2002－9（第17期）

Marko Boger

X-Programmer 非程序员

软 件 以 用 为 本

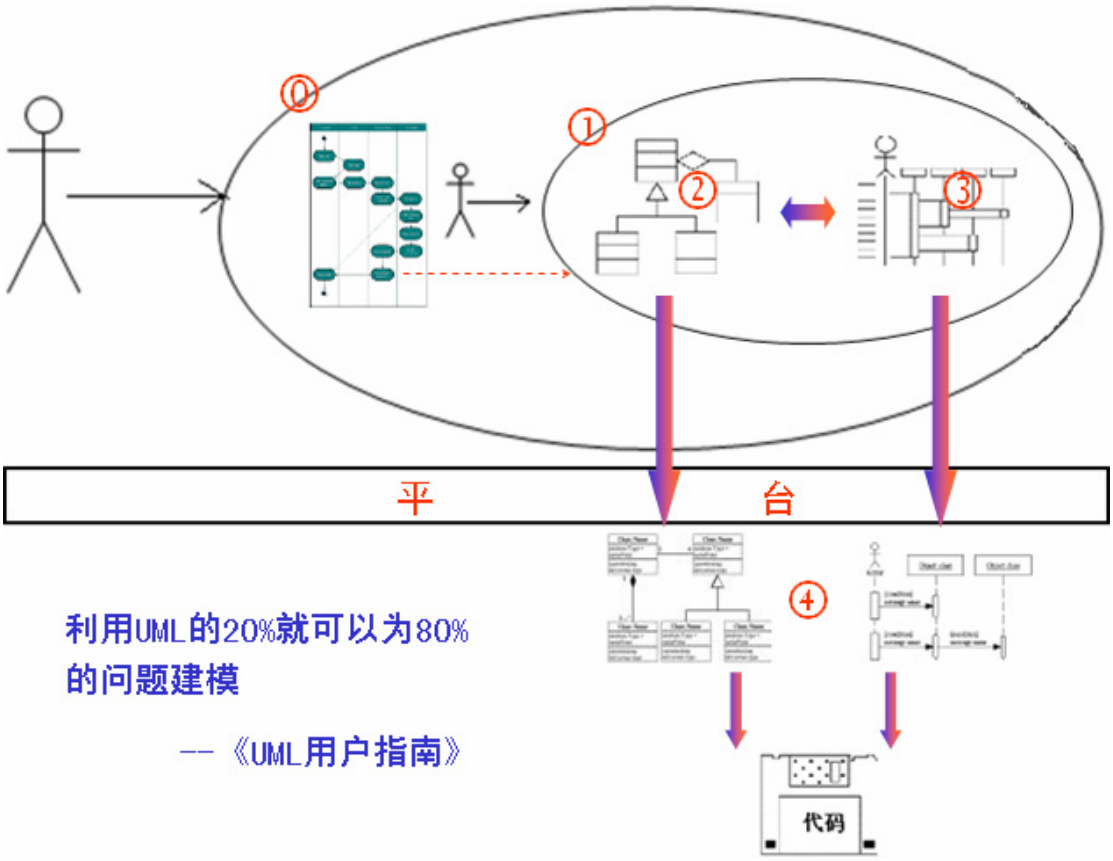
主编：davidqql

审稿：davidqql，think

投稿：editor@umlchina.com

反馈：think@umlchina.com

http://www.umlchina.com/



北京公开课，2002 年 9 月 14 日至 15 日

北京中关村 1 号海龙大厦 713-714 报告厅

报名交费请联系：bjcourse@umlchina.com

Marko Boger： XP、UML 和 Poseidon

查看评论

编注：

北京时间 8 月 6 日（星期二）下午 15:00-17:00，Marko Boger 先生作客 UMLCHINA 讨论组的聊天室。Marko Boger 是 Gentleware AG 的创始人和 CEO。Marko Boger 在汉堡大学获计算机科学博士学位。专著领域：Java、敏捷建模、UML。Gentleware AG 的产品是基于开源项目 ArgoUML 的商业 UML 建模工具：Poseidon for UML。人民邮电出版社出版的《极限编程研究》（XP Examined）有一篇他关于极限建模的文章。以下是交流实录。由李巍翻译。原文链接。

markoboger：先让我来介绍一下我自己。我在汉堡大学作了多年的研究工作。在那里，我使用的实际上是一种叫 Dejay 的分布式语言。为了将这些成果能更好地表达出来，我开始在工作中使用 UML 工具。于是我便发现了 ArgoUML。最初，我和我的一些学生采用 Argo 来满足我们的要求，我们便成了这个开放源代码项目的领导者。在研究过程中，我们也涉及到了一些软件工程的问题，如 XP 和极限建模（extreme modeling），取得了一些非常不错的成果。再以后 ArgoUML 变得非常流行起来，于是我们决定成立一个公司，来实现我们的软件工程观点。

reduce：我们使用过 ArgoUML 和 GEF，但它在用的时候好像有很多 bug。

markoboger：对，Argo 的确有许多小的 bug 和问题。

reduce：你们是如何在 ArgoUML 的基础上来对 Poseidon 进行开发的呢？

markoboger：把一个像 Argo 这样的研究项目做成一个功能完整的产品（如 Poseidon），在这中间需要走一段很长的路。通过 Poseidon，我们努力地挖掘 Argo 的潜力，并将其转化成一个完全成熟的、高效可用的 UML 工具。OK，我们现在感觉已经差不多达到这个任务，该是回到我们最初目标的时候了，即开发一个在建模时便能支持 XP 的工具。

reduce：Poseidon 是在 ArgoUML 的基础上重新构造（Restruction）的，还是改建（rebuild）的呢？

markoboger：我们重新构造了 Argo。

netsong：我们怎样才能得到 ArgoUML 呢？

markoboger：在 www.argouml.org 可以找到。

netsong：我们能够在 XP 的哪些地方使用 UML 呢？

markoboger：你可以在 XP 中非常好地使用 UML。我想你们可以访问 agilemodeling.org 站点，那里有很多想法是描述如何在 XP 中使用 UML 的。那里的主要作者是一个顾问，他不得不整天与那些可用的工具（the tools available）在一起。

reduce：你们计划着如何去做（新的 UML 工具）呢？

markoboger：现在我们作为工具的制造者，我们感觉有必要做一个工具，能够改进一个包含 XP 和 UML 的过程。我们在内部使用了大量的 XP 的元素，但这并非是严格意义上的 XP。当然，我们也用到了很多的 UML。

reduce：请问您使用了多少人员来开发 Poseidon for UML 呢？

markoboger：在我们公司，目前大约有 7 个开发人员在做这个工作，进展很好。

founder_chen：UML 2.0 即将对实时设计进行支持，请问您对此是怎么看的？

markoboger：Gentleware 公司非常关注 UML 2.0 的进展。我们负责一份图形互换（diagram interchange）的提案。

yuansj：请问 Poseidon for UML 项目的开发周期是多少？

markoboger：在每一个小的版本之间的周期大约是二至三个月。

freekany：您有关于 ArgoUML 和 XP 一起使用的成功案例吗？

markoboger：我们在内部使用了大量的 UML 和 XP，这算不算是案例呢？但还是遗漏了一些 XP 的元素。举个例子，我们发现很难将测试引入到一个现有的尚未针对测试进行设计的项目中。而且对于图形用户界面的测试将更难以进行。于是我们不得不对 XP 作一些调整来满足我们的需要和环境。在很多领域里这并不是很难，但是仍然会涉及到许多相关的背景知识。

netsong：如果没有足够的文档，您如何管理自己的项目？

markoboger：没有最后期限，没有严格的计划。

inevity：请问 ArgoUML 与 Poseidon 有什么不同呢？

markoboger：ArgoUML 是一个开放源代码的项目，这便是它与一般的项目所不同的。这也只是简单的提法。当然，Poseidon 还会有很多不同之处。

reduce：你们的 Poseidon 是下一代的 UML 工具吗？但是 Rational 公司使用了大量的人员来开发 Rose。您有没有想过要转移到 RUP 这边来呢？

markoboger：很显然，我们是 Rational 的竞争对手。如果 Rose 是更好的工具的话，我想我们就不会开始 Argo 的开发了。

founder_chen：ArgoUML 支持重构吗？

markoboger：我们开发了一个重构的工具，并将其作为 Poseidon 的一个插件。它是在上一次在意大利的 XP 会议上提出的。

timwap_cn：您是如何从开放源代码的项目中获得利润的呢？

markoboger：我们没有从开放源代码的项目中获得利润。我们是对这个开放源代码的项目进行改进，并从中获得利润。与 Argo 相比，Poseidon 现在作了许多的改进，我们所出售的是这种改进。顺便说一下，我们有一个免费版本的 Poseidon。

timwap_cn：是靠提供服务（来获得利润）吗？

markoboger：我们也提供一些服务，如咨询，培训，提供专门的工具等等。

freekany：能够直接得到 ArgoUML 的源代码吗？

markoboger：是的，你可以得到 Argo 的源代码。

zjuxxl：对于 ArgoUML 的开发，你们是否提议过一份 UML 形式语义？

markoboger：UML 2.0 已经在语义上作了很多的改进。一个叫 2U 的团体制订了一份定义得非常好的形式语义。该团体从 pUML 演变而来，可参见 www.puml.org。

wubuy：请问 pUml 是什么？

markoboger：pUML 是一个团体（称为精确的 UML），他们的大多数成员我都认识。

timwap_cn：有没有可能将 XP 和 RUP 并入一个过程中呢？

markoboger：那应该很容易，但是需要我们了解整个过程。

founder_chen：RUP 和 XP 哪一个更好呢？或者 RUP + XP = XUP？

markoboger：Uncle Bob（译注：Robert Martin）在 XP 会议上作过一个演讲，题为 XP 是 RUP 的一个实例。一般说来，RUP 是一个很模糊的概念，实际上任何过程都可以看作是 RUP。

freekany：实例？

markoboger：是的，一个实例，一种表现形式。RUP 类似于一种元过程（meta-process），是一个过程模板的描述。你可以从它构建一个过程，即一个实例。

zjuxxl：在开发 ArgoUML 前，您还做过怎样一些关于 UML 的工作呢？

markoboger：在 Argo 之前，我是一名汉堡大学的研究员。1995 年，我写了第一篇关于 UML 及其如何改进的论文。后来我提出一种图，来更好地反映方法的结构及其间的交互。序列图可适用于这些示例，但其并非 big picture，于是我便开发了方法图（method diagram）。

reduce：我们能在 Poseidon 看到方法图吗？

markoboger：不能，我们仅有一个方法图的原型，但它并没有在我们的产品中出现。

freekany：Argo 没有代码生成的功能，而 Rose 则可以最后生成项目的代码。

markoboger：我们有一个 Poseidon 的插件，实际上它可以从状态图（state chart）中来生成代码，并且能够执行。于是你就会在图中看到有关状态的变化。如果你愿意的话，可以将它认为是一种模拟。

freekany：什么插件呢？

markoboger：这个插件的名字叫 State2Java。当前这只是一个 beta 版，我们会进一步去改进它。而这也意味着当前它是免费的，但是你需要使用 Poseidon 的标准版。你们可以从我们的站点 www.gentleware.com 上得到它。

wubuy：我从来没听说过这个东西，您能告诉我什么是 State2java 吗？

markoboger：State2Java 是我们执行状态图的一个插件。它真正反映了一些我们关于极限建模的思想。由于类图无法对行为进行建模，所以你将无法执行它们。而状态图是可以对行为进行建模的，因此如果你使用它们的话，实际上就可以执行模型。它的功能非常强大，但仍需要对其多加考虑。它是一个可以在编码与建模之间切换的范例，只要你能按照那个步骤来做，你的模型就能够执行。

zjuxxl：ArgoUML 是否支持模型的检查、跟踪和模拟呢？

markoboger：你当然也能够对模型进行测试。序列图可以非常好地表达这样的测试，而用例图也可以作为一个测试的规划图。

ozzzzzz：请问什么是极限建模？

markoboger：极限建模是由我开发的一个过程的建议。在一本 XP 书中有一篇关于这方面的文章，我想它已经被翻译成中文了。另外还有我的一个站点 www.extrememodeling.org，虽然尚未全部完成，但我想你还是能够从中得到一些好的想法的。

cinc：极限建模与其他的建模方法有哪些主要的不同吗？

markoboger：你应该多从模型上去考虑。实际上 XP 注重的是模型，而非代码。

timwap_cn：请问该工具有没有提供建模方面的指导呢？

markoboger：敏捷建模（agile modeling）专注的是当前可用的工具。而如果使用像 Rose 这样的工具，则很难保持敏捷，于是敏捷建模得出的一个结论就是最好使用白板来代替它。我们现在进行的是工具的开发，我们便能够以自己所需的方式来构建它们，并使之变得敏捷起来，这便是我们的优势。但我们还没有到那一步。State2Java 仅仅是一个 beta，还需要对其进行更多的改进。例如我们已着手做一个支持 UML 的重构浏览器，它同样需要使用工具来进行敏捷建模。这是一个复杂的话题，我们仍需要时间来开发我们需要的所有工具。

ozzzzzz：对我来说，Rose 太贵了！

markoboger：是的，Rose 的确非常昂贵。我们尽量提供一个价钱低得多的工具，我们能够做到，因为我们采取的是不同的市场与发行策略。

cinc：请问可以在极限建模中使用 CRC 卡吗？

markoboger：CRC 卡是一个不错的主意。但我想它应该是你在进入极限建模前的一个步骤。

reduce：您有没有打算在 Poseidon 中兼容 Rose 呢？

markoboger：是的，我们的确准备在下个版本中能够导入 mdl。那时候你便可以在 Poseidon 中读取 Rose 文件。

reduce：那下个版本是不是要等到半年或一年以后呢？

markoboger：不会的，你需要经常发布版本，你需要知道你的工具是稳定的，这一点非常重要。不会等半年，最多也就是 2 个月。

timwap_cn：你们的主要客户是哪些呢？

markoboger：我们的客户多数是一些小到中型的公司，以及一些小到中型的 Java 项目。

wubuy：现在我们有一个项目，我想在项目中加入 XP 的方法。您能向我们说点什么吗？

markoboger：XP 非常不错。

wubuy：怎么不错呢？

markoboger：你们是从零开始吗？

wubuy：是的。

markoboger：如果是的话，那么请保证能够在非常早的时期就开始测试。

wubuy：我不知道怎样测试。我对 XP 或测试方面很生疏。

markoboger：在 XP 中测试是非常重要的，你必须要学习。你首先要写一个测试，然后再编写代码来对其进行实现。对于每一个特性，你都需要进行测试。

firebird_online：XP 对于一个仅有四个成员的团队有用吗？

markoboger：你需要一个团队。我们发现在一个很小的团队里实施 XP 是很难的，最好能有六至八个人。我们现在这个团队里有七名开发人员，现在工作得很好。

wubuy：在我的项目中只有三个人。

markoboger：如果你人员太少的话，将无法进行结对编程，因为你可能会经常找不到所需要的同伴。请继续并且去尝试。虽然 XP 对于一个中型的团队是比较适合的，但即使你的团队很小，XP 仍然比其它的要好。

awarmer：在开发 Poseidon 的过程中，你们使用了哪些工具呢？

markoboger：建模方面，我们当然是使用自己的工具。编程方面，我们则使用 IDEA，此外我们还使用 Netbeans 和 eclipse，还有 emacs。不过现在大多数开发人员都使用 IDEA，它是一个很不错的产品。

coolie2k：请问需求文档化的工作能否使用结对的方式来做呢？

markoboger：我们做了大量的有关结对建模方面的工作。我想你可以尝试做一些结对需求建模(pair requirements modeling)方面的工作;-)

wubuy：那如何开始呢？

markoboger：如何开始实施 XP？这是个难以回答的问题。但测试往往是一个绊脚石，必须要熟练运用 JUnit。从一开始就写测试，这会使你的 API 变得非常简单，而且是可以测试的，这一点非常重要。测试必须要由编写代码的人来做。也许你们仍然需要一个测试过程以及测试工程师，但你还得让开发人员编写测试代码。因为只有那样，才有可能发现所有的问题。你可以一种不同的方式来使用这些测试。你不是要竭力地去找出 bug，而是要竭力地去证明你的特性可以在所有情形下工作。如果你能保持所写的测试能够随时更新的话，那么你便可以去做一些改动，然后看看是否会对别的什么地方造成了破坏。这样做将会在很大程度上鼓励你去进行改动，从而获得敏捷。

firebird_online：敏捷？那 XP 是如何保证质量（quality）的呢？

coolie2k：但是我想如果让一个人同时担任这两个角色，质量（quality）会更好。

markoboger：质量：这是一个需要不断学习的问题。一个人在项目中扮演的角色越多，则其对整体的把握也就越强，这将会提高质量。

reduce：如何来进行结对编程呢？是两个人同时做同一件事情吗？

markoboger：是的，让两个人同时坐在同一台计算机面前，一起查看代码。其中一个人更多的是负责具体的细节，而另一个人则更多关注整体。但是这两个人的角色可以随时交换。

firebird_online：这两个开发人员经常要改变各自的角色吗？

coolie2k：开发人员承担了两个角色。

markoboger：开发人员承担着很多的角色，以至于你仅能称他们为开发人员。

reduce：你们在 Poseidon 中使用结对编程了吗？

markoboger：是的，我们在 Poseidon 中使用了大量的结对编程的方式，它确保了所有人都理解架构，这也减少了一些设计方面的问题。

wubuy：我听说过结对编程，但是它看起来好像效率不够高？

markoboger：它增添了很多乐趣，而且产出也比较高。我们认为它是非常高效的。毕竟，大多数的时间是用来修改代码，你可以通过结对编程的方式来缩减所花的时间。

cinc：在 XP 中，你应该先写测试然后再写代码，并使之测试通过，这是一个很重要的 XP 原则。

markoboger：说得对，你必须在编程时就编写测试案例。实际上你必须要先写测试，然后再进行编程。

cinc：你们是如何进行自动测试的呢？是使用 Jarkata ant 吗？

markoboger：是的，测试必须能够自动进行。你必须能够持续不断地来进行测试。

wubuy：您能不能举一个实际的例子来解释 XP 呢？

markoboger：好吧，我举一个 XP 的例子。事实上在我遇到 Kent Beck 时，XP 还不怎么有名。有一次我们都参加了一个会议，在返回的火车上我碰到了他。于是我们开始交谈，他说他正在做一个名为 XP 的东西，下面我将会举他用到的这个小例子。实际上我们那时坐在火车上，他使用的是自己的便携式电脑，然后我们决定作一个货币转换（converts currencies）的小程序。你们可以作一个小测试，来描述一下什么是你们实际要做的，以及应该得到的结果是什么。对于一个货币转换器而言，你可能需要将 100 美元转换成欧元。在目前这是非常简单的，因为汇率大约是 1，于是转换的结果应该是 98。你可以使用 JUnit 来编写一个测试，并且编写一个接口来使编译通过。虽然现在会测试失败（因为现在还没有对其进行实现），但这个接口已经是可以测试的了，因为你事先已经设计了测试。这样一个例子好像看起来并不是那么有价值，但是对于一个大型项目来说，将会非常重要。发现测试失败

后，现在让我们来实现这个方法，看看是否能得到我们想要的结果。而你仅仅需要重新运行该测试。记住，我们的测试都会自动进行，所有它们的结果不是 true 就是 false。现在我们便实现了这样一个小特性，即将 100 美元转换为 98 欧元。然后进行测试，如果它对于负数也同样工作的话，那么-100 美元的结果将是-98 美元。再编写一个测试，看看结果是不是这样。让我们继续，现在我们想要自己来决定使用哪种汇率。于是你需要首先写一个这样的测试，并使之编译通过，现在你有了正确的接口，但这时测试会失败，因为该接口还没有被实现。在这里结对编程便显得愈加重要，因为我们需要讨论最好应该使用哪种接口，并且可能会有各种不同的想法。所以说，接口的确是非常重要。现在实现这个接口可能会（或不会）破坏你的第一个测试，但是不必担心，因为你有一个针对它的测试。如果你做的一些事情使得代码被破坏了，那么测试也同样会失败，而这样你便能够迅速地确定问题出在哪里。这一点也会使你获得比用其他方法更快的速度。你的调试将会变得更加快速，因为你已经对每一种小特性都进行了测试。你会确切地知道什么能够让你出错，以及什么能够用来发现问题。而在做这些之前，我们用到了大量的 UML 建模。通常情况下，我们多是从结构（structure）上进行思考和工作，而且 UML 也给了你一个比较好的关于结构的描述。而我们则使用自己的 roundtrip 工具来做这件事情。

charliexia：我对架构不是很清楚，您能结合一个简单的例子谈一谈什么是一个应用的架构吗？

markoboger：架构。如果你去看一座桥，便会非常清楚地看出它的架构。有的桥是用金属绳索拖拽而成的，有的桥是拱形的，有的桥则显得非常结实与稳固等等。而对于软件而言，其架构则很难能一眼看出。但两者类似的是，架构是可以复用的。例如在 Poseidon 中我们有一个树形的层次架构（layer architecture），即一个存储区（repository），控制器（controller）和接口。而这些接口本身又被构建为三层。然后我们使用了一个事件机制来传达发生的变化。如果你的代码干净，便可以从你的 UML 图中发现这样的架构。而模式则是一种很好的描述软件架构的语言。

freekany：模式？

markoboger：不错，是模式。有一本非常不错的书，作者是 Eric Gamma 以及其他三个人，他们也被称之为四人帮（GoF）。他们描述了一组可以在软件中复用的模式，并对它们进行了命名，同时他们也描述了这些模式。顺便提一下，Gamma 还与 Knet Beck 一起编写了 JUnit。

charliexia：在应用程序的开发过程中，将不能对这个应用的架构进行改变，您认为对吗？

markoboger：一个架构是能够演变的（evolve），但最好是从一开始便能正确地获得基本的架构。可是在开始时你又如何能知道呢？我们在构建 Poseidon 时，就不得不对 Argo 的架构作了很多次改动。这要做很多工作。

firebird_online：“我们的测试都是自动的”指的是什么意思？是指使用 JUnit？

markoboger：是的，所有的测试都应是自动的。我们使用 JUnit 来对代码进行测试，使用 QFtest 来测试用户界面。

reduce：Poseidon 中的 GEF 部分是公开源代码的吗？

markoboger：GEF 是公开源代码的。

wubuy：目前在中国，有一些项目（并非是少数）只有二个或三个人。他们该如何使用 XP 呢？如果根据您刚才说的那样，我想他们的人数太少了。那您能给我一个最好的建议吗？

markoboger：你可以从二个或三个人开始做起，但如果能有六个或八个人则会更容易些。这是因为要进行结对编程，所以只有三个成员是比较困难的。但即使你无法进行结对编程，你也必须要确保能够正确地进行测试。

firebird_online：顺便问一句，需要什么样的开发人员来进行结对呢？

markoboger：你团队中的所有人都可以。如果他们能够具有不同的技能，那就太好了，他们可以互相学习。

coolie2k：我能把协议层（通信）映射到您说的架构上去吗？

markoboger：是的，层是架构的一个重要表现形式。

timwap_cn：在实施 XP 过程中会不会遇到一些阻挠呢？

markoboger：阻挠。是的，可能会有。但这往往是一些管理上的阻挠。对于 XP 而言，最大的阻挠通常来自管理层。而我们则没有这种问题，因为我是作为一个管理者来引入 XP 的。

freekany：架构与框架（framework）有什么不同呢？

markoboger：它们是两个不同的东西。一个框架可以有一个架构。而框架与库也是不同的。库是让你从自己的应用程序中来对其进行调用的，而框架则是来调用你的应用程序的。它们彼此是一个相反的过程。例如 EJB 是一个框架，你必须要编写自己的代码，这样 EJB 才会对你的组件进行调用。而 Swing 则是一个库，你可以对其进行调用和使用。作为你的代码的结合体，框架和库都属于架构的一部分。

firebird_online：我们公司有十六个开发人员，对于软件工程来说这有些少，那您觉得有没有可能来实施 XP 呢？

markoboger：十六个开发人员对于 XP 来说太多了。XP 的一个问题便是，它无法用在一个大的项目中。如你所言，可以项目中的开发人员分成十个人和六个人两组。但我认为在一个 XP 项目里最好不要超过十个人，任何一个十

人以上的项目都是难以控制的。

timwap_cn：那管理呢？

markoboger：在 XP 中最重要部分的就是团队领导者（team leader）。你必须要有一个好的团队领导者，要知道这便是管理。

timwap_cn：有没有一些关于 XP 方面的度量标准（metrics）呢？

markoboger：我们正在考虑把 XP 的度量标准做成一个 Poseidon 的插件，因为这样会帮助你发现一些架构中的问题。但它还是会很容易造成误解。良好的测试更加重要。

firebird_online：XP 的领导者会不会也必须是一个好的编码者呢？

markoboger：领导者不必非要是一个好的编码者，尽管这样会更好。他需要是一个好的测试员和架构师，但最重要的是一个好的领导者。他必须要说服人们来遵循 XP，他必须要教会他们应如何去做，他必须要创造这样的氛围。

timwap_cn：请问 XP 的根本（base）是什么？项目的一个团队，还是一个项目。请问有没有这样做的一些指南呢？

markoboger：有很多非常不错的有关 XP 的书，由 Kent Beck 开始。

charliexia：没有人能够写出一个预先设计好的架构（up-front architecture），这是因为软件是“软”的，它不仅能够而且应该是易于演变的！请问您同意吗？

markoboger：软件是能够演变的，这一点我赞同。在 Poseidon 中我们拥有非常多这方面的经验，在构建过程中，我们必须能够学会如何构建这样一种工具。经验是非常重要的，但是如果你没有经验，你会如何开始呢？获得经验唯一的方式就是开始去做。去看看那些已经做过的人们，并且 XP 本身就是这样一个很好的例子。我们必须开始去做，我们使用那些可以为我们所用的 XP 部分。另外，我们也仍极力地想去获得一些其他适合的东西。

charliexia：此外 RUP，XP 等等，所有的这些方法论都是建立在一些 guru 的经验和一些成功项目的基础上的，它们并没有经过科学的方法来进行论证。

markoboger：RUP 也是这样。但它主要是为大型项目所设计的，同时它也是用来为 Rational 公司获取利润的。而 XP 则是为小型项目所设计的，它从一些优秀程序员的痛苦中演变而来。

charliexia：经验是很重要的。但有时别人的经验不一定适合你！

markoboger：对，但只有当你尝试过后才会知道，对不对？所以说你需要获取经验。

timwap_cn：在你们团队中已经实施 XP 多长时间了呢？

markoboger：这是一个持续的过程。

charliexia：我想体系架构并不像 Booch 所说的那样重要！

markoboger：体系架构的确很重要。但是如果你遵循了如 Booch 所建议的那样一个过程，你便无法对架构进行一些改动。而如果你使用 XP，则仍然可以不断地使架构进行演变。

reduce：在 XP 中，客户是作为我们团队中的一部分的，但是当他们和我们的工作出现分歧时，该如何处理呢？

markoboger：你必须要具体问题具体分析。我们也有这种问题，有时候我们的确没有一个客户（或者有很多个客户）。于是我充当了客户的角色。你可以让自己团队中某个人充当客户角色，但是这样会更难以真正地实现客户的需求。

charliexia：请问您能解释一下系统隐喻（system metaphor）和体系架构之间的不同吗？

markoboger：很难对隐喻进行解释。它指的是一种对比，一种描述，与体系架构没有什么关系。

firebird_online：您认为 XP 适合作为科研项目吗？

markoboger：XP 很难用于研究。因为在研究中，你可能无法真正地知道什么是你所期待的，便很难对其编写测试。

firebird_online：但是科研项目没有客户。

markoboger：对，而且你会经常独自工作。但实施 XP 总比什么都不用要好的多。在任何情况下，对于任何类型的项目，你都可以从 XP 中学到很多东西，而你仅仅需要准备用它来适应你的需要。

firebird_online：真槽糕，我经常在科研项目中工作。

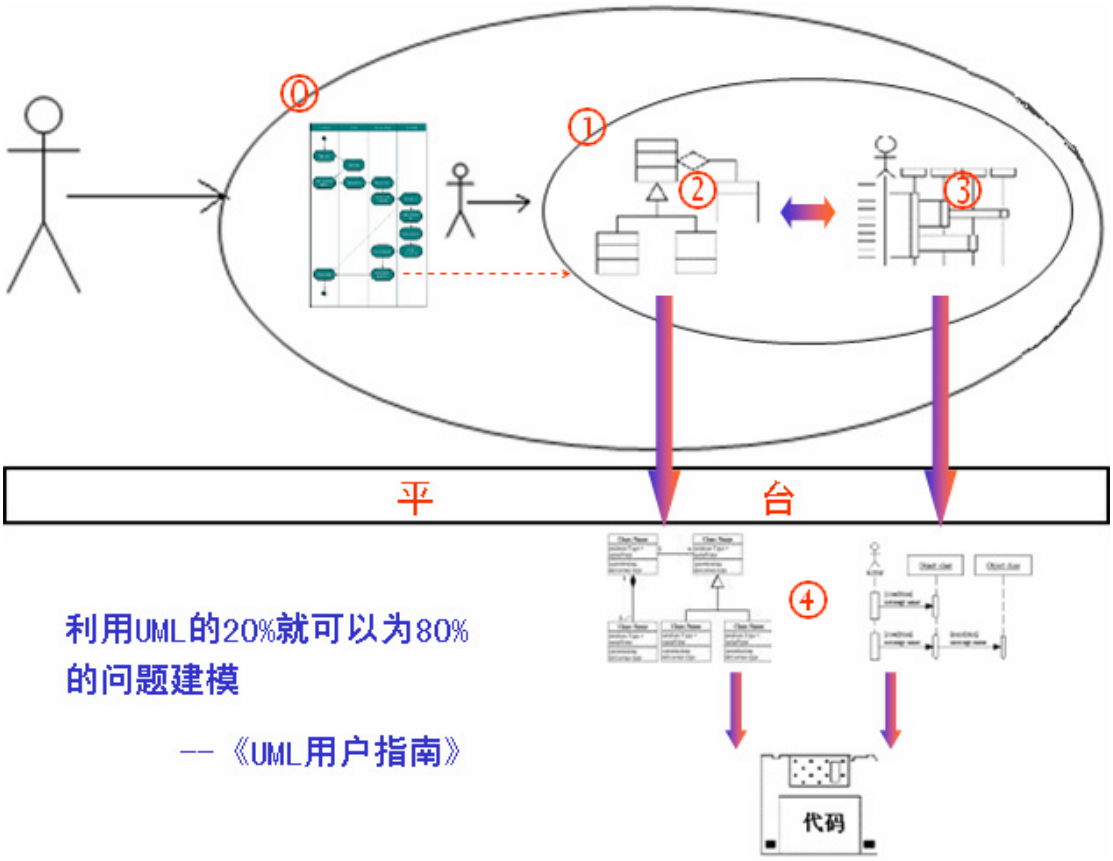
markoboger：我很喜欢研究工作，它可以让你自由地去实验。我想我就是在研究中领会了 XP。没有管理者，没有时间限制...你不断地去尝试并从中学习，这样好像有些不真实，但是对于学习来说却是非常有益的。

reduce：我们在项目中进行用户需求分析时，如果出现了不同的结果，但我们又不能确定哪一个是正确的，请问如何再进行我们的项目呢？

markoboger：这个问题比较难说，我无法回答你。你可以到 www.gentleware.com 上去看一看 Poseidon。

征稿

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

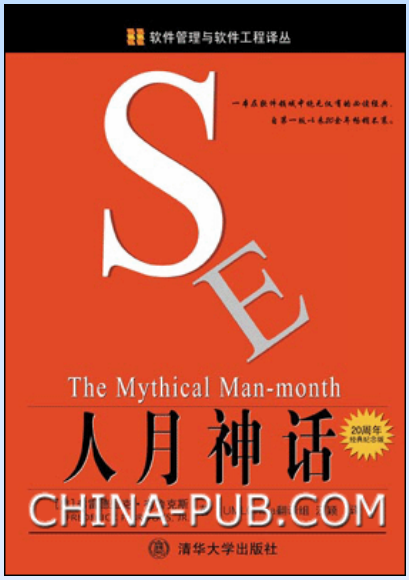


北京公开课，2002 年 9 月 14 日至 15 日

北京中关村 1 号海龙大厦 713-714 报告厅

报名交费请联系：bjcourse@umlchina.com

《人月神话》



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 Adams Wang

散文笔法，绝无说教，大量经验融入其中

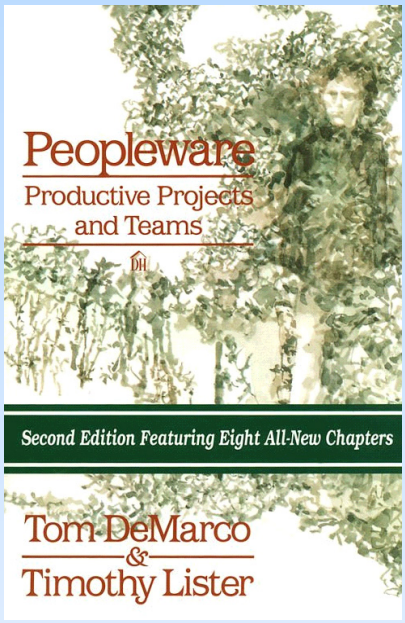
在所有恐怖民间传说的妖怪中，最可怕的是人狼，因为它们可以完全出乎意料地从熟悉的面孔变成可怕的怪物。为了对付人狼，我们在寻找可以消灭它们的银弹。

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

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

中文译本即将发行！

《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

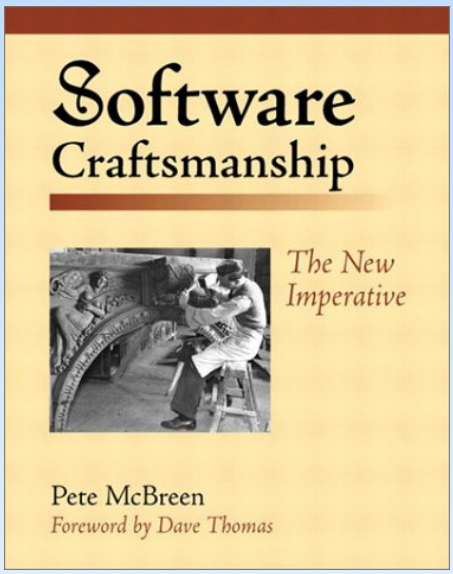
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

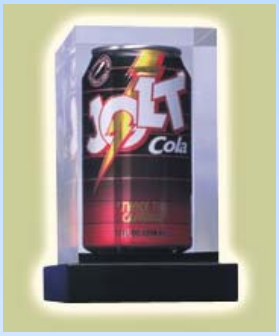
许多开发经理采用了类似的态度，他们竭尽全力地使自己确信没有人能够取代自己。由于害怕一个关键人物要离开，他强迫自己相信项目组里没有这样的关键人物，毕竟，管理的本质是不是取决于某个个人的去留问题？他们的行为让你感到好像有很多人物储备在那里让他随时召唤，说"给我派一个新的花匠来，他不要太傲慢。"

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！

《软件工艺》





2002 Jolt Awards 获奖书籍

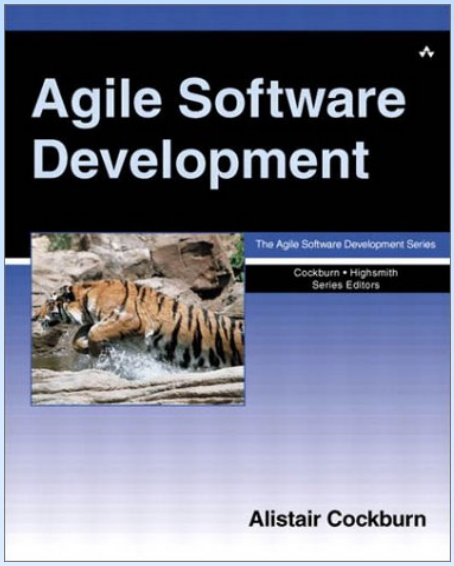
Pete McBreen

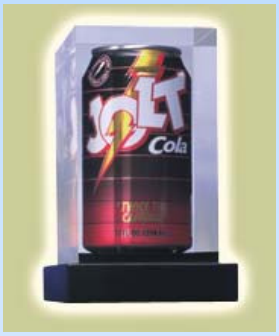
翻译：UMLChina 翻译组

工艺不止意味着精湛的作品，同时也是一个高度自治的系统－－师傅（Master）负责培养他们自己的接班人；每个人的地位纯粹取决于他们作品的好坏。学徒（Apprentice）、技工（Journeyman）和工匠（Craftsman）作为一个团队在一起工作，互相学习。顾客根据他们的声誉来进行选择，他们也只接受那些有助于提升他们声誉的工作。

中文译本即将发行！

《敏捷软件开发》





2002 Jolt Awards 获奖书籍

Alistair Cockburn

翻译：UMLChina 翻译组 Jill

我是一个好客人。到达以后，我把我的那瓶酒递给女主人，然后奇怪地看着她把酒放进了冰箱。

晚餐时她把酒拿了出来，说道，“吃鱼时喝它，好极了。”

“但那是一瓶红酒啊。”我提醒她。

“是白酒。”她说。

“是红酒。”我坚持道，并把标签指给她看。

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

我们大笑，然后回顾我们为了验证各自视角的“真相”所作的努力。究竟为什么，她问道，她已经看过这瓶酒很多遍却没有发现这是一瓶红酒？

中文译本即将发行！

通俗而且深刻的软件开发思想杰作—UMLChina 推荐

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

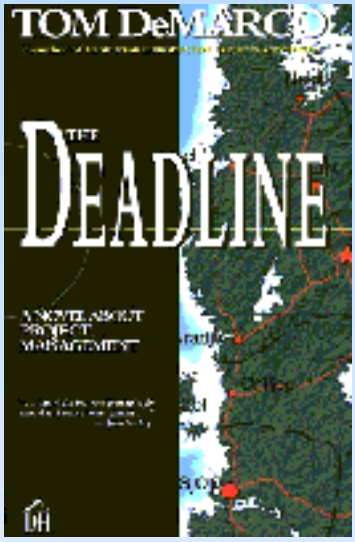
翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

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

中文译本即将发行！

《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。"

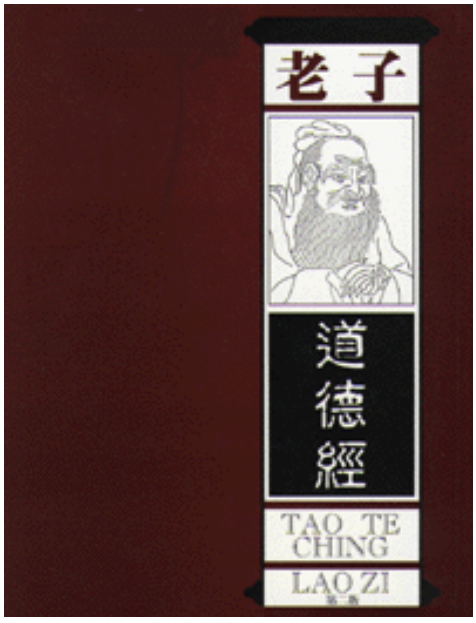
中文译本即将发行！

Peter Merel：妻子告诉我该睡觉了

查看评论

编注：

北京时间 8 月 12 日上午 10:00-12:00，Peter Merel 先生作客 UMLCHINA 讨论组的聊天室。Peter Merel 是构造企业级系统的专家，有 20 年以上的行业经验。曾经在 GMAC, HP, AC Nielsen, Foxboro, Fujitsu, Telstra, Plessey 的项目中担任领导。人民邮电出版社出版的《极限编程研究》（XP Examined）有他的文章“The Tao of Extreme Programming”（极限编程之道）。1998 年 Peter 加入 Ward Cunningham 的 Portland Pattern Repository，并把 XP、RUP、MSF 结合起来，提出了 XUP。以下是交流实录。由 extreme 翻译。原文链接。



Peter Merel 在作演示

umlchina: 你好，教授。

petermerel: 我想说明一点，至少在西方我没有教授头衔。我是一个工程师，大多数人都叫我 Peter.不过，你想怎么称呼我都行。:-)

umlchina: 其余人 10 点钟到。我们现在可以开始了吗？

petermerel: 当然，我们随时都可以开始。到我们本地时间晚上 9 点钟，我可以一直待在这里。也许还可以晚一点，这取决于我妻子的安排……

extreme: 我读了你的文章“极限编程之道”。显然，你对中国文化有相当的了解。

petermerel: 我只了解一点中国文化。很久以前，我曾有机会参加一个澳大利亚国立大学的汉学邮件列表，并且我向 Dan Lushaus 和 Chad Hasen 等人学了一些东西。不过，我从没有到过中国，也不会说中文,可能我知道的关于“道”的东西也许并不正确。我怀疑，对于我在我的文章”极限编程之道”中曲解了老子和孔子, 你们这些家伙感觉有多糟。在这里我感到也许会受到批评。当 umlchina 邀请我到这里来作嘉宾时，我感到非常荣幸。我住在圣地亚哥，在悉尼长大。在澳大利亚我们有一句谚语，“把煤卖给纽卡斯尔”。这里，纽卡斯尔是一个煤炭加工中心。我想，到这里来讲“极限编程之道”可能正是在卖“煤”。我非常好奇，这篇文章是怎样翻译成中文的，还有，对你们而言，这篇文章是不是不正确或者有点傻气？

extreme: 我的问题是关于单元测试：我同意，Xunit 的哲理非常不错，但是，有时我发觉设计测试用例极其困难，尤其是当方法非常复杂，并且要完成很多功能的时候。比如说，向数据库中插入一条记录，我们怎样测试它呢？有没有什么建议？

petermerel: 作数据库程序的单元测试，你通常都需要花一定时间去开发一个适当的和可重复使用的小程序（fixture）.重要的是，不要把这个小程序和验收测试搞混了，那需要专门的测试人员去做。

extreme: 问题是，怎样让它自动化以及花费合理（cost-effective）？

petermerel: 用你的数据库模式（schema）创建一个小的数据库，在设置测试环境时将数据拷贝到你建的数据库中，在这个数据库上进行测试，然后在清除测试环境时，删掉它。你的小程序可以做测试环境的设置和清除工作，这样，许多数据库单元测试就可以以很小的代价完成。抱歉，用了不少”…”,因我的浏览器文本输入框有长度限制。对于 OODB 系统和其它非关系数据库系统，你可以考虑在测试中使用代理对象。我想在 C2 wikiweb 上的一个网页中可能有更多与此相关的信息。

extreme: 你觉得对于一个小项目组，比如说 10 人以内，这样测试的代价会不会太高了？

petermerel: 你指做一个数据库测试小程序？ 我们只是在一对程序员中做这件事。我觉得拷贝一个小数据库，是很简单的事。

extreme: 什么是 C2 wikiweb ？

petermerel: C2 是 Ward Cunningham 的伟大成就。http://www.c2.com/cgi/wiki?FrontPage 是一个很好的入口 。

extreme：比如说插入一条记录，我们通常做的是，打开数据库，看一看刚才插入的记录是否我们所预期的。这样做很简单。问题是，这样的测试不能被重用。

petermerel: 听起来是一个不错的简单测试。你当然会先写测试用例啦。

extreme：如果我们想让这个过程自动化，似乎我们必须做较复杂的编程。

petermerel: 如果你在 setup()中拷贝数据库，运行测试程序，然后在 tearDown()中删掉拷贝的数据库，这样测试就可被重用。

extreme：明白了。那么，你们真的这样进行测试？

petermerel: Extreme, 的确是这样。先写测试，观察它运行失败，然后写代码，观察测试运行成功。如果感觉良好，继续下一个任务。

founder_chen：在你的关于 XUP 的文章中，MSF 指什么？

petermerel: MSF 是"Microsoft Solutions Framework"。这对 GMAC 非常重要，他们的方法在 MSF 的框架中工作。

extreme: 我也读过你写的 XUP 的文章，非常棒。有一个关于文档的问题。在 RUP 中，我们发现文档太多了。我想知道，对于一个 10-30 人的 JAVA 项目组，你认为哪些文档是真正需要的？

petermerel: 当你缺乏业务、工具、架构和其它相关方面的知识的时候，文档相当重要，否则，你只能依靠“现场客户”和经验丰富的开发人员以减少对文档的需要量。

niusun：我谈谈我对 RUP 和 XP 的一些思考，希望你能给我一些建议。RUP 的第一步是根据客户需要找到一个架构。架构解决的是对客户最重要的业务需求和风险。对于业务领域，我想 RUP 处理得非常好。对于应用层，XP 很不错。RUP 适用于开发成熟的系统，而 XP 适用于开发未成熟系统(Premature，意指未定型的系统，如新业务系统-译者)。对于银行、电信、文教，RUP 不错。不过，对于一些新业务，XP 较适合。

petermerel: 在一些大的开发项目，我喜欢 RUP 的某些方面。它在管理大的组织体系和划分开发小组方面非常有用。我不同意 XP 只适合于未成熟系统的说法。对于我来说，这只是一个项目的规模的问题。一个 12 个人的小组可以用 XP 做很多事情，超过 12 人的小组……你需要处理超出 XP 所能考虑的复杂组织体系和工期的问题。我曾经在澳大利亚 Telstra 的一个电信项目中，使用过和 XP 相似的方法。不过，我们有一个好的项目经理，他负责将整个

团队和外界的干扰隔绝开来。

niusun：RUP 预测未来，XP 考虑现在。对于未成熟，XP 很适合。

petermerel: Niusun, 我想这又回到了 Beck 的一个关于驾驶的比喻上来了。RUP 尽一起努力使你的车对准遥远的城市，而 XP 给你的客户一个方向盘（这样，你就可以随时调整方向 – 译者）。

extreme：关于“简单性”的一个问题：做能够工作的最简单的事，那“架构”怎么办呢？如果刚开始时没有花足够时间在架构上，以后你会花多得多的时间，你觉得怎么样？

petermerel: 我一直把自己称作是一个“软件架构师”。在西方，自从 XP 流行起来，软件架构师这个词已开始变得有点贬意，但是，XP 假设你在开始迭代开发前，先要在一个抛弃型原型中（spike）构建系统的架构。问题是，对于你来说，架构是什么？架构可以是工具集，可以是逻辑上和拓扑上的关系，还可以是网络配置……XP 试图让整个开发团队对所有这些东西负责，而且是在开发工作的初期阶段。在项目的后期变化架构和 metaphor 代价昂贵。

extreme：对于我而言，架构是某种至关重要的东西，一种你现在不做好以后会花加倍时间的东西。不错，“拓扑关系”。类、接口之间的关系等等。所以，我们需要在开始的时候就考虑它，而不是让它通过“重构”自然地“进化”出来。

petermerel: 我说的“拓扑关系”意指在整个开发过程中不会改变的东西。通常，一两个 UML 图可以很好地记录这方面的共识。

extreme：那么，我们怎样做“能够工作的最简单的事”？所以，我们应当先考虑 UML 图，然后再编码。

petermerel: Extreme, 不要对此想得太多。对各种可能的方案作简单的抛弃型原型，然后，如果你对其中一个结果较满意，而你又需要和其他人交流，画图。先作原型，达成共识，再画图。不过，让你的图小一点！。你的另一种选择是用 CRC 卡。Beck, Jeffries 和 Cunningham 更喜欢用卡片。我也喜欢卡片，不过，我发现画图更自然。

extreme：我完全赞同你的意见。我发现图更清楚。

petermerel: 我想，架构是图能发挥作用的唯一地方。有谁需要将整个系统的图画出来？你需要能自说明（self-documenting）的源代码，这样，整个系统就没有必要画图了。

extreme：那么，你同意，在开始编码之前，我们仍然需要一些设计方面的考虑。这样做是否违背了 XP 所建议的“简单性”实践原则？在 XP 里，简单性意味着做可能工作的最简单的事，而设计会从编码中自然地演化出来。

petermerel: 我同意，在编码之前，我们需要考虑架构 – 那些在所有迭代过程中不变的东西。但是，设计 - ，不，在 XP 中，设计是通过重构（Refactoring）来完成。最简单的事，是指在迭代过程中做的事。在迭代开始前，你需要选择好“简单性”可以被识别出来的环境。

BirdGu：如果有人需要维护这样一个自说明（self-ducomenting）软件系统，他需要去读所有源代码以得到软件的一个整体视图吗？

petermerel：应该向需要维护软件的人提供足够的关于软件架构方面的背景知识，如一些架构图。系统只是架构元素的混合和表现，而架构是词汇和语法。一旦你了解了这些东西，源代码本身 – 如果它被写成自说明的形式，就完全足够了。

extreme：那么，对你来说，设计和架构是两个完全不同的概念？

petermerel：是的，设计通过重构完成 – 你需要和谐地使用架构中的元素。重构解决"最简单的事"引发的混乱（虽然在这种混乱状况下系统还能工作），让整个系统变成最简单的系统。

extreme：明白了。谢谢。

extreme：关于反馈的一个问题：对于一个外包项目组，客户和项目经理都离开发小组很远，现场客户（Onsite-customer）完全不可能，对于增进反馈或沟通你有何建议？

petermerel：我曾经好几次遇到过这种情况，我并不认为有非常完美的解决方案。最好你的远端客户能指定一个本地的代表，这个人有相关领域业务知识并能定期和客户沟通。在 HP 工作的时候，曾经有一次有三个小组介入了我负责的一个项目，主要的两个在圣地亚哥。除了相当昂贵的电视会议系统的花费外，我们每个月还必须派人飞来飞去。

虫七：当你考虑软件架构的时候，你做什么？

petermerel：我做什么？真是一个好问题。我首先考虑系统的意图（intent）,我试着去搞清楚谁需要它们。然后，我用简单的术语表现这些意图，在其中，我找出问题描述中不变的东西。

extreme：对于一个外包项目组，获得一个本地客户代表是不太可能的。我的情况是，项目经理和客户都在美国，而我们开发小组在中国。

petermerel:这就比较困难了 – 你需要克服国家和公司两方面的文化差异，很容易产生混淆。

niusun：你能解释一下“Spike the alternative”中“Spike”的意思吗？

petermerel：“spike”是一种快速的、试验性的解决方案，其目的是增进你对项目的了解，而不是构造最终要交付的系统，这是一种探索，以确保你能作出正确的决定。

BirdGu：你怎样和开发小组作业务领域和商务概念方面的沟通？

petermerel:我的小组通过于成对开发、站立式会议、迭代计划会议、源代码进行交流。借助于一个现场客户的帮助，问题域和商务概念通过卡片和白板得到交流。

extreme：我想你说的“spike alternative”是指”抛弃型原型开发”,对吗?

petermerel: 是的，Extreme,必须把它抛弃。这种东西几乎从来不实施成对开发，而我们把所有不成对开发的代码都视作可抛弃的。

BirdGu：白板上的东西会被记录吗？如果是，以什么形式记录？UML 或者是 CRC?

petermerel: BirdGu, 如果客户想出了新的定义和变化，白板上的东西就需要记录下来。通常，它们最后变成具体的开发任务(engineering tasks)。对于记录下来的细节，我们使用一个 versioned wiki 来组织，使它们和 CVS 中的源代码保持一致。

extreme：关于成对开发的一个问题：如果我们想从 XP 类型的方法中获益，你认为我们必须采用成对开发吗？在中国，让管理层接受这种概念有点困难。

petermerel: Extreme, 我认为，无论你采用什么方法，成对开发都是一种好的工程实践。不过，对于特别适合一个人单独解决的问题，最好不要使用成对开发。比如一些非常困难的问题或者是发明一种新的东西。成对开发适合于两个搭档都有足够的和问题相关背景的知识的情况。

虫七：Extreme, 我不这样想。在中国，成对开发被广泛接受。不过，我不认为这是一个”必须”使用的方法。

petermerel: 没有成对开发，像 XP 这样很少使用文档的方法要侥幸成功是非常困难的。强烈建议不要这样做（指在实施 XP 时不做成对开发-译者）。先训练成对开发技巧，然后是测试技巧，最后才是 XP 的实践，这就是我试图在“极限编程之道”中描述的东西。

extreme 对虫七说：技术人员也许会接受，管理层则有点困难。

petermerel: 至于说管理层，他们是不可或缺的恶人（necessary evil）。没有管理层承诺的支持，你不可能实施任何 XP 的实践。缺乏管理层的支持，也许短时间内不会有问题，不过你最终一定会失败。说服管理层，可以先小规模地演示，用一两对程序员做一两个星期，然后解释，如果继续下去，你将怎样提高生产率和降低风险。但是，对于很多经理来说，这仍是件很可怕的事情，毕竟，他们自己也需要向高层经理解释。

extreme：明白了。关于项目经理的一个问题：对于一个人数不超过 30 人的团队，你认为项目经理的职责是什么？这个职位必须要有怎样的背景？在这种环境下，项目经理需要是一个技术专家吗？

petermerel: 我想，一个项目经理应同时具备技术和业务领域的背景，这样，他就能很好地解决开发者和客户之间沟通方面的障碍。不过，这取决于组织的架构。在“极限编程之道”中，对于项目经理这个角色我没有作更多阐述，因为对我而言，界定教练（coach）角色更为重要。就算项目经理很差劲，你仍能完成不少事情，但一个差劲的教练则会毁掉整个项目。

extreme：不错，项目经理应该懂技术。我的问题是，对于小项目组，他有没有必要成为首席专家？

petermerel: 不，Extreme,但是教练必须受到技术人员的普遍尊重。

extreme：教练在这似乎是一个首席专家，一个导师？这样，我们就有两个关键角色，一个是项目经理，一个是教练？

petermerel: 没有人在各方面都是专家。教练必须熟悉各种开发规范，并且知道怎样充分发挥整个团队的专业技能。XP 中有三个重要角色。功能性的角色-也就是，一个人可同时担任几个角色，这样做虽然困难，但不是不可能。这些角色是教练(coach)、记录者（tracker），和经理。经理代表管理层，教练专注于技术和内部问题交流，记录者保证上述两个角色知道项目的进度和资源使用等各方面的相关情况。

虫七：(My name is the7thworm in English.) 我仍在思考你关于架构设计的回答...在你把软件蓝图确定后，你考虑软件的层次吗？如果有些东西被其它模块共享，或者有一些特殊的东西，你怎样组织？

petermerel: 7th worm, 我不能确定你讲的"层"和"蓝图"指什么。你能说得更清楚一些吗？架构在模块、任务、开发者之间共享。这是在所有迭代过程中不会变化的东西，如数据库的选择、语言、网络配置、 软件配置管理、源代码结构...它为其它东西提供了一个背景（context）。

BirdGu：那么，客户又怎么样呢？次要的吗？

petermerel: 客户绝非不重要。现场客户也可视作是一个功能性的角色 – 不过，一个人要既是一个客户同时又是一个开发者是非常困难的。成为一个好的 XP 客户，也需要一个学习过程。我想，客户需要教练提供的帮助几乎和开发者一样多。

niusun：XUP 的定义是什么？比如，RUP 是用例驱动、以架构为中心、迭代和增量。

petermerel: 在 XUP 中，我试图通过映射角色在 RUP 和 XP 之间架起一座桥梁。对某些人来说，工作起来还不错，不过，我把它当作一个实验。如果要在实际项目中实施，你要自己承担风险。

extreme：有没有任何实际采用了 XUP 的案例？有没有相关资源？我注意到，在 XUP 中，你加入两个新角色，一个是教练，一个是记录者。

petermerel: 曾有一些人写信告诉我他们在尝试使用 XUP，并且很喜欢这种方法。我没有听说过不好的报告。我在 GMAC 工作期间，GMAC 内部的一些企业架构设计师在使用它去制定一些内部标准。XUP 也曾在 Omnigon 内部使用过，我记不起来有没有其它地方也用过它了。我主要在 GMAC 内部用它作为一个框架，在一个高级管理层可以接受的环境里。

extreme：什么是 GMAC ？

petermerel: GMAC 是通用汽车的一个部门。三年多前，我在那里的金融部作“教练”和架构设计师。

extreme：高级管理层？出于沟通的需要吗？

petermerel: 高级管理层=你的经理的经理。我想，一个人可以身兼两职。我想把人和角色区别开来，也就是说，团队中的角色、组织中的角色，是指工作的头衔。

extreme：那么，你认为一个人能同时作项目经理和架构设计师吗？有必要吗？可能吗？

petermerel: 至于说是否必要，这取决于具体情况。有时，只有这么多资源，或者只有这么多的专业人员，你只能根据人员的实际情况作安排。

extreme：对小项目来，也许还能行。对大些的项目，我想比较困难。

petermerel: 是的，Extreme, 我是这样想的。XP 的确需要有人去协调客户和开发者之间的关系，去处理报告、预算、资源等问题。尤其在一个大的组织中，项目经理至关重要。

虫七：我读了（“道”文），我觉得是篇有趣的文章。你对中国文化的了解让我惊讶。

petermerel: 我的中国文化的知识都是二手货。我曾经两次和一个名叫 Ian Dunn 的汉学家一起共事。他也是个工程师，经常到中国访问。在西方，多数工程师都有兴趣尽量扩大他们的知识面。我们鼓励多角度看问题，这样，遇到我们不知道的东西时，不会感到惊讶，至少，不会过于惊讶。只有日文版的翻译我较深介入过。译者和我交流了好几个月，这是段令人愉快的经历。听说已经有中文译文我很吃惊，我倒想能和这个译者谈谈，看看他翻译时会遇到什么困难。也许一整天都行。

extreme：所以，去年我到新泽西工作时，我到教堂去过好几次，想更多地了解西方文化 。

niusun：XUP 的方法论是什么？

petermerel: 也许 Rational 的网站是最好的起点。我曾花过一些时间研究过 RUP，不过却很少真正用它来工作。RUP 基于一种传统的、重型（heavy）的方法论，幻想对项目的管理进行控制。如果能不用它，我就不会用它。

cinc：一个程序员如何估计他的进度呢？我想，这对每个程序员都很重要。

extreme：我有和 cinc 同样的问题。你有没有关于怎样衡量程序员工作的建议？

petermerel: 对于每个程序员来说，坚持以最佳工作日估计和跟踪自己的工作，这是一个预计项目开发速度的好方法，也是一条很好的个人纪律，不过，XP 想跟踪的是整个项目组的进度情况。在估计和测量之间有众多变数，项目成员之间也互相影响，所以，我们并不尝试去发现单个开发人员的工作负荷（load factor）。Extreme, 你通过完成的单元测试的数量和源程序中删除的代码行来衡量程序员的工作。

niusun：你能介绍一些工具吗？

petermerel: 什么工具？

niusun: 可以帮助我使用 XUP 的工具。

petermerel: 我知道的最好的工具是 3＊5 的卡片，以及白板、CVS，我们也使用一些表格。

虫七: 我读过一篇关于 Adaptive 方法和 XP 方法比较的文章。作者说，XP 开发者必须在同一个地方，否则就无法实施。

petermerel:我想很多 XP 的实践都可用于地理上分布的团队，尤其是先测试、YAGNII 、重构甚至某种程度的成对开发。但是，要完整实施 XP，你的确需要在同一个地方。XP 进行得很快，你需要人们面对面地工作，以保持项目良好的状态。我同意，若不在同一个地方，XP 几乎无法工作。在 Omnigon，我们很小心地给每个团队和每对程序员分配自己的工作空间，成对开发通常很吵。

cinc: 但是，当程序员分布在不同地方，怎样进行成对开发？

petermerel: 你可以使用网络电话和电子白板。使用 CVS 去管理源程序，制定明确的准则以确定谁主导程序开发过程，以及测试、编码或者重构。

extreme: 被删的代码行？

BirdGu: 被删除的代码让代码干净且简单，对吗？

petermerel: 是的，删除。如果你能删除一些代码行，而所有的测试仍然能通过，你就是在做有用的工作。

ozzzzzz: 我想了解一些关于重构的东西。

petermerel: 重构使整个系统易于维护，是编码完成后的设计过程。

虫七: Ozzzzzz, 你可以在 Amazon 买一本 Martin Fowler 写的《代码重构》。

petermerel: 7th worm, 我想 Fowler 的书很棒。他最精彩的观点是，代码重构应该一点一点进行。我们说：“温柔地，温柔地”。

niusun: 在设计软件架构的时候，我发现很难决定何时该停下来。能给我一些建议吗？

petermerel: 你应当使架构越小越好。这是一个贯穿整个应用的主线，就像一个字母表。当团队中的开发者同意已经有足够的”字母表”时，就可以开始迭代开发过程了，不过，这里要小心，一定要先达成共识。

虫七: 我认为好的方法并不是万能钥匙，对吗？我能享用 XP 建议中的一些好东西，事实上，我们正这样做。

petermerel: 7th worm, 当然没有万能药。相反，我们这里有这样一种说法，“所有万能药都会变成毒药”。XP 中，有一条规则，“他们只不过是些规则”。但是，约定（convention – 意指规则等 – 译者）提供了一种环境，这种环境使工作进行得更快。

cinc: hehe, wiki 是一个好工具，CVS 也是，我们通过 MSN 和其他人交流。

petermerel: MSN 的确有用，不过，我更喜欢免费软件。

extreme: 我想，MSN Messenger 是免费的。

petermerel: MSN messenger 是免费使用，我更喜欢免费的软件许可证，有很多方法可以进行即时通信 ...

cinc: 在 XP 中，我们怎样将需求文档化，仅仅使用用户故事（user story）？

petermerel: 在 XP 中，需求就是用户故事（user story），它们会被分解成任务。

BirdGu: 和用例（use case）比，用户故事（user story）有何优点？

petermerel: 那么 ... 用户故事和用例：一个用户故事封装了一定范围的用户“意图”（intent），用户故事有定义好的工作粒度，他们还有相对的优先级。故事描述用户意图，而并非使用的场景...它们也定义工作的粒度（以最佳工作日估计），数量方面变数（多少，频度以及速度），还有相对的优先级。

BirdGu: “意图”和“场景”之间的什么区别？

petermerel: 意图”是一种需要-这是用户需要的东西，而不是用户怎样和系统交互的过程，”场景”则描述用户如何与系统交互。

extreme: 那么，即使我们有了故事，仍然需要用例？

petermerel: XP 中，一个用例主要通过 CRC 卡片游戏来完成。你没有必要文档化一个用例。很久以前，我们就已经不这样做。CRC 卡片让我们对项目获得共识，而不用花很多时间去画图，以后又花更多时间去向经理证明，我们可以将它扔在一边。:-)

BirdGu: 故事能用于描述非功能性需要吗？比如安全、性能等。

petermerel: 如果结果中包含特殊的定义和参数，的确如此。非功能性需求和每个故事相关联。XP 的计划过程分解和安排故事以使它们有合适的粒度。

cinc: 对不起，不需要附加文档如 UML 图吗？

petermerel: UML 可以非常方便, 但是，我很少自找麻烦去画 UML 用例图。我更喜欢故事和 CRC 卡。

petermerel: OK,各位朋友，我的妻子告诉我该睡觉了。最后一个问题，谁来？

cinc: 能告诉我们有哪一个 CRC 卡工具可以免费使用的吗？

petermerel: CRC 卡片仅仅是一些纸，3＊5 规格。你要能够把它们撕了或者扔掉，将它们发给其他人，经常，你需要能在手里把玩它们。在卡片游戏中，当某些人将卡片传来传去，你可以知道，他们真正理解了卡片上的东西。如果他只是坐在那里观望，那就意味着他们没有懂。这里的要点是，通过卡片游戏，设计思想得到了戏剧化的表现，所有人都会记得很清楚。就好像看一场戏，所有人都知道，剧情是怎样发展的。

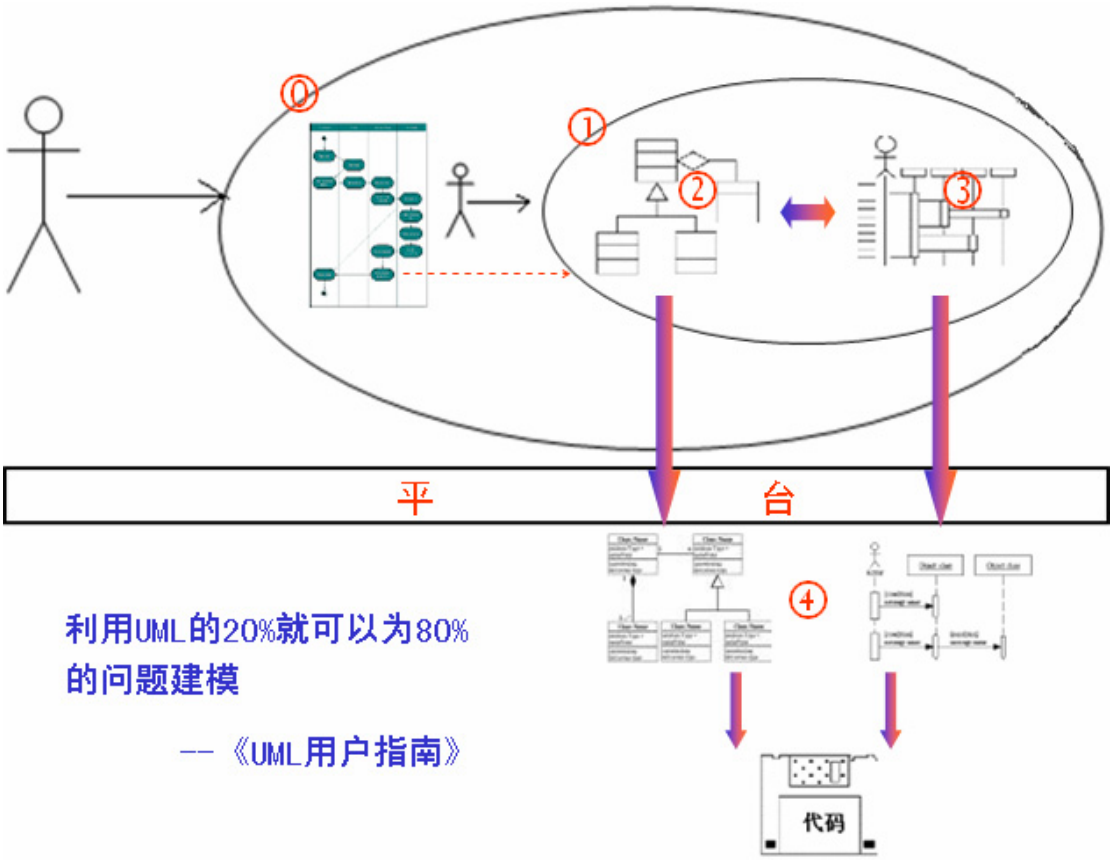
cinc：纸片不易长期保存，你可能会将它丢了。有没有什么可以在电脑中使用的工具？

petermerel: 你没有必要保存它们。它们很快就会过时。更多的意见，请读“极限编程之道”:-)

petermerel: Okay, 该睡觉了。谢谢大家。

征稿

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



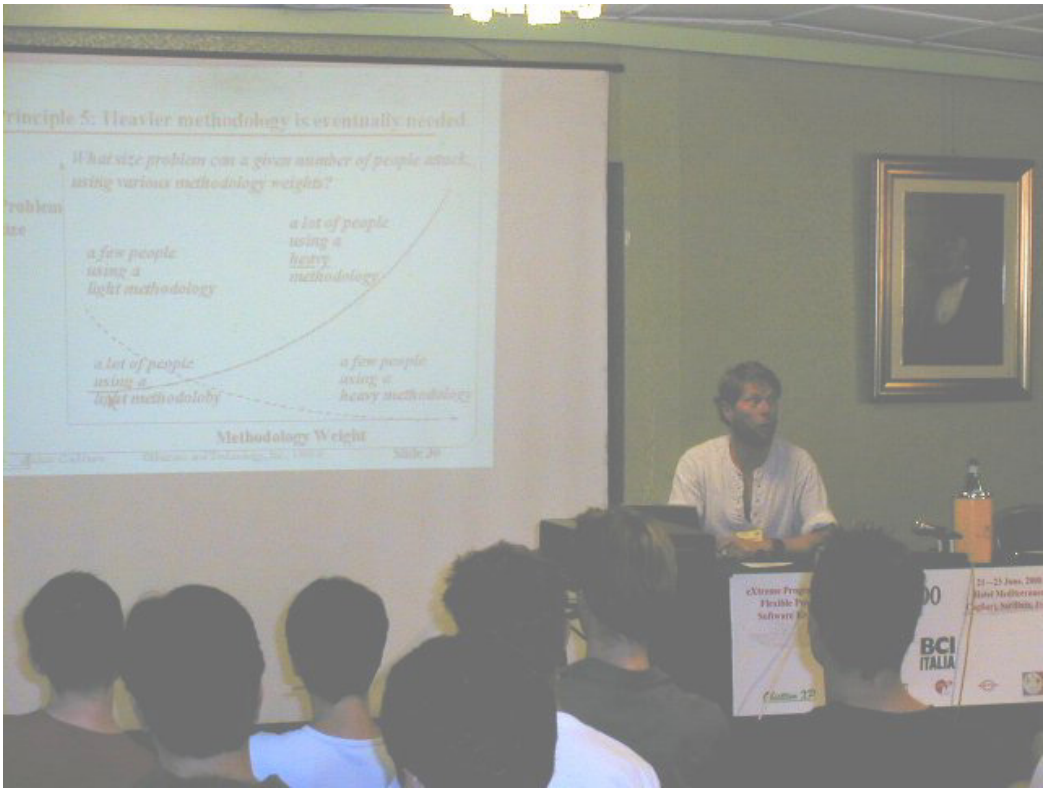
北京公开课，2002 年 9 月 14 日至 15 日

北京中关村 1 号海龙大厦 713-714 报告厅

报名交费请联系：bjcourse@umlchina.com

Martin Fowler

Jutta Eckstein



Alistair Cockburn

用户界面的 UML 建模

Paulo Pinheiro da silva, Norman W. Paton 著, 李巍 译

查看评论

摘要

统一建模语言（UML）是对应用程序进行面向对象建模的标准标记语言（notation），因此我们会很自然地将其作为用户界面（UI）建模的选择。但是，我们并不清楚如何使用 UML 来进行 UI 的建模。本文给出了一个使用 UML 进行用户界面建模的案例。该案例指出了那些无法使用 UML 标记来进行建模的 UI 侧重面，以及一组可用于 UI 建模的 UML 构建法（constructor）。其中的建模问题说明了使用 UML 进行 UI 建模的一些缺陷，而采用的这组构建法也同时表明了一些该方面的能力。这些被证实的能力和缺陷可作为公式来表达成一种对 UML 进行扩展（extend）的策略，从而对用户界面的设计提供更好的支持。

1 介绍

UML[10,2]是软件应用程序进行面向对象建模的标准语言。用户界面，作为大多数应用程序的一个重要部分[17]，也应该采用 UML 来进行建模。事实上，UML 会很自然地被用作 UI 建模的标记语言。然而，我们一直就不清楚如何使用 UML 来进行用户界面的建模。而对于像用户任务（user task）和表现层（presentation）这样的用户界面元素，是如何在 UML 应用模型中得到支持的，这一点并不能很容易进行标识。而且很少有报告能够很明确地描述一个应用 UML 来建模 UI 的项目。此外，许多 UI 设计过程中出现的建模问题，例如本节所描述的案例，并没有全部在基于 UML（UML-based）的设计方法或 UML 规约（specification）中予以提及（address）。

许多已出台的提议（proposal）使用了一些不同的标记，用来支持 UI 元素的设计模型。例如，关于用户任务（user task）设计方面的研究，可参见 Kirwan 和 Ainsworth 的书[14]以及 Johnson 的书[13]。另外，还有一些提议使用其所公布的模型来进行 UI 设计，具体描述可参见 Griffiths 等人的书[9]和 Szekely 的书[19]。因此，最好的情形是，如果目前已有的建模方式能够有效地用来构建 UI，那么就不必再发明一种新的方法。进一步说，一种好的方法就是无论对于 UI，还是应用程序的其它部分，我们都能使用相同的方式来进行构建。的确，使用单一的标记，对于整合一个面向对象的用户界面的全部设计是非常有用的。这个领域同样也有许多的研究成果，例如

Kovacevic[15]，但是仍然未能很清楚地标识出哪些 UI 侧重面可以使用 UML 来进行描述，而哪些则不能。

本文旨在概括性地描述一个较为全面的用户界面建模的 UML 案例。该案例需要说明以下两个意图：（1）使用 UML 时的一般 UI 建模问题；（2）应用程序开发者在进行 UI 设计时，可以使用的一组 UML 构造法和 UML 图。从这些建模问题中，我们可以看出哪些 UI 侧重面（aspect）是在 UML 中所没有涉及到的。从这组构造法中，我们也能看出哪些 UI 侧重面已经包含在 UML 中。因此，该案例提出了一种简易的方式来使用 UML 进行 UI 建模。另外，为了对用户界面的设计提供更好的支持，本文还描述了哪些元素可用于开发一个对 UML 进行扩展的策略。

该案例有意地没有描述任何一个在 UI 设计过程中使用的方法。的确，为了克服已知的难点，许多方法可以作为一种方式来考虑，并以此找出可供选择的建模方法。这里指的是仅使用 UML 来解决那些已确认的 UI 建模问题。

该案例只考虑基于窗体（form-based）的用户界面。事实上，一些比较重要的用户界面种类，例如数据库系统的 UI 和 web 应用的 UI，都主要是基于窗体的。基于窗体的用户界面的局限性则是指，像游戏，文字处理软件以及仿真器等应用程序的用户界面并不包含在该案例中。尽管如此，本案例介绍的这个 UI 模型可作为一个 UI 模型基线（model baseline），来开发基于 UML 的更全面的 UI 模型。诚如该案例所考虑的那样，对于大多数 UI 设计者而言，对可视化组件（窗口组件）的使用进行建模，要比对窗口组件本身进行建模显得更为重要。因此，该案例论述的是窗口组件如何被应用程序使用，而不是如何对窗口组件进行建模。

2 案例：图书馆系统

使用一个关于图书馆的案例，来标识用户界面的建模问题[9]。该案例中的图书馆系统已经进行了简化考虑，用以捕捉那些在用户界面建模过程中需要面对的实际问题。的确，与真实的系统相比，该图书馆系统要简单许多，但其复杂度也足以确认一个在用户界面建模领域内能够标识出的问题的范围。

图 1 所示的用例图（use case diagram）表明了图书馆系统的角色（actor），以及它们的用例（use case）。角色有 *Librarians（图书管理员）*和 *Borrowers（借阅者）*。角色 *LibrarySystemUser* 是 *Librarians* 和 *Borrowers* 的泛化（generalisation）。图书管理员使用图书馆系统来对书目和借阅记录进行管理。当图书从借阅记录管理系统中登入（check into）或登出（check out）时，图书管理员仅需要通知图书馆系统。因此，创建与 *Librarian* 相关的用例 *BorrowBook（借阅图书）*和 *ReturnBook（返还图书）*。*LibrarySystemUser* 可与图书馆系统建立连接，并列出一个图书馆用户所借阅的图书清单，以及根据图书的作者，名称，年份或者这些信息的组合来进行检索。所以，与 *LibrarySystemUser* 相关的用例有 *ConnectToSystem（连接系统）*，*ListBooksBorrowedByUser（列出用户的借阅清单）*和 *SearchBook（检索图书）*。用例 *ConnectToSystem* 是指当一个 *LibrarySystemUser* 在登录系统时所进行的密码检查。

角色 *Borrowers* 可以浏览书目而无需指定任何条件。因此，用例 *BrowseBooks（浏览图书）*是与 *Borrowers* 相关联的。检索和浏览操作可以在上一次检索或浏览的结果上重复进行。

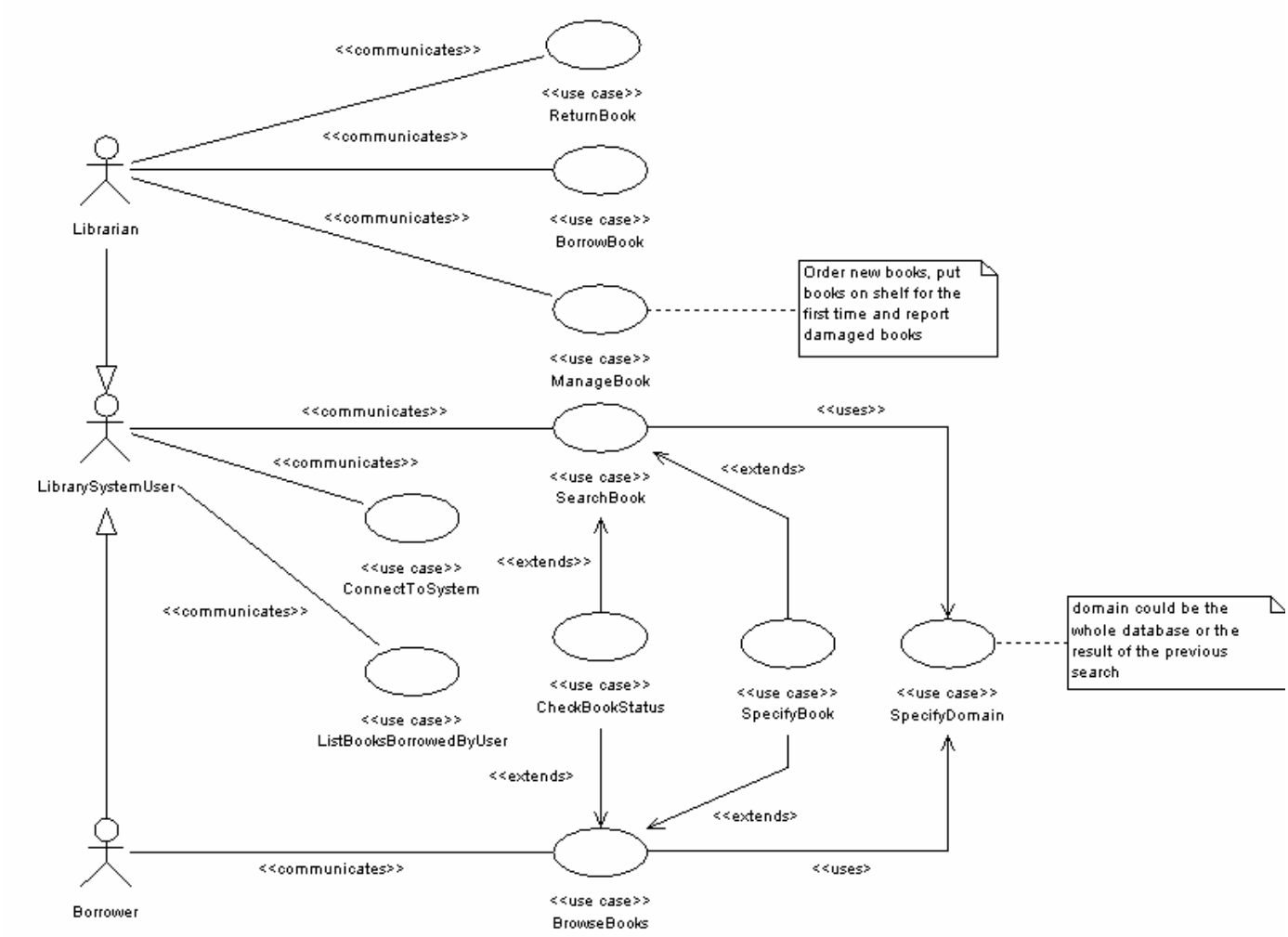


图 1：用例图

图书馆系统必须保证只有已注册的用户才能对系统登录。进一步而言，无论是对于浏览者，还是图书管理员，系统必须确保只能执行与其相关的服务。

用户在检索或浏览书目时可以选择图书。一旦选定一本图书，用户便可以检查该书的可用状态（availability）。用例 *CheckBookStatus* 可作为 *SearchBook* 和 *BrowseBooks* 的一个扩展（extension）来进行建模。实际上，用户在进行状态检查前，必须通过以上的一个操作来选定一本图书。

一些用例具有相似的行为特性。例如，*BrowseBooks* 和 *SearchBook* 是两个都要进行指定图书的用例。所以创建一个名为 *SpecifyBook（指定图书）*的新用例来对这种公共的行为来进行建模。创建一个单向关联关系（unidirectional associations）来对 *SpecifyBook* 和 *BrowseBooks* 之间，以及 *SpecifyBook* 和 *SearchBook* 之间的《extends（扩展）》关系进行建模。同理，*SpecifyDomian（指定*领域）也是 *BrowseBooks* 和 *SearchBook* 所共有的行为。

从用例图以及在本文中没有进行完全描述的系统规约中，我们可以获取到关于领域模型（domain model）的设计，其具体类图（class diagram）可参见图 2。该类图包括以下实体类（《entity》class）：*Person（人）*，*Librarian（图书管理员）*，*Borrower（借阅者）*，*Book（图书）*，*Loan（借出）*和 *StockItem（藏书项）*。前三个《entity》类分别对应着（correspond to）角色 *LibrarySystemUser*，*Librarian* 和 *Borrower*。

一个 *Book* 实例指的是图书馆书目中的一项。为了对图书的藏书进行管理，图书馆系统使用 *StockItem* 类表示图书馆所藏图书的副本信息。然而有时一些图书可能列在图书馆的书目上，却没有在图书馆里进行收藏。例如，在新定购的图书尚未送到时，或者当图书被损坏时。而实际上图书在定购时便会向图书馆的书目中插入相应的记录。所以,此时将会创建一个与角色 *Librarian* 相关联的用例 *ManageBook*。

一个 *Loan* 实例在用例 BorrowBook 所建模的过程中被创建，在用例 ReturnBook 所建模的过程中被销毁。一个 *Loan* 对象实质上指的是一本图书应返还的日期。

本文自始至终地使用了《entity》,《control》和《boundary》三种构造型（stereotype）。它们是由 Ivar Jabcoson 在其《Object-Oriented Software Engineering》[12]的书中提出的，后来被并入到 UML 中。构造型《entity》用于标识那些对单独存在的事物或对象来进行建模的类及类实例。构造型《control》用于标识那些执行系统行为的类及类实例。构造型《boundary》用于标识那些处理系统用户和系统间交互的类及类实例。

该案例还对引入 UI 设计的环境进行了描述。

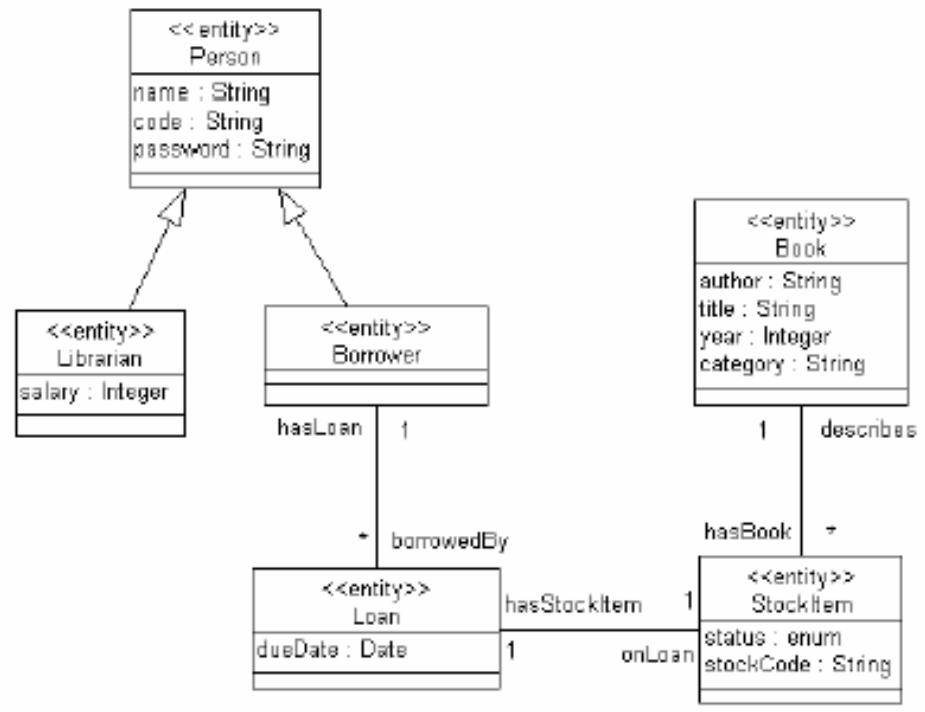


图 2：领域模型

3 任务建模（Task Modeling）

图 1 所示的用例 *BrowseBooks* 表明一个借阅者可以浏览图书。但是，借阅者必须登录才能执行其它任何功能。若使用 UML 术语即指，角色 *Borrower* 只能在之前使用了用例 *ConnectToSystem 的情况下*，使用用例 *BrowseBooks*。事实上，在这种情况下稍微有些复杂。一个借阅者在使用 *ConnectToSystem* 前并不意味着他/她登录进了系统。比如当借阅者尝试去登录却返回失败时，便会发生这种情况。同样地，用例图还表明了用例 *CheckBookStatus* 扩展了用例 *SearchBook* 和 *BrowserBooks*，但是并没有解释该情况会如何发生。

以上所描述的问题是指，用例图是为需求分析进行设计的。但是它们并不提供与任务相关的控制流（control flow）的信息。如图 3（a）所示的活动图 (activity diagram)表示了一个借阅者如何与图书馆系统的用户界面进行交互（interact）。我们可以注意到，活动图并不等同于用例图，活动 *Log in（登录）*对应着当系统用户成功登录时所使用的用例 *ConnectToSystem*。该活动图还表示了一个用户在登录系统后，需要选择下面的一个选项：检索一本图书，浏览图书，或者退出与应用程序的交互。而且，用户只有通过活动 *Select book（选定图书）*来选定一本图书后，才能检查该书的状态。

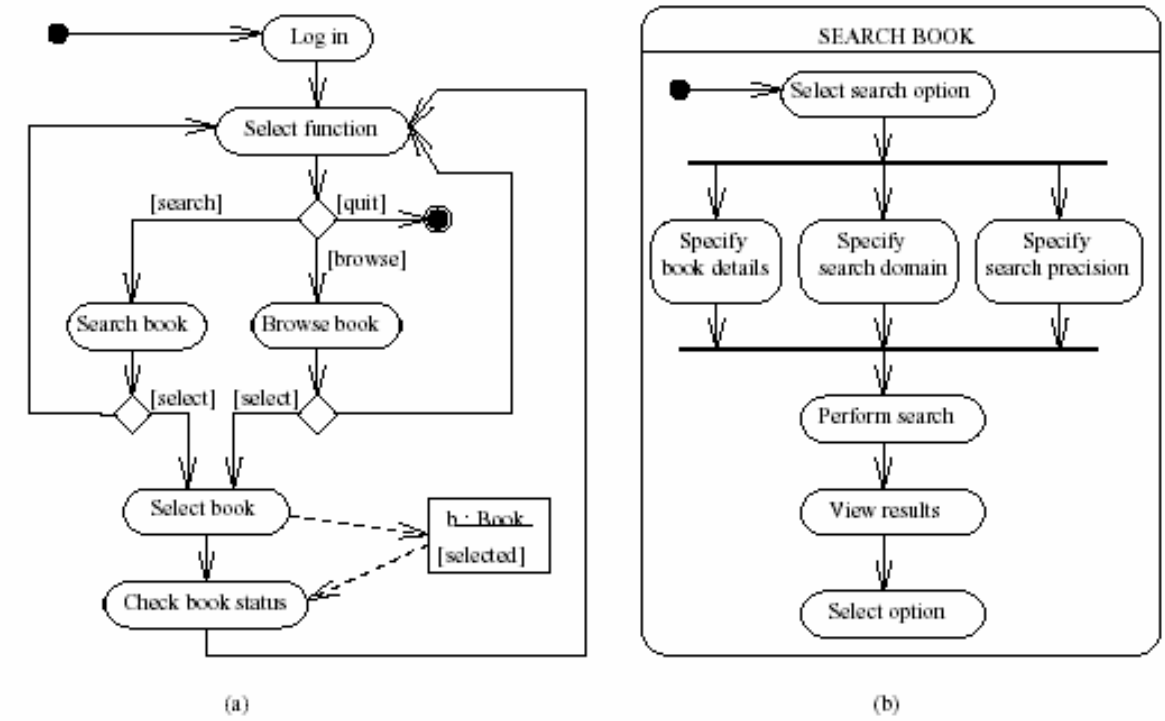


图 3：任务模型的部分视图

像传统的层次任务分析（HTA，Hierarchical Task Analysis）[13,14,4]一样，使用活动图来控制用户界面的导航，已被广泛用来描述用户任务模型（user task model）。活动（activities），用例以及任务（task）并非完全等同，尽管它们具有相似的特征。例如，可以把用例考虑为高层次的任务，而有些活动往往类似于任务。然而，用例和活动之间的关系在 UML 中并没有特别地进行区分。实际上，用例不提供经常与用户需求相关联的一些特性，如目标（goal），前置条件（pre-condition）和后置条件（post-condition），但是这些特性对于活动图的设计却是很有帮助的。（？？）

一个单独的活动图能够对整个任务流控制进行建模，但是活动还能够进行分解。事实上，活动并不是原子的，这就意味着它们在执行一段时间后可以被中断[2]。利用这种可分解性，图 3（a）中的活动 *Search book* 可另外作为一个附加的活动图，来进行更精确地说明，如图 3（b）所示。

许多任务需要来自领域模型的信息，及用户提供的信息[8]。例如在图 3（a）所示的活动图中，一本被选定的图书需要从活动 *Select book* 传递到活动 *Check book status*。这种活动图中的数据流（data flow）可以使用对象流（object flow）来进行建模。如图 3（a），活动 *Select book* 标识了 Book 类的对象 b 被传递到活动 *Check book status* 中去。这些数据项也可在与用例相关联的交互图（interaction diagram）中进行获取。然而，活动图所提供的对象流技术，避免了通过检查交互图来发现任务是如何进行信息访问的必要。

4 抽象表示层建模（Abstract Presentation Modeling）

在进行应用程序建模时，很自然地会需要对 UI 表示层进行建模。甚至是对于非常简单的场景（scenario）而言，UI 表示层部分的建模都是必不可少的。在这个阶段我们并不需要一个详细的 UI 表示层模型，只要能够了解到 UI 是由哪种组件（components）组成，共有多少组件，以及它们是如何分组的即可。我们还需要知道这些 UI 元素具有哪些操作。因此，我们需要一个*抽象表示层模型（abstract presentation model）*。

可以通过对一个用户成功登录图书馆系统的建模，来举例说明如何使用一个抽象表示层模型。图 4 所示的便是用例 *ConnectToSystem* 在该场景下的序列图（sequence diagram）。根据图 1 所示的用例图，*ConnectToSystem* 是与《actor》*LibrarySystemUser* 相关联的。因此，本次交互由 *LibrarySystemUser* 发起，然后发送一个消息（message）至 *Library System* 实例，该实例代表的是整个图书馆系统。

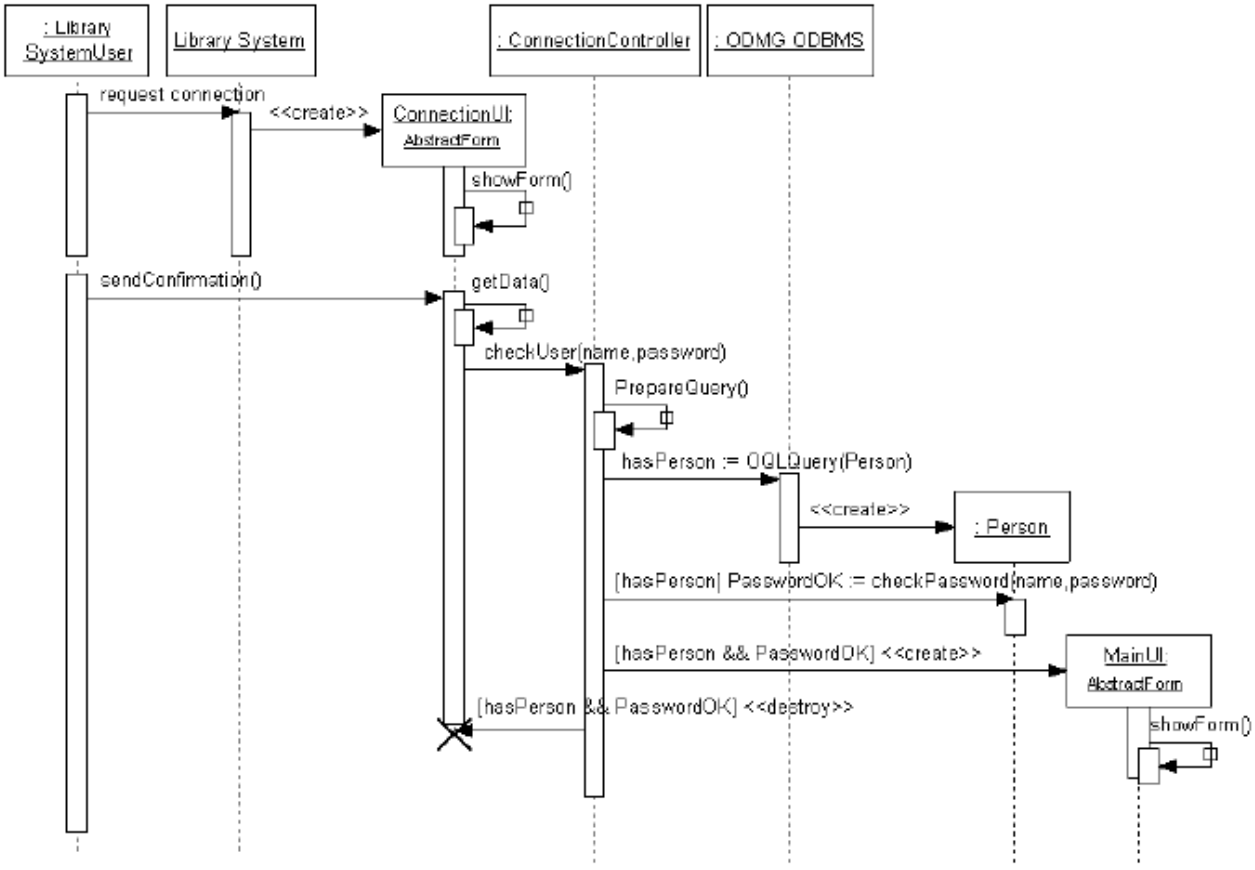


图 4：用例 *ConnetToSystem* 的序列图

从实际的角度来说，比如在 Windows 环境下鼠标双击图书管理系统的图标，将会产生 *request connection* 消息。系统会立刻创建一个 *ConnectionUI* 类的《boundary》对象，该对象执行一个名为 *showForm()*的操作。可以将对象的创建建模为一个对象向新建对象发送《create》消息。*ConnectionUI* 对象一经创建，便会向用户显示为一个建立连接的用户界面，并请求输入登录名称和密码。其类似的用户界面可参加图 5 所示。图 5 并非一个 UML 图，这是因为在 UML 中并没有规定任何关于 UI 表示层设计的标记。事实上，我们并未要求 UML 应该具有关于 UI 模型的标记，从而提供 UI 布局（layout）和组件选择方面的支持。但是，我们通过讨论认为 UML 需要一种标记来更好地描述抽象用户界面的结构，而非类图和对象图。事实上，这样的标记可用在 UI 设计的初期，甚至可通过使用活动图来支持任务的设计。

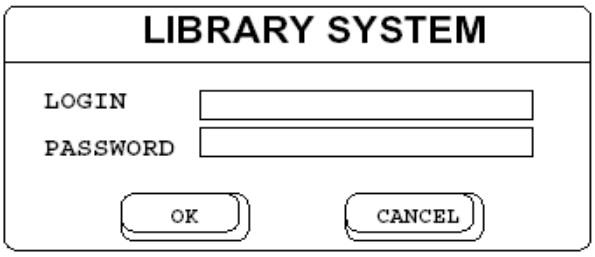


图 5：*ConnectionUI* 的显示

关于 ConnectToSystem 的序列图还需要进行一些辅助的定义，这将会在后续章节中予以说明。

4.1 抽象表示层的结构（Abstract Presentation Structure）

如图 6 所示，抽象表示层模型具有一个顶层的容器（container），《apm》*AbstractForm*，其包含了许多组件，《apm》*AbstractComponent* 以及其它的容器（即《apm》*AbstractContainer*）。事实上，容器提供了关于 UI 表示层元素的一种分组机制。《apm》*AbstractComponent* 指的是一个广义上的（generic）抽象组件。参见图 6，可专门（specialise）将 *AbstractComponent* 分成三类（categories）：*StaticDisplay*，*ActionInvoker* 和 *InteractionControl*。构造型（stereotype）《apm》标识的是抽象表示层模型类。

- *StaticDisplay* 类是与那些提供一些可视化信息的组件相关联的，如标签（labels）。
- *ActionInvoker* 类是与那些能够接收系统事件并作为系统操作来进行传播的组件相关联的，如按钮（buttons）。
- *InteractionControl* 类是与那些能够接收系统事件的组件相关联的，其一般是用来建模那些作为 UI 导航的用户选项，如菜单（menus）。

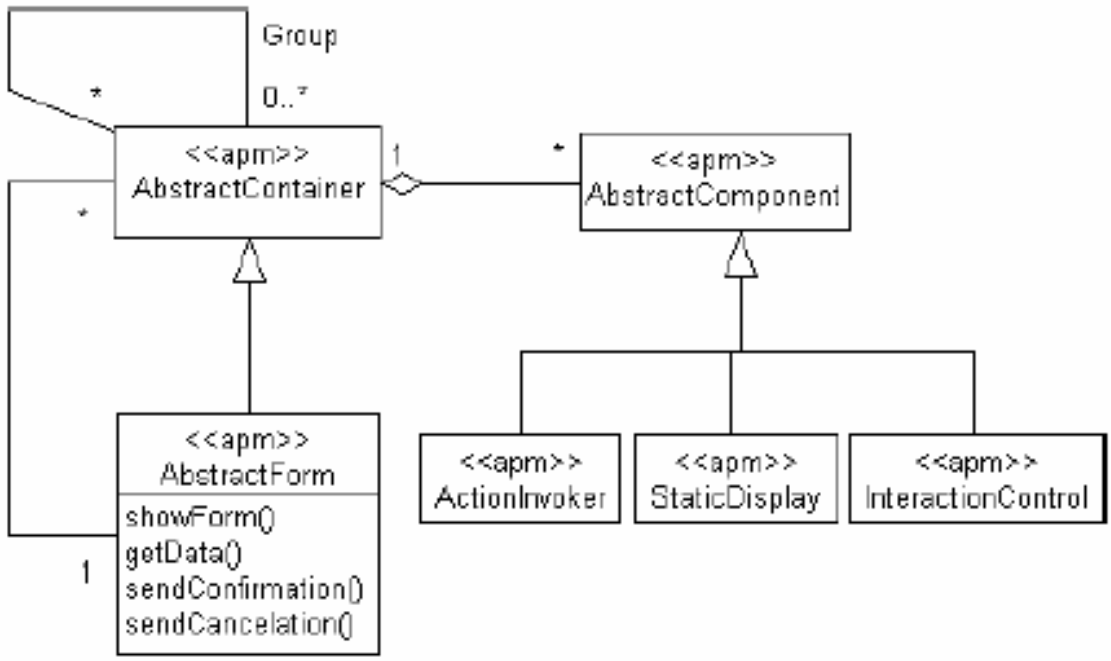


图 6：抽象表示层模型

关于抽象组件（abstract components），Bodart 和 Vanderdonckt[1]提供了更精确的论述。

图 6 所示的类图可作为一个框架（framework）来对概念意义上（conceptual）的用户界面进行描述。可使用该类图的一个对象图来提供用户界面的概念性描述。*ConnectionUI* 的概念性描述可参见图 7。*AbstractComponents* 和 *AbstractContainers* 之间的链接（link）可使用 compose 标签来标识。*AbstractContainers* 的两个实例之间的链接可使用 integrate 标签来标识。

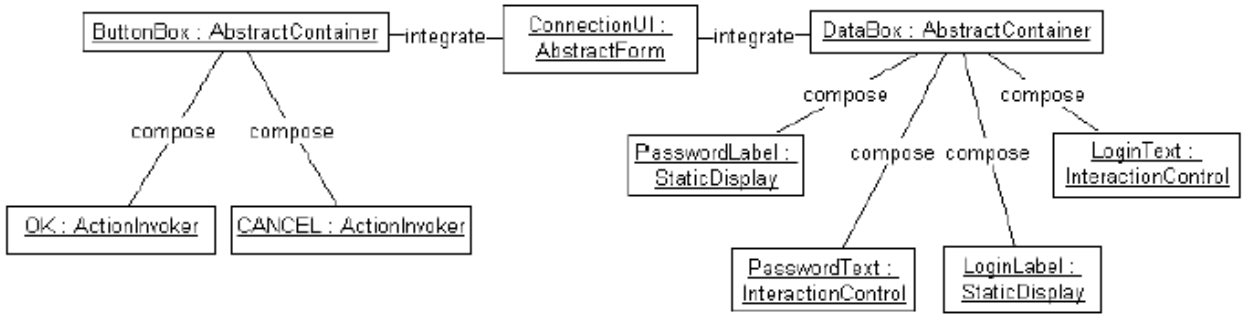


图 7：*ConnectionUI* 的抽象模型

4.2 抽象表示层的行为（Abstract Presentation Behaviour）

AbstractForm 类定义了一组如下所示的四种操作：*showForm()*，*getData()*，*sendConfirmation()* 和 *sendCancellation()*。在该阶段的设计过程（design process）中，这四种操作可被基本理解成一组能够被《boundary》

对象以某种方式实现的可能操作。

- *showForm()*一般用来作为一个自我委托（self-delegate）的消息，在输出设备上进行窗体（form）的绘制工作。当《boundary》对象创建时，该方法可自动执行。

- *getData()*负责收集在一次交互后用户所提供的信息，并将这些信息根据需要转换成系统操作的适当参数。

- *sendConfirmation()*和 *sendCancellation()*消息用于建模那些在一个系统用户和《boundary》对象间进行交互的底层系统操作。它们是由于《boundary》对象的聚合组件（aggregate components）产生了系统事件而相应产生的。*sendConfirmation()*操作通知《boundary》对象系统用户正在向系统提交信息；而 *sendCancellation()*则指明系统用户需要中止（abort）与窗体的交互，而不进行任何信息的提交。

4.3 使用抽象表示层模型

先让我们回到图 4 所示的 *ConnectToSystem* 的序列图中去，*Library System* 创建了 *AbstractForm* 类的《boundary》*ConnectionUI* 对象，其将执行 *showForm()*方法。该方法负责绘制 *ConnectionUI* 窗体并将其显示给用户。通过与 UI 进行交互，用户可以向 *ConnectionUI* 对象发送一个 *sendConfirmation()*消息。*sendConfirmation()*消息可作为一个与 OK 按钮（参见图 5）相关联的事件，但这并没有在抽象表示层建模过程中进行详细说明。*ConnectionUI* 对象执行一个 *getData()*操作，以获取那些由用户所提供的数据。当数据收集完成后，*ConnectionUI* 对象发送一个系统操作的消息 *checkUser()*给《control》*ConnectionController* 对象，并将登录名称和密码作为传递参数。*ConnectionController* 对象提交一次对于数据库管理系统的查询操作，数据库负责实例化一个 *Person*。然后，《control》*ConnectionController* 对象向《entity》*Person* 对象发送一个消息，该消息用于验证所提供的密码。若密码正确，*ConnectionController* 对象将会创建 *MainUI* 对象，并销毁 *ConnectionUI*。以上提到的序列图仅给出了当用户成功登录进系统的场景，而针对登录系统失败时场景的建模可参见节 6.1。

在节 3 中提出的那些活动与抽象表示层模型之间通过活动图中的流对象进行弱关联（weakly connect）。的确，*AbstratComponents* 应该用在活动图中来解释说明 UI 与底层应用之间的数据流。然而我们相信，一个定义良好的活动与 *AbstractForms* 实例之间的关系，能够促进任务和抽象表示层的设计。例如，用户交互中包含的活动应该由《boundary》对象来提供支持。但是，我们很难从一个活动中标识出《boundary》对象，或者从《boundary》对象中标识出活动。

5 具体表示层建模

抽象表示层模型没有描述每一个《boundary》类是由哪些组件组成的，也不提供关于布局的任何描述。此外，它们也并没有描述用户界面组件的事件与《control》类的操作是如何进行关联的。所以在 UI 设计的过程中，有时往往需要*具体表示层模型*。

5.1 具体表示层的结构和布局

从具体表示层的观点来看，图 6 中所示的抽象表示层模型可以作为 UI 表示层模型的设计模式规约。该模式称为*表示层框架（Presentation Framework）*。事实上，设计模式的方法是由 Gamma 等提出[5]并最终并入 UML，该方法可用来描述如何在使用抽象表示层模型元素的图中适应各种不同的环境，如图 4 所示的序列图。其实，具体表示层模型是依赖于环境的，因为它们往往是作为环境类（environment class）及组件的术语来进行描述的。在我们的术语中，“环境”指的是面向对象程序设计语言中的类，组件或两者都是。我们可以使用 Java[6]来说明 UI 类是如何与*环境*类进行关联的。构造型《cpm》可用来标识这些环境类。

图 8 说明了一个使用了*表示层框架*和一些 Java AWT 组件的具体表示层模型。*表示层框架*使用了 UML 中的协作（collaboration）符号来说明以下五个不同的角色（roles）：AbstractForm，AbstractContainer，StaticDisplay，InteractionControl 和 ActionInvoker。遵照抽象表示层模型中的关系（如图 6 所示），*表示层框架*模式可以清晰地描述出抽象表示层类是如何被具体表示层类进行替换的。事实上，图 8 也表明了《cpm》*Frame* 对应着《apm》*AbstractForm*，《cpm》*Container* 对应着《apm》*AbstractContainer*，《cpm》*Label* 对应着《apm》*StaticDiaplay*，《cpm》*TextField* 对应着《apm》*InteractionControl*，以及《cpm》*Button* 对应着《apm》*ActionInvoker*。

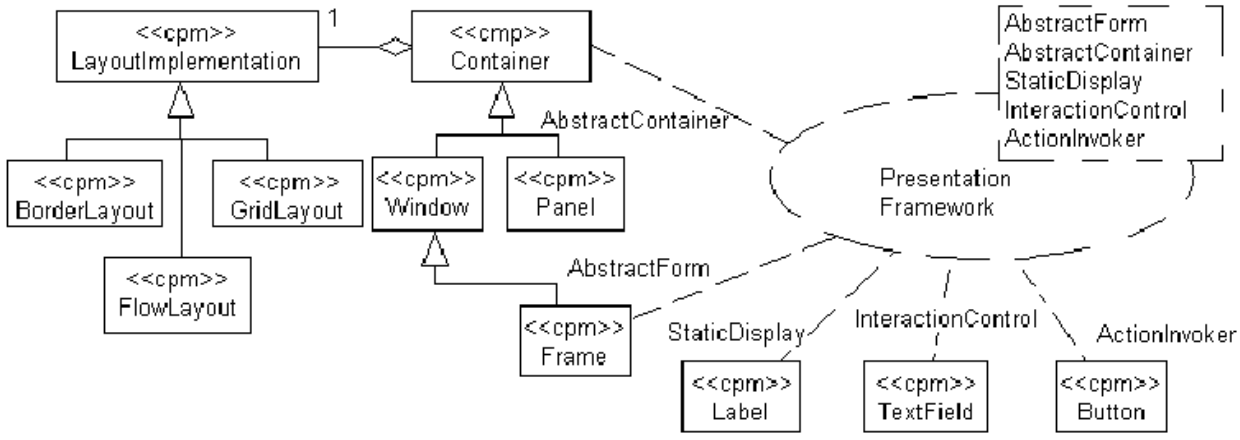


图 8：具体表示层模型

可以对*表示层模型*模式进行扩展，以使多种组件作为 StaticDisplay，InteractionControls 和 ActionInvokers。这种扩展能够提供一个可公布的标记，来建模抽象和具体组件间的映射关系。然而，图 8 所示的具体表示层模型给出了在对我们的案例进行建模时，什么是所需要的。另外，具体表示层模型还描述了如何指定表示层的布局。每一个 *Container* 类的实例必须聚合（aggregate）一个 *LayoutImplementation* 实例。有几种布局对象（layout object）可作为 *LayoutImplementation* 的子类，被添加至具体表示层模型之中。我们并没有对 UI 表示层布局的建模进行完全说明。实际上，具体表示层模型依赖于环境这样的语义。对于在图 8 中所示的具体表示层模型的情形，Java 提供了内嵌于方法中的算法，并以此作为模板来对 UI 表示层模型的布局进行建模。

基于*表示层框架*的具体表示层模型，不会导致很大部分的设计需要依赖于具体的环境，它提供了一个在《apm》类和组件类之间具有灵活、稳固关系的模型。比如，虽然图 8 是使用 Java 的 AWT 组件进行建模的，但是 Swing 组件也能够非常自然地对 AWT 组件进行替换，而不会破坏抽象表示层模型。

图 9 中的抽象表示层模型便是对 *ConnectionUI*（如图 5 所示）所进行的建模。该模型为一个对象图（object diagram），其所用到的链接包括：节 4.1 中的 *compose（组合）*和 *integrate（集成）*链接，与 Frame 实例（其可作为 *AbstractContainer*）以及它们各自的 *LayoutImplementation* 实例相关的 *organise（组织）*链接。对于每一个 *Frame* 实例来言，该连接是必不可少的。进一步说，图 9 表明 *Panels* 可以用来代替 *Containers* 对非顶层容器（non top-level containers）进行建模。当边界类（bound classes）的子类作为具体表示层模型的一部分时，便会出现该情形。

5.2 具体表示层的行为

一旦我们了解了如何对具体表示层模型的结构建模，就需要对表示层的行为进行建模。如图 9 所示，大量的组件具有与其相关联的行为。例如，当用户按下按钮时，*OK* 对象（*Button* 类）将会触发一个与其相关联的事件。而另一方面，*PasswordLabel* 对象（*Label* 类）则没有关联任何需要应用程序进行处理的事件。

关于对表示层的行为进行建模的第一个问题是，如何从 UML 图中标识出可能与《boundary》对象相关联的事件。在我们的应用程序模型中每一个《boundary》对象就是一个 *Frame* 实例，它具有两个与其自身进行事件关联的操作：*sendConfirmation（）*和 *sendCancellation（）*。

通过对应用程序模型进行检查我们可以发现，*sendConfirmation（）*消息已在图 4 所示的序列图中进行过描述，而 *sendCancellation（）*则根本就没有进行过建模过。实际上，每一个从角色（actor）发往《boundary》对象的消息都表明了一个与 UI 组件相关联的事件。因此，一个可行的方法是可以为每个 UI 事件生成一个交互图，用来建模所有的表示层行为。在 *ConnectToSystem* 的用例中我们仅建模了一个场景：用户成功登录至系统。所以，我们需

要对当 *ConnectToSystem* 中的 CANCEL 按钮被按下时的场景进行建模。

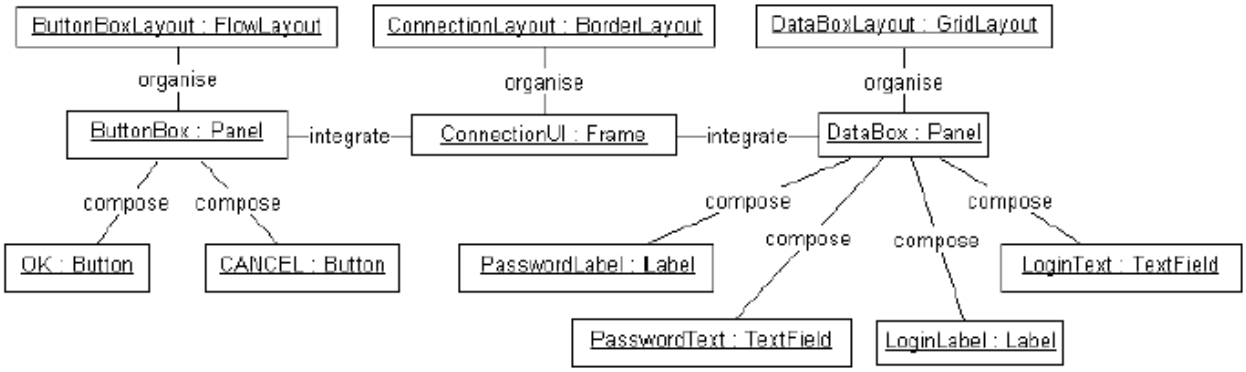


图 9：ConnectionUI 的具体表示层模型

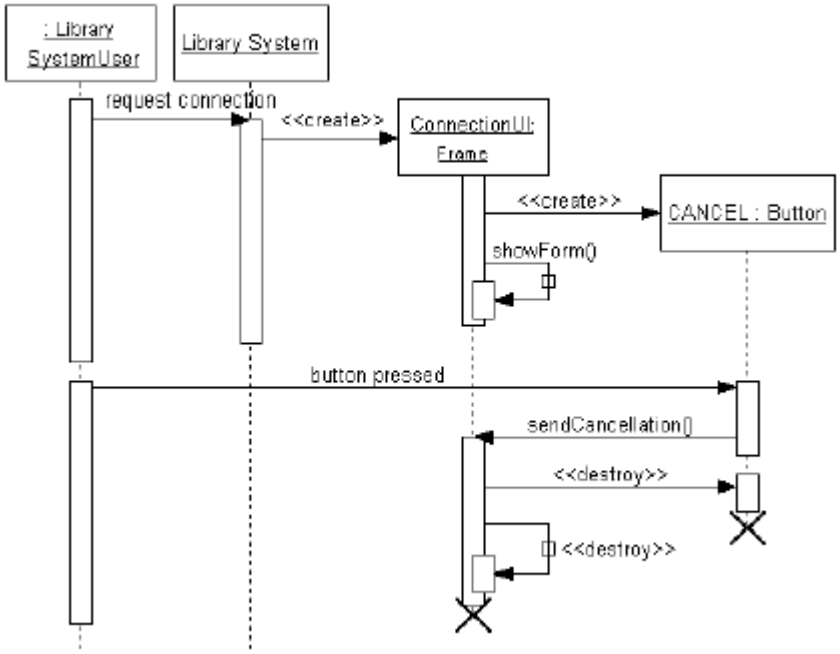


图 10：ConnectToSystem 用例的第二个序列图

图 10 显示了在 *ConnectToSystem* 中当用户按下 CANCEL 按钮时的序列图。可将 CANCEL 按钮的被按下事件表示成一个*按下按钮*的消息，并由图形组件 CANCEL 按钮的用户进行发送。事实上，*按下按钮*的消息是一个具体表示层模型的事件，这是由于它标识了按钮组件可以使用哪种类型的事件来触发一个 *sendCanellation()*消息。因此，*sendCanellation()*消息将会发向 ConnectionUI 对象，并中止用户与图书馆系统的交互。

然而，使用一个交互图来对每个 UI 事件进行建模并非一个很好的策略。这是因为在一个单独的用例场景内，一个用户可能是与多个《boundary》对象进行交互的，而一个单独的《boundary》对象往往可能有数十，甚至数百的事件与其关联。因此，需要一个能够处理所有的事件组合的策略，从而使用最小数量的交互图来描述所有可能的事件。虽然事情并不像看起来的那样糟糕。许多组件都对应用程序的行为部分进行了封装。而且复杂的组件能够减少一个《boundary》对象需要处理的事件数量。

6 事件建模

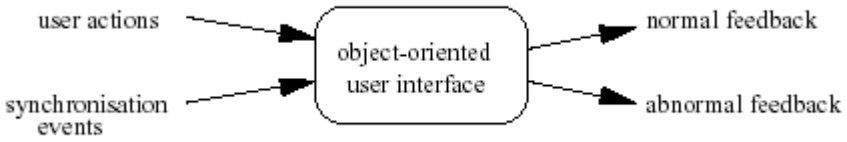


图 11：事件模型

诚如 Booch[2]所言，*事件*是指“发生的事情”。的确，在我们使用一个应用系统过程中会有许多事件发生：按键和按钮被按下，鼠标移动，发向网络的消息等等。我们将这些都称之为发生的*事件*。

在一个面向对象的用户界面中，输入和输出皆为事件流[7]。图 11 显示了一个通用的事件模型（event model），其中*用户动作（user actions）*和*同步事件（synchronisation events）*将会作为输入事件发送至一个面向对象的用户界面。应用程序通过其用户界面，来对输入事件做出响应并产生输出事件，称之为*可视反馈（visual feedback）*。可视反馈分为*正常反馈（normal feedback）*和*非正常反馈（abnormal feedback）*。其中非正常的可视反馈，例如错误消息（error messages），与在一个用户任务的制订过程中所遇到的故障紧密关联。

用户动作，系统事件以及正常的可视反馈事件已经在本文各处予以论述。如在图 4 中，用户与《boundary》对象进行的交互，《control》对象与《boundary》对象及《entity》对象之间进行的消息发送和接收，而《entity》对象与《control》对象之间进行的消息发送和接收。因此，本节论述的策略是为了对以下用户界面的特性进行建模：

- 异常（Exceptions），是一种特殊类型的事件，在系统发生不可预期的情形时产生。的确，交互式应用程序的许多功能需要支持非预期情况的出现。
- 同步事件，产生于应用程序内部，以确保用户界面上的显示数据与应用程序内的数据保持同步。

非正常的可视反馈可伴随着同步事件发生，而同步事件可由系统动作产生。但是，可以分别对它们进行建模。在下节中将对这些特殊的事件依次进行论述。

6.1 异常处理建模

异常，由 Meyer 定义[16],其作为运行时事件（run-time events）“可以造成一个例程（routine）调用的失败”。另外，当例程由于一个状态不满足例程的契约（contract）而终止执行时，则该例程将会调用失败。这些定义会很复杂，因为它们需要更进一步的定义，例如*例程的契约*的定义，而这些进一步的定义也同样复杂。因此，要区别什么是一个调用失败，什么是一个异常并非显而易见的事情。除了正常的定义，这里的异常可能更类似于面向对象的程序设计语言（如 Java[6]和 C++[18]）中那些异常的概念。

在用户界面方面，异常的重要方面是它们有时无法由异常处理（exception handler）完全解决，这时应用程序将会以可视化的方式向用户反馈某些地方已经出错（或者至少不像预期的那样）。事实上，异常处理一旦被激活，便会尝试着去解决那些由异常所标识的问题，而不会通知用户。而不幸的是，异常处理并不能解决每一种类型的问题。所以，应该将这些无法解决的异常通知给用户，或由用户来选择一种处理该问题的方案。

于是现在的问题便成了：如何进行与用户界面相关的异常处理方面的建模工作。

6.1.1 UI 异常处理的结构

在应用程序模型中有许多地方使用到了异常和异常处理。例如，对于一个在执行数据库查询过程中产生的异常，可由设计者选择在 *ConnectionUI* 窗体的某个位置来显示一条错误消息。

在 UML 标记中，异常可作为构造型《send》来进行建模，而构造型《send》标识的是从一个类的操作到一个异常处理类[2]的依赖关系（dependency）。图 12 标识了从《control》*ConnectionController* 类的 *checkUser* 操作至《exception》*DatabaseFail* 类的一个《send》依赖关系。此外，Booch 等人[2]提出的异常处理的分层结构则是使用构造型《exception》来进行标识。通常，那些未被捕捉到的异常将会被抛至的更高一层的异常处理中，直到它们被一个异常处理捕捉到为止，或最终到达分层结构中的顶层异常处理。如图 12 所示，若一些异常未能被《exception》*DatabaseFail* 所处理，则其将会被《exception》*Exception* 进行处理。

任何类都有可能产生异常，这是因为类一般都具有方法（即例程），而这些方法所具有的契约（contract）都有可能被打破。尽管异常处理可以作为《control》类，但它们还不能完全像《control》类那样进行建模。而是，它们能够捕捉任何类型的类的异常（事件）。在这种情况下我们可以引入《exception》类。这些类的操作可从任何类的任何方法中进行调用，甚至是其它《exception》类的方法。

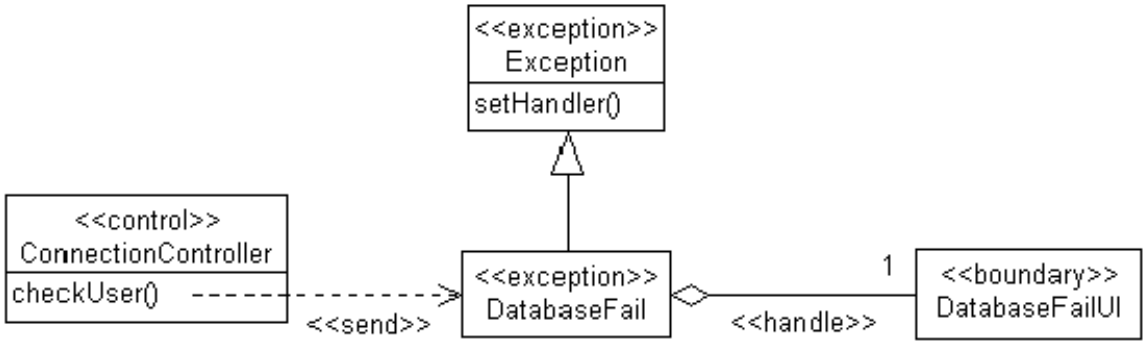


图 12：UI 与事件处理之间的关系

当异常发生时，《exception》类的其中一个角色（role）就是可以作为《boundary》类的《control》类。然而在很多情况下，《exception》类不能对一个《boundary》类进行控制。例如，当异常处理需要一些来自用户的决定，*退出（quit）*或*重试（retry）*，而原先的《boundary》对象并没有组件来处理这样的一次交互，那么就可以通过创建一个新的《boundary》对象，以提供在异常处理和用户之间的通信。

然而，在用户界面方面，重要的是要了解《boundary》类是如何与这个异常分层结构进行关联的。《exception》类的对象可以作为《control》类的对象。因此，《exception》类能够聚合《boundary》类。参见图 12，《exception》*DatabaseFail* 则是作为一个《control》类，来对《boundary》*DatabaseFailUI* 类进行处理。可以使用构造型《handles》来标识《boundary》类及其控制者的关系。

6.1.2 UI 异常处理的行为

异常也会影响用户界面的任务模型，这是因为它们能够更改一次用户交互中的活动控制流。例如，图 3(b)中的活动 *Perform search* 在执行一次查询时可能会引发一个数据库异常（database exception）。

由于 UML 的活动图提供了一种分支标记，使得我们能够直接建模那些在任务模型的控制流中所可能做出的改动。根据布尔型的警戒表达式，可选择不同路径外向转移（outgoing transition）至不同的活动中去。为了对异常处理进行建模，图 13 中扩展了图 3(b)所示的活动图。在执行 *Perform search* 中发生异常时，活动 *Perform search* 之后的一个分支（标识为一个菱形）可以选择不同的路径来对控制流进行更改。当一个 ODMGException 异常没有被其处理者进行处理时，警戒条件[non-solved ODMGExceptions] 便会将控制流的路径选择到 *Handle ODMGException* 活动上去。否则，控制流则按照正常的路径进行（由关键字 else 进行标识）。

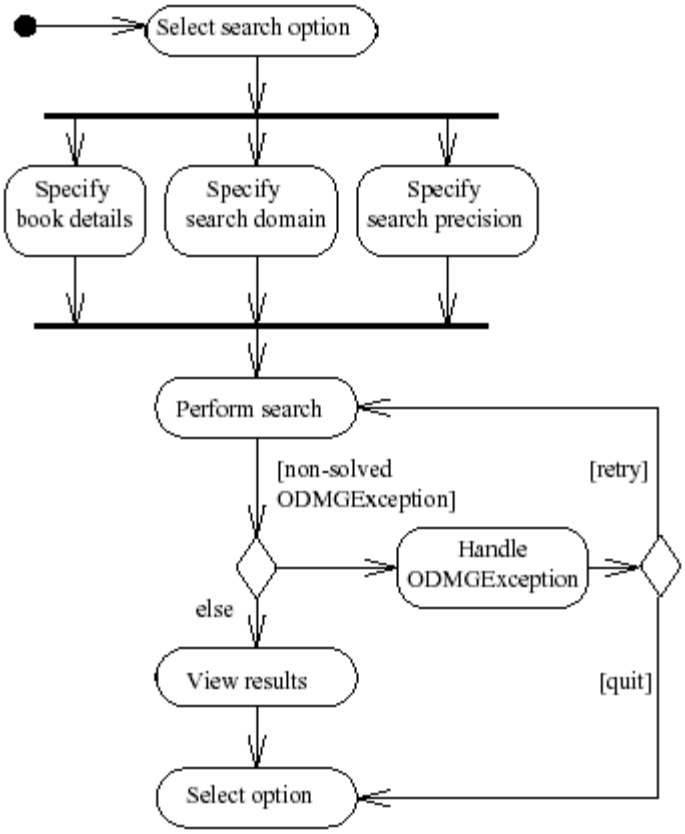


图 13：任务模型中的异常

6.2 同步事件建模

同步 UI（synchronous UIs）指的是那些当《boundary》对象可见（visible）时，能够频繁地对所显示的数据进行更新的 UI。否则便为异步 UI（asynchronous UIs）。

用户界面，特别是指图形用户界面，通常用异步消息（asynchronous messages）[4]来实现。所以另外一个需要考虑的问题便是，如何只使用异步消息来对同步 UI 进行建模。解决这个问题的一般思路是，按照所要求的频率通过数据更新来完成《boundary》对象的刷新（refresh）。由于事件的产生能够引起 UI 的更新，因此可将其作为同步 UI 建模的一种可能的方法。在该种情况下，产生的事件称为*同步事件（synchronisation event）*。我们能够很自然地想到，《entity》对象能够产生同步事件，因为它们是存储更新数据的地方。而《boundary》和《control》对象也能够产生同步事件，但是由于它们在产生每个同步事件时，需要对《entity》进行查询来获取所需的更新数据。因此在这里，我们只考虑同步事件由《entity》对象产生这一种情况。

图 14 所示的类图表明了一种可能的建模方法，即使用《entity》对象来产生同步事件。

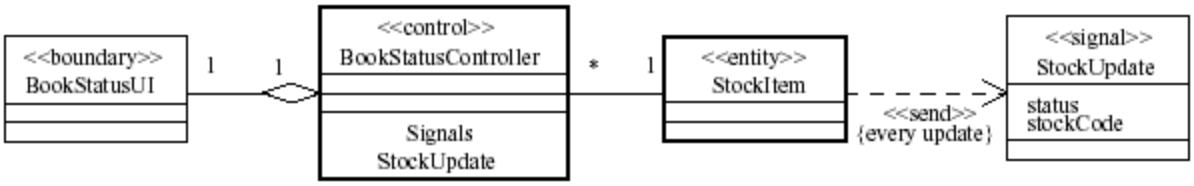


图 14：同步 *BookStatusUI* 模型

我们还可以使用一些上面没有讨论到的 UML 特性来对同步 UI 进行建模：使用粗线表示的类，具有一个信号隔间（signal compartment）的类，一个依赖关系的时间约束，和构造型《signal》。因此，在解释模型之前，我们需要先来解释一下这些特性。

- *BookStatusController* 和 *StockItem* 上的粗线是指这些类的实例为主动对象(active object)。实际上，*BookStatusController* 实例必须具有一些主动的行为来“接收”（receive）由 *StockItem* 实例所产生的 *StockUpdate* 事件。

- 约束{every update}，位于 *StockItem* 和 *StockUpdate* 之间的《send》依赖中，则是指当 *StockItem* 每次被更新时，*StockItem* 实例都必须产生一个 *StockUpdate* 事件。该约束的意思是 *StockItem* 实例具有一种主动的机制来标识状态的更新。

- 构造型《signal》指的是该类的实例将会发送信号来通知系统，当前产生了一个 *StockUpdate* 类型的事件。在该环境（context）下，这个事件是则指 *StockItem* 类的对象正在将藏书项（stock item）的更新*状态*推（push）给那些监听该种事件的类。

- *BookStatusController* 中的 *Signals* 隔间，则标识了该类的实例能够作为 *StockUpdate* 事件的监听器（listener）。这便意味着每当一个 *StockUpdate* 事件产生时，该实例便可作为监听器被通知。

让我们回到图 14 所示的类图中去，可以注意到每当发生一个 *StockItem* 的更新时，则 *StockItem* 实例便会产生一个 *StockUpdate* 同步事件，该同步事件具有这个藏书项的当前状态。然后，*BookStatusController* 实例将会监听到这些事件，并发送消息至《boundary》类的实例，进行数据的更新显示。信号 *StockUpdate* 内部的属性 *stockCode* 可作为一种身份鉴别机制，以防止 *BookStatusController* 对象显示的是不同库存项的状态。

该建模方法尤其适合那些以同步 UI 为主的系统。事实上，《entity》对象内部所用的推机制的复杂程度是相同的。举个例子来说，复杂的《entity》对象可能具有许多可以改变其状态的操作。然而，由于对状态的更新进行监听，则使用该建模方法就不必再对对象内部方法中的那些状态修改进行标识。此外，《entity》对象无论是被一个或多个《boundary》对象来显示，其推机制的复杂程度都是相同的。实际上，发往每个对象的事件都具有一个签名（signature）。所以说，一个单独的同步事件无论有多少个监听器，都能够进行通知。

7 装配：打包应用程序

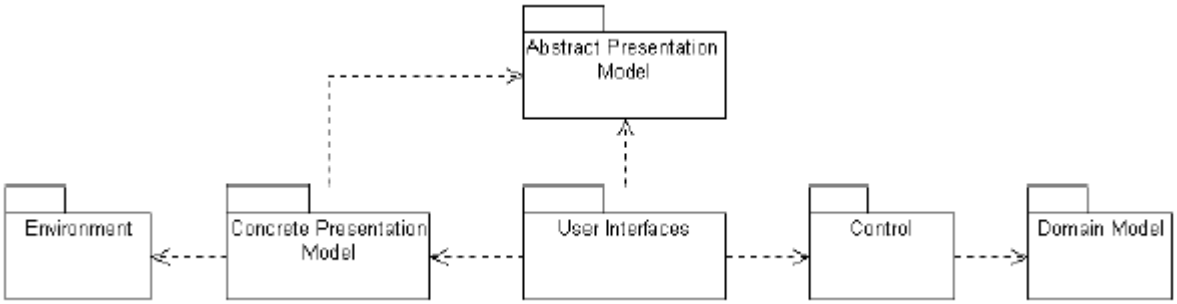


图 15：图书馆系统的包图

图 15 所示的包图（package diagram）给出了整个系统的概况。进一步而言，该包图展示了系统各个组件之间的依赖关系。

类和类的实例被分成六个包（package），如下。

- *User Interface* 包由《boundary》类及其对象组成。
- *Abstract Presentation Model* 包由图 6 所示的《apm》类组成。
- *Control* 包由《control》类组成。这些类可参见图 6。
- *Domain Model* 包是由《entity》类组成。这些类的类图可参见图 2 中所示的领域模型。
- *Environment* 包则包括那些用于构建用户界面的类。*环境*可以是一门面向对象的程序设计语言，如 C++[18]，Java[6]和 Smalltalk，也可以是一些组件，如 ActiveX 和 Java Bean，或者甚至是组件和面向对象程序设语言的组合体。环境主要是作为用户界面的可视化部分来考虑的，但它还可以对用户界面的行为进行负责，特别是在使用复杂组件的时候。

■ *Concrete Presentation Model* 包中所包含的是，在 *Environment* 包中与*表示层框架*模式中的《apm》类相对应的那些类。

8 结论

本文使用了一个图书馆系统的案例，来论述了关于用户界面的建模。通过使用统一建模语言来对应用系统进行建模，也说明了 UML 可以用来进行用户界面的建模。其实 UML 有着一组丰富的构建法（constructor），完全可用来对基于窗体的用户界面架构的各个方面进行建模。然而，这样的 UI 建模方法并没有像所期望的那样可直接作为一个过程（process）来对待。事实上从该案例中我们还是认识到一些关于建模方面的问题：

■ UML 并没有清晰地描述用例与活动（参见节 3）之间的关系。用例并没有提供一些有关用户需求方面的信息（像目标，前置条件和后置条件），来帮助进行活动图的设计。

■ UML 并没有一种描述抽象表示层（参见节 4）的标记。事实上，我们认为 UML 需要这样的一种标记来支持 UI 表示层的设计。

■ UML 并没有提供一种在抽象表示层的类（参见图 6 中的 AbstractForms）与活动之间的对应关系。事实上，对于那些与包含了用户交互的活动相关联的 UI（详情可参见节 4.3），我们很难予以标识。

此外，该案例还提供了一个使用多个 UML 构造法的图例，来阐述有关用户界面方面的建模。表 1 总结了所用到的 UML 图，以及在本文图中所用到的构造法。

用户界面元素	UML 资源
领域模型	类图
任务模型	活动图
表示层模型（抽象和具体）	使用了设计模式的类图 交互（序列）图 对象图
事件 – 与异常处理相关的 UI 事件 – UI 同步	类图 活动图 类图

表 1： UI 元素建模中所用的 UML 图的总结

由于我们使用了活动图[10]，因此可能会用到一些状态图（statechart）[11]的构造法。而且本文也没有考虑有关窗口组件（widget）的设计。基于这个原因，我们在对窗口组件进行建模时，更多地是在论述对象间（inter-object）的转移（活动图），而非对象内（intra-object）的转移（状态图）。

在这个图书馆系统的建模过程中，还能学到以下内容：

■ 一个用户界面的设计是一个复杂的过程，这是因为它需要对用户界面的组成元素完全进行理解。其实，在设计的初期，并不需要清楚地知道一般的 UI 是由多少个元素组成的。

■ 用户界面的元素之间往往存在着大量的依赖关系，可如图 15 所示。所以在设计过程中，我们应该将 UI 建模作为一个整体来看。

在此，我们对如何使用 UML 来对用户界面建模作了一些深入的讨论。事实上，还有许多其他的方法可以通过使用 UML 来表现用户界面。本文的案例仅仅给出了其中的一种。

致谢

本文的第一位作者 Conselho Nacional de Desenvolvimento Cientifico e Tecnologico – CNPq(Brazil) 。

参考文献

[1]F. Bodart and J. Vanderdonckt. Widget standardization through abstract interaction objects. In *Advance in Applied Ergonomics*, pages 300-305, Istanbul – West Lafayette, May 1996. USA Publishing.

[2]G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.

[3]R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Francisco, CA, 1997.

[4]D. Collins. *Designing Object-Oriented User Interfaces.* Benjamin/Cummings, Redwood City, CA, 1995.

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

[6]J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996.

[7]M. Green. A survey of three dialogues models. *ACM Transaction on Graphics*, 5(3):244-275, July 1986.

[8]T. Griffiths, P. Barclay, J.McKirdy, N. Paton, P. Gray, J. Kennedy, R. Cooper, C. Goble, A. West, and M. Smyth. Teallach: A Model-based user interface development environment for object databases. In *Proceedings of UIDIS'99*, pages 86-96, Edinburgh, UK, September 1999. IEEE Press.

[9] T. Griffiths, J.McKirdy, N. Paton, J. Kennedy, P. Barclay, R. Cooper, C. Goble, and P. Gray. Exploiting model-based techniques for user interfaces to database. In *Proceedings ofVDB-4*, pages 21-46, Italy, May 1998.

[10]Object Management Group. *OMG Unified Modeling Language Specification*, June 1999, Version 1.3.

[11]D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31-42, 97.

[12]I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley, Reading, MA, 1992.

[13]P. Johnson. *Human computer interaction: psychology, task analysis and software engineering.* McGraw-Hill, Maidenhead, UK, 1992.

[14]B. Kirwan and L. Ainsworth. *A Guide to Task Analysis.* Taylor & Francis, London, UK, 1992.

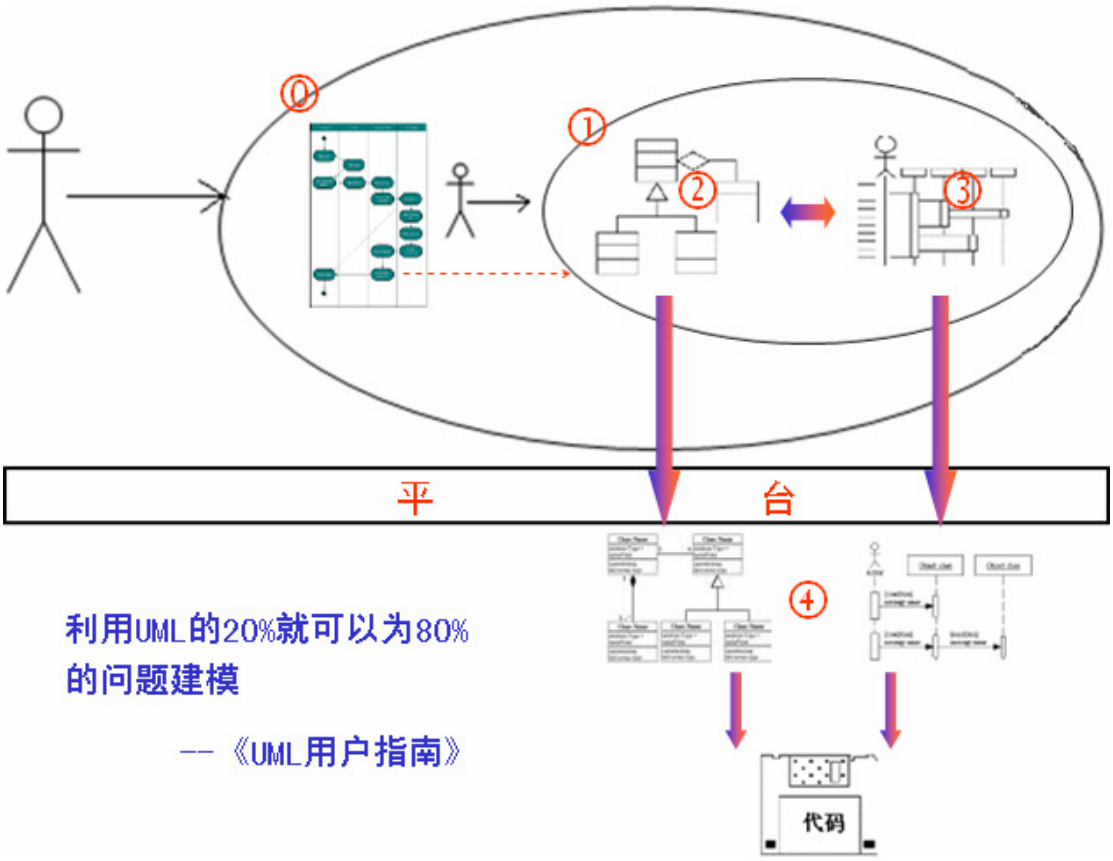
[15]S. Kovacevic. UML and user interface modeling. In *Proceedings of UML'98 International Workshop*, pages 235-244, Mulhouse, France, June 1998. ESSAIM, Mulhouse.

[16]B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, Upper Saddle River, NJ, second edition, 1997.

[17] B. Meyers and M. Rosson. Survey on user interface programming. In *Proceedings of SIGCHI'92*, Pages 192-202,1992.

[18]B. Stroustrup. *The C++ Programming Lauguage.* Addison-Wesley, Reading, MA, third edition, 1997.

[19]P. Szekely. Retrospective and Challenges for model-bases interface development. In *Computer-Aided Design of user Interface*, pages xxi-xliv, Namur, Belgium, 1996. Namur University Press.

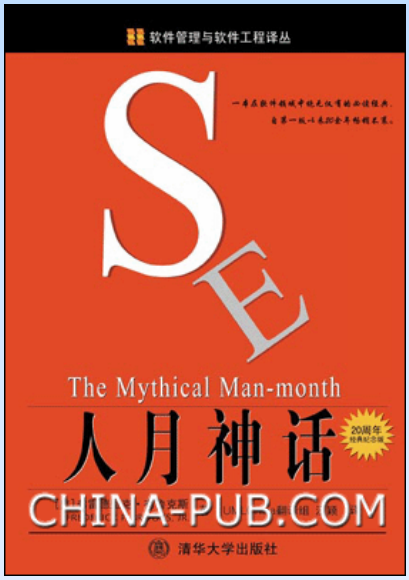


北京公开课，2002 年 9 月 14 日至 15 日

北京中关村 1 号海龙大厦 713-714 报告厅

报名交费请联系：bjcourse@umlchina.com

《人月神话》



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 Adams Wang

散文笔法，绝无说教，大量经验融入其中

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

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

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

中文译本即将发行！

界面耻辱纪念堂—术语

iarchitect 整理，金哲凡 译

查看评论

如果程序开发者和用户具有相同的知识背景，那可就太好了。程序就会按用户完成任务的需要来设计，双方都知道对方说的是什么。不幸的是这样的情况太少了。有太多的程序给人的印象是：它们使用不同的语言。下面就是一些非常不清楚的程序元素的例子。

Olli Siltanen 建议我们研究一下用于 HTML 语法错误文档检查的软件 *CSE HTML Validator v3.05*。我们发现它的 Option 对话框的这个部分特别有问题。

“Flag”检查框用于设置关于语法检查的和其他的一些选项。让我们提些问题，做一个快速的测试：

1. 哪个 flag 设置 Internet Explorer 的 HTML 标记规范？
2. 哪个 flag 设置 Netscape Navigator 的标记规范？
3. 哪个 flag 设置对<CENTER>标记的警告？
4. 哪个 flag 控制在验证过程中发出声音与否？
5. Flag 14 的作用是什么？

答案如下：

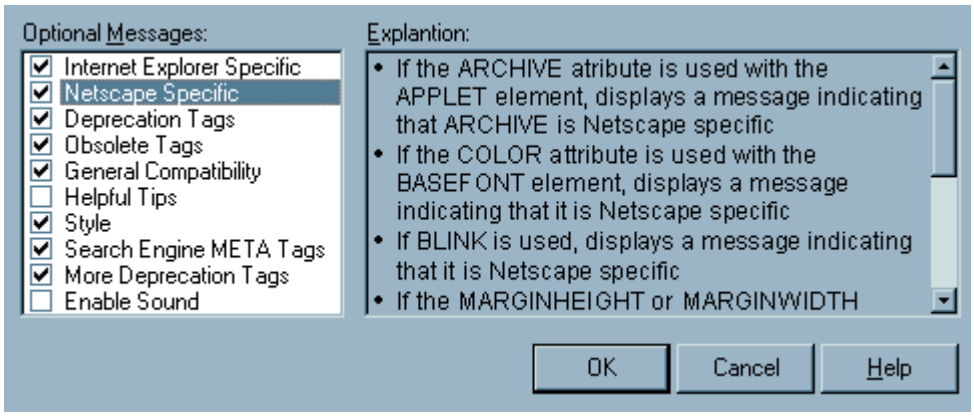
1. Flag 1。
2. Flag 2。
3. Flag 9。它也对<BASEFONT>， <FONT>， <S>， 和 <U>标记进行警告。

4. 当然是 Flag 10。

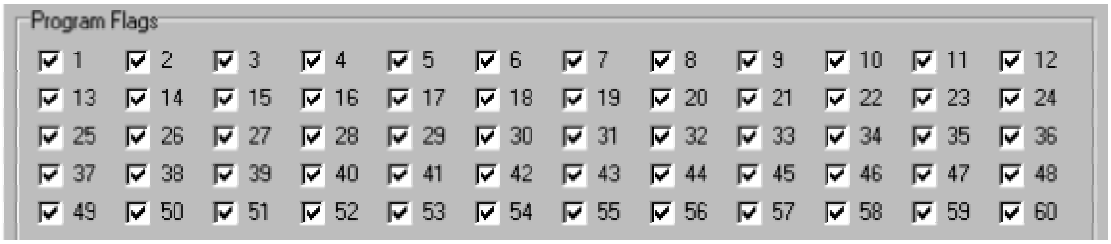
5. 什么作用也没有。OK，这个问题是个陷阱，flag 14—20 什么作用也没有，它们只是以后可能的扩展功能的占位符。

每个 flag 控制对若干 HTML 标记的验证，用户要知道 HTML 标记和 flag 之间的关系，必须访问帮助文件。目前这种设计仅仅是程序员头脑中系统模型的直接映射，对用户并没有什么帮助。开发者关注的是使程序能适应各种不同的浏览器，并能针对 HTML 规范的改变作出相应的扩展。

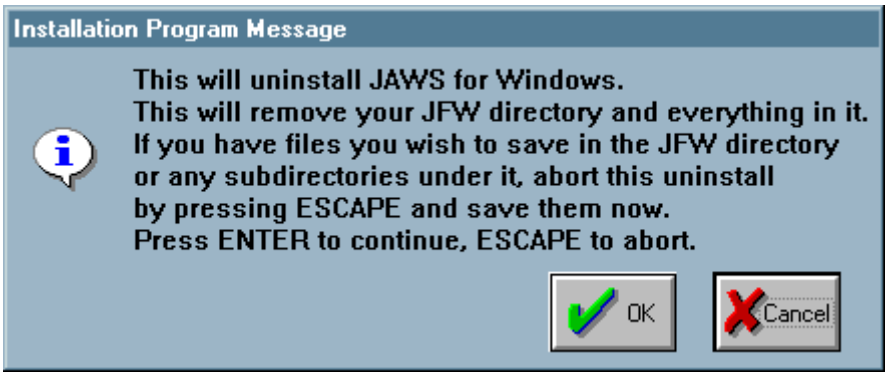
嘿，我有个主意：如果 Flag 1 表示 IE 的规范，为什么不把它叫作，哦，我不知道，也许……IE-Specific？用我们下面给出的这个设计，一定会使用户和开发者都获益。这个界面提供了对程序设置的直接访问，同时开发者不必修改表单和说明文字的设计，就能加入新的功能。



最近 *CSE HTML Validator* 发布了 4.0 版本，开发者的坏主意使它变得更糟糕了。现在用户不是要跟 20 个 flag 斗争，而是不得不在 60 个 flag 里挣扎，请看：



另一个新变化是新版本包含一个基于 HTML 的帮助文件，它使用一个导航的结构，使得找到某一 flag 的解释变得更加困难。



JAWS 是一个非常有用的屏幕读取程序，在卸载 JAWS 时会出现这样一条信息。这个对话框虽然简单，但它暴露出不少重要的问题。

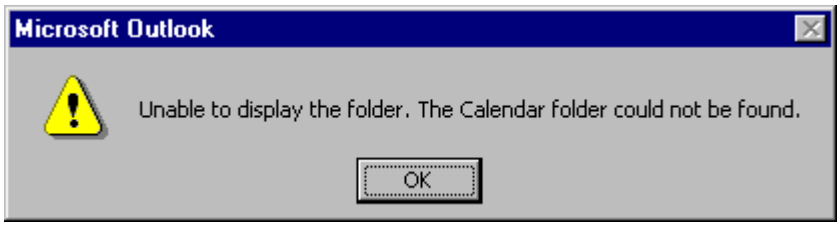
用户已经不幸地习惯于使用确认对话框了，而这个特殊的确认对话框需要用户特别注意。在卸载 JAWS 时，它的安装目录及其子目录下的所有内容，不论是否属于 JAWS，都将被删除。这是一个不必要地危险操作，而那些刚好随便让确认对话框通过的用户将忽视这一操作的后果。

理解这个消息的重要性的用户，有可能进入文件系统做一些文件处理的工作。这里“Save”是一个相当不合适的说法，可能会让没有经验的用户感到疑惑。由于程序即将销毁整个目录，而不是删除相应的文件，所以用户应该“move”文件或子目录。

这个对话框还有其他的问题。它包含了 OK 和 Cancel 按钮，但是指示文字却指向 ENTER 和 ESCAPE 键。新用户不大可能知道 ENTER 键等价于缺省的命令按钮，或者 ESCAPE 等价于 Cancel 按钮。另外，如果缺省命令按钮改变了，一个不经意的键盘动作可能导致键盘映射不再正确。如上图所示，按下 ENTER 键的结果是选择了 Cancel 按钮。

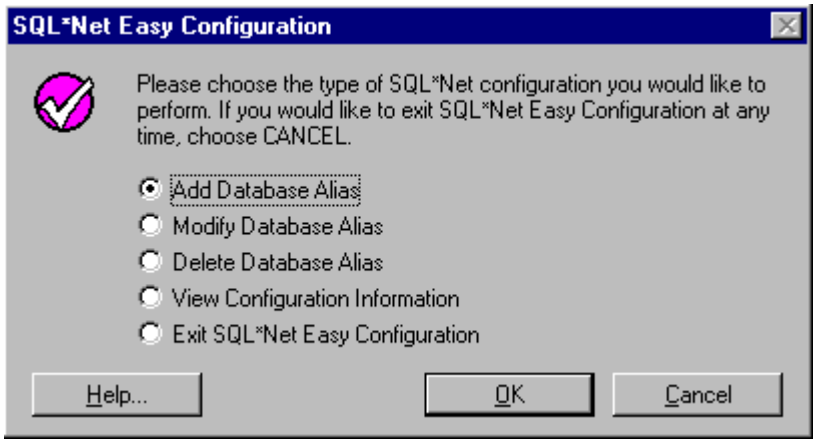
JAWS 卸载程序的一个特别阴险的问题是它不删除安装程序在用户的启动菜单里创建的半打目录。用户被迫打开 Windows95 的文件树去清理懒惰的程序员留下的垃圾，过后还会担心会不会有什么别的东西被粗心地留在了系统里。

Alexander Lewis 寄来了 Microsoft's *Outlook '98* 的一个容易让人误解的出错警告的图片。



当一个用户想要访问另一个用户的日历，而他又没有这个权限时，这个消息就会产生。这个消息给人的印象是似乎另一个用户的配置存在什么问题，而实际上，这是个访问权限的问题。如果试图访问的用户被赋予了日历的读权限，这个消息就不会出现了。

我们在安装 TransSoft Ltd.的 *FTP Control* (v. 3.33)的演示版时碰到了这个令人疑惑的界面。安装程序为了引导用户经过一系列配置屏幕而使用了 Wizard，这是其中的一部分。对话框中提供了一个“Next”按钮，并且还是缺省的，但文字却指示用户选择“Finish”按钮。我们敢打赌，尽管有文字指示，大部分用户会按下“Next”按钮，结果他们会发现自己回到了配置过程的起点。

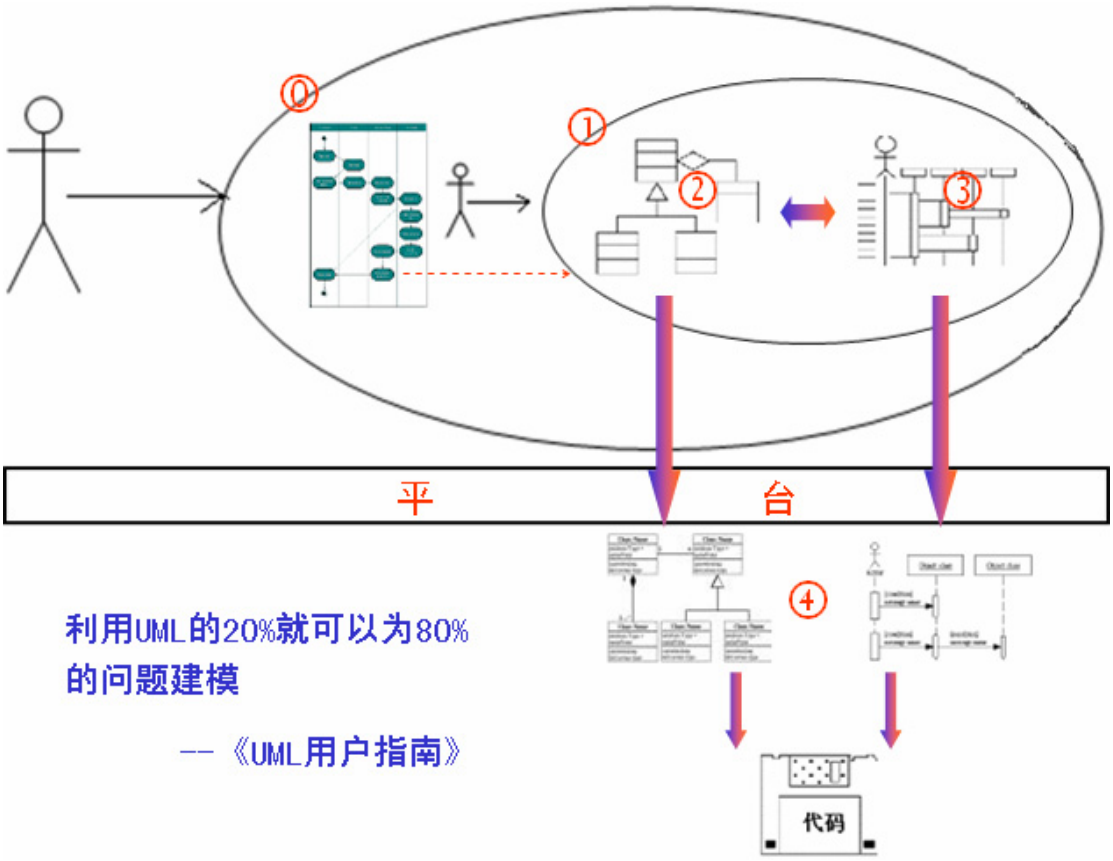


这个图片是 Nigel Harris 寄来的，它是 Oracle 的 SQL*Net Easy Configuration 工具的一个主要的界面。问题之一是 Cancel 按钮实际上没有 Cancel 任何东西，而是使用户退出了应用程序。更仔细地研究这个界面的说明文字我们会发现：

任何时候你想退出 SQL*Net Easy Configuration，选择 Cancel

这个主意怎么样：把 Cancel 按钮标记为“Exit”?

说明文字没有指出退出程序的另一种方式：选中“Exit”选项并按下“OK”。Cancel 实际上就是退出，既然如此，为什么还有给出一个“Exit”选项呢？



北京公开课，2002 年 9 月 14 日至 15 日

北京中关村 1 号海龙大厦 713-714 报告厅

报名交费请联系：bjcourse@umlchina.com

John Vlissides, Erich Gamma & Ralph Johnson

Erich Gamma & Kent Beck

用户界面软件

Jens Coldewey 著，Tom.X 译

查看评论

本文中的模式语言逐步深入地探讨用户界面架构的设计，它基于人机工程学，足以形成一套完整的体系。如果你对这方面有兴趣，请参考*[Tog92]*，*[Coo95]*和*[Col95]*。

本文不讨论用户界面的布局，相反，本文是关于那些驱动用户界面的软件的，这个模式语言的一览图如下图 1 所示。它从最基本的模式用户界面层（*User Interface Layer*）开始，然后由另外两个模式描述这个体系架构：分离转换（*Separate Transformation*） 解释了如何处理复杂的交互，而配件模型（*Widget Model*）则帮助对界面进行结构化。尽管看上去两个模式描述的是不同的事物，实际上它们经常是被绑在一起的，从而形成基本模式 *User Interface Layer*。

不过，还有更多细节要处理。首先，需要在用户的不同的交互中提供上下文支持（*Context Support*）。根据系统的需求和架构，可以使用多种不同的模式（为了简短起见，本文中仅使用了它们的缩略形式）；除此之外，基于域层面的存取（*Domain Layer Access*）也是一些模式的基础。这些模式中有些是大家都熟悉的，但有些是在用户界面中特有的。

大部分的用户界面架构都可以用这一套模式来描述。

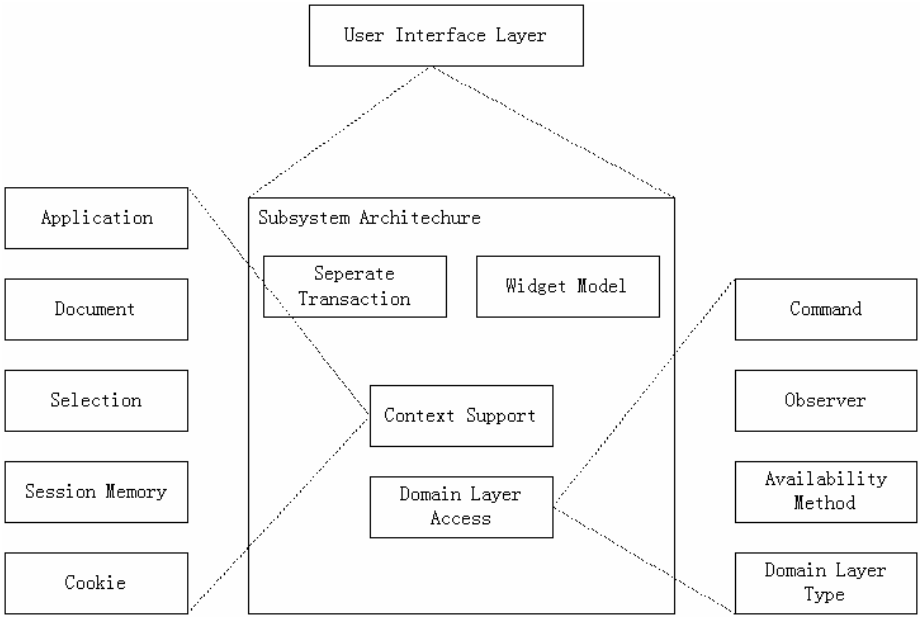


图 1

部分术语说明

在这里，有些术语可能是不大常见的，这里给出它们的简要定义，一些说明会告诉你如何寻找更详细的信息。

面向对象的用户界面　“面向对象的用户界面”指的是这样一种界面范型。用户先选中一个对象，然后在上下文菜单里面选中一个操作，或者选中一个对象直接操作。Macintosh 的用户界面就是这种界面的一个例子：用户选中一个文件，然后决定对它进行什么操作，她可以双击它来进行操作，也可以把它拖到打印机图标上去打印，或者干其它想干的事情。术语“面向对象”在这里指的是一种交互风格，而不是实现技术，尽管使用面向对象设计和编程通常是构建这类用户界面的好主意。在*[Col95]*中有更详细的例子。

基于表单的用户界面　在“基于表单的用户界面”里面，用户开始时选中某个业务处理（模块），然后应用程序就使用一系列的表单来引导用户完成整个处理过程。大型机系统上的大部分用户界面都是这样子的。*[Cok97]*中有更为详细的讨论。

面向对象系统　在这里指的是用面向对象技术设计和实现的应用程序。在本文里面，最重要的特征是它们自己有一套对象，这一套对象有自己的生命周期，而且相互作用，从而完成应用程序的功能。单个的用例或者事务不和架构相联系。

面向事务系统　　在这种系统里，系统架构围绕事务进行，每一个事务都会调用一个不同的程序来操作一个数据库。大部分使用事务处理器的程序，如 CICS 等，都是用的这种方法。[GrR93] 中有详细的讨论。

本文阅读指引

在阅读模式语言的时候，一般不会走马灯似的看，都希望能仔细地阅读和研究。熟悉了本文的结构和布局，对文中的模式，你想深入到什么程度就深入到什么程度。想看看某个模式是说什么的，可以看看其概要。只要阅读了每个概要，你就可以很容易地了解整个语言的概况。如果你对某个模式的详细思想有兴趣，则可以阅读“上下文”部分，“问题”部分，和“解决方案”部分；如果你对这个模式的约束以及解决方案的取舍有兴趣，可以阅读相关的“约束”部分和“结果”部分。两部分都有主要说明，以标准字体出现，而小字体则用来讨论这些陈述。下图 2 表示了建议的阅读途径。

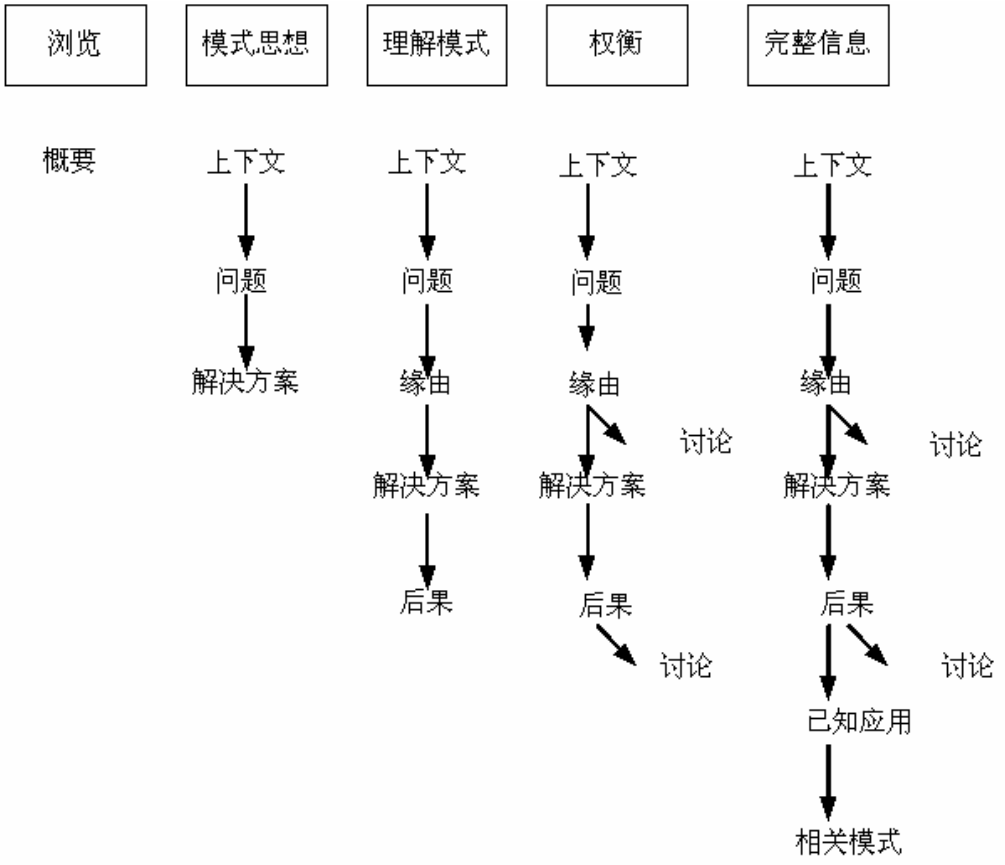


图 2

用户界面层（User Interface Layer）

概要　　驱动一个用户界面总是通常比域复杂和容易变化……所以，把域层面从用户界面软件里分离出来，将用户界面封装在一个独立的层面中。

上下文　　当设计一个复杂商业系统的架构时，通常从一个大概的架构开始：如都有哪些主干等。在以前，技术因素是首要因素，最终会在内部结构大量地应用技术。那是一些系统有几种“标准架构”的时代，是一个项目上有 100 多人的时代。今天，很多架构师在开始对技术进行考虑之前，首先使用域将系统分割成更小部分，称之为业务对象。

无论你走的是哪条路，迟早你都得考虑技术方面，将系统分的更细。其中一点就是如何将系统展现在用户面前——这就是这个模式所探究的。它不单应用在面向对象系统里，还应用在面向事务的设计，使用 *tool-materia*l 隐喻的系统*[Rie97]*，以及基于表单的应用程序*[CoK97]*等等。

问题　　该怎样将大型系统的用户界面溶入整个架构？

下面几件事情你得注意：

约束　　用户界面表示了和你的需求模型一样的功能需求……但是，它还有不同的非功能性需求，这些需求导致很多用户不喜欢碍人的细节。

用户界面只是你的系统的一个界面，所以，它的内容是和你的需求分析模型中的需求分析是一模一样的。一个初级方案是将所有的分析对象表示在用户界面上，将这些对象的方法作为界面上的动作。不过，这办法好吗？一个好的分析模型的目标是减少冗余，是进行抽象化，以支持更好的灵活性。在你对域对象进行设计时，要考虑这些：性能和耦合是主要的，还有并行性以及其它的避免软件工程师厌烦的方面。但是，这些要考虑的东西，用户根本就不想介入。好的用户界面仔细地依据用户在其领域的思维模型，尽可能地支持她的工作流程。当很多关于用户界面的改变需求来临时，一点也不用惊奇，这些改变对分析模型毫无影响。一群新的用户有不同的关于这个领域的思维模型，可能就需要一个新的用户界面，但哪怕是一丁点对象的改变都不用。因此，一个用户界面的非功能性需求和那些域对象是相差很大的。

一种标准的满足不同的非功能性需求的技术是对子系统进行不同的考虑……但是一个用户界面要求有大量的域层面的信息，以符合比较高的人机工程标准，所以，这些分开的子系统还是紧密地耦合在一起的。

一个软件架构师的标准反应是将不同的非功能性需求放在不同的子系统里面。这种方法在优化自己的子系统时比较自由。如果是用户界面的话，这样子就需要一个子系统管理用户界面，另一个子系统管理域层面。不过，这种方法好吗？子系统界定的一个主要标准是它们之间的相互传送的信息量。现代的用户界面给用户提供大量信息：列表框可以提供选择信息，树状示意图帮助定位特定的对象，灰色菜单和按钮让你避免做蠢事。所有的这些都会使得子系统之间的耦合性很高。

现代的开发环境通过特定方法可以很方便地操作用户界面……但复杂的系统需要有不同的界面来满足不同的需求，为了满足这些不同的需求，这些用户界面又可能需要不同的设计。

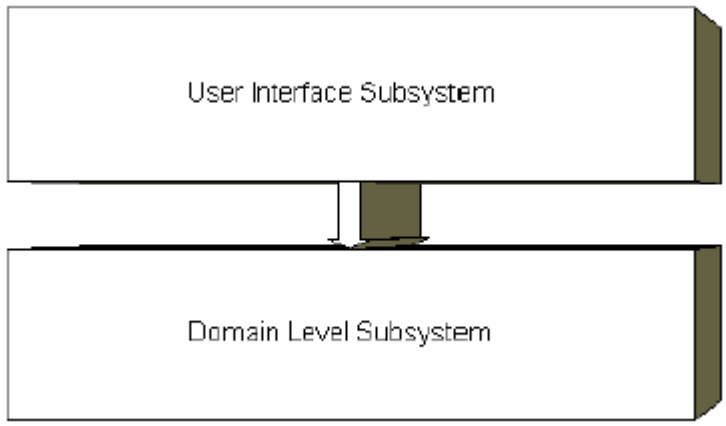
象 SmallTalk 或 Java 这些语言，通常需要一行代码来使得某个菜单选择失效，或者弹出一个简单的对话框来提醒用户。所以，当你的域层面的代码（domain code）里发生了一些奇怪的事情时就总想提示用户，问："要不要继续运行？（是/否/帮助）"然而，这真的是一个好主意吗？或迟或早你就会发现有不同的用户要对系统进行操作。它们必须每天重复上千遍的工作，希望按尽可能少的键来让这个系统帮他完成工作。一个弹出的对话框反而会妨碍他的工作，相反，他们希望系统把问题记录在问题记录表中，等以后再看。你就需要一个完全不同的用户界面布局。或者考虑一下你的系统在批处理运行模式中的情形：当然，如果让一个系统从早上 2:30 开始等到用户的决策，直到 9 点钟操作人员来了，看到的是屏幕上一个无成效的提示的话，是没有人会开心的。最后，你的系统可能在没有任何用户界面的环境中运行，比如，某个荒无人烟的地方的一个电话交换系统：没有人在现场操作，而是远程操作。这时如果在本地终端上显示一个对话框，而等待一个 300 英里外的操作员去应答，这可就不是一个好主意。类似的情形在全球商业系统中会出现得越来越多。所以一个远程操作界面就需要一个和需要输入大量数据的系统完全不同的架构，它需要一个和有高程度交互的系统不同的架构。大部分的开发环境不能很好地支持这些架构。

最后，域层面的软件可能有些抽象，用户不能理解……但是域层面的结构经常将这些显露在用户界面上。

在进行分析时，大部分系统都会变形，开始时是领域专家们很具体的想法，到了最后就成了抽象的对象模型，而且对可复用性，可维护性，是否容易设计等进行考虑。如果这个系统要满足广泛的需求，分析家们甚至可能会决定建立一个元模型。这些是软件专家而不是领域专家们的决定。通常情况下，最后得到的结果是领域专家看不懂的。为了保证可用性，你又得将抽象的领域模型转换回用户的角度。不过，这是一个好主意吗？最起码，将对象结构转换成不同的结构是额外的工作。假如领域层面的模型和你要展示在用户面前的模型不兼容，这个工作量可能就是一场噩梦。最好的例子就是一个在面向事务的领域模型上的面向对象的用户界面：这些系统通常情况下直观程度和性能是比较差的。

解决方案

所以要将用户界面和领域层面分离，放在两个不同的子系统里面。确保领域层面子系统不对用户界面部分进行任何引用，然后形成一个层状架构。



必须说明的是，这个方案里层状特性非常重要：用户界面层可以调用领域层，但反之则不行。当一个“回调”无法避免时，我们需要特殊的机制来保证这个层状方案，存取领域层（Access Domain Layer）讲的就是这个。

后果

使用这种架构很容易对两个层面的非功能性需求进行优化，但是你仍然需要小心不要将功能需求重复实现。

现在，两个层面可能有完全不同的设计。比如，用户界面层可能使用配件模型（Widget Model），以大量的视图来支撑，而领域层则基于分析模型。当布局设计发生变化时只会局部影响界面层，但是性能需求的变化则集中在领域模型上。但是，又有新的地方需要注意：如果在两个层面冗余地实现功能性需求的话，是有很大的风险的。例如，当一个对话框中没有符合业务规则的相关内

容时，“OK”按钮要被禁止使用。我们会很希望将这个业务规则放在对话框类里面，令一方面，在你的领域逻辑模型中也需要这样的标准（规则）。现在，当这个规则发生变化时，事情就乱套了：你得在两个或更多地方进行修改。为了避免这些风险，域模型经常会提供这个领域的亚信息。有效性方法（*Availability Method*）和领域层面类型（*Domain Level Type*）就处理这个问题。

层状设计让我们可以对它们分开考虑……但仍然需要一些预防措施，使得两个层面耦合，可控。

经过这种模式的严格训练，你可以确保将表示部分和领域层面部分严格分离。当你为职责分配（比如，应该放在界面部分呢，还是领域部分时）而举棋不定时，关键问题是：如果我将用户界面换成批处理方式，这个功能还需要不需要？作为一个大致的规则，如果答案是“需要”，可能将其放入领域层更合适一些；如果答案是“不需要”，你可能逮着了一个用户界面职责（功能）。但是，在最后在两个层面间还有一个很广的接口。这些信息可以如下分类：

用户当前在使用的对象的只读上下文信息，

- 格式化规则，
- 关于某个特定动作能否在当前上下文中执行的信息，
- 领域层面的状态和进度信息，
- 触发行为（事务），以及
- 错误信息

领域层面存取（*Domain Layer Access*）讨论这些话题。

两个层面之间的实际信息流量依用户界面风格而定。假如你的用户界面是基于对话框的，有一系列的表单，通常情况下就比“工具加材料”的模式上下文信息少一些。这就是为什么基于 Web 的应用程序和大型机系统通常使用表单而好的基于 PC 的系统则经常使用 tool-material 隐喻的深层次原因。

不是所有的现代开发环境都支持这种架构，假如支持的话，通常也需要更多的工作……然而，这种层次架构仍然使得你能自由地对同一个系统进行不同的表示。

很多流行的框架（Framework）采用以用户界面为中心的方案。当用户点击了一个按钮，框架就会调用相应的按钮对象的“点击”方法。至于在这个方法里面的动作则是程序员的事情。通常情况下，是没有标准的过程甚至类来管理和领域层的通信的。所以，就需要通过训练来保证分层－这不容易，特别是在进度安排紧张的情况下。但是，当你需要另外的表示时一个干净利落的分层可以节省很多工作。你只需要将图形用户界面类替换为批处理方式类，或者一个 CORBA 接口。即使你不打算使用 CORBA 接口，你也在分析操作时可以很快地给同一个领域层以不同的表示。例如，很多界面支持通过剪贴板和拖放（Drag & Drop）来复制。两者都要求有不同的用户界面活动和相同的领域层功能。

额外的层面可以使得用户界面适应一群用户的特别需求……但仍然不能把稻草变为黄金。

假如在领域层面使用了这个领域的元模型，顶层可以使用这些信息来组成一个界面，来组合成一个用户的思维模型。例如，你可以表示域对象的硬编码属性，用同样的方法表示属性列表，然后在用户面前隐藏这个属性列表。然而，领域层还必须支持转换。以一个面向事务系统的图形界面化为反例：很多客户想，用一系列的 HTML 表单来处理 3270 个基于主机系统的表单，来形成一个“现代的”用户界面已经足够。然而，一个成功的基于窗体的用户界面却来源于完全不同的范式。现代界面不对业务过程建模，但提供一个工具集由用户自己来修改业务元素。显示在屏幕上的大部分信息和用户在使用的特定业务过程没有关系——只是辅助用户的额外信息。从技术观点来看，这意味着需要要比传统的系统提供更多的信息，在传统的系统里面用户知道其业务代码，能正确地输入系统所需的数据。结果是这些系统的领域层被优化来处理事务，而不是提供所有的额外信息。在最好的情况下，你会撞上性能问题，在最坏的情况下，你不得不重写整个系统。

已知应用

几乎所有的流行的用户界面架构都使用这种模式。我在这里举三个例子：

1. Seeheim 用户界面架构的特点是有一个应用核心的领域层和一个用户界面层。后者被分为两层，叫做表示层和对话控制层。因为这个架构和面向事务系统有渊源，没有从应用核心到用户界面的回调机制。

2. 可能最流行的例子是模型－视图－控制器（MVC）架构*[BMR+96]*。视图和控制器类构成用户界面层，而模型类构成领域层。视图和控制器可以发送特定消息给模型，反之则不行。一个观察者（*Observer*）提供从模型到视图的回调机制。

IBM Visual Age for SmallTalk 允许程序员用部件（parts）构成应用程序，这个架构中有可视化部件（Visual Parts），例如菜单，树状图，窗体，还有非可视化部件包装了一般的 SmallTalk 类。有条理地使用的话，可视化部件对应于用户界面层，而非可视化部件则对应于领域层。

参考 [BMR+96]对层状架构进行了深入的讨论。

子系统架构

在你决定使用用户界面层（*User Interface Layer*）后，工作才刚刚开头。牢记分层的主要目的：封装复杂的用户界面。所以，通常情况下，将整个用户界面层实现为一块并不是一个好办法。因此，下一步就是寻找子系统。有四个子系统组成了整个架构：分离转换（*Separate Transformation*），配件模型（*Widget Model*），域层存取（*Domain Layer Access*），以及上下文支持（*Context Support*）。你几乎可以在每个用户界面架构里面找到 *Separate Transformation*, *Widget Model*, *Domain Layer Access* 的影子，但 *Context Support* 则视环境而言。

分离转换（Separate Transformation）

概要 将信息显示在屏幕上和处理用户的动作是两件复杂的不同任务。所以将用户动作到领域层调用的转换，和同领域信息到屏幕信息的转换分离。

上下文 在你决定使用一个用户界面层（*User Interface Layer*）之后，工作才刚刚开始。你得定义用户界面的详细结构，通常输入和输出会很清楚地分开考虑。特别是当你使用直接操作，和 Tool-Material 隐喻时。显示信息和处理操作都是非常复杂的。因此……

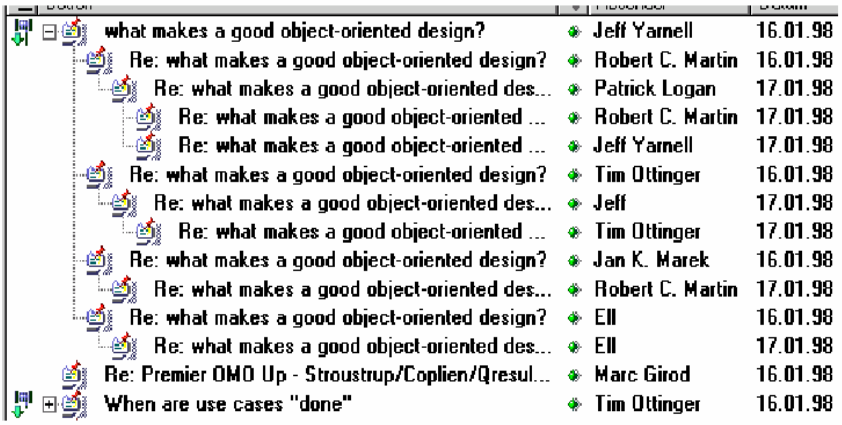
问题 如何将输入和输出过程分配给子系统？

要得到一个解决方案，你需考虑几个约束：

约束 可视化和操作是复杂的任务，但是它们是紧紧联系在一起的。

让我们来看一下一个新闻阅读器在不同线索下的树状图。它由几个不同的图形元素，如线条，图标等构成。所有的这些图形元素显示领域层的信息，比如状态等。对这些表示的控制需要一些很棒的复杂的软件。除了外观之外，这个窗体还要能用多种方式阅读帖子，进行标记，回帖，或

者保存。用户使用弹出菜单或下拉菜单，以及拖放技术来完成大部分的动作。控制这些动作，以及将这些动作转到领域层也是复杂的任务。所以，将可视化和操作分离的话，很有吸引力。不过，这个办法行得通吗？两部分都相互紧紧地联系在一起：首先，它们都在同样的领域层对象上，比如一个新闻帖子；另外，同一界面实体可能同时代表可视化和操作。比如每个作者名字前面的小菱形，在可视化方面表示目前我还没有阅读过这个帖子（尽管早应该读了），如果你点击一下它，阅读器就将相应的帖子标记为已读，这很明显是是一个操作任务。



可视化和操作强调了领域层的不同特性……但通常工作在同样的领域层对象上。

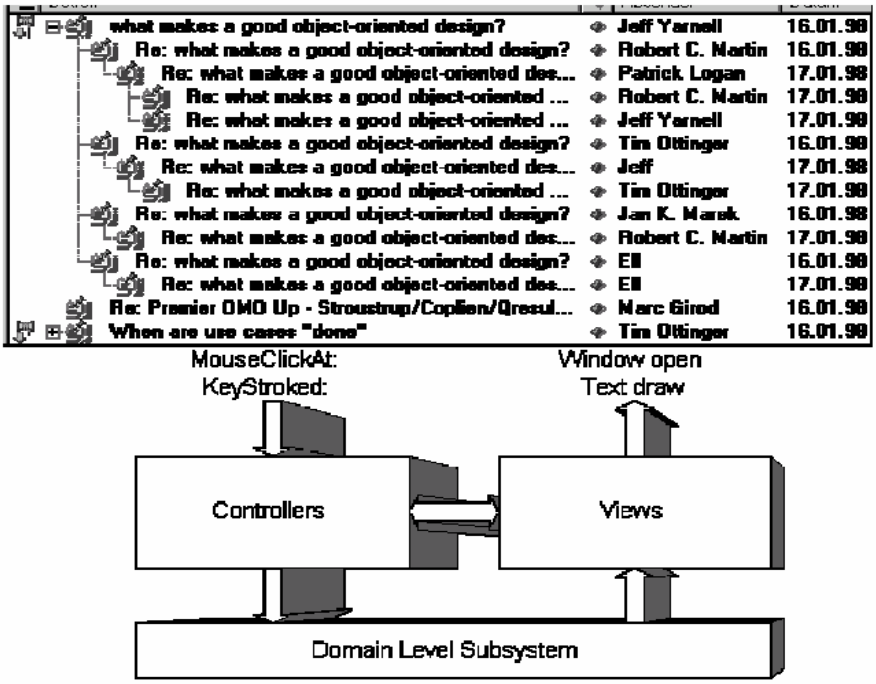
尽管可视化和操作可能在一个界面上看起来是相同的，它们通常强调的是领域核心的不同特性。最起码它们分别调用 getter 和 setter 方法。在树状图的一个上下文敏感的弹出菜单里面，菜单项可能代表这个节点本身，它的上层节点，甚至是一类节点的方法。所以一个认为可视化和操作是两个完全不同的方案非常有吸引力。但是，这真的是一个好方案吗？在一个设计良好的用户界面中，操作通常最起码和选中的对象有关。所以你一般从当前的选择开始。维护冗余的选择很容易导致出错。结果，如果你把可视化和操作分离的话，它们之间需要大量的通信。

你的设计需要直接的分离，但通常框架已经提供了一些分离。

有时候，基于其它约束，可能很容易找出一种最好的分离方法，所以，你倾向于在任何情况下去实现它。但这真的是好办法吗？在大部分情况下，你使用类库或者函数库，来调用用户界面控件的笨拙细节。大部分控件组成框架，有了一定程度的分离。换另一个，可能是可行的，但最多是一堆额外代码，同时，还有额外的软件需要编码，测试和调试。根据经验法则，改变你的框架的体系架构是不明智的，尽管对它进行扩展和优化是明智的。

解决方案

所以，将用户界面处理分为两个部分：视图（*View*）在屏幕上显示数据，控制器（*Controller*）处理用户输入。而让当前选择（*Selection*）来定义在对象上的上下文。一般来说，每一个单独的视图有自己的控制器。



后果

这种设计方案成功地分离了不同的任务，尽管两部分仍然紧密地联系在一起。

两个类都清晰地定义了职责，实现了不同的概念。如果你想让同一对象有不同的视图而有相同的操作机理，完全可以定义一个控制器类和多个视图类。但是，它们是紧密地联系在一起的。外观的改变经常可能会导致交互的改变，反之也一样。现在，领域层对象的改变甚至会影响另外两个类的对象。

控制器主要调用领域层对象的 setter 方法，而视图则通常使用 getter 方法，而且遍历这些对象的网络。不过，这些对象（控制器和视图）还是得一起在同一领域层实例上工作，所以你得做一件额外的工作－为它们公共上下文提供上下文支持（*Context Support*）。

在一个初级方案（以及不少第四代语言）里面，控制器仅仅设置领域对象的属性。在一个更复杂点的系统里面，操作就没有那么简单了。在用户进行某个动作之前，你得建立菜单，激活菜单项和按钮，而这些是依当前的领域层面的状态和用户所作的选择而定的，这些使得你不得不严格要求程序员。当一个用户完成了一个操作之后，可能你得去获取另外的参数，对它们进行检查

和校正，保存撤销信息等等。你可能可以猜得到，这些工作对一个单独的类来说是多了点。因此，控制器经常会使用命令（*Command*），*Availability Methods*，和 *Domain Level Type*。

视图的任务经常是比调用一个 getter 方法用一个字符串来初始化标签（Label）要复杂得多：结果要格式化好，而表示方式又依所调用的对象的实际子类而有很大不同。例如，一个电路图的视图：那些符号就要视你要显示晶体管还是电容而定。领域层类型（Domain Level Type）就帮助你做格式化。作为底线，视图和控制器表示了领域层对象原型的不同部分。但是，它们仍然引用相同的领域对象，通常就是用户已经选中了的那一个。这样，就会有两个可能：用户可能在一个配件，比如前面所说的在帖子“树状图”中选择一个对象，但在一个基于表单的用户界面里面，用户会通过一些查询的方式，如代码匹配搜索来选择对象。一般来讲，有另外一个上下文来定义视图和控制器使用哪些对象，因而你还需要另外的工作来进行上下文的支持（*Context Support*）。

在任何情况下，确保你的架构和这种分离一致。

通常，去实现一个不遵守你的框架所支持的架构的设计是不明智的。假如你认为对你的应用来说分离转换是好主意，但你的框架又支持配件模型的话，你可以要么选择另外一种框架，要么改变你的用户界面。万丈高楼平地起，不要想在一个只适合盖教堂的地方盖摩天大楼。只有象 AWT 这样的底层框架才同时支持两种架构。

已知应用

1. 模型视图控制器（MVC）*[BMR+96]*是这种模式的经典例子。模型就是领域层对象，同时也是上下文，而视图和控制器则和模式中的一一对应。

2. http，超文本传输协议也使用了这个模式的一个小小变种。如果将 html 页面看作视图，则 CGI 脚本就是控制器，脚本的参数就是上下文。无论你是否给页面清晰地定义了领域层的对象，更多的高级页面是使用 java applets 来实现复杂的用户界面，因为 AWT 支持配件模型（*Widget Model*），这通常会形成一个混合的架构。然而，AWT 是很底层的，所以可以使用 AWT 来进行分离转换。

3. CICS 也对可视化和操作提供两种完全不同的机制。业务代码决定了你要启动哪个程序一然后也决定了你要进行什么操作。Maps（映射）封装了到终端的输出。因为 CICS 是面向事务的，而不是面向对象的，它不维护任何上下文。有几种方法可以提供上下文支持（*Context Support*），我会在下面对其进行讨论。

4. Dirk Riehle 用“Trennung von Interaktionund Funktion”（交互和函数的分离）来描述这个模式，在 tool-material 隐喻的上下文中*[Rie97]*。

相关模式 如果视图和控制器之间的联系太紧密的话，配件模型（Widget Model）是另一个架构方案。实际上，在架构层面上用户界面架构通常使用配件模型，而在更低层面上使用分离转换。如果操作过于复杂，则你会发现相反的次序。

配件模型（Widget Model）

概要 屏幕上的每一个配件都有自己的数据和功能。因此，用户界面的一套对象就形成了屏幕上的配件。

上下文 大部分的图形用户界面有一套配件体系：窗体有子窗体，子窗体有自己的领地，上面有基本配件等。大部分的配件显示一些信息和进行一些操作。经常是每个基本配件表示一定的领域层对象或这些对象的一个属性。配件的集合也经常表示领域层对象的集合。因此……

问题 假如你的配件结构和你的领域模型的关系过于紧密时你怎么建立用户界面？

为了得到解决方案你得考虑几个原因（约束）：

约束 你想用这个架构去定义尽可能多的结构，但是通常情况下很难找到一个简单的架构来覆盖所有的主题。

你的架构定义的越好，你就越容易得到一个统一的容易维护的系统。特别地，一个没有经验的架构师则倾向于找覆盖了所有需求的架构，而这些架构在十年后才可能出现。不过，这是个好方法吗？简单性是一个架构的最重要的特性之一。越是复杂，出严重错误的风险也就越大。简单而功能强大是大部分架构所追求的。

你想让类之间的耦合程度低一些，但同时你又希望它们的职责尽可能地少。

这听起来象是一个空前的动力二人组，但是这是形成这个模式的重要约束之一。在糟糕的设计里面，用户界面的耦合有两个来源：几个类一起表示同样的领域层对象，以及用户界面元素，而后者又要一起形成用户看到的完整的用户界面。为了消灭第一个来源，你可以将所要表示的领

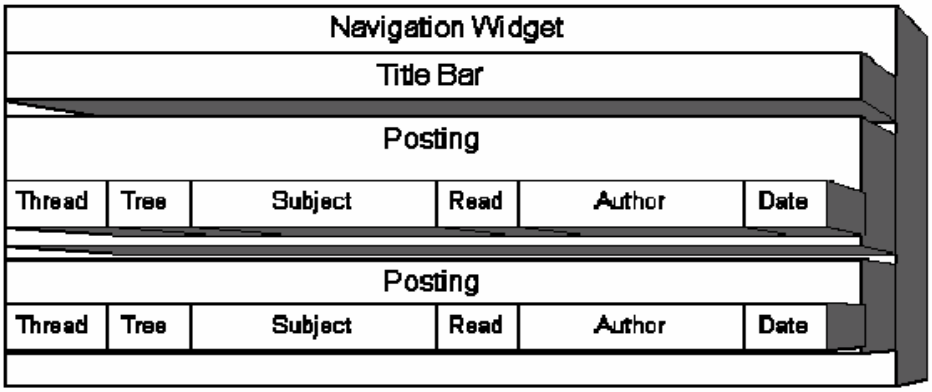
域对象的知识放在一个单独的类里面；而为了消灭第二个来源，你可以使用一个所有用户界面的通用原型。不过，这是好方法吗？将所有的领域表示放在一个类里面，可能会导致巨类，象一个什么都有的购物清单，而不是一个严格地一个类表示一个概念的方案。要获得你所需的灵活性，通用原型也非常难以定义。

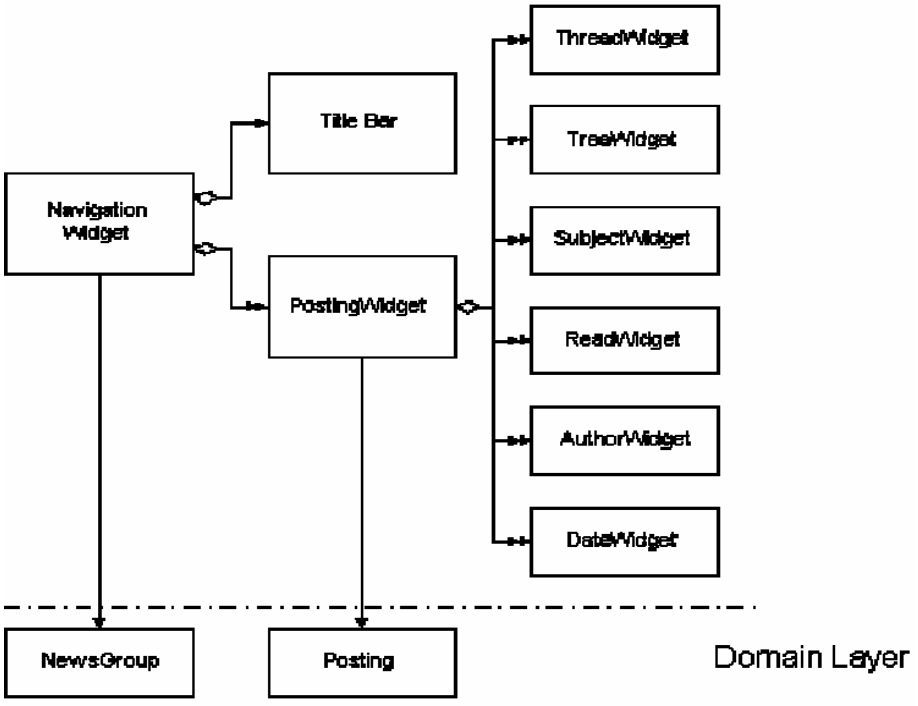
当碰到一个相同的配件时，你想直接复用一些类，但有些时候，你用元信息可以得到最好的复用。

这听起来也好像是一个非常普遍的约束，但也需要深入地探究。给定了用户界面的复杂度，最好的代码是你不需要重新写的那些代码（根据 Kent Beck 的说法）。特别是当你看到大型系统基于表单的用户界面时，你通常会发现一些重复来重复去的子表单，它们相互之间只有一些很小的差别。自然而然地，你想用相同的类来表示这些子表单。但是，这个主意好吗？要复用这些类就意味着它们要足够灵活，可以覆盖你所有的用法。假如，你的所有用法都是一样的，很好办。假如它们都是不同的，你要么为每一个特定用法配置一个类，要么和其它策略一起使用让这些不同的使用方法就范。两者都可能导致爆炸式的演化，最后一团糟。假如你的那些表单依其领域而不同，你可能可以使用另一种完全不同的方案：你可以从表单的元信息里抽象出类。需要注意的是在一个灵活的环境中也可能可以这样做。不过，这个架构仍然需要为所有你要产生的类定义一个小规模的，统一的原型。

解决方案

所以，在用户界面上的每一项都有一个简单的配件（Widget）类。这个配件类链接到一个完整的领域层对象或这个对象的某个属性。它负责处理这个领域对象的表示和操作。这些配件对象形成一个体系，和用户界面上的元素形成的体系对应，根对象（root object）就对应于主窗体。这个配件体系中的每一个节点都控制着衍生下来的节点的生命周期。下图就是新闻阅读器的设计演示。





后果

这个架构顺利地定义了用户界面的结构，而且这些结构的粒度比较好，不过，还是有些重要的事情要说。

如果画一张你的用户界面的图，这个模式定义了这个软件的大部分结构：将图形转换为用户界面元素的体系，比如窗体，子窗体等等。这是当窗体被激活时的一个实例树形结构图。但是，事情还没有完：大部分的应用程序都有一个跨越窗体的上下文，以剪贴板为例。你不能将剪贴板和某个可视化部件绑在一起，你需要另外一个类来支持上下文（*Context Support*）。另外，还需要一个总体的类，处理在打开第一个窗体之前的初始化工作和关闭最后一个窗体之后的清理工作。一个分离的应用程序（*Application*）类处理这个主题。

假如你改变了领域对象，你只需要改变一个单独的类……不过，这个类还有一大堆复杂的职责。

在面向对象的用户界面里面，配件模型也意味着反映领域对象。一个配件经常表示一个单独的领域对象或一部分。因为配件负责和这个对象的所有交互，这个配件就专注于这个类。但是，这可能会导致这些类变得复杂。在更坏的情况下，一个配件不仅要关注和领域对象的双向通信，而且还要管理衍生配件的生命周期和布局。对一个复杂的配件而言，这经常会导致类过快增长。因此，复杂的配件经常使用分离转换（*Separate Transformation*）作为微架构。另外一种方法是使用自动布局（*Automatic Layout*）来帮助管理布局。其它有助于分解类的有 *Domain Layer Access*, *Command*, *Availability Methods*,和 *Domain Layer Type*。

一个配件模型（*Widget Model*）导致自然而然的对配件的复用，不过，对于衍生的配件仍然不大合适。

复用一个配件是很直接的：你只需要从另一个父配件相同的类建立一个新的实例即可。但是，你衍生得越多，类也会变得越复杂，因为需要更大的灵活性。为了适应所有的衍生配件，你必须实现钩子（*hooks*）和策略（*strategies*）。要摆脱这种情形，最简单的方法就是重新设计用户界面，避免任何衍生情况。

要从领域对象的元信息中产生出配件是不容易的。

对一些基于表单的用户界面，知道了一个表单中的对象和它们的属性就可以产生配件。给表单类设一个属性列表，运行一个自动布局（*Automatic Layout*）显示。你可以用这种方法来产生一大堆的“插入，更新，删除”表单的操作。有一个很重要的地方要指出来的是操作部分是稳定的，仅仅数据会变化。这种方法也要求每个表单的的原型是高度标准化的。一个配件模型不鼓励两种需求。就这种方法而言，分离转换（*Separate Transformation*）是一个更好的架构。

已知应用

1. 应用文档视图（*Application Document View*）是这种方案的最流行的例子。Macintosh 首先使用这种架构，大部分的窗体库都采用了它。应用程序和文档管理上下文，而视图是打开的窗体的根。每个视图由子视图，配件等构成。

2. 表示抽象控制（*Presentation Abstract Control*）*[BMR+96]*将这个概念扩展到域层面，形成了一个相互协作的“代理（*Agents*）”体系。

3. AWT，Java 的图形用户界面库，用“构件”组成用户界面，用“构件”管理数据的表示和操作。

4. Visual Basic 是使用这种架构的最流行的非面向对象的开发环境，一个“表担”由“子表单”递归地组成，“子表单”最后由“控件”组成。它们都有一些事件，在用户选择了做某件事情时就会被调用。用 Visual Basic 做大项目的一个主要问题就是缺乏为每个表实现更细微的架构的机制。

相关模式

如果你能把输入和输出清楚地分离，则分离转换（Separate Transformation）也是可以的。经常是两个模式绑在一起，形成一个视图－控制器的衍生体系。

责任链（Chain of Responsibility）[GHJ+94]是管理配件之间通信的经典设计模式。

中介（Mediator）[GHJ+94]是减少几个子配件之间耦合的一般方法。

域存取（Domain Layer Access）

概要

领域层面的界面有时候是比较复杂的。因此要指定一套对象来存取领域核心。

上下文

如果你决定使用一个使用界面层，层面的结构不仅告诉你如何给用户界面构建类，也要考虑功能上的需求。花俏的窗体和表单帮助用户使用系统，但它们本身没有任何价值。因此……

问题

你怎样从用户界面软件里使用领域层的功能呢？

解决方案 给领域层引入一条方法和类。确保你在设计这些类时考虑到下面的事项：

- 领域对象的命令，命令（*Command*）模式负责这个主题。
- 命令出现异常时的错误处理。命令（*Command*）模式是一个处理错误的好地方，错误处理器（*Error Handler*）*[Ren96]*对更加复杂的错误处理更合适。
- 为了获得命令是否有效的信息，你可以使用 *Availability Method* 作为通用的原型。
- 要获得各入口的可能值，*Domain Data Type* 对这个问题进行了讨论。

观察者（Observer）

概要 有时候，你要用户同时显示同样的领域对象的不同视图，但又不想在领域对象改变时让领域对象调用用户界面。因此，让视图在领域对象上注册，当领域对象发生变化时就向所有的注册过的对象发送变化通知。

参考 *[GHJ+94]*

命令（Command）

概要 给一个领域对象发送命令经常和额外功能联系在一起，比如决定操作的有效性，获取其它参数，错误处理，撤销功能，以及为事务设定边界等。因此将这些命令按各自的职责形成不同的类。

参考 *[GHJ+94]*

有效性方法（Availability Method）

概要 在大部分的用户界面里面，你需要在当前的上下文中不用真的执行某些动作而检查某个动作是否合法。因此，给每个针对领域对象的操作专门加一个有效性方法（*Availability Method*），传送和这个操作相同的参数而返回一个布尔值，不用对领域层作任何改变。

领域层类型（Domain Layer Type）

概要 一个入口数值的格式和合法性通常由要显示或输入的数据的本质来决定。因此，专门使用一些类，有领域层类型的公共原型，这些类的主要职责就是决定入口数据的格式和合法性。

参考 整个值（*Whole Value*）*[Cun95]*

上下文支持（Context Support）

概要 特别地，在一个面向事务的环境里，系统不会存储可以更好地帮助用户的上下文信息。因此要考虑为每个用户支持上下文。

上下文 当用户在盯着显示器搔头时，用户界面经常需要维护一些上下文信息。你得保存诸如显示的对象，当前的选择，对话框的状态等信息。

问题 你将这些信息保存在何处？

解决方案 引入保存上下文的机制。你的分布体系，客户端和服务器的有效带宽和你的系统所提供的可扩展性决定了这些机制。对于单用户系统和胖客户端架构，应用程序，文档和选择帮助处理信息的上下文。对于瘦客户端架构，可以用对话记忆来牺牲可扩展性优化带宽，或者用 *Cookie* 来牺牲带宽优化可扩展性。对于高度复杂的系统，可以将两者结合起来更好地调整。

应用程序（Application）

概要 在一个窗体环境中，需要一个地方来存储那些不能在窗体中保存的信息。所以就使用一个应用对象（*Application Object*）作为单体（*Singleton*）*[GHJ+94]*，负责初始化，清除和保存一般信息。

文档（Document）

概要 在大部分的窗体环境中，用户可以在同一套领域对象上打开多个视图，所以就给每一个窗体分配一个文档（*Document*），管理所表示的领域对象的根。

选择（Selection）

概要 因为当前选择控制着一个用户界面的大部分元素，很难将其分配给其它元素。所以就引入一个选择类（*Selection Class*），其中有当前被选择的领域对象的引用，使用这些引用将所有的操作动作发送给领域对象。

对话记忆（Session Memory）

概要 在一个基于事务的环境中，系统不保留两个事务间的上下文，但是对话框又经常是跨越几个事务的。所以，就为在这个主机上的每一个客户端分配对话内存，以保存上下文信息。

Cookie

概要 在一个有大量的或者不可预料的客户端的 C/S 结构中，服务器端可能会因资源短缺而不能给每个客户端保存上下文信息。所以以数据的形式给客户端发送数据，让客户端保存。这一小块上下文信息就叫做 Cookie。

基于表单的用户界面的一些特殊模式

集中控件（Centralized Control）

概要 基于表单的控件经常有这样的特点，一个复杂的状态机模型，有很多的公共事件，状态变化的频率非常高，而每一个对话的单个步骤又是一个样的。所以，引入一个集中的对话控件来维护这些状态。

自动布局（Automatic Layout）

概要 如果你有很多的表单的话，各表单上的各个配件的相互位置的安排是一件非常麻烦的工作。但它也是遵循可计算的。所以，将这些规律在一个布局构件里面实现，而不是手工来画每个表单。

致谢

在这里我要感谢每一位在我写这份文档的时候支持过我的人。Wolfgang Keller, Andreas Hess, Uli Zeh 和 Klaus Renzel 在这个模式语言的早期版本中提供过帮助，Jim Coplien 以深邃的眼光发现了它的结构，和Alistair Cockburn 长时间地讨论，帮助去掉了过程模式（*Process Patterns*）以及发现模式的名字。特别地，我要对我的 PloP 引导者 Bob Hanmer 的帮助和耐心表示感谢。

参考文献：

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Michael Stal, Peter Sommerlad: *Pattern-Oriented Software Architecture - A System of Patterns*; John Wiley & Sons, Chichester, 1996

[CoK97] Jens Coldewey, Ingolf Krüger: *Form-Based User Interfaces - A Pattern Languag*e; Proceedings of EuroPLoP '97, Siemens AG, TR 120/SW1/FB, 1997

[Col95] Dave Collins: *Designing Object-Oriented User Interfaces*; Benjamin Cummings, Redwood, California; 1995

[Coo95] Alan Cooper: *About Face - The Essentials of User Interface Desig*n; IDG Books Worldwide, Foster City, California, 1995

[Cun95] Ward Cunningham: The CHECKS Pattern Language of Information Integrity in Coplien, Schmidt: Pattern Languages of Programming, Addison Wesley, 1995

[GHJ+94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Softwar*e; Addsion-Wesley, 1994

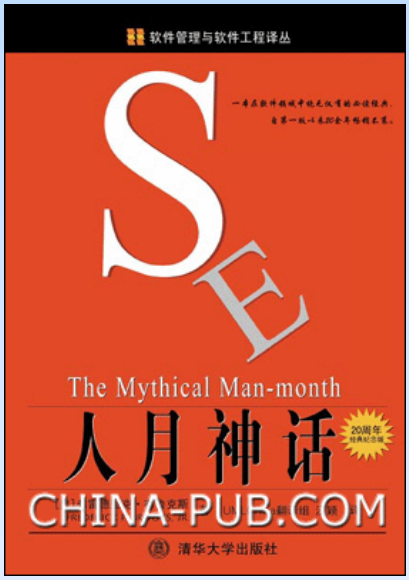
[GrR93] Jim Gray, Andreas Reuter: *Transaction Processing - Concepts and Techniques;* Morgan Kaufmann Publishers, San Francisco, California, 1993

[Ren96] Klaus Renzel: *Error Handling - A Pattern Languag*e; sd&m GmbH&CoKG; 1996; available via http://www.sdm.de/g/arcus/

[Rie97] Dirk Riehle: Entwurfsmuster für Softwarewerkzeuge - Gestaltung und Entwurf von Anwendungen mit grafischer Benutzungsoberfläche; Addison-Wesley, Bonn; 1997

[Tog92] Bruce Tognazzini: *TOG on Interfac*e; Addison-Wesley, Reading, Massachusetts, 1992

《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

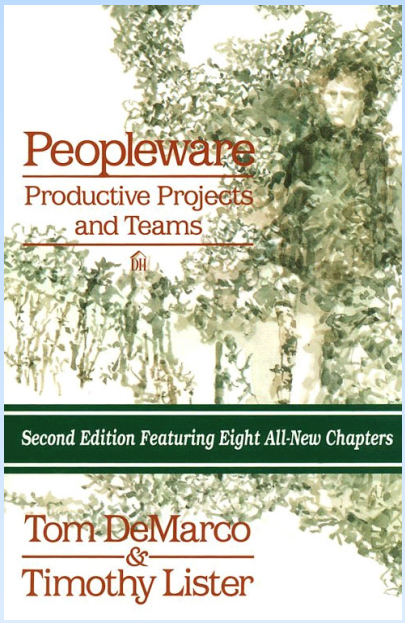
在所有恐怖民间传说的妖怪中，最可怕的是人狼，因为它们可以完全出乎意料地从熟悉的面孔变成可怕的怪物。为了对付人狼，我们在寻找可以消灭它们的银弹。

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！

《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

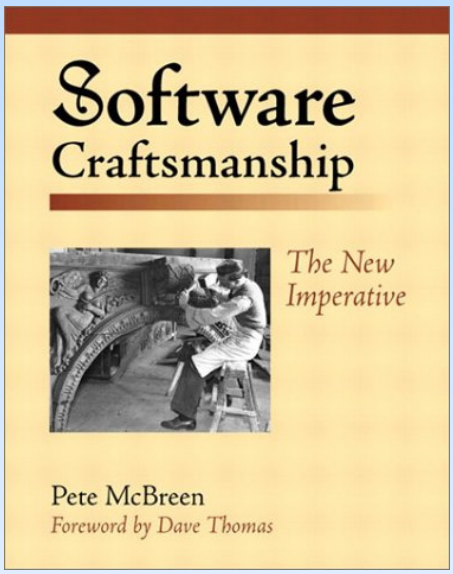
在一个生产环境里，把人视为机器的部件是很方便的。当一个部件用坏了，可以换另一个。用来替换的部分与原来的部件是可以互换的。

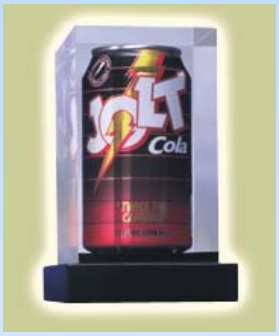
许多开发经理采用了类似的态度，他们竭尽全力地使自己确信没有人能够取代自己。由于害怕一个关键人物要离开，他强迫自己相信项目组里没有这样的关键人物，毕竟，管理的本质是不是取决于某个个人的去留问题？他们的行为让你感到好像有很多人物储备在那里让他随时召唤，说"给我派一个新的花匠来，他不要太傲慢。"

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！

《软件工艺》





2002 Jolt Awards 获奖书籍

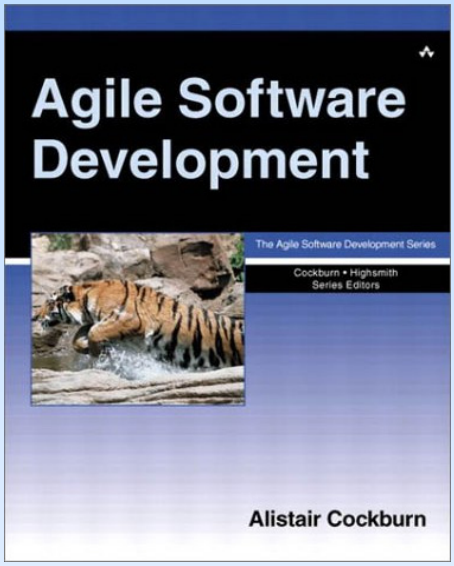
Pete McBreen

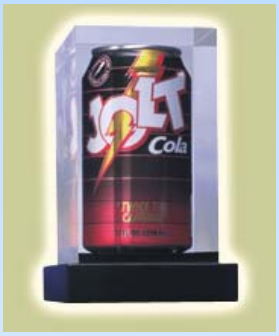
翻译：UMLChina 翻译组

工艺不止意味着精湛的作品，同时也是一个高度自治的系统－－师傅（Master）负责培养他们自己的接班人；每个人的地位纯粹取决于他们作品的好坏。学徒（Apprentice）、技工（Journeyman）和工匠（Craftsman）作为一个团队在一起工作，互相学习。顾客根据他们的声誉来进行选择，他们也只接受那些有助于提升他们声誉的工作。

中文译本即将发行！

《敏捷软件开发》





2002 Jolt Awards 获奖书籍

Alistair Cockburn

翻译：UMLChina 翻译组 Jill

我是一个好客人。到达以后，我把我的那瓶酒递给女主人，然后奇怪地看着她把酒放进了冰箱。

晚餐时她把酒拿了出来，说道，“吃鱼时喝它，好极了。”

“但那是一瓶红酒啊。”我提醒她。

“是白酒。”她说。

“是红酒。”我坚持道，并把标签指给她看。

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

我们大笑，然后回顾我们为了验证各自视角的“真相”所作的努力。究竟为什么，她问道，她已经看过这瓶酒很多遍却没有发现这是一瓶红酒？

中文译本即将发行！

通俗而且深刻的软件开发思想杰作—UMLChina 推荐

《面向对象项目求生法则》

《面向对象项目求生法则》

Alistair Cockburn

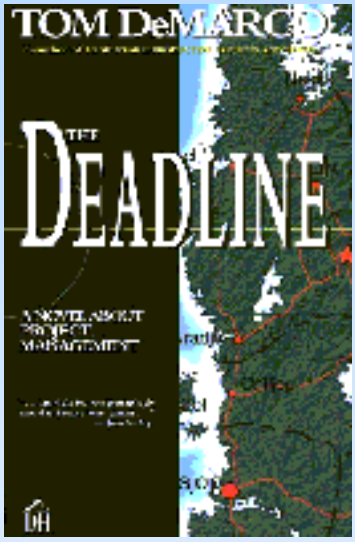
翻译：UMLChina 翻译组乐林峰

Cockburn 一向通俗，本书包括十几个项目的案例

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

中文译本即将发行！

《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！

分析模式学习笔记：LOG－日志记录模式

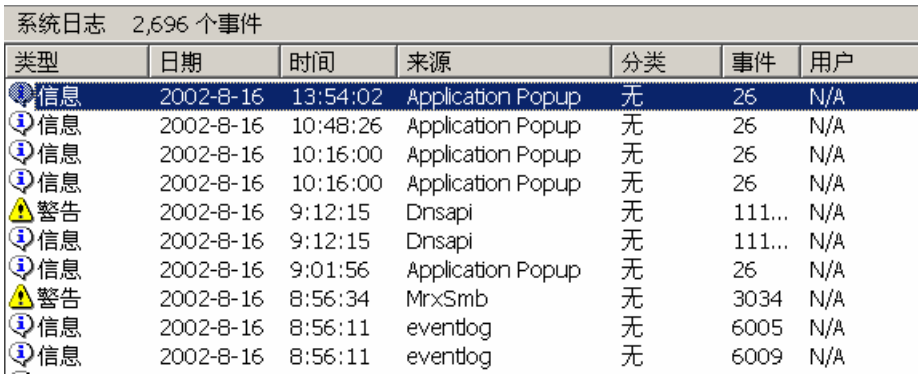
Windy.J 著

查看评论

介绍

*轻轻地我走了，正如我轻轻地来，我挥一挥衣袖，不带走一片云彩。*这样的情况在有了日志记录的系统里是不可能发生的，因为，日志把发生的一切都“记录在案”了，这一节，我们就来看看日志记录建模和实现的各种思路。

我们首先来看一张 Windows 系统日志的图片：



系统日志 2,696 个事件

类型	日期	时间	来源	分类	事件	用户
信息	2002-8-16	13:54:02	Application Popup	无	26	N/A
信息	2002-8-16	10:48:26	Application Popup	无	26	N/A
信息	2002-8-16	10:16:00	Application Popup	无	26	N/A
信息	2002-8-16	10:16:00	Application Popup	无	26	N/A
警告	2002-8-16	9:12:15	Dnsapi	无	111...	N/A
信息	2002-8-16	9:12:15	Dnsapi	无	111...	N/A
信息	2002-8-16	9:01:56	Application Popup	无	26	N/A
警告	2002-8-16	8:56:34	MrxSmb	无	3034	N/A
信息	2002-8-16	8:56:11	eventlog	无	6005	N/A
信息	2002-8-16	8:56:11	eventlog	无	6009	N/A

还有另一个文件记录日志的显示输出（见下图）：

```
08/15/2002 22:37:20 - 已使用内存：51%
08/15/2002 22:37:20 - 物理内存：  253424 KB
08/15/2002 22:37:20 - 页面切换文件：  612260 KB
08/15/2002 22:37:20 - 页面切换文件 (Avail)：  483572 KB
08/15/2002 22:37:20 - 物理内存 (Avail)：  123548 KB
08/15/2002 22:37:20 - 虚拟内存：  2097024 KB
```

由上面的图片可以看到，日志是一种重要工具，可以看到运行状况，显示一些跟踪信息，进行审计和安全性分析。它在软件系统中的应用也非常广泛，系统运行的时候，可能想记录一些信息，表示系统运行的状况，一些提示，不影响运行的小错误，以及其它想要记录的事件和说明，类和属性的变化情况，等等。日志记录在商业软件系统中非常常见，例如，Windows 的事件查看器里就有三种日志：系统日志，安全日志，和应用程序日志。

通常日志关心的信息是，谁，什么时候，发生了什么事，因此时间和事件描述是必不可少的；

记录日志的形式主要有文件和数据库两种，因为日志信息需要阅读和分析，所以，合理的日志格式也很重要；然后，还有日志将随时间增长，那么对它的空间管理也是需要留意的地方。

如何在我们的系统中加入日志记录呢？

全局函数

在 C 和 C++时代，一个简单而直接的方法是利用全局函数，设计好一个稳定的日志记录接口，把实现日志记录功能的全局函数放在公共模块中，其他需要调用的地方就调用这个全局函数，这样做的好处是，无需引入类与类之间的关系，也没有额外的初始化工作，完全由全局函数来管理跟日志有关的事务，这样的实现在 C 或者是小型的 C++系统里是可以接受的。例如一个简单的函数如下：

```
void log(String SubjectName, String Description)
{
    …
}
```

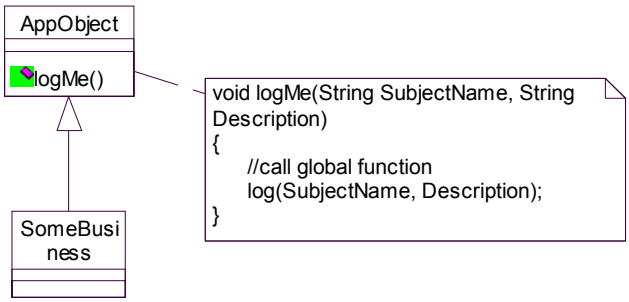
它的不足之处，就是代码里到处充满了这种突兀的句子，象现代高速公路上还经常出现牛车一样不协调。

公共父类包装

对这种方法进行改进的一个思路是，系统中的类都从一个父类派生，然后在公共父类实现一个记录日志的方法，在这个方法里面调用全局函数来完成日志的记录。这样还是省去了类间的初始化，并为整个类体系结构提供了统一的日志记录接口。它的缺点是，只能提供一种日志记录方法，而且当记录方法需要改变时（例如从文件方式改到数据库方式），需要直接修改全局函数。但是这种方法仍然是非常经济、统一的方法，对小系统或者对日志的要求比较稳定的系统已经完全足够了。

```
Class AppObject          //系统公共父类
{
    …
    void logMe(String SubjectName, String Description)
    {
        //调用全局函数
        log(SubjectName, Description);
    }
}
Class SomeBusiness : public AppObject    //其他类从公共父类派生
{
    …
}
```

图示如下：



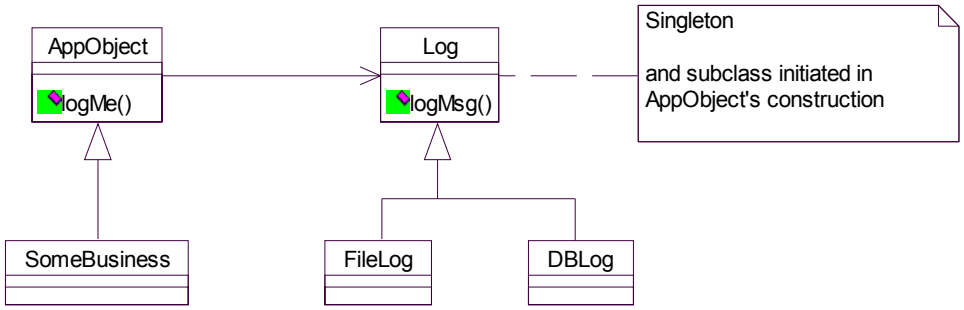
使用的时候：

```
SomeBusiness abusiness = new SomeBusiness(…);
abusiness.logMe(“abusiness”, “Created”); //直接调用基类实现的 logMe 函数
```

但是，在 Java 和 C#里已经不支持全局函数啦，所以对这两种语言来说，上面的方法是不行的。

日志类

那么一种解决方法是，委托一个专门的日志类完成事件日志工作；并可以派生子类来进行变化，只要保持日志记录的接口一致就可以了。缺点是需要在每个使用该类的类里初始化日志类的指针。解决方法是声明事件日志类为单子（Singleton 模式），然后在父类构造函数里调用事件日志类的静态函数初始化一个事件日志类的指针，如下图所示：



这种方法看起来接口一致，实现也不复杂，它的缺点是在父类的定义中已经选择了某种日志记录类的具体类型，如果确信无需考虑系统在这方面的变更，那么它是可行的；如果系统可能需要进行改变，它就成为一个非常遗憾的地方：我们不得不改写公共父类的实现，进而重新编译整个系统。当然，父类初始化时采用一些配置文件（或配置信息）可以减轻这种影响。

对此方法的扩展是，再委托另外一个类（日志配置类）来负责日志类的初始化工作，等于保证了公共父类初始化日志类时的接口，如果需要改变日志类的具体实现，就采用派生子类等方法；如果需要改变父类引用的日志类类别（日志类的不同子类），可以改变辅助日志类的实现。这样跟上面的方法区别不大，只是把变化的部分转嫁到了日志配置类而已。

逐个打包

如果系统的类不能从一个公共的父类派生，那就需要在每个使用日志类的类中留下接口，把日志类打包进入该类，然后使用日志类。这种方法适用于：

- 只要少数几个类需要记录日志，那么不需要整个类体系结构都提供接口；

- 各个需要记录日志的类各自需要不同的方法来记录日志，可以灵活改变自己的日志类类型。
- 缺点是一致性不好，并增加了初始化的工作量。

简单的样例伪代码如下：

```
public class SomeBusiness
{
    //每个类都需要提供登记 Log 类的接口
    //要不就在构造函数里增加参数
    void registerLog(Log logarg)
    {
        log = logarg;
    }
    void logMe(String SubjectName, String Description)
    {
        //使用 Log 类
        log.logMsg(SubjectName, Description);

    }
    private Log log;
}
```

小结

上面介绍了几种在应用系统中实现日志的模型，在 Martin Fowler 先生的 Audit Log 一文中也提到了日志，“A

simple log of changes, intended to be easily written and non-intrusive. ”上面的每一种方法都在一定的情形下适用，可以称为经典日志模型，同时我还高兴地看到，有些系统提供的类或第三方提供的类自动支持日志功能。

另一种出路

但是这些方法每一种有着同样的特点，那就是要求程序员在需要调用的时候写上相应的语句（例如logMe(SubjectName, Description);），这些句子将充满在代码中，和其他实现系统行为的代码混杂在一起，并给代码编写和代码检查带来额外的工作量，由于以上的解决模型，它们看起来理应如此，然而，这个问题可以解决吗？

说到这里，我忍不住激动起来，是的，可以解决！

要解决上面这个问题，我们得改变一下思考的方法，在以往的面向对象分析和设计中，我们认为系统是由类和类之间的协作组成的，就象演戏一样，布景，演员，演员之间的关系，演员之间的语言，动作来组成一部戏，但是在后台，还有一些别的参与者，例如化妆师，他的职责是为每个演员化妆；导演，他会干预到每个演员的演出。也就是说，我们换一个角度，看到系统里还有另外一种类，他们的特点是跨越了多个类的内部职责，例如日志记录就是这样一种典型的类，类似的还有对上下文敏感的错误处理，性能优化等，这种关系也叫“横切关系”（*crosscutting concerns*），统一组织这样的关系，并进行模块化，就可以解决上面的问题了，对，它就是面向 Aspect 的编程/面向 Aspect 软件开发（AOP/AOSD）。

在面向对象分析和设计中，系统的最小单位是类，然而有些系统行为跟许多类都紧密相关，AOP/AOSD 可以将这种传统思维方式下非常松散，分布在系统各个地方的行为重新用一种有效的方式组织起来，进行模块化，可以大量精简系统用传统方法实现的代码，并保持相差无几的性能。它通过 *Aspect* 单元，把跟横切关系有关的实现集中起来，这样，所有跟横切关系有关的功能非常容易管理，可以方便地插入，修改，删除，甚至重用。

Aspect 单元的典型形式如下(AspectJ 样例代码)：

```
public aspect AutoLog{

    pointcut publicMethods() : execution(public * org.apache.cactus..*(..));

    pointcut logObjectCalls() : execution(* Logger.*(..));

    pointcut loggableCalls() : publicMethods() && ! logObjectCalls();

    before() : loggableCalls(){
```

```
        Logger.entry(thisJoinPoint.getSignature().toString());
    }
    after() : loggableCalls(){
        Logger.exit(thisJoinPoint.getSignature().toString());
    }
}
```

在上面的代码中，首先有 aspect 声明（类似于类声明），然后有 pointcut 指定需要切入的点（*join point*），在这里，第一个切入点是 org.apache.cactus 包中的所有公用（*public*）方法的执行；第二个切入点是 Logger 类的所有方法，第三个切入点是以上两个切入点的逻辑计算。然后用 before():和 after():表示在切入点之前和之后的切入操作，在这里是调用 Logger 类进行记录。

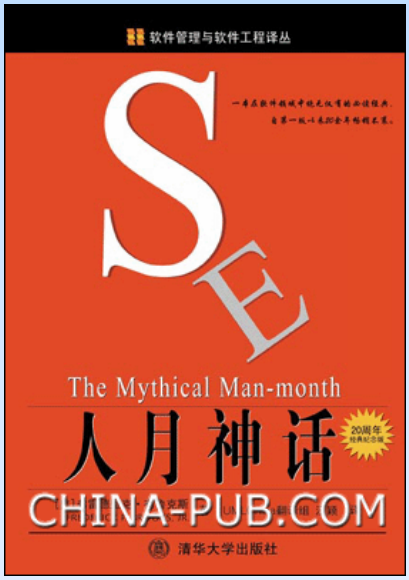
用类似以上这样的代码就可以替换掉经典模型中的成百上千条 Logger 类方法的调用，是不是很神奇呢？因为 AOP/AOSD 解决了面向对象和其它过程性语言未能处理的一块问题空间，它是一项值得立即使用的“多面编程方法”，也从实现的角度补充了我们对于这类问题的分析模式。Grady Booch 曾经说过，过去五年里发生的几件大事，包括面向对象技术成为主流，UML 语言的创造，设计模式概念，在这些过程中，一步步提高了软件系统的抽象层次同时降低了复杂度，多面运动的发展也同样如此，而在他的“Through the Looking Glass”（*Software Development*，2001 年 7 月）一文中将 AOP 引用为编程领域内首批多面运动之一（多种方法快速编写同一软件）。

当然在这里只是介绍了 AOP/AOSD 基本的概念和一个样例用法，如果想对 AOP/AOSD 了解更多，请访问 http://aosd.net/ 站点，同时，Xerox 使用“Mozilla 公共许可证”（Mozilla Public License）发布了 AspectJ，它是开放源码的，而且现在产品免费，这意味着用 Java 构建的系统可以马上应用令人兴奋的 AOP/AOSD 了，而对 C++，C#的 AOP/AOSD 扩展正在研究之中。

参考文章：

1. Audit Log MartinFowler，http://martinfowler.com/ap2/auditLog.html
2. 使用面向 Aspect 的编程改进模块性（Improve modularity with aspect-oriented programming）Nicholas Lesiecki（ndlesiecki@apache.org），http://www-900.ibm.com/developerWorks/cn/java/j-aspectj/index.shtml

《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！

电子商务应用系统的几种模式

Dragos A. Manolescu, Adrian E. Kunzle 著, 杨德仁 译

查看评论

摘要

软件开发业见证了从桌面应用系统的开发向高扩展的、分布式的、基于服务器的电子商务应用系统的开发的转移。大多数开发者来自于 PC 世界，很少知道如何处理分布式应用、服务器、并发性、扩展性、高可用性和容错事宜。电子商务应用开发模式将使开发者意识到他们需要处理的核心问题，并向他们展示解决这些问题的方法。本文的构建电子商务应用系统的模式集迈出了这方面的第一步。

开发软件困难，开发电子商务应用软件更困难。当前的电子商务是分布式、C/S、并发和网络化系统的混合体。所有这些体系要求解决核心问题。因电子商务处于现代商业的前沿，它也要求处理可扩展性、高可用性和容错性。

对于外行而言，开发电子商务应用软件如同构建“传统的”应用软件。然而，电子商务的特征使得前者比后者更具有挑战性。AEA[A2000]等人评述说：“在分布式系统上执行的软件代表着应用代码和管理需求如用于多样性、可扩展性、安全性和可用性的代码的一种唯一合成。”构建电子商务应用软件的开发者应该意识到将要遇到的障碍。电子商务应用软件模式有助于开发者预先理解这些问题，也展示如何解决这些问题。

基于构建电子商务应用软件的经验，我们总结了几种用于电子商务应用软件的模式。它们并不新颖，它们中的许多事实上代表了广泛用于其它类型的系统（如分布式系统）的技术。然而，电子商务赋予了它们有趣的新意和转机。

本文中的大多数模式属于体系模式类型。它们回答了有关电子商务应用的高级组织性问题。我们以应用服务器开始，它回答了问题：“电子商务应用软件部署在哪儿？”服务器端会话（SSS）结合了电子商务应用软件的状态性与目前以服务器为中心的 web 应用软件的模型。垂直截面（VS）描述了有些层在并发会话中共享时，如何组织应用软件。通过服务器群集容错（SC）和工作流（JD）显示了两种改进电子商务应用软件可用性的方法。心跳（H）描述了跟踪在群集内每个机器状态的技术。Web 检测器（WI）说明了如何把 Web 技术用于检测和管理电子商务应用软件。BCAOR 描述了访问对象的一种方法，为使用 Web 浏览器的交互类型而优化。最后预构造的义务对象（PBO）说明了如何用可重用的组件集成业务对象。

1 应用服务器

环境

你在构建一个电子商务应用软件。体系结构涉及 Web 浏览器、Web 服务器和互联网。Web 浏览器的角色是瘦客户端，把应用数据显示给用户，并从 HTML 页面收集用户数据。Web 服务器给浏览器提供 HTML 内容，并从浏览器获取用户数据。互联网通过 HTTP 协议连接客户端和服务器。

问题

把应用软件部署在哪儿？

约束

- 用户只通过 web 浏览器与应用软件交互。
- Web 浏览器只限于提供 HTML、运行 JavaScript 和 Java 字节代码。
- 所有面向用户的应用数据通过 Web 服务器传递。
- 所有面向应用软件的用户数据通过 web 服务器传递。
- 服务器端对象共享资源。
- 部署软件是耗时和代价昂贵的。

因此

使 web 服务器成为用户和应用软件交互的访问单点（Single Point）[YW1997]。Web 服务器提供内容传递技术。与应用软件结合，它代表了一个**应用服务器**。详见图 1 中的 UML 协作图。

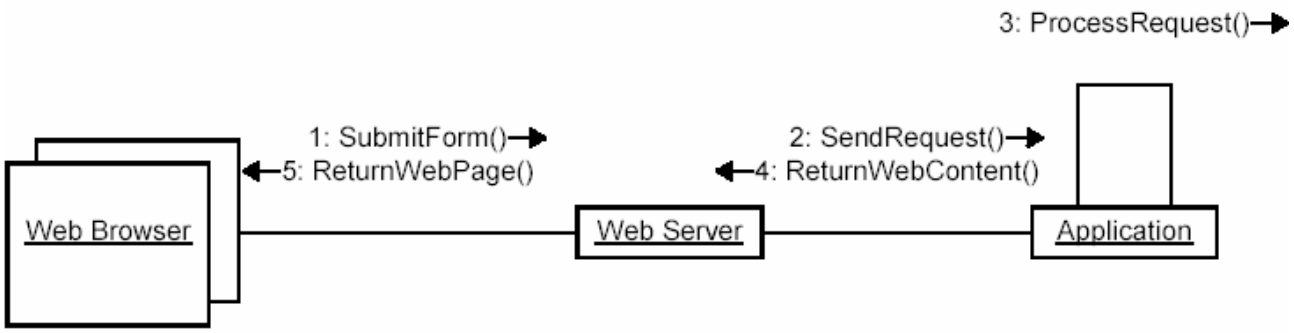


Figure 1: The APPLICATION SERVER consists of a Web server and an application.

应用软件并不直接处理表示的视觉问题，而通过表示技术（如 JSP、ASP、PHP 等），web 服务器能将这种表示技术转换成 HTML 并提供给浏览器。表示技术限于窗口小组件（widget）选择和用户交互的类型的选择，两者必须表示成 HTML 结构。

浏览器用 HTTP 协议把用户数据传递给 Web 服务器，Web 服务器解码数据并传递给应用软件。

Web 服务器要获取应用软件的把柄（handle）以便与之通信。使把柄成为一个著名的对象。先以单件（S）模式[GOF1995]开始，一旦需要一些特性如负载平衡，那么考虑使用一个命名服务（即 JNDI）。

Web 服务器控制着应用软件的生命周期。一旦启动，Web 服务器就启动应用软件。同样，当 Web 服务器关闭时，它也关闭应用软件。但生命周期能是独立的。需求或许要指定应用软件和 Web 服务器分别运行。这时，必须处理连接和断开问题。

应用服务器允许开发者在完成测试后就发行软件新版本，部署费用为零。

在应用服务器上运行电子商务应用软件有安全风险。一个恶意用户一旦访问服务器就损害安全性。而且，服务器是失败单点，可能成为瓶颈。这将影响所有使用应用软件的用户。

已知应用

1998 年至 2000 年 PLOP 会议注册系统用作 Apache Web 服务器内的一个 Servlet[M1998]。这个 Web 服务器解码那些通过 Web 表单提交的请求并传递给 Servlet。这个 Servlet 允许代表进入、浏览和修改其注册信息。它也让管理员浏览注册代表的汇总信息、删除用户和导出注册信息。

以应用推理企业对象框架（AREOF）构建的电子商务应用软件用一个协调器作为应用服务器的应用方。用 BEA 的 Weblogic 服务器时，开发者把协调器类作为在 Weblogic 的属性（Properties）配置文件中的一个性质。在启动时，Weblogic 激发了这个协调器的单件[S]实例[GOF1995]。同样，Web 在关闭时也关闭了协调器。

应用服务器是 Sun 的 J2EE 体系结构的一个核心组件。

BEA 把应用服务器定义为开发和部署业务逻辑提供支持的服务器端的任何应用软件[BEA]。

相关模式

应用服务器作为由应用软件使用的服务的管理器[S1997]。例如，Skillgames.com 有诸如持久性、信用卡处理、安全性等服务。

从客户端看，Web 服务器代表着对应用服务器的访问单点[YB1997]，也是客户端/分配器/服务器（CDS）模式[POSA2 2000]的一个特例。

本文中的这个和其它模式依赖于线程的正常管理。POSA2[POSA2 2000]书中有一些关于服务器体系实施方面的精彩模式。

结果

你现在有了一个用 Web 作为传输机制的多用户应用软件。

2 服务器端会话（SSS）

环境

Web 作为科学家之间共享文件的一种方法问世。其设计者的目标是开发优先技术，如 HTTP 传输协议、HTML 标记语言。自从那时起，人们就认识到了 Web 的潜能并开始利用之。随着 EC 应用软件的普及，Web 正在迅速转型成一个强活动型或强事务型环境。

很不幸，现有技术并不完全适用于这些需求。特别是，HTTP 协议是无状态的。

考虑在线购物网站注册的一个顾客。注册以要求姓名和合同信息的一个表单开始，用户在提供信用卡号和失效日期之后才能购物。目前的技术要求应用软件跟踪注册过程的状态，因传输机制（Web）跟踪不了。

问题

如何保存用户和 EC 应用软件交互的相关状态信息？

约束

- 你要为在线顾客保存状态。
- HTTP 是无状态的。
- 客户端设备种类繁杂。
- 若用户中断了过程，则不必保持状态信息。
- 不能让客户访问状态信息，以免威胁系统安全。
- 要尽可能最小化网络流量。
- 向用户隐藏服务器冲突。

因此

利用服务器端的一个会话对象，这个会话保存着应用软件需要的所有状态信息。

难道不能把会话对象放在客户端？

目前，EC 应用软件以服务器为中心的体系结构使得服务器端会话成为自然选择。然而，服务器端会话有几个不利因素。首先，服务器成为故障单点，这使得容错困难。其次，服务器是访问单点，黑客一旦进入，就能破坏所有用户的帐户。最后，服务器成为瓶颈。它必须能处理大量用户的并发访问。

只要 EC 应用软件在客户端做一点处理（如目前技术要求的），客户端会话负载沉重。把会话放在客户端引起各种问题。首先客户端与服务器的通信成为瓶颈。访问状态信息要通过网络。其次，这种方案向客户端暴露了服务器应用软件的细节情况。这是一个安全问题，例如 Schneier 讨论了改变在线购物车的商品价格的一个 Web 进攻[S2000]。最后，客户端不尽相同。在桌面计算机上储存几兆的会话信息是可行的，而对一个小内存系统如具有 Web 功能的 PDA 肯定不行。例如图 2 所示的 PDA 提供了 512K 内存的 web 浏览功能。若不用予指定的时间间隔，会话会超时。超时会释放由会话保存的资源。要有观察用户与服务器交互的机制，在一定时间的发呆后要触发时间超时。用户关闭窗口或转移到另一个地址时，客户端会话也应该能通知服务器。

Figure 2: Browsing the Web on a wireless PDA with 512 KBytes of SRAM.

考虑到上述限制，一个方案是把各种状态信息保存到服务器和客户端。大量的会话状态驻留在服务器以减轻网络负载。客户端储存额外信息有利于容错（如用于服务器端会话的 ID）和改进安全性能（如密钥）。例如，BEA 的 Weblogic 服务器依靠在主、从服务器之间复制会话来达到容错目的。客户端的 Cookie 存储了指向应用服务器的两份拷贝的位置的主键。若主服务器宕了，客户端有需要查找的所有信息，可以切换到从服务器。

已知应用

基于 Web 的应用软件利用一个会话对象在 Web 服务器保存状态。许多 Web 服务器支持保存服务器会话。有些机制强调了识别会话、包括（在客户端的）Cookie 和统一资源定位器（URL）重写。其它机制则强调把会话复制到其它服务器以改善可用性。

应用推理企业对象框架（BCAOR）提供了服务器端的会话上下文（Sessioncontext）对象。这个框架让开发人员构建有状态的、基于服务器的 EC 应用软件，而无论基础技术是否提供会话功能。

相关模式

服务器端会话象有主键的字典那样发挥作用。开发者在命名空隙中存储与会话相关的信息。在效果上，这种会话利用了 Property 模式[FY1998]。

Martin Fowler 在其信息系统体系结构模式的文集中讨论了存储会话信息的几种选择[F]。Martin 的状态会话（TS）模式把会话信息存储在服务器上。

3 垂直截面（VS）

环境

你有一个运行 EC 应用软件的应用服务器，客户通过 Web 浏览器在互联网上使用这个应用软件。他们并发地、独立地和以异步方式使用。你不能控制他们正做的事情和何时做。

近二十年把计算机带到了每个人的桌面。人们安装软件和运行软件的单独、相同的拷贝。而运行在应用服务器上的 EC 应用软件改变了这种平衡。你不再有单独的应用软件，而只有单独的视图。在线 3270 终端的风格再次时髦起来（这次并不用绿色 CRT）。

问题

那么，如何构造应用软件支持多并发用户，而又能保留域对象的一致性呢？

约束

- 由于内存限制，不可能在应用服务器端上给每个用户一份应用软件拷贝，特别在业务信息量大时。
- 在并发用户中共享公用的子系统是一个难题。
- 没有认真和严格的访问控制，与单个逻辑服务层交互的多用户遇到了扩展性问题。
- 应用软件和业务逻辑要了解用户和业务模型。
- 业务模型应该意识不到有多少个客户与系统交互。
- 在多线程中共享的对象应是线程安全和等幂（idempotent，即若对一个对象，重复的信息与单个信息有相同的效果，则称这个对象是等幂的）的。

因此

构建应用软件，使得每个用户看到一个垂直切面，从用户接口（顶层）到应用服务（底层）。

每个用户都要求他的对象在会话范围层内。例如，用户在登录到应用软件时，将得到唯一的客户描述对象。相反，在应用软件层的对象被共享，我们把这些对象叫做服务提供者。图 3 中的 UML 对象图示显示了共享三个服务的两个会话。例如，使用 EC 应用软件的所有客户共享由对象关系影射器提供的相同持久性机制。

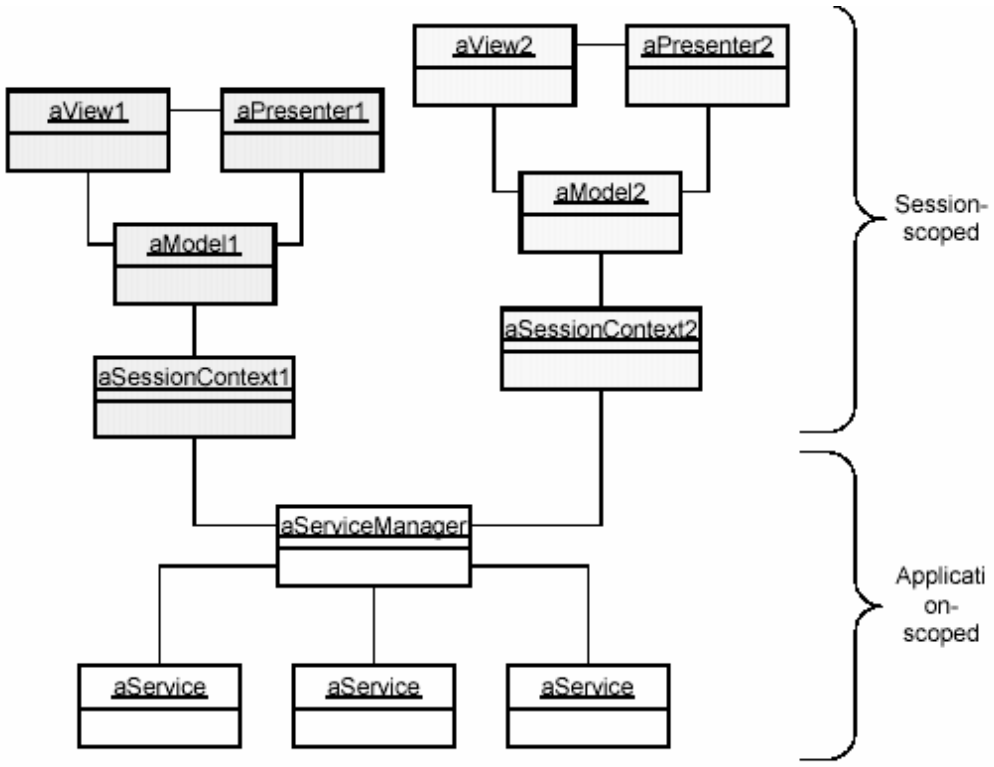


Figure 3: An application slice consists of session- and application-scoped objects. The session-scoped tiers shown here use the Model-View-Presenter pattern [MVP], but that is not a requirement.

会话范围的对象能按更传统的桌面应用软件方式设计，假定在某一个时刻只有一个用户/线程使用它们，即使它们在服务器上终止运行。你得到了类级别的应用软件/业务逻辑重用，亦即：你创建了业务对象的多个实例，每个会话都有一个。虽然它们都来自于相同的类模板。

应用软件范围的对象更难，要考虑并发性。所有入口点必须是线程安全的。这些对象不能假设连续的发送消息都来自于同一发送方。这意味着你要花更长时间设计和创建它们，你得到的报酬是对象级重用（访问相同实例的许多会话）和更有效地利用系统资源。

每个垂直切面的会话部分将需要一种访问应用部分对象的机制。利用遵循管理器（M）模式[S1997]的服务管理器。这个服务管理器控制着服务提供者的生命周期和提供对它们的访问。

已知应用

用应用推理企业对象框架（AREOF）构建的 EC 应用软件给每个用户赋予了完整应用软件的一个垂直切面。从逻辑上看，表示、会话和应用层驻留在这个切面的会话范围。同样，服务层驻留在切面的应用范围内。

Sun 公司的 J2EE 体系结构就使用这种模式。垂直切面的会话范围包括 HTTP 会话和会话 beans。应用范围包括实体 beans。

Weblogic 的 T3 服务让开发者在 Web 应用软件中共享服务。在效果上，应用软件能在垂直切面的会话范围内共享服务。

与 X 服务器对话的 X 终端共享类库和其它系统服务（也适用于 Windows 环境的 Citrix）。

相关模式

垂直切面切割从用户界面到应用服务的各层。这对应于在（POSA1996）中分层（L）模式中描述的分层化体系结构。

服务管理器表示管理器模式（S1997）的一个实例。在垂直切面会话范围内的对象以名字通过这个管理器访问服务对象，这个管理器是 Property 模式（FY1998）的一个实例。

Doug Lea 为 Java 并发编程提供了设计原理和模式的一个扩展[L1999]。

4 通过服务器群集容错

环境

EC 应用软件正运行在一个应用服务器上。当一个用户连接到这个站点时，web 服务器从数据库检索对应的业务对象并初始化一个服务器端会话。例如，一个 CustomerProfile 对象保存用户注册时提供的信息，即姓名、地址、信用卡号和失效日期等。这个 Web 服务器还运行产生动态内容的代码。几种技术如 JSP、Servlets、ASP、PHP 等支持在服务器端产生 Web 页面。

高流量 web 站点依靠在几个 web 服务器的群集中分布负载来改善可扩展性和可用性，如图 4 所示。一些产品如 Weblogic 服务器提供了对群集的支持。

一旦群集启动，运行的各种问题可能导致个别服务器宕机。例如，硬件错误如一个坏盘或 CPU 过热需要停止服务器维护。软件升级或冲突也需要宕机。你要从群集中取出服务器，完成维护/升级工作，然后再把它加入到群集中。其它几台服务器之一要承担由那个宕机的服务器所承担的处理过程。

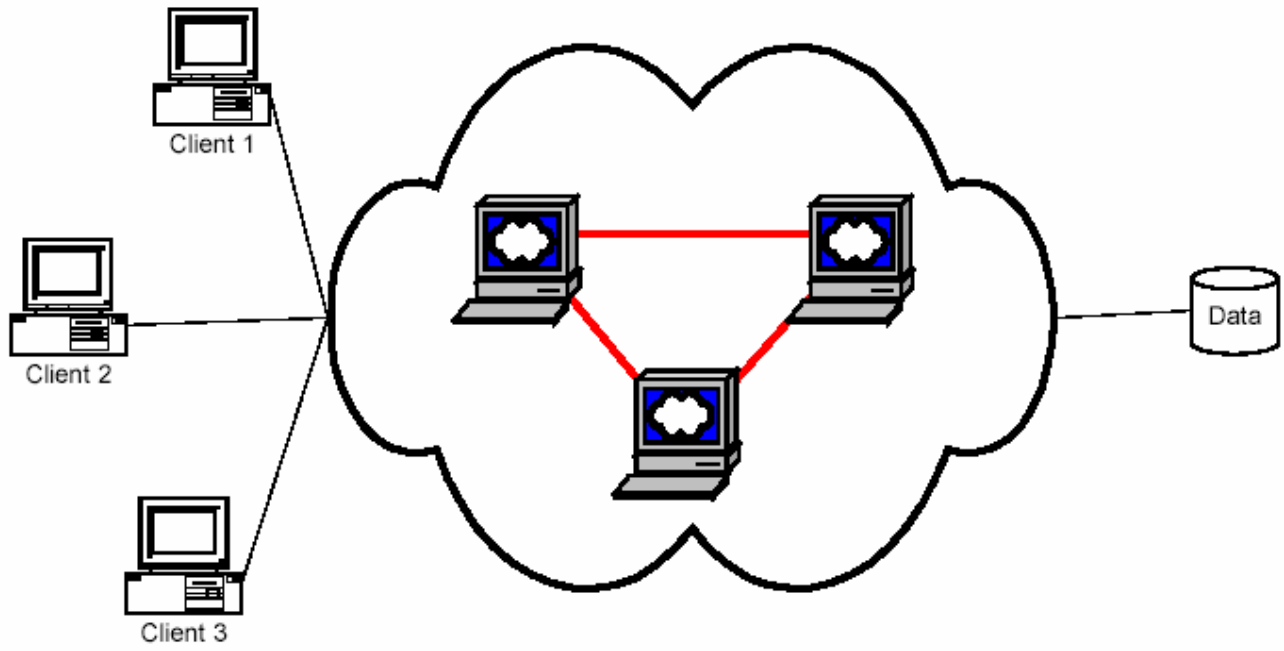


Figure 4: Clustering increases the availability of the Application Server.

问题

如何构建 EC 应用软件以支持容错？

约束

- 典型的 EC 应用软件要处理大量的并发用户。
- 应用软件的错误是电子商务世界中的大问题，你需要高可用性。
- 在应用软件状态变化时就保存这些状态，代价相当大。
- 许多 Web 服务器提供对群集的支持。
- 群集支持被群集的服务器之间的复制。
- 在群集中复制所有对象并不扩展规模。
- 不能完全隐藏服务器失败，因为当前的 Web 浏览器是逐步显示 HTML 的。

因此

构建 EC 应用软件来与群集支持兼容。这种支持应来自于一个商业产品，不需要自己构建！

在大多数情景中，用户只与主服务器交互，主服务器把对象复制给一个或多个从服务器。假若主服务器不可用，从服务器有能代替主服务器所需的所有状态信息。一个从服务器接替，应用软件继续运行，虽然在群集内的不同位置。外界察觉不到主服务器的宕机和应用移向一个从服务器。

例如，Weblogic 支持 HTTP 会话、EJB 和 RMI 对象在内存内的复制。服务器从主机复制 HTTP 会话的内容和 EJB/RMI 对象到一个从服务器。

理解商业产品如何支持群集是非常重要的。对象的复制代价在网络流量和内存消耗上急剧增大。例如，Weblogic 利用 Java Serialization，当你有大的对象图形时，状态很少变化，这不是最佳选择。要清楚你必须要复制什么，要理解能用其它资源如持久层重新构建而避免什么。

总之，这种模式提供了通过冗余屏蔽服务器故障的机制。这是分布式设计的一个关键性原理[P1995]。

已知应用

BEA 的 Weblogic 服务器支持 Web 群集和组件/对象群集。Web 群集在垂直切面的会话部分复制对象。同样，组件/对象群集复制驻留在应用部分的 EJB 和 RMI 对象。

虽然大多是无状态的，用于财务设备定价的服务器一般总是群集。因为缺乏状态，它们没什么可复制的，这简化了要考虑的问题。然而，在不影响用户的情况下，机器能随意加入或取出聚簇。

变体

地理性容错允许系统管理员选择在不同位置的主、从服务器。若存放主服务器的地点不可用（由于电力问题），从服务器不受影响能继续工作。ATG Dynamo[D]就提供了这种特征。

相关模式

许多分布式系统利用备份改善可用性。例如，可恢复的分布器（RD）模式[ID1996]提供了几种用于容错的协议。一个协议是主/备份，切换到备份机恢复工作。可恢复的分布器也能在备份系统上重新构建主服务器的状态，这耗时较长，但不需要用于维护日常更新拷贝的资源。

应用服务器利用这种模式来增加能支持并发的服务器端会话的数目、和隐藏服务器故障。

工作流（JD）模式提供了一种工作，让系统管理员在服务器群集不可用时把机器关掉。

心跳（H）模式描述了检查群集内每个服务器的可用性的机制。

5 工作流（JD）

环境

你在 Web 服务器的群集上部署 EC 应用软件。每个服务器运行不同的应用软件实例。你要按计划宕下服务器维护，必须按对用户透明的方式来做。

若用服务器状态复制，如 EJB 服务器聚簇，能去掉服务器，而同时用户察觉不到。然而，若服务器提供按应用软件不兼容的方式复制，或服务器在运行长事务，要慎重，不能简单地拔掉服务器。

问题

当应用服务器不支持群集时，如何友好地执行维护计划呢？

约束

- 在按计划要宕的机器上有服务器端会话的用户不应受影响。
- 你不能提升基础设施的聚簇能力。
- 应用软件的状态复制代价高。
- 事务能长时间运行。

因此

阻止按计划宕机的服务器接受新请求。这涉及到监听进入的请求和检查它们对应的是新工作还是存在的工作。让对应存在的工作的请求进入，而把对应新工作的请求重新指向其它服务器。

重新定向路由请求让服务器完成正运行的工作。一旦完成，你就可以安全地宕机了。

谁来重新定向路由请求？群集服务器用一个负载均衡器在聚簇成员中分配新进入的请求。因所有请求通过均衡器，重新路由就可以在此完成。这涉及到在负载均衡器上增加一个分发器，这个分发器也要能执行代码，详见图 5 和图 6。均衡器加分发器子系统要设计成多个，以免它们成为故障单点。分发器要处理流过整个站点的流量。除此外，它还要意识到运行在每个服务器上的工作。在作业数量大时，这或许不会扩展规模。

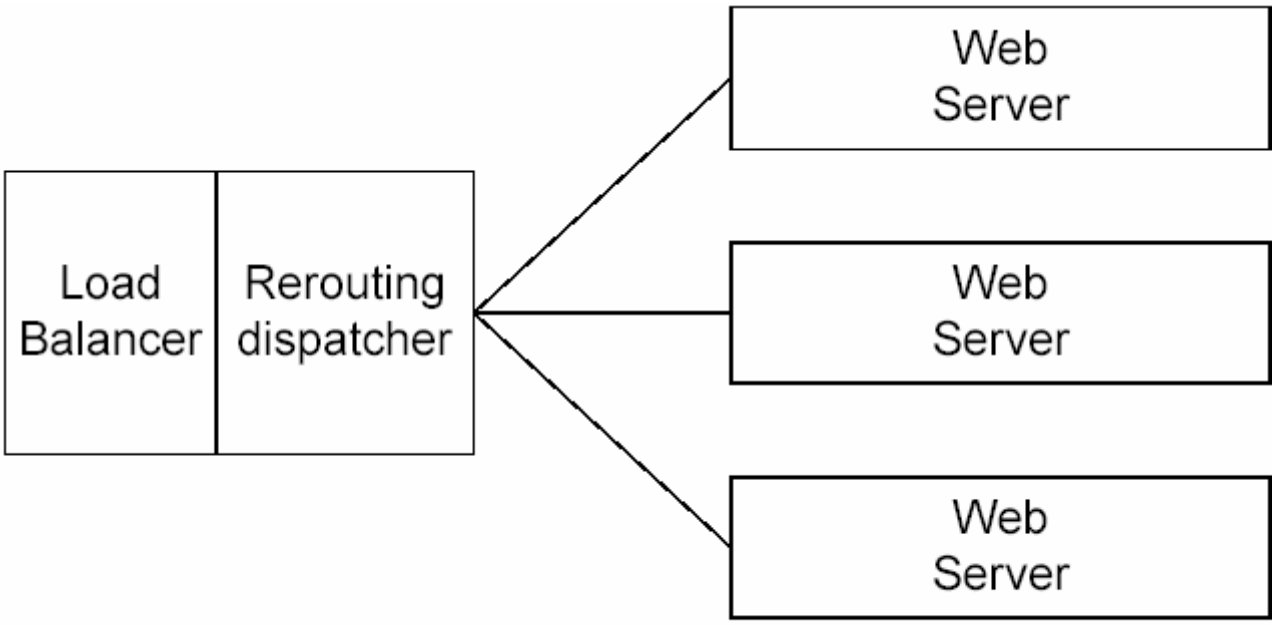


Figure 5: Job draining with a Rerouting Dispatcher

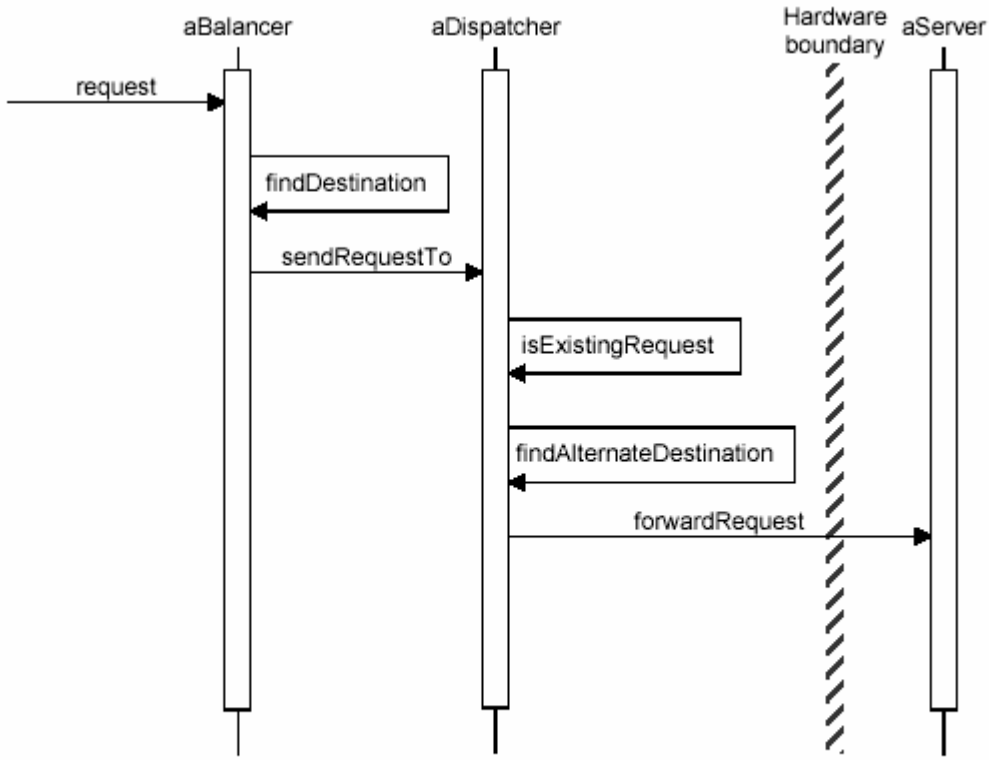


Figure 6: Rerouting Dispatcher, UML sequence diagram. The sequence shows the flow of messages when isExistingRequest() returns false, and findAlternateDestination() returns an alternate destination.

一种选择是在每个服务器上利用重新定向路由的代理机制，见图 7 和图 8。当一个服务器按计划宕机时，它的代理开始向其它服务器发送重新定向路由请求。这种方案有优势，即每个代理只跟踪局部作业。你也能用一个“哑”负载均衡器只按循环（round-robin）方式分发请求。然而，一旦服务器状态改变，重新定向路由的代理就需要通信。

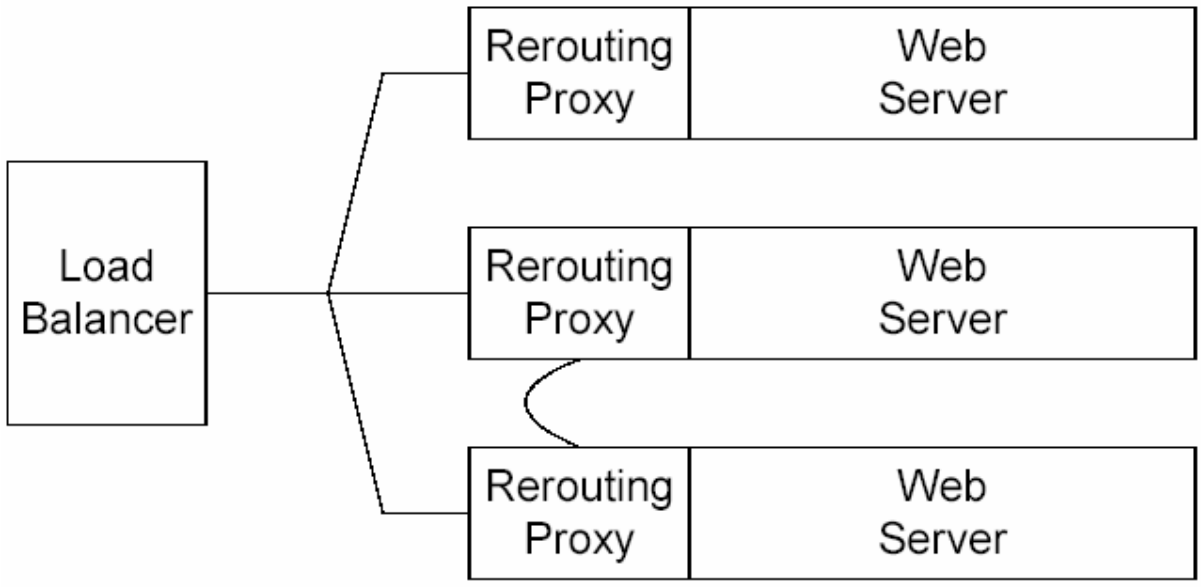


Figure 7: Job draining with Rerouting Proxies

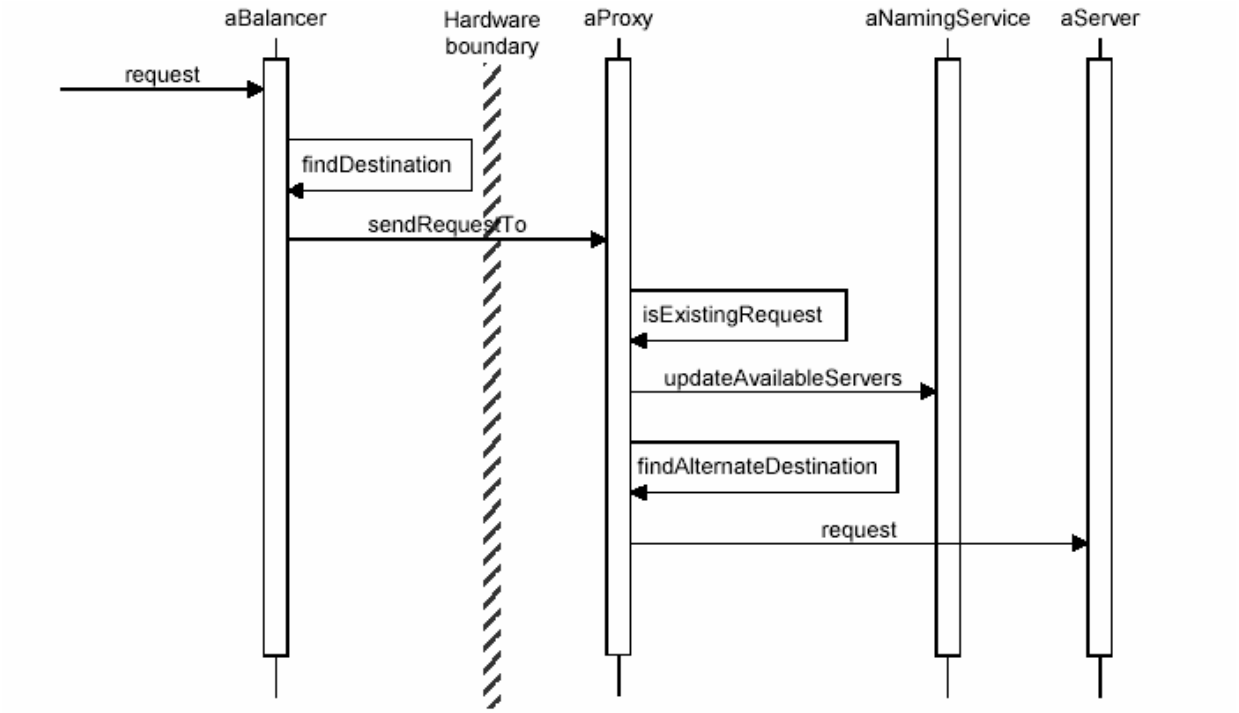


Figure 8: Rerouting Proxies, UML sequence diagram. The sequence shows the flow of messages when isExistingRequest() returns false, and findAlternateDestination() returns an alternate destination.

已知应用

skillgames.com 利用重新定向代理，把进入的请求指向可用的 web 服务器。这个代理利用 Java 消息服务（JMS）相互通信。

NCSA 负载共享设施（LSF）体系让管理员为按期维护其超型计算机流水化作业。LSF 利用作业安排器整合重新定向路由的分发器。

相关模式

当你有一个应用服务器而不能依赖于这种设施去复制服务器端会话时，作业流水化有助于取得有限的容错级别。

重新定向路由代理描绘了代理（P）模式[GOF1999，R1999]的一个实例。

结果

你需要一种方法检测应用服务器何时不可用。

6 心跳

背景

你在一个群集上部署了应用服务器，每个群集成员都运行应用软件的一个实例。

问题

当一个 Web 服务器不可用时，如何知道？

约束

- 了解服务器的可用性以便采取积极措施警示局部系统。
- 最好集中管理断开与重新连接机制。
- 等待超时影响了性能，也消耗了系统资源。
- 不断检查 Web 服务器的可用性消耗了系统和网络资源。
- 有错误发生时，要决定谁负责采取行动。

- 群集内的错误要对用户透明。

方案

使每台正在作业的服务器周期性地广播“我在作业”的信息。一个群集监视器侦听这些广播并重新设置每个服务器的计数器的超时值。监视器把那些在给定时间间隔内没有重新设置计数器的服务器视同不可用。监视器通知群集，后者采取积极措施补救不可用的服务器。

周期性广播的频率依赖于应用软件。一方面，低频率转化成在客户端可察觉的迟延，有惹恼客户的风险。另一方面，频繁广播会因控制信息而使网络负载加重，也消耗了系统的资源。

已知应用

Linux 核心支持几种形式的看门狗。若看门狗设备没有定期被写，系统会重新启动。

BEA 的 Weblogic 服务器利用 IP 组播去广播心跳信息，宣布在群集内服务器的可用性。

当联网设备不可用时，如断电，以太网利用心跳/脉冲信号告警。

Tibco，一个商业性消息中间件产品，在它的所有分布式代理之间传送心跳，以说明它们处于工作状态。

根据 Bruce Schnerier [Schneier 2000]，安全系统利用这种模式防止窃贼把它们从电话系统中断开。

经典的 Blend 和 DCOM 在分布式组件环境中利用心跳。

变体

可恢复性的分布器[ID 1996]通过调查而非广播去检查局部故障把柄（LFH）的可用性。因此，不是广播“我是活动的信息”，而是全局故障把柄（GFH）在所有局部故障把柄中循环并给它们发送信息。假若在几次联系后接收不到认可，全局故障把柄就认为局部故障把柄发生了故障。

相关模式

工作流需要检测群集中服务器状态的一种措施。心跳提供了集中式检测措施。

7 Web 检测器（WI）

环境

在规范的软件部门，开发者利用功能丰富的工具编写正确和高效的代码，并有效地调试问题。这些工具包括步调试器（大多数 IDE 有这些工具），描述器（jprobe）和对象检测器（visual Age 的 Scrapbook 等）。对于要部署成桌面可执行的两层应用软件，这些工具足矣。在三层或多层应用软件中，如一个应用服务器，还没有这种工具。也难于向每个桌面完全复制 n 层部署环境。其结果是开发者桌面运行完好的代码在部署环境中就有问题了。分布式调试器可行，但总是针对 IDE 或特定部署平台，在系统运行时，难于配制或用上。远程描述也可行，是获取计时和对象分配信息的一个好方法。它也非常有助于开发者在部署环境中检测运行的对象，检查变量的值，甚至执行信息。

问题

把应用软件实施在一个远程应用服务器上，怎样才能快速和容易地检验什么工作在进行，并在线重新配制？

约束

- 构建分布式应用软件困难，调试分布式程序也确实困难。
- 典型部署的应用软件不包含调试信息。
- 服务器并非物理上不可接近，它被用一个胖管道连到互联网，重新分配到了一个宿主服务。
- 防问限制能禁止使用远程调试器。
- 方案应该与供应商无关。
- 最好是零部署。
- 大多数检测工具是应用软件意识不到的。
- 提供暗箱访问具有大量安全漏洞。
- 需要搭架服务器，否则你不需要这样做。

方案

在系统内构建通过 Web 浏览器对检测应用软件状态的支持。可以采取几种形式：①用预定义信息询问已知的对象，②允许检测任何对象，只要能找到它（需要反射）。③允许向任何对象发送信息，允许在线改变状态（需要反射和“编译”）。

要构建这种支持，你要向应用服务器增加这种功能。这必须按受控制的方式做，涉及到高安全性、认证和审计。

已知应用

Jprobe Java 描述器让开发者查询，通过 Web 浏览器从远程收集到的信息。

几个 Cable/DSL 路由器运行在一个内部 Web 服务器上，让用户通过 Web 浏览器配制它们。图 9 显示了一个例子。

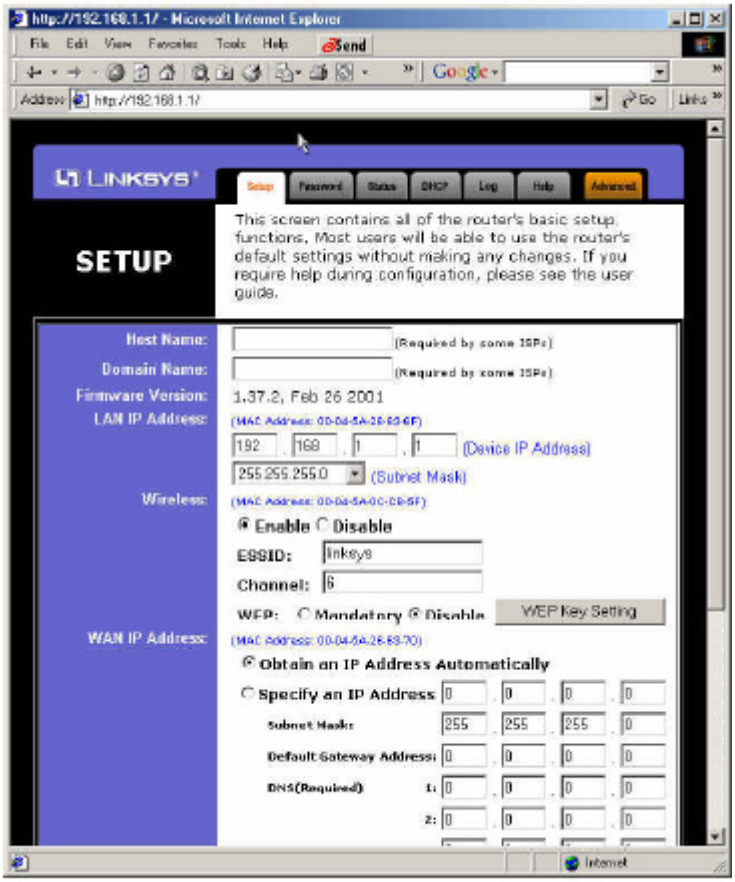


Figure 9: Web configuration panel for a Linksys cable/DSL router.

Web 服务器如 Apache 和 Weblogic 提供一个“管理控制台”，系统管理员借此配制它们。

相关模式

当要构建一个应用服务器并想能观察其运行情况时，Web 检阅器就有用了。

反射（R）模式[POSA1996]描述了以模式形式用于反射的机制。

8 业务上下文有关对象检索（BCAOR）

背景

在其生命周期内，一个服务器端会话与许多对象交互。通常，这些对象将是复杂对象图形的组成部分，从各种数据源（数据库、平面文件、远程系统等）中引进。

例如，当登录一个在线贸易服务系统时，你想首先检查你的有价证券的概况，你关注的股票的价格和重要市场新闻。这就要求系统访问浏览各种信息。其次你要查看所持有的股票的细节情况及其市价，这要求访问每个持有股的细节情况。

问题

如何以既有效又匹配用户导航的信息路径的方式把对象图形带进应用服务器？

约束

- 用户不按深度优先的方式浏览站点。
- 应用服务器要支持大量的并发用户。
- 丰富的业务对象模型的传递封闭性总是很大。
- 对象检索代价高。
- 与传统的桌面应用软件相比，基于 Web 的 EC 应用软件有不同的使用模式。

因此

利用智能代理 stub out 那些不再被需要的对象。当用户要访问一个 stubbed out 的对象时，应用服务器要求代理把它描述的对象挑选(fault in)进来。

例如，一个存储最近值的有价证券对象，不需要访问每个持有股来向用户显示价值。因此，构建用户的汇总页面时，你只需要访问有价证券对象，而 stub out 它的所有组件。

同样，顾客描述对象保存顾客地址。应用软件不用这个地址除非顾客要编辑他的描述时，当顾客登录应用服务器时，只检索顾客描述实例，而 stub out 地址。

你需要一种机制去指明在父辈的什么实例变量应该 stubbed out 而非与父辈一同被检索。GemStone/S 对象持久性存储和 OpenTalk 分布式应用框架，通过一个类/实例边描述符提供了这种机制。

已知应用

当一个顾客第一次登录到 Skillgames.com 时，只有顾客描述对象的基本部分，一些喜好从数据库中装载。其它子对象如地址、电话号码等 stubbed out。这样做可以缩短登录时间。70%的时间，用户将终止会话而不去“我的帐户”（要求子对象的站点）。30%时间，要去，其代价是只稍微增加了去“我的帐户”需要 fault in 子对象的时间。

许多有价证券，特别是公共基金，每天都有新价。这意味着昨天的总价值通常存储在有价证券对象自身。其结果是有价证券的汇总价值甚至不用检索就能显示。当需要实时价格时，每个持有股被重新计算和汇总，系统必须要 fault in 持有股。

相关模式

这种模式实际上是代理[GOF1995，R1995]的变体。

9 预构建的业务对象（PBO）

环境

若调查 EC 应用软件，它们都需要一套相似的核心对象。每个应用软件之间的区别在于你要集成哪些部分。差别在整体而非局部。

例如，你在构建一个系统，包含顾客的表示。你先创建一个叫做顾客描述的对象，包含顾客的所有状态和性质。然而，随着系统的完善，你发现也需要内部用户的表示。初一看，它们很相似，包含有姓名、地址、SSN 等。因此你开始重构。

这时产生了两个容器对象，一个是顾客描述，另一个是内部用户，还有子组件如地址、电话号码等。你在组件层次有共性，而明显差别在于它们在容器层次的组合方式。例如，顾客描述保存家庭地址，而内部用户则不。

随着系统不断完善，各种对象有额外需求，容器对象最终包含的直接状态越来越少，包含的对象则越来越多。它基本上变成了外观（Façade），把它的许多被需求的属性委托给子对象。在极端情况下，客户描述仅有一个单状态，即唯一标识符如用户 ID。

问题

如何把一些在业务域内没有单独意义的小对象组成一个有意义的大对象呢？

约束

- 如何平衡各种对象的封装与它们的接口之和，而这些接口将提升到容器的接口；换言之，在哪一点上容器的接口变得太宽，就应该开始直接用被包含的对象。
- 使容器对象太定制于特定的需求而降低了重用性和扩展性。
- 若容器中的一个组件具有你不需要的特性，如何处理这种功能？你应该屏蔽它，使它离开容器的接口，但你违犯了 Liskov 替换原则。
- 特别对于顾客对象，扩展性非常重要，因为它们面向许多 EC 应用软件，业务希望是通过现有的第三方组件库，应该满足基本功能。然而，业务几乎总是需要某些程度的定制。

方案

从一个最小的核心对象开始，设计一个提供所有附加行为和数据的丰富外观。也要用策略模式提供需要时的操作中的灵活性。

例如，我们想创建一个客户描述对象，它是电子商务框架的组成部分，因此在“箱子之外”有一套丰富的能力。我们允许客户选择在部署应用中要包括的子集和为他们增加自身特殊功能的能力，我们也要确保框架是通用的。

核心对象叫做客户核心，它有一个唯一标识符，即用户 ID 和一个姓名。然后由一个外观即基本客户描述包装，这又增加了家庭地址、工作地址、家庭电话、工作电话。我们现在有了一个可用的客户描述对象。

购买框架的人有两种方法扩展我们提供的功能。第一种是创建他们自己的外观，替换基本客户描述，直接保存具体组件（如地址和电话对象）。第二种是创建他们自己的 FACADE，保存基本客户描述。两种方法都不受继承性限制，因此提供了强大的灵活性。

已知应用

在财务系统中的交易对象通常由许多信息组成，如对方结算细节、定价日期、基本交易信息等。容器（交易）几乎总是非常薄（通常只有一个交易 ID），还有放在可重用的组件中的所有行为。

相关模式

由现有对象组成的对象称为外观（F）模式［GOF1995］，它也被认为是特殊的整体 / 部分（WP）模式［POSA1996］。你可以用享元（Flyweight）模式[GOF1995]去节省空间，在外观中共享组件。例如，若雇员对象是一个预构造的业务对象，几个雇员能共享相同的业务地址。

10 把它们组合起来

公司决定你正在构建的新系统应具有Web 功能。你也认为组合简单的 CGI 脚本、数据库访问和 HTML 满足不了需求。那么，如何做？

首先使业务逻辑与表示分离。然后选择最好的表示技术满足需求，如 JSP、Servlet、Dhtml、PHP 等，把它挂在业务逻辑上。这两个就形成应用。最后增加一个 Web 服务器，你就有了一个应用服务器。

其次，你要决定在客户与站点交互时应保存哪些会话信息？你把这种信息体封装进一个服务器端会话中。

你也应该把应用分成业务逻辑和表示层，你现在要把业务逻辑分成几部分以便它们在会话中被共享、这些部分对每个会话都是唯一的。事实上，你在构建你的垂直切面。业务上下文可知对象检索以与顾客导航 Web 页面兼容的方法带进对象。跟随预构建的业务对象，你将从现有的、已测试的业务对象组成域对象。

你可以利用由你选择的基础设施提供的支持，你也可以利用通过服务器聚簇的容错机制。也可以选择工作流，与重新路由的分配器或重新路由的代理，心跳将告诉你哪台服务器宕了。

一旦部署，Web 检测器让你使用 Web 浏览器检看应用的幕后情况。

11 致谢

感谢以下来自 Applied Reasoning 的同事和朋友，他们审阅了本文的草稿。Eduardo Fernandez 也提供了深刻的见解。

12 参考文献

[Astley+2001] Mark Astley, Daniel C. Sturman, and Gul A. Agha, *Customizable Middleware for Modular Distributed Software*, CACM May 2001, Volume 44, Number 5, pp. 99—107.

[BEA] *What is a Java Application Server?*, BEA Inc. white paper, available on the Web at http://www.bea.com/products/weblogic/server/paper_Java.shtml.

[Foote & Yoder 1998] Brian Foote and Joseph W. Yoder, *Metadata and Active Object-Model,* Fifth Conference on Patterns Languages of Programs (PLoP '98) Monticello, Illinois, August 1998. Available as Technical Report #WUCS-98-25, Dept. of Computer Science, Washington University, September 1998.

[Fowler] Martin Fowler, Information Systems Architecture (work in progress). Available on the Web at http://www.martinfowler.com/isa/.

[GoF 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, AWL 1995.

[Islam and Devarakonda 1996] Nayeem Islam and Murthy Devarakonda, *An Essential Design Pattern for Fault-Tolerant Distributed State Sharing*, CACM October 1996, Volume 39, Number 10, pp. 65—74.

[Lea 1999] Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns*, 2nd edition, AWL 1999.

[Manolescu 1998] Dragos A. Manolescu, *An Object-Oriented Framework for On-Line Registrations.* Available on the Web at http://micro-workflow.com/CAT/.

[MVP] The Model-View-Presenter pattern. Available on the Web from http://www.objectarts.com/EducationCentre/Patterns/MVP.htm.

[POSA 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal, *Pattern-Oriented Software Architecture*, Volume 1, Wiley 1996.

[POSA2 2000] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, *Pattern-Oriented Software Architecture*, Volume 2: Patterns for Concurrent and Networked Objects, Wiley 2000. Page 24 of 24

[PLoPD1] James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design,* Addison-Wesley 1995.

[PLoPD3] Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design, Volume 3*, Addison-Wesley 1997.

[PLoPD4] Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design, Volume 4*, Addison-Wesley 1999.

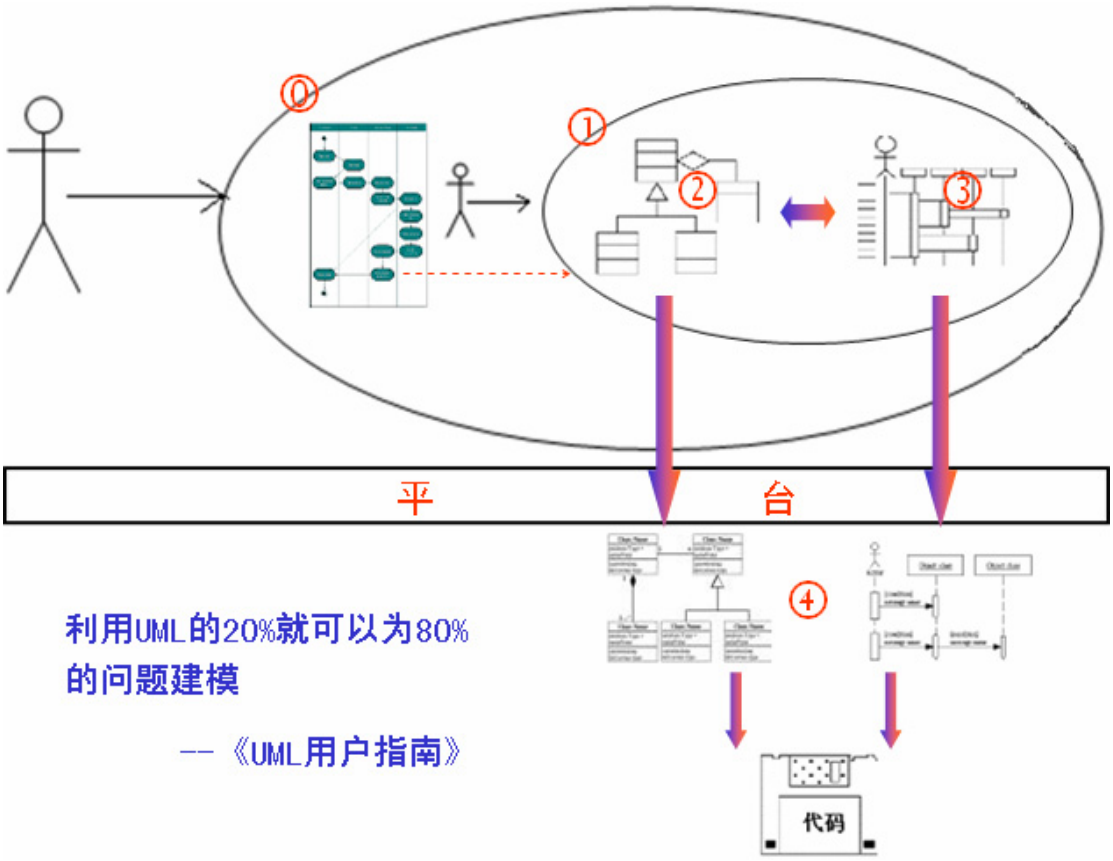
[Pradhan 1995] Dhiraj K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall, 1995.

[Rohnert 1995] Hans Rohnert, *The Proxy Design Pattern Revisited*. In [PLoPD1].

[Schneier 2000] Bruce Schneier, *Secrets & Lies—Digital Security in a Networked World*, John Wiley and Sons, 2000.

[Sommerland 1997] Peter Sommerland, *Manager*. In PLoPD3 [PLoPD3].

[Yoder & Barcalow 1997] Joseph W. Yoder and Jeffrey Barcalow, *Architectural Patterns for Enabling Application Security*. In PLoPD4 [PLoPD4].



北京公开课，2002 年 9 月 14 日至 15 日

北京中关村 1 号海龙大厦 713-714 报告厅

报名交费请联系：bjcourse@umlchina.com

重构过程中的行为保持[1]

William Opdyke 著，透明 译

查看评论

显然，重构应该保持程序的行为不发生变化。在本章中，我将讨论几个与“行为保持”相关的主题。在第1节中，我首先向读者介绍几个很容易在重构过程中被破坏的程序属性。在第2节中，我对重构的作用范围做了一个定义。在第3节中，我列出了用于描述重构前提条件的函数式，这些前提条件可以保证重构前后程序的行为保持一致。

1. 程序的属性和行为保持

在重构之后，程序在语法意义上必须是正确无误的。比如说，如果你新建一个类，而这个新的类又继承一个超类，那么后者必须是一个已经存在的类。再比如说，如果你在子类中覆盖超类中定义的一个函数，那么子类中的函数必须与超类中对应的函数保持类型一致[2]。

如果程序代码中有语法错误，编译器会找到它们。因此，要防止这类错误出现有一个办法：在重构之前先把程序代码的当前版本保存下来，然后进行重构；重构完成之后重新编译程序；如果编译器指出了语法错误，就退回到先前保存的旧版本。

但是，这种办法存在着两个主要的问题：

1. 这种办法会慢得令人难以接受。特别是对于那些比较高级的、由一系列低级重构组成的重构操作，速度会更慢。

2. 更重要的是，某些错误可能改变程序的行为方式，但却不会被编译器发现。

请看图1所示的错误重构。函数F1是Super类的一个保护成员，F2则是在Super类的子类Sub1中定义的。F2的参数列表和返回类型恰好跟F1相同。在Sub1类中，函数F3调用了从超类继承得到的F1函数。

现在，我们对Sub1类进行了重构，将函数F2改名为F1。这次改名可以得到语法上正确无误的程序（修改后的程序可以顺利地通过编译）。但是，程序的行为却可能已经发生了变化。现在，函数F3调用的是本地更名之后得到的F1函数（即原来的F2函数），而不是从超类继承来的F1函数。如果这两个F1函数的行为有所差异，那么F3函数的行为就会发生变化。但编译器却无法找到这一错误。

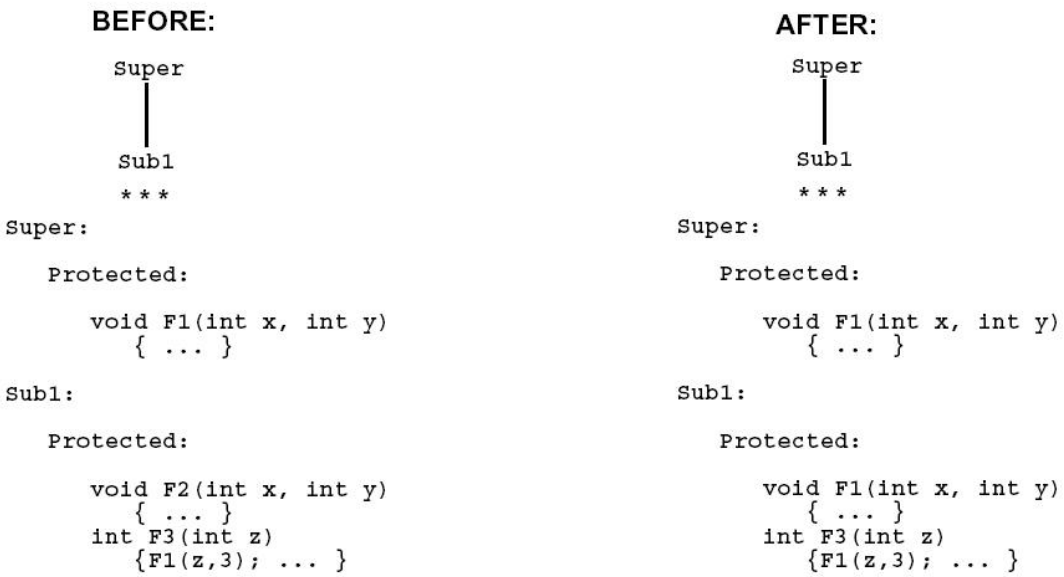


图1：给成员函数F2改名造成的错误

如果在重构的时候没有一些规则来确保程序的固有属性不被破坏，就很可能在程序中造成语法错误，甚至使重构后的程序能够正确编译、但行为发生了变化。

在研究过程中，我们找出了一些程序的属性（如下所示）。这些属性既包括语法角度的，也包括语意角度的。如果不在重构之前先把这些属性检查清楚，重构的过程就很容易破坏它们。在重构的定义中，“行为保持”的说法就是根据这些程序属性来定义的[3]。这些属性涉及继承、作用域、类型兼容性和语意相等性。如果你破坏了其中前六项属性中的任一项，重构之后的编译会指出错误；但是如果你破坏了第七项属性，则编译器可能不会发现错误。下面就是所有的七项属性：

1.**唯一的超类。**重构之后的类最多只能有一个直接超类，并且其超类不能是该类的子类。这一属性将我们的研究锁定在单继承体系中，而不必操心循环继承图的问题。

2.**唯一的类名称。**在重构之后，每个类都必须有一个唯一的名称。在我们的研究中，不考虑嵌套类的存在，也就是说，每个类的作用域都是整个程序。

3.**唯一的成员名称。**在重构之后，任一个类中所有成员变量和成员函数都必须有唯一的名称。但是，不同的类可以有同名的成员，子类中的成员也可以覆盖（override）超类中的成员。

4.**被继承的成员变量不能被重新定义。**在子类中不能重新定义继承自超类的成员变量。

5.**成员函数重定义时保持函数签名一致性。**在重构之后，如果超类中定义的成员函数被子类重新定义了，那么这两个函数的所有性质（除了函数体之外）都必须保持一致。用C++术语来表示：我们认为所有的函数都是虚拟（virtual）的，因此可以在子类中覆盖。在C++中，当且仅当虚函数的不同版本的返回值类型和参数类型完全匹配时，我们认为这两个版本是一致的。[4, 5]

因此，你可以在子类中覆盖超类中定义的函数，只要保证函数签名一致就行了。在重构中，这种“覆盖被继承函数”的能力是很重要的。例如，在定义抽象类的时候，你可能会在超类中定义一个空的函数，然后在子类中覆盖这个函数，给它定义实在的函数体。

6.**赋值时的类型安全。**在重构之后，如果要将一个表达式赋值给一个变量，那么该表达式的类型必须是该变量所定义的类型，或者（如果被赋值的是一个指针变量的话）是该变量的子类型[6]。对于赋值语句和函数调用，此项属性都同样有效。

7.**引用和运算的语意等价。**下文将专门讨论这个主题。

1.1 引用和运算的语意等价

正如前文所提到的，对于重构来说，“行为保持”的含义绝不仅仅是说“能够得到合法的程序”。重构前后的程序必须在引用和运算上都保持语意角度的等价性。我们所说的“语意等价”定义如下：以main函数作为程序对外的接口；假设以同样的一组输入调用main函数两次（重构前后各一次），得到的输出值必须是完全相同的。

根据这个定义，我们可以在程序中各处进行修改，只要保持输入/输出之间的映射关系不变，修改前后的程序就是语意等价的，我们的修改动作就是语意保持的。你可以假想在程序中画一个圈，将所有被重构影响到的部分都围在圈中。你对圈中的部分作了修改，但是从这个圈的外部看来，一切都保持不变。对于某些重构，这个圈的范围几乎达到了整个程序。举个例子：如果某个变量在程序的各处都被引用到，那么“给这个变量改名”的重构就会影响到程序的许多部分。而对于某些重构，这个圈只覆盖了很小的区域。例如你把某个函数调用替换成被调用函数的函数体时，被影响的只有调用函数中的一部分。然而，无论被影响的范围是大是小，最重要的是：从外部调用该程序的结果（以及副作用）、从外部引用该程序中某个对象的效果不能有变化——整个圈从外面看来必须保持不变。

以下几种修改是不会影响语意等价性的：

- 简化表达式，删除死代码。如果条件语句所判断的是不会发生变化的条件（例如类的不变量），那么就可以将该条件语句化简。变量、函数和类，如果没有被任何程序引用，就可以删掉。
- 类似地，没有被引用的变量、函数和类也可以随时被加入到程序中。
- 变量的类型可以在重构中被改变，只要所有引用该变量的运算都等价地接受该变量的新类型即可。
- 对某个类中变量和函数的引用可以替换成对另一个类中等价的变量或函数的引用。这就意味着：本地定义的成员可以替换为继承得到的成员（反之亦然），只要两者的声明等价即可。另外，一个合成对象中某成员的引用也可以替换为该对象中某组件中等价成员的引用（反之亦然）。

有一种极具限制性的办法可以确保重构前后的程序保持语意等价性：要求重构之后所有变量和函数的引用都与重构之前完全相同。的确有几个重构遵循了这一要求，例如“添加新变量”或者“修改现有变量名称”之类。但是，对于大多数有用的重构操作，这一要求都是不切实际的。

实际上，即使将限制条件放宽一些，重构之后程序的行为仍然可以与重构之前保持一致。在继承体系中将类成员移上移下、在合成类与组件类之间移动成员都属于此类例子。当成员变量从一个类移到另一个类的时候，如果旧变量与新变量有严格的一对一关联，并且满足下列条件，那么行为仍然是可以得到保持的：

- 新的变量与旧的变量有着同样的类型和生存周期；
- 所有对旧变量的引用都被修改为引用新的变量；
- 或者新变量没有被任何代码引用。

相比继承体系中的重构，涉及“合成类和组件类”的重构中更难检查这些前提条件。在后面的章节中，我将详细讲解这个问题。

某些重构会改变对象的大小或者变量在对象内部的相对物理位置。这些重构只有对于设计良好的C++程序才是行为保持的。为说明这个问题，请考虑C++语言中类的物理布局：从超类继承得到的变量放在本地定义的变量之前。只有当程序的行为不依赖于变量的物理顺序时，“将变量从子类移到超类”的重构才是行为保持的；如果程序中使用成员变量的地址做了计算，那么当变量被移动之后，程序的行为就可能发生变化。类似地，考虑“将变量从合成类移到组件类”的重构——两个类的大小都被改变了。如果程序需要检查对象的物理大小，那么移动变量之后，程序的行为就可能发生变化。

在本篇论文中定义的重构不能应用于那些依赖对象物理布局和大小的程序。还好，绝大多数面向对象程序并不依赖于这些东西。面向对象语言最大的魅力之一就在于：它们提供了一个抽象层，将用户和对象的底层表现形式隔离开来。在面向对象的世界里，“依赖于底层表现形式”的程序通常被认为是不好的。

1.2 总结：程序属性和行为保持

总而言之，重构必须保持程序的行为不被改变。这就意味着：（1）重构必须得到合法的程序；（2）重构后的程序的行为方式与重构之前等价。在本篇论文中，“行为保持”是根据上面介绍的七种程序属性来定义的。这些属性是在重构中最容易破坏的。

本论文的第5章将介绍一些比较低级的重构。在这些低级重构中，“行为保持”是根据前面介绍的七种程序属性来定义的。第6章到第8章将介绍一些比较高级的重构。在这些高级重构中，“行为保持”的定义则是：它们是由低级重构组合而成的，因此它们也是行为保持的。

2 重构的范围

我把重构可以作用的目标分门别类。不同类型的重构目标以及它们各自的属性将在下面分别定义。

每个变量和函数都有一个与之相关的属性：作用域。在C++中有四种不同的变量：全局变量、类成员变量、函数参数变量和局部变量。在我们的研究中只考虑两种C++函数：唯一一个全局函数main()，以及类成员函数。

在添加新变量时，如果在新变量的作用域内有代码引用了与新变量同名的、作用域包含新变量作用域的变量，我们就说“发生了名字冲突（name collision）”。为了检查名字冲突，我们对变量的作用域做如下规定：全局变量的作用域是整个程序；类成员变量的作用域（1）如果变量的访问控制模式为private，则为包容该变量的类；（2）否则为包容该变量的类及其子类；[7]函数参数变量的作用域是包容该变量的函数；类似地，局部变量的作用域是包容该变量的代码块。

函数则分为两种。全局函数只能有一个，即main()函数[8]；此外每个类都可以有一组成员函数。超类中定义的函数可以在子类中覆盖，但要求子类中不引用超类被覆盖的方法，或者新函数的功能与被覆盖函数等价。

程序的属性如下所示：

- program:
  - (set of class) classes
  - function globalFunctionMain[9]
  - (set of variable) globalVariables
  - (set of enumeratedType) globalEnumeratedTypes
- class:
  - class superclass
  - string name
  - (set of function) locallyDefinedMemberFunctions
  - (set of variable) locallyDefinedMemberVariables
  - (set of variable) SetOfComponents
  - predicate classInvariant

- enumeratedType:
  - string name
  - (set of string) enumeratedValues
- function:
  - (class + program) owner
  - string name
  - type returnType
  - (list of variable) arguments
  - statement body
  - (nil + private + protected + public) accessControlMode
- variable:
  - (statement + function + class + program) owner
  - string name
  - type type
  - (set of constant) initializationArguments
  - (nil + private + protected + public) accessControlMode
- type: program.classes + PrimitiveTypes.
  - string name
- statement:
  - (statement + function) owner
  - (list of (statement + expression)) components

- expression: assignment + variableRef + functionCall + dynamicObjectExpression
- assignment:
  - (expression + statement) owner
  - variable destination
  - expression source
  - type type
- dynamicObjectExpression:
  - class instantiatedClass
  - (list of expression) parameters
- functionCall:
  - (expression + statement) owner
  - function calledFunction
  - (list of expression) parameters
  - type type
  - expression prefix.[10]
- variableRef:
  - (expression + statement) owner
  - variable refdVariable
  - expression prefix.[11]
- predicate: 用于定义类的不变量[12]。

在重构的定义中，可以使用“点”符号来引用局部属性。例如，成员函数F的返回值可以用“F.returnType”来表示。

3 描述前提条件的函数

对于大多数重构来说，只有在满足某些前提条件的情况下，它们才是行为保持的。例如，只有在包含某变量的赋值语句保持类型安全的前提下，你才能修改该变量的类型。

下面，我们将用一组函数来描述重构的前提条件。在这里，有一小部分原生函数，而更多的是从原生函数中衍生得到的派生函数。对于返回类型不是布尔类型的函数，在函数名之前列出返回类型。

在每个前提条件中，函数名和参数通常是描述性的。本节中给出的定义完整而详尽，很可能已经超出了部分读者的兴趣范围。如果读者只是想了解重构的应用，可以跳过本节内容。

3.1 布尔型原生函数

1. compilesP (function F, (class + program) Scope) ≡

F在Scope范围内可以通过编译.[13]

2. convertibleToFunctionP (statement S) ≡

S可以被转化为一个合法的函数.[14]

3. qualifiesAsComponentP (variable V) ≡

V是组成其包容类的一个成员变量.

4. noSideEffectsP (expression E) ≡

E没有任何副作用.

5. satisfiesClassInvariantP (expression E, class C) ≡

E所创建的实例能满足C的类不变量.

6. semanticallyEquivalentP ((function or statement) F1, (function or statement) F2) ≡

F1与F2是语意等价的.

7. subtypeP(type T1, type T2) ≡

T1是T2的子类型.

8. unrefdOnInstancesP(function F, class C) ≡

函数F不被类C的实例所引用.

3.2 其他原生函数

1. firstConditionImpliedByInvariant (statement S (a conditional), predicate P)

return S中第一个被P所暗指的条件.

3.3 布尔型派生函数

1. inheritedMemberFunctionNamedP(class C, string S) ≡

∃ member ∈ inheritedMembers(C) ∧

member.name = S.

2. matchingAttributesP(variable V1, variable V2) ≡

(V1.name = V2.name) ∧

(V1.type = V2.type) ∧

(V1.initializationArguments = V2.initializationArguments) ∧

(V1.accessControlMode = V2.accessControlMode)

3. matchingAttributesP(function F1, function F2) ≡

matchingSignatureP(F1, F2) ∧

(F1.body = F2.body).

4. matchingSignatureP(function F1, function F2) ≡

(F1.name = F2.name) ∧

(F1.returnType = F2.returnType) ∧

(F1.accessControlMode = F2.accessControlMode) ∧

length of F1.arguments = length of F2.arguments ∧

for i = 1 to (length of F1.arguments),

F1.arguments[i].type = F2.arguments[i].type.

5. memberFunctionNamedP(class C, string S) ≡

∃ F ∈

(C.locallyDefinedMemberFunctions ∪ inheritedMemberFunctions(C)) ∧ F.name= N.

6. redundantIfAddedP(function F, class C) ≡

∃ F2 ∈ (C.superclass).locallyDefinedMemberFunctions ∪

inheritedMemberFunctions(C.superclass)) ∧

F.name = F2.name ∧

matchingAttributesP(F, F2).

7. varNameCollisionP(string S, (class + function + statement) Scope) ≡

∃ collidingVar ∈ Program ∧

collidingVar.name = S ∧

Scope ⊂ scopeOf(collidingVar).

3.4 其他派生函数

1. (set of function) allFunctions(program P) ≡

P.globalFunctionMain∪(∪$_{class∈P.classes}$class.locallyDefinedMemberFunctions)

2. (set of function) allVariables(program P) ≡

return P中声明的所有变量.

3. (set of functionCall) callsTo(function F1) ≡

$\cup_{F2 \in allFunctions(Program)}$ {FC | FC ∈ F2.statement ∧ FC.calledFunction = F1}

4. (set of class) classesInternallyReferencing((function + variable) X) ≡

{containingClass(ref) | (ref ∈ referencesTo(X)) ∧ (ref.prefix = nil)}

（即：返回所有可以在其内部引用X的类）.

5. (set of class) classesPubliclyReferencing((function + variable) X) ≡

{containingClass(ref) | (ref ∈ referencesTo(X)) ∧ (ref:prefix ≠ nil)}

（即：返回所有可以通过包容X的类提供的公开接口引用X的类）.

6. (set of class) classesReferencing(class X) ≡

{containingClass(ref) | ref ∈ referencesTo(X)}

（即：返回所有包含类X的引用的类）.

7. (set of class) classesReferencing((function + variable) X) ≡

classesInternallyReferencing(X) ∪ classesPublicallyReferencing(X).

（即：返回所有包含X引用的类）.

8. function collidingFunction (class C, function F) ≡

return 添加到C中后会与F发生名字冲突的函数.

9. (nil + class) containingClass(

(variable + function + (set of statement) + expression) containedItem) ≡

```
    if ((containingItem.owner ∈ Program.classes) ∨
        (containingItem.owner = nil)), /* 全局函数 */
            containingItem.owner,
    else containingClass(containedItem.owner).
```

10. (nil + function) containingFunction(

((set of statement) + expression) containedItem) ≡

if ((containedItem.owner ∈ allFunctions(Program)),

containingItem.owner,

else containingFunction(containedItem.owner).

11. (set of class) directSubclassesOf(class C) ≡

{subClass ∈ P.classes | subClass.superclass = C}

12. directSuperclassOf(class C) ≡ C.superclass

13. (set of variable) expressionsAssignedTo(variable V) ≡

(∀ assignment expression ∈ scopeOf(V),

where destination = V, return source) [

(∀ function call ∈ scopeOf(V),

if V是该函数的参数变量,

return 相应的被作为参数传递的值).

14. (set of expression) expressionsAssignedToArgument(variable argVar)

∀ functionCall ∈ callsTo(argVar.owner),

return 与此参数处于对应位置的表达式.

15. (set of function) functionsCalledBy(function F) ≡

所有被F.statement调用过的函数.

16. (set of function) functionsThatOverride (function F) ≡

{F2 ∈ allFunctions(P) | F2.owner ∈ subclassesOf(F.owner) ∧ F.name = F2.name}

17. (set of (variable + function)) inheritedMembers(class class1) ≡

inheritedMemberFunctions(class1) ∪ inheritedMemberVariables(class1).

18. (set of function) inheritedMemberFunctions(class C) ≡

if C.superclass=nil, return nil

else return F ∈

(inheritedMemberFunctions(C.superclass) ∪

C.superclass.locallyDefinedMemberFunctions),

where (F.accessControlMode ≠ private) ∧ ~localMemberFunctionNamed(F,C).

19. (set of variable) inheritedMemberVariables(class class1) ≡

if C.superclass=nil, return nil

else return V ∈

(inheritedMemberVariables(C.superclass) ∪

C.superclass.locallyDefinedMemberVariables),

where (V.accessControlMode ≠ private) ∧ ~localMemberVariableNamed(V,C).

20. (set of (function + variable)) locallyDefinedMembersOf(class class1) ≡

(class1.locallyDefinedMemberFunctions ∪

class1.locallyDefinedMemberVariables)

21. (set of variable) localVariablesIn(function F) ≡

F.statement中所有的局部变量.

22. (function + nil) memberFunctionNamed(class C, string S) ≡

F, where F ∈ (inheritedMemberFunctions(C) ∪ C.locallyDefinedMemberFunctions)

∧ F.name = S.

23. (function + nil) memberVariableNamed(class C, string S) ≡

V, where V ∈ (inheritedMemberVariables(C) ∪ C.locallyDefinedMemberVariables)

∧ V.name = S.

24. (set of(variable + function)) membersOf(class C) ≡

inheritedMemberFunctions(C) ∪ inheritedMemberVariables(C) ∪

C.locallyDefinedMemberFunctions ∪ C.locallyDefinedMemberVariables.

25. (set of (expression + variable)) referencesTo(class C) ≡

instancesOf(C) ∪

{V | (V ∈ allVariables(Program)) ∧ (V.class= C)}

26. (set of functionCall) referencesTo(function F) ≡

callsTo(F)

27. (set of variableRef) referencesTo(variable V) ≡

$\cup_{V2 \in allVariables(Program)}$ {VR | VR ∈ F2.statement ∧ VR.refdVariable = V }

28. (statement + function + class + program) scopeOf((variable + function) item1) ≡

if item1.owner ∉ Program.classes, /* 全局、局部或者参数变量 */

item1.owner,

else if item1.accessControlMode = private,

item1.owner,

else (item1.owner ∪ subclassesOf(item1.owner)).

29. (set of variable) setOfComponentVariables(class componentClass) ≡

$\cup_{class \in Program.classes}\{V \mid V \in class.SetOfComponents \wedge V.type = C\}$

30. (set of class) subclassesOf(class C) ≡

directSubclassesOf(C)的传递闭包.

31. (set of class) superclassesOf(class C) ≡

directSuperclassOf(C)的传递闭包.

32. (set of variable) variablesAssigned(function F) ≡

(∀ assignment expression ∈ scopeOf(F),

where source = a call to F, return destination) ∪

(∀ function call ∈ scopeOf(V),

if 对F的调用被作为参数传递,

return 相应的函数参数变量).

33. (set of variable) variablesAssigned(variable V) ≡

(∀ assignment expression ∈ scopeOf(V),

where source = V, return destination) ∪

(∀ function call ∈ scopeOf(V),

if V被作为参数传递,

return 相应的函数参数变量).

34. (set of variable) variablesAssigned(dynamicObjectExpression E) ≡

(∀ assignment expression ∈ Program,

where source = E, return destination) ∪

(∀ function call ∈ Program,

if E被作为参数传递,

return 相应的函数参数变量).

35. (set of variable) variablesReferencedBy(function F) ≡

{VR.refdVar | (VR ∈ F)}

4 总结

显然，重构应该保持程序的行为不发生改变。本文讨论了容易被重构过程破坏的程序属性。此外，我还定义了重构的作用范围，以及用于定义重构前提条件的函数。

译注

1 本文系 William Opdyke 的博士论文 REFACTORING OBJECT-ORIENTED FRAMEWORKS 中的第 4 章。
2 对于覆盖函数的“类型一致”，不同的语言、乃至同一语言的不同时期都有着不同的要求。严格的类型一致要求覆盖函数与被覆盖函数的签名（signature）完全相同；宽松的类型一致则只要求两者的名称（name）和参数序列相同即可，对返回值类型不作要求。
3 这与Jay Banerjee和Won Kim在Semantics and implementation of schema evolution in objectoriented databases一文（发表于1987年ACM SIGMOD大会）中所做的属性分析很相似。不过，他们关心的是重组相关的数据，而我的研究则同时关注函数和数据的结构变化。从语意角度来说，C++是一门非常复杂的语言，它支持机器级别的操作（例如指针算术）。这样的复杂度使得我们很难比较精确地给“C++程序中的行为保持”下一个定义。
4 在其他的面向对象语言（例如 Eiffel）中，“函数签名的一致性”有着不同的含义。
5 译注：此处对于 C++函数覆盖规则的引用出自 The Annotated C++ Reference Manual。按照新的 C++标准，虚函数的不同版本只需保持参数类型完全相同即可，不必保持返回值类型完全相同。但是，在研究重构的时候，我们要求：覆盖函数与被覆盖函数之间必须保持严格的签名一致性，即两者的返回值类型也必须一致。
6 在 C++中，子类型化（subtyping）是通过子类化（subclassing）来实现的。

7 如果变量的访问控制模式为public，那么程序中其他任何地方都可以引用这个变量。但是，除了包容该变量的类及其子类之外，其他地方要引用该变量必须在变量名之前加上包容该变量的对象名。

8 之所以严格要求只能有唯一的全局函数，是为了回避重构中的某些复杂情况。只需对重构规则稍加扩展，就可以支持多个全局函数。

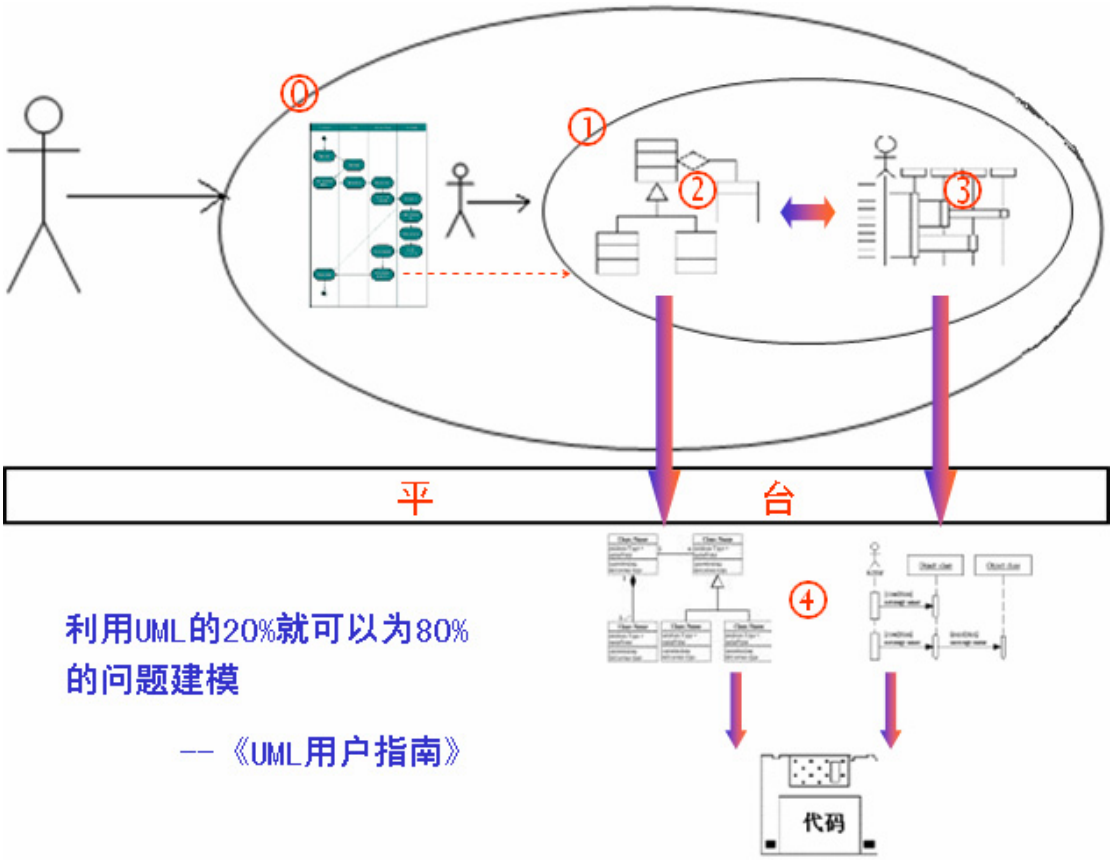
9 C++允许程序中有多个全局函数。但是，在这里我们要求每个程序只能有一个全局函数即main函数，并且该函数不可改名。

10 如果calledFunction是通过对象的公开接口被调用的，本属性的内容即为用于识别该对象的表达式；否则本属性的内容为nil。

11 如果refdVariable是通过对象的公开接口被引用的，本属性的内容即为用于识别该对象的表达式；否则本属性的内容为nil。

12 译注：“不变量”的概念在本论文第 6 章中有介绍。
13 在添加一个新函数或者给函数添加函数体时，会检查这一点。

14 S不能定义在其之外被引用的局部变量。S不能有某个分支运行到当前函数的其他部分中，当前函数的其他分支也不能运行进入S中间（如果S是复合语句的话）。对于S可见的函数和变量应该对新函数也可见。



北京公开课，2002 年 9 月 14 日至 15 日

北京中关村 1 号海龙大厦 713-714 报告厅

报名交费请联系：bjcourse@umlchina.com