

X-Programmer

总第 5 期

软件工程的播种机

非程序员

2001 9

www.umlchina.com

目录

方法

面向对象分析和设计技术	1
数字录音机设计方案	34
怎样避免用例陷阱	75
使用 UML 设计数据库应用	85

过程

统一变更管理的威力	102
项目管理规范-RUP 管理实施（四）	106

本期审稿：Think，黎宇，Windy

本杂志资料文章仅供学习交流之用

去往讨论组→
(已超过 11,000 人)

联系方式

投稿：editor@umlchina.com

广告：adv@umlchina.com

反馈：think@umlchina.com

播种机：<http://www.umlchina.com>

征稿

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

[请点击查看详情>>](#)

[Interface Hall of Shame](#)

[GOF patterns for GUI Design](#)

[请点击查看详情>>](#)

面向对象分析和设计技术

[Martin Fowler](#) 著, [easehawking](#) 译

我从事面向对象技术工作已超过十年了,并且掌握了许多分析和设计面向对象信息系统的知识。这是一个令人愉快的工作领域,而且每年都有新的进展。本文精选了一些我在这期间研究过的课题。

近来面向对象这门技术由“统一建模语言”UML (Unified Modeling Language) 所主导,它现在已成了面向对象开发的统一建模语言。如果你正在使用其它的旧技术,我强烈建议您马上转用 UML,因为它明显地将成为符号系统的统一标准。如果你正在考虑开始使用设计符号来工作,UML 是一个好的选择,因为它已经统治业界了。

然而,对使用 UML 有一个忠告:真正好的 UML 教材依然缺乏。当然我必须提及我自己的书《UML 精华》(UML Distilled),但它并不是一本教材,只是一本简单的指南,面向那些已具有面向对象分析和设计并需要简要了解 UML 的读者。从 Grady Booch 即将推出的书中我所见的部分来看,这本书还不错,但要在 1998 年末才能出版。

本文分成三大部分。第一部分概述一些普通的分析和设计话题:技术要素、标准化和 UML, CASE 工具以及为什么要不辞劳苦地研究这些技术。接下来的第二部分我会介绍不同的建模技术如类图、交互图和用例图等。最后一部分阐述处理技术,例如逐步交付 (evolutionary delivery)、模式和再加工 (refactoring) 等。我提到的这些技术既有 UML 的也有非 UML 的。尽管 UML 是主导的建模语言,但除此之外还有一些重要的技术作为补充。如果需要,我会毫不犹豫地使用它们。

请让我知道你们对这个网站的看法,我希望这个网站能发展成为面向对象方法的一流的信息资源。另外,如果你有什么这方面的技术问题,不要犹豫,欢迎来联系我。我向来是很喜欢回复电子邮件的。我还从事这些技术的培训和咨询工作,如果您有兴趣,请告诉我。我的书 UML Distilled 中更详细地论述了这里的许多问题,如果您觉得这些文章简单,您可以看看那一本书。

● 普通问题

标准化和 UML

几年前我们看到许多不同的方法被各自的元老推广，而用户迷乱地不知道该选择哪一种。标准化好像是不可能的，因为许多元老声称这是没必要的。现在我们听说UML是这些方法中的新标准。这就是全部的事实吗？它将对你有什么影响？

UML是标准的符号但不是一种方法

统一建模语言(UML) 是一套用于面向对象系统建模的标准符号，受到对象管理组织OMG (Object Management Group) 的支持，包括OMG的各成员公司，也就是说，面向对象软件商业中的每个人。

然而，认识到UML仅仅只是一套标准符号这一点是很重要的。实质上它定义了一些你可以用于描述一个系统的图表以及这些图表表示什么。它不涉及你要去建构造一个软件的过程。这样一个过程，或方法，包括一个需要完成的任务列表、它们要完成的顺序、产生的交付品 (deliverable)、每个任务所需的技巧等。而传统的正规方法学包括了符号和方法。

这观点就是：通过符号的标准化，让一个方法中所有的交付品都使用UML，软件开发者就能更好地进行交流。但是不同的组织可以自由地选用无论哪个他们想用的方法去着手开发软件。使用UML的一些方法被提出来了。Rational 发布了他们的Objectory方法（我听说是称为统一过程），几乎是基于Ivar Jacobson的工作。HP的Fusion方法也是另一个被广泛谈论的方法。

你应该使用UML吗？

一个字：是！旧的面向对象符号正快速地消逝。它们还会残留在UML稳固前出版的书上面，但新的书、文章等等将会全部以UML作为符号。如果你正在使用旧的符号，你就应该在1998年间转换到UML上。如里你正要开始使用建模符号，你就该直接学习UML。

但您需要认识到UML是惊人复杂的。这是因为它试图为你需要处理的各种情况都提供图。如果你准备开始学UML，你应该有选择性。着重于使用一部分符号。好的开始是使用类图和交互图。即使是类图的时候也是从基本的概念开始，不要想一下子掌握全部。还有，不要片面地认为只有UML才是实用技术。有些技术例如CRC卡，尽管不是正式的UML，但还是很有实用价值的。

对于这种方法，它取决于你所打算做的和你的软件中的处理步骤的数量。寻找一种为你心目中的软件而设的方法。一个为实时系统而设的过程是不可能很好地适用于商业信息系统的。有一种强烈的趋势偏向

于更小尺度的方法学，试图减少传统方法学的步骤数。这尤其适合于由几十个或更少的开发者进行的中规模的开发工作。

你可以在官方UML站点上免费地获得关于UML的信息，但这冗长的材料是针对方法学专家而不是一般的开发者。书正在出现。我还没对UML的新书作过全面的调查，因此在这里无法作出评价。即使我作了，在我出第一本的UML书（《UML精华》UML Distilled）的时候也难以作出不带偏见的调查。旧的方法书正慢慢地转向UML符号，例如Jim Odell的基础书。

发展历史

在几年前曾经有过一些面向对象方法标准化的努力。OMG的一个分析设计特别兴趣小组（SIG）开了几年会，发表了一份厚厚的面向对象方法报告，但遭到广泛的忽视，除了一封由元老派联名批评方法标准化的观点的公开信。Grady Booch尝试了一个更不正式的策略，但在OOPSLA93的早餐会上他发现其它的元老并非那么感兴趣。我记得在OOPSL93的一个座谈会上，元老们说，尽管这是可能是一个好东西，但没有人可能为此作出努力。

震动这个小世界的事件是OMT的重要元老Jim Rumbaugh加入了Rational公司(在那里Grady Booch是首席科学家)。他们许诺要一起工作，建立一种统一的方法。作为主要方法学家中的两个，组合后的 Rational 统一方法将是很有分量的。

那事件使方法学家们陷入恐慌。一方面Rational一直称他们将得到Microsoft方式的标准化。Grady和Jim宣布方法之战结束了：“我们胜利了”。但另一方面方法学家们建立一个反Booch 联盟。真是有趣。

更重要的是1995年6月OMG恢复了分析设计特殊兴趣小组。在早期成形时它在OMG的小角落中工作，得不到OMG管理层太多的注意。现在OMG正认真地对待它并派Mary Loomis和Jim Odell出任副主席。显然OMG不想让Rational在不受外界干预的情况下制定标准。特殊兴趣小组为标准准备了一个RFP（建议请求）：建模接口和语义及其符号。

在1995年的OOPSLA会议之前，Rational已准备好精心装点地推出他们的第一个统一方法（Unified Method）的方案（版本0.8），但尽管有Jim Rumbaugh的歌颂，最后却反成了笑柄。但Rational还是暗中留了一手：购买了Ivar Jacobson的Objectory公司。有三个顶级的方法学家在Rational麾下，它怎会不能强推一个标准呢？

1996年期间，Rational将Ivar Jacobson的思想融入到统一方法（Unified Method）中，并将其名字改为“统一建模语言” UML（*Unified Modeling Language*）。他们也开始进一步向其它观点开放，并得到了许多有

名的公司像Hewlett-Packard、 Microsoft、 Oracle和Texas Instruments等的认可。

在1997年初期一些提案补提交到OMG 分析和设计任务特别工作组。Rational的提案是最受注目的一个，但还是有一些其它组织的提案，比如最引人注目的IBM ObjectTime。到了这里整个过程烟雾弥漫，还有很多元模型之类的处理。重要的成果是，所有提交最后汇合到统一的UML。

不是所有的方法学家都赞成轻率匆匆奔向UML。最公开反对的是由Don Firesmith、Brian Henderson-Sellers和Ian Graham领导的OPEN联盟。他们称UML过分根源于已不适合于对象建模的旧数据建模技术。他们提出一个竞争方案：开放式建模语言（Open Modeling Language (OML)）。

但UML控制了局面。在1997年中前，所有OMG的提交者都达成共识，得到了UML1.1版。这在1997年未被OMG正式采纳。OMG最后搞出了混乱，取得了Rational 1.1版并改名为OMG 1.0版。至于他们为什么要制造这样的混乱我就不得而知了。现在一个工作组正在整理这个规范，准备推出1.2版（对两种版本模式皆然）。1.2较1.1并没有显著的改变。

为什么要使用分析和设计技术？

说到这里，软件开发的真正要点是精简代码。图表毕竟只是漂亮的图片。用户不会多谢你这些漂亮的图，他们要的是可运行的软件。因此在你正考虑用这些技术的时候，重要的是问问你自己，为什么要这样做，它们在你编码的时候能有怎样的帮助。没有任何的经验证据来证明这些技术是好的还是坏的，只是我在这里谈论一些我时常使用的理由。

面向对象学习

许多人在谈论与面向对象相关的学习曲线，臭名昭著的“范型转移”（paradigm shift）。在某些情况下，向面向对象转变是容易了；另外的情况下，在使用对象来工作时会有些障碍，特别要发挥它们的最大优势的时候。并非说学习用面向对象语言编程是困难的，而是说，要花一段时间学习去挖掘对象语言所提供的优点。

这个站上的技术的目的是在某种程度上帮助人们用好面向对象，但不同的技术有不同的优点。面向对象学习中最有价值的技术之一是CRC卡。它们被设计用于首要的用途，并刻意地有别于传统的设计技术。它们因为注重责任，没有复杂符号，显得特别有价值。交互图非常有用，因为它使消息结构非常明确，而且这样也可以突出一个对象完成所有工作的过集中化设计。类图对学习对象有好处也有坏处。它们与数据模型体贴地相似，许多建立好数据模型也适用于建立好的类模型。主要的问题很容易建立了一个非常面向

数据而不是面向责任的类模型。

模式（Patterns）是面向对象学习中极端重要的技术，因为它们让你集中注意好的面向对象设计并通过例子来学习。一旦你已经懂得简单的建模技术，如简单的类图和交互图，就要开始看模式了。

另一个重要的过程技术是演进式开发。它并非以什么直接的方式帮助你的学习，但却是发挥面向对象效力的关键。如果你一开始就作演进式开发，那么你就会从背景中认识到恰当的过程类型，并开始理解为什么设计者建议按它们所说的去做。

在早期使用一种技术时你倾向于按着书来做。一旦你习惯之后你会发现它并不按你喜欢的方式去做。别犹豫去改变事物。技术是为你服务的，而不是相反。如果你觉得某些构想没帮助，那就别用它们。如果你需要某种有用的构想，那就加上它吧。只要你确定你真的从概念上和实现上理解这些构想的意思。我喜欢从三个视角看各种构想：概念，规格和实现。

与领域专家交流

开发的最大挑战是建立一个正确的系统。因为我们必须用我们的行话与用户交流，而用户又有他们自己不可思议的行话，使得开发变得困难。（我做过许多有关医疗保健领域的工作，在那里行话甚至还不是英文的！）与用户良好的交流和对用户世界的好的理解是开发好软件的关键。

这里明显的技术是用例。用例是你的系统的外视图，它们被用来表达系统功能有好一段时间了。一个好的用例集是理解用户需要的核心内容。它们也提供了项目规划的有效手段，因为它们控制演进式开发。演进式开发能反馈给用户软件开发进行到哪了，其本身就是一种有价值的技术。

除了就表面的东西交流，重要的是探讨更深入的东西。你的领域专家怎样理解他们的世界？类图在这里格外重要，但要以概念方式去使用。将每个类看作用户头脑中的概念，他们语言的一部分。这样你作的类图就不是数据的图或是类，而是你的用户语言的图。文献[Odell foundations]是这种类图思想的好的信息资源。如果过程是用户世界的重要部分，我发现活动图非常有用。因为它们支持并发过程，能帮助你摆脱不必要的序列。类的连接被放到后来的设计中处理，这种方法在这更概念化的阶段变成了一种优点。

理解大图

作为一个顾问我常要在一个复杂的项目中走马观花，并要在一段非常短的时间内展示才智。我发现那样的话那些设计技术没有价值，因为他们给我的是系统的整个视图。对类图的一瞥能迅速告诉我在系统中出现了什么样的抽象概念，需要进一步工作的部分在哪。更深的查究时我要看类如何合作，就要求看系统中关键行为的交互图。

如果这对我这外人有用，那它对一般的项目组也同样有用。在大项目中很容易只见树木不见森林。手头上拿着些精选的图表，你会发现更容易把这软件看清。用包图做一张路线图（Road-Map）来从更高层次上的总括类图。在你为路线图作类图的时候请使用规范的观点。非常重要的是在做这种工作时要隐藏实现。不要归档每个交互，而仅仅是关键的那个。使用模式去描述系统的重要思想，它们有助你解释为什么你的设计是这样的。它也助于说明你摒弃的设计，和什么要这样做。我总是在最后忘记那样的决定。

CASE 工具

结构化方面早已培育了一种工具产业来协助方法的工作。这种被称为计算机辅助软件工程的工具承诺了一个美好的未来，但带来的却不多。有时它声称有朝一日要取代程序员，但可能它们没有意识到COBOL也曾作过同样的夸口！

关键的问题是它们为你作的是什么？它们能加速软件生产的过程吗？它们能帮你建造更高质量的设计（来让你快速发展）吗？凭我的经验，CASE工具可归结为两个时期：设计文档和代码生成。

设计文档（Design Documents）

在纸上画图形模型很费时间而且容易出错，特别是在它们需要经常修改的时候。因此专业的画图软件能像为其它行业提供的CAD那样支持软件工程。不幸地CASE工具好像常假设用户拙于模型从而对用户作了众多限制。它们只让你画它们认为合法的，以便于清晰的表达。

软件工程的最好的画图工具是白板（印刷白板更好）。它可以快速而简便地画和擦拭，便于多人看，而且灵活。它激励了思想的尝试和摒弃，并支持在分析设计中相当重要的团体思考。没有任何计算机工具能与之相比。然而对于特有的文档一些计算机工具是非常令人满意的。首先值得考虑的是标准的画图工具，例如Visio (你可以从Navision得到好而且免费的Visio模版)。这样一个工具便宜而且稳定（根据CASE工具标准），运行于标准桌面平台上。许多画图工具缺少的重要功能是让线条粘在方框上，使得方框移动时线也跟着移动。作CASE工具对作图造成的局限性通常是缺点多于优点。

除了作为绘图工具，它们还提供了知识库（repository）。知识库实质上是一个与图示工具相关的数据库。这样一个数据库应该能够按字母顺序列出所有的类和一个类所有特性（关联、属性和操作等）的列表。通常这样能合并表述在多个图表中的模型。

在这里元素之间的链接的正确性是一个重要的因素。一个简单的实验是改变一个类或模型的名字，然后看看工具是如何让所有东西保持最新的。理想地一个工具应该更新知识库中所有地方里关于这个类的引

用。这不常发生。特别要检查的是在用于非正式描述的文本中的类引用。通常这些不会改变，而且没有工具能有助于跟踪这个变化。另一个要考虑的地方是不同技术之间的可追溯链接。如果交互图中一个操作调用在它的特征标记（Signature）中引用到一个类，这也要改变吗？为了让工具有用，必须为建模者提供良好的支持，让他看到所作的一点小变化引起的后果。

另外一个要考虑的因素是，这种情况下信息要怎样访问。除非每个人都习惯于在线工作，知识库和图表都要打印成一份设计文档。工具应该能以最少的代价产生出达到桌面排版系统质量的设计文档。这又遭到普遍的失败，许多工具要花上几天的功夫才能将一份设计的内容转化成象样的纸张形式。

这里的基线是粗陋的字处理工具，提供了很好的设计文档功能，但不便之处是难以与图表的变化以及费时的需求校对保持同步。然而这样需要的时间比CASE工具所用于生成设计文档的要少，而且具有实用的查找和置换机制，能够捕捉到全部对名称的引用。另一个选择是使用简单的数据库。

代码生成和执行

代码生成可能是CASE工具吹嘘得最厉害的功能了。代码生成器实际上是编译器的一个同义词，而且代码生成器应该用相同的标准和相同的因素来判断。（因此我对代码生成器挖苦的定义是：工作不正常的编译器。）

第一个要问的问题是：生成了多少代码？绝大部分的代码生成器实际上只是接口生成器：包括了属性和操作的类定义。并没有任何进行实质性处理的代码。用这些工具时要注意看看接口生成的程度。所有属性的访问和更新都生成了吗？操作是不是只有名字，还是包含了正确的特征标记（signature）？

这种方法明显的局限是只有生成了接口，还留下大量的主体代码的编程工作要做。另一个考虑是怎么处理模型和代码的演化。如果一个代码生成器不能支持分析设计模型的改变引起的实现变化，那它的作用是有限的。许多CASE工具似乎都假设代码只在设计完成后才生成。在实际的项目中分析、设计和实现交替进行。有效的接口生成器应该提供CASE工具和编程环境之间的紧密连接。尽管如此，如果能和编译器密切结合，这些工具就更有价值了。

有些工具还支持逆工程。这能让最终代码，最好是包括不是由CASE工具生成的，被逆转进CASE工具。这样工具能提供比源代码更高程度的可读性。自然地这么一个功能需要一种高度面向实现的技术，因为期望一个工具能逆向解析工程分析是不合情理的。

生成代码是很好，但你要调试时会发生什么事呢？你是被置之于编程环境只能依靠乱糟糟的名字来帮你，还是你干脆是调试状态图去看看哪里出错了。如果你打算用一个工具来生成代码，那么一定要先尝试

过整个过程。在你使用演进式开发时，你不能只是生成代码，然后就后悔。

不同的工具让你控制代码生成过程的程度也有所不同。在有些工具中它们只能生成它们认为是好的。但这与你的想法想一致吗？有些工具能让你指示如何去生成代码，通过提供提示或甚至是用代码片断来指明该如何生成代码。检查一下工具的个性化程度，看看它是否适合你。

带注释的文献目录

在我开始做这种比较工作时我打算要读遍全部的面向对象分析设计的书和绝大部分的这方面的论文。但很快我就不得不放弃了——这方面的材料实在太多了。下面是我读过了许多特别有趣和实用的作品以后做的注释文献目录。

[Beck]

Beck, K. Smalltalk Best Practice Patterns. Prentice Hall, Englewood Cliffs, NJ, 1997.

严格地说并不是一本分析设计的书，但很实用，因为它提出的许多模式深入研究了文档尚稀的再加工技术。它的许多模式对其它的面向对象语言也很有用，所以尽管你不是smalltalk用户，这本书也值得一读。

[Beck和Cunningham]

Beck K and Cunningham W "A laboratory for teaching object-oriented thinking" Proceedings of OOPSLA 89

提出CRC卡的论文。

[Booch]

Booch, G. Object-Oriented Analysis and Design with Applications, (Second Edition Addison-Wesley, Menlo Park CA, 1993.

如果你正在找一到面向对象分析与设计的入门书，没什么人会否定这是一个有价值的地方。这本书分几章叙述面向对象概念，描述它的符号和方法，还有一些例子来讨论如何通过C++来使用这些技术。作为UML的作者之一，他对UML的发展起着领导的地位。这本书的最大不足是过于着重于实现视角，那么用[Odell foundations]来调剂一下也好，虽然它不是用UML符号的，但它的观点仍具价值。

[Booch solutions]

Booch, G. Object Solutions: Managing the Object-Oriented Project, Addison-Wesley, Menlo Park CA, 1996

这是项目管理中的Booch，也是一份好的读物。他强调积极地抗击风险的演化过程。他加入很多战争故事和一堆有道理的著名规则。而且它也不太长，没有[Goldberg and Rubin]那么全面，但还是不错的。

[Buschmann et al]

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M, Pattern-Oriented Software Architecture: A System of Patterns, Wiley, Chichester UK, 1996

Siemens的几个开发者收集的模式。包括一些体系结构模式和设计模式。

[Coad, North, Mayfield]

Coad, P., North, D. and Mayfield, M. Object models: strategies, patterns, & applications, Prentice Hall, 1995.

Coad等从事模式。这些与PLoP风格不一样。它们是模型的典型组合。书的重要部分是第6章，在那里Coad解释如何用他的模式与生成一个模型。根据这种方法，首先是要识别出易于发现的对象。对于I.S.系统这些对象是容器和事务，对于实时系统则从设备开始然后才是容器和事务。这些模式说明了类的典型构造，因此你可以看看所有这些模式，从中找到容器，以及看看它们可以应用到问题中的哪个地方。这些模式在各个使用者中展开。每个模式都暗示了典型的行为。

http://www.oi.com/oi_home.html

[Cook 和 Daniels]

Cook, S. and Daniels, J. Designing Object Systems: object-oriented modeling with Syntropy, Prentice Hall International, Hemel Hempstead, UK, 1994.

我的最爱之一。Cook和Daniels想要很多形式化的严格方法，然而又将这些形式隐藏在OMT风格的图背后。对那些需要形式化的人这是最好的书了。尽管你不需要形式化，他们的思想和严谨性还是值得一看的。特别有价值的是他们对类图中观点的讨论。这也是Harel状态图的最好的信息资源。然而这是一本高级的书，我不建议作为入门书。

[Cockburn]

Cockburn, A. Surviving Object-Oriented Projects, Addison Wesley, 1997

着重于项目管理的好书，十分强调演进式开发。每个领导项目的人都应该读读这本书。

[Fowler]

Fowler M, Analysis Patterns: Reusable Object Models, Addison-Wesley, Reading MA, 1997

抽取我在开发面向对象软件中建立过的领域模型，包括了健康护理、金融贸易、会计、规划和分层建筑等方面的例子。

[Fowler, UML]

Fowler M with Scott K, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley, Reading MA, 1997

UML的简单指南（浅略的）。为那些已经了解基本的面向对象建模而需要快速概览UML重要部分的人而写。

[Gang of Four]

Gamma E, Helm R, Johnson R, and Vlissides J, Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, Reading MA, 1995

模式运动开出的第一朵花，且是近十年来出版的最重要的软件书。一整套的23个附有基本原理、模型和示范代码的通用设计。如果你是个很认真的面向对象程序员你就应该拥有这本书。

[Goldberg and Rubin]

Goldberg A. and Rubin K., Succeeding with Objects: Decision Frameworks for Project Management, Addison-Wesley, Reading MA, 1995

一个全面介绍项目管理和对象技术的采用的书，带有许多案例研究。覆盖范围比 [Booch]更广。

[Graham]

Graham I., Migrating to Object Technology, Addison-Wesley, Wokingham UK, 1995

最好的规则资料之一，包括将模糊逻辑融入对象系统的一次尝试。并有面向对象开发过程的详细阐述和用例的实用观点。

[Jacobson]

Jacobson I., Christerson M., Jonsson P. and Övergaard G. (1992) Object-Oriented Software Engineering: a

use case driven approach, Addison-Wesley.

这本书首次将用例引入对象方法，极具影响力。对用例的处理方法是该书最有价值之处。它许多建模技术被吸收到UML中。

[Jacobson BPR]

Jacobson, I., Ericsson, M. and Jacobson, A. The Object Advantage: business process Reengineering with object technology, Addison-Wesley, 1995.

Jacobson将用例应用于商业。有点宣扬投机和对计算机内涵的延伸，但每个使用用例的人都应该想到更广的商业用例。

[Martin]

Martin, R.C. Designing Object-Oriented C++ Applications Using the Booch Method, Prentice Hall, Englewood Cliffs, NJ, 1995.

包含了一系列大型（整本书的）而详细的案例研究。在每个案例中他带你贯穿Booch方法中的类图、交互图和包图的设计。在这些设计中他向你展示如何深入运用C++。在我所见到材料中，这本书对包图的叙述是最好的。

[Meyer]

Meyer B. Object-Oriented Software Construction, Prentice Hall, Englewood Cliffs, NJ, 1997

出了新版并变庞大了，但尽管它很大部头，从软件工程的立场来看它仍然是一本有价值的书。它最特别之处是对按合同设计（Design By Contract）的讨论。

[Odell foundations]

Martin, J. and Odell, J. Object oriented methods: a foundation, (UML Edition), Prentice Hall, Englewood Cliffs, NJ, 1995.

这本书从概念的视角看对象，因而显得重要。是少数几本真正从分析角度看待对象的书之一。很好地叙述了多重分类和动态分类，以及事件图的细节，那是UML活动图的基础。最近它升级到UML符号，因而也成为了第一本关于UML深入使用的书。

[Odell pragmatics]

Martin, J. and Odell, J. Object oriented methods: pragmatic considerations, Prentice Hall, Englewood Cliffs, NJ, 1996.

着眼于模型向代码的转化、重用和方法工程——为开发组织建造定制方法的过程。

[PLoP 1 and 2]

Coplien, J.O. and Schmidt, D.C. Pattern Languages of Program Design, Addison-Wesley, Reading, MA, 1995.

Vlissides, J.M., Coplien, J.O. and Kerth, N.L. ed. Pattern Languages of Program Design 2, Addison-Wesley, 1996.

模式会议的会议录。范围广泛的模式，涵括了整个软件领域。

[Reenskaug]

Reenskaug, T., Wold, P. and Lehne, O.A. Working with objects: the OOram software engineering method, Manning Publications, Greenwich, CT, 1996.

从不同的侧面考察交互中的角色并用类来合成这些角色。Reenskaug给出了进行有效合成的指导方法，保证合成的模型无需经过分析就能正确地运作。通常合成工作不必经过显式的步骤就能完成，因此大部分方法注重于类而不是角色。

[Rumbaugh]

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. (1991) Object-Oriented Modeling and Design, Prentice Hall.

在这本书中Rumbaugh等人正式发表OMT，这可能是最流行的面向对象方法，而且是UML的关键基础。同样地，它是读者最多的建模文献之一，很好地阐述了类图 and 状态图。但老实说，这本书现在太旧而且过时了。

[Shlaer and Mellor recursive design]

Shlaer S and Mellor SJ, Recursive Design of an Application Independent Architecture, IEEE Software, 14:1 (1997), p61-72

目前对Shlaer/Mellor的递归设计方法及变换的最好论述。

[Wirfs-Brock]

Wirfs-Brock R., Wilkerson B. and Wiener L. (1990) *Designing Object-Oriented Software*, Prentice Hall.

这本书真的是太旧了以至不适合出现在这里，而且它已经确定是过时了。那是因为强调的是责任和对象设计的基础——即使在今天，这也经常不被充分理解。这是一本很好的CRC卡资料。

● 建模技术

用例

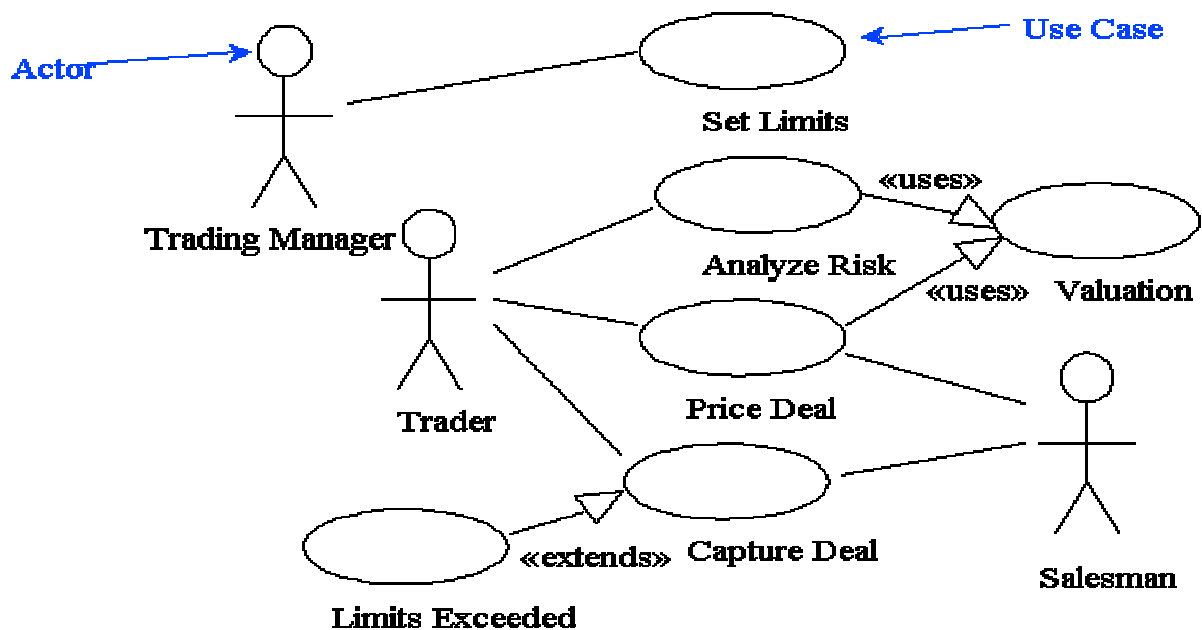
在相当长的一段时间内，对象开发者努力地要搞清楚该从哪里获得建模和开发的信息。他们常假设有某种需求文档，但又不确定这种文档该是怎样的。这里Ivar Jacobson工作成果的影响一直是非常好的。用例背后的思想并非一时冲动，而是许多人已经用了很久的普遍方法。重要的是Jacobson将它推到了舞台中央。

那什么是用例呢？这真是个问题，用例是那些难以定义的概念之一。文献[[Jacobson](#)]说它们是一个用户在与系统的对话中进行行为上相关的事务序列的时刻。通过一些例子可能会有助于理解。字处理器的用例通常会包括建立索引和插入图片。这样它们就与菜单命令对应起来。它们可以相当大（建立索引）或者相当小（设某些文本成粗体）。它们一般不会只涉及一个单独的命令。一个用例可能要保证文档中的文本格式一致，并没有命令来做这个，但正是这用例驱动了对样式单的需要。

后一个例子说明了得到好用例的难处。技巧在于识别用户的目标而不是系统功能。这样做的方法之一是将用户的业务看成一个用例，然后看看计算机系统要如何去支持它。

用例图提供一种手段去描述系统外部视图和它与外界的交互。凭这点它与传统方法的相应的图类似。在这种表示方法中外部世界用参与者（actor）代表。参与者是由不同的人或计算机系统构成的角色。对角色的强调相当重要：一个人可以扮演不同的角色，而一个角色也可以由许多人扮演。就这样，用例成了执行者与系统的典型交互。

去往讨论组→
(已超过 11,000 人)



用例可以被看成一个大的非结构集，或者是可以被以某种形式结构化。[Jacobson] 提供了两种结构化机制。第一种机制允许使用在不同用例中的行为被抽取出来放到被其它的独立用例中。这有点类似于将多个例程共用的子例程抽出来。第二种机制允许用一个用例去扩展另一个用例。新用例在原始用例之上定义了某些特点并取而代之以新的行为（这已被比作编程补丁）。扩展常用于表示异常行为和特殊实例以免让模型中的用例图变得臃肿（[Graham]用边注脚本来达到同样的目的）。用例充当交互图的结构化机制。一般来说在以后的设计中会为每个用例画一个交互图。

用例的一个大的危险是模仿用例来构造软件。用例提供系统的外部视图，而软件常常会以完全不同的方式来建构。最大的危险是把每个用例转化成一个个只在简单的数据上工作的过程控制器。在使用用例的时候切记它们只是外部视图。

什么时候使用它们

用例是面向对象开发的重要部分。在任何你要理解系统需求的时候都要用它们（实际上就是总要用了）。你是否用用例图是另一回事。若把用例以结构化列表存在数据库中它就是有价值的。每个用例都需要一个名字和几段文字来说明它。在规划演进式开发过程时它们要起核心作用。它们应该指导系统测试。

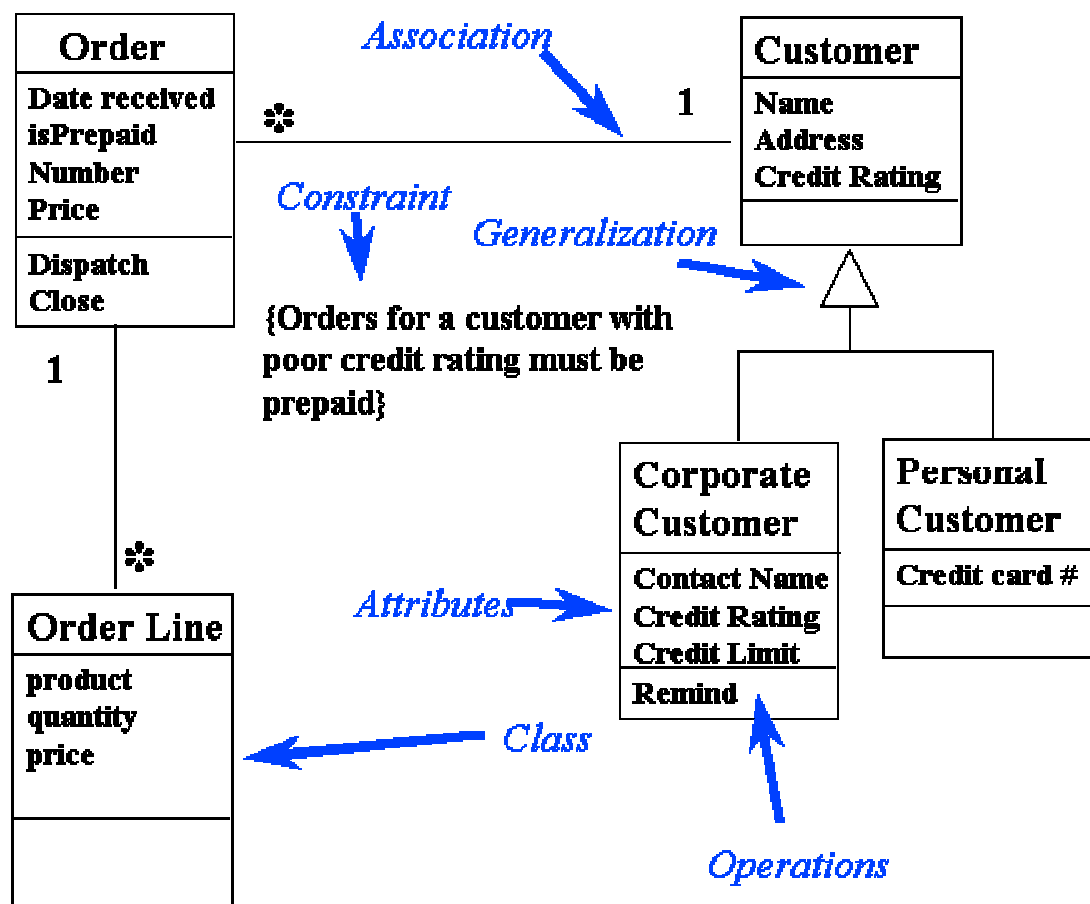
哪里可以找到更多信息

我认为这个世界仍在期待一本用例方面的真正好的书。自然[Jacobson] 作为始祖是个好的资源，但我真的忍不住要跳去推荐另一个更好的选择。[Jacobson BPR] 很实用，着眼于商务过程用例，这些用例可以证实应该要一直采用。[Graham] 也包括了一些好的观点（他使用“脚本”（script）这个术语）。

我读过的最好的论文之一是Alistair Cockburn的大作，可以在他的主页上找到。

类图(class diagrams)

类图几乎是贯穿所有面向对象技术中的核心建模技术。类图描述了系统中的各类对象以及它们之间存在的各种静态关系。其中有三类关系是很重要的：关联（一个顾客可能租了多张影碟）、子类型（护士是人中的一种）和聚合（引擎是飞机的一部分）。各种OO方法都用了不同（而且常常冲突）的术语来代表这些概念，面向对象语言的轻率极端令人灰心，但又不可避免。在这方面，UML简化了这些不同的图，带来很大的好处。在这一节中我将会用UML的术语作为主要术语，在需要时也会用到其它的术语。



视角 (Perspective)

在我开始讲述类图之前我必须指出人们在使用类图时的一个微妙之处。这一点通常没有文档记载，但对你解释一个图的方式有重要的影响，因为它确实关系到你用模型描述的所有系统。请参考[Cook和Daniels]。我认为在画类图时有三个视角可以使用（或实际上可用于任何模型，但在类图中更值得注意）。

- 概念：在这里你要画一个图来表示所分析领域的概念。这些概念自然地关联到实现它们的类，但经常不是直接映射关系。实际上模型建立时很少或者根本没有考虑用来实现的软件，而且通常是语言无关的。（Cook和Daniel称之为本质视点，而我沿用我长期使用的概念视点的用法）

- 规格化：现在我们来看软件，但只是看它的接口，并不是实现。因此我们看的是类型（Type）而不是类（Class）。面向对象开发很强调类型和类的区别，但在实践中则常被忽视。依我看，将接口（类型）和实现（类）分开来是很重要的。绝大部分的面向对象语言并没这样做，受其影响的方法也跟着这样。这种情况正在改变（JAVA和CORBA在这里有一定的影响）但是改变得并不够快。类型表示一个有许多种实现方法的接口，根据实现环境、性能特点或供应者等不同。在一些基于委托（delegation）的设计技术中这个差别将会是非常重要的，因而在[Gang of Four]中有过讨论。

- 实现：在这个视角我们已经有了类，只单单考虑实现问题。这可能是最常用的视角，但有许多方面，规格化视角者常常才是更适合采用的。

理解这些视角无论对画图还是看图都是至关重要的。在进一步讲述这门技术的时候我会着重指出这技术的每个元素是怎样依赖于视角。在你画图的时候，只从一个视角入手；当你看图的时候，注意让自己搞清楚它是从哪个视角出发的。如果你要恰当地解释那些图，这些知识是必需的。不幸的是不同视角间的界线并不清晰，而且绝大部分建模者在作图时并没有认真地分清他们的视角。

关联、属性和聚合

关联表示类型实例间的关系（某人在某公司工作，某公司有一些部门）。他们的解释随所用的视角不同而不同。从概念的角度它们表示相关类型间的概念关系。

在规格化的角度，这些负责分辨，通过访问和更新操作来明确表现。这可以意味着在订购者和顾客之间存在指针，但在封装时隐藏了。一个更面对实现的解释暗示了指针的存在。因此，为了正确地解释一个模型，知道用于建立它的视角是非常重要的。

关联可以是双向的（可以在每个方向中游历）和单向的（只能单向游历）。从概念上说所有关联都可以看成是双向的，但单向关联对于规格和实现模型有重要的意义。对于规格模型来说双向关联更为灵活但是导致更大的耦合度。对于实现模型双向关联意味着成组的指针，这对许多设计者来说都是难以应付的。

关联的一个要素是关联的基数（有时只作数量）。它指定一个人可以为几个公司工作，一个母亲能有

几个孩子，等等。这对应于实体关系图中的强制、可选、一对多、多对多等关系概念。UML已经将已有的混乱的各种方法统一成一个标准。

泛化

子类型是对用于面向对象的ER图最显而易见的补充，相当于面向对象编程中的继承。然而面向对象团体不应该忘记，对象技术占统治地位之前，子类型已经广泛应用在数据建模中了。

一个典型的子类型的例子是，考虑一家商店的个人或团体顾客。它们有所不同但是也有许多共同之处。相同点可以放在一个通用的类中并以个人顾客和团体顾客为子类。

还有，这个现象在建模的不同层次中有不同的解释。在概念上，我们可以说团体顾客是顾客的一个子类，如果所有团体顾客的实例在定义上也是顾客的实例。在规格模型中，我们会说团体顾客的接口必须和顾客的接口一致。这就是说，团体顾客可以用在顾客所用的地方，而调用者不必知道实际上出现的是子类（可替换性原则）。团体顾客可以对特定的命令有与其它顾客不同的响应，但调用者对这些区别无须关心（多态性）。

在面向对象语言中继承和子类是一种实现手段。就是说，子类继承了超类的数据和操作。这和子类型有许多相似之处，但有着一些重要的区别：

子类仅仅是实现子类型的方式（参看[Odell pragmatics]和[Fowler]）。子类也可以不用子类型，但绝大多数学者反对这种做法。新的语言 and 标准不断地试图去强调接口继承（子类型）和实现继承（子类）的区别。

约束规则

近年来，面向对象分析设计中受到更多注意的内容之一是规则。基本的观点是将人工智能中的基于规则的系统应用到面向对象建模。实际上这些在对象团体中早已出现一段时间了。Eiffel长期支持要将断言式作为按合同设计原理的一部分，但不幸地在许多面向对象方法中都被忽略了。从结构上来看，首要的断言式就是约束。这是关于类型的逻辑表达式，必须是永真的。基数表达了一些约束，但不是全部。UML用花括号{}符号来标示结构化模型上的约束。

什么时候用它们

类图是几乎所有面向对象方法的主心骨，因此你会觉察到你一直在用它们。问题是类图内涵太丰富，你难以熟练掌握。下面有一些技巧：

- 不要试图一下子用上所提供的全部符号。从简单的开始：类、关联、属性和泛化。只有

在需要的时候才引入其它的符号。

- 选出绘画模型的视角。如果你正在分析就画概念模型；在开发软件时专注于规格模型；只有在你要阐明一种特别的实现技术时才采用实现模型。

- 不要把所有的东西都放进模型中，只着重于关键的内容。宁愿只有少量你能用并保持更新的模型，也比有许多但经常忘记而且过时的模型要好。

用类图会引来的最大危险是：你很容易就会陷入实现细节的泥潭中不能自拔。要避免这种情况可以用概念或者规格视角。如果你遇到这些问题，那你就会深深体会到CRC卡是非常有用的。

哪里可以找到更多信息

在这里我的建议取决于你偏向于实现视角还是概念视角。如果是实现视角试试参看 [Booch]，而概念视角则试试 [Odell foundations]。你看完你的选择之后也看看另一种吧，两个视角都是很重要的。另外其它的面向对象书籍也会提供一些有趣的见解。我特别喜欢[Cook 和Daniels]，原因是它对视角的处理和它们提出的形式化方法。

CRC 卡

80年代末最大的对象技术中心是Portland Oregon 的Tektronix的研究室。他们是Smalltalk 的主要用户，从那里产生了许多重要的对象技术。那里Ward Cunningham和Kent Beck。他们过去和现在都热心地毫无保留地教授Smalltalk的高深知识。从如何教授对象技术的问题中引出了CRC(Class-Responsibility-Collaboration，类-责任-合作)卡的简单技术。

这种技术用4×6格的卡片来表示对象，而不是像绝大多数的方法论那样用图来设计对象。在卡上写的是责任而不是属性和方法。那什么是责任呢？这确实是一个从高层描述类的目标的方法，其思想是尽量不要去管那点数据和过程描述，而是用几句话去捕捉类的目标。卡的选择是经过深思熟虑的，你所写的不能超过卡的容量。

最后的那个C指的是合作（Collaborator）。对每个责任你要指明要与其它的哪些类合作来完成它。这让你有一定类间链接的概念，但仍然是高层的。

CRC卡是一种进取的非正式技术，也欢迎讨论。在CRC交流会上开发者挤在一起，每人拿着一张卡来说明这类是怎样参与某些用例的。通过让开发人员拿着一些东西去表示类，能使设计的全面交流更加容易。对责任而不是数据和方法的着重有助于他们避开无形的数据提供者，便于他们去从高层次理解类行为。

<i>Order</i>	
<i>Check items are in stock</i>	<i>Order Line</i>
<i>Determine the price</i>	<i>Order Line</i>
<i>Check for valid payment</i>	<i>Customer</i>
<i>Dispatch to delivery address</i>	

什么时候使用它们

有人认为CRC卡很棒，但也有人认为CRC难以捉摸。我强烈认为你应该体验一下，来看看开发组是否喜欢用它。特别是在你发现开发组太早陷入细节中，或者正在识别混在一起没有明确定义的类的时候。

用类图和交互图去捕捉和形式化CRC建模的结果。如果你正在使用类图，保证每个类有其责任描述。

注意责任清单的长度，如果它不能写在4×6表内，你肯定是没有抓住重点。

哪里可以找到更多信息

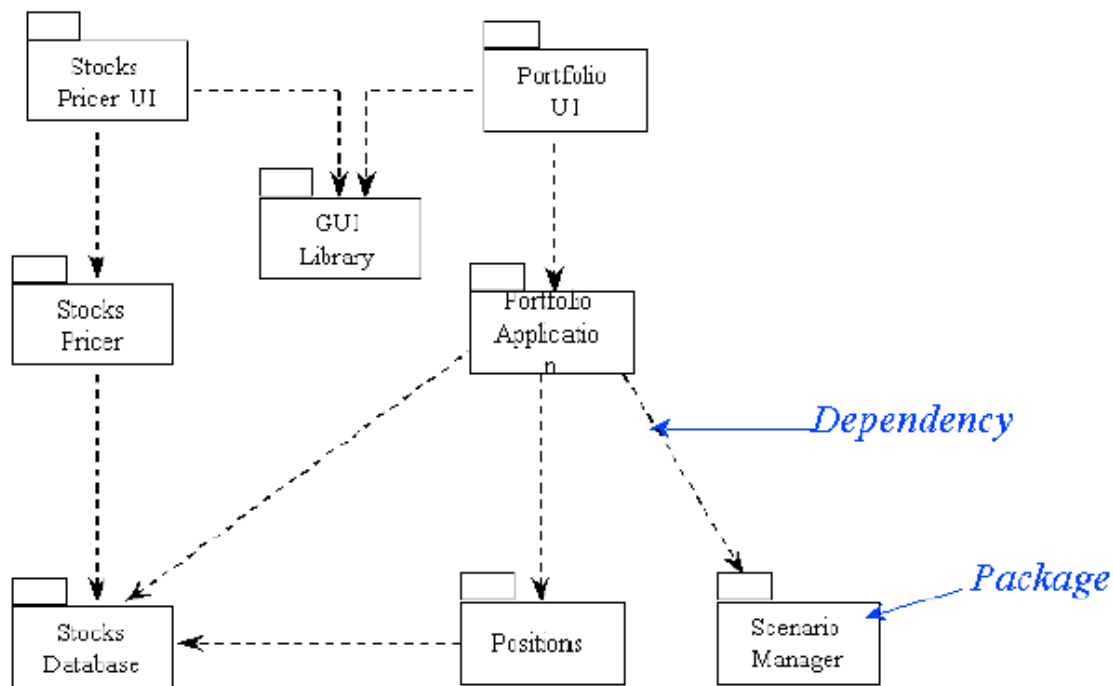
阐述了这种技术和使用责任的全部思想的最好的书是文献[Wirfs-Brock]。它是旧书不过换句话说，是成熟的。你也可以看看Beck 和Cunningham原始网页。

包图

大系统引出了特殊的难题。为一个大系统画的类图会是太大从而难以理解，因为在类之间有太多的链接，不便于理解。你可以简单地将类图砍成几页，但虽然这样可以使读图变得简单些，但以后作出的软件却可能会含有许多让代码变得难读且脆弱的链接。

有效处理这个问题的技术是UML的包，它基于Booch的类目录（class categories）。根据这种技术，每个类放入一个包中。一个类要使用同一包中的类则没有任何问题。如果要使用不同的包中的类，它必须画一条依赖线到那个包中。整个系统的图就成了包和它们之间的依赖。目标是尽量减少依赖数。

在包里面，类可以是公用(public)或私有(private)的。公用的类能够被那些具有视察权限的包看到，而私用类只能同一个包中的类看到。包可以成为全局的，即全部其它包都能看到它，这适合于通用组件如整数、字符串和集等。



这个方法的一个重要之处是，依赖不是传递的。这我是指，如果Portfolio UI依赖Portfolio Application而且Portfolio Application依赖Positions，但并不表示Portfolio UI可以用Positions中的类。实际上整个系统是一个层次结构，其中Portfolio Application向Portfolio UI屏蔽了Positions的类。

非传递性是很重要的，也是包与C++中的include、Smalltalk的需求项（pre-requisite）的重要区别。编译需求项只能是传递性的，而好的包结构采用层次思想。包的依赖和Java的包及import语句（也是非传递的）是完全一样的。比Java更强的是，如果使用类时没用import语句，依赖也存在。

什么时候用他们

包对于大型系统开发是重要的技术。对于小系统没有它你也能做得很好。包对Smalltalk没什么作用，因为Smalltalk是一个动态环境，带有能迅速找到方法层依赖的工具。

UML没有把包图作为一个独立的技术，而成当成类图里的图标之一。我更喜欢将他们看成独立的技术，但把它们组合起来显示在同一张图上常会很实用。

哪里可以找到更多信息

在所有技术中这个是方法讨论得最少的。[Booch]一般是作为最值得推荐，但它在这方面的说明太少，让人失望。叙述这个技术最好的是[Martin]，在它书中大量的例子中包含使用包（仍使用Booch类目录的旧称）的一些例子。[Fowler]也讨论了一些使用这个技术的模式。

按合同设计(Design By Contract)

按合同设计是Bertrand Meyer提出来的并且是他所开发的Eiffel语言的核心。按合同设计并不是单纯为Eiffel而设的，而是对任何程序设计语言都有价值的技术。

按合同设计的中心是断言式。断言式是一条布尔语句，永远不能取假值，如果取了假值，只可能是由于出错。典型地，断言式只在调试的时候才被检查，在产品执行的时候就不会检查。实际上程序不该假定要检查断言式。按合同设计使用三种断言式：前条件，后条件和不变式。

前条件和后条件用于操作。后条件（在这里后讲述）是说明一个操作之后环境以该是怎样的语句。例如，在定义一个求平方数操作时后条件式应该是这个形式：`result = this * this`（这里`result`是输出而`this`是操作涉及的对象）。后条件在说明要做什么时很有用，不必说明如何做，将实现和接口分离开。

前条件式是说明我们在执行操作前的环境该是怎么样的语句。因而我们要在平方前定义前条件`this > 0`。这样一个前条件说明对负数作平方运算是错误的，会导致无定义的结果。（译者注：这个例子不太妥当。）乍看来这像是个坏主意，让我们要在某个地方放些检查来保证平方运算被正确调用。重要的问题是誰负责做这件事。前条件明确指出要由调用者负责。如果没有了这条明确的规定，我们可能会作了太少的检查（两者都以为应该由对方负责）或者是太多的检查（两者都检查了）。太多的检查是一件坏事，会导致许多重复的检测代码，显著地增加了程序的复杂性。明确规定负责方有助于减少这种复杂性。调用者忘记检查的危险能够通过调试和测试时的断言式检查来减少。

从前条件和后条件的定义中我们可以看到异常的强定义。异常发生于一个操作在前条件满足而返回时后条件不满足的时候。

不变式是关于一个类的语句。例如对一个账号类可以有这样一条不变式：`balance == sum(entries.amount())`。对于类的所有实例不变式是“一直”为真的。“一直”是指一个对象可用于执行操作的任何时候。实质上这意味着将不变式加入到类所有的公用操作的前条件和后条件。在方法的执行期间不变式可能会取假值，但在其它对象对此作任何操作时必须恢复成真。

断言式在子类化扮演特别的角色。多态性的危险之一是，你可以通过与超类操作不一致的方式重定义一个子类的方法。断言式会禁止你这样做。不变式和反条件必须也应用到全部子类去。子类可以加强这些断言式但不能减弱它。乍一看这显得奇怪，但这对于实现动态绑定很重要。你将可能像对待超类的实例那样对待子类对象。如果子类的前条件加强了，超类的操作应用到子类上时就可能失败。实质上断言式只能增加子类的责任。前条件是将责任交给调用者的语句，减弱一个前条件就会强加类的责任。实际中所有

的这些有助于更好地控制子类，并帮助你保证子类的正常运行。

理想地，断言式应该包含在代码中作为接口定义的一部分。编译器应该能够在调试时打开断言式检查而在产生正式产品时去除它。可以使用不同阶段的断言式检查。前条件能通过最少的处理代价来获得最好的捕捉错误的机会。

在什么时候使用它们

无论何时你编程，按合同设计都是非常有价值的技术。它是帮助你建立清晰的接口的重要技术。只有Eiffel支持将断言式作为语言的一部分，但不幸的是，Eiffel不是一种广泛使用的语言。C++和Smalltalk增加支持断言式的机制是可能的。Java的问题就比较多。

哪里可以找到更多

[Meyer]是一本面向对象设计的书，介绍了很多断言式。你可以从ISE (Bertrand Meyer的公司)获得更多的信息。

状态转移图 (State Transition Diagrams)

状态转移图刚从面向对象建模出现时就开始用了。基本思想是定义一个有一些状态的机器（因此叫做有限状态机）。机器从外界接收事件，每个事件能引进机器从一个状态转移到另一状态。例子见图1。这里的机器是瓶厂中的一个瓶子。它开始于状态“空”。在那状态下可以接受“喷出”事件。如果“喷出”事件发生了，它则进入状态“满”，否则还是留在状态“空”（以转本自身的转移来表示）。在状态“满”的情况下“封帽”事件会导致向状态“封固”的转移。图中表明满瓶不接收“喷出”事件，空瓶不接收“封帽”事件。这样你能清楚知道什么事件能发生，和它们对对象有什么影响。

状态转移图在对象建模之前很久就存在了。它们给出清晰的而且形式化的行为定义。它的一大缺点是它意味着要定义一个系统的所有可能的状态。对于小系统它总是适用，但在大系统中会很快崩溃于状态数的指数增长。这个状态数爆炸问题使得在实际应用中的状态转移图变得过于复杂。为了解决这个状态数爆炸问题，面向对象方法单独为每个类定义状态转移图。每个类都相当简单，能够采用易于理解的状态转移图，这样就大大缓解了爆炸问题。（然而这又引来另一个问题，难以从几个独立的类图中直观显示整个系统的行为，从而指引人们去建立交互和活动模型。）

对象方法中状态转移图最著名的一种是Harel Statechart，如图1。这是由Rumbaugh引入，由Booch推广并被UML所纳。它是状态转移图最强大最灵活的一种。它的一个特别有价值的功能是状态生成，能帮

你分析出共同转移（这样我就能通过增加一个“进行中”超状态来表示“打破”事件对“满”状态和“空”状态都适用）

还有灵活的手段去处理过程。瞬间的过程（也就是说不可中断的）可以绑定到转移或或状态的进口或出口那里，这样称为动作（action）。

长的（可以被中断）过程被绑定到状态，称为活动。转移还会附带有一个条件，表示仅当条件满足时转移才发生。还可以有并发状态图，让对象有多于一个图去描述其行为。

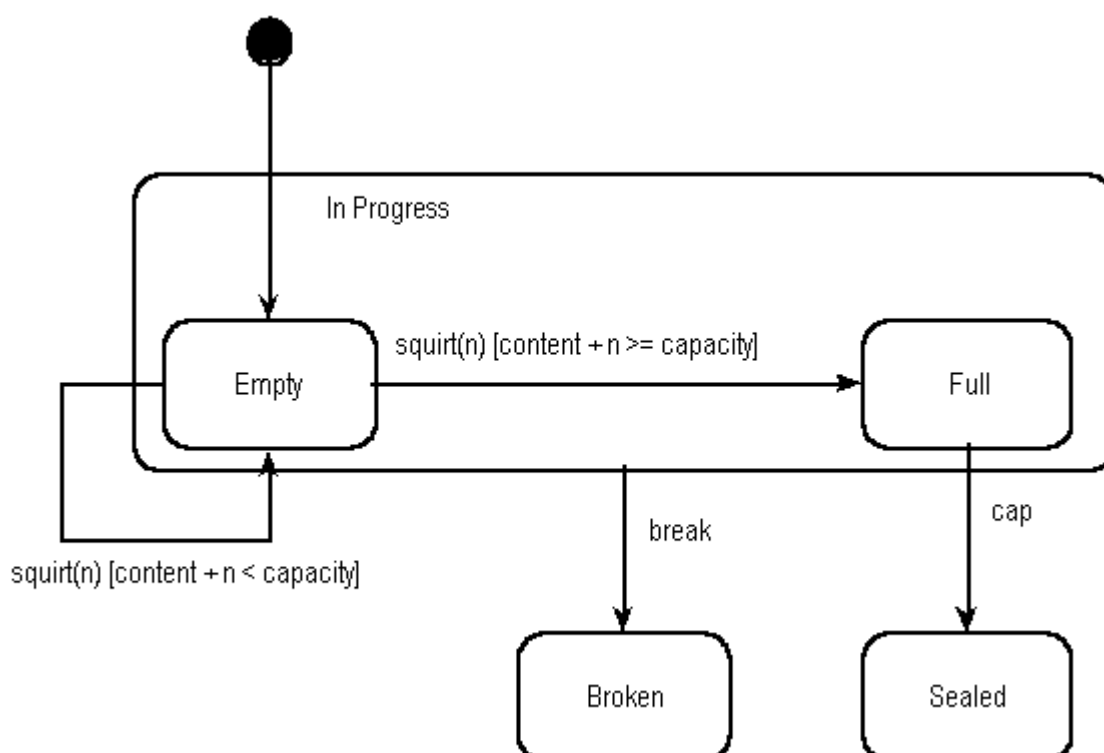


图1 Harel状态图

什么时候使用它们

状态模型是描述单个对象行为的理想方法。他们还是形式化的，工具能够建立能实现它们的实体。它们最大的局限性是不便于描述涉及几个对象的行为，对这些情况要采用交互图或活动图。

人们常常不会觉得给几个对象画状态图是描述过程的自然方式。在这些情况下，你可以试试为这个过程画一个状态图，或者使用活动图。它定义了基本行为，它们在你后来将一组对象作划分时要用到。

哪里可以找到更多信息

UML使用Harel状态表，它已经成了对象方法中最流行的状态图。作为最初的入门材料我还要推荐 [Booch]。更深入的外理方法请看看 [Cook and Daniels]；它们很详细阐述了形式化、集成按合同设计以及

结合子类使用状态图的讨论。

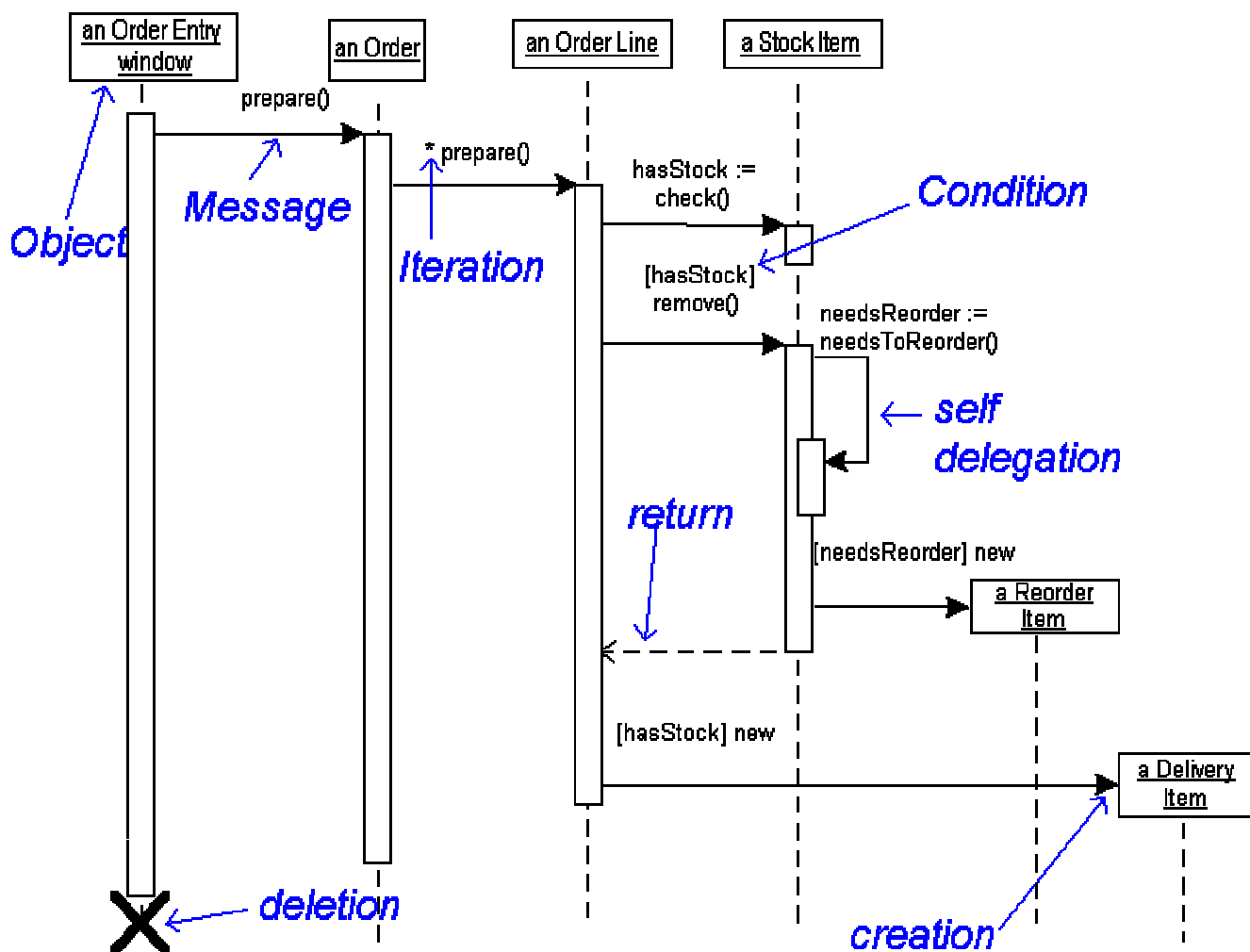
交互图

交互图是描述一组对象如何在某个行为——典型地是一个用例——中合作的模型。这些图显示一些示例对象和用例中在这些对象在传递的消息。

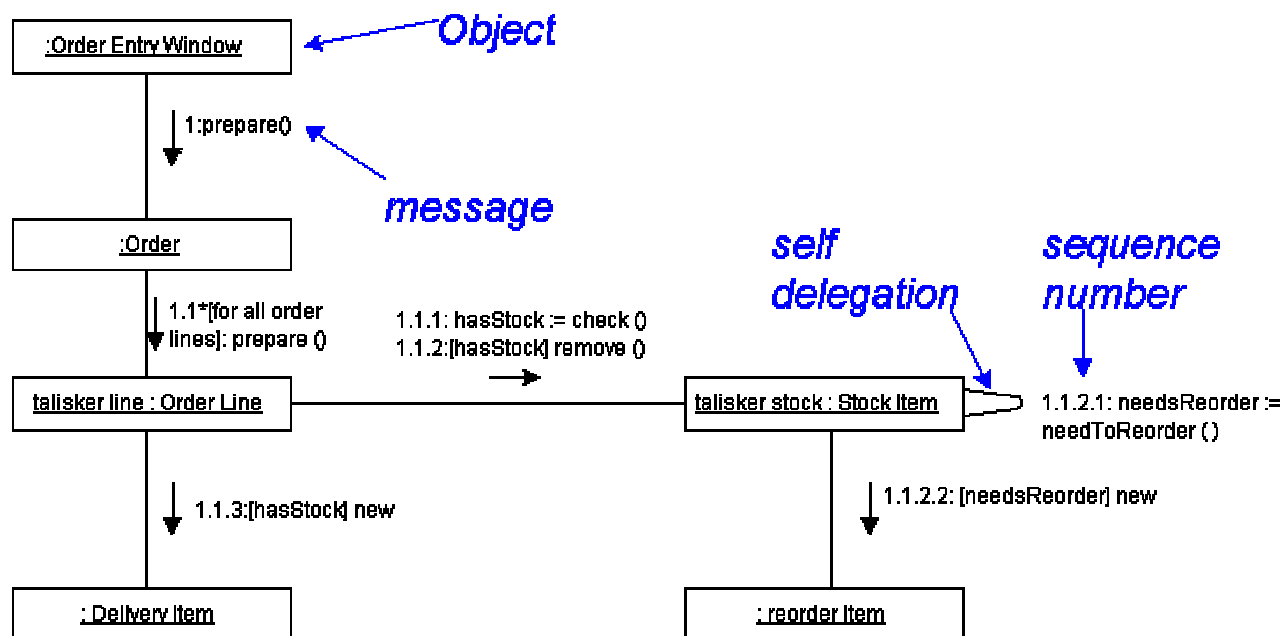
下面我要用个简单的用例来演示这种方法。在这个行为中，订单项窗口（order entry window）发送就绪消息到一个订单。然后订单发送就绪消息给每个订单队列。订单队列首先检查库存项目，如果检查返回“真”就从库存项中去掉库存。如果库存跌到低于进货线就请求新的送货。

交互图有两种形式，都在UML中。第一种形式是顺序图。在这种形式中对象表示成垂直线，消息表示它们之间成水平线。这个形式最初由Jacobson推广。

下图用UML符号来表示这种形式。消息的顺序是从上而下。



另一种交互图形式是合作图。其中示例对象显示成图标。箭头还是表示用例间发送的消息。这时顺序用数字方式来标记。简单的合作图简单地按顺序对消息编号。更复杂的图将另一个消息实现部分发送的消息用小数形式表示。另外还可用字母来表示并发线程。



交互图最大的优点之一是它的简单。要给交互图多写点东西很难，因为它太简单了。然而它们确实也有弱点，最主要是，尽管它们很适合于描述行为，但不定义它。典型地它们不会显示所有迭代和控制，来从运算意义上完整地说明计算。已经有各种各样的方法尝试和补救UML的这一点。

我对这些提高可执行性的倾向有着复杂的感情。对于我来说，交互图美妙的地方是它的简易，而这些附加的符号损害这一点去追求计算上的完备性。因此我建议你不要急于用交互图的复杂形式，你会发现更简单的反而有最大的价值。

什么时候使用它们

在你要分析一个用例中多个对象的行为时，就应该使用交互图。它们很有效地表示对象之间的合作，但并不那么适用于精确定义行为。

如果你想分析一个对象在多个用例中的行为，请使用状态转移图。如果你想分析交叉于多个用例或多线程中的行为，考虑用活动图吧。

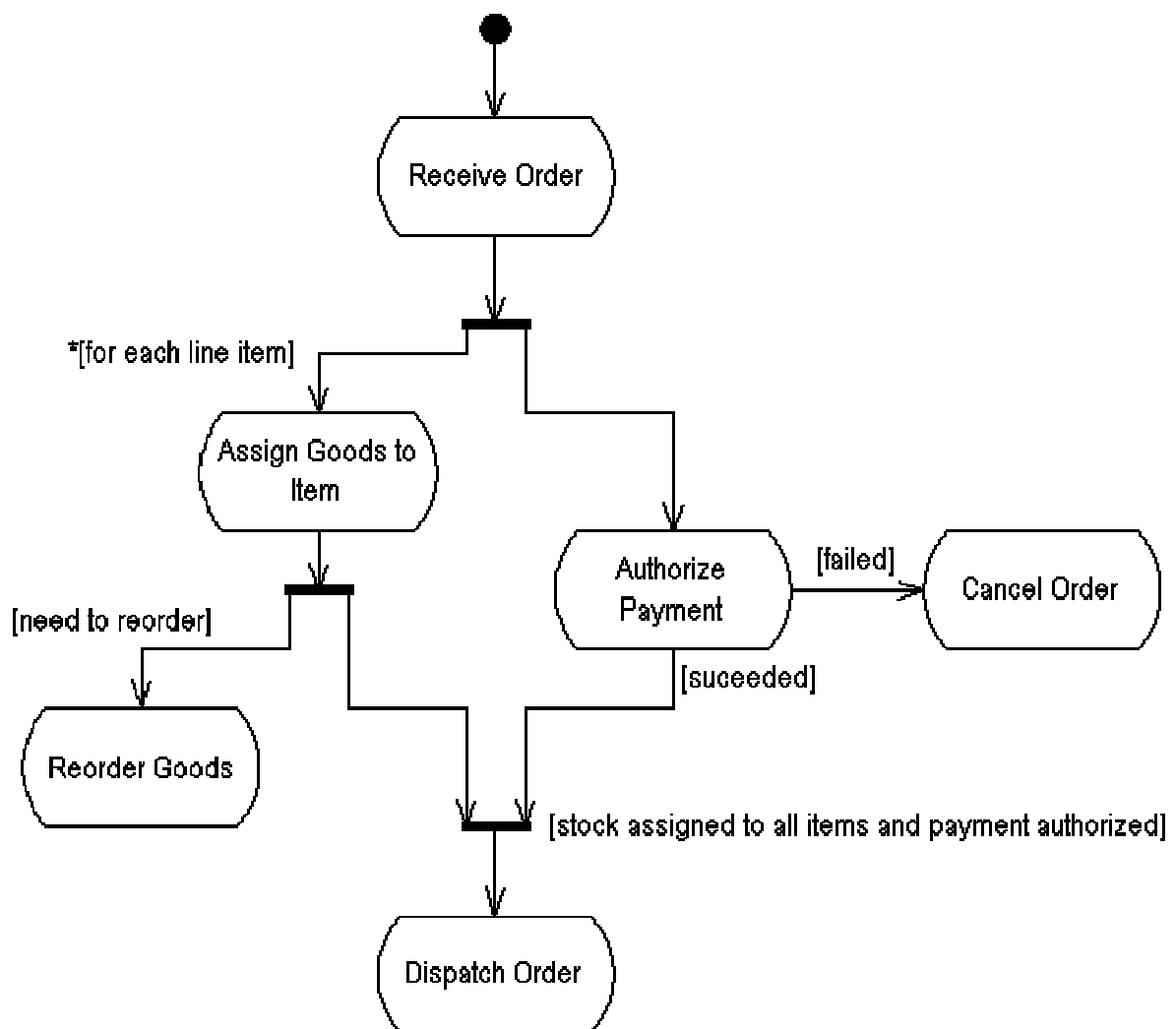
哪里可以找到更多信息

绝大部分面向对象建模的书都有阐述交互图的某种形式。目前标记符号多种多样，但它们最后都会适时地统一到UML。一个稳妥的建议是文献[Booch]，它给出了你真正需要的简单的概述。

活动图

对于许多正在看最新版UML的人来说，最不熟悉的图就是新的活动图了。它尤其陌生是由于它在Booch、Jacobson和Rumbaugh的成果中都没有出现过。事实上它是源于Odell的事件图的，尽管符号与Odell的书中的很不相同。

活动图着重描述活动和大的过程，它们可能与方法或成员函数及其活动序列相对应，也可能不对应。从这种意义上，它很像流程图，而不同之处是活动图显式支持并发活动和它们的同步。在下图中我们看到receive order活动触发了authorize payment和check line item两个活动。实际上check line item在订单中的每个line item中触发。由此如果一个订单中有三个line items，receive order活动就会触发四个并发活动，产生四个（至少是概念上）独立的线程。这些线程，和其它由receive supplies 起动的线程，在订单被发送前是同步的。



注意这个图没有终止点。活动图的特点是定义一个同步几个从外来事件的事务过程。你可以想想定义对该过程的反应的图，该过程持续到所有要做的都做完了为止。你也可以作有单一起点和终点的活动图。这样的过程图更像传统的过程调用。这样的过程图也可以使用并发行为。

活动图最大的不足之处是它们没有显式表示出哪个对象执行哪个活动以及它们之间的消息工作方式。你可以为每个活动标记上负责的对象，但那不会使对象之间的交互变得清晰（那就是你要使用交互图的时候了）。一种常能奏效的方法是，在为一个过程建模的早期，先画活动图来帮助你获得对整个过程的理解。然后你可以用交互图来帮助你将活动分配到类中。

什么时候使用它们

在你要描述一个并发的过程或者表示在多用例交互中的行为时，活动图能派上用场。

在你要表示对象如何合作去完成一个活动图时使用交互图。用状态图来表示对象在其生命期期间如何变化。

哪里可以找到更多

这里最好的资源是[Odell foundations]，在他原创的事件图的基础上给出了活动图全面的叙述。

● 过程技术

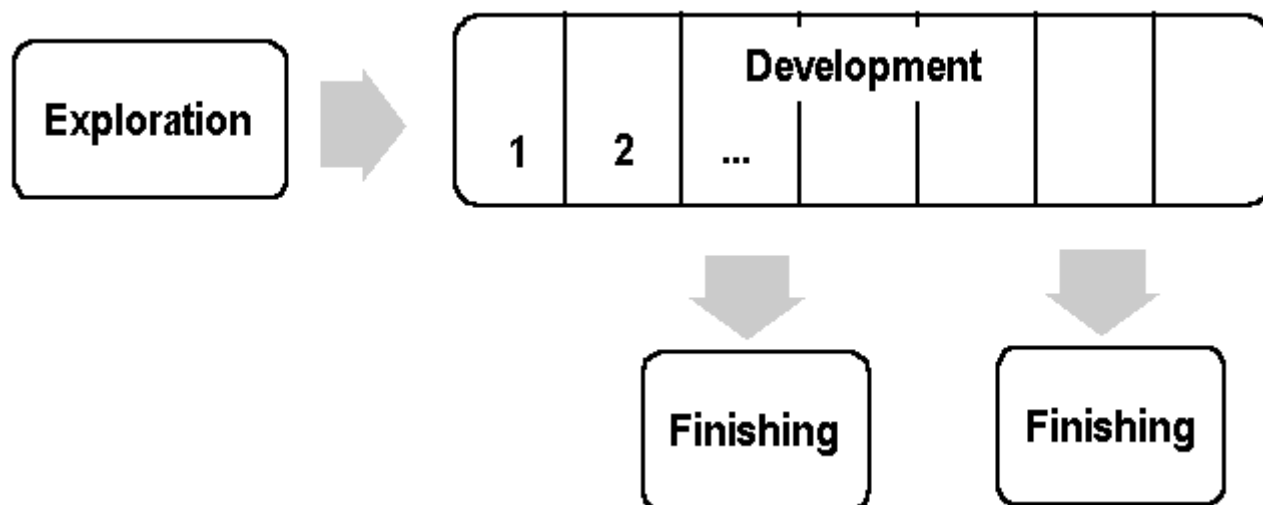
演进式开发

面向对象开发早期的书没有太多地介绍面向对象系统的开发过程。习惯于死板的方法学步骤的开发者找不到什么指导，更不用说去说约束它们了。在近几年间作者们已经更加注意过程了。一些作者作出了一个使用交付的建议活动的详细清单。在细节上它们看起来非常不同，但有一条共同的线索。而且共同点常常比不同点更有意义。

所有人都强调瀑布生命周期应该丢到故纸堆里。这种生命周期规定了分析、设计和构造的固定阶段。取而代之地他们偏好于渐进模式。渐进（也叫做重复或演化）式开发常常被当作非受控开发的一个借口。实际上演进式开发比瀑布开发能提供更多的控制，所以虽然它要花点时间来习惯，但经理和开发者都可以养成使用它的习惯。

演化开发的本质是以一系列增量形式完成的开发阶段。每次增量建立完整系统的一个子集。每次增量是一个包括了分析、设计、编码、测试和集成等阶段的完整的生命周期。它可能要交付给用户，但也不一

定。如果是要交付就需要些结尾工作：优化、打包或类似的工作。开发阶段之前还有探讨阶段。



在探讨阶段你的目标是理解那些对你最为未知的地方。典型地这个阶段包括：

- 发现用例
- 建立问题领域的概要模型
- 建立体系结构的轮廓，来描述各块主要的技术是如何整合到一起的。

通常随着这项工作的进行最后会做成一个原型，这样你就能深刻理解各块所需要的。

这个探讨过程通常约占全项目总长的1/5。重要的是在这个阶段里我们只寻找概要模型。详细的分析和设计是由后面的增量步骤来完成的。

探讨阶段衡量什么时候完成的好的标准是什么时候开发者能够以最接近的人/周数估计出设计和建造全部用例所用的时间。

第二个阶段是软件的增量交付。这个阶段划分成一系列增量。每次增量有相同的持续时间，由一个完全集成的产品的发布来标志。这个阶段产品经过一套完整测试，满足整个需要的某子集。需求的分解可以通过分配用例到各增量来实现，这样每个增量实现全部用例中的一部分。就这样项目就有了清晰可见和可测试的里程碑，可以让项目的主办者跟踪。正是这可测试性让一个经过良好计划的增量过程比一个瀑布式项目更受控制，毕竟无法用如“分析完成”这样一个不可测试的里程碑来糊弄你自己。

增量该是多长呢？绝大部分作者认为大约2-3个月，但也有一些（包括我自己）喜欢争取更短的增量期。每步骤2至3周的长度是非常有争议的，特别是使用象Smalltalk这样的快速开发语言的时候。每个增量过程包括分析、设计、编码和每个分到这个增量的用例的测试。增量的长度应该是固定的，如果事情要花比应

该的时间多，一些用例就要推迟到以后来做，而绝不更改期限。这就让项目养成遵照常规期限的习惯。

增量首先要结构化去对付最大的风险。另外，最优先的用例要先实现。一个对用户可用的发布版应该尽早地完成。

什么时候使用它们

如果你想项目成功就应该使用演进式开发。

哪里可以找到更多信息

我所喜爱的项目管理和过程的书是[Cockburn]。简短，切入要点，而又包括了所有该说的重要东西。

翻译 (Translation)

翻译 (Translation) 是Steve Mellor杜撰的一个术语，他用这个术语去强调他的方法和常见的面向对象方法的主要区别。那些常规的方法被他称为是苦心经营的方法。这个区别覆盖了你得到一个领域概念分析模型到你在特定的实现环境中把它转化成代码的过程。

通过苦心经营，你得到一个你的领域的高层分析模型，然后逐步添加你需要的细节来将它转化成工作代码。用翻译的话你只要把分析模型细化到严格和可执行的程度，而不需要增加任何实现相关的细节。设计过程反而涉及确定如何实现分析模型的构想 (*constructs*)。这些决定被收集到一个原型集中。这原型集本质上是一个编译器的规范，能将你的分析模型按你的实现环境编译。

翻译的宣扬者宣称有以下优点：

- 将领域分析和实现分离开来，能让你容易地改变实现。你可以为一个实现环境使用一个原型集，在重用领域设计模型时转用另一个原型集来生产。有些工具能解释翻译风格模型，对于这个过程相当理想。
- 翻译允许你有更专业化的技巧。那些熟练操作领域模型的人集中精神于制作一个好的分析模型而无需担心实现问题。另一些人可以专门地开发与任何特定领域无关的原型。
- 你开始了一组好的原型之后可以在以后在相同实现环境的项目中重用它们。一个流行的实现环境还会有可购买到的原型。

理论上翻译得到许多支持。关键问题是在实践中这理论有多大的作用。只有在详细分析模型比传统编程语言更容易掌握以及有足够多的原型可用的条件下，翻译才能体现它的优点。翻译的否定者认为这两点

都不可能做到。要做足够详细的分析需要足够详细的技术，通常依赖状态机，那本身就是复杂的编程。工具可以产生目标代码，但调试器也很重要而且也难制作得多。翻译方法最终成了私用的语言，公司越来越多地选取用更多标准语言而不是把自己绑死在单一个供应商上面。

一进入这个主题我们就进入到一个商业战场。Steve Mellor的公司Project Technology 提供支持基于翻译方法的工具和咨询服务。他们声称在这方面取得了巨大的成功。自然地他们的竞争者对此表示怀疑。

然而Project Technology并不是使用翻译的唯一公司。ObjectTime 使用ROOM方法。他们的建模语言更为复杂，他们的支持者称这能更好地支持这种类型的工作。它们也更着重于实时开发。Project Technology的主要经验在于实时，但他们称他们的方法是通用的。

什么时候使用它们

也没有有力的证据表明翻译像它所声称的那样好。当然它是值得看看的。根据你的项目需要来评估特定的工具。有适合你的环境的原型吗？你觉得值得去建立这样的原型吗？仔细看看调试过程，调试器通常比编译器更难制作，然而又是严谨开发的关键。

哪里可以找到更多信息

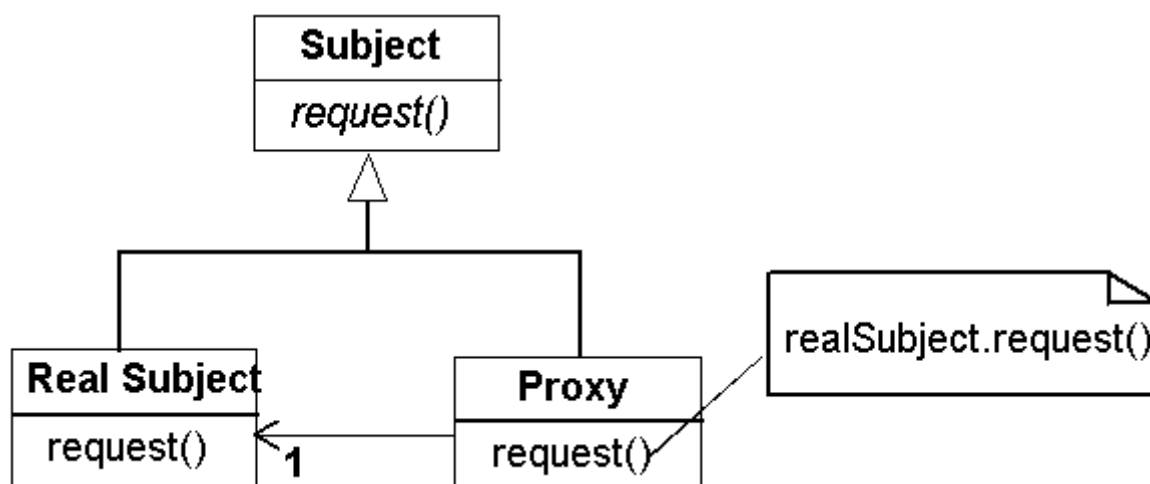
Steve Mellor 是这种方法的最大支持者，他和 Sally Shlaer 写了一个 IEEE 软件文档来阐述他们的方法 [Shlaer and Mellor recursive design]，显然这也是一本关于这方面工作的书。自然地 Project Technology 是一个这个话题的好信息源，以翻译为招牌的 ObjectTime 也是。[Odell pragmatics]讨论了一些通用的建立原型(他称之为“设计模板” (Design Templates))的方法。

模式 (Patterns)

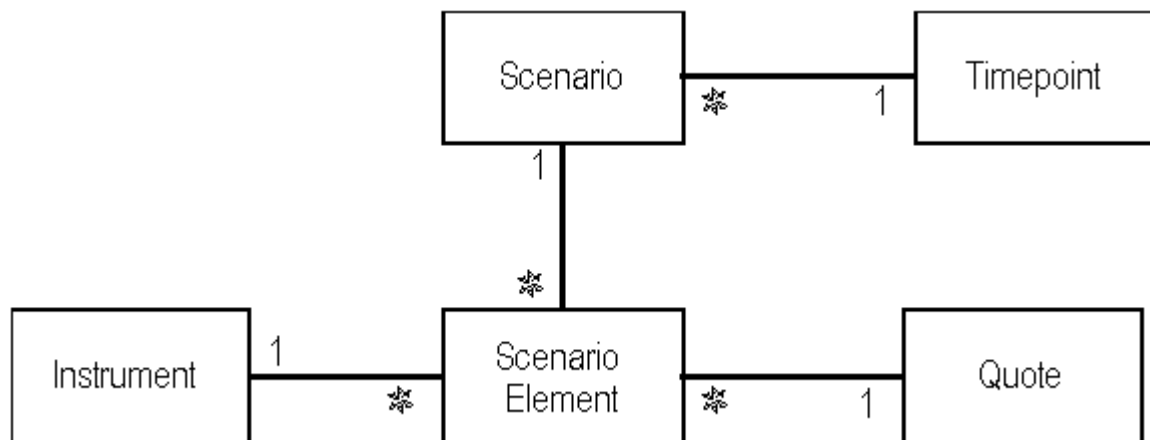
我这里谈到的绝大部分分析和设计技术是为了给你一些关于怎样表述一个面向对象设计的指导的。相反模式从过程的结果——示范模型——着眼。许多人已经指出，项目出问题是因为从事项目的人并不知道有些设计在经验更丰富的人看来是常见。模式说明了做事的通用方法。它们在人们从做过的设计中发现重复的主题时被收集起来，然后分开每个主题并描述它们，以便其他人能看到模式就知道怎样应用。

通过一个例子会更容易理解。假设你有些对象在一个桌面的进程中运行，它们要和运行在另一个进程的其它对象通信，可能在桌面上也可能在其它地方。你确实不想你系统上的对象得去关心如何在网络上找到它或执行远程过程调用。你能做的就是你的本地进程为远程对象建一个代理 (Proxy) 对象。这个代理对象又有和远程对象一样的接口。你的本地对象和代理对象通过通常的进程间 (in-process) 消息传送。

然后代理负责将消息传送给真正的对象，无论它在哪里。



代理是个设计模式，因为它说明一个设计技术。模式还可用到其它地方。假设你正在设计一个财政市场的管理风险。你需要理解一个股票证券的价值是怎样随时间变化的。你可以通过记录这股票的价格和相应的时间来做到这点。但你还要能考虑假设形势下的风险（“石油价格下跌后会怎样”）。要做到这点你可以建立一个包含股票整套价格的场景。然后你可能做分散的场景，如上周价格、下周最佳估计、石油价格下跌时的下周等等。这种场景模式是一种分析模式，描述了领域建模的片段。



模式的一个重要部分是它远不是一个模型（Model）那么简单。一个模式还必须包括为什么是那样的理由。人们常说模式是一类问题的解决方案，因而模式必须把问题交代清楚，并要解释为什么它能解决问题，以及在什么情况下有效而在什么情况下无效。喜欢编写模式文档的人们正在开展有意义的运动。这个模式团体开办会议和写了相当多的书。模式很重要，因为它们是理解语言的基础和建模技术之后的下一个阶段。

模式给你一系列解决方案，并告诉你什么构成一个好的模型，怎样动手建构一个模型。它们通过范例来教你。

在我动手的时候我老在想为什么我必须从头开始创造，为什么我没有能告诉我怎样做通用的事的手册。模式团体正在努力写这些手册。

什么时候使用它们

无论什么时候都应该使用模式。无论何时你要开发某样东西，在分析、设计、编码和项目管理时，你都应该查找任何有助于你的可用的模式。现在模式领域还年轻，因此没有太多的材料可用。因为模型团体还没确定怎样给这些材料编索引，这也是个好的工作。

哪里可以找到更多信息

首要的模式信息源是Web上的Patterns Home Page，它能给你所有书、会议之类的信息。一些模式和相应信息存放在Ward Cunningham的Portland Patterns Repository。

模式方面最在影响的书是 [Gang of Four]，这是一本关于设计模式的书。另一本重要的书是模式编程语言（Pattern Languages of Programming PLoP）会议论文集[PLoP1 and 2]。我也不故作谦虚，推荐分析模式方面的[Fowler]。

再加工（Refactoring）

你见到过软件熵原理吗？它暗示出，程序从设计良好的状态开始，然后新的功能一点点加进去从而损失了原先良好设计的结构，最后变成意大利面般的一团糟。其中的一部分就是规模化，你写了个完成特定工作的小程序，人们叫你增强它，让它变得更复杂。尽管你试图明了设计但这种窘况还会发生，因为你做的越多，程序就变得越复杂。

造成这个状况的原因之一是，当你要加入新的功能时，常常是以原来的程序没有设想到的方式在原有程序的上面更改。在这种情形下你可以重新设计你的程序来更好地支持你的更改，或者按你要增加的拐弯抹角地处理。虽然理论是重新设计你的程序会更好，但这通常会增加额外的工作，因为重写原有程序会引来新的bug和问题。记住老工程师的格言：“如果它没崩溃就不要修理它。”然而如果你不重新设计你的程序，新增部分会比应有的更加复杂。渐渐地这额外的复杂性会变成严重的惩罚。因此你就是作出变通：重新设计是用短痛换取长期的效益。在进度的压力下，绝大多数的人宁愿把痛苦推延到以后。

再加工（Refactoring）这个术语用于描述一种减少重新设计的短痛的技术。在你进行再加工的时候你并没有改变你的程序的功能，而是改变它的内部结构使之更容易理解和加工。再加工更改通常是小的步骤，比如重命名一个方法，将字段从一个类移到另一个类，或者将两个类似的方法统一到超类中。每个步骤都

很小，但几个小时之后你就能将程序大大改善。

下面的原则有助于再加工的进行：

- 不要同时作再加工和增加功能。工作的时候将它们明确分开。你可以在短步骤之间交换进行：半小时的再加工，一小时的增加功能，再半小时的再加工刚才新增的代码。
- 保证你在开始再加工前作了有效测试。尽可能频繁地作测试，这样如果你的更改破坏了什么时，你能马上知道。
- 用预备好的短步骤：将一个字段从一个类移到另一个，融合相类似的方法到超类中。再加工常是作许多局部的变化来产生大规模的变化。如果你保持用小步骤，并且每步之后作测试，就能避免延长调试。

什么时候要再加工？

- 当你要向程序中增加功能但又发现旧的代码挡着去路时。如果这开始变成一个问题，停止增加功能，先再加工旧代码。
- 当你读代码而又觉得难以理解的时候。再加工是帮助你理解代码的好方法，而且能将这理解保留到以后。

你会常常发现自己想要再加工别人写的代码。你这样做的时候，和代码的作者一起进行。用别人能轻易理解的方式写代码是很难的。

再加工最好的方式是，让另一个人尽力理解代码，并将你的理解和他们的不熟悉结合在一起。

什么时候使用它们

再加工是门远未充分应用的技术。它才刚被认识到，主要是在Smalltalk群体。然而我相信它是改良软件开发的关键技术，无论用什么环境。让你的顾问教你的开发者这技术，保证他们理解如何去有序地再加工。

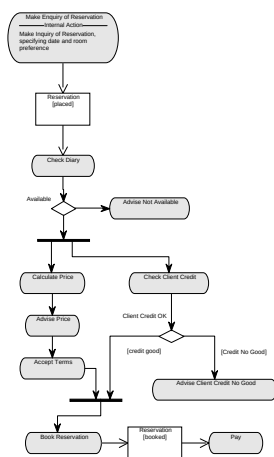
哪里可以找到更多信息

因为再加工还是非常新的技术，关于它的写得不多。William Opdyke's 博士论文可能是这方面最新的资料，但它是针对再加工工具而不是人们现在所能用的技术。Kent Beck是再加工的倡导先驱之一，他的模式书[Beck]包括许多以再加工为核心的模式。

如果你用VisualWorks Smalltalk，你可以下载一个支持再加工的工具Refactory。

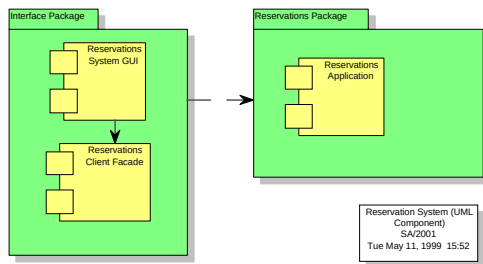
UML Activity Diagrams

(UML 活动图)，可以完整地描述企业的工作流程，即人员完成工作所需经历的步骤，能补足 Class diagram 不足之处。它最主要的价值是去发现所有的并程序，以排除不必要的商业程序。

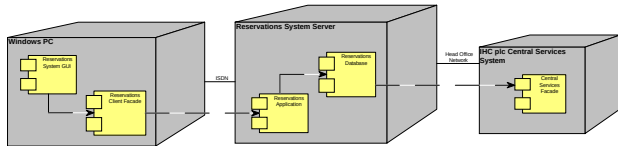


Implementation

A Component Diagram (构成图) 用来塑造软体结构的模型，包含附属在软体间的元件、二进位码元件及可执行的元件。

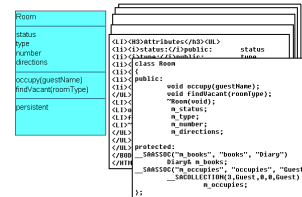


Deployment Diagrams (配置图) 用来模拟处在执行状态的元素及其所涵盖的软体的元件、程序及物件的配置情况。在这图型里，可以模拟实体结点 (Physical Nodes) 及结点间沟通的联结关系。



Generating Code

经由 Class Diagram 可以自动产生多种程式码；如：C++，Java，Visual Basic，HTML，CORBA IDL，Smalltalk，及 Delphi Object Pascal。



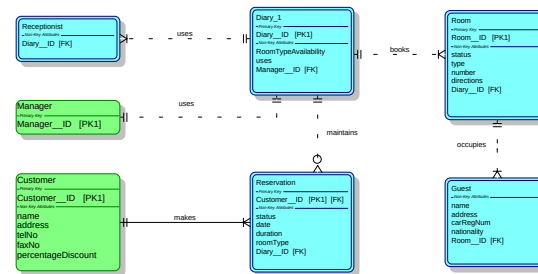
Reversing Code

System Architect 2001 可提供 C++ 及 Java 程式码的逆向工程。



Generating an ERD

您可以利用 UML Class diagram 自动产生 Entity Relation Diagram (实体关系图)。Classes (类别) 的设计是与 Entity (实体) 相对映。Class Attributes (类别属性) 与实体属性相对映。



System Architect 2001

UML Object Oriented

物件导向系统

简介



戊丰科技股份有限公司

台北市敦化南路一段 21 号 4 楼之二

Tel: (02)2570-1877 Fax: (02)2570-1876

Methodology Outline

System Architect 2001 支援数种物件导向的方法论，包括：OMT/Rumbaugh, Booch, Shlaer/Mellor, and Coad/Yourdon, 另也支援 Jacobson Use Case Diagram 以及 UML。

The **Unified Modeling Language (UML)**.



Unified Modeling Language (统一塑模语言) 是一个标示物件导向系统分析和设计的发展标准。System Architect 2001 支援 UML 1.3 的规格。

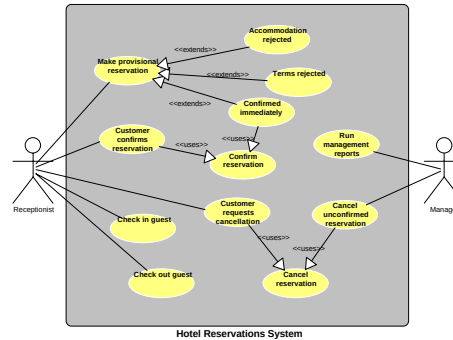
虽然 UML 并不是规定一个发展物件导向的标准程序或方法，但它是指定一个标准的图型及标记的组合，强迫使用者塑造确切的系统模型。

System Architect 2001 Features

- 32 位元使用者界面，及完整的企业模型整合工具。
- 多合一的电脑辅助设计工具 (CASE tool)，包含商业描述、物件导向、元件设计及资料模型。
- 支援的法论包括：UML, IDEF0, IDEF3, IDEF1X, Yourdon, Ward-Mellor, Gane & Sarson, SSADM, Catalyst, OMT, Booch and Shlaer/Mellor
- 支援多人使用的网路版本，整合 Microsoft VBA、互动式关联矩阵(Matrix)及影响分析。
- 可以产生 Microsoft Word 文件及 HTML 网页文件。
- C++, Java, Dynasty & Magic – Generation & Reversing, HTML Generation, Simulation.

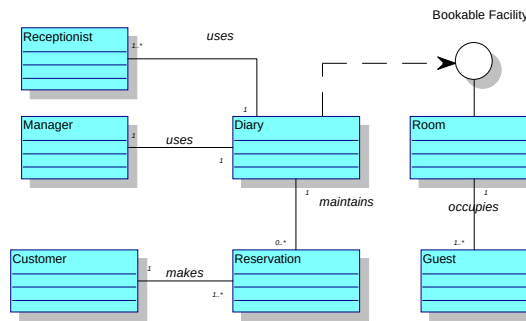
Business-Level Analysis

Use Case 技术可以获得记录该系统或企业目前所做的事及应该做的事的资讯。Use Case 技术可帮助您建立系统作业流程的架构模型。它是一个通向物件导向系统分析的绝佳方法。



Static Analysis & Design

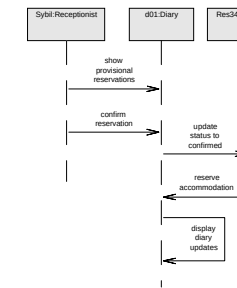
The Class Diagram 是一个系统主要的静态分析图，在这图型里，利用类别和继承的关系，释明了系统的类别结构。



System Architect 2001 支援物件间沟通的介面符号，但沟通介面无法执行(Implementation)，它们缺少属性(Attributes)、状态(States)或关联(Association)；只有操作的方法(Operations)。一个沟通介面 (Interface) 可能以印有"Interface"的矩形类别 (Class) 符号显示，或是以类似棒棒糖的符号来表示。

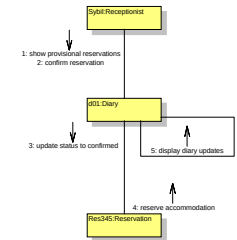
Dynamic Analysis & Design

UML Sequence Diagram 是用来塑造系统架构执行 (Implementation) 细节的模型。UML Sequence Diagram 表达出物件间交换的讯息 (Message) 以产生期望的结果。



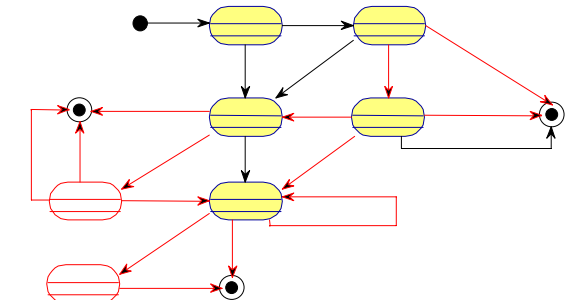
两个物件间沟通的讯息 (Message)，由被呼叫的方法(Method Called)所定义；它是由传送的物件(Sending Object)指向接收物件(Receiving Object)；被呼叫的方法 (Method) 必须属于接收物件(Receiving Object)中所定义类别实例 (Instance of Class)。

UML Collaboration Diagram (UML 联合图) 是描绘物件间如何交互影响的模型架构。System Architect 2001 会自动同步给予 Collaboration Diagram 及 Sequence Diagram 相同的图型名称。



State Diagram (状态图) 是描绘系统运转的状态。状态图是描绘系统内物件的每一个类别的典型，包含重要的动态变化。

状态图 (State Diagram) 能利用父子图型的连结 (Parent/Child Link) 附属于类别图 (Class Diagram) 里的类别符号 (Class Symbol)。



数字录音机设计方案：

一个使用统一建模语言（UML）设计嵌入式系统的实例

[Ivan Porres Paltor](#), [Johan Lilius](#) 著, [乐林峰](#) 译

摘要

本文内容是采用统一建模语言（UML）对一个嵌入式系统进行面向对象分析和设计的实例。被分析的系统是一个数字录音机或称口授留声机。设计实现采用一个嵌入式微处理器和 C++ 编码。

关键字：面向对象的设计，UML，嵌入式系统

1 简介

统一建模语言(Unified Modeling Language-UML)[BJR1,BJR2]提供了一套标准的符号来表示面向对象的软件分析和设计方法[CY90,MO92,SS95]。使用 UML 图表能够对很复杂的系统建模，包括实时嵌入式系统。

然而，UML 并不是软件开发过程，也没有严格区分软件工程中各开发阶段。

UML 定义了一套由不同图表组成的标记符号，但是并没有描述怎样创建和应用这些图表。[Dou98] 介绍了使用 UML 和面向对象的分析设计技术建造实时嵌入式系统。本文由 Douglass 撰写，描述了使用 UML 对一个数字录音机（或口授留声机）系统进行面向对象设计和实现的过程。

目前市场上已经有一些数字录音机商业化产品。本文我们描述的模型参照了一家著名厂商的类似产品的设计。相关需求说明参见本文第二节。第三节讨论系统的对象模型给出主类图。第四节继续进行面向对象的分析，但是主要讨论每个对象内部的行为。第五节进行系统结构设计。包括数字录音机的硬件结构，以及并行处理模型，在该模型内，为每个对象将分配一个可执行的线程。第六节将定义不同对象之间的合

作关系。并通过设计模式[GHJV95] 将分析阶段定义类整合。详细设计在第七节进行讨论。最后，第八节讨论系统实现。我们使用 C++进行软件的开发，硬件平台采用一个 32 位 RISC(精简指令集计算机)嵌入处理器。

2 需求分析

数字录音机是一个电子消费产品，用来录制和回放语音。通过内置的麦克录制语音信息，然后将信息保存在数字内存中。使用者可以很方便的在任何时候回放任意录制好的信息，声音将通过设备前端的扬声器播放。该产品将具有体积小、重量轻、使用方便等特点。图 2.1 是我们设计的数字录音机的外观示意图。它是一个手持系统，具有一块平面显示屏以及一些按钮。

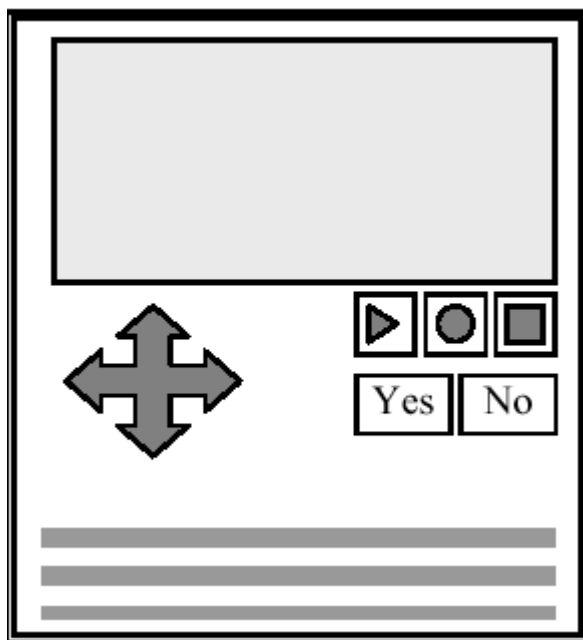


图 2.1 外观示意图

在设计中将要考虑到的系统特性如下：

- 最多存储10条信息，每条信息的长度将受系统剩余内存的限制。
- 简单易用的屏幕菜单。
- 直接存取任意语音信息。
- 具有时钟功能（不存在2000年问题），用户可以每天设置闹钟，闹钟铃声60秒钟后自动停止，用户也可以按任意键停止铃声。

- 全功能液晶显示屏. 屏幕上总是显示当前日期和时间。同时显示提示信息如操作指南, 当前操作信息等。
- 电池电量显示, 如果电量不足, 系统会发出哔哔声警告。
- 为了节省电池. 在不使用的情况下, 系统将关闭外部设备的电源, 用户可以通过按任意键使系统回复正常。
- 提供好的声音回放质量。使用6Khz的8位采样频率。

2.1 外部事件

嵌入式系统经常要跟外部环境交互。在目前的分析阶段, 我们把系统视为一个可以响应外部环境的请求和消息的黑盒子。 外部环境由一些参与者 (Actor) 组成。每个参与者与我们的系统进行不同目的的交互并交换一组不同的信息。

系统范围图/Context-Level Diagram

图 2.2 显示了所有与我们的系统进行交互的参与者。图中我们定义了3个参与者: 用户(User), 电池(Battery)和时钟 (time)。 图中还显示了接口和用于实现系统和参与者交换信息的传感装置和执行装置。这里指的信息可以是参与者向系统提出的请求, 也可以是系统作出的响应或提供给参与者的服务内容。传感装置(Sensors)包括麦克和按钮。执行装置(Actuators)包括扬声器、电池电量显示表。

每当一个时间计时结束时, 参与者‘时钟’向系统发送一个内容类似于‘下一秒’或者‘下一个小时’的消息。在实际实现中, 我们将用一个硬件定时器来控制具体时间。

umlchina 提供 UML/OOAD 培训

通过讲述一个真实 N-Tier 系统的开发过程, 使学员自然领会 OO 技术。概念并不按部就班讲授, 只是由实例带出。

详情请联系 think@umlchina.com

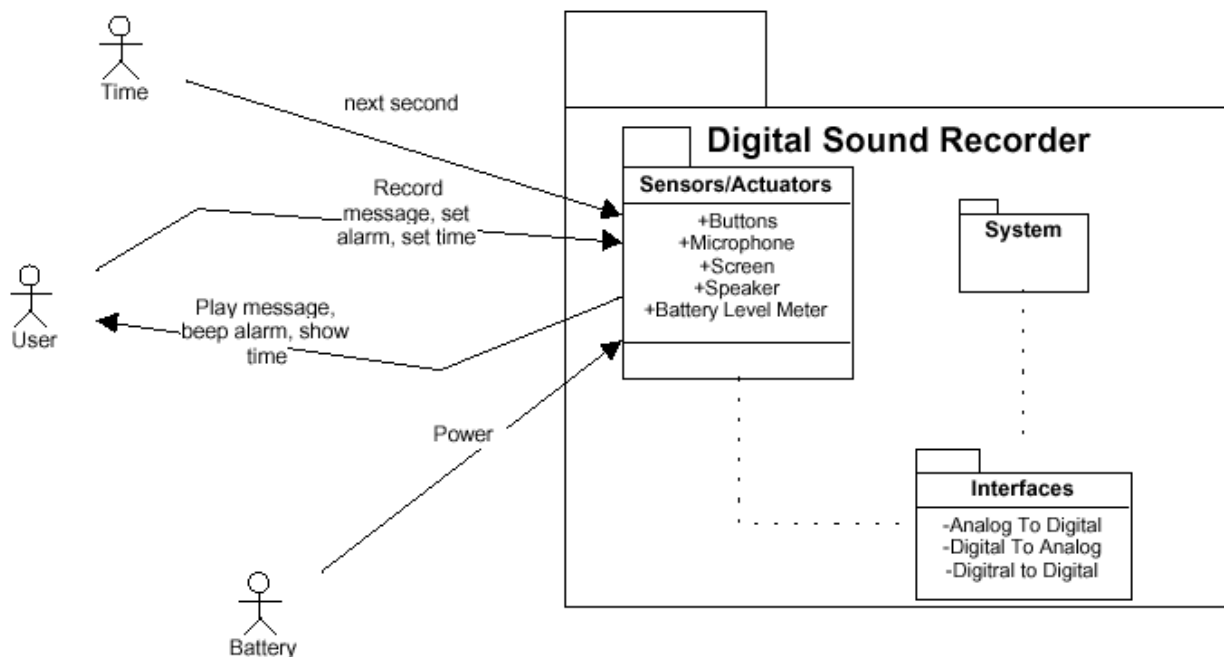


图 2.2 系统范围图

事件/Events

事件是环境与系统之间传递的重要消息。一个实时响应系统必须在某个限定的时间内对外部事件作出响应。下表中内容显示了针对我们的系统所有可能发生的外部事件。

事件的方向：‘In’表示该事件由环境传递给系统，‘Out’表相反。

事件发生模式(A)：

周期事件 (Periodic):表示该事件的发生是遵循某个固定的时间周期的。在表格中用P表示。

随机事件: (Episodic): 表示该事件随机发生。在表格中用E表示。

响应时间: 表示系统对该事件作出响应所需时间的上限，如果在给出的时间内系统没有完成响应，将会引起系统错误。

No	事件	系统响应	方向	A	响应时间
1	某一秒计时结束	a,更新内部时钟 b,检查闹钟设置 c,更新时钟显示 d,更新任务条	In	P	0.5秒
2	一个采样周期结束	a,回放或记录下个采样	In	P	1/2周期
3	用户按下某个按钮	A,显示任务条。 B,开始录音或回放一条记录	In	E	0.5秒
4	用户按‘停止’按钮	A,停止当前任务。 B,更新屏幕内容	In	E	0.5秒
5	电量不足告警	A,警告用户并终止当前任务	In	E	1秒
6	进入等待模式	A,关闭显示器	In	E	1秒
7	当系统处于等待模式时, 用户按任意键唤醒系统	A,离开等待模式, 打开显示器以 及其它外设。	In	E	1秒

2.2 用例/Use Cases

用例是从使用者的角度描述系统的功能。使用者可能是一个人或者某个设备。每个用例表示一种使用系统的方法，每个用例完成后产生不同的结果。在我们的系统中共有6个用例。如图 2.3所示。用例描述如下

记录一条信息（Record a message）

使用者选中信息存储目录中的某个信息存储位置，并按下‘录音’（Recording）键。如果在该位置已经存储了一条信息，该记录将被删除。系统开始通过麦克录音，直到用户按‘停止’（Stop）键或者内存耗尽。

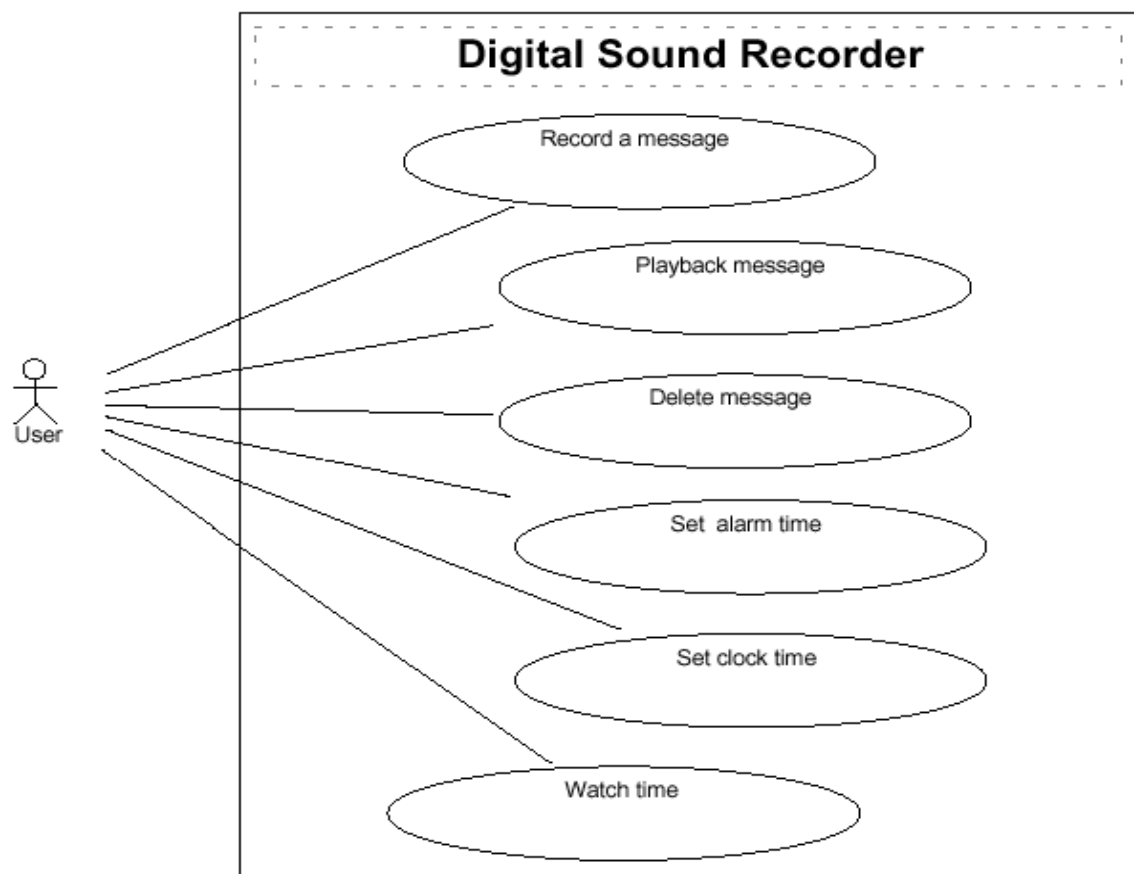


图 2.3 用例图

回放一条信息(Playback a message)

用户选择某个信息存储位置，然后按‘播放’（Play）按钮。如果该位置保存有信息，则系统通过扬声器回放该声音信息，直到回放完毕或用户按了‘停止’按钮。

删除一条信息（Delete a message）

用户选中一个存储位置按‘删除’（Delete）按钮，该位置存储的信息将被永久删除，所占用的内存将被释放。

闹钟定时（Set the alarm time）

用户可以设定是否开启（关闭）闹钟，以及闹铃时间。这些操作将在闹钟菜单中进行。

设定系统时钟/Set the clock time

用户可以通过此功能调整校准系统时间，或者将系统时间改为当前时区的时间。

显示系统时间/Watch the time

用户可查看显示屏上连续显示的当前时间、日期。

2.3 场景/Scenarios

场景用来描述系统与外部参与者（user,battery和time）之间的交互行为。一般来说，即使在每一个参与者的角色都很明确的情况下，仍然很难将所有可能的交互行为都考虑到，事实上也没有必要这样做。我们只要研究必须的场景以满足系统需要就行了，比如，我们必须研究在系统回放声音的时候电池电量不足以及当系统播放声音时闹钟响起来这两种情况。图 2.4 表示了播放声音信息的用例。

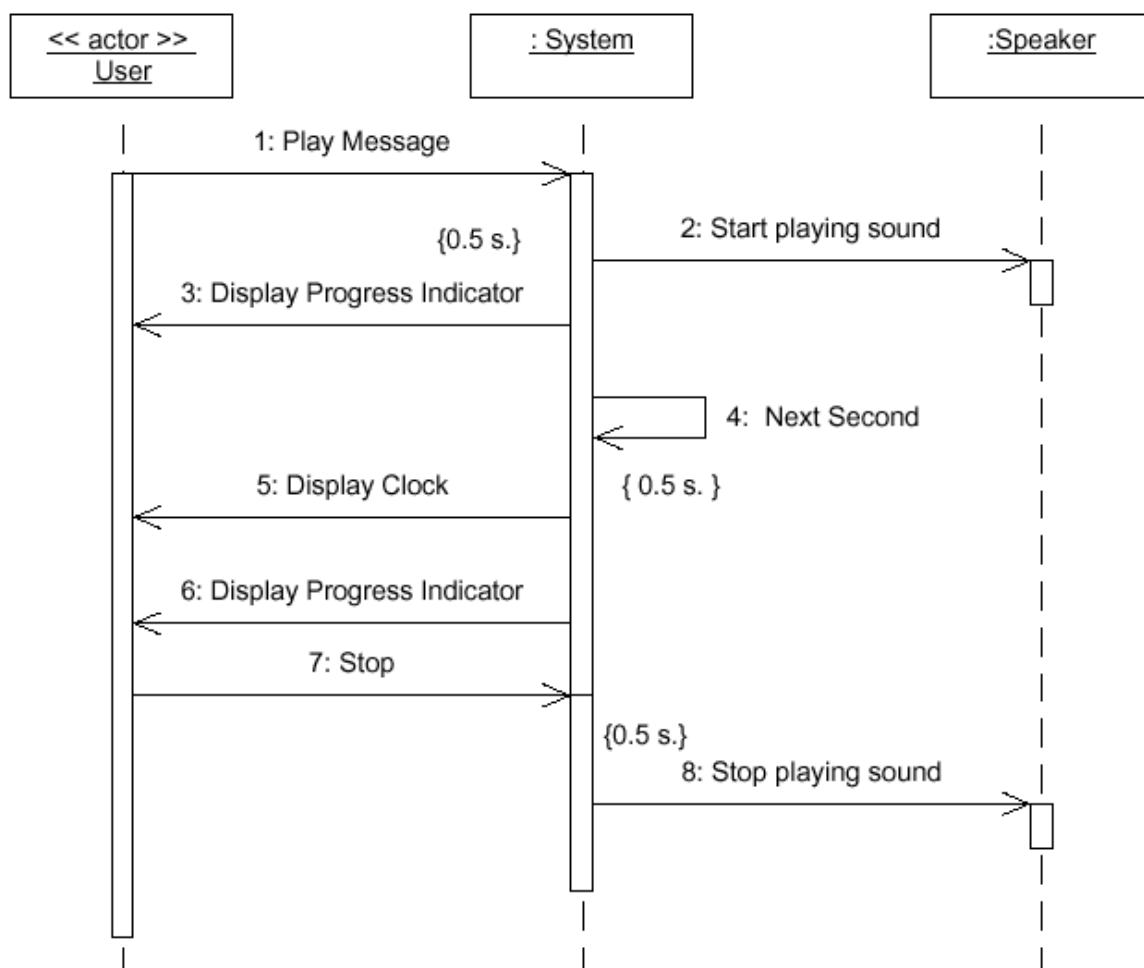


图 2.4 播放声音信息的场景

图 2.5 显示当播放声音时闹钟响起来的情况，我们决定给闹钟响铃一个较高的优先级。

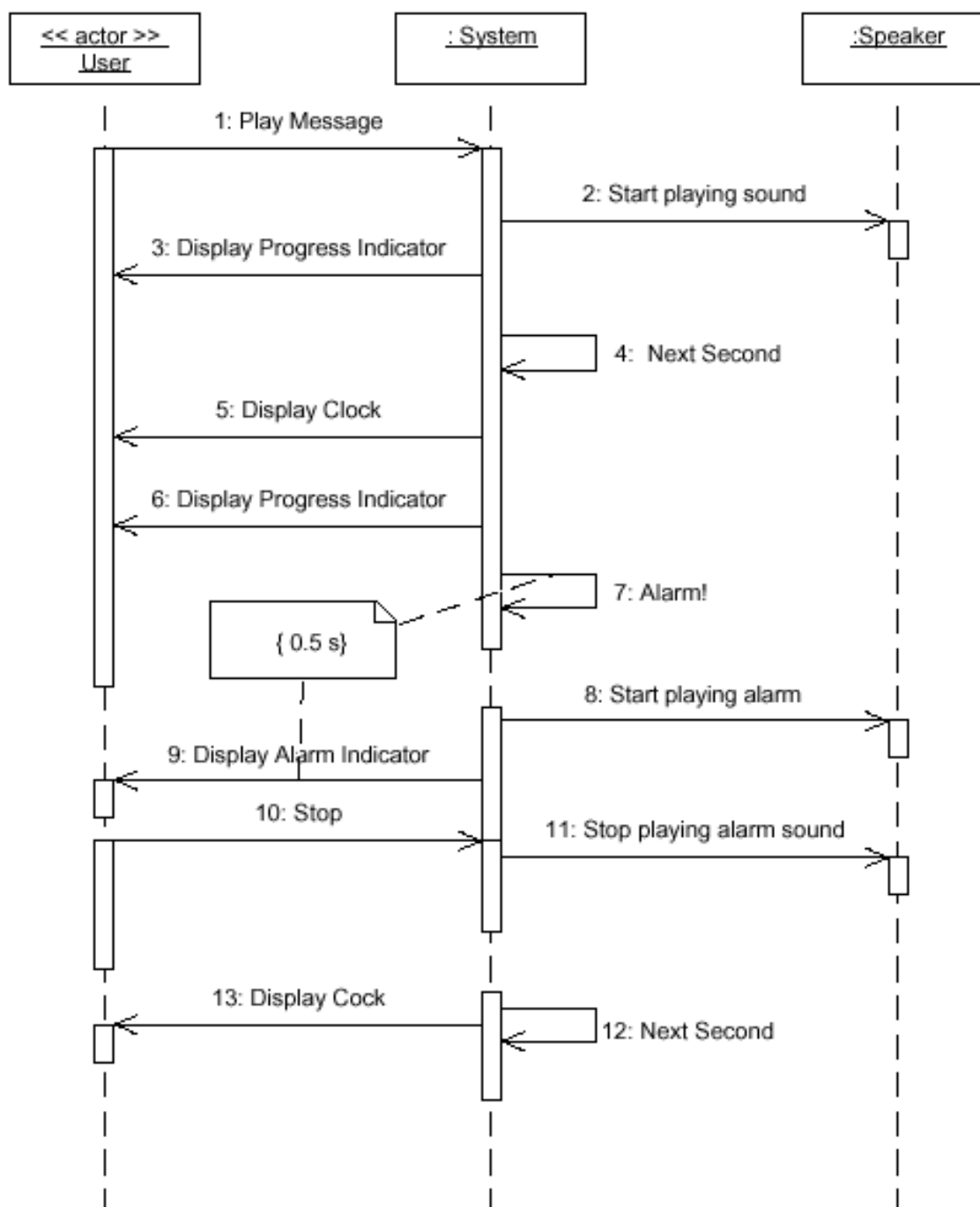


图 2.5 回放声音时闹钟响铃的场景描述

系统可以关闭显示屏背景灯、麦克和扬声器。这些部件都是要耗费大量电池电量的。通过关闭它们来接生能源以及延长电池的使用寿命。电池在电量不足的时候能够向系统发出警告，系统会关闭所有外设并切换到等待模式。用户为电池充电后，系统离开等待模式。当系统处于等待模式时，所有信息会被保存在内存中。图 2.6 显示系统进入等待模式，被闹钟唤醒，然后收到电量不足的警告，重新进入等待模式的场景。

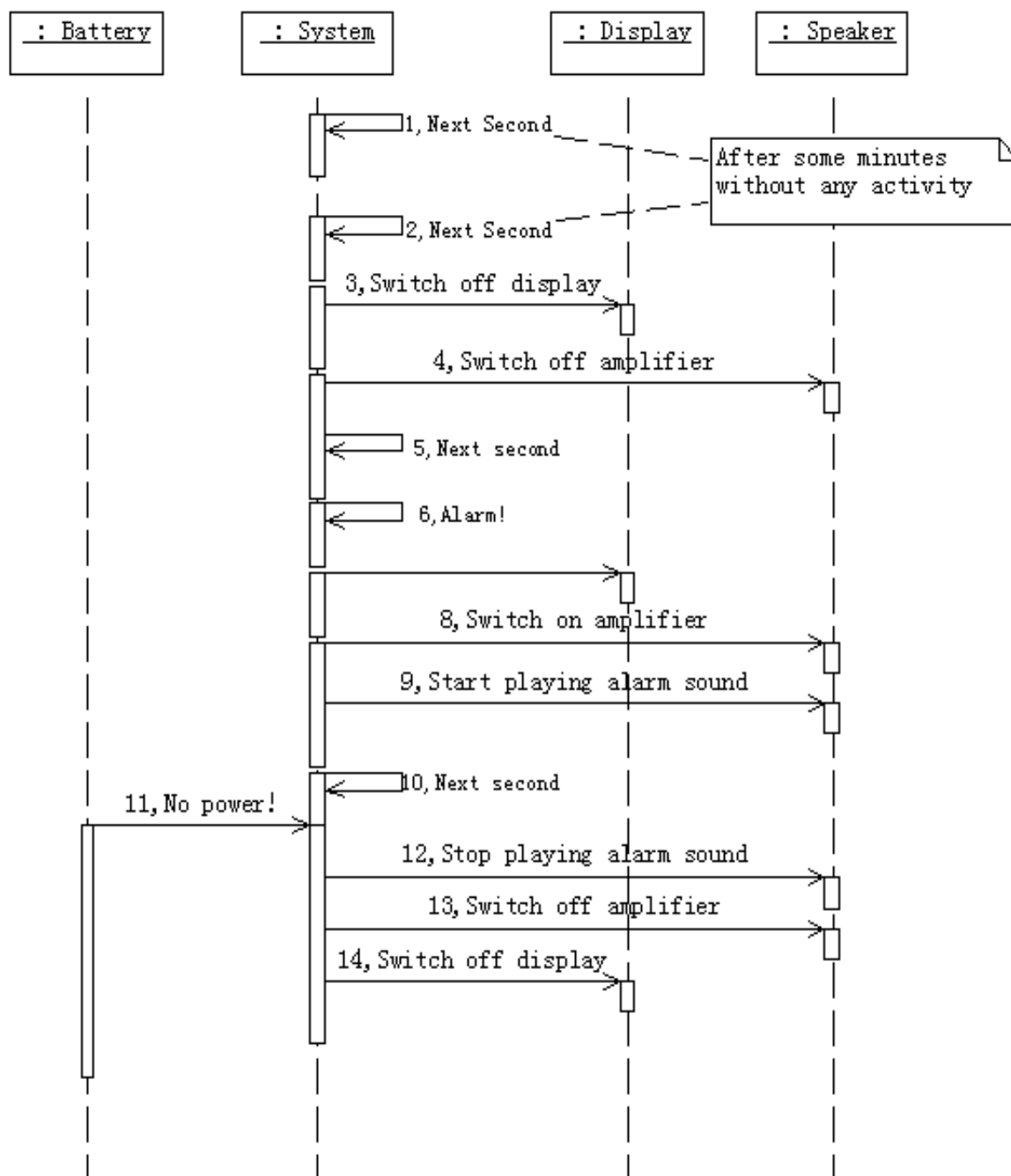


图 2.6 进入和退出休眠模式的场景描述

3 分析：对象结构/Analysis: Object structure

在需求分析之后[SS95]，我们将开始系统的领域分析。在这个阶段，我们会根据已有的需求，给出问题整体解决方案的类图。图中将包含系统主要的类及其相互关系，但是不包含它们之间接口的描述。

建立主类图的第一步是标识出系统的相关对象。

3.1 对象标识/Identifying objects

活动对象	声音信息的播放和录制设备 一盒数字磁带
	按钮，屏幕 用户通过按钮操作系统。屏幕将显示菜单和提示信息。
	闹钟 显示时间和提醒功能
辅助设备	电池电量传感器 测量电池剩余电量
	模数转换器 模拟信号转换成数字信号，是信号由麦克向处理器传递的接口 数模转换器 数字信号转换成模拟信号，是处理器向麦克传递信号的接口
	屏幕控制器 在液晶屏幕上产生图象
现实内容 物理设备	日期
	时间
	语音，声音
	麦克，扬声器
	电池

	键盘，显示器
关键概念	语音，采样，声音采样
	声音信息
永久对象	已经记录的信息的集合 数字磁带中的内容
	当前时间，日期
	闹钟定时 闹铃响起来的时间
事务处理	消息
可视元件	菜单，菜单设置选项，菜单选择
	时钟，闹铃开启和关闭指示
	日历
	状态条，帮助行，任务进度指示条
	播放，录音，停止，是，否，上，下，左，右等9个按钮 使用者将通过上述按钮操纵系统。



图 3.1 用户接口/User Interface

图 3.1 显示了用户接口的所有可视元件。系统操作采用菜单驱动，屏幕上将显示当前菜单。顶部显

示当前时间，日期及闹钟定时。底部显示提示信息。

3.2 建造类图/Building the class diagrams

数字录音机系统类图/The Sound Recorder Class Diagram

使用者通过Display类（显示器）和keyboard（键盘）与系统交互。这些都是被动对象，因此我们决定提供UserInterface（用户接口）类来管理上述交互行为。UserInterface类依赖AudioContoller（音频控制器）类来完成任务。AudioContoller类是整个系统的核心类，它将执行用户接口类要求执行的任务。

AudioContoller类通过Audioinput（音频输入）类和AudioOutput（音频输出）类来完成对声音的处理。这两个类分别与硬件封装类Microphone（麦克）和Speaker（扬声器）相关联。[AKZ96]信息存储将由MessageMemory类完成，该类维护一个信息存储目录，负责为新信息提供分配控件以及删除旧信息。

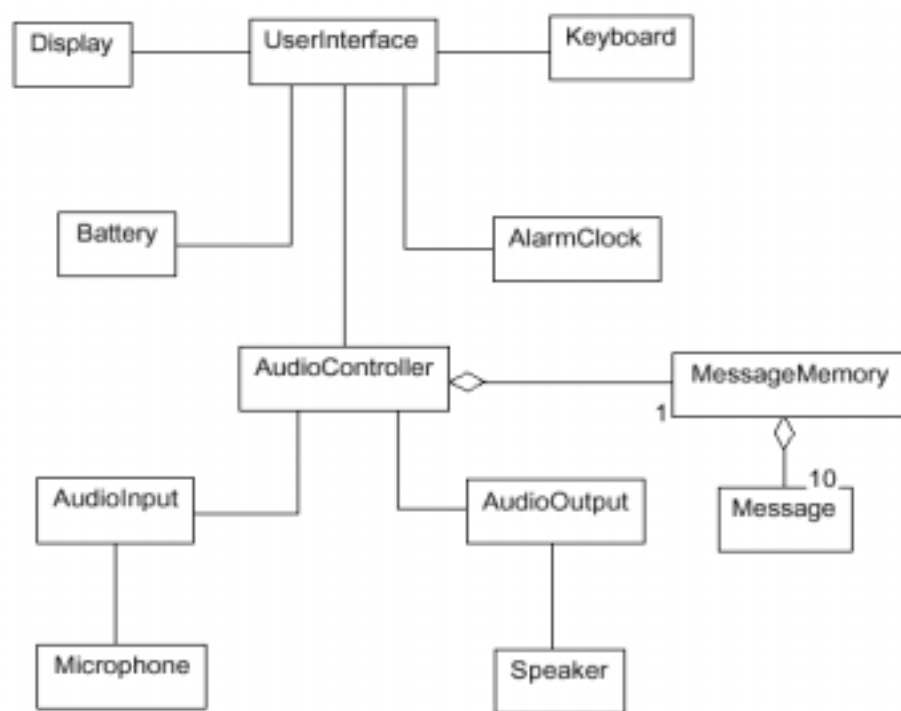


图 3.2 数字录音机系统类

AlarmClock类更新内部时钟并检查是否到了应该响铃的时间，如果是就发出铃声。在这种情况下，它会将闹铃事件通报给UserInterface类，UserInterface类收到消息后会在屏幕上显示闹铃指示并通过AudioControl类发出闹铃声。

图 3.2 以图形的方式描述了数字录音机系统主要的类。类图通过对每个类细节抽象，为我们提供了系统的概要描述。

我们决定将类图中的类归总为5个子系统。分别为alarm clock子系统, battery子系统, user interface子系统, memory子系统 和audio子系统. 图 3.3描述了这个划分. 后续章节将对每个子系统深入研究并逐步扩展。

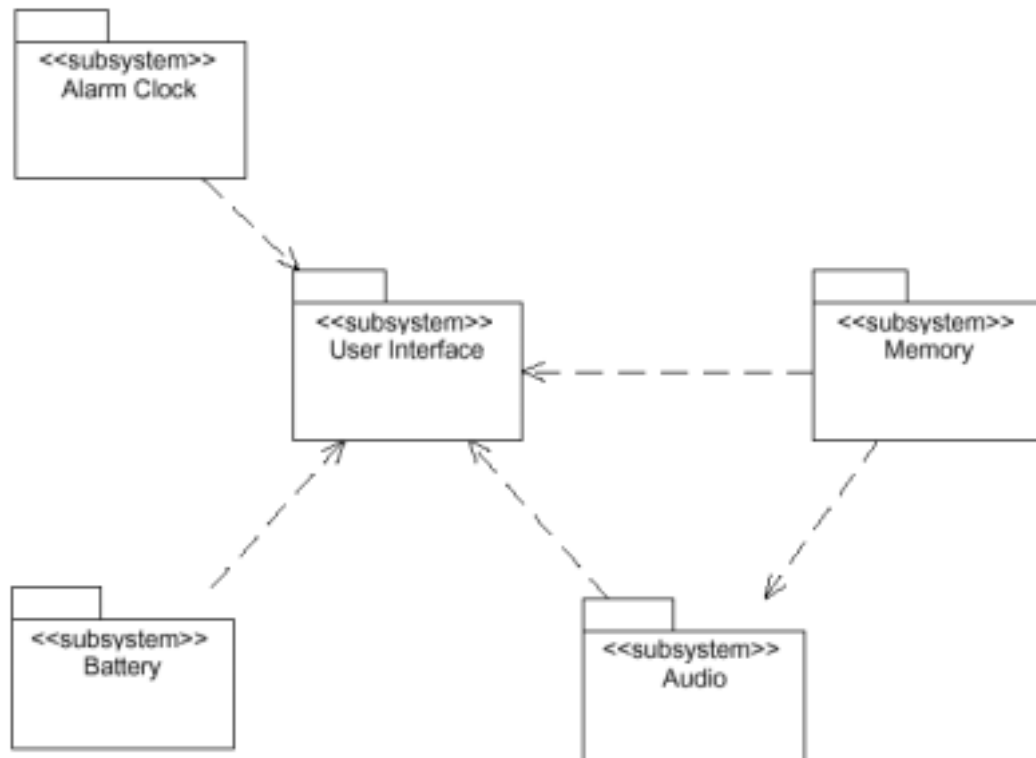


图 3.3 数字录音机的子系统

音频子系统类图

每条信息由一组音频块（audio blocks）组成，而每个音频块又包含了一组音频样本。音频子系统总是记录/回放一个完整的音频块。音频输入和输出类是实时工作的，Timer类是硬件定时器的封装类，它为AudioInput和AudioOutput类提供精确定时。Microphone是麦克的封装类。一个Microphone类记录一个声音样本。

Speaker类能够通过扬声器回放声音样本。

去往讨论组→
(已超过 11,000 人)

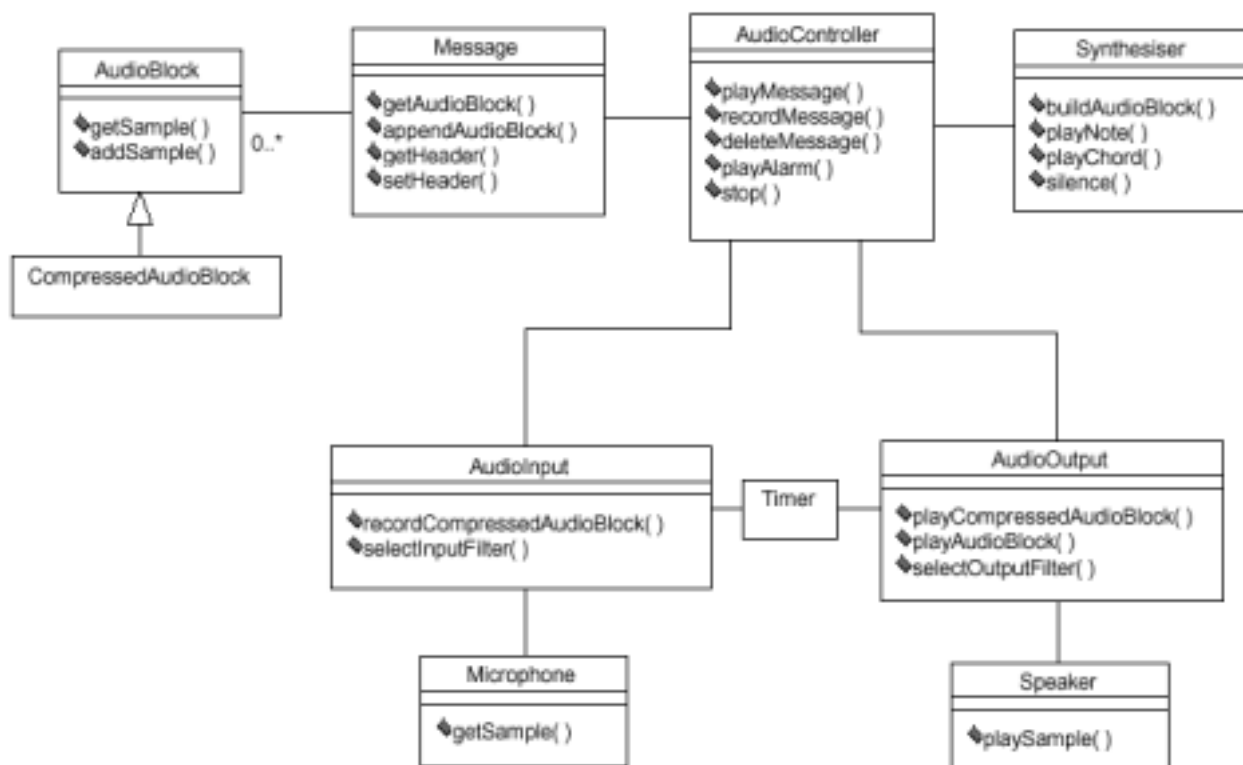


图 3.4 音频子系统类图

为什么需要3个不同的类来播放信息呢？因为回放和记录一条信息是一个复杂的过程，需要精确的定时以及同硬件交互。而且，我们认为在一个处理过程中逐步处理信息的方案会增加设计的复杂度。另外，这样的设计能够很容易扩展成可以处理包含2声道音频块或使用管道压缩（pipelined compression）技术的立体声信息。

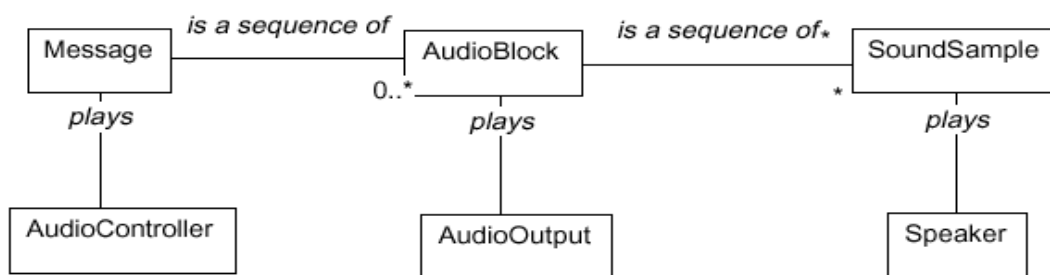


图 3.5 声音信息的内部表示

图 3.5 显示信息（Message）、音频块(AudioBlock)和声音样本(SoundSample)之间的分解关系，以及播放(play)各元素所需要的类。

粒度/Granularity	播放/Play	录制/Record
音频样本, 1/6000秒	Speaker.playSample()	Microphone.recordSample()
音频块, 2000个音频样本。1/3秒	AudioOutput.playAudioBlock	AudioInput.RecordAudioBlock
声音信息 (不定长) 举例: 60个音频块, 20秒	AudioControler.playMessage	AudioControler.recordMessage

图 3.6 显示进行声音回放操作时的信息变换。该顺序图针对可读性进行了优化。实际情况是, 一个 `AudioOutput` 对象在处理一个音频块时会调用2000次`playSample` 方法。

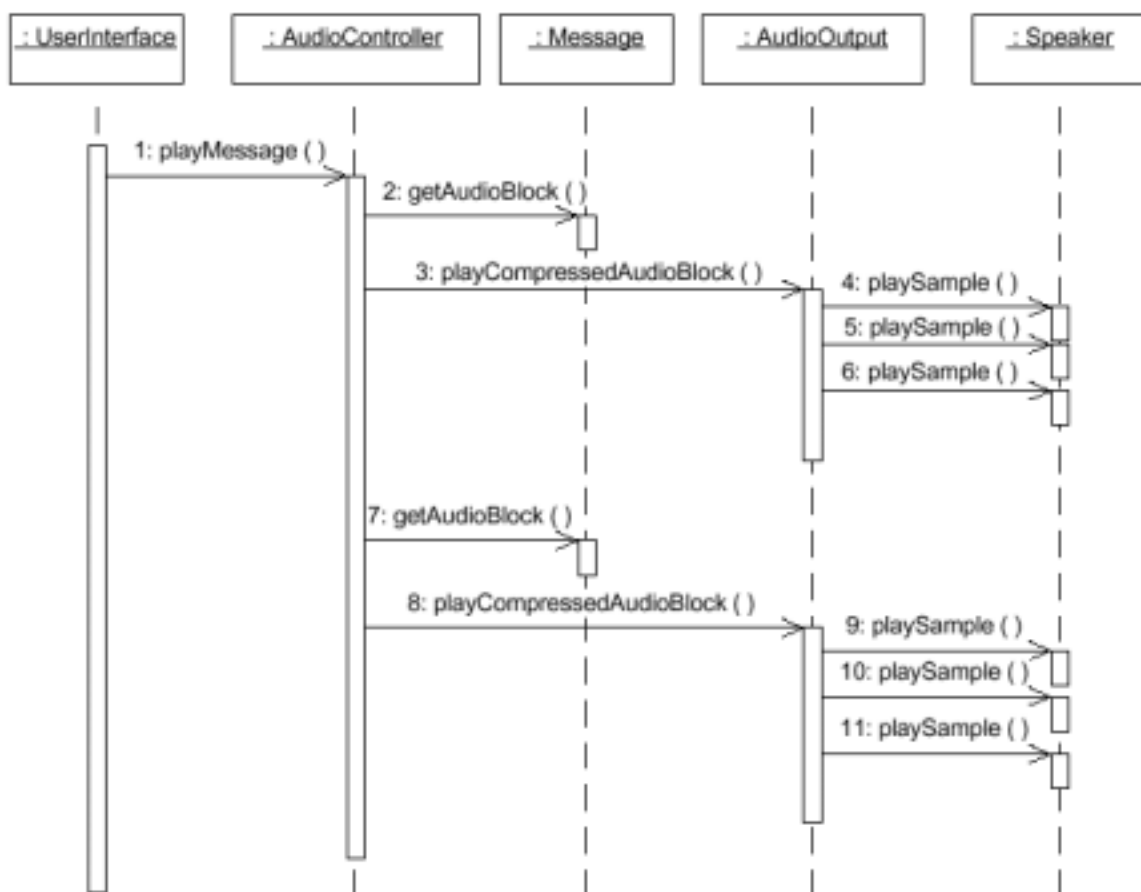


图 3.6 回放信息的顺序图

信息内存子系统类图/The Message Memory Subsystem Class Diagram

MessageMemory 类负责管理系统的存储空间，它维护了一个目录来保存已有的信息同时为新信息分配存储空间。图 3.7 为内存子系统的类图。

用户接口通过MessageMemory 类取得已有信息清单，但不能对之进行修改。如果想修改MessageMemory管理的内容，只能通过AudioController 类的进行。如果用户接口需要删除一条信息，可以通过MessageMemory类的deleteMessage过程。该过程将调用AudioController类中的相关方法对信息进行删除。这个技术避免了用户删除一条正在被Audio Controller类回放或记录地信息。图 3.8 描述了这种情况。

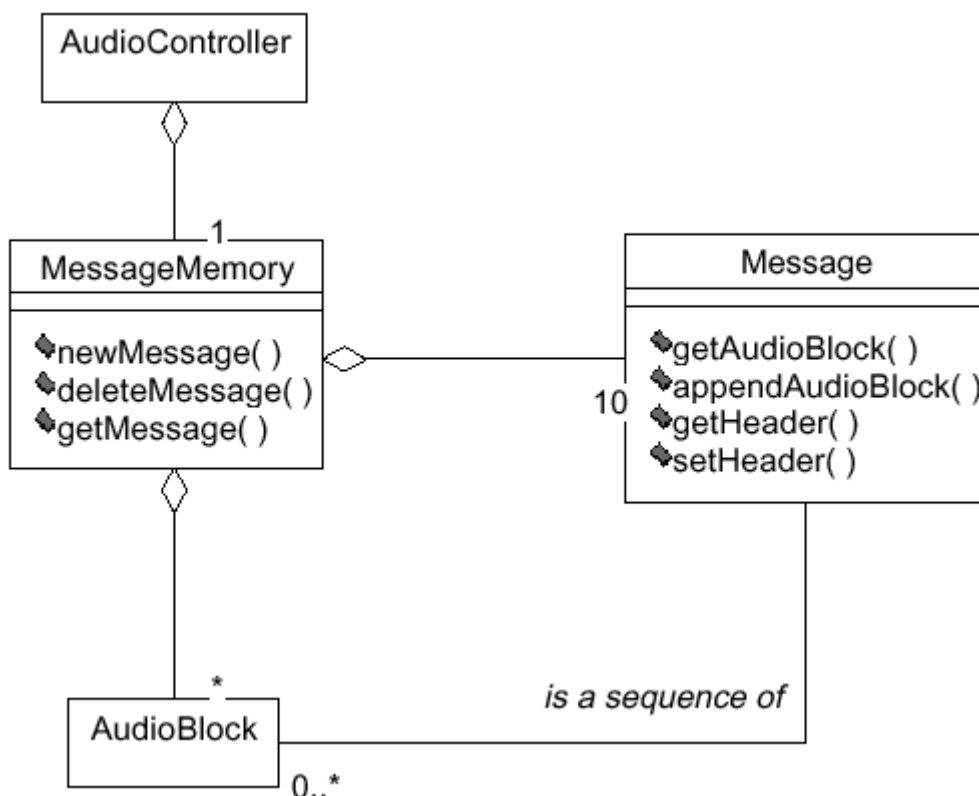


图 3.7 Message memory 类图

去往讨论组→
(已超过 11,000 人)

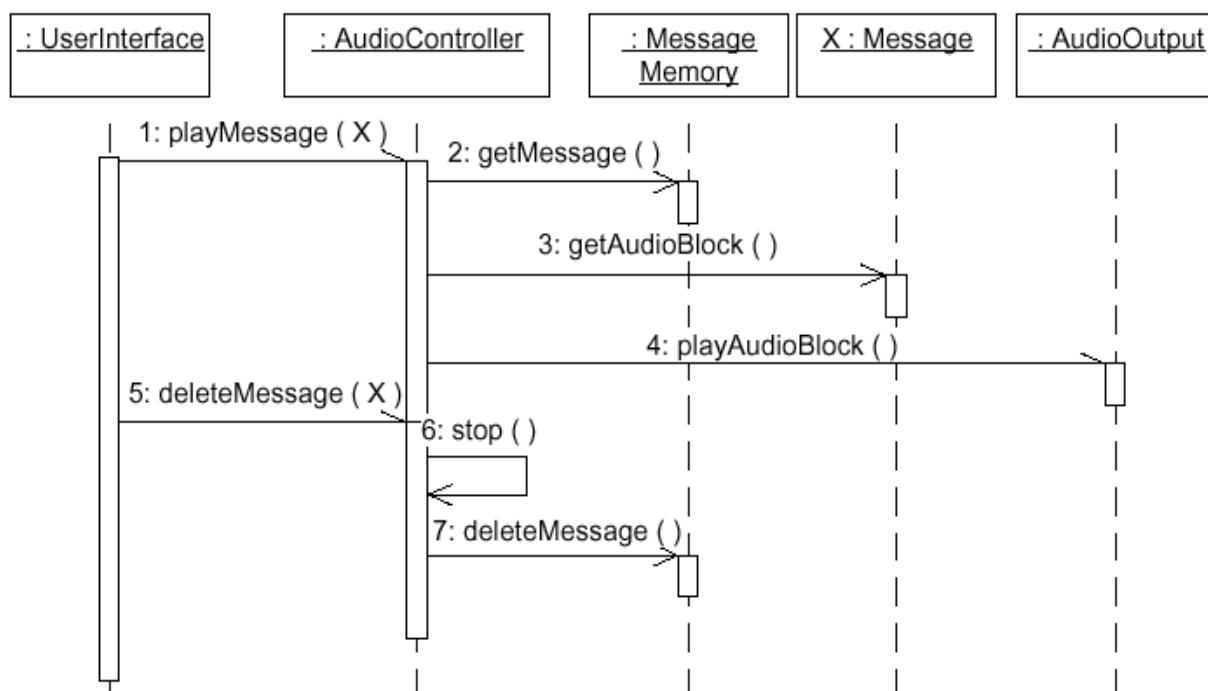


图 3.8 删除一条正在播放信息的情况

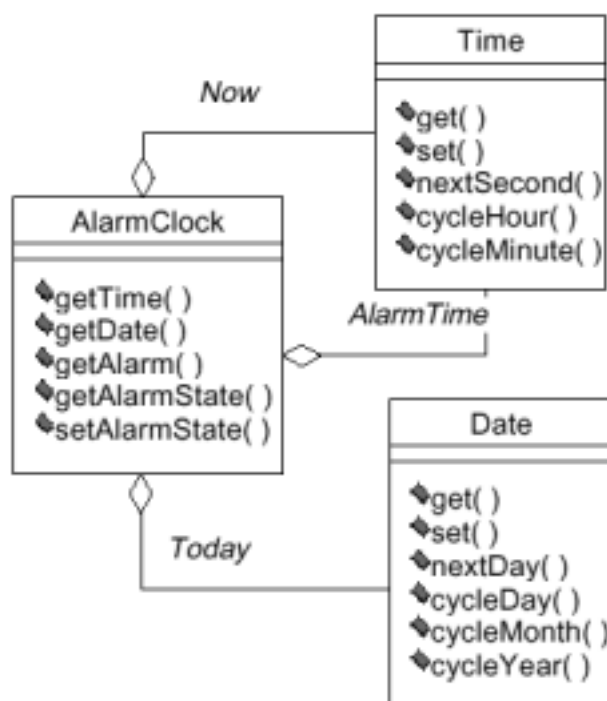


图 3.9 AlarmClock类图

AlarmClock类图

AlarmClock 类保存当前日期、时间及闹钟定时。

AlarmClock 通过一个定时器来测量时间，每秒钟都调用nextSecond 方法更新内部时钟。当时间增加

到新的一天时，调用nextDay方法更新日期。AlarmClock类在图 3.9.中描述。

UserInterface类图

UserInterface 类管理系统与使用者之间的交互行为。它接收使用者的键盘输入，然后在屏幕上作出反应。Display 类是硬件的显示接口，为了节约能量在不用的时候可以关闭。抽象的绘图区用来在屏幕上画图，它提供一些基本的作图功能如画点、线、文本或者用某个颜色填充一个矩形区域。一个绘图区就是屏幕上的一个矩形区域。它负责本地坐标到全局坐标的几何转换。View 类利用绘图区提供的简单的作图功能显示系统对象，如当前时间或者菜单等。每个View对象都拥有自己的绘图区，图3.10 对这些对象进行了可视化描述。

UserInterface 还接收从AlarmClock类和Battery类传来的消息。某些事件，如电池电量不足等称为非模态事件，这类事件不管什么时候发生，系统总会作出响应。

非模态事件由UserInterface对象进行管理。相对的，另外一些事件，比如在当前用户模式下用户按了一个按键。如果用户按的是'Down'键，系统所做的操作可能是选中下一个菜单项或者减少当前事件。这类事件被称为模态事件。它们是由UserInterface的常规事件处理器传递给具体的User Mode类的进行处理的。

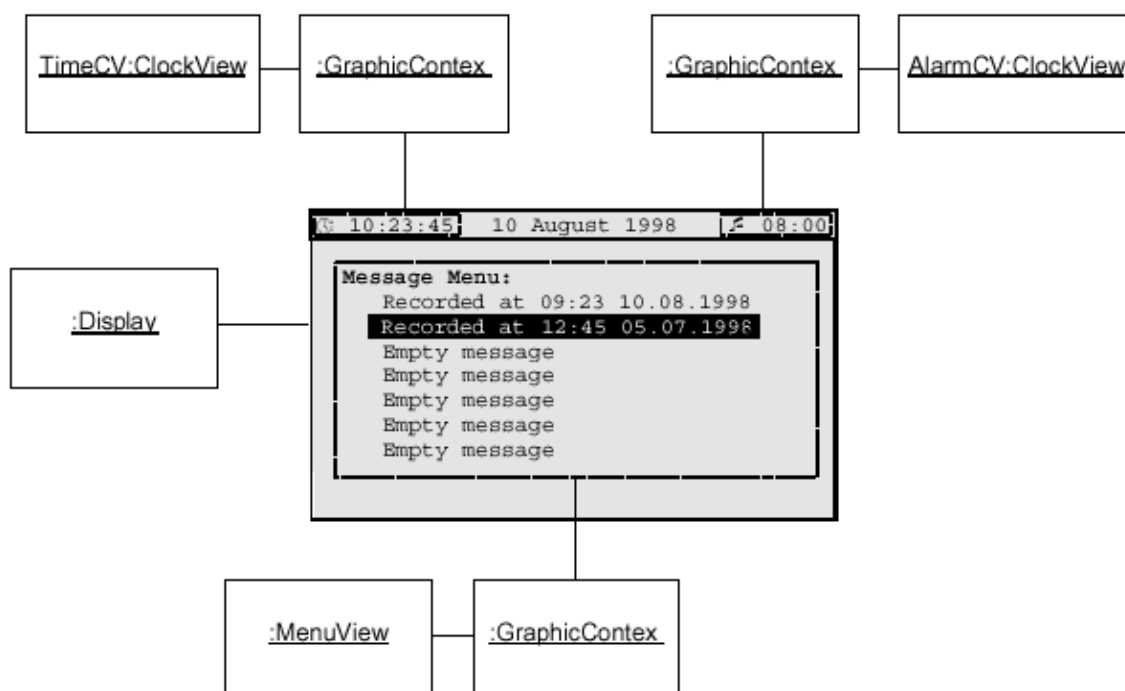


图 3.10 图形对象的可视化描述

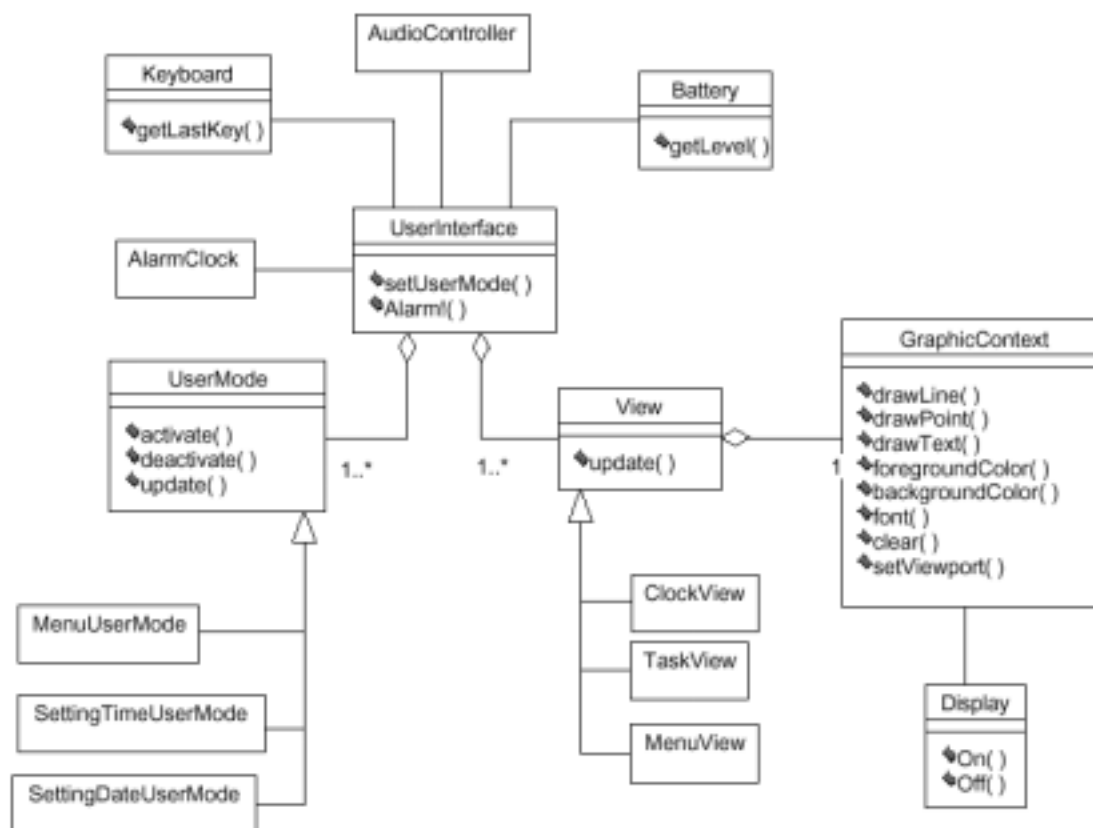


图 3.11 用户接口子系统类图

MenuUserMode 类和 MenuView 类允许用户通过屏幕菜单选择不同的选项。

SettingDateUserMode 和 SettingTimeUserModes 负责改变当前日期和时间，并通过

DateView 和 TimeView 对象在屏幕上显示日期和时间。

4 分析：定义对象的行为

类图是我们系统的静态视图。为了更好的理解系统的行为，我们将建立新的图表来表现设计的动态内容。状态图(statechart diagrams), 协作图(collaboration diagrams)以及顺序图(sequence diagrams) 用来描述系统的动态行为。我们将利用状态图来描述某单个类的内部变化。

音频控制器/Audio Controlle

一个音频控制器对象负责控制声道。声道是用来录制/回放声音信息或播放闹铃的声音。

去往讨论组→
 (已超过 11,000 人)

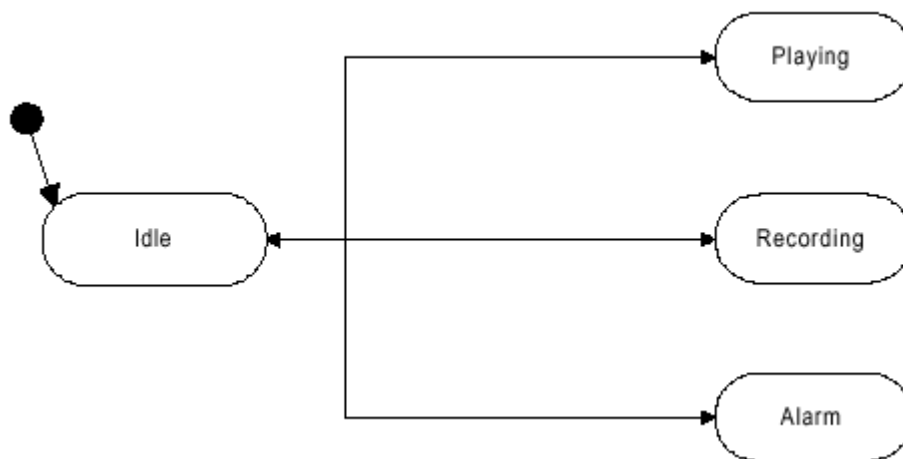


图 4.1 AudioController的行为

音频输入/Audio Input

一个Audio Input 对象控制一个声道. 在DMA通道的帮助下, 通过麦克记录一个音频块并将之压缩。

图 4.1描述了Audio Input 对象的行为。

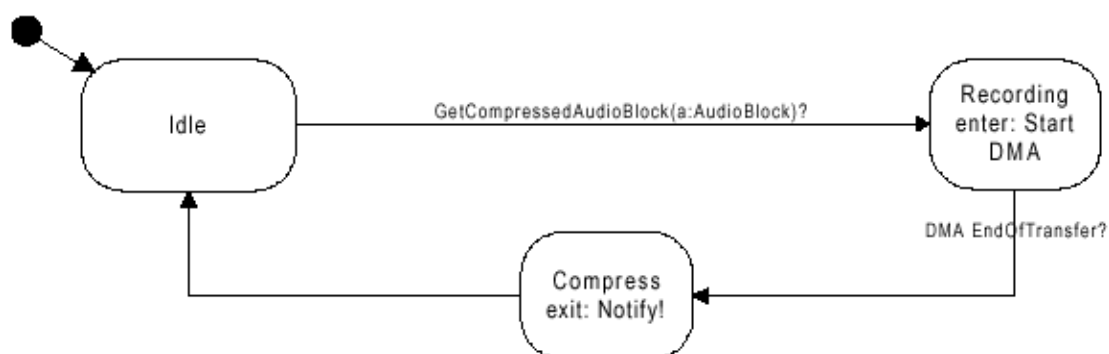


图 4.2 AudioInput 状态图

音频输出/Audio Output

一个AudioOutput 对象控制一个声道, 能够通过扬声器播放一个音频块。

去往讨论组→
(已超过 11,000 人)

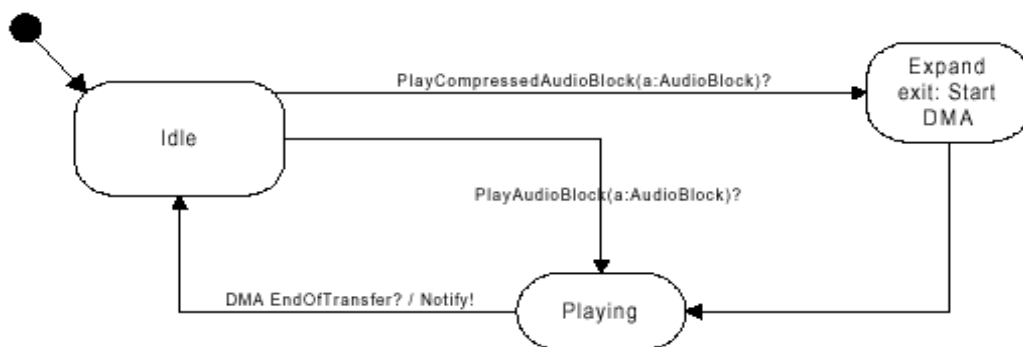


图 4.3 AudioOutput 状态图

麦克/Microphone

Microphone 对象是硬件麦克的包装类,它具有一个可开关的输入放大器,当麦克打开的时候,它能周期性的够捕获声音采样。

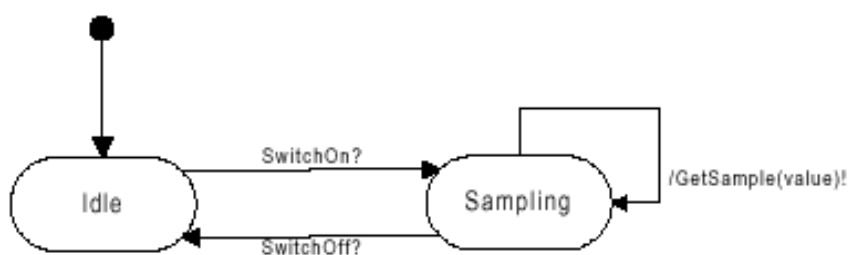


图 4.4 Microphone 状态图

扬声器/Speaker

Speaker对象是真实扬声器的包装类,它具有一个可开关的输出放大器。

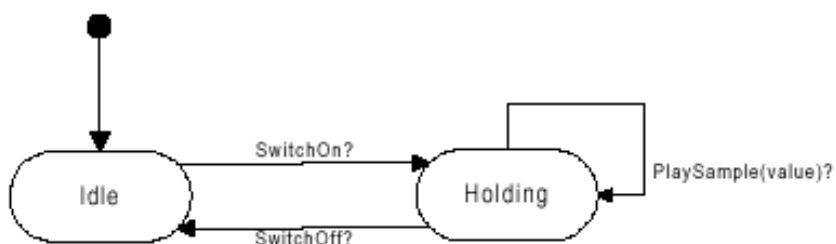


图 4.5 Speaker 状态图

时钟/Timer

一个硬件时钟负责测量时间, Timer 对象是时钟的包装类。

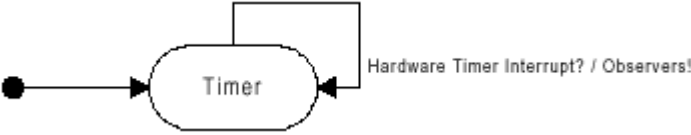


图 4.6 Timer 状态图

4.1 用户接口

菜单用户模式/Menu User Mode

菜单用户模式是用户接口中最主要的用户模式，用户可以通过按钮浏览菜单并选择想要的内容。用“up”和“down”按钮下一个和上级菜单。用“right”和“yes”按钮调用菜单功能。“left”按钮选择上级菜单。

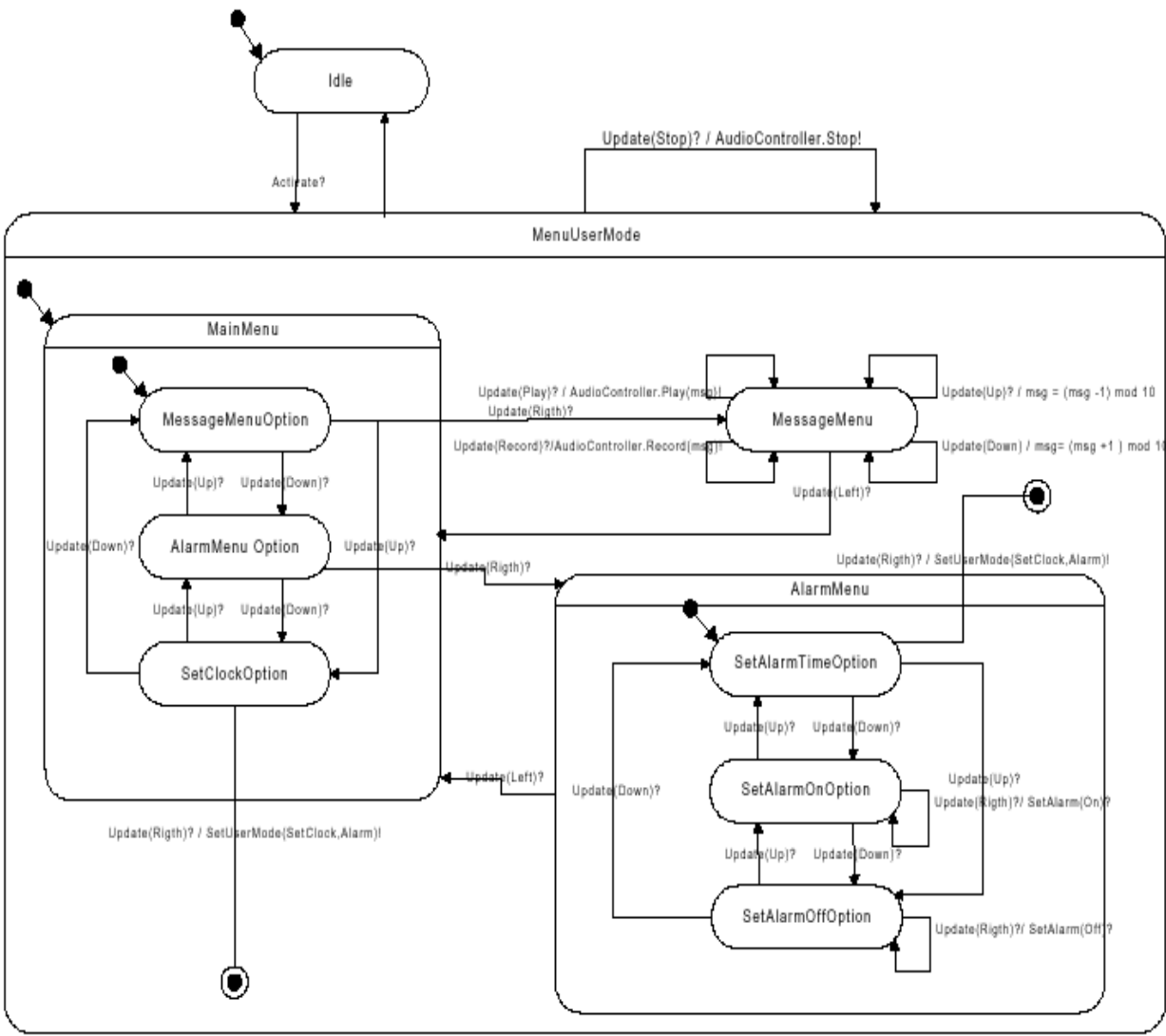


图 4.7 菜单用户模式状态图

时钟设置用户模式/Setting Clock User Mode

用户在时钟设置模式可以设置当前时间和闹钟定时。通过“left”和“right”按钮在分钟和小时位置上进行切换。“up”和“down”按钮用来修改当前位置的内容。

日期设置用户模式/Setting Date User Mode

日期设置用户模式用来修改当前日期，“left”和“right”按钮用来在日、月、年位置之间进行切换，“up”和“down”按钮修改当前位置的内容。

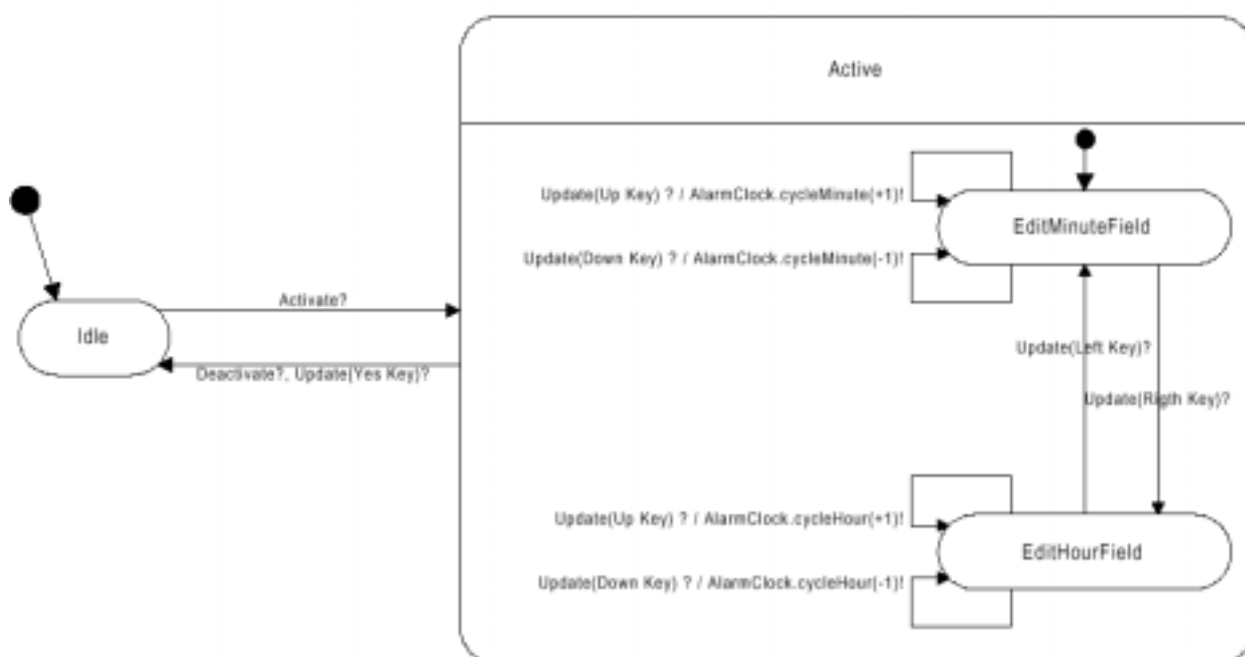


图 4.8 时钟设置用户模式状态图

umlchina 提供 UML/OOAD 培训

通过讲述一个真实 N-Tier 系统的开发过程,使学员自然领会 OO 技术。概念并不按部就班讲授,只是由实例带出。

详情请联系 think@umlchina.com

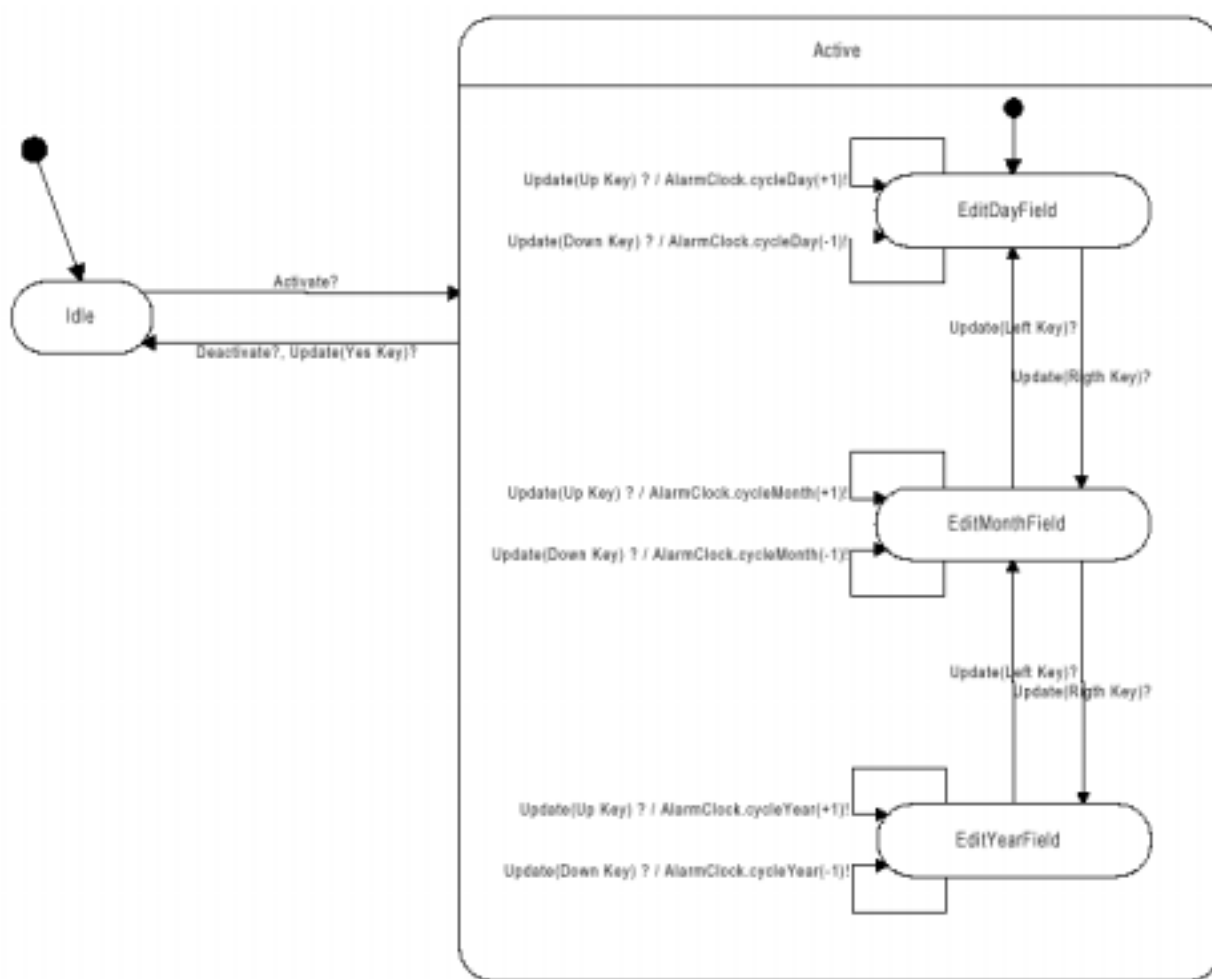


图 4.9 日期设置用户模式状态图

5 构架设计/Architectural Design

在本节中，我们将描述设计中所涉及的所有硬件资源。在我们的最终产品中，硬件和软件是同样重要。一个消费者不会只买软件而不要配套的硬件或只买硬件设备而没有相应的软件，他需要的是一个由硬件和软件共同构成产品。

需要提的一点是，我们没有试图对系统硬件作任何优化工作。举例来说，目前我们采用的处理器比系统实际需要的功能要强，这可能造成系统的成本增加，体积和耗电量变大。但也正是因为这一点，我们可以集中精力进行系统设计，从而避免在系统实现方案的选择和优化上花费更多的时间。

5.1 物理构架/Physical architecture

主处理器采用Hitachi SH7032 16MHz工作主频，带8Kb 内置RAM，内核采用常规的RISC整数管道技术，负责运行机器指令。系统内置外设包括一个模数转换器、一些定时器和计数器、DMA通道和一个监控器。

手册[HSH1]中包含处理器内核指令描述，硬件设备和内置外设的描述在手册[HSH2]中。图 5.1 显示了建议构架。本系统原型配置了64Kb EPROM, 256Kb SRAM 和128Kb 显存。EPROM 包含引导代码和一个调试程序。系统的两个RS-232 兼容串口MAX232用来下载和调试软件，它们与处理器的串行通道连接。

初始化串口的程序也保存在EPROM中，串口的收发速率可达到最大38400 bps。

模数转换器有7个输入通道，采样频率为60KHz，采样精度12位。它的作用是通过麦克录制声音。主板描述在手册[HEVB]中。声音回放装置为外置数模转换器AD7524。液晶显示器 [HLCD]为320x240像素、黑白显示，有背景照明功能。

LCD 控制器采用一个Yamaha YGV610B，支持16级灰度。LCD 控制器产生所有液晶屏幕显示的信号。它具有独立的显示内存。它通过系统总线与处理器交换信息。LCD 控制器的端口和显示内存将映射到系统内存中。

5.2 构架模式

系统对软件没有安全和可靠性方面的要求，也不支持硬件容错。或许会出现在设备报废之前，由于用户使用不慎如进水或掉到地上而造成某些部件损坏的情况。

监控器负责在发生故障的情况下重新启动系统。监控器重新启动系统会造成内存中内容的丢失。因为是封闭式嵌入系统，所以数字录音机不具备与其它系统进行通信的功能。同时，也没必要提供仅供测试和诊断使用的通信功能，因为系统很简单，通过屏幕所显示的提示进行测试即可。事实上，本系统在设计上就是不能测试和维修的。处理器通过内存地址空间访问外设，它们被紧密的结合在一起，根本就不需要任何特殊的通信机制。

5.3 并行性设计

并行模式必须明确软件中所使用的线程以及线程之间的通信机制。为了使设计尽可能简单，我们决定在当前的设计中不采用实时操作系统而在将来的设计中采用。在当前设计中，有两个可执行线程。用户线程是一个交互线程，用来管理使用者与系统之间通过按键和屏幕进行的交互行为。音频系统线负责调度响应对象和实时对象。

音频系统线程在处理器收到中断请求的时候被激活。它享有比用户线程更高的优先级。在音频系统线程中有一个调度对象来负责调度其它对象的运行。硬件定时器周期性激活该调度对象来处理任务表。任务表中每个任务元素都包含一个方法的指针及该任务的执行周期信息。调度对象具有最高优先级，目的是为了尽可能快的执行任务并得到返回结果。不同线程中的两个对象通过Reactive Subject模式（详见附录A）

进行通信。

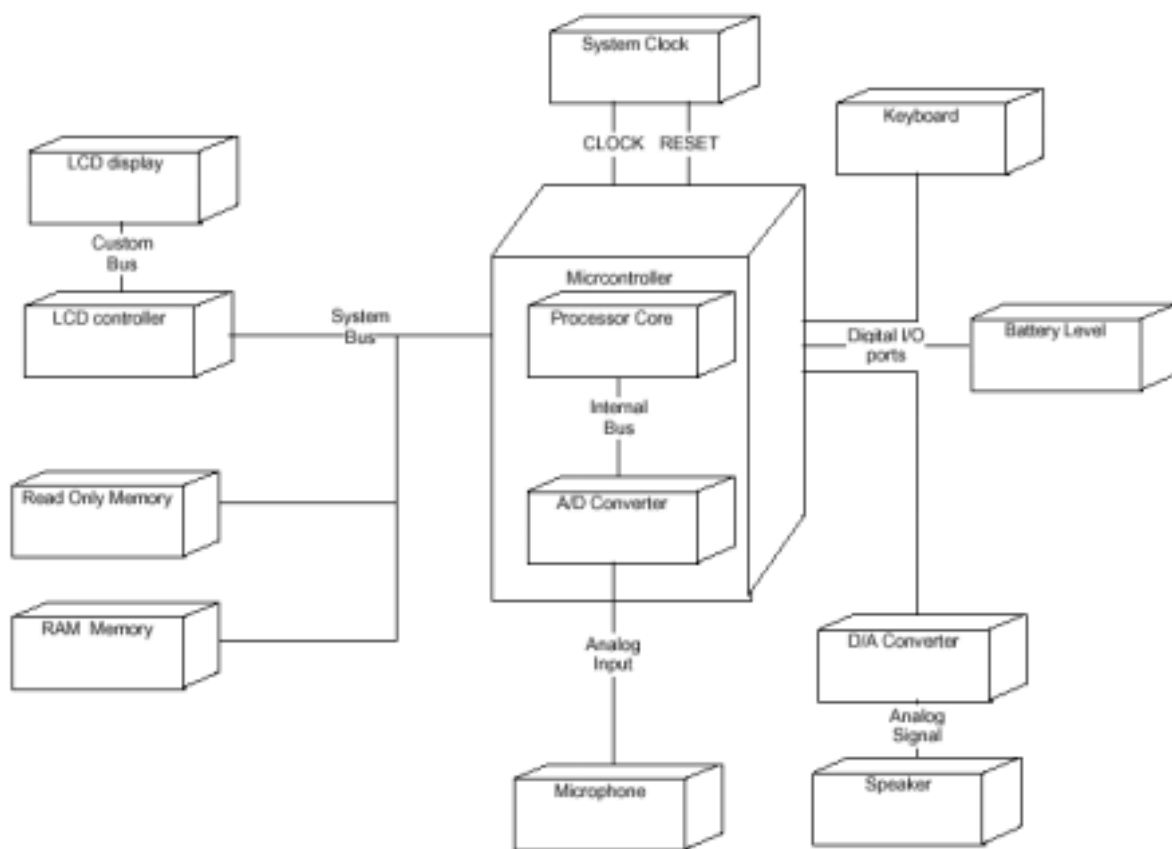


图 5.1 数字录音机系统硬件结构

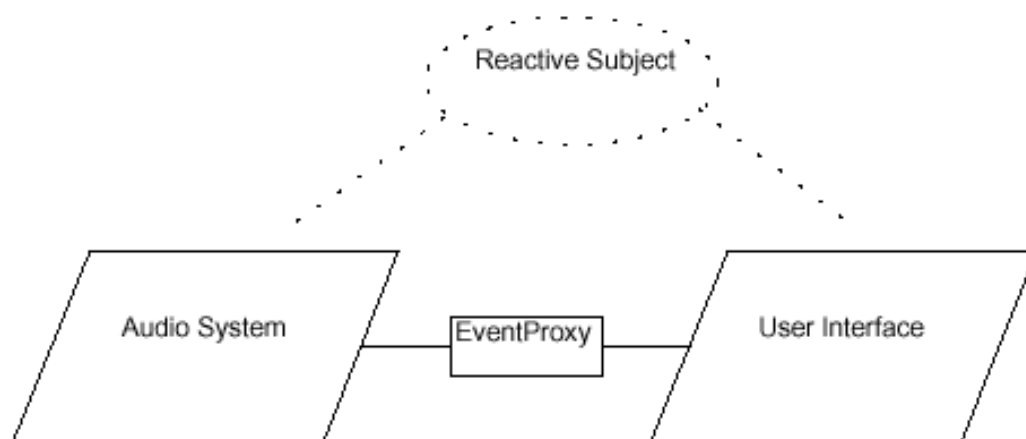


图 5.2 任务图

6 机制设计/Mechanistic Design

在本章中，我们将讨论不同的软件对象是怎样通过协作而完成它们各自的功能。第四章介绍了每个独

立对象的内部行为。然后我们又描述了在收到消息的时候对象的状态变换。现在，我们将关注一下对象之间是怎样交换消息的。我们还将使用‘软件模式’对不同的对象的外部行为进行描述。

6.1 硬件与反应对象之间的协作

我们可以将硬件视为一个参与者。这些参与者可能在分析阶段已经定义了。硬件可以通过中断请求发送给运行程序一个事件。当一个硬件设备需要给程序发送时间的时候便会产生一个中断请求，在某个时间点处理器会接受中断请求，并停止当前正在执行的程序流程，然后调用中断处理过程。中断处理过程处理硬件请求后必须尽快返回，以允许当前程序进程继续进行。不幸的是，中断服务程序(ISR-interrupt service routine)不能是一个对象的方法，因此，没有任何相关的背景信息，也不能使用大多数OO语言支持的‘this’或者‘self’指针功能。

所以，设计者必须建立一个将硬件中断请求转换为对象信息的机制。我们将这种机制封装在ISR抽象类中。ISR类的子类就可以象实现普通方法那样实现中断服务程序。

6.2 反应对象与用户接口之间的协作

键盘、电量表(Battery Level Meter)、闹钟(alarm clock)、音频控制器(audio controller)与用户接口(user interface)之间的协作遵循**条件反应模式**(Reactive Subject pattern)。反应对象只向事件代理发送事件，并不需要等待用户接口读取事件。用户接口定期检查时间代理中是否有新事件。一旦发现有新事件则委托屏幕和控制器产生响应。

6.3 Scheduler 和 Alarm Clock, Keyboard和Battery Level Meter之间的协作

Scheduler 对象为依赖时间信息的类提供精确定时和时序安排。AlarmClock 从scheduler类订阅信息。每秒钟Scheduler都会通报AlarmClock类有一秒钟时间已经结束。

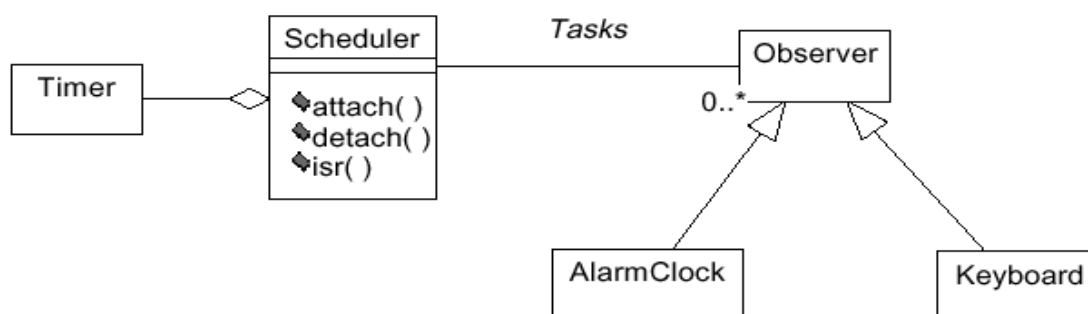


图 6.1 Scheduler类图

Keyboard 对象需要定期取得物理键盘所有按键的状态。我们决定每秒钟检测10次键盘输入。所以，

如果用户按下并放开一个按键的时间小于1/20秒，则该按键输入会丢失。

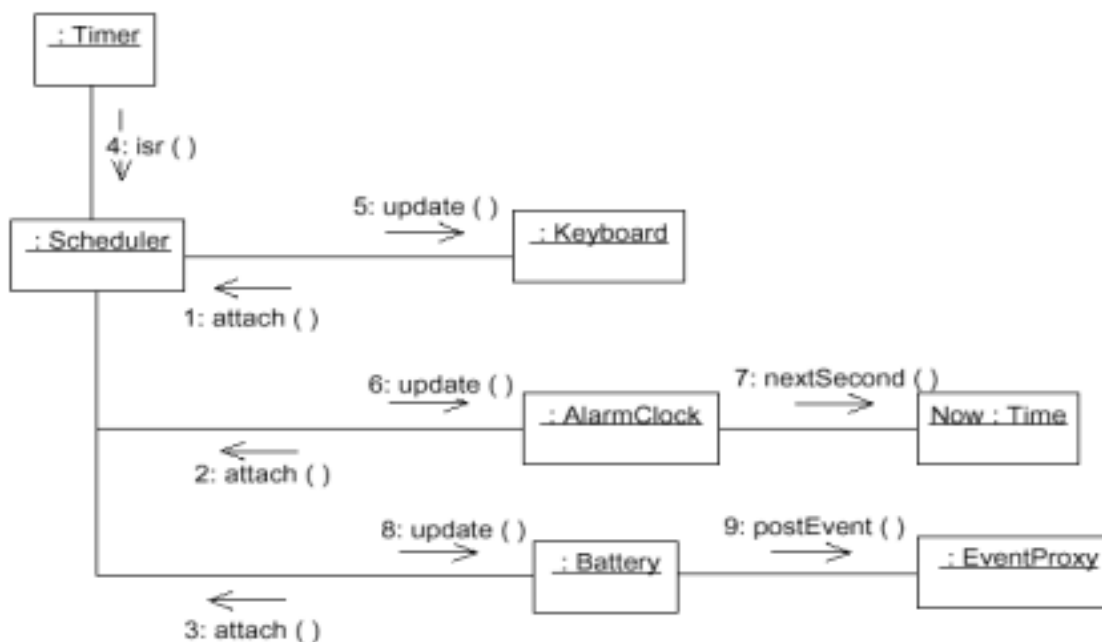


图 6.2 Scheduler对象与其客户对象之间的协作

物理键盘还能够在按键时产生一个硬件中断，与定时检测方法相比，中断法能够减轻CPU的负载，但是需要增加所需的硬件。

BatteryLevelMeter 类每5秒钟测量一次电池的电量。

Scheduler负责周期性的激活Keyboard 和BatteryLevelMeter类。图 6.2 显示了scheduler是如何周期性的唤醒系统的反应类的。

6.4 MessageMemory、Message和AudioController对象之间的协作

MessageMemory 是Message对象的容器。这种协作关系遵循Container模式。AudioController在需要访问Message对象的时候会用到MessageMemory类。

6.5 SettingTimeUserMode, AlarmClock, Keyboard 和ClockView 对象之间的协作

这些对象都遵循Model View Controller模式。AlarmClock为ClockView提供一个模型用来在显示屏上显示时间。UserMode 对象控制系统与用户之间的交互行为。由于AlarmClock 是反应对象，ClockView 是交互对象。它们之间的协作将遵循Reactive Subject 模式。Keyboard 对象向UserInterface类通告用户按键事件也遵循该模式。图 6.3 显示了当前用户模式为SettingTimeUserMode.时，用户按下‘up’键后所产生一系列消息的顺序过程。

6.6 UserInterface, AudioController, Messages 和AudioOutput 对象之间的协作

UserInterface 和AudioController 之间的协作采用了Reactive Subject 模式。

AudioController, Message, AudioInput 和AudioOutput 之间的协作采用Observer模式。图 6.4 显示了播放一条声音信息所的消息传递顺序。为了简化，声音信息用一个音频块表示。这种合作有点复杂但支持回放和录制两个不同的信息。同时，还支持录制和回放立体声声音，但每条信息将有两个音频块流组成。

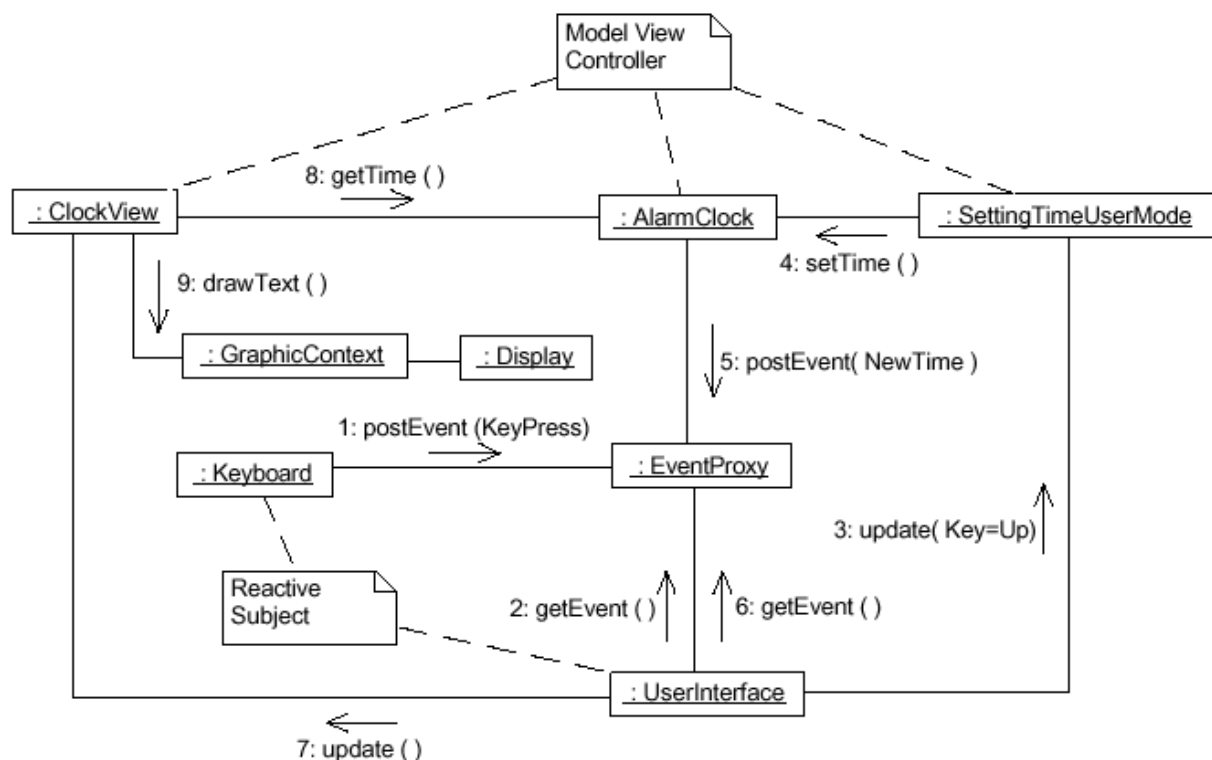


图 6.3 Model-View-Controller 协作

umlchina 提供 UML/OOAD 培训

通过讲述一个真实 N-Tier 系统的开发过程，使学员自然领会 OO 技术。概念并不按部就班讲授，只是由实例带出。

详情请联系 think@umlchina.com

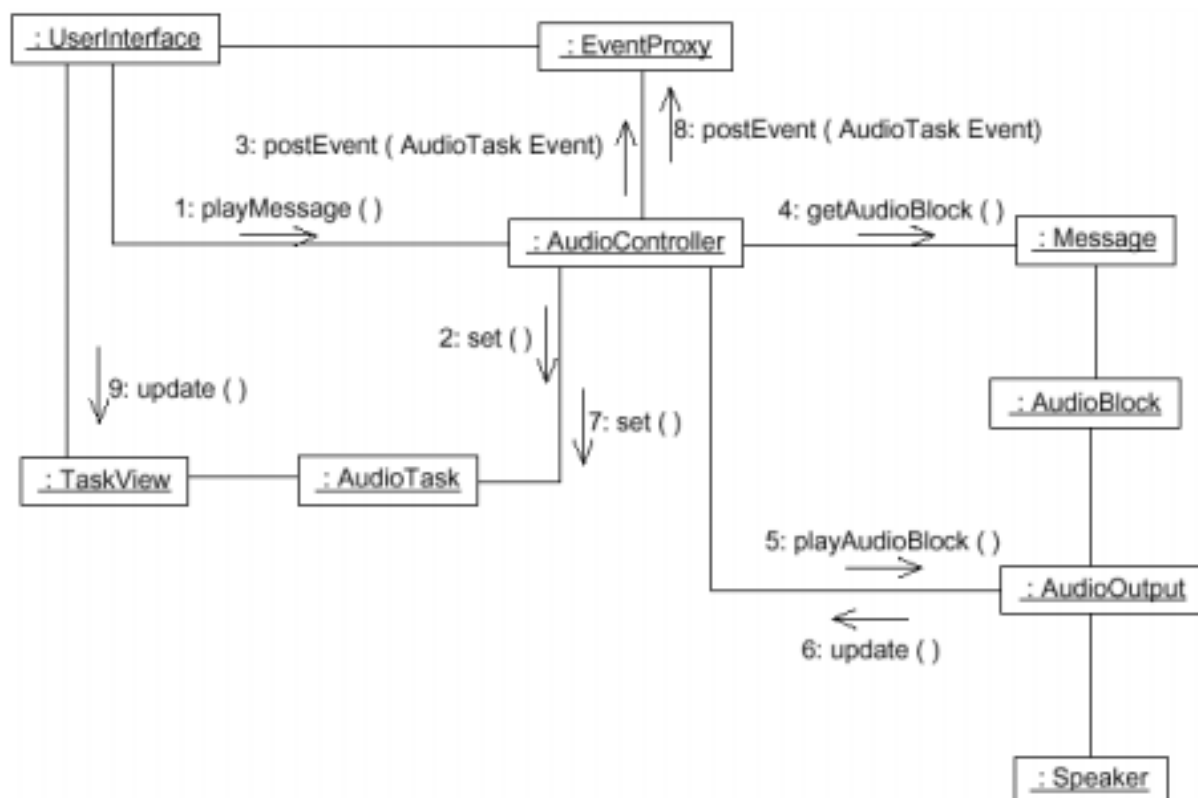


图 6.4 UserInterface 和AudioController对象之间的协作关系

7 详细设计/Detailed Design

7.1 硬件的对象建模

一个硬件封装类是对一个硬件设备的软件对象化描述，是应用程序与硬件之间的接口。硬件封装类的构造函数将初始化硬件设备，然后该类所对应的硬件就可以使用了。可以通过封装类的方法对硬件进行配置、启动或停止等操作。由于硬件的状态将由封装类的状态表示，通常情况下封装类都不会有太多的属性。

详细设计和实现硬件封装类需要很准确的硬件方面的知识。本系统中的硬件封装类包括speaker, microphone, timer和keyboard 对象。

7.2 音频压缩算法

一个很明确的需求就是，必须以6Khz频率,8位采样样本录制声音。这表示在录制过程中，每秒钟需要至少6KB内存。我们可以通过对输入信号的压缩处理来减少内存的需要量。最简单的音频压缩是适应性Delta脉冲编码调制技术（Adaptive Delta Pulse Code Modulation — ADPCM）[ITU727]. ADPCM 很容易实现并且CPU占用率很低。

7.3 使用直接内存访问通道

在录制声音信息的时候，一个AudioInput 对象每秒钟需要从麦克获得6000个样本。有多个途径可以完成这个任务。

第一个方案是使用scheduler来计时，由一个定时器在每秒钟内激活Scheduler6000次。scheduler 被激活后便通过AudioInput对象采集一个声音样本。很显然，由于只需要传输一个字节，CPU周期的大部分事件被浪费了。

另一个解决方案是通过一个新的定时器直接激活AudioInput 对象采集声音样本。这种情况下，CUP 占用率会低一些但仍是相当可观的。中断服务程序在大多数RISC处理器系统中的表现都大打折扣，也不宜采用。

所以第三个可选方案是使用直接内存访问通道（direct memory access channel—DMAC）。一个DMAC 一次可以将几个word由内存的一处移到另一处而不需要占用CPU资源。由于所有内置外设的端口都映射到了内存地址空间中，所以通过DMAC实现外设之间和外设与内存之间的数据传送是可能的。在我们的设计中，我们将使用DMAC由麦克向一个音频块传输声音样本，或者将音频块传送给扬声器。一个DMAC 单元拥有一些寄存器来控制它的行为，其中主要有源地址寄存器、目的地址寄存器、传输计数寄存器和一些标志寄存器。标志寄存器用来保存是否在每次传输结束后需要增加源/目的地址的空间，或者用来标识被传输的单元是一个word还是一个byte。寄存器的使用使数据能够以更快的速度传输并与定时器保持同步。在编程时，我们将DMAC的A/D转换器的地址作为录制声音操作的源地址。目的地址指向一个能够在每次传输后根据传输内容自动调整空间的内部缓冲区。传输单元是一个字节与声音样本的大小相同。我们使用一个定时器，通过编程实现每秒产生6000次中断以及对每次传输初始化。当定时器向CPU发送中断请求后，DMAC 单元开始传输数据。如果CPU的中断响应被屏蔽的话，DMAC也会传输数据，但不会被接收。因此而造成数据丢失。实际上，我们需要中断被屏蔽，所以定时器不干扰CPU。在声音样本被存储到缓冲区后，DMAC单元产生并发送中断请求给AudioInput。AudioInput对象接收到请求后，将缓冲区的内容压缩成一个音频块。图7.1 显示了录制一个音频块的过程。一条声音信息的所有音频块都是重复此过程而获得的。

去往讨论组→
(已超过 11,000 人)

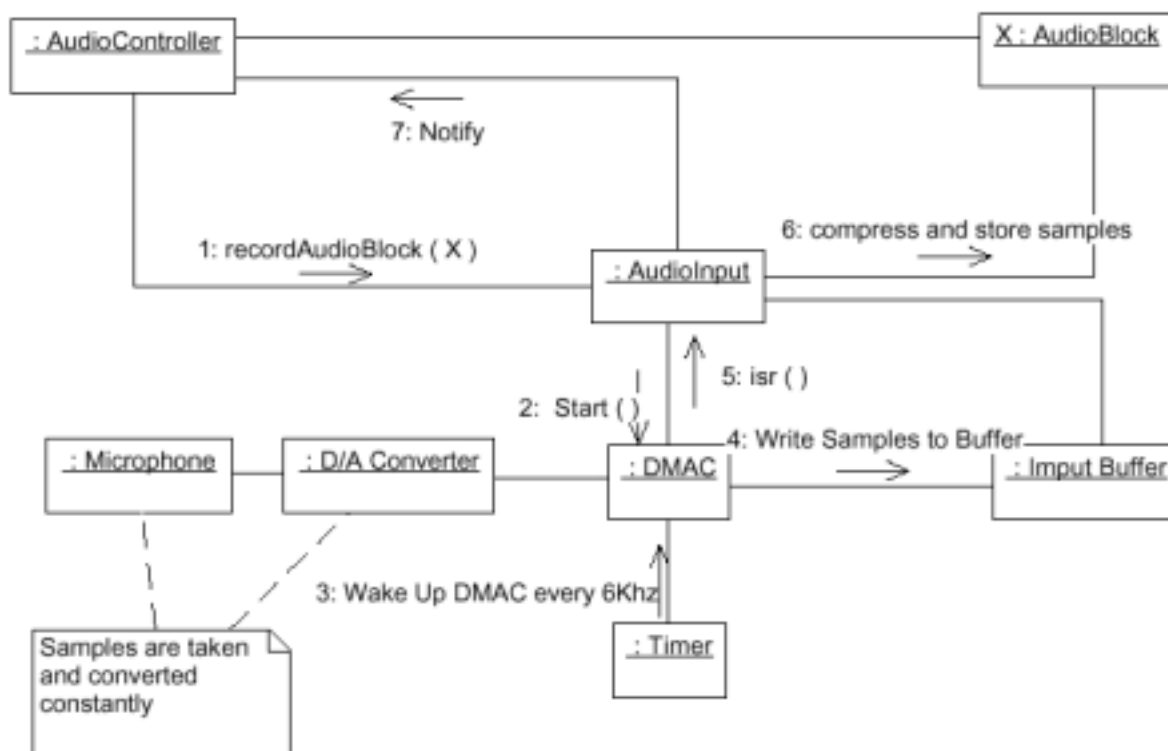


图 7.1 记录一个音频块

设计中还需要讨论的一点是音频块的尺寸。如果太小，CPU将花费大量时间用于对象间的信息传递以及控制DMAC操作同步上。所有声音信息都是由 n (n 为整数)个音频块组成。一条声音信息回放时间长短以及尺寸大小取决于音频块的长度和尺寸大小。如果音频块的尺寸过大则造成内存浪费和系统响应缓慢。

声音回放与声音录制的机制很类似的。先由AudioOutput对象将音频块解压缩为声音样本并存储到缓冲区。然后由DMAC将声音样本传送给D/A转换器。此机制只有在CPU能够很快速的压缩和解压声音样本的情况下才能被接受。否则，用户会在两个回放音频块的间隙听到低频噪音。但这个问题可以通过以下途径解决：优化传输音频块的过程或者考虑以音频块流的方式传输音频块。由于CPU和DMAC是同步工作的，我们可以通过一个管道（pipeline）处理音频流，使DMAC在传输一个音频块的时候，CPU解压下一个音频块。

去往讨论组→
 (已超过 11,000 人)

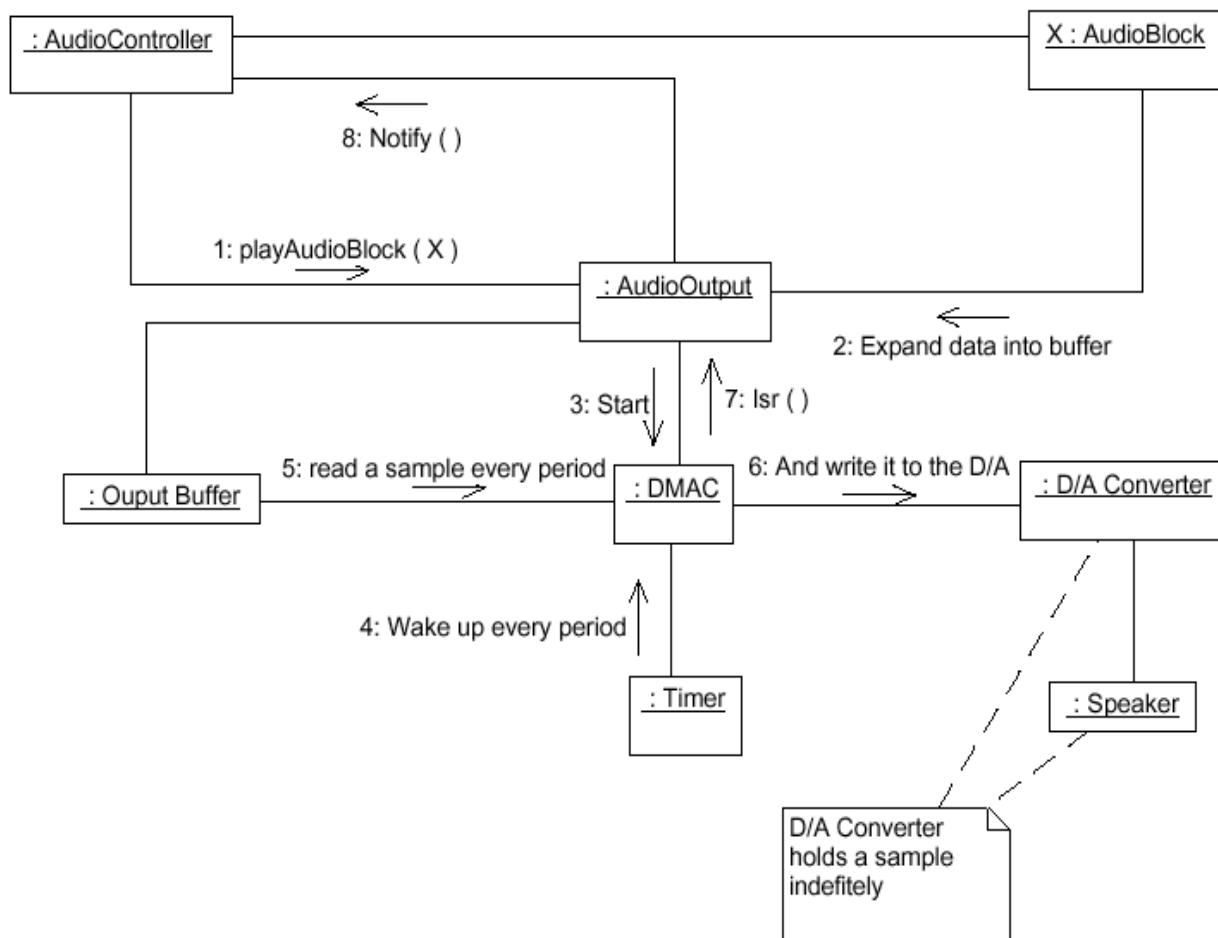


图 7.2 回放一个音频块

7.4 硬件资源的分配

我们必须将硬件和外设资源分配给不同的反应对象。由于中断向量、DMA通道、定时器和输入/输出端口的数量限制。我们不得不仔细计划如何使用它们。值得注意的是，很难（也许是根本不可能）并行使用大部分上述硬件资源。

对象	资源
显示器 (Display)	硬件定时器0
调度 (Scheduler)	硬件定时器1 IMA1 中断
麦克 (Microphone)	模拟信号输入7 A/D 转换器 (自动转换模式)
扬声器 (Speaker)	Digital outputs: PB7 to PB0 硬件定时器2
DMA 通道 DMA Channels	屏蔽IMIA2, IMA3 中断
音频输入AudioInput	DMA 通道1 DMAC1 DEI 中断向量
音频输出 AudioOutput	DMA 通道 2 DMAC2 DEI 中断向量
键盘	数字输出端口: PB4 to PB12 数字输入端口: PC3 to PC0

7.5 内存分配

MessageMemory对象的作用是分配和释放存储声音信息的内存空间。由于有效内存很有限而且系统没有虚拟内存，所以我们不得不设计一种能够保证内存的使用最佳化的机制。这包括通过内存碎片整理和对象排列技术来避免内存空间的浪费。当一个声音信息对象被创建时，它将分配一个数组用来存储音频块，

数组中的每个元素包含一个使用标记和一个指向信息流中下一个音频块的指针。我们通过内存块重用以及预分配技术避免出现内存碎片，不采用在每次录制声音信息的时候分配而在删除信息的时候释放内存的做法。通常情况下，内存分配函数只能由存储队列中分配尺寸确定的内存空间，音频块的尺寸是存储队列组成因子尺寸的整数倍。这样就可以保证不浪费一个字节。

8 实现

考虑到嵌入式系统的最终软件产品不是软件映射到内存中，而是常驻在永久性内存中。因此，程序的所有连接库必须静态调用，必须为程序的所有符号分配绝对地址。程序中必须包含硬件初始化、硬件检查以及重新排列RAM中可执行程序代码。程序能够初始化处理器堆栈，将只读存储器中的变量拷贝到随机存储器中，将未初始化的变量置为0。在目标原型产品中包含带有一个小型调试器的永久性内存。系统打开后调试器被激活，它等待从主机通过通信连接如串口线或网络连接下载主程序。主程序被下载后可以运行或对其进行调试。当软件开发完成后并通过测试后，可以保存在永久性内存中如ROM或EPROM。这些内存可以被插到目标机器上，只要机器启动程序就能运行。

8.1 开发语言和软件工具

我们采用C++编程语言实现系统软件。选择何种开发语言编写嵌入式系统取决于可供使用的工具的数量。因此，我们采用了GNU系列工具，包括C++交叉编译器GNU g++，跨平台调试工具GDB以及GDB的前端图象界面DDD。

8.2 生成软件

目前的程序包括3500行C++代码和2000原有C代码。程序在宿主计算机上被编译，可被下载到产品原型的ROM或EPROM中。目标代码占用40kb ROM和10kb的RAM空间。

8.3 制造硬件

我们制造了一个数字录音机的原型。在网站[SR] 中包含了该原型的硬件构成的信息。硬件中包括了一些模拟信号设备，如信号放大器、麦克的噪声过滤器等。所有描述信息都是在本文讨论范围基础上的扩充。他们为了改进声音质量做了很多努力，用户会因此而更喜欢我们的系统的。

9 结论：使用 UML 进行设计

本文对数字录音机的主要元素进行了定义、列举、分析、设计和实现。

我们使用UML作为设计的图形符号化工具。在下个版本的设计中，我们就将使用更多的UML图表，

减少文字描述。为了达到这个目的，我们需要更多的面向实时嵌入式系统的软件模式。我们所加的文字信息将只是用来阐明，如何修改某个特定的软件模式以实现设计。我们将模型以一些独立的图表表示，以便于管理，同时又可以从不同的角度展示系统。比如UML类图显示的是系统的静态视图，顺序图则描述了系统的动态行为。但是，在UML语法定义[BJR2]中，并不提供对上述两种图表一致性和有效性的判断功能。

UML帮助设计者表达他的设计思想，同时也帮助程序员理解它们。但是，UML只是设计的描述方法，不能使你的设计更完美。而且UML在图表和实现语义方面仍然存一些缺陷。也没有一个系统的方法能够完成由UML图表到最终实现的转换，而且也不能检查最终实现结果是否完全符合UML设计。图 9.1 显示了一个面向对象的嵌入式系统设计的所有内容。UML的图形符号不只局限于描述软件系统，同样也用来描述如何构造它们。本文中所描述的系统就是图 9.1所示的UML类图的一个实例。

感谢

感谢 Marat Vagapov，是他为我们提供了数字录音机的所有硬件设备。

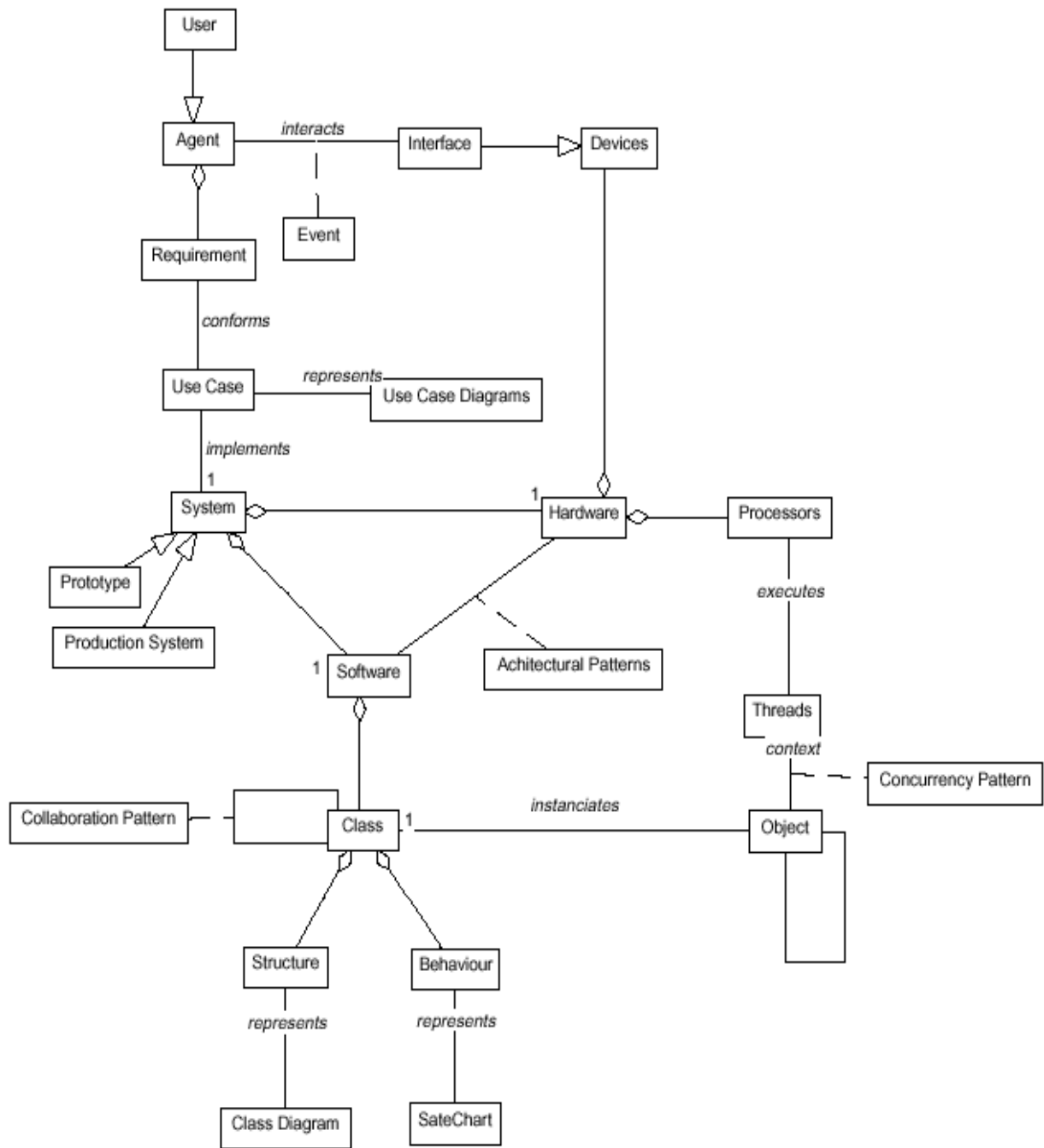


图9.1 嵌入式系统设计类图

附录 A. 模式: Reactive Subject

问题

一个反应实时对象（reactive real-time object）如何与一个交互非实时对象(interactive none real-time)通过事件进行通信。

环境

我们在一个程序中混合使用交互和反应对象，交互对象需要反应对象的信息来更新自己的状态。反应对象通常实现一个实时系统的核心。而交互对象则实现的是用户接口、系统监视或者数据记录等功能。

约束

嵌入式系统必须严格按照规定时间完成某些任务。交互系统就没有这个限制。因此也更易于设计和实现。反应对象不能规定交互对象的响应时间。

在某些应用中，交互对象需要处理反应对象产生的所有事件，而有些情况只需要处理最后一个包含有足够更新信息的事件即可。

解决方案

反应对象（Reactive Object）向事件代理（Event Proxy object）发送事件。事件代理保存该事件的指针。这个保存过程必须在反应对象发送下个事件 之前完成。交互对象（Interactive Object ）定期调用事件代理对象的checkEvent 检查是否有新事件。如果存在新事件，事件代理返回该事件的指针给交互对象。否则返回一个空指针。交互对象通过返回的指针检索状态信息，在这个过程中，交互对象不执行其它操作。事件代理对象提供一个checkClearEvent方法来检查某个事件是否被处理，如果是则从时间表中删除该事件。

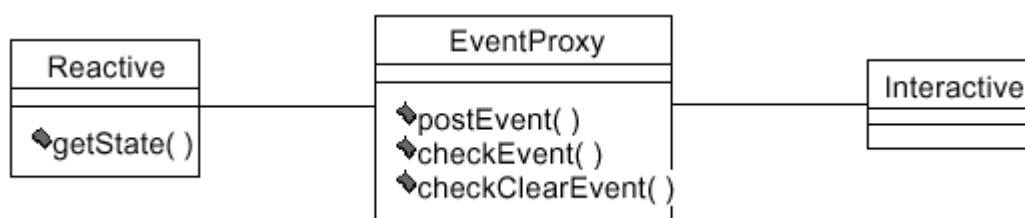


图 A.1 事件代理（EventProxy）类图

基本原理

通过对此系统模式的扩展，我们把软件中反应的、面向控制的部分和交互的、面向用户的部分解耦。这会带来如下好处：

首先，某个对象不用知道其合作对象的地址甚至不用知道合作者是否存在。

这使一个反应对象通过向另一个反应对象发送消息以获取状态信息成为可能。

第二，各对象不是共享一个线程，因此也就不需要对每个执行过程进行锁定和干预。这就使实现事件代理功能而不需要任何互斥技术和竞争机制成为可能。同时，

这也需要系统能够自动读写对象变量的指针。最简单的实现方法是将事件表保存在一个对象指针数组中，并按事件类型建立索引。

例子

前文所描述的AlarmClock 对象TimeView 对象之间的协作关系是一个很好的Reactive subject模式的例子。

umlchina 提供 UML/OOAD 培训

通过讲述一个真实 N-Tier 系统的开发过程，使学员自然领会 OO 技术。概念并不按部就班讲授，只是由实例带出。

详情请联系 think@umlchina.com

参考书目

- [AKZ96] M. Awad, J. Kuusela, J. Ziegler, Octopus: Object-Oriented Technology for Real-Time Systems, Prentice Hall 1996.
- [BJR1] G. Booch, I. Jacobson, J. Rumbaugh, editor. UML Notation Guide (Version 1.1). Rational Corporation, Santa Clara, 1997.
- [BJR2] G. Booch, I. Jacobson, J. Rumbaugh, editor. UML Semantics (Version 1.1). Rational Corporation, Santa Clara, 1997.
- [CY90] P. Coad, E. Yourdon. Object-Oriented Analysis. Yourdon Press, Prentice Hall, 1990.
- [Dou98] B. P. Douglass, Real-time UML: developing efficient objects for embedded System, Addison Wesley 1998.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, j. Vlissides, Design Patterns, Addison-Wesley 1995.
- [Gog98] M. Gogolla, UML for the Impatient, University of Bremen, FB3, Computer Science Department.
- [HEVB] Hitachi Ltd., Low-Cost Evaluation Board US7032EVB1 User Manual.
- [HLCD] Hitachi Ltd., Low-Cost Evaluation Board for liquid crystal displays LCMEVB-001 User Manual.
- [HSH1] Hitachi Ltd. SH7000/7600 Series Programming Manual.
- [HSH2] Hitachi Ltd. Hitachi Single-Chip RISC Microcomputer SH7032 and SH7034 Hardware manual.
- [ITU727] International Telecommunication Union. Recommendation G.727 (12/90) - 5-, 4-, 3- and 2-bits sample embedded adaptative differential pulse code modulation

(ADPCM) 1990.

[MO92] J. Martin, J.J. Odell. Object-Oriented Analysis and Design. Prentice Hall 1995.

[SR] I. Porres Paltor, Hardware Overview of the Digital Sound Recorder.

<http://www.abo.fi/~iporres/sr.html>

[SS95] Ed. Seidewitz, M. Stark, Reliable Object-Oriented Software, Sigs Books 1995

原文链接 <http://www.umlchina.com/zippdf/TR234.zip>

umlchina 提供 UML/OOAD 培训

通过讲述一个真实 N-Tier 系统的开发过程,使学员自然领会 OO 技术。概念并不按部就班讲授,只是由实例带出。

详情请联系 think@umlchina.com

去往讨论组→
(已超过 11,000 人)



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/>

怎样避免用例陷阱

讨论请点击

Susan Lilly 著, [qian_x_j](#) 译

无论是你的系统边界是模糊的，还是你在用户接口中的用例被搞得混乱，你很快会发现你一开始建模时会很容易趋向于一些相当可预见的问题领域。不过你走运了，这也很容易避免。

用例是一项日益流行的技术，用来文档化系统和软件的需求。凭借它们简单的图形符号和接近自然语言的规格定义，用例对开发团队颇具吸引力——甚至那些对常规需求规范经验甚少的团队也是如此。然而，这个简单是有欺骗性的。虽然开发项目的团队使用用例，一开始没有遇到大麻烦，但是许多团队在较大的场合下应用用例时，会遭遇到相似的问题。

虽然从实践中学习可能是件好事，但是这在商业世界里却是昂贵的教育方法。你的项目团队不必要亲身经历这 10 个最常见的陷阱。我有四个建议，可以帮助避免许多常见的用例陷阱，并且其中有一个将帮助你发现并且移除陷阱。

用例的前 10 个陷阱

1. 系统边界模糊或者变化无常。
2. 用例描写来自于系统（并非角色）的观察点。
3. 角色名称互相矛盾。
4. 过多的用例。
5. 角色和用例的关系连接象蜘蛛网一样（复杂）。
6. 用例规格叙述过长。
7. 用例规格叙述混乱。
8. 用例没有正确描述功能。
9. 用户不可理解用例。
10. 用例从来不会结束。

来自于 Lilly, S., 用例陷阱：在实际项目中使用用例的前 10 个问题，TOOLS USA '99, IEEE 计算机社学报，1999。

[1]系统在哪儿？（系统边界）

要清楚系统边界。用例模型具有直接的符号。在它的最简单的形式中，它仅仅是一个标志性的盒子来描述系统边界，而角色（棒人形）画在盒子外面，用例（椭圆形标记）画在盒子里面。线和箭头连接着角色和用例。

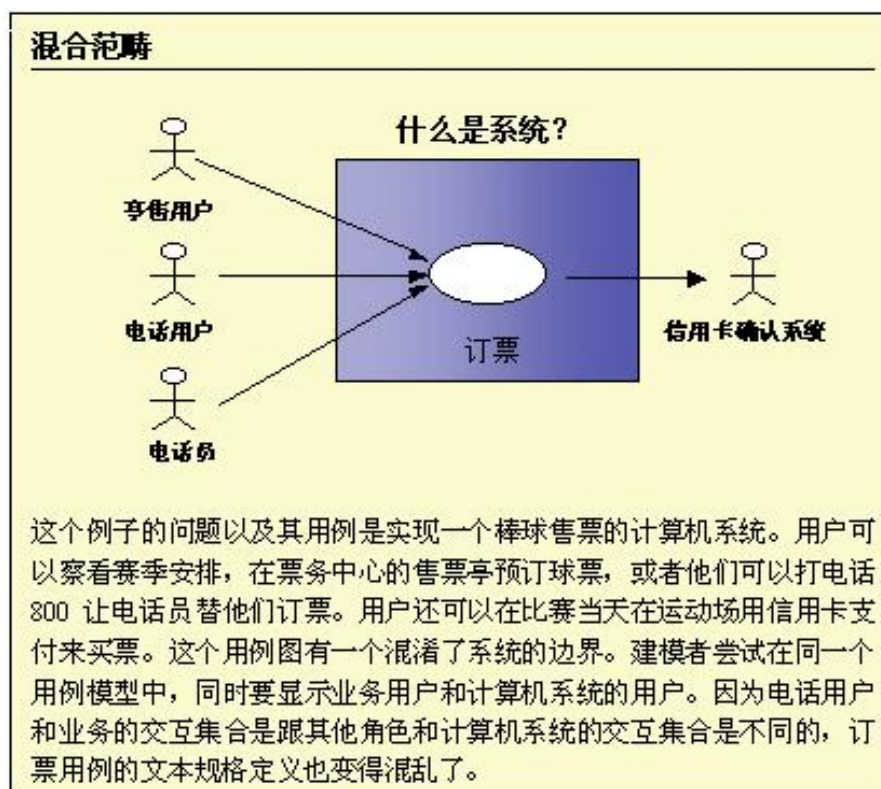
但是我们称为系统的盒子是什么呢？我们是在谈论计算机系统么？是一个应用程序？是一个子系统？抑或一整个商业企业？用例可以合理地描述这些任何的系统边界，但是它们应该在一个时间里只注意一点。特定的角色和用例对某个系统边界是恰当的，却很可能对另一个不同的系统边界是不正确的。

让我们看看一个棒球订票系统的例子吧。一个亭售用户（售票亭买票的客户）使用计算机来订票，或者一个电话用户拨打电话号码 800 来进行这次票务生意，而电话员（一个售票业务的雇员）使用计算机系统来订票。谁是角色呢？在混合系统边界的用例图中，建模者试图在同一个用例图中同时体现业务用户和计算机用户。订票用例的文本规格定义也变得混乱：因为电话用户和业务的交互的集合是跟其它角色和计算机系统的交互的集合是不同的。

边界的不明确或者矛盾（陷阱 1），可能是在我所观察过的许多第一次使用用例的项目中，最普遍的问题。当系统边界被搞乱了，你就很难知道要输入什么，要输出什么。你的工作可能会由于以下情况而终止：

- 。过多的用例（陷阱 4）。
- 。角色和用例的关系混乱（陷阱 5）。
- 。庞大的，却又混乱的用例规格定义，尝试描述多个范围层次（陷阱 6 和 7）。
- 。用例不能终止，因为你的团队试图让“整个大洋燃烧起来”（尝试分析每件事情）。

去往讨论组→
(已超过 11,000 人)

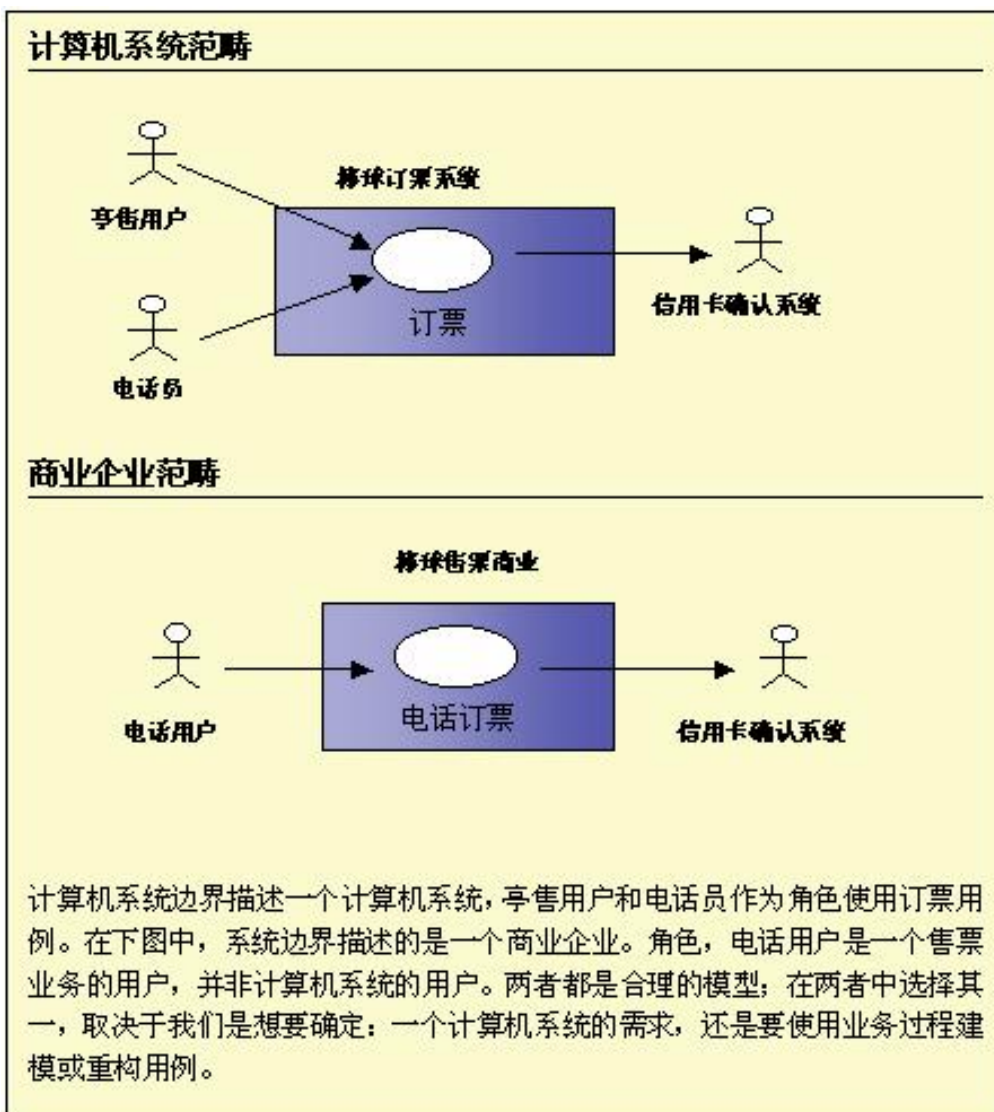


这些问题可以通过明确范畴并标出系统边界来避免。比如，系统边界可以描述为计算机系统，亭售用户和电话员作为订票用例的角色。或者，系统边界可以描述为整个企业。电话用户作为角色，是一个售票业务的用户，但并非计算机系统的用户。两者都是合理的模型；如何选择它们之中的一个，取决于你正试图定义计算机系统需求，还是在业务过程建模或重构中使用到的用例。

一个相关的问题是，整个系统的边界丢失。这种问题经常发生在用例模型使用可视化建模工具（比如 Rational Rose，市面上面向对象建模的头号工具）上，它们不允许在用例图中放置边界盒。

如果你正在使用类似的工具，你仍应该努力使系统边界明确。即使它（边界）并不在图上，它也应该在你的脑子里。在图上放置角色和用例时，要假象那个盒子在那里。

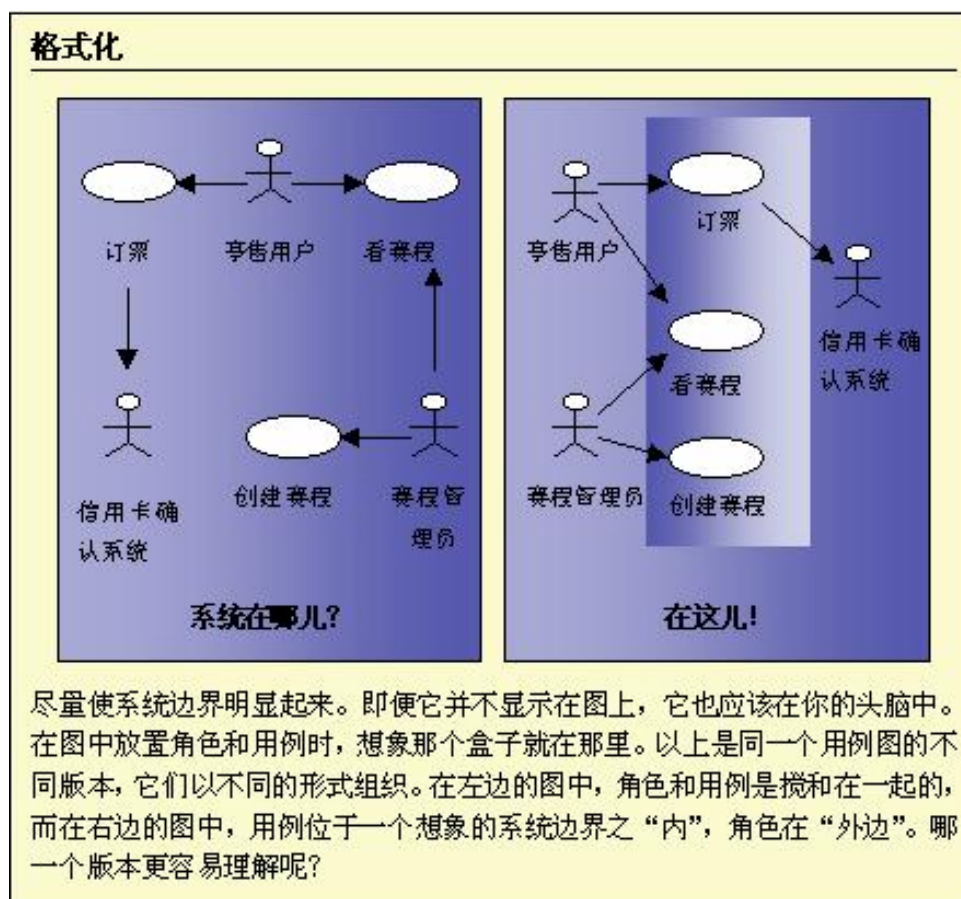
去往讨论组→
(已超过 11,000 人)



umlchina 提供 UML/OOAD 培训

通过讲述一个真实 N-Tier 系统的开发过程，使学员自然领会 OO 技术。概念并不按部就班讲授，只是由实例带出。

详情请联系 think@umlchina.com



[2]模板

为你的用例规格说明使用一个标准化的模版。虽然用例符号已经成为面向对象管理组织的统一建模语言（UML）中的标准，但用例的自然语言规格却没有。若想要在你的工程中成功的书写用例规格说明，为它们创建一个模版吧（当你真正需要使用它时就更适合了）。模版提供了用例和鼓励坚持项目标准的可靠性。

由于缺乏工业标准，许多相似的领域集里的用例规格模版被提议。**用例名（Use Case Name）**，就是出现在用例图中椭圆形下面（或者里面）的名字。**角色（Actor）**就是启动用例的主要角色。从对于主要角色或角色们的观察来看，**目标（Goal）**就是用例目标的一个简洁描述。当**触发器（Trigger）**作为具体事件导致用例启动，在用例应用中的**环境（Context）**就是背景“故事”了。**正常流（Normal Flow）**包括几个步骤，它们描述在角色和系统之间通常发生的最典型的交互序列。**变体流（Alternative Flows）**是正常（可成功的）流的变体路线，每一个都描述了它的触发条件（是什么导致从正常流跳入），过程和结果。**异常流（Exception Flows）**是错误的路线，每一个都描述了它的触发条件（是什么导致从正常流跳入），过程和结果。还有一个可选的问题领域是，指出所有的假设前提，就是用例是基于什么的。至少，用例规格应该回答这些基本问题：

- 。谁？（角色）
- 。为什么？（目标和/或环境）
- 。何时？（触发事件）
- 。什么？（正常流）
- 。还有什么？（变体流和/或异常流）

另外，对于在每个用例中的定义域，定义一个标准也很有帮助，那些信息是要为一个小组，或者一个相关用例的“包”而说明的。这些信息应该包括角色名和其它重要术语的术语表（防止陷阱 3），还包括交叉用例的业务规则，一个“故事”（或者动态图（active diagram））显示用例在时间过程中的关系，还有其它等等。

在我的公司，我开发了微软 word 形式的模版来定义用例包，具有自定义风格的主要元素及其帮助说明。你如果想要把基于 word 的文档自动的导入需求管理工具，自定义风格（word）是非常有用的。内建的帮助说明旨在方便那些用例书写者，因为他们不愿意在写文档时回过头去察看项目标准或者过程要点。它提供了，符合用例命名惯例的提示语（比如，以“action”动词开头的短语），规格定义规则（比如，使用主动形式的主谓句；避免逻辑分支），还有实用的提示。

设计一个工程的使用例模型，你不必从打草稿开始；你可以从网上获取例子并且裁剪它们。Alistair Cockburn 有一个网址(<http://members.aol.com/acockburn/papers/OnUseCases.htm>)，那儿提供了很多有用的用例支援，包含了用例模版，有 HTML 形式，微软的 Word 形式和文本形式。

[3]观察点（目标，而非琐碎的东西）

注意目标。在外观察以上来看，基于用例的需求对于传统的系统需求（比如，“Shalls”）的最大差异是棒人形了。这代表的意义并非完全是它的样子所要描述的，它旨在强调规格化系统要基于已定角色（人类用户和其他扩展实体）的观点。在面向用例的世界观里，系统的存在是因为，角色有一些让它们使用而满足的目标。我们书写用例，是要指出角色和系统之间交互的本质，结果就是角色目的满足。（Ivar Jacobson，用例之父，使用术语“值的结果（results of value）”来暗示在用例中考虑角色的真正需要或者目标的重要性）

把这一点牢记在心，我们应该能够扫视用例模型，并且列举那些在用户使用系统中想要做的事。这些并非是琐碎的用户的交互，而是真正的目标。选择那些反映附带的行为，而不是真正的角色目标，会导致

过多的用例（陷阱 4），并且导致用户的问题表述同基于用例的需求规格的脱离。

所以，要选择好的用例，就要注意那些反映角色的真正的目标。然后，从角色角度命名用例，而不是系统角度。

比如，“处理订票”和“显示比赛安排”是系统要做的，所以不是好用例名。“订票”和“察看比赛安排”是系统用户的目的，这才是好的用例名。

另外一个用户目标获取的迷失具有讽刺意味，某些建模者期望于用例“面向对象”。这种情形的症状是，存在臃肿的“CRUD”用例，那包括了所有的对于业务对象可能的行为。（“CRUD”是对象创建（Create），读（Read），更新（Update），删除（Delete）能力的首字母。）这些用例经常包含的名字有“维护”（“maintain”），“管理”（“manage”）或者“处理”（“process”）。

CRUD 用例有什么错呢？第一，也是首要的，它们经常不是直接的相关于角色目标。它们经常描述许多角色目标的联合体，来自于多个业务环境和过程，由于许多目标共享一个对象而被组合在一起。CRUD 用例还有什么错呢？它的规格定义是典型的过于冗长（陷阱 6），它经常会关联过多不同的角色（陷阱 5）并且这些角色对于整个用例的“功能定义”是典型的令人怀疑（陷阱 8；还有其它）。简言之，当我们运用面向对象方法分析时，很多时候我们要注意于对象，但是选择用例本身作为对象并非恰当。

不要把用例规格定义同用户界面设计混淆起来。有一个流行的观点，就是：因为用例有关于角色和系统之间交互，所以把用户界面放到用例规格定义中去是个好主意。粗粗一看，屏幕快照好像很有用，因为它们在规格定义中描述了角色/系统交互。

然而，坏处远大于好处。假如我想要使自己掉入了陷阱 10（“用例从来不会结束”），我可以把屏幕快照（界面）放入我的用例规格定义中。这儿有个我观察过的项目，曾经这样做：用例，试图定义系统操作需求，并且完结于包含系统的用户界面设计。一旦需求需要被核准并且被定为基准；这些设计元素，很自然的，还是要被更改。所以用户还是不能停止制作需求文档，因为它包含了已经完成了的用户界面设计，错误的和/或矛盾的。

即便是用例已经被核准，它会在短期内是成功的（短期维持）。然而用户界面设计很可能随着时间而需要更改。我们不想把系统需求依赖于设计。这种依赖应该用其它方式解决——用户界面设计必须满足用例需求。

在把界面放入用例的工程中，我还见到过其它问题。为了尝试反映在用例与屏幕之间一对一的通信，他们选择了反映大量用户界面而非用户目标的用户。（典型的，用户目标比用户屏幕“更加”值得交互。）

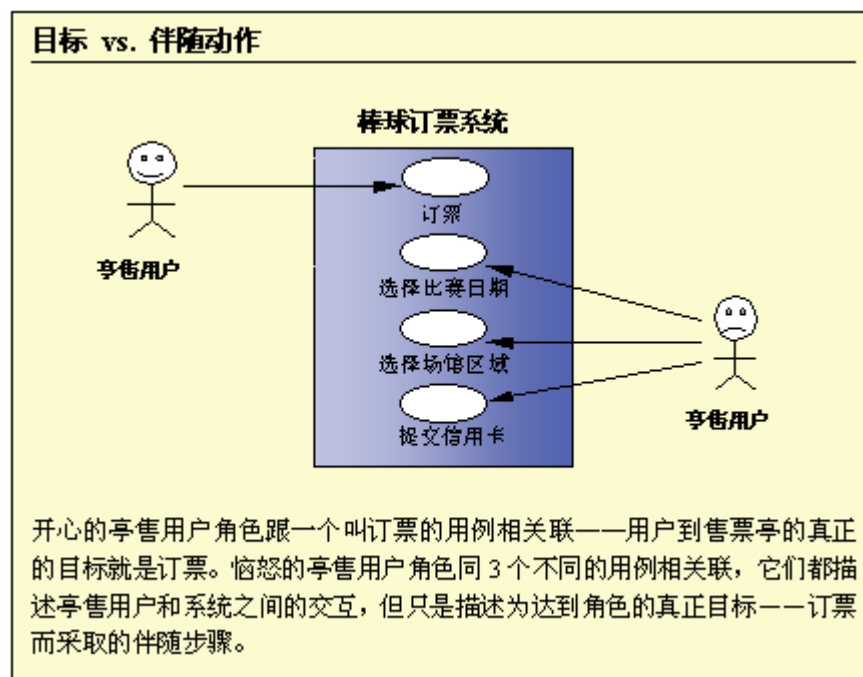
他们试图通过使用内建用例关系（“使用”或“扩展”），把大量的界面用例揉合到完全的用户目标中去。这样就导致了在用例模型中，形成了关系蜘蛛网，过多的描述了用户界面的切换，而忽略了用户目标和功能定义。用户界面切换图是一个有用的系统文档；然而，它是属于用户接口（设计）文档，而非用例（需求）文档。

一个更好的方法是，让用户接口细节和用例交互松散的耦合。一点点的耦合是合适的；包括“低分辨率”的用户界面图片，用于理解用例。但是不要过分的把功能定义同用户接口（UI）机制结合起来。用户接口更容易发生更改。在规格定义中，注重于建立角色要做什么（比如，“选择一个游戏”，“提交一个请求”），而非怎样交互（比如，“双击提交按钮”）。

另外，不易被发现的问题是，试图在用例和用户接口之间通信可能导致不正确的功能定义（陷阱 8）。

比如，若我们声称很多用户能够使用赛程安排界面来察看赛程，但是通过适当的验证，一个用户（比如，赛程管理员）还可以使用那个界面更新赛程安排。建模者决定把察看和更新功能组合到同一个处理赛程的用例，基于公共的赛程用户界面。

这个用例跟只能察看界面的角色关联，当然也跟能够更新的角色关联（也许是一个变体流）。这个结果是不正确的功能定义。只要看看这个模型，它呈现的是，亭售用户可以察看并且更新赛程，如同赛程管理员能做的一般。分离各个用例，要基于实际的用户目标和功能定义，而非组合基于用户接口，这样才会得出正确的模型。



[4]经常检查（复查）

复查你的用例图和规格定义。假如你不能防止所有的陷阱，你至少能够尽可能早的截取问题并且修正它们。这个主意并不是新的并且迷人的。由这种经典的方法，进行代码走读已经经历了 25 年了，而且相当于这种复查过程的应用，在成熟度模型等级 3（Capability Maturity Model’ s（CMM） Level Three）中是一个关键的过程。

怎样把它应用到用例呢？在我们公司的项目中，应用多层通过的复查过程，导致了好的结果。首先，用例图的复查是，通过察看用例规格定义的简略的存根（仅仅是用例名及其目标或者简单描述）。

在对于用例规格定义进行细化以前，对于草图，这种复查进行的越早越好。对于草图复查以后，由于图是正确的，用例规格定义的细化便可顺利完成了。然后，执行最终的用例图及其规格定义的复查。

我们发现两阶段复查过程要优于一次“大的”复查。在书写规格定义细节上投资许多时间以前，用例图中大量的组织性问题被揪了出来。当用例图需要重新组织时，这种方法缩减了必须做的重复工作的总量。

使用例复查有效的其它窍门：

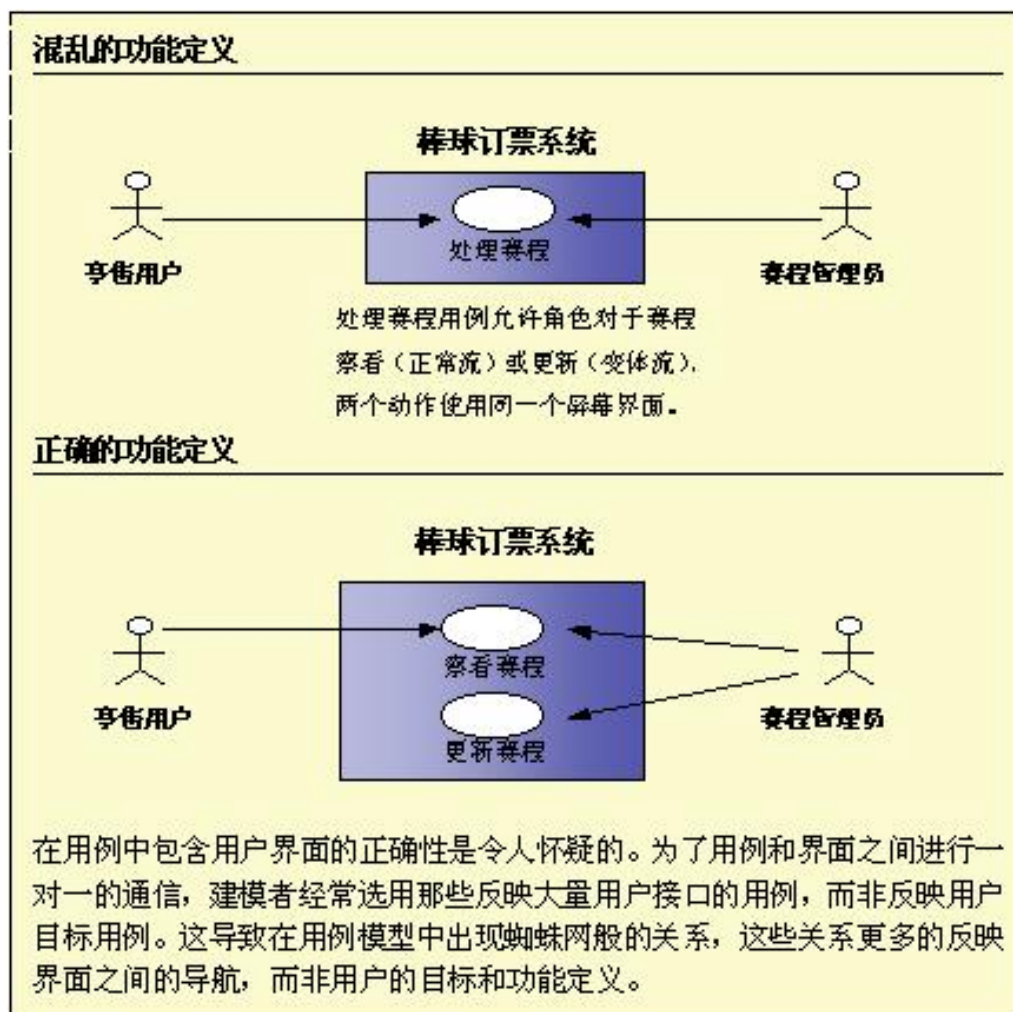
- 。使用复查校验表来帮助发现常见的用例问题。

- 。描绘原素材来“讲一个故事”。有时候，描绘图形素材，要比平实的文字要吸引人。我们建议列出一个“广度优先”的概览，然后是若干小组深入的相关的用例。比如，描述者可以概述用例图的一部分，它是相关于一个角色或者一个特定的业务过程；然后通过细节遍历这些规格定义，然后回到用例图，考虑用例集合的另外相关的部分。但是永远不要把按字母排序作为描述的策略！（是的，我真的看到过有人这样做）

- 。在最初的模型复查时，考虑使用非常规的复查技术。我喜欢使用“黄色粘纸墙上复查（Yellow Sticky Wall Review）”的方法（名称来源于几年以前在 IBM 对象技术大学中的经历），对非常规的图形模式进行复查。用例图被打印出来，贴在复查处的墙上。复查者在黄色粘纸上写上他们的注解，把纸贴在用例图上模型附近合适的部位。

常见的错误

我所描述过的陷阱并非对用例的控告；应该说，对于那些从来没用过它们的初用者来说，它们是典型的困难；并且多数应用用例的开发团队确实存在新手。



此外，用户，共同参与需求分析的最终使用者和领域专家们，一般没有使用用例经历。由于建模符号的简单，并且是自然语言的规格定义，所以很容易上手。

然而，简单的形式不应该掩盖一个真相，就是需求分析和定义并不是次要的工作。很多团队在初次尝试用例时遭遇了相似的陷阱。

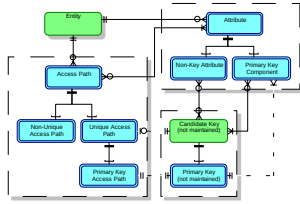
这些建议能够帮助你的团队避免这些问题。

去往讨论组→
(已超过 11,000 人)

原文链接 <http://www.umlchina.com/Indepth/howto.htm>

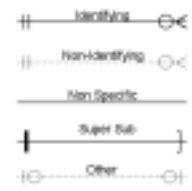
Access Paths

Access Paths specify the attributes that can be used to locate an individual record that is represented by an entity. Access paths correspond to indexes in the physical data model.



Relation Lines

Relation lines 表示实体间的关系，可能是父 / 子关系 (parent/child relationship)、Super-sub types or non-specific. The nature of the parent/child relationship can be further categorised as Identifying or Non-Identifying.



Physical Data Models

实体资料模型 (**PDM, Physical Data Model**) 是用来描述一个特定的资料库，因此，实体资料模型不同于实体关系图，它只建构一个已完成的资料库模型。实体资料模型 (**PDM**) 可以直接转换成实体关系图 (**ER Diagram**)，相同地，实体关系图 (**ER Diagram**) 也可以直接转换成实体资料模型 (**PDM**)。

Schema Generation

SA Schema Generator 准予您由实体资料模型 (PDM) 产生资料库架构，并且支援最普遍使用的资料库管理系统(DBMS)，SA Schema Generator 能输入至一个档案，或这个资料库能经由现存的 ODBC 做修改。

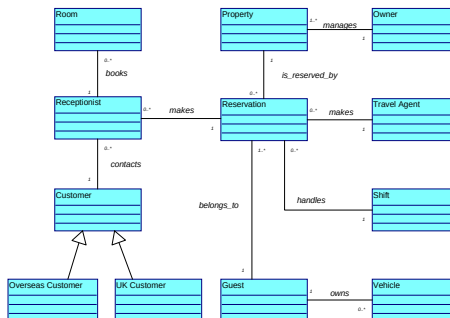


Reverse Data Engineer

资料逆向工程 (Reverse Data Engineering)，能让您将已存在的资料库结构转换成实体资料模型 (**PDM**)。转换介面也提供第四代程式语言 (4GL) 应用程式，例如：Visual Basic, PowerBuilder, Delphi and Magic。

ER Diagram to OO Model

您能由实体关系图 (**ER Diagram**) 自动产生一个 UML Class diagram ; 实体 (Entities) 对映到类别 (Classes)，属性 (Attributes) 对映到类别属性 (Class Attributes)。



System Architect 2001

Data Modelling

Quick

Reference



戊丰科技股份有限公司
台北市敦化南路一段 21 号 4 楼之二
Tel: (02)2570-1877 / Fax: (02)2570-1876

Methodology Outline

System Architect 2001 提供对商业模型、物件及元件模型、关联性资料模型及结构性分析及设计等四个面向的整合性支援。

Data modeling 包含实体关系模型（Entity Relation models with subject-areas）、独立的实体模型（physical models）、schema generation 以及资料逆向工程（reverse data engineering）。

System Architect 2001 provides the capability to build multiple project data models in a project encyclopedia. Project Data Model 能由实体关系模型图(Entity Relation Model Diagram) 及一个或多个 Subject Area Diagrams 所构成。图型及关系的定义显示出一个资料库系统，它可能是一个单一的资料库或一个分散式资料库系统。

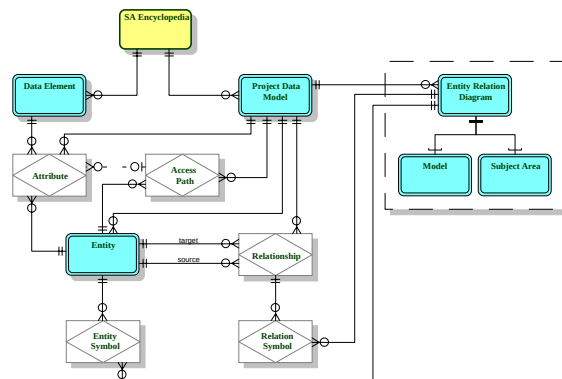
The Model Diagram 包含 project data model 内所有的实体及关系。The Model Diagram is built upon subordinate Subject Area diagrams. 每一个 Subject Area diagram 表示 Model Diagram 的一部份。

System Architect 2001 Features

- 32 位元使用者界面，及完整的企业模型整合工具。
- 多合一的电脑辅助设计工具（CASE tool），包含商业描述、物件导向、元件设计及资料模型。
- 支援的法论包括：UML, IDEF0, IDEF3, IDEF1X, Yourdon, Ward-Mellor, Gane & Sarson, SSADM, Catalyst, OMT, Booch and Shlaer/Mellor
- 支援多人使用的网路版本，整合 Microsoft VBA、互动式关联矩阵(Matrix)及影响分析。
- 可以产生 Microsoft Word 文件及 HTML 网页文件。
- C++, Java, Dynasty & Magic – Generation & Reversing, HTML Generation, Simulation.

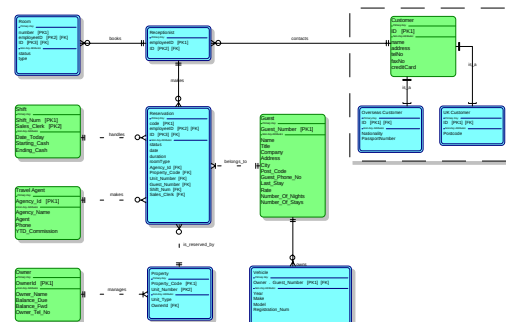
Project Data Model

Project Data Model 能由实体关系模型图(Entity Relation Model Diagram) 及一个或多个 Subject Area Diagrams 所构成。



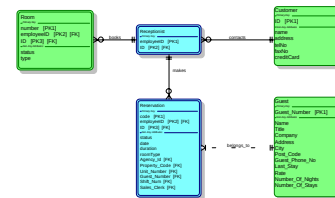
Model Diagram

The Model Diagram is built upon subordinate Subject Area diagrams. Subject Area diagram 的改变，会自动反映在 Model Diagram。



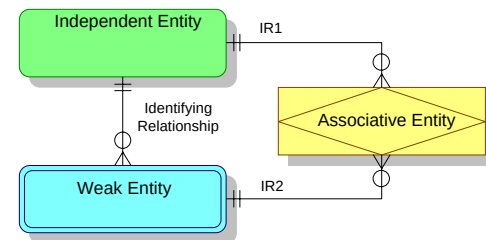
Subject Area Diagram

每一个 Subject Area diagram 表示 Model Diagram 的一部份。



Entity Symbols

实体（Entity）表示您企业组织内用来记录的资料内容，包含人、事、物、概念等实体。System Architect 2001 提供所有的实体符号(Entity Symbols)。



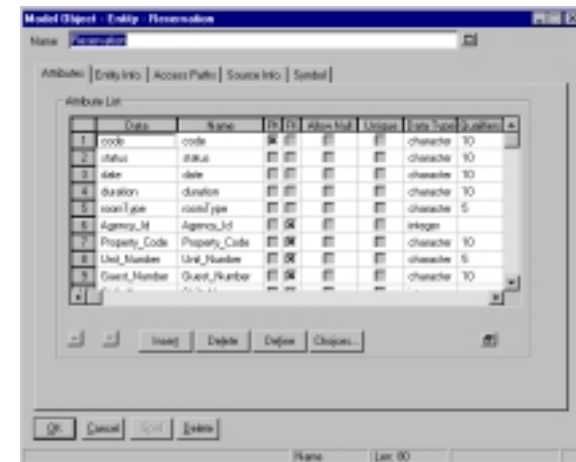
System Architect 2001 自动改变实体符号，以反映出适当的从属关系。

Entity Attributes

属性（Attribute）是定义实体（Entity）与资料项目（Data Item）间的关系。



The Attributes grid 准许您定义目前的实体及资料项目属性，也能被重复使用在其他的 SA design activities, process modelling, Data Flow diagrams.



使用 UML 设计数据库应用

Michael Blaha 著，张启鹏 译

我们的前一篇文章¹展现了一个用来选择数据密集型应用的数据储存系统的方法。有许多选择，包括内存内储存、文件、面向对象数据库和关系型数据库。这篇文章陈述如何用关系型数据库实现 UML 模型。假设我们熟悉 UML 对象建模表示法和关系型数据库。

1. 介绍

许多人认为面向对象概念和关系型数据库相互不一致，并且不能结合。事实上完全相反！经过灵活的使用，一个关系型数据库能够为面向对象（OO）模型提供一套优秀的实现。同样的模型能够用来开发编程代码和建立关系型数据库结构。

关系型数据库技术是意义深远的、强大的，但它比许多开发商使你相信的要难得多。单个表是简单易懂的、直观的，但是要彻底了解由数以百计的表组成（这是常见的）的应用是相当困难的。这正是 OO 模型有用之处。

OO 模型使你深入地、连贯地思考问题。OO 模型提供一种问题的超结构（superstructure）的思考方式，然后该方式能够用关系型数据库的更低层的组成块来实现。

本文章综合地讨论了关系型数据库技术，而不是集中于特定的产品上。我们将不讨论物理设计细节（例如存储分配和物理聚集），因为它们是依赖于产品的。

用关系型数据库实现 UML 模型有两个方面：映射结构（第 2 节）和映射功能（第 3 节）。第 4 节注解了面向对象到关系型数据库的扩展。第 5 节总结本文章。

2. 结构映射到表

UML 对象模型在本质上只是一个扩展的实体-关系 (ER) 模型²。用来设计数据库的 ER 模型的方式受到普遍接受，而我们将讲述一种近似的但更强大的方式—使用 UML 对象模型。OO 模型的主要优势在于编程和数据库使用相同的模型工作。而且，作为考虑功能性的一种方式 (第 3 节)，我们强调 OO 模型的导航。这一节显示如何实现 UML 对象模型的主要构造。

2.1 标识 (identity)

实现对象模型的第一步是处理标识。我们从定义几个术语开始。

- **候选键** (candidate key) 是一个或多个属性的组合，它唯一地确定某个表里的记录。一个候选键里的属性集必须是最小化的；除非破坏唯一性，否则属性不能从候选键删除。候选键里的属性不能为空。
- **主键** (primary key) 是一个特定选定的候选键，用来优先地参考记录。
- **外键** (foreign key) 是一个候选键的参考。外键必须包括每个要素属性的一个值，或者它必须全部为空。外键用来实现关联和泛化。

正常地你应该为每个表定义一个主键，尽管偶尔有例外。我们强烈建议所有的外键都只指向主键而不是其它的候选键。

定义主键有两种基本的方法：

- **基于存在的标识**。你应该为每个类表加一个对象标识符属性，并将它设为主键。每个关联表的主键包括一个或更多的相关类的标识符。基于存在的标识符有作为单独属性的优势，占位小且大小相同。只要你的关系型数据库管理系统 (RDBMS) 支持，基于存在的标识符就没有性能的劣势。（多数 RDBMS 提供有效的基于存在的标识符的分配顺序号码。）唯一的劣势是基于存在的标识符在维护时没有固有的意义。
- **基于值的标识**。一些真实世界的属性的组合确定了每个对象。基于值的标识有不同的优势。主键对于用户有固有的意义，容易进行调试和数据库维护。在另一面，基于值的主键很难改变。一个主键的改变需要传播到许多外键。而一些对象没有自然的真实世界里的标识符。

我们推荐你在超过 30 个类的 RDBMS 应用里使用基于存在的标识。基于存在和基于值的标识都是所有 RDBMS 应用的可行选项。

2.2 域（属性类型）

属性类型是 UML 术语，对应于数据库著作里的域的术语。比起直接用数据类型，域提升到更一致的设计，并便利了应用的定位。

简单域很容易实现。你仅仅要定义相应的数据类型和大小。并且每个用了域的属性，你都必须为每个域约束加入一条 SQL 检查子句。简单域的一些例子是：名字（name），长字符（longString）和电话号码（phone-Number）。

一个枚举域把一个属性限制在一系列的值里。枚举域比简单域实现起来更复杂，图 1 显示了四个方法。

实现方法	优势	劣势	建议
枚举字符。 定义一条 SQL 检查约束，把该枚举限制在允许的值里。	简单。受控的方便搜索的词汇表。	大的枚举难以使用检查。约束难以编码。	我们正常地选择。
每个枚举值一个标记。 为每个枚举的值定义一个布尔型属性。	回避命名的难处。	冗长 — 每个值一个属性。	当枚举值不是互相排斥的并且多个值可能同时地应用时使用。
枚举表。 把枚举定义存储到一个表里。不是每个枚举一个表，也不是所有的枚举一个表。	高效地处理大的枚举。不用改变应用的代码就可以定义新的枚举值	偶尔使用时很麻烦。必须编写通用的软件来阅读枚举表和加强值。	适合大的枚举和没有结尾（open-ended）的枚举。

枚举编码。把枚举值编码作为有序的数字。	节省磁盘空间。有助于用多种语言处理。	大大地复杂化了维护和调试。	避免使用，除非你要用多种语言处理。
---------------------	--------------------	---------------	-------------------

图 1 枚举的实现方法。

2.3 类

正常情况下，我们把每个类映射为一个表，每个属性映射为一个列。你可能因一个已产生的标识符（基于存在的标识符）、隐藏的关联（第 2.4 节）和通用鉴别器（第 2.5 节）需要一些另外的列。

2.4 关联

现在我们讨论关联的实现。我们已经把我们的陈述分为建议的映射（我们正常使用的映射），可选的映射（我们偶尔使用的映射）和不鼓励的映射（我们遇到的应该避免的错误）。我们所有的例子都采用基于存在的标识。

2.4.1 建议的映射

- **多对多关联**。用一个单独的表（图 2）来实现一个多对多关联。关联的主键是每个类的主键的合并。那些省略号 (...) 表示在模型里没有显示出来的属性。主键用黑体字体显示。
- **一对多关联**。把一个外键隐藏在“多”表（图 3）。角色名字成为外键属性名字的一部分。
- **零或一对一关联**。把外键隐藏在“零或一”表（图 4）。
- **其它一对一关联**。把外键隐藏在任一表里。

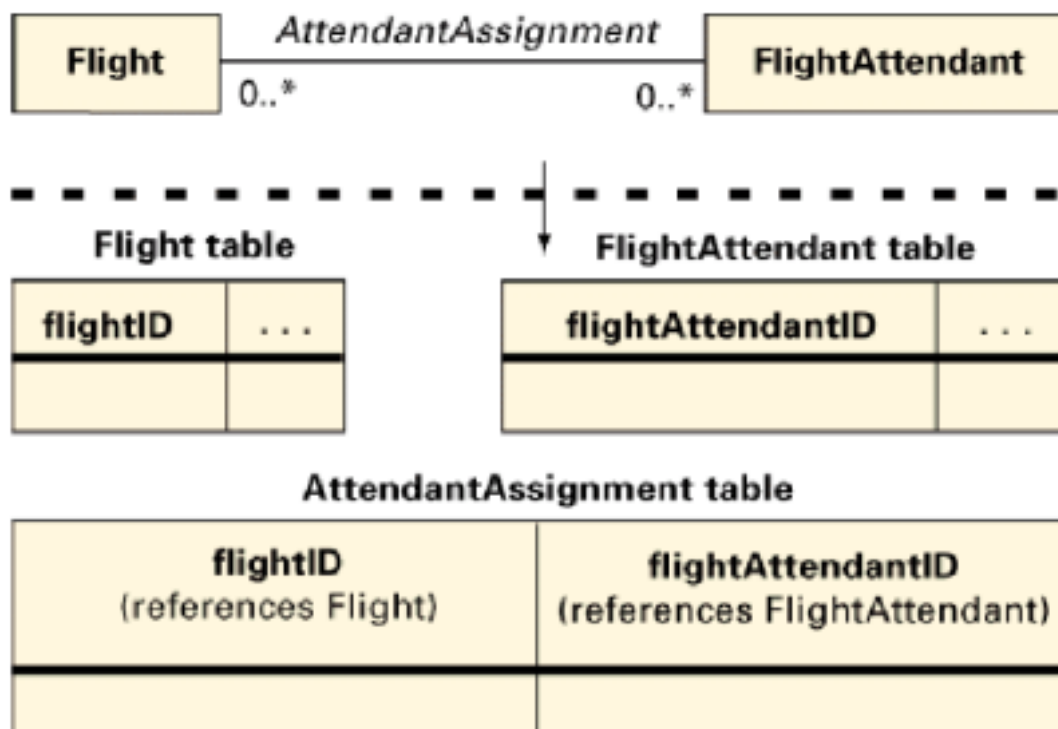


图 2 建议的实现：单独的多对多关联表。

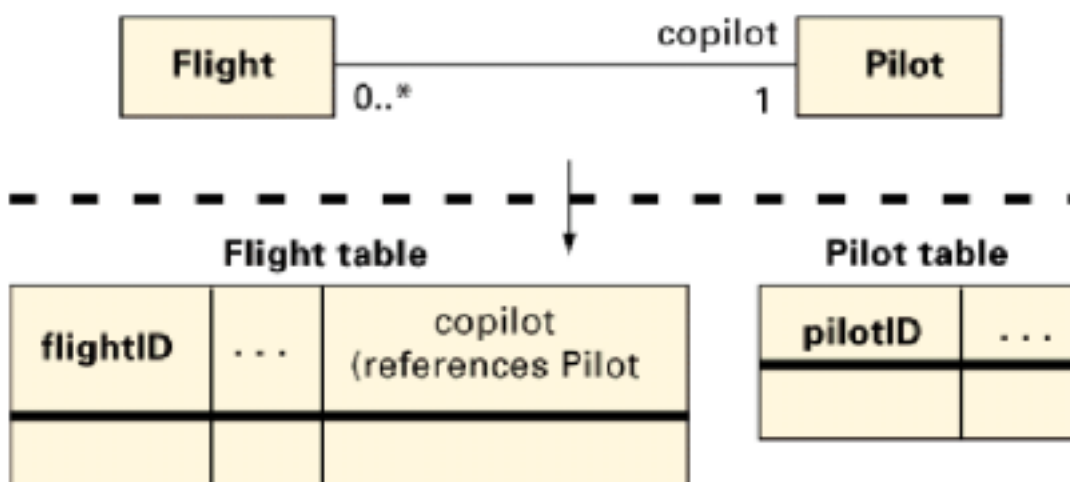


图 3 建议的实现：隐藏的一对多关联。

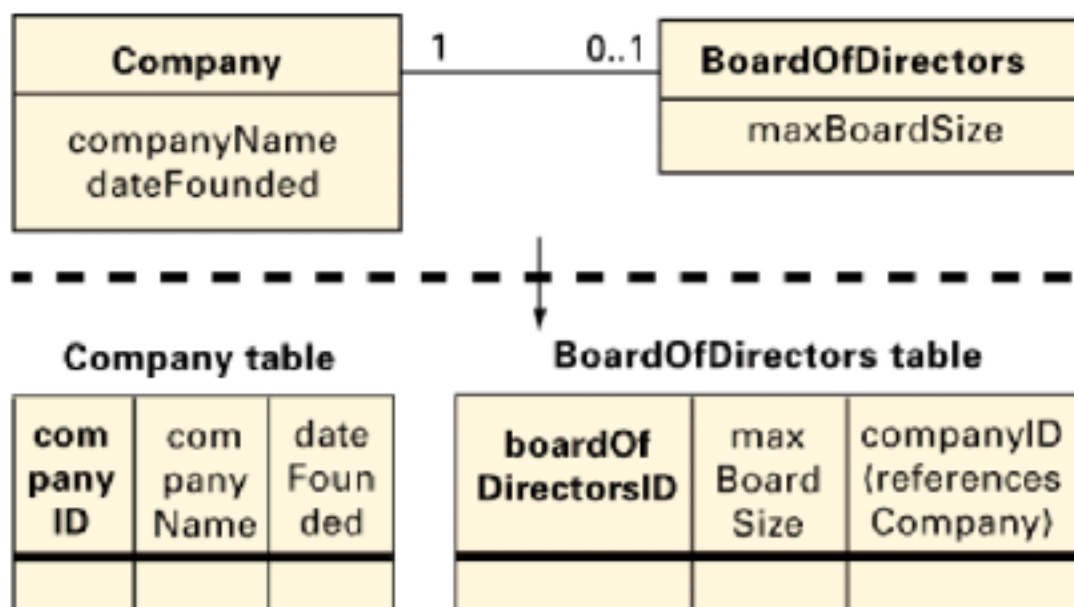


图 4 建议的实现：隐藏的零或一对一关联。

2.4.2 可选的映射

正常情况下我们使用建议的映射。但有些偶尔的情况，可选的映射更合适。

- **单独的表。**你也可以用单独的表（图 5）来实现一对多和一对一关联。单独的表给了你更统一的设计和更大的扩展性。无论如何，单独的关联表打碎了数据库，并增加了表的数量。此外，单独的关联表不能强迫一个更低的多重性限度为“一”。

2.4.3 不鼓励的映射

我们已经注意到有些开发者选择有缺陷的映射。我们要注意避免这些映射。

- **合并。**不要合并多个类，不要把关联强制成为一个单独的表（图 6）。这样减少了表的数量，但会干扰第三范式。
- **两次隐藏一对一关联。**不要把一个一对一关联隐藏两次，每次隐藏在一个类里（图 7）。这是多余的，无助于性能。
- **相同的属性。**不要用相同的属性来实现多个关联角色（图 8）。相同的属性使编程复杂，降低了扩展性。

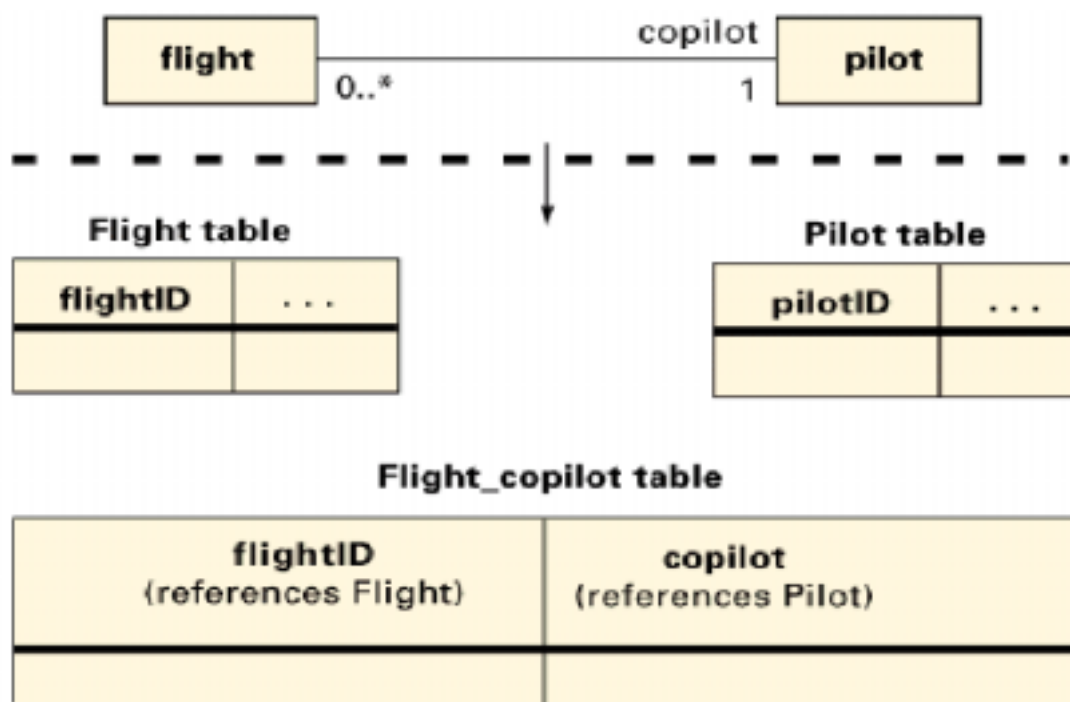


图 5 可选的实现：单独的一对 x 关联表。

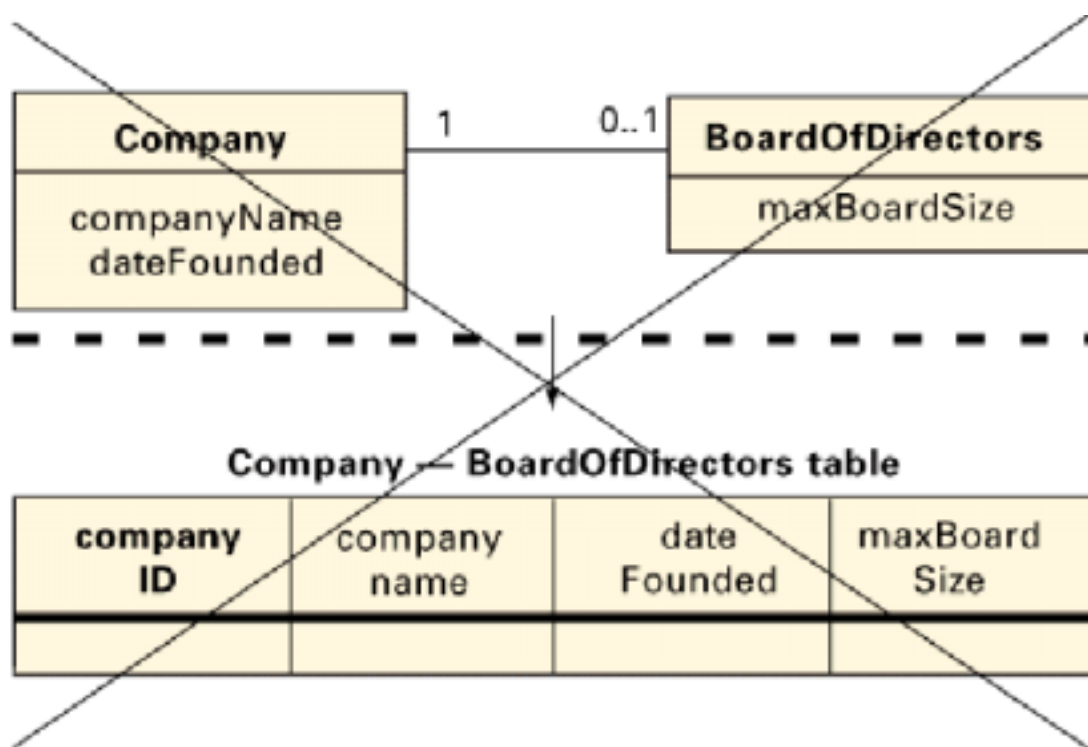


图 6 不鼓励的实现：类的合并。

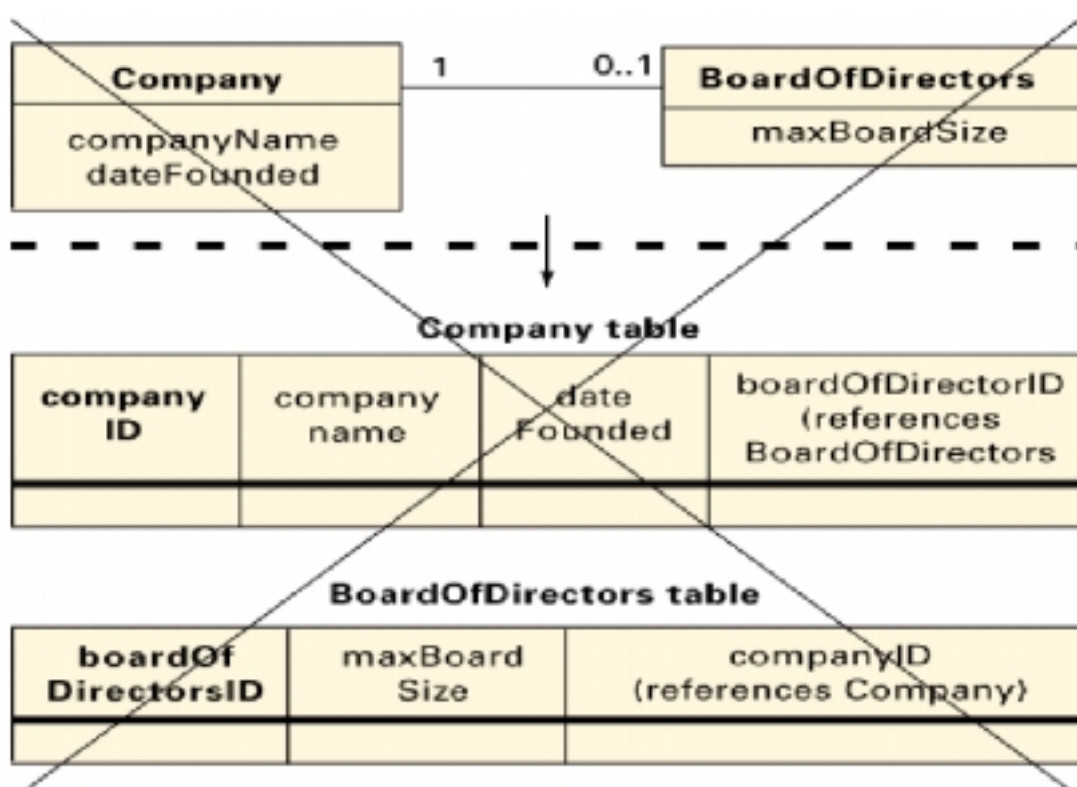


图 7 不鼓励的实现：两次隐藏的一对一鼓励。

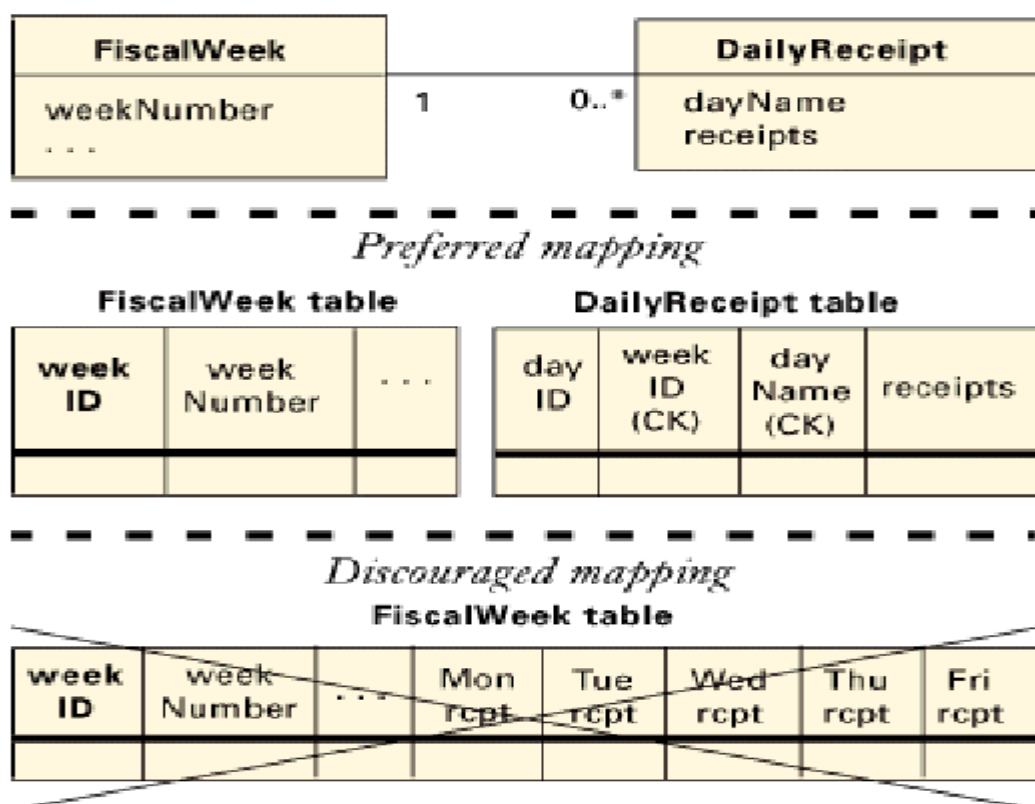


图 8 不鼓励的实现：相同的属性。

2.5 泛化

现在我们讨论泛化。我们这里只论述单个继承。

2.5.1 建议的映射

最简单的方法是只映射超类和每个子类为一个表。所有的表共享一个共同的主键。应用必须执行子类的划分，因为 RDBMS 并不能执行。（关于后者的详尽的描述，请参阅第 4 节。）

- **单独的表。**映射超类和每个子类为一个表（图 9）。所有的表共享一个共同的主键。鉴别器指出每个子类记录的适当的超类表。

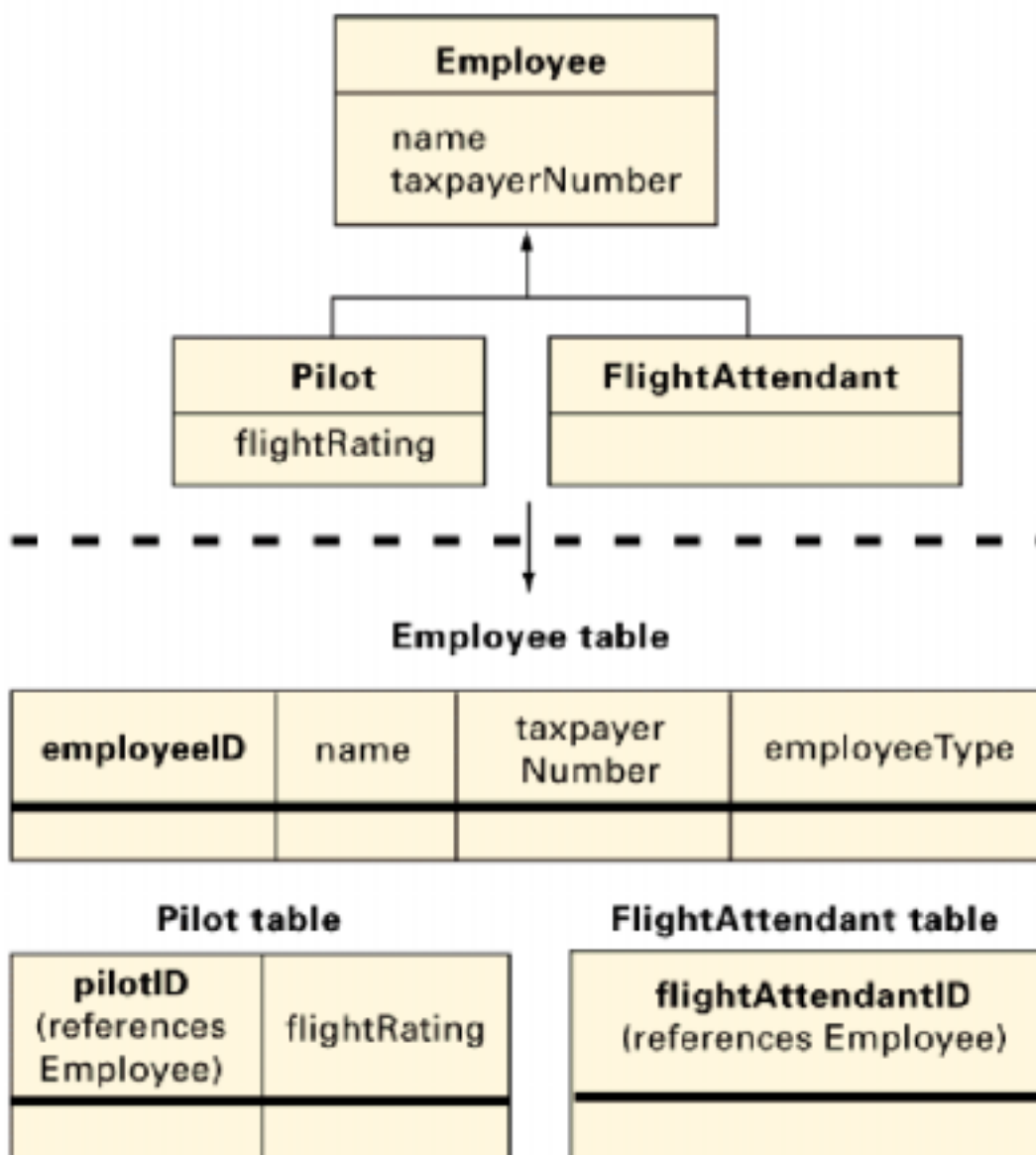


图 9 建议的实现：分开的超类和子类表。

2.5.2 可选的映射

泛化有几个可选的映射。

- **消除。**你可以优化除去那些除了主键外没有别的属性的类（图 10）。这样减少了表的数量，但提供更少的正规实现。
- **减少超类属性。**你可以除去超类表并把超类属性复制到每个子类（图 11）。这样可以有描述每个对象为一个表的优势。无论如何，它将引起数据库结构的冗余，你查找一个对象时可能需要搜索多个子类表。
- **增加子类属性。**作为第三个可选项，你可以除去子类表并存储所有的子类属性到超类表里（图 12）。这样用一个表描述每个对象，但干扰了第二范式。

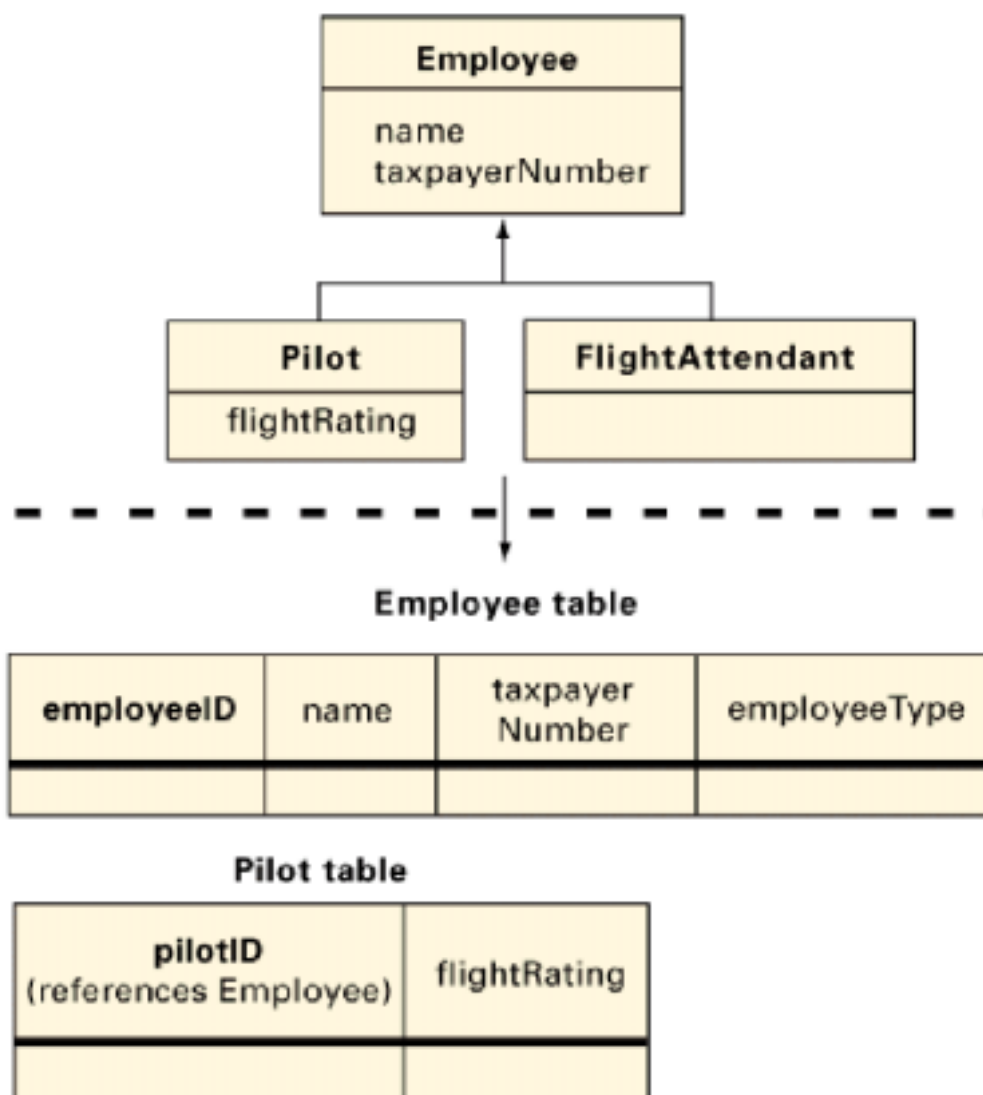


图 10 可选的实现：消除不必要的子类表

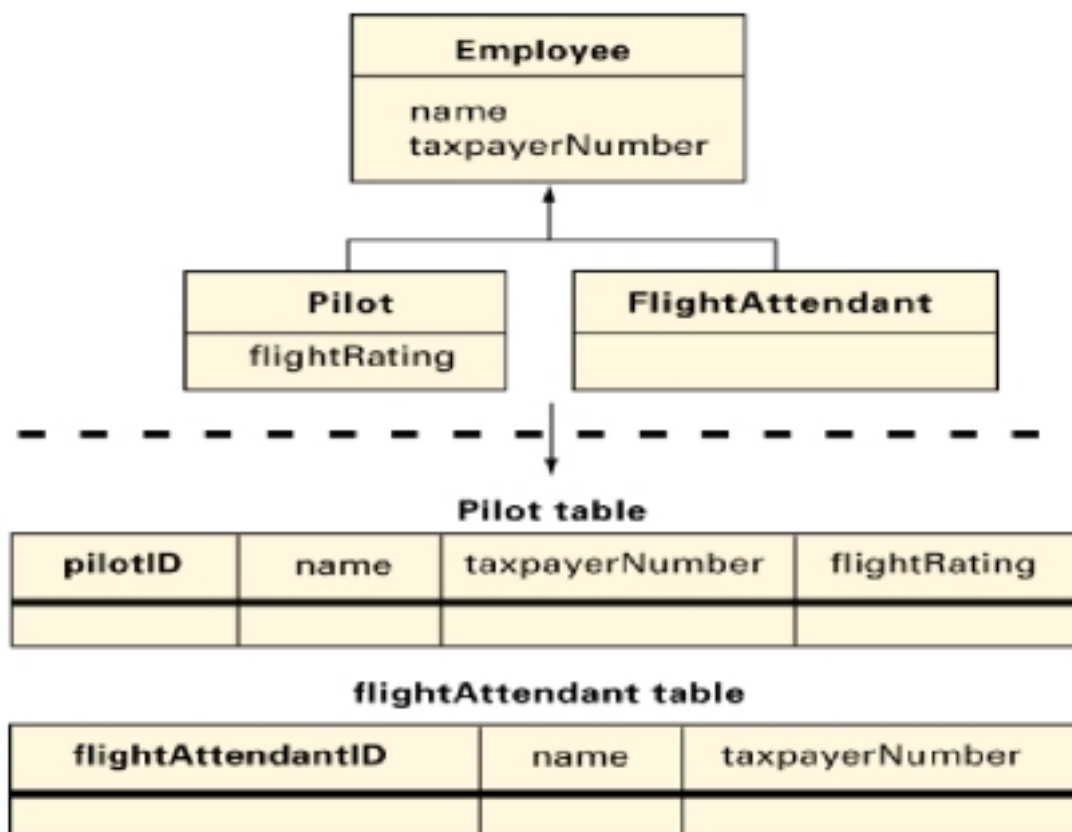


图 11 可选的实现：减少超类属性。

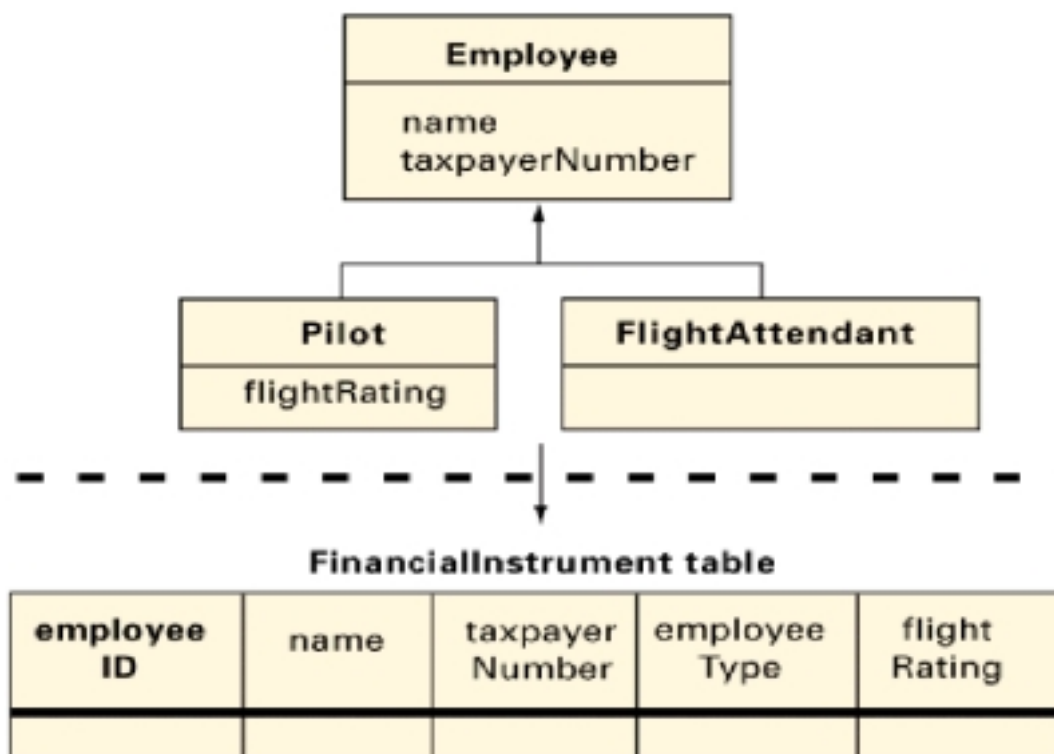


图 12 可选的实现：增加子类属性。

2.6 参考完整性

一旦你已经建立了表，你就应该定义参考完整性动作来明确对象模型的意义。（不要使用 SQL 触发器来实现参考完整性！）如果你使用基于存在的标识，你将不需要传播更新的结果。我们建议以下对删除的参考完整性方针：

- **泛化**。级联从泛化实现中产生的外键的删除。
- **隐藏的关联，最小多重性为零**。正常地把外键设为空，但有时候你可能要禁止删除。
- **隐藏的关联，最小多重性为空**。你可以级联一个删除的结果或者禁止该删除。
- **关联表**。正常地我们对关联表里的记录进行级联删除。可是，有时候我们禁止一个删除。

我们已经简要地论及参考完整性，因为它是个高级话题。参考文献³有更多的解释和例子。

2.7 索引

实现数据库结构的最后一步是加入索引来调整数据库性能。正常地，你应该为每个主键和候选键定义一个唯一的索引。（多数 RDBMS 作为 SQL 主键和候选键约束的副作用来建立唯一的索引。）你也应该为每个被主键或候选键所约束的外键建立一个索引。

我们强调索引的重要性。外键和主键的索引使在对象模型里能快速地遍历是不容怀疑的。你必须包括这些索引，否则你将使用户感到灰心。你应该在你的数据库开始设计阶段里加入索引，因为它们很容易加入并且也没有什么好理由推迟加入。

数据库管理员（DBA）可能为经常请求的查询定义了额外的索引。DBA 也可能采用产品相关的调整性能的机制。

2.8 范式

范式是关系型数据库设计的提高数据一致性的有效方法。我们的书³讨论了范式，但我们关于这个问题却说错了。我们将利用这篇文章的机会来澄清我们的观点。如果你不熟悉范式你可以跳过这节。我们的说明是针对关系型设计人员，他们正在尝试用面向对象适应他们原有的技能。

范式是正确设计关系型数据库的精确的原则。同样地，它们与使用了什么开发技术是无关的 — 基于属性的设计、基于实体的设计、面向对象设计或其它什么。

过去使用基于属性设计的方法，开发人员不得不非常注意范式；范式提供了分组数据的根据。相反地，范式对于基于面向对象（或基于实体）的开发不是很重要。如果你采用 OO 方法并且你的模型经过很好的构思，那你就正在把数据组织成为有意义的单位，也在本质上满足了范式的规定。如果你愿意，你仍能够检查范式，但这样的检查是不必要的。

2.9 摘要

图 13 总结了我们已经陈述的映射规则。这些映射规则的完整例子，包括一个 UML 对象模型，能够在这篇完整的扩展版本里找到（Adobe Acrobat PDF 文件）。

概念	对象模型构造	推荐的 RDBMS 映射
域	简单域	映射为一个数据类型和大小
	标识符	用 RDBMS 顺序号
	枚举	通常储存为一个字符串
类	类	映射每个类为一个表
关联	多对多关联	单独的表
	一对多关联	隐藏的外键
	一对一关联	
泛化		分开超类表和子类表

图 13 推荐的映射规则的摘要。

3. 把功能映射到 SQL 命令

对象模型为数据库应用提供三种主要的用途。

- **结构。**对象模型指明数据库结构。我们已经在第二节探讨了这方面。
- **约束。**对象模型也指明了能存储的数据上的重要的约束。相匹配的实现必须为迎合这些约束而努力。我们的映射规则的处理方法以及第二节里的参考完整性指出了许多约束。（本文没有论及的另外的 UML 构造，能获取更多的约束。）
- **潜在估算。**一个对象模型指明潜在估算；它是关于引起哪些查询和如何公式化的蓝图。第三节将

简要地阐明第三个目的。

对象模型不仅仅是被动的数据结构，相反它们能够帮助你思考一个应用的功能。你可以根据遍历一个对象模型说出它的许多功能。例如，根据我们对一个模型检查用例时的遍历，我们进行思索。这强调对象模型的估算能力对于 RDBMS 应用是特别重要的，因为遍历表达式可以直接映射到 SQL 代码。

UML 对象约束语言（Object Constraint Language，OCL）有助于表达遍历。点符号导航从对象到对象和对象到属性，方括号表示对象集合的过滤器。我们加入冒号（:）操作符来表示泛化的遍历；因为我们正常地用多个表来实现一个泛化继承，清楚的遍历很有用。

图 14 里的遍历表达式例子是基于我们创建的 UML 对象模型上的（请参阅本文的扩展版本（Adobe Acrobat PDF 文件）），我们把它们映射为 SQL 代码。我们用冒号开始 SQL 编程变量。

遍历表达式	含义	SQL 代码
anAirline.Employee	给定的一条航线，找出相应的员工。	SELECT employeeID FROM Employee WHERE airlineID = :anAirline;
anAirline.Employee.name	给定的一条航线，找出相应的员工的名字。	SELECT name FROM Employee WHERE airlineID = :anAirline;
anAirline.Flight [getMonth(date)==aMonth].pilot	找出在某个给定的月份里为一条航行飞行的机师。	SELECT pilotID FROM Flight WHERE airlineID = :anAirline AND GetMonth(date) = :aMonth;
anAirline.Flight [getMonth(date)==aMonth]. pilot[flightRating==aFlightRating]	找出在某个给定的月份里为一条航线飞行的在一个给定的飞行等级里的机师。	SELECT P.pilotID FROM Flight F, Pilot P WHERE F.airlineID = :anAirline AND getMonth(F.date) = :aMonth AND F.pilotID = P.pilotID AND P.flightRating = :aFlightRating;

aFlight.copilot:Employee.name	为一次飞行找出副机 师的名字。	<pre> SELECT E.name FROM Flight F, Pilot P, Employee E WHERE F.flightID = :aFlight AND F.copilotID = P.pilotID AND P.pilotID = E.employeeID; </pre>
-------------------------------	--------------------	---

图 14 对象模型遍历和 SQL 代码的例子。

4. RDBMS 的 OO 扩展

数据库团体对 RDBMS 的 OO 扩展有兴趣。产品和 SQL 标准正尝试加入到 OO 扩展里。我们将简要地陈述一下这个技术的方向。

- **抽象数据类型 (ADT)**。这是个好主意，扩展 RDBMS 的能力。开发商为这个技术使用了许多名字，例如 Oracle cartridge 和 Informix data blades。ADT 的缺点是它们把你紧紧绑在特定的一个开发商上；ADT 的范畴超越了 SQL 标准。因此，你应该只在 ADT 的好处很明显的时候才使用。

在关于 ADT 如何适合数据库开发的著作里有一些混乱。如果你使用 OMT 开发过程，你能够用属性实现简单域，用 ADT 实现复杂域。你仍应该用表实现类。

- **SQL3 指针**。最新的 SQL 标准的版本，SQL3，加入了作为一种数据类型的指针符号。显然，其意图是支持导航和面向对象。我们对于 SQL3 指针最友善的评语是，它们是嫁接的，是可以忽略的。更深入的批评是，指针在理论上是荒谬的，增加了复杂性，又没有扩展 SQL 的表达能力。CJ Date 在上次的九月对象/关系型会议上雄辩地讨论了这一点⁴。

好了，那么我们冷淡地对待抽象数据类型和 SQL 指针的指责。但我们相当喜欢面向对象技术和关系型数据库。有两个为 RDBMS 的扩展可以使它们更容易用于 OO 技术。我们将很乐意看到 RDBMS 开发商把这些能力加入到他们的产品中。

- **扩展的参考完整性动作来支持泛化**。当前的参考完整性机制是单向的。为了完全支持泛化，我们需要一个双向的机制。这样，一条超类记录就可以依赖一条子类记录。并且，一条子类记录就可以依赖一条超类记录。我们通过例子可以最好地解释这点。

图 15 摘录于我们的在参考文献³的财务案例学习。我们用资产超类来统一某些没有显示在摘录里的通用的数据和功能。一项资产可以是一支股票或股票期权。一支股票可以有許多它的股票期权。例如，IBM

股票可以有許多写明达到价格和过期日期的期权。

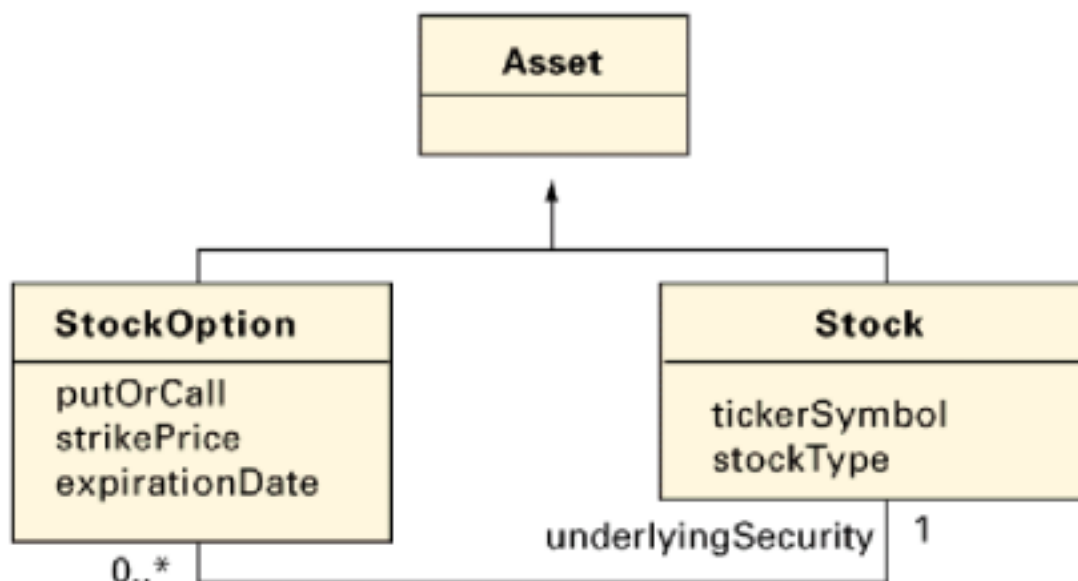


图 15 参考完整性和泛化的例子。

我们推荐的泛化实现是单独的表——映射该超类和每个子类为一个表。然后，我们就可以使用参考完整性使股票期权和股票的记录依赖于资产。一个资产记录的删除级联到相应的子类记录、股票期权或股票的删除上。我们也能够定义一个参考完整性动作，这样一个股票的删除就级联到关联的股票期权记录的删除。

现在问题如下：如果我们删除的一项资产是一支股票，资产记录的删除级联到引起股票记录的删除。随后，股票记录的删除级联引起所有股票期权记录的删除。但现在参考完整性使我们失败了：一条股票期权记录的删除并不引起一项资产记录的删除。删除级联只能从超类走到子类。为了完全的行为，级联应该双向地走下去。

当前有用的参考完整性的工作是做更多的编程（也即做更多的工作和承担更多的故障风险）。在我们的案例学习的实现里，用户随时要删除一项是股票的资产，我们不得不书写额外的代码来首先检查关联股票期权的存在性，然后删除它们。

- **支持交叉表的记录划分。**单继承（泛化的最常见方式）的含义是一个超类的每个实例都是用最多一个子类来例示。现在的 RDBMS 不能容易地加强这个约束。例如，没有什么防止下面的情形。一支股票可以用 ID18 加入到资产表，用 ID18 加入到股票表，并且也可以用 ID18 加入到股票期权表。再一次地，我们为了确信行为的完整，不得不作额外的编程，而不是写一个简单的声明约束。

5. 结论

本文陈述了用关系型数据库实现 UML 模型的快速的概观。我们希望本文向你演示的技术是十分适用的。一个训练有素的开发人员能够用关系型数据库准备一套优秀的 OO 模型的实现。如果你要关于实现机制的更多的细节，参考³有另外的信息，并且也覆盖了我们没有在这里讨论的一些高级模型建模结构。

关于作者

Michael Blaha 是纽约 Schenectady 的通用电气研发部的毕业生（译者按：这是作者幽默的说法，意思是他已经跳槽了）。在过去的五年里，他已经成为面向对象技术、建模、数据库设计和逆向工程领域的独立的产业顾问。Blaha 博士是多篇论文、五个专利和两本书的作者。可以通过 www.omtassociates.com 或 blaha@acm.org 和他联系。

William Premerlani 从 1975 年开始就在通用电气研发部供职。他的主要研究兴趣在软件工程、元建模（metamodeling）、数据库技术和复杂工程应用等领域。Premerlani 博士是许多论文、二十五个专利和两本书的作者。通过 premerlani@acm.org 和他联系。

原文链接 <http://www.umlchina.com/Indepth/usinguml.htm>

¹Michael Blaha and William Premerlani. Object-Oriented Design of Database Applications. Rose Architect 1,2 (Winter, 1999).

² PPS Chen. The Entity-Relationship model - toward a unified view of data. ACM Transactions on Database Systems1, 1 (March 1976).

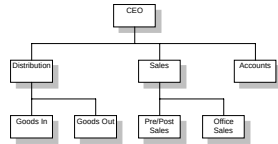
³ Michael Blaha and William Premerlani. Object-Oriented Modeling and Design for Database Applications, Prentice Hall, Englewood Cliffs, New Jersey, 1998.

⁴ CJ Date. Don't mix pointers and relations! Presentation at Third Annual Object/Relational Summit sponsored by Miller Freeman, Washington D.C., September 16-19, 1998.

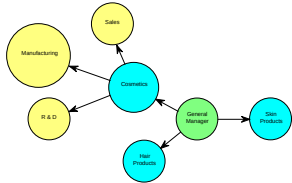
Organisation

组织模型提供一个可浏览组织的现状及未来的功能，包括组织文化、劳动力结构及能力。

Organisation charts 组织图能说明组织报告的关系及各组织单元之间的关系。



Decision Making Charts 决策制定图是企业高层视各组织单元间的关联，做为下决策时所依循的参考模式。



Location

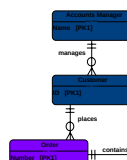
The Location view 是描述商业场所的型态，一般场所及实体的场所。也可能是描述在某建筑物里的场所。

Elementary Business Process	Location Type	Head Office	Warehouse
"Accept Order"		☒	
"Analyse Order"		☒	
"Assemble Packing List"			☒
"Reject Order"		☒	

The **Process / Location Type matrix** 显示某个作业程序在那个场所型态里被完成的。

Data Modelling

System Architect 2001 利用实体模型（Entity Modelling）来描述实体、属性及实体间的关系。

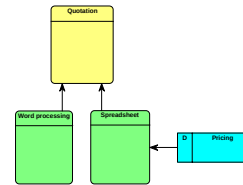


Application

Capture details of the current and future applications software required to support the business functions.

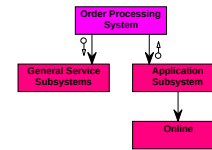
Application Area Map - high-level graphical representation of current applications.

Application Architecture Diagram - 企业高层描述欲建构的应用系统以满足商业的需求。



Application Context Diagram - high-level data flow diagram showing an application with its interfaces to existing or new computer systems.

Subsystem Structure Diagram - decomposition of a subsystem into programs.



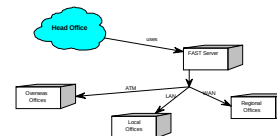
Logical Process Hierarchy - uses an object-oriented approach to form mapping between Business processes and UML.



Flow Diagram - free-form flow of information or control at subsystem level.

Technology

Network Concept Diagram - 让企业高层了解企业整体的网路概况。



System Architect 2001

Business Process Modelling

Quick

Reference



戊丰科技股份有限公司
台北市敦化南路一段 21 号 4 楼之二
Tel: (02)2570-1877 Fax: (02)2570-1876

Methodology Outline

System Architect 2001 支援企业模型架构 (Enterprise Modelling Framework) 让您能轻易地驾驭企业的改变及塑造资讯系统模型。

System Architect 2001 支援六个主要的产业元件, 任何公司能在完成商业流程分析及设计的过程中都应该考虑到这六个元件。

Process 定义企业要做什么及如何做。

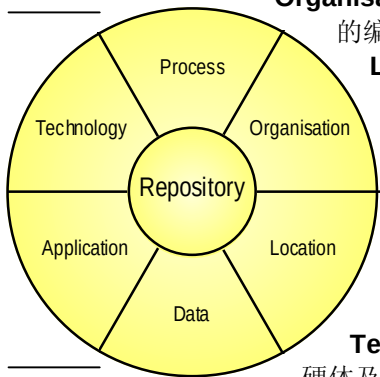
Organisation 定义企业内人员的编制。

Location 定义企业营运的场所。

Data 定义组织内应被记录的商业资料规则。

Application 详细描述软体支援企业结构的功能。

Technology 说明企业内硬体及通讯设备。



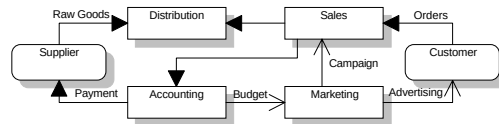
System Architect 2001 Features

- 32 位元使用者界面, 及完整的企业模型整合工具。
- 多合一的电脑辅助设计工具 (CASE tool), 包含商业描述、物件导向、元件设计及资料模型。
- 支援的法论包括: UML, IDEF0, IDEF3, IDEF1X, Yourdon, Ward-Mellor, Gane & Sarson, SSADM, Catalyst, OMT, Booch and Shlaer/Mellor
- 支援多人使用的网路版本, 整合 Microsoft VBA、互动式关联矩阵(Matrix)及影响分析。
- 可以产生 Microsoft Word 文件及 HTML 网页文件。
- C++, Java, Dynasty & Magic – Generation & Reversing, HTML Generation, Simulation.

Business Process

企业流程模型 (Business Process Model) 提供所有流程及功能的分析, 也提供基层作业流程分析, 并显示出动流程及必须的流程, 使作业流程能完成工作。

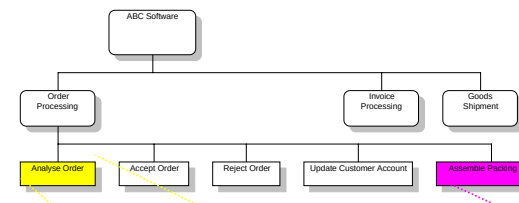
Relationship maps 显示功能间或群组间的相互影响。



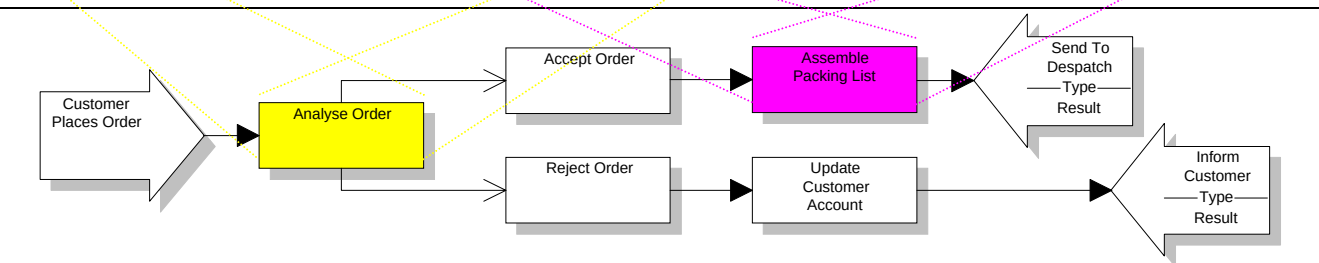
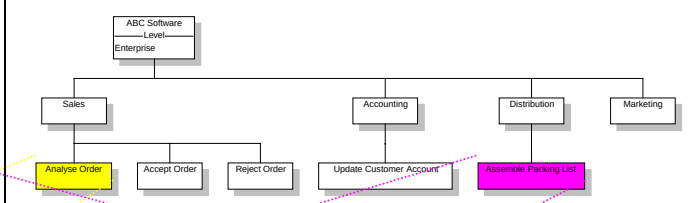
一个 **EBP(Elementary Business Process)** 是商业活动的最小单元:

- Is worth examining
- Is performed by or in support of a single user
- May include both manual or automated activities
- Has its user involvement occur at one time

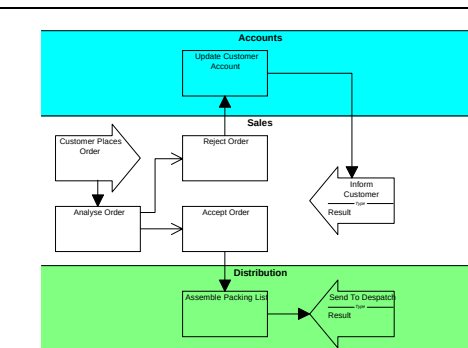
Process Oriented Hierarchy Diagrams 此图是将作业流程依流程群组作分解。



Functional Hierarchy Diagrams 此图是将作业流程依功能群组作分解。

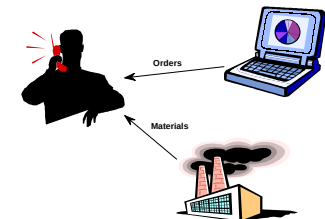


Process Chart 显示一连贯的作业流程。包含: Events, Results, Iterations, Flow Breaks, optionality and exclusivity. **Process Chart** 表示整个或一部份的 Process Thread。



Process Map 规划个别的作业程序应归属于那个组织单位内被完成。

Business Concept 以类似卡通的图案提供企业高层了解作业流程如何工作。



统一变更管理的威力

Brian White 著, [James Zhong](#) 译

变更管理 (CM) 术语是指一个组织或项目用来计划、执行以及对软件系统变更进行跟踪的流程和工具。统一变更管理 (UCM) 是 Rational 与我们的客户联合开发的一套特定的变更管理流程。UCM 在管理文件、目录、组件和系统的生成与修改等方面为软件项目小组提供支持。从学术上说, 变更管理流程由两条 workflows 组成:

- 软件配置管理 (SCM)
- 缺陷与变更跟踪 (DCT) 【批注: 亦称作变更请求管理 (CRM), 此处避免与“客户关系管理”混淆。】

SCM 处理版本控制、工作空间管理、软件集成、软件编译 (Build)、软件分发及发布等流程。DCT 处理流程与过程, 通过这些流程与过程, 缺陷、增强型请求与新特性被提交、评估、实施、验证及完成。

Rational 拥有两种工具对这两条 workflows 提供相应支持。首先是 Rational ClearCase, 自动化 SCM 相关流程。其次是 Rational ClearQuest, 自动化 DCT 相关流程。同时使用这两种工具, 就能自动化 UCM。事实上, 使用 ClearCase 和 ClearQuest 能够自动化几乎任何变更管理流程, 但是如果需要为变更管理提供黑盒支持, 那么 UCM 是最佳选择。

在 Rational, 我们已对问题“什么是 UCM 流程?” 提供了多种方式的回复。我们提供产品文档、论述 ClearCase 与 UCM 的书、可免费订购的多媒体光盘。因此, 如果已经对 UCM 有了一定了解, 可能会问: “UCM 相对其它变更管理流程的优点是什么?” 这里将对此问题作一阐述。

首先要说的是, 一种流程可能不是对所有软件项目都是最合适的。因而, 脱离实际软件开发项目的环境来描述 UCM 较之其它变更管理流程更优, 并无意义。所以相反, 将要描述 UCM 与传统变更管理 (CM) 流程的差异。然后你就能决定怎样将这些不同点应用到自己的软件开发项目中去。

用 UCM 进行更高级别的抽象

如果你观察软件开发语言，具有几十年历史的计算机科学与工程，显而易见，其抽象水平已从机器代码级大为提升。在最底层，全都是 1 和 0，只能期望非常早的程序员在此级别上工作。汇编语言很快出现，它对 1 和 0 进行抽象以提供雏形的机器指令如“load register X with value Y”。接下来出现了 Pascal 与 C 这样的语言，它们提供了更高水平的指令诸如“if-then-else”语句。那么，在今天，我们开始认识到可视化“编程”的潜力。通过对软件系统的行为建模，我们能让代码被生成。随着这些抽象的引入，更复杂的软件系统编程对开发人员来言已变得更容易和快速。

至于 CM 工具的演变，类似的情况正在发生。最初，CM 工具仅仅由存储版本的存储库构成：文件及目录在保存并标识的给定时间点上的内容，可以必要时重新取出。接着，出现了允许用户管理工作空间的工具：可为特定任务或活动选择特定版本的文件及目录集合。同时，当类似存储库及工作空间这样低层次的抽象变得普遍起来并被广为接受时，高水准功能可凸显出来以简化变更管理流程。UCM 就是干这件事的。让我们看看 UCM 包括的三个核心抽象：项目、组件基线、活动。

项目

典型地，软件开发小组以项目方式组织。这些项目，有时有子项目等等，有时没有，因此一个项目可能是非常大或非常小的。从变更管理角度来看，按项目方式组织与以下三个目标相适应：

- 第一，它标识了小组成员。这对安全目的及协作目的是有用的，两个目的对良好的变更管理都至关重要。
- 第二，项目限制了小组需要关心的文件和目录的范围。也就是说，对于全部文件和目录，在特定公司正使用的整个存储库中，项目标识了指派到该项目的某个开发人员需要考虑的精确子集。
- 第三，项目标识了小组成员正在完成的工作中的公共集成点。

所有这些可能听起来非常地稀松平常，但是由 ClearCase 和 ClearQuest 实现的 UCM，其关键优势在于，项目实际上是存在于 CM 系统中的物理对象，该对象映射到现实世界的一个项目。与作为非 CM 工具概念的项目相比，可能达到的自动化与安全程度高得多。当新的开发人员加入一个 UCM 项目时，举例来说，他们被自动给出一个由他们所需文件与目录的正确版本预配置好的工作空间。

组件与组件基线

UCM 的第二个核心抽象是组件与组件基线的概念。大多数的版本控制系统包含了存储库的思想在里面，它存储文件与文件版本的集合。一些工具，像 ClearCase，还将这些文件组成到目录中去并存储目录与目录的版本。几乎所有重大开发成就都有大量数目的文件，这些文件包含了开发中需要编译生成系统的代码。这些文件通常组织成目录，这种组织常常以系统的软件架构来排列。在传统的 CM 系统中，这些关键目录从不以不同于其它目录的方式对待。然而，UCM 引入了组件的概念，以区分并存储这些目录。UCM 组件就是一个目录树，该树由具有一个组件根目录的文件及目录组成。

UCM 缘何如此？

对项目来说，UCM 组件的主要优点是更好的自动化。理解这一点的最佳方式是着眼于基线的概念。基线标识了文件版本的一个相关集合。基线，换言之，是在组件中选定每个文件的单一版本。几乎所有版本控制工具声称为基线化提供支持，但如果你仔细观察，通常你会发现他们真正支持的只有标签概念。标签是这样：你选定一个标签名并将此标签名称贴至一个或多个文件版本。通过将相同名称贴至多个不同的文件版本，你得到一条伪基线。

用这种途径完成的基线存在着这样的问题，标签名称并未包含语义——除了那些如何使用工具的说明。无法通过查看一个标签就能知道什么文件与之相关联。事实上，当你在研究哪些文件具有该标签之时，同时，该标签可能被关联至新的文件、移至新的版本、或者从所选文件移除。当然，你能实施控制和锁定以强制你自己的标签语义，但 UCM 基线可为你解忧。这些基线在语义上是富涵 (Rich) 对象，可标识 UCM 组件的一个“版本”。通过使用它们，可确信组件中的所有文件都与相同版本关联。还能够确信基线不会从你的掌握下改变。一旦创建，UCM 基线就不可改变，并且可用于定义更高层次的配置。一个完整的系统，举例来说，可由组件基线的集合装配起来。

活动

可能关于 UCM 的最大特色是，它是基于活动的变更管理模型。这意味着什么？意味着根据变更原因对文件的变更进行了分组。假定你正在修复一个缺陷或实现一项增强，比方说。不管何时你更改一个文件，你通过在签出时间声明一个活动，标识正作此变更的原因。活动可能是缺陷、增强型请求，或者仅仅是变更的一行描述，这依据你的缺陷与变更跟踪流程所需的严格度而定。UCM 支持所有这些类型的活动——以及被选来定义你自己的其它任何类型的活动。

基于活动的变更管理的主要优点是，如果没有相关理由，那么没有文件能被更改。第二个优点是，变更被集成（或晋升）为单个、一致的整体。大多数时候，当你作变更时，你需要修改多个文件。例如，

如果修正一个缺陷，可能需要修改一个 C 文件和一个头文件。你时常需要修改很多的文件。对于 UCM，所有要做的就是选择“活动”来录制为所有文件而创建的全部新版本。正如对项目 and 组件一样，UCM 将物理活动对象引入至 CM 系统中，它映射了一个现实世界的对象：“工作单元”。这具有立竿见影的好处：当你完成了一项指定任务时，比方说，你可以仅通过签入活动，而达到一次签入你所有工作的目的。

不过，还会有更深远的自动化与信息化益处。UCM 在活动级别上推动贯穿系统的变更。也就是说，如果准备集成变更，可以“交付”活动。这与其它 CM 方式不同，它们需要使用文件集合或人工向某人发送物料清单，此人再列出包含于变更的版本。

事实上，基于活动的方式最大的好处在于活动与基线的联合工作方式。在组件被许多人修改以后，创建了一条新的基线。通过活动与基线的使用，使基线间不同点的判断流程的自动化成为可能。基线间的比较不仅产生自一条基线到另一条基线的变化文件清单，还有活动的清单！这具有巨大的好处：能自动生成发布版本说明；协助测试人员判断所必需的回归测试集合，以在每晚的编译后运行；等等。

基于客户系统

本文针对 UCM 的诸多能力及优点，权作抛砖引玉。从根本上讲，软件项目变更管理流程——通过 Rational ClearCase 和 Rational ClearQuest 来自动化——提升了抽象级别，以及通过将现实世界对象引入到 CM 系统中的自动化可行性。这些对象有项目、组件基线和活动。如果你是 Rational ClearCase 的长期用户，可能会在 ClearCase 的定制中意识到 UCM 流程的点点滴滴。很多这些基于脚本的变更管理流程，建立于 ClearCase 基础之上，在定义 UCM 的涵义中扮演了主角——同时将在决定 UCM 的未来变化中继续如此！

umlchina 提供 UML/OOAD 培训

通过讲述一个真实 N-Tier 系统的开发过程，使学员自然领会 OO 技术。概念并不按部就班讲授，只是由实例带出。

详情请联系 think@umlchina.com



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/>

项目管理规范-RUP 管理实施

（第四部分）

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

4. 目前实施项目规范的考虑

由于目前软件开发实际情况的约束，完全按照 RUP 实施项目管理是一个理想化思路。在实际实施中会受多方面的影响（如客户要求、小组成员素质等），这种盲目的追求标准只会做成项目的阻碍，基于这种考虑，我认为根据实际情况采用分步实现的方式，逐步建立起一支稳定的、技术分工合理的开发 TEAM，才能采用 RUP 管理规范实现省时、合理的软件开发。

下面我根据自身对项目管理的认识，对原有项目管理中的问题进行总结，并对目前如何实现软件工程，及对本项目管理规范的实施进行了相应说明。

4.1. 对项目管理的几点感受

目前我们的软件开发多为瀑布式开发，项目管理可以说是没有。所有的管理与软件设计都是由项目经理一个人来负责，这样对项目经理的要求较高。项目开发主要依赖于个人能力，开发中又总是依靠程序员高手来支撑，最好是要有一批“快枪手”。而高手又因为多种因素不能集中在一个项目中，同时高手的流动会给项目致命的打击。以上等等只是项目管理中的一部分问题。我对项目管理中之不足归纳为以下几个方面：

4.1.1. 人员素质

项目开发人员的个人素质不同，根据素质进行角色与行为划分是项目管理中的一个非常重要的环节。

- 责任心

我在项目管理中曾遇过这样一种情况，有个项目成员技术不错，但是在开发的关键时期为个人的一点小事而不顾整个项目的进度，而其在项目中担任了比较重要

的职务，在他看来，项目不如自己个人的小事（可做可不做）重要，所以造成项目进度的延误，我不得不去物色另外一个人来代替他。但后果已成。

还有一种情况，安排了某人工作，工作量在你的估算内，但是到了检查时，他对你说“我不会做”，“我不知如何做”之类的话，而同时他还在上网或做其他的事，但作为已有计划的你来说，则是项目进度被打乱的后果。

这种人当然在下一个项目不会被用，但是如何避免这种情况的发生呢？我认为应当将工作量细化到天甚至于小时，遇到这种员工则可立即开除，同时损失的工作时间是可控的。从而减轻了对项目开发的影响。

● 个人能力

程序员的领悟力各不相同。对不同的人要采用不同的方法来工作。

对个人能力高的人你可以务虚的谈工作任务，他在接受任务的同时就可以提出这样那样的问题，甚至想到你没有想到的问题，直到把问题搞清楚。这样的人你可以放心的让他去做，因为他已经完全理解了你的意图。

对个人能力一般的人，你讲什么他不能理解的很清楚，有的为了面子不懂也不说，但是在做的时候出了问题，在你指着他做的程序大骂的同时，你的项目进度在拖延。对于这种情况，你只能写出对他工作的要求，写出你的意图，明确工作目标。

对个人能力差的人你不能说只是写个要求就行的，那样的话他写的程序依然会让你忍无可忍，怎么办，把系统中技术要求较低的部分分出来，然后按 1，2，3，4 步骤写出来。有人说这样还不如自己写完算了，但是这种工作往往是重复量很大。如做界面等。

当然，RUP 的实施方针希望是淡化个人力量，而注重流程与大家的协作，同时细化了系统的每一部分。但是 RUP 的实施不能教条化，在实行中还要根据项目实际情况而变化。

● 自我约束能力

自我约束能力主要是指对管理者而言的。因为项目成员工作基本是由项目管理者来监督实施的，那管理者呢，可能没有人来管理，如果管理者的自我约束能力不强，那样建立什么样的软件工程规范都无法实施。

在实施项目管理规范时，由于条件限制，不可能一下完备软件工程所需的各个机构部门，从而也可能不存在与项目组平行的流程管理机构，质量审批机构等。由

于采用逐步实现的方法，所以提高约束力只能采用其他方法。

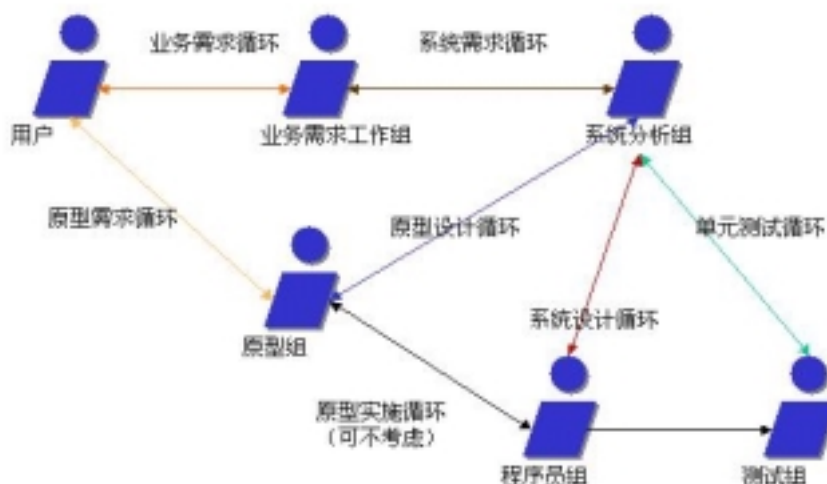
针对这种情况，可以采用各种提醒方式：电子系统如掌上电脑；人员提醒如部门技术秘书等。

4.1.2. 工作流程

4.1.2.1 工作流程中的多重循环

用户需求的不断变化是令项目开发中最头痛的问题了。我们在做项目时传统的做法是，新需求来了，赶快叫程序员们改，在这里加一块，那里补几行代码，只要实现就行。结果是项目做完了，再看自己设计的系统已面目全非。还谈什么可重用性，可定义性，产品化，甚至下一个差不多的项目还要花费同样的时间。

RUP 中强调的**循环概念**就是针对于这种情况提出的，通过项目小组之间各种角色的循环，实现对用户需求变化的问题解决。如下图



这样采用迭代法开发可以实现此问题，是的，但是要有所牺牲，那就是时间。时间充足我们自然可以按照标准的 RUP 思路来进行。而实际上呢，又是我们牺牲不起的。现在国内大多数项目具有这样一个共同点：用户要求急。如我们做一个基金的核心业务系统，用

户说开发时间为两个月。而由于基金是个新行业，无可利用之部件，两个月的时间是不可能进行迭代开发的。同时我们还要在一定程度上避开需求不断变化。我建议采用的方式如下：

- **按阶段地进行循环**

实现阶段性的循环，并且存在多个循环的并发。如业务需求循环在一个阶段内将生成相应的模型与业务需求文档，并得到用户的认可，然后转入系统需求循环，在系统需求转入另一个循环时，再继续业务需求循环，但此时只是记录用户需求，而不进行下一个循环。待根据最初需求制作的版本生成后，再根据新的业务需求及与用户协商结果再决定是否开发下一个版本。

- **采用快速原型法**

快速原型法的作用是为了让用户在感性上了解系统的概貌，通过与用户的交流，从而固定了相应的元素（如输入参数、跳转顺序等），通过快速原型法与用户进行交流，能很好的理解用户的意图与需求，是缩短开发周期的良好途径。

- **生成一个版本**

用户需求是不断变化的，而且任意性很大，如果一个项目根据用户需求的变化而变化，那么开发周期是无法估量的。一个三个月开发周期的项目做了六个月，还没有一个象样的系统出来，用户则不断的埋怨开发力度不够，管理不好等，项目组又认为是用户随意性太大，导致了这种后果。结果是双方互不满意。

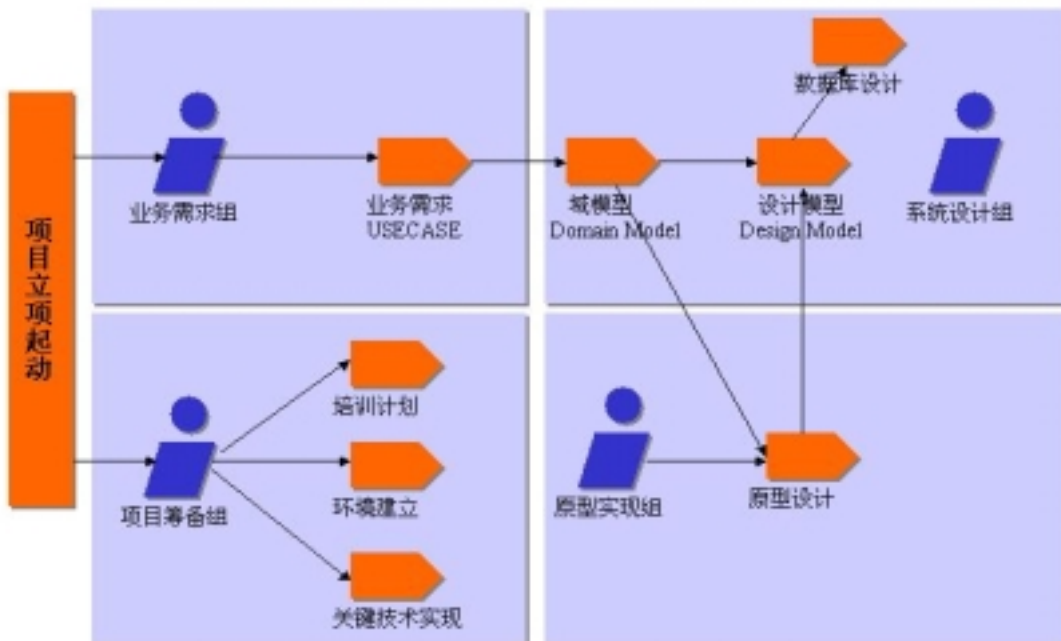
根据这种情况，我认为在业务需求循环到一个阶段，生成正式的业务需求文档与模型，并得到用户的认可后，则进入下一个循环。以至于生成一个版本。这样由于存在这样的版本，用户则不能将项目的延迟归于项目组。而项目组也存在一个完整的版本管理体系。

4.1.2.2 工作流程中的并发性

有个朋友曾经问我在开发项目时什么时候让你感到最浪费时间，我想了想，说是“当我很忙，但项目成员却无所事事的时候”。细想原因，归根到底是工作流程的问题。为什么这样说呢，传统开发时，当项目经理与系统分析员进行业务需求和设计时，通常认为项目成员没有什么可做的，在项目过程中，做一些系统更改时，认为程序员水平不够，帮不上忙。于是乎，程序员似乎理所当然地休息一下，上上网，打打游戏。时间就这样被浪费了。

这个问题就是我们在此规范强调的重点之一：**工作的并发性**。

下图是实现工作并发的一种方式：



在工作中实现并发，是合理地安排项目开发的重要环节，以避免不必要的浪费，同时通过合理分配工作量的方式，减轻了传统项目中项目经理的压力。

4.1.3. 项目管理制度

- 淡化个人的力量，突出团队的协作

在项目开发过程中，最麻烦的就是个别‘高手’的要挟。这种‘高手’掌握了系统关键的部份，并且此时无人可替，非他不可，这时‘高手’借机要求加薪，升职。。。。。

如何避免这种情况的出现呢？

我认为是细化工作量，不要象以前的开发中那样，说某某，你负责某个模块。而是尽量细化工作内容，基本上应细化到每工作日，如果细化到工作时则更好。同时建立相互依赖关系，实现开发上的并发，每一个程序员的工作延迟，将牵涉到几个程序员的工作，这样，其他程序员为了在工作日中完成工作，就

必须相互帮助。从而实现了团队的协作。

- 做好项目总结

项目总结是非常必要的。在项目总结中要对项目或开发阶段中出现的问题进行一一归纳。其中包括技术总结，工作总结，行为总结，从而促进项目人员的成长并能在下一个阶段或下一项目中避免相应的问题。

技术总结主要是对开发中所出现的技术问题进行总结。一个程序员开发的程序在被测试员测试后，或被代码复审人员检查后，发现了问题，如废代码过多，调用错误的参数等，此时你不应立即打断程序员的工作，因为那样会打扰程序员现有工作的思路。在项目总结会上，可以把他写的代码公布给项目组人员共同阅览，让大家给他提意见，这样使之有了进一步的提高。

工作总结是指在开发过程中出现的其他问题，有些人能力差，有些人能力强，能力差的总是拖大家的后腿，这样导致众多意见，这样可以通用大家总结的方式，一来可以为项目管理人员重新安排工作量提供参考，二来也在以另一种方式对程序员进行激励。

行为总结是指对项目开发中个人行为所出现的问题进行总结。如个别人出现了消极怠工现象，那样大家来总结一下，如果是外界因素如家庭等的影响，则看项目组成员是否可以帮上忙，或者由项目管理人员进行工作安排的协调。如果是上面说的‘高手’故意摆谱，则把事情讲清楚，由项目管理人员进行相应的处理。

- 流程管理

习惯于传统开发模式的程序员可项目经理可能不愿意按照项目管理规范来做。认为这样做麻烦，要写的文档无数。在实施的过程中往往按自己的一套来做，从而造成项目的拖延。所以说，对项目中的各个流程都有相应的机制来监督，RUP 中所谈到的流程管理是与开发相并行一个机构，但在目前的情况下，可能无法实现。那样就必须建立相应的机制来处理，以避免不必要的损失。

4.2. 分步实施

由于我们现在还处在原始的项目管理阶段，实现一步到位是非常困难的，所以采用

分步方式实施：

第一步：实现初步规范（针对项目组级）

- 实现项目阶段、角色层次的初步划分

将项目阶段划分四个大的阶段（如第一章），以里程碑的各项指标（指标根据实际情况缩减）考核项目组。

将项目组人员划分成不同角色（一个人可能是多个角色），明确分工，加强协作。（角色根据实际情况缩减，基本上是业务人员、架构设计师、系统分析员、项目经理、环境配置人员、数据库设计人员、系统集成人员、程序员、测试员、复审员、文档员、界面设计人员）

- 项目工作重心转到分析设计部分

将工作重心转移到分析设计来，分析设计分为两个方面，一方面是对系统功能与架构等系统设计，另一方面是指将系统分析细化至功能及类一级，并写出对类的要求，如参数，功能等。

- 规范工作流程

将工作流程划分为业务需求、环境配置、分析设计、项目管理、实施。严格按照项目规范进行管理。

- 更改分析设计方法

采用 OO 设计方式，实现 CMM2 可重复级。

- 实现文档模型规范化

形成各种所需文档模板，采用 RATIONAL ROSE 进行 UML 建模。建模程度至少达到 Logical View 一级。并形成相应的规范，如编码规范、建模规范。

- 建立简单的流程控制机制

主要通过技术秘书，部门经理进行流程控制。

第二步：根据项目管理规范组织一支合理的开发队伍（针对部门级）

- 统一规范化部门的开发队伍

以部门为单位建立统一的开发队伍，将程序员级与测试级分离出来，并考虑采用外包方式进行代码一级实施。培训分析设计人员，建立强有力的分析力量，并细化工作分工，不再以单纯的项目组的形式来进行组织。

- 细化工作流程

细化工作流程的各个环节，减少一个人所承担的角色数量。加强相互间合作。

- 增加细化工作模型与文档模板

细化规范各部门的文档要求，增加细节文档，并提供文档模板与模型要求。

- 建立相应的产品化机制

增加产品化工作流程，由产品化工程师对相应行业系统形成产品，实现 CMM3 可定义级。

- 定量管理和控制软件过程和产品

收集详细的软件过程和产品质量的特征

- 建立相应的流程管理机制

由流程工程师控制软件流程

第三步：实施统一的软件工程（企业级）

建立软件工厂，达到 CMM5 级（优化级）

umlchina 提供 UML/OOAD 培训

通过讲述一个真实 N-Tier 系统的开发过程，使学员自然领会 OO 技术。概念并不按部就班讲授，只是由实例带出。

详情请联系 think@umlchina.com