分析模式：可重用对象模型（笔记版）

作者：Martin Fowler
ADDISON-WESLEY 1997

整理：Windy J（2001/2/14）

注：本文采用意译的方法，重在表达原文的意思，而不是逐字逐句的翻译，如有错误，请指正。

目录

前言

不久以前市面上还没有关于面向对象分析和设计的书籍，但现在这类的书非常多以至于人们无法全部一一掌握他们。大部分的这些书致力于讲解一种注记方法，介绍一个简单的建模过程，并提供一些简例来说明。但*分析模式：可重用对象模型*显然不同，它关注的主要是建模的结果——那些模型本身，而不是关注建模过程——怎样建模。

我是一个信息系统面向对象建模咨询/顾问人员，为客户提供建模人员培训和项目指导。我的绝大部分经验来自建模技术知识和怎样运用它们。不过，更重要的是我在实际建模中的经验，以及看到问题经常会重复出现，正是因为这样我可以重用以前建立的模型，改进它们，并用于新的要求。

最近几年越来越多的人们发现了这一现象。我们意识到典型的方法学的书，尽管很有价值，却只提出了学习过程的第一步，这样的学习过程，还必须捕捉实际的事物本身。这样的意识发展成为了“模式（*Patterns*）”运动，但是怎样给模式下一个唯一的定义呢？我

的定义是：模式是一种思想，它已经适用于一个实践环境中，并有可能适用于其他的环境。（A pattern is an idea that has been useful in one practical context and will probably be useful in others.）模式可以有多种格式，每一种格式都增加了一些有用的特别化特征。

本书讲述分析中的模式，也就是反映商业过程的概念模式，而不是具体的软件实现。大多数的章节讨论不同的问题域中的模式，这些模式难以按传统的行业分类（例如制造、金融、医疗保健等），因为他们经常适用于多个领域。模式可以帮助我们理解人们对世界的认识，而且基于这样的认识来建立计算机系统，并试图改变这些认识（或者可以称为商业过程重组工程 Business Process Reengineering——BPR）是非常有意义的。

当然，概念模式（Conceptual Patterns）并不能孤立存在，对于软件工程人员来说，只有当他们看到如何实现时概念模型才有意义。所以在这本书里我还提供了可以将概念模型实现成软件的模式，并将讨论该软件如何适合一个大型信息系统的结构，还将给出和这些模式有关的具体实现技巧。

建模人员将在这本书里找到在其他新的领域里有用的思想，这些模式包括有用的模型，设计背后的理由，还有什么时候适合和不适合应用。这些可以帮助建模人员在遇到具体的问题时更好地应用这些模式。

这本书里的模式还可以用来回顾已有的模型——来看其中哪些可以省略，哪些可以找到替代的方式来改进它们。当我回顾一个项目时，经常拿它们和从以前的项目中学到的模式相比较，就这样，我发现模式意识使我更容易应用以往的实践经验，这样的模式也远远比简单的课本更容易揭露模型的要点和本质。通过讨论我们为什么这样建模，将会使我们对如何改进这些模型有更深的理解，即使我们并没有直接运用这些模式。

本书结构

这本书分为两大部分：第一个部分讲述分析模式——来自概念商业模型的模式；他们提供来自贸易、测量、记帐（Accounting）、组织关系等多个问题域的关键抽象。这些模式之所以是概念性的因为他们关注的是人们对业务的思考和认识，而不是计算机系统的设计方法。这个部分的章节着重于可用的可选模式，和这些可选模式各自的优点和弱点。而且尽管每一个模式可用于特定的问题域，那些基础的模式还可以用于其他的领域。

第二个部分讲述支持模式，通过支持模式对这些分析模式提供使用帮助。支持模式描

述分析模式怎样适合一个大型信息系统的结构，描述这些概念模型如何转换成软件接口和实现，还有那些特定的高级（advanced）模型构造怎样和更简单结构相关。

为了描述这些模型，我需要一种注记方法。附录中包括对本书所用注记方法的简要讨论，以及符号的意思，还包括哪里可以找到我所用（注记）技术的指南。

每一个部分都划分为章节，每个关于分析模式的章节包括那些在一个松散的主题空间相关的模式，这些模式也会受到产生它们的项目的影响。这样的组织方式说明任何模式都来自于实际的环境。每个模式出现在一章的各个小节，不象其他模式作者一样，我并没有为每个模式提供单独的标题，而是提供了一个类似于描述原始工程的格式，并增加了模式在原始问题域中以及在其他的环境下如何应用的例子。关于模式，其中最大的困难就是如何抽象到别的问题域；不过我认为这个问题最好留给读者自己去思考。

因此，这本书可以当作一个目录，而不是需要从头到尾读完。我努力使得每个章节独立于其他的章节，（虽然这经常是不可能的，所以每个章节如果引用了其他的章节，我会在介绍里声明。）每个章节的介绍部分包括该章节的通用主题空间，总结这个章节中的模式，以及模式产生的项目资料。

怎样阅读本书

我建议先详细阅读第一章，再看每章的介绍部分，然后就可以按照你的兴趣选择阅读各章，不管你用什么顺序。如果你不熟悉我建模的方法，或我采用的注记和概念，可以阅读附录。在模式表格中给出了每个模式的简要总结，以后回顾本书的时候可以查阅它们。非常重要的是这些模式在它们产生的问题域之外也非常有用，所以我建议你阅读那些也许不在你兴趣之内的章节，例如，我发现医疗保健行业中关于观测和测量的模式在公司金融分析中证明非常有用。

谁将阅读这本书

最大的读者群应该是面向对象计算机系统的分析和设计人员；

一个小规模但非常重要的读者群应该是那些建模项目的问题域专家；

希望程序员们可以钻研这本书，尽管本书缺乏代码和有着概念倾向，我建议你们着重注意第十四章，这一章讲述了概念模型和结果软件之间的关系；

数据建模人员；

经理们将发现这本书是开发活动的一个起点。从模式开始有助于简化目标，项目计划也可以获益于广大的模式设计背景；

学生并不是我定位的读者，但他们同样可以阅读这本书。

一本动态的书

书出版之后作者将难以改变书的内容，但是我一直在努力学习，而且这样的学习一定会改变我的原有的看法，所以我希望这些改变也能让读者们知道。

幸运的是 Addison-Wesley 公司为这本书提供了一个网址 <http://www.aw.com/cp/fowler.html>，将用来提供更多的资料。使得这本书可以保持动态改变，我希望该网址会包括以下内容：

- 我学到的关于这本书的模式的新的东西；
- 对于这本书的问题解答；
- 其他人关于模式的有用解说；
- UML 注记出现的时候我会重画书中的图表并上载它们；

这个网址将是本书的补充部分，所以别忘了关注它，并可用来告诉我怎样改进和发展本书的思想。

答谢（略）

参考

1．Martin, J., and J. Odell. *Object-Oriented M*ethods: A Foundation. Englewood Cliffs, NJ: Prentice-Hall, 1995

第1章 介绍

1.1 概念模型

大部分面向对象建模的书都提到分析和设计，但对于分析和设计的分界点却一直没有达成统一的意见。在对象开发过程中一个重要的原则是需要设计软件，并使得软件的结构

反映问题的结构，这项原则的一个结果是从分析和设计产生的模型结束时都有意变得十分类似，从而导致许多人认为这两者之间没有区别。

我相信分析和设计之间的分别仍然存在，但它正日益成为一个重点。进行分析时你需要努力理解面临的问题，对于我来说这不仅仅是用用例列出所有的需求。在系统开发中虽然用例即使不是必需的，也是很有价值的一个部分，但捕捉完它们并不意味着分析的结束。分析还包括深入到表面需求的背后，来得到一个关于问题本质的精神模型（或理论模型：Mental Model）。

例如有人需要开发一套软件来模仿桌球游戏，问题可能用用例来表述表面的特征：“游戏者击中白球，它以一定的速度前进，并以特定的角度碰到红球，于是红球在某个特定的方向上前进一段距离”。你可以拍下几百个这样的事件，测量不同的球速，角度和距离，但靠这些样例要写出好的仿真程序远远不够，因为除了这些表面现象，你还必须了解背后的本质，那就是和质量有关的运动定律，速度，动量，等等。了解这些规律将更容易看到软件可以怎样建立。

桌球问题非常例外，因为那些定律早已成为公理。在许多企业中，规律并没有得到很好的理解，而需要我们努力去发现它们。因此我们建立一个概念模型来理解和简化问题。某种概念模型是软件开发过程中必要的一部分，就连最不受控制的黑客们都这样做，差别只是：建立模型到底是它本身的一个过程还是整个软件设计过程的一部分。

别忘了概念模型是一个人工产物，它们代表了现实世界的模型，但它们是人类创建的。在工程方面，模型让我们更好地理解现实世界，而且，建模人员也可以使用一个或更多的模型。例如，桌球问题中，人们可以采用牛顿模型或爱因斯坦模型，只是它们的精确程度和复杂程度不一样。

建模原则 模型没有对错之分，只有适用性不同。（*Models are not right or wrong; they are more or less useful.*）

模型的选择影响着结果（系统）的灵活性和可重用性。在系统中考虑太多的灵活性将使系统复杂度提高，这是不好的设计。设计需要在建造和维护代价上进行折衷。要得到满足某种目的的软件，你需要建立一个和需求相应的概念模型。你需要你能得到的最简单的模型，不要为你的模型增加不可能用到的灵活性。

最简单的模型不一定是你首先需要考虑的，因为简单的方案需要不少的时间和精力。但这些时间和精力往往是值得付出的，它们不但使得系统易于建立，更重要的是易于维护

和将来易于进行扩展。所以值得用可以运行的更简单的软件代替原来运行的软件。

你将怎样表达概念模型？很多人把概念模型建立在他们的开发语言中。好处是你可以执行一个模型来验证它的正确性，和进行后续的研究。另一个好处是你最终将把模型转到开发语言，所以可以省区翻译转换工作。

坏处是，很容易陷入语言细节而忘记原本要面对的问题；还有很容易为特定的语言建模，使得模型向别的语言移植变得困难。

为了避免这些问题，许多人开始采用分析和设计技术，这些技术可以帮助人们关注概念上的而不是软件设计上的问题，问题域专家们也容易学会这些技术，由于它们使用图形，因而显得更有表现力。它们可以是精确/严格的，但并非必要如此。

我采用分析和设计技术的一个主要原因是为了和问题域专家一起工作，因为在概念建模时必须有问题域专家参与。我相信有效的模型只有那些真正在问题域中工作的人才能建造出来，而不是软件开发人员，不管他们曾经在这个问题域工作了多久。不过，当问题域专家开始建模时，他们需要指导。我曾经为客户服务监督、医生、护士、金融商货和公司财务分析员教授面向对象的分析和设计技术，同时，我发现，对于他们来说，有没有 IT 背景其实是无关紧要的，例如，我知道的最好的建模人员是伦敦一家医院的外科医生。作为一名专业的分析家和建模人员，我给这个过程提供有价值的技巧：我提供严格（的标准），我知道怎样使用这些技术，而且我“局外人”的眼光可以质疑模型中看似正确的地方。但是，这些是不够的，在建立分析模型时最重要的是专业知识。

分析技术倾向独立于软件技术，理想情况下，一种概念模型技术最好完全独立于软件技术，就象运动定律一样。这种独立性可以防止技术阻碍对问题的理解，而且产生的模型对各种不同的软件技术一样有用。但是实际上这种纯度不可能达到，例如，（实现）语言的变化也会影响到我们建模的方法。

在这里希望大家注意的一点是，概念模型更接近软件的接口（Interfaces）而不是软件实现（Implementations）。面向对象软件的一个特点就是接口从实现中分离，但在实际中很容易忘记这一点，因为通常的语言在这两者中没有显式的区别。软件组件的接口（它的类型：type）和它的实现（用以实现的类）之间的区别非常重要，《设计模式》一书许多基于委托的重要模式就依赖了这种区别，因此，当实现模型时也不要忘了这些区别。

建 模 原 则　Conceptual models are linked to interfaces (types) not implementations

(classes)。

1.2 模式的世界

最近一些年，模式成了面向对象界最热门的话题。它们引发了巨大的兴趣和普遍的宣传。我们也看到一些争论，其中就包括：模式到底是什么；当然难以下一个一般的定义。

模式运动有着不同的起源。近年来越来越多的人发现软件界不善于描述和利用好的设计实践经验。方法学家多起来了，但他们定义了一种描述设计本身（而不是描述实际设计方法）的语言。仍然缺乏描述基于实践设计的技术论文，这些论文可以用于教育和鼓励。象 Ralph Johnson 和 Ward Cunningham 提出的：*“哪怕有了最新的技术，项目也会因为缺乏一般的解决方案而失败”*。

模式也从不同的起点发展而来，例如，来自亚历山大·克里斯托弗的思想，来自为软件体系结构编写的手册，来自 C++的习惯用法等。

94 年《设计模式》一书的出版和 Hillside Group 关于 PLoP（“编程模式语言”）的会议带来了模式运动更广泛的知名度。

当然，软件模式思想并不局限于面向对象技术。

1.2.1 亚历山大·克里斯托弗

许多人看来，软件界出现模式这个词完全是由于亚历山大·克里斯托弗的工作，亚历山大·克里斯托弗是位于伯克利的加利福尼亚大学的一位建筑学家。他发展了一系列建筑学上的模式理论并出版了多部著作。

不过也有很多人否认亚历山大·克里斯托弗在软件模式运动发展中的中心地位，例如，《设计模式》一书的四个作者中有三个在写书之前并不了解亚历山大·克里斯托弗。

1.2.2 文字格式

模式写作的一个显著特征是它遵循的描述格式。一些人遵循亚历山大提出的格式，还有一些人则采用了《设计模式》中的格式。

通常来说，一个模式，不管它怎么编写，有四个必要部分：模式适用的*上下文*（context）陈述；模式提出的*问题*（problem）；形成解决方案过程中的*约束*（forces）；对这些约束的*解决方案*（solution）。这种格式带标题或者不带标题，但是是许多已经发布的模式的基础，这种格式很重要因为它支持对模式的如下定义：“在一个上下文中对一种问题的解决方案”。——一种定义限定模式为单一的*问题-解决方案*型。

一种固定格式使得模式和一般的软件技术作品显著区分开来，但固定格式也有它的缺点，例如在这本书中，我经常发现找不到*问题-解决方案*这种相应关系来说明一个模型单元；而且本书的几种模式都描述了对单个问题的不同解决方式，取决于不同的考虑/代价。

关于模式格式的另一项原则我毫无保留地表示赞同，那就是：模式需要命名。好处是可以丰富开发词汇，设计思想可以得到沟通。再一次强调，这不是模式独一无二的特点，这是技术作品为概念打造术语的常规做法，不过寻找模式促进了这一过程。

1.2.3 作者的抽象级别

模式产生于日常开发而不是学术发明。本书中所有的模式都是一个或多个实际项目的结果和项目中精彩部分的描述。

相信这些模式对别的开发者有所帮助，它们通常不只适用于模型本身的领域，而且经常在其他领域一样有用。例如 9.2 节中的 portfolio 模式，最开始作为金融合同分组而创建，但是，通过定义一个隐含的查询，它可以用来分组任何对象，并抽象到可以用在任何问题域中。

那么我面临的一个问题是我应该作何等程度的这类广泛抽象。如果我发现一个模式可以用于更多的问题域，那么我该让它变得有多抽象？问题是如果抽象一个模式，在使其超越原本的问题域时我不能保证它的有效性。所以我认为你必须判断模式对你的问题域是否

有用，这一点你比我肯定，或者你已经接触到相当多的问题域专家。任何超出原始问题域的模式的使用都是实验性的，但是它们可以激发你的想象力，让你问自己：“对我确实有用吗？”

1.3 本书中的模式

模式来自实际项目中具有通用性的部分，所以，可以说发现模式，而不是发明模式。

本书中的模式分为以下两类：

分析模式：一组概念，代表业务建模过程中的公共结构，可能只和一个问题域有关，也可能扩展到其他的问题域。分析模型是本书的核心。

支持模式：本身有价值的模式，在这本书里扮演特别的角色，那就是：描述怎样选择分析模型，怎样应用它们，使它们真正实用。

1.3.1 具体建模样例

离开上下文环境，有很多问题难以理解并需要建模人员有相关的建模经验。而模式提供了一些好的方法来面对这些问题。本书中的许多模式通过考察一个领域中特定的问题来处理通用的建模结果，这样易于理解。例如，可以连接到单一对象实例的方法的处理（6.6 节）；状态图的图表类型（10.4 节）；模型的知识层和操作层分离（2.5 节）；通过查询利用 portfolio 模式分组对象（9.2 节）等。

1.3.2 模式来源

如上所示，本书的模式都基于我在大型企业信息系统应用面向对象建模技术的个人实践，也就是说，模式来自我曾经参与的项目。

我希望这本书跨越不同领域并形成交叉，所以我描述了多个问题域的模型，但我没有

详细讨论模型的细节，一部分是由于客户机密的原因。

对于这些模型我并非完全自信，由于几个原因我作了一些改动。我简化了一些抽象；保留了原来的精神但让它更容易解释和接受。我还在一些具体的问题域上对一些模型进行了一些小小的抽象。有些情况下我改动了模型来反映我的意见。

对于模式的命名，我采用了用原项目命名的原则。

1.3.3 跨域模式

不管你工作在哪一个领域，我希望你学习你的领域以外的模式。本书的大部分内容包括通用建模要素和建模领域外可用的经验。其他领域的知识有助于抽象。很多专家不象我这么幸运，可以工作在多个领域。在不同领域考虑模型经常会在不相关的环境中蹦出新的主意。

我怀疑有一小部分高度通用的（商业）过程绕过了传统的系统开发和业务工程边界，诊断和治疗模型就是一个；另一个是记账和盘点模型（见第 6 章）。许多不同的业务可以共用一组相似的抽象过程模型。这对工业界各行业原来发展垂直类库的希望引起了一些大问题。我相信真正的业务框架会沿着抽象概念过程而不是传统的业务线来组织。

1.4 概念模型和业务过程重组工程（BPR）

读者可能认为本书中的概念模型是为了帮助开发计算机系统，其实它们还有其它的用途。好的系统分析员早就知道，采用一个已有的过程，并简单地将它计算机化并不是一种好的做法，计算机可以让人们以一种不同以往的方式办事。但是这种思想并不容易被人们接受，因为系统分析员们的技术好象还是太依赖于软件思考方式。如此看来，IT 业的人们还需要一段艰难的时间才能让商业领袖们理解这样的思想。

和 Jim Odell 的合作经常让我陷入到业务建模而不是软件建模；John Edwards 在 BPR 成为一个热门称号的很久以前就把他的方法叫做“过程工程”（process engineering）。用面向对象技术进行概念建模可以让系统分析和 BPR 变成同一回事。我所教授的问题域专

家们很快掌握了它潜在的功能，并且开始用一种新的方式思考他们自己的领域。只有问题域专家们才能真正使用和应用这些思想。

因此本书中的模型在讲述软件建模技术的同时也讲述了业务工程（business engineering）。尽管业务工程的重点是关于过程（process），这些模式的一大部分是静态类型模型，这样的主要原因是我在涉及的领域：在医疗保健领域我们发现尽管我们建立通用类型模型，应用到这个领域的各个部分，却很难建立多的通用动态模型。

类型模型非常重要，我喜欢把它们当作业务定义的*语言*，这样的模式提供了一种方式，带来有用的概念，并使这些概念成为大量过程建模的基础。Accountability 的概念证实在医疗保健行业中的保密政策建模时很有用，在薪酬管理中我也看到建模改变了这个过程的语言和对这个过程的理解。

1.5 模式和框架

如果随便问一个专家面向对象技术的主要好处在哪里，答案通常是*重用*（reuse）。软件界的美好愿望是有一天开发者可以利用实验和测试过的可用组件来组装系统。不过很多这样的愿望实现得很慢，可以说重用才刚刚开始，更多的是在 GUI 开发和数据库交互上，还没有出现的地方是在业务级。

在医疗保健、银行、制造业或其他行业没有组件，是因为在这样的领域没有标准的框架（framework）。为了达到信息系统的组件重用，必须建立通用的框架。一个有效的框架不能太复杂，也不能太松散，应该基于一个大问题域的有效概念模型并可以跨越问题域广泛应用。建立这样的框架非常困难，不管在技术上还是在政策上。

本书不打算为不同的工业领域定义框架，而是描述一种状况下的可选建模方法；框架是关于选择一种特定的模型。希望这本书可以鼓励人们思考这样的框架并影响它们的开发。

1.6 模式使用

模式是软件技术上一种新的发展，我们也在发展新的方法来帮助人们在他们的工作中

学习这些模式。面对书中大段的模式，很容易有一种被淹没的感觉（或不知所措的感觉）。

首先需要得到一个大致的方向，看完本章的介绍之后，建议你看看每一章的介绍，从那些介绍中可以了解到章节覆盖的主题。这时你就可以继续并阅读每一个章节了，当然这本书的写作方式并不要求你仔细读完每个章节。如果你在一个特定的领域工作，就可以阅读你认为合适的一些章节。另一种人们建议的方式是浏览书中的图表，如果发现感兴趣的地方，再看看样例，你可以从样例中看出该模式是否会对你有用。模式表也提供了模式的一个小结，可以从它开始或以后用来巩固记忆。

一旦你确定一个可能有用的模式，试一试。我发现要真正理解一个模式如何工作最好的办法就是自己找一个问题进行实验。可以只是在纸上划一个特定的模型，或写些代码看看。要尽量使模式合适（对问题），但也不要花费太大精力，因为可能这个模式并不是最合适的，那样的话没有浪费你的时间，因为你已经学到了关于模型的一些知识，可能还有关于问题的一些知识。如果模式确实不符合你的要求，尽管修改它。模式只是建议，而不是规定。不管有多适合，请详细阅读关于模式的所有说明，以确认你已经了解该模式的局限性和重要的特点。使用之前和使用之后都请别忘了这么做。如果你发现模式中一些没有给出说明的地方，不要只是骂我——发个邮件让我知道（100031.3311@compuserve.com），我非常希望能够看到别人怎样使用这些模式。

当在项目中使用模式时，我必须了解客户的意见。有些客户不喜欢自己和其他任何客户相似，他们自认为与众不同，并且怀疑外来的思想。对这些客户我不展示模式。在某个模式可能适合的地方，我用它来帮我设计问题，用这些问题引导客户走向这个模式，但我不是直接这么做，而是用问题来刺激他们。

另外一些客户高兴看到我公开使用模式，并放心让我重用以前的工作。对这些客户我在他们面前实验模式，并询问他们看他们是否喜欢。对我来说，他们必须明白，我并非把模式看作福音书，而且如果他们觉得不好，我会试试别的（模式）。因为这类客户可能存在不假思索地接受这些模式的危险。

对于回顾你自己或其他人的工作，模式也是非常重要的。对于你自己的工作，看看是否有和模式相近的，如果有，用模式试试看。即使你相信自己的解决方案更好，也要使用模式并找出你的方案为什么更适合的原因。我发现这样可以更好地理解问题。对于其他人的工作也同样如此。如果你找到一个相近的模式，把它当作一个起点来向你正在回顾的工作发问：它和模式相比强在哪里？模式是否包含该工作中没有的东西？如果有，重要吗？我

把正在回顾的工作中的模型和我知道的模式相比较，经常发现在这个过程中，当我问自己“它为什么这样处理？”的时候，学到了很多关于模式和关于问题的东西。简单地问问为什么就能学到那么多东西，真让人惊奇。

写一本书经常意味着一定的权威性，读者容易把书当作肯定性的声明。尽管一些作者觉得他们说的相当正确，我却不这样认为。这些模型都来自实践，而且最多我能肯定它们对你有用。然而，我比其他任何人都更痛苦地知道它们的局限性在哪里。要得到真正的权威性，这些模型必须被大量的应用验证，不只是我的实践经验。

这并不表明这些模式没有用，它们代表了大量谨慎的思考。就象它们在我的建模工作中给我一个主要的起点，我希望它们也能对你有所帮助。重要的是，它们是一个起点，而不是一个终点。花些时间理解它们如何工作，但也要去了解如何开发它们和它们的局限性。不要害怕继续向前，并请努力产生更新的更好的思想。同客户一起工作时，我并不把模式看成教条，哪怕是我自己创造的模式。每个项目的需求都会让我采用、改进和提高这些模式。

建模原则 Patterns are a starting point, not a destination.

建模原则 Models are not right or wrong, they are more or less useful.

注：

本文采用意译的方法，重在表达原文的意思，而不是逐字逐句的翻译，如有错误，请指正。

《分析模式：可重用对象模型》

学习笔记之二：责任模式 讨论

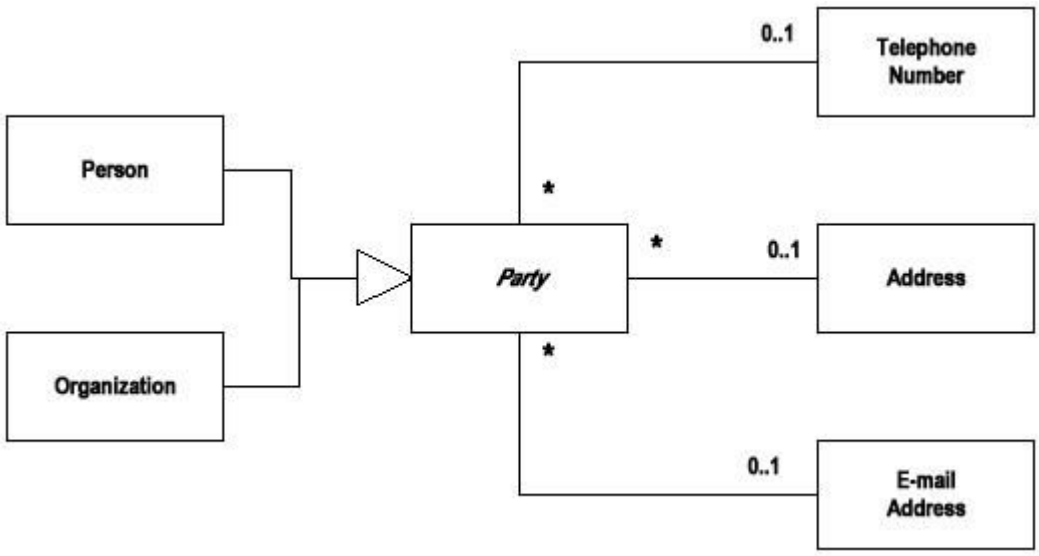
Windy J（windy.j@163.com）

2 1　责任模式

这一章关注的重点是**关系**，以及怎样为错综复杂的关系建立模型，另外，所有的插图都来自原书（《Analysis Patterns：Reusable Object Models》），并遵循 UML 标准。

2.1 1.1 Party 模式

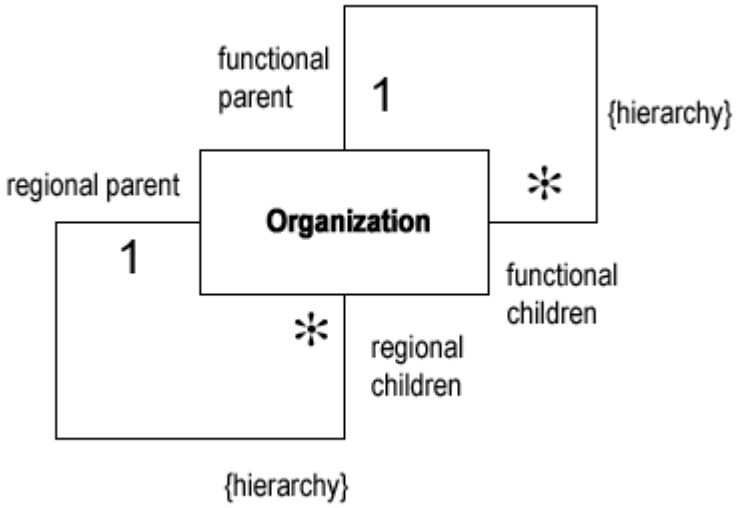
在这一章中，首先我们接触到是是 **Party** 模式，在进行系统分析和概念模型设计的时候，经常发现***人和各种各样的组织有着同样的行为***，例如，固定电话的计费可能是针对个人，也可能是一个单位；需要各种服务的时候，你可能求助于一个服务公司，或者服务公司一个特定的业务员。总之，因为人（**Person**）和组织（**Organization**）表现上的一致性，如下图所见，我们从中抽象出 **Party**，作为 **Person** 和 **Organization** 的抽象父类。



2.2 1.2 组织（Organization）的内部结构

第二步，如果我们把注意力转移到组织（**Organization**）的内部结构，就会发现一些有趣的问题，通常最常见的一种结构是金字塔结构，因此建模时可能按照这样的结构建立线性的模型，例如：

这样的模型并没有错误，但是有缺陷，首先不能满足比较复杂的组织关系，更严重的是，一旦***需要更多的层次关系***，例如存在部门直接上下级关系以及区域附属管理方式，必将引起整个模型的更改，对系统的影响可想而知，在这种情况下，最通常的改进措施是引入层次关系，如下图所示：

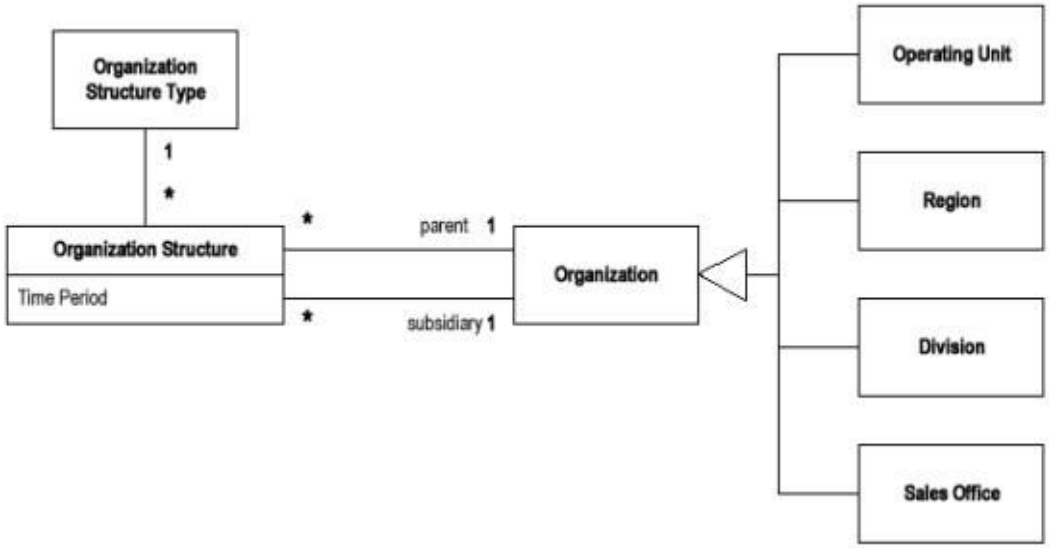


通过增加新的关联关系，可以灵活实现组织（**Organization**）之间的各种关系以及可能的变化。在上图中，**{hierarchy}**是一个约束（**constraint**）来限定关系。

2.3 1.3 组织关系抽象

第三步，在一般的情况下，以上的模式已经足够解决问题，但当这样的层次和组织关系很多而且复杂时（超过两种），例如现在流行的矩阵管理，就可以将关系本身抽取出来独立处理，如下图所示，作者此时考虑到组织结构的有效时段，所以加入了一个时间段属性来记录组织结构的存在时间。

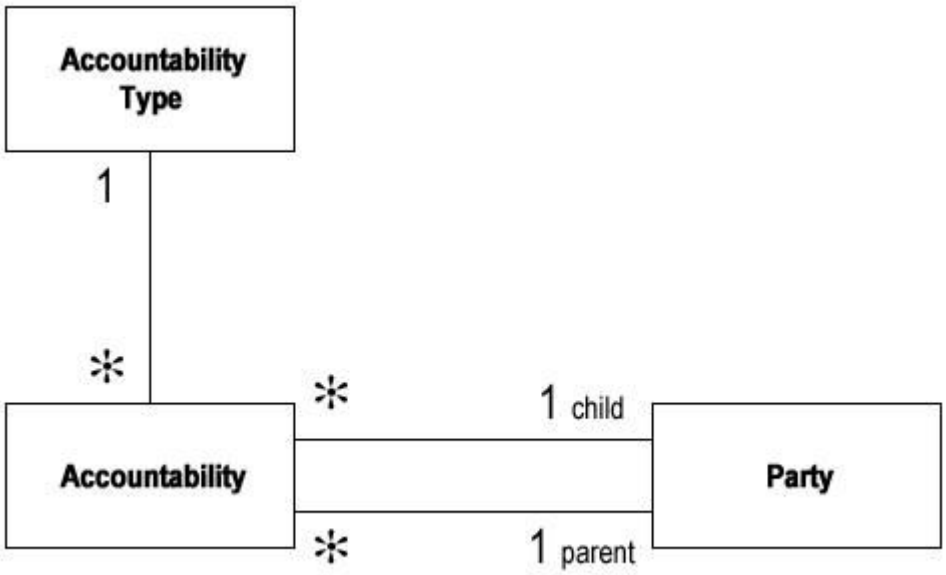
Figure 2.6



请注意，在这个模式中，**Organization Structure**才是模式的核心，在系统中，由两个**Organization**的实例（分别充当parent和subsidiary），以及一个**Type**实例来说明该结构的类型。在这样的结构中，可能存在许多的规则（**Rule**），这些规则可以根据情况分别处理：如果**Type**很多，而且规则主要跟**Type**有关，就分配给与**Type**相关联；如果**Type**并不多，但主要根据**Organization**的子类型变化，就可以分布到**Organization**的子类型中。

2.4 1.4 责任（Accountability）模式

第四步，从第一步看到，Party是Person和Organization的抽象父类，因此把Party代入上面的模式（有点象我们小时侯代数里常用的代入），正式形成责任（Accountability）模式。



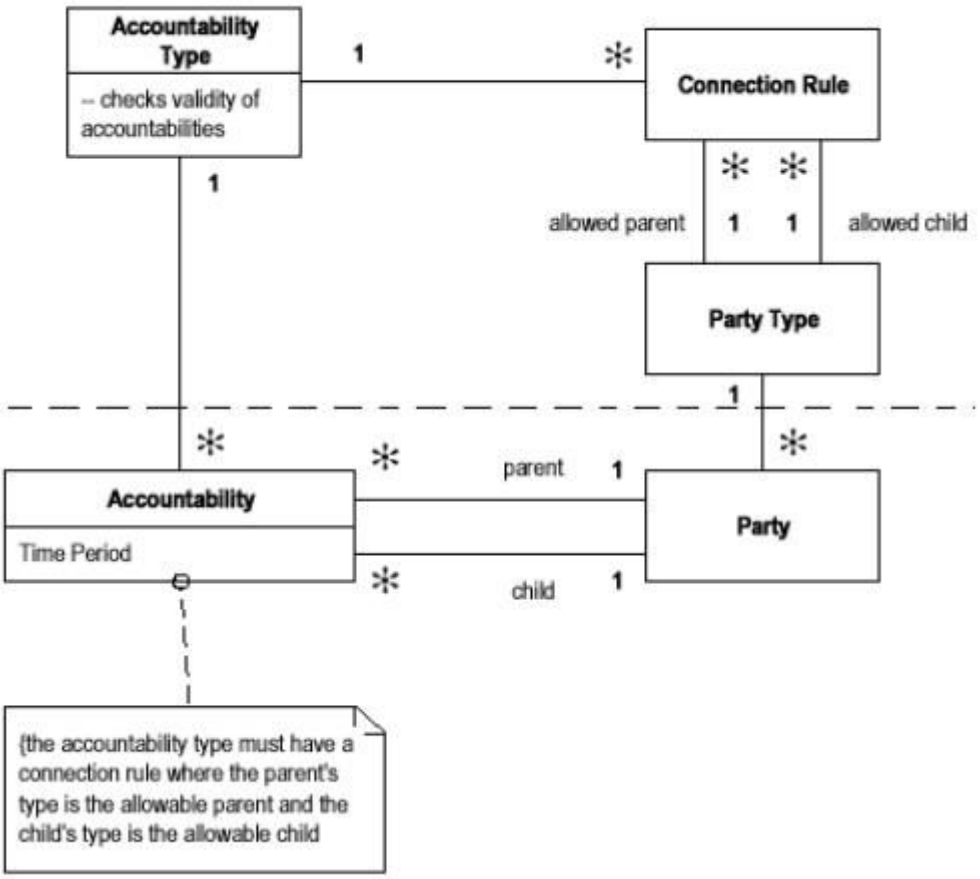
2.5 1.5 知识层（Knowledge level）和操作层（Operational level）分离

出现这样一种想法是考虑到以下情况：当**Accountablity Type**的数量比**Accountability**的数量多很多的时候，处理**Accountablity Type**的规则也变得更为复杂，要解决这样的问题，就可以引入知识层和操作层的分离。

由下图可见，用虚线隔离开的，就是知识层（**Knowledge level**）和操作层（**Operational level**），在这个模型中，知识层（**Knowledge level**）由三个类协作完成，它们分别是**Accountablity Type**、**Connection Rule**、**Party Type**，在**Connection Rule**中定义合法的**Party**关系规则，并通过**Accountablity Type**对**Accountablity**进行创建时的合法性检验。它的另一个好处就是，可以将知识层的实例化独立出来，作为操作层（**Operational**

）运行时的配置；换句话说，当知识层的规则改变时，系统的行为将被改变，而不需要任何其他代码的改动，这当然是一种比较理想化的情况。

level



由此想到，构建专家系统的设计思路也可以从这个模式得到一些启发，这是笔者一时的感触。

在原书中，如何实现这样的模型提得比较模糊，但是笔者认为，可以将它们作为正常的模型来实现，两个层次的区分只是表明它们各自担负的任务和地位不同。知识层倾向于描述系统可能存在的各种形式，并设定判断系统是否有效的各种规则；操作层则描述在这样的配置下系统实际的行为。通过改变内在的配置来改变外在的行为，就是这个模式的目的。

由于这个模式的特点，改变系统行为时不必更改操作层的代码，但是，并不意味着改变系统行为连测试也不必要做。同样，也需要调试、配置管理。

作者也提到，这样的模式用起来并不轻松，甚至在一般的系统中也不必要，但当你发现有必要用它的时候，别犹豫（感觉象用降落伞一样）！

2.6 1.6 小结

从简单到复杂，前面分五步介绍了适用于解决**Party**及其关系的各种模式，每种推荐的模式都有其表现的机会，希望我这篇文章可以起到一些抛砖引玉的作用，并欢迎大家对其中的错误进行指正，欢迎发表意见，进行交流。

原文参照《Analysis Patterns：Reusable Object Models》，Chapter 2，Martin Fowler

作者：Windy J

参考文章：Party, by Martin Fowler

《分析模式：可重用对象模型》

学习笔记之三：观察和测量

Windy J（windy.j@163.com）

3 1　观察和测量（Observation and Mesurements）

许多计算机系统记录现实世界中各种对象的信息，这些信息通常表现为计算机系统中的记录、属性、对象等其他各种各样的形式。最典型的方式是把某项信息记录成某个对象的一个属性，例如，一个人体重 70 公斤记录成“人（Person）”类的体重（Weight）属性，值为 70。本章将讲述这种方式的不足，并提出一些更合理的解决方法。

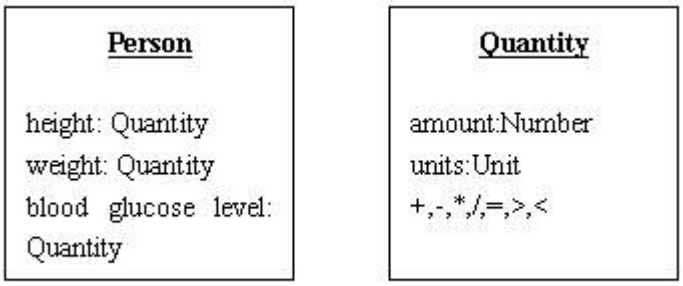
本章的模式来自与医疗领域有关的项目，所以采用了许多这一领域的例子。

本章中的模式图均由笔者以通用的 UML 格式重画。

3.1 1.1 数量（Quantity）

当用上面提到的方法记录数据时，最常见的不足之处就是单纯的数字不足以代表它的意义，是 70 公斤，70 磅，或者别的什么？我们需要确切的单位。可不可以创建属性和个单位之间的关联？可以，但那样系统中将会出现错综复杂的关联，从而增加了系统的复杂度。

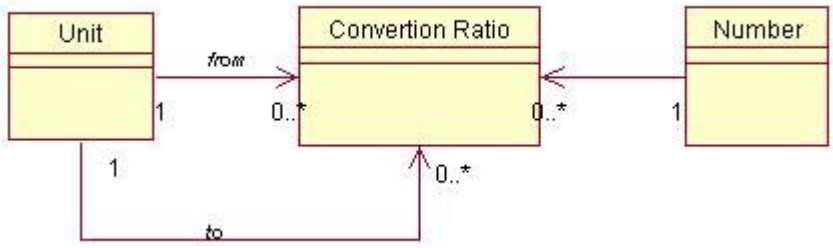
如果改用一个 Quantity 类来表达，这样的意义将会更简单。如下图所示，Quantity 类包括一个 amount 属性，记录数值，一个 units 属性，记录单位，并支持一般的运算操作。



3.2 1.2 转换比率（Conversion Ratio）

这个模式主要是解决不同的单位间转换的问题，通过需要转换的不同单位，以及它们之间的比率，就可以实现各种固定比率的单位转换，这在系统中也是相当有用的。

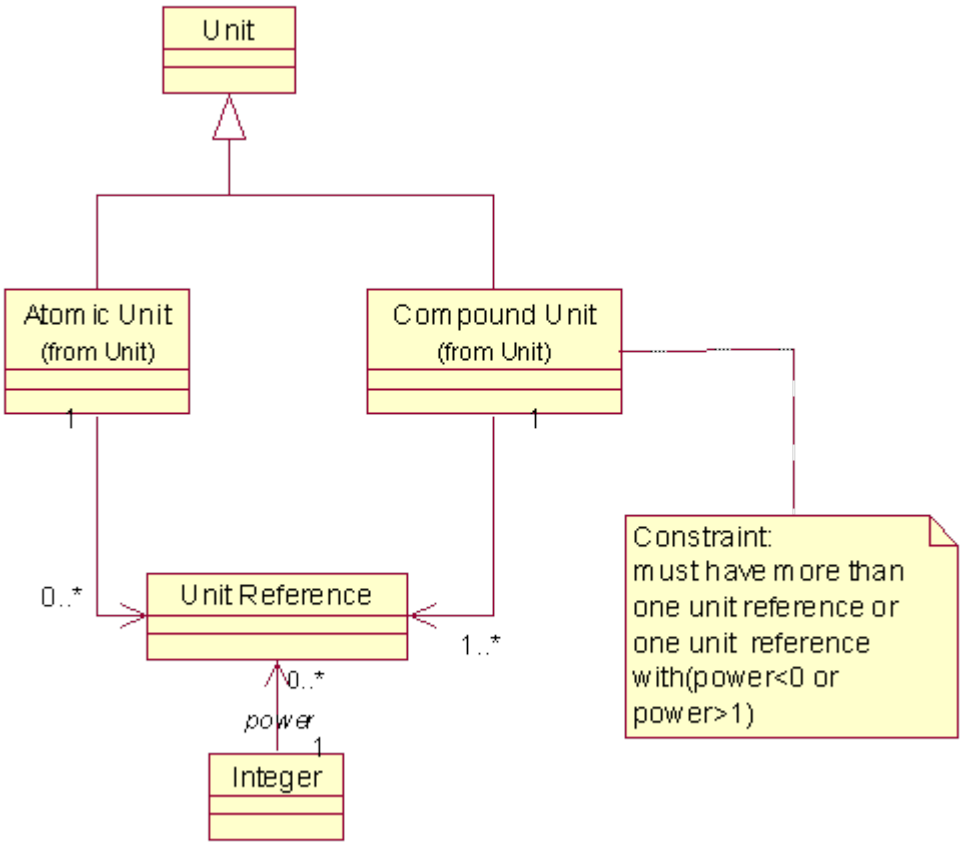
但它也有不足的地方，如果转换比率不固定，可能需要额外的计算函数。对于一般情况来说这个模式已经足够了。



3.3 1.3 复合单位（Compound Unit）

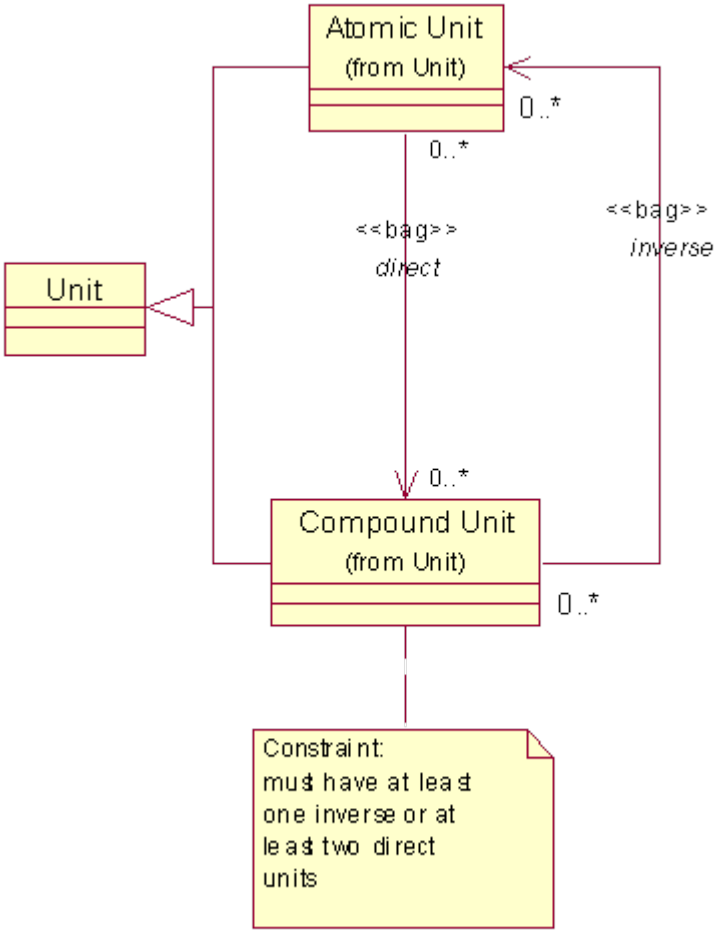
单位可以是复合的也可以是不可再分解的（基本单位），如何来表达复合单位？请看

下面两个模型：



在这个模型中，复合单位（Compound Unit）通过单位引用（Unit Reference）来记录基本单位（Atomic Unit）和它们的幂（power）。这是一个较直接的模型。

下面这种模型由于使用了 bags，从而显得更紧凑：



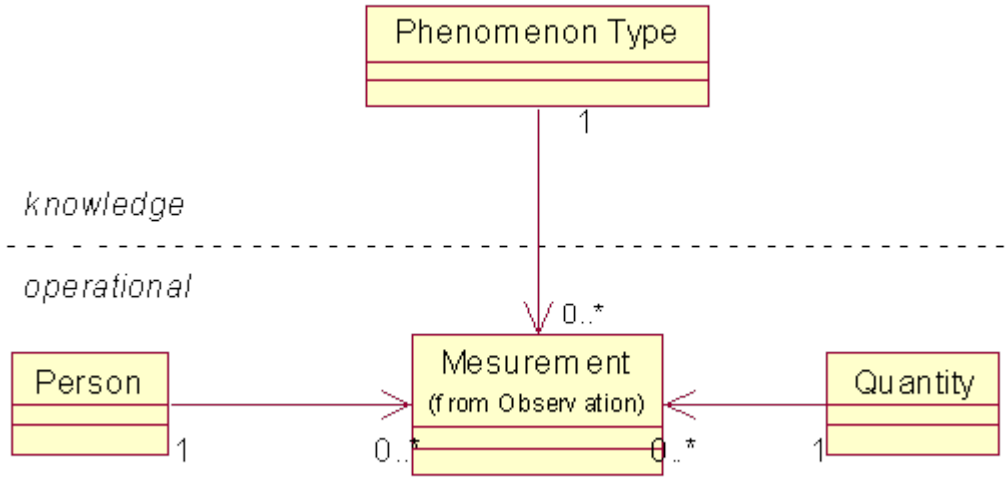
这两种模型差别不大，只是一个采用了 bag，而另一个用了 Unit Reference，至于在什么时候采用这样的模型，要不要引入复合单位，则完全取决于客户和应用的需要。

3.4 1.4 测量模式（Mesurement）

当一个复杂的系统中包括成千上万的测量活动，需要记录那些测量数据时，光靠数量模型是不够的。如果还是把测量数据当作属性来处理，系统中可能充斥着杂乱无章的属性，从而导致一些非常复杂的界面（Interface）。这里的解决方案是把各个测量项目（例如医院需要的身高、体重、血压、血糖浓度等）当作对象，并把对象的类型引申成“现象类型

（Phenomenon Type）”，这个时候，问题的复杂度就转移到各种各样的 Phenomenon Type，以及各个测量（Mesurement）实例。

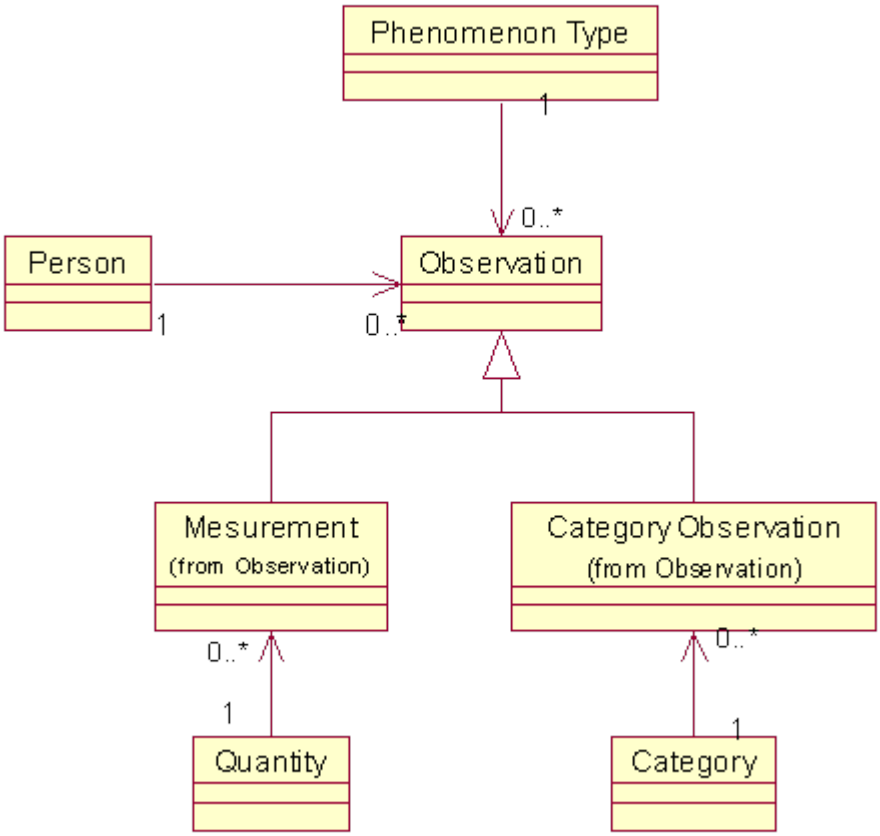
如下图模型所示，测量（Mesurement）实例已经包括了类型（Phenomenon Type），测量对象（Person），结果（Quantity）。依照我们上一章提到的知识层和操作层分离的思想，可以把现象类型（Phenomenon Type）划分到知识（knowledge）层，因为这些对象表示了千千万万要进行的测量项目，这部分信息是相对不变的。



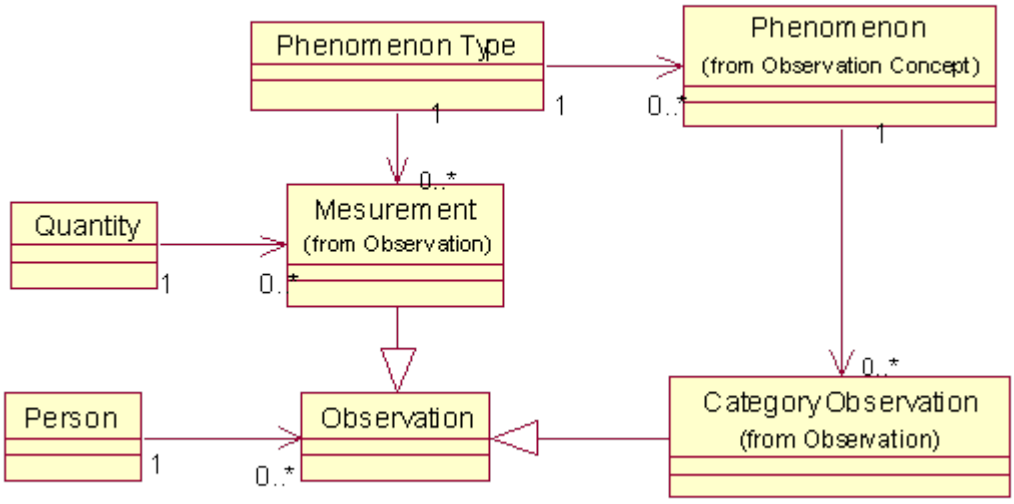
也可以将单位信息跟类型（Phenomenon Type）关联，而测量实例中只保存单纯的数值，但是为了对类型（Phenomenon Type）进行多单位的支持，我们还是倾向于采用上图的模型。

3.5 1.5 观察模式（Observation）

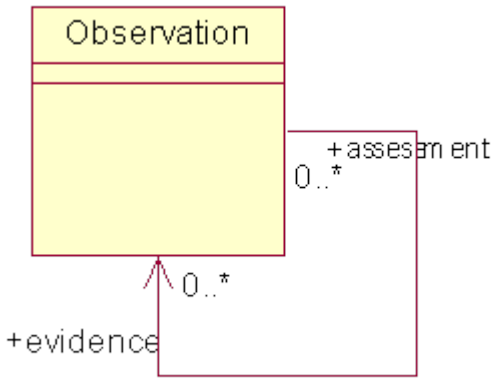
由于进行测量时除了要记录许多测量项目具体的数值，例如身高，体重，血糖浓度等之外，还有一种测量项目只需记录条目类别，例如血型，常见的有 A，B，O，AB 四种，又例如体形又可以记录为肥胖，正常，偏瘦等类别，所以我们有必要在上述的模型补充种类（Category），从而正式形成观察（Observation）模式：



在这里，种类（Category）最好和现象类型保持较为简单的关系，可以减少歧义，降低复杂性。在下面的图中，Category 被改名为 Phenomenon，并被移动到知识层，由 Phenomenon 来定义某些 Phenomenon Type 的一组可能的取值。



对于观察（Observation），还要提到的是，在实际的诊断过程中，医生需要根据某些现象来推断某些测量活动，而这些测量活动又为其他的测量记录提供证据，所以可能还存在以下的关系：



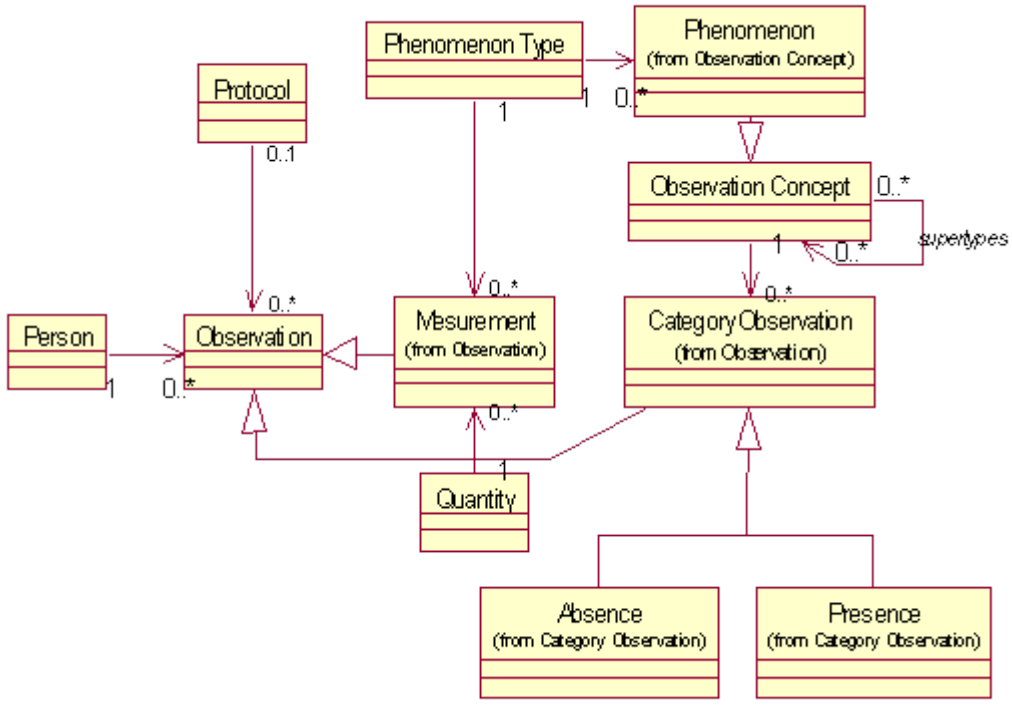
3.6 1.6 对 Observation 的完善

在以上的图中，还遗漏了一种可能的情况，那就是对于有的记录项目，结果可能就是

简单的“有/存在（Presence）”或“没有/不存在（Absense）”，例如针对各种疾病而言，某个人或许有糖尿病，或许没有。这就是 Category Observation 出现两个子类 Absence 和 Presence 的原因。

在这里，为 Phenomenon 增加了一个父类 Observation Concept，这是为了在处理例如疾病的时候它们可以不必跟 Phenomenon Type 相关联。

还有，可以看到，在 Observation Concept 上加入了一个名为 SuperType 的自关联，是因为各类疾病可能互相存在关系，其中的一种关系是上下级的关系，例如糖尿病和 A 类糖尿病，B 类糖尿病；如果存在 A 类糖尿病，那么一定存在它的上级；如果 A 不存在，并不能表示上级存在与否。



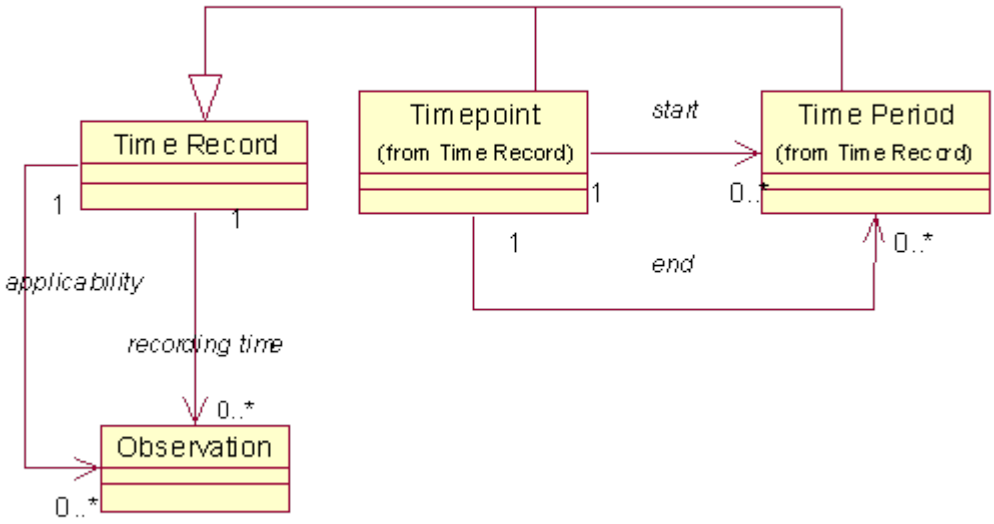
3.7 1.7 协议（Protocol）

在上图中，Protocol 是一个重要的知识层概念，表示进行观察时采用的方式。例如我们量体温的时候体温计可以放在腋下或含在口中，某些时候，很有必要记录这些不同的方式。而且，可以根据不同的观察方法判断结果的精确度和灵敏性。

把这部分信息单独放在 Protocol 中，简明而易于处理。

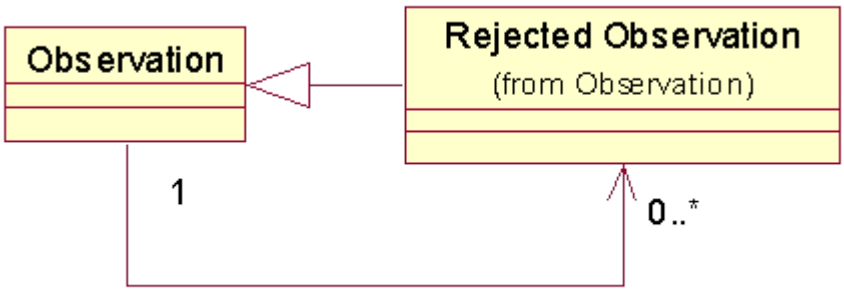
3.8 1.8 双时间记录（Dual Time Record）

Observation 经常有一个有限的有效时间，在下面的图中，Observation 跟两个时间（TimeRecord）有关，一个表示记录时间，另一个是发生时间，Time Record 可以同时表示时间点和时间段，这可以由它的子类体现。



3.9 1.9 被拒绝的观察

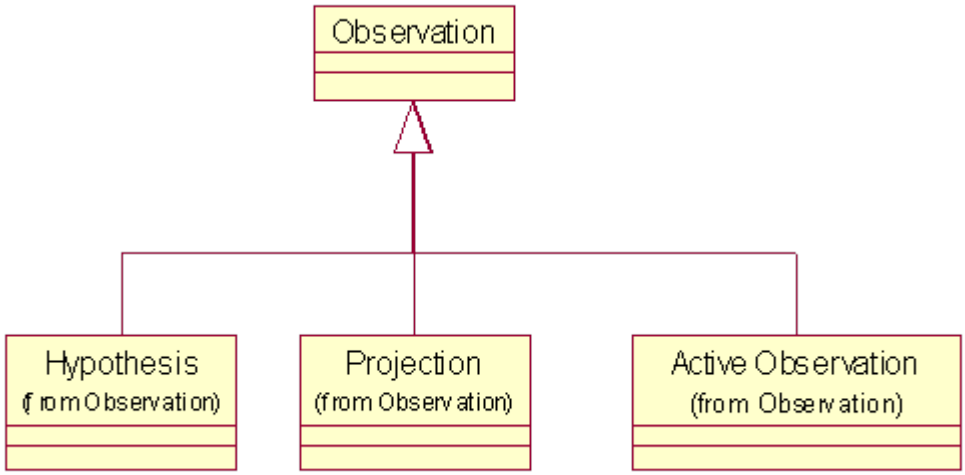
在进行观察时，总有一些对象是被拒绝的，可能是错误的，不合适的，过期的，那么，这部分的 Observation 成为 Rejected Observation，并作为 Observation 的子类。每一个 Rejected Observation 都与一个 Observation 有关，而这个 Observation 正是确定原来的 Observation 成为一个 Rejected Observation。



3.101.10 主动的观察、假设（Hypothesis）和 Projection（估计）

在医疗系统中，一个医生经常需要根据已有的观察来判断，做些推测和估计，这些也许并不是实际的测量，因此在这里为 Observation 引入可能性，由它产生主动的观察、假设（Hypothesis）和 Projection（估计）三个子类。Hypothesis 需要进行更多的观察来确定，如果 Projection 是真的，则需要更多的主动观察来支持它。

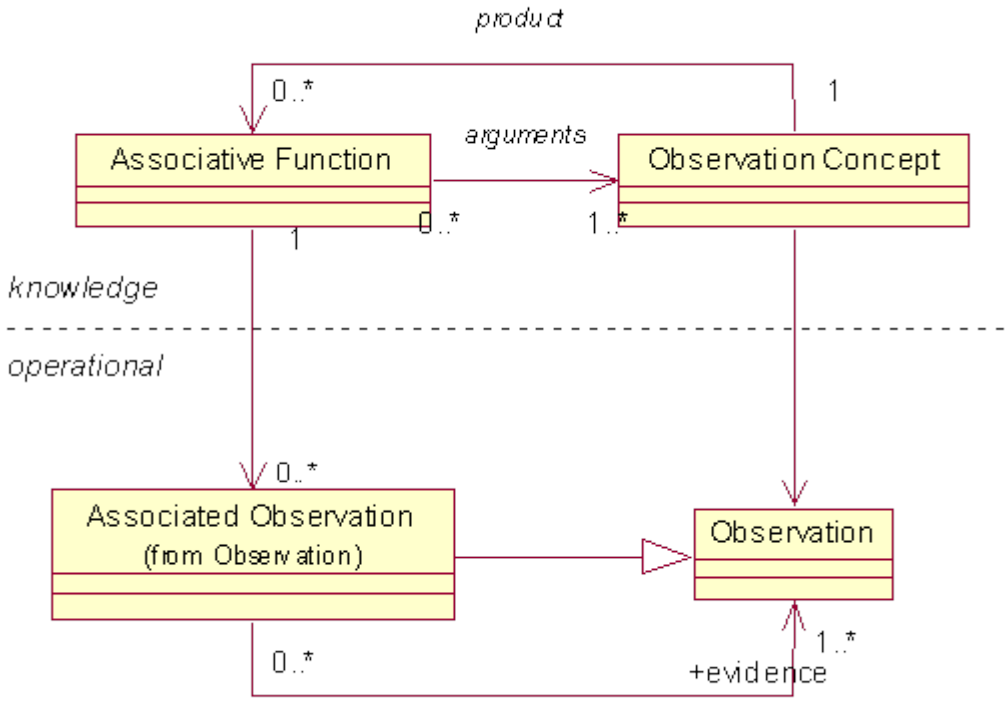
观察结果的确定程度会引起大家的注意，但是这个方面可能由医生来决定比较好。



3.11 1.11 观察关联

在一个诊断里往往有一连串相关的观察，在以下的模型里，观察（Observation）可以与观察（Observation）相关联，在知识层 Observation Concept 也要与 Observation Concept 相关联，这样的关联通过在知识层定义 Associative Function，通过 Observation Concept 作为参数和关联的结果，制订观察相关联的规则，从而对给定观察，可以找到与其有关的关联。

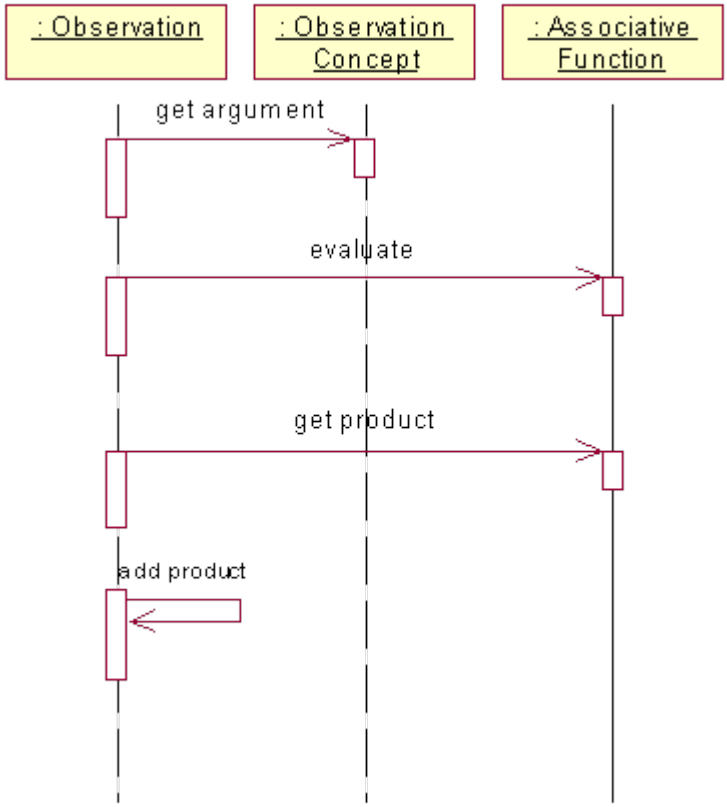
知识层与操作层并非完全形成镜向映射，可以看到，Associative Function 并不是 Observation Concept 的子类。一个 Observation Concept 可能有几个 Associative Function 来得到它作为结果，但一个特定的 Observation 只有一个 Observation 集合作为证据。



3.121.12 观察过程

有必要再来看看进行观察的过程，一个简要的过程是，从进行观察（可能是某些测量）开始，然后从知识层以及上一节的模型得到与之相关的观察，并把这些观察列入建议在下一步进行的观察中，这是一个循环的过程。

更复杂一点的规则和描述是，从一个观察开始，找到 Observation Concept，查找该 Observation Concept 作为参数的 Associative Function，对于每个 Associative Function 进行验证，得到相应的 Observation Concept 作为 product，将 product 作为答案进行下续的观察，也是可以循环的。可以观看序列图如下：



一个更加详细的观察过程还包括把观察区分为主动的观察、假设（Hypothesis）和Projection（估计），并对其中的Projection（估计）和一部分活动的观察进行分析，得到后续的观察，对其中一部分活动的观察根据互相矛盾的观察结果来判定它们是否可被拒绝（Rejected Observations）。

3.131.13 小结

从常见的数量（Quantity）模型开始，到最后整个观察过程的介绍，这一章的内容包括医疗系统中可以适用的通用模型，其实这些模型并非用于特定的某个领域，如果发现它们跟你的项目情况有相似之处，就可以考虑使用或做一些改动；而下一章，则会介绍观察模式在企业分析中也同样适用。

原文参照《Analysis Patterns：Reusable Object Models》，Chapter 3，Martin Fowler

作者：Windy J

参考文章：

From Analysis to Design of the Observation Pattern

Observation Model

An Extension and Implementation of Fowler's Observation Analysis Pattern

The Validated Observation Pattern

Role 分析模式（一）

角色对象基本概念

概要

在任何应用系统中，每一个对象都可能扮演多种角色，你在家里是父亲，在公司则可能是一个程序员，一个为你提供原材料的公司可能同时又是你的客户。，这样的问题一次又一次的出现，我经常看到应用系统不能很好处理这些问题，在本文中，我将仔细描述这一重要的分析模式，并使用 TAOERP 基本业务元素(TBCC)展示如何使用这一分析模式处理组织机构相关的问题。

By 石一楹

本文从《ERP 之道》www.erptao.org 转载

==========

一个应用系统经常需要某一个对象在不同的上下文中具有不同行为的情形，最常见的例子是客户和供应商的问题。

例子：

某制鞋企业有很多为它们提供真皮的合作公司，在处理采购订单时，这些合作公司是它的供应商，但这些合作商同时从该制鞋企业采购皮鞋，所以在处理销

售订单时，这些公司又变成了它的客户。

许多建模人员在处理这类问题的时候，经常轻率地做出判断，当用户需求不能满足时，他们才发现这样的判断是不正确的。

正如Martin Fowler所言，作为一个分析模式，我在这里主要关心在何种情况下需要使用什么样的模型，而并不是太关心具体的实现如何。本文中出现的Java代码虽然来自Tao BC库，但是出于示例的目的，都进行了简化。读者也应该把主要精力放在模型本身，或者说是接口上而不是实现。

我要提醒的是，本文虽然论述了比较复杂的Role建模方法，但是读者在应用这些模式之前需要仔细考虑它们是否必要应用到你的模型之中。因为，任何一种复杂的模型虽然能够解决复杂的问题，但是不可避免地会带来额外的开销，所以，如果能够用简单的模型处理你手头上的问题，不要用更复杂的。面向对象和框架开发方法最大的好处之一就是能够在你需要的时候进行修改而对系统的影响最小。按照Kent Beck的Extreme Programming 理论：“只做你现在需要的”。当然，一旦问题发生变化，你需要用其它面向对象的方法进行Iterator，这是另外一个主题，我也许会在其它的专栏中介绍。

上下文和动机

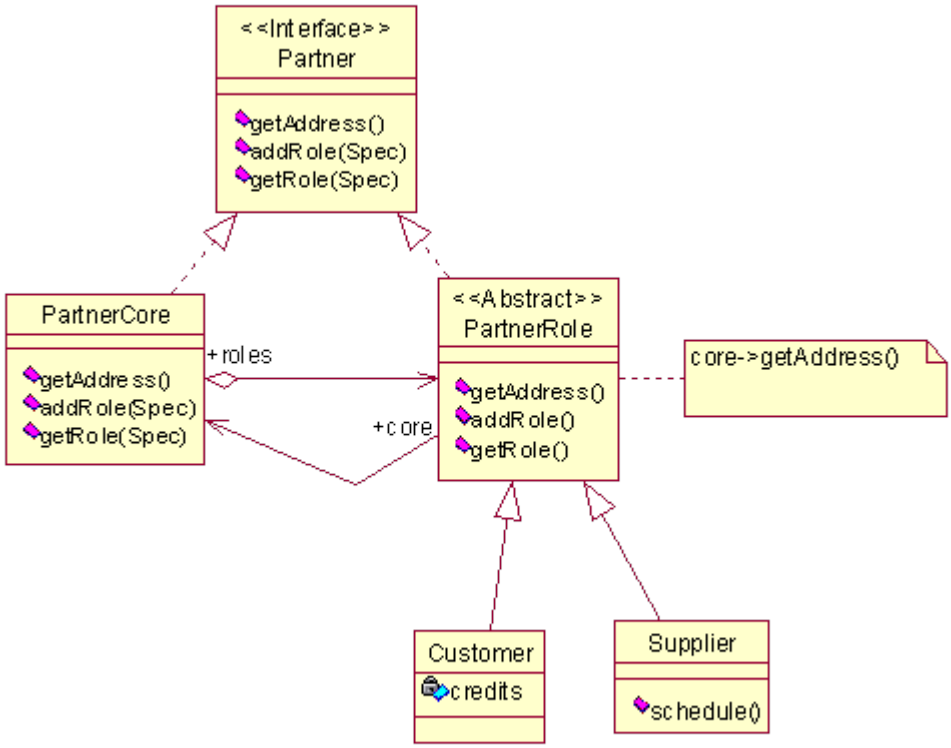
假设我们现在要开发一个企业销售管理系统，在这样的系统中，关键的抽象之一就是合作伙伴客户，因此我们的设计模型中将包括客户Customer类。该类的接口提供对客户名字、地址、电话、Email，客户信用度等属性的操作。

现在，假设我们也需要一个采购管理系统，这时候我们需要一个供应商的抽象，尽管供应商在很多方面和Customer一样，但是譬如供应商的提前期等操作显然和客户有所不同。这是我们抽取一个不同的Supplier类的原因。

但是，当我们的销售系统和采购管理系统一起集成到供应链管理的时候，我们会发现独立抽象Customer和Supplier这样的类会碰到很多问题。一个供应商可能同时是你的客户，又或者你的一个供应商后来变成了你的客户。你也许考虑供应商和客户从一个抽象的Person类继承。

但是，这还是有问题。首先，从对象标识的角度来讲，虽然我们的Customer和Supplier继承自同一个Partner，但是他们的对象属于不同子类的实例，所以他们不可能相同。因此，代表同一个合作伙伴的Customer和Supplier的两个对象具有不同的对象标识。它们之间的相同只能通过其它的机制模拟实现。如果两个对象指代同一个实际对象，他们从Partner上继承的属性应该相同。但是，我们会在多态搜索时发生问题，如果我们搜索系统中所有的Partner,那么相同的Partner(包括、供应商、客户)可能会重复出现，我们必须小心处理这些重复问题。

角色模式把对象在不同上下文（销售系统、采购系统）相关的视图建模成 Role Object,这些角色对象动态地连结到核心对象(Core Object)。这样形成的对象聚集表达一个逻辑对象，它能够处理包含多个物理对象的问题。



象 Partner 这样的关键抽象建模为一个接口，也就是没有任何实现。Partner 接口指定了处理类似于伙伴地址、帐户这样的通用信息。Partner 另一部分重要的接口是维护角色所需要的接口，象上面的 adRole()和 getRole().PartnerCore 类负责实现这些接口。

Partner 的具体角色类由 PartnerRole 提供，PartnerRole 也支持 Partner 接口。PartnerRole 是一个抽象类，它不能也不打算被实例化。PartnerRole 的具体子类，如 Customer 和 Supplier，定义并实现特定角色的具体接口。只有这些具体角色类在运行时被实例化。Customer 类实现了该伙伴在销售系统上下文中的那一部分视图，同样，Supplier 实现了在采购系统中的那一部分视图。

像供应链管理系统这样的一个客户（使用这些类的代码），可能会用 Partner 接口来存取 PartnerCore 类的实例，或者会存取一个特定的 PartnerRole 的具体类，如 Customer.假设一个供应链系统通过 Partner 接口获得一个特定的 Partner 实例，供应链系统可能会检查该 Partner 是否同时扮演 Suppiler 的脚色。它会使用一个特定的 Specification 作为参数做一个

hasRole()调用。Specification 是用于指定需要满足的标准的一个接口，我们会在后面考察这个问题。现在处于简单目的，我们假设这个 Specification 就是一个字符串，该字符串指定了需要查找的类的名字。如果该 Partner 对象可以扮演一个名为”Supplier”的角色，销售管理系统可以通过 getRole(“Supplier”)得到具体的供应商类的对象。然后供应链管理系统就可以通过此对象调用供应商特定的操作。

什么时候使用该模式

如果你想在不同的上下文中处理一个关键抽象，而每一个上下文可能对应自己的一个应用程序，而你又想把得到的上下文相关的接口放入同一个类接口中。

你可能想能够动态地对一个核心抽象增加或删除一个角色，在运行时而不是在编译时刻决定。

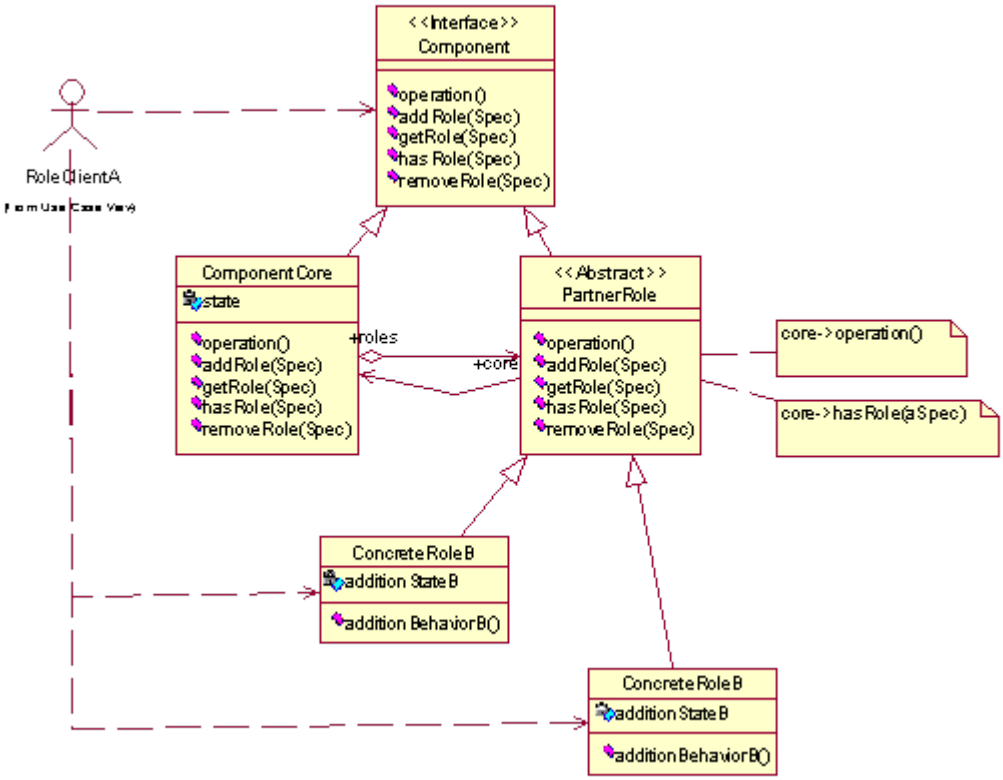
你想让角色/客户应用程序相互独立，改变其中的一个角色可以不影响其它角色对应的客户应用程序。

你想能够让这些独立地角色互相转换，并能辨别不同的角色实际上属于同一个逻辑对象时。

但是，该模型并不适用于所有情况，如果在这些角色之间具有很强的约束和交叉关系。这种模型不一定适用于你。尽管我们后面可以看到，某些约束和关系用 Role 来处理可能更直观和简洁。

结构和原理

下图显示了 Role Object 的基本结构：



我们可以看到客户应用程序大多数情况下使用 Component 接口来进行一些通用的操作。如果该应用程序并不关心具体的子类。在上面的供应链管理系统中，如果我们只关心 Partner 的一些基本信息，譬如地址、电话等等，客户应用程序只需要存取 Partner 接口(Component)。Component 接口还提供了增加、修改、查询和删除对应 Specification 具体类的管理接口。客户应用程序在需要特定角色的操作时，可以使用 getRole(aSpec)得到。

对那些通用的操作而言，真正操作是由 ComponentCore 来实现的，不论客户使用 Component 接口还是具体的角色类。在客户应用具体类的时候，由于每一个具体类都继承了 ComponentRole, ComponentRole 将这些操作传递给 ComponentCore,最后由 Core 来完成操作。我们同样也可以看到，那些角色管理的接口也是有 ComponentCore 来实现的，它有一个所有具体对象的列表，可以在其中进行查询和其他操作。

使用 RoleObject 的优缺点

从概念上来讲，Role Object 为系统提供了一个清晰的抽象。属于同一个逻辑对象的各个不同上下文中的角色都服从于这个抽象的接口。Role Object 也提供了一个一致的接口对这些角色进行管理。这种一致的管理使得对这个关键抽象具有很强的可扩展性，你可以动态增加、删除角色而不影响客户应用程序。也就是使得客户应用程序和对应 Role 的绑定很少。

如果我们从面向对象的语言所能提供的设施来看，Role 角色实际上提供了一种多重继承的特性。在我们上面的例子中，如果一种语言提供多重继承，我们很可能会让该 Partner 同时从 Customer 和 Suppiler 继承下来。但是，现在绝大多数面向对象语言都不支持这样的继承。即使支持的话，也没有办法动态增加和删除它的继承类。

但是，Role Object 模式的应用不可避免地带来一些麻烦。首先如果客户代码需要使用一个具体类的接口，它必须首先询问这个 Component 是否能够扮演那样的角色。语言本身也不能进行强制的类型检查。

问题在 Role 之间具有相互约束的时候变得更加复杂。我们在这里申明，如果你所构建的具体类之间具有很复杂的相互约束关系，那么 Role Object 可能不是你的选择。当然，对 Role Object 进行适当的扩展和调整可以很好地处理某些约束关系。我们会在后面详述。

实现

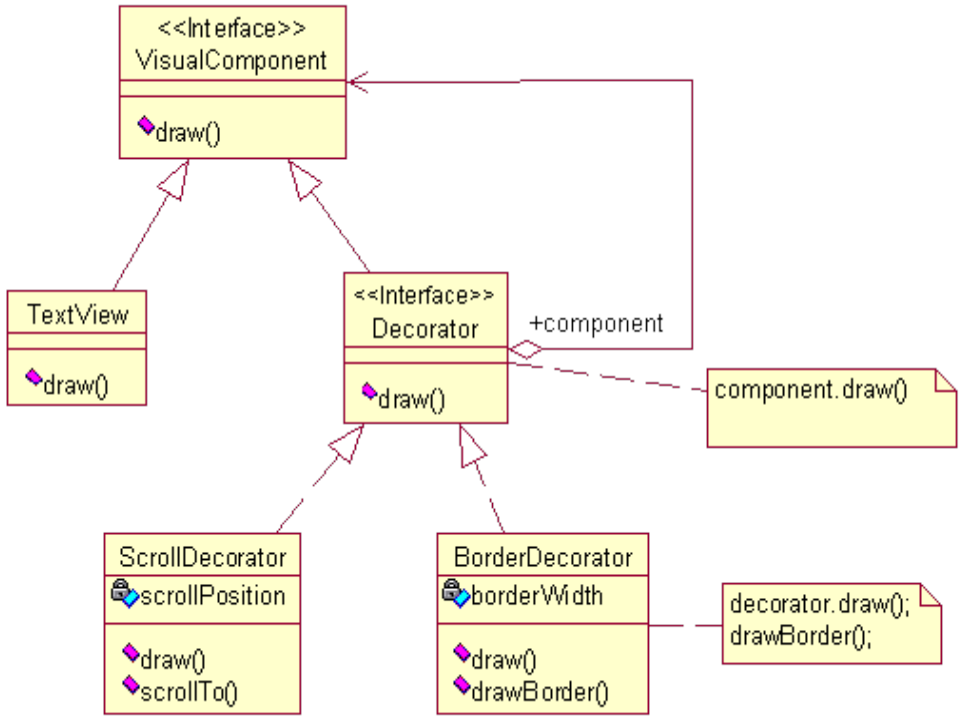
基本实现

实现要从 Role Object 的基本意图来着手，使用 Role 进行建模最主要的两个目的是透明地扩展关键抽象、动态管理角色。

在面向对象社团中，已经有很好的实践和理论来处理这样的问题。Decorator 是满足透明扩展的范例，而 Product Trader 模式可以让客户进行对象创建，这些被创建的对象匹配一个特定的产品角色协议并且满足附加的 Specification.,使用这两个广为模式社团所承认的模式使得 Role Object 模式更加坚固。

我们所要解决的第一个问题就是 Component、ComponentCore 和 ComponentRole 之间的关系，从上面的原理图可以看到，Component 抽象了所有角色都具有的通用接口。以后，我们可以使用角色管理协议增加、查询、删除角色。这些角色都从 ComponentRole 继承得到，它负责把具体角色的操作传递给 Core 来完成，也就是说 ComponentRole 实现了对 ComponentCore 的装修，同时，所有的具体类也对 Core 进行了装修，这一点和 Decorator 十分相似，但是至少有两点与 Decorator 是不同的：

1． 意图：Decorator 按照 GOF 的说法是动态地给对象增加一些额外的职责，也就是增加功能。我们可以看下面的一个 Decorator 的实例：



在这里，核心的元素是 TextView，ScrollDecorator 和 BorderDecorator 的目的是为了透明地为 TextView 增加滚动条和边框，而客户应用程序仍然可以使用 Component 接口，使用抽象的 Decorator 使得装修的增加不会产生组合爆炸问题。

Role 模式的意图是展现不同上下文中的不同角色，而其中的 ComponentRole 和 ComponentCore 之间具有等同的接口，也就是 ComponentRole 虽然包装了 ComponentCore，但是它的目的不是增加功能而是作为一个中间传递者。把具体 Role 的所有通用接口（同时也是关键抽象 Component 的接口）转移到 Core 的具体实现上。

2. 核心实例和包装实例之间的关系

装修模式参与者之间的装修者都相互链在一起，一个装修者指向另一个装修者，最后指向核心，而所有的 Role 都直接指向核心，这是因为装修生成的最后类的实例贯穿了所有的包装物，而角色之间基本上都是相互独立的，除非它们之间具有约束，这是后话。

我们关心的第二个问题就是角色的管理,角色的管理需要我们考虑下面几个重要的问题：

1. 角色对象创建：角色实现了对核心对象的运行时包装，所以最重要的问题是如何创建角色实例，以及这些实例如何与核心对象相连。注意客户代码不知道如何以及何时创建这些角色对象，创建过程由核心来控制。

2. 角色对象删除：角色对象的删除是一个标准的垃圾收集问题。某一个客户代码应当无法删除一个角色对象，因为它不知道是否还有其他客户在使用这个角色对象。

3． 角色对象的管理：为了让一个核心对象能够管理它的角色，Component 接口声明了一个角色管理协议，其中包括对角色对象的增加、删除、测试和查询。为了支持该接口协议，核心对象维护了一个 Map,其中由角色的 Specification 映像到具体地角色实例。

下面是一个简单的例子：

```
public interface Partner {

public String getAddress();

public void addRole(String roleName);

public boolean hasRole(String roleName);

public void removeRole(String roleName);

public PartnerRole getRole(String roleName);

}



class PartnerCore implements Partner {

String address;

private Map roleMap = new HashMap();

public String getAddress() {

return address;

}

public void addRole(String roleName) {

PartnerRole aRole = CustomerRole.createFor(roleName,this);

if (aRole!= null)

roleMap.add (aRole);

}
```

```
public boolean hasRole(String roleName) {

return roleMap.containsKey(roleName);

}

public void removeRole(String roleName) {

if hasRole(roleName) {

roleMap.remove(roleName);

}

}

public PartnerRole getRole(String roleName) {

if hasRole(roleName) {

return (PartnerRole) roleMap.get(roleName);

} else {

return null;

}

}

}
```

在这里，我们使用 roleName 作为 Specification,角色的 Specification 和实际的 Role 实例之间的映象用一个 Map 来实现。

现在我们定义一个 PartnerRole 抽象类：

```
abstract class PartnerRole implements Partner {

private PartnerCore core;

public String getAddress() {
```

```
return core.getAddress();

}

public void addRole(String roleName) {

core.addRole(roleName);

}

public static void createFor(String roleName, PartnerCore core) {

Creator creator = lookup(roleName);

if (creator == null) {

return null;

}

PartnerRole aRole = creator.create();

if (aRole != null)

aRole.core = core;

return aRole;

}

}
```

这里有几点需要注意，PartnerRole 是一个抽象的类，它不能被实例化，对所有角色都通用的操作以及角色管理协议操作均被重新定向到 core。在这里 PartnerRole 最有意义的操作是它的静态方法 createFor，注意 createFor 的格式，它的第二个参数是一个 ComponentCore 而不是更通用的 Component 接口，因此我们就限制了一个 ComponetRole 不会担当 ComponentCore 的责任。

接下去需要实现具体的 Role 对象，譬如说我们的 Customer,Supplier:

```
public class Customer extends PartnerRole {

public Money getCredit() {
```

…….

}

}

public class Supplier extends PartnerRole {

public void schedule() {

…..

}

}

下面是用户可能的使用方法：

Partner aPartner = Database.load(“North Bell”);

Customer aCustomer = (Customer)aPartner.getRole(“Customer”);

if (aCustomer != null) {

Money credit = aCustomer.getCredit();

….

}

这里在获得特定的角色之后需要进行强制的类型转化。

本文从介绍 Role Object 的动机开始，逐渐深入到它的原理、组成、优缺点和基本实现，并给出了一个基本的实现。Role Object 的应用和实现在许多方面都需要进行扩展，包括如何使用 Specification 模式管理 Role,应用 Product Trader 模式创建 Role，以及如何处理 Role 之间

的相互约束问题。同时，Role Object 实现 Role 概念的一种方法，我将在后续的文章中继续本文的内容，提供更为深入和广泛的讨论和研究。欢迎建议。

关于作者

石一楹是一个专注于面向对象领域的系统设计师。目前的工作是国内一家 ERP 公司的 CTO. 他住在浙江杭州，有一个两个月的女儿。

Role 分析模式（二）

角色对象创建和管理

概要

Role Object（一）提出对解决Role问题的基本方法，但是我们还有许多问题需要深入，这些问题包括如何创建具体的Role实例，如何管理这些角色并实现角色之间的约束。
By 石一楹

本文从《ERP之道》www.erptao.org转载

==========

Role Object提出了对解决Role问题的基本方法，但是我们还有许多问题需要深入，其中的一个问题是如何创建具体的Role实例。

动机

对Role的创建有几个基本的需求。前面我们使用角色的名字作为Specification来创建Role.譬如我们用字符串”Customer”作为Specification来创建Customer这个具体的Role对象。在大多数情况下，这样做可能就足够了，但是有时候确不行。

假设我们有一个核心的抽象叫做Person。我们知道Person可以是员工，所以有一个叫”Employee”的角色。但是可能有不同类型的员工，如销售人员、开发人员等等。所以一般Employee会建模为一个接口，而由Employee的子类来实现具体的角色。当一个客户应用需要知道一个人的工资时，它从核心去获取一个”Employee”的角色，但是现在把它作为类型的名字显然不适合，因为实际的角色可能具有的名字是”Salesman”,”Developer”等等。

这个关于Specification的问题同样出现在对角色的管理过程，如果单独使用一个类名字不能建立或者获取到一个具体的Role，那么整个管理协议都适应这个Specification进行，而不是单独把类名字当作Key进行管理。

创建过程

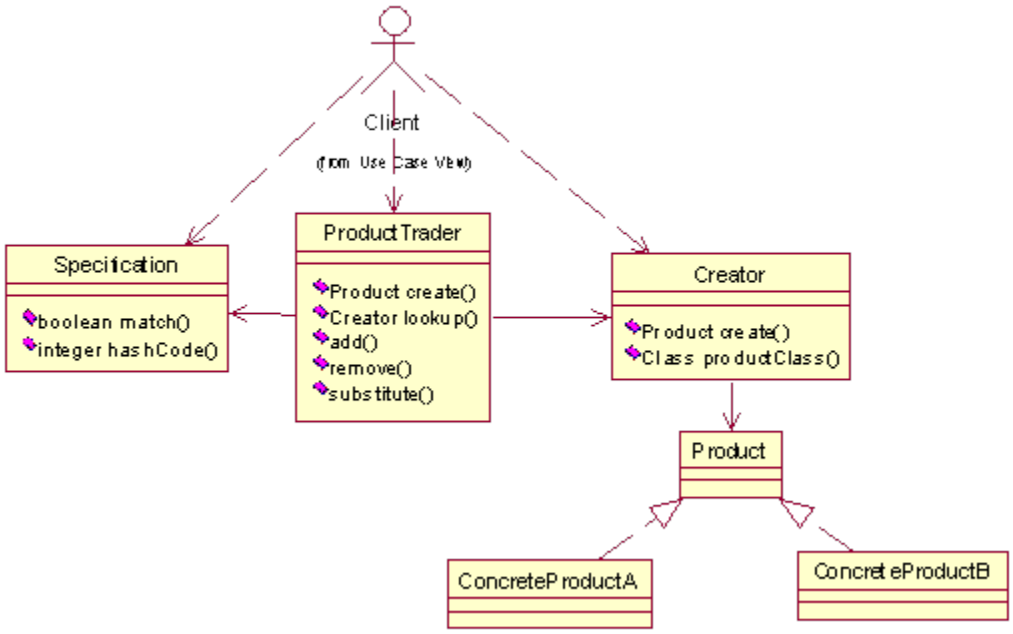
对于创建过程来说，我们所要解决的核心问题是需要一个类，当我们向这个类提出某些需求或标准的时候，这个类就会返回满足我们需求的具体对象实例，这种满足可能具有层次关系、或者是其它的关系。

所以，有两个问题，一个是如何表达这个Specification，另一个问题是由谁来做这个中间类，我们是需要重新定义一个，还是使用原来就有的ComponentRole.我们怎样能够这个中间类的使用可以让用户对这个创建过程不可见。

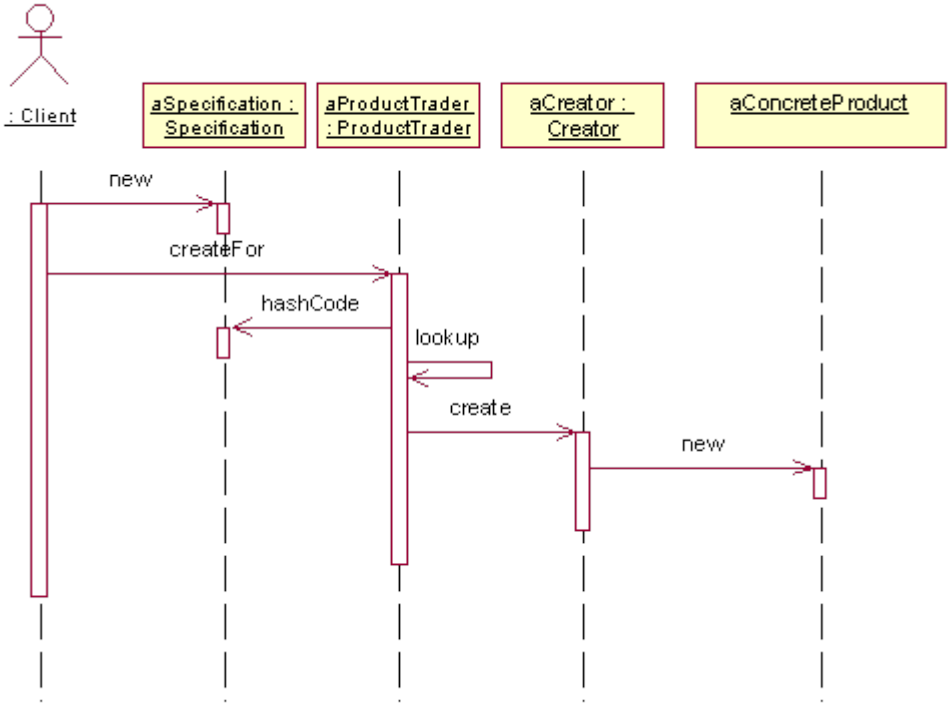
我们对设计模式的认识显然排除了那种使用一个大case的可能性，这样的代码在我们需要动态加入一个角色的时候无能为力。工厂方法看起来适合我们的基本

需求，但是你很快就发现我们上面的例子中对任何一个“Employee”，它只能返回一种类型的角色。

幸好，解决方案是存在的，我们有一个模式叫做 Product Trader,下面是它的原理图：



在上面的图中，客户负责为每一个具体的产品类建立一个 Specification,然后为 ProductTrader 提供这一



在上面的图中，客户负责为每一个具体的产品类建立一个 Specification,然后为 ProductTrader 提供这一个 Specification,Product 是一个抽象的接口，它定义了需要实例化的子类的操作。Product Trader 这个类则实现了从 Specification 到一个具体的 Creator 的映射。并且提供各种操作来管理这个映射关系。这里的映射关系可以是一个非常简单的 HashMap 或者是一个具有完备功能的中间对象 trader.现在如何创建的任务落到了 Creator 上，Creator 知道如何实例化一个具体的产品。Specification 是一个表达满足关系的抽象，我们可以看到最简单的例子是一个字符串，可能是一个层次关系或者最复杂的情况下可能是一个需要经过很多计算的公式。它的目的是作为一个关键字搜索到具体的 Creator.下面是具体的序列图：

针对我们的 Role Object,可以看到 ProductTrader 的责任由 ComponentRole 来承担，而每一个具体的 Role 则是在上图中的 ConcreteProduct.而客户实际上是 Role Object 中的 ComponentCore.

为什么使用 ProductTrader 能够使得我们的角色对象可以被更好地创建呢？

首先，ProductTrader 如同其它的创建性设计模式(Factory Method 等)，可以使得客户独立于具体的产品类。也就是使得 ComponentCore 独立于具体的 Role.应用 ProductTrader 使得 ComponentCore 完全独立于具体 Role 的类层次。ComponentCore 只需使用某一特定的 Specification 调用 ComponentRole 的

createFor 方法，由 ComponentRole 来做具体的创建。

其次，具体的角色类可以在运行时决定。ProductTrader 的应用可以让 ComponentRole 在运行时把某些范围或者条件转化为一个 Specification,然后使用这样一个 Specification 进行具体角色类的搜索。因此，你可以具有可变的运行时才决定的角色创建过程。这个优点带来的最直接的后果就是，你可以进行方便的配置，你可以任意增加、删除特定的角色，只要你通过 ComponentRole 的方法增加、替换及删除相应的 Creator 即可。这些 Creator 可以按照 Specification 搜索特定的角色类。

我们引入 ProductTrader 的一个最重要的原因是它可以让你轻松地加入具体角色的类层次。由于 ComponentCore 成功地和具体角色的类层次、类名字、层次结构和实现策略分离，所以对上述内容的改变不会影响到 ComponentCore 创建具体角色的接口和方法。

能够很好地配置以及改变具体角色类层次使得加入和删除一个具体的角色类变得什么简单，我们不需要修改任何存在的代码，只要对配置文件或脚本修改即可。

实现的考虑

在角色对象中应用 ProductTrader 需要考虑四个主要的问题，他们包括：

l 如何实现从 Specification 到 Creator 的映射

我们需要在一个 ProductTrader 中维护一个 Specification、Creator 和它们之间的一个映射关系。

l 如何实现 Creator

有很多设计模式可以用来处理这样的麻烦，我们耳熟能详的就有 prototype、类对象等等。总而言之，无论用何种方法，你都不会希望每次增加一个新的角色，都需要手工去建立一个 Creator。所以，对于 Creator 的要求就是这个类里面由一种机制可以直接创建一个角色实例.

```
class Creator {

public Creator(Specification aSpec) {

aSpecification =aSpec;

}
```

```
public Specification getSpecification() {

return aSpecification;

}

public ComponentRole create() ;

private Specification aSpec;

}


class ConcreteCreator extends Creator {

private class concreteRoleType;

public ConcreteRole(class concreteRoleType,Specification spec) {

super(spec);

this.concreteRoleType = concreteRoleType;

}

public ComponentRole create() {

return new concreteRoleType.newInstance();

}

}
```

这个Creator相当简单，注意其实在第一个抽象类Creator中，Specification的实例应当放在ComponentRole抽象类里面用于映射到某个Creator，但是我们处于管理方面的目的，在这里加入了这个成员，你可以在后面看到。

其次，要注意的是，如果使用象C++一样的template，我们可以做到类型检查，但是在这里所有的Specification是抽象的Specification,而具体的Creator生成的是ComponentRole而不是ConcreteRole.

再仔细看一下这个Creator，你会发现create过程没有参数，某些时候，你可能需要其它的创建参数，那么也许你要通过反射得到concreteRoleType这个

class 中符合你参数的构建方法，然后用你的参数进行创建，而不是直接使用 newInstance().当然，前提是在 create 方法中假如你的参数。

1 如何实现 Specification

Specification 可以使用某些原语类型来实现，如 String 代表 Class name.当然，复杂的情况需要使用一个自有的对象。如果使用一个专门的接口而并非简单的字符串，Specification 的接口看起来如下：

```
public interface Specification {

public void newFormClient(Object anObject);

public void newFromManager();

public void adaptTo();

public boolean matches(Specification spec);

}
```

这些接口需要由具体的 Specification 来实现，其中 getRoleClass 得到为该 Specification 对应的 Role Class. Matches 用于判断两个 specification 之间的匹配性。NewFromClient 则由客户使用。它可以创建一个 Specification 然后交给 ProductTrader. 而 adaptTo 方法用于把某一个具体的 specification 和该 roleClass 相对应，它由 ComponentRole 使用，ComponentRole 通过该方法直接从 roleClass 中获取到建立 Specification 所需要的信息。

Specification 中 newFromManager 可以直接缺省实现为 adaptTo 即可，matches 可以实现为相等。我喜欢使用的方法之一先定义接口，然后实现一个 abstract 类：

```
public abstract class AbstractSpecification implements {

public void newFormClient(Object anObject);

public void newFromManager() {

adaptTo();

}
```

```
public void adaptTo();

public boolean matches(Specification spec) {

getClass().equals(spec.getClass());

}

}
```

这时候，我们可以实现ComponentRoleSpecification的如下：

```
class ComponentRoleSpecification extends AbstractSpecification {

private String roleType;

//假设这里的表示方法是一个字符串roleType，你可以使用其它的如
roleClass等等

public void newFormClient(Object anObject) {

roleType = (String)anObject;

}

public void newFromManager() {

adaptTo();

}

//注意这里需要ComponentRole有一个静态的getRoleType方法

public void adaptTo() {

roleType = ComponentRole.getRoleType();

}

public boolean matches(Specification spec) {

if (!(super.match(spec))
```

```
return false;

if (!(spec instanceOf ComponentRoleSpecification))

return false;

return roleType.equals( (ComponentRoleSpecification)spec.getRoleType
());

}

public String getRoleType() {

return roleType;

}

}
```

最后的问题是ComponentRole如何建立映射，从而把一个Specification映射到一个Creator，这些方法包括addCreator(Creator),removeCreator(Creator),substitue(Creator),我们可以看到前面的Creator中包含了getSpecification这个方法，这是我们可以用如下代码：

```
addCreator(Creator creator) {

mapping.put(creator.getSpecification(), creator);

}
```

除了这些维护方法外，还需要有lookup,这个lookup仅仅就是通过一个Specification找到creator的过程，相当简单。

ProductTrader模式的应用可能走得更远，它甚至可以用于整个应用系统的所有对象的创建，对ProductTrader的深入讨论可能会超出Role角色所需要的内容，如果读者需要进一步研究该模式，请参见PLOP3.

Role角色管理

在ComponentRole中，还需要管理那些角色，最重要的操作包括：

hasRole(Spec)

getRole(Spec)

removeRole(Spec)

addRole(Spec)

我们看到，前面的match用于两个Specification之间的精确匹配，理由是我们用它来精确匹配对应的Creator。同时，我们看到，这固然解决了ComponentRole具有多层次的问题，但是却不能提供上面的这些管理接口所需要的所有操作。

回忆一下我们在动机一节提出的问题：

“假设我们有一个核心的抽象叫做Person。我们知道Person可以是员工，所以有一个叫”Employee”的角色。但是可能有不同类型的员工，如销售人员、开发人员等等。所以一般Employee会建模为一个接口，而由Employee的子类来实现具体的角色。当一个客户应用需要知道一个人的工资时，它从核心去获取一个”Employee”的角色，但是现在把它作为类型的名字显然不适合，因为实际的角色可能具有的名字是”Salesman”,”Developer”等等。“

这里的问题是我们并不明确指定一个完全的对等关系，我们需要类层次中一样的概念，如果一个Salesman继承自Employee，那么所有Salesman的对象都是Employee，所以我们需要在Specification接口中加入两个新的操作，分别是：

isSpecialCaseOf(Specification)和isGeneralizationOf(Specification).

这时两个Specification之间可以比较。Specification B是Specification A的一个特殊情况当且仅当对于任何对象X,如果X满足A,那么X就满足B.从我们的角色对象来看，如果它所可能具有的任何一个角色的Specification是用户查询的Specification的一个特殊情况，那么该角色对象就具有用户指定Specification所代表的角色。IsGeneralizationOf则正好相反。

在我们的角色对象中，每次用户通过一个addRole(Spec)加入一个角色对象，ComponentCore在自己维护的一个Map中包存了spec实例到ComponentRole的一个映射，当用户使用hasRole(spec)或getRole(sepc)进行查询时，我们可以遍历该Map，如果发现有某一个spec满足用户的参数，则可以返回。对getRole来讲，可能有两种情况，一种情况是找到一个即返回，另外一种情况则是返回所有满足Specification的角色。

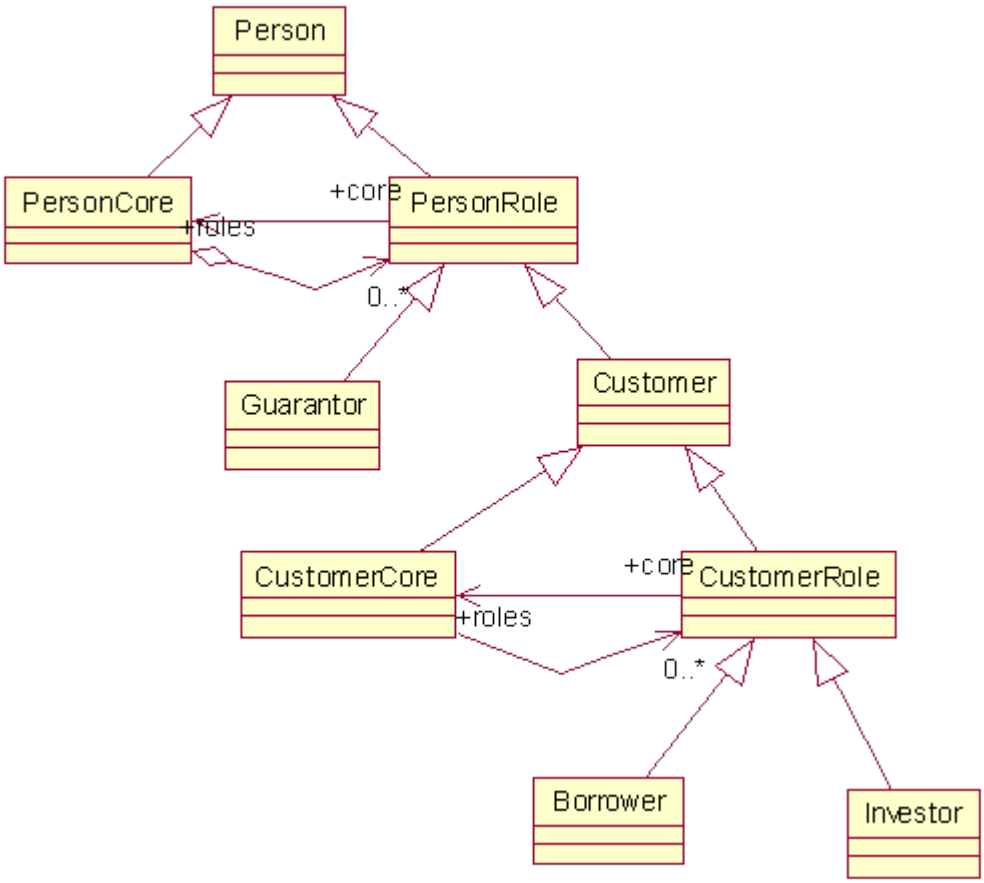
Role和Role之间的约束

我们在一开始就讲到,Role Object 很难处理具体 Role 之间具有约束的问题。这是由 Role Object 本身的特点所决定的。具体 Role 之间没有相互的关联，他们不象 Decroator 一样一个指向一个，最后指向 ComponentCore。

如果 Role 具体对象之间的关系非常复杂，我们可能需要一个知识层(knowledge level)来处理 Role 之间的关系。我们在我以后讲到 Party[待写]的时候看到，TAO BBC 如何使用知识层和 Role 组合完成一个具有 Role 的 Accountability 模式。

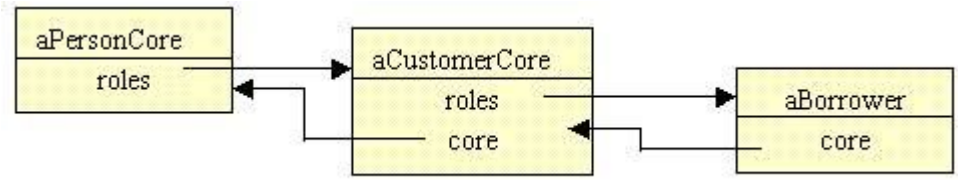
其实，我们的 Specification 也可以看作是一个知识层，对于某些约束，Role 可以通过 Specification 来解决。Specification 可以解决具体 Role 具有一个类层次的问题。那么我们可以递归使用 Role Object 模式来处理这样的约束。

在实际应用中碰到最多的约束问题可能是，某一个对象的一个角色 A 可能是该对象能够充当其它角色的一个前提，在银行业务中，如果一个人希望充当贷款人、投资者，那么它首先必须是一个客户。



你可以看到，在上图中我们重复应用了 Role Object 模式，从而限制了一个

Investor 角色首先就必须是一个 Customer 角色，从而形成如下的对象链。



Role 的属性列表

Role 可能需要动态增加属性，并且处理这些属性之间相互约束，请参见专栏中的文章动态属性（属性列表）

Role 的存储

我们在这里没有关心 Role 的实际存储，我们将 Role 设计成具体的存储无关，具体的存储方法可能需要另一个包装。譬如使用 O/R mapping,或者 EJB，我们需要实现对应的 RoleEntity 层次，然后把 RoleEntity 实现的行为分派到具体的 Role 模型。这是我们将在 TAO J2EE 模式研究中看到的课题。TAO BBC 将使用这样的模式来实现具体的存储。

本文使用 Product Trader 模式和 Specification 模式讨论了 Role 的一些具体实现和管理问题。在下一篇文章中，我将描述实现 Role 的另一种方法。新的方法将采用类 Decorator 模式来处理 Role，这种方式和 Role Object 各有自己的优缺点。

关于作者

石一楹是一个专注于面向对象领域的系统设计师。目前的工作是国内一家 ERP 公司的 CTO.他住在浙江杭州，有一个两个月的女儿。

项目经理面试指南

作者：Patricia L. Ferdinandi 著
zhoufang 译
本文选自：UMLCHINA

简介

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

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

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

什么是真正的项目管理

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

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

项目经理是项目团队的领导。他/她的职责是激励团队以积极的方式完成任务。该职位需要具有技术和人际技能，需要每天关注的内容（顺序如下）如下：

业务
公司
项目
团队
个人
技术和方法的变更

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

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

你可能已经意识到自己忽视或缺乏某些领域的知识。因此，本文的读者为：

没做过项目经理的人 ；
已经是项目经理，但认为自己的技能已经过时的人

项目经理是什么

项目经理角色

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

管理层需要什么样的人

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

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

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

人员管理技能

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

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

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

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

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

你不能管理你无法控制的东西

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

度量方法：度量方法如果没有管理好或运用好，会产生负面影响。度量方法可以作为计划编制的“输入”，可以在项目进展过程中和结束时进行统计，为下一

个项目或项目的下一个阶段提供参考。用度量方法来评估员工的绩效是不恰当的。

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

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

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

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

项目管理的奖励/压力

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

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

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

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

准备面试的方法

书、杂志、组织和研讨会

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

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

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

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

你应该了解的软件

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

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

如果你使用过此类工具，把这些内容列在你的简历中。当然，不仅要掌握工具，你还必须具有坚实的基础知识和项目管理方法。

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

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

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

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

寻找思想

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

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

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

项目计划技术

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

图表类型：

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

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

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

工作分解结构：项目任务和/或活动的结构图。

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

可交付成果：证明一个或多个任务完成的有形事物。例如：逻辑数据模型。

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

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

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

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

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

a) 功能点（Allan Abrecht）

b) 重要事件 (Tom DeMarco)

c) 加权平均

d) 代码行

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

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

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

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

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

面试中的表达的要点(就算问题没被问)

如果你没有管理经验

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

如果你的技术技巧在未来的技术的环境中是落伍或不同的

你不需要理解技术环境的内部是如何工作的，但是你应该理解一般的概念和特征决定环境的能力和弱点。许多项目管理技巧是超出技术范围的。因此,如果你的

技术技巧是落伍的，你仍然能强调你在技术上能负独立责任。提及你管理的应用类型和及其商业作用。提及团队是如何有效地完成目标的。强调你的管理哲学。提到上级，与你地位同等的人，你的用户和部下是如何评价你的管理能力的，记住提起任何你掌握的商务领域知识。在面试时应该将你对你的技能落后的恐惧抛在一旁。一旦你拥有这个工作，你将能向公司内的专家询问。在所有组织中都有各方面专家的非正式的机构。你可以到处打听一下，把他们找出来。

问面试官的问题：

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

1. 公司优先权是什么？

2. 本项目的执行资助者是谁？

3. 公司使用的开发原理体系是什么？

4. 本项目最后期限是什么？

5. 有量度项目成功的方法吗？

6. 你的新经理将怎样保持项目信息灵通？

7. 你的新经理管理哲学和风格是什么？

8. 项目上的人们的技能水平是什么？

9. 你将管理的项目的范围被充分地定义吗？

10. 技术环境已经选好了吗？

以下是典型的项目管理面试中通常会问到的问题（期望的回答）：

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

1. 项目管理软件工具知识

2. 编制项目计划的技术

3. 人员管理技能

4. 沟通技能

5. 原理体系知识（标准开发生命周期和项目管理）。

项目管理软件工具知识

问题 1：工期和工作量之间的差异是什么？

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

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

问题 2：怎样和为什么要在编制项目计划时考虑依赖关系？

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

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

问题 3：你怎样将人的工作步调与计划结合？

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

问题 4：你怎样将培训，假日和个人教育时间表结合起来？

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

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

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

问题 6：实况报告对计划的作用以及实况与最初预计的比较有何价值？

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

做项目计划的技能

问题 7：你为什么制定项目计划？

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

问题 8：你将怎样着手做项目的计划？

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

问题 9：你将怎样着手制定项目计划？

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

问题 10：怎样确定人员需求？

答案 10：不考虑资源限制进行计划开发。在任务旁边加上诸如数据模型制作者，业务分析员和用户等角色。再加上能将任务重叠起来的补充性的资源。在计划中要考虑开发团队包括支持团队和用户代表失去一个或多个资源的情况，要在每个任务上增加 15%的余量。要使项目小组的组成容易理解，要有角色所必备的技术水平的说明。

问题 11：给项目加上测量标准有什么价值？

答案 11：如果使用得当，测量标准是一个有价值的工具。它们提供测定开发系统的复杂性和工作量的方法。度量结果为制定项目计划提供了信息输入资源，并且是确定发展方向的有价值的历史信息。软件测量标准将有助于开发更好的软件。不过，最好有 3 年的历史资料。

问题 12：你怎样在计划中运用新技术？

答案 12：在增加培训任务的同时要扩大工作量，缩小每个工作单元。在评价新技术在开发中的影响的过程中加上额外的原型和检查点（里程碑）。

人员管理技能

问题 13：你作为项目经理要做的第一件事情是什么？

答案 13：除了注意公司的发展方向并从中发现自己的发展道路外，在头脑中要建立项目经理所关注事物（商务，公司，项目，团队，个人，技术和方法论的变化）的优先顺序。因此，和部门经理开会确定优先顺序，安排用户和职员会议，得到全部成员的状态报告和评价。重要的是能尽快处理业务，项目和个人有关的事情。

问题 14：当你的职员减少了 30 %你将怎样着手完成公司的项目？

答案 14：首先，确定和区分项目的优先次序，哪些项目是必须在今后的 18 个月内完成的。把绝对的最小的总人数与每个项目联系起来。向管理者和用户说明对进度表的影响。因为两者都也许不愿意接受进度表的变化，因此或许可以给你一些例外。

减掉顾问比去掉一个雇员要好。每个项目的顾问也许可以用雇员代替。坚持运用学习曲线理论并逐步减少顾问人数。可以把一些顾问的工作从一周降低到一星期中的 2 或 3 天以应付人员削减。

如果公司有提前退休的一览子法案，赶紧寻找一些有资历的、适用的雇员。牢牢记住失去“老资格的人”你也许就失去了有价值的知识。尽可能将一个快退休的人和新手组合在一起。

以满足业务目标为前提，确定剩下员工的重要性以及他们在每个项目中的重要性。使新手和经验丰富人员的比例适当。两者都是确保项目和公司不断成功的财富。

问题 15：你的团队主要是由新手组成的，并且进度已经落后。你将做什么？

答案 15：需要记住一个项目很少因为在截止时间内没有完成而被取消的。项目被取消，主要是诸如缺少资金，用户支持或不能满足的业务目标。

因此，要做的第一件事是培训，无论在室内还是室外，在课堂或通过录像带。另一种附加方法就是让资深的雇员或高级顾问充当教师。

举办针对个人评估和辅导的会议。帮助每个员工准确评价他们各自的优点和缺点。同时明确任务，将所有必须遵守的标准或准则阐述清楚。为每个员工提供从成功项目中得到的模板作为指南，还要允许他们发挥自己的才能。如果需要，和他们一起工作。对任何问题或完成的任务做出迅速的反馈。

对于较大的任务，看看他们的计划，有助于确定他们是否了解任务的范围和目标，以便了解他们是否能完成任务。倾听员工的观点，也许他们会有完成任务的正确的方法和途径。然而也要防止雇员陷入挫折和士气低落的困境中。

问题 16：你将怎样和与你竞争相同职位的员工相处？

答案 16：这是经常发生的不愉快情况。雇员总是认为他们能胜任某个职位而管理层还没有意识到这一点。因此，要进行如下调查：

l 发现员工的管理能力

l 阅读评估和状态报告

l 当雇员变得不合作时试图发现一些变通的方法并且针对这种状况进行一些个人谈话，谈话内容包括：

l 弄清楚状况

l 与员工一起分析他/她具有的能使他/她得到提升的资历

l 强调在初期协作的必要性和管理层是如何高度重视合作关系的

问题 17：在决策和工作风格方面你会给你手下多大的自由？

答案 17：自由的大小取决于每个人的技能和专业水平。一个好的经理是“面向结果的”并且能创造一个能使团队广泛交流的环境。无论如何，每个员工每周需提交项目和商业目标有关的状态报告并且经理要进行审查。这有利于加强组织建设并使每个员工致力于他们自己应完成的工作。

问题 18：如何对待即将退休的员工？

答案 18：即将退休的员工能提供大量的信息。一个人在把所有业务知识和关系

网拒之门外时必须三思而后行。因此，要利用这些人的能力：他们在某些特殊技能方面可以作为新手的老师。明确主要的工作利益，要使项目能充分利用这些技能，可以利用他们从非正规途径得到的必要支持（不用通过正规的，官僚的途径完成工作）

问题 19：对一个一贯迟到的员工你会怎么办？

答案 19：好的经理是通过结果与所花时间来评价一个员工的。然而，还需要了解迟到会在公司和团队中造成什么影响。一个人经常迟到人们会感到领导在徇私并且会影响团队的士气。这个人也许可以按期完成自己的任务但可能会影响到别人的进度。职业特性包括可靠性。如果别人的工作进度取决于他们的工作进度，那么，他们的进度对于整个团队就很重要。

首先判断这些员工的模式。换句话说，是偶尔还是一贯如此。其次，明确公司有关考勤方面的政策，确定迟到及其相关处理方法。要了解该员工的工作是否与进度相符并了解与他一起工作的人对他迟到的反应。

最后，必须与他们进行客观的谈话。

谈话的主题包括：

l 公司的规章制度

l 对团队的影响

l 对个人评价的影响

l 强调时间进度

l 达成谅解

问题 20：在费用削减的情况下，你将怎样鼓舞士气？

答案 20：钱不是仅有的激励因素。人们需要了解他们是否对项目有积极的贡献。因此，要强调拥有的自豪感并且举行业务会议，在会上让用户谈谈他们对项目组的良好印象。同时，让用户对他们的功能和业务提出一个概括。培训是一个激励因素。因此，状况会议可以作为一个非正式的培训课程。不定期地举办有关新技术的内部研讨会。如果培训课程费用太昂贵，可以租赁技术录像带。订阅杂志，有许多技术杂志是免费的。必须记住的是，忽视培训将使团队的精神低落。这样会影响产品的质量和数量。

问题 21：你如何雇人？

答案 21：首先做一个工作所需技能的描述。如果你不了解现在的需求就很难雇

到合适的人。接下来要了解团队成员的个性。列出团队现在缺乏的技能或工作风格。与人力资源部门讨论所有这些情况，包括调动现有员工。当候选人到来，针对现有工作进行面试，同时还要了解他是否具有新岗位所需的技能。

问题 22：你将如何解决团队中的个人冲突？

答案 22：辨别出人的不同个性。分别向员工表述每种风格的价值。当与冲突双方讨论试图分析申诉或冲突的原因时应持有客观的态度。

问题 23：你将如何监控/管理顾问？

答案 23：顾问也是人，也需要得到尊重。他们还需要明确的目标和任务。坚持做工作周报，将工作时间和工作完成情况联系起来。

问题 24：你将如何管理外援？

答案 24：和管理顾问的方法相同。不过，他们可能有一个经理来负责外包合作。首先要和这个经理一起组织日常会议。坚持做工作周报和可交付产品的拷贝。

问题 25：你将如何同一个似乎总是不能按时完成工作的员工一起工作？

答案 25：直到找到问题的原因时，问题才能解决。原因不一定是分析问题或解决问题的能力差。可能是一个管理方面的问题。

该员工可能没有得到适当的培训，他的工作可能超出了他的能力范围。另外一种可能是这个人有太多的事情要做而且这些事情都是最重要的或者他不清楚交付日期。

如果不是上述原因，要注意观察，找出原因所在。例如当所有人遇到问题时，都会找这个人。那么，这个人的工作经常会被无数次地打断。

沟通技巧

问题 26：你将怎样使用户参与和了解项目的每个阶段？

答案 26：贯穿整个项目的原型是得到用户肯定的方法。让用户对有形和无形的利益进行研究，以做出成本效益分析。和用户一起开发测试数据，测试大纲和验收标准。e-mail 里程碑状态报告和更新/修改的项目计划。在项目进行阶段性检查时的同时对可交付产品进行检查。

问题 27：你将如何发现和解决内部和外部问题？

答案 27：从所有可能的资源获取实情并客观地记录下来。然后在相关方参与下，尽量自己解决问题。如果这种方法无效，按照组织的管理结构提出问题并参照可

能的解决方法。

问题 28：你将如何得到供应商的一贯支持？

答案 28：虽然供应商是在管理范围之外的，但也可以将他们包含进来，如果他们：

l 得到尊重

l 了解业务目标

l 预先购买

l 将供应作为计划的输入，这样会对他们产生影响

l 参与设计

因此，在项目的早期阶段就应该考虑供应商的管理。确保他们了解业务目标和工作的利益。

问题 29：如何处理“是否能破除一些规矩”现象？

答案 29：单纯为了技术而采用某种技术是不能说服用户或领导的。任何人都可能抵制那些会改变现状的变化。然而，如果将技术与商业利润联系起来，用户会支持你的建议。

问题 30：你如何应对不同的商业用户，如果他：

a) 拒绝确认需求

b) 经常改变主意

c) 不肯花时间

d) 坚持不现实的截止日期

答案 30：无论客户有多难应付，都应该记住正因为他们我们才有工作做。他们是客户。必须以高度的职业精神，完全尊重他们。

因为他们不能了解我们的工作正如我们不能完全了解他们的那样，沟通变得比较复杂。因此，我们要花时间作规划并解释其中包含的内容。用户需要感到他们没有浪费时间，正在取得成果，并且他们的意图被很好地理解。制作原型是一个有用的工具。它提供了一幅用户能理解的、灵活的图画。

另外，对工作风格的理解也很重要。拒绝承认或不断地改变想法可能源于对问题缺乏理解，或是对未来的担心。

用户往往不愿意花时间与 IT 人员交谈并认为这样做是浪费时间，因为 IT 人员过分关注他们自己的任务。应该对过去交付产品的历史进行检查。如果用户来了多次但并未发看到有价值的输出，他们将拒绝花更多的时间。在这种情况下，你应该做你擅长的商业领域的项目以期得到用户的尊重。

召开一个历时一小时（并且要限定在该时间范围内）的需求讨论会来讨论特殊的问题。会议结束时应让用户知道下一步该怎么做（并要取得共识）。用户的观点被记录在“会谈纪要”上。这些会让用户感到他们的意见已被听取并且允许他们更改错误。

一个项目被取消往往是由于没有经济合理地达到用户的业务要求。如果在项目的整个过程中，一直保持与用户的有效沟通，他们将看到他们的要求正在逐步达到。项目很少因为延期而被取消。要注意范围变更。在原有的截止日期上增加额外的任务，将会产生不现实的截止日期。

问题 31：在一个不编程，就认为你没在工作的环境中，你如何开展工作？

答案 31：如果用户认为你了解了他们的业务目标，他们就希望早些开始编程。以一种他们能够理解的形式制作需求文档，提供一种开放的沟通方式，并让他们知道你了解什么，你正在做什么。通过项目计划，状态报告和原型同样能够表明项目的进展。通过让用户审查需求，原型和状态报告的形式，让用户参与项目。

方法论知识

问题 32：生命周期是什么，它的作用是什么？

答案 32：一个开发或维护生命周期是描述一个特定项目的开始，中间环节和完成的方法。一个生命周期包含了完成特定目标的所有步骤，任务和/或活动。每个活动可能有一种特定的方法。例如,制作数据模型可能会按照 James Martins 建模方法。对象建模可能会采用 Ivan Jacobson 方法。生命周期通过运用所有方法来完成业务目标。

问题 33：描述你的项目计划中应包括的阶段、活动和可交付产品。

答案 33：项目计划中应包括如下阶段（不是以瀑布/线性次序）：

1. 项目管理：

典型活动：很多人忘记加入诸如开发和维护项目计划，状态会议和报告，评估

的资料收集和汇报，制作演示资料和向上级和用户进行演示等诸如此类需要花时间的，内部的项目管理活动。

典型交付：项目计划，状态报告，评估报告（例如：有多少个功能点）

2. 需求分析：

典型活动：范围定义，成本利润初步分析，建议。

典型交付：范围文档，物理和逻辑分析，实体关系图，成本利润分析，商业规则申明，任务定义和概要说明。

3. 设计：

典型活动：建立开发和测试环境，制作逻辑模型，技术系统设计，执行计划。

典型交付：逻辑数据模型，事件模型，对象模型，网络模型，物理设计，适合开发环境的规格说明，经过修改的规格说明书，测试计划，流程图。

4. 开发：

典型活动：编码，单元测试和制作用户文档。

典型交付：测试说明书，过程手册，程序。

5. 测试：

典型活动：软、硬件测试，线性测试，系统测试，集成测试，回归测试和平行测试。

典型交付：测试结果，问题报告和跟踪纪录。

6. 实施和支持：

典型活动：第一阶段成果打包；培训。

典型交付：问题报告过程。

7. 检查:

典型活动：交付后的三到六个月对目标成本，开发工作，可见/不可见收益进行检查。

典型交付：实施总结报告。

问题 34：制作原型应该在项目生命周期的那个阶段？

答案 34：贯穿整个项目。眼见为实。因为它是验证功能，业务规则，用户需求数据和测试的一个好工具。值得注意的是，原型不会成为粗制滥造的产品。原型需要较好地维护。原型应能在过程和数据不完全的情况下，显示各个窗口和窗口间的导航关系。

问题 35：在项目生命周期中，基于客户端/服务器端开发与基于大型机开发的区别是什么？

答案 35：基于客户端/服务器端开发的项目需要额外的任务编制各部分的计划。各部分计划中必须包括对事件，数据和网络位置的检查。必须根据用户的要求决定服务器/客户端的分布。在服务器/客户端环境中，要运用外观建模技术和制作图形界面的原型相结合和方法。

问题 36：在一个维护项目中如何管理和保证质量？

答案 36：维护本身就含有负面意义。许多公司认为维护工作是不好的，第二位的。费钱的，并且是对现有应用的不断修改。必须懂得维护也有它的生命周期。因此，应建立一个围绕维护活动的控制和质量工作的计划。新的开发计划包括交付产品和每个任务分配的时间。项目计划应考虑到需求变更的情况。这样可以使项目经理和用户看到变更对项目进度的影响。

维护阶段/活动有：

变更的确定（是否会造成产品问题，是否增加了新的功能，或技术平台的变更）

1. 正式记录变更，

2. 变更确认并初步估计变更的大小，

3. 对现有变更进行优先级排序，

4. 变更分析，

5. 对变更进行编程，

6. 对变更和变更对系统产生的影响进行系统/回归测试，

7. 用户确认变更，

8. 产品递交，

9. 生产。

问题 37：面向对象的开发与传统的开发方法在管理技术上有什么不同？

答案 37：面向对象的项目团队人员较少，团队成员不需要有太多创意。重要的是技术和个人的角色。每个成员需在项目的不同阶段承担不同的角色。因此，每个成员必须了解他们自己的优缺点。围绕一个或多个人员的角色有：

1 设计师（系统的整体结构）

1 抽象工程师（类和类族）

1 应用工程师（完成和组装类和类之间的消息）

由于传统的开发方法，个人角色是不能互换的。软件开发是个人的努力的结果。即使是由最优秀的，最聪明的人组成的团队，如果他们不能为共同的目标而工作，那么就是最简单的项目也不能成功完成。

问题 38：你如何在处理雇员关系，项目管理，文本工作之间分配时间？

答案 38：人是最宝贵的财富，因此需要花费最多的时间。然而，项目经理必须关注事物的次序应该是：

1. 商业目标，

2. 公司的目标，

3. 项目，

4. 团队，

5. 个人，

6. 技术和方法的变化

问题 39：什么是 PM-CMM?

答案 39：人员管理能力成熟度模型。PM-CMM 和 CMM 都是卡内基.梅隆大学的软件工程研究所开发的概念模型。PM 提供了人力资源管理的组织方法。五个层次是：

1. 随意的：人员管理没有连贯性，

2. 可重复的：组织在人员管理方面有一些政策方针，

3. 明确的：将人员管理与业务特点相结合，

4. 可度量的：对人员管理可进行目标量化，

5. 优化：有组织地致力于不断地提高人员管理水平。

小结

一个成功的团队是指由不同技能、才华、工作风格和知识的成员组成的士气高涨的团队。项目经理的职责就是将这些人组成团队并激励他们。本文通过复习一般性的概念、术语和面试中经常会问到的问题，为面试做准备。你可以根据你有关如何成为一个好的项目经理的知识和经验，对答案进行整理。不管怎么回答，尽量给你所应聘的组织留下印象。应以一种积极的态度面对。应侧重于人员管理，同时还有一个良好的技术背景。应具备应有的常识、自信、倾听和作决定的能力。

动态属性(一)

几乎我们碰到的所有对象都有属性，通常我们会在设计时(Design Time)在一个类里面对属性进行声明，这叫做静态属性（Fixed Property）。绝大多数情况下这样就足够了，但是如果需要大量的属性，并且需要频繁的改动，并且这些改动可能发生在运行时(Run Time)的话，静态属性就不能满足要求了。这迫使我们使用多种类型的动态属性。所有的动态属性都有这样一个共性：用一个带参数的查询方法获取属性。

静态属性

静态属性最关键的是他在设计时就被固定了，并且在运行时它的所用实例都必须遵循这个决定。在一些情况下，这是一个麻烦的限制。假如我们要设计一个通讯录，其中有些属性是固定的，比如家庭地址，家庭和工作电话，Email。但每个实例都稍有不同，有人需要记录父母的地址，有人可能有兼职工作的电话。要想把所有情况都预料到几乎是不可能的。如果修改程序的话，就必须经过编译，测试，发布这么一整套流程。在这种情况下就需要使用动态属性。

我们将要讨论动态属性的好几个变种，他们的差异在于对灵活性和安全性的不同权衡。

Flexible Dynamic Property 是最简单的一种模式。

---

4 Flexible Dynamic Property

提供一个以 String 为参数的 Attribute 仅仅用 String 就能定义一个属性。

这个模式的本质是：在 Person 上增加一个 Qualified 关联，关键字(Key)是一个简单值，通常是 String。如果要为 person Kent 增加一个 vacation address 属性，只需用 code list3 的代码，不必重新编译 Person 类，甚至可以从 GUI 或者文件读取要增加的属性，这样连客户代码也不用重编译了。

```
class person {

    public object getValueOf( String key );

    public void setValueOf( String key, object value );

}
```

---

code list2

```
kent.setValueof( “VacationAddress”,  anAddress);

Address kentVacationAddress = ( Address ) kent.getValueOf( “vacationAddress” )
```

---

code list3

既然 Dynamic Property 具有这样的灵活性，那么我们为什么还要使用 Fixed Peoperty 呢？其实灵活性是有代价的, Dynamic Property 牺牲了软件各部分之间依赖关系的清晰性。你可以非常方便的为一个 Person 对象加上 vacation address 属性，但以后我怎么得到它呢？如果使用 Fixed Property 只需要察看 Person 类的接口，就能知道它的所有属性。并且编译器会帮你检查哪些能做哪些不能做。而使用了 Dynamic Property 将失去设计时检查，Person 的接口更加难以确定，不仅要看在 Person 类里面定义了哪些接口，还要到代码里去察看加入了哪些 Dynamic Property(通常它们根本就不在 Person 类里面)，把它们发掘出来。

不光是属性难以查找，它使得依赖关系简直像噩梦一般。当我们使用 Fixed Property 的时候，客户代码 仅仅依赖于 Person 类，这种关系很容易跟踪。比如你改变了属性的名字，编译器会告诉你哪些代码需要修正。但是 Dynamic Property 会产生对任意某段代码的依赖性。它可能会是客户根本就不可见的某个类里的一段代码。如果有人改变了关键字字符串（Key String）会怎么样？如果有人改变了关键字字符串对应对象的类型又会怎么样？现在编译器帮不上你的忙，你甚至不清楚这些潜在的改变该从何找起。

这些问题在 Flexible Dynamic Property 模式中表现得最严重。属性可以在设计时由 Person 的任何客户代码来创建。如果其他的客户代码使用了这个属性，那么这两个客户类之间产生了依赖关系，并且这种依赖很难被发现。属性还可以在运行时通过 GUI 或从文件添加。我们根本不可能在运行时确定哪些是 Person 合法的属性，当然我们可以向 Person 询问是否有一个属性叫做 vacation address，假设不存在这个属性，那么到底是说明 Person 没有表达 vacation address 的属性，还是说明 Person 没有一个叫 vacation address 的属性？即使这个属性现在不存在，也不能说明过了一会这个属性是否会出现。

Flexible Dynamic Property 还有一个关键的缺点，要把属性由存储数据改变为操作很困难。对象的封装的一大好处就是，使用属性的客户根本不知道属性是对象中存储的数据，还是由某个方法计算得到。这是一种面向对象的重要方法。当存在子类化的时候，可以在超类中存储数据，而在子类里进行计算，反之亦然。当你使用的是动态属性，要将存储数据改为计算值，只能在动态属性的通用获取方法中加入代码（Code List4）。这样的代码显得很脆弱，而且难于维护。

_

```
class Person {

    public Object getValueOf(String key) {

        if (key = “vacationAddress”) then return calculatedVacationAddress();

        if (key = ”vacationPhone”) then return getVacationPhone();
```

```
    // else return stored data
```

–

动态属性的改进形式将帮助你解决以上问题，当然不可能是全部问题。动态属性最本质的缺点在于降低了接口的清晰性，缺少设计时检查。不同动态属性提供了不同的运行时检查能力，唯一的问题是在运行时提供的接口清不清晰，能提供多少运行时检查。对于 Flexible Dynamic Property 这两项都达不到要求。

动态属性经常用在数据库相关的应用中，因为改变数据库的定义是一件很痛苦的的事情，特别是那些涉及到大量的数据迁移的情况。像 Corba 这样的分布式对象接口，因为类似的原因也会用到动态属性，大量的远程客户使用这些接口，动态的改变在所难免。在这来两种情况下，设计时和运行时没有明显的区分。

带定义的动态属性（Defined Dynamic Property）

提供一个以某一对象作为参数的属性，要定义一个属性就要创建一个新的对象实例。

为了增加运行时的检查，第一步就是使用带定义的动态属性（Defined Dynamic Property）。它与 Flexible Dynamic Property 的本质区别在于，动态属性的关键字不再是任意的字符，而是一个类的实例。(Figure 3)

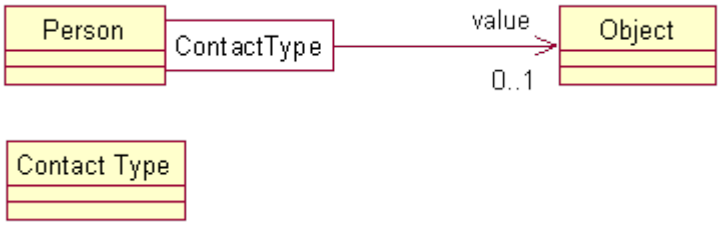


Figure3　Defined Dynamic Property

从表面上看使用 Defined Dynamic Property 并没有多大变化，事实上接口几乎是一样的。（Code List5 Code List6）只不过关键字不再是任意的，而是受到 Contact Type 实例的约束。我们仍然能够在运行时添加属性，只不过要创建一个新的 Contact Type 实例。现在至少有一个地方可以得到可用的关键字的列表，而不用遍历整个程序代码。

现在你可以建立一些检查来避免读取不存在的属性这样的错误(Code List7),我会抛出未检查错误(unchecked exception),因为我认 get()方法的先决条件(preconditon).应该是客户提供合法的 Contact Type 名字。客户可以通过使用 hasInstanceNamed（）方法进行先决条件检查。很多时候客户代码会坚持使用 Contact Type 对象，而不是字符串。

通常 Contact Type 会被放到一个字典里，这个字典一般用字符作为索引。这个字典可以是 Contact Type 的静态字段（Static Field）。

```
class ContactType {

    public static ContactType get(String name) {

        if (! hasInstanceNsmed(name)) then throw new IllegalArgumentException(“no”);

        // return Contact Type

    }
```

CodeList 7 Checking use legal Contact Type

–

Defined Dynamic Property 已经能为我们提供多得多的关于属性的信息，但它仍然是无类型的，我们并不能强制 vacation address 属性中一定是存入 Address 类型的数据。Typed Dynamic Property 在这个方面做了改进。

–

带类型的动态属性（Typed Dynamic Property）

提供一个以某一对象作为参数的属性，要定义一个属性就要创建一个新的对象实例,并且指定属性的值类型。

–

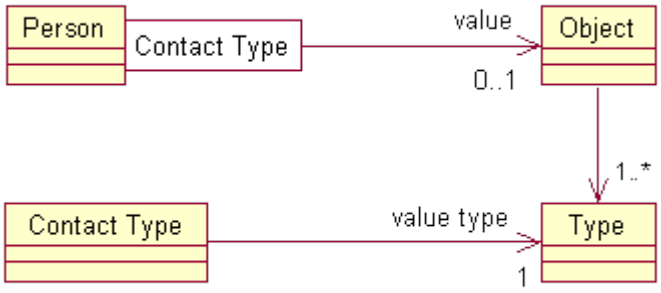


Figure 5 Model for typed dynamic propery using qualified association

Typed Dynamic Property 在 Defined Dynamic Property 的基础上添加了属性的类型信息（figure 5，figure6）现在 Contact Type 的实例不仅指明 Person 对象理有哪些属性，而且指明这些属性的类型

---

class Person {

public Object getValueOf(ContactType key);

```
    public void setValueOf(ContactType key, Object value)
```

```
class ContactType {

    public Class getValueType();

    public ContactType(String name, Class valueType);
```

Code List 8

```
class Person {

    public getValueOf(ContactType key, Object value) {

        if (! key.getValueType.isInstance(value))

          throw new IllegalAgumentException("Type Error");

        // set the value
```

Code List 9

做这样的类型检查可以帮助我们避免错误，但还是不如静态属性清晰。这些检查是在运行时，而不是在设计时，这样就降低了有效性。无论如何比起完全没有类型检查要好得多了，特别当我们是在一个强类型的环境里。

待续：

1. reflect
2. seperate property
3. dynamic property with multi-valued associations
4. typed relationship
5. knowledge level in dynamic property

动态属性（二）

动态属性与反射（reflect）

当我们深入研究动态属性，会发现丰富的关于反射的例子，反射是一种在运行时能够描述自身对象的体系模式。即使语言本身不支持反射，动态属性也提供了某种反射能力。

分离的动态属性属性（Separate Property）

为每个属性创建一个独立的对象，把属性中的内容作为这个对象的属性。

对于动态属性来说，Separate Property 和 Qualified Associations 经常是两个可用的选择。所以我在描述 Flexible 和 Defined Property 时使用 Qualified Associations，因为它提供了一个易用的接口。当然如果你愿意，也可以使用 Separate Property。当我们碰到更复杂的 Typed Property 时，Separate Property 逐渐显示出它的优势。

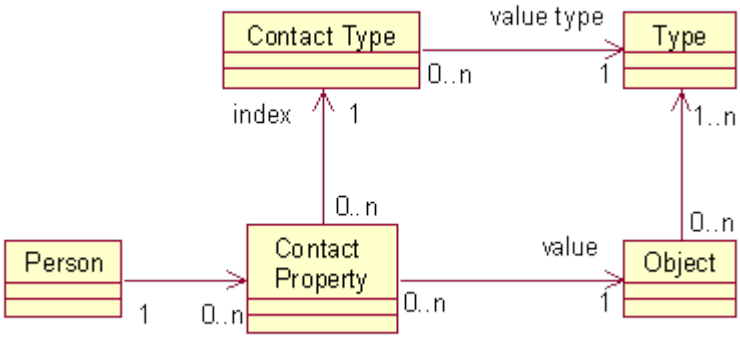


Figure 6

```
class Person  {

   public Enumeration getPropertys();

}

class ContactProperty  {

   public Object getValue();

   public Class getType();

   public ContactType getIndex();

}
```

Codelist 10

Separate Property 和 Qualified Associations 并不是相互排斥的。你可以很容易的同时提供这两个接口，这样你可以获得这两者的优点。当然这样会使得接口比较复杂，所以应该首先考虑 Person 类的客户到底需要什么，没必要使接口过分复杂，提供他们需要的接口就可以了。通常使用 Qualified Associations 接口获取属性值，但是这样有时不能很好的表达类型信息。

在这里可以考虑一下接口和实现之间的差别。在本篇文章里我们的讨论集中在概念层，它对应于软件的接口而不是实现。值得一提的是可以提供 Qualified Associatisons 接口，而使用 Separate property 作为实现。如果 Person 类的客户需要获得 Contact Property 对象，那就必须使用 Separate Property 接口。我们经常需要隐藏 Separate Property 来简化接口。

Separate Property 的一个重要优点在于它允许你将有关属性的信息放在属性对象里。

多值关联的动态属性（Dynamic Property whit Multi-valued associations）

以上的例子都集中于一个关键字对应单个值的情况，但是一个属性对应多个条目的情况也是常见的。比如人的朋友属性就应该是多值的。有两种方法解决这个问题，不过一种简单但不能完全满足要求，而另一种能满足要求但过于复杂。

简单的解决方法只需要允许动态属性的值是一个集合就可以了。我们无需更改接口就能像对待任何其他属性一样的使用（List11）。这样又简单又好，不需要对基础的动态属性模式做任何变动。（这个例子虽然是一个 Typed Property，但这种方法其实是普适的）之所以说这种方法不能满足要求，是因为它不符合我们处理多值属性的方式。如果 Friends 属性是一个静态属性，我们更希望是 CodeList12 这样的接口。因为在这种情况下我们不愿将集合暴露出来。这样处理的话，Person 类会失去在元素添加或删除时做出反应的能力，也会失去改变集合类型的能力。

_

Person aPerson = new Person();

ContactType friends = new ContactType(“Friends”, Class.forName(“Vector”));

Person martin = new Person(“Martin”);

martin.setValueOf(“Friends”, new Vector());

Person kent = new Person(“Kent”);

martin.getValueOf(“Friends”).addElement(“Kent”);

Enumeration friends = martin.getValueOf(“Friends”).elements();

_

Codelist 11 Using a conllection value in a typed dynamic property

_

```
class Person {
   public enumeration getFriends（）；
   public void addFriend（Person arg）；
   public void removeFriend（Person arg）；
}
```

_

CodeList 12

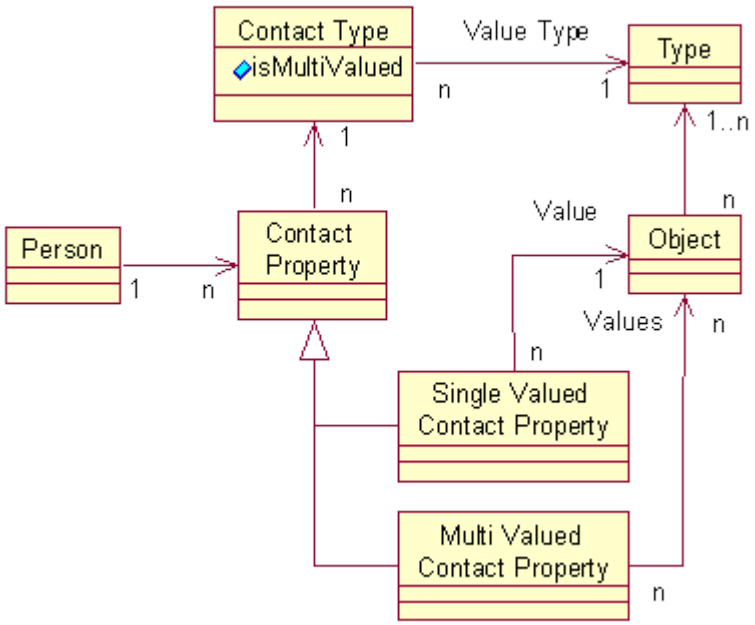


Figure 7

在使用动态属性时，为了得到 CodeList 12 所示的接口，Figure 7 演示了这样一个模型。它比较复杂，通过一些编码技巧我们把这些复杂的东西隐藏到接口背后去了（CodeList 13，CodeList 14），并且使它很易用（CodeList 15）。不过客户还是必须知道哪些属性是单值的，

哪些属性是多值的，而且关于使用是否正确的检查都发生运行期。这种复杂性导致使用它比使用静态属性要痛苦得多。

---

class Person {

public Object getValueof（ContactType key）；

public Enumeration getValuesOf（ContactType key）；

public void setValueOf（ContactType key，Object newValue）；

public void addValueTo（ContactType key，Object newValue）；

```
    public void removeValueFrom（ContactType key，Object newValue）；
}
class ContactType {
    public Class getType（）；
    public boolean isMultiValued（）；
    public boolean isSingleValued（）；
    public ContactType（String name，Class valueType，boolean isMultiValued）；
}
```

CodeList 13

```
class Person {
    public getValueOf (ContactType key) {
        if (key.isMultiValued()) then
            throw illegalArgumentException(“should use getValueOf”)
        //return the value
```

```
    }

    public void addValueTo (ContactType key, Object newValue) {

        if (key.isSingleValued()) then

            throw illegalArgumentException("should use setValueOf");

        if (! key.getType().isInstance (newValue)) then

            throw illegalArgumentException("should use setValueOf");

        //add new value to the colllection

    }

}
```

CodeList 14

```
fax = new ContactType ("fax", Class.forName (FaxNumber), false);

Person martin = new Person ("martin");

martin.setValueOf ("fax", new FaxNumber("1234 5678"));

String martinFax = martin.getValueOf ("fax");
```

```
friends = new ContactType (“friends”, Class.forName (“Friend”), true);

martin.addValueTo (“friends”, new Person (“kent”);
```

CodeList 15

这种复杂性来源于多值属性和单值属性同时使用，为处理这种情况必然有两个不同的接口。当然我们会遇到只有多值属性的情况，带类型的关系(Typed Relationship)模式是这种场景下的一个通用模式（Figure 8）。一个人可以在多个公司存在多个不同的雇用关系(甚至可以合同一个公司建立多个雇用关系)。

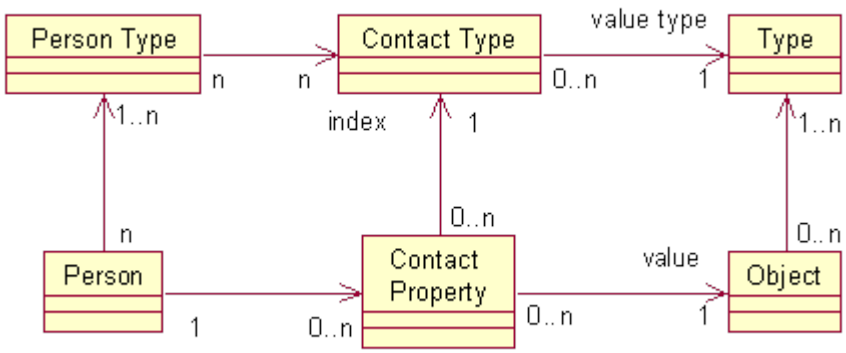


Figure 8

```
class Employment {

   public Employment (Person person, Company company, EmploymentType type);

   public void terminate ();
```

```
}

class Person {

   public Enumeration getEmployments ();

   public void addEmployment(Company company, EmploymentType type);
```

CodeList 16

当我们看到 Typed Relationship 的时候，很快会发现它就象在使用一个多值的 Defined Dynamic Property，只不过是用 Separate Property 而不是 Qualified property 进行表达。Figure 9 表达了在这种场景下怎么样使用 Defined Dynamic Property 接口。所以说 Typed Relationship 没有对这个模式语言增加任何新东西。Typed Relationship 在缄默过程中是一种通用的模式，但是大部分人没有发现它与动态属性模式之间的联系。

带类型的关系(Typed Relationship)

在两个对象间创建一个关系对象将他们联系起来，并且为关系对象建立类型对象来描述这种关系的含义。（这个类型对象就是一个多值的属性）

Typed Relationship 的强项是处理双向的关系，它提供了一个为关系增加属性的简单方法。你可以为这个模式增加一个知识层,代码与 Typed Dynamic Property 中的很相近.Typed Relationship 要求客户必须知道 Employment 对象,这就象任何使用 Separate Property 的情况.其实人们习惯于把属性对象看作一个完满的对象,就像那些 Person 或 Company 中的属性。但是 Quanlified Association 常常能提供一个较简单的接口。所以当我们看到或考虑使用 Typed Relationship 的时候，就应当同时想到 Quanlified Association 模式。

这两种模型其实并不完全相同。如果你使用 Figure 9 的模式，就表明对于特定的 Employment Type 某个 Employer 只能做一次 Employer。而如果你使用 Figure 8 的模式，就不会有这样的限制，虽然如果 Employment 没有附加信息的话，这个限制一直隐含的存在。通常 Employment 都会有附加信息，最常见的例子就是日期范围的附加信息。

Figure 9

```
public Person {

   public void addEmployer（Company company， EmploymentType type）；

   public Enumeration getEmployerOfType（ EmploymentType type）；
```

CodeList 17 interface for Figure 9

到现在为止我们一直假设只有一种类型的人，我们为一个人定义的任何属性对所有人也是有效的。但是肯定有这样的情况：存在不同类型的人，而且不同类型的人有不同的属性。比如说，主管需要一个管辖部门的属性，行政人员需要一个房间钥匙号的属性。

5 动 态 属 性 知 识 层 Dynamic Property Knowledge level

建立一个知识层，是为了放置不同类型的对象和不同类型的属性间的规则。

为了应用动态属性知识层（Dynamic Property Knowledge Level）模式，我们为 Person 对象添加一个 Person Type 类型对象。然后我们就可以这样认为：Person Type 到 Contact Type 的关联为我们指明一个 Person 有什么样的 Person Type，那么就能确定哪些属性对它是可用的。

Person Type 对象可以用来检查属性的使用是否正确。

Figure 11 Dynamic Property Knowledge Level

```
class Person {
   public Object getValueOf（ContactProperty key）；
   public boolean hasProperty（ContactProperty key）；
   public void setValueOf（ContactProperty key, Object newValue）；
class PersonType {
   public boolean hasProperty（ContactProperty key）；
   public Enumeration getProperties（）；
```

CodeList 18

```
class Person {
   public Object getValueOf（ContactProperty key） {
      if（！hasProperty（））
```

```
        throw illegalArgumentException（ ”  innapropriate key ”）；

    // return value
```

CodeList 19

当我们开始象这样使用知识层的时候，Separate Property 变得越来越重要。这种情况下我们将不再把它看成是属性，而是一些自治的对象。到底是属性还是对象的界限是模糊的，完全取决于你对事物的看法。

关于动态属性的一些总结

这篇文章我们讨论了不同种类的动态属性。但是我还是要反复强调：如果有可能的话应该尽量避免使用动态属性。动态属性又非常多的缺陷：缺乏一个清晰的接口，如果属性不是存储数据而是执行操作会相当的困难。当我们别无选择而使用动态属性时，这篇文章会变得有用：提供一些可供选择的方案，并比较他们之间的优劣。

动态属性主要出现在那些接口变化很困难的情况下。比如在处理分布式对象时，允许在不与客户协调的情况下改变接口----在一个分布式系统里要发现谁是你的客户相当的困难。

任何时候只要你向动态属性的关键字集里添加了新的关键字，你就立刻改变了接口。所有的动态属性要做的就是用运行时的检查代替编译时的检查。你还会碰到如何是你的客户及时更新的问题。

动态属性的另一个通常的应用使一些数据库的问题。这中应用不仅来源于接口的问题，而且来源于数据的迁移。改变数据库的结构不仅会影响到与数据库相关的程序，而且会引起复杂的数据转换工作。使用动态属性允许你改变对象的而不改变数据库结构，从而避免了数据转换工作。在大型数据库系统中，这种做法有很大的优点。

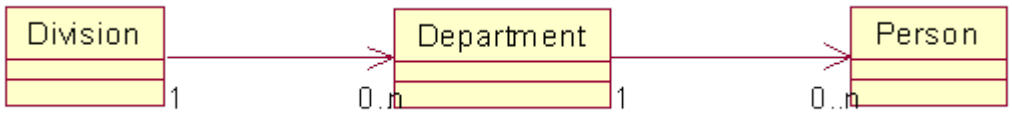
Party 分析模型、设计和实现考虑（一）

Martin Fowler《分析模式》

6 介绍

6.1 简单模型

几乎在每个业务系统中都需要管理组织机构相关的问题，如果举一个简单的例子，你的公司可能有几个部门，每个部门下面有员工。因此，你可以简单建模如下:



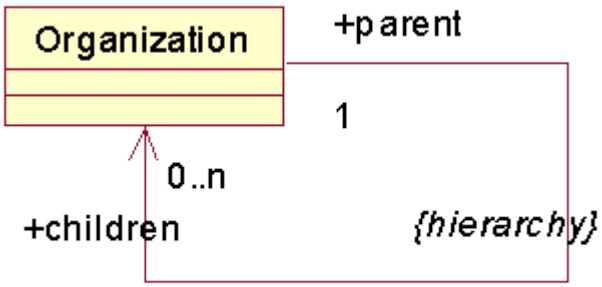
这样的模型虽然简单直接，但有两个主要缺陷：

1. 组织之间的共性没有抽象出来

2. 在模型中固定了组织的分类，如果公司增长或变化，你就不得不改变此模型.譬如你增加了一个子公司，既需要单独建立一个子公司类，以及 Division 和 Department 以及新加入的类和它们之间的关系。

6.2 组织机构层次

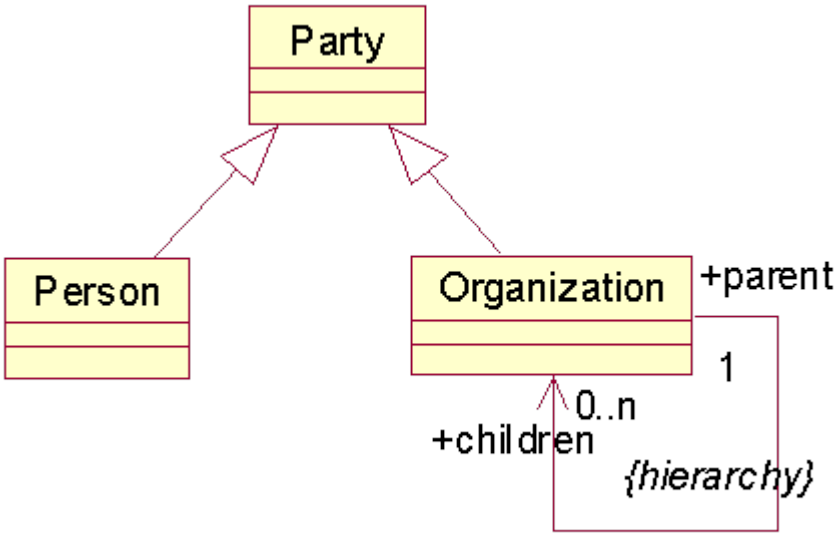
因此，我们会抽取一个 Organization 的超类，模型变为：



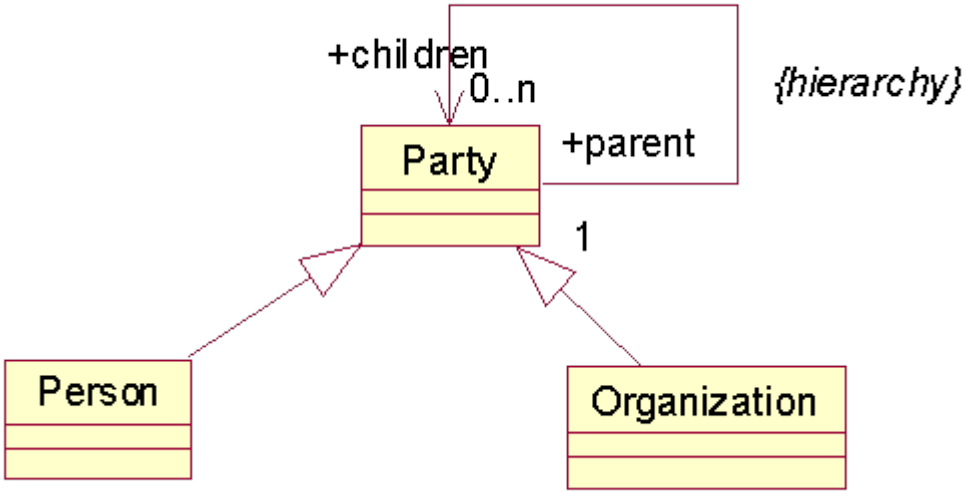
Organization 特别适合于机构之间差异不大的情况，加入新的机构几乎不用对模型作出变化。如果确实机构之间有较大的差异，你可以继承相应的子类。

6.3 Party

事实上，一个并不非常明显，但顺理成章的事情就是把 Person 加入到这个模型中。查阅一下你的通讯簿，你很可能发现不但记着人的名字，上面还有公司的条目。很多单位都有一种名片称为单位名片。显然，Organization 和 Person 之间具有共同的特性，如他们有名称，联系地址,email,电话，等等。在 PLOP1 里面有一篇文章是 Bell 实验室的一位设计人员，他告诉我们 Bell 实验室如何因为没有抽象这种特性而导致系统升级的极端困难：



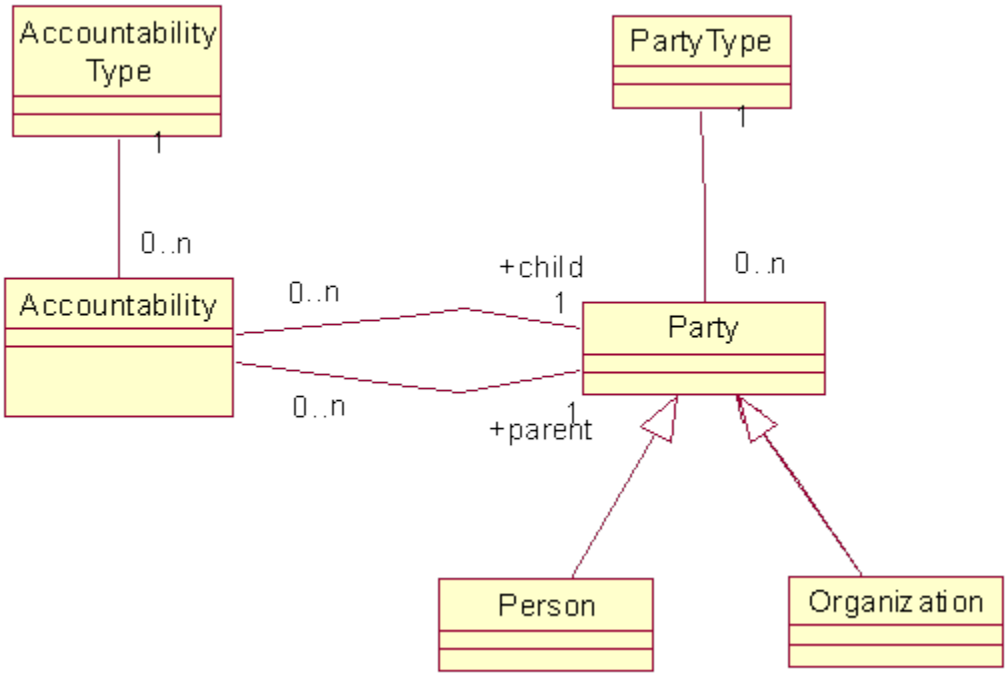
仔细分析这个模型的结果，我们发现，组织与组织之间的关系和组织与个人之间的关系没有太大的不同，因此，层次关系可以上推。



6.4 Accountability

一般的组织图可以用上面这种方式得到很好的体现。但我们经常需要处理矩阵型的组织机构。如 XX 集团公司浙江分公司销售部从功能上属于集团公司销售部管理，从地理上又属于浙江分公司管理。一个明显地解决方案是再建立一个组织机构层次，但是这样的方式同时使得你的模型难以管理和扩充。

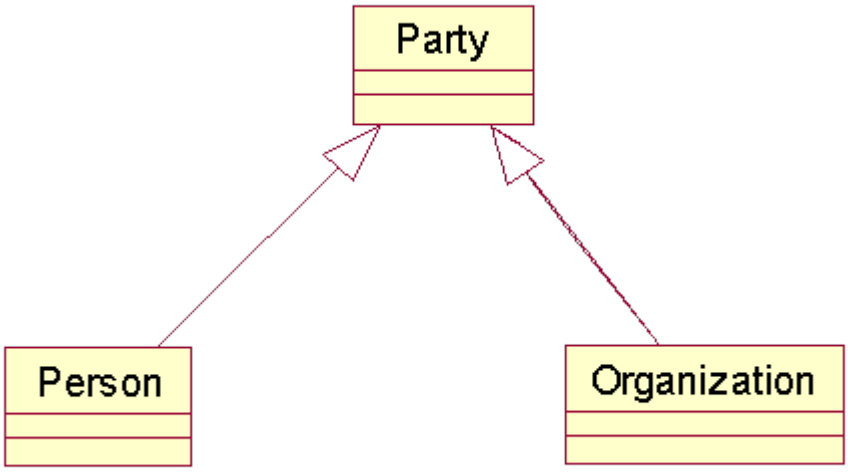
这就导致了我们的 Accountability，Accountability 的基本想法就是让对象之间的关系本身用对象来表示，事实上，这就是所谓的超对象，而更好的建模术语就是知识层对象：



7 Party

在一个业务系统中，你都需要处理人和组织的信息。简单而明显的方法是把他们独立不相干地进行处理。但是很多时候你都需要对他们进行同等的处理。譬如，你都需要维护他们的电话、地址、email,最重要的是，他们具有某些相同的行为，他们可以以相同的方式赋予权限和责任等等。

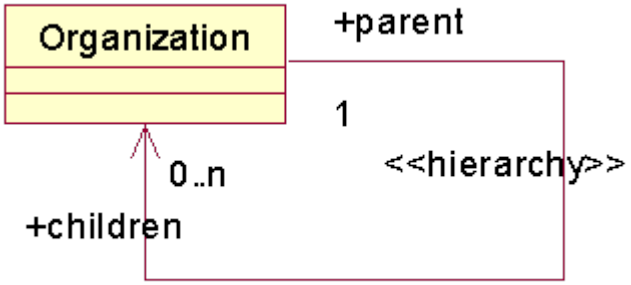
这导致了著名的 Party 模式：



Party 是对人和组织的一个抽象。

8 组织层次

组织层次是一种非常自然的组织机构之间的管理关系的反映。



人们很自然地用这样的层次关系来表达一个公司内部的组织层次。这种层次关系是可以递归的。

但是你还需要考虑到其他问题：

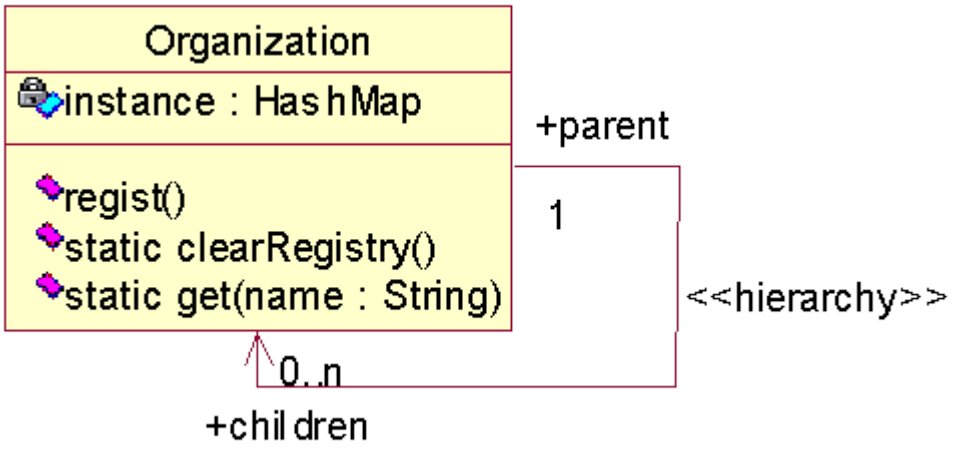
1. 通常，每个组织都有一个父，但是顶部那个组织该如何处理

2. 你还要考虑组织机构维护的循环递归问题，一个组织不能是它的父的父(不能成环)。

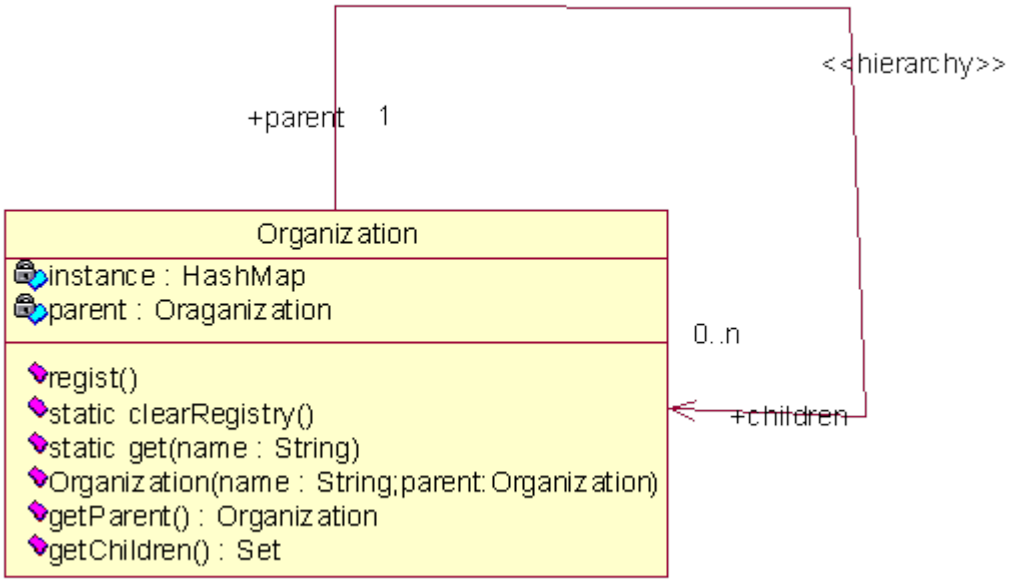
8.1 设计

上面的概念模型很容易让我们看到 Composite 模式的应用：

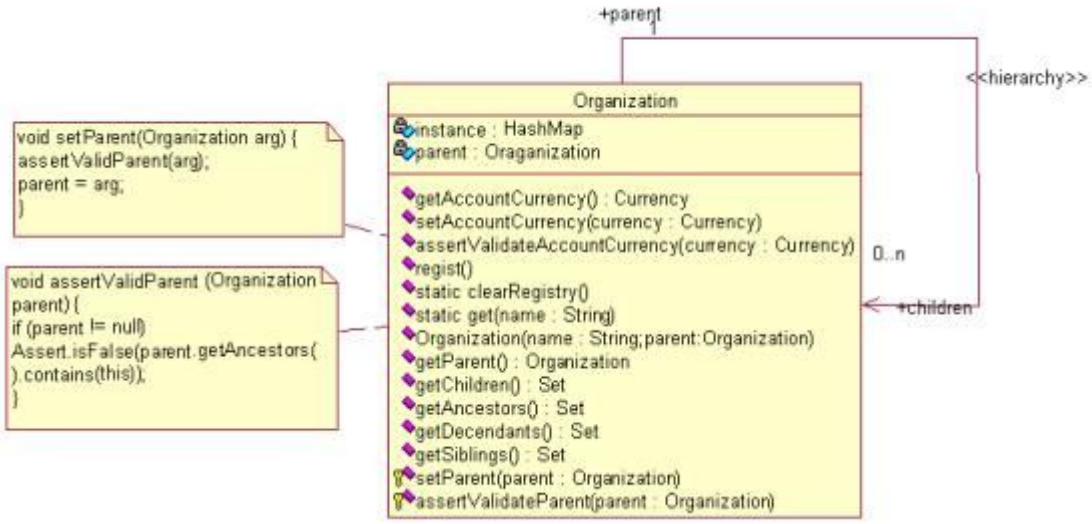
1. 为了能够直接按名字查找到组织，我们使用一个 Registry,通过内部的一个 HashMap 来实现。



2. 使用一个私有变量 parent 来放置父，创建方法可以把自己设给某一个父，自己的子可以通过查询注册表得到：

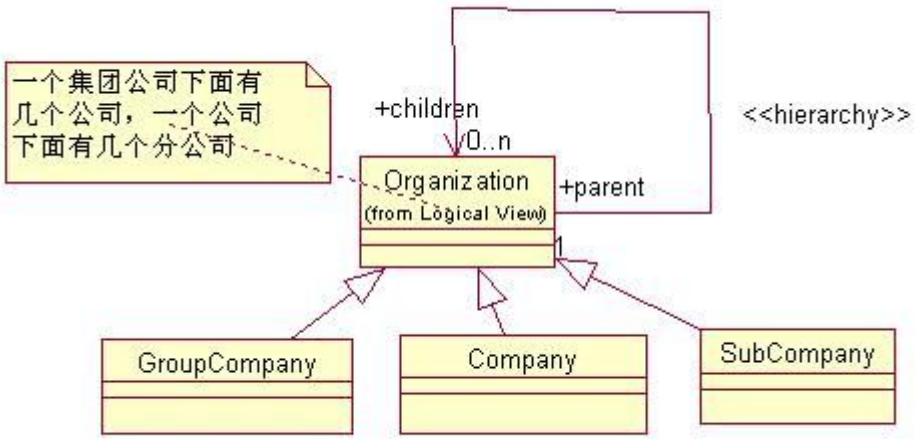


3. 其他需要实现的方法因此可以加入，你需要实现的应当对循环递归成环的校验。



9 变种：级别的处理

如果组织层次关系之间没有特别特别的限制，上面的模型就可以处理，但通常这种限制是存在的。譬如，一个集团公司下面有几个公司，一个公司下面有几个分公司，你可能需要建立组织机构的不同子类：



9.1 实现考虑

设计考虑的问题是把级别关系放在哪里，可以有两种方法，一种是把级别关系放在Organization父类中，另一种方法是把这种关系放入特定的子类中，这里的选择取决于你特别地使用情况，如果你最常用的是在组织层次之间进行导航，那么放在超类中可能更方便，但显式的层次维护成为一个主要问题时，你可以把这种操作放在子类中，你也可以混合两者使用。

1．运行时校验

你可以对

```
class Company extends Organization ...

void assertValidParent (Organization parent) {

        Assert.isTrue(parent instanceof GroupCompany);

        super.assertValidParent(parent);
```

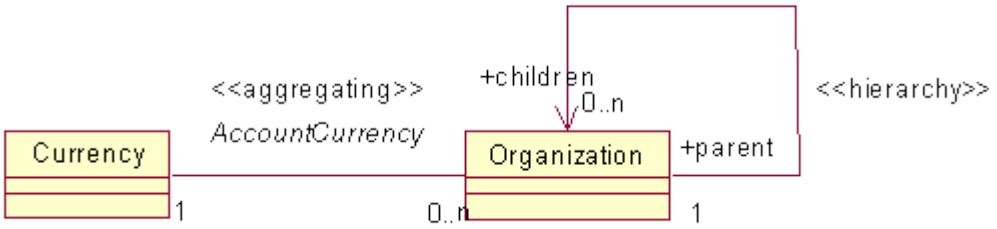
2.编译时校验

```
class Company extends Organization

Division (String name, GroupCompany parent) {

super(name, parent);

}
```

10聚集属性

某些时候你会发现，在一个组织机构内部，某些属性值从上倒下得以共享，譬如，XXX鞋业公司的本位币同时也是他公司、公司分公司的本位币。这决不是巧合的，事实上，这说明除非特别指定（覆盖），子组织服从父组织的属性约束。

这个问题虽然可以用这样的模型来表达：

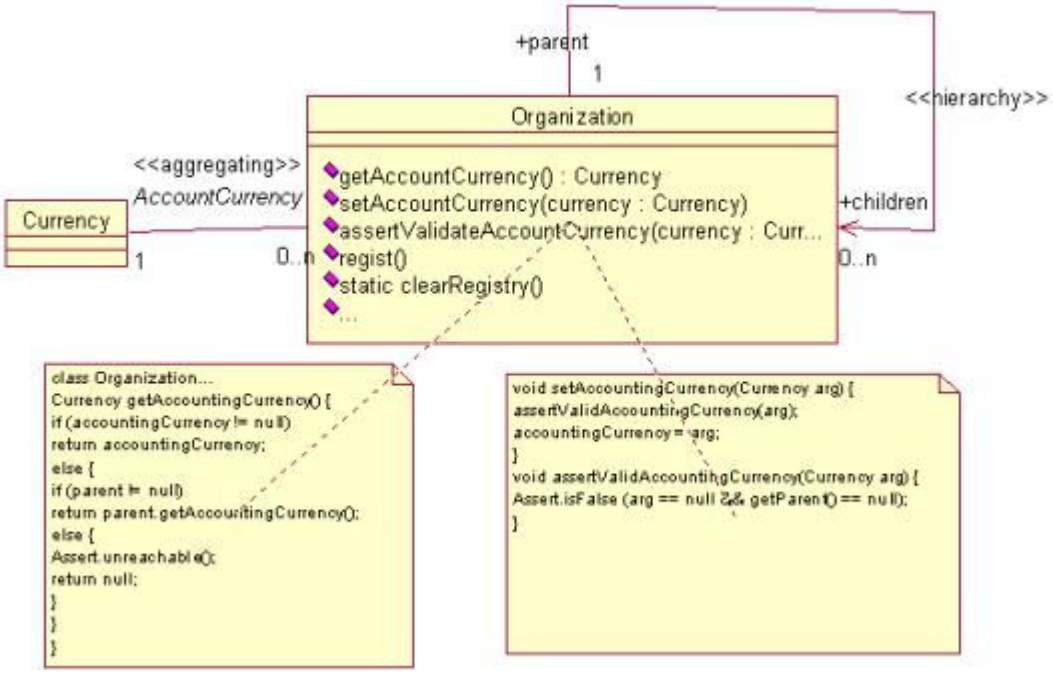


这里,<>这个 stereotype 也没有标准的 UML 定义，它的含义如下：这里 Organization 端的 n 表示多个组织可以使用同一个本位币，0 表示如果一个组织如果没有 Currency(属性为 Null，则求助于父组织)。

聚集属性的问题可能需要进一步思考，如果父组织的属性值发生变化也同时适用于子组织，那么这是一个聚集属性。但我在 BPCS 这一套 ERP 系统中经常看到模板属性，他的含义是父组织通常带有一套模板属性，子组织一旦得到这些属性以后，父组织的所有变化都与他无关，那么这可能不是聚集属性。BPCS 的做法是，如果子组织没有，那么自动从父组织查询得到，如果有的话，那么与父组织无关，也许这可以叫做模板属性，有点类似于操作系统的 Copy-on-write.

聚集属性的处理还要考虑的另外一个问题就是当一个子组织有多个父时，他到底继承哪一个父的聚集属性。我们可以通过指定特别的属性—父属性映射关系来解决这个问题，但显然有点复杂了。

10.1设计和实现



getAccountingCurrency 可以动态查询得到聚集属性的值，最后如果 parent 已经为空了，而你还是找不到一个真正的属性值的话，那么就是有问题了。

这个实现有一个效率上的问题，也就是每次你需要得到一个属性需要一直找到组织共享属性的根部，折衷的方案是在子类中保存属性值的缓存，但你不能改变外部行为。也就是，如果你对父类的聚集属性进行改变的话，你必须考虑到同时对所有的缓存进行无效处理。

Party 包设计：石一楹

实现：陈宏伟

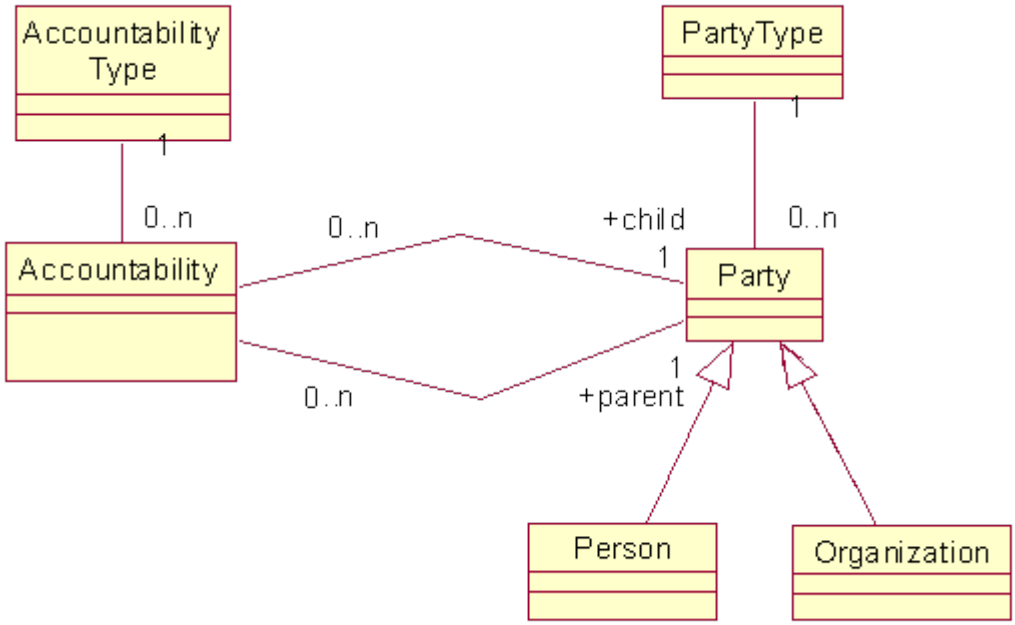
Party 分析模型、设计和实现考虑（二）

Martin Fowler《分析模式》

11 Accountability

如果你只需要处理一个组织层次，或者有数的几个组织层次，那么上面的组织层次应当能够解决你的问题,但是如果一个组织很大，那么在它的子组织和人员之间就可能存在复杂的关系，如果你使用组织层次解决所有这样的组织关系，那么模型就会变得非常复杂，难以维护和扩展。

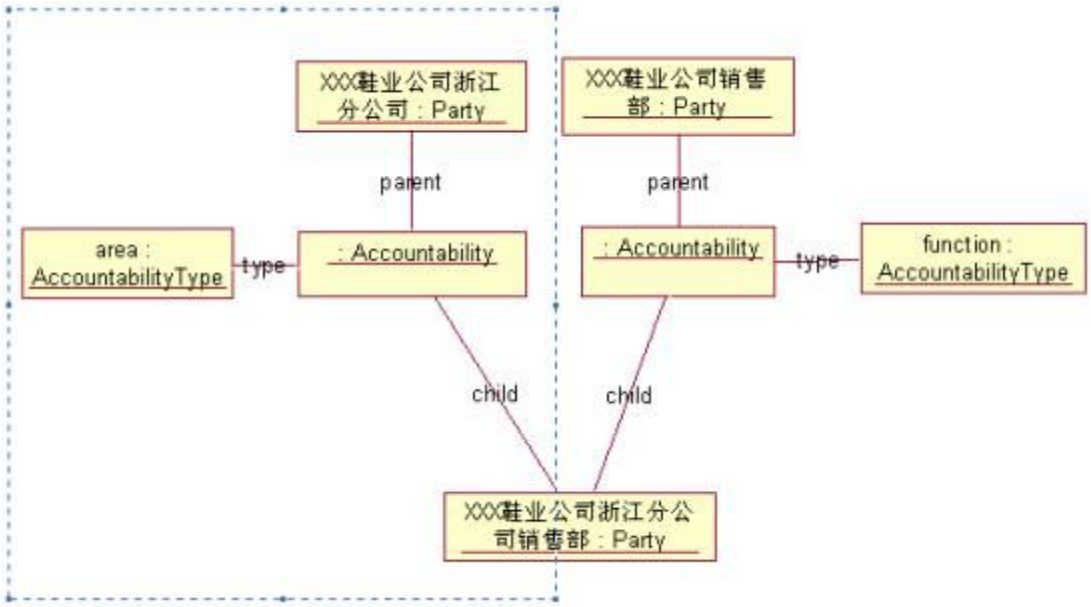
Accountability 顾名思义就是负责任的意思，它的责任其实就是指它负担起对于某一种类型而言两个 Party 之间的关系，在 MartinFowler 的 Party 之前，我更喜欢采用领域 domain 来表达这样的模型，但既然 Martin 的分析模式如此流行，我们用它的说法就是了。更何况如果 Party 涉及到 Person 的时候，用“责任”显然更自然。



Accountability 使用类型化关系模型达到模型的灵活性。这里有两个概念，一个就是 Accountability 的实例，Accountability 的一个实例表达两个 Party 之间的一个关系。而 AccountabilityType 则表达这种关系的属性，按我的说法就是领域。

用这样的模型来解决实际问题，你只需要为每一种关系（一个领域）建立一个 AccountabilityType 实例，然后用具有这种 AccountabilityType 的 Accountability 实例来连接两个 Party.

举个例子，一个鞋业公司有销售部和采购部，它有浙江分公司和江苏分公司，那么这里需要两个 AccountabilityType 的实例，一个是 Area,一个是 Function,



我们来看看上图左半部用虚线圈起来的部分，这里起作用的 AccountabilityType 是 Area,这表示其中的 Accountability 实例具有 Area 这种类型，也就是我们从区域这个角度来看浙江分公司销售部和浙江分公司之间的关系，浙江分公司销售部是浙江分公司的子。同样，右边表示，从功能的角度来看浙江分公司销售部和鞋业公司销售部之间的关系，浙江分公司销售部是鞋业公司销售部的子。

你应该已经注意到，和组织层次不同，我们没有直接说哪个 Party 是另一个 Party 的子，而是说某一个 Party 从某一个 AccountabilityType 的角度来说，是另一个 Party 的子。

11.1 Accountability 和组织层次

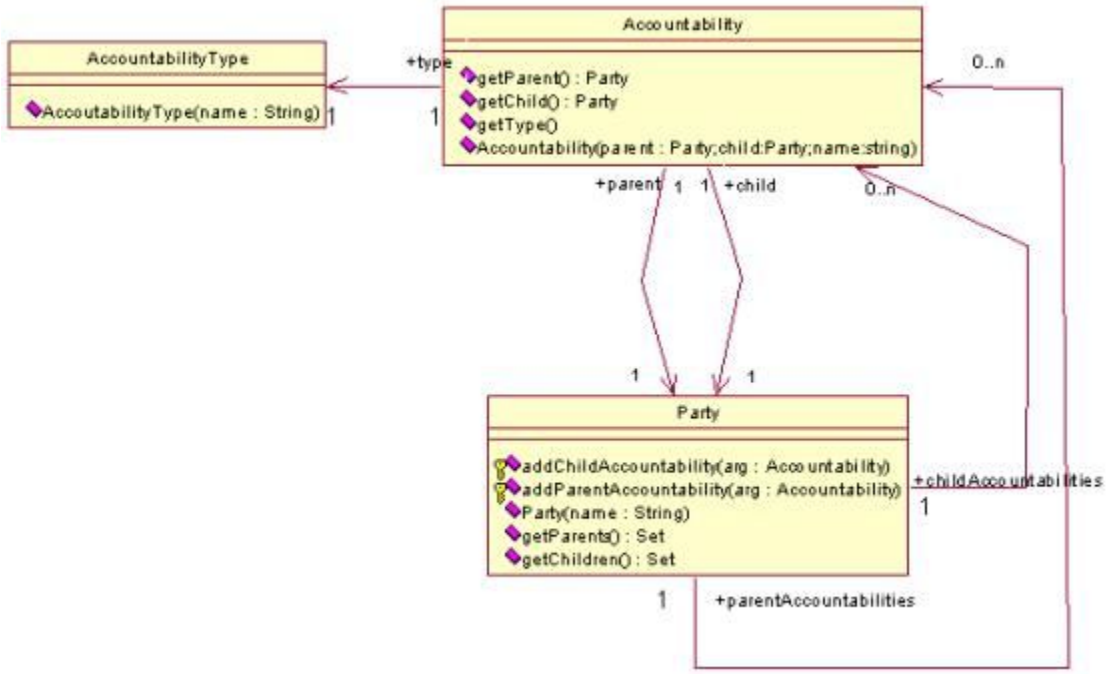
Accountability 能够包含组织层次模型，但是适度的复杂性是一个好模型的基本特性。如果你的组织机构不需要处理不同领域下不同的父子归属关系的话，组织层次显然是一个好的解决方案。

实际的情况是一般在一个系统内（或者一个组件框架内），你总是同时需要这两种模型的具体实现。你首先实现组织层次，在此基础上实现 Accountability,MartinFowler 建议一个实现同时拥有这两个模型实现所需要的接口，也就是内部用 Accountability 实现，但在外部可以保留组织层次所需要的接口。

从我们实现过的系统来看，最好把这两块分开，因为他们为不同的需求提供不同的模型。

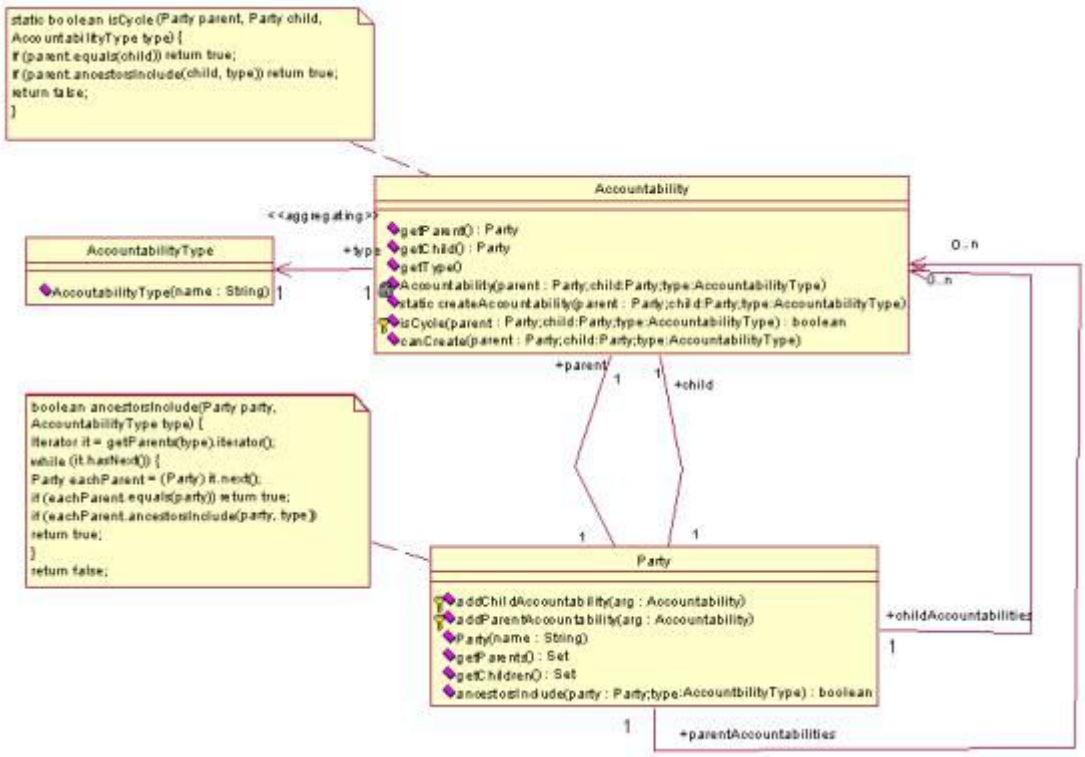
11.2设计

Accountability 有三个类，分别是 Party,AccountabilityType 和 Accountability:



Accountability 当然也不能逃避父子成环的检测问题，只不过这里可能要稍微复杂一点。首先我们需要在创建 Accountability 的时候检查 Parent 和 child 是否成环。

构建函数不能直接这样做，因此我们需要一个工厂方法。然后我们在这个工厂方法判断是否能够创建。该方法求助于 Party 来判断 parent 是否已经是 child 的子。



12约束和知识层

Accountability 提供了复杂的组织机构模型，但是我们对这个模型只有一个约束，那就是不能父子成环。如果没有违背这一个约束，你几乎可以用任何类型的 Accoutablity 把任意两个 Party 连接起来。

显然，在某些情况下，这种约束是不够的。要增加新的约束关系，一种方法是象我们处理成环问题一样把它硬性加入到模型的实现中，但这种方式不但使得每次增加一种新的约束关系都需要改变模型，也使得日后对不同 Party 之间约束关系的维护和配置变得异常复杂，不可重用。

知识层通常是一种超信息，这里我们可以通过对 PartyType 和 AccountabilityType 之间的连接来建立这样的知识层。

12.1级别式组织约束

知识层通过 PartyType 和 AccountabilityType 实例之间的关系来约束 Party 之间可能以某个具有 AccountabilityType 的 Accoutability 进行连接。最直接的方式是直接在 AccoutabilityType 和 PartyType 之间建立关系。

假设一个鞋业公司的组织机构可以描述为：

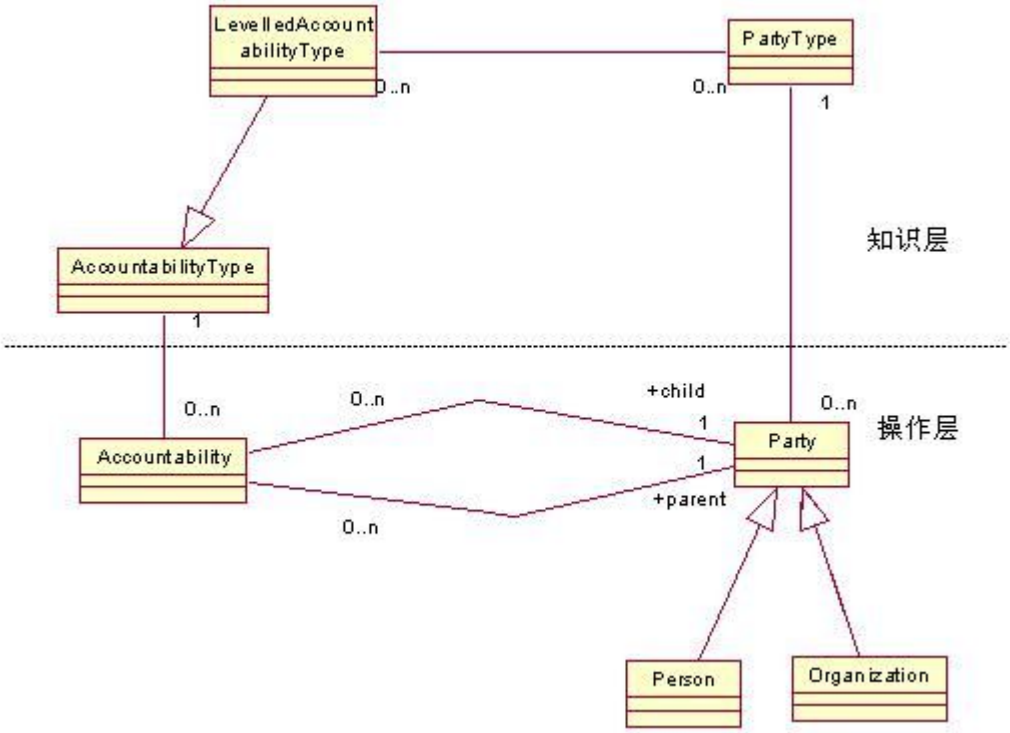
1. 集团公司，有销售部和采购部

2. 每一个子组织都有销售部和采购部

3. 集团公司有浙江分公司和江苏分公司

4. 浙江分公司有杭州分公司和绍兴分公司

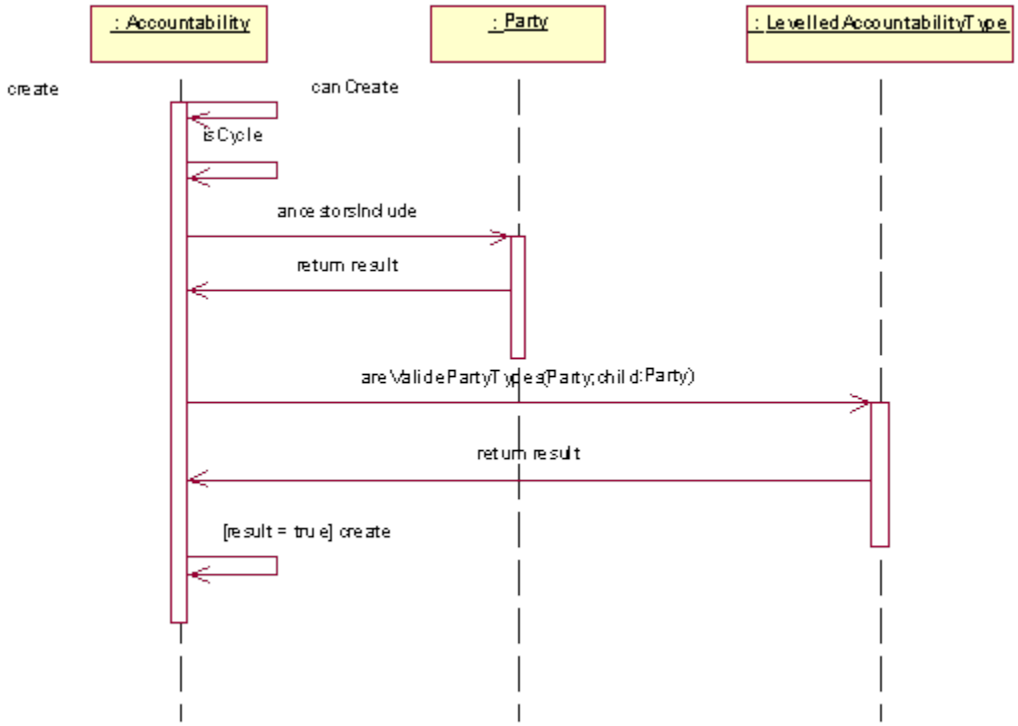
在 Accountability 中，我们没有办法限制杭州分公司和集团公司之间直接建立父子关系。但考虑一下杭州分公司的 Party 类型是城市(City),而浙江分公司的 Party 类型为 State，而总公司的 Party 类型为 Whole，那么我们可以通过严格限制 Whole-State-City 这样的级别式关系来决定对某一个 AccoutabilityType(这里是 Area)而言所具有的严格级别限制。



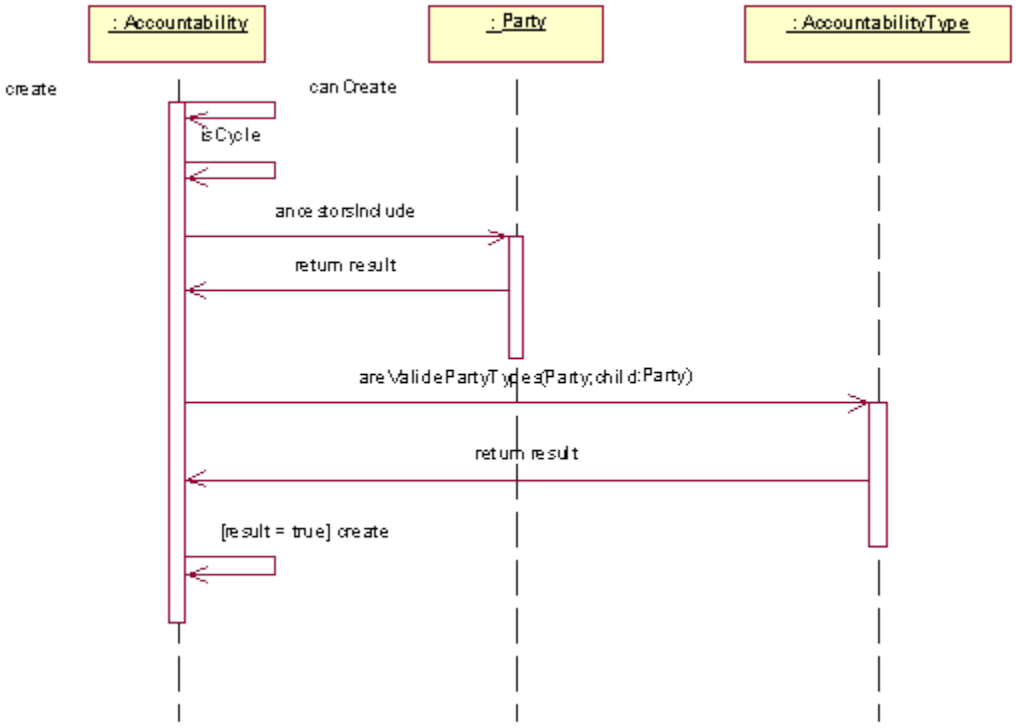
12.1.1设计

要运用知识层对组织进行约束，显然在工厂方法中除了判断是否成环，还要判断 Parent 和 Child 之间是否符合知识层的约束（这里的约束是一个严格的层次）。

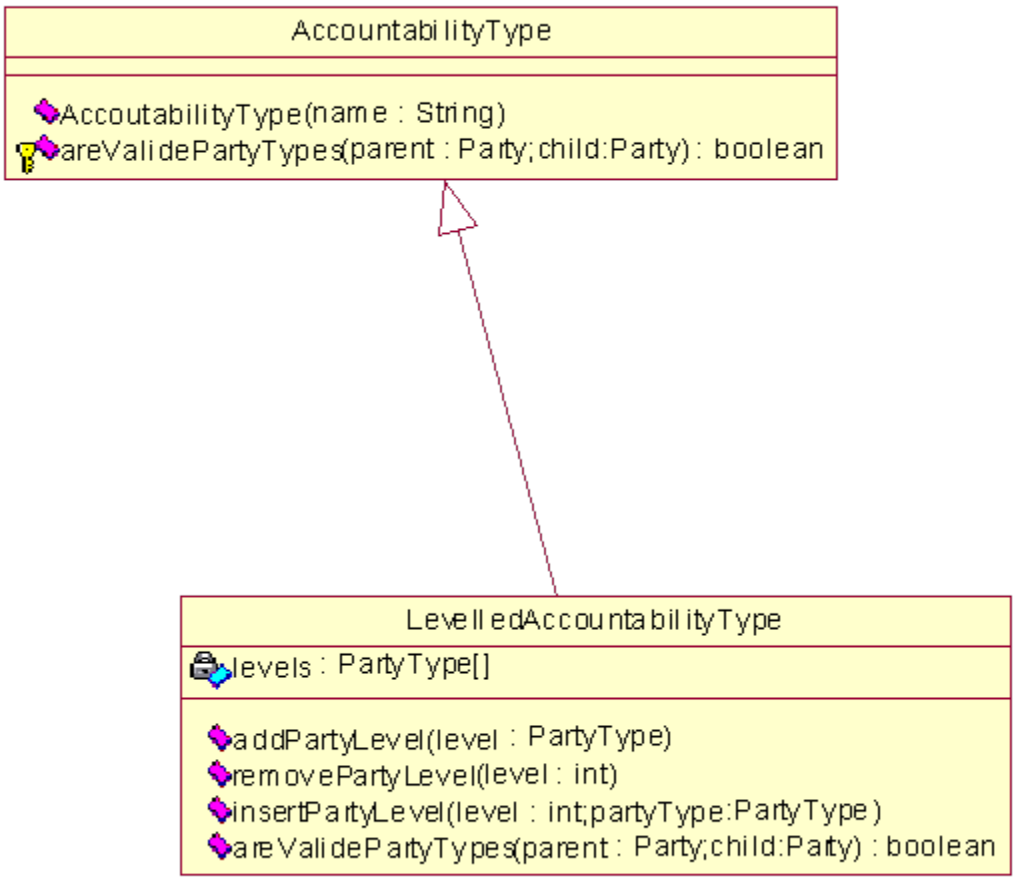
所以在判断能否创建这样的 Accountability 的时候，不但需要判断原先的 ancestorInclude,还需要求助于 LevelledAccountabilityType 判断这样的 Party 组合是否合理 areValidPartyTypes, 下面是这样的一个交互：



从这里的交互进一步可以看到，areValidPartyTypes 可以用于其他方式的校验，同时我们的知识层不只局限于严格的级别层次,Accountability 在交互中也不应当依赖于 LevelledAccountabilityType，我们使用模板方法即可。



把 areValidPartyType 上移到 AccountabilityType,



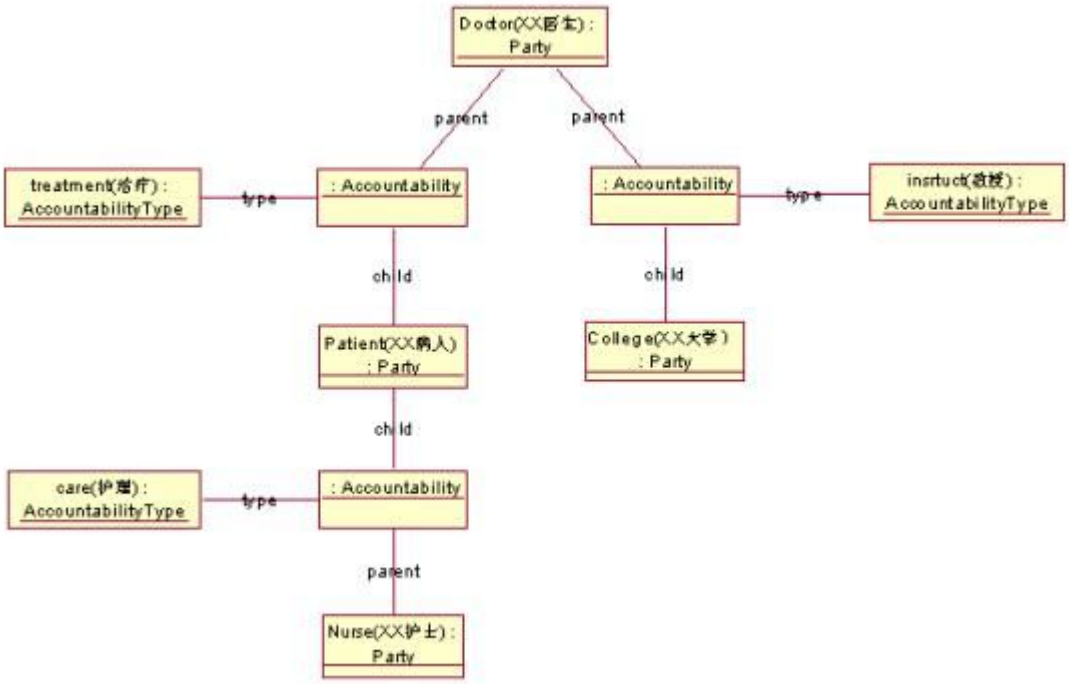
12.2连接规则

严格的级别式约束能够解决很多组织机构的问题，但当我们把整个 Party 模型加以考虑会发现，很多 Person 和组织之间的关系并不是严格的层次级别约束关系。

举个医院的例子，医生可以给病人看病，护士对病人进行护理，某些医生可以在大学中任教。

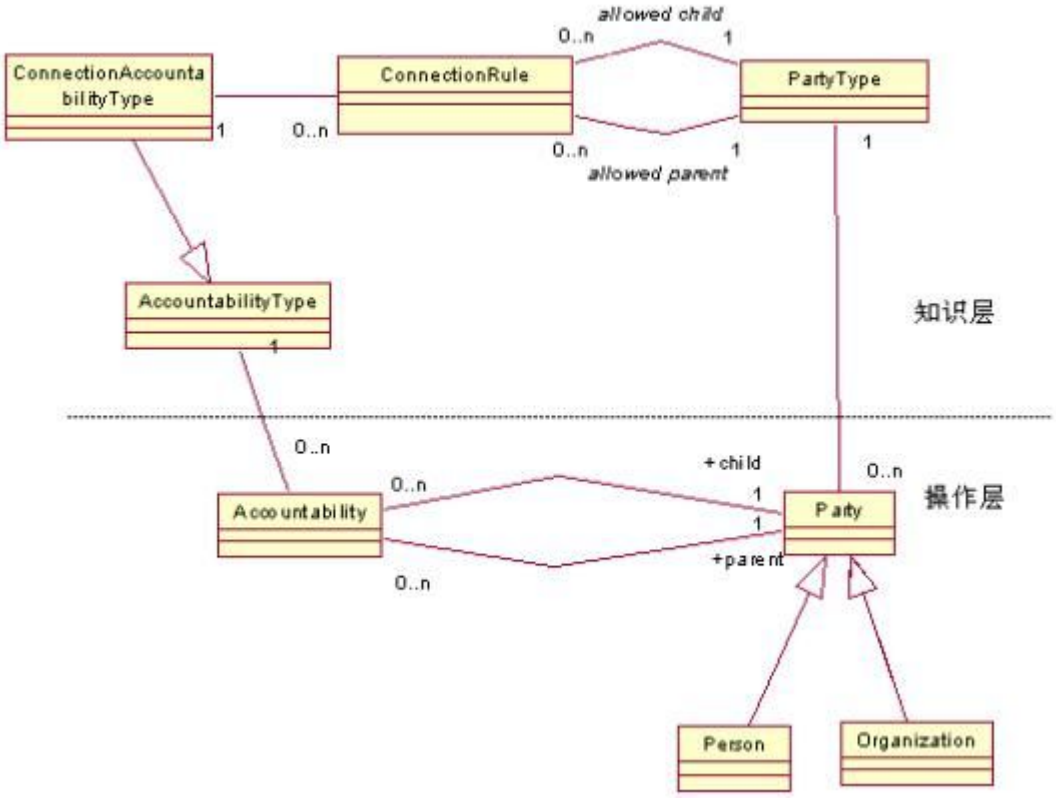
这里有三个 AccountabilityType,治疗 treatment,护理 care,教授 instruct。

具有 treatment AccountabilityType 的 Accoutability 实例可以用来连结医生和病人，具有 care AccountabilityType 的 Accoutability 实例可以用来连结护士和病人，具有 instruct AccountabilityType 的 Accoutability 实例可以用来连结医生和大学。



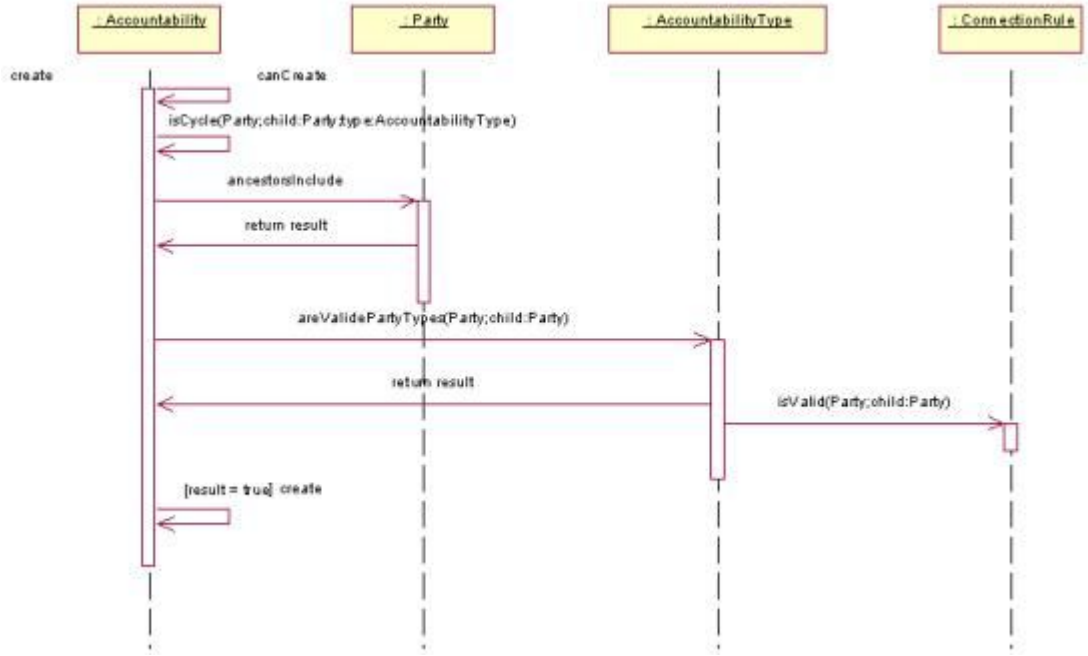
但是，你没有办法阻止一个病人 Party 通过一个具有 instruct AccoutabilityType 的 Accountabiliity 实例和一个大学相连，难道让病人给大学去上解剖学？

如果沿着严格级别约束再往前走一步，我们可以看到，这样的连接关系同样可以在 AccountabilityType 和 PartyType 之间进行关联得以解决。在严格级别约束模型中，AccountabilityType 通过包含一个 PartyType 顺序数组来解决这个问题。而这里需要在一个 AccountabilityType 和多个 PartyType 之间建立一种连接关系，因此：

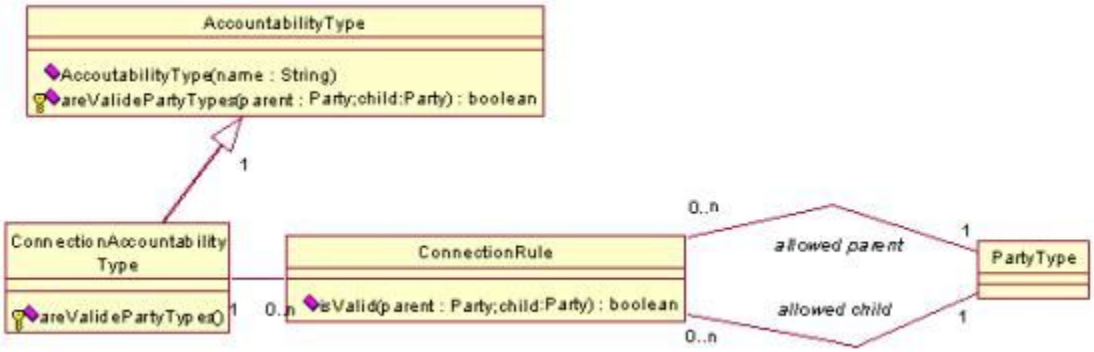

allowed child
0..n
1
ConnectionAccountabilityType
ConnectionRule
PartyType
1
0..n
0..n
1
1
allowed parent
AccountabilityType
1
知识层
0..n
+child
0..n
0..n
Accountability
Party
操作层
1
1
+parent
0..n
Person
Organization

12.2.1设计

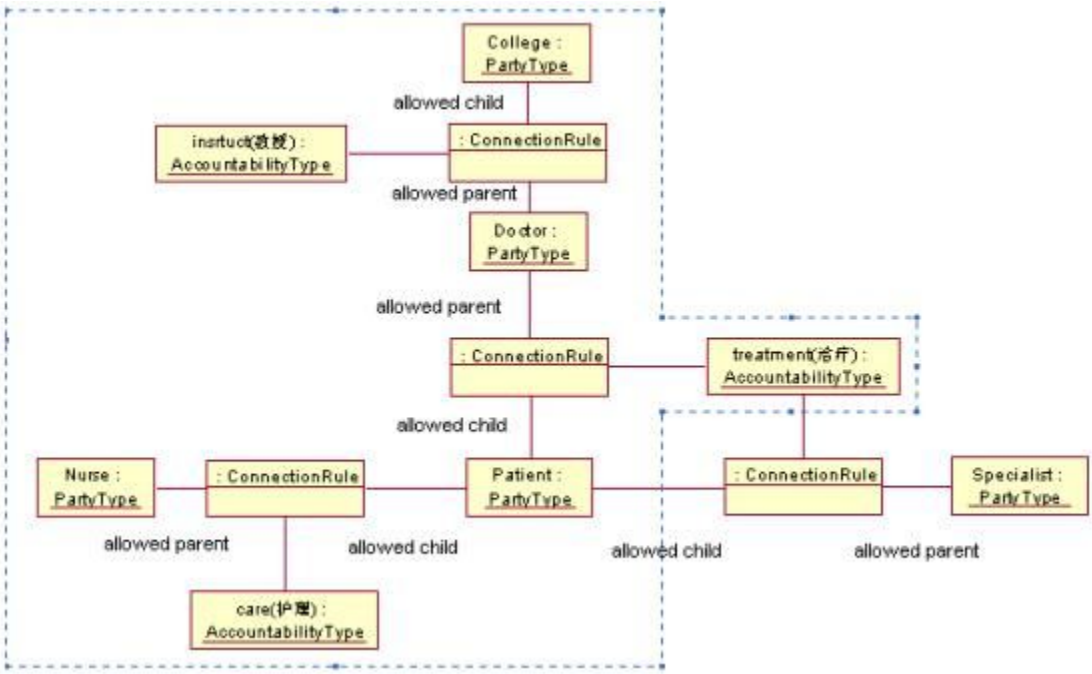
模拟一下 ConnectionRule 参与之后的交互过程：



因此，我们的设计如下：



加入 ConnectionRule 以后，我们可以用它来描述具有何种 AccountabilityType 类型的 Accountability 可以连接何种 PartyType 的 Party:

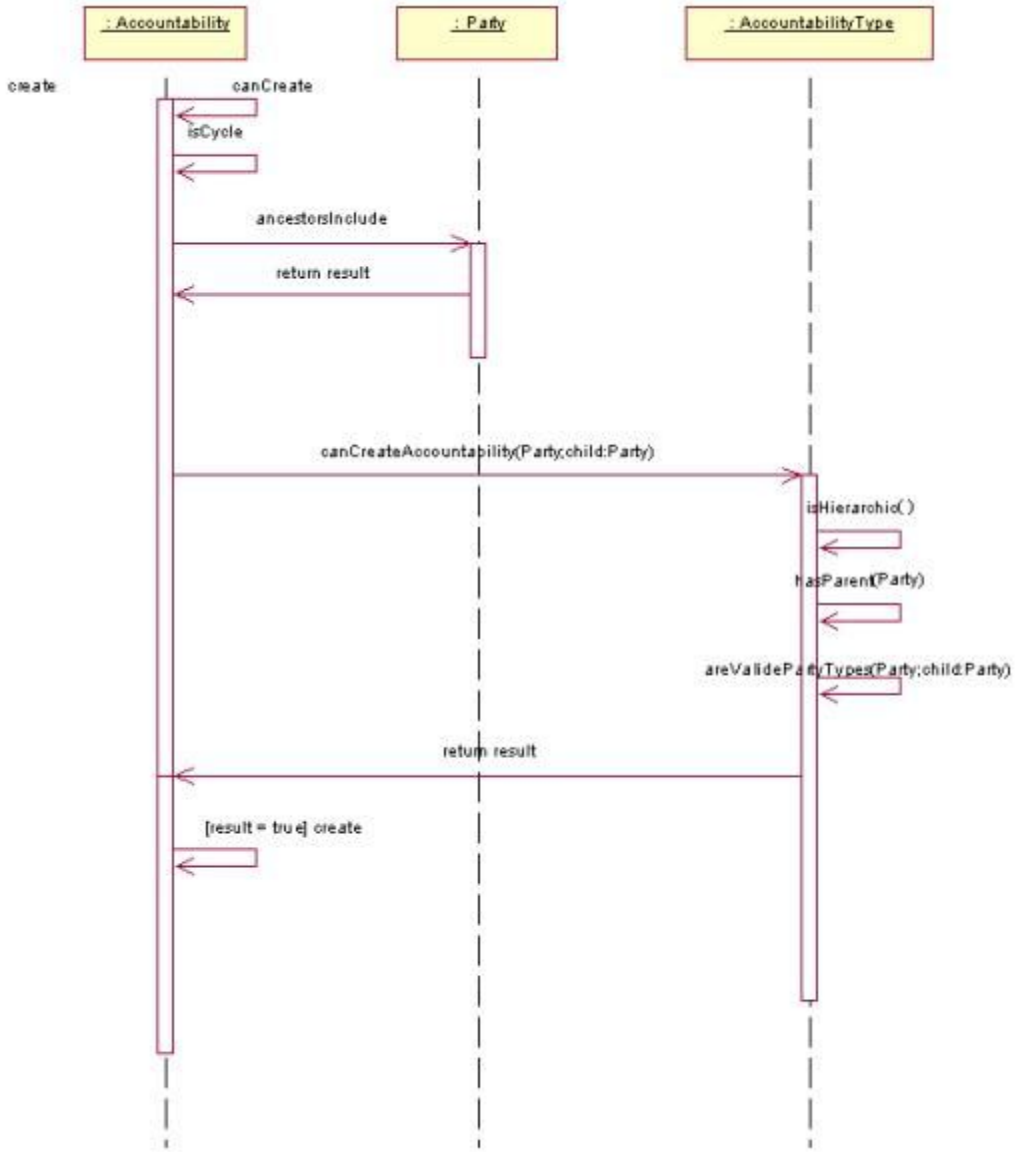


用虚线框起的部分描述了最开始我们给出的医生、护士、病人之间的连接规则。新加入的部分表示对于 treatment 这个 AccountabilityType 的 Accountability 实例还可以用来连接专家和 Patient.

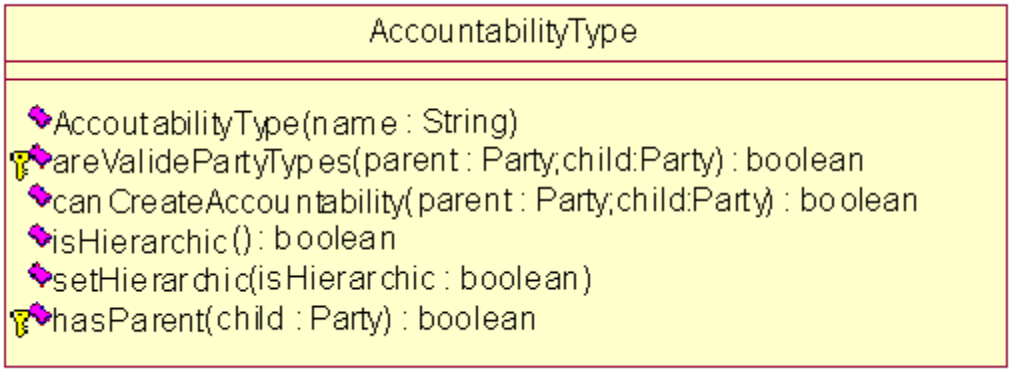
12.3单父限制

如果你足够仔细的话，你会发现 Accountability 模型并没有一个子 Party 在某种 AccountabilityType 上能够多少个父 Party 进行限制。

很多情况下，你需要进行这样的限制，组成严格的层次关系。你可能想继承一个新的 AccountabilityType 子类，但事实上层次对于 Levelled 和 Connection Accountability 都适合。我们可能利用一个标志进行限制。



前面，我们直接在 Accountability 中调用 AccountabilityType 的 areValidPartyTypes 是因为这样的协作已经足够了，Levelled 和 Connection AccountabilityType 都只需实现自己的判断即可。现在出现的问题是正交的，解决这种问题的最好办法就是建立一个新的方法包容原来的 areValidPartyTypes, Accountability 中调用 AccountabilityType 这个新方法 canCreateAccountability, canCreateAccountability 首先判断是否层次，如是，还需判断对 AccountabilityType 自己这个实例而言，这个 Party 是否已经有父了，如果有父，则不能建立。不然再去调用 areValidPartyTypes。



erptao 基础业务框架 Party 包文档

Party 包设计：石一楹

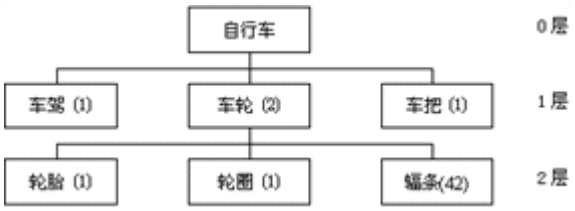
实现：陈宏伟

BOM 在 ERP 中的作用

要想了解到 BOM 在 ERP 中的关键作用，首先从业务上去分析，只有在这个基础上，才能用计算机去“描述”出 BOM；其中，我将穿插一些技术方面的实现方法。为了保证通用性，我这里用最普遍的树形构造法描述问题。

一、BOM 是计算机识别物料的基础依据

制造企业的产品，大都呈如下结构（前文说过，我们仅仅讨论一般的情况）：



图表 1 自行车产品结构图

这是一个大大简化了的自行车的产品结构图，它大体反映了自行车的构成（为了讲解方便，这里的例子也将是最简单的）。当然，这并不是我们最终所要的 BOM。为了便于计算机识别，必须把产品结构图转换成规范的数据格式，这种用规范的数据格式来描述产品结构的文件就是 BOM。

层次	物料号	物料名称	单位	数量	类型	成品率	ABC码	生效日期	失效日期	提前期
0	GB950	自行车	辆	1	M	1.0	A	950101	971231	2
1	GB120	车架	件	1	M	1.0	A	950101	971231	3
1	CL120	车轮	个	2	M	1.0	A	000000	999999	2
2	LG300	轮圈	件	1	B	1.0	A	950101	971231	5
2	GB890	轮胎	套	1	B	1.0	B	000000	999999	7
2	GBA30	辐条	根	42	B	0.9	B	950101	971231	4
1	113000	车把	套	1	B	1.0	A	000000	999999	4

图表 2 物料清单

这就是经过转换后的规范格式，它能为计算机所识别。这就是上一篇文章我讲过的**BOM 是计算机识别物料的基础依据**。为了叙述方便，这里我解释几个下文将常用的名词。

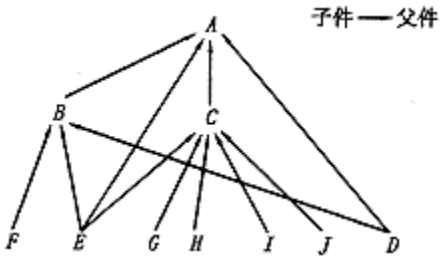
物料： 英文为 Item 或是 material、part 等。 是指为了产品出厂，需要列入生产计划的一切不可缺少的物的统称。不仅仅是通常意义上的原材料和零件，而是除了原材料和零件，还包括配套件、毛坯、在制品、半成品、成品、包装材料、产品说明书、甚至工装工具、能源等一切生产中的“物”；

物料类型（Item type）： 物料的来源；（图 2 中，类型为 M 为自制件，为 B 的为外购件）

物料分类（Item Class）：应管理上的需要将物料分类，基本作用是查询库存物料用。

物料编码（Item Number or Part Number）：通称物料号，要求物料号唯一，即一个物料一个编码。稍有不同的物料，其物料号也必须不用（物料编码为字符串类型）。

事实上，一个制造企业的产品结构一般来说不会如图（1）这么简单。对于ERP，用户可以认为一个产品的结构是树形结构的，而对于一个ERP程序编制人员，一个产品结构就是一张有向无环图。因为在一个产品中，一个物料是可以同时出现在不同的层次的。也可以出现在不同的中间件中的，只要不构成循环。如下图所示，是一个产品的通常组成结构：



图表 3产品结构拓扑图

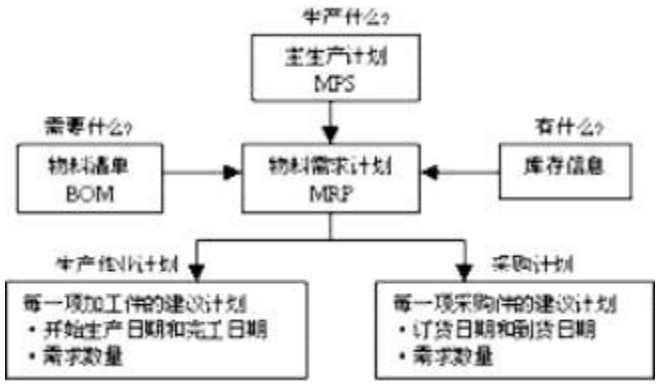
注：这是一张有向无环图，表示A有B、E、C、D等子件构成，C由E、G、H、I、J等子件构成……

二、BOM是编制计划的依据

对于（主生产计划）MPS：企业拿到订单后，必须保证按客户需求履行合同，企业履约率是持续不断地获取更多需求订单的前提。自然，首先得知道生产什么，只有在此基础上才能展开计划。所有产品都必须在BOM上体现，否则编制任何生产计划毫无意义，更不用说进行MRP计算。

注：当然，有的订单是按订单设计（engineer-to-order，ETO），这种情况将首先有产品设计部门设计产品，但最终还是需要进入BOM。

对于MRP（物料需求计划）：MPS是MRP展开的依据，MRP还需要知道产品的零件结构，即BOM，才能把主生产计划展开成物料计划；同时，必须知道库存数量才能准确计算出零件的采购数量。如下图：



图表 4 MRP 逻辑流程关系

三、是配套和领料的依据

车间要生产产品，自然要知道领什么料，领多少料，这也需 BOM 的支持。一个简单的例子，请参照图表 5 物料清单，组装自行车的车间需要组装 100 辆自行车，那么，我们告诉 BOM 管理软件需要生产 100 辆自行车，那么，他就自动展开 BOM 表，以“数量”这个字段为依据，计算出需要的物料。根据图表 6，就可以计算出配套领料结果：

车架： 100 件；车轮： 200 个（轮圈：200 件；轮胎：200 套；幅条：8400 根）车把： 100 套。

这样，实现了配套领料的自动化，而且还能以这个作为考核车间的依据。避免了物料的浪费和流失。

PS：在实际的生产管理中，还涉及到物料的有效日期、损耗率等等。比如幅条的损耗率为 20%，那么实际幅条的领料为（100/90）*42*2*100=9333 根。

四、 根据它进行加工过程的跟踪；

在一个分工明确的生产企业，往往工序比较多，从库存的物料到最终成形的产品，往往需要经过多次中转，在中转过程中，往往因为某个生产环节脱节，导致整个生产流程不能有效的进行下去，甚至出现 B 车间生产能力饱和、而与之相关的 B 车间却在等料， B 车间后续的 C、D、E……车间也随之生产停滞，这样造成了很糟糕的恶性循环。 上述情况用人工管理比较繁杂，而且不好控制，因为涉及到的不仅仅是一个车间或者一个部门，那么，BOM 提供了整体把握的基础数据，在一个完善的 ERP 产品的 BOM 表中，应该提供一个工序号字段，然后可以根据工序号以及配套领料单对生产过程进行跟踪。还可以以此作为考核各个车间的依据

五、 是采购和外协的依据

通过物料清单，可以定义外协以及采购件，通过展开生产计划，需要知道目前库存的原料是否足够应付生产，以确保安全库存以及生产需要。所以说 BOM 是采购和外协的依据。

六、 根据它进行成本的计算

一般说来，在 MRPII 系统中，产品的成本类型至少包括三种：标准成本（也叫计划成本、目标成本、）、实际成本、虚拟成本。MRPII 系统是按照成本发生的实际过程来计算产品成本的。它的计算基础是产品结构（BOM）。所有制造的产品都是从采购原材料或外购件开始的，也就是说，所有 BOM 中的最底层都是外购件，这层发生的成本是采购件进货价和采购间接费。二者之和即为物料价，实际的产品成本不可能仅仅包括物料价，还应该包括人工费、间接费等。以图表 7 中的产品为例，粗略演示成本计算过程。

1、 得到最底层（轮圈：2 件；轮胎：2 套；幅条：84 根）各个物料的配比量。

2、 父项（车轮）物料价=∑（子件物料价*子件绝对配比）

3、 父项（车轮）实际价格=父项（车轮）物料价+人工费+间接费

4、 上滚一层，

5、 是否为 0 层，是则退出，否则继续

6、 得到该层的各个物料的配比。

7、 跳到第 2 步

按照这个过程，就得到了一辆自行车最终的成本。根据上述的计算可以得到成本物料清单（costed BOM）。

七、可以作为报价参考

自然，有了成本计算结果，就可以以此作为报价的参考。在竞争激烈的商场，能以最

快最准确的投标报价对于企业非常重要。甚至于可以临时变更产品结构（替代物料等），以求更具有竞争力产品。

八、进行物料追溯

实际生产管理过程中，往往遇到这样的问题：想知道一个物料在哪些产品或者半成品中的物料清单出现。物料清单就提供这种功能。比如，一个企业既生产服务器，也生产家用计算机，那么我们查找内存条在哪些产品中用到，那么根据物料清单物料追溯，得到内存条在服务器以及计算机都需要内存这个物料。

九、使设计系列化,标准化,通用化

在产品设计过程中，往往碰到这么一些情况：一组物料在多种产品中都用到，但是，不同的设计人员，不同的设计部门都有自己的命名规则、不同的处理方式；一个物料在不同的设计部门有不同的命名规则等等。这些情况，往往增大管理难度，增加了冗余数据。

BOM 能很好的解决这个问题：当一组物料是通用的，那么我们可以将为该组物料定义一个通用件（插件），这样，无论任何设计部门都可以将此通用件作为“一个物料”直接使用；一个物料只能有一个编码，否则系统认为这是两个不同的物料。

当然，物料清单对于改进产品设计（设计系列化,标准化,通用化）的作用不仅仅这些，这里我只是举些例子而已。

总之，物料清单在 ERP 中起着**基础作用**。