X-Programmer 总第6期
软件工程的播种机
www.umlchina.com
非程序员 2001 10

目录

本期审稿：Think

本杂志资料文章仅供学习交流之用

去往讨论组→
（已超过 13,000 人）

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>>

Interface Hall of Shame
GOF patterns for GUI Design

请点击查看详情>>

讨论请点击

《设计模式》作者 John Vlissides

透明，think，Qingrun 整理

9月20日，面向对象技术专家、IBM研究中心研究员、软件开发的里程碑著作《设计模式》的作者John Vlissides在UMLCHINA（http://www.umlchina.com）的“嘉宾聊天室”里与中国的软件开发者在线畅谈。下面是部分交流实录。

关于软件开发技术

huangcy：您能告诉我们在框架和软件构架之间有什么重要的区别么？我经常在一些抽象的层次上对他们感到疑惑，无法区分。

John Vlissides：首先，“框架”是代码，它体现一类软件的设计。例如，会计系统和图形编辑器。而“软件构架”是一种成型的规范。它与建筑学中的“构架”又不完全相同，有很多具体的区别存在。对于越大的项目构架就越重要。最明显的，构建构架是在处理关于建造物的事情。不容易发现的是，软件是很容易扩展和改变的，有很多实际情况会限制你尽可能快的修改你的构架，并且在开发过程中也是非常容易

中断的。

Lipy：您可以告诉我您是如何开始您的软件设计的么？

John Vlissides：设计应用于每件需要继续下去的事物。我总是在写软件的时候开始这一切。后来，我就会学会知道什么在起作用，什么却没有。在我的书中，我也仅仅是将一些我已经学会的知识和能力记下来并发表而已。

lovelybug28: John，您如何看待XP（Extreme Programming）？

John Vlissides：我喜欢在XP中应用模式。我认为XP的核心有五个原则：快速反馈、简单设想、渐进改变、吸收变化、高质量的工作。如果你在一个项目的开发过程中，可以参考一下XP编程方面的书籍和Larman的《Applying UML and Patterns》这本书（即将有新版本）。我认为XP很伟大，特别是对于一些无法准确定义需求的中小型的项目。

Sealw：大多数项目的需求都是很难定义的，许多中国的程序员都被RUP误导了。

John Vlissides：事实上不一定。较大的项目一般会有比较完整容易收集的需求。例如，针对航天飞机设计的飞行控制软件就有非常规范的需求。RUP对于它所擅长的部分的确是非常好用的——即：带有可控需求的比较大的项目。

sealw：是的，在成熟的工业中，需求一般都是比较规范的。但是在中国，大多数是不规范的。

John Vlissides：我谈到的是所有的人，不仅仅是中国。软件技术仍然处于幼年期，看看传统工业发展起来需要多少时间，百年？千年？而我们才写了多少年软件，50年？

关于写作情况

lipy：请问您的新书的名字是什么？

John Vlissides：我个人并没有出版新书。我写的最新的一本书是《Pattern Hatching》。同时我们已经为《设计模式》的第二个版本工作了好几年了，其中大多数代码例子都是基于Java的，现在第二版中没有出现什么新的观点，但我感觉新鲜的东西就快出现了。大多数时候我们都在使用Java，因为它可以最好地

表达我们想表达的思想，另外，它也很流行。顺便说一下，我是一个泛型编程（Generic Programming）的强烈爱好者。我曾经使用Krzysztof Czarnecki工作过。而且我还在考虑出版一系列“compound patterns”（混合模式），我在C++ Report中写了许多关于它们的文章。

babysloth：那，什么是混合模式呢？您能否给我们介绍一下您的这本新书？

John Vlissides：混合模式，请到http://www.research.ibm.com/designpatterns/publications.htm#Articles上查看相关的文章。顺便说一句，它仍然不是一个项目，而仅仅是一个观念。

关于学习

lipy：我刚刚开始学习UML。您能给我一些重要的建议么？

John Vlissides：你可以看一下Fowler的《UML Distilled》。如果你想更深入一点，可以学习Rumbaugh的《UML参考手册》。

Pnren：一些人说win2000是专门为进入互联网设计的，对么？

John Vlissides：难道说unix的网络价值比较低吗？

pnren：我正在自学Linux，我觉得它非常复杂，不是吗？我们应该研究Linux的什么？或者说我们应该如何学习？

John Vlissides：嘿，它可是一个操作系统！（你只需要学会使用它）不过我强烈建议学习Linux。我曾经看到它被证明成功应用于安全应用，特别是密码技术中。我对Linux的内部实现了解得不多，所以无法指出它是否使用了哪些模式。但是我肯定模式存在于其中。研究这样一个可以作为范例的软件是一个好主意。

关于设计模式

paofan：模式是原理么？不仅仅是建模吧？

John Vlissides：对于模式是有一些原理性的观点，特别是Alexander的。Jim Coplien已经完成了模式的“禅”。

Jeffray: John，我想问您一些关于使用模式的规则方面的问题。我的意思是说，是否有一些选择模式的基本原理。

John Vlissides：我不能给你一套很严格的规则，只能建议：你必须在应用这些模式的过程中来学会他们。

Qingzuozhou：在您的设计模式中，主要是关于设计具体的小的组件。那么我如何使用这些模式来构建大型系统呢？是否那些构建大型系统的设计模式都是基于这些小模式的呢？

John Vlissides：不一定使用小的组件，但是在通常的意义上，它们不能成为大型构架的核心流程。

Qingzuozhou：在23种模式中没有任何一种是构建基于数据库应用的，这是为什么？

John Vlissides：那是因为我们没有数据库应用方面足够的经验。

Huangcy：John Vlissides教授，我认为设计模式是应用框架的组件，所以当你要实现一个框架的时候，你需要将这些设计模式具体化。但是当我抽象软件框架的时候，我经常会处理得过于详细，您能否给我一些关于这方面的建议呢？有些时候，我认为问题太过于详细，并且我不能从模型中抽象出正确的构架。如果框架过于详细，就会减少它的可重用性。

John Vlissides：是的，你能够将模式作为一种示例来思考它们，但是…通常模式仅仅是一个将你的需求进化到设计的起点。在Pattern Hatching中，我使用模式设计了一个文件系统API。在不仅仅一个例子中，模式采用了非传统的方式。并且从这些方式中得到了发展。模式意味着对手边的问题做适当剪裁。这就是为什么它们不是代码，而是散文。它们教育你，使你能通过你自己的思考来证明并解决问题。

Huangcy：我没有阅读过您的Pattern Hatching，但是我读过您的Design Patterns，Pattern Hatching主题是什么？

John Vlissides：Pattern Hatching的内容，部分是关于应用模式的注释、部分是指南。部分是我们四个开发模式的背景材料。

http://www.amazon.com/exec/obidos/flex-sign-in/ref=cm_rate_rev/104-5101538-2496757#rated-review

Qingzuozhou：我想知道在设计模式第二版中是否有关于数据库应用的模式。

John Vlissides：没有。模式年鉴有一套关于这些模式的参考。

lovelybug28：我是设计模式的一个初学者，您能否给我一些有用的建议？

John Vlissides：你可以去阅读一下Shalloway写的一本非常出色的新书Design Patterns Explained（编者注：由umlchina翻译组成员透明翻译的中文译本《设计模式精解》即将出版）。这本书对初学者会有很大的帮助。除了这些，我的Pattern Hatching这本书也展示了如何使用设计模式。最重要的是，你应该从应用的角度来学习他们，而不仅仅是阅读它们。

d_jt：我认为设计模式的核心部分是组合和虚拟继承，您同意吗？

John Vlissides：不，恐怕我不能同意。我想这取决于你说的“核心部分”的具体含义。虚拟继承过分依赖于C++了。不过组合是一个重要的部分，如同代理一样。但这些东西都是关于实现的。对于描述问题，模式都是平等的。而人们面临的最大问题是他们不知道哪些问题是前人曾经解决过的。而模式有几个优点：它们向人们展示那些反复出现的问题的优秀解决方案，这些方案是经过时间考验的；它们的名称为开发者之间的讨论构造了一个通用的词汇表；它们提供了重构代码的目标；而且它们可以帮你对不能肯定的方法进行检验。这些巨大的优点，才是模式的核心。所谓“重构”（Refactoring），这是一种修改代码的技术，它让代码更优雅，而不改变代码的功能。

fcx123：我认为在分析模式和设计模式之间没有任何联系，对么？我认为分析着重于现实世界的设计模型。但设计是着重于如何实现它。

John Vlissides：我认为在分析模式和设计模式之间有非常多的关系。同样的关系也存在于分析和设计之间。你不能将分析从设计中独立出来，如果你打算这么做，那么你一定会失败的。分析模式捕捉反复出现的领域建模问题的通用解决方案；设计模式捕捉反复出现的程序设计问题的通用解决方案。同其他模式一样，分析模式也不能保证复用，而且分析模式特别无法保证复用。关于分析模式，最流行的一本书是Fowler的Analysis Patterns：

http://www.amazon.com/exec/obidos/ASIN/0201895420/qid=1000995590/sr=2-1/ref=sr_2_3_1/104-5101538-2496757

windy.j: 在《设计模式》一书出版之后，又出现了越来越多的模式。怎样学习（和选用）那些新的模式，您能给我们一些建议吗？

John Vlissides：学习模式最好的方法就是使用它们——根据你的需要去选择，而不是先入为主的决定。寻找新的模式则是另一种方法。看看《模式年鉴》或许会有帮助：http://www.amazon.com/exec/obidos/ASIN/0201615673/qid%3D960385247/104-5101538-2496757 。关于模式的选择和学习，我不能给你一套很严格的规则，只能建议：你必须在应用这些模式的过程中来学习它们。有一个关于模式的主页：http://hillside.net/patterns/ 。

关于软件行业

paofan: IBM加入到Linux和Java集团，这是否暗示蓝色巨人将会更开放？是否微软和IBM代表两种不同的商业途径——垄断的途径和开放的途径？

John Vlissides：IBM在开发源码领域也是一大巨头。我不知道这是否意味着“两种途径”。不过它们都是商人，都是为了赚钱。微软看重对消费电脑市场的控制；而IBM则看重对更大范围的商用电脑市场的控制。

paofan: 中国程序员无法在一个团队中协同工作。他们认为“破解”可以显示他们的聪明。他们用一种愚蠢的方式进行破坏，而不是建设。他们攻击行业巨头，因为他们自己太渺小。而不管微软还是IBM都没办法控制破解的行为。因此一些本来清白的人也开始变成破解的信徒。

John Vlissides说（迷惑的）：控制“破解”的行为？（似乎没有概念）不过Alexander从中国文化中得到了很多灵感。

关于自己

lipy：John先生，您现在是否在中国？

John Vlissides：不，我在纽约，但是我一直很希望能访问中国。我父亲在1972年访问过中国，就在尼克松访华之后。

babysloth: 现在情况怎样？

John Vlissides：不能说一切都好，但已经安定下来了。

lovelybug28：教授，请问您最常用的语言是什么？C++、Java、还是其他？另外，请问您的年龄？

John Vlissides：到目前为止是Java——尽管我已经没有从事大量的编程工作了。我想，我已经老了。我已经满40岁了，就在8月2日。

babysloth：我对您是如何加入到Gang Of Four(或者应该是Gang Of Three)的故事很感兴趣。您可以告诉我们相关的事情么？

John Vlissides：我可以送给你一篇文章的草稿，它会告诉你我们是如何到一起的。你可以发信到vlis@us.ibm.com来索取这篇文章。…umlchina建议我谈谈Erich Gamma—好的，当然☺Erich来自瑞士，但他看起来象个意大利人。在http://www.research.ibm.com/designpatterns/pubs/ddj-eip-award.htm有他的一张照片（当时我们还比较年轻）。请看说明（我是左起第三个）…哦，lovelybug28指出我是左起第四个，我真笨！（谢谢lovelybug28）也许这是我无意中说出的心里话——我希望成为Erich！☺☺

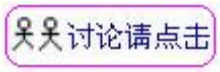
lovelybug28：对不起，John，私人问题：9.11那天你在纽约吗，是否离世贸中心很近？

John Vlissides：是的，听到消息的时候，我正在公司的停车场，距离大约30英里。

Paofan：你是否碰到过Chirstopher Alexander？

John Vlissides：我在OOPSLA '96上碰到过他。

英文记录： http://www.umlchina.com/Chat/vlissides.htm



Temporal（时效）[1]模式

Andy Carlson, Sharon Estepp, Martin Fowler 著，透明 译

抽象

在面向对象设计中，我们不断使用“对象”（object）这个词。对象不仅仅用来表现真实世界中存在的物件，它们也被用来表现那些曾经存在但已经消失了的物件，以及那些可能存在于未来的物件。

上述的要求给我们的建模工作提出了一个特别的挑战，因为如果建模者必须考虑物件随着时间的变化情况，出现在某一特定时间点的对象的复杂度就会大大增加。

这篇文章将向读者介绍三个模式，并通过它们向读者展示怎样通过改进对象模型来解决上述问题，以及最终得到的模型怎样为那些不关心时效问题的客户提供支持。

[1] 译者注：我把“temporal”这个词译为“时效”，意思是“与时间有关的……”。请各位指正。

模式一：Temporal Property（时效属性）模式

别名：历史映射（Historical Mapping）

环境

假设你正在建设一个复杂的信息系统，其中对象的属性必须可以随时间变化而变化。另外你还需要可以跟踪：1、属性在过去怎样变化？或2、属性在未来将怎样变化？或3、以上两点。很自然的，你会让这个属性包含一系列离散值，用这些离散值来表示时间间隔对属性的影响（不能用这个方法来表示象温度这样可以连续变化的属性）。为了实现这个系统，你还可能需要使用某些数据库（可能是关系数据库）。

问题

在考虑“对象怎样随时间变化”时，通常我们的意思是“它的属性怎样随时间变化”。这些属性（property）可能是变量（attribute）、关联（relationship）或者查询操作（query operation）。例如，请考虑图1所示的模型：

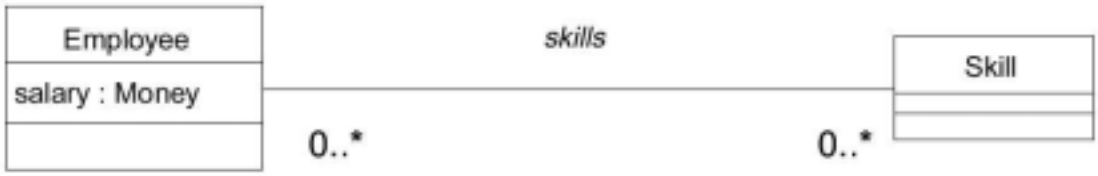


图1：基本的雇员（Employee）模型

在这个模型中，我们关心雇员的两个属性：他的工资和他的技能。于是我们得到了图2所示的Employee接口：

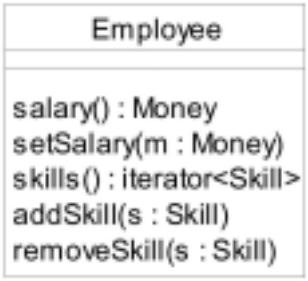


图2：基本的Employee接口

我们也不排除给Skill类提供一个相似的接口的可能性，如果那样做的确有用的话。

现在，我们来看看怎样让这个类体系支持变化，从而让我们可以记录某雇员以前的工资和技能、可以预测未来需要的雇员的工资和技能。

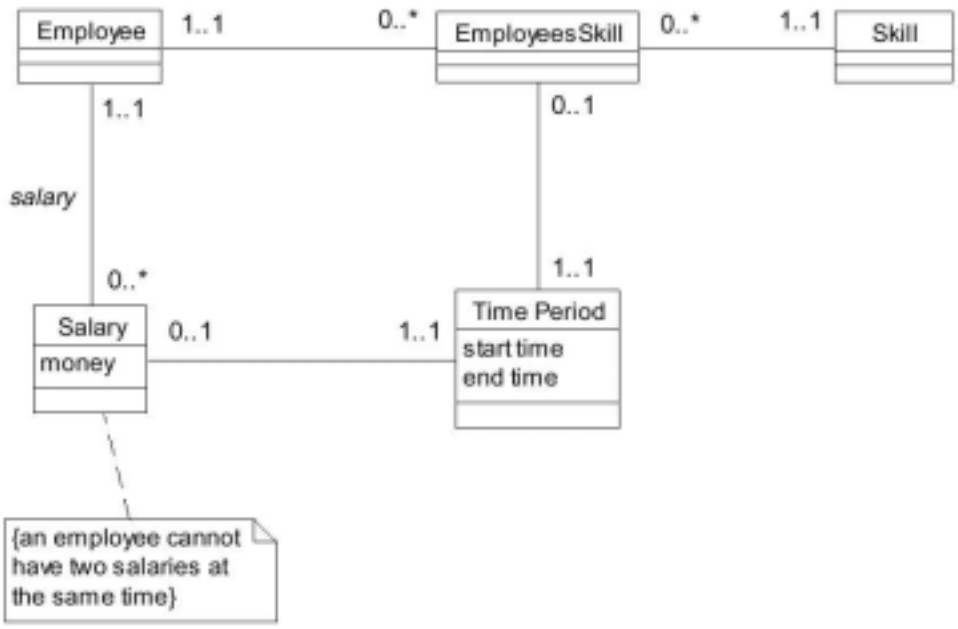


图3：改进的雇员模型，可以显示属性随时间的变化

正如图3所示的，我们给Employee类的属性加上了Time Period（时间段）这个修饰。以Salary为例，我们必须创建一个单独的类来保存工资值，并用一个与之相关的Time Period指出这个工资值有效的时间段。这样，我们就可以允许多个Salary与同一个Employee发生联系（它们拥有不同的时间段）。再以Skill为例，这个类的一个实例可以被许多Employee的实例共享，所以有必要引入一个中间对象来保存相关的Time Period信息。

使用这个模型，我们可以表现如下的信息：

- Dinsdale Piranha于1998年1月1日进入公司，那时的工资是$75,000；然后在1998年4月1日得到提升，薪水升到$85,000。

- Dinsdale于1998年3月1日参加了一个自信训练课程，从此他的技能中加上了“自信”这一条。

现在，假设我们希望知道Dinsdale在1998年2月1日拿多少工资。为了获得这一信息，我们可以检查Dinsdale所有的Salary对象，找到包括“1998年2月1日”这一日期的那个对象——那就是我们关心的。相似的，我们也可以从EmployeesSkill对象中找到Dinsdale在2月1日的技能。在这个例子中，我们很可能找到不

止一个Skill对象。

我们还可以很轻松的给Dinsdale指定一个未来的工资（比方说，为了做他的1999年度薪酬展望[2]），只需要创建一个时间区段从1999年1月1日开始的Salary对象就可以了。

现在我们的模型得到了我们需要的表达能力，但我们必须付出如下的代价：

- 从前我们只有两个类、一个联系、一个属性，现在我们有了5个类、5个联系和一个约束。
- 为了回答关于任何时间点的问题（包括“现在”），我们必须取得并处理比从前多得多的对象。
- 我们失去了原来模型的透明度，尤其是在关心集的容量（cardinality）时：我们不再能轻易断定一个雇员是否可以在同一时刻拥有一份或多份薪水。[3]

总而言之，这个模型的时效方面的改进把其他方面的优点都淹没了。实际上，应用系统中的时效联系比我们这里的多得多。我们把这作为一道练习题留给你：确定一个应用系统中支持时效所需的类、属性和联系的数量。

约束

- **简单的模型更具吸引力**——开发者通常都有“保持事情简单（keep things simple）”的愿望。他们希望在设计时使用最小数量的类和关联，并且尽量不关心时效方面的问题。
- **简单的模型有时候是不够的**——有时候，的确会有变化跟踪（change track）的需要（对于对象的过去和未来的跟踪）。
- **复杂的模型将不那么清楚**——模型的本质可能会被时效部分隐藏起来。
- **变化的影响**——类之间关系的表现形式会发生改变，这会导致的代价是：所有的客户——如果他依赖于对此表现形式的了解——都必须做相应的改变。

---

2 译者注：原文完稿于 1998 年 8 月 3 日，因此 1999 年就是“未来”。

3 译者注：这句我也没有完全理解，请各位一起斟酌一下。原文如下：“We have lost the clarity of the original model, particularly where cardinality is concerned : we can no longer tell at a glance whether an Employee can have one or multiple salaries at a given instant.”

解决方案

为每个需要跟踪其变化的属性建立一个模型（就象“问题”一节中的模型那样），用此模型来表现属性的有效时间段。本模式有几种可选的实现方法，所以我不可能在这里给出一个抽象结构图——如果不使用一个版型（stereotype）（后面将给出一个版型）。“实现”一节中包含了一些可选的模型。

确定了一个模型之后，你应该尽量让客户容易操纵它，因此你应该把时效部分隐藏在一个方便的接口后面，接口中只应该包括客户感兴趣的属性。你还应该为不关心时效功能的客户提供支持，因此你应该添加不需要时间参数的方法，并为这些方法设定一个默认时间值。这个接口如图4所示：

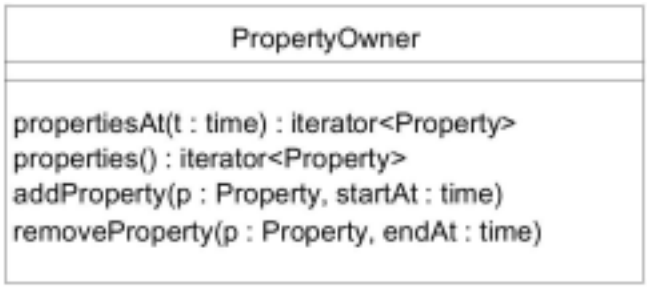


图4：PropertyOwner接口

如果你使用的建模符号允许（就象上面的UML这样），使用一个叫做《temporal》的版型可以进一步简化模型结构（见图5）。

图5：使用版型的Temporal Property模式结构

在我们的例子中，“工资”和“技能”属性可以通过如图6所示的接口进行访问。

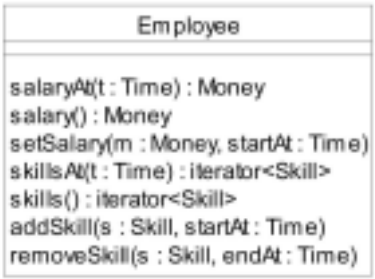


图6：使用Temporal Property模式的Employee接口

正如“问题”一节所说，如果有必要，我们不排除为Skill类提供一个类似的接口的可能性。由于使用了《temporal》版型，“问题”一节中复杂的图变得简单多了（见图7）。

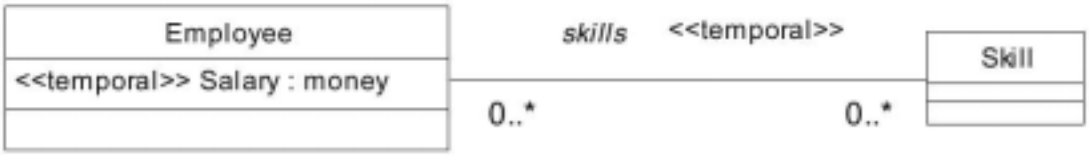


图7：使用版型后的Employee模型

效果

√ 我们为原始模型添加了表现属性随时间变化的能力。

√ 通过使用《temporal》版型表示时效支持解决方案，我们重新得到了原始模型的简洁性。

√ 与简洁性一起，我们还重新得到了原始模型的可表达性（expressiveness）——现在我们可以断定：一个Employee可以同时拥有超过一个的Salary。

√ 我们把关于关联表达（relationship's representation）的主要信息都放在PropertyOwner中，从而确保其他的客户不会与此表达发生紧耦合。

√ 使用这个模式暗示着：对拥有时效属性的对象建模过程将更有一致性。

√ “未来值（future value）”将变成“当前值（current value）”，然后再变成“历史值（historical value）”，这些变化都会随着时间流逝自行发生，不需要对任何存储信息做任何更新。

× 如果你需要关联中间对象更多的信息（例如：EmployeeSkill类中某个技能的级别），本模式将不能满足你。你的需求指出了对中间对象的第一个类的需求，这可能指出了对*Temporal Association*模式的需求。

实现

本模式最简单的实现包括了对一个关联对象（PropertyAllocation）的创建,其中在PropertyOwner和Property之间使用了一个Time Period对象（如图8所示）。

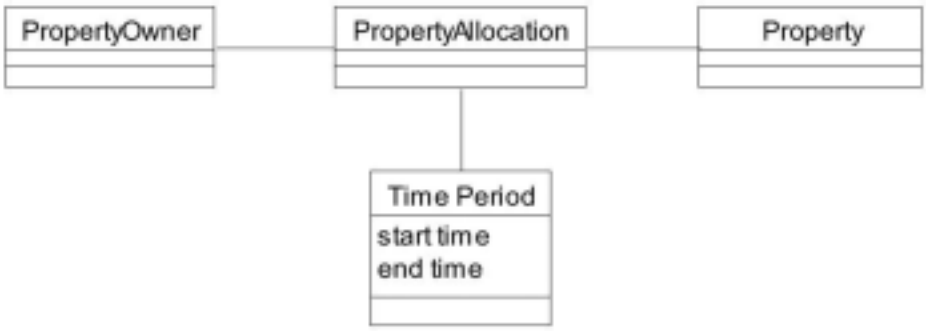


图8：用PropertyAllocation对象实现Temporal Property模式

在这里你可以确定：Property实例不能在没有PropertyOwner实例存在的情况下单独存在；如果只有一个PropertyOwner类，Time Period可以直接附着到专用Property对象（DedicatiedProperty）上。具体的情况如图9所示。

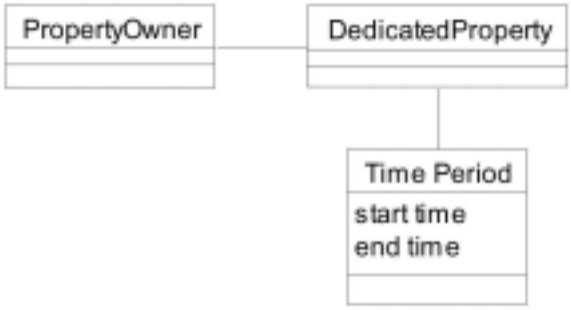


图9：Temporal Property模式实现专用属性（dedicated property）

作为一种选择，你可能发现使用一个特制的Temporal Collection类对实现本模式是有用的。当不能使用关系数据库时，这种实现方式是最合适的。

使用关系数据库实现本模式时，你通常需要提供一个类和相应的数据表（table）来保存表现类关联所需的信息。但当可以使用专用属性实现时，这也是可以避免的。在这种案例中，你可以把时间域包含在属性类/表中。如果拥有者对象（owning object）和专用属性对象之间的关系总是一对一的，情况还可以进一步简化。在这种情况下，你可以假设一个属性（旧的）的结束时间是下一个（新的）属性的开始时间，于是你可以免去对结束时间的储存和维护。

已知应用

AT&T Rialto™套件的当前实现使用时效模式来表现几个关联，未来这个模型被扩展到新的领域时，

更多的关联可以被加入其中。

User（用户）与User Identity（用户身份）的关联图解说明了Temporal Property模式的专用属性实现形式（如图10所示）。

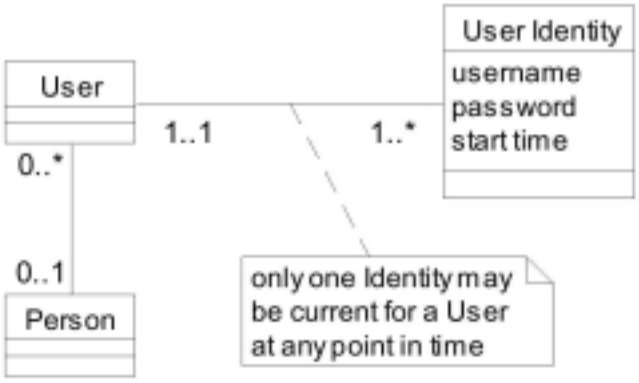


图10：User和Identity模型

用一个与User分离的类保存用户身份，于是用户可以改变他（或她）的用户名，而系统不会失去对这个动作的追踪，这样用户依然是同一个用户，关于这个用户的任何信息（例如义务或权利信息）都不会受到影响。这是我们在“实现”一节中提到的简化的一个例子：我们假设User Identity没有结束时间，下一个身份的创建时间就是上一个身份的结束时间。

我们再使用版型技术，得到的模型如图11所示：

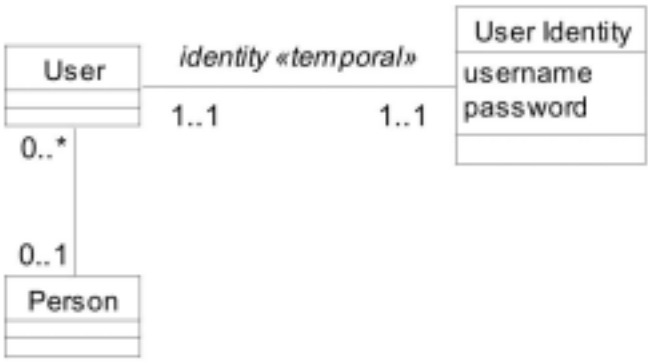


图11：使用版型得到的User和Identity模型

现在这个模型表现了前一个图的信息，而在图10中需要的注释，图11不再需要了。

一个Tariff（价目表）对象有一个PricePlan对象表现当前的价格信息，这也展现了本模式的专用属性形

式（如图12所示）。

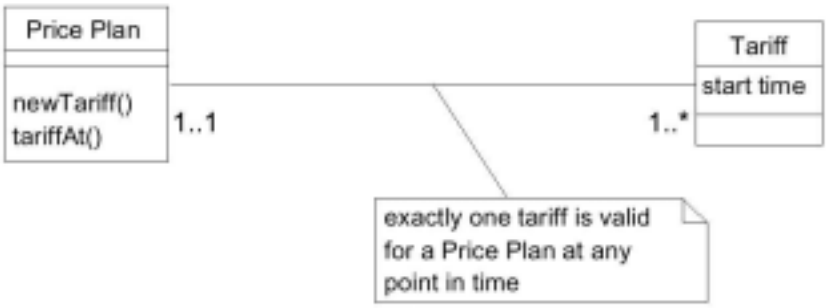


图12：价目表模型

再使用版型，图12变成了图13的样子。

图13：使用版型之后的价目表模型

请注意，本案例中的注释没有被拿掉，但其中的内容发生了改变。通过观察关联的元组数目，我们可以看出任何时间的具体情况。需要有一个新的注释，是因为关联的双向性。如果没有这个注释，我们就无法告诉客户“temporal property模式不能同时在两个方向上工作”。

相关模式

本模式是对Martin Fowler的*Historical Mapping*模式[FOWLER97]的发展。

Time Period的抽象是Martin Fowler的*Range*模式[FOWLER97]的一个例子。

本模式与*Temporal Association*模式有着紧密的关系，如果你发现自己面前有一个与这一特性有关的问题，也许你应该认真考虑究竟应该使用哪个模式。*Temporal Association*模式适用的情况是：我们（从建模角度上）对关联中间对象感兴趣。请注意，就象我们这里的例子，如果只有一个中间对象，我们不排除使用*Temporal Property*模式的可能，因为这个中间对象可能仅仅是因为具体实现的需要才存在的（例如关系数据库中的一个“交叉引用”表）。

很可能你会发现同时有多个关联和/或属性随时间发生了变化。这种情况下，你可能同时需要*Temporal Association*模式和*Temporal Property*模式。

如果在使用*Temporal Property*模式时你还希望观察一个或多个属性，并且将历史信息移除，你可以考虑使用*Snapshot*模式。当你使用多个*Temporal Property*和/或*Temporal Association*和*Temporal Property*的混合时，*Snapshot*模式可能特别适用。

在使用*Temporal Property*模式表现属性值未来的变化之前，你应该先问问自己：此属性将怎样保存未来的值？观察未来的属性值，你会发现至少有两种方法。第一种：假设属性会在未来某个合适的时刻得到新的值，或者属性值可用于确定一个即将来临的属性变化时间表。第二种，假设未来的值是对未来可能发生事情的估计或计划。在这种情况下，同一个属性可能有几个可选的计划。*Temporal Property*模式只准备第一种选择，也就是未来的值会用于判断属性怎样随时间变化。如果你面对估计或计划的问题，你可以考虑换用Matin Fowler的*Plan*模式[FOWLER97]。

模式二：Temporal Association（时效关联）模式

别名：关联对象（Association Object）

环境

假设你正在建设一个复杂信息系统，其中一些对象的关联必须能随时间而改变。你需要能够跟踪：1、关联状态在过去如何改变？或2、关联状态在未来将如何变化？或3、上述两点。你也可能需要用一些数据库（可能是关系数据库）实现这个系统。

问题

本模式的问题与Temporal Property模式的问题是相似的，即：怎样为现存的模式添加时间的表示方法。但是在这个案例中我们对一个更复杂的情况感兴趣，这里总有一个表现关联的中间对象。这个中间对象可能是一个“第一类（first class）”对象，也可能是《分析模式》[FOWLER97]一书中讲到的“关联类（association as class）”、“连接属性（link attribute）”和“相关类型（associative type）”。

作为一个实际案例，请考虑图14所示的模型。

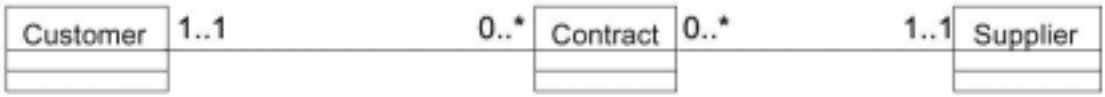


图14：Contract（合同）模型

举个例子，如果我们现在希望知道一个合同是否已经到期或者尚未开始发生效力，我们可以为Contract类添加时效信息（如图15所示）。

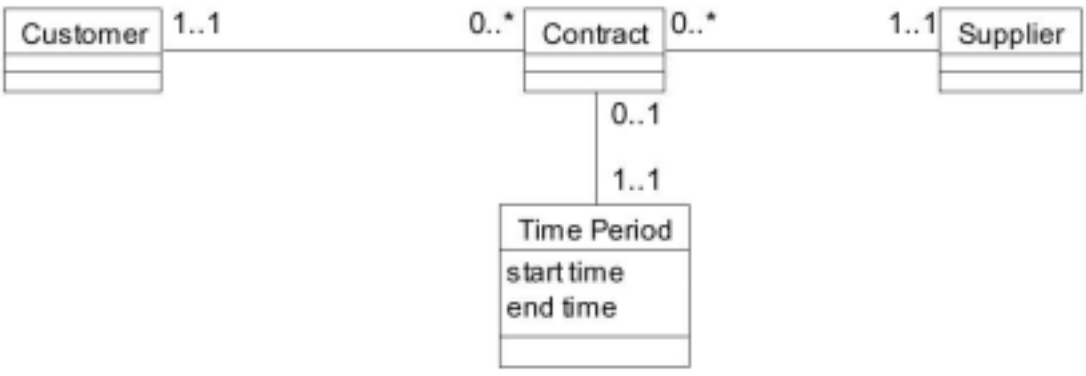


图15：包含有时效信息的Contract模型

乍一看，图15和Temporal Property模式中讲到的Employee-EmployeesSkill-Skill关联是一样的，所以你可能会问：“在这里使用这个模式，合适吗？”——这个问题是很有道理的。很显然，如果把Contract类从模型中排除出去（使用版型）并且在Customer和Supplier之间建立直接联系，这个模型就不能继续工作了。为什么会这样？我们可以这样猜测：Contract对象携带着其他的信息（比如发票数据），因此将它从模型中移除是不合适的。

Contract类的重要性是来自Supplier类接口的，这个接口如图16所示：

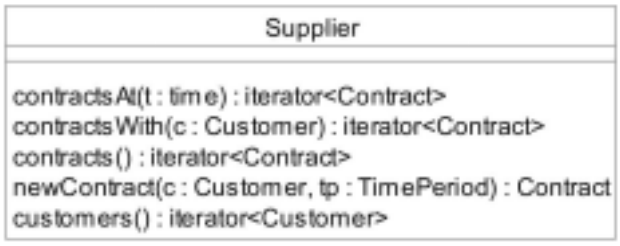


图16：Supplier接口

由此可以推测：Customer类可能有一个相似（但可能功能更少）的接口。

与Temporal Property模式形成对比，我们不必因为添加时效信息所需的结构而使模型变得复杂化。

约束

- **简单的模型更具吸引力**——开发者通常都有“保持事情简单（keep things simple）”的愿望。他们希望在设计时使用最小数量的类和关联，并且尽量不关心时效方面的问题。

- **简单的模型有时候是不够的**——有时候，的确会有变化跟踪（change track）的需要（对于对象的过去和未来的跟踪）。

- **复杂的模型将不那么清楚**——模型的本质可能会被时效部分隐藏起来。

- **变化的影响**——类之间关系的表现形式会发生改变，这会导致的代价是：所有的客户——如果他依赖于对此表现形式的了解——都必须做相应的改变。

- **对称性是有吸引力的**——如果与Temporal Property已采用的建模约定相似，本模式的建模约

定将更受欢迎。

解决方案

与Temporal Property模式形成对比，本模式的关联中已经存在一个中间对象，我们要追踪它随时间的变化情况。因此，我们不必再引入新的类或关联，而是直接把Time Period信息直接添加到中间对象上。于是我们得到图17所示的结构：

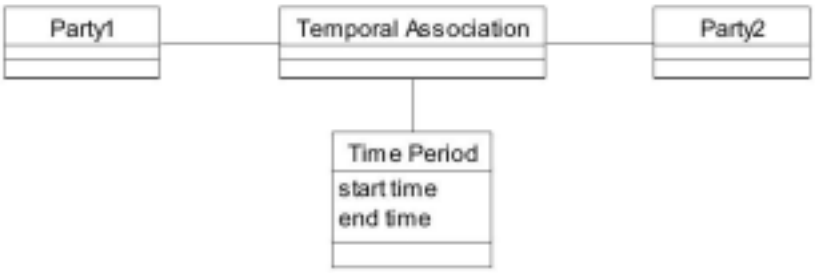


图17：Temporal Association模式结构

因为Association对象有其自身的重要性，所以通过使用版型来简化模型是没有用的。

效果

√ 我们为原始模型添加了表现属性随时间变化情况的能力。

√ 通过在关联中将Time Period对象的属性与中间对象结合起来，我们可以在一个关系数据库中表现时效。

√ 时间信息的“拥有者”由与之相关的对象保存。

√ 客户对关联的了解不需要超过原有对象的范围。

√ 通过向客户暴露关联两端的对象接口，中间对象保留了自己的重要性。

√ “未来关联（future relationship）”将变成“当前关联（current relationship）”、以后将变成“历史关联（historical relationship）”，这些变化都将随时间流逝而发生。客户不需要对保存的任何信息进行任何更新。

× 本模式实现的弹性（flexibility）不如Temporal Property模式。

× 本模式中不能使用版型，因此无法在这方面与Temporal Property模式做比较。

实现

在用关系数据库实现本模式时，最好不要用一个独立的对象（以及独立的数据表）保存Time Period，因为这样没有实际用处。将Time Peirod属性与中间对象结合起来，你可以减少数据表的数量。于是，Contract类会变成图18的样子。这样你还可以减少原始模型中类的数量。

图18：结合了Time Period属性的Contract类

如果在实现时使用关系数据库并将“开始时间”作为键信息（key information），很可能你必须为从属于主表的每个实体（例如：合同的一些条款）引入“开始时间”属性，因为你必须确保引用一致性。另一种选择方案是为主表添加一个列，这个列由主表的每个行的唯一身份标志组成，然后在从表中包含这个列并将它作为键的一部分。

已知应用

在“问题”一节中引用的Contract示例是AT&T Rialto™套件当前实现的的简化版本。

S3+™——Policy Management Systems Corporation （PMSC）开发的一个财产、意外保险系统支持软件，其中的保险费和客户应用程序在底层关系数据库中使用了Temporal Association模式。有效日期和终止日期被包含在中间对象中，这些中间对象包括保险单、保险单-客户关联、保险单的赔偿标准、保险总额等。

相关模式

本模式是对Andy Carlson的Chronology模式的发展，在ChilipLoP 98的一个作者讨论会上曾讨论过Chronology模式[CARLSON98A]。本模式与Lorraine Boyd的Association Object模式[BOYD97]也有相似之处。

本模式与Temporal Property模式关系相当紧密，当你发现面前的问题与这一联系有关时，你应该认真考虑：究竟应该使用哪个模式？Temporal Property模式关心的情况是：我们（从建模的角度）对关联的中

间对象不感兴趣，这可能是因为并没有中间对象，也可能中间对象仅仅是因为一个特定的实现而存在的。

很可能你会发现同时有多个关联和/或属性随时间发生了变化。这种情况下，你可能同时需要Temporal Association模式和Temporal Property模式。

如果在使用Temporal Association模式时你还希望观察一个或多个属性，并且将历史信息移除，你可以考虑使用Snapshot模式。当你使用多个Temporal Association和/或Temporal Association和Temporal Property的混合时，Snapshot模式可能特别适用。

模式三：Snapshot（快照）模式

别名：无

环境

假设你在创建使用一个复杂信息系统的客户程序，你使用的信息系统中的对象属性和/或关联能够随时间发生变化。你的创建的客户程序应该能够追踪：1、这些变化在过去如何发生？或2、在未来将发生什么变化？或3、以上两点。另外，你创建的客户程序中，有些对信息的变化感兴趣，另一些则不感兴趣。

问题

为对象模型引入时间表示有很多种方法，其中包括追踪属性（例如：工资）或关联（例如：合同）的离散变化、或者属性的连续变化（例如：温度）。“相关模式”一节中给出了一些例子。在上面的两个模式中，我们已经给我们的模型赋予了表示时间的能力，现在我们希望使用这一能力。

在考虑如何使用这个模型的时候，将我们感兴趣的时效部分与信息的存储（和表示）相分离，这样会给我们带来一些便利。很自然的，我们把信息的使用者考虑为 客户（client） 。这些客户可能是：

- 综合的信息处理程序。
- 需要周期性输入的其他系统或本模型的接口。

让我们再次考虑Temporal Association模式中的Contract的例子（如图19所示）。Temporal Association模式准许我们表现Contract的有效时间范围，并且回答了“供应商X在1998年1月1日有什么合同”这一类的问题。

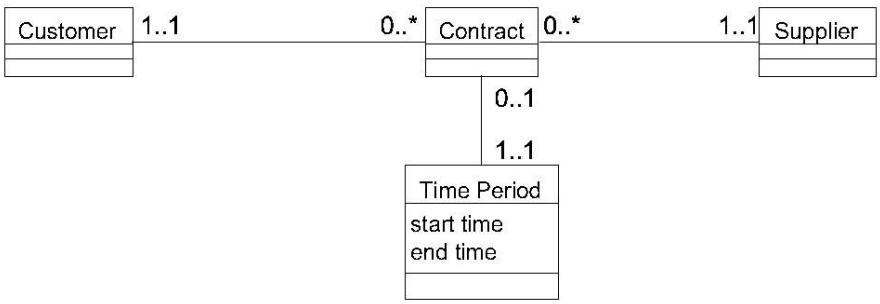


图19：Contract的例子

现在设想我们正在开发一个帐目系统。很明显，获得合同信息是这样一个系统的重要需求。具代表性的，一个帐目系统需要在操作过程中查询几个时间点以了解信息的变化。一个好的帐目系统的另一个重要特性是，它不会给一个合同已经结束（或尚未开始）的客户寄发票。因此我们必须聪明些，在我们的设计中合并一些东西，就象Temporal Association模式那样。

现在假设我们已经为我们的合同信息及其他属性和关联（例如可以随时间变化的帐单地址）实现了一个足够的时效模式，这个帐目系统也就把我们的处境弄得相当困难了。我们为帐目系统引入了时效模型，那么时效模型是否要求我们的系统遍历所有的时效关联以寻找它感兴趣的时间点呢？当然我们可以用方便的接口（例如：Supplier.contractsAt(time)）把这些复杂性包装起来，但是我们的系统还是需要做遍历的工作。如果——这是经常发生的——帐目处理的不同阶段都需要访问合同信息，我们就会一遍又一遍的重复同样的复杂的遍历工作，尽管实际上只有一次交易。其实我们很可能只关心一个时间点的关联，当然这个时间点也就是整个交易过程的终结点。

时间元（time dimension）的引入会把事情的很多方面都变复杂，这不仅仅是对于模型设计者和开发者，对于模型的客户也是一样。如同我们在上面的例子中看到的，一个或多个客户可能希望对某个特定的重要的时间点（例如交易终结点或年终）的信息执行一些复杂的处理。作为一种降低复杂度的方法，一些客户（特别是扩展系统）可能有一个特别的“视图（view）”：他们请求的信息可能不包括时间元（例如请求一个周期性的信息输入，其中只包含当前的信息，而不包含信息改变的历史）。

约束

- **简单的模型有时候是不够的**——有时候，的确会有变化跟踪（change track）的需要（对于对象的过去和未来的跟踪）。

- **复杂的模型将不那么清楚**——模型的本质可能会被时效部分隐藏起来。

- **变化的影响**——类之间关系的表现形式会发生改变，这会导致的代价是：所有的客户——如果他依赖于对此表现形式的了解——都必须做相应的改变。

- **性能**——存在消耗（我们很希望避免）：在同一时间点重复遍历时效关联，以找到所需要的状态。

- **单纯的客户**——一个或多个客户或扩展系统可能不接收历史数据。

解决方案

修改客户通常使用的接口，添加方法以允许向客户提供一个叫“Snapshot（快照）”的新对象。Snapshot对象为一个或多个属性或关联提供去掉了时间元和历史信息的视图。对于那些申请时间信息的客户，Snapshot对象把时间作为一个可见的属性提供给它。于是，客户可以反复使用Snapshot对象，或者把它传递给其他的客户。

与Temporal Property模式相结合的Snapshot模式的结构如图20所示。请注意，本模式同样适用于其他的时效模式（参见“相关模式”）。

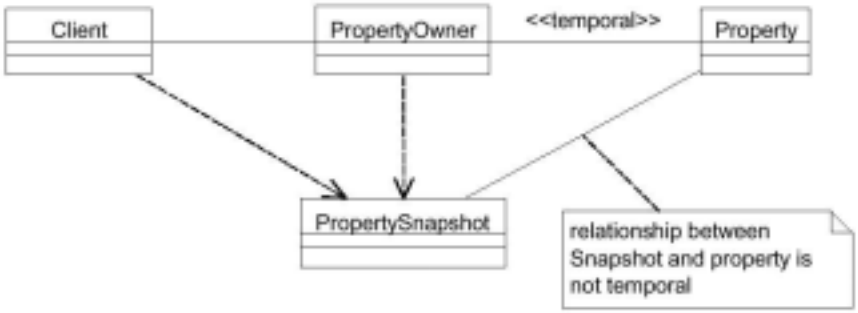


图20：Snapshot模式的结构

本模式的协作关系如图21所示。

图21：Snapshot的协作

首先，客户向PropertySnapshot的拥有者指定一个时间。如果Property Owner没有客户请求的时间的快照，它就创建一个新的Snapshot对象。然后，PropertySnapshot根据与自己相关的时间向Property Owner请求属性值。PropertySnapshot对象被返回给客户，客户就可以不必经过重新计算有效性而重复得到属性值。而

且客户可以把PropertySnapshot对象传递给另一个不知道时效性质的客户。

在Temporal Property模式的案例中，我们将改变PropertyOwner接口，添加propertySnapshotAt()方法（如图22所示）。

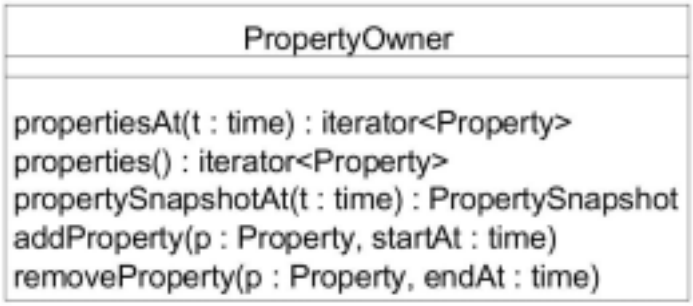


图22：加入了Snapshot的PropertyOwner接口

PropertySnapshot接口可能会象图23这样：

图23：PropertySnapshot接口

PropertySnapshot接口不提供任何新的操作。但是，这些操作与PropertyOwner中对等的操作有两点重要的区别：

- 实现不需要遍历时效信息以确定结果，因此节省了开销。
- PropertyOwner接口总是应用于一个默认时间（可能是当前时间），而PropertySnapshot接口可以适用于客户指定的任何时间。

为了使用一个更具体的例子（来自Temporal Association模式），Supplier接口可能变成图24所示的样子：

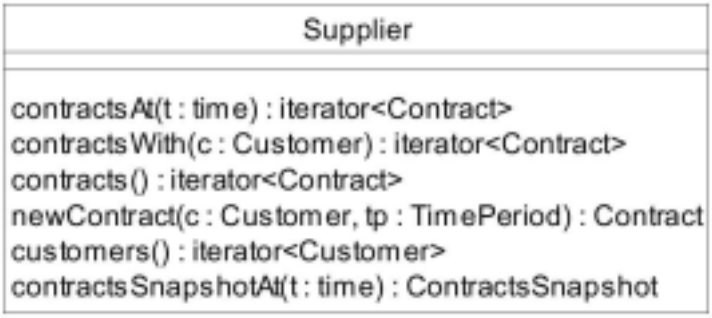


图24：结合了Snapshot的Supplier接口

ContractsSnapshot类可能是象图25这样的。请注意，在本案例中，Snapshot支持多个查询操作。

图25：ContractSnapshot接口

效果

√ 使用Snapshot排除了对时效模型重复遍历的开销。

√ 不知道或不使用历史信息的客户也可以使用时效模型。

√ 客户可以使用简单的接口，不必在接口中加入默认时间的限制。

√ Snapshot的使用是可选的——客户仍旧可以按照原始时效模型提供的接口访问时效信息。

√ 如果Snapshot被保存下来，它们就可以为客户提供永久的信息记录。客户可以在这个基础上继续处理自己关心的信息，而不必操心原始时效信息的变化。

× 引入了一个新的类，有一些功能有重复。

实现

你可能也希望为基本的Snapshot机制实现一些改进——如果这些改进对你有用的话。

- 如果可能有多个互不依赖的客户同时请求同一时间点的Snapshot对象，你可以在接口中加入

缓存的功能，让客户可以随时访问。

- 由于属性或关联是以离散的方式发生变化的，与其让Snapshot对象保存它的时间点，还不如为它计算它的有效时间段。这样的做法可以提高缓存（cache）的命中率。[4]

已知应用

AT&T Rialto™ Rater使用本模式在交易终结时间“冻结”复杂时效关联。现在的主要应用是Tariff子系统，这个子系统中有多个时效关联，允许整个Tariff或其中一部分随时间变化。

Rialto™套件还有一个名叫Access Control的组件用于实时网络用户认证，这个组件的设计思想以保证访问速度为主旨。该组件从一个存储了丰富的时效模型的仓库（repository）中接收信息。Access Control组件对仓库中的历史信息不感兴趣，而对历史信息的遍历会严重降低用户认证的吞吐量。因此数据以Snapshot的形式提供给Access Control组件，以确保Access Control总保有已认证用户的当前映像。

S3+™——Policy Management Systems Corporation （PMSC）开发的一个财产、意外保险系统支持软件，其中的Claims应用程序在其底层关系数据库中使用了Snapshot模式。在意外发生的时候，保险单相关的信息（例如保险额和保户对象）被作为快照。然后Claims应用程序使用这个快照，因此不必遍历时效关联，应用程序也可以访问访问意外发生当天的保险单相关信息。

CyberLife® Administration，由CYBERTEK（PMSC的一家公司）开发的人寿保险管理软件，也在其底层关系数据库中使用了Snapshot模式。在被保人逝世的日期，系统保存一个保险金总额的快照。

相关模式

本模式是对Andy Carlson的Snapshot模式的发展，在ChilipLoP 98的一个作者讨论会上曾讨论过Snapshot模式[CARLSON98A]。

本模式可以被大多数加入了时效信息表示的模式使用。实际的例子包括Temporal Property模式和Temporal Association模式。

---

4 译者注：实在很对不起大家，这一句我真的译不出来。请各位看看，原文如下：This could dramatically improve the hit rate in the cache, depending on the typical duration between changes in the state and the chronological distribution of request for *snapshots*.

参考书目

[BOYD97] L Boyd. *Business Patterns of Association Objects* from *Pattern Languages of Program Design* 3. R Martin, D Riehle, F Buschmann (eds.). Reading, MA: Addison Wesley Longman, 1998.

[CARLSON98A] Andy Carlson, ChronologyPattern.

[CARLSON98B] Andy Carlson, Snapshot Pattern.

[FOWLER97] Martin Fowler. *Analysis Patterns : Reusable Object Model*s. Menlo Park, CA: Addison Wesley Longman, 1997.

[RUM91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object Oriented Modeling and Desig*n. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

致谢

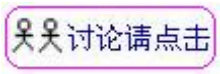
我们要感谢我们的导师Dirk Riehle，他为我们这篇文章做了很有用的注释。

注册商标

关键字

时间（time）、时效（temporal）、历史（history）、关联（relationship）、属性（property）、快照（snapshot）

原文链接 http://www.shecn.com/zippdf/carlson98temporal.zip



Java 中的 Singleton (上)

石一楹

Singleton模式

在面向对象的程序中，某些类只需要一个实例。譬如，在一个窗口应用程序中，我们只需要一个主窗口。又如在一个数据库应用程序中，我们往往希望将所有的数据库连接集中于一处，并能为整个程序所使用。

最初，我们很容易考虑使用静态成员变量，但是这种语言本身提供的机制并不能阻止我们对类本身进行多次实例化。

再深入考虑，假设开始我们对某一个数据库系统只有一个连接许可，我们可以在Singleton类中控制它只允许建立一个连接对象。当数据库系统允许我们进行多个连接的时候，我们只需控制Singleton内部的连接建立方法，而外部的程序仍旧可以从这个单一点来访问它能获取的数据库连接，虽然这时候Singleton可以建立多于一个的对象。

也就是，Singleton的目的是控制类实例对象的创建，并且允许整个程序只在一点对它进行访问。Singleton本身类只能创建一个，可是它具有很好的适应性可以在情况发生变化时建立多个对象。这种特性使得Singleton模式经常用于控制对系统资源的控制。

有状态和无状态Singleton

Singleton可以有状态，假设我们需要产生一个全局有效的序列号，如1，2，3….,我们必须保证每一次对Singleton的请求均有Singleton保持状态，当下一次请求来到时，Singleton接口能提供一个连续不重复的序列号。

无状态的Singleton则类似于非面向对象编程中的实用函数，java.util包中有大量这样的类，第一次存取类和下一次存取类之间毫无关系。

实现Singleton

Singleton 类的实现首先要保证它只能有一个唯一的实例，从语言本身的角度来讲，这个实例和其他类的实例没有任何区别。

最简单的方法如下：

```
public class Singleton {
protected Singleton() {
// 构建函数. . .
}
public static Singleton getInstance() {
return _instance;
}
private static Singleton _instance = new Singleton();
//类的其余部分
}
```

如果再精雕细琢一下，我们可以实现如下：

```
public class Singleton {
protected Singleton() {
// 构建函数. . .
}
public static Singleton getInstance() {
if (_instance == null)
_instance = new Singleton();
return _instance;
```

}

private static Singleton _instance=null;

//类的其余部分

}

这里，_instance在需要的时候才被创建和保存，我们只需保证Singleton在它的首次使用之前被创建好即可。

注意我们将Singleton的构建函数声明为protected。如果Singleton的客户直接实例化一个Singleton对象，编译程序就不能通过。我们用这种方法确信实例只能被创建一次。当然，存取函数getInstance必定要被声明为static，因为客户不可能直接去实例化Singleton，所以只能通过static成员函数对这个类进行存取。存取static成员函数可以让java加载类。

如果不使用singleton，而是采用一个全局或静态对象，那么至少有以下一些缺点：

1. 不能保证静态或全局变量只被实例化一次；
2. 不管你是否使用这个变量，都需要首先将建立这个对象

Singleton类允许创建子类。但由于getInstance()是一个静态方法，所以子类不能覆盖getInstance()方法。而Singleton的作用是保证实例的创建的唯一性，所以事实上，这里的继承子类就是为了能够控制子类创建的唯一性，因此，Singleton中_instance私有成员必须用对应的子类进行实例化，而getInstance()是实现这一过程的最佳位置，下面的方法显示了使用环境变量实例化子类：

public class GenericSingleton {

protected GenericSingleton() {

}

public static GenericSingleton getInstance() {

if (_instance == null) {

String style =getEnv("style"); //getEnv没有实现，表示取环境变量

if (style.equals("son"))

```
_instance = new SonSingleton();
else if (style.equals("daughter"))
_instance = new DaughterSingleton();
else
_instance = new GenericSingleton();
}
return _instance;
}
private static GenericSingleton _instance=null;
}
public class SonSingleton extends GenericSingleton {
}
public class DaughterSingleton extends GenericSingleton {
}
```

GOF已经指出这种方法的弱点，不管何时GenericSingleton需要有一个新的子类，它的getInstance方法必须被修改。如果GenericSingleton是一个框架中的抽象工厂(Abstract Factory),那么这种情况几乎是不可避免的。所以我们可能会选择注册表的方法.

注册表的想法也很简单，我们不让getInstance方法显式定义可能的GenericSingleton和它子类的集合，而是在加入一个子类时，用对应的子类名字在一个注册表中注册它们对应的实例。如此，通过这个字符串名字和对应的子类对象之间建立映射关系。当客户应用程序调用getInstance来获取一个Singleton时，按照名字在注册表中查找，并返回此实例。

```
import java.util.*;
class Singleton {
public static void register(String name, Singleton singleton) {
```

```
_registry.put(name, singleton);
}
public static Singleton getInstance() {
String singletonName = getEnv("SINGLETON");
Singleton _instance = lookup(singletonName);
return _instance;
}
protected static Singleton lookup(String name) {
return (Singleton)_registry.get(name);
}
private static Map _registry = new HashMap();
};
```

这种实现方法让用户在"SINGLETON"环境变量中指定要获取的子类名字。那么子类何时在此注册表中注册自己呢？最合适的地方是它自己的构建函数：

```
class MySingleton extends Singleton {
MySingleton() {
super();
register(getClass().getName(),this);
}
}
```

问题的关键是如果不实例化MySingleton子类，这个构建函数就不会被执行。GOF在Design Patterns一书中指出，可以用静态实例，但在Java中我们需要在某处显式初始化。

```
static MySingleton theSingleton=new MySingleton();
```

下次，我们只要给定环境变量SINGLETON的值，就可以如下获取singleton的对应子类，例如：

```
public class Test {

public static void main(String[] args) {

System.out.println(Singleton.getInstance().getClass().getName());

}

}
```

保证Singleton的唯一性

Singleton模式依赖于它的唯一性，但是在特定的情形下，我们上面的努力并不能保证它的唯一性，我将在此列举那些已经被模式社团所发现的情况，并指出它们的解决方案：

在两个或更多的虚拟机中存在多个Singleton

在分布式计算环境如EJB,Jini或RMI中，对象的创建透明于客户程序。假设你使用一个无状态会话EJB来实现Singleton,那么在你两次不同的调用之间，EJB容器可以随时丢弃、重新生成这个无状态会话EJB.它完全可能在不同的虚拟机中重新启动这个EJB,但客户程序却丝毫不知情。甚至对实体EJB而言，在你的多次调用之间也完全可能被持久(persistent)于磁盘或数据库，然后下次在另外一个虚拟机中被加载。

因此，我们不能保证Singleton能够象在本地VM中这样，对它的成员属性和成员函数保持唯一性。

所以，如果你在这样的分布式环境中使用Singleton模式，请务必保证你的Singleton是无状态的。更进一步，用EJB来实现一个无状态的资源管理Singleton也是不适合的，这些本来就是EJB容器的责任。

由不同类加载器同时加载的多个Singleton

多层计算结构对Singleton模式有很多重要的影响。在GOF的design patterns一书中并没有对这些上下文进行分析。除了上面的分布式应用程序中多虚拟机的麻烦外，在同一个虚拟机内部还可能有多个类加载器的问题。

当两个类加载器(class loader)加载一个类时，实际上你就有此类的两个备份，在我们的Singleton情景下，这意味着每一个类都可能产生自己的Singleton实例。某些servlet引擎中运行的servlet就有这个问题，如iPlant对每个servlet都有自己的类加载器。如果有两个servlet存取同一个Singleton,就可能产生两个Singleton实例。

类加载器发生的问题可能远远超出你的想象。浏览器需要从网络上下载applet类到本地，对于每一个

来自不同服务器地址的Applet类，它都使用一个独立的类加载器。类似地，Jini和RMI系统可能对它们从不同code base下载而来的类文件使用不同的类加载器。或者你可能使用定制的类加载器，那么同样的事情也会发生。

如果由不同的类加载器加载，相同名字两个类，即使包名字相同，也会被当作不同--事实上，就算它们完全一模一样也是如此。也就是，Java的命名空间除了类名、包名外，还有加载器。对于我们的Singleton而言，它们在不同的类加载器中都有各自的实例，于是就破坏了唯一性原则。

Singleton类被垃圾收集器销毁，然后被重新加载

当程序没有任何对象包容对Singleton对象的引用时，java VM会在适当时候自动销毁此Singleton对象。但是如果接着程序中另外一个对象需要Singleton引用，Singleton就会被重新加载。当一个Singleton类被垃圾收集然后又被重新加载后，就会产生一个新的Singleton实例。你可以清楚地看到，在这之前对Singleton任何静态成员的修改早已丢失，所有内容都会在第二次加载时重新初始化。

事实上，情况随着Java版本的变化，还有其它更复杂的情景出现：

Java 1.0 和 1.1中的Singleton

所有的Java标准都要求我们保持一个对singleton对象的引用，不然它就会被垃圾收集。但是，在Java1.2之前的Java标准中，垃圾收集器不但收集singleton对象，它甚至把Singleton类都收集了！！对一个类的垃圾收集也被称为“类卸载”。

实际上，Java1.0.x没有实现类卸载，所以我们的Singleton实现运行起来很正常。但是到了Java1.1.x，类卸载已经实现，然后我们就有了麻烦。

Java1.2中的Singleton

Sun听到了很多类似这样的抱怨（Collection框架的设计有时简直离谱），Sun在Java 2中对标准做了不少修改。

关于是否卸载一个类的新规则如下：

所有通过本地加载的类，系统类加载器从不卸载。

所有通过其它类加载器加载的类只在那个类加载器被卸下以后才被卸载。

就算如此，就象我们一开始所说，对象被垃圾收集的问题还是一个麻烦，我们一般会如此解决：

1. 在main()方法（或者是多线程中的run()）中使用一个局部变量保持对singleton的引用。

2. 在你定义你main（run）方法的类中使用一个实例变量保持对singleton的引用

这些技术可以解决很多问题，但是如果你使用第三方库而又不能获得singleton，那么你就惨了。

所以，请记住，如果你的程序需要长时间运行，而它又频繁地重新加载类（如从一个远程类加载器而来的类）那么你一定要小心保持引用。

有意的Singleton类重载

类重载不一定只会发生在类垃圾收集之后；有时会应Java程序的请求而被重载。servlet 标准允许servlet引擎任何时候都可以这样做。当servlet引擎决定卸载一个servlet类时，它调用destroy(),然后将此servlet类丢弃。以后，servlet引擎会重新加载此servlet类，实例化servlet对象，然后调用init()进行初始化。实践中，这种卸载和重载进程可能会在一个servlet类或者JSP发生变化时出现。

类似于前两种情况，这种情况也会导致一个新加载的类。依赖于servlet引擎的不同，当老的servlet 类被卸载时，相关的类可能被丢弃或不被丢弃。所以如果一个servlet类有一个对Singleton的引用，你可能会发现有一个Singleton对象和老的servlet类相关连，另外还有一个Singleton对象和新的servlet类相关。

Servlet引擎的类加载机制因产品而异，所以Singleton的行为几乎是不可预期的，除非你对你所使用的servlet引擎的类加载机制十分清楚。

错误的同步造成多个Singleton实例

在最开始的Singleton中，我们使用lazy initialization来建立实例。也就是说，实例并非在类被加载的时候建立，而是在第一次使用时建立的。采用这种方法的最大问题可能是不顾及同步的问题，这样可能导致多个Singlton实例的产生。

这是我们最初的版本：

```
public class Singleton {

protected Singleton() {

// 构建函数. . .

}

public static Singleton getInstance() {
```

```
if (_instance == null)
_instance = new Singleton();
return _instance;
}
private static Singleton _instance=null;
//类的其余部分
}
```

因此，我们需要考虑同步的问题，下面是一种正确的解决方案：

```
public class Singleton {
private static Singleton _instance;
protected Singleton() {
// construct object . . .
}
// For lazy initialization
public static synchronized Singleton getInstance() {
if (_instance==null) {
_instance = new Singleton();
}
return _instance;
}
// Remainder of class definition . . .
}
```

在多线程环境下，如果不考虑同步问题，两个线程可能会同时去获取instance，然后就会出现两个实例。Singleton模式的最终目的是为了给用户一个统一的单一存取点，Singleton实现必须隐藏其中的各种细节问题和复杂性，所以，我们也必须将同步问题考虑在内。

如果没有使用正确的方式进行同步，多个Singleton实例的情况还是会出现，下面的解决方案只在调用构建方法时使用synchronized(this)块：

```
// Also an error, synchronization does not prevent
// two calls of constructor.
public static Singleton getInstance() {
if (_instance==null) {
synchronized (Singleton.class) {
_instance = new Singleton();
}
}
return _instance;
}
```

如果第一个线程发现_instance为空，然后它进入if之中，假设这时此线程被挂起，第二个线程也进入，它发现_instance还是为空，它也进入if之中，这时不管上一个线程是否在synchronized (Singleton.class)处被同步，也就是不管这两个线程谁能先开始往下执行，它们都将创建一个Singleton实例，从而破坏Singleton实例的唯一性。

也许你会想到可以在同步块内部在进行判断此实例是否为空，这叫做Double-Checked Locking方言，可惜的是，不管采用何种修正方法，最后都不能成功，我将在本章的最后一节详细讨论这个Idiom。唯一正确的方法是如上面黑体所示同步整个getInstance方法。

子类化Singleton产生的多个Singleton实例

在本章的最开始，我们已经描述了子类化一个Singlton所采用的方法。由于需要子类化，singleton类的构建函数用protected声明，而它的子类则将其声明为public,正如我们前面所描述的，我们需要在某个地方

实例化这个子类，以保证它可以被注册到Singleton内部的注册表中。但这同时也导致其它人可以自己建立这个子类的其它实例。

一个被要求建立多个对象的工厂创建多个Singleton实例

我们之所以不直接使用静态方法而采用Singleton模式的重要原因之一是，Singleton模式除了能够控制唯一的存取点之外，它在你需要多个对象时能够改变Singleton的内部实现而不涉及客户代码。

例如，许多servlet在它们的servlet引擎内作为Singleton运行。因为这可能会引起线程相关问题，一种方法是servlet可以实现SingleThreadModel.。此时，servlet引擎可以（如果必要的话），创建多于一个的实例。如果你惯于使用一般的Singleton servlet，你可能会忘记某些servlet可能会有多个实例。

串行化和重新读取会产生Singleton对象的备份

如果你有一个已经被串行化的对象然后读取两次，你就会获得两个不同的对象，而并不是对同一对象的两个引用。

Java.io包并不是java中仅有的对象串行化技术。现在已经开发了很多使用XML的对象串行化机制，包括SOAP,WDDX等等。在使用这些技术时，你同样要注意Singlton的唯一性问题。

工厂系列模式产生的多个对象问题

在创建性设计模式的factory家族中，通常将工厂设计为一个Singleton。工厂使用于建立其它类实例的Singleton机制。但是如果其它的类没有类似Singleton的保护机制，那么即使工厂本身被正确唯一化，我们也不能保证其它那些类的对象一定能象我们所预期的那样唯一。

Clone造成的多个Singleton对象

上面所有的实现都有一个问题。我们知道，Java允许通过Clone来建立对象。所以如果你要防止Clone,你应该将Singleton定义成如下形式：

```
final class Singleton {
//…..
}
```

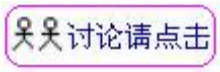
因为Singleton继承自Object,而Object中的Clone方法被声明为protected.所以客户程序不能直接调用clone，不然编译器就会产生错误。但是如果客户程序从你的Singleton中继承，实现Cloneable接口，并将Clone

覆盖声明为public.那么其它程序就可以创建多个Singleton实例对象了。

参考书目：

1. Design Pattern: Elements of Reusable Object-Oritened Software, Eric Gamma etc，ISBN 7-111-07575-7
2. When is a Singleton not a Singleton?, Joshua Fox , JavaWorld, January 2001.
3. Reality Check:Douglas C. Schmidt ,C++ Report ,March 1996
4. "Singleton's Rule" by Tony Sintes JavaWorld, December 2000:http://www.javaworld.com/javaworld/javaqa/2000-12/03-qa-1221-singleton.html
5. "Programming Java Threads in the Real World, Part 7," Allen Holub (JavaWorld, April 1999) describes threading issues relevant to Singletons: http://www.javaworld.com/javaworld/jw-04-1999/jw-04-toolbox.html
6. "The Double-Checked Locking is Broken Declaration," David Bacon, et al.: http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html
7. "Class Loaders as a Namespace Mechanism," Stuart Halloway (Java Developer Connection Tech Tips, October 31, 2000): http://developer.java.sun.com/developer/TechTips/2000/tt1027.html#tip1
8. Thinking in Pattern with Java:Bruce Eckel, http://www.topica.com/lists

建模鸡汤



Scott Ambler 著，乐林峰 译

我们期待自己成为一个优秀的软件模型设计者，但是，要怎样做，又从哪里开始呢？

将下列原则应用到你的软件工程中，你会获得立杆见影的成果。

1. 人远比技术重要

你开发软件是为了供别人使用，没有人使用的软件只是没有意义的数据的集合而已。

许多在软件方面很有成就的行家在他们事业的初期却表现平平，因为他们那时侯将主要精力都集中在技术上。

显然，构件（components），EJB（Enterprise Java Beans）和代理（agent）是很有趣的东西。但是对于用户来说，如果你设计的软件很难使用或者不能满足他们的需求，后台用再好的技术也于事无补。

多花点时间到软件需求和设计一个使用户能很容易理解的界面上。

2. 理解你要实现的东西

好的软件设计人员把大多数时间花费在建立系统模型上，偶尔写一些源代码，但那只不过是为了验证设计过程中所遇到的问题。这将使他们的设计方案更加可行。

3. 谦虚是必须的品格

你不可能知道一切，你甚至要很努力才能获得足够用的知识。软件开发是一项复杂而艰巨的工作，因为软件开发所用到的工具和技术是在不断更新的。而且，一个人也不可能了解软件开发的所有过程。在日常生活中你每天接触到的新鲜事物可能不会太多。但是对于从事软件开发的人来说，每天可以学习很多新东西（如果愿意的话）。

4. 需求就是需求

如果你没有任何需求，你就不要动手开发任何软件。成功的软件取决于时间（在用户要求的时间内完成）、预算和是否满足用户的需求。如果你不能确切知道用户需要的是什么，或者软件的需求定义，那么

你的工程注定会失败。

5. 需求其实很少改变，改变的是你对需求的理解

Object ToolSmiths 公司（www.objecttoolsmiths.com）的 Doug Smith 常喜欢说：“分析是一门科学，设计是一门艺术”。他的意思是说在众多的“正确”分析模型中只存在一个最“正确”分析模型可以完全满足解决某个具体问题的需要（我理解的意思是需求分析需要一丝不苟、精确的完成,而设计的时候反而可以发挥创造力和想象力 - 译者注）。

如果需求经常改动，很可能是你没有作好需求分析，并不是需求真的改变了。

你可以抱怨用户不能告诉你他们想得到什么，但是不要忘记，收集需求信息是你工作。

你可以说是新来的开发人员把事情搞得一团糟，但是，你应该确定在工程的第一天就告诉他们应该做什么和怎样去做。

如果你觉得公司不让你与用户充分接触，那只能说明公司的管理层并不是真正支持你的项目。

你可以抱怨公司有关软件工程的管理制度不合理，但你必须了解大多同行公司是怎么做的。

你可以借口说你们的竞争对手的成功是因为他们有了一个新的理念，但是为什么你没先想到呢？

需求真正改变的情况很少，但是没有做好需求分析工作的理由却很多。

6. 经常阅读

在这个每日都在发生变化的产业中，你不可能在已取得的成就上陶醉太久。

每个月至少读 2、3 本专业杂志或者 1 本专业书籍。保持不落伍需要付出很多的时间和金钱，但会使你成为一个很有实力的竞争者。

7. 降低软件模块间的耦合度

高耦合度的系统是很难维护的。一处的修改引起另一处甚至更多处的变动。

你可以通过以下方法降低程序的耦合度：隐藏实现细节，强制构件接口定义，不使用公用数据结构，不让应用程序直接操作数据库（我的经验法则是：当应用程序员在写 SQL 代码的时候，你的程序的耦合度就已经很高了）。

耦合度低的软件可以很容易被重用、维护和扩充。

8. 提高软件的内聚性

如果一个软件的模块只实现一个功能，那么该模块具有高内聚性。高内聚性的软件更容易维护和改进。

判断一个模块是否有高的内聚性，看一看你是否能够用一个简单的句子描述它的功能就行了。如果你用了一段话或者你需要使用类似“和”、“或”等连词，则说明你需要将该模块细化。

只有高内聚性的模块才可能被重用。

9. 考虑软件的移植性

移植是软件开发中一项具体而又实际的工作，不要相信某些软件工具的广告宣传（比如 java 的宣传口号 write once run many – 译者注）。

即使仅仅对软件进行常规升级，也要把这看得和向另一个操作系统或数据库移植一样重要。

记得从 16 位 Windows 移植到 32 位 windows 的“乐趣”吗 ？当你使用了某个操作系统的特性，如它的进程间通信(IPC)策略，或用某数据库专有语言写了存储过程。你的软件和那个特定的产品结合度就已经很高了。

好的软件设计者把那些特有的实现细节打包隐藏起来，所以，当那些特性该变的时候，你的仅仅需要更新那个包就可以了。

10. 接受变化

这是一句老话了：唯一不变的只有变化。

你应该将所有系统将可能发生的变化以及潜在需求记录下来,以便将来能够实现（参见“Architecting for Change”，Thinking Objectively, May 1999）

通过在建模期间考虑这些假设的情况，你就有可能开发出足够强壮且容易维护的软件。设计强壮的软件是你最基本的目标。

11. 不要低估对软件规模的需求

Internet 带给我们的最大的教训是你必须在软件开发的最初阶段就考虑软件规模的可扩充性。

今天只有 100 人的部门使用的应用程序，明天可能会被有好几万人的组织使用，下月，通过因特网可能会有几百万人使用它。

在软件设计的初期，根据在用例模型中定义的必须支持的基本事务处理，确定软件的基本功能。然后，在建造系统的时候再逐步加入比较常用的功能。

在设计的开始考虑软件的规模需求，避免在用户群突然增大的情况下，重写软件。

12. 性能仅仅是很多设计因素之一

关注软件设计中的一个重要因素--性能，这好象也是用户最关心的事情。一个性能不佳的软件将不可避免被重写。

但是你的设计还必须具有可靠性，可用性，便携性和可扩展性。你应该在工程开始就应该定义并区分好这些因素，以便在工作中恰当使用。性能可以是，也可以不是优先级最高的因素，我的观点是，给每个设计因素应有的考虑。

13. 管理接口

“UML User Guide”（Grady Booch，Ivar Jacobson 和 Jim Rumbaugh ,Addison Wesley, 1999）中指出，你应该在开发阶段的早期就定义软件模块之间的接口。

这有助于你的开发人员全面理解软件的设计结构并取得一致意见，让各模块开发小组相对独立的工作。

一旦模块的接口确定之后，模块怎样实现就不是很重要了。

从根本上说，如果你不能够定义你的模块“从外部看上去会是什么样子”，你肯定也不清楚模块内要实现什么。

14. 走近路需要更长的时间

在软件开发中没有捷径可以走。

缩短你的在需求分析上花的时间，结果只能是开发出来的软件不能满足用户的需求，必须被重写。

在软件建模上每节省一周，在将来的编码阶段可能会多花几周时间，因为你在全面思考之前就动手写程序。

你为了节省一天的测试时间而漏掉了一个 bug，在将来的维护阶段，可能需要花几周甚至几个月的时间去修复。与其如此，还不如重新安排一下项目计划。

避免走捷径，只做一次但要做对（do it once by doing it right）。

15. 别信赖任何人

产品和服务销售公司不是你的朋友，你的大部分员工和高层管理人员也不是。

大部分产品供应商希望把你牢牢绑在他们的产品上，可能是操作系统，数据库或者某个开发工具。

大部分的顾问和承包商只关心你的钱并不是你的工程（停止向他们付款，看一看他们会在周围呆多长时间）。

大部分程序员认为他们自己比其他人更优秀，他们可能抛弃你设计的模型而用自己认为更好的。

只有良好的沟通才能解决这些问题。

要明确的是，不要只依靠一家产品或服务提供商，即使你的公司（或组织）已经在建模、文档和过程等方面向那个公司投入了很多钱。

16. 证明你的设计在实践中可行

在设计的时候应当先建立一个技术原型， 或者称为“端到端”原型。以证明你的设计是能够工作的。

你应该在开发工作的早期做这些事情，因为，如果软件的设计方案是不可行的，在编码实现阶段无论采取什么措施都于事无补。技术原型将证明你的设计的可行性，从而，你的设计将更容易获得支持。

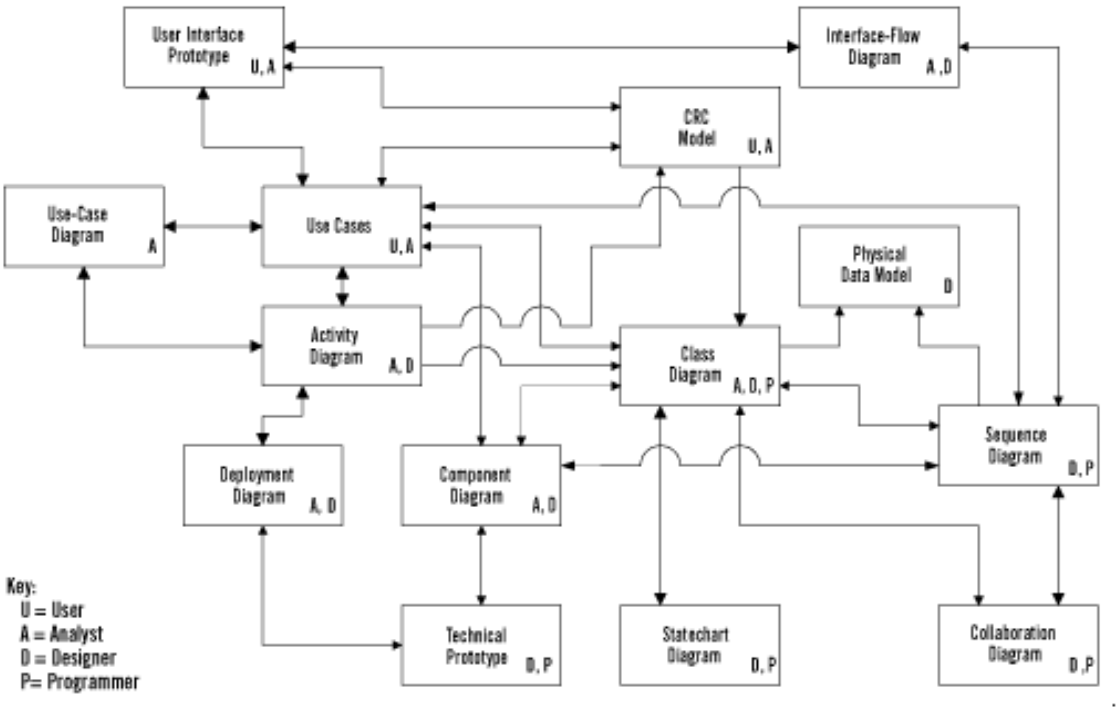
17. 应用已知的模式

目前，我们有大量现成的分析和设计模式以及问题的解决方案可以使用。

一般来说，好的模型设计和开发人员，都会避免重新设计已经成熟的并被广泛应用的东西。http://www.ambysoft.com/processPatternsPage.html 收藏了许多开发模式的信息。

18. 研究每个模型的长处和弱点

目前有很多种类的模型可以使用,如下图所示。用例捕获的是系统行为需求，数据模型则描述支持一个系统运行所需要的数据构成。你可能会试图在用例中加入实际数据描述，但是，这对开发者不是非常有用。同样，数据模型对描述软件需求来说是无用的。每个模型在你建模过程中有其相应的位置，但是，你需要明白在什么地方，什么时候使用它们。



19. 在现有任务中应用多个模型

当你收集需求的时候，考虑使用用例模型，用户界面模型和领域级的类模型。

当你设计软件的时候，应该考虑制作类模型，顺序图、状态图、协作图和最终的软件实际物理模型。

程序设计人员应该慢慢意识到，仅仅使用一个模型而实现的软件要么不能够很好地满足用户的需求，要么很难扩展。

20. 教育你的听众

你花了很大力气建立一个很成熟的系统模型，而你的听众却不能理解它们，甚至更糟－连为什么要先建立模型都不知道。那么你的工作是毫无意义的。

教给你开发人员基本的建模知识；否则，他们会只看看你画的漂亮图表，然后继续编写不规范的程序。

另外， 你还需要告诉你的用户一些需求建模的基础知识。给他们解释你的用例(uses case)和用户界面模型，以使他们能够明白你要表达地东西。当每个人都能使用一个通用的设计语言的时候（比如 UML-译者注），你的团队才能实现真正的合作。

21. 带工具的傻瓜还是傻瓜

你给我 CAD/CAM 工具，请我设计一座桥。但是，如果那座桥建成的话，我肯定不想当第一个从桥上

过的人，因为我对建筑一窍不通。

使用一个很优秀的 CASE 工具并不能使你成为一个建模专家，只能使你成为一个优秀 CASE 工具的使用者。成为一个优秀的建模专家需要多年的积累，不会是一周针对某个价值几千美元工具的培训。一个优秀的 CASE 工具是很重要，但你必须学习使用它，并能够使用它设计它支持的模型。

22. 理解完整的过程

好的设计人员应该理解整个软件过程，尽管他们可能不是精通全部实现细节。

软件开发是一个很复杂的过程，还记得《object-oriented software process》第 36 页的内容吗？除了编程、建模、测试等你擅长工作外，还有很多工作要做。

好的设计者需要考虑全局。必须从长远考虑如何使软件满足用户需要，如何提供维护和技术支持等。

23. 常做测试，早做测试

如果测试对你的软件来说是无所谓的，那么你的软件多半也没什么必要被开发出来。

建立一个技术原型供技术评审使用，以检验你的软件模型。

在软件生命周期中，越晚发现的错误越难修改，修改成本越昂贵。尽可能早的做测试是很值得的。

24. 把你的工作归档

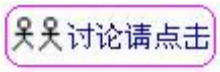
不值得归档的工作往往也不值得做。归档你的设想，以及根据设想做出的决定；归档软件模型中很重要但不很明显的部分。 给每个模型一些概要描述以使别人很快明白模型所表达的内容。

25. 技术会变，基本原理不会

如果有人说“使用某种开发语言、某个工具或某某技术，我们就不需要再做需求分析，建模，编码或测试”。不要相信，这只说明他还缺乏经验。抛开技术和人的因素，实际上软件开发的基本原理自 20 世纪 70 年代以来就没有改变过。你必须还定义需求，建模，编码，测试，配置，面对风险，发布产品，管理工作人员等等。

软件建模技术是需要多年的实际工作才能完全掌握的。好在你可以从我的建议开始，完善你们自己的软件开发经验。

以鸡汤开始，加入自己的蔬菜。然后，开始享受你自己的丰盛晚餐吧。



回顾过去，展望未来

调查研究你的上一个项目可以改进你的下一个项目

Karl Wiegers，Johanna Rothman 著　亚玲 译

Pat，一位刚刚起步的Internet公司的副总裁，对她的项目组先前发布的产品版本感到自豪的同时，对产品发布的时间比预期的时间要晚好几个星期也很关注。在项目回顾会议上，Pat听到好几个组员说：“我本来可以更早完成的，但是我无法准确知道’完成’的含义。如果我知道我的部分什么时候真正完成的话，我们可能会更早地发布产品。”她感到非常惊讶。这是一个有价值的评论，Pat根据她下个项目的信息定义了清晰的发布标准，从而帮助她改善了项目组的时间安排。

回顾提供了一个有组织的机会，来回顾项目或者项目的某个阶段。你可能从中总结可以继续保持的、很有效的实践，或者差的实践并且知道该如何去改善。回顾帮助你从通常很痛苦的经验中得到教训。

一个聪明的项目管理人员在开始一个新项目之前，会温习从先前项目中得到的教训以节约时间和避免不眠之夜。你可能不能对从过去的经验中得到的价值进行定量分析。但是，回顾，这种小小的投资，将几乎肯定地获取比它大的收益。在今天速度驱动和有明显交货底线的开发世界里，你无法承受重犯过去的错误和一个项目接一个项目地遇到相同的意外。

对回顾的定义

回顾是从一个已经完成的项目或者项目阶段中收集知识、洞察力、可能的度量和工具。度量可能包括进度表的真实结果，努力成果，成本和质量，它们可以提高你未来的估算水平。你可以将关键的项目文档，计划和规格说明书存档，以便作为未来项目的范例。

回顾提供一种使参与者远离日复一日的压力，共享他们的观察结果的方式。即使这个项目是一个巨大的失败，你从评价失败中得到的经验教训也会带来积极的影响和有益的改进机会。

谈论什么是回顾可能看起来很傻，但是名字使人产生很多联想。当某个人称回顾某个已死的人，我们就会想知道谁死了；但有时候项目是成功的！如果解释回顾为分娩后的新生命则意味着新“婴儿”产品可

能某天会长大，但是离成熟还很远。回顾和事后项目回顾是中性的术语，意思是通过思考分析先前的项目经验，获取实践真知。

无论何时你需要收集项目信息或者评价工作进行方式，请进行回顾。许许多多的项目重复一系列的开发周期，所以你可以在每个开发周期后积累经验教训以帮助你后续的开发周期。当先前项目中有特别有效的实践或者特别差的实践时，思考反省经验教训特别值得。对于持续几个月的项目的早期所发生的事情往往很容易忘记，所以一个持续时间长的项目在到达每个主要里程碑时进行一个短回顾是很必要的。时间的治疗效果和最近成功的喜悦通常会减轻你在前些时候所遭受的痛苦，但是那些痛苦的记忆往往包含未来改进的种子。

回顾过程

一个有效的回顾过程如图1所示。这个过程的关键人物是发起回顾的项目经理，项目组组员和一个项目协助人员。关键人物是发起回顾的项目经理，项目组组员和一个项目协助人员。

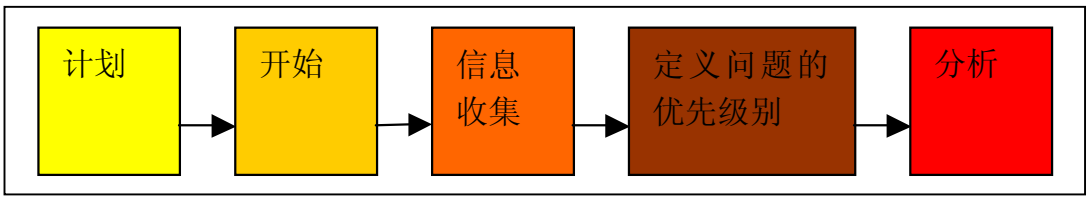


图1. 回顾过程

计划。计划开始于要求回顾的项目经理和项目协助人员共同决定回顾的范围(整个项目或者一部分)，要检查的活动和任何需要仔细研究的具体问题。在计划阶段，还需要你确定需要访问的人，选择合适的设备(最好是不在现场并且避免分心)，选择你将使用的协助技巧，并定义日程表。

将参与者分到不同组中，我们得到了不同的结果。在一次回顾中，Karl和另外一个项目协助人员分开主持了两个讨论。其中一组包括6个管理人员，而另一组由15个软件专业人员组成。项目协助人员和发起回顾的项目经理认为，如果软件专业人员的项目管理人员在场的话他们会不愿意提一些问题。在这种情况下将参与人员分开效果很好，虽然我们必须仔细将两组的问题合并和决定问题的优先级别。但是在另外一种情况下，将参与者分开是不应该的。参与的两组人员包括一个软件开发小组和在一个较大的Web站点开发项目的可视化设计小组。Karl低估了参与者的合作倾向。尽管两组之间对于某些问题看法有摩擦，两个组在一起讨论他们共同的问题将会更好。

Johanna从来没有把参与者分组，因为她认为分组将会降低人们的感知力。在培训管理人员时告诉他

们回顾可能发现让他们感觉不舒服的问题，是非常重要的。正如Weinberg在“*The Secrets of Consulting* (Dorset House, 1985)”一书中所说，“不管它看起来像什么，它始终是人的问题”，有时候人的问题是因为管理产生的。如果管理人员阻碍了讨论的话，可以考虑请求管理人员离开房间。

开始。在有所有参与者在场的短暂的开始期间，发起回顾的项目经理介绍项目协助人员和其他不认识的面孔。项目经理还需要指出回顾的目标，并首先感谢所有参与者的参与和诚实的意见。项目经理应该基于回顾结果清楚地陈诉他将采取具体行动的承诺。然后由项目协助人员列出日程安排。要建立良好的建设性环境，项目协助人员要定义一些基本的规则，包括：

- 允许每个人发言。
- 尊重他人的经验和观点。
- 避免批评其他参与者的意见和思想。
- 避免为已经过去的事件批评他人。
- 找出根本原因，而不仅仅是现象。
- 集中精力于理解、学习和向前看。

数据收集。当某些回顾收集具体数据和项目工件时，核心活动是从参与者那儿收集问题，观察报告和关注的问题。在回顾时我们探索三个基本问题：什么是好的实践(我们想在将来重复),什么本来可以做得更好(我们可能想要改善它),什么是使我们意外的(这些可能是要在以后的项目中继续观察的一些风险)？

一个有经验的项目协助人员会使用很多技巧从不同的参与组里引出信息。传统的方法是让协助人员站在挂在组员之前的活动挂图(一种由很多页组成的顶部夹在一起的图表，翻动时能够连续地展示资料)旁边，并要求参与者提出问题。项目协助人员标记好的经验为正而标记不好的经验为负。还可以采用循环的方法，协助人员要求每个参与者轮流提出一个问题并且循环通过每个听众直到每个人都通过。这种产生问题的途径通常需要60-90分钟。一旦问题被提出之后，项目协助人员(或者记录人员)在分开的索引卡片上记录提出的每个问题。参与者将卡片分成相关组(或者相似组)并且为它们命名。

这种传统的协助途径有下面几个缺点：

- 顺序地产生问题速度很慢
- 很容易让少数直率的参与者主宰讨论

- 很容易陷入讨论某个热门话题，而不是找出所有问题
- 一些人对在公共讨论会提出问题不太自在
- 有影响力或者有强控制欲的参与者可能阻止其他人提出问题

如果你担心上述任何一个因素，可以考虑使用其它的协助途径。让参与者安静地产生问题的技巧可能会比公众的顺序的方法更为有效和全面。在使用这种方法的一次实践中，协助人员和回顾组织人员在回顾会议之前标出了可能会提出的几种问题。软件开发项目的通用种类包括交流、组织、联合作业、管理、需求、设计、建造、测试、子承包工厂、商业问题和过程。可能不是每个回顾都包括所有这些问题。提前定义问题种类可能会冒限制提出问题范围的风险，所以你可能更愿意进行小组讨论，并在问题产生之后为这些问题的种类命名。

在每一页写下每个问题种类的名称，然后将每页分成几个部分：好的实践、可以本来更好的实践, 以及吸取到的经验。在会议期间，让参与者在分开的3个部分(每部分有5个夹在一起的页)写下他们的每个问题，并指出它们相应的问题类别。项目协助人员将这些问题放在活动挂图页的相应部分。花费大约20分钟整理好的实践，然后花另外20或30分钟整理本来可以更好的实践。参与者可以在房间内走动，看其他的人在活动挂图上写的东西以激发他们的思考。

这种方法克服了传统的“协助人员在活动挂图旁边”的大多数缺点。参与者一起工作可以在给定时间内产生更多问题。每个问题被提出时被打搅的可能性较小。而且，那些可能不愿意大声陈述他们观点的人，可以以安静和匿名的方式陈述问题。但是，项目协助人将不得不大声地读出活动挂图上所有棘手的意见，以确保每个问题被清楚地陈述和合适地分类。他还必须集合相关的问题。

要结束回顾的数据收集部分，你可能会对每个项目组组员提出两个问题：这个项目的哪一点你想在将来的项目中保持？这个项目的哪一点你想在将来的项目中有所改变？

定义问题的优先级别。一个成功的回顾将产生远比项目组实际陈述多得多的问题。你必须从这些问题中找出参与者同意的最有价值的东西。一些高优先级的问题可能针对有效的实践，这些有效的实践你可能想在将来的项目中固定下来。而其它反映当前项目实践缺点的方面也需要迅速地陈述出来。

一个典型的划分优先级的技巧是由pareto提出的。 每个参与者投票给他认为优先考虑的问题，通常是所有问题的20%。为了简化一个大组里的数据收集，一些项目协助人员给每个参与者3-5票。投票过程中将优先问题着色是一个好方法。参与者环绕房间走动，检查活动挂图，然后将他们认为最重要的问题做彩色标记。那些获得点数最高的问题就是最成熟的要立刻采取行动的问题。但是，看见意见簿上的点数可能

影响后面投票的人，他们可能不想浪费票数在前面投票人没有投的问题上。要避免这种情况发生，参与者可以将投票点数放在意见簿的后面。

分析。 如果你在回顾期间有时间的话，花费15或20分钟讨论每个有高优先级的条目。或者，在回顾会议之后，聚集一个小组来研究这些问题。对于实践效果很好的条目，要确定它们成功的原因以及它们带来的好处。找出可以确保当前项目中行之有效的实践也可在将来项目中行之有效的方法。对于高优先级别的“本可以做得更好”的条目，找出不如人意的原因，每个条目(问题)不如人意的结果，和下次可以做得更好的建议。

影响回顾成功的因素

只有在无偏见的，没有指责的环境下，回顾才能成功，并且开放式的交流也是不可缺少的。如果一个项目困难重重或者不成功，必须寻找一些出路；但是，项目协助人员必须限制寻找的出路和方法是在建设性的轨道上。确保你的回顾不沦落成迫害。回顾必须强调从共享的项目经验中无指责地进行学习。让我们看一下一些回顾的关键成功因素。

确定你的目标。作为项目管理组织人员，你应该确定你回顾的目标和应该聚焦的项目的某些具体方面，以及某些行动的潜在好处。如果在回顾过程中收集的信息导致某个建设性的过程改变，谁应该首先公开宣布？另外，如果问题的根本原因被揭示出来，谁的脸色将会很难看？ 记住，你不是寻找替罪羊，但是你需要明白什么是确实已经发生的，为什么会发生。

派用一位熟练的无偏见的项目协助员。期望项目管理人员客观协助回顾是不现实的。项目管理人员可能另有苦衷或者想要维护他的声誉。一些项目管理人员可能无意识地阻碍了参与，尽管他们的意图是好的。其他参与者可能被迫对重要问题保持缄默或者管理人员可能对某些问题只看重自己的意见。

要避免这些问题，邀请一个该项目组外、有经验、无偏见的项目协助人来主持回顾。项目协助人的主要目标是在一个建设性和学习性的环境中触及关键性的问题，使回顾成功。考虑让一个在回顾活动中不活跃的人记录产生的问题。

安排合适的参与者。当然，基本的参与人员是项目组组员。管理人员代表只有在他们是项目组的成员时才被邀请入回顾会议。但是，你应该提供经验教训的总结给可以从中获益的公司高级管理人员或者其他管理人员。

有些项目组可能太忙了，太大或者在地理位置上分布太远，以致让所有的项目组组员同时参加回顾会议不太可能。在这种情况下，可以从这个项目包括的多个功能领域内选择代表。如果一个大项目被划分为

多个子项目，每个子项目应该进行自己的回顾会议。然后每个子项目的代表可以在总项目回顾会议上进行回顾。

当某些我们认为对项目有重要看法的人太忙不能参加回顾会议时，我们询问他们是否认为项目的一切都进行顺利。通常，他们对项目有一些重要的看法和建设性的意见。然后我们帮助他们妥善安排当前的时间，以使我们能够听取他们对先前项目的意见。

如果项目涉及多个组，并且这些项目组互相指责项目问题或者拒绝一起坐下来研究共同的问题。我们可能首先讨论这些组之间的矛盾。很多情况下会发现一些重要的项目问题。如果这些组在回顾中合作不愉快，他们极有可能在项目中也存在冲突。回顾可能说出这些组在下次合作中需要改进以更好合作的地方。

培训参与者。如果参与者不习惯于回顾，并且被研究的项目具有严重的问题，邀请参加回顾将会引起混淆或者抵抗。一些参与者可能会非常焦虑，而其他人将会急于摆脱指责。通过邀请材料提供信息和保证给参与者。提前描述回顾过程，通过强调回顾是一个面向未来的，改进项目过程的活动，建立一个合适的建设性的倾向。

聚焦事实。回顾应该陈述项目的过程和结果，而不是对参与者进行人身攻击或者指出他们的错误。项目协助人员应该确保参与者不互相指责或调和矛盾，而是集中讨论已经发生的事情。但是，人们通常以不同的方式体验事情。理解不同的解释可以敞开心扉，增加新的见识。

找出行动计划领导人。找出要写下并负责采取改进行动计划的人，由他观察这些计划是否能导致可见的好处。为计划中的每个行动条目指定一个人，他负责实现并报告进度给计划行动领导人。行动计划领导人必须对这些人完成他们行动项目的能力了如指掌。

回顾行动计划

回顾完以后，不要立即应付所有提出来的问题。从最高优先级的问题清单中选择三个问题，其它的留待以后处理。写下你的行动方案，包括你的改进目标，纠正问题的步骤，说明谁负责什么行动以及行动完毕后可以交付的文挡清单。在下一次回顾中，检查这些行动的结果是否理想。记住，一个不能导致具体行动的行动计划是无用的。Karl曾经协助同一个Internet开发小组两年。一些在后来回顾中的问题曾经在两年前就提过。不能从过去的实践中吸取经验一定会使你重复失败。没有什么能够比不实施建议能更快地阻止项目组织进行回顾了。

吸取的教训

要使项目组织获取最大的利益，请积累从回顾中获取的经验。我们更愿意以一种无偏见的方式记录教

训，所以，我们是从好的实践中还是从坏的实践中获取经验并不重要。在“吸取的经验”数据库中的信息可以包括：

- 教训的陈述
- 吸取的教训种类
- 标明教训入库的时间
- 项目名称
- 忽略教训的代价
- 实施教训的建议
- 涉及的工作产品
- 相关的生命周期

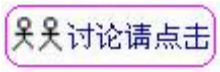
人的因素

因为我们是人类，我们的个性影响我们对各种问题的反映方式。在Johanna协助的一次回顾中，当他描述一个特殊的问题时，某个管理人员参与者流下了眼泪。Johanna在休息时和他交谈，他说他感到管理层不信任他，但是他觉得说这些也并不舒服。他同意让Johanna提出这个敏感的问题。

重新开会后，Johanna说某个参与者感觉到不被管理层信任，理由是她在活动图表上写下的三点。整个房间保持沉默了一分钟。这时候有个人也说：“我也是”。其他几个人也有共鸣，然后他们看着高层管理人员。Johanna提醒项目组：我们不是评价个人，而是提出我们关注的问题并解决它们。高层管理人员然后问道：“这里有人信任我吗？”没有人回应。这是一个奏效的时刻。高层管理人员列出了信任作为一个问题。

项目组是最好的信息来源，他们提供一个刚刚结束的项目或者开发阶段的真实情况。使用回顾帮助项目组组合整个项目的情况，以便项目领导人可以利用这些信息，在下个项目中创造一个更有效的环境。但是记住，所有的项目组改变都需要一定时间、耐心和涉众的许诺。如果人们不想改变，他们就不会改变。

原文链接 http://www.umlchina.com/ProjMan/lookback.htm



功能点过程

Adams Wang

1 目的

本规程的目的是基于软件需求产生软件规模的估计。功能点是基于应用软件的外部、内部特性以及软件性能的，一种间接的软件规模的测量。功能点与软件成本具有重大的成本估计关系（CER：Cost Estimating Relationship）。功能点可以作为经验统计参数化软件成本估计公式和模型的输入，以对软件的成本进行估计。功能点方法被广泛的认可在信息系统、数据库密集型、4GL应用系统开发的规模测量。

2 范围

功能点是对软件功能和规模的间接定量测量，它基于客观的外部应用接口和主观的内部应用复杂度以及总体的性能特征。该规程由三个逻辑部分组成：决定未调整的功能点计数、加权因子和功能点。

决定未调整的功能计数包括对外部输入、外部输出、外部查询、内部逻辑文件和外部接口文件的计数。决定加权因子包括划定系统、输入和输出、应用复杂度的级别。决定功能点包括将未调整的功能点和加权因子整合在一起。

功能点具有两个独立的目标。第一个目标是作为软件测量、对比和分析（如，软件度量方法）的基础。第二个，也是更重要的目标是作为软件成本估计模型（如，公式）和产出工作量（如，工时）工具的输入，软件成本估计模型和工具则基于功能点和工作量之间的经验成本估计关系（CER）。

3 角色和职责

任务经理：任务经理负责为软件成本估计进行对功能点的估计。任务经理必须基于外部应用接口估计未调整功能点，和基于应用程序的复杂度和性能对加权因子进行估计。 任务经理必须辅助决定功能点以及从技术人员处获取输入。

下列任务承担责任：

- 6.1 决定未调整功能点计数
- 6.2 决定加权因子
- 6.3 决定功能点

4 输入

事务和文件清单：应用软件功能和规模的间接的定量测量基于外部应用接口客观的数量，如：外部输入、外部输出、外部查询、内部逻辑文件和外部接口文件。

- 外部输入：数据由外向内跨越边界的基本处理过程。数据可能来自于数据输入屏幕、电子输入或其它应用程序。数据可以是控制信息或业务信息。如果数据是业务信息，它用于维护一个或多个内部逻辑文件。如果数据是控制信息，它不必更新内部逻辑文件。
- 外部输出：导出的数据由内向外跨越边界的基本处理过程。数据创建发送给其它应用的报表或输出文件。这些报表和文件由一个或多个内部逻辑文件和外部接口文件所创建。
- 外部查询：包括输入和输出构件的基本处理过程。输入和输出构件导致一个或多个内部逻辑文件和外部接口文件的数据检索。该信息被发送出应用程序边界。输入过程不会更新任何内部逻辑文件以及输出不包含导出的数据。
- 内部逻辑文件：完全驻留在应用程序内部的逻辑相关数据的用户可识别的组，通过外部输入所维护。
- 外部接口文件：仅用于引用目的的逻辑相关数据的用户可识别的组。数据完全驻留在应用程序外部，由其它应用程序所维护。外部接口文件是其它应用程序的内部逻辑文件。

下列任务需要输入条件：

- 6.1 决定未调整功能点计数

系统的总体特性：间接和主观的应用软件功能和规模的测量基于内部应用复杂度和整体性能特性级别的划分，如系统、输入和输出以及应用复杂度。

以下的任务需要输入条件：

- 6.2 决定加权因子

5 输出

功能点：功能点作为软件成本估计基础，是软件规模、功能和复杂度的测量。

下列任务需要输出条件：

- 6.3 决定功能点

6 步骤

6.1 决定未调整功能点：

决定未调整功能点的数目包括对外部输入、外部输出、外部查询、内部逻辑文件和外部接口文件的计数。

6.1.1 决定外部输入：

外部输入是数据由外向内跨越边界的基本处理过程。数据可能来自于数据输入屏幕、电子输入或其它应用程序。数据可以是控制信息或业务信息。如果数据是业务信息，它用于维护一个或多个内部逻辑文件。如果数据是控制信息，它不必更新内部逻辑文件。

对于低、平均或高，将数目分别乘以3、4或6。

6.1.2 决定外部输出：

外部输出是导出的数据由内向外跨越边界的基本处理过程。数据创建发送给其它应用的报表或输出文件。这些报表和文件由一个或多个内部逻辑文件和外部接口文件所创建。

对于低、平均或高，将数目分别乘以4、5或7。

6.1.3 决定外部查询：

外部查询是包括输入和输出构件的基本处理过程。输入和输出构件导致一个或多个内部逻辑文件和外部接口文件的数据检索。该信息被发送出应用程序边界。输入过程不会更新任何内部逻辑文件以及输出不包含导出的数据。

对于低、平均或高，将数目分别乘以3、4或6。

6.1.4 决定内部逻辑文件：

内部逻辑文件是完全驻留在应用程序内部的逻辑相关数据的用户可识别的组，通过外部输入所维护。

对于低、平均或高，将数目分别乘以7、10或15。

6.1.5 决定外部接口文件：

外部接口文件是仅用于引用目的的逻辑相关数据的用户可识别的组。数据完全驻留在应用程序外部，由其它应用程序所维护。外部接口文件是其它应用程序的内部逻辑文件。

对于低、平均或高，将数目分别乘以5、7或10。

6.1.6 决定未调整功能点总数：

将带有权重的外部输入、外部输出、外部查询、内部逻辑文件和外部接口文件总和在一起。其结果即为未调整功能点。

6.2 决定加权因子:

决定加权因子包括划系统复杂度、输入和输出复杂度和应用复杂度的级别。

6.2.1 划分系统复杂度级别：

采用0~5的分值划分每个系统复杂度，分别代表无影响（no influence）、偶尔（incidental）、适度（moderate）、平均（average）、重大（significant）和根本（essential）。

6.2.1.1 划分数据通讯复杂度的级别：

具有多少数据通讯设备？

数据通讯描述了应用软件与处理器直接通讯的程度。应用软件使用的数据和控制信息在数据设备上发送和接收。局部直接与控制单元连接的终端被认为会使用通讯设备。协议是一系列规约，它允许在两个系统和设备之间传输或交换信息。所有的数据通讯链接需要某种协议。

以下是记分的指南：

0 应用软件是单纯的批处理或独立的PC。

1 应用软件是批处理，但具有远程的数据入口或者远程打印。

2 应用软件是批处理，但具有远程的数据入口和远程打印。

3 应用软件包括在线连接至批处理或查询系统的数据搜集或TP（远程处理）终端。

4 应用软件不仅仅是终端，并且支持一种通讯协议。

5　应用软件不仅仅是终端，并且支持多种通讯协议。

远程处理现在非常普遍。仅仅10%的项目是“低于平均”的分值2或以下；56%则具有“高于平均”的分值4或5。

对银行项目和个人PC开发的项目，该分值较低。从1991至1996，它具有持续降低的趋势，从高于平均水平降至平均水平。

6.2.1.2　划分分布式处理复杂度的级别：

分布式数据和功能如何被处理？

分布式数据处理描述了应用软件在各个组成部分之间数据传送的程度。分布式数据或功能处理是应用软件边界内部的一种特性。

以下是记分的指南：

0　应用软件无系统组件之间的数据传输或功能处理。

1　应用软件为系统其它组件上的最终用户处理，如PC电子表格或PC DBMS准备数据。

2　数据为传输做出准备，接着被传输以及在其它系统组件上被处理（并非最终用户处理）。

3　单方向的在线的分布式处理和数据传输。

4　双向的在线的分布式处理和数据传输。

5　功能处理动态的在相应的系统组件上执行。

在所有的常见系统特征中，该值取“低于平均值”具有非常大的比例。其统计分布是双峰值的：系统要么是单机，或者分布式处理是作为系统一种比较重要的特性。

在工程系统中往往具有更多的分布式处理。分布式处理在中范围的平台上较其它平台上更为普遍。分布式处理在交易/生产系统和办公信息系统中较管理信息系统和决策支持系统更为普遍。新的开发项目较改进项目中更重要一些。

6.2.1.3　划分性能复杂度的级别：

用户对响应时间或吞吐量是否有所要求？

性能描述了对响应时间和吞吐量性能方面考虑对应用软件开发的影响程度。用户以响应或吞吐量所陈述或认可的性能目标，影响着（或将影响）设计、开发、安装以及支持。

以下是记分的指南：

0 用户没有特殊的性能需求。

1 性能和设计需求被陈述和评审，但不需要特殊的活动。

2 响应时间或吞吐量在峰值时间是关键的。对CPU利用没有特殊的设计。处理的极限在日后考虑。

3 响应时间或吞吐量在所有的工作时间是关键的。对CPU利用没有特殊的设计。与其它交互系统处理极限方面的需求是强制的。

4 迫切的用户性能需求，要求在设计阶段进行性能分析方面的工作。

该特征具有较分散的分布：32%的项目低于平均值，30%的项目处于平均水平，38%的项目高于均值。

性能对于交易/生产系统较管理系统更为重要。新的开发项目较改进项目中更重要一些。

6.2.1.4 划分配置项负载复杂度的级别：

对当前的硬件平台的使用程度？

配置项的使用程度描述了计算机资源对应用软件开发的影响程度。需要特殊设计考虑的满负荷运行的操作配置，是应用软件的一个特征。例如，用户想在现有的或指定的满载设备上使用应用软件。

以下是记分的指南：

0 没有明显的或隐式的操作限制。

1 存在操作约束，但限制较典型的应用较小。限制不需要特殊的工作。

2 存在某些安全性或时序的考虑。

3 特殊应用软件部分存在特殊的对处理器的需求。

4 所要求的操作限制对中心处理器或主要处理器上的应用软件需要特殊的限制。

5 另外，在系统的分布式组件上存在特殊的限制。

本特性的分值普遍较低：66%低于均值，20%处于平均水平，14%高于均值。

交易/生产系统和办公信息系统的分值较管理信息系统和决策支持系统低。新的开发项目较改进项目中低；中等项目较其它平台高；工程系统较高。从3GL项目至4GL项目，分值会增高。

6.2.1.5 决定系统复杂度的级别：

4个带权重的分值相加即为系统复杂度。

6.2.2 划分输入和输出复杂度的级别：

采用0~5的分值划分每个输入和输出复杂度，分别代表无影响（no influence）、偶尔（incidental）、适度（moderate）、平均（average）、重大（significant）和根本（essential）。

6.2.2.1 划分事务率复杂度：

事务执行的频繁程度？

事务率描述了业务交易（事务）影响应用软件开发的程度。如果事物率高，它会影响设计、开发、安装和支持。

以下是记分的指南：

0 预计没有峰值的事务处理周期。

1 预计存在峰值的事务处理周期（如：月、季、年）。

2 预计每周存在峰值的事务处理。

3 预计每日存在峰值的事务处理。

4 用户需求中要求高的事务率或者服务级别的约定足够的高，要求在设计阶段进行性能分析。

5 用户需求中要求高的事务率或者服务级别的约定足够的高，要求在设计阶段进行性能分析。另外，需要在设计、开发和/或安装阶段使用性能分析工具。

事务率的分值在分布在0~4的范围内；5分情况较少。

事务率在银行系统中较一般情况重要性高，在工程系统中则较低。在大型机其它平台重要性高。尽管可能期望对于事务/生产系统而言，重要程度高一些，但在应用类型之间没有重大的差别。从1991年至1996年，该分值有着稳定的提高。

6.2.2.2 划分在线数据项复杂度：

百分之多少的信息是在线输入的？

在线数据项描述了数据通过交互式事务输入的程度。应用软件提供在线数据项和控制功能。

以下是记分的指南：

0 所有的事务以批处理的形式处理。

1 1%至7%的事务是交互式数据项。

2 8%至15%的事务是交互式数据项。

3 16%至23%的事务是交互式数据项。

4 24%至30%的事务是交互式数据项。

5 超过30%的事务是交互式数据项。

直到现在，该特性在所有的调整因子中是最高的，并且变化是最少的。60%的项目对该特性的取值为5分，最大的可能值。

根据IFPUG指南，5分意味着超过30%的事务包括交互式数据项。对于现在而言，作为阀值可能30%过低；较高的取值可能能够提供更有用的区别。

对于单个机构COBOL!主机/银行项目，该分值较低（通常3分）。而5分的取值近乎适用于其它一切情况。

6.2.2.3 划分用户使用效率复杂度：

应用软件是否就最终用户使用效率上有所设计？

最终用户使用效率描述了对人为因素和应用软件用户的易用性的考虑程度。在先功能强调了最终用户使用效率的设计（如，漫游帮助、菜单、在线帮助和文档、自动游标移动、滚动条、在线事务的远程打印以及预定义功能键）。

以下是记分的指南：

0 无

1 上文中的1或3项。

2　上文中的4或5项。

3　上文中的6项以上，但无特定相关于使用效率的用户需求。

4　上文中的6项以上，用户使用效率的需求要求就人的因素安排设计任务（例如，最少击键次数、最大化默认值、模板的使用）。

5　上文中的6项以上，用户使用效率的需求要求使用特殊的工具以展示达到即定的目标。

该特性具有广泛的分布，有些高分值的趋势：34%低于均值， 43%高于平均值。

用户使用效率对于信息管理系统较事务/生产系统重要。对于新开发项目，该分值较增强项目低，并具有较扁平的分布。同样的，从3GL项目至4GL项目，分值会增高。

6.2.2.4　划分在线更新复杂度：

多少内部逻辑文件会被在线的事务更新？

在线更新描述了内部逻辑文件在线更新的程度。应用软件为内部逻辑文件提供在线更新。

以下是记分的指南：

0　无

1　在线更新1至3个控制文件。更新量较少，恢复容易。

2　在线更新4个或4个以上的控制文件。更新量较少，恢复容易。

3　在线更新大量的控制文件。

4　另外，遗失数据的保护是关键的，并在系统中进行特定的设计和编码实现。

5　另外，大数据量带来了恢复过程中的成本考虑。需要最少人为干涉的高度自动化的恢复步骤。

在线更新的分值倾向高（半数高于均值），但大多数在3~4，5分较少。

事务/生产系统的分值较高。个人PC平台比其它平台低。同样的，从3GL项目至4GL项目，分值会增高。

6.2.2.5　决定输入和输出复杂度：

4个带权重的分值相加即为输入和输出复杂度。

6.2.3 划分应用软件复杂度的级别：

采用0~5的分值划分每个应用软件复杂度，分别代表无影响（no influence）、偶尔（incidental）、适度（moderate）、平均（average）、重大（significant）和根本（essential）。

6.2.3.1 划分复杂处理复杂度：

应用软件是否具有大量的逻辑或数学处理？

复杂度处理描述了处理逻辑对应用软件开发的影响程度。以下是一些处理情况：灵敏度控制、特殊的监控处理、安全性处理、逻辑处理、数学运算、异常处理、复杂度处理以及设备无关性。

以下是记分的指南：

0 无灵敏度控制、逻辑处理、数学运算、异常处理或复杂度处理。

1 包括灵敏度控制、逻辑处理、数学运算、异常处理或复杂度处理中的任何一种。

2 包括灵敏度控制、逻辑处理、数学运算、异常处理或复杂度处理中的任何两种

3 包括灵敏度控制、逻辑处理、数学运算、异常处理或复杂度处理中的任何三种

4 包括灵敏度控制、逻辑处理、数学运算、异常处理或复杂度处理中的任何四种

5 包括所有的灵敏度控制、逻辑处理、数学运算、异常处理或复杂度处理。

该特性具有正态分布，主要分布在均值3，0和5的分值较少。

复杂处理的分值在大型机上是最高的，而微机上是最低的；在3GL项目中最高，4GL项目中最低。该分值在新的项目中较增强型的项目高，并具有较扁平的分布。处理复杂度从1991年~1996年稳定的增高。

6.2.3.2 划分重用性复杂度：

应用软件开发以满足一个或是多个用户的需要？

重用性描述了应用软件和软件中的代码特定的被设计、开发和支持，以在其它软件中重用的程度。

以下是记分的指南：

0 无重用代码。

1 可重用代码在应用软件中重用。

2　10%以下的应用软件考虑多个用户的需要。

3　10%以上的应用软件考虑多个用户的需要。

4　应用软件特定的被打包和/或文档化以易于重用，应用软件被用户在源代码级别客户化。

5　应用软件特定的被打包和/或文档化以易于重用，应用软件通过用户参数维护的方式被客户化使用。

重用性的重要性通常较低，59%项目低于均值，仅有14%高于平均值，但它处于非常混合的状态。

决策支持系统中重用性考虑比其它类型多一些，而个人PC上的重用性的考虑较大型机少。

6.2.3.3　划分安装容易程度复杂度：

转换和安装的困难程度多大？

安装的容易程度描述了环境的变化对应用软件开发的影响程度。转换和安装的容易程度是应用软件的特性之一。转换和安装计划和/或转换工具在系统测试阶段被提供和测试。

以下是记分的指南：

0　用户未提出特殊的要求，安装不需要特殊的调整。

1　用户未提出特殊的要求，但安装不需要特殊的调整。

2　用户提出转换和安装的要求，转换和安装指南被提供和测试。项目中转换的因素不被认为是重要的因素。

3　用户提出转换和安装的要求，转换和安装指南被提供和测试。项目中转换的因素被认为是重要的因素。

4　在2的基础上，自动转换和安装工具被提供和测试。

5　在3的基础上，自动转换和安装工具被提供和测试。

该特性具有最广泛的分布性，总的来说分值较低（54%低于均值，22%高于均值），但是两种极端的情况均有体现。安装的容易程度在20%的项目中不被考虑，而在15%的项目中非常重要。

增强型项目的分值比新开发的项目高；大型机比其它平台高；工程系统比其它业务领域高。

6.2.3.4　划分操作容易程度复杂度：

应用软件在启动、备份和恢复的有效性/自动化程度？

操作的容易特征描述了应用软件在操作方面（如，启动、备份和恢复过程）的考虑程度。操作的容易程度是应用软件的特性之一。应用软件最小化人工活动，如磁盘的mount、文件处理和直接的现场人工干涉。

以下是记分的指南：

0 除了平常的备份过程，用户没有要求特殊操作上的考虑。

1 任意一种人工启动、备份和恢复；自动启动、备份和恢复；磁盘mount的最小化；或文档（paper）处理最小化。

2 任意两种人工启动、备份和恢复；自动启动、备份和恢复；磁盘mount的最小化；或文档（paper）处理最小化。

3 任意三种人工启动、备份和恢复；自动启动、备份和恢复；磁盘mount的最小化；或文档（paper）处理最小化。

4 任意四种人工启动、备份和恢复；自动启动、备份和恢复；磁盘mount的最小化；或文档（paper）处理最小化。

5 应用程序设计为无人操作。无人操作意味除了启动和关闭系统，系统不需要操作员的干涉。自动错误恢复是应用软件的特色。

操作容易程度是不怎么考虑的问题。该特性的分值近乎最低：62%低于均值，仅有14%高于平均值。分值分布在0~3，2是最普遍的情况。

当项目根据应用类型分组时，出现的唯一重大的差异：信息管理系统和决策支持系统的分值较交易/生产系统和办公信息系统高。

6.2.3.5 划分多个地点复杂度：

应用软件是否设计支持多个地点场所/机构？

多个场所描述了应用软件被开发以适于多个地点和用户机构的程度。应用软件特定的被设计、开发、支持，以工不同的组织机构在不同地点安装。

以下是记分的指南：

0 用户需求不需要考虑多个用户/安装地点的需要。

1　设计中考虑了多个场所的需要，应用软件设计在相同的软硬件环境中操作。

2　设计中考虑了多个场所的需要，应用软件设计在相似的软硬件环境中操作。

3　设计中考虑了多个场所的需要，应用软件设计在不同的软硬件环境中操作。

4　文档和支持计划被提供和测试，以支持应用软件在不同地点的使用；应用软件如1或2所述。

5　文档和支持计划被提供和测试，以支持应用软件在不同地点的使用；应用软件如3所述。

该特性在所有的因素中具有最低的取值：68%低于均值，33%具有最小的可能值0。

分值对于法律系统非常低，而对于工程系统较高。新开发的系统比增强或重新开发的系统高；3GL项目比其它的高；中型机比大型机高。同样，管理系统和决策系统的分值较交易/生产系统和办公系统高。

6.2.3.6　划分修改容易程度复杂度：

应用软件是否被设计以方便于修改？

修改方便描述了应用软件被开发以利于处理逻辑或数据结构修改的程度。下列特性适用于应用软件：处理请求的灵活的查询和报表（如，简单、平均和复杂）和使用每日或隔日更新的表保存业务控制数据。

以下是记分的指南：

0　无

1　任何一种简单、平均或复杂的查询和报表，或者即时的或隔日的业务控制数据维护。

2　任何两种简单、平均或复杂的查询和报表，或者即时的或隔日的业务控制数据维护。

3　任何三种简单、平均或复杂的查询和报表，或者即时的或隔日的业务控制数据维护。

4　任何四种简单、平均或复杂的查询和报表，或者即时的或隔日的业务控制数据维护。

5　所有五种简单、平均或复杂的查询和报表，或者即时的或隔日的业务控制数据维护。

该特性的每个分值均有较好的体现，但普遍较低：53%低于均值，20%高于平均值。分布是双峰值的，两个通常的取值是0和3。

对于3GL项目取值较低，4GL项目较高。新开发的项目低；大型机低；工程项目高。并不令人奇怪，该特性对信息管理系统和决策支持系统较重要，而交易/生产系统的重要性较低。

6.2.3.7 决定应用复杂度

6个带权重的分值相加即为应用软件复杂度。

6.2.4 决定加权因子：

系统复杂度、输入和输出复杂度和应用软件复杂度相加即为加权因子的值。

6.3 决定功能点：

决定包括未调整功能点和加权因子的功能点。

6.3.1 决定复杂度因子：

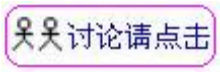
将加权因子乘以0.01，加上0.65，作为复杂度因子。

6.3.2 决定功能点：

将未调整功能点和复杂度因子相加得到功能点。

参考资料

- Fischman, Lee, Evolving Function Points, Crosstalk, February 2001.
- Garmus, David, & Herron, David, Function Point Analysis: Measurement Practices for Successful Software Projects, Addison Wesley, 2001.
- International Function Point User's Group (IFPUG), Function Point Counting Practices Manual (Release 4.1), May 1999.
- Longstreet, D., Function Points Step by Step, Longstreet Consulting, Inc., January 1999.
- Lokan, C. J., An Empirical Analysis of Function Point Adjustment Factors, University of South Wales, December 1998.
- Garmus, David, & Herron, David, Measuring the Software Process: A Practical Guide to Functional Measurements, Prentice Hall, 1996.
- Albrecht, Allan J., Measuring Application Development Productivity, Proceedings SHARE/GUIDE IBM Applications Development Symposium, October 1979.



CASE 工具赛马

Gary K. Evans 著，张启鹏 译

高级软件技术公司（Advanced Software Technologies，AST）、冠群公司（Computer Associates，CA）和 Aonix 公司提供有着不同个性的，健壮的建模产品。

正在搜索完美的面向对象建模工具？谁不是呢？那些由开发者创建并且也是为开发者创建的工具，总是承诺即将是成熟的产品，却鲜有实现。随之，我们已经学会了讨厌它们，但又不能离开它们。在过去的五年里，OO 世界已经从工具能力令人绝望的缺乏，变成令人困惑的选择过剩 -- 至少 16 种基于统一建模语言（UML）的工具。因为在第二个千年末之际，OO 软件用纸张和白板设计太过复杂，现在，建模能力和 Java 知识一样非常珍贵。今日，在任意 OO 商店里，问题不是是否使用建模工具，而是采用哪个工具。

为了评估当代 OO 建模工具，我最近考察了三个有名的冠有“完美的 OO 建模工具”称号的竞争者：高级软件技术公司（Advanced Software Technologies，AST）的 GDPro，冠群公司（Computer Associates，CA）的 Paradigm Plus，Aonix 公司的 Software through Pictures（StP）。尽管瑞理软件公司（Rational Software）声称其议论多多的 Rose 产品要“占据面向对象分析、建模、设计和构造工具的市场，”但是，很清楚地，其它面向对象工具提供了有竞争的特征，值得研究。

所有这些工具都具有重要的相似性：广泛覆盖 UML 表示法、支持多用户开发、共享的信息仓储、众多外部工具和应用的集成性、多语言代码生成和逆向工程，以及基于规则的通过类似 Visual Basic 的脚本语言的可配置能力。事实上，要成为一个受重视的竞争产品，一个 OO 工具必须有这些能力。

但是，和所有这些一样重要的是，这些工具如何良好地支持它们的主要目的--创建 OO 模型？该工具是否可用？是否直观？你能一天 10 小时地用它来获取、沟通你的思想吗？是否它交付了 UML 表示法的工具，抑或开发商只创建了其中一部分？

我决定集中于每个产品如何处理核心的 UML 图，即用例图、类图、顺序图和状态图。如果一个 OO

CASE 工具不能处理好这些图，那么甚至它如何处理其它的五个图根本不值得考虑。

GDPro版本3.2

GDPro 是这次评论中最新的一个产品。第一个版本发布于 1997 年九月，现在是版本 3.2，在本文付印时，版本 4.0 beta 即将面市。我无法评论 4.0 beta，但它所承诺的特征相当诱人。

GDPro 在一个“system”内管理兼容 UML 1.0 的模型。创建一个系统是一个单步的操作，并且 GDPro 自动提示你初始图和将创建的名字。与这次评论的其它产品相似，GDPro 的图伴随着与之相适应的工具调色板。这些工具调色板设计得很华美，也集中得很完整。

稍一使用，在颜色的帮助下，我发现自己很容易地就瞄准在正确的工具栏图标上。在图标上的简单点击就选择了该建模元素，然后点击一下制图窗口就创建了该元素的一个实例。

GDPro 3.2工作区

用例图总是相当的直观，GDPro 在管理用例椭圆形和角色上做得相当不错。在这点上只缺少一个主要的特征，就是把一个用例椭圆形连接到一个外部用例描述文件（例如，Word 文件）的能力。我发现 GDPro 的工作于用例图的设计思路的一个要点是：我可以在图上画出通信关系线，不必隶属到任何角色或用例椭圆形！这点使我晕倒。在其它图上的研究证明了相同的不寻常的“不合语法”的表现。

我发现 GDPro 跟随这样的建模思路：支持一个相当松散的信息元素的关系，而不是在 Paradigm Plus 和 Software through Pictures 里发现的严格的 UML 语法。在这个方面，GDPro 更趋向于以用户为中心而不是以 UML 为中心。而这种随意的方法在该产品的其它领域里有例子说明。Primitives 工具栏让你在任意的图上放置通用的图形画和文字。这是 GDPro 是发展自 AST 的 Graphical Designer（图形设计器）产品的自然结果。这也是 GDPro 在 UML 建模是灵活的方法的合乎逻辑的结果：保持建模人员在 UML 上正确，但不仅仅把他或她限制在 UML 上。你也能够用 Insert Object…（插入对象……）功能来把你能想到的任何对象逐个地直接放置到一幅 GDPro 图上。对于 UML 纯粹主义者这是异端，但对于我们中正在试图用我们能用的任何方式交流复杂想法的那些人而已，这是我们的工具箱里的另一个有用的工具。

设立类图很简单的。点击该图窗口下拉出一个空的类箱。Class Association 工具栏图标让你画出你所要的关联关系。要改变关联关系为聚集关系（aggregation）或组合关系（composition），你可打开 Association Properties 对话框。这也是设置大量的其它关联关系信息的地方，而我不得不得出结论，GDPro 里的数据输入机制是它的阿基里斯脚踵（译者按：阿基里斯，Achilles，是希腊里的神，传说他全身刀枪不入，除了脚踵。Achilles’ heel 意寓唯一致命的弱点。）。大多数数据不是通过图输入的，而是通过数据输入对话框。甚

至只要你打开这些对话框一次，你就必须点击“Modify”按钮来实际录入或修改一个数据字段。当要画一个有 30 个类的图时，这显得很枯燥乏味。（我很高兴地告诉大家，其中一部分已经在 4.0 Beta 版本里得到修正。）例如，要在一个关联关系、聚集关系或组合关系上设置多重性，我必须通过两个对话框。第二个对话框要求我手工地录入多重性的 LowerBound 和 UpperBound。这是完全没有必要的。单个下拉列表框就可以容易得多地完成所有这些工作，而且更少导致错误。

我发现，在这里评论的工具里，只有 GDPro 允许我把一个类框的多个复制放在一个图上。这对于很巨大的图非常方便，那些图原先我不得不画多条很长的、曲曲折折的线以连接其它类到一个关联的类上。

GDPro 的顺序图使我迷惑。我不能从浏览器窗口拖一个类到顺序图窗口里，但我可以指示 GDPro 自动放置所有的类到该图上。当我创建对象和对象之间的消息后，我发现我可以“rubber band”（译者按：橡皮圈，在这里是圈住的意思）一组消息线，然后作为一组在图上移上移下，但我不能移动单条消息线！GDPro 不支持消息线编号方式，但我很高兴地发现它能在接受的类里作为操作自动地加入消息。

在 GDPro 里设立一个状态图相当直观。GDPro 只直接地支持事件和动作作为语义元素，但不从动作（actions）中分辨状态活动（activities）。这不是个问题，除了对 UML 纯粹主义者，但我很失望，我被迫手工地录入作为事件的预先确定的 UML 关键字“entry”、“exit”和“do”。除了这个麻烦以外，监护条件（guard condition）在状态里受支持，创建组合（也就是，嵌套的）状态也非常地容易。子状态连接到组合状态，我能够把它们整个作为一组移动。

GDPro 4.0，发布于 1999 年十二月，有几个改进将给建模人员带来很大的方便：

- 改进的文本格式化。在版本 3.2 里当放大或缩小时字体经常不一致。4.0 看起来挺好。
- 图上的数据录入。我测试的所有图都允许我单击、图上的可视化数据录入。在用户界面上的主要改进措施。
- 回退/重复（Undo/redo）：这些动作现在实际上没有限制了。
- 顺序图的巨大改进。支持对象名称在滚动时显示在图的顶部，“message-spreader”（消息延辗机）在消息线之间挤出空间使得可以插入新的消息。
- 直角的（方形的）连接。现在这些在所有图上都支持。

年轻的小马

GDPro 已经自觉地发展为支持 Windows NT 环境。结果，它在 Windows 95 下相当不稳定。但不管它

年轻的缺点，GDPro 在这次工具的“赛马”中还是算好斗的年轻小马。它在图形表达上的适应性，使它在这次评论里从其它更成熟的工具中脱颖而出。它让建模人员录入数据到系统里的用户界面需要好好改写，但 GDPro 的图上编辑功能是个例外，不包括在需要重做的部分里面。不过它的灵活性是一个巨大的特征，它允许我用放置另外的消息或图形到图上的方式补充 UML 表示法。

Paradigm Plus 版本3.6.2

Paradigm Plus 是领袖技术员的运动场。贯穿帮助文件的术语“元模型”（meta-model）的再三出现迫使人心情紧张 — 甚至是核心建模人员。（如果你需要一本计算机字典来理解这解释，那你真的得到帮助了吗？）尽管这是个有修养的滑稽表现，但 Paradigm Plus 有一套一致的设计思想和可用性轮廓。

为了使用 Paradigm Plus，要创建一个项目，指明你使用的建模表示法。Paradigm Plus 支持八种表示法或范例：UML、Booch、Rumbaugh（OMT）、Fusion、Coad/Yourdon、OOCL、OOIE 和 Shlaer-Mellor。在一个项目内，你要创建适应其范例的图。Paradigm Plus 支持所有九种 UML 图，并坚持 UML 1.1 规范。

在一个项目内创建图，和在菜单栏上选择“Diagram | New”一样简单。新图出现在浏览器窗口的右边，你随时可以滚动它。每个图都有一个集中了与该图相关的图标的工具调色板。先点击图标，然后点击图窗口，该模型元素就出现了。

Paradigm Plus工作区

浏览器窗口是一个由 Paradigm Plus 仓储维护的大量信息的完整视图。许多操作可以在一个图窗口里执行，也可以直接在浏览器窗口直接执行。Paradigm Plus 擅长于让你在一个图里或者甚至在一个单模型元素里，指定并过滤你要看或不要看的元素， 但浏览器总是让你看到所有东西。

用例图没有什么令人惊奇，不过那些 Human、Software 和 System 的特别的角色图标显得不必要。用例名能够在行内编辑，UML 1.1 的<<extends>>和<<uses>>关系都受支持。外部用例的描述文档可以用用例符号连接，但，因为一个令人讨厌的疏漏，Paradigm Plus 没有提供一个“Browse…”（浏览……）按钮来搜索该文档；你必须手工地输入完整的路径名称。

类图是直观的。类能够从浏览器窗口拖放到图上。当两个类用一条关系线连接起来后，一个你可以从中选择关联关系、聚集关系等等关系的选择列表立即出现了。继承线是自动变直的，这点不错，但这可以轻易地撤销，如果你情愿要倾斜的线。

在顺序图里有几个奇怪的置疑。当创建一个顺序图时，你不能简单地从浏览器窗口拖放一个类名字。这是因为 Paradigm Plus 残酷地坚持 UML 语义限制，即顺序图只能包括对象，而类不是对象。要加对象到

顺序图里，你必须打开 Object 对话框，从列表框里选择一个类名字，然后给该对象一个名字。同样，Paradigm Plus 将不允许你创建一个匿名的（没有名称的）对象。UML 语法没有这样规定，但 Paradigm Plus 不允许这样，因为它的仓储独立地用它们的名称参考对象。

如果你要插条几个消息到一个图的中间，你将绊倒在 Paradigm Plus 的顺序图的最大的漏洞里。如果你需要把多条线在图上往下移动以腾出些地方，你必须一条一条地移动。理论上可能把多条线作为一组一起移动，但只有所选小组里的每条线都正确地与一个对象的生命线接触 — 很精确很困难的排列。为了实际应用的目的，你不可能这样做。例如，如果你有 100 个紧紧捆绑着的消息，并需要在第五个消息后插入一个新消息，你就得逐条逐条地移动 95 条线。

状态图同时支持简单的和嵌套的状态。从浏览器拖放到一个状态图表上的能力是一个很不错的特征。因为类框很大，请调整它的大小，然后把状态和转移加入该类框里。这将导致类和它的状态之间的语义联结。输入和退出动作可直接支持，但不支持活动（表示为“do/…”）。再次地，Paradigm Plus 追随令人疑惑的 UML 1.1 语义规定，即活动应该在活动图上，因为“活动图是一个特别类型的状态图。”我不同意这点 UML 的强调；活动是类执行的，但活动图没有连接到任何指定的类上。我为 Paradigm Plus 在这点上已跟随了 UML 感到遗憾。

所有 Paradigm Plus 模型都支持强硬的语义；换句话说，Paradigm Plus 将不让你建模某些东西，除非它符合 UML 元模型定义。

具有讽刺意味的是，因为 Paradigm Plus 在语义正确方面的精确性，我可以创建几乎没有业务值的建模关系。在用例图上，Paradigm Plus 允许你从角色到角色画通信关联关系 — 一个可疑的做法，因为角色-角色的关系是在用例的范围之外的。Paradigm Plus 也允许你从角色画通信关联关系到它自己。

当加入预先定义的 UML 多重性（*，1..*，0..1）到一个关联关系或聚集关系上时，你可直接在图上或者在 Properties 对话框里的 Multiplicity 字段里编辑这个多重性，但指定了数字的多重性（1，2-4，8）则不行。这些跟对待约束一样，必须被输入到 Properties 对话框里的 Constraint 字段。更甚的是，约束覆盖了任何存在的多重性。首先这点非常令人迷惑，我至今仍不喜欢它运作的方式。虽然它符合了元模型，但没有帮助我无缝地完成我的工作 — 实际上它导致我的模型不正确，直到我发现 Paradigm Plus 所做的方式。

良种马

如果你使用 Paradigm Plus，你将不得不学习元模型。这是没有必要的坏事情。但如果你的目的只是简单地在你的业务域里完成工作，一开始显得好像 Paradigm Plus 在你的路上设了路障。幸运的是，你不用通

过大学课程就可以把 Paradigm Plus 使用得相当有效率。Paradigm Plus 是良种马，不是拉车的驮马。它是第一流的、精确的 — 有时候精确过度以致碍事 — 但具有你需要完成工作的所有马力。

Software through Pictures版本7.2

Aonix Software through Pictures（StP）是为大项目设计的，甚至于它不能运行在 Windows 95 或 98 下。它有着悠久的大型政府和军事项目的历史 — 说明它的跨越众多 Unix 平台和 Windows NT 4.0 的广阔支持 — 版本 7.2 标榜比旧版本更新、更快、更轻松和更友好，而且有使用的乐趣。它也支持 UML 1.1 规范。

StP Desktop（桌面）和Class Editor（类编辑器）

不像 GDPro 和 Paradigm Plus 作为一个单独的应用出现。StP 是一套协作的应用：StP Desktop、Diagram Editor（图编辑器）和 Table Editor（表编辑器），以及其它。因为这个架构，它感觉起来明显不同。因为 StP 要载入该应用，启动一个 Editor 引起了一阵暂停。许多图的细节分离到 Table Editor 里去了，这让人觉得图里的信息杂乱无章。例如，要加入可视性说明符到类属性和操作里，你必须调用 Class Table Editor 并把这些信息输入到里面。不能直接输入到类图里。上述中，Aonix 的按需载入信息到表里的选择，简化了 StP 仓储的完整性管理。当我载入一个 Table Editor，我可以确信我取得了仓储里的最新信息。

当你启动 StP，你看到简明的桌面窗口。这是报表生成、代码生成和逆向工程的大本营。你从桌面创建的 OO 图存在一个“system”里。从桌面窗口创建一个系统是一个简单的、单步的过程。一旦系统创建后，你可以用 Diagram Editor 创建新的图。

画用例图是很简单的，跟其它图编辑器相似，一条工具栏提供了对应于编辑器类型的图标。StP 运行按你所设立名称和图标的尺寸。StP 也让你把用户特定的构造型应用到角色上，我发现这点对于明显不同的角色职责非常重要。和 GDPro 一样，StP 在这方面的主要问题是，你不能把用例描述文档连接到图上的用例符号上。这很糟糕，因为单独的 UML 用例图几乎没有业务价值；所有重要的信息都存在于用例描述文档里。

Class Diagram（类图）编辑器很直观。我可以创建类并通过点击相应工具栏上的图标来加入属性和操作。StP 支持我曾见过的所有产品所具有的最广泛的 UML 类图饰品，包括多元关系、参数化类和绑定参量以及“或”关系。虽然说 StP 有符号化的马力，但我很奇怪地发现，我不能改变用于图里的字体族，在图的图标里设颜色或改变关联线的标题的位置。

类之间的关系的设置，展现了 StP 的构架特性。在所有的 StP 图上，你都要通过画一个弧线 — 在每个 Diagram Editor 里的工具栏图标 — 在模型元素之间建立一个关系。在用例图制图当中，一个角色和用

例之间的弧线是作为通信关联关系的补偿的。在一个用例和另一个用例之间画弧线，是显示为<<uses>>或是<<extends>>，取决于默认的弧线（Default Arc）定义，这是用户指定的。

在一个类图里，我设默认弧线为“关联关系”。后来我在类之间画的每个关系都呈现为一个关联关系。要指定这些关系中的某些为聚集关系、组合关系或泛化，我只得选择每个关联关系然后打开 Association Properties（关联关系属性）对话框。这个对话框 — 不是实际的图 — 是你要指定详细信息的地方：构造型、多重性、限定词、聚集关系/组合关系、导航性等等。StP 不支持图上编辑或多重性、角色等的规格说明，。你必须通过 Properties 对话框完成这些工作，这显得相当乏味。例如总是要指定属性的多重性，就必须手工地输入。我对 Aonix 公司没有在这个字段上放置一个下拉列表框感到很失望，因为除了“numerically specified”（编号指定的），所有的多重性在 UML 里都预先定义了。

顺序图支持匿名对象，并且把对象加入到图里是非常简单的，只要选择由 StP 提供的列表就行了。可以通过选择 UML | Shift Messages 来移动一组消息线。你也可以指定顺序号码，按那些号码排序或改变顺序排法。另外，你能够指挥 StP 自动把信息传输到接受的类的操作里。尽管这是些可靠的功能，但还是没有提供简单地在顺序图上放置场景文本的功能。

在 Statechart Editor（状态图编辑器）里，创建状态图标以及在状态之间的转移是很简单的，除非你试图加入动作、活动或监护条件到状态里。这些都必须通过 State Table Editor 输入，甚至动作也不实际出现在状态图上！只有一个表显示符号显示在状态图标上，还有提示查找别的某地方的状态细节。

驮马

尽管它有一些怪癖，但这个版本重新确定了 StP 作为驮马的名声。为了从 StP 取得最大的利益，你将不得不定制化它，而它也已经设计为高度的可定制化了。尽管它的用户界面需要某些调整，也有一条学习曲线来指出是运行一个 Diagram Editor 还是 Table Editor，但是 StP 总体而言还算是和它所说的特性相一致的。而因为 StP 有着在本文评论的产品中最广泛的 UML 表示法的覆盖，注意，更重要的是，它几乎没有私有的 UML 扩展。

等待理想的工具

这些建模工具的任何一个都将帮助你达到你的组织信息和管理项目复杂性的目标，但每一个都带着它所具有的不同特性到工作台上，并使得你要占用些时间来成功使用它。GDPro 有着比其它两个产品更多的自由滚动的特性，并且，虽然版本 3.2 有着令人印象深刻的特征，但对我而言，它的用户界面有太多的不一致，我不能每天花 10 小时用它工作。

作为一种美德，Paradigm Plus 有着内在的一致性，而它确实给你提供个选择：是跳入到它的仓储原模型的深度里，还是只用它所支持的 25%的功能来建立模型。最后，StP 是一个有着持久 的能量，也是 Unix 平台上的一个主要的 OO 工具。但它强调 Table Editor，表明倾向于一种更老的、更以数据为中心的方法。

当然，这些工具中没有一个是开发人员所说的完美的面向对象建模工具。如果说这个回合的较量中，发现了一匹年轻的小马、一匹精制的良种马和一匹拉车的驮马，那么我将祈求新的参赛者要进入这个领域。我仍然在等待一个冠军。

原文链接 http://www.umlchina.com/Tools/acase.htm