X-Programmer 总第7期
软件工程的播种机
www.umlchina.com
非程序员 2001 11

目录

本期审稿：Think

点击此处查看与本期相关的参考资料

联系方法

投稿：editor@umlchina.com
广告：adv@umlchina.com
反馈：think@umlchina.com

播种机：http://www.umlchina.com

征稿

关于需求，设计，构造，测试，维护，配置管理，管理，过程，工具，质量... 原创或翻译均可。

请点击查看详情>>

XP On A Large Project
Strategies for Surviving CMM

请点击查看详情>>

UMLChina 两岁了

1999 年 11 月 25 日，umlchina.com 投入使用，主要以资料下载为主。
2000 年 3 月，在中文热讯开设 BBS，第一个发言者：诗风。
2000 年 6 月，开通自己的论坛，当时活跃的有平凡人、茶博士、3nt、Dr.OO、Peter Wang...等等
2000 年 11 月，论坛搬至 smiling.com.cn，成为讨论组。
2001 年 1 月，第一期嘉宾交流举办，嘉宾方春旭。从此成为常规活动。
2001 年 2 月，第二期嘉宾交流，嘉宾钱五哥，聊天室里的人数最高达到 58 人。
2001 年 5 月，《非程序员》第一期发行
2001 年 6 月，高焕堂参加第六期嘉宾交流，随后几期，Alistair Cockburn、John Vlissides 等名家相继来到。
2001 年 8 月 10 日，讨论组成员达到 10,000 人
2001 年 9 月，和出版社合作，由 UMlChina 翻译组翻译的第一批书籍开始。
2001 年 10 月，专家答疑板开通，Think 主持“现代软件工程”沙龙。
2001 年 11 月，《非程序员》已发行到第七期，讨论组人数将近 15,000 人。

Ivar Jacobson & Think

讨论请点击

Alan Cooper 访谈

亚玲 译自 uidesign.net

"要创造一个更好的产品，最重要的事情是在早期引入交互设计。"

第一部分：定义和介绍交互设计

2000年的元旦左右，我正在思考如何最好地超越uidesign.net中的书评。与其分析作者所著，还不如直接去问作者--访谈。那么，"应该首先对谁进行访谈呢？" 新的站点商标和口号，"交互设计师的网上杂志"为我提供了答案--非Alan Cooper莫属。作为两本优秀的书籍的作者，《关于脸》("About Face")和去年的，《精神病人管理精神病院》("The Inmates are Running the Asylum")（国内中文译本译作《软件创新之路——冲破高技术营造的牢笼》），他是交互设计之父，甚至可能是交互设计最伟大的倡导者。使我喜出望外的是他答应约一个月后我对他进行访谈。终于我和他在星期五的2月11日进行了长达80分钟的电话交谈。

因为不可能访谈Alan Cooper的交互设计(Cooper Interaction Design)的方方面面，所有的问题必须预先提交以获许可。令我惊讶的是我们从一开始就偏离了预先安排的问题并且从来没有回到这些问题。这是非常精彩的80分钟的谈话，这些谈话不仅有助于明确uidesign.net探讨的内容，还有助于对交互设计涉及到的领域有一个整体框架印象。在第一部分，Alan花时间准确解释了为什么应该说是交互设计而不是界面设计，

以及交互设计在整个以用户为中心的设计领域中的地位。

在访谈之前我还认为我有一些机会和Alan谈论自己的观点。我本认为我对交互设计有一点认识并且可以问一些聪明的问题。我们有一个好的访谈计划，并且我们有1个小时的时间。但是，本来预期的一个访谈立即变成了对大师的拜谒。当Cooper开始撕掉整个技术产业的规则手册并且抛弃它的时候，很难进入预备的访谈了。他谈到了为什么交互设计和整个系统结构有关，并且还提到了关系数据库中的漏洞；文件系统的漏洞；为什么交互设计不仅仅是界面设计；为什么他也不喜欢实用性。

我们从介绍uidesign.net开始，并讨论到了读者和信息…。

AC：AlanCooper，DJA：uidesign.net

承认“这里存在问题”的事实

AC：Uidesign.net的读者可能是可以称作"意识到光凭程序不能解决问题的程序员"。这种认识是解决问题的必要的第一步。有些程序员认为解决问题的方案是一种他们还学不会的编程技巧。这就是一个问题。它聚焦于单独的技术本身，而单独的技术只能把我们带入混乱。仅仅聚焦于技术并不能使我们摆脱困境。这就像是将某个人看作一个和他人毫无关联的个体，治愈这个人的第一步是他们必须承认自己的看法有问题。

DJA：可以毫不夸张地说，我喜欢你书中着重写一个被否认的产业。很多写出来的东西使我想到，"是的，这些人仍然被否认。"

AC：是的。在网络世界里，这是普遍性的。在PC的世界这里有很多机会。你可能被否认，但仍然有经济上的成功。在网络的世界里，因为所有扭曲的评价，你可能完全被否认却有巨大的经济上的成功。隐藏一个坏的用户经验非常容易。

DJA：是的，我确信。一个已经身价9亿美元的站点，投资者不会看到进一步改善该站点有利可图。

AC：是的，非常正确。

第二部分：拯救这个世界 -- ″一点一滴地″

"让人做他们胜任的事情，让计算机做他们真正胜任的事情。"

在我对Alan Cooper访谈的第二部分，我们试图回到原先计划好的问题。如果你还没有读第一部分，你需要读完后在看这一部分。在第二部分，我们谈到了Cooper的书。《关于脸》和《精神病人管理精神病院》，我们谈到了Web站点、可携带、无线Web设备和WAP电话的交互设计问题。我们谈到了旧工业时代市场销售技巧，为什么信息时代不是广播媒体和旧大众市场销售所设想的那样，以及"设计"这个单词的不妥之处。

在整个过程中，Alan Cooper证实了他不仅仅是一个革新人物，具有交付更实用产品的天赋能力。决不仅仅是。除了这一点，他对这些问题也有深刻认识，包括：高技术产业的整个框架、为什么你需要看到所有的问题--市场销售，产品设计，决策，商标和技术以及为了打败竞争对手而以用户为核心的角度。

复杂性不是Web交互的症结

DJA：我感觉到网络上可以得到信息的数量飞速增长。商业业主对信息的需求和信息，也是飞速发展的。所有这些都以比我们能够理解以及解决这些问题更快的速度发生着。结果是差距越来越大，因为我们不能很快地处理这些问题。你同意吗？

AC：我同意！我谈一下其中的原因。问题的症结不在于复杂性。人的大脑在理解和管理复杂的事物方面，是一个糟糕的工具。但是这也是人的大脑擅长的东西。人的大脑处理奇妙的复杂的任务。人的大脑在瞬间看到一张脸，认识它并记住它。迄今为止，我们还没有任何计算机程序能够做到这一点。连小孩子都可以做到这一点--记住成千上万的脸。

当你在第二天醒过来，环顾四周，你看到如此多的东西，以致于如果复杂性是一个问题，我们就会昏到过去。

所以复杂性不是问题的症结所在。

记住程序员所做的事情：他们扮演的是一个中央处理器。如果你正在扮演这个角色，然后让计算机来完成这个识别脸的任务，那么你就太荒诞了。你说："天哪，问题的症结在于复杂性。"

人的大脑不是那样工作的，人的大脑和传统的中央处理器是非常不同的，我强烈推荐这本书，Steven Pinker的《人脑如何工作》"How the Mind Works"。

关键在于，让人去做他胜任的事情是一件非常好的事。让人做人擅长做的事情，而让计算机做计算机真正擅长的事情。

例如，使用任何一种操作系统，比如，Linux, NT, MacOS, 当你创造了一大堆数据，你必须把它放到一个有名字的数据区或者某个文件中。然后你必须把它放在某个定位的存储层次的某个位置。所以你必须选择一个名字，选择和指定一个地方，树中的一个结点。如果你想重新获取那个信息，你必须记住你给它的名字，以及你存储它的位置。然后你到那个地方，按照它的名字获取它。你可以找到那个数据。这是非常逻辑也是非常适合的，但这是为计算机指定的模式。

问题在于人类不擅长记忆名字和位置。在这种确切性，特别是递归层次中，不管什么层次结点总是类似的。当然，计算机恰巧擅长于记忆此类东西。但是，计算机并不有助于记忆。计算机只是代理人类做"记忆"方面的工作。这是因为发明计算机操作系统的人 …那种方法来自Kernighan 和Plauge，来自Unix. 如果你是一个计算机程序，那么记住某个目录中一个简单的文件名字只是小菜一碟。他们（操作系统的发明人）却将这个任务分配给了计算机用户。

但是对于人类就不那么容易了。没有人曾经返回然后说："啊，这是对的吗？"在我的第一本书《关于脸》("About Face")中，我详细谈论了这个问题。

真正有趣的事情之一是万维网真正减少了记忆的强度--即计算机程序世界要求用户处理递归，层次性文件系统以及记住名字和存储位置的问题。这就是为什么使用web要比使用桌面软件要容易得多的原因。

问题在于这样做的花费是web的功能要弱得多。现在我们看到的是web越来越强大，用户不得不做更多的记忆工作，我们倒退到了旧的模式，结果是web也变得越来越难使用。

问题的症结不在于复杂性，在于不合适地分配任务。

如果我告诉你，"这儿有一个你要记住的14位的数字"，如果你是一台计算机，让你完成这个任务是合适的；而如果你是一个人，让你做这件事就是不合适的。

无线Internet

DJA：如果你要通过一个无线设备进入Internet，并且必须使用电话机数字按键键盘，会更困难。

AC：是的。你看着电话，看见12个按钮(数字加上*和#)。这12个按钮非常非常古老了。从目标导向的观点来看，它们是那么地不堪一击。

即使我一直使用我的电话机，但是我最不想做的事情是根据号码查阅我的电话号码薄。事实上，我也不想根据名字查阅电话号码薄。所以比起我记356-xxxxxx等来，1-800-flowers要方便得多。我真正想做的是能够看一幅花的图或者能够对电话说"花"。

DJA：所以，好的交互设计是要大量削减人必须记忆的东西，注重目标是什么，人类所擅长的以及机器擅长的。让机器做它擅长的事情。

AC：的确是。

如果你有一个现代的移动电话，那么你可以把50或者100个电话号码记录到存储器中，并且可能有8或10个你经常拨打的电话。如果你要多次从头想起一个电话号码--区号，前缀，号码 (可能是10个中的一个)，其它9次，你使用电话已经记住的号码。但是电话的物理接口是看起来是要你从头到尾拨打这些数字。

你需要转到功能区，并且转到元模式以便拨打电话已经记住的你经常拨打的号码。这是因为我们并不是从目标导向的角度来思考问题。我们并没有看到人们做这些事情的方式。相反，我们看到了技术，然后我们说，"打电话给某个人的方法是拨打某个号码。"然后我们说，"嘿，我们可以让它记住电话号码以便使我们更方便。"

为什么我们不说，"人们总是打电话给固定的几个人"。然后你可以象在电话的顶端设置一个象门把一样的东西，它可以旋转到你要经常拨打的20个人。如果偶尔碰到了一个陌生的号码，再转到拨打号码（数字按键键盘）的模式。

当你使用目标导向的方式思考问题，你可以很容易得到结论。否则如果你总是从技术或者商业的角度思考问题。商业总是看到竞争，而技术人员总是倾向于看到昨天的技术，然后扩展它。

告诉我们关于《精神病人管理精神病院》一书

DJA：自从《精神病人管理精神病院》一书出版以来，已经有一年了。它的观点被接受吗？读者是否同意书中的观点？

AC：这本书是一个巨大的成功。它就好比一个小池塘的一条大鱼 (他笑了)。如果你观察Amazon.com，你就会发现大约43种评论。其中的40种热切赞扬该书，3种说，"这不是编程人员，而是其他人写的"，这证实了我的观点。

我绝对感觉到这本书与众不同。

一共花费了20个月来写这本书。在1999年4月出版。当我开始这本书时，这个世界包括程序员和管理人员。我感觉到管理人员逐步降低了他们负责的角色然后把产业的控制权交给程序员。但是与此同时，程序员尽力地做得最好但是却使用了错误的工具。这就是我这本书名字的来历，《精神病人管理精神病院》("Inmates are Running the Asylum")。我正在向管理人员推荐这本书并且告诉他们，把权力下放给程序员是在犯错误。

我要说的是，正确的做法是你们(管理人员)应该把设计带入这个世界。

所以，欢迎来到2000年。现在，设计人员是任何事情的一部分。今天的公司，无论是小的还是大的，旧的还是新的，已经在VP这个层次为用户经验、用户交互和首席经验执行官提供了位置。这些都是很典型的。你可以在从IBM到Yahoo!，甚至你从来没有听说过的小公司找到这些角色。

DJA：这是一个硅谷现象吗？目前为止，我自己，还没有遇到这个问题。

AC：它风卷了整个产业。我认为1到2年内人们会普遍地看到这一点。现在它已经影响巨大了。美国商业并不会轻易创造一个VP职位。所以这是很严肃的。人们相信用户经验是重要的。有一个设计思想是解决方案的一部分。

提升交互设计为一个职业

DJA：我猜我的工作是交互设计。但是，我发现几乎不可能找到愿意雇佣交互设计师的公司。我所有做这件事情的同事都有别的工作头衔。我感觉到交互设计没有得到应有的重视。

我看到很多web设计人员谈论到信息结构以及它对于一个好的web站点的设计是如何的重要。所以这里有信息工程，实用性工程，HCI(人机界面)等等，以及Jakob Nielsen推荐实用性工程的书的巨大成功。

那么，我们要做什么才能提高对交互设计的认识和提高对"交互设计不是实用性工程"这一问题的认识？

AC：是的，你触及到了我们遇到的问题。不幸地是商界对"设计"有不同的解释。设计被认为是一个创造性的过程，它被运用在重大举措的最后并且对产品影响很小。所以"设计"这个词是有问题的。

从另外一方面讲，实用性工程，以及所有实用性的事物，具有非凡的优势，即可以用假的科学数据来影响许多人。

我把我的任务看作是“使世界因为人性而安全，一点一滴地！”

所以我着手如何做到这一点。我认识到你必须做的事是创造一个学科，这个学科以用户、电子客户和另外一边的人为中心。所以，我们在这里创造了我们的方法学，称为"目标导向"的途径。通过理解他们的目标来理解用户。通过理解用户来理解目标。

我认为这就是设计过程。我们把目标导向的途径填入我们的设计过程，并且那样对待它。我称自己为一个交互设计师。但是，这是一个有趣的问题。当你说，"设计"，你把你自己放在产业的边缘。我们认识到的是，如果你采取目标导向的途径并且把它应用到界面设计，你就可以得到一个更好的界面。但是这不是重要的。它不会对于产品的成功非常深刻的影响。它只会使它更好。但是，产生深刻影响时，将为时已晚。

使用交互设计和目标导向的途径将对产品结构决策产生巨大的好处，以及对公司的决策位置产生巨大的有益影响。事实证明目标导向的途径具有巨大的决策好处和价值。

DJA：所以我们该如何称呼它呢？我们是不是该称呼它为交互体系结构而不是交互设计？

AC：是的，我们也在考虑这个问题，而且我们正在权衡。

设计是有害的

AC："设计"这个词在商业的世界里是有害的。

第一，最重要的事情是，要使质量和效益最大化，并使顾客高兴，你，产品开发先生要做的事情是**在过程的早期引入目标导向的交互设计。**

当你说"设计"这个单词时，商人告诉你在过程的最后，所以我们处于这个奇怪的对氧气熟视无睹的世界里，我们选择的名字对于我们要达到的目标是有害的。我们还不知道解决问题的方案。

这是一个巨大的问题。

实用性工程的所有事情是非常非常昂贵的。也就是说，"我们会使产品更好"。这是很好的！但是它并没有说这样做的费用是多少。这就好比空气清洁装置。空气清洁装置，是一个好事情吗？如果我们首先没有污染我们纯净的天空不是更好吗？是的！

所以我发现自己处于一个人人皆知的批评实用性的位置。我不是一个实用性的批评者，因为我认为这没有效。不。这就好比批评化学疗法因为如果我们阻止癌症的话就会更好，而不是使用非常严格的方法去治愈它。

DJA：所以我们可以理解互设计的概念为防止而不是治愈？

AC：我想说的是好的目标导向的设计允许你在第1阶段时就产生第3阶段质量的产品。

人口统计和角色

DJA：让我们给市场销售人员谈论一点角色和使用场景。

我最近有一个拥有价值昂贵的市场调查信息的客户。这个调查定义了市场销售部分，形形色色潜在的客户。问题在于我对定义的模糊认识。

但是，客户的销售人员确实认为他们的调查遇到了这个问题，所以他们对角色的定义感到怀疑并且公开地说这是在浪费时间。你对提出这种反对的人该说什么呢？你怎样使你的方法学被接受呢？

AC：对，这的确是一个问题。所有人都希望有更好的解决方案和更高兴的客户。

产业设计是一门学科，这门学科已经存在一段时间了，并且发展为一门帮助消除产业时代冗余的方法学。是的，产业时代已经结束了。现在是数字时代。现在我们需要数字方法学。产业设计对于制造真正的好按钮是一件好事情。要人们理解按下按钮后的结果是不必要的。后果是一个交互设计的问题。这是一个数字问题。

对于市场销售道理也是如此。所有的销售人员是暂时的市场商人。他们被授以市场销售技巧。在五六十年代收音机和电视达到最大潜力的时候这些方法是成熟的。这些是广播的典型例子。

DJA：所以是不是市场销售人员没有阅读Regis McKenna,或者Geoffrey Moore和其他人的理论？或者他

们阅读了但是没有理解？或者虽然阅读但是误解了意思？

AC：所有这些情况都有。"那件事情"没有被反映。他们有一套他们使用的现成方法。在广播媒体时代这些技巧的确管用。当你广播你的消息，你必须考虑到人口统计学和市场销售渠道。当你在数字年代，处理同样的复杂的事情，你必须考虑到用户和特定场景。

我使用的例子是一个在加州，Haywood汽车厂装配线工作的小伙子。他是一个工人阶级的小伙子，喜欢看电视和喝啤酒，在周末他跳入他的敞蓬小型载货卡车，然后到达内华达的山脉进入他最喜欢的鲑鱼垂钓小溪流和浅滩，穿着他的防水长靴，用假蝇钓鱼。

这里有一个是汽车厂董事会的执行主席。他在昂贵的法国餐厅里吃饭。他挣许多的钱。他开着奔驰牌子的越野车。他也想远离所有这一切。所以他开着他的奔驰牌子的越野车，到同样的一条溪流中，穿者他的昂贵的Orbis Hip牌子的防水长靴，站在离一个为他工作的装配线工人不到50英尺的地方。他们都想抓住一条鲑鱼。

从人口统计学的观点看，这两个人并没有区别。现在，如果你拥有www.flyfishing.com这些人就是你的市场。这不再是一个广播世界。所以所有旧的市场规则都不再适用。

现在还说因为计算机化这个世界正在改变已经是陈词滥调了。但是，人们并没有意识到计算机化也带来了一些负面的影响。市场销售人员称万维网为"新媒体"。问题在于他们认为它跟新媒体一样。事实上不是的！他们认为TV就是收音机加上图片，但不是那么简单。万维网和那些广播媒体大不相同。

市场销售人员习惯于依据人口统计学和思考诸如如何销售牙刷之类的问题。是的，那和如何销售手机大相径庭。怎样使用牙刷是相同的，因为牙刷是一个机械装置。我并不需要考虑你是高还是矮，是瘦还是胖。你怎样使用牙刷和别人怎样使用它是一样的。

但是，你可以从人口统计学的角度说，同样的性别，同样的年龄，同样的收入，但是你使用手机的方式可能大不相同。因为手机是一个信息实体而不是一个物理实体。

所有以前的市场销售思考方式不再适用。

所以，所有的市场调查数据不是浪费，但是你需要有所侧重， 并且让人们意识到它不再是大众市场销售。这是个人市场销售。

DJA：我当然发现市场调查非常有用，因为它引导我们从广泛的信息中选择特别的角色。

AC：绝对正确。所有的调查都是非常重要的。我们博采众长。我们特别需要市场调查。我们会消化所有的信息。我们也需要亲自调查以确实。

我们需要闲逛，寻找惊奇。有用的信息往往是令人惊奇的。如果我们可以预测惊奇，就不需要我们去找了。

新媒体和旧媒体

DJA：所以简单地试图"灌输信息"不再有效，因为你不能确信人们想听到这个信息。

AC：说"旧媒体，新媒体"很容易，然后理解旧媒体为水彩画，新媒体为油画。但是根本不是那么回事。

旧媒体是石头里的雕刻，而新媒体是同另外一个星球交流。这是两件截然不同的事。我们还没有得到"正确的答案"。

如果你走出web的世界，你可以看到对大众传媒难以置信的强调。我相信大众传媒将会慢慢退出舞台。但它不会消失。同样的，收音机并没有使报纸消失，而TV也没有使收音机消失，新媒体和万维网也不会使大众传媒消失，但是会永远改变。收音机没有使报纸消失但是它改变了报纸在我们日常生活中的地位。TV没有使收音机消失，但是它降级到我们的汽车里去了。

DJA：就象摄影改变了绘画？

AC：千真万确！ Internet将会使大众传媒对人们生活影响降低。你仍然可以看电视广告，寻找最便宜的日常生活用品，但是这不再是大部分人获取信息和商业广告的主要渠道。

过去通常人们和自己的邻居交换货物，逐步到城市和商店交换，逐步到超级市场和百货商店，然后零售经验改变了。人们仍然进行物物交换但是那只是非常非常小的一部分。不是所有的零售都要在web上进行，但是巨大的一部分交易将要在web上进行。

所以你仍然可以看见人们逛商场和诸如"胜利女神城"之类的地方，然后购买一些作为购物纪念的东西。人们仍然进行这种零售。但是，如果你要购买一双和现在已经烂掉的旧鞋一样的鞋，我不会到商店去。

我会到Internet上去购买。

总结

DJA：总之，你还想谈论其他的吗？我们好象偏离了正题。

AC：很清楚的，你抓住了这个术语，"我也是"。

令人鼓舞的事是，当我最近和自己人谈论到这个话题的时候，我说，"当我们8年前开始的时候，我们说我们正要改变这个世界因为人们还未意识到目标导向设计的强大力量"。我们正在改变然后让人们意识到这一点。

所以第二天，我对人们说，"我们赢了！"，因为现在每个人都意识到了。虽然这一思想并没有传播到世界的每个角落但是基本上已经做到了，这不是讨论是否值得注意的问题，这是个"我们应该多注意"的问题。我认为这是一个巨大的胜利。

我真的认为我们已经有所进展，并且我真的很激动。虽然，每天，我看到实在糟糕的数字产品的例子，我知道那些生产"数字产品"的公司的领导层，他们明白自己活在借来的时间里并且他们最终要解决那些问题。

我认为事情在变好之前仍然会更差。但是，我确信事情会越来越好。但是5年前，还在怀疑这个问题是否存在。

DJA：Alan,非常感谢你接受我的采访。

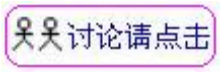
AC：总之，"再接再厉, David！"

钱五哥答疑

http://www.umlchina.com/expert/qlw5.htm

计算机科学博士，专业方向：软件工程。现为 Bell Lab Research China 高级研究员。1995 年起开始参与各种软件开发项目，后来逐渐转入组织/项目过程管理的研究。重点：软件过程。钱五哥的主页：http://qlw.126.com

Gigix译文专栏



模式与 XP

Joshua Kerievsky 著，Gigix 译

概述

模式和极端编程（XP）都为软件设计、开发者提供了无法用金钱衡量的帮助。但是迄今为止 XP 大量关注于重构（refactoring），而对模式只字不提。在这篇文章中，我问“为什么”，并且最终描述出模式怎样以 XP 的方式更好地实现、以及 XP 怎样因为包含对模式的使用而变得更好。

致谢

非常感谢 Kent Beck、Martin Fowler 和 Ward Cunningham，他们为这篇文章提出了友善的评论。

仍在所知不多的时候我们就开始了自己的程序设计生涯，生产出的软件也反映出了我们的缺乏经验：我们创建的代码臃肿、错误百出、脆弱、难以维护、难以扩展。随着时间的流逝，我们成为了更好的软件设计者：我们从技术作家、专家那里学习，我们从自己的错误中学习。现在我们编写具有高度灵活性的软件，它适应广泛而且坚固。当被请求编写一个新的系统时，我们知道查明当前和将来的需求，这样我们可以设计软件来处理当前和将来的需要。

在软件开发生涯的这个阶段，极端编程告诉我们，我们经常对软件过分设计（over-engineer）了。我们从自己的错误中学到了太多，我们不希望重复这些错误，所以我们在系统生命周期的早期做了大量的努力来创造灵活而坚固的设计。不幸的是，我们没有认识到：如果这个系统永远不需要这个程度的灵活性和坚固性，那么我们所有的工作就都没有意义了。我们过分设计了。

我也曾经过分设计过。说实话，与其他设计者坐在一间房间里考虑如何设计软件来适应许多当前和将来的需求，这的确是一种乐趣。我们把自己学到的所有东西——尤其是那些最好的经验——应用在设计中。我们常常知道需求的列表会改变，但用户或客户总是改变需求。不过，我们认为我们可以足够聪明地设计软件，使软件足够灵活，使它能应付所有的需求变化。

典型的过分设计。

今天，极端编程将告诉你这是多么愚蠢的做法。XP说，我们必须让设计自己显现出来，而不是去预测设计将是什么样子。XP说，“做可能起作用的最简单的事”，因为“***你将不再需要它***”。另外，Kent Beck说：

> **你需要在一个强调沟通、简单、反馈和勇气的价值系统中选择最好的工作方法，这样你才能勇敢的脱离过分设计。[Beck1 00]**

同意。但是，现在我必须提到我的朋友Norm Kerth。Norm在软件开发领域有丰富的经验和见识。一年以前我问他“对XP有什么想法”。他说：

> **我喜欢XP里的每样东西。我关心的是：还有什么不在XP中。[Kerth 99]**

当时，我只认为Norm是一个保守派。但现在我不能确定了。XP明显缺少的就是使用模式的经验。尽管一些XP的创始人帮助建设了模式社团，但没有哪一个坚定清楚的说明模式如何适应XP。

一开始，这还没有让我感到迷惑。但现在，我的确感到迷惑。

我感到迷惑，因为我在XP和模式上的经验让我相信：在XP的场景中模式会工作得更好；并且当XP包含模式时，XP也会工作得更好。

这需要一些解释。我将从描述我自己使用模式和XP的一些经验开始。

从1995年开始，我开始沉浸入模式之中。我学习模式文献、主办了一个每周一次的模式学习组、使用模式设计和开发软件、并进行UP（一个关于使用模式的国际学术会议）的组织和运转工作。说我“热衷于模式”实在是一种保守的说法。

当时，就象很多第一次学习模式的人一样，我有一点过分渴望使用它们。这不是一件好事，因为它会让你的设计比需要的更复杂。但我没有意识到这一点，直到我开始学习重构。

大概在1996年，我第一次接触到了重构。我开始实证它并很快观察到重构带我离开了我在模式学习中学到的某些原则。

举个例子，那本里程碑式的书——**《设计模式：可复用面向对象软件的基础》**——中的一个原则是：

针对接口编程，而不是针对实现编程。[GHJV1 95]

《设计模式》的作者们做了相当精彩的工作来解释为什么我们需要遵循这条建议。几乎在所有的模式中，都讨论了当你针对某个特定实现编程时你的软件如何变得缺少灵活性和可修改性。几乎每一次都是接口过来帮忙。

但如果我们不需要灵活性和可修改性，情况又是怎样？为什么我们要在开始设计时预料一些可能永远不会出现的需要？这是我的一次觉悟。所以随后我记录下了下面这个JAVA技巧：

不要分离类和接口

我曾经习惯于在我的接口名字后面加上一个“I”。但当我继续学习更多的重构技术时，我开始看到一种明智的做法：把类名和接口名设计成一样。下面是原因：在开发过程中，你知道你可以使用一个接口来让某些东西变得灵活（使实现多样化），但可能你现在根本不需要让实现多样化。所以，放下预测太多的“过分设计”吧，你仍然保持简单，仍然把东西放在一个类中。在某个地方你会写一个方法语句来使用这个类的对象。然后，几天、几星期、几个月之后，你明确“需要”一个接口。因此你就将原来的类转换成一个接口，再创建一个实现类（实现新的接口），并且让你原来的语句保持不变。[Kerievsky 96]

我继续学习类似于重构的课程，逐渐的，我使用模式的方式开始改变了。我不再预先考虑使用模式。现在，我更加明智了：如果某个模式可以解决某个设计问题，如果它提供一种方法来实现一个需求，我就会使用它，但我将从可以编码出的模式的最简单实现开始。晚些时候，当我需要增加或修改时，我将让这个实现更加灵活、稳固。

这种使用模式的新方法是一种更好的方法。它节约了我的时间，并让我的设计更简单。

由于我继续学到更多关于XP的知识，我很快开始考虑这样一个事实：那些清楚介绍“XP是什么”和“XP如何工作”的人毫不提及模式。看起来，焦点已经全部从开发转向了重构。构造一点，测试一点，重构一点，然后再重复。

那么，模式怎么了？

我收到的一般的答案是：模式鼓励过分设计，而重构保持事情简单、轻量级。

现在，我和其他任何人一样喜欢重构——我回顾了Martin Fowler的书的关于这个主题的两份手稿，然后知道重构将成为一个标准。但我仍然喜欢模式，我发现模式在“帮助人们学会如何设计更好的软件”方面是无价之宝。所以，XP怎么能不包括模式呢？！

我小心的在Portland Pattern Repository上写下了我的不安。我问：是否完美的XP模式应该由完全不知道模式的程序员和指导者组成，是否他们应该完全依赖重构来“让代码去它该去的地方”。Ron Jeffries，世界上最有经验的XP实践者，与我争论了这个主题，并且这样写：

> **一个初学者不能倾听代码所说的话。他需要学习代码质量的模式（在一般意义上）。他需要看好的代码（以及，我猜，差的代码），这样他才能学会写出好的代码。**
>
> **一个问题——我的意思是一个可能的问题——是，现在的模式是否被用于帮助提高代码的质量。我想Beck的Smalltalk Best Practice Patterns会有帮助，因为那些都是非常小型的模式。我想设计模式都更值得怀疑，因为模式和讨论有时变得相当大，而且它们可能造成看起来合理的庞大解决方案。Martin Fowler的精彩的分析模式也有同样的危险：在可以选择一个小规模解决方案的时候选择了大规模的解决方案。[Jeffries 99]**

一个非常有趣的关于模式的观点。尽管我已经看到模式可以被明智的实现、使用，但Ron看起来却认为它们是危险的，因为它们“让庞大的解决方案看起来合理”。在其他地方，Ron观察了一件经常发生的事情：第一次学习模式的人们如何过分渴望使用它们。

我无法不同意后面这个观察结果。就象任何新事物——甚至是XP——一样，人们可能会过分渴望使用它们。但模式真的鼓励在可以使用小规模解决方案时使用大规模解决方案吗？

我想这主要取决于你如何定义、使用模式。举个例子，我观察了许多模式的初级使用者，他们认为一

个模式与它的结构图（或类图）是完全相同的。只有在我向他们指出“模式可以根据需要以不同的方式实现”之后，他们才开始发现这些图只是表示实现模式的一种方式。

模式的实现有简单的也有复杂的。诀窍是：发现模式针对的问题，将这个问题与你当前的问题进行比较，然后将这个模式最简单的实现（解决方案）与你的问题进行比较。当你这样做时，你就不会在可以使用小规模解决方案的时候使用大规模解决方案。你获得了解决问题最好的平衡。

当人们没有受过模式的良好训练时，困难就可能出现。Ron提到人们使用模式的方式是“现在构成的”——这就是说，他们如何与现在的作者沟通。我同意模式文献有一些缺点。关于模式的书很多，你可以花一些时间来理解模式解决的问题，这样你就可以聪明的根据自己的特定需要选择模式。

这种选择是极其重要的。如果你选择了错误的模式，你可能过分设计或仅仅把你的设计揉在一起。有经验的模式使用者也会犯错误，并且经常看到这样的结果。但这些专家有其他的模式作为装备，这些模式可以帮助他们面对自己的错误。所以他们最终经常把自己真正需要的模式换成了不那么理想的模式。

那么，你将怎样成为一个有经验的模式使用者呢？我发现除非人们投身于大量模式的学习中，否则他们就有可能陷入误解它们、过分使用它们以及用它们过分设计的危险之中。

但这是避免使用模式的一个原因吗？

我想，不。我发现模式在如此多的项目中如此有用，以至于我无法想象不使用它们来进行软件设计和开发。我相信对模式的彻底的学习是非常值得的。

那么，XP对模式保持沉默是因为感觉到它们将被误用吗？

如果情况是这样，也许问题已经变成：**我们怎样使用模式中的智慧，而避免模式在XP开发场景中的误用呢？**

在这里，我想我必须回到《设计模式》。在“结论”一章、“设计模式将带来什么”一节、“重构的目标”小节中，作者写道：

> **我们的设计模式记录了许多重构产生的设计结构。在设计初期使用这些模式可以防止以后的重构。不过即使是在系统建成之后才了解如何使用这些模式，它们仍可以教你如何修改你的**

系统。设计模式为你的重构提供了目标。[GHJV2 95]

这就是我们需要的观点：重构的目标。这就是重构和模式之间的桥梁。它完美的描述了我自己在如何使用模式方面的进步：从简单开始，考虑模式但将它们保持在次要地位，小规模重构，只有在真正需要模式的时候才把重构转移为模式。

这个需要训练和仔细判断的过程将很好的适应XP所包含的最好的习惯。

而且这个途径很明显与“故意不知道或不使用模式而只依赖重构来改善设计”的方法非常不同。

只依赖重构的危险是：没有目标，人们可能使设计小小进步，但他们的全面设计将最终受损害，因为这种方法缺乏顺序、简单性和效力，而聪明的使用模式则可以让开发者拥有这些。

引用Kent Beck自己的话：**模式生成体系结构**。[Beck2 94]

但模式不保证有纪律的使用。如果我们在设计中过多、过早的使用它们，我们就又回到了过分设计的问题。因此，我们必须回答这个问题：“在设计的生命周期中，何时引入模式是安全的？”请回忆上面对《设计模式》的引用：

在设计初期使用这些模式可以防止以后的重构。

这是一个聪明的主张。如果我们不知道“何时配置一个模式”的基本规则，那么我们就很容易在设计周期的早期就陷入过分设计。

再一次，问题又全部集中在一起：如何将项目中的问题与一个合适的模式相匹配。

在这里，我必须讲述我为不同行业开发软件得到的经验。

有一家客户要求我和我的团队用JAVA为他们的网站构造软件，这将是一个很酷的交互式版本。这个客户没有任何JAVA程序员，但仍然要求能在他们需要的任何时候、任何地方修改软件的行为，而不必做程序的修改。多么高的要求！

在对他们的需要做了一些分析之后，我们发现Command模式将在这个设计中扮演一个非常重要的角色。我们将编写命令对象，并让这些命令对象控制软件的整个行为。用户将可以参数化这些命令、将它们

排序、并选择命令运行的时间和地点。

这个解决方案工作得很完美，Command模式正是成功的关键。所以在这里，我们不会等到重构的时候才使用Command模式。相反，我们预先看到了使用它的需要，并从一开始就用Command模式来设计软件。

在另一个项目中，系统需要作为独立应用程序*和*WEB应用程序运行。Builder模式在这个系统中发挥了巨大的作用。如果没有它，我不敢想象我们会拼凑出一个多么臃肿的设计。Builder模式的作用就是解决“多平台、多环境运行”这样的问题。所以在设计早期就选择它是正确的。

现在，我必须声明：即使在设计的早期引入了模式，但一开始仍然应该按照它们最原始的样子来实现它们。只有在晚些时候，当需要附加的功能时，模式的实现才能被替换或升级。

一个例子会让你更清楚这一点。

上面提到的由命令对象控制的软件是用多线程的代码实现的。有时候两个线程会使用同一个宏命令来运行一系列命令。但一开始我们并没有被宏命令的线程安全问题困扰。所以，当我们开始遇到线程安全造成的莫名其妙的问题时，我们必须重新考虑我们的实现。问题是，我们应该花时间构造宏命令的线程安全吗？或者有没有更简单的方法来解决这个问题？

我们用更简单的方法解决了这个问题，并且避免了过分设计：为每个线程提供一个独立的宏命令实例。我们可以在30秒内实现这个解决方案。请把这个时间与设计一个线程安全的宏命令所需的时间做一下比较。

这个例子描述了XP的哲学怎样在使用模式的情况下保持事情简单。没有这种简单化的驱动，过分设计的解决方案——就象线程安全的宏命令——很容易出现。

因此，简单化和模式之间的关联是很重要的。

当程序员需要做出设计决策时，很重要的一件事就是：他们应该试图保持设计简单，因为简单的设计通常比庞大而复杂的设计更容易维护和扩展。我们已经知道，重构意味着将我们保持在简单的路上：它鼓励我们以小而简单步骤逐渐改进我们的设计，并避免过分设计。

但是模式呢？难道它们不是帮助我们保持简单吗？

有些人会说“不”。他们认为模式尽管有用，但容易造成复杂的设计。他们认为模式会造成对象快速增加，并导致过分依赖对象组合。

这种观点是由于对使用模式的方法的错误理解。有经验的模式使用者会避免复杂的设计、对象的快速增长和过多的对象组合。

实际上，在使用模式的时候，有经验的模式使用者会让他们的设计更简单。我将再用一个例子来说明我的观点。

JUnit是一个简单而有用的JAVA测试框架，它的作者是Kent Beck和Erich Gamma。这是一个精彩的软件，其中满是精心选择的简单的模式。

最近一些人要求我对JUnit进行DeGoF，也就是说，将JUnit中的设计模式移除掉，以观察没有模式的JUnit是什么样子。这是一次非常有趣的练习，因为它让参与者认真考虑应该在什么时候在系统中引入模式。

为了描述他们学到的东西，我们将对JUnit 2.1版中的一些扩展进行DeGoF。

JUnit中有一个叫做TestCase的抽象类，所有的具体测试类都派生自它。TestCase类没有提供任何多次运行的方法，也没有提供在自己的线程中运行测试的方法。Erich和Kent用Decorator模式很优雅的实现了可重复测试和基于线程的测试。但是如果设计团队不知道Decorator模式呢？让我们看看他们会开发出什么，并评估一下他们的设计有多简单。

这是Test Case在JUnit框架1.0版本中的样子（为了简化，我们忽略了注释和很多方法）：

```
public abstract class TestCase implements Test {

   private String fName;

   public TestCase(String name) {

        fName= name;

   }
```

```
    public void run(TestResult result) {
        result.startTest(this);
        setUp();
        try {
            runTest();
        }
        catch (AssertionFailedError e) {
            result.addFailure(this, e);
        }
        catch (Throwable e) {
            result.addError(this, e);
        }
        tearDown();
        result.endTest(this);
    }
    public TestResult run() {
        TestResult result= defaultResult();
        run(result);
```

```
        return result;

    }

    protected void runTest() throws Throwable {

        Method runMethod= null;

        try {

            runMethod= getClass().getMethod(fName, new Class[0]);

        } catch (NoSuchMethodException e) {

            e.fillInStackTrace();

            throw e;

        }

        try {

            runMethod.invoke(this, new Class[0]);

        }

        catch (InvocationTargetException e) {

            e.fillInStackTrace();

            throw e.getTargetException();

        }

        catch (IllegalAccessException e) {
```

```
                e.fillInStackTrace();

                throw e;

            }

        }

        public int countTestCases() {

            return 1;

        }

    }
```

新的需求要求允许测试重复进行、或在它们各自的线程中进行、或以上两者。

没有经验的程序员通常在遇到这样的新需求时进行子类型化。但是在这里，因为知道TestCase对象将需要能够在同一个线程中重复运行、或在各自独立的线程中重复运行，所以程序员知道：他们需要考虑得更多。

一种实现方法是：将所有的功能都添加给TestCase本身。许多开发者——尤其是那些不了解设计模式的开发者——将会这样做，而不考虑这会使他们的类变得臃肿。他们必须添加功能，所以他们将功能添加到任何可以添加的地方。下面的代码可能就是他们的实现：

```
public abstract class TestCase implements Test {

    private String fName;

    private int fRepeatTimes;

    public TestCase(String name) {

        this(name, 0);
```

```
    }

    public TestCase(String name, int repeatTimes) {

        fName = name;

        fRepeatTimes = repeatTimes;

    }

    public void run(TestResult result) {

        for (int i=0; i < fRepeatTimes; i++) {

            result.startTest(this);

            setUp();

            try {

                runTest();

            }

            catch (AssertionFailedError e) {

                result.addFailure(this, e);

            }

            catch (Throwable e) {

                result.addError(this, e);

            }
```

```
                tearDown();

                result.endTest(this);

            }

        }

        public int countTestCases() {

            return fRepeatTimes;

        }

        }
```

请注意run（TestResult result）方法变大了一些。他们还为TestCase类添加了另外的构造子。到目前为止，这还不算什么大事。并且，我们可以说：如果这就是所有必须做的事情，那么使用Decorator模式就是多余的。

现在，如果要让每个TestCase对象在其自己的线程中运行又怎样呢？这里也有一个可能的实现：

```
public abstract class TestCase implements Test {

    private String fName;

    private int fRepeatTimes;

    private boolean fThreaded;

    public TestCase(String name) {

        this(name, 0, false);

    }
```

```
    public TestCase(String name, int repeatTimes) {
        this(name, repeatTimes, false);
    }
    public TestCase(String name, int repeatTimes, boolean threaded) {
        fName = name;
        fRepeatTimes = repeatTimes;
        fThreaded = threaded;
    }
    public void run(TestResult result) {
        if (fThreaded) {
            final TestResult finalResult= result;
            final Test thisTest = this;
            Thread t= new Thread() {
                public void run() {
                    for (int i=0; i < fRepeatTimes; i++) {
                        finalResult.startTest(thisTest);
                        setUp();
                        try {
```

```
                    runTest();
                }
                catch (AssertionFailedError e) {
                    finalResult.addFailure(thisTest, e);
                }
                catch (Throwable e) {
                    finalResult.addError(thisTest, e);
                }
                tearDown();
                finalResult.endTest(thisTest);
            }
        }
    };
    t.start();
    result = finalResult;
} else {
    for (int i=0; i < fRepeatTimes; i++) {
        result.startTest(this);
```

```
                setUp();
                try {
                    runTest();
                }
                catch (AssertionFailedError e) {
                    result.addFailure(this, e);
                }
                catch (Throwable e) {
                    result.addError(this, e);
                }
                tearDown();
                result.endTest(this);
            }
        }
    }
    public int countTestCases() {
        return fRepeatTimes;
    }
```

}

唔，这看起来开始变得更坏了。为了支持两个新的特征，我们现在拥有了三个构造子，而且run（TestResult result）方法的大小迅速的膨胀起来。

即使不管所有这些新代码，我们这些程序员还没有满足这些需求：我们仍然不能在各自的线程中重复运行测试。为了这个目的，我们必须添加更多的代码。算了，我就放过你吧。

重构可以帮助这些代码减小尺寸。但是只需要稍做思考：如果再接到一个新的需求，我们要怎么办？现在JUnit 3.1支持四种不同的TestCase修饰器，它们可以轻松的随意组合以获取所需的功能。同时，JUnit的实现仍然简单——没有混乱的代码。这种设计保持TestCase类的简单、轻量级，用户只需要在需要的时候对TestCase对象进行装饰即可，而且可以选择任何组合顺序。

很清楚，这是一个模式帮助简化设计的例子。这个例子也说明了缺乏经验的开发者怎样改善他们的设计——如果他们知道模式指出的重构目标。

使用模式来开发软件是聪明之举，但如果你缺乏使用模式的经验，它也可能是危险的。出于这个原因，我极力提倡模式学习组。这些学习组让人们在同伴的帮助下稳步前进而精通模式。

当人们了解模式并以受过训练的方式使用它们时，模式是最有用的——这种受过训练的方式就是XP的方式。以XP的方式使用模式鼓励开发者保持设计的简单、并完全根据需要对模式进行重构。它鼓励在设计早期使用关键的模式。它鼓励将问题与能帮助解决问题的模式相匹配。最后，它鼓励开发者编写模式的简单实现，然后根据需要发展它们。

在XP的场景中，模式的确更有用；而在包含对模式的使用时，XP开发则更有可能成功。

参考书目

[Beck1 00] Beck, Kent. Email on extremeprogramming@egroups.com, January 2000.

[Beck2 94] *Patterns Generate Architectures*, Kent Beck and Ralph Johnson, ECOOP 94

[GHJV1 95] *Design Patterns: Elements of Reusable Object-Oriented Softwar*e, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. 中译本：《设计模式：可复用面向对象软件的基础》，李英军等译。

[GHJV2 95] *Design Patterns: Elements of Reusable Object-Oriented Softwar*e, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Pages 353-354 中译本：《设计模式：可复用面向对象软件的基础》，李英军等译，第6章。

[Jeffries 99] Jeffries, Ron. Patterns And Extreme Programming. *Portland Pattern Repository*. December, 1999

[Kerth 99] Kerth, Norm. Conversation, circa March, 1999.

[Kerievsky 96] Kerievsky, Joshua. Don't Distinguish Between Classes And Interfaces. *Portland Pattern Repository*. Circa 1996

预订和使用可重用实体的分析模式

Eduardo B. Fernandez，Xiaohong Yuan 著，Shane 译

摘要 本分析模式描述怎样预订然后使用可重复使用的实体。我们通过用例来表达模式的需求，使用类模型、状态图和序列图对模式进行具体的描述。本模式对应一个最小的语义单位。

1．介绍

我们在此提出一个分析模式，描述怎样预订然后使用可重复使用的实体。该模式归于我们称为语义分析模式(semantic analysis patterns)的范畴[Fer98]，因为它们强调的是应用模型的语义侧重面，而不是为了增加设计的灵活性。

我们认为这类模式对于以需求为出发点开始建模过程的设计方法是有用的。例如，识别需求中的一些模式产生一个初步的模型，这个模型可用作后续设计的指南。这些模式也能被用来开发框架和组件。

本文的分析模式强调预订的基本侧重面而不处理它的扩展、异常处理和各种变体；可以在以后添加这些内容来定义预订的模式语言。

2．问题

一个客户（个人或机构）需要预定一个可重用的实体（如旅馆房间、车辆、演出座位）供他随后使用。这些实体数目是有限的，能够按相同种类或类型进行分组。

所有这些情形隐含着这样一个请求：在某个具体日期使用某类实体。如果有一个请求的实体可用，它就被请求它的客户预订。实体在客户使用的时候被分配，同时建立一条使用记录描述该实体正在使用。

被预订的实体是可以重复使用的；那就是说，它们不会被客户永久拥有，客户只在一定的时间段对它们有使用权。这一点意味着通过让用户在使用实体后负责将它们完整归还，使用记录同时具备（明确的或隐含的）合同的功效。

图 1 显示一个客户预订旅馆房间的类图。在 Customer 类和 Room 类之间的关联关系描述一个客户的每个预订情况。Reservation 关联类描述正常情况下一个预订中应该包含的信息。Availability 类描述房间的结构和它们的占用情况。

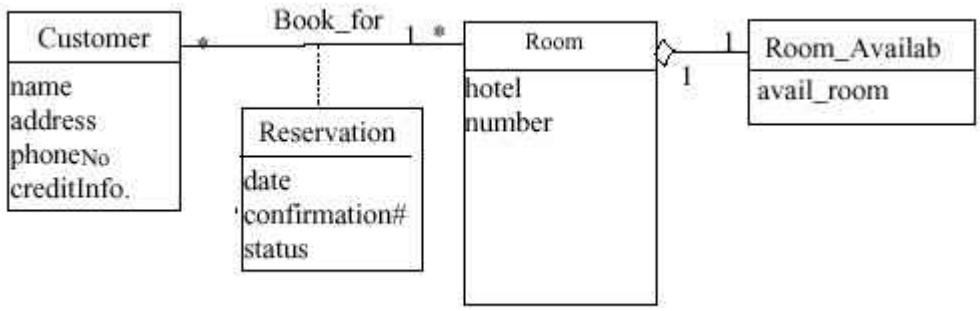


图 1. 预订一个旅馆房间

3. 约束

本模式使用以下约束：

- 正规的分析目标应用于解决方案。一个分析模型必须是对需求的真实可靠的表示，并且必须不能包含任何实现细节。
- 基本语义。模式必须描述一个基本的语义单元。这意味着模式应该是简单的，而且能够描述各种变化的情况。这种方法已在[Ngu98]使用过，在那里被称为一个“最小限度的”(“minimal”)的应用程序。
- 现实文档的模型表示。预订和使用记录通常同时使用文档和它们的软件形式进行记录。

模式语言的约束也应包括方便有效的客户服务（如排队工具）和可靠性（如异常情况的防备措施）

4. 解决方案

4.1 需求

解决方案相应于下列用例要实现的功能。

- 建立预订。主角是客户（个人或机构）。客户请求预订一个一定类型的实体，给出开始使用的日期和使用的时间。如果有适合客户要求的实体，预订成立，该实体被预订。上述相关信息记录在一个预订文档中。

- 使用预定的实体。主角是建立预订的同一客户。在实体被使用前需要建立一个使用记录。使用结束后，实体被归还又可供其它客户预订使用。

- 修改预订。对已建立的预订进行修改。这意味着确定预订的有效性和取消老的预定。

- 取消预订。取消已经建立的预订。一些机构政策可用来定义取消预订的效果。(译者注：原文为“Some institution policies may apply to define the consequences of the cancellation.”)

注意：要求修改预订的功能在建立和取消预订的用例中可以使用。同时，这些用例的异常处理流程未被考虑。

4.2 类模型

图 2 是这些用例实现的一个分析类图。该图对可被预订的实体（类 Entity_type）和使用时被分配的单个实体（类 Entity）进行了区分。UseRecord 类描述实体的使用情况。Availability 类描述检查和跟踪房间占用情况的一般知识。Reservation 类包括正常情况下保存在预订中的信息。Reservation 类和 UseRecord 类之间的关联揭示了这样一个事实：使用记录是建立在预订信息基础上的。

4.3 动态行为

图 3 和图 4 是关于一些最重要的类的状态图。例如，一个预订能够处于已建立（Created）和已确认（Confirmed）状态。图 5 显示一个对实体建立预订和随后使用的序列图。

5. 效果

本模式有以下效果：

- 它能被用于旅馆房间、会议室、飞机和演出的座位、书和录像带等实体的预订。这各种应用表明本模式包含预订问题的基本方面。它也是一个简单的模式，很容易理解。

- 它表示了用例的各个方面，没有包括实现方面的细节。

- 类模型明确包括了预定和使用记录类（Reservation 类和 UseRecord 类），它们是在上述预订系统中使用的基本文档。这简化了跟踪预订历史和正在使用的相关实体的工作。使用记录（UseRecord）作为有效合同的角色也很明确。

然而，本模式对上述所有情形的描述并非完全相同。

- 客户预订飞机和演出的座位时有真实的票据，而在预订车辆或房间时却没有（尽管客户

可以获得一封确认信）。票据代表一个合同。

- 当车辆被客户拿走使用时，车辆的预订明确地转变为一个合同；而旅馆房间的预订是当客户签名入住时隐含地转变为合同。

- 车辆和房间的预订在使用时对具体实体进行分配，而演出和飞机座位的预订是在预订时就对实体进行了分配。

- 有些实体，如车辆和房间，预订一段时间表示为几天，而另外一些实体，如飞机座位，表示为对一个具体的事件进行预订。

- 一个合同不需要预订也可以建立，一个预订也可以不转变成一个使用记录。

此外，还有很多方面的内容没有在本模式中表示：

- 关于实体上下文和环境的描述。例如，一张飞机票应该涉及飞机、机场和起落次数（the number of stops）等。

- 怎样跟踪实体可用性。

- 当实体不可用时怎样对请求进行排队。

- 怎样处理优先客户

- 帐目处理

- 历史

最后，异常流程处理也已被忽略，如无演出和超额预订等情况。

可以考虑将所有这些方面定义为一个模式语言或一个模式家族，这是我们将来要进行的工作。

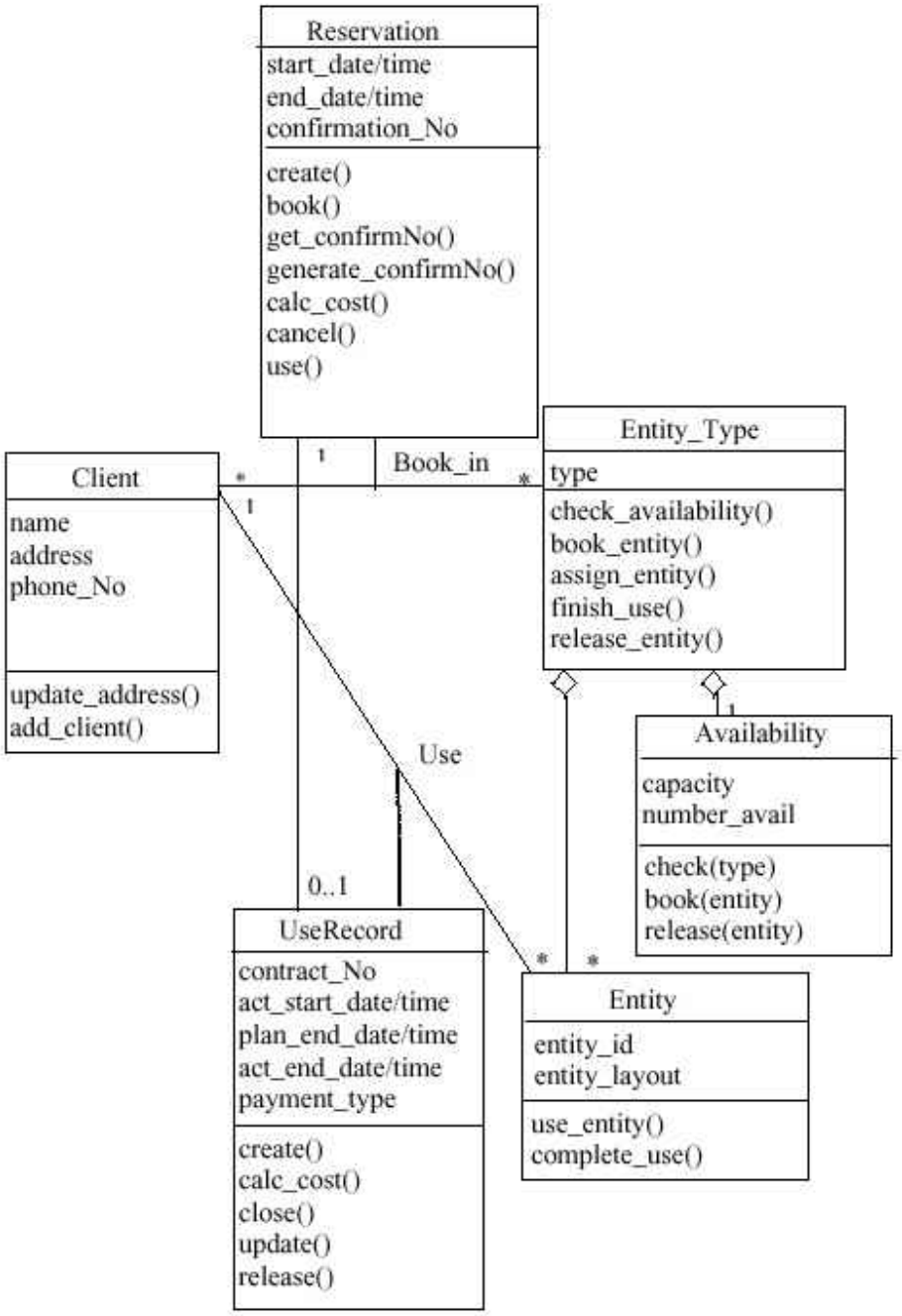


图 2. 预订/使用模式类图

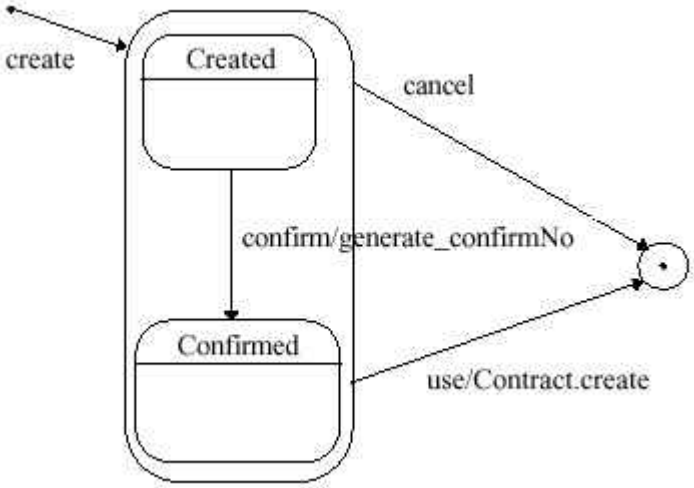


图 3. 类 Reservation 的状态图

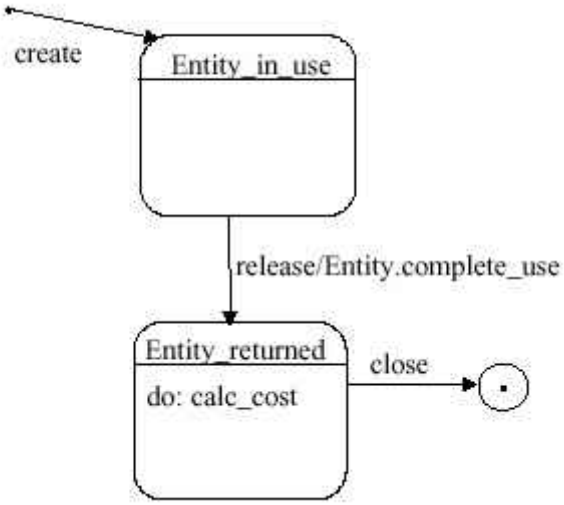


图 4. 类 UseRecord 的状态图

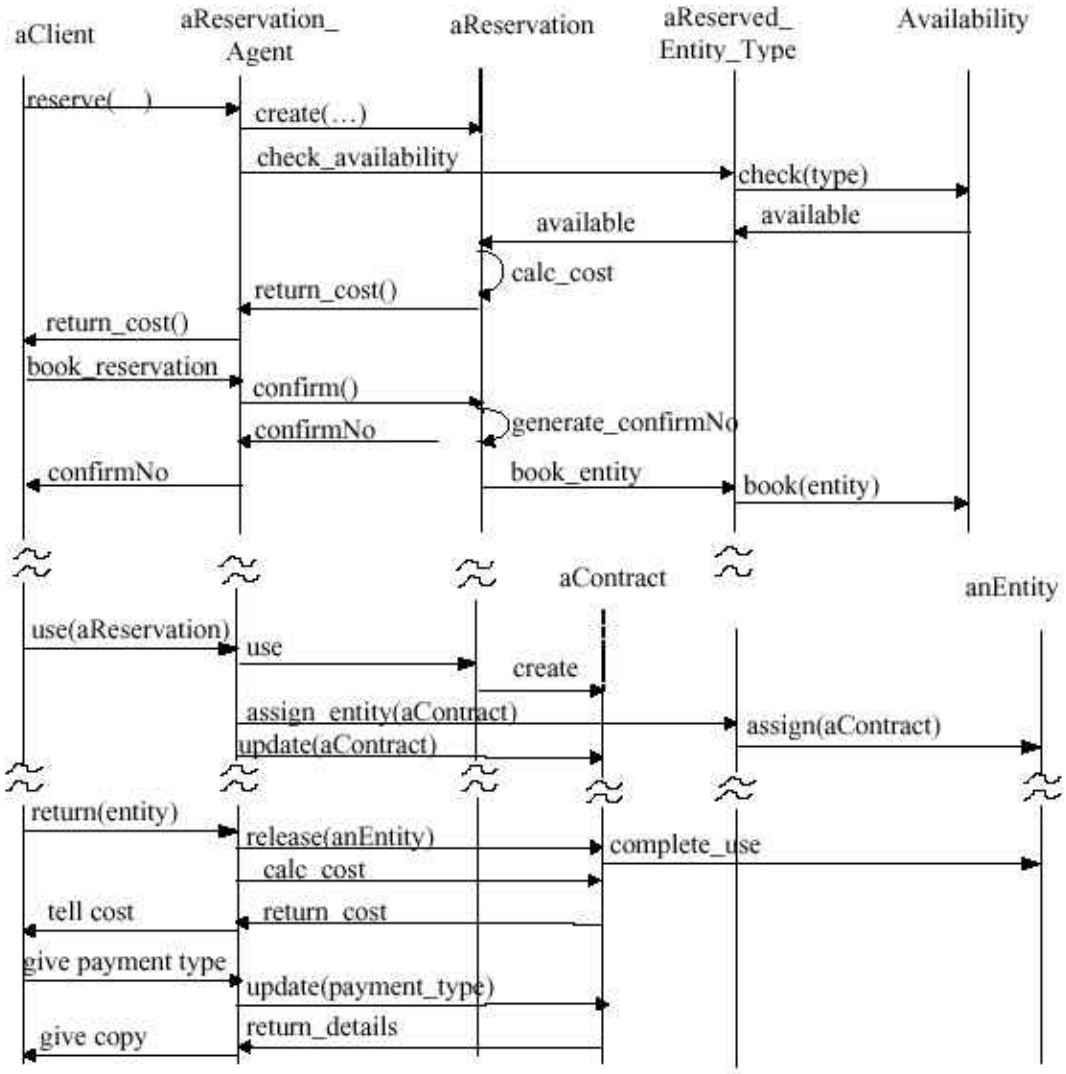


图 5. 预定和使用的序列图

6. 已知应用

根据下列文献和我们的经验，我们发现本模式已被用于以下场合：

- 客户预订演出或飞机的座位[Bak97]
- 客户预订旅馆房间[Fer98a]
- 客户预订车辆[SA98]
- 客户向图书馆预订图书[Fer98a]
- 客户向音像店预订录像带[Fer98a]，[Joh96]

7．相关模式

NATURE 项目[Nat98]中有一些模式，如资源分配和资源使用，从不同观点讲述了某些与本模式相通的内容。Coad[Coa97]提出过一个与本模式在某些方面相通的资源请求模式。Fowler[Fow97]讨论了计划和资源分配。制造业零部件的预订和随后分配方式也与本模式有某些方面的共同点[Fer97]。在所有这些案例中，实体没有被重复使用，因此不需要使用记录。[Bra98]中讨论了书籍和音像制品预订的某些内容，但没有考虑被预订实体的使用情况。

我们的模式作为 Type Object 模式的一个组件使用[Joh96]。事实上，正如[Fer98]中描述的那样，这些复杂的模式通常包括一些本身就具有实际意义的子模式(subpattern)。

参考文献

[Bak97] S. Baker, CORBA: Distributed objects using Orbix, Addison-Wesley 1997.

[Bra98] R.T.V.Braga, F.S.R.Germano, and P.C.Masiero, "A confederation of patterns for resource management", Procs. of PLOP'98, http://jerry.cs.uiuc.edu/~plop/plop98

[Coa97] P. Coad, “Object models: Strategies, patterns, and applications” (2nd Edition), Yourdon Press, 1997.

[Fer97] E. B. Fernandez and Z. W. Peng, “ An object-oriented model for manufacturing inventory control systems”, Tech. Report TR-CSE-97-30, Dept. of Computer Science and Eng., Florida Atlantic University, April 1997.

[Fer98] E. B. Fernandez, “Building systems using analysis patterns”, Procs. 3rd Int. Soft. Architecture Workshop (ISAW3), Orlando, FL, November 1998. 37-40.

[Fer98a] E. B. Fernandez, Class Notes for COP5330, Introduction to Objected-Oriented Software Design, Dept. of Computer Science, Florida Atlantic University, Boca Raton, Florida.

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

[Gam95] E. Gamma et al. Design Patterns – Elements of reusable object-oriented software, Addison-Wesley 1995.

[Joh98] R. Johnson and B. Woolf, "Type Object", Chapter 4 in Pattern Languages of Program Design 3,

Addison-Wesley, 1998.

[Nat98] NATURE project, http://www.city.ac.uk/~az533/main.html

[Ngu98] K. Nguyen and T. Dillon, "An alternative solution to the observation pattern problem", in PLOP98. http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions

[SA98] Software Architects. Exercise in course SA4012, Testing Object-Oriented Systems, 1998.

仓库管理器：一个库存的分析模式

Eduardo B. Fernandez 著，邓克 译

摘要 库存可以让我们知道一个机构有什么，如：零件，成品，家具，机器等。一个好的库存系统对任何一个现代商务或制造系统来说，都是必要的。本文中，我们列举了一个针对库存系统的分析模式，这个库存系统跟踪仓库中物品的数量和位置，并且能够根据制造或生产的不同阶段(从零部件采购到产品发运)，动态更新有关物品的数量。这是一个通用的模型，是根据对一个实际运行库存系统的抽象而定义出来的，并且可以通过扩展，实现一个有更详细需求的或相类似的应用。这是一个组装模式，由两个原子模式组成。

1 介绍

现代制造系统中，制造过程中所涉及信息的管理已经成为降低产品成本，提高产品质量的一个关键因素。很多公司和机构在这个领域投入了大量的资源，制造资源计划系统（MRP）已经变得重要了[Salv92]。库存是 MRP 系统中最重要的一部分，用来跟踪目标对象的数量和位置。

这里，我们开发了一个库存模式－仓库管理器，它满足对可重用性和可扩展性所期望的要求。这个模型不仅考虑了系统的静态视图，还考虑了它的动态方面。因为一个库存控制系统不能被完全地模型化(除非顾及到整个制造系统中几个其他方面的问题)，所以模型中也包含了制造系统的一些功能性部分（非库存方面的）的作用。这个模型可以作为开发一个完整的制造系统模型的起点或是作为一个语义分析模式（Semantic Analysis Pattern）[Fer00a]的实例,这种实例实际上是一个领域范围内可以使用的最小应用。这个模式也是一个组装模式[Rie96]，我们识别出了两个原子模式：基本库存(Basic Inventory)和物品分布(Item Distribution)。

2 问题

商业活动、制造车间、图书馆怎样跟踪各种不同种类的物品和它们的位置？

3 环境

所有机构，不管是大型的，还是小型的，都使用零部件来构建产品或供应其他商家。他们需要能够跟踪物品的数量。在制造车间里，这些物品是零部件和产品；在图书馆里，它们是书，磁带，或者是档案；在服装店里，它们是服装及附属物，等等。这些机构也需要能够找到所需的物品。

4 约束

- 在一个公司里，物品可能是正在制造或已经完成产品的原材料。货物有损失或遭到破坏的可能。机构必须能够跟踪仓库中物品的实际数目。

- 其他功能单元可以改变物品的数量：例如，物品无论何时、何地迁移或使用时，应该更新其相应的库存数量。

- 解决方案必须描述一个基础的语义单元，这意味者它必须足够简单，以便应用到各种环境中。这是可重用性的基础。

- 解决方案必须包括现实的文档解释。

5 解决方案

5.1 需求

一个库存系统的基本功能或用例（Use Case）可以总结如下：

- 把不同类型的货物分开，如：物料或零部件与成品。

- 跟踪仓库中的每个物品。可能需要几种类型的数量。手头（onHand）数量指示了物品的现存数量。定购（onOrder）数量指示了将来有多少物品可以入库。保留（reserved）数量指示了为订单保留的数量，它暗示了一个员工以后可以使用的数量（可获得数量）是手头数量与保留数量的差额。在更复杂的系统中，可能还会用到其他类型的数量。

◆ 跟踪物品的位置。库存应该记录物品在特定位置上的分布。

由这些功能的本质所决定，几乎所有其他子系统都会对库存中记录的数量有影响。一些对库存系统有直接影响并更新库存记录数据的功能是：采购、接收、物料的分发、审计、报废、发运、制造。

5.2 原子模式

我们从定义一个跟踪物品的库存模式—基本库存模式开始。

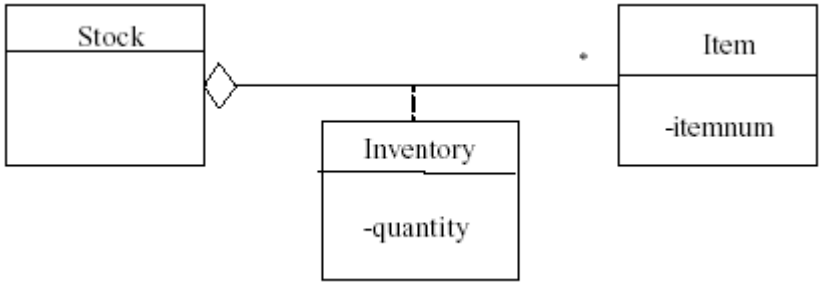


图 1.基本库存模式的类图

库存系统的基本模式如图 1 所示。物品是任何我们想知道它的存在位置和数量的东西。每一个物品属于一个唯一的类型，这个类型提供了一个标识，如：物品号（物品编码，型号）。库存(Inventory)类持有每个物品我们所关注的各种数量。这可以被认为是一个原子模式，如语义分析模式（SAP），它既可以单独存在，又可以作为一个更大模型的一部分。它的动态方面在组装模式中显示。

图 2 显示了物品分布模式的类模型。这是一个描述一系列物品的位置以及在那些位置上的分布的模式。它是另一个原子模式，将和基本库存模式一起组装成仓库管理器模式。为了指示分布状态，我们需要把库存数量分解成多个位置数量。（一个库存物品通常存放在几个不同的位置上）。这要求在库存(Inventory)类和位置(Location)类之间定义多—多的关联关系，并且使用一个关联类－分布(Distribution)来指示物品在位置上的分布。

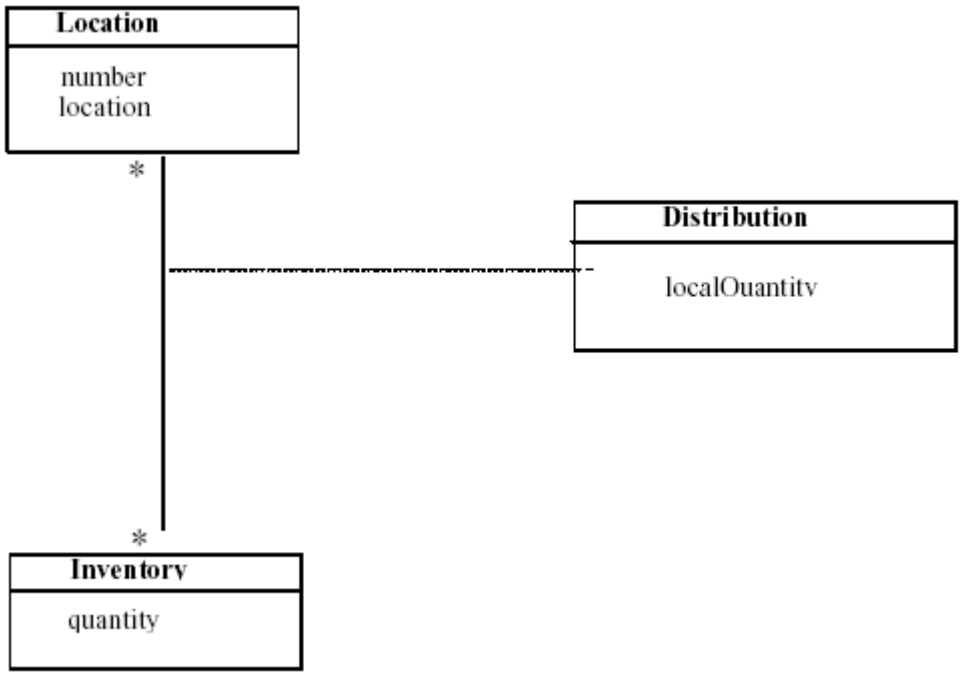


图 2 物品分布模式的类图

5.3 货物管理器的类模型

如下图所示：

图 3 显示了满足需求 5.1 的库存模型。仓库(Stock)和零部件/产品(Component/Product)通过一个聚合关联发生关系。[1] 库存(Inventory)类中的数量(quantities)属性是仓库(Stock)与零部件/产品(Component/Product)类的结合点，不同的链接，有不同的值。这个模型允许设计者定义不同类型的仓库，作为单独的物品集合；例如，零部件仓库，产品仓库。库存的不同类型可以归纳为一个库存(Inventory)类。

[1] 这是一个松散的聚合类型，两个类都可以独立存在。[Booc99]

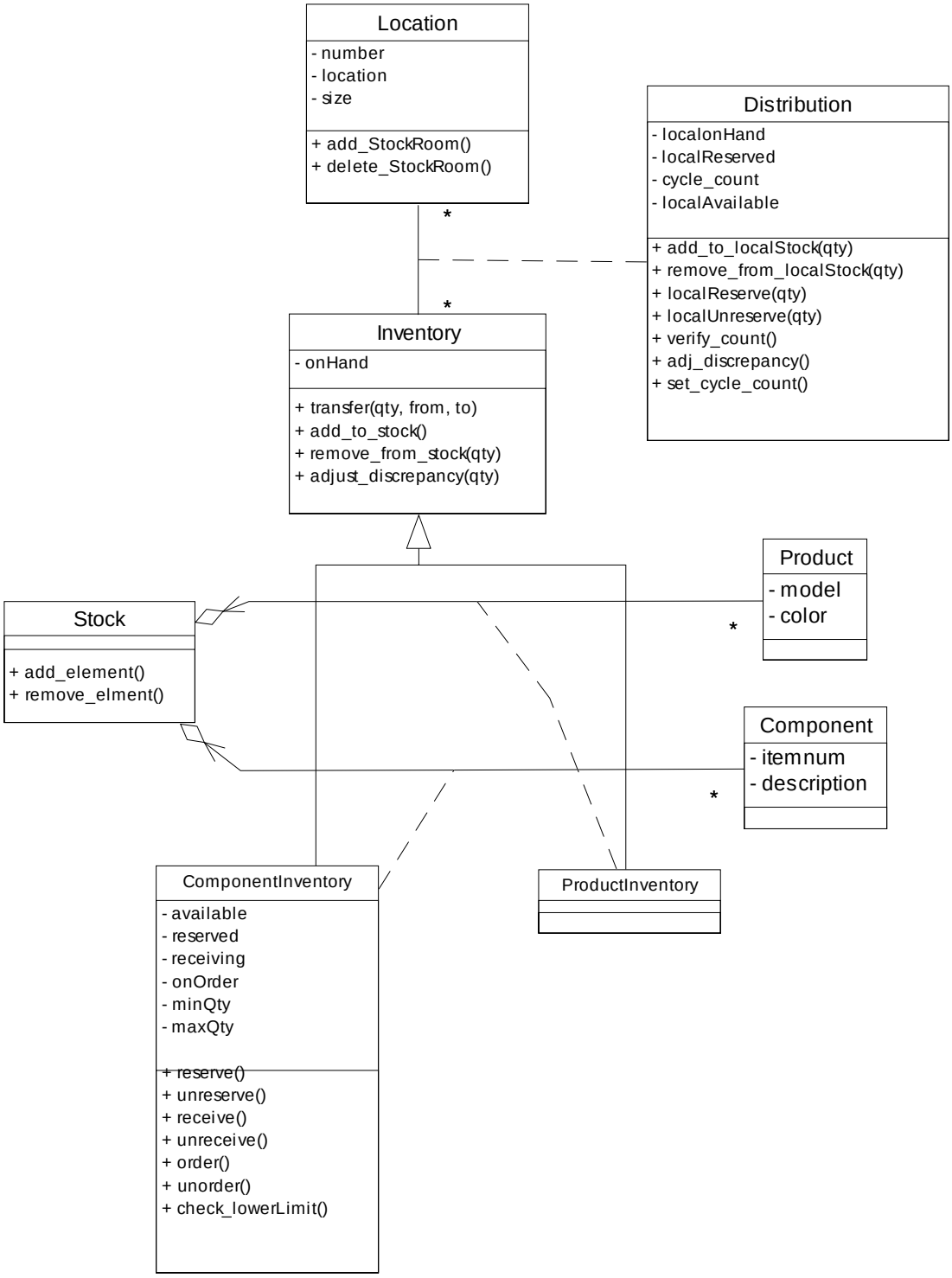


图 3.仓库管理器分析模式的类图

【译者注：原图中关联类 ComponentInventory 和 ProductInventory 的关联目标搞反了。这幅图是译者自己用 Rational Rose 2000 作的。另外，原图中的'/'符号，在 UML 中没有，应该是作者的笔误。】

物品被划分成两个不同的类型：成品和用于制造产品的零部件。其他的划分方法也是可能的。产品(Product)类中的“型号(model)”属性，用来作为一个唯一的标识符，其他属性描述了顾客选择时可能用到的特征，如颜色等。零部件(Component)类具有物品序号，描述，类型等属性，物品序号用来做唯一的标识符。产品(Product)和零部件(Component)通常是多-多的聚合关系（一个型号的产品使用几种类型的零部件，一种类型的零部件可以被用在几种型号的产品中），不过这与库存模型无关，因此在图中没有标明。

物品(Item)类被划分成产品(Product) 和零部件(Component)子类的原因是：在这两个实体的管理中，存在着很多不同。例如，产品是由一些零部件制成的，在制造过程中，库存系统需要跟踪零部件数量的变化。换句话说，零部件库存比产品库存更复杂。另一方面，两者又在一些方面有相同之处，例如，都需要跟踪手头（onHand）数量（合计的总数量）和实际的存放位置。这里可以应用泛化关系，定义库存(Inventory)类为一个父类，含有共同的特征，而产品库存(ProductInventory)和零部件库存(ComponentInventory)类作为子类，含有每一种不同库存类型的独特特征。图 3 的模型中也包含了从动态分析中引申出的操作，这将在下一部分描述。

5.4 动态分析

图 3 中的类模型提供了库存系统的静态视图。这是不够的，我们也需要动态模型来显示库存怎样随时间而变化。在动态模型中，我们定义了为完成通用的操作而应该具有的一般特征。具体的库存系统的其他特征可以从该模型的基本配置中派生。图 4 是动态模型的序列图。

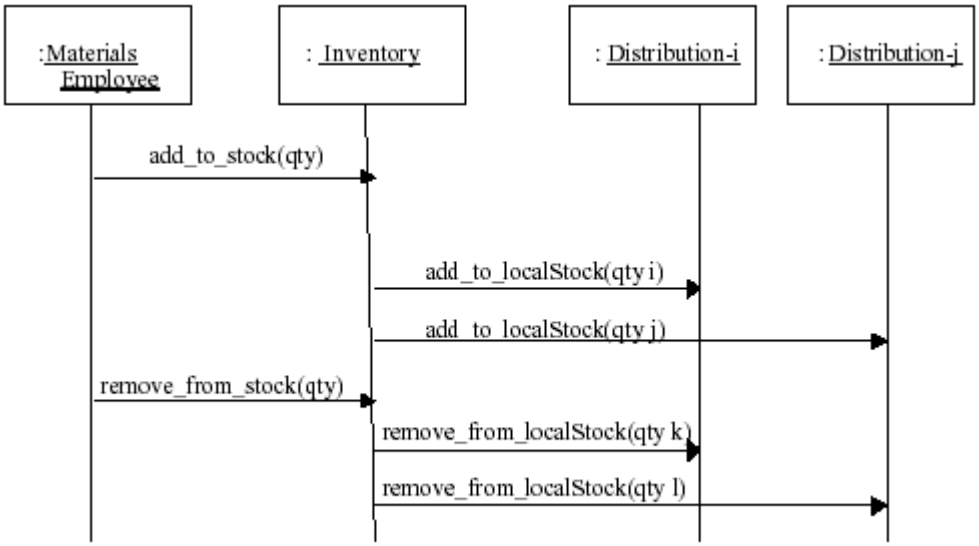


图 4.物品入库、出库的序列图

当零部件或产品入库时，通过调用 add_to_stock 操作，它们的手头数量（onHand）增加。当零部件或产品出库时，通过调用 remove_from_stock，它们的手头（onHand）数量减少。而 add_to_stock 操作根据预定的规则，决定物品放在哪儿。当零部件或产品被放到某个特定的仓库区域时，操作 add_to_localStock 增加局部的手头数量（localonHand）。相似地，remove_to_localStock 减少局部手头数量。

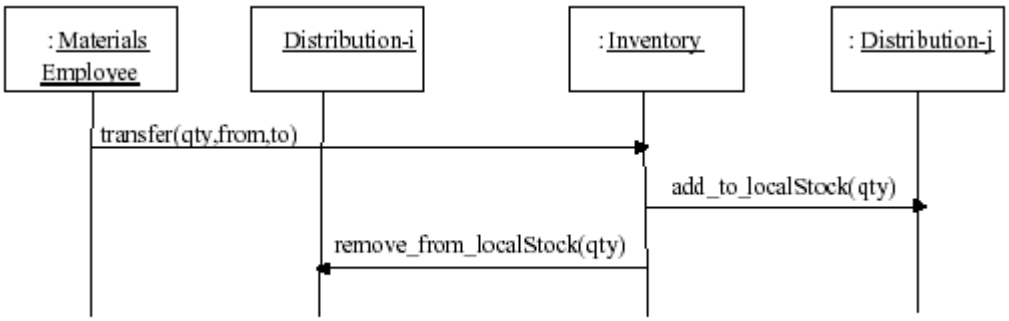


图 5.物品迁移序列图

零部件或产品可以从一个仓库区域迁移到另一个仓库区域，操作 transfer 执行这个动作。因为所有的移动操作既可以应用到零部件（由 Component 类表示），又可以应用到产品（由 Product 类表示）上，所以我们把这个操作放在父类 Inventory 中。当零部件或产品移出某个仓库区域时，remove_from_localStock 操作被用来更新局部手头数量（localonHand）。相反情况下，使用 add_to_localStock 操作。图 5 显示了物品迁移的序列图。

最复杂的库存变化发生在制造过程中（见图 6）。我们首先假设顾客的订单已经被处理，并形成了一张表单。表单中详细指示了在制造某种类型的产品时所需的零部件--什么型号的，以及每种型号的数量；这张表单就是生产指令单。当物料员根据生产单备料时，根据生产指令单上指示的数量，所需零部件保留数量的值增加。当零部件从仓库区域实际被领出时，保留数量和手头数量的值减少了。当制造过程结束时，产品库存中的手头数量增加了。通常，生产指令单从处理到结束，需经历几天时间。备料、领料到制造等阶段使人们知道生产指令单正处于什么状态。图 6 显示了制造过程中的库存变化。

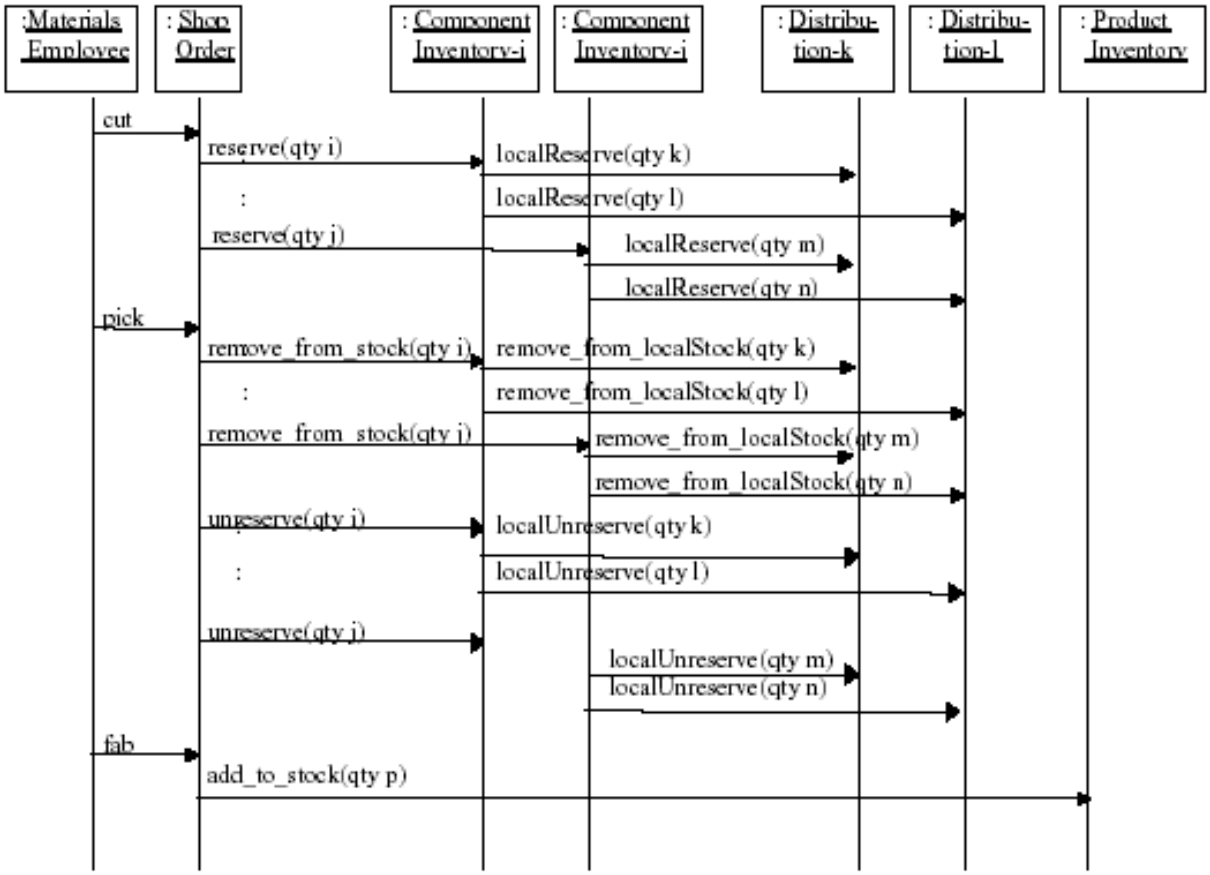


图 6.按照顾客的订单来制造的序列图

6　在图书馆库存管理中应用此模式

为了展示本模式在不同领域的使用，我们把它应用到一个图书馆库存中。如图 7 所示：

现在，作品仓库(Works Stock)类定义图书馆各种材料的仓库。分布(Distribution)类用来指示某个位置上的数量。这里的位置(Location)类可以表示借书处、图书存放架、缩微胶片中心、公共阅读区，等等。这里，我们显示了一个作品仓库；然而，这个仓库或许可以分成几个独立的仓库：书库，视频磁带库，CD 库，等等，因为这些种类的资料的处理方式是不同的。这个系统的用例(Use Case)是诸如借/还书的动作。这些用例(Use Case)的有些序列图与前面给出的序列图相似，如书的位置的迁移；而有些则不同，如顾客订单的处理。

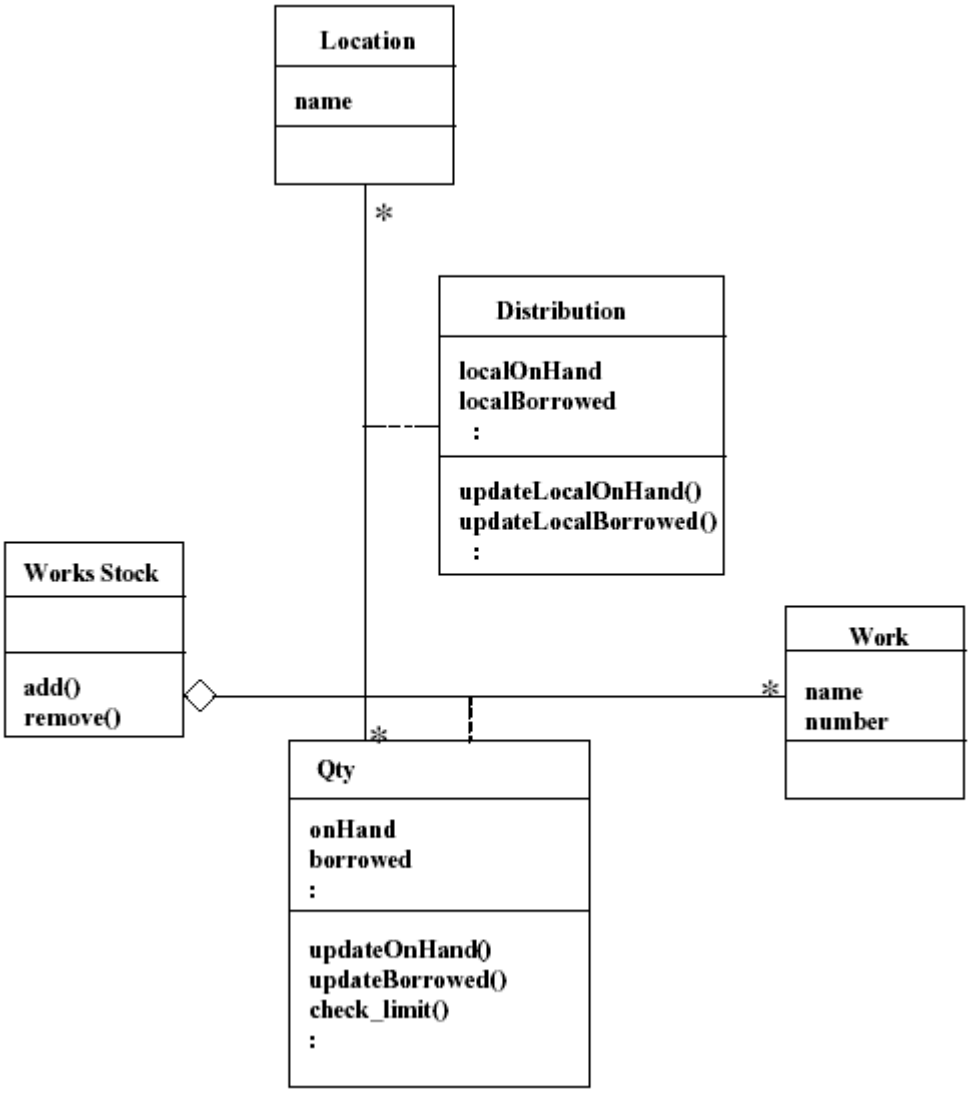


图 7.图书馆库存系统

7 结论

产生的模型满足了以下的约束：

- 该模式可以用于跟踪不同类型仓库中目标物品的数量和它们的分布位置。
- 其他动作产生的影响可以通过调用适当的操作反映出来。如图书馆的例子中指出的那样，对库存产生影响的动作在不同的应用中可以是不同的。
- 尽管该模式是用制造业中的术语描述的，但它仍可以用来表示诸如图书馆、商务活动、或者其他

相似的地方的库存。

- 文档，如顾客订单，和其他一些东西，被认为是与库存系统相互作用的外部系统的组成部分，因此没有在模式中表示出来。

为了使该模式能够在多个领域里应用，我们省略了：

- 物品的细节，如结构和描述。
- 存放位置的细节。
- 描述内部动作的文档，例如，在不同位置间货物的迁移。
- 异常处理，例如，保留数量比实际能获得的数量大，订单取消等。
- 提示低库存水平的报警。
- 历史信息
- 资金流动
- 当库存物品为液体形态时的处理。（度量液体时应该使用不同的方法）
- 这些方面应该通过其他模式来增加，或者是开发一个包含它们的完整框架。

8 已知的使用

一些库存模型的例子是：

- 在为位于美国佛罗里达 Boynton Beach 的摩托罗拉公司寻呼机产品分部的信息技术部开发一个库存原形时，应用了该模型。[Fern97]
- 在 Hay 的库存模型中[Hay96]，资产是些有实际物质形态的东西，(目前)必须存放在一定的场所，如仓库中的某一位置，制造工作中心。更确切的说，一项资产是一个场所中某种资产类型的物质形态。考虑了两种性质的资产：离散的物品和库存。离散的物品可以被单独的信息条所标识（如

序列号，公司标签号【译者注：原文中的 serial member 为笔误，应该为 serial number.】)。库存只用来跟踪大批量的资产。资产的类型又可以分为产品类型、物料类型、和零件/设备类型，这是因为以作者(Hay)的观点，物理形态不同的产品、零件、物料是按不同的方式来跟踪的。总的来说，贺氏模型中包含了我们忽略的一些方面。然而，它的模型没有包含动态的方面，属性，或操作；也没有把仓库（stock）与库存(Inventory)分开。

- Fowler 的文献[Fowl97]中有一章（第六章）写的是关于库存和会计的。他强调会计中的货物和资金的流动。他的模型中的另一个重要的方面是元数据（处于知识层中）和操作的分离。

- Hellenack[Hell97]讨论了作为其他业务模式一部分的库存。她分离了成品库存和原材料库存（与我们的产品[Product]和零部件[Component]类等价），也包含了一些资金流动。她的模型中只包含了静态的方面并且没有物品分布的管理。

9 相关的模式

保留和实体使用模式(Reservation and Use of Entity Pattern)[Fern97]，订单和发运模式(Order and Shipment Pattern)[Fern00b]，在几个应用中与本模式相辅相成，如制造业。[Brag98,Brag99]中的模式应用到了资源控制领域，并且看来会在很多应用中与本模式合作。一个可以使用“仓库管理器”模式的模式是依赖性需求模式(Dependent Demand Pattern)[Hau97]。因为产品和零部件是有类型的，所以类型对象模式(Type Object Pattern)也是相关的。最后，仓库/产品、仓库/零部件的聚合关系，是容器/环境模式(Container/Context Pattern)的一个实例。

10 致谢

Motorola 公司库存系统的原型是由 Michael Lynch, Chin- Shu Lee, Zhiwei Peng,and Mahbub Anwar 构建的， 他们都为本模式中的这些构思做出了贡献。Rosana T.V. Braga，我们的主管，以她富有洞察力的意见，对改进这篇论文的质量起到十分重要的作用。笔者在 PLoP 2000 的工作室也为改进这篇论文提供了有价值的建议。

参考文献

[Booc99] G.Booch, J.Rumbaugh, and I.Jacobson, "*统一标记语言用户指南*", Addison-Wesley 1999.

[Brag98] R.T.V. Braga, F.S.R.Germano, and P.C. Masiero, "资源管理模式的联合使用", *Procs. of PLoP'98,*http://jerry.cs.uiuc.edu/~plop/plop98

[Brag99] R.T.V. Braga, F.S.R.Germano, and P.C. Masiero, "业务资源管理的模式语言", *Procs. of PloP'99,*http://st-www.cs.uiuc.edu/~plop/plop99

[Coa97] P.Coad, "对象模型：策略，模式和应用" (第二版),Yourdon Press, 1997.

[Fern97] E.B. Fernandez , M. Lynch, and C.S. Lee, "库存控制系统的面向对象原型的开发", Report TR-CSE-97-32, Dept. Of Computer Science and Engineering, Florida Atlantic University, April 1997.

[Fern99] E.B.Fernandez and X.Yuan, "保留及实体使用的分析模式", *Procs. of Pattern Languages of Programs Conf.* (PloP'99),http://st-www.cs.uiuc.edu/~plop/plop99

[Fern00a] E.B. Fernandez and X. Yuan, "语义分析模式", *Procs. of the 19th Int. Conf. on Conceptual Modeling (ER2000),* 183-195.

[Fern00b] E. B. Fernandez, X. Yuan, and S. Brey, "产品订单和发运的分析模式", *Procs. of PLoP 2000.*

[Fowl97] M. Fowler, *分析模式 – 可重用的对象模型*, Addison- Wesley,1997.

[Hau97] R. Haugen, "依赖性需求—供应和需求平衡的业务模式", *Procs. of Pattern Languages of Programs Conf.*, PloP97, http://st-www.cs.uiuc.edu/~plop/plop97/Workshops.html

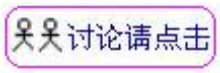
[Hay96] D.C.Hay, 数据模型模式—*思考的常规*, Dorset House Publishing, New York, 1996.

[Hell97] L. J. Hellenack, "面向对象的业务模式", *Object Magazine*, January 1997, pp. 23-30 and 70.

[John98] R. Johnson and B. Woolf, "类型对象", 程序设计的模式语言*3* 中的第四章, Addison-Wesley, 1998.

[Rie96] D. Riehle, "组装设计模式", Procs. of OOPSLA'97, 218-228.

[Salv92] G. Salvendy, 工业工程手册, Wiley- Intersience Pubs. ,1992.



使用模式集成 UML 视图

Alexander Egyed 著，davidqql 译

摘要

模式在系统组合（合成）期间对养成重用可重复设计和体系结构配置的习惯很重要。本论文研究关于模式的知识，它也可用于系统分析检验系统模型的完整性。

为了支持自动分析过程，该工作引入一个视图集成框架。自从每个视图（例如，框图）增加一个额外针对模型的软件系统观点，来自一个视图的信息可能用于验证其它视图的完整性。这种形式的集成要求对视图表明什么以及它们可以共享（或约束）什么信息有更深的理解。因此关于模式在结构和行为上的知识，也是一个用于视图集成自动化的有价值的来源。

介绍

为支持软件产品开发，我们频繁使用通用软件开发模型（和工具），例如统一建模语言(UML)。然而，通常意义的软件开发和特定的软件设计（正是我们工作的主要焦点）要求不仅仅是大多数通用模型所能提供的内容。体系构造是关于：

1) 对实际问题充分建模

2) 解决模型问题并

3) 在现实世界中解释模型方案

这样做的主要重点被置于体系结构的视图（例如框图）内和之间不匹配的鉴别与调和上。我们经常发现这方面的情况，分析和（体系结构的）说明的解释在大多数通用语言中是次重点的。我们构造体系不仅仅是因为我们想建立（创作），而且因为我们要理解。这样，体系构造有许多分析和校验产品模型的概念完整性、一致性和彻底性的工作要完成。

已完全成为事实上 OO 软件开发标准的 UML 的出现，在这个问题上也没有任何例外。本工作阐述在 UML 视图中体系结构不匹配的原因，以及说明模式和集成技术怎么能够以更自动化的方式应用于识别并解决他们。为了做到这一点，本工作讨论视图集成框架，它的主要活动——映射（Mapping）、变换(Transformation)和分化(Differentiation)。

本论文将研究模式的角色，而不是集中于大量的集成技术（它们支持上述活动）。这样，我们将研究模式的知识怎样有助于保证软件系统模型一致性。通过那样，我们按以往很少使用的方式利用模式：我们用模式用作系统分析，而不是将模式用作构建材料作为系统成分。

视图和模型

在软件开发中，我们利用模型和视图处理软件系统的复杂性。在这里，模型是指视图的集合或者视图可以看作模型的一个方面（或视点）。IEEE 标准（草案）1471[AT&T1993]将视图归结于“提出一个或多个系统利益关联者（Stakeholder）的利害关系”。对于利益关联者，我们定义为分享系统注意或兴趣个体或组（例如，开发者，用户，消费者等等）。应用于我们的语境，视图是模型的片段，它也要细小到我们能够理解，但是也包含关于特定关系的关联信息。在 UML 中，视图本质上是图形的，且往往通过框图来实现。视图（例如类或序列图）服务于下列意图：

- 抽象并简化模型
- 使得不同的利益关联者协调工作
- 为不同解释进行补充（不同观众/利益相关者）
- 提取关于特定关联的相关信息

因此，将会用到什么类型的视图以及什么时候用到它们是强烈依赖于哪个人正在使用和需要完成的相关任务。然而，视图并不是软件开发的银弹，因为它们具体表达基本问题；它们内部及它们之间表现出对等数量的建模元素冗余。

要给出一个简单例子，考虑我们有个设计（例如按照 UML 类图的形式）的软件开发案例和产品实现（例如，C++代码）。类图和代码表述不同的视图，用不同的方法表达相同或类似的信息。虽然，代码可以从设计中自动产生，这种逼近是有限的，还必须多次加入一些信息。更糟糕的是，现在这些冗余的信息片断必须保持一致——后者大多是手工活动。这样，无论什么时候设计变更了，代码就会变得不一致（反之亦然），我们要应用一些视图调和活动找到产生的不一致并一再保证模型概念的完整性。

视图不匹配和冗余

既然视图是我们处理复杂性唯一有效手段，我们不能指望用某些较少冗余的事物来替代它们。我们需要视图在任何给定的时间对软件开发者不得不处理的信息总数进行分解。“这不是带来复杂性的细节数量本身，而是我们不得不同时了解的细节的数量。”[Siegfried 1996]

然而，冗余性是一个必需的不幸。这暗示我们需要某种鉴别和解决视图之间不匹配的自动化活动的方法。这样，我们所需要就是一些集成和分析视图的框架形体。有趣的是，视图不匹配问题可能的逼近方法是基于它特有的问题——冗余性。我们利用一套视图之间的冗余性意味着一个视图包含关于其它视图并可用作约束该视图的信息。这样，我们使用冗余信息来检验视图之间的一致性和完整性。

例如，如果我们使用一些体系结构模式的形态来构造系统（例如，分层风格），那么设计必须反映体系结构对立的约束。这意味着如果体系结构定义了三层结构，那么体系结构隐含地定义了处理中第一层不使用第二层而与第三层直接对话是不允许的。如果后来系统使用 UML 设计（例如，使用类和序列图），那么设计元素之内的调用依赖要求与上述的体系结构约束一致。我们将在后面说明一个例子。

UML视图描述 vs 集成

启用视图集成来确定和调和视图要求两个层次的集成——符号集成和语义集成。对于符号集成，意思是模型需要完整表达视图的能力。语义集成通过定义什么信息可以交换以及怎样交换作更进一步精炼。只有在什么和怎样在确立之后，不一致性才可以确定。

统一建模语言（UML），像其他通用软件系统开发模型一样，只是不能很满足上述语义集成。UML 提供一个模型用于表达不同框图的视图来处理类、对象、活动、状态、包及其它（参看图 1）。然而，UML 不擅长集成他们。在 UML 视图之间的关联和依赖关系很少被捕获。如果后者没有完成，模型仅仅是松散耦合的或完全无关联的视图集。虽然 UML 及其元模型在细节上定义了单个视图的符号和语义特征，视图内关联的获取仍然是不够的（在视图之间只存在一些微弱形式的依赖关系，例如类和对象）。

图 1 也表明问题的另一个方面——那就是扩展 UML 以表达新的和外部概念（例如，风格和模式）。例如，怎样才能在 UML 中使用更高级的模式例如分层的体系结构模式[1]或代理设计模式？在 UML 中，我们需要再次区分仅仅表述模式和完整集成它们。

[1] 注意我们如[Bushman et al 1996]所作，可互换地使用术语模式和风格。

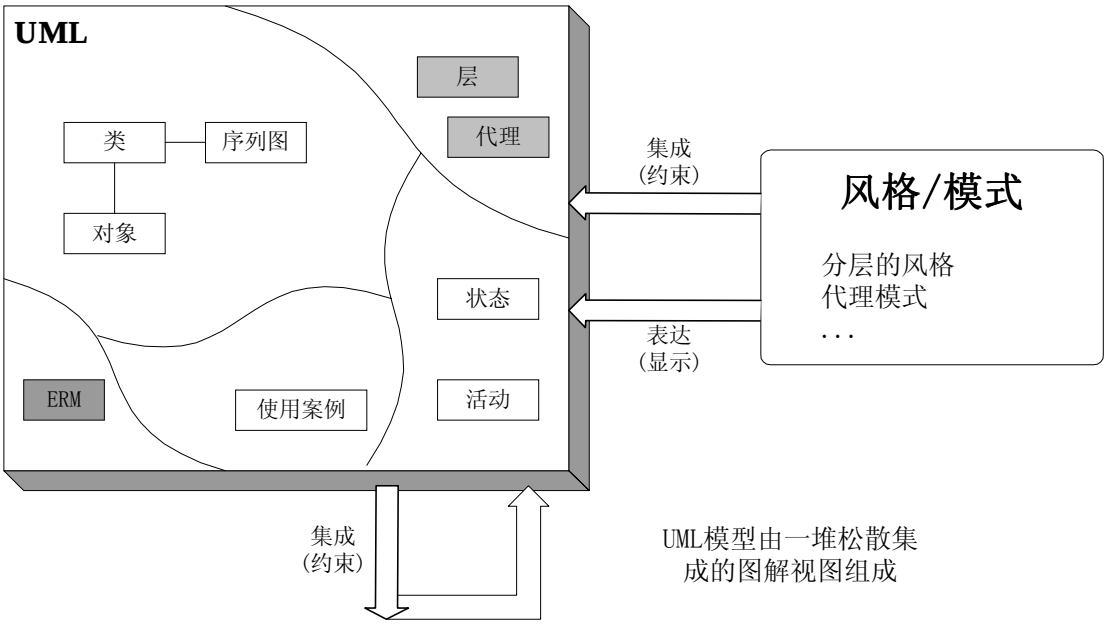


图 1 UML 中表示法 vs. 集成

视图集成框架

视图集成的主要的障碍是缺乏完好定义的（工程的）模型基础。视图经常使用迥然不同的表示信息方法，而这使得确定它们在哪里和怎样出现重叠非常困难。这样，组合和比较视图的任务经常是手工的而且潜伏着错误的。集成框架的目标是要补偿鉴别和解决体系结构不匹配自动化辅助手段的不足。

如前面简短的叙述，这样做的时候，我们的框架需要处理什么信息和怎样进行交换。我们在那里写到什么信息可被交换以及它怎么能交换的判断需求。在我们的视图集成框架中，我们提到**映射**和**变换**这两个活动。我们也说只有在这些活动定义之后，才能够尝试鉴别不一致性。后者我们称之为分化（参看图 2）。

图 2 在高层次式样上描写我们的视图集成框架。在那里，系统模型用于表达软件系统的知识基础。软件开发者使用视图增加新的数据到系统模型并且复审现有数据（视图综合）。对于 UML，系统模型可能被看作通过 UML 设计工具（例如 Rational Rose）完成的 UML 模型和视图综合。

系统模型和视图综合两者交互影响就是视图分析。一旦加入新的信息，它就可以针对系统模型进行验证以保证其概念完整性。图 2 表示视图分析可以怎样进一步细分为其上述三个主要活动：

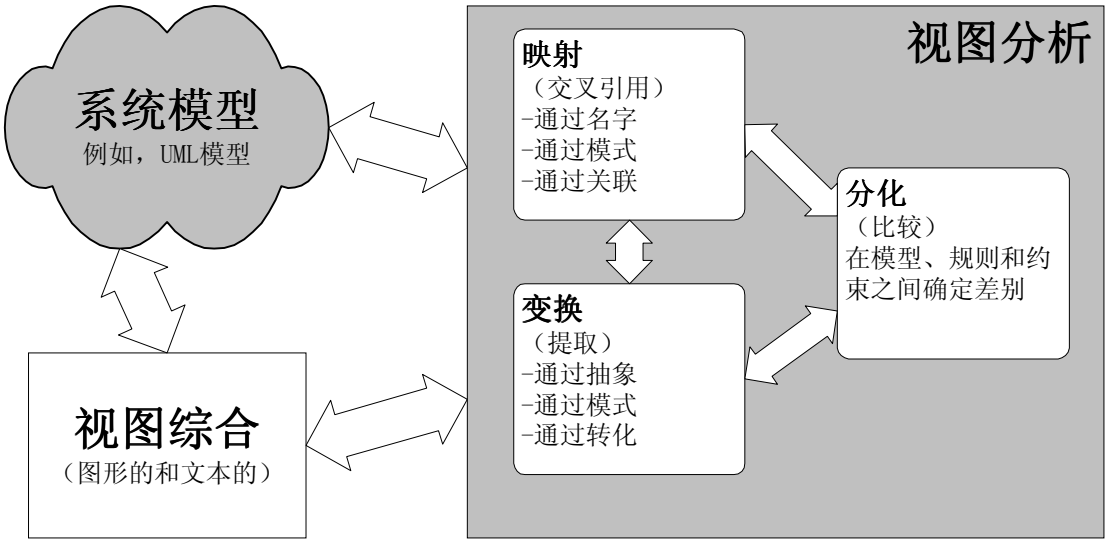


图 2 视图集成活动

- **映射**：通过使用命名字典、跟踪和跟踪仿真（例如相同物理类和方法的使用）以及某种关联/模式形式（例如公共接口）来确定相关的信息片。
- **变换**：在视图中操作模型元素以便它们（或它们的片段）可以在其他视图中共享（或在系统模型自身中表述）。例如，我们可以使用抽象技术泛化一个详细的框图，我们可使用视图转化在不同类型的视图之间交换信息，或者我们可以按不同的风格重新排列模型元素（或片段）产生新的视角（例如拼结和分割）。
- **分化**：详细研究系统模型鉴别系统模型中（潜在的）不匹配。（潜在的）不匹配按规则和约束的形式讨论。此外，不匹配解决规则可与将要怎样解决它们可选方法的不匹配标识规则相关联。分化是强依赖于变换和映射的。

然而，必须注明的是，那些活动不是相互正交的。显然地，我们只有在知道模型元素的正确映射时才能做出有用的变换。这种依赖关系反之也是成立的。通过视图变换导出的信息可以澄清许多映射中的二义性。这样，一个视图可以用于澄清其它视图中的二义性。

此外，如图 2 所示，视图集成不只局限于模式，然而，模式对视图集成构成了非常重要的基础。我们将在后续章节中讨论这种面向模式的视图集成方向。其他非模式相关的视图集成方向在[Egyed1999a]和[Egyed1999b]中论述。上述框架通常在 UML 之外也适用。

产品订货系统案例

我们将使用一个简单的产品订货系统贯穿全文进行指导，该系统被分成两个主要的子系统——*订单获取子系统和订单处理子系统*。第一个子系统通过销售代表从消费者获取订单和付款信息。后一个子系统获取仓库管理员对产品订单队列的处理。产品订货系统合成两个 COTS（Commercial off the Shelf，已下架的商品化）产品：存货系统处理产品存货，而订单仓库作为数据库（后者既用于产品订货系统又用于存货系统）。

表格 1 表示我们的系统体系结构使用分层模式（或分层风格）进行设计。该体系结构模式将在后面由设计模式和其他设计特征补足。

表格 1 讨论产品订货系统的分层体系结构模式

产品订货系统
- 用户界面（订单获取界面，订单处理用户界面，存货用户界面）
- 订单框架（消费者，付款，订单，订单行，记录器）
- 存货系统
- 网络（LAN）
- 订单仓库

模式怎样有助于视图集成？

在文章展开论述时，我们将揭示关于产品订货系统更多的细节。虽然，由于篇幅的限制，我们既不能在此表述整个系统，也不能阐述所有的集成技术。相反，如前面提到的，我们将集中于视图集成时模式承担的角色上。对应于图 2 中简述的三个集成活动，模式支持如下活动（下节将说明例子）：

映射

模式支持不同抽象层次的视图之间的映射（交叉引用）。例如，一个高层框图指出使用一个已知后来在低层次框图中实现的模式。这样，这类模式存在于低层次框图的知识以及该模式粗略了解（如在[Gamma et al. 1994]和[Buschman et al 1996]中定义）的知识有助于在低层框图中自动鉴别。

模式也支持不同类型视图的映射。例如，模式描述经常指定其结构和它们的动态行为。那么，可以在我们的模型中使用这些知识来交叉引用结构化的和动态的信息。

变换

模式应用于变换的方法与它们在映射中的用途类似。对于变换，我们可以把它们用作抽象和转化。对于抽象，我们意指简化视图的处理。例如，如果我们想知道高层视图和低层视图是否一致，那么我们需要精炼高层视图或者抽象低层视图以使直接比较成为可能。前者不能自动进行，而后者可以。为了抽象视图，需要确定相关的片段，然后，它们被更加简单的事物替换。对此，模式是完美的原始资料，因为它们提供哪些片段属于一起的知识。我们可以使用模式知识抽象低层模式到高层模式（或者对等的单个部件）。

模式也可以如我们在映射中讨论的那样用于静态与动态结构之间的转化。既然模式经常用两种式样描述，我们可以通过结构推导行为（反之亦然）。这样，利用模式我们也可以转化视图。

分化

如图 2 中简述，分化强依赖于映射和变换。这意味着要在视图中找出不一致性，我们需要确定其建模元素的关系，并且需要找出从一个视图到另一个视图变换信息的方法。没有前一个步骤我们不能知道要比较什么信息，没有后一步，我们不知道怎样比较。一旦上述步骤完成，我们可以使用两个主要技术比较视图：

- （图形）比较：依靠变换的视图和原始视图的对比进行比较。该技术暗示：一个视图可以保留充分完整的风格变换为另一个视图。
- 约束和规则检查：我们频繁地发现视图不能广泛的变换为另一个视图，但它的少部分和片段可以。在这种情况下，我们可以使用规则和约束来论述和分析这些片段。

对于**分化**，设计模式并非直接有用，然而，我们通过映射和变换收集到的有关视图的信息却可以。我们已经简短地讨论模式怎样有助于映射和变换视图。在这些例子中，比较视图是直接的，因为映射和变换启用直接（图形）比较。无论如何，模式在约束和规则检查中也非常有用。例如，我们在表格 1 中介绍的分层模式定义了自然的约束：用户界面只允许与订单框架对话，订单框架依次只能与存货系统对话等等。该体系结构的模式影响了设计的结构和它的行为。

在产品订货系统中使用模式

本节通过说明模式可以怎样在我们的产品订货系统语境中应用于集成活动的例子补充上述论述。

分化

图 3 使用 UML 包说明我们系统的高层设计视图。该图显示主要订单系统部件（或子系统）的交互。关于分层体系结构的知识现在可以帮助我们在表格 1 中的体系结构和图 3 的设计之间自动确定不匹配。表格 2 总结两个视图对立的约束。

体系结构视图约束从表格 1 导出。它们定义我们系统层次之间的调用依赖关系（例如，用户界面依赖于订单框架）。图 3 是设计视图约束的基础。该图说明一个含有一套包以及它们之间调用依赖的 UML 包图（包和依赖的语义在[Booch-Jacobson-Rambaugh 1997]中定义）。

表格 2 体系结构上和设计视图对立的约束

体系结构的视图约束

体系结构[用户界面取决于订单框架];

体系结构[订单框架取决于存货系统];

体系结构[存货系统取决于网络];

体系结构[网络取决于订单仓库];

设计视图约束

设计[订单获取 UI 取决于订单框架];

设计[订单处理 UI 取决于订单框架];

设计[存货 UI 取决于存货系统];

设计[订单框架取决于存货系统];

设计[订单框架取决于网络];

设计[存货系统取决于网络];

设计[网络取决于订单仓库];

映射

设计[订单获取 UI]　　映射到 体系结构[用户界面]

设计[订单处理 UI]	映射到 体系结构[用户界面]
设计[存货 UI]	映射到 体系结构[用户界面]
设计[订单框架]	映射到 体系结构[订单框架]
设计[存货系统]	映射到 体系结构[存货系统]
设计[网络]	映射到 体系结构[网络]
设计[订单仓库]	映射到 体系结构[订单仓库]

完整性规则

对于所有体系结构的视图约束存在设计视图约束；

例如：

体系结构[用户界面取决于订单框架] =>

存在：设计[订单获取 UI 取决于订单框架] 或者

设计[订单处理 UI 取决于订单框架] 或者

设计[存货 UI 取决于订单框架]

一致性规则

对于所以设计视图约束存在体系结构约束；

例子：

设计[订单处理 UI 取决于订单框架] =>

存在：体系结构[用户界面取决于订单框架]；

表格 3 描述产品订货系统的分层体系结构模式

产品订货系统	
用户界面（订单获取 UI,订单处理 UI,存货 UI）	
订单框架	存货系统
网络（LAN）	
订单仓库	

建立这些约束是变换的责任，并且在这种情形下能够自动完成。例如，利用我们在表格 1 中的分层

模式的知识我们可以自动导出层间的调用依赖关系。优点是模式的语义只需要定义一次并且可以在以后重用。视图的语义和符号也可以看作模式。这样，图 3 的包图带有一套预定义的相关约束。一旦定义，就可以对包图的不同实例导出设计约束。

表格 2 的映射部分定义了体系结构视图（表格 1）和设计视图（图 3）部件之间的关系。在这个例子中，用于映射的模式使用不是那么明显。我们将在后面讨论它们对于映射的使用。

表格 2 中最后两个部分是部分的分化活动。在那里，定义了两类规则，一个用于一致性，另一个用于完整性。如果体系结构定义的一些部件或连接器没有反映在设计中，就可能显示潜在的不完整。在另一方面，如果设计与体系结构抵触，那么这可能指出潜在的不一致性。另外，对于每套我们比较的视图，规则只需要定义一次；那些规则然后可以重用。在体系结构和设计实现之间确定不匹配现在是评估规则和约束的事情。这样揭示了没有完整性方面的不匹配情况，然而，有些不一致性方面的不匹配：

1) 存货 UI 部件对于存货系统的依赖与分层体系冲突，用户界面不允许不经过订单框架直接与存货系统进行交流。

2) 类似地，订单系统要求使用存货系统访问网络。

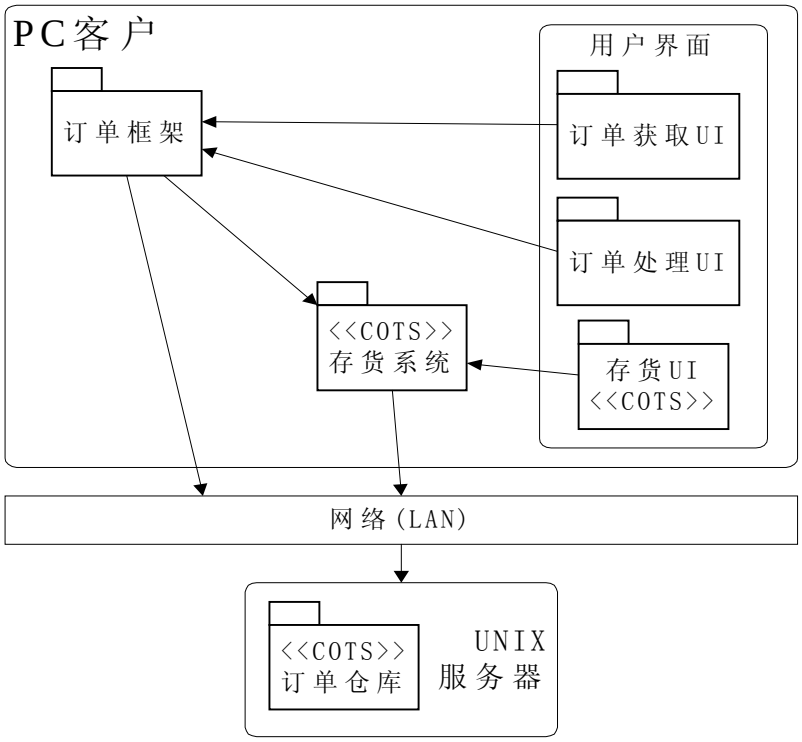


图 3 UML 中的高层设计（包图和依赖）

确定这些不匹配并没有对他们的起因给出任何反馈。例如，是体系结构还是设计错误？表格 3 说明

通过提升存货系统到订单框架同一个级别来解决上述所有不匹配而不会引入新的不匹配的可能方法。我们不相信实际的不匹配解决方法将彻底地自动完成。这种方法与先前的自我纠错源代码编译器的尝试相同，这种努力最后因为所包含的社会和技术的复杂性而失败了。可是，我们相信提供给设计者不仅仅使用（潜在的）不匹配，而且用关于怎样在处理不匹配以及理解它们这两者高度有利的情况下解决它们。此外，它可能对发现处理具有更好选择情形的技术非常有好处。我们将在[Egyed 1996]中更详细地讨论这种情形。

映射

图 4 表明我们系统的另一个视图，图 3 的一个精炼。另外，该视图可以对修订的体系结构和高层设计都能进行校验，然而，并没有发现不匹配。该视图用另外的设计模式例如模板[2]（Template）模式（通过<<Template>>构造型指定）和代理[3]（Proxy）模式（通过《Proxy》构造型指定）。我们将使用它们进一步探究模式在视图集成中的价值。

[2] 对共同使用的模板隐藏源模板和类元
[3] 为其他对象提供一个代理限制对它的访问[Gamma et al 1994]

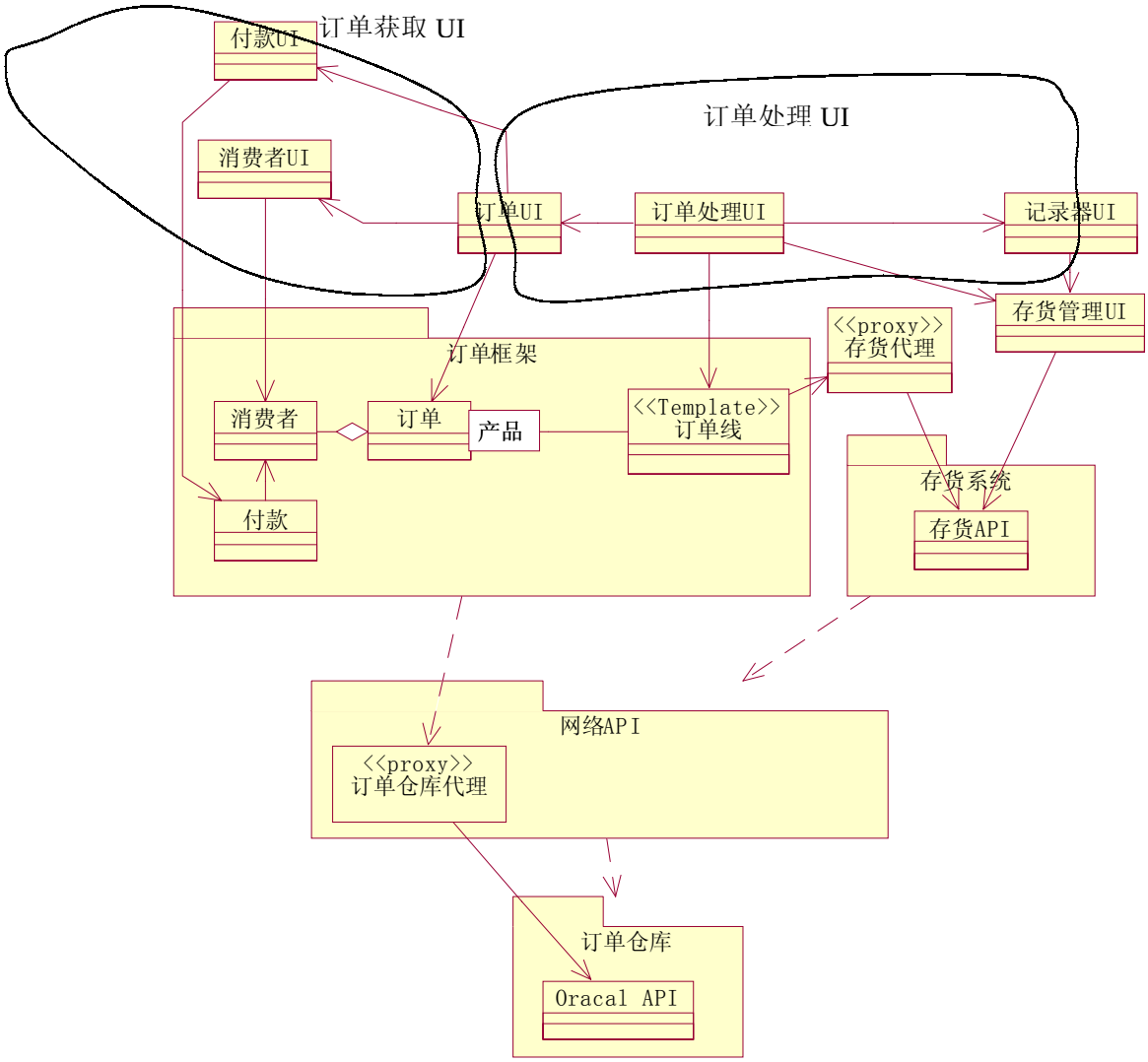


图 4 高层设计视图使用 UML 类和包的精炼

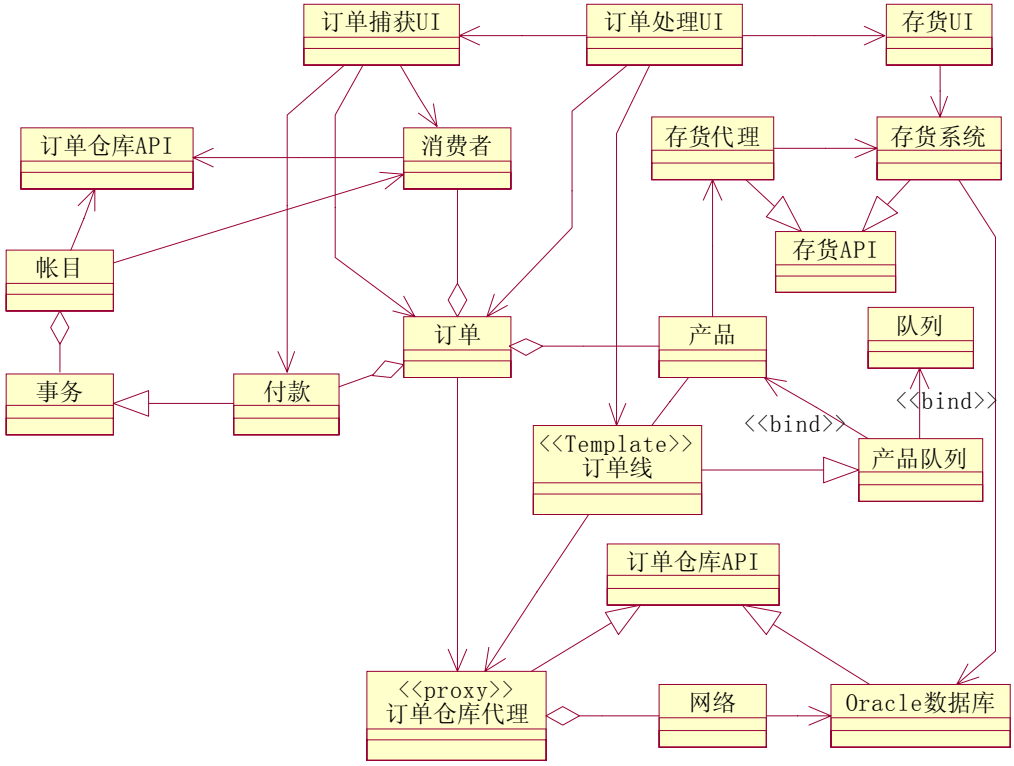


图 5 低层设计视图使用 UML 类图

图 5 描写了对应的低层实现，它不只是解决用于图 4 中的模式，而且也精炼其它的一些建模元素。顶部三个类对应用户界面层，存货系统可以通过存货代理来访问，仓库可以类似地通过仓库代理来访问。要找出该视图是否和先前的视图一致，我们可以运用几个集成技术。

首先，我们需要找出哪些模型元素互相对应（映射）。这有几种技术可以应用，例如名字的相似性等等。可是，在该例子中模式的广泛使用使我们能够开发关于它们用于映射的知识。通过图 4 中的高层设计我们明白几个事实：

- 模板模式用于订单行（OrderLine）
- 代理模式用于桥接订单行（OrderLine）（产品）与存货系统
- 代理模式用于使用 Oracle 数据库桥接订单框架子系统
- 接口特征（例如，消费者类与订单、付款、和消费者 UI 接口）

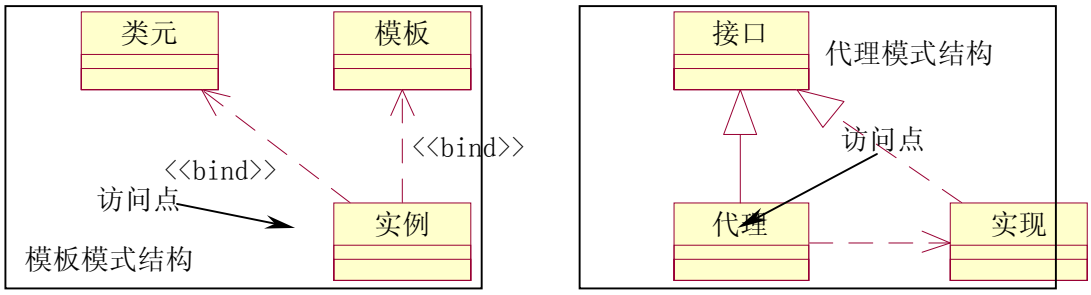


图 6 结构化模式知识（改编自[Buschman et al. 1996]）

虽然从技术上讲，最后一项并不是一个清楚定义的模式，然而它构成类配置的知识——这样，接口特征可以看作模式，即使它们大部分只是领域模式。关于领域的模式知识如同预定义的模式（参看图 6）一样使我们现在可以推论建模信息的关系。映射图 4 和图 5 的任务被精简为使用如图 6 所论述的预定义结构化模式知识来确定上述模式和（接口）特征的位置。

用图 6 作为指导，我们可以容易地确定模板模式（产品，队列，以及产品队列对应于订单行（OrderLine））的对应。虽然，它也可以容易找到代理模式，怎样能够区分它们却不够清楚。注意到目标是使得计算机自动鉴别它们。要在后来做到这一点，我们可以使用上面讨论的接口特征（模式）。想法是，至少存在一些映射或者能够通过名称相似性来建立。使用该额外信息它可能自动从 Oracle 模式区分存货（Inventory）模式。

不幸的是，通过模式的映射仍然比上述例子可说明的要更困难一些。我们作了简化的假定，就是模式的结构和行为是静态的。虽然，通常模式不是那些精确定义的，并且我们需要它们更一般的描述。图 5 表明这样一个例子，仓库代理模式看上去不象定义的那样。然而，既然网络是部分代理类，它就可以合并到代理类中，并且代理类继承它所有的依赖关系（下一节的 Rose/Architect 将表达一个模型来这样做）。

该映射技术的另一个问题是模式，在识别的时候，有时可能只是粗糙地指出它们的位置。例如，对应于队列，产品，和产品队列，订单行队列（OrderLine Queue）可以在低层框图中找到。虽然这也是正确的，但是它丢失了同时在低层框图中表示的组成订单行（OrderLine）。

变换

模式和抽象

单独的介于图 4 中的高层视图以及图 5 的低层视图之间的映射不足以确定不匹配。例如，在图 4 可以看到付款是消费者的一部分，然而在图 5 中这种关系更加复杂。为校验两个图是按相同的方法讨论关系，

我们可以使用 Rose/Architect 概念。

Rose/Architect [Egyed-Kruchten 1999]认为模式分组为三类，并且使用及物关系替换为更简单的模式。在类图中，及物关系论述那些不直接连结的类之间的关系。然而一个关系可以通过其他类（例如 helper 类）存在，它在它们之间形成了一个桥（例如，假设上述例子中付款和消费者并不直接相连，但是它仍然通过 helper（帮助者）类事务（transaction）和账目（account）类给出关系）。这样，如果发现某些公式，它们可以按有效精度导出及物关系，那么，可以按工具形式提供简化和抽象类图方面的自动支持。这将允许设计者通过删除“帮助者类”从现有的、更细节化的模型中抽象出重要的类，这样将使得它们更进一步描写和分析类之间中间关系，即使这些类分散在整个模型的不同位置（例如，不同的框图，或者在不同的包和名字空间中）。

RA 提供这种机制而[Egyed-Kruchten 1999]详细地讨论了这种技术。当前，RA 模型由大概 80 条抽象规则组成。例如规则 4，论述了一个类被第二个类泛化（继承的反义词）并且父类是第三个类的聚集（部分）（参看图 7）的情形。这种三类模式现在可以删除中间类来简化并创建一个从第一个类到第三个类的及物关系（在该例子中的一个聚集）。下面的 RA 模型讨论这些规则以及必须怎样应用它们来产出有效的结果。

图 7 表明我们的低层设计视图（图 5）中的付款给消费者关系案例的 RA 精炼步骤。在应用两个规则（分别是规则 4 和规则 17）之后我们得到一个具有两个类以及它们之间依赖关系的简化模式。如果这也可以对其他类以及在映射小节中讨论的模式[4]进行处理，我们就可以自动产生更高层次的类图（图 8）。产生的这个抽象视图现在就可以与我们在图 4 中描写的原始的更高层次类图直接进行比较了。这样，通过模式的使用我们可以转换视图以便它以非常类似其它视图的方式表达信息。比较视图现在可以简单地使用一些图形比较算法（也可参看上面所说的分化）的形式来完成。图 8 也显示了可能的不一致性。例如，从付款到订单的聚集丢失了，存货系统对 Oracle 数据库的调用没有使用网络部件，以及一些链接是不同的（例如，使用关联链接而不是依赖链接）。在变换（抽象）之后，这些不匹配可以容易地识别。

必须注意的是抽象没有彻底地自动完成。虽然，模式有助于在某些建模信息之间确定映射和变换，它们不能完全自动化该过程。因此，需要一些手工辅助来导出图 8。而且，RA 工具当前只能对付如图 7 讨论的简单的三类模式而不能处理图 6 中讨论的更加高级的设计模式。这样，由于该作业的缘故，抽象设计模式（例如代理）由手工完成。然而，一旦更加复杂的设计模式概念体现到 RA 中，这种抽象可以是自动的。对此，设计模式需要按照它们的输入（源）和目的以及它们的访问点进行讨论。

[4] 注意：我们可以用高层代理简化较低层次设计模式

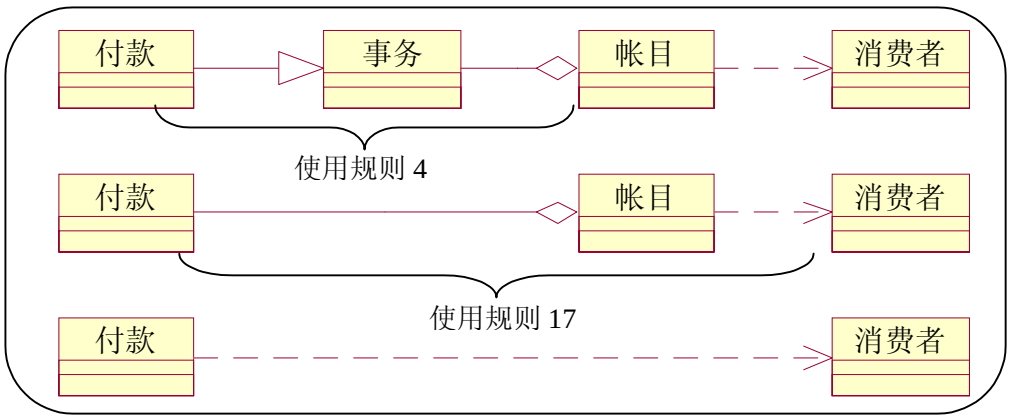


图 7 通过 Rose/Architect 的模式抽象

当前，另外一个 RA 缺点是它只能用于抽象类图信息。虽然该技术也可以扩展到抽象其它类型的视图，但是当我们想要比较不同类型视图的建模元素时它仍然对我们没有帮助。下一小节将论述这种情况。

模式和转化

鉴于抽象处理简化视图的情形，转化处理在不同类型视图之间共享建模信息的情况。例如，上述讨论中，我们广泛使用类和类图，它们按结构化形式表达信息。既然结构视图在表示系统行为通常是不足的，那么行为视图例如序列图就要用来填补这个缺口。

例如，再考虑表格 1 中体系结构分层模式的约束（与高层设计视图的约束一起）。在那里体系结构的约束决没有覆盖全部分层体系结构的行为。一个分层的体系结构不只是限制哪些层允许互相对话，它也限制交互的方向。虽然结构框图比如类图可以描写某些方向性的依赖，但是它可能忽略其它的方面。

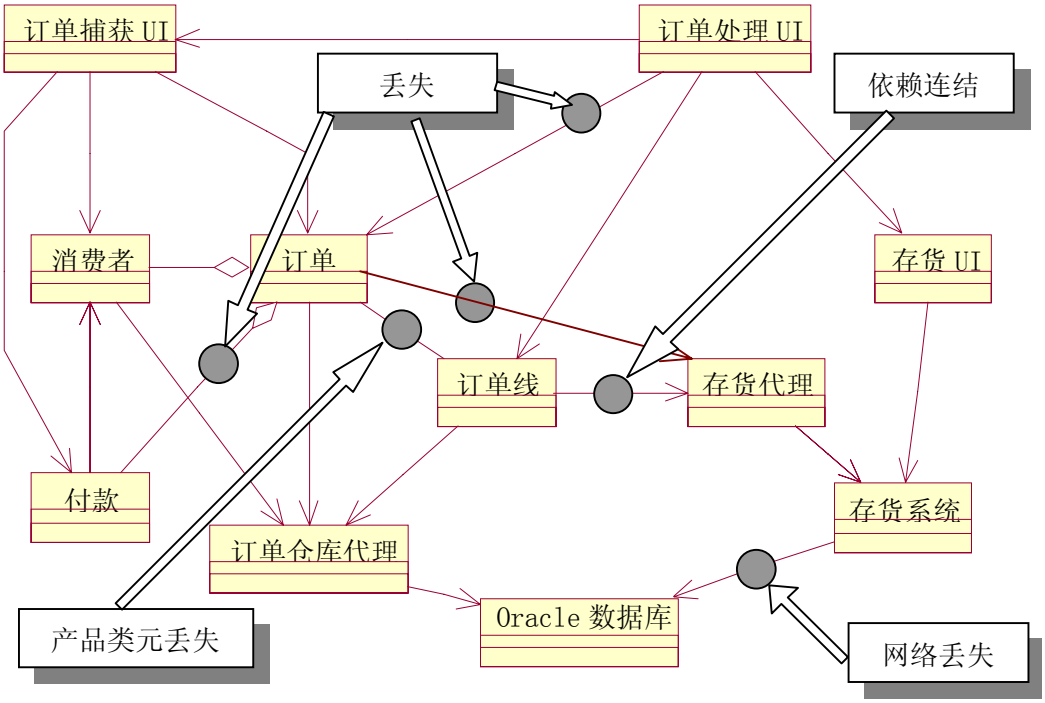


图 8 使用 Rose/Architect 和设计模式抽象的类图

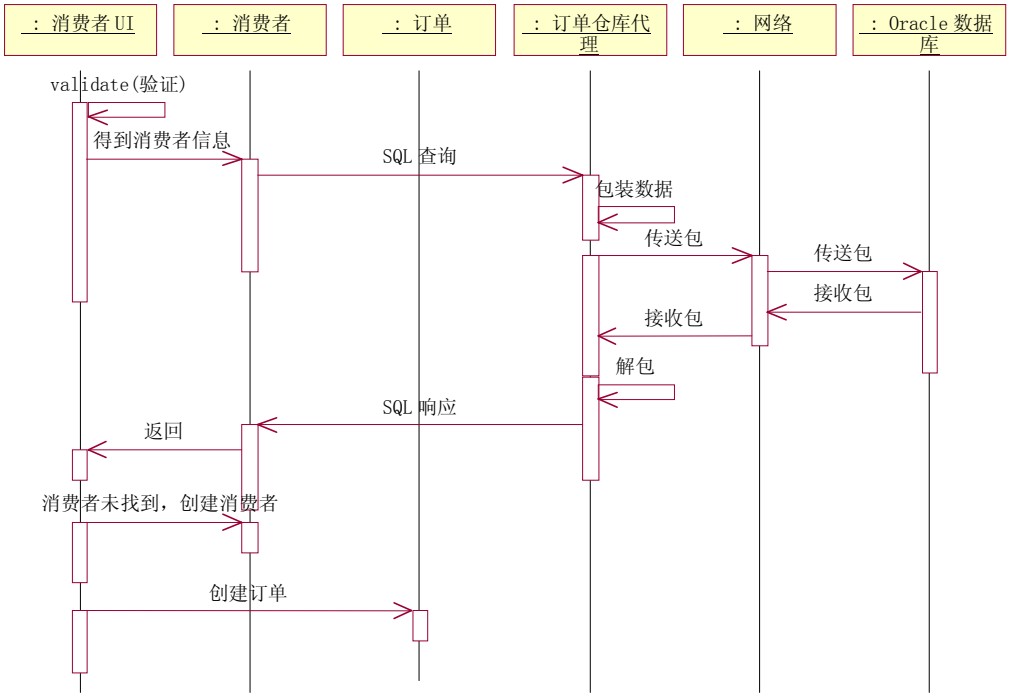


图 9 关于新订单创建的序列图（对应于低层次框图）

图 9 显示了一个更加复杂的行为案例。一个 UML 序列图在这里用于描述创建一个新的消费者订单的可能场景。在某些用户输入被校验之后，它检验消费者是否存在。如果不存在，新的消费者被创建，在此

之后，建立新的订单。该场景描写低层次设计视图（图 5）某些行为方面。照此我们可以使用图 5 自动检验类（或对象）之间的调用依赖，在这种情况下没有揭示不匹配情况。该序列图也符合我们的体系结构，因为所有的层次约束都观察到了（两者都可以自动检验）。既然按照组件的行为结构视图具有高度二义性，我们可以再次利用我们的模式知识精炼行为的信息。

图 9 利用订单仓库代理，网络，和 Oracle 数据库以及在映射中我们发现的那些对应代理设计模式的类。如在[Buschman et al 1996]中所发现的那样，图 10 显示代理模式结构的和行为的定义。效应上，行为定义是结构的一个转化。这样，我们可以使用该知识来检验图 9 的正确性。

因为在图 10 中的代理定义和图 9 中代理实例化之间的抽象级别并不相同，我们需要先抽象图 9。基本上，我们可以使用 Rose/Architect 概念通过合并网络到订单仓库代理中来抽象网络（这是有效的，因为网络是代理的一部分）。此后，定义和实例化之间的直接比较是可能的。在该案例中没有发现不匹配。

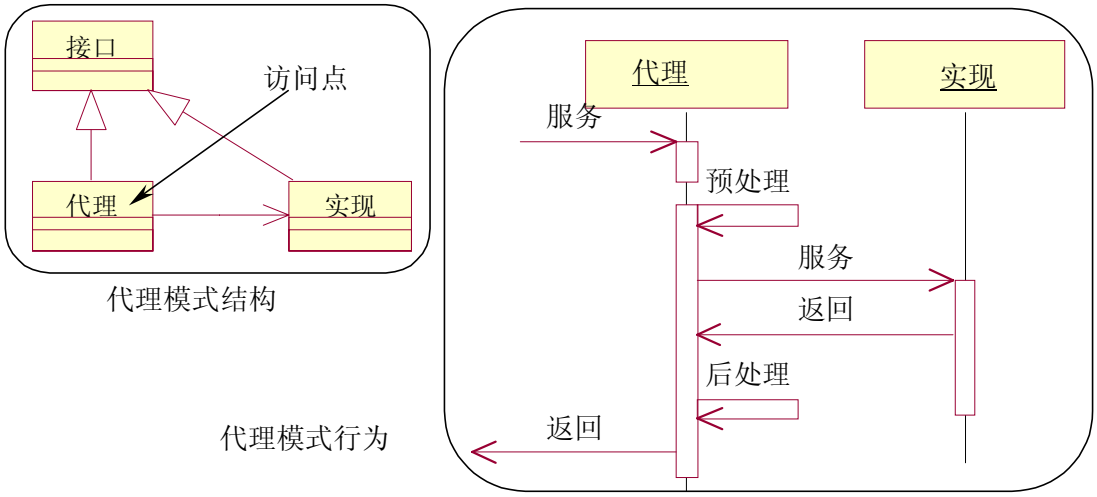


图 10 代理模式的结构和行为

由于篇幅的限制，我们不可能进一步讨论这个过程更多的细节。请参考[Egyed 1999a]和[Egyed 1999b]得到更多信息。

相关的工作

视图集成的缺乏并不是新发现。相反，很多模型描述都谈论到保持视图一致的需求。有时，处理模型提供有关什么任务可以提升体系结构概念完整性的附加方针。例如，一个关于使用 WinWin Spiral Model(螺旋模型)[Boehm et al 1998]的案例研究建议在 LCO(life-cycle objectives, 生命周期目标)和 LCA(life-cycle

architecture，生命周期体系结构)阶段之后使用体系结构复审板（Architecture Review Boards）[AT&T 1993]来检验和证实分析和设计的完整性。类似的观点可以在其他多不胜数的研究工作中看到：

- [Sage-Lynch 1998]讨论集成（企业范围）的不同方面。他们一再地强调“体系结构在系统集成中承担重要的角色。”他们针对三个主要的视图表达需求：企业视图，系统过程和管理视图以及技术实现视图——并且他们强调在这些视图之间保证一致性。
- [Rechtin 1991]非常强烈地要求和接口定义同等地强调需求的有效性和一致性。他进一步促成问题检测和诊断的需要。
- [IEEE 1998]体系结构评估演说。“评估的意图就是要确定一个体系结构的描述质量，以及通过它评定关联的体系结构的质量”。他们进一步陈述了确定哪种体系结构可以进行校验的评价准则需求。
- [Nuseibeh 1995]写到“不一致性是一个复杂的、增量软件开发过程不可避免的部分”，还有“增量软件系统开发包括不一致性的检测和处理”。
- [Perry0Wolf 1992]早就了解到软件体系结构的重要性，并且他们将它陈述为体系结构四个主要好处之一，它们是“用于依赖性和一致性分析的基础”。
- [Shaw-Garlan 1996] 非常激奋地讨论体系结构，作为“系统设计的实质上的民间传说，很少具有一致性和精确性”。他进一步陈述“软件体系结构在框图和通俗散文中找到它的根基。不幸的是，框图和描述是高度二义性的。”

这些以及更多的参考文献，谈论到集成的需要（或缺失）。然而，他们通常不详细地讨论包含的活动（也有些例外）。有时这些工作并不是出于对集成逼近特别的喜爱。在另一方面，有时提议的技术经常只是打算让人们能够互相交流。例如，体系结构复审板[AT&T 1993]或者检查过程（Inspection Process）[NASA 1993]主要地应用于使大多数有能力的人们在一起工作，以便他们能够共享信息。这些技术可能遵循已定义好的过程（例如，检查单）并且可能出产有效的结果，但是实际上，确定和解决瑕疵的活动仍要手工完成，没有多少自动协助。

结论

本工作讨论了在 UML 视图中体系结构不匹配的原因以及说明了集成技术可以怎样按更自动化的方式

应用于鉴别和解决它们。我们用统一建模语言(UML)的语境及其视图（框图）来表述本工作，并使用一个例子来引导视图集成的主要阶段。在本论文中表达的视图集成框架并不限制于 UML。它也可以应用于其它模型和视图（例如体系结构描述语言）。

本工作进一步介绍了在分析软件系统模型概念完整性中的模式使用。既然模式是完好描述和文档化的，在结构和行为两方面，我们可以频繁地使用这些知识用于视图分析。照此，我们说明了关于模式的知识可以用于映射视图（相关建模信息的交叉引用），也可以从一个视图到另一个变换信息。后来我们说明了抽象（Rose/Architect）和变换技术。

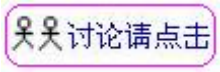
虽然视图集成框架创建的意图是支持自动的视图分析，手工介入经常还是有必要的。既然本文只是集中于模式，其他集成技术也就没有讨论（也可参看[Egyed 1999b]）。我们将发表的内容为：

- 发现（或开发）覆盖更加广泛视图范围的集成技术
- 处理可量化性
- 增加更多的精确性到 UML 中澄清二义性。
- 论述自动化支持的不匹配解决的情况（而不仅仅是不匹配的自动确定）

不管这些问题，我们已经取得在该领域广泛的进步并且我们感觉到使用集成技术的好处，例如在本文中所讨论的，是无限的。我们已经表明不匹配鉴别的任务自动化是可能的（至少部分可以），既然计算机在比较视图无疑更加有效率，这就意味着本质上节省了人工劳动。我们近似的另一个好处是不匹配可以在它们创建的时候就确定。每逢新的数据加入到模型中，工具就可以验证它们。

参考文献

- AT&T (1993) “最佳趋势实践：软件体系结构确认,” AT&T, Murray Hill, NJ.
- Egyed, A. (1999a) “UML中体系结构视图集成,” 提交给 ESEC/FSE’99, http://sunset.usc.edu/TechRpts/Papers/usccse99-511/usccse99-511.pdf.
- Egyed, A. (1999b) “在UML中集成体系结构视图,” 资格报告, 技术报告,软件工程中心, 南加州大学, USC-CSE-99-514, http://sunset.usc.edu/TechRpts/Papers/usccse99-514/usccse99-514.pdf.
- Egyed, A. and Kruchten, P. (1999) “Rose/ 设计师：可视化软件体系工具”, 第三十二届系统科学会议年报议题
- Booch, G., Jacobson, I., 和 Rumbaugh, J. (1997) “用于面向对象开发的统一建模语言,” 文集, 版本 1.0, 瑞理软件公司
- Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996) “模式系统：面向模式的软件体系,” Wiley.
- Boehm, B., Egyed, A., Kwan, J., and Madachy, R. (1998), “使用 WinWin 螺旋模型：案例研究,” *IEEE 计算机*, July, pp. 33-44.
- Gamma, E., Helm, R., Johnson,R., Vlissides, J. (1994) “设计模式：可重用面向对象软件元素:,” Addison-Wesley.
- NASA (1993) “正式软件检验过程标准,” NASA-STD-2202-93.
- Nuseibeh, B. (1995) “软件开发中的计算机辅助非一致性管理,” 技术报告 DoC 95/4, 计算系, 帝国大学, 伦敦 SW7 2BZ.
- Rechtin, E. (1991) “系统体系，创建和建立复杂系统,” Prentice Hall, Englewood Cliffs, NJ.
- Perry, D. E. and Wolf, A. L. (1992) “软件体系结构研究基础,” *ACM SIGSOFT 软件工程笔记*, 十月号.
- Sage, Andrew P., Lynch, Charles L. (1998) “系统集成和体系构造：原理概述，实践和远景,” 系统工程, 国际系统工程会议期刊, Wiley Publishers, 卷 1, 第 3, 176-226页。
- Shaw, M. and Garlan, D. (1996) “软件体系结构：形成原理的透视,” Prentice Hall.
- Siegfried, S. (1996) “理解面向对象软件工程,” IEEE Press.



项目经理面试指南（上）

Patricia L. Ferdinandi 著，zhoufang 译

关于本文

简介

本文的目的是为应聘项目经理提供帮助。项目管理是升迁的途径，需要运用你过去的开发经验，而且薪水通常高于程序员。应聘项目经理的准备工作包括：复习一些常用的概念、术语，问自己一些在面试中经常问到的问题。学会运用一个或多个项目管理计划编制工具。通过以上的准备，将为你应聘这个职位增加信心。

想好你要说的内容并准备回答涉及面广泛的问题是成功应聘的重要方面。与应聘技术职位不同的是，项目管理问题的答案往往是主观的。要牢记技术项目的项目经理的职责是组织项目成员通过完成技术任务而达到某种商业目标。该技术任务应该是可应用或维护的，都必须满足客户/用户的要求和期望。

本文的目标并不是教授如何进行项目管理。这方面有许多很好的书、杂志和研讨班。本文或本文的参考书目中将列出一些。本文将介绍如何回答有关应聘问题的方法和思路。你可以根据自己的经验，观察其他项目经理，应聘职位的岗位描述对答案进行组织。无论被问到什么问题，无论你如何回答，记住运用一个项目经理最有用、最重要的特性.....**常识**。

什么是真正的项目管理

任何成功的项目都不可能是某一个人的功劳。一个成功的项目是多个部门的众多人员共同努力的结果。这些人，组成一个项目团队，具有不同技术水平，才能，工作风格和知识。

项目团队需要有一个共同目标，共同的前景，并且清楚的知道他们要做的工作。该团队，无论采取何种报告结构，必须能够很好地工作和激励以达到商业目标。

项目经理是项目团队的领导。他/她的职责是激励团队以积极的方式完成任务。该职位需要具有技术和

人际技能，需要每天关注的内容（顺序如下）如下：

- 业务
- 公司
- 项目
- 团队
- 个人
- 技术和方法的变更

项目经理的技能应包括技术技能和管理技能，坚实的技术基础能够在技术方面对团队起指导作用，管理技能有助于沟通和解决问题。

管理技能不仅限于技术方面，还包括解决问题的能力，估算能力，编制计划的能力，人际和沟通能力。

你可能已经意识到自己忽视或缺乏某些领域的知识。因此，本文的读者为：

- 没做过项目经理的人
- 已经是项目经理，但认为自己的技能已经过时的人

项目经理是什么

项目经理角色

项目管理是估算、计划编制、重组、整合、评估和修正等过程的不断重复，其中包括管理人员，用户参与和解决问题，直至达到项目的商业目的。

管理层需要什么样的人

每个经理都在找有能力完成某一商业目标的人。最困难的是要了解他们懂什么和能做是么。比较困难的是，不知道需要多少人。

因此，你必须使招聘人员认为你是真诚可靠的。这不仅限于项目范围内，还包括与管理层和客户保持联系。

管理是指无论在有利或不利的环境中都能应对自如。在问题没有被详细表述或没有可选的解决方案时，你必须表现出你的管理才能。如果你让管理层来解决所有的问题，那要你还有什么用，管理层正在做

你做的工作呢。

人员管理技能

了解人们的心理和他们的工作方式是项目经理必需的素质之一。每个人都不同。通过了解你的和别人的工作方式，可以缓解压力，便于沟通。

IBM 多年来的口号是“尊重每一个人”。这具体表现为了解你日常工作中接触的人。要做到这点，你必须了解你自己并且知道你是如何激励别人或对别人施加压力的。

阅读迈尔斯-布里格斯（Myers-Briggs）人格类型分析方面的书籍是一个很好的开端。Katherine Briggs 和她女儿 Isabel Briggs-Myers 制作的问卷（MBTI 迈尔斯-布里格斯人格类型定向）用于帮助人们发现他们的个人风格及对团队产生的影响。该问卷是在 Carl Jung 的“心理类型”基础上发展而来的。此类书在书店有关自我提升和心理学的分类中均能找到。

你应该理解个人工作风格，并且牢记这些实践经验。以下所列的项目应该成为与人相处的第二种本能。也是每个想成功的项目经理必备的常识：

- 尊重每一个雇员（供应商）
- 虚心倾听
- 做出见识广博的决策
- 不要当众批评别人
- 了解自己的实力和做事的先后顺序
- 真诚地听取团队成员的意见和建议
- 对目标和交付产品有清楚的了解
- 在 IT 团队中提倡合作和信息共享
- 了解每个人的做事风格及他们的优缺点
- 表扬应以团队成员喜欢的方式，真诚地表达
- 将负面影响视为成长的机会
- 以积极的方式提供指导

你不能管理你无法控制的东西

如前所述，项目管理是执行一系列可重复的任务以完成某个商业目标。为了完成任务，你必须建立控制体系。因此，应对下列方面的问题有所准备：

度量方法：度量方法如果没有管理好或运用好，会产生负面影响。度量方法可以作为计划编制的“输入”，可以在项目进展过程中和结束时进行统计，为下一个项目或项目的下一个阶段提供参考。用度量方法来评估员工的绩效是不恰当的。

项目计划：通过制定项目计划能够得到正在执行的任务的关键检查点。这些检查点是达到商业目标的路标。要记住项目计划不仅只对新的开发项目有用。他们在支持和维护中同样重要。许多项目经理都犯同样的错误，他们编制一个十分出色的计划，但从不付诸实施。事实上，他们很少按计划进行工作。

预算：估算和编制计划的同时要做预算。许多项目经理要制作和管理他们自己的预算。如果你能使实际工作进展和计划一致，那么你的工作就会变得比较简单。大多项目管理工具都具有使费用（按小时，天，或年计）与某个资源相关。许多公司的财务部门认为的资源费用包括企业一般管理费用。另外一些公司可能根据项目名称或用户，管理方式，员工和顾问分别计算。（对于顾问，还要考虑他们的加班费）设备费用也要单独考虑。记住还要考虑运行项目应用所需的软件工具和硬件。（例如销售部门的彩色打印机）

员工工作计划：人是任何项目中有价值的。一个人可以促进项目成功或项目进展顺利，也可能对项目产生破坏。员工工作计划能对员工的成长起到建设性和实际作用。大多组织有自己的格式。但无论形式如何，下列事项必须包括：

- 职责明确
- 客观地评价员工的优缺点
- 为员工提供参与制定其发展方向和对其进行评估的机会

项目管理的奖励/压力

项目经理的角色是一柄双刃剑。这个职位要承担一定的压力，也会得到相应的奖励。一旦你成为项目经理，就必须对这两方面做好准备。

成功地完成一个系统，每个人都会得到奖励。能够帮助员工开发他们的潜能是项目经理特有的回报。在任何任务中，人都是最重要的元素。通过运用自己的管理技能造就了一个充满活力的团队，是一件值得骄傲的事。

人员同样是最大的压力。人毕竟会受到那些不受你控制的事物的影响。团队成员的家庭困难，彼此间的个性冲突都需要项目经理来处理。

任何有关应用或团队成员的事情首先要找的就是项目经理。上层领导和用户认为你是对项目的拖延、需求的遗漏、系统中的 bug 和不正确等唯一的负责人。

准备面试的方法

书、杂志、组织和研讨会

本文的参考目录中列出了许多能得到有效的管理实践信息的地方。去寻找管理方面的书籍，包括技术管理和商业管理两个方面。阅读管理大师，例如：Peter Drucker，C. A. Gallagher和A. Maslow写的书和文章。他们提供了在任何领域都使用的管理知识。信息管理大师例如：Tom DeMarco, M. Page-Jones, Ed Yourdon, L. L. Constantine等等提供了许多条理清楚的、经过实践检验的方法。

如果你要同用户一起工作，要阅读一本有关领域的专业书籍。了解业务比了解技术环境更重要。事实上，让用户参加面试过程越来越流行。要准备得更充分，可以买一本《哈佛商业评论》（*Harvard Business Review*）这是一本很好的杂志，适用于商业读者同样也适用于IT管理。许多IT杂志例如《CIO杂志》及在参考书目中列出的书目中都有有关项目管理和人员管理方面的文章。这些杂志中还包括概括或详细的技术性文章。

可以和美国管理协会（AMA）和其他商业组织取得联系，获取管理信息。值得一提的是，卡奈基梅隆大学的软件工程研究所（SEI）在90年代提出的管理软件过程，最新标准版本为SEI9000。

许多技术研讨会，例如数字咨询和技术转换研究所（Digital Consulting and Technology Transfer Institute）有许多不同领域的项目管理和技术研讨会。另一种途径是通过你所在的组织。他们也许会提供有关授权、谈判和倾听技巧等的课程，所有这些都有助于你准备项目管理。

你应该了解的软件

掌握一种项目管理工具。例如微软的Project和Applied Business Technology/Project Workbench。所有这些工具都有许多有效的项目管理方法和术语字典。

除了上述提到的工具外，还有一个越来越流行的工具可以针对不同技术环境中的项目在计划编制、费用估算和管理方法上提供帮助。这个工具就是LBMS/Process Engineer，具有CASE界面的工具。

如果你使用过此类工具，把这些内容列在你的简历中。当然，不仅要掌握工具，你还必须具有坚实的

基础知识和项目管理方法。

一个项目经理必须足智多谋。通过email进行通信已经取代了电话和邮寄备忘录。许多公司有自己的系统，还有许多公司使用Lotus Notes。无论是用何种产品，必须具有如下性能：

- 能够与处于不同地理位置的人取得联系
- 能够有效地通知团队（包括供应商）范围，进度的变更
- 能很快地解决小问题

要记住人们工作方式的差别，性格内向的人更愿意通过email沟通。这样他们可以有时间思考问题的答案而不是在会议上立刻做出答案。

作为一个项目经理，你可能会作报告（report）和介绍(presentation)。因此，需要掌握字处理软件和图形软件。这些软件在市场上都可以买到。在你的简历上列出你会使用的此类软件。

寻找思想

任何行业都有好的项目经理和差的项目经理。你可以从两种项目经理身上得到启示（什么是应该做的而什么是应该避免的）。如有可能，问一些优秀的项目经理他们是如何做的。如果你对你的职业发展道路还不太清楚，你可以拿一篇刚刚读过的有关文章，问问这些项目经理对此文的观点。

一个成功的项目经理的标志有拥有一支气氛融洽的积极的团队，上层领导的信任和用户的尊重。一致的行动是另一个标志，它是衡量领导能力的基础。优秀的项目经理应该了解每个雇员的长处和短处。他们认为失败并不是缺点，而是一次学习机会。

项目经理必须建立一套专业标准。但按照一套完美的例子来进行管理却是一个失败的项目经理。这虽然说明他们的多才多艺，但更体现了他们在授权和沟通方面的能力不足。使原来想积极工作的员工变得消极的做法可以毁了项目经理。你在技术方面的能力应该用于指导和培训员工。如果你参与编程或设计，你不是在开发你的团队，也不是在做项目经理。

项目计划技术

以下是在面试中通常会提到的有关项目计划编制的术语和图表。大多项目计划编制工具都会使用到一些或全部术语和功能。你应该复习一下有用的一个或多个项目管理工具，这有助于你进一步熟悉常用的技术和功能。

图表类型：

甘特图：用图形，特别是条形图，描述项目进度的图表。每一个条形符号代表不同的意义。例如：关键任务的条形符号及/或颜色可能与非关键任务的不同。概要任务（活动或阶段）的符号可能于其他任务不同。

Pert图：用流程图来表示所有任务的现行依赖关系。PERT的意思是计划评价与审查技术，是一种网络图。

任务列表：文本/纵向地列出项目计划。通常至少应包括以下栏目：任务编号，任务名称，开始日期，结束日期，持续时间和工作效率。

工作分解结构：项目任务和/或活动的结构图。

关键路径：是贯穿整个项目的一条路径，表明在限定的时间成功完成项目涉及的各任务间的依赖关系。调整关键路径上任务的时间进度将会影响整个项目的交付时间。关键路径方法（CRM）图是一种网络图，用于项目的进度控制和协调项目的活动和事件。

可交付成果：证明一个或多个任务完成的有形事物。例如：逻辑数据模型。

依赖关系：任务间的联系会影响一个或多个任务的开始时间。例如：在没有弄清需求前，不能开始编程。

JAD/简化方法：联合应用程序设计(简化方法是90年代的术语)。一套面向结果的,大脑风暴式的,有一个共同的商业目的信息集合/分享会议。该方法是IBM公司在1970年开发的，由固定的，结构化的过程组成，并在一个有经验的实施者的领导下进行。简化方法去掉了一些结构,然而,仍要求所有各方都必须参加所有的会议和一个有建模技术的记录员作记录。参加者们包括项目团队,管理(与用户)和行政官员。为会议的成功，每个人必须理解和同意目的并且尽快解决他们的任务。

延迟：是任务的结束时间和与其相关的任务的开始时间之间的延迟时间。这允许任务结束时间和开始时间的重叠和拉长。

方法论：一种明确的、有组织的、可重复的、结构化的方法/技术，以完成一个通用的目的。这些技术或指南定义步骤，任务，角色，目的和可交付成果，这些是任何系统的成功的实现所必须的。

衡量标准：一个一致并且可重复的测量一个项目的大小和复杂性的方法。标准准备在整个项目生命期中使用许多方法中的一个。今天公司使用的流行方法是：

a) 功能点（Allan Abrecht）

b) 重要事件 (Tom DeMarco)

c) 加权平均

d) 代码行

里程碑：在项目生命期的一个重要的事件的结束。通常一个里程碑是在关键的路径上的一项活动。它不必是一个有形的可交付产品例如一个逻辑数据模型，但可以是用户对工作成果的肯定。

阶段/活动/摘要标题：概要级的概念。不是所有的项目管理工具都强调特定的阶段和摘要一级的格式，然而许多标准的开发方法用这些术语进行工作分解。

RAD：快速的应用开发(如果不正确地使用会有破坏作用)。通过应用程序生成器，建模和快速原型工具的使用加快开发工作的一条途径。最大的改进是在整个开发生命周期中加入快速原型。这在编码前了解清楚用户需求提供优秀的工具。

资源限制：一个基于可得到的资源的数量，每个资源的技巧的水平，资源工作时间表而开发的计划和时间表。

范围变更：对原先设计要求的功能增加而没有对人员，时间或费用的影响进行评估。范围变更可能是一个商业用户或一个热心的程序员提出的。两者影响系统的交付并且不能被估计，分析，或记录。

面试中的表达的要点(就算问题没被问)

如果你没有管理经验

对于那些从未正式管理过一个项目的，可能是非正式地管理过的人。在那些情况中，当强调他们的技术背景优势的同时需要明确说明他们没认识到他们已掌握的那些技巧。你可以提及你是怎么不得不在没有授权的情况下领导一个大型的开发团队进行工作的。需要强调的是没有一个稳固的技术的基础,你的工程任务和估计的决定可能被过分简单化。当你是项目的领导人，你需要提供技术的连贯避免团队超负荷工作。

如果你的技术技巧在未来的技术的环境中是落伍或不同的

你不需要理解技术环境的内部是如何工作的，但是你应该理解一般的概念和特征决定环境的能力和弱点。许多项目管理技巧是超出技术范围的。因此,如果你的技术技巧是落伍的，你仍然能强调你在技术上能负独立责任。提及你管理的应用类型和及其商业作用。提及团队是如何有效地完成目标的。强调你的管理

哲学。提到上级，与你地位同等的人，你的用户和部下是如何评价你的管理能力的，记住提起任何你掌握的商务领域知识。在面试时应该将你对你的技能落后的恐惧抛在一旁。一旦你拥有这个工作，你将能向公司内的专家询问。在所有组织中都有各方面专家的非正式的机构。你可以到处打听一下，把他们找出来。

问面试官的问题：

即使你通过面试，得到了这个职位，你还需要信息进行估价，这时是你的好机会。如果这将是项目经理的第一个工作任务，这尤其是关键。你需要明白你的工作环境。因此，你可以问下列问题：

1. 公司优先权是什么？
2. 本项目的执行资助者是谁？
3. 公司使用的开发原理体系是什么？
4. 本项目最后期限是什么？
5. 有量度项目成功的方法吗？
6. 你的新经理将怎样保持项目信息灵通？
7. 你的新经理管理哲学和风格是什么？
8. 项目上的人们的技能水平是什么？
9. 你将管理的项目的范围被充分地定义吗？
10. 技术环境已经选好了吗？

以下是典型的项目管理面试中通常会问到的问题（期望的回答）：

很多的问题的答案是主观的，面试官想知道你的观点是否和他们的及公司一致。问题的构成如下：

1. 项目管理软件工具知识，
2. 编制项目计划的技术，
3. 人员管理技能
4. 沟通技能
5. 原理体系知识（标准开发生命周期和项目管理）。

项目管理软件工具知识

问题1：工期和工作量之间的差异是什么？

答案1：工期是商业/日历上的天数，与人数和工作量无关。工作量是与日历天数无关的人的工作。例如：

一天的工作量对于一个一只花50%在时间在上面的人来说，他的工期就是两天。如果两个人全职工作，工期是1天，而工作量是两个工作日。

问题2：怎样和为什么要在编制项目计划时考虑依赖关系？

答案2：根据使用的软件包，依赖关系可以通过将任务及其后续任务的标识符进行关联来表示。依赖关系说明了任务之间关联/并列的要求。依赖关系可以是指在另一个任务能开始之前有一个任务必须完成。例如，逻辑模型必须在物理模型前完成。但测试并不是要在所有编程工作完成之后才开始，如果没有完成的程序对线性测试没有影响。

项目计划加入依赖关系，就能找出项目的关键路径并且能够确定它对项目工期的影响。

问题3：你怎样将人的工作步调与计划结合？

答案3：根据组织使用的具体的工具，可以将资源拆成更小的资源/单位，或者可以将任务拆成更小的任务。

问题4：你怎样将培训，假日和个人教育时间表结合起来？

答案4：每个产品都有标明不工作的天数的公司/全球的日历。每个产品都也有个人的资源日历标明个人不工作的时间。如果项目需要教育和培训，应该把它们象任务那样写在项目计划上。

问题5：你怎样安排类似状态会议这样贯穿整个项目但只需要极少的时间和工作量的任务？

答案5：它的工期将和整个项目时间一样长，占工作量的百分比很小。被分配给任务的每个人花在该任务的时间占他时间的百分比极低。

问题6：实况报告对计划的作用以及实况与最初预计的比较有何价值？

答案6：根据组织使用的特定的工具，每个工具都为实况报告中输入相互独立的要素/域信息。也可以将报表进行分类，来向团队成员和其他相关团体说明关键路径的变化或时间表的调整。这些报告对已实现工作评价和作为在计划下一个工程或阶段的输入有价值。另一个把估计和实况报告比较的有价值的用途是

把范围变更对项目的影响记录下来。

做项目计划的技能

问题7：你为什么制定项目计划？

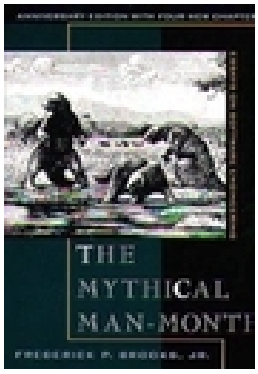
答案7：项目计划是实现成功的系统的路线图。它提供了一种手段来通知每个人希望他们做什么及何时完成。它帮助项目经理使管理层，商务用户和支持团体了解项目状态和调整特殊的资源。逐项列记的“一览表”协助对任何变动的影响进行迅速评估。当实况报告与计划联系起来后，项目计划为今后项目的任务划分和估算提供了有用的信息。

问题8：你将怎样着手做项目的计划？

答案8：进程安排是一门艺术。根据已知有关业务目标的事实，公司一般标准，以及可以利用的过去的经验。可以从清楚地定义范围和目标开始。把项目的风险和制约做成文件。差的估计源于对业务知识和项目范围缺乏了解。可以从项目任务分解入手，例如先划分阶段，然后定义每个阶段的活动，再定义每个活动中的任务。识别和文档化里程碑和可交付产品。项目计划是当信息变得可以利用的时，不断细化的有生命文件。很好地记录进度的变化对项目经理，开发团队，支持团队，以及管理层，商业用户都有益处。

问题9：你将怎样着手制定项目计划？

答案9：在适当的活动和阶段或其他的概括的标准说明下，输入确定的任务。将适当的可交付产品及里程碑和特定的任务联系起来。连接全部需要依赖关联的任务。把资源角色或资源名字加到每个任务上。应用度量结果确定事先的任务工作量，把更多的时间用于需求收集，设计和测试。考虑所有已知的节假日，培训，休假或其他的资源停工时间。计划草案将同支持团体，管理层和商务用户一起复查，做为补充性的输入和最终的批准。



讨论请点击

《人月神话》节选

Fred Brooks 著，Adams Wang 译

Fred Brooks 在 1956-1965 之间领导 IBM System/360 大型电脑的开发计划，由于复杂的需求，以及当时软件工程水平低下，使得 System/360 的开发工作陷入了前所未有的、最可怕的“软件开发泥潭”，并催生了最著名的失败论著--《人月神话》（"The Mythical Man-Month"）。 1975 年出版的《人月神话》一书，是软件工程经典名著。1995 年，为纪念该书发行 20 周年，第二版上市，第一次发行印数就达 250,000 册。1986 年，Brooks 发表了重要文章——“No Silver Bullet”，激起众多人士对软件危机与奇迹的好奇与争论，而"No Silver Bullet"也成为烩炙人口的名词，一直持续到今天。《人月神话》的中文译本即将发行，译者为 UMLChina 的 Adams Wang，这是翻译草稿的片段。

焦油潭（The Tar Pit）

史前史中，没有别的场景比在焦油潭中垂死挣扎巨兽的场面更生动逼真。上帝见证着恐龙、猛犸、剑齿虎与焦油束缚的搏斗。它们越是挣扎的猛烈，焦油纠缠的越紧，没有任何猛兽足够的强壮或者具有足够的技巧能挣脱，最后都沉入了坑底。

过去几十年的大型系统开发犹如这样一个焦油潭，许多大型和强壮的动物在其中剧烈的挣扎。大多数开发出了可运行的系统——其中，非常少数的满足了目标、时间安排和预算。各种团队，大型的和小型的，庞杂的和精干的，一个接一个在焦油潭中被束缚。看上去没有单个的问题会导致困难，任何一个都能被解决。但是它们的相互纠缠和同时累积使行动越来越慢。每个人似乎都被问题的麻烦程度感到惊讶，并且很难洞悉问题的性质。然而如果我们想解决问题，就必须试图去理解它。

因此，首先让我们来识别系统开发这个职业以及它内在的乐趣和苦恼。

编程系统产品

某个人在报纸读到新闻，讲述两个程序员如何在改造的车库中编写了超过大型团队工作的重要程序。接着，每个编程人员准备相信这样的神话，因为他知道自己能以超过产业化团队的 1000 语句/年的生产率来构建任何程序。

为什么并不是所有的产业化队伍没有被专注的二人组所替代？这里必须看一下产出的是什么。

图 1.1

在图 1.1 的左上部分是程序（Program）。它自己是完整的，准备在作者开发的系统平台上运行。这通常是车库中产生的产品，以及单个的程序员在估计生产率时使用的标准。

存在两个途径使程序转变成更有用的，但是成本更高的东西。它们表达为图中的边界。

水平边界以下，程序变成编程产品（Programming Product）。这是可以被任何人运行、测试、修复和扩展的程序。它可以在多种操作平台上，供多套数据使用。欲成为通用的编程产品，程序必须以通用的风格来编写。特别的，输入的范围和形式必须推广以适用于所有合理采用的基本算法。接着，程序必须被彻底测试，以使它可以被依赖。这意味着一个能探测输入边界和范围的详实的测试用例库必须被准备、运行和记录。最后，程序向程序产品的提升需要完备的文档，以使每个人均可以使用、修复和扩展。作为经验数据，相同功能的编程产品的成本至少是经 debug 程序的三倍。

垂直边界的右边，程序变成编程系统（Programming System）中的一个组成部件。它是在功能上协作、具有规范格式的、相互交互的程序集合，从而通过组装形成整个系统。欲成为编程系统部件，程序必须编写使输入和输出一致于精确定义接口的语法和语义。同时程序需要设计成满足预先定义的资源限制——内存空间、输入输出设备、计算机时间。最后，程序必须同其它系统部件，以任何所设想的组合进行测试。测试的范围非常广，因为根据组合测试用例会不断增加。相同功能的编程系统部件的成本至少是独立程序的三倍。如果系统具有多个部件，成本会增加得更多。

图 1.1 的右下部分代表编程系统产品（Programming Systems Product）。它同以上的所有情况不同，成本高达九倍。然而，它是真正有用的产品，大多数系统编程工作的目标产出物。

职业的乐趣

编程为什么有趣？作为回报，它的从业者期望着什么样的快乐？

首先是一种创建事物的纯粹的快乐。如同小孩从玩泥巴中感到愉快一样，成人喜欢创建事物，特别是自己的设计。我想这种快乐一定是对上帝创造世界的映射，一种呈现在每个独特、崭新的叶片和雪花的喜悦。

第二是创建对其他人有用东西的欢乐。内心深处，我们需要其他人使用我们的劳动成果并感到有所帮助。从这个方面，编程系统与小孩用粘土为“爸爸办公室”捏制铅笔盒没有本质的区别。

第三是将相互啮合的部分组装在一起，并看到它们精妙的工作，实现预先所内建的结果，所体现的魔力。程序化的计算机具有像弹珠游戏或点唱机所有的令人入迷的魅力。

第四是学习的乐趣，来自于该工作的非重复的特性。问题在某个或其它方面总不同，解决的人员可以学习新的事物：有时是实践上的，有时是理论上的，或者两者兼有。

最后，是在如此易于驾驭的介质上工作的乐趣。程序员，像诗人一样，几乎工作在纯粹的思考中。程序员凭空地通过实现想象来建造自己的城堡。很少有创造的媒介如此的灵活，如此容易的完善和重建，以及如此容易的实现概念上的构架。（如同我们将看到的，容易驾驭的特性具有它自己的问题）

然而程序同诗歌不同，是实实在在可以移动和工作的，以及产生独立于程序的可见输出。它打印出结果，绘制图形，发出声音，移动支架等。神话和传说的魔术在我们的时代变成了现实。即在键盘上键入正确的咒语，显示屏幕活动起来，显示从来没有或者存在的事物。

编程非常有趣，因为它满足我们内部深处的创建渴望和愉悦了所有人共同的感受。

职业的苦恼

然而并不全部都是喜悦，了解一些固有的烦恼能使它们出现时，更容易忍受。

首先，必须追求完美。计算机也以这种方式来变戏法。如果咒语的一个字符、一个停顿没有与正确的形式严格一致，魔术不会出现。人类不习惯于完美，只有很少的人类活动需要它。我认为，向完美的方式进行调整是学习编程的最困难的部分。

其次，其它的人设定了目标，供给资源，提供信息。编程的人员很少控制工作的环境，甚至工作的目标。用管理的术语而言，某人的权威对他所承担的责任是不充分的。不过，看起来在所有的领域中，对完成的工作，很少能提供与责任一致的正式的权威。而现实情况中，工作推进的动力需要实际（相对于正式）

的权威。

对其他人的依赖的影响比较突出，这对系统编程人员尤其痛苦。他依赖者其他人的程序。通常这些程序设计得不合理，实现拙劣，发布不完整（没有源代码或测试用例），或者文档记录得很糟。所以系统编程人员必须花费时间研究和修改在理想情况下应该完整、可用和有用的东西。

下一个烦恼——设计概念是有趣的，但寻找琐碎的 BUG 仅仅是一项工作。伴随着创造性活动的，往往是枯燥、沉闷的时间，艰苦的劳动；程序编制工作也不例外。

另外，人们发现调试查错具有线性的收敛，或者更糟的是具有二次方的复杂度。所以，测试一拖再拖，寻找最后一个错误比第一个错误花费更多的时间。

最后一个苦恼，有时也是一种无奈——当某人花费大量劳动的产品在完成以前或完成时，显得陈旧。可能同事和竞争对手已在追逐新的、更好的设想。可能替代的方案不仅仅在构思，而且已经在安排了。

现实的情况通常比上述的好一些。当产品开发完成时，新的、更佳的产品通常并不可用；仅仅是被大家所谈论。另外，它同样需要数月的开发时间。事实上，仅现实需要时，才要求使用最新的设想。因为所实现系统的价值已能体现回报，满足要求。

诚然，产品开发所基于的技术不断在进步。一旦设计被冻结，在概念上就已经陈旧了。不过，实际产品的实现需要阶段化和进行度量。实现的落后情况需要根据其它现有的系统进行测量，而非未实现的概念。我们所面临的挑战和任务是在现实的时间、有效的资源范围内，寻找实际问题的切实可行的解决方案。

这，就是编程。一个许多人痛苦挣扎的焦油潭以及一个乐趣和苦恼共存的创造性活动。对于许多人而言，其中的乐趣远大于烦劳。本书的剩余部分试图为通过这样的焦油潭搭建一些木板小径。

神话的人月（The Mythical Man-Month）

在众多软件项目中，时间进度的缺乏是造成项目偏移的最主要原因，比其他所有因素加起来影响还大。导致这种普遍性灾难的原因是什么呢？

首先，缺乏有效的估算技术。更加严肃的讲，它反映了一种无声，但并不真实的假设——一切都将运作的很好。

第二，我们采用的估算技术，错误地将进度与工作量相混淆，隐藏了人和月是可以互换的假设。

第三，由于对估算不确定，所以我们缺乏一种谦逊的坚持。

第四，对安排的进度缺少跟踪和监督。在其他工程领域被验证的技术和常规的程序在软件工程中往往是快速演化的。

第五，当进度的偏移被识别时，自然（且传统）的反应是增加人力。这就像使用汽油灭火一样，只会使事情更糟。越来越大的火势需要更多的汽油，从而开始了一场注定导致灾难的循环。

进度监督是另一篇论文的主题。本文我们对该问题的其他方面进行更详细的考虑。

乐观主义

所有的编程人员都是乐观主义者。可能这个现代的魔术特别吸引那些相信美满结局的人。可能成百上千琐碎的挫折赶走了所以的人，除了那些习惯上只关注结果的人。可能仅仅因为计算机还很年轻，程序员更加年轻，而年轻人总是些乐观主义者。但不管所选择的过程如何工作，结果是勿庸置疑的：“这次它肯定会运行。”或者“我刚刚找出了最后一个错误。”

所以系统编程的进度安排背后的第一个假设是：一切将正确运作。即每一项任务仅耗费它所“应该”花费的时间。

对这种在编程人员中乐观主义的弥漫理应受到慎重的分析。Dorothy Sayers 在她的《The Mind of the Maker》一书中，将创造性活动分为三个阶段：构思、实现和交流。书籍、计算机、或者程序的出现是首先作为一个构思概念或模型，在时间和空间之外构建，而在作者的脑海中完成。它们通过钢笔、墨水和纸，或者通过电线、硅片和铁氧体，在现实的时间和空间中被实现。当某人阅读书本、使用计算机和运行程序的时候，创作过程得以结束，从而与创意的作者相互交流。

以上 Sayers 的阐述不仅仅可以描绘人类创造性活动，而且描述基督教义的形成过程。它可以协助我们

日常的工作。对于创造者，我们构想的不完整性和不一致性只有在实现的过程中才能发现。因此，对于理论家而言，书写、试验以及“工作实现”是基本、必要的部分。

许多创建性的活动中，实施的媒介很难掌握，如木头切割、油漆、电器安装。这些媒介的物理约束限制了能被表达的构思，同样它们对实现造成了预料之外的许多困难。

由于物理媒介和构思背后的不完善，创造性的实现需要花费大量的时间和汗水。对遇到的大部分实现上的困难，我们总是责怪那些物理的介质；因为它们并不是顺应“我们”构思的思路，且我们的骄傲主观引导了我们的判断。

相应的，计算机编程基于非常容易掌握的介质。编程人员凭空通过思考来创建程序，即概念和和非常灵活的表达。因为介质的易于驾驭，我们期待在实现的过程中不会遭遇困难；因而造成了乐观主义的弥漫。而由于我们的构思是具有缺陷的，因此会有 Bug；所以我们的乐观主义并不是得到认可的。

在单个的任务中，“一切都将运转正常”的假设在时间进度上具有随机性。由于将遭遇的延迟具有一定的概率分布，所以事实可能如计划安排的那样，而“不会延迟”仅具有有限的概率。大型的编程工作，或多或少包含了许多任务，某些任务间具有前后的次序。而一切正常的概率变得非常小，接近于无。

人月

第二个谬误的思考方式是在估计和进度安排中使用的工作量单位：人月。成本的确随产品的人数和时间的长短变化非常大。进度却不是如此。从而使用人月作为测量一项工作的规模是一个危险和带有欺骗性的神话。它暗示着人数和时间是可以相互代替的。

人数和时间的互换仅仅适用于某个任务可以被分解给参与的人员，并且他们之间不需要相互交流（图 2.1）。这在割小麦或收获棉花时是可行的；而在系统编程中近乎不可能。

图 2.1

当任务由于次序上的限制不能分解时，人手的添加对进度没有帮助（图 2.2）。无论多少个母亲，孕育一个生命都需要十个月。由于调试、测试的次序要求，许多软件均具有这种特性，

图 2.2

对于可以分解，但子任务之间需要相互沟通、交流的任务，沟通的工作量必须添加到待完成的工作中。因此，相同人月的前提下，使用人数的增加来减少时间的最好情况也较未调整前也要差一些（图 2.3）。

图 2.3

沟通所增加的负担由两个部分组成，培训和相互的交流。每个成员需要进行技术上的、项目目标以及总体策略上的培训。这种培训不能被分解，因而这部分增加的工作随人员的数量线性的变化。

相互之间交流的情况更糟一些。如果任务的每个部分必须分别同其他的部分单独的协作，则工作量按照 n(n-1)/2 递增。单对单交流的情况下，三个人工作量是两个人的三倍；四个人则是两个人的六倍。而如果需要在三个、四个人之间召开会议，大家共同的解决问题，情况变得更糟。所增加的沟通的工作量可能会完全抵消对原有任务的分解，则我们会被带到图 2.4 的境地。

因为软件构建本质上是一项系统工作——相互错综复杂关系下的一种实践——沟通、交流的工作量非常大，它很快会支配源于任务分解所节省下来的个人时间。从而，更多的人手的添加，实际上延长了，而非缩短了时间进度。

系统测试

时间进度中，没有其他的部分所受到次序限制的影响，比构件调试和系统测试所受到的影响彻底。而且，所需要的时间依赖于所遇到错误、缺陷的数量和难以捕捉的程度。理论上，缺陷的数量应为零。由于乐观主义，我们通常希望缺陷的数量比实际出现的要少。因此，系统测试进度的安排常常是编程中最不合理的部分。

对于软件任务的进度安排，以下的经验法则我使用了很多年：

1/3 计划

1/6 编码

1/4 构件测试和早期系统测试

1/4 系统测试，所有的构件可用

在许多重要的方面，它与传统的进度安排方法不同：

分配给计划的时间较寻常的多。即使如此，仍不足以产生详细和稳固的计划说明，且不足以包括对全新技术的研究和摸索。

对所完成的代码的调试、测试，投入近一半的时间，比平常的安排多许多。

容易估计的部分，即编码，仅仅被给予六分之一的时间。

通过对传统项目进度安排的研究，我发现很少项目允许为测试计划一半的时间，但大多数项目实际上为上述目的花费了一半的时间。它们中的许多在系统测试之前，仍能保持进度；或者说，除了系统测试，进度基本能保证。

特别的，不能为系统测试安排足够的时间，尤其是一场灾难。因为延迟产生于项目快完成的时候；直到项目的发布日期，才有人发现进度上的问题。坏消息，没有任何预兆、很晚的出现在客户和经理面前。

另外，此时此刻的延迟具有不寻常的、严重的财务以及心理上的反馈。项目配置了充足的人员，每天的人力成本达到了最大的程度。更重要的是，软件是用于支持其他的商业活动（计算机硬件的到货，新设备、服务的上线等等），延误这些活动的商业代价相当高——软件已经到了即将发布的时间。

实际上，上述的二次成本远远高于其他开销。因此，在早期进度策划时，允许充分的系统测试时间是非常重要的。

空泛的估算

观察一下编程人员，同厨师一样，某项任务的计划进度可能受顾客要求的紧迫程度所支配，但紧迫程度无法控制实际的完成情况。约定好在两分钟内完成一个煎蛋，看上去可能进行得非常好。但当它无法在两分钟内完成时，顾客有两个选择：等待或者生吃煎蛋。软件的顾客具有相同的选择。

厨师还具有其他的选择；他可以将火开大。其结果常常是无法“挽救”的煎蛋——一面已经焦了，而另一面还是生的。

现在，我并不认为软件经理内在的勇气和坚持不及厨师，或者不及其他工程经理。但为了满足顾客期望的日期，所进行的不合理进度安排，在软件的领域却较其他的任何工程领域要普遍的多。而使用非量化方法，少的可怜的数据支持，主要借助于软件经理的直觉的情况下，很难产出健壮的、可靠的、规避风险的估算。

显然我们需要两种解决方案。开发并推行生产率图表、缺陷率、估算规则等等。整个组织最终从这些数据的共享上获益。

在基于可靠基础的估算之前，项目经理需要挺直腰杆，坚持他们的估计，确信自己的经验、直觉总比从期望派生出的估计要强。

重复产生的进度灾难

当一个软件项目落后于进度时，通常的做法是什么呢？自然是添加人力资源。如图 2.1 至 2.4 所显示的，这可能有所帮助，也可能无法解决问题。

我们来考虑一个例子。设想一个估计需要 12 个人月的任务，分派给 3 个成员 4 个月，在每个月的末尾安排了可测量的里程碑 A、B、C、D（图 2.5）。

现在假定两个月之后，第一个里程碑没有被达到（图 2.6）。项目经理面对的选择方案有哪些呢？

假设任务必须按时完成。假设仅仅是任务的第一个部分估计的不得当，即图 2.6 确切的表达了上述情况。则剩余了 9 个人月的工作量，时间还有两个月，即需要 4 1/2 的人力资源。所以在原分派 3 个人的基础上增加 2 个人。

假设任务必须按时完成。假设整个任务的估计偏低，即图 2.7 描述了该情形。那么还有 18 个人月的工作量以及 2 个月的时间。将原来 3 个人增至 9 个人。

重新安排进度。我喜欢 P. Fagg，一个具有丰富经验的硬件工程师的忠告：“避免小的偏差（Take no small slips）”。也就是指，新的进度安排中分配充分的时间，以确保工作能仔细、彻底的完成；从而无需重新的确定时间表。

削减任务。现实情况中，一旦开发团队观察到进度的偏差，总是倾向于对任务进行削减。当项目延期所导致的后续成本非常高时，这常常是唯一可行的方法。项目经理的相应措施是仔细的、正式的调整项目，重新安排进度；或者默默的注视着任务项由于轻率的设计和不完整的测试而被剪除。

前两种情况中，坚持不经调整的任务在四个月中完成是灾难性的。考虑重复生成的工作量，以第一种为例（图 2.8）。不论两位新的员工多么有能力，在多短的时间内被雇用，他们均需要接受一位有经验的职员的培训。如果培训需要一个月的时间，那么三个人月会被投入到原有进度安排以外的工作中。另外，原先划分为三个部分的工作，要求重新分解成五个部分；从而某些已经完成的工作会丢失，系统测试必须被延长。因此，在第三个月的月末，7 个人月的工作仍然残留着，但此时只有 5 个有效的人月。如同图 2.8

所示，产品还是会延期，如同没有增添任何人手（图 2.6）。

期望四个月内完成项目，仅仅考虑培训的时间，不考虑任务的重新划分和额外的系统测试，在第二个月末需要增添 4 个，而非 2 个人员。如果考虑任务划分和系统测试的工作量，则还需增加人手。现在所拥有的不是 3 人的队伍，而是 7 人以上的团队；此时小组的组织和任务的划分在类型上都不尽相同，已经不是程度上的差异问题。

注意在第三个月的结尾时，情况看上去很糟。除去管理的工作不谈，3 月 1 日的里程碑仍未达到。此时，存在着很强的重复上述循环的诱惑——添加更多人力。疯狂、愚蠢就在此中。

前面所述的假设第一个里程碑估计不当。如果在 3 月 1 日，项目经理做出了比较保守的假设，即整个估计过于乐观了，如图 2.7 所示。6 个人手需要被添加至原先的任务中。培训、任务的重新分配、系统测试工作量的计算作为练习留给读者。毫无疑问，重现“灾难”所生成的产品，比未增加人手，重新安排开发进度所产生的产品更差。

简单、武断地重复一下 Brook 的规律：

向进度落后的项目中增加人手，只会使进度更加落后。（Adding manpower to a late software project makes it later）

这就是除去了神话色彩的人月。项目的时间依赖于次序上的限制。人员的个数依赖于独立子任务的数量。从这两个数值，可以用较少的人数和较多的时间推导出进度时间表。（唯一的风险是产品可能过时。）使用较多的人数和较少的时间不能得到可行的进度安排。在众多软件项目中，时间进度的缺乏是造成项目的偏移最主要的原因，比其他所有因素加起来影响还大。