X-Programmer 总第3期
软件工程的播种机
www.umlchina.com
非程序员 2001 7

目录

方法

过程

本期审稿：Think，Mirnshi，June

联系方法

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

播种机：http://www.umlchina.com

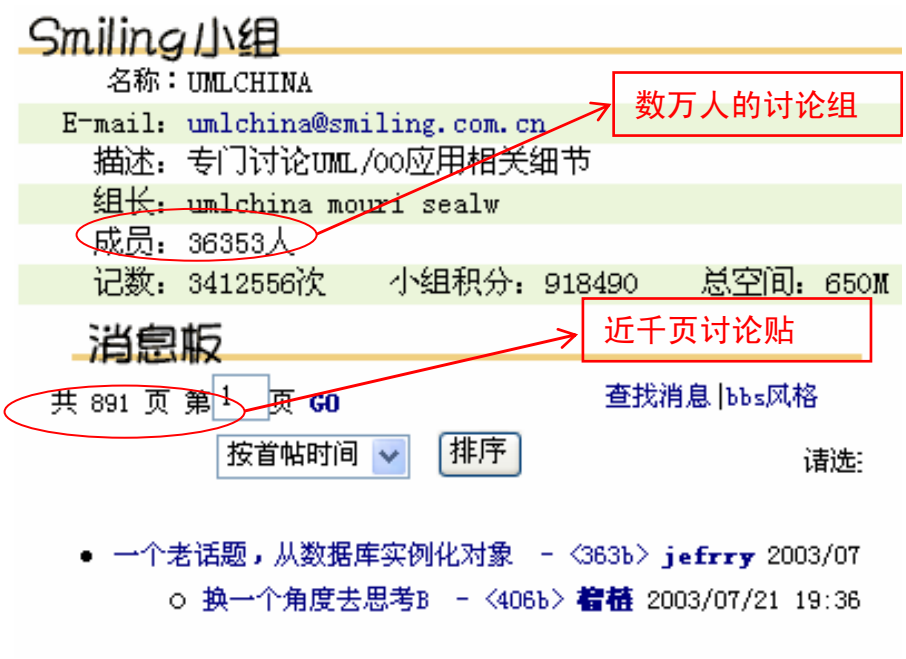
征稿

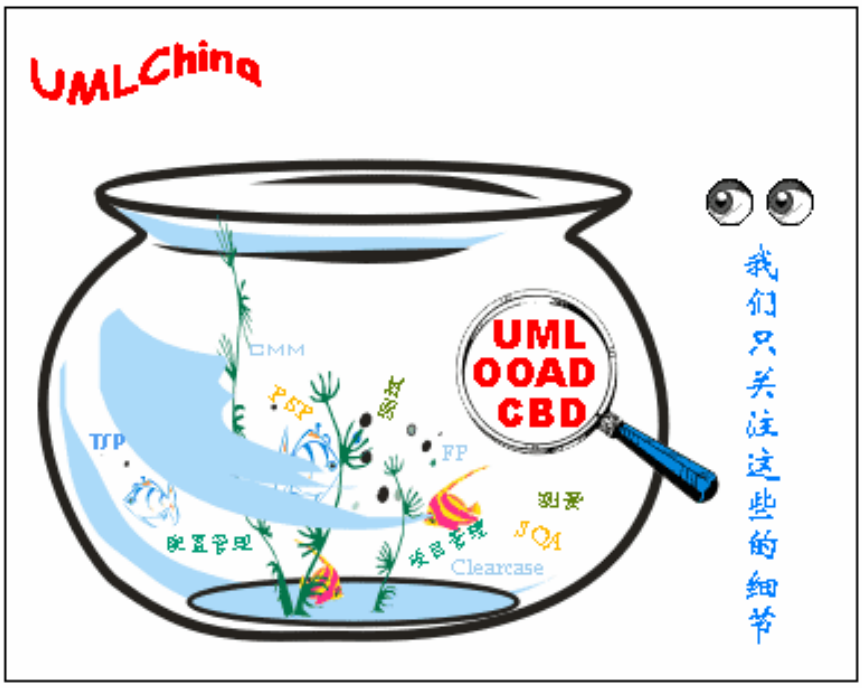
关于需求，设计，构造，测试，维护，配置管理，管理，过程，工具，质量... 原创或翻译均可。

请点击查看详情>>

征稿：
Using Patterns to Integrate UML Views
Interface Hall of Shame
Looking Back, Looking Ahead

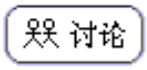
请点击查看详情>>





UMLChina提供以下服务：团队内训，项目指导

模式讨论 FAQ



Doug Lea著，Qian x j译

Doug Lea 维护。有什么意见、评论请发邮件： mailto:[dl@cs.oswego.edu] 。最后更新于 2000.11。

这不是通常所理解的 FAQ(常见问题解答)。本文的话题提炼并精简于模式讨论列表，格式为问答方式。条目列表内容的取舍只是反映维护者的个人观点。这些 FAQ 会不定期的更新。

有关模式(patterns)的更多信息，包括：在线模式的链接，模式的论文，有关模式行为的书籍描述，讨论会议的日程，有关模式的邮件列表，请参见The Patterns Home Page（http://www.hillside.net/patterns）。

1、“模式”这个术语为什么没有一个很好的定义？

为什么大部分工程术语没有一个很好的定义？“模式”一词跟“对象”一样，看上去至少还过得去。似乎没有人介意使用这个短语，“在某一个情景下的问题解决方案”，。然而，这个短语会产生混淆。解释这些术语，对你会有所帮助：

- 场景（context）是指模式应用其中，并可重复出现的一些情形。
- 问题(problem)是指一套发生在场景中的约束(forces) -- 目标(goal)和限制(constraints)。
- 解决方案（solution）是指一种规范设计范式或者设计规则。人们可以用来解决这些约束(forces)。

有关更加广泛的模式定义的讨论，可以参见：

The Patterns Home Page（http://www.hillside.net/patterns）和

WikiWikiWeb（http://c2.com/cgi/wiki?WelcomeVisitors)。

2、难道不能用一个比“模式”更好的词来描述这些事物吗？

你可以用任何自己喜欢的词来称呼他们，但是现在要改变这个叫法已经太晚了，多数人已经这样叫了。

3、设计模式(design pattern)跟模式(pattern)有何不同？

模式的概念非常广泛，它能够适用于各种场景。

“四人帮”(Gang of Four -- Gamma, Helm, Johnson, and Vlissides)所著的《设计模式》一书几乎完全专注于处理微观结构(micro-architecture)(也被称为对象结构(object structures))的模式。这

些结构是在面向对象的开发中形成的一些静态和动态对象(和/或它们的类)之间的关系。术语“*设计模式*”开始涉及这一类模式。它们已经成为了模式著作中最常见的模式了。

有人错误的把术语“*设计模式*”用到任意的对象结构上，甚至是那种在任何意义上都不能成为模式的东西。请不要这样做。

4、模式还可以在什么地方得到应用？

与软件相关的现有例子包括：

程序设计习惯用法（idiom）

例如,C++中的嵌套类的特殊应用,Java 中的接口,Smalltalk 中的层叠调用(cascaded calls),…。

编码习惯用法（idiom）

例如在 C 中的习惯用法：while (*dest++ = *src++)。

数据结构

例如树（trees）和缓存(buffers)。

算法

例如，那些关于并行处理(parallel processing)的模式。

协议

例如，那些并发对象(concurrent object)系统的应用模式。

新框架的开发（一系列扩展类）

例如，那些关于用户界面（UI）工具包的模式。

现有框架的应用

例如，OpenDoc, JavaBeans，…。

分析模型

例如，那些用会计学规则处理的模式。

系统结构

例如，黑板和代理结构。

开发组织

例如，开发团队的组织与发展。

开发过程

例如，面向对象分析设计的步骤和策略。

另外，模式已经被应用到了软件开发以外的一些领域。

5、模式和编码习惯方法（coding idiom）有何不同？一个设计？一个或者多个 OMT 或 UML 图？一个用例（use case）？一个协议？一种算法？一种试探法（heuristic）？一个结构？一种编码规则？一种编码风格？一种开发方法？

一个模式可能围绕着上述中的某一项 ，但是单独的一项不能构成一个模式。一个模式描述了在一个假定的开发场景中这些项是如何被应用，以及为什么被应用，并提供应用它们的指导。

6、模式和类有何不同？是一个可重用的组件？一个参数表示的（模板）类？一个类库或者包？一个框架？一个工具？一个代码产生器？

一个模式并非是一个实现（implementation）。它描述了：什么时候、为什么、并且怎样，去创建一个实现或者其它工程产品。

有一些（并非很多）解决方案有必要通过“实现”来描述（比如类、框架、工具、或者其它什么的）。实现可以告诉开发者，他们将需要知道或者需要做的全部东西。即便如此，代码本身并非模式。

7、模式跟类似于“如何做”这种向导有何区别？

模式中描述的解决方案可以是由一些精简的短句表示的步骤，这些步骤就像那些“如何做”向导和烹饪的菜谱中的步骤。但是这些步骤只是形成了模式的一方面。并且，模式追求更广的适用范围和场景的普遍性，同时在明确和解决约束问题上比那种典型的“如何做”向导更具权威性。

8、模式的研究跟专有领域的软件体系结构、软件复用以及其它领域的软件工程有何区别？

看来有一些重叠。

9、我在哪能找到有关 XXX 方面模式的出版物或者在线资料？

没有一个为模式而设立的集大成之地，但是查找这些资料并不是很难。

下面是一些启蒙点：

- Hillside Patterns Home Page ；
- Linda Rising 的书 《模式年鉴》（The Pattern Almanac） （也是模式的前辈，模式手册），囊括了大部分已发表模式的参考；
- Wiki ；
- Pattern Depot ；
- Open directory ；
- 还有你喜欢用的搜索引擎。

10、克里士多夫.亚历山大（Christopher Alexander）是谁？

亚历山大是一个建筑师（是建造业的，并非软件业），他发明了模式。有关他的简单传记以及其相关文章、网页的链接可以在此处找到：Nikos Salingaros's Notes on Christopher Alexander。

11、模式最佳的形式是什么？

拿出你所精选的。多数的亚历山大模式表现为以下形式：

假如 你发现你自己在 **场景** 中

对于例子 **例子**，

和 **问题**，

使其承担 **约束**

那么 因为某些 **理由**，

应用 **设计形式和（或）规则**

来构建 **解决方案**

推导出 **新的场景** 和 **其它模式**

注，原文：

IF you find yourself in CONTEXT

for example EXAMPLES,

with PROBLEM,

entailing FORCES

THEN for some REASONS,

apply DESIGN FORM AND/OR RULE

to construct SOLUTION

leading to NEW CONTEXT and OTHER PATTERNS

还有许多风格的变体。现存的讨论模式的书中没有两本使用完全相同的形式。可选的有包含纯粹的叙述性的波兰形式)。或许最流行的形式（在设计模式一书中使用）与此相反，他们开始于设计形式和（或）设计规则，然后才描述问题，场景，和他们怎样应用的例子。

超越模式不同的形式，其结构和内容的共有要求包括：

最佳实践 的描述

或者至少*被广泛认可的实践* 。有些人把模式看作一个为了构建权威性的软件工程手册的步骤。

适度的普遍性

表明模式是会*重复出现* 的。这就要求你应该从几个已知的应用中抽象出东西来。在涉及二选一的模式时，这可能要求提及模式*无法* 应用的情况以及选择性的参考模式。

范围

应用模式的场景要被完整的描述。恰当时，还可以包括典型的导致应用此模式的其它模式。

建设性

模式这个词语的意思就是：一个让人们建立解决方案的实例的方法。某些情况下，可能需要具有表现基本关系的一系列图形。另外，可能还需要一系列步骤，以便模式的使用者可以按部就班，依照解决方案中应该出现的步骤的描述和（或）例子进行下去。

完整性

所有相关的约束都被描述到。

实用

表明解决方案（solution）成功的解决了焦点问题（约束）；或者当它们仅仅被部分的解决，并且（或者）当它们为可能应用的相关模式引入了新的约束、参考时。

样例

解决方案步骤的举例说明，以及现有软件中的应用实证

适当的抽象

例如，“加入另一个中间层次”可能是有益的探究，但是它可能由于过于泛泛而且有多种可理解意义而不能成为好的模式。

少一些创意

新的解决方案不是模式。（虽然导致产生模式的描述文档的“提取，综合，和改写”也许有新意。而且，虽然模式也可能导致产生新颖的用法。）

合适的称谓

简明的、描述性的名称可以有助于为开发人员提供共用的词汇表。

清晰明了

好的叙述和风格很容易让人们判断出这个模式是否可以很好的工作并且怎样工作，因而他们自己就不必重新发明它（模式）了。

12、什么是约束（force）？

约束的概念概括了软件工程师用来校准设计和实现的各种准则。例如，在计算机科学中的经典算法研究中，主要约束(force)要解决的是*效率*（时间复杂度）。然而，模式处理的是大量的、难以度量的并且相互冲突的目标集，以及在你曾经创造每个作品的开发中遇到的约束。比如：

正确性

- 解决方案的完整性和正确性
- 静态类型的安全，动态类型的安全
- 多线程的安全，生存期
- 容错性，事务能力
- 保密性，坚固性

资源

- 效率：执行性能，时间复杂度，消息发送数，带宽需求
- 空间的利用：内存单元数，对象数，线程数，进程数，通信通道数，处理器数，...
- 递增量（需要时随选性 on-demand-ness，此句有疑问）
- 策略动态：公平性，平衡性，稳定性

结构

- 模块性，封装性，耦合性，独立性
- 可扩充性：子类化能力，协调性，演化能力，可维护性
- 重用能力：开放性，整合性，轻便性，可嵌入能力
- 场景独立性（内聚性）
- 协同性
- ...其它“能力特性”和“质量因素” *##原文为：... other “ilities”and “quality factors”译注：我向原文编者发email询问，得到回复：”ilities” is a cute abbreviation for “reliability”, ”traceability” , and several other software quality terms that end in “ility”. ##*

解释

- 易懂性，小巧程度，简单度，优雅性
- 实现的错误倾向
- 与其它软件共存
- 可维护性
- 在开发进程上（中）的影响 ## on/of ##
- 在开发团队的组织和发展上（中）的影响
- 用户参与上（中）影响
- 生产力，日程安排，花费上（中）的影响

惯用法

- 习惯的规范
- 人为因素：学习能力，撤销能力，...
- 对于变化中的世界的适应性
- 审美

- 医疗和环境的影响
- 社会，经济和政治上的影响
- ...其它的有关人类存在的影响

Tres Seaver 补充道：惯用法远比软件要古老：在建筑行业，或者在机械/土木工程中，一个设计实体存在于一个相互作用的物理约束系统的关系中。对于某个约束问题没有得到解决的设计，连同整个的系统都会失败（例如，塔科马纽约湾海峡大桥坍塌，是因为没有解决大风导致的颤动；德克萨斯州 A&M 的大火，是因为没有考虑移动工人引起的负载*##原文： the bonfire at Texas A&M failed to allow for the loads induced by moving workers ##*）。延伸开去，设计必须满足复合体相互作用的需求和有时无法预料的情形；术语“约束”就是扩展开来包含这些情景。

13、什么是约束的分辨度？

亚历山大的模式描述包含了一个思想，就是模式应该描述一种平衡的约束。（甚至是亚历山大也被人（甚至是他自己）批评过，他一直没能提出令人信服的方法。）在计算机科学的分析与算法中，与此相同的概念是最优解的例子，但是它们却被用到前面提到的各种难以衡量的问题的约束描述中。

分析证明某个最优地解决了约束的解决方案是不可能的。（事实上，很难定义此处“证据”的概念，或者甚至很难看出这种证据有什么用处。）另一方面，也很容易得出“就是如此”的这种谎言，它提供了错误的或欺骗性的原理作为解决方案。甚至一些模式作者有时候不能完全理解为什么一个解决方案如此有效，或者不能评定它的适用范围。就像 Ralph Johnson 的文章中所说：

> 通常很难把一个模式解决的问题描述清楚。你可以断定它是一个模式，因为你经常看到它，而且你也知道它是一个好的模式，因为介绍它会给世界带来好处。但是当你注视世界时，你发现说出“为什么事情会如此”是很困难的。我想，弄清一个模式解答的问题的方法是：当我们设计东西时观察我们自己。什么是触发我们使用模式的条件？
>
> 我想，亚历山大为什么并不总是描述模式的问题的主要原因是：他并不是总是懂得的。那么，为什么 GOF（4 人小组）的书（设计模式）描述问题的不尽完善也是当然的。这并非形式上引导作者忽略问题，而是他们对问题的理解引导他们会写出何种形式。

因为这些原因，模式社区需要下列支持的论据：

经验证明是优的

那个解决方案已经在多个场景被合适的使用了。“3 次规则”有时候被当作准则来用：不要认定某一些东西就是模式，除非你知道它在 3 个独立的地方被应用。

对比

对于其他已知的解决方案或实践的比较。这些可能会包含一些证明了弱点或失败的例子，因为恰当的解决方案还没有被应用。

独立的原创作者

模式不应该由那些第一次发明或实现它们的人们记述。

评论

让其他人批评，包括让密切相关的领域中的或不密切相关领域中的人们批评。一个模式评论的流行的形式（用在模式语言程序(PLOP)的讨论会）是Writer’s Workshop。

只有提供了这些证据，一个模式有时候才可以称为“原型模式”（proto-pattern）——一个候选模式。

14、什么是无名质量（Quality Without A Name(QWAN)）？

对于这个问题没有又短又好的答案。你应该看看“持久的构建方式”（The Timeless way of building）。

十分讨厌的是模式的热衷者喜欢伪称亚历山大从来未写过关于 QWAN 的东西。十分古怪的是一些人喜欢伪称 QWAN 让模式同某些神秘事物紧密联系，听起来古怪的是形而上学的人鼓吹你要注意名称。

15、模式可否给出否定的形式，告诉你什么不该做？

也许理想状态下是不需要的 — 一套好的模式可以帮助你丢掉你提出来的许许多多糟糕的设计（有时被称为反模式：antipatterns），同样将一个给定的模式应用于所有的场景中，这也是不合适的。但是有一些想法非常有必要明确的指出来。一种方式是在模式描述中包含类似“常见的陷阱和缺陷”的章节。蹩脚解决方案（不相称的方案）的描述也能成为一个好的解决方案中的动机、基本原理和约束的组成部分。模式也可以描述从糟糕的解决方案到好的解决方案的改进过程（有时体现在“以前/以后”或者“修正”这样的章节中）。

16、模式是否可以介绍一些可选的解决方案而非单一的方案呢？

也许理想状态是不需要的 -- 因为每个解决方案应该紧密地关联于它的最佳应用场景。但是有时候那太难了。提及可选的解决方案还是比什么都不提及要好，因为可以建立起框架，通过探索那些不同解决方案的约束来进一步完善这个模式。甚至当它们不同时，在同一介绍中，是否组合一套模式，来共用多数场景和约束，这也是一个格式上的问题。

17、为什么费心写的模式归根结底就如同我祖母给我的建议？

因为一些模式非常优秀和有用，甚至连你的祖母都知道它们了。把它们记下来，把建议的场景、评价、和暗示变得比你的祖母所可能做的更加清晰明了。

18、所有的模式必须像某些模式一样[低级或高级、通用或特殊、抽象或具体]吗？

当然不是。

19、模式的理论基础是什么？

没有通常意义上的正式基础。模式表现的是滋生于各种理论和经验基础的设计思想。另外，许多指导模式的思想产生于跨越不同工程领域的经典或不是那么经典的“设计理论”著作。（详见在模式主页列出的书目清单）

20、模式可否用一些详细的形式和符号表示？

欢迎你去尝试，但要牢记的是，如果忽略了描述场景，忽略了要解决的问题，忽略了解决方案合理性的证明，忽略了构造或实现的指导方针，或者忽略了与其他模式的关系，那么这些形式符号化的设计规则或设计表现就不是模式了。

21、我为什么要使用模式？

和你重用好的代码的原因相同：一些人在理解场景（context）、约束（force）、解决方案(solution)中，相对你已投入或者想要投入的努力，他们付出的更多，你应该从他们的知识和经验中获益。此外，因为模式的适应性、可重用性要高于代码，你可以在由于特殊的场合而不能重用已有的组件的情况下利用模式建立软件。

模式还向开发者提供了一套通用的名称和概念来处理通常的开发问题，这样就加强了沟通。

22、拥有一些基于模式的 CASE 工具应该是个好事吧？

也许吧，但模式是围绕于人和人之间的交流的。假如这个媒介增强了同他人的沟通，那是很好的。如果它仅仅为了借助计算机提高模式的执行度，那就不值了。

23、一种模式语言和一套模式有什么不同？

一种模式语言是一套相互关联的模式，[它们]共用相同场景，并且可能分类。

这个术语源由于亚历山大。亚历山大的“语言（language）”这个术语是不合惯例的，但却不是错误

的。假如你以非正规角度来看，对它超形式化，模式条目其实就是“产品规则”。假如你记得你[学过]的自动化理论，你会记起那些规则是描述反复可列举语言的唯一途径。

24、为什么没有更多的关于任意的(WHATEVER)模式？

因为你还没有写出它们来。若你感兴趣，你很可能懂得一些东西，那么当你懂得了一些东西，你就能够写模式了。

25、我怎样开始写模式？

下面是一些通常的建议：

- 避免为写模式而写模式。
- 找到一些你熟知的并且多次发生的事件和（或）记述的内容作为模式素材。或者相反，从现有的软件中挖掘出新的可能的模式。
- 检查是否其他人已经写出了相似的模式。
- 注意模式的质量而非数量。
- 领会为什么模式会存在或被应用。
- 找出记录它的合理格式。
- 散发给其他人（例如，通过网页，模式讨论组，或者提交给邮件列表），并听取意见。
- 提交给媒体，例如 PLOP，在作家工作室（writer’s workshop）那儿可以得到评论。
- 不断的反复和提炼。

另见：Ward Cunningham's Tips for Writing Pattern Languages

26、我怎么知道我的某个主意（或设计、构造）是一个模式？

试试把它当作模式来写。Writing it as pattern.

27、现在有多少模式？

有些人认为，相对于几乎每个人都该知道的模式，还没有被发现的模式很少。某些人认为在更多的领域里，还有非常多的模式应该构建出来。这两个观点都可能正确。试着写一些更多的模式，让我们把它们找出来。

28、已有的众多模式不会产生在查找，分类，检索，使用和维护时的问题？

也许。

29、我能否在*[一些详细的面向对象的分析和设计的方法]*中应用模式？

大概可以。虽然假设你这样做了，你可以不必拘泥于很多符号的表示方法。

30、模式的应用是否需要有迭代？

大概，你可能非常幸运，针对你的问题已经有一套完整的模式，模式的每个应用都顺利进行，并走向最终的产品，而且没有回溯。但是人们从未这么幸运过。

31、与模式使用相关的开发过程，有可以推荐的吗？

大概如此：即便我们对此讨论很多，但是我们还是不知道（什么是被推荐的开发过程）。使用了模式，我们才能找出（这是不是要推荐的开发过程）。

32、模式在商业实践和软件行业中会有效吗？

回答同上。（意思就是用了才知道）

33、是不是教人们写出模式要比教他们使用现有的模式更有效呢？

两者都需要。没有哪一个比另一个更需要。

34、在我工作的地方我们怎样把模式的应用制度化？

一般的建议包括：

- 为模式挖掘你拥有的最好的代码。
- 扩充设计文档，回顾实践经验，参与设计***模式***
- 把已有的设计模式用到实际情况中。[Run courses on the use of existing design patterns]
- 在“作家工作室”（writer's workshop）回顾已有的模式。
- 用基于模式风格的模板编写设计文档，这样它们就可以发展为模式。
- 建立一个学习小组，一星期聚会一次讨论模式。一旦爱好者的群体产生了，他们就会要求将其制度化了。

35、为什么亚历山大的模式未被建筑师广泛应用？

这可能并非是单一的原因造成的。人们列举的因素包括：同长期已建立的惯例冲突；同建筑师们的职业文化冲突；经济的因素；Alexander 贯注于人们设计和建造自己的房子的事实；还有，关于模式语言中的特定的模式在实际中究竟有多大用处或多少好处，观点不尽相同。

36、你们能否在一个很大的开发中使用模式？

已经有报导这样做了

37、模式是否被夸大了

当然。这是难免的。

38、模式真的有用吗？

请提出更明确的问题。

Doug Lea

最后更新： 2000.11.7 星期二 美国东部时间 07:27:54

原文链接：http://www.umlchina.com/Pattern/PatternsdiscussionFAQ.htm

KCOM 技术介绍

KCOM 是 KCOM 公司独立开发的一系列开发平台和计算机语言技术的总称。
它包括：KCOM 组件模型，KCOM Basic 语言，KCOM Stage 运行环境，KCOM Space 开发平台四项技术。

KCOM 组件标准：
KCOM 组件标准是一个完善的组件标准，可以自定义属性，方法和事件，并使用事件驱动的方式编程。同时源代码方式存储有利于在网络上传输。

KCOM BASIC：
语法结构完善，易学，可以使用中文进行编程，例如使用中文变量名称，中文方法名称，属性名称等等。独有组件运算功能能够实现代码的可视化。

KCOM Stage：
KCOM Stage 是 KCOM 组件的解释器，由于采用了代码的结构化存储，在解释之前不需要对语句进行扫描和文法处理，同时采用了高效的寻址方式，所以比一般的脚本语言的解释效率要高。

KCOM Space：
能够方便的编写 KCOM 组件和代码，是一个可视化开发平台，具有丰富的界面排版，代码编写和代码调试功能。

KCOM Space 以其自定义的组件标准(KCOM 组件标准)为核心，完整地实现了一门计算机语言（KCOM Basic)、一个完整的组件化开发平台（KCOM Space）和一个高效的虚拟机（KCOM Stage)。作为国产软件在高端领域的突破，KCOM 表现出了与世界同步的技术水准。由于采用了全面组件化的思想，抛弃了传统的设计方法，全面革新了开发平台从底层的语言到上层的操作界面的设计。

KCOM 公司的产品在 2000 年中国国际软件博览会上获得创新奖。并受到倪光南院士的高度评价。

KCOM 公司合作意向：

KCOM 公司在长期对于开发平台和计算机语言技术的关注与不断开发的基础上，形成了自己独特的技术特长与优势。在如今的软件领域，软件和开发平台日益紧密成为趋势，其中突出地表现在：

软件与基于该软件的二次开发平台：如 MS Office，Notes，GIS 应用软件及开发平台，3D Max，AutoCAD，Director 等等应用软件。 一些稍小的软件如 InstallShield 也在其中使用了自己定义的解释语言，以提供强大的扩展功能。

企业流程快速建模与软件生成工具：此类工具在国外的企业应用解决方案市场中被大量使用。而国内的系统集成企业正在开始开发和形成自己的解决方案与定制工具。

嵌入式企业应用前端设备：该设备是我们所看到的一个巨大的市场机会，基本原理在于软硬件一体的显示与操作前端，采用 NC 体系，运行企业应用。

在以上几个领域中，KCOM 公司都能够以自己长期的积累和技术特长，与合作方展开广泛的技术合作，希望能够来信探讨：zhangiii@163.net

http://www.kcomspace.com.cn/

Abstract Class（抽象类）模式 —类行为型模式

讨论

Bobby Woolf 著，透明 译

意图

为一个类体系（hierarchy）定义接口，并将具体实现交给子类。抽象类（Abstract Class）在保留类的多态性的同时，让子类可以重新定义接口的实现。

别名

利斯科夫置换原则（Liskov Substitution Principle, [LW93]）、契约设计（Design by Contract, [Meyer91]）、基类（Base Class, [Auer95]）、模板类（Template Class, [Woolf97]）。

动机

设想我们要实现一个简单的算术功能。每个应用程序都需要使用整数和浮点数这样的简单数字量，并且需要在这些简单数字量上进行加、减、乘、除这样的简单算术。

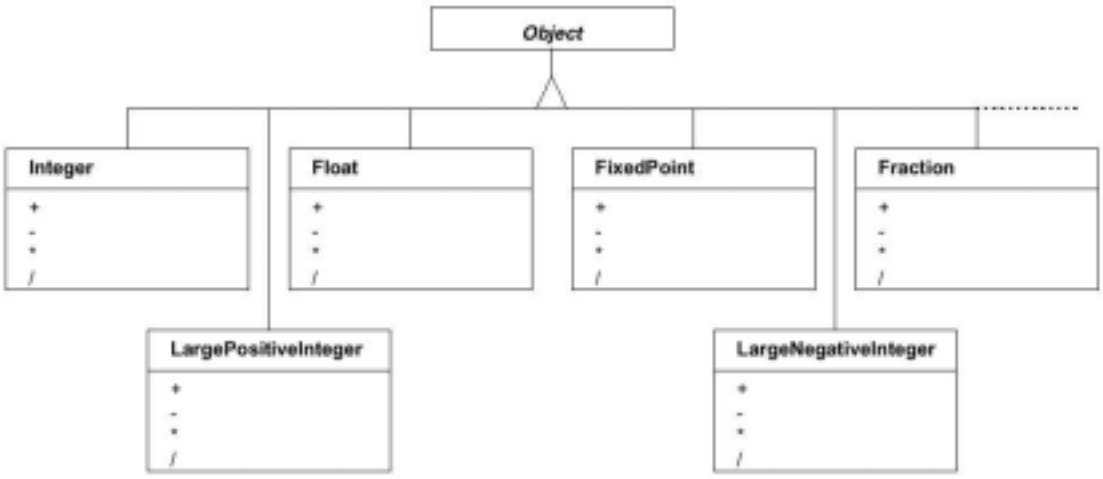
处理这样简单数学问题的一个明显的方法是将它交给CPU去完成。现在的CPU都有对整数和浮点数做简单算术的内建指令。这是做这些计算最有效的途径。

但还有一个问题摆在我们面前：并不是所有的数字量（number[1]）都能用CPU的整数和浮点数来表示。对于CPU来说，整数的范围是有限的，浮点数的精度也是有限的，并且在二进制和十进制浮点数之间转换时还会有精度的损失。

一个健壮的面向对象的数字量体系(framework)应该尽可能地利用CPU的效率。但是，为了使这个体系更加健壮，它必须在需要的时候克服CPU的限制。它应该能够为整数提供一个虚拟的“无限”的范围，包括那些极大和极小的数值。它应该为小数——至少为特定的一部分小数——提供完整的精度，它在进行简单浮点数算术运算时应该不损失精度。它甚至可以进行复数运算，只要先对之进行某些简化。

[1] 译者注：number 这个词在这里不太好译，叫“数据”、“数字”、“数”……似乎都不好，最后决定叫“数字量”。希望各位指正。

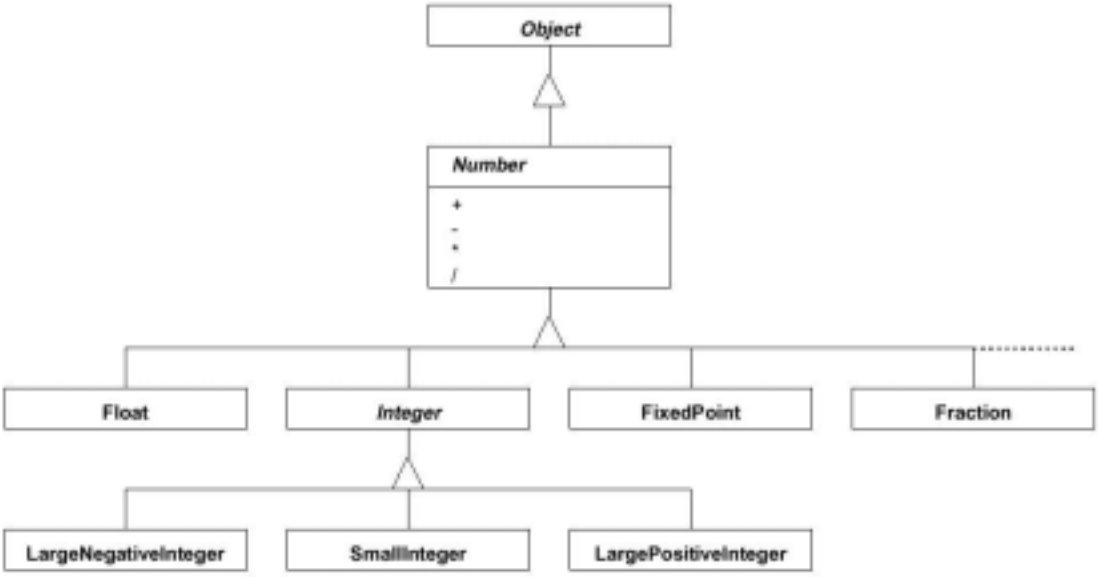
为了达到这些目标，这个健壮的数字量体系将使用不同的类：作为CPU数的Interger和Float、为超过CPU范围的整数准备的LargePositiveInteger和LargeNegativeInteger、为完整精度的小数准备的FixedPoint、为没有舍入（round off）的除法准备的Fraction，等等。这样，这个体系可以进行CPU能直接进行的所有计算，并且还通过使用其他的类表示了CPU不能直接表示的数字量。下面的图对这个体系中使用的类做了一个展示：



对于这些数字量类来说还有一个问题：系统的其他部分不需要也不应该知道它们。系统的其他部分只知道有一些数字量对象，这些对象知道怎样进行算术运算。当系统某个地方的代码中有象“x+y”这样的一个语句时，它并不关心“x是一个Float”或者“y是一个Fraction”这些实现细节。这些代码只知道：x和y是数字量，这些数字量知道怎样进行加法运算。这暗示着：如果有一些数字量对象知道怎样做加法运算，则所有的数字量对象都应该知道这一点。

因此这个数字量体系需要的并不仅仅是不同的数字量类。它还需要清楚的指出哪些类是这个体系中的一部分。它应该要求这个体系中所有的类都能完成最小数量的某些功能，比如加法。另外，这个体系需要用多态的方式提供所有的数字量操作，这样它才能对系统的其他部分隐藏自己使用多种子类的复杂性。

为了实现这些功能，这个体系使用了一个普适（generalized）的超类（superclass）——Number。一个Number的对象可以表现任何数字量，无论是整数、浮点数还是其他。它定义了最少的、所有的数字量都必须提供的功能，比如加法。它不对数字量的结构做定义，也不具体实现任何功能。这些细节都被推迟到Integer、Float等子类中实现。我们用Number作为超类实现Integer，并通过相似的途径得到下面的类图：



现在，所有的数字量类都是Number的子类，它们都已经被定义为可以提供基本的算术运算。使用一对数字量对象的客户（client）知道所有的数字量都可以进行基本算术运算，而不用管这些对象是Number的哪个子类。

我们这里的Number类就是Abstract Class模式的一个范例。在《设计模式》一书中这样描述抽象类：“抽象类（Abstract Class）的主要目的是为它的子类定义公共接口。”[GHJV95，第11页] Number类定义的类型可以用几种不同的方式实现，但所有的实现都将拥有同样的接口，所以客户可以交替使用它们。这样，客户代码将得到简化，它只需要描述它要做“什么”而不是去指定“怎样”做。客户代码甚至可以使用那些未知的、未来的实现，只要这些实现遵循这个公共接口。

Abstract Class模式的关键是一个超类，它定义了一个类型及为这个类型提供不同实现版本的子类。抽象类可以是一个纯接口，这样它的子类必须实现所有的细节；但更实用的方法可能是让抽象类实现对所有子类都适用那一部分功能。这样子类只需要继承已实现的部分并完成未实现的部分。

这个超类被称为“抽象的”是因为它的实现是不完全的，客户程序不能创建它的实例。抽象类的子类——客户可以创建这些子类的实例——被称为“具体的”（concrete class）。[WWW90，第27页]

关键点

一个包含Abstract Class模式的应用体系（framework）有如下的特点：

- 一个超类，它定义了一个类型。
- 一个或多个子类，它（们）实现超类定义的类型。
- 子类间的多态性（polymorphism），因为它们共享超类定义的接口。

这个体系还可能对本模式做一些变化：

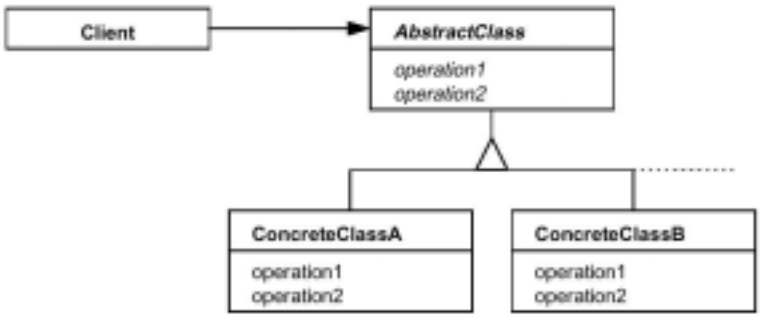
- 超类可能提供一部分但不是全部的实现。
- 超类可能提供完整的实现，这将是一个最小化的默认实现。
- 超类可能在定义接口的同时也定义变量（state）。
- 子类可能扩展超类定义的接口，在其中加入新的功能。但是，这些扩展的接口不能与其他子类形成多态。

适用性

在以下情况使用Abstract Class模式

- 一个体系需要几个子类，这些子类拥有相同的接口，或者它们的接口相交迭而形成一个公共的核心接口。
- 公共接口应该在一个地方定义，这样所有的类都知道它们必须遵循这个接口，客户也知道应该使用这个接口。
- 一个体系需要有可扩展性，你可以在未来向其中添加新的子类而不必改变已有的超类或者客户代码。

结构



参与者

- **AbstractClass (Number, Integer)** —— 抽象类
  - 为所有具体类ConcrenteClass定义公共接口。
  - 不定义变量，也不做实现，除非它对所有具体类（concrete class）——包括未来的具体类——都将是通用的。
  - 本身可能是另一个AbstractClass的子类。
- **ConcreteClass (FixedPoint, Float, Fraction, LargeNegativeInteger, LargePositiveInteger, SmallInteger)** —— 具体类

- 某个AbstractClass的直接子类或间接子类。
- 对继承自AbstractClass的接口进行实现。
- 在实现接口的过程中，声明需要的变量。

- **Client** —— 客户
  - 通过AbstractClass的接口与ConcreteClass的实例协作。

协作

- 客户使用抽象类接口与对象交互，而这个对象可能是任何一个具体类。
- 具体类的实现依赖于抽象类所提供通用的默认实现。

效果

Abstract Class模式的优点如下：

- *类多态性*。具体类之间是多态的，因为它们都支持抽象类定义的公共接口。这也就是说，客户可以使用任何一个具体类而不必管它究竟是哪个具体类。这样的公共接口使扩展变得容易，因为客户程序将可以使用尚未实现的具体类，只要这些类遵循公共接口。
- *算法复用*。如果抽象类包含了一些实现细节，那么它通常会以模板方法（Template Method，[GHJV95，第214页]）的形式出现。这样的实现（在一些案例中甚至是变量）必须对所有的具体类——现在的和将来的——都是通用的。这样只需要在抽象类中实现一次，所有的具体类都可以复用它们。

使用Abstract Class模式可能会遇到下面的问题：

- *抽象和具体*。客户通常认为他们可以创建任何类的实例，但对于抽象类，他们不能。客户不应该尝试创建抽象类的实例，而只能创建具体类的实例。我们把本模式中的超类叫做“抽象类”是因为它的实现并不完全，所以客户是不能创建它的实例的；而子类被叫做“具体类”是因为它们的实现是完全的，所以客户可以创建它们的实例。
- *体系单一*。本模式强迫所有的具体类聚集到一个单一的类体系中，它们有共同的超类——AbstractClass。有时候，一个类从类型上看应该属于某个体系，但为了继承一些功能它实际上属于另一个体系。在这种情形下，最好是根据类型将这个类放在合适的体系中，然后让它委托另一个类的实例来实现所需的功能。

  另一个可能导致这个问题的情况是：完全不同的类需要实现同样的操作。你可能让它们拥有公共的超类，并在这个超类中实现所需的操作，这样的做法当然是有效的。[2]但是这样会把这个超类变成所有的

[2] 译者注：本句可能译得有问题，请各位指点。原文如下：The default implementation for this operation is usually defined in the first superclass they all have in common. This in effect makes that superclass an AbstractClass for those subclasses.

类——即使是那些不需要这个操作的类——的抽象类。很显然，要定义这个操作并复用它，继承并不是最好的方法。需要这个操作的类可以将之委托给有这个操作的其他对象，这会是更好的解决方法。

- *过分精确的接口*。有时候一个具体类并不打算实现抽象类规定（specify）的所有操作。比如说，Smalltalk中的Collection抽象类规定了add：和remove：操作，但是Array子类不会实现它们。除非所有的具体类都能够实现一个操作，否则抽象类就不应该规定它。如果一个操作已经被规定了，具体类就无论如何必须实现它。如果具体类不能做出相应的操作，它应该给出一个错误信息。

实现

在实现Abstract Class模式时需要考虑以下几点：

1．*对类进行分解*。通常我们用一个类来实现一个对象，这个类中定义接口并做出实现。这使得这个类所代表的抽象很难被复用。Abstract Class模式建议：一个对象应该用两个类来实现，一个抽象类定义接口，一个具体类实现这些接口。

2．*不要使用成员变量*。通常抽象类不会声明任何成员变量。如果它这样做了，所有的具体类都将被迫继承这些变量。如果某个具体类的实现不需要这些变量，这就是效率上的损失。但是，如果所有的具体类都需要某个变量，而且未来的具体类也可能需要它，那么可以在抽象类中声明它。

3．*通过模板方法实现*。抽象类通常用模板方法[GHJV95，第214页]来实现。抽象类只定义接口，而将实现留给具体类。但是，如果一个操作可以有适用于所有具体类的默认实现，也可以在抽象类中实现它。这样的实现通常是一个模板方法或者一个基本的成员函数。

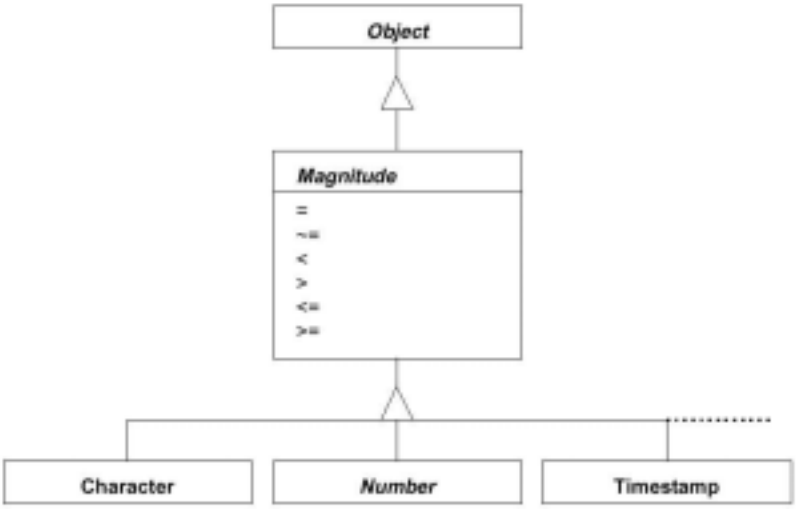
4．*不要在抽象类中加入私有信息（message）*。抽象类为整个体系定义接口。接口中只应该有公有信息，具体类将实现它们。抽象类不需要定义私有信息，通常它也不会这么做。但是抽象类中也可能有私有信息，这通常是在实现模板方法时需要调用的成员函数。

代码示例

Smalltalk中的Magnitude体系是Abstract Class模式的一个精彩的例子。它包括了我们在“动机”一节中讨论的Number体系，因为数字量也是数量[3]。

[3] 译者注：此处将“magnitude”译为“数量”，请注意与“数字量（number）”相区别。

Magnitude类有六个主要的操作：等于（=）、不等于（~=）、小于（<）、大于（>）、小于等于（<=）、大于等于（>=）。Magnitude的例子包括Number（数字量）、Timestamp（时间）和Character（字符），它们都是Magnitude的子类并且懂得Magnitude的操作，如下图所示：



六个方法中的四个是用模板方法实现的：不等于、大于、小于等于、大于等于。它们是根据两个成员函数而实现的：等于和小于。这两个成员函数被推迟到子类中实现。

Magnitude>>= aMagnitude

^self subclassResponsibility

Magnitude>>~= aMagnitude

^(self = aMagnitude) not

Magnitude>>< aMagnitude

^self subclassResponsibility

Magnitude>>> aMagnitude

^(self <= aMagnitude) not

Magnitude>><= aMagnitude

^(self = aMagnitude) or: [self < aMagnitude]

Magnitude>>>= aMagnitude

^(self < aMagnitude) not

因此，Magnitude的子类只需要实现这两个成员函数，然后它就可以得到其它的四个方法了。举例来说，Character类假设字符是以ASCII码形式出现的，所以它按照ASCII码的顺序将字符排序。为了实现这个目的，它将使用一个方法返回一个字符的ASCII值，这个方法的名字可能是asciiValue。

Character>>= aCharacter

^(self asciiValue) = (aCharacter asciiValue)

Character>>< aCharacter

^(self asciiValue) < (aCharacter asciiValue)

这样，抽象类Magnitude为不同的具体类大大简化了实现工作。在上面的例子中，具体类只需要实现两个方法，就可以自由使用另外了四个方法了。

抽象类还简化了客户代码必须理解的接口。客户必须知道它在对两个同一子类型的对象进行比较：两个Character或者两个Number，等等。但是，客户不必知道它们到底是哪个子类型，因为所有子类型的行为都是一样的，所有子类型都有相同的六个比较方法。

为什么这一实现很有使用价值？一个例子是SortedCollection的工作方式。SortedCollection是Collection的子类，它可以对其中的元素进行排序。默认情况下，它认为其中的元素都是Magnitude（并且有相同的子类型），然后SortedCollection使用<=操作完成元素的排序。因此，SortedCollection不必关心这些元素究竟是Character、Number还是Timestamp，因为它们都是Magnitude，它们都将以正确的方式处理<=操作。

已知应用

Abstract Class模式是如此基础，以至于几乎在任何一个多级类体系中你都可以发现它的身影。在这些体系中，超类通常是抽象的，最终类（leaf class）必须是具体的。整个Smalltalk 类体系的根类——Object——是这一语言最根本的抽象类。在JAVA中则是java.lang.Object扮演了这一角色。如果一个类体系是作为体系的根类被人了解的（就象Number、Collection、Stream、Window等等），这个根类几乎一定是一个抽象类。

几乎每一个成文的设计模式——比如《设计模式》[GHJV95]一书中讲到的那些——都使用了一个或多个抽象类。如果需要创建一个对象实例，这些模式通常会建议使用抽象类。举例来说，Composite模式[GHJV95，第107页]用抽象类Component为Leaf类和Composite类定义接口。为了在RealSubject类上使用Porxy模式[GHJV95，第137页]，开发者应该使用抽象类Subject为RealSubject和Proxy定义共享接口。当一个模式说某个参与者为几个子类“定义一个接口”[“State”，GHJV95，第203页]或者“声明（declare）一个接口”[“Strategy”，GHJV95，第209页[4]]时，它就是在描述一个抽象类。

Auer对怎样开发可复用的、可扩展的类体系进行了讨论[Auer95]，他也建议用一个基类来定义接口、让子类来实现它。

相关模式

绝大多数设计级的模式都使用了抽象类。《设计模式》一书所介绍的23个模式中，有20个建议使用抽象类（Singleton、Facade和Memento除外）。[GHJV95]

一个抽象类经常会用模板方法[GHJV95，第214页]来实现。

Auer给出了一个简单的类体系开发过程，其中的接口就是用抽象类定义的。[Auer95]

参考书目

[Auer95] Ken Auer. “Reusability Through Self-Encapsulation.” *Pattern Languages of Program Desig*n. Edited by James Coplien and Douglas Schmidt. Addison-Wesley, 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Addison-Wesley, 1995. 中文版：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社2000年9月。[5]

[LW93] Barbara Liskov and Jeannette Wing. “A New Definition of the Subtype Relation.” *ECOOP ’93, Lecture Notes on Computer Science 707*. Berlin, Heidelberg: Springer-Verlag, 1993, pp. 118-141.

[Meyer91] Bertrand Meyer. “Design by Contract.” *Advances in Object-Oriented Software Engineerin*g. Edited by Dino Mandrioli and Bertrand Meyer. Prentice-Hall, 1991, pp. 1-50.

[WWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Softwar*e. Prentice Hall, 1990.

[Woolf97] Bobby Woolf. “Polymorphic Hierarchy.” *The Smalltalk Repor*t. January, 1997. 6(4).

致谢

我要感谢Dana Anthony和Steve Berzeuk，他们帮助我对这篇文章做了改进。

原文链接：http://www.shecn.com/zippdf/woolf97.zip

[4] 译者注：在李英军先生的译本中此处为“定义所有支持的算法的公共接口”，并没有出现“声明”字样。
[5] 译者注：本文中提到这本书时，页码都以中文版为准。

Document-View-Presentation模式 讨论

Ku-Yaw Chang等 著，透明 译

抽象

在设计一个交互式系统时，我们将面临这样的挑战：我们必须一致性地保存数据模型，同时我们必须提供独立于用户界面（user interface）而操作数据的功能核心。我们通常会把翻译函数（rendering function）独立出来，用翻译函数将数据翻译成合适的形式，用经过翻译的数据作为用户界面的输出部分。换句话说，数据的翻译过程和翻译结果的输出方式是紧耦合的。这个局限性造成的结果是，每当数据发生改变，我们都必须为每个用户界面重新翻译一次数据（甚至在多个用户界面共享同一个翻译结果的情况下）。

本文描述了用于交互式软件系统的“文档-视图-表示（Document-View-Presentation，DVP）”模式。本模式建立在Document-View模式的基础上，并且很有效的将翻译函数与翻译结果的输出相解耦。解耦的效果是，我们可以只对数据做一次翻译，然后用不同的方法重复输出翻译的结果。对于那些有翻译算法代价昂贵的交互式系统——例如图形图象处理系统——来说，本模式尤其适用。而且，DVP模式还为交互式系统引入了三层结构（3-tier）的概念，使交互式系统拥有“瘦”客户（即用户界面）。这样的瘦用户界面可以分布在另外的进程或者另外的机器上，而这不会对系统产生重大的影响。这样的分布式性能可以让一个交互式系统进化成一个基于WEB的应用程序（Web-based application），而不需要开发者做太多的努力。

Document-View-Presentation模式

Document-View-Presentation（DVP）结构模式将交互式应用程序分割成三个组件（component）。文档（document）组件包含了核心功能和数据；视图（view）组件处理服务请求并翻译文档中的数据；表示（presentation）组件接收事件或服务请求，并输出视图翻译的结果。这样的“变化-传播（change-propagation）”机制可以帮助：（1）文档与它的视图，（2）视图与它的表示--保持状态同步。

例子

请想象一个用于三维图象生成和诊断分析的医学应用系统，例如[1]中提到的Discover系统。这样的系统通常提供好几个窗口来表示三维图象，让用户可以方便的对三维图象进行交互式操作。这些窗口通常包含各自不同的翻译算法，比如体积翻译（volume redering）或表面翻译（surface rendering）。通常，这些翻译算法的运算量是如此之大，以至于有时候我们要通过分布式运算来缩短运算时间。对于同一个翻译算法的结果，系统支持三种不同的输出方法，包括在屏幕（窗口）上显示、保存到磁盘上、传递给另一个组件做进一步处理。当三维图象数据发生变化时，所有的输出方法必须立刻反映出这一变化，它们必须马上运行一次（而不是三次）翻译算法（如图1所示）。在特定情况下，对于那些翻译算法代价非常昂贵的交互式系统来说，这些算法被执行一次和三次对于系统的性能表现是有很大不同的。

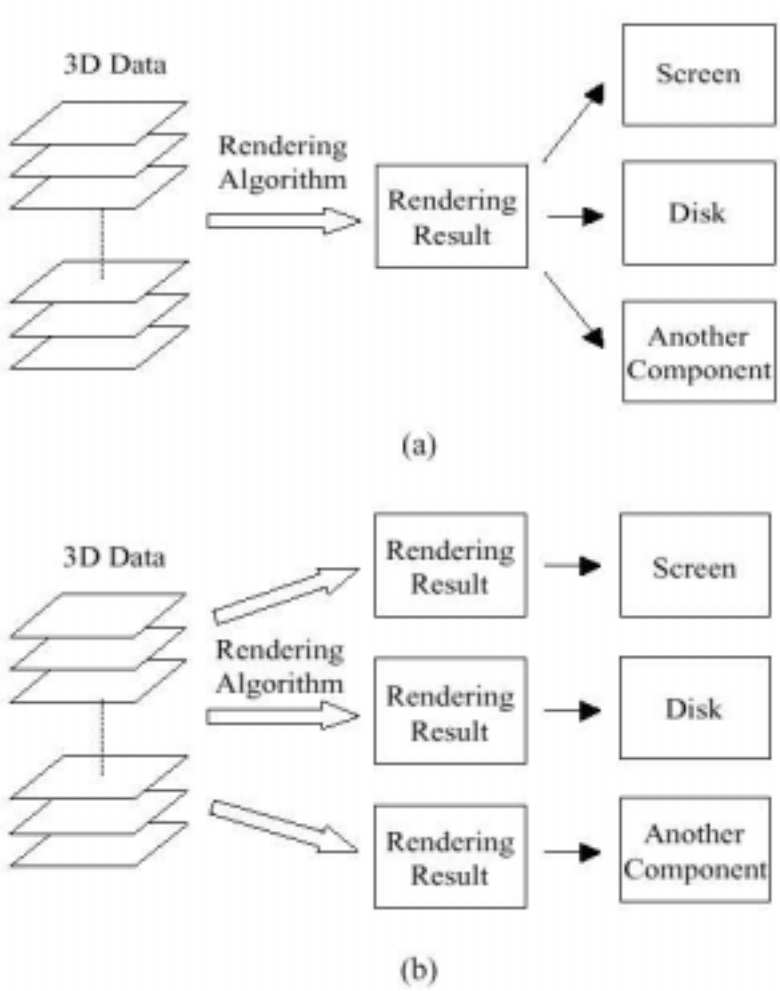


图1：为反映数据变化而对三维数据进行（a）一次（b）三次翻译

在上述案例中，我们需要分别处理不同的输出方法，但又不能对系统的性能造成明显的影响。在运行时，系统对不同的输出方法（甚至于输入方法）应该是可以“即插即用（plug-and-play）”的。

环境

数据翻译算法代价昂贵的交互式应用系统，比如图形图象处理系统。

问题

在设计交互式系统结构时，请注意一个要点：保持数据及功能核心与包含输入输出的用户界面相独立。通常我们的设计会让数据模型有许多不同的用户界面与之相关联。当数据发生变化时，所有与之相关联的用户界面必须立刻反映这一变化。而且，负责将数据翻译成合适的视觉形式的翻译函数通常被看做用户界面的一部分。这样的设计思路的局限性是，每当一个用户界面需要被更新时，我们都不可避免的需要执行翻译函数。

假想我们的三个不同的用户界面有同一个翻译函数，并且它们通过这个翻译函数被关联到同样的数据。所有的用户界面都需要在数据发生变化时被更新。前面的设计的局限性就是，当数据发生变化时，翻译函数必须被执行三次——为每个用户界面执行一次。但是，如果这些用户界面拥有同一个翻译函数，它们就可以共享同一个翻译结果。也就是说，翻译函数只需要运行一次，翻译的结果可以被不同的用户界面复用（reuse）。简单的说，如果把翻译函数看成是用户界面的一部分，在反映数据变化时为了更新所有用户界面，就可能需要成倍的运算时间。尤其当翻译算法的运算代价非常昂贵时，运算时间的浪费就更加严重。

在确定解决方案时，需要注意以下几点：

- 结果的输出和应用程序的行为必须立刻反映数据的变化。
- 翻译结果应该可以被不同的输出方法复用。
- 对于同一个翻译结果，添加、分离不同输出方法都应该是容易的，甚至可以在运行时进行。
- 支持不同的输出方法不应该对应用程序的核心（包括表现数据的翻译函数）有影响。

解决方案

本模式建立在Document-View模式[2，3]的基础上，将交互式系统分割成三个组件：document、view和presentation。

document组件包含了数据和处理数据的核心功能。本组件独立于任何特定输入输出方法。

view组件包含了应用程序的翻译算法。它们被隐藏在presentation组件后面，不与最终用户发生直接交互。view组件接收用户请求，然后调用document提供的核心功能或自己的翻译算法实现相关的服务。view组件还从document组件获取数据，然后用不同的算法来翻译数据。一个document组件可以有一个或多个view组件与之相连。

presentation组件是view组件输入输出的代理。它们从其他组件接收用户事件或消息，再把这些事件

和消息处理成服务请求。presentation组件不直接实现这些服务，而是把这些服务请求转发给相关联的view组件。另外，presentation组件还从相关联的view组件获取翻译结果，再根据自己的特征把翻译结果输出到合适的目的地，例如显示在屏幕上或保存到磁盘上。一个presentation组件可以只负责输入、只负责输出、或者同时负责输入和输出。presentation组件对于用户可以是可见的或者不可见的。用户可以通过那些可见的presentation组件与系统直接交互。

每个presentation组件只能与一个view组件相关联。但是，一个view组件可以拥有几个presentation组件，这取决于应用程序的需要。实际上，DVP模式中的view组件和Command Processor模式[2]中的controller组件扮演着相似的角色。它允许系统支持不同样式的输入输出（即不同的presentation组件）。

将document组件与view组件分离，于是我们的系统就可以对同样的数据进行不同方法的翻译。当presentation组件接收到用户触发的事件时，它首先把事件翻译成服务请求，然后把请求转发给与自己相连的view。view响应服务请求，根据请求改变document中的数据。一旦document中的数据被改变，document就通知所有依赖于它的view组件，要求view组件翻译数据。当翻译结果发生改变的时候，view就通知所有与它相连的presentation组件。每个presentation组件都将从view组件接收到当前的翻译结果，并将这些结果输出到不同的设备（例如屏幕、磁盘或网络）上。这样的变化-传播机制可以用Publisher-Subscriber模式[2]或Observer模式[4]来实现。

结构

document组件维护数据模型，并且包含处理数据的功能核心。它的导出函数用于执行应用程序指定的处理，所以依赖于它的view组件可以调用这些函数提供服务、访问数据。当然，document提供了变化-传播机制。document组件维护一个注册表，记录依赖于它的view组件信息。数据的改变将触发变化-传播机制，于是document组件会向所有记录的view组件发出通告。

view组件对document组件中的数据进行翻译，并保存翻译结果。每个view组件都应定义一个update函数，由变化-传播机制调用这个函数。当update函数被调用的时候，每个view组件会从document组件接收到数据，然后对这些数据进行翻译。实际上，每个view组件也提供另一个变化-传播机制。view组件也维护一个注册表，记录依赖于它的presentation组件。翻译结果的改变将触发变化-传播机制，于是view组件会向所有记录的presentation组件发出通告。view组件还接收来自presentation的服务请求，并通过调用document提供的核心功能或自身的翻译算法来实现presentation所请求的服务。

presentation组件接收输入消息并将输入消息翻译成不同的服务请求。每个presentation组件也定义一个update函数，这个函数也由变化-传播机制来调用。当update函数被调用时，每个presentation组件都从与之相连的view组件接收当前的翻译结果，并依照自己的特征输出翻译结果（比如将翻译结果显示在屏幕上）。

document、view和presentation三个组件的类-责任-协作（Class-Responsibility-Collaborator，CRC）卡片如图2所示。

图2：document、view和presentation三个组件的CRC卡片

图3展示了DVP模式的组件之间的主要关联。view组件和presentation组件分别是document组件和view组件的订户（subscriber）。换句话说，一个document可以有不止一个view与之相连，同时一个view也可以有不止一个presentation组件与之相连。在一个使用C++的实现中，每个组件都被定义为一个单独的类。view类和presentation类拥有一个共同的父类——Subscriber类，它在接口中定义了update函数。相似的，document类和view类拥有另一个共同的父类——Publisher类，它在接口中定义了attach、detach和notify等函数。

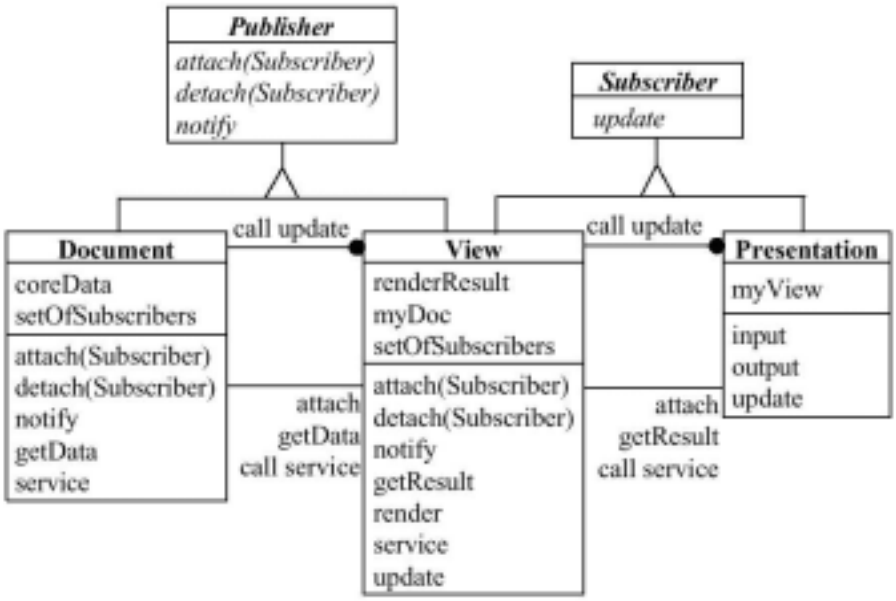


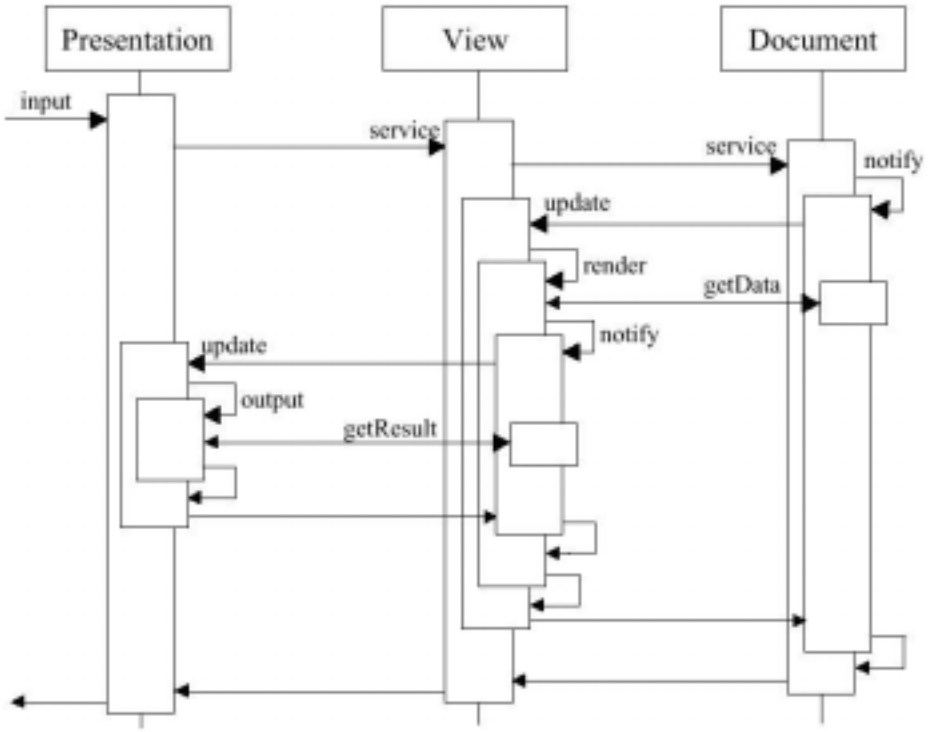
图3：用对象建模技术（Object Modeling Technique，OMT）构造的DVP模式对象模型

动态分析

我们将图解下列两种场景下DVP结构的行为。

场景I展示出输入消息怎样改变document的数据、这一改变又怎样触发变化-传播机制。为简单起见，在这个图表中只使用一个view-presentation组件对。

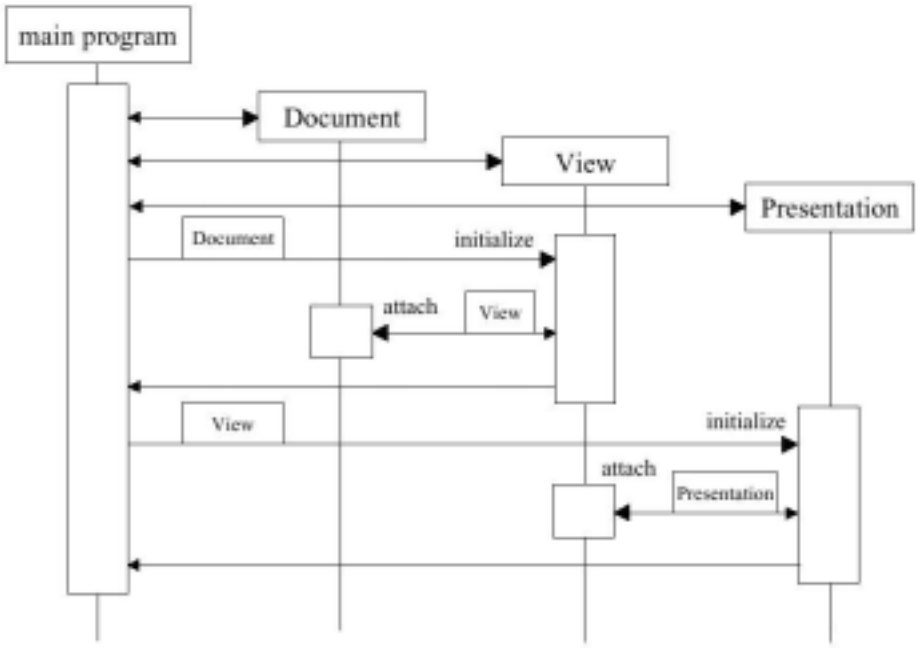
- presentation接收到一个输入消息，并将输入消息翻译成view能理解的服务请求。
- view调用document提供的服务功能。
- document实现被请求的服务，通常这会导致document中的一些数据发生改变。
- document调用所有与之相连的view的update函数，对它们发出通告。
- 每个view都接收到来自document的数据，并调用render函数对数据进行翻译。
- view调用所有与之相连的presentation的update函数，对它们发出通告。
- 每个presentation组件都接收到来自view的翻译结果，并输出翻译结果。



场景I

场景II展示了DVP三元组如何被初始化。这个初始化过程通常由主程序中的代码按照下面步骤进行：

- 用初始化数据结构创建document实例。
- 创建一个view对象，将一个document的引用作为初始化参数。
- view对象通过调用document对象的attach函数而预定变化-传播机制。
- 创建一个presentation对象，将一个view的引用作为初始化参数。
- presentation对象通过调用view对象的attach函数而预定变化-传播机制。



场景II

实现

在实现DVP模式时，请遵循以下几个基本的步骤：

1. *定义数据模型和核心功能*。分析应用领域并将之映射到一个合适的数据模型上。将核心功能与输入输出行为分离开。设计document组件来包装数据和核心功能。提供访问数据所需的函数。

在我们的Discover例子中，一个BS对象（一个二维Binary Slices的集合）被用于表现三维对象。每个document都包含一个BS对象，并提供访问和操作数据（即BS对象）的方法。因为document扮演着发行者（publisher）的角色，它继承自Publisher基类。我们将在第二步中讲到这个基类。

```
class Document: public Publisher
{
    private:
        // BSOBJ is the data structure for BS objects
        BSOBJ coreData;
    public:
        Document();
        // methods for views to access the core data
        BSOBJ * getData();
```

```
        // methods for views to manipulate the data
        void doThreshold(int low, int high);
        void doCut(CONTOUR * con2r, int alpha, int beta);
    }
```

2. *实现第一个变化-传播机制*。在Publisher-Subscriber模式的基础上，我们为document指定了发行者的角色。进一步扩展document，为它加上一个注册表，在其中保存观察对象——view——的引用。提供一组函数，允许view预定和取消预定数据变化通告。document的notify函数调用所有观察对象的update函数。

在我们的C++例子中定义了两个抽象类——Publisher和Subscriber。Publisher类保存当前的订户，并提供attach()和detach()方法，允许观察对象预定和取消预定。Subscriber类提供update接口函数。任何会修改document的状态、类似doThreshold()的方法都会调用notify()方法来通知当前的订户。notify()将遍历注册表中所有的Subscriber对象，并调用它们的update()方法。

```
class Publisher
{
    private:
        Set<Subscriber *> registry;
    public:
        virtual void attach(Subscriber * s) { registry.add(s); }
        virtual void detach(Subscriber * s) { registry.remove(s); }
    protected:
        virtual void notify() {
            // call update for all subscribers
            Iterator<Subscriber *> iter(registry);
            While (iter.next()) {
                Iter.curr()->update();
            }
        }
};
class Subscriber
{
    public:
        // default is to do nothing
        virtual void update() { }
};
```

3．*定义翻译结果的内容*。分析不同的输出行为需要的信息，定义翻译结果的内容。

对于一张三维图象，我们出于不同的目的会需要它的不同方面的信息，比如颜色、深度（z坐标值）和透明度等等。在这里，我们用RENDER_DATA这样一个数据结构来收集所有需要的信息。

4．*设计并实现view类*。为每个view类设计翻译算法。指定并实现一个翻译函数用来翻译document中的数据。update函数调用翻译函数进行实际翻译过程。为每个请求提供一个服务函数。通过调用document的核心函数或自己的算法满足服务请求。view对象在初始化过程中应向document对象提出预定。

在Discover的例子里面，所有具体view类共享一个抽象基类view，其中定义了不同的view类的公共行为。view类同时继承Publisher和Subscriber两个基类。view类包含两个成员变量，其中一个表示翻译结果，另一个表示view与document的关联——view的构造函数向document发出预定而建立这一关联；析构函数取消预定，从注册表中移除这一关联。view类还提供从presentation组件接收服务请求的方法。每当update()方法被调用时，view对象就产生一个新的翻译对象，并向所有注册的观察对象通知这一变化。

```
class View: public Publisher, public Subscriber
{
    protected:
        RENDER_DATA renderResult;
        Document * myDoc;
    public:
        View (Document * doc) : myDoc(doc) { myDoc->attach(this); }
        virtual ~View() { myDoc->detach(this); }
        // methods for presentation components to access
        // the render data
        RENDER_DATA * getResult() { return (&renderResult); }
        // methods for presentation components to
        // manipulate the data
        void doThreshold(int low, int high) {
            myDoc->doThreshold(low, high);
        }
        void doCut(CONTOUR * con2r, int alpha, int beta) {
            myDoc->doCut(con2r, alpha, beta);
        }
        // define the default behavior to reflect changes of
        // Document's data
```

```
        virtual void update() {
            this->render();
            this->notify();
        }
        // abstract method to be redefined:
        // default is to do nothing
        virtual void render() {}
};
```

SurfaceRenderingView类的定义给出了一个本系统中具体view类的示例。它重载render()方法，通过基于表面的方法（surface-based approach）对BS对象进行翻译。另外，它还提供形如doRotate()和doSetColor()这样的函数来改变翻译结果，但不真正改动document中的数据。

```
class SurfaceRenderingView : public View
{
    public:
        SurfaceRenderingView(Document * doc) : View(doc) {}
        // method for surface rendering algorithm
        virtual void render() {
            // get core data from Document
            BSOBJ * bs = myDoc->getData();
            // do the surface rendering algorithm
            SurfaceRendering(&renderData, bs);
        }
        // methods to change the rendering attributes
        void doRotate(int alpha, int beta);
        void doSetColor(int r, int g, int b);
};
```

5. *实现第二个变化-传播机制*。为view指定publisher的角色。扩展view类，为它添加一个注册表，在其中保存观察对象（presentation组件）的引用。提供一组函数让presentation组件可以预定和取消预定翻译结果的变化通告。view的notify函数调用所有观察对象的update函数。请注意，对于document来说，view是订户；对于presentation组件来说，view是发行者。

6. *设计和实现presentation组件*。为每个presentation组件指定具体的输入和输出行为。presentation组件的初始化需要向view组件发出预定请求。

所有的具体presentation类共享一个抽象基类presentation。presentation类继承Subscriber基

类。presentation类包含一个成员变量，用以表示它与view的关联——presentation对象的构造函数向view对象发出预定，建立这一关联；析构函数取消预定，将关联从注册表中移除。

WndPresetation类定义了是一个presentation组件的示例，它可以接收用户输入并在屏幕上显示翻译结果。它重载了update()方法，以获取当前的翻译结果并在屏幕上显示。另外，它还提供回调函数来接收用户输入。

```
class Presentation: public Subscriber
{
    protected:
        View * myView;
    public:
        Presentation(View * view) : myView(view) {
            myView->attach(this);
        }
        virtual ~Presentation() { myView->detach(this); }
        // define the default behavior to reflect changes of
        // View's rendering result
        virtual void update() { this->output ();}
        // abstract method to be redefined
        virtual void output() {}
};
class WndPresentation : public Presentation
{
    public:
        WndPresentation (View * view) : Presentation(view) {}
        // method to output the rendering result
        virtual void output () {
            RENDER_DATA * result;
            result = myView->getResult();
            // display the image on the screen: …..
        }
        // call-back functions to receive user inputs
        void onThreshold() {
            int low, high; // required parameters
```

```
            // open a dialog box to obtain parameters: ...…
            myView->doThreshold(low, high);
        }
        void onRotate() {
            int alpha, beta; // required parameters
            // open a dialog box to obtain parameters: …..
            myView->doRotate(alpha, beta);
        }
};
```

已知应用

Discover[1，7]是一个用于科学显像的分布式交互系统，该系统从1993年开始已经在台湾Cheng-Kung大学医院投入试用。该系统就是建立在DVP模式结构基础之上的。除了普通的成像分析、图象生成的功能之外，Discover还提供如下的功能：

1. 图象综合（image integration）

不同的翻译算法或者不同的医学成像设备会生成不同但是相关的三维图形对象。有时候，医师为了进行诊断分析，需要找出这些不同三维图形之间的关联。Discover可以让医师把几个现有的三维对象（即：源对象source object）综合成一个新的三维对象（即：综合对象integrated object），以达到医师需要的观察要求。

如图4所示：在图象综合过程开始之前，每个view对象都用一个可见的default presentation组件（这个组件可以接收用户事件、在屏幕上显示翻译结果）进行初始化。当一个三维对象被选做源对象时，系统在运行时创建一个output presentation对象，并将之附加给源对象所属的view对象。output presentation组件对用户是不可见的，它完全不接收输入消息。同时，output presentation也是一个发行者，它将view组件的翻译对象输出给它的订户。另外，系统还为综合对象创建一个新的document、view和default presentation三元组。新的document对象注册所有源对象的output presentation，这样document对象就可以分别获取源对象的翻译结果。

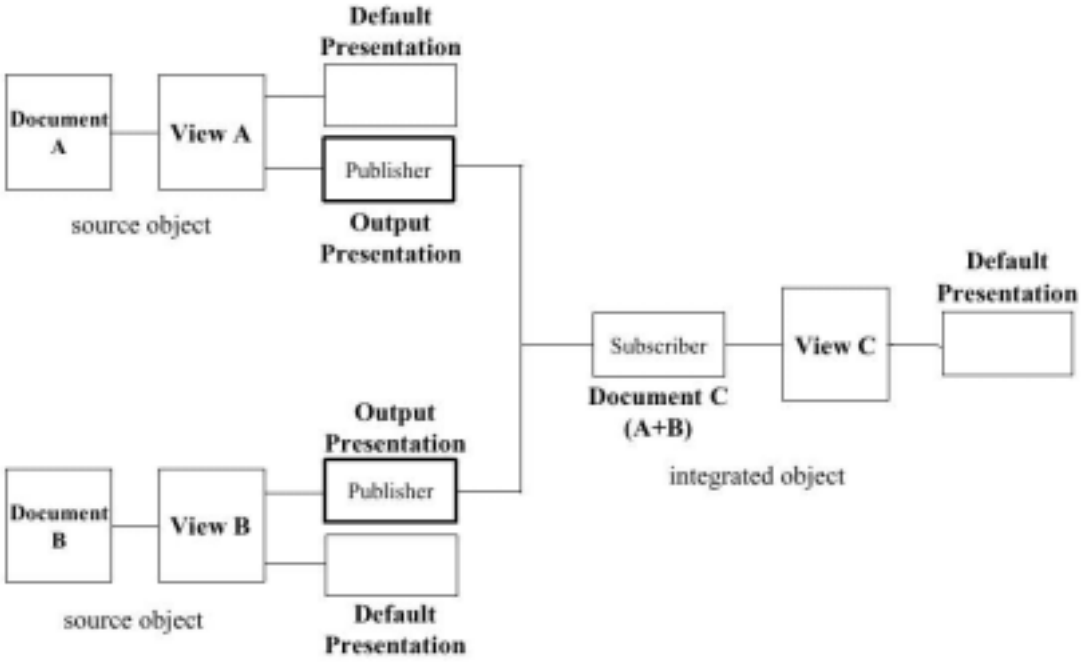


图4：添加每个源对象的output presentation的引用，就可以轻松的获取一个综合对象

因此，当医师通过任何一个源对象的default presentation修改源对象时，这个源对象所属view的翻译结果会得到更新，这个过程只需要执行一次翻译算法。default presentation组件和output presentation组件都必须立刻反映这一变化：前者为医师显示新的翻译结果，后者向综合文档（integrated document，综合对象的document）发出变化通告。综合文档从output presentation取回新的翻译结果，更新自己的数据。然后，综合对象的view对综合数据进行翻译，并通知自己的default presentation对象向医师显示翻译结果。

2. 基于WEB的应用（web-based application）

通常Discover提供的图象生成和诊断分析功能对实时性要求是非常强烈的。尽管我们可以为每个医师的桌面计算机都安装Discover系统，但也不是每台桌面计算机都有足够的运算能力——它必须进行如此沉重的运算，还必须有可以接受的响应时间。所以，Discover还允许医师使用web浏览器来控制它。因此，我们可以只在一台计算能力很强的服务器上安装Discover系统。医师们可以用他们的运算能力不同的桌面计算机通过网络来控制Discover系统。如果不考虑网络流量和服务器负荷的影响，所有的桌面计算机都将可以获得几乎相同的响应时间。

如图5所示：当web服务器接收到一个来自浏览器的控制Discover的请求时，它运行一个通用网关接口（Common Gateway Interface，CGI）程序来实现服务。CGI程序首先通过自动化机制（automation mechanism，[8]）为view组件创建一个input presentation和一个output presentation，然后将服务请

求转发给input presentation。当view保存的翻译对象被改变时，output presentation将翻译结果交付给CGI，CGI再将翻译结果转发给web服务器。最后，翻译结果被发送给web浏览器，医师就可以在浏览器上看到所需的图象了。

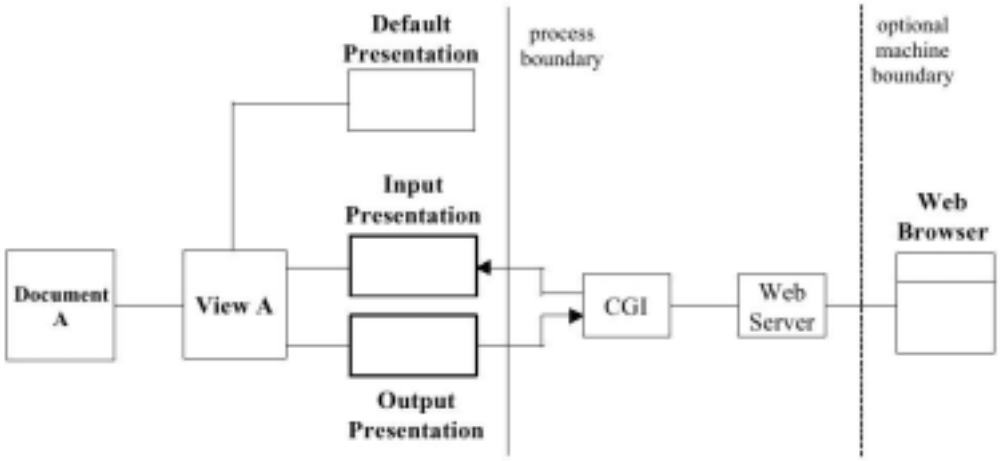


图5：通过input presentation和output presentation的合作，可以用web浏览器控制Discover系统

效果

Document-View-Presentation结构模式有几个优点：

1．*复用翻译结果*。当多个用户界面（即presentation组件）共享同一个翻译结果时，只需要执行一次翻译函数。于是我们可以节省大量运算时间，从而得到更快的响应，特别是当翻译函数的运算量很大的时候。

2．*要点分解*。对翻译过程和输入/输出行为进行分解，这使得软件开发者将注意力集中在翻译算法上，而不必操心任何输入/输出操作。当然，他们还可以将注意力集中于提供不同的输入/输出方法，而不必陷入翻译算法的开发中。

3.*“可插入的”表示组件*。可以实现多个presentation组件，并用一个view组件使用它们。这些presentation组件可以在运行时添加甚至替换，而不会对view组件造成任何影响。

4．*瘦用户接口*。将翻译过程从输入/输出过程中移除，这使用户接口变得简练。这样的瘦用户接口可以被容易的分布到其他进程或其他机器上。比如说，如果我们通过HTTP协议来分布用户接口，一个交互式系统就可以变成一个基于web的应用。

5．*使用先进的Windows应用程序框架很容易实现DVP模式*。好几个开发Windows应用程序的框架——例如Visual C++的MFC和Borland C++的OWL——都采用Document-View模式作为它们的默认系统结构[3]。从本质上来说，DVP模式是Document-View模式的扩展。所以，我们只需使用这些框架提供的document-view结构相关类和通信机制就可以很容易的实现DVP模式。我们可以禁用框架提供的view类的输入/输出函数，并

提供附加的presentation组件。当然，我们还需要实现view和presentation之间的通信机制。

本模式的缺点如下：

1. *复杂性增加*。用户界面被分割到view和presentation两个组件中，这会导致系统结构变得复杂。如果对document中的数据的翻译算法比较简单，这一分割只会增加复杂性，却得不到明显的性能改善。

相关模式

- Model-View-Controller（MVC）模式[2]将应用程序分割成三个组件：model包含核心算法和数据；view向用户显示信息；controller处理用户输入。Document-View（DV）模式——MVC的一个变体——将MVC中的view和controller结合成一个组件。在这两个模式中，向用户显示信息的过程实际上包括两个过程：翻译（或解释）数据、在屏幕上显示结果。

  表1总结了DVP模式和上述两个模式中的组件以及它们的主要责任。这三个模式的共同点是：它们都把核心功能和数据放在一个组件（即model或document）中。DVP模式与MVC/DV模式最大的不同点是：DVP模式将“原始”输入/输出与其他组件分离。这包括（1）对document中的数据显示分成两部分：数据翻译和输出，（2）将用户输入和服务请求的内部联系解耦。

MVC	DV	DVP
Model •core functionality •data	**Document** •core functionality •data	**Document** •core functionality •data
View •display (render + output)	**View** •display (render + output) •user input	**View** •rendering functionality •accept service requests
Controller •user input		**Presentation** •output •user input

表1：交互式系统的三个模式的特点总结

另外，MVC/DV模式对交互式系统结构的分割可以看成是在应用程序内部形成了两层的client/server体系结构（如图6所示）。数据和功能核心运行在server端，翻译函数和图形用户界面则运行在client端。这样的两层系统如果有大运算量的翻译算法，则client端会变得非常“肥胖（fat）”。与之相对的，DVP模式可以看成一个三层体系结构（如图7所示）。在三层系统中，中间层由业务逻辑单元组成。这些业务逻辑单元通常不依赖于终端应用程序，因此它们可以被不同的应用程序复用。在Discover系统中，数据翻译过程（计算机图形应用的业务逻辑单元）从client端被移到了server端的中间层，从而保证了client端的精简。同时，这样的变化还使得不同的client（presentation组件）可以复用中间层（view组件）的翻译结果。

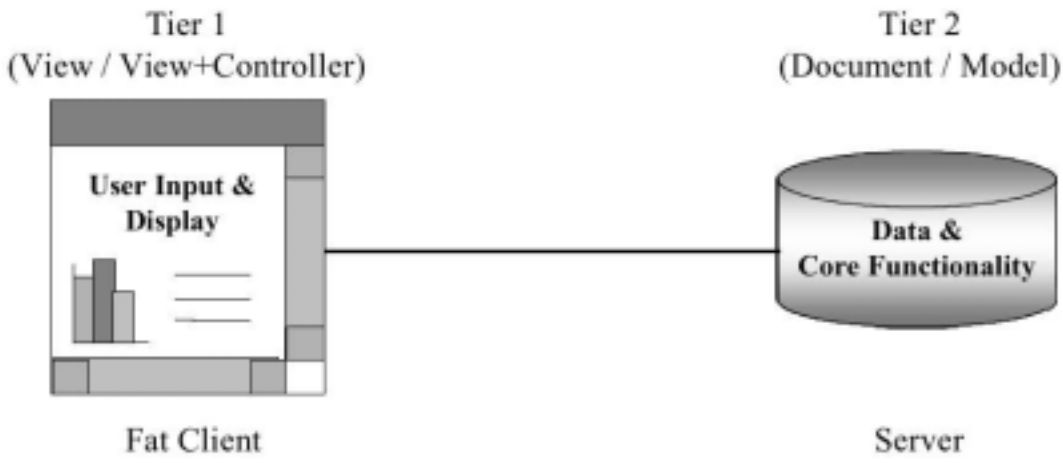


图6：MVC模式和DV模式都是把交互式应用程序分解成两层

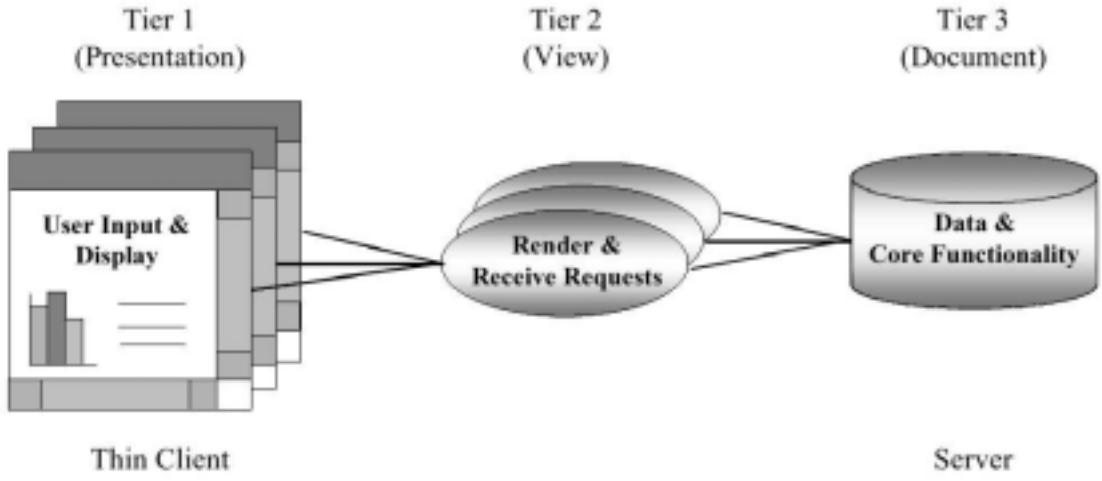


图7：DVP模式将交互式应用程序分解成三层

参考书目

[1]. P. W. Liu et al., "Distributed Computing: New Power for Scientific Visualization," *IEEE Computer Graphics and Applications*, Vol. 16, No. 3, May 1996, pp.42-51.

[2]. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *A System of Patterns* -

Pattern-Oriented Software Architecture, John Wiley & Sons, New York, 1996.

[3]. D. Kruglinski: *Inside Visual C++*, Microsoft Press, 1996. 中文版：《Visual C++技术内幕（第四版）》，潘爱民等译，清华大学出版社1999年1月。

[4]. E. Gamma, E. Helm, R. Johnson and J. Vlissides, *Design Patterns -Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. 中文版：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社2000年9月。

[5]. K. Beck and W. Cunningham, "A Laboratory For Teaching Object-Oriented Thinking," *Proceedings of OOPSLA'89*, N. Meyrowitz(Ed), Special Issue of SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 1- 6.

[6]. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

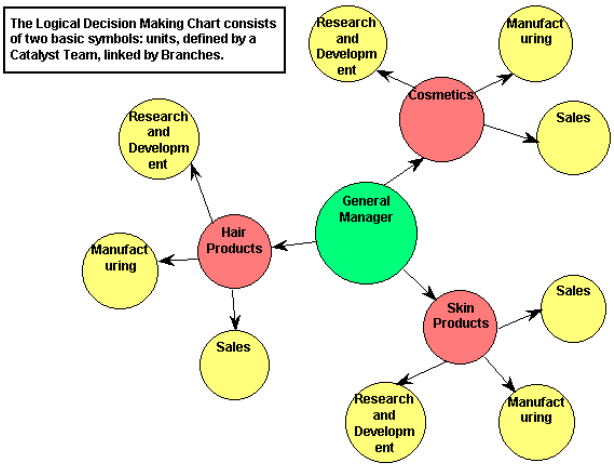
[7]. K. Y. Chang and L. S. Chen, "Using Design Patterns to Develop a Hyper-controllable Medical Image Application," *Proceedings of PLoP'98*, Monticello, Illinois, Aug 11-14, 1998.

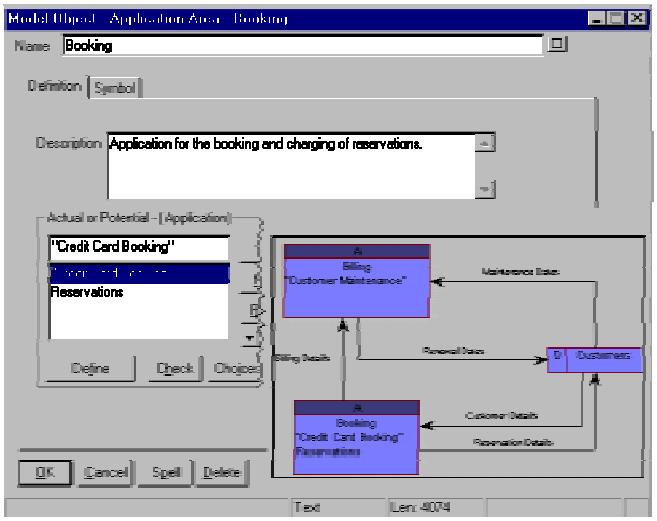
[8]. R. M. Adler, "Emerging Standards for Computing Software," *Computer*, March 1995, pp. 68-77.

致谢

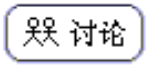
在此特别向Rosana T. Vaccare Braga表示感谢，她为改善这个模式给了我们很多有价值的建议。

原文链接：http://www.umlforum.com/zippdf/chang.zip





Role Object（角色对象）模式



Dirk Bäumer等 著，透明 译

意图

通过向组件对象加入对用户透明的角色对象（role object）——每个角色对象扮演了组件对象需要在一个用户的环境（context）中扮演的角色——使组件对象能适应不同用户的需要。每个“用户的环境”可能是它自己的一个应用程序，通过使用这个模式，应用程序与其它应用程序的耦合度将得到降低。

动机

一个面向对象系统总是以一系列特征性的基本抽象（key abstract）为基础的。每个基本抽象都由一个相应的类根据其抽象状态和行为模拟实现。在设计小规模的应用程序时，这个机制通常运行得很好。但是，当我们要把系统的规模扩大成一套完整的应用程序时，我们就必须面对各种不同的客户，这些客户希望根据我们的基本抽象得到各自不同的、根据环境指定的视图（view）。

假设我们正在为银行的投资部门开发一个支持软件，我们的基本抽象之一是表示“投资者”的概念。因此，我们的设计模型中应该有一个Customer类，这个类的接口提供了管理消费者的姓名、住址、储蓄、存款余额等属性的操作。

现在，假设这个银行的贷款部门也需要软件支持。看起来，我们的设计不能满足处理一个贷款者客户的需要。很明显，我们必须为Customer类增加更多的状态和操作，使之能管理客户的贷款额、信用级别、债券等信息。

把几个根据不同环境指定的视图包含在同一个类中，这样的做法最可能的后果是把基本抽象变成一个臃肿的接口。对这样一个臃肿的接口的理解和维护都是非常困难的。当一个未知的变化发生时，你无法温和的处理它，而通常只能任它引发对一系列程序模块的重新编译（recompilation）。而对用户指定的接口部分的变化则很可能影响到其他的子系统或应用程序。

这里有一个简单的解决方案：扩展Customer类，加入Borrower和Investor两个新的子类，让它们分别捕获贷款者和投资者各自特殊的事件。从对象身份的角度来考虑，子类化就意味着分属两个不同子类的对象是不同的。因此，一个既是投资者又是贷款者的客户就要扮演有截然不同的身份的两个对象。对象的“身份”只能以附加的机制来模拟（例如C++的运行时类型识别RTTI），如果两个对象有同样身份，它们必定

有恒定不变且互不相同的继承属性。无论如何，假设要构造系统中所有客户的列表，我们将不可避免的陷入多态搜索（polymorphic search）的麻烦。除非我们很小心的避免复制（duplicate）对象，否则同一个客户对象将作为不同的角色重复出现。

角色对象模式模拟了根据环境指定的对象视图，各个单独的角色对象被核心对象（core object）动态添加、移除。最终得到的对象聚合体作为一个逻辑对象进行它自己的行为。

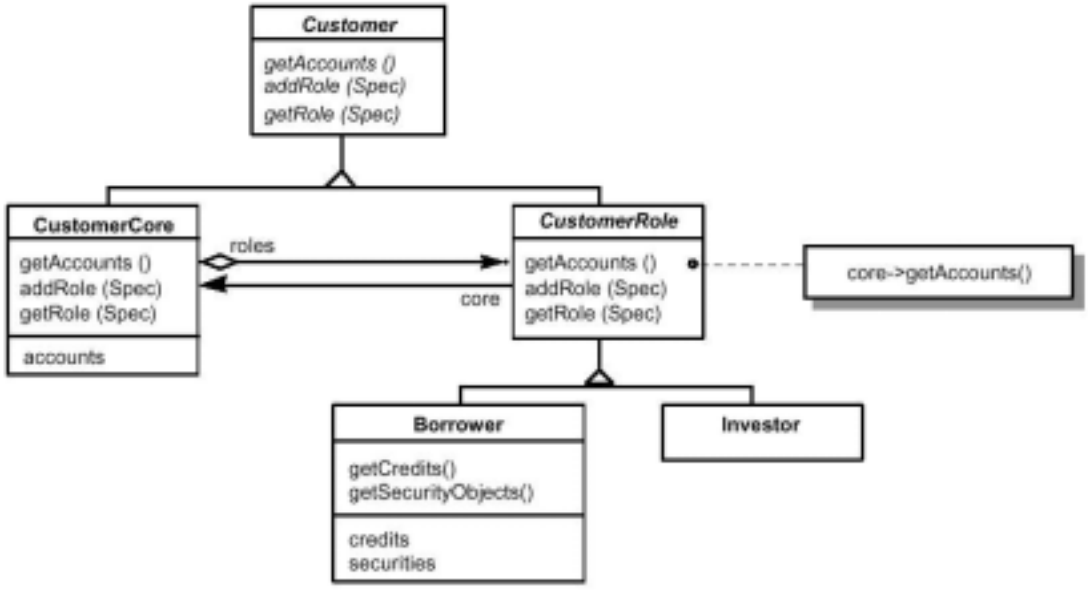


图1：银行环境中Customer类继承关系

象Customer这样的基本抽象被定义成一个抽象超类。它只提供一个接口，不做任何实现。Customer类指定了处理客户住址、存款余额这样的操作，并且定义了一个管理角色的最小化协议。子类CustomerCore实现了Customer的接口。

CustomerRole则提供了用户指定角色的超类，并且它也支持Customer的接口。CustomerRole是一个抽象类，它不能被实例化。CustomerRole的具体子类——例如Borrower或者Investor——定义并实现了用户指定角色对象的接口，只有这些子类才能在运行时被实例化。Borrower类定义了贷款部门需要的客户对象视图，它定义了一些附加操作以管理客户的信用和债券等信息。同样，Investor类也添加了投资部门指定的客户视图需要的操作。

象“贷款应用程序”这样的一个用户可能通过Customer的接口使用CustomerCore类，也可能使用某个具体CustomerRole子类的一个对象。假想这样的情况：贷款应用程序通过某个Customer实例的Customer接口了解到它，并且应用程序可能想检查这个Customer是否扮演着贷款者的角色。于是应用程序调用该对象的hasRole()方法，这个方法会告诉调用者该对象的角色。在本例中，我们假设我们的每个角色都有一个简单string类型的名字。如果这个Customer对象可以扮演名叫“Borrower”的角色，贷款应用程序会要求它传递一个自己的引用给相关的对象，然后应用程序就可以使用这个引用调用贷款者专用的操作了。

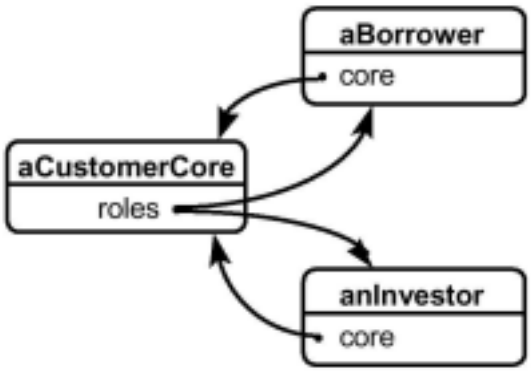


图2：角色对象模式的一个对象图

与此相似，“投资者”角色被它自己的应用程序处理。如果这两个应用程序不需要互相了解，那么角色对象的使用就可以帮助它们解耦。贷款应用程序不需要加载投资者角色，同时投资应用程序也不需要加载贷款者角色。

适用性

在以下情况下可以使用Role Object模式：

- 你希望在不同的环境中处理同一个基本抽象（每个环境可能是一个独立的应用程序），并且你不希望把所得到的环境相关的接口放到同一个类接口中去。
- 你希望能动态决定处理何种有用的对象，希望角色能根据要求在运行时被添加或移除，而不是在编译时被静态绑定。
- 你希望在不同的环境中保持逻辑对象的一致性。
- 你希望让角色/用户（role/client）互不依赖，这样对角色的修改不会影响到与之无关的用户。

如果有如下情况，不要使用这个模式：

- 你的潜在角色有很强的相互依赖。有其他的一些使用“角色”的方法，Fowler给出了使用这些方法的向导，并指出应该在什么时候使用什么模式[Fowler97]。

结构

下面是Role Object模式的结构图。

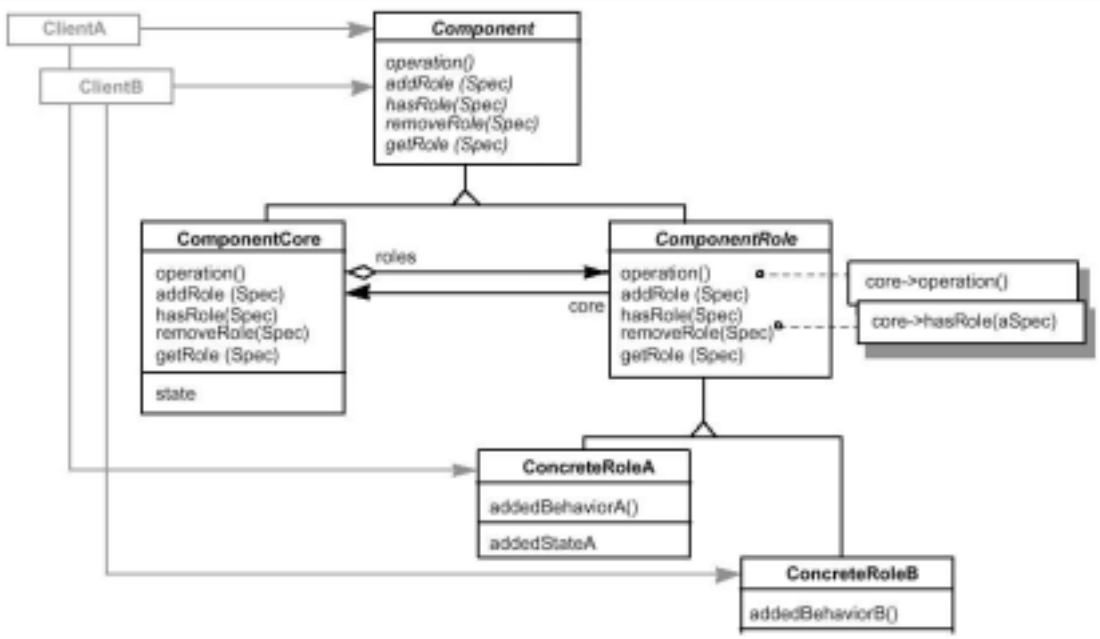


图3：Role Object模式的结构图

参与者

• *Component (Customer)* ——*组件*

- 通过定义接口模拟出一个特定的基础抽象。
- 指定一组协议用于对角色对象的添加、移除、测试和查询。用户（client）为具体角色（ConcreteRole）子类提供角色规范。在最简单的情况下，角色用string类做为标志。

• *ComponentCore (CustomerCore)* ——*组件核心*

- 实现Component接口，其中包括对角色管理协议的实现。
- 创建具体角色实例。
- 管理属于自己的角色对象。

• *ComponentRole (CustomerRole)* ——*组件角色*

- 保存自己所装饰（decorate）的组件核心对象的引用。
- 实现Component接口，把对此接口的请求转发给自己装饰的组件核心。

• *ConcreteRole (Investor, Borrower)* ——*具体角色*

- 模拟并实现根据环境指定的组件扩展接口。
- 可以用一个ComponetCore引用作参数被实例化。

协作

核心和角色对象进行如下协作：

- ComponentRole将请求转发给它的ComponentCore对象。
- ComponetCore创建并管理ConcreteRoles。

用户与角色、核心对象有如下交互：

- 用户可以给核心对象添加新的角色。它可以按照规范得到它希望的角色。
- 当用户需要一个核心对象扮演某种角色的时候，它向这个核心对象查询这个角色。如果该核心对象正在扮演这个角色，它就把角色的引用返回给用户。
- 如果该核心对象没有扮演这个角色，它将抛出一个错误。核心对象永远不会给自己创造角色对象。

效果

Role Object模式有如下的优点和效果：

- *简明地定义基础抽象*。组件接口很好的集中精力于所模拟的基础抽象的本质性状态和行为，而不会因为环境指定的角色接口而变得臃肿。
- *角色可以被很容易被扩展，而且彼此之间互不依赖*。扩展Component接口是很容易的，因为你不需要改变ComponentCore类。一个具体角色类就可以让你创建新的角色并实现它，同时保护了基础抽象。
- *角色对象可以被动态添加或是移除*。你可以在运行时添加或移除角色对象，只需简单的将它附加到核心对象上或是从核心对象上分离即可。这样，只有在特定情况下需要的那些对象才会真正被创建。
- *应用程序得到更好的解耦*。通过明确地把Component接口从它的角色中分离出来的手段，内含各种角色的应用程序的耦合度被降低了。一个应用程序（ClientA）使用Component接口和一些环境指定的ConcreteRole类，另一个应用程序（ClientB）也是如此，那么ClientA不需要知道ClientB使用的那些具体的角色。
- *使用多继承（multiple inheritance）避免了“组合爆炸（combinatorial explosion）”*。这个模式避免了类的组合爆炸，因为它使用多重继承，由此可以在一个类的基础上组合成多个角色。

Role Object模式有如下的短处和缺陷：

- *客户程序有可能变得更复杂*。使用一个类的角色来处理这个类的对象需要做稍微多一些的编码工作——与组件自己提供接口相比。客户程序必须检查对象是否扮演自己需要的角色。如果得到肯定的结果，客户程序还必须请求这个角色；如果对象没有扮演这个角色，客户程序有责任在自己的环境中扩展核心对象使之能扮演这个角色。
- *维护角色之间的约束（constraints）变得困难*。因为逻辑对象（logical object）由几个互相依赖的对象

组成，维护它们之间的约束并保持全局一致性可能会变得困难。在“实现”一节中我们对出现的几个问题进行了讨论。

- *角色之间的约束不能由类型系统（type system）强制执行*。在一个组合体中，你可能希望拒绝某些角色的添加。或者，某些角色的存在可能依赖于另外一些角色的存在。在Role Object模式中，你不能依靠类型系统帮你形成约束。你必须用运行时的检查来实现这些约束。
- *维护对象身份也会变得更复杂*。核心对象和它的角色实例共同构成了一个概念上的单元，这个单元拥有它自己的概念上的身份。实际存在的对象的身份可以被任何程序语言直接处理（通过比较对象引用来实现），检查一个概念上的身份却需要Component接口的附加操作。这可以通过比较核心对象的引用来实现。

实现

实现Role Object模式必须遵循如下两个关键点：用角色对象透明地扩展基础抽象，动态管理这些角色。为了达到透明扩展的目的，我们会使用Decorator模式[Gamma+95]；为了创建和管理角色，我们使用Produce Trader模式[Bäumer+97]。这样，Role Object模式结合了两个广为人知的模式，并且增加了新的语义。

- *提供接口一致性（conformance）*。因为我们希望无论在任何地方使用核心对象，角色对象都能被透明地使用，所以角色对象必须支持一个共同的接口。从以类为基础建模的观点来看，一个角色类是被看做它的核心类的特殊化版本的（即是说：一个Investor“是一个”（is-a）Customer）。从角色建模的观点——把角色看作建模第一位的类——来看，Borrower和Investor是Customer类的角色[RG98]。

  首先，我们给所有的对象同样的接口，这样我们可以为它们动态添加角色。在结构图中，这个接口是由Component类提供的，ComponentCore类实现了这个接口。
- *隐藏角色对象的创建过程*。角色的实例在运行时被用于装饰一个核心对象。一个核心的问题是：怎样创建一个具体角色的实例并将它附加到一个核心对象上。请注意，具体角色的创建不应该由客户程序来负责。更好的办法是：由ComponentCore来进行角色的创建，从而避免了一个角色对象单独存在（即：没有依赖于任何核心对象）的情况。这个方法还防止了客户程序知道怎样实例化一个角色对象。
- *将角色对象与核心对象解耦*。角色的创建和管理都应该以普遍（generic）的形式进行。另外，如果允许用新的未知角色扩展ComponentCore，则很难保证不改变ComponentCore的实现。因此，角色的创建和管理都必须独立于任何具体的角色类；ComponentCore的代码中也不能静态引用任何具体角色类。

  使用规范（specification）对象可以达到这样的效果。当客户需要请求一个角色时，它传递一个规范对象给核心对象。最简单的解决方案是：把类型名字当作规范使用（参见“代码示例”一节）。核心对象返回与规范相匹配的角色对象。在[Riehle95, Evans+97]中讨论了更多的详细的规范机制。

  创建角色对象时也可以使用同样的规范。为了达到这个目的，你可以使用Product Trader模式

[Bäumer+97]。角色对象的trader维护了一个容器，其中包含了规范对象及与之相关的creator对象（例如类对象、原型（prototype）或是标本（exemplar））。当一个客户程序想添加一个新的角色时，它把一个规范对象传递给核心对象。核心对象会把创建过程委托给这个trader。

- *选择合适的规范对象*。在很多案例中，使用类型名称作为规范对象就已经足够了。但有时候也值得使用更加复杂的规范：

  假设你设计了一个叫做Person的核心类。一些人（person）可能是雇员（employee），所以这里应该有一个Employee角色。因为雇员也有各种各样的，你可能还希望把Employee设计成一个接口并且设计一些具体角色作为Employee的子类，比如售货员（Salesman）、开发者(Developer）和经理(Manager)。当客户程序需要获得一个人的工资信息时，它会向核心对象请求“Employee”角色。如果只用类型名称作为规范你将不能这么做，因为这些具体的角色对象的类型名称将是“Salesman”之类的名称，而不是“Employee”。在这种情况下，你可以使用类型对象（Type Objects）[Johnson+97]作为规范。通过对子类/超类关系的评估，核心对象就可以找回客户程序请求的角色对象了。

- *管理角色对象*。为了让核心对象管理它的角色，Component接口声明了一个角色管理协议，其中包括对角色对象的添加、移除、测试、查询等操作。为了支持角色管理协议，核心对象维护一个字典（dictionary），其中的内容是角色规范到具体角色实例的映射表。当一个角色对象被附加到核心对象上时，这个新的角色对象与它的角色规范一起被注册到角色字典中。

  请注意：一个核心对象是通过ComponentRole类型的角色对象引用来管理它们的。绝对不要在这里使用ComponentRole的实例！因为这样做会使核心对象无法鉴别它们**真实的**类型。另外，由于核心对象拥有它的角色，它必须细心管理它们。特别是，核心对象在自己被销毁（delete）的时候必须负责销毁它拥有的角色对象。

- *维护核心对象和角色对象的状态一致性*。核心对象或角色对象发生改变时，其他的角色对象也能也需要更新。下面是一个例子：改变一个Person的名字，而这个Person同时还是一个Borrower。当这个Person接收到它的新名字的时候，在Borrow的面前应该有一个摇晃着小旗的人告诉它：“你的名字被改变了！”这个摇旗者同时也告诉系统：“一个贷款者更改了他的名字，你应该把这个事件报告给管理贷款者信用的国家机构。”——对于银行来说，这样的通知是必须的。有几种可能的解决方案可以确保这些约束，但是每种方案都会带来额外的开销；并且这些依赖会导致编码的困难。我们将在下一个条款中讨论一个更详细的解决方案。

- *用Property和Observer模式来维护角色的属性约束*。如果很多的相互依赖使状态的集合变得复杂，核心对象的属性（或其中的一部分）可以用一个属性列表（property list，[Riehle97]）来实现——属性列表也被称为变量状态（Variable State，[Beck96]）。属性列表是一个键/值（key/value）二元组列表，我们用它来分别表现属性的名称和值。它常见的实现方式是一个字典，在其中将属性名称映射到属性值（对象）上。

  然后，一个角色对象与一个特定的属性——这个属性被定义为核心对象状态的一部分——联系在一

起。当一个改变此属性的方法被调用时，角色对象会得到一个通知。为了做到这一点，每个可以改变核心对象属性值的角色对象都必须在它这样做的时候通知核心对象，这样核心对象才能向相关的角色对象发出通告。

属性列表通常被看成一种不好的东西，因为它们破坏了对实现的封装。如果属性的名称发生改变，可能所有与之相关的角色类都需要相应改变。而且，差劲的代码可能会带来隐蔽的错误。但是，只要小心处理，这些问题都是可以避免的。这个模式最广为人知、应用最广泛的例子可能是抽象语法树上的带修饰节点（decorated nodes），这是许多编译器和软件开发环境的基础抽象。

- *维护概念上的身份（conceptual identity）*。Role Object模式让你可以象管理一个有综合状态的概念对象（conceptual object）一样管理核心和它的角色。因此，你必须让客户有办法确定两个截然不同的技术对象（technical object）是否是同一个逻辑对象（logical object）的部分，它们是否拥有同样的概念身份。当不同的角色对象拥有同样的核心对象时，它们将共享同一个概念对象身份。

  比如说，假设这样的一个客户应用程序：它使用Component接口直接处理核心对象，并通过一个具体的角色接口引用核心对象唯一的角色实例。从技术的观点来看，这两个引用不可能指向同一个对象，因为它们引用了两个——在技术上——截然不同的对象。因此，为了发现“这两个引用实际上表示同一个概念对象”这一事实，Component接口必须提供特殊的身份比较方法，然后由客户程序来使用这一方法。通常这种身份比较的实现方法是：直接对两个核心对象的引用进行比较。

- *维护角色之间的约束*。在角色（不仅仅是它们的状态）之间可能有一些约束。一个常见的案例是：核心对象只有在扮演角色A的情况下才能再扮演角色B。例如：如果Customer和Borrower都是Person的角色，那么Customer角色对象的存在就是Person能扮演Borrower角色的前提。这些是角色级别的约束。对象扮演角色B的能力受限于对象是否已经扮演角色A。如果对象没有扮演角色A，则它一定不能扮演角色B。如同我们在上面的例子中看到的，这些约束是来自应用领域的。

  通常说来，角色B不会仅仅依赖于角色A的存在，而是会依赖于角色A存在于某种特定的状态。在最复杂的案例中，你将不得不使用一个约束解决系统（constraint solving system）。幸运的是，在实际应用中，情况几乎永远不会变得那么复杂，很多实际用到的解决方案都足以应付。在典型案例的解决中，我们用了一个两阶段提交协议（two-phase commit protocol），这个协议在最终做出什么请求——比如对一个角色对象的移除——之前先询问所有的角色，从而保证了约束的完整性。

- *递归使用Role Object模式以维护角色之间的约束*。许多角色级别约束的问题都可以通过递归使用Role Object模式得到解决。如果角色A是角色B、C、D……的前提，那么角色A可以被看成这些角色（B、C、D）的基础抽象。“贷款者（Borrower）”和“投资者（Investor）”可以看成是“客户（Customer）”的角色，而“客户”和“担保人（Guarantor）”又可以看成是“人（Person）”的角色。因为做一个担保人不需要首先是一个客户，所以你不必把Guarantor设计成Customer的角色。Role Object对象递归使用的一个范例见下图：

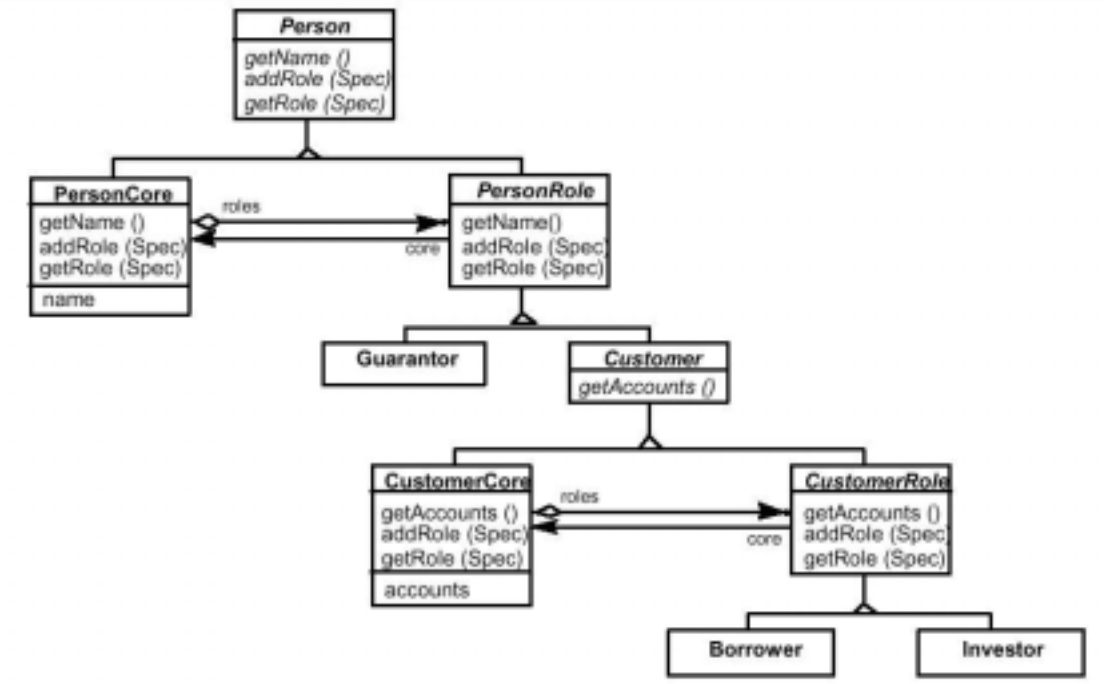


图4：递归使用Role Object模式

在运行时，这种递归使用形成了一条角色对象和核心对象的链（chain）。下面的图展示出了这个情况：

图5：角色上再附加角色形成的动态对象图

我们的角色级约束是：除非Customer角色已经存在，否则不允许Borrower或Investor的角色对象出现。在上面的解决方案中，这个约束已经很简单的被实现了。由此可见，把Customer设计成更具体角色的另一个基础抽象，这样可以为Person基础抽象提供很重要的角色约束保证。

代码示例

下面的C++代码展示了“动机”一节中讨论的客户范例的实现。我们假定存在一个叫做Customer的Component类。

```
class CustomerRole;

class Customer {

public:

// customer specific operations

virtual list<Account *> getAccounts() = 0;
```

```
// Role management
virtual CustomerRole * getRole(string aSpec) = 0;
virtual void addRole(string aSpec) = 0;
virtual void removeRole(string aSpec) = 0;
virtual bool hasRole(string aSpec) = 0;
};
```

CustomerCore类的实现可能是这样：

```
class CustomerCore : public Customer {
public:
CustomerRole * getRole(string aSpec)
{
return roles[aSpec];
};
void addRole(string aSpec)
{
CustomerRole * role = NULL;
if ((role = getRole(aSpec)) == NULL)
{
if(role = CustomerRole :: createFor(aSpec, this))
roles[Spec] = role;
}
};
list<Account *> getAccounts() { ... };
private:
map<string, CustomerRole *> roles;
};
```

角色的规范被实现为与其具体角色类名相同的字符串。角色规范与角色对象之间的映射关系用一个字典来实现。

然后，我们定义Customer的一个名叫CustomerRole的子类，我们将从CustomerRole再衍生出子类而得到不同的具体角色。CustomerRole通过核心对象的引用变量对CustomerCore进行装饰。对于Customer接口的操作请求，CustomerRole将之转发给核心对象。请注意，核心对象引用变量的类型是CustomerCore，这样可以保证客户角色不会被用做核心对象。角色规范和相应creator对象之间的映射表可以实现特定角色的实例化，这个映射关系可以用查找表（lookup table）来实现。在[Bäumer+97]中有对

creator对象的实现和管理的详细讨论。

```
class CustomerRole : public Customer {
public:
list<Account *> getAccounts() { return core->getAccounts() };
CustomerRole * addRole(string aSpec) { return core->addRole(aSpec); };
static CustomerRole * createFor(string aSpec, CustomerCore * aCore)
{
CustomerRole * newRole = NULL;
if (newRole = lookup(aSpec)->create()) newRole->core = aCore;
return newRole;
};
private:
CustomerCore * core;
};
```

CustomerRole的子类定义了特定的角色，比如Borrower类加入了一些操作用以处理债券、信用值等信息。子类不能对所继承的角色管理操作进行重载。

```
class Borrower : public CustomerRole {
public:
list<Security *> getSecurities() { return securities; };
private:
list<Security *> securities;
};
```

请注意，客户程序必须将从核心组件返回的角色引用进行向下转换，这样它们才能调用这个角色对象特有的操作：

```
Foo()
{
...
Customer * aCoustomer = Database :: load(“Tom Jones”);
Borrower * aBorrower = NULL;
if (aBorrower = dynamic_cast<Borrower *> aCustomer->getRole(“Borrower”))
{
// access securities
list<Security *> securities = aBorrower->getSecurities();
```

```
}
}
...
```

已知应用

这个模式被广泛应用于GEBOS系列面向对象银行项目中[Bäumer+97a]。这个项目为银行的出纳、贷款、投资、内部服务、记帐等几个部门提供了软件支持。GEBOS系统基于一个通用的商业领域层，并且对银行的核心概念进行了模拟。具体的应用程序则用Role Object模式扩展了这些核心概念。

在[Riehle+95a, Riehle+95b]中描述了实现Role Object模式所用的工具及原料框架，并对角色建模设计空间的问题进行了探索，其中用到了copy&paste协议、多继承、修饰和包装等等方法。Fowler对这些变化做了更简明的描述[Fowler97]。

在[Kristensen+96]中有Kristensen和Østerbye关于使用Decorator模式为编程语言引入角色的报告。但是，他们没有细致讨论角色对象的创建、管理的问题。作为例证，我们用了一个领域相关的例子——Person及其角色。这个例子是如此通用，以至它实际上已经成为了一个单独的模式。因为在许多环境中都需要一个“人（Person）”的抽象，都会要求它扮演各种各样的角色。Schoenfeld讨论了这个问题的一些例子，比如在以文档为中心的商务处理中的人及其角色[Schoenfeld96]。我们则用“客户”的角色把“人”和它的角色拿到银行商务的环境中。另外一个例子则是在政府官员体系及其相关的薪水管理问题中的人及其角色。

Role Object模式的一个与“人”无关的应用是对抽象语法树（abstract syntax tree，AST）的节点的修饰。AST是大多数软件开发环境最重要的抽象，许多不同的工具——例如语法导向的编辑器、符号浏览器、交叉引用、编译支持、分析依赖、变化影响——从不同的角度展示和使用了它。这些工具通常只对AST的一小方面感兴趣，并且需要用自己特定的信息对AST节点进行解释。使用Role Object模式可以很好的帮助这些工具为节点提供工具特殊的接口。Mitsui et al在[Mitsui+93]讨论了在C++编程环境下使用Role Object模式的情况，虽然这是对于具体情况的讨论，但它也实用于更广泛的目的。

相关模式

Extension Object模式[Gamma97]有同样的出发点：用满足环境需求的方式来扩展对象，从而使组件得到扩展。但是这个模式没有告诉我们一个实现上的关键方面：怎样透明地处理Component和ComponentRole的对象。而且Extension Object模式只关心扩展对象（角色对象）的创建和管理的问题。我们可以把Role Object模式的核心部分看成是Decorator模式和Product Trader模式的综合。

Zhao和Foster[Zhao+97]以及Schoenfeld[Schoenfeld96]都曾经在角色建模（role modeling）中使用过

Extension Object模式。Zhao和Foster把角色对象当做扩展对象来讨论，也就是说他们并不对核心对象进行透明的包装。他们的基本范例是一个运输软件系统，其中的关键对象是Point（及其角色）。Schoenfeld则选择了与我们相同的主要范例（Person及其角色），但他也使用了Extension Objects模式，而没有用Decorator来对核心进行透明包装。

[Fowler96]中的Post模式描述了Role Object模式的一个有趣的变化。与Extension Object模式相似，Post模式描述了某个特定应用程序环境中核心对象的职责。但是，一个Post对象的存在独立于核心对象，即使不被赋值给一个核心对象，Post对象也可以存在。

致谢

我们要感谢Ari Schoenfeld，我们的导师。他帮助我们对这个模式的表达和内容做了改进。

参考文献

[Bäumer+97] Dirk Bäumer and Dirk Riehle. “Product Trader.” In [Martin+97]. Chapter 3.

[Bäumer+97a] Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, and Heinz Züllighoven. “Framework Development for Large Systems.” *Communications of the ACM* 40, 10 (October 1997).

[Beck96] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice-Hall, 1996.

[Coplien+95] James O. Coplien and Douglas C. Schmidt (editors). *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[Evans+97] Eric Evans and Martin Fowler. “Specification Patterns.” Submitted to PLoP ’97.

[Fowler96] Martin Fowler. *Analysis Patterns.* Addison-Wesley, 1996.

[Fowler97] Martin Fowler. “Role Patterns.” Submitted to PLoP ’97.

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. 中文版：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社2000年9月。

[Gamma97] Erich Gamma. “Extension Object.” In [Martin+97]. Chapter 6.

[Johnson+97] Ralph Johnson and Bobby Woolf. “Type Object.” In [Martin+97]. Chapter 4.

[Kristensen+96] Bent Bruun Kristensen and Kasper Østerbye. “Roles: Conceptual Abstraction Theory and Practical Language Issues.” *Theory and Practice of Object System* 2, 3 (1996): 143-160.

[Martin+97] Robert C. Martin, Dirk Riehle, and Frank Buschmann (editors). *Pattern Languages of Program Design 3.* Addison-Wesley, 1997.

[Mitsui+93] Kin'ichi Mitsui, Hiroaki Nakamura, Theodore C. Law, and Shahram Javey. “Design of an Integrated and Extensible C++ Programming Environment.” *Object Technology for Advanced Software* (ISOTAS-93, LNCS-742). Edited by Shojiro Nishio and Akinori Yonezawa. New York: Springer-Verlag, 1993. Page 95-109.

[Riehle+95a] Dirk Riehle. “How and Why to Encapsulate Class Trees.” In *Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’95). ACM Press, 1995. Page 251-264.

[Riehle+95a] Dirk Riehle and Heinz Züllighoven. “A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor.” In [Coplien+95]. Chapter 2.

[Riehle+95b] Dirk Riehle and Martin Schnyder. *Design and Implementation of a Smalltalk Framework for the Tools and Materials Metaphor.* Ubilab Technical Report 95.7.1, 1995.

[Riehle97] Dirk Riehle. *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose.* Ubilab Technical Report 97.1.1. Zürich, Switzerland: Union Bank of Switzerland, 1997.

[Riehle+98] Dirk Riehle and Thomas Gross. “Role Model Based Framework Design and Integration.” In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’98). ACM Press, 1998.

[Schoenfeld96] Ari Schoenfeld. “Domain Specific Patterns: Conversions, Persons and Roles, and Documents and Roles.” In *Proceedings of the 1996 Conference on Pattern Languages of Programming* (PLoP ’96). Washington University Department of Computer Science, Technical Report WUCS-97-07, 1997.

[Zhao+97] Liping Zhao and Ted Foster. “A Pattern Language of Transport Systems (Point and Route).” In [Martin+97]. Chapter 23.

原文链接：http://www.umlforum.com/zippdf/P25.zip

KCOM 技术介绍

KCOM 是 KCOM 公司独立开发的一系列开发平台和计算机语言技术的总称。
它包括：KCOM 组件模型，KCOM Basic 语言，KCOM Stage 运行环境，KCOM Space 开发平台四项技术。

KCOM 组件标准：
KCOM 组件标准是一个完善的组件标准，可以自定义属性，方法和事件，并使用事件驱动的方式编程。同时源代码方式存储有利于在网络上传输。

KCOM BASIC：
语法结构完善，易学，可以使用中文进行编程，例如使用中文变量名称，中文方法名称，属性名称等等。独有组件运算功能能够实现代码的可视化。

KCOM Stage：
KCOM Stage 是 KCOM 组件的解释器，由于采用了代码的结构化存储，在解释之前不需要对语句进行扫描和文法处理，同时采用了高效的寻址方式，所以比一般的脚本语言的解释效率要高。

KCOM Space：
能够方便的编写 KCOM 组件和代码，是一个可视化开发平台，具有丰富的界面排版，代码编写和代码调试功能。

KCOM Space 以其自定义的组件标准(KCOM 组件标准)为核心，完整地实现了一门计算机语言（KCOM Basic)、一个完整的组件化开发平台（KCOM Space）和一个高效的虚拟机（KCOM Stage)。作为国产软件在高端领域的突破，KCOM 表现出了与世界同步的技术水准。由于采用了全面组件化的思想，抛弃了传统的设计方法，全面革新了开发平台从底层的语言到上层的操作界面的设计。

KCOM 公司的产品在 2000 年中国国际软件博览会上获得创新奖。并受到倪光南院士的高度评价。

KCOM 公司合作意向：

KCOM 公司在长期对于开发平台和计算机语言技术的关注与不断开发的基础上，形成了自己独特的技术特长与优势。在如今的软件领域，软件和开发平台日益紧密成为趋势，其中突出地表现在：

软件与基于该软件的二次开发平台：如 MS Office，Notes，GIS 应用软件及开发平台，3D Max，AutoCAD，Director 等等应用软件。 一些稍小的软件如 InstallShield 也在其中使用了自己定义的解释语言，以提供强大的扩展功能。

企业流程快速建模与软件生成工具：此类工具在国外的企业应用解决方案市场中被大量使用。而国内的系统集成企业正在开始开发和形成自己的解决方案与定制工具。

嵌入式企业应用前端设备：该设备是我们所看到的一个巨大的市场机会，基本原理在于软硬件一体的显示与操作前端，采用 NC 体系，运行企业应用。

在以上几个领域中，KCOM 公司都能够以自己长期的积累和技术特长，与合作方展开广泛的技术合作，希望能够来信探讨：zhangiii@163.net

http://www.kcomspace.com.cn/

Matcher-Handler（匹配器-处理器）模式

Frank Metayer著 ，透明 译

讨论

意图

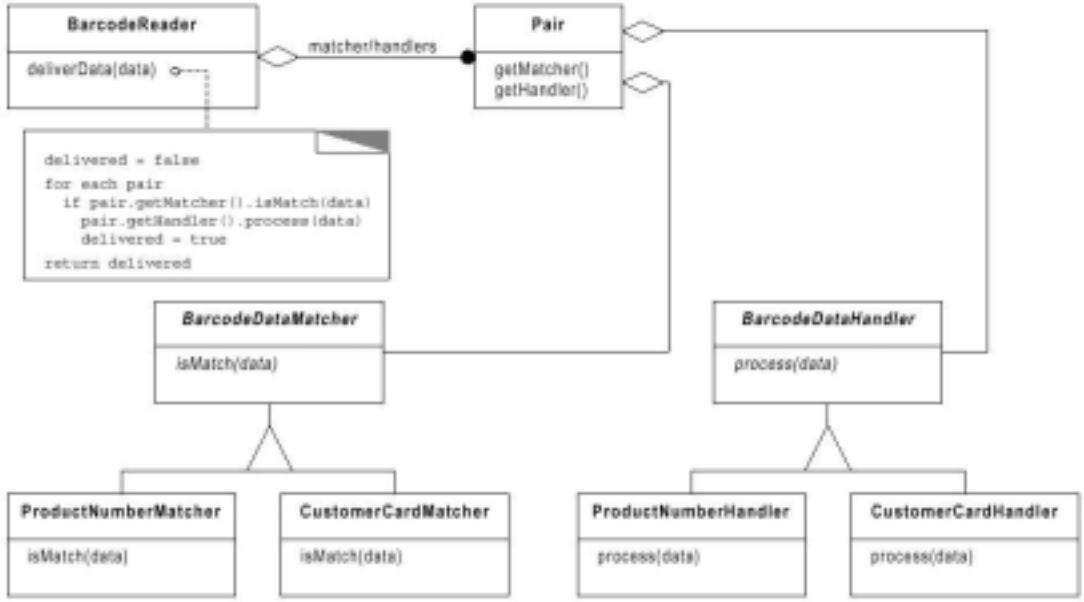
Matcher-Handler模式定义了一个普遍的机制，用一种松耦合的方式将原始数据分发传递给一个或多个数据处理器。这个模式显式地将数据鉴别匹配（Match）的责任从数据处理（Handle）中分离出来。

动机

许多工业用的电脑在输入设备的配备上都有多样性，它们可以从周围环境中接受多种不同类型的输入数据。比如说，象现金出纳机和抽奖终端等“售货点”型设备通常都配备有一个或多个光学扫描设备。现金出纳机上的光学扫描设备是一个条码阅读器，它将读出商店的货物、优惠券、消费卡……上的条码数据。当它扫描到一个条码之后，出纳机的前端应用程序必须对原始数据进行足够的处理以确定数据的类别（例如：优惠券），然后将数据传递给合适的应用对象进行处理。开发一个有普遍性的数据路由框架来分发这些原始数据对于其他应用的开发是很有好处的。

这样的一个框架应该有如下的特征：对所有输入设备——不管它们输入的数据是何种类型——提供一致的行为；明确定义应用编程接口（API）以引入对新的数据格式的支持；在增加对新的数据格式的支持时，将同时引入bug的风险减至最小。

Matcher-Handler模式可以用做开发这样一个框架的基础，因为它概括了数据识别和数据分发的方法。上面描述的条码例子中相关的类可见图1（见下页）。



BarcodeReader实现了应用程序的前端部分，其职责是从输入设备接收数据[1]并把数据分发给适当的实体进行处理。它的实现是普遍的、可扩展的。它管理了一个“匹配器/处理器”二元组列表，并将所有的数据识别及处理都分别推迟分配给这些对象。

BarcodeDataMatcher和**BarcodeDataHandler**这两个抽象类定义了对数据进行匹配和处理所需的普遍接口。具体的匹配器——比如**CustomerCardMatcher**——对原始数据进行查询以判别数据是否与特定的规则相匹配（这些规则包括数据的尺寸、数据头的内容等等）。如果数据通过了验证，则函数**isMatch()**返回true；否则返回false。而具体的处理器——象**CustomerCardHandler**——只有当数据被确定为它们特定的类型时才会被通知。

从**BarcodeReader::deliverData()**方法的伪码中我们可以看到，可能有多个处理器同时接收到同样的数据。这对某些数据处理的方法可能是有利的。但是你也可能你不希望使用这种分发数据的方式，那么你可以重新构建这个for循环以保证只有一个处理器接收到数据。（**BarcodeReader**一种更为灵活的实现方式是：把for循环的行为提取到**BarcodeReader**类的外部，放置到一个strategy [Gamma+95] 对象中，并在**BarcodeReader**被创建时将strategy的引用传递给它。）

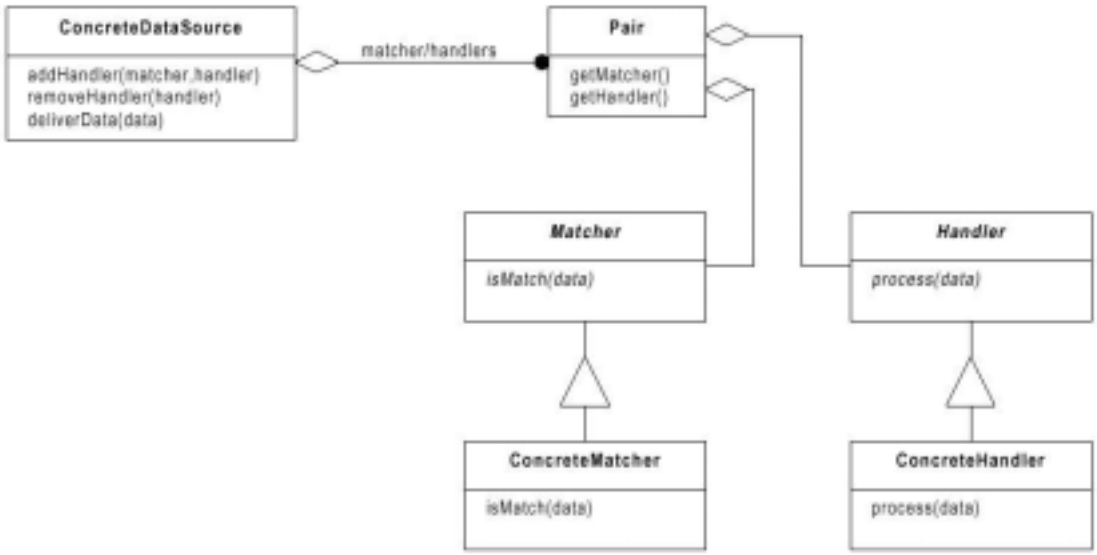
适用性

Matcher-Handler模式适用于以下的情况：

- 在不明确指定接收者的情况下分发数据。
- 需要有多个对象同时而又各自独立的处理同一个数据事件。
- 需要动态指定处理离散数据事件的对象。
- 需要用一个处理器处理来自多个数据源的数据。
- 你希望在未来可以很方便地为这个应用程序引入新的数据类型。

[1]通常此数据不是以它自己的对象形式出现，而是作为简单的原始数据出现并被鉴别的。

结构



参见图2：类关系图

参与者

- **ConcreteDataSource (BarcodeReader)** ——具体数据源

  - 对数据源（例如：输入设备）进行控制，接收或产生数据事件。当它要开始给数据处理器传递数据时，调用自己的deliverData()方法。

  - 查询匹配器以确定与之相关的处理器是否应该接收一个数据，然后把数据传递给合适的（一个或多个）处理器。

  - 通过“注册方法”addHandler()和removeHandler()管理一个匹配器/处理器二元组序列。

- **Matcher (BarcodeDataMatcher)** ——匹配器

  - 定义一个接口，数据源用这个接口来鉴别一个特定的数据事件是否与一个特定的处理器需要的标准相匹配。

- **ConcreteMatcher (CustomerCardMatcher)** ——具体匹配器

  - 实现匹配器接口以鉴别数据是否符合标准。

- **Handler (BarcodeDataHandler)** ——处理器

  - 声明一个接口，数据源用这个接口把数据传递给处理器。

- **ConcreteHandler (CustomerCardHandler)** ——具体处理器

  - 处理数据源按照应用程序定义的某种途径传递来的数据。

协作

1. ConcreteDataSource向每个匹配器询问一个数据事件是否与它们各自包装的匹配标准相匹配。
2. 如果数据与某个匹配器的标准匹配（即：isMatch()返回true），ConcreteDataSource将数据传递给与

此匹配器相连的处理器。此处理器将处理这些数据。

3. 如果数据与某个匹配器的标准不匹配，与之相连的处理器不会得到通知。ConcreteDataSource将查询列表中的下一个匹配器。

效果

Matcher-Handler模式有以下优点：

- 降低耦合度。使用这个模式，数据源不必知道“哪个对象可以处理哪种数据事件”这样的事情。现在数据源只知道数据被应用程序接收或忽略。数据源和数据处理器不再需要对对方有任何显式的了解。
- 数据处理的运行时适应性。因为处理器序列是动态指定的，所以应用程序可以在任何时候更换当前使用的处理器。如果一些处理需求发生变化，应用程序可以增加或移除处理器。借助Memento模式[Gamma+95]或者其他相似的机制，应用程序还可以再恢复以前的处理器配置。
- 使应用程序容易被扩展。Matcher-Handler模式提供了一个响应自发数据事件的框架的基础，对于这个框架来说，温和地忽略未被承认的或尚不支持的数据也是很重要的。这在一个全新的应用程序的开发早期是特别重要的，在那个时候，整个应用程序的全貌还没有呈现出来。当你创建一个新的对象以支持一种新的输入数据类型时，这个对象已经被注册到合适的数据源、可以开始接收输入数据了。
- 减少了引入新功能带来的风险。为支持新的输入数据类型而引入的代码不会要求数据源的代码做一行改动。这也就是说，Matcher-Handler模式可以用于开发具有扩展性的应用框架，而且此框架可以以二进制形式（而不是源代码形式）发布。更重要的是，“引入新功能而不需改变数据源”意味着“引入新的处理器时破坏原先存在的代码”这样的风险被大大降低了。

Matcher-Handler有这样一个缺点：

匹配器可能对其它的匹配器有过多的了解。如果数据不是作为其原来的对象、而是作为简单数据被处理，则具体的匹配器会知道其他匹配器针对特定数据源的实现。导致这种情况的原因是，一个匹配器必须知道与它匹配的数据相对于其他数据的特点。所以，它一定会知道其他数据类型的一些情况。

举例来说，仓储产品的条码数据是标识这种产品的一个数字，用户信用卡的条码数据可能包含一个消费者的名字、住址等等信息。如果预先规定产品标志总是被编码为8字节的整数、用户信息总是被编码为40字节的整数，那么这个应用程序将只能处理这两种数据，匹配器也只能根据数据的长度来实现其匹配细节。

当不得不引入新的数据类型时，这种假设将被打破。由于以前的假设被打破，匹配器将必须用新的匹配标准重新实现。但是请注意，这个问题不会影响数据处理过程，因为处理过程被隔离到另一个类——处理器——中了。这正是我们最初、最主要的动机：将匹配器的实现从处理器的实现中分离出来。

实现

在实现Matcher-Handler模式时，应该牢记以下几个要点：

- 在“动机”一节中介绍的例子声明了匹配器的接口，这个接口返回一个boolean值，以此指出一个数

据是否应该被传递给与匹配器相连的处理器。但在一个正规的应用程序中，匹配器最好能返回一个更高级的对象。当数据通过验证时，这个对象将原始数据包装在其中；当数据未通过验证时，匹配器返回null（或者一个Null对象[PIoPD3]）。数据源将这个匹配器返回的对象传递给处理器。这样，处理器不需要重新解释（甚至不需要知道）原始数据。同时，如果处理器操作高级对象而不是原始数据，它可以对这些对象进行多态操作，而这些对象的具体类型只有匹配器才知道。

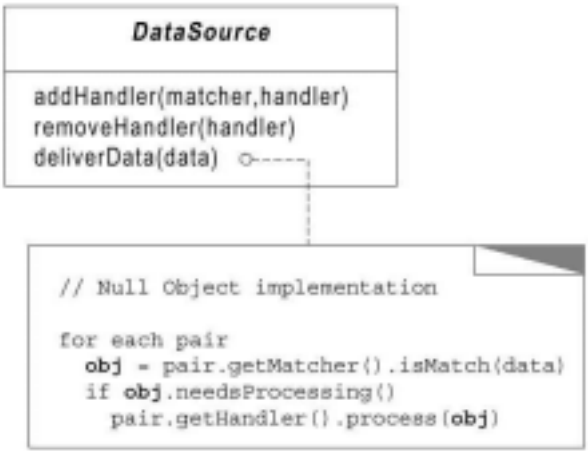


图3：匹配器的变化

- 在同一个类中实现匹配器和处理器的接口，以使一个对象可以进行匹配和处理的操作。这将减少应用程序中类与对象的数目及——外观上的——复杂性，但会带来很大的风险。如果一个对象同时可以进行匹配和处理的操作，在实现这些行为时就会有这样的倾向：匹配器通过实例变量向处理器传递信息。当新的数据类型被引入、匹配标准必须改变（参见“效果”一节）的时候，由于两个方法之间微妙的依赖的影响，很有可能你必须同时修改数据处理的代码。
- 当数据的处理过程可能会耗费相当长的时间时，deliverData()方法更好的实现可能是：一次性查询所有的匹配器，然后创建一个应该接收数据的处理器的列表；将数据传递给这些处理器之后，将它们的处理过程放到一个优先级较低的后台线程中。用这样的实现方法，对deliverData()方法的调用将更快得到回应。这将使数据源可以马上向用户提供一个声音和/或图象的反馈，以提示用户数据是否被接收或拒绝。（deliverData()的行为方式是由程序员决定的，它受很多因素影响。可以考虑用Strategy[Gamma+95]模式实现deliverData()方法。）
- 如果一个应用程序中有几个不同但是相似的数据源，为**ConcreteDataSource**引入一个抽象的超类是有用的。这个新的超类可以实现匹配器/处理器的注册和数据分发等行为。具体的子类则把注意力集中在输入设备的控制上。抽象超类的加入使数据处理器接收来自不同数据源的数据变得简单。举例来说，一个单独的处理器实体可以接收并处理所有的数据，不论它们来自条码阅读器、磁性条纹阅读器、还是智能卡阅读器，而且处理器不必为任何输入设备做任何的准备。

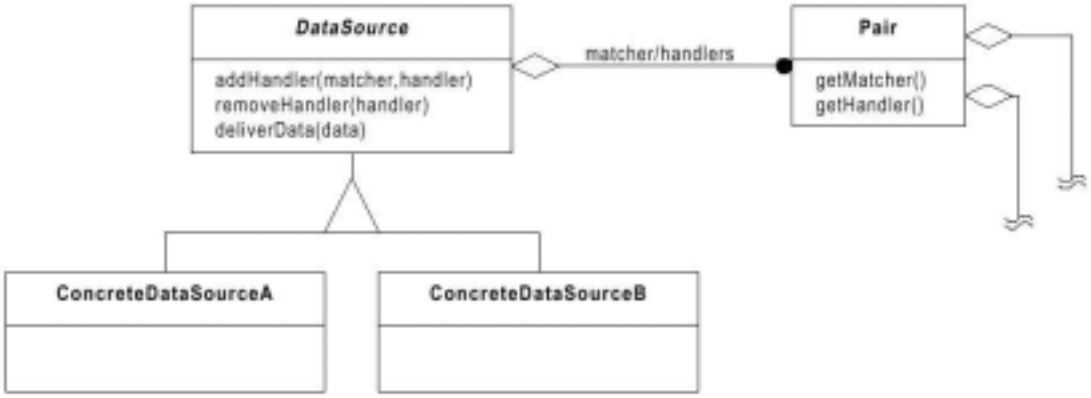


图4：抽象数据源

- 数据源用Pair对象来维护匹配器和处理器之间的联系，这只是连接这两个对象的几种方法中的一种。这个变化提供了相当大的适应性，因为它允许一个处理器的实例被注册给几个不同的匹配器、或是一个匹配器的实例被注册给几个不同的处理器。如果这个多对多的关联不是你需要的，那么请考虑使用这种方法：把一个对象的显式引用提供给另一个，从而把匹配器和处理器连接起来。下面的图显示了这样的情况：每个匹配器拥有与之相连的处理器的显式引用。

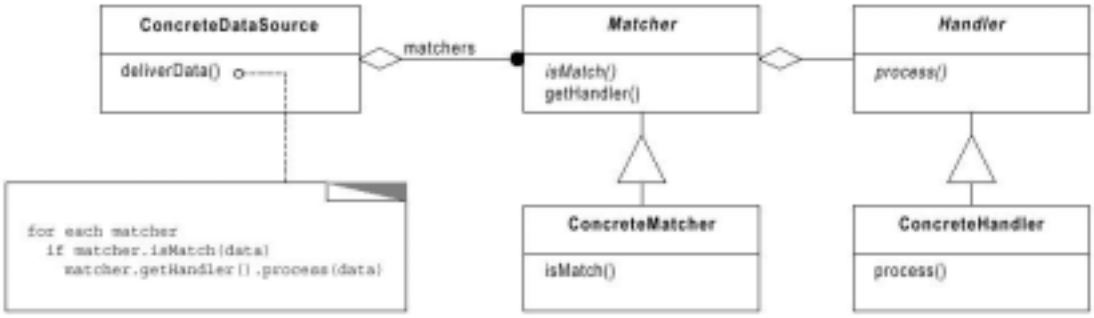


图5：匹配器拥有处理器的引用

代码示例

这是“动机”一节中讨论的“条码阅读器”（参见图1）的一个java实现。示例代码从**BarcodeReader**开始。因为我们不想在这里讨论具体的输入设备，所有控制输入设备的代码都被省去了。

```
public class BarcodeReader extends Thread
{
private Vector _handlers;
// Matcher/Handler registration.
public void addHandler( BarcodeDataMatcher matcher,
BarcodeDataHandler handler )
{
_handlers.addElement( new Pair(matcher,handler) );
}
```

```
// Deliver data to the appropriate handler.
protected boolean deliverData( byte[] data )
{
Enumeration enum = _handlers.elements();
boolean delivered = false;
while( enum.hasMoreElements() )
{
Pair pair = (Pair) enum.nextElement();
if( pair.getMatcher().isMatch(data) )
{
pair.getHandler().process( data );
delivered = true;
}
}
return delivered;
}
// Device Thread. Accept data and deliver it, forever.
public void run()
{
while( true )
{
byte[] data = _waitForInput();
boolean delivered = deliverData( data );
if( delivered )
_flashGreen();
else
_flashRed();
}
}
...
}
```

下面的两个类是匹配器接口和一个具体的匹配器实现。这个匹配器以数据长度为根据，判断数据是否是一个产品标志号。

```
public interface BarcodeDataMatcher
{
public boolean isMatch( byte[] data );
}
public class ProductNumberMatcher implements BarcodeDataMatcher
{
public boolean isMatch( byte[] data )
{
return data.length == 8;
}
}
```

下面的两个类是处理器接口和一个具体处理器的例子。这个产品标志处理器根据标志号造出新的产品实例（用factory method[Gamma+95]模式），然后根据一个销售者名单（singleton[Gamma+95]模式）邮寄产品实例。

```
public interface BarcodeDataHandler
{
public void process( byte[] data );
}
public class ProductNumberHandler implements BarcodeDataHandler
{
public void process( byte[] data )
{
// 'data' is an ASCII encoded integer
String ascii = new String( data );
Integer pid = Integer.decode( ascii );
Product prod = Product.newInstance( pid );
Sales.getInstance().add( prod );
}
}
```

主程序代码可能会是这样：

```
public class Application
{
public static void main( String[] args )
```

```
{
BarcodeReader reader = new BarcodeReader();
BarcodeDataMatcher matcher;
BarcodeDataHandler handler;
matcher = new ProductNumberMatcher();
handler = new ProductNumberHandler();
reader.addHandler( matcher, handler );
matcher = new CustomerCardMatcher();
handler = new CustomerCardHandler();
reader.addHandler( matcher, handler );
// add more handlers...
reader.start();
}
}
```

已知应用

- **Altura[2]图象阅读器。**这是一个在诸如收据、游戏名单、游戏者注册卡等文本上扫描的图象阅读器。扫描器获得的数据通过一系列的匹配器-处理器二元组被传递给应用程序。一个类型的数据只被分发给要求得到它们的处理器。处理器在一个独立的后台进程中被调用，所以图象阅读器可以很快的向用户提供接收或拒绝的反馈。
- **Altura通信框架。**在抽奖终端运行之中，它将从中心计算机系统那里收到很多它不请自到的消息。这些消息包括新闻、游戏结束通知、获胜号码公告等等。处理这些类型的消息的处理器每一个都和其他的处理器有很大的区别。举例来说，游戏结束通知会被分发给一个运行在终端上的游戏原型（prototype[Gamma+95]），这个原型将告诉游戏者“游戏已经结束了”；获胜号码公告可能会被分发给另一个游戏原型，这个原型将发出“现在将重新开始一局新的游戏”的信号。在这个实现中，只有一个匹配器进行消息的匹配。这个匹配器读入消息报文的报头信息，并根据报头的内容进行分发。
- **Windows注册表。**当一个windows95/NT应用程序被安装到计算机上的时候，它将在注册表中唯一标志自己。应用程序告知注册表：对于在某个特定的文件类型上的Open、Print等等请求，它可以处理。当用户提出对一个指定的文件的Open请求时，windows判别文件的类型（根据文件扩展名）、并与注册表内记录的文件类型做匹配。如果找到一个匹配的结果，相应的应用程序——它已经注册支持Open操作——将被调用以处理这个Open请求。

[2] Altura™是由GTECH公司开发的一个抽奖终端。驱动这个终端的应用程序是一个基于框架的面向对象的JAVA软件。

- **Smalltalk的方法选择。**当一个消息被传送给一个对象时，Smalltalk会用一种名叫“方法查找”的机制来决定调用哪个方法来处理这个消息。Smalltalk对消息的标记和对象的类中定义的方法标记进行比较。如果找到一个匹配的方法标记，Smalltalk将调用与此标记对应的方法来处理这个消息。
- **远程过程调用（RPC）服务分派。**当一个服务器收到一个远程过程程调用请求时，一个分派器会识别出本地机器上可以为这个请求服务的过程，并调用这个过程来处理这个请求。

相关模式

和Observer模式[Gamma+95]相似，Matcher-Handler的主要工作是分发信息，使这些信息看起来就象是发生在它们感兴趣的对象身上，尽管这些对象实际分布于整个应用程序之中。

Observer的焦点在于：当某个目标的状态发生改变时通知应用程序对象。目标的状态发生任何改变，其观察者都将被通知到，而不是仅仅在观察者关心的改变发生时才通知。与之形成对比的是，Matcher-Handler模式中的处理器（观察者）只有在它们等待的特定信息到达数据源（目标）时才会被通知到。

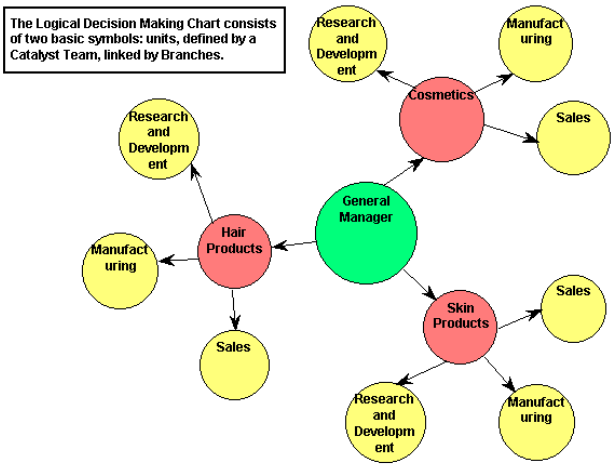
Matcher-Handler模式中的匹配器对象是Strategy[Gamma+95]模式的一个范例，尽管它们的动机有一些差别。Strategy模式的目标是——特别是在运行时——提供可选择的行为。匹配器对象的动机则是来自如下的事实：匹配标准在应用程序的整个生命周期中都有可能变化，将匹配行为隔离出来对于减少匹配行为的改变对系统其他部分的影响是有意义的。

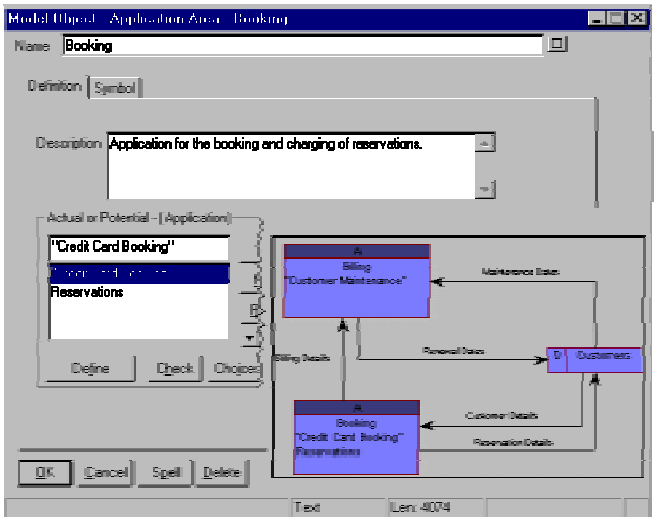
Matcher-Handler模式和Chain of Responsibility[Gamma+95]模式有相似之处：避免数据源（client）与其处理者的耦合，允许多个对象处理一个请求，准许动态指定处理者。Matcher-Handler模式的结构与Chain of Responsibility也有不同之处：Matcher-Handler中处理器的先后顺序取决于数据源；而Chain of Responsibility中处理器的顺序是预先确定的，并且由相连的处理器共同维护。

参考文献

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.中文版：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社2000年9月。

[PloPD3] R. Martin, D. Riehle, F. Buschmann. Pattern Languages of Program Design. Reading, MA: Addison-Wesley, 1998.





Alternator（置换器）—对象行为型模式

John Liebenau 著，透明 译

讨论

1．意图

在一个层次化结构（hierarchical structure）中允许多个候选子树（alternative subtree），并且对客户（client）隐藏这个结构的细节内部。

2．动机

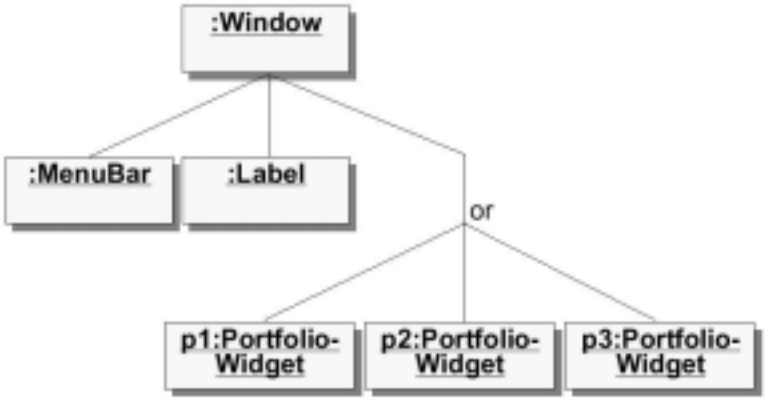
假设我们正在开发一个证券（stock）交易应用系统。我们的主要目的是为交易者（trader）提供对证券组——股票（portfolios）[1]——的可视化访问，交易者可以通过它买入、卖出股票。我们将使用一个面向对象的GUI框架来构建这个应用程序的图形用户界面，其中的每个窗口（window）都将包含一个可视化元件的体系（hierarchy）。组织这个应用程序的一条简单的途径是：在每支股票自己的窗口中显示它。通过选择一个窗口，交易者就可以选择一支股票进行处理。在通常情况下，创建股票与窗口之间的映射，这样就可以很好的满足我们的需要。但是，交易者的工作环境又向我们的应用程序提出了新的需求。

交易者的工作环境中已经存在另外的一些应用程序，它们对相关的证券信息进行显示和分析。一个典型的交易者可能需要同时使用多个工作站，或者需要显示与他/她的工作相关的新闻。新增加的视频显示和额外的窗口需求不可避免的造成了信息超载（information overload）。如果我们的应用程序只提供交易者需要的信息和当时所需的最小数量的窗口，以降低信息超载造成的影响，交易者会更喜欢使用它。

我们的证券交易系统需要提供这样一个机制：使用一个窗口选择并逐个显示股票信息。于是我们找到了问题的根本：窗口必须可以提供几种可选的视图（view）供交易者选择。下面的图展示了在三支股票的情况下我们的证券交易系统的元件体系。

---

[1] 译者注：我把“stock”译为“证券”、“portfolio”译为“股票”，以区别这两个词。

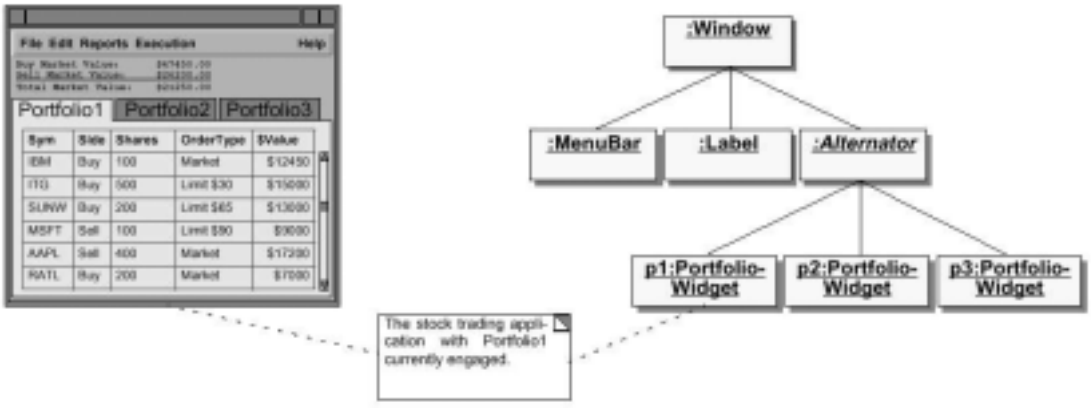


我们的证券交易系统的图形用户界面（GUI）只有一个窗口，它包含了一个菜单条（menu bar）、显示当前股票概要信息的标签（label）、以及显示股票信息的几个窗口部件（widget）。我们还需要一个机制用以：

- 管理股票窗口部件的放置和尺寸。
- 管理股票窗口部件的可见性。
- 管理对股票窗口部件的选择。

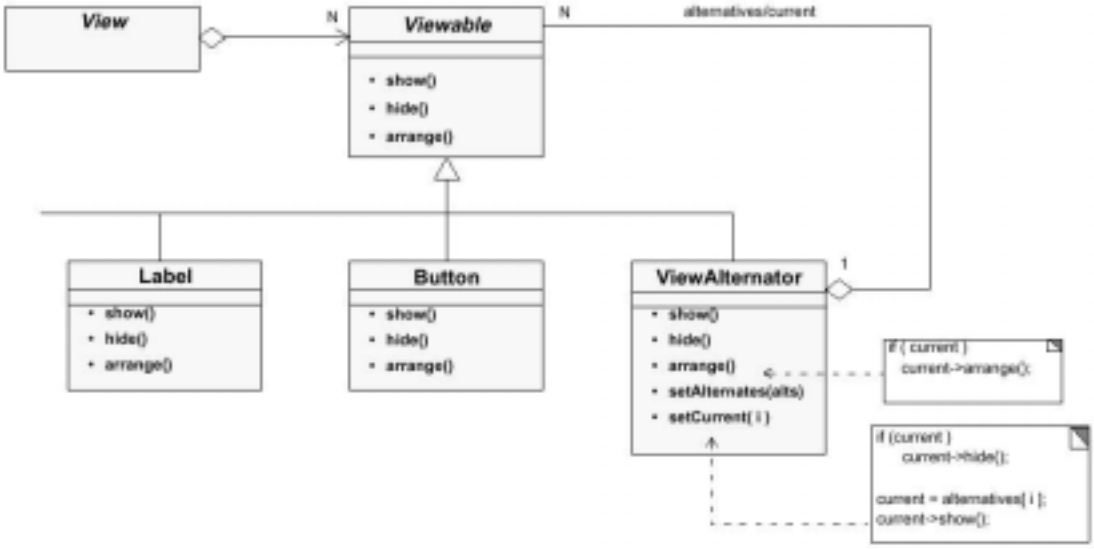
通过创建Window类的一个子类——这个子类提供处理我们的特定情况的定制算法——我们可以为我们的窗口对象实现上面的机制。但是，这个解决方案还有几个缺陷：它强迫我们在每个需要可选视图的地方写一个Window的新子类；它阻碍了我们在原来的GUI框架基础上发展新的算法。

为了解决这个问题，我们可以用一个置换器（Alternator）对象来管理对可选元件组的选择、可见性和排列。对于元件来说，置换器既是拥有者（place holder）、又是容器（container）。通过引入置换器元件来管理可选视图，我们对上图进行了扩展，得到下面的一个图：



下面的类图展示了一个小规模的图形用户界面框架的类结构。这个框架提供了Viewable类，它为所有的GUI对象定义了通用接口。View类维护一个Viewable对象的引用，Viewable对象组成了View对象的内容。Label、Button、ViewAlternator等类则对Viewable接口进行扩展，它们也有各自特有的操作。ViewAlternator

类维护它的可选元件及当前选择元件的引用。View类只把ViewAlternator对象看成另外一个Viewable对象，



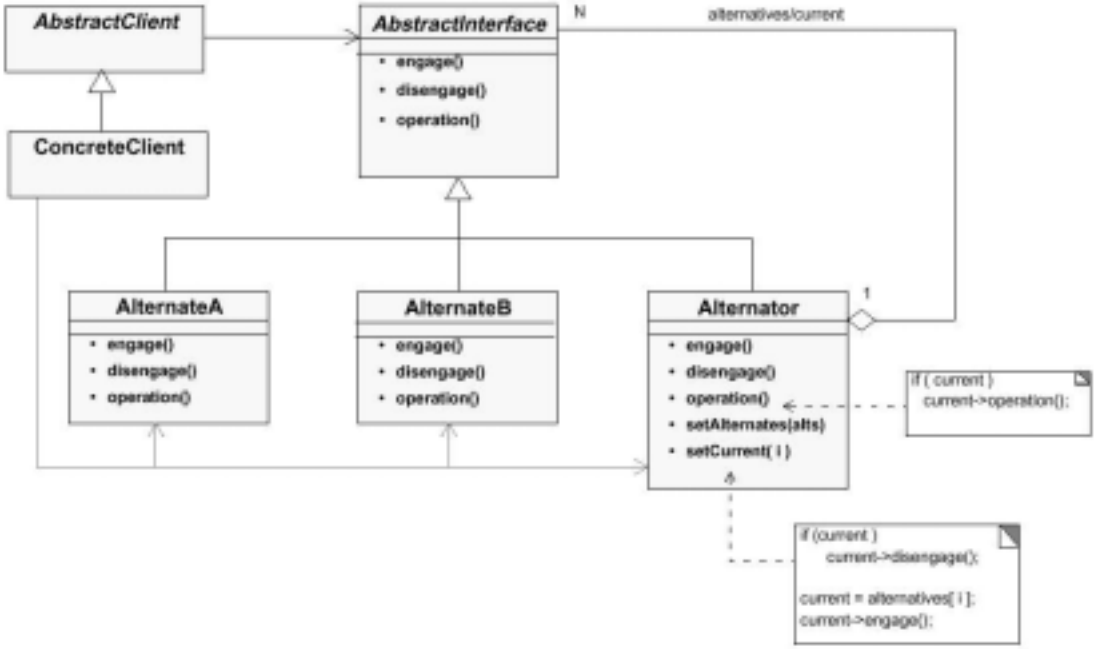
并且View类对一组可选元件的放置是一无所知的。

3．适用性

以下情况适用Alternator模式

- 一个对象是一个体系的一部分。
- 一个对象由几个子对象组成，但在同一时刻只使用一个子对象。
- 你希望单个对象在同一逻辑空间表示多个对象。
- 你需要从多个可选件中进行选择。
- 可选对象拥有相同的接口。

4．结构



5．参与者

• AbstractInterface (Viewable) —— 抽象接口

- 为Alternator和Alternate提供公用接口。
- 提供使用（engage）和挂起（disengage）操作（比如show和hide），并将之作为公共接口的一部分。

• Alternator (ViewAlternator) —— 置换器

- 提供设置选项、选择选项的接口。
- 提供一个机制，用以从一个选项转换到另一个（也就是说，挂起一个元件，使用另一个）。

• Alternate (Button,CheckBox,...) —— 可选件

- 提供AbstractInterface的一个具体实现。
- 可以提供与其他Alternate不同的附加接口。

• AbstractClient (View) —— 抽象客户

- 通过AbstractInterface使用Alternator和Alternate。

• ConcreteClient (SpecificView) —— 具体客户

- 通过Alternator和Alternate自己的特定接口使用它们。

- 可能有创建、配置Alternate和Alternator的责任。

6．协作

- Client使用AbstractInterface对对象进行操作。如果被操作的是一个Alternate对象，它直接处理Client的请求；如果被操作的是一个Alternator对象，它将这个请求转发给自己当前使用的Alternate对象。
- Client也可以使用Alternator类和Alternate类的特定接口。特别的，客户可以用Alternate特有的接口对它进行配置，然后用一个Alternate对象列表配置Alternator对象。Alternator当前使用的Alternate对象也由客户指定。

7．效果

Alternator模式有如下益处：

- *让几个可选对象看起来象一个对象*。这可以使使用这些元件的对象系统的算法结构得到简化，因为这些算法可以以统一的形式对待所有的元件。
- *透明的管理可选对象的激活和非激活*。这些可选对象在对象体系中占用同一个逻辑空间。在GUI的环境中，我们所说的“逻辑空间”也就是元件在屏幕上的排列。当一个新的可选对象被选中时，Alternator自动将先前使用的可选对象挂起。

Alternator模式有如下的不足：

- *可能过分要求可选件一般化（general）*。Alternate通过它们的公共接口而被客户操作。这个依赖于类体系的接口可能非常一般化。如果客户需要向下转换或者维护附加引用，他可能需要访问具体可选对象的特有接口。
- *可能禁止或阻碍对被挂起的可选对象的访问*。Alternator的设计使它在体系中看起来是一个单独的元件，但一些客户可能需要在访问当前使用的可选对象的同时也访问被挂起的那些。这会破坏Alternator作为单独元件的视感，从而抵消本模式的一些优点。

8．实现

在使用Alternator模式时，需要注意以下几个实现要点：

- *可选件的创建和置换器的配置*。典型的情况是：客户程序直接或者使用Abstract Factory模式[GHJV95]创建可选件和置换器，然后用合适的可选元件对置换器进行配置。在另外一些情况下，可能需要由置

换器负责创建它自己的可选元件。典型情况下，置换器会把它们的可选元件封装起来，不允许客户程序对它们进行任何直接访问。

- *保持使用和挂起行为的一致性*。作为置换器的元件应该在它们的使用/挂起方法与其他行为之间保持一致性。一般来说，元件应该有一些“开关”形式的变量。在“动机”一节讨论的例子中，屏幕上的GUI元件或者是可见的，或者是不可见的。可见的元件可以接收输入，不可见的元件则不能。
- *维护可选件的引用*。在强类型语言（C++，java）的实现中，Alternator只为体系中的元件提供一个公用接口（本例中的AbstractInterface）。如果你需要使用某个可选件特有的方法，你或者需要这个可选件的引用（与这个可选件类型相同），或者需要做向下类型转换。
- *管理可选件的状态*。有时候，可选件之间是完全独立的。只有当它们被使用的时候，它们的状态才会被更新。另一些时候，可选件之间有一定的相互依赖。在这种情况下，当前使用的可选件的状态被更新的时候，与之相关的其他可选件的状态也必须同时更新。这可以由Alternator元件来负责处理：如果一个操作改变了可选件的状态，置换器将把这个操作转发给每个可选件。以我们在“动机”一节中描述的ViewAlternator类为例，如果这个类实现了一个方法来设置它的x坐标或者y坐标，则所有可选件的坐标都需要同时被更新：

```
void ViewAlternator::setX(int x)
{
    for (vector<Viewable*>::iterator alternate( alternatives.begin() ); alternate != alternatives.end(); alternate++)
        (*alternate)->setX( x );
}
```

- *用Composite模式实现Alternator模式*。Alternator模式被用在层次化对象体系中，这些对象体系可以用Composite模式来实现[GHJV95]。Alternator模式可以看成是Composite模式的一个形式，但不同之处是Alternator模式引入了这样的一个概念：选择一个子对象，使用（或者说激活）它；其它的子对象都是非激活的。
- *同时使用多个可选件*。在某些情况下，可能需要同时使用多个可选件。为了实现这一目的，置换器要维护一个“当前使用元件”的列表，而不再是单个引用。此外还必须添加新的方法，用以设置当前使用的可选件。

9．代码示例

下面的示例代码是“动机”一节讨论的例子的程序片段。抽象类Viewable为所有GUI元件指定了通用接口。

```
class Viewable
{
```

```
public:
    // ...
    // Modifiers
    virtual void setX(int x);
    virtual void setY(int y);
    virtual void setHeight(int h);
    virtual void setWidth(int w);
    // Selectors
    virtual int getX()const;
    virtual int getY()const;
    virtual int getHeight()const;
    virtual int getWidth()const;
    // Actions
    virtual void arrange()=0;
    virtual void show()=0;
    virtual void hide()=0;
};
```

Button类扩展了Viewable抽象类，提供了按钮元件特有的方法。Button类对Viewable类中声明的纯虚函数做了实现。

```
class Button: public Viewable
{
public:
    // ...
    // Modifiers
    void setLabel(const string& lbl);
    void setCommand(Command* cmd);
    // Action
    void doCommand();
};
```

ViewAlternator类是一个GUI元件，它负责管理其他的GUI元件（也管理其他的ViewAlternator元件）。ViewAlternator类允许客户选择他们想要显示和激活的可选元件，而其他的可选件就被自动隐藏起来并且失去活性。

```
class ViewAlternator : public Viewable
```

```
{
private:
    vector<Viewable*> alternatives;
    Viewable* current;
public:
    // ...
    // Viewable: Actions
    void arrange();
    // Modifiers
    void setAlternatives(vector<Viewable*>& alt);
    void setCurrent(Viewable* v);
    // Selectors
    Viewable* getCurrent()const;
    int getNumAlternatives()const;
};
```

ViewAlternator对象将函数调用转发给当前使用的可选件。

```
void ViewAlternator::arrange()
{
    if ( current )
        current->arrange();
}
```

在选择可选件的时候，ViewAlternator使用（显示）一个可选件、并且挂起（隐藏）其他的可选件。

```
void ViewAlternator::setAlternatives(const vector<Viewable*>& alt)
{
    vector<Viewable*>::iterator alternate;
    alternatives = alt;
    alternate = alternatives.begin();
    if ( alternate != alternatives.end() )
    {
        current = *alternate;
        current->show();
        alternate++;
    }
```

```
    for ( ; alternate != alternatives.end(); alternate++)
        (*alternate)->hide();
}
```

当客户选择一个新的可选件时，ViewAlternator隐藏以前使用的可选件，把新的设为可见。如果以前使用的可选件和新的是相同的，这个方法立刻返回，不做任何改动。

```
void ViewAlternator::setCurrent(Viewable* v)
{
    if ( current != v )
    {
        for (vector<Viewable*>::iterator alternate( alternatives.begin() ); alternate != alternatives.end();    alternate++)
        {
            if ( *alternate == v )
            {
                current->hide();
                current = *alternate;
                current->show();
                break;
            }
        }
    }
}
```

10．已知应用

在GUI框架2.0——ITG内部使用的一个用户界面框架——中有几处用到了Alternator模式。这个GUI框架提供一个ViewAlternator类，用来管理可选的Viewable对象。Alternator模式的一个更具体的实现是Selector类，它可以根据用户选择，把自己当作ScrollingList、CheckBox、OptionMenu或者MultipleChoice来显示。

JAVA至少包含了Alternator模式的两个实例。JAVA Swing库中有一个名叫JTabbedPane的类[Topley98]，这是一个可视化元件，通过对拥有一个标题和/或图标的标签（tab）的点击，用户可以在一组可视化元件之间切换。JAVA AWT库中有一个名叫CardLayout的类[CL98]，这是一个可视化元件的容器，它在一个时刻只显示一个元件，将其他的元件都隐藏起来。CardLayout提供了一组方法，用于设置当前可

见的元件。

微软基础类（Microsoft Foundation Class，MFC）库提供了名叫CpropertySheet的类[Microsoft95]，这也是一个容器，其中包含一个或多个CPropertyPage元件，这些CPropertyPage元件又分别包含其他窗口元件。通过点击与某个CPropertyPage相关联的标签，用户就可以选择这个CPropertyPage。

Motif也提供了一个名叫Notebook的窗口元件[OSF94]，它包含了多个含有其他窗口元件的页面。通过点击标签，用户也可以选择任何一个页面。

11．相关模式

Alternator模式可以用Composite模式实现。

Alternator模式与Backup模式[ST95]相似，只是Alternator模式解决的问题更具普遍意义。Backup模式关注于为指定的函数提供候选函数，它们将在主函数失败的时候自动被使用；而Alternator模式关注于一个普遍意义上的任务：为候选项提供一个放置的地方、一种选择使用的方法。

Alternator模式与Sponsor-Selector模式[Wallingford98]也有关系，这两个模式都处理对可选资源的选取行为，这样这些资源可以被动态的改变。这两个模式之间的区别在于：Alternator模式集中精力管理可选元件，使它们在对象体系中看起来就象一个元件，从而使处理这个体系的算法得到简化；Sponsor-Selector模式集中精力从一系列资源中为指定的任务选取最合适的资源。

Alternator模式和State模式[GHJV95]也有相似之处，它们的置换器对象（State模式中的State）看起来都在运行时改变了它们自己的类，这取决于它当前使用的可选件（State模式中的Context）。这两个模式之间的区别在于：在State模式中，从一个状态到另一个状态的转变有一个预先规定的顺序；而Alternator模式对转化是一无所知的，它通过选择操作决定可选件的使用和挂起。

参考文献

[CL98] Chan, Patrick and Rosanna Lee. The Java Class Libraries Second Edition, Volume 2. pp 208-220, Addison-Wesly Longman Inc. 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesly Publishing Co. 1995. 中文版：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社2000年9月。

[Microsoft95] Microsoft Foundation Class Library Reference: Part Two, Volume Four. pp 1350-1368, Microsoft Press. 1995.

[OSF94] OSF/Motif User's Guide: Revision 2.0. pp 3.28-3.29, Open Software Foundation, Inc. 1994.

[ST95] Subramanian, Satish, and Wei-Tek Tsai. Backup Pattern: Designing Redundancy in Object-Oriented Software. Pattern Languages of Program Design 2, Addison-Wesly Longman Inc. 1996.

[Topley98] Topley, Kim. Core Java Foundation Classes. pp 566-582. Prentice Hall Inc. 1998.

[Wallingford98] Wallingford, Eugene. Sponsor-Selector. Pattern Languages of Program Design 3, Addison-Wesly Longman Inc. 1998.

原文链接：http://www.umlforum.com/zippdf/alternator.zip

KCOM 技术介绍

KCOM 是 KCOM 公司独立开发的一系列开发平台和计算机语言技术的总称。
它包括：KCOM 组件模型，KCOM Basic 语言，KCOM Stage 运行环境，KCOM Space 开发平台四项技术。

KCOM 组件标准：
KCOM 组件标准是一个完善的组件标准，可以自定义属性，方法和事件，并使用事件驱动的方式编程。同时源代码方式存储有利于在网络上传输。

KCOM BASIC：
语法结构完善，易学，可以使用中文进行编程，例如使用中文变量名称，中文方法名称，属性名称等等。独有组件运算功能能够实现代码的可视化。

KCOM Stage：
KCOM Stage 是 KCOM 组件的解释器，由于采用了代码的结构化存储，在解释之前不需要对语句进行扫描和文法处理，同时采用了高效的寻址方式，所以比一般的脚本语言的解释效率要高。

KCOM Space：
能够方便的编写 KCOM 组件和代码，是一个可视化开发平台，具有丰富的界面排版，代码编写和代码调试功能。

KCOM Space 以其自定义的组件标准(KCOM 组件标准)为核心，完整地实现了一门计算机语言（KCOM Basic)、一个完整的组件化开发平台（KCOM Space）和一个高效的虚拟机（KCOM Stage)。作为国产软件在高端领域的突破，KCOM 表现出了与世界同步的技术水准。由于采用了全面组件化的思想，抛弃了传统的设计方法，全面革新了开发平台从底层的语言到上层的操作界面的设计。

KCOM 公司的产品在 2000 年中国国际软件博览会上获得创新奖。并受到倪光南院士的高度评价。

KCOM 公司合作意向：

KCOM 公司在长期对于开发平台和计算机语言技术的关注与不断开发的基础上，形成了自己独特的技术特长与优势。在如今的软件领域，软件和开发平台日益紧密成为趋势，其中突出地表现在：

软件与基于该软件的二次开发平台：如 MS Office，Notes，GIS 应用软件及开发平台，3D Max，AutoCAD，Director 等等应用软件。 一些稍小的软件如 InstallShield 也在其中使用了自己定义的解释语言，以提供强大的扩展功能。

企业流程快速建模与软件生成工具：此类工具在国外的企业应用解决方案市场中被大量使用。而国内的系统集成企业正在开始开发和形成自己的解决方案与定制工具。

嵌入式企业应用前端设备：该设备是我们所看到的一个巨大的市场机会，基本原理在于软硬件一体的显示与操作前端，采用 NC 体系，运行企业应用。

在以上几个领域中，KCOM 公司都能够以自己长期的积累和技术特长，与合作方展开广泛的技术合作，希望能够来信探讨：zhangiii@163.net

http://www.kcomspace.com.cn/

Authenticator（认证者）模式

F. Lee Brown, Jr.等 著，透明 译

抽象

想象一个为各种互不相关的分布式用户[1]充当对象仓库（repository of object）的服务器系统，它很可能需要一种限制用户访问权限的方法，而且这种方法应该是基于提出请求的用户的身份的。身份鉴定与认证，这些重要的协议却常常被现在的分布式对象系统忽略了。

Authenticator（鉴定者）模式描述了一个一般性的机制，它为服务器提供了对用户的身份鉴定与认证方法。它还有另外的特点：它可以在使用同一个程序的过程中进行协议谈判（protocol negatiation）。这个模式的工作方式是：提供一个认证谈判对象来认证用户身份，只有认证成功之后才提供被保护的对象。

1. 意图

Authenticator模式在决定分布式对象的访问权限之前对提出请求的进程进行身份认证。

2. 动机

常用的分布式对象系统都提供命名注册服务，一般情况下它允许任何访问者通过访问注册表获得任何分布式对象——只要访问者知道这个对象的名字。而在提供多种服务的分布式对象系统中，为了确定每个用户是否有访问权限，使用一个登录-认证（login-and-authenticate）协议是很必要的。

身份认证对于为用户——来自内部（Intranet）和外部（Internet）——提供多种访问权限的分布式系统来说是尤其重要的。

远端对象也需要重新配置，以解决客户机与服务器之间性能差异带来的问题。

这个模式背后隐藏着这样的能力：

- 用户对于不同的远程对象（remote object）可能拥有不同的权限。在服务器决定用户的访问权限之前，它必须认证访问者的身份以确定它的权限。
- 远程对象需要适合这两个相关的系统的性能。这些性能差异只能由一个谈判（negotiating）进程在创建对象之前进行协调。
- 需要一种一般化的认证方法，该方法不依赖于任何特定的认证方法。

3. 适用性

[1] 译者注：本文中的“client”通常是指从远端计算机连接到服务器的远程用户。当它指一个远端机器时，我把它译为“客户机”；当它指机器上的一个进程或是使用机器的人时，我把它译为“用户”。

以下情况使用Authenticator模式

1．需要对远端用户进行身份鉴别和认证

2．有几种不同的认证方法都可以使用

3．附加的协议谈判（密码选择、软件版本支持等等）需要首先获得一个远端对象，但低层分布式系统不支持这样的要求。

4．结构和参与者

Authenticator模式使用了一个可以从远端访问的分布式对象，这个对象将对请求的发起者（“请求者requestor”）进行身份鉴别和认证，并且这个对象可以进行某些协议谈判。当且仅当认证和谈判成功时，认证者才创建请求者要求的分布式对象，并使请求者可以使用它。

Authenticator模式由以下元件（component）组成：

1．*Authenticator* —— *认证者*。这个抽象类定义了一个接口用于鉴别一个连接进程或谈判进程的参量。应用程序需要定义一个具体类来实现这个接口，从而提供特定的认证或谈判协议。得到的这个类应该被实例化，并注册到一个合适的名称服务（naming service），成为一个可以在网络的任何地方访问的分布式对象。Requestor在得到一个Authenticator实例的引用后，就可以调用它的`authenticate`方法。在认证成功以后，Authenticator对象创建一个Requestor请求的分布式对象的实例，然后Requestor就可以使用`get`方法访问自己请求的对象了。关于实现本模式的一个建议是：把一个类型为*Object Factory*的引用传递给Authenticator的构造函数以创建这个被保护的对象。

2．*Object Factory* —— *对象工厂*。这个抽象类只包含一个方法，`create`。这个方法的实现需要创建被保护的对象。作为谈判过程的结果，Authenticator可以让Object Factory进行另一些特定的操作。

3．*Requestor* —— *请求者*。尽管并没有严格规定远程请求者作为这个模式的一部分，但我们通常按照一种与具体Authenticator类相匹配的方式实现它，并将它按值传递给`authenticate`方法，Authenticator用它的值来完成认证过程。

图 1 展示出 Authenticator 模式的参与者关系。

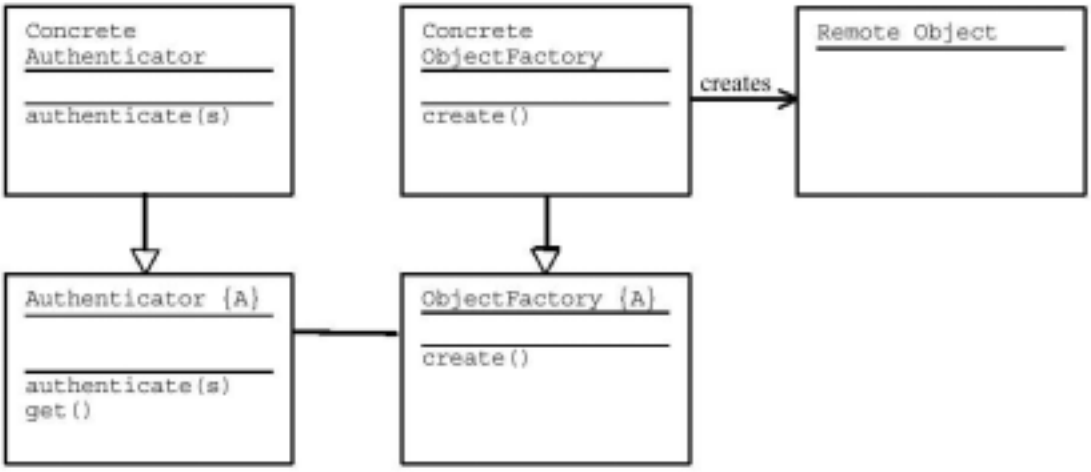


图 1 - Authenticator 模式参与者

5. 协作

Authenticator模式有四个阶段操作：

1. *初始化*。为了能让客户远程访问Authenticator对象，这个对象必须被注册到分布式对象系统的名称服务上。这阶段的操作是系统在模式外部指定的，但是与模式的耦合很紧。

2. *连接*。当一个远程对象——请求者——得到一个Authenticator对象的引用之后，它调用`authenticate`方法传递一个字符串给实际的Authenticatord。`authenticate`的返回值是另一个字符串，请求者用这个字符串判断认证（或谈判）是否成功、是否（怎样）创建另一个字符串来进行另一次`authenticate`方法的调用。请求者继续调用`authenticate`方法，直到认证（或谈判）完成。

3. *创建*。当实际Authenticator对象确认一个认证过程成功之后，它创建一个保护对象，准备响应请求者的`get`方法。一个很理想的处理流程是让`authenticate`调用一个对象工厂方法来创建保护对象。不过，为了让一个（或多个）对象可以被远程访问，你可以用任何方式使用Authenticator对象和对象工厂，

4. *获得*。最后，请求者调用`get`方法获得一个保护对象的引用。

下图是这个模式的操作图解。

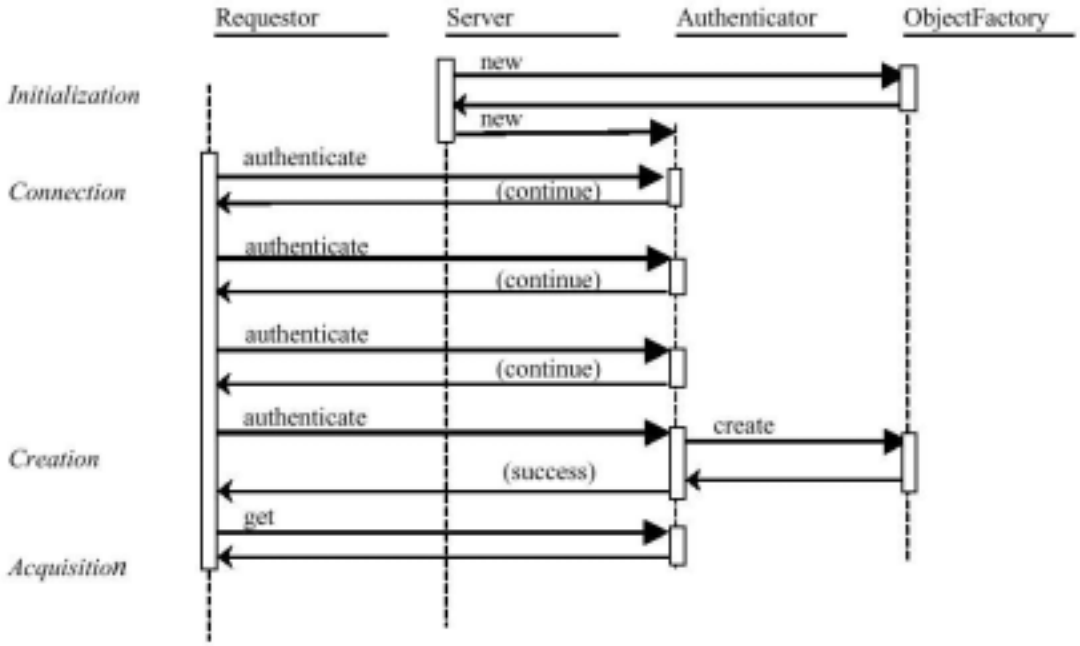


图2 - Authenticator模式顺序图解

总结：

- *初始化*。服务器创建并注册Authenticator对象，把一个对象工厂对象作为构造参数传递给Authenticator对象。

- *连接*。Requestor查询服务器的注册表，得到一个Authenticator对象的引用。

- Requestor调用`authenticate(`*string*`)`方法。

- Authenticator返回代表“完成/失败/继续认证”的字符串。

- 如果有必要，重复上面的两个步骤，直到认证/谈判完成。

- *创建*。在返回成功标志之前，`authenticate`方法调用对象工厂以创建/初始化保护对象。

- *获得*。Requestor调用`get`方法得到保护对象的引用。

6．效果

Authenticator模式提供如下益处：

1．*为需要复杂身份认证的应用程序提供一种选择*。可以在Authenticator模式中实现不同的认证方法，从而允许多种用户使用他们各自的认证方法。

2．*为需要复杂协议谈判的应用程序提供一种选择*。在服务器提供对一个对象的访问权限之前，应用程序可能需要与服务器进行一些谈判。这些谈判可以包括密码方法谈判、密钥交换、最大公约数版本谈判（greatest common denominator version negotiation）、保护对象参数化（即：传递一个保护对象的构造参数给对象工厂）及应用程序的其他任何需求。

3．*准许多个客户同时访问远程对象*。如果操作系统和硬件平台支持多CPU，这个模式可以允许授权多个客户同时访问远程对象。

4．*为身份认证提供一条一般化的、不依赖于客户或被访问对象的途径。*

Authenticator模式有以下不足之处：

1．*本模式的能力限制了分布式对象系统的逻辑能力。*一个分布式系统可能已经可以提供重复认证/协议谈判的能力。本模式会与低层系统的这些能力发生功能交迭。但是，如果我们设计的是一个新的系统，这就不成为一个问题了。

2．*实现与调试困难。*对于Authenticator模式来说，认证/谈判的编程、服务器端的调试都可能会相当困难，尤其是在有客户访问的时候。

7．实现

下面讨论了实现Authenticator模式时需要考虑的一些要点：

1．*安全性。*出于安全性的考虑，实现Authenticator模式时应该把对象工厂类隐藏起来。非可信的客户也可以远程访问Authenticator对象，但对象工厂不是一个远程对象、所以客户也不能访问它。（对象工厂不是远程对象，也意味着任何程序对它的访问必须受到限制。）在谈判没有完成之前，Authenticator绝对不能提供对对象工厂的任何调用。

2．*谈判。*设计认证/谈判过程时，必须重视多个用户的并发访问（比如说，把Authenticator设计成一个Singleton），必须重视网络掉线或超时、请求错误或顺序错乱以及其他无心或有心的错误情况。它还必须提供一种有效的方法向客户指出谈判进程的当前状态：成功？失败？或者正在处理？`authenticate`方法的返回值必须提供足够的信息以引导客户进行下一步工作。

3．*参数化对象创建。*用对象工厂进行参数化对象创建可能是有必要的。在本文后面的例子中，对象工厂的`create`方法的参数列表为空；但是只要Authenticator的实现和对象工厂允许，就可以给它一个参数列表。Authenticator对象的参数列表必须使用客户端支持的数据类型，出于这个原因，参数列表应该在谈判过程中由客户端指定，或者由Authenticator对象提供一个附加方法来让客户端指定参数。

4．*多个对象的创建。*一旦认证完成，我们下一步怎么做才是正确的？只提供一个可访问的对象？还是允许客户创建任意数目的对象？如果允许客户创建任意数目的对象，这些对象都必须是同一个类的实例吗？本文后面的例子为我们展示了“只提供一个可访问对象”的正确做法。`authenticate`方法强制只创建一个实例，所以即使多个客户调用`get`方法，它们得到的实际是同一个对象。另一种选择是：如果`authenticate`方法已经确定认证过程成功完成，`get`方法就可以在每次被调用的时候创建一个新的实例。

8．代码示例

这里展示了一个简单的JAVA实现。构成本模式的两个抽象类以JAVA接口（interface）的形式出现。

```
// Object factory interface. The create method constructs and returns
// an object as determined by the implementing class.
//
```

```
public interface ObjectFactory
{
    public Object create();
}
// Authenticator interface for a remote object. The implementation
// provides specific authentication requirements.
//
public interface Authenticator
    extends Remote
{
    // Potential response strings for the authenticate method.
    //
    public final static String AUTHENTICATED = “AOK”;
    public final static String AUTHENTICATION_FAILED = “ERR”;
    public final static String AUTHENTICATION_CONTINUE = “MORE”;
    // Accept a key string that should be the next response from
    // the client in the negotiation protocol, and return a prompt
    // string or status indication.
    //
    public String authenticate(String key)
        throws RemoteException;
    // Return the protected object. This method returns null until
    // the negotiation is successfully completed.
    //
    public Object get()
        throws RemoteException;
}
```

应用程序必须实现Authenticator和ObjectFactory接口。下面的JAVA代码向我们展示了一个框架示例，在实际的应用程序中可能用到这个框架。

```
// Object factory implementation. The create method returns a remote
// object of a predetermined type.
//
class ObjectFactoryImpl
```

```
    implements ObjectFactory
{
    public ObjectFactoryImpl()
    {
    }
    public Object create()
    {
        MyRemoteObject obj = null;
        try
        {
            obj = new MyRemoteObject();
        }
        catch (RemoteException x)
        {
            System.err.println("RemoteException error: " + x);
            return null;
        }
    return ((Object) obj);
    }
}
// Authenticator implementation. This implementation defines the
// authentication requirements and any other protocol negotiation.
// Successful completion of the negotiation immediately creates an
// instance of the protected object which can then be retrieved
// using the get method.
//
class AuthenticatorImpl
    extends UnicastRemoteObject
    implements Authenticator
{
    private ObjectFactory factory;
    private Object robject;
    public AuthenticatorImpl(ObjectFactory inFactory)
```

```
        throws RemoteException
    {
        super();
        factory = inFactory;
        robject = null;
    }
    public Object get()
        throws RemoteException
    {
        return robject;
    }
    public String authenticate(String key)
        throws RemoteException
    {
        String response = new String("");
        /* Application-specific authentication */
        if (response.equals(Authenticator.AUTHENTICATED))
        {
            robject = (Object) factory.create();
        }
        return response;
    }
}
```

Authenticator的实现将用类似下面的语句注册到JAVA RMI名称服务上（Java RMI naming service, [9]）。

```
Naming.rebind(name,new AuthenticatorImpl(new ObjectFactoryImpl()));
```

因为Authenticator接口的构造函数和get方法很可能用同样的方法实现而不管具体应用程序的情况，所以补充一个实现了Authenticator接口的DefaultAuthenticator类会是很有用的。然后应用程序可以由这个类衍生出自己需要的类。

当然，Authenticator的实现必须重视连续、并发的认证请求。上面展示的程序框架暗示着所有的远程客户都使用一个单一的对象。在真实世界的大多数应用程序中，都需要使用多线程或其它什么方法来区分远程Authenticator对象。

9. 已知应用

大多数HTTP服务器允许用户自由访问任何资源，或者根据请求者的源地址（这个地址可能是伪造的）来控制访问权限。尽管如此，许多WEB资源仍然需要一个登录密码框来认证用户的身份。一个与之对等的、甚至是更有力的认证机制——就象Authenticator所提供的机制——对于非交互式的远程资源访问来说是同样有用的。

在创建一个保护对象或者在向远程用户提供访问权限之前，可以用Authenticator模式创建一个请求者和服务器之间的对话。客户端和服务器之间的谈判内容可以包括身份鉴别（例如：用户登录）、认证（例如：简单密码、提问/应答(challenge/response)、多重提问/应答）、密码方法谈判、密钥交换、最大公约数版本谈判、保护对象参数化（即：传递参数给对象工厂作为保护对象的构造参数）、以及其它任何应用程序需要的谈判。

Cyberguard，Fort Lauderdale，FL用上面讲到的途径设计了一些内部使用的软件，并且已经考虑将它用在一些产品中。很显然，由于本模式满足了一些非常基础的需求，实际中必定有本模式的另一些实现。

10．相关模式

认证只是安全的一个方面。完整的安全性还需要一些授权机制或相似的安全机制来确定角色的权利[2]。一个类似于Bodyguard[1]的模式可以用于实现这个意图。

Authenticator模式是Abstract Factory模式的一个变化，它也是基于一个工厂类的。但与Abstract Factory对象静态指定创建对象的类型不同的是，Authenticator模式用一个交互式谈判过程来确定*是否*创建一个对象（还可能需要确定*什么*对象）。

因为分布式应用与普通应用的分离，本模式与Schmidts的模式[4-8]实际属于同一范畴。

参考文献

[1] F. Das Neves and A. Garrido, "Bodyguard", Chapter 13 in *Pattern Languages of Program Design 3*, Addison-Wesley 1998.

[2] E.B. Fernandez and J. C. Hawkins, "Determining role rights from use cases", *Procs. 2 nd ACM Workshop on Role-Based Access Contro*l, 1997, 121-125.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley, 1995. 中文版：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社2000年9月。

[4] D. C. Schmidt. “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching.” *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[5] D. C. Schmidt. “Acceptor: A Design Pattern for Passively Initializing Network Services.” *C++ Report*, vol. 7, November/December 1995.

[6] D. C. Schmidt,. “Connector: A Design Pattern for Actively Initializing Network Services.” *C++ Report*, vol. 8, January 1996.

[7] D. C. Schmidt. “A Family of Design Patterns for Application-Level Gateways.” *Theory and Practice of Object Systems*, J. Wiley & Sons, vol. 2, no. 1, December 1996.

[8] D. C. Schmidt. “Acceptor and Connector: Design Patterns for Initializing Communication Services.” *The 1st European Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, July 1997.

[9] “Java Remote Method Invocation.” Sun Microsystems, Inc., Redmond, Washington, 1977.

原文链接：http://www.umlforum.com/zippdf/Authenticator3.zip

KCOM 技术介绍

KCOM 是 KCOM 公司独立开发的一系列开发平台和计算机语言技术的总称。
它包括：KCOM 组件模型，KCOM Basic 语言，KCOM Stage 运行环境，KCOM Space 开发平台四项技术。

KCOM 组件标准：
KCOM 组件标准是一个完善的组件标准，可以自定义属性，方法和事件，并使用事件驱动的方式编程。同时源代码方式存储有利于在网络上传输。

KCOM BASIC：
语法结构完善，易学，可以使用中文进行编程，例如使用中文变量名称，中文方法名称，属性名称等等。独有组件运算功能能够实现代码的可视化。

KCOM Stage：
KCOM Stage 是 KCOM 组件的解释器，由于采用了代码的结构化存储，在解释之前不需要对语句进行扫描和文法处理，同时采用了高效的寻址方式，所以比一般的脚本语言的解释效率要高。

KCOM Space：
能够方便的编写 KCOM 组件和代码，是一个可视化开发平台，具有丰富的界面排版，代码编写和代码调试功能。

KCOM Space 以其自定义的组件标准(KCOM 组件标准)为核心，完整地实现了一门计算机语言（KCOM Basic)、一个完整的组件化开发平台（KCOM Space）和一个高效的虚拟机（KCOM Stage)。作为国产软件在高端领域的突破，KCOM 表现出了与世界同步的技术水准。由于采用了全面组件化的思想，抛弃了传统的设计方法，全面革新了开发平台从底层的语言到上层的操作界面的设计。

KCOM 公司的产品在 2000 年中国国际软件博览会上获得创新奖。并受到倪光南院士的高度评价。

KCOM 公司合作意向：

KCOM 公司在长期对于开发平台和计算机语言技术的关注与不断开发的基础上，形成了自己独特的技术特长与优势。在如今的软件领域，软件和开发平台日益紧密成为趋势，其中突出地表现在：

软件与基于该软件的二次开发平台：如 MS Office，Notes，GIS 应用软件及开发平台，3D Max，AutoCAD，Director 等等应用软件。 一些稍小的软件如 InstallShield 也在其中使用了自己定义的解释语言，以提供强大的扩展功能。

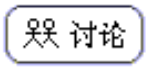
企业流程快速建模与软件生成工具：此类工具在国外的企业应用解决方案市场中被大量使用。而国内的系统集成企业正在开始开发和形成自己的解决方案与定制工具。

嵌入式企业应用前端设备：该设备是我们所看到的一个巨大的市场机会，基本原理在于软硬件一体的显示与操作前端，采用 NC 体系，运行企业应用。

在以上几个领域中，KCOM 公司都能够以自己长期的积累和技术特长，与合作方展开广泛的技术合作，希望能够来信探讨：zhangiii@163.net

http://www.kcomspace.com.cn/

用户需要什么-软件的工程可用性



（第二部分）

Larry L. Constantine 著，Huang Yin 译

以使用为中心的设计

用户和用户界面的问题并不总是发生在现在。在一开始，并不存在用户，只有操作员以及只是疯狂操作电脑的程序员，他们反复操作开关并观察着操作台上的灯，并不存在给用户的真实界面。有打孔卡片，打孔带，还有打印机。你把卡片或带打上孔并不断送至读卡机，并将打印机里打印的每一页存储起来，终端用户就获得有一行行记录的报告。其中的一些内容被格式化，还被排列成多少有点可读性的序列。

技术人员更注重的是技术而不是人，因此程序常在发觉用户的存在之前就形成用户界面。这样注意力被集中在技术问题上，如屏幕的绘制和域长，数据确认以及退出键的设计。偶尔也曾意识到用户，就是真正的人，正坐在系统另一端注视着屏幕，而且敲击着功能键。也许轻视和忽视用户的症状由来已久，专业通行“友好”的用户界面，但只有平淡无味的交互，这样往往就形成了覆盖在同样陈旧的狭隘和顽固的编程方式上的一层薄纱。

对不起，Bruce Marby ID 77623901 “August”不是一个有效的数字，你必须输入一个正确格式的日期。请再试一次。

除了那些 Bob-like 软件的热心开发者，主流已从亲用户界面的假象转移到了将用户的权力涉及到整个开发过程的每一个中心上。以用户为主（User-centered）的设计产生了，最后被命名为以用户为中心(User-centric)，使它听上去有了很大的改进。

以用户为中心的设计听上去不是个坏主意，但是，它也遗漏掉了关键的一点：软件系统仅仅是一个工具而已。由于好的工具支持工作，使工作更简单，快，简化，更多放松，或更多乐趣，所以最为重要的是建立一个围绕使用（Usage）而不是围绕用户(User)的软件。让软件和应用程序的开发者了解用户也许是好的，但是最应该了解的应当是用户要做些什么或是试图要做到什么，了解预期或是需要的用途。用户并不是总体的中心所在。设计更有用的软件的主要关键不是用户也不是用户界面，而是使用。

为什么要问“为什么”？

围绕可用性的设计是软件开发的一个相关的新的走向，它始于一个简单的问题：为什么？为什么这是必要的？为什么用户要与这个系统交互？他们想要做到什么？

为何要问“为什么”？因为只有询问了用户为什么要用这个系统之类的问题才能帮助我们关注那些有关系统可用性的基本要点。

为什么有人要用在线的电话目录。因为他们要联系那些他们不知道或是忘了联系方式的人。关键的设计要点是在不完全或是不精确的信息的基础上有效的设定一个系统入口。一个好的用户界面将以此作为焦点，减少操作步骤来找到号码。为什么用户要把银行卡插入自动取款机？是为了鉴别身份。带有磁性条纹码的卡是送到终端的一种方式，另外还有声波调用辨别或是红外线扫描也能达到这个目的。为什么在文档中用户要调用一个对话框用于输入一些特定的字符？因为一些符号或是字符在键盘上不存在。这个对话框会显示一些用户希望得到的字符，而不是在键盘上存在的字符。以可用性需求和被标识出的焦点事件来开始一个项目将可以节省开发时间。用户界面的体系结构将根据这些要点组织起来。开发人员可以避免在调整对可用性影响极小的特性上浪费时间，而关注更为重要的事件。

使用的实例

为了建立一个支持使用（usage）的系统，设计者应该了解并能表示出使用的结构。“基本用例建模”（Essential use case）是 Lucy Lockwood 和我为设计基于应用结构的用户界面而开发的一项技术。它源自 Ivar Jacobson 关于面向对象软件的著作。我们吸收了 Steve McMenamin 和 John Palmer 所开发的基本建模的概念，将 Jacobson 的用例进行扩展使之运用于用户界面的设计。“基本建模”是系统分析和设计的一种经过考验的方法，它试图将问题的主要关键点从不必要的细节以及纯技术或人为的约束中分离出来。基本用例建模技术帮助开发人员填补关于用户需要做什么以及需要系统给他们提供怎样的支持的空白。

一个基本用例是一个关于用户与系统交互的序列的抽象描述。它是从用户的观点出发，使用用户语言来描述的一种简单的叙述性对话形式的模型。每一个基本用例表述了一个使用系统的案例，它对用户是完整而且有意义的。基本用例是根据使用的核心要点进行简化并得以产生，它们还剔除了对物理性质的特殊描述和既有的用户界面的细节特征，其目标是要表示出用户所想要做到及想从系统中得到的东西的本质，即真正交互的目的。例如，在超级市场中使用回收机可以利用以下的基本用例进行描述：

用户意图	系统反应
放入可循环的原料	
	接受或拒绝
	给出价格或折扣的收据
拿着收据离开	

上面并没有提及任何关于按钮或是信息提示的内容。我们还没有规定（回收机）开口的形状和什么物质能进行回收。这个叙述形式的表格仅关注用户在交互中所感兴趣的东西，也就是说，取其精华，去其糟粕。

主要原则

如果你不懂得好的用户界面的设计，你不可能够设计出更有用的程序。让两个程序员来考虑同样的一个用户界面，将会产生一场争论。每一个人都会有自己的观点和看法。但是真正的关键是**什么在起作用**。因为缺乏客观的测试，我们不得不依赖于良好的人机对话的通用原则。Jacob Neilsen 将其归纳为 10 条清晰显著的启发式。从我们以往与开发者工作的经验中我们发现以下的基本原则是易于学习和易于应用到实际设计决策中的。其中有五项（原则）被非常夸张的称为可用性规则的指标；它们为好的用户界面提供了一个框架以及通用的对象。其它六项（原则）则涵盖了好的用户界面的特殊方面的准则。

牢记这些主要的准则并不能保证一个好的用户界面，但是在已发布的原则的基础上作出的决策将提高这种可能性。一个观念是：在一条或更多的已被验证的准则上而不是在个人观点和看法的基础上谨慎且自觉的做出用户界面设计决策。这当然不可能涵盖所有事情；甚至艺术和美学也没有被提及。一个好的用户界面常有图形的优美和可视化的要求。但换句话说，在将美学放在必要用途之前考虑是一种普遍的错误，这将导致产生一个漂亮却难以使用的用户界面。

第一条准则：可用：一个好的系统应该不需要任何帮助和指导，就能被那些具有系统所涉及领域的经验和知识却对该系统没有经验的用户使用。

第二条准则：效率：一个好的系统不会干预或中断那些对系统有丰富经验的熟练用户的高效使用。

第三条准则：渐进：一个好的系统在用户渐渐获取系统使用经验的同时，在知识，技能，易用性以及适应性上有相应的持续改进。

第四条准则：支持：一个好的系统能设法帮助用户轻松，简单，快速而且有趣味地完成任务。

第五条准则：环境：一个好的系统可以适应于实际配置的环境中的条件和情况。

结构原则：使用有意义且有用的方法，在用户所能清楚了解的简明一致的模型中，将有关联的事物合并，把无关联的事物分离，以此来有意识地组织用户界面。

简单化原则：使简单通用的任务易于执行， 用用户自己的语言直接沟通，并为长的操作过程提供好的快捷方式。

可视化原则：保持所有需要的选项和材料可视，并避免让一些外部的或是多余的信息来分散用户的注意力。和 WYSIWYG（所见即所得）相比，我们提倡 WYSIWYN：What-You-See-Is-What-You-Need.

复用原则：通过使用外部或内部的组件或行为来减少用户的再记忆或再思考，保持和目标一致性更胜于仅仅武断而得的一致。

反馈原则：使用对于用户而言简明、清晰的语言及时向用户告知：行动和解释、条件和状态的改变以及错误或例外。

容错原则：灵活和宽容。通过容许不同的输入和顺序，和合理地解释清楚合理的行为，来避免可能的错误；利用“重复”（redo）和撤销（undo）来降低错误或误用的代价。

基本用例有点象场景，也许更像它们所基于的用例。最好是通过例子来区分场景、用例和基本用例。场景是对一次非常明确的交互的具体描述。例如，一个用户拨打计算机控制的帮助热线的场景：

Ian Smith 在凌晨 4 点拨通了 TechnoTech 的支持热线，他听到了要求输入用户 ID 的提示。于是他在电话键盘上键入 17682002，此后他听到了 TechnoTech 的产品线的菜单。

不会有人建议专门为一个通宵工作的名叫 Ian Smith 的人设计一个系统，但是也很难判断如何从这样的场景中获得通用的案例，以及这样的一个具体例子当中又包含有什么不必要的细节。

用例是一种通用的场景，它描述与特定用户界面的一种类型的交互。TechnoTech 的用例为：

用户拨通支持热线并听到登录的提示。之后用户键入用户 ID 号听到菜单。

用例假定了一个特殊的用户界面，在该事件中有一个供声音输出及数字键入的电话机。在这个用例中还隐含了一个用户界面的假设。用户通过在电话按键上键入的用户 ID 号来鉴别他们的身份，选项被指定为声音菜单。

基本用例是一种用于表示抽象交互的广义的理想化的用例。在这个例子当中它可以用以下的形式表示：

调用帮助热线；鉴别自身；选择帮助。

换句话说，场景、用例、基本用例表示了抽象、理想化、一般化的继承关系。基本用例被简化到交互的最小化极点。如果我们仔细的设计一个用户界面并使其最接近基本用例。我们会很幸运的设计出最小而且最易于使用的系统。无论如何，我们要保证始终关注如何真正的满足用户的需求。

真正的应用系统包含了很多相互关联的基本用例。一个用例也许就是另一个用例的特例。例如，在远程银行应用系统中**获取存款余额（GettingSavingBalance）**就是**查询帐户（QueryingAccount）**的特例或是子集。一些基本用户案例也许作为附属进程依附于其他用户案例。例如在一个制图工具当中，**做标记（Labeling）**就作为一个**设置员工职位（InsertingStaffPosition）**的附属进程。另一个非常有用的关系就是由 Jacobson 提出的扩展（extension）。扩展是在其他用例进行期间插入，用于表示一种选择或是例外，或是变更交互的一种用例。在一个往文档中插入特殊符号的工具中，**浏览符号（BrowsingSymbols）**可能是**插入符号（InsertingSymbol）**的一个扩展。在用户看不到他们需要的符号时将用到**浏览符号（BrowsingSymbols）**。扩展，特殊化和次要性(?)使我们能够更缜密的描述应用程序的完整结构，因为我们不需要重复的写入或者拷贝同样的用例。它们还使我们明白用例之间如何关联，由此我们可以组织用户界面的架构。有时在一个项目开始阶段我们不能很好的了解特定用例之间的关系，但是我们可以认为他们有着共同的一些特性。

Role me over

做笔记，这是一次测验

可用性测试是制作更好的软件的重要部分。只有客观的测试才能最终解决关于什么可以工作什么不能工作等并不清晰的问题。精心构造的测试也能够揭露没有设计方针可以覆盖和没有专家检查能识别的可用性的问题。可用性测试有许多变体，但其主题是简单的：你使用户或是有效的替身在一些版本的软件前就坐并观察他们试用。你让他们持续去做，尝试，然后通过你观察分析所得到的就是什么会使系统更有趣，有时，更昂贵。有效的做法就是建立一个可用性测试的实验室，用成排的计算机和声视频设备装备起来，再配备上心理学家，技术人员和人机交互专家。这将使你可以向用户以及工业分析员指出可视化体系来证明你对于软件的可用性的最高优先权的承诺。

现场测试（Field testing）是可用性测试的贫穷的姊妹。它没有明亮的实验室供相片成册，所有需要的设备仅是笔记本和铅笔。从另一方面，低预算的现场测试也有一个优点，就是它着眼于人们在普通装备的工作环境下进行真实的操作时要做些什么。这并不令人惊奇，当技术人员走出他们的办公室进入到实验室时，倾向于以不同方式来思考和行动。测试并不能解决软件可用性的问题，因为测试来得太晚。正如他们在整体质量管理中提及的，你不能有测试质量的方式。在促成错误，然后发现它，之后设计补丁或是其它工作上已花费了太多。一开始就正确地设计和构造它或很早就发现错误将是更经济的。如果不是这样，就算是拥有成千上万的beta 测试员的大量的测试程序也不能发现足够的错误。通过测试得到的大量的模糊的不足将被不确定的终止，因为没有人知道如何去修正他们或是问题已经深入以及密切的嵌到在软件的体系结构当中。你可以利用可用性测试来协调好有争议的特征，或者解决关于改变方法的途径的争论，备份大块的硬数据或大胆的证明新的偏差是正确的，但是你别指望能测试出真正有用的软件的方法。

通常基本用例可以由那些帮助我们将思考的焦点由用户转向使用的抽象模型中衍生而来。用户通过不同的角色与系统交互。角色是一种用户和系统间的抽象关系。正如软件方法论学者 Rebecca Wirfs-Brock 所提出的，它是共同兴趣、行为模式、和期望的集合。

再次思考在线电话目录应用程序。许多用户仅用中等频率访问这个系统，通常是要寻找单个的电话号码。这些用户用我们称之为**偶然用户（Casual Caller）**的角色进行交互。通常他们都确切的知道他们需要什么，但是偶尔也只是有一些模糊的、大概的想法或是部分的信息。另一类用户则是大量地访问系统，经常要和在一个特定的集合或组里的一系列的人联系。例如一个特殊职能的小组，或是所有的产品经理。**公关助理**（Social Coordinator）角色可供秘书、委员会主席或是意图组织一个排球队的程序员使用。然而另外一个角色是**目录管理员（Directory Administrator）**，可以供那些有机会可能向系统加入条目、对系统进行定义或对组重新排列、更改电话号码的人使用。不同的用例用于支持不同的用户角色。例如，为了支持**偶然用户**角色我们需要开发一个用于查找一个名字或方位不确定的人的电话号码的用例。我们把这个用例称为**寻找她（GettingWhatsHerFace）**：

用户目的	系统反应
这是我知道的	
	显示找到的信息
『继续直至满意』	
	离开

“我知道的”可能包括姓的第一个字母加上部门名字，可能还是对姓的拼写的猜测或是电话号码的区号。

这个用例可以被另一个可选用例**拨号（DialingNumber）**扩展：

用户目的	系统反应
给我这个	
	拨号码

将**拨号**作为一个扩展分离出来的好处是可以把它作为其它用例的扩展。当以**目录管理员**角色创建新的条目时，拨打该号码检查是否有效，或者查找已知道确切名字的人时，这个用例都是有用的。

在上下文中

用户角色和基本用户案例帮助我们了解用户想做什么以及系统需要为用户做到什么，但是并不能告诉我们用户界面上应该有什么以及如何组织。我们可能经常直接从用户案例中获得用户界面原型，但是通常需要跨越一个很大的鸿沟。

这有助于我们在不过多顾虑到外观或者是 UI 小部件的准确形状或选择的情况下考虑为了支持特定用例而必须放置在用户界面上的部件。我们需要一个抽象模型以便更容易探索几条不同途径来使得在不必详细构造及设计的情况下就可以组织用户界面架构，这就是内容模型（content model）。（我们采用 Contextual Inquiry 中的工作环境模型的想法，Contextual Inquiry 是由 Karen Holtzblatt 和 Hugh Beyer 开发的定义需求的方法）

内容模型是一种对工具和物品，控件和容器的抽象化表示，一个用户界面需要根据不同的用户角色来向用户提供一种对一个或几个用例的支持。我们需要一些简单的形状－－长方形或椭圆形－－来表示这些抽象工具。即时贴可以很好的满足这个需要，因为它们易于移动并且有着普遍的外观来使我们关注本质而不是用户界面小部件外观或性能上的细节。

例如，为了支持基本用例**寻找她（GettingWhatsHerFace）**和**拨号（DialingNumber）**，我们可能在一开始就要定义一些我们认为用户在实施以下这些相关用例时所需要的工具和物品。

工具/控件

- **数字选择器**
- **拨号器**
- **组选择器**

物品/容器

- **名字容器**
- **找到的人信息的文件夹**

这些最终可能会象图 2 那样排列。注意我们已经开始加入一些关于行为甚至是实现符号的想法。使用上下文模型有点象被我们说的低保真模型。它不怎么象真的屏幕布局或是对话框设计，但是它需要一些基本元素来支持基本用例。

深入研究

在更复杂的设计中，当我们要实现不同的用例时，需要仔细地考虑在对话框之间、屏幕之间的用户导航。为此我们需要一张导航图来在上下文交互中链接用户导航到交互序列当中。导航图包括一些表示交互场景的标记符号，和连接他们的标记有转换状态的线条。例如，在线电话目录需要支持由用户维护的个人列表以及集中的整体列表。如果一个用户需要编辑整体数据库中某个列表中的一个域，那么授权机制将检验他的权限。这些我们可以在图 3 的导航图中看到。

基本知识

如果你想要学习可以迎合你的用户的基本需求的软件工程知识，看一看这些资料，首先从学习 Larry 的文章：**“基本建模：用于用户界面的用户案例”**（Essential Modeling: Use Cases for User Interface）（ ACM Interaction April 1995）以及**“图形导航”**（Graphical Navigation）（Windows Tech Journal ,August 1994）开始。很多关于可用性的期刊包括在 **Constantine on Peopleware**（Prentice Hall, 1995）和 Jacob Nielson 的 **Usability Engineering** (Academic Press, 1993)中。更多关于用例的书籍，可以查阅 Ivar Jacobson 的**基于对象的软件工程（Object-Oriented Software Engineering）**（Addison－Wesley，1992）和 lane Graham 的**迁移至对象技术（Migrating to Object Technology）**（Addison－Wesley，1994）。关于基本建模的经典资料依旧是 Steve McMenamin 和 John Palmer 的**基本系统分析（Essential System Analysis）**（Prentice Hall，1984）。

从用户到用户界面

所有这些抽象建模的目的就是为了获得密切接近用户所试图完成的工作的本质的用户界面架构设计。详细的模型和导航图将作为用户界面原型的粗略的向导。但是依然有很多工作要做，还有很大的空间留给有创意的图形设计和明智的软件工程。图 4 给出了明显未完工的在线电话目录应用的最初纸上原型。在这个版本中，通过微调基本用例来实现对 **FindingWhatherFace** 快速灵活的支持。随着用户的键入，系统的不断进行搜索，在下面的带有可编辑区域的行的数据表格不断挖掘。一旦根据用户填入搜索域的数据使得搜索范围有效的缩小，只要简单的双击所要的数字或者选择了数字后再点击拨号按钮就可以进行拨号。

焦点放在目标和本质上，以整个流程的使用为中心。如果不用这种方法，最终的屏幕草图或是纸上原型看上去可能是有据可依的，但结果常常是令人惊奇的难以使用。如同图 5 中的设计，使用了常用的控件和重叠的对话框，这是一种典型的学生式解决方案，但甚至也出现在一些管理个人信息的商业产品当中，直到你真正使用它时才能发现毛病。

尽管这些对于基本用例建模的概述使它看上去是象一个相当线性和严格的过程，实践中我们常常发现自己可以从柱子跳到即时贴。整个过程在图 6 中进行了描述。对于电话目录应用系统，我们始于一个单一视图的想法，把数据库分成两部分，个人列表存放在每台 PC 上，整体列表放在服务器上。这确实是一个内部应用软件的设计决策。我们可能会有一个极具灵感的用于拨号按钮的图标的想法，一个真实的用户界面原型的细节。在我们写出支持 **DirectoryAdministrator** 的基本用例之前，我们也许尝试了所有的方式来产生用于 **FindingHerFace** 的上下文。只有这时我们才开始完成导航图。

换句话说，这个策略确实是一个灵活的并行建模过程。所有的基本建模和用户界面原型对于交付使用都是极其重要的，但是他们不需要一个很严格的开发顺序。目标是跟随工作流程，以任何一种可以方便地构筑出简明而有活力的用户界面架构的方式来工作。

程序员的动力

我曾在我的演讲中提到，如果一个组织不能够使成员的工作有用处，那么他们将不可能做出更好的软件。我已经慢慢意识到这是不够的。大多数看起来是小的设计决策，影响了编程和分析，最终对系统的可用性造成巨大的影响。可用性变成每个人工作的一部分。可用性专家并不打算为你解决问题，直到你决定将可用性设计应用到软件当中为止。

原文链接：http://www.umlchina.com/zippdf/WhatUsers.zip

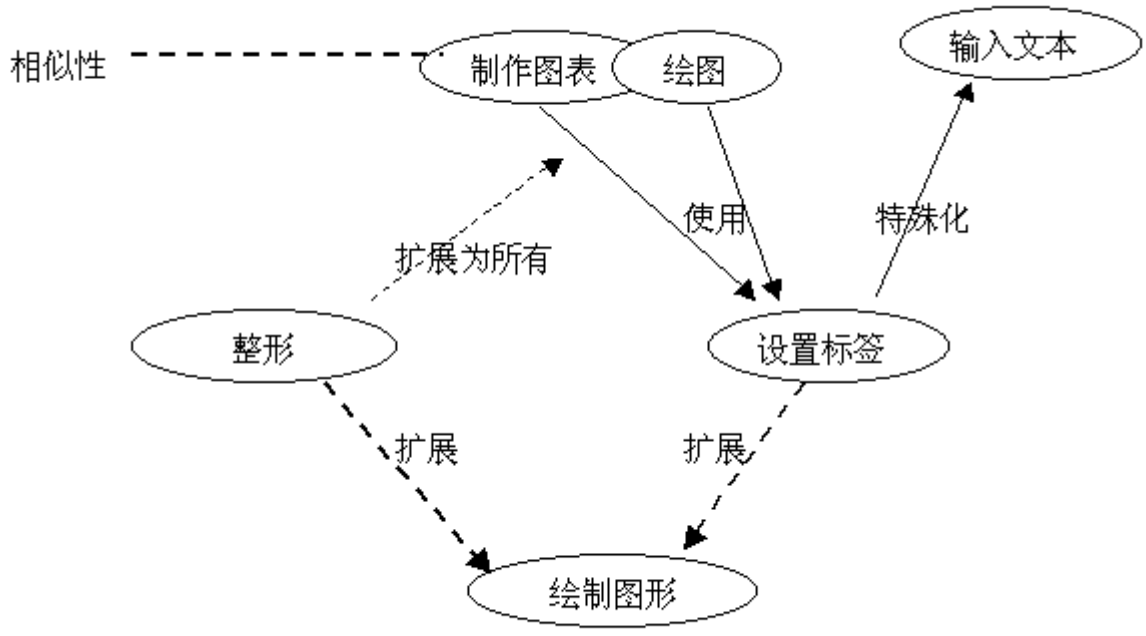


图 1：一个显示在建模当中基本用例关系的混合范例

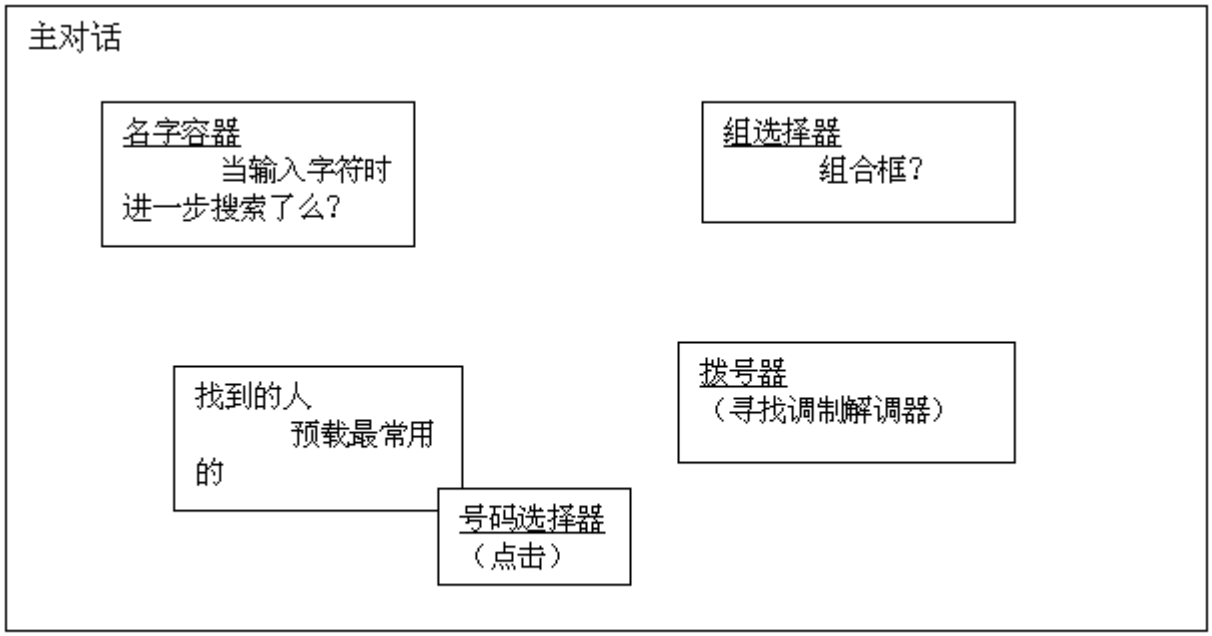


图 2：FindingHerFace 和 DialingNumber 用例的初步模型

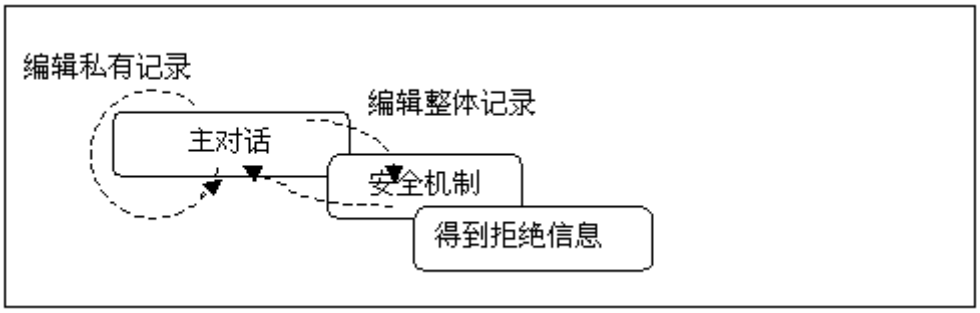


图 3：在线电话目录应用系统的部分导航图

图 4－在线电话目录应用系统的部分最初纸上原型。

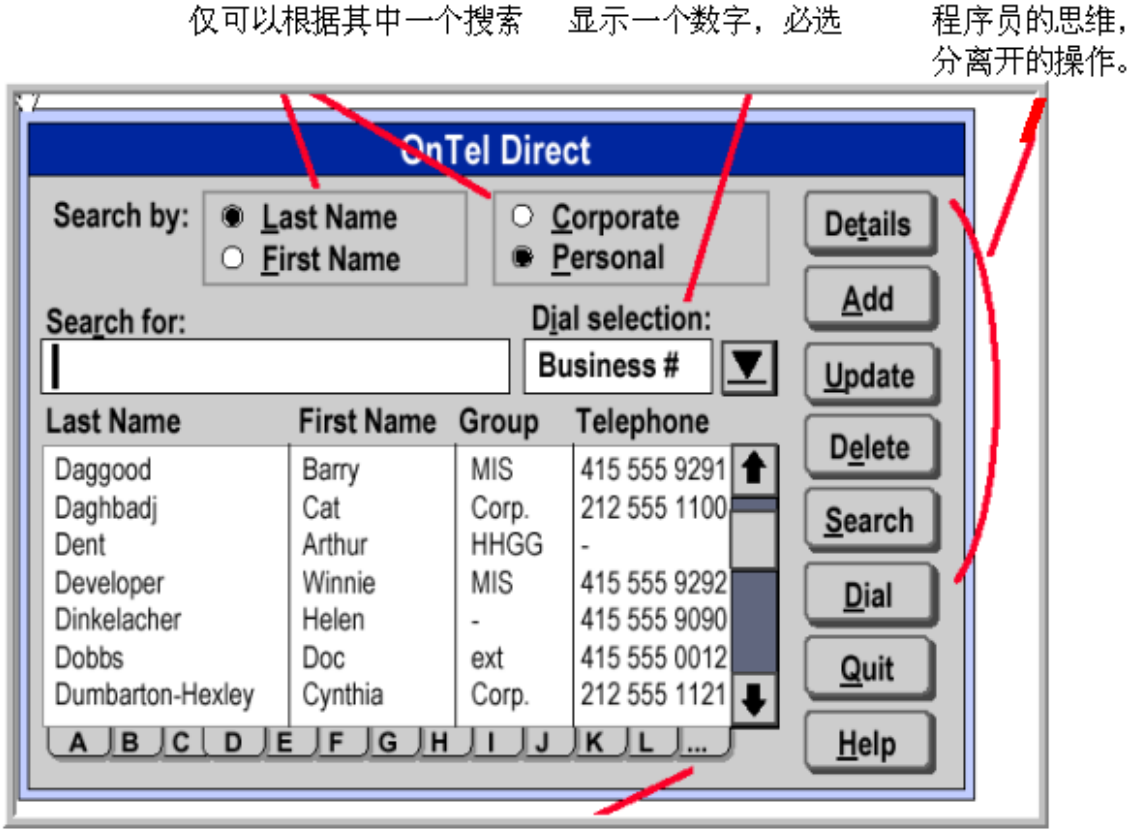


图 5－在线电话目录的传统纸上设计原型示例

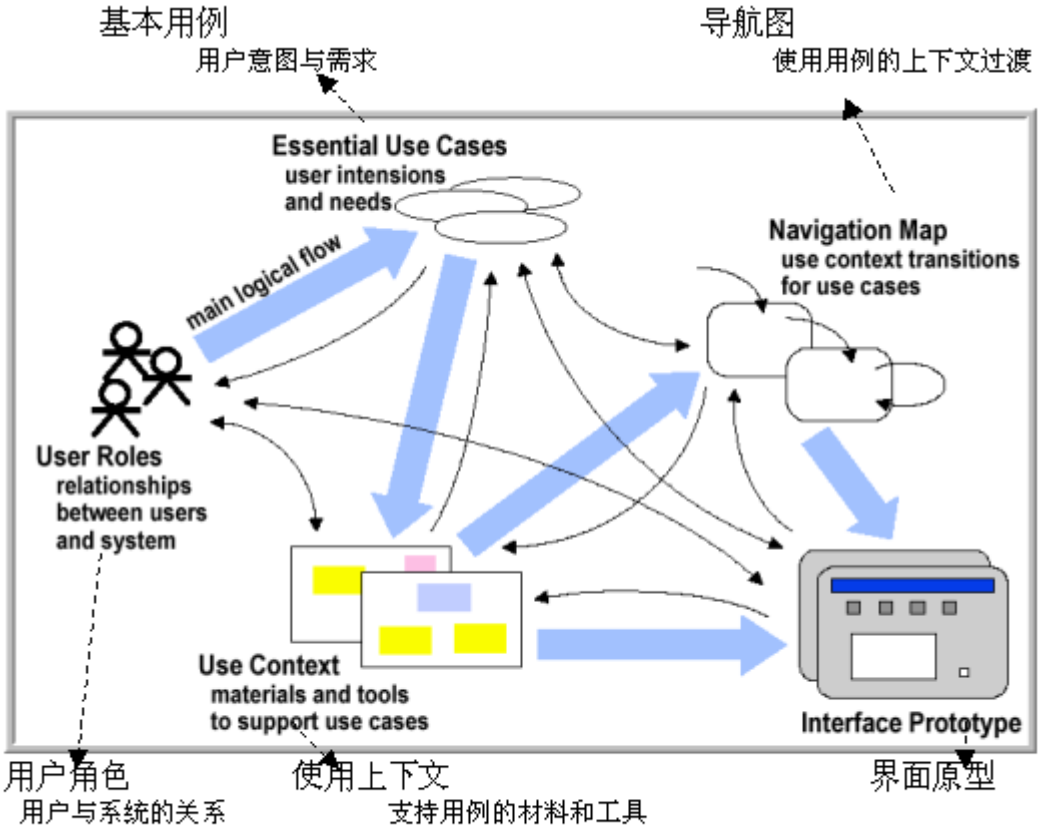
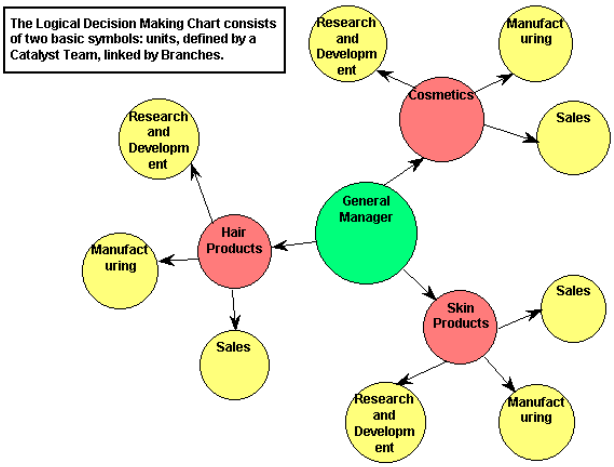
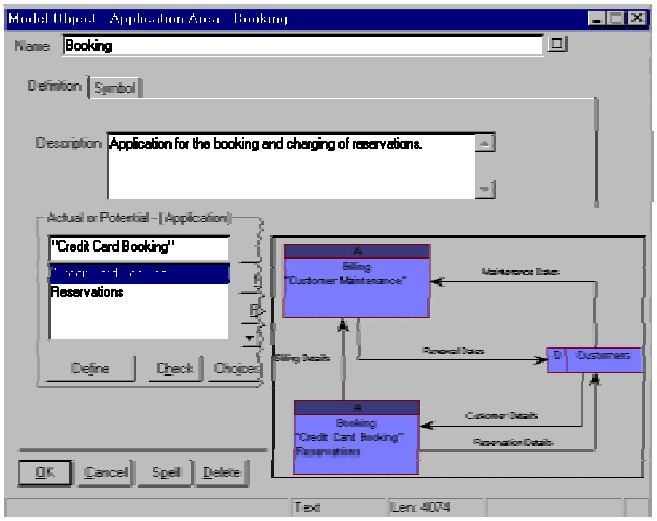


图 6－以使用为中心的并行建模过程的概观





项目管理入门

讨论

Karl E. Wiegers 著，mirnshi 译

当你预期的那一天，也许是害怕的那一天，终于来到了：从工程师的队伍里你被提拔到了软件项目领导或者团队领导的位置。这也许就是你选择的职业道路，或许你不太情愿，将就尝试一下。无论在哪种情况下，你都可能缺少工程学科、人员管理以及领导能力的相关教育。

这需要更多的领导能力和管理(它们不是一回事)，而不能象 Dilbert（译注：著名 IT 漫画主角）那样简单地和老板对抗了。当你考虑新的目标时，请考虑下面的活动计划列表。一次就抓住了每个亮点，这是不可能的。但是这份建议说明可以帮助你将注意力放在可以提高你和你的团队绩效的活动上。

建立优先级

作为经理，首先要做的、最重要的事是你需要有意识地建立优先级。当你仍陷于繁重的软件开发活动中时，你需要一套新的职责。过多的经理新手不能抗拒技术的吸引而陷于此类活动，这将导致项目组的其他人员想要获得经理的帮助时，却得不到帮助。

有成效的领导知道他们首要的任务是为其他组员提供服务。这些服务包括训练和指导、解决问题和冲突、提供资源、建立项目目标和优先级、提供适当的技术指引。要使每个组员都能清楚的知道，你总是可以帮助他们。我发现将自己定位于为被我监督的人工作是非常有意义的，而不是相反的。在你所作的事情中，对于组员要求你帮助他们这件事，应该具有非屏蔽中断的优先级。

第二重要的，是使你的客户满意。作为一名经理，没有直接的能力使客户满意，因为你已不再是作为个人提供产品和服务完成这点。相反，你必须建立一种环境，准许你的组员最大程度上满足客户的需求。经理提供了强有力的方法，有效地提高客户的满意度。

第三重要的，是为你的项目工作。因为也许还有其他许多技术上的项目，或者其他经理的请求帮助，诸如为指导委员会工作。当这些和二个高级别的发生冲突时，都要准备推辞掉。

很明显，使其他经理满意的事情是你最不重要的事情。在一个有秩序的组织里，如果你在三个以上的重大环节上获得了成功，其他的经理都会很激动的。我们并不都能很幸运地工作在一个良好的环境里，但一定要对你任务单上排在最前面的工作任务努力尽到最大的责任。集中精力有效地、快乐地、尽可能地帮助你的组员，不要将精力放在使你上司满意的上面。

分析你的技能差距

除非你已经为新位置做好了准备，否则相对于你当前的领导能力和管理技能，你会感到一些差距。出色的技术背景或许是你被选为领导角色的一个因素，但是你要想干得出色，你需要更多的技能。针对别人的评论和项目，真实地列出你的长处和短处，然后减少差距。

软件人员并不以令人满意的人际关系技能出名。你会希望增强处理人际关系的经验：解决冲突、说服以及灌输想法。你也不得不处理包括招聘、解雇、商谈计划表，以及在你的办公室里评论某人业绩使其伤心落泪等一些事务。

我发现从一堂倾听技能课开始我的管理职业是非常好的。当作为个体提议人，积极地将我们自己的技术议程提交小组时，我们经常对此感到非常惬意。有效的管理要求更多的合作和善于接受的人际关系方式。要花点时间学习如何(何时)巧妙地引导自己的自然判断。倾听技能课提供了一种交流机制，我已经发现在许多场合下都很有用。

接着，到讲台的另一侧，提高你的演讲能力。如果你真的不适应公开场合的讲话，学习戴尔.卡内基的课会有帮助的。你会发觉，通过这样的培训获得的经验，以及获得提高的交流能力，都可以帮助你更好地适应将来的工作。

作为项目领导，为了计划和跟踪项目，以及当需要项目回退而采取修正措施时，你有责任调整其他人的工作。参加项目管理的培训课，阅读一些有关项目和风险管理的书籍和文章。参加项目管理学会，阅读其月刊--PM Network。SEI 的软件能力成熟度模型对于软件项目计划和项目跟踪提供了很多有用的建议。建立优先级的能力、控制有效果的会议、清晰的交流，对于你，作为一名经理的绩效将会有实质上的影响。

定义“质量”

几乎每个人都会认真地对待质量问题而且都希望生产出高质量的产品。然而，对于软件的质量含义，没有一个统一的定义。传统上的软件质量观点和“足够好”的软件观点有着激烈的争论。为了帮助小组走向成功，需要花一些时间和你的组员、客户共同探讨质量的含义。

这两种阵营在思想上经常不会有相同的定义，可以很容易的就不同目的开展工作。关注交付计划的经理对于想正常地检查每行代码的工程师会不耐烦的；认为可靠性非常重要的客户对一个带有很少使用但带有很多 bugs 的特性的产品是不会满意的；一个很好的 GUI 也许会让用户厌烦，因为用户已经熟记了如何有效地使用前一个版本的产品。

为了更好的理解客户对软件质量的看法，在 Kodak，我的小组曾经邀请了我们的客户和他们的经理就这个议题在一个开放的论坛展开讨论。这个论坛是很有意义的，那些使用我们产品的人有着自己的理解，通过讨论，我们可以知道我们制定质量的思路有哪些和他们是不相符的。明白了不同，就可以使你集中精

力，照顾客户的最大利益，而不是使开发人员获得最大满意。

软件质量的传统描述包括要与说明书一致，满足客户的需求，代码和文档没有缺陷。“六个Σ质量”(six-sigma quality)这个流行词， 建立了一个非常高的尺度，用于监测失败的频率和密度。但它不适用于如快速产品交付，可用性，充足的特性集，已支付价钱的交付意义这样的质量尺度，。对于我们生产和购买的产品，我们总是热衷于尽可能涵盖所有的这些质量特性，然而，妥协总是必须的。

在一个项目的需求阶段，我们制定了包括十项质量属性的一个列表，如效率，协同性，正确性以及宜于学习，我们认为这对于用户来说是最重要的。我们请客户关键人物代表小组以 1 到 5 的尺度评估每项属性。一旦我们决定了哪些属性是最重要的，我们就可以设计并实现这些目标。如果你在了解了对于客户的质量含义并在设计实现质量属性的过程中没有麻烦的话，而且客户对质量属性表示满意，那你是很幸运的。

在众多关注的质量说明中，我曾听到过一个：“客户回来了，但产品没有” 。和你的客户、开发人员一起对每一个产品都确定适当的质量目标。一旦决定了，就给出达到质量目标的明确的最高优先级。以身作则，按很高的质量标准要求你自己的工作。采用这个座右铭：“力求尽善尽美，满足于优秀。”

表彰成绩

对你组员成绩的表彰和奖励，是激励他们的一种很重要的手段。除非你的小组中已经有了一种表彰程序，否则这应是你最重要的事情之一。表彰包括象征性的东西(证书，旅游奖励)以及实际的东西(电影票，餐馆礼品券，兑现奖)。在送赠品时要说一些亲切的话语：“感谢你所给予的帮助”或者“祝贺取得了成绩”。在表彰和奖励上花费很少的心思和钱，就可以获得很多的友好和将来的合作。包括客户代表，以及为项目成功做出过贡献的支持人员等等开发组外的人员也可以获得表彰。

和你的组员讨论，了解他们感兴趣的表彰和奖励的方式。使得无论大小成就的表彰活动成为小组文化的一个标准组成部分。对每位组员对其所作的工作表现出发自内心的兴趣也要给与含蓄的表扬，为消除所有影响他们战斗力的障碍尽你的力量。表彰是展示组员以及小组外的其他人的一种方式－－你要知道并感谢他们为小组成功所作的贡献。

学习过去

你的小组在过去承担的一些项目有可能没有取得完全的成功。甚至在成功的项目上，我们也能经常认为一些事情我们下次会作得更好。当你进入了新的领导角色，需要花点时间了解早期的项目为什么失败，并要计划避免犯同样的错误。对于软件开发，每位经理花时间处理每种可能要发生的错误是非常困难的，学习过去的成功和失败就是个成功的开始。

可以从过去你们小组承担的一个没有经过检查评估的项目着手，不要管其成功还是失败，实施项目后

的回顾(有时称作事后调查分析)。你的目标不是判定责任，而是为了在将来项目中作得更好。借此，可以了解什么已经作得很好，什么应该作得更好。在当前每个项目的主要里程碑时，通过集体讨论或公平的组织者，用同样的方式，领导小组用头脑风暴的方式对其展开分析。

另外，要了解领悟已有的软件工业的最佳准则。一个好的起点是 Steve McConnell 的 Jolt Award 获奖作品：快速开发(*Rapid Development*，Microsoft Press, 1996)的第三部分 ，叙述了 27 个最佳准则。也要避免 McConnell 叙述的 36 个常见的软件开发错误。你的组员也许反对新的工作方式，但是你的角色是作为一名领导，要确保团队一致连续地使用最佳可用的方法、过程和工具。积极促进组员之间的信息共享，这样局部单个最好的实践经验就能成为每个开发人员的工具箱的一部分。

建立改进目标

一旦你对过去的项目建立起了回顾，确立了质量对小组的意义，你就要建立短期以及长期改进的一些目标。目标要尽可能量化，所以你要划分几个简单的阶段，标明你是否采取了适当的过程朝着目标前进。

例如，如果你认定由于需求的不稳定导致项目经常延期，你可以建立一个改进需求稳定的目标，在 6 个月内提高 50%。这样一个目标需要你确切知道每周或每月需求的变化数，清楚他们的出处，采取行动控制那些变更。这可能要求你要改变与那些提交需求改变的人的交流方式。

你的目标和阶段是软件过程改进程序的组成部分，你要使之有序。作为缺乏创造力的官僚主义的最后避难所，轻视“过程”很流行。虽然事实上，每个小组都能找到改进其工作的方式。当然，如果你总是用已有的工作方式工作，你也就不要期望你会得到比以前更好的结果。

有两个强烈的原因要求改进过程：校正问题，防止问题。确保你的改进努力要围绕着已知的或可预知的可能威胁项目成功的问题。领导你的小组找出当前正在使用的方法的长处和短处，以及项目面临的风险。

我的小组召开了一次“两段式头脑风暴”练习，来确定改进软件生产力和质量过程的绊脚石。在第一次会议中，参会者在便条上写出他们关于会议主题的想法，一个便条一个想法。组织者将他们写在便条上的想法收集上来并分组。最后，我们就会得到一打主要的分类，并将其记录到活动挂图上。

第二次会议，相同的参会者在便笺上写出解决这些障碍的思路，并贴在挂图的合适位置。进一步细化，归纳出一些详细的活动，就可以成为我们努力的一部分，清除障碍，帮助组员实现软件的质量和生产力的目标。

建立可度量和可达到的目标，便于你集中精力实现改进。要使目标具有明显的优先级，并可周期性地监视过程。记住你的目的是，提高你的项目和公司完成的技术和业务上成功，不要满足于一些过程改进书籍里提到的期望细节。要把改进的工作视为迷你项目，具有可分发、资源、计划和有责任的小项目。否则，过程改进活动将总处于比诱人的技术工作低的优先级上。

缓慢的开始

这篇文章提供了许多建议，帮助你，一位软件经理新人，带领你的小组走向伟大的成功。在日复一日新的工作压力面前，要努力保持你的头脑清醒。在长时间的塑造软件开发小组的文化和习惯上，你还是个非常重要的角色。你不必一次性都作完，可以选择跟环境最相关的的几个开始。

作为软件经理，除了项目要按时按照预算完成外，你要担负的责任还很多。你还要：领导技术人员，将他们形成一个具有凝聚力的团队；建立协同团队工作的环境；鼓励和奖赏高级软件工程师的实践应用；平衡来自客户、公司，组员和你自己的需求。

这是项重大的任务，祝你好运。

原文链接：http://www.umlforum.com/zippdf/48prjmandocs.zip

项目管理规范-RUP 管理实施

（第二部分）

讨论

李杰(poorjim@sina.com)

第一部分：项目阶段
第二部分：核心工作流程
第三部分：角色划分
第四部分：目前实施项目规范的考虑

概述

软件开发的产品质量水平，是一个由来已久的话题。而提高软件企业的产品质量水平，必须改进软件产品的开发过程。但是这里没有什么百试百灵的灵丹妙药，我们必须根据本企业的实际情况，参考国内外先进企业的经验，总结出一种适合本企业的软件开发模式。

此规范是基于 CMM 模型规范，以 RUP 软件工程过程为蓝本，由我本人根据项目实际情况而选择修改，从而使之适应当前应用级系统设计开发的需要。

本文主要以 RUP 的软件工程框架为主，省略复杂概念部分。着眼点放在控制软件产品开发流程上，由于人员配置与软件分工现行状况的限制，对其中的部分细节进行了合并可省略，从而适应目前国内软件开发所要求。

Rational Unified Process（简称 RUP）是一套软件工程过程（在下面介绍）。

在 RUP 过程中，我们可以看到它非常强调一点：**循环**。

现在我们做的每一个项目都存在不断变化的问题。用户需求变化、系统设计变化（可能是需求变化也可能是存在了技术问题）、编码变化（由测试与复审等环节引发的）等问题困扰着项目进行。解决这些问题的方法就是不断的循环。

这个规范是我根据自己的观点整理编写而成的，有不足之处请指教。

李杰

2001 年 3 月 14 日

2. 核心工作流程

软件工程中的工作流程分为两部分：核心工作流程与核心支持工作流程

核心工作流程（在项目中的流程）

- 业务需求建模
- 分析设计
- 实施
- 测试
- 部署

核心支持工作流程（在组织中的流程）

- 环境
- 项目管理
- 配置与变更管理

2.1. 业务需求建模

2.1.1. 目的

业务建模的目的在于：

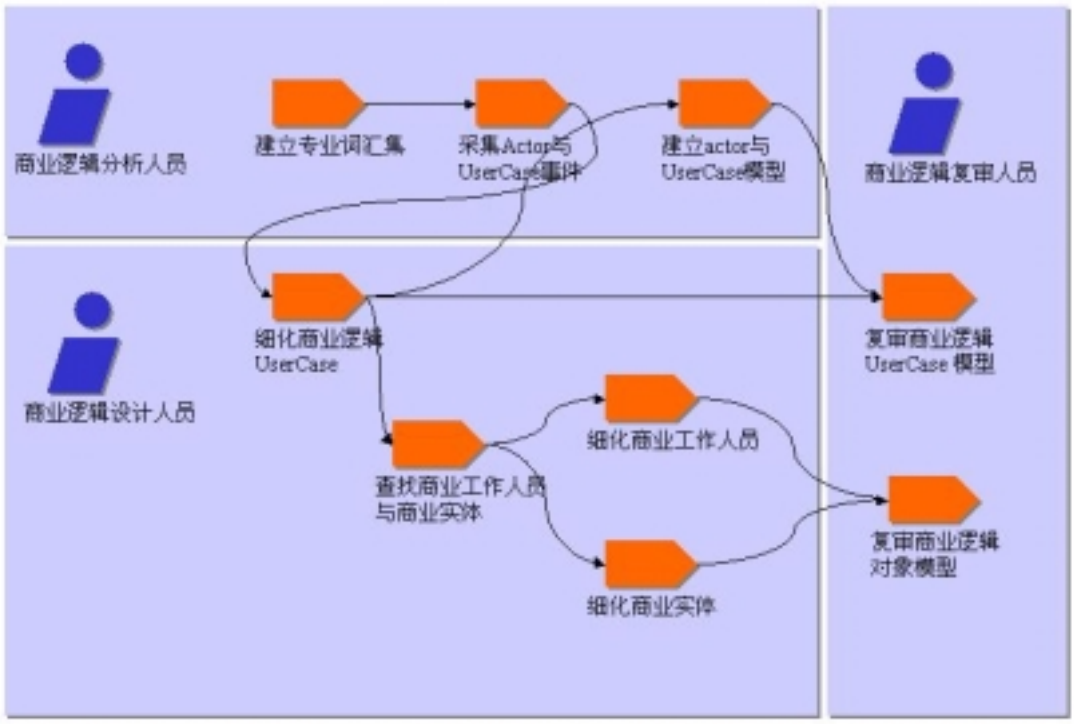
- 了解目标组织（将要在其中部署系统的组织）的结构及机制。
- 了解目标组织中当前存在的问题并确定改进的可能性。
- 确保客户、最终用户和开发人员就目标组织达成共识。
- 导出支持目标组织所需的系统需求。

为实现这些目标，业务建模工作流程说明了如何拟定新目标组织的前景，并基于该前景来确定该组织在业务用例模型和业务对象模型中的流程、角色以及职责。

作为对这些模型的补充，还编写了以下文档：

- 补充业务规约
- 词汇表

2.1.2. 业务建模工作流程



2.1.3. 提供的文档与模型

- 商业逻辑建模（USE CASE）（ROSE）
- 业务需求说明书（MS WORD）
- 专业词汇表（英汉对照）（MS WORD）
- 风险说明（MS WORD）
- 复审说明书

2.1.4. 文档模板

参见项目管理规范目录下业务需求文档模板子目录

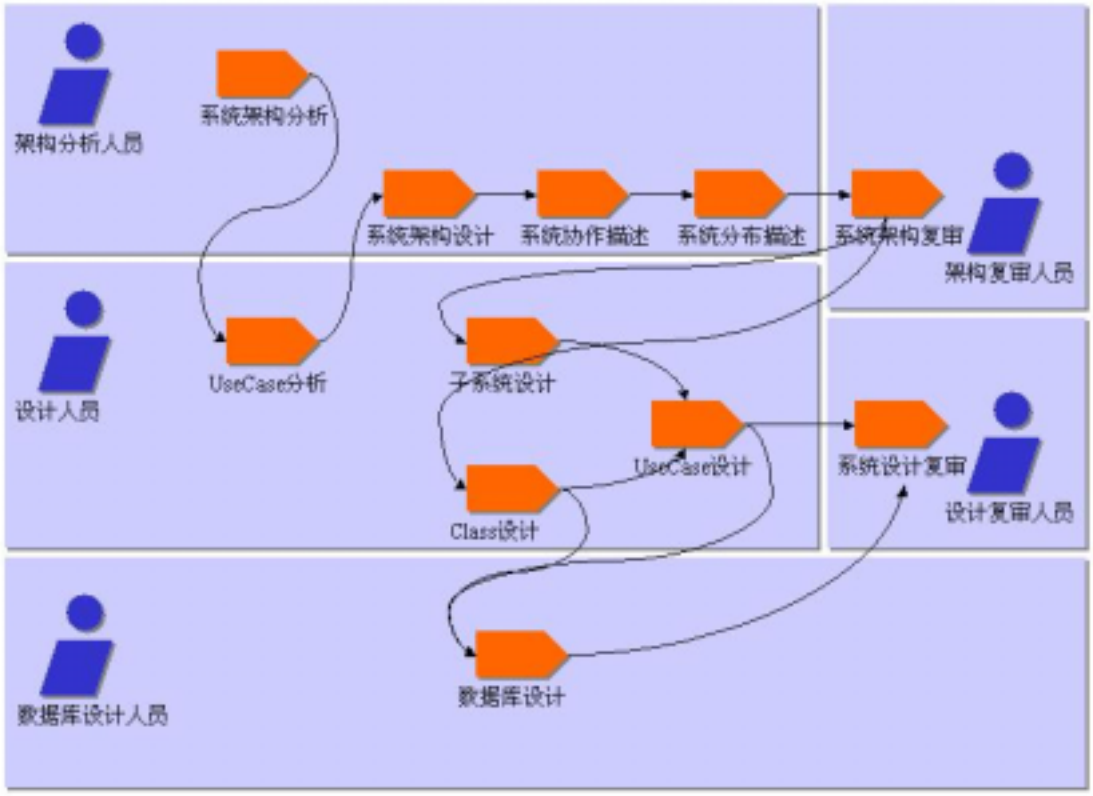
2.2. 分析设计

2.2.1. 目的

分析设计的目的在于：

- 将业务需求转换为未来系统的设计。
- 逐步开发强壮的系统构架。
- 使设计适合于实施环境，为提高性能而进行设计。

2.2.2. 分析设计工作流程



2.2.3. 提供的文档与模型

- 系统总体设计报告（MS WORD）
- 系统设计模型 DOMAIN MODEL（ROSE）
- 系统设计模型 DESIGN MODEL （ROSE）
- 数据库设计模型 （POWER DESIGNER）
- 数据字典（MS WORD）
- 系统详细设计报告（MS WORD）
- 工作量化书（MS WORD）

2.2.4. 文档模板

参见项目管理规范目录下分析设计文档模板子目录

2.3. 实施

2.3.1. 目的

实施的目的包括：

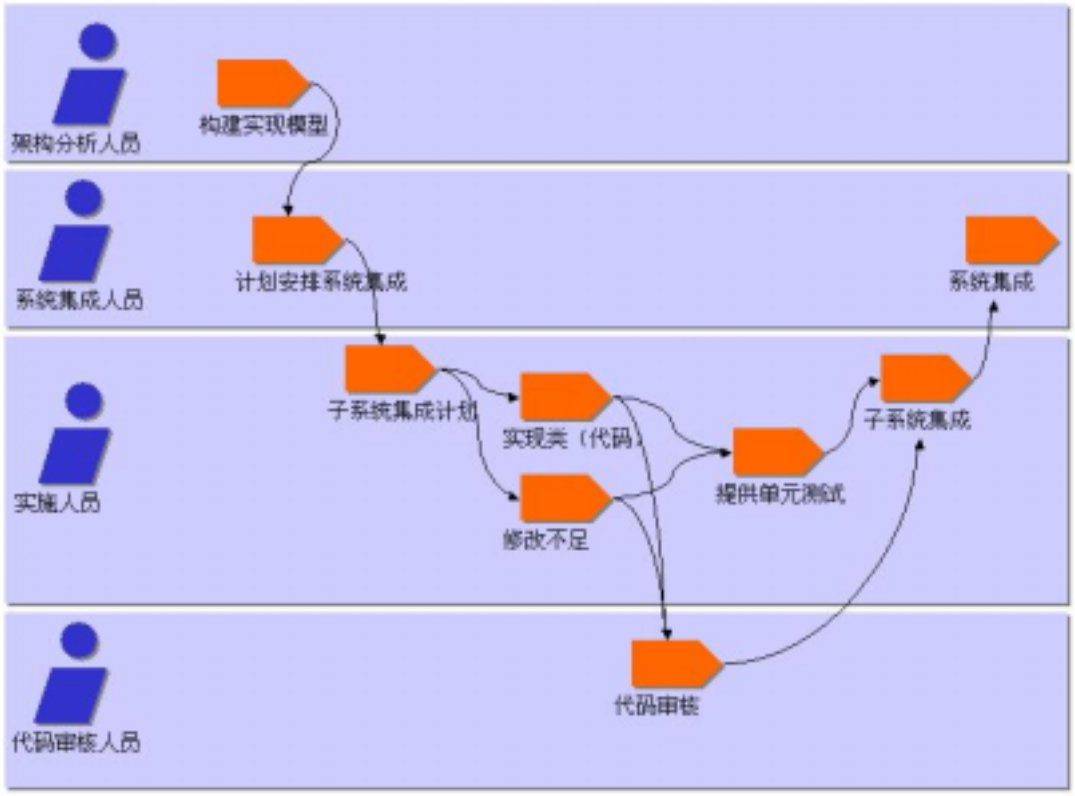
- 对照实施子系统的分层结构定义代码结构、
- 以构件（源文件、二进制文件、可执行文件以及其他文件等）的方式实施类和对象、
- 对已开发的构件按单元来测试，并且
- 将各实施员（或团队）完成的结果集成到可执行系统中。

实施工作流程的范围仅限于如何对各个类进行单元测试。系统测试和集成测试将在测试工作流程中进行说明。

测试的目的在于：

- 核实对象之间的交互。
- 核实软件的所有构件是否正确集成。
- 核实所有需求是否已经正确实施。
- 确定缺陷并确保在部署软件之前将缺陷解决。

2.3.2. 实施工作流程



2.3.3. 提供的文档与模型

- 实施总结书（MS WORD）
- 实施模型（ROSE）

- 系统集成书（MS WORD）
- 代码审核意见书（MS WORD）
- 源代码（MS WORD）
- 用户使用手册（MS WORD）
- 错误解决记录手册（MS WORD）
- 构件及其说明

2.3.4. 文档模板

参见项目管理规范目录下实施文档模板子目录

2.4. 项目管理

2.4.1. 目的

本部分的目标是，通过提供一些项目管理的环境，使这个任务更加容易完成。它虽然不是成功的秘诀，但它介绍了可以显著提高成功交付软件可能性的项目管理方法。

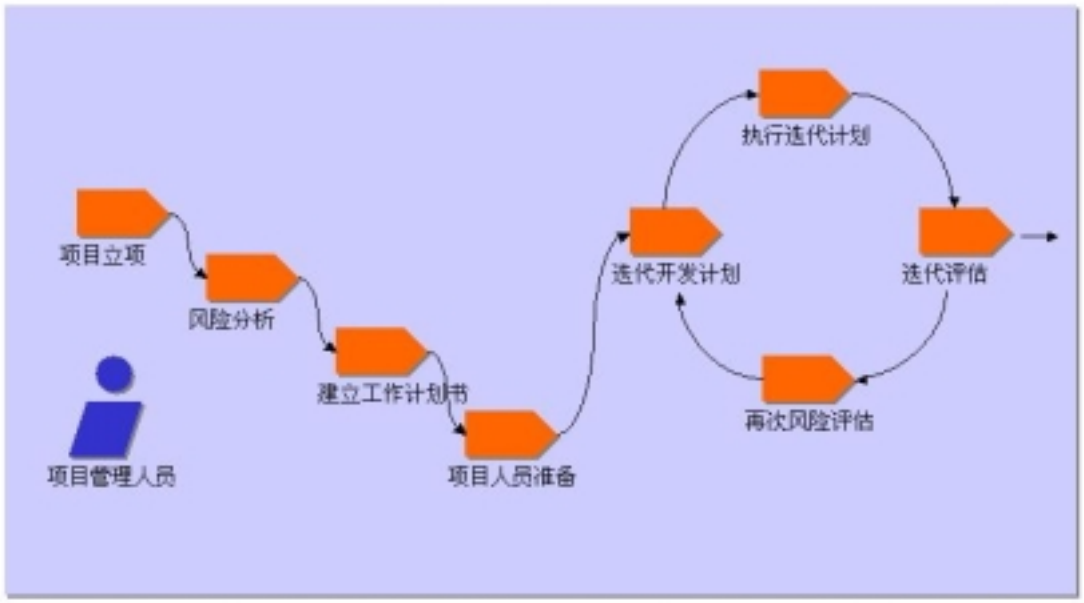
项目管理的目的是：

- 为对软件密集型项目进行管理提供框架。
- 为项目的计划、人员配备、执行和监测提供实用的准则。
- 为管理风险提供框架。

该工作流程主要侧重于迭代式开发流程的以下重要方面：

- 风险管理
- 计划迭代式项目，贯穿生命周期并针对特定的迭代
- 监测迭代式项目的进度、指标

2.4.2. 项目管理工作流程



2.4.3. 提供的文档和模板

- 风险管理计划（MS EXCEL）
- 工作计划书（MS EXCEL）
- 风险列表（MS EXCEL）
- 迭代计划（MS EXCEL）
- 问题解决计划（MS EXCEL）
- 测试计划书（MS EXCEL）
- 系统集成计划书（MS EXCEL）
- 子系统集成计划书（MS EXCEL）
- 工作单（MS EXCEL）
- 产品验收计划（MS EXCEL）

- 评测计划（MS EXCEL）
- 项目计划复审意见书（MS WORD）
- 开发总结（MS WORD）

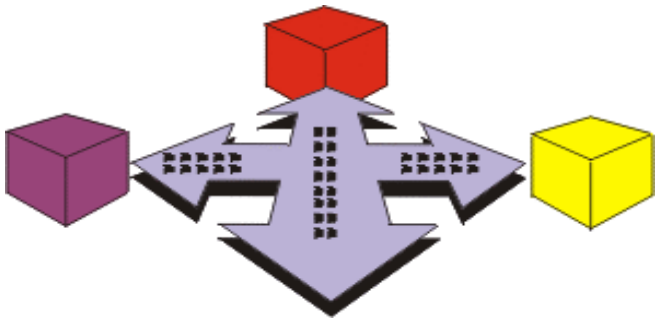
2.4.4. 文档模板

参见项目管理规范目录下项目管理文档模板子目录

2.5. 部署

2.5.1. 目的

部署工作流程用来描述那些为确保最终用户可以正常使用软件产品而进行的活动。



部署工作流程描述了两种产品部署的模式：

- 自定义安装
- 通过 Internet 使用软件

在每个实例中，都强调要在开发场所对产品进行测试，并在产品最终发布之前进行 Beta 测试。

尽管部署活动主要集中于产品化阶段，但在较早的一些阶段中也会有一些为部署进行计划和准备的活动。

2.5.2. 提供的文档和模板

- 部署计划
- 安装文档
- 发布说明