第 45 期（2005 年 1 月）

【新闻】

1 微软："UML and more"的延期和期待…

【访谈】

8 重构极限编程（下）

【方法】

20 《UML参考手册》2.0中译本摘录

32 《对象设计》中译本样章

42 UP实作的一些常见问题（中）

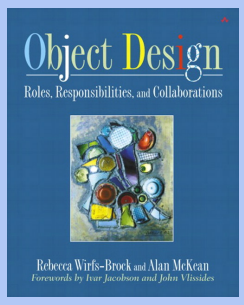
48 《About Face 2.0》中译本精彩节选

61 用例点估算方法

66 扩展敏捷思考用于业务改进

【书籍】

73 《人月神话》和《功夫》……



回归

X 非程序员
-Programmer
软 件 以 用 为 本

联系：think@umlchina.com

http://www.umlchina.com/

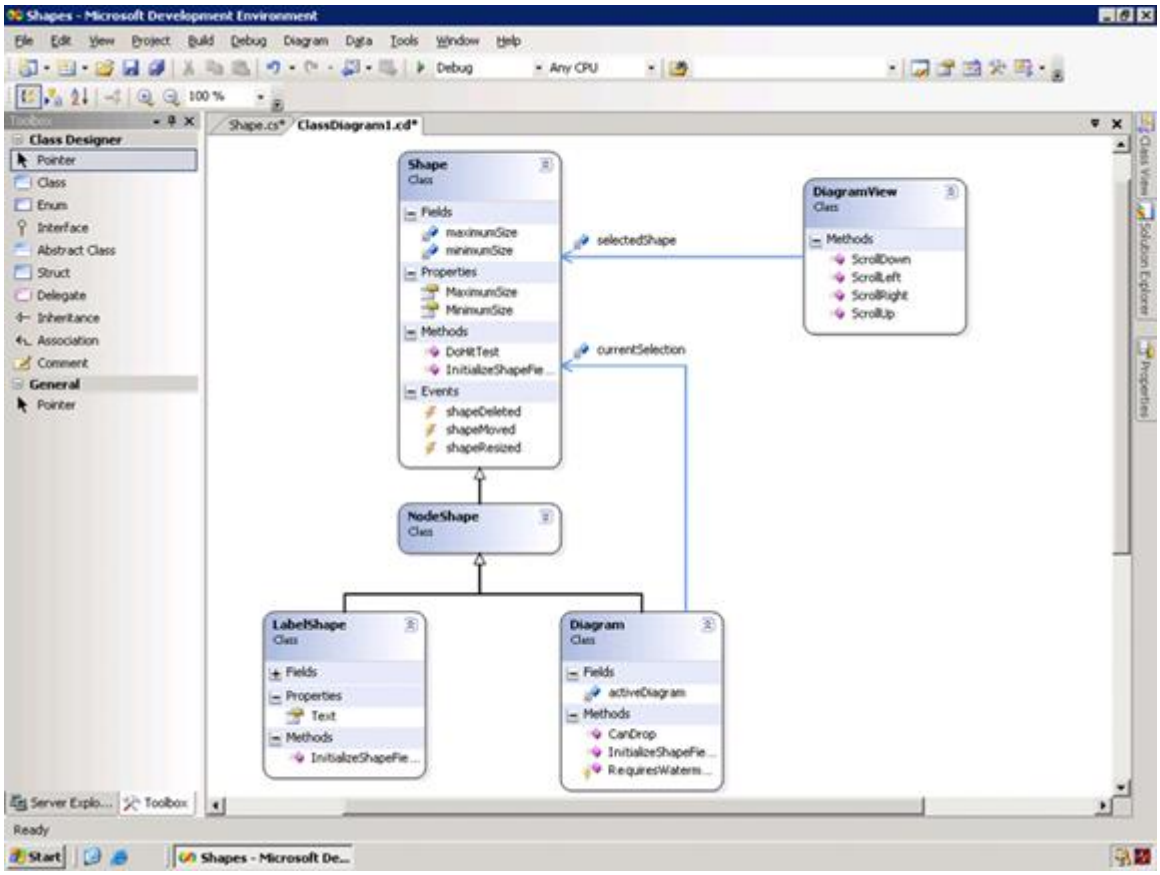
本电子杂志免费下载，仅供学习和交流之用
文中观点不代表电子杂志观点
转载需注明出处，不得用于商业用途

微软："UML and more"的延期和期待

[2005/1/1]

微软并没有如期发布大家期待已久的 Whidbey、Yukon 或者 Longhorn，它们原定是在 2004 年发布的。但 2004 年，微软给出了扩展 Visual Studio 产品线的计划，为 Visual Studio 增加一个高端的团队系统，其中包括微软自己的建模、测试、变更管理工具。除此之外，也有精简 Express 版本的计划，以回应该工具日益臃肿的指责。

除了这些产品的延期和对未来的承诺之外，微软在 2004 年发布了四个服务器。去年 3 月发布集成服务器 BizTalk 的新版本，支持 BPEL（Business Process Execution Language）以及和 Visual Studio 的协作；另外，还有 Windows System Server 家族的新成员，支持语音应用开发的 Speech Server；MapPoint Location Server，可以为地产销售人员提供精确定位的能力，增加到已有程序中。去年 8 月，微软更新了其 Host Integration Server，以简化利用.NET 应用来访问其它系统中数据的流程。



等啊等

微软宣布了 Whidbey（Visual Studio 的下一版本）以及 Yukon（SQL Server 的下个迭代版本）的延期。 两个都很时髦，但 Visual Studio 的延期对开发人员的冲击要小一些，而后者更加严重。SQL Server 的支持.NET 新版本的延期发布将不利于微软开发平台的推广。更麻烦的是导致微软在和 IBM 以及 Oracle 的数据库的战斗中处于不利之地。后两家一直在不断地升级他们的企业级产品，而微软在 2000 年来并没有对 SQL Server 做什么改进。

有消息称 Longhorn（Windows XP 的后续版本）将继续延期。去年 4 月，微软宣布 beta 版本将在 2005 年发布，到夏天的时候又宣布要到 2006 年才可以发布。更糟的是，其中不包括 WinFS－基于 XML 的存储系统。去年夏天还有更多其它的坏消息：for 64 位系统的 Windows Server 2003 以及 Windows Server 2003 SP1 被延期到 2005 年上半年。

另外，2004 年是微软“与他人和睦相处”的一年。微软修补了和 Sun 和 Novell 的关系，很小心地定位他们的生命周期开发方面的产品、团队系统。这些不是作为完全的微软环境发布的，而是作为开发人员可以在其它工具（包括那些竞争对手）中使用的插件。微软很狡猾地把他们计划中的建模工具称为是“UML and more”，而不想很激烈地和这个业界的建模标准语言分裂。

尽管如此，微软还是落在了竞争对手 IBM 的身后，后者已经发布了自己的生命周期开发平台 Atlantic。

（自 sdtimes，UMLChina 袁峰 摘译，不得转载用于商业用途）

征稿

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

Forrester 发布 MDA 和基于模式开发报告

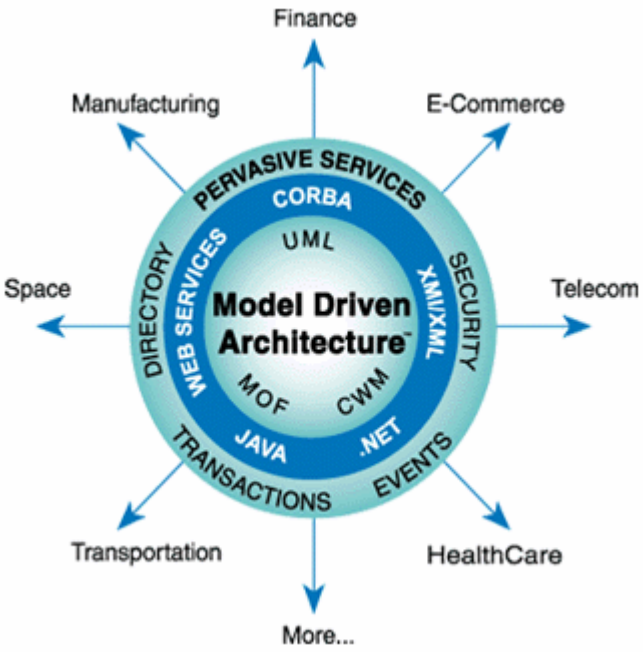
[2005/1/18]

Forrester 研究中心上周发布了两份应用发展报告，报告结果对模型驱动架构 MDA 并不正面，而看好基于模式的开发（pattern-based development）。

两份报告基于 Forrester 联合 Application Development Trends 杂志对 389 家 IT 组织的调查结果。名为：“‘模型驱动’比 MDA 更有用”和“基于模式开发的吸引”。

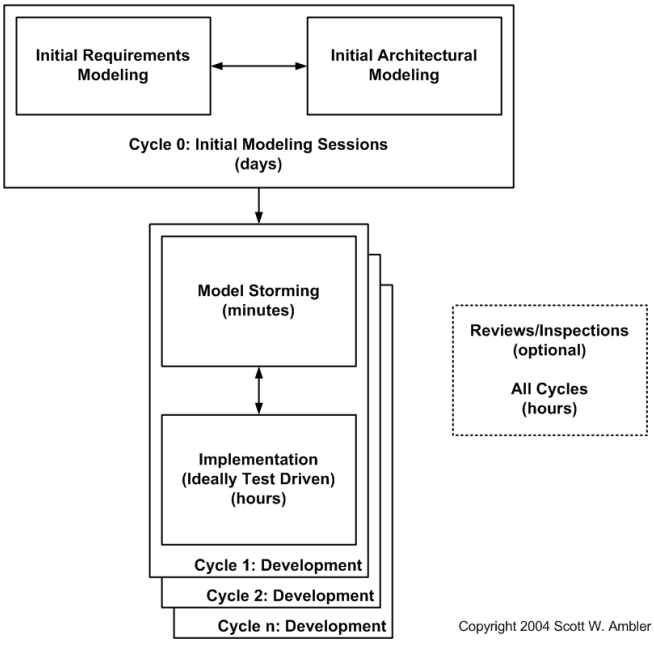
“‘模型驱动’比 MDA 更有用”报告中指出：众多 IT 组织都对 OMG 的 MDA 很感兴趣，但在实践中他们往往选择更实际可行的方法，例如模型驱动的开发（model-driven development）。

MDA 是 OMG 指定的平台无关的模型应用方式，但是，调查报告显示，目前采用这一方式的人还太少。



根据 OMG 的定义，MDA 中会创建平台无关的模型来文档化业务功能和应用的行为，并和具体实现的技术平台代码相分离。

相反，模型驱动的开发 MDD（model-driven development）中降低了 MDA 那种严格的要求，并为模型驱动的转换提供更多的实用价值；报告中指出，用户需要的是较低层次但更加实用的开发支持。



（http://www.agilemodeling.com/essays/amdd.htm）

这两份报告的作者，Forrester 应用开发和底层架构研究组的分析员 Carl Zetie 指出，软件企业在一听到 MDA 的时候都非常感兴趣，但是当听到实施 MDA 必须满足的条件时，他们退缩了。他们觉得 MDA 是正确的，但同时也是缓慢并且难于实施的。

报告指出，尽管 MDA 的工具厂商们吹得天花乱坠，MDA 产品除了转换或简单的代码生成之外，其它的开发支持了了无几。

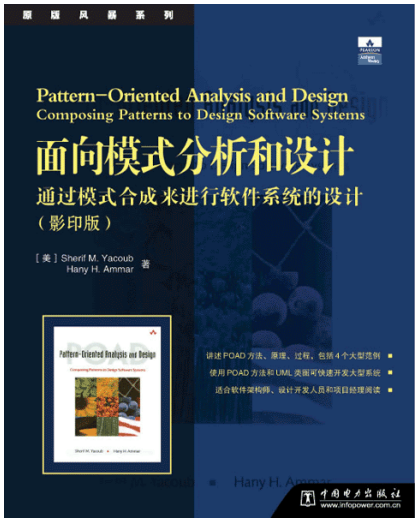
Zetie 指出，部分原因源于一些开发商把“MDA”的招牌强贴到他们的产品上，尽管他们并不是。

报告显示，很多组织都认为 MDA 将会“重要”或“非常重要”，但报告也指出，这也许是 MDA 开发商们市场行为的结果。

Zetie 指出，用户真正的需求是不要手工去写每一行代码。一旦他们实用了 MDA，他们会发现这并不是最好的解决方法。

同时，Forrester 的报告“基于模式开发的吸引”从生产力的角度关注了自动化代码生成的限制和好处以及带来的灵活性。

报告指出，模式封装了“最佳实践”的支持，基于模式的开发使用模式来解决一些公共的设计问题、技术实施并解决垂直领域的一些特定问题。



调查显示，被调查人员几乎都用过代码生成，但大多数人用得很少。由于代码生成的种种约束，开发人员缺乏经验并且和模型紧密绑定。

Zetie 指出，基于生成的解决途径几乎没有弹性，并且生成过程非常繁复。对于 100%采用代码生成的公司来说，改动的空间几乎没有，尤其是当开发人员需要对模型进行改动时。

通过围绕特定的设计模式设计和编码、开发人员可以按照觉得合适的方式对生成的代码更加灵活地修改或者变动。不再约束于基于一个抽象模型简单地生成代码。

Zetie 指出，应用开发正在朝向一个更加注重实效的方向发展。没有人完全依赖于 MDA 或代码生成，而是在不同的程度上应用它们。

敏捷建模是实用主义的一个代表。和传统的建模方法相比，敏捷建模是轻量级为软件建模的法则和实践的集合，它正得到更多的关注。

Zetie 指出，人们使用部分的代码生成、大量的最佳实践的模式，以及大量的敏捷性（agility）来保证他们沿着正确的轨道前进。

（译者注：请注意，这两份报告是对现状的调查。MDA 仍在路上，目前的很多垢病是现状的不如人意，也许，这是成长的烦恼？……）

（自 SearchWebServices，UMLChina 袁峰 摘译，不得转载用于商业用途）

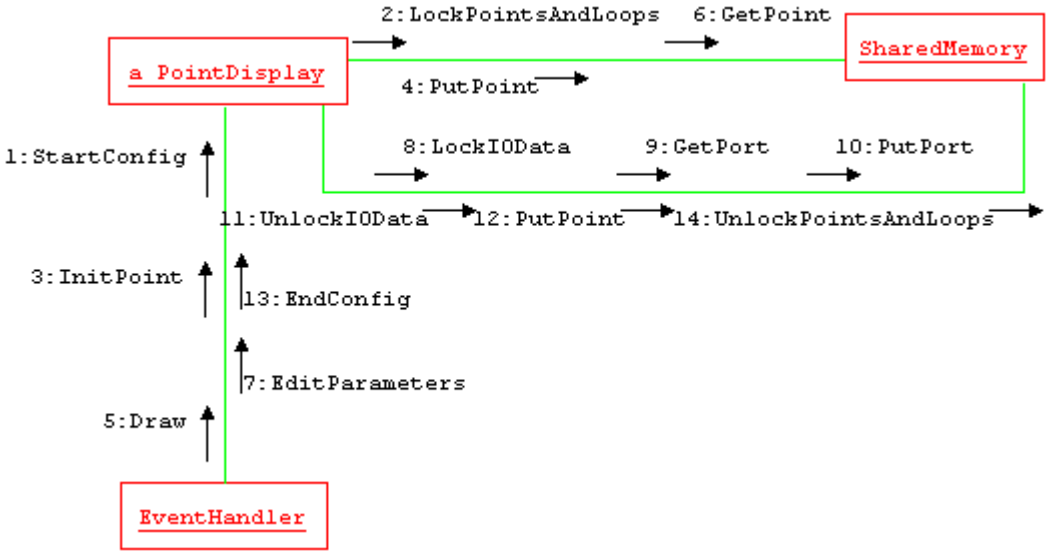
Excel 公司正式发售 WinA&D 4.0

[2005/1/3]

Excel 软件公司开始正式发售 WinA&D 4.0。这是用于需求管理、软件建模、代码生成、再工程以及报告生成的工程工具。包括 GM、Raytheon、Northrop-Grumman、Lockheed Martin、IBM、EDS 和 CSC 在内等公司的数千名软件开发人员应用 WinA&D 来进行结构化分析和设计、多任务设计（multi-task design）、基于 UML 的面向对象设计和数据库设计。

WinA&D 4.0 中的主要加强了如下功能：建模、再工程、报告生成、项目管理、度量、资金、信息的显示和打印。并基于 QuickHelp 提供了应用、报表脚本和编程接口等方面的新的在线帮助。以及 6 本用户手册及相应的 PDF 格式文件。

WinA&D 在面向对象设计方面用到了全套的 UML 符号。其中，包和类图用来显示类的结构；用例捕捉功能需求；状态图定义特定的系统状态和事件；对象的交互在交互图（communication diagram）或顺序图中表示，活动图用来表达并行行为；部署图展示硬件和软件组件的运行时关系。



WinA&D 支持多任务、实时系统的结构化分析和设计。数据和控制流图用来表示信息在系统中的流向。状态图和表用来标示方法的模式（mode）和事件处理。任务图（Task diagram）用来表示由操作系统服务关联起来的多个执行线程。结构表（Structure charts）用来表示模块结构之间的调用。

数据库设计人员使用逻辑和物理数据模型来表示表、视图、约束、断言、触发器、索引、存储过程以及其它的 SQL 元素。然后为一个 RDBMS 生成 SQL Schema 代码。

WinA&D 为需求管理提供一个集成的解决方案，支持需求的标识、描述、分组、跟踪以及控制。并且为面向对象、结构化和数据库设计提供了代码生成的支持。和 WinTranslator 联合使用，可以从代码中生成得到类模型、结构和数据模型。另外，验证、balancing、导入和导出的命令帮助对信息的处理、错误检查以及和其它用户、开发工具及平台的工作分享。内嵌的全脚本支持的报告生成工具可以生成大量的 HTML 和 WORD 格式的标准报表，从中可以浏览项目信息。

WinA&D 适用于 Windows 95 到 XP 及新的各种 Windows 版本。完全开发者版本价格为$1995。桌面版本和教育版本的价格分别为$1295 和$845。更多信息请访问 www.excelsoftware.com。

（自 thomasnet，UMLChina 袁峰 摘译，不得转载用于商业用途）

Domain-Driven DESIGN

Tackling Complexity in the Heart of Software

众多问题根源在于
领域模型没有捕获到
业务的深层内涵

Eric Evans

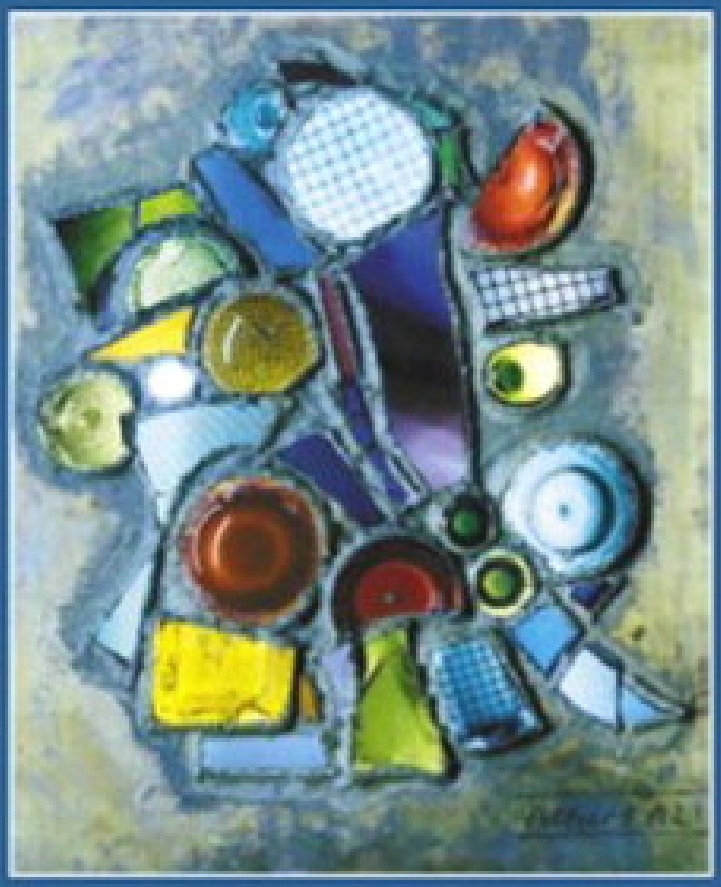
Foreword by Martin Fowler

《领域驱动设计》中译本

即将由清华大学出版社出版，UMLChina 审稿

Object Design

Roles, Responsibilities, and Collaborations



一说起“面向对象”就是“设计模式”？ ——对象的回归！

Rebecca Wirfs-Brock and Alan McKean

Forewords by Ivar Jacobson and John Vlissides

《对象设计》中译本即将由人民邮电出版社出版

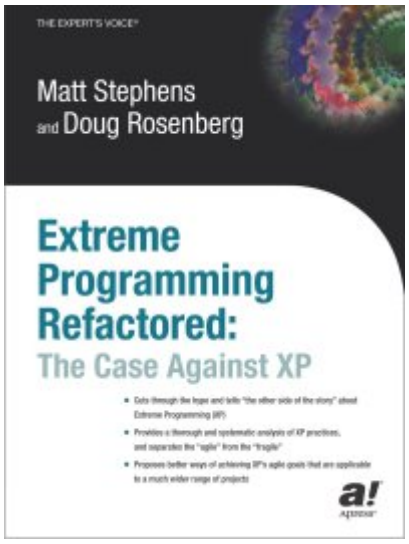
鲍志云 译，UMLChina 辅助教材

重构极限编程（下）

Mark Collins-Cope 著，张江 译

讨论

Mark Collins-Cope 采访了 Doug Rosenberg (《User Case Driven Modeling with UML》的作者) 和 Matt Stephens (与 Doug 一起合作编著《XP Refactored》，此书于 2003 年夏天发行)，关于他们对于极限编程的反对意见。



（接（上），发于《非程序员》42 期）

因此，“在有效的前提下做最简单的事” 实际上行不通。它只是按照您期望的，给您快速描述一种成功的幻觉，却不能持久。XP 让人惊喜的，就是它成功地为您创造其实并未实现的成功感觉。我们看过一些文章比如 Chet Hendrickson 的”戴姆勒 克莱斯勒：世界上最好的团队”，发表在 IEEE Computer 的 1999 年 10 月刊（在该项目取消前两个月）。但是 XP 的宣传造势机器比其他竞争者更高明的地方，就是在这个项目取消之后能够继续成功的造势。Matt 在“The Economist”上找到一篇发表于 2000 年 12 月的文章（此时项目差不多已经取消一年了），仍然在引用 C3 作为一个成功案例，认为它正用于支付八万六千名克莱斯勒员工的工资，并且把它的成功归结于 Beck 先生。我们想知道像这样的东西怎么还能发表。当然在我参与 OTUG 上争论的时候（从 Wiki 网上知道 C3 被取消之后），人们仍然在吵吵嚷嚷它是如何的成功。您知道，大多数的 XP 作者（比如 Beck, Jeffries, Hendrickson, Wells 等等）都是来自于 C3 项目。

[Mark] 因此您并不反对简单化，只是不喜欢 XP 主张的最小远景规划，对吗？

[Doug] 我觉得这种说法是实事求是的。简单设计很好。就我而言，不进行基础规划就盲目的开发未来需求，详细阐述 YAGNI（译者注：you aren’t going to need it，不需要的东西），这是不好的做法。

[Mark] XP 的设计方法，是通过重构让设计逐步显现，需要时再让它符合新的需求。XP 者特别反对大规模进行前瞻设计。这种对于大规模前瞻设计的对立看法，不是情有可原的吗？有没有经验证明当开始具体实现的时候需求改变了，设计也不再有效，花费在前瞻设计的时间大多成了浪费？

[Matt] 无论是 XP 项目还是非 XP 项目，需求变更都是它们所面临的普遍问题。它们解决这个问题的办法也基本一致，那就是通过把项目划分为小迭代来降低风险。它减少了“爆发式交货“，这个项目经理和客户都担心的问题。

短期迭代同时也意味着整个开发过程中不会有太多大规模前瞻设计和瀑布的感觉。Scrum（译者注：与 XP 并列的另一种敏捷开发方法）讲究一个月冲刺，XP 的两到三周迭代把时间更缩短了一步，但这还不是 XP 的最坏方面。

在这个阶段，每一次交付可能是也可能不是面向最终用户。如果它是（如 XP 所推荐），那么理所当然就遇到以下问题：项目在大约只完成总体的一成或者两成功能，很快就进入需要维护的状态。开发 进度不可避免会放慢，因为您现在已为用户安装一个基线，让设计突然不断大量涌现看上去并不是一个好主意。设计或者全局架构的变更会导致不可预见的混乱，在对用户提供支持和培训的时候，在维护历史数据的时候，在因此升级帮助文件的时

候，在测试数据迁移脚本的时候。解决这个问题最好的办法就是让设计超前正确 — 或者至少尽可能正确。有时候这是不可避免的：包括超越下次迭代的提前考虑。让设计正确意味着需要在设计上投入比 XP 所推荐更多的时间。

[Doug] 我的个人软件开发经验证明，在前瞻考虑上每投入一小时的时间，在编码和测试时总能节约至少三小时。不过这也有可能使前瞻设计走上极端，导致我们所称的“分析瘫痪”。但是其实也有很直接的办法实现前瞻分析和设计并且不落入分析瘫痪。这就是“用例驱动对象建模(Use Case Driven Object Modeling)”。

对此我们有很多的讨论，我认为 Matt 提出的 EDUF(Enough Design Up Front，足够的前瞻设计) 正是我们所期待的。这里 Rob 和我对于这种极端的倾向有首诗…它配披头士乐队的”Nowhere Man”这首歌的旋律….它的开头是“这是一个很酷的项目计划….紧接着就尽快开始编码…不需要任何人的任何需求”。

[Mark] 那您怎么知道何时已经获得“足够的前瞻设计”呢？

[Doug] 嗯，我再重复一下已经在我的头两本书里写的：我想知道我们将要实现的所有场景里必需的行为；我想知道有哪些领域对象；我想投入合理的精力，来识别“难点场景”，异常情况，还有我们所称的“备选行动方针”；我想看到有健壮性分析图的有效场景描述，并且有序列图来展现运行时这些对象如何通信。这些事情做起来实在不是那么印象深刻。我这样教人们已经十年了。总体来说，如果您理解您将要实现的系统，您应该能在大约十五分钟内写一个两个段落的用例描述，然后再花十分钟用一个健壮性分析图检查一下正确与否，然后可能再用半小时画一个序列图。因此如果对每一个将要实现的场景您都投入一个小时，那么您将获得数倍的时间回报。当然了，如果您不理解行为需求就要花更多的时间，因为您不得不去调研这个行为应该是怎样的。但是我宁愿这样做也不愿仓促开始编码实现。

重构

[Mark] 在 XP 出现之前，大家都忽略了一个事实：对任何迭代/增量风格的项目（以及瀑布项目的维护阶段）重构都是必要的。另一方面来说，在我们做方案的时候，设计的不断完善是我们借以符合特定需求的方法。下个迭代我们又有新的需求，前面的设计也不再最优。如果我们打算让软件不再继续变烂就必须要重构，不是吗？

并且，在大多数软件工程项目中，团队都要经历一个学习的过程。他们要学习领域知识，学习新的技术等等。在这种背景下重构很重要 — 我们在学习中认识到有更好的工作方式 — 此时我们不应该更新我们的代码来反映这种新的认识吗？

[Doug] 重构确有一席之地。总的来说我并不反对重构，我反对的是用重构来代替前瞻设计。我反对被称作“编码后定期重构”的思想，这种思想是要尽快得到能够运行和工作的最简单产品，然后再用重构来把设计逐步实现。重构无罪，但是重构不足以代替前瞻考虑。

[Mark] 因此您同意 XP 是一种把重构放进软件进度表的开发过程，并且重构确实发挥了作用？

[Doug] 与您这句话的第二部分相比，我更倾向于同意第一部分。我并没有看到作为前瞻设计的替代品之后，重构究竟发挥了什么作用。我认为如果在合理的前瞻设计之后用重构作为补充，那么重构还是有用的。但我感觉 XP 不是这样用重构的，所以我认为 XP 在重构的使用上遗漏比得到更多。我相信 XP 在轰轰烈烈开展单元测试上做了些有用的事情，然而，如果这件事本身有意义的话。

团队和个人问题

[Mark] 结对编程是 XP 另一个有争议的方面。结对编程有没有实际的好处 — 在人员交接，知识和技术方面，以及“集体大于个人的总和”这个问题上？

[Doug] 如果他们是自愿的，我没有见到结对工作有什么问题。所以我从不禁止结对编程。但是在作为程序员工作十五年之后，我把和平、安静和空间看得很重。我曾经看见一个在 IBM 的 Santa Teresa 实验室的案例，在那里他们发现如果把程序员安排在各自独立关上门的办公室，能获得比小隔间更高的工作效率。因此把所有的程序员安排在一个大房间里，我认为会带来工作效率上的大大降低。个人而言我会感觉要疯了。

我也曾经与一些极有天分的程序员工作，他们对结对深恶痛绝。特别有一位是低血糖症患者，当他的血糖降低的时候就很容易激动。他是一位很聪明的程序员，编写了海量高效并且条理清楚的代码。我和他在一起工作的很好，但是我从来没有想过要和他共用一张桌子和键盘。然而作为经理，我又不能不利用他的本领。所以我绝不会强令结对编程。

当然，我对于 XP 这种方法最大的意见，就是结对编程变成了不做前瞻设计的借口。我认为这是绝对的歪曲。

[Mark] XP 对于各种角色（架构师、首席设计师等）的看法是，不仅不该设置，也不该让这些角色在团队中自发的出现（“出现”这个词用了很多次☺）。从团队角度来说，这当然总比他们尚未在团队中证明自己的价值就获得光辉的头衔要好。

[Matt] [大笑] 希望他们能在公司的什么地方证明价值获得这个光辉头衔。实际上，XP 有一个好的方面，就是在每次迭代的时候，每个人要签下其中自己最感兴趣的任务，所以他们在团队中的自然角色确实能显现。我推荐在任何项目中这样做，无论是 XP 还是别的 — 让大家自己选择最感兴趣的事情。人们最乐意做自己擅长的事情，并且也最有工作效率。然而有些地方您也需要一分为二：总有一些琐碎杂事需要人来做。另外有一个危险就是在满屋子程序员当中，没有人愿意做设计或者测试自己写的代码，或者编写文档。他们只想编码：任何东西都只是探索，除非要求他们，否则没有什么产品能面向客户。有时候您需要处于领导位置的人（比如团队领导）来确保不好玩的事情也有人做。否则就像一房子的孩子，他们被允许整天看电视、吃冰激凌，却没人去洗碗…..

[Mark] 听上去其实像我的房子… 但是 XP 的开发方法不正是用来整天发布版本，激励软件工程师的吗 — 他们毕竟喜欢“创造点什么”。这不正是高度迭代方法开发软件的最大争议之一吗？一个有效激励的团队总能比士气低下的团队做出更好的产品吗，无论使用什么样的过程？

[Doug] 嗯，从这点来说我们应该提供免费啤酒和舞女来帮助我们做出更好的产品。您知道…这里我们需要进一步解释。说实话激励的确很好。说来也怪，我发现我教的人总是很积极的去学习一些技术，来帮助他们更好的工作。这些人通常已经认识到代码并不是一切，关于软件至少还有四件重要的东西…需求，计划日程等等。有很多种方式可以激励人。我当然不反对多发布一些版本。但是我个人倾向于不要为同一样东西做十五个版本。

集成

[Mark] XP 者看重持续集成带来的很多好处— 任何的“集成错误”都会在其中很快的暴露，得到团队的注意。

[Matt] 每日版本绝对是一件好事情。比如 Scrum 就这样建议。如果您很久不集成，代码就开始偏离，并且越来越难以合在一起。然而对我来说，XP 所主张的尽可能多次集成却有些过分了。我相信物极必反。即使发布工作版本和集成工程用不了多久 — 比如说，五分钟 — 毕竟还是要花时间的。这还得向上计算。毕竟假如团队的交流很好，并且设计也不错，代码偏离就不会成为多大问题 — 这种情况下让每个人每天集成一次就已经足够了。

测试

[Mark] XP 很注重自动功能测试、单元测试以及相关实践，比如先测试再编码。这是不是件好事情？特别是测试先于编码，迫使开发者们在开始实现前，就要仔细考虑类以及组件接口的设计。换句话说，测试先于编码是否能够保证重点放在设计清晰的接口上，并且能够减少因为实现细节而带来的接口增加？

[Doug] 您知道，我是单元测试的崇拜者，并且我认为，重视严格和自动化的单元测试正是整个 XP 所表现出的最好方面。我没有任何特别的理由反对在编写代码之前编写测试（我们教如何写用例，就是要求在编写代码之前编写用户手册，与其类似），尽管我认为如果您已经有了一个好的设计比如序列图和类图，先编写测试就没有太大必要。并且我坚决反对先写测试能够代替前瞻的面向对象分析设计这个观点。这一点（单元测试即是设计）正是我所不同意的极限见解。我们则称之为“首次测试之后的设计”。

因此，“单元测试”：非常重要和非常好的东西。“自动测试套件”：也很好且重要。“编码前写测试”：我认为很好。“单元测试是设计的替代品”：不！

这很有趣。现在回到 XP 刚出来的时候，当时我提议把我支持的前端用例驱动面向对象分析设计方法与 XP 的后端编码和测试技术综合起来。现在我仍然认为这是业界最终的方向。

当我在 OTUG 上提出这种综合的想法，印象中我收到了来自 Beck 先生的个人回应，他的语气我觉得有点奇怪。他说，所有的 XP 技术都必须是整套使用的，顺便问一下，您写过几行代码。从那天起三天里，我不断收到来自 XP 狂热者的脱机邮件，问我写过多少代码。当我告诉他们我做程序员已十五年，他们都走开了。但是这是我初次接触到的极限文化，对任何胆敢反对他们观点的人进行人格攻击。

但是我仍然认为这种前端采用用例驱动方式，后端积极地进行单元测试的综合，将会带来项目的成功。我在我们下一本书里所提到的“用 ICONIX 过程进行敏捷建模”，将是这个议题的扩展。

相关于我们一直在做的研究，Rob 和我最近几天创作的一首歌已经在我的脑子里（甚至 Matt 都还没看到过，它是新歌，但是它涉及到我们已经讨论过的一些观点）。如果您的杂志还有地方，您的读者或许对此有兴趣…它叫“单元测试编写者”，它配上披头士乐队“Paperback Writer”的旋律。

单元测试编写者

别喜欢 UML

伙计它太难了

宁愿在卡片上随便打些草稿

嘿设计已经死了

事情并不见好

我刚找到一份工作

我将成为一名单元测试编写者

单元测试编写者

单元测试编写者...者...者...

啊需求

是我心中的痛

从 Kent Beck 的书上淘到了好东西

它说这都是客户的错

它说的太对了

因此我不再关心

所以我要做一个单元测试编写者

单元测试编写者

单元测试编写者...者...者...

不要设计架构

不要有这个冲动

宁愿直接写代码实现它

每天下午五点

您知道我的工作正在进行中

进度安排不是我们的工作

伙计做单元测试编写者真有趣

单元测试编写者

单元测试编写者...者...者...

我想它提出了相当多的问题。如果您喜欢这首歌，您就一定会喜欢这本书…

文档

[Mark] “代码就是文档”这句话是 XP 的格言。大多数项目文档都会完全过时并且产生危害，因此一上来就写文档是浪费时间，这句话不对吗？

[Doug] 这里有我一个最喜欢的故事。多年以前，Mark，您曾发表一篇“Bob 大叔”Martin 的采访录，他说“极限编程者不怕口头文档”。这始终是我最喜欢的极限引文之一。他把自己的这种行为伪装为充满勇气和胆量，我们却认为应该是“我们太懒了，所以不愿为我们的工作写文档”。满是胡说八道！就是这个故事…你这篇采访录发表于三年前，我的儿子那时大约十岁，当看到我拼命的朝着这行字大笑的时候，他刚放学正在我的办公室里。很多天以后的一个早晨我正开车送他上学，他问我“爸爸，口头文档是似是而非的吗 …. 或者它是低能的?” 在 UMLWorld 大会我的一篇主题演讲上我引用了他的话，所有观众都爆发出一阵大笑。

第二年的 UMLWorld 大会上我和 Bob 大叔以及 Martin Fowler 在同一个评审小组，一个听众又提出了这个话题。我记得当从评审组传过我的话筒，看见一个人说“评审组成员，除了 Doug，都已经脱离实际了“。

然而，文档的自由主义无疑是 XP 在牛仔编码者中泛滥的主要原因之一。这些家伙说他们在奉行 XP，但他们既没有结对编程也没有做单元测试等等。他们只是“勇敢的”不给他们的工作写文档。

[Mark] 但是回到我的问题，文档很快的过时这个问题是不是该真的关注一下？我们在这个循环当中到底该做些什么 — 是花宝贵的时间来升级文档，还是继续进行开发？有没有一个中间方法仅仅维护最小集合的文档，其他的比如帮助我们思考或者帮助团队交流的文档，用完以后是不是就可以扔了？

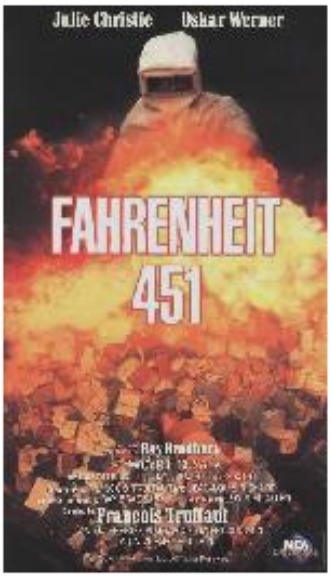
[Doug] 我是一个绝对的简单最小主义者。我不喜欢六百页这样的大文档。与其为将死的东西编写文档，我更关心前瞻考虑的过程。只要是设计过程中自然生成的文档我都喜欢。

[Matt] 需求文档，无论是什么形式 — 用例，用户情景，要点，可执行的验收测试 — 都需要不断维护，因为没有这些核对清单，就很难测试需求是否已经被实现了 — 特别是如果没有人记得都要满足哪些需求的情况下。情景卡片可能会很快结束，但是验收测试必须保持维护更新。当然这又提出一个问题，一堆可执行脚本这种编写需求的方法是否可接受— 但我认为我们已经在其他问题的答案中回答了这个问题…

我们已经为保持设计更新付出了代价 — 至少在高层次上（比如架构）。我从不理解为什么一些程序员在升级他们的设计文档时有问题。我觉得这个问题很简单，他们只是不喜欢解决它：因为它是杂活。但是它的确不会占用太长时间，特别是如果您刚开始的时候，已经花了专门时间来保证设计的正确 — 这种情况下设计会相当稳定。有一个让代码前的设计稳定的有效方法，就是（或许，可能）先写些代码：不为完成什么，只是一些例子代码让您感受一下自己在做什么。仅仅这样的少量原型（可能只花一两天）就能对设计产生深刻的影响。这也是 XP 期待提供的（比如通过编码来增进对设计的理解），但是您的确不需要为了它而“一直拨到 10（push the dial up to 10）”（这里我的意思是您不需要老做原型）。

保持设计更新的一个非常有用的好处 — 类似于当初您为什么把设计写下来 — 更新它就是再次回顾设计。仅仅重构一个类图被局限于仅仅揭露一些错误或者可能的设计改进，因为您在代码里的所有“噪音”都被隐藏起来了。

建模是纯设计；编程是…嗯，它是噪音加设计。写下设计另一个有用的好处，就是它也将被提供给团队里的其他人。不需要再花一个小时甚至几天来为新人讲解设计。相反的，只需要说“这就是设计….请随便问问题，我的桌子就在六尺外…”。这个备选办法就像科幻小说“华氏 451”。不管如何要解释整本书得需要多少时间？而且第二天还要再对团队里另一个人讲解一遍，然后是质量保证，再然后又是客户的负责人等等。



讲述的是一个关于未来的故事，在那个时代里，书籍是一种违禁品，人们一旦被查出私藏书籍，将立刻遭到逮捕，而搜出的书籍也将被焚烧。因为统治者们认为，书籍会使人产生不必要的各种幻想从而影响现实的生活。那个时代的消防员的工作不是负责救火，而是专门来烧毁书籍。451 号消防员本来工作努力，就要得到晋升，结果却在一个邻居女孩的影响下，开始私藏并阅读本应烧毁的书籍，并且从此开始对自己的人活与未来产生了怀疑。就在他决定有所行动而警醒世人的时候，他的妻子秘密告发了他……

普遍问题

[Mark] 关于 XP 现在又提出一个可实现性（Scalability）的问题…您对此是如何看待的？

[Doug] XP 在可实现性方面有很多问题。没有写下的规格需求，所有的程序员都在一个大屋子里，没有模型只有代码，等等，等等。关于 XP 的可实现性，我所听过的唯一答辩就是“嘿，看看 C3 — XP 正在这样的大项目中发挥作用”。除此以外它没有用处。我曾经在 OTUG 上问，有没有谁曾做过或者听说过任何成功的 XP 大型项目。没有人回答。我问了好几次。我仍然从未听说过一个，您呢？

总结

[Mark] 那么你们能否对我总结一下对于 XP 的观点

[Doug] 对我来说，多年前当他们努力要让 C3 项目引人瞩目的成功，结果却收获凄惨的失败，极限就已经失去可信性了。还记得“其他任何人都在卖东西”这句话吗？我认为整个社区都在卖东西，不仅仅是“其他任何人”。然后我们被要求收起对这个公认奇异的方法是否有用的怀疑。当我们在写这本书的时候，我们发现一些事实让我们直摇头。为什么 C3 项目在 2000 年一月已经取消了，声誉良好的出版物比如“The Economist”在它取消一年后仍然把它称为完全的成功？这种事情怎么会发生？无论如何，我现在很难相信这些家伙将来再宣扬的成功事例。

我还记得 “现场客户”被要求做 XP 核心要求的“良好交流”的垫脚石。但是“现场客户”的定义不断变化。我最后一次看到它（最近由 Kent Beck 引用），它被描述为“一个比编程团队更大或者至少一样大的团队”。因此我猜想如果您雇佣一个极限开发团队来编写您的代码，您必须要投入一大帮您自己的人和他们一起工作，至少得与编码人员一样多。因此就我最新看到的这个要求，难道每个键盘上必须要有两个程序员，以及陪着这对程序员的两个客户？这样难道就比编写规格书更高效？

[Matt] 批评 XP 的一个不幸后果就是大家认为您是反敏捷的。这绝对是背离我们思想的。我认为 XP 的观点就是他们的目标必须足够高尚：给客户想要的；给客户机会改变主意，甚至在项目的后期；早日提供结果以便客户能够尽早反馈，等等。但是 XP 实现这些目标的方法让我毛骨悚然。XP 本身不断演化、重新定义以及“修正”的事实（比如 Doug 刚提到现场客户的定义的变更；以及他们建议做前瞻设计的数量）说明它的基本设计是根本错误的。

[Doug] 是的， XP 为了应付它所遇到的争论而随时重定义…基本上他们随时在编造。

[Mark] Doug 和 Matt, 谢谢你们抽空接受访谈。

[Doug and Matt] 不客气。

推荐读物

http://www.softwarereality.com/ExtremeProgrammingRefactored.jsp

http://c2.com/cgi/wiki?ChryslerComprehensiveCompensation

http://c2.com/cgi/wiki?CthreeProjectTerminated

“The Best team in the World”:

http://www.computer.org/SEweb/Dynabook/DaimlerChryslerSdb.htm

Distributed Computing, October 1998. Reprinted on xprogramming.com:

http://www.xprogramming.com/publications/dc9810cs.pdf

MATT STEPHENS – Matt Stephens 已经从事软件开发超过十年了（如果您把他 11 岁时的第一次编程经验也算上，已经超过 20 年）。他的大量商业工作包括系统集成，企业架构以及团队领导。Matt 住在英国的伦敦西南。他已婚，目前作为技术架构设计师和敏捷过程培训师在伦敦中央区工作。他也是一家软件开发者和经理的讽刺网站 http://www.softwarereality.com 的编辑和定期撰稿人。

DOUG ROSENBERG – Doug Rosenberg 来自 ICONIX 软件工程公司，他已经从事系统开发工具的开发和培训超过二十年，特别是在面向对象方法上。他在 1993 年开发了 Unified Booch / Rumbaugh / Jacobson 设计方法，先于 Rational 公司的 UML 好几年。他对于对象技术制作了十多个多媒体教程，包括 COMPREHENSIVE COM 和 COMPLETE CORBA，以及 Applying Use Case Driven Object Modeling with UML.

（摘自 ObjectiveView）

"*About Face 2.0* could completely redefine how software applications are created!"
— Pete McBreen, author, Software Craftsmanship

ABOUT FACE 2.0

THE ESSENTIALS OF INTERACTION DESIGN

ALAN COOPER
& ROBERT REIMANN

当木匠看到锤子时，他不想和锤子讨论钉子的问题。他会直接用锤子钉钉子。在车里，如果司机想改变方向，他转动方向盘。司机喜欢通过合适的设备从车子和外部环境直接获得反馈：挡风玻璃外面的视野；仪表板的读数；疾驰而过的风声；轮胎压在道路上的声音；对侧向重力的感觉以及路面传来的的振动。木匠也希望有类似的反馈：钉子下沉的感觉，铁互相击打的声音以及举起锤子的感觉。

International Bestseller
Completely revised and updated

UMLChina 辅助教材。中译本即将由电子工业出版社出版，詹剑峰 译

审稿：国内首个交互设计网站 De Dream'

WILEY

TIMELY. PRACTICAL. RELIABLE.

UMLChina 指定教材

Agile Database Techniques

Effective Strategies for the Agile Software Developer

《敏捷数据》

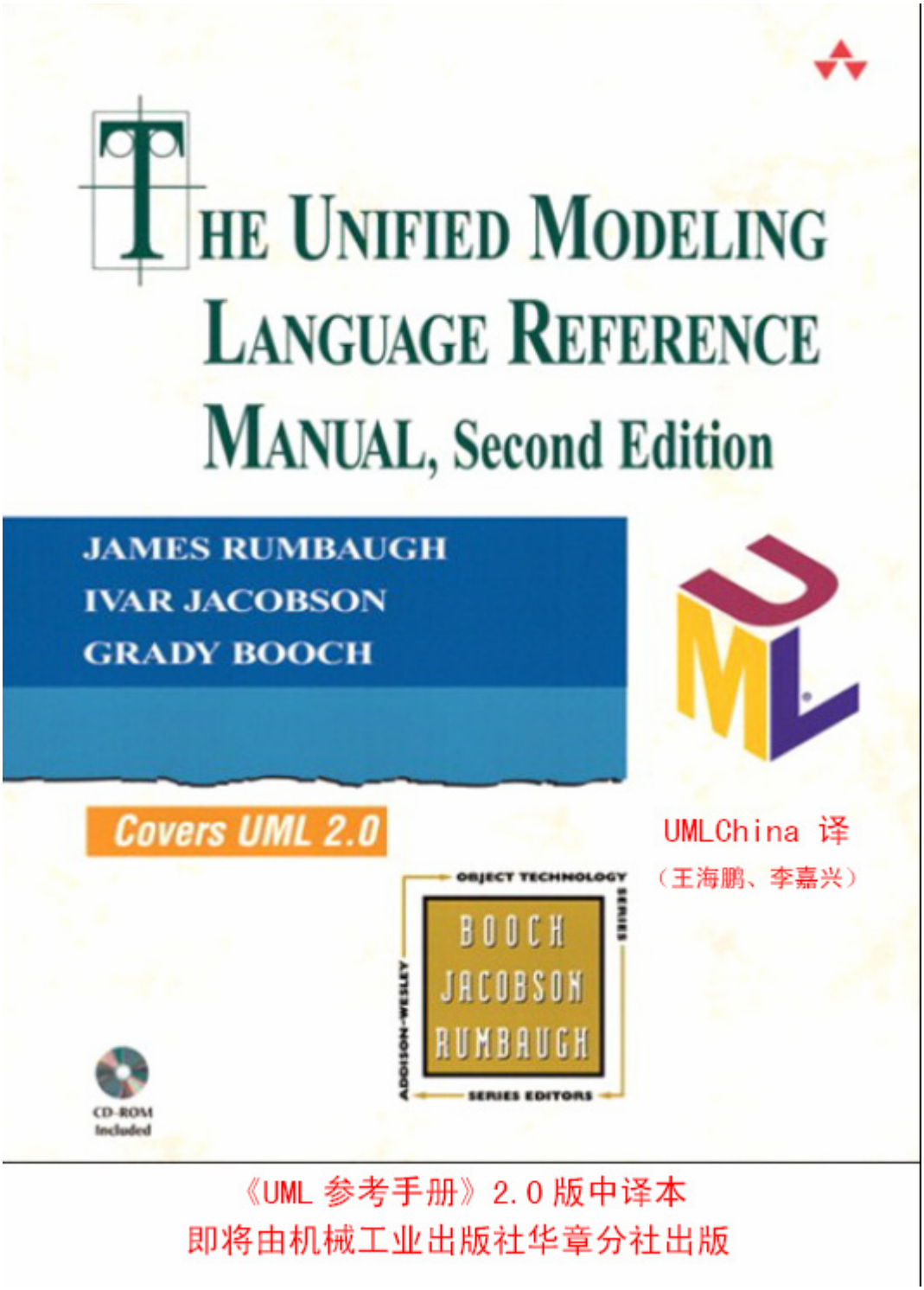
UMLChina 李巍 译

机械工业出版社即将出版

Scott Ambler

《UML 参考手册》2.0 中译本摘录（草稿）

讨论



《UML 参考手册》2.0 版中译本
即将由机械工业出版社华章分社出版

前言

本书的目的

本书是关于统一建模语言（UML, Unified Modeling Language）的一本全面实用的参考书,可供软件开发人员，软件架构师，项目经理，系统工程师，程序员，分析员，用户以及任何需要研究、设计、开发或理解复杂软件系统的人员参考。书中对 UML 的概念和组成做了详细介绍，包括其语义、语法、标记符号和用途。对广大专业软件开发人员来说，这是一本使用方便、内容全面的参考资料。此外，本书还讨论了有关标准文献没有解释清楚的细节问题和 UML 标准中一些规定的基本原理。

本书不是一本UML标准文献的指南，也不是一本关于UML标准文献中所包含的元模型内部结构的指导手册。研究开发方法的方法学家和UML工具的开发者会对元模型细节感兴趣，而一般的软件开发人员无需了解这些包含在对象管理组织（OMG, Object Management Group）文献中的不易为人理解的细节。本书涵盖了能够满足绝大部分软件开发人员需要的内容，对于某些源于原始标准的细节，都指明了其出处。 希望参考原始文献的读者可以从OMG的网站（www.omg.org）得到相关的标准文献。

在阅读本书之前，读者应具备一些面向对象技术的基本知识。为方便初学者，书后的参考文献中列出了我们和其他作者早期的原作。虽然这些书中采用的某些标记符号现在已有了变化，但是一些书中介绍的面向对象的概念仍然有用，如[Rumbaugh-91]、[Booch-94]、[Jacobson-92]和[Meyer-88]等书，所以这里没有必要重新阐述这些基本概念。[Blaha-05]使用 UML 的标记符号更新了[Rumbaugh-91]。如果读者希望初步了解 UML，学会如何使用 UML 对常见问题建立模型，可参考《UML 用户指南》（The Unified Modeling Language User Guide）和《UML 精粹》（UML Distilled）。

使用 UML 并不局限于某一种专门的开发过程，本书也不针对某一种开发过程进行讨论和介绍。尽管 UML 可用于许多开发过程，但它最适用于迭代的、增量的、用例驱动的、以软件构架健壮性为中心的开发过程—我们认为这是开发现代复杂软件最适宜的开发过程。UML 是软件开发的工具，为了将其置于这种上下文之中，本书定义了这种开发过程的各个阶段，但是这些阶段并不属于 UML 标准。《统一软件开发过程》（The Unified Software Development Process） [Jacobson-99]详细描述了这样一种开发过程，我们认为这种流程是对 UML 的补充和对软件开发的最好支持。

第二版和 UML 版本

《统一建模语言参考手册》的第二版对 1999 年出版的第一版做了全面的修改。UML2.0 规范已经获得了OMG通过，这次的第二版基于UML2.0 规范，同时包含了OMG专案小组正在筹备的正式规约中预计出现的调整。任何因为 OMG 修改 UML 规约而需要对本书做出的纠正将被公布在本书出版社的网站上（www.awprofessional.com/titles/0321245628）。到 2004 年六月为止，本书的内容都是准确的。

原始的规约文献、UML方面的最新进展以及相关的主题都可以从OMG的网站（www.omg.org）上获得。

参考手册和 OMG 规范

UML 是一门博大的、汇集了诸多特性的建模语言。一本只是复述原始规范的参考手册将不会给读者带去多大的帮助。就像任何一本字典或者百科全书那样，我们必须尽可能清楚地归纳知识，同时减少所包含资料的数量。我们常常选择略去晦涩难懂的特殊情况，或者不使用冗长的方式来表述一些概念，而是强调常见的用法。这并不意味着那些略去的技术是无用的，而是对大多数读者来说，不使用这些技术已经能够成功了。但是，《参考手册》不应该被视为 UML 语言最具权威的解释。就像任何一种标准一样，最具权威的解释来自于正式的规范，应该参考这些规范来解决出现的分歧。

我们试着遵循以下原则：

- 解释一个概念的主要意思而不过多讲述其元模型（metamodel）表现形式的具体细节。
- 不去讨论抽象元类（abstract metaclass）。建模人员最终必须使用具体的元类，具体的元类如果将其内部的各抽象层折叠起来能够更容易的描述。
- 不去讨论元模型（metamodel）的封装打包。元模型封装成包也许对工具的开发者十分重要，但是建模人员大多数情况下不需要知道。如果有读者需要知道这些信息，可以查阅具体的规范。
- 根据完整的 UML 规范描述概念。OMG 规范包含了许多中间层和明确控制点（compliance points），这些都使得理解 UML 变得非常困难。我们描述了 UML 的全部功能。如果您的工具没有实现所有这些功能，那么你也许不能使用其中的一部分，但是，了解这些功能并没有坏处。

- 从概念的常规用法角度描述概念。OMG 规范往往用非常复杂的方式来表达概念的一般性。对于规范而言，这是恰当。但是我们觉得，如果将概念放到一个特定的上下文中去，然后再一般化，读者会更容易理解这些概念。如果您担心一个概念在复杂的、具有不确定性的情况里无法应用，并且感到《参考手册》中的解释不够充分，您可以查阅原始规范。然而不幸的是，即便是 OMG 规范，在复杂的情况里有时也是模糊不清的。

本书概要

《UML 参考手册》分为四个部分：（1）UML 发展史和建模知识概述，（2）UML 基本概念综述，（3）UML 术语和概念词典，以及（4）简短的附录。

第一部分是 UML 概述—UML 发展史、目的及用途－帮助读者理解 UML 的起源和它试图满足的需求。

第二部分是 UML 基本概念的简要综述，以便读者能够认识 UML 的所有功能特性。该部分综述了 UML 所支持的各种视图，并说明了各种组件如何协同工作。该部分首先介绍了一个用到了各种 UML 视图的例子，接着分章介绍每一种视图。概述的目的不是提供一个完整的教材或对各种概念进行全面叙述，而主要是总结性地阐述 UML 的各种概念，将这些概念联系起来，它是进一步详细阅读本书中术语和概念词典的起点。

第三部分包括了各种参考信息，这些信息被组织成一个个相关主题以便于查找。本书的主体是一个按字母顺序排列的所有 UML 概念和构造的词典。所有 UML 术语，不论重要与否，在词典中都有对应条目，词典尽可能提供全面信息。因此，凡是第二部分提到的所有概念，在词典中都有更详细的进一步阐述。相同或相似的信息有时在词典中的许多条目中都予以列出，以便读者查阅。书中包含了一些常用的面向对象术语，这些术语并不是正式的 UML 概念，只是为了为例子和讨论提供上下文环境。

附录列出了 UML 的元模型（metamodel）和 UML 标记符号的总结。附录还给出了一个有关面向对象知识的主要参考文献，但不包含所涉及到的 UML 概念或其他方法的出处。参考文献中所列的许多文献都提及了一些优秀的书籍和杂志文章，有兴趣的读者可据此进一步研究这些方法和概念的形成和发展。

UML 创作人员

我们希望感谢那些参与制定 UML 规范的创作人员，他们多年来不断的举行会议，进行激烈的讨论，出版和发表自己的观点。自从 UML1.0 以来，向 UML 规范贡献自己思想的人员数量不断增加，OMG 规约已经不再列出

主要贡献者的名单，因为这份名单至少包括 20 到 50 位满足条件的创作人员。如果算上那些其工作对 UML 有影响的人员，这个数字还会更多。现在已经不可能制定出一份完整的名单了。

尤其重要的是，我们要对所有对 UML 思想作出贡献的人表示感谢。他们提出了许多有益的见解和想法，这些想法涉及面向对象技术、软件方法学、程序设计语言、用户界面、可视化编程和许许多多计算机方面的其他领域。在此我们不可能一一列举他们的名字，不经过学术上的讨论也难以理解他们的见解所具有的影响，并且本书是一本工程方面的书，并不是历史传记。这些见解有的广为人知，有的却因为提出这些见解的人运气不佳而不被人了解。

参考书目中包含了一些影响过作者但不是很出名的书籍。

第一部分 背景知识

这一部分介绍了 UML 的基本原理，包括 UML 建模的性质和目标以及 UML 覆盖的所有功能领域。

。。。。（此处略）。。。。。

UML2

有了若干年使用 UML 的经验之后，OMG 提出了升级 UML 的建议方案，以修正使用中发现的问题，并扩充一部分应用领域中所需的额外功能。建议方案自 2000 年 11 月起开始起草，至 2003 年 7 月完成。之后不久 UML2.0 规范就被全体 OMG 会员采纳。采纳后的规范经过正式的 OMG 定稿过程修正初始实现中的错误和问题后，预计在 2004 年底或者 2005 年出可得到最终的规范。

本书中，UML1 指的是 UML 规范 1.1 至 1.5 的所有版本，UML2 指的是 UML2.0 规范及更高的版本。

新特性。总的说来，UML2 和 UML1 大部分是相同的，尤其是最常用的核心功能。UML2 更改了一些问题区间，增加了一些大的改进，修正了许多小的错误，但是 UML1 的使用者应该不会有什么问题使用 UML2。新的规范可以被看作是一种编程语言或一个应用程序的新版本。一些对使用者明显的重要改变有：

- 序列图（sequence diagram）中的元素和标记符号主要基于 ITU （International Telecommunication Union）制定的消息序列图标准，从而使其更具面向对象的特性。
- 将活动建模(activity modeling)概念从状态机中分离出来，并使用了业务建模领域中通用的符号标记。

- 统一了 UML1.5 中添加的动作建模（action modeling）和活动建模，从而提供了更加完整的过程模型。

- 为类（class）和协作（collaboration）两者的内部组装（internal composition）添加了上下文建模（contextual modeling）元素。这些元素既允许松散且严密地封装也可以联结来自于更细小模块的内部结构。

- 重新定义了组件（component）代表的是设计元素，而制品（artifact)代表可部署的物理实体。

内部机制。其他改变影响的是 UML 元素（元模型）的内部表示方式和他们与其他规范的关系。这些变更不会直接影响大多数的使用者，但是对工具制造商十分重要，因为这些变动影响到了与其他多种规范的互操作能力，因此这些改变将会间接地影响使用者：

- 统一了 UML 核心部分和 MOF（Meta-Object Facility）中的概念建模（conceptual modeling）部分。这使得 UML 模型可以使用普通的 MOF 工具和储存库进行处理。

- 重新组织了 UML 元模型，去除了多余的元素并且允许重用其他规范明确定义好的元素子集。

- 给出了用来定义新域（domain）和对个别技术扩展 UML 的协议。

其他来源

除了上文提到的各种开发方法和稍晚出现的许多开发方法之外，一些 UML 视图还受到了个别来自非面向对象领域的巨大影响。

静态视图（static view），反映的是类和类之间的关联关系，它受到了 Peter Chen 1976 年开发的关系实体模型（Entity-Relationship model）的巨大影响。这种影响通过大部分早期面向对象方法转入了 UML。关系实体模型也深远地影响了数据库系统。如果没有这种模型，编程语言世界和数据库世界很可能不幸地发展成相互独立的形式。

状态机模型（state machine model）在计算机科学和电子工程领域已经使用很多年了。David Harel 的状态图是对传统状态机的重要扩展。它添加了嵌套状态（nested state）和正交状态（orthogonal state）。Harel 的概念被 OMT 采纳，随后又被其他方法接受并最终融入了 UML。在 UML 中，他们构成了状态机视图的基础。

UML2 序列图的符号标识来自于 ITU （International Telecommunication Union）制定的消息序列图标准[ITU-T Z.120]，稍作修改以使它更好地符合其他 UML 概念。这个标准在电信行业被广泛地使用，它替换了 UML1 中序列图的符号标识，增加了许多结构化元素以解决 UML1 中出现的问题。ITU 正在考虑是否将一部分或者全部变动加入 ITU 标准。

UML2 中结构化类元（structured classifier）的概念受到了 SDL[ITU-T Z.100]、MSC 和 ROOM[Selic-94]方法中实时工程（real-time engineering）概念的巨大影响。

UML1 以及更多 UML2 中活动图的符号标识受到了各种业务过程建模符号标识的影响。因为没有单一的业务过程建模标识符号处于主导地位，所以 UML 中的符号标识取自于多种不同的建模方法。

UML 还受到了很多其他方面的影响。一个概念的最初提出者往往不是那个因推广这个概念而出名的人。大约有 20 人是 UML1 规范的主要贡献者，许多其他人员以稍次要的方式参与。大概有 30 个左右的人员在 UML2 的开发中担任了重要角色，同时还有很多其他人员提交建议、复审方案和撰写书录。不可能将所有对 UML 做出贡献的人都列举出来。我们所列出的简要参考无疑会遗漏一些重要的贡献者，关于这点希望得到理解。

统一的意义

“统一”这个词在 UML 中有下列一些相互关联的含义：

统一了以往的方法和表示法。UML 合并了许多面向对象方法中被普遍接受的概念，

对每一种概念，UML 都给出了清晰的定义、表示法和有关术语。使用 UML 可以对已有的用各种方法建立的模型进行描述，并比原来的方法描述得更好。

跨越了软件开发的生命期。UML 对于开发的要求具有无缝性。开发过程的不同阶段可以采用一套相同的概念和表示法，甚至在同一个模型中可以混合使用它们。在开发的不同阶段，不必转换概念和符号标识。这种无缝性对迭代式的、增量式软件开发是至关重要的。

跨越了应用领域。UML 适用于大多数应用领域的建模，包括大型的、复杂的、实时的、分布式的、集中式数据或精密计算的系统。也许用某种专用语言来描述一些专门领域更有用，但在大部分应用领域中，UML 不但不比其他的通用语言逊色并且更好。

脱离了编程语言和开发平台。 UML 可应用于运用各种不同编程语言和开发平台实现的系统。其中包括程序设计语言、数据库、4GL、组织文档及软硬件固件等。在各种情况下，前部分工作应当相同或相似，后部分工作因各种开发媒介的不同而有某种程度上的不同。

脱离了开发过程。UML 是一门建模型语言，不是对一个具体开发过程的描述。就像通用程序语言可以用于许多风格的程序设计一样，UML 作为建模语言适用于大部分现有的或新出现的开发过程。尤其适用于我们所推荐的迭代式增量开发过程。

统一了内部概念。在构建 UML 元模型的过程中，我们特别注意揭示和表达各种概念之间的内在联系，并试图用多种适用于已知和未知情况的办法去把握建模中的概念。这个过程会增强对概念及其适用性的理解。这不是统一各种标准的初衷，但却是统一各种标准最重要的结果之一。

UML 的目标

在 UML 开发的背后，有许多目标。第一个且最重要的目标是使 UML 成为一门通用的建模语言，所有建模者都能使用。它并非某人专有，而是建立在计算机界普遍认同的基础上。UML 特意包含了主流建模方法的概念，从而可以作为它们的建模语言。至少，我们希望它能够替代 OMT，Booch 和 Objectory 方法，以及参与 UML 方案制订的其他人所使用的方法模型。UML 希望让人觉得尽可能熟悉，所以如果可能，我们都采用源自 OMT Booch, Objectory 及其他主要方法的标记符号，这使得 UML 能够很好地支持设计工作，比如封装、问题划分、记录模型构造思路。此外，我们希望 UML 解决当前软件开发中的热点问题，比如大规模、分布式、并发、模式和团体开发等。

UML 并不试图成为一个完整的开发方法。它不包括一步接一步的开发过程。我们认为一个好的软件开发过程对软件开发获得成功是至关重要的，我们向读者推荐一本书[Jacobson-99]。认识到 UML 和使用 UML 的软件开发过程是两回事，这一点很重要。我们希望 UML 可以支持所有的，至少是目前现有的大部分软件开发过程。UML 包含了所有的概念，我们认为这些概念对于基于健壮的构架来解决用例驱动式需求的迭代式开发过程是必要的。

UML 的最终目标是在尽可能简单的同时能够对实际需要建立的系统的各个方面建模。UML 需要有足够的表达能力以便可以处理现代软件系统中出现的所有概念，例如并发和分布式，以及软件工程中使用的机制，如封装和组件。它必须是一个通用语言，像任何一种通用编程语言一样。然而，这样就意味着 UML 必将十分庞大，不可能像描述一个近乎于玩具一样的软件系统那样简单。现代语言和操作系统比起 50 年前要复杂多，因为我们对它们的要求越来越多。UML 提供了多种模型，不是在一天之内就能够掌握的。它比先前的建模语言更复杂，因为它

更全面。但是你不必一下就完全学会它，就像学习任何一种程序设计语言、操作系统或是复杂的应用软件一样。

UML 的复杂性

UML 是一门博大多变的建模语言，适用于许多层次和开发生命周期的不同阶段。有人批评它过于庞大和复杂，但是对于任何现实世界中实际使用的通用应用程序来说，复杂性都是固有的，比如操作系统、编程语言、多媒体编辑软件、电子表格编辑器和桌面系统。这些应用能够变得很小，如果把他们做成玩具。UML 的开发者们并不希望它只是一个玩具。

必须从 UML 的演变过程理解其复杂性：

- UML 是带着不同目标和利益的人们达成共识后的产物。从 UML 上可以找到民主过程产物的特征。和根据单一思路开发的产品相比，它不够清晰和连贯。它包含了多余的特性（但是不同的人会对什么是多余的特性产生分歧）。UML 还包含了相互重叠的特性，这些特性并不总是能够非常好的整合到一起。而最关键的是，UML 缺少一致的视角。和一门编程语言不同，UML 适用于各类系统，从业务建模到图像处理，而一门编程语言的使用范围却相当有限。广泛的适用性往往会牺牲专业特性。

- UML 最初融合了四种或五种主流的建模方法，随后的目标变成了需要适应许多已有的符号标识方法，比如 SDL（Specification and Description Language，[ITU-T Z.100]），不同的业务建模语言（业务建模本身也没有统一的标准），行为语言（action language），状态机表示法等等。保留原有符号的要求往往导致了特性之间的不一致，并且为了照顾到一些用户群的使用习惯而包含了多余的符号标识。

- 正式规约由不同的团队编写。因此文献的各部分的风格、完整性、准确性、一致性都有很大的不同。

- UML 不是以形式化语言的方式给出的精确规范。尽管计算机界很赞成形式化，但是很少主流的编程语言是准确定义的，形式语言对专家而言有时候也是很难理解的。还需要指出的是，建模和编写代码不是一回事。在建筑业，尽管图纸是使用许多建筑工人共识的习惯，以非正规的风格绘制成的，但是还是能够成功地建成大楼。

- 语义部分有时包含了模糊的语句，没有足够的解释和示例。术语在元模型中引入，却没有很好的和其他术语区分开来。有太多细小的差别有人认为很重要但是却没有解释清楚。

- 过多的使用了泛化关系（generalization）却损失了必要的区分。从最早开始，继承总是好的这个说法就一直是面向对象的祸根。

- UML 中概念建模和编程语言表现这两方面的概念之间存在着对立关系，缺少一致的指导规范。

UML 评价

- UML 是凌乱的、不准确的、复杂的和松散的。这种看法是一种错误但也是一个事实。任何适于如此广泛应用的都一定是凌乱的。

- 你不必知道或者使用 UML 的每一项功能，就像你不需要知道或使用一个大型软件或编程语言的每一个功能一样。有一些被广泛使用的核心概念。其他的特性可以逐步的学习，在需要时再使用。

- UML 能够并且已经在实际的开发项目中使用。

- UML 不只是一种可视化的表示方法。UML 模型可以用来生成代码和测试用例。这要求适当的 UML 表述文件、使用和目标平台匹配的工具以及在多种实现方式中做出选择。

- 过多的听从 UML 专家的建议是没有必要的。有很多种正确使用 UML 的方法。优秀的开发人员会从很多工具中选出一种使用。但是不必使用这一种方式解决所有问题。你可以做出改变以满足自己的需要，如果能够得到同事和软件工具的配合。

UML 概念范围

UML 的概念和模型可以分成以下几个概念范围。

静态结构。任何一个精确的模型必须首先定义所涉及的范围，即确定有关应用程序的关键概念、它们的内部特性及其相互关系。这一组元素在 UML 中称为静态视图。静态视图用类（class）来表达应用程序中的概念，每个类由一组包含信息和实现行为的离散对象组成。对象包含的信息被作为属性（attribute），它们执行的行为被作为操作（operation）。多个类通过泛化关系（generalization）可以共享一些共同的结构。子类在继承（inheritance）的共同父类的结构和行为的基础上增加新的结构和行为。对象与其他对象之间也会发生运行时的联系，这种对象与对象之间的关系被称为类之间的关联关系（association）。一些元素之间的关系可以被归纳为依赖关系（dependency），这些依赖关系包括在抽象（abstraction）的不同层面之间、模板（template）参数的绑定（binding）、

授予许可以及一个元素使用另一个元素等。类也许会有接口（interface），接口描述了他们对外可见的行为。另一类关系是用例的包含（include）和扩展（extend）关系。静态视图主要使用类图以及类图的变体。静态视图可用于生成程序中用到的大多数数据结构声明。在 UML 视图中还要用到其他类型的元素，比如接口、数据类型、用例和信号等，这些元素统称为类元(classifier)，它们的行为很像类，只是每种类元都具有一定的限制和附加条件。

设计元素。UML 模型既可用于逻辑分析又可用于以实现为目的的设计。某些元素提供了设计单位。结构化类元（structured classifier）扩展了类，将类实现为一组通过连接符（connector）联系在一起的部件（part）集合。类可以在外部可见的端口（port）之后封装其内部结构。一个协作（collaboration）模拟的是一组在短暂的环境中相互作用的对象集合。一个组件（component）是系统中可替换的部分，它按照一组接口来设计并实现。它可以方便地被一个遵照同样规格说明的组件所替换。

部署元素。节点（node）是运行时的计算资源，它定义了一个物理位置（location）。制品（artifact）是计算机系统中表现信息或行为的物理单元。制品被部署在节点上。制品可能是一种显现（manifestation），也就是，一个组件的实现。部署视图描述了运行系统中的节点配置和节点上的制品布置，这包括显现和组件之间的关系。

动态行为。有三种方式对行为建模。一种是根据一个对象与外界发生交互的生存周期；另一种是一系列相关对象之间当它们相互作用实现行为时的通信方式。第三种是经过不同活动时执行进程的演变。

孤立对象的视图是状态机——对象基于当前状态对事件做出反应，作为反应的一部分执行动作，并从一种状态转换到另一种状态的视图。状态机模型用状态图来描述。

一个交互（interaction）包括一个结构化类元或者一个协作，连同各部分之间传递的消息流。交互用序列图（sequence diagram）和通信图（communication diagram）表示。序列图强调的是时间顺序，而通信图强调的是对象之间的关系。

活动（activity）表示的是一段计算过程的执行，用一组活动节点（activity node）表示，各节点用控制流和数据流连接起来。活动既可以模拟顺序的行为也可以模拟并发的行为。它包括传统的控制流元素，比如判断（decision）和循环（loop）。活动图可以用来展示计算过程，也可以展示组织中的工作流。

对所有行为视图起指导作用的是一组用例，每一个用例描述了一个用例执行者或系统外部用户可见的一个功能。用例视图（use case view）包括用例的静态结构和用例的执行者，连同在执行者和系统之间传递的动态消息序列，通常用序列图或者文本表示。

相互作用对象的系统视图是一种协作，一种与语境有关的对象视图和相互之间的链，通过数据链对象间存在着消息流。视图点将数据结构、控制流和数据流在一个视图中统一起来。协作和互操作用顺序图和协作图来描述。

模型组织。计算机能够处理大型的单调的模型，但人力不行。对于一个大型系统，建模信息必须被划分成连贯的部分，以便工作小组能够同时在不同部分上工作。即使是一个小系统，人的理解能力也要求将整个模型的内容组织成一个个适当大小的包（package）。包是 UML 模型通用的层次组织单元，它们可以用于存储、访问控制、配置管理以及构造包含可重用的模型片断的库。包之间的依赖关系是对包的组成部分之间的依赖关系的归纳。系统的整个构架可以决定包之间的依赖关系。因此，包的内容必须符合包的依赖关系和有关的构架要求。

扩展组件（profile）。无论一种语言能够提供多么完善的机制，人们总是想扩展它的功能。我们已使 UML 具有一定的扩展能力，相信能够满足大多数对 UML 扩充的需求而不改变语言的基础部分。构造型（stereotype）是一种新的模型元素，与现有的模型元素具有相同的结构，但是加上了一些附加约束，具有新的解释和图标。代码生成器和其他的工具对它做出不同的处理。构造型定义了一组标记值（tagged value）。标记值是一个用户定义的属性，能够应用到模型元素上，而不是系统运行时的对象上。举例来说，标记值可以表示项目管理信息、代码生成指示信息和应用域相关的特殊信息。约束（constraint）是用某种特定约束语言（如程序设计语言、特殊的约束语言或者自然语言）的文本字符串表达的条件。UML 提供了一个表达约束的语言，名为 OCL。扩展组件是一组有针对性的构造型和约束，可以被应用到用户包上去。可以为了特殊的目的开发扩展组件并将扩展组件储存到库中以供用户模型使用。扩展组件与所有其他扩展机制一样，必须小心使用这些扩展机制，因为有可能形成一些别人无法理解的方言。但这些机制可以避免发生更多根本性的变化。

（经机械工业出版社华章分社允许刊登）

征稿

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

THE UNIFIED MODELING LANGUAGE REFERENCE MANUAL, Second Edition

JAMES RUMBAUGH
IVAR JACOBSON
GRADY BOOCH

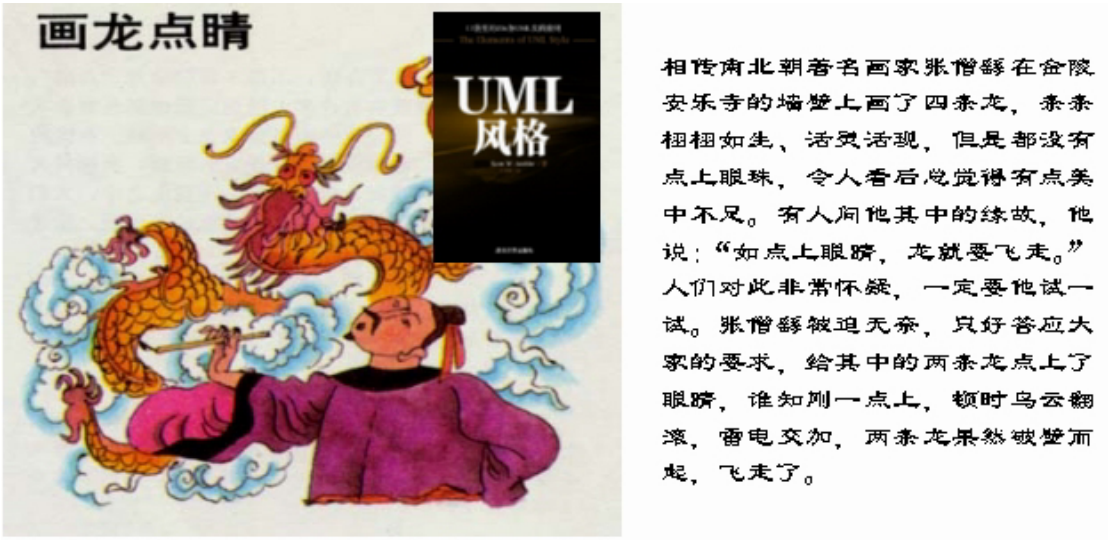
Covers UML 2.0

UMLChina 译
（王海鹏、李嘉兴）

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

CD-ROM included

《UML 参考手册》2.0 版中译本
即将由机械工业出版社华章分社出版



互动出版网 WWW.CHINA-PUB.COM
華儲® www.huachu.com.cn
dearbook.com 第二书店·第一服务

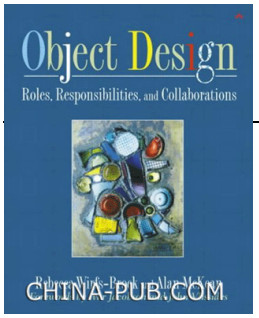
它不讲概念，它假设读者已经懂了概念。

它不讲工具，它假设读者已经了解某种工具。

它不讲过程，它假设读者已经了解某种开发过程。

它只是在读者已经了解方法、过程和工具的基础上，提醒读者在绘制 UML 图时需要注意的一些细节。

在这本类似掌上宝小册子中，Ambler 提出了 200 多条准则，帮助读者在画龙的同时，点上龙的眼睛。



《对象设计》，Rebecca Wirfs-Brock, Alan McKean 著，鲍志云 译

UMLChina 训练教材，人民邮电出版社即将发行中译本。以下为部分草稿。

第 1 章 设计概念

对于软件对象，Alan Kay 最喜欢的类比是生物系统。和细胞一样，软件对象并不知道别的对象内部是怎么工作的，但是它们互相沟通，并且协同执行复杂任务。与此相反，“浑然一体”的单体式软件则像机械钟那样包含无数的齿轮。每个齿轮都不具备智能，只和其他邻近的齿轮发生关系。这样的设计不仅仅是具有瑕疵，而且无可救药。Kay 说过，“当你制造机械钟的时候，最后复杂性总会达到某个地步，就像搭积木一样，再多垒一块整个结构就会倒塌。”

。。。（此处略）。。。

第 3 章 发现对象

Joseph Albers 可以让颜色闪耀或是黯然：“我把颜色看作是动态的……把两种颜色放在一起让我激动。它们一起呼吸，就像脉冲一样……我喜欢选一种很淡的颜色，通过让它和邻居共同作用变得丰富和美丽。我可以扼杀哪怕最耀眼的红色，只要把它放在紫罗兰色旁边。我可以让世界上最阴暗的灰色闪光，如果把它和黑色形成对比。”Albers 是 20 世纪最伟大的画家之一，是通过形态和颜色呈现视觉效果的大师。通过小心地把颜色、纹理和形状比邻放置，他可以让形象跃然纸上。Albert 将之称为“1+1=3”效果。好的设计要大于它各部分的综合。糟糕的设计则使得本该强调的部分模糊不清。胡乱使用粗线条和颜色，或者添加虽好看但没有价值的东西只会产出视觉垃圾，反而分散了人们本应集中于重要信息的注意力。在图形设计中，组合、形式和焦点就是全部！对象设计则面临相似的挑战。生动的抽象和构成良好、和整体结构和谐配合的对象可以改善设计；明显的不一致或者模糊的概念则会破坏设计。

你选择的抽象会对你的整体设计有重大影响。一开始，你的选择比较多。在你寻找候选对象的同时，你也在创建和发明。每个发明都是一次着色，会束缚你后续的选择。一开始，应该寻求重要的、生动的抽象，也即那些表现了领域概念、算法和软件机制的抽象。突出那些重要的抽象。如果你创造了太多的抽象，你的设计可能会变得过分复杂。如果没有足够的抽象，那么你最终会发现建立了一大堆平淡的缺乏生机的对象。

你的目标是以合意的方式来创建和组织对象。你的程序会分成一个个邻域，在邻域中一组对象为了一个目标共同工作。你的设计的形态取决于抽象的数量和质量，以及它们如何互补。组合、形式和焦点就是全部。

图形设计师通过层次和分离来强调重要的信息，使焦点集中于数据而不是其包含者；通过使用多种表示来消除歧义。

寻找方法

找出对象是个高度创造性的活动，但并不神秘。找出好的候选对象这个主题并没有获得很多关注。早期的对象设计书籍，包括 *Designing Object-Oriented Software* 都只是谈论了通过辨识写在设计规约中的内容来找出对象。现在看来，这种方法似乎有些天真。今天，我们并不提倡划出名词并简单地为现实世界中的事物建模。找出对象要比这复杂得多，它意味着辨认出作为应用程序领域一部分的抽象及其执行机制的抽象。在最好情况下，它们同现实世界中的事物的对应关系可能也不是很密切。即便是在为领域概念建模时，你也需要仔细地观察这些对象如何融入你的应用程序的总体设计。

良好的抽象和对它们如何互补的关注会对对象设计的质量有直接影响。本章讨论了如何在最早的对象设计阶段发现并组织软件对象。最终的目标是开发出解决问题的现实方案。但是，我们发现，这样的设计通常在美学上也是令人愉快的设计。

SAMPLE

虽然软件对象不会坐在那里等你去发现，但是你可以以具有一定系统性的方法去辨识它们。虽然可能有很多不同的因素驱动你的设计，但是有一些寻找对象的标准场所，你会发现很多灵感的源泉。你可以利用你对应用程序领域的知识，对所需应用程序机制的理解，从别人那里吸取到的教训，以及你过去的设计经验。

我们为发现和评估候选者提供的菜单包括如下步骤：

- 写一个简短的设计故事。在故事中，描述对你的应用程序重要的东西。
- 借助这个故事来辨认出定义你的应用程序的一些关键关注点的主要主题。
- 寻找围绕和支持每个主题的候选对象。利用现有的资源来获得灵感：系统行为的描述、构架、性能、结构。
- 检查这些候选者，看它们是否表示关键概念或表示了代表你的软件对外部世界的视角的事物
- 寻找表示其他结构、机制的候选者
- 命名、描述和刻画每个候选者。
- 组织你的候选者，寻找把你的应用环境划分成邻域（为共同问题而协作的一组对象）的自然方式。
- 检查它们是否适宜。检查它们是否表示了合理的抽象。
- 寻找包括每个候选者的理由。
- 若发现的过程变得缓慢，那么就去为职责和协作建模。

本章会覆盖所有这些步骤并描述更多细节。但是请注意，你并不总是需要在进入下一步之前完成前面的步骤。发现和创造的过程要比上面描述的更具灵活性。有时候你会同时进行几个步骤。有时候你会抛弃一些候选者并从头来过，如果它们看起来不符合形成中的设计。只要你以在设计故事中描述什么对你的应用程序的成功最重要作为开始，那么你就可以有组织地开始寻找支持这一核心的对象了。

在最初探索结束时，你会拥有不少仔细选择并检查过的候选者。随着设计继续进行下去，你会创造出更多的候选者。这些最初的候选者是你有意选出来支持系统的一些关键方面的。它们是设计其余部分的种子。寻找和创建第一批候选者是需要认真对待的。

一开始我们建议你寻找候选角色和对象。若你有了什么想法，想法不会消失。做出决定，如何来把它们实现为接口和类。

寻找对象和角色，然后寻找类

要寻找的第一批候选者应当表示重要的东西：概念、结构、机制。一般这类候选者是比较具有智能的，它们负责做事情。也可能它们知道事情，但是会基于知道的信息来执行动作。一开始，请进行具体的思考。当你看到更具体的对象并理解了它们之间的关系之后，抽象在稍后会随之而来。作为开始可以先辨别出具有明确角色的对象。然后再决定候选者应当知道和做些什么事情（它们的职责），它们同谁一起工作（它们的协作）。

然后，进行更抽象的思考。你可以去找出一些候选者共有的方面。把你的关注焦点从对对象以及它们各自的角色的思考转移到对象具有哪些共同之处。只有在你判断出不同的候选者都具有的共同职责后，你才能定义公共角色。我们一直都把这些对象和角色看作候选者，直到它们的价值被证实。只有在证实之后我们才决定如何把它们实现成类和接口。

SAMPLE

当你从候选者转移到类和接口，你就有了观点。你可以引入继承、抽象、接口和协作来构建划分良好的灵活的设计。你将规定抽象和具体的类以及接口。抽象类提供了职责的部分实现，并强迫子类实现特定的职责。具体类则提供了完整的实现。接口以方法签名的方式更精确地规定了职责，但是它们的实现却是开放的。任何类都可以实现一个接口，不管它在哪个类继承层次体系中的什么位置。

抽象类和具体类是我们用来说明实现的基石。说明接口是让它更具灵活性和可扩展性的一种方法。可复用的角色最好是用可被一个或多个类实现的接口来规定。

。。。。。。。。。。。（此处略）

。。。。。。。。。。。

搜索策略

一旦你找出了主要的主题，那么你可以把它们作为灵感的源泉。基于应用程序的本质和对其关键的事物来对可能需要的发明的类型进行推测。候选者通常会表示如下内容：

- 直接受应用程序影响或者和它关联的东西（其他软件、物理机器、硬件设备）
- 流经你的软件的信息
- 决策制定、控制和协作行为
- 对象的结构和组合
- 应用程序需要知道的现实世界中的事物的表示

我们从这些角度来指导我们的搜索。我们寻求创建的类型同角色构造型密切相关。

如果应用程序的核心任务最终是计算，那么试着用扮演服务提供者（演算、计算、变换、估算）角色的对象来构成应用程序。你很可能会创建表示算法或操作的对象，同时还有控制工作过程的对象。如果你的应用程序的主要角色是组合信息并把信息从一处搬到另一处，那么寻找可以为信息建模的对象，以及协调这些对象的搬移的对象，把它们作为候选者。如果你的应用程序同别的系统相连，那么创建外部衔接者用于形成这些连接。大多数设计都需要控制或协调其他对象工作的对象。根据控制的复杂性，这一设计决策可能很重要，也可能不重要。如果你的应用程序需要在相关对象之间建立和组织连接，那么可能需要找出组织者。在你寻找的对象类型和你的软件执行的工作的本质之间有相对直接的联系。

评估表示外部事物的潜在候选者的最好方式是转换视角。进入你的软件，观察外部世界。从你的应用程序的角度去看。问自己，你需要对你的用户知道些什么，对你连接的系统知道些什么，你会影响到哪些东西。

在你寻找候选者的时候，可以问的一个问题是，“我们的软件对相联系的外部和虚拟世界中的东西需要知道多少？”在边界，把同其他系统的连接建模成衔接者对象。你可以在你的设计中包含表示这些其他软件系统的对象。这些服务提供者会被应用程序的其他部分调用到。但是你何时需要为计算机之外的东西建模？比如，你的软件的用户？如果只有他们的动作有意义，他们是谁不重要，那么就不要在设计中包括他们。用户的动作可以由用户界面对象来传递（这些对象负责翻译用户请求和信息并传递给系统的其他部分）。不需要知道按应用程序中的按钮的人是谁。但如果对不同的用户你的软件会有不同的行为，那么把用户的一些表现作为候选者。如果你的软件需要基于跟谁在交互而做出决策，那么需要对用户有所了解（也需要表示这一了解的对象）。例如，如果不同的用户对帐户的访问权限不同，或者继续游戏的能力需要知道游戏者是谁，那么设计中应该包含对这些用户的一些表示。

SAMPLE

表3.1和3.2概括了在这两个应用程序中我们的搜索策略。虽然我们考虑了每个角度，但通常对每个特定的主题只有一两个角度有意义。如果我们发现某个特定的角度不会带来见解，那么我们就忽略它继续前进。对每个主题，我们简要地概括了可以带来见解的角度，以及我们所寻找的候选者的类型。

主题	视角	支持……的候选者
在线银行功能	我们的系统执行的工作	执行金融交易、查询账户
	我们的软件影响的东西	账户、后端银行系统交易
	流经我们软件的信息	关于交易的信息、账户余额、交易账户、账户历史、支付
	现实世界中事物的表现	客户、用户、它们访问的账户
灵活配置行为	我们的软件影响的东西	后端系统的公共接口
	流经我们软件的信息	Web 页面标语、文字、消息和账户格式的可配置显示
分享稀缺资源	对象结构和分组	管理到后端系统的有限连接和我们的在线银行应用程序数据库
账户的不同视图和访问	我们的系统执行的工作	限制用户的视图和执行修改账户余额的银行交易的能力
	决策制定、协作和控制	除非用户具有特定权限，否则禁止对账户的访问

表 3.1 一开始对在线银行应用程序候选者的搜索是基于对 4 个主题的探索

主题	视角	支持……的候选者
游戏建模	我们的系统执行的工作	把游戏者赋给游戏，裁判，保存和继续暂停的游戏，进行游戏，判断走棋的合法性，判断走棋的结果，显示每个游戏者的棋盘的状态
	流经我们软件的信息	关于走棋和游戏者请求的信息
	现实世界中事物的表现	游戏者和他们的动作
	对象结构和分组	管理保存的游戏，不同的游戏，棋子，棋子在棋盘上的位置
计算机玩游戏	我们的系统执行的工作	跟用户玩游戏
	决策制定、控	基于当前对游戏的视图（应该和人类游戏者看到的受到同样限制）来判

SAMPLE

	制和协作	断如何合理地走棋
在分布的组件间分配职责	决策制定、控制和协作	把游戏者的请求发给裁判，在游戏者之间发送游戏状态，检测游戏者是否依然在线
	流经我们软件的信息	游戏者走的棋，棋盘的更新，游戏状态

表 3.2 一开始对 Kriegspiel 应用程序候选者的搜索是基于对游戏建模、计算机智能下棋，分布式游戏这几个主题的探索

我们将找出支持相关视角的候选者。有时候，从我们的简单描述中，一眼就能看出候选者；基于我们需要有支持游戏对“游戏者及其动作”的现实世界视图的候选者，“游戏者”和“游戏者动作”会是好的候选者吗？很可能是。在其他的时候，我们必须思索我们的软件具体是如何工作的，这样才能找到候选者。会不会有一个“银行服务连接管理器”来管理“银行服务连接”；或者会有“数据库连接管理器”来管理“数据库连接”这样的稀缺资源？不同的主题和视角常常反复告诉我们需要某种候选者。这是好事。这样我们就可以相信，某个特定的候选者对我们的应用程序确实是必要的。在其他的时候，可能想法不会那么快出现，我们必须更深入地思索来找到潜在的候选者。

我们不会在第 1 遍就找到所有的关键候选者。我们起初对候选者的想法也不会固定不变。随着我们对候选者给出进一步的定义，我们的想法会改变。我们一开始找出的候选者是我们的设计的种子，所以给每个候选者一个强有力的名字来暗示它的角色和目标是非常重要的。所以，在继续寻找候选者之前，我们先来探讨一下怎样才能找出有用的名字。

名字包含了什么

好的名字可以增进设计的活力和动力。你可以依赖一个好名字。当说出软件对象的名字的时候，设计者可以推断出关于对象角色和职责的一些东西。所以有经验的对象设计者会说“认真地选择名字”。好名字可以建立起到过去经验和日常行为的联系。不管我们是否喜欢，名字总会有含义。我们的大脑结构会在我们已经知道的事物中寻找联系。所以，给对象一个好名字的关键是让它的名字跟你已经知道的东西发生联系，而且要有助于让你知道它应当做什么事情。大多数名字都应符合名字的系统。不同的命名模式会同时存在，哪怕在一个应用程序中也是如此。没有单一的通用命名系统。

“……思维和单词之间的关系不是事物而是过程……思维不只是用单词表达的，而是因单词而存在。所有的思维都趋向于把一些东西和另一些东西关联起来，建立事物之间的联系。所有的思维都会移动，成长，发展，实现功能，解决问题。”——Lev Vygotsky

修饰一般的名字。给是“某种一般概念的特殊情况”的事物命名的一种模式是在一般名字前加上对特殊情况的描述。

> 日历表示某个特定地方的日期和时间系统。公历扩展了日历类。根据传统，我们还可以创建罗马公历或者农历类。其他熟悉这个模式的人就可以推测出它们的实现同公历类有什么不同。

在名字中只包含最能给人启示、最显著的事实。所有描述模式的缺点是名字会变得很长。不要把对象所有不同的特性都放在名字中，把可能会改变或者别的对象不应该知道的细节隐藏起来。

> 人们真的有必要关心他们用的是“精确到两毫秒的毫秒计时器”吗？还是仅仅告诉它们“计时器”就足够了？详细的设计决定不应该表露出来，除非它们不太可能改变而且对于对象

SAMPLE

的使用者有已知的影响。暴露实现细节会使得它们难以改变。.

考虑一下 *Design Patterns* 一书中描述的 Singleton 模式。这个模式确保一个类只有一个实例，只能通过一个全局访问点访问。我们可以把所有符合这个模式的概念都命名为“某某单件”。但是，根据我们的原则，我们建议不要这样命名。因为单件这一分别对类的实现者更重要，对使用者则没那么重要。起的名字应当对将要使用候选者的人有意义，而不是对实现它的人。如果使用你的候选者的人必须知道它的实现细节，那么你很可能没有去做更好的抽象工作。这一规则的一个可能例外是，若使用者非常有必要知道一个类是单件，那么可以把 Singleton 加在类名后面。

给服务提供者起“worker”这样的名字。另一个英语命名惯例是在工作名称后面加“er”。服务提供者对象是“worker”、“doer”、“mover”、“shaker”。如果你可以找出一个“worker”这样的名字，那么它对于对象的角色是个强有力的暗示。

很多 Java 服务提供者都遵循这个“worker”命名模式。一些例子是 StringTokenizer、SystemClassLoader、AppletViewer。

如果 worker 类型的名字听起来不太对，那么另一个惯例是在名字后面加 Service。在 CORBA 框架中，这是常见的惯例。比如，TransactionService、NamingService 等等。

如果一个对象的名字意味着宽泛的责任，寻找可以补充它的其它对象。有时候候选者代表了宽泛的关注点，有时候它的焦点更集中。如果你遇到一个意味着很多职责的名字，那么检查一下你有没有给候选者起错误的名字。有可能你的候选者应该有更窄的关注焦点。也可能这意味着你发现了一个宽泛的概念，需要扩展它。寻找可以细化或补充宽泛的名字的对象可能可以让你找到一组相关的对象，以及一组相关的候选者。很多时候我们在设计中既需要特定的概念也需要一般的概念。如果事物具有更一般化的名字，那么它可以定义每个特定候选者所共有的职责。

名为“会计服务”的对象很可能会执行一些会计功能。“会计服务”这个名字并不明确。我们无法仅通过看名字就推导出关于它执行的会计服务的类型的信息。要么它负责应用程序中所有类型的会计功能，要么它代表了一个抽象，其他具体的会计服务对象会基于它来扩展。如果是这样，我们我们会期待其他的候选者，每个都有一个更明确的名字，比如“余额查询服务”、“支付服务”或“转账服务”。这些名字更明确的候选者会支持特定的会计活动。

用更特定的候选者来突出一个一般的概念。如果你能想出至少 3 种特殊情况，那么把一般概念和特定概念都保留下来。如果你稍后发现那些更特定候选者并没有共享任何职责，那么你可以随时抛弃那个更抽象的概念。但是，如果你只是给一个候选者起了一个太宽泛的名字，那么一定要为它换个名字。

如果你的候选者可以表示很多其他事物的历史记录，那么最好给它一个更一般的名字“历史”。如果你的意图是为交易历史建模，那么把你的候选者命名为“交易历史”。你可以决定要具体到何种程度。

通过观察两种具体情况来找出抽象可能也是可行的，但是比较和对照 3 到 4 种情况更好。可以比较和对照的密切相关的概念越多，越容易找出它们的共同部分。

命名也有艺术：选择能传达足够含义的名字，但也不要施加太多限制。留下可能性，可以给候选者刚好它能够处理的职责，可以稍加修改就把它用在其它不同的场合。如果一个候选者在几个不同的场合

SAMPLE

都能使用，那么显然这是更强有力的设计。另一种办法是在每种不同的情况下都用不同类型的对象来处理，这是可行的，但远没有前者优雅。

选择不会约束行为的名字。不要选择意味着太窄的动作余地的名字，这样会限制候选者的潜能。如果可以选择，那么挑选允许对象承担更多职责的名字。

思维塑造单词，单词影响思维。名字会微妙地改变我们对候选者的行为的期待。

考虑一个候选者的两种选择：“账户”或者“账户记录”。每种都可以作为保持客户信息的对象的名字。根据常识我们知道，记录的一个含义是“写下来的信息或事实”。如果我们让角色符合对名字的习惯用法，那么“账户记录”除了具有保存信息的职责之外不太可能具有其它职责。而“账户”则为更多的职责留下了可能性。账户对象可以就其表示的信息作出决策，它听起来比账户记录更具活力，更加主动。

选择在候选者生命周期都有效的名字。听说一个 90 岁的老人叫“Junior”是很好笑的。只让对象的名字体现它最初的职责而忽略它稍后可能会做的事情是个错误。如果对象的工作改变了，那么不要满足于停留在你给它的第一个名字。

定义了初始化应用程序并接着监控来自外部的关机或重新初始化事件的职责的对象取名为“应用程序协调者”要比“应用程序初始化者”好，因为“应用程序初始化者”并没有暗示对象会在应用程序启动并开始运行之后继续负有职责。而“应用程序协调者”则比较好，因为它具有更一般的含义，覆盖了更多的职责。

选择符合你当前设计的背景的名字。当你选择名字时，请选择符合你当前设计的背景的名字。否则的话，你的候选者的名字可能听起来会比较奇怪。在会计应用程序中听起来很合理的名字可能在工程应用程序中听起来就很不和谐。

一位经验丰富的 Smalltalk 开发者在开始用 Java 工作时努力地避免偏见。虽然他预期到 Java 的类可能会有完全不同的职责，但是当他发现 Java 的词典类是抽象类的时候还是很吃惊。在 Smalltalk 中，词典对象会被频繁地创建和使用。

如果你过去的偏好与你现在的工作不符，那么不要把偏好变成偏见。

不要一名两用。在日常口语中，单词常常具有多个含义。但是对象名字应该只有一个含义。若在同一个应用程序中有两种不同类型的账户对象共存，而且它们的角色有很大不同，这并不是什么好事。有些面向对象编程语言允许你给不同的类起相同的名字，但是当你在代码中引用特定的类时，你必须独一无二地标识出名字。比如在 Java 中，不同包里的类可以具有相同的名字。为了独一无二地指明一个特定的类，必须在类明前加上定义这个类的包的名称。

Java 设计者可以用相同的名字来定义类，只要它们位于不同的包中。只有当一个包被设计为用来代替另一个包时，你才应当这样做。

同时存在于同一个应用程序中的事物应当具有不同的名字。不要一名两用。程序员只看到一个背景：运行的应用程序。用来解释名字的也只是这一个背景。请不要增加另一处混淆之源，因为程序员们要考虑的事情已经够多的了。编译器很擅长自动在名字前加上正确的修饰，但人类不擅长做这样的事。

SAMPLE

通过增加形容词来消除名字的冲突。有时候最好的名字已经被用过了，但你还是要为候选者起一个名字。那么你可以在名字前加上一个描述性的词组，这样你就可以获得一个独一无二的名字了。

> 如果一个候选者的理想名字和 Java 的“属性”类冲突，那么不妨给它起名为“事务属性”。

告诫：如果你的候选者具有很不相同的含义，那么不要用和已有的名字类似的名字。遵照惯例。若取了类似的名字，那么熟悉已经存在的名字的设计者会期待你的候选者也能在相应场合执行类似的工作。

通过选择具有类似含义的名字来消除冲突。有时候，最好还是找一个同义词。每个同义词之间多少还是会有些差别，所以可能找到一个满意的名字不是那么容易的。

> Property（Java 类库中定义的类）的同义词有：characteristic、attribute、quality、feature、trait。或许“attribute”或者“feature”可以适用，但“characteristic”看起来就比较沉闷，“quality”则有些矫揉。

选择广为人知的名字。名字不应该太简洁，不要为了节省键盘的敲击而抄近路或者把含义浓缩。如果你希望别人不需要研究对象如何工作就可以大体知道对象的角色，那么就给它一个描述性的名字吧。描述性的名字未必是很长的。

> “Acct”太令人费解，还是“Account”比较好。

如果你的问题领域中有大家都知道并且理解的缩写，比如银行领域中的 USD，技术领域中的 Mhz 或 Gbyte，那么把这些包含在候选者名字中是合理的。

描述候选者

我们通过名字和角色符合得有多好以及角色和场合符合得有多好来评判对象。对候选者的角色划分构造型为快速建立对于对象期望用法的认知提供了方便的手段。当你找到一个候选者，给它起个名字，然后把它划入一个或多个构造型。候选者可能会是服务提供者、控制者、协调者、组织者、信息持有者或者衔接者。你可能还需要更细致地划分 3 种不同的衔接者：用户衔接者（同用户打交道的对象）、外部衔接者（介于你的应用程序和其他应用程序之间的对象）和系统衔接者（衔接应用程序的不同部分的对象）。

为了更明白地表述意图，你可以区分对象是否被设计成被动的，只保存相关信息（信息持有者），还是会在管理和维护信息时扮演更主动的角色（信息提供者）。如果这些更细的区别看起来太不明显了，那么就不要为它们费神。不要担心能否给对象“正确”的构造型。如果你的应用程序中有很多对象看齐来不符合这些预先定义好的构造型，那么就想出你自己的构造型。构造型是为了帮助你思考候选者的，而不是为了阻碍你前进。

如果你对候选者将扮演的角色还不确定，那么就推测一下。用它的构造型作为创建简单定义的指导。在定义中，解释你的候选者可能会做什么，并列出它与别人不同的特性。在 CRC 卡片没有横线的一面写下这个简单的定义（参见图 3.1）。

SAMPLE

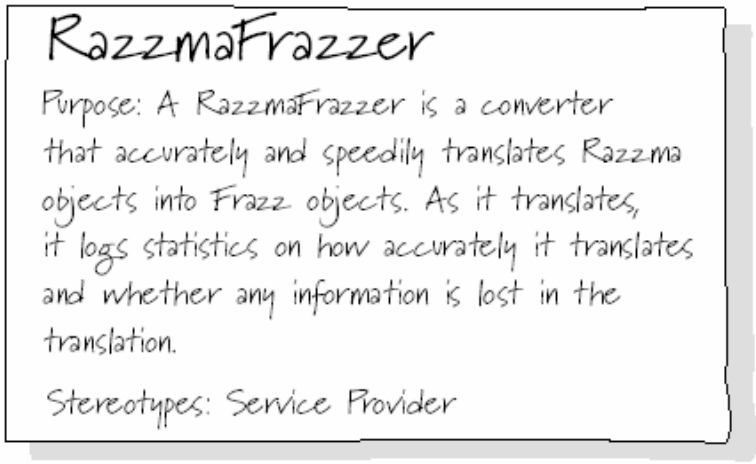


图 3.1 CRC 卡片没有横线的一面用来描述对象的目的和构造型。在这个例子中，RazzmaFrazzer 只有一个构造型

一般地说，当描述对象时，可以使用下面的模式：

> 一个对象是一类东西，它知道一些事情，做一些事情。简洁地说这些事情是什么。然后提到一个或多个关于这个对象的有意义的事实，比如它做什么或知道什么或与谁协作的一个细节。这只是为了提供更多的背景。

服务提供者、控制者和协调者是根据它们做的事情来区分的。下面是描述这些构造型的简单方式：

> 服务提供者（或者控制者、协调者）是一类东西，负责做某种工作。简洁地描述这种工作。然后提到一些关于它执行的工作或者它同谁交互的有意义或重要的事实。

如果你一个人工作，那么你可能不会觉得迫切需要把这些想法写下来。毕竟，你知道你自己是什么意思。但即便如此，草草地记下想法的概要也是有帮助的。你不希望下周五就忘记那个可恨的 RazzmaFrazzer 为何重要。类似地，若你在一个团队中工作，那么别人很可能不知道一个候选者有什么重要性，除非你告诉他们。你能写下的关于一个候选者目的以及你期望它做什么的任何描述都是有帮助的。

考虑一下这个定义：

> 编译器是“把源代码翻译成机器语言的程序”。

把它和这个略微简单点的定义对比：

> “编译器把源代码翻译成机器语言。”

这两个定义几乎一模一样。第一个定义多出来的是“编译器是一个软件程序”。这在软件设计者看来纯属多余。我们都知道编译器是程序。但是第一个定义提供了刚好够的背景，这样其他跟我们行业背

SAMPLE

景不同的人也能把编译器和其他的计算机程序联系起来。只要你能够把某样东西和一个广为人知的概念（比如计算机程序）联系起来，那么就有助于所有人更清楚地了解它的含义。

如果你和其他设计者把设计当作生活的一部分，那么可能很多东西不会说出来或者写下来。你们彼此理解，因为你们的以类似的方式思考。但是，如果你们对候选者是什么会有问题或分期，那么可能是你们做了不同的假设。为了让意图更明了，应该增加足够的细节以消除疑问，然后讨论一下谁的主意更好。既要描述候选者是什么，也要描述它不是什么。把它和类似的东西联系起来。

我们通过举出候选者将被如何使用的例子以及给出关于它的职责的一般讨论来给出更多细节。当你描述可以被几个不同的对象扮演并扩展的角色时，这尤其重要。

金融交易表示了一个账户交易，由我们的在线银行应用程序执行。成功的交易会导致对客户的帐户的更更新。特定的金融交易会和银行系统通信以执行实际工作。例子包括转账交易、支付交易。

如果常见的含义适合于候选者，那么用它来形成基本定义。不要为了发明而发明术语。如果有多种可选的定义，那么选择最接近于你的应用程序的主题的定义。如果标准的定义合适，那么就以标准定义作为开始。然后再描述什么使得你的对象在应用程序中与众不同。（以下略）

（经人民邮电出版社允许刊登）

SAMPLE

设计模式精解

Design Patterns Explained

看不懂《设计模式》？
不用惭愧，别人也一样苦恼。
先把它供起来吧，看这本！

UMLChina 训练
辅助教材

dearbook.com

Allan Kelly & James R. Trott 著
熊节 译

清华大学出版社

UMLChina 训练辅助教材

UP 实作的一些常见问题（中）

潘加宇 文

讨论

分析&设计

5. 受数据库习惯影响的类图

数据库建模技术可能是许多开发团队仅有的一种建模技术，群众基础最好。数据建模人员转向面向对象建模时，有时会把数据库的习惯带到类图中：

无必要地拆分多对多的关联为两个一对多关联

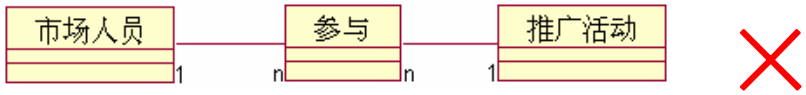


图 1 无必要的中间类

如果系统只需要记住哪些市场人员参与了哪些推广活动，并不需要记住何时参与，参与的细节等，就没有必要在中间再放一个类，直接画成关联即可：

图 2 画成关联即可

当然，这个类图映射到数据库时，会自动生成第三个表来存放两个表的外键。

属性里带上另一个类的“编号”

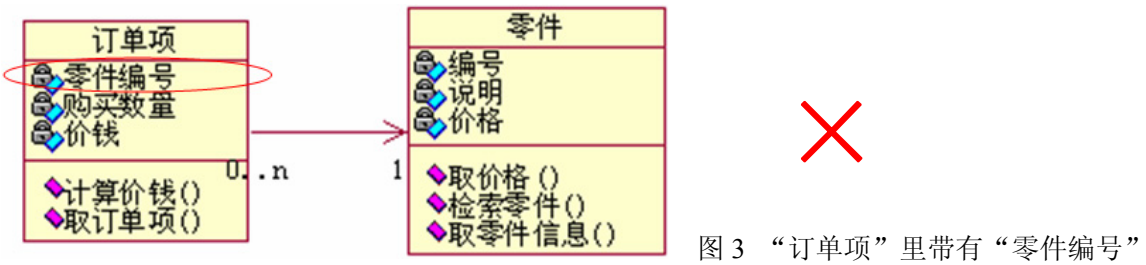


图 3 “订单项”里带有“零件编号”

这种习惯比较隐蔽。“订单项”里带有“零件编号”，看起来很符合现实——订单打印出来，每一个订单项上会有零件编号，但“零件编号”并不是“订单项”的属性，而是零件的属性。“订单项”知道的是“零件”。映射到关系数据库时，确实需要把“零件”表的主键（可能不是“编号”而是系统生成的代理主键）放在“订单项”表中建立外键连接，但类图并不需要理会这些。

6. 滥用整体－部分关系

UML 中的关联有三种表现形式：连接、聚合、组合。建立关联的时候，一些开发人员喜欢使用聚合/组合关系，例如：

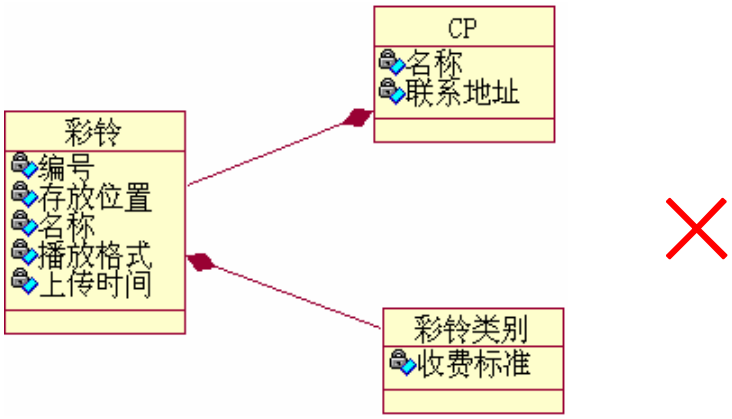


图 4 滥用整体－部分关系

聚合/组合表达的是一种整体－部分关系，通过建立这种关系，使整体把部分封起来，这往往暗示着在职责分配上，外界对象应该先把消息发给整体对象，再由整体对象分解、分配给所包含的部分对象。

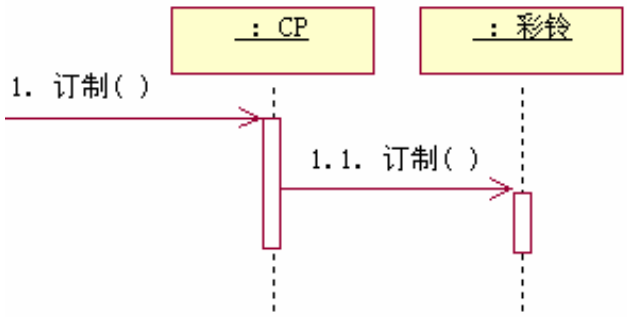


图 5 职责分配时，“整体”要把“部分”挡住

如果“彩铃”是涉及许多业务的核心类，这种职责分配是不合理的。建立成以下连接应该更加合理：

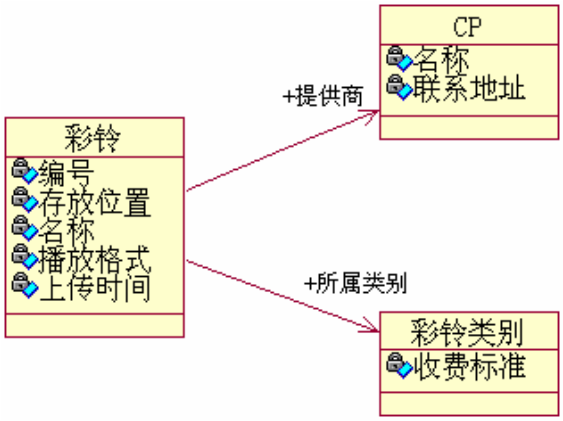


图 6 连接关系更加合理

有时候建模者的想法是这样的：本来想把它建立为连接，但关联名字或者角色名字起什么好呢，算了，懒得想，建立成“有一个”还省事！这样，这些菱形就被滥用了，虽然说连接、聚合、组合更多的区别在于概念上，在编码实现上基本没有区别。

7. 摆设的顺序图

顺序图经常出现的问题：

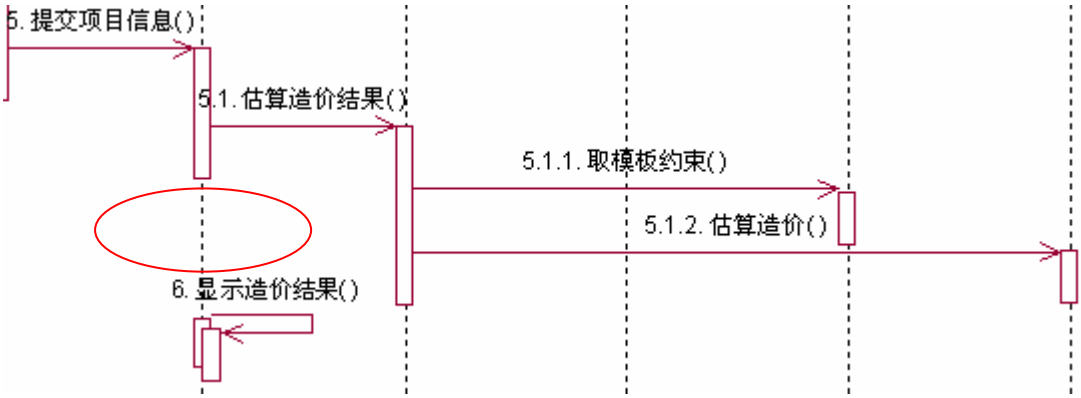


图 7 中间断开了

上图的中间断开是没有来由的。对象不会无缘无故停下，然后在没有激活的情况下又有了动作。

另一种常见的情况是往返消息：

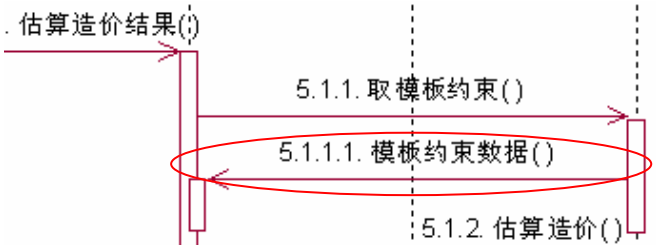


图 8 又有一个“模板约束数据”指回来

一般的应用层面的软件，消息大多为简单的消息，消息本身就已经包含了去和回的信息，并不需要添加另外的元素来说明。而且指回来还暗示了一个新的操作，这是错误的。如果要特别强调说明某种返回结果，可以用虚线的“return”。可能新接触到顺序图的开发人员觉得不太习惯，其实只要了解顺序图的道理就好理解了。

顺序图是一种职责分配的方法，把用例文档中界定的系统责任分配到系统的各个类上。顺序图上消息的传入，可以映射类对象所具有的操作，也就是职责；消息的传出则是为了完成某种职责需要其他对象帮忙的协作，往往暗示了其中协作的代码结构。

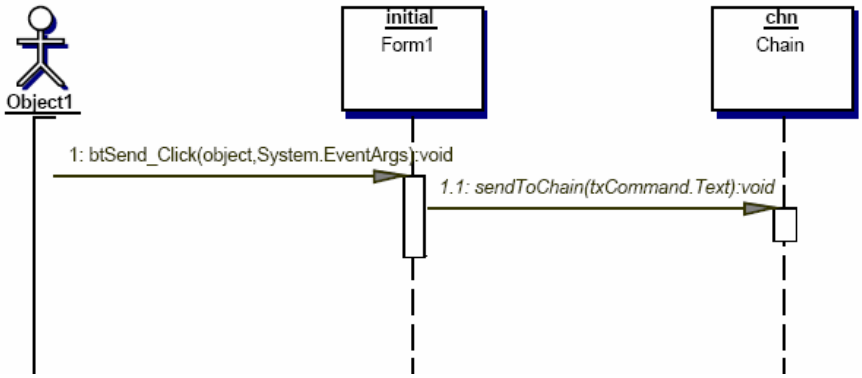


图 9 顺序图上的职责和协作

有箭头指进来，表示 Form1 类需要有一个 btSend_Click 操作；根据指出去的箭头，可以看出这个操作的代码可能是这样：

```
private void btSend_Click(object sender, System.EventArgs e)
{
    chn.sendToChain (txCommand.Text );
}
```

这样，在绘制顺序图的过程中，就可以不断为类分配职责。如果没有理解顺序图的作用，很可能画出来的顺序图只是一种摆设。

8. 摆设的状态图

如果问你，“订单”的状态有哪几个？你可能会说就是很熟悉的那几个嘛，“空订单”、“未结账”、“已结账”、“已发货”…。那么人有几种状态呢？只是分生和死，还是需要再分老中青？我今年 30 岁，明年 31 岁，我的状态变了吗？

状态并不是指类的属性里带有一个“状态”属性（这样的属性往往是冗余的），而是指在系统中表现出相同行为的属性值组合，“有哪些状态”和系统上下文是密切相关的。像下图：

一个售楼系统，有个“房源”类的状态图如下：

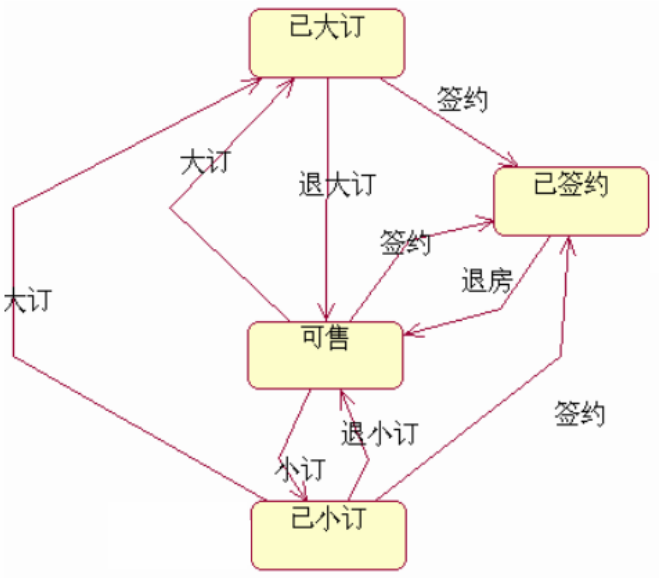


图 9 漂亮的状态图

看起来是合理的，但是，如果有一个“下小订”用例的顺序图是这样画的：

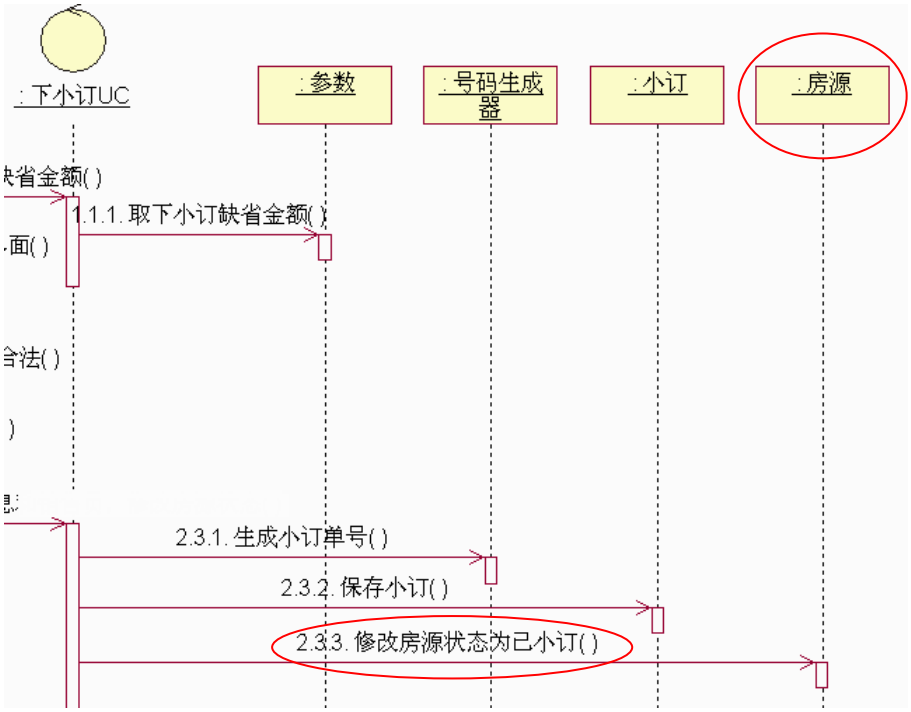


图 9 对不上号的顺序图

“房源”对象上发生的行为只是“修改房源状态为已小订”，并不是“小订”，“房源”如何知道“小订”事件的发生呢？上面的状态图就成了一纸空文。要么顺序图不合理，要么状态图不合理。显然，根据顺序图把状态图上的事件改为“修改状态”看起来很滑稽，所以需要修改的是顺序图上的职责分配：

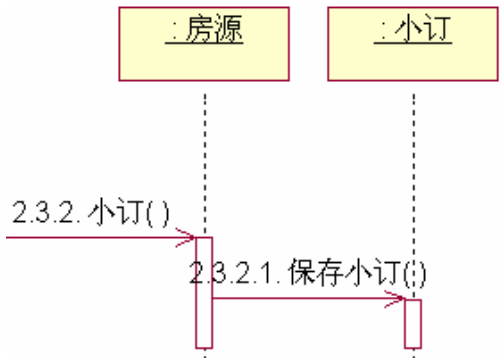


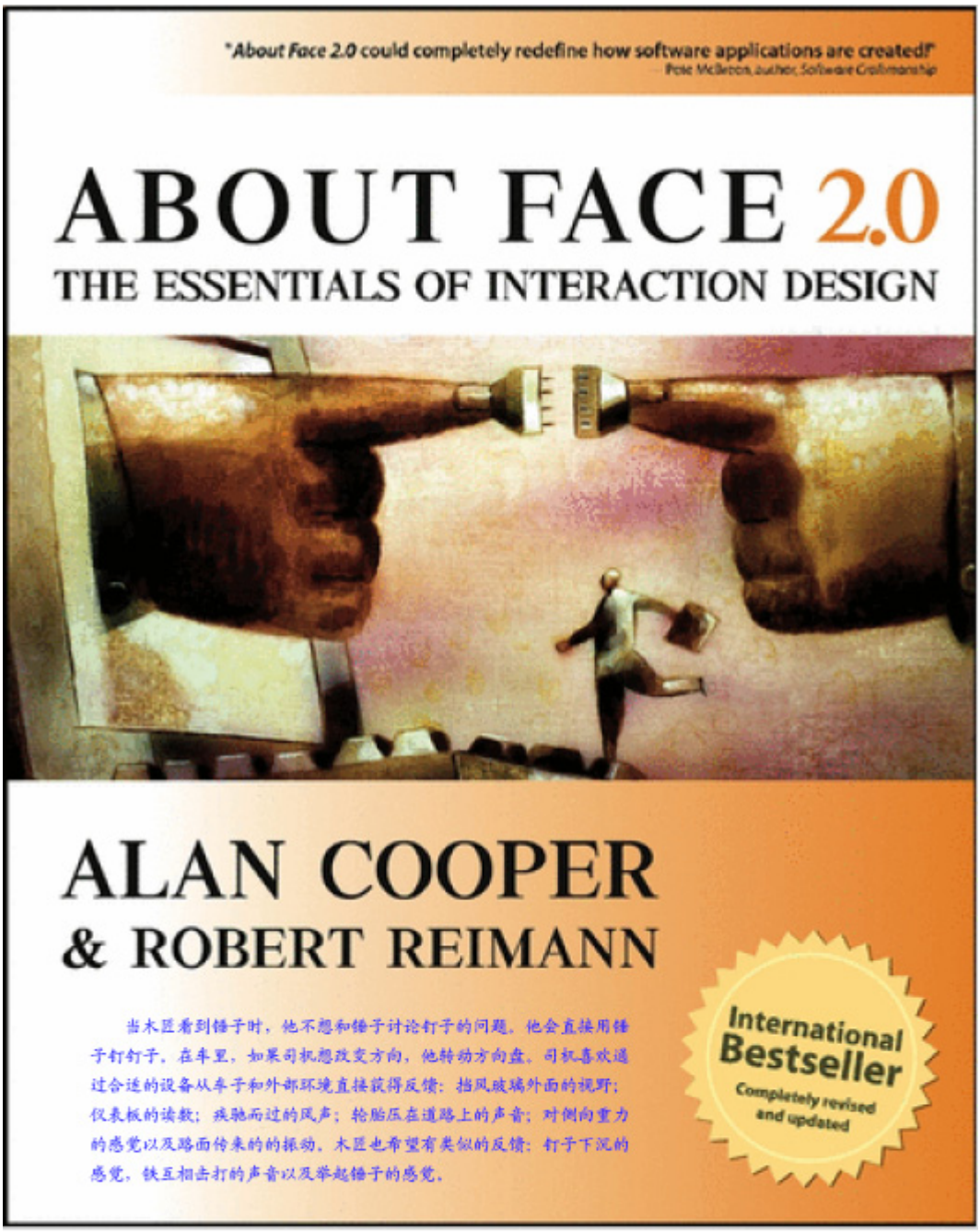
图 10 根据状态图调整顺序图

（待续）

（本文首发于《程序员》2005 年 1 期）

《About Face 2.0》精彩节选

讨论



UMLChina 辅助教材。中译本即将由电子工业出版社出版，詹剑峰 译

审稿：国内首个交互设计网站 De Dream'

第一章 目标导向设计

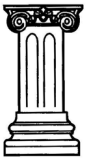
产品成功的关键是目标，而不是特性

程序员和工程师——对技术感兴趣的一类人——有着从功能和特性的角度考虑产品的强烈倾向。这很自然——因为这正是程序员创建软件的方式：一个功能接着一个功能。问题在于这不是用户想使用产品的方式。

确定某个功能是否应该包括在产品中不应该取决于它的实现技术。背后的决定因素永远不能只是我们拥有完成这项功能的技术能力。决定性因素应该是这一功能是否直接或者间接地帮助实现用户目标，与此同时仍然满足商业的需要。

即使在产品开发周期中的压力和混乱下，成功的交互设计师也应该对用户的目标高度敏感。虽然我们在本书中讨论了其他很多技术和工具，但我们始终回到用户目标。它是交互设计实践的基础。

目标导向的过程，还有它在设计决策时清晰的理论基础，能更容易地说服工程师，能让市场和管理部门参与进来，能确保正在考虑的设计不是凭空猜测，或者仅仅是团队成员个人偏好的反映。



公理　交互设计不是凭空猜测。

目标导向设计也是回答在定义和设计数字产品过程中出现的最重要问题的强有力工具：

- 如何发现用户？
- 如何知道哪些是用户试图完成的事？
- 产品应该如何工作？
- 产品应该采取什么形式？
- 用户将如何与产品交互？
- 如何能最有效地组织产品的功能？
- 产品以何种方式面向首次使用的用户？
- 产品如何体现在技术上容易理解和容易控制？

- 产品如何处理用户问题？
- 产品如何帮助比较生疏的用户变得更熟练？
- 产品如何为专家级用户提供足够的深度？

在这本书的余下部分，我们将帮助您回答这些问题。您会得到您所需要的工具来识别产品的关键用户，理解他们及他们的目标，以及通过使用目标导向设计将这些理解转变为成功的设计方案。

第 2 章 实现模型和心智模型

计算机工业界经常使用计算机文化（computer literacy）这个术语。权威人士经常谈论一些人具备这种文化，而一些人则没有；具有这种文化的人又如何会在信息经济中取得成功，而缺乏计算机文化的人则不可避免地落入社会经济学的断层里。然而所谓的计算机文化只不过是一种委婉的说法（euphemism）——让用户极力去理解计算机奇怪的逻辑，而不是让基于软件实现的产品灵活地适应用户的思考方式。

在本章中，我们将讨论缺乏对用户和他们对待数字产品方式的了解如何加速了计算机文化鸿沟(divide)的产生，而符合用户思考和工作方式的软件又将如何帮助解决这些问题。

实现模型（Implementation Model）

任何机器都有它达成目标的机制。例如，电影放映机使用复杂的移动图片序列来创建这种动态的感觉。它在一个瞬间让明亮的光线透过半透明的微缩图象，然后在它移向另外一副微缩图像的瞬间挡住光线，接着在下一个一瞬间再次投射光线。电影放映机每秒二十四次放映新的图像，重复这个过程。基于软件实现的产品并没有这样的机制，取而代之，它们使用算法和相互通信的代码模块。这种有关机器和程序如何实际工作的表达被 Dnald Norman（1989）和其他人称为**系统模型**（System model）。作者更愿意使用**实现模型**（implementation model）这个术语，因为它描述了程序在代码中实现的细节。

用户心智模型（user mental model）

从电影观众的角度来说，在观看一部引人如胜的电影时，很容易忘记影片齿孔间的细微差别和瞬间的光线打断。很多看电影的人，实际上根本不知道放映机如何工作，或是它工作的方式与电视有什么不同。在观众的想像里，放映机只不过是发射出在大屏幕上移动的图片而已。这就称为用户的**心智模型**，或者**概念模型**。

人们并不需要知道复杂产品的实际工作细节来掌握它的使用方法，为了便于使用，人们在认知上创建了一种简洁的解释方式，这种方式对他们与产品的交互来说已经足够了，但并不一定能够反映产品实际的内部工作机制。比如，很多人想像，当他们将真空吸尘器和搅拌机接到墙上的电源插座时，电流会像水一样通过黑色的小电线管从墙里流向电器。这种心智模型很适合于家用电器。但实际上家用电器的实现模型并不涉及什么液体类的东西在电线中流动，而是电压每秒钟反转 120 次。尽管电力公司需要知道这些细节，但这些细节与用户无关。

然而，在数字世界里，用户的心智模型和实现模型通常差别很大。我们经常忽略像“手机的工作方式和固定电话不同”这样的细节。相反，它实际上是一个在两分钟的通话过程中，要在半打不同的基站天线中交换连接的无线电接收器。了解这些细节并不会有助于我们对它的使用。

对于软件应用来说，实现模型和心智模型之间的差异非常彻底。在这种情况下，复杂的实现使得用户不可能看到用户行为和程序反应之间存在的机械关系。当用户使用计算机编辑数字声音或者创造视频特效，如变形时，无法在机械世界中找到类似的例子，所以我们的心智模型有必要和实现模型不同。即使这种关系有可能是可见的，但它们可能对大多数人来说还是无法理解。

表现模型

软件显示给外界的行为外表是由程序员或者设计师创造的。这种表达并不一定精确地描述了软件实际上在计算机里如何工作，虽然很不幸的是它经常如此。与其他媒介相比，这种独立于计算机的真实行为来表达其功能的能力，在软件世界里更加明显——它允许聪明的设计师隐藏软件如何真正完成工作的细节。在数字化世界里，“实现了什么”和“提供了什么”之间缺乏的联系带来了数字化世界里的第三种模型，设计师的表达模型——设计师选择如何将程序的功能展现给用户的方式。Donald Norman（1989）简单地称之为设计师模型（designer's model）。

在软件世界里，程序的表现模型可以（也常常应该）与程序的工作结构迥然不同。例如，操作系统能够使网络文件服务器看起来像本地磁盘一样。这种模型并没有表现出物理磁盘可能在数里之遥的事实。在机械世界里，表现模型的概念并不存在类似事物。这三个模型的关系见图 2-1。

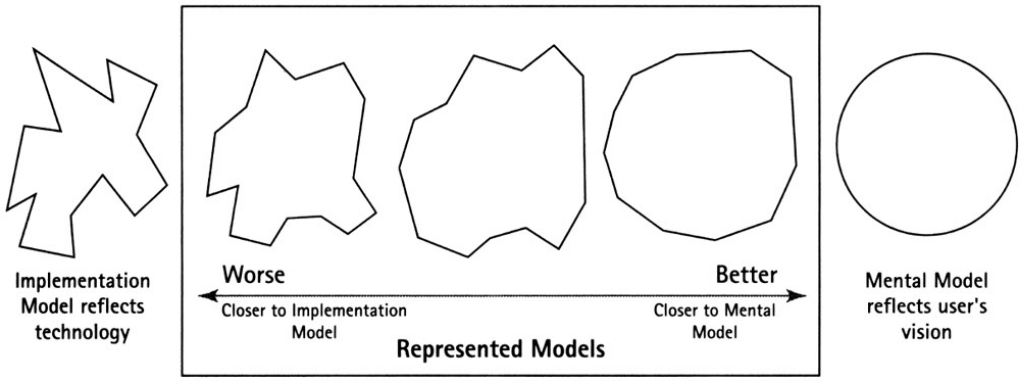
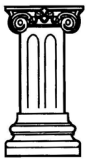


图 2-1　工程师开发软件的方式通常是给定的，并且常常受制于技术和商业约束。有关软件实际上如何工作的模型称为实现模型。用户与软件交互的心智模型是用户理解他们所需要进行的工作，和程序如何帮助他们完成工作的过程。这种模型基于他们自己如何完成任务，还有计算机如何工作的想法。设计师选择如何将软件工作机制表现给用户的方式称为表现模型。表现模型和其他两个模型不同，它是设计师能够极强地对软件进行控制的体现。设计师一个很重要的目标是使表现模型和用户的心智模型尽可能彼此匹配，因此设计师能否详细地理解目标用户所想到的软件使用方式非常关键。

表现模型离用户心智模型越近，用户就会发现程序越容易使用和理解。一般来说，假如（通常也如此）用户有关任务的心智模型不同于软件的实现模型，向用户提供过分接近实现模型的表现模型会严重地降低用户学习和使用程序的能力。

我们倾向于形成比现实更简单的心智模型。如果创造了比实际实现模型更简单的表现模型，我们就能帮助用户更好地理解。踩下汽车的刹车踏板，可能让你想到按下控制杆，摩擦车轮来降低车速。而实际的机制包括水压柱，管道和金属垫板挤压一个穿孔的圆板。为了简化事实，我们创造了更有效但不精确的心智模型。对于软件来说，我们想象：当点击滚动条时，电子表单软件将新的单元格显现在视图里。实际上没有单元表格，只不过存在一个紧凑的数据结构，以及它们之间的指针，在这个基础上程序实时合成一个新的图像来显示而已。

理解软件如何实际工作常常有助于人们使用它，但这种理解需要很大的代价。表现模型允许软件创造者通过简化软件透明的工作方式来解决问题。这种成本完全是内部的，用户永远不必知道。抛弃了实现模型而更接近用户心智模型的用户界面更好。



公理　用户界面应该避免实现模型，而支持用户心智模型。

第三章 新手，专家和中间用户

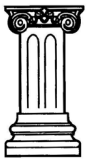
永久的中间用户

大多数用户既不是新手，也不是专家，相反，他们是中间用户。

进行某种活动的不同经验层次人数分布，就像大多数人口分布一样，遵循着经典的统计正态分布曲线。对于几乎所有需要知识和技巧的活动来说，如果我们针对不同的熟练程度画出人数曲线，位于曲线左边的新手和位于曲线右边的专家人数相对而言都是较少的，大多数都是位于曲线中间的中间用户。

然而，统计数据并不能说明所有问题。正态分布曲线只不过是瞬间时间上的快照，虽然大多数中间用户倾向于保留在这一类型中，而新手不会永远是新手。要维持高水平的技术程度很困难，因此专家们也在快速变化。新手和专家随着时间变化都会倾向于变成中间用户。

虽然每个人都会在一段时间内以新手的形式存在，但没有人会长期停留在这个状态。人们不喜欢显得不称职。而就定义来说，新手意味着不称职。相反，学习和提高是令人高兴的，所以新手努力尽快成为中间用户——或者，他们干脆放弃。例如，所有的滑雪的人，都会在新手层次停留一段时间，但那些不能很快取得进步，也就是摔跤多过滑雪的人会很快放弃这种运动。剩下的人则会从初学者变为普通的运动者。只有少数人会成为滑双黑钻雪道的高手。



公理：没有人愿意停留在新手级别。

第五章 用户建模：人物角色和目标

人物角色

要创建一个能满足广大用户群的产品时，逻辑告诉你，要使产品的功能尽可能广泛来适应最多的用户。然而，这种逻辑是错误的。成功地适应大量不同用户的最好方式是，为具有特定需要的不同个体类型进行设计。

当任意而广泛地扩展产品的功能时，你会增加所有用户的认知负担以及导航成本（导航的含义见第 11 章）。能够愉悦某些用户的功能可能会影响其他用户的满意度（见图 5-1）

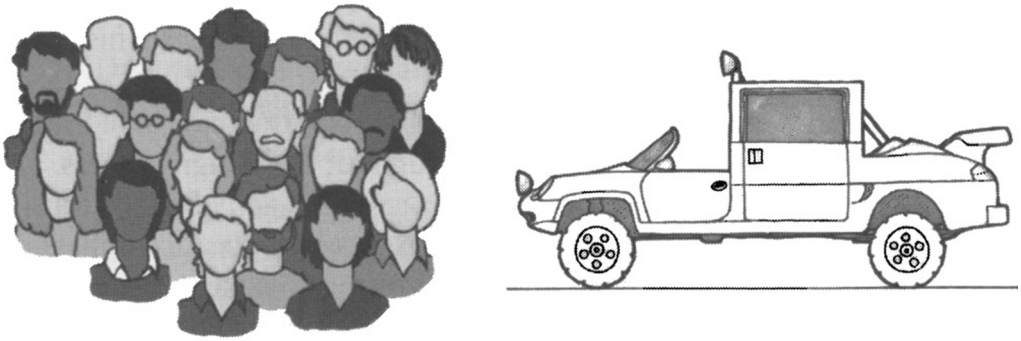


图 5-1　一个简化的例子，来说明人物角色的用处。如果你想设计一辆汽车，希望能让所有可能的驾驶者开心，最后的结果是，这辆汽车拥有所有的功能，但没有一个人喜欢。今天的软件设计也是如此，企图满足过多的用户，导致用户满意度下降。图 5-2 提供了另外一种可供选择的方法。

关键在于选择为之设计的合适个体，这些个体的需要代表着很大一部分关键成员的需要（见 5-2），也在于了解如何区分设计元素的优先级，解决最重要用户的需求，同时尽量不为其他用户带来明显的妨碍。作为一个强有力的工具，人物角色能够帮助我们理解用户需要，区分不同的用户类型，并且对用户进行优先级排序，以便在功能和行为设计中确定最重要的目标。

图 5-2　人物角色实用性的简单示例。通过为不同具体目标的用户设计不同的车子，我们的设计同时能使其他与目标驾驶者有相似需求的人也感到满意。对于数字产品和软件设计来说，这种情况也同样成立。

第 9 章 协调和流

引导，但不要讨论

很多开发人员认为理想的界面应该与用户进行双向交流。然而，大多数用户都不这样想。例如，他们更愿意用和自己的车交互的方式与软件交互。打开车门，上车，然后去目的地。要继续向前时踩油门，想停下来时踩刹车，转弯的时候打方向盘。

这种理想的交互情形不是对话，更像是在使用工具。当木匠看到锤子时，他不想和锤子讨论钉子的问题。他会直接用锤子钉钉子。在车里，如果司机想改变方向，他转动方向盘。司机喜欢通过合适的设备从车子和外部环境直接获得反馈：挡风玻璃外面的视野；仪表板的读数；疾驰而过的风声；轮胎压在道路上的声音；对侧向重力的感觉以及路面传来的的振动。木匠也希望有类似的反馈：钉子下沉的感觉，铁互相击打的声音以及举起锤子的感觉。

司机当然不期望车子通过对话框与自己交互，木匠更不希望看锤子上显示这样的信息（如图 9-1）。

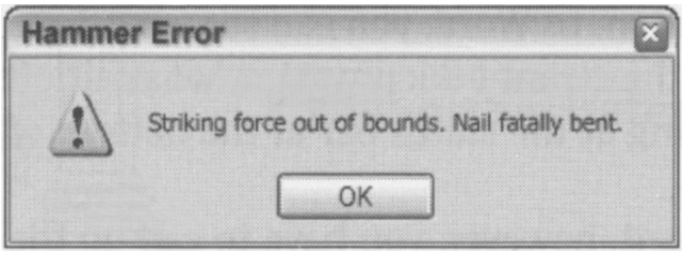


图 9-1　程序员习惯看到这样的消息，并不意味着其他行业的人们也希望这样。没有人希望他使用的机器责备它。如果以错误的方式启动机器，我们期望获得错误的响应。当然，这可以防止我们犯严重的错误，但责备不同于保护。

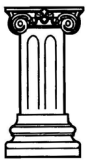
软件常激怒用户的原因之一就是它们并不像车子或者锤子那样工作。相反，软件经常不合时宜地把我们引到对话框，指出我们的缺点，并且要求我们做出响应。从用户的角度看，软件颠倒了自己的角色。应该是用户提要求，而软件响应。

使用直接操作能够获得我们想要的效果。如果用户希望将对象从 A 处移动到 B 处，那么用户点击它并把它拖放到合适的位置。总的来说，更能引导流状态的界面，是那些具有丰富而完善的直接操作习惯用法的界面。

……

提问与提供选择

提问不等于提供选择，两者之间的差别就像逛商场和面试。人们通常认为，和被提问的人相比，提问的人更优越。权威者提问，而从属者做出响应。向用户提问让他们感觉低了一等。



公理：提问不等于提供选择。

对话框（特别是确认对话框）是提问，工具条是提供选择。确认对话框停下来，要求用户回答问题，直到得到答案才会消失。另一方面，工具条始终存在，安静而礼貌，像琳琅满目的商店一样提供商品。为你提供手指一点就舒适地选择的机会。

与大多数软件开发者所想的不同，问题和选择并不是一定会让用户感觉强大。更普遍的是，这会让用户感到有压力，并为此烦恼。你喜欢汤还是色拉？色拉。你喜欢甘蓝还是菠菜？菠菜。你喜欢法国、千岛群岛，还是意大利的菠菜？法国。你喜欢本地的还是普通的？够了！快给我汤！你喜欢杂脍还是鸡肉面汤？

用户不喜欢被问。不断的提问向用户暗示着程序是：

- 无知的
- 健忘的
- 功能不强的
- 不主动的
- 不能自理的
- 烦躁的
- 过分要求的

这些都是人类不喜欢的性格。为什么我们期望软件这样呢？程序问的不是一些智力问题，也不是期望像朋友们在宴会上那样交流。相反，那是无知的行为，并且表现出不适当的权威。程序对我们的观点不感兴趣。它需要信息，通常是那些一开始并不需要向我们询问的问题（如何避免这些问题的讨论见第 15 章和第 34 章）。

比单个问题更糟的是反复提出那些不必要的问题。你想保存文件吗？你是否想现在保存文件？你真的想保存文件吗？用户感觉那些很少提问的软件更聪明，更有礼貌。因为如果用户不知道如何回答，就会觉得自己很愚蠢。

在《The Media Equation（Cambridge University Press, 1996）》一书中，斯坦福大学社会学家得出了引人注目的结论：人类对待和响应计算机以及其他交互产品的方式和人类相同。我们是否应该注意产品显现出的人格特性。胜任工作、乐于助人，或者抱怨，唠叨，强迫或者找借口。我们会在第 14 章讨论如何让软件更有礼貌和体贴人。

选择很重要。但基于表达的信息做出自由选择和被程序以模态的方式审问之间存在差异。用户更愿意像在街道上驾驶汽车一样引导他们自己的软件。汽车向用户提供了全面的选择，而不发起一次对话框。想像一下图 9-7 描述的情况。

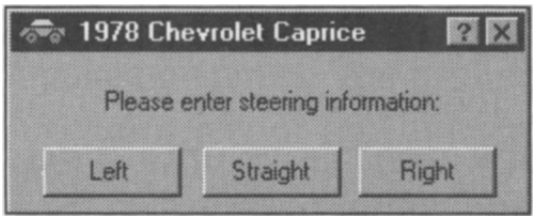


图 9-7 想象一下，如果必须点击对话框上的按钮来驾驶汽车，这会让你感受到普通人对对话框的感受：让人难堪，不是吗？

直接操作方向盘对于汽车来说是最合适的习惯用法，它让用户感觉自己处于优越的位置，引导你的汽车去目的地。没有用户愿意像审判席上的嫌疑犯那样被人质疑，然而这正是我们的软件通常要求我们做的。

第十章 消除附加工作

什么是附加工作

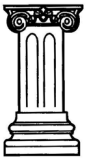
当我们决定开车前往办公室时，必须打开车库门，上车，启动发动机，停下来，在动身前往目的地之前关闭车库门。所有这些动作都是为了支持汽车，而不是为了支持我们前往目的地。如果有星际迷航的传输器，我们呼叫相应的目的地，然后立即出现在那里。不需要车门，不需要发动机，也没有红绿灯。我们的意思不是抱怨开车复杂，而是要区分完成日常任务时需要采取的两种动作类型。

任何大的任务，诸如开车前往办公室，涉及到很多较小的任务。有些任务直接实现目标，像操纵方向盘直到你的办公室。另外一方面，附加工作不直接实现目标，但与此同时，对于实现任务非常必要。这些任务包括打开和关闭车门，开动引擎，在红绿灯面前停车，给车子加油以及进行周期性的维护。

附加工作是在我们努力实现目标的同时，满足工具或者外部代理需要的其他工作。这两者有时候难以区分，因为我们习惯于附加工作成为任务的一部分。对于经常开车的人来说，区分打开车门的活动和开车前往目的地的活动非常困难。开关车门通常是我们为车子做的，而不是我们自己。它不像油门踏板和方向盘那样将我们带向目的地。在红灯面前停下是我们所设置的要求，并不能帮助我们实现自己真实的目标（在这种情况下，它确实能帮助我们实现安全到达办公室的相关目标）。

软件在目标导向的任务和附加任务之间存在相当清晰的分界线。像汽车一样，一些软件的附加任务是微乎其微的，操作它们不需要多大困难。另外一方面，一些软件的附加任务像补车胎一样令人讨厌。安装就是这样，还有配置网络、备份，连接在线服务等都是附加任务。

附加任务存在的问题是在消耗我们的精力，而不是直接实现我们的目标。在消除了附加任务的地方，我们能够让用户更加有效率和更有生产率，并且能够改善软件的可用性。作为一个软件设计师，你应该对附加工作的存在非常敏感，并且向治疗感染的医生一样有激情地采取步骤根除附加工作。



公理： 消除附加工作使得用户更加有效率。

第十一章 导航和调整

提供合适的控件——功能映射

映射描述了控件、它影响的事物以及预期结果之间的关系。当控件与它影响的事物之间没有视觉上或者象征上的关系时，映射关系很不自然。不自然的映射关系影向用户的进度，用户不得不停下来思考控件和它影响的事物之间的关系，从而打断了流状态。控件到功能的不自然映射也增加了用户的认知负担，并且可能会产生严重的用户错误。

一个很好的例子来自煤气灶，而不是数字世界。任何做过饭的人，都会对做饭用的煤气灶旋钮没有合适地映射到它所控制的火炉感到恼火。如图 11-8 所示，典型的煤气灶有四个火炉，每个火炉占用正方形的一角。然而，控制火炉的旋钮却在前边的面板上成直线排列。

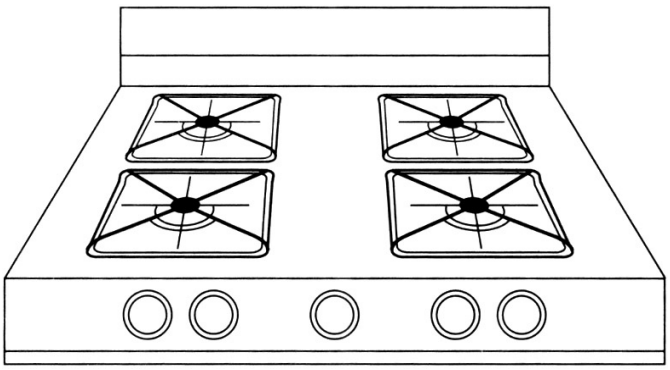


图 11-8 煤气灶表面控件的物理映射很不自然。最左边的旋钮控制的是前面的火炉还是后面的火炉？用户必须在使用煤气灶时，重新确定它们之间的关系。

在这个例子中，问题是物理映射。使用控件的结果应该显而易见：当你旋转一个旋钮时，会点燃一个火炉。然而，控制的目标——哪个火炉会变暖？——不清楚。旋转最左边的旋钮控制的是左边前面的火炉，还是后面的火炉呢？用户必须通过试验或者参考旋钮下的图标来确定。这种不自然的映射，迫使用户在每次使用煤气灶时都不得不重新确定映射关系。虽然随着时间推移，这种认知过程会慢慢变成无意识的行为，但它仍然存在。如果用户匆匆忙忙或者注意力不集中（在人们准备膳食时，通常会这样），就会很容易出错。用户因为旋错了旋钮而感到自己很愚蠢，或者在用户意识到自己的错误之前，食物还没有加热，这些都还算好的，最糟糕的是，可能会不小心引起厨房的火灾。

解决方案是调整旋钮的物理位置，这样它们能够更好地表示所控制的火炉。旋钮不必以火炉同样的模式来布局，但它们应该定位成每个旋钮的目标很清晰。图 11-9 中的炉子就是一个有效的控件映射例子。

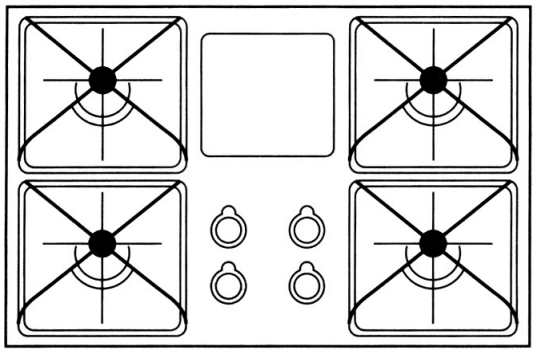


图 11-9　清晰的空间映射。在煤气灶上，旋钮和火炉的映射非常清晰，因为旋钮的空间布局清晰地将它们与火炉联系起来。

在这种布局中，很清楚，左上端的旋钮控制着左上端的火炉。每个旋钮的位置都在视觉上清晰地显示出它所控制的火炉。Donald Norman(1989)将这种更直觉的布局称为“**自然映射**”。

另一个不同类型的不自然映射例子如图 11-10 所示。在这种情况下，概念到动作的逻辑映射不清晰。

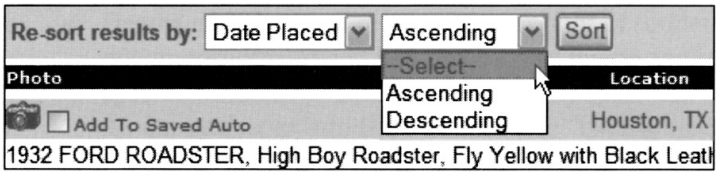


图 11-10 逻辑映射问题的例子。如果用户想首先看到最近的项目，他是选择 Ascending 还是 Descending 呢？这些术语和用户对时间的理解之间没有很好的映射关系。

这个网站使用成对的下拉菜单，根据时间对搜索结果进行排序。对第一个下拉菜单的选择决定了第二个下拉菜单提供的选择内容。在第一个菜单中，当选择 Date Placed 重新排序时，第二个下拉菜单显示 Ascending 和 Descending 选项。

不像前面的炉子旋钮，这里控件的目标非常清晰：下拉菜单选项会影响下面显示的列表。然而，使用控件的结果不清晰，如果用户选择上升时，会获得什么样的排列次序呢？

正是这些用来表达日期分类选项的术语，使得用户如果想在列表中先看到最近的项目时，无法确定该怎么选择。上升和向下不能很好地映射到用户对时间的心智模型。人们不认为时间存在上升和向下之分。相反，他认为时间和事件可以分为最近的和最老的。快速修正该问题的方法是，将选项使用的单词改为“首先显示最近的”和“首先显示最老的”，如图 11-11 所示。

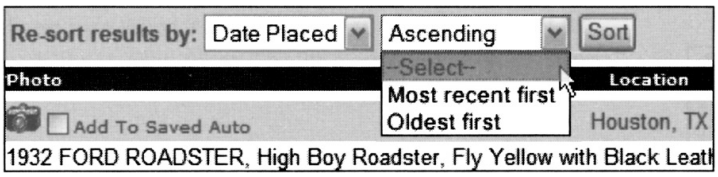


图 11-11 清晰的逻辑映射。使用术语“最近的”和“最老的”使用户容易映射到基于时间的分类。

无论是创建家电、桌面应用还是 Web 站点，你的产品都可能存在映射问题。映射是一个关注细节就会得到回报的领域：即使没有多少时间来做出改变，你仍然可以通过寻找和解决映射问题来让产品得到明显的改善。如果说结果，那就是产品会更加容易理解，并且使用起来更愉悦。

软件与系统思想家温伯格精粹译丛

现代需求技术的基石

探索需求

设计前的质量

Donald C. Gause
Gerald M. Weinberg
／著
章柏幸　王媛媛　谢攀／译

Exploring Requirements: Quality Before Design

UMLChina 训练辅助教材

清华大学出版社

Agile软件开发丛书

PEARSON Addison Wesley

有效用例模式

Patterns for Effective Use Cases

Foreword by Craig Larman

[美] Steve Adolph
Paul Bramble 著

车立红 译

UMLChina 审

UMLChina 指定教材

清华大学出版社

用例点估算方法

Mel Damodaran、Aqua Netta E. Washington 著，徐异婕 译

讨论

摘要

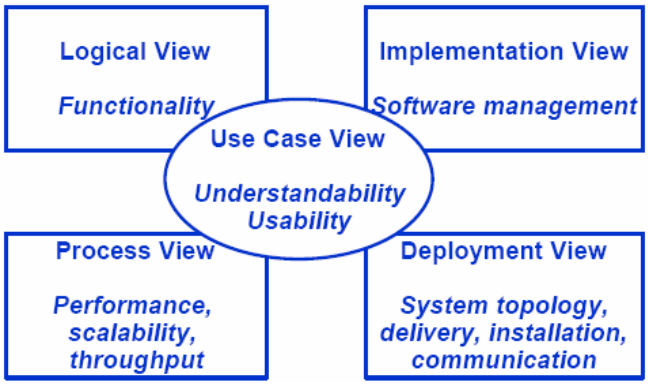
用例模型越来越多地用于捕获和描述软件系统的功能需求，可以有不同的途径和方法使用用例估算工作量。一些研究者试用了用例点方法，并分析了他们的发现。结果（虽然不是结论）表明，用例点方法有潜力成为估算的可靠素材，它非常象功能点方法；而且，软件开发项目的规模对估算结果影响很大，特别是当和专家估算法结合使用时。另外，用例建模技术的应用日益广泛——它不仅用于描述软件和系统需求，还是设计、开发、测试、部署、配置管理和维护的基础——因此，基于用例模型的估算方法是很有意义的。本文讨论成功运用用例点方法来估算软件开发项目的规模和工作量的可能性，以及使用中的主要限制，并提供可能的解决方案。

关键词

用例点、软件开发工作量估算、软件项目估算

1 介绍

捕捉软件项目的功能需求时，常使用用例模型。用例建模是一项业界广泛使用的技术，用于描述和捕捉软件系统的功能需求。用例点（use case point）是一项估算软件开发工作量的新方法。既然用例和场景技术已成为需求采集和分析的标准部分，并且它们捕捉了用户需求的精确描述，那么，基于用例和场景——而不是诸如功能点和代码行等其他技术——进行更加困难的开发规模和资源的估算工作，是很有意义的。基于用例估算的另一个优点是，通过使用现代需求管理工具，可维护其双向可跟踪能力。可以在用例和多种软件工程工件之间，维护这种双向可跟踪能力，包括设计、编码、测试、配置管理、架构和部署等相关的工件。用例位于统一过程的4+1模型的中心。用例是基于使用的和以用户为中心的，而不是面向系统的或面向设计的；它描述“what”而非“how”，因而它比功能点和代码行方法有更高的健壮性和稳定性。用例是工件，是对用户有意义的功能单元。



Bente Anda (Anda 2001)比较了用例点方法和专家估算法，后者是由有经验的软件开发者进行的。结果发现，用例点方法给出的估算和有经验的软件开发者给出的估算很接近。该估算方法的相对误差值和相对误差平均值都在可接受范围内。研究结果表明，用例点方法可以成功地用来估算软件开发工作量。

Objectory AB公司（后来被Rational Software收购)的Gustav Karner开发了软件估算的用例点方法(Karner 1993)，其灵感来自经典的功能点方法。Arnold and Pedross (Arnold and Pedross 1998)也研究并报告了他们使用用例点方法的经验，其具体方法类似，但并非受Karner影响。他们发现的结果证实了他们最初的假设，即用例是估算软件开发工作量的可靠素材；另一方面，他们注意到用例模型分解的详细程度不同，估算的规模也不同；他们还发现，因为用例没有标准格式，所以仅依据用例估算软件规模比较困难。

基于用例的估算还有其他方法，其中一种是使用用例模型作为统计功能点的手段，然后再进一步获得工作量的估算值;还有一种方法是,将用例模型用作估算代码行数的手段,然后再进一步获得工作量的估算值(Anda 2001)。上述两种方法都试图利用业界广泛应用的基于功能点和代码行的估算方法。

业界当前使用用例点估算方法还很少。一个值得注意的应用案例是，印度班加罗尔市Infosys公司的Rakesh Agarwal等人为他们为客户开发的一个Internet项目(Agarwal 2001)。作者报告称，他们使用用例点方法“作为近似估算系统开发工作量的手段”。业界没有更广泛使用该方法的一个可能的原因是，该方法没有被集成到流行的项目估算工具之中。另一个原因可能是，尽管用例是描述需求的标准方法，但用例点方法相对较新；没有良好的历史生产力数据，不可能用它估算工作量和成本。

人们设想，基于用例模型来估算规模和工作量是可能的。因为用例模型被广泛用于捕捉功能需求，所以人们认为，应该有基于用例的估算方法来代替功能点方法。然而，有几点困难可能阻碍基于用例的估算方法的应用(Smith, 1999)：

- 存在很多种用例规约风格和格式，这使得定义量度（metrics）非常困难。
- 用例应当描述外部角色对系统的观点，所以一个50万行代码的系统的用例，和一个5千行代码的系统的用例，根本就不是一个级别。
- 不同的用例其复杂性不同，无论是编写用例规约时，还是实际实现用例时，都是如此。
- 用例应从角色的视角描述行为，但这可以很复杂，特别是——这其实很常见——当有不同的系统状态时。所以，为了描述该行为，试图捕捉行为本质的时候，可能得到一个包含太多功能分解和细节的系统模型。

问题依旧，我们应该放弃使用用例，而使用分析模型和设计模型来完成估算吗？可是这会带来另外一个问题——延误了估算的时间，不能使采用这种方法进行项目估算的项目经理满意。早期估算是需要的，因此必须使用其他方法。如果能获得早期估算，项目经理进行计划就更加有效，然后可以不断地改进它们；而不是延期估算过程，无计划地开展工作(Smith 1999.)。

关于用例描述和用例规范，有大量书籍，但很少是关于如何从用例获得估算量度的。文献(Graham 1995)提出了使用“任务脚本”来解决用例的诸如长度和复杂程度变化很大的问题。Graham的“原子任务脚本”是收集“任务点”量度的基础。该任务脚本方法的问题在于，任务脚本的层次非常低，不可再分解了，理想情况下是一个句子。John Smith (Smith, 1999)倾向于把根任务（root task）看成和底层用例相似的概念，而原子任务脚本看成用例中的步骤。因此，“层次”问题依然存在。

本文的其余部分，将首先讨论Karner的用例点方法。然后，将之与功能点方法比较，并分析该估算方法的暗含之意；在讨论该估算方法的潜力时，我们也会给出更广泛应用它的建议。

2 KARNER 方法

现在，简要说明Karner的用例点方法的步骤 (Karner 93.)。首先，将用例模型中的角色按照简单、一般和复杂三个等级来分类，并统计每类角色的数量，分别乘以各自的权重系数（weighting factor），然后相加，得出总的“未调整角色权重（unadjusted actor weight，UAW）”。然后，依据用例中的事务数量（包括备选流中的交易），将用例按照简单、一般和复杂三个等级来分类，并按如下步骤计算未调整用例权重（unadjusted use case weights，UUCW）：统计每类用例的数量，分别乘以各自的权重系数，然后把这些乘积相加。最后，将UAW和UUCW相加，得到未调整用例点（unadjusted use case points ，UUPC）。

接下来，基于技术系数和环境系数调整用例点的值。每个系数是介于0到5之间的一个值，具体由它们对整个项目的假定影响来确定。该步骤相关的3个公式如下：

技术复杂性系数：

TCF = 0.6 + (.01 * Tfactor)

环境系数：

EF = 1.4 + (-0.03 * Efactor)

已调整用例点：

UCP = UUCP * TCF * EF

最终，UCP要乘以一个代表生产力的历史数据的系数，比如，每个用例点20人时（person hour），得到项目估算结果——完成项目所需的人时总数。

3 用例点估算方法的相关讨论

尽管用例点方法受到了功能点方法的影响，但它们有几点不同。第一，功能点方法不要求标准的符号；相反，用例点方法基于标准的用例模型，使得开发自动计算用例点的估算工具比较容易。第二，有计算功能点的国际标准；而用例点还没有标准和标准实践，因此关于如何开发用例以及如何计算用例点，每个组织各不相同。

用例模型本身会影响估算过程。例如，将相似的角色合并成一个角色——角色超类，角色数的改变就会影响估算结果。由于这样只将角色计算了一次，因而提高了估算精确度。另一个问题是，估算软件项目时，包含用例和扩展用例是否应当计算。有的研究将这些特殊用例忽略，而另一些研究却决定算上它们。显然，应当对这个特殊问题进行更多研究，因为当前研究的结果太不确定。另一个影响估算的因素是用例描述的详细程度。用例中事务的数量衡量每个用例的规模，当建立用例时，很难决定每个事务的详细程度，也没有现成的标准可以参考。而详细程度一定会影响事务数量，后者又对用例点方法的估算结果造成直接影响。

但另一方面，对一个组织来说，开发他们自己的用例分析粒度标准，并将用例定为简单、一般和复杂三个级别，并不难。从而，每个用例成为功能单位，所有用例构成了软件——界定了项目范围，就象功能点界定项目范围一样。对用例点和功能点两种方法而言，非功能需求和其他约束，都必须作为项目规模最终估算值的调整系数被考虑进去。如果组织有用例使用标准，在不同项目和每一时刻推行，就可能拥有一个不错的历史度量数据库，

该数据库可以支持进行可靠的成本和资源估算。

4 结论

总之，估算工作量的用例点方法非常值得加入到为项目经理提供的工具之中。该方法很可靠，至少象诸如COCOMO、功能点和代码行等工作量估算工具一样可靠。所有这些估算方法都有误差，并且要求精确的生产力历史数据，以使估算在组织范围内真正有用。用例点方法对总是采用用例技术的系统开发项目尤其有用，它可以和业界普遍重视的功能点方法相媲美。用例点方法有时比专家估算法更准确。但是，专家估算不应该被估算过程排斥出去，而是应该和用例点方法结合使用，以保证估算精确。最后，本方法有潜力成为成熟的、被广泛接受的估算工具，这少不了标准化工作以及国家和国际的支持，这些支持曾推动了功能点方法被广泛接受。

参考文献

[1]Agarwal, Rakesh, Santanu Banerjee and Bhaskar Gosh, 2001, “Estimating Internet Based Projects: A Case Study.” Quality Week 2001, Paper 6W2

[2]Anda, Bente, D. Dreiem, Dag Sjøberg and Magne Jørgensen, 2001, “Estimating Software Development Effort Based on Use Cases - Experiences from Industry”, In M. Gogolla, C. Kobryn (Eds.): UML 2001 - The Unified Modeling Language.Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, LNCS 2185 Springer

[3]Arnold, P. and Pedross, P. 1998,” Software Size Measurement and Productivity Rating in a Large-Scale Software Development Department.” Forging New Links. IEEE Comput.Soc, Los Alamitos, CA, USA, pp.490-493.

[4]Graham, Ian, 1995, Migrating to Object Technology. Addison-Wesley, 1995.

[5]Karner, G, 1993, “Metrics for Objectory”. Diploma thesis, University of Linköping, Sweden. No. LiTHIDA-Ex-9344:21. December 1993.

[6]Smith, John, 1999, “The Estimation of Effort Based on Use Cases. Rational Software.Cupertino, CA.TP-171. October.

Domain-Driven DESIGN

Tackling Complexity in the Heart of Software

众多问题根源在于
领域模型没有捕获到
业务的深层内涵

Eric Evans

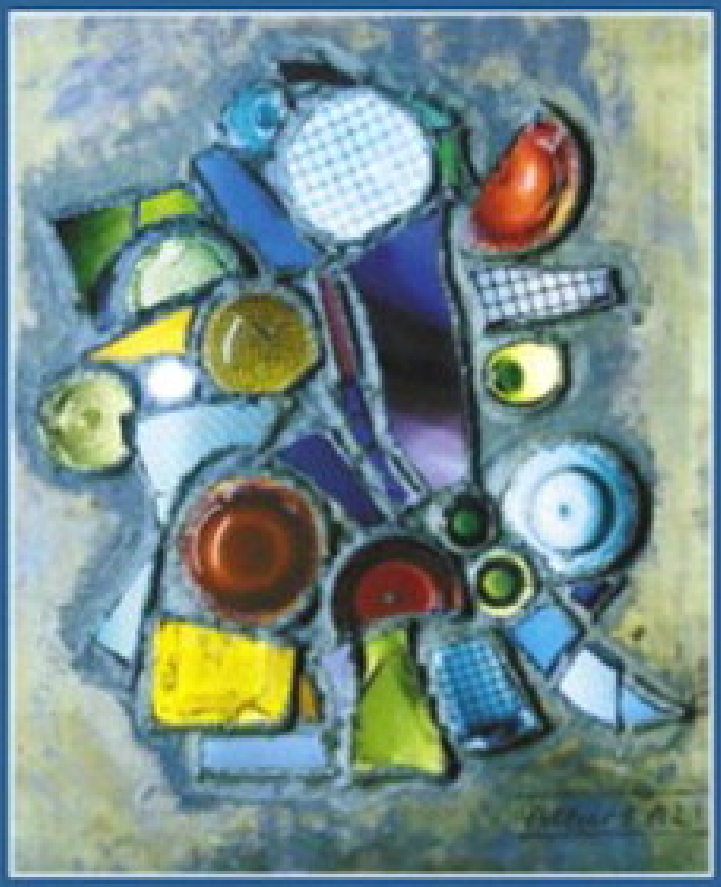
Foreword by Martin Fowler

《领域驱动设计》中译本

即将由清华大学出版社出版，UMLChina 审稿

Object Design

Roles, Responsibilities, and Collaborations



一说起“面向对象”就是“设计模式”？ ——对象的回归！

Rebecca Wirfs-Brock and Alan McKean

Forewords by Ivar Jacobson and John Vlissides

《对象设计》中译本即将由人民邮电出版社出版

鲍志云 译，UMLChina 辅助教材

使战略规划和敏捷开发相一致：扩展敏捷思考用于业务改进

Colin Rand、Bruce Eckfeldt 著，liteng 译

摘要

许多开发团队成功地采用了敏捷开发而成就了高质量的软件，但是这些产品在对于整个公司成功的贡献上却往往是失败的。我们认为这种失败应归咎于许多公司的战略规划并没有同利用敏捷方法的灵活性优点协调一致而导致的。

我们确信敏捷开发的力量，但是我们认为仅仅这样是不足以使每个软件项目都获得成功的。整个业务目标要成功，项目必须采用符合战略的方法，如果敏捷开发想在商业社会继续成长下去，就必须开发出同敏捷哲学所一致的战略规划方法

1 介绍

Cyrus Innovation是纽约一家专注于敏捷软件开发可用性设计和实施咨询的咨询团体。我们确信在软件项目的开发过程中，我们象开发人员一样会忘记我们是在为别人生产一个系统，而且尤其是并非所有的业务问题都能通过没有几个Bug的情况下通过写软件来解决。

敏捷的哲学是确信在项目中会发生任何事情，因此准备应对变化，将决策延迟会对整个项目有利。因此，当软件开发过程能够处理变化，而业务自身却不能应对变化，此时会发生些什么呢？

为了解决这个问题，我们延伸了敏捷开发的哲学，在系统整体设计中从管理方法学的角度采用了系统分析和全面设计以建立一个全面的解决方案。当我们采用这种方法的时候，在开发过程中出现了一些开始时所意想不到的有趣的事情，而且如果我们仅仅采用敏捷技术的话，这些事也不会被注意到。

以下就是我们在这种软件开发之外的范围延伸中所获得的一些特别的经验的详细说明。

2 案例情况

我们的一个客户——一家在曼哈顿有着25家连锁店和在长岛的一个中心物流供应的餐馆连锁企业，Qeens（距曼哈顿5分钟车程），出于快速增长而引起一些方面的问题，需要改进他们的生产过程。他们的主要目标是减少每天食品切割处理时所产生的浪费的生产成本。生产总监认为如果他能够减少1%的食品损耗，他就能在成本上一年大概节约200000美元。他希望新的订单系统能够将损耗降低15%。

我们客户的主要目标是：通过改进订单系统，使他的连锁店和物流供应中心降低因为订单信息不准确而导致的生产过剩。他认为让店长改进订单的精确性，使物流供应中心可以减少过剩生产。为了做到这一点，他想从他们的基于传真的系统迁移到基于web的订单系统，从而可以获得每天的订单，店面库存以及当日的全部的损耗情况。

基于上述情况，采用敏捷过程，我们开始建立了一个简单的订单系统。

3 关注于项目目标实现

我们意识到，尽管我们已经进行了功能收集，但是如果仅仅把这些功能在系统中实现的话，并不能给客户带来真正的价值…

在典型的敏捷开发的项目中，我们第一步要做的会是同客户开会确定他们所想要实现的功能，然后进行评估和确定优先次序。

在刚开始同客户的会议中，我们对他们想要 实现的功能和优先次序进行了讨论。因为目标是降低产品损耗，他们首要的是想知道实际损耗到底是多少，包括：

提供给店长跟踪其每日库存水平的损耗跟踪报表
生产经理能够看到每个店长每天会有多少过剩订单
生产部门能够看到按产品类型分类的报表
店长能够看到每天会有多少过剩订单的报表
生产经理可以下载CSV格式的订单历史记录

表1 开始的故事

经过这个起始阶段，我们意识到，尽管我们已经进行了功能收集，但是如果仅仅把这些功能在系统中实现的话，并不能给客户带来真正的价值。因为我们并不知道从开始到现在的库存情况，我们发现只进行损耗跟踪并没有多大用处。

然后，我们花了一些时间快速地分析了系统将要实施的生产环境。我们的分析关注在生产过程中的一些步骤和顺序，以及这些步骤如何为我们的客户带来商业上的价值。结果就是一张反映公司价值流的图，从产品销售开始，到下订单补充库存，然后是生产更多的产品，最后是产品运到店里。

依照这个模型，我们同客户坐下来，并且决定再做一次设计。主要是关于我们如何能够改进这个流程以增强它产生商业价值的能力。这就变成：

提供给店长跟踪其每日库存水平的损耗跟踪报表
物流中心能够看到每天的订单的总计资料
物流中心能够打印每个店的订单的包装
物流能够打印每个店的订单的运费发票
店长能够记录每天剩余订单的数量

表2 经过价值流分析后修订的故事

在这个周期之后，如XP所需要的，同时也加入了敏捷业务的价值需求，这个例子中我们的客户拥有的不仅仅是一个高质量的软件产品，他们拥有了一个清晰，有效地带来业务价值的系统。如果他们决定改变他们的优先权

而关心他的公司的另外一个部分，那么这个系统对生产而言还是非常有价值的。

4 假象的诱惑

“我们制造了假象的诱惑 - 客户在尚未了解订单系统是否稳定的情况下就作出了采用新方法的决策”

在项目的第二次迭代周期，我们遇到了一个情况，这种情况现在看来是一个泄漏的信号：它并不关注于整个过程而仅仅关注部分过程的优化

他们现在的订单系统显示店长在下午4点向物流供应中心发出订单，但是这个时间对物流供应中心准备第二天的产品是不够的，因此他们实际上是在早上10点开始产品的生产。结果是物流供应中心经常会生产出比店里订单多出大约20%左右的产量。

店长的问题在于他们直到下午4点才能确定当天的销售量，这个量是他们确定第二天销量的最好的依据。基于他们目前的生产过程，物流供应方面不存在能够解决这个问题的方法。虽然我们考虑了实时方法和小的分散式的物流供应，但是在目前并不可行。

在确信对订单处理过程有控制能力以及希望降低物资损耗的情况下，客户要求我们实现一个在上午11点前将店长锁在系统之外的功能，以此迫使他们在掌握实际需要的供货量之前就把订单发出来。我们同他在物流中心一起坐下来开了一个会议，他说：“哈哈，我现在能让所有的店长在中午之前就把他们的订单发出来！”

考虑到这样作只会使店长们多发订单从而应对销量的变化，我们决定向Fordham 大学的Georgantzas博士求教，他是业务策略和系统动力学的教授。同他的讨论使我们认识到为了解决订单下达时间的问题我们必须理解产生变化的各种因素以及开发一个系统的方法来改进整个处理系统。

Georgantzas博士向我们解释了我们首先要做的事情是确定在新的系统中我们需要捕捉哪些数据。他同我们共同创建了一个初步模型，模型反映了系统中的库存流动，从而确定了在哪些点可以收集信息。

他给我们的建议是我们应当在试图降低损耗的时候首先去发现这些损耗到底是在哪里产生的。现在我们客户的生产过程是要求物流中心多生产20%以处理店里的订单变化。如果我们迫使店长们更早地发出订单而避免使物流中心减少剩余生产，只会使店长们发出更多的订单，结果使损耗仅仅在整个过程中换了一个地点发生罢了。

我们的业务建模经验告诉我们最初的这种系统功能并不可行。在这个案例中，决策应该交给店长和物流供应经理他们自己。改变订单发出的时间并不会降低系统的损耗，它仅仅使损耗发生的地点从物流供应换到了门店。更糟糕的是如果门店里发出的订单并不足以满足销售的话，会导致销售上的损失。

由此，我们认识到了一个拍脑袋实现的功能对于系统的改进是无济于事的，也不会帮助公司降低损耗。

5 系统动力学

“……整个系统的关键在于掌握每天的实际需求量”

在最初的系统上线并为店长们开始使用之后，我们同客户召开了一次会议以确定下一步系统中所需要添加的功能。

我们讨论了一些功能，诸如向店长提供决策支持和中心订单系统。但是每次我们开始进一步考察系统需要处理的细节的时候，我们发现在这个企业的业务中存在太多的变量，从而令我们无法确定哪些功能是最重要的。

我们发现在计划开始的时候，我们的客户认为自己对整个生产的细节都已经很了解了，但是当我们进入功能设计和开发的时候，我们发现实际上他对这个物流系统的动态性和复杂性并不了解。

于是再一次借助专家的意见，我们同 Georgantzas博士开发了一个系统的动力学模型，以更深地了解整个系统中的库存运转情况。我们分析了当事人(店长，物流经理、生产经理，还有购买产品的顾客)对整个系统的影响和交互过程。很快我们认识到，整个系统的关键在于把握某一天真实的需求量。然而，由于捕获客户的偏好过程费用昂贵以及很难理解顾客在门店的购买决策是如何做出的，如果不影响门店销售的话，最后的方案是从当前系统中捕获关键数据去理解诸多因素是如何相互作用的。

认识到这一点，我们能够在订单系统中开发收集指定时间段的特定类型的信息。结果是，当我们同客户再次碰面的时候，我们使用捕获建模所需数据来确定哪些功能是最重要的。

现在我们使用这些功能所收集到的数据在动力学模型中来模仿不同的情景，并且来测试哪一种最准确的可以模仿历史数据。得到的结果将是我们用来决定哪些功能为客户带来最大的商业利益。

我的经验是，绝大多数客户对想要实现的功能都有很多的想法，但是因为对于业务环境的动态性和复杂性缺乏了解，很难去权衡哪些功能在长期内甚至短期对企业是真正有效的，通过建立系统动力学模型来进行不同的场景测试，有可能更好地了解哪些功能可以改进原有系统和增强企业竞争力。

6 反思

采用系统的方法来判断哪些功能更为重要对于开发过程是非常必要的。传统的业务分析技术并不能满足需要，因为他们没有考虑到系统在变化，他们实际上并不愿意面对变化。因此，敏捷开发过程不能依赖于传统的技术，而必须寻找更适合的技术，诸如系统动力学（systems dynamics）

在这个项目中，我们采用了两种方法来确定功能的优先顺序，并且设计了第三种。对于第一种，我们使用了流程分析来确定如何给我们客户的业务尽快地增值。第二种则使用了系统动力学来确定功能的优先顺序。第三种方法则是当数据采集完毕后要进行的，通过在模型中进行情景分析来确定哪些功能可以为客户降低成本发挥最大的作用。

XP的客户经常会倾向于不考虑功能对整个业务系统的影响来驱动开发。象Dr. Porter在《竞争战略》一书中所述，“企业中的每个功能部门肯定都会全力以赴地进行专业方向的发展和激励。但是，这些部门的努力之和却往往不会达成最好的战略。”这种倾向会导致一个系统中局部范围的优化，而对关键的全盘改进却是影响非常有限的。

从Lean Thinking的观点看来，即使有效地提交了一个没人需要的产品也依然是一种浪费。这正是现在许多XP开发队伍所面临的一个问题。。如果业务经理们不采用同XP方法所一致的分析技术，那么开发人员自己就变得只关心软件开发过程中的优化，结果也就造成XP自身不能驱动整个系统的改进，然后为不能维继下去。

近些年来，已经出现的一些技术来推进整个业务的改良，包括Lean Thinking和约束理论。而系统动力学的出现则提供了一个强大的工具来对系统中哪些部分需要改进来进行预测。在系统分析中综合采用这些方法和敏捷开发对业务过程的改进而言将会是一个非常好的机会。

7 致谢

感谢Georgantzas博士和Luca Blanzuoli博士帮助我们建立了系统动力学模型。

参考文献

[1] See Chapter 18 of Agile Management for Software Engineering by David J. Anderson (Prentice Hall, 2004) for a discussion of the business benefits of agile methods.

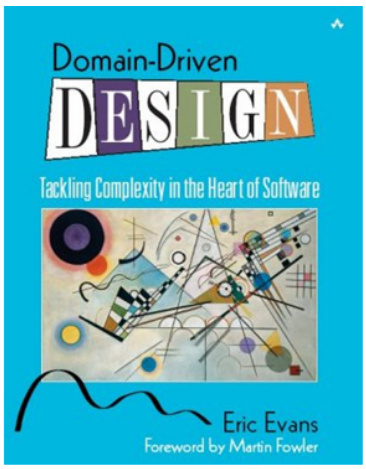
[2] One of the rules of Lean Thinking is to empower workers to make their own decisions

[3] As described on systemdynamics.org, "System dynamics is a methodology for studying and managing complex feedback systems, such as one finds in business and other social systems."

[4] Competitive Strategy, Michael Porter, Free Press 1980

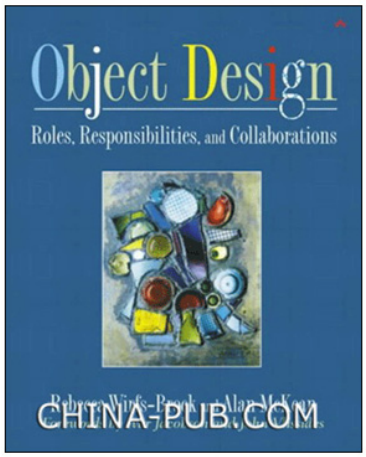
[5] Described in Lean Thinking, Womack and Jones, Free Press 1996,2003

[6] Described in The Goal, Goldratt and Cox, North River Press 1992



《领域驱动设计》

对象技术的回归



《对象设计》

"*About Face 2.0* could completely redefine how software applications are created!"
— Pete McBreen, author, Software Craftsmanship

ABOUT FACE 2.0

THE ESSENTIALS OF INTERACTION DESIGN

ALAN COOPER
& ROBERT REIMANN

当木匠看到锤子时，他不想和锤子讨论钉子的问题。他会直接用锤子钉钉子。在车里，如果司机想改变方向，他转动方向盘。司机喜欢通过合适的设备从车子和外部环境直接获得反馈：挡风玻璃外面的视野；仪表板的读数；疾驰而过的风声；轮胎压在道路上的声音；对侧向重力的感觉以及路面传来的的振动。木匠也希望有类似的反馈：钉子下沉的感觉，铁互相击打的声音以及举起锤子的感觉。

International Bestseller
Completely revised and updated

UMLChina 辅助教材。中译本即将由电子工业出版社出版，詹剑峰 译

审稿：国内首个交互设计网站 De Dream'

THE UNIFIED MODELING LANGUAGE REFERENCE MANUAL, Second Edition

JAMES RUMBAUGH
IVAR JACOBSON
GRADY BOOCH

Covers UML 2.0

UMLChina 译
（王海鹏、李嘉兴）

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

CD-ROM included

《UML 参考手册》2.0 版中译本
即将由机械工业出版社华章分社出版

WILEY

TIMELY. PRACTICAL. RELIABLE.

UMLChina 指定教材

Agile Database Techniques

Effective Strategies for the Agile Software Developer

《敏捷数据》

UMLChina 李巍 译

机械工业出版社即将出版

Scott Ambler

《人月神话》和《功夫》……

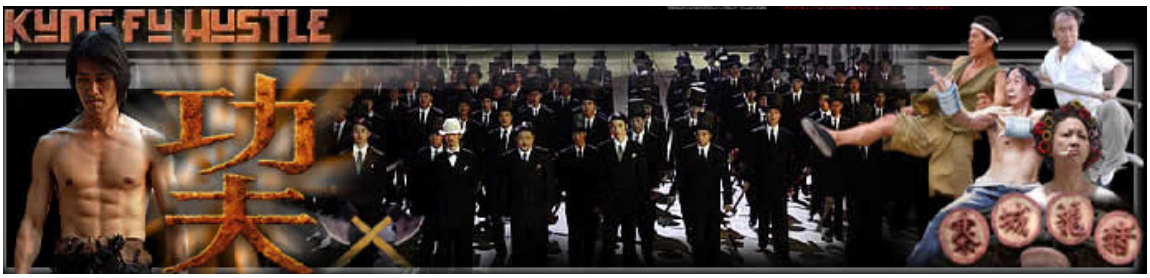
UMLChina 整理

讨论

http://cnweblog.com/asj/

功夫

呵呵，不是观后感，因为我还没看这部片子。不过赶上这个热潮，天天耳边有广告给你喊“功夫，功夫”，不由得有点感想。



现在说起功夫这个词，恐怕任何人第一个想到的都会是双截棍和“啊～～呀～～～～”的一声怪叫，李小龙实在是太成功了，不仅让外国人了解了功夫，也从新定义了国人心中的这个词。让我们几乎忘记了它的更普通的含义。

功夫代表时间：“要作出这么个东西，可得花功夫的。”

功夫代表工作量和主观努力：“不下功夫怎么会作得好呢？”

功夫代表了技术含量和人员的水平：“这活，没真功夫的绝对玩不来。”

忽然发现“功夫”是一个描述软件开发度量的理想单位（终于绕回本行了），想想《人月神话》中反复讨论的人－月之间的关系，想想项目管理三角中的资源－时间－功能和三角中间的质量。“功夫”两个字，似乎都已蕴含其中了。

我们的民族有长久而辉煌的工艺史，也有着从长期工艺实践中提炼的哲学智慧。就像武术从文人的个人修养中借用“功夫”这个概念一样，利用我们的文化资源，迎合中国读者习惯的“武侠文化”未必就是一件坏事。

如果我们用“这个软件需要花 xx 功夫”，而不是代码行、人月、功能点之类的来和客户交流，是否会使得项目的规划更容易避免踏入人月的误区呢？也许，正是一件需要下功夫的事。

http://blog.codelphi.com/huk/archive/2004/11/29/30020.aspx

了解了这个神话，我们就可以采取主动行动.

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

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

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

新员工需要培训，薪水低. 培训好的员工会向薪水高的地方流动，公司不得不找新人，再度开始培训. 这是个死结. 这个世界上并没有那么多的熟练工存在. 即使是劳动密集型工厂也是如此，何况软件公司. 我们的教育体制有 2 个极端，要么就是象在培训"科学家"(教学一些似乎很深奥的课程)，要么就是培训"记忆"(除了背诵后能解决考试问题,其他什么也不能解决),我也是这个体制中出来的，幸亏我还有兴趣看自己喜欢的书... 这样的教学体制很难让我们更强大. 但是目前也没有更好的体制能提出，谁让我们的人口这么多呢.(关于人口和教育体制的因果关系很复杂，不是我有兴趣写的范围)（陆麟）

http://think.blogdriver.com/think/225643.html

前几天，把人件看完了。从去年上半年（？）开始到现在，终于把人月神话、人件、最后期限这几本书通读了一遍。因为中间经常被其他的事情打断，所以阅读战线拉的很长。看大师的作品，果然如行云流水，是一种享受。现在凭着记忆，讲一下自己最强烈的一些零星的感受，我的理解也许会有偏差：

一、人月神话：

1、把项目的工作量定义成人月，并认为人和月可以互换，例如认为一个 500 人月的项目，如果 100 个人做 5 个月可以完成，而 500 个人做只需要一个月就可以完成，这只是一个神话。人类的认识需要时间周期、项目的运行也需要时间周期，这个周期相对固定，不会因为人多而变短；相反，向一个进度滞后的项目中添加人手，只会使项目更加滞后，因为需要额外的培训、交流，需要占用现有人员的时间。

2、项目初期就安排 100%的人手是没有必要的，只有设计完成后，才大规模地需要人。我觉得这一点可能适用于大系统，对于像敏捷开发和极限编程这样的方法来说，并不适用。

3、最好将项目的完整概念保持在一个人的头脑中，这样可以保持整个项目的概念完整性。如果项目过于庞大，可以由一个小组来充当项目的“大脑”工作，总之整个系统的完整概念（我理解是 level0 的架构）应该由一个或很少的人来负责。

4、将一个能够运行的项目的程序，提升成产品，工作量的代价是 1：9

二、没有银弹

没有任何一种方法或者工具，可以在十年内使得软件开发效率有数量级的提高。根据我最近做过的两个 J2EE 项目（用到 jsp＋java bean＋XML，没有 EJB）的数据，我们的人均 java 开发效率为：

项目 A：java 287 行/人天 加上 jsp+xml 297 行/人天

项目 B：java 275 行/人天 加上 jsp+xml 149 行/人天

因为项目中包含了很多以前积累的现成的框架程序，所以实际的效率可能还要低一些。这个数据跟汇编、c、c++等大概都在 250 行/人天的数据大致相符。

所以说，软件行业距离成熟的路还很远，软件行业的大幅提高生产率的“工业化革命”还遥不可及，现在基本处于“手工作坊”阶段，差别只是作坊的大小不同而已。

三、人件

软件难就难在人是其中的决定性因素，用传统的工业化管理方式管理软件开发人员是行不通的。

四、最后期限

我感觉是大家在被无数个最后期限搞的精疲力竭之后，向往的一个神话，也是一个寓言。是个放松的好材料。（thinking 2004 年 07 月 2 日, 星期五 18:21）

http://www.donews.net/bluez/rss.aspx

抽烟看报是可以同时进行的，甚至和“排毒养颜”一块进行都无所谓。

但是抽烟时候你能和女孩打 Kiss 吗？

游泳洗澡是可以同时进行的，如果你带齐家生，顺便做做护理都行。

但是游泳时候你能看到岸边 MM 的衣领没扣好吗？

写程序跑客户也是能够同时进行的，如果你腿够快，中间还能抽空去小个便。

但是，各位老大啊，你们有谁能一边写程序一边钻桌子修电脑？

前些天领导训话时候曰的，说很多项目经理都不写程序，对自己也是浪费，又耽误事。。。等等等等。

并举出外科医生手术的例子，把项目经理和主刀医生扯一块。

领导认为：团队里面，项目经理应该亲自编程，其他人都是支援人员等等。

领导认为：一个程序员为什么就不可以多懂点编程之外的事呢？比如说看看客户脸色，揣摩一个客户喜好如程序界面喜欢什么颜色等等。不要固步自封嘛。

领导认为：每一个人都应该出去面对客户嘛！

领导继续认为：程序员的工资应该是浮动的，根据代码质量发放奖金。

领导谈到了《人月神话》

领导说：散会，晚上有空开另一个会。。。

http://babyfacer.computer.mblogger.cn/

经过那个学期我就形成了一种观点，基础永远是最重要的，只要把基础学习扎实了，什么东西很快就能学会，触类旁通！所以花更多时间在学习高级的一些应用上面还不如花在基础上面更划算呢！你见过程序设计都没有学好的同学想通过学习《人月神话》,《Peopleware》这些书来达到高手的境界的么？还有些同学光关注程序语言的学习，关注 practical 的学习，对其中本质的东西确不闻不问！在这里我又想引用候老师的一段话，个中味道自己去品位吧：程式语言是程式员最基础的功夫，必须熟练它掌握它。但若只是如此，编写不出好程式、大程式、专业程式。大家都会说中文，独独有人旁徵博引，左右逢源，字字珠玑，文思隽永，这是为什麼？功夫在语言之外！

http://www.blogit.cn/rss.php?blogId=748&profile=rss20

我的运气一直就是很好的，出生后就赶上了改革开放，上学的时候有很多书看，这些破烂先不提。

特别是毕业后刚好赶上招工高峰时期，求大于供，去了一家日资企业（很多当时读研的同学现在毕业后，工资福利待遇什么的远不如我一个破本科生），然后就是出国研修，回来时辞职刚好赶上有家公司需要我这样的人，然后项目结束后走人的时候又赶上朋友推荐的猎头公司的挖人招聘，正好有适合的工作岗位，简直不能再幸运了，呵呵。

就连《程序员》杂志，《CSDN》杂志，《深入浅出 MFC》等书籍的出版时间都是严格的配合我毕业初次工作，技术提高，项目转换等时间的。严丝合缝，分毫不差，精确度令人吃惊。

我编码的时候，市面上主要就都是编码方面的书籍，等我做管理工作了，市面上的管理书籍也开始大量的涌现。连《人月神话》《设计模式》等书籍中文版的出版都是严格的配合我的工作侧重点的。

前一段时间离职的时候考虑想要总结一下小企业项目风险承担如何进行科学量化的问题，结果刚好就出现了《与熊共舞》这本书，都不用我总结了，而且还帮助我学会了不少科学的量化的东西。

最不可思议的，nnd 我刚跳槽到咨询管理公司，市面上就出现了一本好评如潮的管理咨询书籍的中文版，叫《咨询的奥秘》。

真邪门了，我都不知道说什么好了。

时势造就英雄还是英雄造就时势，我不去讨论了，但可以肯定的是，当前时势已经有了，我是否是英雄，是否能够成为英雄，还需要我自身的不懈努力。

http://www.surstar.com/blogx/Rss/

关键是我们如何做预防审美疲劳呢？

软件的开发的过程是一个反复的过程，对自己的界面 时时刻刻的面对，一种很简单而豪无意思的说法是：换一个角度来看的产品， 也许有一些人可以做到，但是他们真的做到吗，这和一个人美术细胞有关，一个修养有关，我认为 要做到：吹毛求疵，在《人月神话》里，作者这样说到，程序员是一个天生就有追求完美的天性，这也是在锻炼过程中生成的！（大意），我认为 对自己的产品吹毛求疵是可以做到的

http://blog.javaeye.com/index.php?op=ViewArticle&articleId=98&blogId=41

看《人月神话》得到的一个项目变更决策启示

《人月神话》中有这么一句话：

“每个产品都应该有数字产品号，每个版本都应该有自己的日程表和冻结日期，在此之后的变更属于下一个版本的范畴。”

之前在处理项目的 BUG 和小的需求变更的时候，总是会遇到应该归到哪个版本下的问题。因为在这些 bug 或者小需求出现的时候，总是会有新的开发分支正在进行中，因此常常会为决定这些修复或者改进工作应该放入哪个版本或者分支而费心，而其结果常常导致版本之间的不一致。不过这里仍然有一个问题：如果前一个版本冻结之后，所有的后续工作放入下一个版本，但是假如（事实上通常会这样）后续的版本正在进行重大开发，那么这些小修改和改进也要和新版本一同发布吗？显然客户不会同意，在他们看来那些只是小问题，根本不应该归并到新版本中去，他们并不想要这个结果，只想要尽快能够解决问题。

不过，不管怎么说，尽可能在恰当的时机冻结版本应该是一个好的决策倾向。（youngS）

http://sanwangx.brain-c.com/archives/2004_04/18_5/

（出去旅行时带上《人月神话》...）

整理好家裡，當然接下來要整理後天要出發的行李囉

還真多咧，不過帶回來的行李一定會更多 :p

書本資料類

出發前 7 天海外旅行英語會語手冊

經學長加持友情贊助的，雖然我英語還可以，不過還是多抱點佛腳才不會太難過咩

The Mythical Man-Month (人月神話)

月初才出版的，正好拿來飛機上殺時間(;-))

前人的 Universal Studio 和 Walt Disney World 各 theme park 的 map guide

「知己知彼，百戰百勝。」先熟悉一下到時進園才能玩得盡興囉，不過學長也口頭描述了好幾個小時了，包準到時玩得直呼痛快

http://rockychen.blogchina.com/blog/article_38647.214276.html

引进版书的书名究竟该如何翻译，学界一直存有争议，有的认为要尊重作者，应该直译；有人却认为要入乡随俗，意译比较妥当。其实直译或者意译都不是惟一的选择，只有按需应变才是正解。

意译虽然能入乡随俗，但却需原作者同意后方能修改。万一作者不同意，那也只能沿袭原先的书名。

其实就算是直译的书名，也有一些相当不错的翻法。比如《人月神话》就是非常成功的一个例子。它原名为《The Mythical Man-Month: Essays on Software Engineering》。最初引进的时候，曾有人想翻译成《神秘的人月》或《被神话的人月》，最后是简化成了《人月神话》。为什么要叫《人月神话》呢？因为这本书就是讲的如何统计一个人单月的工作量。

http://chekhovchina.blogchina.com/blog/article_27886.95866.html

产品概念完整性和团队构成（2004 年 09 月 04 日）

目前汉化 X 国人的程序马上就要告一段落，中间顺带读了读 X 国人的程序，虽然同事都说 X 国人的程序很臭，但是我觉的 X 国人的程序很不简单：上百兆的源代码，几十个工程中有大量的重用模块；所有程序的界面风格几乎一致；调用方式一致（但也可以明显看出这是不同水平的程序员写的程序，因为有的程序往往在犯一些很低级的错误，例如写一个错误的 SQL 语句）。对照一下自己，以我现在的水平，似乎很难作到这一点，在以往的项目中，我们的程序经常会出现 N 个人通过不同的方式在读取一个 INI 文件，或是每个人都写自己的身份证号码校验

程序，如果一个程序由多个子系统组成，那么各个子系统的界面和调用方式也是根据不同人的习惯而有所不同。整个程序下来，功能可能是都完成了，但是却怎么看怎么别扭，同时由于程序模块的大量重复和风格的不一致，程序的维护也有很多的问题。

自从买了《人月神话》后，当每个项目结束时，我都会重新读一次这本书，因为从这本书里，的确可以找到很多在项目进行中碰到的问题的答案。这次重读《人月神话》，我觉的我在它的 3，4，5，6，7 章中找到了部分答案，即如何保持产品的概念完整性，而要保持产品的概念完整性，又和整个开发团队的构成密切相关，所以在 3，4，5，6，7 章中，也涉及到了开发团队的构成和沟通方式，当然，团队构成和沟通是个很大的话题，但是这次我只写写同产品概念一致性有关的团队构成的一些感想，其他就不提了。这篇文章，就算是《人月神话》这几章的读书笔记吧。

以前研究所的软件开发中，往往是以这种方式进行：团队中骨干人员写出需求，然后骨干人员进行一个分析，画出 ER 图并且划出各个功能模块，然后根据模块功能不同，骨干人员完成最复杂或最核心的那部分，其他复杂的模块交给其他骨干，一些简单的模块交给新手。最后一个综合，交付了整个程序。这种开发方式带来了很多问题，主要表现在：

- 模块的可重用性差,整个程序依据功能模块的不同交给不同的人开发,因此经常出现一个程序中有十多个身份证号码校验函数的情况.
- 程序风格严重不一致,虽然在开始开发前,整个团队会规定一些规格说明,例如背景色,字体,程序风格等,在开发过程中,不同的开发人员也会对这些风格不断的调整,力求一致,但是因为人的差异,最后交付程序的风格还是差异很大
- 需求无法最后贯彻执行到程序中,当程序人员根据需求完成程序的开发时,需求人员却发现功能虽然完成,但是却没有贯彻需求的整体意图(往往体现在界面和操作方式上),当需求人员要求开发人员改动时,因为界面和操作方式的改动往往是个体力活,没有人愿意作重复工作,所以造成了开发人员的抵触.

后来去公司混,又感觉到了一种新的概念,就是在写方案和写需求时,头儿往往要求往大了写,具体就是,在写一个方案时,为了打动用户,能想到什么就写什么,能怎么吹就怎么吹,头儿提的要求是,”想到不写出来,是能力问题,写不出东西来,就代表你水平差,至于写出来了作不作,能不能作,那另说”.这样带来的后果就是方案,需求,和开发完全脱节,开发人员面对给客户看的方案和需求,不知所措,按需求来吧,方案写的天花乱坠,作下去要死人的;不按需求来吧,那就不成体统了,所以最后也没有人把需求当回事。

对于研究所那种方式,我以前想的解决方案是,通过增强设计能力,用一个好的系统设计,来保证模块的可重用性,按我以前的设想,在需求结束后,设计是个很重要的阶段,我希望能够达到的目标是,在设计阶段,就能事先规划好程序模块的重用;设计,可以直接指导编码.所以,当时看到 UML 后,如获珍宝,整天琢磨着边界类,实体类,控制类等等,以为按这套东西,就可以达到设计这个境界.可是时间一长,发现自己的力道的确不够,怎么也达到不了这个境界,也许有的大牛可以,但是我到不了.

在《人月神话》第 3，4 章中，Brooks 提出：

- 概念完整性是系统设计中最重要的考虑因素
- 为了获得概念完整性，设计必须由一个人或者具有共识的小型团队来完成
- 要得到概念上的完整性，就必须有人控制这些概念，这实际上是一种无需任何歉意的贵族专制统治

当然，在 Brooks 的概念中，概念完整性和体系结构，更多是指的是需求上概念，这里我扩展一下，我想表达的概念完整性，是指在整个系统开发过程中，需求，设计，编码的一致性。也就是说，在最后交付的程序，能够很好的表达需求人员的设想；整个程序具有统一的风格，所有的程序具有统一的结构（例如相同的调用方式，可重用的模块）；开发人员能够很好的贯彻需求人员和设计人员的意图。

但是，在实际中，概念完整性却很难执行，这主要表现在：

- 设计的不确定性，也许有人能进行很好的设计，但是我更接受源代码即设计这个概念（或者说我只能达到这个层次），只有在编码时，才能发现大量可重用的模块。这个时候谈模块的重用有点晚。例如，大家很容易把通讯模块交给 A 开发，B 开发业务模块 1，C 开发业务模块 2，B,C 都调用 A 开发的通讯模块，但是 B，C 却都自己写自己的身份证代码校验模块。
- 对执行的漠视，国内有句俗话，四等人员 CPP，三等人员 DOC，二等人员 PPT，一等人员 TMD。有经验的成员，不到万不得已，往往不愿意进行具体的开发工作，他们往往让新手或一般程序员来完成编码工作（CPP），自己热衷于写出各种设计文档（DOC）来指导新手或一般程序员进行开发，而产品的设计人员多用一个 PPT 来说明产品的概念，当最后交付程序时，发现产品的概念和现状差别太大时，最高领导大骂 TMD 一顿后，继续把任务交给新手来开发。实际上，我觉的，越是高层，越是有经验的人员，越要深入到开发一线，通过细节的执行来贯彻整个产品的完整性。

- 开发人员水平的不一致性导致产品的不一致性，由于在开发过程中往往是按功能模块划分，如果开发人员水平高，那么这个模块可能看起来会比较舒服，也比较稳定，反之，另外一个由新手开发的模块就看起来很丑陋。
- 团队成员沟通上的障碍，当需求人员完成需求后，如果交给其他人员设计，中间会丢失一部分信息，设计人员在交给开发人员后，也会丢失一部分信息。最后就导致产品的设计概念和交付产品的严重不一致性。

所以，从上面的分析看出，如果想保持产品概念的一致性，关键在于要建立起一个成功的开发团队。按 Brooks 的说法，应该建立起一只外科手术队伍，这个队伍包括

- 外科医生（也就是首席程序员）来定义说明书，设计程序，编制源代码，测试和书协技术文档。
- 副手：能完成外科医生的任何一部分工作，但是主要是作为设计者，讨论者和评估人员，我的理解上这个人主要是同外科医生进行沟通，通过强强交流来保证设计的正常进行
- 管理员：替外科医生节省管理上和行政上的时间。类似 PM
- 编辑 / 文秘：协助外科医生和管理员完成文档等工作
- 程序职员：他把程序员从文书等杂事中解放出来，同时保证文档等其他杂事的质量，我感觉应该是配置管理员。
- 测试人员：不多说
- 语言专家：对技术进行研究，为外科医生解决技术上的问题
- 工具维护人员：我没有理解这个人的作用。

毫无疑问，这是一只很理想的队伍，通过外科医生，来保证概念的一致性，同时大量的辅助开发人员，把外科医生从繁琐的日常事物中解放了出来。但是，这太理想了，我估计国内的开发中几乎不可能形成这么一只队伍。实际情况往往是：

- 项目经理：这个人多半是技术出身，他制定计划，写出需求，作出设计方案，和客户进行沟通，有时还要亲自动手，完成若干核心模块的开发。也有的时候，这个项目经理是不动手的，只作所谓的管理，但是却拥有技术的决定权。

- 骨干：如果项目经理在这个团队中搞技术，那么骨干的作用多半是负责某个模块，或是某个模块的核心，如果项目经理不搞技术，那么这些骨干多半就写需求，或写设计方案等等。
- 新手或一般人员：他们就是编码。

这样带来的后果就是::

- 无法保证概念完整性，需求人员，设计人员没有精力去深入到编码，也就是产品的细节当中，因而也就无法在最后的产品中体现概念的完整性，同时，也无法进行模块的重用
- 造成人员之间关系的紧张与对立，在开发前期，由于沟通问题，产品的需求和设计人员（PM 或骨干）无法和实现人员（新手或一般人员）就琐碎的细节进行沟通，在交付产品时，项目经理或骨干发现产品的不一致性，要求实现人员返工，导致 2 方关系的对立和紧张。
- 结构设计人员制定出无法实现，或代价高昂的需求或设计，使所有的人陷入困境。
- 最低层开发人员觉的工作无聊，因为涉及不到核心工作，他们会觉的自己是在进行着无意义的体力工作，他们的创造性和构思被项目经理或骨干剥夺，从而士气低落。

Brooks 似乎已经给出了答案

- 尽早交流和持续沟通能使结构师有较好的成本意识，以及使开发人员获得对设计的信心，并且不会混淆各自的责任分工。
- 牢记是开发人员承担创造性的实现责任；结构师只能提出建议。
- 时刻准备着为所指定的说明建议一种实现的方法，准备接受任何其他可行的方法。
- 对上述的建议保持低调和平静。
- 准备对所建议的改进放弃坚持。
- 听取开发人员在体系结构上改进的建议

我对此的总结和补充是：

- 产品的一致性必须在制度上给予保证，不能分割需求，设计和开发的一致性，这个产品的技术负责人（无论他是 PM 还是架构师）必须建立技术上的绝对权威来保证产品的一致性。
- 技术负责人必须严格的贯彻执行能力，也就是说，在需求，设计完成后，技术负责人必须深入到编码的细节中，他应当指导一般开发人员编码并帮助他们解决具体的问题，而不是命令或要求他们解决问题。
- 不要把编码变成一种体力劳动，编码才是真正的创造产品的决定性部分，在编码过程中，保证编码人员与技术人员的良好的沟通和交互，不要等到产品提交测试时才由设计人员提出这样不行。而是由一开始就由开发人员和设计人员来决定什么样才行，实现也是一种快乐。
- 在不同阶段，都要保持所有人员的持续参与与沟通，鼓励开发人员对设计提出意见，并迅速作出反馈。让开发人员也参与设计，但是不决定设计，通过这种持续的反馈和沟通来实现模块的重用
- 加功能由结构师（PM 或设计人员）来决定，减功能由开发人员来决定。

上面这五点应该体现出 3 个关系

设计与实现

管理和技术

集权和放权

但是在实际中，要想处理好这 3 个关系，仍然存在这很多困难，主要表现在

- 对技术负责人压力过大，在中小型团队中，往往是技术负责和管理负责是同一个角色，同时具有管理技能和技术技能的人很难找到，同时具有思想家和实干家就更难找。很难有人能够同时处理好管理与技术，宏观与微观的关系。
- 一般编程人员一开始就没有热情，现有的工作对他们没有多少吸引力，他们只是为工作而工作，因而在项目中拒绝沟通，对项目也没有多少的兴趣。

http://lilybbs.yaoge123.com/vd953334/blogdcon?userid=photogis&m=9&d=24

每当想对程序与程序员的职业发表看法，就想起了人月神话的第一章，然后就长出一口气，我所想说的，都被人家说过了。

http://teacher.zjnu.cn/lester_ruan/more.asp?name=zjlester&id=2

[偶尔发酸]这几日看的几本书

五一的假期可以做一次很好的精神旅行，因为没有打扰且没有体力劳损之虞

。。。。。。

也看了一本关于合作学习学术著作，只是尚未完工，引文甚丰富，文字也不见晦涩，算是学术类书籍类中不错的一本。

只是中国学者在这方面大多缺乏自信，自己的体系没有太多坚持。

有时候我想：类似〈人月神话〉的随笔风格写学术说不定更会受欢迎。<人月神话>单就行文风格就很让我们怀念

http://threeseven.mysmth.net/pc/pccon.php?id=342&nid=101241&s=all

你看到我今天穿了仔裤滑雪服蹬着翻毛皮鞋背着我结实的包迈着大步进电梯的模样

我坐在椅子上一手搭在那小小的台面一手托着下巴痴想

没有窗户的教室冷气对着我嗖嗖作响

谁知道我现在构思我要去西藏

幻灯片里讲的是几个人对着项目拍拍脑袋画着圈圈线线

什么 Eearly start Late finish 资源有限人工有限赶紧交货时不我待

人月神话

这真的只不过是个神话

http://www.51one.net/info/6909784635566102.htm

不过另一个我现在依然不得其解的问题是有关开源团体如何协作开发的问题（请原谅，我还是跑题了）。像 JBoss 那样有数十人专业团队者自不待言，而至于 Apache 那样的数万人开发规模（甚或 Linux），他的协作方式，至少 Brian 的解释（《程序员》2003 年第 5 期）并没有解除我的疑惑。因为，正如 Brooks 在《人月神话》里所提到的，人越多并不代表越容易完成既定任务，沟通的成本，最主要的是某些任务具有不可分割的特性，又岂能轻松“摊派”给多人呢

http://blog.onweb.idv.tw/mt-comments.cgi?entry_id=365

相信我，在台湾，我还没看过哪家软体公司能做到对时程实务的估算，往往 PM 所做的所谓 "人/月"，那根本叫造假应付，一点都不切实际。(可参考：人月神话 一书)

而所要求要团队成员所写的 "工作报告(TimeSheet)"，我们团队都戏称为 "黑奴系统" ^_^

http://wenj.blogchina.com/blog/article_672.30197.html

《人件》-一本让中国老板害怕的书

这是一本可与《人月神话》相媲美的书，是一本让擅于给员工洗脑的任某某、柳某某害怕的书，是高举人本主义大旗的书，我特别推荐给工作狂们阅读。中国员工看过本书后的反响见详细内容。（射木，2004 年 07 月 31 日）

http://brandworker.blogchina.com/blog/month.6742.200407.html

刘亚东的“软件恐惧症”与计算机科学家布鲁克斯的论断不谋而合。这位曾获得图灵奖的学者在其著名的作品《人月神话》中说道，无论哪种方式，过去几十年的大型系统开发就像一个焦油坑，很多大型和强壮的“动物”在其中剧烈地挣扎。他们中大多数开发出了可运行的系统——不过，其中只有非常少数的项目满足了目标、时间进度和预算的要求。各种团队——大型的和小型的，庞杂的和精干的，一个接一个淹没在了焦油坑中……

在中国，这个“焦油坑”其实不仅仅在刘亚东所熟悉的电信业，伴随各行各业信息化的加快，这样的“焦油坑”随处可见。尤其是在那些金融、电子政务、大型制造行业等，往往因为竞争与效率的要求，他们的信息化走得更为积极与迫切。“而这，也就有更多的机会去制造一个又一个的‘焦油坑’。”

http://www.popsky.com/bbs/viewthread.php?tid=1010

已然不记得 2003 年的元旦是如何过的了。当时的心情彷徨而茫然，心乱如麻。随手翻开《人月神话》，赫然印的是“焦油坑”！真是焦油坑啊，无论是工作还是感情，感觉自己在不断下陷，下陷……

那段日子想来十分不堪：一直到过年，几乎没写一行代码，没看一页书，每天不是往别的办公室跑就是木然的对着电脑扫雷。对一个以程序为天职的人来说，这是很可怕的事情，感觉上接近崩溃的边缘。

http://www.flashanywhere.net/mxna/blogview.cfm?blogid=98http://www.flashanywhere.net/mxna/blogview.cfm?blogid=98

前天，我的朋友 sniper 说他在读《人月神话》，并从中得到了很大的教益。这本书很早以前就听说过，而且还是很受推崇。但自己一直没有去读过，只是感觉自己还没有去读的水平。其实读一本好书也是有要求的。如果没有书所设定的认知基础很难读懂，就算是读了也只是过眼就忘。一本好书更值得反复读，“温故而知新，可以为师矣”。为师很难说，但知新一定是有的。

http://rmwz.blogchina.com/blog/category.php?id=925&pageno=2

［转载］为了学习技术而失去了很多东西的人进来看看

学习不单单是学技术的，学习怎么和人相处，学习怎么让人开心也是学习的，怎么和人协调，怎么样让别人注意到自己，怎么样去表现自己，这都是要学习的，然而这些东西在书上是学不来的，特别是在计算机科学类的书里，更是没有。

不知道来这里的人有多少是看过《人月神话》、《人件》的？如果看过的话，而且承认里面的观点的话，就应该知道，项目的成功，以及个人的成功，并不是单单依靠技术的，很大程度上是依靠社会科学的。

而且，看样子，好多人想一下子成为行家里手，拼命的学习，节假日，正常的休息，等等时间，都投入了学习，好像都想在多少多少岁之前把所有的东西都学全，于是埋头于书本和程序之间，忘记了人间，说起来真的好感动，在感动的时候却带着些许的同情和悲哀。

为什么要那么拼命学习呢？仅仅因为兴趣吗？如果是兴趣，那为什么要拼命的赶时间的去学呢？怕比别人落后吗？也许吧，如果你真的是在走自己的路，那为什么要以别人的目标为自己的目标呢？

如果，你编程的目的是为了那诱人的工资，为的是职业的发展；如果，你想升经理；不要全情投入到技术当中去吧，已经有了太多的例子说明单单是技术不能让你得到这些，仅仅靠技术得到这些的人实在是太少了，但是，却有很多的例子告诉我们，高工资，当经理不需要精湛的技术。去跟不是程序员的朋友去走走，不用整天的学技术，跟女朋友去看看电影，逛逛公园，去做做这些无聊的事情吧，跟朋友去喝喝酒，聊聊天，可以让平淡的心多一些色彩，可以让自己不那么郁闷。把你学习的时间分一部分出来吧，用在生活的琐事上。

如果，你编程是为了挑战难度，好吧，你更是应该看看《人月神话》和《人件》了，在这两本书里，都提到了技术并不是最难的，什么比技术更难？这两本书里也有答案，还是各位看官自己从里面找答案吧，找到了之后，你在去挑战这个比技术更难的事情。会让你更有成就感的。

程序员们，美好的生活就在身边，升职的机会也就在身边，只不过，需要你们暂时放下手中的技术，走出去看看。

另外就是，把上网聊天的时间用在身边朋友的身上吧，更应该把这个时间放在自己的女朋友身上。（2004 年 10 月 22 日）

http://bbs.gameres.com/showthread.asp?threadid=1749

Re: 痛定思痛.....大规模游戏的设计

大规模软件开发，如同在焦油坑中垂死挣扎．软件开发虽然可以给人很大的创造力和想象力，但这是一条曲折之路．经验告诉我，一个和谐统一的架构是从血与泪中锻造出来的．看看 Brooks 博士的＜＜人月神话＞＞，他能给你很多启发，在某种程度上他也解脱了我被束缚的心灵．

http://noslopforever.blogchina.com/blog/rss.php?blogId=29875&profile=atom

如果有人想学炮轰韩寒的那位老师那样，做好了准备对"我曾说过"四个字大加批判的话，我只能说"有人曾说"："人总是在创造未来的同时创造历史，人总是在歪曲历史的同时歪曲未来，即便认识到这一点也无济于事"。让我想想，《人月神话》中好像也有一句"无济于事"："项目总会超期，即便认识到这一点也无济于事"，虽然这句话被骂得不行，但是事实是大部分项目都是在超期的基础上完成的。

http://www.msnspaces.net/members/beloved/feed.rss

我今天去预定了下周二回北京的票，出差都一个多月了。感觉已经疲惫到了极点，没有心情继续工作下去了。《人月神话》中说的那样，如果程序员过于疲惫的去工作的话必然会导致编写的程序漏洞百出，工作效率低下，所以应该只在不得以的时候才考虑进行加班。可是很少有管理人员意识到这一点，在原来做计划的时候，计划的时间不够，最终只有采用加班加点的方式进行赶工，而这样的结果必然是程序 bug 百出。维护成本极高。

http://libnetnt.cosoft.org.cn/linglei/archives/000275.html

想当年，在黑客是建设而不是破坏的那个时期，黑客就意味着突破一切不可能到可能。黑客们走在程序语言和逻辑的平衡木上，创造了 GNU、Linux、Emacs、Perl，影响了一代人。黑客的精神是一种古典精神。布鲁克斯的《人月神话》里认为，黑客行为提供了五种乐趣：创建事物的快乐；开发对他人有用的东西的乐趣；将零部件组装在一起，看到它们以精妙的方式运行的乐趣；面对不重复的任务，不断学习的乐趣；工作在单纯的思考中，工作在如此易于驾驭的介质上的乐趣。简而言之，黑客享受着造物主的快乐。它聆听着天籁之音，为存在本身构造着对象。

http://ymvivian.blogone.net/

培训第二天

昨天还信誓旦旦要好好听讲,结果今天带了一本书去,结果就可想而知了.

这本书叫<人件>,是一本很畅销的讲管理的书,放在我书柜里好久了,一直都没有机会看,结果今天培训的主要内容成了这本书的阅读过程,不过,我收获良多.

(一本厚厚的书,我居然马不停蹄的在一天之内看完了,连我自己都觉得惊讶-看来阅读并非难事,难的只是没有时间和决心.)

和这本书齐名的还有一本,叫<人月神话>,那本注重于软件开发的过程和控制,而这本则关注从事这项工作的人-光从内容上,我更倾向于后者,因为凡事我都喜欢从人的角度出发,更关心人的生存环境和思维模式.

在听课的过程中,惭愧的很,老师的话只当耳旁风,倒是读到有意思的观点时想大笑起来,或是击节赞赏.那是种被认同或者被理解的痛快淋漓.

http://blog.jxppp.com/rss2.php

其实我也一样。我每天只有两三个小时的高效时间。看着别人那么卖力的干，还有点不好意思。不过呢，我总是组里出活最多。由此可见，“人件理论”和极限编程都坚持不加班，每周只干 40 小时，还是有点道理的。他们都清楚这么做不会降低一个小组的生产能力。

每天只能干两小时还没让我太担心，真让我担心的是完全干不了活的那些天。

http://reallike.mblogger.cn/

买了五本《人件》，自己留一本，送给四个头一人一本。

http://blog.joycode.com/5drush/archive/2004/01/08/11027.aspx

我记得李维的〈Borland 传奇〉中有段〈人件〉的作者对于微软的评价，很是经典。对于为什么那么多人瞧不起微软，他只引用用当时 IBM 做 PC 行业龙头老大时，被人控诉垄断。IBM 最后胜诉，理由是， IT 行业不象传统行业，它是一个充满风险和竞争的行业。任何小企业随时都可以发达，任何大企业随时都可以倒闭。而在这个充满风险和竞争的行业里，微软居然还能做十几年的，而且越做越好，不得不让人敬佩。

http://spaces.msn.com/members/alinnb/

12 月 11 日加班之另类心情

今天加班，整个公司就两个人，很安静，气氛很好，没有平常的压抑感（画外音 boss：平常压抑吗？你自己的心理问题吧 ）。静静地坐在电脑面前，头脑无比地轻松，思维特别敏捷，工作效率也特别地高，让我感到原来加班也是一种享受啊。

前阵子看到国外作者出的一本书，中文叫《人件》，名字还真是有趣，应该是从 software 和 hardware 改的吧 。其实是一本关于项目管理的书籍，探讨软件开发中关于人以及团队的管理。里面提到了一个公司应该给员工足够好的工作环境，包括技术低迷期不解雇员工；维系同事间的友谊；把“看得见风景”的窗口留给员工；允许员工调换岗位；不限制员工的办公地点和时间；帮助员工在家里安装宽带；员工平均每年参加 6 个培训班等等。看到这些，就不仅为这些国外的公司所折服，呆在这样的公司里，员工是无论如何也不会感到工作的压力，在他们看来，工作就是一种享受，所以，还有什么理由离开这个公司呢？

要想所有的公司都做到这些无疑是困难的，但是如果公司能够尽自己的努力给员工更好的环境，让员工感受到工作的乐趣，这就已经足够了。庆幸的是，我现在就有这种感觉 ，我只想说：我要更加努力的工作！

http://veijerd.mblogger.cn/posts/5149.aspx

项目估计是进度延迟，老板和师兄们心情都不好，已经连续 2 天开会了，明天还要开会。据说老板曰“项目你们拖着，毕业就拖着”，唉~~~师姐的意思好象是 “拖着就拖着”，有什么办法，做这种项目，需求总是在变，企业那边简直就是他们说“加这个字段”，你就要加这个字段。唉~~要不人家说做产品好呢。以后我得总结一下为什么陷入泥潭的因素。

以后我们也要朝九晚五了，郁闷~~~你这样可以，但你怎么发钱呢？有没有奖励的机制？做的多了是不是给钱多？给多少钱？等这个学期结束了，偶要问老板，银子以后怎么给法。偶不愿意去做 parttime，可偶还有自己的负担，如果实验室里解决不了的话，偶不会去理会什么朝九晚五。老板也该去读读“人件”哈。

《人月神话》网站