2002－4（第１２期）

Scott W. Ambler

X-Programmer 非程序员

软 件 以 用 为 本

主编：davidqql

审稿：jason chan，甄镭，davidqql，think

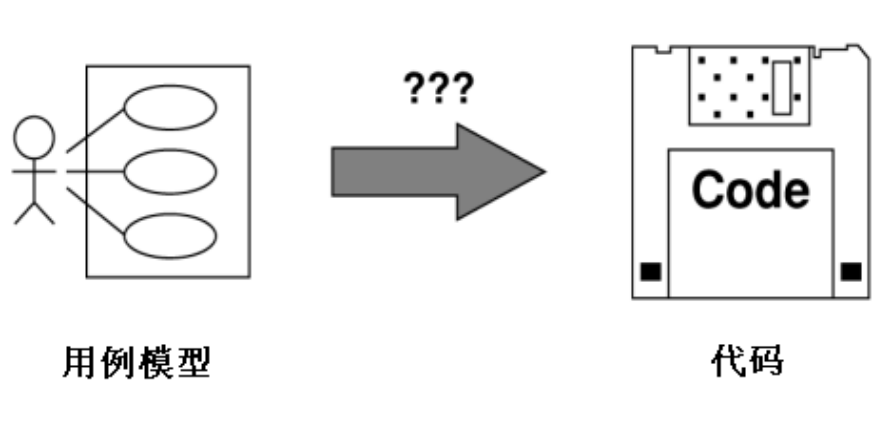
投稿：editor@umlchina.com

反馈：think@umlchina.com

http://www.umlchina.com/

UMLChina 培训

The real thing



利用 UML 的 20%就可以为 80%的问题建模

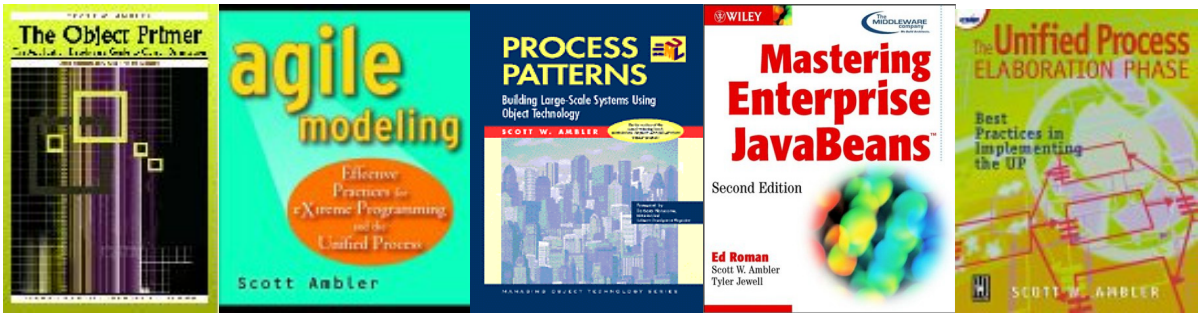
--《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

Scott W.Ambler: 空手道和太极拳

编注：

北京时间 2002 年 3 月 11 日上午 10:00-12:00，Scott W. Ambler 先生作客 UMLCHINA 讨论组的聊天室。Scott W. Ambler 是 Ronin International 的总裁，该公司是一家专门提供面向对象过程指导、体系结构建模和 Enterprise JavaBean (EJB)开发的咨询公司。他的个人网站位于 http://www.ambysoft.com/。Scott W. Ambler 的很多文章已经被翻译成中文，传播很广。以下是交流实录。由 think 和 gigix 翻译。原文链接。



gigix: 我是个敏捷建模方法的爱好者。

scottamble: 谢谢，我也是。

jay@: 请问"AM"是"敏捷制造（Agile Manufacture）"吗？

scottambler: 不。AM 是"敏捷建模（Agile Modeling）"的缩写，它是一种建模方法。AM 这个名字是在一年前由 Bob Martin 想出来的，我以前想的名字是"极限建模"，实在不太好。还有人提出别的名字，但是我觉得"敏捷建模"这个名字最好。

gigix: 您能向中国的朋友们介绍一下敏捷建模吗？

scottambler: 敏捷建模（AM），这是一种基于实践的建模方法。相关的网址：www.agilemodeling.com

gigix: 为什么您说 AM 是一种"建模方法"呢？难道它不是一个完成的过程吗？

scottambler: 的确不是，它只关注建模和文档记录。

gigix：我发现 AM 的第五种价值是"谦逊"。您能否告诉我们，为什么您要强调"谦逊"呢？

scottambler：我会强调"谦逊"，因为许多开发者都认为自己无所不知，认为自己不需要和别人合作，认为他们的用户都是傻瓜。在短期看来，他们也许可以生产出好的产品；但是长此以往，他们会害了自己，也害了整个团队。

gigix：那么，按照 AM 的观点，您怎么看待程序员？程序员的角色是什么？

scottambler：我喜欢把所有的人都看成"开发者"，而不仅仅是"程序员"。我希望他们都有建模能力、编程能力和测试能力，这样他们就是非常有价值的项目成员。但是，做程序员是开发者生涯最好的起点，不过要记住：开发远不止是编程。

myou：我是一个博士研究生，想做 EJB 和 AM 方面的学术研究，请问您有什么建议？

scottambler：千万别拿 EJB 和 AM 去做博士研究项目，把它们当成喜欢的东西去探索一下就行了。

higoals 说：在您看来，AM 的未来是怎样的？

scottambler：我希望未来会越来越好。我想，许多其他的方法都会将 AM 融入其中。Pretince Hall 出的一本 XP 的新书里面，就专门有一章讲 AM。我希望到明年能让各位看到一本完整的 AM 书，请耐心等候。另外，到 UML 2.0 发布的时候，我会重写 Object Primer(www.ambysoft.com/theObjectPrimer.html)，详细讲述 UML 2.0 和 AM。

myou：您是否同意：AM 属于管理研究领域？

scottambler：我觉得 AM 属于软件开发或者软件工程领域。

gigix: AM 说"只有到破坏的时候才去更新"，难道它不进行重构吗？

scottambler：这是一种文档实践，而重构是代码实践。你可以对代码进行重构而不更新模型或者文档。

415918：我读过您的一些关于 EJB 的精彩文章，例如《Mastering EJB 2》。您对 EJB 怎么看？

scottambler：我觉得 EJB 是一种很好的技术。

415918：但是在中国，很多采用 EJB 的项目都失败了，一些厂商使用 EJB 来开发 ERP，但是……

scottambler：为什么这些项目会失败？

415918：性能太差。

scottambler：性能差在哪儿？是因为 EJB 吗？还是来自糟糕的数据库设计？

415918：我想的确存在数据库设计失误和不正确使用 EJB 的问题。

scottambler：设计对象的人和设计数据的人必须协同工作。如果他们不这样做，就常常会造成项目的失败。我也在 www.sdmagazine.com 上写过关于这个主题的一些文章。

415918：在加拿大和美国有成功使用 EJB 的 ERP 或者 CRM 案例吗？

scottambler：我想可能有，不过我从来没做过 ERP，所以也不知道确切的情况。

sliuhao：您好，我是一个 Java 开发者。我对 persistence layer（例如 JDO 或者 ODMG API）很感兴趣，您能告诉我它在加拿大的应用情况吗？

scottambler：在加拿大，我在好几个项目中使用了 persistence layer，它们都工作得很好。如果有超过 40 个业务类，我就会考虑使用 persistence layer。

Charity_Zhou：您好。我正在做一个项目。我希望能使用 J2EE，但是大多数的人都是 MS 的追随者，我应该怎么办？

scottambler：对于你的这个项目，跨平台能力很重要吗？如果不是，你就没必要叫他们改变习惯。

jay@, Charity_Zhou：请问您对 J2EE 和.net 的未来有什么看法？

scottambler：我想，它们都是非常好的技术，都将存在相当长的时间。

dejol：据我所知，有些开发者在项目中不使用 EntityBean，您对此怎么看？

scottambler：在实际的应用中，EntityBean 真的没什么用。大多数人都使用 Session Bean 和普通的业务对象。

gigix：那么，UML 在 AM 中有什么作用呢？

scottambler: UML 定义了几个可以创建的 artifact。不过，有没有 UML 的 artifact，你都可以使用 AM。我曾经写过一篇文章，从现实的角度来观察 UML，你可以在 www.agilemodeling.com/essays.htm 找到这篇文章。

gigix：如果不使用 UML，那您会选择什么建模语言呢？

scottambler：比如说，你可以用 AM 的方法来做数据建模。使用 COBOL 的结构化项目可能需要使用数据流图、数据模型和结构图。

gigix：也就是说，AM 并不是纯面向对象的方法，对吗？

scottambler：对。不过大多数的人还是会把它和 OO、基于组件技术一起使用。

davidqql：我的一些同事认为 UML 太复杂了。我应该怎么对待它？

scottambler：UML 的确太复杂，但同时又不够大。在 UML 2.0 中会有重大的改进，其中之一就是定义大约 20%的核心符号。你可以先从这些核心符号开始。

gigix：您对敏捷方法怎么看？它们最适合于哪种情况？

scottambler：敏捷方法应该用在需求了解不充分、有可能变化的情况下。另外，你的团队应该足够小，并且需要和用户有足够的沟通。

davidqql：您写过什么关于敏捷方法的文章吗？

scottambler：我在 SD magazine（www.sdmagazine.com）上写过一个专栏，关于敏捷开发的。另外，还有一些资料在 www.agilemodeling.com 上。

sprighu：请问如何在嵌入式开发领域中使用敏捷方法？

scottambler：必须小心！我从来没做过嵌入式开发，所以也无法给你多少提示。对不起。

normanwu：请问如何在一家传统的软件公司中引入面向对象的方法？

scottambler：任何一家公司，要引入变化都是困难的。我会从小规模开始，逐渐扩大范围。

lzhihua：您认为用例（use case）是收集需求的好方法吗？

scottambler：在收集使用需求方面，用例是非常好的工具，但是对于其他需求就不那么好了。用例是整体设计的一部分，所以 AM 告诉你：应该使用多种模型，这样才能得到好的效果。

cliffxiao：用例不适合哪种需求？

scottambler：技术性需求、业务规则和约束、UI 需求等都不适合用用例来捕捉。

lzhihua：您上面提到的这些都是非功能性需求，对吗？

scottambler：对，非功能性需求。

cliffxiao: 刚才您说"用例不适合用来捕捉业务规则"，那么有什么别的方法可以有效地捕捉业务规则吗？

scottambler: 有的。通常我会把业务规则记录在用例之外，在用例中引用它们。Word 或者文本编辑器都很管用，还有索引卡。

dejol: Entity Beans 并没有带来更高的效率，您认为它的未来会怎样？在 EJB2.0 里有什么新特性吗？

scottambler: 我想在很长一段时间内，他们将会继续保留 entity beans，因为对低性能应用和原型来说，它们很管用。

gigix: 重构并不改变程序的行为，所以我可以不改动文档。

scottambler: 重构改变了设计，所以你可能需要更改你的设计文档来反映代码的变化。

ying_mo: 作为一种方法，AM 是独立于语言的，是吗？

scottambler: 是的。我认为 AM 是独立于语言的。你可以用在 Java 项目、COBOL 项目或 C#项目中，它也独立于建模语言，如 UML。它是少数不阐明具体工件的建模方法之一，例如你要建立的用例或数据模型。它很象 XP--XP 没有说你必须使用 Java。AM 致力于解决如何有效率地进行建模和写文档的问题，这些问题是开发人员一直在苦苦斗争的；另一方面，UML 致力于解决哪种标记和语义应该应用到 OO 建模和构件化软件中的问题。每种技术都是为了解决不同的问题，并互相形成补充。

myou: 有什么工具支持 AM 吗？

scottambler: 很难回答的问题。你可以用一种"敏捷"的态度使用工具，或者不使用工具。白板和索引卡很有效，能产生代码的 CASE 工具也很好，重要的是工具是否给你的投资带来了好的回报。

higoals: 有什么讨论 AM 的聊天室吗？

scottambler: 我不知道聊天室的知识，但我有一个 AM 的邮件列表，www.agilemodeling.com/feedback.htm，列表上有来自世界各地的人。

jay@: 我非常崇拜您，我想知道，除了技术之外，您的其它兴趣是什么？

scottambler: 好问题！我学习刚柔流空手道和太极拳，也很喜欢风景摄影。我也喜欢旅行，去年我去过南极，希望有一天会去中国。我很想看到长城、沙漠还有很多其它东西。

lzhihua: AM 对架构是怎么看的？

scottambler：架构很重要。我写了一篇关于 AM 和架构的小文章，在 www.agilemodeling.com 上，可能在 www.agilechina.com 也有吧。

ying_mo：据说重构对 Java 应用更有效，您如何看？

scottambler：大多数重构目前都是针对 OO 代码，例如 Java 和 C++。但是，这种技术也可以应用于非 00语言。有一些关于数据重构的工作，但很麻烦。

gigix：有一天会来中国吗？

scottambler：我一定会去的。去年我本来有些事情要去北京，但由于 9.11 取消了。以后我一定会去度假的。

Charity_Zhou：我想继承树应该保持在 2 至 3 层以内，否则映射到 DBMS 时会出问题。

scottambler：这取决于你如何映射层次。有三种主要的方法，但层次越深，性能越受影响。

ying_mo：重构还不太成熟，是吗？

scottambler 对 ying_mo 说：很多人非常成功地应用了这种技术，我想重构是相当可靠的技术。

gigix：你想对中国的程序员说些什么吗？

scottambler：我想对程序员来说，最重要的就是保持学习。注重编程以外的更多东西，学习建模技巧、沟通技巧、商业技巧和与工作无关的东西。

lzhihua：需求的改变是否会影响架构？

scottambler：需求改变是现实，你必须接受这一点并采取相应的行动。我建议不要过分设计架构，不要过分建模，不要过分写文档，否则需求的改变真的会影响你。

jay@：我想，您一定是一整天都在思考，您觉得累吗？象一个哲学家一样思考？您喜欢哲学吗？

scottambler：看我的心情了，我想。自己独处的时候，我喜欢以哲学的方式思考。我发现我的一些最好的思想是在飞机上或驾车的时候得到的，可能是这个时候，我的大脑有机会冷却下来。软件开发是我的主要工作，写作是业余爱好。因为我是作咨询的，我经常是在拜访客户的路上，所以我有很多空余时间，把我正在做的事情写下来。我也会提前很多年计划一本书，然后慢慢记录心得。例如，现在我正在做一本关于 UML 建模风格的书，相关信息在 www.modelingstyle.info，后面还有一本新的 AM 书在等着。

ying_mo：我对重构很感兴趣，但很少有相关的书。我怎样开始学习重构呢？

scottambler：可以从 Martin Fowler 的书和网站开始。XP 社团也会经常谈及。据我所知，在 5 月份即将到来的意大利 XP 2002 会议上，重构将成为 XP 的重要主题，我想 8 月的芝加哥 XP/Agile 全体大会上也一样。关于这方面的论文会不断出版，信息会越来越多，请耐心等待。现在也有一些有趣的重构浏览器（refactoring browsers），例如 IDEA for Java，值得一看。

simaetin：学习建模的最好方法是什么？开始一个新项目还是在遗留项目上提高？

scottambler：我想，这与人有关。一些人喜欢新事物，另一些人喜欢遗留项目。遗留建模是一个很少有人写到的领域，我打算把它作为我的第二本 AM 书的主题之一。过些天我可能会在 www.agilemodeling.com 上发表一两篇关于这方面的短文。

Charity_Zhou：我们现在需要用到.net 吗？

scottambler：为什么不呢？微软有最多的开发工具供你使用。

simaetin：项目越大，开发人员学到的东西越多，对吗？

scottambler：要看情况而定。在小的项目组里，你要做很多事情，因为没有很多人。在更大的组里，你可以关注某项特定的活动，你对这项活动的熟练程度远远高于其它活动。

jay@：随时进行哲学思考？您说您经常要和客户交谈，有没有感到困惑？可能有一些客户很难沟通，特别是在您的层次上？

scottambler：这总是一个问题。我发现，我从客户那里得到的最重要的思想是他们问的问题，以及他们正在苦苦挣扎面对的难题。这也是前些年我关注对象映射到关系数据库的原因，在和他们谈话之前，我不知道有如此多的人在这个问题上挣扎。

simaetin：我同意。但实践中，很难为小项目找到有丰富经验的、开发人员能从中学习的 mentor。

scottambler：是的，小团队经常没有他们所需要的 mentor。

ying_mo：我想重构，还有 AM，现在在中国还不是广为人知，我还没有看到 Fowler 先生的书的中文版。

scottambler：我不知道这本书是否已经被翻译，如果没有，可能是某些人的一个机会。

Charity_Zhou：我确实感到微软有一些很好的想法，有一些很好的产品，象 Office。但还没有使用.net 的成功项目，这就是问题。

scottambler：是的，我们把这称作北美的"第 22 条军规"。你必须等待和丧失一些市场机会，或者冒使用.net 的风险和微软下一轮工具会带来改变的风险。

Charity_Zhou：根据您的经验，Java 在 Linux 上运行的性能如何？据我所知，Linux 没有虚拟机，并不真正支持线程（它把线程当做进程处理），使得性能下降。

scottambler：我的公司 Ronin International, www.ronin-intl.com, 在 Linux 上运行 Java 方面有很多经验。大多数 Java 性能问题，至少在商业应用中，是来自糟糕的数据库设计，与纯 Java 问题无关。Linux 没有虚拟机？我觉得这个说法不对，可以在 java.sun.com 上走走看看。

jay@：噢？那么你的客户是开发人员？

scottambler：和项目有关。通常我会和各个层次的人一起工作，从开发人员到管理层。

simaetin：请问您的年龄？如果不是冒犯的话。您现在还编码吗？对普通人来说编程工作的年龄极限是多少？

scottambler：不冒犯。我 35 岁。我也做一些编码工作，但没有我想做的那么多。我通常都会在一个高级人员的角色，不需要象过去那样写那么多代码。这是个问题。

ying_mo：有没有专注于敏捷方法方面的书？I

scottambler：有一些致力于敏捷开发的书，参见 www.agilemodeling.com/resources.htm。AM 的书在两周内将在北美上市。

lzhihua：我想，要想避免过分构造，必须要先知道什么是过分构造。能给一些建议吗？

scottambler：过分构造（Overbuilding）是指你把并不是马上需要的特性放在架构中，或者使得它们比真正需要的还要复杂。很多开发者致力于得到一个"真正酷"的东西，结果就是过分构造。一个常见的例子就是在项目初期花费几个月来建造框架和可复用代码库。

xkfy：哪种编程语言你最喜欢？

scottambler：我喜欢的语言是 Smalltalk，虽然这段时间我的绝大部分工作是在 Java 上。我开始注意 C#，但还没有用在项目上。

Charity_Zhou：我更喜欢用 Linux 作为应用服务器和数据库服务器，它是一个廉价的选择，但我感觉这也是一个糟糕的选择，因为用户和维护人员只有少量在微软平台上的经验。

scottambler：这和你的环境有关。如果每个人都擅长于某个平台，那么你还是跟随它为好。

ying_mo: C++怎么样呢？这是我的最爱！

scottambler：我过去常用 C++，它非常好，还有以前的 C 也是。我甚至用过 COBOL 和 Fortran 来完成工作。

lzhihua：谢谢。但另一方面，我担心糟糕的设计会导致重构成本过高，这么看来重构只适合很好的团队。

scottambler：你用来重构的设计不应该太差。如果你有一个很差的团队，我想，用什么方法都会有问题。

jay@：有句话说"天底下没有新鲜事"，您认为我们可以从古代哲学得到所有的方法学吗？例如希腊哲学和中国哲学？

scottambler：我相信会有一天有人写一本书叫"敏捷开发之道"。你可以把现在的工作映射到古代哲学。一个优秀的例子是孙子的著作已经应用在商业书籍中。我打堵，如果他活到今天，他会为此开怀大笑的！

ying_mo：我看过"Effective C++"(Scott Mayers)，我认为这本书很有趣，而且从中学到了很多东西。

scottambler: Effective C++和它后续的书都非常优秀。

xkfy：您能对比一下 C++ 和 Smalltalk 吗？

scottambler: Smalltalk 比 C++更高级，因为它是纯 OO 语言。C++提供更好的性能，和硬件靠得更近。但这两种语言都已经被 Java 超越，虽然 C++依然流行，Smalltalk 却已经快死了，很不幸。

xuxu1976：您怎么看 RUP？

scottambler：RUP 在合适的环境里非常优秀，但你必须有一个项目团队，并遵循 RUP 的指示。在软件工程里有很多东西，如果你用错了，将不会有效果。其中有很多方法需要实施，显得很重，这是它的缺陷。

Charity_Zhou：我们能用 COM+/DOCM 代替.net 吗？

scottambler：看你想做什么了。微软看来已经偏离了它们，要小心。

ying_mo：我已经学了一些设计模式方面的知识，您怎么看设计模式？

scottambler: 模式很重要。AM 包括一个实践叫做"优雅地应用模式"。

ying_mo: 优雅地应用模式？

scottambler: 基本的意思就是说，在你想到要使用某个模式的那一刻，不用马上去应用它，而是等到你确定为止。例如，可能你有一个公式要实现，你可以立刻使用策略模式，但策略模式可以适用于多个公式。你可以等待直到你有第二个要实现的公式，这时考虑重构你的代码，使它容易实现策略模式。当第三个公式出现时，再重构你的代码，使它更容易实现策略模式，需要实现第四个公式的时候，你就可以正式地考虑采用策略模式了，因为这个时候，它是最简单的方法。如果你当即使用策略模式，但只实现一个公式，你就是做了一些无用的额外工作。

xuxu1976: 对 10 个人的公司，什么方法是合适的？文档重要吗？是否应该把文档加到代码里？

scottambler: 在 10 个人的公司，我可能会尝试一些类似极限编程或功用驱动开发（Feature Driven Development）的方法。没有足够具体的环境，很难说清楚。关于文档，我更喜欢把大多数文档放进代码里，但你仍然需要外部文档，例如用户手册和系统概述文档。

higoals: AM 的本质是什么？

scottambler: AM 的本质？AM 是一个前所未有的东西，它是另外一种看待模型的方法，我想这就是它的本质。保持事物简单，关注真正的需求。

xuxu1976: 可以使用轻量级方法进行组件开发吗？

scottambler: 当然可以使用轻量级方法进行组件开发了，很多人使用 XP。

simaetin: 我们不应该在"抽象"的时候实施模式，而是留待后面来重构，是这样吗？

scottambler: 没错，等到你确实需要实施这个模式的时候。

ying_mo: AM 是实用而且面向现实的，是吗？

scottambler: 我希望 AM 对开发有帮助，我已经发现它对我正在做的项目很有帮助，看起来我的客户也已经熟悉了这种方法。时间会证明一切。我在尝试尽可能保持 AM 的实用，很多学术的理论感受到了 AM 的威胁，这可能是 AM 实用性的一种征兆。;-)

xuxu1976: 您能给我一个使用轻量级方法进行组件开发的网址吗？

scottambler: 我不知道。但你可以在 XP 邮件列表上发一个帖子，我想有人会指点你的。

simaetin：您能展望一下软件开发的前景吗？

scottambler：我想，软件开发将会越来越敏捷，但需要一代人的时间。在北美，我们喜欢说，钟摆现在摆到另一边了。大概需要 20-30 年的时间。软件开发曾经变得越来越重，现在正在变得越来越轻。时间会证明一切。技术方面，我想 Java 和 C#等 OO 语言将在很长一段时间内存在。Web services 很有趣，但并不是每个人的救世主。

ying_mo: AM 更适合小团队和频繁变化的环境，是吗？

scottambler: AM 最适合于沟通密切的环境，倾向于小团队。不过如果你的需求不是频繁改变，可能不需要 AM 的迭代和增量特性。

gigix：在软件开发中，更轻意味着更安全？为什么？

scottambler：更轻在某些环境下意味着更安全，更轻意味着更少的文档，更关注切实地开发软件。AM 的原理之一就是"你的首要目标是软件"，意思是你应该关注手上的工作，而不是写文档或工作状态报告。太多的人忘记了这一点。

simaetin：那根据钟摆定律，重型方法迟早会因为某些原因回来的。

scottambler：可能吧，不过那时候我已经退休了！重型方法不会消失的，在某些情况下，它们很适用。我做了一些称为企业统一过程（Enterprise Unified Process，EUP）的工作，详情请看 www.ronin-intl.com。当 RUP 被扩展以包括一个新阶段或两个新的规范/工作流的时候，就会相当的重。

fly_sky: XP 是一种暂时的技术，还是编程的新方向？

scottambler：我想 XP 是真实的、现实的，它将随时间发展。

alou：我想，很多人相信 AM 适合我们，是因为我们很懒。

scottambler：很多人相信，最好的开发人员是很懒的。你应该只是因为 AM 是你最好的选择而使用它，而不是出于逃避写文档。事实上，AM 的书和网站也谈了相当多关于文档和如何有效书写文档的知识。很多开发者会发现 AM 要求他们做比以往更多的建模工作。

simaetin：以 Martin Folwer 的观点，敏捷方法的本质是 adaptive 和 people-intensive，这两点和以往的老方法有很大差异。

scottambler：确实如此。www.agilealliance.com 有文章谈及这一点。

ying_mo: 我对 AM 很感兴趣，希望能应用到实际中！

scottambler: 最好的方法就是在你的下一个项目尝试 AM。

jay@: 最好的开发人员很懒？有趣！

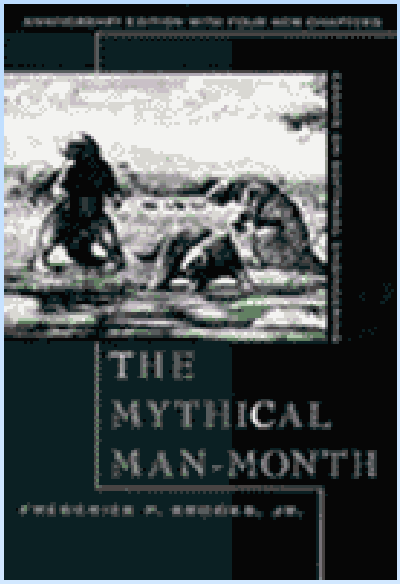
scottambler: 懒人通常会寻找更好的做事方法，所以他们要做的事也就少。顺便说一下，我马上就要结束这次交流，在我离开之前，哪一位有什么重要的问题要问我吗？

scottambler: 我希望有一天会去中国，到时候也很希望见到你们中的某些人。

欢迎访问

www.umlchina.com

《人月神话》



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 Adams Wang

散文笔法，绝无说教，大量经验融入其中

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

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

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

中文译本即将发行！

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

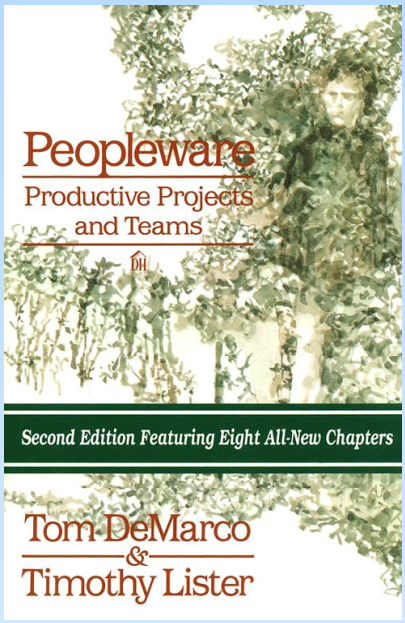
翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

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

中文译本即将发行！

《人件》



《人件》第 2 版

Tom Demarco 和 Tim Lister

翻译：UMLChina 翻译组方春旭、叶向群

微软的经理们很可能都读过—amazon.com

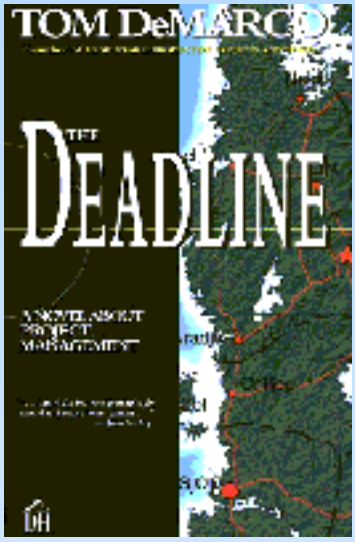
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

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

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

中文译本即将发行！

《最后期限》



《最后期限》

Tom Demarco

翻译：UMLChina 翻译组 透明

这是一本软件开发小说

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

....

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

莱克莎点点头："这是最近的数字。他们都会在你的手下工作。"

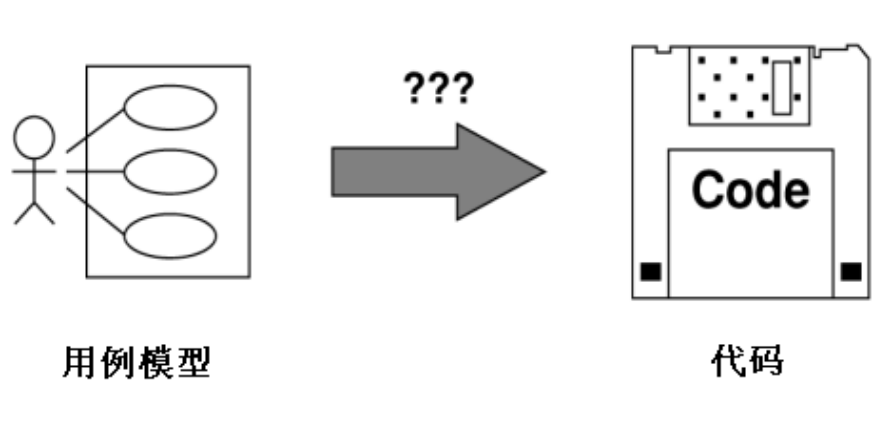
"而且据你所说，他们都很优秀。"

"他们都通过了摩罗维亚软件工程学院的 CMM 2 级以上的认证。"

中文译本即将发行！

UMLChina 培训

The real thing



利用 UML 的 20%就可以为 80%的问题建模

--《UML 用户指南》，第 32 章

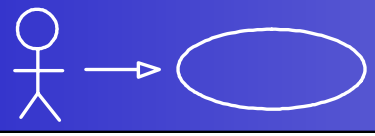
详情请垂询：think@umlchina.com

面向对象？

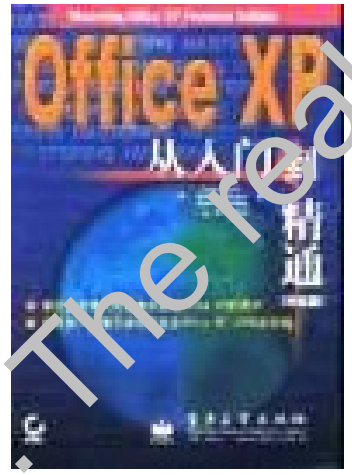
我们编写COBOL程序。以前我们用COBOL编写COBOL程序，现在我们用C++编写COBOL程序。

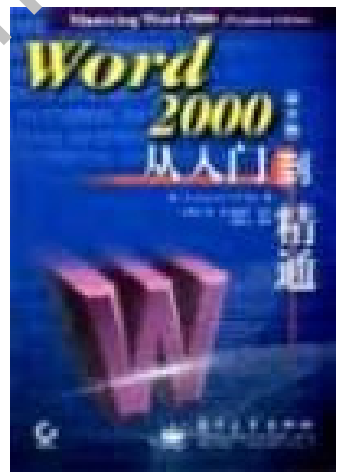
--《实用面向对象软件工程》

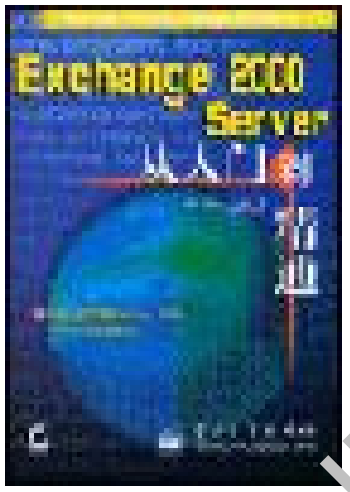
Edward Yourdon

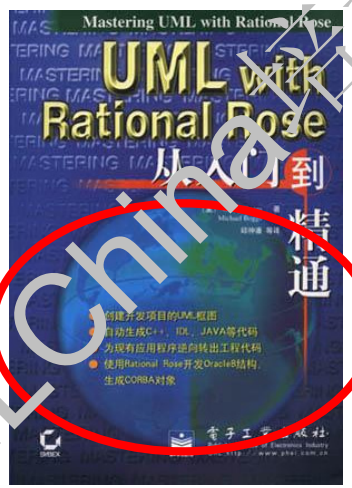


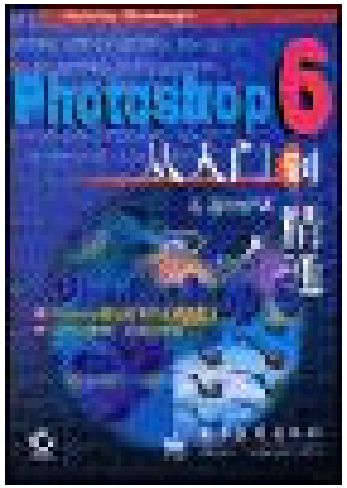
这就是面向对象？

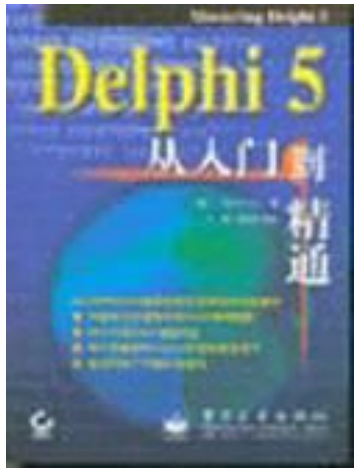


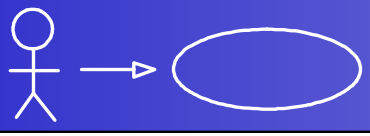








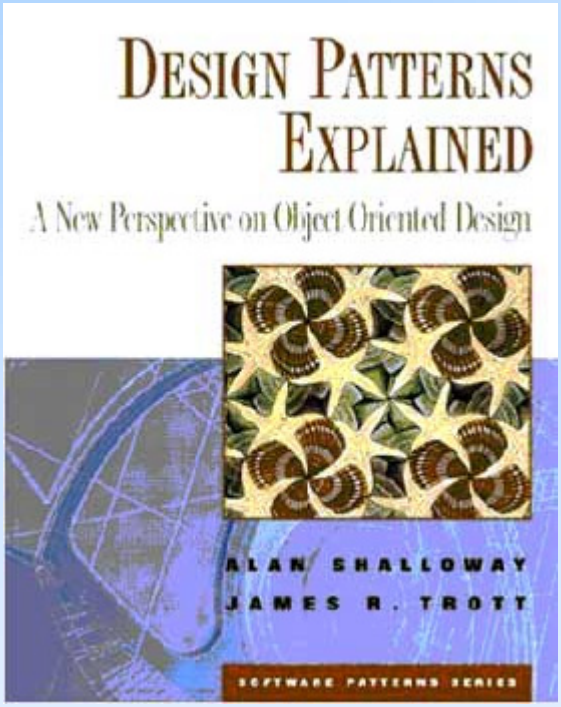




看不懂《设计模式》？

它的作者推荐《设计模式精解》

透明 译



直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 **Alan Shalloway** 的《设计模式精解》。

--John Vlissides 写于 UMLChina 答疑板

中译本即将上市！

发明软件

Larry Constantine 著 zhen_lei 译

——创新是一个过程，不是一种产品。一些软件公司宁愿称自己为创新者，却不愿做实际的创新工作

任何软件公司都希望开辟新天地，只是他们不想成为先驱。在这样一个创新占统治地位，每个新的“.com”公司都把创新（first-to-market）作为自己的战斗口号的时代，令人吃惊的是，很多软件开发者和他们的经理实际上破坏了他们自己领导创新之路的创造性潜力。实际上，一些公司只愿称自己为创新者而不去做创新的事情。

不像好莱坞和电视剧表演的那样，创新不是偶然的。发明家很少仅由于运气而偶然获得好想法。发明有用产品的多产发明家知道创新突破出现在特定的情况，产生于特定的实践过程。有用的发明经常从边缘而不是中心切入。创新成功于车库实验室，伴随艰苦的工作，产生于聪明的、不遵从传统的团队。在一定的条件下，实际的创新可以根据要求产生，而且这种成功可以一次又一次地重复出现。其中的秘诀是认识到创新是一个过程而不是一种产品，这几乎被业界所有的创新团队所认识。

多年来，我的公司和用户采用这个秘诀有计划地创造用户界面突破。最终，经我们训练、指导或与之合作的团队生产了一系列软件专利应用，其中的一两项被认为是革命性的图形化用户界面。

我们没有自大地认为仅有我们发现了这个规律。实际上，我们所采用的创造工程过程是可以学习的和传授的，对此，我们深信不疑。最近，我们甚至推出了一个叫做“发明界面”的研讨班。这是一个实验。使我们吃惊和高兴的是，想要参加这个课程的人数超过了房间允许的限制，而且还得到了许多热情的评论。尽管如此，它只是那些真正揭示当今软件工业所面临问题的评价方式的脚注和说明。让我们看一下阻止软件发明的一些内在因素。

寻找答案

我们的“发明界面”课程被策划为集中于动手实践创新过程的研讨班，但一些参加者抱怨将时间花费在学习、实践和采用创新工程技术上。他们希望更多地看到其他人创新的用户界面实例。

画家通过绘画学习怎样绘画。创新工程要通过创新工程过程来学习。这看上去理所当然，但是模仿是创新的对立面。我可以给你看很多创新的用户界面，但你仍然不知道设计者是如何达到这些创新的。如果你希望购买一个现有方案，仔细查看其它人的创新作品是重要的。如果你希望创造些新的并且可以真正工作的产品，你需要知道如何自己去做。

正如一个有关捕鱼的谚语所说的，给人现成的鱼，只够他们一天的生活，而教会他们捕鱼，可以使他们生活一生。我们课堂上的问题是，我们是老师而不是渔夫，但有些参加者只想带几条鱼回家。

对我来说，编程——这种对我们来说首先是职业的工作——的真正乐趣在于解决问题，把事情想清楚，而如此多的程序员都一直是在寻找简单的答案。当经理们一直注意他们的竞争对手正在做什么，而不是思考他们自己应该做什么时，这个错误就更严重了。

如果你寻找答案，特别是简单的答案，你通常能得到的最好结果是从解决别人问题的陈旧方案演变而来。实际上，做好你自己的家庭作业，彻底研究那些以前的问题，甚至是有时和你刚好相反的问题，特别是当你需要探索新领域的时候。

我有一个终生的习惯，在看别人的解决方法之前，自己动手尝试解决问题。在我自己努力解决一个问题，得到一个解决方案或者至少发现一个方向后，我会反过来检查自己的思路，与其它人的工作进行对比，而不是把自己陷入到前人的工作中。我经常发现许多事情在我看来是显然的，别人往往会忽略掉。好！又是一个突破。广泛的研究和背景资料的学习对于书写学术论文或学位论文可能是正确的工作模式，但在真实世界中，这会干扰你的思路，将你引入其他人已走过的路。

当然，应该去看别人是什么做的，这是有好的理由的。这些理由包括事后的对比，检查自己的思路与其他人的不同，并且发现有用的补充和创造性的提高。在工作之前，你应该仅为获得灵感而不是寻找答案而去看其他创新设计者对相似问题的解决方案。

只有一条出路

查看其他人的工作的第三个理由是可以帮助打破只有一种方法或只有一种正确方法的思维模式。尤其针对开发经理，需要学习不去接受程序员们的肯定说法。“这可能不灵活（或效率不高，浪费资源，粗糙），但这是采用 Java（HTML、C++或 API、MFC）仅有的一种方法。”，程序员说。“胡说，”精明的经理说，“回去，找到另一种方法。”

更糟糕地是，程序员会这么说，“无路可走！你不可能在 Windows 中实现这个！”在编程中，没有不可能的事情。在软件中，所有事情都是可能的，只是你愿意在上面花费多少时间和精力的问题。一个可接受的回答是：“根据当前给定的交付计划，我不能为解决它去寻找一个方法。”这个回答说明了问题的真正实质：没有找到方法。

这种“只有一种方法”或“没有办法”的想法将会限制你，阻碍你发现解决难题的创新方法。我们客户的一位有才华的程序员最近为一个创新的工业自动化工具制作一些 Windows 控件。不幸的是，他的标签对话框不能像新应用所要求的那样工作。当这个问题被提出后，他绝对地坚持那个要求是不可能实现的。我从事编程工作是很

多年前的事情了，但我同样绝对地肯定，事情不是他说的那样。实际上，我不熟悉 VB、MFC 及 OLE，但我具有理解问题实质和用户界面编程逻辑的经验。在这个问题上，我“发明”了一个技巧，逻辑告诉我它行得通。几周内，我收到了一个 Email，告诉我所建议的技巧管用了。

传统的镣铐

传统的影响加重了许多开发员“无法实现”的心理障碍。传统的用户、技术和特性经常阻碍前进的脚步，因为 Microsoft 在 Windows 中这么做，或是因为产品前一个版本的错误做法，你就有理由在下一个版本中不做更改。

人类历史上的几乎所有进展都受到人们传统观念的抵制，人们认为老的方法挺好甚至更好。我不是指我们业界中的 6 个月就出现一组新名词这种“进步”，我指的是那些可能产生新事物或改进旧事物的真正进步。

对许多软件开发员来说，用户的建议是仍然基于老的软件或系统。如果你错误地询问你的用户或顾客，他们是否愿意使用老的用户界面，大多数人会回答接受这个建议。多数人宁愿忍受已知的痛苦，而不愿面对不熟悉和不确定。“至少我知道现在系统的毛病。”问他们，他们就会建议你参考已有的某个笨拙的 Windows 应用程序。别听他们的。如果我们总是听从这样的用户建议，我们可能还在使用模拟电子打字机的行编辑终端呢。

已经安装的旧系统是一个在做设计决定和规划业务策略时必须要考虑的实际问题。但这个问题已被胆小的经理和设计者夸大为一个影响整个软件的巨大问题。除了很少的例外，大多数传统的东西最终会让位于更好的替代品。驾驶“没有马的马车”采用的方向控制和绳子，尽管对于传统用户来说非常熟悉，最终还是让位于能很好控制方向的方向盘。只要时间足够，大部分现在使用的令人厌烦的 Windows 应用程序都会成为历史。

关于传统特征和传统界面技术的规则实际上比较简单。经理们不要将是否继续支持传统用户作为一个问题，而只是决定什么时候停止支持他们。只有采用这种方法，你才可以创新并且进行有效的过渡。

与旧系统很相近是完美的神话。沿用传统界面是因为（争论来了）市场的力量，用户不断增长的老练程度和技术进步已经消除了 Windows 的糟粕。这种争论错误地将现在的先进水平等同于最好。这种状态的辩护人经常忽视他们争论的前提。实际上改进从未停止，市场从未停止选择，用户从来没有被固定到一种图形用户界面。仍然有进步与革新的巨大空间，如果你的新产品确实不错，并且有合适的销售方法，市场就会接受它。

那些具有勇气和创新能力的开发员会引导这个方法。如果你真的相信目前的方法是唯一的，并且一直会这样的话，只要比较一下 Windows 3.1 和 Windows 2000 就可以了。 Microsoft 明白，你和你的用户也能明白。

放弃

作为用户界面设计者和设计公司的合作者，我的经验是，好的想法很少是一蹴而就的。即使是最肥沃的土地也需要耕作才能获得好的收成。

在软件中，创新也是如此。许多富于创造性的革新者开始于正确的方向但没有得到结果，因为他们在出现一点问题或发现一点失败的迹象时就准备放弃他们好的想法。不断产生软件专利的创新小组知道，不要过早放弃一个想法。他们将最近发现的缺点、局限、缺陷作为需要创造性解决的新问题。

如果需要是创新之母的话，顽强的坚持不懈就是接生婆。另一方面，如果你想产生软件创新，你必须学习什么时候改变战略。实践中，你会认识到，在重新前进之前什么时候后退和前进以获得对问题的新视角。

未定义的问题

为了解决问题，你要知道需要解决的问题是什么。创新重点放在采用正确的方法定义问题上。例如，每个人都知道多行标签对话框难以使用。甚至 Microsoft，尽管可用性实验室里挤满经过高度训练的专业人员，仍采用使用户感到迷惑的多行标签(multi-row tabs)对话框。可用性领袖 Jared Spool 得出结论，没有好的解决方法，最好不去用它。

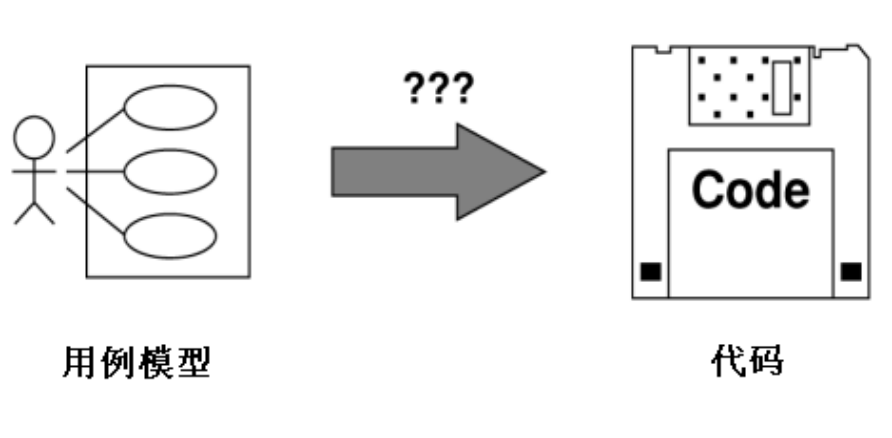
但问题是什么？问题不是多行标签而是它们的行为难以向用户解释清楚。程序员坚持认为，当你点击一个处于后面的标签，系统的反应是简单而合乎逻辑的，但对用户来说这个反应好像是随机的，好像行和标签之间发生了互换。

将这个问题作为一个视觉感知问题——用户没有能力理解或感觉实际发生了什么——为找到可行的方案指出了方向。采用动画简明清楚地解决了这个问题。使被选中的行弹到最前面，使其它行隐藏到它的后面，你就获得了一个多行标记对话框，它可以完全被理解，甚至更“直观”。

这个问题解决了，还有其它软件问题，它们正等待有人努力地发现好的解决方案，那可能就是你和你的团队，为什么不呢？

UMLChina 培训

The real thing

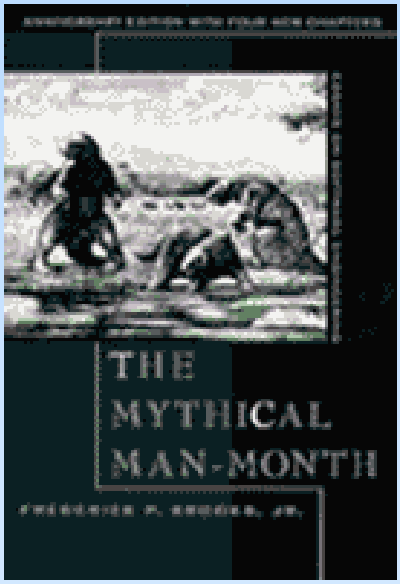


利用 UML 的 20%就可以为 80%的问题建模

--《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

《人月神话》



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 Adams Wang

散文笔法，绝无说教，大量经验融入其中

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

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

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

中文译本即将发行！

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

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

中文译本即将发行！

软件设计模式的非软件例子

Michael Duell 著，Wu 译

摘要

软件设计模式来源于 Christopher Alexander 的建筑学模式和对象运动。根据 Alexander 的观点，模式就是一个对于特定的系统的通用解决方案本身的重复。对象运动关注于将现实世界模化为软件内部的关系。基于这两个原因，软件设计模式对于真实世界的物体而言同样应当是可以重复的。这篇文章呈现了现实的世界中的非软件的模式实例，这些模式来源于《设计模式—可复用面向对象软件的基础》（*Design Patterns - Elements of Reusable Object-Oriented Software*）[13]一书。这篇文章也举例讨论了模式语言对非软件的表现力和设计模式的练习。

简介

在软件行业中，模式支持者的团体正在扩大。模式发展的起源可以在建筑师 Christopher Alexander 的著作中找到，他认为模式是世界上特定系统的通用解决方案。他描述的模式可以在日常的建筑物中观察到。《模式语言》（A Pattern Language）[2]中的每个模式都包含了一张该模式原始范例的图片。

虽然物质是主流世界的观点，而模式为软件世界所信奉，模式也有其体现事物发展的根源[9]。不幸的是软件设计模式的例子不象 Alexander 模式那么丰富，因为软件设计表现的是精致的构思而不是那些最初产生的想法[13]。当今大多数软件的专有性限制了我们接触一流设计的机会。

根据 Alexander 的说法，现实世界中模式总是重复自己，因为在一个特定的环境下，它们总是很好地适应现有的环境因素[1]。在软件中，要么现实世界的问题被完全地模式化，要么现实世界的物体被转换成为硬件和软件，用来产生现实世界的结果[5]。既然软件设计模式根源于 Alexander 的样式和对象，那么在现实世界中找到软件设计模式也是很正常的。这并不是说软件设计模式是现实世界事物的必然模型，而是说在契合的对象之间相互影响的关系可以在“现实世界”和软件对象中同样地观察到。为了验证这个假设，我们将为每一种设计模式找出一个现实世界的例子来。这些例子在下面的第二节至第四节列出。

创建型模式

作者（指《设计模式》的作者—译注，下同）总结了五种创建型模式。创建型模式的例子可以在制造业，快餐，生物和行政机构中找到。

抽象工厂（Abstract Factory）举例

抽象工厂的目的是要提供一个创建一系列相关或相互依赖对象的接口，而不需要指定它们具体的类。这种模式可以在日本汽车制造厂所使用的金属冲压设备中找到。这种冲压设备可以制造汽车车身部件。同样的机械用于冲压不同的车型的右边车门、左边车门、右前挡泥板、左前挡泥板和引擎罩等等。通过使用转轮来改变冲压盘，这个机械产生的具体类可以在三分钟内改变[16]。

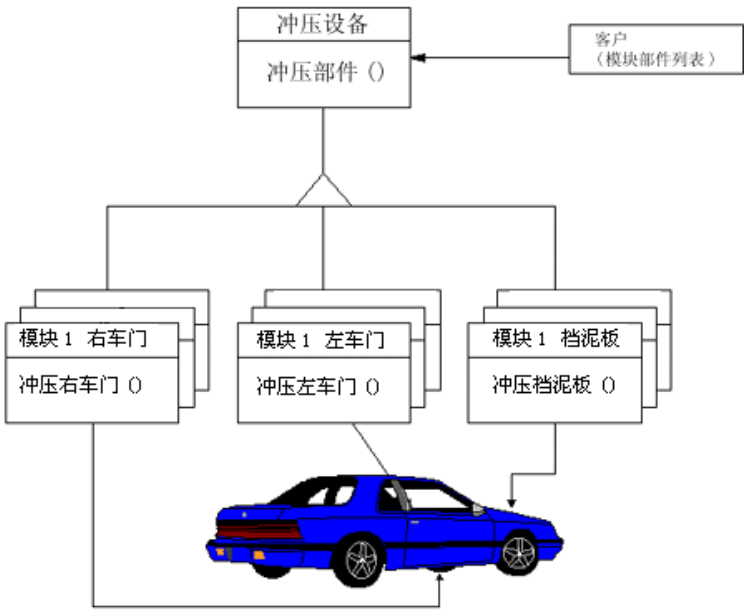


图 1：抽象工厂的冲压例子

生成器（Builder）举例

生成器模式将复杂对象的构建与对象的表现分离开来，这样使得同样的构建过程可以创建出不同的表现。这种模式用于快餐店制作儿童餐。典型的儿童餐包括一个主食，一个辅食，一杯饮料和一个玩具（例如汉堡、炸鸡、可乐和玩具车）。这些在不同的儿童餐中可以是不同的，但是组合成儿童餐的过程是相同的。无论顾客点的是汉堡，三名治还是鸡肉，过程都是一样的。柜台的员工直接把主食，辅食和玩具放在一起。这些是放在一个袋子中的。饮料被倒入杯中，放在袋子外边。这些过程在相互竞争的餐馆中是同样的。

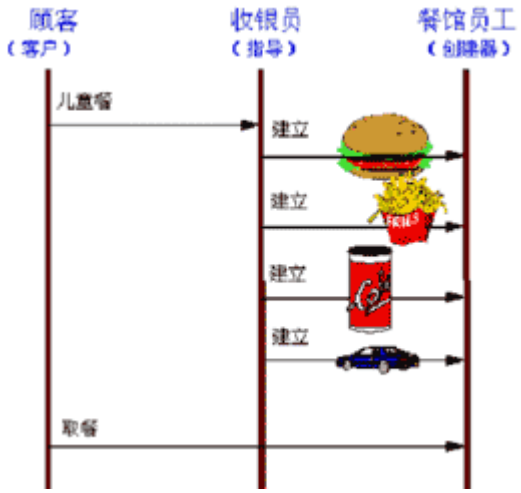


图 2：使用儿童餐作为例子的生成器模式的对象作用表

工厂方法（Factory Method）

工厂方法定义一个用于创建对象的接口，但是让子类决定实例化哪个类。压注成型演示了这种模式。塑料玩具制造商加工塑料粉，将塑料注入到希望形状的模具中[15]。玩具的类别（车，人物等等）是由模具决定的。

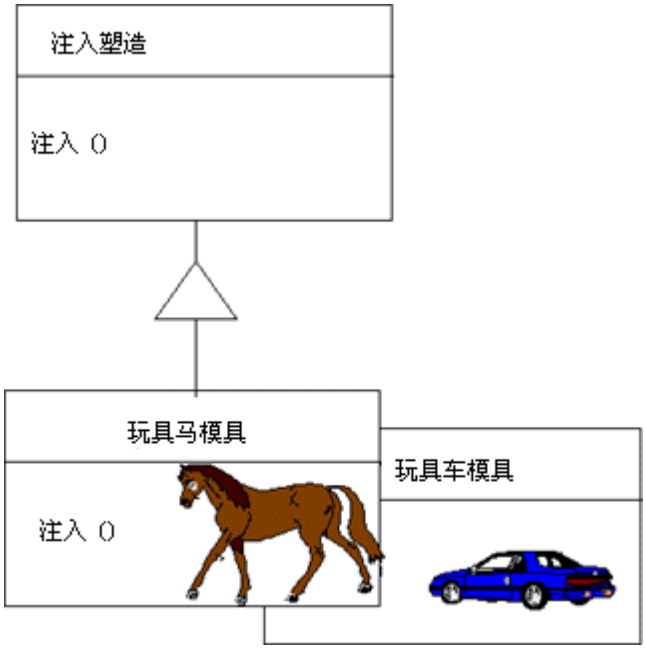


图 3：使用注入成型为例子的工厂方法的对象图

原型（Prototype）举例

原型模式使用原型实例指定创建对象的种类。新产品的原型通常是先于全部产品建立的，这样的原型是被动的，并不参与复制它自己。一个细胞的有丝分裂，产生两个同样的细胞，是一个扮演主动角色复制自己原型的例子，这演示了原型模式。一个细胞分裂，产生两个同样基因型的细胞。换句话说，细胞克隆了自己。

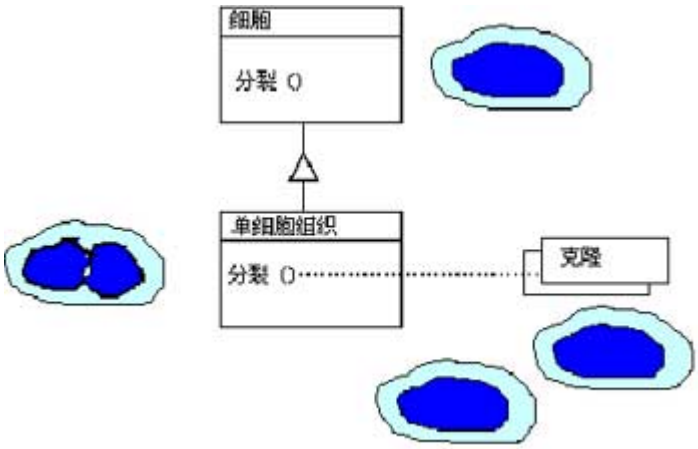


图 4：使用细胞分裂例子的原型模式对象图

单件（Singleton）举例

单件模式确保一个类仅有一个实例，并提供一个访问它的全局访问点。单件模式是模仿单集命名的，单集的定义是每个集合仅含有一个元素。美国总统的职位是单件，美国宪法规定了总统的选举，任期以及继任的顺序。这样，在任何时刻只能由一个现任的总统。无论现任总统的身份为何，其头衔“美利坚美利坚合众国总统”是访问这个职位的人的一个全局的访问点。

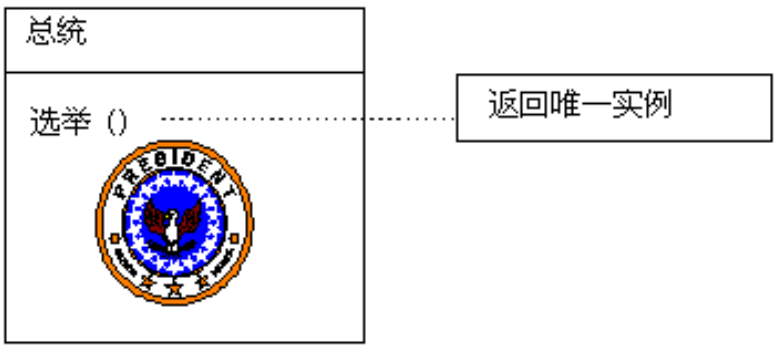


图 5：使用总统例子的单件模式对象图

结构性模式

作者总结了七个结构型模式，这些模式的例子可以在工具、住宅配线、数学、节日传统、零售目录和银行业中找到。

适配器（Adapter）举例

适配器模式允许将一个类的接口转换成客户期望的另一个接口，使得原本由于接口不兼容而不能一起工作的类可以一起工作。扳手提供了一个适配器的例子。一个孔套在棘齿上，棘齿的每个边的尺寸是相同的。在美国典型的边长为 1/2”和 1/4”。显然，如果不使用一个适配器的话，1/2”的棘齿不能适合 1/4”的孔。一个 1/2”至 1/4”的适配器具有一个 1/2”的阴槽来套上一个 1/2”的齿，同时有一个 1/4 的阳槽来卡入 1/4”的扳手。

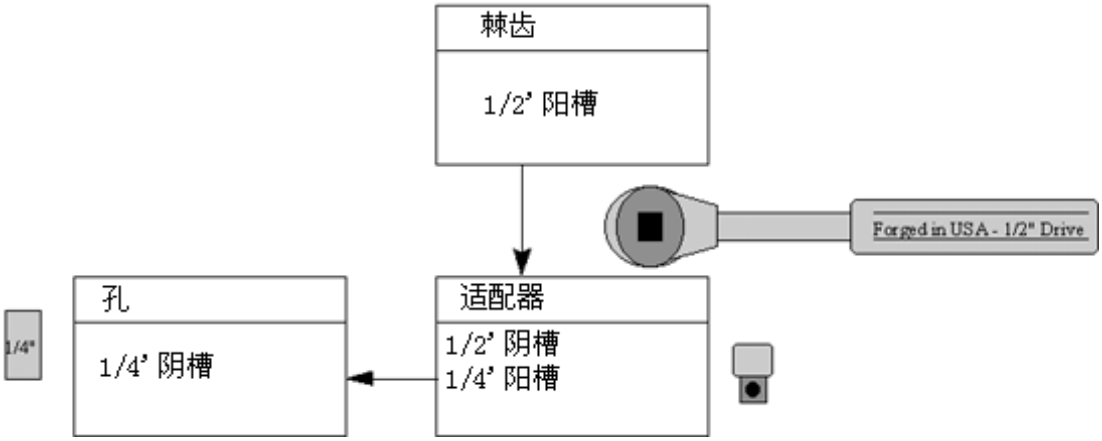


图 6：使用扳手适配器例子的适配器对象图

桥接（Bridge）举例

桥接模式将抽象部分与它的实现分离，使它们能够独立地变化。一个普通的开关控制的电灯、电风扇等等，都是桥接的例子。开关的目的是将设备打开或关闭。实际的开关可以是简单的双刀拉链开关，也可以是调光开关。

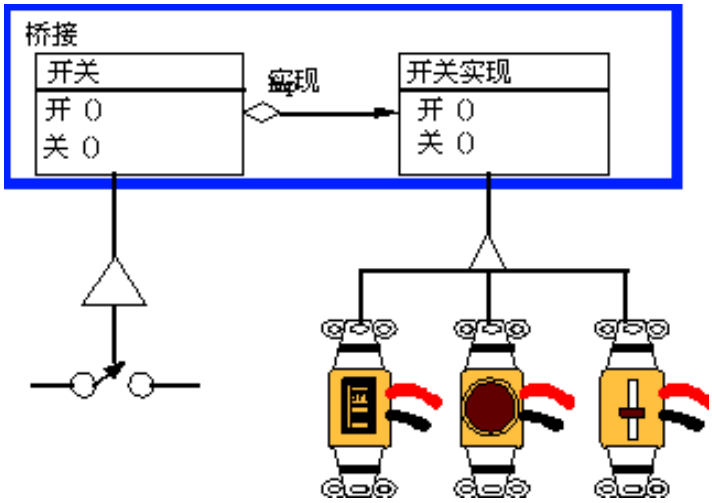


图 7：使用电子开关例子的桥接对象图

组合（Composite）例子

组合模式将对象组合成树形结构以表示“部分-整体”的层次结构。让用户一致地使用单个对象和组合对象。虽然例子抽象一些，但是算术表达式确实是组合的例子。算术表达式包括操作数、操作符和另一个操作数。操作数可以是数字，也可以是另一个表达式。这样，2+3 和（2+3）+（4*6）都是合法的表达式。

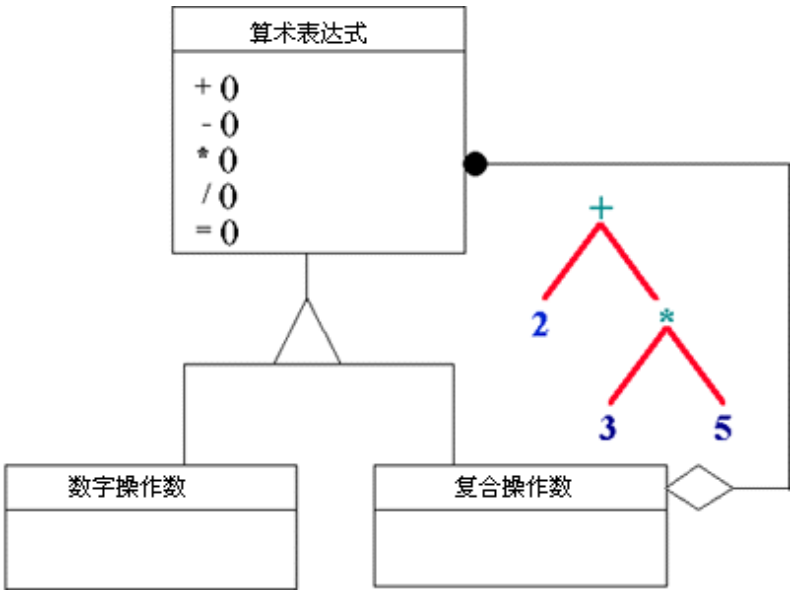


图 8：使用算术表达式例子的组合模式对象图

装饰（Decorator）举例

装饰模式动态地给一个对象添加额外的职责。不论一幅画有没有画框都可以挂在墙上，但是通常都是有画框的，并且实际上是画框被挂在墙上。在挂在墙上之前，画可以被蒙上玻璃，装到框子里；这时画、玻璃和画框形成了一个物体。

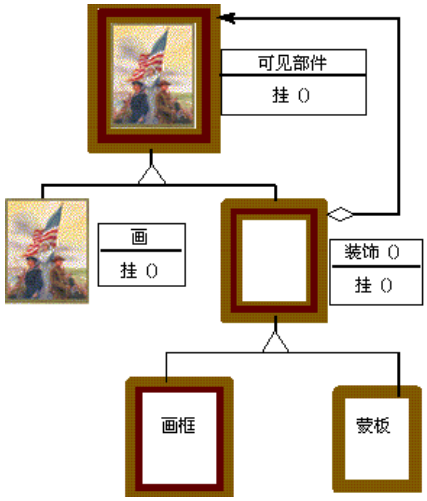


图 9：使用有画框的画作为例子的装饰模式对象图

外观（Facade）举例

外观模式为子系统中的接口定义了一个统一的更高层次的界面，以便于使用。当消费者按照目录采购时，则体现了一个外观模式。消费者拨打一个号码与客服代表联系，客服代表则扮演了这个“外观”，他包含了与订货部、收银部和送货部的接口。

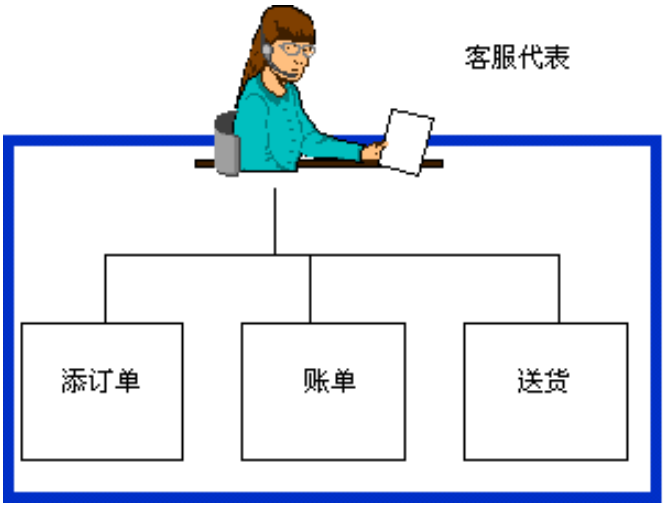


图 10：使用电话订货例子的外观模式对象图

享元（Flyweight）举例

享元模式使用共享技术有效地支持大量细粒度的对象。公共交换电话网（PSTN）是享元的一个例子。有一些资源例如拨号音发生器、振铃发生器和拨号接收器是必须由所有用户共享的。当一个用户拿起听筒打电话时，他不需要知道使用了多少资源。对于用户而言所有的事情就是有拨号音，拨打号码，拨通电话。

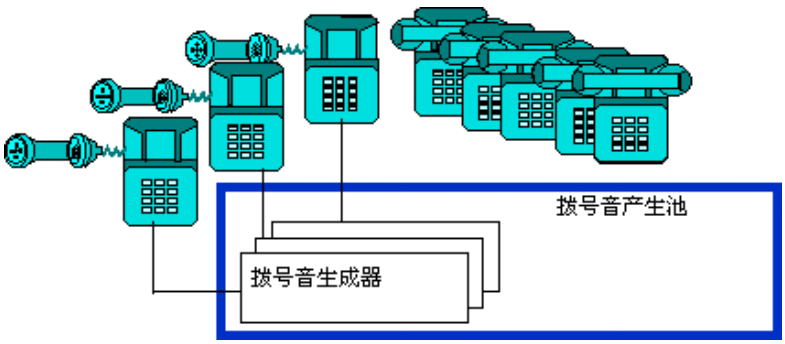


图 11：使用拨号音发生器例子的享元模式对象图

代理（Proxy）模式

代理模式提供一个中介以控制对这个对象的访问。一张支票或银行存单是账户中资金的代理。支票在市场交易中用来代替现金，并提供对签发人账号上资金的控制。

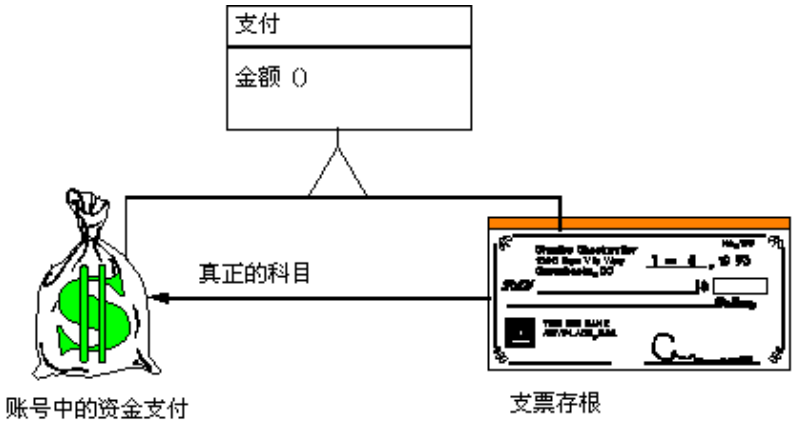


图 12：使用银行存单例子的代理模式对象图

行为模式

作者总结了十一种行为模式。这些模式可以在硬币分类银行、餐馆订餐、音乐、运输、汽车修理、自动售货机和家庭建筑中找到例子。

职责链（Chain of Responsibility）举例

职责链模式使得多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。机械硬币分拣银行使用职责链。这里并不是为每一种硬币分配一个滑槽，而是仅使用一个滑槽。当硬币落下时，硬币被银行内部的机械导向至适当的接收盒。

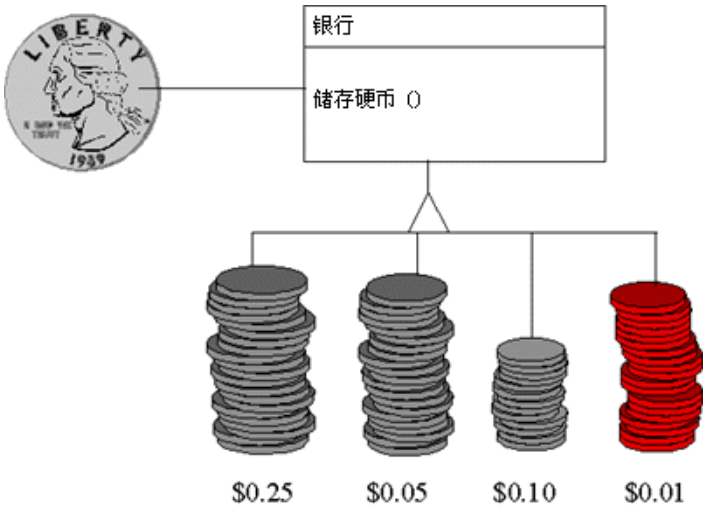


图 13：使用硬币分拣例子的职责链模式对象图

命令（Command）模式

命令模式将一个请求封装为一个对象，从而使你可以使用不同的请求对客户进行参数化。用餐时的账单是命令模式的一个例子。服务员接受顾客的点单，把它记在账单上封装。这个点单被排队等待烹饪。注意这里的“账单”是不依赖于菜单的，它可以被不同的顾客使用，因此它可以添入不同的点单项目。

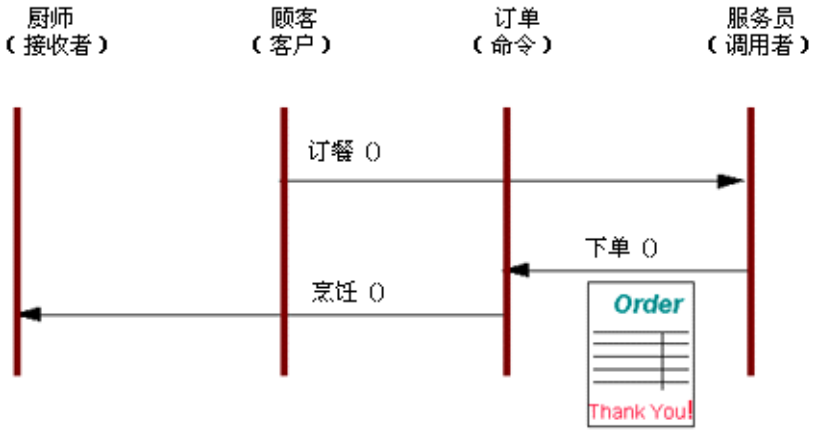


图 14：使用用餐例子的命令模式对象图

解释器（Interpreter）举例

解释器模式定义了特定语言的文法表示和解释该文法的解释器。音乐家是解释器的例子。音阶和它的持续时间可以用五线谱上的符号表示。这些符号就是音乐语言[14]。音乐家按照乐谱演奏，就可以反复重现同样的音乐。

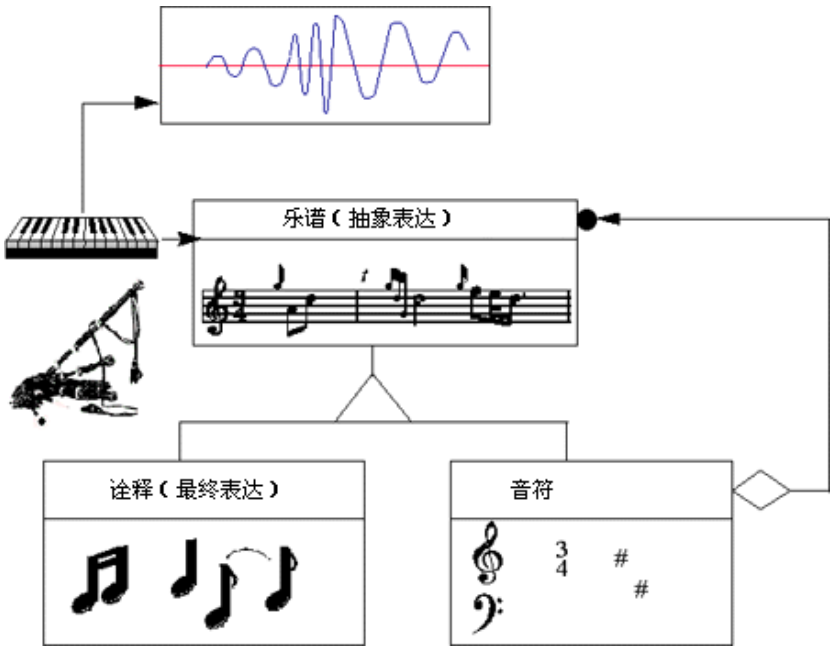


图 15：使用音乐例子的解释器模式对象图

迭代器（Iterator）举例

迭代器提供一种方法顺序访问一个集合对象中各个元素，而又不需要暴露该对象的内部表示。在早期的电视机中，一个拨盘用来改变频道。当改变频道时，需要手工转动拨盘移过每一个频道，而不论这个频道是否有信号。现在的电视机，使用[后一个]和[前一个]按钮。当按下[后一个]按钮时，将切换到下一个预置的频道。想象一下在陌生的城市中的旅店中看电视。当改变频道时，重要的不是几频道，而是节目内容。如果对一个频道的节目不感兴趣，那么可以换下一个频道，而不需要知道它是几频道。

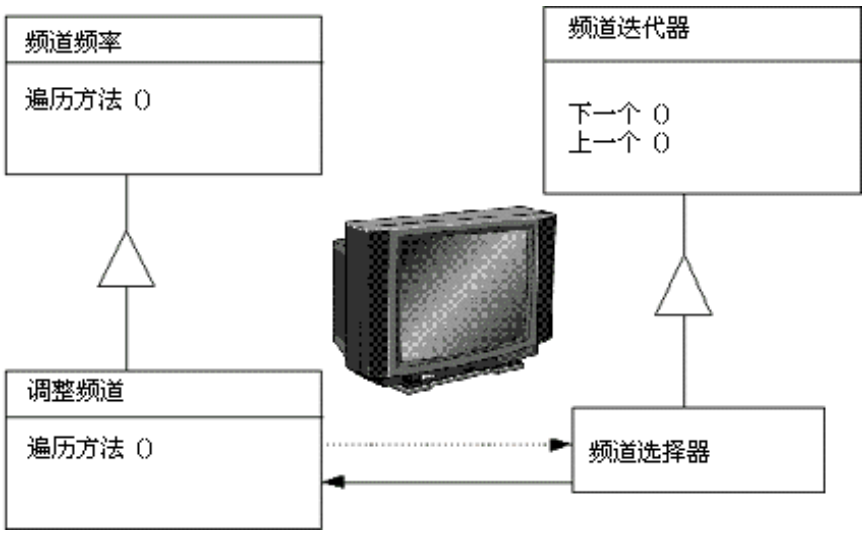


图 16：使用选频器作例子的迭代模式对象图

中介者（Mediator）举例

中介者模式用一个中介对象来控制一系列的对象交互。通过中介者实现各个对象之间的松散耦合，而不是彼此直接作用。机场的控制塔很好地演示了这种模式。降落或者起飞的飞机直接与塔台通讯，而不是彼此间直接通讯。谁可以起飞或降落是由塔台决定的。这里需要注意的是塔台并不控制整个飞行过程。它只负责飞机在机场附近的区域。

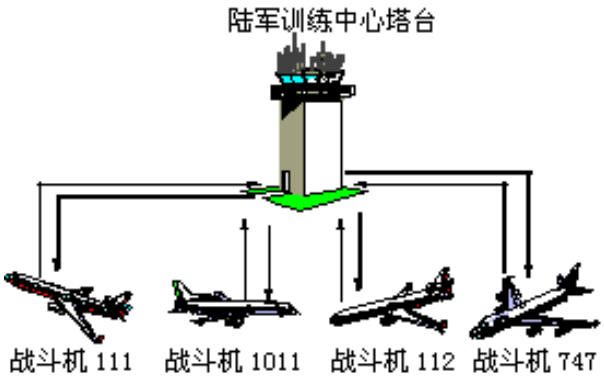


图 17：使用训练中心为例子的中介者模式

备忘录（Memento）举例

备忘录模式捕获并且在外部保存一个对象的内部状态，使得以后可以将对象恢复到该状态。这种模式通常体现在你自己修理汽车的刹车时。首先移开两边的挡板，露出左右刹车片。只能卸下一片，这时另一片作为一个备忘录来表明刹车是怎样安装的。在这片修理完成后，才可以卸下另一片。当第二片卸下时，第一片就成了备忘录。

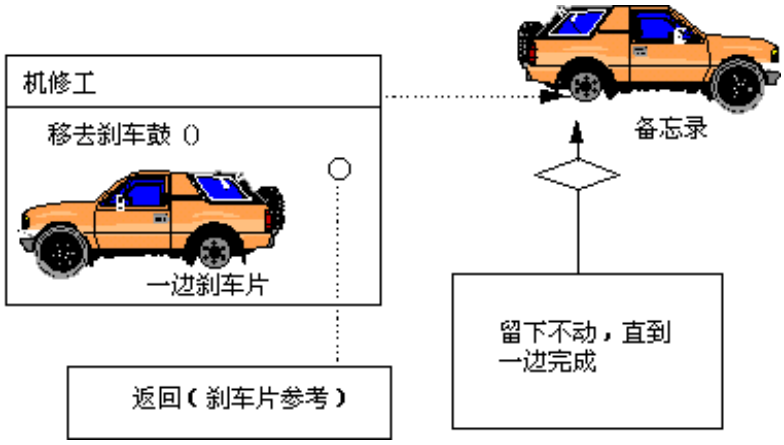


图 18：使用刹车片例子的备忘录模式对象图

观察者（Observer）模式

观察者定义了对象间一对多的关系，当一个对象的状态变化时，所有依赖它的对象都得到通知并且自动地更新。拍卖演示了这种模式。每个投标人都有一个标有数字的牌子用于出价。拍卖师开始拍卖时，他观察是否有牌子举起出价。每次接受一个新的出价都改变了拍卖的当前价格，并且广播给所有的投标人进行新的出价。

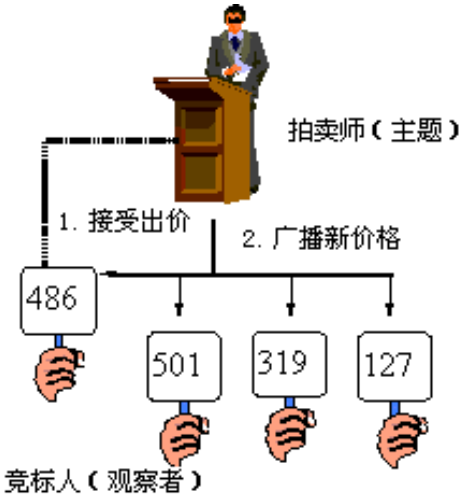


图 19：使用拍卖例子的观察者模式

状态（State）模式

状态模式允许一个对象在其内部状态改变时改变它的行为。这种模式可以在自动售货机上观察到。自动售货机的状态包括列商品清单，收款，找钱和选择商品等几种状态。当投入硬币并选择了一个商品时，自动售货机可以完成以下几种操作，包括：送出商品不找钱、送出商品并找钱、由于投币不足不送出商品、由于商品售完不送出商品。

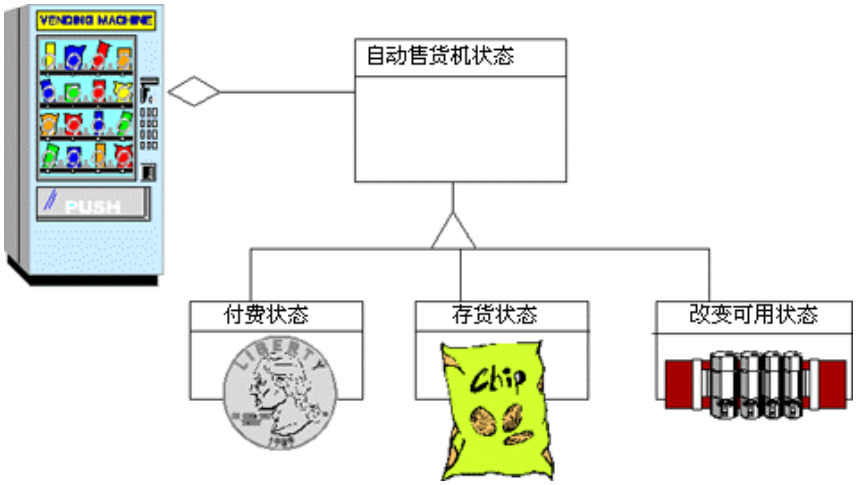


图 20：使用自动售货机例子的状态模式对象图

策略（Strategy）模式

策略模式定义了一系列可以相互替换的算法。不同的到飞机场去的方式就是一个策略模式的例子。有几种选择：自己开车、坐出租车、乘机场班车、乘公共汽车或使用专车服务等等。对于某些机场，地铁和直升机也是可能的选择。任何一种方式都可以把你送到机场，它们可以相互代替。你必须根据价格、便利性和时间做出选择。

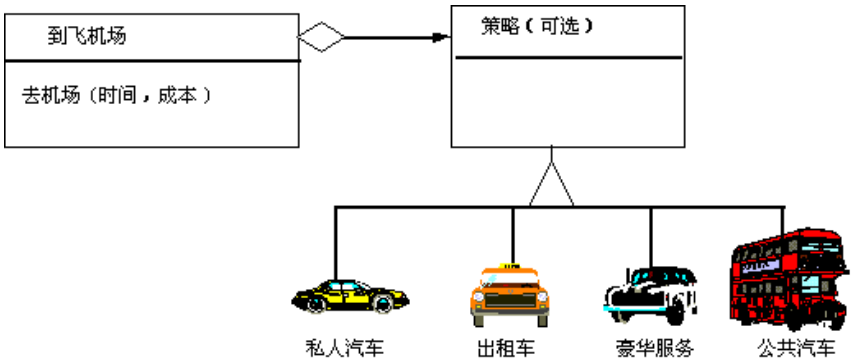


图 21：使用去机场作为例子的策略模式对象图

模板方法（Template Method）举例

模板方法定义了一个操作中算法的骨架，而将一些步骤延迟到子类中。房屋建筑师在开发新项目时会使用模板方法。一个典型的规划包括一些建筑平面图，每个平面图体现了不同部分。在一个平面图中，地基、结构、上下水和走线对于每个房间都是一样的。只有在建筑的后期才开始有差别而产生了不同的房屋样式。

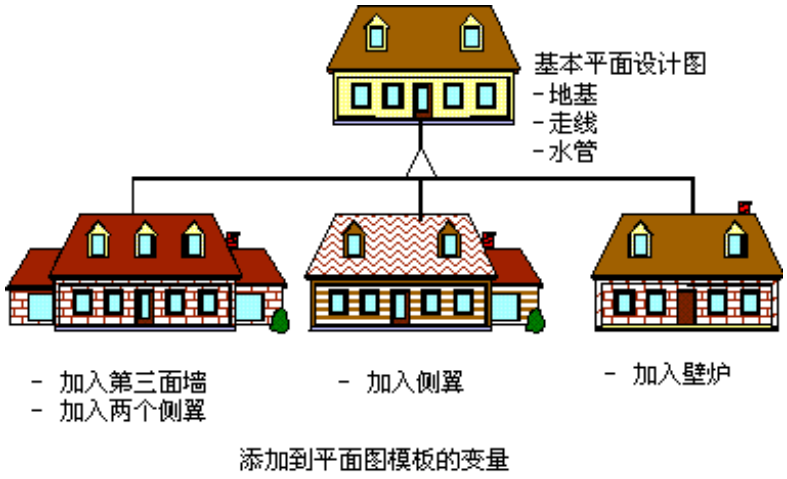


图 22：使用建筑平面图为例子的模板方法模式

访问者（Visitor）举例

访问者模式表示一个作用于对象结构中各元素的操作，定义这个操作并不会改变元素的类。这种模式可以在出租车公司的运转中观察到。当一个人给出租车公司打电话时，他（她）就成了公司所有顾客的一员。然后公司指定一辆车去服务（接受访问者）。在进入出租车之后，这个人（访问者）就不再控制他（她）的旅程了，而是由出租车（驾驶员）负责。

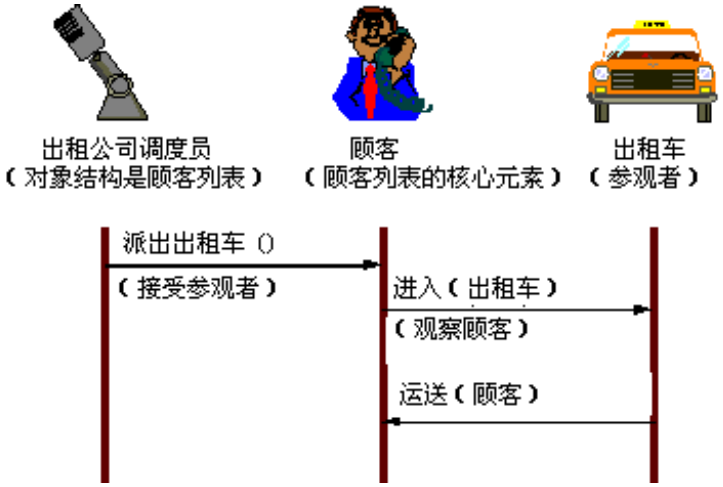


图 23：使用出租车例子的观察者模式对象图

意义

作者已经表明了软件设计模式的非软件例子的存在。也许有人想知道这些例子的实际意义。非软件例子有助于增进模式语言的表达力和辅助模式的学习。

增加模式语言的表达能力

Alexander 觉得真正的模式要融入一种通用的语言以便所有人都能够分享[2]。在软件设计的人群中，模式被认为是在同事之间一种约定俗成的开发方式[4，17]。模式提供了一种比模块、过程和对象更高层次的概念[10]。

一种语言中至关重要的因素是同语言形象所对应的心灵影像。在一种语言中，仅当一个人能够领会一个符号的含义，能够在心里描绘出这种含义时，这个符号的外形才是有意义的[7]。Alexander 没有忽视模式语言的这种重要特征，他规定：一种语言只有在它所产生的建筑类型能够被具体地看到之后，这种语言才是完全形态化的[1]。在软件设计中，Richle 和 Züllighoven 认识到具体的例子在指导我们对应用领域的理解的重要性[12]。

如果软件设计模式成为程序员中通用的语言，其基础则是统一的含义。如果设计决定下达了，但是没有被理解，则设计师被迫通过假设来完成工作[19]。平凡的例子更便于理解，其原因在于人们必须在记忆中找到相关联的内容才能够理解[20]。在广泛使用模式的 AG Comm□nication Systems 公司的项目中，常常使用非软件例子来解释模式之间的关系。这个例子有助于在设计师间提供统一的理解。通过在设计过程的先期建立统一的理解，使得在整个项目生命周期中，设计师间的沟通更加容易。

非软件例子作为学习模式的助手

在新概念出现时，学生需要例子。这可以在 AG Comm□nication Systems 公司的模式推行过程的总结中见到，同时这也是被其他人总结出来的[12]。在学习新东西时，学生很自然地会使用学过的知识来帮助新概念的理解[6]。由于这个原因，许多例子应当在学生第一次接触软件设计模式时提出[12]。某些例子应该在学生学习了一段时间以后提出，而不是等他们成为专家以后。提供相似的例子并不增加需要学习的新内容。这时，选取学生课外的例子可以避免学习素材短缺的情况。既然模式最终要停留在人的头脑中[11]，因此可以使用大多数人熟悉的例子和来自于记忆中的训练材料。

结论

在非软件例子中软件设计模式的体现表明了模式不是局限于特定领域的。软件设计师可以从这些日常事物的模式举例中受益，哪怕这些例子并不是以程序设计语言表达的。这篇文章尽可能举一些大部分人所熟悉的例子（尽管某些倾向于北美文化）。通过对共同的经历的描述，这些例子有助于对特定的设计模式的理解，并且能够帮助对设计模式的学习。

※ 感谢

本文作者非常感谢 Dalmatian Gro□p 的 Brandon Goldfedder，AG Comm□nication Systems 的 Linda Rising，H□mans and Technology 的 Alistair Cockb□rn 和伊利诺思大学的 Ralph Johnson 对本文提出的有益建议。

※ 参考书目

1. Alexander, C. *The Timeless Way of Building.* Oxford University Press, 1979.

2. Alexander, C., et al. *A Pattern Language.* Oxford University Press, 1977.

3. Anthony, D. "Patterns for Classroom Education", in *Pattern Languages of Program Design II*, Addison-Wesley, 1996.

4. Berczuk, S. "Finding solutions through pattern languages", *Computer,* Vol. 27, No. 12. December, 1994.

5. Booch, G. "Object Oriented Design" in *Tutorial on Software Design Techniques,* pp. 420-437, IEEE Computer Society, 1984.

6. Carroll, J. *"The Nurnberg funnel: Designing minimalist instruction for practical computer skill",* MIT Press, 1990.

7. Chierchia, G. and McConnel-Ginnet, S. *Meaning and Grammar: An Introduction to Semantics.* MIT Press, 1990.

8. Chilton, *Chilton's Auto Repair Manual,* Chilton Book Company, 1985.

9. Coplien, J. "Broadening beyond objects to patterns and to other paradigms", *Position statement for the ACM Workshop on Strategic Directions in Computing Research*, MIT, June 14-15, 1996.

10. Coplien, J. "Idioms, Patterns, and Other Architectural Literature", IEEE Software, November, 1996.

11. Cunningham W., Johnson, R., Introduction to Pattern Languages of Program Design, Addison-Wesley, 1995.

12. DeBruler, D. "A Generative Pattern Language for Distributed Processing", in Pattern Languages of Program Design, Addison-Wesley, 1995.

13. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

14. Leonhardt, C. Discovering Music Together 7, California State Department of Education, 1967.

15. President and Fellows of Harvard College, Good Time Toy Company, Publishing Division, Harvard Business School, 1986.

16. Hill, C.W.L. "Toyota: The Evolution of Toyota's Production System" in Cases in Strategic Management, Houghton Mifflin, 1993.

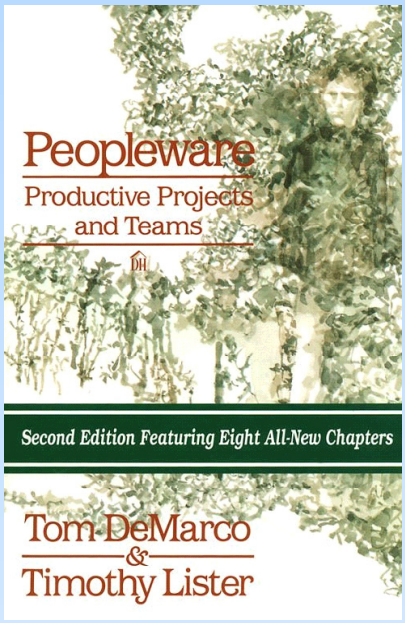
17. OOPSLA '95 "Patterns: Cult to Culture?", Panel Discussion in the Addendum to the Proceedings. ACM Press, 1996.

18. Richle, D, Züllighoven, H. "A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor", in Pattern Languages of Program Design, Addison-Wesley, 1995.

19. Ross, D., and Schoman Jr., K. "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, Vol. SE3, No 1., January, 1977.

20. Schank, R. "Tell Me a Story: A New Look at Real and Artificial Memory", Charles Scribner's Sons, 1990.

《人件》



《人件》第 2 版

Tom Demarco 和 Tim Lister

翻译：UMLChina 翻译组方春旭、叶向群

微软的经理们很可能都读过—amazon.com

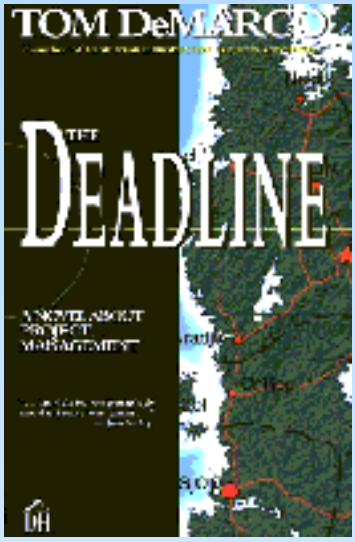
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

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

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

中文译本即将发行！

《最后期限》



《最后期限》

Tom Demarco

翻译：UMLChina 翻译组 透明

这是一本软件开发小说

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

....

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

莱克莎点点头："这是最近的数字。他们都会在你的手下工作。"

"而且据你所说，他们都很优秀。"

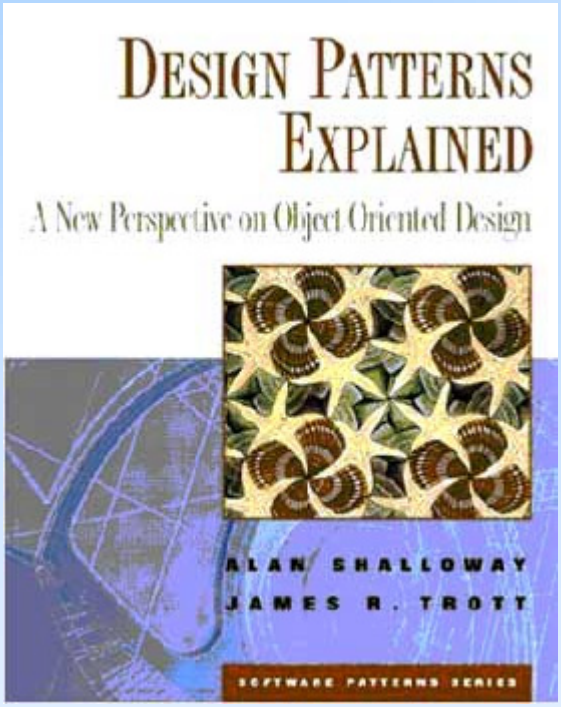
"他们都通过了摩罗维亚软件工程学院的 CMM 2 级以上的认证。"

中文译本即将发行！

看不懂《设计模式》？

它的作者推荐《设计模式精解》

透明 译



直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 **Alan Shalloway** 的《设计模式精解》。

--John Vlissides 写于 UMLChina 答疑板

中译本即将上市！

GOF 模式用于 GUI 设计

James Noble 著，cntang 译

摘要

《设计模式》一书中为面向对象的软件设计引入了 23 个设计模式。这些模式被广泛应用，但只限于软件设计的目标领域。我们描述了如何在图形用户界面的概念性设计中使用 6 个设计模式。通过使用这些模式，设计者产生的界面能够更一致，更好地利用屏幕空间，更容易使用。

简介

本文展示了用于图形用户界面设计的 6 个模式（Prototype，Singleton，Adaptor，Composite，Proxy，Strategy），每个模式都起源于 GOF 中的同名模式。这些模式从 Star 和 Smalltalk 的早期工作开始就被广泛使用。我们用最近的 GUI 的一些例子来演示这些模式。为了把这些模式应用到 GUI 设计，我们已经把 GOF 模式从面向对象的软件设计领域转换到 GUI 领域。虽然 GOF 模式和我们的 GUI 模式都用了对象，接口，消息等术语，但我们已经把这些术语翻译到 GUI 领域。我们的对象指单个可控的 GUI 组件，比如一个图标，一个对话框，或一个窗口。单个对象可以不止一种表现形式。例如一个文档在关闭时可以表现为一个图标，而打开时可以是一个窗口。对象拥有一些属性，比如可编辑文本框，单选按钮，多选按钮;对象的行为可以由按钮，菜单，击键引发或直接控制这个对象而引发。一组具有相似属性和行为的对象形成一类，比如，所有的窗口对象可以归为窗口类。通过继承关系来捕捉对象之间的细微差别。类可以分组，例如，所有的目录窗口可以归为一类，所有的应用窗口可以归为另一类，而他们都是从窗口类继承而来。

这种转变表明了 GOF 领域和 GUI 领域的重要区别。GOF 模式可以指对象的实现，而 GUI 模式只是必要地涉及 GUI 对象的接口。对那些主要和接口有关的的模式来说(比如代理或原型)，区别很小。对那些主要和封装有关的模式来说(比如策略)，相对应的 GUI 模式引出了模式的第二层面，比如参数化和定位对象。

这些模式并不涉及可用性，虽然我们在为可用性而扩展设计的界面中可以找到这些模式。我们相信这些模式和用户界面设计的其他技术(比如走查法(walkthrough)，原型法，metrics，可视化方法(visualisation))是不相关的，就像软件开发中的模式是和其他软件开发技术是不相关的。相应的，GOF 模式处于编程语言和和方法学之间，而这些模式也处在用户界面指南(用户界面语言规格)和用户界面方法学之间。

形式和内容

本文的大部分内容展示了我们识别出的 6 个 GUI 模式。这些模式以和 GOF 书中相对应的秩序和方式展示和组织。我们先从创建型模式--原型和单体模式开始，然后展示结构型模式--适配器模式，组合模式，代理模式，然后以行为型模式--策略模式结尾。本文包含了一个模式列表，而不是一个模式系统或一种模式语言。

因为文章长度上的原因，我们用一种简化的方式来展示这些模式。每一种模式都有一个名字，从 GOF 模式相应的段落中导出的目的声明。一个简短的问题说明描述了模式的动机和应用，然后模式所要解决的约束显式地列出条目。解决方案的结构、参与者、合作者和实现则组合在一个段落中。一幅图片演示了应用中的模式的一个例子，取代了 GOF 模式中代码例子。最后，列出了模式的积极和消极后果以及对已知应用例子的评论。这种形式把 GOF 形式中的主要叙述章节压缩成两个短小的段落（问题和解决方案），每个都带有相关列表（约束、后果等等）。

大多数的约束都来自于模式的内容，和 GOF 模式中的约束是同构的。这种同构是重要的，因为它确保跨领域的解决方案在形式上相似。这些模式同时也参考了 GUI 设计领域特有的约束。

- 界面应当保持一致是 GUI 设计领域一个明显的约束。在 OO 设计中，一致性用于解决其他问题，比如可重用性。
- 屏幕资源是重要的，通常是稀缺的资源
- GUI 中的对象标志意味着一个对象通常只能在屏幕的某个地方出现一次。

相关模式

假如第一个软件模式语言确实是关于界面设计的，那么这个领域只有如此之少的模式和模式语言是令人吃惊的。工具和语言隐喻已在模式语言中有所描述，虽然语言本身涉及的实现和设计一样多。一些模式语言已经成文，并用于设计基于文字的 WEB 站点和表单式窗口。

ENVOI

我们相信一些 GOF 模式抓住了面向对象软件设计深层次的共同特性。本文是一个实验，想确定这些模式在相关但又显著不同的 GUI 设计领域是否可行（和有多好）。

GUI 原型（Prototype） 创建型对象

目标 通过原型的实体来创建指定种类的对象，通过拷贝原型来创建新对象。

问题 GUI 支持许多种不同的对象，这些对象的功能不同。用户需要创建能够存放他们的数据的合适对象。例如，桌面支持几种不同的文档类型，比如字处理文档，电子表格，数据库等。用户如何创建这些新对象?

约束 GUI 原型模式解决了下列外部约束:

- 用户需要创建不同种类的文档
- 你不可能预见到系统中的所有类别
- 对象应当用一致的方法产生

解决方案 为每一个类创建一个原型对象。让用户通过拷贝恰当原型的方法来创建新对象。每一个类都应当理解"拷贝"消息，而且都是从所有用户对象的(父)类继承下来的。

案例 在 SELF 系统用户界面中，所有的对象都是通过复制现有的原型来创建的，并没有"CREATE"这样的操作。MACINTOSH 系统 7 允许文档被标志为"信笺板"。当用户打开信笺板时，信笺板被拷贝，然后拷贝被打开。

结果 GUI 原型模式有下列优点和缺点

+ 对所有的可拷贝类来说，用户可以用同一种方法来创建对象
+ 可以通过增加新原型来增加新类
+ 通过拷贝已经初始化过的对象，用户可以创建他们自己的类

- 用户也许需要清空新拷贝来的对象中的数据

- 用户也许忘记拷贝对象而偶尔直接编辑原型

已知应用 原型首先在 Sketchpad 中使用。模式被显著地应用在 Self UI 系统，MoDE Composer，Macintosh 等系统中。我们曾经给许多用户建议象本能一样地使用原型。

GUI 单体（Singleton） 创建型模式

目标 确保一个类只有一个实体，同时提供能够作为全局变量使用

问题 一些类应该恰好只有一个实体。例如，代表真实硬件资源的对象，如硬盘，打印机，网络，特殊的系统对象如垃圾桶，在任何一个 GUI 中都应该只有一个。你该如何管理这些独一无二的对象呢？

约束 GUI Singleton 模式解决了以下约束：

- 这些对象应该只有一个实体。
- 单个实体应该能够容易找得到。
- 用户界面应当一致。
- 屏幕资源是有限的。

解决方案 创建一个 Singleton 对象，设计类使得用户不能拷贝或删除对象。GUI 启动时就创建它，把它放到屏幕上（通常是桌面），这样它总是容易找到。

案例 WINDOWS95 里面的垃圾桶是一个 Singleton。它没有大多数对象提供的删除和重命名命令。

结果 GUI Singleton 模式有如下优缺点：

+只有一个单体实体能被创建

+单体很容易在桌面上找到

+单体具有象其他对象一样的行为

GUI 适配器（Adaptor） 结构型对象

目标 把一个类的接口转成使用者期望的接口。，适配器模式让接口不兼容的类可以协同工作。

问题 每个接口有他们自己的世界，都有自己的文化，语言，样式。 使用者经常需要保有好几个世界。例如，当使用从 GUI 桌面继承下来的框架应用，或从另外一个桌面使用这个桌面。用户如何使用不兼容接口呢?

约束 GUI Adaptor 模式解决了以下约束：

- 用户需要使用有不兼容接口的对象
- 你不想改变这个接口
- 用户界面应当保持一致

解决方案 创建一个适配器对象，这个对象包容了另一个对象的接口。适配器是有主接口的正常对象，包容了被包含的接口。用户可以通过适配器与被包容的接口交互。适配器通常还提供了带外（out-of-band）操作来控制两个接口之间的连接。

案例 Window NT 接口可以适配以便在 X Windows 系统下运行(Tektronix 菜单条控制适配器)。Windows NT 可以 通过终端模拟器来容纳文本式应用程序。

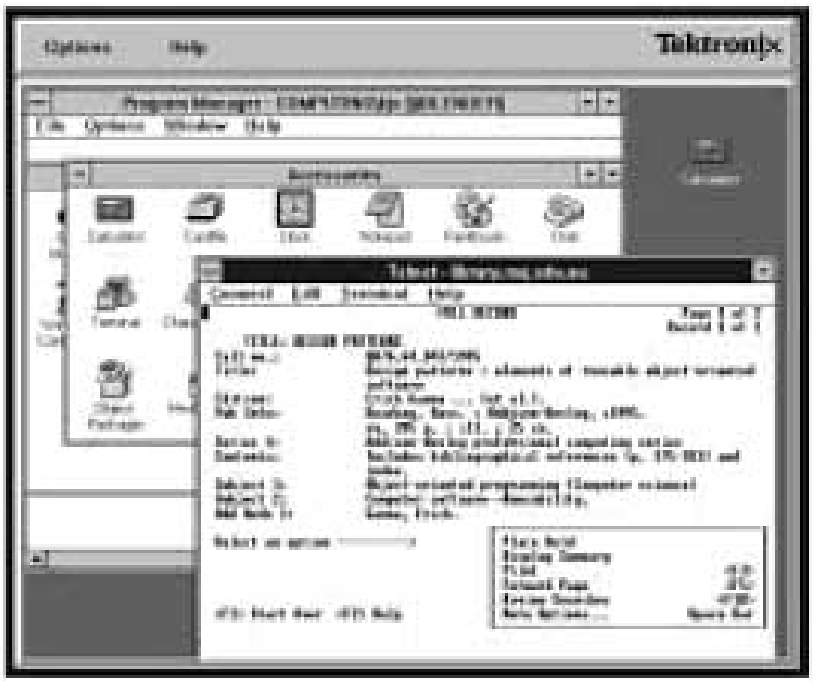
结果 GUI 适配器模式有如下优缺点：

+可以通过主接口来使用被包容的接口

+两个接口都无需改变

+通用适配器可以适配一系列接口，比如所有的文本应用程序。

-如果适配器中未遵循主接口的习惯用法，会导致用户接口不一致



已知应用 大多数 GUI 通过终端模拟器来支持文本式应用程序。Windows 和 Macintosh 界面可以适配运行在 X 系统中，或者相反。网页浏览器把 www 用户界面适配到范围很广的主界面。

参看 一个老式的包装器（Wrapper）是这种模式的更通用的形式。

GUI 组合（Composite） 结构型对象

目的 把对象组合成树状机构来代表部分-整体层级。组合模式可以让用户统一对待单个对象和对象的组合。

问题 许多 GUI 对象是由其他 GUI 对象递归组成的。比如，磁盘包含了目录，目录又包含其他目录和文件。用户应该如何操纵这些组合对象？

约束 GUI 组合模式解决了如下约束：

- 用户需要控制整个组合对象
- 用户需要控制组合对象中的单个部分。
- 用户界面应当一致

解决方案 创建能够递归包含其他对象的组合式对象，包括组合对象和原始对象。为所有由原始对象和组合对象共享的操作建立一个共同类。具体对象应继承于这个类，并扩展以提供它们自己的操作。

案例 在 Self UI 系统中，任何一个图形对象都可被组合（嵌入）来创建组合式结构。Window NT 把目录结构表示成组合对象。

结果 GUI 组合模式有如下优缺点：

+用户既可以操纵整个对象，也可以操纵其中的一部分。

+通用操作的接口是一致的。

-很难把特殊的一部分从整体中独立出来。

已知应用 Macintosh 和 Windows95 界面使用组合对象来代表目录结构。MacDraw 和许多其他图像编辑器通过显式分组和非分组图形对象来创建组合对象。

GUI 代理（Proxy）　　结构式对象

目的 提供一个代理或放置地以方便另一个对象来存取。

问题 某些对象从来不在用户想要的地方。比如，用户想要存储目录结构下很深的一个文件，想让它很容易存取。或者用户希望下载一个网页，放弃任何包含的图像而保持文档结构不变。一个对象如何才能同时出现在两个地方呢？

约束 GUI 代理模式解决了如下约束：

- 用户想要一个对象同时出现在两个地方。
- GUI 对象识别要求一个对象只能出现在一个地方
- 你不想改变或移动原始对象。
- 原始对象的检索很困难或很昂贵。
- 用户界面应当一致。

解决方案 创建一个代理对象来代表远程或昂贵的对象。把代理对象放在你想要放原始对象的地方，但不能让代理象原始对象一样操作，而是可视地把代理同原始对象区别开来。

案例 Wwidows95 快捷方式行为方式象代理一样，所以一个对象可以出现在桌面的许多个地方。Netscape 用图标作为还未下载图片的代理。

结果 GUI 代理模式有如下的优缺点：

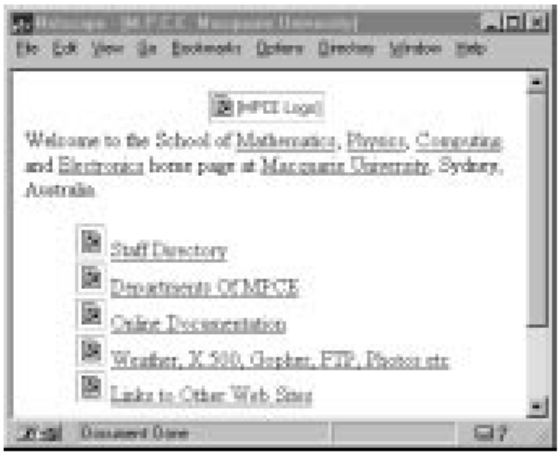
+原始对象可以同时出现在两个地方

+原始对象不需要改变

+通过代理存取原始对象是透明的

+用户可以区分原始对象和代理

-如果原始对象不可用，代理也会不可用



已知应用 Macintosh 别名和 Window95 快捷方式的行为方式象放在另外一个地方的远程代理。Netscape 使用图标作为还未下载 的图像的虚拟代理。许多网页使用缩略图来作为大图的代理。

参看 《面向模式的软件体系结构》一书也描述了代理模式。

GUI 策略（Strategy） 行为式对象

目的 定义一个算法家族，并使它们可以互换。

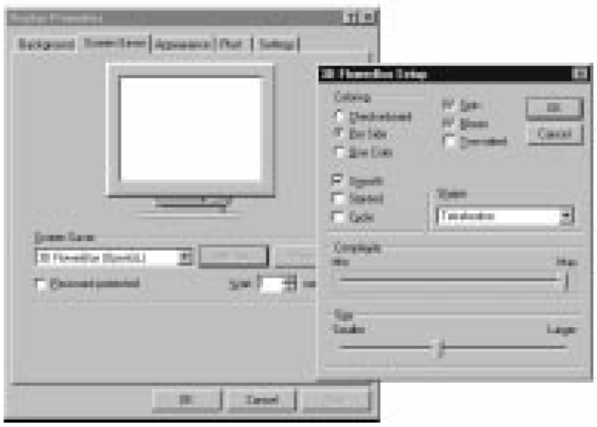
问题 一个对象使用几种算法，每个都有自己的接口和客户参数设置。用户需要为它们选择的算法设置参数。例如，一个屏保程序提供了几种不同的显示算法（文本，二维图像，三维图像）每个都有自己的参数（要显示的文本，颜色，纹理，三维对象）。 用户如何处理这些不同的算法呢？

约束 GUI 策略模式解决了如下约束：

- 一个对象需要不同参数的不同算法
- 用户应该选择算法和其参数
- 用户界面应当一致。
- 屏幕资源有限

解决方案 创建一个独立的策略对象来代表每一个算法，把每种算法的参数作为策略对象的属性。让策略对象从属于使用算法的对象。用户可以通过主对象选择一种算法，然后通过和策略对象交互来设定算法的参数。

案例 Windows NT 使用策略对象来设置屏保应用程序的参数。



结果 GUI 策略模式有如下的优缺点：

+ 策略对象显式代表了不同的算法

+用户可以选择一种算法和设置算法的参数

+在提供一个策略对象的同时保留资源

-改变策略对象也许使界面一致性变差

-接口包含了逐渐增长的对象。

已知应用 Windows NT 使用策略对象来为打印机驱动程序设置参数，同样还有屏保。PaintShop Pro 和 XV 使用策略对象来设置文件转换算法的参数。

致谢

感谢 Kent Beck，他是 EuroPLOP'97 中这篇论文的导师，感谢 Michael Richmond，Jonathon Tidswell，Geo Outhred，and Larry Constantine 为论文的草稿提出意见以及在 Macs 和 PC 方面上的帮助。

参考文献

[1] Apple Computer, Inc. Human Interface Guidelines: The Apple Desktop Interface. Addison-Wesley, 1987.

[2] Apple Computer, Inc. Inside Macintosh, volume VI. Addison-Wesley, 1991.

[3] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. Technical report, Tektronix, Inc., 1987. Presented at the OOPSLA-87 Workshop on Specication and Design for Object-Oriented Programming.

[4] Mark Bradac and Becky Fletcher. Developing form style windows. In Pattern Languages of Program Design, volume 3. Addison-Wesley, 1997.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture. John Wiley & Sons, 1996.

[6] Dave Collins. Designing Object Oriented User Interfaces. Benjamin/Cummings, 1995.

[7] Larry L. Constantine. Getting the message. Object Magazine, September 1996.

[8] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns. Addison-Wesley, 1994.

[9] Adele Goldberg. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, 1983.

[10] J. Johnson, T. L. Roberts, W. Verplank, D. C. Smith, C. Irby, M. Beard, and K. Mackey. The Xerox Star: A retrospective. IEEE Computer, 22(9), 1989.

[11] Microsoft Inc. The Windows Interface Guidelines for Software Design. Microsoft Press, 1995.

[12] Diane E. Mularz. Pattern-based integration architectures. In Pattern Languages of Program Design. Addison-Wesley, 1994.

[13] James Noble. Prototype based user interfaces. Technical report, MRI, School of MPCE, Macquarie Uni-versity, Sydney, 1996. Presented at the COTAR'96 Workshop, Melbourne, 1996.

[14] Robert Orenstein. A pattern language for an essay-based web site. In Pattern Languages of Program Design, volume 2. Addison-Wesley, 1996.

[15] ParcPlace Systems. VisualWorks Smalltalk User's Guide, 2.0 edition, 1994.

[16] Dirk Riehle and Heinz K□ullighoven. A pattern language for tool construction and integration based on the tools and materials metaphor. In Pattern Languages of Program Design. Addison-Wesley, 1994.

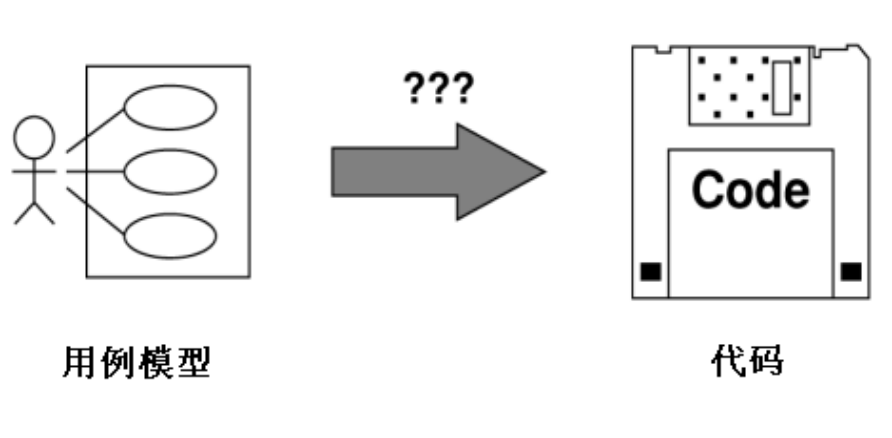
[17] Yen-Ping Shan. MoDE: A UIMS for Smalltalk. In OOPSLA Proceedings, October 1990.

[18] Randall B. Smith, John Maloney, and David Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and exibility. In OOPSLA Proceedings, 1995.

[19] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In Proceedings AFIPS Spring Joint Computer Conference, volume 23, pages 329{346, Detroit, Michigan, May 1963.

UMLChina 培训

The real thing

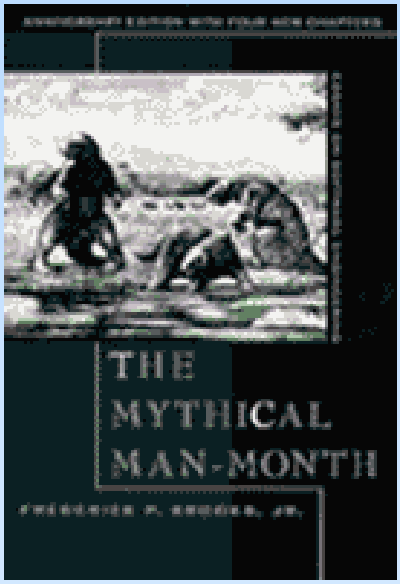


利用 UML 的 20%就可以为 80%的问题建模

--《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

《人月神话》



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 Adams Wang

散文笔法，绝无说教，大量经验融入其中

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

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

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

中文译本即将发行！

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

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

中文译本即将发行！

业务资源管理模式语言

Rosana T. Vaccare Braga 等著，zhen_lei 译

摘要

本文描述了一种处理业务资源管理的模式语言。该语言覆盖了业务系统中的大量应用，包括资源出租模式、交易模式和维护模式，并且是在实际信息系统开发的基础上设计的。在这种语言中采用了现有的可重用模式，是这些模式在这个特定的问题域的实例。我们的想法是尽可能地将这种语言设计完整，使其可以用于这个问题域中的各种应用设计。文中将 15 个模式放在同一张图中，用来检验它们使用的方便性。模式结构描述和实例描述中的对象模型都采用 UML 的描述方式。这种模式语言的应用表明它使分析变得容易，因为它是工作的指导。

引言

最近发表了许多对信息系统开发有用的通用模式，如 Type-Object[Joh 98]，Association-Object[Boy 98]，Specific-Item-Transaction，Transaction-Transaction Line Item 以及 Item-Specific Item[Coa97]。这些可复用的模式可以被广泛地应用在跨领域的信息系统应用中。

我们展现的处理资源管理应用的模式语言针对一个特定领域的信息系统。该模式语言是在总结了在该领域中 10 多年中小规模信息系统的设计经验而得出的。这些系统中存在相似性，值得考虑建立一种模式语言，使分析员在开发类似系统时使用。本文描述的模式语言，叫做业务资源管理模式语言，是许多模式的组合，其中一些是上面提到的可复用模式的实际应用。实际上，该模式语言的定位是比那些可复用模式更高的抽象层次上。它应用在特定的领域并且包含该领域中应用系统的特定术语。它给没有经验的开发人员提供开发新系统所必要的充实的材料，它在开发过程中指导他们，提供可选择的解决方案并且指出下一步需要做的工作。

业务资源管理模式语言用来帮助软件工程师开发处理业务资源管理的应用，这些应用需要记录诸如资源出租、交易或维护等过程。对于事务（transctions），我们的定义与 Coad[Coa 97]相同：一个重要的、需要记住的事件，也就是，一个系统必须按时间序列记忆的事件。资源出租主要关注物品满足某种临时需要，如内科医生的出诊时间或是为观看影片而使用的录象带。资源交易关注物品所有权的转移，例如，产品销售。资源维护关注一种特定物品的维护，使用劳动或产品，如电器维修商店。

业务资源管理模式语言

15 种模式组成了这种语言。根据具体应用的特点，在许多情况下可以选择使用。图一表现了检查这些模式的先后顺序。虽然每个模式都提供了选择方向，但作为一个整体，图一的优点是非常明显的。语言中的主要模式是“资源出租（RentTheResource）”（7），“资源交易（TradeTheResource）”（8）和“资源维护（MaintainTheResource）”（9），以粗线表示。它们互相之间不是相互孤立，实际上，在应用中它们可以安装在一起。“资源维护”可能用到“资源出租”或“资源交易”，例如在汽车维修店的应用系统中，部件是购买的，而劳力是出租的。

第一个模式是“资源标识（IdentifyTheResource）”。模式 11、12、13 表示在一个方框内说明它们适用于箭头指向它们的所有情况。指向 11 的没有源头的箭头表示该模式首先被检查，紧跟着是模式 11 和模式 12。每个模式的“下一模式”部分对图一进行详细说明。如图一所示，模式按照它们的目的分为三个部分。采用 UML（统一建模语言）[Eri 98]描述这些模式。基础类方法，创建对象、设置获得属性、添加删除对象连接以及删除对象没有包含在略图中，因为它们将增加图的复杂程度，而且效果不明显。我们假设这些方法已经在每个类中都存在了。为了加强整个系统的可理解性，我们赋予系统操作到类方法中，在已有的类中选择适合的操作功能。实际上，系统操作不仅仅是方法，它们反映了真实世界的输入系统的事件，它们的功能通过调用多个不同的类中的多种方法实现。

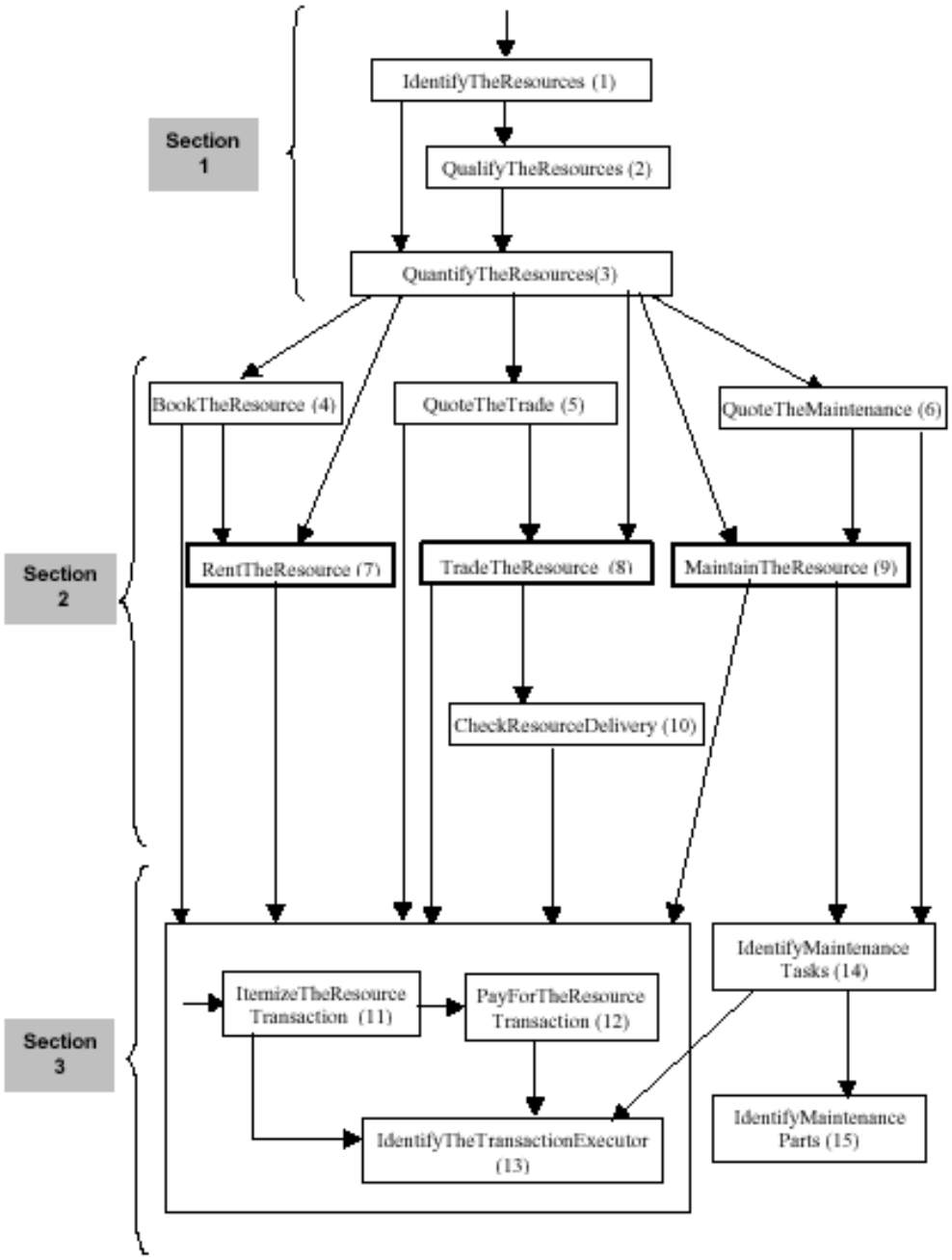


Figure 1 – Dependencies between patterns

图 1 模式间的依赖关系

第一节：最开始，关注应用中包括的资源。首先，必须**标识资源**（1），下一步，检查资源限定（2），同时**量化资源**（3）

模式 1——IdentifyTheResource（资源定义）

上下文：

你的业务系统处理如下事务：定单、销售、采购、出租、租赁、转让、预订、修理或维护。这些业务涉及到，例如，商店中的商品，出租店中的录象带，图书馆中的图书，诊所中内科医生的时间或者机械修理场中的汽车，这些资源由特定的系统管理。

问题：

如何表示系统处理的事务中的业务资源？

约束：

- 业务资源经常具有相同的属性或特征。保存每一特定资源的信息对管理这些资源的组织非常重要。
- 如果资源只有少数属性，他们可能被放置在表示组织事务的类中，这将简化实施，尽管可能限制软件发展。

结论：

评价所有事务的主题，识别出那些可以认为是资源管理的部分。

解决方案：

为每种被管理的资源创建一个“Resource”类，定义它的所用重要属性。属性“IdCode”经常作为每个资源的唯一标识，属性“Description（描述）”用来描述资源的主要特征。其它属性根据特定资源设定。除了基础类方法没有在这里出现（原因前面已经解释了），提供了特定操作如根据 IdCode 显示资源和根据 Description 显示资源，这些方法经常被业务经理使用。

略图：

图 2 显示了 IdentifyTheResource 模式的略图。“Resource（资源）”类包括所有资源的公用属性和方法。

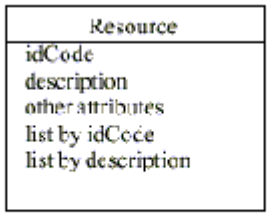


Figure 2 – IdentifyTheResource Pattern

图 2——IdentifyTheResource 模式

示例：

图 3 表示了 IdentifyTheResource 模式的实例，其中“Product”扮演“Resource”角色。

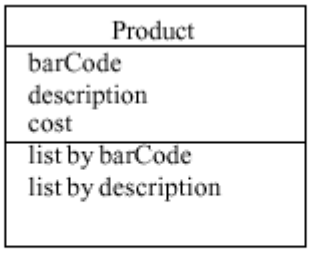


Figure 3 – Instantiation of the “IdentifyTheResource” pattern

图 3 ——IdentifyTheResource 模式的实例

下一个模式：

完成 IdentifyTheResource，下一个模式是 QualifyTheResource（2）(限定资源)

模式 2 ——QualifyTheResource（限定资源）

上下文

你已经确定了系统所要处理的所有资源和它们的主要属性。其中一些属性可能由许多资源实例共有。例如，零售店里的许多部件可能是属于同一个制造商，或者上千辆汽车属于同一个型号。

问题：

如何确定作为分类的资源性质？

约束：

- 业务资源经常需要分类。例如，在一个录象带出租商店，录象带被分类为“探险”、“悬念”、“浪漫”、“喜剧”等。这些限定条件在获得有意义的报告时非常有用。例如，用户喜欢那一类影片，希望购买那些。这种分类可以通过在资源类中添加一个属性来实现。这种方法在分类本身只是一个描述而没有自身属性时可以采用。
- 当分类属性本身具有共同的属性和方法时，将分类属性独立为一个类更加合适。为每个资源都保留相同的属性描述会导致冗余。但是，分开处理可能增加系统的处理时间。在优化系统性能时需要考虑这些问题。

结论：

评价所有业务资源的属性，确定那些起到分类作用的属性。

解决方案：

为每一个作为分类的属性建立一个“资源类型（Resource Type）”类，并且与“资源（Resource）”类相连接，

它们之间的关系必须是“多对一”，说明多个资源共享相同的资源类型。

略图：

图 4 表示 QualifyTheResource 模式。“Resource（资源）”类中添加了一个方法，根据类型列出资源。新创建的“Resource Type（资源类型）”类具有“Code（代码）”属性（可选），“Description（描述）”和由特定情况决定的其它属性。

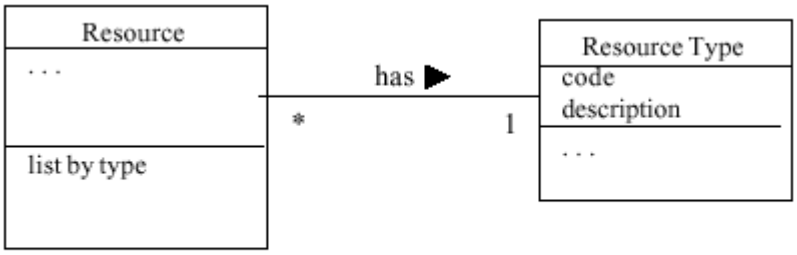


Figure 4 – QualifyTheResource Pattern

图 4——QualifyTheResource 模式

示例：

图 5 表示 QualifyTheResource 模式的一个实例，其中其中“Product（产品）”扮演“Resource（资源）”的角色，其中“Manufacturer（制造商）”扮演“Resource Type（资源类型）”角色。

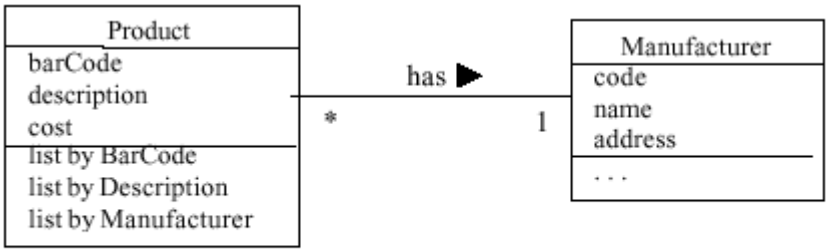


Figure 5 – Instantiation of the QualifyTheResource Pattern

图 5——QualifyTheResource 模式实例

变体：

为了实现多种分类，资源可以有多种分类方式，例如在会计系统中。为实现这种需求，可以根据分类的不同角度，引入新的类。例如，在录象带出租中，除了上面提到的分类方式，可能存在另一种分类方式，按出租率进行分类（例如，“A”表示经常出租，“B”表示出租率中等，“C”表示很少有人租）。根据这种分类就可以确定出租价。

相关模式：

“QualifyTheResource”是“Type-Object”模式[Joh 98]和 Accountability Type[Fow 97]的一个应用。

下一模式：

完成了 QualifyTheResource（或者没完成），可以进行 QuatifyTheResource（3）（量化资源）。

模式 3 ——QuantifyTheResource（量化资源）

上下文

你已经确定了系统所要处理的所有资源并且可能也确定了资源相关的性质。现在需要考虑的一个重要问题是资源的量化形式，在许多实际应用中，跟踪资源的每个实例非常重要，因为它们每个个体会独立处理。例如，图书馆中的一本图书可能有多个样本，可以借给不同的读者。一些应用处理资源的量，在这些应用中，不需要知道被处理的具体的资源实例。例如，买了一定数量的铁。在其它应用中，资源作为一个整体处理，例如，需要维护的汽车或给病人看病的医生。

问题：

如何量化资源？

约束：

- 确切地知道应用系统采用什么量化方式在分析过程中非常重要。在这个问题上的错误结果会危及以后的进展。
- 在需要跟踪资源的特定实例的情况下，冗余信息可能储存在一个资源的多个实例中，但这种冗余是非常不明智的。
- 为了避免冗余，创建一个新类用来储存相同资源所有实例的信息。但是这种方法的代价是需要处理两个类而不是一个，例如，需要更多的处理时间。

结论：

确定如何处理资源，并采用三种 QuantifyTheResource 模式中的一种建立量化模式。

解决方案：

针对这个问题，根据量化方式的不同有三种略有区别的解决方案。当识别资源实例非常重要时，需要创建一个新的附加类“ResourceInstance（资源实例）”。属性“Status（状态）”被加到“ResourceInstance（资源实例）”类中，为了控制其独立的生命周期。例如，一本书的样本的生命周期状态包括四种可能的状态：“可以使用”、“预订”、“借出”、“预订并且借出”。

当资源采用特定量进行管理时，为了处理总量控制，属性“Quantity in stock（存货量）”应该被加到“Resource（资源中）”。在这种情况下，属性“Status（状态）”不适用，因为这种情况下，系统一次处理大量的资源，不能独立地控制一个资源的生命周期。

当资源是唯一时，属性“Status（状态）”应该加到“Resource（资源）”类中用来控制生命周期，例如，一个汽车修理商店，汽车的状态可能是：“工作”、“报废”和“修理”。

略图：

图 6 表示了 QuantifyTheResource 模式的三种子模式。采用图 6（a）的子模式，叫做“ResourceInstance”。当区分资源实例非常重要时；当资源采用定量管理时采用图 6（b）的子模式，叫做“ResourceMeasurement”。当资源唯一时，采用图 6（c）中的子模式，叫做“SingleResource”。

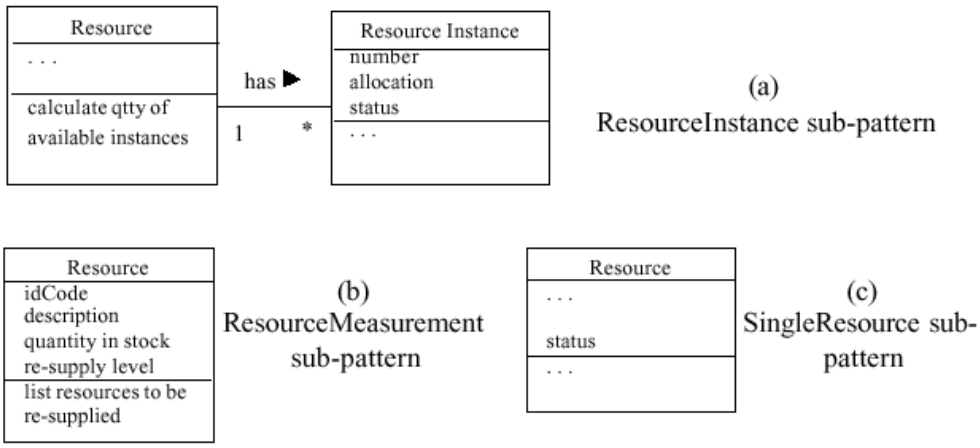


Figure 6 – QuantifyTheResource Pattern

图 6——QuantifyTheResource 模式

示例：

图 7 表示了 QuantifyTheResource 模式的实例。在图 7（a），“Video（影片）”扮演“Resource（资源）”，“Videotape（影带）”扮演“ResourceInstance（资源实例）”。在图 7（b）中，“Product”是被控制的资源，在图 7（c），“车辆”是独立管理的资源。

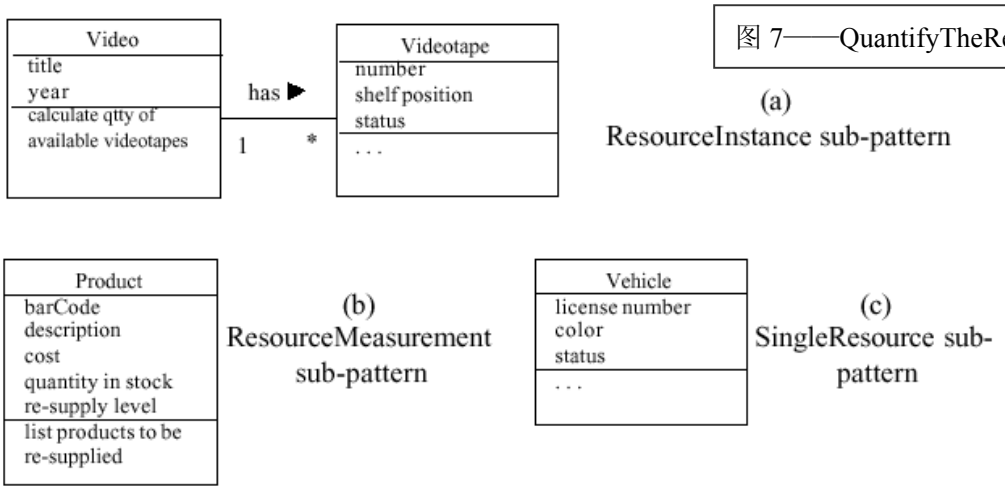


图 7——QuantifyTheResource 模式实例

Figure 7 – Instantiation of the QuantifyTheResource Pattern

相关模式：

ResourceInstance 是“Type-Object”模式[Joh98]，the ItemDescription 模式[Coa92]和 the Item-Specific Item 模式[Coa 97]的实例。

下一模式：

完成 QuantifyTheResource 模式后，应该检查所开发的应用用来处理什么类型的资源事务。如果应用关心资源交易，也就是资源的买卖，你应该采用 TradeTheResource（8）（交易资源），可能还有 QuoteTheTrade（5）（交易询价）和 CheckResourceDelivery（（10）检查资源交付）。如果应用关心出租或租赁资源，你应该采用 RentTheResource（7）（出租资源），可能还有 BookTheResource（4）（预订资源）。如果应用处理资源修理，你应该采用 MaintainTheResource（9）（维护资源）和 QuoteTheMaintenance（6）（维护询价）。注意，有许多应用需要几种模式的组合。例如，汽车出租系统，除了预订和出租汽车，我们还要控制购买、维修和报废车辆。

第二节：当处理资源时，多种事务可能参与管理。以下 7 种模式关注这些资源管理事务。一个资源可以被预订（BookTheResource（4）），可以被出租一段时间（RentTheResource（7）），可以在购买之前询价（QuoteTheTrade（5）），可以被买或卖（TradTheResource（8）），可以有交货核查（CheckResourceDelivery（10）），可以在维修前询价（QuoteTheMaintenance（6）），可以进行维修（MaintainTheResource

模式 4 ——BookTheResource（预订资源）

上下文

你的应用软件处理资源出租，该资源可以是借给顾客一段时间的物品，也可以是由专家进行服务的一段时间。你已经确定、分类、量化了应用软件需要管理的资源。一些依赖资源出租的业务系统允许预先预订资源，在你想要的时间使用它。例如，在录像带出租店，经常要预订一个影片，主要是当它被其它顾客借走时。如果它没有被别人借走，预订就没有必要了。

问题：

如何在实际出租前管理资源预订。

约束：

- 仅仅在相关资源上加入一个预订状态会将预订的顾客数限制在一个。实际上，可以有多个顾客对一个资源感兴趣。

- 登记预订细节对好的资源管理是必要的，经常被预订的资源是库房中应该增加的候选者。
- 保留旧的预订信息会增加对存储的要求，分离的预订信息会占用多的处理时间。

结论：

确定你的应用是否需要资源预订。

解决方案：

建立一个与“Resource（资源）”类关联的“Resource Booking（资源预订）”类来表明预订资源过程的细节，如日期，期限，价格等。建立一个与“Resource Booking”类关联“Source Party（来源）”类表明组织中负责预订的分支机构或部门。建立一个“Destiny-Party（目的）”类表示要求预订的顾客或客户。在这个模式中，“Source Party”为可选项，因为在小系统中组织本身负责预订，没有必要创建一个类来表示。还提供了一个用来处理老的预订的操作，避免空间的浪费。

略图：

图 8 表示“BookTheResource”模式。预订与一个来源方，一个目的方和一个资源相关。使用 ItemizeTheResourceTransaction（资源项事务）（11）模式来在一次预订中处理多个资源。在资源预订中有许多相同的属性，预订日期、客户希望租用的时间，租金（也就是顾客可以预付一部分租金），预订情况。也有公用的方法：预订资源和取消预订。以及通过来源获得预订、通过目的获得预订、通过资源获得预订等方法。如图 8 所示。

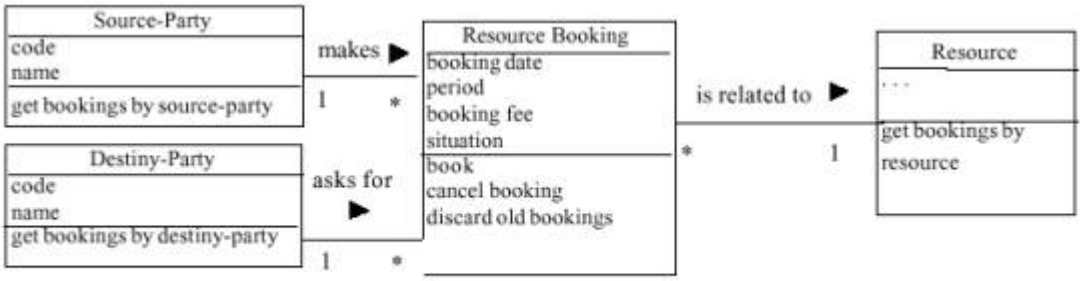


Figure 8 – BookTheResource Pattern

图 8——BookTheResource 模式

示例：

图 9 表示 BookTheResource 模式的一个实例，在一个录像带出租系统中，其中“Video（影片）”扮演“Resource（资源）”，“Video Booking（录像带预订）”扮演“Resource Booking（资源预订）”，“Branch（分店）”扮演“Source-party（来源）”，“Customer（顾客）”扮演“Destiny-Party”。

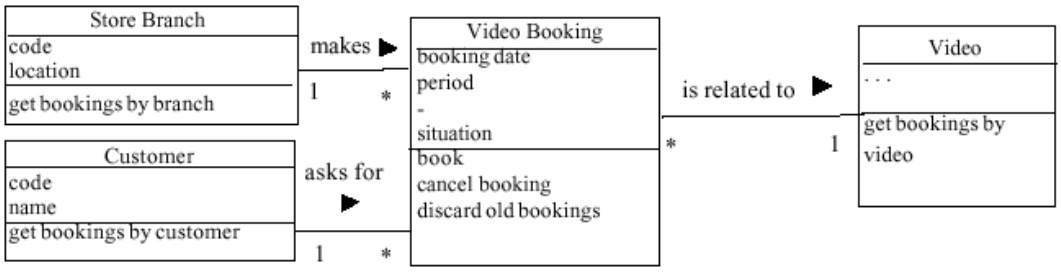


Figure 9 – Instantiation of the BookTheResource Pattern

图 9 ——BookTheResource 模式实例

变体：

预订与资源量化有关“QuantifyTheResource（3）”。例如，图书馆的情况，采用 Resource Instance 模式，预订的是资源，但实际出租的是样本。但是，一些情况下，图 8 中应将“Resource”替换为“Resource Instance”。 例如，在上面的录像带出租例子中，顾客可能希望预订原版带而不是翻译版本，这种情况下，最好不预订影片，而预订录像带本身，而录像带是资源实例。在这两个方案中进行选择非常重要。

相关模式：

“BookTheResource”是“Association-Object”[Boy 98]模式和“Time-Association”[Coa92]模式的特定应用。它也是“Participant-Transaction”和“Specific Item-Transaction”[Coa97]的组合应用。

下一模式：

在决定了是否可以进行资源预订后，下一个是“RentTheResource（7）”。也可以看第 3 节的模式，在对其它通用的事务细节建模时，它们非常有用。

模式 5 ——QuoteTheTrade（交易询价）

上下文

应用系统处理资源交易，资源可能是组织买卖的商品。你已经确定、分类、量化了应用软件需要管理的资源。买卖商品的过程是一个由付钱获得商品一方计划的活动。例如，组织经常在决定是否交易前询问商品的价格。

问题：

在成交前如何保存资源询价的过程？

约束：

- 保存询价信息对处理买和卖同样重要。首先，制作比价图来支持决策；其次，提供给用户询价服务，系统可以跟踪没有成交的询价过程，提交给组织管理者以分析可能的竞争对手。

- 处理这些信息需要附加的空间和处理时间。

结论：

确定应用系统需要对资源销售或采购进行询价。

解决方案：

创建一个与“Resource（资源）”相关的“TradeQuotation（交易询价）”类表示询价的细节。同样创建与“BookTheResource（4）”相同的“Source-Party”和“Destiny-Party”。

略图：

图 10 表示 QuoteTheTrade 模式，询价与一个来源方，一个目的方和一个资源相关。使用 ItemizeTheResourceTransaction（11）模式实现一次询价过程中处理多个资源。TradeQuotation 属性包括：询价日期、询价有效期和价格本身。有将询价与交易关联的方法。图 10 中还表示了通过来源获得询价、通过目的获得询价等方法，在“Resource”类中增加了一个通过资源获得询价的方法。

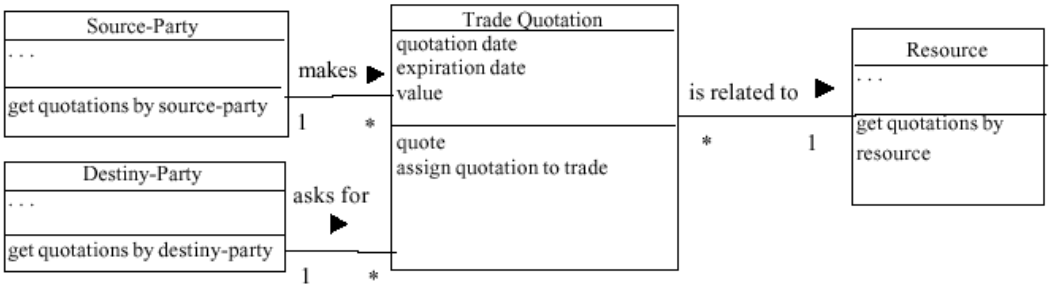


Figure 10 – QuoteTheTrade Pattern

图 10——QuoteTheTrade 模式

示例：

图 11 表示了 QuoteTheTrade 模式的一个实例，在存货控制系统中，其中“Product”扮演“Resource”，“Purchase Quotation”扮演“Trade Quotation”，“Supplier”扮演“Source-party”，“Store-branch”扮演“Destiny-Party”。

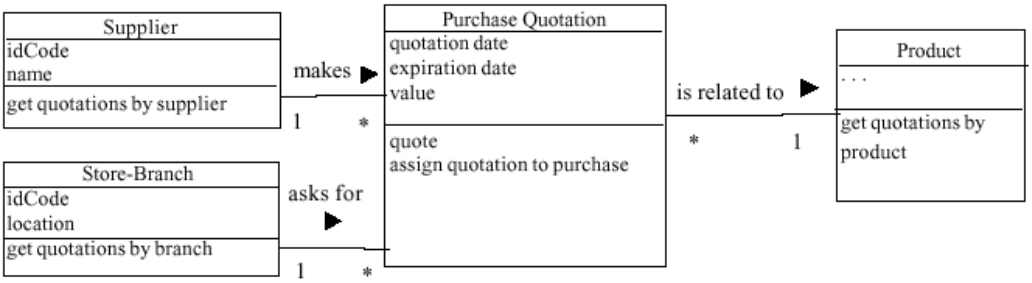


Figure 11 – Instantiation of the QuoteTheTrade Pattern

图 11——QuoteTheTrade 模式实例

变体：

根据量化模式的不同，“Resource”类可能被“Resource Instance”类替换。例如，如果资源的不同实例的价格不同，询价也不同。

相关模式：

QuoteTheResource 是“Association-Object”模式[Boy98]，和“Time-Association”模式[Coa 92]的特例。它也是“Participant-Transaction”和“Specific Item-Transaction”[Coa97]的组合应用。

下一模式：

询价后要进行的就是交易，TradeTheResource（8）。也可以看第 3 节的模式，在对其它通用的事务细节建模时，它们非常有用。

模式 6 ——QuoteTheMaintenance（维护询价）

上下文

应用系统处理资源的维护和修理。你已经确定、分类、量化了应用软件需要管理的资源。在这种情况下，资源基本上是用户的一个物品发生故障或需要定期维护。为了重新使用，它必须经过修理。或是为了防止发生故障在一定时期内必须进行维护。例如，汽车、电视、电子应用产品和计算机是在生命周期中经常发生故障的资源。许多顾客在修理前要做一下费用估算，因为如果修理费用与其本身价值相比的比率较高，修理就不值得了。

问题：

如何记录顾客针对维护的询价过程？

约束：

- 有时询价信息不重要，除非询价后有维修工作。在这种情况下，询价属性可以与维护属性放在一起。例如，在你把电视机送去修理之前，你会询问修理价格来决定是否修理。电视修理店可能想也可能不想记录一个没有修理的询价。
- 但是如果你去汽车修理店询价，你可能会付费，回答问题的工作人员会得到报酬。因此汽车修理系统需要记录询价过程，即使没有修理发生。
- 处理这些信息需要附加的空间和处理时间。

结论：

确定应用系统在实际维护前需要询价。

解决方案：

创建一个与“Resource”相关的“MaintenanceQuotation”类表示询价的细节。同样创建与“BookTheResource（4）”相同的“Source-Party”和“Destiny-Party”。

略图：

图 12 表示 QuoteTheMaintenance 模式，询价与一个来源方，一个目的方和一个资源相关。使用 ItemizeTheResourceTransaction（11）模式实现一次询价过程中处理多个资源。QuoteTheMaintenance 包括许多公用属性：询价日期、询价有效期和价格本身。还有公共的方法：询价和与维护关联（如果询价有效）。图 12 中还表示了通过来源获得询价、通过目的获得询价等方法，在“Resource”类中增加了一个通过资源获得询价的方法。

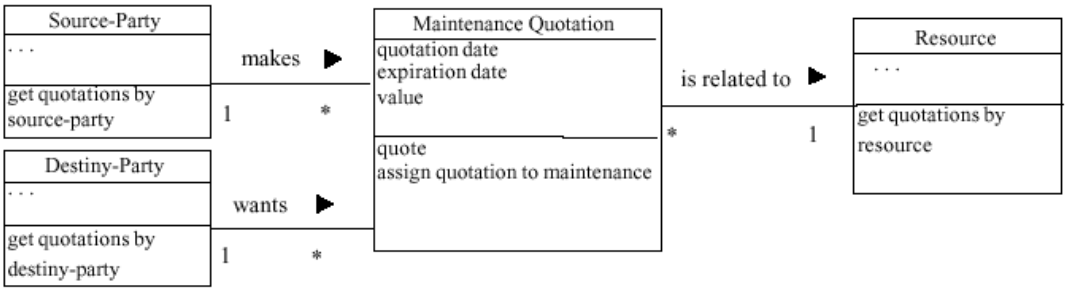


Figure 12 – QuoteTheMaintenance Pattern

图 12——QuoteTheMaintenance 模式

示例：

图 13 表示了 QuoteTheMaintenance 模式的一个实例，在汽车修理店系统中，其中“Vehicle”扮演“Resource”，“Repair Quotation”扮演“Maintenance Quotation”，“Repair shop branch”扮演“Source-party”，“Customer”扮演“Destiny-Party”。

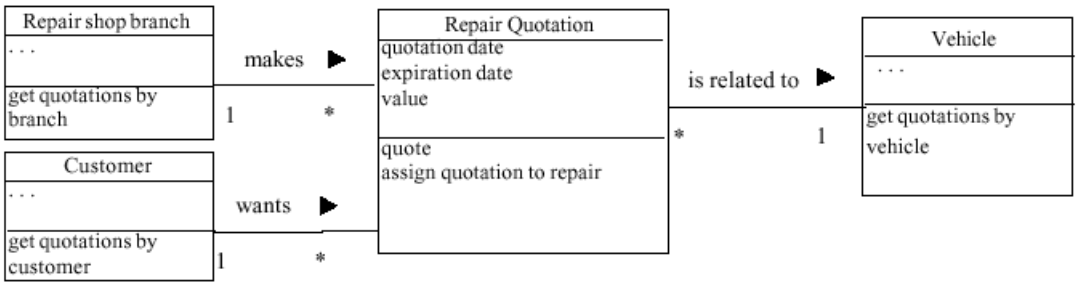


Figure 13 – Instantiation of the QuoteTheMaintenance Pattern

图 13——QuoteTheMaintenance 模式实例

相关模式：

QuoteTheMaintenance 是“Association-Object”模式[Boy98]，和“Time-Association”模式[Coa 92]的特例。它也是“Participant-Transaction”和“Specific Item-Transaction”[Coa97]的组合应用。

下一模式：

维护询价完成后就是维护资源 MaintainTheResource（8），也可以看第 3 节的模式，在对其它通用的事务细节建模时，它们非常有用。

模式 7 ——RentTheResource（资源出租）

上下文：

你的应用软件处理资源出租，该资源可以是借给顾客一段时间的物品，也可以是由专家进行服务的一段时间。你已确定出租前是否允许预订。

问题：

应用程序如何管理资源出租？

约束：

- 出租资源过程包含了很多细节信息。保存这些信息对出租资源管理非常重要。
- 出租情况的历史信息可以帮助管理者预计哪些资源值得投资。
- 必须特别注意，需要采用好的系统功能弥补附加的存储空间和处理时间。

结论：

确定应用系统是否允许资源出租。

解决方案：

创建与“Resource”类关联的“Resource Rental”类表示所有与出租相关的细节如日期、阶段、费用等。

略图：

图 14 显示了 RentTheResource 模式，出租与一个来源方，一个目的方和一个资源相关。使用 ItemizeTheResourceTransaction（11）模式实现一次出租过程中处理多个资源。如果你采用“BookTheResource”模式（4），就要将“Resource Rental”类与“Resource Booking”类以“0..1 to 0..1”关系相连接，因为预订的结果可能出租也可能不出租，出租的物品可能经过预订，也可能不经过预订。这种情况下，不需要将“Source Party”和“Destiny-party”与“Resource Rental”连接，因为这种关系与预订相关的关系相同。如果你没有采用模式（4），也就是说，你的应用软件不需要预订功能，就要创建“Source Party”和“Destiny-party”与“Resource Rental”连接分别用来表示组织分支与顾客。资源出租有共同的属性：开始日期与结束日期，顾客支付的租费。还有一些通

用的方法：出租资源，归还资源（当顾客交回资源时）和计算收入（例如每月计算）。图 14 中还显示了通过来源获得出租情况、通过目的获得出租情况等方法。

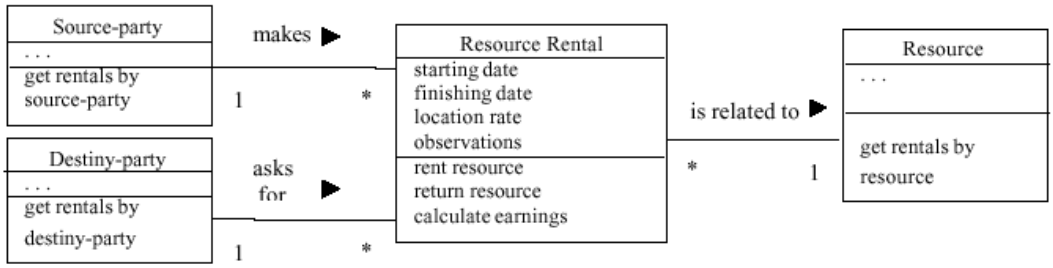


Figure 14 – RentTheResource Pattern

图 14——RentTheResource 模式

示例：

图 15 表示 RentTheResource 模式的一个实例，在一个录像带出租系统中，其中“Videotape（录像带）”扮演“Resource（资源）”，“Video Rental（录像带出租）”扮演“Resource Rental（资源出租）”，“Branch（分店）”扮演“Source-party（来源）”，“Customer（顾客）”扮演“Destiny-Party”。

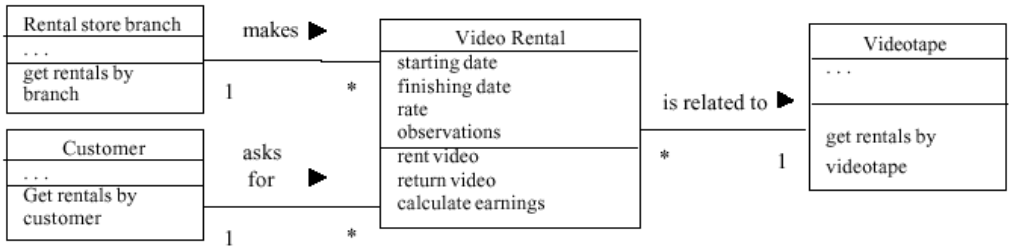


Figure 15 – Instantiation of the RentTheResource Pattern

图 15——RentTheResource 模式实例

相关模式：

RentTheResource 是“Association-Object”模式[Boy98]，和“Time-Association”模式[Coa 92]的特例。它也是“Participant-Transaction”和“Specific Item-Transaction”[Coa97]的组合应用。如果你考虑类“Resource Booking”（pattern 4）和“Resource Rental”（本模式），这里有一个“Transaction——Subsequent Transaction”pattern[Coa 97]的应用。

下一模式：

第 3 节的模式，利用它们详细说明其它细节。

模式 8 ——TradeTheResource（资源交易）

上下文

应用软件处理资源交易，可能是资源销售，也可能是购买资源。你已经确定、分类、量化了应用软件需要管理的资源，也可能应用了 QuoteTheTrade 模式（5）。资源交易可以看作资源所有权的转移，一方拥有的资源变为另一方所有。在销售过程中，如果没有库存资源，顾客可以填订购单。在采购过程中，系统需要向供应商请求，供应商在一定时间内将货物送到。

问题：

如何管理应用系统交易的资源？

约束：

记录交易信息是基本功能，因为通过交易信息可以获得重要的资源需求报告（许多这一领域的系统关心利润）。在做性能价格比分析时必须考虑到处理交易过程所需要的存储空间和时间。

结论：

确定应用系统是否有资源交易。

解决方案：

建立与“Resource（资源）”类相连接的“Resource Trade”类表示交易中的所有细节。如果采用了“QuoteTheTrade”（5），将“Resource Trade”类与“Trade Quotation”类采用“0..1 to 0…1”关系相关联。因为询价后可能成交也可能不成交，一笔交易前可能有询价过程，也可能没有询价过程。“Source-Party”总是表示资源原来的拥有者，“Destiny-Party”总是表示资源最终的拥有者。因此，在销售资源的情况下，“Source-Party”指组织的分支机构或部门，“Destiny-Party”指购买资源的顾客。在采购资源的情况下，“Source-Party”指供应商，“Destiny-Party”指采购资源的组织中的分支机构或部门。尽管应用了模式（5），你还是要将“Resource Trade”（此处原文似有误，译注）与“Source-Party”和“Destiny-Party”连接，因为询价可以进行多次，但实际的交易只有一次。

略图：

图 16 表示了 TradeTheResource 模式。交易与一个来源方，一个目的方和一个资源相关。使用 ItemizeTheResourceTransaction（11）模式实现一次交易过程中处理多个资源。“Resource Trade”状态属性描述交易进展：正在执行，部分完成，完全完成。如果采用“QuantifyTheResource”模式中的“Resource Measurement”子模式，“Resource Trade”中包含一个表示量的属性。“Resource Trade”类中除了交易、取消交易以及获得未发货交易等方法外，还包括通过来源获得交易情况，通过目的获得交易和通过资源获得交易，如图 16 所示。

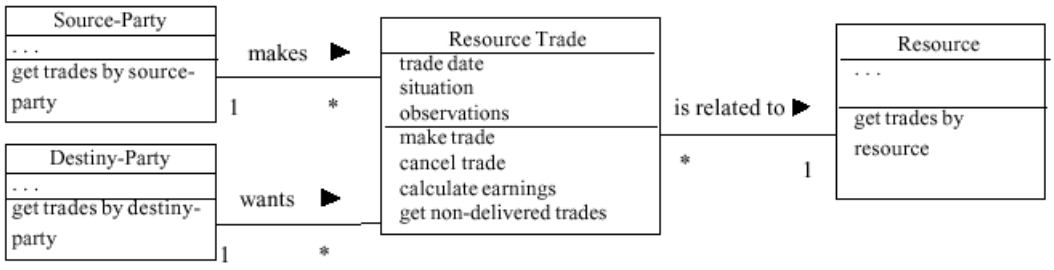


Figure 16 – TradeTheResource Pattern

图 16——TradeTheResource 模式

示例：

图 17 表示了 TradeTheResource 模式的一个实例，在存货控制系统中，其中“Product”扮演“Resource”，“Purchase”扮演“Resource Trade”，“Supplier”扮演“Source-party”，“Store-branch”扮演“Destiny-Party”。

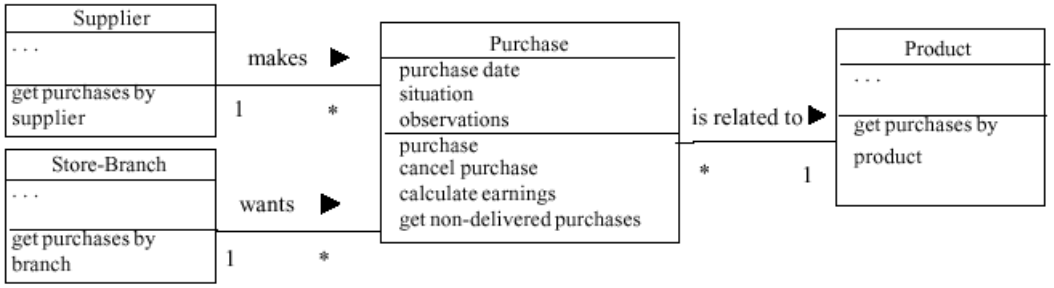


Figure 17 – Instantiation of the TradeTheResource Pattern

图 17——TradeTheResource 模式实例

相关模式：

TradeTheResource 是“Association-Object”模式[Boy98]，和“Time-Association”模式[Coa 92]的特例。它也是“Participant-Transaction”和“Specific Item-Transaction”[Coa97]的组合应用。

下一模式：

交易下一个步骤是发货，采用 CheckResourceDelivery 模式（10）。也可以看第 3 节的模式，在对其它通用的事务细节建模时，它们非常有用。

模式 9 ——MaintainTheResource（维护资源）

上下文

应用系统处理资源的维护和修理，象在模式 6 中描述的那样。你已经确定、分类、量化了应用软件需要管理的资源，也可能应用了 QuoteTheMaintainence 模式（6）。

问题：

如何在应用程序中控制资源维护？

约束：

- 保留维护信息对顾客和组织都很重要。如果维护不能令人满意的话，顾客有权投诉。为了财务上的目的，组织经常需要这些数据。如果这些信息不需要保留，一个简化处理方法是在资源类中加入一个属性保存上一次维修日期。
- 保存维修信息需要大量的存储资源，并且每个资源都有很多维修记录，需要花大量的时间来处理。

结论：

确定应用系统是否包括资源维护和修理。

解决方案：

创建与“Resource”类相关的“Resource Maintenance”类来控制维护过程。如果采用了“QuoteThe Maintenance”（6），将“Resource Maintenance”类与“Maintenance Quotation”类采用“0..1 to 0..1”关系相关联。因为询价后可能维护也可能不维护，维护前可能有询价过程，也可能没有询价过程。如果没有采用模式 6，那么创建“Source Party”和“Destiny-party”与“Resource Rental”连接分别用来表示组织分支与顾客，道理与在模式 4 中描述的一样。

略图：

图 18 表示了 MaintainTheResource 模式，维护与一个来源方，一个目的方和一个资源相关。使用 ItemizeTheResourceTransaction（11）模式实现一次交易过程中处理多个资源。在“Resource Maintenance”类中，必要的属性有“接收日期”和“离开日期”以及描述资源问题的“故障表现”。其它方法有：开始维修，注册待修资源，结束维修，完成维修，获得等待维修资源，获得未完成维修资源。通过来源获得维修、通过目的获得维修以及通过资源获得维修等方法添加到相关类中，如图 18 所示。

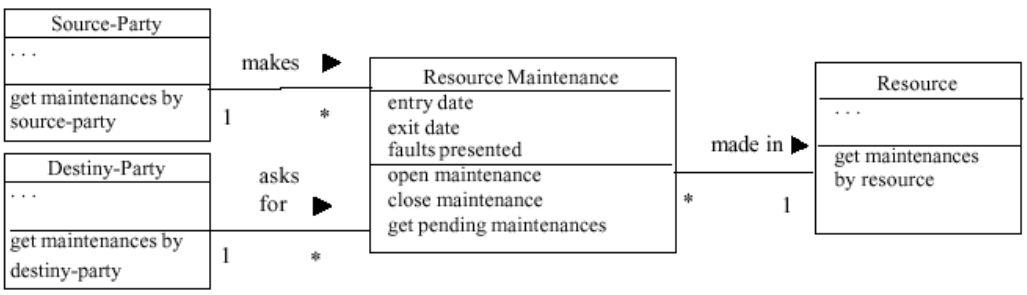


Figure 18 – MaintainTheResource Pattern

图 18 ——MaintainTheResource 模式

示例：

图 19 表示了 MaintenanceTheResource 模式的一个实例，在汽车修理店系统中，其中“Vehicle”扮演“Resource”，“Repair log”扮演“Resource Maintenance”，“Repair shop branch”扮演“Source-party”，“Customer”扮演“Destiny-Party”。

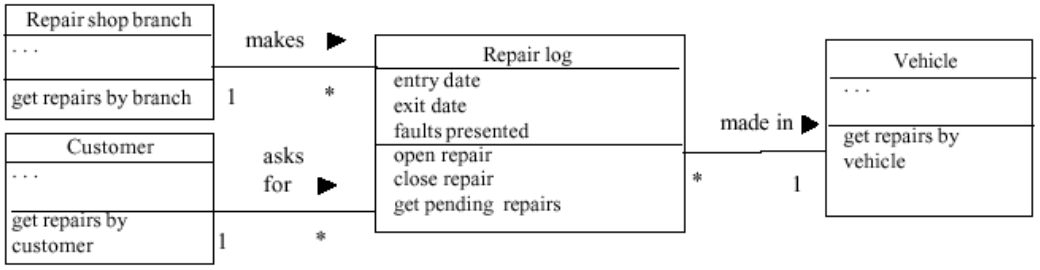


Figure 19 – Instantiation of the MaintainTheResource Pattern

图 19 ——MaintainTheResource 模式实例

相关模式：

MaintainTheResource 是“Association-Object”模式[Boy98]，和“Time-Association”模式[Coa 92]的特例。它也是“Participant-Transaction”和“Specific Item-Transaction”[Coa97]的组合应用。如果你考虑类“Maintenance Quotation”（pattern 6）和“Resource Maintenance”（本模式），这里有一个“Transaction——Subsequent Transaction”pattern[Coa 97]的应用。

下一模式：

第 3 节的模式，利用它们详细说明其它细节。

模式 10 ——CheckResourceDelivery（资源交付检查）

上下文

应用软件处理资源交易，可能是资源销售，也可能是购买资源。你已经确定、分类、量化了应用软件需要管理的资源，也可能应用了 QuoteTheTrade 模式（5），并且已经应用了 TradeTheResource 模式（8）。在有些系统中，需要提供一种检验交易发货的机制。例如，当你进行了一次采购，还没有获得货物，你会收到详细的采购清单。必须建立一种机制在增加库存前，检查与采购相关的交货。

问题：

如何管理应用系统的资源交付？

约束：

- 在有些应用系统中，只有当交货完成后才进行交易登记。因此，在其间不知道交易的事情。这就导致许多报告中缺乏细节信息。这种方法，虽然经济，但不反映实际情况。
- 如果检查仅仅是为了确认交易，在交易中添加一个表明交货日期的属性就可以了。但是，如果与交货相关的信息很多，并且对系统的效率有影响，那么就需要系统记录下来。
- 知道以前的交货情况对选择供应商非常重要。
- 当交货独立注册时存储空间和处理时间都要增加。

结论：

确认应用系统是否包括交货检查。

解决方案：

创建一个与“Resource”类相关的“Resource Delivery”类来控制资源交付的确认过程。因为交货是与交易相关的，“Resource”类与“Resource Delivery”类是“1 to 1”关系。如果你采用了子模式“Resource Measurement”或者采用了“ItemizeTheResourceTransaction”模式，这种关系会发生变化。

略图：

图 20 表示了 CheckResourceDelivery 模式。一次交货必然与一次交易相关，一次交易后必然有一次交货。使用 ItemizeTheResourceTransaction（11）模式实现一次交易过程中处理多个资源。除了“Resource Delivery”中进行交货和取消交货方法外，根据资源获得交货方法被添加到“Resource”类中，将交易与交货相关的方法在“Resource Trade”类中，根据来源获得交货与根据目的获得交货应该放置在“Source-Party”和“Destiny-Party”中。

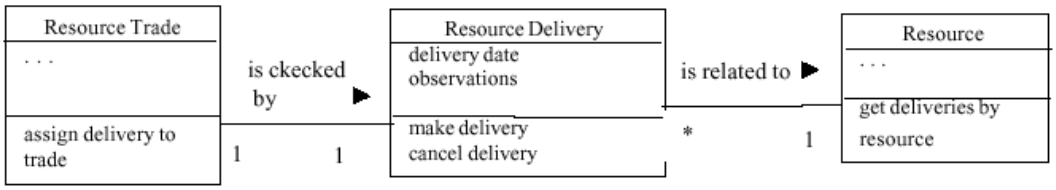


Figure 20 – CheckResourceDelivery Pattern

图 20——CheckResourceDelivery 模式

示例：

图 21 表示了 CheckResourceDelivery 模式的一个实例，其中“Product（产品）”扮演“Resource（资源）”，“Delivery（交付）”扮演“Resource Delivery（资源交付）”，“Purchase（购买）”扮演“Resource Trade（资源交易）”。

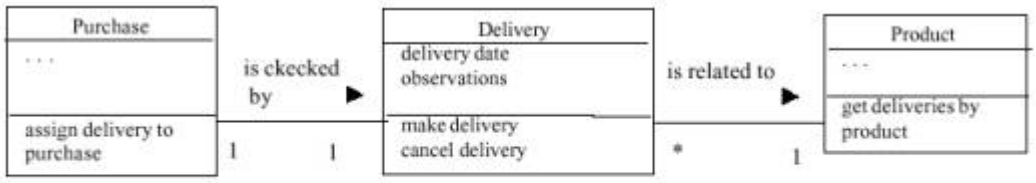


Figure 21 – Instantiation of the CheckResourceDelivery Pattern

图 21——CheckResourceDelivery 模式实例

相关模式：

CheckResourceDelivery 模式是“Association-Object”模式[Boy98]，和“Time-Association”模式[Coa 92]的特例。它也是“Participant-Transaction”和“Specific Item-Transaction”[Coa97]的组合应用。如果你考虑类“Resource Trade”（模式 8）和“Resource Delivery”（本模式），这里有“Transaction-Subsequent Transaction”模式[Coa 97]的一个应用。

下一模式：

第 3 节的模式，利用它们详细说明其它细节。

第三节：在前面讨论的 Resource Transcations（资源事务）中有许多共同的行为。其中一个行为可以包含多个项目，每个项目对应一个不同的资源（ItemizeTheResourceTransaction（11）。事务可以产生一些报酬（PayForTheResourceTransaction（12））。事务也可能由一个人或小组完成，这对系统非常重要（IdentifyTheTranscationExecutor（13））。当一个资源需要维护，我们必须关注维护所必需的人力服务（IdentifyMaintenanceTasks（14））和这个过程中所使用的部件（IdentifyMainanceParts（15））。实际上，人力服务和部件都是在维护中可见的资源，被维护的资源是一个属于顾客的物品。

模式 11 ——ItemizeTheResourceTransaction（记录资源事务）

上下文

应用系统管理资源而你已经采用了第 2 节中的一个或多个模式。在一些应用中，一次事务可能要包含对多个资源的处理，例如，一个顾客在录像带出租店可以一次租多个录像带。或者，当向供应商采购时，一次可能采购多种产品。因此，允许一次事务包含多个条目会让人感到方便。表 1 中列出了第 2 节中可能遇到这种情况的模式中和模式中的类。

表 1——可能的资源事务

Table 1 – Possible Resource Transactions

Pattern	Resource Transaction class
BookTheResource (4)	Resource Booking
QuoteTheTrade (5)	Trade Quotation
QuoteTheMaintenance (6)	Maintenance Quotation
RentTheResource (7)	Resource Rental
TradeTheResource (8)	Resource Trade
MaintainTheResource (9)	Resource Maintenance
CheckResourceDelivery (10)	Resource Delivery

问题：

如何在一个事务中管理多个资源？

约束：

- 当需要在一次事务中处理多个资源时，只在表示事务的类中加入数量属性不是一个可行的方案：它只能处理在一次事务中对同一种资源不同单元的情况。
- 在有些应用中，没有必要在一次事务中包括多个项目：例如，在汽车修理店中，一个顾客一般一次只修理一辆汽车。在这种情况下，可能的例外可以当作两个或多个事务处理，因为为这种很少出现的情况花费过多不值得。

结论：

确定一个事务中是否需要处理多个资源。

解决方案：

建立一个“Transaction Item”类聚合到“Resource Transaction”中，控制一个事务中的多个项目。

略图：

图 22 表示了 ItemizeTheResourceTransaction 模式，“Resource Transaction”类和“Resource”类（或是某些情况下“Resource Instance”类）间的连接应去掉，建立从“Transaction Item”类到“Resource”类（或是 “Resource Instance”类）间的连接，如图 22 所示。“Transaction Item”类表示一次事务中处理的多个项目。仅当采用“ResourceMeasurementPattern（3b）”时，它有可选属性“数量”和“价格”。在“Resource Transaction”类中增加了一个“统计事务条目”的方法。

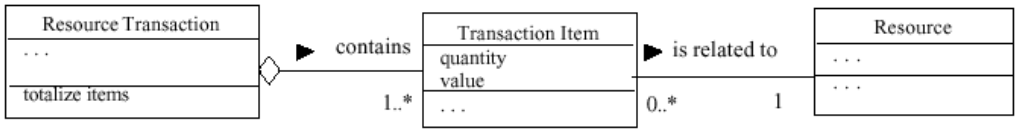


Figure 22 – ItemizeTheResourceTransaction Pattern

图 22——ItemizeTheResourceTransaction 模式

示例：

图 23 表示 ItemizeTheResourceTransaction 模式的一个实例，其中“Product”扮演“Resource”，“Delivery”扮演“Resource Delivery”，“Purchase”扮演“Resource Transaction”，“Purchase Item”扮演“Transaction Item”。

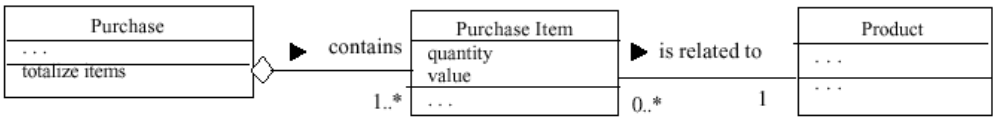


Figure 23 – Instantiation of the ItemizeTheResourceTransaction Pattern

图 23——ItemizeTheResourceTransaction 模式实例

相关模式：

ItemizeTheResourceTransaction 模式是“Behavior across a Collection”模式，和“State across a collection”模式[Coa 92]的特例。如果你考虑类“Resource Transaction”和“Transaction Item”（本模式），这里有“Transaction Line Item”模式[Coa 97]的一个应用。如果你考虑类“Transaction Item”和“Resource”，这里有“Item-Line Item”模式[Coa97]的一个应用。

下一模式：

下一步考虑是否采用“PayForTheResourceTransaction”（12）和 IdentifyTheTransactionExecutor（13）。

模式 12 ——PayForTheResourceTransaction（资源事务支付）

上下文

应用系统管理资源而你已经采用了第 2 节中的一个或多个模式。许多资源交易有多种支付方式。例如，租房子，购买冰箱或者修汽车，顾客要付一大笔钱。如果一时拿不出全款，一些应用系统提供分期付款。其结果是管理更加复杂，要引入对收入和分期付款的控制。

问题：

如何控制与资源交易相关的付款？

约束：

- 只在“Resource Transaction”类中加入一些属性来控制付款往往是不够的，对收入与分期付款进行好的管理需要更精确的信息。
- 保存顾客的历史数据对组织确定顾客的信用起更好的支持作用。
- 对每笔分期付款独立处理将增加系统的难度。这种情况必须考虑，例如，所有的顾客支付现金。

结论：

确定资源事务可以采用分期付款。

解决方案：

如果一次付款，只要在“Resource Transaction”类中添加“付款日期”和“金额”就可以了。否则，就要创建一个与“Resource Transaction”类相关的“Payment”类跟踪每一笔付款。

略图：

图 24 表示了 PayForTheResourceTransaction 模式。方法“确定支付方式”添加到“Resource Transaction”中，确定顾客希望分几期付款以及相应的利息。“Payment”类中包括属性“开始日期”、“付款日”、“分期序号”、“金额”和“状态”。最后一个属性控制分期付款的可能状态，例如，已经付清、拖期等。方法“Coming installments”列出了将要分期付款的日期。方法“Overdue Payment”列出了所有拖期的付款。方法“Register Payment”负责完成客户的付款。方法“Payment done”列出在一定时期内所有的付款。

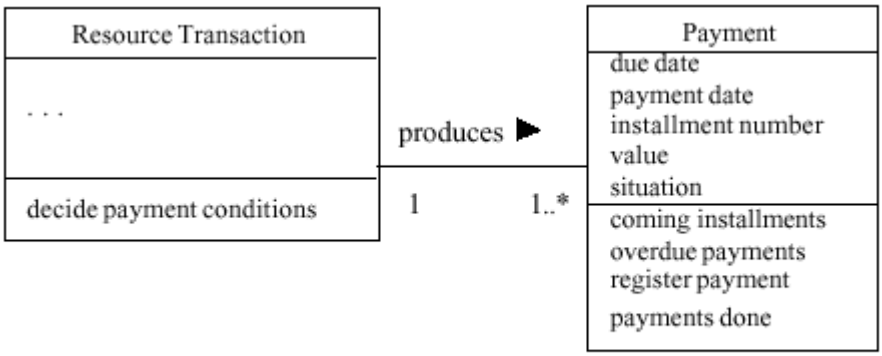


Figure 24 – PayForTheResourceTransaction Pattern

图 24——PayForTheResourceTransaction 模式

示例：

图 25 描述了 PayForTheResourceTransaction 的一个实例。其中，“Sale”扮演“Resource Transaction”，“Accounts Receivable”扮演“Payment”。

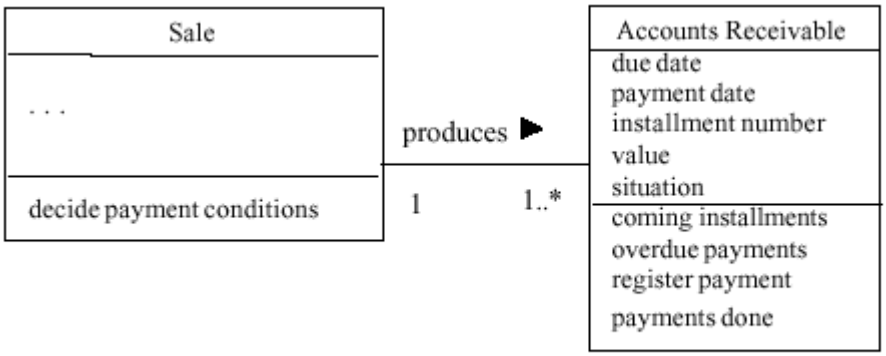


Figure 25 – Instantiation of the PayForTheResourceTransaction Pattern

图 25——PayForTheResourceTransaction 模式实例

相关模式：

PayForTheResourceTransaction 是“Transaction-Subsequent transaction”模式的一个特例。

下一模式：

下一个可能的模式是 IdentifyTheTransactionExecutor（13）

模式 13 ——IdentifyTheTransactionExecutor（识别事务执行者）

上下文

应用系统管理资源，而你已经采用了第 2 节中的一个或多个模式。许多资源交易有多种支付方式。对于实际系统，知道是谁完成的事务非常有用。例如，在一个计算机商店，销售人员卖出计算机并且按照每周或每月获得佣金。因此，为了提供佣金报表，系统需要这些信息。

问题：

如何识别事务的执行者？

约束：

- 在“Resource Transaction”类中加入“执行者”属性对于仅关心执行者姓名的小系统来说是一个好方案。但是在一些系统中，“执行者”具有管理所必须的其它属性，例如，固定的薪水，特殊佣金比例，最小销售额等。
- 只有当应用系统需要时，才值得将每个执行者信息分开存放，因为这样会需要更多的存储空间和处理时间。

结论：

确定执行者对系统是否重要。

解决方案：

创建与“Resource Transaction”类（表 1 中所示）相关联的“Transaction Executor”类，表示可能的事务负责人或团队。

略图：

图 26 表示了 IdentifyTheTransactionExecutor 模式，提供与特定事务相关佣金的方法被添加到“Resource Transaction”类中。“Transaction Executor”类有属性代码、姓名、专业、佣金比例、最小销售额和薪水，还有通过执行者获得事务方法，和列出已付佣金方法。

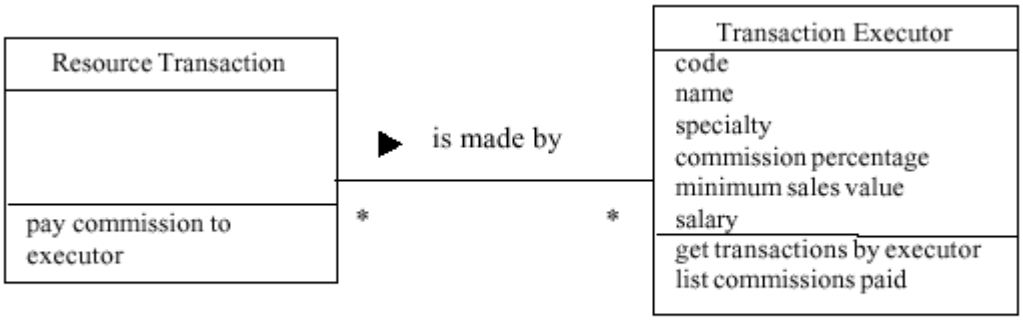


Figure 26 – IdentifyTheTransactionExecutor Pattern

图 26——IdentifyTheTransactionExecutor 模式

示例：

图 27 表示 IdentifyTheTransactionExecutor 模式的一个实例，其中，“Sale”扮演“Resource Transaction”，“Salesman”扮演“Transaction Executor”。

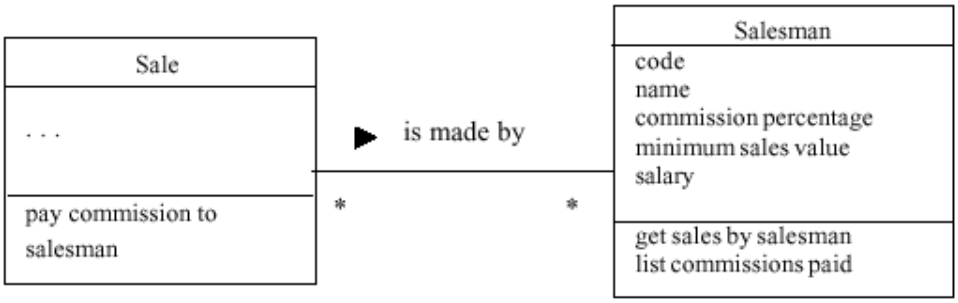


Figure 27 – Instantiation of the IdentifyTheTransactionExecutor Pattern

图 27——IdentifyTheTransactionExecutor 模式实例

相关模式：

IdentifyTheTransactionExecutor 是“Participant-Transaction”模式的一个特例。

变体：

大型应用软件中，执行者可能是一个小组，因此可能有必要包括多个类，与 Transaction Executor 连接，控制小组成员间的佣金分配。

下一模式：

如果你的业务与维护，那么确定是否需要 IdentifyMaintenanceTasks（14）和 IdentifyMaintenanceParts（15）。否则，检查表 1，看看是否有模式没有包括在内。

模式 14 ——IdentifyMaintenanceTasks（确定维护任务）

上下文

应用软件处理资源维护，已经应用了 MaintainTheResource(9)和其它可选模式 6，11，12，13（或者是这些模式的组合）。当一个资源出现故障需要维护时，经常需要人力服务。例如，一辆有刹车故障的汽车可能需要更换刹车版，调节刹车线或者需要添加润滑油。有些情况下，每一种服务由不同的人提供。因此，确定资源维护过程中的任务就非常重要了。

问题：

如何确定维护业务或维护询价过程中的任务？

约束：

- 如果只需要关于维护动作的少量信息，只要在维护中加一个属性就可以了。但是，这样所有的维护情况就被限定在一定数量的任务内。如果数量少，可能不会覆盖所有情况，如果数量大，就会浪费空间。
- 许多系统希望将每个维护动作独立处理，这样可以使不同的维护情况便于控制和比较。这种信息可以提高对新情况的报价和工作安排。

结论：

确定资源维修是否包括多个任务，可能由不同的人执行。

解决方案：

为“Resource Maintenance”建立一个聚合类“Maintenance Task”，带有属性“需要解决的问题”，“人力描述”，“花费时间”和“成本”。

略图：

图 28 表示 IdentifyMaintenanceTasks 模式，一个维护可以有多个任务。“维护执行者”类是可选的，相当于图 26 中的“Transaction Executor”类。是否使用该类取决于 IdentifyTheTransactionExecutor 模式。如果采用，将它与“Maintenance Task”类（如果每个任务由不同的人完成）相连，如图 28 所示，或是与“Resource Maintenance”类（如果整个维护工作由一个执行者完成）相连。

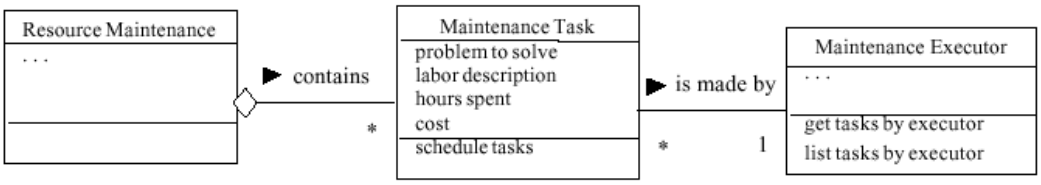


Figure 28 – IdentifyMaintenanceTasks Pattern

图 28——IdentifyMaintenanceTasks 模式

示例：

图 29 表示 IdentifyMaintenanceTasks 模式的一个实例，其中“Veicle repair”扮演“Resource Maintenance”，“Labor task”扮演“Maintenance task”，“Repairman”扮演“Maintenance Executor”。

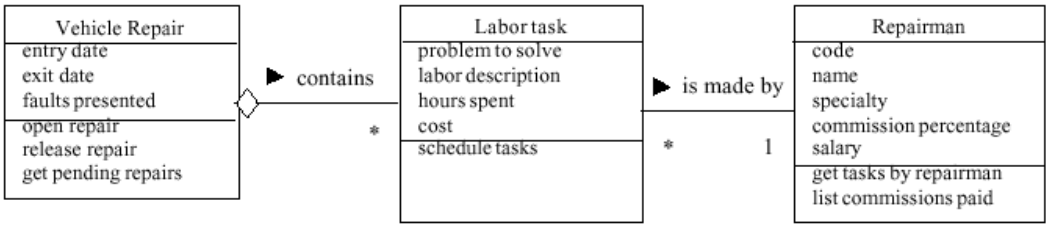


Figure 29 – Instantiation of the IdentifyMaintenanceTasks Pattern

图 29——IdentifyMaintenanceTasks 模式实例

相关模式：

如果你考虑类“Resource Maintenance”和“Maintenance Task”，那么是“Transaction-Transaction Line Item”模式的一个特例[Coa 97]。

下一模式：

确定是否需要 IdentifyMaintenanceParts（15）模式。否则，检查表 1，看看是否有模式没有包括在内。

模式 15 ——IdentifyMaintenanceParts（确定维护部件）

上下文

应用软件处理资源维护，已经应用了 MaintainTheResource(9)和其它可选模式 6，11，12，13（或者是这些模

式的组合)。在资源维修过程中，资源的某些部件可能需要更换，因为出了故障或是就要出故障。例如，如果汽车的刹车除了问题，刹车版可能需要更换，润滑油需要补充。在这种情况下，区分资源维护中使用的部件非常重要。

约束：

- 在有存量控制子系统的应用中，需要区分在维护中使用的部件，因为这些信息可以应用在存量控制中，用来减少库存，因此该信息连接了两个子系统。将部件单独处理可以使保修控制变得容易，虽然系统需要的存储空间和处理时间要增加。
- 另一方面，如果部件没有被任何子系统记录，那么维护中使用的部件可能要作为维护的一个属性来记录。采用这种方法要么限制部件在每次维修中的数量，要么建立一个固定的清单使用列表。

结论：

确定知道资源维护过程中的部件是否必要。

解决方案：

创建与“Resource Maintenance”类相关的“Part used in maintenance”类集合。创建“Part”类包含所有组织中可用部件。

略图：

图 30 表示 IdentifyMaintenanceParts 模式，每个资源维护可能用到多个部件，维护中使用的每个部件对应库存中的一种部件。“Resource Maintenance”类中增加了计算部件总量的方法。

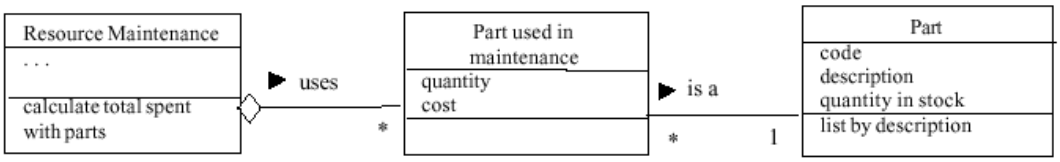


Figure 30 –IdentifyMaintenanceParts Pattern

图 30——IdentifyMaintenanceParts 模式

示例：

图 31 表示 IdentifyMaintenanceParts 模式的一个实例，其中“Vehicle repair”扮演“Resource Maintenance”，“Part used”扮演“Part used in maintenance”，“Part”扮演“Part”。

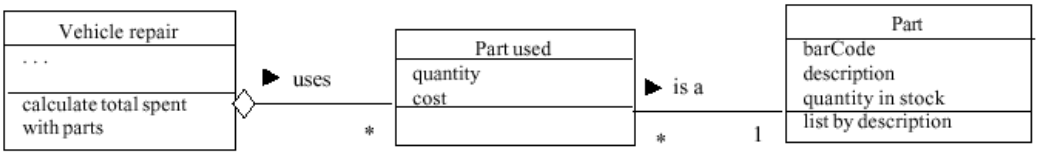


Figure 31 – Instantiation of the IdentifyMaintenanceParts Pattern

图 31——IdentifyMaintenanceParts 模式实例

相关模式：

如果你考虑类“Resource Maintenance”和“Part used in maintenance”，那么是“Transaction-Transaction Line Item”模式的一个特例[Coa 97]。如果你考虑类“Part”和“Part used in maintenance”，那么是“Item Line Item”模式的一个特例[Coa 97]。

下一模式：

现在，检查表 1，看看是否有模式没有包括在内。

一个应用实例

采用模式语言对一个小的汽车修理店应用系统建模。使用了模式（1）、（2）、（3b）、（3c）、（8）、（9）、（11）、（12）、（13）、（14）和（15）。图 32 描述了最终对象模型。表 2 汇总了模式语言在这个应用中的关系。按从上到下的顺序阅读“Repair subsystem”和“Purchase subsystem”，这个顺序与模式语言的描述一致。最终的对象模型可以为其它系统，如“Accounts Payable”/“Accounts Receivable”以及“Purchased part”/“Requested Parted”提供生成类。

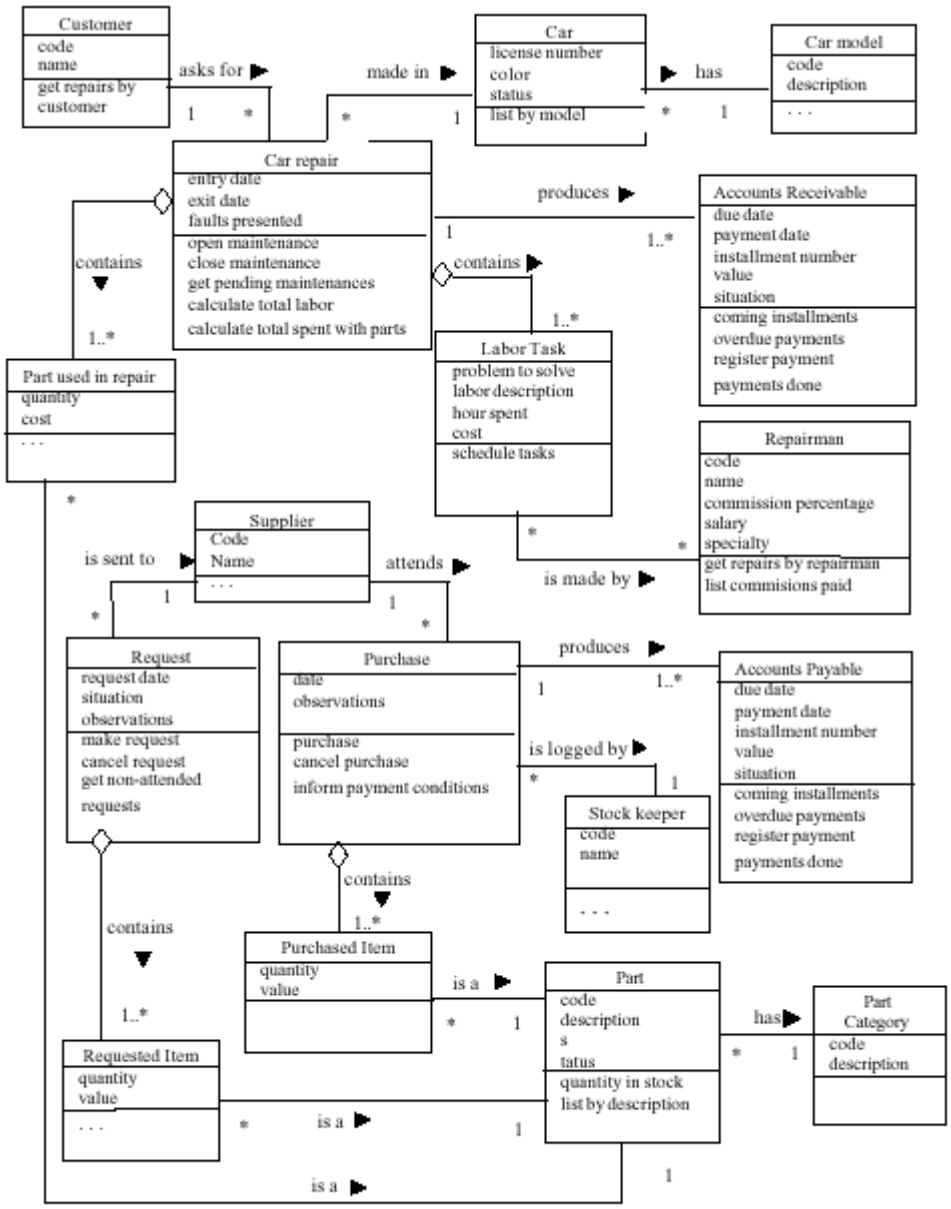


Figure 32 – Application of the Pattern Language to a simple Car Repair Shop

Table 2 – Summary of the pattern language application results

Pattern	Class Participant	Repair subsystem	Purchase subsystem	
1 – IdentifyTheResource	Resource	Car	Part	
2 – QualifyTheResource	Resource type	Car model	Part category	
3b – QuantifyTheResource			No classes added, only attributes	
3c – QuantifyTheResource		No classes added, only attributes		
8 – TradeTheResource	Source-Party Destiny-Party Resource Trade		Supplier - Request	
9 – MaintainTheResource	Source-Party Destiny-Party Resource Maintenance	- Customer Car repair		
10 – CheckTheDelivery	Resource delivery			Purchase
11 – ItemizeTheResource Transaction	Transaction item		Requested item	Purchased item
12 – PayForTheResource Transaction	Payment	Accounts receivable		Accounts payable
13 – IdentifyTheTransaction Executor	Transaction executor	Repairman		Storekeeper
14 – IdentifyMaintenanceTasks	Maintenance task Maintenance executor	Labor task Repairman		
15 – IdentifyMaintenanceParts	Part used in maintenance Part	Part used in repair Part		

总结

本文的模式语言反映了十年资源管理系统开发的职业经验。它的应用使分析新系统变得容易，因为它为系统分析提供了指南，包括了这一领域需要注意的主要问题。我们计划扩展这种语言，包括仓储管理和更好地处理付款，基于这种语言的框架也会开发出来。

✧ **致谢**

本文开始于另一篇文章。我们感谢 Norm Kerth 在这个过程中，建议将模式细分为本文所描述的模式语言。我们还要感谢 Bruce Whitenack 的支持，他是本文的指导。

✧ **References**

[Boy 98] Boyd, Lorraine. *Business Patterns of Association Objects*. In “Martin, Robert C. (ed.); Riehle, Dirk (ed.) and Buschmann, Frank (ed.) Pattern Languages of Program Design 3”, Addison-Wesley, pp. 395-408, 1998.

[Braga 98] Braga, Rosana T. V.; Germano, Fernão, S.R.; Masiero, PauloC. *A Confederation of Patterns for Resource Management.* Workshoped in the 5 th Conference on Pattern Languages of Programs (PLoP’98), Monticello-Illlinois, August 1998.

[Coa 92] Coad, Peter. *Object-Oriented Patterns*. Communications of the ACM, V. 35, nº9, pp. 152-159, September 1992.

[Coa 97] Coad, P.; North, D.; Mayfield, M. *Object Models: Strategies, Patterns and Applications*, Yourdon Press, 2 nd edition, 1997.

[Eri 98] Eriksson, Hans-Erik and Penker, Magnus. *UML Toolkit*, Wiley Computer Publishing, 1998.

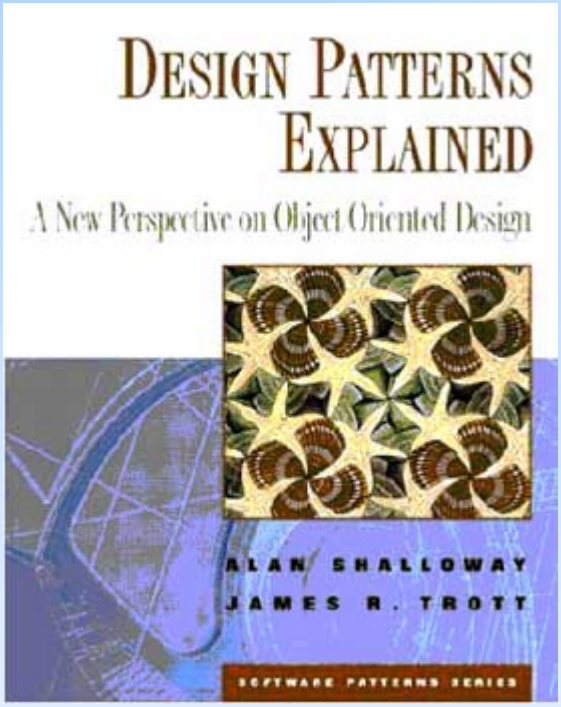
[Fow 97] Fowler, Martin. *Analysis Patterns*. Addison-Wesley, 1997.

[Joh 98] Johnson, Ralph and Woolf, Bobby. *Type Object*. In "Martin, Robert C. (ed.); Riehle, Dirk (ed.) and Buschmann, Frank (ed.) Pattern Languages of Program Design 3", Addison-Wesley, pp. 47-65, 1998.

看不懂《设计模式》？

它的作者推荐《设计模式精解》

透明 译

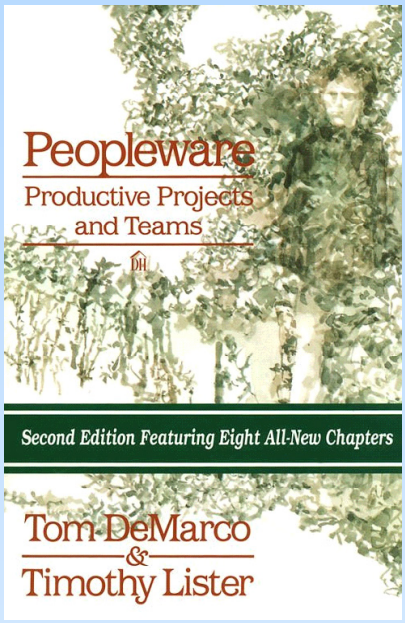


直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 **Alan Shalloway** 的《设计模式精解》。

--John Vlissides 写于 UMLChina 答疑板

中译本即将上市！

《人件》



《人件》第 2 版

Tom Demarco 和 Tim Lister

翻译：UMLChina 翻译组方春旭、叶向群

微软的经理们很可能都读过—amazon.com

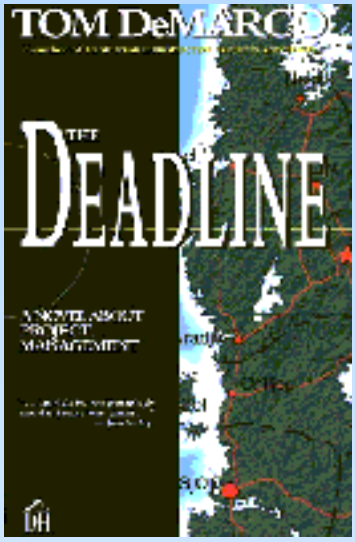
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

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

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

中文译本即将发行！

《最后期限》



《最后期限》

Tom Demarco

翻译：UMLChina 翻译组 透明

这是一本软件开发小说

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

....

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

莱克莎点点头："这是最近的数字。他们都会在你的手下工作。"

"而且据你所说，他们都很优秀。"

"他们都通过了摩罗维亚软件工程学院的 CMM 2 级以上的认证。"

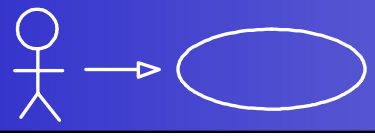
中文译本即将发行！

面向对象？

我们编写COBOL程序。以前我们用COBOL编写COBOL程序，现在我们用C++编写COBOL程序。

--《实用面向对象软件工程》

Edward Yourdon



用户界面设计从抽象到实现

——基于规范抽象组件的抽象原型

Larry Constantine,* Helmut Windl, James Noble, Lucy Lockwood 著，Lin George 译

2000 年 7 月，一群同行召集起来就以使用为中心的（usage-centered）设计的现状和未来开了个会。这次会议不仅作为一个论坛来回顾和巩固在以使用为中心方面累积的经验，它更是一个研讨会，用来精炼和改进这种方法，特别强调和其他设计开发过程与模型的交叉渗透。那次讨论衍生出许多概念上和实践上的突破，其中之一是抽象原型，很引人注目的一种改进形式。它基于任务模型，简化并加速了高质量用户界面设计的生产。

抽象原型

抽象原型是以使用为中心设计的一个强有力工具[Constantine, 1998; Constantine & Lockwood, 1999]。抽象原型允许设计者描述一个用户界面的内容和全局组织，而无需详细说明其外观或行为。因此，它是待设计用户界面结构的模型。我们和我们的客户发现，抽象原型是一座有效的桥梁，它沟通了基于任务案例（本质用例）的任务模型和实际原型形式的最终设计。在文章和在软件中都是这样。尤其通过集中关注内容、组织和功能，而非布局、外观和行为，我们一再认识到，抽象原型有助于可靠的体系结构和创造性的变革[Constantine, 1998]。如果有恰当的任务模型作驱动，抽象原型能帮助设计者作出实用而新颖的用户界面解决方案[Constantine, 2000]。

当前技术水平状况

在进行以使用为中心的设计时，内容模型和导航图构成了抽象原型最常见的组成部分。内容模型包含一系列由抽象组件组装成的视图（交互语境），抽象组件亦即用户完成每个视图支持任务所需的工具和原料。导航图对内容模型进行补充，它表现用户界面里所有视图（交互语境）相互连接时可能的路径和转换。

常规的内容模型中，最典型的是贴纸形式，每个视图通常由一张单独的标签纸表示，纸上粘贴着抽象组件。简单图示符（小图标）用来区分原料和工具，原料给用户提供感兴趣的容器和信息，工具则为用户操作这些原料或执行其他的动作。图 1 是常规抽象原型的一个示例。

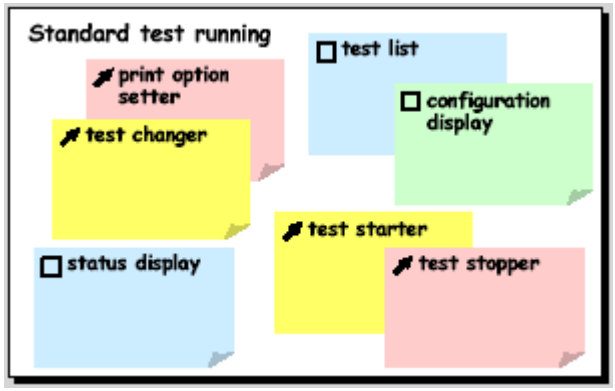


图 1 - 常规（完全）抽象原型示例

其他抽象原型的变体形式包括“线框（wire-frame）”模型和抽象布局图表。如图 2 所示，线框模型描绘可见用户界面元素的相对大小和位置。区域的色彩编码也可以用来说明元素对应的类型，或者信息、功能的相对重要性或优先权。这种变体形式在基于 web 应用的图形设计者中有一定程度的流行。

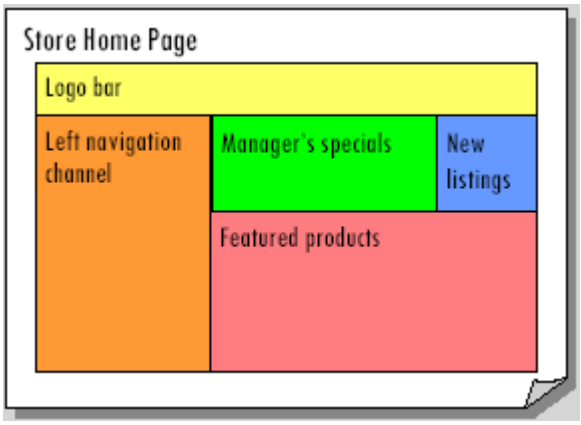


图 2- 线框模型示例

如图 3 所示，抽象布局图表是一种“低保真度”的原型形式。它显示了用户界面元素的相对大小和位置，但没有确切的外观。

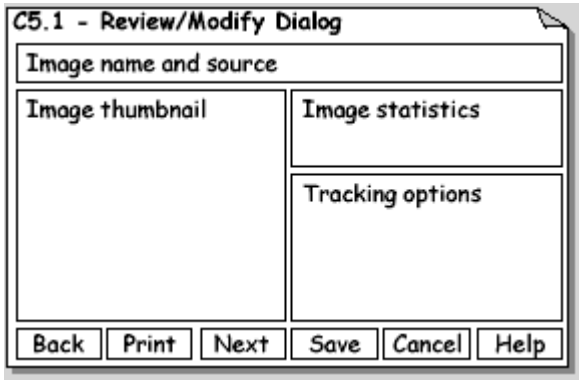


图 3 - 抽象布局图表示例

各式各样形式的纸上（图表）原型可以按照从最抽象到最具体或者说最实际作如下排列：

1. 常规的（完全抽象）内容模型；
2. 线框（wire-frame）模型
3. 抽象布局图表
4. 低保真度纸上模型（粗略的草图）
5. 高保真度纸上模型（真实的细节设计）

抽象时的难题

尽管已经证明抽象原型作为设计工具很有效，但也证实对某些设计者是块绊脚石，尤其那些相对缺乏经验或对以使用为中心的设计还在入门阶段的设计者。最常见的问题包括：

- 用抽象术语命名或描述组件的困难
- 区别工具和原料的困难
- 将抽象组件转化为物理组件的困难
- 通过抽象视图布置屏幕和其他用户界面上下文的困难

在命名组件用以支持任务案例的时候，经常发现缺少经验的设计者在抽象条件下思考是困难的。他们常常竭力去设计近似的不承担义务的抽象名字。应该叫做“Employee Record”，还是“Employee Record Holder”，或是“Employee Description Holder”，抑或别的什么名字？如果没有对术语进行仔细的选择，设计者可能在实现真正的用户界面时，不经意地混淆想象中的假定和组件应该具有的最终形式。比如说，一个名为“Employee Data Grid”的组件，可能意味着特别的用户界面数据控制。

因此，以使用为中心的设计不赞成用过于特殊或一体化的技术名词或行话来做抽象组件的名称。比如，相对于用“Search Criteria Entry Field”，是否用“Sought-Person Description Holder”更值得鼓励。然而不幸的是，即使这种习惯在所有特定执行中都作为义务遵守，如果严格地遵循，会在后期把抽象原型转化为执行模型时带来自身的问题。如果由于抽象的影响，应用领域的精确术语，例如实际域的类或方法的引用在上下文模型中被弃用，那么，模型——尽管如预料中那样直接得自任务模型——也可能和其他设计模型以及项目其它部分已制定的词汇表脱离。

遵循推荐的抽象组件命名协议（比如，“Name Holder”、“Constraint Stuff Getter”，诸如此类），设计者最终能得到的不但和最终物理设计脱离，而且和其他模型脱离的模型。这些脱离的联系最后必须在设计里恢复和重建以完成和建造设计。(比如，在一个项目里，设计组常常必须在设计可视原型前回顾每个视图中哪个域对象相互关联。)

高度抽象组件组成的内容模型也难以和外行或开发组的其他部分共享：基本上，如果开发的时候你不在场，那么它们对你而言没有什么意义。

刚起步的设计者还为这样的问题所折磨：要满足任务模型的某个特定要求，到底应该用主动组件还是用被动组件？也就是说，该用抽象工具还是抽象原料？可进行编辑的显示项是主动工具还是被动原料？争论还在持续！

沟通语义隔阂的桥梁

暂且不考虑上述困难，对于多数设计者，一旦他们有过一些实践，就会发现自任务案例派生的初始内容模型通常较为直接。基本上，所要做的就是一步一步完成任务案例解说，确定用户界面所需的工具和原料来完成每个步骤。如图 4 的例子可以看到，立足于建模的角度，任务模型和内容模型之间的语义隔阂相当小。从各个步骤到抽象工具和原料之间存在着映射，映射即使不是完美的一一对应，也是相对简单的。

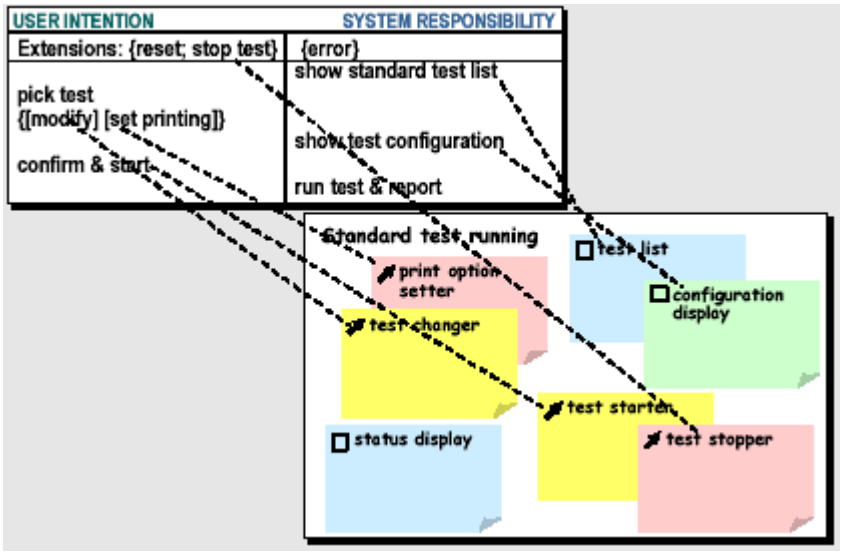


图 4- 从任务案例到内容模型映射示例

相反，充分抽象的内容模型和好的最终设计之间的语义隔阂通常可能非常巨大。实际的纸上模型看起来和内容模型毫无相似之处。在用户界面的抽象模型和实际模型之间存在着许多决策和困难的权衡。必须选择或设计实际的用户界面组件，必须决定屏幕布局，还必须解决其它外观和行为等方面的问题。非常熟练能干的设计者往往有能力较为轻松地跃过这类隔阂，尤其是在对以使用为中心的设计有过实践之后。而入门者和呆板的设计者通常会在这点上被绊倒。其结果可能是混乱不堪。

尽管最初发明抽象内容模型是用来帮助将任务模型转换为实际原型，但经验表明，它太过接近任务模型，又太过远离完成一个可视化交互设计的目标。相对于其收益，最初构思的抽象原型常常被证明太难于开发。

建造更好的桥梁

对一系列项目经验的回顾凸显了抽象原型法的许多缺点，但也让我们确信，任务模型和最终用户界面设计或实现模型之间的模型中介有整体上的优势。对于初学者，比起典型没有使用原型而达成的初始设计，抽象原型看来能通往实质上更好的设计。对于高级的成熟设计者，抽象原型促进创造性思考，导向更为创新的解决方案。

需要一个抽象原型的变体，它位于更接近最终设计的位置，用以充当任务模型和实现模型之间更优秀的翻译者。这样一个模型的修正形式要求：（1）在开始更容易、更自然地开发；（2）更容易转化成实际的可视交互设计；（3）通过域模型更有效和模型的其余部分连接。关于（3），一个显而易见的解决方法是，保证抽象组件的描述和名字使用领域和用户的词汇表，如同以使用为中心的设计中的其它所有模型。

要达到目标（1）和（2），我们认为用一组标准的抽象组件来构造抽象布局图表的形式能最好地构建抽象原型。对于经验较少的设计者，标准的抽象工具和原料能简化并引导抽象原型的构建，还能压缩最终设计的选择。比如，可以有一个设计者能够进行选择的列表，里面是抽象选择器可能的实现。对于更为高级的设计者，一组简单、标准化的抽象组件可以加速和简化建模过程，让设计者专注于细致的问题和创造性的解决方案。标准化抽象组件还应该能够让认可和描述某些模式更容易，这些模式支持特殊的可视、交互设计方案。尽管我们不相信描述抽象问题的标准方法能称心如意地把用户设计过程变成依葫芦画瓢，但它也许可以帮助设计组对某些标准问题作出好的解决方案来。

规范抽象组件

如同图形用户界面提供一套标准的实际组件工具包给设计者选择，规范抽象组件提供了一套标准的抽象组件“工具包”用于抽象原型构建。这套建议的规范抽象组件是经过多次持续求精和反复的尝试性应用之后才作出的。当前版本决不是一个理论最小集或无懈可击的，但我们相信它是实用有效的，它覆盖了在以使用为中心的用户界面设计中出现的所有公共案例。

表 1 概要地说明了这套规范抽象组件。规范抽象组件用名字和简单图表或图示符作标识，后者给高级设计者作为图形速记符号，并帮助抽象原型的可视化识别和判读。（我们敏锐地意识到，要让这样一种速记成为真正的捷径，适当的工具是必须的。）新的符号分别来源于两套已经用于表示工具和原料符号的主题和变种。尽管不是所有的符号第一眼看上去都能显得合乎直觉，但我们已经竭力使其在初次看到时就能充当有效的提示。

如表 1 所概述，建议的规范抽象组件包括：（a）普通或通用抽象组件（b）一套核心的附加基本抽象组件和（c）少数辅助的专用组件，也就是那些即使理论上不需要但实践的观点公认经常是需要的。（可选的组件在表 1 中用灰色高亮突出。）所有原料是一般容器的有效特殊化，而所有工具则是一般操作/动作的特殊化，所以一般组件总是可以用于任何目的。

MATERIALS	DESCRIBED BY	EXAMPLE
container	contents	Configuration holder
element	contents	Current customer ID
collection	contents [set]	Personal address list
notification	message/condition	Overheated actuator alarm
TOOLS		
operation/action	action	Print symbol table
start	[Do/Start] action	Do consistency checking
quit	[action]	Finish inspection session
select	[Select] element	Group member selector
create	[Create] element	New customer
delete	[Delete] element	Break connection line
modify	[Modify] element	Change shipping address
move	[element] [to/from]	Put into address list
duplicate	[Copy] element]	Copy address
accept	[Accept] contents	Accept search terms
go/link	[To] target	To home page

表 1-规范抽象原料和工具概要

原料

有三种基本的抽象原料：

- 容器（普通的）
- 元素
- 集合

加上一种辅助组件

- 通知

抽象原料建议的命名约定是完全使用内容的名字，亦即对象、类、数据元素，或类似的表示。如果阐明了模型的话，添加诸如“持有者”或“容器”之类术语也是可接受的，但不是必需的（例如，“当前机器配置”或“被翻译语言持有者”）。对于集合的约定是，或者使用复数来暗示是多个内容（例如，“特殊符号”），或者描述集合的种类或类型（“地址列表”），以此达到适当和必需的明确性。

无论是在纸上，还是在 CASE 工具里，或预打印表格中，抽象组件可以用图表单独标记，也可以用图表加上跟随用户提供名字的组件类型。例如：

集合：个人地址列表

个人地址列表

事实上，公告是一个消息容器或指示器，应该由表达的消息、条件和事件来命名（例如：“太多条目”或“机器不同步”或“开启”）。

工具

操作和动作是两类截然不同的抽象工具。操作是在原料上进行作业的抽象工具，动作则是导致或触发某些行为的抽象工具。类属于动作/操作，有八种基本抽象工具。

1. 动作：
    1) 启动开始
    2) 终止退出
2. 操作
    1) 选择
    2) 创建
    3) 删除
    4) 修改
    5) 移动
    6) 复制
3. 辅助工具
    1) 接受
    2) 转向连接

对于建模，接受工具可以当作一个主动原料使用，即一个容器从用户处取得输入；也可以当作一个操作容器的工具使用。对于一切实践上的目的，最一般的工具是启动/开始。

命名抽象工具建议的约定仅仅是说明动作。对于一般的动作，前缀“做”或“开始”是可选的（例如，“做符号检查”或“打印符号表”）。对于模型预定目的的结果清晰的地方，可以使用图形符号或只用名字。假设图形符号和动作名称在多数情况下可以互相替代。因此下边是对同一组件的三种不同描述方法：

关闭配置

配置

关闭：配置

在需要清晰的地方，操作原料的工具（操作）应该给原料命名。抽象原型里，操作只能放置在操作的原料上或一起放置在任何这样便于用图表表示和有意义的地方。

这在实践上整个看起来像什么？我们现在相信，对于多数的用户界面设计，最有用的抽象原型形式是抽象布局图表，其抽象组件的大小和相对位置是有意义的。图 5 是一个使用规范组件的抽象布局的例子。它只是特意用作说明，并不是可仿效或值得仿效的。如例子所示，有时候其它组件内的抽象组件嵌套既是必要的又是有利的。

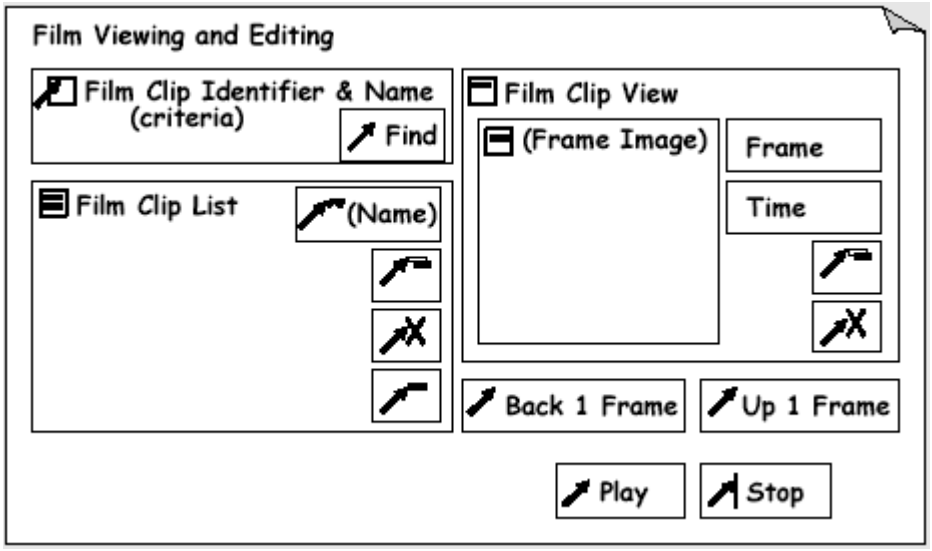


图 5– 使用规范组件的抽象布局示例

正如已经讨论过的，不同的抽象原型形式有不同程度的抽象。另外一种有价值，尤其对基于 Web 工程或非常大型的设计问题很有用的变体是基于文本的非图形形式，它仅仅列出视图和它们的内容。如下例子所示（基于图 5 的简单翻译），不管有没有图形符号，使用抽象规范组件改善了可读性，有助于阐述基于文本的内容模型。（注意抽象组件的嵌套。）

actions:

□ initiate/start

□ terminate/quit

operations

□select

□create

□delete

□modify

□move

□duplicate

auxiliary tools:

□accept

□go/link

源自规范原型的设计过程

无论对于经验丰富的、成熟的设计者，还是经验与才能都一般的设计者，在抽象原型里使用一套规范组件总是提供了潜在的好处。从规范原型到实际原型或最终原型的转化包括了两个并发而且相互依赖的关联动作：可视化设计和交互设计。

可视化设计包括：

（a） 可视化组件的选择或设计，以实现每个规范组件能够结合

（b） 界面上这些可视化组件在视图或上下文里的布局。

交互设计包括：

（a） 交互习惯用语的选择或设计

（b） 描述界面和底层系统必要的行为，

（c） 组织视图或上下文之间及内部的交互工作流或次序。

对于常规设计，每个规范组件只需要用一个或多个标准用户界面窗口小部件实现。要达到最佳效果，设计者应该对每个抽象组件在各式各样可选实现做出决定。构造纸上原型的实验布局是基于这些方案的初始选择。典型地，实际用户界面组件的选择蕴含着许多交互设计，而布局则决定了工作流。为有效支持任务案例，应该相对于被支持的精炼任务案例，和组件选择与布局一起，检查作为结果的交互设计。

在以高性能或突破性设计为目标的地方，规范模型为创造性设计提供额外指导。使用规范组件使得有经验的设计者更易于辨认和区分模式或隐含了某种问题或解决方案的公共状况，。比如，对以往设计工作的回顾表明，确定配置的成熟时机往往是发现新的用户界面控制——既能高效使用又能更好利用屏幕真实状态。例如在高级设计组的手里，容器、集合或内含（嵌套）工具元素之间的嵌套结合常常可以变成有效的非标准用户界面控制。

概括地说，基于规范原型的创新设计的过程是这样进行的：首先，记录联系紧密或成组的抽象组件，尤其是嵌套的联合。对于每一个这样的组或联合：

1. 识别常规/例程实现和创新的部分；
2. 选择有希望的联合；
3. 合成并精炼。

应用实例

作为一个简化的例子，考虑图 6 显示的抽象原型部分。其中，条目的集合可以被用户任意按新的顺序重排。许多可视设计的近似途径是可能的，除了其他的以外，还包括：

- 一个可编辑的顺序数字列表
- 一个列表，有 up 和 down 按钮，可以在表内移动选中的项
- 一个临时储存列表，允许在主列表内移去和重新插入项

每一个近似途径中包括了略微不同的可视化组件和布局发布。

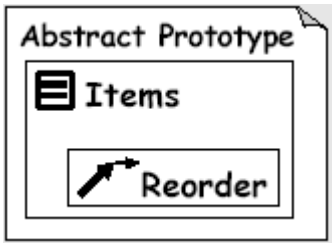


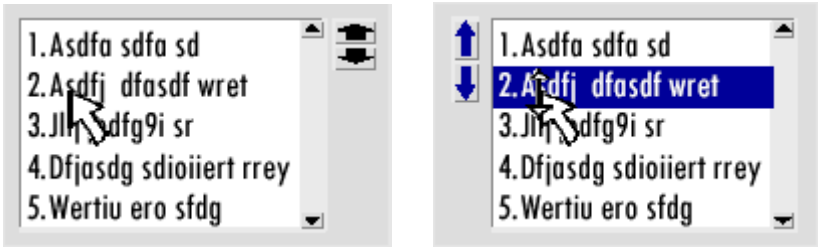
图 6– 工具在容器内嵌套的示例

交互设计包括识别可能的界面行为以及解决这个问题的交互习惯用语。包括：

- 点击选择源点和目标
- 拖放
- 编辑顺序数字
- 点击移上或移下按钮

一个有前途的、同时支持新手和更高级使用模式的联合应该既支持在列表内用上-下按钮移动，又支持用拖放移动。初始设计可能类似于图 7（a），它凸显了几个问题。如果将上-下按钮和滚动条按钮放在他们通常的位置上，很容易造成混淆。如果只是简单地移到左边，则它们可能容易被忽略，并且它们的功能可能不清晰。

这样的问题可以用适当的可视交互设计来解决。如图 7（b）所示，通过把上-下按钮移到左边，改变它们的形状，以及用彩色图示符突出它们，可以增强它们的独立性。如果一开始屏蔽（变灰）了上-下按钮，然后，当选了列表中的条目时，激活并用彩色高亮，那就能通过这种逐步启动的方式使界面带有启发性。可以在列表内拖放条目是一个值得采用的功能，尤其对高级用户而言，但这种功能是一种没有给用户合适反馈的隐式行为。只要列表中的条目被选中，在鼠标向下时把光标变成上-下移动的形式，可以移动的信息就能够传达给用户。



（a） （b）

图 7– 加工问题可能的解决方案

结论

使用规范抽象组件的抽象布局图表提供了一个令人兴奋的新工具，它可以平滑和加速以使用为中心的设计过程。如本文所描述的，这些“规范原型”。

- 由特殊的抽象组件构造，它们从用标准符号描述的一小套标准组件中选择
- 显示布局，包括相对大小，位置和组件的嵌套或覆盖
- 和整个项目中其它所有模型使用相同的用户与应用领域标准词汇表

因此，抽象原型能简单而直接地从任务案例构造，并且比先前的抽象原型形式更接近最终设计。规范原型允许设计者轻易地对特殊的用户界面内容建模，以及试验大概的布局而无需提交外观或图形设计的细节。设计者拥有一整套标准但抽象的组件，并从中进行选择，用来表达用户界面设计的内容和大概布局。由于作为结果的模型在忽略细节的时候更为接近实际用户界面，规范原型促进了最终设计，而不至于失去创造性或非标准实现的可能性。

未来的研究可以进一步提高规范原型的价值。对于任何特殊的执行环境，可以预先对每一个不同的规范组件把不同的有用实现编成目录。特别对于新手，这样的指导相当有用。对于更高级的设计者，可以根据规范组件的联合和构造来组织和描述用户界面设计模式。

✧ 参考书目

1. Constantine, L. L. (1998) “Rapid Abstract Prototyping,” Software Development, 6, (11), November.
2. Constantine, L. L. (2000) “Inventing Software,” Software Development, 8, (5), May.
3. Constantine, L. L., & Lockwood, L. A. D. (1999) **Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design.** Boston: Addison-Wesley.

✧ 术语表

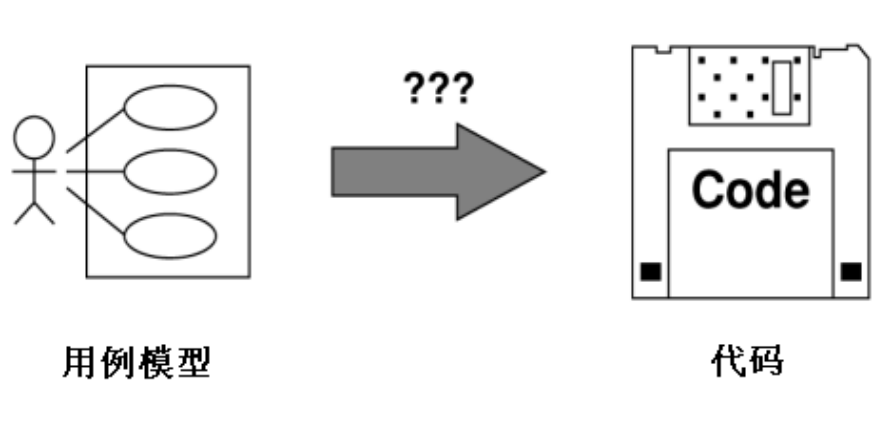
抽象布局	用户界面视图的低保真度原型，展示了布局，包括组件的相对大小和位置，但没有外观的精确细节
抽象原料	抽象用户组件，代表容器、信息或数据
抽象原型	一套模型里的任何一个，抽象地代表用户界面设计
抽象工具	抽象用户组件，操作原料或启动某些动作
规范组件	抽象工具和原料的标准集中的一个
规范原型	一个抽象原型，显示根据用户和应用域的词汇表所描述的规范组件之布局、大小和位置
内容模型	一个抽象原型，只表示视图里的抽象工具和原料，而与它们的外观、行为或布局无关
任务案例	本质用例[Constantine & Lockwood, 1999]，支持一个或多个用户角色
视图	交互上下文，用户界面的一部分，在其中用户可以和系统交互，例如，屏幕、对话框、窗口，或其它类似的部件

✧ 注释

1 为了表达上的简单和经济，以及和其他模型与符号的兼容性，以使用为中心的设计的一些术语还在修订过程中。见术语表。

UMLChina 培训

The real thing

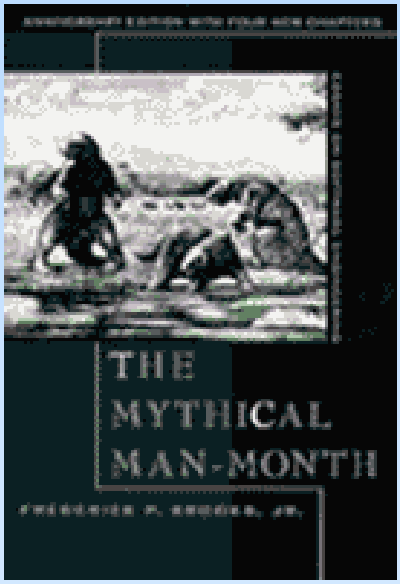


利用 UML 的 20%就可以为 80%的问题建模

--《UML 用户指南》，第 32 章

详情请垂询：think@umlchina.com

《人月神话》



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 Adams Wang

散文笔法，绝无说教，大量经验融入其中

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

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

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

中文译本即将发行！

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

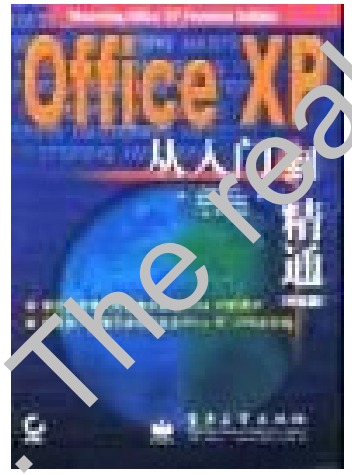
翻译：UMLChina 翻译组乐林峰

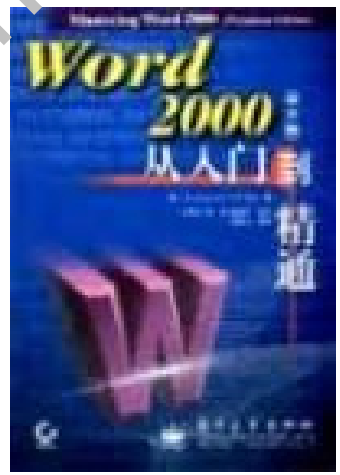
Cockburn 一向通俗，本书包括十几个项目的案例

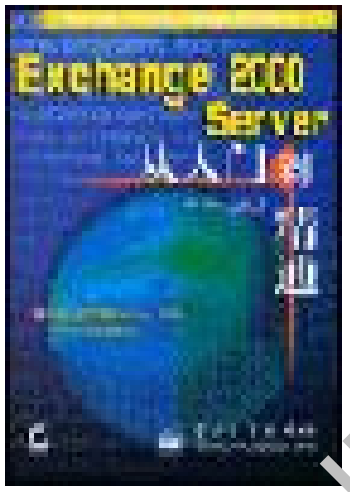
面向对象技术在给我们带来好处的同时，也会增加成本，其中很大一部分是培训费用。经验表明，一个不熟悉 OO 编程的新手需要 3 个月的培训才能胜任开发工作，也就是说他拿一年的薪水，却只能工作 9 个月。这对一个拥有成百上千个这样的程序员的公司来说，费用是相当可观的。一些公司的主管们可能一看到这么高的成本立刻就会说"不能接受。"由于只看到成本而没有看到收益，他们会一直等待下去，直到面向对象技术过时。这本书不是为他们写的，即使他们读了这本书也会说（其实也有道理）"我早就告诉过你，采用 OO 技术需要付出昂贵的代价以及面临很多的危险。"另外一些人可能会决定启动一个采用 OO 技术示范项目，并观察最终结果。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

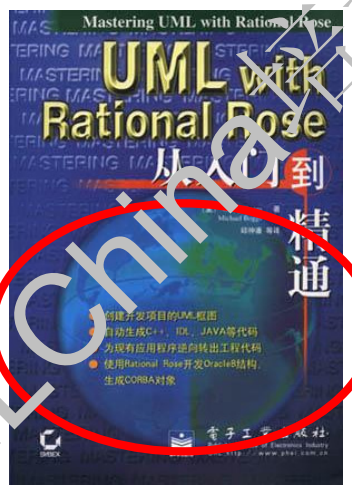
中文译本即将发行！

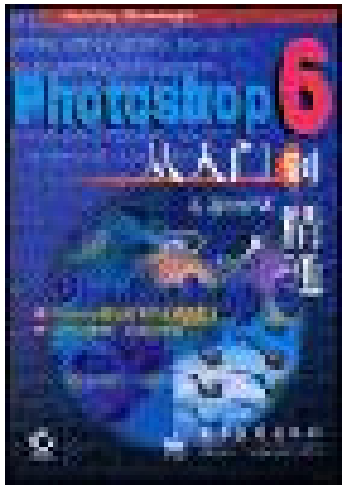
这就是面向对象？

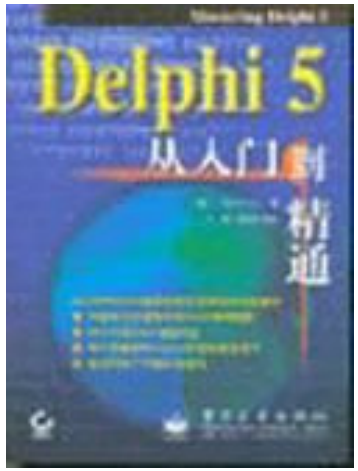


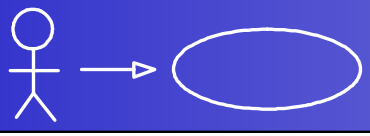












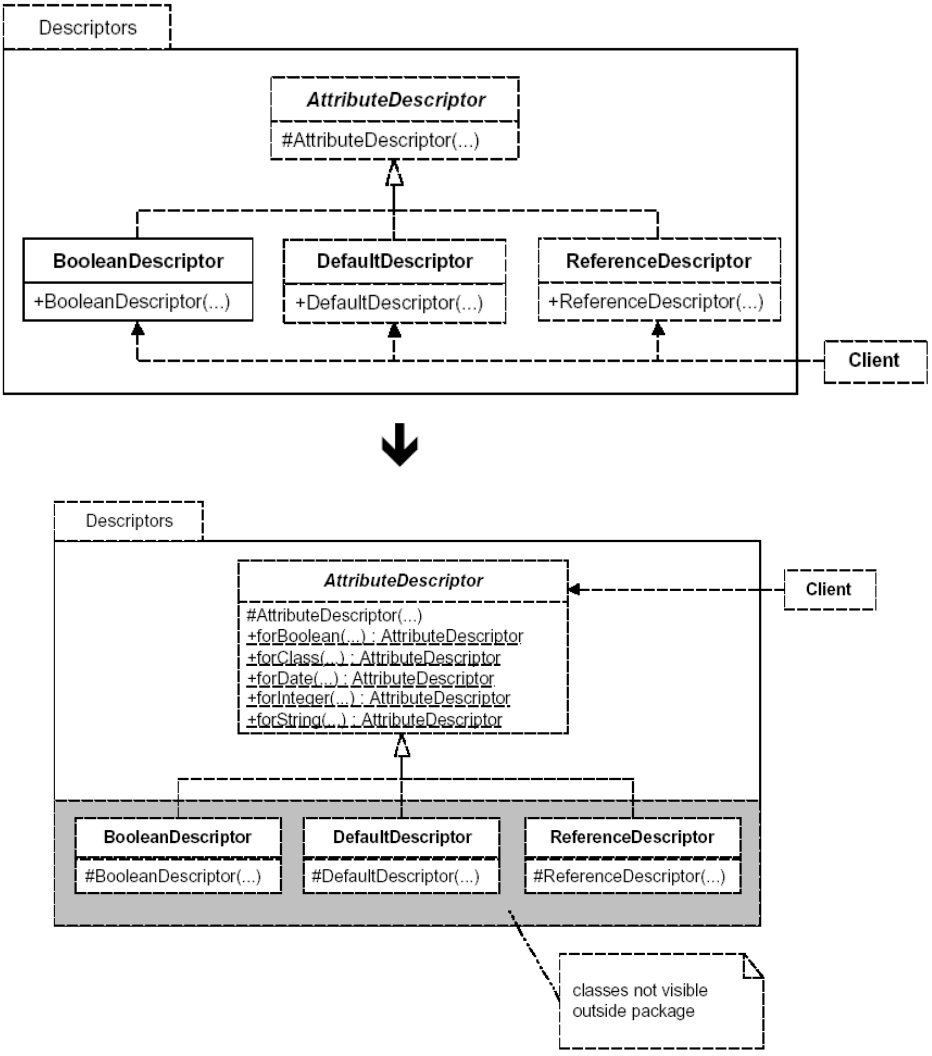
用创建方法封装子类

Gigix 译文专栏

John Vlissides 著，透明 译

不同的类隐藏在包的内部，实现共同的接口，但是客户却直接实例化这些类

将类的构造子设为非公开，并让客户通过超类的创建方法来得到它们的实例



动机

如果客户（client）需要知道每个具体类的存在，那么让客户直接控制这些类的实例化也是个不错的选择。但是，如果客户不想知道这些，又该怎么办呢？如果这些具体类都被放在一个包的内部，并且都实现了同一个接口，而这个接口又不太可能发生变化，那么就应该把这些具体类隐藏起来，让包外部的客户去使用超类公开的创建方法（Creation Method），并通过创建方法得到满足需要的实例。

这样做的动机有几点。首先，可以确保客户只能通过通用的接口来访问不同的类，以确保“分离接口与实现”[GoF]；其次，可以将不必为外界所知的类隐藏起来，从而减少一个包的“概念重量”[Bloch]；第三，由于对象的创建都通过创建方法来进行，而创建方法的名称可以更好地揭示创建过程的意图，所以可以使实例的创建过程更容易为开发者所理解。

尽管有所有这些好处，还是有人对这个重构持保留态度。对于他们的疑惑，我做出了如下回应：

1、他们不喜欢让超类知道子类的信息，因为这会导致循环依赖——如果你创建了一个新的子类或者对子类的构造子做了增改，就不得不在超类中添加新的创建方法。不过我会告诉他们：这个重构发生的场景是一个包，其中的子类都实现同一个接口。这时他们就会闭嘴了。

2、他们不喜欢在超类中把创建方法和其他实现方法混在一起。我并不担心这个问题，除非创建方法让我难以看清超类的行为——如果真是这样，我就会使用“提取创建类”（Extract Creation Class）的重构。

3、在源代码编译成目标代码之后，他们就不愿意再做这个重构，因为使用目标代码的程序员是不能再添加或修改非公开的类和创建方法的。对此我更加同情。如果包内部的可扩展性的确很重要，而用户又得不到源代码，我就不会把类都封装起来，而是会提供一个创建类（Creation Class）来生成实例。

本文开始处的那幅图简单画出了关系数据库映射过程中的一些对象关系。在采用这个重构之前，程序员们（包括我自己）有时会选错了要实例化的子类，或者用错了参数（比如说，我们可能会调用一个接受 Java 内建的 int 类型作为参数的构造子，而真正需要的却是接受 Integer 对象作为参数的那个构造子）。这个重构会把关于子类的信息封装起来，客户只能从一个意义明确的地方得到子类实例，从而减少了创建错误的机会。

沟通	重复	简化
如果你希望客户代码只通过一个接口与你的类沟通，那么你就必须用代码来反映出自己的要求。公有的构造子没有任何帮助，因为客户通过公有构造子就可以跟子类型耦合在一起。要达到你的要求，就需要把构造子隐藏起来，然后通过超类中的创建方法来生成对象，并且将创建方法的返回类型定为所有子类共同的接口或抽象类。	使用这个重构的时候，重复不是问题。	如果你想让客户只通过一个接口与所有的子类打交道，那么把这些类公开只会把事情搞得更复杂：程序员们会直接实例化子类，并把自己的代码和子类型耦合在一起。这种做法就好象在说：去扩展这些类的接口吧，没有关系。 如果不允许直接实例化这些子类，只通过超类的创建方法提供实例，那么情况就简单多了。

约束

- 你的所有类应该有共同的公有接口。

这是根本的条件，因为在此重构完成之后，所有的客户代码都只能通过这个共同的接口来访问所有这些类的实例。

- 你的所有类应该属于同一个包。

过程1

在超类中为每种实例（一个构造子生成的实例称为“一种”）编写创建方法，创建方法的名字应该能清楚地说明自己的意图。创建方法的返回类型应该是所有可创建对象共有的接口类型。让创建方法去调用相应的构造子。

1. 选定一种实例，将所有对应于这种实例的构造子替换成超类中相应的创建方法。
2. 编译、测试。
3. 重复步骤 1~3，直到一个类中的每种实例都通过创建方法来创建。
4. 将这个类的构造子声明为非公有（例如 protected 或者 default）。
5. 编译。

重复上面的步骤，直到所有的构造子都变成非公有、所有的实例都可以并且只能通过创建方法来获取。

范例

1、我们从一个比较小的类体系开始，这个类体系被放在descriptors包里面。这些类在对象-关系数据库的映射中起辅助作用，可以把数据库属性转换成实例变量。

```
package descripors;
public abstract class AttributeDescriptor {
protected AttributeDescriptor(…)
public class BooleanDescriptor extends AttributeDescriptor {
public BooleanDescriptor(…) {
super(…);
}
public class DefaultDescriptor extends AttributeDescriptor {
public DefaultDescriptor(…) {
super(…);
}
public class ReferenceDescriptor extends AttributeDescriptor {
public ReferenceDescriptor(…) {
super(…);
}
```

抽象类 AttributeDescriptor 的构造子是 protected 的，三个子类的构造子则是 public 的。由于三个子类情况相似，所以我们只需注意 DefaultDescriptor 就可以了。首先，我们需要识别出 DefaultDescriptor 的构造子创建的那一种实例，所以请看下面的客户代码：

```
protected List createAttributeDescriptors() {
Vector result = new Vector();
result.add(new DefaultDescriptor("remoteId", getClass(), Integer.TYPE));
result.add(new DefaultDescriptor("createdDate", getClass(), Date.class));
result.add(new DefaultDescriptor("lastChangedDate", getClass(), Date.class));
result.add(new ReferenceDescriptor("createdBy", getClass(),
          User.class,RemoteUser.class));
result.add(new ReferenceDescriptor("lastChangedBy", getClass(),
          User.class,RemoteUser.class));
result.add(new DefaultDescriptor("optimisticLockVersion",
          getClass(), Integer.TYPE));
return result;
}
```

[1] mechanics 一词，以前我译为“技巧”。但是看完《Refactoring》之后，愈加觉得“技巧”这个译法不好。《英汉能源大词典》中对 mechanics 一词的解释是：“n.力学,机械,例行手续”，我采用“例行手续”之意，译为“过程”。请读者指正。——译者

我看出来了：DefaultDescriptor被用来表现Integer和Date之间的映射。它还可以用来映射其他类型，但是此刻我只能注意一种实例。所以，我先编写一个创建方法来为Integer对象提供属性描述：

```
public abstract class AttributeDescriptor {
public static AttributeDescriptor forInteger(...) {
return new DefaultDescriptor(...);
}
```

我把创建方法的返回类型规定为AttributeDescriptor，因为我希望让客户通过AttributeDescriptor这个接口与其子类进行交流，从而使descriptors包之外的客户无需知道这些子类的存在。

如果你有“测试优先（test-first）”的编程习惯，那么在开始这个重构之前，你应该首先编写一段测试代码，从超类的创建方法中获取AttributeDescriptor的子类实例，并判断所得实例的类型是否正确。

2、现在，客户如果想生成Integer版本的DefaultDescriptor，就必须调用超类中的创建方法：

```
protected List createAttributeDescriptors() {
List result = new ArrayList();
result.add(AttributeDescriptor.forInteger("remoteId", getClass()));
result.add(new DefaultDescriptor("createdDate", getClass(), Date.class));
result.add(new DefaultDescriptor("lastChangedDate", getClass(), Date.class));
result.add(new ReferenceDescriptor("createdBy", getClass(), User.class,
RemoteUser.class));
result.add(new ReferenceDescriptor("lastChangedBy", getClass(),
 User.class,RemoteUser.class));
result.add(AttributeDescriptor.forInteger("optimisticLockVersion", getClass()));
return result;
}
```

3、编译、测试，确保新的代码运转正常。

4、现在，我继续为 DefaultDescriptor 的构造子能创建的其他种类的实例编写创建方法。我又得到了另外的两个创建方法：

```
public abstract class AttributeDescriptor {
public static AttributeDescriptor forInteger(...) {
return new DefaultDescriptor(...);
}
public static AttributeDescriptor forDate(...) {
return new DefaultDescriptor(...);
}
```

```
public static AttributeDescriptor forString(...) {
return new DefaultDescriptor(...);
}
```

5、现在，将DefaultDescriptor的构造子声明为protected：

```
public class DefaultDescriptor extends AttributeDescriptor {
protected DefaultDescriptor(…) {
super(…);
}
    }
```

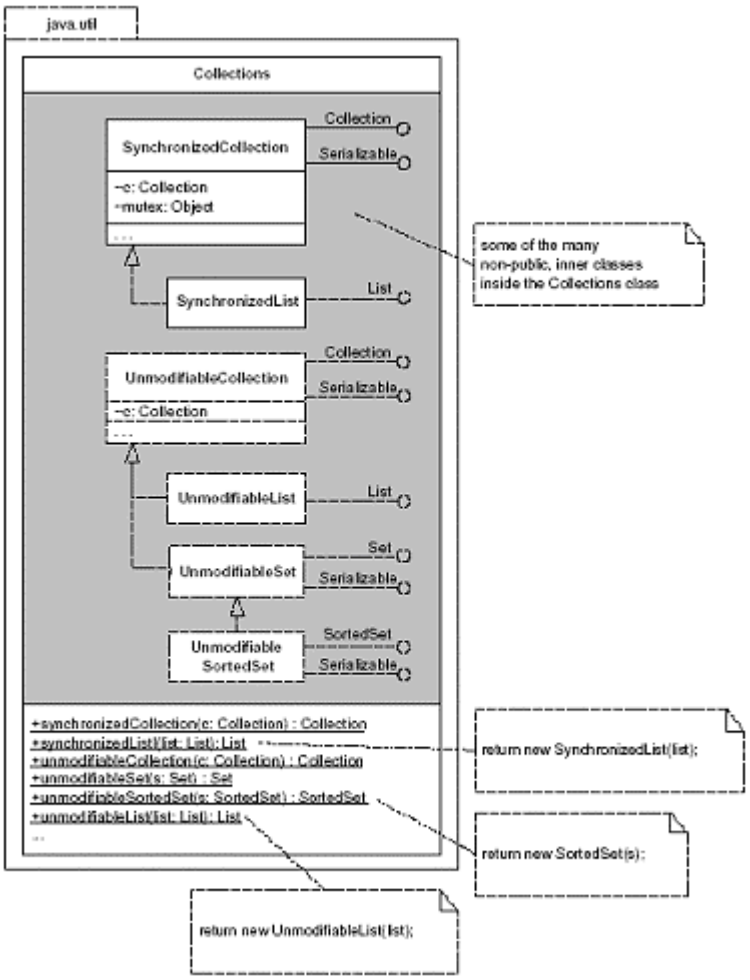
6、编译一下，一切尽在掌握。

7、现在，针对AttributeDescriptor的每个子类，重复上面的步骤。完成以后，新的代码应该

- 通过AttributeDescriptor提供对其子类的访问
- 保证客户只能从AttributeDescriptor这个接口获取子类的实例
- 不允许客户直接实例化AttributeDescriptor的任何子类
- 让其他的程序员明白：AttributeDescriptor的子类是不公开的——应该通过超类和通用接口来访问它们。

封装内嵌类

JDK 中的 java.util.Collections 类是“用创建方法封装子类”的一个典型范例。这个类的作者 Joshua Bloch 需要为程序员们提供一种办法来保证 Collection、List、Set 和 Map 的不可修改性和（或）同步性。一开始，他很聪明地用 Decorator 模式来实现。但是，他没有创建公开的 java.util.Decorator 类并要求程序员们用它来装饰自己的 Collections 子类。他的做法是：将 Decorator 定义为 Collections 类的非公开内嵌类（Inner Class），并在 Collections 类中设计了一组创建方法，程序员可以通过这些创建方法得到自己需要的、经过装饰的容器。下面是 Collections 类中的内嵌类和创建方法的设计草图：

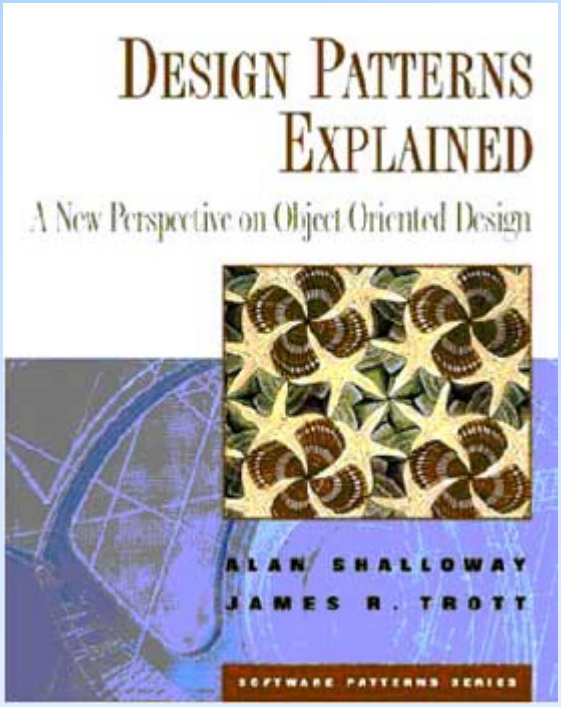


请注意，java.util.Collections 类甚至还包含了一个小小的内嵌类继承体系，其中所有的类都是非公开的。每个内嵌类都有一个相应的创建方法，这个方法接受一个容器，在其上进行装饰，并用预先定义的常用接口类型（例如 List 或者 Set）返回装饰后的实例。通过这个方案，程序员无须再去了解那么多的类，并且需要的功能也一点不少。同时，java.util.Collections 类也是一个创建类（Creation Class）的典型范例（参见“提取创建类”）。

看不懂《设计模式》？

它的作者推荐《设计模式精解》

透明 译

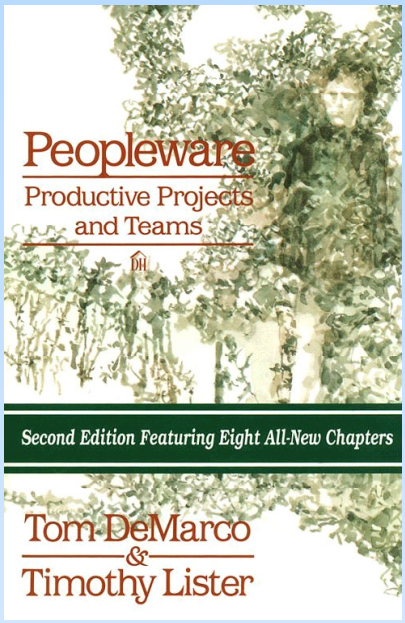


直接学习《设计模式》这本书是非常困难的。作为入门，我推荐 **Alan Shalloway** 的《设计模式精解》。

--John Vlissides 写于 UMLChina 答疑板

中译本即将上市！

《人件》



《人件》第 2 版

Tom Demarco 和 Tim Lister

翻译：UMLChina 翻译组方春旭、叶向群

微软的经理们很可能都读过—amazon.com

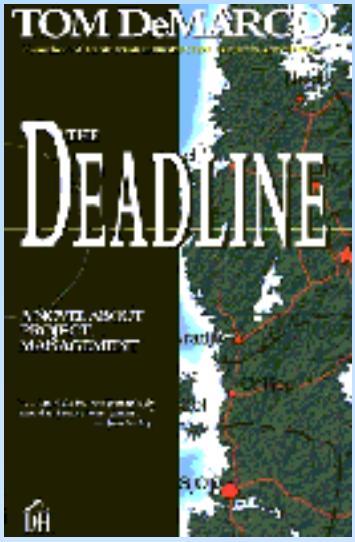
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

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

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

中文译本即将发行！

《最后期限》



《最后期限》

Tom Demarco

翻译：UMLChina 翻译组 透明

这是一本软件开发小说

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

....

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

莱克莎点点头："这是最近的数字。他们都会在你的手下工作。"

"而且据你所说，他们都很优秀。"

"他们都通过了摩罗维亚软件工程学院的 CMM 2 级以上的认证。"

中文译本即将发行！