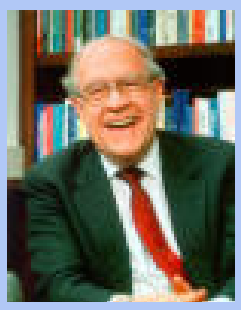
2002－11（第19期）



Fred Brooks

X-Programmer 非程序员

软 件 以 用 为 本

主编：davidqql

审稿：davidqql，think

投稿：editor@umlchina.com

反馈：think@umlchina.com

http://www.umlchina.com/

UMLChina

非程序员

软件工程杂志，已有数万用户

UMLChina讨论组

已有数万名成员

UMLChina.com

1999年11月成立，，专注UML技术的应用

专家交流

提供与世界级专家交流的窗口

Borland 以 1.85 亿美元收购 UML 工具厂商

[2002 年 10 月 30 日]在并购配置和版本管理软件厂商 Starbase 之后，Borland 继续它的并购狂热。该公司签署了最后的协议，以 1 亿 8 千万美元的现金和股票收购 Java 分析设计厂商 TogetherSoft。此项交易以 8 千 2 百 5 十万现金和 9,050,000 股 Borland 普通股支付。

Borland 这次在设计方面有目的的扩张对 UML 巨头 Rational 形成强有力的挑战。TogetherSoft 由 UML 领袖之一 Peter Coad 创立, 迅速在分析设计领域获得关注，与其他 Java IDE 一起广泛使用。Coad 有望在提升 Borland 在软件开发中的地位方面扮演重要角色。Borland 的 CEO Dale Fuller 说，“客户需要依赖于更快开发软件的能力，加速应用开发的生命周期很重要，这为我们提供了竞争优势。”（摘自 adtmag，think 译）

建模动力：UML2.0 使模型驱动的开发更加容易

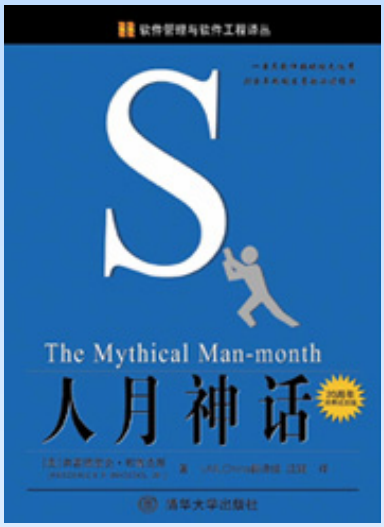
[2002 年 10 月]UML 规约的新版本将很快提交给 OMG，新的改动希望能够简化模型驱动的开发。Rational 公司新加坡分部的高级软件工程专家，Mark Hermeling 认为：UML2.0 根据工业界使用 UML1.x 的经验作了相应改进，目的就是为了帮助简化模型驱动的开发。UML 的目前版本是 1.4,它提供了方便开发团队在分析设计、需求管理等活动中进行交流的整套工具，以及一个软件开发生命周期模型。

在使用当前版本进行 UML 模型驱动的架构时，使用者发现还缺少一些支持，如 bug 修复等，UML2.0 中将增加这部分内容，它将成为适用于企业建模和数据建模的庞大而灵活的符号语言。在 UML2.0 中，将对语意部分进行增强，这一点可以帮助 UML 模型更好地生成代码，以得到更加实用的模型。在即将推出的版本中，还将包括增强的组件处理、对商业过程模型的支持，并更好地支持元数据交换。这些努力都是为了使 UML 作为一种胜过大多数文本语言的高层次的语言，能够生成代码和进行反工程，甚至直接生成某些可执行的 UML 模型。目前，在各种工具之间进行模型交换时，只能保存非图形化的信息，而象绘制的各种图、尺寸、坐标这样的内容都会丢失。在 UML2.0 中，将提供保留图形信息的能力。

Hermeling 认为，工程师与开发人员将越来越多地看到对建模的需求。他认为，对于一个较大的开发团队来说，需要有一个可视化的模型以保证所有人员都能理解总体的设计思路，建模的需求是显而易见的。利用业务过程建模，应用 UML 可以得到业务的可视化模型，其作用类似于建筑工程中的结构图。这个可视化模型可以使你在构造整个软件系统之前，就可以理解并预知设计的一些关键特性，判断设计是否可行。事实上，除了软件工程，在众多工程领域中，建模都是非常关键的规避风险的技术。

UML 正在将工具开发商们凝聚在一起，很多公司都参与了 UML2.0 的修改过程。除了 Rational 之外，Microsoft、Sun、IBM、Oracle、Borland、Telelogic 等公司也都是 UML 协会的成员。（摘自 computerworld，袁峰 译）

《人月神话》发行了！



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 汪 颖

二十五年后，我们仍然在读这本书

——Ed Yourdon

高焕堂：何谓世界软件之神

查看评论

编注：

北京时间 10 月 11 日晚上 19:00-21:00，高焕堂先生第二次作客 UMLCHINA 讨论组的聊天室。高焕堂先生是台湾杰出的资深 OO 专家，1995 年创办“物件导向杂志”和 MISOO 物件教室，在台湾实践和普及 OO 方法，育人无数，并著（译）有 8 本 OO 书籍。以下是交流实录。

ool: 我刚学了 UML，想在工作中使用它，但去发现不知道该怎样去做？请问是不是每个新手都会遇到这种情况，该怎么做去克服这个困难？

misoo: 我想，大家刚开始都一样吧！

idlecrook: ooad 是不是 RUP 最核心的部分？

misoo: yes!

idlecrook: 谢谢：）RUP 中的 OOAD 跟传统 OOAD 在概念上是不是一致的？

misoo: 我觉得是一样的「思维」。

idlecrook: RUP 中的业务建模属于不属于 ooa 的范畴？

misoo: yes, 它是针对 domain 的分析，ooad 是包括系统与 domain 两个层次的分析。

oceandeep007: 高先生：现在 UML 中没有针对架构的部分，你对此有何见解？

misoo: 不是 uml 的问题，是因为我们没有重视 architecture 的分析与设计问题。

powerpeople: 高先生：在 RUP 中，构架分析设计要做些什么啊，我觉得进行用例分析设计就可以了

misoo: 用例分析偏重于流程分析，但是架构分析还包括静态的结构分析，而且要包括企业组织的架构分析。

eport: RUP 中先启阶段必须做的事有哪些呢？总是很糊涂，好像很多工件在不同的阶段都出现，掌握不好要做的程度！

misoo: rup 的先期阶段应该是需求分析，尤其着重于 domain 分析，分析出 business 的 entity object。

ool: 请问新手如果发现使用 UML 无从下手，该怎么办？

misoo: 应该是物件思维要先重于 uml 的语法..大家好像只关心 rup 与 uml, 这都是表面的。但是重要的是，我们如何看待一个系统，以及这个系统背后的企业组织。我想，我们会比较专注于对系统分析与设计的观点和思维，至于 uml 与 rup，只是这些思维的呈现而已。

imcaptor: 高先生，架构设计要注意什么问题啊？

misoo: 关于架构(architecture)，应该先分辨 architecture 与 development process，如 rup 或 cmmi 之间的关系。

caidehui: 对于开发一个新的系统，高先生一般按照一个什么样的思路来进行工作呢

misoo: 不知道各位在接触到一个新系统的时候，会怎样去思考？一个系统就像一只蝌蚪，我们要来设计软件，给这只蝌蚪穿，请问，有人认为蝌蚪会变青蛙，有人认为青蛙会变王子，那么，你心中会认为是什么？认知不同，所设计出来的系统就不一样。

donquixote: 是不是意味着在设计阶段就要十分注意到系统的扩展

misoo: 答案是 "yes", 我们要注意到需求的变化，要 design for the future, for the change

idlecrook: 关键问题很多人不知道什么是需求，更不用说需求变化了。

misoo: 不知道需求常是起因于需求的变化。

ozzzzzz: 所以我做项目时先确定范围，然后才是具体的需求，需求总是会有这样那样的问题，refactoring，所以我不怕有问题。而我比较喜欢 agile 的思路 所以我不会为未来考虑太多。

misoo: refactoring 的目标是 refactor the architecture design, and when to do it?

happyhummer: 现在你为将来进行的预先设计，并不一定符合将来的需求!!

misoo: 对了！这就是 architecture design 的真正意义，它必须 mapping from the vision to the reality。

montling: 高先生,您好！ 请问 uml 中如何实现业务模型向系统模型的转换？

misoo: 可以参考「物件导向 12 期」，两段式 ooad 的思维。

donquixote: 对于需求的变化，需要注意。但是怎么做才能做到恰到好处呢？

misoo: 对于需求的变化，我们必须保持 architecture 的弹性，也就是藉由 component 的 interface 设计让 component 能够很易于抽换。保持弹性与元件的抽换性可能是一个最佳的 pattern。

idlecrook: 先进行需求建模还是先进行业务建模？

misoo: 如果有需求不变的系统，那就不需要太多设计了，只要编写代码就行了！

idlecrook: “我们必须保持 architecture 的弹性”会不会导致过分设计？

misoo: 不用担心，因为 architeture 是抽象的，而不是像代码是具象的。architecture 比较不容易以代码表现出来，xp 在 architecture 的 design 上，虽然也蛮重视，但并没有太多的 how-to。

happyhummer: 请问怎样才能实现 architecture 的弹性？这才是我们关心的问题！是不是靠经验和灵感？

misoo: 『序中有乱』，先建构次序，然后，允许变化。易经的坤卦里头，强调繁杂，它意味着繁荣、复杂与多变化。所以，「序中有乱」，才能产生软件的威力。

caidehui: 架构的设计是在什么时候做的呢？

misoo: 架构设计在 project planning 的过程中要先进行，然后，依据 architecture 的设计来 plan the project。

idlecrook: 也就是说如果我们改变一个 architeture，就意味着对已有代码做很大改动（甚至抛弃），可是有的时候似乎显得有点浪费

misoo: 所以 architectue 设计的过程本身必须采取 iteration and prototyping。

uml2000: 高先生，您好！请问用.NET 的技术来实现一个大型的 ERP 系统，您认为什么样的架构最为合适？

misoo: 要把客制化与非客制化部分分离，非客制化部分转变成 framework。

smilemac: 高先生,您好,我认为 architecture design 的目标应该是为了更简洁地表达目前的 problem field,而不是为了未来的变化，能够适应未来变化只是简洁表达的一个副产品，您认为呢？

misoo: 关于这点，有一本书，写得不错，就是「uml components」，它也认为今天软件所面临的最大问题是 everything is changing，它也提到，component 的易于抽换，是当今最重要的技术。

idlecrook: 我的意思是 iteration and prototyping 的代价会不会太高？如何把握？

misoo: 如果我们要 design for the change and for the future，只有 iteration and prototyping 才是最可靠的，不可靠的代价才是最浪费的。

loseinworld: 高先生，请问用例是否都是先由用例主角驱动的？

misoo: 不一定，有时候是由系统里头的 event 驱动的，或者是由某个 use case 驱动的。

caidehui: 架构由哪些部分组成呢？

misoo: caidehui, 想一想一个高楼大厦，它的架构，就是栋梁，这是架构的一部份，但是可以从这里开始想起。架构最主要包括 component 的 interface, 所以 interface 的设计，是很重要的，而且是大型项目分工的依据。依据 architecture 而分工与 cmmi 方式的接力赛分工是不一样的。

smilemac: 高先生，谢谢，您对耦合度在系统设计中所处位置有什么看法？即 coupling。

misoo: 现今系统整合的趋势会比较朝向「loose couping] 来结合，例如 web service。

misoo: 大家会不会觉得谈「desing for the change」似乎远离了 software engineering 方向呢？.NET 今天所强调就是 web service, 那就是希望各 sub-system 或 component 都提供标准 interface, 让这些 sub-system 或 component 都能轻易抽换，也就是达到 "plug & play" 的目的，因此，design for change 是大家应该关心的事。

powerpeople: 高先生：我一直不太理解 interface，能解释一下吗？

misoo: 像汽车的轮盘就是 interface, 而轮胎是 component, 有了 interface，就能随时把坏的轮胎抽换掉，结果 reuse 整部车子里其他的好的 component。

montling: 高先生，对于 UML 建模，有人认为先 USE-CASES，后域模型，另有人认为是先域模型后 USE-CASES，您认为呢？

misoo: 如果对 domain 不是很熟悉的话，从 use-case 着手比较方便，那么如果对 domain 很熟悉的话，从 domain 模型着手比较理想。

smilemac: 高先生，是的，但是我认为耦合度在整个 architecture 设计中应该处于一个基础及导向的位置，它也是问题域的基本属性之一。不知您这么看待。谢谢。

misoo: coupling 的决定必须依据 domain 的变化而决定的，例如，变化的起因有所不同的话，就必须设计 interface 来降低其 coupling。

uml2000: 能谈谈 use-case 的粒度问题吗？

misoo: use-case 的粒度应决定于 user 的 view, 但是大家常认为是由 developer 决定，所以常会伤脑筋，但是，我并不为它伤脑筋。如果，两个 use-case 可以收到比较多的钱的话，我会做成两个 use-case :)

powerpeople: 高先生：我们现在正在进行一个项目，先启阶段已经结束了，现在应该是开始架构设计了，但我不知道现在应该做些什么，以前我们都是直接开始用例分析设计的。

misoo: 我可以介绍您一本比较完整的书，就是：architecture-centric software project management, 你可以仔细看看。

smilemac: 高先生，谢谢，一个对象的接口容易设计，但一个对象的 side-effect 却比较难控制，请教您有没有什么经验或观点，关于 side-effect.

misoo: 可否请您举例 "side-effect"有哪些？

caidehui: 这样作会不会增加成本

misoo: 如果能增加收益，能给 user 更多方便，为他考虑未来的需求，即使增加一些成本也是划算的。

timwap_cn: 高先生，拜读过你的文章，非常同意你的观点。不知你对大陆的软件业有何建议？

misoo: 希望在推动 cmmi 的时候，也重视 architecture 分析与设计。

ufjl1572: What's the difference between sub-system and component?

misoo: 这是 "level of view" 的问题，sub-system 也可以看成 component, fine-grained component 通常是指 entity, 而 sub-system 是 coarse-grained。

hrun: 请问在台湾的公司中，PM 的地位和职能是怎样的呢？

misoo: 台湾的公司，把 PM 比较当作是 "Resource Support"。逐渐的，PM 与 CSA(chief software architect) 两者相互合作，共同领导项目的进行。

caidehui: 大陆 PM 基本上都是 Tech Leader。

misoo: 基本上，PM 与 CTO 最好能在 CSA 的节制之下，而由 CSA 领导开发团队，并与 Boss 站在平行的地位。cmmi 的不足是 CSA 的角色并不明显，但是，我们看看 microsoft 公司, BILL GATES 的头衔是：CSA，Rational 公司的 Grady Booch 也是 architect, 所以我们谈 PM 也不要忘记谈 CSA。

s200: PM 需要懂技术吗？

misoo: 如果有 CSA 的角色，那么 PM 可以不太懂技术。

caidehui: CSA 这类人才不好找

misoo: 是的，不好找，但是，qualified 的 PM 也不好找。

smilemac: 在大陆，很多 CSA 与 PM 是一个人，有的公司是 PM 和一个 SE(SENIOR ENGINEER)配合。

misoo: CSA 在估计 schedule 及 cost 时，会采取 top-down 以及 bottom-up 两者一起进行，但是，一般 PM 都偏向 top-down。

timwap_cn: CSA 只是负责技术？

misoo: 记得，CSA 负责 architecure, architecture 分析包括需求分析、组织分析、技术分析、热情分析，所以 CSA 负责的范围是很广的，但是，是抽象而重要的，技术只是 architect 负责的环节之一。

montling: .NET 技术在台湾应用的情况如何？您认为该技术还有什么急待改进的地方？

misoo: 从 .NET 中，可以学习到不少好的 design 技巧，并且，简化实作上写码的难度。

frankxing: 我们平时在做项目的时候经常会发现我们的需求，设计，编码往往是一个反反复复的过程，有时候的改动是非常大的，请问您认为在这方面应该怎么尽量减少这种重复及其造成严重后果？要知道有很多时候在开发一个项目时时间压力都是很大的，我们的考虑总不可能尽善尽美，总是在进行中还会发现问题。

misoo: 反复过程，分为两阶段，第一阶段是针对 architecture, 第二阶段是针对 parallel development 的反复，如果没有第一阶段，那么系统的修正会很大。

smilemac: 请您介绍一下“热情分析”好吗？第一次听说，很有意思。

misoo: 最近有一本书叫：from good to great, 里面有比较详细的说明，没有热情，一定不会有好的 design，所以，CSA 必须分析团队的热情。

s200: .net 的分布式构架要求不常连接数据库的，对于需要大量录入、修改操作的应用，是否有问题？

misoo: 一般而言，这是写码人员很大的 "盲点", 想想，100 万笔资料是不可能藉由如 DataSet 元件来 disconnected, 但藉由 Cursor, 又回到了 two-tier 的时代，也是不对的。软体的 design 是不应该将投资在于其中的。

nongren: 请问高先生，我刚开始学习 UML，同时从事系统分析工作，虽然买了不少 UML 方面的书，可应用起来还是不知道从哪下手，请问我该如何去做？

misoo: 也许，你可以从 CRC 的技巧练习，然后才使用 UML 表达出来，先强化 object-thinking, 可能容易下手。

s200: 不投资在其中？难道你指望它能自动完成？

misoo: 我的意思是，应该要分工，designer 不需要考虑特殊 platform，留给 programmer 决定即可，软件的架构是可以独立于 platform 的。

caidehui: 我还没有见到在 CRC 上有一定造诣的人，大陆的 OO 基础也不太好。

misoo: 大陆的软件人员如果能够 designer 与 programmer 互相做明确的分工，可能会更好。

smilemac: “designer 不需要考虑特殊 platform”，但是如果设计采用了 multi-thread 技术，而实现时却发现 unix 对 thread 支持不好，那怎么办？

misoo: 我们强调的是分工，而不是一个团队都不需要有 platform 的人才。

s200: 如果 designer 都没有 programming 的功底，能做好 design 吗？

misoo: 答案是：可以更好！

s200: 不懂 programming 的人去做 design, 会做得更好？我真的不明白了，能详细一点吗？

misoo: 请问张良他也没带过兵，为什么他能够运筹于帷幄之中，而决胜于千里之外呢？

frankxing: 我认为没经历过 programming 的人，肯定不能做出好的 design

misoo: programming 能力太好的人，做不好 design 的原因，就是古代的人所说：「文人相轻，自古而然」所以，团队要做好分工，才是项目成功的保证。

smilemac: 高先生，所以张良可以做军师，但不能做大将呀。

misoo: designer 也不是大将，CSA 才是领导者。

frankxing: 帅才可以没打过仗，但是将才却必须是身经百战。张良在现在可以做一个好的项目经理，但做不出一个好的 design。

misoo: 项目经理可能是「萧何」比较恰当。

smilemac: 项目经理应该是韩信

misoo: 如果，项目经理是定位为 "resource supporter"，那么，他会是负责后勤支援部分，所以，萧何会是比较恰当的。

timwap_cn: 高先生，构架能够被验证吗？

misoo: 架构是抽象的，所以必须透过 iteration 不断产出具象的 prototype 来验证，就像先建构飞机的 prototype 一样。

caidehui: 分工也是一种 Interface，越大的工程分工要越细

misoo: 对的，所以我们需要很好的 CSA，才能够建构像高楼大厦般的巨大系统，才能做好 parallel development, 能够大幅缩短项目的时程。

s200: 高先生，在大陆有软件工程专家说 uml 存在三大硬伤，“上不着天，下不着地，一盘散沙”，是说它无论对系统分析员还是对编程人员来说都并不实用，你对此有什么看法？

misoo: 我想 component-based, architecture-centric 是重点，否则 uml 是没有用的。

timwap_cn: 高先生，飞机的 prototype 也会在风洞中进行试验，那软件构架建立后，是否也应测试？

misoo: 当然。

caidehui: 这是不是意味着系统的开发实际上是处在架构的驱动之下

misoo: 答案是：Yes, 所以必须 architecture-centric, 所以我们一直主张大中华地区应该推广 architecture-centric cmmi。

timwap_cn: 高先生，构架师的工作是依据经验，还是一定的方法，如果有方法，那方法是什么？

misoo: 架构师的经验是很重要的，但也有明显的原则和修炼方法，请参阅 www.wwwisa.org

s200: 一个新人，是否可以直接按照 designer 的要求去培养？

misoo: designer 和 sa(system analyst) 应该是不容易分开的，所以，大家才会常提到 ooad，那么 ooad 的主要工作是：分析 domain knowledge 的架构，然后，作为软件系统的架构，您可以从 CRC 着手训练，每一张 CRC 卡表达一个 domain concept。

caidehui: 架构驱动项目本身比较清楚了，那什么来驱动架构呢

misoo: iteration 与 prototyping 来不断修正 architecture 的分析与设计。吴清源大师说：「整体、和谐、创新」代表的即是在以架构为主的设计上，要能注重团队的和谐、一致性。在团队的脑力激荡下，是需要发挥 design 的创意的。「兵无常势，水无常形，能因敌之变化而取胜者，谓之神」，希望能藉以孙子兵法的这句话让各位对软件的 design 有更多的深思。因为软件的需求是善变的，所以善于利用需求之变化而取胜者，将能称霸软件世界，谓之：世界软件之神。

caidehui: 请您为我们进一步学习和应用架构指定一些资料

misoo: http://www.wwisa.org 有许多文章。

montling: 请问高先生，构架作为软件产品的表现形式，表现在哪些地方？

misoo: Inerface!

montling: 高先生，您这里的 Interface 是否组件技术中的接口？

misoo: 是的，就像汽车的轮盘。

goldenstone: 我非常想知道台湾，美国，日本，印度和欧洲这些相对发达地方的开发团队的组织情况？

misoo: 美国与印度是分工的，所以美国团队比较重视 CSA，CSA 代表 leadership，印度比较重视 PM，PM 代表 management。台湾目前的方向不明确，正在寻找中，也是迷途中。

jasonchen3: 这样会不会有一个问题：已知的不足，造成 Interface 不稳定呢？

misoo: 知之为知之，不知为不知，是真知也。知道的部分，就直接写成代码，不知道的部分，就实现为接口。

goldenstone: “凡事皆项目”，这是不是主要来自印度的经验？

misoo: 印度比较重视项目管理，这印度经验及模式是否让大陆发展太快，却忘了寻找自己的方向！

尤克滨：简单正是用例的价值

查看评论

编注：

北京时间 10 月 21 日晚上 19:00-21:00，尤克滨先生作客 UMLCHINA 讨论组的聊天室。尤克滨先生是瑞理(Rational)软件公司的技术顾问，研究领域为软件需求。以下是交流实录。

higoals: 组织需求文档时，可以结合设计方案吗？

barbeque: 需求文档的核心内容是用户的问题，和设计方案是一对多的关系，过早考虑方案不宜

vbjava: 请问在系统需求分析阶段，UML 还不成熟，无法与流程管理软件，如 Aris, SA 相比？

barbeque: 需求的主体内容是文字，图示作为辅助，UML 主要是可视化需求的概要内容

higoals: 那么如何组织用户的需求呢？我脑袋中老去不掉未来可能的设计方案啊

barbeque: 用户的想法需要被有条理的捕获出来，而不是随机的捕获，然后在用设计人员的思路组织

gallenliu: 有人说：最清楚的需求文档就是 user manual,你是否同意？

barbeque: 清楚的需求文档有助于快速地建立 user manual。需求的内容应该是 user manual 内容的超集

myzoucp18: 请问做需求时，是不是可以将性能方面的问题放在一边？

barbeque: 不是，性能需求是非功能需求的重要组成部分，尽管和应用逻辑没有直接的关系，但不等于不重要

evpu: 那么,uml 其实不重要,对不对?怎样才算有条理呢?

barbeque: 不是 UML 不重要，要针对场合。Use Case 的需求捕获强调的是文字描述。有条理就是：用户能够自己去直接修改需求的内容，而不需要设计人员辅助，或者说，遵循用户的逻辑。当然这主要是指功能需求。

vbjava: 业务建模工作是 IS 需求分析的常用方法，这方面是不是 UML/ROSE 也不能给我们帮助？

barbeque: 不尽然，至少泳道图是很实用的。

myzoucp18: 谢谢，你说用户的想法需要被有条理的捕获，但中国用户大多无经验，又如何做到这一点

barbeque: 没有经验的人未必心中没有需求，需要我们的引导，Use Case 是一种好方法。

jasonchen3: 如同 chinalbh 所说，我们也同样遇到需求无法稳定的问题，不知 Barbeque 有什么看法?

barbeque: 需求不稳定不见得是消极的，通常情况下，需求的变化是积极因素的反映，尤其是早期的变化。

evpu: 啊?!用户自己去直接修改需求?这个动作是发生在需求还没确定的时候,是吗?这样的话,就是说以用户的观念(思路,条理)来组织,设计需求的结构和细节,是吗?

barbeque: 用户有能力自己修改需求的内容，隐含的意思是：用户完全理解我们写出来的东西。

jasonchen3: 其实我也非常赞同应该顺应使用者的需求，只是我们可能需要将这些改变反映到成本上…

barbeque: 用户能够理解的需求至少能够带来两方面的成本降低：指定 test case 和 user manual. 因为对等内容的异构表现形式通常大幅度增加成本。

higoals: 在需求调研中，用户和我，谁应该是主导角色？

barbeque: 有两种情况：如果是软件适应稳定的业务模式，必须由用户主导；如果软件起到改变业务模式的作用，很可能是代表领域专家意见的人员主导

evpu: 我觉得,use case 就一个小人一个圈,再加一个箭头,其实还是要靠大量的文档,最大的用处还是在于圈定范围,是不是?

barbeque: Use Case 就是以文字为主体的功能需求描述方式，同时也是一种捕获方式。Use Case 图可以看作为 Use Case 需求的“主页”

perl5: 想请教您如何把握界面原型在软件需求中的位置？

barbeque: 界面原型在软件需求中的作用很大：建立用户界面原型的主旨是在启动实质开发活动之前揭示和检验拟建系统的功能与可用性，在大规模投入资源之前，提供一个确认拟建系统正确性的机会。

chen09: 俺觉得不存在“用户不能够理解的需求”，因为如果用户不理解，那么用户就不会允许你继续往下开发。

barbeque: 用户一般不会说看不懂我们写的需求，而是等我们开发出他们认为不对的结果之后告诉我们，原因是我们没有在早期提供一个用户作出正确判断的机会（组织形式）。各种方式表述的需求在微观层面没有差异，但组织形式完全有可能阻碍用户正确地理解。

Charity_Zhou: 有必要对用户进行 UML 的培训吗？

barbeque: 不懂 UML 的用户完全可以帮助你得到高质量的 Use Case 模型。

vbjava: 是不是 use case 是给系统架构师和系统分析员沟通用的，而不是给用户看的，即便在 use case workshop。

barbeque: 不是，Use Case 模型是用户眼中的 Architecture. 当然 Use Case 可以为后续的开发活动提供诸多便利，它将需求按照用户的目标以高内聚、低耦合的方式组织在一起。

perl5: 如果有很多类似的功能(如多种不同的查询)，是画一个 use case 就可以；还是画多个，并使用“泛化”关系？

barbeque: 好的 Use Case 可以不使用 Use Case 之间的关系表述。Use Case 之间的关系主要是用于优化 Use Case 模型本身（目的是便于维护需求的内容），而不是更好地反映用户的要求。

vbjava: 采用以文字为主体的 use case 表达方式是不是说明 uml 在 ooa 方面还不够成熟呢？

barbeque: 作 Use Case 并不是做 OOA。

vagawind: 但是 use case 是 OOA 很好的表达形式，对吗？

barbeque: 不对，OOA 是用 OO 的方式将需求转述出来,其要素是所谓的分析类.Use Case 中的内容是用自然语言表述的需求,是 OOA 的理想依据.

rareren: 怎样把一个或多个功能作为一个 UseCase（如 Insert,update ,delect 分开还是做为一个维护）

barbeque: Use Case 是某类用户的目标,达到一个目标可能会有多种可能的场景,每个场景中有若干个动作(交互). 通常的误区是将动作或者场景当作 Use Case. 注意 Use Case 中的 Case 和 Test Case 中的 Case 是有区别的.

ltxd: 能不能详细说明一下用例这间的三种关系的区别

barbeque: 比如你上楼,你也许必须要走楼梯(include),当然你有可能在中途去一下洗手间(extend).

ltxd: 在<统一软件开发过程>一书中,居然说用例之间的"泛化"就是 jacbson 在<basic use case modeling>中说的"use",对吗?

barbeque: 泛化的关系主要用于说明概念. 举一个不恰当的例子:做饭是一个 Use Case, 那么煮饺子 Use Case 和煮面条 Use Case 与做饭之间的关系类似于泛化

zhouxp2005: Use Case 中的 Case 和 Test Case 中的 Case 是有区别的？有什么区别呢？

barbeque: Test Case 中的 Case 是个案, Use Case 中的 Case 并不是这个含义

higoals: 是否是用例，须看是否实现用户的一个目标，是吗？

barbeque: 是,包括实现该目标的各种可能场景

casual_wen: 我想问一下，use case 应当由什么人来负责绘制呢？BA 还是 SA?

barbeque: 不一定,USe Case 的易理解程度甚至允许用户自己绘制(实践说明)

montling: 请问在将业务模型转化为 USECASE 时有什么技巧？

barbeque: 业务用例模型和业务对象模型和 USe Case 模型有比较简单的映射关系,正是由于简单,导致灵活性较高. 技巧在 RUP 中有所涉及.

txiaofeng: use case 是否存在最初原型和分析模型这两种概念？

barbeque: Use Case 是面向用户的,主要是记录应用场景,还没有到分析的时候.

ltxd: 问一个 RUP 的问题,在初始阶段末, 应该建立一 candidate architecture, 是不是意味着要建不止一个以备选择(在某一个失败时)?

barbeque: 构架师的经验通常能够保证 candidate architectue 的有效性,当然不排除更换的可能.

ltxd: 在做某些大型系统时,是否可以考虑顶层的 use case 只做几个, 然后针对每个顶层的 use case 再进行细化, 对它又再建 use case model, 这样形成层次关系, 保证单个平面单个区域上的 use case 不要太多呢?

barbeque: 不是. 有些文献中建议在不同层面应用 Use Case 的概念,但并不意味这不同层面的 Use Case 之间存在细化的关系. 不同层面主要是指用户对系统理解的不同概括程度.

openeyes: 我想问一下，象库存不足时自动触发采购流程这样的 Use Case 的 Primary actor 是什么呢？

barbeque: 要看如何获知"库存不足",如果是采用"轮询"的方式,可以用时钟表述.

casual_wen: 那么经过需求分析阶段，应当产生哪些文档呢？

barbeque: 不同的规范有不同的文档体系,如果是参照 RUP,主要有"前景","Use Case 规约(组)",界面原型,补充规约,词汇表

ltxd: 那这么说 use case 就应该存在一个特定的粒度大小问题了，即可以概括地说一般一个 USE CASE 映射到多少个分析类,多少个设计类，能否给出一个数字，我现在就是把握不好 USE CASE 粒度的大小，谢谢!

barbeque: Use Case 通常不能一蹴而就,开始先得到 Use Case 规约的骨干内容:基本事件序列的步骤和备选事件序列的名称. 横向比较 Use Case，此时有些明显过大或者过小的 USe Case 可以进一步分解或者合并,当然不能影响用户对其基本内容的理解.

powerpeople: 用例规约中的，基本流和备选应该怎么理解啊？

barbeque: 你做 100 次同一目标的事情,80 次的情形类似,它就是基本事件序列.

txiaofeng: use case 的分解与合并依据什么？

barbeque: Use Case 的内容更易于理解和管理.

openeyes: 是不是可以这么说，“事件”可以作为 Use case 的 Primary actor.比如计时器事件？

barbeque: 要看如何获知事件的出现.如通过定周期的轮询

txiaofeng: 对 use case 的分解、合并是否可以基于将来自系统、甚至功能的划分？

barbeque: 不是

powerpeople: 请问，我一个仓库管理系统中，增加出库单和修改出库单是不是应该分为两个 use-case 啊

barbeque: Use Case 的划分主要是和用户直接讨论,你的问题不能脱离特定的上下文.

jasonchen3: 请问，一般的系统的 Use Case 个数维持在多少个会比较恰当呢?

barbeque: 针对一个特定的上下文,Use Case 模型并不存在唯一的答案.

ltxd: jacbson 的 use case 的定义是否只是针对所谓的 concrete use case 的(而不是被 included, extended 和被 inherited 的?)

barbeque: abstract use case 已经不是具有基本含义的 Use case...在应用 USe Case 方法捕获需求时,建议不使用 Use Case 之间的关系. 这种关系是当 Use Case 模型已经相对完备之后用于优化 Use Case 模型本身的, 对用户理解需求并没有显著的积极贡献

casual_wen: 但是如果 use case 之间不建立关系的话，use case 是不是太简单了呢？

barbeque: 简单正是该方法的价值

rareren: 怎样说明一个事件流的开始？

barbeque: 说明起始条件 Pre-condition,然后开始描述一唱一和的交互.

powerpeople: 前置条件和后置条件怎么理解啊？

barbeque: 后置条件的翻译容易引起误解, 建议理解为"结束状态"

powerpeople: 比如说，我增加了一条信息，然后需要刷新界面，这算不算是“结束状态”啊

barbeque: 也许"界面被刷新"和"界面维持不变"更类似于"结束状态"

rareren: 不过我有点搞不明白一个 Use-Case 到底什么时候开始？如一个查询功能从打开查找条件编辑界面之前说起呢?还是之后说起?

barbeque: 从用户需要做的第一个动作开始

powerpeople: 那上面的例子是不是要从用户按查询按钮开始说起

barbeque: 要看用户是不是愿意从按"查询"钮开始. 也许是从一个 link 开始...

openeyes: 对一个企业的库存不足时采购，是不是没有相应的 business use case？

barbeque: 要看有没有 biz actor

perl5: 组织边界之内的人员可以做为 biz actor 吗？

barbeque: 看怎么定义"组织"

rareren: 其实我觉得难处就是在于对不同的 Use-case 用户需要做的第一个动作到底是什么?也就是 use-Case 怎样划分的问题吧？不知是不是这样?

barbeque: 不是,Use Case 的划分的关键是用户最终要什么结果...

perl5: “从用户需要做的第一个动作开始”，那不是每次都要从系统登录开始？

barbeque: 通常可以从 logon 之后的动作开始,因为 logon 是每类人做每类事的必经之路,它通常被当作前置条件,并且 logon 单独作为一个 Use Case. 几乎成为惯例..

openeyes: 具体的说，企业的客户是不会对企业的库存有什么 goal,库存不足的采购自能是企业内部的要求。这时候对企业作 Business use case 时，这个采购流程应该不会出现的吧!那么，这样对企业的业务分析是不是太不足了？谢谢

barbeque: biz use case 是对业务的外在可见内容建模,这并不是业务模型的全部...你说的内容可以通过 biz object model 描述.

Charity_Zhou: Jacobson 在 software reuse 一书中强调 UC 是可复用的，关键是要寻找变点和变体。但在 ROSE 中无法体现变点以及变体

barbeque: 广义上,Use Case 可以复用主要指其内容及组织形式可复用. 并不是强调 Use Case 被其它的 Use Case 复用(尽管可以),因为这并不能体现出对 Actor 的价值.

perl5: 但流程性的东西如何通过 biz object model 描述呢？我们是为每个 biz use case 用一个 activity diagram 说明其流程，这样做法对吗？

barbeque: 流程可以通过泳道图表述,当然,由业务对象参与的时序图也可以用于描述流程。如果一个图中的内容很烦杂,也可以用多个,通常标识出泳道比较好。业务建模的目的往往是为软件需求建模和分析设计提供一个上下文环境。

powerpeople: 怎么理解子系统啊?子系统和包有没有什么关系啊？

barbeque: 包就是一种一般的组织方式, 子系统强调行为特征(其内容通常组织在包中).

Charity_Zhou: 这样一个例子：典型的政府待业审批，多个业务，流程是一样的，不同的是业务对象以及涉及的业务角色以及细节上的一些处理，是否是每个业务都要做相应的活动图呢？我觉得是没有必要，通过标志出变点和变体，就可以很好地解决这个问题，但是在图中无法表示，必须加以说明，不直观

barbeque: 先找出一些候选者,然后和用户讨论. RUP 有一些获取 Actor 的典型问题可以参考

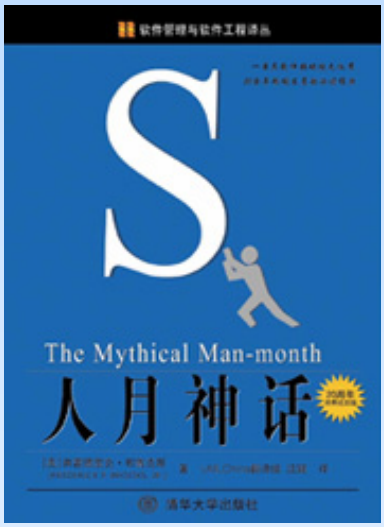
wy666666: 我觉得因为 OO 就是分析、设计的技术，这与业务建模、需求是没有什么关系的。业务建模、需求完全可以用非 OO 的技术来实现，经过讨论，甚至发现根本不必用计算机去实现都是有可能的！OO 并不是业务建模、需求的必要条件。您认为呢？

barbeque: 对。不过,OO 是可以用于业务建模的...:)

征稿

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

《人月神话》发行了！



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 汪 颖

二十五年后，我们仍然在读这本书

——Ed Yourdon

致面向对象技术初学者的一封公开信

Alistair Cockburn 著（1996 年 2 月），袁峰 译

查看评论

介绍

首先我要解释一下为什么会写这封公开信。这似乎已经成了一种习惯，但这个步骤还是需要的。过去 6 年中，我曾经无数次地在饭店、酒吧、旅店大厅等各种地方以同一种方式度过愉快而漫长的夜晚：和同样追求真理、光明和智慧的伙伴一起探讨面向对象的真谛。现在，我已经可以回答很多当年我遇到的问题。这些同样的问题也在困扰着我的一位新同事，在一家饭店里，我花了整整一个晚上和他讨论这些问题。结果第二天，他的同事又来问这些问题，并建议把我们的谈话内容记录下来，这样他可以拿去给他的同事看。考虑到还有很多和他的同事一样询问这些同样问题的人，我决定写下这篇文章。

主要的问题是：

- 为什么只要稍有一点不是严格或纯面向对象的做法、说法，OO 专家就大惊小怪的？use case 并不是面向对象的，为什么还这么流行？而且，OO 建模似乎和数据建模非常相似？
- 数据建模（data model）得到的模型和对象模型的结构部分会不会很像？流程建模（process model）和对象模型的行为部分呢？
- 业务结构建模中为什么要用 use case 和场景 （scenario）？

OO 的新手们反复问这些问题，但实际上，只有在日常工作中坚持应用面向对象的思维进行工作，积累一定的经验，才能得到满意的答案。

为什么只要稍有一点不是严格或纯面向对象的做法、说法，OO 专家就大惊小怪的？use case 并不是面向对象的，为什么还这么流行？而且，OO 建模似乎和数据建模非常相似？

我想分三步来回答这个问题。

首先，我举我和 Bob（著名的 OO 专家）一起工作的例子，当我们讨论 OO 的时候，彼此都有一个共识，知道对方拥有面向对象工作的丰富经验并且是这项技术的坚定支持者。而且，对诸如对象识别（object identity）、多态、数据和行为的封装、实例职责 、继承等对象技术都是手到擒来。因此，当我说：“明天对程序表单进行数据建模吧”，Bob 不会产生我要会因为关系表而放弃对象这样的误解，他知道我指的是在对象模型中体现出来的结构化特性进行建模。他知道我会说些什么，因此我使用或误用这些术语不会造成什么误解。

但作为一个对象技术的初学者，如果 Bob 发现你把数据和行为完全分离开了，并且没有使用（或者说忽视了）对象识别或者多态等技术，这时候，如果你说“数据建模”，Bob 会像一堵墙一样逼近你，直到你明白该怎样改变。这样工作几个月，你会发现，你的模型（以及建模）中渐渐有了对象识别、多态、数据和行为的绑定，这时候再用“数据建模”这个词就不是那么危险了，但 Bob 还可能会担心你走回到老路上。换句话说，他对你还不够信任，因此，你不得不很小心地使用这些术语。

就算一个对象模型可以分为“结构”和“行为”特性，我们也不会使用“对象建模”和“流程建模”这种术语，以免引起混淆。事实上，为对象模型的“结构”特性建模可以看成是数据建模的特殊形式，只不过建模的对象不再是表，而是需要捕获的信息的结构。我们将它称为“概念数据模型”，而不是逻辑数据模型或物理数据模型。

第二步，让我们考虑两个 OO 使用者一起讨论的情况。如果其中一个家伙说到“流程建模”这样的词，肯定会让他的拍档琢磨半天：这家伙是说用标准数据流图作流程建模吗？如果这样的话，以后 OO 实现的时候不是相当麻烦了吗？他是不是指模型的行为特性？是不是说在一个对象内部对流程进行建模？（如果这样的话，那会很有意思，因为很少有人这么做的。）通过这个例子我们可以看到，这种谈话中使用“流程建模”这种意图不明的词实在是太危险了，很容易就将交流变得非常困难。

最后来说 use case 和场景的问题，它们都是获取需求的主要手段，和实现技术无关。其好处是可以为设计时的讨论提供内容和范围。它们不是“面向对象”的，这是事实，它们类似于功能分解，这也是事实，而且这一点吓坏了很多人，但这些都无所谓。重要的是它们为设计提供了内容，在用来描述内部设计时，它们表现了系统的行为。Flow chart、交互图、Petri 网、数据流图、use case 都可以用来描述系统的行为特性，但各自用途不同，各有优劣。关键是要知道：对象不仅有数据，也有行为，认识到这一点，就可以大胆地去考虑怎样可以更好地捕捉、描述对象内部和对象之间的行为。

数据建模（data model）得到的模型和对象模型的结构部分会不会很像？流程建模（process model）和对象模型的行为部分呢？

根据我的经验，数据建模人员可以分为两种－一种是为存储的数据建模，而另一种是为系统或组织中的信息建模。这两者截然不同。

前者讨论和思考所用的概念通常都很具体，比如说数据表。他们的模型和 OO 建模者的模型大相径庭，而且他们并不愿意看到这些技术之间的相似性。在数据建模社区中，只有讨论逻辑数据模型或物理数据模型才不会受到攻击

后者得到的是概念数据模型。根据我的经验，他们得到的模型和那些有经验的 OO 建模者得到的模型非常相似。他们的工作更加类似于业务人员，而不是逻辑数据建模人员，这种说法可能会有助于理解概念数据模型和逻辑数据模型的区别。

似乎有一套学问可以帮助人们比 OO 建模人员更快地得到结果。我多次发现这样的事实：OO 建模人员花了三四次迭代才得到的模型，实际上和（概念）数据建模人员一个循环得到的模型是一样的。这个发现使我对数据建模人员充满了敬佩。事实上，在进行对象设计的时候，我有时就直接去把数据建模人员的产品拷贝过来看看，从中来看我最后得到的模型大概会是什么样的。

我曾经召开过数据建模人员和对象建模人员之间的会议。采取的方法是“一个听众的 CRC 会议（CRC sessions for an audience）”。四个经验丰富的 OO 设计师坐在长桌一端，业务专家沿着长桌坐下，他们负责回答问题并纠正对业务的误解。接着是数据建模人员，长桌的另一头是其它有关人员。总的来说，屋里大概是十几个人，但谈话主要是在四个 OO 设计师之间进行。

讨论大概一个小时之后，OO 设计师可以得到对象设计的一部分。这时候，咨询数据建模人员，这个模型和他们已经得到的模型本质上是不是一样的。他们说，“是的”，重复这个过程两次以上，每次都询问数据建模人员同样的问题，除了使用的符号技术是不同的，例如：他们没有用继承，但在同样的地方有同样的占位符，在本质上，这个模型和他们的模型没有什么不同的。接着，我们分成几个小组，每个小组包括一个业务专家、一个数据建模专家和一个 OO 专家，很快就剩下的设计达成一致，找出技术上一些小的不同之处，并且进行排序。一般情况都是这样的：要么数据建模人员考虑得更加合理，要么 OO 建模人员考虑得更加合理，小组要做的是在他们的设计之间进行协调。

从上面的这些经验可以看到，使用 CRC 进行 OO 建模得到的模型和概念数据建模得到的结果非常相似。

另外，根据经验，基于逻辑（存储的信息）的关系建模和 OO 建模是不同的。大多数情况下，区别是由于技术的不同导致的，例如，在 OO 模型中可以自由地使用继承和多对多的关系。由于技术上的差异，两种建模人员之间不能很好地交流，这是最大的困难。

数据建模部分的问题就说这么多吧。

对流程建模而言，情况却不一样了。90 年代初，涌现出各种方法、计划、会议和项目，试图将数据流图的模型转变为对象模型。曾经有几个项目声称获得了成功，但都是后续无音，业界一致的结论是：这个转变太困难了。大多数使用数据流图的建模人员最终都抛弃了这项技术，投入了对象设计的怀抱（Martin-Odell 和 Ptech 小组是著名的例外）。对于流程建模，现阶段我的回答是：其模型无法与 OO 模型相比较。

但这并不是最终的回答，OO 建模人员正在开始进行本质上的流程建模，下一步的发展将会怎样，我也并不清楚，流程将变成过程那样（过程编程复活？），或者变成数据流图那样，还是类似于封装的对象？

业务结构建模中为什么要用 use case 和场景 （scenario）？

use case 和场景……

……为讨论提供范围和内容，

……预示内容，包括什么，不包括什么，讨论多广，多深入，什么时候停止，

……为设计的压力测试提供参数。

我见过一些不使用“场景”的小组，他们建模的时候实际上就是在问题域内作随机的探险。负责人根本不知道下一个该问什么问题？经常都是凭直觉。一般说来，他们也不知道现在捕获的信息最后是否真正需要，就算知道也是凭直觉或者瞎蒙。

在一个寂静的深夜，几个真正专家级的 OO 设计师向我说了心里话：根本就没有一个有效的规则来指导漂亮的对象设计。其中一位说，“我们有时真的是在毫无目的地讨论”，另一位也相当赞同，“有时候我们就象没头的苍蝇一样到处乱撞，直到突然把一些好的对象给撞出来。”

使用场景也不能防止盲目的讨论和到处撞墙，只是可以为讨论提供内容和边界。我曾经见过有的小组不用场景，长时间在很大的建模范围内四处乱撞，失去控制。因此，使用场景和 use case 的第一个理由就是为建模的努

力提供一个范围。

使用场景的第二个理由是决定需求获取到什么程度算足够。

假设现在让你为一家货运公司建模。你建模的内容是什么？卡车的购买价值……实际价值……转售价值……车内颜色……快送单据的数目……旅途的数目和目的地？如果你想要把所有的内容都包括进来，那你将永远无法结束。除此之外，还有其它的问题：从何处开始，何时结束，包括什么，不包括什么，需要多少细节？

场景可以回答关于内容的问题，答案是：你需要足以回答场景中所有问题的信息。在结构化模型或完全的对象模型中，如果用一个方框来对应一个场景。然后在浏览场景的过程中将涉及到的元素（译者注：这里的元素，是比如在 OO 的类或对象）涂成红色，会发现最后所有的元素都按某种顺序连接起来了（不会有遗漏的元素）。如果对所有的场景都执行这个操作，最后所有的元素都会被涂成红色。在建模过程中，讨论会经常离题，用这种方法可以保证针对当前的需要制定一个边界，这样不会跑题太远。

最后，场景还提供了压力测试的参数以保证质量。

怎样比较两个模型的优劣？“和业务相符”，这是必要的，但肯定不够，可能有很多模型都可以“和业务相符”，怎样来度量这些模型的质量呢？

软件变更的成本很高，另外，变更越靠后，变更的成本也就越高。（变更成本曲线）。软件设计的一个目的就是将变更隔离开来，每次变更只需改变一个模块。这并不是什么时候都能做到的，正如 Kent Beck 所说，“如果你可以只改变一个类，那软件的质量将得到显著的提高”。和软件相比，业务模型的成本函数并不是很清楚，但将变更影响的范围降低到最低肯定是应该的。

为了测试变更曲线，需要参数，而场景可以提供这些参数。如果用户想要“something-or-the-other”，怎么办？模型中到底需要多少元素？像 CRC（class-responsibility-collaborator）这样的技术鼓励用户在现场快速得到场景，揭示可能的未来变更。最后，检查这些可能的变更，检查需要为之改动多少元素。对于两个都“和业务相符”的模型，哪个模型受场景影响的点少一些，哪个模型就要好一些。

其它很多地方也可以找到压力测试的参数。例如相关产品的结构，变更时是不是可以只改一点就可以了？在评价模型时，这些参数都应该用上。

场景还可以用在很多地方，例如：同用户一起检查需求，为系统提供功能测试的用例。以上所说，就是在建模活动中采用场景的三个理由。

高焕堂答疑录（一）

查看评论

整理自高焕堂先生在 UMLChina 的答疑板，网址：http://www.umlchina.com/expert/misoo.htm，截止至 2002 年 10 月

akaji: Sir,what is the mapping between the functional module and object or class?一般来说，我们开发的系统整体上看仍是按照功能模块划分的，我想知道面向功能的模块划分和对象模型、OO 的思想有什么差距？

Tom Kao: Objects 是元素, module 是组合物, 就像 H2O 是水, 而 H 和 O 是分子, 如何从水 mapping 出 H & O？先有 H & O 而后有水 ！module is what user want to buy. objects make up modules.

slovenboy: 面向对象分析的一般过程是什么样子的？？

Tom Kao: 一般而言, 先厘清 Domain 里的 concepts, 然后从 concepts 选出 key concepts 成为 components, 再把流程的 tasks 分派给适当的 components.

hotant: 你好，请问在做 OOAD 的过程中，在一个系统的开发中根据具体情况的不同在一个系统内使用多种设计模式是否可行？选择什么样的设计模式是否与语言实现完全无关？谢谢

Tom Kao: 一个 project 可使用不同的 design patterns. 一般"design" patterns 与语言是无关的. 可否针对前一项举个您的实际例子呢?

Li jiayu: 高先生，您好！我有几个问题想向您请教。UML 既然已经成为一种标准的东西，如果要对 UML 进行研究，有哪些方面还有值得研究的潜力？请问您主要研究 UML 的哪些方面？如果要研究 UML 建模的支持工具，您认为哪些方面是有研究价值的？谢谢！

Tom Kao: 我主要是研究设计思维, 而非 UML 就如同研究如何写唐诗宋词 而非研究中文语法

TamRhui: "Robustness analysis"主要是做什么工作？从书面上理解，是“健壮性分析“，在 Ivar Jacobson 的“Object-Oriented Software Engineering”书中介绍，是分析阶段的其中一个子阶段，根据 requirements model 进行 Robustness analysis，输出 Analysis model,但是，在这个分析中，我们主要针对什么进行，如何进行？请指教。

Tom Kao: robustness 分析是在寻找 business workers 由它们合作完成 business use cases, 再视这些为 IS 的 candidate actors, 就能顺利导出 IS 的 use cases 了.

KICK: 我现在正在搞一个项目的设计，那么搞到概要设计到详细设计的时候很头痛，因为，我突然发现自己不知道要搞到什么地步才可以停下来。请问项目的大小、开发时间、和建模的程度是否会因为某一因素的限制改变其他因素，如果改变的话（我认为是无法避免的）改变到什么程度是不可容忍的。因为这个是我用 OO 做的第一个设计。项目不大，用 UML 建模。十分感谢

Tom Kao: 您是采取 Waterfall 还是 Iterative approach? 如果是 Iterative, 在从 conceptual level 到 specification level 的设计过程中, programmers 由没有参与呢? 如何 feedback programmer's idea back to designers 呢?

sunshine: 我开始用 COM/DCOM 开发三层结构的应用系统，想请教几个问题，1.中间层如何用 COM 封装对数据库的访问？2.数据库中的表在 OOA 中为一些 Class/Object，这些 class 如何和数据库中的表交互？

Tom Kao: 中间层的 object 不一定要成为 COM component, 一般把 sql statement 封装于 stateless 的 object, 其交给 COM 即可, 把 business rules 封装于 stateful 或 stateless 内, 不一定 要给 COM 因为它不需要 sql and join transactions.

westwf: 请问软件工程中是如何估算软件工程师在一个项目中的贡献和产生的效益的,因为最终的代码运行在客户的机器上，而只凭几页文档资料是很难说明什么的;代码和文档资料很可能不一致!!!

Tom Kao: 这是软件比较不易于工程化的原因, 如果 Software Engineering 含括分析与应变的话,效益是 development team 所有,而非工程师专有,包括协助 project 的所有 stakeholder 都有贡献,此为整体的贡献,想一想一支篮球队得到冠军时, 如何计算出每一位球员的贡献呢? Software Engineering(SE)一辞因人而异, 而且数学难以计算整体和谐的个别价值,SE 本身有许多盲点,也有其极限!

zlqf: 请问 Delphi 下的 Midas 的三层结构的开发和所说的 OOA 方式有什么异同点？如果 ERP 的产品用 OOA 的方式来开发，其成本是否很高？另外 ooa 分析出来的对象最终和用户界面如何整合，是否有这方面的范例？

Tom Kao: Midas 已经是 4 年前的系统了,并非标准的 3-tier 架构, OOA 所分析出来的 objects 不易落实于 Midas 环境, 建议您采用 Delphi 公司新的 EJB-based Application Server 就能发挥 OOA 带来软件弹性的益处.ERP 使用 OOA 的开发成本会增高因为采用新的 PM(project management)人员需要训练成本. 但在 Deployment 及营运成本则能大幅降低. 至于最后的问题可否再叙述清楚些?

zlqf: 谢谢高先生指点,我最后一个问题是想问:对于大数据量要返回到用户界面在 ooa 里处理的技巧性.

Tom Kao: OOA 是要厘清 Domain 的 objects, 及 object 之间的 relationship, 例如 OOA 关心的是一个鸟巢里含有哪些鸟蛋.然而在 Programming 时因使用 relational DB, 其 query 时是以 Set of objects 为目标, 而非以 object 为目标,所以 programmer 每次 query 时是要找出 a set of 鸟巢, 及 a set of 鸟蛋, 让 user 挑选, 但 OOA 层次并无此 Set 思维. 从 OOA 的 instance 思维到 programming 的 table 思维之转换是 OOD 的主要工作, 解决方案是: 您可透过 stateless object 去 query 一个 set, 如 RecordSet or .Net 的 DataSet, 直接送往 client side, 在 mid tier 不用 object 去留下 set 或 object 的内容, 只有在 update, delete , insert 时才用到 mid-tier 的 object 去处理 object 或 record 的 state.

illfly: 你在做 ood 的时候，遵循那些原则？在整个开发的过程中，你又遵循那些原则？另外，你能推荐一些关于这方面的文章吗？

Tom Kao: 请问您, 您指的是 ood 还是 ooa&ood? 请问您, 您指的整个开发程序是指整个 ood 程序还是 the entire project development life cycle? 我想, 首要的原则是: architecture-first. 例如是 real-time system? 2-tiered client-server system?还是 n-tier enterprise / web service system ? 不同的 architecture 其开发的 detailed process and principle 是有很多差异的。例如 开发 n-tier enterprise system 时, 有个重要原则就是: 先 design mid-tier 里的 objects, , 之后才分析 user 的流程细节.

illfly: 高先生，您好首先，谢谢你的回答。我的 OOD 的意思是面向对象设计的原则，我想您提到的体系结构优先原则，应该是对于软件设计阶段而言的。我的整个开发原则，是指 Entire project life cycle。您认为设计多层结构的系统要从中间层开始，为什么不是从其它层，比如数据层（最低层）开始？另外，对于不同的体系结构，他们的过程和原则不同在何处？如果您太忙的话，能否推荐一些文章？

Tom Kao: N-tier architecture 的目的是要提升系统的弹性,以强化应变能力. 基于这个前提, software developer 就得细心分析您的系统的 change 因素以及来源为何.例如您认为 DB 是稳定还是善变呢? 如果您的答案是”善变”, 您就不能从 DB 层开始了. 因为房子的地基越稳固越好. 依据 layered architecture 原则, 善变者应该属于上层, 所以 DB 若善变就不应该视为底层, 此为 View 的问题. 在 web service 环境下 DB 是极端善变的.

wangxh: 高先生，请教两个有关 Actor 的问题。我有一个用例是“警告提示”，这个用例是由内部事件触发的，如“发生温度的异常现象”，我在选择 Actor 的时候选择什么呢，是选择“定时器”，选择“系统时钟”？我找了一些解释，用“定时器”作 Actor 的说明中并没有说事件触发的这种情况，另外也没有找到任何资料说“系统时钟”可以作 Actor 的，究竟应该如何理解？第二，如果用例分成主用例和从用例，Actor 连接在哪种用例上好呢，我到网上查了一下，怎样的画法都有。比如，我的“警告提示”主用例 include 了“显示处理”的从用例，我有一个

Actor“显示硬件单元”，是连在“显示处理”从用例上呢，还是连在“警告提示”主用例上，还是其他？请您帮我解答一下，谢谢

Tom Kao: 您的 system 内有细分为更小的 components 吗? System 内的 Worker 会扮演小 component 的 actor 的角色. ‘计时器’或’clock’能 trigger 某些 component 的工作流程. 一旦您使用 object-oriented thinking 去细分出小 components, 会发现’计时器’trigger 某个 component 的 use case ---- ‘警告提示”此 Component 可能继续 trigger 另一个 component 的 use case ---- ‘显示处理”. 这些内部小 use cases 串起来就是 system 的内部 work flow, 而非 system 的 external behavior---- 即 use case. 记得 use case thinking 必须搭配 object thinking 才能解决 OO 的设计问题.

pherery: 我想问您一下，有关对象继承的使用，一直有两种观点，一个是 Code reuse，一个是 Behavior Subtype。诚然这两者是截然不同的，但我始终不太明白，这样的区分，对于实际的应用开发来说，有什么意义呢？如果可以，您能举一个鲜明的例子说一下这样做的意义么？

Tom Kao: Behavior Subtype 为介面继承, 注重于 object 的抽换性, 即支持 polymorphism, 此是为了 software 的弹性, 以便应付多变的 user requirements.这是近来对继承的主流思维, code reuse 可走 binary code reuse, 就是经由 interface 的 reuse, 因为经由继承的 source code reuse 会有大的副作用---- dependency 太高.

coolwizard: 高生，我现在从事基于 J2EE 的 WEB 开发，在这些系统中尤其是前端 JSP,Serlvet 的设计上比较难以把握，我尝试画的 Class Diagram 我觉得有些象部署图，因为 JSP 的显示可能比较复杂，不知道你觉得在 Web 系统的设计上该注意些什么

Tom Kao: 如果您使用像 IBM 或 BEA 公司的 Application Server(如 BEA 的 Web Logic system) 的话, Servlet 的内容就只剩下 GUI logic 而已, 因为 Business logic 移到 mid-tier 的 ap server 了. 此时 servlet 的 class diagram 里的 classes 是代表 html files 就是 pages, 包括 client page 及 server page. App Server 里的 class diagram 的 classes 就代表 business concepts 就是所谓的 business components.

coolwizard: 高生，我用的是 IBM WebSphere.我想问你的是在作 WEB 系统开发的时候怎样进行页面 UI 的设计，我觉得这一块要用 UML 表示出来比较棘手，因为 JSP 页面可能会很复杂。你可以看看图片共享中的（在线专家 Class 图、在线专家 use case 图），我总觉得我的设计有问题，有些杂乱。有人建议我用组包的方式划分结构，但是总是要有一个图来表示他们之间的关系不是吗？我就是采用三层结构！在在线专家 Class 图中我并没有明显划分他们是那一层的，我只是要说明他们之间的关系。只是给我的感觉这个图有点象部署图，我只想知道怎么避免这个问题，怎样改进它

Tom Kao: 您看过 Rational 公司的 Jim Conallen 所写的一本书 -----‘Building Web Applications with UML’吗? 也许它是个好得参考, 其中的 icon 都已经落实于 Rational Rose 2000e version 里了.

fox: 我感觉 uml 建模过程是使用文字描述和用例图描述企业流程,辅之以顺序图和合作图.但是这种模型不太直观,因为顺序图和合作图中没有表达业务数据的流动(虽然文字描述中有)，在 IDEF 分析方法,企业流程图要表达业务数据的流动,我以前使用 uml 进行 MIS 类系统的分析时,在业务分析阶段仍然使用业务流程图进行企业业务的描述，请高老师给与解答

Tom Kao: 所谓 business process 是有很多个 views, 有些人重视 activity flow, 就会抽象出 activities 而抽掉 data 及 worker 等, 成为 activity diagram. 有些人重视 data flow, 就会抽象出 data 而抽掉 activities 及 worker 等, 成为 Data Flow Diagram. 有些人重视 workers 之间的 message passing 及互助合作, 就会抽象出 objects 而把 data 及 activity 纳入 objects 内, 成为 object model. UML 是来自 OO 领域, 比较偏重 object model, 需要习惯于 object thinking, 一般人会觉得不太直观. 但是要将 MIS 系统落实到 N-tier 或 component-based 环境, 就必须将 activity flow & data flow 的 model 转为 object model, 因为 object model 的内涵(semantics)最丰富, 能提供充足的 information 给 software developers. UML2.0 已经吸收 IDEF, 强化了 business activity flow 的表达.因为 manager 喜欢 activity flow diagram, UML2.0 比较能抓到 manager’s view 了. 记得, use case 不是用来表达 business 内部的 work flow!

liuhongchao: 高先生，占用您一点宝贵的时间，我是刚毕业的计算机系的学生，想学习 UML 统一建模语言，我对它的重要性已经有了了解，但是不知道从哪里入手开始学习。我想请你推荐一本入门级的书籍，可不可以？？谢谢您！

Tom Kao: 我想, UML Distilled 一书是蛮好的入门书.

潘瑞辉: 我学 UML 有段时间了，我觉的用例图描述企业流程,辅之以顺序图和合作图应该来讲不是很困难（因为现在我做系统分析只是做到这一步而已），但是如何提炼出类图我觉得很不好处理，有没这方面的好书或建议提供一下，谢谢！

Tom Kao: Identifying classes 的途径有很多, 从 use case 的 description 去找是个方法, 但是副作用也很大. 也就是说, 最好不要由企业流程去找 classes, 而应该从 domain model 下手, 从 domain concepts 去找 classes 才是最佳的. 可参考 Charles Richter 的 Designing Flexible Object-Oriented systems With UML 一书. 还有 Peter Coad 的 Object Models 一书.

yinxm: 高先生，您好，我在三层设计的时候可以把用户界面和商业逻辑层分开，可是商业逻辑层和数据层的区分程度我总是把握不好，业务逻辑总要修改数据库吧，我想请教一下，这一部分有没有好的方法？谢谢您！

Tom Kao: 这与您使用的 platform 有很大的关系，如果您使用 EJB，只要 follow 其 framework 就行了---- 将 business logic 落实到 session bean 而 data logic 落实到 entity bean. 如果您采取 Microsoft .Net，就会受到其 DataSet 用法的影响.每个 platform 都有其 assumption 和框架,但是都会支持相同的目标---- 让 DataBase 或 Data Source 能激烈变化，但不至于激烈影响到 mid-tier 及 representation tier. 这一部分没有一定途径，需要理解 platform 的特性,心中追求上述目标，加上一些练习就会有把握了.

vans: 商业逻辑层和数据层的区分: 我的设计是在商业逻辑层和数据层之间加上一对象访问层，它负责动态产生数据访问语句，逻辑层对数据的访问类似于类对其属性的访问，这样逻辑层和数据层的耦合度就降低了。 当然，这样做的前提是需求的 OOAD 要做好！ 不知道，高 Sir 对这种方法有什么意见，请多指教。

Tom Kao: 加上物件访问层是合理的，但访问层的物件应该是 stateless，其 db connection 的运用率才会提高，既然是 stateless 物件，就是 functional 思维的产物，与 OOAD 似乎是无关的. 除非 Business 物件与访问层的物件是一对一. 但是一对一并不是很好! 因此，加上物件访问层是合理的，但不会多加负担.

wlq: Sir,你好！ 请问数据库服务方面，中间层的业务对象中是怎么样来进行与数据层的交互的，对于 Persistent 是怎么看的，在业务对象与数据库之间加上这个思想设计你觉得如何，用来解决跟数据库的交互问题，以进行跟业务的隔离，这样和加上一个对象访问层，哪一个好些？

Tom Kao: 如果 Data Sources 是很单纯的 RDB，而且是集中式的 RDB，各种方法的效果大致相同. 如果 Data Sources 是多样化的，而且是分散式的，采取物件访问层方式应是较理想的. 稳定的介面是重要的考量.

steven: 高先生，您好！我觉得 UI 建模好像和 OO 设计的其它方面有很大不同，以下几个问题让我觉得疑惑：1）我应该把每个 window 作为对象体现在设计模型中吗？这样好像会使得设计模型非常易碎。2）如果把 Window 作为对象来建模的话，好像很难决定它的 behavior，因为它接收的大多数消息来自用户的操作而不是来自其它内部对象。3）如果一个 window 中包含其它复杂的界面元素（例如 TreeView 等），我是否应当将这些界面元素也作为对象抽取出来？这样好像会变得很琐碎。4）普通的业务对象和控制对象都容易写出单元测试代码，但是 UI 对象怎样进行单元测试呢？window 对象所响应的请求（比如鼠标点击）无法用代码激发呀！5）如果我使用 RAD 工具进行开发，情况有何不同？非常感谢您！

Tom Kao: 过去在 2-tier 时代, windows 内涵有流程的行为控制, UI 会蛮复杂的, 很耗时间, 因而想去建立 model 寻求 reuse. 然而随着 HTML, XML, Web service 的潮流, UI 的 model 对象是 Page 或 XML Document 等,而不是 Form 及 buttons 等了. 此外可为 UI 建构出 UML statechart 及 use case model.叙述系统 UI 的整体外部行为, 可参考物件导向杂志第 12 期 page. 82.

wang: 高先生，我一直留意着您对 N-tier 架构及其层次划分问题的回答，细细品读之后还是不得其要领。我想请教高先生，对于 N-tier 架构的观念的掌握，应如何学习，可否为晚生介绍一些相关书籍，谢谢！

Tom Kao: N-tier 架构设计最主要的考量是 change 的因应策略. 也就是将”应变的策略”写入 software 里, 而不是把所有可能的变化皆写入 software 里. 听起来似乎很玄, 其实道理并不难, 只不过是中国哲理里的”虚实相依”之道理而已. 所谓相依就是洋人的 dependency. 谈到 dependency 就牵涉到 interface 概念了. 如果 object A uses an interface of object B, then A depends on B. 这意味着如果 B 是善变的话, A 就不得安宁. 此架构就是 bad! 例如, 如果 database 是善变的, 在 ap server 上的 business objects 若直接呼叫 DB 的 API, 就是 bad design. 因为 business objects 需要稳定, 它是 software 系统的主干, 需要稳定, 系统才能稳定发展. Business objects 和 database 皆是实的, 两者之间必须有一层虚的, 就是 persistent objects. 只要您能 with change in mind, 就能掌握 N-tier 的设计了, 就像古代的大禹治水---- 顺应 change , 而鲧的治水 ---- 想控制 change, 两者得出不一样的设计. 易经的乾卦也告诉我们: 龙是善变而高能量, 只要有艺术之心就能 leverage 它的潜能. 所以软件工程和软件艺术的均衡,是 N-tier 设计的基本心理素养. 关于书籍---- John Cheesman 的 UML Components ISBN:0-201-70851-5 ---- Heineman 的 Component-based Software Engineering ISBN: 0-201-70485-4

learningUML: 用 OO 做系统分析与设计有一些困惑，希望能够得到您的指点。困惑之一： 在设计阶段应该主要设计出什么东西才能进行下一步编码？ 困惑之二： OOADP 时以什么为任务分配单元？比如：在结构化方法中可以以功能模块为任务分配单元

Tom Kao: 这视您的 process 而定了, 若采用 RUP 的话, 应该设计出 design classes. 不过, 在 RUP 里如果您的目标是 2-tier 或是 sevlet/ASP-based 的 2.5-tier 架构, 这些 design classes 并不复杂. 但是如果您的目的是发展 3-tier 或 component-based 系统时, 您必须将 control objects 里的 control logic 下移到适当的 entity objects 里, 并且设计这些 entity objects 之间的关系, 并且考虑到 transaction, stateless, stateful objects 等 distributed computing 的问题. 还是建议您采取 iterative approach, 逐步增加的设计复杂度, 只要 programmer 觉得能编码就尽快去编码, 不必太重视 D&P 的分界, 因为设计越细腻, 常错误越多! 多 test 才是良策.

weblane: 在表示层与商业逻辑层之间添加中间层，能有效的分离表示层上的页面代码和程序代码，方便测试与维护。请问高先生对此怎么看？有这方面的资料吗？

Tom Kao: 是的, 通常会添加一个 façade object, 避免 business objects 的 interface 被 client 层抓住, 创造出 business objects 的弹性空间, 此 façade object 相当于 RUP 里的 control object ---- 此 control object 内部的 control logic 大部分都已经 assign 给 business objects 了,成为虚化的 object. 就满足 façade pattern 的精神了. 这只是 façade pattern 的小小应用而已. 可参考 Design Pattern 一书.

zhou: 我想请问一下如何对一个 n-tier enterprise project 的代码量进行估算，一个程序员大概的每日编码量，以及有什么工具可以用来计算一个完成项目的代码行数，谢谢

Tom Kao: 在 n-tier 上, 除了 GUI 外, 我们很少计算 SLOC(source lines of code).但是也有专家常使用 SLOC 计算 productivity. 例如您可参考 Jeffrey S. Poulin 的文章 ----- The book: “Component-Based Software Engineering ---- Putting the Pieces Together” edited by George Heineman ISBN 0-201-70485-4 的 pp.435-452.

Carroll: 高 sir，在您给潘瑞辉的回答中有这样一段话：“Identifying classes 的途径有很多, 从 use case 的 description 去找是个方法, 但是副作用也很大. 也就是说, 最好不要由企业流程去找 classes, 而应该从 domain model 下手, 从 domain concepts 去找 classes 才是最佳的.”我现在对 domain model 感觉很模糊，domain model 到底指的是什么模型呢？是不是指系统用例？

Tom Kao: Use case 呈现特定 user 的观点(view), 一个系统是一个整体(whole)Use case 是局部(part), 一个整体不等于所有局部的总合. 如果一个系统是 tree,而 use cases 是 leaves, 当我们知道全部 leaf 的 information 时, 并不表示我们得到整棵树的 information. 至少从树叶无法得知树干的 info ！所以 classes 最好来自系统的整体观(the whole view). 对此整体观的 model 就 domain model. 在 Jacobson 的想法中, domain models 偏向 business use cases (企业的外观)及其 ideal model(企业的内观), 再从 ideal model 导出 system use cases. 然而,在另一位专家 ---- Peter Coad 的想法中, 从企业的重要事件(event)着手, 可得出较稳定的 domain models, 再反过来支援 use cases. 就像树干支援树叶一般. 如此, 树干能支援未来的 use cases. 如果您的 classes 只是要支援现在的 use cases, classes 是手段而 use cases 才是目的, 那就不必讲究 domain model 了, 如果您认为 use cases 只是用来提炼 classes, 而 use cases 是手段而不是目的, 那就会很讲究 classes 的设计了. 请参阅 Peter Coad 的”Object Models: Strategies, patterns, and applications”一书.

edward: how to measure oo program,Can you recommend some books,articles or web sites about this.thank you.

Tom Kao: The book: "Component-Based Software Engineering ---- Putting the Pieces Together" edited by George Heineman ISBN 0-201-70485-4 is an excellent reference. Especially, Part VI (pp.431-552) about measurement and metrics for software components.

goldarcher: 高先生，你好！我想请教一下，对于中间件系统，怎么通过 USECASE 来捕获用户需求呢？对于这种系统，ACTOR 常常都不是人，并且系统的范围也很不好划定。这种情况下是否不适合用 USECASE 来分析需求呢，请您给些建议和指导，谢谢！

Tom Kao: 只要是它的 client, 来用它时必然有其 goal 或 expectation. 所以使用 uc 捕获功能需求是适当的. 这种系统通常采取 iterative approach. 需求分析人员会分辨轻重缓急, 挑出重要的 uc, 重要的 process, 重要的 components, 得出系统的 architecture, 再逐渐增加 uc, 流程细节及 components. Uc 和 系统范围都是可变的. 虽然这是 use case-driven, 但却是 architecture-centered.

yanqi: 高先生，你好。我想请问在 ROSE 中如何添加自己设计的一些 diagram（原有的 diagram 之外的），就我所知，ROSE 支持自己设计 stereotype 和菜单选项。 如果可以加入自己设计的别种 diagram，请给出方法或相关链接，谢谢！如果不可以，请问自己开发 ROSE 插件能行吗？

Tom Kao: 依据 rose 的 stereotype 还是可以发展出自己的 diagram 加入 Rose 环境成为 model 的一个新 view. 但是受到其 meta model 的限制, 您必须找到结构相同的 notation 去附加 stereotype. 这方面我们的经验不多, 新加的 diagram 都只限于既有 diagram 的 specialization 而已.

kingfanqi: 高先生 你好，请教一个问题，在设计类时，如果三个类之间存在多对多的引用关系，并且存在先后关系，第一个类引用第二个类，第二个类引用第三个类，那在设计三个类时，可不可以反向引用，即第二个类中可以引用第一个类，第三个类可以引用第二个类？

Tom Kao: 如果您的是 real-time 系统, 双向引用是常见的, 因为 real-time system 常会 share object state, 所以会共用 instance. 如果您是 database-based 的 business system 就可以尽量避免双向的引用, 因为 object state 是 persist 在 DB 里, 并不需要 share instance. 每一个 use case 执行时, 都各自诞生其所需要的 instance. 这个有利的条件, 让我们有更多的 design choice. 我们就能将之改变成为单向的 client-server 相依(dependency)关系, 使得系统能呈现 layered & tiered 结构, 这样的系统具有理想的稳定性与弹性. 您可参阅 Ivar Jacobson 的 Software Reuse 一书的 Layered Architecture 部分.

linqier: 我想请问一个问题：我们在编码的时候，类之间有 3 种明显的关系，聚合、继承和关联（消息），我在用 Rose 进行设计的时候，也发现这 3 种关系的标识，但是，当我转换代码的时候，发现关联（Association）和聚合，二者没有任何区别，我不明白是怎么回事？

Tom Kao: Aggregation 与 Association 的差异在其行为(behavioral)和涵义(semantics)面向. 在涵义上 Aggregation 具有包含之意. 在行为上, Aggregation 往往其 lifetime 是一致的, 也就是聚合类必须负责 create 被聚合类的 instance. 您只要观察其 instance 的诞生时刻就知道了.

wlq: 高先生您好一个典型 WMS 系统的描述 1.生管部门下达订单 2.订单经过几个制程，这几个制程是相似的： a.本制程内的派工 b.手工/扫描入库 3.产品出货如何用 UseCase 图分析,特别是，各个制程由不同的人分析，最后如何把他们画的图综合在一起

Tom Kao: 使用 use case 图表示制程时, 必须注意到 use case 并不适合表达生产线上的工作流程. 1) use case 只用来表达下订单者与此生产线的互动对话的过程(dialogue process). 2) 该生产线有许多的 workers, 您可以为每一个 worker 建立数个 use cases 以表示该 worker 与此 WMS 系统的对话流程. 3)整个制程就是这些 workers 的 collaboration, 应该以 sequence 图表示, 而不是完全由 use case 图来表示. 以上我所谈的是 use case 表达制程时, 很常见的错误. 记得 use case 是 actor 与系统的对话流程, 而不是系统内部的工作流程,所以 use cases 之整合是透过 actor 整合的, 而不是单就 use cases 之间来作整合. 在过去的 function-based 时代只谈流程, 但当今的 component-based 不能只谈流程, 而必须以 component 为中心, 才有意义.

qgf: 高先生您好！我是个 UML 新手，有个浅显的问题请教，关于用例建模中用例的大小和事件流的关系：我们把目前着手的系统分成 11 个包，对于，例如“性能管理”包中的用例“管理性能任务”，它的事件流中，创建性能任务、删除、修改等等 10 项操作，彼此独立，放在 1 个用例的事件流中该如何描述对以后的 OOAD 有无影响？若分成 10 个用例似乎太多了（其它的包也有同样问题）。

Tom Kao: 因为他们独立而且非例外状况, 所以一般都摆在同一个 use case 的 normal course 里, 各个都是 normal flow 里的 subflows. 如此描述就可以了.

柴青锋: 请问高先生，当我们从 use case 中导出类图时应注意些什么

Tom Kao: 从 use case 导出 class diagram 时必须注意 use cases 的不完整性, 也就是您必须掌握所有可能的 use cases 中的 80%才能得出稳定的 class diagram. 还有 use case 必须是对的,但是 use case 的设计偏偏常是错误的开始, 因为 use case 很容易学但很容易误用.

lucy: 我是一个 uml 新手，现在正在做远程教育系统的 uml 建模。 在划分用例的时候，比如用例课程管理和它的子用例添加课程、修改课程、删除课程之间，是用 extend，还是用 include??? 我看到的一些例子，用例和子用例之间有的用 extend，有的用 include。 在 uml1.1 中用例间有 extend 和 use 关系，而 uml1.3 中出现了 include、refine、derive、extend 等关系，请问它们之间有什么联系和区别。我看到一本书上说 1.3 上是用 include 代替了 use，这种说法确切吗？

Tom Kao: 请您留意 ---- ‘课程管理’应该是一个 subsystem, 而不是一个 use case. 一旦您能分辨 subsystem(或称 package)和 use case 的区别时,您的问题就可解决了. Use case 是有头有尾的一连串事件活动,但是课程管理的活动并非一连串, 它是一个 function area 而不是特定 actor 的 expectation.

Kent LEE: How are you? I am learning the O-O Modeling. I am confused about visibility of attributes especially when I am modeling a composer writes composition. The composer's attributes, like birthOfDate and nationality, are protect or public? How to determine what visibility markers I should use when I’m modeling? Is it according to its real visibility(I mean we can actually see with our eyes) or the scope of the attributes like in programmers? Thanks.Also.what is the difference between composition and aggregation?

Tom Kao: Visibility 通常反应出设计的 considerations ---- 主要是 replace 的问题。Component 如果要达到高度的 replacability, 就会采取黑箱(black-box)的设计, 既然是 black-box 那该 component 的 attributes 就必须是 private. 例如 PC 的 mother board 上有个 CPU, CPU 里有 control unit 和 adder 两个 components, 一般此 CPU 是采取黑箱设计, 所以 CPU 里关于 control unit 和 adder 的 attributes 都是 private ---- 也就是 CPU 的 client 是看不到的. 如此的设计, CPU 就是一个 easy-to replace 的 component. 当然您将 CPU 设计为白箱---client 可直接跟 adder 沟通, 但 CPU 的抽换性就降低了。 Aggregation 是 composition 的一种。

阿肖: 您好我目前在写硕士论文，主要内容是探讨用 UML 进行数据仓库建模。我的考虑出于以下：我手头有一本《面向对象数据仓库》，我觉得能使用面向对象方法应该就能用 UML 来表达。现在我的内容基本写好了，UML 方面主要做的就是定义了相关的版型，用类图描画出了星型架构。但是我还希望能得到一些理论上的支持，尤其是面向对象领域和数据仓库领域的专家在这方面的论述。我在网上找到的不多，不知道高先生在这方面能不能做点推荐，谢谢。

Tom Kao: 我手上关于面向物件资料仓库的理论部分并不多。我们也一直把它视为 OOAD 的应用, 也就是透过 OOAD 程序而分析 business domain & decision requirements 而建立 UML 的 class diagram, 然后导出 star schema。使用 OOAD 及 object model 的理由就在于 easy-to-change, 因为 business decision environment 是多变的, 透过 iterative

approach 能达到此效果。

jenny: 面向对象的思想在 C++,C#,Java 上各有什么优缺点，不同？

Tom Kao: C++比较偏向传统 OO 思维---- 也就是 class-based (即 program to class)思维, C#和 Java 则偏向 Component-based(即 program to interface)思维, 在 Web 环境里后者比较有弹性,易于分工与快速整合。C#与 Java 在 language level 是一致的。

xiuyanfu: 我刚刚开始学习使用 UML，现在我需要对一个软件画用例图和顺序图。这个软件是多线程的实时程序。我们开发的代理软件 gameproxy 的功能就是使我们自己的服务器能够实现 VLAN 中游戏的互通，保证游戏的联机对战。gameproxy 把本地 server 地址报告给中心服务器；而中心服务器将向各 VLAN 中 gameproxy 发送的信息列表；各 gameproxy 再把中心服务器中的列表下载到本地，并保存列表，当客户发出请求时给予响应。 软件开始运行时，首先根据 Config 的内容启动 FetchTask 和 InventorTask 两个线程，并将 ProxyHandler 注册到 Reactor。FetchTask 的任务是每隔 30 秒向中心服务器发一次请求，然后接收中心服务器的响应（即返回中心服务器统计的服务列表），再将服务列表注册到本地 ServerVector 数据库中。InventorTask 的任务是查找本地 Server，并发送给 FetchTask，激活 FetchTask。ProxyHandler 的任务是接收客户的请求，查询 ServerVector，将返回的服务列表发送给客户。 程序大体就是这样，但我怎么也想不出来该怎样画它的顺序图，它需要与本地服务器和中央服务器进行通信，还要接受客户的请求并给予响应，但顺序图中该如何表示呢？不知我是否表达清楚？ 我真的很着急，请您在百忙之中尽快给我答复好吗？谢谢您!!! 您可以将您的答复发到我的信箱。

Tom Kao: 我的建议是----1)先以 Component diagram 表达出 中心 server, 本地 server 和 client 之间的 dependency relationship 凸显出整体结构。2)建立 a hierarchy of use case diagrams. 因为在 FetchTask 里本地 server 是中心 server 的 actor, 就导出中心 server 的 use case diagram 了.同理, 在 InventorTask 里 client 是本地 server 的 actor, 就导出本地 server 的 use case diagram 了. 同理客户是 client 的 actor, 就导出 client 的 use case diagram 了. 例如, client 的 use case 之一是 ---- UC:客户 login, 此 use case 会去 trigger client 的某些 use cases, 而 client 的 use case 又会 trigger sever 的一些 use cases.3)针对各 use case 而去绘制其 sequence diagram. 所以应该有 a hierarchy of sequence diagrams 而不是集中在一个 sequence diagram.4)至于 use cases 之间的 trigger 关系并不表示在这些 sequence diagrams 里, 因为这些 sequence diagrams 是表达 use case 内的状况, 而不是 use case 之间的状况, 其是不同 level 的事.5)根据这些 sequence diagrams, 就能分工去编码了, 然后再根据需要 trigger 等关系而将之组合成整体系统。记得, 擅用 use case 能对复杂的 sequence diagrams 作理想的 partition 而简化系统的设计与编码。

boyi_cj: 高老师您好,我是 uml 的初学者,在许多 uml 的书籍中,经常提到 uml 用于非软件工程领域,可是这方面的资料太少,不过我觉得 uml 的图形化建模的方式,很值得其他专业在构建应用模型方面借鉴引用,我现在在写关于物流配送方面的论文,设计到许多繁杂的数学模型,不知道 uml 能否给我带来更广阔的思路?盼您指点

Tom Kao: 从物流配送的 domain 里找出 domain concepts, 视为 objects, 然后尝试把数学模型里的 functions 指定给不同的 objects, 例如预测 POS 的销售量, 此预测的数学 function 可成为’POS’ class 里的 method. 原来 POS 是被动地被预测, 转变为 POS 自己会预测, 则物流配送就能思考为物品, 车辆, 企业组织单位等 objects 的沟通合作, 协同执行复杂的数学模型. 不要把物品视为 data 而已, 请把物品视为 object, 其有行为能力.

大宝: PLEASE HELP ME ,A TEST QUESTION,顾客将钱投入自动售票机，机器显示可以买的票（低于投入的钱）。客户选择要买的票，如果低于投入的钱，那么出票找钱，如果钱不够，那么等待客户投入钱。画出 use case 图

Tom Kao: 就画出一个椭圆, 中间填上’售票’, 就行了. So easy, 但也许我误解您的意思了.

Lean: 你好，我有个关于 Use case 图的问题：该系统是对很多硬件设备进行管理的，有三方：第一方为硬件设备，中间方为 IP-SV（主要是负责数模转换的），第三方是中央信息处理。现在对中央处理画 Use case 图，并且文档主要是介绍硬件设备过来的处理流程。请问我是以各硬件设备作为一个 Actor，还是以 IP-SV 作为一个 Actor?

Tom Kao: 这要看谁 trigger 中央信息处理的, 如果是由 hardware 直接发出 event 给中央系统,actor 就是 hardware. 如果是 IP-SV 系统 trigger 的, 则 IP-SV 就应该是 actor.

xzhqzz: 我想请教一下。我正在做一个发短信的系统，采用三层模式设计，客户端把短信提交到数据库中，然后中间层从数据库中读取数据，发给短信中心，但是客户端提交给 10000 个左右手机的时候，会出现丢包现象，请问如何处理这个问题？ 对于远程分布式的大批量数据提交，如何处理？？

Tom Kao: 在企业组织上是 3-tier, 但是软件设计上并非 3-tier. 例如在客户端您是采用 2-tier. 此客户端与组织的中间层之间是透过 Database 进行资料转档, 而不是软件系统透过 XML 直接沟通, 所以我认为此并非 3-tier 软件系统. 在软件系统上分散式大量资料传输已不是问题, 但依据您所述还不足以判断出问题点.

DzWang: 我刚刚用 Use Case 作需求分析，现有好多不解之处，请您多多指教。一、我认为 Use Case 在获取的需求情况基础上进行分析的工具，但 Use Case 的抽取原则是什么呢？二、Use Case 的粒度多大合适？管理客户档案（包括增、删、改、查 4 项操作）和分成新增客户档案等 4 个 Use Case 哪个更合适？新增操作也是有一定流程的。三、Use Case 事件流描述是否有必要写权限判定和输入数据的完整性检查？四、Use Case 事件流描述中是否有必要进行画面流程的设计？

Tom Kao: 所谓需求就是 What users want. UC 就要抽取 what users want to buy. 关于 UC 的问题大多来自于 view point 上，只要您能换个立场看看，UC 的问题就能迎刃而解了. 想象您是 user，您会列出您想买哪些 service，逐一列出并给予名称，就得出 use case diagram 了.别忘了，天文学家哥白尼换个立场---- 想象自己坐在太阳 ---- 而得到不同的认知. 请您想象自己是 users，您会把[增、删、改、查]视为一体吗？如果会，就合成一个 UC. 如此就很自然而决定粒度大小了. 软件是工程与艺术的结晶，艺术创作取决于思维角度. UC 是 model，不要追求完整性，而在于抽象出重点，细节流程可交给 programmer 去处理即行了.

liu090: 我要在 rose 的类中加一个属性，且是 integer 型的数组， 好象不可以吗？如何做？？？？？？？？？ 另外 uml 是否会束缚人的创造力？？？？我很困惑？？？

Tom Kao: 在 rose 里可以加上 collection 型态的 attribute. 不同语言特性会塑造不同的思维方式，常会束缚人的创造力，但 UML 的 visualization 效果却有助于创意. 同时因为 UML 的标准化，促进沟通激荡，也有助于 think out of box，对创意大有帮助. 像音乐的五线谱语言对音乐创意也是有帮助的，再如写诗，好的诗不是来自语言的技巧，而是来自诗人对一切事务的看法和心灵. 当您愈熟悉软件设计的美感时,就愈有能力超脱 tools(如 rose 和 UML)的限制.

Sunche: Dear 高老师.我有个问题要请教您,当从 business concept 萃取来的 business rule 如何表现在 business object 中,尤其是当 business rule 中有很多的 check 条件且 check 的条件是善变时.如:C 制造光碟类别有一个 interface 叫做取出制程(float mm , CExptool machine)...它会传回 一个制程的类别(C 制程)制程往往和点几制程有关,和制造机台也有关..现在有可能和生产时程有关,..所以我改变 interface 变成 C 制程 取出制程(float mm , CExptool machine , Date duedate)..虽然是同一个 interface 却有这不同的参数,需要 check.我心中觉得不够美,那一天有多个 check 条件我不是又要修改 interface?

Tom Kao: 您的目的似乎想让 procedure ----- 取出制程(para-1, para-2, … , para-n)的参数个数不变，是吗？如果 yes，有些可选择的方案. 因条件新增致新增一些流程片段，此片段您可以新增到 business class 内，其势必更改 business class 的内容. 在此情形下，para-n 等只是纯 data 而已，您就可以把这些 data items 摆入 collection 里，则变成为 ----- 取出制程(collection para_coll). 之后 business object 从 para_coll 里取出 data items，逐一处理之. 另一种途径是定义一个 super class ---- ’条件’,然后针对各条件而建立 subclasses ----例如 ‘生产时程’，它继承’条件’class. 流程的新增片段,各写在 subclass 里，则变成为 ----- 取出制程(collection of instance of '条件’)，之后 business object 会呼叫 subclasses 里的新增流程片段，business class 并不需要随着条件的异动而更改，这是 polymorphism 的应用.

sakurracn: 请问 1.针对商务应用活动，我们对活动图的 join 扩展构造型，包括《send》和《receive》.在 join 处，每次最多一个《send》发送信息，而允许多个《receive》接受信息。用 OCL 如何描述？2.UML/MOF/XMI 的关系。有很多关于 UML 在代理技术中的应用的文章，不仅提到 UML，还包括了 OMG 的其他产品。为什么，这同样是 UML 应用吗?

Tom Kao: 在 activity diagram 里的一个'join'element 符号是一个 instance, 此 element 能 link 到多少个《send》elements 和多少个《receive》elements 的限制(constraints), 这是 instance-level 的 link 关系, 会在 class-level 的 association 关系上定义, association 的 visibility 能限制 link 对象的个数。 在 activity diagram 里的一个' join' 符号是一个 instance, 其 class 存在于 UML metamodel 里, 而不是在 UML model 层级去撰写 OCL 叙述, 您可以定义一个新的 stereotype of 'join',并且设定其 visibility property 之值。 UML/MOF/XMI 的关系很简单, 您使用 UML 去建立应用系统的 models, 使用 MOF 去定义及管理这些 models 的 metadata, 利用 XMI 将 models 转成 XML DTDs 然后能在 internet 上传递。OMG 的其他产品大多是来支援 UML 的应用, 也有应用到 UML。

ken: OO 的分析方法,除了 UML,您是否推荐其他方法,UML 又有什么缺失?

Tom Kao: 因为 UML 只是一种'表达'方法, 但不是'分析'方法, 我想您指的是'表达方法'吧! 我认为表达方法的目的是要沟通(communication), 大家越一致, 逐渐改进, 集思广益, 就越像音乐里的五线谱及其音符般, 放之四海皆准, 就够棒了. 例如 UML 逐渐将 IDEF 的流程表达之优点涵括进来, 是它逐渐改进的现象. 请留意, 将'表达'方法与'分析'方法做细腻的分辨, 就会更了解 UML.

Linkspeed: Most OO books talks about the concepts of class and object. Is there any definition in mathematics, such as formal language? In my impression, the set and type theories are about all the computing languages, not just OO. There is another book that about object theory in which I could not understand one single page. Can you tell me what relationship between such theories and the class and object in practical language? How can I map the concepts in theory and the real practices together? What's the difference between an OO language and Functional language or Imperative language at those theory level? They must have some differences, otherwise, there would not have a theory call "object theory" Many many thanks Linkspeed

Tom Kao: 我个人常从 cognitive 观点来诠释 OO concepts, 我想我的 mathematics 或 formal language 等知识, 并不足以回答您的问题. 谨此

xinxuesheng: 请问：一个完整的 Rose 设计文档至少包括哪些图？后一张图和前一张图的关系如何？哪些图是自动导出的？

Tom Kao: 设计含有艺术成分，其基本原则是: 用最少的图而能清晰表达出您心中的完整设计,是最棒的。因为有艺术成分，就常无法完全自动导出，反之能自动导出的图如 E-R model 或 XMI model 等就不需要人去做 abstraction ---- 也就是不需要做’设计’了。由于是人在做设计，则常见的行为是 iterative，没有严格的前后关系。

sakurracn: 关于 UML 在代理中的应用细节问题!! 代理技术中，每个代理具有一个多层目标树。为了完成顶层目标，可能通过下层子目标的“and"或”or“完成。且不同的环境情况，顶层目标的完成可能会有不同的最优实现路径。对这种大任务分成小任务，好像结构化设计中的模块设计，应如何利用 UML 处理？用例图可以吗？但它似乎重点在系统与外界的区分，而不在内部细节上。

Tom Kao: 您可把 agent 视为一个 system，目标视为其 use cases，每一个 use case 背后皆有一个它专有的 control object，于是就有了 a hierarchy of control objects 了. 于是能使用 Gamma 的’composite’design patterns 表达出目标树之结构. 接着，identify 出各 use cases 共用之 entity objects，然后从各 control object 里尽量将共用的 operations or rules 移到适当的 entity objects 里，将 control objects 虚化，而将 entity objects 实化，就能 work 了. 至于如何将 operations or rules 正确 assign 到适当的 entity objects 里，请参阅’Applying UML and Patterns’一书(ISBN: 0-13-748880-7)的 Grasp patterns. 以上只是可行的 design 之一但非唯一之 solution.

yangboy: 我正在学 rational rose 2000，我发现从 use case 中实现具体的 sequence diagram 是个难点。实现具体的类难把握，能否介绍一些 材料或书。有无具体的实现例子，如财务软件，通讯软件，从 use case view 到 logic view. 因为要考虑代码的复用性等。

Tom Kao: 从 use case view 到 logic view 的前提条件是---- use case diagram 和 use case description 必须正确，但是在实务上，常欠缺此条件，尤其是初学者. 解决途径之一是采取 iterative 策略，另一途径是 parallel 策略，建议您参考”Advanced Use Case Modeling”一书(ISBN: 0-201-61592-4)的 chapter 11 & 12，其中也有蛮好的例子. 请留意, use case view 背后是 flow thinkng，而 sequence diagram 和 logical view 背后是 object thinking，只要两种思维能均衡互补就能顺利从 use case view 落实为 logic view 了.

KSHANG: 我是 UML 的初学者，准备用 DELPHI 开发一个与数据库有关的 FTP 客户端应用；系统的功能是实现一些客户数据文件的远程传输，（客户信息保存在数据库中），怎样设计一个 3 层模式来实现？

Tom Kao: 因为您的功能只是单纯的数据传输, 并无显着的 business logic, 所以在客户端的应用只需 2-tier 系统就够了。至于 Server 端可能为了维护数据库的高效率而需要 3-tier 模式，此 3-tier 模式也很简单, 只需在 App Server 上摆上一些 Stateless objects 就行了。如果有较复杂的 business logic 时才需要复杂的 3-tier 模式。两端若皆提供 Web service，就互相呼叫对方的 Web service 就通了。

CFL: 1.在收集 User requirement 使用 Use Case 时, 若用 package 来定义 Use Case boundary 是否有思维上的错误? 但 package 若以功能来区分时,并由此 package 的分类来 group Use Case 会不会太早陷入功能需求,而非 User 需求? 2. 从 Business Use Case 及 Business worker's goal 导出 system actor 及 System Use Case 来进一步分析而导出 business object ,并以其互动关系来验证是否满足 Use Case 您认为这样的思维正确吗? 如果正确,在 model 里该如何衔接此二种 model?? 谢谢!!

Tom Kao: 在 Business level 上，像公司、部门等大多是依功能而区分，因此用 Package 来 mapping 这些功能需求是恰当的。为何要避免功能分解呢? 其理由是: 功能分解无法引导人们发现共享(reuse or share)的机会，如果您的目的是共享 software components，则在 business level 用 package 来定义 boundary 是合适的。到了 software system level，就必须有个跨 package(每个 package 内有数个 use cases)的 class diagram 来呈现 sharable components, 这就是 object-decomposition 了。所以您所提的导出过程是正确的。两种 model 是 abstract 与 detailed 之 mapping 关系, 只是共享相同的 business concepts (例如两个 level 都有 Customer、Account 等 concepts)而已, 我不明白您”衔接”的意思。

freeyourmind: 老师你好：我现在正在看 user interface patterns 这本书，里面有个术语，比如；We incorporated these decisions into patterns event Handler, Complete Update, and Multiple Update and used these patterns to refractor Subform,Alternative Sub forms,Subform Selection,Subform Match,and Subform Mismatch.这里的关键词就是 subform, 直译就是从属形式，但是我觉得放在界面模式里面讲好像不大通，我想翻译成子表类，但是完全和字典的意思不相关，不知道您怎么说。谢谢

Tom Kao: 依据我的理解, Subform 是指 GUI 上的 Form 及 Subform 而已，也许直称为”子画面”就行了。

aloely: 请问，如何使用 rational rose 实现最后的代码转换？是不是一定要分析到组件图?还有，rose 中框架代码生成是如何实现的呢？我现在只会用 rose 画一些用例图这样的东西是不是先生成框架代码然后我们可以以这些代码作参考编程序？

Tom Kao: 通常设计到 Class Diagram 就行了。框架代码生成大多依据 Class Diagram 的静态关系而产生的，因为框架代码几乎都是静态结构，您必须自行添加动态的部分。先画用例图及类图，再生成框架代码然后以这些代码作参考编程序。

maji: 您好！我正在学习《uml 用户指南》，有几个问题搞不懂想请你指教。 1。crc 卡是何东西 2。ocl 书写约束表达式是什么 3。在书中多次提到了 uml 的工具。请问这有哪些工具。 谢谢！

Tom Kao: CRC 是个好东西，让我们学习安排角色(role)，及各角色之间的合作，这是 oo 软件设计的基础思维(thinking)。过去软件设计者都是安排计算机担任所有工作(job)，其心中的焦点是”安排计算机依序工作”，好象安排一位演员依序演出水浒传，心中只关注演员。CRC 让您忘记计算机(演员)，而专心写水浒传，于是您关心的是宋江、武松等角色了。这有待您的领悟了。Ocl 约束表达式是针对一个集合，设定集合内的元素的 property 或 behavior。例如，一个”学校”class 内有许多”学生”objects，您能写 OCL 约束表达式限制只有”好”学生才能参加童子军等。至于 Tool 请您参考我回答曲江的问题。

kenzomancn: RequisitePro 如何与 Microsoft SourceSafe 集成?

Tom Kao: RequisitePro 与 Microsoft SourceSafe 系统与系统的直接连结，我们并没有这方面的经验，但是我们把 RequisitePro 用在前段的需求分析，而 Microsoft SourceSafe 用在后段的开发管理，之间的 trace 是人为的，似乎有些落伍了。

sword_hero: 想问您一个问题，我在概要设计是这样做的，根据需求分析先想到一个框架，其中某些问题就直接想出来就行了，对于某些问题的细节我就通过编程来试试看实现，先编个大致的框架，在编程中我就可以大致知道很多需要注意的地方以及如何分配功能，但由于工期比较紧，往往在我编程中想概要设计应该如何设计的阶段 boss 就要来催了，如果我说概要设计还没好，他就会说，我都看你在编程了，怎么回事？那我只有把编的代码略加修改就投入使用了，而概要设计书一般写的比较晚，我这样做行不行？有什么后果，请您指点一二，谢谢

Tom Kao: 因工期紧，所以 User 和 Boss 愿意接受”quick & dirty”的 code,是全世界都头痛的问题，对于讲究结构美感的工程师而言，是一件难受的事。随着系统愈来愈大愈复杂，也愈分散，boss 和 user 愈来愈重视企业流程(business process)的整合串接。如果您能将 Prototyping 的目的转向优先展示”系统整合设计”，然后才是测试”细节设计”，通常您的 Boss 会欢迎。因此，您应该坚持您的理想，但必须改进 Prototyping 的焦点就行了。

Albert: 请问在做面向对象分析设计的阶段，要不要考虑通讯（比如 socket）和控制（多进程、线程），还是在构造的时候再考虑？举个简单的例子就是，开发一个分布式的系统，所以对象可能分布在不同机器，它们之间的消息只能通过 socket 通讯实现，但要不要在设计阶段就考虑呢？

Tom Kao: 在分布式系统里，subsystem 之间的接口是 Architect 所考虑的重点通常在正式系统分析之前的 Architectural Prototyping 时就已经分析并测试了，这个 very front-end 的分析就是通称的 Architectural Analysis。因为分布式的 Web Service 系统整合愈来愈热门，所以 Architecture-centric 策略愈重要，Architectural Analysis 也愈是大型系统建构的基础。

曲江: UML 的建模工具，有多少种，分别是什么呢？它们之间的差别在于什么呢？哪一种建模工具更好一些呢？谢谢!!

Tom Kao: UML 的建模工具有很多种，各有特色，例如 Rose 最为流行，Together J 提供完整的 Patterns 及多花样的逆向生成功能，Visio 较便宜 ….. 我想没有好坏，只是适合不适合，或习惯的问题而已。

张人清: 我在使用 Rational rose 2000 中，想根据现存的代码逆向生成软件模型，但很难受的是该功能的头文件搜索功能好像很差，当然，由于我们系统较大，头文件目录很多，嵌套层次较深。请问有无高招?先谢了！噢，我使用的是 C++标准。

Tom Kao: 以我们的一些经验而言，这似乎没有高招，凡是高手写的 code，逆向生成都很困难。

唐文明: 我只会 VB，能学 UML 吗？

Tom Kao: It’s sure. 尤其 VB.net 已经能充分表达 UML 的所有 concepts 了，您可从 VB.net 的 class, interface 等所学到的 concepts 迅速理解 UML notation 的涵义，是很有帮助的。

macro_fu: 现在我要对客户买东西建模，客户类和商品接口的关系是聚合、依赖还是关联？谢谢！

Tom Kao: 您可能需要另外两个类:”购买”类和”购买细项”类。 客户类与购买类之间是 association，购买类与细项类之间是 aggregation，细项类与商品类之间是 association。

zjuxxl: 高先生，你好！我选择了“基于 UML 自动测试系统建模方法的研究"作为博士课题，最初想法是通过对自动测试系统的共性与特征分析，提出一个通用的、可复用的对象框架，使用扩展的 UML 对框架进行描述，并提出一种形式语言来描述扩展的 UM，最后基于该形式化语义研制 CASE，随着课题的深入，感觉题目太大，希望高先生提一些宝贵意见。

Tom Kao: 我想这是很好的研究方向，不过我这方面并无许多心得，无法给您意见。

Wallace: 高先生，您好！我有个关于 OO 对象如何存储于关系数据库的问题向请教您。假设我现在有个用户类 TUser，属性是 Name,Age,方法是 AddUser()它的一个对象叫 User,现在系统操作员想增加一个用户，他在窗体上输入 User 的相关信息，然后点击“确定”按钮。就这样一个过程，怎样报用户信息保存到关系数据库上，最好能给出相应的代码。多谢！

Tom Kao: 如果您的 RDB 有个 Tuser table，那就在 AddUser()里发出 SQL statement 给 RDBMS 就行了，如果您的 RDB 经 Normalization 时将 Tuser table 分解为数个小 tables，则 AddUser()就必须逐一 update 这些 tables 了。我不知您的 RDBMS 厂牌及 DB schema 所以无法提供代码。

bobby: 请问高老师,我现在写一个程序,想实现这样的目标,1.程序本身能非常容易的实现拆卸,组装和 UI 的改变.2.在本工程实现的组件很非常容易的被以后的工程复用.3.如何处理好 UI 和其功能的关系,采用什么结构比较好？请问如何设计能实现以上目标？非常感谢！注:本程序没有采用 WIN32 的标准 UI.

Tom Kao: Step1. Server 组件提供 webservice 给 UI 使用. Step2. 分析出 Server side 的 Business Components, 落实于 Server 的 container 里, 例如成为 EJB 的 session beans. 这些 business components 是复用的目标. Step3. 建构 DB 的 façade components 去 encapsulate DB, 例如落实为 EJB 的 entity beans. 我想这是基本结构, 还要加上您的 domain 特色与设计技巧.

xzhqzz: 请问 sun 的 jms 您用过吗？如何保证消息的可靠传输？什么时候需要用到 jms？还有 IBM 的 MQ 什么时候应该使用？

Tom Kao: asynchronous 的沟通，只能依赖 message queue system 的反向 notify 而得知消息的接收, 或是 server 透过其它管道如 mail 回传。当两个 use cases 的执行无法 synchronous 时，jms 或 MQ 或 MSMQ 是必要的。

XGE: 老师您好：我现在做一个比较大的项目设计，用 UML 建模，用 ROSE 作为工具,请问我是不是要对每个有相同操作而不同内容的 user case 画一个 Sequence 图。例如系统里有订单管理、合同管理，这两个模块都有增、删、改功能，是不是相应的都要画 Sequence 图。

Tom Kao: 您所提到：”有相同操作”。 但是 use case 的分合依据通常不是”操作的相不相同”，而是依据”goal 的相同”来判断的，如果 goal 相同则应该合，否则就应该各自叙述了。记得, goal 是从 user’s view 而得的, 操作是由 developer’s view 才看得到的。

Zheng.w: 高先生，您好。 注意到您说过"我主要是研究设计思维"，这也是我想深入提高的，想请您推荐几个与此相关网络资源，谢谢。

Tom Kao: 关于设计思维, 我目前的焦点是 Architectural Style, 包括结构美学, 老子的虚实相依, 孙子的应行于无穷, 吴清源的整体、和谐、创新, 印度的空无哲学等等。大部分都跟 Software Architecture(如 Framework 设计等)有关。建议的网站是 www.wwisa.org。 软件工程包括有两项：结构设计和施工控管。像 CMMI 比较偏向施工控管, 降低成本。设计思维追求美感、创新应变和多样化。

wildmum: 我现在想根据人员信息库，比如有工号，姓名，部门，同时还有调动表：调动时间，工号，前部门，后部门，离职表：工号，部门，时间。根据上面的想动态获得某月的人员情况，如在职人员情况，离职人员情况，请问有没有一个合理的主表设计，可以很简单的获得动态的人事信息，我根据上面的表设计后处理起来很困难。现在这个问题在实际中经常碰到。比如想统计以前某月公司的人员情况。查询某人在某月在公司的情况请问有没有一个很好的数据表设计方案呢 。这个问题我想 了很久了，就是没有想出一个很理想的方法。

Tom Kao: 我想您的情况已经接近于 OLAP 或 Data Warehouse 的使用了, 您需要增加一个新的 Month 表, 套用 Data Warehouse 的 terms, 可说您有两个 fact tables ----调动表和离职表, 各 link 到三个 dimension tables ---- Month, Employee, Department。以上看法您认为如何呢？

wonzoon: 高先生，请介绍一些讨论 N-tier 结构和 layering 观念的网路资源和书籍，谢谢！

Tom Kao: 可参考: 1.Component-Based Software Engineering, ISBN: 0-201-70485-4. 2. Developing Enterprise Java Applications with J2EE and UML. ISBN: 0-201-73829-5. 3. UML components. ISBN: 0-201-70851-5. 4. Realizing e-Business with Components. ISBN: 0-201-67520-X. 5. Software Reuse. ISBN: 0-201-92476-5.

wmz: 请教高先生：一个数据库应用程序，功能是根据用户输入的条件，查询公司的人事信息。界面由两个 dialog 配合工作，一个用于接收用户的查询条件，另一个用于显示查询得到的记录。我想问的是在此应用中,哪些处理应纳入到“business object”中？或者说“business object”中到底包括哪些动作？SQL SELECT 字符串的生成处理应放在哪里？我对于类似的数据库应用程序的架构的划分总是一头雾水，请高先生指点一下，谢谢！

Tom Kao: 广义的 business objects 包括: 1) business process objects, 2) business entity object 又称为 business conceptual objects, 3) business data-source objects. 在您所提的 case 中, 可设计一个 bpo 类----Class EmployeeQuery, 一个 bo 类 ---- Class Employee, 一个 bdso 类---- Class Employee_db, 构成一个 4-tier 架构. SQL Select statement 应该放在 Empolyee_db 类里面。

UMLChina

非程序员

软件工程杂志，已有数万用户

UMLChina讨论组

已有数万名成员

UMLChina.com

1999年11月成立，，专注UML技术的应用

专家交流

提供与世界级专家交流的窗口

构建 EJB 应用—模式集合（下）

Eberhard Wolff 著，杨德仁 译

查看评论

持久性优化模式

速查表

假若……	那么使用……
…旧数据必须保留而非删除，例如因为法律原因	历史化状态（Historized State）：保留旧数据并引入删除标志和时间戳。
…访问数据库或旧系统中的数据太慢	中间层缓存（Middle Tier Cache）：数据缓存于应用服务器中
…数据库更新频繁但数据很少改变	延期存储（Deferred Store）：只有数据真正改变才存储，利用标识去标记改变了的数据。
…完整的实体豆(Bean)被装进应用服务器，而通常只有部分会用到	惰性状态（Lazy State）：在需要时才装载状态。

历史化状态（Historized State）

因法律或其他原因你要保留旧数据，然而 EJB 规范要求一旦调用 *ejbRemove()*，就要删除数据，且没有内建的归档标志。

通常，不保留旧数据就删除实体组件或改变数据，因此一个改变仅仅是用新数据覆盖旧数据。但是，往往因法律原因这种做法不可接受。法律要求旧数据归档而非删除，以便能重建旧数据。在这种情形下，通常存储旧数据以便能用手工 SQL 查询语句（历史化）重新产生它就足够了。但存储旧版本以备发生错误时撤销就有意义了。在这种情形下，旧数据必须能被应用和用户（版本）访问。

因此

按照保留数据而非删除或仅仅改变的方法实现实体组件。在数据库中引入一个时间戳或版本。将数据库记录标记删除标志而非真正删除。

概要

基本思想是按照在数据库中保存旧数据的方法实现实体豆(Bean)的生命周期回调组件需要的操作。这意味着没有进行 SQL 删除操作，而是用 SQL 更新数据标记删除标志。如果数据要更新，将用 SQL 插入新版数据代替 SQL 更新操作。SQL 查询获取数据库记录要用的最高版本号，而不能返回标为删除的记录。

有趣的问题是主键。通常主键固定不变，即若没有历史化数据，主键是一个字符串，而时间戳仅内部保存。因此若要用某个主键访问实体豆(Bean)，你得到的数据行具有这个主键和最高版本号或时间戳。要注意，实体豆(Bean)的主键不能是数据库中的主键：因为或许存在具有相同主键而不同时间戳的数据，它并不唯一。

要使客户端能访问实体豆(Bean)的不同版本，你必须使主键包含时间戳和原来的主键，否则无法区分实体豆(Bean)的不同版本。这个主键通常包括时间戳和旧主键例如顾客姓名或编码。这引起通常对特定时间戳并不感兴趣但在某个确定时间点版本有效的问题。设想你要查看 2000 年 3 月 21 日的顾客地址，你并不对时间戳是 1 月 2 日还是 3 月 30 日感兴趣，而只对 3 月 21 日的有效版本感兴趣。要定义发现方法允许用户查找在某个日期有效的 Bean 版本。对版本化的数据的访问应该是只读的，因为它是归档数据，只是展示给用户。这种数据的改变通常没有意义，也难于处理。这意味着不用实现 *create()*方法或 *jester()*或 *set……()*方法。

相关模式

要保证新版本号只在数据确实改变时产生，你应该用***延期存储***模式。

惰性状态

你注意到，许多时间用于装载实体豆(Bean)的状态。它们代表着具有许多数据的复杂业务实体。这个数据被装载到一个大的数据库操作（“大餐”）中，但通常只需要部分数据。

实体豆(Bean)代表着具有所有数据的完整业务实体。若用容器管理持久性，或用实体管理持久性作一个自然实现，这意味着每个实体所有数据都从数据库中装载并在数据处理后存储。对于小而简单的实体豆(Bean)是可接受的。事实上，在这种情形下，它是最好的选择，因只用到一个数据库操作，保持了较小开销。然而，对于大而复杂的数据，这种操作很费时。若通常只用到部分数据，这既耗费时间又占浪费内存。把实体豆(Bean)分成多个小实体豆(Bean)并非良策，因组件应该是粗粒度，这也会引起额外的复杂性、导致逻辑实体被分成多个技术实体，从而使得客户难以使用这种组件。

因此

在真正要用到实体豆(Bean)的数据时才装载。例如，在访问器操作中可以这样做。因你要自己开发数据库代码，你就得用 Bean 管理持久性。

概要

这种模式也广泛应用于面向对象数据库系统（OODBMS）中。它们通常只装载一个对象而不装载任何引用的对象。只有对其它对象的引用被解析时，引用的对象才装入内存。OODBMS 试图把经常整体访问的几个对象集群以改进性能，然后一次装载整个对象块，这样，在引用的对象访问时，它们已经在内存中。利用所有这些先进技术，***惰性状态（LAZY STATE）***甚至在只引用一些小对象时，性能都表现良好。

当***惰性状态***与 EJB 共用时，事情就不同了。每个片断都用自己的数据库操作装载。只有大批量数据的装载延迟，性能好处才能达到。否则，数据库操作的开销太大以致很难注意到任何性能改进。

相关模式

利用***类型管理器***（TYPE MANAGER）能取代它。***类型管理器***直接从数据库中装载所有数据，从不装载任何不需要的数据。然而，这也意味着对数据的每次访问都引起数据库访问，而用***惰性状态***时，数据保存在实体豆(Bean)之内，因此被缓存。

中间层缓存（Middle Tier Cache）

如果应用系统频繁使用业务实体，缓慢的数据库访问成为严重的瓶颈。通常因为许多应用并发操作数据库，慢速数据库经常在频繁使用老旧落后的系统中找到。然而，在许多情形下，应用系统只需访问旧数据库中的某些数据子集。

考虑一个管理股票交易数据的应用系统，一个大型后台数据库存储着 20 多年来的每个股票符号。在线交易应用系统把数据输入数据库，许多其它系统也并发访问这个数据库。

现在设想你要构建一个基于 web 的股票符号系统。对于基于 web 的服务，慢数据库性能不可接受。在峰值时处理大量的用户负载会进一步减慢数据库，引发交易应用系统的问题。

在诸如数据访问是典型的只读操作的情形，只能有规律地访问数据的子集，即当日的股票符号。

因此

在应用中安装一个中间层缓存，它从旧系统中预装载经常需要的数据。你的实体只用这个缓存——它们不了解后台支撑数据库。决定缓存哪个数据、何时把（潜在的）改变写回旧数据库是缓存的职责。

附加信息

中间层缓存负责保持实体与数据库同步。如果用缓存的应用只读取数据，或是修改特定数据项的唯一应用系统，这个模式最有用。

第一种情形，缓存主要用作读缓存，直接向后台提交变化情况，在我们股票交易的例子中，这是最好的方案。然而，也有第二种情形的应用：应用系统只用于整天收集数据，而只有最后一项（或某些平常情况）必须永久地存储在后台中。那么，那个缓存会整天收集数据，并每天只向旧数据库存储一次。

在缓存遗失情形下，即被请求的数据得不到，缓存必须查询后台并取得数据。然而，如果它判定这些数据或许再也用不到，则它不必再缓存这些数据以备其它访问。

中间层缓存并非完全的通用工件。要有用，它至少需要知道去缓存哪个数据以及缓存多长时间。因为这通常依赖于应用需求和使用模式，因此难于全部自动完成。

中间层缓存能、但不应该用于克服容器的缓存和缓冲池策略中的弱点，理解这点很重要。中间层缓存可以使用数据访问模式的信息预先提取和缓冲那些很可能被用到的数据。而容器没有这种信息，也就不能提供这种服务。

缺点和忠告

在多服务器体系结构中，中间层缓存只应存在于一个实例。否则，不同的中间层缓存必须同步内容。

变体

一个中间层缓存能用几种不同的持久性技术实施。最简单的情形，数据能存储在内存中，当然，这也有缺点，如果服务器崩溃，这些数据也就消失了。另一个方案是用另一个相同类型（至少要兼容）的数据库做备份，并使用脚本语言和数据库命令行工具做传输工作。另一种良好方案是选择使用面向对象数据库，因实体的数据模型（通常由*依赖对象*构成）或多或少可以直接持续存在。

相关模式

这种模式与*实体组件*和*类型管理器*很好平等地配合。毕竟它仅仅是实体的另一种持久性层次。中间层缓存的接口能实现成一个有利于编程的组件。它也可以做成一个外部服务，或通过使用直接（缓存）数据库访问。

迟延的存储（Deferred Store）：

你注意到，虽然实体豆(Bean)很少改变，而数据库频繁更新，这导致性能下降。

至少在交易结束时，实体豆(Bean)状态被频繁存储到数据库中。但实体豆(Bean)的状态经常不变，因此数据库操作浪费了时间。频繁的调用只是用于同步多个表示同样数据行的实体豆(Bean)：假若两个实体豆(Bean)代表相同的顾客，这个数据在每个方法调用前装载和在调用后存储，实体豆(Bean)表现出通过数据库同步。然而，容器没有办法检测这个状态是否真正变化。这样，或许发生许多不必要的更新操作。

因此

只在状态改变时才存储到数据库。在实体组件中引入一个实例变量跟踪状态是否变化。

概要

这种技术也经常在面向对象的数据库系统中使用。若一个对象改变了，它被标记成“脏”，然后才能存储到数据库中，否则不进行存储。问题是，当对象改变了，面向对象的数据库（OODBMS）如何观察到。有些 OODBMS 要求开发者手工完成，其它的用内存管理特技或在应用系统中插入额外的代码来达到目的。

对于企图的模式，这必须手工完成，也就是每当实例变量的值改变时，必须设置一个布尔实例变量的值，以便记录写入数据库。在写回数据库或从数据库读出这个状态后，该布尔变量必须复位。

从性能方面看，这种模式应该用于每个实体豆(Bean)，因为检查和设置变量的开销可以忽略不计，而减少数据库操作的收获很可观。问题是在*实体组件*的状态改变时要真正设置变量。若在这个代码中存在错误，那么可能不会更新数据库，因此改变就丢失了。

变体

实现一个较细粒度的修改控制也是可能的：如果有些实例变量包含大多数数据，为它们每个都实现一个布尔变量指示变化、只存储那些改变了的实例变量是有意义的。这与***惰性状态***密切相关：当惰性状态尽量避免复杂数据的非必要装载时，***迟期存储***在这种情形下能用于防止复杂数据不必要的存储。

如果在***实体组件***中包括更复杂的业务逻辑，事情就更加复杂化了，这会引发更多改变组件状态而没有设置变量的可能性。你可以通过使用访问方法而非直接访问变量来尽力避免。为此，你创建一个类，包含实例变量作为私有变量，为其提供公共访问方法。在实体组件中实际实现的子类中，你可以实现业务方法。因为只能通过使用访问方法存取实例变量。

状态访问模式

速查表

假若……	那么使用……
…一个实体豆(Bean)的一些属性经常一起修改	批量设置器（Bulk Setter）：引入一种新方法，允许修改具有一种方法调用
…一个实体豆(Bean)的一些属性经常被同时读取和修改	值对象（Value Object）：利用一个对象描述所有被访问的属性。
…几个实体豆(Bean)的属性经常被同时读取和修改	组合值对象（Combined Value Object）：使用包含各种来源的数据的一个值对象。
…一个实体豆(Bean)的某些属性经常同时被读取和修改。哪个属性被修改了？	散列表值对象（Hashtable Value Object）：把属性存储为散列表中的键值对，用作值对象
…一个实体的有些属性经常被同时读取和修改，访问的属性有许多种组合	值对象创建器（Value Object Creator）：提供通用工厂方法创建不同类型的值对象
…许多数据集合要从服务器传输到客户端，例如，要显示它们	批量阅读器（Bulk Reader）：引入一种会话豆(Bean)，直接访问数据库并返回值对象集合
…许多数据集和被按同样方法修改	实体批量修改（Entity Bulk Modification）：引入一种会话豆(Bean)，直接修改数据库
…许多数据集合要逐个从服务器传输到客户端，例如，要修改它们	服务器端迭代器（Server Side Iterator）：数据被透明地批量转移

批量设置器（Bulk Setter）

对实体豆(Bean)的每次调用都耗费时间，因为要远程调用。这使得这种调用代价高。而且，对 Bean 的每次调用都受到容器侦听和检查，致使开销更大。取决于容器的实现，对于同步数据库与实体豆(Bean)状态的每个操作，每个调用甚至包含到一个或多次数据库访问。这样，反复调用对一个 Bean 的设置器操作是低效的，应该避免。

设想一个人（*Person*）豆(Bean)，它有设置器操作如 *setName()*，*setFirstName()*，*setDateofBirth()*。虽然能逐个设置这些属性，但通常需要同时更新这三个。特别地，如果你使用 GUI 对话框，也许需要设置一组属性。使用这些设置器操作需要三个单独调用，每次都要通过网络并受到容器侦听。

因此

提供额外的操作，即所谓的***批量设置器***，它把一组经常同时修改的属性当作参数。无论何时更新这些属性组合，利用***批量设置器***而不使用连续调用单个原子设置器操作。

优点

利用这种模式允许应用能更好伸缩，减少网络开销问题。这种模式的副作用是，如果把许多设置器操作绑成一组，属性将在同一事务内设置。不过，性能是这种模式的动机。

变体

当使用这种模式时，完全清除原子设置/获取操作是可能的，然而，这并非总是良策。这取决于用例——如果只要改变一个属性，则原子访问器可行。如果只更新属性组（例如，表单提交之后），那么就能安全地从***组件接口***中部分或完全清除它们。***批量设置器***调用也有副作用，就是所有参数在一个操作中更新，它通常意味在一个事务中。因此，若要保证一套属性原子性地更新，那么使用该模式并清除原子设置器操作。

若原子设置器方法可用，那么利用这些方法实现***批量设置器***操作。因此，你仍然只在代码的一处修改成员，并有可能局部增加或修改设置行为。不必使原子设置器操作对客户可见。它们或许是内部操作，即它们不需要成为***组件接口***的组成部分。

相关模式

除了更新外，如果还要使读取过程更有效，或参数构成一个语义组，那么使用***值对象***代替它。与值对象比较，***批量设置器***的一个优点是，因为***批量设置器***不给应用添加新类，它更容易实现。增加一个简单操作给组件增加的复杂性很小。

如果这些参数形成了语义组，以及参数集的生命周期与实体豆(Bean)的生命周期相联系，那么使用***依赖对象***。如果有许多***批量设置器***或你经常要增加新***批量设置器***，因此需要经常改变豆(Bean)的接口，那么使用***散列表值对象***或***值对象创建器***模式。

值对象（Value Object）

对实体豆(Bean)的每次调用都相当耗时。这是因为远程调用是必需的并且代价高。因为对豆(Bean)操作的每次调用，容器都要检查安全性和事务设置。因此许多获取操作调用是低效的。

设想一个*人（Person）*豆(Bean)，它有获取成员如姓、名、生日等操作。需要经常同时得到这三个值。一个例子是你要实现一个用户接口，经常有具备同时显示的属性组的对话框。你需要调用豆(Bean)的许多操作去获取在对话框中显示的所有属性。因为对容器的有多个调用，每个都要耗时完成，显示对话框花费很长时间。

因此

增加额外的一个类，它包含你要同时得到的所有属性。也给豆(Bean)增加一个*工厂方法*，返回新类的实例包含所有数据。这便是所谓的值对象，当你要获取所有信息时，利用实体豆(Bean)的工厂方法一次调用就获得所有数据。当然，*值对象*要用值传递。

概要

值对象必须用值传递。确保不能扩展 Java.rmi.Remote 接口。要保证对象的内容不变化，仅在***值对象***中实现获取方法。

优点

利用这种模式允许你的应用能更好扩展，并使得网络和容器侦听开销问题较少。可以把***值对象***用作其它操作如设置器和业务操作的参数。因此你为实体豆(Bean)做了小而明晰的接口。

缺点

使用***值对象***的问题是你在客户端保存豆(Bean)的内部数据。若数据仍然有效或在同时已被改变，那么你将得不到信息。因此，应尽量短期保存数据以减少这个问题或在使用数据之前检查数据正确性。也可利用某种时间戳机制去检查数据是否仍然正确。在某些情形下，这并非必要，因客户端并不需要总是显示数据的最新版本，或者保存在***值对象***中的数据是只读数据，因此在此期间不能改变。

值对象的另一个问题是，其成员和结构大多由客户端用例驱动。因此豆(Bean)的接口严重依赖于客户端用例，因此它在项目生命期内或许不稳定。要处理这个问题，参看***值对象创建器***和***散列表值对象***模式。这两种模式为新的值对象处理从需求去耦组件接口的问题。

变体

值对象也有其它一些方法。例如，这些方法能用于检查成员的一致性。实现实体豆(Bean)的一个特殊方法是把所有成员从实体豆(Bean)移到***值对象***，并且只把这个值对象作为实体豆(Bean)的成员。实体豆(Bean)也能从***值对象***继承。

值对象也能作为惰性装载其它的值对象或访问该实体豆(Bean)本身的一种方法。例如，只包含一个 *Person* 实体姓名信息的一个*值对象*能提供操作 *getEntity()*，返回实体本身或提供操作获取额外的*值对象*诸如 *getAdressInfo()* 或 *getJoBinfo()*。这样，不同的值对象能提供实体的不同视图，同时仍然提供对实体本身的访问，如果有必要的话。例如，一个表视图能用值对象包含所需要的信息，只有在显示的某个实体被选中用于编辑时，才能直接访问实体。

相关模式

批量设置器模式解决的问题与值对象一样，但只用于设置操作，***值对象***与***依赖对象***相似，但没有语义意义。

值对象能与本书描述以外的一些其它模式结合。一个例子是，在***批量阅读器***模式中，返回属性在返回客户端时要被分组，*值对象*能用于此目的。

*散列表值对象*和*值对象创建器*解决了客户端用例对实体豆(Bean)接口有直接影响的问题，因为需要*值对象*的*工厂方法*。

组合值对象（Combined Value Object）

在一个用例期间，客户端需要几个实体的状态（或部分状态）。在一般方法中，客户端将分别单独接触每个实体，查询属性（或更可取地获得相关的值对象）。然而，这涉及几个网络跳动，它要求知道客户端实体间的关系以及如何导航它们。

考虑一个*订单(Order)*实体，一个订单包含了直接与订单相关的数据，它与几个*项目*有 1:n 的关联，而每个*项目*都引用到*产品（Product）*。当然，*出价（Offer）*有对请求*订单*的*顾客(Customer)*的引用。

这个*订单*的两个不同视图似乎有用。一个用于几个订单的表视图中。你可用*批量阅读器*获得*订单*的值对象清单。另一个用于检查一个*订单*的细节情况。利用标准的，基于***值对象***的方法，你可访问订单得到其值对象，然后访问*项目*及其相关的*产品*，然后访问*顾客*，逐个从中得到***值对象***。这再三地导致许多网络跳动，而这正是应该减少的。

此外，客户端程序员要知道实体之间的关系，并必须手工导航它们。

因此

提供一种特殊的会话豆(Bean)，从几个相关实体收集数据，并创建一个*组合值对象*)，这个值对象然后返回给请求的客户端。

优点

这种模式有助于为几个实体创建用例的或客户端特定的值对象，然而，不同的实体并不直接相互依赖。*外观（Façade）*会话豆(Bean)只了解这种关系。

利用这种模式可以改进性能，因为减少了网络跳动数目并且降低了实体之间的依赖性。因为这些组合值对象为“外部”创建。

缺点

当然，在不好的方面，客户端编程员需要知道，还有一个额外组件提供要求的聚合信息。当然，创建***组合值对象***的***组件***依赖于它使用的实体——但这并不成问题，因为这种***会话组件***通常总是用例特定的。

若要扩展使用这种模式，你应该评估体系结构，因为***组件***可能粒度太细，你应该使用***依赖对象***而非组织的状态。

相关模式

如果这种模式用于***业务组件***语境中，就特别有用，业务组件用*外观（Façade）*对客户端屏蔽其内部结构。

如果这种***组合值对象***的内容有规律改变，那么为这种会话豆(Bean)使用***值对象创建器***。

如果客户端需要***组合值对象***的多个实例，那么考虑使用批量阅读器。

哈什表值对象（Hashtable Value Object）

利用值对象设置和获取实体豆(Bean)的属性子集。当你要设置或获取属性的许多不同置换时，也需要在组件内有许多不同的*值对象*和*工厂方法*。因此有许多只处理值对象的操作。

如果客户端变化迅速，你也要经常改变组件接口，这也引发频繁重新部署的需要。因此使用许多值对象会导致不稳定的、臃肿的*组件接口*。

设想实体豆(Bean)有很多属性。例如，代表在分布式项目管理系统中的项目的实体豆(Bean)。因为有不同的 GUI 对话框访问工程参数的子集，要引入不同的值对象按可行的方法访问属性。项目实体豆(Bean)的接口负载着设置和获取*值对象*的方法。这使得使用项目实体豆(Bean)复杂化且易于出错。

作为一种替代办法，实现一个值对象保持实体豆(Bean)的所有成员。这也并非上策，因这会导致即使客户端用例只用到一小部分子集，很大的值对象也必须传输到客户端。

因此

把属性按名字/取值对插入散列表。用散列表作为参数去设置或获取实体豆(Bean)的属性。这便是所谓的*散列表值对象*。利用*散列表值对象*你将获得一个小而稳定的组件接口设置属性变量集。

概要

散列表值对象本质上是反射数据结构的简单形式。客户端能对对象支持的属性查询。属性的数量和类型并非在编译时预定义。结果，如果客户端应用能对付这些动态变化的属性，这种模式最有用。典型是那些与用户交互的应用：表视图可以容易为揭示的每个属性增加一列；产品信息页面能显示某个特定产品实例所有可用的属性；也能动态创建编辑器，为每个被揭示的属性提供编辑界面部件。

优点

散列表值对象带来的性能优点与平常的值对象一样，因为许多对实体豆(Bean)的获取或设置操作的调用都被分组成一次调用。

另外，你为实体豆(Bean)获得了稳定的接口，因为新属性集并没有产生新的值对象。你不必改变***组件实现***或重新部署组件，只要***组件***能处理来自调用者的名/值对。而且，无论要获取或设置的属性集是否改变，都不用写新的***值对象***类。

另外，还有所需类较少的优点，因此维护代价更低、应用程序规模更小。这在 EJB 客户端程序是 Applet（小应用程序）时特别重要。

缺点

这种模式的缺点是，你再也不能依赖编译器的类型检查。你可以在组件操作中实现类型检查，但是要把这种类型检查从编译时移到运行时，这导致较高的测试和调试代价。

变体

一个实现变体是不采用散列表、而采用一个从散列表导出的类去传递这些属性。这赋予散列表增加方法的可能性。例如用这种方法，你能检查散列表中的变量。

相关模式

*散列表值对象*是*值对象*的变体。

值对象创建器（Value Object Creator）：

值对象的问题是客户端用例赋予所需值对象的结构和成员。正如在值对象模式中描述，每个值对象都需要一个工厂操作。因此，如果客户端改变了对实体豆(Bean)成员的需要，你就要改变实体豆(Bean)的组件接口，这不仅导致组件接口的频繁变化，而且要经常重新部署实体豆(Bean)。你还要更新访问改变后实体豆(Bean)的其它客户端。

在实体豆(Bean)中具有客户端用例特定的组件接口，折衷了组件接口的重用性。这也会导致接口臃肿，因此应该避免。

设想表示人的实体豆(Bean)。创建两个值对象分别检索姓名（由姓、名组成）和地址成员组。对于每个*值对象*类，创建它们的实体豆(Bean)中有一个*工厂方法*。

现在，你的客户端改变了 GUI，提供新的对话框，用户能同时编辑人的姓名和地址。因此，客户端要在一个步骤中为新对话框取回姓名和地址数据。你必须创建新的*值对象*和*工厂方法*在一次调用中获取需要的所有数据。因此，为了维护客户端用例，你要改变实体豆(Bean)的组件接口。

因此

给实体豆(Bean)增加一个通用的工厂操作，即所谓的*值对象创建器*。这个*值对象创建器*以一个身份标识（id）做参数，识别客户端需要的*值对象*类型。利用这个身份标识动态创建对应的值对象。

然后调用*值对象*方法，从实体豆(Bean)中取出所有必要的数据，因此这种回调方法让值对象用来自实体豆(Bean)的数据填充它自己。最后，将值对象返回调用者。

概要

利用某些配置机制，例如***配置参数***映射值对象的身份标识到值对象的类名（见***配置参数***模式）。

优点

如果使用***值对象创建器***，在实体豆(Bean)可得到更稳定的组件接口，因为这种接口并不依赖于客户端用例，后者定义了哪个工厂操作用于你需要的值对象。

你甚至能在运行时实体豆(Bean)增加新的值对象，因为不必改变实体豆(Bean)的实现。唯一需要的是容器必须访问值对象的实现。

缺点

一个缺点是，因为取出映射信息以及值对象的动态创建，会带来轻微的性能开销。如果完全不用值对象，通常这种开销比做多次远程调用耗时要少得多。

编译时类型检查较少，因为你要在使用***值对象创建器***以后要在客户端 downcast（类型转换）值对象。因此，你的客户端应该能处理通过有缺陷的部署，服务器返回了错误值对象种类的情形。

值对象创建器的实施有相对复杂的结构，这使实体豆(Bean)实现更难于开发、维护和部署。

相关模式

有些配置机制，如在***配置参数***中所述，需要配置身份标识和***值对象***类名之间的映射关系，身份标识描述***值对象***类型。

另一种将组件接口从客户端用例去耦的方法是***散列表值对象***。***散列表值对象***本质上是一种可以用于改变客户端要求的反射数据结构，它的实现更简单。

批量阅读器（Bulk Reader）：

客户端要按表视图显示一组实体，或希望执行另一种批量检索，其中只要求每个实体状态的小部分子集。通常，每个实体（实体豆(Bean)或类型管理器）必须单独访问，因此引起网络流量开销很大（对实体组件更是如此）。

一个客户端应用想显示某类*产品*的清单，这个清单应该只显示产品的 ID、名称及其价格。产品描述成实体组件的自然方法是利用本地接口、并执行一个查找器操作指明所需类别。这个查找器将返回一个引用清单，轮流访问每个引用，在容器中请求一个逻辑组件实例并且对每个请求值对象检索。这严重影响了性能，应予以避免。

对于用***类型管理器***实现的*产品*，情形好些。避免了逻辑实例，但是，每个实体仍然要逐个访问以取出***值对象***。

因此

提供一种特殊***会话组件***提供查找器操作直接访问数据库并为每个找到的实体返回***值对象***清单。

优点

使用这种模式的优点是查询只取出所需要的部分实体状态，并且只查询真正要求的实体而非全部实体。可以通过直接查询数据库做到，而不必访问实体，性能明显提高。通常，返回的信息是只读的，因此没有不一致性问题。

缺点

若实体的结构改变了，不但要改编实体，而且要修改访问（数据库）结构的会话。

变体

如果***值对象***已用于访问实体状态，相同的***值对象***类也可用作对批量阅读器中的查找器操作的返回值。如果只有查找器操作所找到的值的子集实际上是客户端所需，你可用***服务器端迭代器***去减少网络负载，只转输实际需要的***值对象***。

附加信息

这种模式的语境中，返回的***值对象***作为真正实体的某种描述性代理。在许多情形（再考虑产品的表视图）下，最终将用到完整的实体（例如，修改某个特定的订单）。在这种情形下，如果返回的***值对象***提供一个操作返回对完整实体的一个引用，这对客户端就很方便。

服务器端迭代器（Server Side Iterator）：

查询结果通常是结果对象（引用或特定的*值对象*）的集合。如果集合非常庞大，任何客户端都不需要访问所有实体，许多不需要的数据从服务器端传输到客户端。

一个*订单管理（OrderManagement）*组件（基于***类型管理器***）提供一种操作去取出本周的所有订单，按日期降幂排序。匹配的实体非常多，有数千种。客户端应用以一个表视图显示这些订单。查询操作的结果是***值对象***（小的，只含***主键***和一些描术性信息）的集合。在许多场合，用户只需要查看部分结果对象，往往是最近的情况。如果查询操作一次返回所有对象，将传送许多不必要的数据。

因此

在***会话组件***中创建服务器端迭代器，客户端一次从这个迭代器（几个结果对象的小集合）取出成块的数据，只有真正需要时才取出更多数据。这种服务器端迭代器跟踪哪些数据已被客户端请求。

优点

这种模式允许客户端只访问那些真正需要的部分数据，避免了不必要的网络开销（和服务器上的锁定），同时又不分别请求每个值对象。

缺点

然而在实现这种模式时，要考虑几件事情：

- 对每个大查询，要在服务器上创建一个带状态的会话豆(Bean)，这种 bean 或存储整个集合并逐个返回，或用某些技术例如数据库游标把“迭代状态”委托给数据库。
- 客户端在用“完”数据后，要不时取出数据块。若要对客户端不可见，需在客户端有一个帮助器对象，在请求下一个结果对象时，若有必要，它自动取出新数据块。在客户端的工厂有助于隐藏它。
- 有时会话豆(Bean)必须删除，若它不再用到的话，一个客户端因此必须明确调用在服务器端迭代器上 “我用这个迭代器已经完成” 的操作。***会话组件***的超时有助于清除失效的迭代器。

取决于模式实施的方法，例如，因数据库游标不可用，需求的值被缓存在***会话组件***中，服务器的资源要求就相对较高。而且，在查询执行后，在数据库中提交的变化在查询结果中没有反映。

变体

作为这种模式的变体，假若客户端跟踪迭代状态以及后台数据库支持游标，*服务组件*可用于代替***会话组件***，这就导致服务器较轻量级方案。

实体批量修改（Entity Bulk Modifications）

用相似的方法更新一个庞大的实体集合需要按顺序访问各个实体豆(Bean)。然而，访问实体豆(Bean)的代价高，因为它涉及两步操作：首先要取出主键，然后才能取出并实例化实体豆(Bean)。常用两次数据库访问完成[1]。而且每次调用实体豆(Bean)将受到容器安全性和事务设置检查。当所有这些操作要针对每个实体豆(Bean)时，性能将降低。

实施企业资源规划(ERP)系统，*职员*实体实现为实体豆(Bean)。你需要一个操作去增加公司所有职员的薪水。你在*职员*实体豆(Bean)中增加一个方法*increaseWages(percentage)*。然后实现会话豆(Bean)对每个职员实体豆(Bean)调用这个操作。这种操作需要非常长的时间才能完成。

因此

要使用具有在数据库中直接修改实体的操作的会话豆(Bean)。根本不用访问代表实体的实体豆(Bean)。直接更新数据库，不涉及实体豆(Bean)。

实体批量修改并非实体豆(Bean)的替代品，而是处理对大量数据的大规模修改的一种方法。相反，***类型管理器***完全是实体豆(Bean)的替代品。

如果使用***实体批量修改***，对应用服务器和数据库产生的负载很低，因为应用服务器不再实例化单个实体豆(Bean)。数据库负载将减轻，因为你不用读取数据（或许两次）然后更新，而按准则直接更新数据库中的数据。数据库也能使用查询优化器改进性能。

然而，***实体批量修改***要用长时间才能完成，一个原因是实体豆(Bean)表示的一些数据或许在库中被锁住，因为你要修改某些实体豆(Bean)从表中保持的数据实际上在你开始使用***实体批量修改***时正在使用。***实体批量修改***操作要等到锁释放。

[1]实际上，用容器管理持久性时，容器会减少为一次数据库访问。然而，现在大多数容器并没有实现这种优化，而是用额外一次数据库访问取出主键并取出实际的 bean 数据。若用 Bean 管理持久性，你的实体 Bean 访问被映射为上述的两次数据库访问

因为***实体批量修改***在其运行时（取决于数据库使用的锁机制）也保持数据库表或数据行的锁定，这就产生严重的性能问题，甚至访问存储在这个表中的实体豆(Bean)发生超时。在需要同时修改许多实体时，要注意***实体批量修改***的这些副作用。

假若在设置中使用了某些缓存，***实体批量修改***也会发生另一个问题。如果你的缓存不能识别数据库数据的直接改变，那么你的实体豆(Bean)将使用旧数据，直至缓存进行与数据库保持同步的操作。要记住，实体豆(Bean)自己是数据库中的实体被“缓存”的实例。

另外，当你要向会话豆(Bean)增加数据库代码时，你的代码不再独立于数据库，即便你使用容器管理持久性，因此为性能牺牲了可移植性。特别在使用容器管理持久性时，你要知道容器实际上如何把实体豆(Bean)的数据映射到数据库上。

EJB2.0 规范引入所谓的本地业务方法（*home business methods*）。这些方法按缓冲池化的状态在 bean 上执行，因此不能访问来自特定 Bean 的状态。这是 java 中静态方法的 EJB 等价。若用 EJB2.0，你应把***实体批量修改***操作实现成本地业务方法。应当如此，因为***实体批量修改***在概念上是实体豆(Bean)的一部分。

实体批量修改是一种模式，用于以相似方式更新大块数据。若要读取大量的实体，那么使用***批量读取器***。如果客户端应用要贯穿大量实体，那么使用***服务器端迭代器***。

组件变异模式

速查表

假若……	那么使用……
…新组件应提供几个接口	扩展接口（Extension Interface）：组件的实际接口访问几个接口。
…现有的组件应该提供几个接口	多接口(Multiple Interface)：用几个不同接口部署组件的实现。
…组件的接口频繁改变	弱类型接口（Weak Typed Interface）：在接口中只提供一种通用方法，能处理任意请求。
…在不修改组件本身的情形下，组件的部分行为应是可变的	行为柔性(Behavioural Flexibility)（也称***策略***）：把部分功能委托给另一个类。该类由 Java 的动态类装裁。
…在不修改组件本身的情形下，在组件中应能存储额外属性	属性列表（Attribute List）：提供通用方法设置和获取任意属性。

属性列表（Attribute List）

组件具有一套属性，这些属性可通过固化编码访问器操作访问，例如设置器/获取器对或者***值对象***。然而，取决于***组件***的使用，额外的数据要用组件存储，也就是额外的属性是必需的。或许只为了某个用例，需要一个新属性，你不想每次都改变组件的实现。

在医院信息系统中，一个*病人*实体有一套必要的属性，诸如姓名、生日、血型等。除这些属性外，每个医院（系统安装）在实施系统时还有特殊需求。例如，他们要存储病人上次进院时间或家庭医生等。病人实体必须能存储这些额外的、可自由定义的属性。

这种属性储存起来提供与外系统的逻辑连接时，这特别重要。

因此

提供具有属性清单即一套名值对的***组件***，它可被设置属性方法 *setAttribute（name，value）*和获取属性方法 *getAttribute（name）*操作（或其批量访问器优化）访问。

优点

这种模式允许用组件存储任何附加信息。这允许开发者根据用例定制实体。

缺点和忠告

这种模式的缺点是对属性访问有两种不同的编程模式。有些规则的属性用固化编码操作如 *getID()*、*getName()* 等访问，而在***属性清单***中的属性用 *getAttributeValue()*操作访问。而且错误的属性名称编译器也不能用检查出来，也就是访问不存在的属性并没检测，除非代码实际执行。即便是属性存在，也会出现错误。当它们作为通用类型返回时，该类型必须进行类型转换。这些转换是另外一些错误的根源。因此***属性清单***比正常属性更易于出错。这导致额外的测试代价。

变体

在最简单情形中，这些设置/获取属性（*set/getAttribute*）操作允许设置或读取任何属性。在更高级的版本中，允许（甚至委派）的属性清单可以用***配置参数***定义。获取和设置方法必须返回通用类型例如*字符串*或*对象*。

可用这种模式的另一种情形是如果***组件***具有一个大的属性集，可能使用该属性集，但每个实例都只用到其中一小部分属性。

附加信息

因为实体是持久的，所以必须确保这些属性能存储为部分组件状态。在 OODBMS 中，这不成问题，因属性直接存储在清单中。在关系型数据库中，必须实现一个垂直数据模型（利用具有名/值对的表），或者把这些值存储成 BLOB 型（例如，以 name1=value，name2=value2……形式）。后者具有难于通过它们查询的缺点。

扩展接口（Extension Interface）

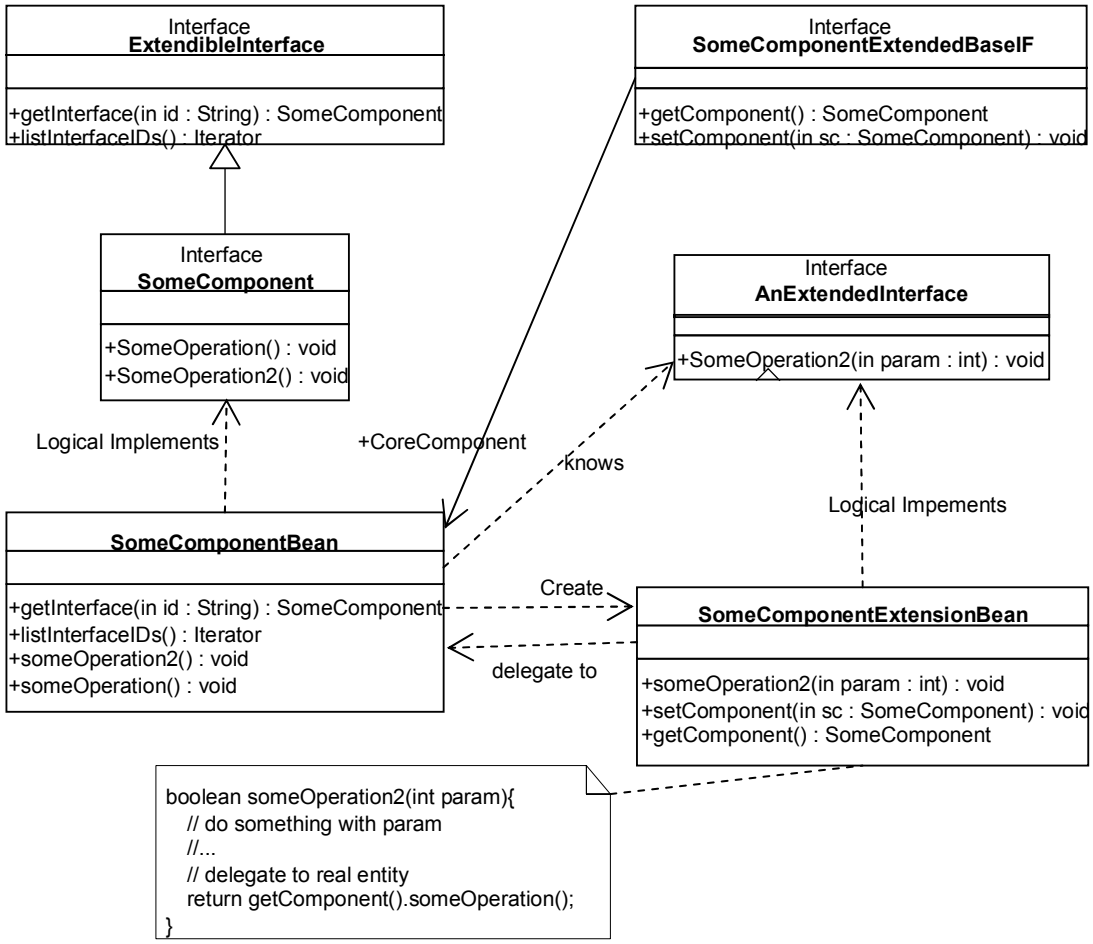
组件通常应该负责一个很好定义的任务。然而，在某些情形中，这种任务难以恰好用一个接口指定，因为不同的客户端具有任务的不同“视图”，或者*组件*简单地承担各种角色（需要不同接口）。共同的问题是接口总在变化，而你希望在允许增加新接口的同时保持旧接口继续可用。

设想一个应用，你要不时改进接口，增加或删除业务方法，例如不再支持的方法。在同一个组件上，旧客户端应可与以前一样访问旧接口，而新客户端应能访问新接口。你不能使用接口继承，因你要从这个组件中改变或删除行为。组件继承是禁止的，概念上也值得怀疑。

这种方法对于***实体组件***特别重要，实体组件通过定义提供业务实体的陈述。对于这种实体，动作或过程可能通常比其数据内容变化更显著。

因此

将客户端使用的操作接口从组件本身分离。让组件明确地管理它支持的接口，允许客户端基于一个键（也许包括版本标签）针对特定接口查询组件。客户端然后用该接口访问组件。



附加信息

这个扩展本身是会话豆(Bean)（不可能使用服务 Bean，因为它们必须跟踪其核心组件），它们最终把操作委托给核心组件。如果针对核心组件的新操作是真正必需的（例如，***实体组件***需要更多的状态），那么这需要在旧的扩展有所反映。这样，旧应用就不能访问该状态。

通常，你不让客户端访问核心组件。如果所有客户端通过扩展达到交互，该模式最有用。

优点

该模式允许接口独立于*组件*实现而进行改进。甚至可能给组件增加新接口而不用重新安装。组件要知道可用的扩展集合，例如通过使用*配置参数*。查询*组件*接口的机制，和必要的扩展 Bean 创建可以平常地实现，正如下例所示。

这种模式对于接口臃肿、特别是针对希望免除的操作是一个强有力方法。这种模式也允许基于角色的开发，因为几种完全不相关的组件可以有相同的接口。这些扩展接口可用于许可控制，注意到这点很重要。

缺点和忠告

一个缺点是，这种模式要求 “返回的”扩展不再需要，否则将发生*会话组件*超时。

相关模式

本质上，这种模式是一种带扩展的适配器[GHJV94]（实际上是对象适配器），这种适配者管理可用的适配器。赋予一个组件几个接口的可选方法是使用*多接口*模式。

这种模式在[POSA2]中做了深入描述。它也能看作角色对象模式[DR??]应用，后者反过来也是一种适配器[GHJV94]。

多接口（Multiple Interfaces）

当一个接口改变时，系统中使用该组件的所有部分也要随之改变，至少要重新编译，因此也需要重新部署。这意味着一个小小的变化会引起一连串的变化。

设想要对一个会话豆(Bean)的远程接口增加一个新方法。尽管这个新版本仍然能支持旧的远程接口，还是要编写新的远程接口。这反过来导致所有旧客户端访问这个组件时失败。因此它们要用新的远程接口重新编译。在高可用性场合中这根本不可接受，甚至在“平常”应用中，许多客户端需要重新编译，这会导致不必要的迟延。

因此

不用改变接口，而提供一种新的（已改变的）接口，同时也保持旧接口可用。两种接口共享实现类，同时在一个实体豆(Bean)情形下访问相同数据。

概要

大多数组件体系结构直接支持一个组件有多个组件接口，因为不同客户端会需要不同的接口。通常在新客户端用新接口时，仍然要提供旧接口。EJB 并不直接支持这种特性。然而，只提供一个接口而实现会话或实体豆(Bean)没有技术理由。一个实现可以创建为支持多个接口，这样接口就容易进化。给 bean 的接口名和 JNDI 名称增加版本标签是个好办法。

当然，这并没有解决提供不同版本实现的问题。假如一个方法的实现应该改变，同样的实现提供两种接口是不够的。然而，实现能取出调用的它远程接口，这样，就提供特定版本的行为。

相关模式

与***扩展接口***相比，这种模式允许对一个组件实现第二个接口，而扩展接口创建额外组件并在组件保留对原来组件的引用，这样，就能提供额外的功能。

多接口也能用于实现*可管理组件*。

行为灵活性（Behavioral Flexibility）（也称为策略）

要保持某些*组件*的行为灵活以促进重用。这种灵活性并不影响*组件*的接口。这种灵活性在组件安装和新的行为中应该是可配置的，实现者预见不到是可能的。

作为一个例子，设想在某个状态发生时，组件要通知其它组件/用户。这种通知能按如下几种方法做到：

- 调用其它组件
- 发送一个 JMS 信息
- 发送一个 Email
- 用一个页面通知一个用户
- 或者其他至今仍然未知的、可行的方法

你不希望让组件的用户去选择合适的行为，如果要求一个新的行为，你也不想改变组件的实现。

这对于那些作为产品销售但行为的某些方面应保持灵活性的组件特别重要。

因此

策略模式[GHVJ94]与 Java 的动态类装载一起使用，在 bean 内部实现不同行为。利用*配置参数*去指定实际需求的类名。如果只允许某个行为选项集，在配置参数中使用键来指定。

优点

这种模式允许你在原来的组件开发者为它计划的地方，“引入”任何一种新行为。然而，请注意，“错误的”行为会损害宿主组件的完整性。

变体

具体类提供给组件的方法基本上有两种。一是利用***配置参数***去指定类名，组件利用动态类装载去创建对象。另一种方法是客户端把对象传递给组件，这需要相应的类也上传到组件。

附加信息

这或许是组件市场的基本技术，因为它允许组件顾客把代码插入到组件中。这使组件具有更强大的定制功能，使得组件应用更广泛。

这本质上是 GoF 为 EJB 世界的策略模式[GHJV94]的改编。

弱类型接口（Weakly Typed Interface）：

给组件增加新功能要求组件接口改变。这通常要求客户端重新编译以便使用新特征。重新部署也是必要的，因为***客户库***的内容也将改变（新接口类等）。这在许多应用中是不可接受的，特别是高可用性的系统。

设想一个客户端应用，用户能选择某些计算在服务器端的一个*计算组件（CalculationComponent）*内执行，或许是作为应用服务提供方案（ASP）的一部分。ASP 供应商很容易更新实现提供新的服务。但改变接口（分布在各个客户端）则困难得多。

这样，ASP 供应商宁愿选择长期稳定的接口，同时也允许由客户端增加和使用新特征。后者要求客户端应用能动态找出该***组件***支持哪些操作。

因此

创建一个通用接口，它具有一个操作允许指定命令，包括参数。然后，这个组件实现解释该命令并返对应的结果。要保证客户端能真正使用新操作，一定要添加反射特征要以允许客户端向组件查询可用的命令。

优点

尽管要修改实现，利用这种模式允许你提供新功能而不用修改组件接口。重新部署通常也是必要的。

缺点和忠告

这种模式也有一些缺点。利用平常的接口，编译器能够进行类型检查——你只能用正确的参数集合执行被允许的操作。利使用弱类型接口将短路这种检验。因此这种模式会引起接口中的隐性依赖性。如果使用一个通用的、反射的客户端，将不会出现这些问题。

变体

使用一些很普遍的格式如 XML（或许加上 HTTP）的弱类型接口能作为一种非常简单的桥，连接另一种*组件*技术（诸如 EJB 对 COM+）中实现的子系统。

附加信息

只有客户端在本质上是通用的，也即如果它能与新发现的命令一起工作，并把这些命令集成到其功能中，才使用这种模式。基于 GUI 的应用似乎特别合适。要使客户端能使用该特征，尚需比明显表露的更多努力。

有些用于分布系统的基础体系结构内建这种模式的实现。例如，CORBA 具有动态调用接口（DII）动态创建对任何方法的调用。与 CORBA 的接口仓库一起，这允许完全动态调用任何 CORBA 对象，而不用提前知道接口。通过使用动态框架接口（DSI），CORBA 提供动态处理远程调用的可能性。这允许远程对象动态实现任意的远程接口。

相关模式

作为业务组件的一个接口，弱类型接口也很有用，因为用到一个合适的通用请求描述（如 XML），集成非常简单。

致谢

撰写如许多的模式确实是一件很繁重的事情，有几个人对我帮助确实很大。首先，我们应该感谢 PLoP 的领导，Ali Arsanjani，他给我们提供了很多有益的建议——令人遗憾的是，由于时间的约束，我们不能全部采纳。但是，它们将在下一个版本中出现。

另外，我们感谢我们在 MATHEMA AG 的同事，他们检查了部分的模式并帮助我们复审了草案。

参考文献

[AG00] Alan Gordon, *The COM and COM+ Programming Primer*; Prentice-Hall, 2000

[AOP] *The Aspect Oriented Programming Homepage*,http://www.parc.xerox.com/csl/projects/aop/

[BOF] Herzum, Sim; *Business Object Factory*; Wiley

[CETUS] Schneider, et. al., *Cetus-Links*, http://www.cetus-links.org

[DR00] Bäumer, Riehle, Siberski, Wulf. "Role Object." In Pattern Languages of Program

Design 4. Edited by Neil Harrison, Brian Foote, and Hans Rohnert; Addison-Wesley, 2000.

[GHJV94] Gamma, Helm, Johnson, Vlissides; *Design Patterns*; Addison-Wesley 1994

[HDSC] IBM Corp, *Hyperspaces*, http://www.research.ibm.com/hyperspace/

[ISE] ISE, *An introduction to Design by Contract*, http://www.eiffel.com/doc/manuals/technology/contract/index.html

[JiniPL] *The Jini Pattern Language*, http://www.cs.wustl.edu/~mk1/AdHocNetworking/

[JS00] Jon Siegel, *CORBA 3,* Wiley, 2000

[JSP00] Subrahmanyam, et. al, *Professional Java Server Programming* ; Wrox Press, 2000

[KJ00] Michael Kircher, and Prashant Jain, *Lookup Pattern*, EuroPLoP 2000 conference, Irsee, Germany

[MH00] Richard Monson-Haefel, *Enterprise Java 豆(Bean)*, Second Edition; Wiley, 2000

[OMGCCM] OMG, *The CCM Specification*, available from http://www.omg.org

[OMGREL] OMG, *Relationship Service Specification*, available from http://www.omg.org

[POSA] Buschmann, Meunier, Rohnert, Sommerlad, Stal; *Pattern-Oriented Software Architecture*; Wiley, 1996

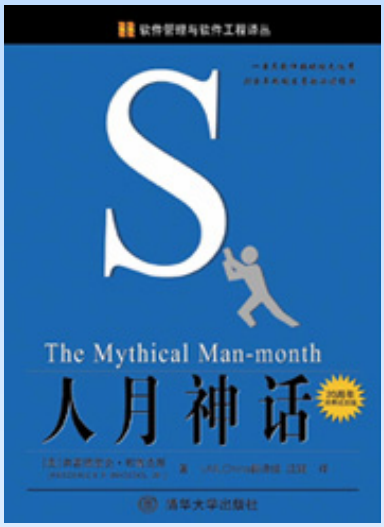
[POSA2] Schmidt, Stal, Rohnert, Buschmann; *Pattern-Oriented Software Architecture, Volume 2*; Wiley, 2000

[PS99] Peter Sommerlad, *Configurability*, EuroPLoP 99 conference, Irsee, Germany

[SUNEJB] Sun Microsystems, *EJB Specification*, available from http://java.sun.com/products/ejb

[Wo00] Eberhard Wolff, *EJB und das Java-Typsystem*, Java Spektrum 06/00

《人月神话》发行了！



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 汪 颖

二十五年后，我们仍然在读这本书

——Ed Yourdon

一种在线拍卖管理的模式语言（下）

Paulo C. Masiero 著，vicky wei 译

查看评论

3 在线拍卖管理的模式语言

3.3 拍卖资源

环境

你的系统用于处理已标识并分类的资源。资源拍卖可视为一种财富交换，通过这种交换，由一方拥有的财富变为由其他方拥有。当通过拍卖来完成资源交易时，*卖方*将资源拿出来卖，其他几家*买方*则期望以最低的价格购得该资源。系统提供给*卖方*各种类型的拍卖方式，至于哪个*买方*将成为赢家，每种方式都有自己独特的判定规则。

问题

你如何处理通过系统执行的各种资源拍卖？

约束

- 拍卖参与者的信息必须保存起来，以便为交易过程和系统功能提供信息。
- 提供多种拍卖方式是很重要的，尤其要注意那些对某些特定类型资源、或待拍卖资源的某种数量、或效率、或某种拍卖类型所提出的约束条件而言最适当的方式，还要考虑到交易发生的环境是Internet。
- 在某些情况下*卖方*可能希望改变关于拍卖的信息甚至拍卖类型，因此有必要为拍卖信息变更建立控制规则以免参与者的利益受损。
- 有必要建立拍卖取消的控制规则以免参与者的利益受损。

结构

图 7 为拍卖资源模式的类图。

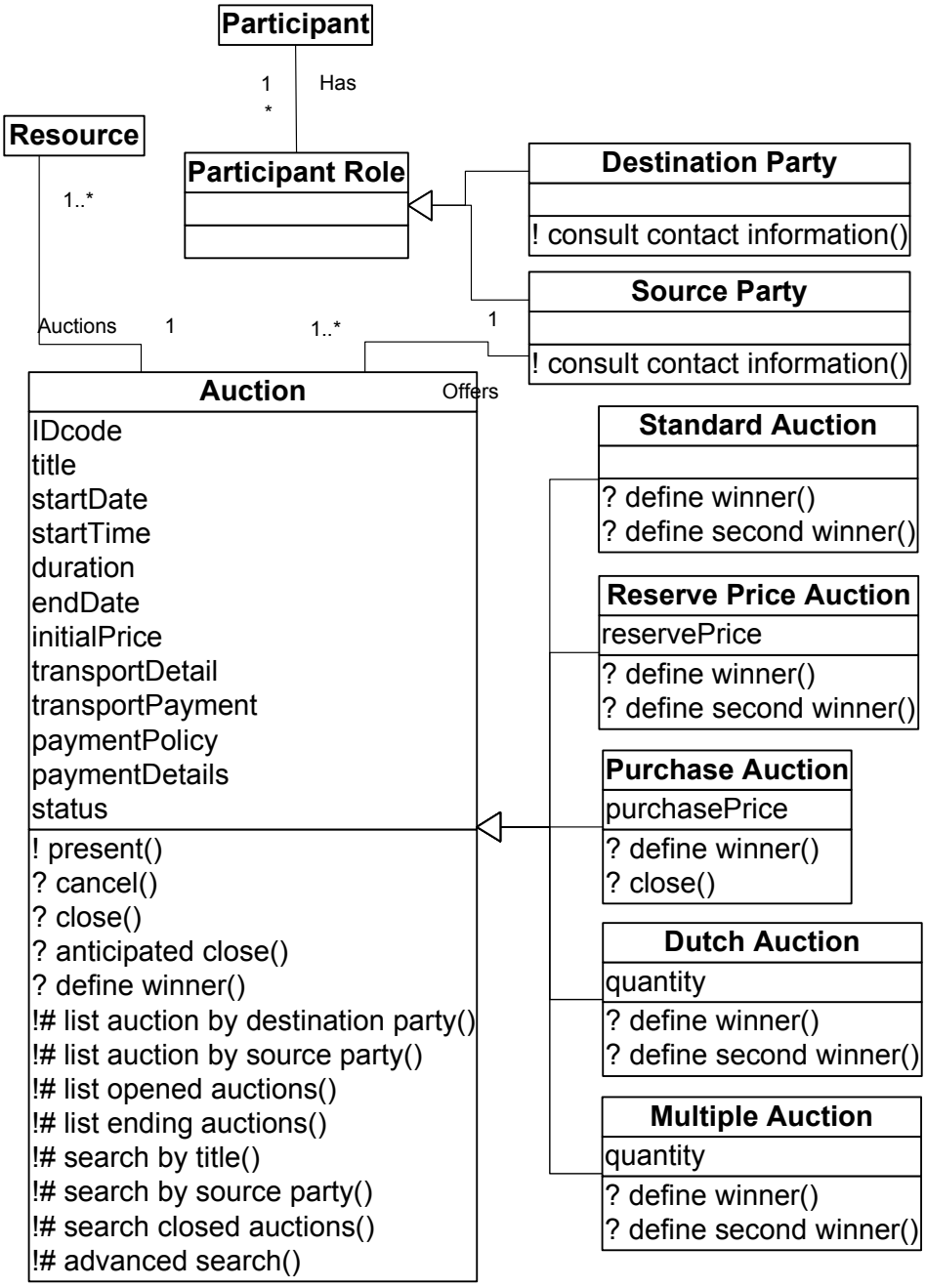


图 7　“拍卖资源”模式

参与者

- *参与者*（Participant）：表示打算拍卖或获取某项资源的相关方（组织或个人）。有两个特例化的*类*：*卖方* Source Party 或*买方* Destination Party（见后文）。注意同一个*参与者*在不同的拍卖中可分别扮演这两种角色，这可通过角色模式[3，7]实现。StatusLogin 属性说明*参与者*是否已提供其 IDCode 和 password，从而决定其是否可参与拍卖活动。注意设计时需按照某种安全策略对密码作特殊的处理[19]。但在此模式语言中我们不考虑这种问题。此*类*有一些基本属性，但也可根据参与者特定实例需要添加其他属性。
- *参与者角色*（Participant Role）：表示在某次特定的拍卖中由*参与者*扮演的角色，可为*卖方*或*买方*（见后文）。
- *卖方*（Source Party）：表示在交易完成前拥有资源的相关方。
- *买方*（Destination Party）：表示竞标并可能在拍卖完成后成为资源所有者的相关方。
- *资源*（Resource）：如识别资源模式中所述。
- *拍卖*（Auction）：表示资源拍卖过程详细细节的抽象*类*。此*类*包含执行一次拍卖所需的所有基本细节信息，与拍卖类型无关。此*类*决定了变更规则的实现，例如何种信息可以变更（此*类*中有的信息）。还有其他一些用于确保变更规则实现的属性，例如拍卖相关信息可以变更几次、拍卖信息可变更的期限、当没有竞标时拍卖是否可发生变更等。这些属性作为拍卖运转的参数，存储在*拍卖系统类*（在管理拍卖系统模式中描述）中。根据特定拍卖实例的细节和功能要求，还可将其他属性添加到此*类*及*拍卖系统类*，这可通过策略模式实现[8]。
- *标准拍卖*（Standard Auction）：这是一种最常见的拍卖方式。在这种拍卖中，多个*买方*为某个*卖方*所提供的*资源*竞标，价高者得之。在拍卖结束并确定了赢家后，联系信息就可提供给买卖双方以供联系（consult contact information）。*拍卖* Auction *类*有一个操作（define second winner），允许*卖方*在第一赢家出现任何问题时也可以第二赢家结束交易（define winner）。第二赢家就是提供次高价的*买方*。这种拍卖类型不允许一次拍卖多个资源项。
- *底价拍卖*（Reserved Price Auction）：在这种拍卖方式中*卖方*除了设定初始价格外还要确定一个底价（reservePrice），其他拍卖参与者事先并不知道底价的确切值。如果竞标的价格低于底价，竞标会被系统接受，但如果拍卖结束时的价格仍低于底价，*卖方*有权不出售该资源项。在这种情况下拍卖结

束，且没有任何赢家，拍卖参与者也不会得到任何联系信息。对底价拍卖而言，达到或超出底价的竞标才可能成为赢家。卖方在拍卖中可降低但不可提高底价。

- *采购拍卖*（Purchase Auction）：在这种拍卖方式中，当竞标价格达到特定的资源采购价（purchasePrice）时，交易自动结束（close）。所以赢家就是出价等同于或高于资源采购价的竞标者。以此*类*表示的拍卖类型不允许一次拍卖多个资源项。这是一个可选*类*，因为系统可能有这种功能需求，也可能没有。
- *荷式拍卖*（Dutch Auction）：这种拍卖类型用于多个同种资源的同时拍卖。*卖方*设定原始价格，多个*买方*竞标并说明所需的数量。系统允许多个竞标者以不同的数量和价格进行竞标。决定赢家的规则是：
  - 赢家是出价最高的那些竞标者。赢家们根据赢家中的最低竞标价付账。这意味着所有赢家最终付账都是一样的。
  - 假如竞标价相同但数量要求不同，则根据竞标顺序决定赢家。输家可以购买剩余的资源。赢家们仍以他们中的最低竞标价付账。

  这是一个可选*类*，因为*多重拍卖*也可用于对多个资源项的拍卖。
- *多重拍卖*（Multiple Auction）：这种拍卖方式和*标准拍卖*类似，不同的是它允许一次拍卖多个同种资源。所以多个*买方*可能同时获取一定数量的资源项。所有的竞标根据价格、数量、时间排序——首先高投标价赢过低投标价；假如竞标价相同，赢家就是需求量最大者；假如价格和需求量都一样，赢家就是第一个竞标者。这也是一个可选*类*，因为某些拍卖系统使用*荷式拍卖*来处理多项资源的拍卖。

范例

图 8 就是在线拍卖网站 Arremate.com 所采用的拍卖资源模式的范例，它采用*多重拍卖* Multiple Auction 方式拍卖单个或多个资源项。Arremate.com 还使用了其他两种拍卖方式：*底价拍卖* Reserve Price Auction 和赢家拍卖 Winner Auction（是*采购拍卖* Purchase Auction 的实例）。eBay 在多项产品的交易中使用*荷式拍卖* Dutch Auction，在单项产品的交易中则使用*标准拍卖* Standard Auction。它还提供*底价拍卖* Reserve Price Auction 和*采购拍卖* Purchase Auction 方式。iBazar 只提供*标准拍卖* Standard Auction 和*底价拍卖* Reserve Price Auction 两种方式。

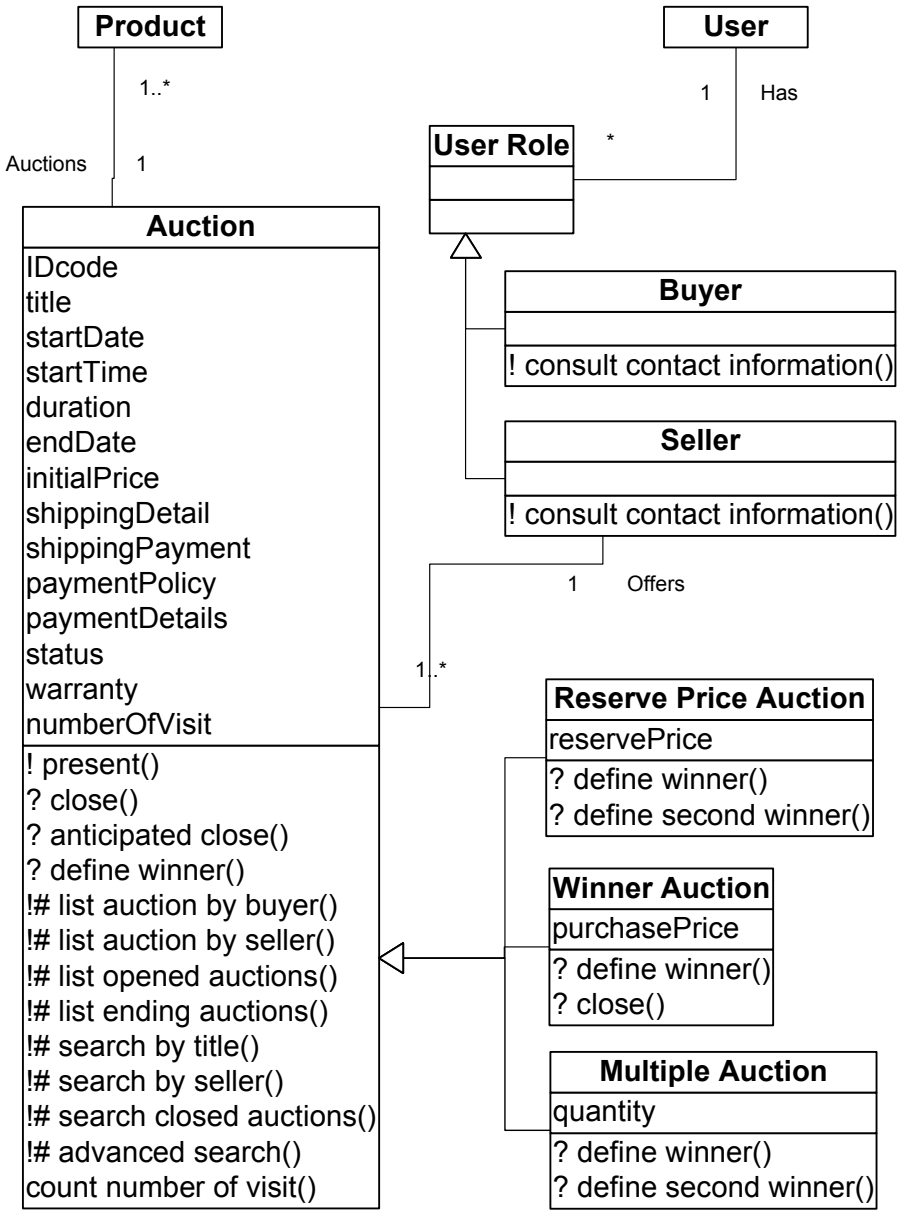


图 8　“拍卖资源”模式范例

变体

为使模式语言适用于不同拍卖类型，可增加表示拍卖规则的新*类*，作为 auction 类的特例（specialization）。这些新的拍卖方式可能需要新的属性和方法。

在某些系统中*标准拍卖*被*多重拍卖*取代。在这种情况下，*多重拍卖*同时用于单项或多项资源项的交易。

为保护系统不被客户乱用，某些拍卖系统会要求用户提供某种身份识别文件（如身份证或驾照），某些拍卖系统则要求客户用信用卡进行注册，而有些则不要求用户提供任何特定个人信息。

某些拍卖系统允许交易双方中的一方取消拍卖，而其它系统则不允许。拍卖的取消必须遵循一些规则，例如只允许在拍卖结束后某个预定时间段内，或根据某些预先设定的原因，或只要因为*卖方*希望就可以取消。类似地，哪些拍卖数据可修改也有必要制订规则。除了拍卖取消中允许的情况外，某些系统可能只允许*卖方*增加信息而不能删除或改变已经提供的信息。取消和更改规则的参数存储在管理拍卖系统模式的*拍卖系统类*中。

相关模式

本模式是模式 ASSOCIATION-OBJECT[1]和 TIME ASSOCIATION[3]的实际应用。它同时也是模式 SPECIFIC ITEM-TRANSACTION 和 PARTICIPANT-TRANSACTION 的一个综合应用。

后续模式

在应用拍卖资源模式后，尝试使用提交协商模式。假如不适用，则直接使用处理竞标模式。

3.4 提交协商

环境

你的系统可处理那些已识别并分类的资源，这些资源可通过已定义的拍卖类型进行交易。但此时客户通常还希望能澄清一些关于这些资源的疑问，或关于他们想参与竞标的拍卖活动本身的疑问。问题直接由*卖方*作出回答。交易双方间的这种沟通类似于传统贸易活动，在拍卖参与方之间提供直接通道，从而产生一种有利于拍卖活动的竞争环境。

问题

错误！未找到引用源。

约束

- 为问题的提出和解答建立规则，对澄清待拍卖资源的细节信息很有帮助。此外假如问题和答案都记录在系统中，则所有用户（而非提问者本人一人）都能从答案中获益。
- 在交易方之间采用讨论列表来沟通有助于答疑，但这种沟通方式在所提出的问题和特定拍卖之间并不存在明显的对应关系。因此要了解关于某次特定拍卖的问题或答案信息，就必须在讨论列表中仔细阅读多条消息。

- *卖方*对问题的回答可能带来更多的竞标机遇，因此建立鼓励*卖方*回答问题的规则很有必要。关于资源和拍卖的细节信息越多，交易的机会就越多。此外对问题的回答也能展示*卖方*在出售资源中的真正目的。激励方式可以是*交易费*折扣，或其它方式的拍卖鼓励政策。

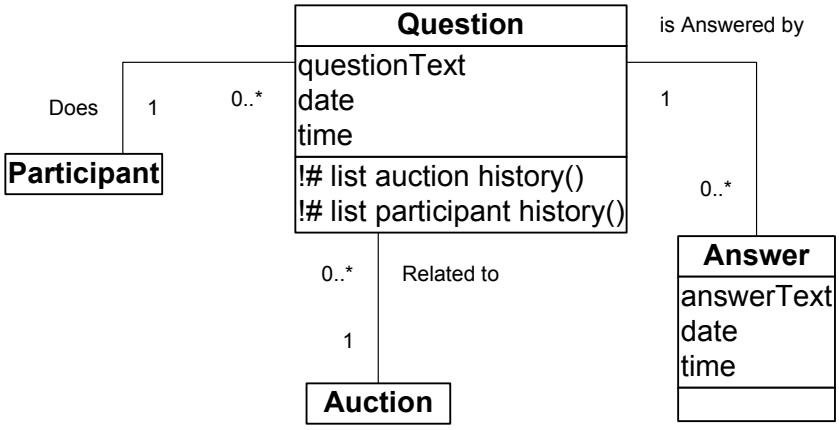


图 9　“提交协商”模式

结构

图 9 为提交协商模式的类图。

参与者

- *参与者*（Participant）：如拍卖资源模式中所述。
- *拍卖*（Auction）：如拍卖资源模式中所述。
- *问题*（Question）：表示由*参与者*向*卖方*提的关于*拍卖*的问题。这些问题通常是*买方*对于拍卖细节信息的疑问。
- *回答*（Answer）：表示*卖方*对*买方*提出的问题给出的解答。

范例

图 10 给出了在线拍卖网站 Arremate.com 所使用的提交协商模式。eBay 和 iBazar 使用同样的方式在交易方之间提供沟通渠道。

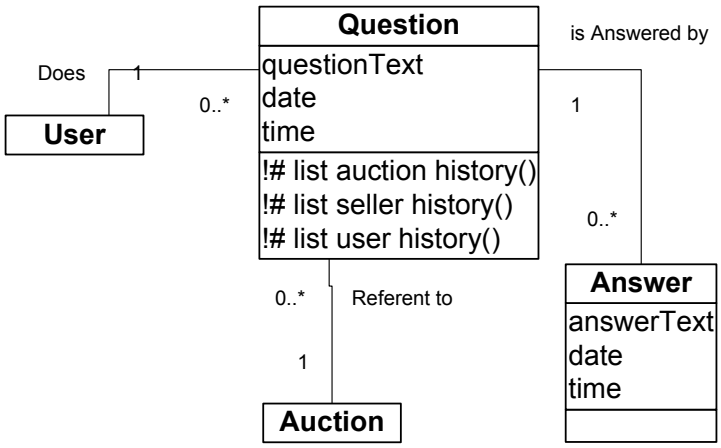


图 10 “提交协商”模式范例

后续模式

在使用提交协商模式后，使用处理竞标模式。

3.5 处理竞标

环境

你的系统处理各种类型的资源拍卖，为交易提供一个框架。在一次拍卖中，资源通过竞标获得。这些竞标和因拍卖类型而异的拍卖终结规则一起，可决定拍卖的赢家（可以是一个或多个）。竞标需遵循拍卖类型附带的规则，例如可拍的数量、竞价时的价格增量等。*买方*可拥有自己的“拍卖关注清单”，以便方便地访问此清单中的拍卖并关注其进展。

问题

你的系统如何处理各拍卖的各种竞标？

约束

- 竞标信息可用于判定拍卖赢家，因此信息的存储非常重要。假如拍卖类型允许同时存在多个赢家，则所保存的相应竞标信息将用于打破平衡。
- 必须确保由拍卖类型决定的竞标规则在竞标中得以遵循。可能存在拥有特殊竞标方式的拍卖。
- 有必要为竞标的取消建立规则，以保护拍卖参与者不受伤害。这些规则必须同时支持由*卖方*或*买方*发起的竞标取消。

结构

图 11 为处理竞标模式的类图。

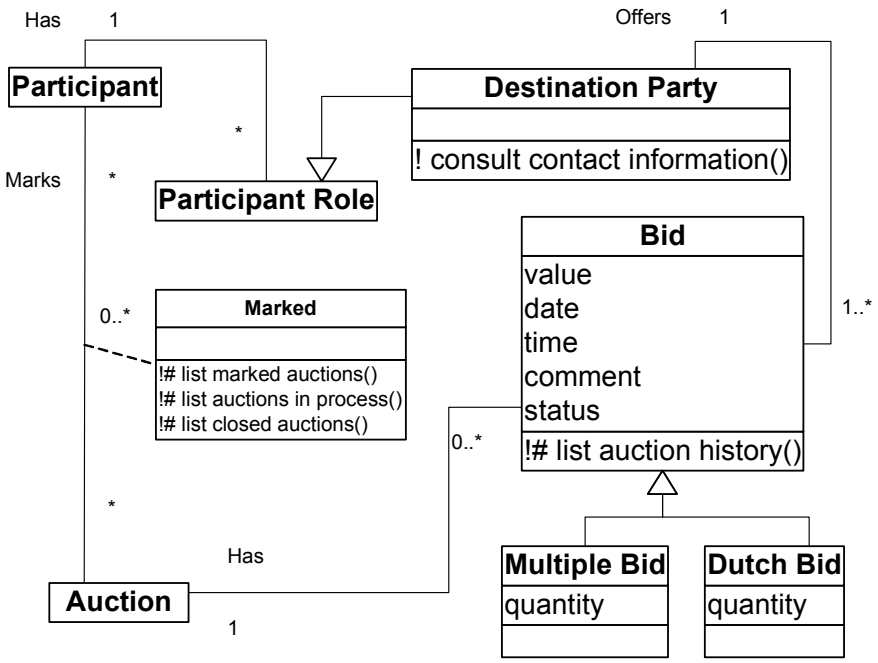


图 11　“处理竞标”模式

参与者

- *参与者*（Participant）：参见拍卖资源模式。
- *参与者角色*（Participant Role）：参见拍卖资源模式。
- *买方*（Destination Party）：参见拍卖资源模式。
- *拍卖*（Auction）：参见拍卖资源模式。
- *竞标*（Bid）：表示在拍卖中获得的投标细节信息。此*类*有一些基本属性，但根据拍卖系统的特定功能也可增加新的属性。这可通过如属性模式的方法实现[5]。
- *荷式竞标*（Dutch Bid）：表示对*荷式拍卖*做出的竞标。因此这是一个可选*类*，仅用于*荷式拍卖*。
- *多重竞标*（Multiple Bids）：表示对*多重拍卖*做出的竞标。因此这是一个可选*类*，仅用于*多重拍卖*。
- *已标记的*（Marked）：表示*买方*特地对某些拍卖做出标记，以便更好地关注其竞标状况并方便地访问这些拍卖。因此*买方*可以轻而易举地找到更感兴趣的那些拍卖。

范例

图 12 给出了处理竞标模式的一个范例，是在线拍卖网站 iBazar 所使用的模式。Arremate.com 和 eBay 也都提供了标记拍卖的功能。iBazar 不支持对竞标的取消。尽管 iBazar 支持多种拍卖方式（包括*底价拍卖*），但这些拍卖方式都不能支持多项资源的拍卖。因此能保存普通竞标数据的*竞标类* Bid 已足以支持这些只拍卖一项资源项的拍卖方式。Arremate.com 和 iBazar 类似，但支持*多重拍卖* Multiple Auction（对多项资源项的拍卖）。因此它需要使用*多重竞标类* Multiple Bids。此外 Arremate.com 还实现了增量加价规则，并提供一种自动竞价服务（细节参见**变体**部分）。eBay 也实现了增量加价规则，同时还支持竞标取消、多重资源项拍卖（通过*荷式拍卖* Dutch Auction）等功能。此外，eBay 还提供只有选定的*买方*才能参与竞标的功能。

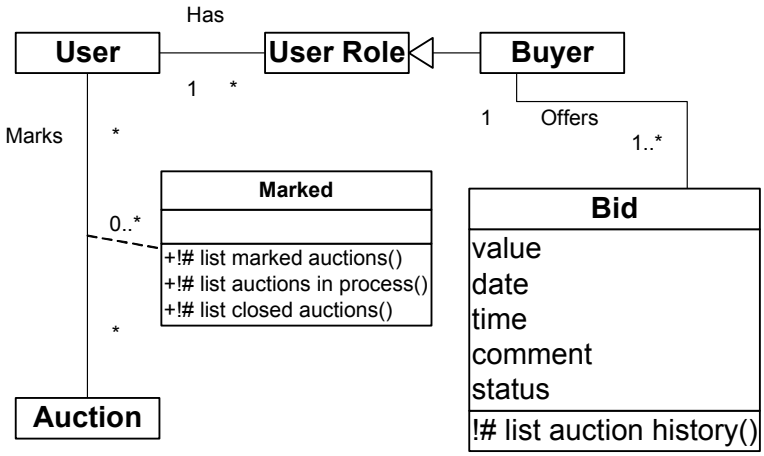


图 12 “处理竞标”模式范例

变体

如同拍卖资源模式中所述，系统可添加多种拍卖方式，并提供相应的竞标方式以支持这些拍卖。假如某种拍卖方式通过对 Auction 类的特殊化（specialization）实现添加，且该拍卖需要某种特定的竞标方式的支持，则该竞标方式就必须通过对 Bid 类的特殊化得以添加。

除了从可能的竞标方式产生变体外，这种模式还可有其他方式的变体。图 13 展示了 Proxy Bidder *类*，即自动竞价功能。因此系统可根据买方设置的一些参数自动地完成竞价功能，从而使交易变得更简单。此功能由买方执行 enable proxy bidder 操作得以实现，若买方选择了自动竞价选项，则在任何其他买方给出一个更高价格时，系统负责自动投出一个新的竞价。例如，系统会立即报出一个超出前一竞价的价格，当然此价格不会超出由买方设定的上限。自动竞价操作的上限由 maximum BidAmount 属性的值决定。

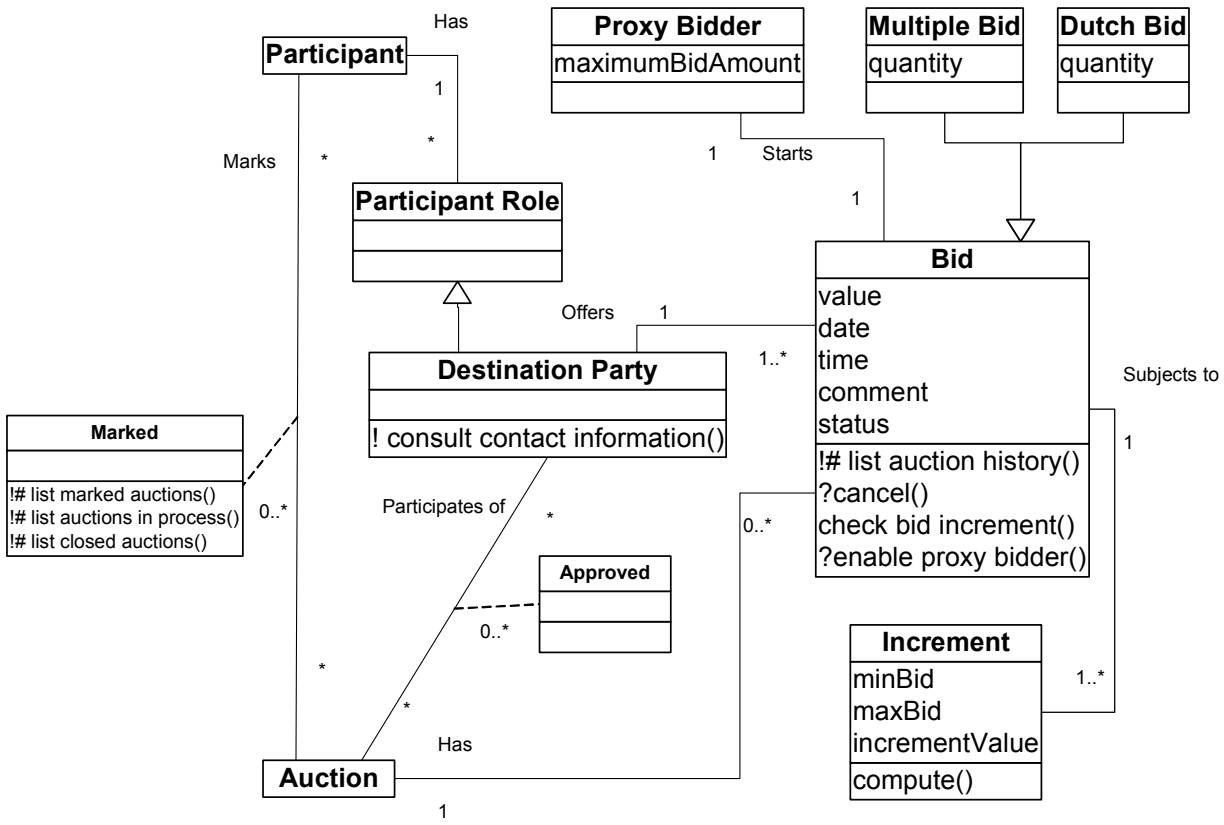


图 13 “处理竞标”模式变体

另一种变体即设置增量竞价规则，这在图 13 中也可看到。此功能通过 check bid increment 操作实现，用于当你希望竞价对总价占一定比例的情况。此选项还会影响到自动竞价功能，因为自动竞价功能需遵循竞价增量规则。因此，自动竞价功能会自动给出一个新的报价，该价格刚好大于前一竞价，同时满足增量规则，且未超出价格上限。

某些拍卖可只允许由*卖方*预先选定的一组*买方*参与竞价。例如，假如某类资源有特定类型的购买者，则*卖方*可选择此选项。图 13 中的 Approved *类*表示能参与某次拍卖活动的所有*买方*。因此，只有预先确定的那些*买方*能在该拍卖中竞价。

某些拍卖系统还可以取消竞价，但不同系统有不同规则。这些规则控制了谁能取消竞价，能取消竞价的时间段，是否有预先规定的取消原因，以及是否有必要解释这些原因等。用于此功能而新增的属性和方法必须通过*类* Bid 和 Auction House 实现。

后续模式

在使用了处理竞标模式后，使用管理拍卖系统模式。

3.6 管理拍卖系统

环境

你的系统管理由*卖方*提供的待拍卖资源，而多个*买方*将竞标以求获取这些资源。这就要求*拍卖系统*建立一些规则来调停交易过程。*拍卖系统*根据系统使用和资源交易情况决定是否收费。此外，*拍卖系统类*需确定引导整个拍卖交易流程的通用参数。

问题

你的系统如何管理参与拍卖过程的拍卖系统所需遵循的规则？

约束

- 当通过拍卖完成交易时，*拍卖系统*的相关信息需要得到保存。
- 需要为*拍卖系统*定义一些基本规则以实现其功能。
- 需要确定交易双方需支付哪些费用以及如何支付。
- 多种付费方式尽管可提供更大的方便，但需要更多的存储空间和处理时间。

结构

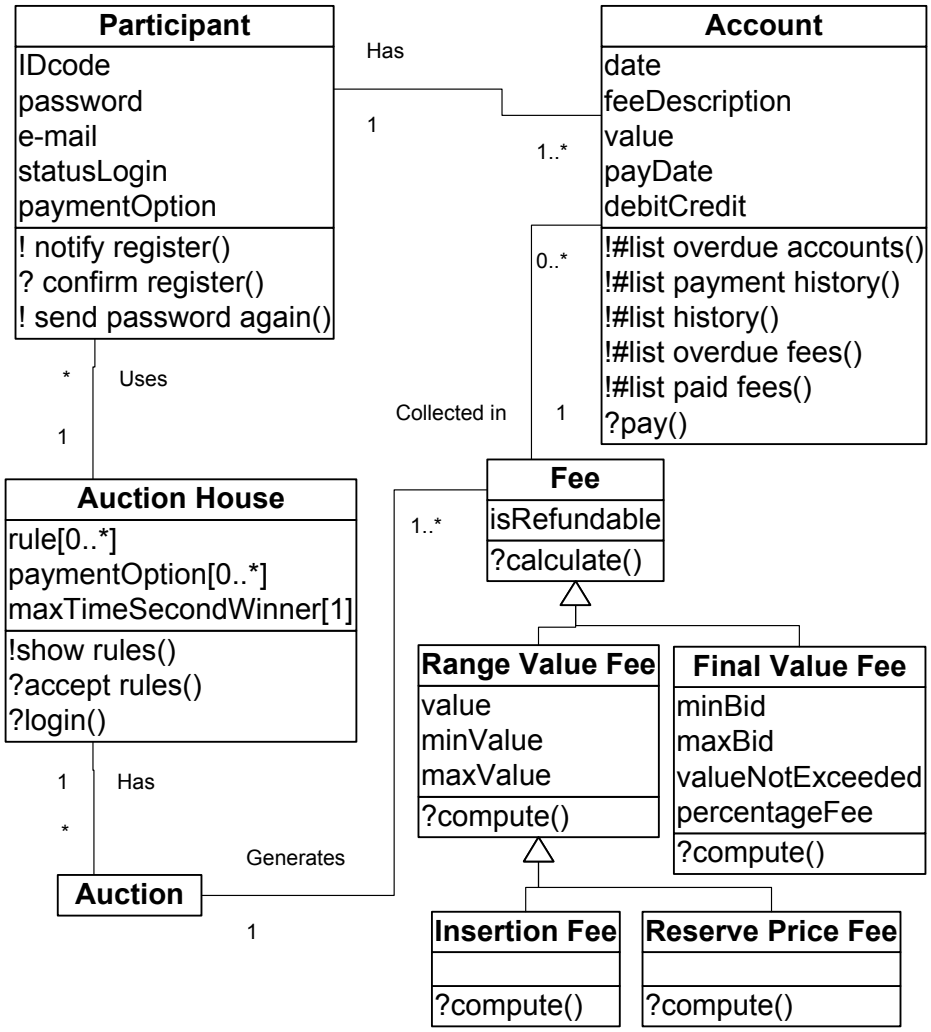


图 14 “管理拍卖系统”模式

参与者

- *拍卖*（Auction）：如拍卖资源模式中所述。
- *拍卖系统*（Auction House）：表示交易系统/交易平台自身，例如可介入交易活动的*拍卖系统*。此类存储了一些会影响到整个拍卖流程的参数，如*卖方*可申请索要第二赢家联系信息的截止日期（maxTimeSecondWinner）。属性 rule 存储了通用行为规范，即通用于*卖方*、*买方*、*拍卖系统*的行为规范——*拍卖系统*所提供服务的法律和合约方面的问题由此属性负责处理。*卖方*和*买方*必须先认可这些规范（通过 show rules 和 accept rules），然后才能使用拍卖系统。

- *交易费*（Fee）：这是一个抽象*类*，表示可由*拍卖系统*收取的交易费用。属性 refundable 指明了此项费用是否可被退换给*参与者*。此类及其所有实例都是可选的，因为拍卖系统可能收费也可能不收费。
- *成交价交易费（Final Value Fee）*：表示按最终竞标价（如成功结束某次拍卖的竞价）的某个比例收取*交易费*。
- *范围交易费（Range Value Fee）*：这是一个抽象*类*，表示以某个取值范围（minValue，maxValue）和某个固定值（value）为基础计算*交易费*。
- *登陆交易费（Insertion Fee）*：表示在资源的起价（initialPrice）基础上计算*交易费*，但不超出某个范围。当任何资源加入到某次拍卖中时都可按这种方式收费。按这种方式收取的费用通常都不能退费。
- *底价交易费（Reserve Value Fee）*：表示在资源的底价（reservePrice）基础上计算*交易费*，但不超出某个范围。当拍卖为*底价拍卖*时，可按这种方式收费。采用按这种方式收取费用时，假如交易未成功，通常可退费。
- *账目（Account）*：表示对某个*参与者*所收取的费用以及可能的退款。因此，数值（value）既可能表示*参与者*的信用量，也可能表示他的借记金额（debitCredit）。此*类*和*交易费类* Fee 密切相关，通常仅当 Fee *类*也适用时才使用此*类*。
- *参与者（Participant）*：如拍卖资源模式中所述。

范例

图 15 给出了由在线拍卖网站 Arremate.com 使用的管理拍卖系统模式的范例。Arremate.com 只对成功交易收取*交易费*，此费用基于最终成交价计算（Success Fee）。系统实现的*交易费*支付规则，只有在顾客的*账目*达到某个阀值时，才会向顾客发送警告，通知他的应付账款。而 eBay 除成*交价交易费*（Final Value Fee）外，还实现了其它的收费方式，如*底价交易费*（Reserve Price Fee）和*登陆交易费*（Insertion Fee）。当拖延支付*交易费*超过某个预先设定的时间段时，顾客对系统的使用权会被封锁。iBazar 则不收取任何方式的*交易费*。

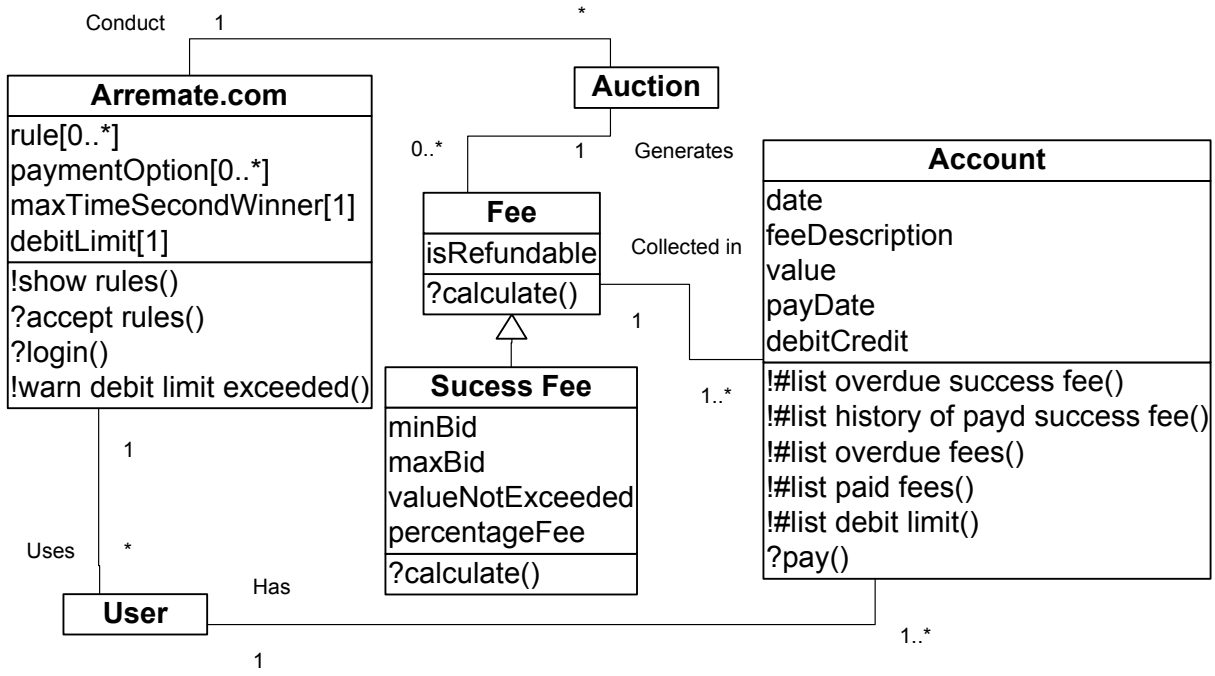


图 15 “管理拍卖系统”模式范例

变体

某些拍卖系统和能完成资源递送及账单收发工作的结构有业务上的联合。这些机构的信息也可存储在*拍卖系统*（Auction House）*类*中。

某些系统提供*交易费*折扣或免收*交易费*，这取决于提供交易系统的机构所建立的市场策略。例如提供拍卖平台的机构规定假如*卖方*多次拿出资源让利销售，或在某一段时间内，或在资源被卖掉前（即交易成功前），免收*登陆交易费*，以实现某种市场策略。一些其他附加参数也可存储在*拍卖系统*（Auction House）*类*中，如时间周期、次数、*交易费*免收、*交易费*折扣等。

相关模式

此模式是 Participant-Transaction 模式的应用[4]。

后续模式

在使用了管理拍卖系统模式后，使用管理信用模式。

3.7 管理信用

环境

你的系统处理资源拍卖。*卖方*提供资源，几个*买方*通过*竞标*来争夺资源。当确定了赢家后交易就发生了，拍卖也就随之结束。拍卖结束后，*卖方*和*买方*可就交易情况互相评价。这些信用评价有助于保护交易双方的利益。此外系统里也可随时找到所有交易方的信用评价。这是交易者找到有信誉值得信赖的交易方的一种好办法。

问题

你的系统如何支持对相关方可信度的确认？

约束

- 只有参与了交易的相关方才能对其他交易方做出评价，并提供可靠的信息。有些拍卖最终会有多个*买方*成为赢家，此种情况必须额外实现特定规则。
- 对个人隐私数据保密非常重要，这不管是对系统的所有者和还是用户而言都如此。
- 为交易双方提供辩解功能，并将涉及的相关方的所有争执保留下来，对保护客户的利益而言很重要。[16]

结构

图 16 是管理信用模式的类图。

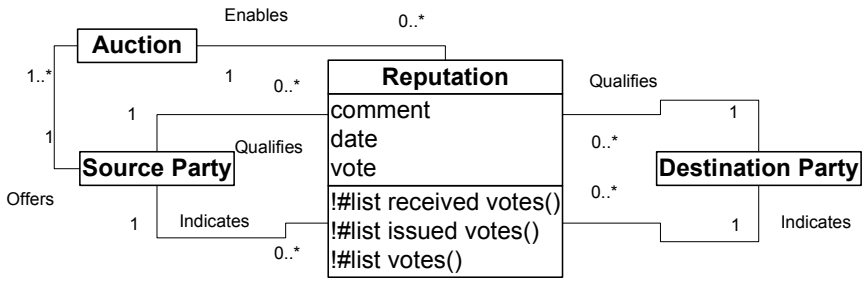


图 16 “管理信用”模式

参与者

- *卖方*（Source Party）：如拍卖资源模式中所述。
- *拍卖*（Auction）：如拍卖资源模式中所述。

- *买方*（Destination Party）：如拍卖资源模式中所述。
- *信用*（Reputation）：表示拍卖结束后，*卖方*对*买方*（或反过来，*买方*对*卖方*）做出的评价。

范例

图 17 是在线拍卖系统 iBazar 使用的管理信用模式范例。IBazar、eBay 和 Arremate.com 都允许交易双方相互做出评价。其他顾客可浏览这些评价。只有 eBay 提供了“是否公开信用评价清单”的选项，因此顾客可能可看到所有的信用评价，也可能只能看到自己做出的评价。

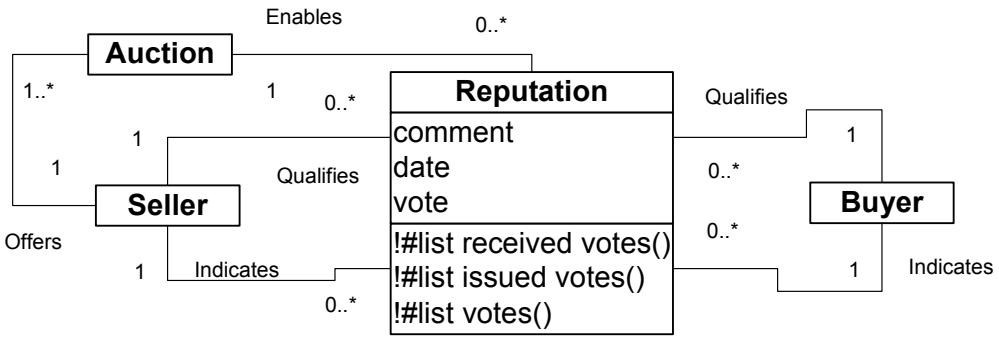


图 17 “管理信用”模式范例

变体

信用评价功能的一种变体就是是否公开信用评价——任何人都能看到，还是只有给出评价的人可以看到？此选项使得顾客可灵活控制是公开自己的评价意见给所有人共享，还是出于某种原因希望保密。大多数系统都鼓励公开评价结果，以便为所有使用拍卖系统的顾客创造一个更安全的交易环境。

在某些拍卖系统中，信用评价可以作出更改，但受到某些规则的限制。不同系统有不同的规则，如更改只能做一次，只能在预定期限内做，可随时做等。完成这些规则所需的参数，如允许的变更次数、变更期限等，都必须存储在拍卖资源模式中所提到的*拍卖系统*（Auction House）*类*中。其他参数，如评价是否已更改，就必须存储在*信用*（Reputation）*类*中。

后续模式

假如在使用管理拍卖系统模式时，使用了 Fee 类或其任何实例，则使用处理退款模式。否则请尝试使用提供消息功能模式，若不适用，则使用管理广告模式。

3.8 处理退款

环境

你的系统处理资源拍卖，*拍卖系统*作为交易平台的提供者，对交易收取*交易费*。在某些情况下，如因为*买方*放弃的原因导致交易实际上未成功时，*交易费*应可获得退款。这就要求*拍卖系统*提供一种机制，以便将已支付的*交易费*返还给*卖方*。从另一个角度来说，受控方（如*买方*）也应有权为起诉方（如*卖方*）的退款请求做出辩解。因此根据交易双方提出的退款请求和相应申诉，*拍卖系统*最终可能退回*交易费*，也可能不退。在做出决定后，*拍卖系统*应说明退款请求如何继续处理——是将*交易费*返还给*卖方*，并对*买方*做出负面信用评价；还是*交易费*不但不得返还，还要对*卖方*做出负面的信用评价。

问题

对于不恰当的收费，你的系统提供何种退款功能？

约束

- 存储来自交易一方的退款请求和来自另一方的申诉，以便*拍卖系统*做出决策。
- 有必要根据拍卖类型和业务规则退回*交易费*。
- 允许对交易双方作出信用评价是对竞争关系的拍卖者的一种保护。

结构

图 18 是处理退款模式的类图。

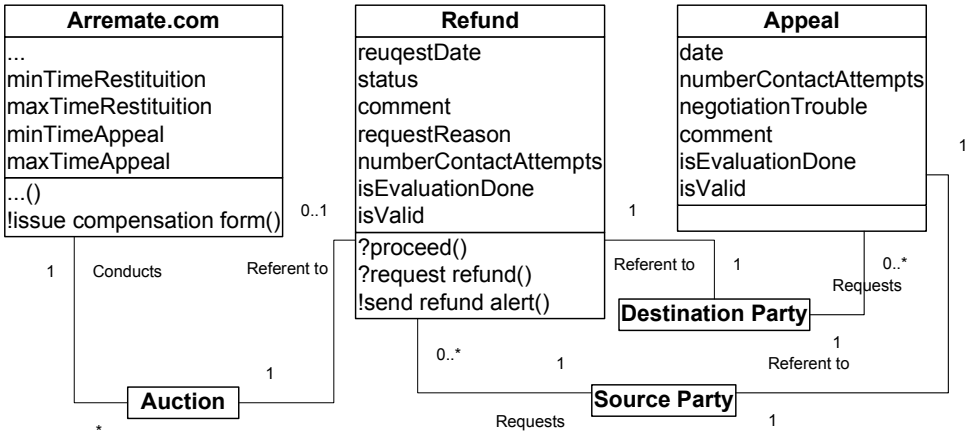


图 18 “处理退款”模式

参与者

- *拍卖*（Auction）：如拍卖资源模式中所述。
- *卖方*（Source Party）：如拍卖资源模式中所述。
- *买方*（Destination Party）：如拍卖资源模式中所述。
- 退款（Refund）：表示由*卖方*提出的退款请求，为某项未成功交易的资源拍卖，请求退回*交易费*。即便拍卖已结束并确定赢家，只要交易未成功，就可提出此请求。然后*卖方*就能请求退回*交易费*。属性 isEvaluationDone 说明*卖方*是否已经给*买方*做出信用评价。
- *申诉*（Appeal）：表示对应*卖方*提出的退款请求，*买方*做出的申诉。每当提出退款请求时，受控方有权为自己辩护，解释交易未能完成的原因。属性 isEvaluationDone 说明*买方*是否已经给*卖方*做出信用评价。
- *拍卖系统*（Auction House）：如管理拍卖系统模式中所述。

范例

图 19 是在线拍卖网站 Arremate.com 使用的处理退款模式范例。当*卖方*和*买方*不能完成交易时，在某个时间段内，*卖方*可为已支付*交易费*但未成功完成的交易请求退费。系统会向*买方*发送一个警告信息，将*卖方*的投诉通知到它。*买方*应在指定期限内做出回应，填写申诉表。*拍卖系统*负责对纠纷做出判决，并决定是否退费。纠纷的结果可能是*买方*（或*卖方*）收到*拍卖系统*的负面信用评价。eBay 的流程几乎雷同。它接受退款请求，然后等待*买方*做出答复，希望交易双方可相互谅解。如果调解失败则会做出判决，然后按照类似 Arremate.com 的流程公布判决结果。iBazar 是免费的，所以不存在退费问题。

相关模式

此模式是模式 ASSOCIATION-OBJECT[1]、TIME ASSOCIATION[3]、TRANSACTION-SUBSEQUENT TRANSACTION 和 PARTICIPANT-TRANSACTION[4]的应用。

后续模式

在使用处理退款模式后，请尝试使用提供消息功能模式，若不适用，则使用管理广告模式。

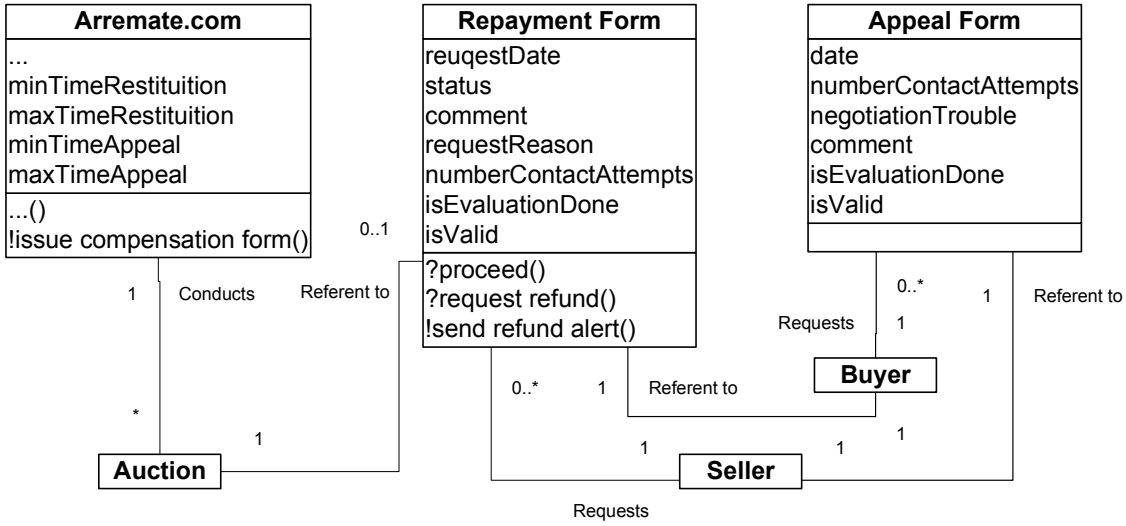


图 19 “处理退款”模式范例

3.9 提供消息功能

环境

你的系统处理通过拍卖进行的资源交易。在这种类型的交易中，会发生一些事件，如拍卖开始、竞价、拍卖结束、确定赢家等。大部分这种事件必须通过消息通报给客户，以便他们监控所有的拍卖阶段。除了和拍卖直接相关的消息外，也可能存在出于其他目的的消息，如广告、拍卖中发生的变化、促销等。

问题

你的系统怎样管理传递给顾客的消息？

约束

- 保存曾发送给顾客的消息，有助于向他们提供拍卖事件历史。此外就顾客所参与的拍卖而言，此消息记录也是做决策时的重要参考信息；
- 有必要让顾客自己设置消息接收规则，从而可自行决定哪些消息对他们而言是重要的和必要的；
- 消息是系统和顾客直接沟通的渠道。*拍卖系统*可将之用作简单的沟通渠道，也可用于发布推销信息。因此消息也可成为一种服务于市场策略的工具。

结构

图 20 是提供消息功能模式的类图。

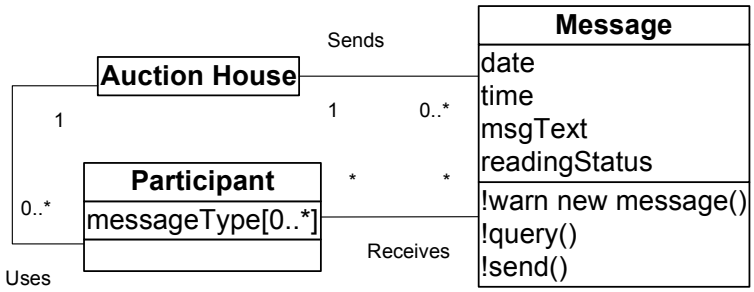


图 20　“提供消息功能”模式

参与者

- *参与者*（Participant）：表示打算拍卖或获取某项资源的相关方（组织或个人），即*卖方*或*买方*。*参与者*选择他们愿意接收的消息，选择结果由此*类*存储（messageType）。这些消息选项可包括拍卖中发生的各种事件，如拍卖启动或结束、谁是拍卖赢家、有什么新的竞标、由其他参与者竞出的最高价格等。
- *拍卖系统*（Auction House）：如管理拍卖系统模式中所述。
- *消息*（Message）：表示所存储并发送给*参与者*的消息。这些消息随时可通过系统或邮件查阅。

范例

图 21 是在线拍卖网站 iBazar 所使用提供消息功能模式范例。此范例不提供消息类型的配置选项，因此顾客不能自己选择愿意接收的消息。所有的消息都保存起来，并只有在顾客主动删除时才会被删除。顾客在登陆到系统时，会收到新消息通知。eBay 和 Arremate.com 都提供了消息类型的配置选项。Arremate.com 不保留顾客的历史消息，消息只通过邮件发送。

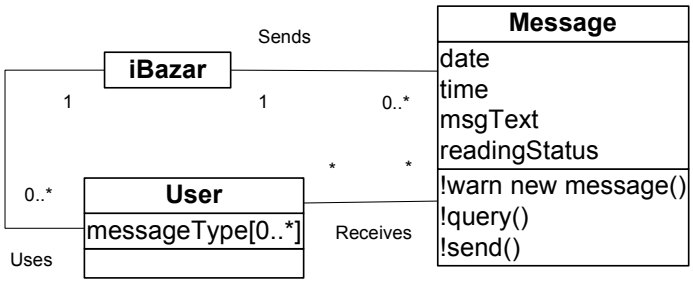


图 21　“提供消息功能”模式范例

相关模式

此模式是模式 ASSOCIATION-OBJECT[1]和 TIME ASSOCIATION[3]的应用。

后续模式

在使用提供消息功能模式后，使用管理广告模式。

3.10 管理广告

环境

你的系统处理资源的拍卖。*拍卖系统*提供了一些方法来拍卖资源。目的是促进并通报拍卖活动，以吸引可能对获取资源感兴趣的*买方*。*买方*在由*拍卖系统*提供的选项中，选择最适合待销售资源的广告类型。因此，根据所选择的广告类型，拍卖以不同的侧重点和呈现特性得以通告。

问题

你的系统如何管理拍卖广告？

约束

- 建立广告规则，对各种拍卖类型进行分类，对促进资源交易和*拍卖系统*的业务很重要。发布拍卖通告对业务很有好处——吸引新客户的可能性越大，交易成交量就越大。
- 重要的是为对顾客有较大影响的拍卖提供一种广告方针，以建立一种广告策略来促进其他拍卖。

结构

图 22 是管理广告模式的类图。

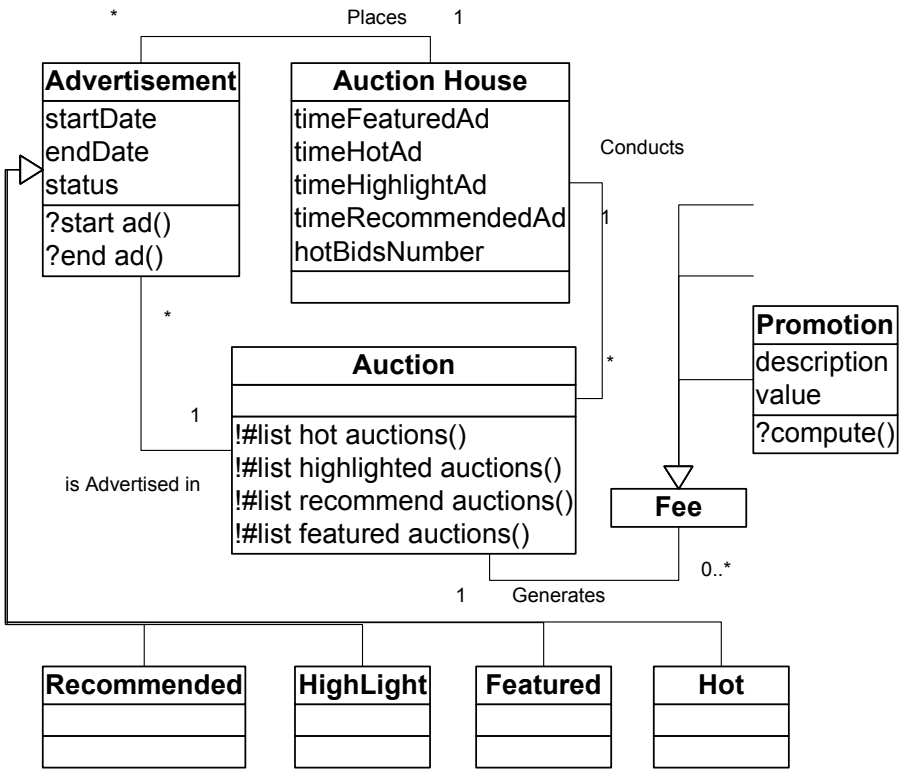


图 22 “管理广告”模式

参与者

- *拍卖系统（Auction House）*：如管理拍卖系统模式中所述。
- *拍卖（Auction）*：如拍卖资源模式中所述。
- *广告（Advertisement）*：表示拍卖的宣传方式。此*类*存储广告规则及能有效地宣传这些广告的数据。广告规则的例子有，*卖方*必须回答所有关于拍卖的问题，顾客不能做出负面信用评价，拍卖必须选择某种类型的促销方式，拍卖必须是某些拍卖类型（如*底价拍卖*、*荷式拍卖*等）中的一种等等，或仅仅为了引起客户的注意。这些规则可通过此*类*的各种特例实现。
- *推荐的（Recommended）*：表示被系统从符合*拍卖系统*业务规则的候选项中选出的拍卖，将被发布在网站最重要的区域。
- *强调的（Highlight）*：表示被系统选中的拍卖，将被发布在网站中某个显著的区域。
- *有特性的（Featured）*：表示在网站页面中易于寻找的拍卖。他们会被发布在主页、查询页面、或分类浏览中。

- *热卖中（Hot）*：表示拥有相当数量竞标的拍卖，以此引起*买方*的注意。拍卖被认定为“*热卖中*”所需具备的竞标数由*拍卖系统类*的属性 hotBidsNumber 确定。*热卖中*的拍卖通常发布在网站主页中某个显著区域。

- *促销（Promotion）*：表示为拍卖提供的各种简单呈现格式（黑体、在清单旁提供图片、置于清单内某个显著区域等）。这些格式和费用直接相关。因此，此*类*不仅可被当作简单方式的广告使用，还可被用作这些方式对应的*交易费*。这些促销方式可被用于确定更细致的广告策略（*有特性的*、*强调的*、*推荐的*）。例如，可规定“拍卖只有在有某种*促销*时才能被称为*有特性的*”。

- *交易费（Fee）*：如管理拍卖系统模式中所述。

范例

图 23 是在线拍卖网站 eBay 所使用的管理广告模式范例。eBay 有两种特定类型的广告：*有特性的*、*热卖中*。“*有特性的*”广告方式要求拍卖有某种特定的*促销*（如，在主页中发布）。*拍卖系统*决定拍卖广告的时间长度以及拍卖的发布顺序。Arremate.com 有三种广告类型，并实现了自己特定的广告规则。尽管如此，它的工作方式还是类似于 eBay。iBazar 则不区分拍卖的广告，它使用*广告*（Advertisement）*类*为所有的拍卖发布通告。*广告*的具体工作方式则类似于 eBay 和 Arremate.com。

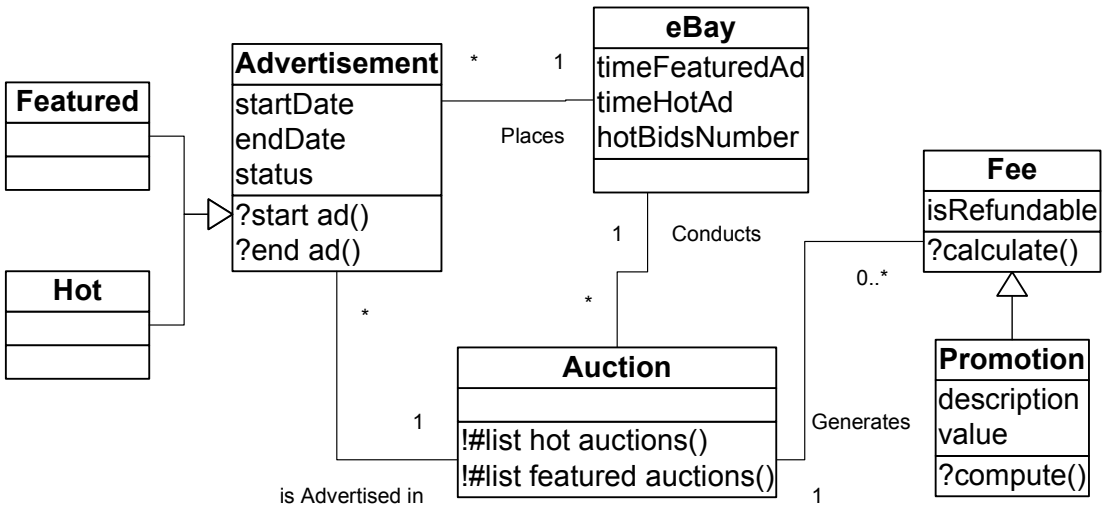


图 23 “管理广告”模式范例

相关模式

此模式是模式 ASSOCIATION-OBJECT[1]和 TIME ASSOCIATION[3]的应用。

后续模式

至此系统分析结束。

4 模式语言应用范例

图 24 展示了应用在线拍卖管理模式语言得出的类模型。标签显示了所指向的类所扮演的角色。标签包含了类似“P#n：role”的标注，其中“n”是模式号，而“role”则指明了该类在模式中所扮演的角色[17]。它被用于满足在线拍卖网站 iBazar 的拍卖过程需求[15]。通过对界面的逆向工程可获得 iBazar 的功能集，而这些功能可通过对以下模式的应用得以实现：(1)识别资源；(3)拍卖资源；(4)提交协商；(5)处理竞标；(6)管理拍卖系统；(7)管理信用；(9)提供消息功能；(10)管理广告。iBazar 没有“客户偏好的资源分类”的概念，因此不需要应用设置个人喜好模式。处理退款模式也不需应用——iBazar 是免费的，因此不需要退款功能。有趣的是，iBazar 不提供多项资源项的拍卖，并将拍卖限制在某些资源类型中。

征稿

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

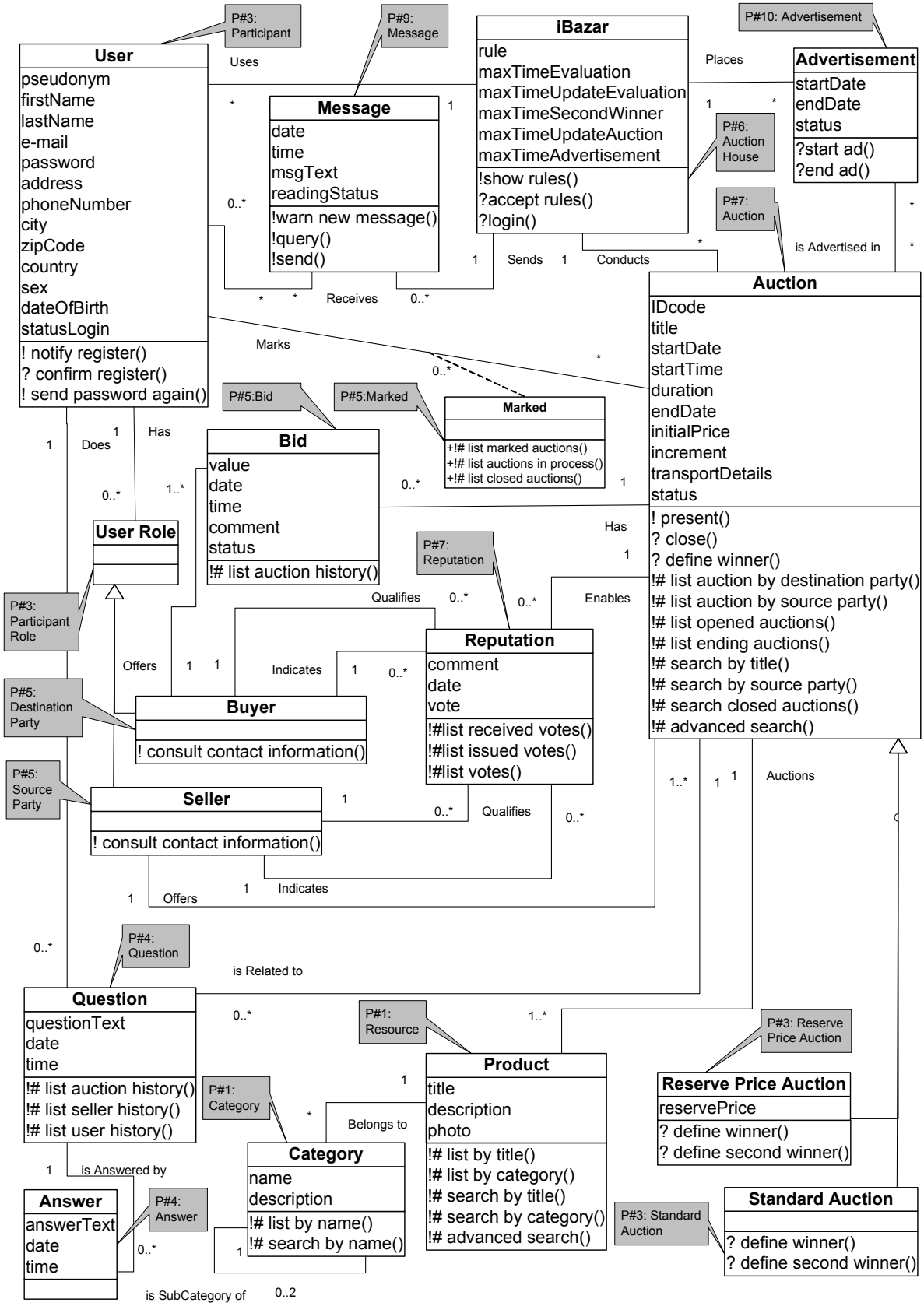


图 24　在线拍卖管理系统 iBazar 的模式语言范例

5 结束语

本文提出的模式语言探讨了各种在线拍卖系统的系统开发，以帮助软件工程师开展这种系统的分析工作。我们正在使用 Smalltalk 开发支持这种模式语言的框架。它将为各模式提供所需的类以及 web 界面，这些界面可以定制，以满足特定的在线拍卖系统的需要。下一步工作主要是扩展这种模式语言，使其覆盖更多类型的拍卖，例如逆向拍卖。

6 答谢

我们非常感谢 Federico Balaguer，他为我们的模式语言当前版本提供了宝贵意见。我们感谢 FAPESP 为此项研究提供资金支持。

参考文献

1. Boyd, L.(1998). *Business Patterns of Association Objects*, Pages 195-408. Pattern Languages of Program Design 3, Martin, R.C., Riehle, D., Buschmann, F. Addison-Wesley, Reading – MA, USA.
2. Braga, R. T. V., Germano, F. S. R., and Masiero, P. C.(1999). A pattern language for business resource management. In *6th Pattern Languages of Programs Conference (PLoP'99)*, Monticello – IL, USA.
3. Coad, P. (1992). Object-oriented patterns. *Communications of the ACM*, 35(9):152–159.
4. Coad, P., North, D., and Mayfield, M. (1997). *Object Models: Strategies, Patterns and Applications*. Prentice-Hall, Upper Saddle River – NJ, USA, 2 edition.
5. Foote, B. and Yoder, J. (1998). Metadata and active object-models. PloP'98 and EuroPLoP'98 Conference, Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, September 1998.
6. Fowler, M. (1997a). *Analysis Patterns*. Addison-Wesley, Menlo Park – CA, USA.
7. Fowler, M. (1997b). Dealing with roles. Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, USA, September 2-5, 1997. Available for download at: http://www.martinfowler.com/articles.html.

8. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts.

9. Heck, E. and Vervest, P. (1998). How should CIOs deal with web-basead auctions? *Communications of the ACM*, 41(7):99–100.

10. Huhns, M. N. and Vidal, J. M. (1999). Online auc-tions. *IEEE Internet Computing*, 3(3):103–105.

11. Isakowitz, T., Bieber, M., and Vitali, F. (1998). Web information systems. *Communications of the ACM*, 41(7):78–80.

12. Johnson, R. and Woolf, B. (1998). *Type-Object*, pages 47–65. Pattern Languages of Program Design 3, Martin, R.C., Riehle, D. , Buschmann, F. – Addison-Wesley, Reading – MA, USA.

13. Kumar, M. and Feldman, S. T. (1998). Internet auctions. In *3rd USENIX Workshop on Electronic Commerce*, Boston – MA, USA.

14. Larman, C. (1997). *Applying UML and Patterns: An Introduction to Object-Oriented Analisys and Design*. Prentice-Hall, Upper Saddle River – NJ, USA.

15. Ré, R. and Masiero, P. C. (2001). Requirements specification for online auctions: Arremate.com, iBazar and eBay. ICMC/USP – Sao Carlos, SP, Brazil. Working Document (in Portuguese).

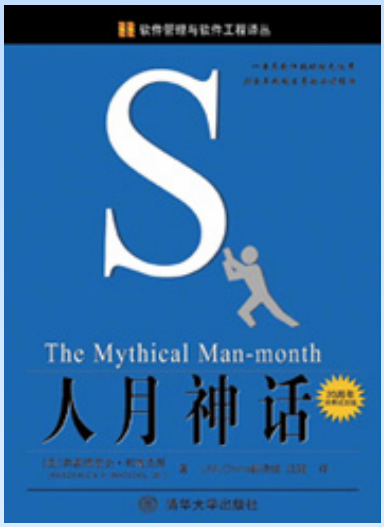
16. Resnick, P., Zeckhauser, R., Friedman, E., and Kuwabara, K. (2000). Reputations systems. *Communications of the ACM*, 43(12):15–48.

17. Vlissides, J. (1998). Pattern hatching. design patterns applied.

18. Wellman, M. P. and Wurman, P. R. (1998). Real time issues auctions. In *First IEEE Workshop on Depend-able and Real-Time E-Commerce Systems*, Denver, CO – USA.

19. Yoder, J. and Barcalow, J. (1997). Architectural pat-terns for enabling application security. In Proceedings of the 4th Conference on Patterns Language of Programming (PLoP'97), 1997. 2.

《人月神话》发行了！



《人月神话》20 周年纪念版

Fred Brooks

翻译：UMLChina 翻译组 汪 颖

二十五年后，我们仍然在读这本书

——Ed Yourdon

一种关于物品修理的分析模式

Eduardo B. Fernandez, Xiaohong Yuan 著, 刘巍 译

查看评论

摘要

文中提到的分析模式用于对修理店中的修理活动建模。(本模式)着眼于修理店中修理活动的基本情况，从最初的(修理情况)估算直到整个修理过程的结束。至于修理店的细节情况和被修理的物品留待在专门的应用或其他补充的模式中去讨论。这种模式代表的最小应用可用于各种情况下而且可以与其他相关模式共同用于描述更复杂的应用。也可被进一步抽象用于描述其他情况而不仅仅是修理过程。

1 介绍

我们介绍一种用于描述修理店中的修理过程的分析模式。这种模式属于我们所谓的语义分析模式[Fer00a]的范畴，所以本文提到的模式仅注重对修理店中的修理活动的基本面的描述，而不去关心修理店、修理的物品或顾客的具体类型。至于修理店的细节情况和被修理的物品留待在专门的应用或其他补充的模式中去讨论。这类模式目的在于给出一个将需求转化为具体的设计的起点。这类模式代表的最小应用可用于各种情况下，而且可以与其他相关模式共同用于描述更复杂的应用。也可被进一步抽象用于描述其他情况而不仅仅是修理过程。

把一件损坏的东西拿到修理店去修理，在日常生活中是一件很平常的事。顾客将损坏的东西，如电脑或者汽车送到修理店时，一个负责接待的技工会对要做的修理作出大致的估算，他同时会对修理的内容作纪录。所有修理活动都被记录到修理日志中。一次修理可能因为缺少零件或其他原因而被耽搁，也可能被取消。

2 目的

本文提及的模式描述了修理店中的修理活动，包含了对要做的修理预先作个估算、指派某些技术工人去修理和对修理的项目进行记录。

3 上下文

顾客将电脑送到电脑修理店是类似的修理店中修理过程链条中一部分（如图一）。顾客可能将电脑送到几个修理店中进行估算（Estimate类），然后再选择一家店去修理。一个接待技工作修理的估算工作，而修理工负责电脑的修理和对修理项目的记录（RepairEvent类）。每个电脑都有一个在这些修理店中所有修理项目的记录（RepairLog类）。

上述例子是普通的物品修理如汽车、手表、家用电器、复印机等很多种类修理中的一个特殊情况。为了使这个分析模式用于概念上的建模，我们将所有这些方面的例子进行抽象，并定义一个通用的模式涵盖所有的情况。通用模式可被修改以适用于特定的情形。

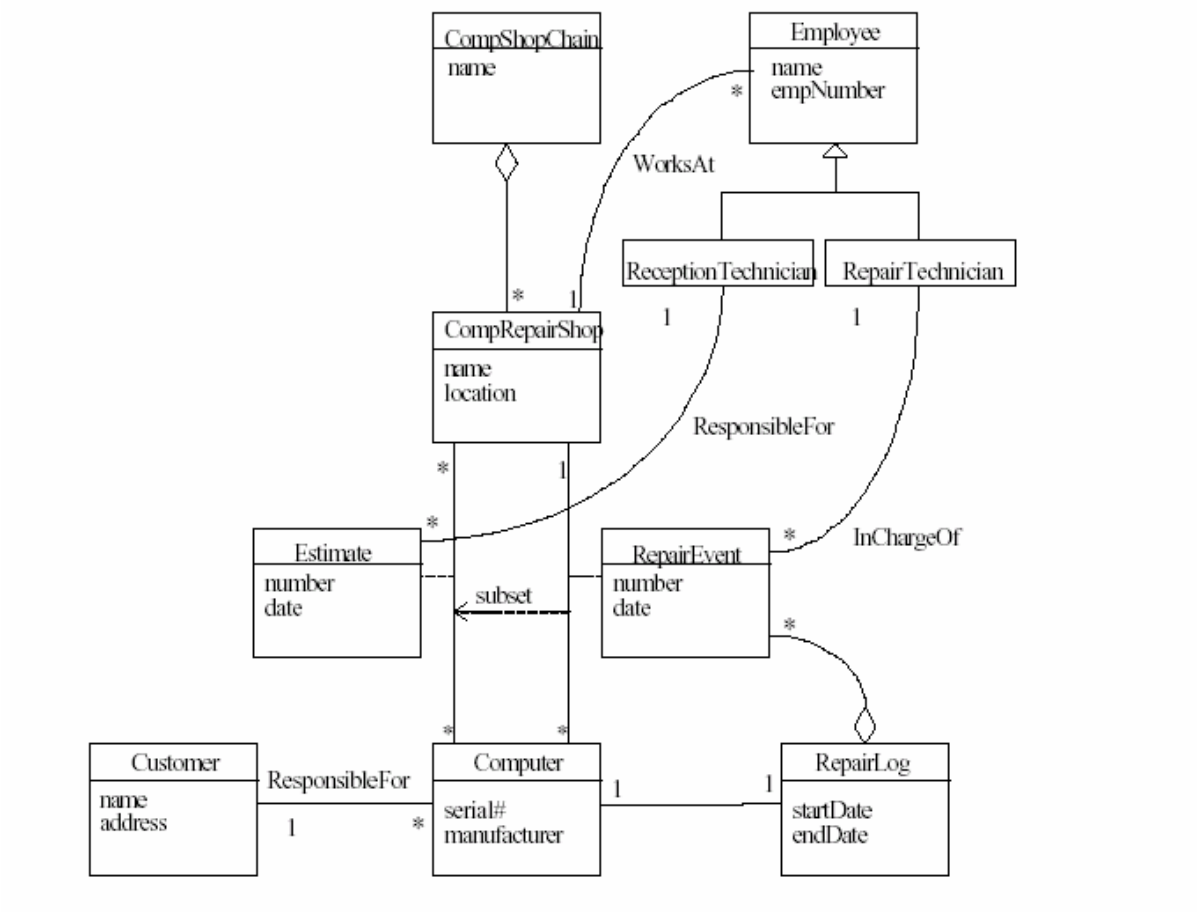


图 一. 电脑修理店的类图

4 问题

如何对修理店中的修理活动建模呢？我们需要开发出一个包括对欲修理物品进行估算、指派修理工进行修理、记录修理项目的初始模型。

5 约束

- 概念模型很难建立。修理店的修理活动尤其不是一个简单的问题。一个初始的通用模型可以起到很多帮助。
- 我们想为修理店的修理活动建模。然而，有很多情况都具备相似的结构而在细节上不同。
- 用户会到几个地方对修理服务进行预估，然而只会到其中一家进行修理。
- 模型必须包含描述实际情况的文档表示，如估算、修理活动、修理记录。否则建立这些必要的文档会很复杂。

6 解决方案

6.1 需求

本解决方案适用于实现下述用例：

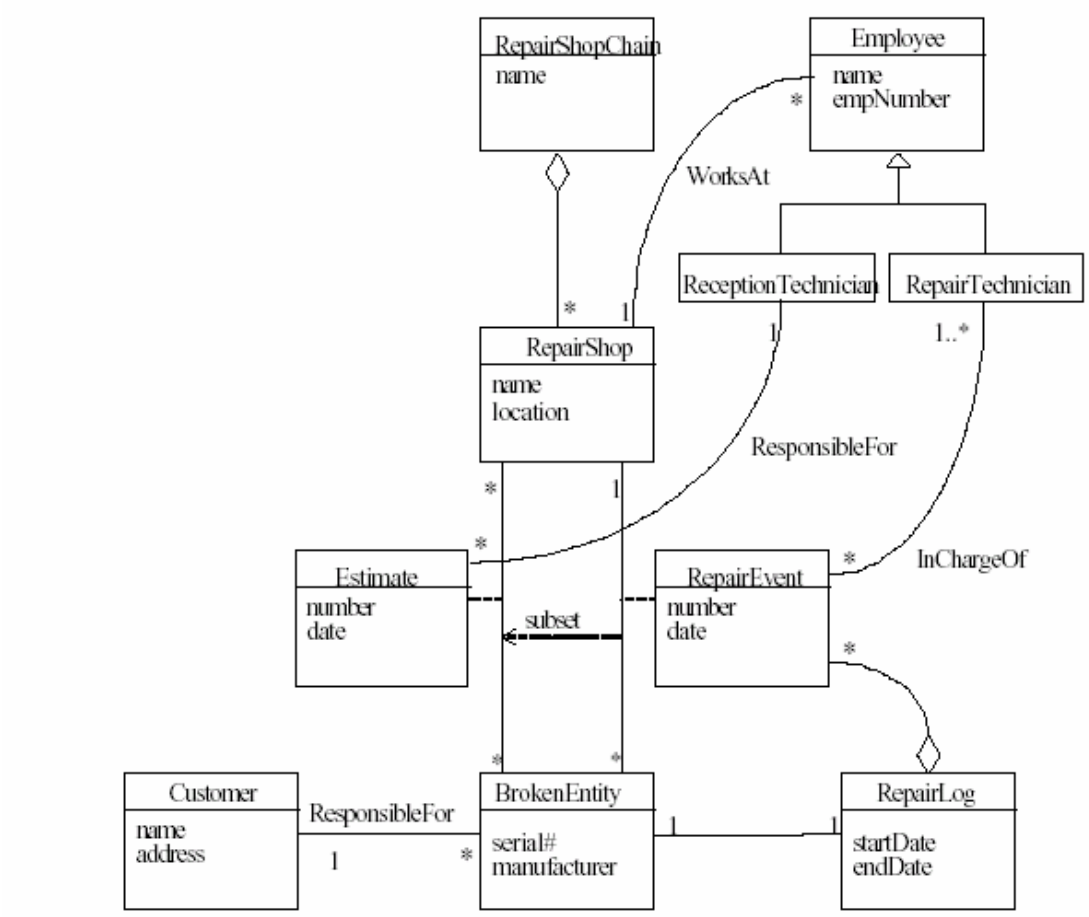
(1)对修理进行估算。接待技工会根据客户描述的问题，检查待修理的物品并给出一个修理价格的估算。

(2)对物品进行修理。客户决定在此进行修理后，修理工会被指派对物品进行修理，修理过程可以由于不同的原因而被中止。

6.2 类模型

图二是实现这些用例的类图，是图一经过提取和扩展而形成。一个**BrokenEntity**（待修物品）定义为**RepairShop**（修理店）的修理对象，**Customer**（顾客）拥有**BrokenEntity**（待修物品）的所有权（负担修理费用）。**BrokenEntity**和**RepairShop**之间的“多－多”关系反映了修理估算可以在许多修理店中进行。它们之间的“多－单”的关系表述了上述的估算中的一个可能在实际中用到。至少被修理过一次的电脑会有关于这些修理情况的记录。所有修理店的合集被指定为类**RepairShopChain**（修理店链）。修理店的雇员中典型地被分为两类角色：

ReceptionTechnician（接待技工），负责修理估算；RepairTechnician（修理技工），负责执行修理工作。



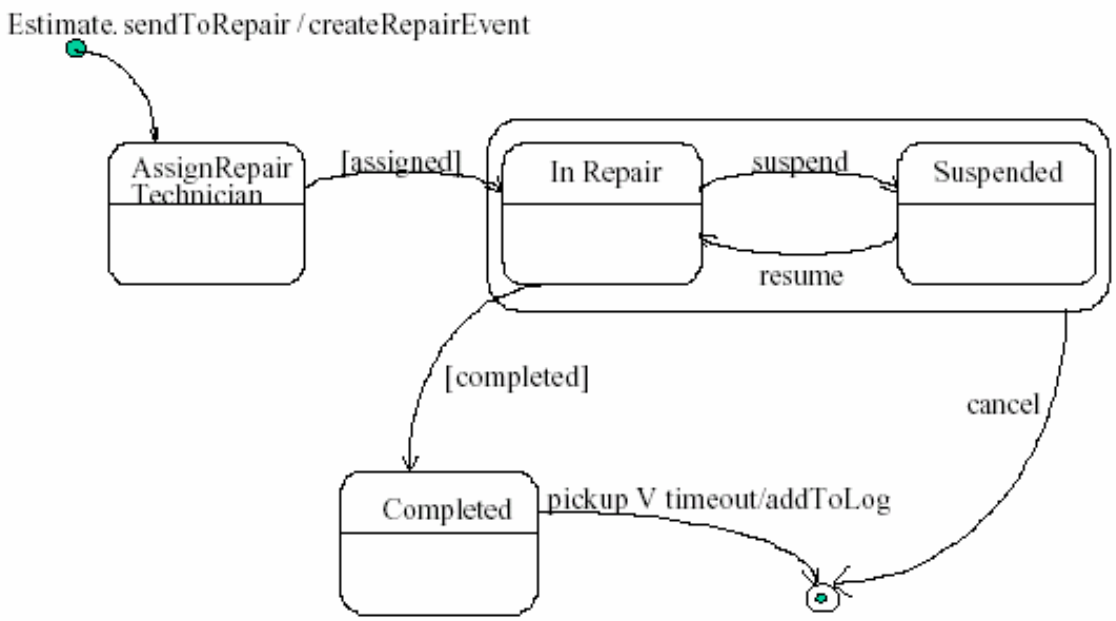
图二. 修理店模式的类图

6.3 动态模型

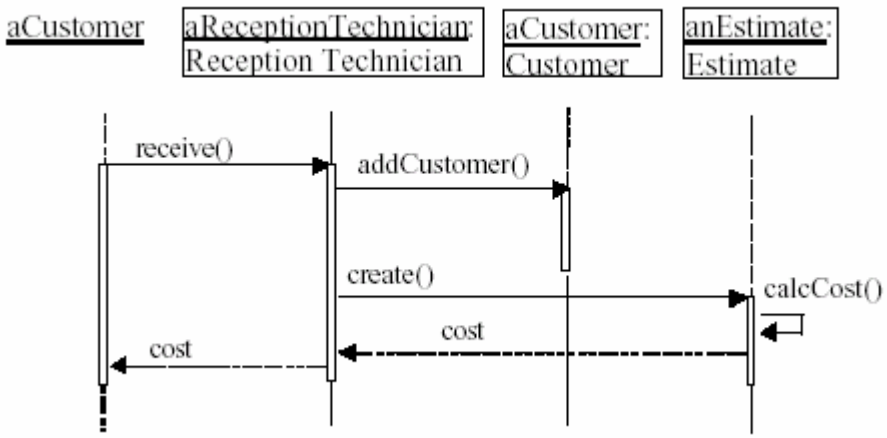
图三表示了类RepairEvent（修理活动）的状态图。修理活动的状态有*AssignRepairTechnician*（指派修理技工）、*InRepair*（修理进行中）、*Suspended*（修理暂停）和*Completed*（修理结束）。包含*InRepair*和*Suspended*的父状态含有修理取消可以从这两个状态中发生。

图四是进行一次估算的序列图，此处假定顾客是新客户，并将此信息添加到客户列表中。

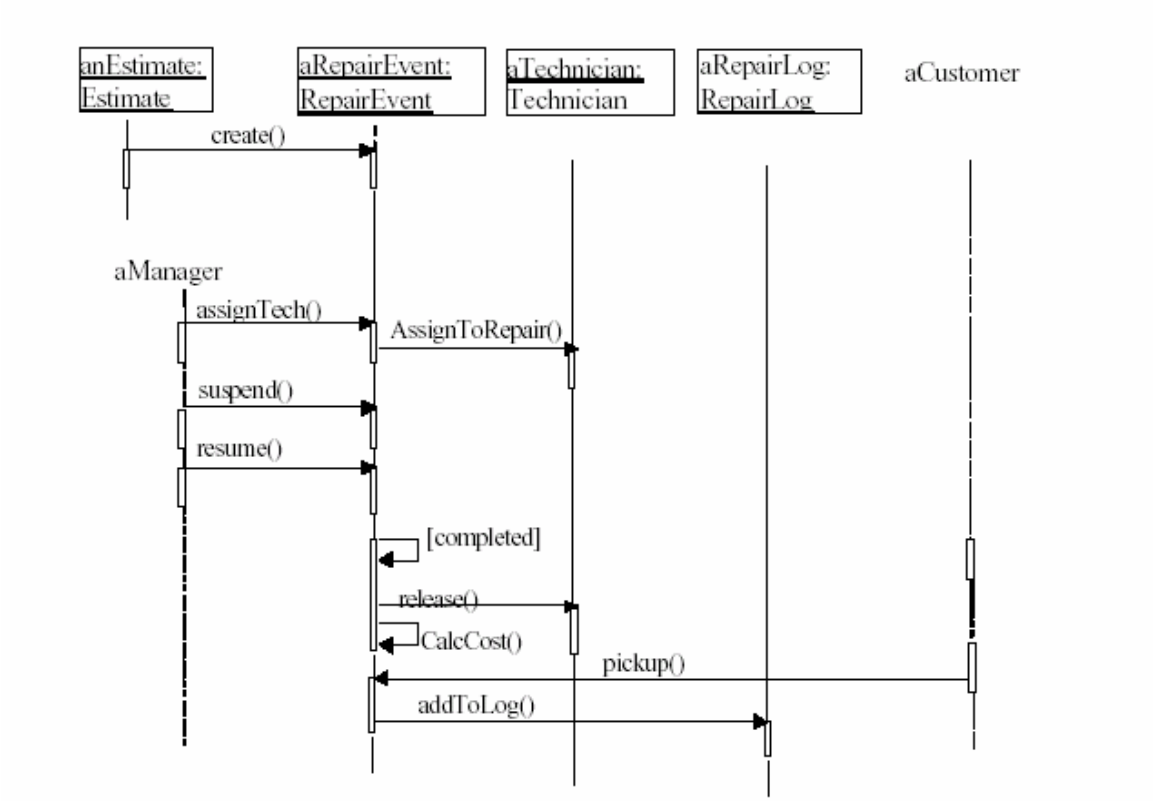
图五显示了指派修理技工工作内容的序列图。



图三. RepairEvent类的状态图



图四、修理估算的序列图



图五、修理工作指派序列图

7 结论

这个模式具备下述优点：

- 提供了现实生活情况（见第8部分）的一个通用的描述模式，即可在不同的应用中使用。
- 典型文档用类来描述，例如估算。
- 为修理活动提供了一个系统化的结构，进而优化了修理店中修理活动的处理。

缺点在于这种模式描述的情形并不都很相似：

- 在例如修理手表或修理汽车的情形，顾客将待修物品带到修理店；也可托运到修理店。在修理一个冰箱的情况下，修理工必须要上门修理。

• 接待技工有时可以根据顾客对问题的报告直接估算出修理费用。而在其他情况下，必须派一个技工进行检查并将（损坏）情况报告后，据此给出修理费用的估算。不管是否在此处修理，顾客可能因为这个检查而付费。一般负责检查的技工会在顾客决定修理后担任修理工的角色。

• 一些修理店会专注于某个特定类型的甚至是某个特定厂商或特定型号的产品的修理。另一些修理店则提供许多类型产品的修理服务，例如冰箱、洗碗机、空调设备和其他家用电器。

• 顾客可能与修理提供商签订一个合同，不必为每次修理单独付费。而是每年根据合同付出一定数量的费用。这样，修理费用就不必进行估算。

• 在产品的保修期（一定时间）内，顾客不必为修理服务付费。所以修理费用在此情况下也不必进行估算。

• 根据修理工是否有空闲，顾客可能要等待数小时或甚至数天才能获得服务。修理的时间也会因为待修物品问题的不同而不同。

上述这些说明这个模式需要进行修改以适应于特定的应用。

模式中没有提到的一些方面有：

• 待修物品的周围环境和前后关系因素的描述

• 计费和付费的策略

• 如何处理顾客的差异因素，例如个人客户、公司客户、首选客户等

• 例外情况，例如客户取回了修理的物品（对修理不满意）

• 如何在没有修理所需零件时定购所需零件

• 如何跟踪零件库存情况

• 如何在没有空闲修理工时对顾客的修理需求进行排队

• 当顾客需要送货时，如何将物品送到顾客处

• 顾客在有修理需求时，先与修理工约定好时间，然后再将待修物品送到修理店或预先约定时间由修理工到用户家里（或其他地方）上门修理的情况。

为使这个模式更通用，上述这些情况没有在考虑范围内。这些情况在补充的模式中或者在直接的应用模型中会有描述。

8 已知应用

以下是该模式应用的例子：

- 一名顾客将汽车送到修理店（或者汽车经销商）处修理的案例
- 一名顾客将损坏的计算机送到计算机商店去修理的案例
- 一个公司为一台复印机预约修理的案例
- 一名顾客预约一台冰箱的上门修理的案例

9 相关模式

当修理所需零件不足时，修理店需要定购零件。所以Order/Shipment [Fer00b]（订货/发货）模式对此进行了补充（编注：见《非程序员》第14期）。顾客可能需要与修理工提前预约时，Reservation（预约）和Use（使用）模式[Fer99]可以被用到（编注：见《非程序员》第7期）。Stock Manager（库存管理）模式[Fer00c]也是补充模式（编注：见《非程序员》第7期），可以被用于跟踪修理店的零件库存情况。考虑到订单付费的资金方面的处理细节可以在[Hay96]和[Fow97]中找到。

本模式包含两个其他模式：Collection（集合）模式（修理店是修理店的集合???），角色模式[Bau00]（雇员在修理技工和接待技工中的角色）。

这个模式可以更通用[Fer00a]，用来给修理活动建模的结构同样也可用于描述申请入学许可或申请几个机构的服务并选择其中之一这类活动。例子包括接收入（医）院和学生申请和注册。如此泛化在建立概念模型时很有用；然而，这样需要根据具体情况要改动的部分要更多。

鸣谢

我们要感谢我们的主管Mary Lynn Manns女士在论文写作中提出的宝贵建议，并为本文的成文提供很大的帮助。

参考文献

[Bau00] D. Baumer, D. Riehle, W. Siberski, and M. Wolf, “Role Object”, Chapter 2 in *Pattern Languages of Program Design 4* (N. Harrison, B. Foote, and H. Rohnert, Eds.). Also in *Procs. of PLoP’97*, http://jerry.cs.uiuc.edu/~plop/plop97

[Fer00a] E.B. Fernandez and X. Yuan. “Semantic analysis patterns”, *Procs. of 19th Int. Conf. on Conceptual Modeling, ER2000,* 183-195.

[Fer00b] E. B. Fernandez and X. Yuan. “Analysis patterns for the order and shipment of a product”, *Procs. of PloP 2000.* http://jerry.cs.uiuc.edu/~plop/plop2k

[Fer00c] E. B. Fernandez. “Stock Manager: an analysis pattern for inventories”, *Procs. Of PloP 2000.* http://jerry.cs.uiuc.edu/~plop/plop2k

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

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

征稿

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

UMLChina

非程序员

软件工程杂志，已有数万用户

UMLChina讨论组

已有数万名成员

UMLChina.com

1999年11月成立，，专注UML技术的应用

专家交流

提供与世界级专家交流的窗口

20 年后的人月神话（The Mythical Man-Month *after* 20 Years）

只能根据过去判断将来。

- *帕特里克·亨利*

然而永远无法根据过去规划将来。

- *埃德蒙·伯克*

I know no way of judging the future but by the past.

- *PATRICK HENRY*

You can never plan the future by the past.

- *EDMUND BURKE*

为什么会出现二十周年纪念版本？

飞机划破夜空，嗡嗡地飞向纽约的拉瓜迪亚机场。所有的景色都隐藏在云层和黑暗之中。我正在看一篇平淡无奇的文档，不过并没有感到厌烦。紧挨着我的一位陌生人正在阅读《*人月神话*》，我在旁边一直等待着，看他是否会通过文字或者手势做出反映。最后，当我们向舱门移动时，我无法再等下去了：

“这本书如何？你有什么评论吗？”

“噢！这里面的东西我早就知道。”

此刻，我决定不介绍自己。

为什么《*人月神话*》得以持续？为什么看上去它仍然和现在的软件实践相关？为什么

它还拥有软件工程领域以外的读者群，律师、医生、社会学家、心理学家，和软件人员一样，不断地对这本书提出评论意见，引用它，并和我保持通信？20 年前的一本关于 30 年前软件开发经验的书，如何能够依然和现实情况相关？更不用说有所帮助了。

常听到的一个解释是软件开发学科没有正确地发展，人们经常通过比较计算机软件开发生产率和硬件制造生产率来支持这个观点，后者在 20 年内至少翻了 1000 倍。正像第 16 章所解释的，反常的并不是软件发展得太慢，而是计算机硬件技术以一种与人类历史不相配的方式爆发出来。大体上这源于计算机制造从装配工业向流水线工业、从劳动密集型向资金密集型的逐渐过渡。与生产制造相比，硬件和软件开发保持着固有的劳动密集型特性。

第二个经常提及的解释——《人月神话》仅仅是顺便提及了软件，而主要针对团队中的成员如何创建事物。这种说法的确有些道理，1975 年版本的前言中提到，软件项目管理并不像大多数程序员起初所认为的那样，而更加类似于其他类型的管理。现在，我依然认为这是正确的。人类历史是一个舞台，总是上演着相同的故事。随着文化的发展，这些故事的剧本变化非常缓慢，而舞台的布局却在随时改变。正是如此，我们发现二十世纪本身会反映在莎士比亚、荷马的作品和圣经中。因此，某种程度上，《人月神话》是关于人与团队的书，所以它的淘汰过程会是缓慢的。

不管出于什么原因，读者仍然在购买这本书，并且常给我发一些致谢的评论。现在，我常常被问到：“你现在认为哪些在当时就是错误的？哪些是现在才过时的？哪些是软件工程领域中新近出现的？”这些独特的问题是完全平等的，我将尽我最大的能力来回答它们。不过，不以上述顺序，而是按照一系列主题来答复。首先，让我们考虑那些在写作时就正确，现在依然成立的部分。

核心观点：概念完整性和结构师

概念完整性。一个整洁、优雅的编程产品必须向它的每个用户提供一个条理分明的概念模型，这个模型描述了应用、实现应用的方法以及用来指明操作和各种参数的用户界面使用策略，。用户所感受到的产品概念完整性是易用性中最重要的因素。（当然还有其他因素。Macintosh 上所有应用程序界面的统一就是一个重要的例子。此外，有可能建立统一的接口，尽管它可能很粗糙，就像 MS-DOS。）

有很多由一个或者两个人设计的优秀软件产品例子。大多数纯智力作品，像书籍、音

乐等都是采用这种方式创作出来的。不过，很多产业的产品开发过程无法负担这种获取概念完整性的直接方法。竞争带来了压力，很多现代工艺的最终产品是非常复杂的，它们的设计需要很多人月的工作量。软件产品十分复杂，在进度上的竞争也异常激烈。

任何规模很大或者非常紧急，并需要很多人力的项目，都会碰到一个特别的困难：必须由很多人来设计，但与此同时，还需要在概念上保持与单个使用人员的一致。如何组织设计队伍来获得上述的概念一致性？这是《*人月神话*》关注的主要问题。其中一点：由于参与人数的不同，大型编程项目的管理与小型项目在性质上都不同。为了获得一致性，经过深思熟虑的，有时甚至是英勇的管理活动是完全必要的。

结构师。从第 4 到第 7 章，我一直不断地在表达一个观点——委派一名产品*结构师*是最重要的行动。结构师负责产品所有方面的概念完整性，这些是用户能实际感受到的。结构师开发用于向用户解释使用的产品概念模型，概念模型包括所有功能的详细说明以及调用和控制的方法。结构师是这些模型的所有者，同时也是用户的代理。在不可避免地对功能、性能、规模、成本和进度进行平衡时，卓有成效地体现用户的利益。这个角色是全职工作，只有在最小的团队中，才能和团队经理的角色合并。结构师就像电影的导演，而经理类似于制片人。

将体系结构和设计实现、物理实现相分离。为了使结构师的关键任务更加可行，有必要将用户所感知的产品定义——体系结构，与它的实现相分离。体系结构和实现的划分在各个设计任务中形成了清晰的边界，边界两边都有大量的工作。

体系结构的递归。对于大型系统，即使所有实现方面的内容都被分离出去，一个人也无法完成所有的体系结构工作。所以，有必要由一位主结构师把系统分解成子系统，系统边界应该划分在使子系统间接口最小化和最容易严格定义的地方。每个部分拥有自己的结构师，他必须就体系结构向主结构师汇报。显然，这个过程可以根据需要重复递归地进行。

今天，我比以往更加确信。概念完整性是产品质量的核心。拥有一位结构师是迈向概念完整性的最重要一步。这个原理决不仅限于软件系统，它适用于所有的复杂事物，如计算机、飞机、防御系统、全球定位系统等。在软件工程试验室进行 20 次以上的讲授之后，我开始坚持每 4 个学生左右的小组就选择不同的经理和结构师。在如此小的队伍中定义截然不同的角色可能是有点极端，但我仍然发现这种方法即使对小型团队也运作良好，并且促使了设计的成功。

开发第二个系统所引起的后果：盲目的功能和频率猜测

为大型用户群设计。个人计算机革命的一个结果是，至少在商业数据处理领域中，客户应用程序越来越多地被商用软件包所代替。而且，标准软件包以成百上千，甚至是数百万拷贝的规模出售。源厂商支持性软件的系统结构师必须不断地为大型的不确定用户群，而不是为某个公司的单一、可定义的应用进行设计。许许多多的系统结构师现在面临着这项任务。

但自相矛盾的是，设计通用工具比设计专用工具更加困难，这是因为必须为不同用户的各种需要分配权重。

盲目的功能（Featuritis）。对于如电子表格或字处理等通用工具的结构师，一个不断困扰他们的诱惑是以性能甚至是可用性的代价，过多地向产品添加边界实用功能。

功能建议的吸引力在初期阶段是很明显的，性能代价在系统测试时才会出现。而随着功能一点一点地添加，手册慢慢地增厚，易用性损失以不易察觉的方式蔓延。[1]

对幸存和发展了若干代的大众软件产品，这种诱惑特别强烈。数百万的用户需要成百上千的功能特色，任何需求本身就是一种“市场需要它”的证明。而常见的情况是，原有的系统结构师得到了嘉奖，正工作在其他岗位或项目上，现在负责体系结构的结构师，在均衡表达用户的整体利益方面，往往缺乏经验。一个对 Microsoft Word 6.0 的近期评价声称“Word 6.0 对功能特性进行了打包，通过包缓慢地更新……Word 6.0 同样是大型和慢速的。”有点令人沮丧的是——Word 6.0 需要 4MB 内存，丰富的新增功能意味着“甚至 Macintosh IIfx 都不能胜任 Word 6 的任务”[2]。

定义用户群。用户群越大和越不确定，就越有必要明确地定义用户群，以获得概念完整性。设计队伍中的每个成员对用户都有一幅假想的图像，并且每个设计者的图像都是不同的。结构师的用户图像会有意或者无意地影响每个结构决策，因此有必要使设计队伍共享一幅相同的用户图像。这需要把用户群的属性记录下来，包括：

- 他们是谁
- 他们需要（need）什么
- 他们认为自己需要（need）什么
- 他们想要（want）的是什么

频率。对于软件产品，任何用户群属性实际上都是一种概率分布，每个属性具有若干可能的值，每个值有自己的发生频率。结构师如何成功地得到这些发生频率？对并未清晰定义的对象进行调查是一种不确定和成本高昂的做法[3]。经过很多年，我现在确信，为了得到完整、明确和共有的用户群描述，结构师应该*猜测（guess），*或者*假设（postulate）*完整的一系列属性和频率值。

这种不是很可靠的过程有很多好处。首先，仔细猜测频率的过程会使结构师非常细致地考虑对象用户群。其次，把它们写下来一般会引发讨论，这能起到解释的作用，以及澄清不同设计人员对用户图像认识上的差异。另外，明确地列举频率能帮助大家认识到哪些决策依赖于哪些用户群属性。这种非正式的敏感性分析也是颇有价值的。当某些非常重要的决策需要取决于一些特殊的猜测时，很值得为那些数值花费精力来取得更好的估计。（Jeff Conklin 开发的 gIBIS 提供了一种工具，能精确和正式地跟踪设计决策和文档化每个决策的原因[4]。我还没有机会使用它，但是我认为它应该非常有帮助。）

总结：为用户群的属性明确地记载各种猜测。*清晰和错误都比模糊不清好得多。*

“开发第二系统所引起的后果（second-system effect）”情况怎样？一位敏锐的学生说，*人月神话*推荐了一剂对付灾难的处方：计划发布任何新系统的第二个版本（第 11 章），第二系统在第 5 章中被认为是最危险的系统。

这实际上是语言引起的的差异，现实情况并不是如此。第 5 章中提到的“第二个”系统是第二个实际系统，它是引入了很多新增功能和修饰的后续系统。第 11 章中的“第二个”系统指开发第一个实际系统所进行的第二次尝试。它在所有的进度、人员和范围约束下开发，这些约束刻画了项目的特征，形成了开发准则的一部分。

图形（WIMP）界面的成功

在过去 20 年内，软件开发领域中，令人印象最深刻的进步是窗口（Windows）、图标（Icons）、菜单（Menus）、指针选取（Pointing）界面的成功——或者简称为 WIMP。这些在今天是如此的熟悉，以致于不需要任何解释。这个概念首先在 1968 年西部联合计算机大会（Western Joint Computer Converence）上，由斯坦福研究机构（Stanford Research Institute）的 Doug Englebart 公开提出[5]。接着，这种思想被 Xerox Palo Alto Research Center 所采纳，用在了由 Bob Taylor 和他的团队所开发的 Alto 个人工作站中。Steve Jobs

在 Apple Lisa 型计算机中应用了该理念，不过 Apple Lisa 运行速度太慢，以致于无法承载这个令人激动的易用性概念。后来在 1985 年，Jobs 在取得商业成功的 Apple Macintosh 机器上体现了这些想法。接下来，它们被 IBM PC 及其兼容机的 Microsoft Windows 所采用。我自己的例子则是 Mac 版本[6]。

通过类比获得的概念完整性。WIMP 是一个充分体现了概念完整性的用户界面例子，完整性的获得是通过采用大家非常熟悉的概念模型——对桌面的比喻，以及一致、细致的扩展，后者充分发挥了计算机的图形化实现能力。例如，窗口采用覆盖，而不是排列的方式，这直接来自类比。尽管这种方法成本很高，但却是正确的决定。计算机图形介质提供了对窗口尺寸的调整，这是一种保持一致概念的延伸，给用户提供了新的处理能力，桌面上的文件是无法轻易地调整大小和改变形状的；拖放功能则直接出自模仿，使用指针来选择图标是对人用手拾起东西的直接模拟；图标和嵌套文件夹源于桌面的文档，回收站也是如此；剪切、复制和粘贴则完全反映了我们使用文档的一些习惯；我们甚至可以通过向回收站拖拽磁盘的图标来弹出磁盘——象征手法是如此的贴切，扩展是如此的连贯一致，以致于新用户常常会被它所体现出的理念打动。

哪些地方使 WIMP 界面远远超越了桌面的比喻？主要是在两个方面：菜单和单手操作。在真正的桌面上工作时，人们实际上是操作文档，而不是叫某人来完成这些动作。当要求他人进行某个活动时，常常是新产生，而不是选择一个口头或者书面祈使句：“请将这个归档。”“请找出前面一致的地方。”“请把这个交给 Mary 去处理。”

无论是手写还是口头的命令，现有处理能力还无法对自由产生的命令形式进行可靠的翻译和解释。所以，界面设计人员从用户对文档的直接动作中去除了上面提到的两个步骤。他们非常聪明地从桌面文档操作中选取了一些常用命令，形成了类似于公文的“便条”，用户只需从一些语义标准的强制命令菜单中进行选择。这个概念接着被延伸到水平菜单和垂直的下拉子菜单中。

命令表达和双光标问题。命令是祈使句，它们通常都有一个动词和直接宾语。对于任何动作，必须指定一个动词和一个名词。对事物选取的直接模仿要求：使用屏幕上不同的两个光标，同时指定两件事物。每个光标分别由左右手中的鼠标来控制。毕竟，在实际的桌面上，我们通常使用两只手来操作。（不过，一只手常常是将东西固定在某处，这一点在计算机桌面是默认情况。）而且，我们当然具备双手操作的能力，我们习惯上使用双手来打字、

驾驶、烹饪。但是，提供一个鼠标已经是个人计算机制造商向前迈进的一大步，没有任何商业系统可以容纳由双手分别控制的两只鼠标同时进行的动作[7]。

界面设计人员接受了现实，为一只鼠标设计。设计人员采用的句法习惯是首先指出（选择）名词的，接着指出一个动词菜单项。这确实牺牲了很多易用性。当我看到用户、或者用户录像、或者光标移动的计算机跟踪情况。我立刻对一个光标必须完成两件事而感到惊讶：选择窗口上桌面部分的一个对象，再选择菜单部分的一个动词，寻找或者重新寻找桌面上的一个对象。接着，拉下菜单（常常是同一个）选择一个动词。光标来来往往、周而复始地从数据区移到菜单区，每一次都丢弃了一些有用的位置信息“上次在这个空间的什么地方”——总而言之，是一个低效的过程。

一个卓越的解决方案。即使软件和器材可以很容易实现两个同时活动的光标，也仍然存在一些空间布局上的困难。WIMP 象征手法中的桌面实际上包括了一个打字机，它必须在实际桌面的物理空间中容纳一个物理键盘。键盘加上两个鼠标垫会占据大量双手所及的空间。不过，键盘问题实际上是一个机会——为什么不一只手在键盘上指定动词，另一只手使用鼠标来指定名词，从而使高效的双手操作成为可能？这时，光标停留在数据区，为后续点击拾取提供了充分的空间活动能力。真正的高效，真正强大的用户功能。

用户功能和易用性。不过，这个解决方案舍弃了一些易用性——菜单提供了任何特定状态下的一些可选的有效动词。例如，我们可以购买某个商品，将它带回家，紧记购买的目的，遵照菜单上不同的动词略为试验一下，就可以开始使用，并不需要去查看手册。

软件结构师所面临的最困难问题是如何确切地平衡用户功能和易用性。是为初学者或偶尔使用的用户设计简单操作，还是为专业用户设计强大的功能？理想的答案是通过概念一致的方式把两者都提供给用户——这正是 WIMP 界面所达到的目标。每个频繁使用的菜单动词（命令）都有一个快捷键，因此可以作为组合通过左手一次性地输入。例如，在 Mac 机器上，命令键（⌘）正好在 Z 和 X 键的下方，因此使用最频繁的操作被编码成⌘z、⌘x、⌘c、⌘v、⌘s。

批注: 页：1
原文中应是苹果型符号，但译者找不到此符号

从新手向熟练用户的逐渐过渡。双重指定命令动词的系统不但满足了新手快速学习的需要，而且它在不同的使用模式之间提供了平滑的过渡。被称为*快捷键*的字符编码，显示在菜单上的动词旁边，因此拿不准的用户可以激活下拉菜单，检查对应的快捷键，而不是直接在菜单上选取。每个新手从他最常使用的命令中学习快捷键。由于⌘z 可以撤消任何单一操

作，所以他可以尝试任何感到不确定的快捷键。另外，他可以检查菜单，以确定命令是否有效。新手会大量地使用菜单，而熟练用户几乎不使用，中间用户仅偶尔需要访问菜单，因为每个人都了解组成自己大多数操作的少数快捷键。我们大多数的开发设计人员对这样的界面非常熟悉，对其优雅而强大的功能感到非常欣慰。

强制体系结构的实施，作为设备的直接整合。Mac 界面在另一个方面很值得注意。没有任何强迫，它的设计人员在所有的应用程序中使用标准界面，包括了大量的第三方程序。从而，用户在界面上获得的概念一致性不仅仅局限在机器所配备的软件，而且遍及所有的应用程序。

Mac 设计人员把界面固化到只读内存中，从而开发者使用这些界面比开发自己的特殊界面更容易和快速。这些获取一致性的措施得到了足够广泛的应用，以致于可以形成实际的标准。Apple 的管理投入和大量说服工作协助了这些措施。产品杂志中很多独立评论家，认识到了跨应用概念完整性的巨大价值，通过批评不遵从产品的反面例子，对上述方法进行了补充。

这是第 6 章中所推荐技术的一个非常杰出的例子，该技术通过鼓励其他人直接将某人的代码合并到自己的产品中来获得一致性，而不是试图根据某人的技术说明开发自己的软件。

WIMP 的命运：过时被淘汰。尽管 WIMP 有很多优点，我仍期望 WIMP 界面在下一代技术中成为历史。如同我们支配我们机器一样，指针选取仍将是表达名词的方式，语音则无疑成为表达动词的方法。Mac 上的 Voice Navigator 和 PC 上的 Dragon 已经提供了这种能力。

没有构建舍弃原型——瀑布模型是错误的！

一幅让人无法忘怀的图画，倒塌的塔科马大桥，开启了第 11 章。文中强烈地建议：“为舍弃而计划。无论如何，你一定要这样做。”现在我觉得这是错误的，并不是因为它太过极端，而是因为它太过简单。

在《未雨绸缪》一章体现的概念中，最大的错误是它隐含地假设了使用传统的顺序或者瀑布开发模型。该模型源自于类似甘特图布局的阶段化流程，常常绘制成图 19.1 的形状。Winton Royce 在 1970 年的一篇经典论文中，改进了顺序模型，他提出：

- 存在一些从一个阶段到前一个阶段的反馈
- 将反馈限定在直接相邻的先前阶段，从而容纳它引起的成本增加和进度延迟

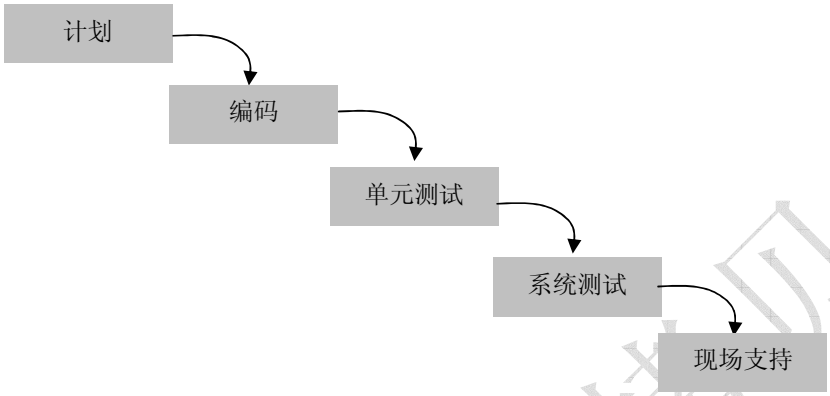


图 19.1：软件开发的瀑布模型

他给开发者提出的建议“构建两次”比《人月神话》的好[8]。受到瀑布模型不良影响并不只是第 11 章，而是从第 2 章的进度计划规则开始，贯穿了整本书。第 2 章中的经验法则分配了 1/3 的时间用于计划，1/6 用于编码，1/4 用于单元测试以及 1/4 用于系统测试。

瀑布模型的基本谬误是它假设项目只经历*一次*过程，而且体系结构出色并易于使用，设计是合理可靠的，随着测试的进行，编码实现是可以修改和调整的。换句话说，瀑布模型假设所有错误发生在编码实现阶段，因此它们的修复可以很顺畅地穿插在单元和系统测试中。

实际上，《未雨绸缪》并没有迎面痛击这个错误。它不是对错误的诊断，而是补救措施。现在，我建议应该一块块地丢弃和重新设计系统，而不是一次性地完成替换。就目前的情况而论，这没有问题，但它并没有触及问题的根本。瀑布模型把系统测试，以及潜在地把*用户*测试放在构件过程的末尾。因此，只有在投入了全部开发投资之后，才能发现无法接受的性能问题、笨拙功能以及察觉用户的错误或不当企图。不错，Alpha 测试对规格说明的详细检查是为了尽早地发现这些缺陷，但是对于实际参与的用户却没有对应的措施。

瀑布模型的第二个谬误是它假设整个系统一次性地被构建，在所有的设计、大部分编码、部分单元测试完成之后，才为闭环的系统测试合并各个部分。

瀑布模型，这个大多数人在 1975 年考虑的软件项目开发方法，不幸地被奉为军用标准 DOD-STD-2167，作为所有国防部军用软件的规范。所以，在大多数有见地的从业者认识到瀑

布模型的不完备并放弃之后，它仍然得以幸存。幸运的是，DoD 也已经慢慢察觉到这一点[9]。

必须存在逆向移动。就像本章开始图片中精力充沛的大马哈鱼一样，在开发过程“下游”的经验和想法必须跃行而上，有时会超过一个阶段，来影响“上游”的活动。

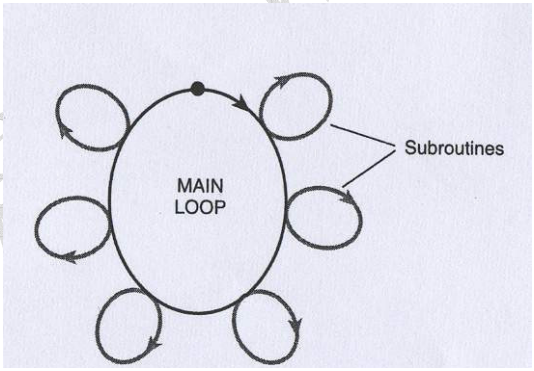
例如，设计实现会发觉有些体系结构的功能定义会削弱性能，从而体系结构必须重新调整。编码实现会发现一些功能会使空间剧增，超过要求，因此必须更改体系结构定义和设计实现。

所以，在把任何东西实现成代码之前，可能要往复迭代两个或更多的体系结构－设计－实现循环。

增量开发模型更佳——渐进地精化

构建闭环的框架系统

从事实时系统开发的 Harlan Mills，早期曾提倡我们首先应该构建实时系统的基本轮询回路，为每个功能都提供子函数调用（*占位符*），但仅仅是空的子函数（图 19.2）。对它进行编译、测试，使它可以不断运行。它不直接完成任何事情，但至少是正常运行的[10]。



注：

- MAIN LOOP－主循环
- Subroutines－子函数

图 19.2

接着，我们添加（可能是基本的）输入模块和输出模块。瞧，一个可运行的系统出现了，尽管只是一个框架。然后，一个功能接一个功能，我们逐渐开发和增加相应模块。*在每个阶段，我们都拥有一个可运行的系统。*如果我们非常勤勉，那么每个阶段都会有一个经过

调试和测试的系统。（随着系统的增长，使用所有先前的测试用例对每个新模块进行的回归测试也采用这种方式进行。）

在每个功能基本可以运行之后，我们一个接一个地精化或者重写每个模块——*增量地开发（growing）*整个系统。不过，我们有时的确需要修改原有的驱动回路，或者甚至是回路的模块接口。

因为我们在所有时刻都拥有一个可运行的系统，所以

- 我们可以很早开始用户测试，以及
- 我们可以采用按预算开发的策略，来彻底保证不会出现进度或者预算的超支（以允许的功能牺牲作为代价）。

我曾在北卡罗来纳大学教授软件工程实验课程 22 年，有时与 David Parnas 一起。在这门课程中，通常 4 名学生的团队会在一个学期内开发某个真正的实时软件应用系统。大约是一半的时候，我转而教授增量开发的课程。我常常会被屏幕上第一幅图案、第一个可运行的系统对团队士气产生的鼓舞效果而感到震惊。

Parnas 产品族

在这整个 20 年的时间里，David Parnas 曾是软件工程思潮的带头人。每个人对他的信息隐藏概念都很熟悉，但对他另一个非常重要的概念——将软件作为一系列相关的产品族来设计[11]——相对了解较少。Parnas 力劝设计人员对产品的后期扩展和后续版本进行预测，定义它们在功能或者平台上的差异，从而搭建一棵相关产品的家族树（图 19.3）。

设计类似一棵树的技巧是将那些变化可能性较小的设计决策放置在树的根部。

这样的设计使得模块的重用最大化。更重要的是，可以延伸相同的策略，使它不但可以包括发布产品，而且还包括以增量开发策略创建的后续中间版本。这样，产品可以通过它的中间阶段，以最低限度的回溯代价增长。

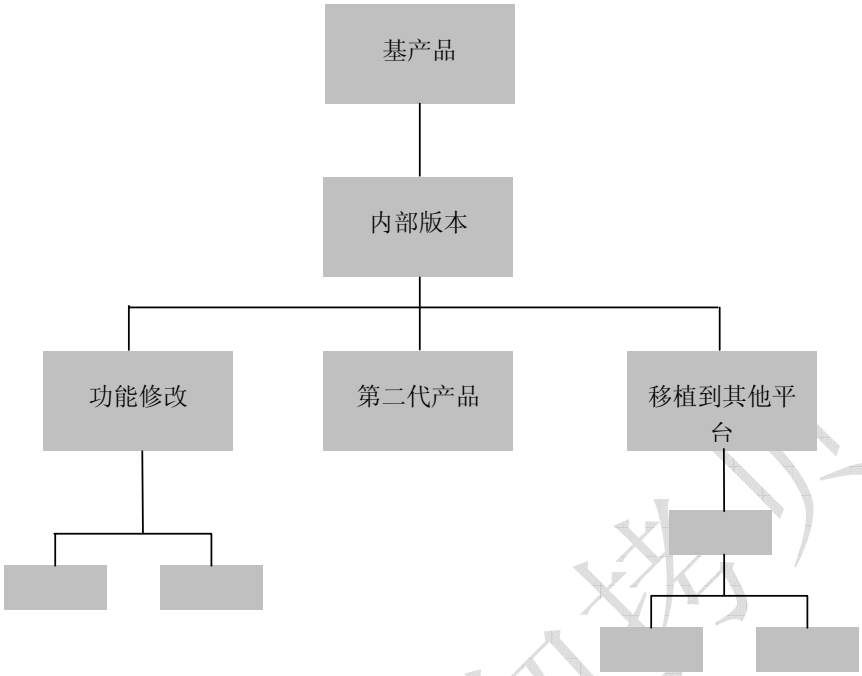


图 19.3

Microsoft 的“每晚重建”方法

Jams McCarthy 向我描述了他的队伍和微软其他团队所使用的产品开发流程，这实际上是一种逻辑上的增量式开发。他说，

在我们第一次产品发布之后，我们会继续发布后续版本，向已有的可运行系统添加更多的功能。为什么最初的构建过程要不一样呢？因此，从我们第一个里程碑开始[第一次发布有三个里程碑]，我们每晚重建开发中的系统[以及运行测试用例]。该构建周期成了项目的“心跳”。每天，一个和多个程序员-测试员队伍提交若干具有新功能的模块。在每次重建之后，我们会获得一个可运行的系统。如果重建失败，我们将停下整个过程，直到找到问题所在并进行修复。在任何时间，团队中的每个人都了解项目的状态。

这是非常困难的。你必须投入大量的资源，而且它是一个规范化、可跟踪、开诚布公的流程。它向团队提供了自身的可信度，而可信度决定了你的士气和情绪状态。

其他组织的软件开发人员对这个过程感到惊讶，甚至震惊。其中一个人说：“我们可以实现每周一次的重建，但是如果每晚一次的话，我想不大可能，工作量太大了。”这可能是对的。例如，Bell 北方研究所就是每周重建 1 千 2 百万行的系统。

增量式开发和快速原型

增量开发过程能使真正的用户较早地参与测试，那么它与快速原型之间的区别是什么呢？我认为它们既是互相关联，又是相互独立的。各自可以不依赖对方而存在。

Harel 将原型精彩地定义成：

仅仅反映了概念模型准备过程中所做的设计决策[的一个程序版本]，它并未反映受实现考虑所驱使的设计决策[12]。

构建一个完全不属于发布产品的原型是完全可能的。例如，可以开发一个界面原型，但是并不包含任何的实际功能，而仅仅是一个看上去履行了各个步骤的有限状态机。甚至可以通过模拟系统响应的向导技术来原型化和测试界面。这种原型化对获取早期的用户反馈非常有用，但是它和产品发布前的测试区别很大。

类似的，实现人员可能会着手开发产品的某一块，并完整地实现该部分的有限功能集合，从而可以尽早发现性能上的潜在问题。那么，“从第一个里程碑开始构建”的微软流程和快速原型之间的差别是什么？功能。第一个里程碑产品可能不包含足够的功能使任何人对它产生兴趣，而可发布产品和定义中的一样，在完整性上——配备了一系列实用的功能集，在质量上——它可以健壮地运行。

关于信息隐藏，Parnas 是正确的，我是错误的

在第 7 章中，关于每个团队成员应该在多大程度上被允许和鼓励相互了解设计和代码的问题，我对比了两种方法。在操作系统 OS/360 项目中，我们决定*所有*的程序员应该了解*所有*的材料——每个项目成员都拥有一份大约 10,000 页的项目工作手册拷贝。Harlan Mills 颇有说服力地指出“编程是个开放性的公共过程”。把所有工作都暴露在每个人的凝视之下，能够帮助质量控制，这既源于其他人优秀工作的压力，也由于同伴能直接发现缺陷和 bug。

这个观点和 David Parnas 的观点形成了鲜明的对比。David Parnas 认为代码模块应该采用定义良好的接口来封装，这些模块的内部结构应该是程序员的私有财产，外部是不可见的。编程人员被屏蔽而不是暴露在他人模块内部结构面前。这种情况下，工作效率最高[13]。

我在第 7 章中并不认同 Parnas 的概念是“灾难的处方”。但是，Parnas 是正确的，我是错误的。现在，我确信信息隐藏——现在常常内建于面向对象的编程中——是唯一提高软

件设计水平的途径。

实际上，任何技术的使用都可能演变成灾难。Mill 的技术是通过了解接口另一侧的情况，使编程人员能理解他们所工作接口的详细语义。这些接口的隐藏会导致系统的 bug。Parnas 的技术在面对变更时是很健壮的，更加适合为变更设计的理念。

第 16 章指出了下列情况：

- 过去在软件生产率上取得的进展大多数来自消除非内在的困难，如笨拙的编程语言、漫长的批处理周转时间等。
- 像这些比较容易解决的困难已经不多了。
- 彻底的进展将来自对根本困难的处理——打造和组装复杂概念性结构要素。

最明显的实现这些的方法是，认为程序由比独立的高级语言语句、函数、模块或类等更大的概念结构要素组成。如果能对设计和开发进行限制，我们仅仅需要从已建成的集合中参数化这些结构要素，并把它们组装在一起，那么我们就能大幅度提高概念的级别，消除很多无谓的工作和大量语句级别的错误可能性。

Parnas 的模块信息隐藏定义是研究项目中的第一步，它是面向对象编程的鼻祖。Parnas 把模块定义成拥有自身数据模型和自身操作集的软件实体。它的数据仅仅能通过它自己的操作来访问。第二步是若干思想家的贡献：把 Parnas 模块提升到*抽象数据类型*，从中可以派生出很多对象。抽象数据类型提供了一种思考和指明模块接口的统一方式，以及容易保证实施的类型规范化访问方法。

第三步，面向对象编程引入了一个强有力的概念——*继承*，即类（数据）默认获得类继承层次中祖先的属性[14]。我们希望从面向对象编程中得到的最大收获实际上来自第一步，模块隐藏，以及预先建成的、*为了重用而设计和测试*的模块或者类库。很多人忽视了这样一个事实，即上述模块不仅仅是程序，某种意义上是我们在第 1 章中曾讨论过的编程产品。许多人希望大规模重用，但不付出构建产品级质量（通用、健壮、经过测试和文档化的）模块所需要的初始代价——这种期望是徒劳的。面向对象编程和重用在第 16 和 17 章中有所讨论。

人月到底有多少神话色彩？Boehm 的模型和数据

很多年来，人们对软件生产率和影响它的因素进行了大量的量化研究，特别是在项目人员配备和进度之间的平衡方面。

最充分的一项研究是 Barry Boehm 对 63 个项目的调查，其中大多数是航空项目和 25 个 TRW 公司的项目。他的《*软件工程经济学*》（*Software Engineering Economics*）不但包括了很多结果，而且还有一系列逐步推广的成本模型。尽管一般商业软件的成本模型和根据政府标准开发的航空软件成本模型中的系数肯定不同，不过他的模型使用了大量的数据来支撑。我想从现在起，这本书将作为一代经典。

他的结果充分地吻合了《人月神话》的结论，即人力（人）和时间（月）之间的平衡远不是线性关系，使用人月作为生产率的衡量标准实际是一个神话。特别的，他发现：[15]

- 第一次发布的成本最优进度时间，$T = 2.5 (MM)^{1/3}$。即，月单位的最优时间是估计工作量（人月）的立方根，估计工作量则由规模估计和模型中的其他因子导出。最优人员配备曲线是由推导得出的。
- 当计划进度比最优进度长时，成本曲线会缓慢攀升。时间越充裕，花的时间也越长。
- 当计划进度比最优进度短时，成本曲线急剧升高。
- *无论安排多少人手，几乎没有任何项目能够在少于 3/4 的最优时间内获得成功！*当高级经理向项目经理要求不可能的进度担保时，这段结论可以充分地作为项目经理的理论依据。

Brooks 准则有多准确？曾有很多细致的研究来评估 Brooks 法则的正确性，简言之，向进度落后的软件项目中添加人手只会使进度更加落后。最棒的研究发表在 Abdel-Hamid 和 Madnick 在 1991 年出版的一本颇有价值的书《*软件项目动力学：一条完整的路*》[16]（*Software Project Dynamics：An Integrated Approach*）中。书中提出了项目动态特性的量化模型。关于 Brooks 准则的章节提供了更详细的分析，指出了在各种假设下的情况，即何时添加多少人员将会产生什么样的结果。为了进行研究，作者扩展了他们自己一个中型规模项目的模型，假设新成员有学习曲线和需要额外的沟通和培训工作。他们得出结论“向进度落后的项目中添加人手总会增加项目的成本，但并不一定会使项目更加落后。”特别的，由于新成员总会立刻带来需要数周来弥补的负面效应，所以在项目早期添加额外的人力比在后期加入更

加安全一些。

Stutzke 为了进行相似的研究，开发了一个更简单的模型，得出了类似的结果[17]。他对引入新成员进行了详细的过程和成本分析，其中包括把他们的指导人员调离原有的项目任务。他在一个真正的项目上测试了他的模型，在项目中期的一些偏移之后，他成功地添加了一倍人手，并且保证了原先的进度。相对于增加更多程序员，他还试验了的其他方法，特别是加班工作。在他的很多条实践建议中，最有价值的部分是如何添加新成员，进行培训，用工具来支持等等。特别值得注意的是，他建议开发项目后期增加的开发人员，必须作为团队成员，愿意在过程中努力投入和工作，而不是企图改变或者改进过程本身！

Stutzke 认为更大型的项目中，增加的沟通负担是次要作用，没有对它建模。至于 Abdel-Hamid 和 Madnick 是否或者如何考虑这个问题，则不是很清楚。上面提到的两个模型都没有考虑开发人员必须重新安排的事实，而在实际情况中，我发现这常常是一个非常重要的步骤。

这些细致的研究使“异常简化”的 Brooks 准则更加实用。作为平衡，我还是坚持这个简单的陈述，作为真理的最佳近似，以及一项经验法则——警告经理们避免对进度落后的项目采取的盲目、本能的修补措施。

人就是一切（或者说，几乎是一切）

很多读者发现很有趣的是，《人月神话》的大部分文章在讲述软件工程管理方面的事情，较少涉及到技术问题。这种倾向部分因为我在 IBM 360 操作系统（现在是 MVS/370）项目中角色的性质。更基本的是，这来自一个信念，即对于项目的成功而言，项目人员的素质、人员的组织管理是比使用的工具或采用的技术方法更重要的因素。

随后的研究支持了上述观点。Boehm 的 COCOMO 模型发现团队质量目前是项目成功最大的决定因素，实际上是下一个次重要因素的 4 倍。现在，软件工程的大多数学术研究集中在工具上。我很欣赏和期盼强大的工具，同样我也非常鼓励对软件管理动态特征——对人的关注、激励、培养——的持续研究。

人件。近年来，软件工程领域的一个重大贡献是 DeMarco 和 Lister 在 1987 年出版的数据，《*人件：高生产率的项目和团队*》（*Peopleware：Productive Projects and Teams*）。

它所表达的观点是“我们行业的主要问题实质上更侧重于*社会学（sociological）*而不是*科学技术（technological）*。”它充满了很多精华，如“管理人员的职责不是要人们去工作，而是是创造工作的可能。”它涉及了如空间、布置、团队的餐饮等主题。DeMarco 和 Lister 从他们的 Coding War Games 项目中提供的数据，显示了相同组织中开发人员的表现之间，和工作空间和生产率以及缺陷水平之间令人吃惊的关联。

顶尖人员的空间更加安静、更加私人、保护得更好以免受打断，还有很多……这对你真的很要紧吗……是否安静、空间和免受打搅能够帮助你的人员更好地完成工作，或者[换个角度]能帮助你吸引和留住更好的人员？

我衷心地向我的读者推荐这本书。

项目转移。DeMarco 和 Lister 对团队*融合*给予了相当大的关注，团队融合是一个无形的，但是非常关键的特性。很多地点分散的公司，项目从一个实验室转移到另一个。从中，我认为团队融合正是管理上被忽视的因素。

我的观察和经验大约局限在六、七个项目转移中，其中没有一个是成功的。*任务*可以成功地转移，但是对于项目的转移，即使拥有良好的文档、先进的设计以及保留部分原有人员，新队伍实际上依然是重新开始。我认为正是由于破坏了原有团队的整体性，导致了产品雏形的夭折，项目重新开始。

放弃权力的力量

如果人们认同我在文中多处提到的观点——创造力来自于个人，而不是组织架构或者开发过程，那么项目管理面对的中心问题是如何设计架构和流程，来提高而不是压制主动性和创造力。幸运的是，这个问题并不是软件组织所特有，一些杰出的思想家正努力地致力于这项工作。E.F.Schumacher 在他的经典《*小就是美：人们关心的经济学*》（*Small is Beautiful: Economics as if People Mattered*）中，提出了最大化员工创造力和工作乐趣的理论。他的第一个原理是引自 Pope Pius XI 教皇通谕 *Quadragesimo Anno* 中的“附属职能行使原理”：

向大型组织指派小型或者附属机构能够完成的职责是不公平的，同时也是正常次序的不幸和对它的干扰。对于每项社会活动，就其本质而言，应该配备对社会个体成员的帮助，

而不是去破坏和吸收它们……那些当权者应该确信遵守“附属职能行使”原理，能在各种各样的组织中维持更加完美的次序，越强和越有效的社会权威将会是国家更加融洽和繁荣的条件[19]。

Schumacher 继续解释到：

附属职能行使原理告诉我们——如果较低级别组织的自由和责任得以保留，中心权威实际上是得到了加强；其结果是，从整体而言，组织机构实际上将“更加融洽和繁荣”。

*如何才能获得上述的架构？……大型组织机构由很多准自治单元构成，我们称之为*准公司。*它们中的每一个都拥有大量的自由，来为创造性和*企业家职能*提供最大的可能机会……。每个准公司同时具备盈亏帐目和资产负债表*[20]。

软件工程中最激动人心的进展是将上述组织理念付诸实践的早期阶段。首先，微型计算机革命创造了新型的软件工业，出现了成百上千的新兴公司。所有这些小规模的公司热情、自由和富有创造性。随着很多小型公司被大公司收购，这个产业正在发生着变化，而那些大公司是否理解和保留小规模的创造性尚待分晓。

更不寻常的是，一些大型公司的高层管理已经开始着手将一些权力下放到软件项目团队，使它们在结构和责任上接近于 Schumacher 的准公司。其运作的结果是令人欣喜和吃惊的。

微软的 Jim AcCarthy 向我描述了他在解放团队上的经验：

每个队伍（30 至 40 人）拥有自己的任务、进度，甚至如何定义、构建、发布的过程。团队由 4 或 5 个专家组成，包括开发、测试和书写文档等。由团队而不是老板对争论进行仲裁。我无法形容授权和由团队自行负责项目的成功与否的重要性。

Earl Wheeler，IBM 软件业务的退休主管，告诉我他着手下放 IBM 部门长期集权管理权力的经验：

[近年来]关键的措施是将权力向下委派。这就像是魔术！改进的质量、提高的生产率、高涨的士气。我们的小型团队，没有中心控制。团队是流程的所有者，并且必须拥有一个流程。他们有不同的流程。他们是进度计划的所有者，因此感受到市场的压力。这种压力导致他们使用和利用自己的工具。

和团队成员个人的谈话，显示了他们对被委派的权力和自由的赞同，同时也反映出真正的下放显得多少有些保守。不过，授权是朝着正确方向迈出的一大步，它产生了像 Pius XI 所预言的好处：通过权力委派，中心的权威实际上是得到了加强；从整体而言，组织机构实际上更加融洽和繁荣。

最令人惊讶的新事物是什么？数百万的计算机

每位我曾交谈过的计算机带头人都承认，对微型计算机革命和它引发的塑料薄膜包装软件产业感到惊讶。毫无疑问，这是继《人月神话》后二十年中最重要的改变。它对软件工程意味着很多。

微型计算机革命改变了每个人使用计算机的方式。Schumacher 在 20 年前，陈述了面对的挑战：

我们真正想从科学家和技术专家那里得到什么？我会回答：我们需要这样的方法和设备：

- 价格足够低廉，使几乎所有人都能够使用
- 适合小型的应用，并且
- 满足人们对创造的渴望[21]。

这些正是微型计算机革命带给计算机产业和它的用户（现在已覆盖到普通公众）的杰出特性。一般人现在不但可以买得起自己的计算机，而且还可以负担 20 年前只有国王的薪水才能买得起的软件。Schumacher 的目标值得仔细思考，每个目标达到的程度值得品评，尤其是最后一个。在一个一个的领域中，普通人同专家一样可以应用新的自我表达方法。

其他领域中进步的部分原因和软件创造相近——消除了次要的困难。例如，文书处理方式曾经是很僵化的，合并更改内容需要重新打字，成本和时间都比较高昂。一份 300 页的手稿，常常每 3 到 6 个月就需要重新输入一遍，这中间，人们往往还不断地产生新文稿。另外，逻辑流程和语句韵律的修订很难进行。而现在，文书处理已经非常方便和流畅了[22]。

计算机同样给其他一些领域带来了相似的处理能力，绘画、制订计划、机械制图、音乐创作、摄影、摄像、幻灯、多媒体甚至是电子表格等。在这些领域中，手工操作都需要重

新拷贝大量的未改变的部分，以便在上下文中区别修改情况。现在我们能享受这样的好处，即立刻对结果进行修订和评估，无须失去思维的连贯性，就象分时带给软件开发的好处一样。

同样，新的、灵活的辅助工具增进了创造力。以写作为例，我们现在拥有拼写检查、语法检查、风格顾问、目录生成系统以及对最终排版预览的能力。

最重要的是，当一件创造性工作刚刚成形时，工作介质的灵活性使得对多种彻底不同的可选方案的探索变得容易。这实际上是一个量变引起质变的例子，即时间变化引起工作方式上的巨大变化。

绘图工具使建筑设计人员为每小时的创造性投资展现了更多的选择。计算机与合成器的互联，加上自动生成或者演奏乐谱的软件，使得人们更容易捕获创作的灵感。数字式相机，和 Adobe Photoshop 一起，使原先在暗室中需要数日的工作在几分钟内就可以完成。电子表格可以对大量“what if”的各种情况进行实验、比较。

最后，个人计算机的普遍存在导致了全新创造性活动介质的出现。Vannevar Bush 在 1945 年提出的超文本，仅能在计算机上实现[23]。多媒体表现形式和体验更是如此——在计算机和大量价格低廉的软件出现以前，实现起来有太多的困难。至于现在并不便宜或普遍的虚拟环境系统，将成为另一个创造性活动的媒介。

微型计算机革命改变了每个人开发软件的方式。70 年代的软件过程本身被微处理器革命和它所带来的科学技术进步所改变。很多软件开发过程的次要困难被消除。快速的个人计算机现在是软件开发者的常规工具，从而周转时间的概念几乎成为了历史。如今的个人计算机不仅仅比 1960 年的超级计算机要快，而且它比 1985 年的 Unix 工作站还要快。所有这些意味着即使在最差的计算机上，编译也是快速的，而且大内存消除了基于磁盘链接所需要的等待时间。另外，符号表和目标代码可以在内存中保存，从而高级别的调试无需重新编译。

在过去的 20 年里，我们几乎全部采用了分时作为构建软件的方法学。在 1975 年，分时才刚刚作为最常用的技术替换了批处理计算。网络使软件构建人员不仅可以访问共享文件，还可以访问强大的编译、链接和测试引擎。今天，个人工作站提供了计算引擎，网络主要提供了对文件的共享访问，这些文件作为团队开发的工作产品。客户－服务器系统则使测试用例检入、开发和应用的共享访问更加简单。

同样，用户界面也取得了类似的进步。和一般的文本一样，WIMP 界面对程序文本提供

了更加方便迅捷的编辑方式。24 行、72 列的屏幕已经被整页甚至是双页的屏幕所取代，因此程序员可以看到所作更改的更多上下文。

全新的软件产业——塑料薄膜包装的成品软件

在传统软件产业的旁边，爆发了另一个全新的产业。产品以成千上万，甚至是数百万的规模销售。整套内容丰富的软件包可以以少于开发成本的价格获得。这两个产业在很多方面都不同，它们共同存在着。

传统软件产业。在 1975 年，软件产业拥有若干可识别的、但多少有些差异的组成部分，如今他们依然存在：

- 计算机提供商：提供操作系统、编译器和一些实用程序
- 应用程序用户：如公共事业、银行、保险、政府机构等，他们为自己使用的软件开发应用程序包。
- 定制程序开发者：为用户承包开发私用软件包，需求、标准和行销步骤都是与众不同的。
- 商业包开发者：那个时候是为专业市场开发大型应用，如统计分析和 CAD 系统等。

Tom DeMarco 注意到了传统软件产业的分裂，特别是应用程序用户。

我没有料到的是：整个行业被分解成各个特殊的领域。你完成某事的方式更像是专业领域的职责，而不仅仅是通用系统分析方法、通用语言、通用测试技术的使用。Ada 是最后一个通用语言，并且它已经慢慢变成了一门专业语言。

在日常的商业应用领域中，第 4 代语言作出了巨大的贡献。Boehm 说，“大多数成功的第 4 代语言是以选项和参数方式系统化某个应用领域的结果。”这些第 4 代语言最普遍的情况是带有查询语言、数据库－通讯软件包的应用生成程序。

操作系统世界已经统一了。在 1975 年，存在着很多操作系统：每个硬件提供商每条产品线最少有一种操作系统，很多提供商甚至有两个。如今是多么的不同啊！开放式系统是基本原则。目前，人们主要在 5 大操作系统环境上行销自己的应用程序包（按照时间顺序）：

- IBM MVS 和 VM 环境

- DEC VMS 环境
- Unix 环境，某个版本
- IBM PC 环境，DOS、OS-2 或者 Windows
- Apple Macintosh 环境

塑料薄膜包装的成品软件产业。对于这个产业的开发者，面对的是与传统产业完全不同的经济学：软件成本是开发成本与数量的比值，包装和市场成本非常高。在传统内部的应用开发产业，进度和功能细节是可以协商的，开发成本则可能不行；而在竞争激烈的开发市场面前，进度和功能支配了开发成本。

正如人们所预期的，完全不同的经济学引发了非常不同的编程文化。传统产业倾向于被大型公司以已指定的管理风格和企业文化所支配。另一方面，始于数百家创业公司的成品软件产业，行事自由，更加关注结果，而不是流程。在这种趋势下，那些天才的个人程序员更容易获得认可，这隐含了“卓越的设计来自于杰出的设计人员”的观点。创业文化能够对那些杰出人员，根据他们的贡献进行奖励。而在传统软件产业中，公司的社会化因素和薪资管理计划总会使上述做法难以实施。因此，很多新一代的明星人物被吸引到薄膜包装的软件产业，这一点并不奇怪。

买来开发——使用塑料包装的成品软件包作为构件

彻底提高软件健壮性和生产率的唯一途径，是提升一个级别，使用模块或者对象组合来进行程序的开发。一个特别有希望的趋势是使用大众市场的软件包作为平台，在上面开发更丰富和更专业化的产品。如使用塑料包装的数据库和通讯软件包来开发货运跟踪系统，或者学生的信息系统等。而计算机杂志上的征文栏目提供了许许多多的 Hypercard Stack、Excel 模板、Minicard 的特殊 Pascal 函数以及 AutoCad 的 AutoLisp 函数。

元编程。Hypercard Stack、Excel 模板、Minicard 函数的开发有时被称为*元编程（metaprograming），*为部分软件包用户进行功能定制的过程。元编程并不是新概念，仅仅是重新被提出和重新命名。在 60 年代早期，很多计算机提供商和信息管理系统（MIS）厂商都拥有小型专家小组，他们使用汇编语言的宏来装备应用编程语言。Eastman Kodak 的 MIS 开发车间使用一种用 IBM 7080 宏汇编定义的自有应用语言。类似的，IBM 的 OS/360 队列远

程通讯访问方法中（Queued Telecommunications Access Method），在遇到机器级别指令之前，人们可以读到若干页汇编语言的通讯程序。现在元编程人员提供要素的规模是宏的若干倍。这种二级市场的开发是非常鼓舞人心的——当我们在期待 C++类开发的高效市场时，可重用元程序的市场正在悄无声息地崛起。

它处理的确实是根本问题。因为包开发现象并没有影响到一般的 MIS 编程人员，所以对于软件工程领域并不是很明显。不过，它将快速地发展，因为它针对的正是概念结构要素打造的根本问题。成品软件包提供了大型的功能模块和精心定制的接口，它内部的概念结构根本无需再设计。功能强大的软件产品，如 Excel 或者 4th Dimension 实际上是大型的模块，而且它们作为广为人知、文档化、测试过的模块，可以用来搭建用户化系统。下一级应用程序的开发者可以获得丰富的功能、更短的开发时间、经过测试的组件、良好的文档和彻底降低的成本。

当然，存在的困难是成品软件是作为独立实体来设计，元程序员无法改变它的功能和接口。另外，更严肃地说，对于成品软件的开发者而言，把产品变成更大型系统中的模块似乎没有什么吸引力。我认为这种感觉是错误的，在为元程序员开发提供软件包方面，有一个未开拓的市场。

那么需要什么呢？我们可以识别出四个层次的软件用成品户：

- 直接使用用户。他们以简便直接的方式来操作，对设计者提供的功能和接口感到满意。

- 元程序员。在单个应用程序的基础上，使用已提供的接口来开发模板或者函数，主要为最终用户节省工作量。

- 外部功能作者，向应用程序中添加自行编制的功能。这些功能本质上是新应用语言原语，调用通用语言编写的独立模块。这往往需要命令中断、回调或者重载函数技术，向原接口添加新功能。

- 元程序员，使用一个和多个特殊的应用程序，作为更大型系统的构件。他们是需求并没有得到满足的用户群。同时，这也是能在构建新应用程序方面获得较大收获的用法。

对于成品软件，最后一种类型的用户还需要额外的文档化接口，即元编程接口（metaprogramming interface，MPI）。这在很多方面提出了要求。首先，元程序需要在整

个软件集的控制之下，而每个软件通常假设是受自己的控制。软件集必须控制用户界面，而应用程序一般认为这是自己的职责。软件整体必须能够调用任何应用程序的功能，就好像是用户使用命令行传递参数那样。它还应该像屏幕一样接受应用程序的输出，只不过屏幕是显示一系列字符串，而它需要将输出解析成适当数据类型的逻辑单元实体。某些应用程序，如FoxPro，提供了一些接收命令的后门接口（wormhole），不过它返回的信息是不够充分和未被解析的。这些接口是对通用解决方案需要的一个特殊补充。

拥有能控制应用程序集合之间交互的脚本语言是非常强有力的。Unix 首先使用管道和标准的 ASCII 字符串格式提供了这种功能。今天，AppleScript 是一个非常优秀的例子。

软件工程的状态和未来

我曾问过北卡罗来纳州大学化学系的系主任 Jim Ferrell 关于化学工程的历史以及和化学的区别的问题，于是他作了一个 1 小时的出色即兴演说，从很多产品（从钢铁到面包，到香水）的不同生产过程开始。他讲述了 Arthur D. Little 博士如何在 1918 年在麻省理工学院建立了第一个化学工程系，来发现、发展和讲授所有过程的共有技术基础。首先是经验法则，接着是经验图表，后来是设计特殊零件的公式，再后来是单个导管中热传导、质量转移和动量转移的模型。

如同 Ferrell 故事所展现的，在几乎 50 年后，我仍被化学工程和软件工程之间的很多相似之处所震动。Parans 对我写的关于*软件工程（software engineering）*的文章提出了批评。他对比了电气工程和软件领域，觉得把我们所做的称为“工程”十分冒昧。他可能是正确的，这个领域可能永远不会发展成像电气工程那样的工程化领域，拥有精确的数学基础。毕竟，软件工程就像化学工程一样，与如何扩展到工业级别处理过程的非线性问题有关。而且，和工业工程类似，它总是被人类行为的复杂性所困扰。

不过，化学工程的发展过程让我觉得“27 岁的”软件工程并不是没有希望的，而仅仅是不够成熟的，就好像 1945 年的化学工程。毕竟，在二次世界大战之后，化学工程师才真正提出闭环互联的连续流系统。

今天，软件工程的一些特殊问题正如第 1 章中所提出的：

- 如何把一系列程序设计和构建成*系统*

- 如何把程序或者系统设计成健壮的、经过测试的、文档化的、可支持的*产品*
- 如何维持对大量的*复杂性*的控制

软件工程的焦油坑在将来很长一段时间内会继续地使人们举步维艰，无法自拔。软件系统可能是人类创造中最错综复杂的事物，只能期待人们在力所能及的或者刚刚超越力所能及的范围内进行探索和尝试。这个复杂的行业需要：进行持续的发展；学习使用更大的要素来开发；新工具的最佳使用；经论证的管理方法的最佳应用；良好判断的自由发挥；以及能够使我们认识到自己不足和容易犯错的——上帝所赐予的谦卑。