2002－6（第14期）

【访谈】

1 Alistair Cockburn：选择你所需要的

【方法】

11 界面耻辱纪念堂--隐喻使用不当

20 界面耻辱纪念堂--可视元素

27 设计模式之路(一)

35 提取创建类

42 利用重构改进面向对象设计

76 一个产品的订货和配送（系统）的分析模式

Alistair Cockburn

X-Programmer 非程序员

软 件 以 用 为 本

主编：davidqql

审稿：davidqql，think

投稿：editor@umlchina.com

反馈：think@umlchina.com

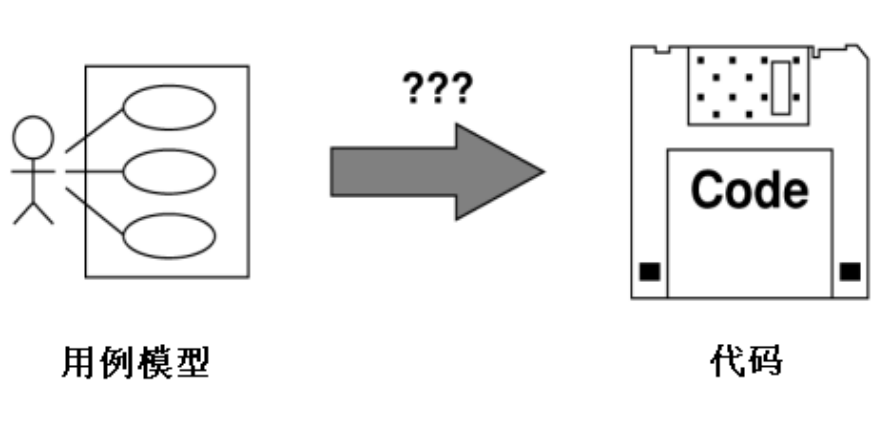
http://www.umlchina.com/

本杂志仅供学习和交流之用

转载需注明出处，不得用于商业用途

UMLCHINA 培训

北京公开课：6/29-6/30



详情请垂询：UMLCHINA 北京培训中心

联系人：傅光远

电话：66130237，13910698898

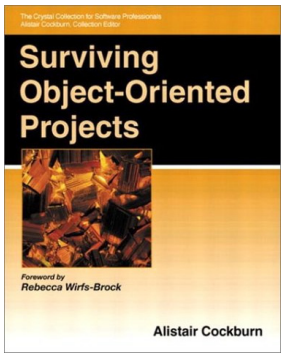
EMAIL：bjcourse@umlchina.com

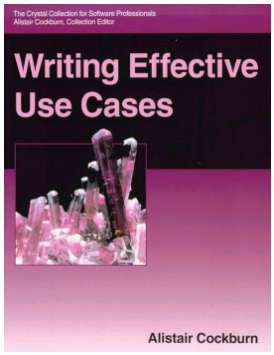
Alistair Cockburn： 选择你所需要的

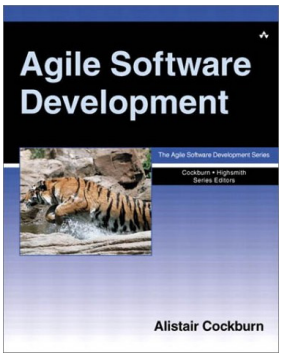
查看评论

编注：

北京时间 2001 年 9 月 1 日上午 10:00-12:00，Alistair Cockburn 先生作客 UMLCHINA 讨论组的聊天室。Alistair Cockburn 是用例、面向对象、项目管理领域的世界顶级专家，最优秀的“传道者”。他的著作“Writing Effective Use Cases”曾获全球软件业最杰出的技术与书籍的奖项 Jolt Productivity Award。他的“Surviving Object-Oriented Projects”已经由 UMLChina 翻译组进行翻译，中文版本《面向对象项目求生法则》即将由清华大学出版社出版。以下是交流实录。由谭红霞翻译。原文链接。







se_cn：你能告诉我用例的目标是什么吗？

acockburn：用例的目的是为了描述系统的行为及与它相关的系统之间的交互，用例不能很好地表示数据内容，甚至根本就不能表示用户界面。

se_cn：用例等同于功能分解吗？

acockburn：…如何说好呢，简单的说，系统应该做什么……及当系统发生错误时该如何做。

evpu：能谈谈如何测试 J2EE 系统吗？

acockburn：很不幸，我对 J2EE 系统不太熟，还不能给出有关测试问题，很抱歉。,对于这个问题，我只能根据同事的评论。到目前为止我还没有访问过使用 J2EE 的公司，因此我不知道最新动态。

evpu：能介绍一下今天上午的主题吗？

acockburn：我想我能谈谈软件开发过程，需求分析，用例和面向对象设计，项目管理，增量开发和迭代开发，Smalltalk及客户/服务器设计，一些 web/java/3 层结构设计……就是这些，哦，实体关系图，测试设计，但是我对具体的开发工具和开发包不熟，因为它们变动得太快了，我跟不上。

acockburn：主题：用例：用例用文本描述系统该为人们做什么。

umlchina：使用用例时最常犯的错误是什么？

acockburn：最常犯的错误是太细化了。有时，人们书写非常非常细的用例，读起来都很烦。用例是一种叙述，我从未写过一个用例超过九步。如果你超过九步，那么要么是出现了界面设计，要么是步骤太详细了。我发现几乎我参观过的每一个公司都有冗长、详细、枯燥的用例。有人把用户界面（UI）设计放进了用例（我用 UC 代表用例）。这是错误的。UI 设计者应该阅读 UC，然后根据它设计 UI。如果你把 UI 设计放进 UC，那么用例不是需求分析而只是对屏幕的一种描述了。

hdw1978：但是有时候，用户要求“漂亮的界面”。

acockburn：那么，“漂亮的界面”——问问你自己--那是谁的工作？是从事系统行为需求的人的工作吗？或者说那是阅读系统必须“做”什么的简短描述的工作吗？谁来设计漂亮的界面呢？问题在于，界面设计是“设计”而用例是针对“需求”的。不要把设计放进需求里，使需求只包含必须的东西。一个用例就象一个功能模型，但是*是站在用户的角度所看到的*。问：什么是用户真正力求*达到*的？

hdw1978：那么，你是说 UC 某种程度上说是功能模型？你不觉得用户界面设计依赖于需求吗？

acockburn：是的，让我们在这个问题上停留一小会，这是个很好的问题。用户界面设计必须顺应需求。我想声明的就是需求和设计之间的区别。需求就是系统必须有的。设计就是设计者选定的。问题是由谁来做决定。写一个用例是为了表明用户使用该系统可以完成某些与他们相关的相对重要的事情。所以，UC 必须是一个动词短语，是说某人想要完成什么。

hdw1978：使用 UC 构造“黑箱”。

acockburn：(是的，做一个有黑箱的系统，决不要提到系统内部结构)

nix00000：如果你有一个项目要做，你如何写用例？

acockburn：好问题。让我们往回看…找到你的问题的答案的方法是问这个：由谁来做决定？把 UML，Rose 和所有图表搁开一会…如果你不知道什么是你要明确的，他们帮不了你)。步骤一：你问用户和投资方，他们想要系统实现什么样功能？步骤二：他们告诉你。这些就是需求。步骤三：但是他们并不是真的清楚他们所被询问的问题的含义。因此你构造一些原型，或者画出图纸，帮助他们理解他们有什么样的选择权。…重复步骤二-三，你和用户/投资方多方协商决定系统应该实现什么功能。步骤四：对需求的某些部分，你和他们就系统应该实现什么功能最终达成一致。现在两种类型的设计者准备着手工作了--用户界面设计者和程序设计者。步骤 4A：界面设计者设计出易于学习和快速使用，用户能接受和使用的用户界面。步骤 4B：程序设计者开始从事程序内部结构的设计。

nix00000：你说的和书上讲的一样。但是，你能给我们举一个详细的例子吗？

acockburn：（如果我说的和书上讲的一样，那么你会知道在书有很多例子，我请你去读这些例子吧。）

ucolic：那么我应该使界面被用户认可吗？或者只是需求？

acockburn：是的，做原型就是做用户界面设计，和用户一起工作没什么问题。或者把这些给用户看，了解他们喜欢什么不喜欢什么。他们最终会接受或拒绝这个设计，因为他们将要使用这个设计。我的一些朋友说“可接受的测试”是最终的需求！

bucolic：可接受测试，意味着我已经完成了需求，然后客户来做可接受测试。

acockburn：回到用例一会儿…他们记录了系统必须承担的独立于用户界面的描述，因此用户界面设计可以改变，甚至当系统必须承担的描述不变时。把用例当做一个好的记录设备来考虑。当出现问题时（你和用户来决定答案之后）你在用例里书写系统必须完成的功能。我喜欢并行地做每一件事。首先，充分理解需求，什么才是用户的目标，然后做些纸上的原型或程序原型，给他们和自己一些反馈，就他们刚才所问的问题，来判断是不是他们想要的。那样，我们已经做了一些需求，一些用户界面和一些程序设计。只要用户高兴，用户界面设计者和程序员都还有很多工作要做，因此，用户能和另一部分人进行另一部分系统的细化。一旦程序设计者想要让设计运转，详细设计就开始了。

bucolic：如果我得到了正确的原型需求，我怎样处理这些原型？放弃它？还是在下一个设计中重新使用它？

acockburn：你可以把原型丢到一边---如果他们没有用。或者保留他们，如果他们有用的话。

qingzuozhou：完成用例的标准是什么？

acockburn：一个用例可以用两种方式中的一种完成，取决于小项目。对于小的本地的项目，无论它多么杂乱，只要每个人都理解了所有用例表达的东西，用例都能完成。大的复杂的项目（军用和金融系统），用例完成于：A）用例名字列表列出了所有用户要求的目标；B）每一个用例捕获所有操作过程中发生的错误。你需要和用户对系统要求一样多的用例。完全一样多。那是多少？嗯，不同的系统是不同的。20 至 70 就行。我有一个系统用了 200。我知道有人在有些系统中用了 1000。

fuzhong：当你考虑面向对象时，你不考虑程序设计语言吗？如果有人对程序语言不熟悉，他能进行面向对象设计吗？

acockburn：主题：面向对象设计：没有面向对象程序设计语言的知识是很难进行面向对象设计的。我主要用 Smalltalk 编程，C++也用得比较多，JAVA 用得不太多，CLOS 从来都没用过。我能设计一个很好的 Smalltalk 设计，一个中等的 C++设计，一个普普通通的 JAVA 设计，一个糟糕的 CLOS 设计。就像用不同的材料建造建筑物，如果你知道如何利用砖或混凝土和钢筋。用不同的材料设计的建筑物的形状是非常不同的。

fuzhong：用不同的材料设计的建筑物的外形是非常不同的？材料是什么？

acockburn：建筑物…看看你的城市。有些建筑物是砖的，有些是木头的，有些是混凝土，钢筋和玻璃的。他们的外形非常不一样。实际上你建造的建筑物所具有的最可能的外形很大程度上取决于你选用的材料。同样的，雕塑家使用不同的材料：泥土，石头，其它材料。当有人从一块大理石开始构造，他们必须很仔细地处理重量和厚度。以便块与块之间紧密连接在一起。当有人打造一块有齿的金属，然后粘些泥土在它上面，它们可以迅速改变，创造许多完全不同的形状。好的，回到软件上来。C++是一种易碎的材料，像石头。类层次的设计必须小心，你不能轻易地改变它。C++有特殊的继承规则，结果是这种特殊的设计贯穿在整个 C++程序中。另一方面，Smalltalk 像泥土--能轻易改变。用 Smalltalk 编程的人始终可以改变他们的设计。关于继承层次规则很少，因此结构是不同的。JAVA 是 Smalltalk 和 C++的结合，因此设计规则也介于两者之间。CLOS 在规则上更接近 Smalltalk，等等...对于你的问题，答案是否太长了？

qingzuozhou：初始设计完成后，程序员开始编码。但是程序员发现设计中存在许多不合理的地方。这该怎么办？改变我的设计还是让程序员改变他的想法？

acockburn：很好的问题。所有的设计都存在着错误。如果你们的项目团队由一个设计主管来做所有的决定，而程序员仅仅是编程，那么…程序员会发现设计错误，但是不能修补它们…这样的话，我更倾向于每一个人对他自己的设计和程序负责，那么，当程序员发现设计错误时，他（她）简单地修复它就可以了。

qingzuozhou：那么一个人有两个或更多的角色。

acockburn：是的，这意味着设计者应该做些程序。我从来没有在一个有不编程的纯“设计人员”的项目上工作过，我听说过这种项目，但是没有听到关于这种项目的好消息。只是有太多奇怪的错误发生，使得程序员花费太多时间向设计者解释。对高级设计者/程序员，给他们系统设计技术中最难的部分，并让他们检查初级设计者/程序员的工作。然后给初级设计者/程序员系统中规模较小的，简单的那部分设计和程序。

qingzuozhou：当加入项目时，我经常对一些诸如 CMM/ISO9001 之类的标准感到迷惑。

acockburn：是的，我一点都不赞成 CMM(UML 的使用也是如此…：-）这有相当多的话题。我认为编程是一门技能和艺术，就如缝纫，建筑，画画，木工…我所访问到的成功团队是建立在个人技能及其协作之上的。如果你没有团队管理技能，别指望 CMM 过程或 UML 建模能挽救项目。另一方面，如果人们确实有技能，那么他们只需要非常少的 CMM 过程和 UML 建模。过程和建模都有用…但是小批量最好。因此，我不反对 UML 存在，但是我反对项目经理以为用 UML 建模技术能代替程序设计。

ee96pyg：请介绍你认为成功团队所具有的一些特性。

acockburn：好，新主题：成功团队的特性。#1.如果你没有曾经在项目中取得成功的人，你更难在当前项目中取得成功。曾经成功的人记得他们上次怎样解决碰到的问题。因此这是团队中最重要的一个人。对于小项目，这个人通常是团队的领导，从事设计中最难的部分。注意别人的工作质量，和用户协商…规划时间表…别睡得太多。如果你有两个这样的人，那么负担就不那么可怕了。否则的话这个人每周得工作 60 至 90 小时。如果你只有一个这样的人，那么给他（她）一些中等或初级的人去帮他（她）。注意开发技能。如果你有一个没有经验的聪明人，这个人会学得很快。成为一个有中级经验的人，这会很有好处。如果你不那么走运，没有具备经验的人。那么最聪明学得最快的人将很快成长从而成为技术领导。你必须发展一个这样的人。问题：如何减少团队领导的负担？…答案，分配尽可能多的其他事情给其他人。找一个项目经理监督时间表，协调人员间的职责、财物、设备。分配部分工作给正在学习相关知识的人…即使他们慢三倍，有时项目仍然会更好。

hdw1978：一个没有领导的团队会注定失败吗？

acockburn：如果没有经验丰富的人，你确实要依赖没有经验的人发现走向成功的道路。确实会发生这种情况，但是不如对立的情况频繁…人们总在原地踏步，不会生产任何东西。太多建模的害处是…你高高兴兴地做了半年的建模工作，却不知道它并不起作用。最好是立即建立一小部分，找出它是否会起作用。

bucolic：能按照书上说的做吗？

acockburn：我的书《面向对象项目求生法则》（编注：此书已由 UMLChina 翻译组乐林峰翻译，即将由清华大学出版社出版。）讲述了通过访问 20 个项目组所学到的东西。然后在我自己的一些项目中应用他们所阐述的方法。那本书里的方法的确有用——那都是成功的团队告诉我，经过我自己试用的。

bucolic：我想你非常倾向于使用原型。

acockburn：这里指的不是“原型”而是得到“反馈”。“原型”的问题是没有人知道是否应该抛弃它。我通常不喜欢丢开原型，除非是用纸做的原型。因为如果原型是用程序设计语言做的的话，投资商不会让你抛弃。所以我从不使用原型这个词。

bucolic：哦，那么你的“原型”能转化为产品。

acockburn：相反，建立系统很的一小部分。正确地构造，加强产品化。它不是原型，它是一个“增量”，是最终系统的一块。你在增加增量，所以丢弃(浪费)很少。

bucolic：哦，我们的问题是原型是不正确的，于是我们不得不放弃它。

acockburn：递增比丢弃原型要好(原型用纸做的除外)。两者都提供了真正重要的东西：反馈。

charles_y：最迷惑的问题是：从哪里开始？

acockburn：从哪里开始？起始有三个准则：应该提供（一小部分）价值给用户；应该能在两个月内建模完成；应该非常容易。如果你在两个月内构建小的、容易的、有用的东西。那么你的用户和投资方就会认为你在做使他们高兴的事。团队将确实地实践生产产品。这些都是极端重要的。一旦你有了小的有用的东西，你应该明白你可以找出你面对的最糟的问题。看你能否解决。你需要另两个月，找到构造内容的方法是构造两个列表：什么是用户需要的，将使用什么技术。你选定最容易体现价值的事，最难的事将会破坏整个项目。这对你的问题有帮助吗？

bucolic：你认为 XP 方法如何？

acockburn：XP：你会注意到 XP 完成我所说的一切事。因此我认为 XP 很好，除了你也必须产生一些文档。无论沿着这条路或在此之后。但是，如果你的产品没有文档，那么项目在其发布后六个月内会产生问题。

bucolic：嗯，但是 XP 不适合开发大项目。

acockburn：Bucolic，请告诉我你所指的“大”项目是什么意思，为什么 XP 不能用于它？

bucolic：一个大程序，我宁愿使用 rup。我认为，如果一个项目团队超过 10 人，XP 则不太有效。

acockburn：正确，XP 只能对 12 个开发者坐在一个有 6 个工作站的房间里有用。但是你得考虑，这十二个程序员将有可能产生和 35 个程序员一样多的系统数量。这 35 个程序员不坐在同一房间里且试图使用 RUP 和 UML。因此对于 XP 是“大”很难说。我参观过的项目，10 个程序员在一年内所做的是 26 个程序员在前一年未能完成的。因此需要小心的是，“大”团队有“大”问题。“大”团队常常效率不高以致于小团队高效的工作能做同样的事。不过，你说的也是…有些系统并不能由 12 个 XP 程序员在一个房间里构造。在那个情形下，必须分割团队，使用一些 XP 实践，代替其他的…或许结果是 XP，或许不是。我认为它将不是。但是很多 XP 实践仍然一直管用。

charles_y：你能告诉我哪里能够买到你提到的书吗？

acockburn：我不住在北京，因此我不知道能哪里能够买到书。问 UMLCHINA，他把这本书译成了中文。

charles_y：我认为你的意思是，应该找到容易的部分，对其进行作业。

acockburn：是的，这正是我所说的（我认为你应该找到容易的一部分，对其进行作业）

fuzhong：或者文档同时意味着交流和设计？

acockburn：在这种情形下，文档意味着你能存储，而且在六个月后能取回的东西，可以是纸张上或联机文档形式。

bucolic：只使用 UML 编写文档足够吗？

acockburn：我从未见过只使用 UML 的文档就足够了。有些事人们想相互交流，他们想用普通的语言进行交流。…例如“我真的想使这看起来象比萨投递系统，因此我的设计看起来象它运转的那样”…

bucolic：只记录我应该记录的。

acockburn：好，新主题：文档。这是一个很难的主题，我发现它很烦人，但是我会做我所能做的。RUP 或许允许所有的和已知的每一种类的文档（包括录象带吗？也许）。在 RUP 中定义了一些文档模板，有时候我想依照它来写文档，但是我发现有些模板没有用。回到文档。RUP 定义了很多模板，这很好。但是如果你试图使用所有的模板，会很糟糕。他们给你提供了一个模板“库”。当你去图书馆时，你不会试图阅读所有的书。你去商店买一些法律表格，你不会买所有的。你读你需要读的书，你买你需要买的东西。对于文档模板也一样。使用你需要的，其他的留在 RUP 模板库中。

bucolic：恐怕不是我认为它需要读或买，而是实际上它应该读或买。

acockburn：需要一些判断和实践以决定哪些是你需要的。

qingzuozhou：据我所知，有很多文档项目完成后没人会再看。哪些应该文档化呢？

acockburn：如果你花费了太多时间，项目慢下来了，没有任何明显的好处。我宁愿少花些时间，让人们更多地决定他们所需要的。最有用的文档看起来如下所示：UI-屏幕间的流程和连接。(使用状态机或状态图来表示)。体系结构：使用组件及其交互。给每一个主要的组件增加一句文本用来描述它的用途。OO 设计：类图表和一些示例序例图。不要太多的序例图，因为它们很快就过时了。(使用网页——是的，使用任何能加快你的工作和与你的用户交流的东西)。数据库：数据库图表，或者实体关系图。需求：用例+跨越系统边界所要求的数据格式。体系结构：和别的系统的连接。体系结构：应该是一篇短（1-5 页）的文本文档和一些体系结构或引导设计者的图片。也就是用词语“解释”系统组成结构和工作。这些词填平所有伴随 UML 图表带来的表达上的鸿沟。

bucolic：有时候我能从序列图中得到类操作。

acockburn：正确——我喜欢直接设计序列图——它们帮助我思考行为。

qingzuozhou：我有一个基于主机系统的软件，计划移植到 J2EE，你能给我一些建议吗？

acockburn：我不是很清楚 J2EE，对不起。

umlchina：你对一些诸如"Create ***"， "read ***"， "update ***"， "Delete ***"的用例有什么看法？

acockburn：(用例的一点补充：Create/Retrieve/Update/Delete 是 CRUD 用例（我试图在一个用例中写全四个，以缩减我们不得不追踪的用例的数量)。(我的一个朋友确实不喜欢我那样做。她喜欢让它们都分开（以便她能追踪开发和每一个分开的安全问题） (我们两人都认为对方所做的是错的：-）)但是没关系，因为我们都知道如何使项目最终成功）

bucolic：通常，是否体系结构该由一个人来设计？

acockburn：大多数人说只有一个人，但是我曾处在一个 40 人的项目中，我们用了三个人作为体系结构组，它工作得很好。

umlchina：有文章说 Rational Rose 的建模工具中不能把系统边界盒放入用例图中，系统边界从图中消失了...

acockburn：对你所说的关于用例图内容很感兴趣。我不知道 Rose 不会显示系统边界。当边界线显示不出来时，个人来说，我不在意，但是我那个不同意 CRUD 用例的朋友真的很难过（她叫 Susan Lilly，或许你看过她的文章《关于写用例的 10 个错误》，那是其中的一篇）恐怕说了太过于否定 Rose 的话。显然由于我不是专门研究 CASE 工具...但是就我来说，我没有听过任何人特别推荐 Rose 胜过其它工具，倒是听到有人推荐别的工具胜过 Rose，

我听到很多有关 Rose 的抱怨。因此看你的运气。最近，我听人一直在说 Together/J 是一个好工具，它确实允许你在 UML 图和代码间来回移动。umlchina 和 bucolic，你们会发现它对这些问题很有启发。我“从未”用过 Rational Rose。为什么不？因为我不画很多 UML 图。我在白板上画它们（可打印的白板最好），我确实画这些，我用手工画，或者用 Visio 一个纯画图工具， 非常便宜非常易于使用，它不产生任何代码。从 1992 年到 2000 年，我简单地试用了一些 Case 工具，发现它们浪费我们的时间。不如找个纯画图工具直接画。这就是为什么当有程序员告诉我 Together/J 确实有用时我感到振奋的原因。当然我自己还没有使用它：-）因此我不是真的知道。

qingzuozhou：讲讲别的工具。

acockburn：别的工具：可打印的白板。数码相机也几乎一样好。我的朋友告诉我 Pixid 软件（好象是 www.pixid.com）很适合清理图片以便以后你能真正使用。其他工具：配置管理/版本控制是继编译器后我推荐的第一大工具。其他工具：性能测量工具。其他工具：Lotus Notes 保存用例非常好，也适合错误报告数据库。你会注意到这些工具：1、编译器，性能调整工具，帮助程序员写代码。2、配置管理工具：帮助团队保持代码同步。3、Lotus Notes：帮助团队保持他们的非代码同步。4、Visio：帮助人们方便地画图。5、最终工具：得到你自己的 JUnit， CppUnit，或 VBunit，开始对你的代码的小部分进行退化测试。有人说刚开始写少量的测试集合很容易写和易于运行，从未再放弃它们。你会注意到我的工具列表中完全没有 Case 工具。

xkfy：smartdraw 怎么样？

acockburn：Smartdraw？我不知道——但是建议使用任何有效的工具。

qingzuozhou：是的，为什么？

acockburn：从 1991 年到 2001 年会见的项目组，我只听到过两个项目组说 Case 工具对他们有用。问一下：我想让这个工具为我做什么呢？如果答案是“产生代码”，那么或许这工具浪费金钱。如果是别的答案，那么我上面提到的其他工具也能完成。

qingzuozhou：这是我第一次听到这样的关于 Case 工具的评论。非常有趣，谢谢！

adamli：Alistair，这是你第一次和中国的开发人员交流吗？

acockburn：Hi， Adamli - 是的，第一次。我完全不知道你们在哪里，你们的年龄、背景或任何别的东西。如果我已经做过好几次，那么或许我会说我很忙。

bucolic：Alistair，如果我有一些问题要问你，我可以给你发 E_MAIL 吗？

acockburn：关于给我发邮件：我一定尽力回复。但是我正努力在下两个星期内完成我的下一本书。因此，在近几个星期我不会回答任何难的或很长的问题。此后，我会回答得更多。我的 Email 地址：acockburn@aol.com，也可以在我的网站上阅读所有的文章。http：//members.aol.com/acockburn。我把我大多数的文章都放在上面了。我下一本书已经完整的放在上面了（尽管是第三次修订版，不是最终版），也可以访问 http://www.crystalmethodologies.org/，那里有一个 wiki 服务器，我们可以继续讨论。

征稿

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

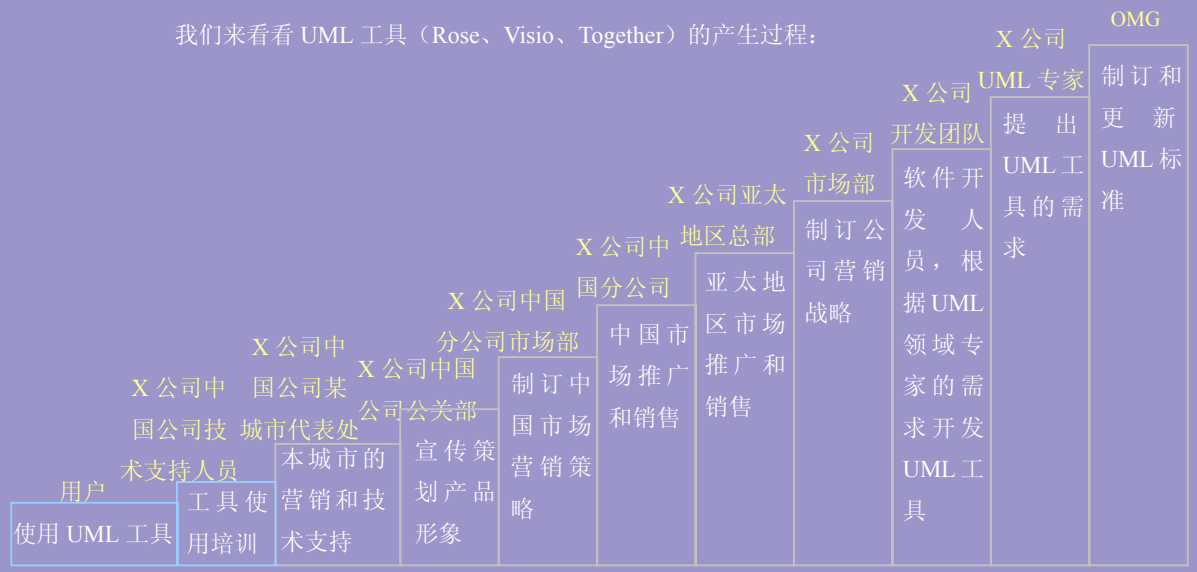
对他们是一点，对我们是一切

UMLChina，the REAL thing >>

微软开发出了占领导地位的产品是 Windows 和 Visual C++，但大家看得最多的<<Windows 程序设计>>（Charles Petzold），《Visual C++技术内幕》（David Kruglinski）等书籍的作者并非开发相应产品的微软员工。这就是现代分工。

建模领域也是如此。关于 UML，最畅销的书籍“UML Distilled”的作者不是 UML 的最初发起人，也不是 OMG，而是 Martin Fowler。

我们来看看 UML 工具（Rose、Visio、Together）的产生过程：

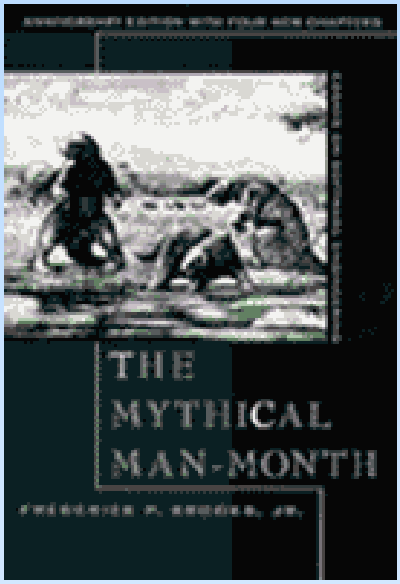


- 工具厂商的产品是工具，不是工具培训，更不是开发技术培训。
- 公司名字是个筐，什么都可以往里装。实际上，目前很多著名公司在中国的主要任务并不是研发，而是销售产品——真正在这个领域做研发的人在大洋彼岸。每个高质量的工具背后都会有资深的对象技术和方法学专家（如 Ivar Jacobson、Peter Coad…）支撑着，如果开发团队在购买产品之后，能得到这些大师们相应的开发方法“培训”，相信会受益无穷。但这样的机会太少了。

UMLChina 培训是 UMLChina 真正用心去打造的“产品”，不是其他产品的附属赠品——内容组织，教材和现有的各种类似培训并不相同，而且会随着技术变化，不断改进。我们小心翼翼地希望 UMLChina 培训能够在这一点点领域里尽量做好，真的能够帮助开发团队（或个人）正确地使用对象技术，开发出更好的软件。

向开发出优秀方法和工具的公司致敬，我们愿意充当斯达舒、吗叮啉，帮助您消化适合您胃口的优秀软件开发技术。

《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！

界面耻辱纪念堂--隐喻使用不当

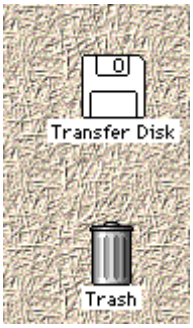
iarchitect 整理，金哲凡 译

查看评论

使用隐喻是界面设计中常用的一种方法，这种方法通过激活用户已有的知识，使用户更好地理解应用程序。一些好的界面隐喻已经广为人知，比如 VisiCalc 的帐本，施乐（Xerox）首创的桌面，Quicken 的支票簿等。而事物总有其反面，由于隐喻使用的不当而损害程序可用性的情况也非常多，比如以下的一些例子。

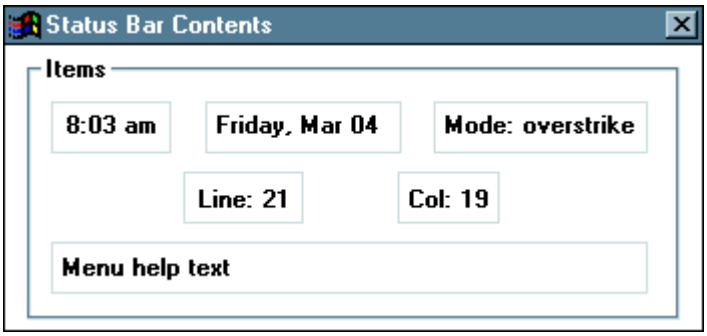
最近更新 1999 年 8 月 29 日

Manhaeve Hendrik 先生给我们寄来了这张图片，它是 Mannesman Tally 程序的打印控制对话框。我们还是第一次见到用录音机隐喻来控制打印的，上帝保佑，希望这也是最后一次。停止和暂停按钮，尽管很勉强，毕竟尚可理解，可倒带按钮呢？Manhaeve 讽刺道：“这个按钮有什么用？把纸倒卷回去，并把打出的字擦掉？”



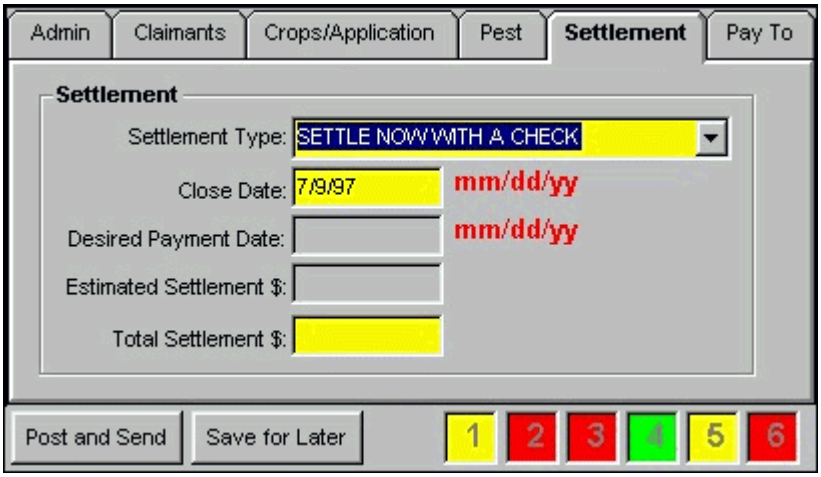
Macintosh 的垃圾箱是一个很不错的直观的隐喻，其作用为删除文件。不幸的是，它的设计者给它增加了一项完全不直观的功能：弹出磁盘。把磁盘的图标拖到垃圾箱，它就会从计算机里弹出。

Macintosh 显然滥用了垃圾箱的隐喻。他们赋予垃圾箱以神奇的力量，却全然不顾及这么做破坏了这个隐喻与“删除”操作之间的自然联系。结果，新手对这个隐喻感到手足无措，有经验的用户也颇有怨言：“我只想让计算机弹出磁盘，并不想把盘上的文件删掉”。



Watcom 的 C＋＋Editor 是一个可怕的例子，它告诉我们不能忘记使用隐喻时必须遵循界面设计的基本原则：隐喻对于用户应该一目了然。设计者应该提供看得见的线索来说明隐喻的功能。

上图所示的窗口的作用是定制应用程序的状态条。用户可以通过拖－放操作来指定显示在状态条上的信息的类型和位置。用户需要做的是点击一个方框，并将其拖到状态条合适的位置上。不幸的是，这个窗口没有任何提示。由于缺少必要的信息，用户不太可能直观地了解这个隐喻地含义，而不得不求诸帮助文件，而这也没有办法从这个窗口直接做到。



我们的一位访客给我们寄来了一组他所在公司使用的一个应用程序中出现的汽车“尾灯”隐喻的图片。他想知道我们会把这个素材放进界面耻辱纪念堂还是界面光荣纪念堂。我们给它找到了合适的去处。

“尾灯”出现在窗口的右下角，其作用是当用户在一个复杂的标签对话框中输入信息时，显示进度。尾灯 1 对应第一个标签，尾灯 2 对应第二个标签，其他依此类推（看出问题了吗？）。尾灯有三种颜色：

黄色： 已输入了一些信息

红色： 尚有必须的信息没有输入

绿色： 所有必须的信息都已输入

这个界面的设计上存在不少一般性的问题，但最严重的问题还是出在尾灯上。

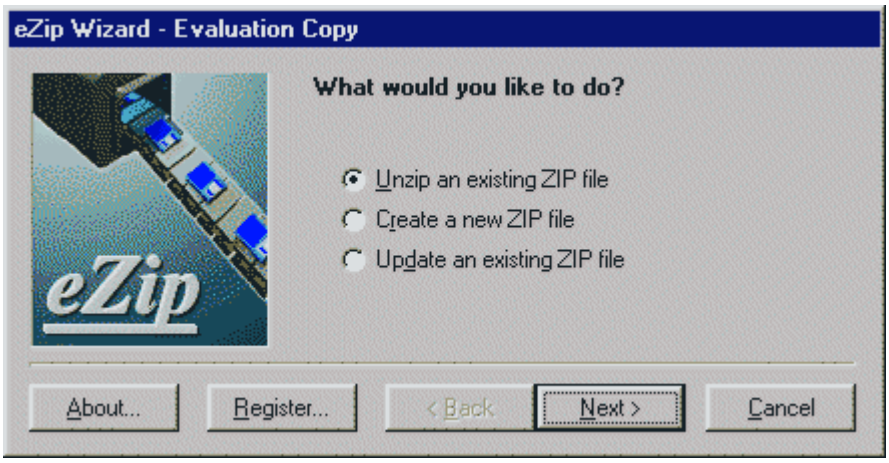
1、 信息冗余 尾灯隐喻要求用户了解三种状态各自的含义，而实际上只有一种是用户必须知道的。对用户来说，唯一重要的信号是某一分页上缺少必须的信息。在这里，使用三种颜色完全没有必要，只会分散人的注意力，而且显得很混乱

2、 自相矛盾 请注意，在上图中，尽管尾灯表明输入信息不完整，“Post and Send”按钮现在却是激活的。我们认为，相对于简单的 disable 该按钮，设计者特意开发出尾灯隐喻，是为了能传递一些额外的信息，即哪个分页还需要填入数据才能发送。但目前这种实现，却出现了这样的情况：它告诉用户表单可以提交，又告诉用户表单不可以提交。

3、 位置和标号 尾灯有 1～6 的标号，而标签上没有。这要求用户物理上或心理上要找到数字相关的标签。而尾灯与标签的距离很远（实际表单比上图大得多），更增加了用户辨识的负担。

我们建议采用的替代尾灯的方案如下图所示，在物理上靠近标签的地方增加一个特殊的“需要数据”的标志。

如果要引导用户完成一个复杂而陌生的操作过程，Wizard 隐喻是非常有用的。不幸的是，不少设计者错误地将 Wizard 的作用理解为使程序“易于使用”，结果却往往适得其反。



ediSys 的 eZip Wizard 就是一个在界面设计中错用了 Wizard 的例子。eZip 的功能是创建、修改和解压缩 Zip 文件，网虫们对这些操作再熟悉不过了。Zip 压缩基本上可视作一个管理文件的过程，Zip 文件就像一个包含一个或多个文件文件夹，用户打开 Zip 文件，并进行读取文件、创建文件夹、加入文件、删除文件等操作。

Wizard 的缺点在 eZip 中特别明显，那就是界面由一串强制性的线性序列组成，程序开发者将此序列固化在程序里，而用户必须遵循。对于第一次使用或不常使用的用户来说，这是有帮助的。但对有经验的用户来说则令人反感。在 eZip 中，这一序列表现为一串固定的问题：

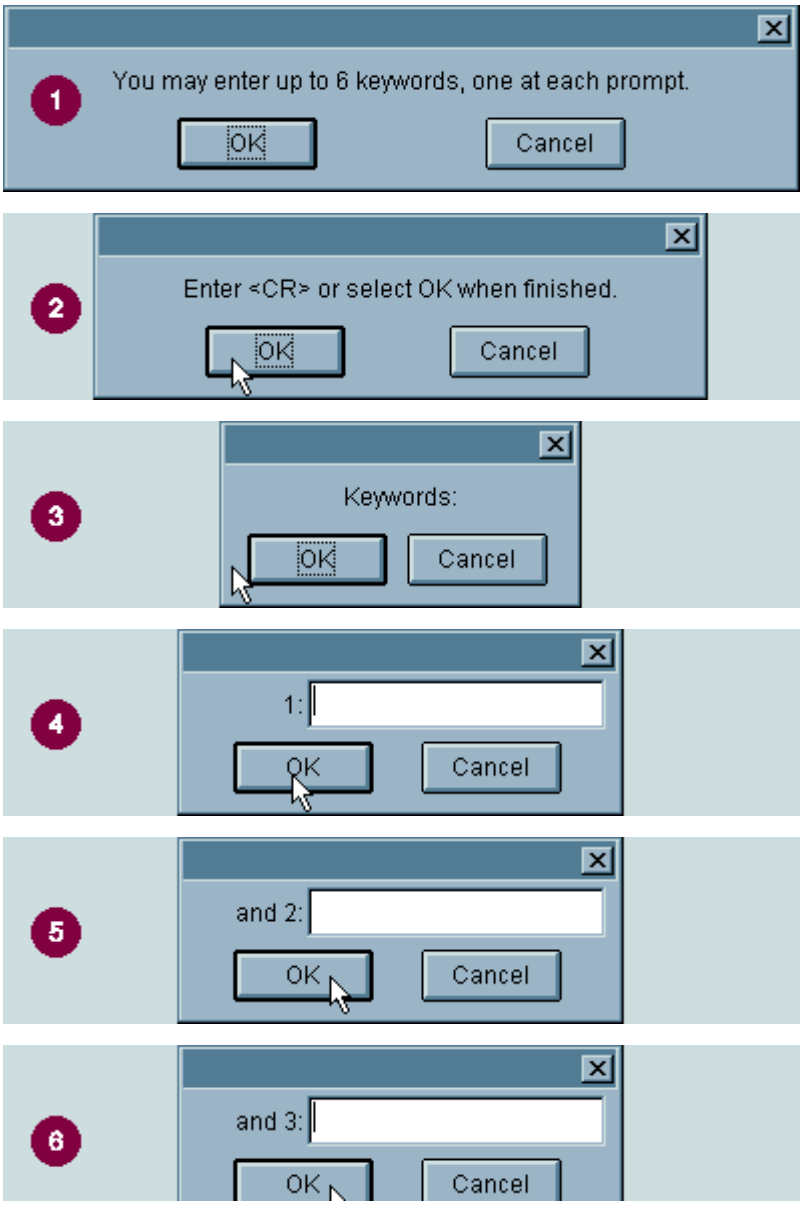
- What do you want to do?
- What options do you want?
- What name do you want to use?
- 其他问题

用户必须回答每个问题，才能进入程序内定的下一个步骤。用户第一次使用时，会发现这种结构很好用，但很快程序的僵化死板会令他厌烦，而频繁进行文件创建、修改操作的用户则会感到不可忍受：连简单的加入、删

除文件的操作都得象受审讯一般回答一连串问题才行。

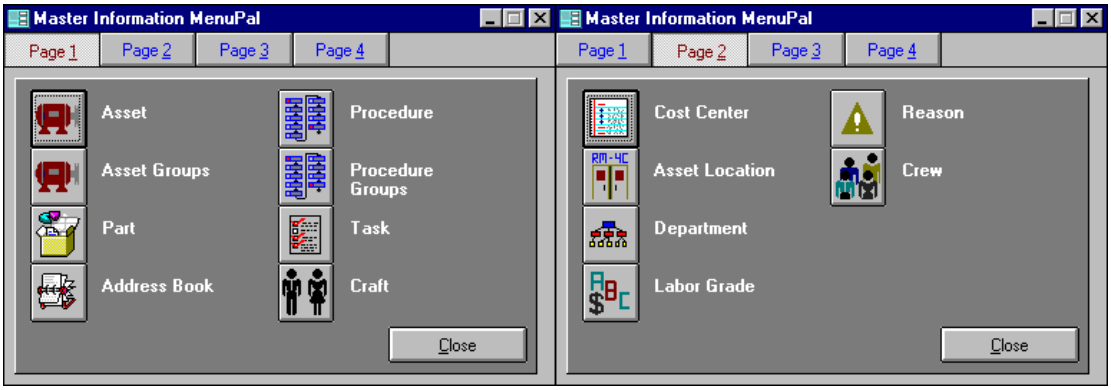
Wizard 应用于不常用的复杂流程，比如配置 modem 或硬盘分区。如果使用不当，Wizard 会限制用户对程序的控制，并容易给人这样一种印象：把用户当作了傻瓜。作为用户我们当然不希望被当作傻瓜，我们会认为程序愚蠢。

以上讨论的 Wizard 隐喻的基础是一个线性的问答模型。下图是 CompuServer 的 WinCim 程序的数据库搜索操作界面。CompuServer 是一个以关键词搜索的常用问题的数据库。如本例所示，一个简单的搜索要求用户与大量对话框交互，数目之大令人惊奇，并且用户得记住好多步之前显示的指令。



线性隐喻本质上是一个问答过程，许多用户也正如此认为。在这里，只要一个对话框就能方便地完成任务，而不需要如图所示的 6 个连接起来的对话框。这清楚地表明程序员只考虑了他（她）编程的方便，而全然无视用户的感受。

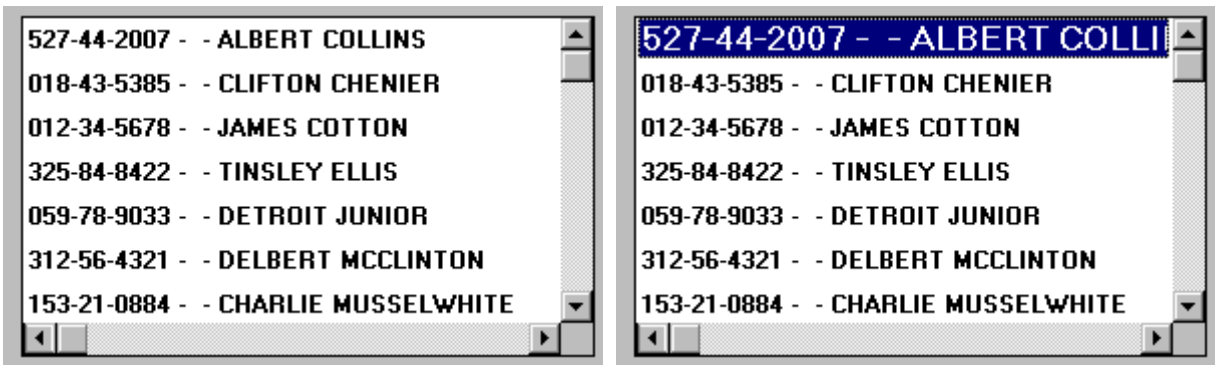
在界面设计中使用隐喻的作用是提供一些基本信息，使用户能产生对系统的正确的理解。有时从界面设计也能看出设计者的思考模式，如本例所示。我们将之称为“可用空间”隐喻，它显示了程序员如何在已有的系统上加入功能，逐步完成任务。



这个伪标签对话框的作用是让用户访问一个复杂系统中的不同类型的信息。不同种类的信息放置在某一个伪标签上的根据是可用空间的大小。当一个标签满了，设计者简单地增加一个标签，并加入新的信息类型。

对话框的标签与其所含信息毫无关系，更有甚者，信息也无明显的分组。比如，“Tasks”放在 Page1，而“Task Types”在 Page3。设计者感兴趣的好像只是把访问功能扔给用户，而不管其方式对用户来说是否有意义。

当分页包含关联信息时，标签对话框隐喻才有用。当难以分组时，简明的字母序的功能排列会更为有效。



我们从 Flexi 中找到了这种罕见的隐喻，这是中等规模业务的记录。我们将之称为“放大镜隐喻”，因为这个列表框会放大显示选中的条目。我们猜想设计者的用意是让用户更易分辨哪个条目被选中了，但我们认为一个单纯的蓝色方框足以实现此目的。放大不仅没有必要，实际上还使信息更难阅读。

再来看看回形针动画，这是微软为了给 Office97 加入交互助手而作的愚蠢而可悲的努力。当你撰写一封商业信函，填写帐目，或使用高生产率的软件进行其他工作时，回形针老是在屏幕上，提醒你它正在监视你的行为。当你开始一个重要的操作时，回形针突然活了过来，跳出来，扬着眉毛，在窗口里左右舞蹈。当你启动一项不常用的操作，回形针会打断你并问你是否需要帮助。

我五岁的侄女爱死了这个回形针，我每次打印，她都会兴奋得尖叫起来，因为回形针会模仿纸张在打印机上的动作，在滚轴上被压扁。五岁的孩子们认为回形针很可爱，Microsoft 应该从中意识到一些问题：五岁的孩子是不会购买$500.00 的软件包的，这是成年人的事，而大部分成年人会很快对这种“可爱”感到厌烦。

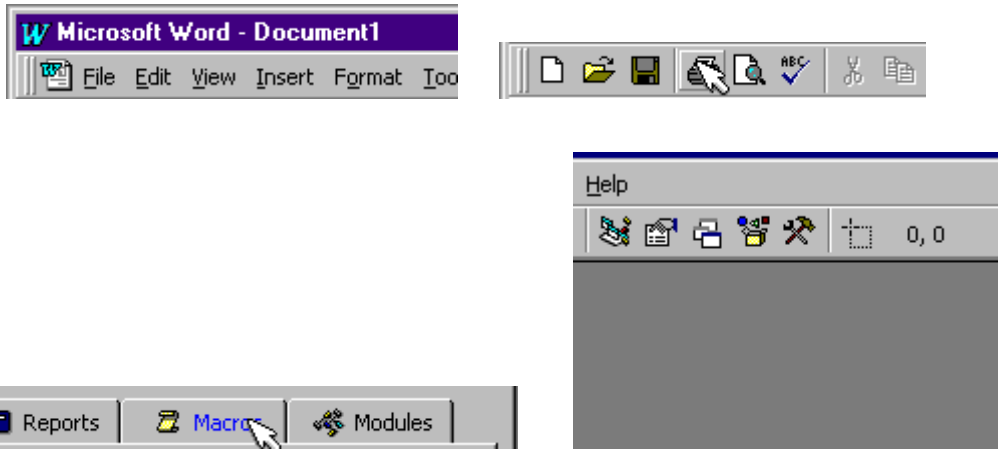
问题并非出在交互助手这个想法上。Isys 的创立者领导了一项关于交互助手的研究项目，其中一位研究生敏锐地指出了回形针的缺陷：回形针不停地动作，左右摇摆，眨眼，甚至弹出一条条提示。当他“说话”时，你除了听着之外毫无办法。用户可以将之关闭，但只是暂时的，当他（她）下一次访问帮助文件时，回形针又活了过来。

看起来设计者对于实现一个交互式助手过于热情，用户初次使用时的一声“wow”和长期使用后的反应哪个更重要，他们已经无法分辨。而市场部门居然也允许这种尝试，真是匪夷所思。

顺便给 Microsoft 提个醒：眨眼不是一种能让人信任的特征，也许“小狗”作为缺省设置会更好。

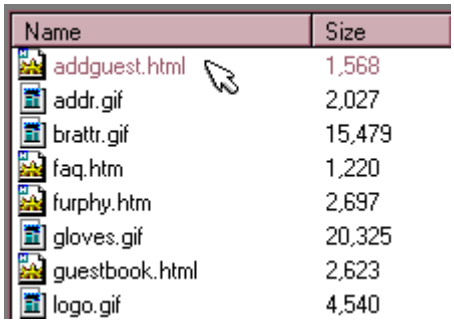
每当我们不得不使用 Microsoft 的新产品时，就不禁想起 1957 年上演的一部由 Spencer Tracy 和 Katherine Hepburn 主演的喜剧《Desk Set》。Tracy 扮演的角色，是一个工作出色的专家，他承担了一项不受人欢迎的任务：用一台叫“EMERAC”的电脑替换某大公司的所有职员。为了尽量让观众讨厌这部电脑，“EMERAC” 被描绘成

墙一样大小的仪表盘，布满了闪烁的灯炮，并且会发出刺耳的警铃和尖叫声。



我们现在将“EMERAC 隐喻”这个短语用于对某些界面设计的评价，专指那些除了耽误用户手上的工作，分散用户注意力之外别无用处的特征。如上图所示，Microsoft 的应用中充满了 EMERAC 隐喻。这些特征对用户没有任何好处，它们唯一的作用是使程序看上去与众不同，也许还可以显示某些人的编程能力。

“EMERAC 隐喻”不值得模仿。我们推荐你使用 KISS（Keep it Simple, Stupid）隐喻，你的用户会因此感谢你。

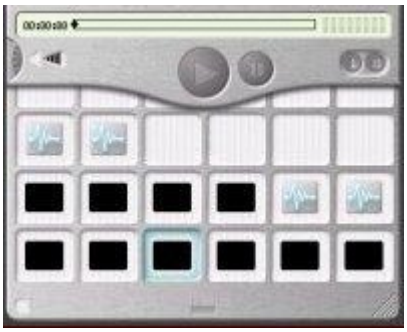


Spidersoft 的 WebZip 使用了一个特殊的隐喻。当光标经过列表的条目时，条目的颜色会发生改变。条目的变色能反映什么呢？只反映光标经过了它。因此，这是“EMERAC 隐喻”的又一个例子。不幸的是，WebZip 走得更远，如果光标停顿了一会儿，光标所指的条目会自动被选中，不需要点击鼠标。

用过这个控件之后，很快就会发现它的问题，这种技术很容易导致无意识地选中。想象这样一种情况，你通过“正常”地方式，点击选中一个文件名，然后，当你把光标移向一个菜单条和命令按钮时，电话铃响了，或者你停下来想想接下来使用哪个命令。如果碰上 WebZip，你的选择会被取消，而光标下的条目由于光标地偶尔的停

留而被选中。由于 WebZip 支持选中多行，片刻的思索可能会毁掉一系列复杂的移动和选中操作。

Microsoft 将这种光标停留自动选中模式称为 Hot Cursor。我们惊奇地发现 Microsoft 已在 Win95 中支持这种“EMERAC”式的隐喻。开发者只要作一定的设定即可。我们希望使用这种无用的特殊效果来损害他们的程序的人越少越好。这种隐喻分散人的注意力，使得某些 List 控件不同与其他的 List 控件，并会导致计算机莫名其妙地取消用户的动作。



Apple 的 QuickTime 4.0 Player 有一个地方特别麻烦：用来收藏用户多媒体文件的 Favorites Drawer。当有文件加入 Favorites collection 时，程序会用一个图片来标志这个文件。可所有的声音文件用的是同一个图片，视频文件则用缩小后的第一个帧来代表这个文件。

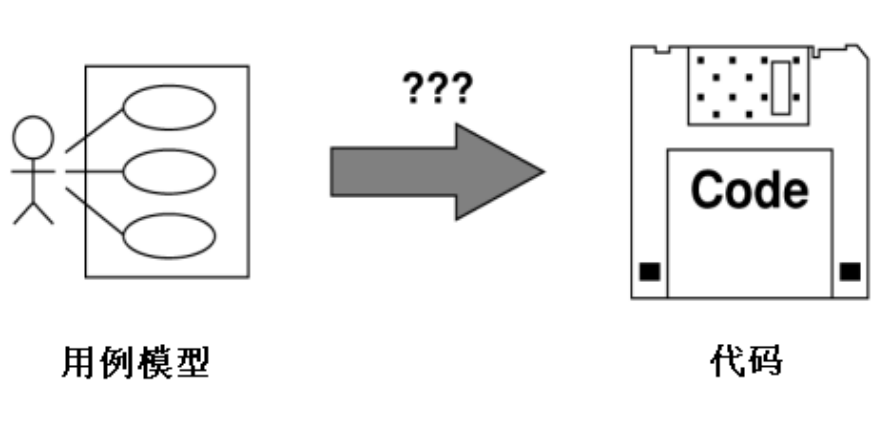
QuickTime Player’s drawer 可能是第一个“巧克力盒子“隐喻，如同电影《阿甘正传》中 Forest Gump 说的：妈妈总是说生活就像一盒巧克力，你永远不会知道你会得到什么”。

选择某一声音文件与从一盒巧克力中选出某一颗并无不同：因为它们看上去都一样，要想找到你要得文件就得一个一个地试，就像每个巧克力都咬一口。找一个视频文件也差不多。因为在 QuickTime Player’s 中视频以第一帧来分辨，而大多数视频文件地第一帧是空白（除了 Apple 自己提供的），用户没法从图片分辨出一个视频文件。

QuickTime Player 的项目负责人非但不承认这是一种糟糕设计的界面，反而把责任推给内容提供商。在他看来，视频制作者应该将第一帧作得能标识自身，可我们知道某些国家版权法明确规定第一帧必须用于版权信息；更荒谬的是，声音制作者应改变 WAV 和 MIDI 文件的格式，使之包含一个图片以标识自己。Apple 告诉对此不满的用户他们应该向内容提供者提议改变文件格式，以便 QuickTime 能以一种合理的方式标识文件。世上可能不会有比他们更傲慢自大的设计者了。

UMLCHINA 培训

北京公开课：6/29-6/30



详情请垂询：UMLCHINA 北京培训中心

联系人：傅光远

电话：66130237，13910698898

EMAIL：bjcourse@umlchina.com

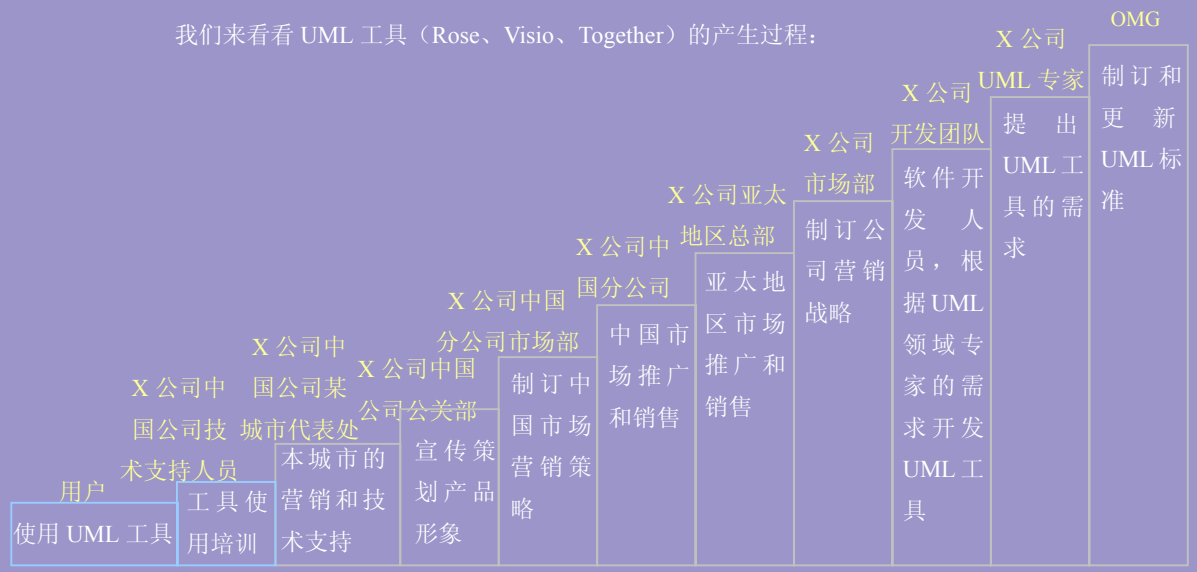
对他们是一点，对我们是一切

UMLChina，the REAL thing >>

微软开发出了占领导地位的产品是 Windows 和 Visual C++，但大家看得最多的<<Windows 程序设计>>（Charles Petzold），《Visual C++技术内幕》（David Kruglinski）等书籍的作者并非开发相应产品的微软员工。这就是现代分工。

建模领域也是如此。关于 UML，最畅销的书籍“UML Distilled”的作者不是 UML 的最初发起人，也不是 OMG，而是 Martin Fowler。

我们来看看 UML 工具（Rose、Visio、Together）的产生过程：



- 工具厂商的产品是工具，不是工具培训，更不是开发技术培训。
- 公司名字是个筐，什么都可以往里装。实际上，目前很多著名公司在中国的主要任务并不是研发，而是销售产品——真正在这个领域做研发的人在大洋彼岸。每个高质量的工具背后都会有资深的对象技术和方法学专家（如 Ivar Jacobson、Peter Coad…）支撑着，如果开发团队在购买产品之后，能得到这些大师们相应的开发方法“培训”，相信会受益无穷。但这样的机会太少了。

UMLChina 培训是 UMLChina 真正用心去打造的“产品”，不是其他产品的附属赠品——内容组织，教材和现有的各种类似培训并不相同，而且会随着技术变化，不断改进。我们小心翼翼地希望 UMLChina 培训能够在这一点点领域里尽量做好，真的能够帮助开发团队（或个人）正确地使用对象技术，开发出更好的软件。

向开发出优秀方法和工具的公司致敬，我们愿意充当斯达舒、吗叮啉，帮助您消化适合您胃口的优秀软件开发技术。

通俗而且深刻的软件团队管理杰作—UMLChina 推荐

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

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

中文译本即将发行！

界面耻辱纪念堂--可视元素

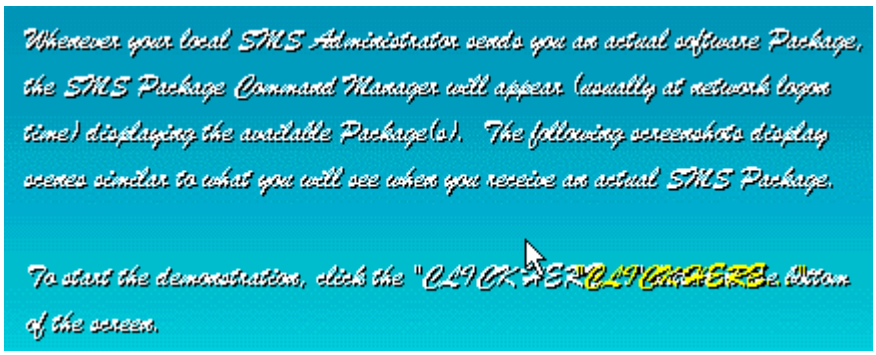
iarchitect 整理，wanghongbao 译

查看评论

看来，一些开发人员用自己的方式走过头了，以至于他们的程序很难阅读或理解。通常，都是因为他们寻求自己的程序看起来新颖、与众不同，或者，原谅我用这个词， "酷"。新奇的事物有它的地位，如果没有它，我们就不会有图形界面。但是，当我们实现的时候如果没有将增强用户的交互作为目标，往往会导致更差的界面，用户从轻微抵制开发人员选择的颜色，到努力阅读不合适的字体而导致眼睛疲劳和头痛。

最近更新 1999 年 9 月 19 日

Jim Brooking 给我们发送了几个从 Popkin 软件公司的 *System Architect* 上截取的图片。其中两个图片说明，由于提供对话框调整大小能力而导致显著糟糕的方法。这些图片太大，不能在此包含，因此，我们提供另外的页面讲述它们。如果你曾经致力于如何提供可调整大小的对话框，请仔细阅读这个页面以避免 Popkin 的技巧。



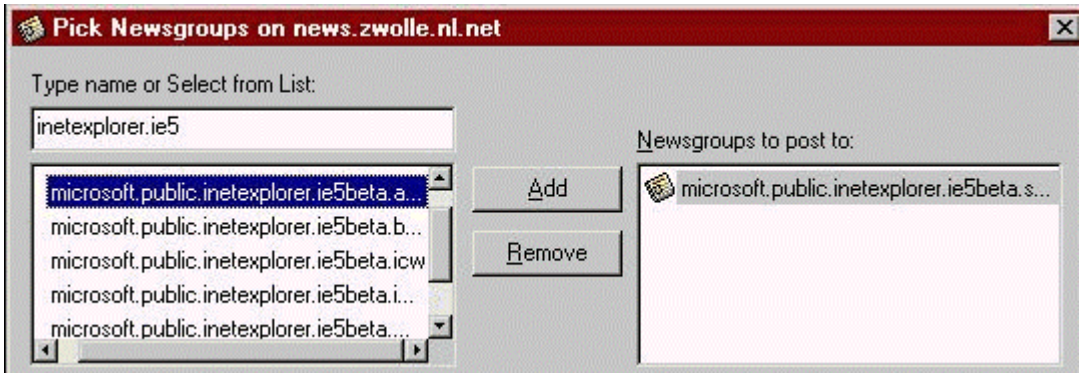
我们已经给程序开发者提出了一个新的规则：

在你的设计和你的用户发生冲突之前，你必须至少看一看它们。

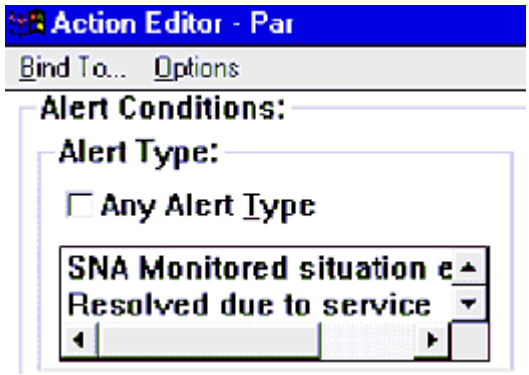
以上图片截取自一个教程，它发布给一个非常大的成员组织，该教程指导他们使用一个新的软件管理系统。图中明显地表明，字体、颜色、和背景的选择已经使这个教程几乎完全不能阅读，而导致这个教程毫无用处。

努力阅读这个图片中的文字，注意所需的努力。产生这样一个可怜的屏幕特性组合的唯一原因是开发人员从来没有看看他或她的创造，除此之外，没有别的理由。

Berco Beute 发给我们微软 *Outlook Express Newsreader* 中的一个对话框图片：

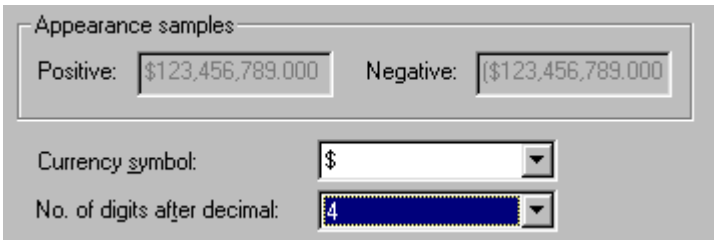


这个对话框出现在用户想把消息发送给一个以上的新闻组时。这里有一个不可忽略的问题：在这个窗口中没有足够的空间显示新闻组的全名。程序提供一个水平滚动条，但是如图中明显表示的，这个滚动条没有用处。并且，在典型的微软设计风格里，这个对话框是不可以调整大小的。设计者固定死了，质量检查部门固定死了，很显然，由于微软的可用性测试部门没有发现这个明显的错误而把它留下来了。



Leif Almberg 发给我们一个从 IBM's NetFinity（一个管理网络和远程计算机的程序）截取的图片。这个图片由一大堆混乱的带边框的控件组成，这些控件占据了整个屏幕。由于他们希望把所有东西填充在一个单独对话框中，IBM 的设计者不得不做一些牺牲。值得注意的一个例子是，设计者牺牲了用户阅读某些控件的能力，例如可

能的警告条件列表，如上所示。用户可以在列表中滚动，一次看一个警告，并且为了看到警告不得不左右来回滚动。IBM 被迫使这个控件具有**更少显示**的效果。



Windows95 中，地区参数选择程序说明了几个特色，这些特色让用户很难了解自己已经做了什么。在此，第一个问题，他们想给样本字段没有响应能力的外观，而不是只读外观。他们宁可使用标准的文本颜色而不是失去响应能力文本的颜色，第二个问题更清楚：他们没有足够的空间给特定的格式完全显示。例子数值应该在小数点的右边有 4 个数字，并且负数应该有一个右括号。唉！

微软 Office97 产品的用户应该认识这个新的界面“特性”。由于一些无法说明的原因，微软决定，在指针经过菜单标题时，在它的周围加一个边框，使得菜单标题像一个命令按钮。在给你的程序加入该特性之前，记住这个特性完全没有提供任何可用性好处。仅仅是一个不必要的花样，甚至会出现我们不期望的效果——它分散用户的注意力。

微软加入这个“特性”仅仅是为了使得他们的程序看起来不一样，和 Office95 里的渐变阴影标题条没什么不同。我们只能希望这个实践和渐变标题条一样短命。我们的建议：不要为了让你的菜单标题像微软而浪费时间；你的时间应该花在改进界面上。

微软的 Access97 说明了另外一个不必要的花样。当指针移动通过主数据库窗口的标签时，标题以一种更亮的颜色显示。和 Office97 中菜单标题的动态边界没什么不同，动态地改变界面元素的颜色对用户没有益处，而是有我们不期望的效果——分散用户的注意力。

微软没有把这种新技巧在这个对话框的命令按钮上使用，也没有在 Access97 中其他有标签的对话框中使用，我们可以由此推断，微软对这个新技巧的态度也是不确定的。

更多的迹象表明，关于在程序里使用新的动态界面元素，微软的态度是不确定的，其中一个是仅仅需要对比一下 *Office97* 里的“Coolbars”和“标准工具条”。Coolbar 按钮直到用户指针通过的时候才成为按钮（否则是平的）。工具条按钮，像在 *Office97* 工具条里的一样，不需要用户交互；他们可以直接认出按钮控件。是我们发现了微软的争论，还是微软不小心的设计？

很不幸，*WebZip* 借鉴了 许多类似 EMERAC 的 *Office97* 特性，包括在对话框中不适当地使用动态工具条按钮（按钮直到指针通过时才成为按钮），而不是使用标准工具条按钮。

下面的插图证明使用动态按钮的基本问题：用户无法凭扫一眼来断定界面元素真的是一个控件，相反，如果用户没有在附带的图像上移动，就不能断定它**是否**一个控件。在插图中，上面的两个图像是 WebZip 的附带图像：它们没有提供功能。当然，这个程序使用动态按钮，新用户一旦学到图像会在指针通过时变为按钮，会理所当然地认为附带图像也有这个特性。

插图中下面的图像暗示使用动态控件另外的问题：静态图像实际上也可能是控件，但是，由于它们在指针通过时没有动态行为，会被认为不是控件。那个“Scroll Down”（向下滚动）按钮不“浮动”，指针也不变成手形（在 *WebZip* 的其他地方是这样），并且那个“Download Now”按钮，用一个环行三角表示，在响应指针移动时并不变化）。

另外，使用动态按钮的问题还有，不使用鼠标进行输入时存在很严重的限制。有意义的键盘访问键不能分配给这种控件，而且它们不支持声音输入。再有，我们发现在有意义的标题按钮（例如，上面图像的"New"、"Open"按钮，和下面图像的"Download Now"按钮）的场所使用含义模糊的图像是很差的设计。我们对 WebZip 的建议是：

开始注意 GUI 中的 U（用户）

WebZip 使用了一个很像这个独特网站的导航栏。当然我们可以看到一个复杂的网站需要这样一个导航栏，但是在一个相对简单的桌面应用程序里使用有些驴唇不对马嘴（在我们看来，他们仅仅是努力模仿一个站点的外观 ，实际上是另一回事.）。

我们把这个例子列举在这里，因为它说明了两个视觉问题：第一个问题是，这个导航栏可以滚动，但设计者隐藏了这个事实。设计者选择自己设计的而不是标准的滚动条，熟悉这种滚动条的用户很少。

设计自己的控件时，有一个你要遵循的好准则：

使用操作系统提供的控件。用户已经对它们非常熟悉，真正懂得它们的用途和操作方法。

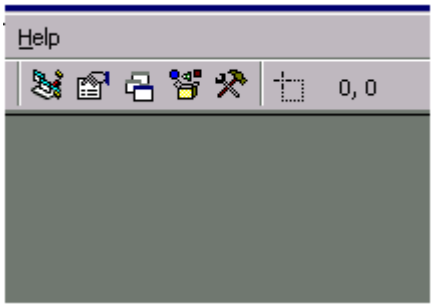
当我们最后意识到这个导航栏是可以滚动的，我们选这个例子的第二个原因就变得很清晰了：WebZip 为新用户提供了一个“Quick Start”选择，这个选择使用了一个向导风格的界面来清晰地说明如何使用这个程序。不幸的是，这个选择是导航栏的最后一项，直到用户知道如何使用这个定制的滚动控件才为用户所见。

这儿是另外一个规则：

让面向新用户的特性可见和可访问

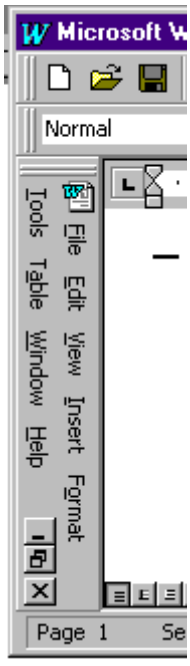
不要将程序的可用特性隐藏起来。

当我们第一次注意到 *Visual Basic 5.0* 菜单的动画效果“特性”时，我们只能嘲笑这种特性的傻气。事实上，我们并不觉得特性本身傻气，而是微软为这个特性投资，然后将这个特性应用到他们所有的主流产品（例如，*Office97*）中的事实。



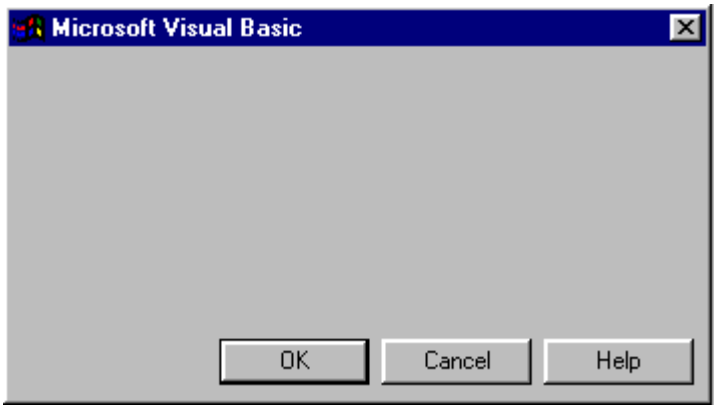
既然设计者有足够的时间，他们或许应该花在提高微软产品的可用性上，而不是花在提供一些毫无用处，只是让小孩用户叫好的花样上。

在 *PowerBuilder* 里，某种工具条按钮使用了一个更不平常的技巧，我们觉得使用这个技巧让这个产品更难使用。如图所示，这种工具条按钮有一个关联的下拉控件，可以显示其子工具条。这个特性本身并不坏，但是我们发现一个问题——工具条改变图像反映被选中选项，使得我们忘记如何返回到原始的选择。

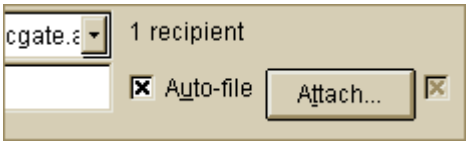


“一致性使得界面为大家熟悉并且可以预见”

(*Windows 用户界面设计指导*， 微软出版社)



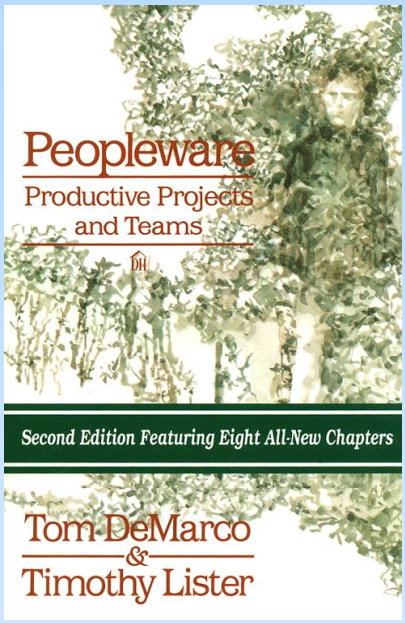
我们希望 Visual Basic 5.0 的大部分开发人员从微软的设计指导里，而不是从微软的产品里学到一致性。上面的图像是 VB5 编程环境里编辑对话框。大多数情况下，VB5 的开发人员依赖于微软的风格，但是，微软新近采用的布局命令将按钮的位置水平地放置在对话框的右下角。不幸的是，负责这个产品的大部分开发人员不知道这个新标准。命令按钮布局除了图中所示的不同位置外，第三方（比如，Crystal Reports）提供的对话框甚至采用了更富有创造性的风格。



这个图片是从 Compuserve 的 *WinCim* 电子邮件程序里截取的，它说明了一个典型的问题——由于把控件作为指示器并且使用不当。许多 Windows 控件的一个长处是：既可以作为控件，也可以作为指示器使用。无论如何，当设计者把一个控件*仅仅*作为指示器时，可能会导致很多问题。在图示的情况下，用户未必能注意到 Attach 按钮右边一个 disabled checkbox（禁用的检查框）是表明一个附件被选中了。相反，那个 Auto-File checkbox 清楚地表明这个消息会自动归档。

说明：在本文中，如果某些东西足够重要，该主题将会重复多次。

《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！

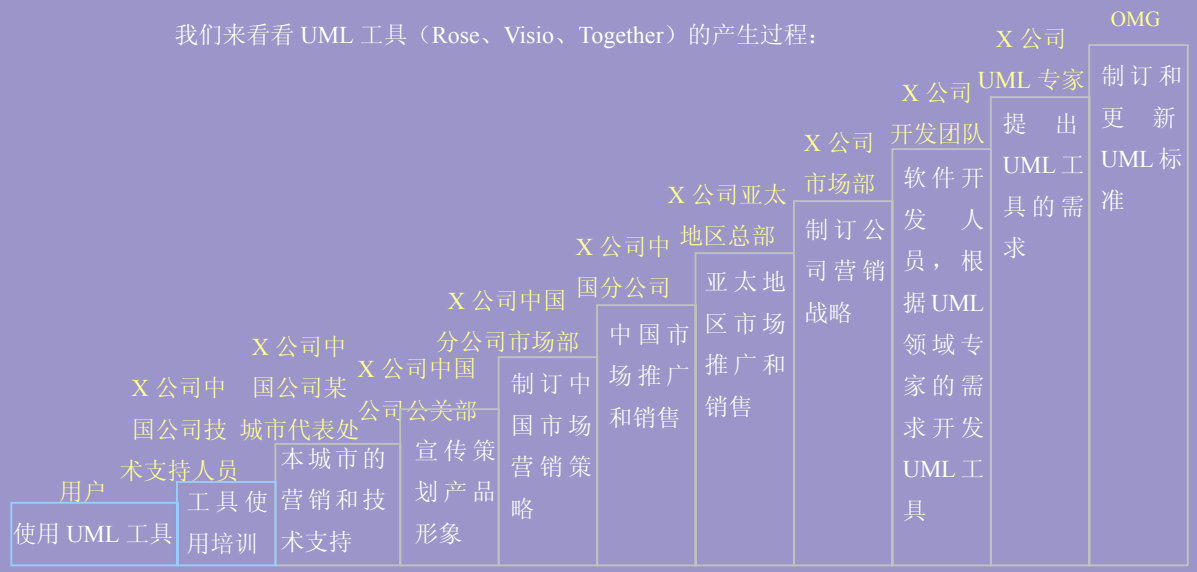
对他们是一点，对我们是一切

UMLChina，the REAL thing >>

微软开发出了占领导地位的产品是 Windows 和 Visual C++，但大家看得最多的<<Windows 程序设计>>（Charles Petzold），《Visual C++技术内幕》（David Kruglinski）等书籍的作者并非开发相应产品的微软员工。这就是现代分工。

建模领域也是如此。关于 UML，最畅销的书籍“UML Distilled”的作者不是 UML 的最初发起人，也不是 OMG，而是 Martin Fowler。

我们来看看 UML 工具（Rose、Visio、Together）的产生过程：



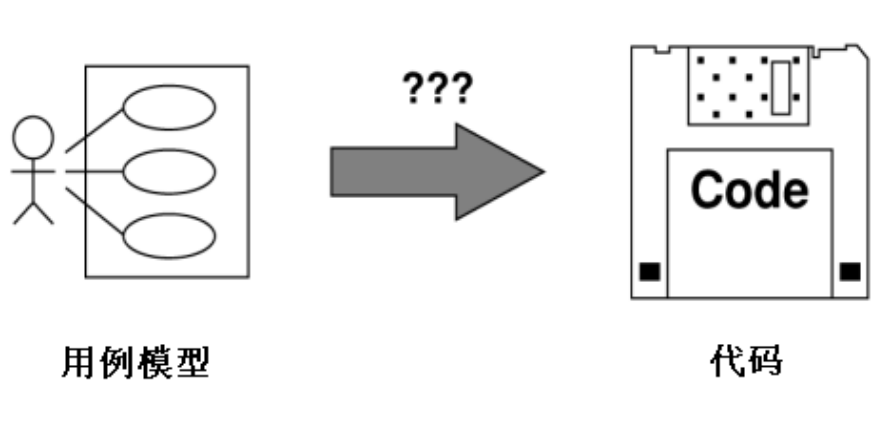
- 工具厂商的产品是工具，不是工具培训，更不是开发技术培训。
- 公司名字是个筐，什么都可以往里装。实际上，目前很多著名公司在中国的主要任务并不是研发，而是销售产品——真正在这个领域做研发的人在大洋彼岸。每个高质量的工具背后都会有资深的对象技术和方法学专家（如 Ivar Jacobson、Peter Coad…）支撑着，如果开发团队在购买产品之后，能得到这些大师们相应的开发方法“培训”，相信会受益无穷。但这样的机会太少了。

UMLChina 培训是 UMLChina 真正用心去打造的“产品”，不是其他产品的附属赠品——内容组织，教材和现有的各种类似培训并不相同，而且会随着技术变化，不断改进。我们小心翼翼地希望 UMLChina 培训能够在这一点点领域里尽量做好，真的能够帮助开发团队（或个人）正确地使用对象技术，开发出更好的软件。

向开发出优秀方法和工具的公司致敬，我们愿意充当斯达舒、吗叮啉，帮助您消化适合您胃口的优秀软件开发技术。

UMLCHINA 培训

北京公开课：6/29-6/30



详情请垂询：UMLCHINA 北京培训中心

联系人：傅光远

电话：66130237，13910698898

EMAIL：bjcourse@umlchina.com

设计模式之路(一)

Jun Qian 著

查看评论

一、道听途说

第一次听说设计模式是我刚进入公司不久， 当时有个模块需要设计，我去找同组的一个老印[1]讨论，老印叫山迪是个 Senior[2]．山迪在黑板上画了个图，然后说就用这个模式吧。我傻乎乎的问了一句，什么是模式？山迪怪怪的斜了我一眼。说道， “你不知道模式?” 我就很诚实的点点头。心里暗骂“有什么大不了的，我计算机学了十几年，什么没见过”。事后， 也没把什么模式当回事。后来，比我稍早进公司的方博士也常在我面前提起模式如何如何好， 同在一个部门的另外两个同事(一个老美，一个老印)也都是模式迷。我当时是部门的教育协会主席，负责组织每两周一次的研讨会．就和他们几个商量了一下，决定搞一个模式的研讨会．由每个人每次介绍 1， 2 个模式．然后大家自由讨论．也就是从那时起，我开始学习设计模式。

二、百思不解

第一件事当然是从网上购买四人帮[3]的设计模式一书．这本书非常非常的有名，好象有人说过是计算机科学界近二十年来最重要的一本书．书很快就到了．印刷得很漂亮．我也迫不及待地翻开来看， 看了不到两页前言，就开始打磕睡,看不下去了，感觉不知所云，模式倒底是什么东西？第二天一早碰到一位同事，我就问她，听说过设计模式 没有？她说听说过呀，她也买了那本书，不过也没有看懂．我心里好像舒服了点，看来这本书对美国人一样不太好读。很快我们的第一次研讨会到了,老印同事给我们简单介绍了一下模式的来源,以及他们参加 OOPLSA[4] 大会里有关模式的一些讲座．这才知道模式来源与一位建筑学家亚力山大－克理斯托夫 ， 他在他的经典著作<<模式语言>>里提到: “每一个模式描述了一个在我们的环境里屡次发生的问题，和解决这个问题的关键的设计法案，这样一来，你就可以应用同样的设计法案解决类似的问题，而不必每次都从头设计一遍”[5]．虽然他是指建筑学里的设计问题，但他的思想被计算机界的人借鉴到软件设计．简单地说模式就是设计重用．代码的重用，大家都好理解，每个程序员都知道”Copy & Paste”自己或是别人的代码。设计模式呢？就是”Copy & Paste”自己或是别人的设计。设计模式这本书， 就是总结了一些软件设计中最为常用的设计问题和对应的解决办法（还是

有点抽象 ☺). 不过既然了解了其出处，理解它的用处也就不远了.

三、初学咋练

很快在我做的项目里就碰到了我平生的第一个模式， Factory 模式. 这个项目是与加拿大一家公司合作的项目，他们的程序是用 COBOL 在 MainFrame 的机器上面写的。而我们的程序是 C++在 Unix 机器上运行的。我们小组的项目就是中间的一个转接层程序（现在流行叫 Multi-tier☺)，通过 TCP/IP 接受加拿大来的信息，然后根据不同的信息类型,送到不同的处理单元去处理,最后通过 TCP/IP 返回给加拿大.Factory 模式的关键是对 abstract class 的应用，实际上是面向对象设计中的高级应用. 简单地说就是通过抽象类(abstract class)和虚拟函数(virtual function)分离制造者(Factory) 和调用者(Application). 这样调用者就不必考虑被调用者的不同性. 大家可以参考设计模式一书，和附一来理解抽象类和虚拟函数的妙用。

四、从头学起

在学习模式的过程中，我发现最根本的障碍还是我对面向对象的设计理解不够深入. 因为我们上学的时候还是 C 的天下，所以虽然以后用到 C++，也是 C 的思路，设计上还是过程分解的办法. 比如，什么时候用 Interface? 什么时候用 abstract Class? 应该用继承还是用引用？虽然天天都在用 Class，实际上并没有用对. 所以我决定先补面向对象的课. 常规的讲继承和封装的教材已经不能满足我的要求. 我需要一本简明易懂， 多讲应用实例，强掉基本设计结构的书. 庆幸得是，我找到了一本非常好的书 “对象的艺术”(The Art of Objects). 不知国内有没有翻译本. 这本书从最容易混淆的几个面向对象的概念讲起，接着介绍一些面向对象设计中最基本的模式. 这些模式可以说是面向对象设计的砖头， 比”四人帮”里的模式要简单， 也好理解， 同时作者用了很多 Java 里面的实际例子. 理解了这些概念和基本模式之后，回过头来再看设计模式， 就感觉不一样了。更重要的是通过这本书，我终于理解到面向对象设计的精髓是以**数据/实体**为中心，而不是传统软件设计的以**过程**为中心. 同一时期，我也买了几本别的讲设计模式和面向对象设计的书,进行到这一步，感觉到真正理解和掌握设计模式已经不远了，现在需要的是实践. 纸上谈兵是永远不能成为专家的.

五、牛刀小试

在我转到一个新的小组不久， 机会来了. 就接手一个小的项目，我可以负责独立的设计和实现。项目不大，是要做一个简单的帐单插入图象管理系统(Bill Insert Table Management)， 用户可以通过浏览器输入，修改，

删除记录等. 数据库是用 Oracle， 服务器是 Weblogic。因为这是一个相对独立，而且数据量不大的模块， 我决定用一个简单的三层设计， JSP 做用户层， Java Servlet 做中间层通过 JDBC 与数据库通讯。我用了两天，参考了"J2EE 设计模式"一书， 就搭好了程序的基本框架(附例二)。我用到了设计模式一书中的 Factory 模式，Command 模式 ， "J2EE 设计模式" 中的 Front Controller， DAO(Database Access Object)， Data Object 模式。效果也是非常的好，Bug 少多了，因为整个结构非常清晰，每个 Class 各司其职，很容易调试. 同时再也没有以前写程序那种失控的感觉，因为应用模式的结果，使得各个 Class 之间相对独立，再没有以前那种修改一个 Bug，就要牵连一大串 Class 的现象了. 这个项目一周就做完了，再经过一周测试组的测试就进入运行了. 做完这个项目之后，信心增加了很多，体会到模式非常有用，也非常好用. 不久以后我几乎是同样的设计结构完成了另外一个用户管理的模块设计和实现. 只不过这次只用了两天时间!

附例一:

```
/*************************************************************************/
// AbstractMessage.h: The Abstract Message class (Product)
class AbstractMessage
{
public:

    AbstractMessage();
    virtual void Process() = 0;
    virtual ~AbstractMessage();
};

/*************************************************************************/
```

```
// TNSwitchMsg.h: The Concrete Message class (ConcreteProduct)

class TNSwitchMsg: public AbstractMessage

{

  public:

    TNSwitchMsg(AAISCRISMessage&);

    void Process();

};



/*************************************************************************/

// MessageFactory.C: The MessageFactory class (Creator)

// Parameterized factory methods


const string   MessageFactory::ms_strng_NOA002S   = "NOA002S";

const string   MessageFactory::ms_strng_NOM261S   = "NOM261S";


AbstractMessage*

MessageFactory::Create(const string&     rc_strng_InMessageType,

                       AAISCRISMessage&  r_ASCRSMssg_InRequest)

{

    AbstractMessage*  p_AbstrctMssg_CreatedMessage = NULL;
```

```
    if(MessageFactory::ms_strng_NOA002S == rc_strng_InMessageType)
    {
        p_AbstrctMssg_CreatedMessage = new TNSwitchMsg(r_ASCRSMssg_InRequest);
        return (p_AbstrctMssg_CreatedMessage);
    }

    if(MessageFactory::ms_strng_NOM261S == rc_strng_InMessageType)
    {
        p_AbstrctMssg_CreatedMessage = new NRetrieveMsg(r_ASCRSMssg_InRequest);
        return (p_AbstrctMssg_CreatedMessage);
    }
    return (NULL);
}

/*************************************************************************/
//CrisSocket.h, CrisSocket.C: The Implementation of Application
class CrisSocket:public MSILSocket
{
  public:
    void Process();
};
```

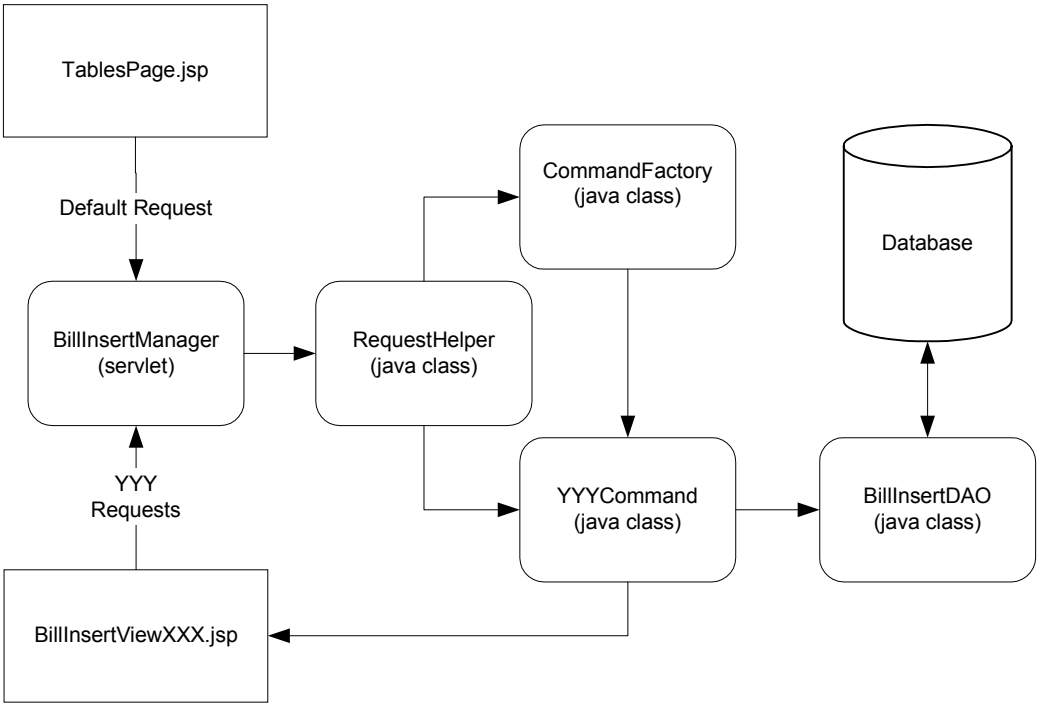
```
void CrisSocket::Process()
{
    // Get the Message from the Client
    // Create appropriate message type
    AbstractMessage*  p_AbstrctMssg_Message = NULL;
    if(NULL == (p_AbstrctMssg_Message
                = MessageFactory::Create(CRSMssg_FormattedMessage->GetMessageType(),
                    *CRSMssg_FormattedMessage))) {
      log_msg(CONSOLE,
          "+++ Error - Invalid message type received +++");
      return;
    }
    // Process the message
    p_AbstrctMssg_Message->Process();

   // Send the result back to the Client
}
```

附例二：

TablesPage.jsp 是图像管理的入口. BillInsertManager 是前端控制器用来接受 HTTP 请求. RequestHelper 是调配者用来分发请求. 命令模式(Command Pattern)用来针对不同的请求产生不同的命令(Command). 具体的命令处理商务逻辑并负责调用相应的 JSP 来显示结果. 命令通过 DAO(Database Access

Object) 模式与数据库通信. 这样我们的模块每增加一个功能(比如删除，修改图像), 只需要增加一个命令和相应的显示 JSP 而已.



BillInsetXXXJSP	YYYrequest
BillInsertCreateList.jsp	Create, List
BillInsertProfileList.jsp	View,Create
BillInsertCreateProfile.jsp	Insert
BillInsertViewProfile.jsp	Delete, Modify
BillInsertModifyProfile.jsp	Update
BillInsertDeleteProfile.jsp	DeleteConfirm
BillInsertMessage.jsp	

YYYCommand	Destination JSP
DefaultCommand	BillInsertCreateProfile.jsp
ListCommand	BillInsertProfileList.jsp
CreateCommand	BillInsertCreateProfile.jsp
InsertCommand	BillInsertMessage.jsp
ViewCommand	BillInsertViewProfile.jsp
DeleteCommand	BillInsertDeleteProfile.jsp
DeleteConfirmCommand	BillInsertMessage.jsp
ModifyCommand	BillInsertModifyProfile.jsp
UpdateCommand	BillInsertMessage.jsp

1. 老印：在美国中国人常称印度人为老印，墨西哥人为老墨，黑人为老黑．以次累推…中性词，无褒无贬

2. Senior：美国公司中，一般对工作年头长或是水平高的工程师的尊称

3. 四人帮：因设计模式是四个作者合著，所以常被人尊称做”Gang of Four”

4. OOPLSA：Conference of “Object-Oriented Programming， Systems， Languages and Applications”的简称，是 ACM 关于 OO 的最重要的年会。

5 “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem， in such a way that you can use this solution a million times over， without ever doing it the same way twice”

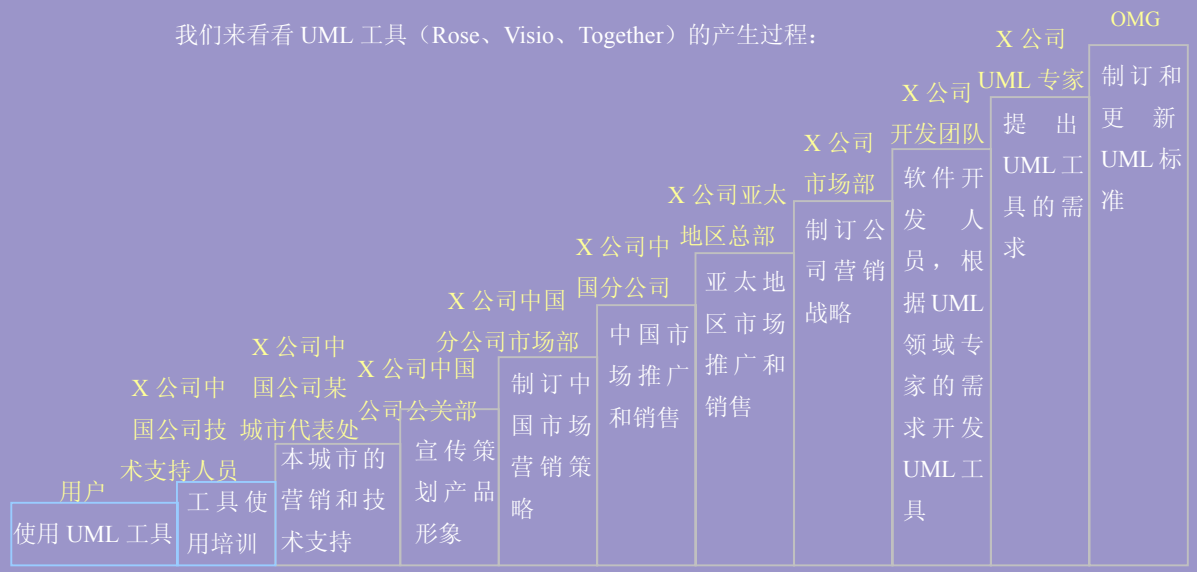
对他们是一点，对我们是一切

UMLChina，the REAL thing >>

微软开发出了占领导地位的产品是 Windows 和 Visual C++，但大家看得最多的<<Windows 程序设计>>（Charles Petzold），《Visual C++技术内幕》（David Kruglinski）等书籍的作者并非开发相应产品的微软员工。这就是现代分工。

建模领域也是如此。关于 UML，最畅销的书籍“UML Distilled”的作者不是 UML 的最初发起人，也不是 OMG，而是 Martin Fowler。

我们来看看 UML 工具（Rose、Visio、Together）的产生过程：



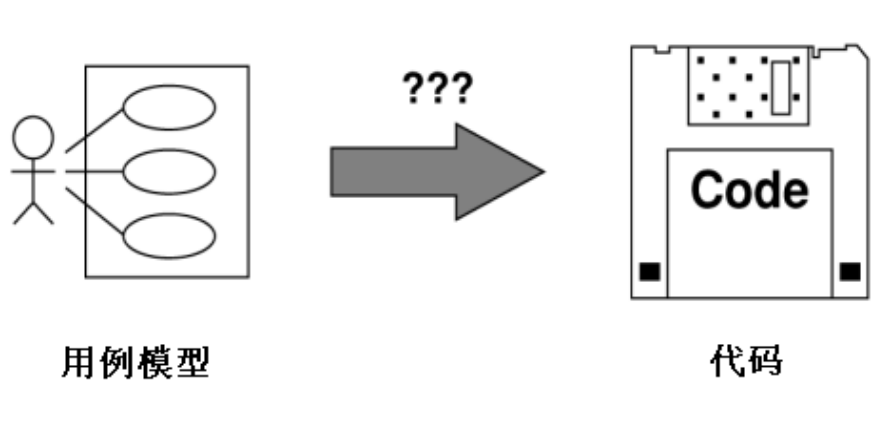
- 工具厂商的产品是工具，不是工具培训，更不是开发技术培训。
- 公司名字是个筐，什么都可以往里装。实际上，目前很多著名公司在中国的主要任务并不是研发，而是销售产品——真正在这个领域做研发的人在大洋彼岸。每个高质量的工具背后都会有资深的对象技术和方法学专家（如 Ivar Jacobson、Peter Coad…）支撑着，如果开发团队在购买产品之后，能得到这些大师们相应的开发方法“培训”，相信会受益无穷。但这样的机会太少了。

UMLChina 培训是 UMLChina 真正用心去打造的“产品”，不是其他产品的附属赠品——内容组织，教材和现有的各种类似培训并不相同，而且会随着技术变化，不断改进。我们小心翼翼地希望 UMLChina 培训能够在这一点点领域里尽量做好，真的能够帮助开发团队（或个人）正确地使用对象技术，开发出更好的软件。

向开发出优秀方法和工具的公司致敬，我们愿意充当斯达舒、吗叮啉，帮助您消化适合您胃口的优秀软件开发技术。

UMLCHINA 培训

北京公开课：6/29-6/30



详情请垂询：UMLCHINA 北京培训中心

联系人：傅光远

电话：66130237，13910698898

EMAIL：bjcourse@umlchina.com

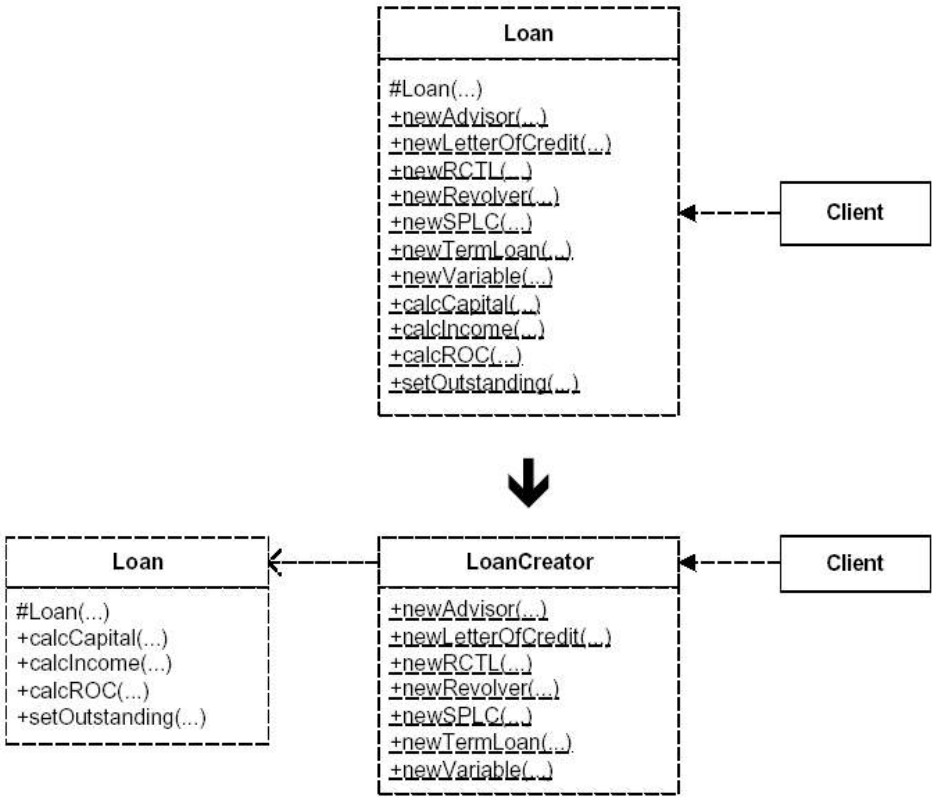
提取创建类

Joshua Kerievsky 著，透明 编译

查看评论

一个类中有太多的创建方法，以至于无法从代码中看清这个类主要的责任。

将一组相互关联的类中的创建方法搬到一个创建类中去。



动机

从根上来说，这个重构其实就是*Extract Class*[Fowler]，只不过是对类的创建方法进行的。一个类中存在创建方法，这很正常；但是随着创建方法越来越多，类本身的主要责任——它存在的主要目的——可能就变得越来越模糊，甚至被对象创建的逻辑给遮蔽住了。如果情况真是这样，就应该对这个类做一些调整，将创建方法移动到一个专门的创建类（Creation Class）中去，还这个类以本来面目。

创建类和抽象工厂（Abstract Factory）[GoF]都可以用来创建一族中的任意一种对象，从这一点来说，它们很相似。但是，它们之间的差异也很大，如下表所示：

	创建类	抽象工厂
运行时可替换	是	否
可以实例化一族产品	是	是
容易支持新产品	是	否
接口与实现分离	可能	是
使用静态（static）方法实现	通常是	否
作为单件（Singleton）实现	否	经常是

一般来说，如果你希望只用一个类来创建对象，那么创建类会比较合适，因为不必在运行时更换对象，直接用创建类的静态方法来创建即可。尽管创建类和抽象工厂都经常用于创建一组相关的对象，但是这两者之间有一个最大的不同之处：你不会在运行时更换创建类，因为你不需要改变产品对象的家族。创建类通常被实现为一个包含静态方法的类，每个静态方法生成并返回一个对象实例。

沟通	重复	简化
当对象的创建在一个类的公开接口中占据了一大半的时候，这个类就已经不能很好地表述自己的主要用途了。创建一个特定的类专门用于创建对象实例，这样两个类都可以更好地表达自己的用途，更好地与使用者沟通。	在这个重构中，重复不是问题。	当创建对象的责任过多地与这个类的主要责任搅在一起的时候，这个类就无法简单。将创建代码提取到一个独立的创建类中，这样可以简化这个类。

过程

1. 识别出创建方法泛滥的那个类（我们把它叫做“A”）。
2. 新定义一个类，作为你的创建类。它的作用是从一组相关的类中创建不同的对象，根据其具体作用给它命名。
3. 将A类中所有的创建方法都移到这个新类中，设置好所有的访问控制。
4. 修改所有获取新对象的调用者，让它们从新的创建类那里获取对象。

范例

虽然我使用的范例代码跟Martin Fowler的不同，但是由于我天生懒惰，所以我还是想反复使用前面的例子。那么，如果你不介意的话，我们还是来看这个傻傻的Loan的例子吧，上面的代码草图画的就是这个例子。请假想下面的范例代码是有相应的测试代码的——我没有在正文中把测试代码写出来，因为这个重构非常简单。

1、我们的Loan类有许多代码都是负责创建Loan对象的：

```
public class Loan {
    private double notional;
    private double outstanding;
    private int rating;
    private Date start;
    private CapitalStrategy capitalStrategy;
    private Date expiry;
    private Date maturity;
    // . . . more instances variables not shown
    protected Loan(double notional, Date start, Date expiry,
        Date maturity, int riskRating, CapitalStrategy strategy) {
            this.notional = notional;
```

```
        this.start = start;
        this.expiry = expiry;
        this.maturity = maturity;
        this.rating = riskRating;
        this.capitalStrategy = strategy;
    }
public double calcCapital() {
    return capitalStrategy.calc(this);
}
public void setOutstanding(double newOutstanding) {
    outstanding = newOutstanding;
}
// ... more methods for dealing with the primary responsibilities of a Loan, not shown
public static Loan newAdvisor(double notional, Date start, Date maturity, int rating)
    return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
}
public static Loan newLetterOfCredit(double notional, Date start,
                                     Date maturity, int rating) {
    return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
}
public static Loan newRCTL(double notional, Date start,
                           Date expiry, Date maturity, int rating) {
    return new Loan(notional, start, expiry, maturity, rating, new RCTLCapital());
}
public static Loan newRevolver(double notional, Date start,
                               Date expiry, int rating) {
```

```
        return new Loan(notional, start, expiry, null, rating, new RevolverCapital());
    }
    public static Loan newSPLC(double notional, Date start,
                               Date maturity, int rating) {
        return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
    }
    public static Loan newTermLoan(double notional, Date start,
                                   Date maturity, int rating) {
        return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
    }
    public static Loan newVariableLoan(double notional, Date start,
                                       Date expiry, Date maturity, int rating) {
        return new Loan(notional, start, expiry, maturity, rating, new RCTLCapital());
    }
}
```

2、创建一个名叫LoanCreator的类。之所以起这样的名字，是因为它唯一的作用就是给客户提供一个可以获取Loan实例的地方：

```
public class LoanCreator {
}
```

3、现在，将Loan类中所有的创建方法都移动到LoanCreator中，把LoanCreator和Loan类放在同一个包中，这样它就可以有足够的访问权限来访问Loan类的构造子和实例化Loan对象了。

```
public class LoanCreator {
    public static Loan newAdvisor(double notional, Date start,
    Date maturity, int rating)
    return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
```

```
}
public static Loan newLetterOfCredit(double notional, Date start,
                                     Date maturity, int rating) {
    return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
}
public static Loan newRCTL(double notional, Date start,
                           Date expiry, Date maturity, int rating) {
    return new Loan(notional, start, expiry, maturity, rating, new RCTLCapital());
}
public static Loan newRevolver(double notional, Date start,
                               Date expiry, int rating) {
    return new Loan(notional, start, expiry, null, rating, new RevolverCapital());
}
public static Loan newSPLC(double notional, Date start,
                           Date maturity, int rating) {
    return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
}
public static Loan newTermLoan(double notional, Date start,
                               Date maturity, int rating) {
    return new Loan(notional, start, null, maturity, rating, new TermLoanCapital());
}
public static Loan newVariableLoan(double notional, Date start,
                                   Date expiry, Date maturity, int rating) {
    return new Loan(notional, start, expiry, maturity, rating, new RCTLCapital());
    }
}
```

4、最后，简单地修改一下调用的格式，重构就完成了。把

```
Loan termLoan = Loan.newTermLoan(…)
```

改成

```
Loan termLoan = LoanCreator.newTermLoan(…)
```

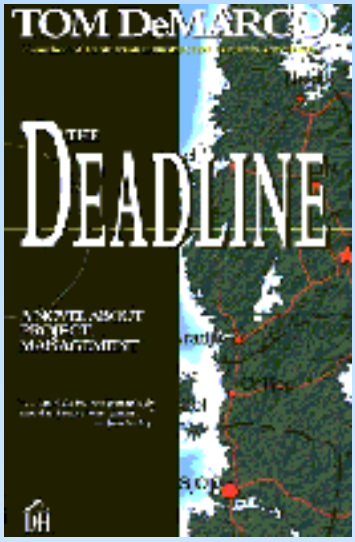
参考资料

[Fowler] Martin Fowler, etc., *Refactoring*, Addison-Wesley, 1999.

[GoF] Eric Gamma, etc., *Design Patterns*, Addison-Wesley, 1995.

征稿

《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！

利用重构改进面向对象设计

Lance Tokuda and Don Batory 著，koun 译

查看评论

摘要[1]

重构是一种保留程序行为的程序转变，能够使得面向对象程序进行设计的自动改进。存在三种设计改进，它们分别是：方案转换、设计模式微结构、特点驱动式趋进。这个研究展示如何利用重构使这三种方式得以自动进行；也给出一个全面的为设计改进的的重构方式的列表，同时分析了所支持的方案转换、设计模式以及特点驱动元模式。更进一步，通过展示重构一些 C++程序，我们评估了重构技术是否能成为主流。我们所用到的程序都曾是被软件工程师手工进行改进的。我们将展示，可以通过一些通用用途的重构实现程序的改进，而达到相同的效果，并且比以前付出少得多的时间和更少的代价。在一个程序中，超过 1 万 4 千行的代码被自动转化，否则的话，它们都需要手工完成。我们用自己的经验来判定在相关产品环境中使用重构技术的转换的好处、限制以及更进一步的研究主题。

1 介绍

在图形用户界面的编辑器发明之前，对于图形用户界面的改进步骤是设计、编码、测试、评估、重新设计。而在图形用户界面编辑器出现之后，这个过程则变成了互动的过程，允许用户设计、评估、重设计屏幕接口，同时能够输出与最后一次设计所对应的可编译的源代码。

我们相信在编辑面向对象类图方面能够出现类似需要。使得编辑一个类图变得如此简单，就像在类之间增加一行代码来表示类的继承关系或者从子类中将一个变量挪到超类中一样。然而，现在的状况是，这样的改变一定要伴随着辛勤的劳动，通过标识出被影响的源程序的行号，手工更新源代码，测试这些改变，修正错误，重新测试程序，直到新错误的风险足够低。

就象图形用户界面的编辑器引发用户界面设计的革命一样，我们相信类图编辑器（在程序的类图上面的改变会自动引发底层源代码的改变）将引起软件设计发展的革命。这个工具所用的技术就是重构－保持行为的程序改

变，它使多层设计[2]的改变这个过程自动化。

我们使用术语“自动”来指对使能条件的检查和源代码的自动处理。使用哪种设计来实现和使用哪种重构并不是自动的，而是需要靠人来选择。于是，人们创建了初始的应用程序的设计，手工选择一系列的重构来改良设计。不是所有的程序上的改变都可以通过重构来自动完成。修正错误和改进算法是一些常见的对程序的改变，但是这些改变却不能进行重构。*重构只关注对一个程序的类结构进行自动的改动*。重构的好处在于减少调试和对改变进行测试的烦闷，要不然，这些都需要手工来执行。

2 重构

名字：

Inherit[*Base*,Derived]

目的：

在两个现存类之间建立超类—子类关系

参数：

Base - 超类名字

Derived - 子类名字

描述：

Inherit[]使 ***Base*** 成为 **Derived** 的超类。

1. ***Base*** 一定不能是 **Derived** 的子类，**Derived** 一定不能拥有超类。
2. **Derived** 的成员变量一定要有一个与 ***Base*** 及 ***Base*** 的父类的成员变量不同的名字。
3. **Derived** 里的重载的成员函数一定要拥有和其所重载函数相同的类型。
4. 如果要创建 ***Base*** 的子类的对象，则其一定要实现所有的纯虚函数。
5. 初始化列表一定不能用来初始化 **Derived** 对象。
6. 在 **Derived** 被实例化后，对于所有被继承的实例变量，如果它的类型是某个类，那么，这个类的创建函数不能有超出这个类创建之外的负面影响。
7. 程序的行为一定不能依靠 **Derived** 的大小和布局.

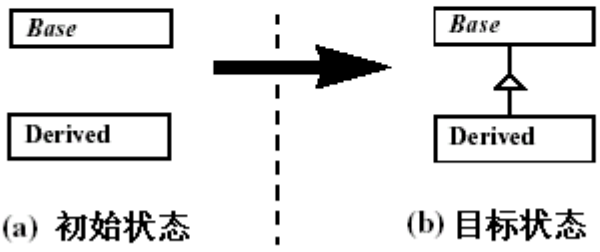


图1：inherit[*Base*, Derived] 转换

重构是一种参数化的保持行为的程序改进，它自动更新程序的设计及相应的代码。重构典型地表现为一个非常小的、对程序代码有直接影响的程序转换，对其所处理的代码的一个直接的影响（但不是那种随意的改变）。比如说一个继承的例子 **inherit[*Base*, Derived]**,它在两个以前没有什么联系的类（***Base*** 和 **Derived**）之间建立了超类－子类的关联。从面向对象类图的观点来看，**inherit** 仅仅只是在 ***Base*** 和 **Derived** 类之间增加了继承关系，但是它仍然检查与最后改变相应的程序代码在转换的时候使能条件是否成立。重构可以更加精确的定义为（a）目的.（b）参数（c）描述（d）使能条件（e）初始状态（f）目标状态。 以这种方式定义的 **inherit[*Base*, Derived]**参见图 1。同手工做相似的处理相比，实施重构更为高级，它允许工程师对现有代码的设计进行改进而不必理会繁琐的代码级的细节问题。在图 2 中列出所有本篇文章所用到的类图符号。

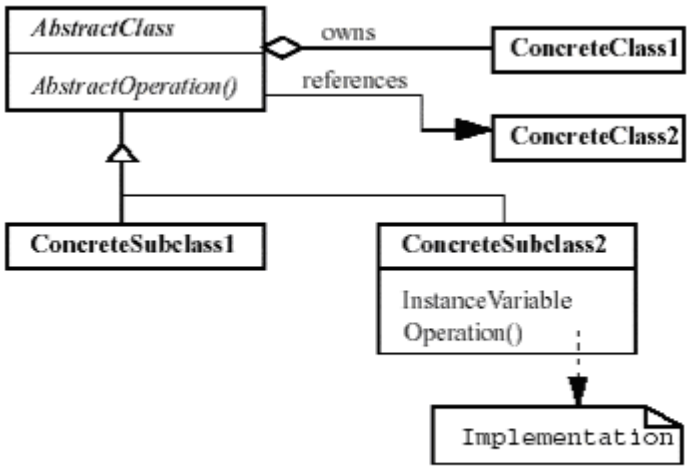


图2:符号

抛开以前的事实，我们采用 Banerjee 和 Kim 曾建议的数据库方案的改进步骤。他们提出一组不变式来描述面向对象数据库方案。Opdyke 建议了一组相类似的六个不变式保留重构的行为。Opdyke 的重构存在着例证，显示了这些不变式能够建立重构的环境。但是他不能够证明保持这些不变式能够保留程序的行为。

举例说明，Opdyke 的第一个不变式是：每个类（如类 A）必须有一个单独的超类，同时这个超类不能是它(A)的一个子类。当重构的行为破坏这个不变式的时候，使能条件的加入来保证这个不变式恒定。对于 **inherit** 的使能条件在图 1 中列出。**inherit** 的第一个使能条件保持了 Opdyke 的第一个不变式。

于是，我们和其他人在重构中用到的工程方法就是定义一组必要的使能条件来保持程序的行为。由于语言的复杂性，这些条件或许不够充分。事实上，我们的经验显示 Opdyke 的不变式对于 C++语言是不充分的。在 5.2 章中，我们指出计数例子－需要额外的条件来保证行为不变。同样，即使我们没有发觉，或许仍然需要额外的条件。直到主流语言形式上定义的语法变得更加简单，并且其行为保留的证据更加实际，我们所采取的方法步骤才占优

势。

然而还是存在使能条件相关的一些困难：如一些条件不能够被自动检查到。举例，图 1 的条件 7 说：“程序的行为一定不能依靠 **Derived** 布局或大小”。是说，如果一个程序开发确定 **Derived** 如何在内存中分布（比如，X 变量的位置相对于对象开始的字节偏移数为 5），这种情况下是没有一个明显的方式来自动检测这个用法。另一个例子，不可能检验重构的“逻辑正确”性。举例，两个有着继承关系的类本身没有任何联系之处（如将“狗”作为“建筑”的子类），但是重构仍然能够“正确”执行，即使这里的逻辑关系没有意义。在本篇文章和任何重构工具中，重构的逻辑正确性被假设是正确的。在使能条件不能被自动检查的条件下，重构工具一定要在进一步处理之前要求人的参与来校验这些条件（或者断言这些条件适合设计）。

在我们的研究中用到的自动重构放在表1中。那些以非斜体列出的是Bannejee、Kim、Opdyke所建议的，而我们自己新加入的重构方法使用斜体来区分。在下面的章节中，我们以图解方式阐明这些重构。要想获得重构的详细信息（比如代码改变，使能条件等），我们建议下面的文章【Ban87,Opd92,Tok99】。

方案重构	
add_variable	move_method_across_ object_boundary
create_variable_accessor	delegate_method_across_ object_boundary
create_method_accessor	extract_code_as_method
rename_variable	*declare_abstract_method*
remove_variable	*structure_to_pointer*
push_down_variable	**C++重构**
pull_up_variable	*procedure_to_method*
move_variable_across_ object_boundary	*structure_to_class*
create_class	**模式重构**
rename_class	*add_factory_method*
remove_class	*create_iterator*
inherit	*composite*
uninherit	*decorator*
substitute	*procedure_to_command*
rename_method	*procedure_ptr_to_command*
remove_method	*singleton*
push_down_method	
pull_up_method	

表1：面向对象重构

Banerjee和Kim[Ban87]的描述	表1中的重构方法
增加一个实例变量	add_variable
删除一个实例变量	remove_variable
改变实例变量的名字	rename_variable
改变一个实例变量的作用域	pull_up_variable 和 push_down_variable
去掉一个实例变量的聚合属性[a]	structure_to_pointer
去掉一个现有的方法	remove_method
改变一个方法的名字	rename_method
将类S作为类C的超类	inherit
取消S作为C的超类	uninherit
增加一个新类	create_class
去除一个现有的类	remove_class
改变类的名字	rename_class

a. 一个类A拥有实例变量B，于是A拥有聚合属性，指明A拥有B. B不能独立于A创建，并且B不能够通过另一个对象的聚合连接来进行访问。

表2：Banerjee和Kim描述的重构

3 设计改进的自动模式

3.1 方案转换

面向对象的数据库管理系统(OODBMS)就象面向对象应用里面的类图。因此，面向对象的数据库管理系统方案的转换在面向对象软件的改进中可以并行处理。比如，把一个实例变量在继承结构中向上挪（图 3）。重构中的 **pull_up_variable** 支持这种转换，即将子类的变量挪到超类中去。Banerjee 和 Kim 描述的 19 个面向对象的数据库方案转换中，我们对其中的 12 个[3]实现了自动重构，这些转换在表 2 中列出。

另外 4 个没有在[Ban87]列出，但仍然是有用的方案转换，我们把它们放在表 3 中。**move** 和 **delegate** 重构描述了一些间接方式来访问被移动的方式和变量。它们的不同是明显的：**delegate** 通过引入委托方法使得原有的类保持完整。**move** 重构去掉原有类的方法，用直接的 foo()调用方式来代替间接 foo()的调用 var->foo()，变量 var 标

识foo()被挪向的那个类/对象。

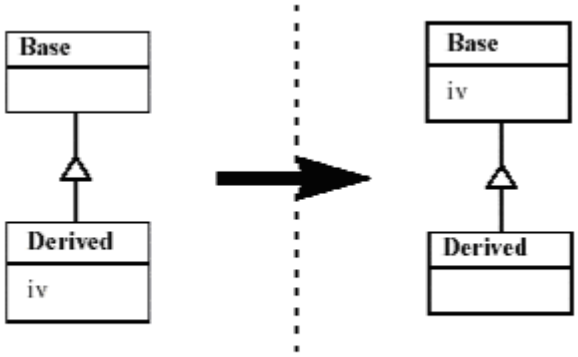


图3：使用pull_up_variable将实例变量"iv"从Derived挪到Base中

描述	表1中的重构
在聚合连接中移动变量	move_variable_across_object_boundary（图4）
在聚合连接中移动方法	move_method_across_object_boundary
在聚合连接中使用委托移动一个方法	delegate_method_across_object_boundary
将一个类的以来关系从C移到C的超类S上	substitute（图5）

表3. 另外的方案重构方式

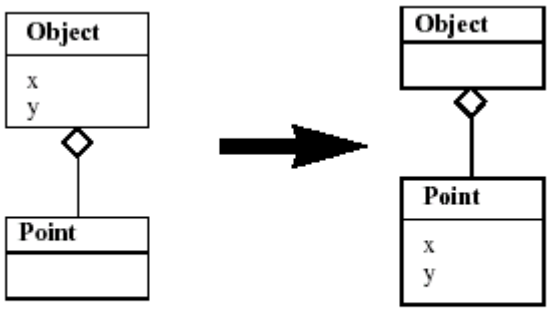


图4：使用move_variable_across_object_boundary移动变量x和y

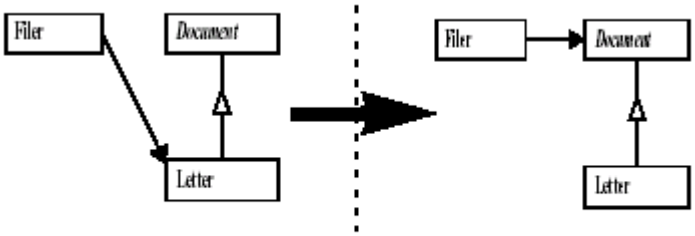


图5：使用替代把Filer对Letter的引用改为对Document的引用

Substitute 重构通过将其子类的引用替代为超类的引用来归纳它们的关联（图 5）。这个重构必须是被高度约束的，因为它经常不能工作。比如说，子类的使用者可以使用那些不在超类中定义、而只存在于子类中的函数，这时，**substitute** 只有在一些特殊条件下，如父类设计为子类接口的时候，它才能够被采用。这种情况下，不应该存取所有子类特有的方法或成员变量。接口可能只存在一些纯虚函数（这意味着所有子类必须提供它们的实现）或者它只提供一些被所有子类共享的方法或方法模板。我们将在 3.2.2 和 3.3.1 里讨论一个 **substitue** 的用法。

3.2 设计模式微结构

设计模式总结出许多通用的面向对象设计问题的专业解决方式：创建兼容的部件、适配一个类到不同的接口上、子类化和子类型、分开第三方接口等等。模式在许多应用里被发掘出来，如 Smalltalck 的 Collections 里、ET++、MacApp，以及 InterViews 里面。随着数据库方案的转换，重构显示为对一些模式的直接实现：如

模式	描述	例子
Command	将请求封装在对象里面，允许使用不同的请求参数化客户端，并支持 Undo 操作。**procedure_to_command** 重构将过程转换为 command 类。	【Tok99b】
Factory Method	工厂方式定义了创建对象的接口，允许它的子类决定对哪个类进行实例化。**add_factory_method** 重构在一个现有类上增加工厂方法。	【Tok95】
Singleton	确保类在全局范围内只存在一个实例。**singleton** 重构将一个空类转换为 singleton	【Tok99c】

我们直接支持三种额外的模式，将其作为重构的方式：

模式	描述
Composite	组合将各个对象组成树状结构来代替部分/整体的结构。**composite** 重构将一个类转换为组合类。
Decorator	Decorator 模式将一些额外的功能动态添加到一个对象上。**Decorator** 重构将一个类转换为 Decorator 类。
Iterator	Iterator 提供顺序访问聚集对象中元素，同时不用考虑其内部实现的一种方式。**create_iterator** 重构产生一个 Iterator 类。

设计模式不但在初始软件设计的时候很有用，就是在软件周期的维护阶段，它们也经常被采用。比如，最初设计者并不清楚某种模式或者系统可能的需求，后来觉得需要 加入模式使其变得灵活。然而，模式可能会将系统引向一种复杂的、间接的，但在软件最初版本并不适合的境况。许多模式都可以被视为以改进设计为目的的自动程序转换。下面两个模式的例子可以说明：

模式	描述	例子
Abstract Factory	抽象工厂模式允许在不指定具体子类的条件下，提供创建一组相关或独立对象的方式。	【Tok95】
Vistor	访问者模式允许在不改变所操作元素类的条件下定义一个新的操作。	【Rob97】

至少还有 5 个其他的模式可以看作程序的转换。

模式	描述
Adapter	适配器允许那些接口不同的类在一起工作。
Bridge	桥模式将接口与实现进行分离，使得两个层次结构都能够独立发展
Builder	组建模式将一个复杂对象从它的表示形式分离出来，这样相同的创建步骤能够创建不同的表示。
Strategy	策略模式允许将算法与使用它们的使用者分开进行独立封装。
Template Method	模板方法允许子类重新定义算法的特定步骤而不改变算法结构。

在所有的情形下，我们可以将重构应用到简单的设计上来用于【Gam95】中创建原型例子的设计。下面章节显示前两个模式如何自动完成。

3.2.1 Adapter(适配器)

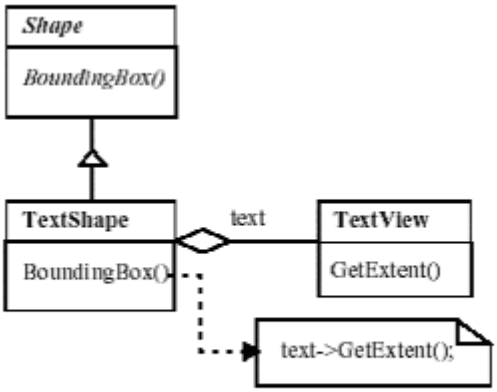


图6：TextShape适配了TextView的接口

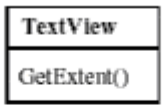


图7：未适配的TextView类

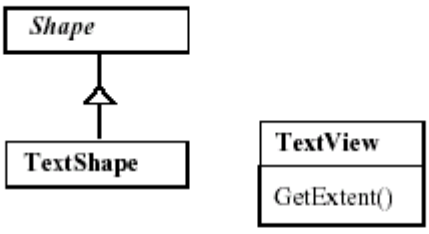


图8：适配器类被创建

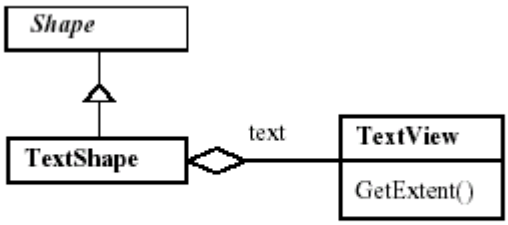


图 9：适配器的适配实例变量

适配器允许不同接口的类在一起工作，在【Gam95】(图 6)中，例子显示 **TextShape** 类适配类 **TextView** 的 GetExtent()方法来实现 BoundingBox()。适配器可以从原有类 **TextView** 中以下面 5 个步骤进行创建：

1. 使用 **create_class** 创建类 **TextShape** 和 ***Shape***。
2. 使用 **inherit**(图 8)使 **TextShape** 作为 ***Shape*** 的子类。
3. 使用 **add_variable**（图 9）向 **TextShape** 增加 text 变量.
4. 使用 **create_method_accessor** 创建 BoundingBox()方法，它调用 text->GetExtend()。不但在方法创建上，**create_method_accessor** 也调用 BoundingBox()来替换 text->GetExtend()的表达。
5. 使用 **declare_abstract_method**(图 6)声明 ***Shape*** 里的 BoundingBox()方法。

3.2.2 Bridge

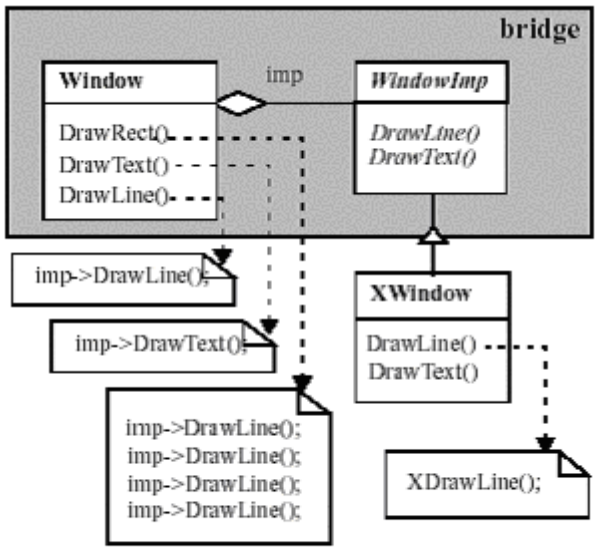


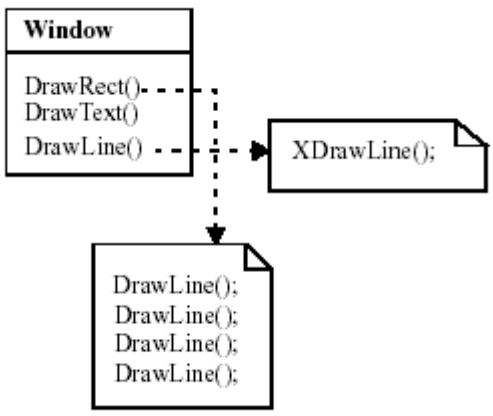
图 10：设计模式"桥"的例子

桥将实现的与抽象进行松耦合。在【Gam96】（图 10）的例子中，Window 抽象和 WindowImp 的实现被放置在不同的实现层次中。所有在 window 子类的操作都在 windowImp 中实现。要支持更多的窗口系统，只要对 WindowImp 层次进行扩展即可。我们将 Window 和 WindowImp 的关系称为桥，因为它使抽象与实现的桥接，使得两方都能够独立发展。

在类似窗口系统的例子的设计中，重构能够安装一个桥的设计。图 11 描述了 X-Windows 的实际。这个系统可以通过重构，以下面 6 个步骤使用桥模式：

1. 使用 **create_class** 创建类 **Xwindow** 和 ***WindowImp***。
2. 利用 **inherit** 使 ***WindowImp*** 成为 **Xwindow** 的超类（图 12）。
3. 使用 **add_variable** 在 **Window** 类中增添 imp 成员变量。（图 13）
4. 使用 **delegate_method_across_object_boundary** 重构将方法 DrawLine()和 DrawText()移动到类 **Xwindow** 中（图 14）。这个重构将 DrawLine()和 DrawText()方法的函数体移到 **Xwindow** 中，同时在 **Window** 类中保留一个代理的方式（如 **Xwindow** 方法可以通过 imp 实例变量进行访问）。
5. 使用 **declare_abstract_method** 在 ***WindowImp*** 中声明 DrawLine()和 DrawText();
6. 使用 **substitute** 把实例变量 imp 的类型从 **Xwindow** 转换成 ***WindowImp***(图 10)。因为 ***WindowImp*** 定义了一些其子类(如 **Xwindow**)必须实现的接口，所以这个转换是可能的。

桥设计模式使用对象组合来提供灵活性。对象组合也在 Builder 和 Strategy 模式中出现。使用继承还是组合的优缺点在【Gam95,pp.18-20】有所讨论。重构允许设计者安全地从静态可检查的设计向使用对象组合的动态定义设计进行迁移。



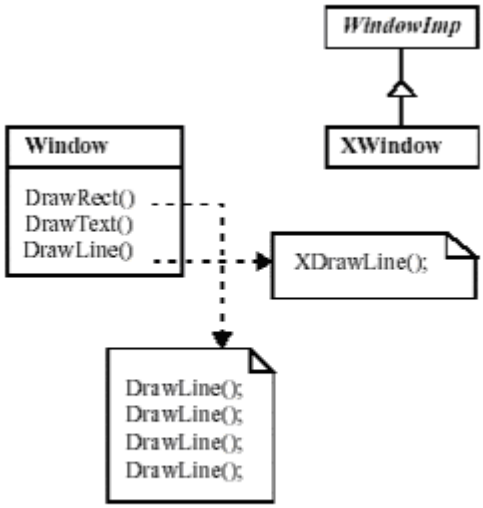


图12：创建的实现者类

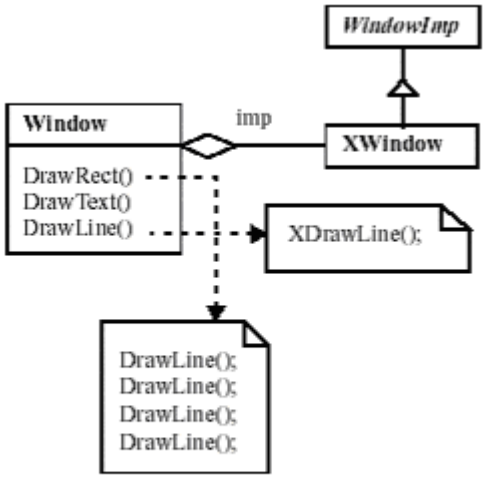


图13：在Window类中增加实现者实例变量

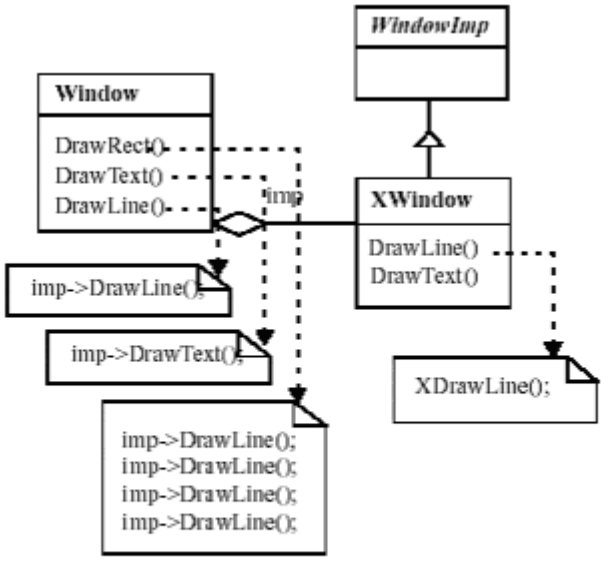


图 14：Window系统特有的方法挪到XWindow类中

3.2.3 设计模式中重构的角色

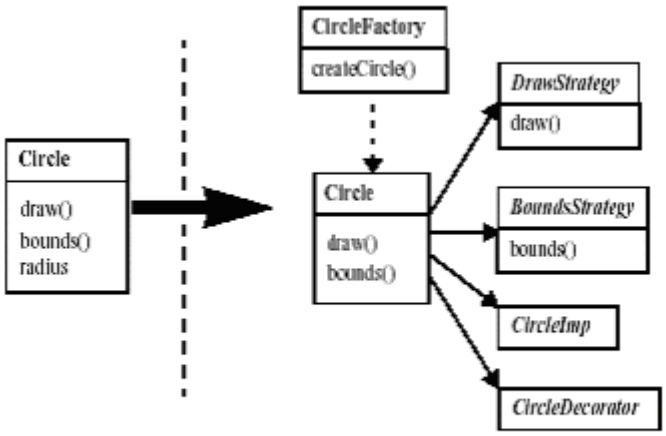


图 15：对设计模式的过分使用

Gamma et.al. 提到设计模式中的一个通常现象就是过分设计：“模式是有耗费的（间接性、复杂性），因此设计应该够灵活就行，而不是越灵活越好”。图 15 中显示的是【Gam96】的例子，一个过分设计者不是简单的创建一个 **Circle** 类，而是增加一个 **Circle** 工厂，并且对类中的每个方法使用策略模式，同时对 **Circle** 采用桥模式、Decorator 模式。这个设计变得十分复杂和间接，而实际上并不需要。从单单一个 **Circle** 类迁移到如图 15 所示的微体系可视为一个转移。这种转移事实上可以通过重构来自动进行 [4]。于是，不是过分设计，而只是从一个简单的 Class 类开始，按照需要逐步添加工厂方法、策略、桥以及装饰模式。重构可以重新构建现有的实现，使得它

们更加灵活、动态、可重用。然而，它们影响算法的能力有限制。象职责链和备忘录模式需要设计模式所用到的地方的算法。于是这些模式被看作程序设计的基础，因为没有可利用的重构改进的步骤。重构允许设计者在一个新设计之初专注于基本的模式，通过重构，再添加当前或者后来需要的但不是代价太大的新模式。

3.3 特点分析（Hot-Spot Analysis）

特点驱动趋进标识一个应用与另一个应用的方案有什么不同之处。这些不同的方便称作特点(*hot-spots*)。当一个数据的特点被发现，就引入了抽象类。当一个函数的特点被发现，就引入了特别的方法和类。

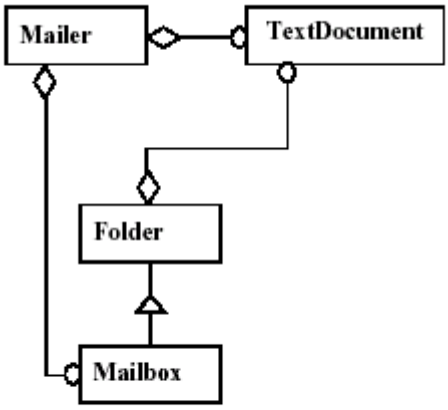


图 16：邮件系统的初始状态

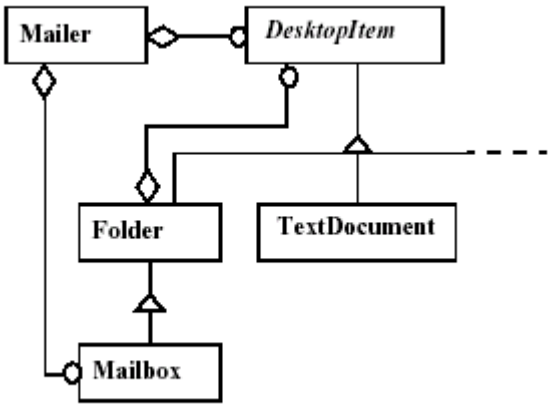


图 17：邮件系统的最终状态

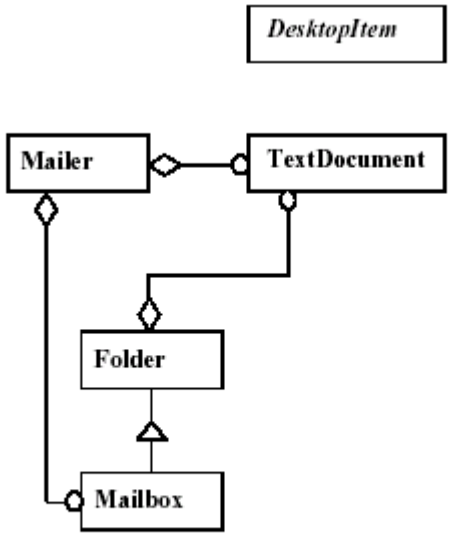


图 18：创建的空的 TextDocument 类

3.3.1 数据特点

当应用之间的实例变量看起来不同的时候，Pree 规定了抽象类的创建。多次演示证明重构有能力创建抽象类【Opd93,Tok95,Rob97】。作为一个例子，Pree 和 Sikora 提供了一个邮件系统以供学习【Pre95】。图 16 显示了软件设计的初始状态。在这个系统中，**Folder** 不能嵌套，只有 **TextDocument** 才能被邮寄。建议的设计在图 17 中所示。在不断的改进设计之下，**Folders** 能够嵌套，任何 ***DesktopItem*** 的子类都能够被邮寄。重构能够自动完成多数这样的改变：

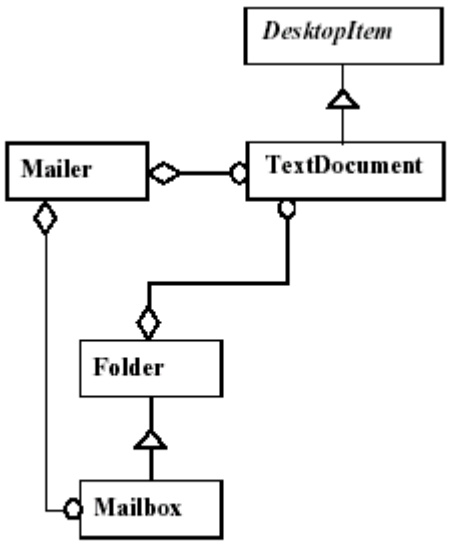


图 19：TextDocument 继承了 *DesktopItem*

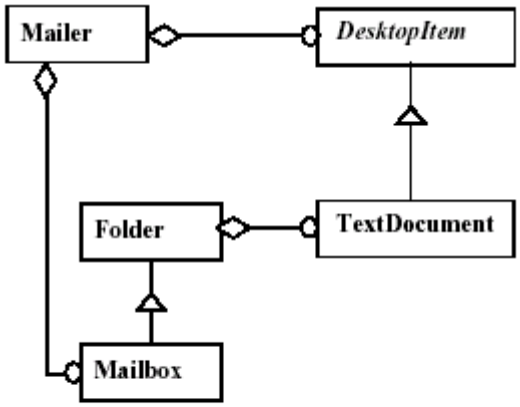


图20：Mailer 和TextDocument到 *DesktopItem* 存在关联

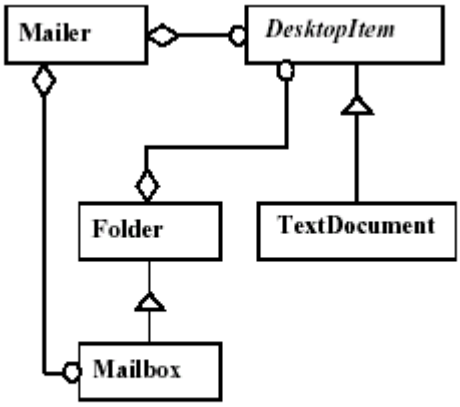


图21：Folder 能够包含任何 *DesktopItem*

1. 使用 **create_class** 重构来创建一个 ***DesktopItem*** 类（图 18）。使用 **declare_abstract_method** 在 ***DesktopItem*** 里为 **TextDocument** 的每个公共方法创建抽象方法；***DesktopItem*** 用来作为所有能够邮寄对象的接口。

2. 使用 **inherit** 使 ***DesktopItem*** 作为 **TextDocument** 的超类。

3. 使用 **substitute** 将 **Mailer** 和 **TextDocument** 之间的联系转换成为 **Mailer** 和 ***DesktopItem*** 之间的联系。（图 20）

4. 使用 **substitute** 将 **Folder** 和 **TextDocumment** 之间的联系转换成为 **Folder** 和 ***DesktopItem*** 之间的联系（图 21）。**Folder** 现在能够包含任何 ***DesktopItem***。

5. 这一步不能自动完成：在 **Folder** 的实现上增加对 ***DestopItem*** 抽象方法的实现。这些方法包括完成邮寄 **Folder** 对象的过程、允许 **Folder** 对象看作是 ***DesktopItem*** 对象。

6. 使用 **inherit** 使 **Folder** 作为 ***DesktopItem*** 的子类（图 17）。**Folder** 能够包含一个 ***DesktopItem***，现在也能包含另一个 **Folder**,同时也能够象其他 ***DesktopItem*** 一样能够被邮寄。

随着设计的改进，***DesktopItem*** 提供了一个超类，使其能够邮寄其他类型的对象。虽然有些改变不能够自动进行，但大多数还是能够自动完成。

3.3.2 功能特点（Functional Hot-Spots）

在不同的功能性之下，基于模板和回调的解决方式是提供其所需要的行为。一个模板方法为各种行为提供了框架。一个回调方法能够被模板方法所调用，并且能够被剪裁修饰以提供不同的行为。图 22 是一个模板方法和回调方法定义在同一个类中的例子。不同的 **T** 的子类能够重载回调方法 M2()，M2()和 M1()有着不同的功能（图 23）。Pree 为模板和回调方法标识出 7 个元模式：统一元模式，1：1 连接元模式，1：N 关联元模式，1：1 嵌套关联元模式，1：N 嵌套关联元模式，1：1 嵌套统一元模式，1：N 嵌套统一元模式(Pre94)。重构自动在设计改进中引入元模式。重构造成的模式之间的转换在图 24[5] 中显示。作为一个例子，我们展示如何使用重构表达前两个转换。

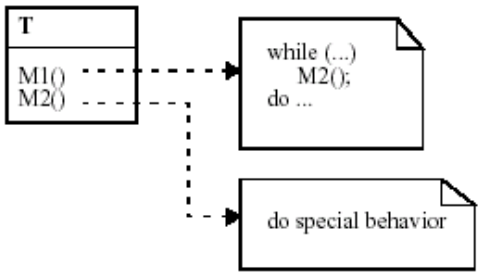


图22：在同一个类中的模板和回调方法

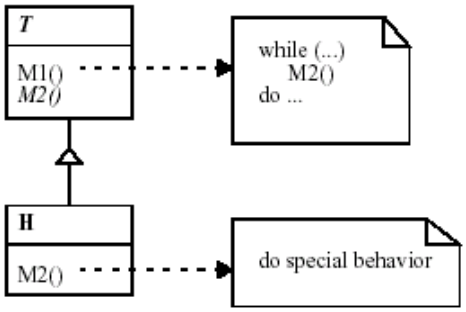


图23：类H中 M2() 是个重载函数

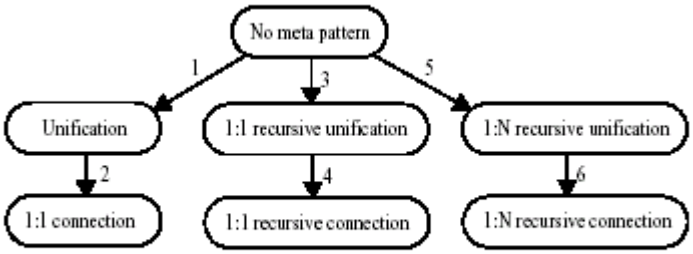


图24：被重构启用的特点元模式的转换

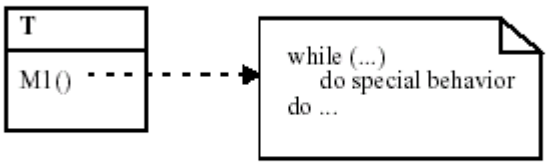


图25：方法 M1() 调用为不同的应用调用一个特别的行为处理

在统一聚合体中，模板和回调方式存在于相同的类中（图 22）。通过在子类中重载回调方法，模板的行为被改变（图 23）。一个没有模板或者回调方法的设计可以使用统一的元模式进行转换（图 24 转换 1）。要看到这种可能性，来看一下图 25 中的类图，**T** 有方法 M1()，这个方法调用一些特殊的行为。一个回调方法能够在重构之下通过一个步骤来加入：

1. 使用 **extract_code_as_method** 创建一个回调方法 M2() 执行特殊的行为。**extrace_code_as_method** 通过调用一个新的方法替代了一个程序块的功能。

在新的微体系结构中，模板方法 M1()包含了公共的行为，而特殊的行为在回调方法 M2()中来执行。在扩展设计中，**T** 的子类重载了 M2()方法，为 M1()提供了可选的行为。可以通过以下三步来增加这个扩展结构。

1. 使用 **create_class** 创建类 **H**。
2. 使用 **inherit** 使 **T** 成为 **H** 的超类。
3. 使用 **push_down_method** 将 M2()的实现移到 **H** 里去（图 23）。这个重构过程保留了 **T** 原先的接口，同时引入了一个被 **H** 重载的抽象方法 M2()。

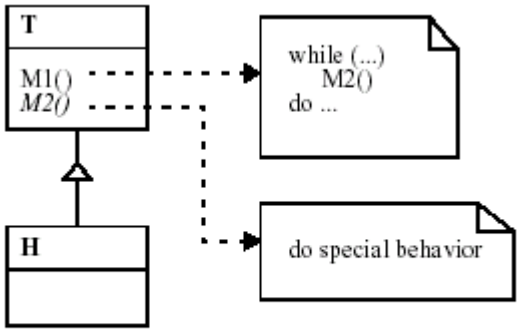


图26：创建了回调类

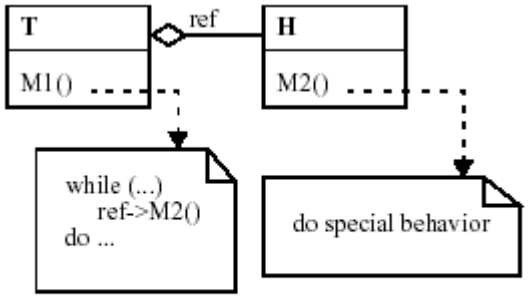


图27：1：1关联

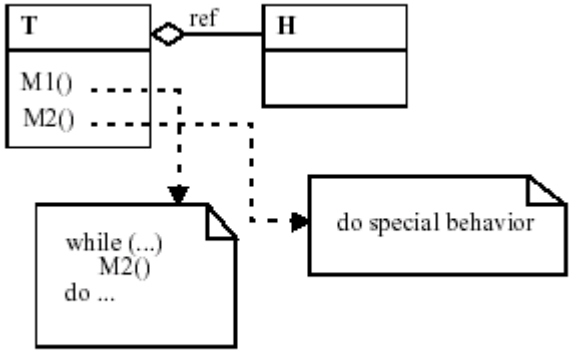


图28：创建了到H对象的关联

作为第二个例子，我们概括了从统一元模式到 1：1 关联元模式的转换（在图 24 中的转换 2）。注意到 1：1 关联元模式保留着模板类的回调方式（图 27）。通过给模板类赋予不同的回调对象，可以在运行时刻改变对象的行为。1：1 关联元模式使用统一元模式（图 22）作为起点，能够通过 3 个步骤来自动实现。

1. 使用 **create_class** 创建类 **H**。
2. 使用 **add_variable** 给 **T** 增建一个类型为 **H** 的变量 ref.
3. 使用 **move_method_across_object_boundary**(图 27)将 M2()方法移到类 **H** 中。

模板方法 M1()的行为可以通过指向拥有不同 M2（）实现的回调类对象来动态改变。其他的在图 24 中的转换有相同的描述。

3.3.3 在特点分析中重构的角色

特点驱动式趋进提供为改进数据和功能管理的设计了全面的方法。Pree 提到“7 个聚合元模式在方案中重复出现”。于是，我们期望存在使用元模式来改进设计的需要。其他的元模式现在是手工操作的。需要检查一些条件来保证模式可以安全加入，受影响的源代码被标出，改变的部分被记录下来，系统要经过测试来检查错误，错误被改变并且系统需要重新测试。需要进行重新测试一直到所期望的错误足够少。

这一章建议了许多模式，这些模式可看作是从简单设计开始的改进。重构在这个过程中，使设计者从自由设计简单方案到根据特点增加需要的模式这一过程变得自动化。

4 改进应用

我们选择了 SEMATECH 的 CIM Works 和 CMU 的 Andrew 用户界面系统作为改进应用的例子。之所以选择这两个例子是因为它们存在不同历史的版本，存在着它们的设计改变过情况。下面的特点使我们的研究独一无二。

设计改进的复制。设计从统一应用的两个版本提取出来。老版本的设计作为最初的的状态，新一点的版本作为目标状态。我们的目的是判断是否可以通过一系列的重构，使设计的初始状态完成到目的状态的转换。并且将这个过程从设计者手工处理转换成为自动过程。相应的，通过自动处理和手工改变的比较，能够提供给我们一个指示：那就是，我们怎样尽可能地使用重构。

非限制功能的应用。改进大型应用来测试重构的伸缩性。在小的应用上（比如只有一千行代码）的一些思想用在大的应用上（超过 10 万行代码）可能会遇到彻底的失败。

主流面向对象语言。选择 C++作为目标语言来试验。它是最广泛使用的，向后兼容 C 语言的一种面向对象编程语言，它可以使用第三方编译器和工具，具有合法的系统兼容性。使用 C++的好处是许多 C++的语法适用于现在日益受欢迎的 Java 语言。

4.1 改进 CIM Works

计算机集成制造（CIM）方案是工业领域最初为书写半导体制造的执行系统而标准化的面向对象方案。CIM Works 是一个窗口应用，来显示和测试 SEMATECH CIM 的方案规范（McG97）.

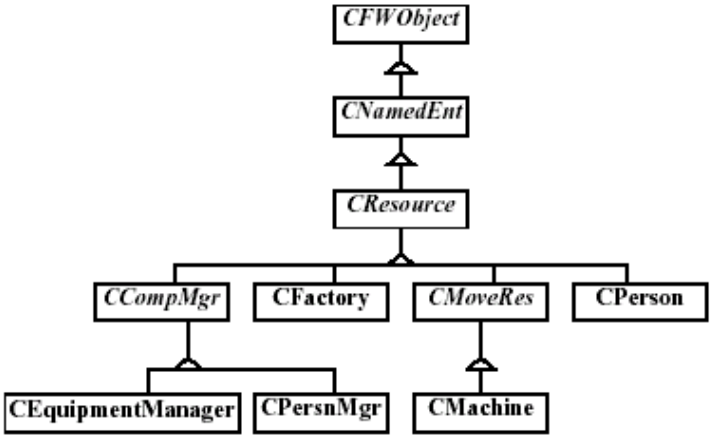


图29：版本2的设计

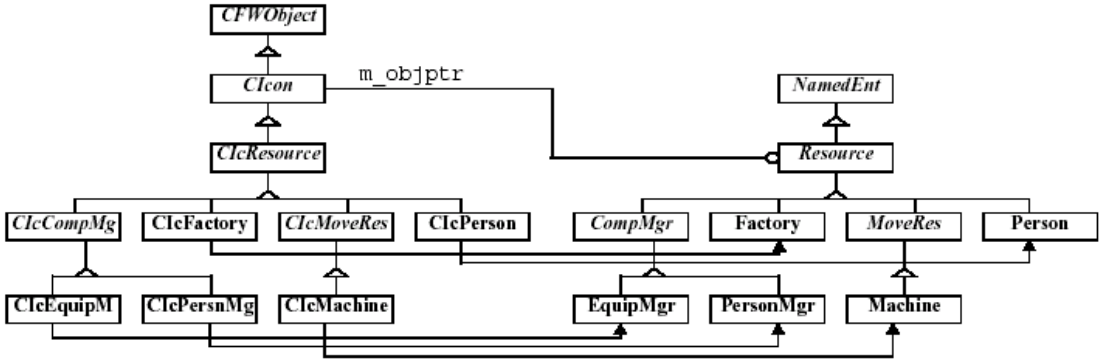


图30：版本4的设计

CIM Works 的主要设计的改变发生在版本 2 和版本 4 中。在版本 2 的设计中（图 29 显示），将数据的存储与显示在同一个对象中表示。比如，**CEquipmentManager** 包含增加和删除部件的方法，这些方法和其他方法一样被组织管理为 GUI 菜单。版本 4 设计中（图 30）将数据的图形表示和组织分为两个类结构。这样做的好处是，在版本 4 中可以使用模型－视图－控制器类图方式自由地创建各种视图，而这些视图使用相同的数据内容。

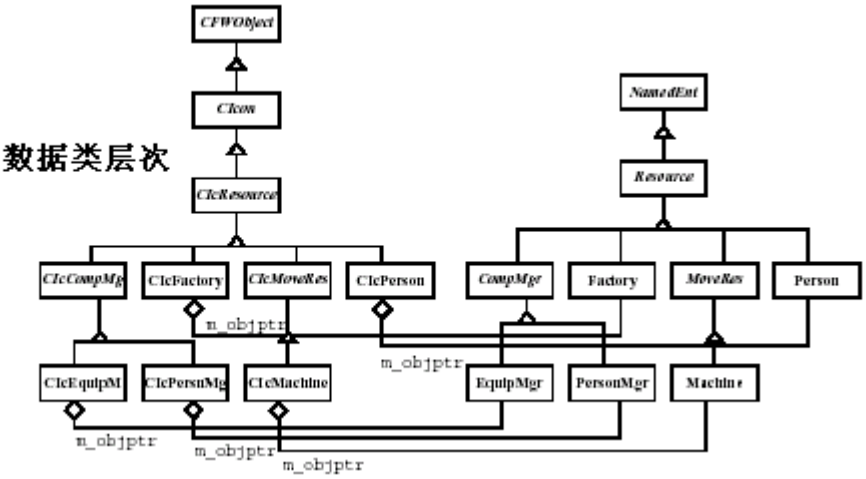


图32：创建抽象数据类层次

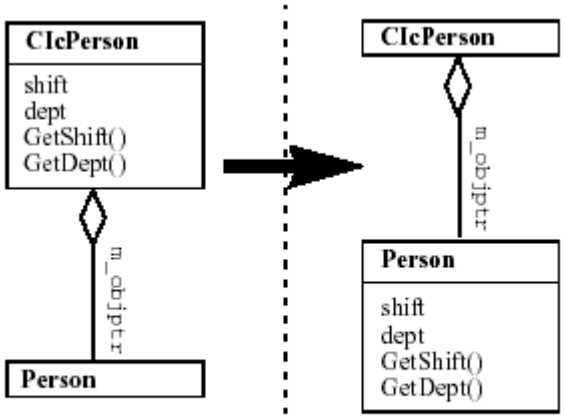


图33：将实例变量和方法移到数据类中去

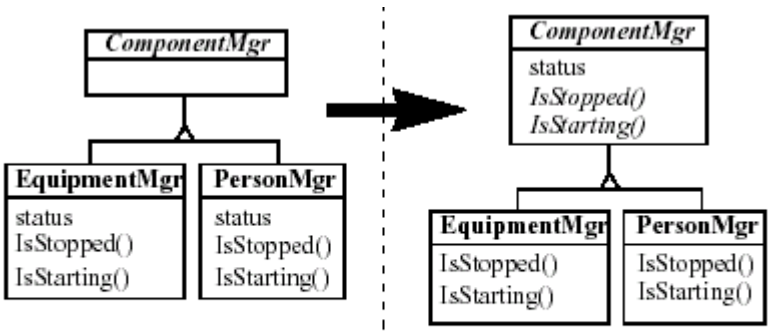


图34：将实例变量和方法的声明移到抽象类中去

版本 2 大约是 1 万 1 千行代码。这次从版本 2 到版本 4 的改进可以通过 9 个步骤来完成，每一步通过实施一序列的基本重构来实现。

1. 使用 **rename class** 重命名原层次中的类，将此层次进行分割（原类保留对象的图形用户部分，同时相应的数据部分在步骤 2、4 中进行封装）

2. 使用 **create_class** 创建实际数据类 **Factory**、**Person**、**EquipManager** 等。

3. 使用 **add_variable** 在实际 GUI 类中添加 m_objptr 实例变量。m_objptr 是相应的数据类类型变量（图 31）。

4. 使用 **create_class** 创建虚数据类 **Resource**、**CompManager**、**MovementResource** 等。

5. 使用 **inherit** 在抽象数据类和实际数据类之间建立联系（图 32）。

6. 使用 **move_variable_across_object_boundary** 将非 GUI 实例变量和方法从 GUI 类中移到数据类中。通过 m_objptr 实例变量来存取数据（图 33）。

7. 使用 **pull_up_variable** 和 **declare_abstract_method**（图 34）将公共实例变量和方法从数据类层次中提出。

8. 将 m_objptr 的类型从一个结构转换成为结构指针。

9. 在抽象类中声明 GUI 对象和数据对象之间的引用。对数据对象的引用使用虚方式（图 30）[6]。

这些步骤通过 81 个重构来完成，对 CIM Works 的 486 行源代码进行了改变。

4．2 Andrew 用户界面系统的改进

CMU 的 Andrew 用户界面系统（AUIS）是一组允许用户创建、使用和邮寄文档、邮寄包含格式化文本与嵌入对象的一组集成工具[Mor86]。要研究的两个版本是 6.3 与 8.0。版本 6.3 使用 C 语言完成，版本 8.0 使用 C++完成；版本 6.3 将动作存储到函数指针里，而 8.0 支持和建议为不同的动作来创建一系列的子类（与设计模式中的命令模式[7]相似）。90 个类将近 800 个动作被影响。这个转换通过下列 5 步完成。

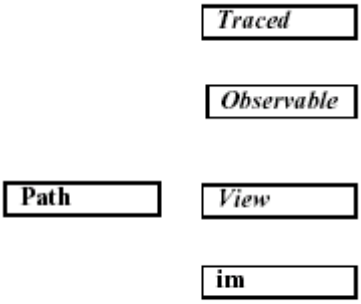


图35：将结构转换成类

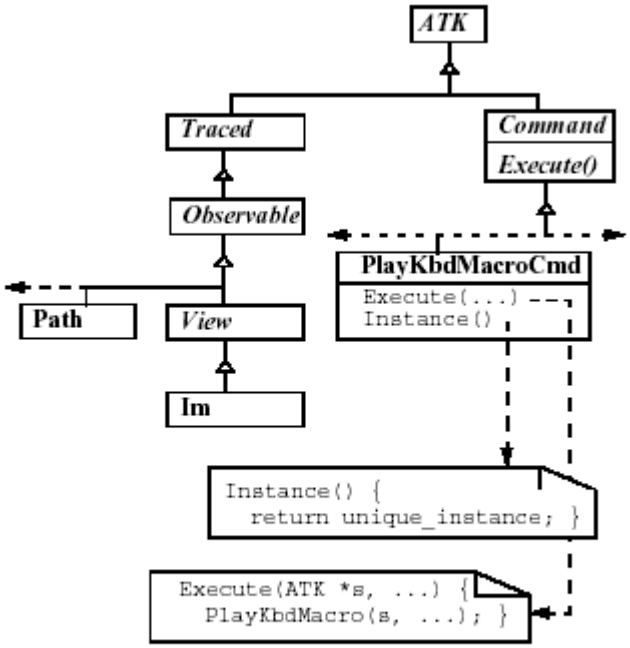


图36：类 Im 和 Command 的微结构

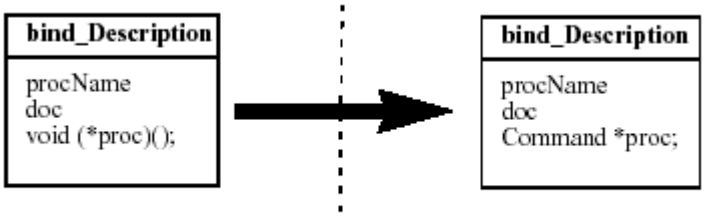


图37：将过程指针转化成Command 指针

1. 使用 **structure_to_class** 将 6.3 版本中 C 定义的结构转换成 C++的类（图 35）。

2. 使用 **create_class** 创建 **ATK** 和 **Command** 抽象类。

3. 在 **ATK** 和其他类之间使用 **inherit** 建立继承关系。

4. 使用 **procedure_to_command** 将每个动作编程派生的命令类。图 36 显示了从 PlayKbdMacro()到 **Command** 子类的转换。新创建的类 **PlayKbdMacroCmd** 包含 Execute()方法，它调用 PlayKbdMacro()。同时它包含一个 Instance()方法，它返回一个独一无二的类的实例。使用 Instance()而不是 new 来创建对象保证了指向 **PlayKbdMacroCmd** 的对象的唯一性。

5. 使用 **procedure_ptr_to_command** 将函数指针转换成命令。在这一步中，函数指针使用的数据类型被转换成为 **Command** 对象指针，函数调用被转换成为 Execute()方法，函数指针的赋值变成 Instance()调用。图 37 显示了 **bind_Description** 结构的转换，proc 实例变量被转换成 **Command** 指针。

在此使用了 800 次重构，有 1 万 4 千行代码被改变[8]。

5 反省和得到的教训

我们的试验在评估重构技术时提供了很多学习经验。在下列的章节中，我们列举出一些在重构中学到的有关重构的优点、局限、以及重构技术取得成功需要解决的问题等方面的重要教训。更多的讨论见[TOK99b].

5.1 重构的好处

自动改动设计。我们的研究得到的最重要的结果是建立能够自动处理的重构过程，来进行上千行代码的自动改进。对通过手工进行这些改变与通过重构工具自动进行处理的结果进行比较是一件很吸引人的事情。我们估计 CIM Works 的那些改变需要我们手工花上两天来实现和调试，但是，通过重构的帮助，这只需要 2 个小时。我们估计 AUIS 的改变需要两个星期来实现和调试，但是采用重构的帮助，我们现在仅仅需要 1 天时间。

减少测试。一个好的重构实现能够减少测试和设计所需要的效果。当重构保留行为时，只有手工编码的那部分代码需要测试。

简单设计。重构减少过分复杂设计的需要。Gamma et.al.指出通常设计模式的缺陷是过分设计：“模式的使用是有代价的（间接性、复杂性），因此，设计应该是满足灵活性的需要即可，而不是越灵活越好”。如果在设计的时候考虑太多的超前特性可能使系统有更多的错误，同时静态类型的检查变得更少[9]。重构有能力使用多种方式

来扩展设计。它鼓励设计者为任务随手创建简单设计，然后使用重构扩展设计，增加新的功能。

确认帮助。使能条件检查能够检查代码级实现和所需设计改变之间的冲突。举例，程序员可能将一个实例变量从基类中移到派生类中，但是忘了在基类对象中对此变量的存取作改变。使能条件检查将检测这个错误。重构能够检查一个很长系列的改变造成的错误结果，而这个改变可能会造成手工执行“取消操作”的费时费力的后果。

轻松创新。重构允许设计者拿新的设计来试验。在方案改进和设计模式手工编码到应用的今天，很显然，此过程自动化能够使得设计者更多关注在设计中发现新思路，而不至于将精力放在编码和调试上。

5.2 重构的局限性

通过对大的应用程序进行作实验，发现了一些在小的、概念化、已经被验证的程序中以前没有发现的局限性。我们来讨论了一下我们最重要的对改变将来研究这些要面对的问题的观察。

预处理信息。我们的 C++程序转换工具不能够处理预处理信息，因为预处理信息不是 C++语言的一部分。我们的试验程序在改进之前是是预先处理的，预处理信息不能够被恢复。同时我们相信工作区是整个程序的重点，我们通常不能够处理所有在大的软件程序中发现的问题。

保留的使能条件。即使在保留的使能条件上使用断言，重构仍然是有用的。举例，**inherit** 转换被 Opdke’s 的不变式保留地限制到单个继承系统。当需要支持多个继承系统时，不需要转换这篇文章所描述的应用，或者增加大量的设计模式和特点元模式【Tok99a】.

自动校验使能条件。大多数但不是所有的使能条件都可以被自动校验。对于 **inherit** 重构，前 5 个条件可以被自动检查（图 1）。

Opdyke 识别出一种不能被自动校验的条件：程序的行为一定不能依赖于其对象布局的大小【Opd92】。大小和布局不是这篇文章和【Tok95,Rob97】讨论的范围，然而使用重构时一定要知道这些限制。

保留行为。我们的试验揭示出 Opdyke’s 的不变式不能充分保证程序的行为。C++需要额外的 3 个不变式：

1. **纯虚函数的实现**。如果一个类是实际类（能够被实例化），它不能拥有纯虚函数。
2. **维护聚合对象** [10]。如果一个程序依赖一个对象的聚合属性，那么这属性一定要保留。
3. **非实例化负面影响**。如果重构能够改变一个类的实例化频率和顺序，那么构造函数不能拥有超出对象创建的负面影响。

按照这个发现，我们明确重构保留行为是因为好的工程学的原因，而不是什么数学上的证明。重构设计师负责判定能够使程序保留行为所需的使能条件。这个重要的问题在【Tok99c】中讨论。

5.3 将来的研究

我们工作的重点放在实施基本重构来改进面向对象应用的实用性上。除了所需功能的实现，我们发现需要更进一步研究的三个问题。

确认。我们如何知道重构的程序的正确性？我们如何知道一个程序是正确的？这是一个在任何情况下都难以回答的问题。因为缺少关于正确性的证据和行为保留的证明，我们期望重构工具能够象今天的编译器一样：随着时间的流逝，我们将认识到重构工具可以做到正确的转换，一直到我们有清楚的、可以重现的错误告诉我们事情不是这样的。随着语言变得更加简单（参看 5.4），有更多的希望做更多的事情。

转换的粒度。我们不是试图将所需要的重构的个数减少到最少。在 CIM Works 的例子中，用到的重构个数很多(81 个)，虽然概念上用到的转换的步骤很少(8)。一种减少重构个数的路子是提供更大粒度的重构。在 CIM Works 的例子中，如果重构是挪动多个变量和方法，则重构的次数会显著减少。和 Andrew 例子相似，大约 800 个转换发生在第四步——将动作程序转换成 **Command** 子类。一个更大粒度的象一组程序到 **Command** 转换能够将第四步变成一次转换，使整个转换减少到少于 20 个。重要的是我们的研究结果是开发出一组基本的重构。象设计模式大小的粗粒度重构可能在实际应用中更加便利。

事务型重构。一个相关的问题是什么是事务型重构。一个事务型重构可以分解成为一组基本转换的聚合，对于这些单个的转换过程，虽然不能够保证程序的行为，但作为聚合体，它们的效果却是能够保留程序行为的。**delegate_method_across_object_boundary** 就是一个例子。概念上说来，它和 **move_method_across_object_boundary**(将方法整个从它原先的类中去除)加上 **create_method_accessor**(它在原类中引入一个新的方法，并且在此新方法上加入一些聚合行为)的功能相同。单个的这些重构不能按顺序进行实施。如果原类的客户引用某个方法，那么 **move_method** 重构的使能条件将限制这个方法，使其不能够进行移动。只有我们将这些重构当作一个整体来考虑其实施的效果时，这些重构的结果才是正确的。

我们通过创建一个新的重构来解决这种问题，这就是 **delegate_method**。然而，一种更好的处理方式是定义一些基本的重构和定义一系列原子性的重构序列。

程序组。事务型系统常会遇见如下问题：一些文件被多个程序所引用。当对这样的一个文件进行转换的时候，转换系统需要对所有使用此文件的相关程序的使能条件进行检查。否则的话，或许对于其中的一个程序，这种转换的结果是正确的，却往往会对其他程序造成问题。C++通过编译选项来对不同的处理器进行对某个文件条件编译，同时考虑到转换的正确性，情况会变得更加复杂。

和其他工具进行集成。重构作为和独立于其他工具的处理，在这种情况下，它可以被集成到大多数的主流开发环境中，因为这些环境大多支持对源文件的命令行存取方式。

高级的集成仍然存在可能，让我们来看它和面向对象建模工具，例如 Rational Rose 进行集成，Ractional Rose 就允许对 UML 类图进行重构操作。将重构与源代码控制系统进行集成能够允许把适当的文件从源码库中拿出来进行转换，再重新放回源码库中，同时对此改变添加适当的重构方面的注释。如果如果想要转换受保护的文件将会提醒此用户，同时将重构工作停下来。象和 Microsoft Visual C++这样的 IDE 系统进行集成会允许改变的代码立刻显示在打开的窗口上。

5.4 和 JAVA 的关系

Java 继承了所有的 C++的重构的好处，同时避免了许多它的限制。第一，它没有预处理器，防止了许多妨碍 C++成功特性实现的阻碍。第二，它没有使用 makefile 机制，简化了对单个文件转换的处理。第三，将方法存放在属于它的类所在的文件中，简化了代码放置位置的处理。Java 没有自由悬浮的程序过程和混合的面向对象语言（比如 C++）。因为这些原因，加上其作为 internet 语言日益变得受欢迎，我们相信 Java 会是主流的实施重构技术最好的载体[11]。现在已经有了这方面的工具【Sim95,Bax97,Bat98】。

6 相关工作

Griswold 为结构化程序开发了行为保留的转换方式。这个系统的目的是辅助分解软件，重新结构化其功能。那些使用传统的结构化设计方式编制的软件要再进行重新结构化已经很困难，因为其定义程序的结构的节点同时传送数据和控制信息。控制信息的存在使得重新分配结构图表的子树变得很困难。结果是，许多转换被限制在一个函数或者一块代码的级别上。

面向对象软件设计提供了对程序重新结构化的可能。Bergstein 定义了一小组对象保留的类转换方式，它们能够在类图上进行实施【Joh88】。Lieberherr 在 Demeter 的面向对象软件环境中【Lie91】实现了这些转换。转换的例子包括删除无用的子类和将实例变量在超类和子类之间移动。

Opdyke 提出“重构”这一术语来描述保留行为的重新结构化面向对象软件系统的程序转换。Banerjee 和 Kim 的方案改进【Ban87】、Johnson 和 Foote【Joh88】的设计原则以及 UIUC 选择操作系统的设计轨迹【May89】发展了重构。作为应用重构的例子如创建抽象超类【Opd92】。已经为 C++【Tok95、Sch98a－b，Tok99a】和 Smalltalk【Rob97】实现了重构。Roberts 提供专为 Smalltalk 的程序设计标准工具【Rob97】。一种在 C++软件上使用的标准就是用户可以命名在程序转换过程中引入的实体名。

重构显示如何自动在面向对象软件系统上增加设计模式【Tok95，Rob97，Sch98a-b,Tok99a】。重构也支持增加 Pree 的特点驱动元模式【Tok99a】。

7 结论

面向对象软件设计的改进可以用到的常规模式是：方案转换，设计模式元模式，特点驱动趋进。许多革命性的改变都可以被看作是使用自动的面向对象重构的程序改进。重构优于手工改动是因为它们检查使能条件来确保改变的正确性，同时能够标出改变的源代码，自动编辑。重构允许在类图层次上改进设计同时将代码级别的细节留给工具自动来进行。

如果存在下面的情况，需要进行设计的改进：

- “从一个简单系统转换而来的复杂系统”－Booch
- “简单开始，然后改进” －Bech

我们研究的最初目的是提供一个主流的工具，能够使得编辑类图象使用 GUI 编辑器来改变用户界面一样简单。这篇文章采取三个步骤来向这个目标靠近：

1. 第一，我们实现了一组重构，它们能够自动实施方案转换、设计模式、特点元模式。它们可以减少和消除标出所影响代码的需要，减少手工改动，减少对这些改变的测试。

2. 第二，我们向大家显示出重构是可扩展的，对于大的、现实世界的应用是有用处的。我们能够使用一组普通目的的重构对上千行程序做自动的改变。

3. 第三，虽然我们的试验清楚地显示使用重构工具的好处，但是也揭示出一些在重构技术在超出学术研究之前，需要处理的限制和问题。

只要我们相信它能够保持重构技术，从我们的试验的和在管理 C++预处理信息上的困难看得出，Java 应该是下一门主要语言。

感谢

我们感谢那些参阅者对早期本文的注解。最后的定稿有很大改善。

参考文献

[Ban87] J. Banerjee and W. Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of the ACM SIGMOD Conference*, 1987.

[Bat98] D. Batory et. al. JTS: Tools for Implementing Domain- Specific Languages. In *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

[Bax97] I. Baxter. and C. Pidgeon. Software Change Through

Design Maintenance. In *Proceedings of the International Conference on Software Maintenance '97*, IEEE Press, 1997.

[Ber91] P. Bergstein. Object-Preserving Class Transformations. In *Proceedings of OOPSLA '91*, 1991.

[Ber91] P. Berstein. Object-preserving class transformations. In *Proceedings of OOPSLA '91*, 1991.

[Bud96] F. J. Budinsky et.al., Automatic Code Generation from Design Patterns. In *IBM Systems Journal*, Volume 35, No. 2, 1996.

[Ell90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990.

[Gam93] E. Gamma et. al. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings, ECOOP '93*, pages 406-421, Springer-Verlag, 1993.

[Gam95] E. Gamma et.al. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[Gam96] E. Gamma et. al. *TUTORIAL 29: Design Patterns Applied.* OOPSLA '96 Tutorial, 1996.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.

[Gri91] W. Griswold. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. thesis. University of Washington. August 1991.

[Joh88] R. Johnson and B. Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, pages 22-35, June/July 1988.

[Kra88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. In *Journal of Object-Oriented Programming*, pages 26-49, August 1988.

[Kim96] J. Kim and K. Benner. An Experience Using Design Patterns: Lessons Learned and Tool Support, *Theory and Practice of Object Systems*, Volume 2, No. 1, pages 61-74, 1996.

[Lie91] K. Lieberherr, W. Hursch, and C. Xiao. *Object-Extending Class Transformations.* Technical report, College of Computer Science, Northeastern University, 360 Huntington Ave., Boston, Massachusetts, 1991.

[Flo97] G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Object-Oriented Patterns. In *Proceedings, ECOOP '97*, pages 472-495, Springer-Verlag, 1997.

[Lie91] K. Lieberherr, W. Hursch, and C. Xiao. *Object-extending class transformations.* Technical report, College of Computer Science, Northeastern University, 360 Huntington Ave., Boston, Massachusetts, 1991.

[Lin92] M. Linton. Encapsulating a C++ Library. In *Proceedings of the 1992 USENIX C++ Conferenc*e, pages 57-66, Portland, Oregon, August 1992.

[May89] P. Maydany et.al. A Class Hierarchy for Building Stream-Oriented File Systems. In *Proceedings of ECOOP '89*, Nottingham, UK, July 1989.

[McG97] P. McGuire. Lessons learned in the C++ reference development of the SEMATECH computer-integrated manufacturing (CIM) applications framework. In *SPIE Proceedings*, Volume 2913, pages 326-344, 1997.

[Mor86] J. H. Morris et.al. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 1986.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[Opd93] W. F. Opdyke and R. E. Johnson. Creating Abstract Superclasses by Refactoring. In *ACM 1993 Computer Science Conference*. February 1993.

[Par79] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(2):128-138, March 1979.

[Pre94] W. Pree. Meta Patterns — A Means for Capturing the Essentials of Reusable Object-Oriented Design. In *Proceedings, ECOOP '94*, Springer-Verlag, 1994.

[Pre95] W. Pree and H. Sikora. *Application of Design Patterns in Commercial Domains*. OOPSLA '95 Tutorial 11, Austin, Texas, October 1995.

[Pre92] R. Pressman. *Software Engineering A Practitioner's Approach*, McGraw Hill, 1992.

[Rob97] D. Roberts, J. Brant, R. Johnson. A Refactoring Tool for Smalltalk. In *Theory and Practice of Object Systems*, Vol. 3 Number 4, 1997.

[Sch98a] W. Scherlis. Systematic Change of Data Representation: Program Manipulations and Case Study. In *Proceedings of ESOP'98*, 1998.

[Sch98b] B. Schulz et. al. On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems. In *Proceedings of the 27th TOOLS Conference*, IEEE CS Press, 1998.

[Sim95] C. Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", *NATO Science Committee Conference, 1995*.

[Ste97] S. Stewart. *Roadmap for the Computer-Integrated Manufacturing Application Framework*. NISTIR 5697, June, 1995.

[Tok95] L. Tokuda and D. Batory. Automated Software Evolution via Design Pattern Transformations. In *Proceedings of the 3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995.

[Tok99a] L. Tokuda and D. Batory. Automating Three Modes of Object-Oriented Software Evolution. In *Proceedings, COOTS '99*, 1999.

[Tok99b] L. Tokuda and D. Batory. *Evolving Object-Oriented Designs with Refactorings*. University of Texas, Department of Computer Science, Technical Report TR99-09, March 1999.

[Tok99c] L. Tokuda. *Design Evolution with Refactorings*. Ph.D. thesis, University of Texas, 1999.

Wei88] A. Weinand, E. Gamma, and R. Marty. ET++ -- An Object-Oriented Application Framework in C++. In Object- Oriented Programming Systems, Languages, and Applications Conference, pages 46-57, San Diego, California, September 1988.

[Win96] Pieter van Winsen. *(Re)engineering with Object-Oriented Design Patterns*. Master's Thesis, Utrecht University, INF-SCR- 96-43, November, 1996.

[You79] E. Yourdon and L. Constantine. *Structured Design*. Prentice Hall, 1979.

注释

1. 我们高兴地感谢国防高级研究工程局和 University of Texas at Austin Applied Research Laboratories 的资助。

2. 我们使用有限的术语"设计"来引用得自 Gamma 的扩展类图符号设计方面的东西。

3. 七个不支持的重构是：改变一个类变量的值，改变一个实例变量的缺省值，改变一个实例变量继承的父类，改变一个方法的继承性，增加一个方法，改变超类的顺序。前三个重构不能够保留程序的行为，紧接着的两个不被主流面向对象语言所支持。第 6 个（增加一个方法）不能自动进行。第 7（改变超类的顺序）不支持的原因是我们这里研究的是单一继承而不是多重继承。

4. 创建一个 Circle 工厂【Tok95】。加入策略（3.2 章）。实施桥模式（3.2.2 章）。最后，加入修饰者。

5. 我们认为 1:N 关联聚合体对于设计是基础。这个模式将一个模板对象连接到一组回调对象上。这显示了模板方法知道如何使用多种对调方式，于是，它不能从 1:1 关联聚合体衍生，因为它是为单个回调方法硬编码的。

6. 在这一步中，将所有指向 Resource 的 Cicon 对象指针通过 m_objptr 实例变量进行概括。这需要在数据对象实例变量被 GUI 对象引用的时候进行适当的对数据类的转换。比如:

CicPerson *p = new CicPerson;

p->person_ptr->f_name = “John”;

被转换为：

CicPerson p;

((Person *)p->m_objptr)->f_name = “John”;

因为 GUI 类与单个的数据类相关联，这里不清楚这个设计是否正确。这个步骤不能自动进行，虽然可能做到这一点。

7. 命令模式表明了一个动作。这个动作被每个派生类中的方法 Execute()所执行。

8. 这个数字是巨大的，因为 AUIS 使用了 800 个动作，这些通过最后从过程转换成为 Command 对象。使用更多的重构并没有带来更多的复杂性。我们发现在 AUIS 上面实施重构比 CIM Works 容易的多，因为 AUIS 的设计只需要 5 个步骤。

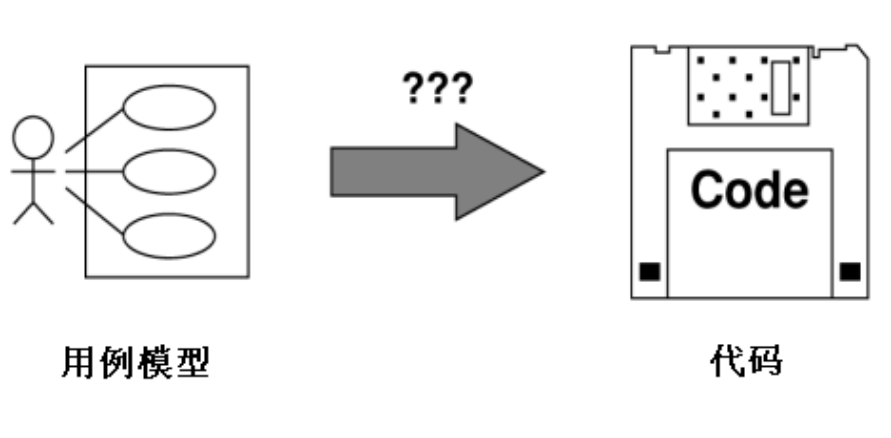
9. 许多设计模式使用运行时聚合而不是继承作为一种扩展的机制【Gam95】。聚合的动态属性预防了静态类型检查。

10. 继承破坏这个属性，因为聚合体不能有超类。

11. 当我们开始这个工作的时候，可用于支持大的 Java 文件的工具还不存在，现在有了。

UMLCHINA 培训

北京公开课：6/29-6/30



详情请垂询：UMLCHINA 北京培训中心

联系人：傅光远

电话：66130237，13910698898

EMAIL：bjcourse@umlchina.com

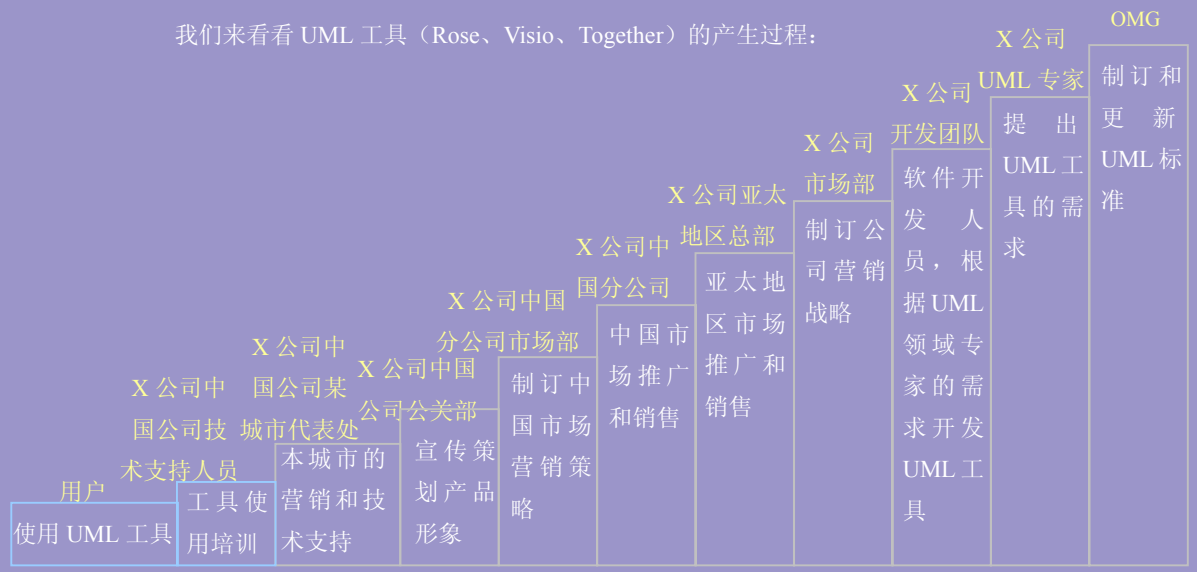
对他们是一点，对我们是一切

UMLChina，the REAL thing >>

微软开发出了占领导地位的产品是 Windows 和 Visual C++，但大家看得最多的<<Windows 程序设计>>（Charles Petzold），《Visual C++技术内幕》（David Kruglinski）等书籍的作者并非开发相应产品的微软员工。这就是现代分工。

建模领域也是如此。关于 UML，最畅销的书籍“UML Distilled”的作者不是 UML 的最初发起人，也不是 OMG，而是 Martin Fowler。

我们来看看 UML 工具（Rose、Visio、Together）的产生过程：



- 工具厂商的产品是工具，不是工具培训，更不是开发技术培训。
- 公司名字是个筐，什么都可以往里装。实际上，目前很多著名公司在中国的主要任务并不是研发，而是销售产品——真正在这个领域做研发的人在大洋彼岸。每个高质量的工具背后都会有资深的对象技术和方法学专家（如 Ivar Jacobson、Peter Coad…）支撑着，如果开发团队在购买产品之后，能得到这些大师们相应的开发方法“培训”，相信会受益无穷。但这样的机会太少了。

UMLChina 培训是 UMLChina 真正用心去打造的“产品”，不是其他产品的附属赠品——内容组织，教材和现有的各种类似培训并不相同，而且会随着技术变化，不断改进。我们小心翼翼地希望 UMLChina 培训能够在这一点点领域里尽量做好，真的能够帮助开发团队（或个人）正确地使用对象技术，开发出更好的软件。

向开发出优秀方法和工具的公司致敬，我们愿意充当斯达舒、吗叮啉，帮助您消化适合您胃口的优秀软件开发技术。

一个产品的订货和配送（系统）的分析模式

Eduardo B. Fernandez, Xiaohong Yuan, and Sandra Brey 著，li.sining 译

查看评论

摘要

本文给出的这些分析模式描述了顾客提交产品订单和订货之后的产品的配送过程。我们首先给出了两个基本的模式－－订单和配送，然后把基本模式组合成为一个我们称之为语意分析的模式，与实际应用中增加灵活性相反，我们强调应用模型的语意方面。这类模式的目的是作为把需求转化为真正系统设计的起点。这个模式代表一个最基本的应用，可以把它应用在不同的场合，也可以与其它的相关模式组合在一起来描述更复杂的应用。产品的订货和配送是现实生活中一个非常通用的问题，这个模式集中在订单和交货以及它们之间的相关过程的基本特征。

1. 引言

我们给出了顾客如何订货，以及之后的配送的分析模式，首先给出两个基本的模式，产品的订购和配送。然后把基本模式组合成为一个我们称之为语意分析的模式，与实际应用中增加灵活性相反，我们强调应用模型的语意方面。这类模式的目的是作为把需求转化为真正系统设计的起点。这个模式代表一个最基本的应用，可以把它应用在不同的场合，也可以与其它的相关模式组合在一起来描述更复杂的应用。产品的订货和配送是现实生活中一个非常通用的问题，包括顾客对一种特定的产品或者服务订货；例如，食物、书籍、磁带等，接下来就是把产品交付给顾客并结账。订货模型集中在订单的基本面，没有涉及产品和顾客的详细特征。配送模型描述了订购产品的运送。它们的组合模型的重点是把订单和相应的完成过程结合在一起。产品的详细资料如制造和可用性等留给具体的应用程序或者补充模型来完成。

2. 订单模式

最基本的模式，描述订单本身。

环境（上下文）

在许多实际应用场合中需要订购产品或者服务，例如：饭店订餐、在电子商务公司订购书籍等等。

问题

如何描述订购一种产品或者服务（的过程）。

要求

- 定单必须被准确获取。
- 订单的状态必须一直可知。
- 订单必须与最终的结果——发货或者交付有关联。
- 顾客需要的往往是一类产品，而不是某一个产品。

解决方案

图1给出了需求信息的类模型，包括描述订货过程的相关类，提交订单的顾客类和产品类。顾客类（Customer）和订单类（Order）的关联表明：每一个订单都是由一个特定的顾客给出，但是每个顾客可以下许多订单。一个订单包括由许多条目的集合组成，每一个条目表示一类特定的产品和数量。图2表示订单状态随时间的变化。订单触发其它事件产生状态变化；完成后产生一个发货对象，关闭订单并把它的信息写入一个日志文件。描述动作的顺序图由图8给出。

结论

- 这个模式只描述了产品的订货，但是它与发货有关联。
- 订货的对象可以是产品或者服务。
- 顾客可以是一个人或者一个系统；例如：一个商店订单的每一个条目都是要求生产产品的顾客订单。
- 虽然订单中必须包括完整的信息，但是只有几个特征与顾客和产品有关。
- 本例中的产品是指一类产品，不是一个单个的产品。很容易扩展到单个产品。
- 顾客信息和信用检查只在顾客第一次订货时候进行。

已知的应用和相关模式

见第5、6节。

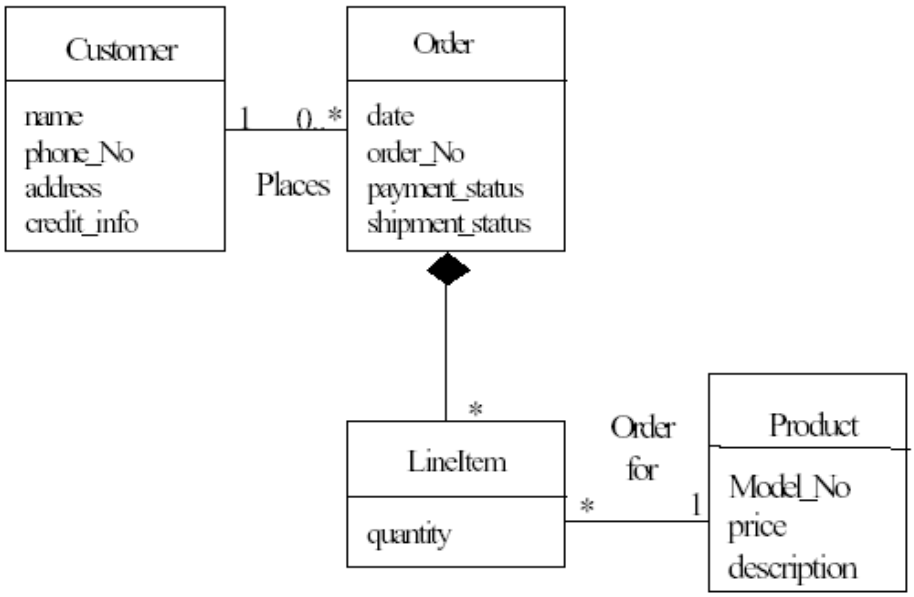


图1. 订单模式的类模型

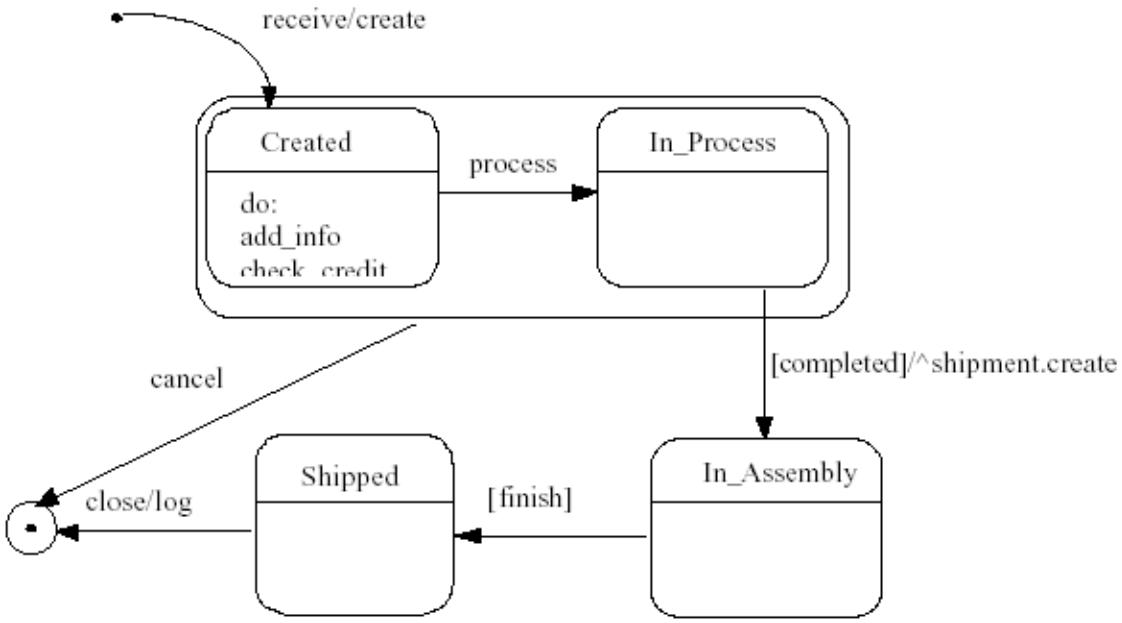


图2. 订单类的状态图

3. 配送模式

环境（上下文）

一个订单已经完成，接着应该将交付给顾客。产品必须按照订单配送。

问题

如何完成一个产品或者一项服务的交付。

要点

- 配送必须按照订单中的描述来完成。
- 配送应该能够描述需要运送的产品和顾客应付的金额（发票）。
- 有相应的顾客作为配送的接收方。
- 产品的运输方式与本模式无关。

解决方案

产品送到顾客手中，他同时收到一张发票作为付款的凭证。图3描述了这些类模型，包括表示这一过程的类：配送、顾客、发票。图4和图5是配送类和发票类的状态图，描述了这些类随事件的变化。当配送完成，生成了发票，关闭配送和发票，写入日志文件，图8是表示这些动作的顺序图。

结论

- 每一个配送必须与对应的订单相关。
- 顾客可以是个人、或者其它系统。
- 送货和结账的细节没有被包括在这个模式里。
- 收货可以与订货不是同一个顾客。
- 发票类描述顾客的应付款金额，对应着一个现实存在的文档凭证。
- 这个模型也可以应用在服务上，配送类代表所要求的服务的交付。

已知的应用和相关的模式

见第5、6节

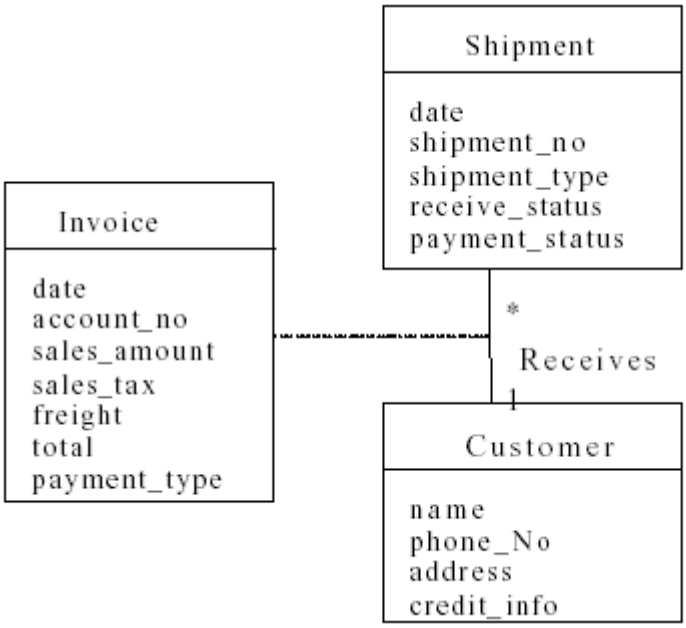


图3. 配送模式的类模型

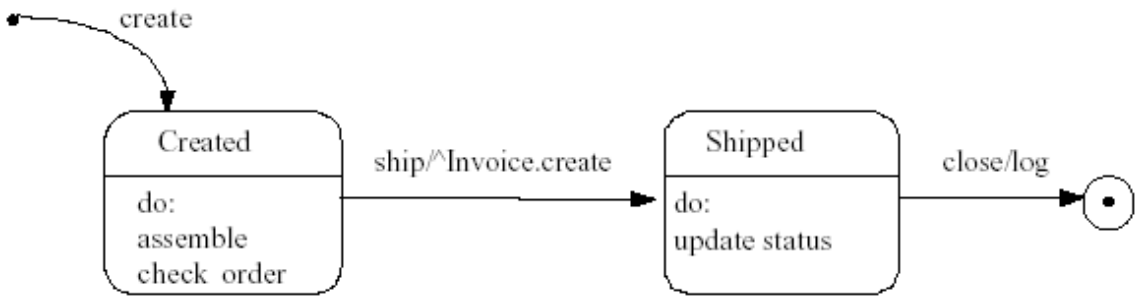


图4. 配送类的状态图

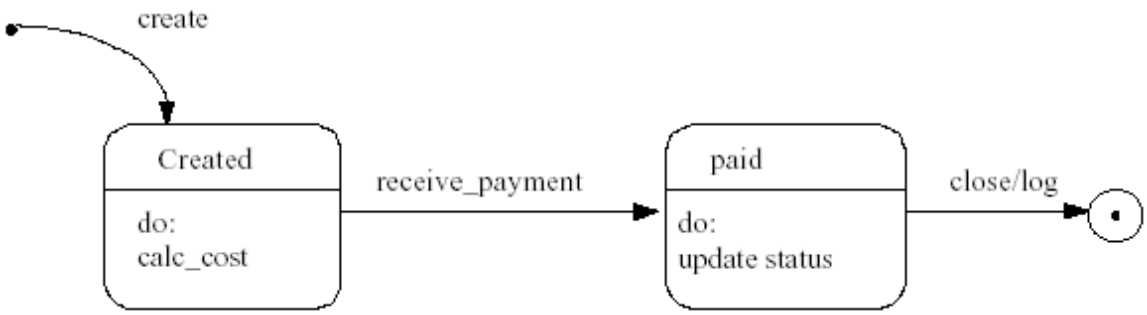


图5. 发票类的状态图

4. 订单/配送模式

目的

这个模式描述产品或者服务的订购以及相应的完成过程。

环境（上下文）

图6给出一个特定的例子，一个顾客从寻呼机生产商那里订购寻呼机。模型中包括的类有：顾客、订单、条目、配送、发票、寻呼机，各个类的含义是显而易见的。配送和订单的关联是：每一个配送都有一个订单相对应，但是一个订单并不一定导致一个配送的产生（例如 订单被取消）。配送产生的发票作为顾客与配送的关联类。

本例是普通订单和配送问题的一个特例，订单和配送问题可以应用于不同的环境，例如：订购某些产品，饭店订餐，预约维修任务。

要求

- 管理机构需要优化订单的完成时间（定性的理解）。
- 管理机构需要跟踪订单的完成情况和顾客的满意度。
- 模型必须包括表示现实生活中的文档凭证，例如：订单、条目和发票。
- 可以用等价产品替换要求的产品。
- 分析模型必须可靠的表示出需求而且不包含具体实现的细节，注意需求可能应用在不同的领域。
- 这个模式必须描述出基本的语意单元。这意味着模式必须足够简化以便可以应用到各种情况。

解决方案

a) 需求

解决方案对应着下列通用用例的实现：

- 接受订单：记录顾客信息（姓名，地址等），检查信用，然后生成订单。
- 取消订单：一个存在的订单被取消，可能用到一些管理机构的策略。
- 订单的交付：按照订单检查配送和服务。生成发票。然后将产品和对应的发票交付到顾客。

这些用例是一般性的，可以对应与不同的应用环境。某些场合其中的一些步骤是隐含的。

b) 类模型

图7是上述用例实现的类图，是图6描述的类图的抽象和扩展，用产品类替代寻呼机类，订单类中的“过程”方法概括了所有产生产品类的必要步骤，同时，配送类中聚集的操作概括了把订单中的不同部分集合在一起的操作。图中还表示出并非所有的产品订单都会最终形成配送或者配送的一些产品可能会与订购的有所不同。

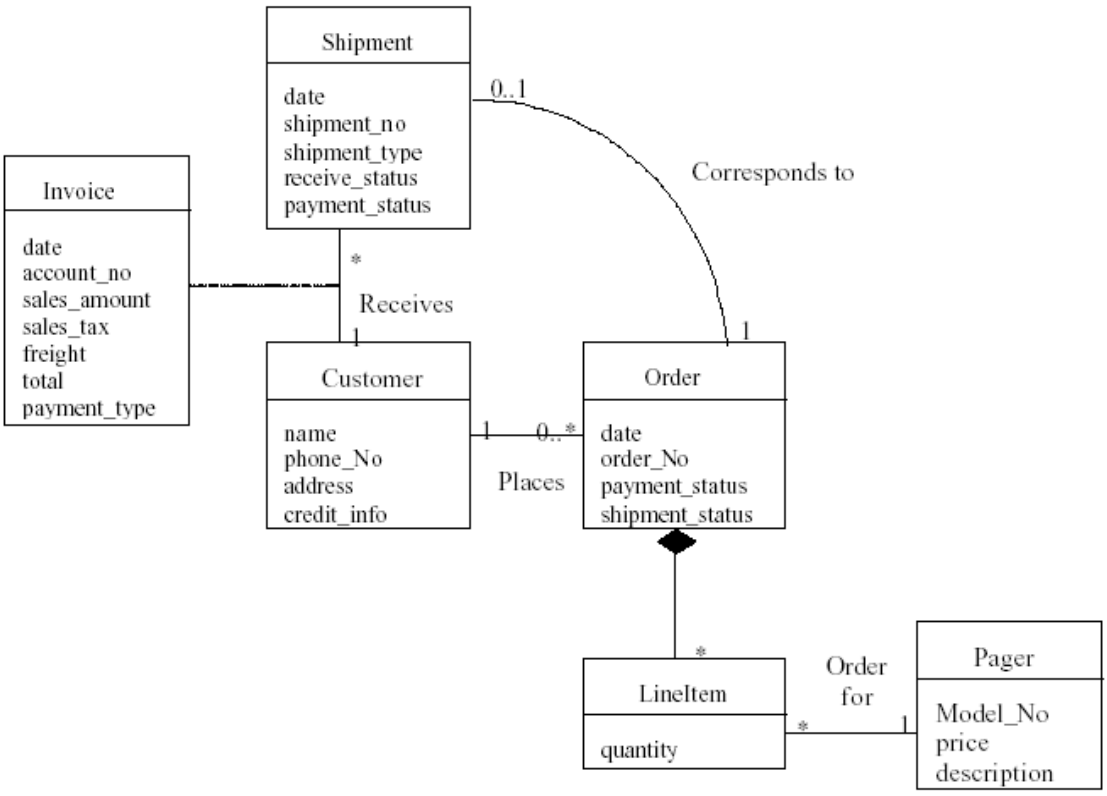


图6. 寻呼机的订货和配送

c) 动态方面

前面已经介绍了订单、配送和发票的状态图。图8的顺序图描述了顾客如何提交产品订单和之后的产品配送过程。图9描述了订货和配送的活动图。注意，在订单的关闭事件中状态图仅显示“关闭”，没有详细说明什么过程关闭订单，而顺序图和活动图显示是结账的事件关闭订单。在图8中，接收产品和结账的顺序可以互换。

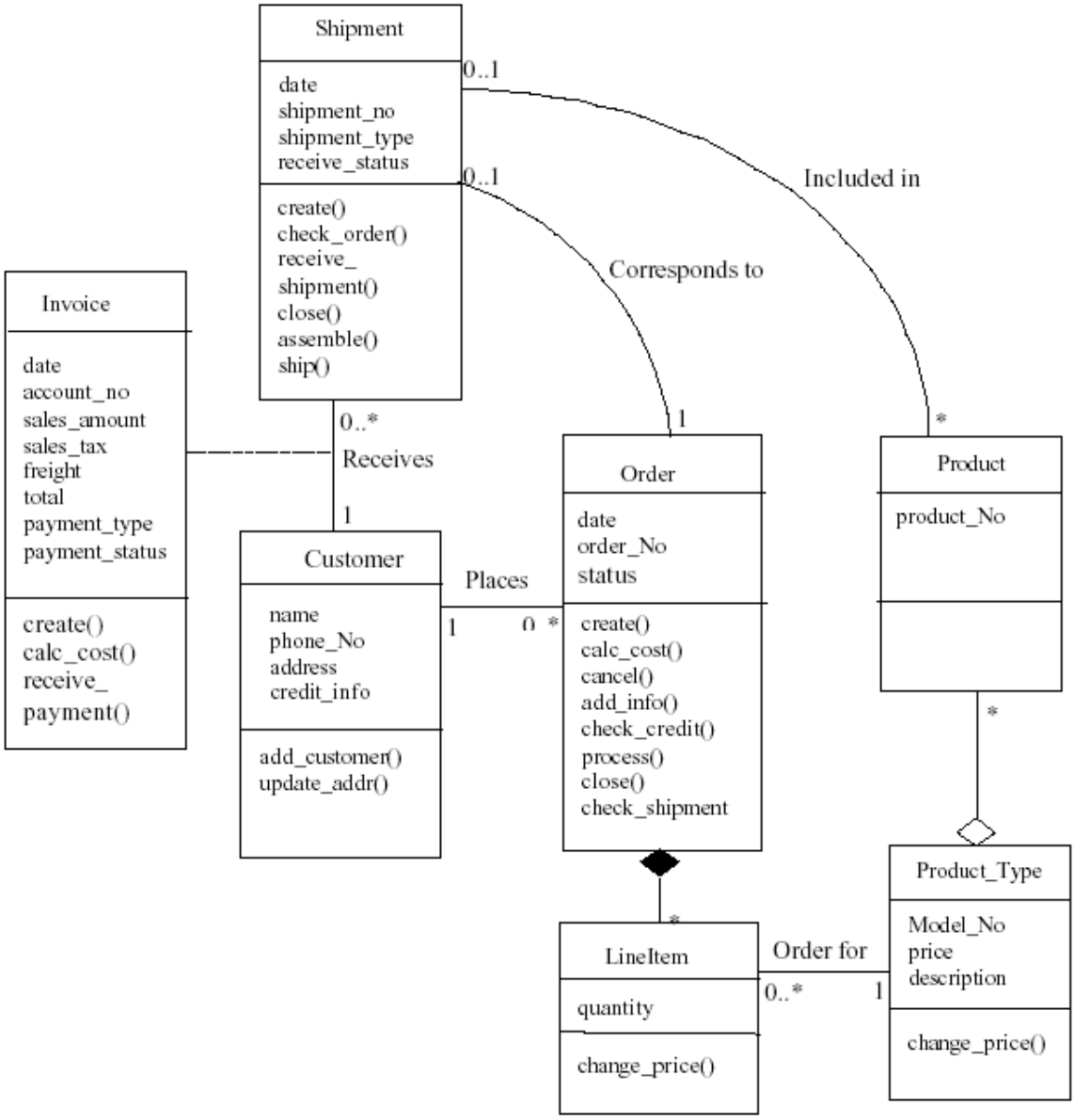


图7. 一个产品的订货和配送的类图

d) 结论

上面的要点对所有这个模型的实现是通用的：

- 一个订单总是按照顾客提出的需求在“系统”中生成。在现代生产工厂或者网上商店，订单几乎肯定是采用数据库中的一条记录的形式。同样的，在历史悠久的快餐企业如Taco Bell，订单是作为计算机系统中的一个虚拟的实体存在的，通过在售货柜员机的一个特殊的按键（图标）即可输入。订单（无疑是虚拟的）可以从顾客的网上请求自动生成。在许多传统的饭店，订单包含账单，由服务生按照顾客的要求手写而成。

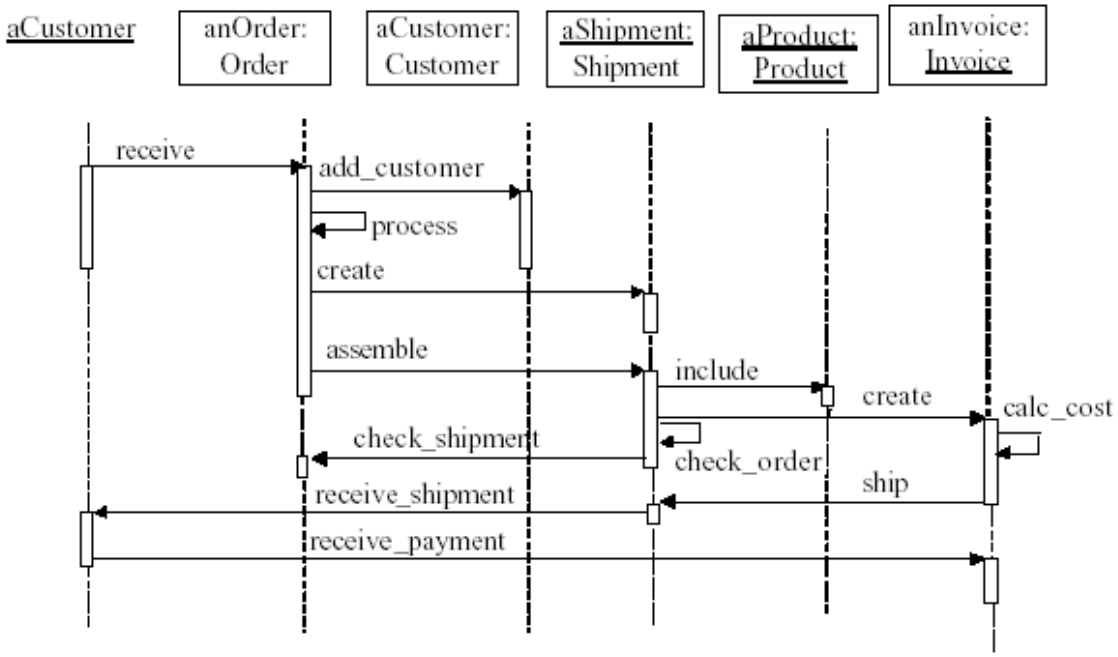


图8. 预定和接收产品的顺序图

- 一个订单总是与一个顾客相关：这意味着，对应的顾客一定能够被识别。在快餐店的滚动流程中，这很简单，可以通过打印的收据上的流水号、饭店账单上的餐桌号码来识别顾客，或者通过更为复杂的系统，使每一个订单都通过一个指针（例如：‘顾客编号’）连接到一个顾客数据库，其中包含信息如姓名、地址、电话、账号状态、折扣等级、购买历史等等。

- 在一条订单的记录上，可以推断下面三种情况之一会发生：

a). 产品有库存，然后订单被执行。

b).产品无库存，商家必须订购或者生产产品。

c).订单在配送之前被取消。

- 总要生成某种文档凭证，一份拷贝作为归档资料，一份拷贝与产品一起送交顾客：

- 这里可以采用现金收据，或者发票，或者两者都要，决定于付款和送货的顺序。凭证包括重要的信息如时间、产品清单和应付（或者已付）价格。

- 订单可以是单个产品，不是一类。

- 产品的接收顾客可以是另外的子系统或系统。

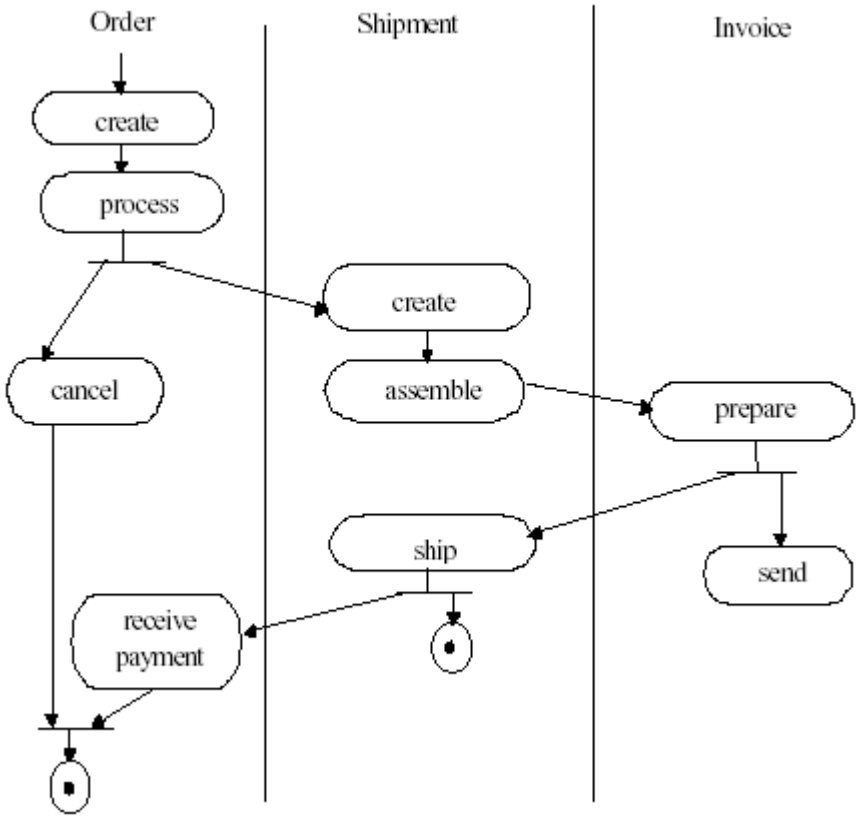


图9. 订单和配送的活动图

这个模式的应用提供了系统的方法来记录订单和它的完成情况，并且能够有助于优化完成周期和满意度。这个模式也应用于服务，这种情况下，配送表现为服务的提供。

类图中应用订单模式到一个通用的产品类，可以通过改变条目（LineItem）和产品类型（ProductType）与产品（Product）的关联来适配特定产品的应用。

并不是说由此模式描述的所有应用情况都是完全类似的：

- 获取和校验用户数据的过程在某些应用场合可能是十分冗长的，而在另一些应用场合可能根本不存在。
- 用户可能直接从商家取回订购的产品，这样模型中的配送过程就被跳过。
- 可能要求在产品送达前付款，也可以送达后结账，决定于顾客的状态和信用等级。

为了一般性，下面许多侧面没有在这个模式中描述：

- 产品的上下文和环境方面。
- 异常，例如：不良信用
- 如何跟踪产品的可用性
- 当产品无货时，如何排队等候。
- 如何处理不同的顾客，例如：个人顾客，集团顾客，优先顾客。
- 订单的更改
- 退货
- 支票付款的策略
- 配送的实现细节，例如：包装。
- 历史记录

所有这些方面需要由其它模式或者特殊的模型来完成。你可以扩展这些模式成为订单的模式语言。

5. 已知的应用

下面是一些应用实例：

- 某种设备的零售/服务提供商，例如：寻呼机，请求在零售店转卖一批设备。
- 顾客从饭店订餐
- 顾客从电子商务公司订购产品，例如：亚马逊（Amazon.com）
- 顾客为她的家订购了一个新的屋顶

文献中有许多这个模型的变种：

- Hay [Hay96] 给出订单和配送模式，但是他没有给出它们之间的关联，也没有考虑动态方面、属性、或操作。

- Fowler [Fow00] 利用订单作为他的UML书中的一个运行实例。我们的活动图就是基于此。
- Berkem [Ber99]描述了更加详细的活动图并把它们与用例连接起来。
- Ambler [Amb97]描述了订单的类图，他没有考虑订单。
- Schneider and Winters [Sch98]枚举了各种应用了订单模式的用例。
- Richter [Ric99]在股票交易环境考虑订单模式。
- K. Brown [Bro96], 对一个订单管理系统应用了设计模式，他没有讨论分析方面的内容。
- [Ope00]一个开放的应用程序组，为订单和配送定义了接口，使之能够在不同系统间协调工作。

6. 相关模式

预定和使用模式[Fer99]，本模式的补充，提供预定正在不断生产的产品的可能性。

股票管理模式[Fer00a]也是一个补充，完成了需求部件的订单并更改了库存内容。

如前所述，本文给出的模式可以作为制造业框架的一部分。类型对象模式[Bau00]在这里作为它的子模式。

顾客在过程可以担当不同的角色，角色对象模式可以用来描述这个过程。

关于货币的详细处理可以在模式 [Hay96] 和 [Fow97] 看到。

依赖请求模式[Hau97]讨论了在生产体系中一个订单可以触发另外的订单。

订单/配送模式可以作为复合模式 [Rie96] 的一个特例来研究。

致谢

感谢我们的带头人和Plop2000写作工作室，它们详细而有深刻见解的评论极大地提高了本文的质量。

参考文献

[Amb97] S. Ambler, “Taking a layered approach”, *Software Development*, July 1997, 68-70.

[Bau00] D. Baumer, D. Riehle, W. Siberski, and M. Wulf, “Role Object”, Chapter 2 in *Pattern Languages of Program Design 4,* Addison-Wesley 2000. http://st-www.cs.uiuc.edu/~plop/plop97/Workshops.html

[Ber99] B. Birkem, “Traceability management from business processes to Use Cases with UML”, *JOOP,* September 1999, 29-34 and 64.

[Bro96] K. Brown, “Experiencing patterns at the design level”. *Object Magazine*, January 1996, 40-48.

[Fer99] E. B. Fernandez and X. Yuan. “An analysis pattern for reservation and use of reusable entities”, *Pattern Languages of Programs Conference*, *PloP99*. http://st-www.cs.uiuc.edu/~plop/plop99

[Fer00] E.B. Fernandez and X. Yuan, “Semantic Analysis patterns”, *Procs. of 19th Int. Conf. on Conceptual Modeling, ER2000*, 183-195.

[Fer00a] E.B. Fernandez, “Stock Manager: An analysis pattern for inventories”, *Procs. of PLoP 2000*.

[Fow97] M. Fowler, *Analysis patterns -- Reusable object models*, Addison- Wesley, 1997.

[Fow00] M. Fowler, *UML Distilled* (2nd Edition), Addison-Wesley, 2000.

[Hau97] R. Haugen, “Dependent Demand—A business pattern for balancing supply and demand”, *Procs. of Pattern Languages of Programs Conf*., PloP97, 14 http://st-www.cs.uiuc.edu/~plop/plop97/Workshops.html

[Hay96] D.Hay, *Data model patterns-- Conventions of thought*, Dorset House Publ., 1996.

[John98] R. Johnson and B. Woolf, "Type Object", Chapter 4 in *Pattern Languages of Program Design 3,* Addison-Wesley, 1998.

[Ope00] Open Applications Group, *Integration Scenarios,* http://www.opeanapplications.org/oagis

[Ric99] C. Richter, *Designing flexible object-oriented systems with UML*, Macmillan Tec. Publ., 1999.

[Rie96] D. Riehle, “Composite design patterns”, *Procs. of OOPSLA’97*, 218-228.

[Sch98] G. Schneider and J.P. Winters, *Applying Use Cases—A practical guide*, Addison-Wesley, 1998.