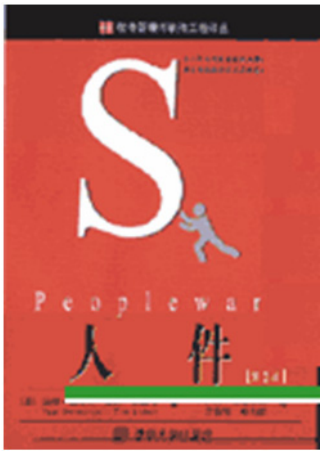
X-Programmer 非程序员

软 件 以 用 为 本

2003-2
第22期

>>目录



投稿：editor@umlchina.com
反馈：think@umlchina.com
下载：http://www.umlchina.com/xprogrammer/Index1.htm

UMLChina 公开课

“UML 应用实作细节”

（2003 年 3 月 1-2 日，北京）

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

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

本课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，阐述如何（只）使用 UML 里的三个关键要素：用例、类、顺序图来完成软件开发。使学员自然领会 OOAD/UML 的思想和技术，并对实践中的误区一一指正。整个过程简单实用，简单到甚至可以只有一个文档，非常适合中小团队。

详情请见>>

SteelTrace 公司集成 Catalyze 套件与 Rational Rose

[2003/1/7]SteelTrace 公司宣布，他们已经实现了 Catalyze（SteelTrace 公司基于用例驱动的需求及过程获取软件）和 Rational Rose（Rational 公司主导市场的 UML 模型工具）两个产品的集成。用 Catalyze 做的项目可以直接在 Rational Rose 中打开，反之亦然。没有 UML 使用经验的非技术人员可以从 Catalyze 中抽取模型及获取用例。一旦需求被获取，技术小组可以使用 Rational Rose 直接打开用 Catalyze 制作的文件。通过使用 UML 用例图来显示用例和 UML 活动图来描述每个用例，确保业务逻辑到 UML 技术实现的顺利转移。这种集成模型提供了从业务规则到技术设计的高度可追溯性，大大降低了项目开发的风险。

“完全集成（Native integration）意味着 Catalyze 和 Rational Rose 两个工具之间的连接是无缝的，提供从业务需求到系统设计，以及产品实现的完全可追溯性，” SteelTrace 公司的 CEO，Mark Melville 说。“Catalyze 通过扩展非技术人员的需求获取能力进而提高了 Rational 套件的使用价值。”

“Catalyze 与 Rational Rose 的集成使 Rational Rose 的用户从中受益，” Rational 软件公司负责开发人员市场的主管，Bill Taylor 说。“非技术用户能够使用 Catalyze 以用例驱动的方式非常容易地获取业务及系统需求。两个工具的集成，为非技术人员和使用 Rational Rose 设计复杂系统的开发人员之间就业务需求提供了一种高效的沟通方式。”

（自 tmcnet，卓锐 摘译，仅供学习交流，不得转载用于商业用途）

Chick-fil-A 统一使用 Together ControlCenter 进行 Java 开发

[2003/1/13]全美第三大快餐鸡饮食公司选用 TogetherSoft 进行应用建模、设计和开发。Chick-fil-A 公司总部位于 Atlanta 的 Chick-fil-A 公司，是全美第三大的餐饮连锁店，目前在 35 个州和 Washington, D.C 拥有超过 1050 个连锁店。在大型购物广场、单店和高速公路出口、Chick-fil-A 小矮屋?、Truett's Grill? 饮食店、以及大学校园、医院、机场、商业和工业场所等地方提供营养食物。

RALEIGH, N.C，1 月 13 日 － TogetherSoft 公司近日宣布全美最大的私人餐饮连锁店之一的 Chick-fil-A 公司采用 Together ControlCenter(TM)进行 Java 应用程序开发。应用 Together ControlCenter，Chick-fil-A 的开发团队可以对整个 java 应用生命周期进行管理。

“开发团队为要为 Chick-fil-A 以及众多操作人员提供高效的 IT 系统，保证高质量的设备运转，为顾客提供高质量的产品，这一点是非常重要的”，Josh Figaretti，Chick-fil-A 的高级分析员这么认为，“Together ControlCenter 的集成软件开发平台为应用建模、设计和编码提供了统一的环境，提高了我们的生产效率”。

Chick-fil-A 成功使用 Together ControlCenter 的早期案例包括使用 OOP 对一个劳动力进度安排系统进行改造。开发团队得到原先的设计模式，重新考虑并编译，由于代码框架已经形成，在开发阶段的时间节省非常明显，这一切，当然得归功于 Together ControlCenter 业界领先的代码同步功能。

“TogetherSoft 认识到应用开发的迭代性并且 ControlCenter 提供了贯穿开发全过程的支持，” Figaretti 继续说，“我们的架构大部分都是基于 java 的，不同的小组之间可以相互协作、技术共享”。

Chick-fil-A 认为在和 TogetherSoft 的合作中另一个好处就是可以从 TogetherSoft 的顾问和咨询队伍那里得到非常有价值的耐心的和令人愉快的服务。

“TogetherSoft 的顾问和咨询团队帮助我们处理团队的痼疾以及工作中的问题直到让我们满意为止。TogetherSoft 公司有很多实践经验很强的工程师可以帮助解决客户的各种咨询，和我所遇到的其它服务团队相比，TogetherSoft 的团队更见多识广一些”。

关于 Together ControlCenter：Together ControlCenter 是涵盖程序设计、开发和部署等各过程的应用开发环境。作为基于 UML 建模的功能全备的 IDE，ControlCenter 帮助开发人员提高创建并部署应用程序的工作效率。另外，ControlCenter 与平台无关，其中集成了多种跨越开发周期的第三方工具。

关于 TogetherSoft 公司：TogetherSoft 致力于改进开发人员的协同工作方式。其软件能够帮助企业在预算范围内快速地构建高质量应用。TogetherSoft 是软件开发提供商中的佼佼者－2001 年的收入增长达到了 81%。目前，他们的解决方案在包括 eBay、Sprint PCS、Charles Schwab & Co、Boise Office Solutions 和 J.D. Edwards 等世界著名的技术先进的公司中都得到了应用，更多信息请看 www.togethersoft.com。（风自由 摘译）

Sun 整合第三方 UML 建模工具

[2003/1/27]Embarcadero 产品符合 Sun One Studio Java 开发平台标准。

Sun Microsystems 星期一宣布把它的 IDE 和 Describe for Sun ONE Studio 整合。Describe for Sun ONE Studio 是 Embarcadero Technologies 开发的 UML 建模工具。

Describe 将作为 Sun ONE Studio 4 update 1, Enterprise Edition for Java IDE 的优先 UML 解决方案，不过由于来自两家公司，这两项产品仍然需要分别安装和购买，但能够一起工作。

提供给 Sun 的 Describe 6.0 版本使得应用建模和开发人员能够在无缝的开发和设计环境中一起工作，在 UML 驱动的集成环境中，UML 模型的可用性将大大提高，还有 Java 源代码同步，高级源代码控制集成以及基于 web 的报告等等功能。

Embarcadero 在设计和建模方面的负责人 Greg Keller 在 San Francisco 说，“我们正在直接为 Java 源代码提供一个模型界面”。

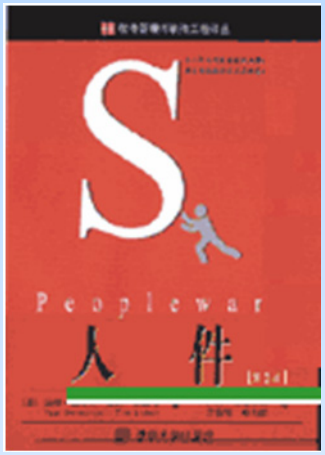
Sun ONE Studio 的市场经理 Jeff Anders 说，“Java 开发者一直在询问不止是能够编辑、编译、调试的工具，他们需要在 IDE 中整合 UML 建模功能”。

Embarcadero 以前曾经为 Sun 的 Forte for Java 提供 UML 支持。

Describe 6.0 的其他特性包括对 OMG XMI (XML Metadata Interchange)的支持，以和其他建模工具交换数据，还包含 UML 2.0 的顺序图。

（自 infoworld，think 摘译，仅供学习交流，不得转载用于商业用途）

《人件》



Tom Demarco，Tim Lister

翻译：UMLChina 方春旭、叶向群

老板，烧毁这本书，别让开发人员看到

Frederick P. Brooks, Jr--《人月神话》20 周年纪念版第 19 章

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品，特别是一些组织，开始认识到软件开发问题与程序语言、软件工程方法、软件过程成熟度无关。正如 Bill Clinton 所言，“是人，笨蛋！”当人件第一版出版时，我写了一份评论，**“我强烈推荐你买一份人件给你或你的老板，如果你是一个老板，那么为你部门的每一个人买一份，并给自己买一份”**。这建议在 12 年后依然有效，并且更加热烈。

Mark A. Herschberg

《人件》正确指出软件工程是对人，而不是针对技术。它看到在软件开发过程中人的许多方面，并指出人并不是软件开发机器中简单的小齿轮。这本书也包含涉及办公环境的章节，并提供证据说明为什么通常的观点并不适用于软件开发。**我只学了这些章节，就辞掉了原来的工作！**

中文译本即将发行！

用例：十年风雨

Alistair Cockburn 著，think 译

查看评论

用例是与外界交互时系统行为的平直叙述。我想你们中的大多数人已经使用或听说过用例。其中一些人可能还听说过关于用例实际上是怎样有用（或怎样无用）的热烈争论。

在过去的十年间，我们对用例的理解已经向前迈进了一大步。本文的目标是为了说明：了解这些发展能让你了解现在人们使用、滥用、或者只是误解用例的方法，以及你自己如何更好地使用用例。为了展示发展的阶段，我把文章分成 4 幕来讨论：史前（比十年前还早），过去十年，现在，未来。

第 1 幕——史前

Ivar Jacobson 在 1967 年定义爱立信 AXE 系统的架构时开始书写使用场景（usage scenarios），这些使用场景是非正式和粗糙的，目的是大体描述 AXE 系统是如何工作的。使用场景和平时人们写的大型正式结构并不是很象（在这上面也有我的贡献，惭愧…）.

在 1980 年代中期，Jacobson 花了很多精力来思考过去十多年的工作方法。他造了一个瑞典语术语 anvendningsfall，大意是“使用情况”（situation of usage）或“用况”（usage case）（译注：现在，“用例”这个中文译法已经被广泛接受，国内现在已出版的数十本 UML 相关书籍中，只有机械工业出版社的《UML 用户指南》和《应用 UML 和模式》使用“用况”，所以本文的翻译也使用“用例”）。当用英文出版时，他发现“usage case”在英语里说不通，所以写作“use case”。如果你不关心“use case”的来源，只需要为每次不用说“usage case”或“anvendningsfall”感到高兴就是了。

Jacobson 最初使用场景时故意写成非正式的。人们过去——现在还是一样——不愿意写正式的东西。事实上，有一次我向 Jacobson 询问用例的正式模型时，他答复，“我的确有用例的正式模型，但唯一的问题时没有人想要用它。”

但是，用例保持完全非正式导致了其他困难。人们问，“什么是真正的用例？”“我怎样才能知道我做对了？”“我怎样才能知道什么时候我做完了？”“用例很多时，怎样把它们关联起来？”

1992 年，在为 IBM 顾问组构建面向对象的开发流程时，我偶然发现了 Jacobson 发表于 1987 年的文章。象很多人一样，我认识到，这些系统范围行为的叙述自然地补充了作为面向对象设计成果的面向组件的内部叙述。所以我，和其他很多人一样，开始探索“什么是用例”。我也曾经陷入过“太正式”、“太不正式”的困境，案例太少，很难发现一条舒服的中间道路。Jacobson 坚持出版用例方面的书籍和文章，但不知何故混淆也延续了下来。

第 2 幕——过去十年

在 1992 至 1997 年间，大多数人在写用例时避免说这些东西〔是〕什么。虽然他们声称用例对项目是如何的有用。尽管看起来没有人能够说出一个用例是什么，或者说清用例和场景的区别，那些基本的、吸引人的想法还是保留了下来：书写短小的、基于文本的、关于系统如何与周围事物交互的叙述，这些交互为用户之一提供一个价值，同时捕获出现错误时的系统行为。

这个想法过去有用，现在也有用，不管文本的书写是正式的还是非正式的。

在此过程中形成的不同流派中，人们建议和推荐了下面这些基本问题答案的几乎所有排列组合。

- 一个用例是一项需求还是只是一个故事？
- 场景是否只是用例的另一个名字？
- 用例的结构是正式的、非正式的、还是半正式的？
- 用例是否有关联结构，还是只是堆在一起？

对某些人来说，用例只是场景的另一个名字，是某人使用系统的简短描述（Martin Fowler 经常开玩笑说，“我想用例就是瑞典语中的‘场景’”），出于这种想法，一个用例没有内部结构，用例的集合也只是堆放在一起。用这种方法书写用例的人们从他们的努力中获得了很好的回报，他们中的许多人现在依然推荐书写这种非正式的用例。

其他人，特别是研究者和 CASE 工具设计者，认为这些非正式的工作产品是不完备的，需要 CASE 工具中的数学基础和支持。他们发明语言，标记，和工具来使用例变得“严格”。这个流派进展并不顺利。人们需要一种比较非正式的媒介来表达他们对系统的早期想法。他们只是不想使用正式的工具。他们不想感觉到在项目生命期中付出更多的劳动。

形式主义者们面临的另一个问题是处理系统必须处理的所有变化。曾经有人问我，“对一个糖果机，我如何说明一个人可以塞进去 3 枚 25 美分的硬币，或者是 15 枚 5 分的硬币，或者是 10 枚 5 分的硬币再加上 1 枚 25 美分的硬币？或者是其他很多种可塞入硬币的组合方法？”

当时我的回答是，现在依然是，“只要写“人投钱进去”就可以了。”

多次陷入太正式和太不正式的困境之后，我选择了一条中间路线：半正式的结构中的半正式文本。使用这些半正式的结构，我们可以：

- 断言用例确实是需求并且需要一个基本的结构，同时，
- 允许人们在需要的时候书写任何想写的东西。

令人惊奇的是，第二个目标比第一个更重要。

这是我最终得到的半正式结构，它允许人们处理两个相互矛盾的目标。

使用目标组织用例

用例是 actor 动作的文字陈述，这很容易理解。1990 年代早期的大问题是：是什么把这些动作联接起来？

近距离聆听了 Ivar Jacobson 的解释和看了他的例子之后，我发现了线索。第一条也是最重要的一条线索是用例描述了一个 actor 试图使用系统达到一个目标。即，如果我们指定其中一个 actor 作为*主要 actor*，所有动作都与 actor 达到他/她/它自己感兴趣的目标有关。

把用例和 actor 的目标连结在一起如此重要，因为它把书写者的注意力从程序员关心的功能列表转移，回到用户身上：用户真正需要使用这个软件来完成什么是他们使用这个软件时的目标。如果软件支持这些目标，软件将产生最大的商业价值。

第二条线索是目标有时会失败。无论是签一份合同，从 ATM 提取现金，把 ATM 卡插入读卡器或甚至只是在填写一张在线表单时移动到下一个字段，任何目标都会失败。思考并写出需求文档中的失败处理，能够在项目过程中为设计团队节省一大笔钱。

因此，一个用例的结构分为两部分：每件事都顺利进行时的事件序列，随后是各种目标和子目标失败时的不同小事件序列的描述。

即使有了这些线索，带着目标工作的结果还是有意外。目标有不同的尺度。

在某个时候我的目标可能是赢得一份商业合同。一个更直接的目标（时间跨度更短）是带客户去午餐。更进一步直接的目标（时间跨度又更短）是为午餐准备一些现金。这个目标又带来一个更小的目标：从 ATM 提取现金。在提取现金的时间里的一小段，我的目标是把 ATM 卡插入 ATM 读卡器以便开始取钱的事务。

所有这些目标在同一时间生效。达到（不能达到）它们中的一个都可以被描述成一个用例。重要的领悟是认识到我们不能用捕获目标的相对尺度来标记用例，否则会使读者产生混乱。很多人在写用例时会工作在多种多样的不同的目标层次。如果他们在早期就在用例文本中说明了目标层次，就可以提醒读者：用例书写者在想什么。这减少了许多误解。

对于 ATM 的行为来说，赢得一份合同是非常高的层次，或者说是“高入云间”类型的目标。从 ATM 获取现金是一个尺度非常自然的目标。把卡插入 ATM 是“泥底下”类型的目标。每一个都有资格成为真正的目标，也值得书写一个自己的用例，标记出目标的层次来提示用例的读者，希望讨论哪些东西。

当书写一个用例时，我们把目标层次的想法和目标完成/失败的想法结合起来。我们在用例中书写每一步行动步骤作为更小的成功目标。我们写，“顾客把卡插入 ATM”，既和低层次目标“插入卡”相关，假设它成功了。往下走是备选或扩展的部分，我们写如果行动步骤失败了将会发生什么（“卡堵在槽里：ATM 机关闭并通知维修公司”）。

加入涉众和利益

Actor 和目标在很长一段时间内是令人满意的，但终于在某个特定的地方露出了缺点。为什么我们要在用例中书写外部不可见的行为？为什么我们要在给钱之前记录“银行检查客户余额”，或“最后记录事务的日志”？

我们以前的答案一直是，“噢，这是一份需求文档，如果你不写下这些，你就得不到好的需求。”虽然这个答案可行，但不够令人满意。

有一天我注意到计算机的动作加强了涉众之间的契约，每一个动作既是为了维护这个涉众的利益，也是为了维护其他涉众的利益。

主要 actor 只是涉众之一。拥有这个系统的组织是另一个涉众，政府的管理部门可能是另一个涉众，可能还有其他涉众。

这里是一个例子。Jim 走向糖果机。他实际上的目的不是花钱，他真实的目的是想要吃糖，如果糖果是免费的，他会更高兴。糖果机的主人的立场相反：实际上她的目的不是为了发放糖果，如果人们只是往机器里放钱而不要糖她会更高兴。糖果机加强一个简单的契约：Jim 放入钱，机器的主人看着他得到糖果。

在保险业务中的相应契约更加复杂。保单所有人出现了意外；受益人提出索赔；保险公司在允许的范围内付出最小的金钱；保险部门检查事务的操作在联邦政策之内。计算机系统保证所有涉众的利益都被满足。用例的书写就是描述这些是如何完成的。

涉众和利益模型填补了 actor 和目标模型的空白。计算机执行的每一个动作和涉众的利益相关。系统接收信息，根据业务规则验证输入值，更新内部状态，或输出数据。主要 actor 和辅助 actor 都照顾到了，就像所有其他在系统操作时不出场的涉众一样。

一位顾问描述他和他的小组第一次为一个系统列出涉众和涉众利益时的情况，此系统最近刚刚交付。他们突然发现大多数变化需求在项目的第一年早就应该产生！换句话说，他们只要在早期为每个用例简单地写下涉众和利益，他们将能够避免各种类型的错误，这些错误就不会等到移交系统以后才显露出来。其他人也报告了类似的经验。一些人说他们在列出涉众和利益时发现了价值，不管他们捕获的需求是如何的随便和非正式。

在这个时候（1997 年），看起来我们的用例理论已经完备了。

反对用例的声音

以上模型被接受的时候，自然也出现了反对的声音。对继续探寻一个完美的正式模型的形式主义者来说，用例依然不是正式的。同时，对另外一些人来说，它又太正式了，需要写太多东西。这些人认为用例是非正式、非结构化的，不是需求。后面这帮人的反应有三种：特性列表（feature lists），故事卡片（story cards）和 task cases。

单个用例可以穿越不同人的编程边界，所以单个程序员的工作分配不是明确可见的。因此一些程序员要求可以被分配到单个程序员的“特性列表”。即使提取和维护这些列表花了很多工作量，使团队的维护负担加倍，它们依然经常被请求。

极限编程（XP）的作者坚持非正式场景的思想，根本不使用任何正式的结构。Kent Beck 创造了术语“用户故事”（*user story*）来描述此类需求。一个用户故事只是包括一些简短的语句或一些写在索引卡上的句子，用于声明用户所需要的东西。在 XP 中，用户故事不用作需求说明，而是作为将来谈话的航标。因此，这些卡片记录的信息只需要让程序员和客户知道将来讨论什么东西就够了。

Larry Constantine 使用场景说明的书写形式来帮助进行用户界面（UI）的设计。他发现他不需要记录系统的内部处理步骤或它与辅助 actor 的交互。他可以成功地使用非常简单的两栏式结构。左边的栏列出用户在每一步试图完成什么。右边的栏列出系统的响应。这种两列式的表提供足够的信息来让人们发明优秀的用户界面。他把它重新命名为 *task case*，避免与系统说明的用例混淆。

依然存在分歧。一些人想要保持用例简短和非正式，另一些人想要让用例变得详细，给出概要模版和自动链接至其他系统文档。在这些不同流派的用例书写者中，除了使用某些叫用例的东西来捕获行为需求之外，他们之间的共同点不多。

第 3 幕——现在

分歧让我们来到现在，鉴于 Jacobson 说过的他有正式的模型但人们不想跟随。从用例的发展中，你应该注意这四条关键的忠告。

准备多种格式

用例不只有一种模版。一些团队使用短的用例格式用得很好，这种格式中，主要的故事展示在一段简单的叙述中，失败处理在随后的几个段落中。其他团队则需要详细的模版和仔细的书写规范。

在“*Writing Effective Use Cases*”（译注：此书英文影印版和中文版《编写有效用例》已经由机械工业出版社出版）中，我命名了书写用例的三种详细程度：简短的、非正式的、正式的。

简短格式用两到四句话来概述用例。可以放在电子表格的一个单元格中，允许电子表格中的其他栏记录业务优先级、技术复杂度、发布版本号和其他计划信息。

非正式格式包括一些文本段落，涵盖以上所提到的条目。

完全正式格式的用例的特点是使用长的模版，模版上有涉众、最小保证、业务规则、性能约束等。

由于业界的两种力量，三种格式不可能融合。第一，在用例书写中，多种项目情况和特点总是需要不同的格式。第二，野心勃勃的人们通过制造一些新东西来成名。一旦一个主题稳定下来了，就会有人去寻找能让他进思想名人堂的替代品。

要点是：总是需要多种需求形式和多种级别的格式。

只在某种形式适当的时候使用它

用例格式实际上只是一种书写的格式化方法，一种短文的形式有两部分。第一部分描述一个基本的场景，包括动作和交互。第二个部分给出一组场景，描述不同的环境下的不同行为。这种书写格式可以在任何时候描述有变体的行为，例如黑盒需求，业务流程，系统设计说明。

作为系统的黑盒需求，用例描述：用户做这个，系统做那个；系统和另一个系统交流；某些事情出错了；系统现在做这些作为替代；等等。它清楚地说明了系统必须完成什么，但没有说系统如何做，也并没有假设系统使用面向对象方法（或其它方法）来设计。

作为业务流程，用例描述：顾客这样做，职员那样做，职员把某些东西交给另一个部门，另一个部门做一些别的事情，等等。但事情出错时，某些部门退回一些工作，这些工作转到另一个不同的部门，等等。这些类别的说明对业务人员来说相对容易书写，未受过训练的人也容易读懂，所以在业务流程重组中的使用正在日益增长。

作为设计文档，用例描述：用户做这个；第一个组件做一些事；该组件转发一个请求给第二个组件；等等。在不同条件下，组件之间的顺序是不同的，等等。用例很少用在设计文档中，因为有太多种其他方法来写设计文档。

要谨记的事情是用例的格式，也就是，一个行为场景紧跟着备选行为的场景，以适合某些书写要求。如果某种格式适合你，就使用用例，否则，如果不符合需要，把它们丢到一边，不要用这种格式。

了解用例的局限

用例不关心系统设计、界面设计、特性列表或测试，即使许多人想要用例这么做。当然，用例不应该关心这些。用例应该描述行为需求，但这并不能阻止人们寻找“自动”从用例获取设计的方法。用例不说明设计，使这些人在悲伤和欢喜之间徘徊。他们悲伤，因为他们希望得到简化软件开发的魔幻公式。他们高兴，因为他们认识到对黑盒说明的需要。

即使有用例的格式允许，用例也不应该用于描述界面设计。有几个原因，这里是其中两个：首先，用例是指一种需求文档，而界面设计是设计，由受过训练的人们在被告知系统该做什么之后创建。其次，界面设计是脆弱的、易变的。在用例中书写界面设计意味着文档不得不经常修改，远远超出了一份需求文档应有的修改频率，也远远超出了项目团队（我碰到过的）所能维护的。其他方式的说明，如状态图、Constantine 的 task cases，还有屏幕快照，都在用户界面方面做得更好。

用例和特性列表存在一个基本的不匹配。同样的系统特性可能在多个用例中作为单个条目表示。结果，一些人不得不把用例文本转化成编程任务。对很多人来说，包括我自己，这不是一个问题。他们只需在头脑中、在谈话中或注释中完成这个转化就可以了。这种工作方法避免了双重维护问题。任何既需要用例又需要特性列表的人是在寻找双倍的工作量。

用例不是测试计划或测试用例。用例没有测试用例需要的数据值。但是，既然用例说的是在不同的条件下应该发生什么，因此用例是测试流程的优秀输入。测试员还是必须创建真实的测试用例来和用例匹配。

避免用例的标准错误

用例书写者会犯很多错误。对项目来说，两个最常见也最昂贵的错误包括太多细节和包括用户界面说明。两种错误都使得用例变长，难以阅读，脆弱。

无论教师如何经常警告不要在需求文档中包括界面说明，新人还是很想描述“OK”按钮、特定字段，屏幕，屏幕序列。这样做的原因是人们喜欢具体的概念。虽然事实如此，用例不适合做这个。用例要求你付出努力，去学习如何用技术中性的方法来书写。例如，“系统显示选项。用户选择一项活动”。这是值得的，你得到的用例更短、更易读、更健壮。

如果用例的主场景短，许多人似乎会感到羞愧，所以他们把它伸展成十五甚至三十五行的长度。以我个人来说，我还没有写过一个主场景长于九步的用例。九不是一个魔幻数，只是说我在适当的层次上得到子目标和移除设计说明时，任务就少于九步了。用例有时可以短到主场景只有三步。

用例的最大价值不在于主场景，而是在于备选行为。主场景可能只占用例长度的 1/4 到 1/10，因为需要处理的备选情况太多了。如果主场景有三十五步那么长，用例会伸展到十页纸，这太难阅读和理解了。如果主场景在三到九步之间，整个用例的长度就会只有两到三页，当然，这也够长了。

如果你可以避免在用例中包含太多细节和用户界面说明，你的用例会更容易阅读，而可读的用例会真正地得到阅读。冗长难读的用例只会令人厌恶，在以后的项目生命期中被抛到一边。

第 4 幕——不远的将来

用例已经到达了一个静止点，这意味着两件事：这是一种有效的模型，可以用于教学和实践；分裂的进化已经开始了。

对于这点我们可以预言不远的将来的一些事情。

1. 作为一种主流技术，用例将在本科课程中、全世界的标准软件开发和方法学课程中被讲授。

2. 工具厂商和理论家将继续寻求用例所隐含的形式主义。

3. 工具厂商将生产和理论匹配得更好的工具，使交叉关联的用例更容易书写、维护和集成到 CASE 工具套件中。

4. 人们将继续寻找这项当前主流技术的替代品，然后建议各种不用用例来进行软件开发的方法。

5. 一些事情依然不变：人们继续在他们的用例中书写太多的界面细节，不管他们的工具和模版是什么；公司将继续使用用例来避免人们面对面交流。

6. 人们将误用用例，不正确地教授用例，曲解用例，然后把随之而来的混乱归结到用例身上。

为了保证用例有用，你所能做的是学习：

- 如何把设计说明排除在用例之外；
- 在用例书写中何时使用不同级别的用例格式
- 何时使用另一种格式来表示，甚至象两栏式表格那样简单的格式。

然后，就可以坐下来观看四处横飞的争吵了。

关于作者

Alistair Cockburn（发音 Co-burn）是“*Writing Effective Use Cases*”的作者，敏捷软件开发运动的发起人之一，“*Agile Software Development*”（编注：中文版《敏捷软件开发》即将由人民邮电出版社发行，译者为 UMLChina 俞涓）的作者。有关他的工作的文章放在 http://Alistair.Cockburn.us。

网址

www.UseCases.org

www.pols.co.uk/UseCaseZone

www.ForUse.com

midwatch.org/design/ucindex.html

书籍

Armour, F., Miller, G., *Advanced Use Case Modeling: Software Systems*, Addison Wesley, 2000.

Cockburn, A., *Writing Effective Use Cases*, Addison-Wesley, Boston, MA, 2000.

Constantine, L., Lockwood, L., *Design for Use*, Addison Wesley, 1999.

Jacobson, I., *The Object Advantage：Business Process Reengineering With Object Technology*, Addison Wesley, 1995.

Kulak, D., Guiney, E., *Use Cases: Requirements in Context*, Addison Wesley, 2000.

文章

Jacobson, I., "Object oriented development in an industrial environment," OOPSLA '87: Object-Oriented Programming Systems, Languages and Applications. ACM SIGPLAN, pp. 183-191.

Cockburn, A., "Structuring Use Cases with Goals," *Journal of Object-Oriented Programming*, Sep-Oct 1997 and Nov-Dec 1997, online at http://members.aol.com/acockburn/papers/usecases.htm

英文原文发表于 STQE 杂志 2002 年 3/4 月期。

UMLChina 公开课

“UML 应用实作细节”

（2003 年 3 月 1-2 日，北京）

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

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

本课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，阐述如何（只）使用 UML 里的三个关键要素：用例、类、顺序图来完成软件开发。使学员自然领会 OOAD/UML 的思想和技术，并对实践中的误区一一指正。整个过程简单实用，简单到甚至可以只有一个文档，非常适合中小团队。

详情请见>>

通往职业初段—使用用例组织需求（上）

think 著

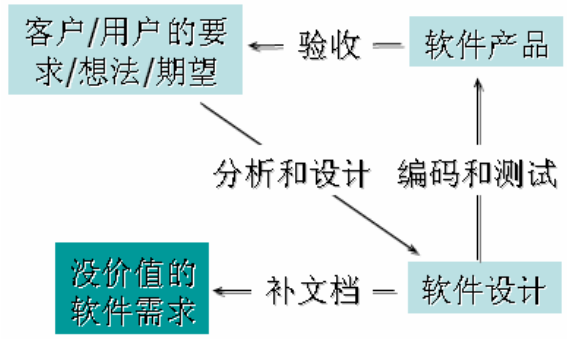
查看评论

在“前言”（见《非程序员》21 期）中，我们给出了一个开发的过程图。图中的第一步，把目光焦点放在系统的边界上，目的是得到软件需求。

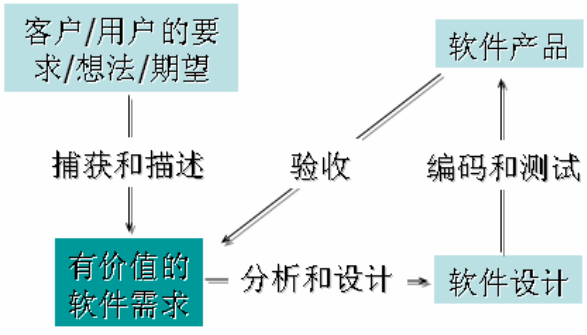
需求是最能省钱的地方。软件项目中 40-60%的问题根源在于需求阶段，而这些问题放到产品发布后再改正，比在需求阶段就改正，需要多付出 68-200 倍的成本【1】。以前说软件开发技术，大多数公司主要注意的是实现阶段的技术，需求阶段草草了事的居多。但是随着软件业的发展，现在很多公司也已经注意到了需求技术的重要。

需求也需要开发

我们先来看看没有“开发需求”步骤的软件开发过程：



客户或用户的要求/想法/期望，草草地记录下来，然后在此基础上进行分析、设计、编码、测试，最终的验收也依赖于此，而这些未经提炼的要求/想法/期望是脆弱、易变的，后果可想而知。更为合理的步骤如下：



客户或用户的要求/想法/期望，经过“开发”，形成有价值的软件需求，为后续的开发步骤打下坚实的基础。【2】如何能得到有价值的软件需求？软件的需求可以分三类：功能需求、非功能需求、设计约束。采用什么方法来组织这些需求呢？

以用例为核心组织需求

以用例为核心组织需求－－所有需求，最终的需求文档可以简单到只有一个用例文档。

伴随着 UML 到来的“用例”，为需求技术在国内的普及起到了至关重要的作用。目前，“用例”是 UML 中讨论得最多的话题之一，从 UMLChina 讨论组的帖子可以统计出。用例是什么？用例是文档还是图？用例的粒度如何把握….都是经常被讨论的话题。

用例是什么

1. 用例是一种需求技术【3】

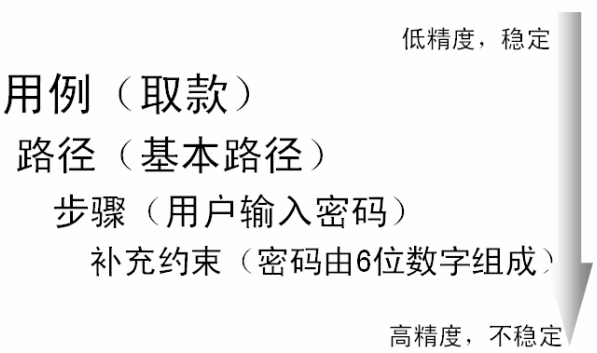
“是一种”也就意味着“还有另一种”，象 IEEE 的标准，功能需求的书写方式为“系统应…”

2. 用例是基于用户目标的需求组织技术

用例把“系统应能够接受用户录入取款金额”，“系统应能够从帐户中扣除取款金额”等需求组织到一个场景中，而整个场景的描述采用的是用户的角度。交互结束后，用户达到了一个目标。

3. 用例是有层次的需求组织技术

通过用例这种需求技术，需求就可以按照“目标－路径－步骤－约束”四个层次组织起来【4】，如下图：



这样，需求就按照不同的精度和稳定性分出了层次。“取款”这个需求精度很低，但很稳定；“用户输入密码”精度高了，稳定性下降了，而绑定到这条需求的“密码由 6 位数字组成”稳定性就更低。分层的意义在于可以分而治之，某层需求的变更不会影响到上一层。

应该怎么做——步骤

使用用例组织需求共分为三个步骤：

1. **识别 Actor【5】;**
2. **识别用例;**
3. **书写用例文档。**

识别 Actor

理解 Actor

RUP 的定义：在系统外部与系统进行交互的人或物。

有必要添加一些解释来帮助理解这个定义。

（1）系统外

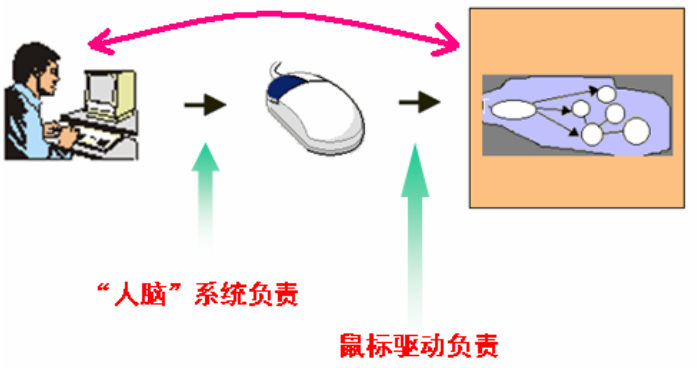
系统外就是指系统边界之外。系统边界界定了讨论问题的范围。Actor 的确定就代表了系统边界的确定。位于系统边界内的，最终要通过程序员的劳动变成代码，系统外的则不是。Actor 说的是系统之外的事物，有时这种事物在系统内会有对应物，但不可以将二者混为一谈。拿 RUP“指南：主角”一章的订票例子来说，旅行者来到旅行社，告诉旅行社工作人员买什么样的票，工作人员使用计算机系统出票。旅行社工作人员是系统的 Actor，可能系统也要保存每一位工作人员的信息，如果用面向对象的方法来分析设计，系统内可能会有一个“工作人员”类。这两个是否相干？Actor 是不是类？外面有 Actor，里面是否要有类？在这里强调：Actor 代表的是系统之外的真实事物（虽然也有角色抽象），识别 Actor 的最终目的是通过它探讨对系统的使用情况，至于位于系统边界内部的设计是否有同名的概念，不关它的事。

（2）交互

Actor 和系统要有交互。还是上面的例子：旅行者没有和系统交互，不是系统的 Actor；工作人员和系统有交互，工作人员是系统的 Actor。这个一般都能理解了。

再进一步：假设工作人员使用鼠标，鼠标也是系统外（造鼠标不属于系统责任）的，而且离系统更近，交互更直接，那么鼠标是不是 Actor 呢？打印机算不算？如果系统是 B/S 结构，工作人员使用的浏览器算不算 Actor？

所以要强调“有意义的交互”，亦即这些交互是目标系统关心的，也是最终要负责（负责的意思就是要变成相应代码）的。人脑里的信号变成手部动作传给鼠标，操作系统（或鼠标驱动）把传入的鼠标信号变成屏幕动作，负责这些交互的分别是：人脑、操作系统厂商（或鼠标厂商）。对目标系统来说，应该关心的是工作人员和目标系统之间的交互，如“工作人员输入目的地”。【6】



（3）人或物

这个可能读者都知道了，可以是人、硬件、外系统…，对于一些没有外部引发者的事件，还可以归为“时间”这样一个外部 Actor。【7】

识别 Actor 的思路

（1）一般的识别思路，现在出版的各种 UML 教材上基本上都有说明，如思考“谁使用系统的主要功能”…等等。这里就不多说了。

（2）如果前面有业务建模的步骤，更通常的是从业务流程中得到。例如：为某 LT 公司提供增值业务的某 A 公司的 A 系统。如果在业务用例“用户：开户”中有以下步骤：用户到营业厅申请新号码并选择 A 公司增值业务；营业员录入用户信息；LT 系统把申请了 A 公司增值业务的用户资料传送给 A 系统，A 公司的操作员….从业务流程的描述中，就可以知道 A 系统扮演的是什么样的角色，直接和 A 系统交互的是谁/什么。

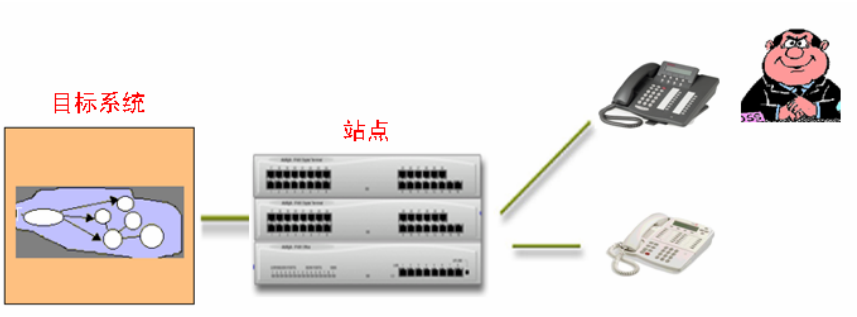
（3）一开始宁多勿少。经常会碰到到底要识别多少个 Actor 的问题。例如象 Foxmail 这样的软件，要不要分发信者和收信者？答案：无所谓，不丢用例就行。不管你是识别了“发信人”、“收信人”，还是只识别了一个 Actor“用户”，他们的用例都是一样的：发信，收信。如果你分不清到底是一个还是多个，就按多个处理好了，最终再清理，这样不会出问题。如果少了 Actor 的话，可能就会丢了用例了。

（4）识别 Actor 的工作过程中，最容易出现的问题是边界理不清。只要注意理清边界，提醒正在讨论的是什么东西，一般别的问题都不大。

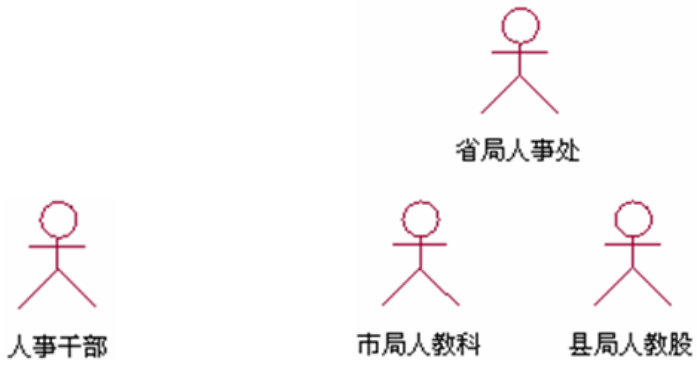
几个实际工作中的问题请读者思考：

以下涉及到名称时，真实名字隐去，图形为说明问题所画，并非实际项目中的图形。

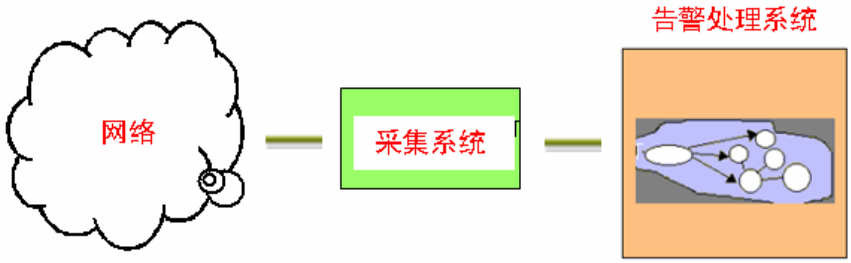
1. A 公司要开发的智能排队服务系统 NTTQ，用在一个基于 IP 的呼叫中心系统中，目的是通过更先进的分配算法，解决分配效率不高、丢电话的问题。对于 NTTQ 来说，它的 Actor 是？



2. 某人事系统，各级都有使用系统的人事干部，到底是设一个 Actor“人事干部”（见下图左），还是设三个 Actor（见下图右）？你认为怎么处理好？



3. 某网络告警处理系统，前端有采集系统不断采集数据，出现异常时，向目标系统发送告警，目标系统分析告警信息后作出相应处理。对目标系统而言，采集系统是 Actor。但也有开发人员提出，什么时候会有告警信息，是无法预计的，是否时间也应该是一个 Actor？



下期杂志将刊登：通往职业初段—使用用例组织需求（中）

【1】Karl E. Wiegers 著，陆丽娜、王忠民等译，《软件需求》，机械工业出版社，2000/7

【2】本段落的图和文字参考了 Rational 尤克滨的讲稿。

【3】用例技术同样适用于组织业务流程，我们先以软件需求为例说明用例技术，相同的技术也可以用于组织业务流程。

【4】约束也可能针对路径、用例、甚至整个系统。

【5】这里所说的 Actor，如果不加说明，都指系统 Actor。关于 Actor，目前没有象“用例”那样明显占上风的中文译法，所以本文 Actor 保持英文。预计最终占上风的是某种两个字的译法。

【6】读者可以自己想一想：开发什么样的软件，鼠标可能会成为 Actor?

【7】经常有人这样做：Timer→出报表。实际上这里的 Timer 应该改为时间，Timer 是一种内部设计，时间是系统外部的事物，可以设想成一个挂在天上的大钟。

征稿

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

UML 与 XP

Alan Cameron Wills 著，刘巍 译

查看评论

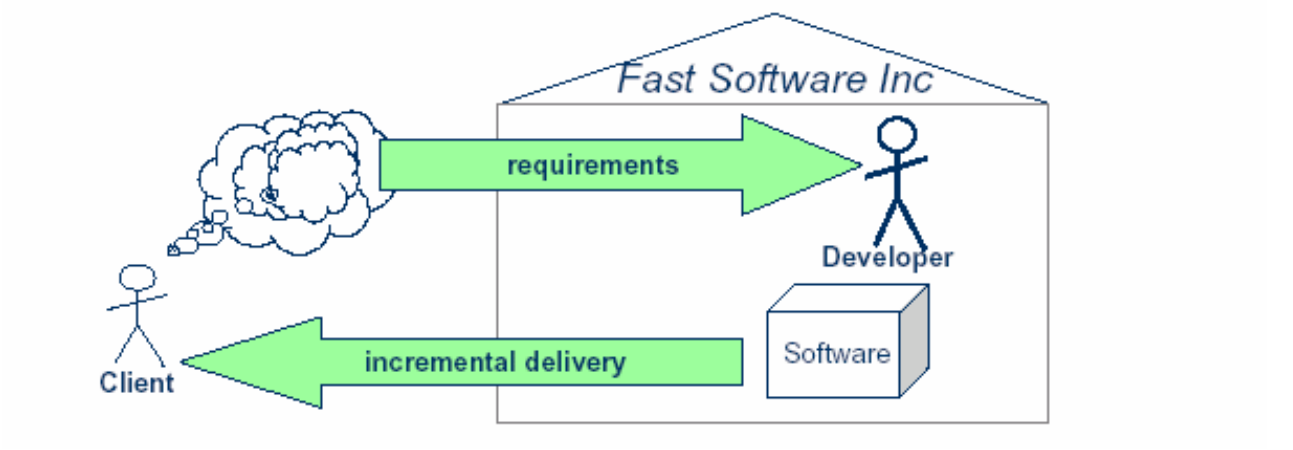
我曾是一个 JFDI 开发方法的坚定信徒和使用者，这种方法可以产生可信的程序，可以在设计过程的中早期发现问题并且发现对需求的任何误解，还能容易地跟踪需求发生的变化。我赞赏极限编程（XP）[Beck]所阐明的这种方法的一些区别于其他迭代开发的优秀指导原则。

可是我也坚信在设计中要用模型去理解抽象意义，并在设计中和之后用模型来进行交流。极限编程的一个我不赞同的方面是通过代码和测试来体现设计而轻视文档和模型。然而，不可否认任何脱离开代码和测试的文档将很难与之同步并很难读得懂。

这篇文章的观点概要描述了我在做很多客户的方法指导顾问时的一些经验，试图同时获得极限编程方法和建模技术两方面的好处。

极限编程（XP）能带给我们什么？

极限编程（XP）是一种严格的软件开发方法，这就意味着在开发实现前要仔细地编写软件开发规格说明，开发实现的过程要靠着规格说明来验证；可执行的测试代码被证明是最适用的规格说明。不象其他早期的严格的（软件开发）方法（如[Jones]、[Morgan]），极限编程（XP）回答了“but how do you know you’ve got the specification right（但是你如何知道规格说明的正确性）？”这个问题。



紧密的客户与开发者的循环保证了开发者对客户到底需要什么的任何误解能够很快改正过来。同样适用于客户对自己到底需要什么的任何误解。

递归测试使增量的细化开发和代码重构成为可能，不使 bug count（错误计数）失控。极限编程（XP）的开发者估计用于设计单元（和集成）测试的时间达 50%；如果测试设计得当则时间的投入可以以稳定可靠的代码的形式回报。

如果我们试图改进正统的极限编程（XP）方法，就不能失掉这些特性。

为什么我们需要的不仅仅是极限编程（XP）？

抽象思想的交流和讨论。我们大多数人的各种抽象设计思想不能仅仅是保留在头脑中而不通过一些方式写下来，比如，需求、关系、协作等。在这方面 UML 一般来说无论是在实现还是测试中比都代码强。自动测试代码可能是最实用的规格说明形式，但作为建模的媒介，其可读性不好。在我工作的中型到大型的项目中，我们发现仅用程序代码和非正式的描述作为讨论需求和系统结构的方式是行不通的，尤其是当有一支大型的、由具备不同经验的人组成的开发团队。

设计分析和开发角色的分离。一些人适于与客户交流并领会其需求；而另一些人更适于写出干净、优秀、高效的解决方案。我们发现有必要将这些角色分开。

组件接口、设计模式。任何种类的泛化都需要：

(a) 抽象和泛化的过程；

(b) 描述如何使用结果的标识方法。

在基于组件的开发中，能够描述组件接口非常重要。

UML 的角色

有很多的工具支持 UML 和程序编码之间的直接映射，可以通过参数化实现在不同的体系结构和风格下运行。设计员能有效地用 UML 编程序。

我发现这些特性非常有用，但是对开发工作来说还不足够。一些重要案例中，我想要建立不代表具体实现的模型：

- 在基于组件的开发中，将接口或连接器的定义与组件的任何具体实现区分开是很重要的，甚至将其与任何一个组件规格说明区分开。

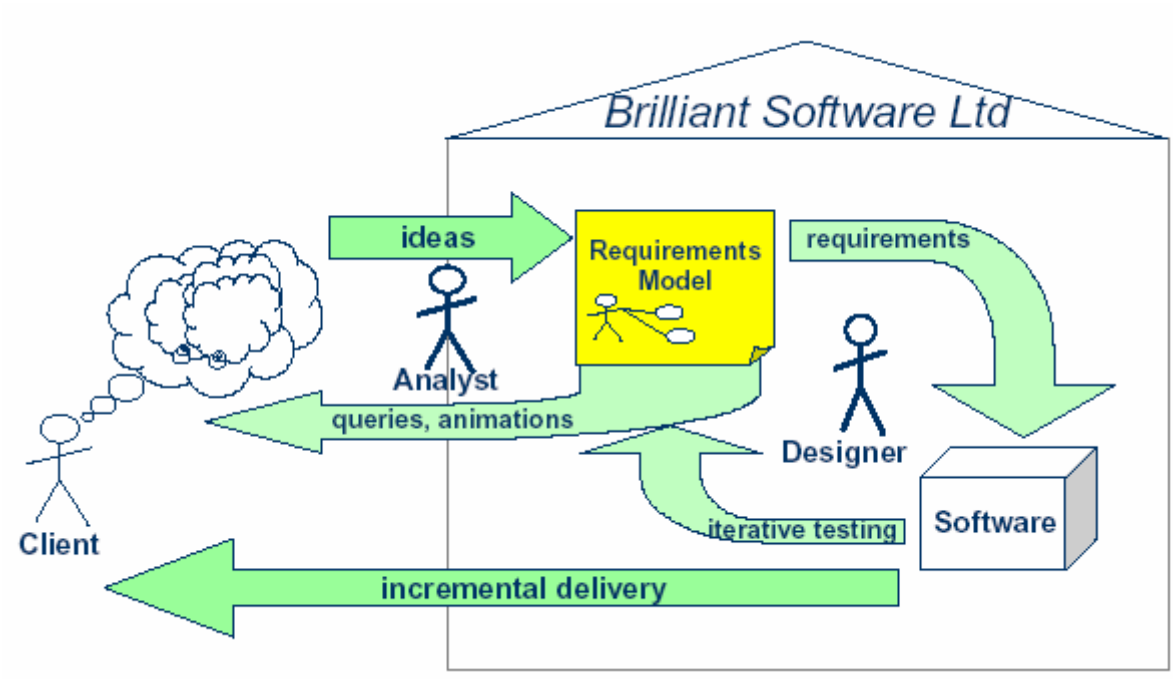
- 这同样适用在企业应用集成中：你感兴趣的多种多样的系统必须有共同的接口语言，但是其具体实现可能是非常多样的，也可能是不知道的。

- 许多实现典型地有一些并不是需要解决的问题的本质而是为了改善性能或者维护性的类。在分析中，我想坚持描述问题而不是描述如何解决问题。

因此，在我们的方法中，将 UML 的两个用途区分开来：需求模型定义了在系统或者组件外面可见的行为；实现模型被设计员用来定义系统内部的结构。需求模型不表述内部的具体实现。

在类 XP 过程中使用需求模型

在我们给客户咨询过程中，我们已经取得了一些适应上述需要的经验，同时保留了极限编程（XP）的优势。当然，其结果非常类似如 RUP 这样的其他的迭代过程。不过，我们注意到了其他方式的许多改进。现在，我们把这种方式叫做 XP/RM。



该过程介绍了一个需求模型，其中包含所有我们打算为用户所见的东西。该文件典型地将包括功能需求的 UML 模型；另外还有非功能性需求（性能、健壮性、可用性以及其他）。它将包括用例、类、业务规则和协议在内。捕获了开发者制作软件需要知道的一切。

需求模型由可能与开发者分开的系统分析员建立和维护。开发者把分析员看作他们的用户。

需求模型是所需要的系统行为的规格说明——而不是反映编码的具体实现模型。（虽然开发者可以为他们自己的目的使用实现模型。）

注意现在有两个主要的循环：增量地建立客户需求模型的分析员循环；和增量地开发满足这些需求的开发者循环。

重要的是要强调图中的箭头表现了信息流是或多或少的蠕动式连续运转的。设计员不必等到模型接近完成之前才开始开发软件。模型应该是并行地或稍稍领先于软件的开发。设计员可以和客户交流并参加一些建模的会议；分析员应该紧密地监督开发情况，在设计员面前代表着用户。

分析员循环

在极限编程（XP）中，用户和开发者之间的循环保证了开发者向所需解决方案的逐次逼近的方向，并对在开发期间里发生的需求变更负责。

在 XP/RM 中，用户和分析员之间的循环建立了一个逐次逼近的模型。好的分析员总是和他们的客户紧密连续地工作在一起；但是，我们强调：

- 需要从泛泛的草图开始到更加精确的关键细节，这样开发者就能开始工作。我们通常通过识别主要的概念和业务用例，以及只是在响应时间、安全、用户数和存储需求等非功能性需求出发。

- 应用分析技术利于暴露客户叙述的需求中不明确和不一致的内容。标准的用例分析在这一点上得分不高；我们使用多个重叠视图和轻量的交叉检验（详细下述）。目的是将问题表反馈给客户，这样就能获得更加精确的模型。

注意需求模型是内部的。我们不必期望分析员的模型作为用户合同的一部分。其目的是我们达成了一致意见，并用来向客户澄清我们提供了什么的这样一个技术文档。将模型解释给客户是分析员的部分工作，通过动画或者图板、或者通过逐项同意用户需求或者就是在建立模型时问一些问题。

设计员循环

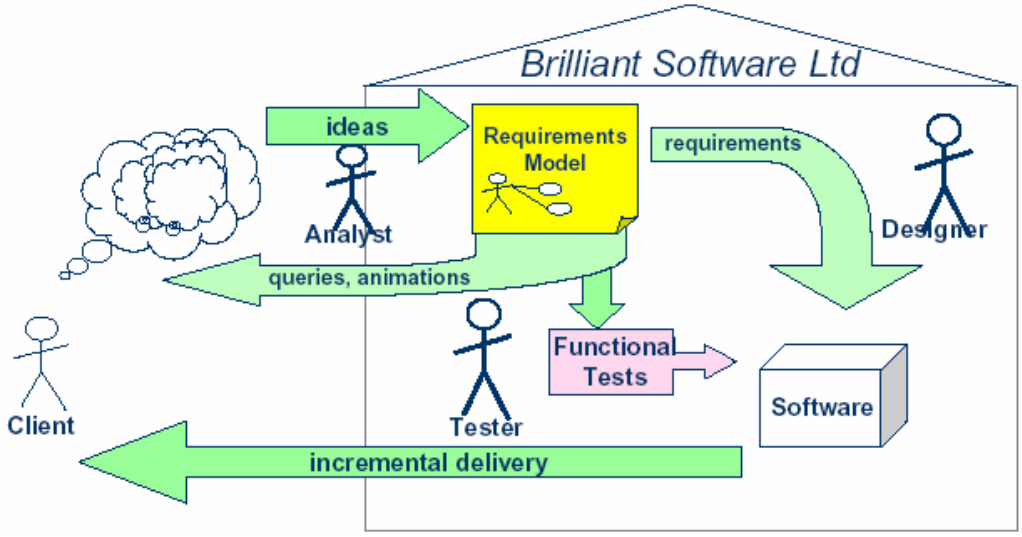
设计员从在需求模型中捕获的需求开始。设计出来的软件在开发过程中对照模型进行测试；并且在通常的关键点时演示给客户（通常大约每两周）。设计员和用户之间的互通比在极限编程（XP）中的紧密性稍少一些：结果是通过分析员得到需求。当用户实际上是一个分散的团体时这样尤其有效。

设计员可以用 UML 来做实现模型：即一个直接反映程序编码结构的模型。类似 Together、Rose 或一些别的工具都可以很好地支持这种模型。编码从图中可直接生成，反之亦然。

设计员可以用 UML 来表现实现；而且他们可以选择（或不选择）通过将需求模型做简单的扩充来设计；但是，我们的重点不在这里。

测试

依照需求模型的测试的过程被更正确地表述为从模型到测试软件的工作流。在实际的项目中，有独立的测试团队作这些。我们应该将之与开发者自己为内部结构写的单元测试区分开；而且用来测试整个系统或组件的功能测试来自需求模型。



（这个过程模型没有任何单元测试，正象它不体现任何设计过程的内部模型一样。）

前期需求模型的好处

我们到现在的经验是这个方法有很多好处：

- **快速地暴露出差距和不一致。**在建立模型时应用我们的检查技术非常早地提出关于需求的问题表：甚至比狂热的极限编程者产生原型编码更快。这样纯粹用极限编程（XP）的很多的错误理解能到很快消除。
- **高层的需求视图。**用户面对工作原型有时会使注意力无用地集中于用户界面上。向用户询问关于用例的场景、用例的目标、对象之间的关系和业务规则能得到更起作用的反馈。
- **减少需求变动。**用户经常在他们想要什么上面自相矛盾，他们可能在改变需求后再一次改变回来。原因也许是用户机构中不同的派别造成的；或者是业界的不确定性造成的。在用户和开发者之间加入分析员会减少这些大的波动。

但是，为了使 UML 前期工作结合极限编程（XP），我们必须考虑：

- 如何避免危害到极限编程（XP）对变更的响应；
- 如何在分析循环中集中注意力在正确的需求上；
- 用例和增量开发工作之间的关系；
- 测试与需求模型之间的关系。

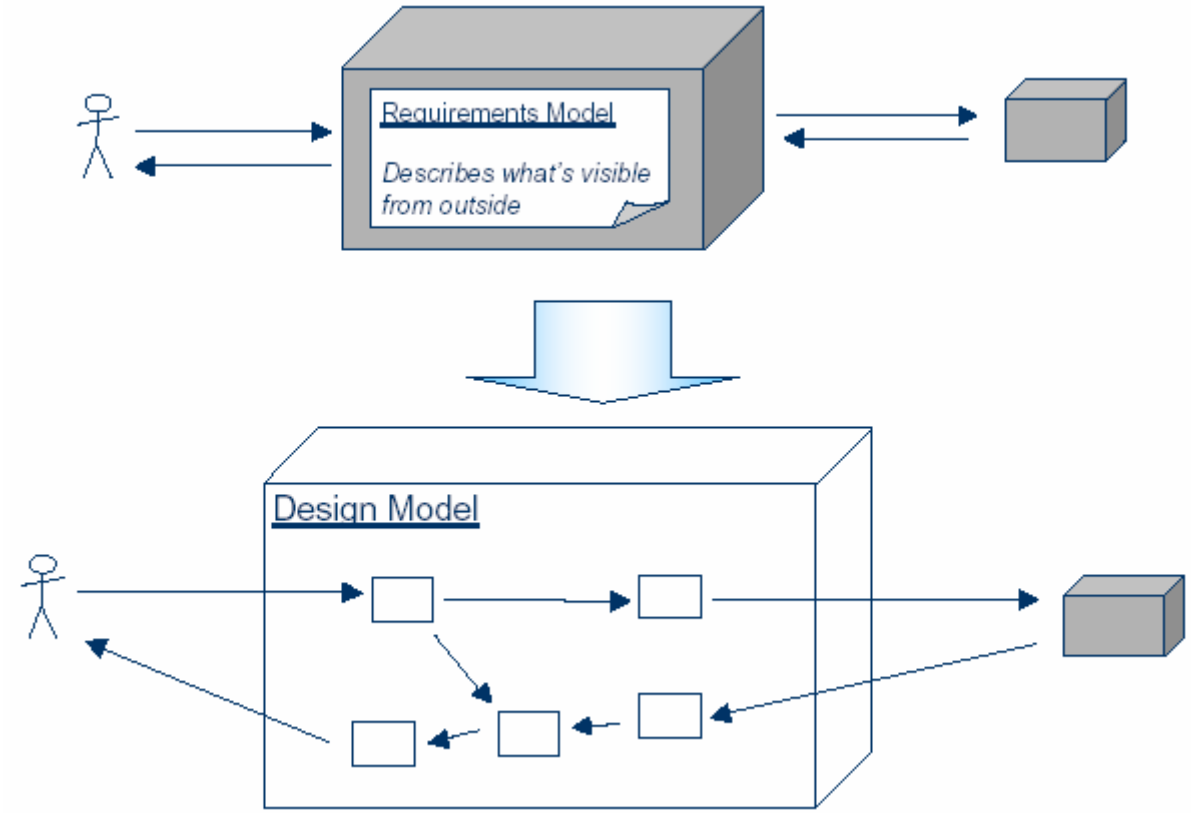
用 XP/RM 建立需求模型

需求模型中有什么？

需求模型不说明内部的结构：没有职责或者操作被分配给类（除非他们在外部接口可见），并且没有定义对象之间的协作。仅有的操作是在指定的系统或组件的外部接口可见的那些操作；

仅有的协作是被指定的组件参与的那些协作。需求模型的确要定义类，但只是那些问题域本身需要的类。

设计的过程决定具体实现如何工作，如果是面向对象的或者基于组件的解决方案，决定对象或者组件都是什么，它们各自的职责都是什么，以及它们如何协作来完成需求。



需求模型包括：

- 用例，主要地根据[Cockburn]编写；但是基本强调 postcondition（后置条件）——每个系统被要求执行操作造成的影响。这在“sea level”（在[Cockburn]在里的术语：用例在细节的水平上描述系统或者组件执行的单个事务如“下订单”或者“接受股票交接”）上尤其重要。

- 类图声明所有的用在用例 postcondition（后置条件）中的概念。类没有下述操作：在用例中描述的动作。（对类的职责的分配是设计员的工作。）类图不描述具体实现的内部的结构；它描述系统理解的概念之间的关系。

- 业务规则，用概念的约束和用例序列图的约束术语定义。

- 非功能需求，用有关类和用例的术语描述。

模型细化技术

分析员使用一套面向建立逐次逼近模型的技术或者模式，从非常简单的事物开始，并通过简单的步骤详细阐述。其中一些步骤是试探性的，通过向客户问有方向性的关于业务规则和系统需求的问题来进行调查；其他则通过检查已设计的部分暴露出的模糊和不一致的部分。

在我们的方法中，建立多个互相联系的视图。这里有多个‘垂直的’视图即不同的用户通过不同的接口来看系统或者组件：它们对于共有的概念必须有相同的意义，服从共同的叠加规则，并且对相同的事物都有一致的系统预期；我们将不深入探讨垂直视图和它们的组成。

也有多个‘侧面’：这里有不同的形式对不同种类的信息进行抽象：例如，用例、类图和状态图。我们通过一个当前在分析员中不太普及的方法将这些联系起来，这是一个发现模糊和不一致的有效方法。

为这个目的，我们有相当数量的模式。我们不是直接地复述，而只是举两三个例子。

找出名词和动词

你以前已经听过这个。名词→ 对象类型、动词→用例。我们并行地建立各类视图。给出第一个模型的近似。

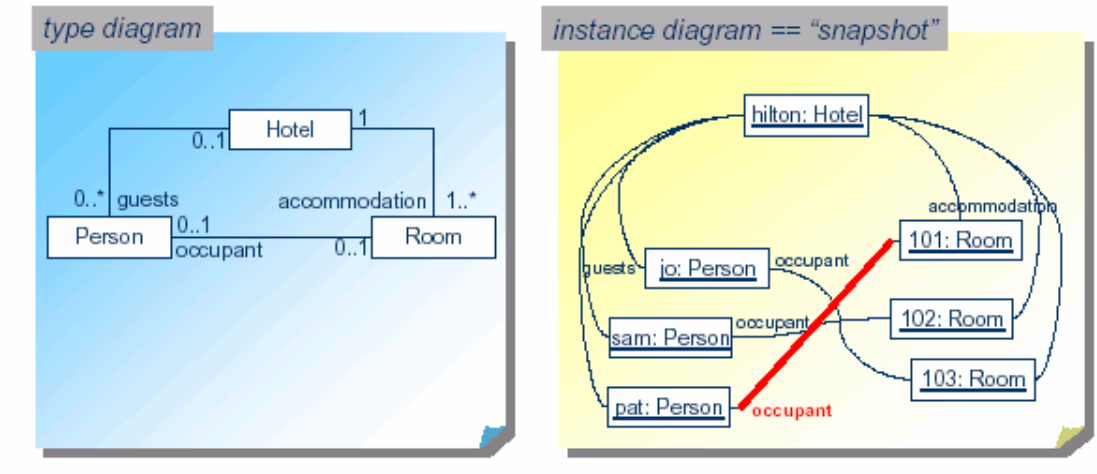
用例用类模型的术语说明

例如，如果我们有：

用例　Guest（客人）入住 Hotel（酒店）

之后　Guest（客人）现在成为 Hotel（酒店）中一个曾空置的房间的住客。

我们能预想实例图中这个用例的实例的前后情况，象征 Room（房间）、Guest（客人）的实例，他们之间的 *occupant*（入住）关系。红色的直连线（pat-101）表示后来变化的情况。



从所有用来阐明全部用例的实例图中，我们可以得到一个概括了用例模型中的概念的类型图，这足以解释它了。

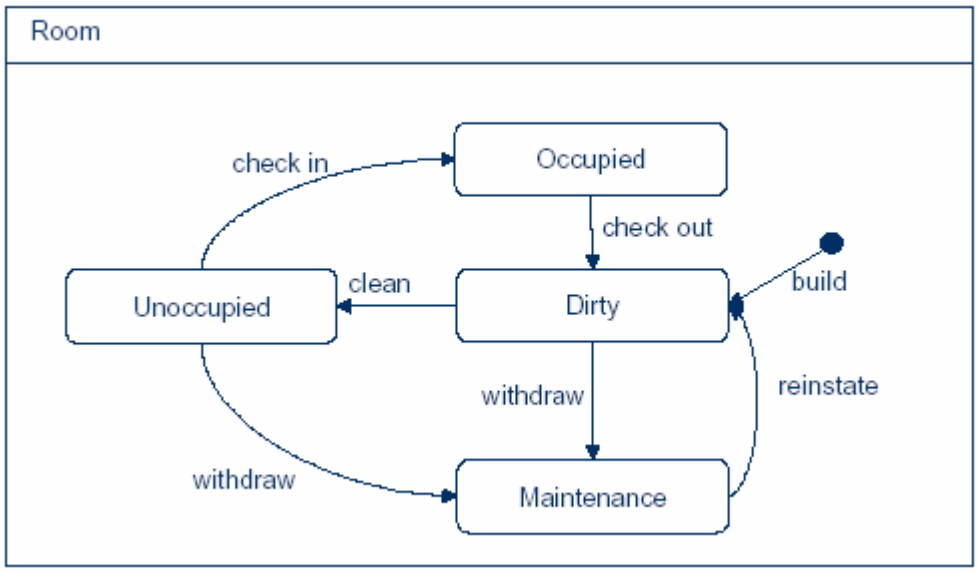
分析员可以从询问酒店中发生了什么开始写用例文档，并从中得到类型图。两件事可以平行进行。

对于我们发现的类型和关联关系，可以问很多标准问题：他们的 cardinality（一个人能占多少房间？等等）；这些对象和关系还发生了什么？检查问题的一个方法就是为选定的类型画一个状态图。

描述性的状态图

我们通过问“其中一个是怎样被建立的？”并且反复问“之后它发生了什么？”，这样就能建立一个状态图。（在需求模型中，这些是纯粹性的描述：状态只是布尔声明。）

例如：



这些转变在这里是在画状态图的过程中发现的用例。另外我们发现了一些 pre（前置的）和 postcondition（后置条件）——例如撤销（客房服务）的一个前置条件是 Dirty（房间脏）或者 Unoccupied（房间闲置）。现在我们能为每一个这些用例编写规格说明。

关联类型

任何关联如果有自己有用的属性和状态，就能被转化为一个对象，例如，我们可以建立一个 Occupancy 类型代表入住关系，然后为此画一个状态图从而发现有取消预定、未付房费、更换房间等情况。

静态约束

类型图中关联间有个循环，有一个实例是否必须、不许或者可能有一个同等的循环的问题。在上面提到类型图：如果一个 Person（例如 Jo）是 Hotel（例如 Hilton）的一个 *guest*，*Room Jo* 入住是这个 *Hotel* 必须 *accommodation*（*提供*）的一部分吗？（回答可能是不，如果 Hilton 为了处理过多的预定与邻居预先有安排的话；这取决于这个关联确切的意味什么：如果客人的连接指向你付帐单的酒店，而提供住宿的连接指向酒店——房间是其物理的一部分。）

这样那样的问题揭示了系统将必须遵循的某种业务规则。

总结：模型细化模式

象这样的模型细化技术（还有其他，我们有很多）确保可以得到对客户需求的全面且准确的理解，而不是原型引起用户对用户界面的不必要的关注。这是对纯粹的极限编程（XP）的一个相当大的改进。

在解决需求中的模糊和差距方面，需求的形式化几乎和写一个原型一样有效；但是它能由作为分析专家而不是编程专家的分析员来完成；并且可以通过不同的系统视图分开来完成，即使它们在具体的实现中是重叠的。

基于组件的开发

电子商务要求软件开发的快速响应：并且我们能达到这个目的的一个方法是通过或大或小的组件来构建软件。这在组件组成一套可互换部件——如 Java windows 系统，或者 Unix 标准库——时运作得很好。每个套件都被设计成适用于某个领域。

用这样的组件套件，我们建立了很多模型：

- 任何组件的设计。模型代表了程序编码中的类，它们的职责和互相的协作关系。
- 组件规格说明。既然可能有几个可互换设计可以符合相同的规格说明——例如存储或者排序——规格说明模型不能与任何设计相关是很必要的。
- 连接器的规格说明。如果我们想要组件能按照许多配置进行组装（就像 Lego），那么我们就需要定义少量的标准接口。在 Unix 中，管道是标准的接口；在 Java Beans 中，有事件和属性；在 Java AWT 中有窗口间的关系，并且在 Java 输入 / 输出有文件流。套件结构中的连接器越少，可获得配置的灵活性就越大。这些连接器的定义必须慎重，尤其是当第三方打算为套件建立组件时。

在前面的两个例子，模型与任何设计都没有联系，除非一个设计应该与它规定的行为保持一致。

如果组件是根据极限编程（XP）的原则开发的，规格说明应该作为测试输出。有一个观点就是测试就是最好的规格说明，并且图表以及其他都是多余的。但是，我发现测试代码使全局观变得模糊：它们很难被抽象出来并在头脑中想象。因此，我倒是希望用 UML/OCL/自然语言来写组件和连接器的规格说明；并且从那些模型进行测试。

CBD 的环境说的很明白，象程序设计语言那样使用 UML 解决所有问题是不实际的。可插入性的根本就是规格说明，纯粹外界可见的行为，以及检验与规格说明一致性的能力。

从需求模型中得到测试

提交的软件当然应该依从于需求模型。我们通常不能从需求模型得到设计，因为需求常常是局部的，因此设计员必须运用自己的技巧和判断来完成设计。但是测试仅仅需要测试需求中确切的东西：所以理想的话我们应该能从需求模型中得到测试代码。如果这能够自动化，这将建立需求和编码之间的可靠可验证的联系。如果我们不能如此，模型和编码就要割裂开来。

不过，需求模型（还有测试）通常比编码更稳定。加入新特性可能意味着要修改存在的编码；但是，我们仅仅需要为新特性添加新的测试而不用改变已有的测试编码。仅仅当目前的需求变化时，测试才需要被改变。这个开发过程的‘monotonic’（不变）属性减少了模型/测试分歧的问题：相对较少的变化使从需求中系统地手工获得测试变得可被接受；这是测试团队的工作。

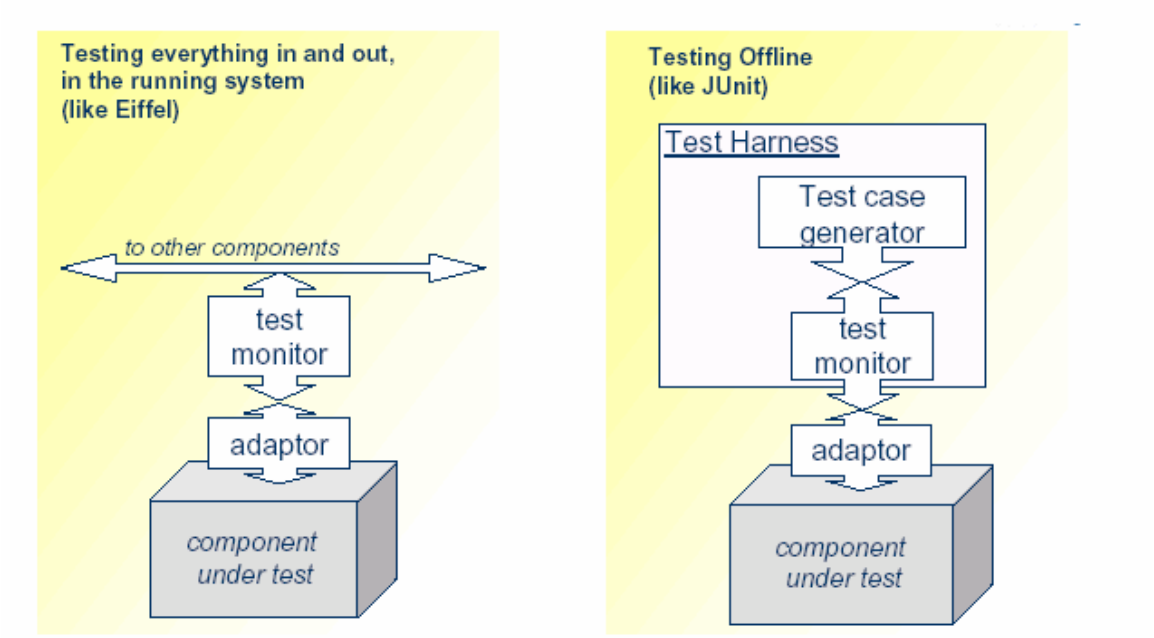
余下的部分研究如何从需求模型系统地推出测试编码。

测试框架

我们要区分 *Test Monitor*（测试监视器）和 Test Harness（测试探针）。*Test Monitor*（测试监视器）处于系统或者组件的接口，监视进进出出的一切，并检查组件的响应是否符合需求。它在运行系统中使用。Eiffel 语言提供 *Test Monitor*（测试监视器）；这个语言允许有 Pre（前置）和 postcondition（后置条件）附加在每条消息上；一个编译器选项将插入代码来检查每个方法的入口和出口。在调试状态，激活了 *Test Monitor*（测试监视器）的系统运行较慢，但功能会被检查。

Test Harness（测试探针）是围绕测试下的组件的一组软件元素；它离线运行，与目标环境脱离。它引发发向组件的消息序列，并检查所有返回正确与否，还能检查性能。

使用 Test Harness（测试探针）是重复自动测试组件的仅有的严格方式。在增量设计环境中，可重复的测试是必需的。



（适配器的目的是在测试中将组件内外的协议进行转换。一个组件可以代表一个 COM 接口和另一个 HTTP 接口；适配器接受 *Test Monitor*（测试监视器）的命令并产生适合的 COM 或 HTTP 序列；并将输出再翻译过来）

测试用例产生

Test Harness（测试探针）被看作是 *Test Monitor*（测试监视器）加上一个要发送的测试消息：一个测试用例产生器。

既然我们不能检查可能空间中的所有检查点，我们必须采用一些策略选择足够多的代表点来检查状态空间中的所有的区域和边界。这也许是使之自动化最难的方面，也是正在研究的主题。

“白盒测试”是根据被测试的程序编码，试图产生测试用例测试整个编码。但是在 CBD 环境中，我们通常不能拿到编码；而且如果规格说明组件有将被插入的可插入的接口，而我们在写测试的时候不知道，。

“黑盒测试”被设计为不必知道测试的组件的内部的结构。在产生测试用例时，我们在问题空间中寻找系统边界。例如，我们将认为除非它是特别编写的，也许任何能在顺序中成功地处理 3，17，33 和 127 项的软件就能处理其中的任何数字的部分；并且我们将特别检查它能否处理 0，1 和 2 项。

我们不能测试每个甚至是这几个例子的组合，另外， 我们需要在问题空间中寻找可能的固有依赖。自变量没有必要在多个组合中测试。例如，如果组件按照购货顺序合计从£0 到£1000 正确处理了订单上的一或者两项，它就能够处理其他数字项的任何总计。

Pre（前置）和 postcondition（后置条件）

我们喜欢为用例写精确的 pre（前置）/postcondition（后置条件），因为它们确切地阐述了会达到什么，而不是如何达到。它们可以直接被翻译成 *Test Monitor*（测试监视器）。

每个 postcondition（后置条件）与前置条件成对出现；后者阐述了前者的适用范围。举个简单的例子：

用例　计算订单的总价

前置　计数订单上>0 的项目

后置　总价＝这些项目价格的总和

这告诉我们在前置条件为真时会有什么。

当前置条件不为真时，它告诉我们什么？

回想在指定需求的时候，能分开处理部分视图是有价值的。这个规格说明告诉我们一个用户的类的系统预期什么。它不禁用其他的视图，这很重要。我们可能有另一个用户角色的类特别对订单上没有项目时发生什么感兴趣。或者我们可能有几个系统的变量，全部依从这个规格说明，但是在 0 情况下有不同的动作。

因此适当的解释是前置条件告诉我们什么时候 postcondition（后置条件）可适用。在前置条件为假时的情况，我们不知道（只是看这部分的话）系统会做什么。再次，这就是‘monotonic’（不变）特性对于可插入性和可扩展对象或者组件设计的重要性。

对应前置/postcondition（后置条件）对的测试是下面的形式：

```
if (<precondition = true>)
{
    stored-values := <@pre-variables used in postcondition>;
    <invoke method under test>;
```

```
if (<postcondition(new values, stored-values) = false>)
    { raise an exception; }
}
else { <invoke method under test>; }
```

（“@pre-variable”是在 postcondition（后置条件）中以“variable@pre”格式提到的变量，指方法被调用前的值。[Warmer]）

有许多关联的结构；请在[Catalysis]中看更多的细节部分：

- 成立条件。成立条件看起来象一个前置条件，但是在为假时没有变化发生。
- 模版。模版是一些可以被参数化并且被包含在模型中的模型（象一个宏）。用模版组装模型不是不变的：前置条件能加强。

属性和关联

如同我们能看见的那样，postcondition（后置条件）等是用对象模型的属性和关联关系术语编写的。例如“The Guest is now the occupant of a room in the Hotel, that was previously unoccupied（现在客人是酒店里的以前空着的房间的住客）”指定*occupant*关系的变化。

为了能测试这个需求，我们需要访问*occupant*关系。但是，等一下：需求规格说明中没有对设计的说明——这样就可能不存在任何这样的关系，至少在通常的数据库术语中不存在。需求模型的关联关系和属性仅仅告诉我们什么信息系统中以某种形式存在。看看类型图，它说明我们能够询问系统一个特定的*Hotel*（酒店）*accommodation*（提供）什么，并且，回答是collection of *Rooms*（房间的集合）；并且，我们应该能询问每个房间的*occupant*（住客），并且，回答应该是一个*Person*（人）或者是Null。

所以任何服从规格说明的设计应该提供对关联关系和属性的查询操作－只有这样我们才能测试它。

根据这个 postcondition（后置条件），*check guest into hotel* 用例的测试将是如此（*hotelsys* 是被测试的组件）：

for all room in hotelsys.accommodation(hotel)

previousOccupant[room] := hotelsys.occupant(room);

hotelsys.checkInGuest (guest, hotel);

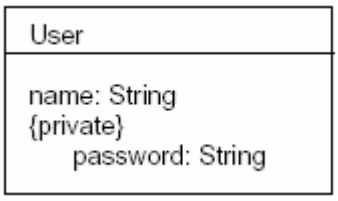
if (not previousOccupant[hotelsys.roomWithOccupant(guest)] = null)

raise exception;

注意我们让 *hotelsys* 组件执行每个查询和操作：我们没有假定任何 hotel，guest 等对象能执行的操作。

私有属性

不是所有的属性都能暴露。例如，我们可以建立一个 *User* 模型有一个属性 *password:String*。“登录”用例关连到一个用户的终端，检查在终端上输入的字符与 User 的 *password* 属性是否符合。



设计绝对不能在系统界面上暴露密码。所以我们不能编写象下面这样的测试：

system.login(user, myWord, terminal);

if (myWord = system.getPassword(user)

and not system.isLoggedIn(user, terminal)) raise exception;

为显示这个限制，我们可以将这个属性设为私有的，禁止设计去实现 *getPassword*。再次，回想一下我们没有说属性自己如何实现。

为了用一个私有属性测试用例，我们不得不执行一系列的操作来表现规格对应的行为，例如：

system.setpassword (user, “thing”);

system.logout (user);

system.login (user, “fiddle”, terminal);

if (system.isLoggedIn(user)) raise exception;

system.login (user, “thing”, terminal);

if (not system.isLoggedIn(user, terminal)) raise exception;

这种测试很难从需求模型自动得到。

不变量

一个典型的不变量可以是“The guests of a Hotel are the occupants of its accommodation”或者更简洁，

Hotel:: guests = accommodation.occupant

足够简单可翻译成测试代码。它应该在任何可影响所涉及的关联关系的用例之后测试。

（规则在组件之间有回调时更复杂。）

非功能（质量）需求

一般来说，类似载入、延迟、可用性、安全等非功能需要专门测试设备和测试过程。

在全部这些情况的中，UML 很少讲如何确定需求。确定它的最好的方法以测试参数的形式赋给给有关的工具。不过，用例和类模型提供的基本的词汇可以被使用：例如，“客人入住酒店需要击键次数在 5 次以下再加上预约号码或客人名字”。

从需求模型迭代设计

需求的开发采用逐次逼近，并且经常交给开发者是基本要求。分析员和设计员之间的区别体现在技术上而不是坐在不同的房间里。

在极限编程（XP）中，设计员应尽早选择高风险的部分来进行，应该有所有极限编程（XP）优势－最简单的东西可能有效，先编写测试，一次只作一个改变，一次改变一个（重构）（不改变测试）或细化（对测试的扩展，先前的测试则继续）。

设计任务

（我们在这里对编程和设计不作区分。）

极限编程（XP）计划工作安排设计任务[Beck]时间表。用户把优先权分配给需求的特性；设计员评估他们需要的时间；他们在一起决定时间表。时间表每两周都要复查，检查进度并根据对需求、架构和任务估计的理解的改变进行调整。

在 XP/RM 方案中，

- 分析员代表用户：或者如果用户是分散的就完全代替用户；或者与用户坐在一起象在警局中的律师和他的代理人一样。

- 用户把特性进行优先排序。特性是一些能向用户展示的，测试团队可以进行测试的交付系统的特征。特性可以是：

—— 一个用例。例如“用户能看到产品列表”。

—— 一组相关的用例。例如用户可以 LOGON 和 LOGOFF。

—— 一个在 postcondition（后置条件）中的子句，或者是相关的一组这样的子句。例如每个登录都被记录下来，并且管理者能得到全部登录的报告。

—— 业务规则。例如用户不能从两个终端上同时登录。

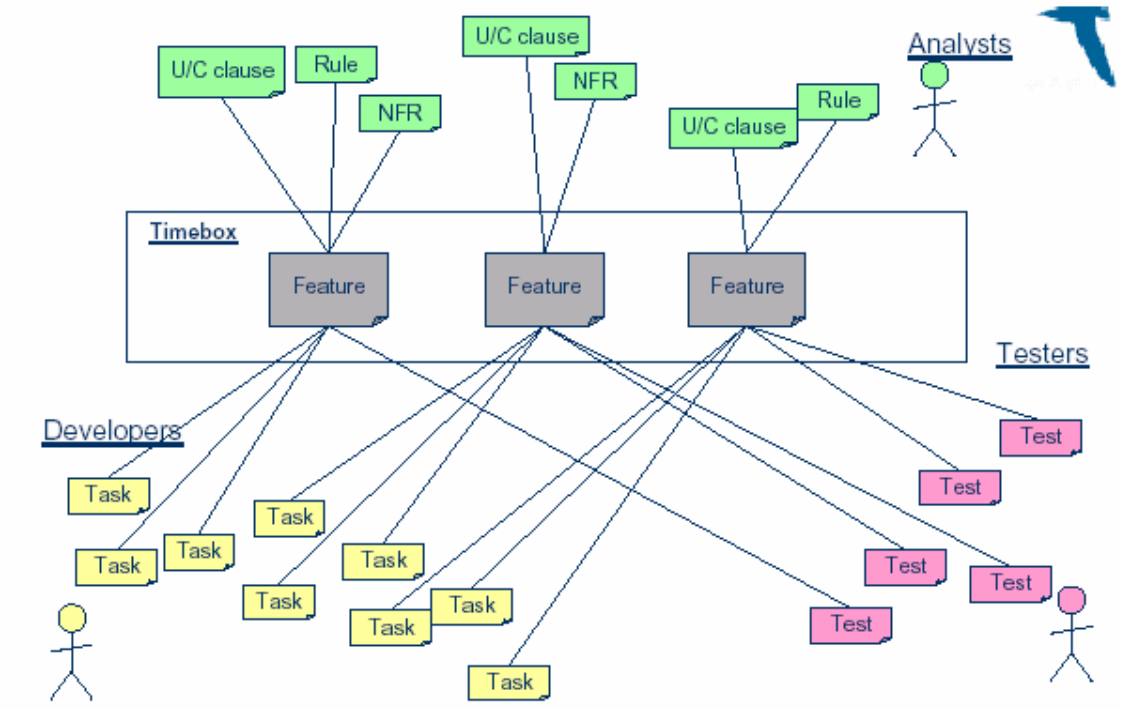
—— 非功能/质量需求。例如 1000 个并发用户时性能不下降。

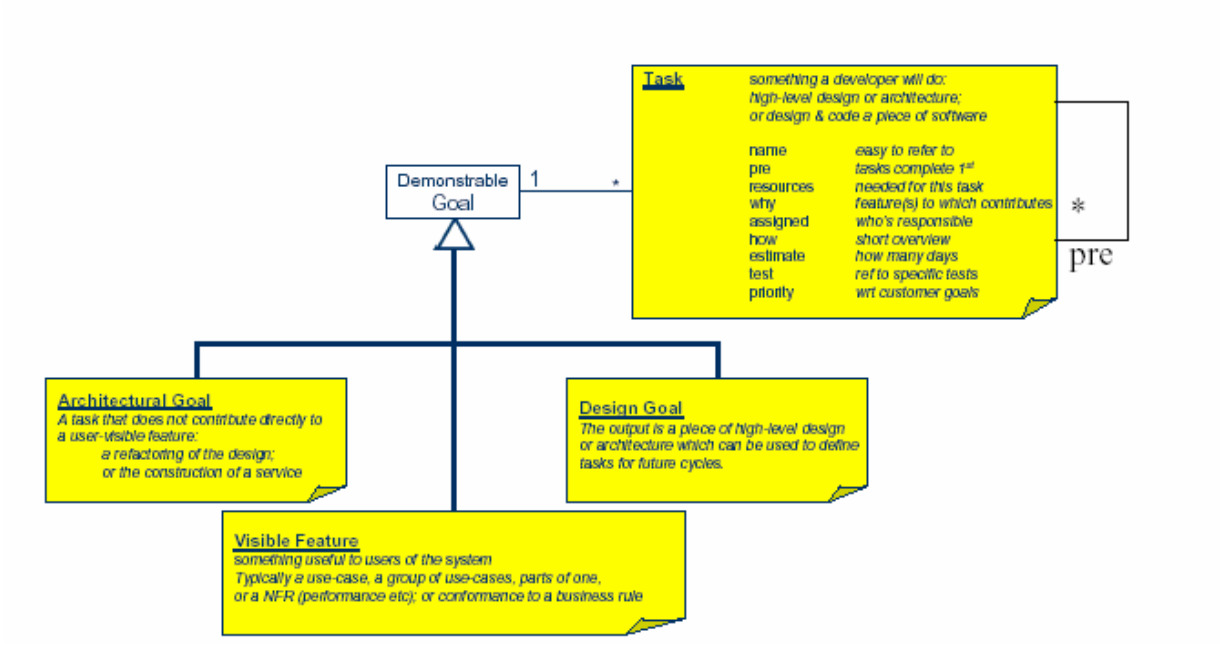
参考需求模型中需要实现的部分，每个特性都写在项目计划中。

- 设计员定义并估计设计 *Task*（任务）。任务是设计员领导的设计或者编程工作的一个单位，少于 5 个设计日。设计可以包括实际的编程在内，或者它可能是高层的设计，或者现存系统的实验。虽然不必演示给用户，但是每个任务都有一个 *demonstrable goal*（可验证的目标）。每个特性都有一个主要的针对它需要完成的任务；但是也有辅助的和结构上的任务。典型的特性可能需要一些用户接口设计，数据库的额外工作，一些内容，一些业务逻辑。而且这些工作要集成起来。这些中的每个都是一个任务（也许是因为不同人有不同的技能）。任务都有先决条件：其他任务必须首先被完成。当定计划的时候，任务也许被重新安排，但是仅仅在先决条件允许的范围内。一些称为“架构任务”的任务是整个范围的先决条件，不是被绑定到某个特性上的。这些对重构和基础架构尤其典型。

- 测试团队也是计划搏弈的一部分。他们检查特性是否已经被很好地定义并足以被测试，并向进度安排提供测试任务计划：一个特性直到被测试后才完成。（测试员测试特性，而不是任务的结果：单元测试由开发者来写。）

在极限编程（XP）中，时间表的执行不断地被监控着。





费用

分析员应尽量在项目的前几个星期内对整个系统有总体了解（在 RUP 的“elaboration"阶段）。在这期间，设计员应该选择一个部分构造原型。使用通常的极限编程（XP）的估计和监控编程工作的技术，开发团队的速度可被校准。

从需求模型中的原型设计方面的大小和实现它的时间，可以获得对需求其他方面所需时间的估计。各种电子表格和工具，象 Optimise，能帮助估计模型的大小。在固定价格的合同要求里这是非常起作用的。

工具支持

XP/RM 在这方面将从支持工具中受益：

- 分析工具帮助我们实施模型细化模式和进行检查。
- 帮助从需求模型中产生测试用例；并且，跟踪测试用例到模型的变更。集成各种测试工具的结果。
- 项目管理工具支持特性和任务的规划工作；能把特性定义连接到需求模型上；支持在计划表中重新安排任务；支持在项目内部网上的审查和任务分配。

总结

电子商务需要快速开发大型系统。类似极限编程（极限编程（XP））的轻量开发过程是成功的根本。极限编程（XP）是一个严格的方法，要求经常的回归测试。不过，它假定用在小组人员分析客户需求和开发方案的小型项目上。这在很多软件开发中是不实用的，除非因为熟练的分析员和熟练的开发者常常不能是同一个人。

我们已经看到了：

- 结合基于 UML 的需求分析而不影响极限编程（XP）的快速响应；
- 对需求模型的很强的一致性和完全性检查；
- 用例和增量开发工作的关系；
- 维护一个测试和需求模型之间的严格关系。

参考书目

[Beck] *eXtreme Programming eXplained,* Kent Beck

[Catalysis] *Objects, Components and Frameworks in UML,* D D'Souza & AC Wills [A-W 98]

[Cockburn] *Writing Effective Use Cases,* Alistair Cockburn [A-W 2001]

[Jones] *Systematic Software Development with VDM,* Cliff Jones [PHI 1986]

[Kruchten] *Rational Unified Process* P Kruchten [A-W 2000]

[Morgan] *Programming from Specifications,* Carroll Morgan [PHI 1990]

[Warmer] *Object Constraint Language,* Jos Warmer and Anneke Kleppe [A-W 1998]

征稿

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

UMLChina 公开课

“UML 应用实作细节”

（2003 年 3 月 1-2 日，北京）

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

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

本课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，阐述如何（只）使用 UML 里的三个关键要素：用例、类、顺序图来完成软件开发。使学员自然领会 OOAD/UML 的思想和技术，并对实践中的误区一一指正。整个过程简单实用，简单到甚至可以只有一个文档，非常适合中小团队。

详情请见>>

分析模式应用－－帐务模式

Windy.J 著

查看评论

代理是一种销售方式，代理商通过出售源厂商的产品或推广服务商提供的服务获得利润，不管是产品还是服务，代理商和产品/服务商之间的利润计算都是基于更优惠的代理价格（对产品代理而言，代理价格可能跟历史代理销售收入有关，如果上年度或上季度的代理销售收入较高，可能代理价格就更优惠），或者基于代理销售收入本身（对服务代理而言，更有可能是这种形式，代理销售收入越高，代理佣金也就越高），或者可能的其他方式；因为基于代理价格的结算相对而言较为简单，结算通过代理商对购买的产品付费已经完成，所以，我们在这里要应用到帐务模型进行处理的，不是基于代理价格的结算，而是基于代理销售收入的结算。

假设有这样一种代理商，代理推销汽车美容服务，代理商会跟服务商签订合同，规定代理条款，付费方式，各自的责任权利，等等。当代理商发展的顾客来消费的时候，顾客的汽车美容活动是顾客的事件，会产生顾客维护账单（帐务条目），然后，这个账单将成为代理商的原始事件，根据合同规定的规则（Posting Rule）产生代理商的帐务条目。

代理结算还有一个特点，就是一般按某个时间单位结算，例如一个月，一个季度，一年之类的，然后代理商的结构可能还会包括代理商下面的客户，客户下面还会有用户（象前面的例子，可能代理商发展了一个公司作为客户，然后公司里头的员工可以作为独立消费的用户），根据用户和客户的消费额不同，合同规则可能会不同，例如，月消费 1 万元以上的可能成为 VIP 客户，代理商可以提成 10%，而对普通的客户代理商只能提成 8%。

用户消费产生的费用应该跟用户相关，而这些费用一般会分为很多种类，例如洗车，油漆，打腊，个性内装，防紫外线处理，发动机清洁护理等，因为每个种类的成本和效益不同，所以它们的提成比例也可能不同。

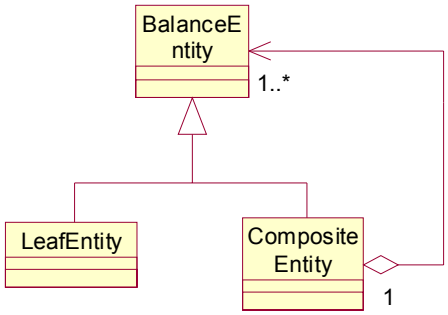
好的，假设代理商和服务商签订了如下合同：

代理商帮助服务商发展客户和用户，推广各项汽车美容服务，对所有客户高中低档（参见项目档次规定说明）美容服务每月累计实收费用进行提成，提成比例分别为高档 20 万元及以下 8%，20 万元以上 10%；中档 6%；低档 3%；其中，实收月消费超过 10000 元人民币的客户成为 VIP 客户，提成比例按实际消费额增加 2%，代理费用按月结算，每月 5 日之前服务商将上月的结算款付给代理商。欠款部分等归还以后再按同样规则计提成。

（注：以上数据仅作讲解示范用。）

根据这样一些特点和相似性，以及上一节提到过的帐务模式，我们发现，帐务模式可以应用到这个问题上，尤其是 Posting Rule，可以解决结算规则的灵活设置。

首先我们来看代理商的层次结构，它可能包含客户，也可能客户还包含用户，也可能代理商不包含客户，只包含用户（象推广连锁店服务），要解决这样灵活的层次结构，在这样可以用到 Party 模式中的层次组织结构模式，并稍作一些调整：

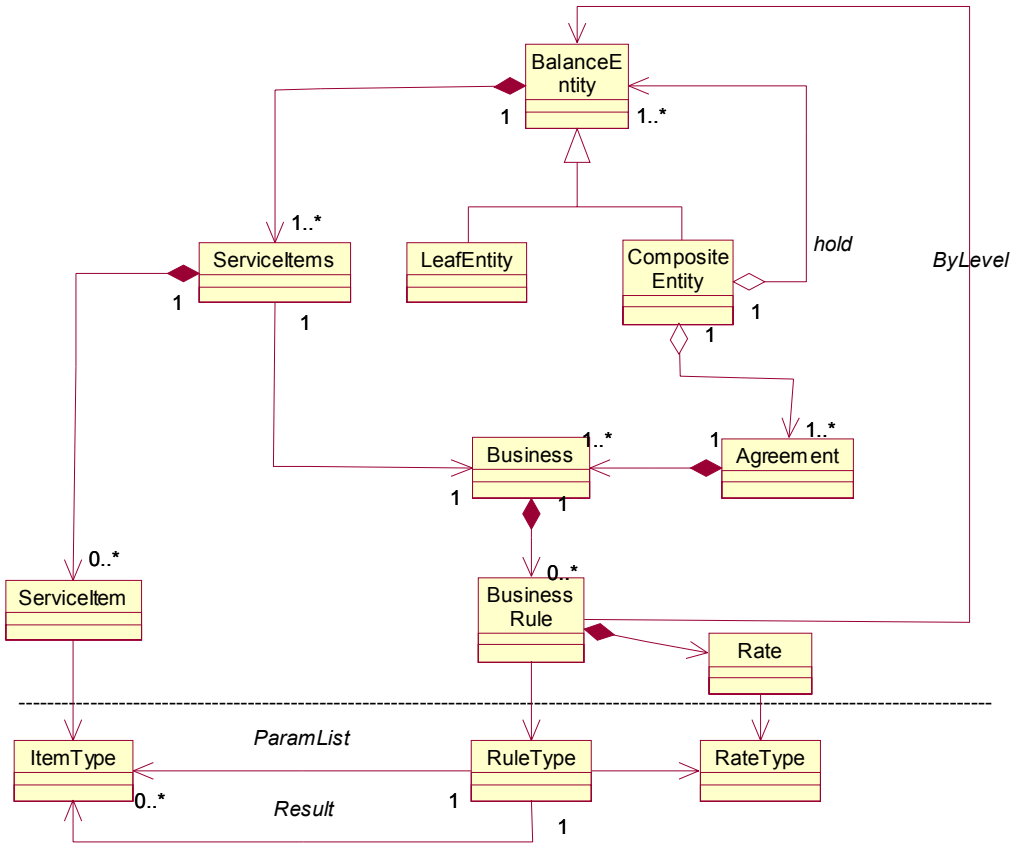


Party 模式中的层次结构模型支持多种灵活的层次结构，但这里我们只要关心上下级的包含关系就可以了，参加结算的称为结算实体 BalanceEntity，不可再拆分的称为 LeafEntity，可以包含下级结算实体的称为 CompositeEntity，因此在这里还可以支持一些象客户群，用户群这样的概念（一般没有那么复杂）。

从前面可以看到，代理商签有合同 Agreement（如有多个代理商，合同内容可能相同也可能不同），在合同里面规定了不同的业务 Business（汽车美容服务推广），每种业务的结算规则（BusinessRule，对应帐务模式中的 PostingRule）各不相同，而结算规则作用在不同的服务项目（ServiceItem）上，因此，我们需要抽象出合同，业务，业务结算规则，服务项目，组成主要的模型结构。

为了从系统外部配置规则，我们同样还需要抽象出项目类别，规则类别和它们之间的关系（需要注意的是，项目从用户向客户汇总属于业务结算规则的一部分，还有最后的结算账目总额，也是规则的一部分，需要纳入配置）。同时，根据“范围”模式一文中的比例模型，这里有些规则需要用到比例，因此，它也需要加入模型。

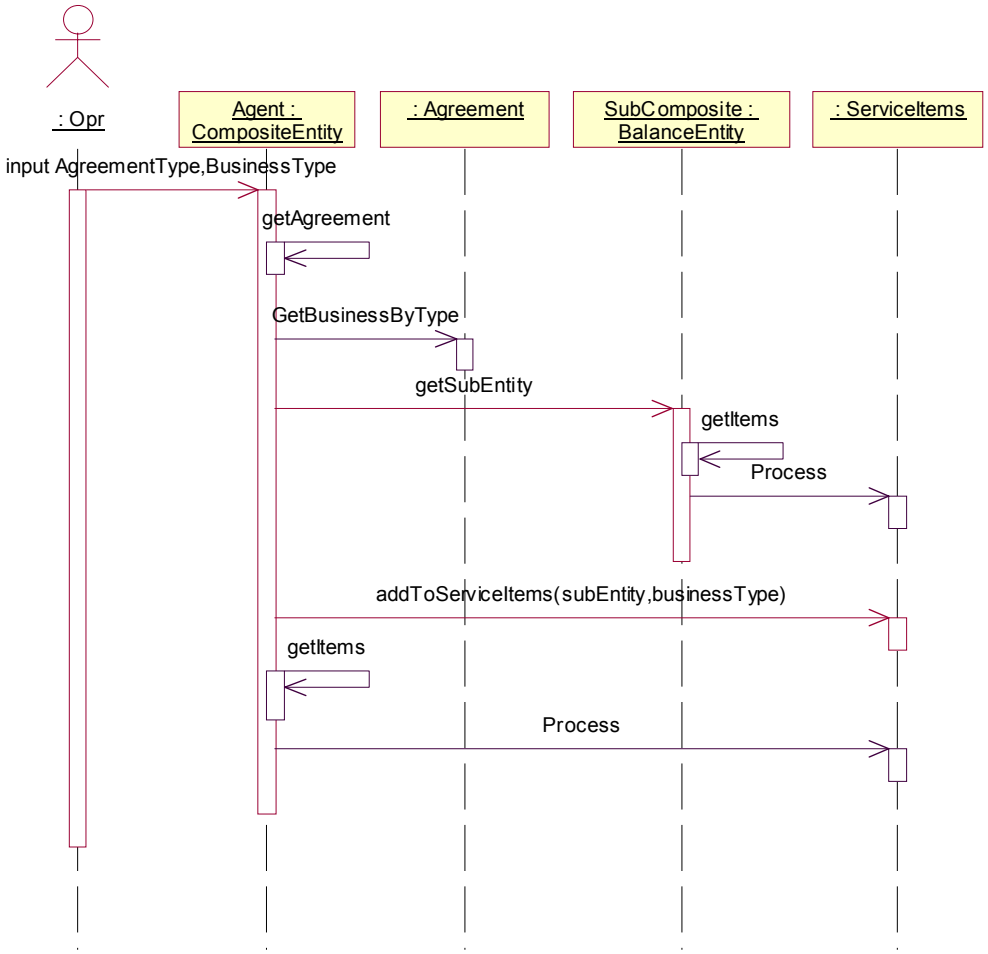
具体的模型如下图所示：



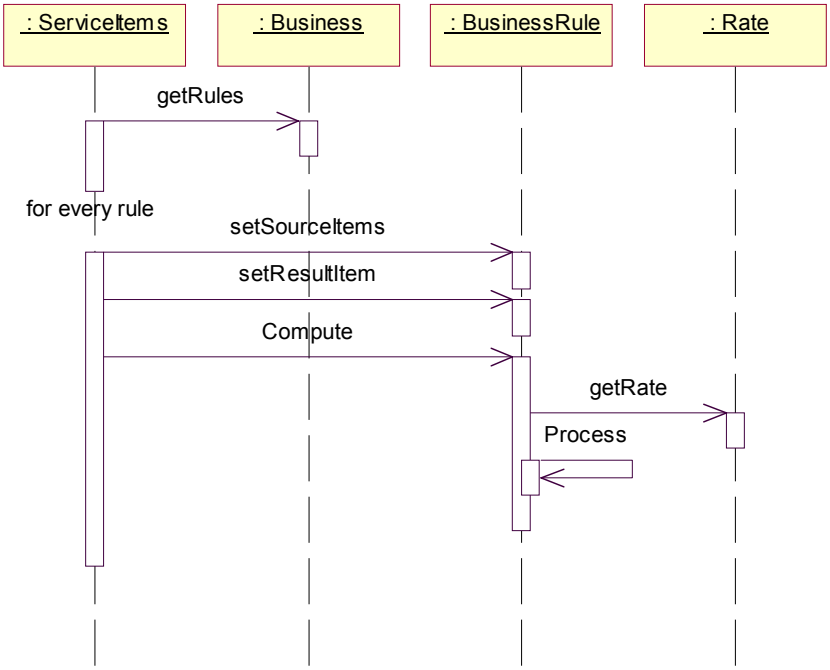
如上图所示，虚线上面是基本的模型结构，下面是用来进行配置的类型和关系，其中的费用项目类别 ItemType 作为规则类别 RuleType 的参数和结果；而 RuleType 跟比例类别相关联，BusinessRule 的计算方法也在 RuleType 中指定。

在前面的例子中，系统初始化的时候，建立代理商，客户，用户（作为叶子实体）的结算实体结构，建立它们之间的关联关系，根据费用项目类型，初始化费用项目（包括所有的具体费用项目，高中低三档的费用累计项目，和所有费用累计项目，普通佣金项目，和 VIP 佣金项目，佣金总和项目），费用项目容器，然后针对费用项目建立处理规则，和需要的比例（包括 3 个固定比例：中档佣金比例，低档佣金比例，VIP 额外佣金比例和 1 个范围比例：高档佣金比例）。

而佣金计算的过程可用下面的序列图来表示：



费用数据集合的处理序列图如下：



模型的扩展

上面的模型基本可以解决代理商和服务提供商之间的结算合同，如果过了一段时间，代理商又和服务商签订了下面的合同条款：

代理商帮助服务商推广连锁店服务，对每家由代理商发展的连锁店，按参加金额 5 万，20 万，100 万及以上分别提成 2%，5%，10%；以后的每月，根据它们的利润，按 1 万，5 万，10 万及以上分别提成 1%，2%，3%。

现在，又怎样面对这个问题呢？

首先来看结算实体结构，代理商，连锁店，好像还比较幸运，结算实体模型可以适用，然后，合同，嘿，合同，业务 Business（新的业务是连锁店推广）都还在，新的服务项目有首月连锁参加金额，该金额累计，该金额对应佣金，连锁店经营利润，利润累计，利润对应的佣金，计算规则和比例同样可以制订，需要对系统增加新的 BusinessRule 子类和实现，如果用表达式解析等方法对规则的计算进行配置，那么不必对 BusinessRule 进行继承或更改实现。

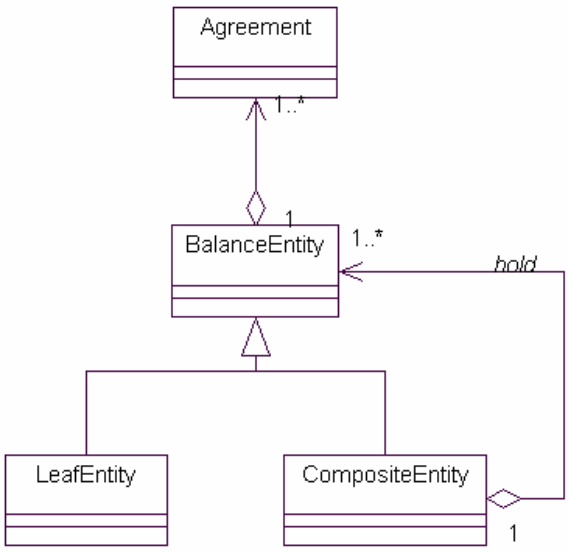
好像问题解决了，不是吗？用原来的模型，需要的东西都可以从中找到，但是，服务提供商说，等等，我们要这个代理商的佣金结算金额，分别给出来，但是，不要分成两次计算，希望应用一次运行就能出所有的结果。

这样好像有点问题，从上面可以看到，我们是分两个合同，两次运行才能得到服务推广佣金和连锁代理佣金，怎么办呢？

回到上面的模型，我们可以看到，代理商和合同之间的关系是 1 对多，可以支持多个合同，但是，两个合同规定中的结算实体完全不同，如果放在一起结算，结果不会出错，但是会导致许多无用的 ServiceItems，和多余的计算过程，想想，同时对服务推广合同下的客户和用户计算连锁佣金和利润提成，以及还要对连锁用户计算服务推广佣金（它们根本就不存在），这是非常不合理的。

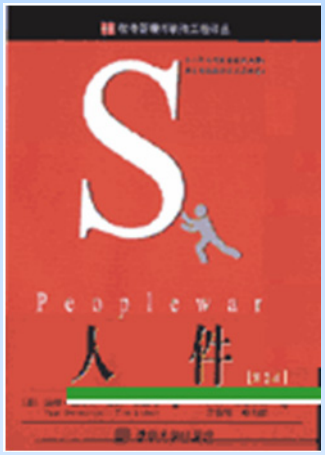
可以这样，将代理商与合同的关联上移到基类，从而可以让代理商关联到两个合同，而代理商下属的结算实体关联具体的合同，并由具体的合同和具体的下属结算实体一起进行计算。这样，模型进行小小的改变，支持了应用的完整性。以后增加新的代理商，新的合同条款，多个合同，一样可以适用。

如图所示：



该模型适合中小型代理结算系统，以及适用于其他行业代理结算业务。

《人件》



Tom Demarco，Tim Lister

翻译：UMLChina 方春旭、叶向群

老板，烧毁这本书，别让开发人员看到

Frederick P. Brooks, Jr--《人月神话》20 周年纪念版第 19 章

近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的《人件》，我衷心地向我的读者推荐这本书。

Edward Yourdon

《人件》在 1987 年出版后，立即成为最畅销的作品，特别是一些组织，开始认识到软件开发问题与程序语言、软件工程方法、软件过程成熟度无关。正如 Bill Clinton 所言，“是人，笨蛋！”当人件第一版出版时，我写了一份评论，**“我强烈推荐你买一份人件给你或你的老板，如果你是一个老板，那么为你部门的每一个人买一份，并给自己买一份”**。这建议在 12 年后依然有效，并且更加热烈。

Mark A. Herschberg

《人件》正确指出软件工程是对人，而不是针对技术。它看到在软件开发过程中人的许多方面，并指出人并不是软件开发机器中简单的小齿轮。这本书也包含涉及办公环境的章节，并提供证据说明为什么通常的观点并不适用于软件开发。**我只学了这些章节，就辞掉了原来的工作！**

中文译本即将发行！

别把开发人员当成牲口

Mike Gunderloy 著， think 摘译

查看评论

Tom DeMarco 和 Timothy Lister 在 1987 年写了《人件》。1999 年，《人件》第 2 版由 Dorset House Publishing 出版，新增了 8 章。该书基于 1970 年代早期开始的研究，某些东西已经有四分之一个世纪的历史。对程序员的岁月来说，这是很长的时间。我好像回想起我们过去用古老工具编程的情景，但记忆相当模糊。

现在，那些炙手可热的新一代 dot-com 开发人员能从这本书中学到什么吗？足够令人吃惊——答案是 Yes！这本书是软件开发领域的经典，只有 200 页多一点，但值得每个团队负责人和经理阅读。

注意，这不是一本编程书籍。该书的子标题是“高产的项目和团队”。DeMarco 和 Lister 所关心的是生产力。用任何尺度来度量生产力，在软件工业的不同团队之间有 10:1 的差距。换句话说，最好的团队开发一个软件需要 1 年，最差的则要十年。原因看起来不止是最好的团队拥有最好的程序员（虽然最好的程序员趋向于加入有培养优秀团队环境的公司）。那么差别在哪儿呢？

DeMarco 和 Lister 认为差别在于：优秀团队有良好的工作环境，不必受制于那些对开发软件毫无益处的荒谬措施；经理不会挡团队的路（如果必要，经理还会把挡住本团队的路的其他人推开）。沿着这条思路，他们谈及大方法论如何使人变得愚笨，必须带门和窗的办公室，在工作中寻求乐趣，必须存在一些混乱，为什么有的组织能够学习，有的不能…

提防家具警察

作者认识到的很多问题归结为管理层把开发人员当成牲口，而不是使用大脑生存的人。例如，作者花了好些段落来把家具警察送上烤肉架。家具警察（说起来很悲哀）在很多成型的公司里都有。如果你曾经在一间“小隔间牧场”工作，下面的文字会为你敲警钟：

用家具警察的眼光来看，地下室的空间确实更好，因为它本身为统一布置提供了更充分的条件。但是人们在自然光下工作得更好。他们在有窗户的空间感觉更好，并且会把这种感觉直接转化为更高的工作质量。人们也不想在一个完全统一的空间工作，而是希望按自己的喜好和品味来布置自己的工作空间…工作空间总是充满了噪声、干扰，没有私人空间，并且没有多少办公用具。有些公司的办公场所比其他一些公司的要好看一些，但是实用性并不强。没有一个人能在那里把工作做好。那个人本来可以像一只海狸一样，在一个带有两个大的折叠桌子和一个关着的门的安静舒适的地方工作…。

幸运的是，在某些公司或者在某些领域，事情还是好转了。Microsoft 因为在办公设施方面“浪费钱”而“声名狼藉”：给开发人员配备带有门和窗的办公室，还提供免费饮料，休息区，还有其他很多无聊的东西。结果呢？Microsoft 的人们确实喜欢呆在办公室，自由自在地集中精神，写出高质量的代码。事实上，Joel Spolsky 曾声称：微软成功的原因之一就是公司里的所有经理都读过《人件》。Joel 推荐软件经理每年重读这本书一遍——这主意不坏。

家具警察只是更大问题的症状和伪装。问题在于对开发成本的错位认识，导致采取削减成本的行为，实际上从长远来看反而多花了钱，因为这些措施阻碍团队的形成，阻碍软件开发。一些其他的例子：

吝啬鬼管理层用那些可憎的“宣传工具”（你知道的，经常是一些全彩色照片加上一些类似“把灵魂献给公司是最高利益所在”的标语）取代了真正能起到激励作用的东西（如更高的薪酬和有门的办公室）。

制度上执着于过程改进程序（特别是 CMM），把精力集中在把事情做得更熟练而不是做市场需要的东西。如果 CMM 是现实的反映，那么完美的泥馅饼应该比缺角的椰奶馅饼好卖。

团队被分散在公司区域的不同角落，因为为他们找到一片连着的区域对家具警察们来说太麻烦。

现在你知道了，问题成堆。DeMarco 和 Lister 勇敢地把它们编辑成册。

不可思议的顺流状态

攀岩者会进入称为“顺流”的状态，在这个状态下，他们精神集中，在岩石上移动是如此的轻松和稳定，整个身体已经溶进攀登之中，每件事进行得都那么顺利。当然，我们可能会辨称，软件开发和攀岩不同，至少我们的代码崩溃的时候不会飞沙走石。但有一件事是相同的：顺流。书中对这种状态有很多描写。如果你在承担严峻的软件开发任务时，没有被各种琐事打断，你就会知道顺流状态什么样。这种状态不只感觉良好，而且也能使你得到好的代码。

顺流是管理层所不了解的东西，这就是家具警察能被授权控制你的工作环境的原因。正如 DeMarco 和 Lister 所指出的，顺流状态完全不为管理层所理解，因为管理者在工作中被经常打断是自然的。管理正是不断应付各种打扰的艺术。不幸的是，软件开发不是。在 5 分钟的电话之后，需要开发人员花 15 分钟或更多，以回到顺流状态。如果 5 分钟的电话来上 12 次，你就死定了。作为一个旁观者，我不喜欢看到有人煞有其事地研究为什么杰出的程序员进入顺流状态比其他人快得多。

记住，不止是别人强制我们不能进入顺流状态，我们自己也在这么做。如果你有顺流状态方面的问题，试着关闭你的 email 客户端几个小时。世界不会因此完蛋，而你就不会每 30 秒受到一次干扰了。

有趣的阅读

如果你已经看过了《人件》，那么现在怎么办呢？两条建议。首先，浏览一下 Atlantic Systems Guild (http://www.atlsysguild.com/)，该公司由《人件》作者帮助建立。其次，把它再读一遍！好好看它，这本书简单有趣。写这篇文章时，我很高兴又看了一遍。这样的精华语句百读不厌：

如果老板特别需要，仪式会议的负担几乎会不受约束地增加。例如我们知道一家公司，每天开两个小时的会议是公司的准则。如果开会时与会者离开会场，就用扩音器叫他们进来希望他们参加会议的全过程。不参加会议被视为一种威胁，要受到严厉的处罚。

你在书中不会发现一行代码。但如果你正在管理一支团队，你会发现聆听 DeMarco 和 Lister 的教诲会使你最终更快地在产品中得到更多更好的代码。

你的老板有这本书了吗？或者你想在他的桌上放一本作为礼物？

（2002 年 3 月）

UMLChina 公开课

“UML 应用实作细节”

（2003 年 3 月 1-2 日，北京）

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

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

本课程秉行 UMLChina 一贯的“只关心细节”的原则，内容完全是由 UMLChina 自行设计，围绕一个案例，阐述如何（只）使用 UML 里的三个关键要素：用例、类、顺序图来完成软件开发。使学员自然领会 OOAD/UML 的思想和技术，并对实践中的误区一一指正。整个过程简单实用，简单到甚至可以只有一个文档，非常适合中小团队。

详情请见>>

《人月神话》关注“软件开发”本身

《人件》关注软件开发中的“人”

第 1 章 在今天的某个地方，一个项目正在失败

本质上，我们工作中的主要问题，与其说是技术问题，不如说是社会学问题。

我们倾向于集中精力做技术方面而不是人际关系方面工作的主要原因，不是因为它更重要，而是因为它更容易做。与弄清楚贺瑞斯为什么恐惧不安或者苏珊为什么在公司只工作了几个月就对公司不满意之类的事情相比，安装一个新的磁盘驱动器肯定是微不足道的。人际交往是很复杂的，并且就效果而言从来都不会是很明晰和清楚的，但是它们比工作的任何其他方面更重要。

如果你发现自己关注的是技术而不是社会方面的问题，你就相当于在一条黑暗的街上丢掉钥匙，却到邻近的另一条街上去寻找。因为“这条街上的灯比那条街上的要亮一些”。

第 2 章 做吉士汉堡，卖吉士汉堡

一个缺少洞察力的例子：一个老板表露出被他部下的个性吓倒的极端信号。他有一个非常有天赋的员工，每年大部分时间花在回访客户的路上，结果以报销单为生。该员工的一份开支分析报告显示，他在吃饭上的花费与其他出差者不协调，他的餐饮费用比其他出差者多一半以上。在一次公开的例会上，老板因一时愤怒竟然污辱那位员工，说他是一个“食物罪犯”。然而，那个员工的总费用并没有超出标准——无论他在饮食方面的额外费用有多少，他都从其他方面节省回来。此人并没有比其他人花费更多，只是与其他人不同罢了。

第 3 章 维也纳在等着你

几年前，我与南加州一个大项目的项目经理交流各自的艰辛历程。他开始叙述将项目和疯狂的时间表压到他下属的身上产生的影响。一是发生的两宗离婚案，其中原因可以直接追溯到与他的人经常加班有关；再者就是一个员工的孩子吸毒，其中原因可能是由于在过去的一年里，孩子的父亲太忙，未能尽到做父亲的责任，最后，测试团队的负责人又神经崩溃。

虽然你的员工在办公室也许被“更长时间、更努力地工作”这句话所影响，但从他们家里听到的是完全不同的话语。在家里听到的那句话是“生活已离你而去，要洗的衣服堆满了壁橱，你的小宝宝没人抱了，

你的配偶正开始朝其他地方张望着。旋转木马的人生只剩下一圈了，只有最后一次中奖机会了，如果你把你的生活用在 COBOL 上……”。

第 4 章 质量——如果时间许可

想像一下以下的臆想实验：问在大街上的 100 个人什么公司或文化或国家以高质量著称。我们预言今天将有一半以上的人会回答，“日本”。现在问另外的 100 个人什么公司或文化或国家以高生产力而著称。预测大部分人又会提到 “日本”。一个被公认为质量领导者的国家同样会以其高生产力著称。

等一下，高质量与高生产力怎么可能共存呢？那公然违背了普通的学识：给产品注入质量意味着你要花费更多的代价来生产它。要寻找线索，看一看田岛和松原这两位最受尊敬的评论员说过的关于日本现象的话：

> 在日本不存在质量与价格之间的交易。相反，高质量导致成本降低的思想却普遍为人所接受。

第 5 章 重新研究帕金森定律

帕金森定律远非一条公理。它不是 “牛顿定律就是一条定律” 那种意义上的定律。牛顿是一个科学家，他按照严格的科学方法研究地心引力所产生的效果，他的定律是经过严格的检查和测试后才提出来的。牛顿定律经受住了后来大约两百年的研究的检验。

帕金森不是一个科学家，他没有收集数据，他可能甚至不懂得统计学推论的规则。帕金森是一个富于幽默感的人，他的“定律”之所以流行，不是因为它真实，而是因为它有趣。

第 6 章 苦杏仁苷

苦杏仁苷是从一种略带苦味的杏仁里榨取出来的无色液体。在瑞典，你可以在杂货店里用杏仁汁的价格买到这种原料，并且你可以像其他任何一种榨汁一样把它用在烘烤中。而在墨西哥，你可以用五十美元一滴的价格购买它，用来“治疗”致命的癌症，当然，它治不了任何疾病。所有证据表明这是一种残酷的欺诈行为。但由于根本没有其他任何人向他们提供药物，不管这些小贩多么令人不可容忍，晚期的患者还是会相信苦杏仁苷小贩的吆喝。非常绝望中的人不会认真考虑这些证据。

同样，很多经理们也是“非常绝望的”，并且他们的绝望使得他们极易成为一种声称能提高生产力的技术苦杏仁苷的牺牲品。几乎很少有任何证据能支持他们所要求买的那些东西。同样，证据可有可无，因为他们的需求是如此巨大。

第 7 章 家具警察

家具警察的头目是那个家伙，在你的员工要搬进新办公室前一天，他先走进去徘徊几圈，脑子里开始这样盘算着：

“看看，每件东西都是那么漂亮、一致！甚至分不出是在 5 楼还是 6 楼！但是一旦那些人搬进来之后，一切都要被毁了。他们会挂上照片和个性化他们小小的工作间，一切都将杂乱无章。他们也许要在我心爱的地毯上喝咖啡，甚至就在这儿吃午餐（耸肩），噢，天哪，噢，天哪，噢，天哪……”

第 8 章 “在早上 9 点到下午 5 点之间，你在这里从未做完任何事情”

一个令人烦扰的可能性是：加班不是提升工作平均质量的手段，而是增加工作时间数量的手段。你听到的证据是真实的，人们经常重复谈及诸如以下这些话题：

“在所有人上班之前，一大早我已经尽力完成了我的工作。”

“一个晚上干晚一点，我可以做两三天的事情。”

“办公室整天像个动物园，但在下午大约 6 点的时候，一切都安静下来了，你终于可以真正完成一些事情了。”

为了提高生产力，人们也许要来早一点来或晚一点走，或者干脆全部逃避，在家呆一天把重要的事情做完。我们的学术讨论会上，一位与会者说她的新老板不允许她在家里工作，因此在交一个重要报告的前一天，她请了一天的病假。

第 9 章 在空间上省钱

如果办公环境让人扫兴，那么人们会寻找一个地方去躲避。他们预订会议室或去图书馆或出去喝咖啡并且不回来了。不，他们不是去秘密约会也不是去密谋政变，他们正在躲避办公环境，出去工作。好消息是你的人真正意识到需要完成工作了，他们会想尽一切办法来完成。没完没了的噪声一开始，大家就都会千方百计地找个能工作的地方，无论它在哪里。

如果窥视一间会议室，你会发现有 3 个人在静静地工作；如果下午 3 时左右在咖啡馆徘徊，你可能会发现有人坐在那里，一人一张桌子，工作铺在他们面前；还有一些员工根本找不到，人们逃出去工作了。如果在你的公司确实是这样，那么这是一种控告，在空间上节省钱可能会使你失去一笔财富。

第 10 章　脑力时间 vs.体力时间

在一心一意动脑筋工作的时候，人们在意识上处于一种心理学家称为**顺流（Flow）**的状态。**顺流**是一种陷入沉思的状态。在这种状态下，有一种精神欢快的轻松意识，一种大部分情况下都感觉不到时间在流逝的意识：“我开始工作。我抬头看，已经过了 3 个小时。”没有意识去努力，而工作进行得很顺利。你经常处于这种状态，因此我们不必向你描述。

如果平均 5 分钟有电话打入，而你重新沉浸的时间是 15 分钟，那么投入顺流状态的时间（工作时间）总共需要 20 分钟。多接几个电话，半天就没有了。于是可以肯定，“你从未在上午 9 点至下午 5 点之间完成什么事情。”

第 11 章　电话

你经常打断与同事或朋友的谈话去接一个电话吗？当然，经常是这样的。你甚至没有考虑不接那个电话。而这样违反了公平的共同原则，让人乱了分寸，只是因为响个不停的铃声吸引了你的注意力。你不仅这样对待别人，也让别人这样对待你。你是如此习惯这种泛滥的行为以至于自己几乎注意不到。只有在最不可容忍的情况下，人们才会清楚诸如此类的行为是完全错误的：

第 12 章　把门带上

但是康耐尔实验包含了一张隐蔽的百搭牌。题目说明要求通过一系列的操纵输入数据流中的数字来形成输出数据流。例如，参与调查的人必须移动每个数字左边的两位数字然后除以一百等等。虽然题目说明并没有直说，但是所有运算的最终效果是每个输出数字必须等于它的输入数字。有些人意识到了这一点，但是有些人没有意识到这一点。那些意识到了这一点的人，绝大多数来自那个安静的教室。

专业员工每天做的事情中，许多是由左脑的顺序处理中心完成的。音乐不会特别干扰工作，因为是大脑的整个右边在消化音乐。但不是所有的工作都由左脑完成。可能有让你说“啊！”的突破会引导你到达一个可以节约数月或数年工作的创造性思路。创造性的飞跃包括在右脑的功能中，如果右脑忙于听背景音乐中的 1001 弦乐，那么就有失去创造性飞跃的可能。

第 13 章　采取保护措施

认为家里不能没有窗户的人们，最终会将他们的大部分白天时间花在没有窗户的工作空间里。亚历山大对没有窗户的空间没有什么耐心：“看不到景色的房间对于不得不呆在里面的人而言就像监狱。”

我们受到的教育是，承认没有窗户的办公室是不可避免的。我们听到，公司希望让我们每个人都有一扇窗户，但那是不现实的，肯定是这样。有一个充分的证据证明，不用花多少钱就可以在一个空间建立足够的窗户。这个现成的证据就是旅馆，任何旅馆。难以想像看到一个没有窗户的旅馆，这会让人受不了，（而且这是一个对你而言仅仅是在里面睡觉的地方。）因此旅馆配置了许多窗户。

第 14 章　霍恩布洛尔因子

当我暗示武断的标准化来自管理的不安全性时，参加室内讨论会的与会者几乎不能抑制自己，他们都有话要说。一个最愚蠢的故事是，下午午休时间，员工在煮咖啡的地方用微波炉爆玉米花时公司的反应。当然，爆米花会留下一种不易弄错的气味。来自高层的某个顶尖人物闻到了这种气味并作出了反应。他在备忘录中宣布“爆米花不够专业”因此爆米花被禁止了。

第 15 章　雇用一个变戏法的人

马戏团经理：你变戏法有几年了？

候选人：　　噢，大概 6 年了

马戏团经理：你能操纵 3 个球，4 个球，5 个球？

候选人：　　是的，都可以。

马戏团经理：你会用燃烧的道具变戏法吗？

候选人：　　当然会。

马戏团经理：……用小刀，棍子，开着的香烟盒，耷拉着的帽子呢？

候选人：　　什么戏法我都能变。

马戏团经理：你有一套与戏法相配套的让人发笑的喋喋不休的话吗？

候选人：　　是特别有趣的喋喋不休的话。

马戏团经理：那么，很好。我想你被录用了。

候选人：　　嗯……你想看我变戏法吗？

马戏团经理：哎呀，我没这么想过。

没有先看变戏法的人的表演就雇用他，想起来真可笑。那是常识，然而当你着手雇用一个工程师，设计员，程序员，一个小组的经理时，常识准则经常被搁在一边。你没有要求看一个设计或一个程序或任何东西。事实上，面试只是谈话而已。

第 16 章　　很高兴在这里

人员流动率成本占所有人力资源开支的 50%。但是那只是流动率看得见的成本。有一笔可怕的、看不见的、更加糟糕的成本。

在流动率很高的公司，人们倾向于一个毁灭性的短期观点，因为他们知道自己不打算长期呆在那里。例如如果你在参加为你的团队人员争取更好的工作环境的运动时，不要对某个高层人物感到惊讶，那个高层人物这样反驳道：

> “停下来，小鬼，你在谈论大笔投资。如果我们给工程师提供那么多的空间和隔音保护甚至私密的环境，可能平均每个人每月的开支是五十美元！将那个数额乘以所有工程师的工作时间，你会得出几万美元的数额。我们不能花费那种钱。我和别人一样非常支持生产力，但是你们看见我们第三季度有多么糟糕吗？”

第 17 章　　自愈系统

一个雇员在人事部怒发冲冠地要辞职。第二天，他或她的老板非常温和地来解释说整个事情是个愚蠢的错误。可能取消他的辞职处理吗？处理这件事的员工疑惑不解地看着部分完成了的辞职报告：以前的某个人制定了辞职处理过程，而这个过程不允许辞职过程中途取消。但是很容易看到什么会使事情重上正轨：例如，我们可以把整个文件扔进字纸篓并且假装它们从未存在过，然后我们让结算薪水支票作废，并且跑到哈利的办公桌处，在他看见之前拿走取消保险的表格……

第 18 章 整体大于部分的总和

胶冻团队是一群紧密结合在一起的人，其整体大于部分的总和。这样一个团队的产量，要比在非胶冻形式下同样的人产量大。同样重要的是，人们从工作中得到的快乐比你所期望的工作本身的本质所给予的要多。在一些情况下，面对其他人可能认为无趣的任务时，胶冻团队也会觉得是一种不平凡的经历。

一旦一个团队开始胶冻，获得成功的可能性显著上升。该团队将会势不可挡，成为追求成功不可抗拒的力量。管理这些有不可抗拒的力量的团队是一种真正的快乐。你将大部分时间用来为他们清除道路上的障碍，并保证其他人不会挡路，他们会说：“瞧，伙计们，他们来了，让开点，抓紧你们的帽子。”不必用传统意识管理他们，当然就不必激励他们。他们已获得了动力。

第 19 章　黑衣团队

从前（相对而言），在纽约州的北部有个大量生产计算机的公司。该公司也生产运行在这些计算机上的软件。该公司的客户是一些非常好的人，但是相对我们而言，在已提交的带有毛病的软件面前，他们可能是十分蹩脚的使用者。有一段时间，公司花力气培训客户以使他们能够更加容忍毛病。但是，这种方法并不奏效，因此他们迎难而上，决定根除毛病。

发现最后的毛病是困难的，但是一些测试人员比另外一些更优秀。公司将这些非常有才能的测试人员组成一组，并且给他们特权，让他们在有争议的软件上市之前进行最终的测试。黑衣团队因此诞生了。

为了加强正在不断增加的无聊画面，团队成员开始穿黑色的衣服（黑衣团队由此而得名）。程序一旦失败他们便可怕地笑着。一些成员留着可以卷成西蒙·烈格雷式的长胡子。他们会聚在一起并且研究出更加糟糕的测试策略。程序员开始嘀咕黑衣团队的变态想法。

第 20 章　团队自杀

我的一个客户是澳大利亚政府的一个代理机构。在一次咨询集会期间，我收集的数据表明，平均每个员工与四个或四个以上不同的项目有关。我向那个专员发了牢骚，他说很遗憾，但是，那种事司空见惯。人的职责被分割，因为人拥有各种技能和知识，他们需要参与其他的工作，除了分配给他们的主要职责之外。他说这是不可避免的，我说这很荒诞。我建议他制定一项同一时间一人只做一个项目的特别政策，并将政策书面写出来广泛分发下去。他乐意地照做了。一年以后，当我回到这里时，发现分给员工的项目平均数低于两个项目。

第 21 章　一顿意大利通心粉晚宴

你们出去了。作为一个小组，你们漫步在超市的通道上。没有人负责，你们的老板似乎除了晚餐之外，什么事情都胸有成竹。她说说笑笑并且讲着一个有关国税局（IRS）的故事。尽管没有一个总的指示，还是有些东西被放进了推车中。一位同事已经拿了制作精美的沙拉的生菜，有人提议做蛤利酱，没有人反对后，你的两个新同事开始谈到细节了。你决定做你拿手的大蒜面包。另外一个人拿了瓶基安蒂*。最后，大家一致同意推车中的物品足够用于晚宴了。

* 译注：Chianti，一种意大利葡萄酒

第 22 章　思想开放

你可能有机会听说有人打电话请病假，你自己可能也有几次打电话请病假。但是你听到过打电话说身体好了的吗？

情形可能像这样：你可能对老板说："听着，自从我开始在这儿工作我就生病了，但是今天我好了，而且我以后再也不会生病了。"

第 23 章　促使团队形成的亲和力

在 20 世纪 70 年代早期，一个客户公司的副总裁将旅行费用的备忘录发给了部门的每一个人。你可能收到过类似关于你自己的备忘录，但是这份备忘录有所不同。它大概是这样说的："关于旅行费用，一些人引起了我的注意，你们一直坐经济舱旅行。这不是经济舱等级公司。这是一流的公司。从现在起，如果你们坐飞机出差，就坐头等舱。"当然那份备忘录将花费很多的钱。这笔开销是实实在在的，而且你唯一能够得到的是精英意识的提高。至少有一个公司认为那是划算的。你说说看，在现实的公司中不会发生这种事情吗？这种事情在施乐公司发生了。

第 24 章 混乱和秩序

整个晚上做项目，不知为什么，反而会增加乐趣。人们喜爱找一个借口来聚在一起疲倦，推迟睡眠并且让他们的同事看到他们头发零散，胡子没刮，衣服皱巴巴的，脾气暴躁，没有任何的化装和伪饰。这使他们觉得更加紧密地被捆在一起。

"在这一事件过程中，我注意到（参赛者之一）在门厅的地毯上打瞌睡。我在那以前已认识他好多年了，并且认为他是个有点傲慢的人。但从那时起，我对他的感觉开始有所不同。我个人对他们的感觉全都发生了变化。我们已经深思过了。"

——摘自一次锦标赛的事后调查分析

第 25 章 "自由电子"

最好的经理的标志是具备这样的能力：能够挑出少数能把前景和成熟恰当混合在一起的关键人才，并使之不受约束。这样的一个经理知道他或她真的不能给这些自然的自由电子下指示。他们已经进步到了这样的程度：在公司的最好利益方面，他们自己的指示比任何可能自上而下的指示都更正确。这时，就不要挡他们的道了。

第 26 章 霍尔加·丹斯克

丹麦城市哥本哈根的正北面是科隆伯格城堡。花上几个克朗，你可以参观城堡的落地窗以及在那儿看传说中沉睡的丹麦巨人——霍尔加·丹斯克斜倚着的样子。他静静地睡着，而国家是宁静的，但是一旦丹麦处在危险当中，霍尔加会醒来，我们会可怕地看见他的愤怒。

- 一个大的政府机构的整个部门都用织物堵塞了电话铃。现在没有大的铃声了——只有轻轻的电话铃声（或者它是霍尔加·丹斯克安静的嗓音？）
- 一家加利福尼亚计算机公司，位于程序员区域的传呼系统受到一支鲁莽的游击队的袭击。从此线路一直就断了。因为程序员坐在以前是一个组装池的地方，天花板（和传呼系统的扬声器）高 16 英尺。谁能爬那么高？也许只有霍尔加·丹斯克了。

第 27 章 再论团队自杀

听起来好像稍微加点班就行了，对吗？你已经让你的团队加速，每周增加数小时的工作（仍然以同样高的生产效率），也许要工作好几个周六。只有一个问题：你的一个同事，姑且让我们称他叫艾伦，却没有你们其他人所拥有的时间上的灵活性，他是一个鳏夫，因而成了他儿子的主要监护人。艾伦每天下午五点一刻要去日托所接孩子，按照你的想像，他的周六和周日（仅有的真正和他儿子在一起的时间）是神圣不可侵犯的。

嘿，那好吧，你想，我们将为艾伦打掩护，在开始的时候我们能够理解，并且你也同样能理解……

第 28 章 竞争

一个团队或工作团组的内部竞争问题是一个复杂的问题，在这个问题上经理们的看法倾向于不一致。你肯定听说过公司通过竞争而存在的说法，推而广之，在公司里的适度竞争是保持竞争优势的一种健康方式。另一些经理得出结论认为，如果团队成员感觉他们被迫彼此竞争，那就出问题了。在极端上，至少竞争肯定抑制了团队的胶冻。例如，如果告诉团队成员，明年只有他们当中最好的一个人可以留下来，你可以肯定他们不会成功地在一起很好地工作。

第 29 章 过程改进的步骤

“女士们，先生们，我们有一些不平常的消息要宣布。我们已经完成了我们的评估工作，并且确定你们的 CMM 等级在第五级上。结果是这个公司已经证明它自己是……第六级！是的，

是第六级。在我们来这里之前我们甚至都不知道有第六级。对我们所有人而言，这是不平常的一天。我们感觉似乎我们在等级阶段表上已经发现了一个新的等级。你们是现在人类所知道的最好的软件开发公司。谢谢。祝你们愉快。”

第 30 章 使变更成为可能

我们需要认识到盲目的忠诚者造成的危险性。他们可能相当没有权力，而且他们跳到看起来很热门的任何东西上面去。他们追随时髦：“我们得马上停止会计软件包的安装，因为通过一个用 Java Decaf 编写的基于 Intranet 的系统，我们自己能够做得更好。等一下。停用 Decaf 语言。我刚才在一个叫“纳秒计算”的网站上看见一则有关 Double-Java latte 程序和 foamy applets 的广告。”他们撤回支持的速度会像他们曾经给予的一样快，他们这样做是为了准备跳到某个新的流行潮头上。

第 31 章 人力资本

公司花在人们身上的钱怎样办？一般按照通常的会计惯例，所有的工资被看成开支，从不看作资本投资。有时这样做有道理，但是有时又没道理。如果一个员工在生产要出售的产品，这不太重要：无论你是花费他的劳动力还是将他的劳动力资本化，支付给这个员工的钱的总数是从产品的销售价格中扣除，以便计算利润。你几乎不会因为像支付工厂取暖费的方式一样支付他的劳动力而被挑剔；劳动力和取暖费都是在月底用完。

第 32 章 公司的学习

我的一个客户从事软件开发工作有很长的历史，从现在算起有 30 多年了。经过这段时期，他们留用了一千或一千以上的开发人员进行工作。因此他们能毫不夸张地说他们有 30,000 人年的软件经验。这给了我很深的印象：可以想象一下，如此多的学习经验都被带到一个新项目中是什么情形。因此我问他们当中的一个团队。当派一位新经理去管理一个新的软件项目，你对他或她轻声耳语的至理名言是什么？他们想了一会儿，然后几乎齐声地回答：“祝你好运。”

第 33 章 管理上的最大恶行是……

你召集你的人员开会，而你却迟到了（因为你接了老板打来的紧急电话），其他人一直等着你。你允

许自己被人叫出去与客户进行一次简短而重要的聊天，但没有你，会议就没有了主题。或者说，可能会议本身就是浪费每个人的时间（可能你的时间除外）。在我们的咨询工作中，我们经常发觉我们自己坐着开会，从外表来看，似乎是为了完成某种工作目的，但是会议可能真正是为了一些其他的事：

第 34 章 社区的形成

将你的思绪向前推，请等一下，推到遥远的未来的某一天，你要寿终正寝了。你活到很大岁数，比如说一百零一岁，你正老死在床上。你没有什么不舒服，只是老了。在这个年纪，思绪都是在过去。你正在整理记忆。你扪心自问，我的一生中什么事情是真正重要的而什么又是最不重要的？一点也不惊奇的是多年来彻底萦绕在你脑海中的许多关心的问题（例如，使巨大的“爆竹”（Whizbang）v6.1.1 的第 27 个工作版本保持稳定）不会十分突出地出现在你的临终评价上。不，你很可能考虑到温馨的家庭关系、孩子和孙子、房子和所有有关房子的记忆。你的工作贡献呢？当然，很高兴它成为正在到来的信息时代的组成部分。能爬上公司的最高层或接近最高层当然很好，但是不要忘了你也在公司内成功地创造了一个真正的社区，一个人们喜欢、重视和忠于的事物。那么，这也是成就。就是那些在你一生做了些什么的意识中可以主要包含进来的东西。你从中得到的快乐就像米开朗基罗在反复体会他的贡献时可能感觉到的快乐一样——不必与当时的货币价值有很强的联系。这就是创造。你告诉你自己，这就是艺术，而让它发生的人是一名艺术家。

……

想像曾经是负责那件事情的人。想像一下在一百零一岁时有那样的事情可以再回首。