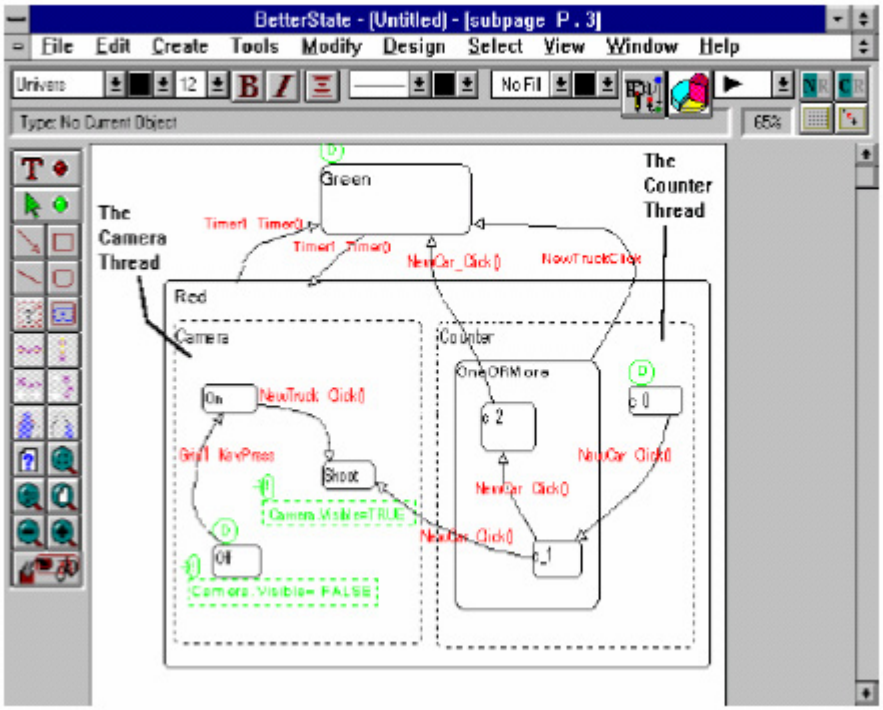


图19 有两个独立控制线程的扩展设计

图 19 展示了整个设计的扩展状态图；两个虚线方块，分别标记为 *Count* 和 *Camera*，称为线程。每个是一个压缩的子状态图，一个对应计数活动，另一个对应照相机活动。这两个线程相互独立运行。例如，当点击 *NewCar* 按钮时，*Counter* 可能从状态 *c_0 变化到状态 c_1*。这时 *Camera* 什么都不做，保持当前状态。**或者**，当点击了 *NewTruck* 按钮，它们可能都发生转换，从 *c_2* 到 *Shoot* 的状态转换是需要一些依赖关系的，当 Counter 记录下 *NewCar* 的两次点击时，Camera 必须改变为 Shoot 状态（使照相机图片可视）。

到现在为止，你应该领会到我们创建的游戏或者说是交通灯的原型和它的控制软件的意义了。这种事件驱动系统可能会变得很大，很复杂，当用单纯 VB 代码来描述会变得难以阅读。而且，调试一个类似程度的 Basic 项目也像一个恶梦，控制流在多个事件过程中跳转。调试这样一个由单纯 Basic 代码完成的包括并行或部分并行活动的设计非常困难，因为 Basic 代码是文本的，连续定义的；反之，并行的活动在概念上是会同步发生的。

因此，在 **WYSIWYG** (What You See Is What You Get 所见即所得)方式里，第一流的图形化编程工具应支持的一个重要能力是能够在图的本身调试图形化的行为。这种调试能力应具有两个主要特性：

动画，当接受输入事件时，图中的状态能够改变颜色；例如，当 *NewCar* 事件发生，*c_1* 状态应变为红色（或闪动，或者其他可视的突出显示方法），被突出显示；反之，*c_0* 状态应变回黑色（它不再是当前状态）。

Break-states，就像 VB 中的断点，当到达某个特定状态时，我们能够暂停调试任务。

关于扩展状态图

为基于事件的系统建模使用扩展状态图是一个强大并被广泛使用的方法。这个概念是由 Dr. David Harel 发明的，并在他的论文“扩展状态图，解决复杂系统设计的可视化方法”中详细描述。该论文发表于计算机编程科学（1987）。这篇论文描述了如何扩展传统的状态转换图，增加分层、并行和历史等功能，为复杂交互式系统创造了强大的可视化编程范例，在今天的复杂应用中，复杂交互式系统经常出现。扩展状态图主要适用于通讯、宇宙航空、汽车和制造业、工业自动化等广泛领域，并且快速发展为这些工业的状态"De Facto"标准。扩展状态图也是大多现代对象的标准组成部分，如面向对象方法论中的对象建模技术（OMT）和统一建模语言(UML)。

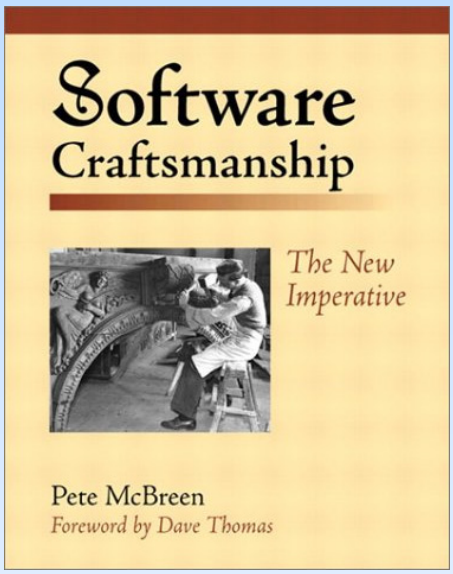
关于作者

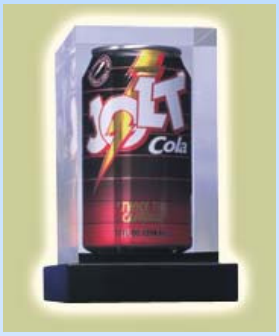
Doron Drusinsky 博士是 BetterState 软件工具的创始人，该工具应用扩展状态图方法论解决真实世界中的复杂应用。他曾经是 David Harel 教授的学生，Doron 在扩展状态图领域有十几年的经验，并发表了多篇论文和文章。他现在是 Integrated Systems 公司的资深工程师，他继续在扩展状态图领域进行创新研究和开发 BetterState 产品。

征稿

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

《软件工艺》





2002 Jolt Awards 获奖书籍

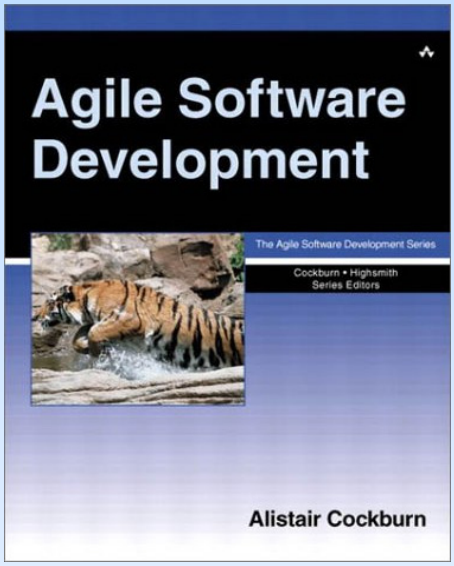
Pete McBreen

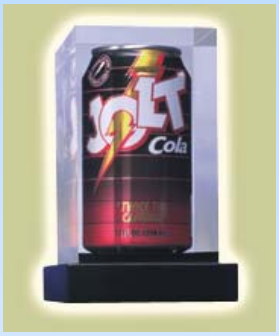
翻译：UMLChina 翻译组 Jill

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。

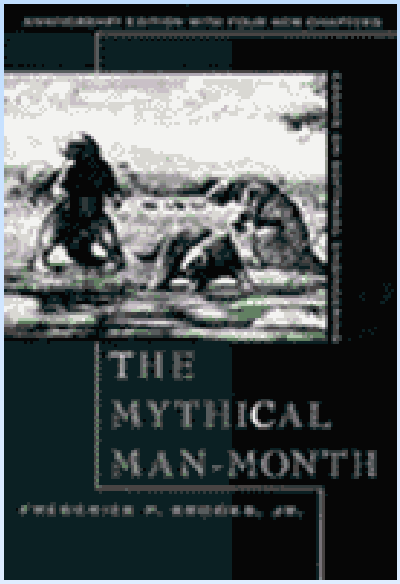
“是红酒。”我坚持道，并把标签指给她看。

“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

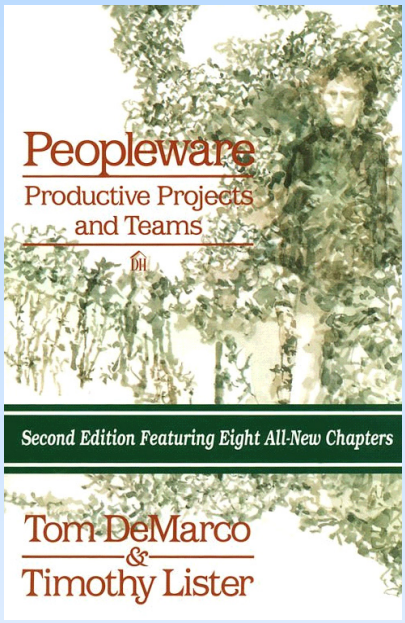
翻译：UMLChina 翻译组乐林峰

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

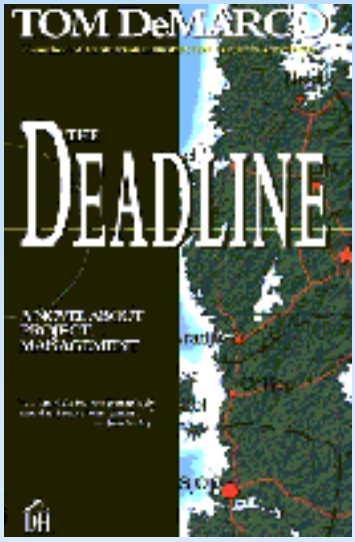
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

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！

保险系统的部分模式

Wolfgang Keller 著，liwenhua 译

查看评论

摘要

对于许多保险公司来说，要建立一个能够缩短产品周期，柔性灵活的保险系统可谓是一个挑战。虽然这个系统有着巨大的市场，围绕这些相同的问题开展了许多项目，但是这些项目似乎仍然有些扑朔迷离。实际上，这个问题没有答案。这篇文章收集了一些模式，他们解释了那些驱动保险系统运转的各个部分在设计上的基本规律和方案。

介绍

随着金融市场诡秘多变、竞争加剧，较短的产品革新周期将有助你在这个充满竞争的金融服务市场领先。服务提供商总想扩张到新的产品领域，所以允许交叉销售。他们尽力提供个性化产品，以便争取个人市场，为现有客户提供更好服务。同时，为了减少投资代价必须要有灵活柔性的系统。而有了灵活柔性的系统，在能够承受的维护开支下就能达到上述目标。

为了达到这些目标，金融业采用了许多系统。这些系统具有很大的相似性。这种相似性就是在银行业和保险业都存在的所谓产品驱动方法。这种方法是从传统制造业借鉴得来的，但是它与传统制造业又存在显著差别。这个差别就是你在生产订单（保险单）的同时就进入了系统的投保服务。毕竟金融产品是无形的。

其它工业的类似情况

本文收集了产品驱动的保险系统中的一些模式，这些模式设计得十分灵活，可以在保险和金融业的许多系统中看到。也许可以在其它工业中找到这些模式的身影，但是，在此将不解释如何不需要其它领域的实际项目背景知识，它们就可以应用在那些领域。如果有人从其它工业也得出了相似的模式，我们可以比较它们，匹配它们。

“产品树”是一种核心模式——这种模式试图以树的结构来呈现产品——其原型来源于制造业，只是到了最近才被应用到保险业（大约 10 年了，参看 Sch96）。服务提供者希望按顾客需要的方式，使用一系列预定义好的单元来配置产品，这种想法在制造业和其它行业中也可以看到。还有好几种称之为批量定制的方法，它们严重依赖信息技术来传递批量定制的产品[And+97]。在计算机行业有一个很典型的例子——DELL 或者 VOBIS 可以定制

个人计算机。在布匹行业也有同样的例子（Levy Strauss 和其它）。

个人非实物产品同样可以在电信业找到。账单计划也可以使用与产品树和产品服务器类似的技术从预定义的构造块中选取来进行配置。

这篇文章并没有全部使用模式语言去表达模式。实际上，有很多方法可以对产品建模，我们这里只使用了其中一种。

现有的保险业参考模型

许多对模式感兴趣的读者可能听说过 IAA（IBM's Insurance Application Architecture），并奇怪这里怎么会与之相关呢？其实 IAA 价格不菲，除了尚可利用的二手资源外，我根本没有直接接触过它，也就无从详细讨论这种相关性。对此我表示抱歉，不过，这也不是我的错。我所能借鉴就只有 VAA（VAA95）了。VAA 是德国保险业参考的一个公开模型，我总是尽可能参考它。

驱动保险应用设计的一般动力

驱动这些模式产生的因素是明确的，它们包括：

1. 市场中的时间因素和系统的灵活性：在欧洲，近几十年来，政府一直严格对保险市场进行规范。但是随着全球市场和欧洲市场不规则因素的出现，保险公司发现他们又处在一种新的竞争空间中。银行，其它金融服务提供者，来自邻国的保险同行，以及其它机构等等，这些新的竞争对手不断涌现。时间在市场中已经扮演了一个决定性的角色。 那些有能力更快设计出新产品并把产品推向市场，同时又能够提供个性化服务产品的公司将比那些好些年才向市场推出一个新产品的公司更具竞争优势。

2. 新产品和个性化产品：Michael Porter 分析了几种竞争战略（Por85），一种是价格竞争，一种是质量竞争，还有一种是综合服务竞争。如果你想开辟新的市场，或者想成为为客户眼中质量信誉的佼佼者，在更多的客户中拥有更多的产品无疑会帮助你实现梦想。在工业化早期，各方面的开销相互冲突，要想做到这点很困难。但是，随着现代计算机技术的日新月异，以较低开销提供客户个性化产品已经成为可能，所要做的就是把原来产品的标准元素模块同客户个性化产品要求结合起来。谁能为客户提供个性化产品谁就具有众多的竞争优势。

3. 服务点和节省开支：有许多策略可以增强竞争力。其中一个策略就是为客户提供贴身服务。许多保险系统还纯粹实行着老的办公制度，大量的公文在多个办公室和销售代表之间流转。节省开支策略就是瞄准公文流转所带来的诸多烦琐，努力精简公文处理的中间步骤。保险系统被部署在更靠近客户的地方，通过运用互联网络和个人电脑，销售代表的膝上电脑可以与事务处理中心紧密集成。

4. 开发成本：保险业提供纯粹的无形服务，既没有硬件也没有可以触及的实物。一个保险公司可以显著地减少数据处理过程的开支，这一点跟制造业企业减少劳动力成本和生产原料成本没有什么两样。做到这一点意味着竞争优势。因此，在开发保险系统的时候，在提供更多功能和灵活性的同时，也要追求开发成本得到降低。

5. 一个保险公司的传统结构对应于产品包和开发开支：就拿我们涉及到的这个新保险业来说，保险公司有那么些时间是繁忙不堪的，是他们的需要扶植了数据处理技术的发展。保险行业的遗留系统和组织一般都是围绕产品分类而建，举个例子，譬如你会看到人寿保险，健康保险和财产保险分而治之，这些反映了公司的组织机构和处理过程的分布。如果按照这样来设计系统，就会存在大量功能上的冗余，例如，财产保险中的客户申请处理系统和汽车保险中的客户申请保险系统绝大多数是相同的，而健康保险和人寿保险都是以人的身体为保险对象。所以围绕产品分类来开发系统会导致不必要的开发开支。从另一方面来说，如果所有的领域专家还是旧的观念，尽管对部分产品很精通却很少知道其它产品，那么，要开发同时能够满足所有产品的系统也是万分困难的。现实中还经常存在分歧，不同部门并不都愿意同心协力为所有产品类别建立系统。

模式

这里呈现的所有模式具有相当强的内在联系，对于你将要应用的领域，这些模式可以组织成一些章节来讲述。

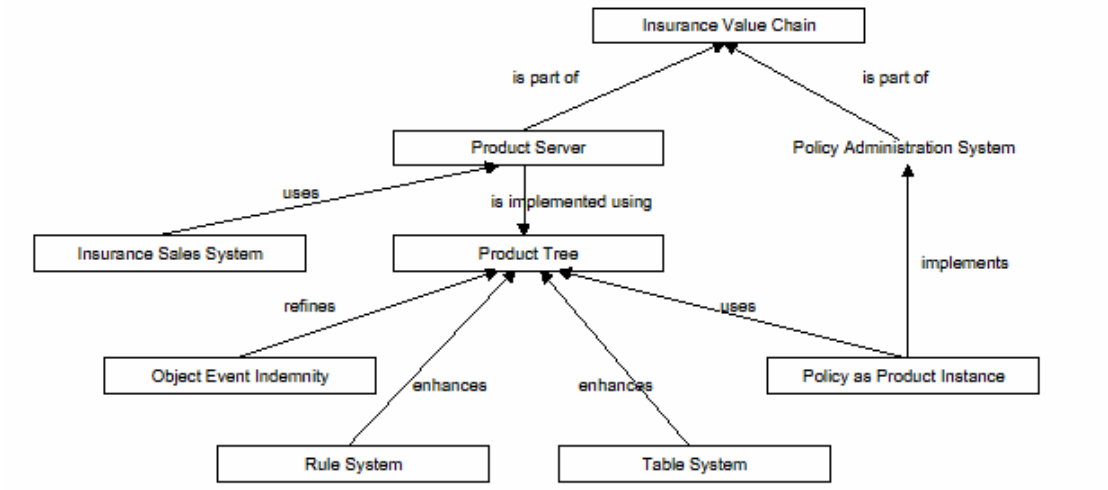


图 1 模式集合的指示图

这里给出的模式只是保险领域所涉及的众多模式中的一部分，如果把它们按照某种逻辑组织起来，将有助于更好地理解。因此我们把这些模式安排到如下章节：

保险系统的顶层结构：

大多数保险系统都涉及如何处理顶层子系统的结构问题。在这一章，将要描述的模式有：

保险价值链，产品服务器和作为产品服务器典型子系统的规则系统。

保险产品模型：这里包含对产品服务器的产品进行建模有意义的一些模式。这一章描述的模式有：产品树和对象事件保证。

策略：

这一章涉及如何实现保险策略，这里只提供了唯一的一种模式：作为产品实例的策略。

分布式的保险系统：在一个典型的保险系统中，假如我们拥有一台虚拟的超级计算机，这台超级计算机把保险组织中的每一名员工紧密联系一起，数据存取无需时间，网络带宽无限制而且免费，那么你会发现其中一些子系统是不必要的。不幸的是，在现实中，依然要开发这些子系统。那么，这里将提供一些用于分布式处理的模式，比如：保险销售系统，或者一个表格系统。

保险系统的顶层结构

在这一章，我们将介绍在多数现代保险系统中可以看到的、依据保险价值链而存在的顶层子系统。保险系统中一个非常重要的产品部件就是所谓的产品服务器。典型情况下，作为一个产品服务器，它存在一个用于处理产品规则的规则系统。

模式：保险价值链

例子

你被分配去为新的保险办公系统设计结构。在现在的系统中，你找到一个一个根据产品分类的系统分支，如人寿保险，财产保险，汽车保险。你发现，这些现存系统中存在大量交叉重复功能，尤其在申报处理系统和政策处理系统。你也发现要增加一个新的产品需要做的工作十分繁琐冗长，它以每年增加两个新产品的速率进行着。这样低效的原因源自于系统管理政策部分和管理申报处理部分的产品知识混乱，更多其他的产品知识还存在于系统其它部分。

问题

对一个保险办公系统中的大部分商务功能和商务对象来说，什么是一个好的领域构架呢？

约束

系统应该这样来构造，它必须能支持你的竞争战略。我们已经讨论了工作中的市场约束。

方案

根据保险中的价值链构架领域结构。产品－政策－申报处理－销售和市场－客户服务，再加上帮助系统，比如伙伴子系统，保险对象子系统，进出支付系统。

结构

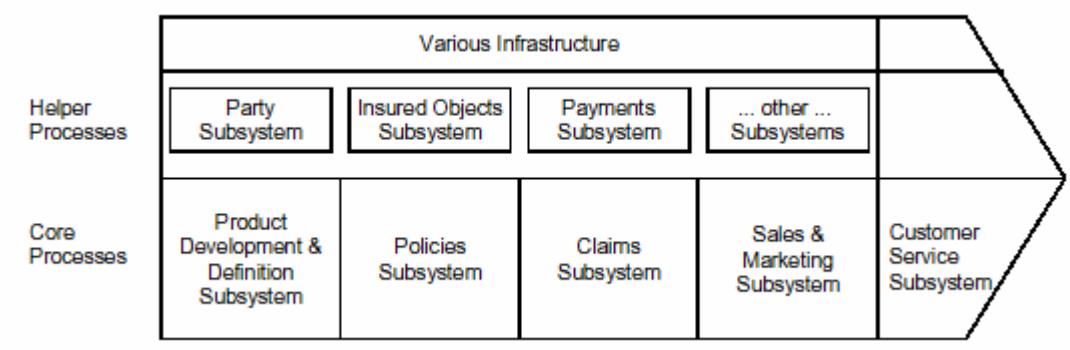


图 2 依据保险价值链的保险系统结构图。

类似于 Porter 的价值链模型，这种结构被保险业所采用。

通过子系统来进行价值链建模：

产品开发和定义子系统：产品为其他子系统所使用前必须给予定义。产品开发和定义子系统负责为其它系统提供产品定义。产品定义由系统其它部分解释。

政策参数子系统：负责存储政策参数，支持所有开展商务活动的用例。

销售和市场子系统：处理面向客户的产品销售

客户服务子系统：有助于向你的客户提供你期望的额外服务，这些服务是在投诉系统所规定的服务之外的。

要建模好所有的商务功能，你需要像下面一样的帮助子系统：

伙伴子系统：存储你公司所有合作伙伴的信息。

被保对象子系统：存储有关被保和投保对象的信息。

支付子系统：处理收入和支出的现金流。

通常你会发现还有好多系统，比如通用名册保持（general bookkeeping），管理信息系统和数据仓库，人力资源系统等等，这些系统不单是针对保险系统，而是普遍存在的，所以我们这里不再进一步讲述。

结论

并非只要采用保险价值链模型就能得到一个灵活柔性的系统和缩短的市场开发时间，你还需要使用更多类似产品服务器、产品实例参数这样的模式来帮助你实现目标。

采用模式你可以建立一个满足所有产品类的系统，但是不能一步到位，多数情况下你需要先从一部分产品类开始，然后逐渐扩充系统以便从遗留系统迁移过来。如果这样，你就避免了重复投资，从而减缩开发费用。

上述结构是为内勤系统设计的。它并不会单独影响你在服务上的态度。

采用上述结构并不意味着能减少通讯的层次，节省开支。要想达到这个目的，你需要启动业务过程重组并采用工作流系统。

实现

保险价值链是一个领域构架模式，要实现一个现实系统你需要一个应用构架。目前，多数领域系统都通过一个树状层次构架来建造。工作流系统也经常用到。通常，你无法在一个地方一下子实现整个系统，或者一下子实现虚拟的单个系统。目前的网络带宽和性能还不如人意，基于这些考虑，你不得不把系统分成内勤和保险销售两个系统。

变种

你常常会发现产品系统和政策参数系统联合工作方式的差异。将要谈到的产品服务器模式与上述两者都不同，而其它像 UDP 之类的框架，没有提供两子系统之间的子系统边界，但是却把政策参数当着产品实例，并且贴切地耦合了两个系统

相关模式

一般，产品子系统被当作产品服务器来实现。

一个政策参数子系统通常采用用户定义的产品框架来实现，这样可以充分利用组件模式和 Type Object 模式的优势。通常，你可以把政策参数当成产品实例。

已知应用

上述构架在保险业几个领域构架，比如 Phoenix 构架，IAA--IBM 的保险应用构架或者 VAA(VAA95)有更详细的描述。

进一步阅读

尽管保险市场巨大，但是有关保险系统的资料还很少，在这一点与其它市场领域相反。有关基于价值链构架的基本概念和原料产品的相关概念可以参阅《Innovative Gestaltung von Versicherungsprodukten》。以上的已知应用

有大量的文档，描述了相近的领域构架。来自 IBM 的 IAA 也是一个巨大的领域构架，但是它已经注册，不向公众公开。

模式：产品服务器

假设你想建立一个产品驱动的保险系统，这个系统一般具有一个后勤系统，用以处理提议，管理政策参数，应付投诉。另一方面，你还需要为用于销售的个人电脑，诸如内部供给或者保险经纪人支持等其它系统提供产品定义。

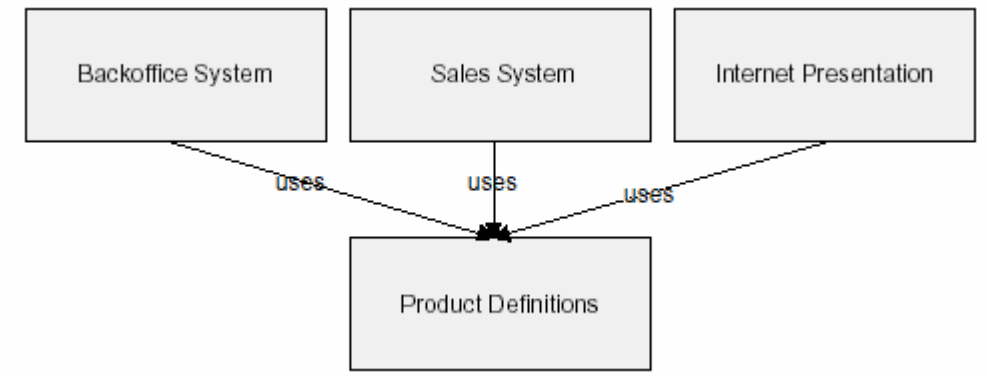


图 3 多个系统需要明确的产品定义

问题

何处定义你的产品知识以及怎样发布？

约束

除了上面提到的，还有几个因素也需要考虑：

平台独立性：后勤系统一般运行在 OS/390 机器上，销售 PC 一般运行 Win95 或者其它个人操作系统，互联网上的客户端可能运行 Java 应用，或者通过嵌入式 HTML 与某种语言写的服务器端程序组成应用。你的产品定义必须在不同平台都可以访问。

界面设计：宽界面容易导致维护困难和软件结构过于内聚。狭窄界面则适合大量客户在不同平台的情况。

产品知识封装对最佳用户界面设计：一方面你希望封装所有产品知识。这样的好处是，例如所有的对话框只解释一个产品定义，而且能够自动改变适应新的产品定义。缺点是自动化的界面缺少美观。这种方法适合内勤系统对话框，但是不适合销售对话框和销售员笔记本电脑上的多媒体产品展示。

需求差异：尽管内勤系统和销售系统都需要使用相同的产品定义，但是它们对产品定义的视角却有差异。对于存储所有政策参数的内勤系统，你既需要老的产品定义也需要实际的产品定义。对于销售系统你只需要你实际销售的产品程序。一个产品定义知道更多的产品，包括那些正在开发中的和销售程序中尚未出现的，都需要为这些系统提供不同的视图。

方案

在产品服务器中封装产品知识。运行期间，在可移植的产品运行系统中封装它并且使用外观提供视图

结构

你在一个典型的产品服务器中会发现如下组件：

产品编辑器：为定义产品的用户提供易于使用的界面。产品定义可能稳定于产品模型中。

固定的产品定义：代表你的产品的固定商务对象。如何最好地组织这些对象将会在产品树和对象事件赔偿两处予以讨论。

前两个组件也可能被组织在一个普通的面向对象树状层次机构中。

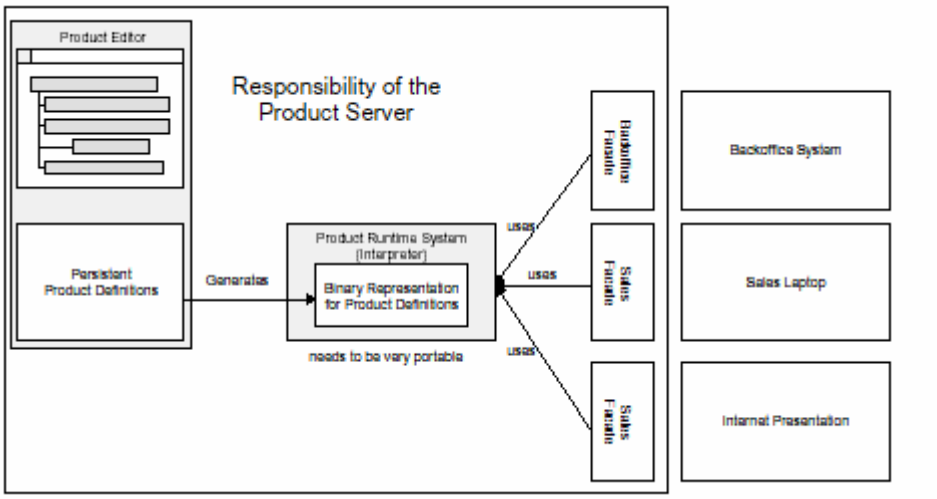


图 4 产品服务器的结构

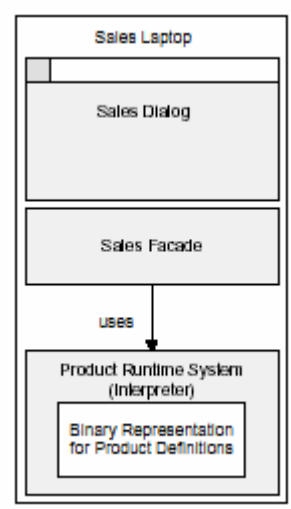
商业对象代表了稳定的产品定义，很少被移植。因此我们需要更多的组件。

发生器：负责从稳定产品定义中产生可移植的代码。

产品运行时系统：可以运行在 OS/390 上以及 Window 平台上，负责解释上面产生的代码。有些系统采用 ANSI C 为产品运行时系统编码以保证系统的可移植性。

如果你想在产品运行时系统中提供一个非常狭窄的界面，但是又不想处理过多的原始界面操作，或许你可以实现：

前端部件：为你的产品定义系统提供不同的视图



所谓的产品服务器包含所有组件的事实并不说明所有这些组件必须运行在单个机器上，也不意味从客户/服务器系统模式来说产品服务器就是服务器。

要了解这一点请看图 5：销售系统的配置。销售对话框通过前端部件使用产品运行时系统。而所谓的产品服务器的剩余部分，产品编辑器，产品商务对象，生成器可能运行在不同的机器不同操作系统和技术架构。

图 5 销售员膝上电脑

结论

保险系统使用产品服务器来提供优良的灵活性和平台独立性。相对于只使用活动对象模型，使用产品服务器可以在你的销售系统、网站等等为同样的产品定义赋予不同的视图。

缺点在于创建过程。与使用活动对象模型和反射相比，使用产生器具有某些缺点。但是这个缺点可以在一个方面为某种事实所弥补。通常正在编辑的时候，你并不希望你对产品的修改同时为其它地方所见。保险产品就如同代码一样也需要经过测试，发布，版本化。一个系统使用一个简单的活动对象模型，产品开发，测试则笼统模糊，生产没有效率，容易陷入困境。

其它结论是：

平台独立性：通过使用一个可移植的产品运行时系统来达到平台独立性

良好的界面设计：为产品运行系统设计一个非常狭窄的接口或许能取得良好的界面设计

最优界面设计：包含某些产品的专门知识，并涵盖其趋势。这是不凡的设计

需求差异：可以通过在产品运行时系统使用不同前端部件来兼容需求差异

实现

通过在树状层次构架中使用活动对象模型，可以把产品编辑器当作普通面向对象应用来实现。

相关模式

要实现产品编辑器，你应该使用产品树，而产品树用对象事件补偿模式组织。你还需要把表格系统和规则系统集成到其中。运行时系统可以采用实现虚拟机所采用的模式。前端部件用来为产品定义提供不同的产品视图。

要为产品编辑器实现产品商务模型，可以采用活动对象模型和发射。

已知应用

采用产品服务器的这种思想可以在许多保险系统中找到。VP/MS 中可以看到部分，如采用 CAF 的产品定义。EA Generali 实现的两个项目也包含了产品服务器的思想：KPS/S，一个正在开发财产保险项目；Phoenix，一个快要完成的人寿保险系统。

模式：规则系统

举例

假如你实现了一个产品编辑器，这个产品编辑器允许你定义产品结构，那么你需要一种机制来为树的节点元素从一种属性导出另一种属性，比如评估政策参数。

问题

你如何来实现真实性检查或者对产品树进行政策评估这样的功能？

动机

系统管理员的使用：如果你打算让你的系统管理员不编码就定义新产品，你必要提供一种解释语言，这种语言能够把产品属性，性能估计连接起来并且可以实现表格查找。如果你按照这个想法去开始你的工作，你很早就会发现这需要一种全新地编程语言才行。有关这些的进一步讨论可以参看 WWW 上的《通过编程实现可定制》"(http://c2.com)。那里的讨论与《硬编码对脚本编程语言》相似：一般硬编码（尤其在 Smalltalk）比试图在 Smalltalk 上再建立一种 Smalltalk 容易。

代价：草率地开发一个规则系统是昂贵的冒险。给一些系统管理员培训 Smalltalk 一般比为他们开发一种可定制的编程语言代价少。另外我们知，道，像 Excel 那样，按可以接受的代价来实现像公式解释器那样的功能也常常为系统管理员乐意接受。

追溯机制的完成：如果要建立一个规则系统，最好使之能够完成产品定义需要的所有属性导出，否则，你就得编程实现。

灵活性：对于像 C++那样的编译语言，要编辑/编译/链接才能完成，对系统管理员来说一个 Excel 表单似乎灵活的多。

测试和调试产品：如果你要在你的产品定义中实现脚本语言一样的编码属性追溯机制，你需要一些其它编程语言环境，以及额外的工具，比如可定制调试器。

方案

建立一个类似于解析树中的语义功能的属性追溯机制，或者 Excell 表单中的单元计算，在运行时解释。

结构

参看图 6：保险产品树。如同编译建造，你可以使用某种编程语言（如 Lex/Yacc)为语义功能编制程序，或者你也可以用各种不同脚本语言定义一套完整的工具。

注意，这跟基于规则的系统，比如人寿保险中的风险评估系统，没有太大的关系。在基于规则的系统中，你经常可以看到专家系统（如以 Prolog 实现的）。我们这里谈到的规则系统在一个相当确定并且可预期的基础上实现商务规则。

结论

结论如同实现途径一样多种多样，究竟如何使用取决于你的动机。这些方面跟编程语言设计非常相似。这里给出一些示例，在我所知道的系统中，这些示例将要发生或者已经发生：

不完全规则定义语言：这种情况下，你将需要部分编码来完成属性追溯和定义，这样的结果是，相对它所提供的灵活性和可定制性，你付出了太高的代价。

超完全规则定义语言：这样一个非常复杂的语言容易出错并且相当昂贵。与其这样还不如使用解释性编程语言。

相关模式

需要相关模式来实现产品树。解释器和虚拟机经常用来实现规则系统。

已知应用

UDP 一文勾画了如何用 Smalltalk 实现一个规则系统。VP/MS 使用了一个版本，这个版本非常像电子数据表。Phoenix 混合使用了硬编码和规则。

保险产品模型

这一章包括了一些模式，这些模式在产品服务器建模保险产品非常有用。这些模式是：产品树和对象事件赔偿。建模产品有很广泛的方式，对象事件赔偿仅仅只是其中一种。参看[Sch+96]可以获得更多信息，或许将来我们还要从中挖掘更多的信息应用到工作中。

模式：产品树

示例

你决心顺着保险价值链去建立一个产品服务器，在设计之前，你需要知道如何建模保险产品。

问题

什么是好的保险产品展示呢？

动机

除了驱动保险应用的常规因素，下列因素也需要考虑：

涉及用户同灵活性和通用性的对比：你需要采用用户能够理解的模型，而工程上使用的是非常难以理解的抽象数据模型。

重用和灵活性：好的方法能够在程序模块级别上实现重用。理想的方式是在产品开发过程中形成模块库，这些库可以直接重用或者经过组合以形成新的产品，就像制造工业一样。

方案

把你的保险产品建模成一颗树，就像平时组装自行车一样。

结构

树的节点是产品或者产品组件

每个节点用属性描述

加入保险费计算功能作为导出规则，模拟编译器解析树的语义功能，或者写你自己的规则系统（参看 Ralph Johnson 的关于 UDP 的论文）。

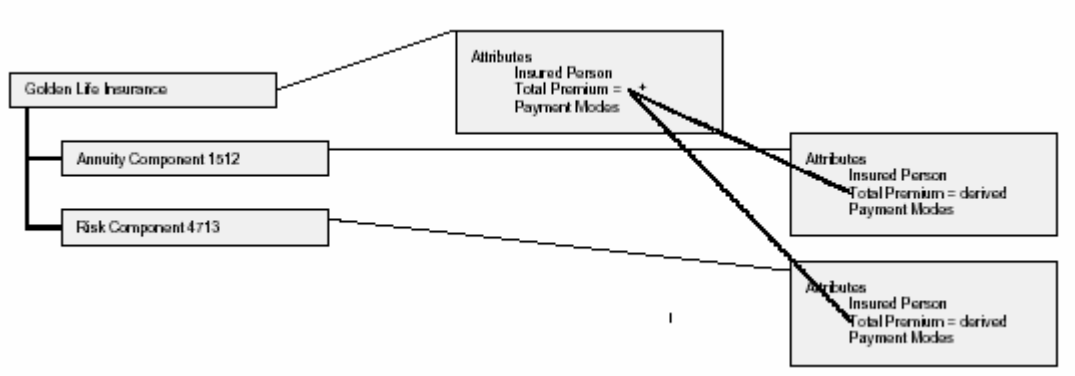


图 6 保险产品树(P15)

结果

产品灵活性：一旦你明确了保险产品的构成元素块，设计和建模新的产品将变得容易和直接得多。

用户包含：开发产品服务器的实际工作显示了，具备一些额外知识的领域专家就能够建模产品树。

重用：一旦你定义了产品构成块，你就能在无论是新的还是现有产品中重用。

实现

严格实现这个设计还需要作出一些额外的设计考虑。产品树建模并不意味你能最好地组织这些产品树，也不意味你有最好的产品模型设计。请参看对象事件赔偿获取一些技巧。

通常你会用组件和 Type Object 模型将保险政策建模为产品实例。组合这些模型和导出参数就可以得出完整的框架，这个框架在 Ralph Johnson 和 Jeff Oakes 正在进行的工作中有描述。

有两个非常基本的替代方式用于设计产品树的运行时系统

1. 第一种方式假设产品编辑者工作在同一个数据库上，并且在生产性系统中当作活动实现来建模。这将引发 Ralph Johnson 和 Jeff Oakes 所描述的活动对象模型。
2. 第二种方式假设开发空间和产品空间严格分离，这就是产品服务器方式。

变种

经常构建模块，那么就会有确定的属性用来共享。考虑下面两个产品元素：养老金组件和风险人寿保险，你会发现两者之间投保人都是相关对象的属性。在一个产品集合中要结合这两者就需要确定两个投保人的属性。这就产生了 DAG 表示（直接非循环图表）而不是树型表示。

相关模式

模式是根据保险价值链实现系统的关键。前面已经讨论过用来实现这种系统规划的模式。

已知应用

这种方法在保险业中非常常见。比如 CAF 的 VP/MS 产品定义系统。EA Generali 开发的两个系统，如正在进行的财产保险系统 KPS/S 和快要完成的人寿保险系统 Phoenix 也都包含产品服务器的思想。其它保险公司也正在或者已经采用了这种方式。

进一步阅读

模仿制造业部件分散制造的原理来建造保险系统的思想已经由 Paul Schonsleben 和 Ruth Leuzinger 在一定层面讨论过了。

Ralph Johnson 和 Jeff Oakes 讨论了如何用面向对象方式实现同样的系统。

模式：对象事件补偿

示例

你已经决定把你的保险产品建模成一个产品树，于是你开始工作了，你的同伴与你合作。当你把由不同人建立的两个系统结合到一起，你发现单单产品树范例就不允许你随意重组从前的构造块。

问题

使用树状结构来建模保险产品，有什么好的方法呢？什么是建模保险产品好的抽象呢？

动机

质量和可理解性：多种编程语言的存在，仅仅一种方法实现保险产品树会带来模型结构不良，不易理解，从而导致维护困难。相似的情况在许多领域也存在，都有一个相当高层的抽象的建模范例，比如对象建模，实体关系建模等等：你需要一些规则或者领域分析模式用以缩小解决方案的范围，另外还需要一些规范用以减轻建模难度。

重用

也要求建模规范。你的保险产品组件建模得越规范，以后在其它产品中重用就越容易。

方案

使用对象事件补偿模式建模保险产品。每个保险产品确保一个或者多个保险对象。如果那个对象相关的特定事件发生了，客户有权申请一定的补偿。

结构

你的产品树应该是如下模样：

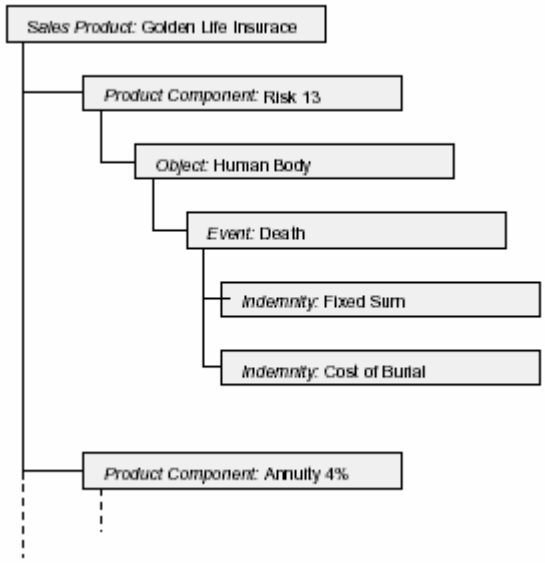


图 7 按照对象事件补偿模型组织的产品树

每个对象具有递归性：一个对象可能包含其它对象，一个产品可能包含其它产品，等等。但是这个树型结构的基础是十分稳定的。

结果

生产力和重用：如果每个开发者都坚持相同或者相似的建模模式，那么大家对产品结构可以取得共同的理解，这样将可以获得更高的生产力和更好的可重用性。

相关模式

对象事件补偿模式只描述了高层产品结构，这里需要更细化的模式用作诸如属性的定义，语义规则，如何查表等等。要挖掘这些模式，你需要参考一些由不同人或者不同公司建立的产品模型。由于产品模型是竞争的机密信息，我们需要较长的时间才能等到它公开。

进一步阅读

这个主题上没有书籍也没有文章予以介绍，但是像 CAF 这样的产品提供商提供培训和相关课程资料。

政策参数

这一章介绍如何实现保险政策参数。这一章只有一个模式，产品实例化政策参数。

模式：实例化政策参数

别名

镜像部件列表

示例

很高兴你刚完成了你的产品编辑器。你用了组合模式来实现它，并在界面上使用了分层的列表框。你已经在叶结点键入产品数据，对象事件补偿也加入了一些。

问题

如何表示保险产品政策参数

动机

一般动机：最大的动机是驱动保险应用设计的动机。关键是灵活性。如果你不能在后勤系统处理新产品，最好的产品服务器也是无用。因此，处理政策参数的系统应该像产品定义工具一样灵活。

方案

创建产品政策参数实例，并用 Type Object 模式来完成，那么政策参数就能够反映树型产品定义。

结构

一些产品实例被当作原型保存在政策参数工厂中。用真实的数据实例化这些原型就形成了政策参数。

结果

灵活性：如同最具反射能力的系统一样，你将获得一个非常灵活柔性的系统，这个系统能够使你同产品定义保持同步。

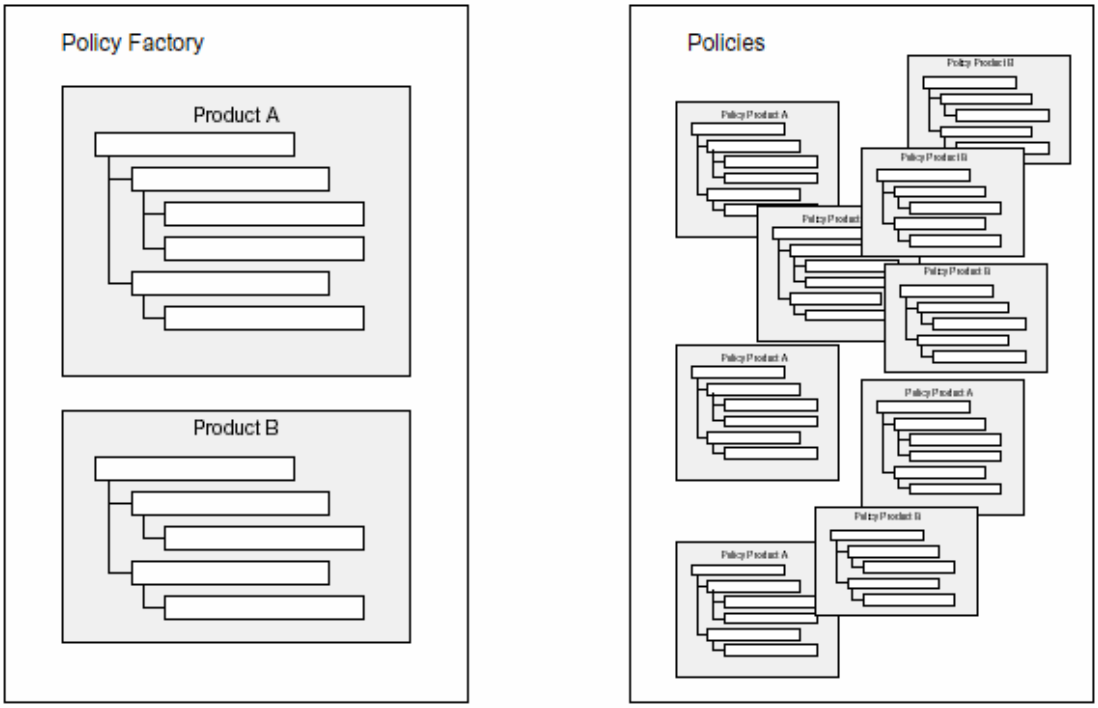


图 8 作为产品实例的政策参数

可移植性：你必须提前考虑到系统的可移植性。假如你用这个模型在一台主机上实现了活动对象模型，如果系统不具有好的移植性的话，你可能需要花相当多时间在销售员的膝上电脑实现同样的系统。

性能：如果你不经常注意性能的话，就如同多数发射系统一样性能可能变得很差。如果你有一个 6 层深度的产品树，并用一个本地数据库存储，要想正常工作，你需要提取比如说 100 条数据库记录才能做到。要避免这种情况，你需要建立一个具更智能的数据库映射。

变种

实际上，你可以找到两种主要的模式变种。

单一数据库系统：这种情况下，产品数据定义系统和政策参数组件在单一的一个数据库实现。你可以使用产品编辑器在政策参数工厂中创建新的产品原型。这种情况下，政策参数工厂也是产品定义系统的数据池。

这种方式的不足：

过分紧密地耦合了产品开发和实际产品的生产。优点是，一旦你定义了产品，这个产品就马上可以在产品系统中使用（对于测试和发布来说，这实际是它的缺点）。

多数情况下，没有办法在一个销售员的膝上电脑上使用单一产品定义数据源。

如果天真地坚持这种商务对象，你会陷入巨大的性能麻烦。

去耦方法：中央政策参数管理系统和保险销售系统共用一个产品服务器：使用产品服务器，把政策参数工厂（参看图 9）作为产品运行时系统的客户端，并且把产品定义对话框从图 9 移去。

这种方法的缺点：实现起来更复杂，不直观，并且有些冗余。

优点：产品定义系统与政策参数系统分离，有更多调整性能的余地。你可以建立其它客户端，例如保险销售系统。

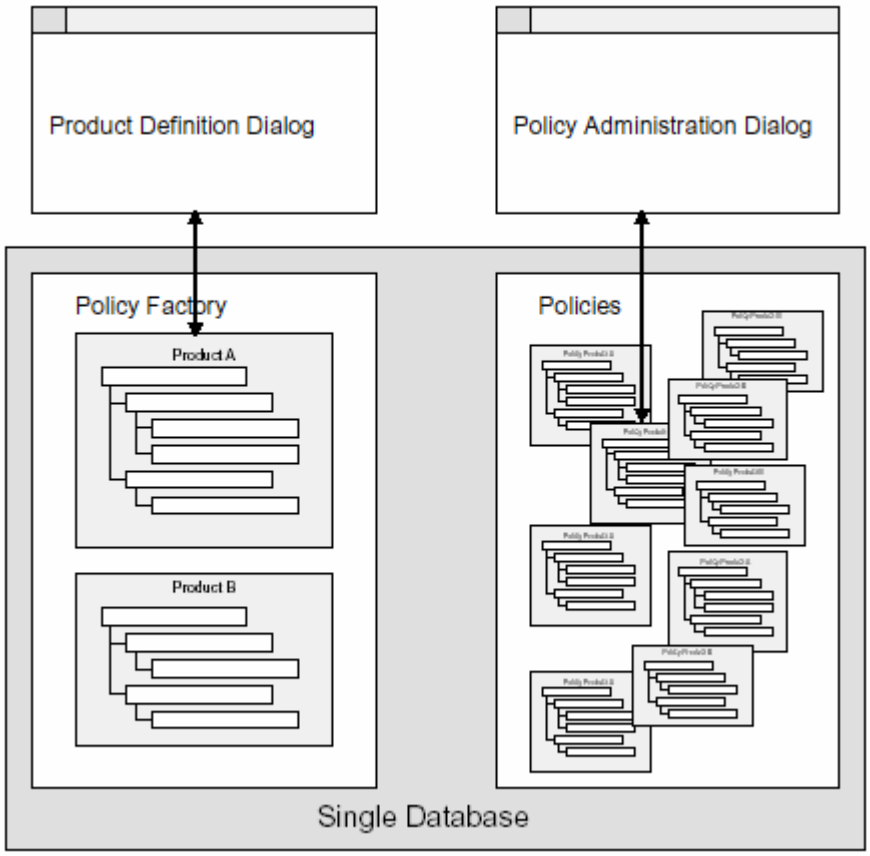


图 9 单一数据库产品定义和政策参数管理系统

相关模式

这种模式是组合和 Type Object 模式在特定环境中的递归使用。另一种叫法是活动对象模型，它是发射模式的变种。

你在产品定义系统中为产品树组件指定类型是第一次使用 Type Object。第二次使用是将一个复杂的树型实例（产品树实例）当作保险政策参数原型（参看图 8）。

使用复合类型建造产品树和政策参数树。一个具体的政策参数树是一种产品树。

已知应用

Generali Munich 的 PVS 就是在一个主机上使用单一数据库。UDP 系统虽然没有明确说使用了哪种变种，但是看起来更像使用单一数据库。在 Generali 的 Phoenix 项目也使用了单一数据库的变种。

我们还没有看到去耦的实现，不过很可能我们后面的某个项目中就会实现它。

进一步阅读

关于在建造产品树方面的面向对象设计考虑的具体讨论，可以参考 Ralph Johnson 和 Jeff Oakes 的文章，他们给出了这个主题深层次的模式。本文就如何用解释器计算导出属性给出一个深层的讨论。

分布式保险系统

在一个典型的保险系统中，在某一假设下，你或许会发现有些子系统并非必要。这个假设就是，假想我们拥有一台虚拟超级计算机，它没有网络带宽的限值，它能够使保险组织中的每个人无延迟地访问而且无需任何费用。不幸的是，这样的系统不存在。现在，我们将搜寻能够处理分布式应用，比如保险销售系统和表格系统的模式。

模式：保险销售系统

示例

你有一个灵活的后勤办公系统，你想要一个膝上前端办公系统使得你的销售代表能够帮助你更好地销售通过产品服务器定义的产品。

问题

如何在销售代表的膝上电脑上构造销售系统的结构？

动机

除了上述一般动机，你还要注意下面一些特殊动机：

系统发布：理想的保险系统应该具有一个分布式虚拟框架，这个框架提供给销售员一个图形用户界面，销售员只要轻轻按动手指就可以获取公司必要的信息。这个系统不能太贵，因为在内存存取，文件和数据库 I/O，网络 I/O 上我们有不同的性能。毕竟没有免费的无限带宽。

低投资对比个人市场表现：今天你也许能够以较低的价格买到销售标准金融产品的系统。但是这并非你所要。你需要的是个人市场的表现而不是标准产品，你需要的是更快得到它们。

上面提到的一般动机详细讨论了这一点。

灵活性：缩短产品创新周期需要一个非常灵活性的系统。

重用和单一源：为了保证费用限值，避免多源带来的冗余问题，你需要一个单一的代码库，并在前端办公系统和后勤办公系统做到最大程度的重用。

创新方面：说“我也能够”，你无法吸引你的客户。你需要一个地方来在你的应用中插入革新的部件。

方案

首先，把所有需要销售的产品和提供服务的东西都放在销售员的膝上电脑上。准备得越多越好。然后把这些功能分散到任何公司通用的外壳，一个合作伙伴子系统，一个销售记录，一个联系子系统，以及把投保转换为政策参数的核心销售部件。喜欢的话可以增加一个多媒体销售部件。把系统脱机连接到中央处理系统单元以便处理投保和交换客户数据。

结构

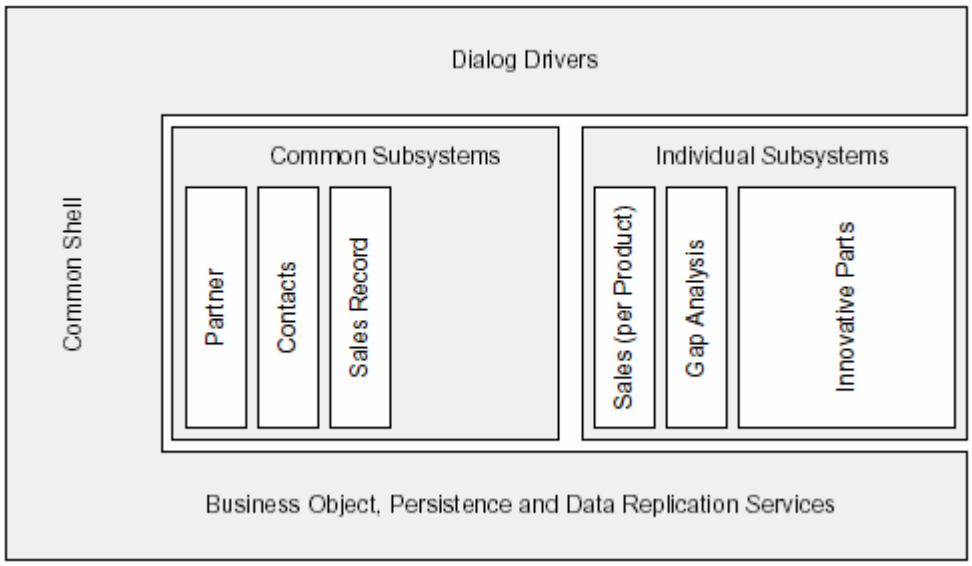


图 10 一个保险销售系统应用的结构

系统中各个组件具有如下责任：

通用 Shell（外壳）提供基于 PC 的面向对象方案，如 MVC 框架、保持框架所需要的所有基础部件。还提供你数据复制，通用应用驱动。通用应用驱动让你方便地插入新的应用。

合作伙伴子系统：允许销售代表收集关于你的客户，客户的金融状况，客户的兴趣爱好信息。所有的措施是使销售平滑，能够从目前群体中有选择地行动。一个合作伙伴子系统一般比后勤办公系统为销售员提供多得多的

功能。

销售记录：允许你跟踪你的伙伴从你这里或者其它公司那里已经购买的产品。个性化程度取决于你想提供的分歧分析组件的质量。

联系子系统：这个组件用来跟踪你的约会，计划定制等等。

销售：提供对话框使得你能够依据一定格式同你的客户一起计算保险单。这些对话框是面向产品的，典型的是使用产品服务器。

分歧分析：一个典型的个性化部件。它能够收集客户的信息，并且把客户已有的产品同总需求作一个比较，最好列出客户可能从你这里购买的产品。

革新部件：任何能够帮助你销售产品部件的，比如多媒体产品信息，都将使你的分析受益。

示例

已经知道了使用外壳的实现方法，现在你正联系一个卖主，启动一个项目，那么你可以把你产品和创新的想法放入外壳。

结论

跨越网络的性能：离线系统优点是在性能上，缺点是无法存取企业所有的数据。你需要销售你的产品而非想像在 POS 上的最大功能。

功能：模式描述了你今天在 POS 看到的系统，而没有描述你将来会看到的系统——我们将看到更多的功能移到 POS。这是一个实在的模式而不只是对未来的展望。

低投资对比个人市场表现：这种模式考虑到在相对低的投资下争取好的个人市场表现（产品和革新部件）。这个模式也为作为插件的革新预留了空间。

灵活性：在销售每个产品部件中，产品服务器提供了简短的产品革新周期。

重用和单一源：可以通过一系列方法达到重用：首先当你不打算重写一个单独的伙伴或者联系应用系统时，可以使用这种模式。其次，可以使用产品服务器。另外一个提高重用的好主意是在部件层次和商务对象层次使用组件的概念。

相关模式

一个销售系统经常使用产品服务器运行时组件，以及表系统和规则系统。

已知应用

有好几个定制产品的示例支持这种模式。例如 NSE 开发的 FINAS 系统(www.nse.de)，CAF 开发的 SILVA 系统(www.caf.de)。一些个性化产品如 EA-Generali 的 KUBIS 或者 Interunfall 的 AdiPlus，都是相似的。

模式：表系统

示例

保险系统，尤其是那些处理产品的保险系统，需要许多表格。这些表格用来提供关键字的合法集合，提供诸如从区域到相关利率映射实现自动保险策略。

问题

你如何提供灵活而且满足需要性能的系统，使得能够顺利存取那些大多数是只读的数据，能够把数据组织在表格中，提供给领域专家随时更新。

动机

性能对比冗余的功能：假如你要实现另一个比关系数据库性能更好，只是功能少一些的数据库系统，你就在制造系统的冗余。相反，提供一个本地只读数据库，它有非常好的存取速度，考虑到后期维护代价，这样的组件重复是完全值得的。

数据分发的投资和代价：在一个中央数据库中，你完全没有数据重复问题。而如果你有另外一个使用文件的数据库，系统在线时使用主存，离线时使用本地存储，这样就存在数据重复的问题。因此你需要在数据分发方便性和数据存取效率上权衡。

测试和新产品的分发：定义产品就象编写代码一样。在一个保险应用中，表中数据是定义的数据，因此必须像对待代码一样对待它：它们需要版本化，需要测试和发布过程代码。如果你想在一个中央数据库系统提供这种功能，无论如何你都得创造些额外的过程。

历史数据需要：保险系统非常依赖于历史数据。由于存在内部审核过程，你需要能够再现任何历史状态。更糟的是一旦政策参数发生变化，你必须能够在任何时间再现任何与保险合同相关的现在的和历史的数据。这就是所谓的二维历史。实际上没有哪个数据库能够天生就支持这种功能。

方案

使用一个表格系统，表格是这个系统的唯一数据提取层。这些表格驻留在运行时的主存中。

结构

对于客户，一个表格系统提供了一系列表格来存储中间抽取数据。

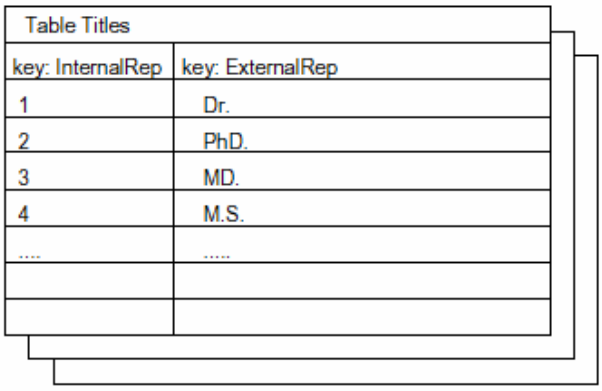
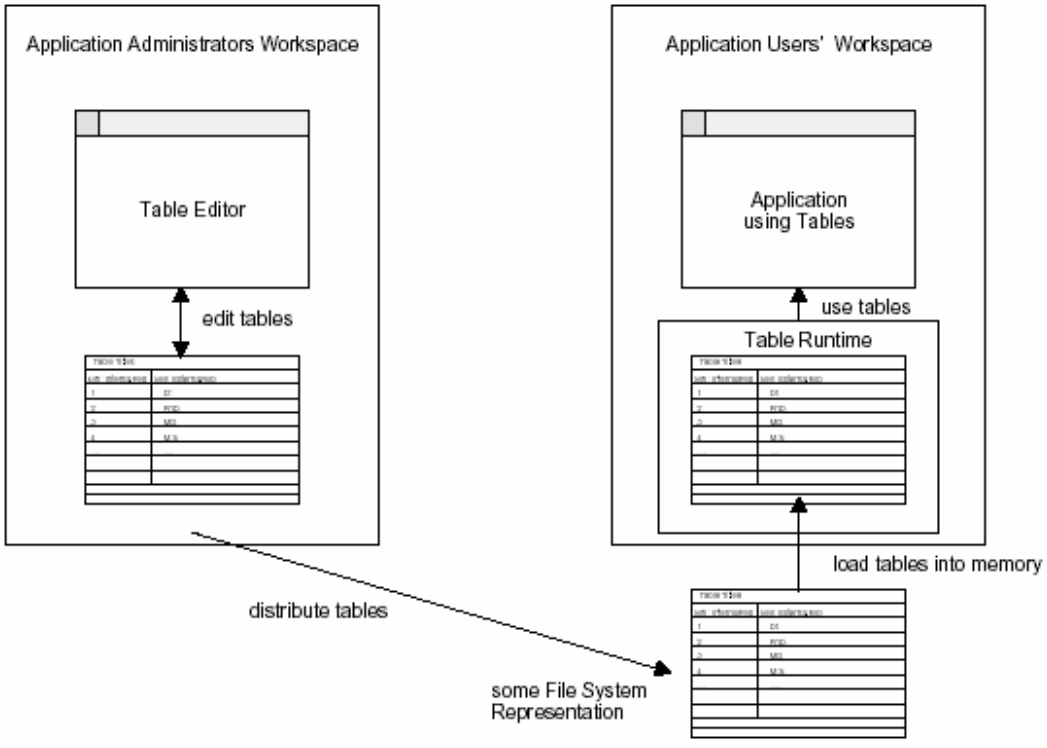


图 11 保险应用中的包含了头衔合法集的表

对于一个表格系统，你需要一些基本部件。



- 一个表格编辑器，用于管理员输入合法数据。
- 需要分发到所有客户的表格。它们通常保存在文本文件或者其它文件系统格式中。
- 当客户系统启动时，需要把所有的表格装载到内存。如果客户系统没有足够的内存转载所有的表格，它可以使用缓存机制和交换机制只读取所需的表格。

结论

性能：表格系统的访问速度远远优于数据库系统（前者是纳秒级，后者是微妙级）

冗余和代价：在你的主存安装一个经过修正的数据库，不得不开发为数据分发，测试，编辑开发程序。一个表格系统不便宜。

变种

市场上可以找到表格系统，这种表格系统要么能够处理历史数据要么不能处理。

相关模式

有关历史数据的模式参看 Fowler 的分析模式集合：历史映射模式和二维历史模式。要想真正实现一个系统，目前给出的细节还不够。我们自己的 Phoenix 框架包含了一个实现，但是一个设计并非一个模式，而且我们的确知道仅有的另一个实现，但非常困难。

已知应用

Table/23 是欧洲保险市场主机和客户系统使用广泛的产品。VP/MS 包含了一个表格系统，这个系统作为产品服务器的一个组件（但是没有包含历史数据）。两个产品都使用在多个保险系统中。VAA 包含了一个自己的表格系统规范，该规范使用在保险商务中。

经常使用的模式和策略

下面列举了本文中经常使用的模式

活动对象模型

无需编程你就可以采用活动对象模型。与此相似的概念是元系统或者反射

业务过程重组

你会经常把实现一个新的保险系统和业务过程重组努力结合起来。有关 BPR 的模式表述参看 Bee97

组合

组合模式是建模产品树时要使用的模式。

解释器

为了实现一个规则系统，你需要创建自己的解释语言，这个时候最常用的就是解释器模式

反射

反射模式以模式的形式描述了元系统。要实现产品的柔性，你有必要使用活动对象模型。

Type Object

Type Object 模式可以被用来连接政策参数和产品，方法是通过建造产品政策参数实例。产品树的递归结构多数情况是复合得到的。

虚拟机

如果你建立一个产品服务器，你或许需要解释你的产品定义。这时你就可以用一个虚拟机从运行产品（产品实例是政策）来去耦产品定义（编程）

整体部分

组合模式是整体部分模式的一个变体。因此，从这个意义来说，无论何时使用组合模式，就是在使用整体部分模式。

感谢

我要感谢我的 PLoP 导师 Richard Helm，感谢他给了我许多有用的提示和建议。

参考文献

[Aho+86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers: Principles, Techniques, and Tools, Addison-Wesley 1986.

[And+97] David M. Anderson, B. Joseph Pine: Agile Product Development for Mass Customization: How to Develop and Deliver Products for Mass Customization, Niche Markets, JIT, Build-to-Order, and Flexible Manufacturing; McGraw-Hill, 1997.

[Ald+98] Robert Aldrup, Jorg Baumann, Christian Weitzel: Die Phoenix Facharchitektur, agens & EA Generali AG, 1995-1998. http://www.agens.com/ - Look for Phoenix.

[Bee97] Michael A. Beedle: cOOherentBPR- A pattern language to build agile organizations, Proceedings PloP 97, http://st-www.cs.uiuc.edu/users/hanmer/PLoP-97/Workshops.html

[Bus+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Pattern Oriented Software Architecture - A System of Patterns, Wiley 1996.

[CAF97] CAF GmbH: VP/MS, Versicherungsprodukt-Modellierungssystem, CAF GmbH, Gilching 1996, 1997, 1998. http://www.caf.de/.

[Fow97] Martin Fowler: Analysis Patterns; Addison Wesley Longman, 1997.

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Elements of Reusable Object-oriented Software, Addison-Wesley 1995.

[Jac+96] Eydun Eli Jacobsen, Palle Nowak: A Pattern Language for Building Virtual Machines, Procedding EuroPLoP 1996; http://www.cs.wustl.edu/~schmidt/europlop-96/ww1-papers.html

[Joh98a] Ralph Johnson: ActiveObjectModel; Wiki Wiki Web; http://c2.com/wiki?ActiveObjectModel

[Joh98b] Ralph Johnson: Personal Communication, 1998

[Joh+98a] Ralph Johnson, Jeff Oakes: The User-Defined Product Framework; Work in progress, available via the author johnson@cs.uiuc.edu.

[Joh+98b] Ralph Johnson, Bobby Woolf: Type Object, in Robert Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998.

[Kel97] Wolfgang Keller: Mapping Objects to Tables: A Pattern Language, in Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany, Siemens Technical Report 120/SW1/FB 1997.

[Pho98] Jorg Kroger, Wolfgang Keller: Phoenix Business Model Framework, User's Guide, Internal Technical Document, EA-Generali AG 1998.

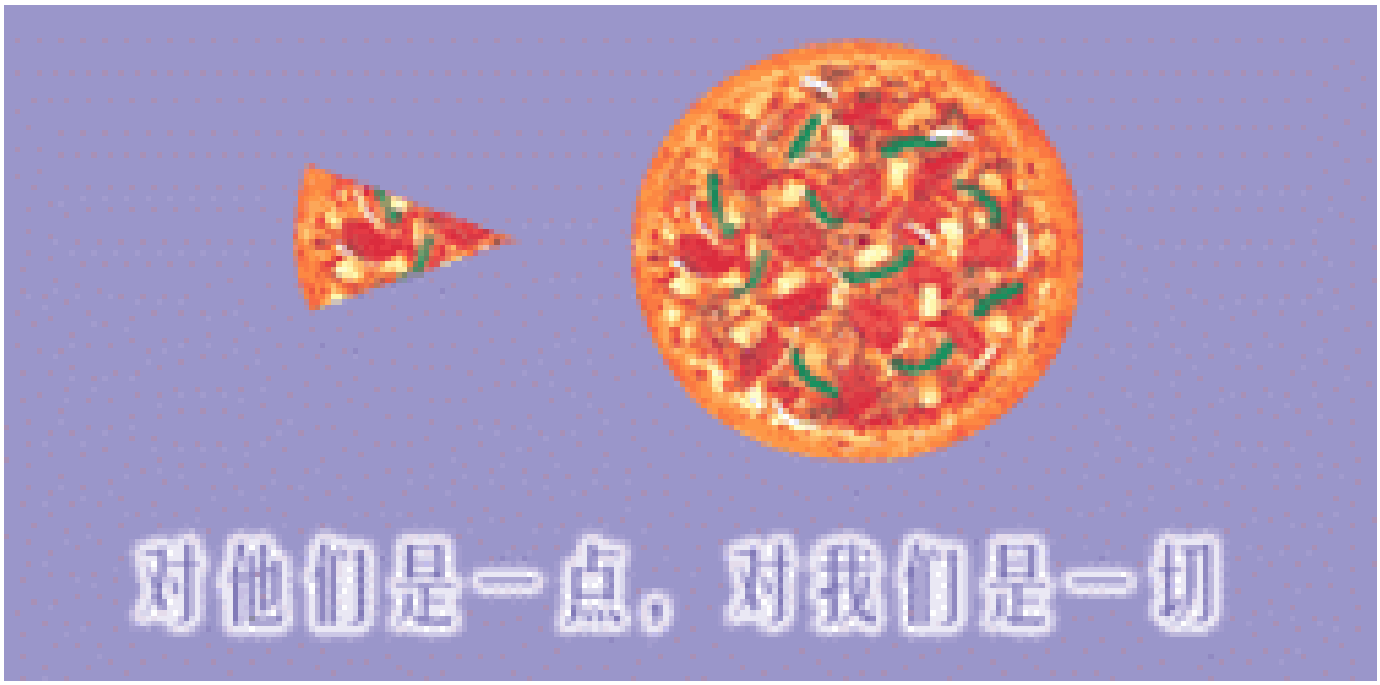
[Por85] Michael E. Porter: Competitive Advantage; The Free Press 1985.

[Ren+97] Klaus Renzel, Wolfgang Keller: Three Layer Architecture in Manfred Broy, Ernst Denert, Klaus Renzel, Monika Schmidt (Eds.) Software Architectures and Design Patterns in Business Applications, Technical Report TUM-I9746, Technische Universitat Munchen, 1997.

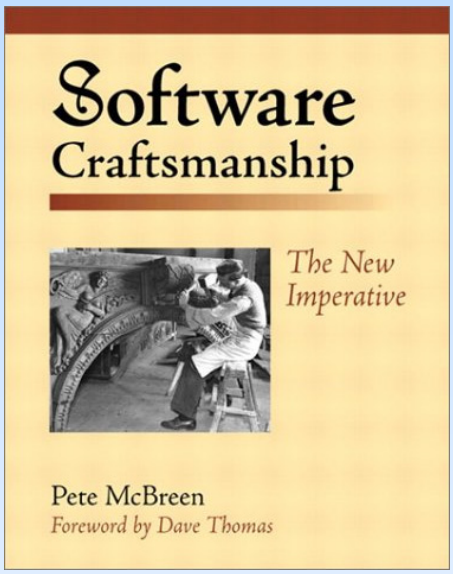
[Sch+96] Paul Schousleben, Ruth Leuzinger: Innovative Gestaltung von Versicherungsprodukten.; Flexible Industriekonzepte in der Assekuranz, Gabler, 1996

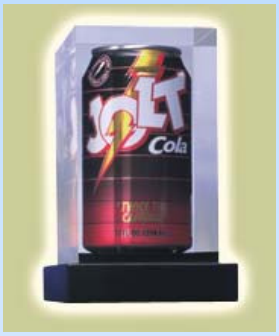
[VAA95] GDV: VAA - Die Versicherungs-Anwendungs-Architektur, 1. Auflage, GDV, Bonn 1995.

[VAA97] GDV: VAA - Die Versicherungs-Anwendungs-Architektur, 2., uberarbeitete Auflage, GDV, Bonn 1997.



《软件工艺》





2002 Jolt Awards 获奖书籍

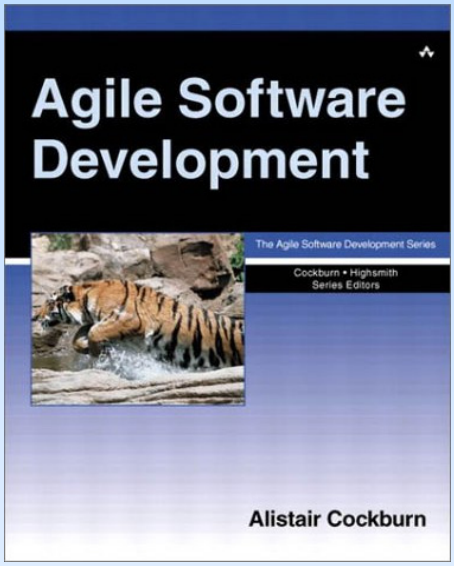
Pete McBreen

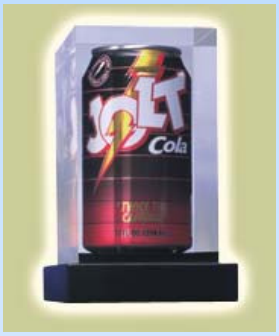
翻译：UMLChina 翻译组 Jill

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。

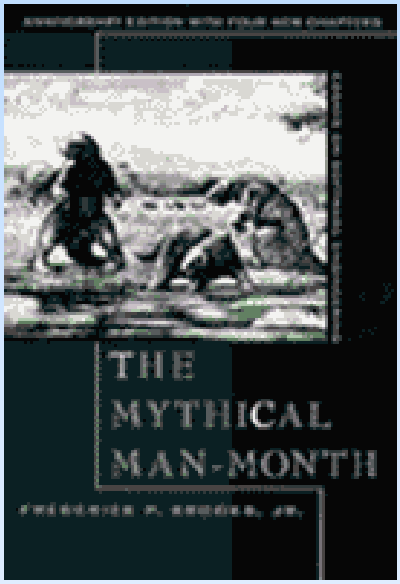
“是红酒。”我坚持道，并把标签指给她看。

“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

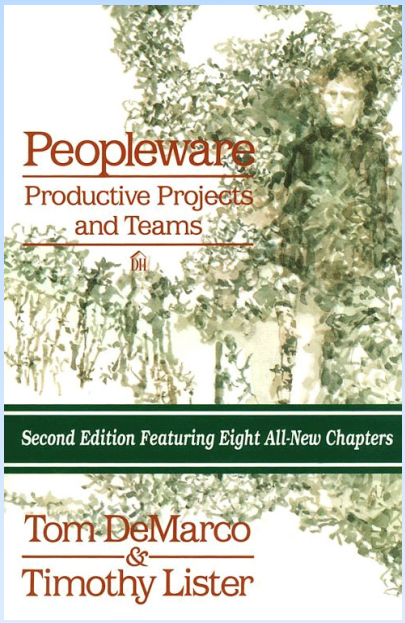
翻译：UMLChina 翻译组乐林峰

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

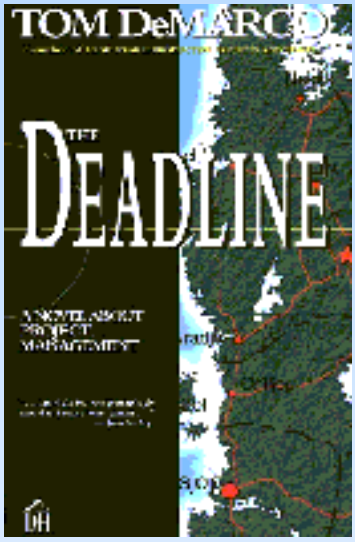
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

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！

XP 的价值和局限

ITS 高级会员 张恂 著

查看评论

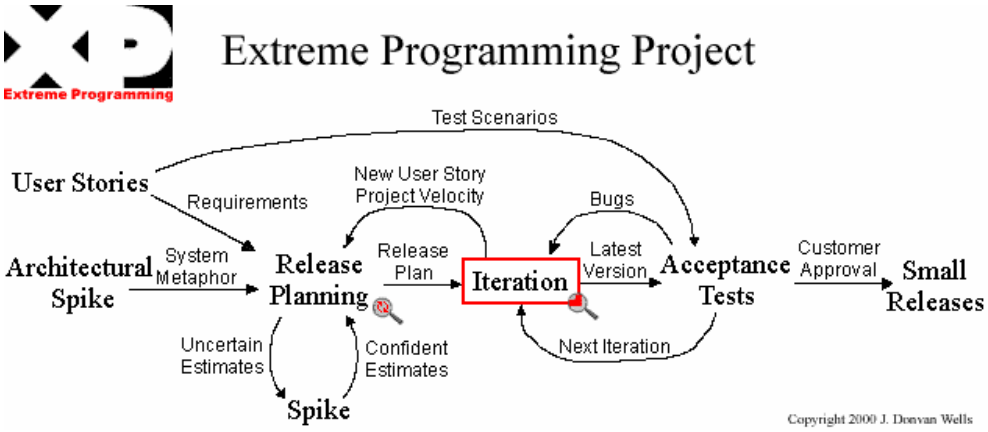
国际著名杂志 IEEE《Software》2001 年最后一期对极限编程（eXtreme Programming，XP）做了深入报道，一共刊出 4 篇文章，分别是：《从 CMM 角度看极限编程》（Mark Paulk，SEI）[PALK01]、《在一家过程密集型的公司中实施极限编程》（James Grenning，Object Mentor）、《恢复、补救与极限编程》（Peter Schuh, ThoughtWorks）以及《在维护环境下运用极限编程》（Charles Poole and Jan Willem Huisman，Iona Technologies）。作为 CMM 标准制定的核心人物来评价 XP，Paulk 的文章确实是一份非常难得的权威性报告，而后三篇文章则从应用的角度介绍了在不同环境下实施 XP 的成功经验。最后，Glass 先生在“Loyal Opposition”评论栏目中还对 XP 发表了一分为二的看法[GLAS01]。

我读了这些精彩的文章，结合自己平日的学习和体会，感受颇多，现整理如下。

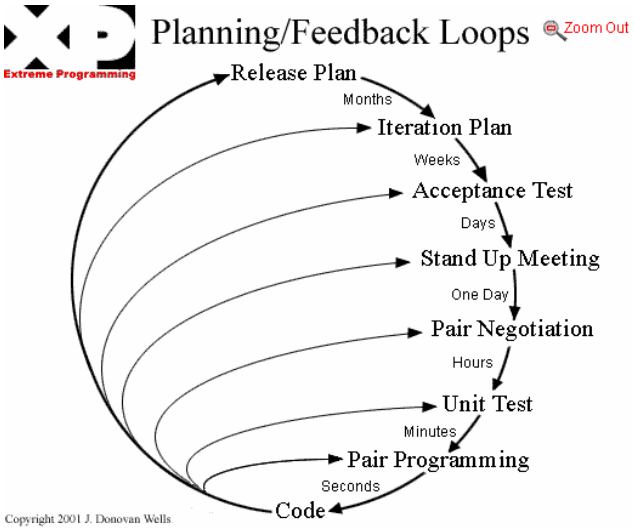
什么是 XP？

首先回顾一下 XP 的基本内容和特点。Kent Beck 在他的开篇之作《Extreme Programmin Explained – Embrace Change》中提出了“极限编程（XP）”这一创新的过程方法论[BECK99]。XP 是一种高度动态的过程，它通过非常短的迭代周期来应对需求的变化，所以 Beck 把这本书的副标题定为“拥抱变化”。XP 一般适用于需求不确定、变化快，项目历时不超过半年，而人数不超过 10 个、在同一地点工作的中小型团队。

XP 提供了一个全局的、价值驱动的开发过程视图，体现了 4 个价值目标：沟通（communication）、简化（simplicity）、反馈（feedback）和勇气（courage）。XP 的生命周期包括 4 个基本活动：编码（coding）、测试（testing）、聆听（listening）和设计（designing）。一个 XP 项目的状态变迁如下图所示[XPORG]：



XP 的特点是，要求首先开发出最重要的特性，迅速向客户提供所需的功能，随着代码的演进通过重构来满足新的需求，从而使整个项目失败的风险减到最小。XP 的计划/反馈循环如下图所示[XPORG]。可以看出，从需求定义开始，XP 省略了常规的系统和架构（architecture）的设计步骤，从简短的计划直接进入编码的迭代循环。在 XP 中，编码和设计是同时进行的，而且特别强调测试的重要性。



在一个充满技术、业务和人员风险的环境中，如果对未来投入过多，例如试图通过详细的架构设计、精确的代码结构以及一致完备的文档来满足将来所有可能的情况，结果很可能造成一种精力和时间上的浪费，变化的需求和技术等许多不确定因素会使过去大量的努力付之东流。因此，XP 弱化针对未来需求的设计，非常注重当前的简化。Beck 还建议推迟重要的设计决定，除非某个特性有这样的需要[BECK99]。

既然未来是未知的，需求和环境都在不停地变化，那么只专注于当前已经明确的、优先级最高、最有价值的部分而不必为现在尚且无关甚至可能永远无关的问题进行设计似乎是合理的。然而我认为，XP 有一个非常关键的基础假定，即开发人员只注重眼前需求而依赖将来不停重构所带来的开销和风险要远小于需求变化使事先充分设

计失效的代价；如果反复重构的风险要远大于事先进行充分设计的开销，那么实施 XP 可能就是不明智的。这一前提条件决定了 XP 适用的范围和场合。

XP 方法评述

以下结合国内软件开发的实际情况对 XP 的核心内容——12 个实践方法（practices）进行简短地评述。

1）计划博弈（planning game）

XP 要求结合业务和技术情况，快速确定下一次发布的范围。在项目计划的 4 要素（费用、时间、质量和范围）中，由客户选择 3 个，而程序员可以选择剩下的 1 个。通常客户从业务角度确定项目范围、需求优先级和开发进度，开发人员则做出具体的成本和技术估计。XP 强调简短和突发性的计划，有时只用几个小时甚至几分钟就能完成，而且可以随时按需进行多次计划。

我们永远不可能做出绝对正确和完整的计划，因为未来是变化的。最好的解决办法是预见变化、控制风险。如果一次计划做得不够好，那就多做几次。在项目规模小、复杂程度低而不确定因素又多的情况下，XP 的计划博弈确实能够既提高效率又减少风险，但是这不等于不做计划，也不是在做游戏，相反需要良好的技巧。

2）小型发布（small releases）

XP 可以迅速让一个简化的系统投入使用，在非常短的周期（比如 2 个星期）内以递增的方式发布新版本，从而很容易估计每次迭代（iteration）的进度，便于控制工作量和风险，客户的需求和反馈也能够得到及时处理，体现了敏捷的优点。

3）系统隐喻（system metaphor）

XP 通过一个简单的关于整个系统如何运作的隐喻性描述（story）来指导全部开发。隐喻可以看作是一种高层次的系统构想，通常包含了一些可以参照和比较的类和模式，它还给出了后续开发所使用的命名规则。XP 不需要事先进行详细的架构设计。

隐喻在某些情况下确实可以代替正规的架构设计，但它通常只适用于小系统[POL201]。另外，Beck 还提出“不断地细化架构”，这是对的。但是问题出在“不断（ongoing）”上。如果把它理解成鼓励随着项目的发展不断改进架构，是不错；但是，如果像一些人误解的那样，可以推迟架构的分析乃至推迟整个项目的计划和系统设计，那就有问题了。

4）简化设计（simple design）

在任何时候都要求代码的设计尽可能简单，恰好满足当前功能的需要，不多也不少。

5）测试驱动（test-driven）

XP 要求“先写测试，后编码”。失败的测试用例（test case）驱动编码和设计，可以减少不必要的开发量。开发人员必须保证单元测试和集成测试始终运行无误，甚至现场的客户代表也要能够编写功能测试程序。

这是所有软件过程方法都一致推荐的做法。无论怎么强调测试的重要性都不为过。

6）重构（refactoring）

重构是指在不改变系统行为的前提下，重新调整、优化系统的内部结构以减少复杂性、消除冗余、增加灵活性和提高性能。

重构不是 XP 所特有的行为，在任何开发过程中都可能发生。通过重构改进已有代码的设计是对的，但重构很容易被误用。如果像有些人理解的那样，重构意味着在开发的时候进行持续不断地设计（ongoing design），那就有问题了。首先，一个程序员重构完的代码对其他人而言可能并不简单；而陷落于老是重构的陷阱之中会使团队停滞不前，依赖重构甚至会成为轻视设计、把设计推迟到编码阶段乃至发布的最后一刻的借口，这对于大中型项目很可能是场灾难[GLAS01]。

7）结对编程（pair programming）

由两名程序员在同一台电脑上结成对子共同编写解决同一问题的代码。通常一个人写代码，而另一个人负责同时保证代码的正确性和可读性。这可以看作是一种非正式的持续同级评审（peer review）。

两个脑袋是否一定胜过一个脑袋？该做法目前在国内外的争议比较多，它究竟能带来多大好处还有待大量实验的验证。问题的一方面是难以找到合适的人选来实施，很多程序员在工作的时候不希望被打搅，喜欢独立思考[GLAS01]，所以结对编程的两个人的性格和编程技能应该匹配；另一方面，管理层可能不太愿意让两个人来同时完成一件工作，认为这是资源浪费。而国外有些研究则表明，结对编程可以有效地加强协作、促进问题解决、减少软件缺陷、缩短开发周期，在一些项目中获得的收益要大于增加的资源开销[PALK01]。

8）代码全体拥有（collectively ownership）

这意味着任何人可以在任何时候改进系统任何部分的代码。这提高了代码的透明度，增进了团队的合作精神。

虽然此举效率很高，但对于大中型系统，采用该方法若管理不善很可能引起混乱[POL202]。

9）持续集成（continuous integration）

XP 提倡一天之中集成、建立（build）成品系统很多次（每当完成一个任务），而且随着需求改变，要进行不断地回归测试。这不禁令人联想起微软非常成功的每日建立（daily build）和烟雾测试方法。

值得注意的是，XP 的小型发布、持续集成和代码全体拥有都需要良好的软件配置变更管理系统和运作流程来支持。

10）每周 40 小时工作制（40-hour week）

XP 要求尽可能安排程序员每周工作 40 个小时，加班不得连续超过两周，否则反而会影响生产率。疲倦的程序员不可能始终保持峰值效率。

这点体现了 XP 的“以人文本”思想，然而实施起来好像有些困难，过于理想化了，但是如何合理地安排工作量和进度的确值得国内软件企业和用户引起重视。

11）现场客户（on-site customer）

XP 要求至少有一位实际用户的代表全天候在现场负责确定需求、回答开发团队的问题和编写功能验收测试。

这确实是解决与客户沟通不畅问题的好办法，但是国内许多缺乏技术实力的客户还满足不了这种要求，而且客户往往认为在给出了含糊不清的需求之后就可以撒手不管了，出了问题则要开发者全权负责。可能关键不在于客户是不是一定要到现场。

12）代码规范（coding standards）

XP 强调通过制定严格的代码规范来进行沟通，尽可能减少除代码之外的不必要文档。

XP 对架构描述和文档的弱化对于很多国内的项目可能是个缺点。毕竟代码的可读性不能与图形化的架构模型和规范的文档相比，代码并不是一种有效的适合所有受益人（stakeholders）的沟通媒介[POL101]。

归纳一下，XP 通过以上做法实现了 4 个价值目标：

- 沟通

让开发人员集体负责所有代码并结对工作，鼓励与客户以及团队内部的不断沟通。

- 简化

鼓励只开发当前需要的功能，摒弃了过多的文档，坚定地专注于最小化解决方案，做好为新特性改变设计，在系统隐喻和公共代码规范的指导下不断重构的准备。

- 反馈

通过单元测试和功能测试获得快速反馈。在编码之前先写测试用例，并在设计改变或集成之后重新测试，客户现场代表也能编写功能测试。

- 勇气

提倡积极面对现实和处理问题的勇气，比如放弃已有代码、改进系统设计。

XP 与 CMM 、RUP 的比较

CMM 告诉组织为了系统化地建立、实施和改进软件开发过程应该做些什么，但没有说明如何去做以及采用哪些具体的技术、策略和方法。CMM 是一套通用的过程实践标准，适用面很广。实施 CMM 要求组织在过程的制度化建设上付出大量努力，通常被认为是重载（heavy-weight）的模型。

XP 是一个针对某种特定环境（需求变化快的小型团队）的具体过程实施模型和方法论。它其实是一种演进式的原型化方法（evolutionary prototyping）[MCNL96]，具有沟通高效、设计简单、反馈迅速等特点，因而是一种轻载（light-weight）、敏捷（agile）的过程方法。

Mark Paulk 在他的文章中把 XP 的实践方法与 CMM 的 KPA（关键过程域）进行了对照。得出的结论是，XP 部分满足或大部分满足了 CMM 2-3 级 KPA 的要求，而基本上没有涉及 CMM 4-5 级的 KPA。这说明 XP 的做法基本符合了 CMM 的目标和 KPA，但还不完备。总体上看，XP 侧重于具体的过程和开发技术，而 CMM 更关注组织和管理上的问题。XP 缺少的一个重要内容是“制度化（institutionalization）”，它不含有被 CMM 认为是使良好的工程和管理实践制度化的关键基础设施和管理要件。[PALK01]

RUP（Rational Unified Process）是一个风险驱动的基于 UML 和构件式架构的迭代递增型开发过程（框架）。RUP 定义了 4 个阶段（起始、细化、构造、移交）和 9 个科目（业务建模、需求、分析和设计、实现、测试、部署、配置和变更管理、项目管理、环境）。这些阶段对应着关键里程碑的划分，而不同科目的工作流和活动在生命周期的迭代中可以并发进行，具体执行的程度则可以调节。RUP 对于角色、流程、工件和活动的要求是灵活的、

可配置的，所以它广泛地适用于各种类型和规模的项目。RUP 集中体现了 6 个软件开发的最佳实践方法：迭代式开发、需求管理、构件式架构、基于 UML 的可视化建模、持续校验质量、变更管理。RUP 是一种以架构为中心的开发过程，而这正是大、中型项目成功的关键。

XP 的编码和设计活动融为一体，弱化了架构的概念，这是它与强调架构设计的 RUP 的最大不同。架构的内涵要远大于一些简单的隐喻，它要考虑结构、行为、环境、使用、功能、性能、可靠性、弹性、重用、可理解性、约束和权衡乃至美学等诸多方面的因素。设计架构的目的不是为了完美地表示系统的全部细节，而是为了消除最主要和最关键的架构风险。RUP 细化阶段的主要目的就是构造出一个可运行的架构原型，作为将来添加需求功能的稳固基础。另外，XP 没有包含业务建模、部署等概念，反映了它以编程为中心、节省一切的哲学。

当然两者也有不少共同点。例如，它们的基础都是面向对象方法（取代传统的结构化方法），都重视代码、文档的最小化和设计的简化，采用动态适应变化的演进式迭代周期（取代传统的瀑布型生命周期）、需求和测试驱动并鼓励用户积极参与等等。

由于 RUP 提供了非常丰富的内容，所以常常被误解为一个重载的过程。通过定制 RUP 这个通用的过程框架，去掉项目不必要的工件（artifacts）和中间环节，把 XP 的做法（比如短小的迭代周期、结对编程、测试优先的设计和重构）吸收进来，也可以实现 RUP 过程的敏捷和轻量化[SMTH01]。“Bob 大叔”（Robert Martin）甚至从 RUP 中裁剪出了一个酷似 XP 的最小化 RUP 过程——dx[MART01]。我设想，XP、RUP 乃至其他工程和管理方法可以统一起来使用，姑且成之为统一软件过程（Unified Software Process，USP）。例如，一个大项目团队在起始和细化阶段采用 RUP 方法完成需求分析和架构设计，在构造和移交阶段采用 XP 的做法来实现部分子系统或模块。因篇幅限制，此处不再展开讨论。

“轻载”、“敏捷”是美丽的词汇，无人会拒之于门外。我想 XP、RUP 的目标是一致的——提高团队效率、开发出高质量的软件，而区别就在于各自的侧重点不同，从而导致两者的适用情况和应用范围有差异。然而，它们是可以互补的，无论是以架构为中心，还是以代码为中心，灵活运用的关键就在于掌握其中的最佳实践方法，实施 RUP、XP 或者融合了两者的过程（比如 USP）都将有助于组织更好地实现 CMM 目标。

XP 在国内应用的问题

那么，国内的软件开发企业应该如何运用 XP，会遇到那些问题呢？

根据 XP 的特点，它尤其适合规模小、进度紧、需求变化大、质量要求严的项目。XP 的一切因变而设，出发点就是希望以最高的效率和质量来解决用户眼前的问题，以最大的灵活性和最小的代价来满足用户未来的需要。XP 在如何平衡短期利益和长期利益之间作出了巧妙的选择。

当然，和任何一种软件方法一样， XP 也不是万灵药。Beck 曾经建议在以下环境中不宜使用 XP[BECK99]：

- 不能接受 XP 文化的组织；
- 中大型（超过 10 个人）的项目和团队；
- 重构会导致大量开销的应用；
- 需要很长的编译或测试周期的系统；
- 不太容易测试的应用；
- 人员异地分布的物理环境。

我想 XP 在国内应用有几种不同程度的实施方式。第一种，全盘接受，严格遵守 XP 的定义执行，估计这种方式不太切合实际；第二种，整个开发过程采用 XP 的生命周期模型，实行 XP 大部分的实践方法，根据实际情况对其中个别做法进行调整或舍弃；第三种，在保持组织原有的开发过程和生命周期模型的情况下，借鉴、采取个别对项目有效的 XP 做法。为了不使管理和沟通效率下降，XP 一般适合于开发小项目的小团队。但根据前面的讨论，XP 其实也适用于开发大项目中的子系统或子模块的小组，这时可以根据该模块在整个架构中的依赖关系来确定该小组的用户和需求。具体在何种环境下采用什么方式来实施 XP 要根据应用类型、项目特点和组织文化而定。

在 XP 的 12 种实践方法中，测试驱动、持续集成、简化设计、代码规范、现场客户、每周 40 小时工作制、小型发布都是我们应该积极提倡和鼓励的，而代码全体拥有、结对编程、重构、隐喻以及计划博弈等做法实施起来要小心，它们并不是在任何情况下都适用的，容易被误用。

目前国内软件开发企业的过程管理水平普遍较低，存在大量轻视或忽视需求、架构、文档、测试以及开发文化和制度建设的现象，许多管理者和开发者不知道在提高开发效率和软件质量方面应该做些什么、如何去做。

举个例子，国内常见的“噩梦”项目一般是这样开始的：草草地做计划、分析用户需求；接着，为了赶工期、赶进度，过分自信地省略必要和清醒的架构分析与系统设计，直接铺开大范围的编码开发，而且基本上不做文档，只在代码中留下一些难懂的注释；然后，没经过充分、认真的测试就急急忙忙推出运行版本；结果，移交阶段软件缺陷层出不穷，麻烦接二连三，再加上业务、市场和技术的不断变化，客户又老是提出新的功能和修改要求，

开发人员只好不停地“重构”代码、到处打补丁（“code and fix”），客户却始终不满意；最终，进度一再延误，开发人员疲于奔命，根本没有工夫顾及产品质量，导致客户抱怨不断，项目似乎落入了黑洞……

对比 XP，我们发现，迫于成本和进度的压力，国内的实践其实已经很“轻量化”了——在计划、分析、设计、测试、文档上花的精力和时间很少，但又像实施了半拉子的 XP。在各种客观因素和借口的名义下，人们在促进沟通、简化和反馈方面做得很薄弱，勇气却很大，敢于冒由于不规范管理而导致项目失败的风险。指望通过重构来减少投入、改进软件，可是没想到一旦“重构”失败造成的影响和损失反而更大，而依赖于个人英雄和没有测试保证的项目往往是在碰运气。

此外，XP 是一个定义规范的过程方法论。它要求开发团队具备熟练的代码设计技能和严格的测试保障技术，在书中 Beck 甚至想当然地假定如今精妙的技术和程序员的技能已经能够满足实施持续设计变更的要求[GLAS01]；再者，XP 的成功实施离不开高度协同的开发文化和环境。那么，国内的软件开发人员是否都已经真正理解了面向对象和模式，掌握了重构和 OO 测试技术，习惯了“先写测试，后编码”，在技能和心理上做好了准备来获得 XP 的价值呢？

鉴于以上原因，尽管 XP 有敏捷、高效等种种好处，但是现阶段国内的组织误解、误用 XP 的可能性还是很大的。所以，一方面强调提高过程制度化和架构设计的水平，另一方面重视掌握运用敏捷方法的技术和技能，是国内软件开发企业应该面对的双重课题。

结语

在软件学科和许多其它领域中，针对同一个问题，通常至少有两种以上的解决方法。相对于传统的以架构设计为中心的自顶向下过程，XP 方法论的出现则证明了以代码设计为中心的自底向上过程的合理性和有效性，这正是 Kent Beck、Ward Cunningham、Ron Jeffries 等 XP 倡导者在多年理论探索和成功实践的基础上为我们做出的杰出贡献。

成功实施 XP 的关键在于积极有效的管理，对测试驱动编程方式的工具支持，对结对编程的资源保证，用户的积极参与以及对开发人员在项目计划、测试用例编写、重构和结对编程等方面的充分培训。

不过，我对 XP 的名称（“极限编程”）有些不以为然。在软件工程的管理和实践中巧妙地掌握各种平衡往往是成功之道，除了帮助吸引更多的眼球外，“极限”（极端）做法究竟有多少可行性？用“XX 编程”命名一个过程方法也真实地反映了它的局限性。

XP 显然值得我们学习、研究和借鉴。不管当代软件项目符合 XP 实施条件的程度如何，我们都可以吸收其经验、汲取其价值，从中获益。实际上，CMM、RUP、XP 都不是全新的发明，它们是对几十年来国外软件开发实践成功经验的总结、继承和发展。但是，如果我们不懂得结合实际、灵活运用反而囫囵吞枣、盲目照搬，就会冒很大的风险。国内软件开发企业尤其应当注重理解、消化这些先进思想方法的外延和内涵，从中学到最佳实践方法，取长补短、持续改进，逐步建立起一套适合自我、行之有效的过程体系。

参考资源

[GLAS01] Robert L. Glass, Extreme Programming: The Good, the Bad, and the Bottom Line, IEEE Software, Vol.18 No.6, Nov/Dec 2001.

[BECK99] Kent Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, Reading, Mass., 1999.

[PALK01] Mark C. Paulk, Extreme Programming from a CMM Perspective, IEEE Software, Vol.18 No.6, Nov/Dec 2001.

[POL101] Gary Pollice, RUP and XP, Part I: Finding Common Ground, the Rational Edge, 2001. (http://www.therationaledge.com/content/mar_01/f_xp_gp.html)

[POL201] Gary Pollice, RUP and XP, Part II: Valuing Differences, the Rational Edge, 2001. (http://www.therationaledge.com/content/apr_01/f_xp2_gp.html)

[MART01] Robert Martin, http://www.objectmentor.com/publications/RUPvsXP.pdf, a chapter from Object Oriented Analysis and Design with Applications, Third Edition, Addison Wesley, 2001.

[MCNL96] Steve McConnell, Rapid Development, Microsoft Press, 1996 (Chapter 21).

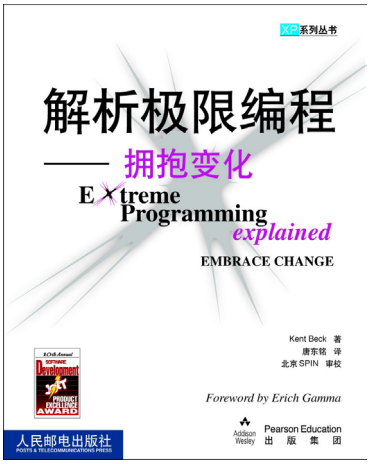
[SMTH01] John Smith, A Comparison of RUP and XP, Rational Software White Paper, 2001 (http://www.rational.com/products/whitepapers/423.jsp)

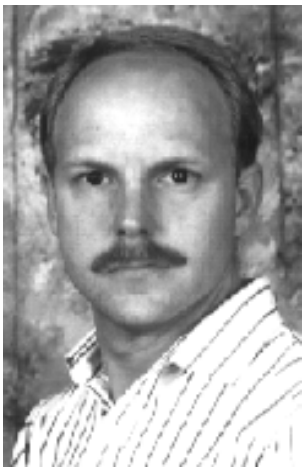
[XPORG] Don Wells, http://www.extremeprogramming.org/

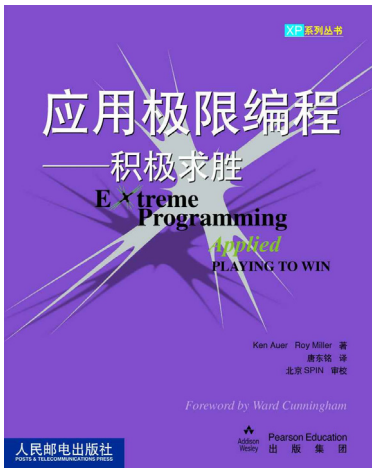
*感谢您阅读此文，如果您对本文有任何意见或建议，请发信到 zhangxun2001@hotmail.com。

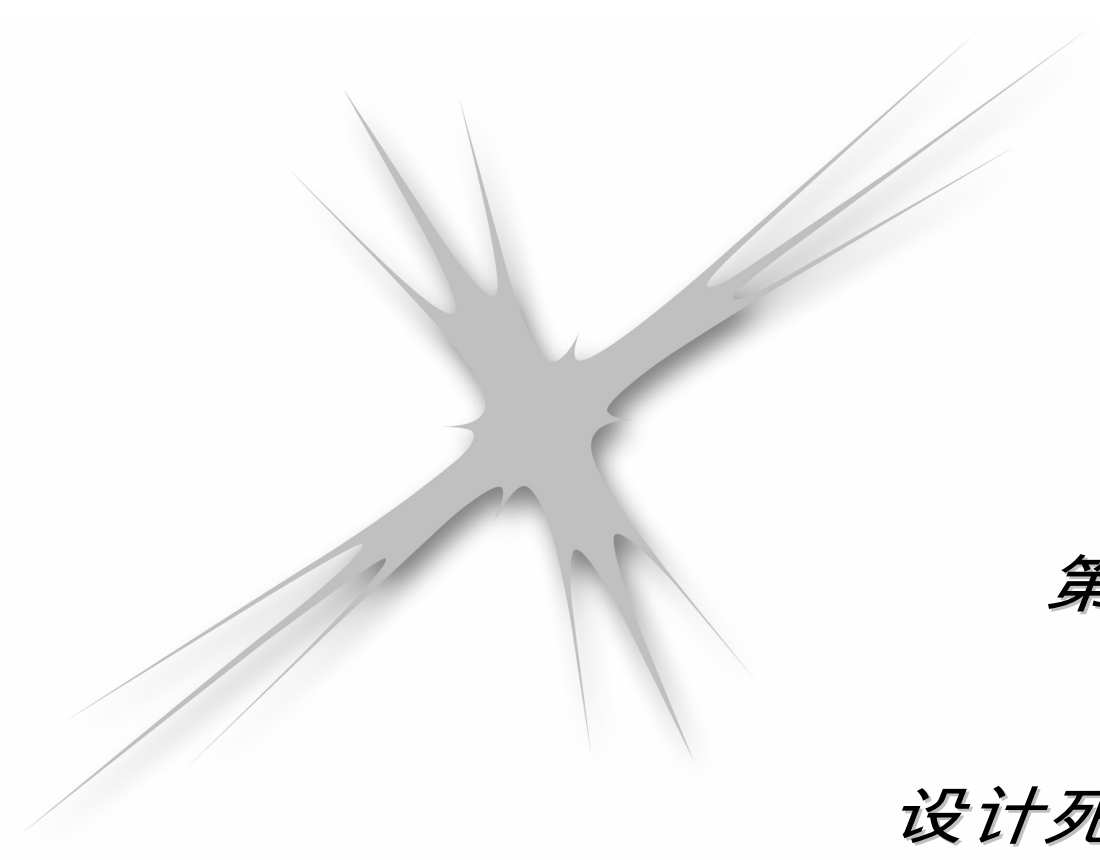
**您在“IT 之源”（http://www.iturls.com/）和 UMLChina（http://www.umlchina.com/）网站上可以得到本文的最新版本和其他相关资料。

人民邮电出版社 XP 系列









第 1 章

设计死亡了吗

—— Martin Fowler[1]

对于许多开始粗略接触极限编程（Extreme Programming，XP）的人来说，似乎 XP 宣告了软件设计的死亡。不但许多设计工作被奚落为“Big Up-Front Design（巨大的前置设计）”，而且诸如 UML、富有柔性的框架，甚至模式这样的设计技术都不被重视或者近乎于忽略了。实际上，XP 包含有许多设计思想，只是采用了一种与既定的软件过程不同的方式来进行设计。XP 借助的多种实践让演进成为一种可行的设计策略，以此使演进设计的概念焕发青春。设计者需要学会如何进行简单设计、如何使用重构技术保持设计清晰，以及如何以演进的方式使用模式，因此 XP 也带来了新的挑战和技巧。

极限编程（Extreme Programming，XP）向软件开发方面的许多共识发出了挑战。其中最富有争议的一项就是它赞成采取一种更为循序渐进的方法，抵制在前置设计（up-front design）上花费巨大精力。对于它的批评者来说，这是走“编码加修正（code and fix）”的开发方式——这往往被嘲笑为“hacking”——的回头路。就连它的倡导者也常常将它看作是对设计技术（比如 UML）、原理和模式的

1．版权所有©2000 Martin Fowler。保留所有权利。本章的在线版本可以在 http://www.martinfowler.com/articles/designDead.html 找到，在线版本偶尔会得到更新。

一种抵制。不必担心设计：如果你聆听代码，良好的设计就会出现。

我发现自己处在这一辩论的中心。我的工作经历太多地牵扯到了图形化的设计语言——统一建模语言（Unified Modeling Language，UML）及其前身——和模式上面。实际上，我本人曾经撰写过 UML 和模式方面的书籍。我倡导 XP 就意味着我放弃了我在这些主题上面所写的一切东西，就意味着我抛弃了我对所有这些反演进（counter-revolutionary）概念的认识了吗？

好了，我不打算让自己让读者苦思冥想而无所适从。简短地回答是——不会。完整的回答就是本文余下的内容。

1.1 计划的设计和演进的设计

对于本章而言，我准备介绍在软件开发中进行设计的两种方式。最普通的方式或许就是演进设计（evolutionary design）。演进设计实质上意味着系统设计伴随着系统实现而成长。设计是编程过程的一部分，而且随着程序的演变，设计也在变化。

按照其平庸的用法，演进设计是一种彻底的失败。设计最终成了一大堆随机应变决策的杂凑，每种决策都使代码更加难以改变。你可能从许多方面都指责说这不是设计；的确，它往往导致糟糕的设计。正如 Kent 所言，设计的目的从长远来看是使你能轻松地不断改变软件。随着设计的恶化，你进行有效更改的能力也会下降。你就遇到了软件失序（entropy）的状态，随着时间的推移，设计变得越来越差。这不但使软件更加难以改变，而且让缺陷更容易滋生且更难以发现和安全地清除。这正是“编码加修正”方式的噩梦，此时随着项目的进展，修正缺陷的代价会呈指数增长。

计划设计（planned design）与之相反，并且蕴涵了一种产生于其他工程学科的概念。如果你想要搭一个狗窝，只要把一些木头拼在一起，有个粗略的形状即可。但是，如果你想要建一幢摩天大楼，就不能采用那样的方法——否则在你还未盖到一半之前大楼就会坍塌的。因此，你要从施工图入手，由一家工程设计公司完成它，就像我妻子在波士顿市区工作的那家公司一样。她在进行设计的同时就会考虑到所有的问题，部分是通过数学分析，但大部分是借助建筑规范。建筑规范是有关怎样根据成功经验（以及一些背后的数学）设计建筑结构的准则。一旦设计完成，她的工程公司接着就把设计交给另一家建造这幢大楼的公司。

软件开发上的计划设计应该也是如此。设计者要事先仔细考虑重大问题。他们不需要编写代码，因为他们并不是在构建软件，他们是在设计它。于是他们可以使用一种像 UML 那样的设计技术，它能抛开编程的细节而让设计者在一个更抽象的层次上工作。一旦设计完成，他们可以把它交给一个单

独的工作组（或者甚至是一家单独的公司）去进行构造。既然设计者考虑的范围更广，所以他们能避开引发软件失序的诸多随机应变的决策。程序员可以遵照设计的意图，而且如果他们遵从设计，那么就会构造出良好的系统。

如今的计划设计方法从 20 世纪 70 年代起就有了，而且许多人都用过它。它在许多方面要比“编码加修正”的演进设计更好。但它有一些缺点。第一个缺点是不可能把编程时需要处理的所有问题都统统考虑到。因此不可避免地会在编程时发现对设计存有质疑的地方。但是，如果设计者已完成工作，被调往另一个项目，怎么办？程序员围绕设计开始编码，这就引入了失序（entropy）。即使设计者还在，但是找出设计问题、改变设计图以及随后调整代码都要花时间。修正过快而且有时间压力是常有的事。因此（又）会引入失序。

此外，常常还有一个文化上的问题。设计者是由技能和经验所造就的，可是他们的设计工作是如此之忙以至于他们再也没有太多时间来编写代码了。但是，软件开发的工具和资料以飞快的速度变化着。当你不再编写代码时，你不仅错失了伴随这股技术潮流所出现的变化，你还会失去编码人员的尊重。

这种建造者和设计者之间的紧张关系同样也出现在建筑工程中，但是在软件工程上更为强烈。之所以强烈是因为有一个关键性的差异。在建筑工程中，设计人员和建造人员的技能之间有一条更为清晰的界线，但在软件工程中则很模糊。在高级设计环境中工作的程序员要非常娴熟才行。娴熟到足以能对设计者的设计产生质疑，特别是当设计者对开发平台的日常实情了解不足时更是如此。

如今这些问题可以得到解决。或许我们能解决人们的紧张关系。或许我们能让设计者娴熟到足以处理大多数问题并且有一个过程足够规范可以改变设计图。仍然还有另一个问题：变化的需求。变化的需求是我参加的软件项目中让人头痛的头号大问题。

要处理变化的需求，一种办法是在设计中引入柔性以便能随需求的改变而轻松地改变设计。但这需要能洞察到预期的变化是什么类型。一项设计可以有计划地处理不稳定的领域，但只对可预见的需求变化有帮助，对于没有预见的需求变化不会有帮助（并且会有害）。所以必须对需求理解得足够充分，把不稳定的领域区分开来，我认为这是非常困难的。

目前这些需求问题中有一些是因为对需求的理解不够清楚。于是许多人专注于需求处理过程上，以此获得更好的需求，希望这会防止需求在以后改变设计。但即便这种做法也可能不是最后解决问题的妙方。许多没有预见的需求是因为业务的变化而出现的。无论需求处理过程怎样认真，那些情况也是不可避免的。

因此，所有这些原因使计划设计显得不太可能。当然，它们极富挑战性。但是我并不倾向于认为计划设计会比以“编码加修正”的最常见方式付诸实践的演进设计要差。实际上，我更喜欢计划设计而不是“编码加修正”的开发方式。但是，我知道计划设计存在的问题，并且在寻求一种新方法。

1.2 XP 中的能动实践

XP 因为许多原因而备受争议，但 XP 中最具有争议性的关键之一是它提倡演进设计而不是计划设计。众所周知，演进设计由于即兴的设计决策和软件开发失序而行不通。

理解这种观点的核心是软件变化曲线。这一变化曲线表明，随着项目的进展，进行改动的代价会呈指数增长。这种变化曲线以阶段的方式来表达就是“分析阶段花 1 美元的变化在生产阶段修正要花数千美元”。具有讽刺意味的是，即便大多数项目在一个没有分析阶段的即兴过程中仍能运转，但这种成本上的指数特性依然存在。指数变化曲线意味着演进设计可能行不通。它还揭示了为什么计划设计必须小心翼翼地进行，因为在计划设计中犯的任何错误都要面对同样的指数特性。

XP 背后的基本假定是有可能将变化曲线拉平到足以让演进设计行得通。XP 拉平曲线并且能利用这种效果。这是 XP 各实践方法相配合的部分结果。特别地，在 XP 中，不启用拉平曲线的那些部分就不能进行要利用平整过的曲线的那些部分。这是引发对 XP 争议的一个通常来源。许多人不理解能动（enabling）和利用（exploiting）之间的关系就进行批评。批评经常源于批评者自己的经历，在这样的经历中他们没有做能动的实践方法好让利用的实践方法行得通。结果他们遭受了挫折，并且当他们看到 XP 时就会记起挫折。

能动实践（practice）有许多部分。核心是测试（Testing）以及持续集成（Continuous Integration）的实践。没有测试所提供的安全性，XP 的其余部分就无从谈起。持续集成对于保持团队的同步非常必要，因此你可以进行改动而不用担心和其他人的集成问题。这些实践方法共同对变化曲线产生很大影响。我在 ThoughtWorks 公司再次体会到了这一点。引入测试和持续集成对开发工作有了显著的改善。改善的程度好到会让人怀疑 XP 的断言，需要用上全部的实践方法才能获得巨大改善。

重构（refactoring）具有类似的效果。以 XP 建议的规范方式重构其代码的人们发现他们的收效与进行更松散、更即兴的重组相比有明显的不同。那肯定是 Kent 指导我正确重构后我所经历的收效。毕竟只有这样的一种剧变才会促使我撰写一整本关于它的书籍。

Jim Highsmith 在其对 XP 的精彩总结中用天平打了个比方。在一个托盘中是计划设计，另一个托盘是重构。在较传统的方法中计划设计占优势，因为前提是你以后不能改变主意。随着变化代价的降

低，你可以在以后采用重构进行的设计也更多。计划设计不会完全消失，但是现在能用两种设计方法的一种平衡来工作。对于我来说，就觉得在使用重构之前我是在用一只手完成自己所有的设计。

这些持续集成、测试和重构的能动实践提供了一种令演进设计可行的新环境。但是，我们尚不能确定的一件事是平衡点在哪里。我能肯定，不论外部印象如何，XP 也不仅仅是测试、代码和重构。在编写代码之前有进行设计的必要。其中有一些设计工作是在任何编码工作之前，大多数设计则出现在针对特定任务的阶段过程中编写代码之前。但是在前置设计和重构之间有一种新的平衡。

1.3 简单的价值

XP 中大声疾呼的两种口号是“做可能有效的最简单的事情（Do the Simplest Thing That Could Possibly Work）”和“你将不会需要它（You Aren't Going to Need It）”（也称为 YAGNI）。两者都是 XP 简单设计（Simple Design）实践的写照。

YAGNI 方法时常会被提起，它的意思是不应该在今天添加任何只供明天需要的功能所使用的代码。从表面上判断，这似乎很简单。问题会伴随着诸如框架、可重用部件以及柔性设计这类东西而出现。这类东西构建起来很复杂。构造它们所花费的额外的前期成本可以期望以后收回。在前置设计中建立柔性的思想被看作是软件设计有效率的一个关键部分。

但是，XP 建议不要为第一次需要某种功能的情况而构建柔性的部件和框架。让这些结构随着对它们的需要而成长。如果我今天要一个只需处理加法不必处理乘法的 Money 类，那么我就只在 Money 类中建立加法。即使我能肯定下一阶段将会需要乘法，还知道如何轻松地做到，并且认为确实很快就能做到，我还是会把它留待下一阶段再做。

这样做的一个理由是效益。如果我去做只用于明天所需功能的任何工作，那就意味着我没有把精力放在完成目前阶段所需的功能上。发布计划表明现在需要干什么，从事其他将来的事情有悖于开发人员与客户间的协议。这会冒本次迭代的素材（story）可能完成不了的风险。即使本次迭代的素材不会有风险，也要由客户来决定应该干什么额外工作——而且可能仍然不会包含乘法。

这种经济效益上的限制会因我们有可能把它搞错而加剧。就算我们已经明确知道这个功能应该如何运作，我们还是有可能把它搞错——特别是此刻我们还没有取得详细需求时更是如此。提前做一件错事比提前做一件正确的事情更浪费时间。而且 XP 专家们通常相信我们更有可能会做错而不是做对（我也有同样的感受）。

采用简单设计的第二个理由是它与轻装上阵法则（principle of traveling light）相反。复杂的设计

比简单的设计更令人难以理解。因此添加的复杂性会使对系统的修改更加困难。这就增加了加入更复杂的设计和需要更复杂设计之间那段时间的成本。

现在许多人为发现这样的建议是无意义的而感到震惊，而且他们那样想是对的。因为你所想象的一般软件开发环境中 XP 的能动实践并没有处在适当的位置。然而，当计划设计和演进设计之间的平衡有了变化时（而且也只有在这样的变化发生时），YAGNI 就变成了良好的实践。

因此结论是，不要浪费精力去增加直到将来的阶段才会需要的新功能。而且即使这样做的成本为零，也不要去做，因为就算现在加入这些功能并不增加成本，也会增加将来进行修改时的成本。总之，你只要在使用 XP 或者一种能降低修改成本的类似技术时明智地依此行事即可。

1.4 究竟什么是简单

因此，我们希望自己的程序代码尽可能简单。这听起来没有什么好争论的，毕竟有谁想要复杂呢？但是问题来了，“什么是简单呢？”

在《解析极限编程——拥抱变化》一书中，Kent 给简单系统制定了四个评价标准[Beck2000]。依次为（最重要的排在最前面）：

- ✧ 通过所有测试；
- ✧ 体现所有意图；
- ✧ 避免重复；
- ✧ 类或者方法数量最少。

“通过所有测试”是一项相当简单的评价标准。“避免重复”也很直截了当，尽管许多开发人员需要指点才能做到。比较有技巧的是做到“体现所有意图”这一条。这到底是什么意思呢？

这里的基本准则就是代码的简洁明了。XP 对易读的代码赋予了很高的评价。在 XP 中“巧妙的代码（clever code）”是个用得过多的字眼。不过，有些人体现意图的代码对其他人来说就是一种巧妙。

Joshua Kerievsky 在其 XP 2000 的论文中举了一个这样的好例子（见本书第 13 章）。他查看了可能是大家最熟知的 XP 代码——JUnit。JUnit 用若干修饰性代码给测试案例增加了可选的功能，比如并发同步和批安装代码。通过把这类代码区分为修饰性代码的方法，基本代码比以前更加清晰了[Gamma+1995]。

但是你必须要问问自己结果代码是否确实简单。对于我来说是，但我熟悉修饰（Decorator）模式。不过对于许多不熟悉的人来说，这相当复杂。类似地，JUnit 使用可插接的方法，我注意到大多数人一开始都没有发觉它的清晰。于是我们可以总结为，JUnit 的设计对于有经验的设计者来说比较简单，但对缺乏经验的人来说比较复杂吗？

我认为针对把重复减到最少的目的，XP 的“做且仅做一次（Once and Only Once）”以及 Pragmatic Programmer 的 DRY（Don't Repeat Yourself，不要自我重复）都是直观和作用极强的好建议。只要遵循这些建议就能让你保持长期有进展。但它并不是全部，简单性仍然是一件确定起来很复杂的事情。

最近我参加了某项被过度设计的工作。它得到了重构而且去除了一些柔性设计。但正如一位开发人员所说的那样“重构过度设计的工作要比重构没有设计的工作简单”。最好要比你需要的程度再简单一点，但稍有点儿复杂也不算是失败。

对此我所听过的最好建议来自 Uncle Bob（Robert Martin）。他的建议是不要在确定什么是最简单的设计上过于拖延。毕竟你能够，也应该，而且必将在以后对其进行重构。到最后，愿意重构的态度要比立即知道最简单的是什么重要得多。

1.5 重构违反了 YAGNI 吗

这个话题最近出现在 XP 的邮递列表中，而且值得我们了解设计在 XP 中的角色时讲述一番。

这个问题基本上源自于这样的观点，重构花费了时间但没有增加功能。因为 YAGNI 的观点是要求你为目前而不是将来进行设计，这是一种违规吗？

YAGNI 的观点是你不必增加目前的素材不需要的复杂性。这是简单设计的实践的一部分。需要重构来尽你所能保持设计简单，于是只要你认识到你能做到更简单，就应该进行重构。

简单设计既利用了 XP 的实践，同时也是一种能动实践。只有有了测试、持续集成以及重构，你才能有效地实施简单设计。不过，同时保持设计简单对于保持变化曲线平坦来说是非常必要的。任何不必要的复杂性会让一个系统在除了你刻意加入复杂的柔性之外的其他所有方面都更加难以改变。总之，人们并不擅长有预见性，因此最好坚持保证简单性。总之，人们初次并不能做到最简单，因此需要进行重构来更接近目标。

1.6 模式和 XP

JUnit 的例子不可避免地让我要提到模式。模式和 XP 之间的关系很有意思，而且这还是个常见的问题。Joshua Kerievsky 主张模式在 XP 中是次要的，并且他详尽地论述了这一观点，因此我不想赘述了。但值得牢记的是，对于许多人来说模式似乎和 XP 是矛盾的。

这一观点的实质在于模式经常被运用过度。世界上到处都有非凡的程序员，他或她在初读《Design Patterns》一书时就能在 32 行代码中包含 16 种模式[Gamma+1995]。我记得有天晚上，在一点好酒的作用下，我和 Kent 讨论了一篇名为“No Design Patterns: 23 Cheap Tricks（不要设计模式：23 条简便技巧）”的论文。我们当时认为这些技巧不过是使用一条 if 语句而不是一种策略。这则玩笑说明了一点，模式经常被运用过度，但那并不会让模式成为一种不好的想法。

对此有一种理论认为简单设计的力量将把你引入模式。许多重构明确地在这样做，但只要遵循简单设计的规则，甚至没有重构你也会遇到模式，而不管你知不知道它们。事实可能就是如此，但它是最好的做法吗？当然，如果你大致知道自己的目标，并且有本书能帮你解决问题而不是被迫自己去克服的话，就更好了。只要我感到要用到模式了，我肯定还会参考 GOF（译者注：GOF 是指设计模式方面的经典书籍《Design Patterns》。该书有四位作者：Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides，所以该书经常被称为 GOF，GOF 是 Gang of Four 的缩写，意思是“四人帮”）。在我看来，有效的设计要求我们知道采取一种模式所花费的代价是值得的——这是我们自己的技能。类似地，正如 Joshua 所建议的那样，我们要对如何轻松地逐渐引入模式更加熟悉才行。在这点上，XP 对待我们使用模式的方式和其他人使用模式的方式有所不同，但肯定不会丧失它们的价值。

不过，在阅读了一些邮件列表之后，我明显感到，尽管大多数 XP 的倡导者同时也是推广模式运动的领导者，但还是有许多人以为 XP 不鼓励使用模式。这是由于他们的看法超出了模式的范畴呢，还是由于模式的概念在其意识中根深蒂固以至于他们再也感觉不到了呢？我不知道其他原因，但对我来说，模式仍然是至关重要的。XP 可能算是一种开发过程，而模式是设计知识的主干，无论对什么样的过程，知识都是有价值的。不同的过程使用模式的方式可能不同。XP 既强调直到需要时才使用模式，又强调通过简单的实现来逐渐演进到模式。但是模式仍旧是要掌握的一项关键知识。

我给使用模式的 XP 用户的建议有：

- ✧ 在研究模式上投入时间。
- ✧ 注重应用模式的时机（不要太早）。

✧ 注重在初次以最简单的形式实现模式，以后再增加复杂性。

✧ 如果加入了一项模式，并且以后又意识到它的量级无法控制——不必担心再次将其剔除。

我认为 XP 应该更加重视对模式的研究。我不能肯定我将如何把模式融入 XP 的实践，但是我能肯定 Kent 会找出一条途径。

1.7 UML 和 XP

从我支持 XP 的事实引发的所有问题之中，最大的问题之一是围绕着我和 UML 的关系展开的。它们两者不相容吗？

不相容的地方有若干。XP 的确在很大程度上不重视图表。即使其表面上遵循“如果有用就用”的原则，但却包含强烈的潜台词“真正的 XP 用户不画图”。人们喜欢 Kent 根本不在乎图表的事实又强化了这一点，实际上我从未看到 Kent 自愿地以任何确定的符号画过软件图。

我想这个问题有两个独立的起因。一是源于这样的事实，有些人发现软件图有帮助而有些人没有。危险在于一方认定另一方应该按自己的想法去做。与此相反，我们应该接受有些人用图而有些人不用的现实。

另一个原因是软件图表往往和重量级的过程相关。这样的过程花费了大量时间来画既没有帮助实际上也没有坏处的图表。所以我想人们应该仔细考虑如何合理使用图表以及避免陷阱，而不只是常常出自 XP 专家的教训之词“只有你必须（当笨蛋）时才用”。

因此，下面是我对合理使用图表的建议。

首先要牢记你正在为什么而画图。其主要的准则在于交流。有效的交流意味着挑出重要的东西，忽略不重要的东西。这种选择性是很好地使用 UML 的关键。不要画所有的类——只要画出重要的类即可。对于每个类，不要画出所有的属性和操作——只要画出重要的即可。不要为所有的用例（编注：本书译者对“use case”采用的是“用况”这个译法，对于这个一直有争议的问题，本套丛书统一为“用例”）和情节（scenario）画序列图（sequence diagram）——只有你一个人用这个图。通常使用图表时所带来的一个常见问题是人们试图让其更为全面。代码是理解信息的最好来源，因为它是最容易和代码保持同步的东西。对于图表来说，全面是可理解的敌人。

图表的一种常见用途是在开始编写代码之前研究整个设计。你常常会有这样的印象，这样的行为在 XP 中是不合规定的，但这种印象并不正确。许多人认为，当你有一项困难任务时，首先召集人员

展开一次简短的设计讨论是值得做的。无论你在什么时候有这样的讨论：

- ✧ 保持简短。
- ✧ 不要试图明确所有的细节（只要有重要的地方即可）。
- ✧ 把得到的设计作为草稿，而不是最终的设计。

最后一点值得展开说明。当你进行某种前置设计时难免会认为设计的某些方面不正确，而你只能在编写代码的时候才会发现。倘若你以后改变了设计，那就不成问题。当人们认为设计已经完成，而且以后不采纳他们从编码中获得的知识并将其融入到设计中的话，麻烦就来了。

改变设计不一定意味着改变图表。绘制图表来帮助你理解设计，然后再丢掉图表是最为合理的。画图有帮助，这就足够了。它们不必成为永久保留的产品（artifact）。最好的 UML 图也不是产品。

许多 XP 用户使用 CRC 卡片。那和 UML 并不冲突。我一直都在混合使用 CRC 和 UML，选择其中最适合手头工作的一种技术。

UML 图的另一用法是在编制文档上。其通常的形式是一种立足于 CASE 工具的模型。它的思想是，开展建档工作，帮助在系统上工作的人。实际上，它往往什么忙也帮不上。

- ✧ 它花费的时间太长以至于无法保持图为最新，所以它们失去了和代码保持同步。
- ✧ 它们隐藏在 CASE 工具或厚厚的包装之中，所以没有人能看到它们。

因此，从这些观察到的问题得出关于编制文档的建议为：

- ✧ 只使用不怎么困难就能跟上更新速度的图。
- ✧ 把图置于每个人都能看到的地方。我喜欢把它们张贴在墙上。鼓励人们用笔就简单的改动对墙上的副本进行编辑。
- ✧ 注意人们是否在用它们，如果没用就丢弃。

使用 UML 的最后一个方面是在移交场合下用于编制文档工作，比如当一个小组向另一个小组移交工作的时候。此时 XP 的观点是生成文档的情节和任何其他情节一样，因而要由客户来决定其业商业价值。倘若有选择性的图表对交流有帮助的话，那么 UML 在这里也会有用。请记住，代码才是详细的信息库；图表只担当了总结和突出重点的作用。

1.8 关于隐喻

我可能也公开地提到过它：我尚未把握这种隐喻的意义。我看着它工作，而且在 C3 项目上工作得不错，但是这并不意味着我对怎样去做有任何思路，更不用说如何解释怎样去做了。

XP 中的隐喻（Metaphor）实践建立在 Ward Cunningham 的名字系统方案之上[Cunningham1995]。其关键之处在于你要提出一种熟知的名字集合来充当讨论领域的一个词汇表。这种名字系统在你给系统中的类和方法取名时发挥作用。

我通过构建一个领域的概念模型建立了一个名字系统。我和领域专家使用 UML 或其前身共同来完成这项工作。我发觉你必须要小心翼翼地去做。你要维护一个最小而简单的记号集合，而且你必须要防备让任何技术问题钻进这个模型。但是，如果你做到了，那么我发现你能用它来建立领域的一个词汇表，领域专家能够理解并用它和开发人员交流。这种模型不能和类的设计精确匹配，但是它足以给出整个领域的一个常用词汇表。

现在我没有看到有任何理由说这个词汇表不能是个隐喻性的词汇表，比如能将薪水册转变为一条工厂装配线 C3 隐喻。但是我也没有看到为什么你的名字系统建立在领域词汇表的基础上是个不好的想法。我也不会在获得名字系统时放弃对我来说不错的技术。

你确实至少需要系统的某种概要设计，人们经常因此批评 XP。XP 用户常常用“那是隐喻”的答复来回应。但是我仍然不认为我已经看到了以一种令人信服的方式对隐喻给予了阐述。这是 XP 中真正的难点，也是要 XP 用户解决的问题。

1.9 所以，设计死亡了吗？

即使不是全部，但设计的特性的确改变了。XP 设计要求的技能如下：

- ✧ 保持代码尽可能清晰简单的坚决愿望。
- ✧ 重构技能，有信心随时根据需要进行改进。
- ✧ 对模式的良好知识背景 :不仅知道解决方案而且知道何时使用它们以及如何逐步演进到它们。
- ✧ 知道如何把设计思想灌输给需要了解它的人，使用代码、图表以及首要的方式：交谈。

那是些令人畏惧的技能，但成为一名良好的设计者总是艰难的。XP 确实不会让这事更容易，至少对我来说不简单。但是我认为 XP 的确给我们提供了一条新的思路来考虑有效的设计，因为它使演进设计再次成为一种可行的策略。而且我是演进方式的一个爱好者——否则谁知道我可能成为什么人呢？

1.10 参考文献

[Beck2000] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000

[Cunningham1995] W. Cunningham. "EPISODES: A Pattern Language of Competitive Development." In *Pattern Languages of Program Design 2.* Vlissides, Coplien, Kerth, eds. Addison-Wesley, 1995

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

1.11 致谢

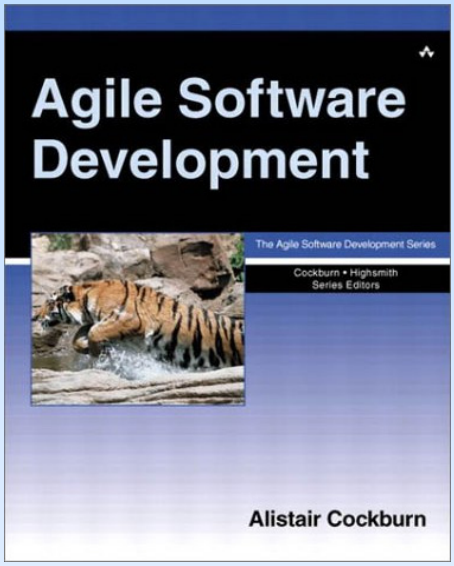
在过去两年中，我从许多善意的人那里获得了许多好想法。这些想法的大多数都在我的记忆中模糊遗忘了。但是我仍能记得来自 Joshua Kerievsky 的好点子。我也记得来自 Fred George 和 Ron Jeffries 的许多有帮助的评论。我还不能忘记从 Ward 以及 Kent 那儿获得的好想法。

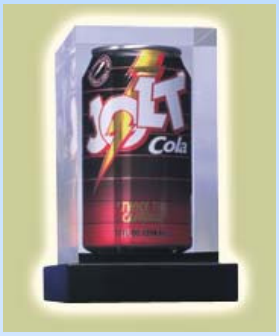
我总是向那些提出疑问和指出错误的人们表示谢意。我一直疏于留下这两类需要表达谢意的人的名单，但是他们中间肯定包括 Craig Jones、Nigel Thorne 和 Sven Gorts。

1.12 关于作者

Martin Fowler 是位于伊利诺斯州芝加哥市的 ThoughtWorks,Inc.公司的首席科学家。作者其他的联系信息和书面文章可以在 www.martinfowler.com 找到。

《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。

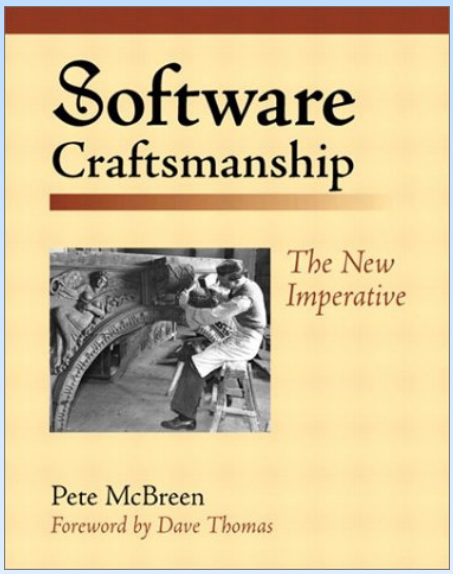
“是红酒。”我坚持道，并把标签指给她看。

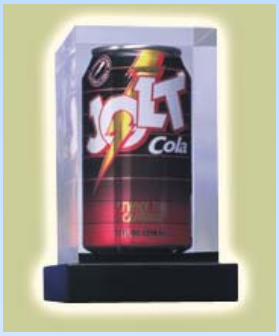
“当然不是红酒。这里说得很明白...”她开始把标签大声读出来。“噢！是红酒！我为什么会把它放进冰箱？”

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！

《软件工艺》





2002 Jolt Awards 获奖书籍

Pete McBreen

翻译：UMLChina 翻译组 Jill

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！